PROBABILISTIC ROADMAPS IN UNCERTAIN ENVIRONMENTS

by

Michael Lyndon Kneebone

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

Intelligent Robotics Laboratory
School of Computer Science
The University of Birmingham
June 2010

Abstract

Planning under uncertainty is a common requirement of robot navigation. Probabilistic roadmaps are an efficient method for generating motion graphs through the robot's configuration space, but do not inherently represent any uncertainty in the environment. In this thesis, the physical domain is abstracted into a graph search problem where the states of some edges are unknown. This is modelled as a decision-theoretic planning problem described through a partially observable Markov Decision Process (POMDP). It is shown that the optimal policy can depend on accounting for the value of information from observations. The model scalability and the graph size that can be handled is then extended by conversion to a belief state Markov Decision Process. Approximations to both the model and the planning algorithm are demonstrated that further extend the scalability of the techniques for static graphs. Experiments conducted verify the viability of these approximations by producing near-optimal plans in greatly reduced time compared to recent POMDP solvers. Belief state approximation in the planner reduces planning time significantly while producing plans of equal quality to those without this approximation. This is shown to be superior to other techniques such as heuristic weighting which is not found to give any significant benefit to the planner.

Dedication

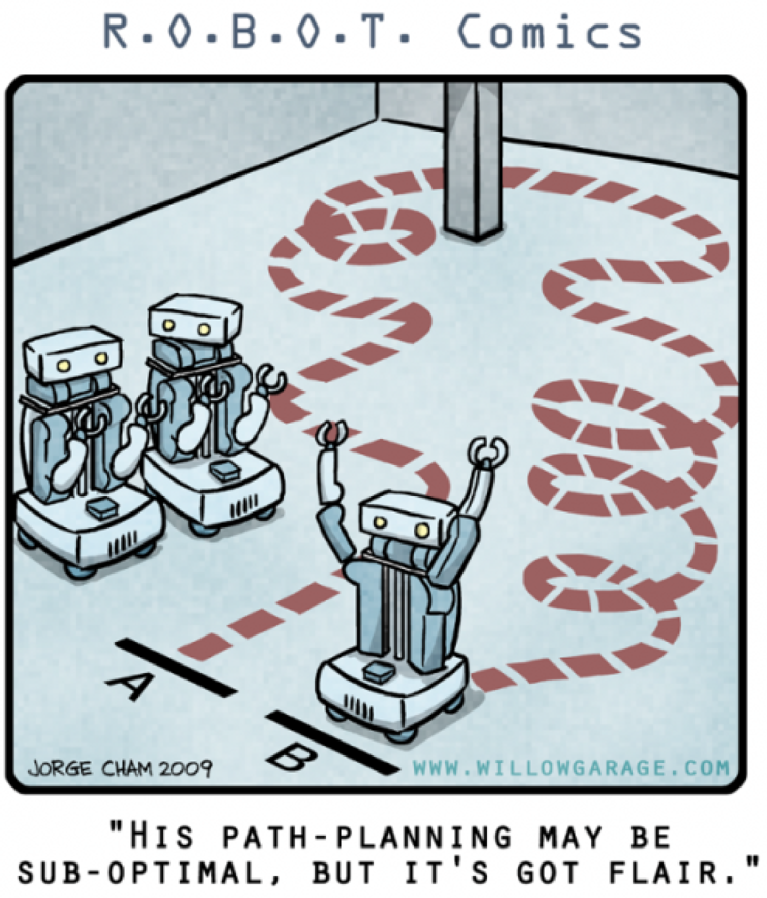
To everyone who encouraged me.

Acknowledgements

I have been helped by so many people during the progress of this research that I can't do them all justice here. My first and biggest thanks must go my supervisor, Dr. Richard Dearden whose constant and invaluable guidance has kept pushing this work towards its final goal. It goes without saying that I would not have been able to achieve this without him. I especially appreciate his continued patience with my total lack of mathematical ability. I still consider counting to ten to be a significant numerical challenge. I also thank my thesis group members, Dr. Jeremy Wyatt and Prof. John Barnden for their support throughout my studies as well as my examiners, Dr. Jesse Hoey and Dr. Manfred Kerber for their time taken reading and providing corrections and improvements to my work. My special gratitude must also go to my family who have loved and supported (and on occasion paid for) me throughout my education, particularly my mother for proof-reading this entire thesis and my father for keeping me interested in anything/everything technical in the first place.

I am also greatly in debt to my varied and eclectic, but continuously fantastic group of friends who have entertained me and reassured me through my studies. As several of them are now studying for/writing up their own theses I know that this at least is a debt that will be repaid. Unlike my student loan. Particular thanks must go to Nathan for many interesting and varied geeky conversations over the years, Becky for listening to me whine about nothing and everything all at once, Mark for repeatedly putting things into perspective for me, Ben for showing me how to stay determined, Kat for organising many outings to places when I couldn't be bothered, Simon, Dave S, Jon and Henry for their continued friendship after those awesome undergraduate years, and last but not least, Dave B for so many humorous comments and hilarious conversations that I can't even remember them, as well as his encyclopædic, borderline savant knowledge of *The Simpsons* and *Futurama* quotes. It would be a great oversight at this point for me not to acknowledge my ever changing group of office mates over the years for their advice, (sometimes dubious) wisdom and insight into the intricacies of Ph.D. life and procrastination techniques. To everyone else who assisted me in someway but I have forgotten to mention here, I am also grateful.

And as it turns out in fact, $11 \neq 15$.
Q.E.D.


R.O.B.O.T. Comics
A
B
JORGE CHAM 2009
WWW.WILLOWGARAGE.COM
"HIS PATH-PLANNING MAY BE SUB-OPTIMAL, BUT IT'S GOT FLAIR."

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Uncertain Navigation

Robot navigation has been important in artificial intelligence research for many years. The rise in the use of robots in domestic, industrial and military settings emphasises this. Although commonplace in certain industrial applications, the domestic use of robots has mainly been limited to toys and low-level household chores such as the Sony® AIBO®[1] robotic dog or the iRobot Roomba®.[2] In a larger class of applications and machines, researchers from various institutions have built robotic cars capable of autonomous driving with some degree of success for competitions such as the DARPA Grand Challenge.[3] All of these implementations must include a navigation facility in order to find their way around their environment.

1.1.1 Motivation

In this thesis we tackle one aspect of robot navigation without focus to a specific application. The problem of uncertainty exists in nearly all navigation and route-finding domains. Navigation—the task of moving the robot from one location to another—is inherently a planning problem since the robot, or *agent*, must decide how it is going to accomplish the task by repeatedly choosing what its next move should be from a set of possible options. In such a problem, multiple sources of uncertainty may be present, such as an agent's uncertainty over its current location, uncertainty about the future state of the environment, or uncertainty in the information it collects from sensing its surroundings. We use the term "environment" to refer to the setting the robot is in, i.e. its surroundings and obstacles that it may interact with. Any single factor, or combination thereof, makes

[1]Sony® AIBO® Europe: `http://support.sony-europe.com/aibo/`.
[2]iRobot Roomba®: `http://www.iroboteurope.co.uk/`.
[3]DARPA Grand Challenge: `http://www.darpa.mil/grandchallenge/`.

navigation planning a hard problem, one which is still not completely solved.

In this research we focus on the problem of sensor uncertainty: the fact that the information a robot receives about its surroundings may not be (and probably is not) completely accurate. The problem is harder than might initially be assumed because any uncertainty present in a situation makes planning significantly harder. We can illustrate this by considering the reverse scenario: imagine the task of planning a typical day; there are usually several jobs that need completing with many constraints on when and in what order they must happen. Particularly when other people are needed to complete a task (e.g. attending a meeting), certain aspects of the day's schedule must be deliberately flexible in order to cope with unforeseen circumstances. Now consider the impact if the times, duration and location of everything were already known beforehand. Every single detail could be planned as precisely as desired under the assumption that nothing would change and would never need updating. This would make devising the initial plan easier as no backup plans would ever be necessary. In robot planning the uncertainty presents similar problems: many possible plans or alternative sub-plans must be processed to find an overall plan that works. To simplify the problem, we make specific assumptions about the world the robot lives in, which are discussed later. However, we also make certain assumptions about the robot to enable some parts of the problem to be abstracted away. In particular, we assume the robot has access to sensors so it can gain information about the local part of its environment. These sensors are allowed to be inaccurate so the robot does not get perfect information from them; however we assume that the robot knows how accurate its sensors are. This is a realistic assumption, since information is easily available from training results and from simulation for most sensors which would quantify the signal-to-noise ratio for a particular sensor type.

We assume we have a small mobile robot that can freely move around its environment and contains onboard resources to operate autonomously, e.g. a power source, the necessary computing hardware and some (unspecified) sensors that allow it to make observations about its environment. The robot's task is to navigate from a given starting position (the start location) to a final position (the goal location) through the environment which may contain other objects which must be avoided, without it being given complete information about their shape or location. We do not deal with the problem of moving obstacles, such as other robots. We want the robot to devise a plan which yields the lowest expected cost in order to reach the goal, where cost is defined as the distance the robot has to travel.

When complete and perfect knowledge of the world is available to the robot, planning the best path through the obstacles becomes a much simpler problem, assuming the robot always knows its own location. Many planning algorithms can plot the shortest route through a known map with perhaps Dijkstra (1959) being the best known example. Examples of this type of planning can be seen in many modern applications such as video games where routes are required. When the world knowledge is known to be incomplete and inaccurate, the agent requires much more complex reasoning to deduce a solution to the problem. In the former case, planning can be conducted entirely *offline*, before the agent starts using the plan, with the agent simply carrying out each step in the plan (or route) sequentially to reach the goal. When uncertainty is present, then, although the complete plan may be generated offline (as is the case for the problems we present), the agent still needs extra reasoning ability to use the plan (plan execution) in order to pick its next action based on how its knowledge of the world has changed. This leads to the fundamental question for the research that the robot must answer given the above scenario:

Given what I know about the environment, which action from the set of possible choices should I choose next?

Essentially, we want the agent to be able to work out how it should act in the various situations it might find itself in, e.g. what should it do if it finds it cannot take the path it originally wanted? The research contained in this thesis is focused on formulating environments and navigation tasks in such a way that the agent can answer the above question.

1.1.2 Example scenario

The typical class of problem tackled in this research can be illustrated with the following example. The situation in Figure 1.1 shows a typical scenario. The robot at S has the choice of taking the upper shorter route, which takes it through the narrow gap between the two obstacles, or the lower route which is safer but longer. Since we ultimately want the robot to incur the cheapest cost, i.e. travel the shortest distance to the goal, the choice seems obvious: pick the shortest route. However, when the robot is uncertain if it can fit through the gap in the shorter route, a choice must be based on its confidence of success. It will not know for certain if the gap is wide enough until it is closer to it. If the probability of squeezing through is too low, then the robot should decide it is not worth the risk and choose the safe route. Conversely, a high chance of success means the risk is likely justified, so the upper route should be chosen. In reality, the robot either can or cannot fit, but it does not know which is the true case. Making the wrong choice in either situation will incur a penalty: be too pessimistic and it could travel farther than necessary, but being over-optimistic means it may pay the heavier cost of having to retrace its path out of the corridor and around the larger object. It must sensibly trade-off the

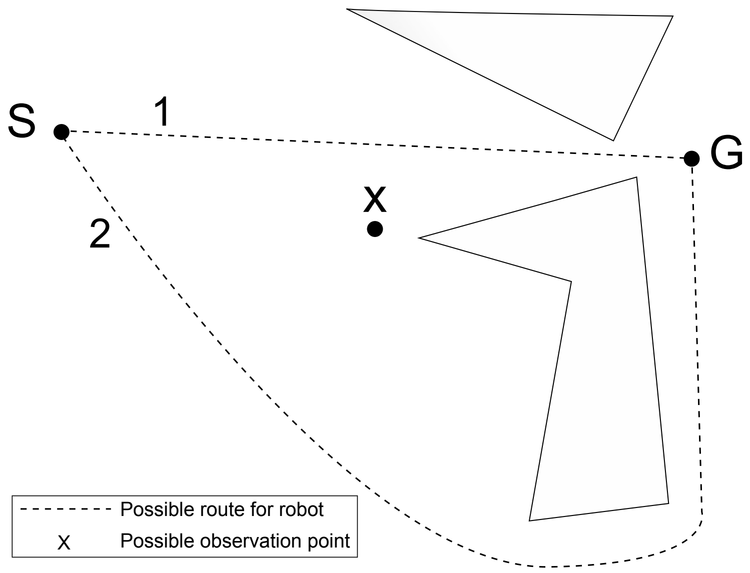

S
1
G
X
2
-------- Possible route for robot
X Possible observation point

Figure 1.1: A simple path planning problem. The start and goal locations are marked 'S' and 'G' respectively. The robot may initially choose either of the two possible routes shown, ignoring the observation point.

risk of taking a short, potentially unusable route against the guaranteed extra cost of a safer route.

What if a third choice was available? In the above discussion, we assumed the uncertainty remained until it reached the gap. More realistically, the robot could make a reasonable observation from some distance away. Now assume the robot may make such an observation at point X in the figure. In addition to the two previous choices, the robot may now choose to travel to X, make an observation of the gap and then decide what to do next. In many scenarios, this will be the better choice. Travelling to X then choosing to use the shorter route may be longer than using the shortest route in the first place. However, it is significantly shorter than going into the narrow gap, finding it cannot fit, then having to retrace its route and travel the long way around.

1.2 Probabilistic Planning

The planning methods we consider are all based on probabilistic algorithms. This allows for the creation of plans more robust to inaccuracies in knowledge than classical non-probabilistic planners. Uncertainty features prominently in most robot navigation and perception problems, so accounting for this in the planning stage makes sense. Using probabilistic techniques, the agent may consider many different possibilities, each of which may be the true case, and plan what to do based upon its confidence (the probability) about which is correct.

With more traditional planning techniques, which may be limited to reasoning about one possibility, the overall result can be of lower quality. This results from having to choose the single possibility the agent thinks is most likely to be correct and then planning for that case alone. For instance, in measuring the width of a doorway, a robot may sample several measurements from the same sensor to account for noise. Conventional planners may take some scalar value from a sample set X, e.g. the population mean $\bar{x}$, or the modal value of the set as the true width of the door and plan under that assumption. A more flexible way is to model the width with a probability density function giving a distribution over possible values, such as with a Gaussian distribution with $N = (\mu = \bar{x}, \sigma^2 = \mathrm{Var}(X))$. The planner can then plan knowing that the doorway is not necessarily $\bar{x}$ wide.

Probabilistic planners provide graceful degradation of performance as information quality deteriorates. These types of planner are also more scalable to complex environments than conventional planners; Probabilistic Roadmaps introduced in the next Chapter are an example of this. This feature mainly arises from the way they explore the space they plan in (e.g. the action space or robot pose space). Planners usually search deterministically according to some algorithm to systematically explore the space. As the complexity of the search space increases, this can drastically increase the time required to evaluate exponentially more possibilities. Probabilistic planners generally explore the space stochastically according to a probability distribution indicating where "good" sections of the space lie. The definition of good is task dependent.

With uncertainty in the environment, the lack of perfect information should also be incorporated into decisions about how the agent should act. While not having access to the complete picture of the world complicates answering the question posed in the previous Section, it is still possible to make informed decisions. A Partially Observable Markov Decision Process is a planning technique which computes the best plan for how an agent should act in an environment with imperfect, incomplete information. Bayesian probability is used to update the agent's confidence regarding the possible states of the world based on new information it receives. Actions are then selected according to its beliefs about the state of the world.

1.2.1 Disadvantages

The use of such methods should by no means be considered a panacea for robotic planning however, as there are drawbacks. Like all planners, probabilistic planners work with models of the real world, containing either approximations of the environment, the robot, or both. Reasoning about the outcomes of actions or states of the world incurs an additional computation penalty if they are non-deterministic. Consider a probabilistic planner that is searching for the next action for its current state from a selection of possible actions.

It must ask itself "What will happen to the state of the world if I apply this action?" for each action and then choose the action leading to the most preferable state. Discrete probabilistic planners maintain a set of states, each with an associated likelihood, as their current state, and therefore have to consider the effect of each action on each discrete state. The choice of next action thus depends on which action is preferable for each state, weighted by the planner's belief that that is the actual world state. The probabilistic planner therefore has increased memory and computation requirements.

That in itself may be sufficient justification not to use one. In many real-time systems, hard time constraints exist that stipulate a maximum response time. Under such circumstances, a probabilistic planner may simply take too long in deciding what to do. Reactive planners are often a good choice in such cases as they can provide a rapid decision to a situation. Reactive planners do little to no computation, providing their answer based on a set of rules in the form of "IF A and B are true THEN use action C"[1] as well as small, supplemental calculations. A hybrid approach can often be used to get the best of both worlds, combining a reactive planner with a deliberative one (Muscettola et al. 2002). Given a situation, a scheduler may choose to run either a reactive planner if a quick answer is vital, or invoke a deliberative planner when sufficient time is available or a higher quality plan is required. Another possibility is to default to using the reactive planner, and only invoke the deliberative planner if the reactive planner fails to produce a good quality plan. In doing so, the planning time is minimised at the expense of plan quality.

1.2.2 Basic approach

As we are concerned with the problem of decision making under uncertainty, the models of the environment and the robot are kept deliberately simple. This differs from other work where the emphasis is on planning the physical path the robot will follow and ensuring that it will not crash. The aim of this work is to investigate how paths to the goal should be selected and how the choice of where to go next is selected. When many choices are available, the usability of some routes will be unknown, so the agent's planner must decide which route to take based on its belief about the state of the environment and any new information it is likely to obtain.

We use a 2D representation of the world and objects within it. The environments in this thesis are modelled through geometric descriptions of the robot and obstacles in it. These take the form of polygonal shapes. These do not need to be closed: an L-shaped robot consisting of two line segments is perfectly valid. The shape of the robot is assumed to be known exactly and we assume that the robot has accurate motor control.

[1]For this reason they are sometimes referred to as "reflexive" planners in the literature.

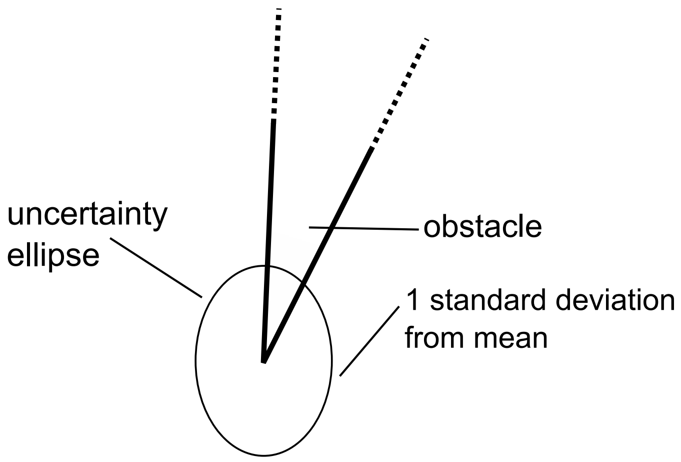


Figure 1.2: The "uncertainty ellipse" visualises the covariance matrix of the Gaussian distribution that specifies the location of the obstacle vertex. The vertex is placed at the mean of the bi-variate Gaussian.

The general shape of an obstacle is assumed to be known and is described in the same way as the shape of the robot. The uncertainty in the environment is present in the obstacles' location in the environment. Obstacles are modelled as in Missiuro and Roy (2006). Each vertex of an obstacle is represented by a bi-variate Gaussian distribution with the mean of the Gaussian located at the most likely position of the vertex. The covariance matrix characterises the uncertainty over the true location of that vertex. Visually this is represented as an "uncertainty ellipse" centred around the mean (see Figure 1.2) with greater covariances producing larger ellipses. Thus, an obstacle that is actually perfectly square, may not appear so to an agent due to its uncertainty concerning the vertex locations. On all diagrams, ellipses are placed at a distance of one standard deviation from the mean. As observations are made, the Gaussian distributions are tightened, which can be visualised as the ellipses reducing in size.

Conceptually, we abstract away the problem of how the observations are actually gathered, as this will often be problem and platform dependent. The methods used in the thesis are not intended to be specialised towards one type of robot architecture. In reality, the observations may come from a variety of sensory inputs such as digital cameras, laser range-finders and infra-red receivers.

For generating paths through the environment, we base our work off a path-planning technique known as Probabilistic Roadmaps (PRM). The nature and details of this algorithm are described fully in Chapter 2, but an introductory description is useful here. PRM uses knowledge of the robot geometry and obstacles in the environment to generate a random network of points (a weighted graph or *roadmap*) across the environment and then connects local points together according to some criteria. Collision detection routines are heavily used in PRM to ensure that the points generated are collision free and to ensure that only collision free edges between points are created. This is carried out under the assumption that the world is static and there is no uncertainty. Obviously, this last limitation does not correspond well with the problems described so far, so the PRM generated graphs form the basis that other planning methods are built on. As obstacle vertex locations are not known with certainty, those that reside close to edges in the roadmap may influence whether those edges are actually collision free or not. There will be some subset of edges in the roadmap that cannot be guaranteed to be collision free.

Definition 1.1 (Uncertain Edge)**.** An uncertain graph edge is one which may be blocked due to an obstacle.

We look at the problem from the perspective of working out which uncertain edges in the graph are actually blocked and which ones are not, as this dictates which edges are usable and what the shortest *valid* route to the goal is. As the robot makes observations about obstacles, it gains knowledge about which uncertain edges are blocked and which are not. This is represented as a probability pertaining to the agent's belief about an uncertain edge. These are always stated as the robot's confidence that the edges are free where:

$$P(A \rightarrow B = free) = 1$$

states that the robot knows with certainty that the edge from point A to B is open and conversely:

$$P(A \rightarrow B = free) = 0$$

states that it is certainly blocked. We model the uncertain PRM graph as a POMDP which is used to decide how the agent should act. Essentially, we use PRMs to explore the environment and create a roadmap of possible routes, and then use a POMDP to decide which route to take.

1.3 Thesis Structure

1.3.1 Required knowledge

A basic knowledge of probability and planning is assumed, but all algorithms discussed are fully explained. Detailed knowledge of route-planning or navigation algorithms is not a prerequisite. The necessary background in Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs) which are central to this thesis is provided in the next Chapter. A background in graph theory is not assumed and all required notation and principles are introduced as necessary. Though by no means the first literature on the subject, *Probabilistic Robotics* by Thrun et al. (2005) provides an excellent insight into many aspects of the field and also contains the necessary introduction to probability theory and statistics. It is not assumed that the reader has read this book, though a basic familiarity with the principles outlined in the first section is advantageous.

1.3.2 Thesis aims

There are two principal aims of this thesis as stated below:

1. Show how path planning in a partially known configuration space can be represented as a decision-theoretic planning problem.

2. Investigate how optimal plans can be generated for the domain in a way that is scalable to real world problems.

The primary aim of the thesis is to show how the uncertain planning problem described above can be solved using a decision-theoretic model of the problem. By modelling the problem in the context of the POMDP framework, the optimal plan can be computed with standard POMDP solution algorithms.

The second aim will be met by applying different techniques to make problem descriptions more compact, that is, looking for ways to simplify the plan model or process. This will extend the range of problems that can be solved. The aim here is not to trade off plan quality for the size of problem we can handle, but to find ways to make the process more efficient. The behaviour of the agent should still be as close to optimal as possible.

1.3.3 Thesis contributions

This thesis investigates the use of decision-theoretic planning models to compute optimal path plans for robot navigation in PRMs with uncertain edges resulting from uncertain obstacle locations and noisy sensor data. We make several key contributions summarised below:

Importance of information When the robot is trying to find the lowest expected cost path to the goal, we show that if it does not account for the value of information gained along the route, the chosen route is unlikely to be optimal. Incorporating the value of information enables the true lowest expected cost path to be found. We also show that computing policies without considering the future uncertainty leads to sub-optimal behaviour.

POMDP and MDP representation When the robot's movement is represented as a static PRM motion graph, we show how the navigation problem can be translated into a POMDP. Since these are complex to solve, we show how that can be converted into an MDP that exploits extra knowledge we have available, such as the robot's location in the environment. This conversion makes the problem easier to solve and more realistic to use on PRM graphs with uncertain edges by reducing the complexity of the associated probability distributions.

Edge clustering More structure present in the domain enables a novel technique to reduce the complexity of the plan space. By reasoning about a lower number of possible world states, the planner is able to work with more complex problems. We show that under certain conditions, this does not reduce plan quality and reduces the required planning time in many situations.

Approximate LAO* We show how the introduction of an approximation technique to the planning algorithm enables plans to be computed far more quickly by exploring a smaller section of the state space of the planner. The reduction in plan time compared to the exact solver is highly correlated to the reduction in the amount of state space explored.

1.3.4 Publications resulting from this research

There are two publications resulting directly from the research in this thesis:

- M. Kneebone and R. Dearden. Navigation Planning in Probabilistic Roadmaps with Uncertainty. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 209–216, Thessaloniki, Greece, September 2009.

- R. Dearden and M. Kneebone. Uncertain Probabilistic Roadmaps with Observations. In *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*, pages 33–40, Edinburgh UK, December 2008.

1.3.5 Thesis layout

In the following chapters we introduce the required background knowledge necessary to understand the thesis, investigate the problems laid out above and show the experimental work conducted as part of the research. In Chapter 2 the background related to this research is explained in detail. The PRM framework, MDPs and POMDPs are covered separately and key algorithms are introduced for each. In Chapter 3 we review the previous work relevant to this thesis from the literature and examine how previous techniques approach the problem. Chapter 4 describes how our navigation domain can be represented as a POMDP formulation and how this can be converted into one of several MDP models. Experimental work presented shows that optimal plans are generated by the POMDP formulation in small problems, the effectiveness of the MDP models and also points to areas for improvement. Chapter 5 examines additional techniques to improve the scalability of the MDP planning models by incorporating approximation into the planner or adding heuristic weighting to the search component, and presents more experimental work to assess their impact. We review the work and indicate possible future work directions in Chapter 6.

Chapter 2

Background

2.1 Probabilistic Roadmaps

Probabilistic Roadmaps are a random sampling path planning technique. They solve the problem of finding a route for a robot between two given points in an environment that can contain arbitrary numbers of obstacles that must be avoided. The robot geometry (its shape and size) are taken into account in the route planning so the route returned will be collision free. For instance, the route will not include narrow passage ways that are narrower than the width of the robot. This is useful to many areas of robotics; in this Section, we describe the algorithm and its relevance. PRM was first introduced by Kavraki, Švestka, Latombe, and Overmars in 1996, building upon considerable earlier research on random path planning and high dimensional robotics. The idea was to develop a new approach to robot motion planning that would not suffer from the same constraints on robot design complexity that contemporary complete space planners did. Limitations on robot complexity hindered the use of such planners, making planning for complex scenarios require oversimplification of the domain, which reduced the usefulness of the plans. Here we describe the algorithm and its internal components as well as some extensions developed since its inception.

2.1.1 Configuration space

The configuration space (C-space) for a robot is the space of all possible poses (*configurations*) that the robot may assume. The configuration space has the same dimensionality as the total number of dimensions needed to completely describe the robot's pose. Each point in the C-space represents an exact position for the robot. Most C-space dimensions are continuous since the variable may take any value within a set interval, e.g. the arm angle in Figure 2.1. While in practice, the controls may have a specific resolution,

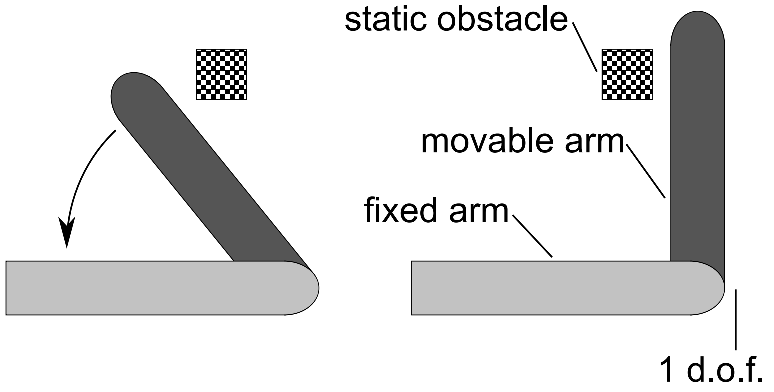


Figure 2.1: The darker arm is mounted behind the fixed arm with full 360° movement. These two poses are close together in the one-dimensional C-space for the arm. In a PRM graph two nodes representing these poses would also be considered close. However, the obstacle blocks the short path—the arm would need to rotate almost a full circle to switch poses, thus the poses are not close in the action space.

thus giving the parameter range a finite size, considering each value for each control as a separate action still results in an intractable search space.

Consider the arm in Figure 2.1; the C-space can then be divided (Kavraki et al. 1996; Kavraki and Latombe 1998) into two sub-spaces: configurations with conflicts and those without, C_{obst} and C_{free} respectively. In Figure 2.1 the region blocked by the static obstacle would be part of C_{obst}, with the remaining space residing in C_{free}. It is important that only poses from C_{free} are included in any plan; therefore collision checking the points sampled is essential to ensure they are valid. In the original PRM algorithm, a large proportion of PRM running time, often around 99% (Sánchez and Latombe 2002a), is spent doing these checks.

2.1.2 Random path planning

Classic graph traversal algorithms such Dijkstra's algorithm (Dijkstra 1959) or A* (Hart et al. 1968) are employed to plot the optimal route over a space in simple planners. This is usually accomplished according to some cost metric, e.g. time or distance. Such a method relies upon being able to explore the search space in its entirety (or sufficient detail) to compute the best path. In low dimensional domains this is a feasible solution, however in high dimensional worlds like high degree of freedom robotics the problem becomes intractable. When the number of dimensions increases, the size and cost of exploring the search space grows rapidly due to the complexity.

Definition 2.1 (Degrees of Freedom)**.** Each controllable variable in a robot's pose constitutes one degree of freedom (d.o.f.) in the robot's movement.

This is sometimes known as the *controllable* degrees of freedom in robotics literature.

A robot has a certain number of joints and actuators it may control, such as the angle of a joint or the rotation of a bearing, each of which is a separate degree of freedom. For example, a simple car with Ackerman steering can control the steering angle of its two front wheels and its forward speed, giving it two degrees of freedom, whereas a human elbow (ignoring the wrist) has only one degree of freedom: the angle at the elbow joint.

Definition 2.2 (Dimensionality)**.** The dimensionality (or total d.o.f.) of a space determines how many dimensions are required to uniquely specify one point in the space.

A single point in 2D Cartesian space requires only an x and y co-ordinate pair to identify it. The car above intuitively exists in 2D space (the flat plane it drives on), but its space is actually 3 dimensional: the third co-ordinate being θ, the orientation of the car. We refer to a robot's controllable d.o.f. simply as its d.o.f. and use 'dimensionality' to refer to the total d.o.f. of the space. The car controller cannot control its position in all three dimensions independently so reaching a particular location and orientation generally requires a sequence of actions to achieve. This limitation implies the car is nonholonomic.

Definition 2.3 (Nonholonomic)**.** A nonholonomic robot has fewer controllable d.o.f than the total d.o.f (dimensionality) of the space that it exists in.

Classic planners begin to fail in high d.o.f. scenarios because they cannot produce a solution in any reasonable amount of time. The core difference in PRM is that it does not explore every point in the search space. PRM planners create the roadmap probabilistically. This is reflected in the paths it generates. The explosion in search space is overcome by exploring the space in a random fashion. Robot poses are sampled from the entire configuration space according to a *sampling strategy*. The simplest sampling strategy is a uniform random strategy that selects values for each dimension from a uniform distribution. Alternative strategies are discussed later. All points reachable by a complete planner are still reachable with PRM since it may generate a sample in any location. Sampling removes the direct link between the dimensionality of the robot and the efficiency of the algorithm.[1] By sampling from the C-space PRM planners are able to abstract away the control actions needed to reach a configuration. A drawback is that poses which appear close to each other in terms of configuration may not be close when considering the action sequence required to move between them. The arm in Figure 2.1 illustrates this problem.

[1]It should be noted that an implicit link between the running time of PRM and the complexity of the domain still exists, but this is related to the number of points sampled and the validity of checking those points, not the search costs of the domain.

2.1.3 Basic algorithm

Given a complete map of the environment and model of the robot, the algorithm takes as inputs the desired start and finish poses for the robot and returns either a collision free route, or declares no such route exists. PRM works by generating a path through a robot's C_{free} space between the given poses. A graph of points in the C-space is created and then a route traversing it is built. The construction and use of this graph form two separate components in the algorithm: the *pre-processing* phase and the *query* phase.

Algorithm 2.1 shows the pre-processing phase, where the roadmap is created. This roadmap is re-used for each query, though it can be built incrementally (Kavraki and Latombe 1998, Section 4.3) where the roadmap is extended with more nodes and edges as queries are processed. Lines 3 to 8 generate the set of nodes V by repeatedly invoking the sampler (the `generateSample` function, line 4) and collision testing the sample by calling the `isInCFree` function. These are domain dependent and are deliberately left unspecified. In basic PRM, `generateSample` samples nodes uniformly at random. Edges for a node v are added (lines 9 to 17) by first finding a candidate set of local graph nodes that are within a maximum distance D_{max} of v. Distance in the basic version of PRM is defined as the maximum Euclidean distance between all the points in the two different poses being measured (the `dist` call). A local planner (Kavraki and Latombe 1998) is used to check for a collision free path between two nodes by connecting the two poses with a straight line through the C-space and collision checking intermediate points on that line (the `freePath` function, line 13). Nodes are tested in order of increasing distance from v and edges are added where no collision will occur. The pre-processing phase of PRM returns as output an undirected graph $G_{\text{PRM}} = \langle V, E\rangle$ where set V contains the sampled nodes and set $E = \langle v_1, v_2\rangle$ contains the connecting edges.

The query phase is where the planner creates routes. The first stage is to connect the start and finish poses to the graph. This is accomplished using the local planner (the `freePath` function) which connects the poses to the closest collision free nodes available. The nodes on the graph are tried in order of increasing distance from the start or finish poses until a suitable one is found. When the start and finish poses are connected to two graph nodes, a standard graph search algorithm (commonly A*) is used to find the shortest route between the nodes. The three parts of the route (start pose to a graph node, path across graph and graph to finish pose) are concatenated together to form the complete route.

PRM created paths may require a post-processing phase due to the randomness of the roadmap. The returned path will be the shortest route across the graph, but poor node coverage can still result in inefficient traversal of open spaces. A common approach is to use a smoothing routine to look for shortcuts between nodes not permitted by the

Algorithm 2.1 Standard PRM pre-processing phase, (Kavraki and Latombe 1998; Hsu et al. 2006)

```
Input: C-space complete description of the configuration space, R robot geometry, $D_{max}$
    maximum distance between two connected nodes, $e$ maximum number of edges per
    node, $N$ desired size of $V$
Output: $G_{\text{PRM}} = \langle V, E \rangle$ where $V$ is the set of nodes and $E$ is the set of edges
    $\langle v_1, v_2 \rangle \mid v_1 \neq v_2$
1: $V \Leftarrow \emptyset$
2: $E \Leftarrow \emptyset$
3: repeat //generate $N$ roadmap nodes
4:   $q$ =generateSample($C$-space) //call sampler to generate roadmap point
5:   if isInCFree($q$,$R$,$C$-space) then
6:     $V \Leftarrow V \cup q$
7:   end if
8: until $|V| = N$
9: for all $v \in V$ do //generate edges between edges
10:   $C \Leftarrow$ sort($v' \in V \mid v \neq v'$) //sort v' into ascending order according to dist
11:   while edges in $v < e$ AND $C \neq \emptyset$ do //$v$ might already have $e$ edges
12:     $y \Leftarrow$ removeHead($C$)
13:     if freePath($v, y$) AND dist($v, y$) $\leq D_{max}$ then
14:       $E \Leftarrow E \cup \langle v, y \rangle$
15:     end if
16:   end while
17: end for
18: return $G_{\text{PRM}} = \langle V, E \rangle$
```

roadmap, e.g. to cut a corner from a triangle. This results in simpler paths with fewer nodes. One smoothing strategy (Kavraki and Latombe 1998) is to select pairs of nodes at random from the route and use the local planner to test for simple paths between them, replacing the relevant route section if possible. A second strategy that is more suited to low d.o.f scenarios is to exhaustively test for shortcuts between all node pairs starting from the beginning of the route. The farthest route node reachable from each node is stored and then a smoothed route created by attempting to reach the goal node with as few nodes as possible. Figure 2.2 shows an example of a raw PRM route with the equivalent smoothed route using the second strategy.

2.1.4 Probabilistic completeness

As stated before, the strength of using PRM is that it can navigate through a high dimensional C-space without mapping it like traditional search space techniques. The drawback to the approach is that it is no longer complete, meaning that if a path between two robot configurations exists, it is not guaranteed to be found. Further, if a path

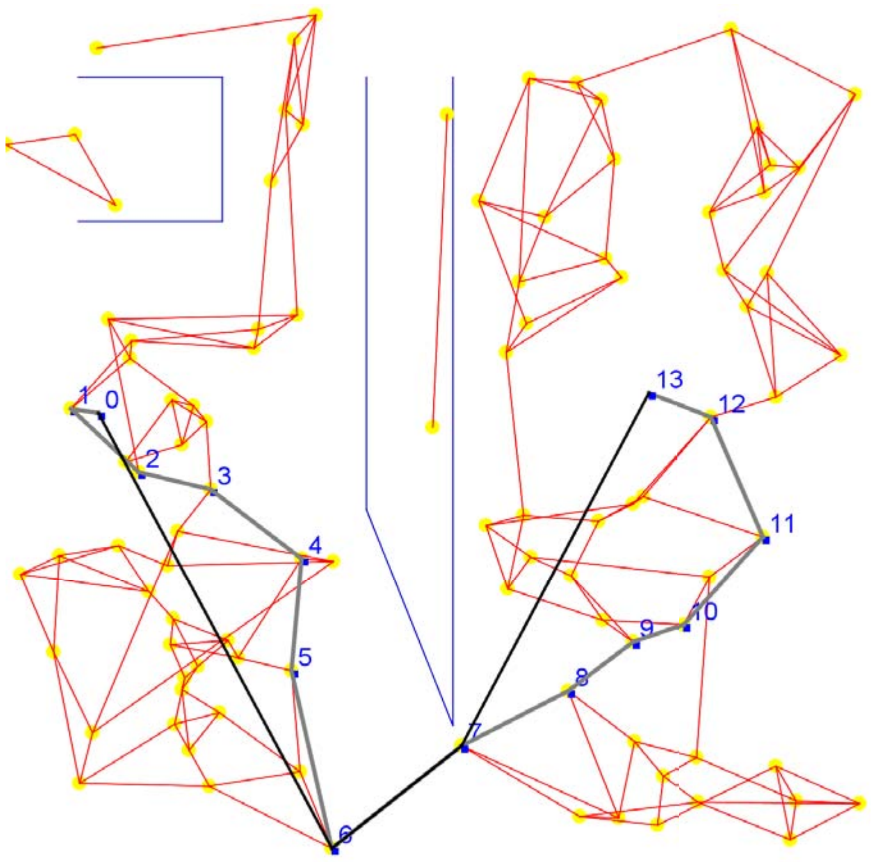


Figure 2.2: An example of a PRM graph with the route generated in the query phase shown by the grey line and the more efficient smoothed route shown in black. Using the smoothed route, the agent would travel directly between the nodes on the black line, e.g. it would move directly from node 0 to 6. The blue lines represent obstacles in the environment.

is found, then it is not guaranteed to be optimal. PRM possesses a weaker property: probabilistic completeness (Sánchez and Latombe 2002b; Hsu et al. 2000; Choset et al. 2005; González-Baños et al. 2006). Given that the planner runs in bounded time, if a solution is not found within that time it returns failure. It can be shown (Barraquand and Latombe 1991) that the probability of PRM returning a collision free path tends to one as the number of graph nodes increases. Conversely, the probability of PRM incorrectly stating that no path exists tends to zero.

Hsu et al. (2006) showed that the running time needed for PRM to create a graph is proportional to the dimensionality of the space and the number of samples generated. Therefore assuming suitable time is available to generate a graph for a particular space, probabilistic completeness implies that PRM will find a path or correctly return failure with a probability approaching one. Experimental results in Kavraki and Latombe (1998) show that in environments where the C-space is well connected and expansive (i.e. does not consist mainly of narrow spaces), then PRM will find the necessary paths 100% of the time without needing excessive amounts of time. A more detailed discussion of the relationship between the expansiveness of the C-space and the runtime performance can be found in Hsu et al. (2006).

2.1.5 Limitations

In the case of the basic PRM algorithm, there are many limitations that researchers have tried to improve upon since its conception. The original PRM algorithm was designed to operate in a static environment with no uncertainties or moving obstacles. This implies the planner has total knowledge of the environment including the agent it is creating a plan for and any (stationary) obstacles present. It also assumes that no external actors can disturb the plan created. The planner lacks the ability to plan for nonholonomic robots, multi-agent worlds, robots with kinodynamic constraints (limits on the robot's kinematics such as maximum acceleration, minimum turning circle) and models where actions are non-reversible.

2.1.6 Extensions

The original PRM algorithm was published in 1996 and many researchers have extended and adapted it. We present a review of a few of these here.

2.1.6.1 Sampling strategies

A very common problem for PRM planners is finding routes through narrow spaces (Hsu et al. 2006; Missiuro and Roy 2006; Sánchez and Latombe 2002a), because if nodes are placed uniformly, then the node density through narrow spaces will be the same as in open spaces, despite the former being harder to navigate through. Different sampling methods that are biased towards sampling near obstacles can achieve better coverage of difficult areas. These are known as *importance sampling strategies*.

Gaussian based sampling (Boor et al. 1999) tries to sample near obstacles only. A first sample, a is chosen by a uniform sampler, and a second sample, b, is then taken uniformly from a circle centred at a. If exactly one of the two is in C_{free} it is kept, otherwise both are discarded. The radius of the circle is itself sampled from a Gaussian distribution based upon knowledge of the obstacle density in the C-space. A sampling strategy that samples exclusively in narrow corridors is Bridge based sampling (Hsu et al. 2003) which is similar to the Gaussian method. It generates pairs of samples and rejects samples that lie in C_{free}, repeatedly sampling points until both a and b are in C_{obst}. It then samples the midpoint between a and b, keeping it if it lies in C_{free}. As noted by Missiuro and Roy

(2006), since this method only keeps points in narrow corridors, it generally needs to be combined with a more general sampling algorithm to adequately capture C_{free}.

Quasi-random strategies that are not entirely stochastic in design include grid based samplers (Bohlin 2001). These create samples based on dividing the C-space into cells and using grid nodes as samples. When used with Lazy PRM (see Section 2.1.6.3) the grid resolution can remain coarse during the construction phase, saving computation time. In the query phase, the resolution can then be varied as the local obstacle density dictates. Finer grid resolutions are used in difficult areas of C_{free} as needed. Connection based samplers detailed in Kavraki (1995), are samplers that are often combined with other strategies to improve coverage in difficult areas of C_{free}. They bias sampling nodes towards areas where disconnected parts of the roadmap exist but C_{free} is not disconnected. A heuristic which estimates the expansiveness of the local region of space (i.e. the difficulty of traversing that region) is used which assigns a weight to each sampled pose in the roadmap. To expand the roadmap, a random sample is selected with probability determined by the normalised weights and then a new sample is generated in the local region. This should lead to improved coverage since areas where the previous two criteria are met will often be in complex parts of the C-space.

There is strong evidence that non-uniform sampling enhances PRM performance too (Hsu et al. 1997). Using no a priori information about the topology of the C-space, options for sampling strategies are limited. Both Gaussian based sampling and Bridge based sampling require some knowledge of the obstacle density of the environment. If such information is calculated, the sampling method can be improved by exploiting it to make more informed choices about locations for sample points. For instance, sampling can be biased to encourage more samples in difficult areas of the C-space and less in expansive regions.

Experimental results (Hsu et al. 2006, Section 4) show an intelligent sampling measure can give a very significant speed up to PRM. In experiments comparing sampling strategies in narrow corridors, the running time with non-uniform sampling stays roughly constant, whereas with uniform sampling, the time increases dramatically below a certain corridor width.

Many other sampling methods have been studied in the literature. For a broader view of sampling strategies, the reader is referred to Section 4 of Hsu et al. (2006) and Section 7.1.3 of Choset et al. (2005).

2.1.6.2 Multi query and single query planners

The original PRM algorithm was multi query: it generated the roadmap once in the preprocessing phase, then ran tens and often hundreds of queries using the same roadmap;

although the roadmap could be improved and expanded as queries were executed. Single query planners adopt a strategy of creating a new roadmap for each query. This is not as wasteful as might be expected and often gives impressive speed ups (Bohlin and Kavraki 2000) by only exploring small areas of the C-space. Single query planners typically work by “growing” a tree of nodes from the initial pose toward the goal using knowledge of the configuration space to explore as little as possible. Examples include planners detailed in Sánchez and Latombe (2002a); Bohlin and Kavraki (2000) and Kindel et al. (2000). Due to the optimisations single query planners use, multi query planners are often only faster in cases where *“several dozens or even hundreds of queries use this [i.e. the current] roadmap”* (Hsu et al. 2006).

2.1.6.3 Lazy and bi-directional planners

PRM spends much of its time collision checking potential edges; therefore reducing these checks will significantly reduce the running time of PRM. The Lazy PRM approach (Bohlin and Kavraki 2000) can be used in single and multi query implementations of PRM. At roadmap construction, it assumes that edges are collision free, forgoing the normal check. At the query stage, the shortest route is found and only then are path sections tested for usability. When a path section with a collision is found, that edge or node is removed from the roadmap. The route is repaired by searching for a new node from the remaining nodes. Results from experiments show that Lazy PRM carries out less than 0.1% of collision checks compared to the original PRM. In the experiments presented, Lazy PRM computes routes in an average of 0.7% of the time required by standard PRM.

Bi-directional PRM (Sánchez and Latombe 2003) is a slightly adapted version of single query planners. In a typical single query planner, a tree of edges and nodes is grown from the initial pose. Nodes are iteratively added to the roadmap until a combination of some termination criteria are met: either the end pose is reached or the robot is within a short distance of the end pose from where a local planner can finish the route (the ‘end-game region’). Bi-directional planners take this a step further by growing two trees, one rooted at the start and one at the goal, with the search terminating when a connection is found between the two trees.

These techniques can be combined to make a planner that has the advantages of each and can offer a substantial improvement over basic PRM. The Single query Bi-directional Lazy collision checking planner (SBL) (Sánchez and Latombe 2003) is a planner that seeks to minimise the amount of collision checking required. Using an algorithm that includes the techniques listed can drastically reduce the computation time to generate a motion plan. While figures comparing SBL to standard PRM are not available, experiments comparing the running time of SBL to a standard single query PRM planner show it computes routes faster by a factor of at least 4. This is attributable to the lazy nature of collision checks carried out by SBL.

A closely related sampling planner is the Rapidly-exploring Random Tree (RRT) planner (Kuffner and LaValle 2000). RRT is a bi-directional, single query planner that builds two trees in the C-space from the start and goal configurations. In the pre-processing phase, new samples are created around the borders of the trees and connected back to existing tree nodes. Sampling includes a bias which encourages sampling in unexplored areas of C_{free} to ensure that the tree is capable of covering all of the valid space. In the merging step, RRT attempts to create edges between nodes in each tree. If successful, then it has found a continuous path from start to goal and returns success.

2.1.6.4 PRM for multiple robots

When planning for multiple robots, two basic strategies are available: centralised and de-coupled planning. Centralised planning essentially considers all the robots to be one single robot with high d.o.f and then plans accordingly (Sánchez and Latombe 2002a, page 16). The drawback is that this creates very high dimensional spaces which require much more time to plan through. De-coupled planners consider each robot separately and usually split planning into two phases (Sánchez and Latombe 2002b). The first ignores collisions between robots and generates paths for each of them individually, while the second phase uses a technique called velocity tuning which adjusts the velocity of each robot. If the velocities of the moving objects are known then the planner can annotate the route and set maximum/minimum speeds and wait-times at appropriate points to ensure the robots do not collide with other moving objects. A two stage algorithm for velocity tuning is used in Kant and Zucker (1986) which first computes a route free from static obstacle collisions and then adjusts the agent's velocity at specific route points in the second stage. Velocity tuning can also be used in multi-robot scenarios (Sánchez and Latombe 2002a) to ensure the various agents avoid each other. Methods for generating velocity profiles for agents can be found in Peng and Akella (2003; 2005).

De-coupled planning has the advantage of lower dimensionality than centralised planning, but is incomplete due to not being able to solve problems that can only be solved by a centralised planner. In experimental results presented in Sánchez and Latombe (2002b), the de-coupled planner fails to produce a valid plan in 30-75% of cases in problems involving 6 robots. The authors note that the de-coupled planner was *"at best only marginally faster than the centralised planner when they were successful."* However, de-coupled planners are still useful in producing plans in situations where centralised planners cannot be used.

2.1.6.5 Kinodynamic constraints

The original version of PRM assumes the world is static and that it is planning for a holonomic robot. To deal with these types of constraint, a new type of PRM planner is introduced in Kindel et al. (2000) that uses a single query, single directional planner to build a roadmap in time $\times$ space for a robot.

The method described expands the roadmap tree by selecting a robot input control at random and then picking a new value for that control. The equations of motion are used to predict the robot pose a short time interval in the future under this control input to reach a new node in the tree. This ensures that the planner implicitly obeys all kinodynamic constraints on the robot—something which cannot be guaranteed if new configurations are simply sampled in the neighbouring region. This change in graph construction changes the PRM algorithm from just using the robot C-space to using its action space as well. The query phase still operates in the C-space, but the graph is expanded by searching through the action space. To expand a node in the graph, a control input is selected and changed. The new configuration(s) reached is added as a child node to the graph. This is significant because adjacent graph nodes are close to each other in the action space as well as in the C-space, resulting from the integration of one action over a short period of time. This avoids the problem in multi query planners where nodes near each other in the C-space need not be close in the action space (see Section 2.1.2).

Experimental results of using the planner on a real robot with strict movement constraints show that the planner can successfully navigate a robot around several moving obstacles. Kindel et al. (2000) reports that the method scales to robots with high d.o.f. and also has the ability to re-plan in the middle of plan execution. This is needed if any of the moving obstacles change course unexpectedly, or to account for new objects. Several disadvantages do exist, however. Principally, the system needs to have full knowledge of the robot's constraints and input controls with their suitable parameters beforehand, since these are used to build the roadmap. Secondly, it *"assumes that the moving obstacles move along straight-line trajectories with constant velocities..."* which is often untrue in reality. Collisions between obstacles are also ignored—objects are allowed to overlap.

2.2 Markov Decision Processes

MDPs are a common tool allowing many problems to be formulated within a decision-theoretic framework, especially non-deterministic control problems. If an agent's environment is non-deterministic, it does not necessarily react the same way to a particular action each time it is performed; one of a number of outcomes may result. The agent has full knowledge of the likelihood of outcomes for an action so the aim of solving an MDP is to find when each action should be applied. The MDP framework is described in this section.

2.2.1 MDP concepts

We formalise the MDP framework by defining the environment the agent lives in and how feedback is given to the agent. In all MDP models, the current situation is defined through the system state. This encompasses all information about the agent's internal state and environment that it requires in order to make decisions. In a simple example of a light switch, the entire state can be represented by the status of the light: on or off. At any point in time, the agent can occupy only one state of a finite set of states $\mathcal{S}$—either the light is on or off. In each state s, the agent must choose one *action* from a finite set of actions $\mathcal{A}$. Actions allow an agent to do something in its environment. When an action is taken, the agent receives a numerical *reward* which indicates how good the choice of action was. The environment also changes to a different state in $\mathcal{S}$, which depends upon the action selected, e.g. executing "press switch" when the system state is "light off" changes the state to "light on". The agent then repeats this process, selecting a further action in the new state and receiving another reward, and so on.

This defines a two way relationship between the agent and its environment, with the agent carrying out actions, perceiving the effect via the change of state and judging the quality of its actions via the rewards it receives. In reality, each action may take variable amounts of time and several may be used concurrently, but to simplify the problem, virtually all MDP models treat the environment as episodic. Each action takes one unit of time and the resulting state change and reward received happen atomically and instantaneously before time is advanced one step. While this might appear to restrict the applicability of such models, this actually translates very neatly to reality in a wide variety of situations. Figure 2.3 illustrates the reward-act cycle of a typical MDP.

Ultimately, we want the agent to achieve certain objectives or *goals*. The agent wants to accumulate as much reward as possible. Therefore, the goals are not explicitly given to the agent, but the rewards given will be structured so that the agent acts to achieve one or more of the goals. Some problems have an explicit termination condition, delivery jobs or games for example, whereas in other problems, the agent is expected to act indefinitely, such as controlling flow rates in a manufacturing process. Individual rewards do not indicate how good a choice of action is in the long-term, they are simply the immediate pay-off for taking that action in that particular system state. Higher numbers generally indicate more reward, with negative rewards given to penalise the agent. However, this may be reversed with numbers simply representing costs. The exact setup is problem dependent, but whether the agent seeks to minimise cost or maximise reward is equivalent

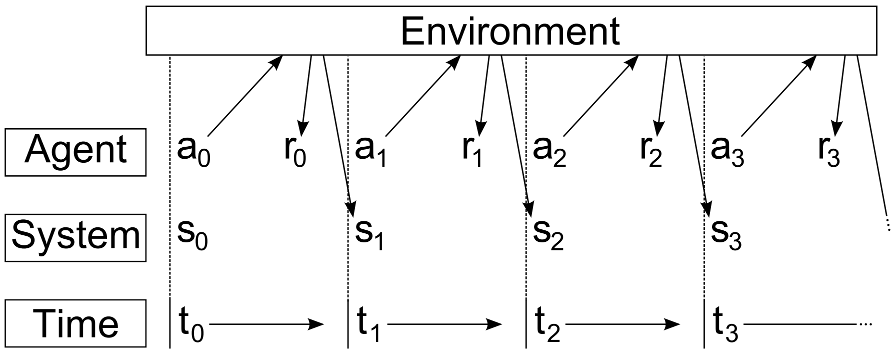


Figure 2.3: The reward-act cycle the agent repeats in the execution of an MDP. The agent observes state s_t starting from time t_0, receives reward r_t and observes the new state s_{t+1}.

within the environment. In many of the route-finding problems discussed later, we look to minimise the cost of various actions where the cost is determined by the length of the route.

The output from an MDP solution algorithm is a *policy* which instructs the agent on what to do in each situation. In standard models (assuming complete observability of the state) the policy is a mapping from states to actions, typically a lookup table. The agent looks at the current state of its environment and then examines the policy to find the best action for that state. After the action executes, the environment will have transitioned to a new state and the process repeats until termination.

2.2.2 MDP models

An MDP is characterised through the following formulation:

- The system occupies one state s, from a set of discrete states $\mathcal{S} = \{s_1, \ldots, s_N\}$. Each state encompasses all necessary information to describe the state of the system e.g. the position of the robot, its knowledge of the surroundings and other data such as "have collected flag" or "door 1 is open".

- In each state the agent selects an action a, from the set $\mathcal{A} = \{a_1, \ldots, a_K\}$ of K possible actions. Not all actions are applicable in every state. This can be modelled by having the action deterministically leave the system in the same state and imposing a large negative reward.

- The next state in the succeeding time step depends on the state and action executed in the current time step. Each state and action pair leads to another state in $\mathcal{S}$ with some probability. Thus, there is a probability distribution over the possible successor states for each $\langle s \in \mathcal{S}, a \in \mathcal{A}\rangle$ pair. These can be represented with an $N \times N \times K$

cube transition matrix $\mathcal{T}$, containing the probability of reaching state j from i by executing a particular action, where $\mathcal{T}_a = [t_{ij}]$ is an $N \times N$ square matrix containing the transition function for action a. The matrix elements t_{ij} specify the probability of reaching state j from state i and must satisfy

$$t_{ij} \geq 0 \qquad i, j = 1, \ldots, N$$

and

$$\sum_{j=1}^{N} t_{ij} = 1 \qquad i = 1, \ldots, N$$

to ensure each matrix row represents a valid probability mass function (p.m.f.). The complete transition matrix $\mathcal{T}$, consists of all K $\mathcal{T}_a$ matrices:

$$\mathcal{T} = \bigcup_{a \in \mathcal{A}} \mathcal{T}_a$$

These specify the probability

$$P(s_{t+1} = s' | s_t = s, a_t = a) \qquad \forall s, a, s', t = 0, 1, 2, \ldots \tag{2.1}$$

where s_t and a_t represent the state and action at time t respectively and

$$t(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$$

is a transition function that returns the probability in (2.1).

- A reward matrix $\mathcal{R}$ of dimension $N \times K$ specifying the rewards for each $\langle s, a \rangle$ pair. This is represented by the reward function:

  $$r(s, a) : \mathcal{S} \times \mathcal{A} \mapsto \Re$$

  which returns the reward for executing a in state s.

The value of any state depends upon the value of the future states that will be visited according to the policy. We want the agent to learn the *optimal policy*. This is the policy that maximises the *total expected reward* the agent receives from any given starting state. The emphasis is to highlight that we do not want the agent to greedily find the highest reward action for each state, but to find the actions which lead to the best eventual reward. Situations commonly arise where an action with a high associated reward in one state leads to other states where only low reward can be obtained. Conversely it is common that an agent must endure some amount of penalty before it can obtain the best rewards.

Blindly selecting the best immediate reward can harm the agent's overall performance in a task. An optimal policy is guaranteed to exist for every properly defined MDP; for proof see Theorem 3.3.2 of Martin (1967).

2.2.2.1 Markovian Assumption

An important property of the MDP formulation is the Markovian nature of the system. This assumes that at each time step, the current state depends solely on the previous state, not the entire history of states since time t_0. By incorporating all the relevant environment information into the state description we relieve the agent from the burden of tracking its state history, since all knowledge is included in the current state. In reality, the optimal action often depends on the entire history, but this formulation allows the following equality (Sutton and Barto 1998):

$$\begin{aligned} P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, r_1, s_0, a_0) \\ = P(s_{t+1} = s', r_{t+1} = r | s_t, a_t) \end{aligned} \tag{2.2}$$

where s_i, a_i and r_i indicate states, actions and rewards respectively at time step i.

There is no explicit encoding for goal states in an MDP (if any exist). Instead, specific goal states are represented through *absorbing* states which prevent the agent from escaping. The transition function for absorbing states are set so that all actions self-transition with probability 1 and all rewards are set to zero (state 'S2' in Figure 2.4). Once in an absorbing state, no further reward can be obtained and no other states can be reached.

A policy for an MDP is denoted π, and the optimal policy is denoted by π^*. These take the form of a one to one mapping from states to actions:

$$\pi : \mathcal{S} \mapsto \mathcal{A} \qquad \forall s \in \mathcal{S}$$

In order to compute π^*, we need to know the utility of being in each state. This is what a value function V^π tells us. V^π is a vector of size N containing the total reward we can expect to obtain from executing policy π starting from each state in $\mathcal{S}$. The value function of π^* is denoted V^*. If V^* is known, then we can extract the optimal policy according to the following:

$$\pi^*(s) = \operatorname*{argmax}_a \left[r(s, a) + \sum_{s'} t(s, a, s') \cdot V^*(s') \right] \tag{2.3}$$

A method for computing V^* is described in Section 2.2.2.3.

2.2.2.2 Infinite Horizon Problems

In many MDPs, the problem is guaranteed to terminate under any sensible policy after a finite number of steps once the agent reaches a goal. The value for any state is then the total reward obtained when the goal is reached:

$$\begin{aligned} V^{\pi}(s) &= r_{t_0} + r_{t_1} + r_{t_2} + \ldots + r_{t_n} \\ &= \sum_{i=0}^{n} r_{t_i} \end{aligned} \tag{2.4}$$

However, in many problems, known as *infinite horizon* MDPs, the agent is expected to act for an indefinite length of time. In this case, some reward will be accrued at each step leading to infinite rewards being obtainable since the MDP will never terminate. This presents a problem since the agent now has infinite time to obtain reward, meaning it has no motivation to make good choices—it will always eventually obtain the infinite reward. One policy, π' is preferred to another policy π if

$$V^{\pi'}(s) \geq V^{\pi}(s) \qquad \forall s \in \mathcal{S} \tag{2.5}$$

thus when infinite rewards are obtainable, it can clearly be seen that all policies are equivalent.

Obviously we still want the agent to try to act optimally, so to avoid this problem we use discounting. The value of future rewards is reduced by some factor $\gamma \in [0, 1]$ (the *discount factor*) so that all rewards are finite (when $\gamma < 1$). Discounting is used to affect the agent's preference for short term rewards versus long term rewards. A low discount factor places more importance on rewards gained a short distance into the future, in essence making the agent greedy as it will seek to gain quick reward, even if that reduces the amount it may eventually gain. A high discount factor encourages the agent to forego short term gain and even suffer penalties if that enables it to obtain a higher eventual reward. This is useful on finite horizon problems as well because we may want to limit how far into the future it thinks, even if the MDP will terminate after a known amount of time.

Setting $\gamma = 1$ is equivalent to an undiscounted value function and will produce the problems outlined above, so is rarely used. Lower values for γ bias the agent more towards short term rewards. When $\gamma = 0$ the agent ignores future rewards completely and chooses a policy that <u>only</u> considers the immediate reward for the next action, i.e. the optimal value function becomes:

$$V(s) = \max_{a} \; r(s, a) \tag{2.6}$$

When using a discount for infinite horizon problems, the value function can be interpreted as the maximum expected discounted reward when following policy π from state s:

$$V^{\pi}(s) = E\left[\sum_{i=0}^{\infty} \gamma^{i} r_{t_i}\right] \tag{2.7}$$

2.2.2.3 Computing V^*

Computing the optimal value function requires finding the value of each state under the optimal policy, π^*. Obviously, calculating this directly is impossible as π^* is unknown. If the value of each future state s' of state s is known under policy π, then we can calculate the value of s as the reward for executing a in s (specified by π) plus the weighted sum of the future values:

$$V^{\pi}(s) = r(s, \pi(s)) + \gamma \sum_{s'} t(s, \pi(s), s') \cdot V^{\pi}(s') \tag{2.8}$$

As π is not available to us beforehand, one possible way to find V^* is to compare the value of states under all different policies. This is possible because N and K are known quantities. Even for small MDPs, the combinatorial explosion in the number of possible policies makes this approach completely unrealistic. Bellman (1957) studied alternative methods for computing V^* and showed that the only efficient, scalable method was through dynamic programming. The principle of dynamic programming (DP) is to start with an initial estimate of state values and then revise that estimate. This initial estimate is typically $V(s) = 0$ for all states; however Dearden and Boutilier (1997, Section 4.1) have shown that using approximate solution methods to pre-compute values for a subset of states can reduce the time required to compute V^*. The estimate of the optimal value function is then repeatedly revised through successive approximation according to a DP algorithm. Several DP algorithms are available, but one of the most common which is efficient for large problems is *value iteration*, or simply VI. We use VI as a solution method throughout this research so the algorithm is presented here.

VI iteratively updates the value estimate for every state in $\mathcal{S}$ by greedily selecting the action that gives the highest future reward. The following is the Bellman optimality equation for V^* which is central to the VI algorithm:

$$V^*(s) = \max_{a}\left[r(s, a) + \gamma \sum_{s'} t(s, a, s') \cdot V^*(s')\right] \qquad s, s' \in \mathcal{S} \tag{2.9}$$

Comparing this to Equation (2.8), the difference is the presence of the *max* operator specifying that the optimal value function is greedy with respect to the action selection.

This is used as the update equation for VI and it is iteratively applied to each state in the MDP to update its value. The current estimate of the optimal value function $\hat{V}^*$, is used as the value function for the next iteration. Thus, VI finds V^* through a series of successive approximations:

$$\hat{V}^*_{t+1}(s) = \max_a \left[r(s,a) + \gamma \sum_{s'} t(s,a,s') \cdot \hat{V}^*_t(s') \right] \qquad s, s' \in \mathcal{S}, t = 0, 1, 2, \ldots \tag{2.10}$$

Algorithm 2.2 shows the VI algorithm in full. Termination requires a little further explanation. The VI update in Equation (2.10) is a contraction, i.e. every iteration brings $\hat{V}^*$ closer to the true V^* (2.9). After infinite iterations, the values for all states are guaranteed to have converged and we will have found the optimal value function as long as two conditions are met:

1. All rewards for all states must be finite and bounded, i.e. no action can receive an infinite reward.
2. The algorithm must allow for the value estimate of every state in $\hat{V}^*$ to be updated infinitely often. This means VI must not ignore states in $\mathcal{S}$.[1]

For proof of the convergence of $\hat{V}^*$, the reader is referred to Bellman (1957), Theorem 3.4.2 of Martin (1967, page 45) and Szepesvári and Littman (1996). In practice, we stop once the maximum difference between subsequent iterations (the *Bellman residual*) of any state value falls below some small parameter ϵ, typically $\epsilon < 0.0001$. The value function is then ϵ-optimal.

2.2.3 LAO*

Standard value iteration is to a certain extent a naïve algorithm because each iteration consists of a full sweep of the state space, performing one update of the value function at each state. This is a very general solution since in many problem classes, implicit structure in the transition matrix $\mathcal{T}$ means that many states have a very low probability of being visited and many will never be reached under an optimal policy. This observation has led to the development of several algorithms for solving MDPs (Dean et al. (1995); Barto et al. (1993); Dai and Hansen (2007) and Sutton and Barto (1998, pp.107-108)) that exploit this for better performance. These algorithms use varying forms of *asynchronous dynamic programming* (Bertsekas and Tsitsiklis 1989) which do not update the value at every state in the system, instead focusing updates on states which are most likely to impact the value of the starting state. LAO* (Hansen and Zilberstein 2001) is one

[1]We can utilise a tighter bound and only update a certain subset of states. See the next Section.

Algorithm 2.2 Value Iteration algorithm, adapted from (Sutton and Barto 1998, p.102) and (Russell and Norvig 2002, p.621)

```
Input: $\epsilon \geq 0$ maximum acceptable error
Output: $V(s)$ = optimal value for $s$, $\pi(s)$ = optimal action in state $s$
 1: $V(s) = 0 \quad \forall s \in \mathcal{S}$
 2: repeat
 3:    maxDiff $\Leftarrow 0$
 4:    for all $s \in \mathcal{S}$ do
 5:       $V_{\text{old}} \Leftarrow V(s)$
 6:       $V(s) \Leftarrow \max_a [r(s,a) + \gamma \sum_{s'} t(s,a,s') \cdot V(s')]$
 7:       $\pi(s) \Leftarrow \text{argmax}_a [r(s,a) + \gamma \sum_{s'} t(s,a,s') \cdot V(s')]$
 8:       maxDiff$\Leftarrow$ max(maxDiff, $|V(s) - V_{\text{old}}|$)
 9:    end for
10: until maxDiff $< \epsilon$
11: return $V, \pi$
```

such algorithm which is well suited to the types of "stochastic shortest-path to the goal" problems central to this research.

For generating the shortest path between a pair of nodes in an undirected graph, a simple algorithm similar to Dijkstra's (1959) algorithm could be used to find the minimum cost to every point from the goal. The route can then be traced forwards from the start by greedily selecting the lowest cost neighbour at each step. This is massively inefficient since the algorithm has calculated the minimum cost to every graph node, when only those on the optimal path will ever be visited. A* improves upon this by using best-first search and a heuristic to guide the search forwards from the start node to the goal node and only exploring a fraction of the nodes in the graph. If the guiding heuristic is admissible and monotonic (consistent), then it has been proven (Nilsson 1986) that A* is optimal and no other algorithm will explore fewer nodes under the same heuristic. The same principle is applied to MDPs in LAO*. As the starting state is often known prior to generating the policy, it makes sense to only reason about states that can be reached from the start. We call this "exploiting state reachability" because it uses the transition function to ignore unreachable states. Russell and Norvig allude to these more efficient algorithms when discussing MDPs:

> *"The freedom to choose any states to work on means that we can design much more efficient heuristic algorithms... if one has no intention of throwing oneself off a cliff, one should not spend time worrying about the exact value of the resulting states."*
>
> (Russell and Norvig 2002, p.625, 2nd edition)

LAO* is a generalisation of AO* (Nilsson 1986) that computes π^* by only considering states reachable from the start state s_0, under the current estimate of π^*. AO* is a heuristic best-first search algorithm for finding the minimal cost path through a conditional AND/OR graph. AO* produces the optimal path from a given start state to a goal in the form of an acyclic solution graph. AO* is not applicable to MDPs in general, since in most MDPs a state may be visited more than once, violating the acyclic constraint of AO*. A full description of LAO* can be found in (Hansen and Zilberstein 1998; 1999; 2001) but a sufficient description follows here. LAO* is heuristic based, like AO* and A* which it is derived from. While its predecessors construct a tree through the states in $\mathcal{S}$, LAO* creates a directed graph. This distinction facilitates its application to MDPs because any state may be visited infinitely often under an optimal policy. This implies that any solution algorithm needs to be able to handle loops in the state history. LAO* is designed to solve stochastic shortest-path MDPs with goals represented as absorbing states, i.e. they are terminal as no action can transition the agent out of an absorbing state. Shortest-path problems have an indefinite horizon because the number of steps required to reach a goal state is not bounded.

To understand the algorithm, some additional definitions will be introduced here. An MDP is often discussed in terms of a non-deterministic finite state machine (NFSM) where each MDP state is a state in the NFSM. As in a standard finite state machine, several actions are available in each state, all leading to new states. Non-determinism entails that any action may lead probabilistically to one of several new states. Each action edge from a state now consists of an arc to several new states, with an associated edge probability showing the probability of that new state being reached. Arcs also have an associated reward showing the $r(s, a)$ value (see Figure 2.4 for an example). All the states and arcs then form the *implicit graph* of the MDP. This is simply a reformulation of the MDP model parameters $\mathcal{S}$,$\mathcal{A}$,$\mathcal{T}$ and $\mathcal{R}$ in terms of a graph. As LAO* runs, it explores this implicit graph similarly to the way in which standard A* explores a weighted graph, building an *explicit graph* (denoted G) of explored states. Initially, this only contains the singleton start state but will contain the complete set of states visited by π^* upon algorithm termination. A smaller explicit graph implies a more efficient computation since less of the implicit graph has been expanded. Terminal nodes in either graph represent goal states as there are no transitions out of them. These have cost zero since no further reward/cost can be collected.

Until the computation is completed, G will only contain a partial MDP solution, including some unexpanded nodes without arcs (transitions) which are not goal states. These are known as *non-terminal tip nodes* to distinguish them, or simply tip nodes where no confusion may arise. The *partial best solution graph* (BSG), is the set of nodes which

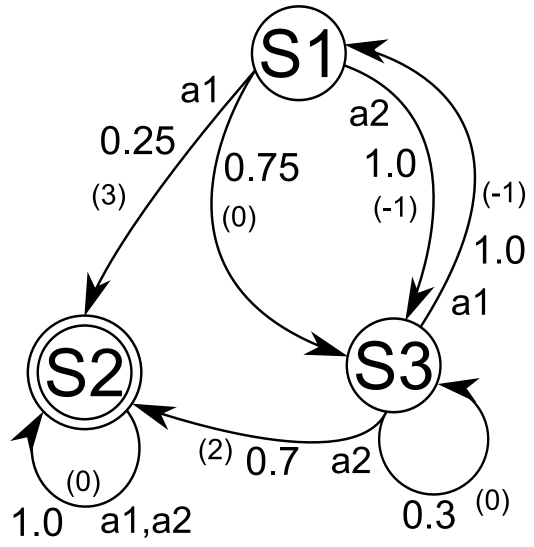


Figure 2.4: An MDP with 3 states and 2 actions. S2 is an absorbing state. Executing a1 in S1 may lead to S2 with probability 0.25 or S3 with probability 0.75. Both states are possible successors for the same action. Executing a2 deterministically leads to S3. Rewards are shown in brackets.

may be visited by the current estimate of π^*. The BSG, explicit and implicit graphs exist in a subset relation:

$$\text{BSG} \subseteq \text{explicit graph} \subseteq \text{implicit graph}$$

Each node in the explicit graph has a *best action* (marked n_a for node n) associated with it. This is the action that is believed to be optimal in that state and which defines the policy for the MDP. When the algorithm terminates, the set of graph nodes (MDP states) in the BSG, along with their best actions, completely describes π^*. A fundamental point that LAO* depends on to guarantee optimality is that only the state values in the final BSG can affect the value of the start state. All other states in the MDP which are not visited under π^* have no effect on the value of states in the BSG. This fact allows LAO* and the other algorithms mentioned earlier (Barto et al. 1993; Dean et al. 1995) to only explore a fraction of the state space yet still be certain of finding the optimal policy.

LAO* needs a heuristic $h(n)$, to guide its search as it generates the explicit graph. The same heuristic constraints apply as with other best-first search algorithms. The heuristic must be admissible and monotonic (a consistent heuristic implies both). Admissibility requires that the heuristic never overestimates the cost of reaching the goal from any state. Monotonicity requires local admissibility. If $h(n)$ is the heuristic estimate of the cost to the goal from state n and n' is a successor state of n, then

$$h(n) - h(n') \leq c(n, n')$$

where $c(n, n')$ is the edge cost between n and n'. That is, the heuristic must not overestimate the traversal cost between two neighbouring states. All monotonic heuristics are admissible. Violation of these can lead to non-optimal policies being returned. The overestimation of a node's cost can lead LAO* to leave a branch unexplored, thus allowing the possibility of a lower cost path to the goal remaining undiscovered.

The full description of LAO* is shown in Algorithm 2.3 (page 37). LAO* works in two phases: graph expansion (lines 3 to 8) and convergence testing (lines 10 to 23). These phases are interleaved and execution may switch between them several times before the algorithm terminates. The first phase is always graph expansion. In all forms of LAO* presented in Hansen and Zilberstein (2001), the expansion step repeatedly expands the explicit graph by selecting a non-terminal tip node. A tip node is expanded by generating all the successor nodes for that node and adding them to the graph (line 5). Successor nodes are any that may be reached by following any action from that node. The successor set Z of node i can be defined as:

$$Z = \{\text{all } j \in \mathcal{S} : \; t(i, a, j) > 0, a = a_1, \ldots, a_K\} \tag{2.11}$$

The `expand` function (line 5) is problem dependent, but in general creates set Z above and assigns each node in Z a value according to the current heuristic, or zero if it is a goal node. Determining which node to expand next is non-deterministic—it does not matter what order nodes are expanded in (though it can affect performance); the optimal solution will still be found. After expanding a node, the values of some nodes in the BSG may need revising as a result. Node values are calculated from the values of their successor nodes, so an altered node value can affect the value of all of its ancestors. The ancestors of a node are the set of all nodes in the explicit graph that can reach the newly expanded node via their best action, i.e. the set of nodes in the BSG which lead to the new node (set Y in line 6). All these values are updated by performing VI on the nodes in set Y (line 7) until their values have converged (according to the VI stopping criteria). The expansion step continues until there are no more unexpanded tip nodes in the graph.

When no tip nodes remain in the BSG, LAO* switches to the convergence testing phase. Here value iteration is performed over all nodes in the BSG (usually using a depth-first post-traversal order). Value iteration continues until one of two stopping conditions are met: either all the node (state) values in the BSG converge to within ϵ of their true value, or a new tip node is discovered (line 24). It turns out we only have to calculate the upper and lower bounds on the starting node s_0 to tell if all the other node values in the BSG have converged. The current value $f(n)$, provides the lower bound $\underline{f}(n)$, for any node. Line 19 defines the mean first passage time for node n with the function $\phi(n)$. This is the expected number of steps to reach the goal under the current policy from node

n. Line 17 computes the Bellman residual for the current iteration of VI. The mean first passage times for all nodes in the BSG form a set of linear equations that can be solved iteratively with a dynamic programming algorithm. The upper bound $\overline{f}(n)$ (line 20) is computed based on these calculations. When s_0 satisfies

$$|\overline{f}(s_0) - \underline{f}(s_0)| \leq \epsilon$$

LAO* terminates having found the ϵ-optimal solution. Under the second stopping criterion, if a node's best action changes during VI, the BSG may change to contain unexpanded tip nodes. When this occurs, VI is terminated and LAO* returns to the expansion phase.

2.2.3.1 Improving the expansion phase

The number of nodes evaluated (backed up) by LAO* has a large impact on its runtime. A more accurate heuristic will expand fewer unnecessary nodes in G leading to fewer evaluations, but improved heuristics are not always available. In Hansen and Zilberstein (2001), it was found that LAO* often performed thousands of backups on many nodes before the algorithm terminated, lengthening the runtime. This is exacerbated by the need to perform VI after each tip node is expanded. They present an optimised version of LAO* which replaces the body of the expansion phase (lines 4–7) with Algorithm 2.4. The convergence testing phase of LAO* remains unaltered. This improved expansion phase limits the number of node evaluations in two ways. Firstly, many tip nodes are expanded together before revising their values and secondly, local value backups coincide with the node expansion step and replace the value iteration step in line 7 of Algorithm 2.3. The nodes in the BSG are visited in depth first post-traversal order and when an unexpanded tip node is found, it is expanded (lines 2–4). Local Bellman updates (equivalent of line 6 of Algorithm 2.2) occur at each node before the next tip node is expanded (line 5). The depth first post-traversal ordering of the BSG ensures that all ancestors are eventually updated. We use this optimised version of LAO* for all experiments in this research.

2.2.3.2 Memory bounded algorithms

Chakrabarti et al. (1989) describe memory bounded adaptations of the well known A* and AO* algorithms (from which LAO* is derived). In MA*, an upper bound is enforced on the amount of memory the algorithm may use. When the number of nodes on the open or closed list in the graph search reaches a certain limit, some nodes are pruned from the lists. This is essentially a trade-off between the number of node expansions and the required memory. There is a computational penalty since pruned nodes may later need to be re-expanded. Heuristic estimates from pruned nodes are back propagated through parent nodes in the search tree to ensure information from their prior expansion is not lost. The MAO* algorithm achieves the same objective for AND/OR graphs. The presence of AND nodes makes the pruning steps more complex—individual nodes may no longer be pruned, only entire arcs may be pruned atomically. OR node arcs only have one successor, so the pruning steps for an OR node remain unchanged. As in LAO*, MAO* maintains a partial solution graph (not a specific path) and the returned result is the best solution graph, not a path of nodes from start to goal.

