MONITORING PLAN EXECUTION IN PARTIALLY OBSERVABLE STOCHASTIC WORLDS

by

MINLUE WANG

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

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
January 2014

ABSTRACT

This thesis presents two novel algorithms for monitoring plan execution in stochastic partially observable environments. The problems can be naturally formulated as partially-observable Markov decision processes (POMDPs). Exact solutions of POMDP problems are difficult to find due to the computational complexity, so many approximate solutions are proposed instead. These POMDP solvers tend to generate an approximate policy at planning time and execute the policy without any change at run time. Our approaches will monitor the execution of the initial approximate policy and perform plan modification procedure to improve the policy's quality at run time.

This thesis considers two types of approximate POMDP solvers. One is a translation-based POMDP solver which converts a subclass of POMDP, called quasi-deterministic POMDP (QDET-POMDP) problems into classical planning problems or Markov decision processes (MDPs). The resulting approximate solution is either a contingency plan or an MDP policy that requires full observability of the world at run time. The other is a point-based POMDP solver which generates an approximate policy by utilizing sampling techniques. Study of the algorithms in simulation has shown that our execution monitoring approaches can improve the approximate POMDP solvers overall performance in terms of plan quality, plan generation time and plan execution time.

ACKNOWLEDGMENTS

I would like to give my sincere thanks to my supervisor Richard Dearden for his continuous support and insights along the way over the last few years. Without his guidance and help this thesis would not have been possible. Some of the work in this thesis has been a collaboration between myself and Richard, and so I have used the word "we" throughout since the ideas and solutions have been contributed by both.

I would like to thank my thesis committee member, Professor Ela Claridge and Behzad Bordbar who have always provided me with constructive criticism and encouragement over the last four years.

Many thanks to all IRLab members, especially Professor Aaron Sloman and Nick Hawes for their great feedback on my research and study.

In addition, a thank you to my office mates: Quratul-ain Mahesar, Sarah Al-Azzani, and Mark Rowan who made my journey as a research student pleasant.

Finally, I am really thankful to my Mum for her constant support throughout my life.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

- Minlue Wang, Sebastien Canu, Richard Dearden. Improving Robot Plans for Information Gathering Tasks through Execution Monitoring. Proceedings of International Conference on Intelligent Robots and Systems (IROS), 2013
- Minlue Wang and Richard Dearden. Run-Time Improvement of Point-Based POMDP Policies. Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), 2013.
- Richard Dearden and Minlue Wang. Execution Monitoring to Improve Plans with Information Gathering. Proceedings of the 30th Workshop of the UK Planning And Scheduling Special Interest Group (PlanSIG), 2012.
- Minlue Wang and Richard Dearden. Improving Point-Based POMDP Policies at Run-Time. Proceedings of the 30th Workshop of the UK Planning And Scheduling Special Interest Group (PlanSIG), 2012.
- Minlue Wang, Richard Dearden. Planning with State Uncertainty via Contingency Planning and Execution Monitoring. The Ninth Symposium on Abstraction, Reformulation and Approximation, 2011.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

Planning is the task of coming up with a sequence of actions for an agent to execute in order to achieve certain goals in the environment [91]. Planning domains can usually be divided into classical domains and non-classical domains. Classical planning (shown in Figure 1) assumes no observability of the world, deterministic actions and a static environment (complete model). A static environment does not mean the environment is static but means the planning domain will capture all the information about how the world changes so things will always evolve as we expect [1]. On the other hand, non-classical domains (displayed in Figure 2) require the relaxation of at least one of these assumptions, for example they might include stochastic actions where actions can have multiple outcomes, imperfect information about the world (noisy observation actions) [2] or a dynamic environment (incomplete model) where exogenous events or actions might occur at any time. In particular, in the context of a dynamic environment, the agent could end up with a total unexpected situation at run-time, for instance, actions in the plan do not produce any anticipated effects as modelled in the domains. There are also other assumptions in classical planning that could be relaxed, such as one action at a time, instantaneous actions, discrete states and so on.

In terms of planning algorithms, there are two main categories. One is called off-line planning which generates a full plan before

1 This is not be mistaken with the notion of a static property which refers to a domain property that does not change over time

2 We use observation actions to represent sensing actions or knowledge gathering actions throughout the thesis

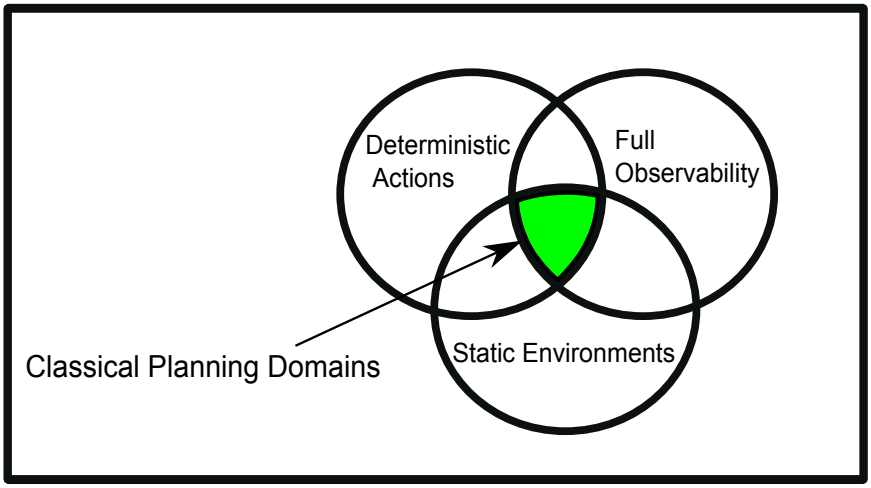

Figure 1: Classical planning domains

executing it, and the other is on-line planning which usually computes the current best action for every single plan step at run-time. Off-line planning algorithms work well for classical domains because things can be observed completely in the world and always turn out as expected. Once a plan is generated by an off-line planner, it can be executed all the way to achieve the goal without any monitoring in classical domains. However, this does not hold for non-classical domains for two reasons. The first one is the environment can be dynamic so exogenous events or actions which are not considered before could occur at any time during the plan execution phase. The second reason is that while the planning problems are becoming more and more challenging, optimal solutions for large domains are difficult to find. Therefore, only approximate solutions are provided at the off-line stage. Both reasons raise the importance of monitoring the execution of a plan. In order to deal with a dynamic environment, an execution monitoring module is required at run-time to detect any unexpected situations and also to try to recover from them. As for the problem of approximate solutions, we do not face the dynamic environment (model is complete), but seek to improve the initial approximate plans at run-time using plan modification techniques. On-line

planning algorithms are designed to make the agent more reactive to the dynamic change of the world since plans are computed on the fly. An on-line algorithm computes a best action for current belief state for each time step [89]. Two simple procedures are performed in order to find the action. The first procedure is building a tree of reachable belief states from the current belief state and the second step is estimating the value of the current belief state by propagating the values from the fringe nodes all the way to the root nodes. However, in practice, there are usually computational and time constraints at plan execution so the on-line algorithms can not expand the tree fully to search the best action. For instance, if there is one second time limit for generating an action at each step, on-line algorithms might not be able to return optimal actions for some large planning problems. Therefore, in this thesis we are interested in how to improve off-line solvers at run-time and also compare these with on-line algorithms.

This thesis examines execution monitoring that works on the off-line planning caused by the second reason mentioned above. In particular, we define *execution monitoring* as follows:

Definition 1 (Execution Monitoring)**.** *Execution monitoring is a continuous process of checking the execution of the plan which involves comparing the future steps of the plans with the current state estimation and repairing the plans if necessary.*

We consider non-classical planning domains but assume the planning model is complete so no additional exogenous events happen at run-time. Given these assumptions, we claim that it is more efficient in many domains to generate approximate policies off-line, but improve them at run-time using execution monitoring and plan repair techniques.

Our novel execution monitoring approaches presented in this thesis aim to improve the approximate solutions generated at planning

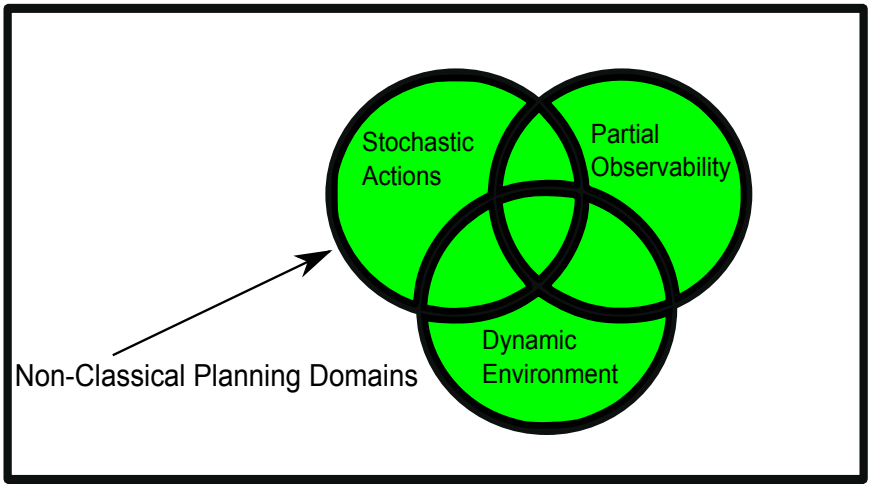

Figure 2: Non-classical planning domains

stage in an on-line fashion so that overall performance can be improved. In order to this, two research questions need to be answered. The first one is when should we decide to modify the original approximate solutions at run-time. Even if we have a mechanism to improve the plan's quality at each modification step, it is unrealistic to repair the original plan for all the steps at run-time because this would result in a massive increase in computational cost. On the other hand, never triggering our execution monitoring module would make the final performance of the algorithms the same as the original ones. Therefore, finding an appropriate monitoring approach to trigger our plan repair procedure plays a crucial part in this work. The second research question is how to repair the approximate solutions when we decide this is necessary. Replanning from scratch will be very time consuming and also means the initial approximate solution will be abandoned completely. The work presented in this thesis will increase the initial plan's final performance while preserving most of its structure.

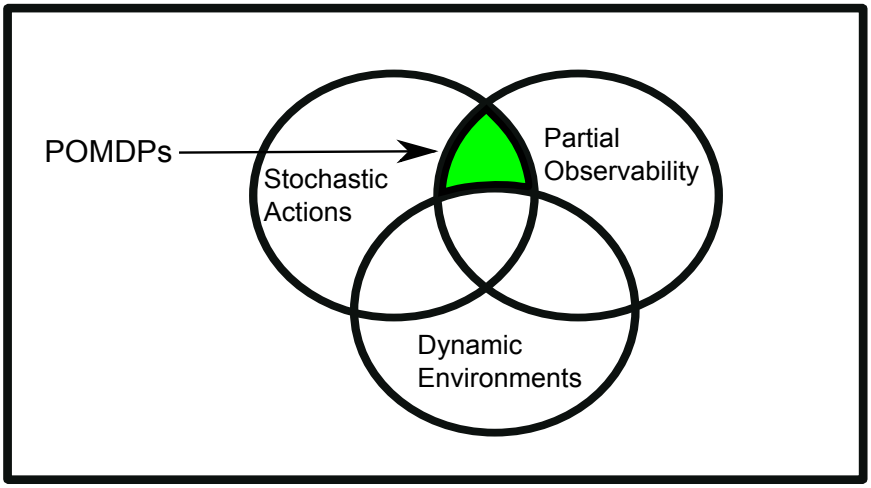

Figure 3: POMDP domains will include stochastic actions and partial observability but no dynamic environments.

1.1 PROBLEM OVERVIEW

As mentioned earlier, classical planning domains do not take into account the uncertainty of the action's outcomes, the observations or the dynamics of the environment. In order to make the domain more realistic for an intelligent agent to execute, it has to incorporate different types of uncertainty. Partially observable Markov decision processes (POMDPs) [104, 100] provide a mathematical framework for representing such planning problems. POMDPs have been widely investigated in many research communities, such as operations research [104], artificial intelligence [18] and robotics [83], with many applications including robot navigation [99] and autonomous underwater vehicle (AUV) [93]. As shown in Figure 3, POMDPs can capture the uncertainty in the initial world states, in action outcomes and in observations. One thing worth noting here is that they assume a static environment so the models have captured all the uncertainties in the problems. Because of the stochastic actions and noisy observations, the agent is no longer sure about the consequence of an action and the current state of the world at run-time. It needs to reason with

this uncertainty in order to successfully complete a task. In a POMDP model, there is a matrix that specifies the stochastic outcomes for each action and a matrix that specifies the uncertainty of the observations. A reward will be assigned at each time step according to the current state and the current selected action. A more detailed description of POMDPs will be represented in Section 2.3.2. The goal of POMDPs is to compute a sequence of actions that can maximize the accumulated reward. A discount factor is also used to get the agent to prefer collecting rewards as early as possible. We classify these planning domains as reward-based problems which differ from classical planning domains (goal-oriental) which usually measure a plan's quality by looking at whether the goal states are achieved or not. Reward-based domains provide a standard and numerical way of evaluating a plan's quality and are used as one of the metrics in our experiments. However, as mentioned in [77], finite-horizon POMDPs are PSPACE-complete, so finding exact solutions for large POMDPs is intractable because of their computational complexity.

Nowadays, engineers from robotics are trying to make low-level state-changing actions more and more reliable. However, as mentioned in [105], some planning problems are still hard because different parts of the environment appear similar to the sensor system of the robot. For example, suppose an office robot is given the task of delivering mail to a destination, navigation in a known environment is easy to accomplish but it needs to determine the correct object first given noisy vision operators. Following Besse and Chaib-draa [6], we use the term *quasi-deterministic partially observable Markov decision problems* (QDET-POMDPs) to describe this interesting class of domains, which differs from *deterministic partially observable Markov decision problems* (DET-POMDPs) [9] in that they allow uncertainty in the observation models of the actions (DET-POMDPs are entirely determin-

istic apart from the initial state). Although QDET-POMDPs are also PSPACE-complete [6], they should be treated differently from general POMDPs because all the state-changing actions are deterministic and the uncertainty of the domains only comes from the observation actions and the initial state. In this thesis, we apply a classical planner FF [51] and a Markov decision process (MDP) solver SPUDD [48] to generate the initial approximate solutions. Since FF and SPUDD are both introduced in order to tackle the domains with no observability, the QDET-POMDPs domains firstly need to be translated into the domains that FF and SPUDD can solve. These solvers will not generate optimal policy for the QDET-POMDPs domains and the approximate policy assumes complete knowledge of the world during the execution time. So we can improve the performance of these approximate solutions by using execution monitoring approaches at run-time.

As for generic POMDPs, we are investigating point-based POMDP algorithms (see Section 6.2 for a survey of point-based algorithms) which have been demonstrated as able to successfully tackle large POMDP domains [103, 61]. Point-based POMDP algorithms search for optimal solutions in a subset of the belief space and expect this approximate policy to work for all the belief points they encounter at execution time. However point-based solvers will not generate policies for those belief points with low transition probabilities and thus result in poor performance when they actually find themselves in those belief points. Therefore we can include execution monitoring at run-time to detect these situations and repair the original policies accordingly.

A diagram of our execution monitoring approaches is displayed in Figure 4. Both execution monitoring approaches aim to improve the approximate solutions at execution time if it is decided that current plans are not good enough for the current situation. It is also worth

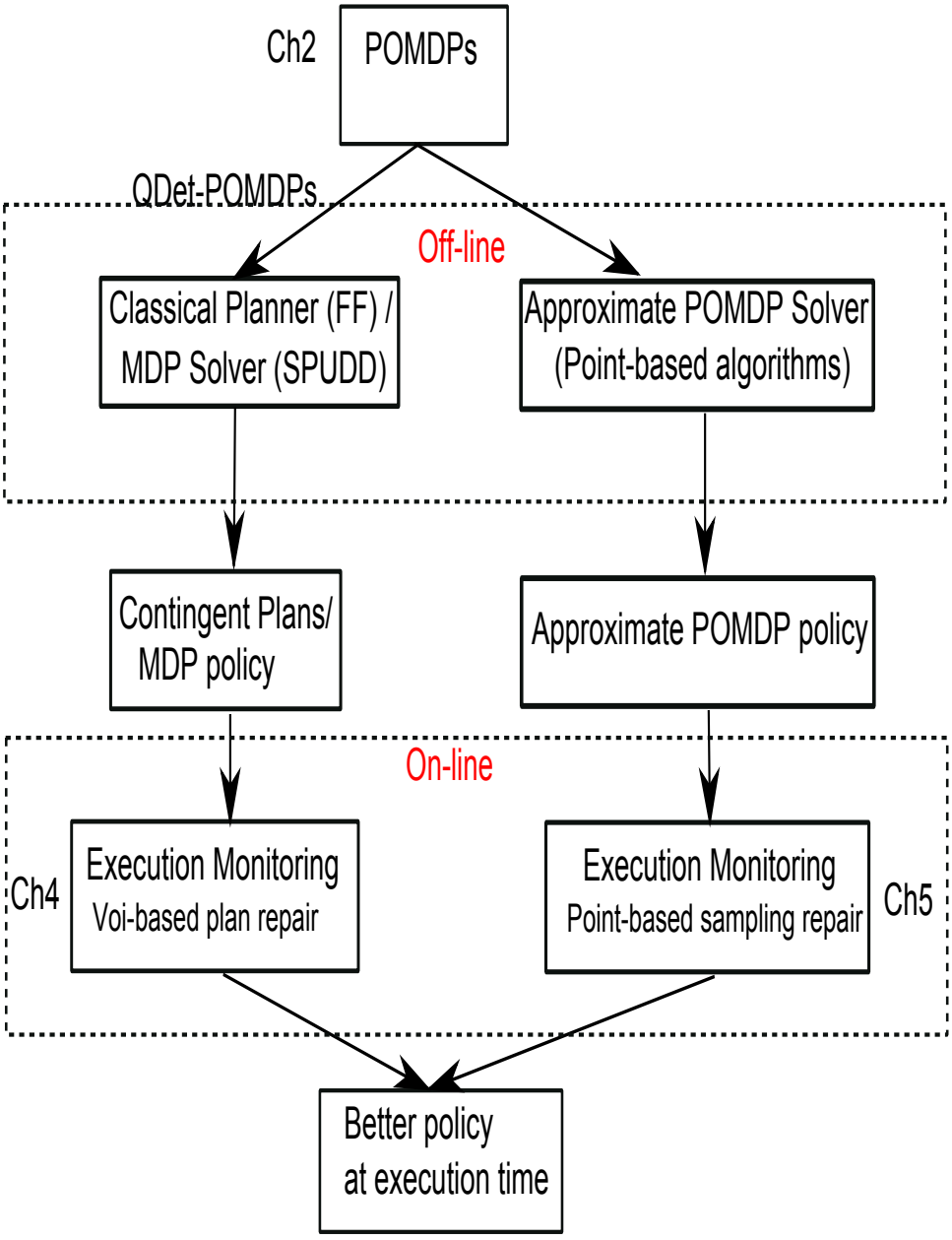

Figure 4: Thesis Structure

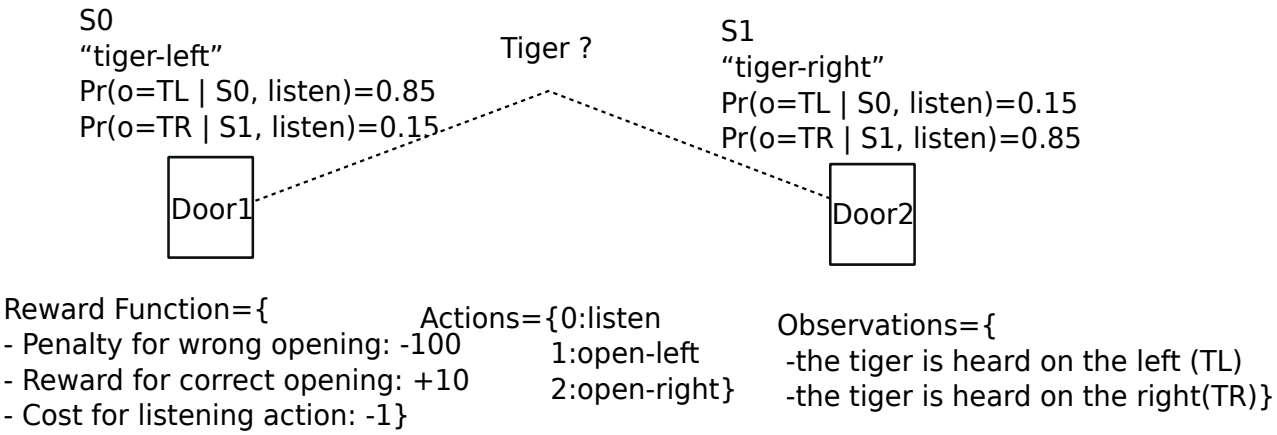

Figure 5: Tiger problems. State space includes tiger-left (S_0) and tiger-right (S_1). Observation space includes hear-left (TL)and hear-right (TR).

noting here that most of the execution monitoring approaches in the literature (Chapter 3) work on goal-oriental planning domains [34, 112] while execution monitoring techniques in this thesis work on reward-based planning domains.

1.2 SOLUTION OVERVIEW

1.2.1 *Execution Monitoring on Quasi-Deterministic POMDPs*

Two translation-based QDET-POMDPs solvers are proposed in this thesis. One uses the classical planner FF to generate a contingency plan which is a branching tree. Different branch plans are followed depending on the outcomes of observation actions. This requires the ability of knowing the exact state of the world during execution time so that the appropriate plan branch can be chosen at run-time. However, due to the nature of POMDPs, observation actions are noisy so no discrete state of the world will be observed directly. In POMDPs, a *belief state* is defined to summarize all the past information including the history of the actions and the observations. The belief state itself is a probability distribution over all discrete states.

As an example, let us look at a tiger problem [18]. In the tiger domain (as shown in Figure 5), a person is asked to open a door which the tiger is not behind. The state-changing actions are opening either the left or right door. The observation-making actions are listening to one of the doors in order to detect the existence of a tiger. If the tiger is actually behind the door and you choose to open it, a large penalty (-100) will be given and vice-versa a positive reward (10) will be assigned if you open the door which the tiger is not behind. The observation-making action *listen* is noisy, as you can see from Figure 5. If the tiger is actually behind the left door (state S_0), the probability of getting correct observation (TL) is 0.85. The person never knows the current state of the world (S_0 or S_1), and he only maintains a belief state which is a probability distribution over the state space. Therefore, the main question from the tiger domain is how many times the listening actions need to be performed so that we believe that the tiger is either behind the door or not behind it. This small example illustrates the same problem we would like to solve by using execution monitoring methods on contingency plans for QDET-POMDPs. As said before, the contingency plan needs to have perfect information about current state of the world in order to select appropriate plan branch at run-time, our execution monitoring will decide how many times the observation actions need to be executed at each branch point in order to gain enough information about the world. Again, the number of times the observation actions need to be executed at each branch point plays a crucial part in getting a good performance from our approach. A value of information approach is then applied to compare the value improvement of executing the observation action with the value of not doing this observation at all. As long as this net value is greater than the cost of the observation action, we will continue executing observation actions. One thing worth noting

here is that our execution monitoring is operating on a belief state, which will be updated after every action and observation iteration. Once we decide there is no need to perform the observation actions at the branch point, the best branch plan will be selected according to the updated belief state and the next value of information calculation procedure will be triggered when we encounter another branch point in the plans. Related to the research questions mentioned earlier, the monitoring procedure is mainly about maintaining a belief state of the world based on the initial state, the history of the actions taken and the observations received. The plan repair procedure is triggered automatically when the next action is an observation action and the value information approach is used as the core of a plan repair procedure at execution time.

Another similar translation scheme is done by converting the QDET-POMDP into an MDP. This can be seen as a variant of the previous FF approach. Instead of generating a contingency plan in the first place, we use the MDP solver SPUDD to generate an initial policy which will map each state in the world into an action. This idea of solving POMDPs using an MDP solver was originally proposed in QMDP algorithms [18] where the state of the world is assumed to be completely observable after the first action is taken which means all sensing actions become uninteresting so no observation actions will be included in the policy. This is the reason why QMDP would perform poorly in domains where observation actions are needed to gather information such as the tiger problem we described above. Our MDP approach differs from QMDP in its way of modelling observation actions and initial state, so that we can maintain as many of the characteristics of the POMDP as possible. Our translation setting will force the MDP solver to include observation actions in the policy so that it can be improved at run-time. The execution monitoring mod-

ule on the MDP policy is similar to the one presented before on the contingency plan except that a complete contingency plan is replaced with a policy on the state space. This MDP translation scheme is more expensive because the MDP solver needs to plan for all the states in the domain. However, this gives us opportunities to modify the initial plan more aggressively in order to get a better performance. Imagine that our observation actions need to be executed after certain set up actions, such as camera calibration for image taking actions. The execution monitoring approach described before will only be concerned with the number of times observation actions are executed, while in this case a better plan might insert certain set-up actions before we actually execute the observation action. These insertions will make the rest of the contingency plan invalid but will not affect our policy execution since a policy is already covering the space over the entire state space. Therefore, execution monitoring with macro-actions is proposed (in this work) to allow the inserting of state-changing actions in the branch points.

1.2.2 *Execution monitoring on generic POMDPs*

The execution monitoring approaches described above work for a sub-class of general POMDPs. The execution monitoring approach we consider here works on the policy generated by POMDP solvers, point-based algorithms. This approach exploits the fact that point-based POMDPs algorithms only compute optimal policies for belief points with high probabilities but ignore unlikely belief regions. At run-time, we use heuristics to estimate when we may have entered a belief state for which the existing policy will perform poorly. We propose and evaluate a variety of heuristics for this. Unlike the previous execution monitoring approach on QDET-POMDPs where as soon

as we encounter an observation action the plan repair procedure is triggered, observation actions are not longer our automatic triggering points. When the heuristic function indicates the policy may be poor, we re-run the point-based algorithm for a small number of additional sampled points to improve the policy around the current belief point. These additional belief points are added to the overall point-based policy so they can be reused in future. Although exact backups are computationally expensive at run-time [45], only by performing plan-repair using heuristics can we require significantly less execution time compared with on-line POMDP solvers which compute the current best action at every time step.

1.3 CONTRIBUTIONS

The major contributions of this thesis are as follows

- Two translation-based approaches to solve QDET-POMDP. The methods generate contingency plans or MDP policies based on the relaxed domains where states of the world are assumed completely observable at run-time.
- A novel execution monitoring approach which works on approximate solutions generated by translation-based QDET-POMDP solvers. The monitoring approach improves the approximate solutions at execution time by inserting relevant actions.
- A comparison of the performance between the translation-based QDET-POMDP solvers and state-of-art POMDP solvers with a range of different benchmarks. It is shown in Chapter 4 that our translation-based approaches with additional execution monitoring mechanism require much less plan generation time com-

pared to a standard POMDP solver symbolic Perseus [83] and provide better plans compared to translation-based solvers alone.

- A novel execution monitoring approach which works on point-based POMDPs algorithms. The key contribution here is proposing several heuristic functions to detect the situation at runtime where the current approximate policy is not good enough for the current belief point. Results from Chapter 5 demonstrate that our execution monitoring on point-based policies out-performs point-based algorithms without any monitoring in terms of the total reward. It works especially well on the domains where low transition probability states exist, such as a factory domain where each component can have a low probability of becoming faulty when the product is being assembled. Comparison is also done on standard POMDP benchmarks.

1.4 THESIS STRUCTURE

The remainder of this thesis is structured as follows. Chapter 2 reviews a variety of planning algorithms such as classical planning, state-space planning, partial-order planning, contingency planning, MDPs and POMDPs. Discussion of value iteration algorithms for computing exact solutions for MDPs and POMDPs is also presented in Chapter 2. A survey of existing execution monitoring approaches from several research communities is given in Chapter 3. Most of the execution monitoring approaches displayed in Chapter 3 take into account the agent's planning information rather than examining the state of individual physical components in the system. Chapter 4 introduces the problem of solving Quasi-deterministic POMDPs and explains the translation-based approaches with value of information execution monitoring module. Chapter 5 focuses on general

POMDPs which relax the assumptions of state-changing actions being deterministic in QDET-POMDP models. Execution monitoring on point-based POMDP algorithms is shown in this chapter followed by systematic evaluation of different heuristic functions for deciding the time of plan repair. Related work on execution monitoring of QDET-POMDP models and general POMDP models are discussed in Chapter 6 including similarities and differences among a variety of point-based algorithms. Finally Chapter 7 concludes the thesis with an overall summary of this work and discusses possible directions for future research.

2

BACKGROUND ON PLANNING ALGORITHMS

2.1 INTRODUCTION

Planning is the task of coming up with a sequence of actions for an agent to execute in order to achieve certain goals in the environment. To do so, a planning domain that describes the dynamic of the world needs to be given in the first instance. Since in reality different problems can have a variety characteristics, many planning domains are proposed to capture these properties. The varieties of planning domains can exist in many aspects. Depending on the outcomes of an action, planning domains can be classified into deterministic domains, non-deterministic domains and stochastic domains. Deterministic domains require all the actions in the domain to have only one outcome if the actions are applicable. On the other hand, actions in non-deterministic domains [2, 25] cannot predict which effect is going to occur before execution. Stochastic domains not only represent actions with non-deterministic effects but also use probabilities for each effect. Another classification of planning domains is done by observability. Full observability gives you complete access to the world, while no observability means there is no knowledge about the state of the world at any given time. In partial observability domains, either only part of the domains can be directly observed or the observation actions are noisy so that the world is not accurately observed. In terms of the goal representation, domains which need to find the actions that will lead from the current initial state to the goal states

are often called goal-directed problems. In goal-directed problems, the correctness of a plan means the goal will be satisfied if the plan directs its execution to stop and the completeness of a plan means it can account for all possible situations in the world [65]. In decision-theoretic planning (MDPs or POMDPs), an optimal policy (mapping states to actions) usually needs to be found to maximise an accumulated discounted reward. As said before, we classify this type of planning domain as a reward-based problem. Planning domains can also be divided into concurrent or non-concurrent categories according to whether the actions can be executed in parallel or not. In particular, the domains with concurrency often need to specify the duration time of actions, while in other cases, actions can be executed instantaneously. Planning domains with continuous state variables also need to be treated differently from the domains with only discrete variables. In the end, most of the planning domains assume a complete model of the problem so no exogenous events will occur at execution time which is also referred to as a static environment, while a dynamic environment can result in unexpected situations happening at any time during plan execution.

Given different assumptions about the world in the planning domain, different planning algorithms have been developed to tackle these problems. In the early stage of planning development, due to computational reasons, the world is assumed fully observable and actions can only have deterministic effects. We often refer to these discrete problems with deterministic actions, no observation actions and no concurrency as classical planning. The reason why observation ability is not needed in classical planning is that it assumes the agent already has complete information about the world. This is often referred to as close world assumption [85]. For example in STRIPS representation, the stored predicates are assumed to have truth value,

while the ones which are not stored are assumed to be false. Later on, a desire to solve more realistic problems led to relaxing some of these assumptions. One direction is assuming the agent only has an incomplete knowledge about the world. There are two main approaches to deal with the problems with incomplete information. One approach is contingency planning [84, 49] where observation actions are available to sense the world and contingency plans are branching plans where each plan branch corresponds to one specific outcome of the observation action [17]. Although we cannot predict which outcome is going to occur prior to execution of the action, if the world is fully observable, we will know exactly which outcome will happen after execution, so the appropriate branch plan can be executed. Some contingency planning problems have partial observability so only a certain part of the world is observable. The other approach is conformant planning [101, 10] where the agent has no observation actions. There are several possible initial states that the agent can start with and this uncertainty can not be resolved either at planning stage or execution stage because there areno observation actions. Therefore, the goal of conformant planning is to search for a sequence of actions that can achieve the goal from any initial state [10]. Actions in contingency planning and conformant planning can be either non-deterministic or stochastic depending on whether the actions are assigned probabilities. Decision-theoretic planners, such as Markov decision process (MDP) or partial observability Markov decision process (POMDP) solvers, have also been developed independently in the operations research community and have drawn a great deal of attention in the planning community in the last few decades [104]. MDP and POMDP both assume the world has stochastic outcomes. The difference between the two is that POMDPs also assume imperfect observation, which means the observation actions reveal the true

Planning	Initial State	Actions	Observability
STRIPS, FF	Known	Deterministic	Full
Partial Order Planning	Known	Deterministic	Full
Contingency Planning	Known or Unknown	Stochastic or Non-deterministic	Full or Partial
Conformant Planning	Unknown	Stochastic or Non-deterministic	No
MDP	Known	Stochastic	Full
POMDP	Unknown	Stochastic	Partial

Table 1: Differences of varieties of planning algorithms

state of the world with pre-defined noises. The policies generated by MDP solvers are similar to the contingency plans which also have branches depending on the outcomes of the action. However, a policy can map any discrete state in the world into an action while contingency plan only accounts for the current initial state and needs to re-plan if the initial state is changed. The differences between planning algorithms are shown in Table 1.

In this thesis, we are interested in the observation problems where the world can not be accurately observed. Although there are observation actions available in the domains, we do not know the current discrete state before or even after the execution of observation action. This is the reason why the plans can benefit from our execution monitoring approaches at run-time. Since POMDP provides a mathematical framework for presenting partial observable problems, we will use POMDP domains as illustrated examples through out this thesis.

2.2 CLASSICAL PLANNING

Let us look at a blocks-world example from classical planning. An initial state and a goal state of the example are shown in Figure 6. In

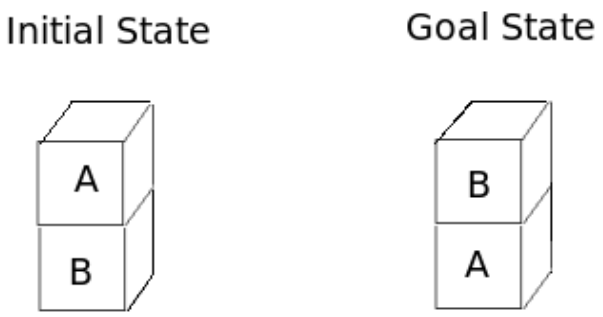

Figure 6: A blocks-world example

this blocks-world example, the task is changing the position of the two blocks on the table. Suppose a robot has four actions available to it, namely *PickUp(x)*, *PutDown(x)*, *Stack(x, y)*, and *Unstack(x, y)*. The *PickUp(x)* action picks up a block x from a table as long as the arm is not holding another block; The *PutDown(x)* action puts down a block x on the table; *Stack(x, y)* puts a block x on the top of block y; *Unstack(x, y)* takes a block x away from the top of block y. The problem is to find a sequence of actions to achieve the goal state from the starting state. As mentioned earlier, original classical planning works on the domains with deterministic actions and full observability so we know exactly where each block is at any given time and the four actions will only have the expected outcome without considering the action's failure or other unexpected situation.

2.2.1 *State-Space Planners*

The Stanford Research Institute Problem Solver (STRIPS) [33] was introduced to solve the classical problems using search techniques in state space. Prior to that, most of planning systems were using

first-order logic to represent the world [87], such as situation calculus [68]. The notations of original STRIPS planning are as follows [33]:

Definition 2 (Planning task). *A planning task* P *is a triple* $\langle A, I, G \rangle$ *where* A *is the set of actions,* I *is the initial state and* G *is the goal states.*

We assume the world state is encoded with a set of propositions. Since there is no uncertainty in the initial state in the original STRIPS representation, the initial state is assumed to be fully known at the first instance.

Definition 3 (State). *A state* s *is a set of propositions.*

Definition 4 (Action). *A STRIPS action* a *is a pair* $(\mathtt{pre}(a), \mathtt{effect}(a))$ *where* $\mathtt{pre}(a)$ *are the preconditions of action* a *and* $\mathtt{effect}(a)$ *are the resulting effects of executing* a. $\mathtt{Effect}(a)$ *is also a pair* $(\mathtt{add}(a), \mathtt{del}(a))$ *where* $\mathtt{add}(a)$ *and* $\mathtt{del}(a)$ *are the adding list and deleting list of action* a *respectively.*

An action is applicable *in state* S *if* $\mathtt{pre}(a) \subseteq S$ *and the resulting new state* $S' = a(S) = S \cup \mathtt{add}(a) \setminus \mathtt{del}(a)$.

In the original STRIPS representation, actions are deterministic so effects of the action will always occur. Representations of extended version of STRIPS actions with conditional effects will be given later on.

Definition 5 (Plan). *Given a planning task* $P = \langle A, I, G \rangle$. *A plan is an action sequence* $a_1, a_2, \ldots a_n$ *that solves the task if* $G \subseteq a_n(\ldots a_2(a_1(I)))$.

Take the blocks-world in Figure 6 for example, the initial state is $\mathtt{OnTable}(B) \wedge \mathtt{On}(A, B) \wedge \mathtt{Clear}(A) \wedge \mathtt{HandEmpty}()$ and goal state is $\mathtt{OnTable}(A) \wedge \mathtt{On}(B, A) \wedge \mathtt{Clear}(B) \wedge \mathtt{HandEmpty}()$. Table 2 shows preconditions and effects of the action PickUp in this example.

The planning in STRIPS is done by maintaining truth values of predicates which are used to perform backward search from goal

Preconditions	Clear(x) OnTable (x) HandEmpty( )
Postconditions	**Add list:** Holds(x)
	Delete list: HandEmpty( ) OnTable(x) Clear(x)

Table 2: Preconditions and Postconditions of action PickUp(x)

states. The plan generated by STRIPS is a *straight-line plan*, for example STRIPS might output a plan as:

$\{\mathtt{Unstack}(A, B), \mathtt{PutDown}(A), \mathtt{PickUp}(B), \mathtt{Stack}(B, A)\}$

for the illustrated blocks-world example. In the next Chapter 3, we show how a system called PLANEX [34] can monitor the execution of the STRIPS straight-line plans in order to deal with non-deterministic actions and dynamic environment. In particularly, we show how PLANEX can make use of the representation of preconditions and effects of theSTRIPS action.

In this thesis, we use PDDL [69, 70] which is a Planning Domain Description Language released in 1998 by the planning community to represent the classical domains. As said in [36], although PDDL was largely inspired by STRIPS formulations, it extended STRIPS to a more expressive language, such as ability to express a type structure for the objects, actions with negative preconditions and the parameters in the actions and the predicates. For instance, the *PickUp* action from blocksworld domain can be written in PDDL as follows:

```
(:action PickUp
 :parameters
  ( Object ?x)
 :preconditions
   (and  (OnTable ?x)
         (Clear ?x)
         (HandEmpty ))
 :effect
```

```
   (and (Hold ?x)
        (not (HandEmpty ))
        (not (OnTable ?x))
        (not (Clear ?x))
    )
)
```

where ?x is object parameter of the *PickUp* action.

One thing worth noting here is that a classical planner called FF (Fast-Forward) [51] will be used to generate classical plans later on. FF has shown great success in AIPS-2000 planning competition [51] and has also been extended to tackle non-classical planning problems [49, 116, 50]. FF utilizes a heuristic function which can be derived from the planning domain and performs forward search in the state space. The heuristic function itself can be computed from GRAPHPLAN system [7] in a relaxed domain where deleting effects are ignored for each action. Original FF will perform on the problems written in PDDL and generate a straight-line plan as STRIPS does in the end.

2.2.2 *Partial-Order Planners*

Partial order planning (POP) [78, 66], sometimes called "Non-linear Planning", generates plans without fully specifying the order of the actions at planning time. They only consider the orders that are crucial to the execution of the plan. For example, if an action a generates an effect e which is the precondition of an action b, then action a needs to be executed strictly before action b and no other actions between action a and action b can change the value of the effect e. Partial order planning [66] utilizes the idea of "least commitment", so only the most crucial commitments are constructed at planning time. This also makes partial order plans more flexible to be executed at

run-time because more options are available to execute the partial order plans compared to straight-line plans. The commitments for a plan could be the ordering of the actions or variable binding. Most POP algorithms [78, 66] make the same assumptions of STRIPS: deterministic actions, no observability and a static environment. In Chapter 3, an execution monitoring approach for partial order planning will be shown to tackle the problems with a dynamic environment. As PLANEX makes use of the representation of actions in STRIPS, the execution monitoring approach for partial order planning also utilizes the data structure of partial order plans at run-time.

2.2.3 *Conformant Planning*

The approaches to classical planning we have discussed so far assume perfect information about the model, including full knowledge of the world state, actions with deterministic outcomes and a static world. As stated before, in order to solve more realistic problems, people have tried to model the planning problems with uncertainty. One possible direction is *conformant planning* where there is uncertainty in the initial state, but no observability at all in the model. So the problem of conformant planning is how to find a sequence of actions that can achieve the goal without knowing at which initial state the agent is. Conformant Graphplan [101] tackles this problem by creating a different plan graph for each possible world and searches all graphs at the same time. However, since there are several initial states that the agent could be in, an initial belief state b_0 can be used to represent this set of states. The problem then becomes finding a sequence of actions that will map this initial belief state b_0 into a target belief state, Bonet et al. [10] have used this idea to search for the solutions in belief space. Both approaches deal with conformant problems with

non-deterministic actions, Buridan [62] was an early attempt at tackling conformant problems with probabilities. Without considering the cost of the action and the maximization of probability of goal satisfaction, Buridan can generates a partial order plan that is sufficiently likely to satisfy the goals rather than achieving the goals every time.

2.2.4 *Contingency Planners*

The only difference between *contingency planning* and *conformant planning* is that sensory information is available for contingency planning at execution time. In the literature [79], the term *conditional planning* is also used for contingency planning. In this thesis, we define *contingency planning* as follows:

Definition 6 (Contingency planning)**.** *Contingency planning is a planning task where an action can have multiple outcomes and the one that will occur at run-time is unknown at planning time. Contingency planning assumes either full or partial observability in the model where only part of the world can be observed.*

and a contingency plan is defined as follows:

Definition 7 (Contingency plan)**.** *A contingency plan is a plan which usually has branches, where each branch corresponds to one or more possible outcomes of an action, and the branch to execute will be chosen at run-time.*

In general, there are three main problems that need to be considered for contingency planning:

- The first question is how to represent the actions with multiple outcomes. As summarised in [17], one can model the uncertainty of the actions strictly in logic using disjunctions (nondeterministic) and the other approach is modelling the action numerically using probabilities (stochastic).

- Prior to the execution of non-deterministic or stochastic actions, it is not known which outcome will occur. However, since the world can be fully or partially observed, some observation information could be available at execution time in order to choose the appropriate branch plan to follow.

- Contingency planning only considers a number of predicted sources of uncertainties [79], such as actions having multiple outcomes. Unpredicted sources of uncertainty, such as an incomplete model or a dynamic environment needs to be dealt with by execution monitoring at run-time.

Conditional nonlinear planning (CNLP) [79] and Cassandra [84] are two contingency planners that model the uncertainty of the action using disjunctions. However, CNLP assumes full observability and Cassandra assumes partial observability in the domain. CNLP is an extended version of the Systematic Nonlinear Planner (SNLP) [66] by adding sensing operator *observe()* in the domain to observe which outcome occurs at run-time. For example, the action *observe(road(b,s))* has two possible outcomes with the labels $\neg$*clear(b,s)* and *clear(b,s)* to indicate the clearness of road from location b to location s. These labels from sensing actions are called observation labels. CNLP works by attaching reason labels and context labels to all the actions in the plan. Context labels are a set of observations needed for executing the current action and reason labels are the goals that the action aims to achieve. Therefore, appropriate actions can be chosen by matching the observations received so far with the corresponding labels.

Cassandra [84] uses the same syntax as in SNLP where uncertain effects of the actions are represented as *conditional effects* or *secondary preconditions*. As described in [84], conditional effects allow postconditions of actions depending on the context in which the action is executed. Let us look back at the blocks-world example in Section 2.2,

if a successful execution of the *PickUp(x)* action for a robot depends not only on the preconditions *OnTable (x)* $\wedge$ *HandEmpty()* but also on the dryness of the robot's hand *(Dry ?hand)*, a contingency plan that accounts for both events needs to be constructed first. A description of this extended version of *PickUp* action with secondary preconditions written in PDDl is shown as follows:

```
(:action PickUp
 :parameters
  ( Object ?x Object ?hand)
 :preconditions
   (and  (OnTable ?x)
         (HandEmpty ))
 :effect
   (when (Dry ?hand)  \\conditional effects
   (effect  (and (Hold ?x)
            (not (HandEmpty ))
            (not (OnTable ?x))
            (not (Clear ?x))))
   (when (not (Dry ?hand)) \\conditional effects
   (effect (and (not (Hold ?x))
           (HandEmpty )
           (OnTable ?x)
           (Clear ?x)
         )
    )
)
```

where ?hand is the additional object parameter of the *Pickup* action.

The other contribution of Cassandra is the separation of the information gathering process from the decision making process. So one information gathering process might be executed once but serve several decisions. For instance, checking the dryness of a robot's hand once can let the *PickUp* action be executed multiple times (if we assume the hand is always dry afterwards). Once again, the observation model for the sensing actions might make the contingency problem even harder. Suppose observation action *check-hand (h)* does not always return perfect information about the dryness of the hand h, it then becomes difficult to choose which branch to follow as we do

not know the current state of the world at execution time. As stated in the previous chapter, this is exactly the problem having uncertainty in both actions and observations, that we would like to solve. C-Buridan [31] planner, which is an extension of Buridan, has tried to tackle these problems by finding a plan that can succeed with a minimum probability. C-Buridan will generate a partial order plan as Buridan does, the only difference is that the plan generated by C-Buridan includes noisy observation actions while Buridan does not consider any observation actions. In the next section, we demonstrate how decision-theoretical planning can represent this problem and find a policy that can maximise the probability of success. In Chapter 4, an execution monitoring approach for contingency plans will be discussed which aims to improve the quality of the plan in stochastic and noisy observability domains.

2.3 DECISION-THEORETICAL PLANNING

Markov decision processes (MDPs) and partially observable Markov decision processes (POMDPs) have been used widely in AI community to formalise the planning problem in stochastic domains [18].

2.3.1 *MDP*

MDPs can formulate sequential decision making problems with stochastic actions and assume full observability of the model so the agent can know which outcome of the action occurred at run-time and the current state of the world at any time. These assumptions are the same for contingency planning. A policy generated by a MDP solver is also a decision tree where each branch corresponds to one outcome of an action. The major difference between MDP and contingency plan-

ning is that the former tries to generate a policy that can maximise an accumulated reward over a fixed finite period of time or over an infinite horizon while the latter only generates a branching plan that can achieve the goals. In this section, we describe the basic MDP model and consider an exact MDP approach, value iteration.

Formally, an MDP is a tuple $\langle S, A, T, R, \beta \rangle$ where:[53]:

- S is a finite set of environmental states that can be reliably identified by the agent. We assume all states are discrete.
- A is a finite set of actions that the agent can take.
- T is a state transition function that maps $S \times A$ into a probability distribution over states S. $P(s, a, s')$ represents the probability of ending at state s' when the current state is s and action a is taken.
- R is a reward function that is a mapping from $S \times A$ into a real-value reward r. $R(s, a)$ is the immediate reward of taking action a in state s.
- β is a discount factor, where $0 < \beta < 1$.

The objective of MDP planning is finding an optimal policy π^* that maximises the expected long-term total discounted reward over the infinite horizon for each s and is defined as follows:

$$E[\sum_{t=0}^{\infty} \beta^t R(s_t, \pi^*(s_t))]. \tag{1}$$

where s_t is the state of the agent and t is the time step at execution stage.

A policy π is a mapping from any state s in the planning domain into an action a which can be represented as $\pi(s)$. Let $V_\pi(s)$ be the value of executing that policy π starting from state s. The V value

for all states in the domain can be calculated by using the following linear equations:

$$V_\pi(s) = R(s, \pi(s)) + \beta \sum_{s' \in S} P(s, \pi(s), s') V_\pi(s'). \tag{2}$$

The V function can be seen as an evaluation method for a policy. On the other hand, if we know the V values of all the states, a policy can be extracted by using the maximum operator, which is shown as follows:

$$\pi(s) = \arg\max_a [R(s, a) + \beta \sum_{s' \in S} P(s, a, s') V(s')]. \tag{3}$$

[53] has shown that there is a stationary policy π^* and an optimal value function V^* for every starting state in the infinite-horizon discounted case. Finding an optimal policy π^* can now be realized by finding an optimal value function V^*. Value iteration algorithms [5] search the optimal policy by incrementally computing V values. The main idea is that, at each iteration the value function V_t is improved from previous value function V_{t-1} by using the following Equation:

$$V_t(s) = \max_a [R(s, a) + \beta \sum_{s' \in S} P(s, a, s') V_{t-1}(s')]. \tag{4}$$

where t represents the number of iterations at planning stage. This process of computing a new value function from the previous value function is often referred to as *Bellman backup*. One thing worth noting here is that the value iteration algorithm utilizes a function $Q(s, a)$, which takes a state and an action as arguments and represents the

Algorithm 1 Value iteration for MDPs.