Memory bounded algorithms also cause problems with heuristic consistency. Zhou and Hansen (2002) show that even with the presence of a "pathmax" feature (Mero 1984), an inconsistent heuristic can still cause a node to be expanded before the best path to it has been found, something which cannot occur in standard A* with a monotonic heuristic. Chakrabarti et al. incorrectly state that inconsistent heuristics remain consistent under memory bounded conditions as long as pathmax is used. Zhou and Hansen (2002) show that this is not the case:

> *"But in graph search, two nodes may occur in the open list before all arcs between them have been generated... a node may be expanded before the best path to it is found."*
>
> (Zhou and Hansen 2002)

Pathmax only keeps an inconsistent heuristic consistent along paths that exist in the explicit graph, i.e. those nodes which have already been generated and expanded. As pruning operations remove parts of the graph after expansion, the part of the information needed to ensure a consistent heuristic is lost.

There is not a simple generalisation of MAO* algorithm for LAO* due to the presence of loops in LAO* solution graphs. In AO*, solution graphs are acyclic, so the cost revision step is a single iteration, back-propagation routine that backs node values up from the leaves towards the start node. This is replaced by the dynamic programming step in LAO* because nodes may be visited infinitely often in a loop (as is the case in an MDP). The authors of LAO* note that a similar memory bounding approach should be possible with their algorithm (Hansen and Zilberstein 1998, "Forward Search"). However, we believe the existence of cycles in the graph is what is ultimately hindering development of an equivalent extension of LAO*. It remains conjecture at this stage, but it would be a useful aim for future research to determine precisely whether a memory bounded version of LAO* could be constructed.

Algorithm 2.3 LAO* algorithm (Hansen and Zilberstein 2001, p.47)

```
Input: MDP model $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, starting state $s_0$, error bound $\epsilon \geq 0$, discount factor $0 \leq \gamma \leq 1$
Output: $\epsilon$-optimal policy returned as a directed graph rooted at $s_0$
 1: $G \Leftarrow s_0$ //explicit graph only contains start node
 2: $BSG \Leftarrow$ create list of nodes in BSG by following best actions $n_a$ in depth first post-traversal order from $s_0$ //initially $BSG = \{s_0\}$
 3: while $BSG$ contains non-terminal tip nodes do //expansion phase
 4:     $n \Leftarrow$ unexpanded tip node in $G$
 5:     expand($n$) //adds successor nodes of $n$ to $G$
 6:     $Y \Leftarrow \{$all ancestor nodes of $n$ that can reach $n$ by following best actions$\}$
 7:     perform VI on nodes in $Y$, updating values and best actions
 8: end while
 9: $BSG \Leftarrow$ recompute BSG
10: repeat //convergence phase
11:     $r \Leftarrow 0$
12:     for all $n \in BSG$ do
13:         $f_{\text{old}} \Leftarrow f(n)$
14:         $Z \Leftarrow \{$all successor nodes of $n\}$ //as defined in Equation (2.11)
15:         $f(n) \Leftarrow \min_{a \in \mathcal{A}} \left[ r(n,a) + \gamma \sum_{z \in Z} t(n,a,z) \cdot f(z) \right]$ //value backup
16:         $n_a \Leftarrow$ minimal action $a$ in line 15
17:         $r \Leftarrow \max(r, f(n) - f_{\text{old}})$ //$r$ is Bellman residual
18:     end for
19:     solve $\phi(n) \Leftarrow 1 + \sum_{z \in \mathcal{S}} t(n, n_a, z)\phi(z) \quad \forall n \in BSG$ //linear equations solvable by dynamic programing
20:     $\overline{f}(s_0) \Leftarrow f(s_0) + \phi(s_0) \cdot r$ //upper bound on $s_0$
21:     $\underline{f}(s_0) \Leftarrow f(s_0)$ //lower bound on $s_0$
22:     $BSG \Leftarrow$ recompute BSG
23: until BSG changes to include a non-terminal node or $|\overline{f}(s_0) - \underline{f}(s_0)| \leq \epsilon$
24: if new non-terminal found then
25:     goto line 2
26: else
27:     return BSG as $\epsilon$-optimal policy
28: end if
```

Algorithm 2.4 The improved expansion phase for LAO* (Hansen and Zilberstein 2001, p.56)

```
1: for all $n \in BSG$ do
2:     if $n$ is unexpanded then
3:         expand($n$) //adds successor nodes of $n$ to $G$
4:     end if
5:     backup node $n$ as in lines 15 and 16 of Algorithm 2.3
6: end for
7: $BSG \Leftarrow$ recompute BSG
```

2.3 Partially Observable Markov Decision Processes

In the previous Section the agent was always aware of the system state; the agent could see all the changes in the environment. In life this will not always be the case—the agent will not be able to see the state directly. There may be many reasons for this, but typically it manifests due to the limited perception of the agent's sensors. They may have limited range or some information will not directly available. Compare playing chess to playing poker: the game state, including all opposing pieces, is completely observable in the former, whereas only your hand of cards and the betting money is observable in the latter.

MDPs, which assume complete observability, are not directly applicable to this new type of environment, so a broader formulation known as a "Partially Observable MDP" (POMDP, pronounced *pom-dee-pee*) is needed. In contrast, a standard MDP can be strictly referred to as a "Completely Observable MDP", though this idiom has never entered common parlance.

A POMDP model consists of the same basic system mechanics as an MDP with $\mathcal{S}$,$\mathcal{A}$,$\mathcal{T}$ and $\mathcal{R}$ remaining unchanged. At any point in time the system exists in exactly one state in $\mathcal{S}$, but knowledge of the precise state is not available to the agent. Instead, the agent receives *observations* from the environment based on the current state and action choice. These allow for an abstraction of sensory input. Many sensory models such as range finders, cameras and wall detectors can be represented through this model. After executing action a from state s, the agent receives one observation z, from a set of observations $\mathcal{O}$. This observation set is often finite, though other models have been explored, such as the continuous observation spaces explored in Hoey and Poupart (2005). Observations are received probabilistically according to an observation function $o(z, s, a)$:

$$o : \mathcal{O} \times \mathcal{S} \times \mathcal{A} \mapsto [0, 1] = P(z|s, a)$$

which is the probability of receiving observation z after executing a in state s. The entire model for a POMDP therefore consists of $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{O}, o \rangle$ containing the standard MDP components plus the observation set and observation function.

Extending the light switch example from the previous Section to a POMDP, imagine the agent can no longer tell if the light switch is on or off. Instead it only possesses an inaccurate light dependent sensor. The *system state* is still either "light on" or "light off" but the agent can only infer the true state through observations. The set $\mathcal{O}$ for this problem consists of "on" or "off", one of which is received whenever the agent presses the switch. The probabilistic nature of observations means the agent may still observe "on" when the state is "light off" and vice versa. Observations do not deterministically infer the state—there is a non-zero probability of observing that the light is off when it is on. The important point from the example is that if the agent is unsure of the system state, receiving an observation(s) will still leave a certain amount of uncertainty in the state.

The same observations can occur in multiple system states due to the non-determinism of observations. This is known as *perceptual aliasing*. Imagine a corridor with several closed doors leading off it, all looking identical. If the agent was placed randomly somewhere along the corridor, it initially would have no information about its location so might arbitrarily decide to move forwards down the corridor. If it then observes a door on its right then this alone does not tell it where it is. It can narrow its choices down to corridor locations with doors on the right hand side but still cannot distinguish the exact one. The same observation would result from being next to any of the right hand doors, or any left hand door when facing the other way.

2.3.1 Belief states

As the agent no longer knows which state it is in, then it must have a way to represent its uncertainty about the state of the system. While assuming that the true state is the most likely state based on previous knowledge or the most recent observation is a possibility, a better approach is to explicitly model the uncertainty. As opposed to storing a single state, the agent now maintains a current *belief state*. This is a probability distribution over all the possible states of the system. A belief state b is therefore a vector of size N where $b(s)$ is the probability of being in state s where b must satisfy

$$b(s) = [0, 1] \qquad s = 1, 2, \ldots, N$$

and

$$\sum_{s \in \mathcal{S}} b(s) = 1$$

to ensure a valid probability distribution. The belief state encompasses the agent's total knowledge about the state of the system, including its uncertainty about the states. For example, in a completely uninformed state, then $b(s) = \frac{1}{N}$ for all s, showing all states are equally likely. A useful property of representing belief states as probability distributions is that they naturally incorporate all the knowledge (i.e. observations) gained since the start of the system. As long as the initial belief state b_0, at t_0 is specified, then b is a sufficient statistic to describe the current belief at time t_i based on all observations $o_0, o_1, \ldots, o_{i-1}$. This property ensures that storing belief states alone is enough to maintain the Markovian property of POMDPs.

The policy in a POMDP is described in a different form to an MDP policy which took the form of a mapping from states to actions. A POMDP policy must be changed to reflect the lack of precise state knowledge. Each belief state is a point in the total belief space $\mathcal{B}$, which takes the form of an $|N-1|$-dimensional simplex covering every possible belief an agent could be in. Therefore, a POMDP policy is a mapping from belief states to actions, not states to actions:

$$\pi : \mathcal{B} \mapsto \mathcal{A}$$

The life cycle of the agent in a POMDP consists of the following steps:

1. Start with initial belief b_0.
2. Execute action a from $\mathcal{A}$ according to current policy π.
3. Receive reward $r(s, a)$ from environment for executing a in state s.
4. Receive observation $z \in \mathcal{O}$ from environment with probability $o(z, s, a)$.
5. Update b according to a and z and go to step 2.

Updating belief state b_{t_i} after executing a and observing z to new belief state $b_{t_{i+1}}$ is done via the following belief update equation for each $s' \in \mathcal{S}$ (taken from Kaelbling et al. (1998, page 11), normalisation from Littman (1994)):

$$\begin{aligned}
b_{t_{i+1}}(s') &= P(s'|z, a, b) \\
&= P(z|s', a, b)P(s'|a, b) \\
&= \frac{P(z|s', a)\sum_{s\in\mathcal{S}} P(s'|a, b, s)P(s|a, b)}{P(z|a, b)} \\
&= \frac{o(z, s', a)\sum_{s\in\mathcal{S}} t(s, a, s')b_{t_i}(s)}{P(z|a, b)}
\end{aligned} \tag{2.12}$$

where $P(z|a, b)$ is the normalisation constant ensuring b sums to unity:

$$\begin{aligned}
P(z|a, b) &= \sum_{s'\in\mathcal{S}} P(z, s'|a, b) \\
&= \sum_{s'\in\mathcal{S}} P(s'|a, b)P(z|s', a, b) \\
&= \sum_{s'\in\mathcal{S}}\sum_{s\in\mathcal{S}} P(s', s|a, b)P(z|s', a) \\
&= \sum_{s'\in\mathcal{S}}\sum_{s\in\mathcal{S}} P(s|a, b)P(s'|s, a, b)P(z|s', a) \\
&= \sum_{s'\in\mathcal{S}} o(z, s', a)\sum_{s\in\mathcal{S}} t(s, a, s')b_{t_i}(s)
\end{aligned}$$

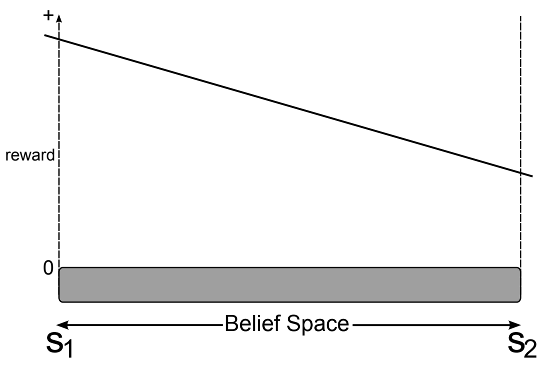
2.3.2 Value function

An MDP value function is specified through a vector of size N giving the expected long-term reward of being in one particular state. In a POMDP we adapt this to specify the value of being in a particular belief state. The first issue is that $\mathcal{B}$ is a continuous space, so a value vector is no longer sufficient since it would need to be of infinite size. If we consider an MDP value function, the value for a state specifies the expected reward obtainable by executing the associated policy from that state. In a POMDP, the value function is represented as a series of hyperplanes ($|N|$-dimensional planes; a 1D plane is a point, a 2D plane is a line) through the belief space. These hyperplanes extend in all directions to the edges of the belief space.

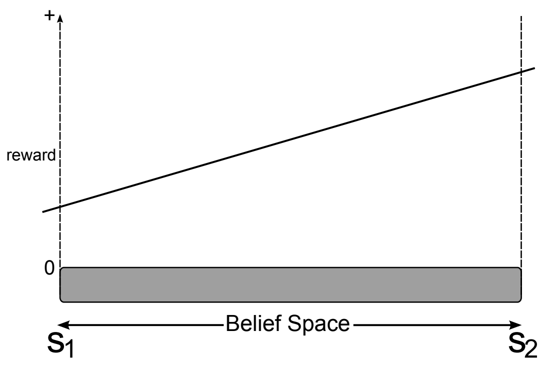
We will illustrate this with an example of a two state, two action POMDP which conveniently restricts $\mathcal{B}$ to one dimension. Looking at Figure 2.5, $\mathcal{B}$ is represented by the horizontal bar at the bottom of each diagram showing the complete range of belief states. When the agent is 100% certain of being in state s_1 its belief state would be at the extreme left of the diagram, conversely the extreme right represents being 100% certain of being in s_2. The vertical axis shows the reward for acting and Figures 2.5a and 2.5b show the expected reward for executing action a_1 and a_2 respectively. In the underlying MDP, executing a_1 obtains a much higher reward in state s_1 than in s_2, as shown by Figure 2.5a. The opposite situation exists in 2.5b where a_2 gains higher reward in s_2. For all the belief points in-between in 2.5a where there is uncertainty over whether s_1 or s_2 is the true state, then the expected reward decreases linearly over $\mathcal{B}$ as the probability shifts from being certain of s_1 to being certain of s_2.

As with MDP value functions, the POMDP value function greedily selects actions that maximise the future reward. Both of the individual action-value hyperplanes shown assume that a sensible (if not optimal) policy is executed after the first action is selected. The complete value function is therefore specified by combining the planes from parts 2.5a and 2.5b as shown in Figure 2.5c. The value function is then the upper surface of all the combined planes over the belief space (shown highlighted). This is the direct equivalent of the max operator in Equation (2.9) because the hyperplane that dominates all other hyperplanes at that point is the one that obtains the maximum reward. A hyperplane dominates at a point b if it is higher than all other hyperplanes at that point in the belief space. Formally, the optimal value function is given by the following (Littman 1994) (compare with Equation (2.9)) where b' can be computed via Equation (2.12):

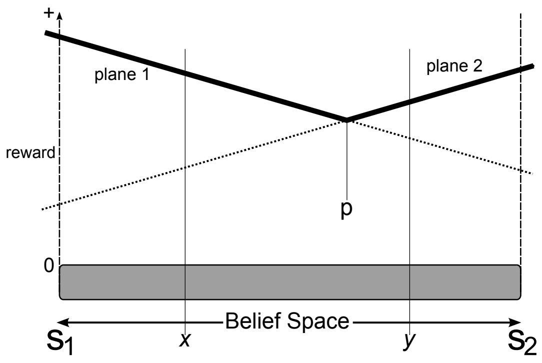
$$V^*(b) = \max_a \left[\sum_s b(s) r(s,a) + \gamma \sum_{z \in \mathcal{O}} P(z|b,a) \cdot V^*(b') \right] \tag{2.13}$$



(a) The expected value of executing action a_1 and following the policy thereafter



(b) The expected value of executing action a_2 and following the policy thereafter



(c) The combined value function from both actions

Figure 2.5: A value function for a 2 state, 2 action POMDP. Parts 2.5a and 2.5b show the reward that can be gained by executing action a_1 or a_2 from any belief state. Superimposing these in part 2.5c forms the complete value function where the upper surface (highlighted) represents the expected value from executing the maximal action. 2 belief points x and y are shown with respective optimal actions a_1 and a_2.

Given that b and b' are N dimensional vectors, in Equation (2.13) the new belief state b', is computed from the current belief state b, action a, and observation z. Equation (2.12) shows how the probability for each specific element in b' is computed.

It may seem like a convenient consequence of the example that the value function here happens to be made of two lines where the maximum reward decreases towards the centre of $\mathcal{B}$. This is where the agent is less certain of the true state. It sounds intuitively reasonable too, because if the agent becomes completely certain of which state is the true one, the POMDP reduces to an MDP where the agent can be certain of the optimal action; increased uncertainty makes it harder to work out which action is best. However, all POMDP value functions have this form, known as *piecewise linear convexivity* (PWLC), irrespective of the number of states, actions or nature of the transition and reward models. Sondik (1971) and Smallwood and Sondik (1973) first proved this and used it to create an initial method for computing optimal POMDP policies. This result is of crucial importance because it means that all value functions for <u>any</u> POMDP can be represented by a set of hyperplanes over the belief space where each hyperplane is dominant for at least one point in $\mathcal{B}$. Convexivity can be proved intuitively as well by looking at Figure 2.5: there is no possible arrangement of the two planes on the diagram that *do not* result in a PWLC value function—even if both are placed horizontally.

2.3.2.1 Policy Extraction

To extract the policy from a POMDP value function, the same logic used for MDPs applies; we want to select the action that obtains the maximum expected reward. In this case, this is the action associated with the maximal hyperplane at the specific belief point b, i.e. the hyperplane that forms the upper surface at b. Fortunately, this is easy to calculate as the dominate hyperplane is the one with the highest dot product with b. Each hyperplane can be expressed as a vector (known as an α-vector) of size N of the co-efficients of the hyperplane, so the value of point b represented by $V(b)$, can be computed according to

$$V(b) = \max_{\alpha \in \mathcal{V}} \sum_{s} b(s) \cdot \alpha(s) \tag{2.14}$$

where $\mathcal{V}$ is the collection of hyperplanes defining V. In Figure 2.5c, two belief states x and y are shown, each with a different maximal action. In all belief states to the left of point p (including x), hyperplane 1 dominates so a_1 is optimal, but hyperplane 2 (action a_2) dominates for all points to the right.

2.3.3 Solution algorithms

In symmetry to standard MDPs, extracting a policy from a given POMDP value function is a simple task, therefore the largest proportion of computational effort is in computing the value function. The optimal value function consists of the minimal set of α-vectors, each of which must be dominant for at least one point in the belief space. Once V^* has been found, the optimal policy can easily be found using Equation (2.14) for any $b \in \mathcal{B}$.

In computing an optimal value function, there are two problems which make finding set $\mathcal{V}$ harder. The first, sometimes referred to as the "curse of dimensionality" (Kaelbling et al. 1998), relates to the size of the state space: N. As the POMDP belief space $\mathcal{B}$ is a continuous space of dimensionality $|N-1|$, the overall size of $\mathcal{B}$ grows exponentially with respect to N. This was a problem for many early solution algorithms (Sondik 1971; Monahan 1982; Cheng 1988) and even small POMDPs (in the order of about ten states) quickly become intractable for these algorithms. More recent algorithms are better suited to larger POMDPs, but the complexity of $\mathcal{B}$ is still a major issue (Papadimitriou and Tsitsiklis 1987; Burago et al. 1996). The second problem termed the "curse of history" (Pineau et al. 2003), refers to the explosion in the number of possible trajectories a system may take as the number of time steps increases. At each time step, the agent executes an action and receives an observation which together determine the new belief state. The number of possible action-observation trajectories grows exponentially with respect to the size of $\mathcal{A}$ and $\mathcal{O}$. If the algorithm constructing the value function is essentially conducting policy search (Hansen 1998; Poupart and Boutilier 2003), then the large number of possible trajectories is a major obstacle in scaling to larger POMDPs. In general solving POMDPs is PSPACE hard (Papadimitriou and Tsitsiklis 1987).

Just as with MDPs, trying to determine an optimal policy or value function through systematic trials of possible policies becomes intractable for problems of even just a few states. The problem is even more pronounced than for MDPs, since we must consider belief states, actions *and* observations as opposed to just states and actions for each possible trajectory. POMDP solution algorithms broadly fall into one of two categories: exact and approximate. Both share a common approach to computing V^*, but achieve it in a different manner.

POMDP value functions are built by iteratively extending the planning horizon further into the future. At some horizon t, the optimal value function for that horizon V_t^*, will become equivalent to the true optimal value function V^*. Point t is guaranteed to exist so successive iteration of POMDP value iteration is guaranteed to eventually find V^* (Littman 1994, Section 3). The initial value function which looks one time step ahead can be simply computed from the reward function r, because the future is not considered at all; we only care about the maximum reward after one time step. Each successive horizon is built from the previous iteration's output as with VI in MDPs. We still deal with belief states instead of system states, and having selected an action, all possible resulting observations must be considered.

2.3.3.1 Exact algorithms

Exact solutions work by iteratively finding V^* for each successive horizon. The optimal value function $V_{t_1}^*$ is used to compute $V_{t_2}^*$ and so on, gradually projecting further into the future. Several exact POMDP solvers (Monahan 1982; Cheng 1988) have been proposed since Sondik formalised the POMDP model, but two fairly recent algorithms with significantly improved efficiency expanded the size of POMDP that could be solved in reasonable time.

The "Witness" algorithm (Littman 1994) uses a different approach to the previous POMDP solution algorithms by considering each action individually. Conceptually, it builds value functions for each specific action and then combines these to form V_t^* from V_{t-1}^*. The action-value functions are the equivalent of asking "What is the value of executing action a and then acting optimally thereafter?" Each of these functions will consist of a set of PWLC hyperplanes (α-vectors). When constructing each α-vector, a region where that plane is known to be dominant exists. For each one, Witness searches for a belief point where that choice of action is no longer optimal. This point is then witness to the fact that the current set of hyperplanes is not yet sufficient to completely describe the value function. The final step in computing V_t^* is to take the union of all the action-value functions and purge the set of hyperplanes that are completely dominated by others. That is, in the combined set, there will be some hyperplanes which are not maximal at any point in $\mathcal{B}$ so they can be safely removed from the set: they never contribute to the upper surface of the value function.

Incremental pruning (Zhang and Liu 1996; Cassandra et al. 1997) tries to improve one of the most computationally intensive steps of the Witness algorithm—the pruning stage. Deciding if a particular hyperplane is dominant for some belief state is achieved by solving a linear program. This either finds a particular belief point where that α-vector is dominant above all others in a set, or proves that no such point exists (in which case it can be pruned). This pruning of α-vectors is a major component of the POMDP solution algorithm and often a bottleneck in the derivation of a solution. The precise nature of the pruning step is intricate and beyond the scope of this review. However, the major contribution of the incremental pruning algorithm is, as its name suggests, that pruning can be carried out more efficiently using an incremental strategy on a per observation basis as opposed to pruning the larger set of alpha vectors used in earlier algorithms. Reducing the size of the pruning sets greatly improves the speed by which the minimal set of α-vectors $\mathcal{V}$ can be found to represent V_t^*. A further algorithm by Feng and Zilberstein (2004) improves on incremental pruning by splitting the belief space into smaller regions and pruning these regions separately.

2.3.3.2 Approximate algorithms

Approximate algorithms are a relatively recent approach to solving POMDPs which trade the optimality of exact algorithms for speed. This bottleneck severely restricted earlier algorithms in the size of problems they could solve (in Cassandra et al. (1997) the largest POMDP has 16 states), which limits their applicability to real world scenarios. Thus, the motivation for approximate algorithms is that a probabilistically optimal solution to a much larger problem is preferable to having to massively simplify the domain, or to having no solution at all. In one sense, these approaches shift from approximation in the model to approximation in the solution. The latter is generally preferred since we obtain nearly optimal solutions to models that are closer to the actual environment.

A significant, recent algorithm in this class (Pineau et al. 2003) is "Point Based Value Iteration" (PBVI). Instead of maintaining a minimal set of α-vectors to represent the value function, PBVI and similar approximate algorithms maintain a pool of belief points B, and the dominant α-vector for each. Given the observation that most optimal policies only visit a small proportion of the entire belief space, then a sufficiently large pool of points distributed over the belief simplex should accurately represent the true value function. Note that since each belief state in B has an α-vector (hyperplane) associated with it, the PWLC nature of the value function is maintained. The value function still extends over the entire belief space without respect to the distribution of the stored belief states. Value iteration in PBVI continues by alternating between two stages: value backup and belief point set expansion. The first stage updates the value α-vector at each belief point in B to look one further step into the future (i.e. increase the horizon by one time step). In the second stage, each point in B is stochastically simulated one step forward using their associated actions (recall each hyperplane has an associated action from $\mathcal{A}$) to generate a set of successor belief states. In the base version of PBVI, only the belief point farthest away from existing points in B (as measured by Euclidean distance) is added to the set. This ensures the size of B does not grow too quickly whilst promoting good coverage of $\mathcal{B}$. B is initialised to only contain the POMDP starting belief state, so B only includes those beliefs that can be reached by following the current estimate of the optimal policy. This is similar to LAO* (see Section 2.2.3) which also exploits belief point reachability. The largest problem tackled in Pineau et al. (2003) contains 870 states; far larger than what can realistically be achieved with exact algorithms.

Many other approximate algorithms exist with the aim of improving the tractability of POMDPs. Grid based algorithms (Lovejoy 1991; Zhou and Hansen 2001) divide the belief space into a discrete $|N-1|$-dimensional grid and distribute belief points at grid points. Not all approaches assume a regular grid (Brafman 1997), but they differ from in PBVI in that coverage of the full belief space is still required in order to adequately capture the optimal value function, since values are only stored at the grid points. Another point based algorithm (Spaan and Vlassis 2004) initialises B by randomly exploring the environment from the initial belief and storing all the belief states encountered. The value function is initialised with a single α-vector and then further α-vectors are added by sampling a subset of points from B and projecting the associated α-vectors one step into the future. The new α-vectors, that have a higher value at their respective belief points than what the current value function predicts are added to the set, while the remaining ones are discarded. In contrast to PBVI, not every point in B is used to update the value function; as the iterations progress, smaller subsets of B are updated. Results show it to be an order of magnitude faster than PBVI on standard POMDP problems while maintaining solution quality. This algorithm is explained in more detail in the next Chapter.

Chapter 3

Review of Related Material

3.1 Uncertainty in PRM

Many extensions to the original PRM framework have been devised since its inception in 1996. Here, we review some of the relevant extensions to the work. As mentioned (see Section 2.1.5), basic PRM is limited by not allowing for uncertainty directly in the plan, forbidding the use of moving obstacles, or any inaccuracies in the agent's perception of the world.

Uncertainty due to moving obstacles presents a challenging planning problem. In (Hsu et al. 2000) moving objects are treated as dynamic constraints. PRM with kinodynamic constraints was discussed in Section 2.1.6.5. The original PRM framework is extended by relaxing the constraint of a static environment. The trajectories of simple moving objects represented with constant velocity (direction and speed) are incorporated into the plan so collisions can be avoided. The planner's ability to handle unexpected trajectory change or the introduction of new objects is quite primitive with replanning the agent's path being the standard solution to such situations. The computational penalty for this action is bounded by restricting the time the agent has to replan to fractions of a second. In the 2D worlds investigated, the planner was allowed 0.25 seconds to re-plan which was sufficient for cases with relatively few obstacles. More importantly, the planner needs complete knowledge of the control input ranges of the agent it is planning for beforehand in order to construct a conflict-free path in the C-space for the robot. This is necessary to avoid violating the kinodynamic constraints of the robot. The problem of sensor uncertainty is not directly addressed (in experiments, the position of objects is measured from an overhead vision system) and the sensor measurements are assumed to be accurate. To account for inaccuracies, the radial size of obstacles is increased over time to avoid the planner erroneously asserting that a location is collision free. Even in simple domains, this is a pessimistic approach as free sections of the C-space would be ignored if the planning horizon extends too far into the future.

"Lazy-evaluation" (Jaillet and Siméon 2004) is another approach for planning around moving obstacles whereby the planner updates the roadmap nodes between queries. Any dynamic obstacles are ignored in the pre-processing phase of PRM and the roadmap is constructed around static obstacles. During the query phase, edges are validated to check that they are collision free, allowing for dynamic changes in obstacle positions. Repair strategies are applied in order of computational expense to re-route around blocked edges. A first attempt is to plan around the blocked edge using the existing roadmap nodes; if this fails then two single-query RRT's (see Section 2.1.6.3) rooted at the blocked edge's end points are grown to find a computationally cheap local route around the object. The third option is to expand the original roadmap with new points to plot an alternative route. In contrast to the previous approach, complete knowledge of moving obstacles at roadmap construction time is not required, but the assumption of accurate information at execution time is still present; there is no provision for uncertainty in the actual object locations. However, the dynamic nature of the query phase should allow for correction from minor inaccuracies in obstacle location data. The lazy-evaluation of roadmap edges allows for moving objects, but the planner is essentially still planning for a static world during repairing/replanning. An object moving close to the agent's path would trigger multiple repair/replan operations, potentially slowing the planner significantly. Using a two stage planner to deal with moving obstacles as Jaillet and Siméon (2004) have done in the above approach is a common technique. There are similarities between dealing with moving obstacles in this way and the velocity profiling methods for multiple agents discussed in Section 2.1.6.4. Typically in the first stage, moving obstacles and blocked edges are ignored and then routes are refined according to appropriate constraints in the second stage. Laumond et al. (1994) apply a similar technique in nonholonomic robot planning by ignoring the nonholonomic constraints in the first stage, then subdividing the route into smaller sections which can be solved while adhering to the robot constraints.

Rodríguez et al. (2006; 2007) have developed a heuristic planner for situations where object movement cannot be predicted which, unlike the previous approach, account for moving obstacles directly in the plan. Heuristic planners can compute motion plans around moving objects without knowledge of their future trajectories by leaving certain aspects of the plan unspecified. Rodríguez et al. distinguish two types of moving object: hard objects which are known, generally static objects such as walls, and soft objects which are smaller movable objects such as people or other agents. Collisions with hard objects must be avoided to ensure plan success, but soft collisions may be tolerated if no other course of action is available. The route planning algorithm described consists of two stages, employing both a global roadmap and a local planner to complete the route. The first stage of the planner uses a PRM-based global roadmap that is periodically updated to keep track of hard objects. The dynamic constraints of the agent and soft objects are ignored in this stage. Routes returned from querying the global roadmap should be free of hard collisions. This route (known as the global route) is treated as the heuristic since some objects may move. The local planner uses the global route to plan the route the robot will actually follow.

The second stage of the planner by Rodríguez et al. incorporates the dynamic constraints of the agent motion as well as the movement of soft objects to generate a path. Similar to the kinodynamic planner of Hsu et al. (2000), this entails that the local planner have full knowledge of the agent's control inputs. The planner incrementally generates the next few stages of the plan locally, since no knowledge about future obstacle movement is assumed. The global route is used as a guide by the local planner to build a tree of possible routes from the current location to the next sub-goal. The sub-goals will be points on or near the global route. Unlike previous methods, a complete path is not computed beforehand, only the path to the next sub-goal is planned at each step. Using the global route as a heuristic helps ensure the agent finds a collision free path; however, the lack of a global plan together with proximity to several soft objects creates the risk that the agent will become trapped in an unrecoverable position. The authors do not mention this possibility, but do acknowledge that when no local path can be found, some domain specific local planner may be required. Experimental data shows this effect when more than 10 soft objects are present in the environment since the probability of plan success drops below 80%.

Uncertainty in objects' motion is only one type of uncertainty inherent in PRM planning domains. Inaccuracy in sensor readings causes uncertainty even if the world is static. The available literature on planning under uncertainty is too diverse for one review, but some relevant approaches are discussed later in Section 3.2. PRM can be extended to deal with sensor uncertainty just as it can be extended to deal with moving objects.

Predictive PRM (Burns and Brock 2006; 2007) integrates sensor uncertainty directly into a PRM planner but without the requirement that the planner's world model incorporates uncertainty. Predictive PRM makes two contributions to the standard PRM algorithm to accomplish this. The Predictive PRM planner uses a lazy approach in line with many modern PRM planners which delay edge evaluation until query time. The roadmap edge costs reflect the utility of edges to the agent, where the probability of edge obstruction is used to alter the edge cost, making risky edges less attractive to the planner. The following is used for estimating edge cost:

$$G(e) = P(e = \text{obs.}) \cdot C(e) + \frac{P(e = \text{free})}{U(e)}$$

where $P(e = \text{obs.})$ is the probability that edge e is obstructed, $C(e)$ is the cost of the consequence of edge failure, e.g. collision, and $U(e)$ is the utility of the edge (task specific).

The idea of replacing the standard distance metric with one that incorporates the uncertainty about the edge usability is not novel. Fuzzy PRM (Nielsen and Kavraki 2000) annotates edges in the roadmap with a probability of success in the pre-processing phase. The motivation in Fuzzy PRM is that roadmap is constructed lazily, so until the local planner is used to verify a collision free connection between two nodes, success cannot be guaranteed. In Predictive PRM, the probability is necessary to account for the uncertainty in the world, whereas in Fuzzy PRM the edge state is only uncertain until the local planner conducts a proper collision check to determine its usability. The Minimum Collision Cost (MCC) planner (Missiuro and Roy 2006) also measures the probability of success of an edge for a similar purpose to Predictive PRM, so that edge costs depend on their usability as well as their standard distance cost.

The second contribution of Predictive PRM is that plans can be refined through additional sensing, at some cost to the agent. The planner selects a fraction of the most uncertain, but potentially useful edges to be refined through sensing, which either reduces or removes the uncertainty about them. In experiments, the planner is allowed to refine a set proportion of edges in a route, with results showing that a refinement rate of 50% can double the fraction of correct routes returned in a 14 d.o.f. scenario. The world model the PRM planner uses for collision detection does not need to directly incorporate uncertainty into its plan. A 3D occupancy grid model is demonstrated, as well as a vision-based model representing obstacle poses with Gaussian distributions. The Predictive PRM planner uses a sensor specific model of sensor error to allow integration with different systems. This is a useful attribute, increasing its applicability to other world-models, but could cause it to suffer when trying to interpret data from world-models that have no useful estimate of sensor error. With an occupancy grid model, the sensor error is based on explicit data obtained from a small number of example environments which may not directly translate to real world problems.

3.2 Uncertain Planners

Domains with inherent uncertainty are typical applications for MDP and POMDP planners, but the computational costs can limit their applicability. Other non-decision-theoretic methods for planning routes under uncertainty generally offer advantages in terms of speed. Sacrificing optimality, heuristics are often used to find routes.

3.2.1 Canadian travelling salesman problem planning

The travelling salesman problem (TSP) is a well-known construct in computer science and complexity theory, often used as a concrete illustration of an NP-complete problem. Given a weighted graph G formed of vertex set V and edge set E, the objective is to find the shortest cost path that visits all vertices (cities) exactly once. Of particular interest to this research is the Canadian TSP (CTP) extension. In CTP, certain roads connecting the cities the salesman must visit have become snowed under, thus unbeknownst to the salesman, not all roads on his map are usable. In routing theory, this means that some subset of edges in E are blocked before the agent begins to act, although their states are static once the agent enters the map. To make the problem applicable to planning research, it is assumed that the agent knows the probability that a given edge is blocked.[1]

Free Space Sensing Navigation (FSSN) (Bnaya et al. 2009) is an algorithm for planning CTP routes when the agent is augmented with some remote sensors. Two types of sensing are possible:

- *Local sensing* is the sensing of the states of all edges connected to the current vertex. This has no associated cost and happens automatically when the agent reaches a vertex.
- *Remote sensing* allows the agent to sense an edge not incident to the current vertex. There is a cost associated with remote sensing, either constant or dependent upon the distance to the vertex. In FSSN, remote sensing will never return an incorrect reading. We relax this assumption in our research and allow noisy observations.

FSSN plans a route from the start to the goal vertex with the lowest total cost which comprises both the travelling (sum of the traversed edge weights) and sensing costs. This is important for agents which need to minimise total resource usage (e.g. battery power). Bnaya et al. assert that while the problem could be modelled as a POMDP, its size is beyond what is feasibly solvable with current planners. While this is true for the general problem, there is much structure that allows for close approximations to the POMDP model to be solved efficiently. Edges in the graph may be in one of three states: free, blocked or unknown. FSSN relies on a "free space" assumption that all edges are free (including unknown ones) until proved otherwise through sensing. Pseudo code for FSSN is shown in Algorithms 3.1 and 3.2. FSSN creates a path from the current node n to the goal g (line 3) and then repeatedly traverses one edge at a time towards g. Local sensing is used at each step to remove blocked edges from G (line 8). At each vertex, the remaining path is verified to check it is usable by calling the `checkPath` function (Algorithm 3.2).

[1]Presumably, many bitter winters have taught the salesman which roads are susceptible to bad weather.

Algorithm 3.1 Free Space Sensing-based Navigation (FSSN) (Bnaya et al. 2009)

```
Input: $G = \langle V, E \rangle$ graph, $s, g \in V$ start and goal vertices
Output: Agent at $s$ moves to $g$ if route available
1: $n \Leftarrow s$  //$n$ is the current node
2: while $n \neq g$ do
3:    $P \Leftarrow$ shortest path from $n$ to $g$
4:    while next edge in $P$ not blocked do
5:       checkPath($P, n, g$)
6:       if success then  //path is OK
7:          $n \Leftarrow$ next vertex in $P$  //move to next node
8:          remove blocked edges from $G$ according to local sensing
9:       end if
10:   end while
11: end while
```

Algorithm 3.2 `checkPath` function for FSSN

```
Input: $P$ path of edges in $G$, $n, g$ current and goal vertices
Output: success if $P$ represents a usable path
1: $U \Leftarrow \{$uncertain edges in $P$ between $n$ and $g\}$  //set of unknown edge states
2: for all $u \in U$ do
3:    if checkEdge($u$) then  //if edge $u$ should be sensed
4:       remotely sense $u$
5:       if $u$ blocked then
6:          $G \Leftarrow G \setminus u$
7:          return fail
8:       end if
9:    end if
10: end for
11: return success
```

If `checkPath` fails or a blocked edge is found, the inner loop fails (line 4) and a new path is planned.

The `checkPath` function is crucial to algorithm behaviour as it dictates where remote sensing is employed. The `checkPath` function decides whether or not to remotely sense each "unknown" edge in path P. If it decides to sense an uncertain edge and finds it blocked, that edge is removed, failure is returned to FSSN (lines 4 to 7) and it will replan.

The unspecified `checkEdge` function allows different sensing policies to be plugged into FSSN. Simple policies such as "Always sense" and "Never sense" either sense all or none of the uncertain edges prior to motion. More complex heuristics that estimate the value of information from sensing can offer lower total route cost. The decision on remote edge sensing is then based on the sensing cost and the penalty incurred if the agent does not use sensing and has to backtrack in the case the route is blocked. Results presented show that utilising the value of information in sensing strategies proved best for nearly all examples, except where sensing cost was very cheap and uncertain edges only had a 10% chance of being blocked. If sensing costs were exceptionally expensive or cheap, "Never sense" or "Always sense" provided lower cost plans respectively.

The methods described above are very effective in this type of domain where the value of information can be easily estimated; however, deriving policies based on remote sensing narrows the range of problems that FSSN can be applied to. The reliance on a perfect remote sensor is often unrealistic; in most real-life situations sensors are noisy. In domains where perfect information is available, it is often provided as a "one-off" cost, for instance phoning a telephone helpline where data about a whole route is provided in one transaction. In other cases, such as consulting a satellite navigation system, the information is free. In either of these cases, one of the always/never sense policies will be optimal. Domains that provide information to the agent for a moderate cost will generally have some probability of noise in the information.

3.2.2 Uncertainty in forward search

Uncertainty can be integrated into a search algorithm by changing the plan space of the search. Standard path planning using best-first search algorithms such as A* plan solely in the space of possible locations (poses) for the agent. Uncertainty can be integrated by modelling the robot pose as a distribution over locations and planning in the extended pose×covariance space. Censi et al. (2008) take this approach and model the robot's location as a Gaussian distribution with the mean centred on the desired pose. The covariance is determined by the previous location's covariance and information obtained through sensors. The world is described as a polygonal environment discretised into cells; each action is selected from a finite set. Search is accomplished with a standard implementation of A*. The extended plan space permits plans that are optimal with respect to different criteria other than just finding the shortest route. By changing the relation defining the ordering of nodes in A*'s OPEN list, the planner can find the minimum cost route (in this case execution time) while keeping uncertainty about the location below a specified threshold (criterion 1). Alternatively, when successor nodes are generated, if nodes with a smaller covariance are explored first, returned plans will instead minimise the uncertainty for when the agent reaches the goal (criterion 2). Under this second criterion, more costly routes are favoured because the agent attempts to stay localised (although a maximum route cost is enforced). Planning in such a framework avoids the curse of dimensionality (see Section 2.3.3) that hampers MDP solvers because the planner does not have to consider all eventualities. The method of localisation is separated from the control, allowing various algorithms to be used (particle filters (Doucet et al. 2001) and

Kalman filters (Kalman 1960) are given as examples) while keeping the implementation flexible.

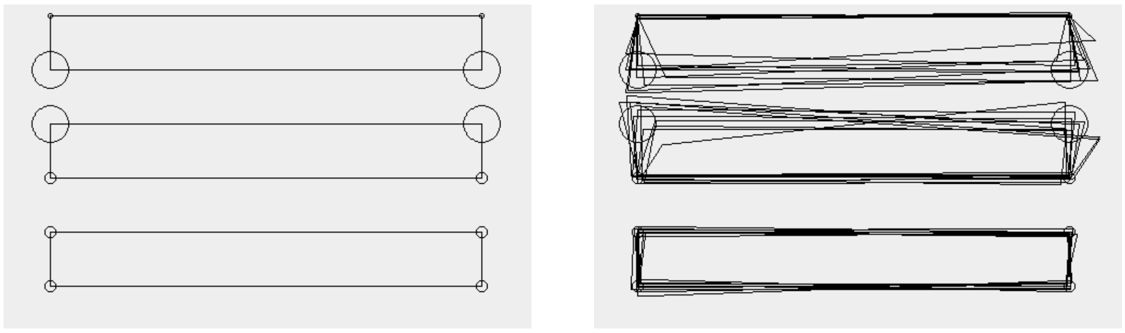
Some assumptions made by Censi et al. to aid localisation and reduce uncertainty are not always applicable. The uncertainty in robot pose is considered to be low with respect to the environment complexity, meaning most of the environment features are distinctive (e.g. obstacles), greatly reducing localisation uncertainty when the agent observes a feature. The produced plans make good use of the environment to find a safe path, but the computational machinery involved is reasonably complex; simpler heuristic planners that prefer routes close to objects would likely produce similar plans. Lastly, the sensors are modelled as a continuous source of information; the robot can gain information by not moving and simply observing. This is used to good effect in experiments with criterion 2 above since the robot deliberately pauses during plan execution until the location covariance drops to a permissible level. It is easy to construct problems where this will not help. Consider the classic localisation problem of a robot moving along a straight corridor with multiple identical doors and no other discerning features. If the robot stops and takes a photo of a door, then it can infer it is by a door, but not which one—a typical perceptual aliasing condition. According to the above assumption, this ambiguity in location could be reduced by standing still and collecting more information. However, collecting more photos (even thousands) will still only tell the robot it is standing by a door; the uncertainty is not reduced. The key intuition is that in many localisation problems, standing still and making more observations does not improve their quality or value, it just validates their original hypothesis, i.e. "I can see ONE of the doors."

3.2.3 Minimum Collision Cost planner

The Minimum Collision Cost (MCC) planner (Missiuro and Roy 2006) is a PRM motion planner that builds risk and uncertainty directly into the PRM algorithm. Some planners use subroutines of the PRM algorithm, such as the sampler, as an intermediary step for an uncertain planner as in Stochastic Motion Roadmaps (see Section 3.3.2.1), or as a high level heuristic route planner as in Rodríguez et al. (2006) (see Section 3.1). MCC uses a combination of adapted sampling routines and an adapted edge cost calculation to directly incorporate uncertainty into the standard PRM algorithm.

MCC deals with map uncertainty where the locations of obstacles in the environment are not known precisely. This could be attributed to inaccurate sensor readings, however unlike planners for noisy sensor data, no sensing is assumed—the agent's knowledge is constant through planning and execution.[1] All obstacles are represented by polylines defined by their vertices. As obstacles' locations are uncertain, each obstacle vertex is

[1] This follows since standard PRM is not an online planner.


(a) Example environment with uncertain obstacle locations

(b) Superimposed obstacle samples from the distributions

Figure 3.1: An uncertain corridor environment with 3 obstacles. The covariance matrices are shown as ellipses at 1 standard deviation. The larger covariances in obstacles forming the upper corridor make traversing it much more risky.

represented by a bi-variate Gaussian distribution with the mean centred on the most likely position for that vertex. The degree of uncertainty is described through the covariance matrix, which in 2D can be represented as an ellipse drawn at a multiple of the standard deviation from the mean. Figure 3.1a shows an example corridor environment with ellipses around obstacle vertices showing the uncertainty. Possible obstacle locations can then be generated by sampling from each of these distributions. Several possible samples for the corridor environment are shown in 3.1b. The distributions forming the upper corridor have greater covariances because the locations are less certain; this is reflected in the greater variation in the samples in 3.1b. Travelling through the upper corridor represents a much greater risk of collision to an agent.

The objective of MCC is, as its name suggests, to find the route through the graph with the lowest collision cost, though this is not necessarily the route with the lowest probability of collision. Risk is traded against edge cost as discussed below. Next, we describe the MCC adaptations to PRM.

3.2.3.1 Sampling

Standard PRM uniform sampling generates a random pose p, then uses a collision checker to verify p lies in C_{free}, rejecting it otherwise. As explained in Section 2.1.6.1 this leads to insufficient roadmap coverage in narrow areas such as corridors, driving the development of specialised algorithms such as Gaussian sampling. In the presence of uncertainty, the MCC sampler additionally assesses the collision risk (p_{col}) of candidate poses based upon the obstacle covariances. Algorithm 3.3 shows the adapted uniform sampling for the MCC planner. The probability of p colliding with each obstacle is calculated by the `colProb` (see below) function in the main loop (line 3). P_{free} is the joint probability of not colliding

Algorithm 3.3 Adapted uniform sampling algorithm for MCC (Missiuro and Roy 2006)

```
Input: $p$ candidate pose, $W$ set of obstacles in environment
Output: accept/reject $p$ when it is tested for collision with set $W$
1: $P_{free} \Leftarrow 1$ //probability of $p$ is collision free
2: for all $w \in W$ do
3:     $P_{free} \Leftarrow P_{free} \cdot (1 - \texttt{colProb}(p, w))$ //$\texttt{colProb}(p, w)$=probability $p$ colliding with obstacle $w$
4: end for
5: $P_{col} \Leftarrow 1 - P_{free}$ //probability $p$ collides
6: return accept/reject $p$ by sampling from Bernoulli distribution with $P$(reject)=$P_{col}$
```

with any obstacle; this is inverted (line 5) to give P_{col}. Pose p is finally accepted or rejected based on the outcome of a sample from a Bernoulli distribution in the last line. A high probability of collision for pose p gives it a low probability of being accepted. A Bernoulli distribution is used instead of a simpler acceptance criterion such as a preset threshold so that poses with a high P_{col} may still be accepted occasionally. This allows the sampler to still place nodes in risky areas of the C-space.

3.2.3.2 Query phase

Adapted sampling biases PRM against placing graph vertices in places with a high chance of collision, such as the upper corridor in Figure 3.1, to encourage safer routes. This alone does not guarantee a low collision cost route, so MCC also considers uncertainty in the query stage. A modified edge cost equation is used:

$$P_{col}(e) = 1 - \prod_{w \in W} (1 - \texttt{edgeColProb}(e, w)) \tag{3.1}$$

$$c_e^{MCC} = P_{col}(e) \cdot C_{const} + (1 - P_{col}(e)) \cdot c_e \tag{3.2}$$

where $P_{col}(e)$ is the collision probability with obstacle set W while traversing edge e, $\texttt{edgeColProb}(e, w)$ is the collision probability of e with respect to obstacle w and c_e^{MCC} and c_e are the MCC and standard edge costs respectively. C_{const} is the problem dependent constant cost of collision. This is a tunable parameter that governs how aggressive or conservative the planned route should be. $P_{col}(e)$ determines the weighting between c_e and C_{const} in Equation (3.2); edges with high collision probability become less attractive as C_{const} contributes more to c_e^{MCC}.

3.2.3.3 Calculating `colProb` and `edgeColProb`

$\texttt{colProb}(p, w)$ is the collision probability of a robot in pose p colliding with obstacle w. Strictly, this is the integral of pose p colliding with w over all possible positions of w.

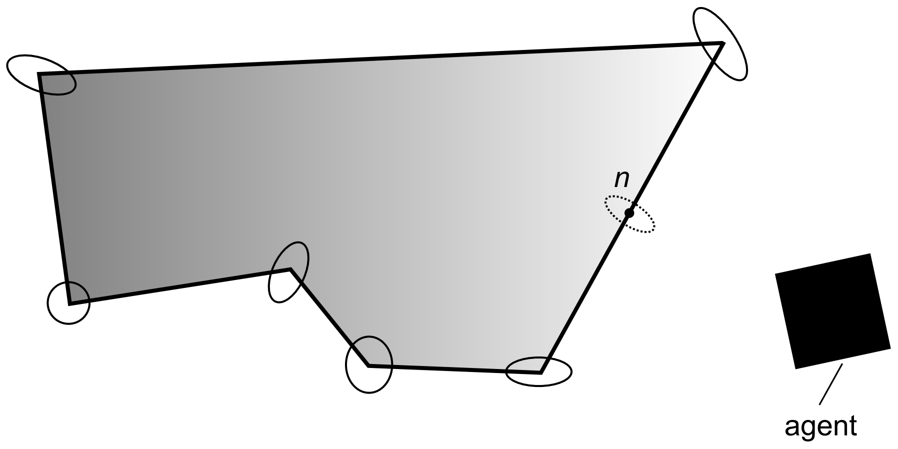


Figure 3.2: Locating the nearest point of an obstacle. Point n is the nearest point on the obstacle to the agent. The covariance matrix of n (shown as dotted ellipse) is interpolated from the covariances of the end points of the line on which n resides.

This cannot be computed exactly because w has an infinite number of positions arising from the Gaussian distributions giving a continuous space of locations for each component vertex of w. Missiuro and Roy (2006) describe an approximation for `colProb`(p, w) by finding the single nearest point on the nearest line in w to pose p, and then estimating the probability that the robot collides with this point, since this dominates the probability of collision with w. Figure 3.2 shows an example of this where point n is nearest to the agent. As the obstacle's location is uncertain, n is represented with a Gaussian distribution. This distribution has a mean centred on the nearest point and a covariance matrix computed from the covariances of the adjacent obstacle vertices as shown in Figure 3.2. In our implementation of MCC, we use a Monte Carlo simulation method to approximate `colProb`(p, w) since the returned probabilities will approximate the integral. Our algorithm first finds the nearest line section of w to p via a Euclidean distance measure. Multiple samples of this single line are generated from the Gaussian distributions at its end points and pose p is tested for collision against each sampled line. The returned collision probability is then:

$$\texttt{colProb}(p, w) = \frac{\text{colliding samples}}{\text{total samples}}$$

As the number of samples tends to infinity, the approximate probability approaches the true probability. With sufficient samples, we have a close approximation to the true collision probability. To aid computation speed, if p is further than a preset distance from w or is found to be completely contained within w (for closed objects), then sampling is skipped and `colProb`(p, w) returns 0 or 1 as appropriate.

The `edgeColProb`(e, w) function is computed using a similar idea to edge collision detection in basic PRM. Recall that collision detection for an edge is achieved by subdividing the edge into a chain of points. Individual collision checks are performed as if the agent was positioned at each point along the edge. Applying the same logic, the edge is subdivided into a set of points P. The collision probability is then maximised over the edge:

$$\texttt{edgeColProb}(e, w) = \max_{\texttt{colProb}(p,w)} p \in P$$

3.2.3.4 Overview

Once the pre-processing phase is complete, MCC can compute plans with minimal computation owing its direct use of PRM as opposed to using it as a step in a larger algorithm. The query phase uses an unmodified A* search that theoretically provides solutions in the same time frame as normal PRM. In practise, computing `edgeColProb` for many edges lengthens the computation somewhat, due to repeated Monte Carlo sampling of wall distributions. To mitigate this, we cache the calculated probabilities so no edge needs to be calculated more than once. As an offline planner, MCC does not replan during plan execution and also does not consider sensor input, thus the produced routes are static. If the agent receives further information while executing a plan it will be ignored, for instance if an object is found to be truly blocking an edge the agent will simply fail to reach the goal. Finding a good heuristic for determining a sensible value for C_{const} is an open issue. While experimental data could indicate a "sweet spot" for particular classes of map or robot, we observe that even a change in obstacle density may require re-tuning the parameter.

3.3 Markov Decision Processes

A significant body of work relating to motion planning and uncertainty has examined the applicability of modelling these domains as MDPs. The MDP framework is a natural candidate for computing plans with uncertain action outcomes since Markov chains are non-deterministic by design. While computational requirements have limited the use of MDP solution algorithms to MDPs with smaller state spaces, advances in asynchronous dynamic programming algorithms have increased the size of MDP that can be realistically handled.

3.3.1 Asynchronous dynamic programming

Standard DP algorithms such as value iteration and policy iteration are synchronous in nature; each state in the MDP is systematically backed up (updated) once per iteration.

The order is undefined and theoretically all DP updates may happen in parallel on a machine with $|\mathcal{S}|$ processors. The succeeding value function estimate $\hat{V}^*_{t+1}$, is based on the current estimate $\hat{V}^*_t$. In practice, this can be implemented with two arrays of size $|\mathcal{S}|$, one containing the state values for the previous iteration, with the second getting updated to hold the new estimates during the iteration, with the roles reversing each iteration.

Asynchronous dynamic programming relaxes the constraint of updating states once per iteration: state costs may be updated in any order, at any time. This also permits the updating of states at different frequencies, e.g. state i may update multiple times in between updates to j, $i \neq j$. With the requirement that an algorithm must still allow all states to update infinitely often, convergence on an optimal policy is still guaranteed (Bertsekas 1982; Bertsekas and Tsitsiklis 1989). The simplest, most easily implemented form of asynchronous DP is the Gauss-Seidel method (Barrett et al. 1994, Chapter 2). Strictly speaking, this is not a true asynchronous DP algorithm because it still performs systematic sweeps of the state space, updating each state once per iteration. However, it exploits the fact that the state updates do not need to occur in a lockstep fashion. Instead of maintaining the current and next value function estimate in two distinct arrays, a single array is used. State updates occur in place, using the current cost estimate of successor states from the same array. This can be seen in Algorithm 2.2, line 6. During one iteration of value iteration (VI), the algorithm is using a mixture of updated costs from the current iteration and the previous iteration. The Gauss-Seidel method should not merely be seen as a consequence of efficient implementation (saving extra memory allocation), as it will generally converge faster to V^* because all state updates use the most recent values for other states.

3.3.1.1 Real-time dynamic programming

Real-time DP (RTDP) (Barto et al. 1993) focuses VI on states reachable from a given start state by exploiting belief state reachability. RTDP organises state updates by repeatedly conducting "trials" over the state space. Each trial is initialised by setting the current system state to the start state. Assuming a given value function (this may be pre-seeded with a heuristic estimate from a partially computed value function), a trial step consists of performing a single backup on the current state (i.e. performing line 6 of Algorithm 2.2) and then selecting the current best action (line 7). Executing that action causes a transition from the current state to a new state according to the probability distribution in the transition matrix. This cycle repeats until an absorbing state is reached or a maximum number of steps have occurred. Multiple trials ensure RTDP evaluates all necessary parts of the state space due to the stochastic nature of transitions. Once RTDP has converged, the policy can be extracted in the standard way by greedily selecting the action in each state that maximises the future reward as in Equation (2.3).

Using trials to find states reachable from the start state is efficient since, if we know that a subset of states in $\mathcal{S}$ will never be visited, they never need to be evaluated. Importantly, the use of RTDP does not preclude the requirement of DP convergence that all states are visited infinitely often; RTDP will still converge with probability one. States not reachable from the start state can be ignored since they have zero impact on the policy; all other states will be visited with a non-zero probability. Thus, in infinite trials, each state will be visited infinitely often, fulfilling the convergence requirement.

Compared to LAO*, a key difference with RTDP is that the probability distribution of the transition matrix affects how often states are updated. The trial based design causes states with a higher probability of occurring to be updated more often. States that occur with a low probability must still be capable of being updated to ensure convergence, even if in practice they do not occur in a finite run of trials. When a node is expanded in the best solution graph (BSG) of LAO*, all successor nodes are generated, and successors on the best action arc are members of the BSG. LAO* state updates are agnostic to the probabilities of those states occurring. Results for the racetrack domain (Barto et al. 1993, Section 4.1) in Hansen and Zilberstein (2001, p.57) show this has two effects. Firstly, both RTDP and LAO* achieve the same eventual reward from a given start state, but RTDP improves its reward faster. Secondly, LAO* converges to an ϵ-optimal solution an order of magnitude faster than RTDP. The authors attribute both of these to the fact RTDP spends more time updating high-probability states.

3.3.1.2 Envelope MDP

Envelope solution methods (Dean et al. 1995) enhance the DP algorithm by only updating a subset of MDP states. Unlike other asynchronous DP algorithms, updates on a subset of states (the *envelope*) are conducted systematically via policy iteration or value iteration. The size of the envelope is expanded until it contains enough states for the value function to converge. To guarantee finding an optimal policy for any MDP under this algorithm, the entire state space must still be evaluated. Transitions to states outside the current envelope are represented by an explicit "OUT" state. *Fringe states* are those states that are outside the envelope, but connected to an envelope state by one transition. The algorithm proceeds with two (or more) interleaved phases of *envelope alteration* and *policy generation*. Policy generation updates the value function estimate and associated policy. Envelope alteration is the process of expanding the boundary of the envelope to include new states. An advantage of this class of algorithm is that it may be terminated at any time prior to convergence and still provide a policy (foregoing the guarantee of optimality). A set of reflex actions specify the agent behaviour in states outside the current envelope.

The envelope can be extended via different strategies. One strategy is to add the N most likely fringe states to the envelope. These are found via a Monte-Carlo approximation technique: the MDP is simulated forwards from states in the envelope to reach the fringe states. The probability of reaching the various fringe states are recorded so the N most likely can be found. Dean et al. also describe "Recurrent-deliberation" models, where the planner runs concurrently with the agent executing the current plan. Under these models, states may be removed from the envelope for efficiency. In navigation experiments (Dean et al. 1995, p.36), envelope methods performed substantially better than RTDP, particularly in set-ups where the agent's motion actions contained moderate levels of noise (e.g. a 30% chance a move action would not move in the correct direction).

3.3.1.3 Prioritised sweeping

The central theme in asynchronous DP is that an intelligent scheduling of state value backups results in faster convergence than naïve systematic sweeps of the state space. Dai and Hansen (2007) compare the performance of different algorithms, including LAO* under the name of Forward Value Iteration (FVI, to distinguish it from Backwards Value Iteration which searches backwards from the goal), in terms of the convergence time and number of states expanded. Ten example domains are compared with the largest having a state space size of 490,000.

Particularly relevant to this research is the result that FVI performs the best out of all the algorithms tested on 8 out of 10 domains, converging about three times faster than the second quickest algorithm. Only FVI and one other algorithm (also forward searching) exploit state reachability (see Section 2.2.3); the others either compute full state space policies or a partial policy for every state that may reach the goal. FVI only computes a policy for states that can be reached by searching forward from the start state.

The notable result is that using a sub-optimal order for state updates is generally preferable to computing the optimal order. Prioritised sweeping (Moore and Atkeson 1993) approaches order the backups using a priority queue. The position of a state in the queue reflects a heuristic estimate of the potential that backing it up has to improve its values. Dai and Hansen conclude that the benefit in convergence time resulting from an intelligent ordering is outweighed by the cost of maintaining the queue. Insertions and deletions happen in $O(\log n)$ time and queue algorithms also perform many more backups than queue free methods such as FVI.

3.3.2 MDP motion planning

3.3.2.1 Stochastic Motion Roadmaps

The *Stochastic Motion Roadmap* (SMR) planner (Alterovitz et al. 2007) is a motion planner for robots with uncertainty in their motor control that combines PRM and MDP techniques. The SMR allows for the generation of motion plans that account for uncertainty in robot motion, with the goal of maximising the probability of success. The motivation is that in some agents with constrained movement, a deviation from the desired path may result in future failure. For example, a slipped wheel near the beginning of a route may cause the agent to inadvertently alter its course, but not to detect the error until it has reached an unrecoverable position. The robot is assumed to have a finite number of control actions available and the planner assumes complete knowledge of both the environment and the robot's motion model. Sensor uncertainty is not considered in this work. The algorithm is divided into a roadmap construction phase and a query phase, as with standard PRM. Graph nodes are sampled using standard PRM methods, but a separate set of edges is generated for each of the robot's actions. For each node v and each action a, m samples are generated of the robot's possible configuration after executing a from v. Each of the m generated samples is then mapped back to its nearest node in the graph, which then gives a probability distribution over a finite number of successor locations when executing a from v.

For the query phase, the SMR is converted to a standard MDP. The set of PRM graph nodes, with an additional dedicated collision state, form the MDP state space ($\mathcal{S}$), and the MDP action set ($\mathcal{A}$) is the set of robot actions. Instead of being based on path length, all rewards in $\mathcal{R}$ are equal, but subject to a small penalty for each transition. The transition matrix is determined by the edge sets above. From any graph node (state), executing an action leads to a finite set of successor nodes, where the probabilities were generated by the sampling described above. The MDP can then be solved via standard DP methods to produce a policy that should always favour the route with maximal probability of success.

The SMR technique is successfully demonstrated in 2D using a simple two-action car that moves forward while either turning left or right (known as a Dubins' movement model) to simulate a flexible, steerable needle (Webster III et al. 2005) common in certain medical fields (e.g. drug treatment delivery or biopsy) (Alterovitz et al. 2005). The number of states (200,000) is quite substantial for the reasonably simple environment described, but the policies produced have high chances of success, often greater than 75%. While the work takes a different approach to Missiuro and Roy (2006) and considers motion uncertainty as opposed to map uncertainty, the behaviour produced is very similar. Both algorithms' preference can be tuned towards shorter routes or "safer" routes. An advantage of the SMR technique over the MCC (see Section 3.2.3) planner is that the route is not fixed at execution time, because the MDP policy will generate an action for each node the robot may reach.

3.3.2.2 Grid decomposition

A different approach to MDP motion planning by Burlet et al. (2004) uses quad-tree decomposition to model a 2D environment for a mobile robot as an occupancy grid. Each cell can subdivided into 4 equal sub-cells, then labelled as either "full", "free" or "mixed" to indicate the presence of an obstacle in the cell. The given map is discretised using recursive quad-tree decomposition. A robot state is then a $\langle c, o \rangle$ tuple where $c \in C$ is the set of cells (of varying size), and o is the finite set of robot orientations. Actions are based on a Dubin's model of motion as in SMR above, except multiple motions are concatenated to make a single action. Rewards are -1 for all actions unless they lead to a goal state, where the reward is 0 to encourage the agent to find the shortest path. Transitions are calculated from the probability of reaching a particular cell, having executed a specific action from the previous cell as follows. A Gaussian distribution is centred on the desired destination cell for a particular action with a covariance matrix that depends on the length and complexity of the action's motion. Any cells the agent may reach with a probability higher than ϵ by executing the specific action (according to the Gaussian distribution) are considered successors in the transition function. Plans are computed by solving the MDP using value iteration.

MDPs suffer the drawback that policy generation times are polynomial in the size of the state space and action space (a full state space sweep of standard VI has a complexity of $O(|\mathcal{A}||\mathcal{S}|^2)$). For a rigorous analysis of MDP complexity, see Littman et al. (1995b). Asynchronous DP methods are one attempt to mitigate this, but another approach is to reduce the state space size. In grid world navigation problems, the simple approach of using a lower resolution grid may not allow a sufficiently detailed plan, as too much detail may be removed. Hierarchic decomposition avoids this as described above by dynamically increasing/decreasing the grid resolution depending on the required detail in that area. Grid decomposition offers a large advantage in state space size over standard grid worlds by keeping cells large in sparse regions on the map. However, sampling based methods such as those based on PRM, do not need this type of discretisation at all, thus offering an inherent advantage. Motion plans in grid worlds produce smooth plans since each action causes the robot to traverse between the centres of grid cells while accounting for uncertainty. Example plans produced show some detours that could be revised as the robot executes the plan, much in the way that smoothing techniques can be applied to PRM generated motion plans.

3.3.3 Information space planning

Using dynamic programming to generate motion plans under uncertainty in an information state planner is considered by Barraquand and Ferbach (1995). The aim of the planner is to plot the route from the start to the goal with the highest probability of success (not necessarily to find the shortest route). The planner assumes a completely known environment but non-deterministic robot motion and limited sensing so there is also uncertainty in the robot's location. The information state for an agent is the combination of all the domain knowledge the robot needs to function. In a navigation domain, this typically includes it's current location estimate and what it has observed through its sensors. The information space is the space of all possible information states. A DP algorithm is used to back-propagate the utilities of various points (states) in the information space from the goal to the initial state. Experiments on a robot in a 2D grid environment with discrete control inputs show the algorithm is able to achieve a probability of success of 60-70% in cases where there is a small amount of noise in the robot motion (80% probability of executing correct action). The demonstrated environments are sparsely populated and some include "landmark" areas wherein agent has perfect knowledge of its location. The discrete grid keeps the size of the information space finite, but the curse of dimensionality is still a major obstacle, precluding the use of more complex scenarios. This weakness in their system is acknowledged:

> *"Indeed, the DP method requires a memory space and a computation time exponential in the dimension of the information space, which is often much larger than the configuration space [of the agent]"*
>
> (p.1)
>
> *"A severe limitation... is the fact that the dimension[ality] of the information space increases at each new sensing operation."*
>
> (Barraquand and Ferbach 1995, p.4)

State aggregation is used to increase the algorithm's scalability. This partitions the information space into separate disjoint sections, treating each as one state. The second quotation above is referring to the fact that the reachable region of the information space increases with every sensing operation the agent makes. Due to the chosen model for experiments, this increases with every action the robot takes, as the state must be updated to include the current location estimate. Unfortunately this means that the reachable space grows as a function of distance from the goal as more grid cells must be traversed. The methods described in this research are distance agnostic because the algorithms do not assume a grid based environment, thus the reachable state space only expands when an observation is made by the agent.

3.3.4 Symbolic LAO*

The application of state aggregation to LAO* is first used in the Symbolic LAO* algorithm (SLAO*) (Feng and Hansen 2002). Combining a reduction in state space with belief-state reachability attacks the state explosion problem from two angles. The principal difference in SLAO* compared to LAO* is that it handles sets of states instead of individual states. SLAO* represents the MDP transition and reward functions and performs all computations through the use of Algebraic Decision Diagrams (ADDs) (Bahar et al. 1993; Hoey et al. 1999). A full description of ADDs is beyond the scope of this research since they are used as a state abstraction technique (Dearden and Boutilier 1997) for representing factored MDPs. Briefly, decision diagrams are a data structure that can be used to efficiently represent a mapping from a set of state variables to a finite set of values, where the state variables collectively define the state space of the factored MDP. Each state variable takes a boolean value where the MDP state space $\mathcal{S}$, is the set of all possible instantiations of those variables. The reader is referred to Bahar et al. (1993) for a full explanation.

SLAO* modifies the LAO* algorithm to deal with ADDs. In SLAO*, it is not necessary to keep an explicit search graph; only a list of expanded states is tracked. In the policy expansion phase of SLAO*, reachability analysis is performed to identify states that are reachable but have not yet been expanded. This is the equivalent of expanding the leaf nodes of the BSG in LAO*. The DP step and convergence testing phase in SLAO* are the same as in LAO* with some modifications to handle ADDs.

3.4 Partially Observable Markov Decision Processes

Approximate solution methods to POMDPs (see Section 2.3.3.2) have received much attention in recent years due to their ability to handle problems on a real world scale. They do not look for the precise optimal value function over the entire belief simplex (the belief space $\mathcal{B}$), but instead compute the optimal value function for some subsection (usually a set of specific belief points) of $\mathcal{B}$. Importantly, the value function is still defined over the entire space, as varying forms of interpolation are used to determine the value and best action for all belief points not explicitly defined by the solution algorithm.

3.4.1 Perseus: a point-based POMDP solver

Point-based value iteration (PBVI) Pineau et al. (2003) was one of the earliest point-based POMDP solvers that avoided the use of interpolation when defining a complete value function. Earlier approaches (Lovejoy 1991; Brafman 1997; Zhang and Zhang 2001) only compute the value function at specific points in the belief space, but all those approaches choose the belief points according to some discretisation scheme. Also at each belief point, only the value is stored. The value at other points is calculated by interpolating (the exact method is algorithm dependent) the value of nearby, known belief points. Best actions are selected in a similar manner. PBVI and its variants store the optimal hyperplane (α-vector) instead of the value of that particular belief point. This gives a major advantage in extracting the optimal policy, since interpolation is never necessary; the set of hyperplanes define a convex hull over the entire belief space $\mathcal{B}$. When backing up a belief point x that is not at a specific grid point or finding the best action for plan execution, interpolation is quite a costly function since it will typically involve numerous distance calculations from x to existing grid points. In point-based algorithms, the use of hyperplanes obviates the need to interpolate; the set of α-vectors from the belief set defines the value function V, so value backup and action selection continue using the same technique as in exact value iteration (see Section 2.3.2.1).

The nature of the problems we consider in this research can produce POMDPs with large state spaces. Although pre-processing techniques are used to minimise the number of states, the models are still intractable to exact solvers. In light of this, we use a recent point-based solver, PERSEUS, that builds on previous point-based methods and has been shown (Spaan and Vlassis 2005) to solve larger state spaces in feasible time, solving problems in a matter of hours not days.

Point-based solvers can be characterised by the method for selecting belief points to include in the set of beliefs B. The set of points chosen must be distributed well enough to adequately represent the POMDP value function, yet also be compact enough to minimise computation time. PBVI switches between expanding the set of belief points B, and carrying out value backups on those points (see Section 2.3.3.2). As the algorithm progresses the size of B increases and successive stages of value backups require more computation.

PERSEUS (Vlassis and Spaan 2004; Spaan and Vlassis 2004; 2005) is a point-based solver which differs from previous solvers by keeping B constant throughout the computation. Value backups are carried out in such a way that each backup is guaranteed to improve the value of at least one point in B, therefore improving the value at all points in the belief space by the end of each iteration.

The PERSEUS solver is shown in Algorithm 3.4. The algorithm initialises by simulating the POMDP belief state forward from the start state s_0, randomly selecting actions and observations to build the pool of belief states B. The initial value function V_0, is a singleton set containing one belief vector with all elements set identically (line 2). This ensures the initial value function is uniformly improvable. PERSEUS then enters its backup loop which

Algorithm 3.4 PERSEUS algorithm, Spaan and Vlassis (2005)

```
Input: $\epsilon$ maximum error bound between two iterations, $n$ number of belief states to sample, $s_0$ the POMDP start belief state
Output: $V_t$ optimum $t$ step value function
1: Randomly explore environment from $s_0$ to collect $n$ belief points for set $B$
2: $V_0 \Leftarrow$ single $\alpha$-vector with all components set to $\frac{1}{1-\gamma} \min_{s,a} r(s,a)$
3: $t \Leftarrow 0$
4: repeat //PERSEUS backup loop
5:    $V_{t+1} \Leftarrow \emptyset$
6:    $\tilde{B} \Leftarrow$ copy of $B$
7:    while $\tilde{B} \neq \emptyset$ do
8:       $b$ =randomly sampled point from $\tilde{B}$
9:       $a = \texttt{backup}(b, V_t)$
10:      if $b \cdot a \geq V_n(b)$ then
11:         $V_{t+1} \Leftarrow V_{t+1} \cup a$
12:      else
13:         $V_{t+1} \Leftarrow V_{t+1} \cup \text{argmax}_{z \in \{a_t\}} b \cdot z$ //$a_t$ is the set of $\alpha$-vectors for horizon $t$
14:      end if
15:      $\tilde{B} \Leftarrow \{b \in B : V_{t+1}(b) < V_t(b)\}$
16:   end while
17:   $t \Leftarrow t + 1$
18: until $\max_{b \in B}(|V_t(b) - V_{t-1}(b)|) < \epsilon$
19: return $V_t$: $\epsilon$-optimal $t$ step value function
```

iteratively generates value functions for increasing horizons (t). The set of hyperplanes for the succeeding value function always starts as an empty set (line 5). The loop operates by selecting a random belief point b from $\tilde{B}$ (line 8) and performing a value backup on that point, where $\tilde{B}$ is the set of belief points whose values have not been improved this iteration. This creates an updated α-vector a, for that belief point (line 9). If the new α-vector improves the value at b, then it is added to the new value function, otherwise the best α-vector at b from the previous iteration's value function is added (lines 10 to 14). This ensures that the new vector added to V_{t+1} improves the value of at least one point in $\tilde{B}$—a necessary condition to ensure termination. Line 15 removes all $b \in \tilde{B}$ from $\tilde{B}$ which are improved by a (or whichever vector was selected from V_t) which reduces the number of unimproved belief points in $\tilde{B}$. When no points remain in $\tilde{B}$, V_{t+1} is complete and a new iteration begins. This process repeats until convergence. Different convergence metrics are available, but here we demonstrate the common criterion of a maximal bound on the Bellman residual (line 18).

The main contributions of the PERSEUS algorithm are, firstly that only a small subset of belief points need to be updated to improve the value at all belief points, thus finding a globally better value function. Secondly, updating the α-vector for any b always improves the value for at least one point, keeping the value function compact by only containing the minimal number of vectors and allowing for fast backups. The cost of the `backup`(b, V_t) operator is linearly dependent on the number of vectors in V_t. Spaan and Vlassis note that one backup operation often removes large numbers of belief points from the set $\tilde{B}$ in the early stages of value iteration. As with other POMDP value iteration algorithms, the precise size of V_t fluctuates as t increases.

3.4.1.1 Symbolic Perseus

The use of factored state spaces to reduce the intractability of large (PO)MDPs is an approach that has received increased attention in the literature in recent years. The state space has to capture all the important features of the environment. In many cases, a complete system state can be described by the state of each feature. If each feature can be in any one of a set of discrete states, then $\mathcal{S}$ is the cross product of all these features. Each combination of feature states is a system state in $\mathcal{S}$. In a factored MDP/ POMDP, it is assumed that the state space can be decomposed into a set of variables $X = \{x_1, x_2, \ldots, x_n\}$ where each $x_i \in X$ has domain D_i. $\mathcal{S}$ is the space of possible instantiations of each of the n variables:

$$\mathcal{S} = D_1 \times D_2 \times \ldots \times D_N$$

Actions in the system have a (non-deterministic) effect on some subset of these variables, thus changing the state of the system. Actions can be described through Conditional Probability Tables (CPTs) which show the post-action states of the variables and their relationship to their parent variables (the pre-action states) on which they depend. CPTs are useful, but disadvantageous in practice because they explicitly show the result of a particular variable for each possible instantiation of its parent variables. This leads to very large memory requirements for their storage and a large amount of redundancy, as many separate instantiations yield the same result.

Using Algebraic Decision Diagrams (ADDs) (Bahar et al. 1993; Hoey et al. 2000) avoids both of these problems by more compactly representing the relationships between pre and post action variable states. ADDs are a generalisation of Binary Decision Diagrams and take the form of a mapping from n multi-valued variables to a finite set of real values. They are usually described as a directed, acyclic tree with the variable names at the nodes and the outgoing edges labelled with possible values for those variables. The tree leaves are the real valued outcomes of the mapping. ADDs can represent mappings for many variables very compactly since only the variables that affect the outcome appear in the tree and redundancy is avoided by merging identical sub-branches together. This makes them ideal for use in factored domains.

The use of ADDs to describe each action in a system allows a complete transition matrix to be entirely described using ADDs. The reward matrix can be similarly described (see Hoey et al. (1999) for details). As such, a solver that can work with factored MDPs described through ADDs could produce policies for MDPs with very large state spaces. SPUDD (Hoey et al. 2000) is such a solver and uses ADDs for all components of the MDP including the value function. In Hoey et al. (2000), the authors describe how to perform standard value iteration using ADDs.

In his Ph.D. thesis, Poupart (2005) showed how the PERSEUS POMDP solver could be adapted to work with factored POMDPs using an ADD representation. Symbolic PERSEUS (Poupart 2005, page 100) uses ADDs to represent the α-vectors and belief states in factored POMDPs. Poupart notes that *"Since* PERSEUS *is very close to classic value iteration, the integration described in those papers [Hoey et al. 1999; Hansen and Feng 2000] can be applied directly to* PERSEUS.*"* Symbolic PERSEUS exploits the conditional independence of the different variables in factored POMDPs to factor the belief state in an intuitive way. Instead of having a single probability distribution over the entire state space, independent probability distributions are used for each variable. Each distribution shows the likelihood of the possible values for one variable. This allows a belief state to be easily represented as an ADD. One further modification that Symbolic PERSEUS makes is to apply an approximation to the ADDs representing α-vectors of the POMDP value function. Similar values in the α-vector are aggregated together to help bound the size of the ADD. In Chapter 4 we will test Symbolic PERSEUS on some problems from our domain.

3.4.1.2 Continuous POMDP

Continuous space POMDPs have not received as much attention in the literature as discrete models despite their suitability to certain domains such as robot navigation. Discretisation of the robot's orientation (heading) with a Cartesian grid for its location forms the standard approach to creating a finite environment for a robot, leading to the disadvantages already discussed (see Section 2.3.3). Further, each additional degree of freedom causes an exponential expansion in the state space size. Continuous state spaces seem initially attractive by removing the requirements for discretisation and methods for estimating the required resolution. They can represent the agent's location and orientation exactly (modulus hardware numerical errors). The difficulty in translating this to the POMDP framework stems from the infinite individual states possible in the continuous state space. A standard POMDP belief state (a probability mass function (p.m.f.) over the discrete set of states) must become a continuous probability distribution function (p.d.f.) over the infinite dimension belief space. The value function α-vectors can no longer be represented with finite dimensions (∞ dimension vectors would be required).