For each $s \in S$ $V_0(s) = 0$, $t = 0$
repeat
  for all $s \in S$ **do**
    for all $a \in A$ **do**
      $Q_t(s,a) = R(s,a) + \beta \sum_{s' \in S} P(s,a,s') V_{t-1}(s')$
    end for
    $\pi_t(s) = \arg\max_a Q_t(s,a)$
    $V_t(s) = Q_t(s, \pi_t(s))$
  end for
until $\max_s |V_t(s) - V_{t-1}(s)| < \theta$

value of executing the action a in the state s and then following the current best policy. So the Equation 3 can be rewritten as follows:

$$\pi_t(s) = \arg\max_a Q_t(s,a) \tag{5}$$

The value iteration can be terminated when the maximum difference between the current value functions V_t and the previous value function V_{t-1} is less than a pre-defined threshold θ in order to find a near-optimal policy. The basic value iteration algorithm is shown in Algorithm 1.

One difficulty of using the value iteration algorithm to solve MDP is the need to enumerate all the actions and states as shown in the Bellman backup process (Equation 4), and each iteration requires $|S|^2|A|$ computation time for enumerating the state space. In particular, the size of the state space $|S|$ grows exponentially with the number of domain variables. There has been a great deal of research on developing representational and computational methods for certain types of MDPs [16, 48] which have shown great success in tackling some large MDP problems. The main idea is that by *aggregating* a set of states according to certain state variables, the algorithms can manipulate these abstract-level states in order to avoid the explicit enumeration of the state space. In this thesis, we use an MDP solver called SPUDD

which represents value function and policy with algebraic decision diagram (ADD) [4].

2.3.2 *POMDP*

MDP requires the ability to know the exact current state of the world in order to execute the policy. What if the agent is not fully observing the world? The POMDP framework provides a mathematical framework for representing such planning problems with uncertainty in initial state, the effects of actions and observations. One thing worth noting here is that, there no distinction is made between actions that can change the state of world and the actions that can observe the world in POMDP. All the actions are modelled so that both effects are in standard POMDP domain. This is different from what we have seen in the contingency planner Cassandra where observation-making actions are defined independently from state-changing actions.

Formally, a POMDP is a tuple $\langle S, A, T, \Omega, O, R, \beta \rangle$ where [56, 18]:

- S is the state space of the problem.
- A is the set of actions available to the agent.
- T is the transition function that describes the effects of the actions. We write $P(s, a, s')$ where $s, s' \in S, a \in A$ for the probability that executing action a in state s leaves the system in state s'.
- Ω is the set of possible observations that the agent can make.
- O is the observation function that describes what is observed when an action is performed. We write $P(s, a, s', o)$ where $s, s' \in S, a \in A, o \in \Omega$ for the probability that observation o is seen when action a is executed in state s resulting in state s'.

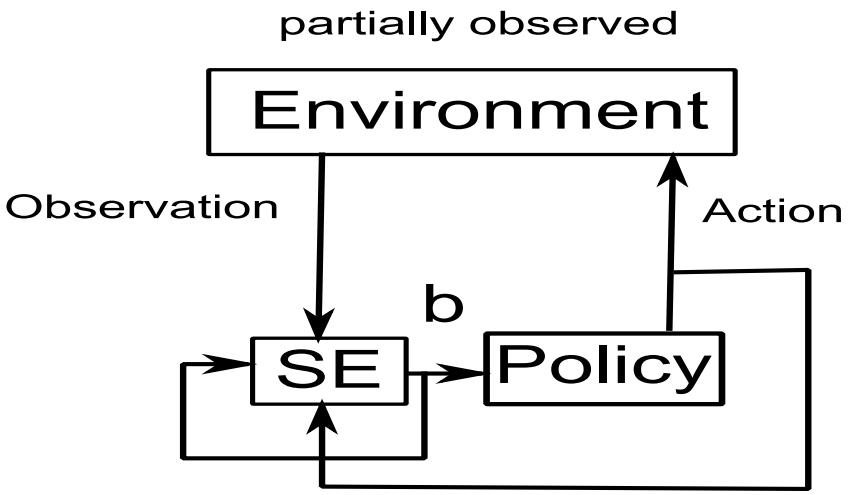

Figure 7: An interactive diagram between an agent that is executing a POMDP policy and an environment. A policy will map each belief state into an action that works on the environment. Once an observation is received, a new belief state will be updated accordingly

- R is the reward function that defines the value to the agent of particular activities. We write $R(s, a)$ where $s \in S, a \in A$ for the reward the agent receives for executing action a in state s.

- β is a discount factor, where $0 < \beta < 1$.

As you can see from the definition of POMDP, state space S, action space A and transition function T are the same as the ones in MDP definition. Additional parameters of POMDPs are observation variable Ω and observation function O which govern the observation model in POMDP.

Since POMDPs do know exactly at which state the agent is , they need to estimate the current state according to the previous experience of the agent. That is, they need to maintain a *belief state*, a distribution over S calculated from the initial belief state and the history of actions and observations. Given this, a policy for a POMDP is a mapping from belief states to actions. The belief state (or sometimes

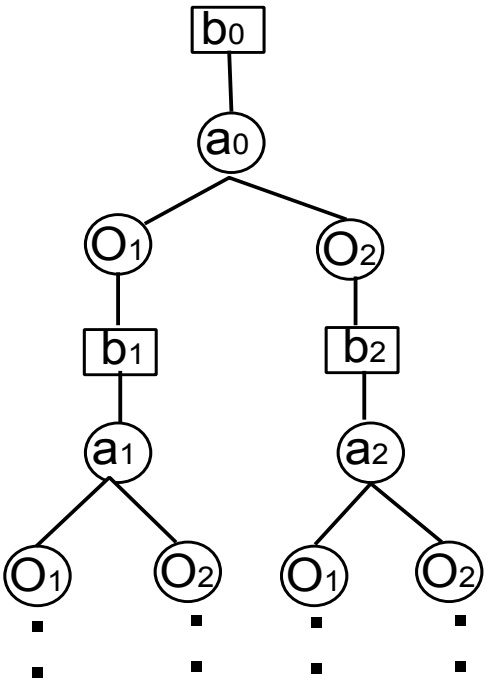

Figure 8: A POMDP policy tree p: the observation space only contains o_1 and o_2, and b_0 is the initial belief state

it is referred as an estimate state) can be computed from the previous belief state b, action a and observation o using Bayes' rule:

$$\begin{aligned} SE_{s'}(a,b,o) &= P(s'|a,b,o) && (6)\\ &= \frac{P(o|s',a,b)P(s'|a,b)}{P(o|a,b)} && (7)\\ &= \frac{P(o|a,s')\sum_{s\in S} P(s'|s,a)b(s)}{P(o|a,b)} && (8) \end{aligned}$$

where $P(o|a,b)$ is a normalisation constant.

A digram of POMDP model is shown in Figure 7 where **SE** stands for *state estimator* which updates belief state according to Equation 6. Because the new belief state b' is deterministic if we know the current executed action a and the observation o, there is only a finite number of possible future belief states which is the number of possible observations we can get after executing an action. A policy tree of POMDP solution is illustrated in Figure 8. As can be seen from the graph, a policy tree p defines a best action a_0 for the initial state b_0

and provides sub-trees associated with possible observations. The execution of this policy tree is similar to the execution of a contingency plan that has observation actions. Both require an appropriate branch plan to be chosen according to the observation outcome received at run-time.

Suppose we have a policy tree p and the agent knows the current state of the world is state s, the expected value of executing this policy tree p can be computed as follows:

$$V_p(s) = R(s, p(s)) + \beta \sum_{s' \in S} P(s, p(s), s') \sum_{o \in \Omega} P(o|s, p(s), s') V_{p_o}(s'). \quad (9)$$

where $p(s)$ defines the action to take when current state is s and $V_{p_o}(s')$ represents the expected value of following policy subtree after observation o.

As mentioned earlier, the agent no longer knows the exact state of the world in POMDP, but only maintains a belief state, so the expected value of executing the policy tree p from current starting belief state b_0 is a linear combination of expected value for all discrete states:

$$V_p(b) = \sum_{s \in S} b(s) V_p(s) \quad (10)$$

Smallwood and Sondick [100] showed that the optimal value function for a POMDP is piecewise linear and convex so it can be represented by a set of $|S|$-dimensional hyperplanes: $\Gamma = \{\alpha_0, \alpha_1, \ldots, \alpha_n\}$. Each hyperplane is often referred to as a α-vector which can map each belief state b in the belief space to a value according to Equation 10. In particular, each α-vector also corresponds to a policy tree, so the current action can be extracted once the best α-vector is found.

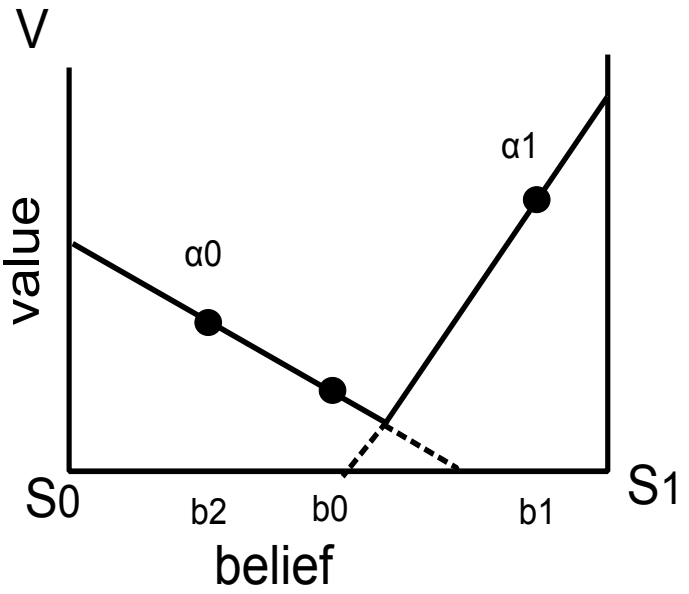

Figure 9: A POMDP policy which contains policy tree α_0 and α_1. α_0 is the current best policy tree for belief point b_2 and b_0. α_1 is the current best policy for belief point b_1.

Algorithm 2 Value iteration for POMDPs.

For each $b \in S$ $V_0(b) = 0$, $t = 0$
repeat
  for all $b \in B$ **do**
    $V_t(b) = \max_{a \in A}[R(b,a) + \beta \sum_{b' \in b} P(b,a,b')V_{t-1}(b')]$
  end for
until $\max_s |V_t(b) - V_{t-1}(b)| < \theta$ for all $b \in B$

The goal of POMDP planning is now to find those α-vectors so that the best policy π^* can be derived as:

$$\pi^*_V(b) = \arg\max_a \alpha_a \cdot b. \tag{11}$$

where each α-vector defines the best policy for the belief points.

For example, suppose we have a POMDP domain which only has two states s_0 and s_1 (Figure 9), α-vectors are lines in 2-dimensional space. As can be seen from Figure 9, current best policy is the upper surface of two α-vectors (policy trees) namely α_1 and α_2, and each α-vector is only accountable for a sub-region of the belief space.

A belief-based discrete-state POMDP can be seen as an MDP with a continuous state space, thus, one of the MDP solvers, value iteration can also be used to solve a POMDP [53]. An algorithm of POMDP value iteration is shown in Algorithm 2. For each iteration, the difficulty of building current value function V_t from previous value function V_{t-1} comes from two aspects: one is the need to consider all the belief points in a continuous space for each iteration while in MDP the number of states are only finite; the other issue is when the previous value function V_{t-1} has $|\Gamma^*_{t-1}|$ vectors, the number of new policy trees is $|A||\Gamma_{t-1}|^{|\Omega|}$ which is exponential in size of the observation space [63]. It has been shown that finding the optimal policy for a finite-horizon POMDP is PSPACE-complete [77]. In Chapter 5, we discuss an approximate POMDP solver point-based algorithm and how the approximate policy can benefit from our execution monitoring approach.

3

BACKGROUND TO EXECUTION MONITORING

In this chapter various execution monitoring approaches from different research communities are surveyed. As mentioned in Chapter 1, execution monitoring is defined as a process of monitoring and modifying plans at run-time by considering the future steps. Much effort has been made in the area of planning, discussed in Chapter 2 for intelligent agents, such as office robots, and autonomous underwater vehicles (AUV). Planning involves choosing a sequence of actions from a planning model in order that the intelligent agent achieves a set of goals. Most of the planning algorithms try to find a complete plan or policy that the agent can follow at execution time. However, in dynamic environment, the agent can encounter differences between the expected and actual context of execution, such as the failure of actions failure or a change in the goal, in these cases, the original plan is not sufficient to achieve the goal. In the context of execution monitoring, we would like to make sure the agent will successfully accomplish its given goals regardless of what changes occur in the world. The term change here means things do not go as we planned; for example, actions do not produce anticipated effects or some goal conditions are changed at run-time.

Chiang et al. [19] define execution monitoring as a system that allows the robot to detect and classify failures, and failure here means execution does not proceed as planned. This definition of execution monitoring should be reformulated as *state estimation* or *fault diagnosis*, since its main objective is to report a failure and possibly find the causes of the failure when a failure occurs at execution time. For ex-

ample, fault diagnosis techniques mentioned in [19] can be applied to detect broken wheels or faulty sensors of an office robot. More details of state estimation or fault diagnosis approaches are discussed in Section 3.2. Execution monitoring in this thesis is more related to a specific high-lever planning system, and it aims to make sure the current plan can achieve its goal in the end or try to gain as much reward as possible from the world with respect to a dynamic environment. Let us look at an example where an office robot has abilities of picking up and putting down an object. The robot's goal is to put an object on a table. A straight-line plan *move(pos-r,pos-o), PickUp(r,o), move(pos-r,pos-t), PutDown(r,o,t)* is computed off-line and sent to the robot to execute, where *pos-r* represents the location of the robot, *pos-o* denotes the object and *pos-t* represents table's location. Suppose that when our robot is moving towards an object, the object is relocated to another position by somebody. The execution monitoring module on the robot needs to re-examine the situation and probably ask its planning module to generate a repair plan from the current unexpected situation. So execution monitoring mentioned in this thesis can be viewed as a complement to the planning system for the intelligent agent and not only deals with fault diagnosis but also needs to react to unexpected situations from the dynamic environment.

In the literature, there are generally two ways of dealing with dynamic environment. One is replanning from scratch when we face a different situation, the other is using plan repair or plan modification technique to reuse the original plan as much as possible. Although Nebel et al. [73] have proved that modifying an existing plan is (worst-case) no more efficient than a complete replanning, in practice, it is still quite costly to abandon previously generated plans and re-plan completely at run-time. There is another motivation for using plan repair techniques, and that is to solve a series of similar planning tasks

[59, 60]. These techniques need to store the plans which are successful in a plan library, so that once a similar task is presented, they retrieve a similar plan from the library and perform modification techniques to change that plan in order to complete the new task. These plan repair techniques are done at the planning stage and can be seen as an another planning algorithm, while most of the plan repair techniques mentioned below are done at execution time and do not have a library of previous plans.

3.1 EXECUTION MONITORING ON PLANS

In this section, we will review several execution monitoring techniques that are used to supervise the execution of the plans. They are divided into two groups. The first one is monitoring a single plan. The plan can have different structures, for instance it can be a straight-line plan or a partial hierarchical plan. Therefore, execution monitoring techniques are different due to differences in the structure of the plan. However, they do share the same idea, which is exploiting the structure of the plan to help monitoring in order to deal with unexpected situations at run-time. The second category of execution monitoring is called reactive execution monitoring. At planning stage, reactive execution monitoring predicts the unexpected situations that might arise at execution time and builds pre-computed responses to them. The reactive means one can decide the current action directly according to the current situation and not commit to any plans beforehand. Some other relevant execution monitoring techniques are also discussed, including continual planning, explanatory monitoring, semantic-knowledge based monitoring, and rationale-based monitoring.

3.1.1 *Monitoring a plan*

3.1.1.1 *PLANEX*

Preconditions	Clear(x)
	On(x, y)
	HandEmpty( )
Postconditions	**Add list**
	Holds(x)
	Clear (y)
	Delete list:
	HandEmpty( )
	On(x, y)
	Clear(x)

Table 3: Preconditions and Postconditions of action Unstack(x,y)

Preconditions	Hold(x)
Postconditions	**Add list:**
	OnTable (x)
	Clear(x)
	HandEmpty( )
	Delete list:
	Hold(x)

Table 4: Preconditions and Postconditions of action PutDown(x)

We firstly show one of the early execution monitoring systems, PLANEX [34], that works on straight-line plans. As explained in Chapter 2, STRIPS is a planning domain language that can be used to produce sequences of actions in order to accomplish certain tasks. The developers of STRIPS also present a higher-lever executor of the STRIPS plans in their system called PLANEX [34]. The actions that the robot can execute in PLANEX have a STRIPS representation, so each action has its own preconditions and postconditions (effects). The monitoring system PLANEX tends to answer questions such as "has the plan produced the expected results" or "what part of the plan needs to be

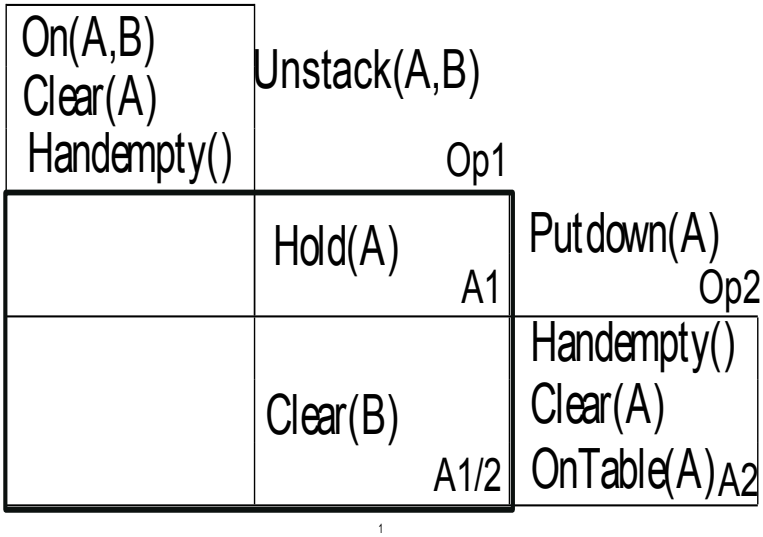

Figure 10: A Simple *TriangleTable*

executed so that the goal will be achieved". Consider a blocks-world problem, the STRIPS representations of the *Unstack* action and the *PutDown* action are shown in Table 3 and Table 4.

A specifically designed data structure for arranging the operators and the clauses, called a *triangle table*, is implemented in the PLANEX system that can be used to react to unexpected situations in a dynamic environment. Suppose a robot is executing a plan which is trying to move block A from the top of block B to a table. The triangle table with two sequential actions Unstack (A,B) and Putdown (A) is illustrated in Figure 10. From Figure 10 we can see that the preconditions of each action are given on the left-hand side, and effects are included in the cell which is right below its operator. In this example, the resulting clauses of executing operator *Unstack (A,B)* which are *Hold (A)* and *Clear (B)* are contained in the cell *A1*. The cell *A1/2* contains clauses in *A1* which are not deleted by the next operator *Op2* and the left-most column includes the preconditions for the entire plan. One property of PLANEX is having the ability to determine whether the rest of the plan is still applicable or not. This can be realized by using a unique rectangular sub-array (as shown in the box

in Figure 10). This sub array is defined as the *kernel* which contains all the supporting clauses that make the corresponding rest of plan applicable. So when an exogenous event occurs after executing the action *Unstack (A,B)*, as long as the clauses in the kernel (in this case *Hold(A)* and *Clear (B)*) are satisfied, it is guaranteed that executing this part of the plan will accomplish the task in the end. The kernel is sorted according to the number of actions left in the plan. The highest kernel corresponds tothe preconditions of the last action in the original plan. In the example, *Hold(A)* and *Clear (B)* are in the highest kernel for the last action *PutDown(A)* to be executed. So PLANEX works by finding the highest kernel that is satisfied at each time step and executes the corresponding rest of the plan. Because the planning domain for PLANEX assumes full and perfect observations about the world, PLANEX is not concerned with the issue of detecting exogenous events from raw sensory data.

3.1.1.2 *SIPE*

PLANEX can only work on straight-line plans, and we would like to demonstrate more execution monitoring techniques that can apply to advanced planning systems. As mentioned in Chapter 2, partial order planning tried to minimise the commitment at planning stage as little as possible, and includes action ordering or variable binding in action arguments. The key idea of partial order planning is allowing these commitments to be made at run-time which provides more alternatives than straight-line plans, where everything is determined prior to execution stage. One work of execution monitoring of partial order plans was included in the system called **System for Interactive Planning and Execution Monitoring (SIPE)**[111]. The plans that are monitored in [112] not only have partial order structure but also have a hierarchical structure that allows different layers of abstractions of

actions to be represented at different layers in the hierarchy. As stated in [112], the execution monitoring part of the SIPE system tries to accept different descriptions of unexpected events and also be able to determine how they affect the plan being executed. In particular, the replanning mechanism wants to utilize the original plan as much as possible in order to recover from unexpected situations.

Compared to the previous PLANEX system which is used to decide which part of the plan is still valid at each time step, the execution monitoring module in SIPE has the ability to modify the original plan more interactively according to the current situations, such as adding new sub-goals into initial plans. One thing worth noting here is that the replanning algorithm in SIPE is implemented as a rule-based system so all possible exogenous events and recovery actions are defined in advance. There are six possible problems that could occur in SIPE, such as the action does not achieve its purpose or the preconditions of the action become invalid, and each problem is associated with certain response actions which determine how to modify the original plan. There are a total of eight replanning actions that are specified before hand in SIPE for dealing with different unexpected events (some events can have multiple choices of recovery actions). For instance, one of the replanning actions **Reinstantiate** will instantiate a variable differently so that the preconditions of the action become true. Suppose that an office robot is asked to move from office A to office B with two possible routes *route1* and *route2*, and the robot decides to choose *route1* by considering the cost and other requirements. The preconditions of taking one route is $\mathtt{clear}(\mathtt{route})$. When the robot is executing the plan, if *route1* is blocked by some obstacles, this will make the preconditions of the action invalid. In this circumstance, the **Reinstantiate** action can choose an alternative route by instantiating the route variable to *route2*. This again demonstrates the idea

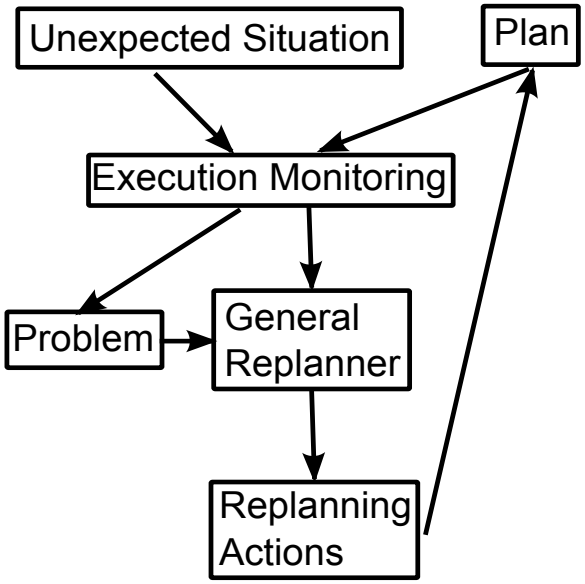

Figure 11: Control and Data Flow in SIPE's Replanner, adapted from [112]

of using the planning structure from the planner to minimise the effort of the replanning procedure at the execution monitoring stage, as the previous PLANEX system does. The actions in partial-order planning were modelled in a more expressive language in order to handle action arguments and have been utilized by the SIPE system.