Continuous Perseus (Porta et al. 2005) is a generalization of Algorithm 3.4 to continuous state spaces. In the value function, α-vectors are replaced by α-functions which approximate the value function over the entire belief space. Belief points are represented through combinations of Gaussian distributions (Porta et al. 2005, part IV-A):

$$b(s) = \sum_j w_j \phi(s|s_j, \Sigma_j)$$

where $b(s)$ is the belief that s is the true state in the continuous state space, s_j and Σ_j are the respective mean and covariance matrix of Guassian ϕ, and w_j are the weightings for the j Gaussians, summing to unity. Linear combinations of Gaussians are used to approximate the true state value since the exact integral over the belief space $\mathcal{B}$, cannot be computed in closed form. The continuous POMDP value function does however retain several of the properties of the discrete model. The continuous space value function is still PWLC over $\mathcal{B}$ and importantly, a recursive computation of the next horizon is still a contraction operation, thus convergence through dynamic programming is still guaranteed. Proofs can be found in Porta et al. (2005). The general Perseus algorithm requires no changes at the highest level to handle continuous spaces because its operation does not depend on the nature of the `backup` operator in line 9 of Algorithm 3.4 as long as the value at b improves. One limitation is that the number of components in the Gaussian mixtures increases with the horizon distance t, a natural consequence of the approximation. In simple perceptual aliasing experiments, a compression technique was used that reduces the number of components in the Gaussian mixtures while retaining the nature of the distributions, keeping computation time within reasonable bounds. Unfortunately no comparisons to other techniques (such as conventional discretisation) are provided to objectively evaluate performance.

3.4.2 Online POMDP solvers

Approximate techniques increased the scale of POMDPs that could be solved by an order of magnitude. For instance, the “Tag” domain (Pineau et al. 2003) with 870 states is now considered to be well within tractable bounds. The exact and approximate techniques discussed so far are all offline algorithms; much like traditional planners, a policy is computed given a model definition, then handed off to the executing agent. This burdens the planner with the requirement of finding every state the agent may encounter in order to devise a complete plan. This is intractable for very large problems.[1] Recent POMDP planners take an online approach where planning is concomitant with execution, either alternating between planning and execution after each step, or allowing n steps of execution before updating the plan. The planner starts from the current belief state and searches forward, creating a tree of reachable belief states where the tree depth is limited by computation time. This limits the size of the search tree the planner must generate, but sacrifices optimality as the values of leaf states in the tree are not exactly known, with them being assigned heuristic values instead. Before introducing a very recent online POMDP planner, two techniques that bridge the gap between offline and online planners are described.

3.4.2.1 QMDP

Although an offline planner by design, QMDP (Littman et al. 1995a) employs a form of approximation. A knowledge of this is useful for understanding the heuristic nature of online algorithms. Conceptually, POMDP algorithms conduct value iteration over the continuous space of belief points as in a standard MDP. In an MDP, we define the value (expected reward) through Q-functions. $Q(s, a)$ is the value of executing action a from state s then acting optimally. The value of state s is then found by greedily selecting the action with the highest Q-value:

$$V(s) = \max_{a \in \mathcal{A}} \; Q(s, a)$$

This is Equation (2.9) restated in terms of Q-values for convenience (see Section 2.2.2.3 for full details). The explicitly stated goal for QMDP is to "to find an approximation of the Q function over the continuous space of belief states" (Littman et al. 1995a, Section 2.2). This approximation is computed by solving the underlying MDP, found by taking the $\mathcal{S}$,$\mathcal{A}$,$\mathcal{T}$ and $\mathcal{R}$ components and disregarding the observation functions. The Q-functions for this MDP can be found using any standard MDP algorithm as detailed in Section 2.2.2.3. Given V^*, then

$$Q_{\text{MDP}}(s, a) = r(s, a) + \gamma \sum_{s'} t(s, a, s') \cdot V^*(s') \qquad \forall s, s' \in \mathcal{S}, a \in \mathcal{A} \tag{3.3}$$

Having computed V^*, these can be quickly computed for all $\langle s, a \rangle$ tuples; the value for any belief state b can then be estimated according to:

$$Q_a(b) = \sum_{s} b(s) \cdot Q_{\text{MDP}}(s, a) \tag{3.4}$$

[1]As with most technologies, the precise value of "very large" changes with time.

where $b(s)$ is the probability of state s being the true system state according to belief state b (defined in Section 2.3.1). Equations (3.3) and (3.4) define the QMDP value function, since they define the value for all $b \in \mathcal{B}$. Two often discussed features of the QMDP algorithm have serious consequences for the resultant policies. Firstly, approximating the POMDP value function with MDP Q-functions generates policies that assume the agent will know the state exactly after one transition, i.e. all uncertainty is removed. Secondly, the agent will never attempt information gathering actions such as sensing operations. While not a major issue in small domains, this leads to significant performance degradation in larger problems where sensing actions are required for optimal behaviour. Performance of the QMDP algorithm is generally poor compared to other algorithms, however its fast speed and intuitive simplicity mean it is often used as a baseline for comparison.

3.4.2.2 Heuristic search value iteration

POMDP value functions represented through α-vector sets represent the PWLC lower bound on the expected reward across the belief simplex. The agent controller selects actions associated with this lower bound. The true value for the lower bound of each belief state is approached as value iteration progresses. Backing up one belief point is a relatively straightforward computation. Most POMDP solvers therefore focus solely on this computation. Heuristic Search Value Iteration (HSVI) (Smith and Simmons 2004) differs by additionally storing the upper bound for the value function. This permits a different convergence criterion to most value iteration algorithms. Instead of measuring the Bellman residual between successive horizons, HSVI terminates when

$$|\overline{V}(b_0) - \underline{V}(b_0)| < \epsilon \tag{3.5}$$

where b_0 is the starting belief state, $\overline{V}(b)$ and $\underline{V}(b)$ are the upper and lower bounds on the value of belief state b respectively. Equation (3.5) is known as the *width* of V. When a policy π is not guaranteed to be optimal, the *regret* of a state is the difference in the value of that state under π and its value under an optimal policy π^*:

$$\text{regret}(b, \pi) = |V^*(b) - V^{\pi}(b)| \tag{3.6}$$

When HSVI terminates, the regret is guaranteed to be less than or equal to ϵ at b_0:

$$\text{regret}(b_0, \pi) \leq \text{width}(V(b_0)) \leq \epsilon$$

HSVI expands the search tree in a depth-first manner, following a single path down the tree until the width of an expanded belief node is below a threshold based on ϵ and the depth of the node. Action and observation selection is based on a forward search heuristic that selects actions and observations that are most likely to reduce width(b_0) the greatest amount. When the width of an expanded node is below the threshold, the upper and lower bounds are updated back up the branch to b_0. This exploration routine is repeated until width(b_0)$\leq \epsilon$.

The lower bound $\underline{V}$, uses a standard vector set representation common in other algorithms. It is initialised with a single vector of uniform components exactly as in Perseus (line 2 of Algorithm 3.4). The upper bound $\overline{V}$, uses a point set representation, similar to the way in which early point-based solvers such as Lovejoy (1991) stored the value of belief points. The point set forms a convex hull over $\mathcal{B}$ on to which belief points can be projected using linear programming techniques to obtain their upper bound.

Compared to other contemporary algorithms, HSVI matches or exceeds the reward obtained in POMDP benchmark problems such as "Tag" (Pineau et al. 2003) and "Hallway" (Littman et al. 1995a), though it is significantly slower to compute the policy. The most likely cause is the significant overhead imposed by solving the linear programs necessary to update $\overline{V}$ which dominates the runtime in smaller POMDPs. In larger problems with thousands of states, this cost is proportionally reduced so does not impede algorithm scalability. Compared to PBVI, HSVI obtains substantial speed-ups in larger models while still deriving policies of identical quality (as measured by the returned value function).

3.4.2.3 Anytime error minimisation search

Unlike the QMDP or HSVI solvers, Anytime error minimisation search (AEMS) (Ross et al. 2007) is a true online POMDP solver. The planner constantly replans after every action is executed to determine the subsequent action. It shares characteristics with both of the former algorithms. Heuristics are employed to estimate the value and decide the best actions to expand in the tree. QMDP can be seen as a tree search where the tree is only ever expanded one level ignoring observations. HSVI and AEMS can expand the tree to an arbitrary depth, both storing the upper and lower bounds of the value function to estimate the true value of a state. AEMS uses the estimated error at the tree root b_0, to determine whether the algorithm may terminate, i.e. if the root value is ϵ-optimal. Unlike HSVI, AEMS only stores values at the belief points in its tree; no vector sets or point sets are used to store the value function bounds. This is necessary in HSVI so it can find the value of any belief point in $\mathcal{B}$, a restriction AEMS does not have.

AEMS builds a tree out from b_0 as an AND/OR tree, where OR nodes represent an action choice and AND nodes represent the possible observations. Loops are not explicitly dealt with, so duplicate belief states can appear in different sub-trees. The tree is expanded in a best-first order as with HSVI, but expansion is not depth first. Any leaf node on the tree may be expanded, but that path will not necessarily be followed until the upper and lower bounds are minimised. Upper and lower bounds are propagated back through the tree from leaves to ancestors; however, as the name suggests, the goal of the tree expansion is to minimise the error in value, not to find the action that maximises the upper bound as in HSVI. A provable property is that as a tree is expanded, the error at b_0 reduces and will become ϵ-optimal after a finite number of expansions. If the value function is already known, then the precise contribution of each fringe node to the error at the route node can be quantified, making selection of the largest contributing fringe node trivial. Obviously this is not possible in practice, so AEMS uses heuristics to estimate the contribution of fringe node errors, where $\hat{e}(b)$ is the estimated error at fringe node b.

The best performing heuristic, known as AEMS2, selects the action at node i that maximises the upper bound estimate on the value of the sub-tree rooted at i (similar to HSVI). Expansion of such fringe nodes reduces the upper bound on the value function at b_0. As noted above, the lower bound of a state's value defines the value function estimate, and as $\underline{V}(b) \leq V^*(b)$, the true state error $e(b) = |V^*(b) - \underline{V}(b)|$. However, this cannot be exactly known so the width defined in Equation (3.5) is used in AEMS to estimate $e(b)$. Expanding action arcs that maximise the upper bound are therefore likely to lead to the largest reduction in the estimate $\hat{e}(b)$.

Empirically, the performance of an online variant of HSVI and AEMS2 are very similar in terms of both policy reward and computation time across multiple domains. This is partially attributable to the many similarities in their design. HSVI exists in both on and off-line versions, making it applicable to a wider range of problems. Following recent trends in POMDP solvers, an anytime modification to HSVI is available making it feasible for use in time constrained environments (Smith and Simmons 2004, Section 3.5). The reader is referred to Ross et al. (2008) for further discussion of online POMDP solvers including those discussed above.

3.5 Concluding Remarks

Uncertainty can be integrated into planners using a variety of techniques and the literature includes many more not discussed here. Numerous classifications are possible, but uncertain route planners can be broadly divided according to the type(s) of uncertainty they tolerate or by the objective of the planner. As noisy odometry is a common feature in mobile robotics, many planners such as SMR and Censi et al. (2008) compute plans that do not depend on the robot executing perfect actions. These planners tend to use obstacles to localise and reduce uncertainty and produce plans that succeed even if the robot deviates from the path. Other planners, such as FSSN and MCC, plan for map uncertainty assuming the robot acts deterministically, or that lower level mechanisms correct motion deviations; they compute plans that specifically avoid obstacles.

A different classification highlights other similarities and differences between planning algorithms. Most of the planners reviewed here such as SMR and Censi et al. (2008), attempt to produce plans that are safe under all conditions, i.e. the route computed has the highest probability of reaching the goal. MCC offers the weaker condition that the route is optimal given the estimated cost of collision, giving the user the option to allow a greater chance of collision in domains where such collisions are not serious (e.g. a vacuuming robot knocking into a table leg). This research, in similarity with FSSN, aims to produce plans where no collision is encountered by using observational data to avoid traversing graph edges that are known to be blocked. Table 3.1 shows the applicable techniques from this review grouped according to category.

Planner	Uncertainty type	Probabilistically safe
Hsu et al. (2000)	sensor	yes/no[1]
Predictive PRM	sensor	yes
MCC	sensor	no
FSSN	sensor	yes
Barraquand and Ferbach (1995)	sensor/motor	yes
SMR	motor	yes
Burlet et al. (2004)	motor	yes
Censi et al. (2008)	motor	yes

Table 3.1: A comparison of uncertain planners by category.

In terms of technical implementation, algorithms can generally be split into two groups: those that use decision-theoretic models to plan for all (or most) circumstances and those which are heuristic based and plan solely on what is currently known. This research falls into the first category along with the information space planner of Barraquand and Ferbach (1995) and Burlet et al. (2004) and many POMDP based motion planners. FSSN and MCC are examples of the second category.

A certain motion model with uncertain obstacles versus uncertain motion with an exact environment should not be viewed as two versions of the same problem.[2] In the first case, this reasoning would assume that all obstacles share the same degree of uncertainty, thus removing uncertainty for one would remove uncertainty for all of them (turning it into a purely localisation issue). However, obstacles are often independent of each other, so

[1]Agent can sometimes require local domain knowledge to avoid collisions.

[2]A case of "Am I moving, or is the room moving around me?"

updating the position of one does not help infer the position of all others.

The main thrust of this review is that many categories of route planner remain the subject of intense research. Finding the right planning algorithm still requires careful examination of the problem domain—a single dominant plan system has yet to emerge. Therefore, we anticipate the already diverse range of literature in this field will continue to evolve at a fast pace, particularly with the renewed commercial interest in the use of application specific domestic robots.

Chapter 4

Solving via MDP

4.1 Problem Definition

In this Chapter, we show how the problem of PRM graph traversal in uncertain graphs can be formulated as a decision problem based on the agent's beliefs about the world. We abstract it into a problem of which edge should chosen based on the accumulation of information gathered from observations. These decisions are made offline, i.e. we would like the agent to know beforehand what it should do no matter what it finds out about its environment (through observations of edges).

We assume that we are given a model of the robot and its environment as input. The robot's size and shape is specified as a 2D polygon and is used during roadmap construction for collision detection. The environment is described via its bounding shape (a closed 2D polygon) along with desired start and goal positions. Obstacles in the environment are described as 2D polygons in a similar fashion to the robot. The difference is that the robot geometry is known exactly, whereas the obstacles are not. Each vertex of the polygon defining an obstacle is represented as a bi-variate Gaussian distribution with the mean representing the most likely location. The covariance matrix then defines the uncertainty over the true location of the point represented visually as an ellipse around the vertex. An example of this can be seen in Figure 3.1 on page 55. The output is a policy that directs the agent from the start to the goal accounting for the uncertainty present about the environment. The policy instructs the agent how to act based on what it currently believes about the edges in the roadmap. To simplify some details of the problem, we do not deal with the problem of localisation and assume the robot always knows its precise location and also assume that its actions are deterministic.

The first stage of the algorithm is to construct the roadmap for the environment. The basic version of PRM is used (shown on page 17) to generate the graph using the known robot geometry and obstacles. Since the standard version of PRM does not handle uncertainty, the Gaussian distribution means are assumed to be the correct locations for the obstacle vertices. We use a uniform sampling algorithm that places PRM graph nodes uniform randomly over the environment, discarding any that fall in C_{obst}. All graph edges are also collision free according to the maximum likelihood obstacle vertex positions. Given that the obstacles in the world are unlikely to have certain locations, this entails that some of the edges in the graph are not guaranteed to be free of obstruction. Uncertainty about an obstacle's location (or even part of that obstacle) may mean an agent may not be able to traverse an edge. This complicates the planning process because the ostensibly simple task of plotting a route across a graph becomes a problem of identifying which parts of the graph are usable. The core dilemma for the agent is then one of risk-analysis: when is a shorter, possibly blocked route worth taking? In reality, it will face several choices and must always balance the trade-off between a long safe route and a short, uncertain route.

We abstract the problem of route planning in an uncertain PRM graph into a model identification problem and assume that the state of some of the edges in the PRM graph will not be known precisely before the agent starts to act (i.e. they are uncertain edges according to Definition 1.1). If the agent tries to traverse an uncertain edge that is blocked, it will be considered to have failed to reach the goal node of the graph. The agent will only fail if it tries to traverse an uncertain edge that is actually blocked. It is acceptable behaviour for the agent to reach a graph node incident to a blocked edge, observe the edge is blocked and then move away on an alternate route. Identifying the optimal route to take across a PRM graph in the presence of these uncertain edges is the core problem of this thesis. Conceptually, each uncertain edge in the graph is either usable or not: it is *free* or *blocked*.

Definition 4.1 (Free Status)**.** The true state of an uncertain edge is either "free" or "blocked". This is referred to as the "free status" of that edge.

Given m uncertain edges which may be blocked, there are 2^m combinations of edges that are obstacle-free or blocked. Part of the problem therefore, is to deduce which combination—or *world*—is the correct one based on the observations of edges that the agent receives. The decision over which route to take is then not only based on the agent's belief about which edges are usable, but also upon the value of being able to make better observations about obstacles in the world. The agent must trade off the cost of having to backtrack if a short route is indeed blocked against the cost of taking a longer route to start with. However, if there is a location in between the two where an observation of the short risky route can be made, the agent must also consider this third alternative. This third option would allow it to reduce the uncertainty about its environment and therefore make a more informed choice as to which route to the goal it should pick.

At various nodes throughout the graph, the agent can make observations of specific uncertain edges. The agent may receive observations of multiple edges when it arrives at a node. Each observation that the agent makes of an edge will result in it receiving a "free" or "blocked" observation. In most cases, these observations are uncertain in that if the edge is collision-free, we may still observe blocked some of the time, and vice versa. The exception is at the two end-nodes of an uncertain edge, where we assume the agent may make a perfect observation of that edge, i.e. the free status of that edge is accurately known (it may still receive uncertain observations of other, non-incident edges). The graph nodes where observations are received are explicitly stated to the planner, along with the probabilities of receiving each of the two observations (free or blocked). The agent knows these probabilities a priori (in practice these can be calculated either by sampling or as a function of distance of the observing node from the obstacle location). For each observation of an uncertain edge, the agent requires two values: $P(\text{blocked}|\text{blocked})$ and $P(\text{blocked}|\text{free})$—these are the probabilities of receiving a blocked observation given that the edge is truly blocked or free. The complement probabilities can be calculated as follows, since they must sum to unity:

$$P(\text{free}|\text{blocked}) = 1 - P(\text{blocked}|\text{blocked})$$
$$P(\text{free}|\text{free}) = 1 - P(\text{blocked}|\text{free})$$

4.1.1 Model formulation

The PRM graph the agent moves in can be described as follows: Let $G_{\text{PRM}} = \{V, E\}$ be a weighted graph where the nodes in the graph $V = \{v_1, \ldots, v_n\}$, represent locations in the world, and the edges $E = \{e_1, \ldots, e_k\}$ where $e_i = \langle v, v' \rangle$, represent paths between locations (we assume that if there is a path $\langle v, v' \rangle$, then there is a corresponding path $\langle v', v \rangle$). We also identify locations v_S and v_G, the start and goal locations. Each edge e has an associated cost (edge weight) c_e of traversing the edge in either direction. Some of the edges are uncertain because we do not know if the agent can traverse them, whilst the remaining edges are certain.

Although we cannot observe it directly, there is in fact a true state of the world in which each uncertain edge $\{1, \ldots, m\}$ is either collision-free or blocked. Let $W = \{w_1, \ldots, w_{2^m}\}$ be the set of all such worlds. While we have abstracted away the obstacle locations in this representation, the fact that the traversability of the edges is based on obstacle locations is important because it implies that for close edges, the probability that the edges are blocked may not be independent. If the edges are independent, then the likelihood that world w is the true state of the world is simply the product of the likelihoods of the free status of each edge. If this is not true (for example, where one obstacle is likely to intersect two edges), then the edge probabilities are dependent (see Section 4.4.1).

4.2 POMDP Representation

The agent has to act in an environment where it never has complete access to the state, but can alter its belief about the true state when it receives an observation. The size of the set of worlds W, is finite and precisely known and the set of available actions is also finite so it is convenient to define the navigation problem within the POMDP framework.

Every uncertain edge can either be free or blocked giving 2^m possible worlds. We define a POMDP as a distribution over these worlds. If the correct world state is known exactly, route planning becomes straightforward. The uncertainty arises from the POMDP deciding which world is correct. Since there is no uncertainty over the agent's location: it always knows which node in the PRM graph it is at, this knowledge must be included in the system state. To translate our world identification problem into a POMDP, we make a copy of the PRM graph for each possible world $w \in W$. Therefore, in the POMDP state space $\mathcal{S}$, there is one state per graph node per world. Within each world, there is no uncertainty about the edge statuses. $\mathcal{S}$ is the union of the states from each world and one absorbing termination state s_T, so

$$|\mathcal{S}| = n2^m + 1 \tag{4.1}$$

for a PRM graph of n nodes and m uncertain edges. An example is shown in Section 4.2.2. The action set is the set of nodes in the PRM graph so there is one action per node. Each action indicates that the agent should travel to that node. This implies that most actions are invalid in most states because the current node must connect directly to the desired node for an action to be usable.

We define the function valid(v, v', w) where $w \in W$, to represent the fact that there is a collision-free edge in world w between v and v'. Formally, valid(v, v', w) is true if $\exists e \in E : e = \langle v, v' \rangle$ and e is collision-free in w.

The POMDP model consists of state space $\mathcal{S}$, action set $\mathcal{A}$, transition matrix $\mathcal{T}$, reward function $\mathcal{R}$, observation set $\mathcal{O}$ and observation function $o(z, s, a)$ where $z \in \mathcal{O}$. For full details refer to Section 2.3. We can now formally define the model identification POMDP as follows:

- $\mathcal{S} = V \times W \cup s_T$ as described above. Each state is a combined $\langle v \in V, w \in W \rangle$ pair.
- $\mathcal{A} = \{a_1, \ldots, a_n\}$ where a_i represents the action instructing the agent to traverse the edge from the current node to node v_i.

- $\mathcal{T}$ is defined as follows:

  All transitions have probability zero except transitions where the agent is moving between PRM graph nodes in the same world and there is a collision-free edge in that world between those two nodes. Recall that in reality, the agent is in a particular system state in one world and cannot change between worlds: it is the *belief* about which world is correct that changes. If $v = v_G$ then all actions transition to the absorbing state s_T which only contains a self-transition for all actions.

$$t(s,a,s') = P(\langle v,w\rangle, \text{goto-}v', \langle v',w'\rangle) = \begin{cases} 1 & \text{if } w = w' \text{ and valid}(v,v',w) \\ & \text{and } v \neq v_G \\ 1 & v = v_G \text{ or } s_T \text{ and } v' = s_T \\ 0 & \text{otherwise} \end{cases}$$

- $\mathcal{R}$ is defined by the graph structure. Each reward is the negative cost of the edge $\langle v, v'\rangle$ if valid(v,v',w) is true. Reward $c_G > 0$ is obtained when transitioning from a goal node to s_T.

$$r(s,a) = r(\langle v,w\rangle, \text{goto-}v') = \begin{cases} -c_e & \text{if } e = \langle v,v'\rangle, \text{ valid}(v,v',w) \\ & \text{and } v \neq v_G \\ c_G & \text{if } v = v_G \text{ and } v' = s_T \\ 0 & \text{otherwise} \end{cases}$$

- $\mathcal{O} = \mathcal{P}(\bar{o}) = W$ where $\bar{o} = \langle o_1, o_2, \ldots, o_m\rangle$ and o_i is an observation of uncertain edge i. Each observation $\bar{o}$ includes an observation of all m uncertain edges. As each $w \in W$ comprises one combination of free and blocked edges, then the set of observations equals the set of worlds.

- $o(z,s,a)$ is defined as follows:

$$o(z,s,a) = P(z|s,a) = P(\bar{o}|\langle v,w\rangle, \text{goto-}v') = P(\bar{o}|v,w)$$

  where $z = \bar{o}$, $s = \langle v,w\rangle$ and $a \in \mathcal{A}$ as described above. $o(z,s,a)$ is the observation function for the POMDP which, in this case, is defined solely by the resulting state. This is the probability of seeing observation $\bar{o}$ given that the agent is at node v in world w and executed action a to transition to node v'. Whenever we perform an action in the POMDP, we get an observation of whether each uncertain edge is collision-free or blocked. From most nodes, most edges cannot be observed

Algorithm 4.1 Calculation of $P(\bar{o}|v,w)$

```
Input: $\bar{o}$ observation, node $v$ and world $w$
Output: Calculation of $P(\bar{o}|v,w)$ based on observation probabilities present in graph
 1: $L \Leftarrow$ {all uncertain edge observations at node $v$}  //each observation comprises
    probabilities $P(\text{blocked}|\text{blocked})$ and $P(\text{blocked}|\text{free})$ of an uncertain edge
 2: $p \Leftarrow 1$
 3: for $i = 1, \ldots, m$ uncertain edges do
 4:   if edge $i \notin L$ then  //$i$ not observable from $v$
 5:     $p \Leftarrow p/2$
 6:   else
 7:     if $i$ free in world $w$ then  //choose probability based on edge $i$ actually being free
 8:       $o \Leftarrow P(\text{blocked}|\text{free})$  //lookup probability from $L$
 9:     else
10:       $o \Leftarrow P(\text{blocked}|\text{blocked})$
11:     end if
12:     if $i$ is observed as free in $\bar{o}$ then  //is edge $i$ being observed as free or blocked?
13:       $p \Leftarrow p(1-o)$
14:     else
15:       $p \Leftarrow p \cdot o$
16:     end if
17:   end if
18: end for
19: return $P(\bar{o}|v,w) = p$
```

so the probability of observing "free" or "blocked" for those edges will be evenly distributed. At nodes where no edge observations can be made, then

$$P(\bar{o}|v,w) = \frac{1}{|W|} \quad \forall \bar{o} \in \mathcal{O}$$

otherwise $P(\bar{o}|v,w)$ is the product of the edge observation probabilities $P(\text{blocked}|\text{blocked})$ and $P(\text{blocked}|\text{free})$ for the edges observable from node v. An algorithm for the calculation of $P(\bar{o}|v,w)$ given node v and world w is shown in Algorithm 4.1.

4.2.1 Belief state

The POMDP belief state is a probability distribution over all the states in the system; however, it is important to remember that the agent's current location (the current graph node) is always known exactly. The uncertainty concerns which world is correct. When the agent is at node v_i in the graph, the belief state is defined as:

$$b(s = \langle v, w \rangle) = \begin{cases} P(w) & \text{if } v = v_i \\ 0 & \text{otherwise} \end{cases}$$

That is, all system states pertaining to graph nodes other than v_i have probability zero. Thus, most states will have zero probability in any particular belief state. In reality, each edge will either be free or blocked, so each possible world must be enumerated. We can refer to a specific world through the free statuses of the edges, e.g. in a two uncertain edge graph, FF refers to the world where both edges are free.

Assume a particular graph has three uncertain edges labelled A, B and C. We can obtain the marginal probability of edge A, B or C being free or blocked by summing the $P(w)$ where the edge status matches, for instance:

$$P(\text{A}{=}\text{free}) = \sum_{w \in Z} P(w) \quad Z = \{\text{all } w \in W : \text{A is free in } w\} \tag{4.2}$$

The belief state combines two important pieces of information: the knowledge of how likely various edges are to be free and also how various edges are correlated to each other. Combined with the Markovian property of the process, i.e. the agent does not need any information other than the current state to act optimally, this entails that the starting belief state alone is sufficient to encode any edge dependencies. If the probabilities that two edges are blocked are not independent, this can be represented in the POMDP model using the belief state. We can represent the fact that, for example, two edges will always have the same status (free or blocked) by making all worlds in which one is blocked and the other is not have prior probability zero. If two edges are independent, then the initial belief state should be such that the sum probability of the worlds in which both edges are blocked is the product of the marginal probabilities that each edge is blocked. In a two uncertain edge graph, if $P(\text{free}) = 0.4$ and 0.3 for the two edges respectively, then the probability of the FF world is 0.12. This POMDP model can now be solved to find an optimal policy under the uncertainty about which world is correct. However, this also illustrates the problems that exist with this model. Despite the fact that this POMDP solution is intuitively the correct formulation since it is the closest match to the actual problem we are trying to solve, it suffers from the computational complexities that affect all non-trivial POMDPs. Even simple PRM graphs with a small number of uncertain edges translate into POMDP models with large numbers of states and high-dimensional belief spaces. The state space grows rapidly even with low numbers of uncertain edges because each additional edge doubles the size of $\mathcal{S}$ (ignoring s_T).

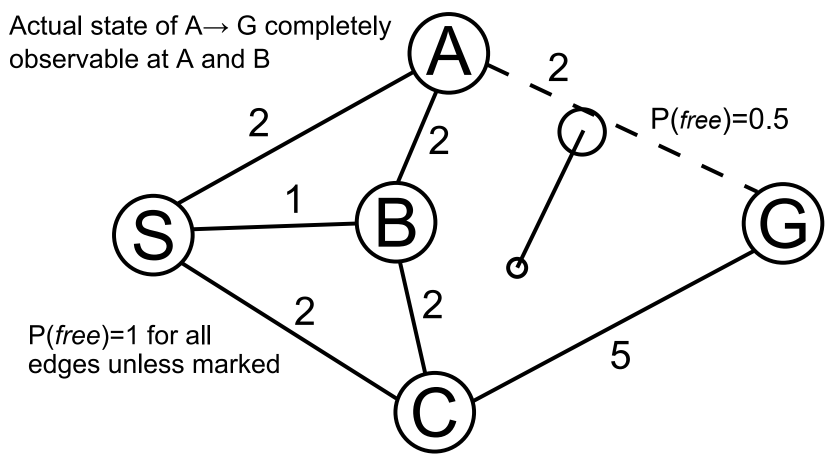

Actual state of A→ G completely observable at A and B
A
2
2
2
P(free)=0.5
1
S
B
G
2
2
P(free)=1 for all edges unless marked
5
C

Figure 4.1: A simple 5 node roadmap with one known wall present. The agent starts from node S and must attempt to reach node G. The dotted line shows the uncertain edge.

4.2.1.1 Implementation note

There is an alternative, more attractive set up for the POMDP formulation where $c_G = 0$ and all rewards are negative. As the POMDP solver should always find the policy with the highest expected reward, this should produce the same results as the model described above that includes a positive reward for reaching the goal state. Even though it is not the most satisfactory set up, we added the goal reward for practical reasons because we found that the solver we used did not produce usable results unless the POMDPs were modelled as laid out above. With larger examples, a goal reward was necessary to ensure that reaching the goal always yielded the lowest eventual cost under the presence of discounting. Without it, we found that in most cases, the agent did not move towards the goal because cost would accrue more slowly if it repeatedly traversed shorter edges in graph. In an undiscounted problem, repeatedly paying a small cost would eventually become more expensive than reaching the goal as the horizon extends, with the latter behaviour becoming the optimal policy. However, with the discount factor limiting how much future rewards are considered, the agent is not guaranteed to do this.

4.2.2 Example POMDP model of a graph

We now present a worked example of converting a small PRM graph into a POMDP model. The graph shown in Figure 4.1 has 5 nodes and 1 uncertain edge so there are only two worlds: w_1 where edge $A \rightarrow G$ is free and w_2 where it is blocked. The agent occupies one state from a space of 11 states (including the absorbing state). The complete POMDP state space is shown in Figure 4.2. $\mathcal{A}$ contains the 5 actions corresponding to the nodes. $\mathcal{O}$ consists of two observations: "F" and "B" as there is only one uncertain

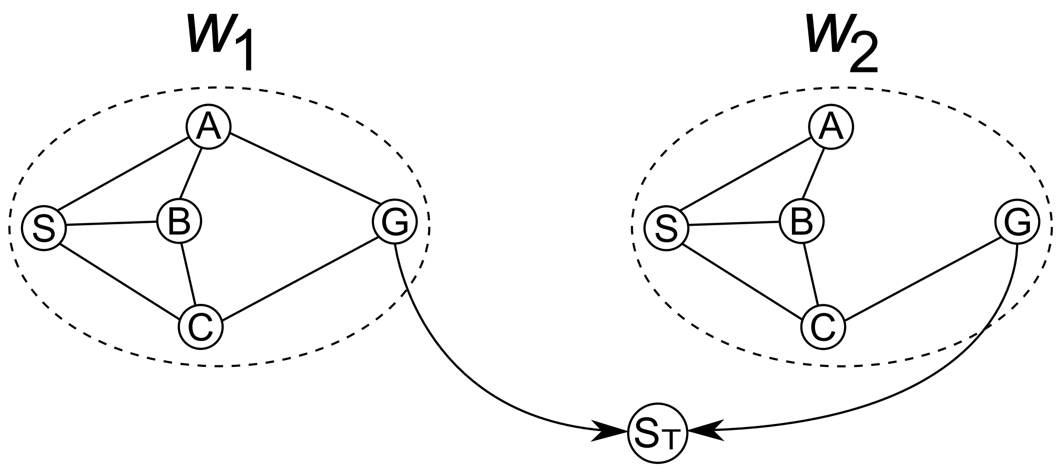


Figure 4.2: The POMDP state space for the graph in Figure 4.1. All transitions are deterministic in the state space and only transitions between nodes in the same world are allowed. The goal states transition to the absorbing state s_T.

edge in this case. Each state represents a particular $\langle v, w \rangle$ pair, so the transition matrix contains the transitions appropriate for the given node and world. For instance, in world w_1, the transition function for node A allows transitions to S, B and G with probability 1 under the appropriate action and 0 otherwise, whereas in world w_2, only transitions to nodes S and B are allowed. Trying to execute an invalid option, e.g. attempting to reach G from node A in world w_2, leaves the agent at the same node with probability 1. Most of the nodes in this graph produce no observation about the uncertain edge, so in those cases the two POMDP observations are received with an equal probability of 0.5. For the purposes of this example, the agent makes a perfect observation of edge $A \rightarrow G$ at nodes A and B, so one POMDP observation is received with probability 1 while the other is never received, depending on the world. The agent makes any observations at a node when it arrives there. There is no actual cost associated with making an observation. The reward function simply contains the negative of the edge costs that are shown in the figure and c_G is set at 10 for this graph as this is higher than the maximum cost of reaching the goal from the start.

The starting belief state gives the agent no priori knowledge about which world is correct, assigning a probability of 0.5 to states $\langle S, w_1 \rangle$ and $\langle S, w_2 \rangle$. Solving this POMDP can be accomplished very easily because it is a very small example. In Section 4.6.1.1 we show that an exact POMDP solver can solve it in 97ms but takes much longer on graphs that are only slightly more complex.

In the example in Figure 4.1, the uncertainty in the graph could easily be avoided by adding further nodes to the PRM graph that avoid the obstacle. Another PRM node near to the $A \rightarrow G$ edge may avoid the obstacle entirely and become the obvious path to the goal. During the query phase of PRM planning, some planners attempt repair strategies based on re-sampling nodes around the obstacle (details can be found in Section 3.1) to avoid any uncertainty. Whilst we would expect this to work in this example, it is not a general solution to our problem. Standard PRM is not an uncertain planner and assumes it has complete, accurate information about its environment. Therefore, when it samples nodes in the pre-processing phase, if it is possible to place nodes and edges around an obstacle that avoid collisions, it is reasonable to expect PRM to have done so. With regards to POMDP planning, each time a PRM node is added to the graph, this change would need to be reflected in the POMDP model specification. While this may be possible in an online planner, it would require the planning process to be re-started in offline planners due to the modified state space, transition model and introduction of new actions.

4.3 MDP Representation

The POMDP formulation is clearly not scalable because the cardinality of $\mathcal{S}$ is exponential in the number of uncertain edges and POMDP solution algorithms traditionally scale very poorly to larger state spaces. We would like to find ways to mitigate the state space explosion and still compute policies in a reasonable amount of time. A common technique is to convert a POMDP into a belief state MDP. Each POMDP belief state is treated as a completely observable MDP state. The transition function is governed by the belief update rule of the POMDP shown in Equation (2.12). MDP value iteration has a lower complexity than POMDP value iteration, so if the number of belief states forming the belief MDP state space is manageable, the optimal policy over those states can be found more efficiently than in the POMDP formulation. If a precise list of belief states visited by a POMDP controller under a (near) optimal policy were available, these belief states would form the MDP state space. Performing standard MDP value iteration over these states would yield their optimal value and a policy which could be applied to the POMDP controller without needing a more complex POMDP solution algorithm.[1] The major obstacle in such an approach is that the resulting MDP state space is infinite due to the continuous nature of the belief space. In our domain, there is a lot of structure in the POMDP state space and belief space. For instance, because the agent always knows which PRM node it is located at, only POMDP states representing that node have a non-zero probability in the current belief state. In this Section we show how this can exploited to convert the POMDP formulation into an MDP formulation with a much more compact

[1]Many POMDP solvers work in the MDP belief space, so solving the belief MDP is now considered a basis for POMDP algorithms in itself.

belief state. We also show how the infinite number of belief states is kept finite for the MDP, such that we can solve it to produce a policy for the agent. This conversion forms the basis for a number of extensions that exploit the nature of this problem domain.

4.3.1 Belief state conversion

Two key conversions are required to convert the POMDP formulation into an equivalent MDP formulation. Firstly, we would like to exploit the structure in the belief space to avoid reasoning about belief states that cannot occur by the definition of the problem (such as the agent believing it is simultaneously at multiple graph nodes). Secondly, we must deal with the continuous nature of the belief space. This is done by discretising the various elements of the distribution of a belief state as explained in Section 4.3.2.

The agent's objective in the POMDP formulation is to identify which of the possible worlds represents the true state of the environment. We can therefore encapsulate all of the uncertainty in the domain by maintaining a probability distribution over the space of possible worlds, i.e. a distribution over W. Augmented with a PRM graph node, this represents the agent's total information about the environment. This forms the basis of a state in the MDP formulation. As an example, consider Figure 4.1 again: a state for the MDP formulation for that graph would consist of the current node label e.g. S, and the agent's current belief distribution over the two worlds e.g. $w_1 = 0.3, w_2 = 0.7$. We can write this as: "S-0.3,0.7" encoding the distribution as two variables. However, this is slightly inefficient, the second variable is actually superfluous as all probability mass functions must sum to unity. We can eliminate the second variable and calculate it as the remaining probability mass once all other worlds are accounted for. Thus, MDP states for the graph in the figure can be represented as: "S-0.3". In a graph with two uncertain edges ($|W| = 4$) an example belief state is "S-0.4,0.2,0.1", where $P(w_4)$=0.3 is not explicitly stated. The number of elements in the MDP belief state has been reduced by a factor of the PRM graph size n, compared to the POMDP belief state. We only maintain the distribution over the set of worlds W, instead of the space of nodes and worlds $V \times W$, so the belief state will contain $2^m - 1$ elements. The agent's location is always precisely known, so given the correct world, the true state is known deterministically.

4.3.2 Discretisation

Conversion of the POMDP belief space to a set of MDP belief states reduces the dimensionality of the distributions, but the resulting set is still continuous and infinite in size, even with only one dimension. Discretising the elements in the distribution restricts the set of belief states to a finite size. We discretise the belief state according to some granularity, or resolution d, to construct a finite belief space for the MDP. With one element in the belief state and a resolution of $d = 0.1$, each node in the PRM graph translates to 11 states in the MDP. For example, for node N we have: N-0, N-0.1,...,N-1. The "round half-up" (arithmetic rounding) rule is employed to discretise each continuous variable in a belief state independently of the others.

Definition 4.2 (Discretisation Resolution)**.** The discretisation granularity must divide into one and specifies the accuracy to which each element in the belief state is rounded.

Zero and one are important to distinguish in these problems as they lead to smaller branching factors in the MDP so we discretise each element into $d^{-1} + 1$ values as follows:

$$D = \{0, 1/d^{-1}, 2/d^{-1}, \ldots, (d^{-1} - 1)/d^{-1}, 1\}$$

where every value in the range $[i/d^{-1} - 1/(2 \cdot d^{-1}),\ i/d^{-1} + 1/(2 \cdot d^{-1}))$ is discretised to i/d^{-1}, and 0 and 1 correspond to the ranges $[0, 1/(2 \cdot d^{-1}))$ and $[1 - 1/(2 \cdot d^{-1}), 1]$ respectively.

4.3.2.1 Invalid discretisation

With a coarse discretisation, it is possible for the rounding process to discretise a valid belief state into an invalid one that no longer sums to unity. Table 4.1 shows such a belief state where a continuous distribution of 4 elements in column (a) is discretised to produce the invalid distribution (b) where the elements sum is greater than 1. We correct this behaviour by iterating over the elements of the distribution according to the following algorithm: first the elements are iterated over in order and any element that is at least $2d$ in magnitude is reduced by d. As soon as the distribution sums to 1, the algorithm terminates. This is shown in column (c) of Table 4.1. If one full iteration over the distribution finishes and it is still invalid, a second round begins with the difference that elements of magnitude d may now be decreased to 0. The distinction between the two stages is to avoid reducing distribution elements to 0 if possible because this forces the agent to believe a particular world is impossible. If a distribution sums to less than one, then the opposite procedure is performed to increase elements in increments of d until the distribution is corrected. In the first stage, only elements that are at least $2d$ less than 1 are eligible to be incremented, while all elements may be incremented in the second stage.

4.3.2.2 State space size

The MDP state space $\mathcal{S}$, is the cross product of the finite set of belief states and the nodes of the PRM graph. The size of the state space is exponential in the number of uncertain

World	(a) Continuous	(b) Discretised	(c) Corrected
FF	0.35	0.4	0.3
FB	0.25	0.3	0.2
BF	0.25	0.3	0.3
BB	0.15	0.2	0.2
$\sum$	1.0	**1.2**	1.0

Table 4.1: A continuous distribution (a) that is discretised to an invalid distribution (b) and then corrected (c). Resolution $d = 0.1$ for this example.

edges. If all possible discrete values for the elements in the distribution are enumerated, the total number of MDP states is would be:

$$n(d^{-1} + 1)^u \tag{4.3}$$

where $u = 2^m - 1$ is the number of elements in the distribution (minus the final element) and n is the size of the set of PRM nodes. However, Equation (4.3) calculates the size of state space, assuming that all possible numerical permutations of D are included, without knowledge that they represent probability mass functions. Therefore, in the belief model, Equation (4.3) overestimates the state space size as many of the distributions will be invalid. As an example, if $m = 2$, there are 4 worlds (although only the probabilities for the first 3 need to be recorded). If $d = 0.1$, systematically enumerating all possible values for each element of the belief state creates many invalid probability distributions. Table 4.2 shows such a distribution where the first three elements have been assigned values summing to 2.1, making a correct choice for the last element impossible. When *every* combination of element values is produced, most distributions are invalid—with 3 uncertain edges (7 explicit elements) this represents over 90%.

World	Probability
FF	0.7
FB	0.6
BF	0.8
BB	??

Table 4.2: An example of an invalid discretised belief state. The first three elements sum to greater than 1, making it an impossible probability distribution.

Theorem 4.1. *For a PRM graph with n nodes, m uncertain edges, and a discretisation resolution of d, the total number of valid (belief distributions summing to unity) MDP states is:*

$$|\mathcal{S}| = n\frac{(d^{-1} + 2^m - 1)!}{d^{-1}!(2^m - 1)!} \tag{4.4}$$

Proof. From Definition 4.2, given a discretisation resolution of d, we have $n = d^{-1} - 1$ discrete values that each of the $k = 2^m - 1$ elements in a belief state can be set to. Given that every belief state must be a valid probability distribution, the sum of all k elements plus the implicit final element must sum to unity. The total number of valid belief states is therefore the number of ways that the n fractions of the total probability mass can be distributed over the k belief state elements. By noting the following:

1. The order of distribution of the probability fractions is unimportant.
2. We may choose to assign multiple probability fractions (of size d) to any element.

we see that the number of distributions is the number of combinations of choosing k elements for n probability fractions, with repetition allowed. By imagining all n fractions as separate probability masses placed side by side in a line, we can arrange $k - 1$ card markers between the masses. From the start of the line, each probability fraction is added to the first element of the belief state until the first card is encountered. The probability fractions between the first and second card are added to the second belief element, and so on. An ordering of the $k - 1$ cards among the n probability masses is equivalent to a distribution of the n masses over the k elements with repetition. The number of distributions of n masses over k elements with repetition equals the number of distributions of $n + k - 1$ masses over k elements without repetition. Therefore, the number of valid belief states h, is given by:

$$\begin{aligned} h &= \binom{n+k-1}{k} \\ &= \frac{(n+k-1)!}{(n-1)!k!} \end{aligned}$$

substituting n and k:

$$\begin{aligned} h &= \frac{(d^{-1}+1+2^m-1-1)!}{(d^{-1}+1-1)!(2^m-1)!} \\ &= \frac{(d^{-1}+2^m-1)!}{d^{-1}!(2^m-1)!} \end{aligned}$$

Each belief state is possible at each of the n nodes in the PRM graph, therefore the total state space size is:

$$\begin{aligned} |\mathcal{S}| &= n \cdot h \\ &= n\frac{(d^{-1}+2^m-1)!}{d^{-1}!(2^m-1)!} \end{aligned}$$ □

Examining (4.4), we see that the major contributors to the number of states are the discretisation resolution and the number of worlds. We can coarsen the discretisation to make $\mathcal{S}$ smaller, but we tend to see instabilities in the results if we discretise too coarsely. Conversely, increasing the resolution quickly results in an intractable state space size. Graphs with 12 nodes, 4 uncertain edges and a discretisation of $d = 0.2$ have a dependent model state space size of 186,048 states (15,504 discrete belief states per node). A slightly finer discretisation of $d = 0.1$ creates an MDP of 39,225,120 states which is a massive increase for a small gain in resolution and beyond what most standard MDP solvers can handle. This highlights the trade-off of discretisation: we want to use as fine a granularity as possible to more closely approximate the continuous belief space, yet stay within the technical bounds of computation. When the discretisation resolution is too coarse, small movements in the belief space (e.g. from a poor observation) are ignored because discretisation 'moves' them back to the original belief point. This is common when $d \geq 0.1$ and will be demonstrated in Section 4.6.2.

A second influence on the choice of resolution is ensuring that it is fine enough to allow each element to have a non-zero probability. With 6 uncertain edges and $d = 0.1$, even most (valid) belief states cannot be represented. Consider the uniform belief state where each of the 64 worlds are equally likely: each element should carry a weight of $\frac{1}{64} = 0.015625$. However, when discretised, this value is rounded down to zero, creating an invalid belief state. When corrected, the distribution becomes:

$$0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0, \ldots, 0$$

Instead of believing every edge is $P(free) = 0.5$, the agent now incorrectly believes many worlds such as the final "all edges blocked" world are impossible. Here, d needs to be several orders of magnitude finer in order to be able to represent the belief state with reasonable accuracy.

4.3.2.3 Maximum discretisation error

One of the disadvantages of discretisation is that it introduces some error into the belief state due to the elimination of the continuity in the belief space; each continuous value is moved to its nearest discrete value. An important question is: what is the minimum discretisation resolution required to bound the maximum error? For example, we may require that the elements of a belief state must be no more than 0.1 away from their continuous values. With a given maximum error in the representation per element, the discretisation resolution must be at most twice that, since a discretisation level of d may only incur an error of $\frac{d}{2}$ due to half-up rounding. Over the entire belief state, the maximum cumulative error is bounded by:

$$2^m \cdot \frac{d}{2} \tag{4.5}$$

where m is the number of uncertain edges. Therefore, if the desired maximum cumulative error over the belief state is Δ, then the minimum resolution required in the dependent case is given by:

$$d \leq \frac{2\Delta}{2^m} \tag{4.6}$$

When an invalid discretisation (see Section 4.3.2.1) is produced as the result of a belief state update, then the total discretisation error can be greater than the bound given in Equation (4.5). This is due individual elements being incremented or decremented to ensure the distribution sums to unity.

4.3.2.4 Dependent belief state

Given a PRM graph with uncertain edges, the belief state can explicitly represent the probability that any particular combination of edge statuses—any world in W—is the correct one. This allows the agent to make inferences about the combinations such as that the statuses of two or more edges may be highly correlated because they are dependent on the same obstacle, hence the *dependent* belief state. The purpose of this distinction is explained in more detail in Section 4.4.1.

Definition 4.3 (Dependent Belief Space)**.** The dependent belief space is formed by the set of discretised belief distributions and is denoted $\mathcal{B}_D$. If $P(w_1)$ is the probability of w_1 being the true world, then let

$$b_D = \{P(w_1), \ldots, P(w_{2^m})\} \tag{4.7}$$

be a discretised probability distribution over W. This is a dependent belief state. The dependent belief space is then defined by

$$\mathcal{B}_D = \mathcal{P}(b_D) \tag{4.8}$$

that is, the set of all possible combinations of discrete probabilities that satisfy

$$\sum_{w \in W} P(w) = 1$$

Definition 4.4 (Dependent MDP Model)**.** When the belief states in the MDP states are defined according to Equations (4.7) and (4.8), we refer to the MDP model as a dependent MDP model.

4.3.3 Belief updating

When the agent receives an observation, it must update its belief state to reflect the new observation. All belief states are updated according to Bayes' theorem which can be stated as follows:

$$P(B|A) = \frac{P(A|B) \cdot P(B)}{P(A)} \tag{4.9}$$

where:

P(B|A) is the conditional probability (posterior) of event B given that A is known to be true.

P(A|B) is the conditional probability of event A given B is true.

P(B) is the prior (marginal) probability that event B occurs, independent of any other information.

P(A) is the prior probability of event A and also acts as the normalising constant to ensure the resulting distribution still sums to unity.

To update the belief state, we want to find the new posterior probability for each element in the belief distribution: $P(w|\text{observation})$ where $w \in W$, the set of all possible worlds. From (4.9) we have

$$P(w|\text{obs.}) = \frac{P(\text{obs.}|w) \cdot P(w)}{P(\text{obs.})} \tag{4.10}$$

The prior probability $P(w)$, is the probability of w taken from the original belief state. The likelihood $P(\text{obs.}|w)$, is the probability of receiving that particular observation given that w is the correct world. This is pre-determined since the agent knows how accurate its observations are at various graph nodes. The $P(\text{obs.})$ normalisation divisor is the prior probability of receiving that observation. This value is not stored explicitly but ensures the resulting distribution sums to one:

$$\begin{aligned} P(\text{obs.}) &= P(\text{obs.}|w_1) \cdot P(w_1) + \ldots + P(\text{obs.}|w_{2^m}) \cdot P(w_{2^m}) \\ &= \sum_{i=1}^{2^m} P(\text{obs.}|w_i) \cdot P(w_i) \end{aligned} \tag{4.11}$$

where m is the number of uncertain edges.

4.3.4 Calculation of transitions

The transition function for the MDP formulation is derived from the graph topology and the edge observations that the agent may receive. The transition function ensures the agent may only traverse between directly connected nodes by setting all other transitions to zero, as in the POMDP formulation. Since each MDP state includes the agent's belief state, edge observations must also be incorporated into the transition function. In the POMDP formulation, we receive an observation of every uncertain edge at each node, but uninformative (uniform) observations do not change the belief over which world is correct. In the MDP conversion, we can ignore these uninformative observations as they are equivalent to receiving no observation at all. In our model, the probabilities of seeing "free" or "blocked" for different edges are independent so we can treat observations of different edges separately. If no informative edge observations are received when an agent traverses to a node, then the belief state will remain unchanged so only the PRM node label in the MDP state changes and the transition function is deterministic. If one or more observations of edges are received at the new node are informative, the transition is no longer deterministic. Equations (4.10) and (4.11) comprise the fundamental calculations for updating the agent's belief state when new observations are received. The MDP transition function probabilities are calculated from the probability of making each particular observation. Next, we show an example of the set of possible transitions from a given belief state and how they are calculated.

4.3.4.1 Worked example

Assume that in a graph with two uncertain edges J and K, the agent is at node a and we have the belief state shown in the table below. We wish find the possible transitions when we take action "Goto-b" to traverse the edge $\langle a, b\rangle$ to reach node b. When describing the current world, edge J is always listed before edge K so "FB" represents the world where J is free and K is blocked.

World	**Probability**
FF	0.45
FB	0.1
BF	0.2
BB	0.25

An observation of uncertain edge J is available at node b. The observation probabilities for J at b are P(blocked|blocked)=0.7 and P(blocked|free)=0.2. We may observe "free" or "blocked" each producing a new belief state. When moving from the MDP state shown above at node a, there are two possible successor states, one for each of the new belief states with the agent located at node b. The new belief state is calculated as follows: the dividend for each posterior probability, $P(obs.|w) \cdot P(w)$, is the multiplication of the prior and the likelihood probabilities given the figures above. The normalisation factor $P(\text{obs.})$ ensures the distribution is still a valid p.m.f.

First, assume that the agent receives the "free" observation at b:

World	**Un-normalised probability**
FF	$0.8 * 0.45 = 0.36$
FB	$0.8 * 0.1 = 0.08$
BF	$0.3 * 0.2 = 0.06$
BB	$0.3 * 0.25 = 0.075$
$\sum = P(\text{obs.})$	0.575

In the first two worlds FF and FB, edge J is truly free, so the likelihood of observing "free" is $P(\text{free}|\text{free}) = 1 - P(\text{blocked}|\text{free}) = 1 - 0.2 = 0.8$. In worlds BF and BB, edge J is truly blocked so the likelihood of observing "free" is much lower: $P(\text{free}|\text{blocked}) = 1 - P(\text{blocked}|\text{blocked}) = 1 - 0.7 = 0.3$. Dividing each element by the normalisation factor $P(\text{obs.})$ gives us the one of the new belief states:

World	**Probability (3 d.p.)**
FF	0.626
FB	0.139
BF	0.104
BB	0.130

Conversely, if the agent were to receive the "blocked" observation at b:

World	**Un-normalised**	**Normalised**
FF	$0.2 * 0.45 = 0.09$	0.212
FB	$0.2 * 0.1 = 0.02$	0.047
BF	$0.7 * 0.2 = 0.14$	0.329
BB	$0.7 * 0.25 = 0.175$	0.412
$\sum$	0.425	1.0

The transition probabilities for reaching each of these states are the probabilities of receiving the observation, i.e. the normalisation factor $P(\text{obs.})$. In the MDP state space, the transition function for executing "Goto b" can be visualised in tree form as is shown in Figure 4.3.

4.3.4.2 Multiple observations

In our model, when the observations available at a node are informative about more than one edge, the number of reachable belief states increases because there is one for each combination of "free" and "blocked" received for each observed edge. As there are two possible outcomes for each observed edge, there will be 2^n reachable belief states for n observed edges. The conditional probability of a combination of observed edges is the

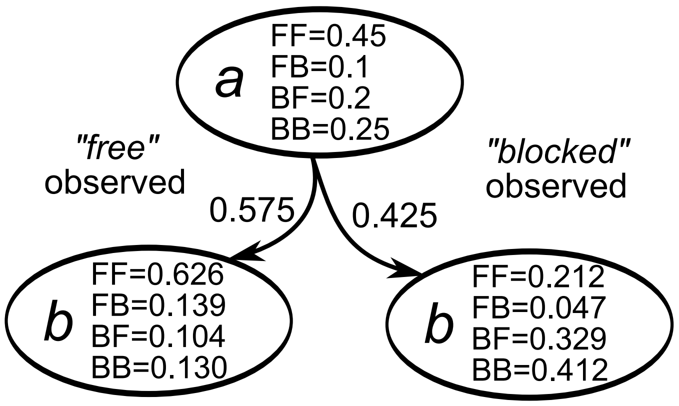


Figure 4.3: An example MDP transition from node a with the belief state shown to node b where the agent receives an observation of one uncertain edge, changing the resulting belief state. The numbers by the transition arcs show the probability of the transition.

product of the individual observation probabilities. To demonstrate this, we will extend the previous example to include a second edge observation. In addition to the observation of edge J, the agent also receives an observation of edge K at b with the probabilities $P(\text{blocked}|\text{blocked}) = 0.8$ and $P(\text{blocked}|\text{free}) = 0.4$. In total, 4 new beliefs are possible. Here we show one of the possible combinations. Starting from the same initial belief state at node a as before, the following table shows the resulting belief state for when the agent observes J as "free" and K as "blocked":

World	**Un-normalised**	**Normalised**
FF	$0.8 * 0.4 * 0.45 = 0.144$	0.493
FB	$0.8 * 0.8 * 0.1 = 0.064$	0.219
BF	$0.3 * 0.4 * 0.2 = 0.024$	0.082
BB	$0.3 * 0.8 * 0.25 = 0.06$	0.205
$\sum$	0.292	1.0

When moving from node a to b the agent will transition to this belief state with $P(J = \text{free}, K = \text{blocked}) = 0.292$.

4.3.5 MDP definition

The set of actions $\mathcal{A}$, for the MDP is the set of nodes in the PRM graph; taking an action will move the agent to that node provided there is a free edge in the graph from the current node to the one chosen. Consequently, as in the POMDP formulation, most actions are invalid in most MDP states since nodes only have edges to a few other nodes. The transition function is built according to the graph topology and observations as described above. The reward function $\mathcal{R}$ for the MDP is straight forward with the reward for each state and action being the negative cost for traversing that edge in the PRM graph.

The formal definition for the MDP conversion can now be formalised as follows:

- $\mathcal{S} = V \times \mathcal{B}_D$. Given the definition of the belief space in (4.8) the state space is the cross product of the set of nodes in the PRM graph and the set of valid dependent belief states.
- $\mathcal{A} = \{a_1, \ldots, a_n\}$ the set of PRM nodes as before.
- $\mathcal{T}$ is defined by $t(s, a, s')$ as follows:

$$t(s = \langle v, b_D \rangle, a \in \mathcal{A}, s' = \langle v', b'_D \rangle : v, v' \in V, b_D, b'_D \in \mathcal{B}_D)$$
$$= \begin{cases} 1 & \text{if } \exists e = \langle v, v' \rangle \in E \text{ and } b_D = b'_D \\ & \text{i.e. all edge observations are uniform} \\ P(\text{obs.}) & \text{if } \exists e = \langle v, v' \rangle \in E \text{ and } b_D \neq b'_D \\ & \text{i.e. one or more edge observations are informative} \\ 0 & \text{otherwise} \end{cases}$$

- $\mathcal{R}$ is defined by

$$r(s, a : s = \langle v, b \rangle, a = v') = \begin{cases} -c_e & \text{if } \exists e = \langle v, v' \rangle \in E \\ 0 & \text{otherwise} \end{cases}$$

where b is the belief state for MDP state s.

4.3.6 PRM graph example with optimal policy

4.3.6.1 Can we avoid solving the MDP?

Solving the MDP or the POMDP for the environments described here require a lot of computation. The disadvantage to the MDP formulation is that in order to find the optimal policy, the MDP state graph has to be rolled out from the start state, which as we will see in the experiments, can be very large. An attractive alternative is to try and calculate the optimal strategy by considering what the agent should do in each possible world, and act based on the probability of each world. This would avoid any dynamic programming element of the computation. If possible, this would allow the system to scale to far more complex graphs with more uncertain edges. It would also be quicker to compute the optimal policies due to the vastly reduced computational requirements. However, as we can demonstrate, this leads to sub-optimal policies.

Given an environment with uncertain obstacles, we generate a PRM graph that avoids those obstacles assuming the obstacle vertices are truly located at the means of the distributions representing them. With m uncertain edges giving 2^m possible worlds, we can

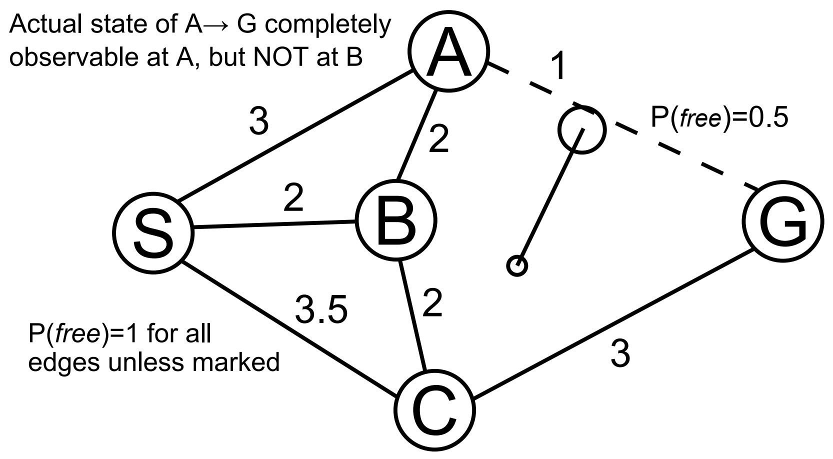

Actual state of A→ G completely observable at A, but NOT at B
A
1
3
2
P(*free*)=0.5
2
S
B
G
3.5
2
P(*free*)=1 for all edges unless marked
3
C

Figure 4.4: Adaptation of the 5-point graph (see Figure 4.1). The observation is no longer available at B and the edge costs are slightly altered.

Successor	**Cost**		**Weighted**
to S	w_1	w_2	**average**
A	4	10	7
B	5	7	6
C	6.5	6.5	6.5

Table 4.3: The calculation of utilities of successor nodes of S, based upon their utility in each world.

determine the probability distribution (the initial belief state) for the worlds by sampling obstacle positions using a Monte Carlo technique. A complete sample consists of a sampled location for all vertices for all obstacles. Each complete sample will render some of the uncertain edges blocked and will correspond to one of the 2^m worlds. By taking a sufficiently high number of complete samples, we can estimate the probability of each world occurring. This leads to a naïve algorithm for calculating what the action should be at each node, given this dependent belief state.

We can demonstrate that the above approach is ineffective, even for small examples. Consider Figure 4.4 which is an adaptation of the 5-point graph from earlier with modified edge costs and the observation at B removed. We will now attempt to find an optimal policy for this simple graph without rolling out the MDP state graph. We will assume a uniform initial belief state at S. To find the best action at S, we will compute the utility of each possible successor node in each possible world and then compute the final utility by weighting these with respect to the belief state. In Table 4.3, the quantities in the w_1 ($A \rightarrow G$ free) and w_2 ($A \rightarrow G$ blocked) columns are the costs of travelling from S to the respective node and then acting optimally in that world. As we have a uniform belief state at S, then the last column is simply the mean of the costs in both worlds. Essentially, we are computing the shortest path to the goal via each successor of S and

Successor to B	**Cost**		**Weighted average**
	w_1	w_2	
S	6	8.5	7.25
A	3	9	6
C	5	5	5

Table 4.4: The calculation of utilities of successor nodes of B, based upon their utility in each world.

then taking the weighted average those costs. For example, in world w_1, travelling via node A costs $c_{\langle S,A\rangle} + c_{\langle A,G\rangle} = 3 + 1 = 4$. Based on this, we see that travelling to node B appears to be the cheapest option as it gives the lowest expected cost to reach the goal. Now we repeat the computation at node B to select the next action in Table 4.4. At node B, travelling to node C is the apparent cheapest option. However, this is where the failure has occurred. The policy has instructed the agent to execute $S \rightarrow B \rightarrow C$ incurring a cost of 4, but the direct edge $S \rightarrow C$ only costs 3.5. Given that the agent receives no information at B or C, it will always travel to C via B, so this policy is never optimal. In this example, the true optimal policy (determined by solving the MDP) is to always travel to C first.

The reason this strategy fails is because at S, it assumes the agent will have perfect knowledge of the world at B and will therefore act optimally. This is of course not true since the agent receives no information at B. In fact, at every step the agent assumes all uncertainty is removed after one action; this is essentially the QMDP assumption, meaning this strategy will never take information gathering actions. As will be shown in Section 4.3.6.2, the optimal policy under uncertainty can be different from any optimal policy in a specific world. In the MDP models the optimal action in any state depends on the belief component of that state. As this is affected by edge observations the agent receives, these observations must be taken into account, thus assuming uncertainty is removed after one step is rarely going to find the optimal policy. Trying to compute the utility of states in this fashion cannot work because their values depend on the values of their descendant states in the MDP graph. Under uncertainty, finding the true utilities of states can only be found by rolling the MDP state graph out forwards until the uncertainty is removed and then propagating the costs backwards towards the start.

4.3.6.2 The value of information

Here we show the optimal policy for the graph from Figure 4.1 (page 84) using the MDP formulation. For the purposes of this example we assume that initially, the agent believes $P(A \rightarrow G \text{ free})$=0.5, i.e. it has a uniform prior over the free status of the edge. Once the agent reaches node A or B, it obtains perfect information about the edge. The optimal policy is easy to compute since the state space is small and we know there are only two possible outcomes of the observation at A or B. Figure 4.5 shows the entire state graph for the optimal policy. The numbers inside each node show the belief state $P(A \to G =$ free). If the agent never visits node A or B and travels to G via C, then the belief state remains at 0.5 (shown in the left section of the figure). Visiting either A or B changes the belief state as a result of the perfect observation. The middle and right columns represent travelling to G via A or B and finding edge $A \to G$ blocked or free respectively. The numbers to the right of each node show the minimum path cost to the goal from that state. As an example, if the agent is currently at state S-0.5 the optimal action is derived as follows:

1. If the agent travels to A first, then the cost to the goal will be either 2 if $A \to G$ is free, or 9 if not (as it would have to detour via C). From S, the expected cost to the goal through A is therefore:

$$\begin{aligned}\text{cost of choosing } A = \; & c_{\langle S,A\rangle} + P(w_1) \cdot \text{cost to goal in } w_1 \\ & + P(w_2) \cdot \text{cost to goal in } w_2 \\ = \; & 2 + 0.5 \cdot 2 + 0.5 \cdot 9 \\ = \; & 2 + \left(\frac{2+9}{2}\right) \\ = \; & 7.5\end{aligned}$$

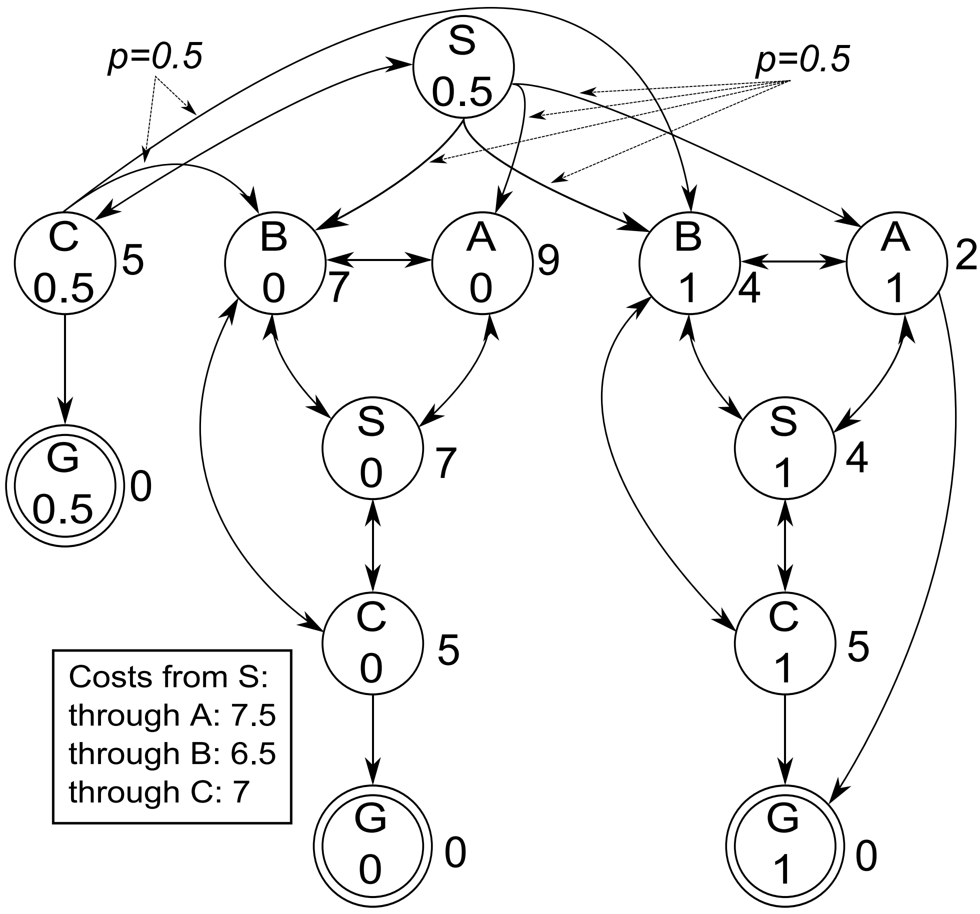
   because there is a probability of 0.5 of receiving either observation.

2. If the agent alternatively travels to B first and makes the observation, the cost to the goal will be 4 if the edge is free or 7 if blocked, so the expected cost is

$$1 + \left(\frac{4+7}{2}\right) = 6.5$$

3. Lastly if C is chosen then the cost to the goal is $5 + 2 = 7$ since no observation is received at C and the direct edge $C \to G$ is always available.

The best policy is therefore to go to B from S because that leads to the lowest expected cost. This makes intuitive sense as well because the agent obtains perfect information for a low initial cost. A key point to this example is that the optimal policy changes due to the presence of the observation at node B. If the agent was certain of the free status of edge $A \to G$ from the start and no informative observation was available at B, then the optimal policy would obviously be to travel to A or C first as appropriate. When the agent is not



Costs from S:
through A: 7.5
through B: 6.5
through C: 7

Figure 4.5: The MDP state graph for Figure 4.1 showing each possible state the agent could reach from the start state at the top. The belief $P(A \rightarrow G = \text{free})$ is shown under the node label and the minimum cost of reaching the goal is to the right.

certain, then the optimal policy would still be to travel to A or C first depending on the agent's confidence of $A \rightarrow G$ being free—travelling to B is never optimal. However, when the observation is available at B, then unless the agent is already confident of the free status of $A \rightarrow G$, travelling to B becomes optimal. The optimal policy under uncertainty is different to either of the optimal policies in the two actual worlds; the agent *must* take an information gathering action in order to act optimally.

4.4 Additional MDP Models

In this Section, we outline two additional ways to represent the state space of the MDP that exploit extra structure present in our domain.

4.4.1 Edge dependency

Edge dependency describes the principle that the free status of one edge in a roadmap is related to the free status of another edge; a relationship between them may exist due to the position of the edges relative to nearby obstacles. To examine why this is useful, consider Figure 4.6, where the location of the obstacle vertex is uncertain. The edges are close together, therefore if the upper edge is blocked it is likely the lower edge is also blocked; if the lower edge is blocked then the upper edge is guaranteed to be blocked: the free statuses of these two edges are closely related. When the agent can represent this dependency it can use that knowledge to derive better policies for acting in the world: information learned about one edge may alter its belief about others.

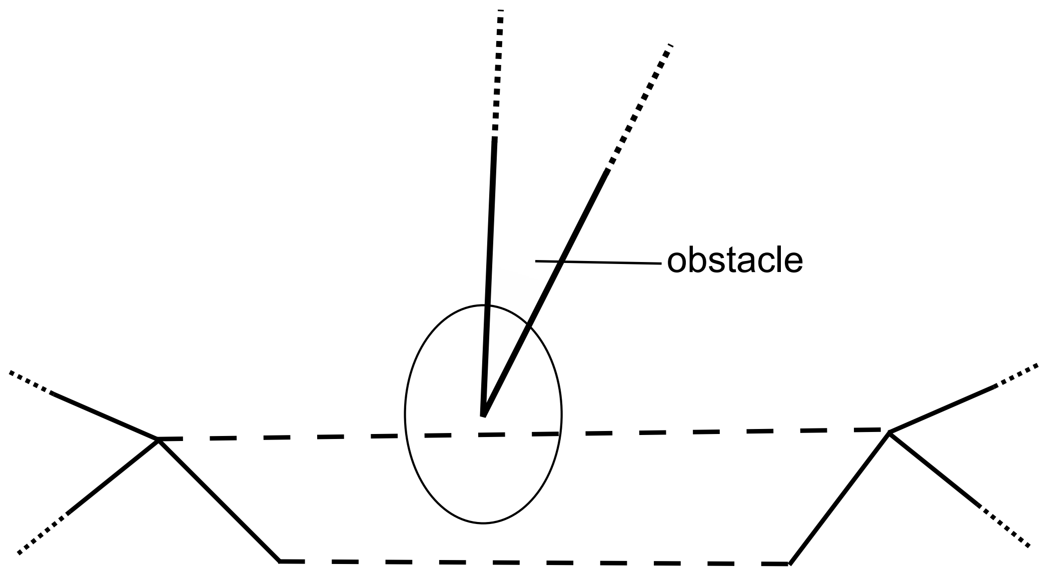


Figure 4.6: The free statuses of the two uncertain edges (shown as dashed lines) are dependent on each other. If the lower edge is blocked by the obstacle, the upper one must also be blocked, thus if the agent observes "blocked" for the lower edge it should deduce the free status of the upper edge. The ellipse represents the Gaussian distribution of the obstacle vertex at 1 standard deviation from the mean.

4.4.1.1 Encoding dependency

A dependency between edges expresses the property that the probability of one edge being free can be modified when an observation of a dependent edge is made. The degree to which marginal edge probabilities are altered by observations of other edges depends on how closely correlated the edges are. In Figure 4.6, the states of the two uncertain edges are closely linked, but not completely correlated. Table 4.5 shows firstly, a uniform belief state where the agent believes there is no correlation between the edges, a fully dependent belief state where the free statuses of both edges must match each other, and thirdly, a belief state showing a possible dependent configuration from the figure.

The knowledge of the correlation between the different edges is contained within the probability distribution over the set W. For example, consider the fully correlated belief state in Table 4.5: the agent knows that either both edges are free or both are blocked and that the probability for either event is equal. The other two worlds where the edge statuses differ (FB and BF) have zero probability so the agent does not believe they can occur. The laws of probability dictate that this fact cannot change as the result of a Bayes' update; no observation the agent makes can cause it to believe world FB or BF has a non-zero probability. The same idea can be seen in the partially correlated example in Table 4.5—the dependency between the two edges of Figure 4.6 is encapsulated by the belief state. The fact that the upper edge must be blocked if the bottom edge is blocked is represented by the probability for the FB world model equalling 0. By carefully constructing the starting belief state for the MDP, we can ensure that it infers the correct information about dependent edges as it receives edge observations during graph traversal.

Upper edge, lower edge	**Uncorrelated**	**Fully correlated**	**Partially dependent**
F,F	0.25	0.5	0.3
F,B	0.25	0	0
B,F	0.25	0	0.3
B,B	0.25	0.5	0.4

Table 4.5: Different starting belief states encode any dependencies between edges

4.4.2 Independence

The dependent MDP belief state represents the agent's knowledge as a probability distribution over the set W, mirroring the POMDP belief space, but this leads to MDPs with high numbers of states. However, there is more structure in this domain which we can exploit to reduce the state space size. The dependent model can represent dependencies between any combination of uncertain edges even though most are not likely to be dependent on one another, especially if they are in different areas of the environment: they are independent. This can be exploited to drastically reduce the dimensionality of the state space. We can assume that every uncertain edge is free or blocked independently of all other edges. Instead of maintaining one probability distribution over all $w \in W$, we maintain an independent, one dimensional probability distribution for each edge. The total dimensionality of the belief space is reduced from 2^m to m.

Theorem 4.2. *The size of the state space for the dependent model given by* (4.4) *is simplified for the independent model to:*

$$|\mathcal{S}| = n(d^{-1} + 1)^m \tag{4.12}$$

Proof. The belief about each edge in the independent MDP model is a one dimensional probability distribution, represented in the independent belief state with one element. By Definition 4.2, each element is discretised according to resolution d into $d^{-1} + 1$ values. Each distribution is discretised separately, so the total number of independent belief states is the product of the m possible distributions, therefore there are $(d^{-1} + 1)^m$ belief states for m uncertain edges. Each belief state must exist for all n nodes in the PRM graph, therefore the total number of independent belief states is given by:

$$|\mathcal{S}| = n(d^{-1} + 1)^m$$

thus proving the Theorem. □

We are only considering two worlds per uncertain edge instead of combinations of edge statuses. This is known as the *independent model* for the MDP. It is only the representation of knowledge about the edges in the world that differs between the dependent and independent models.

Definition 4.5 (Independent MDP Model)**.** When the belief state in an MDP state consists of m independent, one dimensional probability distributions (one per uncertain edge), we refer to the MDP model as an independent MDP model.

4.4.2.1 Converting between models

We can convert belief states between the dependent and the independent models, but the independent model cannot infer any relationships between edges. To form the separate probability distributions for each edge in the independent model, we need to calculate the marginal probability of each edge being free (the blocked probability is implicit since the distributions are one dimensional). The marginal probability for any edge can be calculated by summing the appropriate elements from the dependent distribution. Table 4.6 shows an example of converting a 2 edge dependent distribution to an independent one and back. For the dependent belief states, the free status of edge 1 is listed first. Finding the marginal probability that each edge is free is carried out by summing the probabilities of each world in the dependent distribution where that edge is free, e.g. the value in Table 4.6b for $P(\text{edge 1=free}) = P(\text{FF}) + P(\text{FB})$ from Table 4.6a. Table 4.6b shows the conversion of the dependent belief state in Table 4.6a. Converting the independent dis-

World	FF	FB	BF	BB
Probability	0.4	0.2	0.1	0.3

(a) Dependent distribution. Edge 1 is listed first.

	Edge 1		**Edge 2**	
World	F	B	F	B
Probability	0.6	0.4	0.5	0.5

(b) Independent distribution

World	FF	FB	BF	BB
Probability	0.3	0.3	0.2	0.2

(c) Reformed dependent distribution

Table 4.6: Dependence information is lost during conversion to and from an independent distribution. "F" indicates that an edge is free and "B" indicates it is blocked.

tribution to its dependent equivalent is simply a case of finding the joint probabilities for the respective edge statuses by multiplying the independent probabilities. For example to find the probability for world FB in Table 4.6c:

$$P(\text{edge 1=free,edge 2=blocked}) = P(\text{edge 1=free}) \cdot P(\text{edge 2=blocked})$$

where the two probabilities on the right hand side are taken from Table 4.6b. Table 4.6c shows the conversion of the independent distribution back to a dependent model. The assumed independence in Table 4.6b means a different dependent distribution is obtained in Table 4.6c that does not express the dependency of the original distribution because there is no way to express the correlation. Every independent belief state has a direct equivalent in the dependent model, but not vice versa; there are many dependent belief states which cannot be exactly represented in the independent model.