Figure 11 shows the diagram of the execution monitoring module in the SIPE system. In this figure, the output *Problem* of execution monitoring module can be thought of *faults* detected. The inputs *Plan* and *Unexpected Situation* of execution monitoring indicate its ability to accept descriptions of the unexpected events at execution time. This execution monitoring process can be characterised as the fault identification stage in traditional FDI theory [24]. However, only a limited number of types of faults can distinguished and it has no ability to handle arbitrary unexpected faults. Once a problem (fault) has been successfully identified, the module *general replanner* will be called to decide the best replanning action from a set of pre-defined rules according to the detected problems. The replanning action will then try to modify the original plan in such a way that most of the initial plan

will be preserved [112]. The new generated plan will be monitored by following the same process.

The previous two execution monitoring systems PLANEX and SIPE both share the same core idea of utilizing the original plan as much as possible when dealing with certain unexpected events at run-time. SIPE performs plan modification according to the types of unexpected situations from a ruled-based system while PLANEX chooses a valid segmentation of the original plan that can achieve the task. However, both systems have the same limitation in that they require the ability to detect any unexpected situation in the environment automatically, but do not address the problem of how to detect such a discrepancy from raw sensory data directly. Another problem that these two systems have not considered is how to generate correct predicates in the planning language from the raw sensory data; for example checking the object's position from the cameras. In the end, both systems assume a perfect world description so no uncertainty or unreliable sensors are considered.

3.1.1.3 *GRIPE*

In order to address the problem of a more reliable verification of the execution of a plan, Doyle et al. [30] proposed a computer program called GRIPE (Generator of Requests Involving Perceptions, and Expectations) to insert perception request before and after the actions. These perception operators will have a set of expected values from sensor data, so if the observed value returned from the sensor is not included in this set, it would imply the failure of the preconditions or actions. As mentioned in [30], GRIPE focuses on generating perception requests and expectations to verify the execution of actions in a plan which is only part of the execution monitoring task.

As described in [30], there are four basic components in his execution monitoring system, namely, **Selection**, **Generation**, **Detection/Comparison**, and **Interpretation**. Selection will choose appropriate pre-conditions or post-conditions of the actions to monitor. After that, generation task will insert appropriate assertions such as pre-conditions and post-conditions in the plans. In particular, they define *Verification Operators* as follows:

Definition 8 (Verification Operators)**.** *Verification Operators represent the knowledge of which perceptions and expectations are appropriate for the pre-conditions and postconditions of which actions .*

Each verification operator will map each assertion,which will have its own expected value based on the current situation, into a set of sensory actions. For example, a grasp action for a robot requires the correct position of the robot's arm and the arm is not holding an object at the current stage. These assertions will be translated into several sensing actions, such as a vision sensor (to check the arm's position) or force sensor (to check what the arm is holding). Then a comparison between the expected value and observed value from the sensory data will be used to indicate the successful execution of the actions. Finally, the interpretation will decide how the failure actions affect the rest of the plan.

As mentioned before, it is intractable to monitor all assertions in the plan due to the limited computational power and time constraint. In [30], they discussed several criteria for selecting appropriate assertions at run-time.

1. **Uncertainty Criteria** Uncertainty can exist in the world model or action outcomes. A stochastic action which has multiple outcomes might need more verification operators to determine the failure of the action compared to deterministic actions.

2. **Dependency Criteria** A critical path in a plan represents those postconditions that will be required by actions later in the plan. In other words, postconditions that are not used by later actions in the plan can be ignored and need not be monitored because they will not affect continuing execution of the rest of the plan. This again can be characterised as determining relevance of the effects of the actions based on the validity of the plan.

3. **Importance Criteria** These criteria are largely related to the previous dependency criteria. Conditions of the actions can be prioritised based on metrics such as the number of subsequent actions that need these conditions.

4. **Recovery Ease Criteria** These criteria focus on how easily it can recover from the failure of an action. If it is quite difficult to recover from the failure of an action, the assertions of the action might need to be examined closely.

3.1.1.4 *Monitoring policy Execution*

Fritz et al 's work [39] focuses on monitoring policies of MDP problems (details in Chapter 2). They apply execution monitoring for MDP policies because of the incomplete model of the planning domains, so unexpected states could occur at any time step. In particular, unexpected situations will not only affect the validity of the current best plan, but will also affect its optimality. For instance, the original sub-optimal branches in the policy might become optimal after the unexpected situations occur. This idea of checking the optimality of plans was first introduced by Veloso et al.[108] and is shown in Section 3.1.3.4. Therefore, their execution monitoring technique needs to decide the optimality of the current best policy at execution time. They claimed re-planning for every unexpected state is costly and often unnecessary [39], so the main contribution of this

work is finding the relevant conditions that will affect the optimality of the current policy. By doing this, execution monitoring will ignore the unexpected states that only contain irrelevant conditions so as to avoid expensive replanning procedures.

One thing that is worthy of note is that they consider forward search-based MDP solvers rather than standard dynamic programming as explained in Chapter 2. As described in [26], a forward search-based MDP solver is an on-line solver that will start with a root node which contains only initial state S_0 and gradually expand its successors until a certain horizon is reached. Forward search-based MDP solvers require a heuristic estimate (V') of optimal value function (V^*) for all states in the domain to be computed, so it can back up these values from the leaf nodes of the search tree to the root using Bellman Backup operators. This can provide a better estimation of value functions for the states in the tree. Given this search tree, the best action for the current state can be selected greedily and also for the subsequent actions. An example of the search tree is illustrated in Figure 12 where circles represent states in the MDP, and rectangles represent action choices. Another thing to be noted here is that all the states and actions are represented in the situation calculus. The initial state S_0 is the root of the search tree and $N[a_1, S_0]$ represents the execution of action a_1 in the initial state. Since actions in the MDP have stochastic outcomes, they refer to the selection of an action outcome as nature's choice and the notion of $N[do(a'_{i,j}, s)]$ indicates the jth outcome of action i. As mentioned in [38], situation label nodes $N[s]$ will be annotated with rewards, and edges $E[a', S]$ will associate cost and probability of that outcome.

In the context of execution monitoring, they [39] want to make sure the current unexpected situation will not affect the validity and optimality of the policy. At first, the forward search-based MDP solver

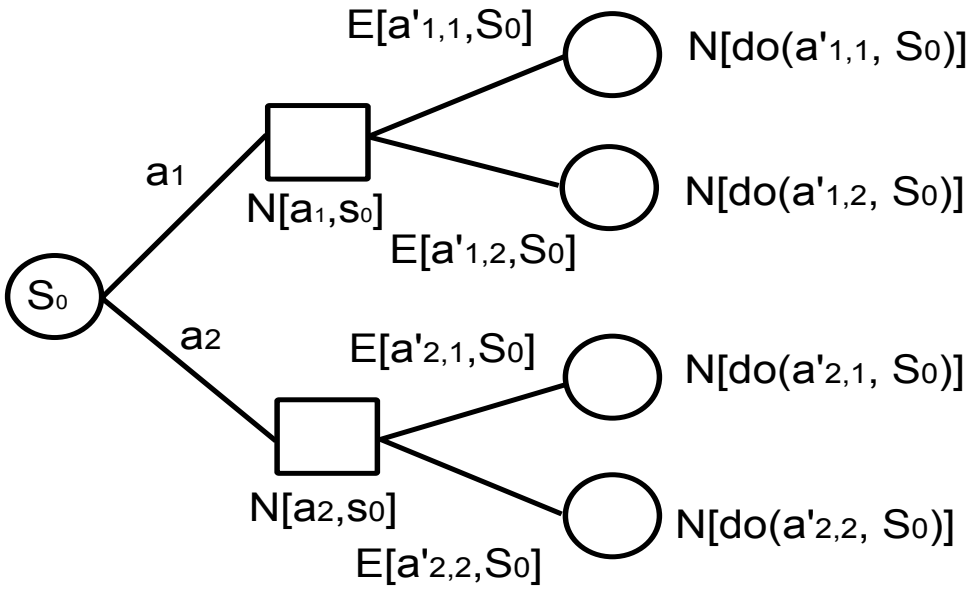

Figure 12: An example of annotated search tree for MDP monitoring, adapted from [37]

will only produce a policy (contingency plan) which contains the best action to take for current state and also for its successors. Given the policy itself, it is not enough to answer the above question, because the policy is extracted from the search tree and does not provide any information about how the optimal or near-optimal policy was selected. So Fritz et al. [39] annotate the policy with the search tree. The annotation is done by associating the root node in the policy with the complete search tree and its following nodes with corresponding sub-search trees. So it is only necessary to check whether the unexpected states affect the current annotating (sub-search) tree at execution time. This is done by regression which is defined as follows [39]:

Definition 9 (Regression)**.** *Regression of a formula* ψ *through an action* α *is a formula* ψ' *that holds prior to* α *being executed if and only if* ψ *holds after* α *is executed.*

By regressing the value function and other useful information from the search tree, such as the cost of an action, all the relevant conditions related to the current choice of the policy will be stored for current state. Therefore, the discrepancy between unexpected states

and actual states can be distinguished as relevant or irrelevant by comparing them with the regressed information.

3.1.1.5 *Abstract model*

Another piece of work done by Fritz [37] proposes an abstract execution monitoring model which is stated as follows:

1. during plan generation, annotate the planning data structures with all information relevant to the achievement of the objective.

2. when a discrepancy between the assumed and the estimated state of the world occurs, use this information to determine the degree of relevance of the discrepancy.

More specifically, *the planning data structure* contains all decision criteria that will affect the choice of the plan and the *objective* can be to ensure either the validity of the plan or the optimality of the plan. Fritz [37] claimed that in general it is too costly to re-plan every time there is a discrepancy occurring in the world, since some discrepancies might not affect the execution of the rest of the plan at all. He [37] also acknowledged that different plan annotations are required and different algorithms are needed depending on what is being monitored. For example, the previously mentioned PLANEX system utilizes the triangle table as an additional planning data structure in the monitoring procedure, because a classical straight-line plan is being monitored. The relevant conditions are computed by using regression techniques; for instance, preconditions of the actions are regressed and stored in the triangle table in the PLANEX system. Fritz [37] claimed that some other execution monitoring approaches also use regression techniques to obtain critical information about the choice of the plan.

3.1.2 *Reactive Plans*

As mentioned before, the previous execution monitoring approaches are trying to decide the relevance of the discrepancy with respect to the current plan and modify the plan accordingly. In this section, we discuss another category of execution monitoring approach called reactive planning which tries to predict beforehand what is going to happen in the environment as much as possible, and also generate corresponding plans for all situations. By doing this, it will only take a small amount of time to react to any situation at execution time by simply switching to the appropriate plan. This is actually a planning process but it aims to achieve the objective of execution monitoring.

3.1.2.1 *Universal Plan*

Universal planning [95] is one of the early attempts to tackle dynamic environments by introducing reactive plans. The author assumes the agent has incomplete knowledge about the initial state and current state. More importantly, he assumes external behaviours might affect the success of executing the plans. As mentioned before, these assumptions are the same reason for proposing execution monitoring techniques. The solution he proposed is that appropriate actions are chosen at execution time based on a decision tree which is constructed at the planning stage, and each node of the decision tree is one possible state of the world and the leaf is the action to perform under this description of the world. There are some obvious drawbacks with this approach. Firstly, even though they assume incomplete knowledge about the initial state of the world, they assume complete observability about the world during the execution, which is often not the case for an intelligent agent; Secondly, not all possible states of the world, including unexpected situations or fault states,

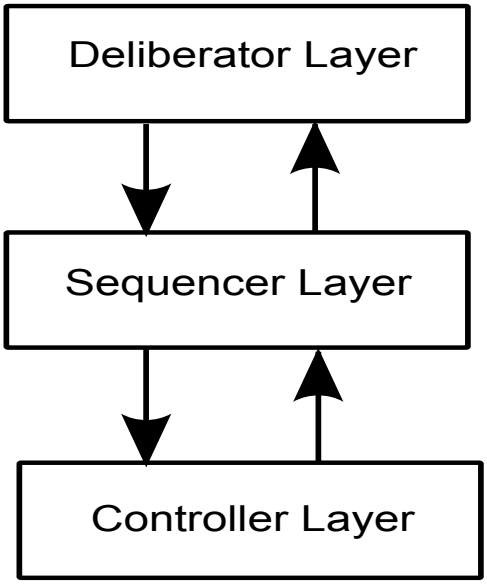

Figure 13: Three layers in 3T architecture for robotic control, adapted from [40]

can be enumerated during construction of the decision tree at the planning stage. Therefore additional execution monitoring needs to be incorporated.

3.1.2.2 *3T Architecture*

Another approach to address the problem of exogenous events and action failures is hierarchical robotic architecture such as 3T [35]. A diagram of the 3T architecture is shown in Figure 13. According to [40], the three layers of the architecture are separated depending on whether they utilize internal states (belief state). Primitive behaviours, such as obstacle avoidance or object tracking, are coded at the lowest level called the *controller layer*, which is highly related to the hardware of the robot [8] and does not use internal states. These primitive behaviours are controlled by a sensor feedback controller so they can react to sensory information directly and rapidly and no internal states need to be used at this stage. For example, the robot's object avoidance can be modelled as a basic skill in the controller layer and does not need to communicate with higher layer when the robot is

performing an action. In the highest level, *the Deliberator*, plans are generated by traditional time-consuming planning algorithms, such as STRIPS or MDPs solvers. Internal states which reflect the agent's belief about the current world are used at this layer. An intermediate level called *the Sequencer*, connects the other two layers. One requirement of this intermediate layer is that it should quickly translate high-level actions from the deliberator into a sequence of primitive behaviours and monitor the outcomes of these primitive actions. A specific language *Reactive Action Packages* (RAPs) [35] is built to fill the gap between the high-level planner layer and the low-level reactive layer by choosing the appropriate sequence of primitive actions according to different situations. For instance, consider a task of moving an object O from location A to location B. It has some primitive actions, such as $\mathtt{PickUp}(O, A)$ which picks up the object O at the location A, $\mathtt{Move}(O, A, B)$ and $\mathtt{PutDown}(O, B)$ which puts down the object O at the location B. In the RAP layer, it assumes there is a sensory action $\mathtt{Check}(O, B)$ which checks the success of this task. In the application of the 3T architecture called *IDEA*[43], *Mode Identification* (MI), which estimates the state from noisy sensory data and *Mode Recovery* (MR) which computes the least costly path from a faulty state to a normal state are also incorporated in the system to make it more robust. This interaction between the controller and the sequencer is where the reactivity comes from. The 3T architecture can be viewed as an extension of universal plans where appropriate reactions to the situations are programmed before-hand in the RAP.

Another application of the 3T architecture can be found in Fichtner et al.'s work [32] where they focus on two aspects. One is how to deal with dynamic information from the world and, in particular, that the information the robot has might be out of data, incomplete, and uncertain. The other is how to reason about and possibly recover

from failure situations at the highest logical level. In order to tackle the dynamic information, they represent all the sensing information as temporal information by attaching each observation with the time of its observation. The gathered information will also need to decay and update after certain time steps so that out-dated information will not mislead the high-level planning process. In their work, two types of sensing actions exist in the system: one is operated along-side the execution of the plan, for instance, when the robot is approaching the target position in a corridor, the information about the door in the corridor will be updated because sensing actions are performed concurrently with the execution of the moving actions. The other type of sensory processing is active sensing which is required directly by the planner. When either of the sensing actions obtain new information about the world, it will be announced to all the levels in the system. One difference between their work and previous 3T-based applications is that they are able to infer the faulty situations and generate recovery actions in the logical planning level. As described before, IDEA only performs state estimation and recovery at the lower two levels and does not allow the planning level to perform high-level reasoning and recovery.

3.1.3 *Other execution monitoring approaches*

After the discussion of execution monitoring on one specific plan and reactive planning, we are going to briefly list a number of other execution monitoring techniques.

3.1.3.1 *Continual planning*

The work on continual planning focuses on interleaving planning and execution given a dynamic environment [27]. According to [27], *continual planning* is defined as follows:

Definition 10 (Continual Planning)**.** *Continual planning is an ongoing dynamic process in which planning and execution are interleaved.*

Continual planning and the previously mentioned execution monitoring all tend to revise the plan during execution, but continual planning considers the revision of the plan as an ongoing process rather than one that is triggered only by failure of current plans [27]. In particular, continual planning is not only dealing with failure situations due to the dynamic world but also seeking new opportunities to accomplish the task more efficiently. For example, suppose an office robot is performing a delivery task in a building, which might involve taking an object O to a desired location L, the robot needs to reason which is the targeted location by navigating around the building. Let us say a person appears in the middle of the robot's route, on one hand, the robot should check whether the human appearance violates the existing plan (preconditions or goals), on the other hand, the robot can also take advantage of the human participants, for example, asking the people directly for the location. The continual planning will be concerned with whether to follow the existing plan or make some refinement to the current plan. According to [27], reactive planning which was discussed previously, can be viewed as a special case of continual planning, because RAPs pre-compute reactive actions in order to deal with certain changes in the world and do not trigger the time-consuming high level deliberation process when the world changes unexpectedly.

The main concern of continual planning is when and how to refine or revise the original plan. The trade-off is how to allow the agent to react to the changes of the world appropriately without consuming too much computation and time. Pollack [82] then introduced a *Bold* agent which rarely reconsiders the current best plan and a *Cautious* agent which always tries to evaluate the current situation at every time step at run-time. It is not difficult to see that not only is the *Bold* agent not going to consume any computational resources in re-planning during execution time, but that it will also fail to respond to any unexpected change in the environment. On the other hand, the *Cautious* agent will waste too much time and computation on unnecessary replanning. Kinny et al. [58] improved this approach by adding more parameters in the agent and the environment. For example, the rate of the environment change is characterized as one parameter. In their experiment, they tried to investigate how the agent's effectiveness changes as those parameters vary. As explained in [58], the effectiveness of the agent is the score it collects during execution divided by the maximum possible score it could have gained.

3.1.3.2 *Explanatory execution monitoring*

Explanatory diagnosis was first introduced by McIlraith [71] and aims to produce a sequence of actions that can explain the current inconsistent observations. It differs from model-based diagnosis (discussed in Section 3.2.1) which is trying to answer the question "what is wrong" in the system, it integrates plan actions into the diagnosis process in order to answer the question "what happened" to the system. Action failure and exogenous events are modelled in the system using the situation calculus language. McIlraith [71] claimed that explanatory diagnosis is analogous to generating plans for certain goals because explanatory diagnosis essentially generates a sequence of actions to

satisfy current observations. Since explanatory diagnosis will suffer from incomplete initial state and potentially large search space, regression and several other assumptions are made in order to focus the search to generate explanatory diagnosis.

Belief Management [44] shares the same basic idea as explanatory diagnosis and is integrated into IndiGolog [22] which is a robot programming language. Belief management is trying to generate all hypotheses about alternative outcomes of actions that might explain the inconsistency. Variations of actions and exogenous actions are also modelled in the domain, for example, the action *pickUpNothing* and *pickUpWrongObjects* can be two alternative outcomes of *pickUp* action. Since the difficulty of finding the most promising explanatory diagnosis from a large set of potential candidates remains, several techniques [97, 98] are proposed to efficiently produce results at run-time. Inspired by Fritz [37], they improve the belief management system by examining the relevance of the inconsistency situation [97] and only when the current situation affects the successful execution of the program, is the belief management system triggered to generate explanatory diagnosis. Belief management also pushes explanatory diagnosis one step forward by recovering from action failure or exogenous events and showing that it increases the success rate of the tasks.

3.1.3.3 *Semantic-Knowledge based Execution Monitoring*

Most of the previous execution monitoring approaches assume the agent has the ability to detect a discrepancy between the effects of the actions and the real state of the world. However, in some cases, it is not a trivial task to do so. For instance, if a robot is asked to delivery a book to an office but ends up in the kitchen, visual sensing might detect a microwave or sink in this room, but questions

remain about how to derive the current location from this implicit information. Execution monitoring incorporating semantic knowledge [12] has been developed to address this problem where a discrepancy between the agent's belief and the current state of the world can not be directly observed and needs to be reasoned with additional implicit information. More specifically, semantic knowledge refers to the meaning of objects expressed in terms of their properties and relations to other objects [12] and is coded using description logics. Instead of answering the question "Is the robot in Room A" in only three ways, namely "yes", "no" or "unknown", work in [13] extends this model to deal with a probabilistic representation and noisy sensors. The idea of semantic-knowledge based execution monitoring is intuitive. Suppose a robot has a 0.6 probability of being in a living room and 0.4 probability of being in a kitchen, seeing a sofa is going to give robot greater confidence that it is currently in a living room, because it has semantic-knowledge that "sofas" are more commonly associated with living rooms than kitchens. A choice of which sensing action to use arises during this monitoring. A measure of information gained from information theory is employed for this purpose, which means it will always choose sensing actions that maximise information gain. It is believed this approach complements other execution monitoring approaches, which work on different layers of the robots, such as hardware layer or high-level planning layer.

3.1.3.4 *Rationale-Based Monitoring*

The work on Rationale-Based Monitoring [108, 67] also tried to relax the assumption of static environments in classical planning and allows the world to change frequently and unexpectedly. However, they realized the sensing actions for gaining information about the state of the world are not free and it is computationally infeasible

to monitor all changes in the world. One major contribution from rationale-based monitoring is that they not only monitor conditions of the current best plan but also consider alternative plans, because the world can change in such a way that alternative plans become more attractive than the current optimal plan. They called monitoring of conditions that are relevant to current best plan as "Plan-based monitors" and monitoring of features in the world that could affect selecting alternative plans as "Alternative-based monitors". The objective of rationale-based monitoring is ensuring a new plan is valid and optimal after the change of the world occurs. One major difference between rationale-based monitoring and other monitoring approaches is they consider the changes and response occurring at plan generation stage rather than plan execution stage. They claimed that the planning task itself might takes a lot of time, for example, a large-scale military operation might require a large amount of time to plan and need to respond to the changes happening at the planning stage. In terms of the response, they defined several types of plan transformations for this purpose, namely adding to the plan, cutting from the plan and jumping in the plan. This is a set of pre-defined rules for unexpected changes in the world as SIPE defined replanning actions for dynamic environment. It might be necessary to add to the plan when true preconditions of the current best plan become false, another sequence of actions will be added to the plan to achieve these preconditions. This process is essentially adding new sub-goals to the existing plan; if sub-goals of the current best plan become true after the changes, they can eliminate those actions which are used to achieve these sub-goals, and this process is called cutting from the plan. Finally, when the current best plan become less attractive than alternative plans, for example if the utility of alternative plans is larger than current best plan, they can jump to alternative plans.

3.1.3.5 *Approximately Optimal Monitoring on Straight-line plans*

Boutilier [14] addresses the problem of approximately optimal monitoring of preconditions of the action on a straight-line plan. He noticed that, on one hand, monitoring all preconditions of actions in a plan at every step of execution is too expensive, but, on the other hand, if we only monitor the current state of the world, it is often too late to repair the plan when we actually discover the fault later. So the approach tries to take into account monitoring actions costs, the probability of each precondition failure and the value of alternative plans in order to determine which precondition to monitor and which plan to follow. From the agent's perspective, there are two stages during each time step of execution, one is selecting the best monitoring action over corresponding preconditions, the other is deciding whether to continue the current plan or switch to an alternative plan after receiving noisy observation actions (monitoring actions). One trade-off that needs to be made for this approach is between information gain from monitoring actions and action costs. The information gain that we can obtain from monitoring precondition B can be computed as follows:

$$VOI(B) = Pr(B) * (v(\pi_B) - v(\pi_{fail})) \tag{12}$$

where $Pr(B)$ is probability of precondition B's failure, $v(\pi_B)$ represents value of best plan if we detect the failure and π_{fail} denotes value of executing current plan without detecting the failure. Although the plan he considered here is only a simple straight-line plan, there is a large amount of prior information that needs to be gained, such as:

- the probability that preconditions may fail.

- the cost of attempting to execute a plan action when its precondition has failed.
- the value of the best alternative plan at any point during plan execution.
- a model of monitoring processes which defines the accuracy of observation monitoring actions.