4.4.2.2 Model benefits

In Figure 4.6, assume the agent receives an observation informing it that the lower edge is blocked and that this observation is totally accurate. With an independent belief model, the agent knows that $P(\text{free})=0$ for the lower edge and will not consider it further. However, the upper edge represents a second opportunity to traverse this section of space so a likely policy is to continue to travel towards the top edge in the hope it is free, even though this cannot be the case. With a dependent belief model that can represent this fact, when the agent receives the perfect observation of the lower edge then the probability of the upper edge being free reduces to 0 as well. Inferring that both edges are blocked from the single observation allows the agent to generate a superior policy that will not waste resources on further investigation: it will start seeking an alternative route. Representing the dependency between two or more edges must obviously come at a price. The dependent model entails an enormous increase in state space compared to an independent model, hence requires much more memory to store. There is also a significant increase in computational effort to calculate new belief states. The trade-off is therefore between computational and memory requirements versus policy optimality. If the edges in the graph *are* in fact independent, then using a dependent model will not offer any advantage over an independent equivalent. In a graph of several uncertain edges, it is possible that dependencies only exist between some edges, while others remain independent. We show how we can exploit this in the following Section.

4.4.3 Edge clustering

The state space size under the dependent model motivated the development of the independent model with a reduced state space due to the lower dimensionality belief space. The disadvantage of such a model is that the agent loses the ability to reason about dependencies between the edges. A better approach would be a compromise between the two that sacrifices some of the expressiveness of the dependent model in exchange for a smaller state space, but without losing all its advantages. In the PRM graph, obstacles can cause multiple uncertain edges in close proximity to be blocked simultaneously and it is very likely that their probabilities of being blocked are dependent. On the other hand, uncertain edges that are far apart in the PRM graph are almost certainly independent of each other. In sparsely populated environments, it is likely that most edges are independent of each other due to the positioning of obstacles. However, while the complexity of the dependent model may seem unnecessary in such a scenario, assuming *all* edges are independent can ignore a dependent edge from which the agent could gain a useful advantage. This observation leads us to a novel approach to state space reduction that tries to simplify the belief space by reducing its dimensionality without losing the ability to express dependencies through combinations of edge statuses. The idea is to cluster together edges that are close to one another (i.e. where a dependency is likely), while leaving unrelated edges independent.

In the dependent model, all uncertain edges are considered as a single set of edges and the agent maintains its knowledge about their free statuses through a probability distribution over the set of all possible combinations. In the independent model, each edge is treated separately and the agent maintains a separate probability distribution for each edge in isolation. Clustering is a hybrid of these two models where the agent considers groups of uncertain edges. Given the m uncertain edges in the graph, we divide the edges into k separate clusters, with every uncertain edge being assigned to exactly one cluster, so no edge is a member of multiple clusters. Each cluster is considered independent of the other clusters, just as the edges are independent in the independent model, but within each cluster, we maintain a probability distribution for those edges as in the dependent model. This means that the agent can still reason about dependencies between edges that are within the same cluster and gain the advantages of doing so, but also gains the computational advantages of assuming independence between edges in different clusters. Independently considering the distributions over different clusters greatly reduces the number of elements required to record the agent's entire belief state, but to a lesser extent than in the independent model.

4.4.3.1 Clustered MDP model definition

As with conversion to the independent model, only the nature of a belief state for the MDP is changed, so only the state space $\mathcal{S}$ and the transition function t need to be updated.

Definition 4.6 (Cluster)**.** A cluster is a subset of the m uncertain edges in the PRM graph, where each edge in cluster c is not a member of any other cluster. The number of edges in cluster c (its size) is denoted m_c.

Definition 4.7 (Cluster Set)**.** A cluster set or cluster configuration describes the arrangement of the m uncertain edges in a given PRM graph into a mutually exclusive set of k clusters where:

$$\sum_{i=1}^{k} m_{c_i} = m$$

Each uncertain edge is assigned to be a member of exactly one cluster. The m uncertain edges in the environment are arranged into a set of clusters C with $k = |C|$. Each cluster contains a probability distribution over the possible combinations of member edges. The sub-worlds for cluster c_i are $w_{c_i,1}, \ldots, w_{c_i,2^{m_c}}$. This distribution forms the agent's belief about those edges. The definition for the belief space for one cluster is analogous to the dependent belief space $\mathcal{B}_D$ defined in (4.7) and (4.8) except that it is only defined for the sub-worlds of the cluster.

Definition 4.8 (Cluster Belief State)**.** Let the belief state for cluster c_i be defined by:

$$b_{c_i} = \{P(w_{c_i,1}), \ldots, P(w_{c_i,2^{m_{c_i}}})\} \tag{4.13}$$

The belief space of cluster c_i then mirrors the definition of $\mathcal{B}_D$ in (4.8):

$$\mathcal{B}_{c_i} = \mathcal{P}(b_{c_i}) \tag{4.14}$$

A belief state for cluster set C is the concatenation of the belief states for each cluster:

$$b_C = \{b_{c_1}, \ldots, b_{c_k}\} \quad 1 \leq i \leq k \tag{4.15}$$

Definition 4.9 (Clustered State Space)**.** The clustered state space is defined by:

$$\begin{aligned}\mathcal{S} &= V \times \mathcal{B}_{c_1} \times \ldots \times \mathcal{B}_{c_k} \\ &= V \prod_{i=1}^{k} \mathcal{B}_{c_i} \end{aligned} \tag{4.16}$$

The state space for the clustered MDP model is the cross product of all the cluster belief spaces and the set of PRM nodes.

Definition 4.10 (Clustered MDP model)**.** When the belief states in the MDP states consist of k probability distributions for the clusters of a given cluster set and the belief space is defined according to Equation (4.16), we refer to the MDP model as a clustered MDP model.

The transition function for the clustered MDP model behaves exactly as for the dependent MDP (see Section 4.3.4). It can be intuitively calculated by converting the entire clustered belief state into its dependent equivalent, computing the transitions as described earlier and then converting back again. In practise, to determine the possible belief states resulting from an observation, we can avoid doing the conversion by applying the transformation to each of the k clusters in turn as follows:

Let $o_e = \langle e \in E, ob \in \{\text{free}, \text{blocked}\}\rangle$ be an informative observation of an uncertain edge e that has been observed as ob (we may see either free or blocked). Since in the MDP formulation, we treat observations of different uncertain edges as separate observations, we can assume the agent receives an observation set

$$O = \{o_{e_1}, \ldots, o_{e_z}\}$$

of z uncertain edges when it reaches a PRM node. Assume we have a function T that given a dependent belief state b_D and an observation set O, returns a new dependent belief state b'_D that results from the making the observations:

$$b'_D = T(b_D, O)$$

T is calculated according to the method shown in Section 4.3.4. For the clustered model, we use function T_C, which is a modification of function T that only updates the belief state for cluster c_i. Not all of the edges in observation set O may be in the same cluster, so function T_C uses a subset of O that only contains the observations about edges in cluster c_i:

$$O' = \{\text{all } o_e \in O : \text{where } o_e \text{ is a member of } c_i\}$$

If O' is empty, then none of the observed edges in O are members of c_i, in which case the belief state for c_i is unaffected by the observations. Otherwise, the belief state is updated in the same way as before, but using the observation subset O':

$$T_C(b_{c_i}, O) = \begin{cases} b_{c_i} & \text{if } O' = \emptyset \\ T(b_{c_i}, O') & \text{otherwise} \end{cases}$$

To update a complete clustered belief state b_C, we simply apply function T_C to every cluster in the cluster set:

$$b'_C = \{T_C(b_{c_1}, O), \dots, T_C(b_{c_k}, O)\}$$

4.4.3.2 Discretisation error

The maximum discretisation error (see Section 4.3.2.3, page 91) for the independent and clustered model are also different from the dependent model. Let δ be the desired maximum discretisation error per one dimensional probability distribution in the independent MDP model. Since each probability distribution only concerns one uncertain edge, Equation (4.6) gives:

$$d \leq \frac{2\delta}{2^1} \leq \delta$$

with $m = 1$. With m edges we have m distributions, so the maximum discretisation error over the belief state Δ, is:

$$\Delta = m \cdot \delta$$

therefore:

$$d \leq \frac{\Delta}{m}$$

The clustered case is slightly more complex since each cluster is considered independently. Let δ_c be the maximum discretisation error for cluster c:

$$\delta_c = 2^{m_c} \cdot \frac{d}{2}$$

so the maximum discretisation error for clustered belief state b_C is:

$$\Delta = \sum_{i=1}^{k} \delta_{c_i}$$

Each cluster may have a different quantity of member edges, so for a desired maximum error over the entire belief state, d must satisfy:

$$d \leq \frac{2\Delta}{q}$$

where q is the total number of elements in b_C:

$$q = \sum_{i=1}^{k} 2^{m_{c_i}}$$

4.4.3.3 Clustered state space size

The reduction in belief space dimensionality leads to a reduction in the size of $\mathcal{S}$ depending on how the edges are arranged in the cluster set. As edge combinations within one cluster are treated the same as edges in the dependent model, the number of sub-worlds in each cluster is still exponential in m_c, i.e. there are 2^{m_c} sub-worlds for cluster c. The size of the state space in the clustered model is a modification of Equation (4.4) to accommodate the multiple distributions.

Theorem 4.3. *Given a cluster set C, the size of the clustered state space is given by:*

$$|\mathcal{S}| = n \prod_{c \in C} |c| \tag{4.17}$$

where the size for an individual cluster c is:

$$|c| = \frac{(d^{-1} + 2^{m_c} - 1)!}{d^{-1}!(2^{m_c} - 1)!} \tag{4.18}$$

Proof. Equation (4.4) calculates the total number of states in the dependent MDP model. Each cluster contains one probability distribution over its member edges as defined by Equation (4.13) in the same way as a dependent state contains one probability distribution over all uncertain edges. Factoring out the number of nodes n, from Equation (4.4) and substituting m for the number of edges in the cluster m_c, gives Equation (4.18). By the assumption of independence between clusters, each cluster in C maintains a separate probability distribution. The size of the clustered belief space in Equation (4.15) is therefore given by:

$$|\mathcal{S}| = \prod_{c \in C} |c|$$

As in the dependent MDP model, each belief state may exist at each of the n nodes in the PRM graph, so the total clustered state space size in Definition 4.9 is given by:

$$|\mathcal{S}| = n \prod_{c \in C} |c| \qquad \square$$

As an example, assume we have a PRM graph consisting of $n = 20$ nodes with $m = 4$ uncertain edges. If we convert this into the three different MDP models using a discretisation level of $d = 0.1$ and using a cluster configuration of 2 clusters of 2 edges, then by (4.4), (4.12) and (4.17) we have:

Model	$|\mathcal{S}|$	**Belief state elements**
Independent	292,820	4
Clustered	1,635,920	6
Dependent	65,375,200	15

As the size of the state space is effectively determined by the dimensionality of the belief space, the number of elements required to represent a complete belief state are also shown. In the independent model, each distribution is one dimensional, hence one element is required per uncertain edge. In the other two models, the number of elements is the sum of the elements in each distribution. For the dependent model, this is $2^4 - 1 = 15$ elements (-1 for the implicit last element) while for the clustered model with 2 edges per cluster, $2 * (2^2 - 1) = 6$ are required. The benefits to the size of the state space using an independent model are obvious with two orders of magnitude difference between it and the dependent model. The state size of the independent model is well within the computable bounds of contemporary MDP solvers, while the dependent model state space is only solvable if the MDP can be represented with specific structures as with the SPUDD MDP solver (Hoey et al. 1999). Of course, the independent model has the disadvantages discussed above, which is why the clustered model represents an attractive compromise; the state increase over the independent model is reasonable, and edge dependencies can still be exploited. The configuration of the clusters is important here since if two dependent uncertain edges are in different clusters, then they will be treated independently. This leads to an important insight that due to the independence between clusters and dependence within a cluster, the clustered model is a generalisation of the other two. The independent and dependent model are the two extremes of edge assignments into clusters. If every uncertain edge is placed into a separate cluster (effectively $k = m$ and $m_c = 1\ \forall c$) then they will all be treated independently, exactly mirroring the independent model. Conversely, if all edges are simply placed in the same cluster ($k = 1$ and $m_{c_1} = m$) it is the equivalent of the dependent model.[1]

It is also important to note two concepts that should not be confused: independence/ dependence in the model and independence/dependence in the actual edges. When the

[1]Internally, this is exactly how our system is implemented.

model is discussed, it refers to the *representation* of the uncertain edge probabilities, i.e. how the agent records its knowledge of the world by assuming all probabilities are independent or dependent to some degree. When the edge independence is mentioned, it refers to whether the edges actually are independent or not. This knowledge is encoded in the starting belief state through the distribution of the probability mass. Some examples are shown in Table 4.5 on page 103.

4.4.4 Summary

We have presented three models for representing the POMDP belief space in an MDP formulation, summarised here for clarity:

- Independent (I)—Each edge is treated independently. There is only one dimension in the belief space per edge and no ability to represent dependencies between edges. $|\mathcal{S}|$ is the smallest of all three models.
- Clustered (C)—Edges believed to be dependent are placed in the same cluster. Each edge only appears in one cluster and clusters are treated independently from each other. $|\mathcal{S}|$ is determined by the cluster configuration, as more edges are placed in the same cluster the belief space increases in dimensionality as it becomes closer to the dependent model.
- Dependent (D)—All edges are treated as being dependent. This is the model that most closely represents the POMDP formulation and can represent all belief states the agent may encounter accurately (according to limits of discretisation). $|\mathcal{S}|$ is the largest of all three models.

4.5 Experimental Configuration

This Section describes additional details specific to the system that are necessary to understand the experimental work of the thesis.

4.5.1 LAO* implementation

LAO* is used to solve the MDP models throughout the experiments in this thesis unless otherwise noted. The implementation we use is based on the optimised LAO* algorithm shown in Algorithms 2.3 and 2.4 (page 37). The `expand` function (line 3 of Algorithm 2.4) and the heuristic estimate h, are somewhat problem dependent so we define them for our problem here.

Algorithm 4.2 Implementation of `expand`

Input: $G_{\text{PRM}} = \langle V, E \rangle$ a PRM graph, G an LAO* explicit graph, $n = \langle v \in V, b_C \rangle$ an LAO* node with clustered MDP belief state b_C of k clusters

Output: n is expanded by one level with arcs to successor nodes

```
 1: for all $a \in \mathcal{A}$ do  //$a$ represents a PRM node label
 2:    if $\langle v, a \rangle \in E$ and edge not believed to be blocked then  //edge exists from $v$ to $a$
 3:       $L \Leftarrow$ set of observable uncertain edges at $a$
 4:       for $i = 1, \ldots, 2^{|L|}$ possible assignments of {free,blocked} to all $l \in L$ do
 5:          $O \Leftarrow$ assignment $i$ to $L$
 6:          $b'_C = \{T_C(b_{c_1}, O), \ldots, T_C(b_{c_k}, O)\}$  //new belief state when $O$ observed
 7:          $n' \Leftarrow$ find node $\langle a, b'_C \rangle$ in $G$
 8:          if $n'$ does not exist then
 9:             $n' \Leftarrow \langle a, b'_C \rangle$  //create new node
10:             $n'$ value $\Leftarrow \begin{cases} 0 & \text{if } a = v_G \\ h(n) & \text{otherwise} \end{cases}$  //0 at goal nodes, otherwise use heuristic
11:             $G \Leftarrow G \cup n'$  //add to explicit graph
12:          end if
13:          add arc from $n$ to $n'$ under action $a$ with transition probability $P(O)$ and cost $c_{\langle v,a \rangle}$
14:       end for
15:    end if
16: end for
```

4.5.1.1 Node Expansion

The `expand` function of LAO* should return all of the successors of a given node n, in the MDP with each arc's transition probability taken from the transition matrix $\mathcal{T}$. For our domain, the successor nodes are determined by the observations (if any) at the successor PRM node and the agent's belief state at node n. Algorithm 4.2 shows the implementation for our MDP models. The computation of the resulting belief state given an observation set is dependent on the MDP model currently in use. The belief state calculation assumes a clustered model, since both the independent and dependent model can be represented through the appropriate configuration of clusters.

4.5.1.2 Heuristic

The heuristic function $h(n)$ must return the estimated cost to reach the goal from node n and must be admissible. The PRM graph itself forms an ideal base for the heuristic because the edge traversal costs are explicitly defined. For the heuristic, we use the cost of the shortest path to the goal v_G, assuming all uncertain edges are free:

$$h(n) = \text{cost of shortest path from } n \text{ to } v_G : P(w_1) = 1 \tag{4.19}$$

where w_1 represents the world where all uncertain edges are free of obstruction. When all edges are assumed to be free, the agent has the maximum number of path choices to reach the goal from node n, so the heuristic will return the cost of the absolute shortest path. We can pre-compute the heuristic value for all PRM nodes before the LAO* algorithm is invoked using a version of Dijkstra's algorithm that finds the shortest path to all points, instead of terminating when a specific node is marked. We can show that this creates an admissible, consistent heuristic with the following lemmas. A consistent heuristic has the stronger property of being locally admissible.

Definition 4.11 (Local admissibility)**.** A heuristic h, has the property of local admissibility if node n' is the successor to node n in the search graph and h obeys:

$$h(n) \leq c(n, n') + h(n') \quad \forall n, n'$$

where $c(n, n')$ is the cost of reaching n' from n.

Lemma 4.1. *Equation* (4.19) *creates an admissible heuristic where $h(n) \leq h^*(n)$, the true cost of reaching the goal from node $n = \langle v, b \rangle$ for all $v \in V$.*

Proof. We prove the admissibility of h by observing that in world w_1 all uncertain edges are free, so the agent has the greatest choice of routes to the goal and the shortest route will be free. If Dijkstra's algorithm calculates the shortest route from PRM node v to node v_G under the assumption that $P(w_1)$=1, the associated belief state b of node n is irrelevant. If the assumption is correct, then the heuristic estimate of $h(n)$ will be the true shortest path cost to the goal: $h(n) = h^*(n)$. If the assumption is incorrect, then some edge in the shortest route may be blocked, in which case the true shortest path to the goal will involve a more costly route. This leads to the inequality stated in the lemma:

$$h(n) \leq h^*(n) \qquad \square$$

Lemma 4.2. *Equation* (4.19) *creates a locally admissible heuristic. If n and n' are nodes in the explicit graph representing nodes v and v' in the PRM graph respectively, where v' is the successor to v, then the h obeys $h(n) \leq c(n, n') + h(n') \quad \forall n, n'$ where $c(n, n') = c_{\langle v, v' \rangle}$, the cost of reaching n' from n.*

Proof. If $h(n)$ and $h(n')$ are the costs of the respective shortest paths to v_G, then at n there are two cases:

1. The shortest route from n to v_G passes through n'. In this case, since h uses the shortest route to calculate the cost and n' is the direct successor to n, $h(n)$ and

$h(n')$ must differ by exactly $c(n, n')$:

$$h(n) = c(n, n') + h(n')$$

2. The shortest route from n to v_G does not pass through n'. In this case $c(n, n')$ must be great enough to prevent the shortest route at n passing through n' and there must be an alternative path to v_G, so:

$$h(n) < c(n, n') + h(n')$$

therefore:

$$h(n) \leq c(n, n') + h(n') \qquad \square$$

4.5.2 Simulator

In standard MDPs, we can compare the quality of two policies by directly comparing the values assigned to states in the value function. Equation (2.5) (page 28) can be used to determine if one policy is strictly better than another. However, this is insufficient for our problem since we are not directly comparing two policies for one MDP, instead we need to compare policies for the agent acting in the environment. The policies generated by the three different MDP models are not directly comparable for two reasons. Firstly, we cannot compare policies directly by their value functions because these do not reflect the true state values. In the PRM graph that the agent acts in, edges may be dependent on one another. However, the independent and clustered MDP models compute their policies on the assumption that all edges are independent (or partially so in the clustered model), so equivalent states will obtain different values from the dependent model and may not be correct. This means that a numerical comparison between states from different MDP models will be almost meaningless. Secondly, recall that while the dependent model can represent any belief state (ignoring discretisation) from the POMDP formulation, the independent and clustered belief spaces are strict subsets of the dependent model belief space. Therefore, transferring a policy from either model to the dependent model before running a policy evaluation cycle leaves open the problem of how to handle dependent belief states that do not exist in the smaller models. While copying the action from the nearest equivalent independent/clustered belief state prior to policy evaluation is possible, the results would still be misleading. However, there are two other issues which eclipse these. LAO* by design does not produce a policy for all states, only for the ones that are reachable from the start state, so policies are naturally incomplete with respect to the entire state space. The most important issue however, is that we wish to compare performance of policies generated using different families of algorithms, such as POMDP solvers and other non-decision-theoretic methods like MCC. Therefore, we need an algorithm agnostic method of comparing the agent behaviour in a PRM graph. Based on this need, we developed a simulator that simulates the agent acting in the actual PRM graph over repeated independent trials.

Each trial starts with the simulator selecting world w_i from W with probability $P(w_i)$ to be the true world. $P(w_i)$ is taken from a provided probability distribution over W. In reality, this is the dependent MDP starting belief state. The true state of the world remains hidden from the agent. The simulator then simulates the environment for the agent, maintaining the agent's current location in the PRM graph and executing the action (if allowed) selected by the agent's policy. The action set is the same as is defined for the POMDP and MDP formulations, namely the set of labels of the PRM nodes. After each action, the simulator samples from the observation probabilities specified in the PRM graph to generate an observation for the agent. This allows the agent to update its knowledge of the world, though we are careful not to use the term "belief state" here because not all algorithms maintain one in the sense described in this thesis. An action selection and the resulting observations form one step of a trial. This action-observation sequence continues until either the goal is reached, resulting in a successful trial, or a preset maximum number of trial steps is exceeded before the goal is reached, resulting in a failure. Trials can also fail if the agent selects an illegal action in the simulator. An action can be illegal for two reasons:

1. Trying to travel to a PRM node when there is no edge to it from the current node.

2. Trying to traverse an uncertain edge that is actually blocked.

Once a trial has ended, the agent and environment are reset for the next trial.

The method of policy selection was left deliberately vague in the above description. We developed a system of "plugins" for the simulator to allow for different algorithms to be used under the same conditions. Plugins are provided with the starting node and belief state (the distribution over W). At each time step, the plugin is asked which action should be selected next and is given any observations generated by the simulator. The method of action selection is left completely unspecified and depends upon the algorithm used to generate the policy.

The simulator collects a variety of data throughout the simulation; most notably it records the cost of each step (the graph edge costs). After all trials have completed, the minimum, mean, maximum and the standard deviation of the total trial costs are reported. The number and percentage of failed trials are recorded separately. In a trial of a graph where enough uncertain edges are blocked to prevent the agent reaching the goal, the simulator will record a failure because it does not provide a give-up action to plugins. Only successful trials are considered in the above data to prevent failed trials from distorting the results. The disadvantage in not counting failed trails is due to the reward based feedback in decision-theoretic algorithms. If the cost for failing is less than the cost of reaching the goal, then deliberately failing appears to be the optimal plan. In the simulator, this behaviour can put the algorithm at a disadvantage because it will appear to be performing worse than an algorithm that is reaching the goal more often. However, since the agent's aim is to reach the goal, then including the cost of failed trials can also be misleading. As an example, in a graph where the paths to the goal have a cost in the region of a few hundred units, assume the first edge costs 50 and that algorithm A reaches the goal but algorithm B nearly always executes an illegal action after the first move, thus incurring a failure. By counting failed as well as successful runs, the simulator would record all of algorithm B's failed trials as costing 50. This would drastically lower the average cost reported for algorithm B, making it appear to have better performance than algorithm A unless the percentage of failed trials is taken into account. By only including successful trial costs in the reported average, the results are not skewed in this fashion.

The simulator does not apply discounting when reporting the costs incurred by the agent: it reports the undiscounted cost. The discount factor γ, affects the agent's preference for a short-term reward versus a long-term reward. Although discounting is applied when policies are being computed, we use a discount factor of $\gamma = 0.999$ which will not affect the policy found even when alternate paths have very similar costs as shown below. When comparing MDP algorithms to other algorithms, such as MCC, which do not have a notion of short term versus long term reward, it is unclear what the discount factor should be set to. We do not specify a planning horizon for the MDP algorithms because we are interested in the total cost to reach the goal, and the number of time steps to do this depends on the PRM graph. The maximum length of simulator trials are limited to 50 steps in the experiments, so using $\gamma = 0.999$, any costs at a horizon of 50 will still contribute nearly all of their value to the utility of the start node. As shown in Equation (2.7), the reward at each time step in the future becomes more discounted than the previous reward when considering the value of a state. For a cost that is incurred 50 time steps into the future:

$$0.999^{50} = 0.951 \text{ (3 s.f.)}$$

When comparing two routes, one where nearly all the cost is accrued in the first step and another where it is accrued at step 50, as long as the difference in the costs is less than 5%, LAO* would still choose the optimal route. This is because whichever route is truly cheaper will still appear cheaper under discounting, so LAO* will still make the correct choice. As this extreme distribution of cost will not occur in practice and the majority of successful trials will be shorter than 50 steps, the probability of LAO* choosing a sub-optimal route due to the discount of $\gamma = 0.999$ is extremely unlikely.

Due to the randomness present in the world selection and the high difference in total traversal costs between different worlds, a high number of trials are conducted to ensure results truly represent the average cost of traversing a graph. In this research, we are primarily interested in two performance measures: firstly, the cost to reach the goal and secondly, the computation time for the policy. The first is derived from the simulator and the second is determined separately when the respective algorithm computes the policy. All policies for all algorithms are computed offline before being passed to the simulator. The one exception to this is the replanning variant of MCC which must replan its route when a blocked edge is found during simulation.

4.5.3 Graph pre-processor

In a POMDP, the belief state changes after every time step, since an observation is received after every action choice. In our domain, not all actions produce observations, since most PRM nodes do not contain observations. At these nodes, the belief distribution over worlds is not changed. In the MDP models, observational PRM nodes are represented through the transition function probabilistically leading to MDP states with different belief states. Transitions to non-observational nodes are deterministic with only the PRM node label component of the MDP state changing. Even so, all of these deterministic transitions and states must still appear as nodes in the explicit graph built by LAO* despite contributing nothing to the agent's knowledge; the agent is essentially just passing through these nodes between observations. These non-observational nodes therefore have no influence on the overall agent behaviour, as it will always be taking the shortest route between observational nodes. For example, traversing between observational nodes a and b via a non-observational node x is directly equivalent to traversing a direct edge between a and b where $c_{\langle a,b\rangle} = c_{\langle a,x\rangle} + c_{\langle x,b\rangle}$. We can exploit this in our POMDP and MDP formulations by pre-processing the PRM graph before converting it into a POMDP or an MDP without affecting the total traversal cost.

The graph pre-processor described in Algorithm 4.3 removes as many nodes as possible from the PRM graph. Not all PRM nodes without observations can be removed because some are needed to ensure "safe" routes around uncertain edges to be used if those edges become blocked (lines 3 to 7). Any shortest paths between nodes in the pre-processed graph must not use any uncertain edges or visit other nodes in the pre-processed graph in their route. The MDP or POMDP state space size (see Equations (4.1),(4.4),(4.12) and (4.17)) is reduced by a factor of r for r removed PRM nodes. Figure 4.7b shows

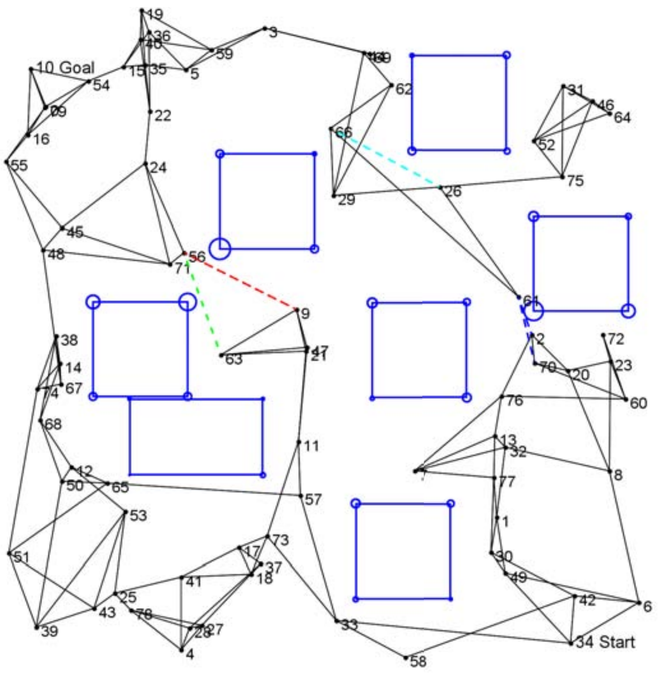
Algorithm 4.3 Graph pre-processing

```
Input: $G_{\text{PRM}} = \langle V, E \rangle$ PRM graph with nodes $V$ nodes and edges $E$, $U = \{u_1, \ldots, u_m\}$
    set of uncertain edges
Output: $G'_{\text{PRM}} = \langle V', E' \rangle$ pre-processed graph with unnecessary nodes in $V$ removed
 1: $V' \Leftarrow \{$all $v \in V$ : no observations at $v$, (i.e. $O = \emptyset$)$\}$
 2: $V' \Leftarrow V' \cup v_S \cup v_G$  //include start and goal nodes
 3: for all $u = \langle v_1, v_2 \rangle \in U$ do  //for each uncertain edge
 4:   if path exists from $v_1$ to $v_2$ without crossing any uncertain edges or any $v \in V'$
      then
 5:     $V' \Leftarrow V' \cup$ last node in shortest path from $v_1$ to $v_2$
 6:   end if
 7: end for
 8: $E' \Leftarrow U$  //include all uncertain edges
 9: for all $v \in V'$ do
10:   for all $v' \neq v \in V'$ do  //try to connect each node to every other node
11:     if path exists from $v$ to $v'$ without crossing any uncertain edges or any $v$” $\in V'$
        then
12:       $e \Leftarrow \langle v, v' \rangle$ with $c_e \Leftarrow$ cost of shortest path from $v$ to $v'$
13:       $E' \Leftarrow E' \cup e$
14:     end if
15:   end for
16: end for
17: return $G'_{\text{PRM}} = \langle V', E' \rangle$
```

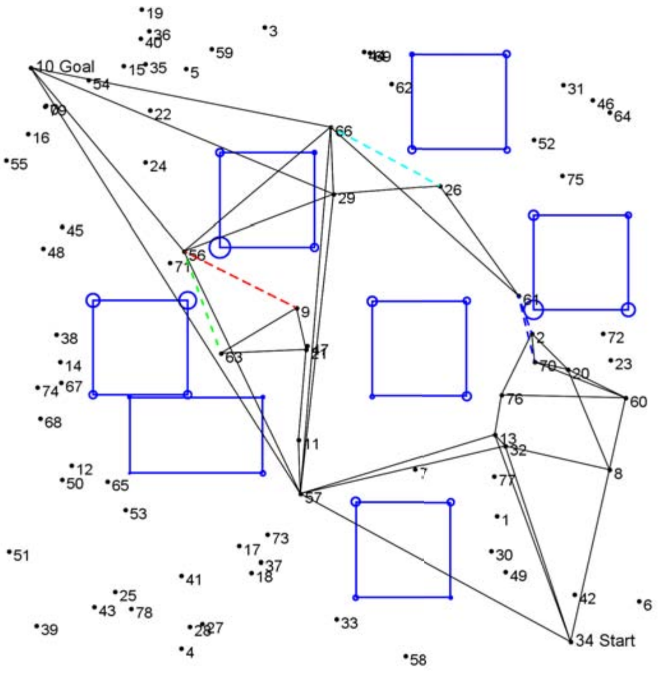
an example of using the pre-processor on the PRM graph in Figure 4.7a, illustrating the difference in number of connected nodes. The edges in Figure 4.7b are only illustrative, indicating that an edge exists between two nodes. The actual edge costs are not based on edge length or the physical path the agent would follow between the end-points. We use the pre-processor for experiments on larger graphs to reduce the memory footprint of the algorithms and to keep the POMDPs tractable. The computation time of the pre-processor is small, requiring between 5-15ms to complete. On average, the pre-processing phase will remove about 75% of nodes based on the example graphs we use.

4.5.4 Hardware and software environment

All experiments in this thesis are conducted with an Intel® Xeon® E5345 processor (2.33GHz core clock, 8MB L2 cache) with 8GB of memory running on Linux CentOS version 5 (kernel version 2.6.18). All implementations with the exception of Perseus and Symbolic Perseus are written and executed using Java™ version 1.6.06 (Sun Microsystems 2008) with 500MB of allocated memory and the 32-bit server virtual machine. Perseus and parts of Symbolic Perseus are written entirely in MATLAB (The MathWorks Incorporated 2008) and executed using 32-bit MATLAB version 7.7.0.471 (R2008b).



(a) An example of a PRM graph.



(b) The pre-processed version of the graph. The edges only indicate a connection between nodes, not the physical path an agent would follow in the cases where edges intersect obstacles.

Figure 4.7: An example of a PRM graph before and after pre-processing is applied. Uncertain edges are shown as dashed lines. The colours of the uncertain edges indicate cluster configurations, with all uncertain edges of the same colour in the same cluster. In this example, the number of graph nodes are reduced from 80 to 20.

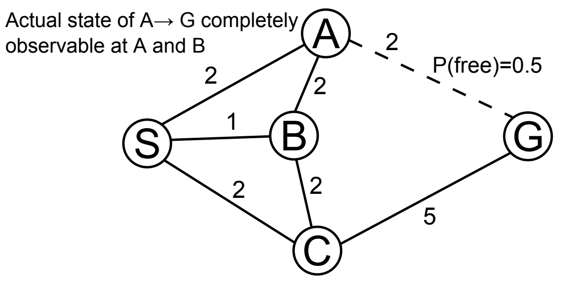
4.6 Experimental Results

4.6.1 Small example problems

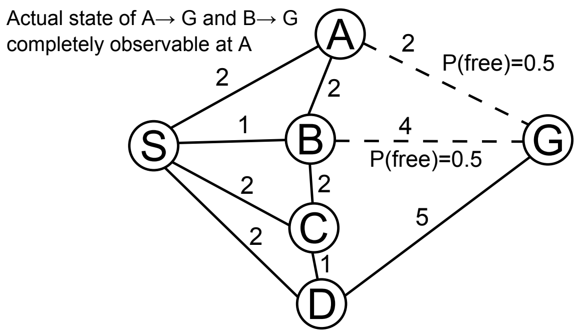
4.6.1.1 Exact value iteration POMDP

First we demonstrate finding the optimal policies on some small examples. These contain a minimal number of uncertain edges, keeping the state space small enough to enable solving via exact POMDP value iteration. The three example graphs are shown in Figure 4.8: 4.8a (the “5 point” graph) shows a trivial graph with only one uncertain edge, small enough to be solved almost instantly by all algorithms; 4.8b (the “6 point” graph) shows a similar graph with 2 uncertain edges. The common topology between both of these graphs leads to similar optimal policies; their main use is in testing algorithms, since their minimal size enables policies and costs to be verified by hand. The third graph in Figure 4.8c (the “3 edge” graph) contains 3 uncertain edges—the minimum necessary to set up an edge cluster configuration that is distinct from fully dependent or fully independent. This is useful in illustrating the difference in agent behaviour under the different MDP models with different assumptions about edge dependence. This will be examined later in this Chapter. The 3 edge graph is also an example of an “unsafe” graph: the agent must cross at least one uncertain edge to reach the goal. If all uncertain edges are blocked, the goal cannot be reached, preventing the agent finishing the graph. This affects the planner’s behaviour as will be seen later.

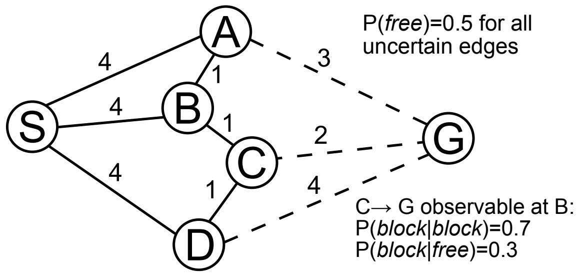
Four algorithms are compared on the graphs in Figure 4.8 and each algorithm is used to compute policies on each graph. The “pomdp-solve” software by Cassandra (2005) implements exact value iteration. Incremental pruning (“inc.prune” in the results tables) was selected for all problems shown as it is the most scalable algorithm for exact value iteration. Approximate value iteration is carried out by the PERSEUS POMDP solver which implements the point-based technique described earlier (see Section 3.4.1). The MDP approximations developed during this research are also presented here. The primary purpose of these experiments is to demonstrate that our techniques achieve the same results as the POMDP models; in these cases we are finding the optimal behaviour for the graphs. Two versions of our algorithm are shown: a standard implementation of value iteration (denoted as “MDP” in the results) and LAO*. Standard value iteration is a naïve MDP solver, implementing Algorithm 2.2 (page 31) to compute the policy for the entire MDP state space. These examples have small state spaces (shown in column $|\mathcal{S}|$) compared to most problems, so standard value iteration is possible. This being the case, a very coarse discretisation is still required to keep this method feasible. This is discussed further below.



(a) The 5 point graph



(b) The 6 point graph



(c) The 3 edge graph

Figure 4.8: Three small example graphs used for testing algorithm behaviour. In all graphs the agent must traverse from S to G. Uncertain edges are shown as dashed lines. All other edges are certain.

4.6.1.2 Algorithm set-up

All the experiments in this Section were run in the simulator with a discretisation level of $d = 0.01$ and a discount factor of $\gamma = 0.999$ unless stated otherwise. All times are stated in milliseconds except where the experiment runtime was greater than an hour and larger units are substituted for readers' convenience. Each policy was run in the simulator for 50,000 trials with a maximum trial length of 50 steps. All policy generation times are averaged over 10 runs. The starting belief state for all graphs was set to the uniform belief state since we are interested in comparing the MDP approximations to the optimal policy at this stage, rather than the effects of edge dependencies. For all models, this gives the planners no bias about the free statuses of the edges, they believe that each edge is independently blocked with probability 0.5. For Perseus, a pool of 50,000 belief states was generated for solving each of the graphs (generation time not included in solution time). For both POMDP solvers, the discount rate is $\gamma = 0.95$ instead of 0.999 to ensure convergence within a feasible time. For these graphs, the goal state is reached within a small number of steps so the small change in discount does not affect the policy. When applied to the larger graphs, it can have a large effect on solution time with little effect on policy. This is demonstrated later in Section 4.6.3.4 on page 140. The stopping criterion is $\epsilon = 1 \times 10^{-5}$ for all algorithms.

4.6.1.3 Results

Algorithm	$\lvert\mathcal{S}\rvert$	Avg.sim.cost	Std.dev	Time (ms)
Inc.prune	11	6.50	1.5	97
Perseus	11	6.51	1.5	5086
MDP	505	6.50	1.5	6
LAO*	505	6.50	1.5	2

Table 4.7: The performance of various algorithms on the 5 point graph in Figure 4.8a.

Algorithm	$\lvert\mathcal{S}\rvert$	Avg.sim.cost	Std.dev	Time (ms)
Inc.prune	49	6.24	2.78	5hr 59m[1]
Perseus	49	6.26	2.78	8132
MDP (ind.)	61206	6.27	2.78	607
MDP (dep.)	1061106	6.25	2.78	7982
LAO* (ind.)	61206	6.27	2.78	4
LAO* (dep.)	1061106	6.25	2.78	5

Table 4.8: The performance of various algorithms on the 6 point graph in Figure 4.8b.

Algorithm	$\lvert\mathcal{S}\rvert$	Avg.sim.cost	Std.dev	Time (ms)
Inc.prune	49	7.99	0.93	6350[2]
Perseus	49	7.99	0.92	39265
MDP (ind.)	7986	8.40	2.89	16577
MDP (clus.)	18876	8.39	2.89	41060
MDP (dep.)	116688	14.92	10.18	256615
LAO* (ind.)	7986	8.40	2.89	405
LAO* (clus)	18876	8.40	2.89	457
LAO* (dep.)	116688	14.87	10.18	144

Table 4.9: The performance of various algorithms on the 3 edge graph in Figure 4.8c. For this graph, $d = 0.1$.

The 5 point graph (Table 4.7) is trivial to solve, even by hand. The optimal policy can be shown diagrammatically and can be seen in Figure 4.5 (page 101). All algorithms compute the optimal policy in a short time because the optimal policy is very simple. Under it, there are only two paths the agent can take to the goal: either $S \rightarrow B \rightarrow A \rightarrow G$ (cost 5) or $S \rightarrow B \rightarrow C \rightarrow G$ (cost 8), depending on the state of edge $A \rightarrow G$ (observed perfectly at B). The standard deviation is therefore 1.5 no matter how worlds are sampled for simulation trials. The small, highly deterministic state space

[1]Time to complete 6 iterations as value function had not converged.

[2]Time to complete 4 iterations before crashing.

lends itself well to incremental pruning which solves the POMDP substantially faster than the approximate algorithm. In very small domains, the overhead of managing the belief pool is overshadowing the more efficient computation of the approximate solver. Incremental pruning is also benefiting from a pure C implementation as opposed to the MATLAB implementation of PERSEUS which is initially interpreted.

The 6 point graph (Table 4.8) can also be solved optimally by all algorithms. The very small differences in average simulated cost can be attributed to the randomness in the world selection during trials. As there are two uncertain edges, we can solve the MDP under independent and dependent models (even though no dependence exists, as explained above) to show the effect on solution time for the naïve MDP method. The large increase in solution time for the dependent MDP is a result of the massively increased state space. The absolute time required for VI is still relatively small, but the memory required to solve an MDP with greater than one million states is the limiting factor. This MDP approaches the fringe of what the implementation can handle. LAO* does not have the same limitation because it only requires memory for the tiny fraction of the state space that it explores (i.e. its explicit graph), not the implicit state space which is the same as for normal VI. In this example, LAO* only needs to evaluate 29 states to find the optimal policy, hence the almost negligible runtimes shown.

Despite the fairly coarse discretisation of $d = 0.01$, the 6 point graph illustrates how standard VI soon becomes intractable, even for tiny problems. It is not suitable for large problems—a tiny graph of 6 points and 2 uncertain edges has a state space of over one million states. Incremental pruning (which of course uses no discretisation) was left to run for over 24 hours, but never converged, so results shown are from the last complete cycle of POMDP value iteration. At this scale, PERSEUS is still competitive with the standard MDP in both policy performance and runtime.

For the 3 edge graph (Table 4.9), the discretisation resolution for the standard MDP solver had to be changed to $d = 0.1$ to make the MDP solvable; the same discretisation was applied to LAO* for comparison. There are a number of important points to note about this set of results. Firstly, the incremental pruning POMDP crashed after 4 iterations after requesting too much memory, hence results are shown from the 4th iteration. However, the policy performance matches that of PERSEUS. This suggests previous intuition (Russell and Norvig 2002, p.623) is correct in asserting that the optimal policy is converged upon before the state values actually converge. The degrading performance of incremental pruning in larger examples also shows that it is not scalable as a general approach. Another differentiating factor for the 3 edge problem is that there is no longer a guaranteed route to the goal; this graph is "unsafe". When the simulator selects the world where all 3 edges are blocked, the agent can *never* reach the goal. The significantly longer runtimes for all algorithms result from the high discount factor increasing the number of iterations VI needed to converge for the states that cannot reach the goal. In all algorithms, the simulator reported a failure rate of at least 12.5% of the 50,000 trials. This is expected, since according to the initial uniform belief state, the world where the agent cannot reach the goal and inevitably fail will occur with a probability of 0.125.

The very coarse discretisation required to keep this graph solvable blocked any of the MDP models (or equivalent LAO* tests) from finding the optimal solution. This is especially evident under the dependent model where the policy is very poor indeed, as evidenced by the high average cost and standard deviation. The rounding used prevents the creation of belief states that can adequately capture the complexity of the underlying belief space. This effect is present in all three models, but is exacerbated with the dependent model because 7 variables are required to describe a belief point (the 8th is implicit), yet only 10 units of probability mass are available to distribute. If $d = 0.1$, then in order to give a non-zero probability to each of the 8 possible worlds, each must have a probability of 0.1, but this entails that only 2 of the 8 can have a probability of 0.2, or 1 with a probability of 0.3. The coarse rounding employed causes many continuous belief states to map to the same discrete belief state. In the next Section we will see that LAO* does find the optimal policy once discretisation resolution is increased.

Under the dependent model MDP, Table 4.9 shows an average cost of 14.87 to reach the goal in the 3 edge graph, even though the maximum cost of any edge is 4. The only way the agent could incur such a high cost is by getting stuck in a loop and repeatedly traversing the same edge. Essentially, the agent is "hopping" between two nodes. We have observed this behaviour under many different circumstances during the course of the research for a variety of reasons. In this case, the agent is hopping because the change in belief state from the noisy observation at node B is hidden due to discretisation; the information is lost and the agent is stuck in a loop. The agent visits the same node until a different observation changes the belief state triggering a change in behaviour. We can verify this by adjusting the maximum trial length as shown in Table 4.10. The maximum trial cost is the important statistic. A clear pattern is present: every 10 step increase in the maximum trial length up to 40 steps increases the maximum trial cost by 40. The agent expends more time traversing the same edge with a cost of 4. Once the agent is allowed more than 40 steps, the maximum stops increasing. 50,000 trials were used each time, though when this was increased, higher maximum costs were occasionally seen when trials were limited to 100 steps. The higher maximum costs are extremely rare because the agent must see a particular sequence of observations to continue the hopping behaviour, the probability of which decreases with length.

These small problems show the intractability of exact POMDP solvers. Although

Max.trial length	**Avg.sim.cost**	**Max.cost**	**Std.dev.**
10	12.82	31	6.56
20	14.83	71	9.96
30	14.92	111	10.18
40	14.90	151	10.21
50	14.92	151	10.21
100	14.89	151	10.18

Table 4.10: Increasing the maximum trial length on the 3 edge graph (Figure 4.8c) under the dependent model with $d = 0.1$.

the optimal policy was found early in value iteration, the lack of convergence shows the problem of scalability in this method and we cannot expect the policy to stabilise after so few iterations in bigger examples. Perseus is more scalable and converges to the optimal value function in a reasonable amount of time, so can be used on larger examples. The effects of low discretisation are also obvious even on graphs of this size, but the resultant state space size precludes the use of the standard MDP solver on more complex examples. Discretisation needs to be sufficiently fine to not "lose" observation information through rounding. This is investigated in more detail next.

4.6.2 Discretisation

The previous results clearly show major degradation in policy performance when the discretisation level is insufficient to represent the complexity of the belief space $\mathcal{B}$. Here, we alter the discretisation level and examine the results on the 3 edge graph and 6 larger graphs.

4.6.2.1 Larger graphs

For the experimental work in the research, we created a series of random graphs that would allow us to test various aspects of the techniques we have developed. Twenty random PRM graphs were generated with between 50 and 80 nodes per graph in an environment with manually placed obstacles. All except three graphs contain 5–6 uncertain edges. The starting belief states were handcrafted to ensure some edges were dependent on each other while others were independent. Observations were placed at nodes where the agent would be able to reasonably observe the object(s) near an uncertain edge, but not obtain perfect information. The prior probabilities of each uncertain edge being blocked were set according to how far the closest obstacle vertex was from the uncertain edge. Unlike many other applications, random graphs are good indicators of real world performance here because the PRM algorithm creates random graphs. Unfortunately, the randomness (in

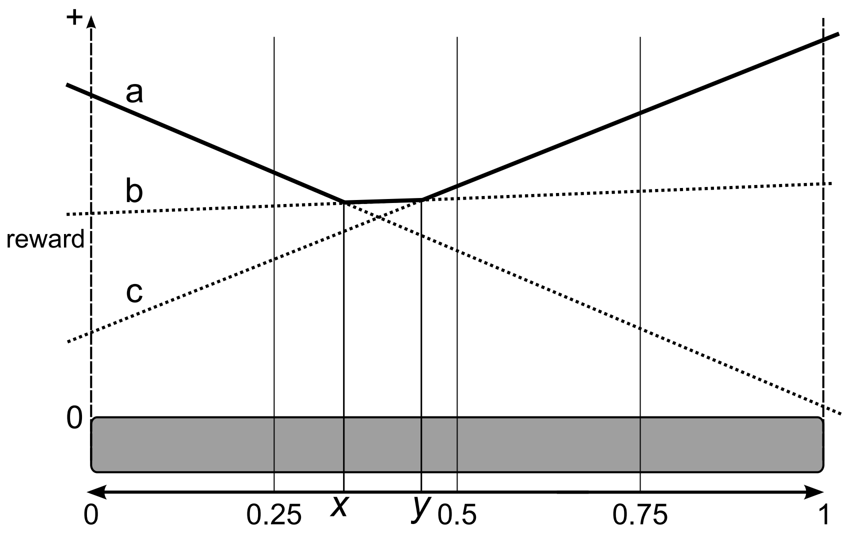


Figure 4.9: The highlighted upper surface shows the true optimal value function, consisting of 3 hyperplanes labelled a, b and c. With $d = 0.25$, all continuous points map to one of the five discrete numbered points. Points between x and y are hidden through rounding, preventing α-vector b from being found.

this case mostly due to the positioning of obstacles) leads to widely varying performance, making a statistical analysis of performance difficult. Appendix A shows the graphs and the sizes and number of uncertain edges present in each.

4.6.2.2 Discretisation results

The results in Table 4.11 show how varying the discretisation resolution affects the performance of the returned policy in larger graphs and how many states are required to describe the policy. Table 4.12 shows the solution times for the policies. Table 4.13 shows the effects of varying the resolution on the 3 edge graph.

In both the larger examples (Tables 4.11 and 4.12) and the 3 edge graph (Table 4.13), the effects of coarse discretisation on the policy quality can be seen. Across most of the graphs shown (Table 4.11), when $d > 0.005$, the average cost to reach the goal is unstable as d is altered, with some graphs showing an increased cost and others not. This indicates that the policy is no longer optimal. As explained above, if too much information is lost in discretisation, then key inflection points in the value function will be hidden, preventing LAO* from computing the optimal policy. Figure 4.9 shows an example of this where the optimal value function consists of the three α-vectors (see Section 2.3.2 for a full explanation of POMDP value functions): a,b and c. With $d = 0.25$, point x is rounded to 0.25 and point y to 0.5, thus hiding α-vector b because it is not optimal at either point. The discretised value function contains only vectors a and c, so the true optimal policy can never be discovered.

Average costs

d	Graph 1		Graph 2		Graph 3		Graph 4		Graph 5		Graph 6	
	I	D	I	D	I	D	I	D	I	D	I	D
0.1	1110.61	1058.80	1437.97	1438.35	933.55	872.15	1216.82	1210.07	1556.81	1711.90	1426.89	1340.31
0.05	1048.35	1015.56	1458.44	1443.90	968.80	899.62	1188.46	1184.85	1663.51	1700.95	1502.30	1411.14
0.01	1056.82	1020.00	1481.69	1471.74	993.66	915.50	1205.61	1205.14	F	F	1485.94	1414.59
0.005	1054.71	1019.92	1474.02	1466.52	984.32	909.49	1211.32	1211.07	F	F	1494.52	1419.03
0.001	1055.37	F	1476.43	1469.13	986.65	912.83	1211.63	1210.47	F	F	1500.36	1422.54
0.0005	1054.23	F	1476.10	1469.37	987.62	912.01	1211.29	1211.23	F	F	1499.98	1425.47
0.0001	1054.71	F	1475.85	1469.32	987.10	F	1211.42	1211.31	F	F	1500.63	1425.46
0.00005	1053.97	F	1476.05	1469.22	986.54	F	1211.57	1211.67	F	F	1499.92	1425.56
0.00001	1055.62	F	1476.30	1469.73	986.57	F	1211.48	1211.69	F	F	1499.64	1425.75
0.000005	1055.03	F	1476.32	1469.37	986.37	F	1211.55	1211.45	F	F	1499.48	1425.74
0.000001	1055.22	F	1475.35	1469.37	987.57	F	1211.49	1211.46	F	F	1499.76	1426.96
0.0000005	1055.72	F	1475.77	1469.17	987.22	F	1211.55	1211.74	F	F	1500.38	1425.32
0.0000001	1055.87	F	1475.58	1469.31	986.06	F	1211.59	1211.48	F	F	1500.33	1425.95

BSG size

d	Graph 1		Graph 2		Graph 3		Graph 4		Graph 5		Graph 6	
	I	D	I	D	I	D	I	D	I	D	I	D
0.1	101	31	137	12	127	58	145	63	313	133	126	43
0.05	119	110	137	21	54	46	145	63	285	7233	119	54
0.01	91	38	169	95	54	127	145	145	F	F	121	409
0.005	91	43	169	119	54	170	145	325	F	F	121	410
0.001	91	F	169	193	54	286	145	373	F	F	121	523
0.0005	91	F	169	200	54	502	145	608	F	F	121	554
0.0001	91	F	169	263	54	F	145	702	F	F	121	649
0.00005	91	F	169	248	54	F	145	898	F	F	121	622
0.00001	91	F	169	301	54	F	145	774	F	F	121	581
0.000005	91	F	169	304	54	F	145	826	F	F	121	585
0.000001	91	F	169	290	54	F	145	859	F	F	121	635
0.0000005	91	F	169	298	54	F	145	823	F	F	121	651
0.0000001	91	F	169	296	54	F	145	780	F	F	121	696

Table 4.11: Key: I=independent MDP model, D=dependent model. The top table shows how the average cost varies with discretisation resolution d, while the bottom table shows how the size of the best solution graph was affected. In both tables "F" indicates that LAO* failed to produce a policy.

Times (ms)

d	Graph 1		Graph 2		Graph 3		Graph 4		Graph 5		Graph 6	
	I	**D**	**I**	**D**	**I**	**D**	**I**	**D**	**I**	**D**	**I**	**D**
0.1	200	167	96	11	518	238	100	72	57263	4947	176	90
0.05	106	1222	120	27	357	565	98	76	182802	2182253	183	185
0.01	96	29234	152	281	525	12221	102	286	F	F	419	3626
0.005	114	62720	173	310	580	18795	103	610	F	F	512	5257
0.001	150	F	177	884	628	74501	98	1293	F	F	883	16276
0.0005	171	F	160	977	558	83350	108	1284	F	F	834	33273
0.0001	215	F	182	1832	527	F	100	1651	F	F	1092	76542
0.00005	251	F	186	1838	558	F	105	2081	F	F	1068	71932
0.00001	306	F	174	2263	555	F	103	1679	F	F	1410	116135
0.000005	422	F	185	2314	659	F	98	1949	F	F	1119	107808
0.000001	498	F	210	2457	629	F	113	2040	F	F	2020	148627
0.0000005	636	F	183	2379	580	F	110	2084	F	F	1530	162953
0.0000001	698	F	190	2454	584	F	114	1855	F	F	1788	192036

Table 4.12: Key: I=independent MDP model, D=dependent model. The solution times for examples in Table 4.11. "F" indicates that LAO* failed to produce a policy.

When resolution is increased enough for all the necessary inflection points to be found, the average cost stabilises. Past the point of stabilisation, further resolution increases serve no benefit—the cost does not decrease further. This indicates that the optimal policy has been found, although we cannot prove it. Examining the BSG size (Table 4.11, lower) and solution times (Table 4.12) show that in several cases, there is an inherent disadvantage to using a finer resolution than necessary.

Examining the dependent model for the 3 edge graph (Table 4.13), a clear increase in $|G|$[1] as d decreases (resolution becomes finer) can be seen. Performing linear regression on the runtime and $|G|$ shows a strong positive correlation. The coefficient in the independent model is 0.7290 and when the outlier at $d = 0.005$ is excluded, the correlation in the dependent model is 0.8080. The BSG is always a subset of G because it only includes states that will be visited by the optimal policy, and thus only has a very weak correlation with the execution time. The runtime of LAO* is clearly highly dependent on the amount of the MDP state space that is explored, but the cause is twofold. Firstly, computation time accrues in the creation and expansion of the graph, including evaluating observations encountered and computing the resulting belief states. Secondly, a significant fraction of the time is spent performing VI on the nodes. Recall that although the BSG contains a small subset of nodes from G, that subset is determined by continuously revising the nodes' cost and rejecting actions leading to higher cost nodes. The larger graphs all show upwards trends in solution time with resolution under both models, even after the resulting average cost has stabilised. More states tend to be expanded when finer resolutions are used, even though this may not improve the policy. The resolution must be selected carefully to minimise computation time while still enabling optimal policy to be found.

[1]G is the explicit belief state graph explored by LAO*, not to be confused with the PRM graph G_{PRM} that the MDP models are created from.

Independent model—3 edge graph

d	**Avg.cost**	**Max.cost**	**Std.dev**	$\lvert G\rvert, \lvert BSG\rvert$	**Time (ms)**
0.1	8.39	40	2.87	166,25	489
0.05	10.98	55	5.88	206,31	1044
0.025	7.85	11	1.34	236,19	441
0.01	8.73	21	3.15	747,34	1558
0.005	7.85	11	1.35	519,19	1067
0.0025	7.85	11	1.35	457,19	898
0.001	7.85	11	1.35	484,19	869
0.0005	7.85	11	1.35	1049,19	1255
0.00025	7.85	11	1.35	635,19	837
0.0001	7.85	11	1.35	1113,19	1273

Dependent model—3 edge graph

d	**Avg.cost**	**Max.cost**	**Std.dev**	$\lvert G\rvert, \lvert BSG\rvert$	**Time (ms)**
0.1	14.92	151	10.21	168,23	152
0.05	15.09	66	10.39	424,49	393
0.025	10.23	55	5.55	337,31	1156
0.01	8.28	13	1.95	1063,22	1259
0.005	8.65	21	3.10	1051,34	2033
0.0025	7.85	11	1.35	1136,19	1054
0.001	7.85	11	1.35	1179,19	1121
0.0005	7.85	11	1.35	1292,19	1461
0.00025	7.85	11	1.34	1396,19	1233
0.0001	7.85	11	1.35	1681,19	1471

Table 4.13: The effect of varying discretisation resolutions on the independent and dependent model MDPs for the 3 edge graph. $|G|$ is the size of LAO*'s explicit graph (the nodes explored during policy generation) and $|BSG|$ is the size of the best solution graph (the nodes that are part of the final policy).

For future experiments we use a constant discretisation of $d = 1 \times 10^{-5}$ because this appears to give a suitable balance between discretisation accuracy and computation time, allowing the optimal policy to be found.

Due to the way that rounding may hide some important belief points but not others, the level of discretisation can interact with the policy performance in unexpected ways. Several examples of this can be seen in the results. As the resolution increases, the BSG size and average cost of the 3 edge graph with the independent model generally decreases. An exception is at $d = 0.01$, where the returned policy performs worse; the same anomaly can be seen at $d = 0.005$ in the dependent model. Other anomalous results can be seen in Graphs 1, 4 and 6 when $d = 0.05$. Graph 6 is similar to the 3 edge problem where the average cost peaks even though resolution was increased from the previous step. Graphs 1 and 4 show a decrease in average cost at that resolution. The interaction between d and the final policy makes analysis difficult, but the most likely explanation is again founded

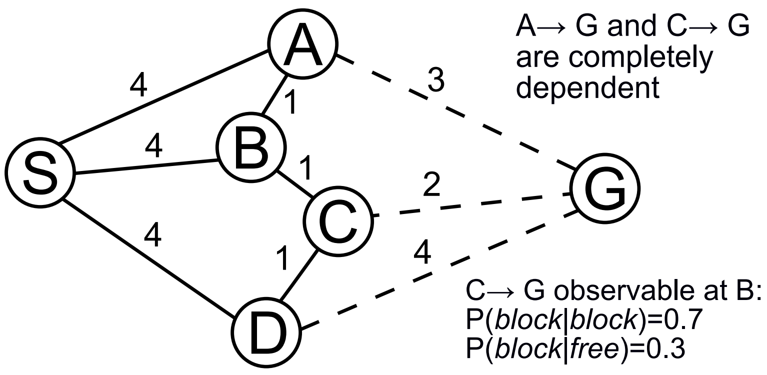

A→ G and C→ G are completely dependent
C→ G observable at B: P(*block*|*block*)=0.7 P(*block*|*free*)=0.3
A
B
C
D
S
G
4
4
4
1
1
1
3
2
4

Figure 4.10: The 3 edge graph. The upper two edges $A \to G$ and $C \to G$, are assumed to be completely dependent, i.e. their free statuses are perfectly correlated.

in the way some belief points will be lost in discretisation. At particular resolutions, it is likely only some of the required points can be discovered by the algorithm, leading to policies and costs that are slightly worse than those found at similar resolutions.

4.6.3 Dependency and clustering

In this Section, we examine how the three different MDP models perform on the range of larger example graphs. Before looking at the results, we use the 3 edge graph to demonstrate explicitly how the dependent model can gain a cost advantage over the independent model.

4.6.3.1 Analysis of dependency on a small example

Here, we show the optimal policies for the 3 edge graph (shown again in Figure 4.10 for convenience) under both the independent model and the dependent model. This allows us to clearly demonstrate the differences between the two policies and show why the dependent model gains an advantage. We assume that the upper two uncertain edges ($A \to G$ and $C \to G$) in the graph are completely dependent—their free statuses always match. The initial dependent belief state is configured so that there is a 50% probability of these two edges being free or blocked and the lower edge ($D \to G$) is independently blocked with a probability of 0.5.

Independent model Under the independent model, the belief state appears completely uniform so the agent believes every edge is blocked independently with a probability of 0.5. The agent does not know that the free statuses of edges $A \to G$ and $C \to G$ must match. The optimal policy can be described as a series of steps that the agent should execute as shown below:

1. Start at node S.

2. Go to node A. Receive observation of $A \rightarrow G$ at A. If $A \rightarrow G$ is free, then go to goal node G, otherwise go to step 3.

3. Go to node C via node B ignoring the observation at B.

4. Receive observation of $C \rightarrow G$ at node C. If $C \rightarrow G$ is free, then go to goal node G, otherwise go to step 5.

5. Go to D. Receive observation of $D \rightarrow G$ are node D. If $D \rightarrow G$ free, then go to goal G, otherwise decide that graph is unsolvable.

Visiting node A first makes intuitive sense since the path $S \rightarrow A \rightarrow G$ would incur a cost of 7 if the $A \rightarrow G$ edge is free. If this fails, the agent tries to reach the goal through the $C \rightarrow G$ edge, and if that is blocked, it lastly tries the $D \rightarrow G$ edge. Reaching G via $S \rightarrow D \rightarrow G$ costs 8, but travelling via $S \rightarrow C \rightarrow G$ costs 7 the same as the path via A, so why visit A first? The agent must also consider the costs when edges are blocked: if $A \rightarrow G$ and $C \rightarrow G$ are blocked then the agent incurs the cost of moving from A to C or D. If $D \rightarrow G$ is free then the order above minimises the cost. If instead, C was visited first from S and both $C \rightarrow G$ and its second choice $D \rightarrow G$ were blocked, it would then have to retrace its steps to reach A which would increase the eventual cost to reach the goal.

Dependent model Under the dependent model, the planner is able to find a better plan because of the dependency between edges $A \rightarrow G$ and $C \rightarrow G$:

1. Start at node S.

2. Go to node D. Receive observation of $D \rightarrow G$ at D. If $D \rightarrow G$ is free, then go to goal node G, otherwise go to step 3.

3. Go to C. Receive observation of $C \rightarrow G$ at C. If $C \rightarrow G$ is free, then go to goal node G, otherwise decide graph is unsolvable.

This may seem suboptimal, since the policy choices made do not always give the lowest cost of reaching the goal in all worlds. However, the expected cost to reach the goal is minimised. An obvious question might be: Why in the dependent model does the agent visit node D first, when the path through C has a lower cost to the goal? The free statuses of $C \rightarrow G$ and $D \rightarrow G$ are independent of each other, so surely the cheapest path should be tried first? The answer lies in what is optimal when all possibilities are considered. When the agent finds the edges open, then visiting C first *is* the cheapest path, but 50%

b_0	**MDP Model**	**Min.**	**Avg.**	**Max.**	**Std.dev**	**% fail**	$\lvert BSG \rvert$
Uniform	Independent	7	7.85	11	1.34	12.58	19
Uniform	Dependent	7	7.87	11	1.36	12.31	19
Dependent	Independent	7	8.33	11	1.88	25.02	19
Dependent	Dependent	7	7.67	8	0.47	24.74	10

Table 4.14: Comparison of policy costs for the 3 edge graph (Figure 4.10) under different starting belief states. Costs averaged over 50,000 trials, $d = 0.001$. With uniform b_0 all edges are independent, P(block)=0.5, with dependent b_0, edges $A \rightarrow G$ and $C \rightarrow G$ are fully dependent (either both blocked or both free).

of the time it will be blocked, as is edge $D \rightarrow G$, and this affects the optimal policy. To learn the free statuses of all 3 edges requires the agent to visit either A and D or C and D. Travelling from C to D is cheaper than A to D so we can rule out visiting A first completely, leaving us two choices: visit C then D or D then C (assuming the first is found blocked). The former ($S \rightarrow C \rightarrow D$) would cost 6 but the latter ($S \rightarrow D \rightarrow C$) only costs 5, so by visiting D, first we learn the same information for a lower cost. Since $D \rightarrow G$ is blocked 50% of the time it is cheaper on average to visit D first.

Table 4.14 shows the results of running the optimal policies from two initial belief states (b_0) under two different models in the simulator. In the uniform initial belief state, all edges are truly independent and have an equal probability of being blocked or free. From the dependent belief state, the upper two edges $A \rightarrow G$ and $C \rightarrow G$, are completely dependent and have an equal probability of being blocked or free together. The lower edge $D \rightarrow G$, is independently blocked with a probability of 0.5. From the uniform starting belief, all 8 (2^3) worlds may occur with equal probability, but for the dependent start state, the 4 worlds where only one of $A \rightarrow G$ and $C \rightarrow G$ is free and the other is blocked have zero probability of occurrence. Thus, the world where the agent cannot reach the goal because all edges are blocked has a higher probability of occurrence than with the uniform initial belief state. The independent model obtains the same policy from both the uniform and dependent starting belief since it does not model the dependency between the edges. However, this causes it to perform worse when the two upper edges are dependent. The dependent model in the uniform belief state performs identically to the independent model since the edges are independent. From the dependent start state, it obtains a lower average cost by exploiting the knowledge of the dependent edges. The reduction in possible worlds is evident in the last row of the table where the BSG for the policy is half the size of the others. Belief states where the free statuses of edges $A \rightarrow G$ and $C \rightarrow G$ are not matched cannot be reached so are not included in the BSG that describes the optimal policy.

4.6.3.2 Replanning MCC

For the experiments in this research, we use a modified version of the MCC planner (see Section 3.2.3). MCC is an offline planner that computes a static route that cannot change during plan execution. This means that during simulation, if the agent following an MCC policy encountered a blocked edge, it would not be aware of it and would fail for trying to traverse it. This is a consequence of not being able to make any observations, but does not make for a very fair comparison between it and other planners that gain information during plan execution. We make a simple modification that allows the MCC planner to replan its route during execution if it senses a blocked edge. During simulation, the agent checks the free status of the immediate next edge in its route to see if it is blocked (similar to local sensing in FSSN). If it is, it marks the edge as blocked and invokes the MCC planner to compute a new route to the goal. The computation time of any replanning is incorporated in the reported times for the MCC planner.

4.6.3.3 Symbolic Perseus

The nature of the state space and the use of multiple probability distributions in the independent and clustered MDP models means that the graphs used in this research lend themselves well to factored representations. The set of variables which define a state comprise one variable for the agent's location, followed by a collection of variables describing the uncertain edge statuses—one variable per cluster. The set of values for a variable is the set of possible free statuses for the edges in that cluster so there will be 2^n values for a cluster of n edges. We use the Symbolic Perseus solver (see Section 3.4.1.1) to solve factored POMDP versions of the independent model of the graphs and compare the results to the other algorithms. The agent receives no reward at the goal and cannot transition to other states, thus making it an absorbing state. In other states, it receives a large penalty if it tries to execute an invalid action, or chooses to remain at the same PRM node. An example of how a graph is described in the ADD format used by Symbolic Perseus can be found in Appendix B.

4.6.3.4 Dependency in larger graphs

Table 4.15 shows the results from running the three MDP models with LAO* on each graph. The discretisation for LAO* was $d = 1 \times 10^{-5}$ and $\gamma = 0.999$ with a memory limit of 500MB. All graphs were pre-processed as described earlier to remove graph nodes that cannot affect the policy or cost. Perseus, Symbolic Perseus and the MCC planner were also compared on each graph. For Perseus, the times are quoted in hours and minutes instead of milliseconds for convenience (belief pool generation times are not included in computation time); the belief pool was sized at 50,000 for each graph with $\gamma = 0.95$ and $\epsilon = 1 \times 10^{-5}$. A time limit of 60,000 seconds was placed on the computation time for PERSEUS. This is approximately 16 hours and 40 minutes, although PERSEUS time limits are not honoured exactly, leading to VI continuing for a few minutes past the limit. Symbolic PERSEUS uses a discount factor of $\gamma = 0.999$ in line with LAO*, with defaults used for other settings except the belief pool size, which was adjusted to the sizes shown in the "Alg." column. As with standard PERSEUS, the generation of the belief pool is not included in the computation time. For MCC, $C_{const} = 300$ and replanning was allowed to prevent MCC failing on all trials where it encountered a blocked edge in its route.

Graph	Alg.	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
Graph 1	I	916.26	1055.47	2278.00	331.39	0	325
	C	916.26	1019.44	1929.02	229.79	0	51274
	D	F	F	F	F	F	F
	MCC	916.26	1054.92	2278.00	330.69	0	2
	PRS	947.19	1024.90	2036.09	252.72	0	7h40m
	Sym.1000	916.26	1030.88	2114.35	274.02	0	35260
	Sym.5000	916.26	1031.45	2114.35	275.30	0	43380
	Sym.50000	916.26	1029.15	2114.35	271.11	0	61610
Graph 2	I	1350.84	1475.85	2096.80	188.52	0	162
	C	1350.84	1476.19	2096.80	188.82	0	220
	D	1350.84	1469.56	1992.75	173.22	0	2276
	MCC	1350.84	1521.93	2124.93	252.10	0	1
	PRS	1813.35	1813.35	1813.35	0	0	2h1m
	Sym.1000	1350.84	1476.38	2096.80	188.81	0	46590
	Sym.5000	1350.84	1476.37	2096.80	189.30	0	50710
	Sym.50000	1350.84	1476.65	2096.80	189.50	0	48680
Graph 3	I	646.92	986.23	1430.84	365.00	0	545
	C	646.92	911.73	1360.16	254.04	0	36905
	D	F	F	F	F	F	F
	MCC	646.92	987.19	1430.84	365.19	0	2
	PRS	646.92	913.41	1351.74	252.69	0	3h23m
	Sym.1000	646.92	912.81	1360.15	254.15	0	51350
	Sym.5000	646.92	911.87	1360.15	253.51	0	66680
	Sym.50000	646.92	911.55	1360.15	253.61	0	75340
Graph 4	I	1153.58	1211.50	1452.10	110.24	0	94
	C	1153.58	1211.65	1452.10	110.37	0	228
	D	1153.58	1211.46	1452.10	110.20	0	2022
	MCC	1165.31	1217.08	1451.34	110.12	0	1
	PRS	1152.82	1210.75	1451.34	110.27	0	2h36m
	Sym.1000	1153.57	1211.09	1452.10	109.92	0	36420
	Sym.5000	1153.57	1211.93	1452.10	110.54	0	39080
	Sym.50000	1153.57	1211.20	1452.10	110.02	0	38600

Table 4.15: Key: I=independent, C=clustered, D=dependent model. "Cr" indicates a crash. "F" indicates that LAO* failed to produce a policy.

Graph	Alg.	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
Graph 5	I	F	F	F	F	F	F
	C	F	F	F	F	F	F
	D	F	F	F	F	F	F
	MCC	1529.18	1649.67	1890.62	170.3	24.97	2
	PRS	1433.93	1569.6	1880.27	189.77	31.93	13h33m
	Sym.1000	1436.55	1460.5	1531.80	41.32	50.06	85740
	Sym.5000	1433.93	1457.74	1529.18	41.24	49.66	166470
	Sym.50000	1436.55	1460.52	1531.80	41.33	50.49	177670
Graph 6	I	1054.93	1500.55	2215.15	445.71	0	1534
	C	1054.93	1426.42	2123.38	370.62	0	6032
	D	1054.93	1425.75	2123.38	370.54	0	116135
	MCC	1128.31	1525.04	2062.04	422.38	0	1
	PRS	1054.93	1425.04	2123.38	370.07	0	10h48m
	Sym.1000	1054.93	1388.75	2123.38	370.07	9.82	111800
	Sym.5000	1054.93	1390.23	2123.38	369.60	10.00	115920
	Sym.50000	1054.93	1386.72	2123.38	369.18	10.06	125150
Graph 7	I	861.99	1120.81	1340.49	237.81	0	38
	C	861.99	1121.35	1340.49	237.77	0	56
	D	861.99	1121.61	1340.49	237.74	0	318
	MCC	861.99	1142.57	1372.37	248.47	0	3
	PRS	1336.75	1336.75	1336.75	0	0	0h5m
	Sym.1000	861.98	1121.76	1340.49	237.73	0	15040
	Sym.5000	861.98	1122.85	1340.49	237.63	0	14970
	Sym.50000	861.98	1121.79	1340.49	237.73	0	11980
Graph 8	I	F	F	F	F	F	F
	C	903.13	1092.22	1911.36	379.36	0	28931
	D	F	F	F	F	F	F
	MCC	903.13	1092.24	1911.36	379.46	0	2
	PRS	903.12	1091.94	1911.35	378.98	0	0h14m
	Sym.1000	903.12	1092.13	1911.35	379.33	0	13190
	Sym.5000	903.12	1090.22	1911.35	378.13	0	12690
	Sym.50000	903.12	1092.55	1911.35	379.54	0	11710
Graph 9	I	775.28	1134.47	2274.68	566.44	0	912
	C	775.28	1045.80	1701.29	302.98	0	1048
	D	775.28	1043.69	1462.23	300.42	0	50679
	MCC	775.28	1199.46	2274.68	624.70	0	6
	PRS	775.28	1191.14	1382.73	264.45	0	16h40m
	Sym.1000	775.28	1042.79	1462.22	302.57	0	73360
	Sym.5000	775.28	1044.79	1462.22	300.55	0	84280
	Sym.50000	775.28	1043.31	1462.22	300.52	0	96650

Table 4.15: Key: I=independent, C=clustered, D=dependent model. “Cr” indicates a crash. “F” indicates that LAO* failed to produce a policy.

Graph	Alg.	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
Graph 10	I	680.46	844.68	1350.08	259.59	0	172
	C	678.07	838.72	1350.60	218.86	0	2482
	D	678.07	839.09	1350.60	218.99	0	35396
	MCC	678.07	866.02	1407.66	282.31	0	5
	PRS	1005.16	1005.16	1005.16	0	0	3h16m
	Sym.1000	680.45	839.10	1350.07	219.38	0	16840
	Sym.5000	680.45	838.76	1350.07	218.89	0	17120
	Sym.50000	680.45	840.15	1350.07	219.10	0	18370
Graph 11	I	1072.92	1218.89	1909.04	314.17	0	16546
	C	F	F	F	F	F	F
	D	F	F	F	F	F	F
	MCC	1072.92	1219.20	1909.04	314.48	0	1
	PRS	-	-	-	-	100	16h43m Cr
	Sym.1000	1072.92	1155.07	1909.04	244.45	8.46	96010
	Sym.5000	1072.92	1219.86	1909.04	314.86	0	107850
	Sym.50000	1072.92	1220.07	1909.04	315.14	0	161190
Graph 12	I	1329.92	1329.92	1329.92	0	0	3164
	C	1329.92	1329.92	1329.92	0	0	3342
	D	F	F	F	F	F	F
	MCC	801.31	1504.07	2527.64	728.40	0	3
	PRS	1329.92	1329.92	1329.92	0	0	14h46m
	Sym.1000	1329.92	1329.92	1329.92	0	0	146570
	Sym.5000	1329.92	1329.92	1329.92	0	0	424530
	Sym.50000	1329.92	1329.92	1329.92	0	0	591540
Graph 13	I	F	F	F	F	F	F
	C	F	F	F	F	F	F
	D	F	F	F	F	F	F
	MCC	694.79	1374.38	2063.96	588.06	0	17
	PRS	F	F	F	F	F	F
	Sym.1000	-	-	-	-	100	953640
	Sym.5000	694.79	781.17	1587.72	122.78	61.71	3117970
	Sym.50000	694.79	757.07	1587.72	113.04	72.07	11036490
Graph 14	I	F	F	F	F	F	F
	C	F	F	F	F	F	F
	D	F	F	F	F	F	F
	MCC	1162.89	1188.00	1231.48	33.04	91.00	1
	PRS	-	-	-	-	100	0h24m
	Sym.1000	-	-	-	-	100	39610
	Sym.5000	-	-	-	-	100	608910
	Sym.50000	-	-	-	-	100	1357150

Table 4.15: Key: I=independent, C=clustered, D=dependent model. "Cr" indicates a crash. "F" indicates that LAO* failed to produce a policy.

Graph	Alg.	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
Graph 15	I	778.82	1140.44	2022.34	526.96	0	1141
	C	778.82	1140.35	2022.34	526.89	0	1465
	D	778.82	1137.60	2022.34	525.77	0	17870
	MCC	778.82	1211.70	3026.55	824.22	5	3
	Prs	1346.38	1346.38	1346.38	0	0	4h51m
	Sym.1000	778.81	785.99	1187.84	37.96	30.92	17230
	Sym.5000	778.81	785.82	1187.84	37.14	31.14	20340
	Sym.50000	778.81	785.60	1187.84	35.57	30.97	22310
Graph 16	I	1795.23	1795.23	1795.23	0	0	27600
	C	1795.23	1795.23	1795.23	0	0	47317
	D	F	F	F	F	F	F
	MCC	628.09	2369.04	2996.09	815.01	0	5
	Prs	1795.23	1795.23	1795.23	0	0	5h21m
	Sym.1000	628.09	628.09	628.09	0	86.51	10160
	Sym.5000	628.09	628.09	628.09	0	86.41	8740
	Sym.50000	628.09	628.09	628.09	0	86.45	9540
Graph 17	I	1403.94	1724.62	2192.85	385.75	0	626
	C	1403.94	1720.54	2192.85	373.42	0	6390
	D	F	F	F	F	F	F
	MCC	1403.94	1724.87	2192.86	385.82	0	2
	Prs	1403.93	1721.33	2192.85	373.53	0	11h47m
	Sym.1000	1403.93	1720.86	2192.85	373.58	0	53280
	Sym.5000	1403.93	1723.09	2192.85	373.87	0	55190
	Sym.50000	1403.93	1723.09	2192.85	373.85	0	52620
Graph 18	I	1050.69	1120.11	1663.72	153.33	0	43
	C	1050.69	1120.26	1663.72	153.63	0	45
	D	1050.69	1120.09	1663.72	153.22	0	590
	MCC	1050.69	1123.22	1663.72	182.11	0	2
	Prs	1128.47	1128.47	1128.47	0	0	16h43m
	Sym.1000	-	-	-	-	100	580870
	Sym.5000	-	-	-	-	100	1661440
	Sym.50000	-	-	-	-	100	3348150
Graph 19	I	1174.51	1337.02	1621.77	213.81	0	51
	C	1174.51	1311.79	1621.77	176.70	0	62
	D	1174.51	1311.54	1621.77	176.64	0	478
	MCC	1174.51	1406.32	2456.24	438.10	0	2
	Prs	-	-	-	-	100	16h42m Cr
	Sym.1000	1174.50	1310.92	1621.76	176.36	0	49020
	Sym.5000	1174.50	1336.58	1621.76	213.75	0	47810
	Sym.50000	1174.50	1311.56	1621.76	176.89	0	46070

Table 4.15: Key: I=independent, C=clustered, D=dependent model. "Cr" indicates a crash. "F" indicates that LAO* failed to produce a policy.

Graph	Alg.	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
Graph 20	I	F	F	F	F	F	F
	C	F	F	F	F	F	F
	D	F	F	F	F	F	F
	MCC	784.98	1303.86	2707.60	688.74	0	2
	PRS	784.98	894.29	1897.92	331.23	41.31	15h1m
	Sym.1000	-	-	-	-	100	25180
	Sym.5000	-	-	-	-	100	17190
	Sym.50000	-	-	-	-	100	17280

Table 4.15: Key: I=independent, C=clustered, D=dependent model. "Cr" indicates a crash. "F" indicates that LAO* failed to produce a policy. Comparison of 3 MDP models, MCC, PERSEUS and Symbolic PERSEUS across range of example graphs. All times averaged over 10 runs. $d = 0.00001$ and graph pre-processor used for all MDP models, PERSEUS and Symbolic PERSEUS. All costs derived from 50,000 simulator trials.

Compared to the smaller examples earlier in this chapter, the variance in the average costs in Table 4.15 are much greater. This is to be expected because the individual edge costs have a maximum cost of 300, contrasting with a maximum cost of 5 in the graphs of Figure 4.8. While the high number of trials run per simulation allows us to be confident about the accuracy of the average cost reported, the real significance of small differences is lower. For instance, in Table 4.14, the maximum difference in average cost of 0.66 across all tests indicates a difference in policy quality, whereas with larger examples the same difference is negligible.

The first prominent aspect of the results in Table 4.15 is the number of tests where policy generation failed (marked 'F'). This happens when the LAO* implementation ran out of memory before the optimal policy was found. As mentioned in Section 4.6.2, the BSG only constitutes a small subset of the nodes explored in the explicit graph G. In the cases where LAO* fails, $250,000 \leq |G| \leq 400,000$ depending on the number of elements forming the belief state. This is in turn determined by the arrangement of the uncertain edges into clusters. The quantity of states requiring expansion to find the optimal policy is entirely dependent upon the individual graph. Graphs where multiple partially expanded routes appear to have similar costs are more likely to fail because LAO* must explore all paths through the MDP until they are either proved non-optimal or the optimal policy is discovered. If similar cost paths are discovered, more of $\mathcal{S}$ is likely to be explored. The dependent model is also more likely to fail than the other two models since the MDP state space is exponentially larger and the quantity of elements required to represent one belief state is greater, requiring more memory per belief state. The results highlight one advantage of the clustered model, as policies can be generated for the clustered model but not the dependent model in 5 of the graphs tested.

Perseus Perseus is vastly slower than all other algorithms tested, though the times vary greatly depending on the particular graph. For some of the graphs, the maximum computation time of 60,000 seconds (16 hours and 40 minutes) is being reached causing Perseus to terminate. In Graphs 11 and 19, Perseus terminated after 60,000 seconds, but the policy file produced contained α-vectors with coefficients of enormous magnitude which caused the simulator to crash upon parsing the file. Although the solution times for LAO* also vary according to the graph, there is only a weak correlation of 0.22 between the times for the dependent model (the closest approximation to the POMDP) and Perseus. This is to be expected due to the major differences in the algorithms.

In terms of performance, the average cost of the policies computed by Perseus match the costs obtained via the dependent MDP model. This is what we expect to see because the POMDP formulation and dependent MDP are the most similar in their representation of the belief state. In the other Graphs, the POMDP has a higher average cost than any of the MDP models. This is surprising, since the POMDP can represent any edge dependencies present, therefore it should be able to match the dependent MDP in performance. The fact that we use a discount factor of $\gamma = 0.95$ is the likely cause of the poor performance. Despite being close to 1, the discount factor is still applying too much bias against the long-term cost to find the policy that is found by LAO*. To test this, we re-ran the graphs where the POMDP policy performed badly, this time with $\gamma = 0.999$, as used by LAO*. Unfortunately, only the POMDP policies on Graphs 7 and 15 improved to match the dependent MDP. All of the other graphs tested either failed to reach the goal in 100% of the trials or gained an average cost far worse than with $\gamma = 0.95$.

Table 4.16 shows the results of re-running Perseus on the graphs where it performed badly, but with a discount factor of $\gamma = 0.99$. The first thing to note is that Perseus failed to converge for all graphs except numbers 7 and 14, instead terminating at the maximum time of 60,000 seconds. The increased discount factor means that the agent is considering the future cost more when considering the cost of an action. It also means that more dynamic programming iterations are likely to be required for convergence, thus increasing the runtime. We see improvements in the average cost of Graphs 7 and 10 to the point where they match the performance of the dependent MDP model. Graph 15 also improves significantly, but the average cost of 1153.97 is still slightly higher than the dependent model at 1137.60. The other POMDPs either show no improvement, or continue to fail (Graphs 11,13,14 and 19). Graphs 2 and 20 show that different policies are being found under the higher discount factor, but these are not necessarily beneficial. In Graph 2, we see an extremely high failure rate when $\gamma = 0.99$ compared to $\gamma = 0.95$. In Graph 20, we see a reduction in the failure rate when $\gamma = 0.99$ at the expense of much higher average and maximum costs. The exceptionally high maximum cost indicates that

Graph	γ	Min.	Avg.	Max.	Std.dev	% fail	Time
Graph 2	0.95	1813.35	1813.35	1813.35	0	0	2h1m
	0.99	1478.78	1795.65	1820.51	87.63	90.7	16h41m
Graph 7	0.95	1336.75	1336.75	1336.75	0	0	0h5m
	0.99	861.98	1126.67	1340.49	232.47	0	0h32m
Graph 10	0.95	1005.16	1005.16	1005.16	0	0	3h16m
	0.99	680.45	838.02	1350.07	218.71	0	16h41m
Graph 11	0.95	-	-	-	-	100	16h43m
	0.99	F	F	F	F	F	11h5m Cr
Graph 13	0.95	F	F	F	F	F	F
	0.99	-	-	-	-	100	16h42m
Graph 14	0.95	-	-	-	-	100	0h24m
	0.99	-	-	-	-	100	2h24m
Graph 15	0.95	1346.38	1346.38	1346.38	0	0	4h51m
	0.99	778.81	1153.97	2022.33	516.66	0	16h42m
Graph 19	0.95	-	-	-	-	100	16h42m
	0.99	F	F	F	F	F	F
Graph 20	0.95	784.98	894.29	1897.92	331.23	41.31	15h1m
	0.99	784.98	4910.58	12076.41	5343.2	9.37	16h41m

Table 4.16: POMDP policy performance when $\gamma = 0.99$. POMDP results from Table 4.15 repeated for comparison. Reported times at $\gamma = 0.99$ are for a single run. "Cr" in Graph 11 indicates a crash. "F" indicates the POMDP failed to produce a usable policy.

Graph	γ	Min.	Avg.	Max.	Std.dev	% fail	Time
Graph 1	0.95	947.19	1024.90	2036.09	252.72	0	7h40m
	0.999	953.99	961.66	2141.27	81.01	31.26	7 days
Graph 6	0.95	1054.93	1425.04	2123.38	370.07	0	10h48m
	0.999	1054.93	1163.44	1988.66	250.91	61.18	7 days

Table 4.17: POMDP policy performance when $\gamma = 0.999$. POMDP results from Table 4.15 repeated for comparison. Reported times at $\gamma = 0.999$ are for a single run.

the policy is poor and the agent is essentially stuck in a loop in the graph for many steps before going to the goal. We also ran PERSEUS on Graphs 1 and 6 with $\gamma = 0.999$ with a belief pool size of 5,000, but increased the maximum time limit from 60,000 seconds to a week (604,800 seconds). Neither of the runs converged in this time, only terminating when the limit was reached. Table 4.17 shows the results obtained. Both policies incur a substantial rate of failure. Adjusting the discount factor has led to moderate improvement in some of these graphs. However, in general, the computation times for the POMDP formulation show that a fully dependent, non-structured approach as used with standard PERSEUS is not a feasible approach to this problem.

Graph	Solution time		Graph	Solution time	
	$\lvert B\rvert = 5,000$	$\lvert B\rvert = 50,000$		$\lvert B\rvert = 5,000$	$\lvert B\rvert = 50,000$
1	5h26m	7h40m	11	16h41m	16h43m
2	1h18m	2h1m	12	9h20m	14h46m
3	1h30m	3h23m	13	16h41m	F
4	1h10m	2h36m	14	0h8m	0h24m
5	7h50m	13h33m	15	2h38m	4h51m
6	7h10m	10h48m	16	1h34m	5h21m
7	0h4m	0h5m	17	7h17m	11h47m
8	0h12m	0h14m	18	11h31m	16h43m
9	F	16h40m	19	16h41m	16h42m
10	3h9m	3h16m	20	8h19m	15h1m
Mean ratio of computation time				0.62:1	

Table 4.18: Perseus solution times with a belief pool size of 5,000 and 50,000 beliefs. All other parameters kept the same. 'F" indicates the POMDP failed to produce a usable policy.

Graph	Min.	Avg.	Max.	Std.dev	% fail	Time
9	F	F	F	F	F	F
13	725.88	1397.34	1907.79	302.75	1.08	16h41m
16	628.09	628.09	628.09	0	86.50	1h34m
17	1472.44	1818.52	2225.54	375.32	0	7h17m

Table 4.19: Results for Perseus when the belief pool is sized at 5,000. Only graphs where results differ from Table 4.15 are shown.

Perseus with fewer belief states The size of the belief pool B, used with Perseus is one factor which can effect the long solution times. To test this, we also ran Perseus on the 20 graphs with a belief pool of 5,000 beliefs and compared the solution times. These can be seen along with the times from using 50,000 beliefs (from Table 4.15) in Table 4.18. The performance of the policies was largely the same between the two experiments. This shows that, with the exceptions shown in Table 4.19, a belief pool of 50,000 beliefs is not necessary to find the optimal policy with Perseus. With 5,000 beliefs, Perseus finished a mean of 34% quicker than before. Graph 16 showed the largest difference in computation time, though the policy is worse than with 5,000 beliefs. With 5,000 beliefs, Perseus performs the same as Symbolic Perseus on that graph. Graph 17 is similar except that the average policy cost has increased. Graph 9 no longer produces a policy as Perseus crashed due to running out of memory. Graph 13 shows where decreasing the size of the belief pool can have an advantage, since it now produces a policy whereas originally it did not.

Symbolic Perseus The results from Symbolic Perseus make it clear that the factored version of the POMDP is far quicker to solve. All problems are solved in the order of seconds instead of hours, though LAO* is still quicker to produce policies in nearly all cases by a large margin. Symbolic Perseus is written in MATLAB and Java as opposed to pure Java like LAO*, so it is unreasonable to compare runtimes closely due to the implementation differences. However, we can see that the performance of Symbolic Perseus is close to the dependent MDP model under LAO* in Graphs 1,6 and 15, and is superior in Graphs 10 and 8 (under the clustered model). It is actually much quicker in Graph 16, though it produces a policy that fails to reach the goal most of the time.

Changing the number of belief states sampled by Symbolic Perseus affects the runtime as could be expected in many of the graphs. The solution times increase massively in some graphs (Graphs 9,11,12,13,14 and 18) when the size of belief pool increases. In others, it only increases slightly or remains unaffected. The graphs where the belief pool size has a large effect tend to be the graphs that are harder to solve. Graphs 11–14 are good examples of this since dependent LAO* fails to produce a policy in all four cases and Symbolic Perseus fails in two. We can also see in two cases that a belief pool size of 1,000 was insufficient to find a good policy. In Graphs 11 and 13 the rate of failure decreases once the belief pool is increased to 5,000 beliefs.

The performance of the policies is good and matches LAO* in the majority of cases. With the exception of Graphs 16,18 and 20, Symbolic Perseus provides a good policy. In Graphs 6,15 and 16 it produces a policy that reaches the goal in some of the trials but not all, thus incurring a higher failure rate than LAO*. The method used by Symbolic Perseus for sampling belief states is the likely reason for this. Changing the belief sampling method from "QMDP" to "random" (as used by standard Perseus) improved the policy performance for Graph 18 but had no effect on the other graphs. An interesting feature of the results is that despite using an independent belief model, the performance matches that of dependent LAO* in some cases where independent LAO* does not perform as well. The use of ADDs in Symbolic Perseus is a possible cause because the merging of approximately identical sub-branches in an ADD could be allowing it to find a superior policy.

Policy behaviour and LAO* performance Examining the average costs across all the graphs in Table 4.15, a distinct pattern in the results emerges. Two classes of behaviour are seen: in many graphs, all three MDP models achieve almost the same performance (Graphs 2,4,7,12,15,16,17,18) while in others, the clustered model surpasses the independent model by varying amounts (Graphs 1,3,6,9,10,19). In the latter category, the computed clustered/dependent policies are exploiting knowledge of edge dependen-

Graph	Alg.	$\|G\|$	$\|BSG\|$	Time	Graph	Alg.	$\|G\|$	$\|BSG\|$	Time
	I	1,355	91	325		I	88,005	161	16546
1	C	287,853	45	51274	11	C	~235,000	~8,000	F
	D	~320,000	~1,200	F		D	≤20,000	N/A	F
	I	1,116	169	162		I	17,010	7	3164
2	C	1,546	216	220	12	C	18,234	7	3342
	D	7,225	320	2276		D	~332,000	~8,000	F
	I	3,993	54	545		I	~188,000	~22,000	F
3	C	198,120	251	36905	13	C	~282,000	~20,000	F
	D	~281,000	~800	F		D	~180,000	~11,000	F
	I	540	145	94		I	~270,000	~22,000	F
4	C	1,089	273	228	14	C	~415,000	~16,000	F
	D	4,038	883	2022		D	~467,000	~12,000	F
	I	~254,000	~4000	F		I	8,428	69	1141
5	C	~301,000	~11,000	F	15	C	11,512	69	1465
	D	~365,000	~5,000	F		D	61,364	189	17870
	I	5,739	121	1534		I	134,204	7	27600
6	C	24,361	150	6032	16	C	237,889	7	47317
	D	73,494	581	116135		D	~213,000	~2,000	F
	I	430	71	38		I	4,239	71	626
7	C	715	71	56	17	C	47,342	385	6390
	D	2,324	77	318		D	~308,000	~2,000	F
	I	~248,000	~4000	F		I	330	48	43
8	C	297,562	16	28931	18	C	330	48	45
	D	~370,000	~500	F		D	2,130	194	590
	I	4,412	95	912		I	523	133	51
9	C	5,923	73	1048	19	C	695	185	62
	D	76,084	139	50679		D	2,251	546	478
	I	942	24	172		I	~196,000	~10,000	F
10	C	13,395	44	2482	20	C	~245,242	~2,669	F
	D	71,151	75	35396		D	~246,000	~6,000	F

Table 4.20: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy. The number of nodes explored by LAO* and number of nodes in the BSG for the graphs in Table 4.15. Times (ms) are re-produced for convenience.

cies in the graph to find more efficient routes to the goal. In the other graphs where the performance remains unchanged across all three models, while edge dependencies are present the computed policies do not exploit any dependency. This indicates that there is no advantage to knowing about the dependencies or that the dependent edges do not feature on the optimal route, otherwise we would expect the dependent model to have a lower average cost. Even if edges are dependent, if the prior probability of them being blocked is high then they will likely not be included on the optimal route—any benefit is outweighed by the risk of them being blocked.

The solution times for the MDP models across all graphs differ greatly from very low solution times as in Graph 18, to much greater times such as those of the clustered model in Graph 1 or the dependent model in Graph 9. The number of states expanded in G is the major contributing factor to the solution time and provides a good indicator of how much LAO* had to explore the state space to find the BSG. Table 4.20 shows the size of the explicit graphs and the best solution graphs for the computed LAO* policies for the graphs in Table 4.15. Only the figures for complete policies are exact because complete information is not reported when LAO* fails—the final policy size is never known. As is the case when LAO* fails, if several branches in the graph appear to have similar utility (these can alter during the VI convergence phase), this tends to cause greater exploration of the state space, i.e. more expansion in G which will clearly increase the solution time as well. Graphs 12 and 15 show good examples of this. The average costs of Graphs 12 and 15 show in both cases that, although there is no inherent advantage to the clustered or dependent models, LAO* likely still explored a relatively large number of nodes in the independent model. The independent solution time for Graphs 12 and 15 are higher than for many other graphs where the solution time is often under a second. Looking at Table 4.20, $|G|$ shows that both the independent and clustered models explored similar amounts of the MDP state space before finding the same size BSG. In Graph 15, the vastly increased state space explored in the dependent model is also reflected in the runtime. Conversely, in graphs where the solution time is small, this indicates that LAO* was able to find the BSG with little exploration, suggesting that the heuristic was directing graph expansion towards the optimal policy.

The timings for the clustered model show it always requires at least as much time as the independent model (Graphs 12,17 and 18) and is often slower. Across all graphs, clustering is on average, 19.0 times slower (median=1.6) to compute the optimal policy compared to independent; ignoring outliers more than 1 standard deviation away, the figure lowers to a mean of 3.4 times slower (median=1.4). The dependent model is equivalently 35.1 times slower (median=13.9) than the independent model, 16.2 times slower (median=13.7) once outliers are excluded. The increased dimensionality of dependent belief states also contributes to the increased time for that model because of the extra computation required to expand a state in the explicit graph. This is reflected across many examples where, even though policy performance is almost identical, the dependent model takes longer to complete. This is also true of the clustered model to a lesser extent because it exists in between the extremes of the other two models in terms of belief state dimensionality.

MCC In all cases, the MCC planner is the fastest by a wide margin, but policy performance never surpasses the independent MDP. The MCC planner treats each edge independently ignoring any dependencies—observing an edge never gives the agent any information about any other edge. Since MCC never considers the value of future information, it can rarely find an optimal policy. In some graphs (Graphs 1,3,11 and 17) it matches the performance of the independent model, which suggests that the optimal policy under the assumption of independence is first to try the shortest route to the goal, then attempt successively longer routes if blocked edges are found. In other graphs (Graphs 2,9,12,15 and 16), MCC performs significantly worse than the independent model. This shows that the simple behaviour of the MCC algorithm cannot match the benefit provided by considering the information gain of noisy observations, or by including the cost of blocked edges in deriving a policy. MCC also requires selecting a suitable C_{const}, which is highly problem dependent, and lacks a simple way to choose 'good' values to guarantee performance. High values make the planner avoid most uncertain edges, preferring safe routes with higher costs. Low values make the planner too optimistic causing performance degradation due to frequent replanning causing the agent to retrace portions of its route. Evidence for this can be seen in Graphs 12,15,16 and 19 where the maximum cost for MCC is much higher compared to other algorithms.

Failure rates Various graphs have failed trials recorded under some of the algorithms tested. Since for some of the graphs, the percentage of failure was substantial and the failed runs are not included in the average cost, the results for those graphs are shown with a cost of failure included in Table 4.21. A fixed cost was given to trials where the agent did not successfully reach the goal and this cost is included in the average cost in Table 4.21. By altering the cost of failure, it is possible to affect the perceived performance of various algorithms. Assigning a cost of 0 to failure would entail that an algorithm which failed in all trials would receive an average of 0. Conversely, assigning an extremely high cost to failure would mean that an algorithm which outperformed others but occasionally failed would have an unfairly inflated average cost. We set the cost of failure to 3,500 as this is greater than the maximum cost to reach the goal in any of the graphs tested and is a good heuristic which will not overemphasise the failed trials in the average cost. The results in Table 4.21 show the average cost with and without failed trials included along with the percentage of failed trials. Several of the results show that when failed trials are included the ordering of the algorithms does change. In Graph 5 the order changes completely, with MCC giving the lowest average cost when failures are included, as opposed to the highest cost before. In Graph 13 the ordering of the Symbolic Perseus tests with 5,000 and 50,000 belief pool sizes switch positions and in Graph 20, Perseus is shown to be performing worse than MCC once failures are included.

Anomalies There are some notable anomalies in the Table 4.15. Firstly in Graph 8, the independent and dependent model fail, but the clustered model produces a result. This is counter-intuitive because, we would expect the independent model to also succeed due to the lower dimensionality of its belief space. Looking at the data in Table 4.20, we can see that the amount of the explicit graph explored is very high for all models; this is expected since LAO* failed from a lack of memory, however the size of the BSG provides better insight. In many cases, the BSG will expand as LAO* explores and the policy becomes more complex, but near the end of the computation we often observe a decrease in $|BSG|$ as LAO* rules out sub-optimal sections of G. This effect is very pronounced here as the clustered BSG contains only 16 states. Therefore, the speculative cause of the independent model's failure was that memory ran out before this contraction of the BSG could occur. Knowledge of edge dependencies in the clustered model probably benefitted it, allowing it to rule out sections of G (limiting the size of the BSG) before attempting further expansion.

The second obvious anomaly is in Graph 12 where MCC obtains a cost that is much lower than all the other algorithms. This is again a symptom of the MCC planner being too optimistic and perfectly illustrates the risks of not considering blocked edges in route planning. It chooses the shortest route to the goal, which seems very attractive assuming all uncertain edges are free. However, when some fraction of those edges are blocked, performance rapidly degrades as shown by MCC's average and maximum costs which are notably higher than the other algorithms. Both LAO* and Perseus plan for edges being blocked and therefore choose a more conservative route, thereby obtaining a lower average cost at the expense of never using the shortest route.

Graph 14 is notable for having very few actual results in it. All three MDP models fail to compute a policy, and while Perseus does find a policy in a relatively short amount of time, that policy fails to reach the goal in every single simulator trial. Graph 14 is an unsafe graph where the only route to the goal is through a narrow passageway where at least 2 of 3 uncertain edges must be traversed, so a high failure rate is expected (the goal is often unreachable). Due to the high probability of all edges being blocked, Perseus may not have been able to compute a suitable policy due to the discount factor of 0.95, which may be too low to give the necessary emphasis on the eventual cost, even in the world where the goal was reachable. MCC plotted a route that failed to reach the goal 91% of the time and shows the rare scenario where MCC has an advantage since it does not consider the future cost when edges are blocked, and plans directly in the PRM graph.

Alg.	Graph	Avg. w/ fails	Avg. w/o fails	% fail
Graph 5	I	F	F	F
	C	F	F	F
	D	F	F	F
	MCC	1649.67	2107.00	24.97
	PRS	1433.93	2180.62	31.93
	Sym.1000	1460.5	2481.47	50.06
	Sym.5000	1457.74	2488.60	49.66
	Sym.50000	1460.52	2490.25	50.49
Graph 6	I	1500.55	1500.55	0
	C	1426.42	1426.42	0
	D	1425.75	1425.75	0
	MCC	1525.04	1525.04	0
	PRS	1425.04	1425.04	0
	Sym.1000	1388.75	1596.07	9.82
	Sym.5000	1390.23	1599.68	10.00
	Sym.50000	1386.72	1599.32	10.06
Graph 11	I	1218.89		0
	C	F	F	F
	D	F	F	F
	MCC	1219.20	1219.20	0
	PRS	-	3500	100
	Sym.1000	1155.07	1353.45	8.46
	Sym.5000	1219.86	1219.86	0
	Sym.50000	1220.07	1220.07	0
Graph 13	I	F	F	F
	C	F	F	F
	D	F	F	F
	MCC	1374.38	1374.38	0
	PRS	F	F	F
	Sym.1000	-	3500	100
	Sym.5000	781.17	2384.43	61.71
	Sym.50000	757.07	2733.90	72.07
Graph 14	I	F	F	F
	C	F	F	F
	D	F	F	F
	MCC	1188.00	3291.92	91.0
	PRS	-	3500	100
	Sym.1000	-	3500	100
	Sym.5000	-	3500	100
	Sym.50000	-	3500	100

Table 4.21: Key: I=independent, C=clustered, D=dependent model. “F” indicates that LAO* failed to produce a policy.

Alg.	Graph	Avg. w/ fails	Avg. w/o fails	% fail
Graph 15	I	1140.44	1140.44	0
	C	1140.35	1140.35	0
	D	1137.60	1137.60	0
	MCC	1211.70	1330.51	5
	PRS	1346.38	1346.38	0
	Sym.1000	785.99	1625.16	30.92
	Sym.5000	785.82	1637.79	31.14
	Sym.50000	785.60	1626.25	30.97
Graph 16	I	1795.23	1795.23	0
	C	1795.23	1795.23	0
	D	F	F	F
	MCC	2369.04	2369.04	0
	PRS	1795.23	1795.23	0
	Sym.1000	628.09	3112.58	86.51
	Sym.5000	628.09	3109.71	86.41
	Sym.50000	628.09	3110.86	86.45
Graph 18	I	1120.11	1120.11	0
	C	1120.26	1120.26	0
	D	1120.09	1120.09	0
	MCC	1123.22	1123.22	0
	PRS	1128.47	1128.47	0
	Sym.1000	-	3500	100
	Sym.5000	-	3500	100
	Sym.50000	-	3500	100
Graph 19	I	1337.02	1337.02	0
	C	1311.79	1311.79	0
	D	1311.54	1311.54	0
	MCC	1406.32	1406.32	0
	PRS	-	3500	100
	Sym.1000	1310.92	1310.92	0
	Sym.5000	1336.58	1336.58	0
	Sym.50000	1311.56	1311.56	0
Graph 20	I	F	F	F
	C	F	F	F
	D	F	F	F
	MCC	1303.86	1303.86	0
	PRS	894.29	1973.76	41.31
	Sym.1000	-	3500	100
	Sym.5000	-	3500	100
	Sym.50000	-	3500	100

Table 4.21: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy. Algorithm comparison across range of example graphs when failed runs are included in the average cost. Cost for failing set to 3,500, all other parameters remain unchanged. The averages from Table 4.15 are included for comparison.

4.6.4 Discussion

Overall, the independent model is the fastest of the three MDP models, but it cannot find the optimal policy when knowledge of edge dependencies is required to accomplish that. The results from the MCC planner show that the average cost achieved under independence can often be matched or closely approximated by using a simpler, vastly quicker algorithm such as replanning MCC. The dependent model offers the best policy performance of all the algorithms tested, but is also more likely to fail on complex graphs and larger examples, which detracts from its usability. Solving the problem via the full, dependent POMDP formulation is infeasible for most problems because of the required computation time, despite good performance in several cases. Perseus, like LAO* gains a computational advantage by exploring the belief graph out from the starting state to reduce the plan space, however it cannot account for the low probability of reaching some states, or for the extra structure in these problems. Although it does not suffer from the approximation introduced by discretising the belief state, Perseus cannot exploit the problem structure as effectively as our approach, so is orders of magnitude slower. Symbolic Perseus is much quicker as a result of factoring the state space and its use of efficient data structures. In many cases, its policies are as good as the dependent MDP model. Although in a few cases its speed is comparable to dependent LAO* it is generally significantly slower.

Given the difference in computation time between the independent and clustered models, there is little reason to choose the former over the latter, presuming sufficient time is available. Allocating more memory to the dependent model will help mitigate the occurrence of failures, but it is still inherently less scalable than other models. The advantage of the clustered model over the independent model does rely on the correct arrangement of the uncertain edges into clusters. Since edge dependencies between edges in the same cluster can be represented, but clusters are assumed independent, if the cluster configuration is such that two highly dependent edges are in different clusters, they will be treated as being independent of each other. The clustered model offers the benefit of exploiting edge dependencies in the graph where present, without the extended computation time of the dependent model and the increased risk of failure.

4.6.4.1 When does a dependency between edges offer an advantage?

The clustered/dependent models have shown a performance advantage with less than half of the graphs despite the fact that all of them contain uncertain edges whose free status is dependent on other edges. This suggests the conditions necessary to produce a benefit from dependencies are complex. During the research we have found that the conditions described below are necessary for there to be an improvement when clustered

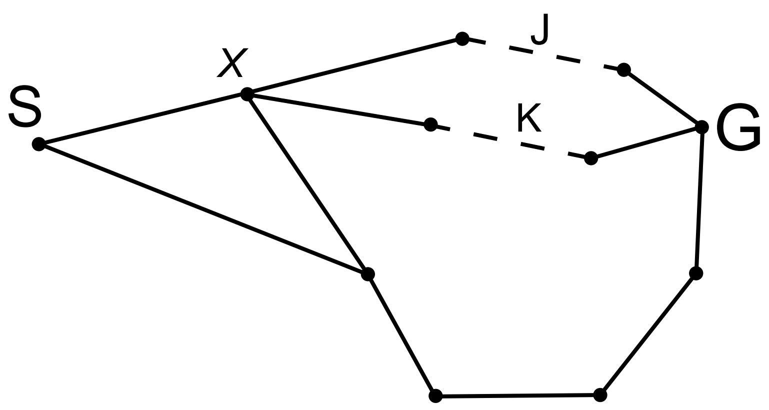


Figure 4.11: An example of where the dependent MDP model can gain a cost advantage over the independent model. The two uncertain edges shown (dashed lines) are highly dependent on each other. The agent may make an observation of edge J at the node marked x.

or dependent models are used.

The graph shown in Figure 4.11 will be used to illustrate the example. It has a short route from the start (S) to the goal (G) that has a choice of two uncertain edges (labelled J and K) whose free statuses are highly dependent on each other. The agent can observe the status of edge J from node x, however does not observe edge K. It also has the choice of the long safe route to the goal, represented by the lower path in the figure. For the sake of discussion, assume that under the optimal policy, the agent moves from the start node S to x and makes an observation of J and observes J as blocked. The agent can either continue on the short route to one of the uncertain edges, or can select the long safe path to the goal. The following must be true for the dependent MDP model to gain a cost advantage over the independent model.

- The uncertain edges must exist on the shorter route to the goal. This necessity is fairly straightforward: there is no point asking the agent to choose between a short, safe route and a risky, longer route.

- The agent must be able to observe one of the dependent edges and not the others before it reaches them. If both edges J and K were observable at node x, then the independent model will obtain the same information as the dependent model and there is unlikely to be a large difference in cost.

- The agent must be confident enough that the uncertain edges are free to cause the agent to accept the risk of the shorter route, otherwise it will choose the long route by default and all models will behave in the same way. This must be true of both uncertain edges in the short route. If edge K (as yet unobserved) has a high probability of being blocked, the independent policy will likely not consider it worth the risk and will choose the long safe path at x anyway. As the dependent policy can infer the second edge is blocked, both models will effectively choose the same policy and obtain the same cost.

- The difference in average cost is based on differences in the agent's behaviour. In the dependent model, once J is observed as blocked, the agent infers that edge K is blocked due to the dependency. It will then head for the safe path. In the independent model, it can make the observation of J yet still believe the edge K is free and continue towards it. When the K is found to be blocked, it must then backtrack to find the alternate route. Avoiding travelling towards the second blocked edge and backtracking is where the clustered/dependent models gain any advantage. The more costly the potential backtrack, the more advantage the dependent models will have, but conversely, the greater the perceived risk will be for the independent model so the chance it will investigate the second edge decreases.

Random graphs tend to have many alternate routes, so the chance that all the above occur in such a way as to give the dependent model a large cost advantage is not great. This is why not as many graphs as might be expected show an advantage under dependent models. Representing dependencies between edges is still an important feature. In situations where backtracking is costly and the minimal cost path to the goal provides an advantage, the clustering approach allows the lower cost to be obtained in a feasible amount of time.

4.6.5 Summary

A number of features have been discussed in the results so far, a summary of which is provided below:

Discretisation This is necessary to keep the state space finite. Discretisation resolution d, must be fine enough to ensure that a non-zero value can be given to each element in the belief states. Generally, once $d \leq 1 \times 10^{-5}$, the discretisation is so fine that any belief state can be represented accurately for the graphs considered. This resolution was sufficient for all graphs tested.

Clustering In our experiments, the clustered MDP model is the preferred model over the alternatives because it gives the most advantageous balance between speed and policy quality. If the cluster configuration is reasonable, then any edge dependencies will be exploited without a large increase in computation requirements. Section 4.6.4.1 gives a detailed description of when dependencies in edges are likely to lead to a lower cost policy.

Graph pre-processing Applying the pre-processor to the PRM graphs significantly shrank their size with a negligible overhead in terms of runtime. As policy costs are not affected, this technique should always be used to minimise the state space size and algorithm computation time.

Chapter 5

Scaling the MDP Models

In the results in the previous Chapter, the dependent MDP models failed to produce a policy for several graphs and the other models also failed in some cases. In this Chapter, we look at some techniques that can be used to extend the range of problems that our approach can handle.

5.1 Heuristic Weighting in LAO*

With an admissible, consistent heuristic both A* and LAO* are guaranteed to find the lowest cost path to the goal. Further, with any given consistent heuristic, A* is provably optimal (Nilsson 1998, pp.146–148) in that no other algorithm will expand fewer nodes when generating the best path. This does not entail that there is no room for improvement however. From Pearl (1984):

> *"Experience shows, however, that in many problems A* spends a large amount of time discriminating among paths whose costs do not vary significantly from each other. In such cases, the admissibility property becomes a curse rather than a virtue. It forces A* to spend a disproportionately long time in selecting the best among roughly equal candidates and prevents A* from completing the search with a suboptimal but otherwise acceptable solution path."*
>
> (Pearl 1984, page 86)

Another relevant insight from Pearl is that the *"combinatorics of the problem may be such that an admissible A* cannot run to termination."* As is the case when trying to evaluate larger PRM graphs with LAO*, the algorithm terminates due to memory restrictions before the optimal solution is generated. For these reasons, we sometimes wish to trade solution optimality for a reduction in time or space requirements of the algorithm. Altering the node evaluation function so that the value is biased more towards the heuristic value than the known value is one method of accomplishing this.

5.1.1 Evaluation function decomposition

The idea of adding a bias to the standard cost computation of A* is not a new one. Pohl (1970) first introduced the idea of changing the "weight" of the heuristic in the standard A* evaluation function and Chakrabarti et al. (1987) showed that a decomposition of the evaluation function of AO* could be similarly achieved. This adaptation supports the weighting of the heuristic function as in the original A* algorithm. In Hansen and Zilberstein (1999; 2001) the authors show that this decomposition can be generalised to LAO* as well.

In A*, the standard evaluation function for a node's cost is commonly formulated as:

$$f(n) = g(n) + h(n)$$

The $g(n)$ function is the known cost of all the graph edges from the start node to n, and $h(n)$ is the heuristic estimate of the cost to reach a goal from n. In LAO* each action may result in one of several successor states (transitions in the MDP state space). Each time a dynamic programming update is performed on a node, the value is derived from two components: the cost of executing that node's current best action in the MDP, and the value of the successor nodes (line 15 of LAO* on page 37). For unexpanded tip-nodes, their value is solely based on the heuristic. When an LAO* policy is complete, the cost of each node is known exactly. For nodes one transition (i.e. one edge in the explicit graph) away from a tip-node, their cost consists of the tip-node's heuristic and the known cost of traversing to that node. As each node in the explicit graph is backed-up (towards the start node) the cost of each transition is included in its value. Unlike in A* where each node stores its current lowest cost from the start, in LAO* each node's value is the current lowest cost to the goal, assuming the marked action at each node is chosen. This allows for a weighting function to be introduced into the Bellman backup equation in a simple manner explained below.

Pohl (1970) suggested the following

$$f_w(n) = (1 - w) \cdot g(n) + w \cdot h(n) \tag{5.1}$$

to adjust the weighting of the component parts of the evaluation function for node n, where $0 \leq w \leq 1$ is the weight used in the policy. When w is placed at either extreme, A* behaves identically to one of two related algorithms. With $w = 0$, the heuristic is ignored entirely causing A* to become uniform-cost search. The optimal path will eventually be found, though the search will generally be less efficient (analogous to the relationship between Dijkstra's shortest path algorithm and A*). When $w = \frac{1}{2}$ the equal weighting causes behaviour identical to unweighted A* and when $w = 1$, it becomes greedy search. The latter case is useful where many goals of similar cost exist and we do not care which one is found by the search, but in other domains such as ours, it can lead to very bad policies.

When used with very optimistic heuristics similar to the one we employ, which may underestimate the true cost by a significant amount, using a high weighting (e.g. $w \geq 0.85$) can be extremely detrimental to the resultant policy. For an admissible heuristic, they will remain admissible as long as $0 \leq w \leq \frac{1}{2}$, but will lose this property when $w > \frac{1}{2}$. Once the heuristic becomes inadmissible, the overestimation of node costs can cause LAO* to ignore optimal nodes in preference to expanding others that are ostensibly cheaper and thus return a suboptimal policy. From Pearl (1984):

Definition 5.1 (Upper Support)**.** A random variable X is said to have an upper support r, denoted $r(X)$, if $X \leq r$ and $P(X \leq x) < 1$ for all $x < r$.

If function $h^*(n)$ gives the true cost to the goal for node n, then

$$\epsilon_n = \max(0, h(n) - h^*(n)) \tag{5.2}$$

can never be greater than 0 for admissible heuristics, where ϵ_n is the error in the heuristic value for node n. Therefore, the upper support $r(\epsilon_n)$ equals 0. Substituting this result into the formula for maximum optimal weight (Pearl 1984, page 206) gives the result required:

$$w_0 = \frac{1}{2 + r}$$

where r is an upper support and w_0 is the optimal weight, i.e. the highest weight that guarantees the admissibility of h. Using $r(\epsilon_n)$ from (5.2) shows that any weighting above $w = \frac{1}{2}$ may lead to a suboptimal policy.

5.1.2 Integration into LAO*

Given the above decomposition of the value function, adjusting LAO* to incorporate heuristic weighting becomes straightforward. Re-examining Algorithm 4.2 (page 113), when a tip-node is first created, its value is calculated from the heuristic alone (line 10), since it is unexpanded with no successors in the explicit graph:

$$n' \text{ value } \Leftarrow \begin{cases} 0 & \text{if } a = v_G \\ h(n) & \text{otherwise} \end{cases}$$

This value does not include the $g(n)$ cost, just the heuristic cost $h(n)$. Weighting is incorporated by substituting line 10 with:

$$n' \text{ value } \Leftarrow \begin{cases} 0 & \text{if } a = v_G \\ w \cdot h(n) & \text{otherwise} \end{cases}$$

where the heuristic is multiplied by w. The remaining component of Equation (5.1) is incorporated in the Bellman update for a node (line 15 of Algorithm 2.3):

$$f(n) \Leftarrow \min_{a \in \mathcal{A}} \left[r(n,a) + \gamma \sum_{z \in Z} t(n,a,z) \cdot f(z) \right]$$

The Bellman update includes the cost $r(n,a)$, the cost of executing action a from node n, which in our domain is the edge cost of reaching neighbouring node a in the PRM graph. Multiplying the coefficient $(1-w)$ into the cost provides the other term for Equation (5.1):

$$f(n) \Leftarrow \min_{a \in \mathcal{A}} \left[(1-w) \cdot r(n,a) + \gamma \sum_{z \in Z} t(n,a,z) \cdot f(z) \right]$$

5.1.3 Heuristic weighting experimental results

To investigate the effects of adding heuristic weighting to LAO*, we first show how the use of weighting can produce a sub-optimal policy by examining a tiny example and showing how the weighted policy differs from the non-weighted policy. We then demonstrate the effects of weighting on a small graph to show how the differences in computed policies affect the average cost, and finally we apply weighting to the larger examples from Chapter 4 and determine some general conclusions on the effect of heuristic weighting.

5.1.3.1 A suboptimal weighted policy

In cases where the path lengths between a shorter, risky route and a longer, safe route are similar, then a small change to the perceived cost of a route can easily change the policy. The obvious factor in route cost is the agent's confidence in the success of a route. Consider the small graph in Figure 5.1 which has only two possible routes from the start to the goal. If the agent's belief that the uncertain edge from A to G was free

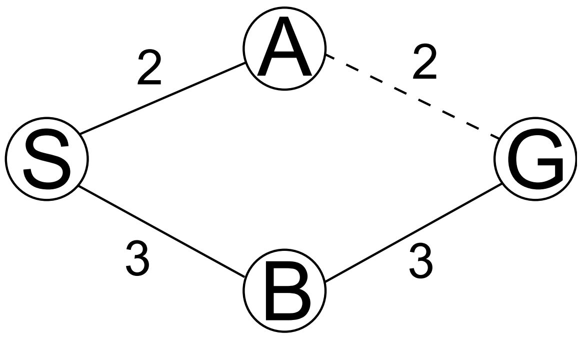


Figure 5.1: Small graph with one uncertain edge. S is the start node and G is the goal node.

was 0.66, then the optimal policy without weighting is to go to the goal via B—even a two-thirds probability of succeeding is not worth the risk. However, if we apply even a small weighting to the heuristic of $w = 0.51$, then the computed policy changes and is no longer optimal. The returned policy under weighting chooses going to node A as the best action at the start. This is non-optimal because the unweighted policy tells us that on average, this will be more expensive, if only by a tiny margin. Assume that the prior probability, $P(A \rightarrow G$ free)=0.66 and $w = 0.51$. LAO* will first expand S to produce A and B with costs $w \cdot h(A) = 1.02$ and $w \cdot h(B) = 1.53$ respectively. The best action at S will be to move to A. Expanding A produces the goal node G.[1] At this point no non-terminals exist in the BSG, so the cost calculation at S proceeds as follows:

$$\begin{aligned}
\text{Start action} \rightarrow \qquad A &= (P(A \rightarrow G \text{ free}) * ((1-w) * A \rightarrow G \text{ cost}) \\
&\quad + P(A \rightarrow G \text{ blocked}) * ((1-w) * \text{cost to backtrack})) \\
&\quad + ((1-w) * S \rightarrow A \text{ cost}) \\
&= (0.66 * (0.49 * 2) + 0.34 * (0.49 * 8)) + (0.49 * 2) \\
&= 2.9596
\end{aligned}$$

$$\begin{aligned}
\text{Start action} \rightarrow \qquad B &= (w * h(B)) + ((1-w) * S \rightarrow B \text{ cost}) \\
&= (0.51 * 3) + (0.49 * 3) \\
&= 1.53 + 1.47 \\
&= 3
\end{aligned}$$

Due to the weighting, the expanded B node with a belief state of [0.66, 0.34] uses the weighted heuristic cost of 1.53, resulting in the node never being expanded as LAO*

[1]An arc back to S is also produced but it is irrelevant here.

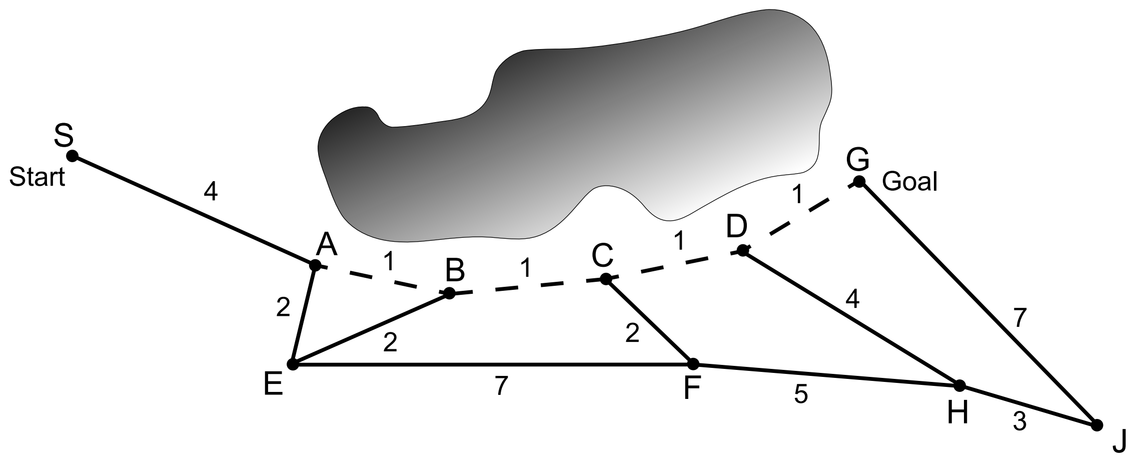


Figure 5.2: Small graph with 4 uncertain edges. The dotted lines represent uncertain edges. The letters represent the node labels and the numbers are the edge costs. In the clustered model, $A \to B$ and $B \to C$ are in one cluster and $C \to D$ and $D \to G$ are in a second cluster.

w	**Independent**	**Clustered**	**Dependent**
0.5	25.27	25.27	25.27
0.6	25.96	26.0	25.96
0.75	26.0	26.0	25.98
0.85	26.0	26.0	26.0
0.99	26.06	26.06	26.09

Table 5.1: Effect of w on policy cost when a uniform start belief is used.

believes the A path to be cheaper. Thus, the algorithm terminates recommending visiting node A from the start.

5.1.3.2 Weighting on a small graph

Consider the graph in Figure 5.2 where the 4 edges across the top of the graph are uncertain due to the presence of the shaded object. Table 5.1 shows how the average costs of independent, clustered and dependent models change with w. The starting belief was uniform, i.e. the true state of each edge is independent with P(free)=0.5. All simulated costs are averaged over 50,000 trials with a maximum length of 50 steps each. When all four edges are fully dependent upon each other this means that they are either all free or all blocked. Table 5.2 shows how weighting affects the policy costs with this fully dependent start state.

In the Table 5.1 (edges are independent) all models achieve a close or identical cost to each other in the simulator. This is unchanged even as w approaches higher values of 0.85 where the policy is heavily influenced by the heuristic. When $w \in [0.6, 0.99]$ the size of the best solution graph (BSG) stays constant at 76 nodes with an average cost

w	**Independent**	**Clustered**	**Dependent**
0.5	22.0	20.08	17.96
0.6	25.98	19.97	18.05
0.75	26.15	20.0	17.98
0.85	26.11	19.99	17.99
0.99	25.95	20.0	18.03

Table 5.2: Effect of w on policy cost when a dependent start belief is used.

		Weight w		
Starting belief state	**Model**	**0.5**	**0.75**	**0.95**
Uniform	I	175	200	460
	C	133	34	44
	D	13	12	27
Fully dependent	I	162	201	456
	C	6	5	7
	D	3	4	3

Table 5.3: Key: I=independent, C=clustered, D=dependent model. Effect of w on policy generation time in independent, clustered and dependent models. All times in milliseconds.

of ∼26.0. This is down to the topology of the graph: the heuristic will direct the agent via the cheapest route to the goal from any node just as the [unweighted] independent policy will investigate each edge to see if a shortcut is available. Essentially, when all edges are independent, the optimal policy turns out to be very close to the greedy policy so weighting does not deteriorate the quality.

The results from Table 5.2 show similar trends as before. Once $w > 0.5$ the policy is no longer optimal. The independent model necessarily performs identically. The lower value of 22.0 occurs when $w = 0.5$ because the policy is optimal in this case and the simulator can only select one of two possible worlds, as opposed to 16 (2^4) when the edges are truly independent. The clustered and dependent policies show no change in performance even when $w = 0.99$ with the respective policy sizes being constant across the range of weightings. The graph layout is the cause again here as the optimal policy is naturally greedy, thus an increased w does not have much effect. Table 5.3 shows the solution times for the previous results for the three different models at different weightings. Across all the results, the clustered and dependent models are consistently quicker than the independent model. From the uniform start state each model produces the same policy and LAO* explores the same amount of the explicit graph. When $w = 0.5$, the explicit graph contained 198 nodes and at $w = 0.95$, the explicit graph contained 172 nodes in all three models. The small difference in nodes expanded cannot explain the large increase in solution time when $w = 0.95$. The longer solution time is caused by extra iterations spent waiting for node values to converge under value iteration (the convergence phase of LAO*)—this can increase quite sharply at higher weightings around 0.95. The clustered and dependent times are much quicker which can be explained by the amount of the graph explored. The clustered and dependent models only have 4 and 2 possible world models respectively to consider. The dependent model knows either all edges are free or blocked while the clustered model cannot represent the relationship between the two clusters of two edges, hence 4 worlds. This shows the rare circumstance where, because all edges are in the artificial condition of being fully dependent, the number of worlds and therefore reachable state space for the clustered and dependent models is smaller than for the independent model, hence the smaller computation times.

What we have demonstrated here is that heuristic weighting can be beneficial in some circumstances where the heuristic estimate is close to the true cost to reach the goal. In Figure 5.2, the optimal policy is naturally greedy, so the heuristic is a good estimate. In these situations sacrificing admissibility can be beneficial by making the LAO* graph search greedier so fewer nodes are explored. However, this is unlikely to be true in many cases as the heuristic we use (see Section 4.5.1.2) can substantially underestimate the true cost to the goal.

5.1.3.3 Larger examples

In this Section, we analyse the effects of weighting when applied across the range of larger graphs. Three weights of 0.5,0.7 and 0.9 were chosen, with 0.5 being the unweighted default. The graphs were pre-processed as before, other settings remain unchanged.

A selection of results can be found in Figures 5.3 to 5.7. The graphs shown were chosen because they are representative of the behaviour seen across all graphs, and this selection allows us to illustrate specific traits. The resulting behaviour of the policies under weighting can be broadly divided into three classes. Several examples where the MDP models failed with no weighting applied (i.e. $w = 0.5$) continue to fail even under heavy weighting. Graphs 5,13 and 14 fall into this category. Others where the dependent model fails with no weight applied, are solvable at higher weighting such as Graphs 1 and 3. An example of this can be seen in Figure 5.3 where no data point exists for the dependent model at $w = 0.5$. Increasing w shifts the bias away from the g component of the decomposition Equation (5.1), and places more emphasis on the heuristic h. As the heuristic underestimates the cost to goal, this effectively makes the LAO* graph search more greedy. This typically means that less of the state space is explored as w increases, but in complex PRM graphs such as those listed above, LAO* still tries to explore too many states in $\mathcal{S}$ to generate a complete policy.

The second type of behaviour seen is where increasing the weight has little effect on the policy performance. Figure 5.6 shows an example of this, where the average policy cost across all models is virtually identical at all weights and only increases slightly as more weighting is applied. Graphs 4,8 and 11 share this trend. The most common pattern is that seen in Figures 5.3, 5.4 and 5.7 where the average policy cost across all models sharply increases once too much weighting is applied. The advantage that the clustered/ dependent models show in graphs when they are unweighted is nearly always eliminated when w is increased. This can be seen in several graphs and is due to the greedy nature of the heuristic. As more of the node's value (f) is derived from the heuristic when $w \geq 0.5$, the cost to goal becomes more significant than the cost already incurred. During policy computation, the advantage in cost that the clustered/dependent models can gain by avoiding unnecessary edge traversal carries less importance and is more likely to be ignored by the algorithm. Effectively, the agent becomes too preoccupied trying to minimise the cost to goal (h) instead of trying to minimise both g and h, as in the unweighted case. The point at which the policy performance decreases varies between graphs, so the graphs where the average cost remains relatively stable when w is increased can occur for two reasons. Figure 5.7 shows an example where the performance does not decrease until $w = 0.9$, compared to $w = 0.7$ in Figures 5.3 and 5.5. Firstly, as is the case with the small graph in Section 5.1.3.2, if the optimal policy is similar to a greedy policy, then the increased weight is unlikely to have a large detrimental effect on average cost. Secondly, as is the case in Figure 5.6, the difference in cost between one route and another may be fairly small. When $w = 0.5$, the policy will still look to use the lowest cost route, but when w increases, the more greedy route selected may have similar cost, so the perceived drop in performance is almost negligible.

The results for Graph 12 (Figure 5.5) show an anomalous result: the cost for both the clustered and independent models increases sharply then decreases again across the range of w. No failures are reported at any weighting with Graph 12. The exact cause for this is unknown, but is likely due to the construction of the graph. The dependent model fails at $w \leq 0.9$, indicating that LAO* had to expand many nodes to compute policies, so it is possible that the section of MDP state space necessary to find the lower cost policies is unexplored by LAO* when $w \approx 0.7$.

The original motivation for adding weighting to the LAO* algorithm was to allow a greater range of PRM graphs to be solved. Since increasing w tends to cause LAO* to expand fewer nodes when computing the policy, this should allow it to find policies for graphs where previously it had failed due to lack of memory. This would essentially trade off memory for policy quality. We can see that adding weighting to the node value calculation (5.1) produces an inadmissible heuristic leading to worse policies, but this is preferable to no policy at all. However, since many graphs that fail without weighting still fail when even a high weight is applied, this shows that such a strategy is not advisable. Furthermore, the differences in average cost seen across all models when weighting is applied can be substantial. The weighting where the average cost increases is also hard to determine and varies between particular graph topologies. Considering both of these conditions, it is hard to recommend heuristic weighting as a useful extension to the LAO* algorithm for these sorts of problems. If no other method succeeds in producing policies for a particular graph, then adding weighting to the value computation could serve as a useful last resort, but even so, it does not appear to work in most cases. It is important not to conclude that this applies to all MDP problems however, since if the optimal policy is likely to be similar to a greedy policy anyway, then weighting can be of benefit. Combined with an underestimating heuristic, adding a small weight to the algorithm may reduce the computation time necessary as fewer nodes are expanded in G to find the optimal policy.

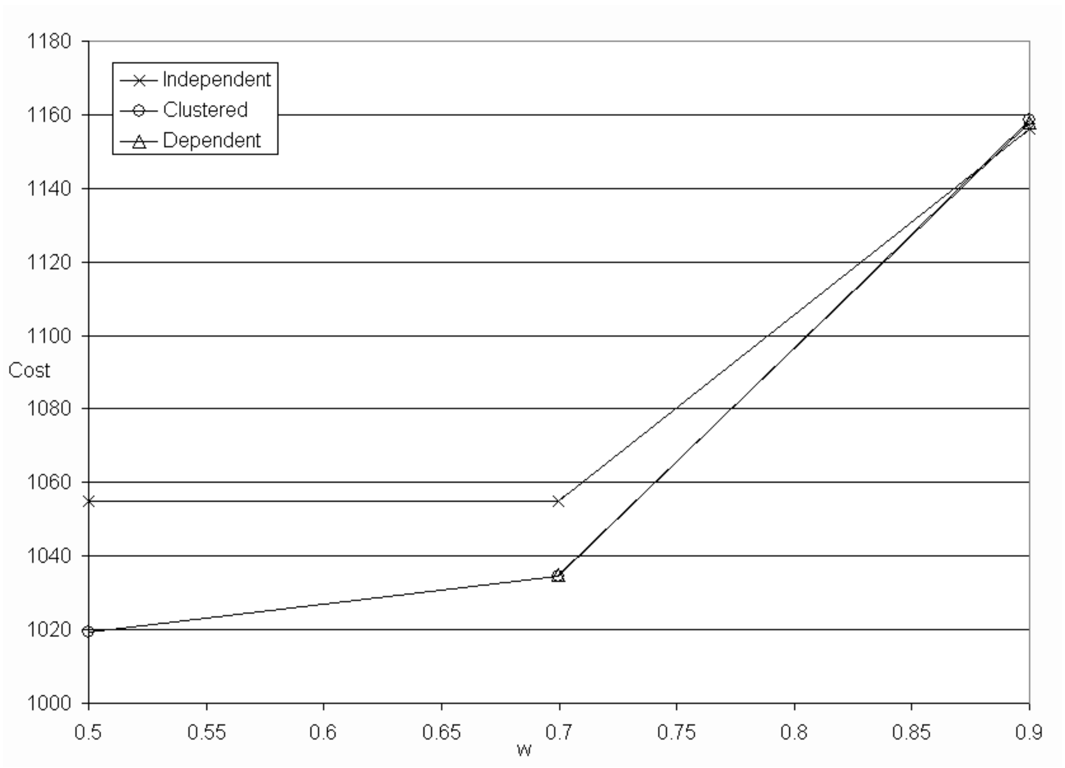


Figure 5.3: Graph showing the effect of weighting on average cost for Graph 1.

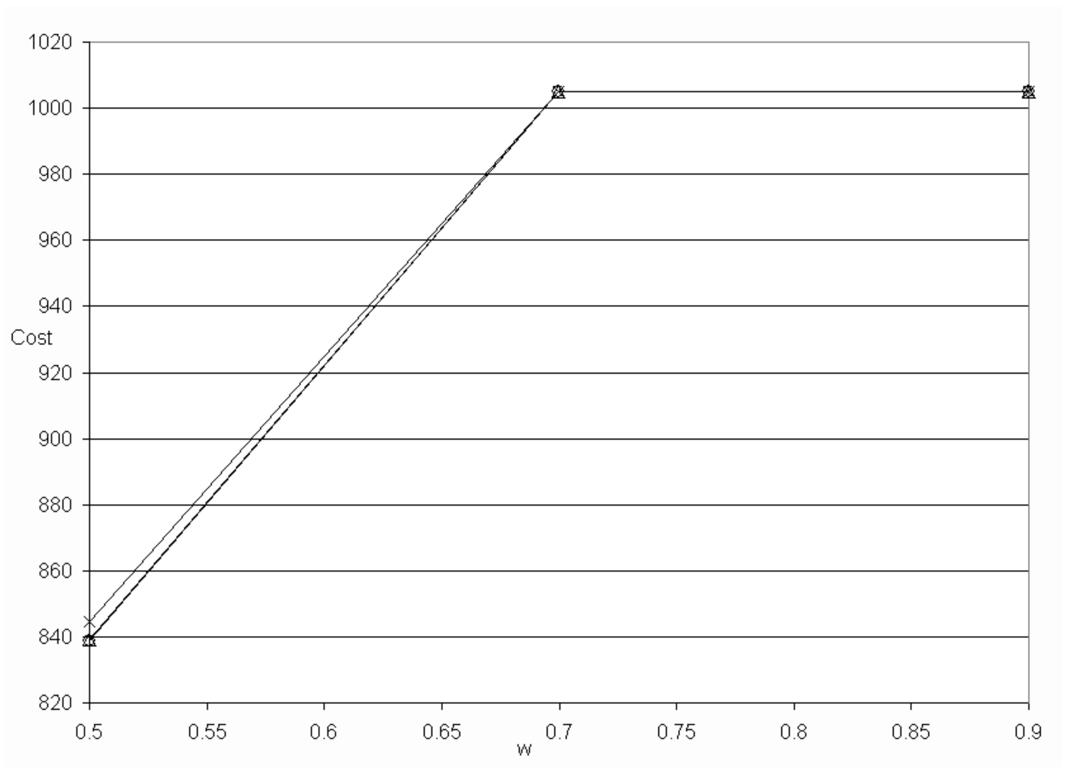


Figure 5.4: Graph showing the effect of weighting on average cost for Graph 10.

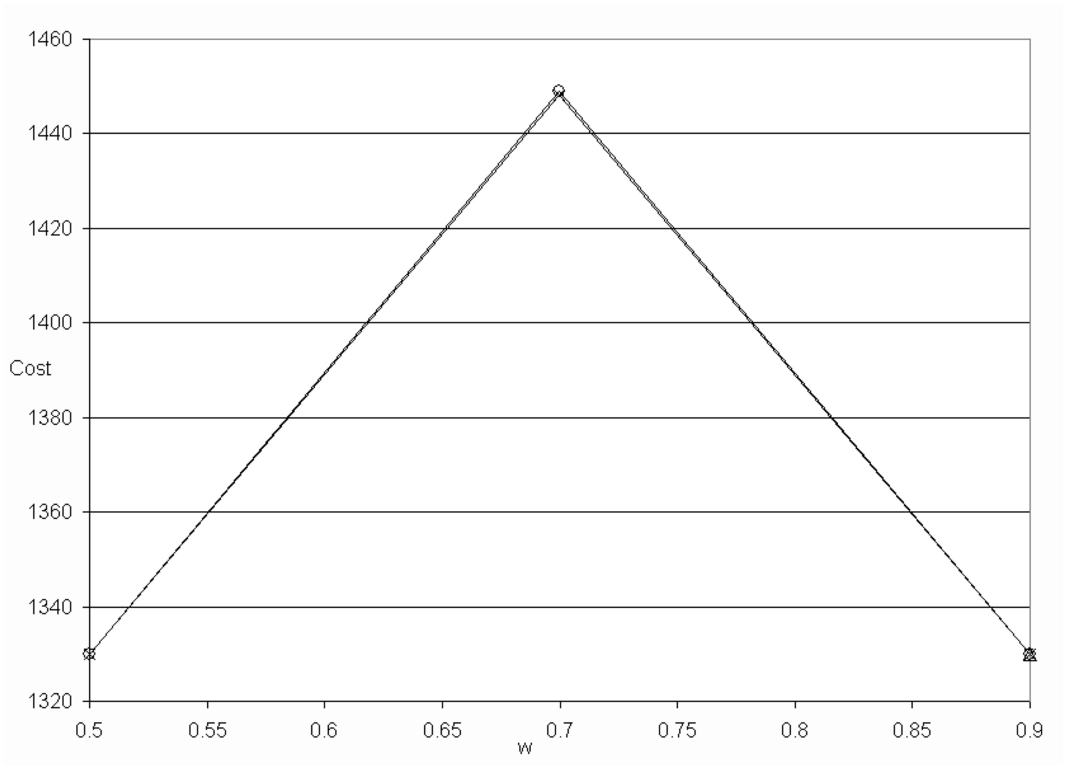


Figure 5.5: Graph showing the effect of weighting on average cost for Graph 12.

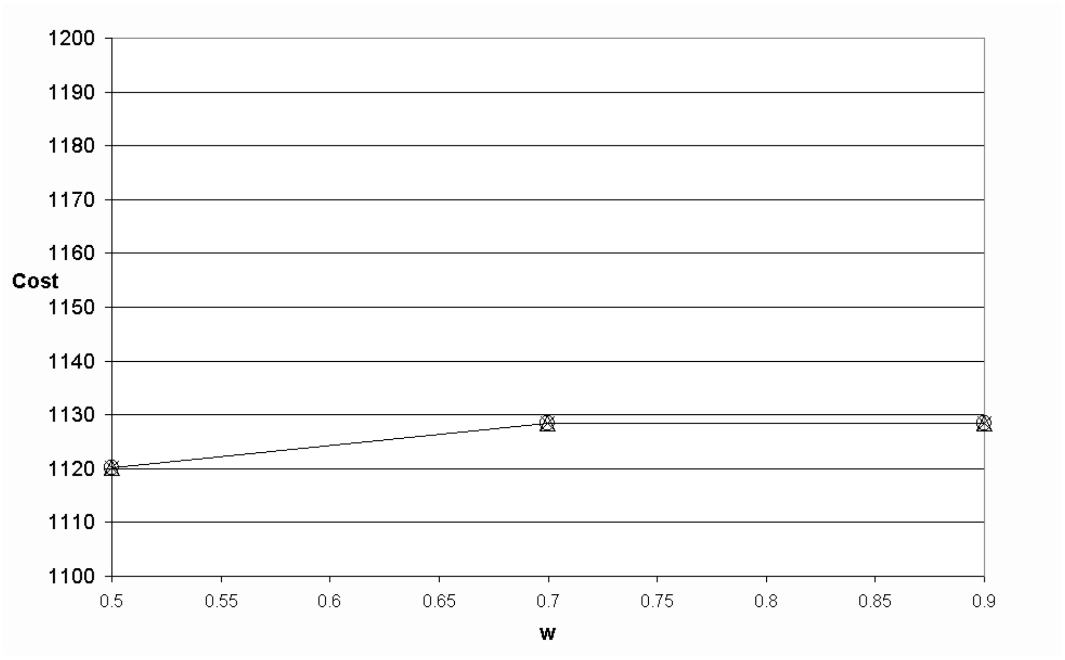


Figure 5.6: Graph showing the effect of weighting on average cost for Graph 18.

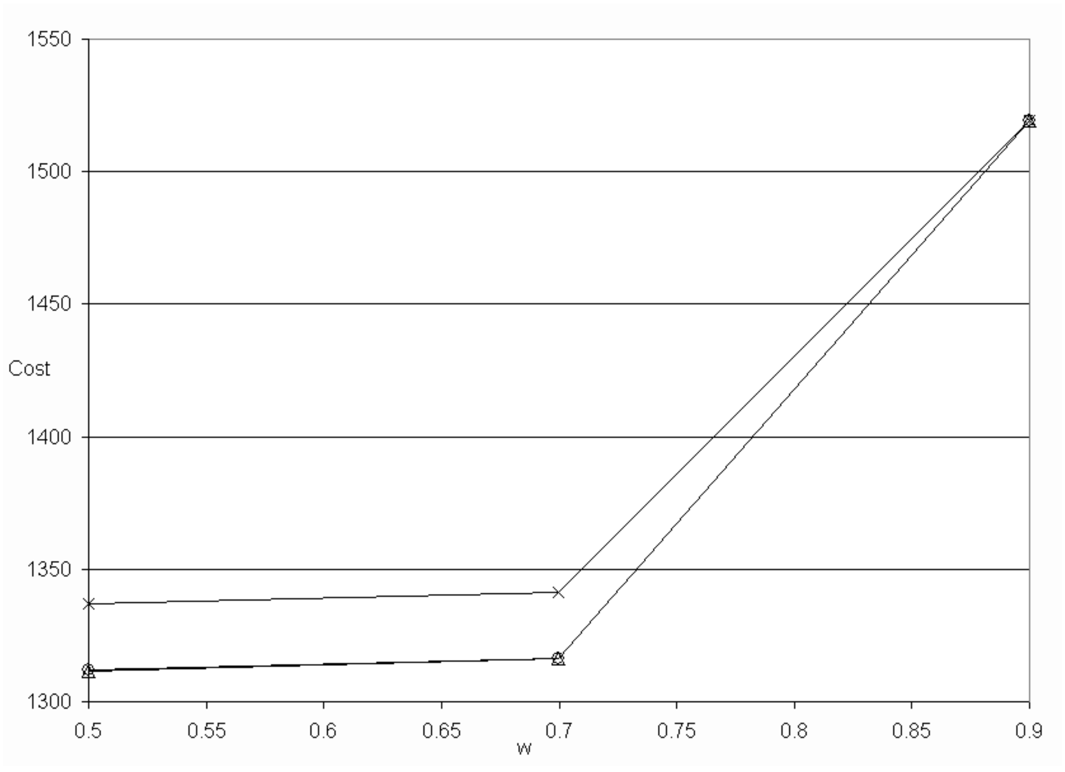


Figure 5.7: Graph showing the effect of weighting on average cost for Graph 19.

5.2 Approximation for LAO*

The results in the last Chapter show that solving the dependent MDP model with LAO* can often fail to produce a policy whilst the other models succeed. This is attributable to LAO* exhausting the available memory. Several experiments showed that the BSG generally comprised only a tiny fraction of the total nodes expanded in the explicit graph G. Part of the reason for the large amount of explored state space is that when observations alter the belief state, LAO* is actually exploring a discretised, adapted (through one of three models) continuous POMDP belief space. Ignoring the discretisation, repeatedly receiving an observation from the same node would give a new belief state each time (unless the observation was certain in which case only one new belief state can be reached), creating an infinite belief state space for LAO* to explore. With discretisation, the belief space and therefore implicit graph is finite but still extremely large, especially considering the discretisation of $d = 1 \times 10^{-5}$ used in the experiments. In this domain at least, LAO* needs to explore a lot of the state space to obtain the optimal policy. Given a perfect heuristic, only nodes in the final BSG would be selected for expansion,[1] so a more informed heuristic is one option for improving efficiency. However, creating a superior heuristic that retains admissibility while not requiring substantial amounts of computation is itself a non-trivial problem.

As in standard POMDP planning, the intractability of exact solutions to larger problems motivates the desire to create approximate solutions. As stated by Pineau et al. (2003), the loss of optimality is compensated for by the ability to handle larger problems and in many cases, the approximate solutions are close enough to optimal that the differences do not matter. Intuitively, the piecewise linear convex nature of a value function suggests that many belief states that are close in the belief space will have similar utility and will share the same maximal α-vector. This is the assumption that drives many approximate POMDP solution algorithms and is also the assumption that Bonet and Geffner (2009) rely on to provide quality solutions in the RTDP-Bel algorithm for POMDP solving. The underlying idea in many of these algorithms is that mapping multiple belief states onto one state (in the case of RTDP-Bel), or having one belief state represent a small locality of the belief space (as in grid based solvers such as Lovejoy (1991) and Hauskrecht (2000)), reduces the computational requirements for the solver. Given that noisy observations in our domain are likely to create belief states that are quite similar to the previous ones, we can apply a similar approximation technique to LAO*. One of the drawbacks to grid-based approaches is that they still require coverage of the full belief space to create a proper value function. We avoid this because, like

[1]Non-optimal siblings of BSG nodes are also expanded as a side-effect, since LAO* adds all of a node's successors to G when it is expanded.

point-based algorithms, LAO* exploits state reachability to avoid exploring irrelevant areas of the MDP state space. We have developed an extension to the LAO* algorithm that is suitable for our domain. We use an approximation technique to map new belief states to those already present in G if they are sufficiently similar. As with the belief pool expansion phase of PBVI, we want to avoid adding belief states that are close to existing states, instead preferring to add those that are far enough away and less likely to share the same utility.

5.2.1 Approximate LAO* design

The key component to the approximation is that we relax the definition of equality between belief states. Normally, two belief states must be numerically identical to be equal, but we relax this criterion to allow belief states that are similar but not identical to be considered equal. This requires some measure of distance to decide if two belief states are similar enough. While a range of measures are possible, such as the Euclidean distance between the belief states, we use the Kullback-Leibler (K-L) divergence (Cover and Thomas 2006) because it measures the relative entropy between two discrete probability distributions. The K-L divergence for two distributions a and b is defined by

$$\mathrm{KL}(a,b) = \sum_i a_i \log_2 \left(\frac{a_i}{b_i}\right)$$

The K-L divergence between two probability distributions is always non-negative with $\mathrm{KL}(a,a) \equiv 0$. However, it is also non-symmetric and is therefore not a true distance measure, i.e. $\mathrm{KL}(a,b) \neq \mathrm{KL}(b,a), a \neq b$. Since we want to measure the distance between belief states (the order of which cannot be predicted), we use the symmetrised K-L divergence (AI Access 2008):

$$\mathrm{KL}_{\mathrm{sym}}(a,b) = \frac{1}{2}\,\mathrm{KL}(a,b) + \frac{1}{2}\,\mathrm{KL}(b,a) \tag{5.3}$$

5.2.1.1 Approximation threshold

Using Equation (5.3), the equality of two belief states can be determined using a threshold $\max_{\mathrm{KL}}$, for the maximum symmetrised K-L distance between two approximately equal belief states. If a and b are belief states as before then

$$a \approx b \iff \mathrm{KL}_{\mathrm{sym}}(a,b) \leq \max_{\mathrm{KL}} \tag{5.4}$$

Essentially, the $\max_{\mathrm{KL}}$ threshold can be viewed as the radius of a sphere in the belief space around a single point (a belief state); any point within that radius will be considered equal to it under approximation. Appropriate values for $\max_{\mathrm{KL}}$ depend on the domain and the sensitivity of policies to approximation. The values chosen for our experiments were determined experimentally, both by measuring the K-L divergence between particular belief states to examine the range of values produced, and by comparing the performance of policies produced under different thresholds. Experimental results from the latter can be found later in this Chapter (see Section 5.2.2.5).

5.2.1.2 Approximate LAO*

Using the above description, we now describe the changes to LAO* necessary to create *Approximate LAO** (ALAO*). This extension is not generalisable to all MDPs, only belief state MDPs or MDPs with probability distributions in their state. The approximation affects LAO* node equality testing, so the actual changes occur in the expansion algorithm (Algorithm 4.2 on page 113). When the `expand` function creates a new successor node (i.e. a node with the new belief state $\langle a, b'_C \rangle$ in line 7), it searches G to see if that candidate node already exists in the graph. Ostensibly, this entails performing an equality test on all nodes in G to locate an already existing node, though in practice more efficient data structures are used. In standard LAO*, a numerical equality test is performed on every distribution in the belief state (for each $b_{c_1}, \ldots, b_{c_k} \in b_C$). If a node with the same belief state (and same node label) is found, an arc to that node is added to G. If none is found, then the new candidate node is added to G. In ALAO*, we substitute this exact equality test for Equation (5.4) for each distribution, so if an approximately equal node exists in G, it will be used instead of creating a new node. ALAO* nodes with differing PRM node labels are never considered equal in order to maintain the agent's perfect knowledge of its location. The K-L distance is calculated separately for each cluster in the belief state as clusters' probability distributions are independent, so for two clustered belief states b_C and b'_C:

$$b_C \approx b'_C \iff \mathrm{KL}_{\mathrm{sym}}(b_{c_i}, b'_{c_i}) \leq \max\nolimits_{\mathrm{KL}} \quad \forall\, i = 1, \ldots, k \tag{5.5}$$

where k is the number of clusters in cluster set C. Equation (5.5) determines the approximate equality between two clustered belief states and is used to make the decision between adding $\langle a, b'_C \rangle$ to G or creating an arc to an approximately equal node.

5.2.1.3 Effects

The behaviour of ALAO* is determined completely through the single free parameter $\max_{\mathrm{KL}}$, so choosing an appropriate value is crucial. The threshold specifies the approximation tolerance for the whole algorithm. Using lower values makes the algorithm less tolerant, causing a stricter equality test that brings the behaviour closer to that of LAO*. Using higher values increases the tolerance and should cause fewer nodes to be created in

Initial belief	Model	ALAO*?	Avg.cost	Std.dev	$\|G\|$	Time (ms)
Uniform	I	N	6.24	2.76	29	< 2
		Y	6.24	2.76	29	< 2
	D	N	6.24	2.76	29	< 2
		Y	6.25	2.76	29	< 2
Dependent	I	N	7.99	2.99	29	< 2
		Y	7.99	2.99	29	< 2
	D	N	6.49	2.49	14	< 2
		Y	6.49	2.49	14	< 2

Table 5.4: Key: I=independent, C=clustered, D=dependent model. Comparison of LAO* and ALAO* for graph in Figure 4.8b. All times averaged over 10 runs. Both edges were fully dependent in the dependent starting state.

G. However, using a value that is very high will cause policy performance to deteriorate. Belief states that need to be distinguishable to the agent to properly account for new information will become equal under approximation—stopping the agent from exploiting that information. Discretisation and ALAO* essentially serve the same purpose: they are both approximation methods for the continuous belief space. Their aim is to reduce the quantity of belief states that have to be evaluated to find a policy. The difference lies in how the two methods determine which belief points to include in the policy. Discretisation is, in effect, performed directly on the continuous belief space before any computation commences. It effectively fixes which belief states are visible to the algorithm. ALAO* is much more dynamic in belief point selection because it is integrated into the solver. The decision on whether a particular belief point should be explicitly created depends on the belief points previously created by the solver.

5.2.2 ALAO* experimental results

5.2.2.1 Small problems

First of all we demonstrate Approximate LAO* (ALAO*) on two of the toy problems from earlier, specifically, the 6 point graph and the 3 edge graph from Figure 4.8 on page 122. The results from standard LAO* and ALAO* are shown for comparison. The discretisation resolution was kept at $d = 1\times10^{-5}$, the maximum K-L distance $\max_{\mathrm{KL}} = 0.1$ and no weighting was applied. Only the independent and dependent models were tested for the 6 point graph, while all three models were tested for the 3 edge graph. The results are shown in Tables 5.4 and 5.5.

The results in Table 5.4 are unremarkable and simply serve to show ALAO* working on a tiny example. The size of the PRM graph means that ALAO* gains no time advantage and none is expected as the amount of explored state space is identical to normal

Initial belief	Model	ALAO*?	Avg.cost	Std.dev	$\lvert G\rvert$	Time (ms)
Uniform	I	N	7.86	1.34	1594	1217
		Y	8.60	4.25	123	166
	C	N	7.85	1.35	2456	1352
		Y	8.61	4.26	123	163
	D	N	7.85	1.35	2649	1517
		Y	8.59	4.22	123	168
Dependent	I	N	8.32	1.87	1594	1223
		Y	8.17	3.56	123	170
	C	N	7.66	0.46	432	151
		Y	13.94	16.14	50	301
	D	N	7.66	0.46	829	267
		Y	13.95	16.19	50	31

Table 5.5: Key: I=independent, C=clustered, D=dependent model. Comparison of LAO* and ALAO* for graph in Figure 4.8c. All times averaged over 10 runs. In the dependent belief state, edges $A \to G$ and $C \to G$ are fully dependent on each other.

LAO*. The smaller size of G in the dependent model starting from the dependent initial belief state is due to the total dependency between the edges limiting the number of possible worlds, as explained in Section 4.6.3.1. The same effect can be observed in the clustered/dependent models in Table 5.5.

There are two interesting aspects to the results in Table 5.5. Firstly, the size of the explicit graph is vastly reduced under ALAO*, where from the uniform initial belief state it has explored at most 7.7% of the state space compared to normal LAO*, and 11.5% from the dependent initial belief state. This is also evident in the computation times where ALAO* finishes in a fraction of the time required by standard LAO*. Secondly, the average cost of ALAO* on the 3 edge graph is worse in every test but one. From the dependent initial belief under the independent model, the average cost in ALAO* is lower than LAO*, though barely statistically significant. Repeating the test with 500,000 simulator trials (shown in Table 5.6) allows us to be more confident of the result. The costs are similar to the previous experiment so we can be sure of the results. The percentage of failed trials reveals the cause. Recall that the simulator does not include the costs of failed trials in the statistics. The higher failure rate under ALAO* shows that the policy is not superior, rather that in cases where it fails, standard LAO* succeeds in reaching the goal with a marginally higher cost, causing the average to increase.