Because Boutilier wants to find an optimal sequence of asserting monitoring actions and selecting alternative plans at run-time, the problem is then formulated as a POMDP problem (as described in Chapter 2). As mentioned in [14], one limitation of this approach is the computationally intractability of finding optimal solutions for large POMDP problems; for instance, a plan with n preconditions might need an observation space with size 2^n and also as many as 2^n states. Another limitation of this approach is that much of the prior information about the model is hard to obtain in reality; for example, finding a utility value of executing the current plan with failure preconditions is not trivial.

3.1.3.6 *MBD for plan execution monitoring*

The work in [107] proposes using model-based diagnosis (MBD) technique for plan execution monitoring. MBD is a consistency-based approach to search for the most likely diagnoses based on discrepancies between the actual observed state of the system and the predicted state of the system. More details of MBD will be in the next section. [107] characterise the execution monitoring on an autonomous robot in three different layers, viz action level, plan level and world level. Execution monitoring on an action level only checks the current action's preconditions in order to make it applicable and also to make sure the postconditions appear after the execution of the current ac-

tion. Secondly, PLANEX or SIPE are characterised by [107] as execution monitoring on plan level, which makes use of the rich structure from the planning to enhance the plan execution. Steinbauer et al. [107] argue that additional information about how the world evolves could make the execution of the plan more robust, so they add an extended background theory to the world model to describe axioms of the world, such as if a robot perceives an object in a location, and, even if the robot moves to another position, the object should remain in the same position with the assumption that no exogenous events will happen. Besides this, certain unexpected events or action failure can also be modelled in this background theory. During execution of the plans, a robot can derive possible diagnoses to explain discrepancies between the states we derived from the background model and the belief state from sensor information.

An illustrated example of an extended background sentence in [107] is:

$$\neg abnormal(vis) \wedge see(obj) \Rightarrow perceived(obj)$$

$$\neg abnormal(loc) \wedge at(pos) \Rightarrow isat(obj, pos)$$

the former states that if the robot's vision system *vis* is working properly and it sees the object *obj* then object *obj* is perceived. Similarly, the latter indicates that if the localisation system of the robot *loc* works correctly, then its output will represent the robot's current location.

An inconsistency is said to be found when the robot's belief is contradictory to the predicted state of world. For instance, after performing action *Move(A,B)*, if the agent determines its current location is C rather than B, then it will report that a discrepancy is detected. Another capability of this approach is that it can reason about the

cause of this discrepancy. A conflict-directed algorithm [113] in MBD can be used to guide the search to find possible diagnoses that can explain the current discrepancy. However, even given a single discrepancy, there are potentially many diagnoses that can result in this discrepancy. For instance, when the robot found itself in an unexpected room after performing moving actions, the cause could be failure of a localisation component or failure of a moving action. Additional information gathering actions are required to reduce the number of possible diagnoses; for example, normally if we can grab something we can make sure it really exists. So if a robot can grab an object then it can reason that the chance of the vision component malfunctioning to produce a ghost object is zero. The author in [107] is also aware that different sensing actions can have different costs and risks, but he does not explicitly address the problem of how to choose from these sensing actions.

3.2 STATE ESTIMATION

Previously discussed execution monitoring techniques aim to make sure the current plan is either valid or optimal no matter which situation occurs at run-time. There are many other execution monitoring techniques in literature that try to estimate the current state of the system given the current observation or a history of observations. These techniques can be seen as the first step of previously discussed plan-related execution monitoring approaches, because the information about current state estimation can be used to decide whether to continue executing the existing plan or to modify the plan if necessary. However, this section is not necessary for understanding the rest of the thesis, the purpose of which is to show current developments of state estimation techniques. In general, there is a model to

describe how each component in the system works and how the observation action works. Each component in the system has its own state, such as normal state and abnormal state. The goal of these fault diagnosis techniques is to find a diagnosis which is usually a complete assignment of the states of all components in the system that will keep the observations consistent. We describe traditional consistency-based diagnosis Model-Based Diagnosis (MBD), which will identify a discrepancy between expected observation and real observation and infer the faulty components that cause such a discrepancy, as well as sampled-based Bayesian filtering methods in this section.

3.2.1 *Model-Based Diagnosis*

Model-Based Diagnosis (MBD) [24, 86, 113] provides a general framework to solve diagnosis problems in AI. A comprehensive first order logic representation of the model needs to be built with the ability to represent how each component works when it is in a normal state or abnormal state and which observations we will get when everything is working correctly.

Formally, a system [23] is a triple (SD, COMPS, OBS) where:

1. SD is a first order description of the system
2. COMPS is a finite set of constants in the system
3. OBS is a set of first order sentences that describe observations of the system.

One of the most popular model-based diagnosis algorithms is called conflict-directed A^* search algorithm [24], introduced in 1987. The goal of the algorithm is finding a diagnosis from a set of candidates that will explain the current observation. According to [24], a *candidate* is defined as follows:

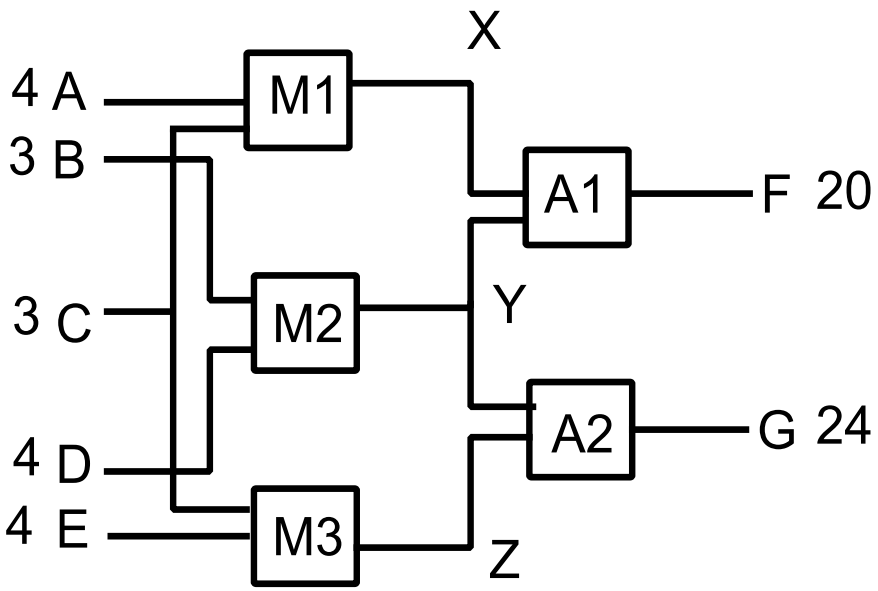

Figure 14: An example of MBD approach, adapted from [23]

Definition 11 (A candidate)**.** *A candidate is a possible assignment of the status of all components in the system based on the description of the components COMPS.*

A diagnosis candidate is a candidate, but it will generate consistent observation. The search algorithm relies on a concept called *a conflict*. Formally, [24] defines a conflict as follows:

Definition 12 (A conflict)**.** *A conflict is a set of components that cannot work normally at the same time because of current observations OBS and system description SD.*

The search procedure can benefit from the concept of conflict in two ways: one is that a diagnosis of the system is a complete assignment of the components that can resolve all conflicts; the other is that the candidates that contain existing conflicts will be pruned during the search because they will never generate consistent observation according to the definition of the conflict.

Figure 14 illustrates a basic example of an MBD model. The components in the system are ADDERs (A) and MULTIPLIERs (M) respectively. In MBD, the adder and multiplier can be modelled as:

$$\mathrm{ADDER}(x) : \neg \mathrm{AB}(x) \Rightarrow \mathrm{out}(x) = \mathrm{input1}(x) + \mathrm{input2}(x)$$

$$\mathrm{MULTIPLIER}(x) : \neg \mathrm{AB}(x) \Rightarrow \mathrm{out}(x) = \mathrm{input1}(x) * \mathrm{input2}(x)$$

where $\mathrm{AB}(x)$ specifies that component x is abnormal in the system, so $\neg \mathrm{AB}(x)$ means component x is currently working correctly. These two sentences govern component ADDER and MULTIPLIER behaviours, specifying how they are working when they are in normal states. In this example, the inputs of the system are $A = 4, B = 3, C = 3, D = 4$ and $E = 4$ and the outputs of the systems are $F = 20$ and $G = 24$. Once we have information about the inputs and the outputs of the system, the conflicts can be decided from them. In this case, if the components M1, M2 and M3 are all working normally, the output F should be 24. Since we had observation that the value of F is 20, we conclude that the set of $M1, M2, A1$ is one conflict that should be resolved, which means components M1, M2 and M3 cannot be working correctly at the same time. After seeing the value of G is 24, we can derive another conflict which is $M1, A1, A2, M3$, because if all components in this set are working correctly, the value of F and the value of G should be the same. In this example, we can produce two distinguishing single faults A1 and M1 that can resolve the previous two conflicts.

3.2.2 *Bayesian Filtering Methods*

Another category of diagnosis approach is the Bayesian filtering method [72]. A dynamic Bayesian network, which is a probabilistic temporal model, is used to represent how a dynamic system evolves. Given a history of noisy observations, Bayesian filtering methods are able to

estimate the current state of the system using a statistical approach. The main difference between Bayesian filtering methods and consistency-based diagnosis is that we need to infer the hidden current state of the system because the observation actions are noisy. Thus we cannot just simulate the expected behaviour and rely on the discrepancy to tell us when it is wrong. In general, there is a system model and an observation model for Bayesian filtering methods which are shown as follows:

$$x_{t+1} = f(x_t, u_t, v_t) \tag{13}$$

$$y_{t+1} = g(x_{t+1}, w_{t+1}) \tag{14}$$

where x_t is the n-dimensional state of the system at time t, which is governed by a stochastic difference Equation (13), an input vector u_t and a q-dimensional state noise vector v_t. y_t is an m-dimensional observation vector and w_t is the observation noise.

Normally, the random variables v_k and w_k are assumed to be independent and to have multinomial probability distributions:

$$p(v) \sim N(0, Q) \tag{15}$$

$$p(w) \sim N(0, R) \tag{16}$$

where Q is an $n * n$ covariance matrix and R is an $m * m$ covariance matrix.

Filtering, which is the estimation of the current state given the observations so far, is one of the inferences we can perform on this model. The goal of filtering is to compute the probability distribution $P(x_t|y_{1\ldots t})$ often referred to as the belief distribution which represents the probability that the system is on each possible state. This is actually the state estimation step in solving POMDPs, as shown in Chapter 2.

According to Bayes rule, the posterior distribution $P(x_t|y_{1\ldots t})$ can be written as

$$P(x_t|y_{1\ldots t}) = \frac{p(y_t|x_t y_{1\ldots t-1})p(x_t|y_{1\ldots t-1})}{p(y_t|y_{1\ldots t-1})} \quad (17)$$

$$= \frac{p(y_t|x_t)\int P(x_t|x_{t-1})P(x_{t-1}|y_{1\ldots t-1})d(x_{t-1})}{p(y_t|y_{1\ldots t-1})} \quad (18)$$

where $p(y_t|y_{1\ldots t-1})$ is a normalizing constant.

The integral here plays an important part in determining how hard it is to solve Equation 18. If the stochastic difference equations f and g are linear equations, and the system and observation noise is Gaussian, then Equation 18 has an analytical and closed form solution called the Kalman filter equations[110]. If stochastic difference equations f and g are non-linear equations, people often try to linearise the difference equation using the Taylor series expansion of Equation 13. This results in the extended Kalman filter algorithm or EKF [3].

There are some fundamental drawbacks of the EKF. Firstly, the EKF needs to evaluate Jacobians at every time step, which is computationally expensive. Secondly, since the EKF neglects the second order and higher order terms in the mean and fourth and higher order terms in the covariance, the accuracy of the estimated mean is only up to the first order. The Unscented Kalman Filter or UKF [54], a variant of the EKF, was developed to address these problems. It converts the

1. For N samples $p_0^i = (x_0^i)$, sample each p_0^i from the prior $P(X_0)$.
2. For each time-step t do
   a) **Prediction:** For each sample p, do:
      i. Sample a new continuous state:

$$\hat{x}_t^i \sim P(X_t|x_{t-1}^i)$$

   b) **Weighting:** For each sample:

$$w_t^i \leftarrow P(y_t|\hat{x}_t^i)$$

   c) **Resampling:** Sample N new samples p_t^i where

$$P(p_t^i) \propto w_t^i$$

Figure 15: The particle filtering algorithm for a continuous state model.

distribution into a set of sigma points then applies equations to each point and computes a weighted mean and covariance of the resulting points. More details about UKF can be found in [55].

The particle filter algorithm [29], also known as sequential important sampling (SIS), approximates the belief distribution by a set of samples. As shown in 15, the algorithm consists of three main steps: prediction, in which a new state x_t is computed according to the system Equation 13; weighting, in which the predicted sample is compared with the observation y_t to obtain a weight for each sample proportional to the likelihood of the sample generating the observation; re-sampling, in which the set of weighted samples are converted into a set of samples of uniform weight. The Particle filter can deal with non-linear dynamic equations and non-Gaussian noise much better but one of its weaknesses is requiring a large number of sample points when the dimensionality of the system increases. More details about the particle filter can be found in [28].

We have listed a number of execution monitoring approaches in this chapter. In the context of monitoring plan execution, three procedures need to be included: the first one is deciding what is the current state of the system according to the observation received so far. Most of the fault diagnosis techniques fall into this category and other techniques which are used to detect unexpected situations also serve this purpose. Secondly, once we decide something is going wrong, we have to determine how severe this will be. The current situation might affect the execution of the action or the future steps of the plans. Finally, a plan modifying procedure needs to be called in order to repair the current plan or improve the current plan according to the current situation. Some execution monitoring approaches shown above used pre-fined rules to associate repair actions to different situations in order to avoid time consuming replanning for every situation.

The execution monitoring approaches which will be discussed in the next two chapters are tackling the planning problem with stochastic actions and noisy observations. Therefore, we do not have to consider state estimation problem because no unexpected situations such as exogenous events will occur during execution time: the system will evolve as we expect. The reason for proposing execution monitoring approaches in such a planning problem is that exact solutions to find optimal plans or policies are infeasible so that only approximate solutions are generated at planning time. The main difference between our execution monitoring approach and other execution monitoring approaches is that we are seeking opportunities, which are more active than the previously discussed approach, to improve our existing approximate plans or policies. The execution monitoring techniques that are closest to our approach are rational-based monitoring and

policy monitoring where alternative solutions are considered to ensure the best plan is executing. However, they do not consider the partially observable environment in the domain.

4

EXECUTION MONITORING ON QUASI-DETERMINISTIC POMDP

In the previous chapter, we surveyed a variety of execution monitoring approaches ranging from the diagnosis community to the planning community. In the diagnosis community, execution monitoring techniques usually do not consider the high-level plan that is currently being executed but aim to identify and recover from any faults that could potentially occur to the components of the system. In the planning community, things are much more complicated because the current state of the system, the current state of the environment, and the current and future steps of the plan need to be taken into account. Generally speaking, there are two steps for monitoring the execution of a plan. Firstly, an execution monitoring approach needs to be able to realize something is going wrong given the current information. It has to detect the situation where initial plans are not valid or optimal at run-time, which could be due to exogenous events such as malicious actions occurring in the environment or the approximation solutions generated by the planner during plan generation do not perform well for the current state. Different criteria (preserving plan's validity or optimality) and different planning domains (complete or incomplete model) often require varieties of monitoring ability in the system. For instance, if the world domain is assumed to be complete, it is not necessary to include sensing ability in the monitoring module to identify unexpected outcomes of the actions because all the outcomes are included in the planning model. Once it has been decided it is time to repair or improve existing plans, execution moni-

toring approaches need to exploit the structure or the information of the original plans at run-time in order to guide the plan modification procedure.

Recent improvements in the robustness and reliability of mobile robots have led to a number of interesting planning problems characterised by partially observable worlds with deterministic or nearly deterministic actions. Examples include planning for Mars rovers, where we generally assume the rovers will execute the commands correctly but we do not know what data each observation will produce, and robotic security and monitoring tasks, where again the vehicle can move reliably to locations but uses unreliable vision and other sensors to detect the objects and people it must interact with. A somewhat different example is the algorithm selection planning problem of [106], where the task is to identify the objects in a scene.

Quasi-deterministic problems are closely related to partially observable Markov decision problems (POMDPs). They can be thought of as POMDPs where the actions are divided into two sets, those that change the state but produce no (informative) observations, and those that provide observations without changing the state. In terms of complexity, finding ϵ-optimal policies for quasi-deterministic problems is in PSPACE [6]. A policy π will be called a ϵ-optimal policy if $p\{V_\pi \geqslant V_{\pi'} - \epsilon\} = 1$ for all policies π', which means the difference between its return value and optimal policy's return value is also less than ϵ. Therefore, it is no easier than solving general POMDPs. However, POMDP algorithms do not scale to the size of the problems we are interested in. Even using point-based approximations and a structured representation [83] we can only solve problems with tens of millions of states, corresponding to classical planning problems with around 25 binary variables.

In this chapter we propose an approach to solving the above quasi-deterministic problems that uses a mixture of plan time and execution time components. At plan time, we remove the stochasticity from the actions to generate completely observable planning problems which we then generate plans for using either a classical contingency planning approach (described in Section 4.2) or a Markov decision problem planner (Section 4.4). At execution time we monitor the actual belief state of the agent as the plan is executed, and re-evaluate the plan in light of that belief state. We use a value of information calculation to determine if there are information gathering actions that will change the belief state in such a way as to improve the expected quality of the remainder of the plan and, if so, we add them to the execution of the plan.

One way to think of this approach is to examine the belief space of the agent as it executes the plan. Plans generated for MDPs or using a classical contingency planner are finding good/optimal actions for the vertices of this space (vertices correspond to belief states where only one state has non-zero probability) and these actions will tend to become further from optimal for beliefs further from the vertices—those that are less certain about the true state of the world. Due to the uncertain initial belief state, at execution time the agent will typically not be at a vertex, so the plan may be arbitrarily poor. The execution monitoring attempts to gather information that will move the agent closer to the vertices, thereby improving the expected performance of the plan.

A more detailed explanation of the translation will be given in Section 4.2 and Section 4.4. Execution monitoring on both contingency plans and MDP policies will be given in Section 4.3 and 4.5. Empirical results for these two approaches will be shown in Section 4.6.

Quasi-deterministic planning problems can be defined as POMDP problems in which the actions are of two types: *state-changing actions* and *observation-making actions* as follows:

Definition 13. *A* **state changing action** a *is one such that:*

$$\forall s \exists s' : P(s, a, s') = 1 \wedge P(s, a, s'') = 0 \textit{ if } s'' \neq s'$$

$$\forall s, s', t, t' \exists o : P(s, a, s', o) = P(t, a, t', o)$$

That is, for every state in which the action is performed, there is one exact state it can transition to, and there is one observation that happens no matter what the current state is.

Definition 14. *An* **observation-making action** a *has a noisy observation function* O *and:*

$$\forall s : P(s, a, s) = 1 \wedge P(s, a, s') = 0 \textit{ if } s' \neq s$$

$$\forall s \exists o_1, o_2 : P(s, a, s, o_1) \neq 0 \wedge P(s, a, s, o_2) \neq 0$$

Definition 15. *A* **quasi-deterministic POMDP** *is a POMDP in which every action is either state changing or observation-making.*

One thing to note here is that Quasi-deterministic POMDPs are different from Deterministic POMDP (Det-POMDP) where the latter assumes both deterministic actions and observations. The only uncertainty which comes from Det-POMDPs is initial belief state. Traditional POMDP solver, such as dynamic programming algorithms described in Chapter 2, needs to enumerate all possible states for transition, observation and reward functions which is problematic if the problems have large state space. Therefore, much effort has been put

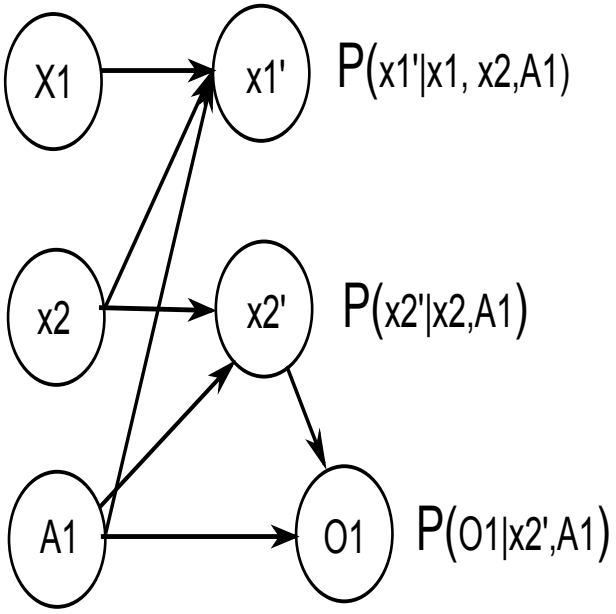

Figure 16: An example of dynamic Bayesian network

into representing these functions in a more compact way [15, 46]. As in classical planning, a system state in a POMDP can be represented as a finite set of state variables X. Each variable X_i is assumed with a domain D_i, so the size of the system states is $|D_1| \times |D_2| \times \ldots \times |D_n|$ if we have n state variables. Given the representation of this system, one way of efficiently representing state transition and observation transition is using dynamic Bayesian network, which is a graphical form of representing stochastic processes. Figure 16 shows a state transition and an observation transition under action A_1. The problem has two state variables x_1 and x_2 and one observation variable O_1. Current state is $\{x_1, x_2\}$ and next future state is $\{x_1^{'}, x_2^{'}\}$. Arcs in the graph represent the dependency of current variables and next step state variables. This compact representation allows computing the distribution of each state variable by looking-at relevant parent variables. For example, next belief state in Figure 16 can be computed as follows: $P(x_1^{'}, x_2^{'}|x_1, x_2, A_1) = P(x_1^{'}|x_1, x_2, A_1)P(x_2^{'}|x_2, A_1)$, which is a product of the conditional distribution of two state variables.

In practice, the Quasi-deterministic problems we are interested in are unlikely to be specified as flat POMDPs. Rather, we expect them to be specified using this dynamic Bayesian representation as shown above.

4.2 GENERATING CONTINGENCY PLANS

Given a quasi-deterministic planning problem as described in the previous section, we seek to generate complete contingency plans where each branch point in a plan is associated with one possible outcome of an observation action. We use the simple approach of *Warplan-C* [109] to generate contingency plans. Warplan-C algorithm tries to generate contingent plans for problems with non-deterministic actions. They assume some actions can have two possible outcomes O_1 and O_2. A straight-line plan which assumes all non-deterministic actions will produce O_1 outcome, was first generated, then incrementally added plan branches accounting for the other outcome O_2. As an example shown in Figure 17, the only non-deterministic action in the domain is action A_1. After a plan without any branching was generated, a newly generated plan considering the O_2 outcome of action A_1 was combined with the original plan to form the final contingency plan.

As for a quasi-deterministic planning problem, since only the observation action can have multiple outcomes, the initial straight-line plan was constructed by considering the most likely probabilistic effect of each observation action. This is essentially the single-outcome determinisation [115] in FF-replan. However, FF-replan only generates a straight-line plan on the deterministic variant of the planning problem, and it executes this plan right away until observing an unexpected effect, so no contingency plan is generated at any time. For example, if there is an object detect action that can successfully iden-

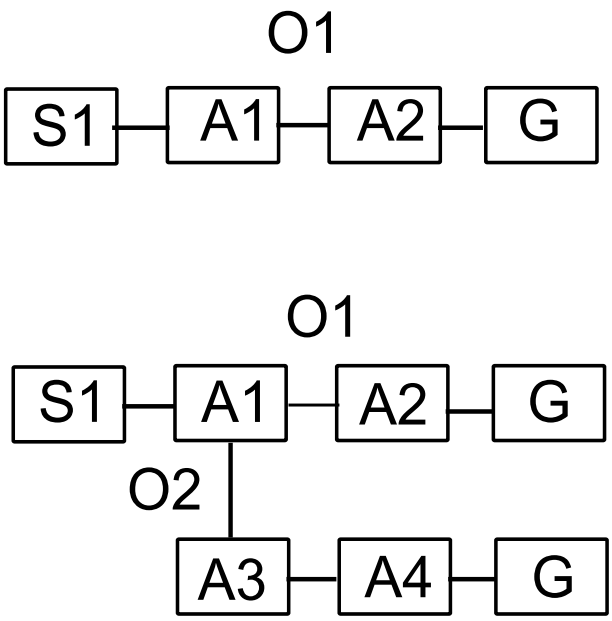

Figure 17: An example of Warplan-c algorithm. S_1 is an initial state, G is a goal state and only action A_1 has two possible outcomes O_1 and O_2

tify an object with 0.9 and miss out the object with 0.1 when the object really exists, the determinised version of this observation action will assume it can always detect the object perfectly whenever the object exists. Similarly, only the most likely state from the initial belief state is used to define the initial state of the determinised problem. For instance, suppose the initial belief state of road A is 0.7 *clear* and 0.3 *unclear*, road A's initial state is assumed to be *clear* after determinisation. This approach will convert a quasi-deterministic planning problem into a standard classical deterministic model. We then forward this determinised problem to the classical state-space planner FF [51] (as described in Chapter 2). FF then generates a straight-line plan to achieve the goal from the determinised initial state.