The higher policy costs for ALAO* in Table 5.5 show one of the drawbacks of approximation: belief states that are equal under approximation in terms of K-L distance can hide belief states necessary to find the optimal policy in some cases. The maximum costs for all of the ALAO* policies were massively increased compared to normal LAO*;

Algorithm	**Avg.cost**	**% fail**
LAO*	8.34	25.01
ALAO*	8.18	39.32

Table 5.6: The results of running LAO* and ALAO* on the 3 edge graph under the independent MDP model starting from a dependent initial belief state. For this test 500,000 simulator trials were run.

in fact ALAO* produces the same behaviour witnessed in Section 4.6.1.3 where the agent hops between nodes A and B. We can investigate this by examining the exact policy the agent follows—we show the policy followed by the independent model from the uniform belief state. The standard LAO* policy has already been shown in Section 4.6.3.1 ("Independent model" on page 131), the ALAO* policy is shown below:

1. Start at S.
2. Go to node A. Receive observation of $A \to G$ at A. If $A \to G$ is free, then go to goal node G, otherwise go to step 3.
3. Go to node B and receive observation of $C \to G$. If "free" observed, then continue to C as in step 3 of policy in Section 4.6.3.1, otherwise if "blocked" observed, go to step 4.
4. Go to node A.
5. Go to node B. Receive observation of $C \to G$ at B. If "free" observed then return to step 2, otherwise if "blocked" observed then return to step 4.

In standard LAO*, the observation received at B (step 3 above) did not affect the agent's actions, it continued to node C anyway. With ALAO*, it continues to C only if it receives a "free" observation. If it receives a "blocked" observation, it hops between node A and B until it receives a "free" observation which causes it to return to the same belief state as it had before it ever visited node B and return to step 2. Following this to conclusion, once the agent is trapped in the loop between nodes A and B, it must receive two consecutive "free" observations to finish the graph. If it does, it reaches the goal with a very high cost, otherwise it fails due to exceeding the maximum number of trial steps allowed in the simulator.

Working from the uniform initial belief and the independent model and reducing $\max_{\text{KL}}$ eventually leads ALAO* to find the same policy as standard LAO* as shown in Table 5.7. Once $\max_{\text{KL}} = 0.0001$, the policy is identical to standard LAO* and the same average cost is achieved. The fact that the 3 edge graph is unsafe may be the underlying reason for the poor policy. If we modify the graph to make it safe by adding

$\max_{\mathbf{KL}}$	**Avg.cost**	**Max.cost**	$\lvert BSG\rvert$	**% fail**
-	7.86	11	19	12.55
0.1	8.60	55	21	19.74
0.01	13.59	60	36	13.56
0.001	13.21	58	38	13.22
0.0001	7.85	11	19	12.59

Table 5.7: The effect of lowering $\max_{\text{KL}}$ on the 3 edge graph in Figure 4.8c. A uniform initial belief state and independent model were used.

$\max_{\mathbf{KL}}$	**Avg.cost**	**Max.cost**	$\lvert BSG\rvert$	**% fail**
-	11.99	41	18	0
0.1	11.97	41	18	0

Table 5.8: Performance of ALAO* on the 3 edge graph when an extra edge $S \to G$ with cost 30 is added.

an edge from S to G with cost 30, then we obtain the results in Table 5.8. When the agent is guaranteed to have a safe route to the goal, then ALAO* performs identically when $\max_{\text{KL}} = 0.1$. The extra edge eliminates the possibility of failure since the agent can always give up and take the expensive path to the goal.

5.2.2.2 ALAO* on larger examples

ALAO* was run on the same set of example graphs as the algorithms demonstrated in Section 4.6.3.4. Settings such as discretisation, memory and simulation settings remained the same. Table 5.9 shows the equivalent results obtained with ALAO* with $\max_{\text{KL}} = 0.1$. The results in Table 5.9 can be directly compared to the results from standard LAO* in Table 4.15 on page 139. We can see that the performance (i.e. average cost) achieved by ALAO* is very similar. In nearly all cases, the minimum, maximum and average costs are the same across LAO* and ALAO*, while the solution times for ALAO* are often substantially smaller.

Graph	**Model**	**Min.**	**Avg.**	**Max.**	**Std.dev**	**% fail**	**Time (ms)**
Graph 1	I	916.26	1054.64	2278.00	330.03	0	78
	C	916.26	1019.06	1929.02	229.04	0	224
	D	916.26	1019.62	1929.02	230.05	0	1091
Graph 2	I	1350.84	1475.76	2096.8	188.26	0	193
	C	1350.84	1475.97	2096.8	188.58	0	158
	D	1350.84	1469.04	1992.74	173.02	0	582

Table 5.9: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy.

Graph	Model	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
Graph 3	I	646.92	985.81	1430.84	365.00	0	134
	C	646.92	912.01	1360.15	254.19	0	424
	D	646.48	912.05	1361.71	252.32	0	1285
Graph 4	I	1153.57	1211.44	1452.10	110.16	0	116
	C	1153.57	1211.51	1452.10	110.23	0	179
	D	1153.57	1211.45	1452.10	110.18	0	521
Graph 5	I	1433.93	1620.81	2964.80	235.95	25.03	69582
	C	1433.93	1662.01	3778.19	303.94	25.05	4382
	D	1433.93	1657.38	3690.29	294.19	25.00	5395
Graph 6	I	1054.93	1499.90	2215.14	445.85	0	247
	C	1054.93	1425.81	2123.38	370.65	0	341
	D	1054.93	1425.09	2123.38	370.65	0	1069
Graph 7	I	861.98	1120.64	1340.49	237.82	0	38
	C	861.98	1121.49	1340.49	237.74	0	36
	D	861.98	1120.18	1340.49	237.76	0	63
Graph 8	I	903.12	1094.00	1918.72	381.68	0	368
	C	903.12	1091.65	1918.72	379.87	0	331
	D	903.12	1092.23	1918.72	380.31	0	1009
Graph 9	I	775.28	1134.56	2274.68	566.65	0	457
	C	775.28	1044.76	1701.28	302.76	0	430
	D	775.28	1043.18	1462.22	300.38	0	2814
Graph 10	I	680.45	844.12	1350.07	259.30	0	78
	C	678.07	844.34	1350.6	259.61	0	122
	D	678.07	844.18	1350.6	259.54	0	380
Graph 11	I	1072.92	1219.76	1909.04	314.91	0	812
	C	1072.92	1149.34	3125.11	231.34	10.92	1642
	D	1072.92	1150.07	3024.49	233.23	10.59	7290
Graph 12	I	1329.92	1329.92	1329.92	0	0	120
	C	1329.92	1329.92	1329.92	0	0	112
	D	1329.92	1329.92	1329.92	0	0	392
Graph 13	I	F	F	F	F	F	F
	C	F	F	F	F	F	F
	D	F	F	F	F	F	F
Graph 14	I	1627.54	2064.01	4878.91	472.88	92.47	6844
	C	1587.57	2759.57	5402.12	893.23	92.16	4631
	D	1587.57	1986.20	5019.33	516.34	92.01	5625
Graph 15	I	778.81	1140.15	2022.33	526.90	0	234
	C	778.81	1140.31	2022.33	526.95	0	282
	D	778.81	1139.31	2022.33	526.56	0	1193
Graph 16	I	1795.23	1795.23	1795.23	0	0	481
	C	1795.23	1795.23	1795.23	0	0	338
	D	1795.23	1795.23	1795.23	0	0	1331
Graph 17	I	1403.93	1725.63	2192.85	385.93	0	43
	C	1403.93	1722.12	2192.85	373.70	0	84
	D	1403.93	1722.07	2192.85	373.66	0	232

Table 5.9: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy.

Graph	Model	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
Graph 18	I	1050.69	1123.25	1663.71	182.16	0	16
	C	1050.69	1123.34	1663.71	182.22	0	12
	D	1050.69	1123.30	1663.71	182.16	0	43
Graph 19	I	1174.50	1337.43	1621.76	213.93	0	59
	C	1174.50	1311.75	1621.76	176.64	0	80
	D	1174.50	1312.40	1621.76	176.86	0	290
Graph 20	I	784.98	1298.65	2950.27	673.95	0	1098
	C	784.98	1295.88	2950.27	672.33	0	1530
	D	784.98	1290.00	3018.62	658.84	0	5406

Table 5.9: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy. Comparison of ALAO* models across the range of example graphs. All times averaged over 10 runs. $d = 0.00001$, $\max_{\mathsf{KL}} = 0.1$ and the graph pre-processor is used for all MDP models. All costs derived from 50,000 simulator trials.

ALAO* produces results for many of the graphs where LAO* failed to under the dependent model. This can be seen in Graphs 1, 3, 12, 16 and 17. In other cases ALAO* produces results for all models where at least two out of the three failed before, such as Graphs 5, 8, 11, 14 and 20. This difference in behaviour is due to the approximation employed in ALAO*. The mapping of many similar belief states to one reduces the memory footprint of the algorithm, thereby allowing it to explore the necessary amount of the state space to complete policies, where before, LAO* ran out of memory.

Graph 13 still fails under all models with ALAO*, showing that scalability issues still affect the algorithm. The previously unsolved Graph 14 now has policies for all three models under ALAO*, but the policies are unlikely to be optimal. The average cost for the clustered model is significantly higher than the independent model and the maximum costs for all three models are extremely high. A similar effect is seen in Graph 5 which is also unsolvable with LAO*. The high fraction of trials that failed with both of these graphs suggests a common factor: they are both unsafe graphs. As demonstrated earlier with the 3 edge graph, ALAO* finds it harder to produce good policies when there is no guaranteed path to the goal. The failure rate is an unavoidable consequence of the graphs being unsafe—the agent cannot reach the goal in some fraction of the trials. The high maximum signals that even in solvable trials, the path being chosen is not the best one.

Graph 11 shows an anomalous result in that both the clustered and dependent models report failure rates of approximately 10% under ALAO*, while under the independent model, or any model under LAO*, no failures are reported. Graph 11 has a safe route to the goal, so there should be no circumstance where the agent cannot succeed. This indicates that the approximation is preventing the optimal policy from being found. Table 5.10 shows results from Graphs 11 and 14 with a lower threshold. When using a lower threshold of $\max_{\mathrm{KL}} = 0.01$ the high maximum route costs seen in both graphs decrease

Graph	$\max_{\mathbf{KL}}$	Model	Min.	Avg.	Max.	Std.dev	% fail	Time
Graph 11	0.1	I	1072.92	1219.76	1909.04	314.91	0	812
		C	1072.92	1149.34	3125.11	231.34	10.92	1642
		D	1072.92	1150.07	3024.49	233.23	10.59	7290
	0.01	I	1072.92	1216.92	1909.04	312.40	0	3515
		C	1072.92	1219.20	1909.04	314.38	0	15744
		D	1072.92	1217.94	1909.04	313.44	0	110898
Graph 14	0.1	I	1627.54	2064.01	4878.91	472.88	92.47	6844
		C	1587.57	2759.57	5402.12	893.23	92.16	4631
		D	1587.57	1986.20	5019.33	516.34	92.01	5625
	0.01	I	1242.67	1561.07	2840.63	238.81	92.26	30319
		C	1165.83	1380.27	3041.12	259.52	92.10	131609
		D	1165.83	1322.04	2622.95	198.88	92.03	184620

Table 5.10: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy. Time is in milliseconds. Demonstration of the effect of a lower $\max_{\mathsf{KL}}$ of 0.01 on two graphs. $d = 0.00001$ and the graph pre-processor used for all MDP models. All costs derived from 50,000 simulator trials. $\max_{\mathsf{KL}} = 0.1$ results copied from Table 5.9.

Model	$\max_{\mathbf{KL}}$	Min.	Avg.	Max.	Std.dev	% fail	$\lvert BSG \rvert$
C	0.1	678.07	844.34	1350.60	259.61	0	44
C	0.01	678.07	840.05	1350.60	219.14	0	41
C	0.001	678.07	838.33	1350.60	218.59	0	41
C	0.0001	678.07	839.06	1350.60	219.36	0	41
D	0.1	678.07	844.18	1350.60	259.54	0	46
D	0.01	678.07	839.54	1350.60	219.28	0	43
D	0.001	678.07	838.64	1350.60	219.90	0	43
D	0.0001	678.07	839.16	1350.60	218.86	0	50

Table 5.11: Varying the $\max_{\mathsf{KL}}$ threshold for Graph 10 (pre-processed) with clustered/dependent ("C"/"D") ALAO*. All other parameters set as in Table 5.9.

by a large amount. The small proportion of failures seen in the clustered/dependent models in Graph 11 also no longer occur with both models now performing the same as the independent model. The solution times have increased as a result of the lower $\max_{\mathrm{KL}}$ threshold. In Graph 14, the high failure rate remains unaffected as it is a consequence of the graph's construction, though the average and outlying costs reported are all much lower, which is also reflected in the lower standard deviation. The lower threshold used here has improved the policy in both graphs as a result of the finer grained approximation.

The final notable result from Table 5.9 is in Graph 10, where the clustered and dependent models perform the same as the independent model, instead of achieving a lower cost as they do under LAO* (see result in Table 4.15). In this case, it is likely that the approximation threshold $\max_{\mathrm{KL}} = 0.1$ is preventing the true optimal policies from being discovered. Table 5.11 shows the results of different $\max_{\mathrm{KL}}$ thresholds for Graph 10. As soon as $\max_{\mathrm{KL}}$ is reduced in either model the minimum, average and maximum costs improve to match standard LAO*. The precise topology of the graph plays a fundamental role in the algorithm's behaviour; however it highlights the fact that in marginal cases, setting the threshold at 0.1 can be too high. This is investigated further in Section 5.2.2.5.

5.2.2.3 Cluster configuration

Table 5.9 also contains an interesting point on cluster configurations that was not visible under standard LAO*. In Graphs 17 and 20, the dependent model produces a policy under ALAO* with a lower average cost than the independent model. In Graph 17, the clustered model also achieves the lower cost, whereas in Graph 20 it does not. The difference in average costs between models is small in both graphs, however the standard deviation in both models shows that the spread of route costs is reduced in the dependent model, suggesting better overall performance. Graph 20 contains a significant dependency between two edges that are in different clusters, whereas Graph 17 only contains dependencies between edges within the same cluster. This shows where the cluster configuration can be important to the performance of that model. If the dependent edges are not in the same cluster, the clustered model cannot exploit them.

5.2.2.4 ALAO* computation times

ALAO* obtains large gains in policy computation time as a result of the approximation mechanism employed. In terms of the implementation, calculating K-L distances between belief states does add some small overhead to the algorithm. Every equality test between two belief states in ALAO* requires computing the K-L distance between them as opposed to the simple numerical equality test employed in standard LAO*. Generally in a normal run of the algorithm, several thousand such equality tests will occur, particularly during node expansion. When new belief states are reached as the result of an observation, the system must decide whether to create a new node in G for the new belief, or whether it is approximately equal to an existing one based on the $\max_{\mathrm{KL}}$ threshold. However, the overhead of these calculations is far outweighed by the gain from having fewer nodes in G. The time savings result both from the small subsection of the state space explored and from having fewer nodes on which to run VI in the convergence phase of the algorithm.

Table 5.12 shows the performance of LAO* and ALAO* for Graphs 1 to 10 from above under the clustered model. The reduction in the quantity of nodes explored and computation time taken with ALAO* are shown, normalised to LAO*. The relationship between the two is characterised in Figure 5.8 which shows the correlation between the size of the explicit graph and the overall time needed for the LAO* algorithm to complete. The correlation co-efficient for the data shown is $r = 0.99$ indicating a very strong relationship between the two, backing up earlier data that showed a similar pattern.

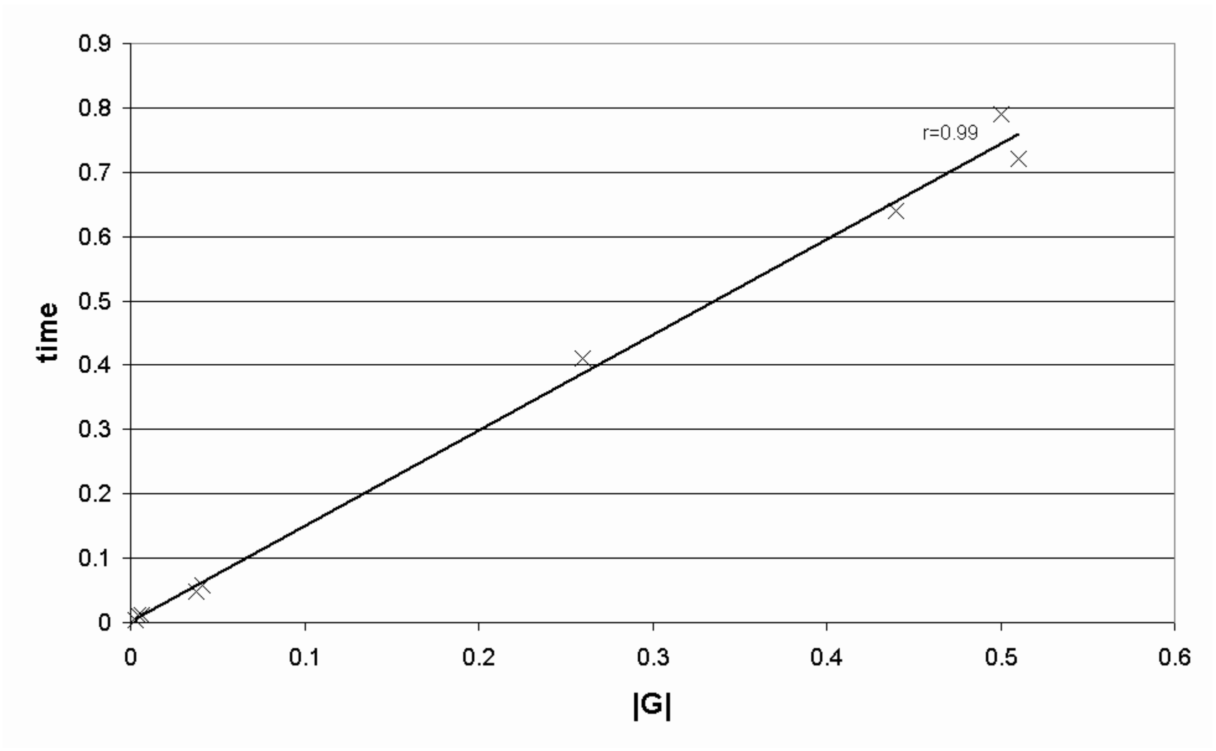


Figure 5.8: Plot of the reduction in size of the explicit graph against the reduction in time when using ALAO* versus LAO*.

Graph	**ALAO*?**	$\lvert G\rvert$	$\lvert BSG\rvert$	**T (ms)**	**T $\frac{\textbf{ALAO*}}{\textbf{LAO*}}$**	$\lvert G\rvert$ $\frac{\textbf{ALAO*}}{\textbf{LAO*}}$
Graph 1	N	287853	45	51274	1.0	1.0
	Y	934	45	224	0.0043	0.0032
Graph 2	N	1546	216	220	1.0	1.0
	Y	788	169	158	0.72	0.51
Graph 3	N	198120	251	36905	1.0	1.0
	Y	1308	80	424	0.011	0.0066
Graph 4	N	1089	273	228	1.0	1.0
	Y	548	141	179	0.79	0.50
Graph 5	N	∼300000	∼10000	F	1.0	1.0
	Y	1542	117	4382	-	-
Graph 6	N	24361	150	6032	1.0	1.0
	Y	1005	150	341	0.057	0.041
Graph 7	N	715	71	56	1.0	1.0
	Y	318	67	36	0.64	0.44
Graph 8	N	297562	16	28931	1.0	1.0
	Y	1325	18	331	0.011	0.0045
Graph 9	N	5923	73	1048	1.0	1.0
	Y	1528	70	430	0.41	0.26
Graph 10	N	13395	44	2482	1.0	1.0
	Y	512	44	122	0.049	0.038
				Mean reduction in ALAO*	0.29	0.20

Table 5.12: Key: T=Time. Comparison of policy computation time of LAO* and ALAO* and $|G|$ for the clustered model over 10 graphs.

5.2.2.5 Effect of increasing the $\max_{\mathbf{KL}}$ threshold

For these experiments, the $\max_{\mathrm{KL}}$ threshold is changed and higher values are selected to show how policy performance deteriorates as coarser approximation is used. All other settings are left unchanged.

		$\max_{\mathbf{KL}} = 0.1$				$\max_{\mathbf{KL}} = 0.5$			
Graph	**Model**	$\lvert G\rvert$	$\lvert BSG\rvert$	**% fail**	**Time**	$\lvert G\rvert$	$\lvert BSG\rvert$	**% fail**	**Time**
Graph 1	I	523	91	0	79	337	99	5.35	83
	C	934	45	0	223	270	91	5.30	81
	D	1129	45	0	1101	305	47	0	156
Graph 2	I	976	169	0	195	731	166	0	133
	C	788	169	0	161	548	145	0	106
	D	771	143	0	590	503	115	0	321
Graph 3	I	722	50	0	130	344	99	0	61
	C	1308	80	0	423	562	59	0	148
	D	1171	108	0	1285	590	113	30.93	468
Graph 4	I	516	145	0	111	401	121	0	89
	C	548	141	0	176	341	127	0	110
	D	556	145	0	523	349	131	0	302
Graph 5	I	10811	636	24.91	70261	1647	96	35.35	7696
	C	1542	117	24.99	4452	670	49	44.85	3254
	D	1754	81	24.94	5440	772	60	41.10	3650
Graph 6	I	973	121	0	243	295	100	26.65	71
	C	1005	150	0	340	355	105	0	81
	D	1077	156	0	1069	375	111	0	235
Graph 7	I	358	71	0	37	223	55	0	23
	C	318	67	0	36	215	55	0	23
	D	304	67	0	63	215	55	0	42
Graph 8	I	1385	18	0	362	344	17	14.22	79
	C	1324	18	0	330	332	17	14.24	71
	D	1365	18	0	1007	347	18	14.29	145
Graph 9	I	1673	94	0	439	755	98	0	164
	C	1528	70	0	424	750	46	0	160
	D	1502	56	0	2825	963	52	0	1458
Graph 10	I	430	24	0	77	125	40	20.34	26
	C	512	44	0	122	135	34	0	29
	D	541	46	0	380	141	35	0	70
Graph 11	I	2089	161	0	799	698	189	12.15	166
	C	2804	345	10.78	1640	741	187	12.04	205
	D	2955	346	10.66	7339	600	154	13.46	537
Graph 12	I	624	7	0	114	376	153	32.01	96
	C	605	7	0	109	364	153	31.97	97
	D	656	7	0	390	399	175	24.00	6908

Table 5.13: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy. Times in ms.

		$\max_{\mathbf{KL}} = 0.1$				$\max_{\mathbf{KL}} = 0.5$			
Graph	**Model**	$\|G\|$	$\|BSG\|$	**% fail**	**Time**	$\|G\|$	$\|BSG\|$	**% fail**	**Time**
Graph 13	I	F	F	F	F	1566	307	30.09	630
	C	F	F	F	F	1859	381	28.02	968
	D	F	F	F	F	2805	408	16.84	13726
Graph 14	I	731	171	92.45	6798	116	39	91.58	128
	C	990	104	92.27	4496	140	43	91.96	1563
	D	971	89	91.96	5505	144	43	91.90	1473
Graph 15	I	941	69	0	224	308	69	2.03	62
	C	1067	69	0	279	333	68	0	69
	D	1069	69	0	1193	375	68	0	239
Graph 16	I	1568	7	0	469	937	7	0	255
	C	1135	7	0	336	418	7	0	92
	D	1140	7	0	1336	430	7	0	330
Graph 17	I	283	64	0	41	219	63	0	33
	C	452	110	0	82	161	60	31.53	31
	D	466	115	0	236	162	62	0	72
Graph 18	I	94	24	0	15	67	19	0	10
	C	94	24	0	51	67	19	0	20
	D	99	26	0	41	72	19	0	29
Graph 19	I	508	133	0	57	400	129	0	45
	C	678	185	0	78	491	155	0	60
	D	686	185	0	291	508	155	0	211
Graph 20	I	2484	136	0	1085	416	103	17.56	158
	C	2769	136	0	1520	428	103	17.63	142
	D	2371	114	0	5404	434	103	30.75	450

Table 5.13: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy. Times in ms. Experiment testing the effects of increasing the $\max_{\mathrm{KL}}$ threshold on policy performance.

The graphs in Figures 5.9 to 5.12 (shown on pages 185 to 186) show the typical behaviours observed across the whole set of graphs. When the maximum K-L distance threshold ($\max_{\mathrm{KL}}$) is altered, the effects can manifest in different ways as explained below. In the range $0.1 \le \max_{\mathrm{KL}} \le 0.5$, we see three general trends. The most common behaviour is seen in Graphs 2,4,6,7,9,15,16,18 and 19 where the average policy cost stays relatively flat even when high values of $\max_{\mathrm{KL}}$ are used, producing policies that perform almost identically to ones produced under a lower threshold. Typical behaviour for these graphs is shown in Figure 5.11. If not many observations are made during execution of the policy, or if those that are made have a high confidence (i.e. are good quality), then changes in the belief state will be fairly large and not very frequent. We would expect ALAO* to perform well with a high threshold under such circumstances because the few changes in belief state will remain above the threshold for new node creation.

The second type of behaviour is where the performance suffers significantly at higher thresholds. Figures 5.9 and 5.12 show this type of behaviour which is exhibited by Graphs 1,5,8,11,12 and 13. At higher thresholds, the policy may appear better due to a lower average cost; however, consulting Table 5.13, we see that the failure rate also rises. This can be seen in many of the graphs in Table 5.13 and the reasons for the lower cost results are very similar to those described in Section 5.2.2.1. The cause is that the policy is worse at higher approximation thresholds. With belief state approximation, exploration will search the same areas of the state space as standard LAO*, but key changes in the belief state can be hidden from the search. With a high $\max_{\mathrm{KL}}$, if a change between two belief states necessary to find the optimal policy is hidden by the approximation, then the algorithm will find a different, non-optimal policy. This is very similar to coarse discretisation resolutions which can hide important belief state changes, as described in Section 4.6.2.

The final effect on the policy that can be induced by a high threshold is shown in Figure 5.9 at $\max_{\mathrm{KL}} = 0.4$ where there is a marked increase in the average cost for the dependent model before decreasing again at $\max_{\mathrm{KL}} = 0.6$. There are no reported failures in this model at 0.4 or 0.5. ALAO* is using a less efficient route than before, but avoids making illegal moves or causing the agent to run out of time in the simulator. Figure 5.12 shows an interesting peak at $\max_{\mathrm{KL}} = 0.3$ where all three models show an isolated increase in average cost. The effects of changing the threshold can have subtle effects specific to some graphs. What is happening in Figure 5.12 is that the policies found at $\max_{\mathrm{KL}} = 0.3$ have a lower failure rate across all models (zero in the independent and clustered models, 8.11% in the dependent) than the policies found at neighbouring thresholds which have failure rates in the range [10.32–14.30]%. We found the same pattern in the dependent model in Graph 1 (Figure 5.9). A combination of the side effects of approximation are present. At some thresholds the failures induced by the policy are reducing the average cost as explained above, while at others, the chosen policy is simply choosing "safer" routes to the goal, thus pushing the average cost up. Across several graphs, a correlation between the failure rate and average cost can be seen where the average cost decreases as the failure rate increases. Neither is a desirable property, however because we would prefer to choose a lower $\max_{\mathrm{KL}}$ where efficient routes are chosen without the presence of failures.

Another common feature seen across many of the graphs is that the degradation in policy performance occurs suddenly at particular thresholds, rather than gradually as the threshold increases. Graph 1 is exceptional in that the average cost obtained by the clustered model degrades gradually as $\max_{\mathrm{KL}}$ increases from 0.2 to 0.6. The typical pattern is seen in Figure 5.12 where the returned policy is much worse above a specific $\max_{\mathrm{KL}}$. This is to be expected, since the average cost the agent obtains in the simulator is derived from the policy it follows. Each belief state in the MDP is an approximation of a POMDP belief state which would have an associated α-vector (as in PBVI). As mentioned above, certain important belief states can be hidden as $\max_{\mathrm{KL}}$ increases. The removal of a belief state may prevent the discovery of an important α-vector and its associated action which is used in the optimal policy. The resultant policy is therefore likely to be significantly worse because it will utilise an action that leads to lower overall performance.[1] The behaviour is somewhat dependent upon the particular PRM graph, but this implies that multiple "cliff edges" in the average cost will be observed as the threshold increases. Once the threshold increases to the point where the belief states are so approximate that the agent has trouble differentiating between free and blocked edges, then failures start occurring in the simulator because the agent attempts to traverse blocked edges.

The policy computation times for Figures 5.9 to 5.12 are shown in Figure 5.13. The explicit graph sizes in Table 5.13 show how $|G|$ tends to be much smaller at $\max_{\mathrm{KL}} = 0.5$ compared to 0.1 explaining the trends shown in the timings. As the greater approximation effectively reduces the number of distinguishable belief states, ALAO* creates fewer individual nodes in G which in turn reduces the amount of computation necessary to find the policy. As shown in Table 5.13 and Figure 5.13, Graph 5 takes much longer to solve than all the other graphs. This is again due to the fact that it is unsafe. As mentioned in Section 5.2.2.1, unsafe graphs can cause problems for the algorithm, making it harder to find a policy. Table 5.14 compares the times for Graphs 1 and 5 using the clustered model and additionally shows the number of iterations of VI used during the convergence phase. This value should not be considered an accurate measure of how much time the algorithm spent performing node backups since it simply counts the total number of iterations of the convergence phase during the execution of ALAO*. It does not account for local backups during the expansion phase or the number of nodes backed up each iteration (which depends on the size of the BSG). However, it can give an indication as to how much value iteration was required before a graph converged. The amount of time spent in VI is greatly increased in Graph 5. Due to the high discount rate of utilised for all (A)LAO* experiments, many iterations of VI are required to reach convergence in the states from where the goal cannot be reached. For states with paths that lead to the goal, convergence will typically only require a few iterations. In unsafe graphs where the goal is not reachable for some subsection of the state space, the true cost is infinite for those states. The discounted cost will be finite, but requires many more iterations of VI to converge. This causes the timings seen for Graph 5 and to a lesser extent Graph 14.

[1]ALAO* does not explicitly represent α-vectors because the value function is implicitly contained within the state graph it creates.

$\max_{\mathbf{KL}}$	**Graph 1**			**Graph 5**		
	Time (ms)	$\lvert G\rvert$	**VI**	**Time (ms)**	$\lvert G\rvert$	**VI**
0.05	444	1368	3	23136	4273	5159
0.1	223	934	3	4452	1542	3425
0.2	96	504	17	3700	973	6138
0.3	77	431	21	2849	649	4579
0.4	93	337	52	2671	675	4452
0.5	81	270	55	3254	670	5185
0.6	86	271	53	2878	612	4746

Table 5.14: Key: VI = value iteration. A comparison of the iterations of VI for graphs 1 and 5 under the clustered model.

Generally, increasing the maximum K-L distance threshold for ALAO* will reduce the computation time, since there are fewer nodes in the explicit graph, but the increased approximation in distinguishing belief states can significantly impact the quality of the policy returned. In all the experiments conducted, if the threshold was kept at 0.1 or below, then the returned policy obtained simulated results that were virtually identical to standard LAO* but with a much lower computation time. This also suggests that in LAO*, many of the belief states in G are expanded unnecessarily and do not contribute to improving the policy. A useful avenue for future research would be to investigate the relationship between graph observations and their relevance to the belief states explored. If belief states that were likely to be useful in the optimal policy could be estimated beforehand, it could be used to estimate the largest $\max_{\mathrm{KL}}$ threshold usable, without negatively impacting the policy. One strategy would be to collect a set of beliefs via random graph walks (as PERSEUS does), then calculate the K-L distance between pairs of states for a significant subset of these. Setting $\max_{\mathrm{KL}}$ close to the median of these measured divergences would then remove many discrete belief states, but should preserve policy quality. A possible drawback is that the chosen threshold may be too low to provide any real benefit in computation time.

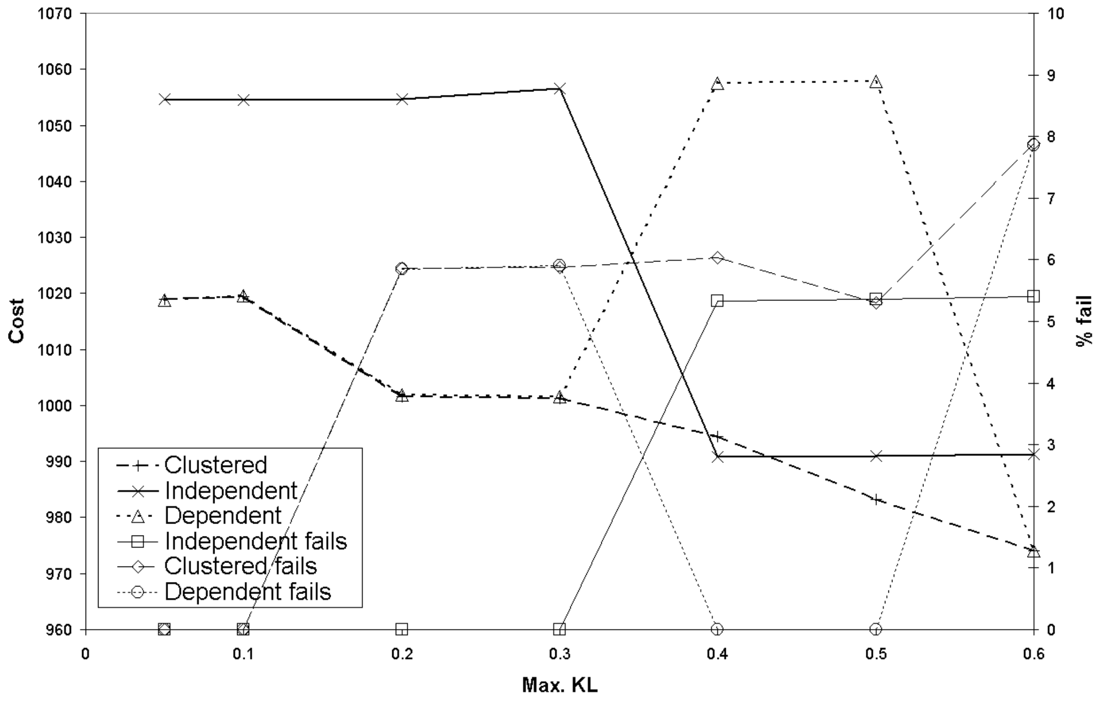


Figure 5.9: The effect of changing the $\max_{\mathsf{KL}}$ threshold on the performance of Graph 1.

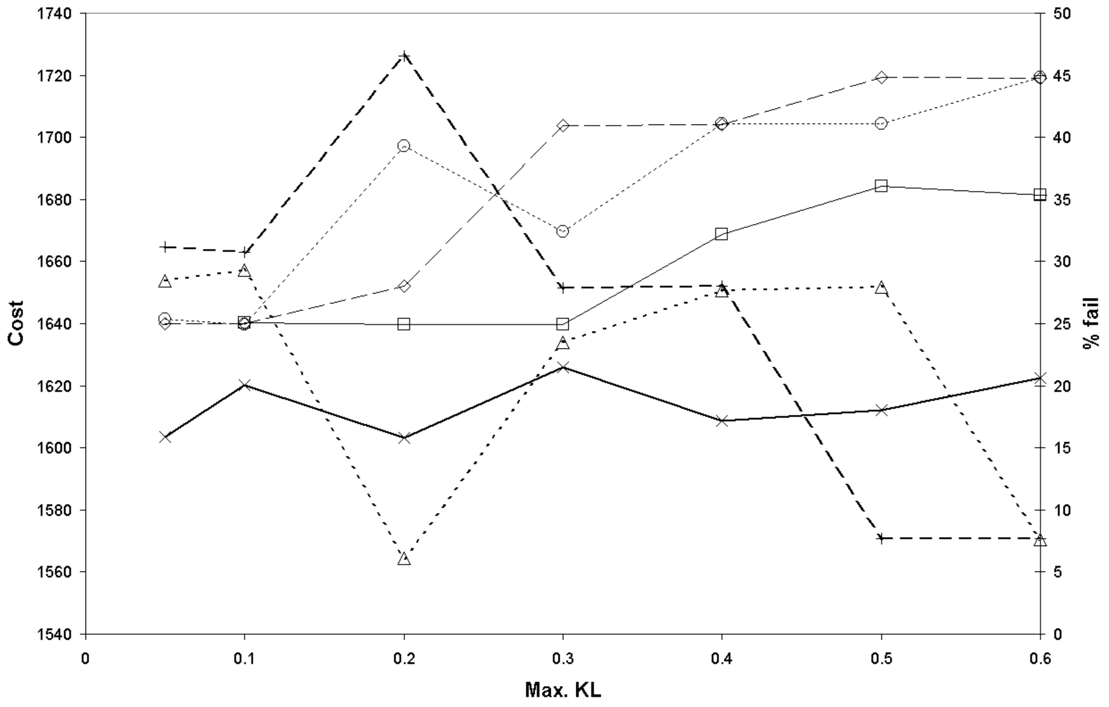


Figure 5.10: The effect of changing the $\max_{\mathsf{KL}}$ threshold on the performance of Graph 5.

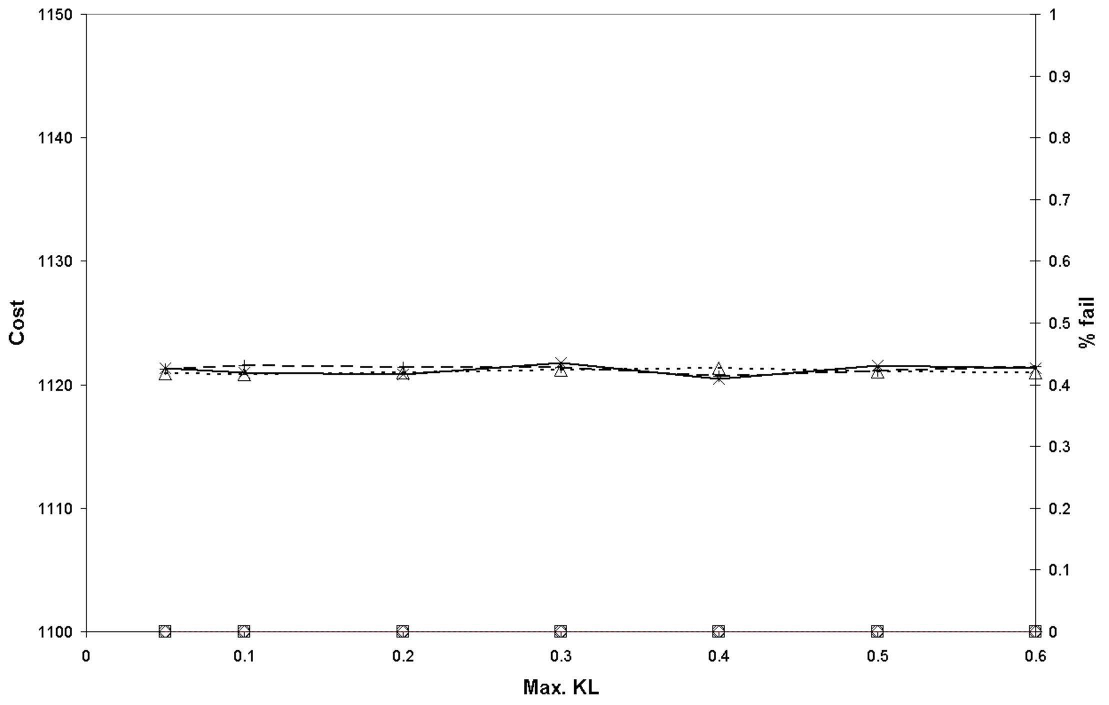


Figure 5.11: The effect of changing the $\max_{\mathsf{KL}}$ threshold on the performance of Graph 7.

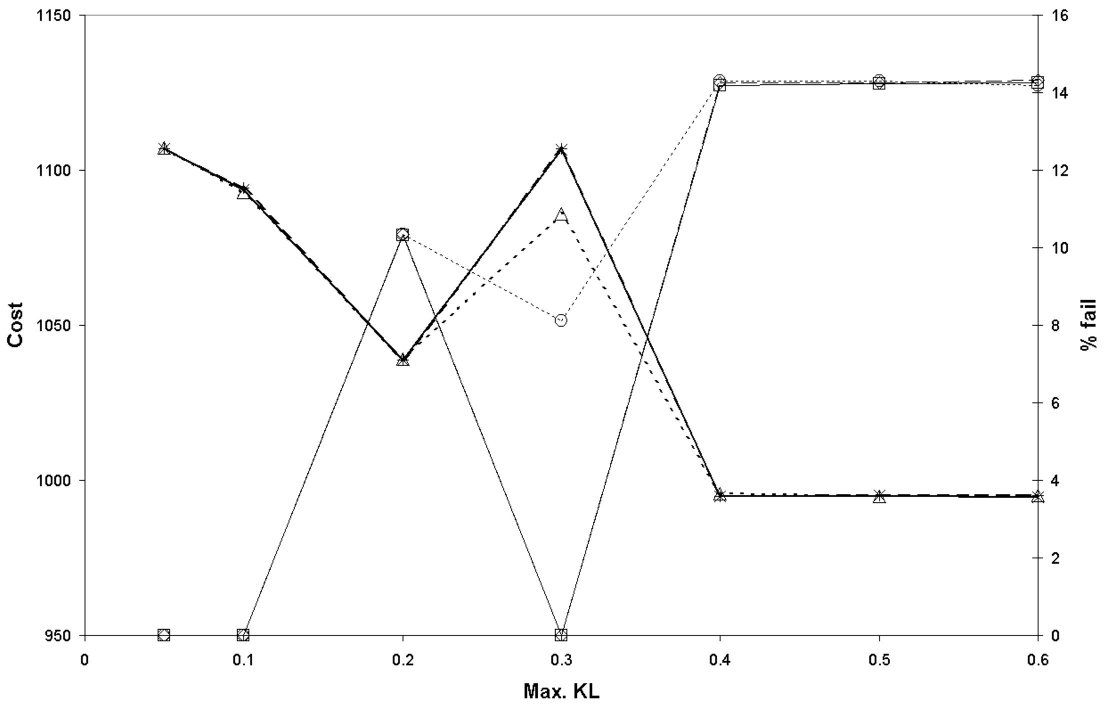


Figure 5.12: The effect of changing the $\max_{\mathsf{KL}}$ threshold on the performance of Graph 8.

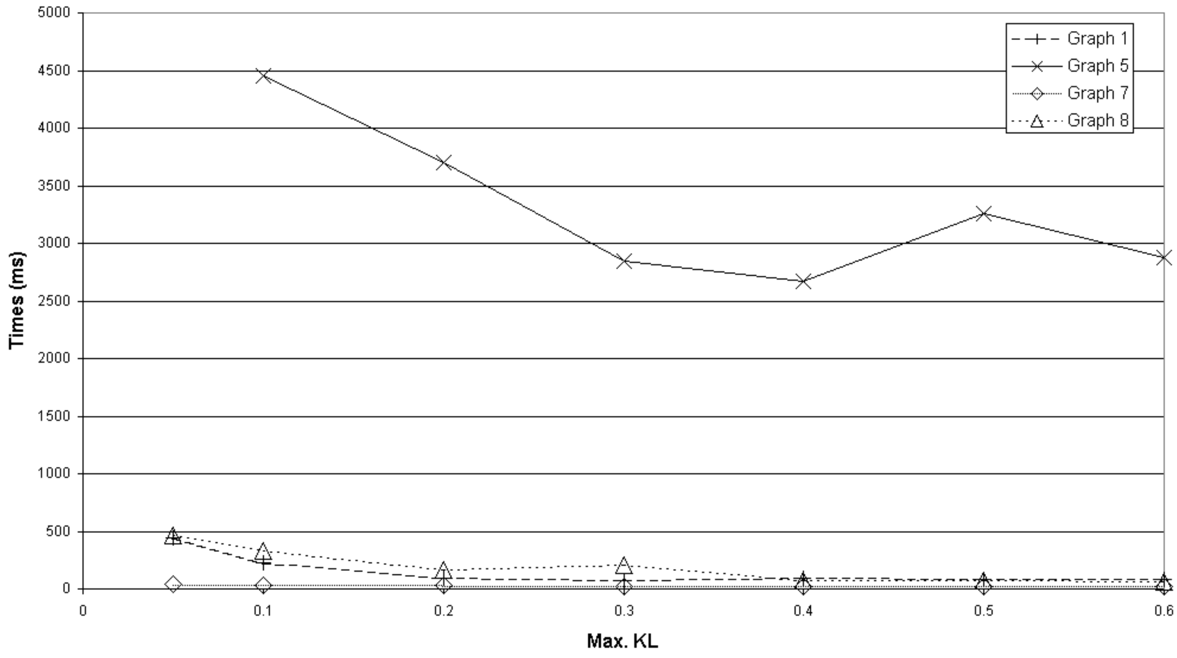


Figure 5.13: The effect of $\max_{\mathsf{KL}}$ on computation time for the clustered model on graphs from Figures 5.9 to 5.12. Each data point averaged over 10 runs of ALAO*.

5.3 Scaling

In this Section we investigate how scalable the algorithms are by increasing the size of the PRM graphs and number of uncertain edges. As has been demonstrated, the performance of LAO* depends on the amount of state space explored, i.e. the number of belief states and therefore MDP states that it visits. As the belief state can only change when an observation introduces new information, then we expect that the number of uncertain edges m, to be the primary contributing factor that affects algorithm scalability.

5.3.1 Increasing graph size

We first test the effects of increasing the number of nodes n, in the PRM graph. In all the graphs in Appendix A, n was in the range $[40, 80]$. In larger real-world scenarios, we would expect typical PRM graphs to contain more nodes either to cover a larger area, to increase node density, or both. While more nodes will obviously have an impact on LAO*, they are unlikely to have a major effect. Each new PRM graph node will become an extra node in LAO*'s explicit graph. Consider a simple graph where the agent makes one observation of an edge soon after the start node and then chooses one of two possible routes to the goal based on the result. The BSG for the policy will consist of one chain of states from the start node to the observation, then split into two chains leading to the goal. Adding nodes to the PRM graph near the goal means that each of the two branches of the BSG after the observation will increase in length because the agent must visit each node. Each additional PRM node will add two nodes to the BSG, thus there is a definite impact of PRM graph size on BSG size. However, now consider the impact that the observation had: without it, the BSG would simply consist of one chain of single child nodes from start to goal,[1] but with it the chain essentially splits into two at the point the observation is made. The number of nodes in the BSG from the observation point to the goal is doubled—the presence of an uncertain edge has a far greater effect on the size of the BSG. Of course, in this example we assumed the agent must traverse the same number of nodes to reach the goal after the observation, which in most cases will not be true.

We created a PRM graph with 2000 nodes using the same obstacles as Graph 17 (see Appendix A). Compared to graphs with lower node counts, the node density is significantly higher in this graph which means that most edges are far shorter. To prevent edges spanning too great a distance, D_{max} (the maximum edge length) was reduced from 300 to 150. 8 uncertain edges were placed around the corners of a few obstacles that were likely to feature on the shortest path to the goal to avoid the policy becoming a straight

[1]On graphs where no observations are made, this is precisely what LAO* creates.

Model	ALAO*?	Min.	Avg.	Max.	Std.dev	T (ms)	$\lvert G\rvert$	$\lvert BSG\rvert$
I	N	1040.23	1043.04	1054.97	2.34	153	766	329
C	N	1040.04	1042.97	1054.78	2.42	170	770	327
D	N	1040.04	1042.96	1054.78	2.41	461	882	355
I	Y	1040.23	1043.03	1054.97	2.34	166	766	329
C	Y	1040.04	1042.97	1054.78	2.42	188	766	327
D	Y	1040.04	1042.97	1054.78	2.42	636	832	355

No fails recorded in all trials.

Table 5.15: Key: I=independent, C=clustered, D=dependent model, T=time. The results from a large graph with $n = 2000$. $d = 0.00001, \max_{\mathsf{KL}} = 0.1$ and no weighting applied. All simulator costs derived from 50,000 trials. No graph pre-processing was applied on this graph.

march to the goal. This is a little unrealistic since the high node density and short edges should create many more uncertain edges; however, this was not done because a graph with high numbers of uncertain edges would be beyond the capabilities of our algorithm.

Table 5.15 shows the results using LAO* and ALAO* without the pre-processor applied. In the experiment, all parameters were kept the same except for the maximum simulator trial length which was increased to 500 steps to ensure the agent could reach the goal when the pre-processor was not used. The costs are the same across all the tests showing that for this graph, the choice of MDP model does not affect the performance. The computation times are interesting because we can see that ALAO* obtains no advantage on this graph. The amount of the state graph explored is similar between LAO* and ALAO* (identical for the independent model), so we would expect similar computation times. If few observations or only high-confidence observations are being made then ALAO*'s performance will be very close to LAO* as explained in Section 5.2.2.5. The time for the dependent model actually increases with ALAO* in both tables despite a decrease in $|G|$. The overhead of calculating the K-L distance between belief states is noticeably affecting the computation time, even though the number of nodes created is similar to that of the other MDP models. The belief states in the dependent model contain many more elements ($2^8 = 256$) than the other two models, again showing the disadvantage of using a fully dependent representation.

5.3.2 Increasing the number of uncertain edges

The number of uncertain edges affects the size of G so is relevant to LAO*'s scalability. We chose Graph 19 for this test because its topology meant that uncertain edges could be created in locations likely to influence the algorithm and agent behaviour. Uncertain edges placed in areas the agent never visits provide little information about the algorithm's scalability because LAO* will not explore them, leaving any resulting belief states unexpanded in G. To see the effect of extra uncertain edges on the algorithm, we have to ensure the agent will observe them and that they can affect the route taken. In the results below, uncertain edges were incrementally added to the graph along with observations and edge dependencies on existing edges where appropriate.[1] Tables 5.16 and 5.17 compare the costs reported for LAO* and ALAO* as the number of uncertain edges increases. The number of nodes in G explored and solution times are also compared in Figures 5.14 and 5.15. As shown in the results, all versions of the algorithm failed to produce a policy when $m = 12$. When $m < 12$ a guaranteed safe path to the goal was present, but when $m = 12$, the final uncertain edge interrupted this path. Unsafe graphs are harder for LAO*/ALAO* to solve because the goal becomes unreachable from certain sub-sections of G. A typical behaviour in these sub-sections is for the algorithm to explore increasingly lower utility areas of G seeking for a suitable path which can lead to the exhaustion of available memory, as is the case here.

The tables show that the dependent model fails when three edges are added under LAO* (Table 5.16, $m = 8$) and when four edges are added under ALAO* (Table 5.17, $m = 9$). In earlier results in Table 4.15, the dependent model is usually the first of the three models to fail. Figure 5.14 shows the dependent model under LAO* exploring many more belief states than other models, thus making larger demands on the memory system. When $m = 7$ with the dependent model, LAO* explored 38,329 nodes whilst ALAO* explored just 2,500 nodes which will clearly require a fraction of the memory. When m is greater than 7 or 8 for LAO* and ALAO* respectively, the memory runs out before a policy can be found. In different tests on other graphs and in other models in Graph 19, policies have been successfully generated after exploring far more than 38,000 nodes, so it is likely that the size of the explicit state graph grows quickly. The size of the explicit state graph is also severely limited by the belief state dimensionality. In the independent model, each new edge only adds two dimensions to the belief space; however, in the dependent model, each edge doubles the dimensionality, thus requiring double the memory space per node to store a distribution. This obviously has a large impact on the memory footprint and limits the number of nodes that can be stored. The effect becomes worse as m increases due to the exponential growth in the quantity of elements in the probability distributions.

Tables 5.16 and 5.17 show that the performance of the actual policies is the same across LAO* and ALAO* using the same model, as is the case in previous experiments. In the original version of Graph 19 where $m = 5$, the clustered and dependent model obtained a lower average cost of ~ 26 compared to the independent model (see Table 4.15

[1]Full definitions of all graphs used in this experiment, as well as the full set of 20 graphs, are available electronically from `http://www.cs.bham.ac.uk/~mlk/graphs.zip` or via the link in Appendix B. An example is shown in Appendix B.

m	Model	Min.	Avg.	Max.	Std.dev	% fail	Time (ms)
	I	1174.50	1408.88	1764.70	225.57	0	164
6	C	1174.50	1384.93	1768.64	191.35	0	176
	D	1174.50	1385.15	1768.64	191.33	0	5203
	I	1174.50	1408.96	1768.64	225.44	0	343
7	C	1174.50	1384.88	1768.64	191.23	0	450
	D	1174.50	1384.95	1768.64	191.12	0	44307
	I	1174.50	1425.01	2258.02	227.26	0	1426
8	C	1174.50	1404.08	2404.90	192.25	0	3058
	D	F	F	F	F	F	F
	I	1174.50	1425.01	2258.35	227.29	0	4241
9	C	1174.50	1403.72	2405.23	192.43	0	8795
	D	F	F	F	F	F	F
	I	1182.22	1471.63	2408.69	271.27	0	9969
10	C	1182.22	1449.30	2555.27	233.45	0	16064
	D	F	F	F	F	F	F
	I	1182.22	1471.24	2408.69	271.17	0	13714
11	C	1182.22	1449.37	2555.57	233.53	0	31624
	D	F	F	F	F	F	F
	I	F	F	F	F	F	F
12	C	F	F	F	F	F	F
	D	F	F	F	F	F	F

Table 5.16: Key: I=independent, C=clustered, D=dependent model. "F" indicates that LAO* failed to produce a policy. The effect of increasing the number of uncertain edges m, on the policy cost for LAO*. All times averaged over 10 runs. $d = 0.00001$ and graph pre-processor used for all graphs. All costs derived from 50,000 simulator trials.

on page 139). This advantage remains almost constant as more uncertain edges are added, showing that the reduced cost is the result of an observation of one of the original 5 uncertain edges, despite the fact that the added edges were often dependent on other edges. The average policy cost of all models increases at $m = 8$ and $m = 10$ because edges that were previously safe become uncertain and cause the agent to switch to a more expensive route when they are found to be blocked.

The solution times shown in Figure 5.15 and in the associated tables do not follow the same trends observed in previous experiments. Under the dependent model, ALAO* is much faster than standard LAO*, as is seen in previous experiments. Under the alternative models, LAO* computes solutions quicker than ALAO*, and when $m \geq 8$ LAO* wins out by a large margin. This is in stark contrast to previous experiments (see Section 5.2.2.4) where ALAO* outperforms LAO* due a large reduction in the explored state space. Figure 5.14 shows that this reduction is still present in all models, where ALAO* never expands more nodes in G than the independent LAO* model when $m \geq 7$. The

m	Model	Min.	Avg.	Max.	Std.dev	% fail	Times (ms)
	I	1174.50	1408.30	1764.70	225.45	0	159
6	C	1174.50	1385.01	1768.64	191.21	0	296
	D	1174.50	1385.15	1768.64	191.42	0	1499
	I	1174.50	1408.47	1768.64	225.41	0	479
7	C	1174.50	1385.33	1768.64	191.34	0	926
	D	1174.50	1385.13	1768.64	191.27	0	9524
	I	1174.50	1424.71	2258.02	227.21	0	2418
8	C	1174.50	1403.76	2404.90	192.11	0	4224
	D	1174.50	1403.76	2409.90	192.19	0	100521
	I	1174.50	1424.69	2258.35	227.12	0	9184
9	C	1174.50	1403.78	2405.23	192.48	0	15766
	D	F	F	F	F	F	F
	I	1182.22	1471.11	2408.69	271.23	0	46559
10	C	1182.22	1449.57	2555.57	233.22	0	65614
	D	F	F	F	F	F	F
	I	1182.22	1470.62	2408.69	271.21	0	96624
11	C	1182.22	1449.52	2555.57	233.88	0	158071
	D	F	F	F	F	F	F
	I	F	F	F	F	F	F
12	C	F	F	F	F	F	F
	D	F	F	F	F	F	F

Table 5.17: Key: I=independent, C=clustered, D=dependent model. "F" indicates that ALAO* failed to produce a policy. The effect of increasing the number of uncertain edges m, on the policy cost for ALAO*. All times averaged over 10 runs. $d = 0.00001$, $\max_{\mathsf{KL}}$=0.1 and graph pre-processor used for all graphs. All costs derived from 50,000 simulator trials.

reduction in $|G|$ is more pronounced when more uncertain edges are present and the difference can clearly be seen in Figure 5.14. ALAO* always expands fewer nodes than LAO* under the same model, though the differences are quite small when fewer uncertain edges are present. This appears to create a contradiction, because ALAO* is exploring far fewer nodes, yet taking longer to generate the policies. The cause is in the implementation of the algorithm. LAO* solution graphs are cyclic by definition, so any newly expanded node may include an arc to a node already present in the graph. An efficient method of locating previously created nodes is therefore a necessity for a fast implementation. We use a direct-chained hashtable for this purpose due to the approximately O(1) complexity of a lookup operation. The hash key function for a node hashes both the node's label (always numerical in our implementation) and the belief state distribution to encourage an even distribution of nodes across hash buckets. For standard LAO*, where belief states must be identical to be considered equal, this makes keys quick to compute and nodes easy to find in the hash chain for a bucket (which must be done through equality testing).

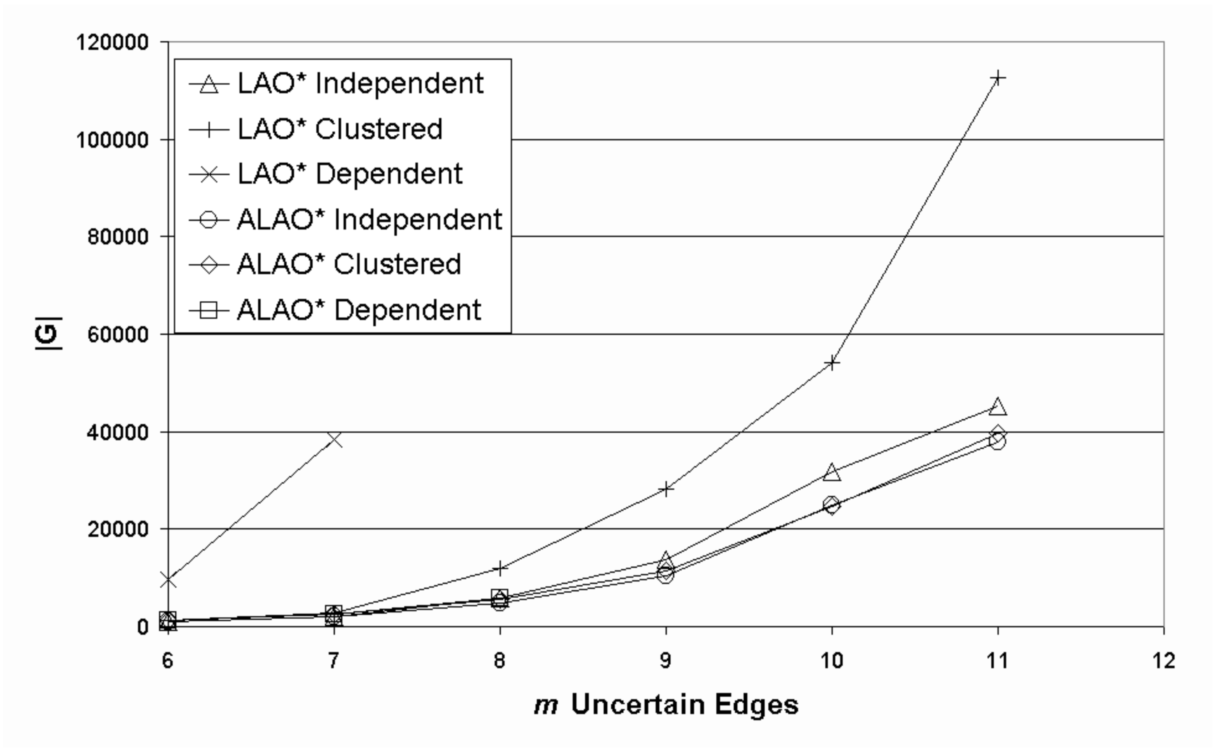


Figure 5.14: Comparison of the explicit graph size for Graph 19 as the number of uncertain edges increases under LAO* and ALAO*.

However, this is not sufficient in ALAO* because of the approximation in belief state equality: two nodes with different belief states and therefore different hashes can still be considered equal. For a hashtable to function correctly, objects considered equal MUST have the same hash key to guarantee finding them again. When a new candidate LAO* node is generated the algorithm must check to see if one that is equal already exists by querying the hashtable. Under approximate equality, if the new candidate node has a different hash key to an existing, approximately equal node, it would not be able to find it in the table, instead creating a new node where the existing one should have been used. For this reason, the hash function under ALAO* only considers the PRM node label when hashing, which restores the contract to ensure the correct hashtable function, but with the side effect of placing all ALAO* nodes with the same PRM node label into the same hash bucket. Unfortunately, this degrades the near constant time complexity of hashtable lookups to a linear search of the hash bucket to check through every ALAO* node with the same PRM node number. This inefficiency is compounded by the necessity to perform a more expensive K-L distance based equality test on each node in the particular chain.

When many nodes are present in each bucket, as is the case when G is large, the overhead of managing the hashtable dominates the algorithm runtime. As an example from profiling data collected on a single run of the clustered model when $m = 10$, hashtable in-

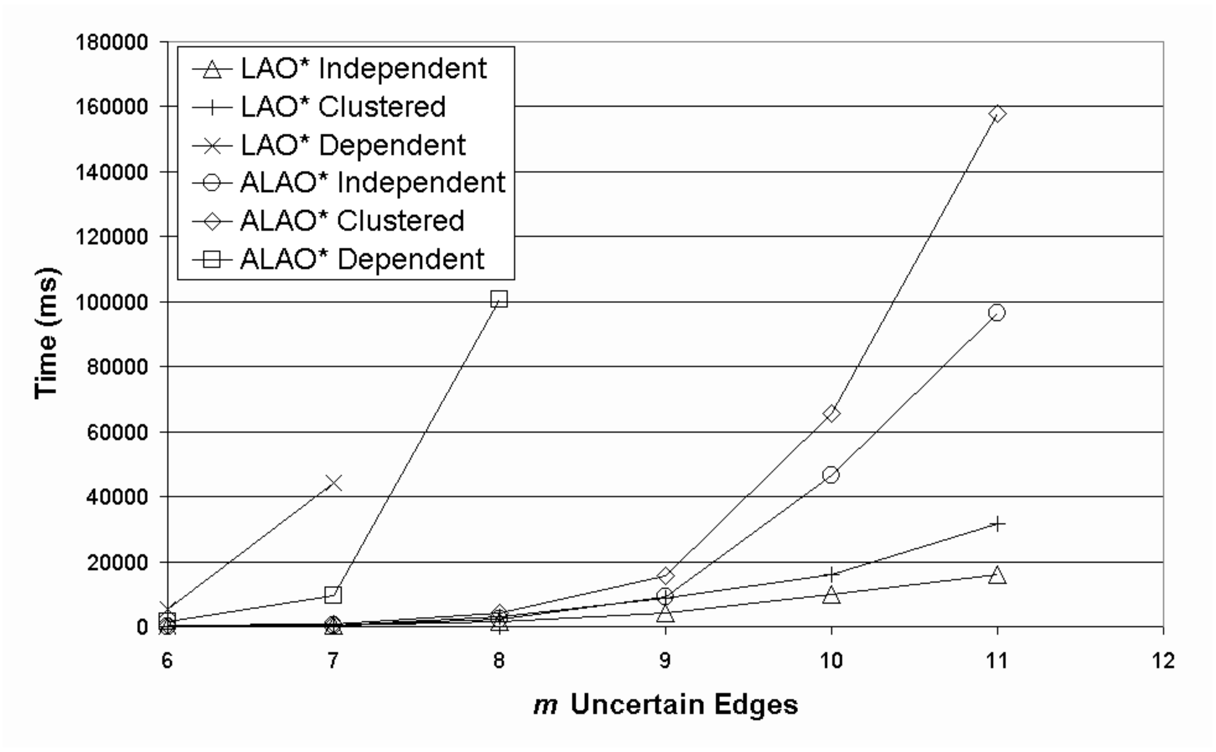


Figure 5.15: Comparison of the solution times for Graph 19 as the number of uncertain edges increases under LAO* and ALAO*.

sertions/lookups accounted for 0.7% of the runtime under LAO* and 84.3% under ALAO*. Full profiling data can be found in Appendix D. Other hash functions were considered, but none could guarantee that all approximately equal nodes would deterministically be assigned to the same hash key. The correct answer is probably to replace the hashtable entirely with a more suitable data structure; unfortunately, I had not had time to investigate this thoroughly. Choosing a more suitable lookup strategy for existing nodes should allow ALAO* to benefit from the reduced explicit graph size in larger graphs.

Considering the sizes of the explicit graph sizes in Figure 5.14, ALAO* scales better than LAO* under all models as the number of edges increases. The rate at which G grows in ALAO* is lower, particularly for the clustered and dependent models, whereas the explicit graphs for the independent model grow at similar rates in both LAO* and ALAO*. The reduced number of states explored means that ALAO* has the potential to explore larger graphs and successfully generate policies for them.

5.4 Concluding remarks

The results from this research show that the current techniques are limited in the number of uncertain graph edges that can be realistically handled. While some of the computation time issues with ALAO* could be solved with a better choice of data structure, this is not the fundamental limitation of the algorithm. The memory required to explore the MDP state space is the limiting factor of this design, the consequences of which can be seen throughout the results. The complexity and size of the explicit graph created by (A)LAO* is always far larger than the original PRM graph (except in trivial cases) and is dependent upon the number of uncertain edges. Graphs with more uncertain edges require more memory when searching for a policy. The cause is actually two-fold: as graphs become more complex, more nodes have to be explored to reach the goal and find the optimal policy, but the dimensionality of the belief space also increases, therefore each node requires more memory. Algorithms which plan directly in the space of the PRM graph such as FSSN (see Section 3.2.1) and MCC (see Section 3.2.3) do not suffer from this limitation and only incur the memory footprint required by the algorithm, rather than a conversion to a new domain. The slower growth of G under the clustered model helps mitigate the effect of the increasing dimensionality and approximation reduces the size of the explored state space. However, neither can completely counteract the fact that each additional uncertain edge and observation increases the state space that must be explored.

5.4.1 Summary

A summary of the results from this Chapter:

Weighting Adding a weight to the heuristic can reduce the amount of graph that LAO* explores by making the policy search more greedy. It generally reduces the policy quality substantially in many graphs, so is not recommended. Graphs that were previously unsolvable are often still unsolvable with weighting applied. When a large weight is applied, a policy may be produced, but it is likely to be of poor quality.

Approximation Adding a small approximation has been shown to have massive benefits on computation time, without negatively affecting the policy quality. The approximation threshold $\max_{\mathrm{KL}}$, should be kept small to ensure enough key belief points can still be identified. In general, even small thresholds can have a large impact on computation time. This is particularly noticeable when used the dependent MDP model. Increasing $\max_{\mathrm{KL}}$ too far should be avoided as it can cause many problems in policy quality such as failure to reach the goal and increased route costs. In our experiments, we found that $\max_{\mathrm{KL}}$ should be set to less than 0.3 in nearly all cases to retain a good policy quality.

Unsafe graphs These can interact badly with several aspects of the algorithms and are generally harder to solve. We often saw LAO* fail to produce a policy on unsafe graphs due to the greatly increased number of states it tried to explore. We also saw ALAO* produce worse policies on unsafe graphs unless a very small threshold was used. A long, but expensive route to the goal in a graph means that the agent can always reach the goal and removes the unsafe aspect of the graph. If a graph is unsafe, it can negate the advantages of clustering and approximation.

Chapter 6

Conclusion

6.1 Discussion

The thesis aims given in the introduction are restated below:

1. Show how path planning in a partially known configuration space can be represented as a decision-theoretic planning problem.
2. Investigate how optimal plans can be generated for the domain in a way that is scalable to real world problems.

The aim of this research was to examine the problem of robot navigation under uncertainty and how path plans can be generated offline that directly incorporate this uncertainty. Given a start and goal within a partially known environment, the agent should compute a plan with the lowest expected cost of reaching the goal. Probabilistic roadmaps were used as the vehicle for the basic planning mechanism. PRMs represent the possible paths around an environment as a weighted graph and are easy to manipulate and convert to a number of different applications. Some of these were examined in Chapter 3. Using standard shortest path algorithms on a PRM allows the generation of shortest routes across the environment, but suffers from the drawback of assuming perfect knowledge of the obstacles. Removing this constraint makes planning significantly harder and generating near-optimal plans in a reasonable time forms the central aim of this thesis. We have abstracted the route finding problem to a planning problem with the objective of balancing the risk between long, safe paths and shorter but uncertain paths. Treating the PRM roadmap as a graph search problem separates the planning from the spatial domain so that the physical dimensionality of the space has a reduced impact on the complexity of the plan space. To provide sufficient coverage of higher dimension configuration spaces, more nodes in a PRM graph are required, which increases the complexity of the plan space, so the two issues are not totally separable. Given that not all the paths in the generated PRM graph are usable when uncertainty is taken into consideration, the agent cannot make any assumptions about any uncertain edges. The presence of this uncertainty about its environment, combined with the ability to reduce the uncertainty through noisy observations, means that this domain fits closely with the POMDP framework. We have demonstrated how this can be formulated as a POMDP (see Section 4.2) that gives us the optimal solution for all uncertain PRM graphs, but as has been shown in this thesis and in the large body of literature available on the subject, solving large POMDPs is a hard, often intractable problem. One of the contributions of this thesis is in demonstrating how the POMDP formulation of the domain could be converted into an MDP. We convert the POMDP to a belief state MDP and also exploit the inherent structure in the belief space. With a focus on the first aim of the thesis, reformulating the robot's environment as a belief state MDP by exploiting the graph structure of PRM allowed great flexibility in how we could approach the problem. Factoring the agent's location out of the belief state in the MDP state space reduces the number of elements in the probability distribution by a factor of $|V|$. This gives us the dependent model MDP. Exploiting more structure present in the problem, the state space is reduced further through the use of grouping related uncertain edges into clusters. This is an approximation technique which applies an approximation to the model that retains the plan quality of the dependent model, while sacrificing little in representational completeness, and is one of the primary contributions of the thesis. The clustering of edges exploits the independence between unrelated edges to increase the scalability of the approach, so also contributes towards the second aim.

In Chapter 5 two techniques were investigated that had the potential to extend the scalability of the algorithm. While heuristic weighting contributed little to attaining this, the approximate extension to LAO* that we developed decreased the computation time required for plan generation considerably (see Section 5.2.2.4 on page 177). Unfortunately, on graphs with a greater number of uncertain edges, the inefficiencies in the data structure used in the implementation of ALAO* negate these gains. This can be seen in Figure 5.15. Approximation via ALAO* differs from approximation via edge clustering by applying approximation to a different part of the process. Edge clustering approximates the model—the $\mathcal{S}$ component of the MDP model is modified to reduce the state space. In contrast, ALAO* applies approximation in the solution algorithm; the MDP model remains unchanged between LAO* and ALAO*. It provides a method for exploring a large state space with greater expedience than an exact solver. The RTDP-Bel (Bonet and Geffner 2009) algorithm applies a similar approach by adapting the RTDP algorithm (see Section 3.3.1.1) to POMDPs, but relies solely on discretisation to map similar belief states to one MDP state. Although we use discretisation in the ALAO* extension because it is present in the MDP models, it is not a requirement for ALAO*. Combined with a clustered model of the MDP state space, this forms the main contribution towards the second aim of this research. The graph pre-processor, while not a central component of the research, also assists in improving the scalability. It is certainly possible to apply pre-processing to other domains to reduce or remove irrelevant sections of the plan space; however, the graph pre-processor was developed through observations about this specific problem class. There is little intuition at the moment about how effective similar pre-processing techniques would be in other domains. In other navigation problems, a pre-processor could search for and short circuit chains of deterministic state transitions into one transition. In any domain, if areas of the state space where no useful observations are obtained can be identified, these states could be amalgamated into one. The focus of this type of pre-processor is to combine groups of states which must be visited en route to the goal, even if they do not affect the eventual policy.

One of the most useful results of the research is that the conversion of the POMDP model into an MDP provides many opportunities to improve the speed of policy computation through various approximations. The MDPs we create can have fairly long search depths and typically have a low branching factor, since most action choices will be invalid at most nodes and edge observations are not available at every node. The first property depends on the PRM graph, but complex graphs with many nodes mean the agent has to traverse many edges to reach the goal. Graph pre-processing can consolidate many of these however, so the effect is reduced.

During the progress of the research, we noticed that less certain observations in a PRM graph tended to be taken repeatedly, i.e. the agent would "hop" between an observational node and the nearest non-observational node. The root cause was that the agent was trying to acquire more information (reduce uncertainty) to choose between possible routes, because the probabilistic observations were not changing the belief state enough to convince the agent to commit to a route. The hopping behaviour is then the cheapest way to gain more information because our domain does not provide a "standstill and observe" action. In hindsight, providing such an action would have been beneficial, as it would have allowed repeated observations without a movement cost (essentially free since we assume no observation costs). While this has close parallels with the work of Censi et al. (2008) described in Section 3.2.2, it may also be unrealistic for the same reasons: standing still does not necessarily mean uncertainty will decrease. The consequence of these repeated observations is that many similar belief states are created in quick succession at the same PRM node(s). It is this attribute that motivated the initial idea and research into the creation of an approximation belief update which was developed into ALAO*.

The work in this thesis also emphasised how much harder planning becomes in the presence of even small amounts of uncertainty. Many pitfalls were discovered through the course of research which drove the progress towards heuristic graph search algorithms such as LAO* and the various approximations described. While the discretisation of the POMDP belief space was necessary to create a finite MDP state space, it caused many minor, unforeseen issues in the implementation. A prime example of this is explained in Section 4.3.2.1, but other more subtle numerical instabilities resulting from discretisation had to be addressed during the work. The difficulty in handling uncertainty also highlights scalability as a weaker area of the thesis. While the scalability is assisted by a variety of techniques, the huge state spaces created still cause difficulties for policy generation. We can identify several promising approaches to aid scalability. One possibility is to use a hierarchical method where, over a large environment, a high-level general motion graph is created with a few uncertain edges which can then be passed to lower level planners. The lower level planner would then generate the actual PRM motion graph the agent would follow between stages of the high-level graph with more accurate uncertain edges. This would keep the number of uncertain edges at each planning level manageable while extending the size of the environment that could be handled. Another similar approach would be to use a heuristic planner that could identify suitable checkpoints between the start and goal locations and treat each successive pair of checkpoints as a separate path planning problem. The best route between each checkpoint pair would be solved as a new uncertain PRM graph. A third option is to improve the pre-processing stage. Before nodes are eliminated, an "edge merging" phase could be introduced which would merge adjacent or close uncertain edges into one edge if it was determined that the correlation between their statuses was high and both would be needed to traverse that section of the graph. An example is shown in Figure 6.1.

We initially expected edge clustering to yield a gain over the independent model more often. We found that the optimal policy does not exploit edge dependencies in more than about 30% of graphs tested, despite edge dependencies existing in all the graphs created. Does this imply that clustering is not worthwhile? We do not believe that is the case because the clustered models do not require much more time to solve than the independent model and the gains in expected cost can be significant.

The abstraction away from the spatial representation provided us with a well defined domain to solve, but separates us somewhat from the original motivation. This is a consequence of choosing to focus on the planning and uncertainty aspects of the problem. Some areas are left open. Particularly, we leave open the question of how uncertain edges should be selected in the first place and how observations should be placed. The obvious way to place uncertain edges is through repeated sampling of obstacle positions and testing

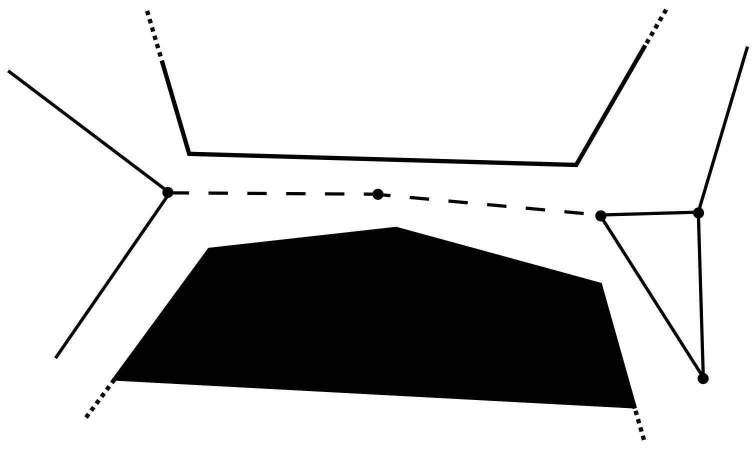


Figure 6.1: An example of two uncertain edges with highly correlated statuses, shown as dashed lines. Since both edges would need to be traversed to pass between the obstacles (shaded), these would be a prime candidate for edge merging.

which PRM edges are intersected by obstacles. This has been tested to a limited degree, but consistently produced too many uncertain edges for the planner to solve. Deciding cluster configurations could be achieved by analysing the starting dependent belief state for correlations between uncertain edges. As worlds are sampled according to this belief state, correlations between uncertain edge statuses would become apparent, indicating which edges would be likely to need to be placed in the same cluster. Appendix C shows a possible algorithm for assigning edges into clusters based on an initial dependent belief state.

One benefit of the abstraction is that it allows the application of the models presented to other problems. While closely resembling the Canadian traveller problem (CTP, described in Section 3.2.1), it is not limited to two dimensional domains, since PRM edges may move through many dimensions in the C-space. This work is also applicable to other problems where a system may have two or more options for reaching a particular state with varying resource costs, risks and the possibility of failure. As well as CTP, consider satellite navigation systems; routes can be blocked or jammed and traffic broadcasts become analogous to "ahead of time" observations of those routes. In many traffic systems, if one main road becomes blocked, a second alternative often becomes the preferred choice of route for drivers, thus the traffic density and usability of those two roads become correlated. This work could help construct plans for such domains to improve traffic flow by taking these correlations into account.

6.2 Future work

There are several directions for future work for this research, some of which have already been mentioned. One addition to the models we have proposed here would be a "stop" action for the agent. While the MDPs we have defined are ostensibly stochastic shortest path problems, it should be noted that in unsafe graphs, where all the routes to the goal require the traversal of an uncertain edge, the agent can reach states where no sequence of actions ever reaches a goal. This can cause problems for goal based solvers such as LAO* and other heuristic solvers because the true cost of those states is infinity. In (A)LAO*, the optimal policy tells the agent to hop between the two closest nodes in the PRM graph when it cannot reach the goal because this incurs the lowest eventual cost. Adding a stop action with a high cost that deterministically transitions to a "failed" accepting state would convert the MDP into a true shortest path problem. The cost of such an action would have to be high enough that the agent would only ever choose it if all routes to the actual goal were found to be blocked. The optimal policy would then be to simply stop when the agent realises it is trapped.

Another interesting area for future work is how more recent MDP solvers may be applied to our domain. The recent work on improved value iteration is of particular interest. Asynchronous dynamic programming (DP) relies on restricting value iteration (VI) to relevant areas of the state space to speed up convergence; however, asynchronous DP algorithms rarely consider how the order of state backups affects the solution time. Since different states typically require varying numbers of backups for their value to converge, there must be an optimal order for performing these backups. That is, when backups are applied to selected states in this order, the value function will have converged to become ϵ-optimal with the lowest number of backups. Topological Value Iteration (TVI) (Dai and Goldsmith 2007) and the more recent Focused TVI (FTVI) (Dai et al. 2009) use the connectivity of the state space to identify which states should be backed up first. In stochastic shortest path MDPs, this usually takes the form of backing up states away from the goal towards the start. The version of LAO* we use performs VI in depth first post-traversal order on nodes in the BSG. Nodes that are closer to the goal are generally backed up first, but no further analysis is performed. FTVI and TVI partition states in the state space into mutually exclusive, strongly connected subsets, and then perform VI on each group separately in topological order. FTVI improves on TVI by performing a brief forward search from the start to eliminate non-optimal actions and reduce the size of each subset. This has been shown to outperform LAO* and RTDP by an order of magnitude in certain domains. This is relevant to our domain if we note that while the agent is moving between nodes with no observations, it can always take the reverse action. When it makes an observation and changes its belief state, this is a one way transition since the agent cannot forget what it has observed. Thus, the state space will contain several strongly connected components with observations occurring at the boundaries of the components. Applying topology analysis to our problem may further extend the size of problem that we can handle.

One of the most interesting ideas which we have not explored is the use of ALAO* as a POMDP solver. The MDP models we have developed are approximations of a specific POMDP model and ALAO* has been shown to be very effective in this domain, achieving an order of magnitude speed-up compared to a point-based POMDP solver. This result, combined with the results in Bonet and Geffner (2009), indicate that solving certain classes of POMDP as belief state MDPs can be very efficient. An advantage of ALAO* over RTDP-Bel in solving POMDPs is that discretisation of belief states is no longer a requirement, since the use of K-L divergence approximation ensures that similar belief states are treated as equal. There are two obstacles to this that would have to be addressed. Firstly, a superior data structure for ALAO* would need to be devised. As was discussed in Section 5.3.2, the hashtable data structure employed in our implementation is insufficient to handle large quantities of stored belief states efficiently. Designing a better hash function may be enough, but we believe a more efficient data structure in general is needed, though we do not currently have any clear intuition about how to accomplish this. Adapting ALAO* to use Algebraic Decision Diagrams (as in Symbolic Perseus (Poupart 2005)) would certainly be of benefit since the ability to aggregate many belief states into a single ADD would reduce the load on the data structure. However, this may not address the need to efficiently search the graph for similar belief states. The expansion routine of ALAO* needs to look for similar belief states to perform approximation testing on potential new belief states and large ADDs could still be a computational bottleneck in this process. The second issue is how to construct admissible heuristics for general POMDP domains. Though LAO* can function optimally without a heuristic, its performance can be greatly degraded (Hansen and Zilberstein 2001). The recent body of work regarding online POMDP solvers already make use of heuristics, so that would seem to provide the best source of information to solve this issue.

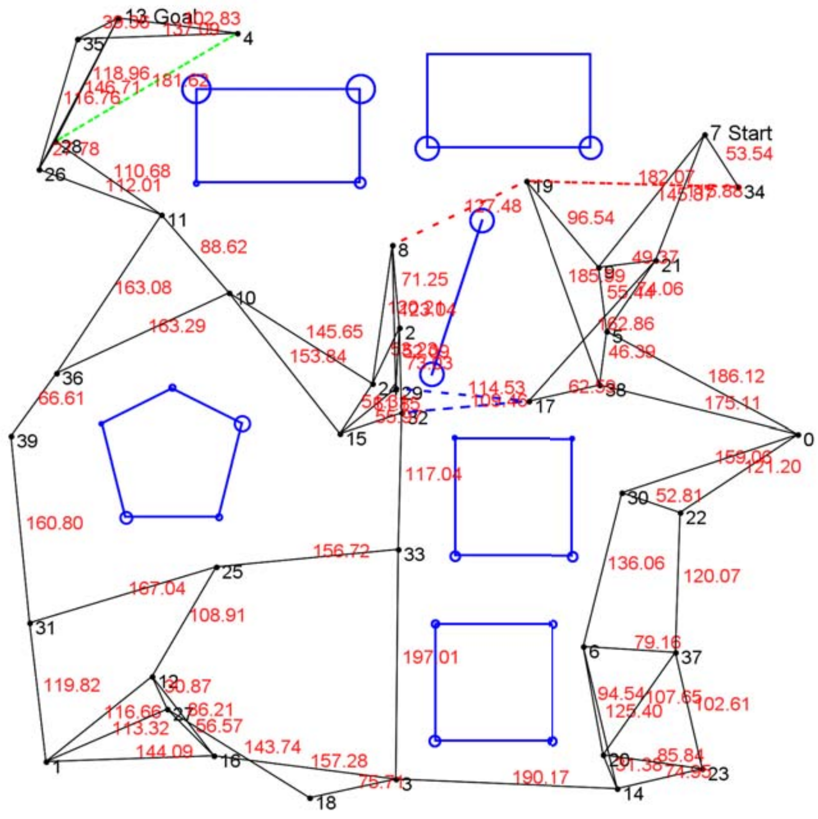
Appendix A

Graph Details

These are the details of the graphs that are used in experiments throughout the research. In all these examples the robot takes the form of a small square that has the ability to rotate on the spot. The uncertain edges are shown as dashed lines and obstacles are shown in blue. The colours of the uncertain edges indicate cluster configurations, with all uncertain edges of the same colour in the same cluster. The edge costs are shown in red. It should be noted that for the pre-processed graphs, the edge costs do not represent the Euclidean distance between the PRM nodes because an edge may represent the concatenation of several edges in the original PRM graph. Key: N=number of nodes in PRM graph, m=number of uncertain edges, I=independent MDP, C=clustered MDP, D=dependent MDP. For state space size calculation a discretisation resolution of $d = 1 \times 10^{-5}$ was assumed.

Full details of the graphs including observation functions, node positions and edge costs can be found in the text definition files which are available electronically in a compressed archive at `http://www.cs.bham.ac.uk/~mlk/graphs.zip` or via the link in Appendix B. An example is shown in Appendix B.

Graph 1



N=40, m=5

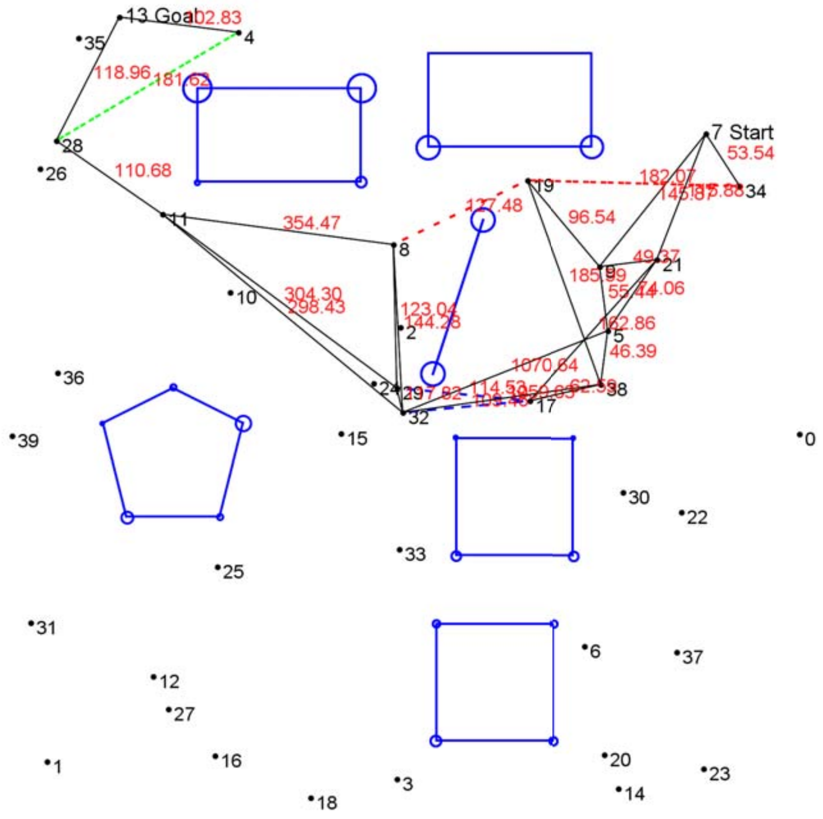
Pre-processed N=15

I $|\mathcal{S}|$=4.000E+26

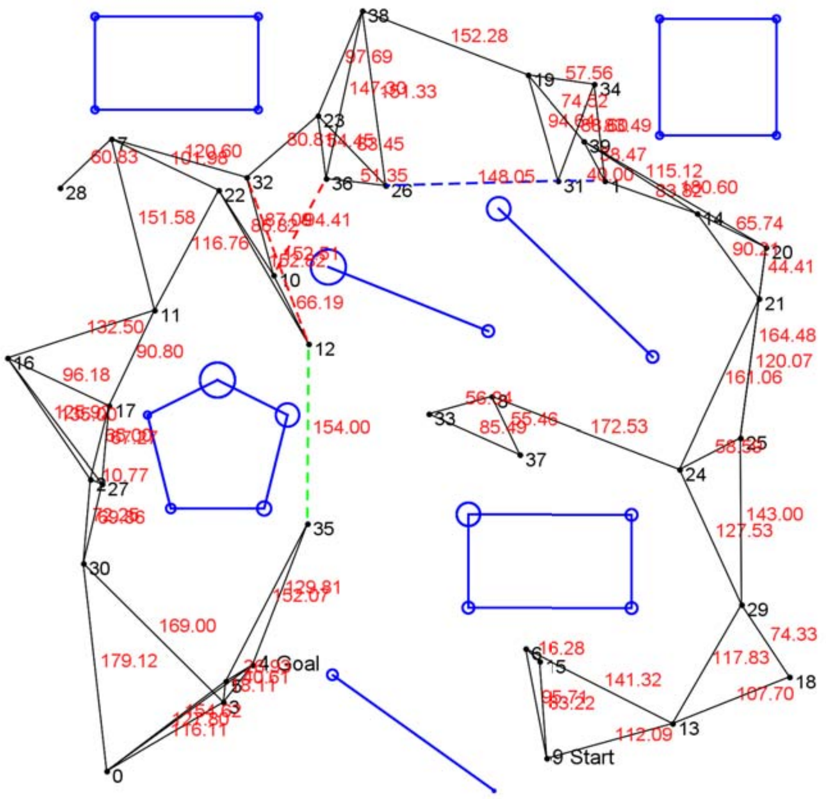
C $|\mathcal{S}|$=1.111E+35

D $|\mathcal{S}|$=4.887E+122

Pre-processed graph

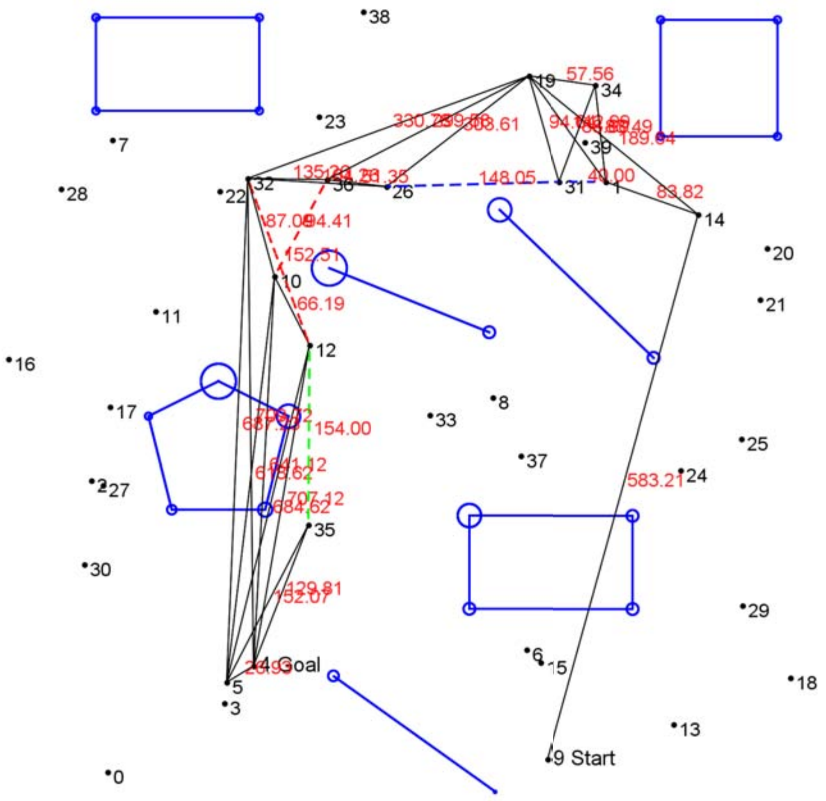


Graph 2

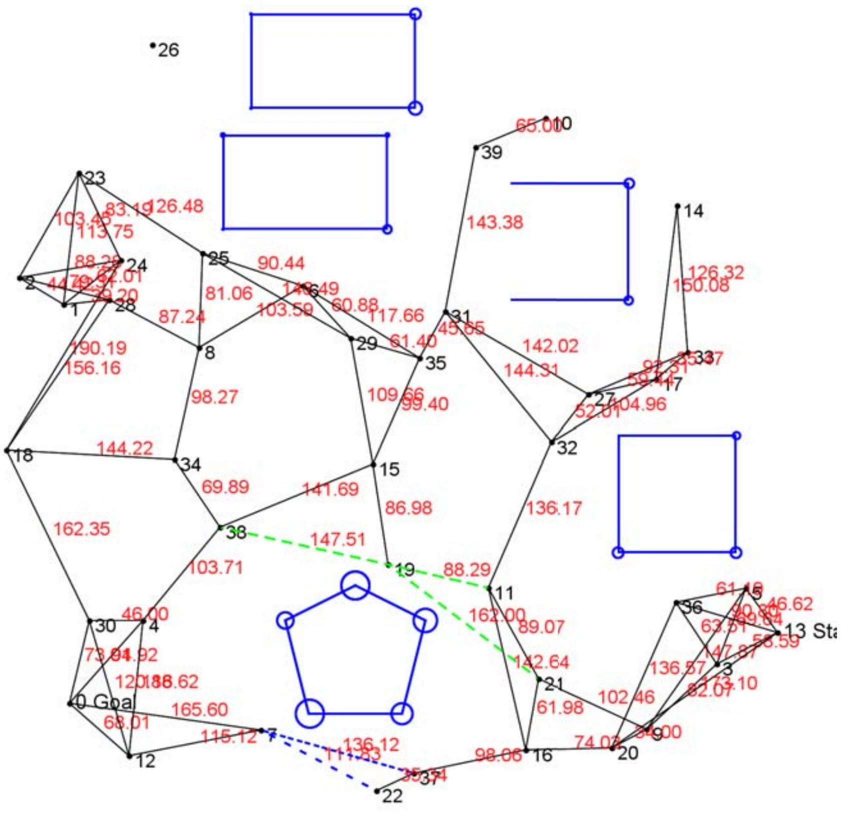


N=40, m=5
Pre-processed N=14
I $|\mathcal{S}|$=4.000E+26
C $|\mathcal{S}|$=1.111E+35
D $|\mathcal{S}|$=4.887E+122

Pre-processed graph



Graph 3



N=40, m=5

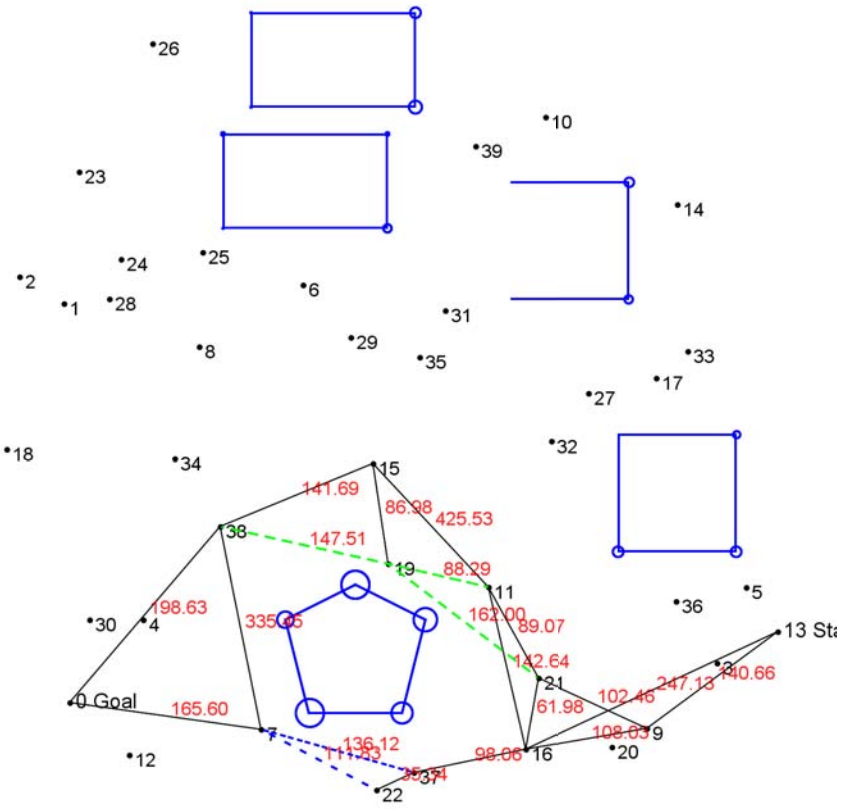
Pre-processed N=12

I $|\mathcal{S}|$=4.000E+26

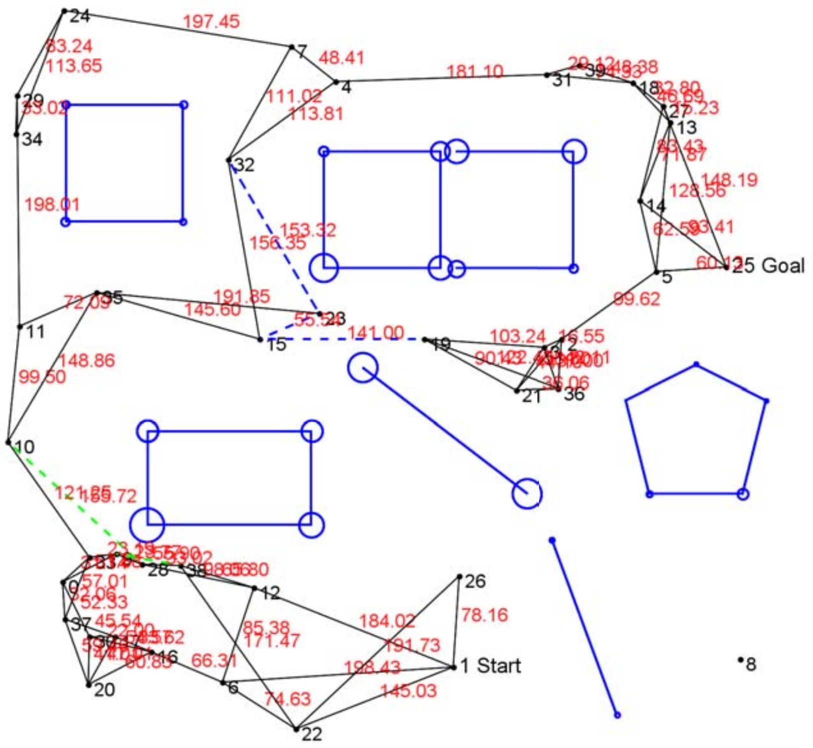
C $|\mathcal{S}|$=1.323E+47

D $|\mathcal{S}|$=4.887E+122

Pre-processed graph



Graph 4



N=40, m=5

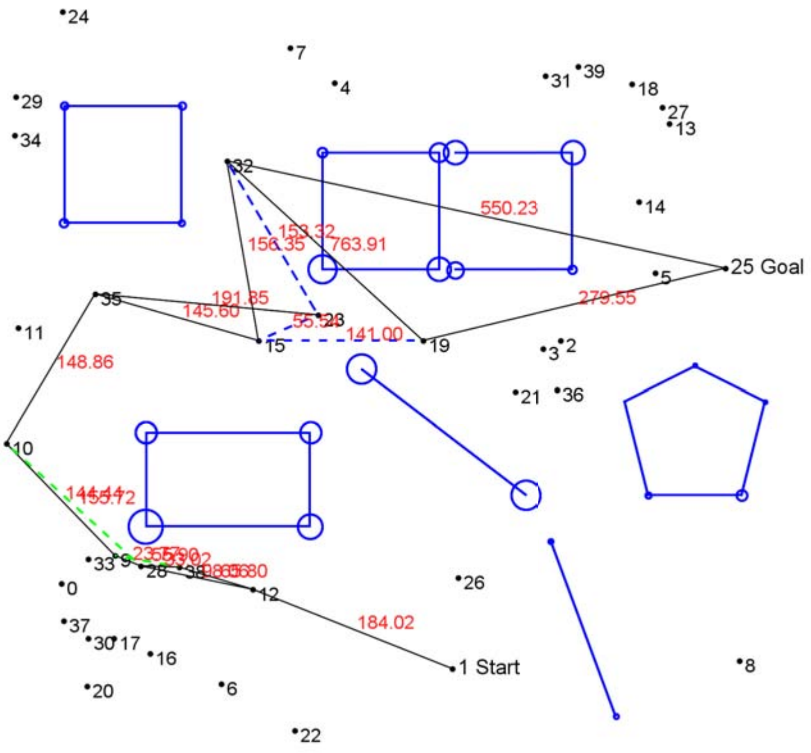
Pre-processed N=12

I $|\mathcal{S}|$=4.000E+26

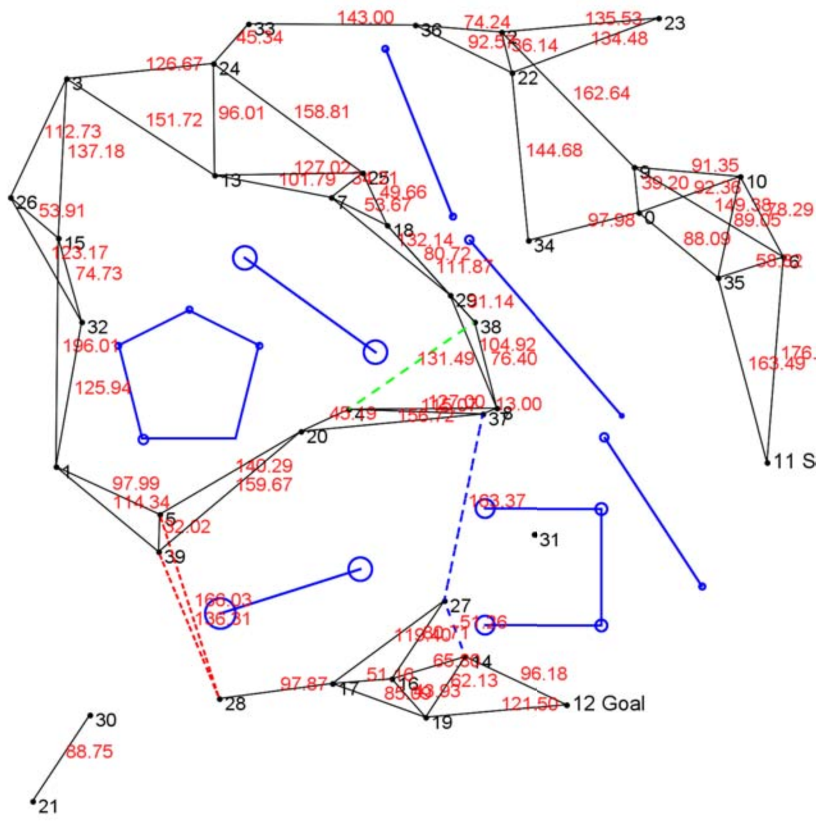
C $|\mathcal{S}|$=1.323E+47

D $|\mathcal{S}|$=4.887E+122

Pre-processed graph



Graph 5



N=40, m=5

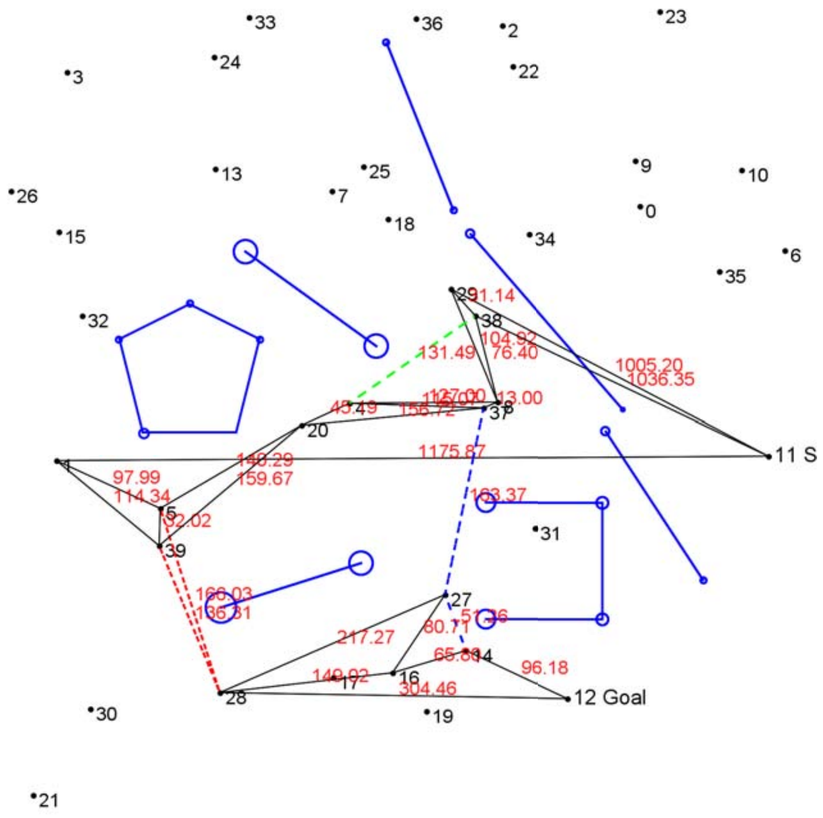
Pre-processed N=15

I $|\mathcal{S}|$=4.000E+26

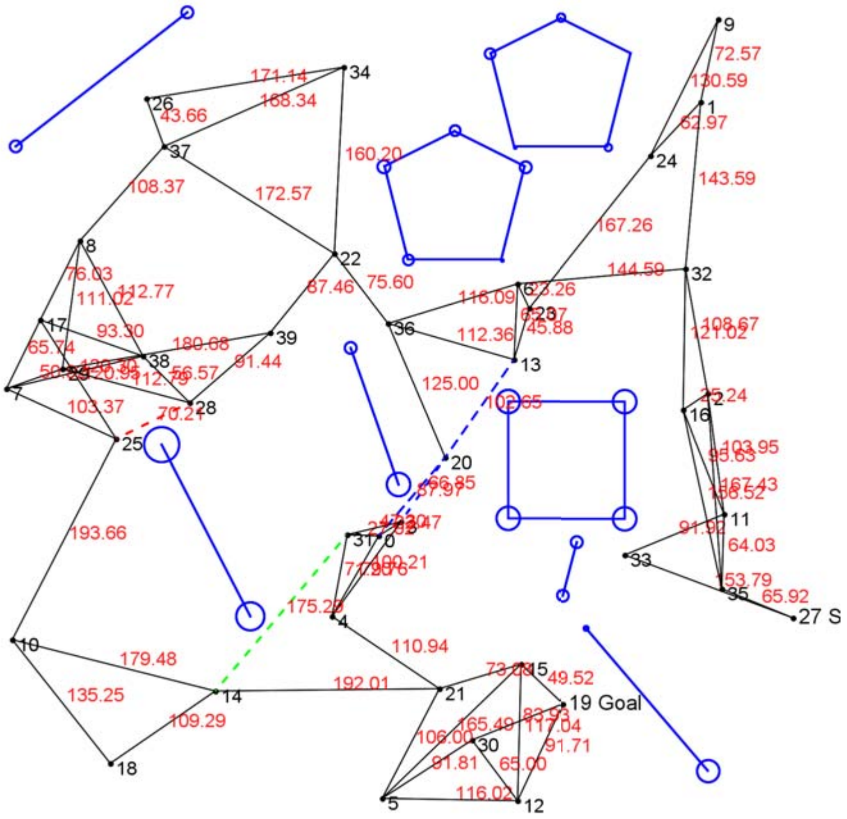
C $|\mathcal{S}|$=1.111E+35

D $|\mathcal{S}|$=4.887E+122

Pre-processed graph



Graph 6



N=40, m=5

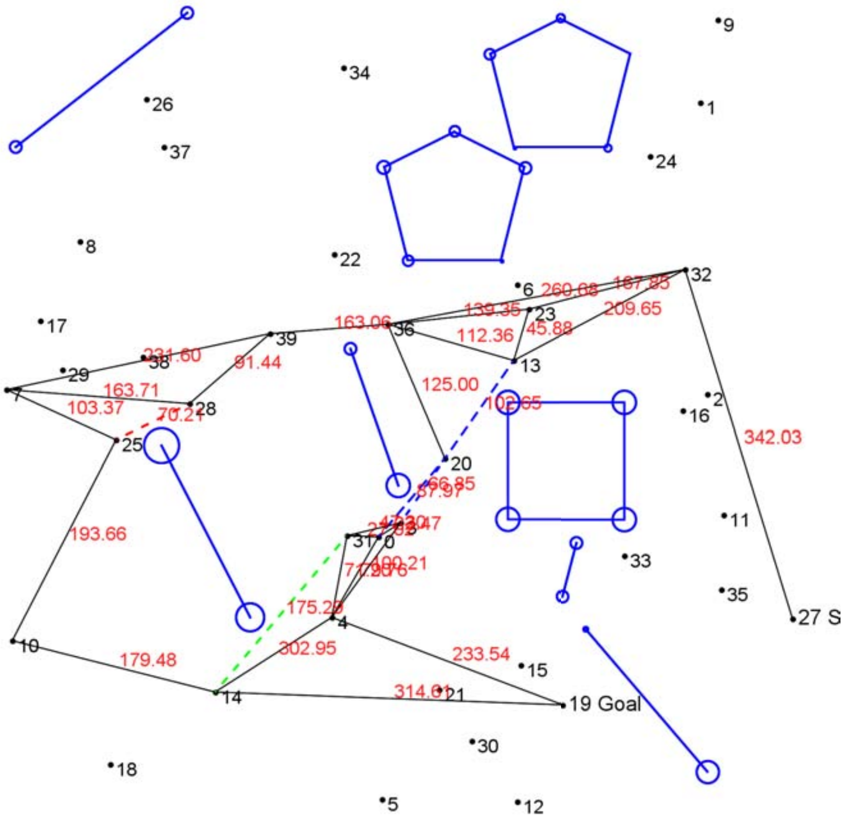
Pre-processed N=17

I $|\mathcal{S}|$=4.000E+26

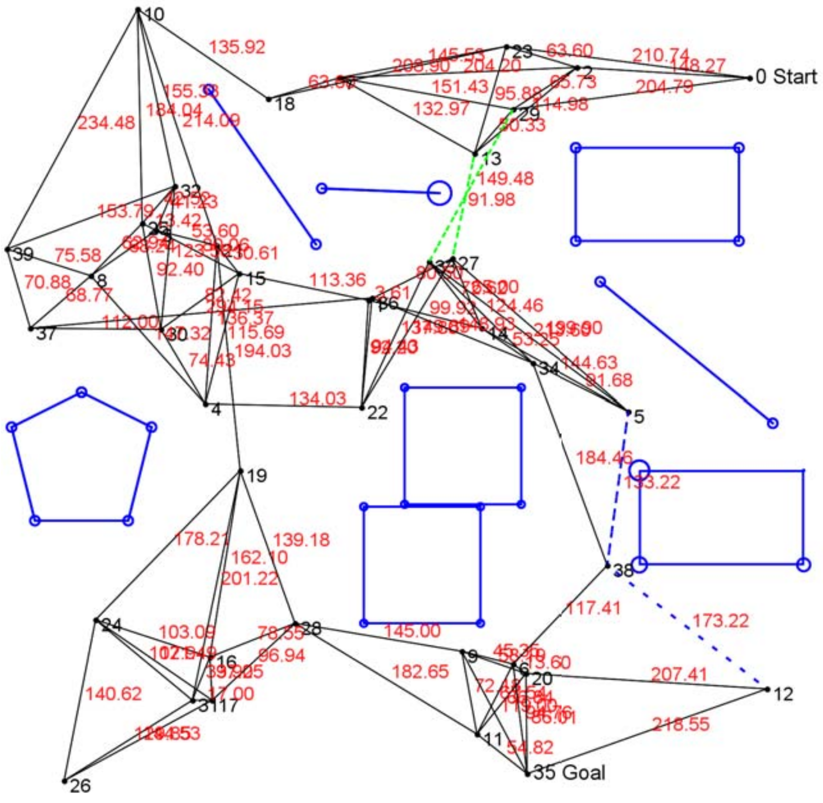
C $|\mathcal{S}|$=7.938E+42

D $|\mathcal{S}|$=4.887E+122

Pre-processed graph



Graph 7



N=40, m=4

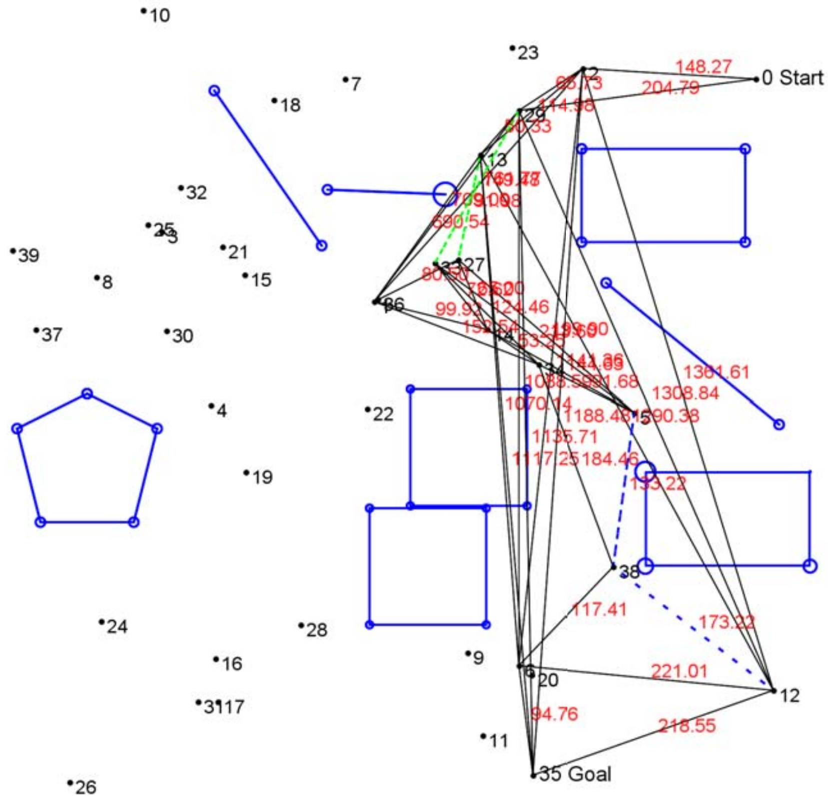
Pre-processed N=14

I $|\mathcal{S}|$=4.000E+21

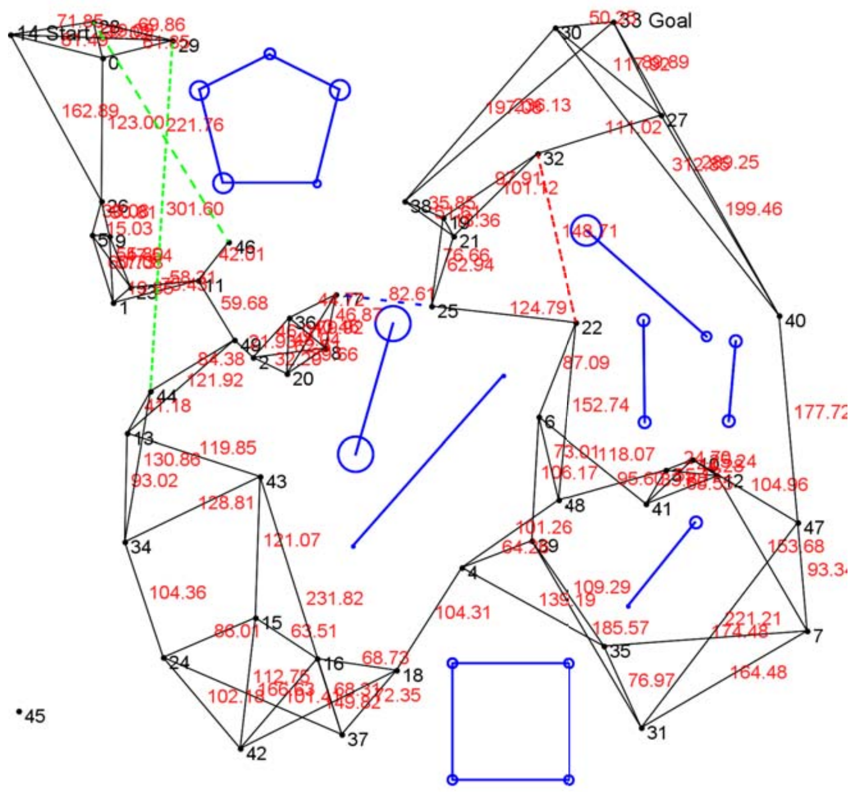
C $|\mathcal{S}|$=1.111E+30

D $|\mathcal{S}|$=3.062E+64

Pre-processed graph



Graph 8



N=50, m=4

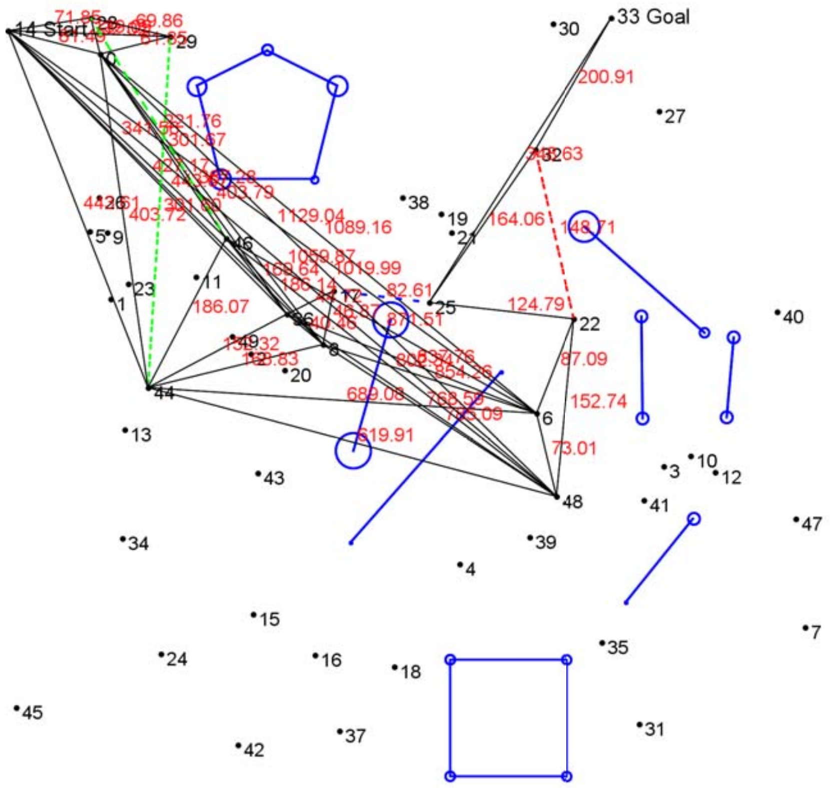
Pre-processed N=15

I $|\mathcal{S}|$=5.000E+21

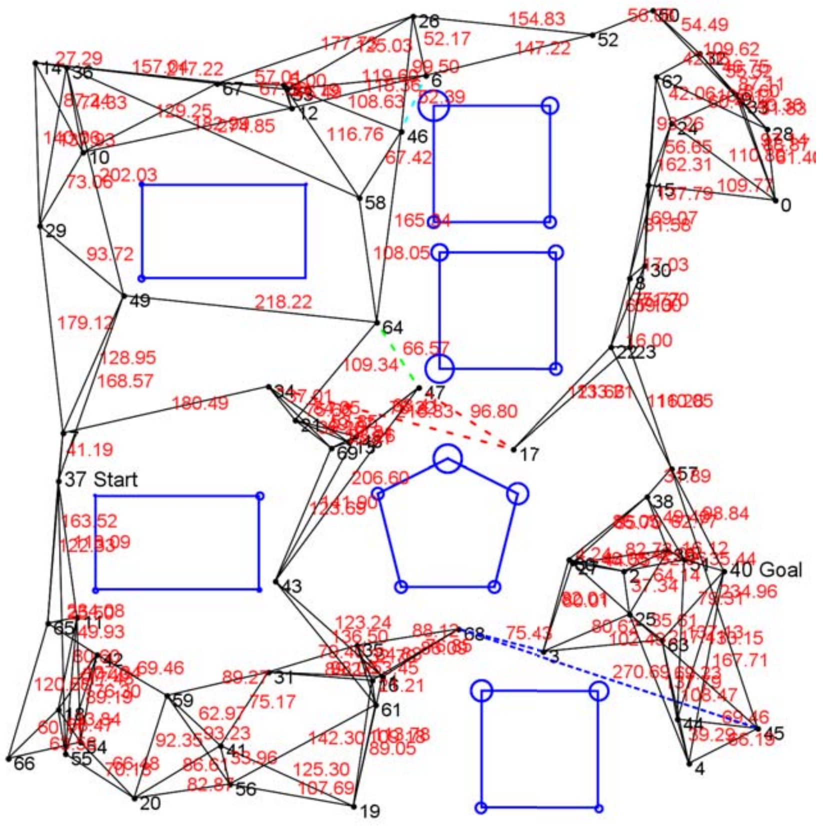
C $|\mathcal{S}|$=8.334E+25

D $|\mathcal{S}|$=3.828E+64

Pre-processed graph



Graph 9



N=70, m=6

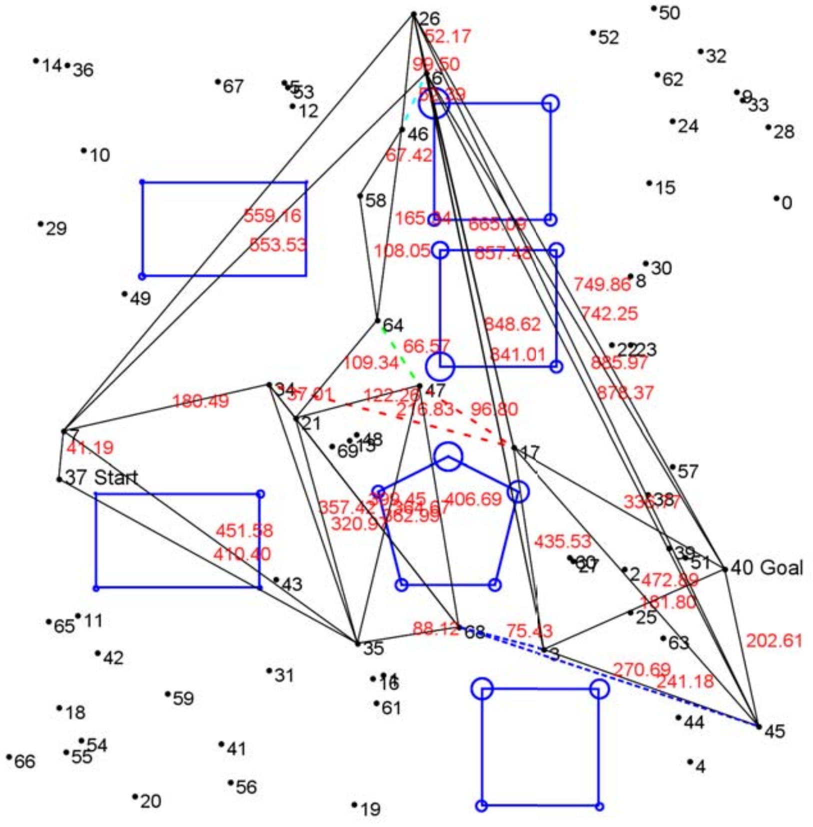
Pre-processed N=16

I $|\mathcal{S}|$=7.000E+31

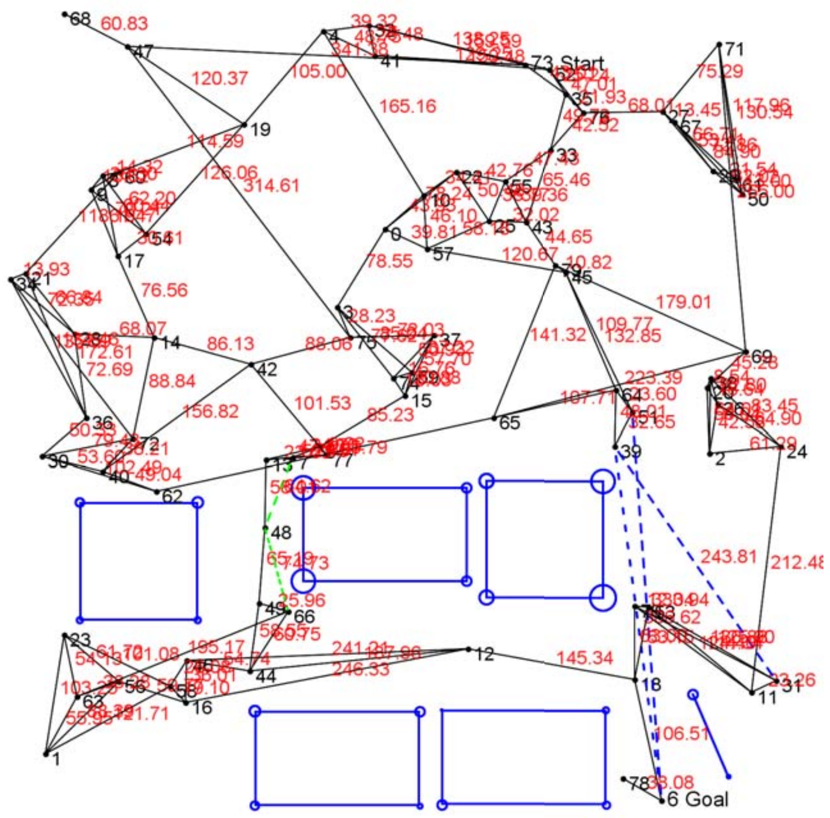
C $|\mathcal{S}|$=1.945E+40

D $|\mathcal{S}| > 1 \times 10^{307}$

Pre-processed graph



Graph 10



N=80, m=5

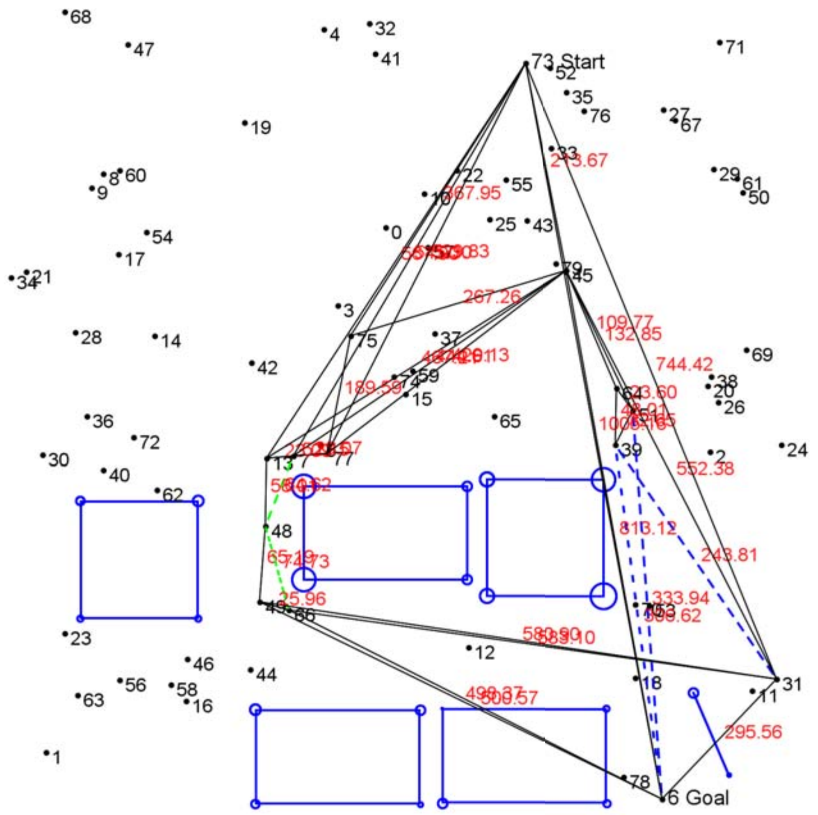
Pre-processed N=14

I $|\mathcal{S}|$=8.000E+26

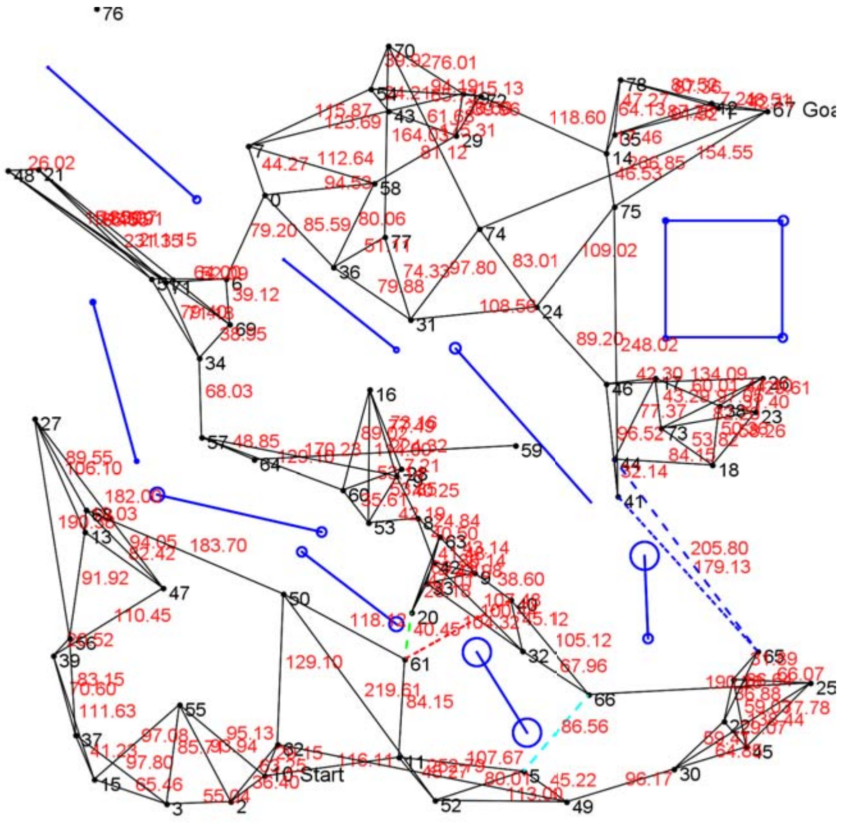
C $|\mathcal{S}|$=2.646E+47

D $|\mathcal{S}|$=9.774E+122

Pre-processed graph



Graph 11



N=80, m=5

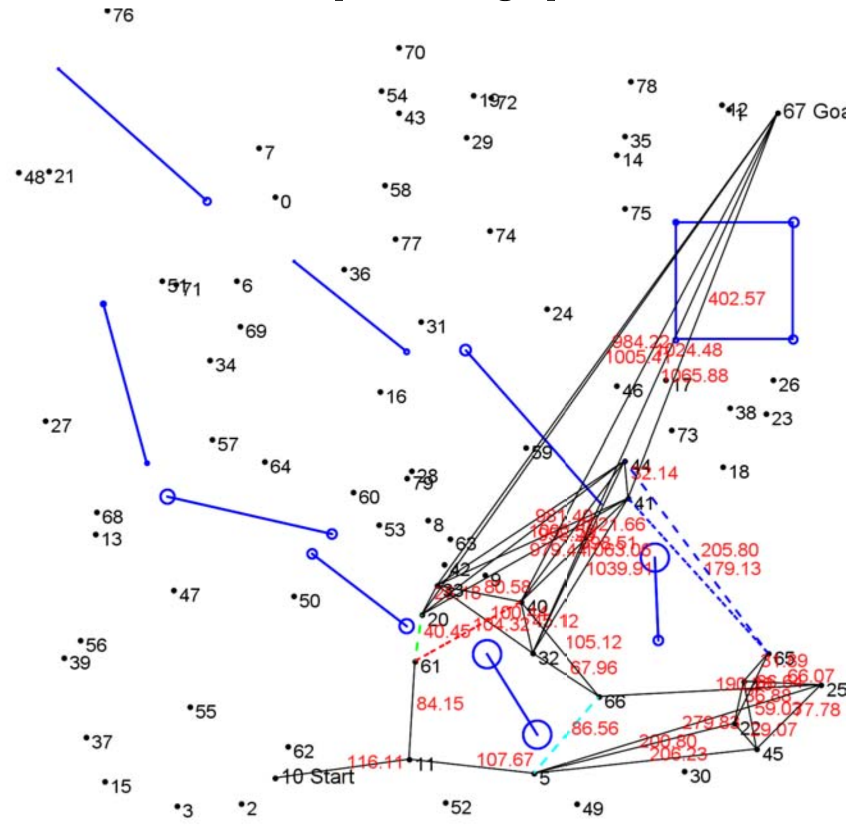
Pre-processed N=17

I $|\mathcal{S}|$=8.000E+26

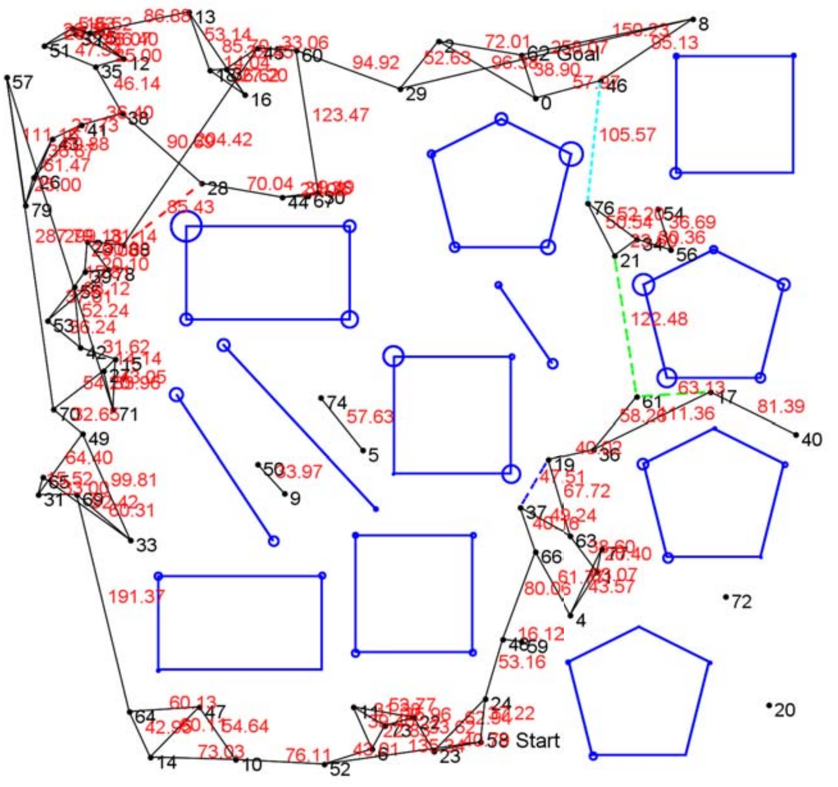
C $|\mathcal{S}|$=1.333E+31

D $|\mathcal{S}|$=9.774E+122

Pre-processed graph

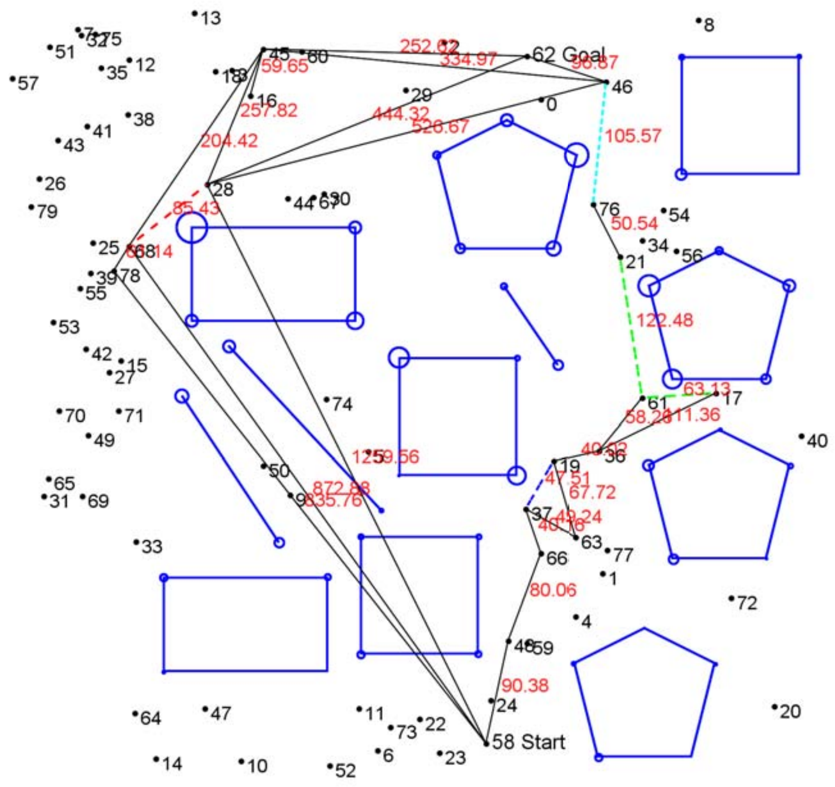


Graph 12

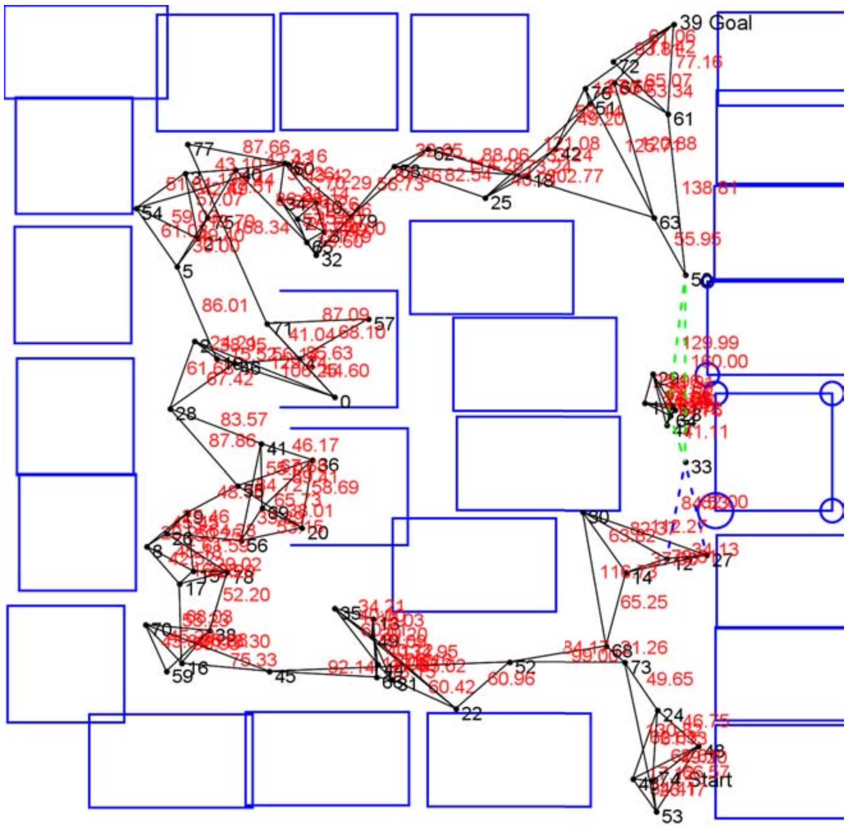


N=80, m=5
Pre-processed N=18
I $|\mathcal{S}|$=8.000E+26
C $|\mathcal{S}|$=1.333E+31
D $|\mathcal{S}|$=9.774E+122

Pre-processed graph



Graph 13



N=80, m=6

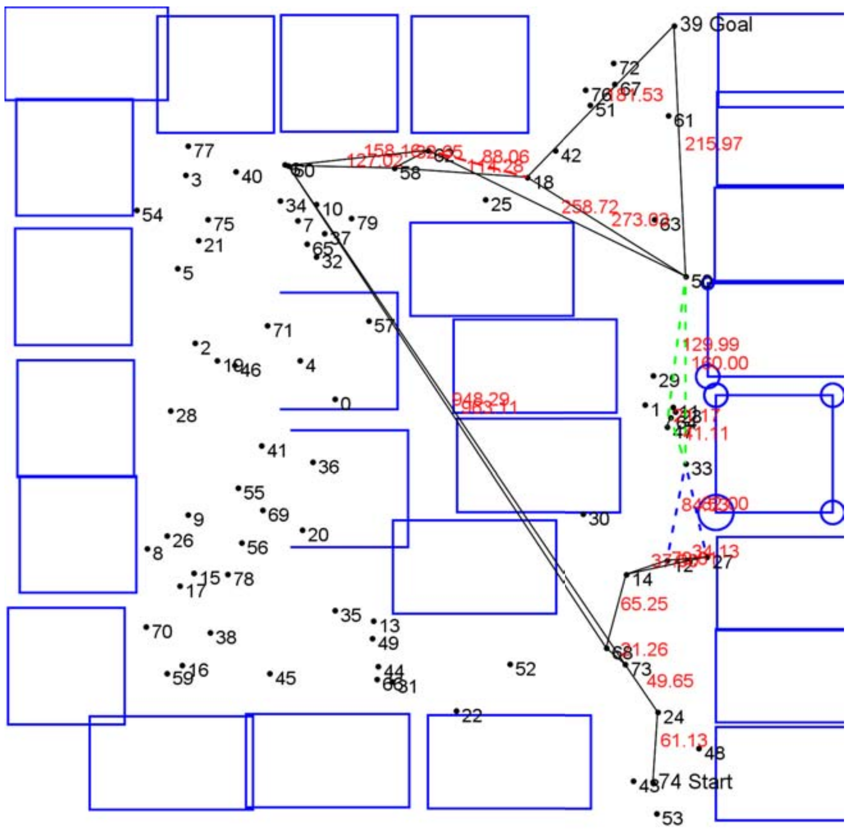
Pre-processed N=16

I $|\mathcal{S}|$=8.000E+31

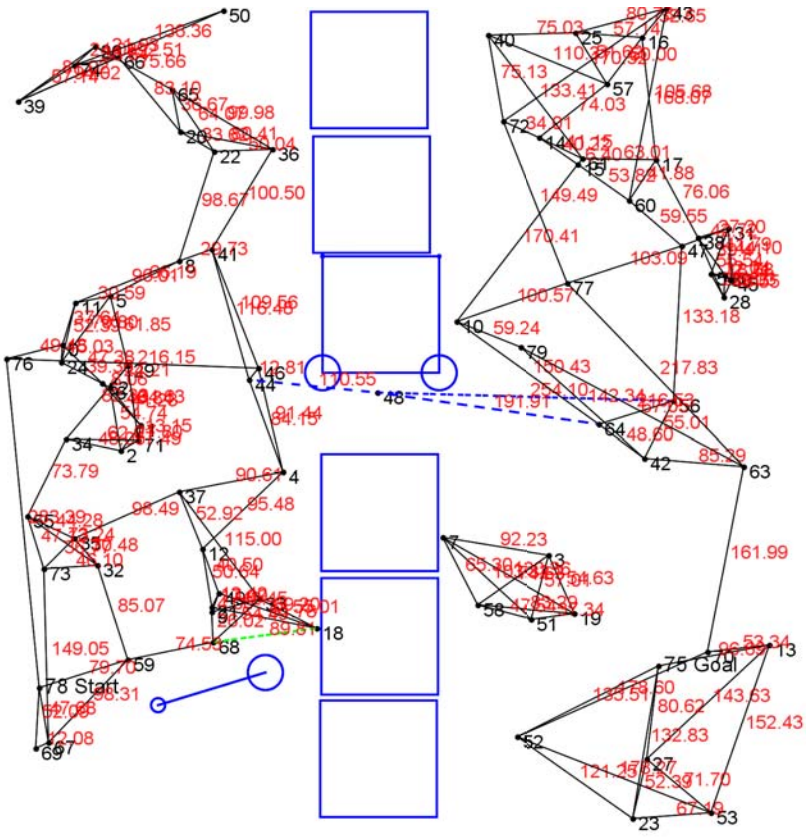
C $|\mathcal{S}|$=2.646E+52

D $|\mathcal{S}|> 1 \times 10^{307}$

Pre-processed graph



Graph 14



N=80, m=4

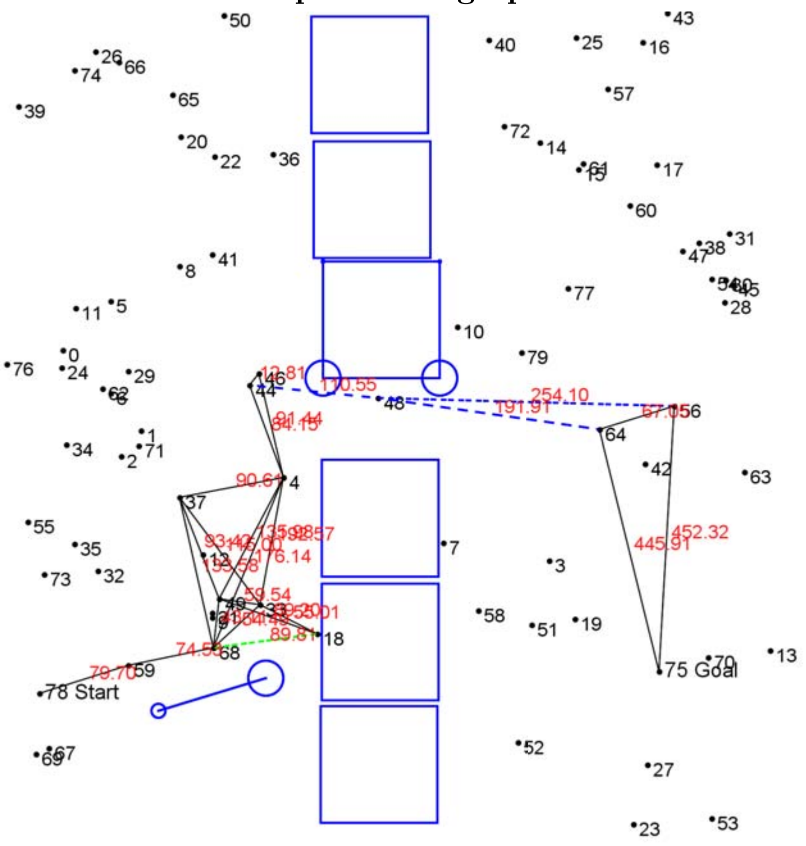
Pre-processed N=14

I $|\mathcal{S}|$=8.000E+21

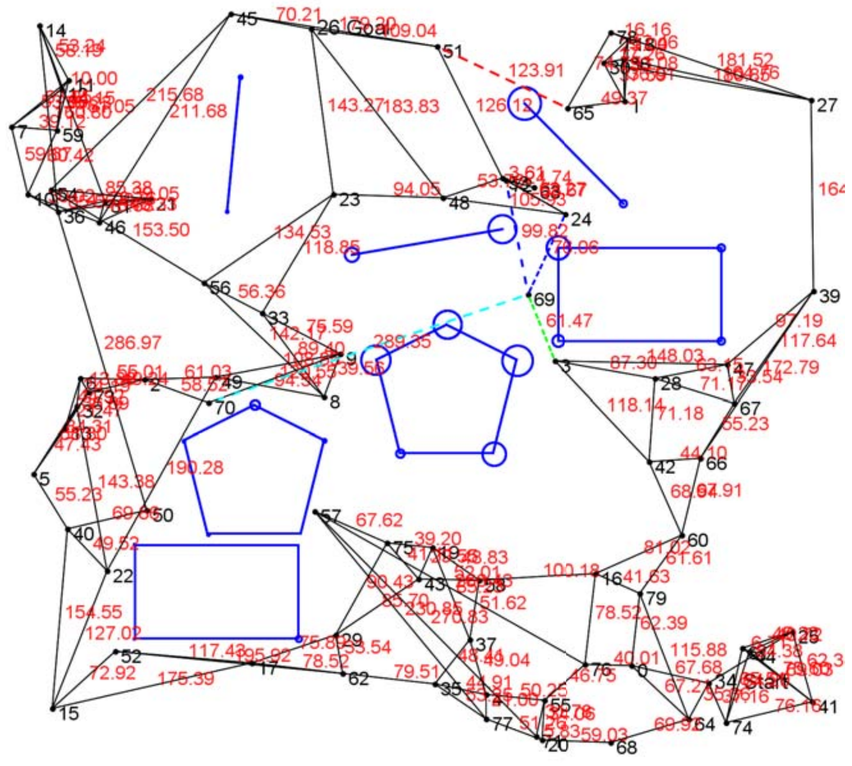
C $|\mathcal{S}|$=1.588E+38

D $|\mathcal{S}|$=6.124E+64

Pre-processed graph

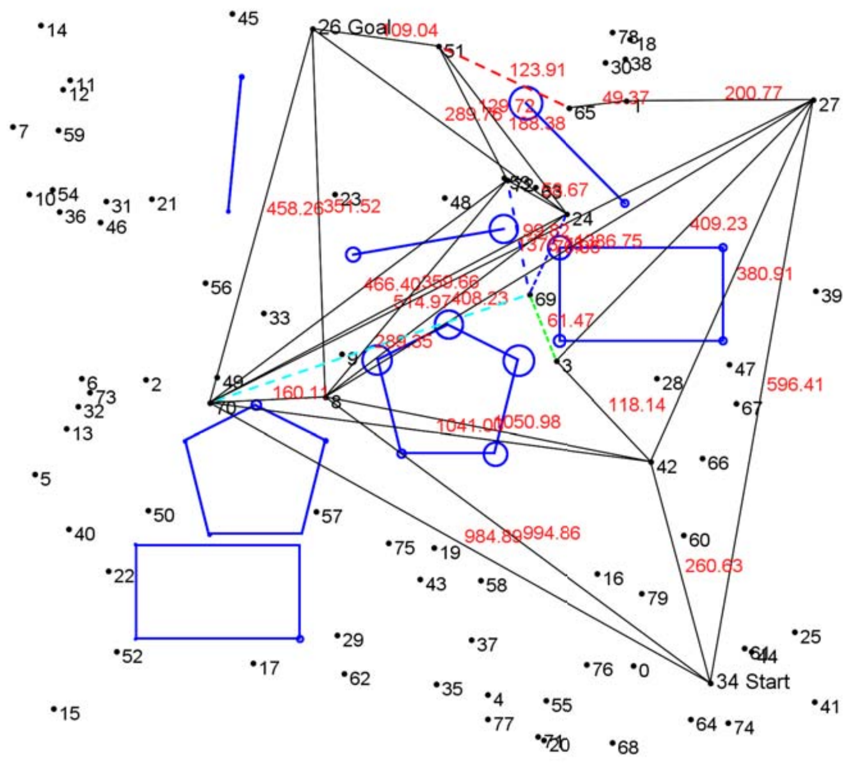


Graph 15

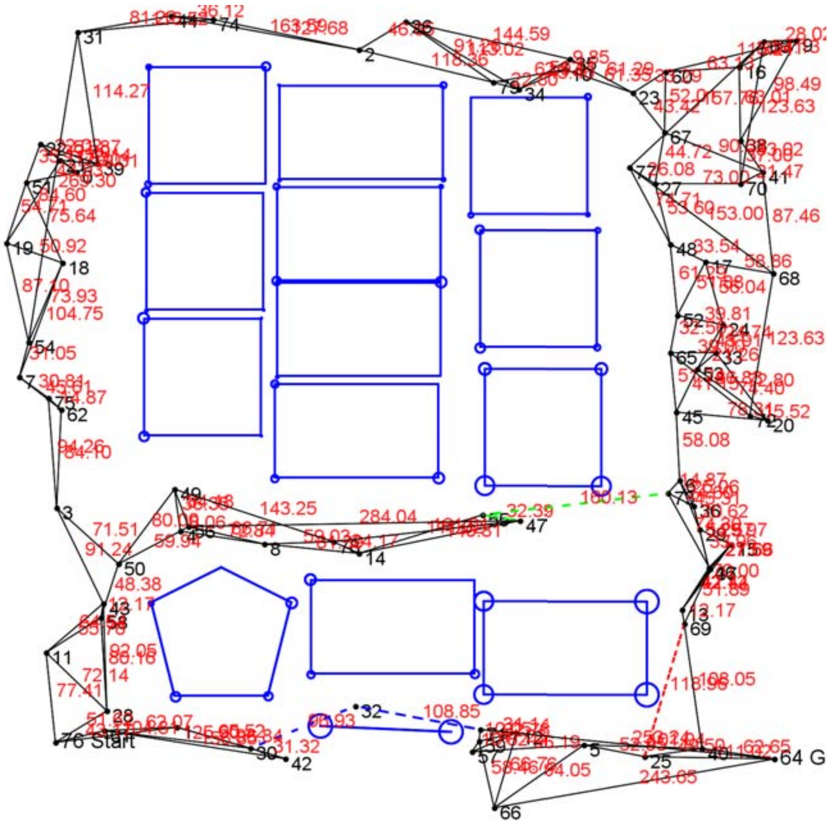


N=80, m=5
Pre-processed N=13
I $|\mathcal{S}|$=8.000E+26
C $|\mathcal{S}|$=1.333E+31
D $|\mathcal{S}|$=9.774E+122