In this thesis, the Quasi-deterministic problems are represented using the probabilistic planning domain definition language (PPDDL) [117] which is designed for completely observable problems. In PPDDL, a database that captures the facts of the environment is sufficient for planning and execution because the agent has complete knowledge of

the environment. However, quasi-deterministic problems assume incomplete knowledge about the environment during planning and execution. Therefore, to use PPDDL for a quasi-deterministic problem, we need to represent the effects of the observation-making actions. Following Wyatt et al. [114], we do this by adding a knowledge predicate *kval()* to indicate the agent's knowledge about an observation variable. This knowledge-level action representation has been proposed in PKS [80] which is able to construct contingency plans in the presence of incomplete knowledge. In PKS [80], only the agent's knowledge is represented by a set of databases and actions are represented as updates to these databases. PPDDL with the additional knowledge predicate *kval()* can be seen as a combination of facts representation of the environment and knowledge representation of the agents.

In this thesis, a *RockSample* domain will be used as an illustration example throughout this chapter to demonstrate the idea of the translation-based approach of solving quasi-deterministic problems. The goal of the *RockSample* domain is to try to sample as many good rocks as possible and move into an exit zone at a grid map. The positions of the rocks are known to a rover and the movement of the rover is also deterministic. The difficulty of this domain comes from the value of each rock which is unknown at the beginning and only a noisy observation action is available to the agent. The trade-off which needs to be made here is when to perform observation actions for each rock. On one hand, if the rover moves closer to the examined rock, it can perform a more reliable, but costly, observation action. On the other hand, the rover can execute a cheaper but less accurate observation action in a position which is far away from the rock. If the value of the rock turns out to be bad, the rover can quickly move to other rocks without wasting too many movement actions. There

are five state-changing actions in this domain: four moving actions and one sampling action. A reward of 20 will be given if the rover samples a good rock and goes to the exit and a reward of -40 if a bad rock is sampled. A large penalty is given if there is no rock at the position of the rover when sampling, or if the rover moves out of grid except to go to the exit. *RockSample*(n, k) denotes a n by n grid with k rocks. The size of the state space is $n^2 \times 2^k$. To capture the idea that observation-making actions are faster and cheaper than state-changing actions, we use a version of *RockSample* where the cost of observation actions is less than the cost of movement actions.

In the *RockSample* problem [102], we use *kval(rovero, rocko, good)* to reflect that *rovero* knows *rocko* has good scientific value. The knowledge predicate is included in the effects of the observation-making action and also appears in the preconditions of other state-changing actions to ensure that the agent has to select observation action to find the value. Because state-changing actions produce uninformative observations, knowledge predicates will only appear in the effects of observation-making actions. For example, a specification of initial state and *checkRocko* action for *rocko* from the original RockSample domain might look as follow:

```
init [
    (rock0value (good (0.6)) (bad (0.4)))
    (rock1value (good (0.6)) (bad (0.4)))
    (rock2value (good (0.6)) (bad (0.4)))
    (rock3value (good (0.6)) (bad (0.4)))
]

Action checkRock0
  at_waypoint(SAMEat_waypoint)
  out(outno)
  exit(exitno)
  rock0value (SAMErock0value)
  rock1value (SAMErock1value)
  rock2value (SAMErock2value)
  rock3value (SAMErock3value)
```

```
  asresult (asresultempty)
  observe
    ovalue (at_waypoint
      (point0 (rock0value
        (good (ovalue' (ogood (0.7)) (obad (0.3))))
        (bad (ovalue' (ogood (0.3)) (obad (0.7))))))
      (point1 (rock0value
        (good (ovalue' (ogood (0.8)) (obad (0.2))))
        (bad (ovalue' (ogood (0.2)) (obad (0.8)))))))
  endobserve
  cost  (at_waypoint
        (point0 (0.4))
        (point1 (0.6)))
endAction
```

where only states with uncertainty (value of each rock) are shown here and *ovalue* is an observation variable which is represented separately from state variables. Depending on the current position of the rover (*pointo* or *point1*), observation action *checkRocko* will have different accuracy and cost. A simpler PPDDL version of checking actions, which have the same accuracy regardless of the current position of the rover and the rock, might appears as follows:

```
(:init
(probabilistic  0.6(rock_value rock0 good)
                0.4 (rock_value rock0 bad))
(probabilistic  0.6(rock_value rock1 good)
                0.4 (rock_value rock1 bad))
(probabilistic  0.6(rock_value rock2 good)
                0.4 (rock_value rock2 bad))
(probabilistic  0.6(rock_value rock3 good)
                0.4 (rock_value rock3 bad))
)

(:action checkRock
 :parameters
  (?r -rover ?rock -rocksample
   ?value -rockvalue)
 :preconditions
  (not (measured ?r ?rock ?value))
 :effect
  (and (when (and (rock_value ?rock ?value)
                  (= ?value good))
```

```
        (probabilistic
          0.8 (and (measured ?r ?rock good)
                   (kval ?r ?rock good)))
          0.2 (and (measured ?r ?rock bad)
                   (kval ?r ?rock bad)))
      (when (and (rock_value ?rock ?value)
                 (= ?value bad))
        (probabilistic
          0.8 (and (measured ?r ?rock bad)
                   (kval ?r ?rock bad)))
          0.2 (and (measured ?r ?rock good)
                   (kval ?r ?rock good))))
 :cost (0.6))
```

As can be seen, the accuracy of the observation action *checkRock* is always 0.8 and the cost of this action is always 0.6 and the action has a parameter *rock* so it can represent checking actions for all the rocks and has a knowledge predicate *kval* in the conditional effect.

After single-outcome determinisation, the initial belief state and the action *checkRock* in PPDDL then become as follows:

```
(:init
(rock_value rock0 good)
(rock_value rock1 good)
(rock_value rock2 good)
(rock_value rock3 good))
(:action checkRock
 :parameters
  (?r -rover ?rock -rocksample
   ?value -rockvalue)
 :preconditions
  (not (measured ?r ?rock ?value))
 :effect
  (and (when (and (rock_value ?rock ?value)
                  (= ?value good))
             (and (measured ?r ?rock good)
               (kval ?r ?rock good)))
       (when (and (rock_value ?rock ?value)
                  (= ?value bad))
             (and (measured ?r ?rock bad)
               (kval ?r ?rock bad)))))
```

So far we have shown how to translate from a QDET-POMDP problem into a classical planning problem. Since in reality each observation-making action in the plan could have an outcome other than the one selected in the determinisation, as Warplan-C algorithm, we then traverse the straight-line plan produced by FF and update the current state as we go, until an observation-making action is encountered. This then forms a branch point in the plan. For each possible value of the observed state variable, apart from the one already planned for, we call FF again to generate a new branch which can be attached to the plan at this point. This process repeats itself until all observation making actions have branches for every possible value of the observed variable. The full algorithm is displayed in Algorithm 3.

Algorithm 3 Generating the contingency plan using FF

```
plan=FF(initial-state,goal)
while plan contains observation actions without branches do
  Let o be an initial observation making variable v = v_1 without a
  branch in plan
  Let s be the belief state after executing all actions preceding o
  from the initial state
  for each value v_i, i ≠ 1 of v with non-zero probability in s do
    branch = FF(s ∪ (v = v_i), goal)
    Insert branch as a branch at o
  end for
end while
```

Since the approach in Algorithm 3 enumerates all the possible contingencies that could happen during execution, the number of branches in the contingency plan is exponential in the number of observation-making actions in the plan. This is precisely why it is useful that the state-changing actions do not generate observations—to keep the number of branches as low as possible. Also, since the determinised problem assumes the observation-making actions are perfectly accurate, in any branch of the plan at most one observation-making action will appear for each state variable in S_p. Thus, in practice, we expect there to be a relatively small number of branches. In

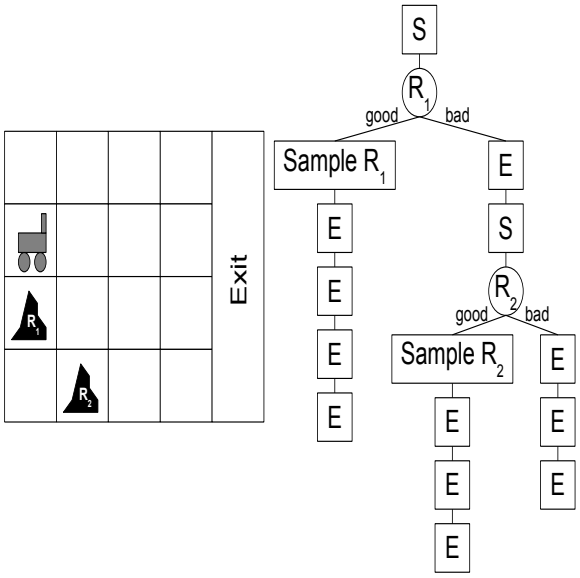

Figure 18: An example of the RockSample(4,2) domain and a contingency plan generated for that problem. The rectangles in the plan are state-changing (mostly moving) actions and the circles are observation-making actions for the specified rock. *S* stands for moving south, *E* stands for moving east, and *R* stands for examining action.

the *RockSample* domain, for example, there is one branch per rock in the problem. This is illustrated in Figure 18. On the left, is an example problem from the *RockSample* domain with a 4x4 grid and two rocks, while the right-hand side shows the plan generated by the contingency planner. We assume the checking action can reveal the true state of the rock. If the first rock turns out to be a bad one, the agent will try to examine the second one; otherwise, it samples the current rock and moves to the exit position.

4.3 EXECUTION MONITORING

The approach we described in Section 4.2 uses FF to generate branching plans relying on a relaxation of the uncertainty in the initial states and observation actions. The results of this are plans that account for every possible state the world might be in but do not account for the

observations needed to discover that state. That is, they are executable only if we assume complete observability at execution time (or equivalently, that the observation actions are perfectly reliable as in DET-POMDPs). If, as is the case in the *RockSample* domain, the sensing is not perfectly reliable and therefore the state is not known with certainty, they may perform arbitrarily badly. To overcome this problem a novel execution monitoring and plan modification approach was proposed to increase the quality of the contingency plan that is actually executed. The execution monitoring approach proposed here can work on contingency plans that are generated by other contingency planners (not just FF), as long as those plans also account for every possible state of the world but do not account for the observation needed to discover the state. During execution, we keep track of the agent's belief state after each selected action via a straightforward application of Bayes rule (Equation 6 in Chapter 2), just as a POMDP planner would. One thing worth noting here is that all the uncertainties from the original Quasi-Deterministic problems are taken into account to update belief state, including initial belief state and noisy observation actions. To select actions to perform when we reach an observation-making action in the plan, we utilise a value of information calculation [52, 92]. Suppose the plan consists of state-changing action sequence a_1, followed by observation action o_1, which measures state variable c. If c is true, branch T_1 will be executed, and if c is false, branch T_2 will be executed. When execution reaches o_1, execution monitoring calculates the expected utility of the current best branch T^* based on the belief state $b(c)$ over the value of c after a_1 as follows:

$$U_b(T^*) = \max_{T_i} U(T_i, b) \tag{19}$$

where $U_b(T^*)$ represents the value in belief state b of making no observations and simply executing the best branch. $U(T_i, b)$ is the expected value of executing branch T_i in belief state b. The number of branches T_i depends on the number of outcomes from the corresponding observation action. Building the complete contingency plan allows us to estimate this value when deciding what observation actions to perform. We do this by a straightforward backup of the expected rewards of each plan branch given our current belief state. The value of $U(T, b)$ (the utility of branch T in belief state b) is computed as follows:

- if T is an empty branch, then $U(T, b)$ is the reward achieved by that branch of the plan.
- if T consists of a state-changing action a followed by the rest of the branch T', then $U(T, b) = U(T', b) - \text{cost}(a)$, that is, we subtract the cost of this action from the utility of the branch.
- if T consists of an observation-making action o on some variable d (observation-making actions for each variable will appear at most once), then $U(T, b) = \sum_d b(d_i)U(T_i, b) - \text{cost}(o)$, that is, we weight the value of each branch at o by our current belief about d.

Next we examine the value of performing an observation-making action o (not necessarily the same o_1 as planned) that gives information about c. Performing o will change the belief state depending on the observation that is returned. Let B be the set of all such possible belief states, one for each possible observation returned by o, and let $P(b')$ be the probability of getting an observation that produces belief state $b' \in B$. Let $\text{cost}(o)$ be the cost of performing action o. The value of the information gained by performing o, is the value of the best

branch to take in each b', weighted by the probability of b', less the cost of performing o and the value of the current best branch:

$$\text{Value Gain(VG)}(o) = \sum_{b' \in B} P(b') U_{b'}(T^{b'}) - U_b(T^*) - \text{cost}(o) \quad (20)$$

Where $T^{b'}$ is the best branch to take given belief state b' according to Equation 19. Both Equation 19 and Equation 20 rely on the ability to compute the utility of executing a branch of the plan, $U(T, b)$.

This ability to estimate the value of each branch is in contrast to the alternative approach of replanning (e.g. see [41], which we discuss in more detail in Chapter 6) where the utility of the future plan is impossible to determine since you cannot be sure what plan will actually be executed until the replanning has occurred. Even in our case, we cannot compute this value exactly as we do not know what additional actions execution monitoring will add to the plan. However, since FF will choose the minimum cost observational action[1] we can be sure that the cost we estimate for the tree by the procedure described above will be an underestimate, thus ensuring that execution monitoring will never perform fewer observational actions than are needed to determine which branch to execute.

For the plan in Figure 18, assuming we get rewards of V^+, 0, and V^- for sampling a good rock, taking no sample, and sampling a bad rock respectively, and costs of C_o for observation actions, C_s for sampling actions and C_m for moving actions, when we reach the observation action for rock R_1 in a belief state b, the value of the "good" branch is:

$$U(T_{good}, b) = b(R_1 = \text{good})V^+ + b(R_1 = \text{bad})V^- - 4C_m - C_s$$

1 This is due to the fact that the determinised versions of the observation-making actions are identical apart from their costs.

while the value of the "bad" branch is:

$$U(T_{bad}, b) = \max \begin{bmatrix} b(R_2 = \text{good})V^+ + b(R_2 = \text{bad})V^- - 3C_m - C_s, \\ -3C_m \end{bmatrix} - (2C_m + C_o)$$

Here the first line of max operator is the value of taking the left branch at R_2, the second line is the value of the right branch, which is simply the cost of moving to the exit without sampling any rocks, and the $(2C_m + C_o)$ is the cost for moving to R_2 and observing its value, which applies to both branches. Note that if the "bad" branch is taken at R_1, then in neither case is any reward gained from R_1, so the belief we have in that rock becomes irrelevant to the plan value. These two equations can be used to compute the value of current best branch $U_b(T^*)$ according to Equation 19. To compute the value gain for an action o, we compare the value of the best branch given our current belief state with the value of the best branch given each possible outcome of o, weighted by the probability according to our current belief state of getting that outcome. Given the current belief state and the observation model from original Quasi-Det POMDP problems, we can easily get probabilities of receiving observations O_{good} and O_{bad} and newly generated belief state b_1 and b_2. Value of performing observation action checkRock1 is just computed as follows:

$$VG(\text{checkRock1}) = P(O_{good}) \times U_{b_1}(T^*) + P(O_{bad}) \times U_{b_2}(T^*) - U_b(T^*) - C_o$$

where $U_{b_1}(T^*)$ and $U_{b_2}(T^*)$ are values of best branch with new belief state and can be computed again using Equation 19.

Our execution monitoring approach works on the contingency plans where branching points are noisy observation-making actions. If a state-changing action is selected to be executed from the contingency plan, we do nothing. If an observation-making action is selected from

the contingency plan, we use the value of information approach (Equations 20) to choose between multiple observation actions by looking at the value gained by every observation action and picking the one that has the highest value. After that action is executed, we continue to choose and execute the best observation-making action until there is no action o with $VG(o) > 0$. At that point, execution selects the best branch and continues by executing it. We might expect that in some circumstances this greedy approach to observation-making action selection could be sub-optimal. The execution monitoring algorithm at an observation action is given in Algorithm 4

Algorithm 4 Execution monitoring at observation-making action o

Let c be the variable being observed by o
Let A be the set of actions that provide information about c
repeat
  Let $VG(a)$ be the value gain for $a \in A$ according to Equation 20
  Let $a^* = \arg\max_a VG(a)$
  if $VG(a^*) > 0$ **then**
    execute a^* and update the belief state b' based on the observation returned
  end if
until $VG(a^*) \leqslant 0$
Execute the best branch given the new belief state b' according to Equation 19

The restriction that execution monitoring can only choose among the observation-making actions is important (if we allow state-changing actions to have non-trivial observations, they may have positive value of information). If execution monitoring was allowed to select actions that changed the state, the rest of the plan might not be executable from the changed state. Because the initial contingency plan was constructed to account for different observation outcomes, changing world state will invalidate the rest of the branch plans. Therefore, this fact limits the applicability of this approach in general POMDPs where actions can have both state-changing and information-getting effects. Suppose when we reach a branch point at execution time for

a general POMDP problem, the current best action is selected according to our value of information approach. Once the new belief state is updated after performing this action, the associate branch plan is not valid any more because not only the knowledge about world but also the actual world state has changed.

4.4 MDP PLANNING APPROACH

One feature of the *RockSample* domain is that observations of the rocks become less reliable the further away from the rock they are made. Our approach of treating the observation actions as completely reliable during plan generation ignores this, so the plans we find tend to observe all the rocks from the starting position rather than driving closer to make better observations. This is an example of a common feature of many quasi-deterministic problems: an observation-making action that requires a state changing setup step. The approach we described in the previous section performs particularly poorly for domains where these are present because, as we have just seen, changing the state of the world might invalidate the rest of the plan.

In this section we present an alternative approach based on computing an MDP policy rather than a contingency plan. This has the advantage that the policy specifies an action to perform in any state, rather than just those that appear in the plan. However, it is computational more expensive than generating a contingency plan because it needs to enumerate all possible states in the world.

The approach we described in Section 4.2 needs to generate contingency plans which are based on determinised version of QDET-POMDPs models. The translation from POMDPs to classical deterministic domains do not consider any probability from the initial state and observation model at planning stage (except choosing the

maximum ones) and needs to add a knowledge predicate *kval()* into the domains. We present a similar approach in this section, which uses an MDP solver to produce an MDP policy for the problem and perform our value of information monitoring technique at run-time to improve the initial MDP policy. It differs from previous translation by taking into account the probability from the initial state and the observation model and not asserting any other knowledge variables. In the past, Q_{MDP}[18] has been a widely used method to generate MDP policies for POMDP problems by ignoring the observation model after one step of belief update. As mentioned in [18], the drawback of the Q_{MDP} is that it treats information gathering actions in the same way as NOOPs actions and often causes the agent to loop forever in the same belief state. NOOPs actions will not change the state of the world and often have zero cost. In order to incorporate observation models into the MDP problems as much as possible, we propose a novel translation from POMDPs to MDPs, which encodes both the initial state and partial observability into the translated MDP domains and allows observation actions to appear in the MDP policies.

4.4.1 *Problem Translation*

A naive approach to applying our execution monitoring technique in an MDP setting would be to convert the quasi-deterministic POMDP into an MDP by simply deleting the observations and initial state uncertainty from the model. The problem with this approach is that it makes all the observation-making actions into NOOPs, so they will never appear in the plan. In the case of the *RockSample* domain, this results in a plan where the rock with good scientific value is immediately sampled and no others are even examined. We would prefer a plan where each rock is investigated, and to do this the MDP plan-

ner needs a notion of what it does and does not know. To achieve this we treat observation actions as actions that switch their respective variables from $unknown$ state to $known$ state, and set all the variables that are initially uncertain (i.e. their probability in the initial belief state is neither zero nor one) to be $unknown$ in the MDP initial state. So for each variable p with domain D, the corresponding variable in the MDP domain, p', has as its domain $D \cup unknown$, and in the initial state $p' = unknown$. We then need to define the transition functions for these observation actions. We use the probability of getting an observation o_i from the initial state as the transition probability from $unknown$ to each value $d_i \in D$ for an observation action a_o:

$$P(unknown, a_o, d_i) = \sum_{d_i \in D} b(d_i) P(o_i | d_i) \tag{21}$$

where o_i is the observation corresponding to value d_i of p.

In the *RockSample* domain, assuming we have a uniform distribution over $rock_0$'s value at the initial state and a sensing action $checkRock_0$ that has 60% accuracy to detect true value of Rock0 at the initial state, which is shown as follows:

```
(rock0value
 (good (ovalue' (ogood (0.6)) (obad (0.4))))
 (bad (ovalue'(ogood (0.4)) (obad (0.6)))))
```

where $ovalue$ is an observation variable in POMDP specification.

For instance, the transition probability of moving $unknown$ to $good$ for variable $rock0value$ computed by Equation 21 looks as follow:

$$\begin{aligned} P(unknown, a_o, S_{good}) &= \sum_{s \in S} b(s) P(O_{good}|s) \\ &= b(good)P(ogood|good) + b(bad)P(ogood|bad) \\ &= 0.6(0.6) + 0.4(0.4) \\ &= 0.52 \end{aligned} \tag{22}$$

A description of the state variables and $checkRock_0$ action in this translated MDP domain looks as follows:

```
(variables
    (rock0value unknown good bad)
    (rock1value unknown good bad)
    (rock2value unknown good bad)
    (rock3value unknown good bad)
)

Action checkRock0
  at_waypoint(SAMEat_waypoint)
  out(outno)
  exit(exitno)
  rock0value (at_waypoint
    (point0 (rock0value
      (unknown (rock0value' (unknown (0.0))
                            (good (0.5))
                            (bad (0.5))))
      (good (rock0valuegood))
      (bad (rock0valuebad))))
    (point1 (rock0value
      (unknown (rock0value' (unknown (0.0))
                            (good (0.5))
                            (bad (0.5))))
      (good (rock0valuegood))
      (bad (rock0valuebad)))))
  rock1value (SAMErock1value)
  rock2value (SAMErock2value)
```

```
  rock3value (SAMErock3value)

  cost (at_waypoint
    (point0 (0.4))
    (point1 (0.6)))
endAction
```

Only when variables have *unknown* value can observation actions take place and assign known values to the state variables according to probability using Equation 21. Initially all the uncertain variables are at *unknown* state.

Note that this translated model makes the assumption that the observation actions are perfectly reliable, so the true value of a state variable is known after performing the corresponding observation action once. However, in reality the observation actions are still noisy, so execution monitoring is still required to improve the plan.

4.5 MONITORING FOR MDP POLICIES

For the MDP case, the execution monitoring is largely the same as in the contingency plan monitoring case described in Section 4.2. The value of information calculation is exactly that given in Equations 19 and 20. The difference comes in the definition of the "branches" in the plan. For the MDP, the branches being chosen are the actions according to the MDP policy for the states corresponding to the possible values of the variable being observed. That is, when we use an observation-making action for a binary variable p, the policies in the MDP states corresponding to $p = \mathit{true}$ and $p = \mathit{false}$ are evaluated. This means that during execution we have to maintain both the belief state according to the original quasi-deterministic POMDP and the current state according to the MDP in order to select actions correctly.

The belief state is used to calculate value of information in order to select appropriate observation action as shown in Equation 20 and current discrete state of MDP is used to generate the appropriate branch plan from the MDP policy.