Pre-processed graph



Graph 16



N=80, m=5

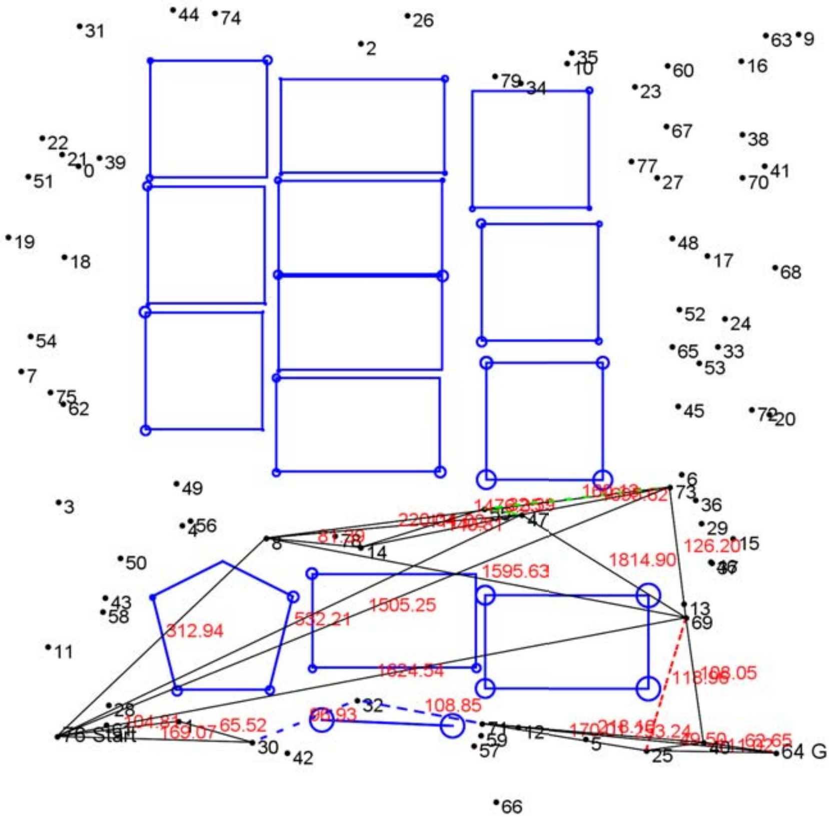
Pre-processed N=14

I $|\mathcal{S}|$=8.000E+26

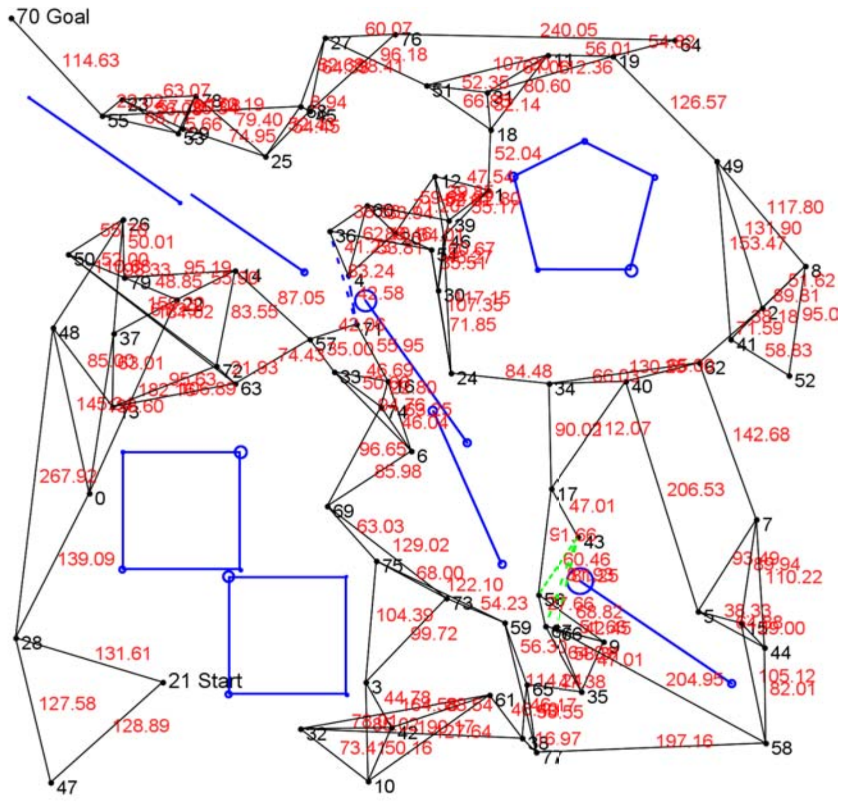
C $|\mathcal{S}|$=2.222E+35

D $|\mathcal{S}|$=9.774E+122

Pre-processed graph

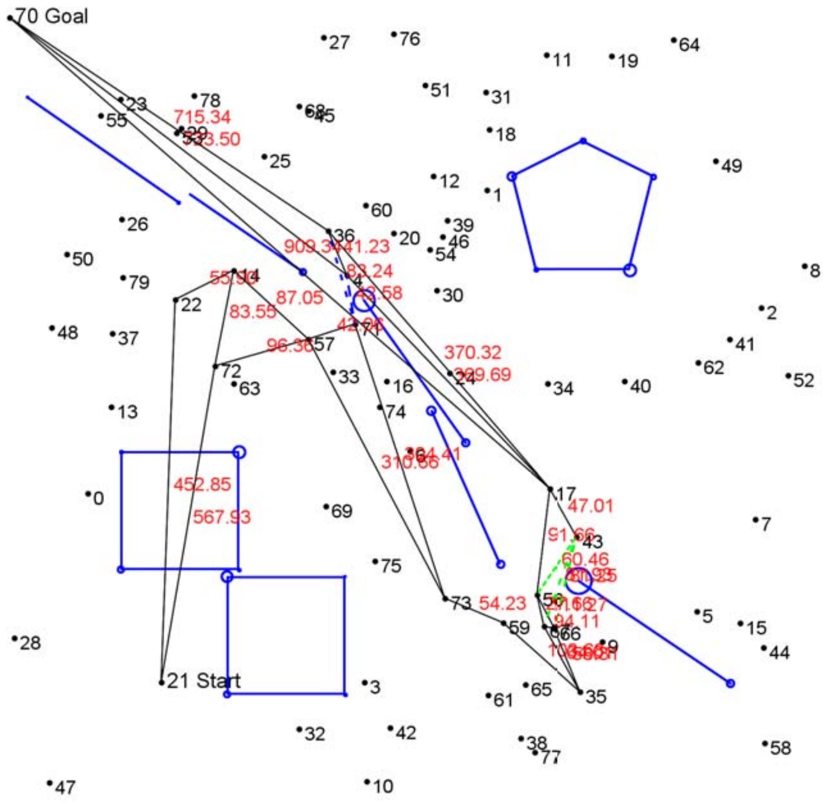


Graph 17

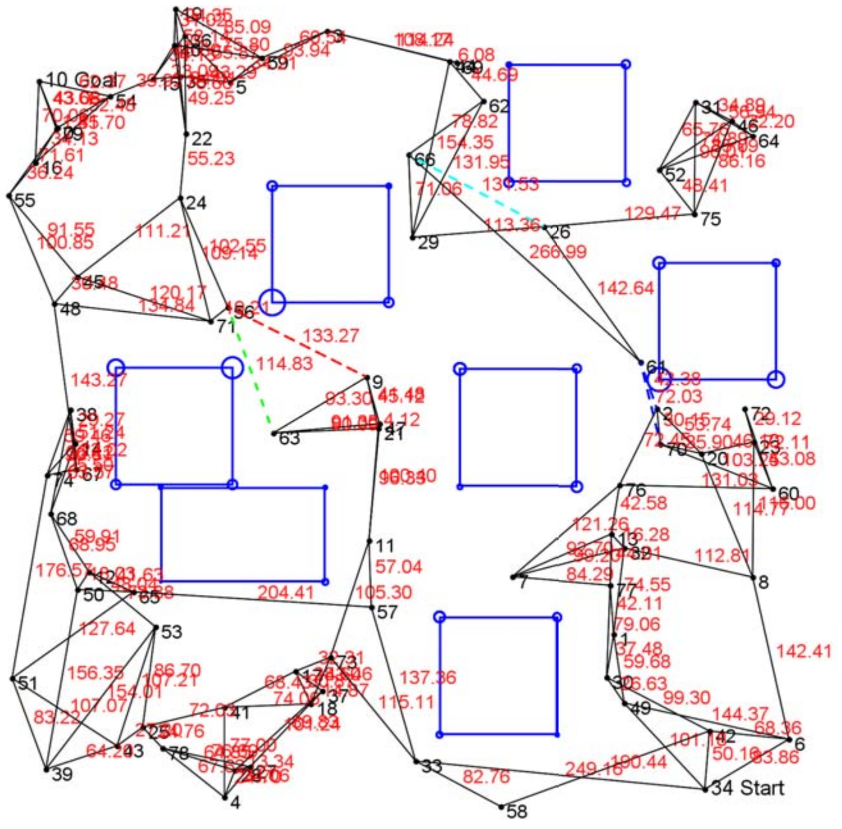


N=80, m=5
Pre-processed N=17
I $|\mathcal{S}|$=8.000E+26
C $|\mathcal{S}|$=2.646E+47
D $|\mathcal{S}|$=9.774E+122

Pre-processed graph



Graph 18



N=80, m=5

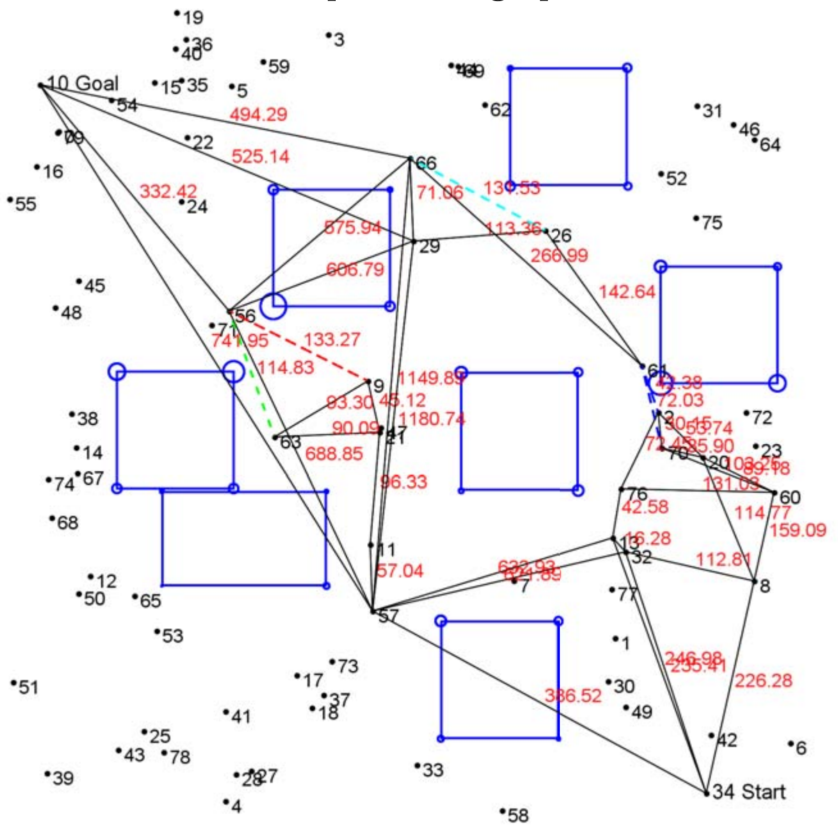
Pre-processed N=20

I $|\mathcal{S}|$=8.000E+26

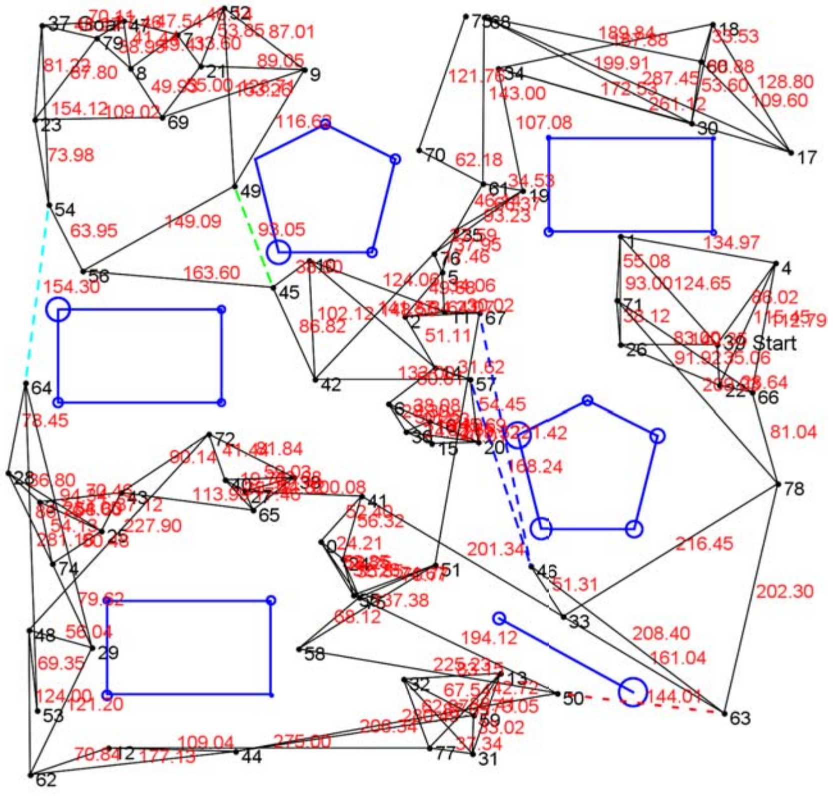
C $|\mathcal{S}|$=1.333E+31

D $|\mathcal{S}|$=9.774E+122

Pre-processed graph

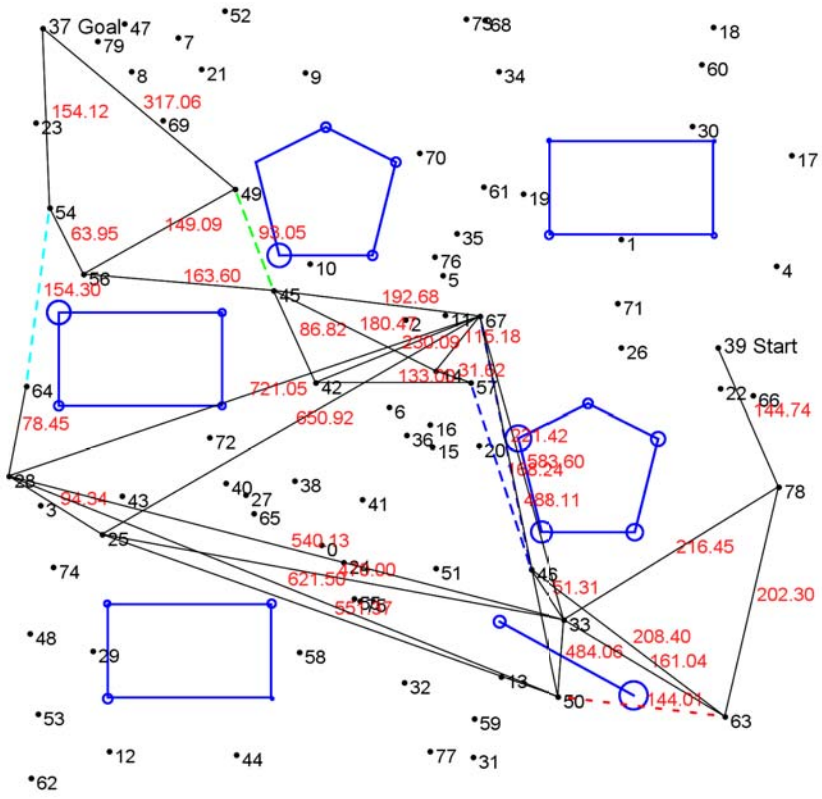


Graph 19

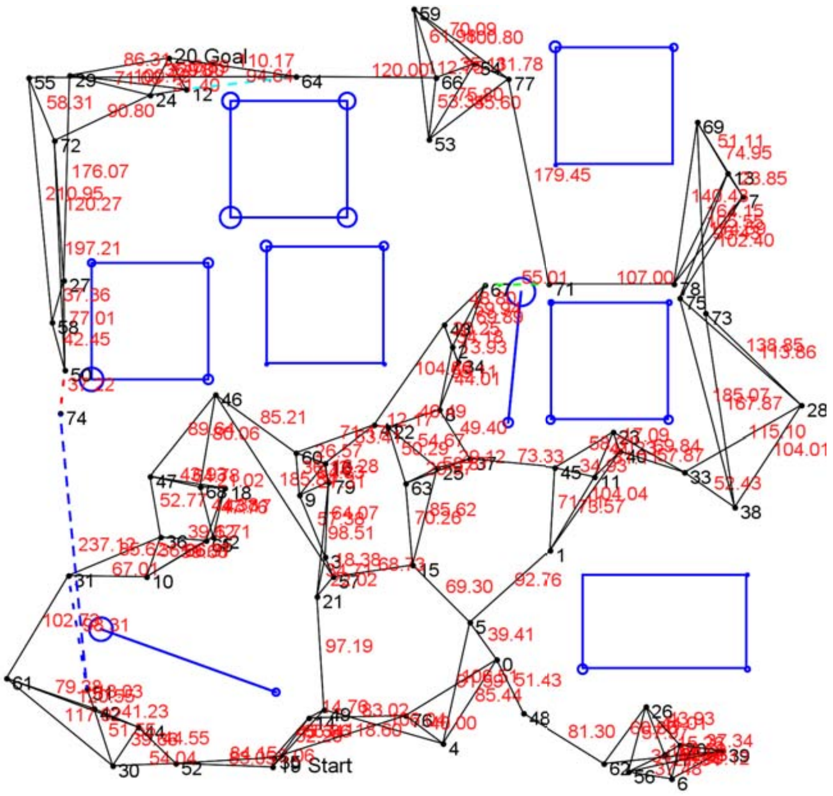


N=80, m=5
Pre-processed N=18
I $|\mathcal{S}|$=8.000E+26
C $|\mathcal{S}|$=1.333E+31
D $|\mathcal{S}|$=9.774E+122

Pre-processed graph



Graph 20



N=80, m=5

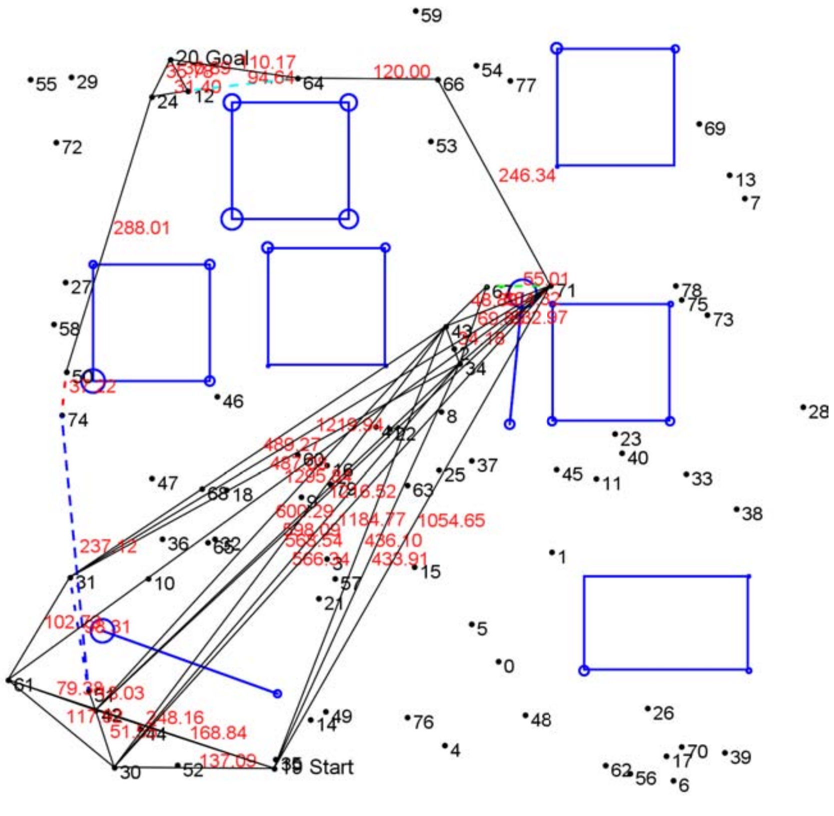
Pre-processed N=17

I $|\mathcal{S}|$=8.000E+26

C $|\mathcal{S}|$=1.333E+31

D $|\mathcal{S}|$=9.774E+122

Pre-processed graph



Appendix B

Format Definition Examples

Below are examples of the various file formats that are used to describe the graphs and POMDP models throughout this research.

B.1 Graph Definition File

The graphs used in the experiments for this research are all available electronically from `http://www.cs.bham.ac.uk/~mlk/graphs.zip`. Here is the definition file for Graph 8 for the purposes of illustration.

`The archive is also attached to this document, accessible via this link.`

Example definition for a graph

```
#Node data
#N=Node ID, x, y, rotation in degrees
N=0, 81.00, 70.00, 48.50
N=1, 90.00, 280.00, 221.09
N=2, 210.00, 327.00, 342.27
N=3, 564.00, 423.00, 279.55
N=4, 389.00, 507.00, 61.37
N=5, 72.00, 222.00, 324.77
N=6, 455.00, 378.00, 256.83
N=7, 685.00, 561.00, 152.33
N=8, 272.00, 319.00, 338.63
N=9, 87.00, 223.00, 195.44
N=10, 587.00, 414.00, 224.07
N=11, 163.00, 261.00, 295.22
N=12, 608.00, 428.00, 132.51
N=13, 102.00, 392.00, 93.52
N=14, 2.00, 50.00, 283.61
N=15, 212.00, 550.00, 87.77
N=16, 265.00, 585.00, 186.48
N=17, 281.00, 273.00, 131.27
N=18, 333.00, 595.00, 259.47
N=19, 373.00, 207.00, 158.75
N=20, 239.00, 341.00, 68.03
N=21, 382.00, 223.00, 354.49
N=22, 487.00, 297.00, 205.16
N=23, 105.00, 267.00, 317.40
N=24, 133.00, 584.00, 125.18
```

```
N=25, 363.00, 283.00, 94.32
N=26, 80.00, 193.00, 251.60
N=27, 560.00, 119.00, 62.89
N=28, 73.00, 39.00, 130.73
N=29, 141.00, 55.00, 229.14
N=30, 469.00, 44.00, 131.20
N=31, 543.00, 644.00, 353.59
N=32, 454.00, 152.00, 245.01
N=33, 519.00, 39.00, 161.74
N=34, 100.00, 485.00, 339.11
N=35, 511.00, 574.00, 46.75
N=36, 241.00, 293.00, 70.37
N=37, 286.00, 650.00, 133.51
N=38, 340.00, 193.00, 310.63
N=39, 449.00, 484.00, 137.12
N=40, 661.00, 291.00, 274.55
N=41, 547.00, 452.00, 316.18
N=42, 199.00, 662.00, 329.38
N=43, 216.00, 429.00, 130.89
N=44, 122.00, 356.00, 268.16
N=45, 9.00, 630.00, 354.50
N=46, 189.00, 228.00, 88.75
N=47, 677.00, 468.00, 105.29
N=48, 472.00, 449.00, 171.17
N=49, 194.00, 312.00, 290.22

#Edge data
#E=Node number 1, Node number 2, cost
E=0, 28, 32.02
E=0, 29, 61.85
E=0, 14, 81.49
E=0, 26, 123.00
E=1, 23, 19.85
E=1, 9, 57.08
E=1, 5, 60.73
E=1, 11, 75.43
E=2, 49, 21.93
E=2, 20, 32.20
E=2, 36, 46.01
E=2, 8, 62.51
E=3, 10, 24.70
E=3, 41, 33.62
E=3, 12, 44.28
E=3, 48, 95.60
E=4, 39, 64.26
E=4, 48, 101.26
E=4, 18, 104.31
E=4, 35, 139.19
E=5, 9, 15.03
E=5, 26, 30.08
E=5, 23, 55.80
E=6, 48, 73.01
E=6, 22, 87.09
E=6, 39, 106.17
E=6, 41, 118.07
E=7, 47, 93.34
E=7, 12, 153.68
E=7, 31, 164.48
E=7, 35, 174.48
E=8, 20, 39.66
E=8, 36, 40.46
E=8, 17, 46.87
E=9, 26, 30.81
E=9, 23, 47.54
E=10, 12, 25.24
E=10, 41, 55.17
E=10, 47, 104.96
E=11, 46, 42.01
E=11, 23, 58.31
E=11, 49, 59.68
```

```
E=12, 41, 65.55
E=13, 44, 41.18
E=13, 34, 93.02
E=13, 43, 119.85
E=13, 49, 121.92
E=14, 28, 71.85
E=14, 29, 139.09
E=14, 26, 162.89
E=15, 16, 63.51
E=15, 24, 86.01
E=15, 42, 112.75
E=15, 43, 121.07
E=16, 37, 68.31
E=16, 18, 68.73
E=16, 42, 101.41
E=17, 36, 44.72
E=17, 20, 79.92
E=17, 25, 82.61
E=18, 37, 72.35
E=18, 42, 149.82
E=19, 21, 18.36
E=19, 38, 35.85
E=19, 25, 76.66
E=19, 32, 97.91
E=20, 36, 48.04
E=21, 38, 51.61
E=21, 25, 62.94
E=21, 32, 101.12
E=22, 25, 124.79
E=22, 32, 148.71
E=22, 48, 152.74
E=24, 42, 102.18
E=24, 34, 104.36
E=24, 37, 166.63
E=27, 33, 89.89
E=27, 32, 111.02
E=27, 30, 117.92
E=27, 40, 199.46
E=28, 29, 69.86
E=28, 46, 221.76
E=29, 44, 301.60
E=30, 33, 50.25
E=30, 38, 197.08
E=30, 40, 312.85
E=31, 35, 76.97
E=31, 39, 185.57
E=31, 47, 221.21
E=33, 38, 236.13
E=33, 40, 289.25
E=34, 43, 128.81
E=34, 44, 130.86
E=35, 39, 109.29
E=37, 43, 231.82
E=40, 47, 177.72
E=44, 49, 84.38

#Start node ID
S=14

#Goal node ID
G=33

#Cluster and uncertain edge data
#C=Cluster ID, <Edge pair 1>, <Edge pair 2>,...
#<Edge pair>=node ID 1, node ID 2 being an uncertain edge
C=0, 17, 25
C=1, 28, 46, 29, 44
C=2, 22, 32

#Edge ordering
```

```
#The start belief state is given as a series of edge status combinations.
#In a graph of 3 uncertain edges, there are 8 combinations: FFF,FFB,FBF,FBB,...,BBB
#The bit positions of the statuses for each edge are given below. So the last character
#in the example above would be the status of the edge at bit position 0, the middle character
#refers to the edge at position 1, and so on.
#EO=Bit position for edge, <Edge pair>
EO=0, 22, 32
EO=1, 29, 44
EO=2, 28, 46
EO=3, 17, 25

#Starting belief state, starting with the all F combination and working through to all B
B=0.089083, 0.089083, 0.010480, 0.010480, 0.062882, 0.062882, 0.237555, 0.237555, 0.022271, 0.022271, 0.002620, 0.002620, 0.015721, 0.015721, 0.059389, 0.059389

#Observation data
#O=Node ID making the observation, <Edge pair> being observed, P(block|block), P(block|free)
O=36, 17, 25, 0.800000, 0.200000
O=17, 17, 25, 1.000000, 0.000000
O=25, 17, 25, 1.000000, 0.000000
O=8, 17, 25, 0.700000, 0.200000
O=29, 28, 46, 1.000000, 0.000000
O=29, 29, 44, 1.000000, 0.000000
O=28, 28, 46, 1.000000, 0.000000
O=28, 29, 44, 0.900000, 0.100000
O=46, 28, 46, 1.000000, 0.000000
O=44, 29, 44, 1.000000, 0.000000
O=0, 29, 44, 0.900000, 0.100000
O=6, 22, 32, 0.800000, 0.200000
O=22, 22, 32, 1.000000, 0.000000
O=32, 22, 32, 1.000000, 0.000000
O=48, 22, 32, 0.600000, 0.350000

#Obstacle data
#OB=x, y coordinates of obstacle in world space, x, y of obstacle handle in obstacle co-ordinates
#then a comma separated list of <Point data>
#<Point data>=x, y coordinates of vertex (Gaussian mean) in obstacle space, cov(1,1), cov(1,2), cov(2,2)
#cov(n,n) specifies co-variances matrix for bi-variate Gaussian distribution with cov(2,1)==cov(1,2).
OB=599.00, 309.00, 103.00, 91.00, 103.00, 91.00, 16.000000, 0.000000, 16.000000, 0.00, 0.00, 169.000000, 0.000000, 169.000000
OB=546.00, 382.00, 1.00, 87.00, 1.00, 87.00, 25.000000, 0.000000, 25.000000, 0.00, 0.00, 25.000000, 0.000000, 25.000000
OB=618.00, 381.00, 0.00, 68.00, 0.00, 68.00, 25.000000, 0.000000, 25.000000, 6.00, 0.00, 25.000000, 0.000000, 25.000000
OB=298.00, 410.00, 0.00, 112.00, 0.00, 112.00, 225.000000, 0.000000, 225.000000, 32.00, 0.00, 225.000000, 0.000000, 225.000000
OB=381.00, 589.00, 0.00, 0.00, 0.00, 0.00, 16.000000, 0.000000, 16.000000, 100.00, 0.00, 16.000000, 0.000000, 16.000000, 100.00, 100.00, 16.000000, 0.000000, 16.000000, 0.00, 100.00, 16.000000, 0.000000, 16.000000, 0.00, 0.00, 16.000000, 0.000000, 16.000000
OB=224.00, 67.00, 60.00, 0.00, 60.00, 0.00, 21.307086, 0.000000, 21.307086, 120.00, 30.00, 78.733990, 0.000000, 78.733990, 100.00, 110.00, 9.270729, 0.000000, 9.270729, 20.00, 110.00, 73.907519, 0.000000, 73.907519, 0.00, 30.00, 71.670428, 0.000000, 71.670428, 60.00, 0.00, 21.307086, 0.000000, 21.307086
OB=424.00, 343.00, 128.00, 0.00, 128.00, 0.00, 1.193371, 0.000000, 1.193371, 0.00, 145.00, 1.488505, 0.000000, 1.488505
OB=532.00, 539.00, 0.00, 71.00, 0.00, 71.00, 1.324871, 0.000000, 1.324871, 57.00, 0.00, 29.013699, 0.000000, 29.013699
```

B.2 POMDP Files For The 5 Point Graph

Throughout this research we utilise the common POMDP model description format created by Cassandra (2003). The format contains the complete description of all POMDP components $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{O}, o, b_0 \rangle$. For problems run using Symbolic Perseus, we use an ADD model description format. The details for the format can be found in the "problems/README" and "problems/SYNTAX" text files contained in the Symbolic Perseus downloadable archive (Poupart 2009).

The following Sections show the conversion of the graph in Figure 4.1 (see page 84) to the two POMDP file specifications. For design convenience in the POMDP files, PRM nodes are labelled numerically not alphabetically, so node S becomes 0, A becomes 1 and so on.

Cassandra POMDP format

```
#Cassandra POMDP model generated automatically from policy graph
#Graph generated on Tue Jun 16 12:46:05 BST 2009
discount: 0.950
values: reward
states: s0F s1F s2F s3F s4F s0B s1B s2B s3B s4B absorb
actions: Goto0 Goto1 Goto2 Goto3 Goto4
observations: F B

start: 0.50000000 0 0 0 0 0.50000000 0 0 0 0 0

T: Goto0
1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1

T: Goto1
0 1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1

T: Goto2
0 0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0 0
```

```
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1

T: Goto3
0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1

T: Goto4
1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1

O: *
0.5 0.5
1.0 0.0
1.0 0.0
0.5 0.5
0.5 0.5
0.5 0.5
0.0 1.0
0.0 1.0
0.5 0.5
0.5 0.5
0.5 0.5

R: * : s0F : * : * 0
R: * : s1F : * : * 0
R: * : s2F : * : * 0
R: * : s3F : * : * 0
R: * : s4F : * : * 0
R: * : s0B : * : * 0
R: * : s1B : * : * 0
R: * : s2B : * : * 0
R: * : s3B : * : * 0
R: * : s4B : * : * 0
R: * : absorb : * : * 0

R: * : s0F : s1F : * -2.0
R: * : s0F : s2F : * -1.0
R: * : s0F : s3F : * -2.0

R: * : s1F : s0F : * -2.0
R: * : s1F : s2F : * -2.0
R: * : s1F : s4F : * -2.0

R: * : s2F : s0F : * -1.0
R: * : s2F : s1F : * -2.0
R: * : s2F : s3F : * -2.0

R: * : s3F : s0F : * -2.0
R: * : s3F : s2F : * -2.0
```

```
R: * : s3F : s4F : * -5.0

R: * : s4F : * : * 10

R: * : s0B : s1B : * -2.0
R: * : s0B : s2B : * -1.0
R: * : s0B : s3B : * -2.0

R: * : s1B : s0B : * -2.0
R: * : s1B : s2B : * -2.0
R: * : s1B : s4B : * -2.0

R: * : s2B : s0B : * -1.0
R: * : s2B : s1B : * -2.0
R: * : s2B : s3B : * -2.0

R: * : s3B : s0B : * -2.0
R: * : s3B : s2B : * -2.0
R: * : s3B : s4B : * -5.0

R: * : s4B : * : * 10
```

ADD POMDP format

```
(variables
     (loc l0 l1 l2 l3 l4)
     (c1 c1F c1B)
)

(observations (obc1 obc1F obc1B)
)

init [* (loc (l0 (1.0)) (l1 (0.0)) (l2 (0.0)) (l3 (0.0)) (l4 (0.0)))
        (c1 (c1F (0.5)) (c1B (0.5)))
]

action goto_l0
   loc   (loc (l0 (locl0))
              (l1 (locl0))
              (l2 (locl0))
              (l3 (locl0))
              (l4 (locl4)) )
   c1 (SAMEc1)
   observe
      obc1 (obc1' (obc1F (0.5)) (obc1B (0.5)) )
   endobserve
   cost (loc (l0 (2000))
             (l1 (2))
             (l2 (1))
             (l3 (2))
             (l4 (0)))
endaction

action goto_l1
   loc   (loc (l0 (locl1))
              (l1 (locl1))
              (l2 (locl1))
              (l3 (locl3))
              (l4 (locl4))
          )
   c1 (SAMEc1)
   observe
      obc1 (c1' (c1F (obc1' (obc1F (1.0)) (obc1B (0.0)) ))
                (c1B (obc1' (obc1F (0.0)) (obc1B (1.0)) ) ) )
   endobserve
   cost (loc (l0 (2))
             (l1 (2000))
             (l2 (2))
             (l3 (2000))
             (l4 (0)))
endaction

action goto_l2
   loc   (loc (l0 (locl2))
              (l1 (locl2))
              (l2 (locl2))
              (l3 (locl2))
              (l4 (locl4)) )
   c1 (SAMEc1)
   observe
      obc1 (c1’ (c1F (obc1’ (obc1F (1.0)) (obc1B (0.0)) ))
                  (c1B (obc1’ (obc1F (0.0)) (obc1B (1.0)) ) ) )
   endobserve
   cost (loc (l0 (1))
             (l1 (2))
             (l2 (2000))
             (l3 (2))
             (l4 (0)))
endaction

action goto_l3
   loc   (loc (l0 (locl3))
              (l1 (locl1))
              (l2 (locl3))
              (l3 (locl3))
              (l4 (locl4)) )
   c1 (SAMEc1)
   observe
      obc1 (obc1’ (obc1F (0.5)) (obc1B (0.5)) )
   endobserve
   cost (loc (l0 (2))
             (l1 (2000))
             (l2 (2))
             (l3 (2000))
             (l4 (0)))
endaction

action goto_l4
   loc   (loc (l0 (locl0))
              (l1 (c1 (c1F (locl4))
                      (c1B (locl1))))
              (l2 (locl2))
              (l3 (locl4))
              (l4 (locl4)) )
   c1 (SAMEc1)
   observe
      obc1 (c1’ (c1F (obc1’ (obc1F (1.0)) (obc1B (0.0)) ))
                  (c1B (obc1’ (obc1F (0.0)) (obc1B (1.0)) ) ) )
   endobserve
   cost (loc (l0 (2000))
             (l1 (2))
             (l2 (2000))
             (l3 (5))
             (l4 (0)))
endaction

discount 0.999
tolerance 0.001
```

Appendix C

Clustering Algorithm

The following algorithm could be used to decide how uncertain edges should be arranged into a cluster set, given a dependent initial belief state and a clustering threshold.

Algorithm C.1 Clustering algorithm to create clusters based on edge dependencies

```
Input: $M$ set of $m$ uncertain edges, $0 \leq t \leq 1$ dependency threshold, $b_D$ initial dependent
    belief state
Output: cluster set $C$ containing $k \leq m$ clusters for edge set $M$
 1: $C \Leftarrow$ set of $m$ clusters, one edge per cluster  //initially assume independence
 2: for all $e_i \in M$ do
 3:    for all $e'_i \neq e_i \in M$ do
 4:       $p \Leftarrow \sum_{w \in b_D} P(w) \;:\; \texttt{freeStatus}(e_i) = \texttt{freeStatus}(e'_i)$  //sum probabilities of all
          worlds where statuses match
 5:       $q \Leftarrow \sum_{w \in b_D} P(w) \;:\; \texttt{freeStatus}(e_i) \neq \texttt{freeStatus}(e'_i)$  //sum probabilities of all
          worlds where statuses differ
 6:       if $|q - p| \geq t$ then  //is there a dependency?
 7:          $c_1 \Leftarrow$ cluster containing $e_i$
 8:          $c_2 \Leftarrow$ cluster containing $e'_i$
 9:          merge$(c_1, c_2)$
10:       end if
11:    end for
12: end for
13: return cluster set $C$
```

Example

The following example will demonstrate the output from Algorithm C.1 given two different two initial belief states. Assume we have a graph with two uncertain edges A and B, and two initial dependent belief states as shown in Table C.1.

World	b_D	
	Dependent	**Independent**
FF	0.5	0.25
FB	0	0.25
BF	0	0.25
BB	0.5	0.25

Table C.1: Two initial belief states for the clustering algorithm. One shows both edges fully correlated and the other shows complete independence.

If the dependent starting belief state is given to the clustering algorithm, the free statuses of edges A and B are fully correlated. When comparing the probabilities of the statuses of edges A and B matching in line 4, then

$$p = P(\text{FF}) + P(\text{BB}) = 0.5 + 0.5 = 1$$

and in line 5 when they differ

$$q = P(\text{FB}) + P(\text{BF}) = 0 + 0 = 0$$

The difference in the probabilities of matching and differing statuses is then $|q - p| = |1 - 0| = 1$, the greatest it can ever be. Thus, edges A and B will be deemed dependent on each other (line 6) and will be placed into the same cluster. The algorithm will return one cluster containing both edges A and B.

Alternatively, if the independent starting belief state is given to the algorithm it will decide to keep edges A and B in separate clusters. Comparing the probabilities as before, we find

$$p = P(\text{FF}) + P(\text{BB}) = 0.25 + 0.25 = 0.5$$

and

$$q = P(\text{FB}) + P(\text{BF}) = 0.25 + 0.25 = 0.5$$

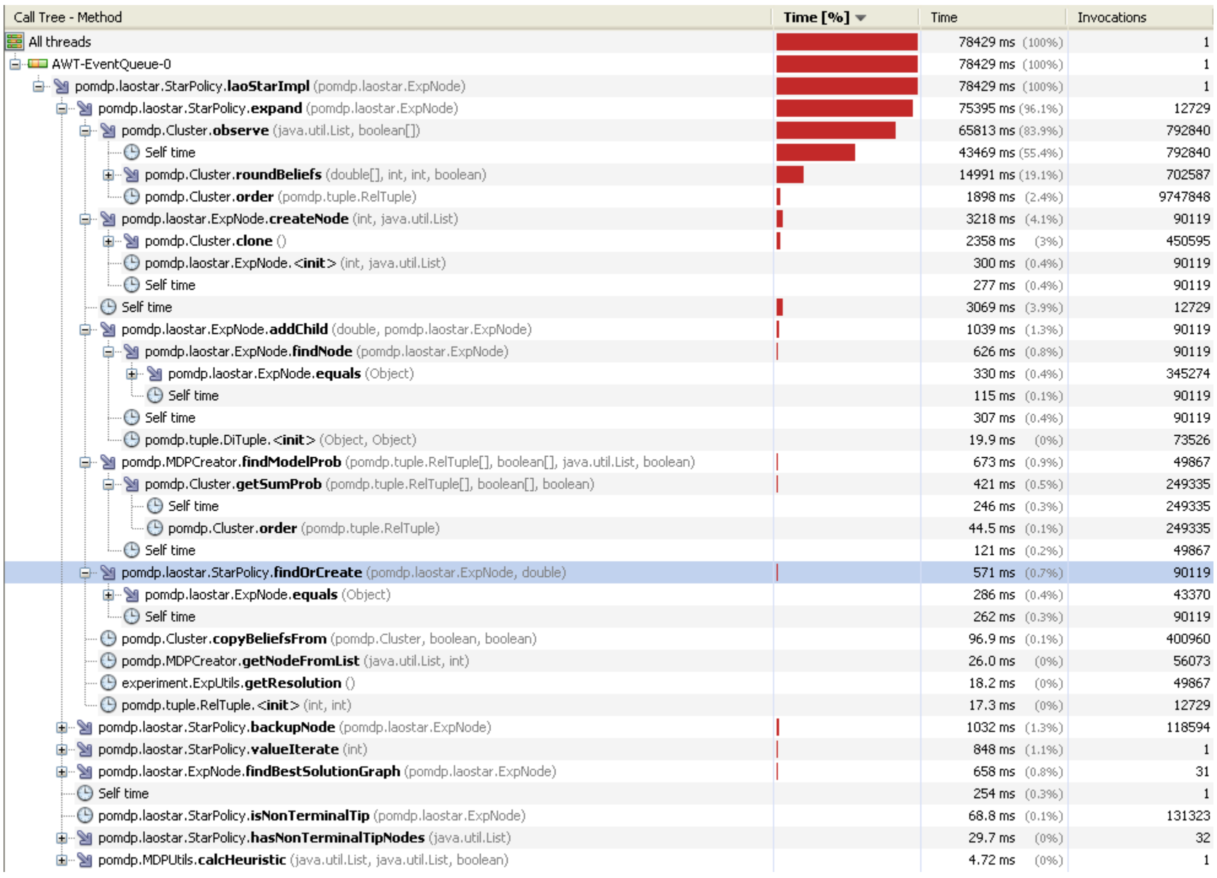
so $|q - p| = |0.5 - 0.5| = 0$. Edges A and B will remain in separate clusters, keeping them independent.

Appendix D

Profiling Graphs for Graph 19

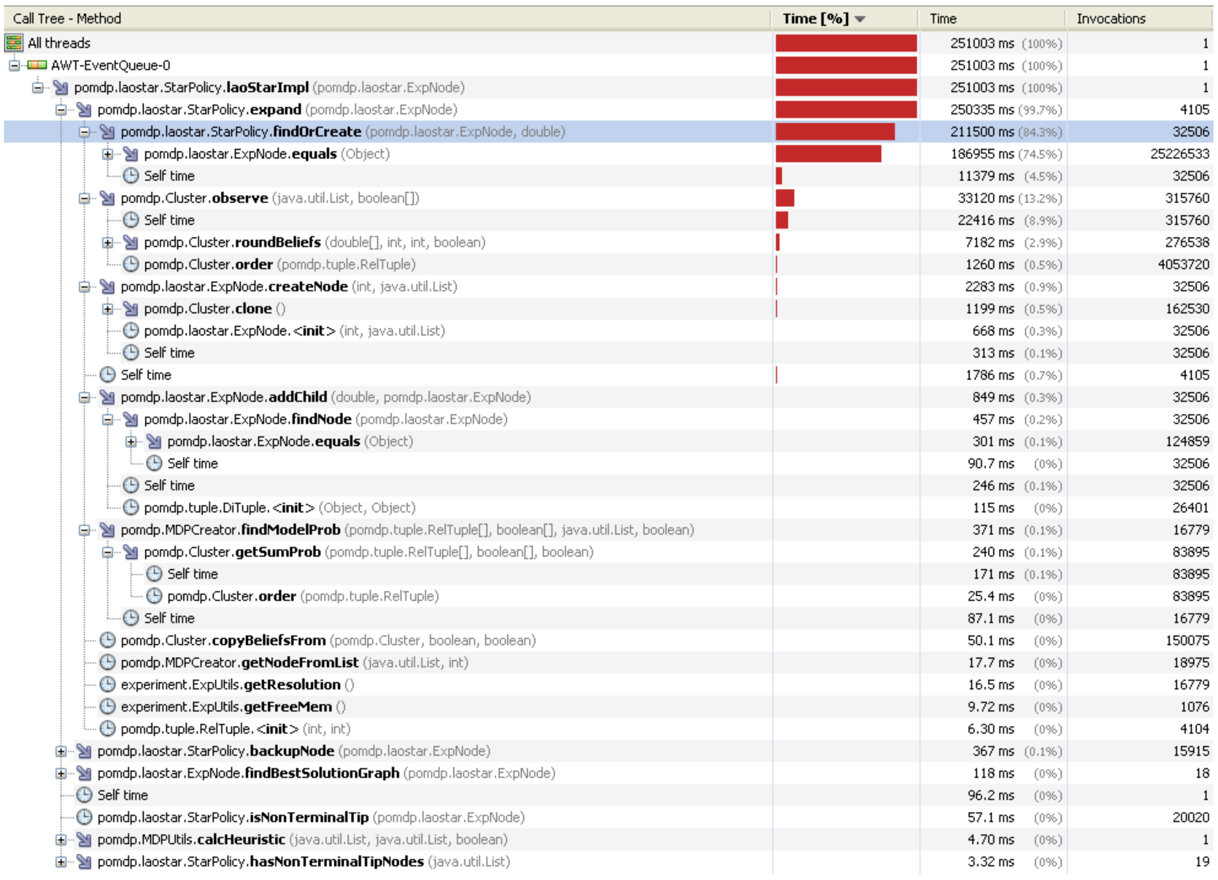
These are the profiling graphs for a run of LAO* and ALAO* for Graph 19 (pre-processed) with 10 uncertain edges using the clustered model. The highlighted lines show the method call that either creates a new node in the explicit graph or finds an equal, pre-existing one. This involves a hashtable lookup, and additionally, an insertion operation if a new node is created. The absolute times seen here are only roughly comparable to other runtimes in this thesis because the profiling data was collected on an Intel® Pentium® IV 3GHz with 1GB of memory instead of the configuration described in Section 4.5.4. The profiler also adds some overhead to method calls so the total runtime is increased compared to a normal execution of the algorithm.

In standard LAO* (Figure D.1) we see that determining new belief states as the result of an observation dominates the runtime (the `observe` method), taking 83.9% of the total time and hashtable manipulation (the `findOrCreate` method) accounts for only 0.7%. In ALAO* (Figure D.2), hashtable manipulation now accounts for 84.3% and belief state calculation only accounts for 13.2% of the runtime.



Call Tree - Method	Time [%]	Time	Invocations
All threads		78429 ms (100%)	1
AWT-EventQueue-0		78429 ms (100%)	1
pomdp.laostar.StarPolicy.**laoStarImpl** (pomdp.laostar.ExpNode)		78429 ms (100%)	1
pomdp.laostar.StarPolicy.**expand** (pomdp.laostar.ExpNode)		75395 ms (96.1%)	12729
pomdp.Cluster.**observe** (java.util.List, boolean[])		65813 ms (83.9%)	792840
Self time		43469 ms (55.4%)	792840
pomdp.Cluster.**roundBeliefs** (double[], int, int, boolean)		14991 ms (19.1%)	702587
pomdp.Cluster.**order** (pomdp.tuple.RelTuple)		1898 ms (2.4%)	9747848
pomdp.laostar.ExpNode.**createNode** (int, java.util.List)		3218 ms (4.1%)	90119
pomdp.Cluster.**clone** ()		2358 ms (3%)	450595
pomdp.laostar.ExpNode.**<init>** (int, java.util.List)		300 ms (0.4%)	90119
Self time		277 ms (0.4%)	90119
Self time		3069 ms (3.9%)	12729
pomdp.laostar.ExpNode.**addChild** (double, pomdp.laostar.ExpNode)		1039 ms (1.3%)	90119
pomdp.laostar.ExpNode.**findNode** (pomdp.laostar.ExpNode)		626 ms (0.8%)	90119
pomdp.laostar.ExpNode.**equals** (Object)		330 ms (0.4%)	345274
Self time		115 ms (0.1%)	90119
Self time		307 ms (0.4%)	90119
pomdp.tuple.DiTuple.**<init>** (Object, Object)		19.9 ms (0%)	73526
pomdp.MDPCreator.**findModelProb** (pomdp.tuple.RelTuple[], boolean[], java.util.List, boolean)		673 ms (0.9%)	49867
pomdp.Cluster.**getSumProb** (pomdp.tuple.RelTuple[], boolean[], boolean)		421 ms (0.5%)	249335
Self time		246 ms (0.3%)	249335
pomdp.Cluster.**order** (pomdp.tuple.RelTuple)		44.5 ms (0.1%)	249335
Self time		121 ms (0.2%)	49867
pomdp.laostar.StarPolicy.**findOrCreate** (pomdp.laostar.ExpNode, double)		571 ms (0.7%)	90119
pomdp.laostar.ExpNode.**equals** (Object)		286 ms (0.4%)	43370
Self time		262 ms (0.3%)	90119
pomdp.Cluster.**copyBeliefsFrom** (pomdp.Cluster, boolean, boolean)		96.9 ms (0.1%)	400960
pomdp.MDPCreator.**getNodeFromList** (java.util.List, int)		26.0 ms (0%)	56073
experiment.ExpUtils.**getResolution** ()		18.2 ms (0%)	49867
pomdp.tuple.RelTuple.**<init>** (int, int)		17.3 ms (0%)	12729
pomdp.laostar.StarPolicy.**backupNode** (pomdp.laostar.ExpNode)		1032 ms (1.3%)	118594
pomdp.laostar.StarPolicy.**valueIterate** (int)		848 ms (1.1%)	1
pomdp.laostar.ExpNode.**findBestSolutionGraph** (pomdp.laostar.ExpNode)		658 ms (0.8%)	31
Self time		254 ms (0.3%)	1
pomdp.laostar.StarPolicy.**isNonTerminalTip** (pomdp.laostar.ExpNode)		68.8 ms (0.1%)	131323
pomdp.laostar.StarPolicy.**hasNonTerminalTipNodes** (java.util.List)		29.7 ms (0%)	32
pomdp.MDPUtils.**calcHeuristic** (java.util.List, java.util.List, boolean)		4.72 ms (0%)	1

Figure D.1: The profiling data for LAO* solving Graph 19.



```
Call Tree - Method                                                                Time [%]   Time                         Invocations
All threads                                                                                  251003 ms  (100%)                      1
  AWT-EventQueue-0                                                                           251003 ms  (100%)                      1
    pomdp.laostar.StarPolicy.laoStarImpl (pomdp.laostar.ExpNode)                             251003 ms  (100%)                      1
      pomdp.laostar.StarPolicy.expand (pomdp.laostar.ExpNode)                                250335 ms  (99.7%)                  4105
        pomdp.laostar.StarPolicy.findOrCreate (pomdp.laostar.ExpNode, double)                211500 ms  (84.3%)                 32506
          pomdp.laostar.ExpNode.equals (Object)                                              186955 ms  (74.5%)              25226533
          Self time                                                                          11379 ms  (4.5%)                   32506
        pomdp.Cluster.observe (java.util.List, boolean[])                                    33120 ms  (13.2%)                 315760
          Self time                                                                          22416 ms  (8.9%)                  315760
          pomdp.Cluster.roundBeliefs (double[], int, int, boolean)                           7182 ms  (2.9%)                   276538
          pomdp.Cluster.order (pomdp.tuple.RelTuple)                                         1260 ms  (0.5%)                  4053720
        pomdp.laostar.ExpNode.createNode (int, java.util.List)                               2283 ms  (0.9%)                    32506
          pomdp.Cluster.clone ()                                                             1199 ms  (0.5%)                   162530
          pomdp.laostar.ExpNode.<init> (int, java.util.List)                                 668 ms  (0.3%)                     32506
          Self time                                                                          313 ms  (0.1%)                     32506
        Self time                                                                            1786 ms  (0.7%)                     4105
        pomdp.laostar.ExpNode.addChild (double, pomdp.laostar.ExpNode)                       849 ms  (0.3%)                     32506
          pomdp.laostar.ExpNode.findNode (pomdp.laostar.ExpNode)                             457 ms  (0.2%)                     32506
            pomdp.laostar.ExpNode.equals (Object)                                            301 ms  (0.1%)                    124859
            Self time                                                                        90.7 ms  (0%)                      32506
          Self time                                                                          246 ms  (0.1%)                     32506
          pomdp.tuple.DiTuple.<init> (Object, Object)                                        115 ms  (0%)                       26401
        pomdp.MDPCreator.findModelProb (pomdp.tuple.RelTuple[], boolean[], java.util.List, boolean)  371 ms  (0.1%)         16779
          pomdp.Cluster.getSumProb (pomdp.tuple.RelTuple[], boolean[], boolean)              240 ms  (0.1%)                     83895
            Self time                                                                        171 ms  (0.1%)                     83895
            pomdp.Cluster.order (pomdp.tuple.RelTuple)                                       25.4 ms  (0%)                      83895
          Self time                                                                          87.1 ms  (0%)                      16779
        pomdp.Cluster.copyBeliefsFrom (pomdp.Cluster, boolean, boolean)                      50.1 ms  (0%)                     150075
        pomdp.MDPCreator.getNodeFromList (java.util.List, int)                               17.7 ms  (0%)                      18975
        experiment.ExpUtils.getResolution ()                                                 16.5 ms  (0%)                      16779
        experiment.ExpUtils.getFreeMem ()                                                    9.72 ms  (0%)                       1076
        pomdp.tuple.RelTuple.<init> (int, int)                                               6.30 ms  (0%)                       4104
      pomdp.laostar.StarPolicy.backupNode (pomdp.laostar.ExpNode)                            367 ms  (0.1%)                     15915
      pomdp.laostar.ExpNode.findBestSolutionGraph (pomdp.laostar.ExpNode)                    118 ms  (0%)                          18
      Self time                                                                              96.2 ms  (0%)                          1
      pomdp.laostar.StarPolicy.isNonTerminalTip (pomdp.laostar.ExpNode)                      57.1 ms  (0%)                      20020
      pomdp.MDPUtils.calcHeuristic (java.util.List, java.util.List, boolean)                 4.70 ms  (0%)                          1
      pomdp.laostar.StarPolicy.hasNonTerminalTipNodes (java.util.List)                       3.32 ms  (0%)                         19
```

Figure D.2: The profiling data for ALAO* solving Graph 19.

Appendix E

Glossary of Symbols

Symbol	Meaning	Section	Page
C-space	total configuration space	2.1.1	13
C_{free}	collision free configuration space	2.1.1	14
C_{obst}	obstructed configuration space	2.1.1	14
D_{max}	maximum PRM edge length	2.1.3	16
G_{PRM}	a PRM graph	2.1.3	16
$\mathcal{S}$	MDP state space of N states	2.2.2	25
$\mathcal{A}$	MDP action set of K actions	2.2.2	25
$\mathcal{T}$	MDP transition matrix	2.2.2	25
$t(s, a, s')$	probability of transition from s to s' with action a	2.2.2	26
$\mathcal{R}$	MDP reward matrix	2.2.2	26
$r(s, a)$	MDP reward function for action a in state s	2.2.2	26
t_0	time zero, the start of state history	2.2.2.1	27
π	a policy vector: $\pi : \mathcal{S} \mapsto \mathcal{A}$	2.2.2.1	27
π^*	the optimal policy vector	2.2.2.1	27
$V^{\pi}(s)$	the value of state s under policy π	2.2.2.1	27
γ	the value function discount factor	2.2.2.2	28
DP	dynamic programming	2.2.2.3	29
VI	value iteration	2.2.2.3	29
ϵ	Bellman residual	2.2.2.3	30
s_0	an MDP initial state	2.2.3	32
NFSM	non-deterministic finite state machine	2.2.3	32
G	explicit graph	2.2.3	32
BSG	best partial solution graph	2.2.3	32
$\mathcal{O}$	POMDP observation set	2.3	38
$b(s)$	the agent's belief that s is the true POMDP state	2.3.1	39
b_0	a POMDP initial belief state	2.3.1	39
$\mathcal{B}$	N dimensional simplex, the belief space for a POMDP	2.3.1	40
PWLC	piecewise linear convex	2.3.2	42
$\mathcal{V}$	set of N size α-vectors representing a POMDP value function, each representing one hyperplane over $\mathcal{B}$	2.3.2.1	43
C_{const}	tuneable cost of collision for the MCC planner	3.2.3.2	56

Symbol	**Meaning**	**Section**	**Page**
p.m.f.	probability mass function	3.4.1.2	69
p.d.f.	probability density function	3.4.1.2	69
m	number of uncertain edges in a PRM graph	4.1	78
c_e	cost of edge e	4.1.1	79
V	set of size n PRM graph nodes	4.1.1	79
E	set of PRM graph edges	4.1.1	79
v_S	start node in a PRM graph	4.1.1	79
v_G	goal node in a PRM graph	4.1.1	79
W	set of possible worlds for a PRM graph	4.1.1	79
s_T	absorbing terminal POMDP state	4.2	80
c_G	reward for reach the goal in POMDP formulation	4.2	81
d	the discretisation resolution for the MDP	4.3.2	87
$\mathcal{B}_D$	the discretised dependent belief space	4.3.2.4	92
b_D	a dependent belief state	4.3.2.4	92
Δ	maximum desired discretisation error	4.3.2.3	92
m_c	number of uncertain edges in cluster c	4.4.3.1	107
C	set of clusters	4.4.3.1	107
k	number of clusters in C	4.4.3.1	107
$w_{c,i}$	sub-world i of cluster c	4.4.3.1	107
b_c	a belief state for cluster c	4.4.3.1	107
$\mathcal{B}_c$	the discretised belief space for cluster c	4.4.3.1	107
b_C	a discretised belief state for cluster set C	4.4.3.1	107
$\max_{\mathrm{KL}}$	the maximum Kullback-Leibler divergence allowed between two approximately equal belief states	5.2.1.1	168

Bibliography

AI Access. Kullback-Leibler distance. In *Glossary of Data Modeling*. AI Access, 2008. URL http://www.aiaccess.net/English/Glossaries/GlosMod/e_gm_kullbak.htm.

R. Alterovitz, A. Lim, K. Goldberg, G. S. Chirikjian, and A. M. Okamura. Steering flexible needles under Markov motion uncertainty. In *Proceedings of the IEEE/RSJ International Joint Conference on Intelligent Robots and Systems (IROS)*, pages 120–125, August 2005. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.1597.

R. Alterovitz, T. Siméon, and K. Y. Goldberg. The Stochastic Motion Roadmap: A Sampling Framework for Planning with Markov Motion Uncertainty. In *Robotics: Science and Systems 2007*, Atlanta, GA, USA, June 2007. URL http://www.roboticsproceedings.org/rss03/p30.pdf.

R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In M. R. Lightner and J. A. G. Jess, editors, *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 188–191, Santa Clara, California, USA, November 1993. ISBN 0-8186-4490-7. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.2128.

J. Barraquand and P. Ferbach. Motion planning with uncertainty: The information space approach. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 1341–1348, May 1995. doi: 10.1109/ROBOT.1995.525465. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=525465.

J. Barraquand and J. C. Latombe. Robot motion planning: A distributed representation approach. *International Journal of Robotics Research*, 10(6):628–649, 1991.

R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, second edition, 1994. ISBN 978-0898713282. URL http://netlib.org/linalg/html_templates/report.html.

A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. Technical Report UM-CS-1993-002, University of Massachusetts, Amherst MA 01003, March 1993. URL http://citeseer.ist.psu.edu/barto93learning.html.

R. E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

D. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27(3):610–616, Jun 1982. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1102980`.

D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Athena Scientific, P.O. Box 805, Nashua, NH 03061-0805, U.S.A., 1989. ISBN 1-886529-01-9. URL `http://hdl.handle.net/1721.1/3719`.

Z. Bnaya, A. Felner, and S. E. Shimony. The Canadian traveller problem with remote sensing. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, Pasadena, California, 2009. URL `http://www.ise.bgu.ac.il/faculty/felner/research/felner126.pdf`.

R. Bohlin. Path planning in practice: Lazy evaluation on a multi-resolution grid. In *Proceedings of the IEEE/RSJ International Joint Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 49–54, Maui, Hi, USA, 2001. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=973335`.

R. Bohlin and L. Kavraki. Path planning using lazy PRM. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 1, pages 521–528, 2000. URL `http://citeseer.ist.psu.edu/article/bohlin00path.html`.

B. Bonet and H. Geffner. Solving POMDPs: RTDP-Bel vs. point-based algorithms. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1641–1646, Pasadena, California, July 2009. AAAI. URL `http://www.ldc.usb.ve/~bonet/reports/IJCAI09-rtdpbel.pdf`.

V. Boor, M. H. Overmars, and A. F. van der Stappen. Gaussian sampling for probabilistic roadmap planners. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 2, pages 1018–1023, Detroit, Michigan, May 1999. URL `http://citeseer.ist.psu.edu/boor01gaussian.html`.

R. I. Brafman. A heuristic variable grid solution method for POMDPs. In *Proceedings of the 14th National Conference on AI (AAAI)*, pages 727–733, 1997. URL `http://www.aaai.org/Papers/AAAI/1997/AAAI97-113.pdf`.

D. Burago, M. De Rougemont, and A. Slissenko. On the complexity of partially observed Markov decision processes. *Theoretical Computer Science*, 157:161–183, 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.8947`.

J. Burlet, O. Aycard, and T. Fraichard. Robust motion planning using Markov decision processes and quadtree decomposition. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, pages 2820–2825, New Orleans, Los Angeles, April 2004. URL `http://emotion.inrialpes.fr/bibemotion/2004/BAF04`.

B. Burns and O. Brock. Sampling-based motion planning using uncertain knowledge. Technical report, University of Massachusetts Amherst, 2006. URL `http://www-robotics.cs.umass.edu/~oli/publications/src/tr06-30.pdf`.

B. Burns and O. Brock. Sampling-Based Motion Planning With Sensing Uncertainty. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, pages 3313–3318, Rome, Italy, April 2007. URL `http://robotics.cs.umass.edu/~oli/publications/src/2007-icra-a.pdf`.

A. Cassandra, M. L. Littman, and N. L. Zhang. Incremental Pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 54–61, San Francisco, CA, 1997. Morgan Kaufmann Publishers. URL `http://citeseer.ist.psu.edu/cassandra97incremental.html`.

A. R. Cassandra. Input POMDP file format, 2003. URL `http://www.pomdp.org/pomdp/code/pomdp-file-spec.shtml`.

A. R. Cassandra. pomdp-solve v5.3, 2005. URL `http://www.pomdp.org/pomdp/code/index.shtml`.

A. Censi, D. Calisi, A. De Luca, and G. Oriolo. A Bayesian framework for optimal motion planning with uncertainty. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, pages 1798–1805, Pasadena, California, 2008. URL `http://purl.org/censi/research/2008-icra-ppu.pdf`.

P. P. Chakrabarti, S. Ghose, and S. C. DeSarkar. Admissibility of AO* when heuristics overestimate. *Journal of Artificial Intelligence Research*, 34(1):97–113, 1987. URL `http://portal.acm.org/citation.cfm?id=42946`.

P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. Heuristic search in restricted memory. *Journal of Artificial Intelligence Research*, 41:197–221, 1989. URL `http://portal.acm.org/citation.cfm?id=74117`.

H. Cheng. *Algorithms for Partially Observable Markov Decision Processes.* PhD thesis, School of Commerce, University of British Columbia, 1988.

H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations.* The MIT Press, 55 Hayward Street, Cambridge, MA 02142-1493, USA, 2005.

T. M. Cover and J. A. Thomas. *Element of Information Theory.* John Wiley and Sons, inc., 111 River Street, Hoboken, NJ 07030-5774, second edition, 2006. ISBN 978-0-471-24195-9. URL `http://www.elementsofinformationtheory.com/`.

P. Dai and J. Goldsmith. Topological value iteration algorithm for Markov decision processes. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1860–1865, Hyderbad, India, January 2007. URL `http://www.cs.washington.edu/homes/daipeng/papers/IJCAI07-300.pdf`.

P. Dai and E. A. Hansen. Prioritizing Bellman backups without a priority queue. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 113–119, Providence, Rhode Island, September 2007. URL `http://www.cse.msstate.edu/~hansen/papers/ICAPS07-015.pdf`.

P. Dai, Mausam, and D. P. Weld. Focused topological value iteration. In *Proceeedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 82–89, Thessaloniki, Greece, September 2009. URL http://www.cs.washington.edu/homes/weld/papers/dai-icaps09.pdf.

T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Journal of Artificial Intelligence Research*, 76(1–2):35–74, July 1995. URL http://citeseer.ist.psu.edu/article/dean95planning.html.

R. Dearden and C. Boutilier. Abstraction and approximate decision-theoretic planning. *Journal of Artificial Intelligence Research*, 89:219–283, 1997.

R. Dearden and M. Kneebone. Uncertain probabilistic roadmaps with observations. In *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*, pages 33–40, Edinburgh, UK, December 2008. URL http://www.cs.bham.ac.uk/~mlk/uncertain_prm.pdf.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. URL http://jmvidal.cse.sc.edu/library/dijkstra59a.pdf.

A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo Methods In Practice*. Springer-Verlag, Springer Publishing Company, 11 West 42nd Street, 15th Floor, New York, NY 10036, June 2001. ISBN 0-387-95146-6.

Z. Feng and E. A. Hansen. Symbolic heuristic search for factored Markov decision processes. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, pages 455–460, Edmonton, Alberta, Canada, July 2002. AAAI Press. URL http://www.cse.msstate.edu/~hansen/papers/aaai02.pdf.

Z. Feng and S. Zilberstein. Region-based incremental pruning for POMDPs. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 146–153, Banff, Canada, 2004. AUAI Press. ISBN 0-9749039-0-6. URL http://anytime.cs.umass.edu/shlomo/papers/uai04.pdf.

H. H. González-Baños, D. Hsu, and J. C. Latombe. *Motion planning: Recent developments*, chapter 10. CRC Press, 2006. URL http://motion.comp.nus.edu.sg/papers/amr05.pdf.

E. A. Hansen. Solving POMDPs by searching in policy space. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 211–219, Madison, Wisconsin, July 1998. URL http://www.cse.msstate.edu/~hansen/papers/uai98.ps.

E. A. Hansen and Z. Feng. Dynamic programming for POMDPs using a factored state representation. In S. Chien, S. Kambhampati, and C. A. Knoblock, editors, *Proceedings of the 5th International Conference on AI Planning Systems (AIPS)*, pages 130–139, Breckenridge, Colorado, April 2000. ISBN 1-57735-111-8. URL http://www.cse.msstate.edu/~hansen/papers/aips00.pdf.

E. A. Hansen and S. Zilberstein. Heuristic search in cyclic AND/OR graphs. In *Proceedings of the 15th National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 412–418, 1998. URL http://citeseer.ist.psu.edu/article/hansen98heuristic.html.

E. A. Hansen and S. Zilberstein. A heuristic search algorithm for Markov decision problems. In *Bar-Ilan Symposium on the Foundation of Artificial Intelligence (BISFAI)*, Ramat Gan, Israel, 1999. URL http://rbrserver.cs.umass.edu/shlomo/papers/bisfai99b.pdf.

E. A. Hansen and S. Zilberstein. LAO* : A heuristic search algorithm that finds solutions with loops. *Journal of Artificial Intelligence Research*, 129(1–2):35–62, 2001. URL http://citeseer.ist.psu.edu/hansen01lao.html.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107, July 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4082128.

M. Hauskrecht. Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 13:33–94, 2000. URL http://www.jair.org/media/678/live-678-1858-jair.pdf.

J. Hoey and P. Poupart. Solving POMDPs with continuous or large discrete observation spaces. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1332–1338, Edinburgh, Scotland, August 2005. URL http://www.ijcai.org/papers/1376.pdf.

J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 279–288, Stockholm, Sweden, August 1999. URL http://www.cs.ubc.ca/~jhoey/papers/spudd99.ps.gz.

J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. Optimal and approximate stochastic planning using decision diagrams. Technical Report TR-00-05, University of British Columbia, Vancouver, BC Canada, June 2000. URL http://www.computing.dundee.ac.uk/staff/jessehoey/papers/TR-00-05.ps.gz.

D. Hsu, J. C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 3, pages 2719–2726, Albuquerque, New Mexico, April 1997. ISBN 0-7803-3612-7. doi: 10.1109/ROBOT.1997.619371. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.448&rep=rep1&type=pdf.

D. Hsu, R. Kindel, J. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. In *Workshop on the Algorithmic Foundations of Robotics*, 2000. URL http://citeseer.ist.psu.edu/article/hsu00randomized.html.

D. Hsu, T. Jiang, J. Reif, and Z. Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 3, pages 4420–4426, Taipei, Taiwan, September 2003. ISBN 0-7803-7736-2. doi: 10.1109/ROBOT.2003.1242285. URL `http://citeseer.ist.psu.edu/hsu03bridge.html`.

D. Hsu, J. C. Latombe, and H. Kurniawati. On the probabilistic foundations of probabilistic roadmap planning. *International Journal of Robotics Research*, 25(7):627–643, 2006. URL `http://citeseer.ist.psu.edu/748393.html`.

L. Jaillet and T. Siméon. A PRM-based motion planner for dynamically changing environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 1606–1611, Sendai, Japan, 2004. URL `http://citeseer.ist.psu.edu/745552.html`.

L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Journal of Artificial Intelligence Research*, 101:99–134, February 1998. URL `http://people.csail.mit.edu/lpk/papers/aij98-pomdp.pdf`.

R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960. URL `http://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf`.

K. Kant and S. W. Zucker. Toward efficient trajectory planning: The path-velocity decomposition. *International Journal of Robotics Research*, 5(3):72–89, 1986. URL `http://ijr.sagepub.com/cgi/content/abstract/5/3/72`.

L. E. Kavraki. *Random Networks in Configuration Space for Fast Path Planning.* PhD thesis, Stanford University, 1995.

L. E. Kavraki and J. C. Latombe. Probabilistic roadmaps for robot path planning. In *Practical Motion Planning in Robotics: Current Approaches and Future Challenges*, pages 33–53. John Wiley, West Sussex, England, 1998. URL `http://citeseer.csail.mit.edu/article/kavraki98probabilistic.html`.

L. E. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 12, pages 566–580, 1996. URL `http://citeseer.ist.psu.edu/kavraki96probabilistic.html`.

R. Kindel, D. Hsu, J. Latombe, and S. Rock. Kinodynamic motion planning amidst moving obstacles. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 1, pages 537–543, San Francisco, California, April 2000. ISBN 0-7803-5886-4. doi: 10.1109/ROBOT.2000.844109. URL `http://citeseer.ist.psu.edu/kindel00kinodynamic.html`.

M. Kneebone and R. Dearden. Navigation planning in probabilistic roadmaps with uncertainty. In *Proceedings of 19th International Conference on Automated Planning* *and Scheduling (ICAPS)*, pages 209–216, Thessaloniki, Greece, September 2009. URL http://www.cs.bham.ac.uk/~mlk/ICAPS09.pdf.

J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 2, pages 995–1001, San Francisco, California, April 2000. doi: 10.1109/ROBOT.2000.844730. URL http://www.kuffner.org/james/papers/kuffner_icra2000.pdf.

J. P. Laumond, P. E. Jacobs, M. Taïx, and R. M. Murray. A motion planner for nonholonomic mobile robots. *IEEE Transactions on Robotics and Automation*, 10(5):577–593, October 1994. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=326564&k2dockey=326564@ieeejrns.

M. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In A. Prieditis and S. Russell, editors, *Proceedings of the 12th International Joint Conference on Machine Learning*, pages 362–370, San Francisco, California, 1995a. Morgan Kaufmann. URL http://www.pomdp.org/pomdp/papers/ml95.ps.gz.

M. L. Littman. The Witness algorithm: Solving partially observable Markov decision processes. Technical report, Brown University, Providence, RI, USA, December 1994. URL http://www.cs.duke.edu/~mlittman/docs/witness-tm.ps.

M. L. Littman, T. L. Dean, and L. P. Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 394–402, Montreal, Québec, Canada, 1995b. URL http://citeseer.ist.psu.edu/littman95complexity.html.

W. S. Lovejoy. Computationally feasible bounds for partially observed Markov decision processes. *Operations Research*, 39(1):162–175, 1991. URL http://www.jstor.org/stable/171496.

J. J. Martin. *Bayesian Decision Problems and Markov Chains.* John Wiley and Sons, inc., 111 River Street, Hoboken, NJ 07030-5774, 1967. ISBN 978-0471573517.

L. Mero. A heuristic search algorithm with modifiable estimate. *Journal of Artificial Intelligence Research*, 23(1):13–27, 1984. ISSN 0004-3702. URL http://portal.acm.org/citation.cfm?id=322944.

P. E. Missiuro and N. Roy. Adapting probabilistic roadmaps to handle uncertain maps. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, pages 1261–1267, Orlando, Florida, May 2006. ISBN 0-7803-9505-0. doi: 10.1109/ROBOT.2006.1641882.

G. E. Monahan. A survery of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, January 1982. URL http://mansci.journal.informs.org/cgi/content/abstract/28/1/1.

A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13(1):103–130, October 1993. doi: 10.1007/BF00993104. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.7241`.

N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA: Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, Houston, Texas, October 2002. URL `http://citeseer.ist.psu.edu/muscettola02idea.html`.

C. Nielsen and L. E. Kavraki. A two level fuzzy PRM for manipulation planning. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 1716–1722. IEEE Press, November 2000. URL `http://citeseer.ist.psu.edu/nielsen00two.html`.

N. J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc. San Francisco, California, 1986. ISBN 978-0934613101.

N. J. Nilsson. *Artificial Intelligence: A New Synthesis.* Morgan Kaufmann Publishers, Inc. San Francisco, California, 1998. ISBN 1-55860-535-5.

C. Papadimitriou and J. N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987. URL `http://web.mit.edu/jnt/www/Papers/J016-87-mdp-complexity.pdf`.

J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. ISBN 0-201-05594-5.

J. Peng and S. Akella. Coordinating the motions of multiple robots with kinodynamic constraints. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 3, pages 4066–4073, Taipei, Taiwan, September 2003. URL `http://www.cs.rpi.edu/~sakella/papers/ICRA03.pdf`.

J. Peng and S. Akella. Coordinating multiple robots with kinodynamic constraints along specified paths. *International Journal of Robotics Research*, 24(4), April 2005. doi: 10.1177/0278364905051974. URL `http://ijr.sagepub.com/cgi/content/abstract/24/4/295`.

J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1025–1032, Acapulco, Mexico, August 2003. URL `http://www-2.cs.cmu.edu/~jpineau/files/jpineau-waml03.ps`.

I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5: 219–236, 1970.

J. M. Porta, M. T. J. Spaan, and N. Vlassis. Robot planning in partially observable continuous domains. In *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005. URL `ftp://ftp.science.uva.nl/pub/computer-systems/aut-sys/reports/Porta05rss.pdf`.

P. Poupart. *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes*. PhD thesis, University of Toronto, Toronto, Canada, 2005. URL `http://www.cs.uwaterloo.ca/~ppoupart/publications/ut-thesis/ut-thesis.pdf`.

P. Poupart. Symbolic perseus, 2009. URL `http://www.cs.uwaterloo.ca/~ppoupart/software.html`.

P. Poupart and C. Boutilier. Bounded finite state controllers. In *Proceedings of the 17th Annual Conference on Neural Information Processing Systems (NIPS)*, Vancouver, Canada, December 2003. URL `http://www.cs.toronto.edu/kr/publications/bfsc.pdf`.

S. Rodríguez, J. M. Lien, and N. M. Amato. A framework for planning motion in uncertain environments. Technical report, Texas A&M University, September 2006. URL `http://parasol.tamu.edu/groups/amatogroup/research/movingobjs/`.

S. Rodríguez, J. M. Lien, and N. M. Amato. A framework for planning motion in environments with moving obstacles. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3309–3314, San Diego, CA, USA, November 2007. doi: 10.1109/IROS.2007.4399540. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4399540`.

S. Ross, J. Pineau, and B. Chaib-draa. Theoretical analysis of heuristic search methods for online POMDPs. In *Proceedings of the 21th Annural Conference on Neural Information Processing Systems (NIPS)*, Vancouver, Canada, December 2007. URL `http://www.cs.mcgill.ca/~jpineau/files/sross-nips07-aems.pdf`.

S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32:663–704, 2008. URL `http://www.cs.mcgill.ca/~jpineau/files/sross-jair08.pdf`.

S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2002.

G. Sánchez and J. C. Latombe. On delaying collision checking in PRM planning : Application to multi-robot co-ordination. *International Journal of Robotics Research*, 21(1): 5–26, January 2002a. URL `http://citeseer.csail.mit.edu/sanchez02delaying.html`.

G. Sánchez and J. C. Latombe. Using a PRM planner to compare centralized and decoupled planning for multi-robot systems. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 2, pages 2112–2119, 2002b. doi: 10.1109/ROBOT.2002.1014852. URL `http://ai.stanford.edu/~latombe/papers/icra02-gil/final.pdf`.

G. Sánchez and J. C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In R.A. Jarvis and A. Zelinsky, editors, *Robotics*

Research: The Tenth International Symposium, pages 403–417. Springer Tracts in Advanced Robotics, Springer, 2003. URL `http://ai.stanford.edu/~latombe/papers/isrr01/spinger/latombe.pdf`.

R. D. Smallwood and E. J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

T. Smith and R. Simmons. Heuristic search value iteration for POMDPs. In *Proceedings of the 20th conference on Uncertainty in Artificial Intelligence (UAI)*, volume 70, pages 520–527, Banff, Canada, 2004. URL `http://www.ee.duke.edu/~lcarin/smith04_hsvi_POMDP.pdf`.

E. J. Sondik. *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, Stanford, CA, 1971.

M. T. J. Spaan and N. Vlassis. A point-based POMDP algorithm for robot planning. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, volume 3, pages 2399–2404, New Orleans, Louisiana, 2004. doi: 10.1109/ROBOT.2004.1307420. URL `http://users.isr.ist.utl.pt/~mtjspaan/pub/Spaan04icra.pdf`.

M. T. J. Spaan and N. Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220, 2005. URL `http://staff.science.uva.nl/~mtjspaan/pub/Spaan05jair.pdf`.

Sun Microsystems. Java standard edition, 2008. URL `http://java.sun.com/`.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998. ISBN 0262193981. URL `http://www.cs.ualberta.ca/~sutton/book/the-book.html`.

C. Szepesvári and M. L. Littman. Generalized Markov decision processes: Dynamic-programming and reinforcement-learning algorithms. Technical report, Brown University, Providence, RI, USA, 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.5887&rep=rep1&type=pdf`.

The MathWorks Incorporated. Matlab, 2008. URL `http://www.mathworks.com/products/matlab/`.

S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. The MIT Press, Cambridge, Massachusetts, 2005. ISBN 978-0-262-20162-9.

N. Vlassis and M. T. J. Spaan. A fast point-based algorithm for POMDPs. In A. Nowe T. Lenaerts and K. Steenhou, editors, *Proceedings of the 13th Belgian-Dutch Conference on Machine Learning*, pages 170–176, Brussels, Belgium, January 2004. URL `http://www.dpem.tuc.gr/users/vlassis/publications/Vlassis04benelearn.pdf`.

R. J. Webster III, J. Memisevic, and A. M. Okamura. Design considerations for robotic needle steering. In *Proceedings of the IEEE/RSJ International Joint Conference on Robotics and Automation (ICRA)*, pages 3588–3594, Barcelona, Spain, April 2005. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1570666`.

N. L. Zhang and W. Liu. Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Hong Kong University of Science and Technology, 1996. URL `http://citeseer.ist.psu.edu/zhang96planning.html`.

N. L. Zhang and W. Zhang. Speeding up the convergence of value iteration in partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 14: 29–51, 2001. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.6282`.

R. Zhou and E. A. Hansen. An improved grid-based approximation algorithm for POMDPs. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 707–716, Seattle, Washington, August 2001. ISBN 1-55860-777-3. URL `http://www.cse.msstate.edu/~hansen/papers/ijcai01.pdf`.

R. Zhou and E. A. Hansen. Memory-bounded A* graph search. In *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference*, pages 203–209. AAAI Press, 2002. ISBN 1-57735-141-X. URL `http://www.aaai.org/Papers/FLAIRS/2002/FLAIRS02-041.pdf`.

$$15 = 15 \ \checkmark$$