In the case that no observation action has greater than zero information gain, as before, we pick the best "branch" of the plan to execute given our belief state. For the MDP, we use the policies for all the MDP states that have non-zero probability of occurring according to the MDP translation of the observation-making action. That is, if the MDP policy for the current state s and current belief state b says do observation-making action o (information gain of o is greater than zero), which observes a state variable p with domain D (which must be *unknown* in the current MDP state or the policy would not choose an observation-making action), after getting an observation value at run-time, the belief state is updated to b' according to the original POMDP model. If our monitoring algorithm decides no other observation actions are necessary by making sure the information gain of all the observation actions is less than zero, then we compute the expected value given our current POMDP belief state b' for each branch according to Equation 19 and select the best branch to follow.

4.5.1 *Macro Actions*

As stated above, the major difference between the output of the contingency planner in Section 4.2 and the MDP planner is that the MDP planner provides an action for every possible state. This means that we can allow execution monitoring to perform state-changing actions, and we will still know what future policy should be performed in the state that results. This is important because it allows us to make observations that require setup actions. To achieve this we allow ex-

ecution monitoring to evaluate the information gain from macro actions consisting of a state-changing action followed by an observation-making action. In the literature, macro actions usually consist of an sequence of actions that will appear multiple times in solutions [21]. Many learning techniques were proposed to generate macro-actions so that the plan can reuse these actions when it faces a similar situation. Off-line learning techniques, such as Macro-FF [11], generate macro-actions from a set of training examples before planning process, while on-line learning techniques, such as Marvin [20], identify macro-actions during the search process. Both techniques need to memorise existing macro actions in order to guide the search later on. In this thesis, macro actions mean we can calculate information gain of two types of action. One is an observation-making action as shown in Equation 20, the other is a sequence of a state-changing action followed by an observation-making one. For example, in the *RockSample* domain this allows execution monitoring to calculate the value of information gained from moving one step towards a rock before observing it, thus making a more reliable observation of the rock. The information gain for a macro action made up of state-changing action a followed by observation-making action o becomes:

$$VG_{b'}(a + o) = R(b', a) + VG(o) \tag{23}$$

Once we decide the current best action, the two situations need to be considered separately as well. If the current best action is a macro action, and hence the immediate action to execute is a state-change action, we simply execute this action and repeat the above procedure. Since it is possible that a macro action will again be best at the next step, we can execute a sequence of state changing actions before making an observation, for example, taking a series of steps closer to a rock before observing it. If the current best action is an

observation action, this tells us that there is no better macro action. If this action has value gain greater than zero, we again execute it and repeat. Otherwise, we pick the best policy given our current belief state as described above. The algorithm is given in detail in Algorithm 5.

Algorithm 5 Execution Monitoring with Macro Actions

Let the current MDP state be s
Let o be the observation-making action which is the policy for s, where the observation is of variable V
Let b be the current belief state
repeat

$$a^* = \arg\max_a \begin{cases} VG(a), \\ \text{if } a \text{ is an observation action} \\ R(b,a) + VG(o'), \\ \text{if } a \text{ is a macro action } (a,o') \end{cases} \tag{24}$$

  if a^* is a macro action (a,o) **then**
    Execute action a and update the MDP state s and the belief state b
  else
    Execute action a, getting observation Z
    $b = \text{beliefUpdate}(b,a,Z)$
  end if
until $VG(a^*) < 0$
Let S be the set of MDP states
$\{s' : s' = s - (V = \text{unknown}) \cup (V = v)\}$
Let π_s be the policy at state $s \in S$
Let a^* be the first action in policy π^* where:

$$\pi^* = \arg\max_{\pi_s} \sum_c b(c)U(\pi_s, c) \tag{25}$$

where c is the POMDP state space
Execute action a^*
$b = \text{beliefUpdate}(b, a^*)$

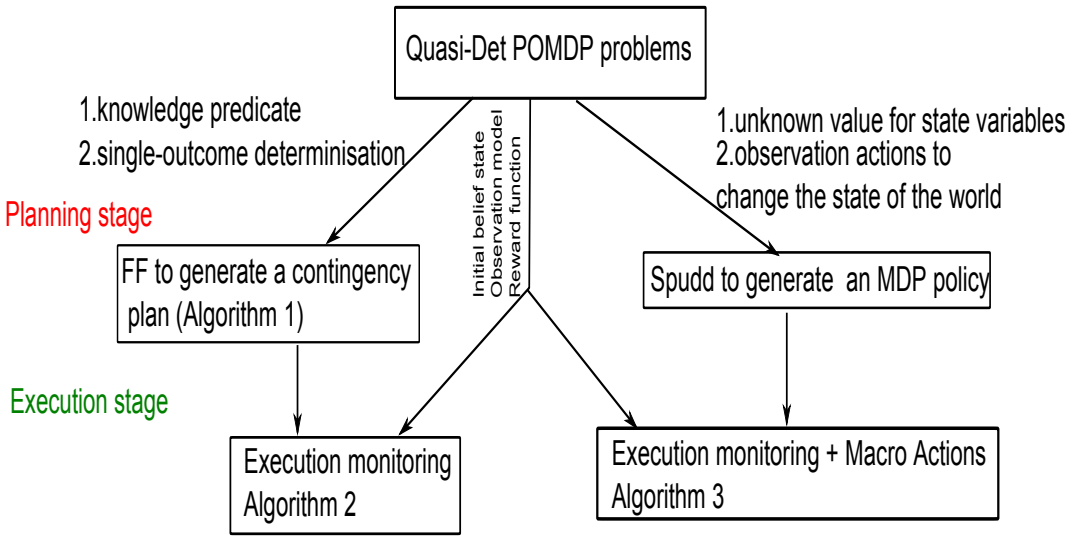

Figure 19: A diagram of the complete planning and monitoring process for QDET-POMDPs.

A diagram to describe the above two execution monitoring approaches on Quasi-Deterministic POMDPs is shown in Figure 19. The true initial state and observation actions are used to update the belief state at execution time and we apply value of information monitoring techniques to improve the original approximate contingency plan or MDP policy.

4.6 EXPERIMENTAL EVALUATION

We test our approaches on the classical POMDP problem *RockSample* [102] and a modified version of the *HiPPo* domain from [106], which involves finding objects in a scene using vision algorithms.

For comparison purpose, we use Symbolic Perseus [83], a state-of-the-art point-based solver that uses a structured representation to solve POMDPs. This algorithm is only approximately optimal and the quality of the policies found depends strongly on how the points for the approximation are selected. However, having run the algorithm with a range of different parameters, we are confident that the policies reported are very close to optimal. Besides, all the sensing

models, the costs of the actions, and the reward functions of the problmes are tuned in Symbolic Perseus so that the POMDP solver can produce sensible plans, they are not selected in favor of our translation approaches. That is the main reason I did not perform a broader range of the experiments. After translating the problem as described in Sections 4.2 and 4.4, we use FF [51] and SPUDD [48] respectively to generate contingency plans and MDP policies. The standard *RockSample* problem allows any rock to be observed from any position with noise depending on the distance between the rover and the rock. This makes the translation between the symbolic Perseus domain specification and PPDDL for contingency plans quite difficult, so we only perform contingency plan execution monitoring in the *HiPPo* domain. We tested *RockSample* domain with two different initial states, one is all the rocks have 0.5 probability of being good, the other is all the rocks have 0.7 probability of being good. If all the rocks have less than 0.5 probability of being good initially, our translation approaches will not generate the plans that examine any rocks at the first place, therefore, there will be no any branching points for executing value of information at run time. We also performed MDP execution monitoring with and without macro actions as described in Section 4.4 in the *RockSample* domain (macro actions give no advantage in the *HiPPo* domain).

To measure plan quality, we compute total reward and discounted reward for both domains. Generation time is recorded, as well as average execution time per run. On the *RockSample* domain, evaluation is done over 200 runs, on each of 200 steps. Since *HiPPo* domains have a larger belief space and do not return to initial state after reaching the goal, we performed 1000 runs on each of 20 steps except for *HiPPo*(5,4) where only 100 runs were performed due to long runtimes. We also compared with Q_{MDP} [18] in both domains. Since ob-

servation actions in our domains are not changing states at the same time, Q_{MDP} would treat them as do-nothing actions for MDPs and performed poorly in both domains. Both translation approaches are implemented in C++. They were compiled with g++ v4.1.2. All the experiments were performed on a PC with 2.33 GHz Intel processor and 2GB memory.

4.6.1 *RockSample*

As Table 5 shows, symbolic Perseus obtained the best plans in terms of total reward and discounted reward for all *RockSample* problems, which is expected as it computes approximately optimal policies for the initial belief state. However, the MDP solutions required much less time to generate the policy. As illustrated in Table 5, adding execution monitoring can substantially improve plan quality, and allowing execution monitoring to choose macro actions also significantly improves plan quality in terms of total reward and discount reward. Although it requires more computation at run-time, the overhead of execution monitoring with macro actions is still far less costly than solving the POMDP directly. This domain also shows the importance of the macro actions where setup actions are needed. Because our implementation of PPDDL does not support functions, it consequently lacks ability to represent observation actions in *RockSample* problems where observation accuracy depends on the distance between the rover and the rock and its inability to use macro actions would have severely restricted its effectiveness (we would expect it to work approximately as well as the MDP solver without macro actions). Table 6 shows a similar result of initial state being $(0.7, 0.3)$, where the MDP solutions, whether with or without execution monitoring are generally closer to the optimal solution compared to results in uniform ini-

Table 5: Results for the *RockSample* Domain comparing symbolic Perseus (POMDP) with the MDP approach(initial state [0.5,.0.5]).

Algorithm	Gen. Time (s)	Exec. Time (s)	Total Reward	Disc. Reward
RS (4,4)				
POMDP	274	0.6	251.8 $\pm$ 69.1	6.9 ± 8.4
MDP with macro actions	13	3.5	161 ± 61.5	2.3 ± 9.5
MDP with EM	13	1.8	141 ± 61.3	1.9 ± 6.9
MDP without EM	13	1.1	41 ± 107.3	-3.5 ± 13.3
RS (5,5)				
POMDP	893	0.8	213 ± 66.8	4.5 ± 8.0
MDP with macro actions	99	4	179 ± 77.3	2.9 ± 10.9
MDP with EM	99	2.4	96 ± 70.2	-0.3 ± 8.0
MDP without EM	99	1.2	61 ± 100.5	-2.8 ± 13.3
RS (6,6)				
POMDP	1098	1.3	188 ± 55.5	3.4 ± 1.8
MDP with macro actions	476	5.0	150 ± 78.5	0.5 ± 9.3
MDP with EM	476	2.6	137 ± 73.3	1.9 ± 10.9
MDP without EM	476	1.7	58 ± 101.3	-2.1 ± 12.8
RS (7,7)				
POMDP	3520	2.4	154 ± 52.1	2.6 ± 8.5
MDP with macro actions	2096	7.2	147 ± 82.3	-0.2 ± 10.7
MDP with EM	2096	3.5	125 ± 75.5	-0.7 ± 10.7
MDP without EM	2096	4.5	78 ± 106.9	-2.2 ± 13.4

Table 6: Results for the *RockSample* Domain comparing symbolic Perseus (POMDP) with the MDP approach (initial state [0.7,.0.3]).

Algorithm	Gen. Time (s)	Exec. Time (s)	Total Reward	Disc. Reward
RS (4,4)				
POMDP	342	0.29	455 ± 64	11.63 ± 6.05
MDP with macro actions	13	3.11	334 ± 68	9.31 ± 8.53
MDP with EM	13	1.28	363 ± 61	10.46 ± 6.56
MDP without EM	13	1.06	254 ± 110	7.72 ± 13.71
RS (5,5)				
POMDP	1052	0.42	406 ± 61	10.05 ± 5.83
MDP with macro actions	99	7.55	325 ± 74	8.47 ± 8.77
MDP with EM	99	1.52	273 ± 94	6.63 ± 13.63
MDP without EM	99	1.20	269 ± 105	6.77 ± 13.82
RS (6,6)				
POMDP	2115	1.08	377 ± 65	10.65 ± 4.35
MDP with macro actions	476	4.50	342 ± 86	9.38 ± 9.99
MDP with EM	476	0.52	270 ± 28	6.04 ± 10.08
MDP without EM	476	1.65	196 ± 101	4.07 ± 15.09
RS (7,7)				
POMDP	3431	2.85	430 ± 64	10.37 ± 4.85
MDP with macro actions	2096	6.9	392 ± 76	7.61 ± 10.61
MDP with EM	2096	3.94	222 ± 124	5.65 ± 11.08
MDP without EM	2096	3.35	226 ± 124	2.66 ± 16.01

tial state. The reason is that all the rocks are more likely to be good, so there is less likely to be false negative from observation action which would result in a large penalty.

4.6.2 *HiPPo*

We also tested our approach in a modified *HiPPo* domain [106]. Originally, the *HiPPo* problem is to find a sequence of visual actions to apply to the regions of interest (ROIs) in a scene in order to for a robot to answer queries, such as "where is the red triangular object". Each object in the domain has both color and shape properties, and there are five different values for each. For color property, the underlying class values could be *red*(R), *green*(G), *blue*(B), *empty*(E) or *multiple*(M). For shape property, the underlying class values could be *circle*(C), *triangle*(T), *square*(S), *empty*(E), or *multiple*(M). *Empty* label means there is no match to any of these values and *multiple* indicates there might be more than two labels to the objects. Two observation actions, *Color* and *Shape*, can be applied to detect the state of each object with some sensing noise, the observation domain Ω is $\{E_c^o, R_c^o, G_c^o, B_c^o, M_c^o, E_s^o, C_s^o, T_s^o, S_s^o, M_s^o\}$. The question is how many color and shape detecting actions need to be applied before we can determine the properties of the object. Originally, the objects are all shown in one scene (e.g. on a table), so the planner also needs to decide which object (ROI) to look at. We modified the problem by putting the objects in a grid map, and making sensing actions usable only when the agent is at the same position as the object. If the agent is not at the same position as the object and executes one of the sensing actions, either E^o or M^o observation only will be received with 0.5 probability. Again, *HiPPo*(n, k) denotes a n by n grid with k objects. Therefore, in order to decide which object has desired prop-

Table 7: Results for the *HiPPo* Domains comparing symbolic Perseus (POMDP) with the MDP and the contingency planning (FF) approaches.

Algorithm	Gen. Time (s)	Exec. Time (s)	Total Reward	Disc. Reward
HiPPo (3,2)				
POMDP	196.25	0.26	-7.35 ± 29.9	-2.13 ± 12.0
MDP with EM	1.04	1.42	-7.45 ± 17.3	-2.28 ± 9.5
MDP without EM	1.04	0.06	-6.57 ± 15.4	-2.83 ± 9.8
FF with EM	4.04	1.32	-7.04 ± 17.3	-3.04 ± 8.9
FF without EM	4.04	0.06	-7.41 ± 16.9	-2.90 ± 9.3
HiPPo (4,3)				
POMDP	4059	9.95	-3.26 ± 11.5	-1.21 ± 10.6
MDP with EM	11.88	4.03	-3.75 ± 23.3	-1.21 ± 11.43
MDP without EM	11.88	0.13	-5.78 ± 21.5	-1.69 ± 12.5
FF with EM	8.05	3.48	-5.79 ± 18.8	-1.87 ± 9.8
FF without EM	8.05	0.12	-5.95 ± 18.5	-1.87 ± 9.7
HiPPo (5,4)				
POMDP	-	-	-	-
MDP with EM	207.65	56.74	-3.60 ± 27.5	-0.61 ± 11.1
MDP without EM	207.65	0.46	-2.31 ± 9.8	-1.42 ± 22.6
FF with EM	16.15	37.38	-3.24 ± 22.2	-0.80 ± 10.4
FF without EM	16.15	0.44	-3.30 ± 20.7	-2.01 ± 9.4

erties, the agent also needs to move to the place of the object and execute sensing actions. State changing actions in this domain consist of four moving actions and termination actions. Termination actions appear when the agent has gained enough information about the objects and is able to answer the query. For example, if the query is “where is the red triangular object", the termination action can be “say red" and “say triangular" on an object, which terminates the process.

Since *HiPPo* domains have a large state space ($n^2 \times 5^{2k}$) and a large observation space 10, problems which are larger than size (4,3) cannot be solved by symbolic Perseus within reasonable time (two hours) and memory usage. As Table 7 shows, although symbolic Perseus still managed to achieve the best plan quality in terms of discounted reward, it requires orders of magnitude more time in generating policies. In this domain the MDP policies are quite good (they run the observation actions once on each object, while the optimal policy is to run them twice if they return the value you are looking for, to ensure reliability), so there is less improvement from adding execution monitoring. The interesting result is the performance difference between the contingency plans and the MDP planner. This is largely because the contingency planner only plans for the initial state while the MDP policy is for every possible state. When we perform observation actions and the robot is not in the same position as the object, it is only possible to get two observation outcomes, which are E and M , so there is no transition probability of turning the color property of an object from $unknown$ to R, G, or B in MDP domains, which would make the robot always move to the place where the objects are and perform observation actions. In this case, macro actions are not necessary because it is always the time to do information gathering when the robot is in the same position as the object. For these domains, because our MDP policies and contingency plans are already

quite similar to the optimal POMDP solutions, they all had unsurprisingly good reward. Furthermore, it also shows that solutions with our execution monitoring can still improve original plans or policies at run-time.

4.7 CONCLUSION

In this chapter we have presented an approach to solve QDET-POMDPs which uses a mixture of off-line planning and on-line planning. At the planning stage we removed the stochasticity from the actions to generate completely observable planning problems, then we produced plans using either a classical contingency planner or a Markov decision problem solver. At execution time we monitord the actual belief state of the agent as the plan was executed, and re-evaluate in light of that belief state. We used a value of information calculation to determine if there were information gathering actions that would change the belief state in such a way as to improve the expected quality of the remainder of the plan and, if so, we added them to the execution of the plan.

For contingency planning, QDET-POMDPs were determinised using most-likely determinisation methods. The state with the largest probability from the initial state was chosen and only the most likely outcome was selected for stochastic observation actions. The translated problem was then fed to the classical planner FF to generate a sequence of actions. The contingency plans were then constructed by enumerating all the possible outcomes from the observation actions which appear in the straight-line plan. At run-time, we put extra execution monitoring to repair the plans by making the choice of sensing actions. A value of information wass applied to monitor the belief state and determine the observation actions that could gain most in-

formation about the current belief state. The trade-off we mad here was how much uncertainty we could remove by applying this observation action against the cost of this observation action. The heuristic value of executing this observation action will be computed by considering all possible branches from the rest of the plans and the reward we collected from the node of the contingency plans will be backtracked to the root of the contingency which is our current monitoring point. We have shown in the experiment section that greedily selecting observation actions can improve the original approximated contingency plans. Although the approach does not produce a better policy than a state-of-art POMDP solver, a lot of the planning time is saved by using a classical planner.

One limitation of the contingency plans is that our execution monitoring will only decide whether or not to continue executing the current observation action, and it will not change state of the world which will invalidate the rest of the plans.

The second part of the work translated the QDET-POMDPs into MDPs. This approach has some advantages over the contingency approach. First of all, there are more similarities between MDPs and POMDPs so the translation is easier than classical contingency planning. Because state-changing actions are non-informative and are also deterministic we can represent these actions in the same way as in an MDP. As for initial states, we assigned all uncertain state variables with a discrete state value “unknown", so that observation actions in the MDP can make these variables transit from the unknown state to a known state with a certain probability. Our experimental results showed that our approach is fairly close to optimal but is orders of magnitude faster than using a POMDP solver. The comparison between contingency and MDP approach also demonstrated that the

latter usually takes more time to generate a policy which covers all states in the space.

5

EXECUTION MONITORING ON POMDP POLICES

In the previous chapter, we addressed the issue of planning in Quasi-Deterministic POMDPs which is a subset of general POMDPs where only observation actions have stochastic outcomes. Execution monitoring is done at run-time by greedily modifying approximate solutions using value of information techniques. The main idea is that we are generating off-line solutions using a deterministic planner or an MDP planner rather than a POMDP solver, and enhance the solutions at execution time. In this chapter, we are going to tackle execution monitoring of general POMDPs where there is no longer a distinction between observation-making actions and state-changing actions. A similar idea is adapted for general POMDPs. Since finding optimal policies for large POMDPs using exact solvers is problematic, many approximate POMDP solvers are proposed so as to be able to produce a good solution within reasonable time. Our execution monitoring procedure then tries to improve these good solutions at run-time. The goal of execution monitoring for approximate POMDP solvers is using extra computation for replanning at run-time in order to get a better policy in the end. Unlike the work in the previous chapter where monitoring was triggered automatically whenever it encountered observation actions, the correct timing of replanning needs to be done to decide when the current policy is not working properly for the current belief state. Therefore, the main contribution of this work is proposing several cheap heuristic functions to decide when to replan at execution time.

As explained in Chapter 2, partially observable Markov decision processes (POMDPs) are powerful models for capturing uncertainty in both action outcomes and observation variables. Exact solutions such as value iterations, as demonstrated in Chapter 2, are introduced to generate optimal policies off-line. Unfortunately, finding optimal policies for large POMDPs is intractable due to two curses: the curse of dimensionality and the curse of history. The curse of dimensionality requires POMDP solvers to compute optimal policies in an $n-1$ dimensional continuous space if there are n states in the planning domain. The curse of history means the number of possible action and observation combinations grows double exponentially with the planning horizon. Exact solutions will suffer from these two curses at the same time.

In response to the two curses, many approximate POMDPs solvers have been proposed in order to generate good solutions within reasonable time. In this work, we focus on point-based POMDPs solvers [102] which utilize the idea of finding optimal policies in a representative belief space which is finite, rather than in the entire continuous belief space. This can be seen as one of the solutions to tackle the curse of dimensionality. Point-based algorithms differ from grid-based algorithms [47] in their selection strategies for belief points. Point-based approaches sample from simulated forward trajectories, while grid-based algorithms sample points uniformly. SARSOP [61], one of the most promising point-based POMDPs solvers, takes this one step further by generating policies in a reachable belief space under optimal policies. However, most point-based algorithms tend to generate sample points with high probabilities to transit to and ignore less likely belief points along the planning trajectory. In this chapter, we would like to detect the situation at execution time where the current belief point is not well sampled which results in bad performance with re-

spect to policy quality. Furthermore, a replanning procedure is called when such a situation occurs in order to produce a better policy for our current belief state.

Take a factory domain for example, suppose we have an assembling machine which is trying to assemble different parts of components for a working product. During the assembly procedure, each component in the system has a small possibility of being damaged which would result in the failure of the whole assembly action. There are noisy observations about the current state of the components and also recovery actions that can repair damaged components. This problem can be formulated as a POMDP problem. As mentioned before, if we choose state-of-the-art point-based algorithms to solve this POMDP problem, because each component only becomes faulty with a very low probability, it is quite likely that point-based solvers will not sample belief points in these areas but focus on states that are more likely to occur such as the components are all working correctly in this case. When we do encounter faulty components at execution time, there will be no appropriate recovery actions available for us to execute because these repair actions are not included in the initial policy. In our approach, we would like to investigate the idea of detecting such a situation and perform replanning at run-time to increase the overall quality of our policy. In the factory domain, when a component accidentally becomes damaged at run-time, our execution monitoring approach will identify this situation and include the appropriate repair actions into the existing policy after replanning.

Replanning is done by performing normal backup operations on newly sampled belief points at execution time. Even though exact backup operations are computationally expensive at run-time [45], we only trigger those operations in a less frequent way compared to other on-line POMDPs solvers which will compute the current best

action at every time step. We showed in the experiments Section 5.3 that it can outperform state-of-the-art POMDP solvers in terms of plan quality and computation time when both planning time and execution time are considered. This can be viewed as complementary for point-based algorithms, which can generate reasonably good policies for most of the domains, but not for the domains where low probability transition exists.

5.1 POINT-BASED ALGORITHMS

Let us recall the POMDP model from Chapter 2, where a POMDP is specified as a tuple $\langle S, A, T, \Omega, O, R, \beta \rangle$. The goal of a POMDP solver is to find an optimal policy that can maximise the expected discounted reward, which is defined as follows:

$$E[\sum_{t=0}^{\infty} \beta^t R(s_t, a_a)] \tag{26}$$

where β is a discount rate, $0 \leqslant \beta < 1$ and s_t and a_t denote the agent's state and action at time t. As described in Chapter 2, the optimal policy π^* maps any belief point b from the belief space B into a particular action $a \in A$ and also induces an optimal value function $V_{\pi^*}(b)$ that is computed as the expected discounted reward for following this optimal policy π^*.

In Chapter 2 we described one of the most popular POMDP solvers which is value iteration. This classic method represents a value function $V(b)$ as a set of vectors $\alpha_0, \alpha_1, \ldots$ which are piecewise linear and convex. The value of a belief point b can be computed as follows:

$$V(b) = \max_{\alpha} b \cdot \alpha \tag{27}$$

Given a value function V, the best action for the current belief point b can be extracted as follows:

$$\pi_V(b) = \arg\max_a \alpha_a \cdot b \tag{28}$$

where each α-vector defines the expected future reward, starting with action a and then continuing to execute the action with the highest α-vector in subsequent belief states.

To compute optimal α-vectors, one of the key operators in POMDP solvers is to build n-horizon value functions from $n-1$-horizon ones using the backup operator H:

$$V_n = HV_{n-1} \tag{29}$$

$$= \max_a \left[\sum_{s \in S} R(s,a)b(s) + \beta \sum_{o \in \Omega} P(o|a,b)V_{n-1}(b')P(b'|b,a,o) \right] \tag{30}$$

where $P(o|a,b) = \sum_{s,s' \in S} O(s,a,s,o)b(s)T(s,a,s')$ is the probability of getting observation o when doing action a in belief state b.

For each observation o, there is only one deterministic transition from the current belief state b to a new belief state b', so $P(b'|b,a,o)$ is an indicator function which can be computed via Bayes rule.

Since the belief space is continuous and high-dimensional, generating α-vectors over the entire space is computational expensive. However, the computation time of the backup operation for a single belief point is $O(|S|^2|A||O||V_{n-1}|)$ where $|V_{n-1}|$ is the number of α-vectors in previous set V_{n-1}, which takes only polynomial time. Point-based value iteration was introduced to take advantage of this by planning in a representative subset of the belief space. If the size of this sub-

Algorithm 6 SARSOP

```
Initialize V̲ and V̄
repeat
  SAMPLE(b_0, V̲, V̄)
  for all b ∈ B do
    V̲ ← BACKUP(b,V̲))
    V̄ ← BACKUP(b, V̄)
    PRUNE(V̲), V̄)
  end for
until V has converged
```

set is constrained, the number of α-vectors in the value function is also limited, because one belief point at most is mapped to a best α-vector. When it comes to backup operations, we only need to backup α-vectors that dominate at our representative belief points rather then searching for α-vectors for the entire space.

SARSOP [61] use both lower bounds and upper bounds to approximate value functions in POMDPs. As you can see from the Algorithm 6, there are three main functions in SARSOP approach, viz, SAMPLE, BACKUP and PRUNE. It first samples a set of points from the belief space using the forward exploration heuristics and updates each point's upper bound and lower bound locally in a reverse order. The pruning techniques are applied to reduce the computation time on backing up α-vectors as lower bounds. During the sampling stage, it uses an action selecting strategy: picking the action with the highest upper bound:

$$a^* = \arg\max_a Q_a^{\bar{V}}(b) \tag{31}$$

As mentioned in [57], selecting actions with the greatest upper bound can guarantee convergence because once its upper bound is lower than other action's upper bound, we can discover this action's

sub-optimality. This does not hold if we select actions with highest lower bound because lower bound is always increasing afterwards.

As for observation strategy, it selects the observation that makes the largest contribution to *excess uncertainty* at parent node. An excess uncertainty is defined as follows:

$$excess(b, t) = width(\hat{V}(b)) - \varepsilon\beta^{-t} \tag{32}$$

where $width(\hat{V}(b))$ is the value difference between the upper bound $\bar{V}$ and the lower bound $\underline{V}$. This excess uncertainty defines how far the current bound is from the terminated condition $\varepsilon\beta^{-t}$. Therefore, the observation o^* is selected as follows:

$$o^* = \arg\max_o [p(o|b, a^*) excess(\tau(b, a^*, o), t + 1)] \tag{33}$$

Thisese action selection strategy and observation selection strategy in SARSOP are the same as in HSVI [102, 103]. The belief points sampled in SARSOP are in reachable belief space, what is more, SARSOP proposed the idea of generating belief points which are in the *optimally reachable belief region* at belief expansion stage. This belief region consists essentially of the belief points we are going to visit under optimal policy. So it is more compact than other sets of belief points that are generated by random walk or other selection strategies. Since we do not know the optimal policy at the sampling stage, SARSOP applies a simple on-line learning technique to predict the optimal value $V^*(b)$ during sampling. This clustered the belief points into several discrete belief spaces according to their upper bound and entropy. Suppose current expanding node b is in a discretized belief region r_i, the predicted optimal value of node b is the average value of belief points in r_i. This estimate value can be used to determine whether to

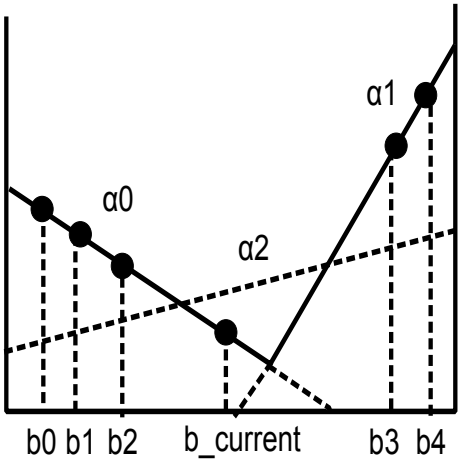

Figure 20: Point-based value iteration needs to interpolate belief point from the sampled one. In this example, b_0, b_1, b_2, b_3 and b_4 are sampled points at planning stage. $b_{current}$ is the belief point encountered at run-time. Current policy includes α_0 and α_1. α_2 is a potentially better α-vector which we would like to find at run-time for $b_{current}$. This figure is reproduced from [81]

continue expanding the current node or not. By doing so, SARSOP can focus planning on the most likely belief states in the space.

SARSOP and other point-based algorithms (shown in Chapter 6) are trying to reduce computation cost by working on a subset of the belief space, and empirical results show that previously unsolvable large POMDP domains can be tackled using these techniques. However, since the value for belief points other than those sampled is interpolated from the sample points, the policies can be much worse for points far from the sampled ones. Since the full belief space is too large to be densely sampled, many point-based algorithms choose sample points with a high probability of being reached and ignore less likely belief points. At run-time, point-based algorithms will compute the best α-vector from our existing vector set for the current belief point at each time step. Thus, when the current belief point, or its neighbour, is not sampled during planning stage, its performance is going to be poor, especially in the domain where there are many low probability action outcomes;for example domains with exogenous actions, or where numerous low probability faults can occur. This prob-

lem could occur in all approximate off-line solutions for POMDPs where at run-time there is no mechanism to improve the existing solutions when the current solution is not suitable for the current belief point. Figure 20 shows a set of α vectors and its representative point set $B = \{b_0, b_1, b_2, b_3, b_4\}$ which are used to generate the policy. As we can see from the figure, our current policy only includes two vectors α_0 and α_1, which are generated by using point set B. When, at execution time, our current belief point $b_{current}$ is far from all the points we used to generate the vectors, it is time to ask the question "Is there a better policy to follow at this point", because there might be a better α-vector (α_2) for current belief point.

5.2 EXECUTION MONITORING

We are proposing execution monitoring on point-based algorithms to address the issues that some belief points might not be well covered by the off-line generated solutions. As explained in the previous section, traditional point-based algorithms simply compute the best α-vector from the approximated solution for the current belief point at each time step and ignore the question of whether the current sub-optimal solution is good enough or not.

Generally, there are two parts in our execution monitoring. The first is applying heuristic measurements to estimate the quality of the current solution. Such information is difficult to compute for traditional point-based POMDP solvers because α-vectors are the only output for point-based algorithms and there is no additional information about how those off-line solutions were generated when we are executing the polices at run-time. Therefore, additional information such as the belief points used for generating those approximate solutions need to be stored along with the α-vectors.

There should be some requirements for these heuristic measurements. One is that they should be easy to compute. Since these measurements need to be computed at every step of plan execution, expensive heuristic functions will put too much computation cost on running the policies. Second, they should provide the best information about whether the current policy is doing well for the current belief point, this is the reason we propose several measures in this section. During execution, we introduce a threshold θ to decide when to trigger the replanning. By doing experiments in different domains, we would like to see whether this threshold parameter is domain independent or whether general values can be established for all the domains.

The second part of our plan repair strategy is replanning when our heuristic functions indicate that the current policy for our current belief point should be improved. A naive replanning from scratch would be too costly. Instead, we treat the current belief point as a new initial state, use the already computed lower and upper bounds provided by the previous invocation of SARSOP, and ask SARSOP to generate new belief points and a new policy. To do this we not only need to store belief points from the off-line stage, but we also need to keep a record of the upper bounds of the optimal policy: the set of belief points and their upper values. Newly generated belief points will be added to the existing belief point set which can be used for determining the quality of the policy for subsequently encountered belief points. In addition, new lower bounds and upper bounds are computed as part of replanning. We limit the time available to SARSOP to prevent replanning taking too long. The plan repair algorithm is displayed in Algorithm 7.

We now describe the several candidate heuristics, and compare their performance in Section 5.3.

Algorithm 7 Execution Monitoring on PBVI

Let b be the initial belief state
Let B be the sampled belief point set from off-line SARSOP generation.
Let $\underline{V}$ be the set of α-vectors from off-line SARSOP generation.
Let $\bar{V}$ be the pair of belief point and its upper value from off-line SARSOP generation.
Set θ and τ where θ is the threshold value for triggering replanning and τ is the replanning time constraint.
repeat
  if Measurement(b, B) $> \theta$ **then**
    $B' \ \underline{V}' \ \bar{V}' \leftarrow$ SARSOP (b,τ, B, $\underline{V}$, $\bar{V}$) {Needs to replan} { Run SARSOP for τ seconds for current belief b }
    $B \leftarrow B'$ {store newly generated belief set}
    $\underline{V} \leftarrow \underline{V}'$ {store newly generated lower bound}
    $\bar{V} \leftarrow \bar{V}'$ {store newly generated upper bound}
  end if
  execute from $\underline{V}$
until reach of plan horizon

5.2.1 *Gap heuristic*

The first heuristic function is using the gap between the upper bound and the lower bound of the current belief point b as shown in Equation 34. The intuition is that if the gap is too large, there is more opportunity of there being a better policy to reduce the gap. However, this heuristic function only works with point-based algorithms that generate both upper bounds and lower bounds such as SARSOP or HSVI. Therefore, we also proposed other heuristic functions that can work on general point-based algorithms.

$$M_{gap}(b) = V_{upper}(b) - V_{lower}(b) \tag{34}$$

Theorem 1. *The time complexity of gap heuristic is* $O(|\Gamma|)$ *where* Γ *is the set of* α*-vectors.*

Proof. For any b', $V_{lower}(b') = \max_{\alpha \in \Gamma} b' \cdot \alpha$ □

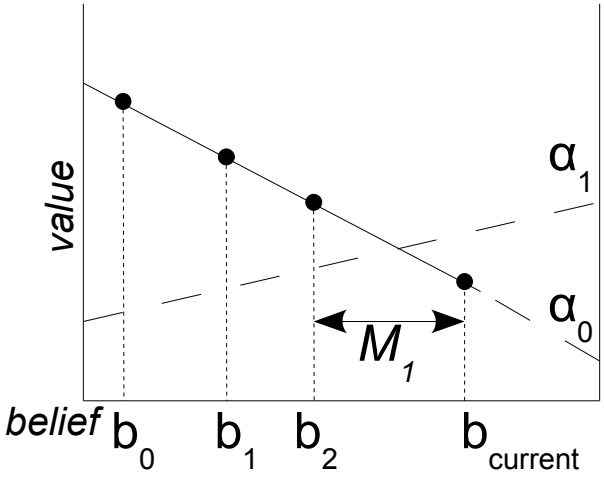

Figure 21: L_1 distance measurement.

5.2.2 L_1 *Distance*

The second heuristic we use is simply the L_1 norm distance[1] from $b_{current}$ to the nearest point in B where B is the current belief point set. Intuitively, if the current belief point is far away from any points that are used to generate the initial policy, it is quite likely that we can find some better α vectors for the current belief. Figure 21 demonstrates this idea. b_0, b_1, b_2 and $b_{current}$ share the same best α-vector. We use an L_1 metric to measure the distance between the points, so in this case the distance between $b_{current}$ and b_2 is computed to make a decision about whether to trigger plan repair.

Theorem 2. *The time complexity of* L_1 *heuristic is* $O(|B|)$.

Formally, the L_1 heuristic for approximating the value function error at some belief point b given belief point set B is computed as follows:

$$M_{L_1}(b, B) = \min_{b_i \in B} \|b - b_i\|_1 \tag{35}$$

1 The L_1 norm distance is the sum of the absolute value of the distance between two vectors.

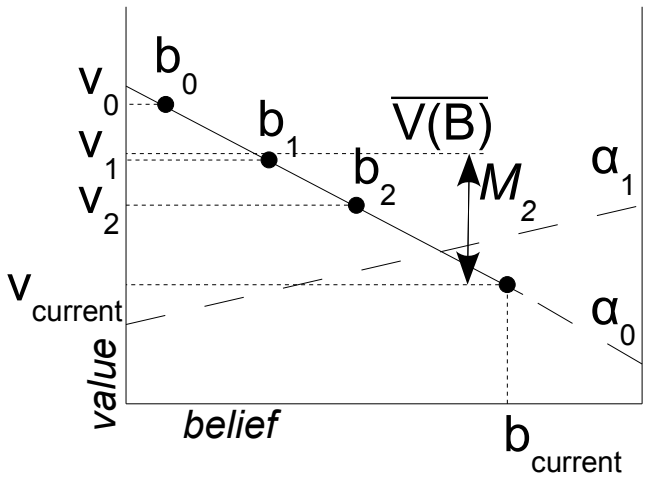

Figure 22: Value distance measurement

If $M_{L_1}(b, B) \geqslant \theta$, we execute our replanning operations where θ is the pre-defined threshold parameter. The best value of θ and replanning time τ are tuned for all the benchmarks at the experiment.

5.2.3 *Value Difference*

The third heuristic we propose is based on the difference between the value of the best α-vector at b and the average value of all the belief points B_α that share the same best α-vector. To produce a scale-invariant heuristic we divide this difference by the average value of the belief points that share the same α-vector. This is again thresholded to trigger plan repair. Figure 22 illustrates this second heuristic measurement. Our current belief point $b_{current}$ shares the same α-vector α_0 with belief points b_0, b_1 and b_2. The intuition is that since the current belief point shares the same α-vectors with these points, it should come from a similar region of the belief space and should have a similar value. Therefore, the average value of belief points b_0, b_1 and b_2 is computed as $v_{average}$ and compared with current value $v_{current}$. Figure 22 shows that the value of the current belief point is

far from the average of the other points and possibly indicates there is a need for replanning.

Formally, the lower bound value measurement is computed as follows:

$$M_{val}(b, B) = \frac{\|V(b) - \overline{V_\alpha(B)}\|}{\overline{V_\alpha(B)}} \tag{36}$$

where $\overline{V_\alpha(B)}$ is the average value of all belief points which have the same best α-vector as point b.

Theorem 3. *The time complexity of value heuristic is* $O(|B_\alpha||\Gamma|)$ *where* $|B_\alpha|$ *is the number of belief points that share the same best* α*-vector with* b *and* Γ *is the set of* α*-vectors.*

5.2.4 *Belief Point Entropy and Number of Iterations*

We observe that, in addition to the proposed distance measures above there are two other factors which affect the overall performance of the execution of the POMDP policy. One is the number of times we have triggered our replanning during plan execution. Obviously, we expect the improvement from replanning to decrease with the number of replannings that have occurred so far, because we are improving our policy gradually each time, so the additional improvement that is possible should, therefore, decrease each time. The second factor is the entropy of the belief point: we find that when the current belief point has larger entropy, it often leads to a bigger improvement from replanning compared to ones with small entropy. One explanation could be that belief points with less entropy are less uncertain and more likely to be in the corners of the belief space and will have already been covered by the initial upper and lower bounds generated by SARSOP.

Unlike the previous two heuristic measurements, these two factors are not related to the sampled belief point set but can be seen as independent properties that will affect our overall performance. We combine the previous two heuristics with these two additional factors to form another two heuristic functions as follows (λ and γ are weights for combining the two factors).

$$M_3(b, B) = \lambda \mathtt{Entropy}(b) + \gamma \mathtt{Iter} + M_{L_1}(b, B) \tag{37}$$

$$M_4(b, B) = \lambda \mathtt{Entropy}(b) + \gamma \mathtt{Iter} + M_{val}(b, B) \tag{38}$$

Since we have more parameters in these equations than in the first two heuristics, there is a risk that by optimising the parameters we will get better performance than with the other heuristics simply because we are trying more variants. To prevent this, for each measure we calculate a "standard" setting of the parameters which work reasonably well on a variety of domains, rather than the best setting for a particular domain.

Theorem 4. *The complexity of* M_3 *and* M_4 *are* $O(|S| + |B|)$ *and* $O(|S| + |B_\alpha||\Gamma|)$ *respectively*

Proof. For each b, calculating its entropy takes $|S|$ time. According to Theorem 2 and 3, it takes $|B|$ and $|B_\alpha||\Gamma|$ to calculate M_{L_1} and M_{val} □

There are several questions we would like to answer in the experiment section. First, how much is our execution monitoring improving straight SARSOP in terms of total reward and total time. If we ignore the time cost for actions, the only difference in terms of execution time would be our extra computation on heuristic functions and replanning. There is an obvious trade-off between the threshold we use to trigger the replanning procedure and the time allowed for replanning. One can easily imagine that a large value of threshold and a small value of replanning time can make the agent replan less and possibly produce the same result as standard SARSOP, while a small value of threshold and a large value of replanning time can possibly leads to a large improvement in the final total reward the agent collected but also consume more computational power and time.

Secondly, we would like to investigate how the heuristic functions affect our execution monitoring approach in three different domains. The more effective the heuristic function we choose, the more accurately we determine whether the current policy is good enough for our current belief point. We would like to see which heuristic function with the best parameter setting can generate best performance for all three domains in terms of total reward and total time. Finally, we need to decide parameters for the heuristic functions. As for the first three heuristic function ($M_{gap}, M_{L_1}, and M_{val}$), threshold and replanning time are only two parameters we would like to look at in the experiment. However, the M_3 and M_4 heuristic function has two additional parameters for the entropy of the belief point and number of replanning so far.

5.3.1 *Domains*

We tested our approach on three general POMDP domains. The first is a factory domain where the goal is to assemble different components of a product using corresponding robot arms. There are three individual states for each arm, which are state "On", "Off" and "Faulty". Two different actions are available in the domains which are *TurnOn* action that turns on the robot arms from "Off" state to "On" state and *Assemble* action that can assemble things. One precondition of *Assemble* is all the arms turned on so that *TurnOn* action for each arm needs to be executed before *Assemble* action. We are adding another interesting element into this simple domain by making a low probability (0.01) of transition when we execute the *Assemble* action. This transition will result in all robot arms becoming faulty. When we execute the *Assemble* action with all arms turning on, a positive reward will be assigned. Therefore, if we do not pick appropriate repair actions to recover from faulty situations of the arms, no positive reward will be gained by following the initial policy. Noisy observations are also available after *Assemble* action in order to check the states of each arms. Consider a factory domain with 2 components: the POMDP associated with it is defined by the tuple $\langle S, A, T, \Omega, O, R, \beta \rangle$:

- $S : S_1 \times S_2 \times S_{ready} \times S_{goal}$, where S_1 and S_2 are state spaces for each component. $S_1 : \{Off, On, Faulty\}$, $S_2 : \{Off, On, Faulty\}$. S_{ready} is an intermediate state for the goal state S_{goal}. $S_{ready} : \{Yes, No\}$, $S_{goal} : \{Yes, No, Fail\}$

- $A : \{\text{TurnOn}_1, \text{TurnOn}_2, Ready, Assemble, \text{Repair}_1, \text{Repair}_2\}$ is the set of actions. The first two are *TurnOn* actions for the components. Action *Ready* can make the intermediate state S_{ready} become *Yes* if all the components are *On*. Action *Assemble* rep-

resents the goal-achieving action and the last two are repair actions when components are damaged.

- $T : S \times A \times S \rightarrow [0, 1]$ represents the state transition function. *TurnOn* action has no effect when the state of the component is either *On* or *Fail* and will change the state of the component from *Off* to *On* with 0.9 probability. When $S_{ready} = Yes$, action *Assemble* has 0.99 probability to make S_{goal} become Yes but also has 0.01 probability to make all the components become $Faulty$.

- $\Omega : S \times A \times O \rightarrow [0, 1]$ is the observation function where $O = \{O_{normal}, O_{fail}\}$. This function reveal the true state of each component with 0.9 accuracy. The state Off and state On are both considered as $normal$ state.

- $R : S \times A \rightarrow R$, specifies the reward function from the state-action space to real number. In our case, $R = 3$ if $S_{goal} = yes$. This value is found manually so that the SARSOP can generate a policy to achieve the goal state.

The second domain is the reconnaissance domain from the international planning competition [94], where an agent is equipped with tools to detect water and life, and also take pictures when it finds life on another planet. A positive reward will be gained if we finally take pictures at the place where life exists. The first interesting part of this domain is deciding between different sensing actions, such as water-detecting actions or life-detecting actions which are noisy. As in a RockSample domain, more accurate sensing actions usually have larger costs while cheaper sensing actions often result in poorer observation ability. Therefore, a trade-off between the quality and the cost needs to be made for the reconnaissance domain. Apart from that, in their original setting, there are some hazardous places where the

agent's equipment, such as a water detector, or cameras, have 0.01 probability of being damaged and it needs to return to the base to get repaired. However, they assume there are observations available about the states of the tools for all the actions in the domain, which means moving actions also give us information about the status of the tools. In order to make the damage to the tools harder to detect, we assume there is observation about the state of tools only when we move out of the hazard places. Consider a POMDP problem associated with our modified version of the reconnaissance domain in a 2×2 grid map. It can be defined by the tuple $\langle S, A, T, \Omega, O, R, \beta \rangle$:

- $S : S_x \times S_y \times S_{p_{0,1,2,3}-has-water} \times S_{p_{0,1,2,3}-taken-picture}$ $\times S_{damaged-water-detector} \times S_{damaged-camera}$.
  - $S_x : \{x_0, x_1\}$ and $S_y : \{y_0, y_1\}$ are rover's position in the map. The hazard place is at $[x_1, y_1]$ and the base is at $[x_0, y_0]$.
  - $S_{p_{0,1,2,3}-has-water} : \{Yes, No\}$ represents the underlying state of water value in each position .
  - $S_{p_{0,1,2,3}-picture-taken} : \{Yes, No\}$ denotes whether the position is taken picture by the rover.
  - $S_{damaged-water-detector} : \{Yes, No\}$ denotes the condition of the water detector. It only become Yes with 0.01 probability when the rover move out the hazard place.
  - $S_{damaged-camera} : \{Yes, No\}$ denotes the condition of the camera. It only become yes with 0.01 probability when the rover moves out of the hazard place
- $A : \{Down, Up, Left, Right, Detect-Water_{0,1,2,3}, Take-picture_{0,1,2,3}, repair-water-detector, repair-camera\}$ is the set of actions.
- $T : S \times A \times S \to [0, 1]$ represents the state transition function. All the movement actions are deterministic. The observation action

$Detect-Water$ has 0.9 probability to reveal underlying state of the water existence in a position. The action $Take-picutre$ can take picture on a position when the rover believes it has water.

- $\Omega : S \times A \times O \rightarrow [0, 1]$ is the observation function where $O : O_{has-water} \times O_{damaged-wd} \times O_{damaged-camera}$.
  - $O_{has-water} : \{Yes, No\}$ is the observation variable for water detection action.
  - $O_{damaged-wd} : \{Yes, No\}$ is the observation variable for checking the water detector's conditions, Yes only appears with 0.9 probability when the rover moves out of the hazard place and has a damaged water detector.
  - $O_{damaged-camera} : \{Yes, No\}$ is the observation variable for checking the camera's conditions, Yes only appears with 0.9 probability when the rover moves out of the hazard place and has a damaged camera.
- $R : S \times A \rightarrow R$, specifies the reward function from the state-action space to real number. In our case,

$$R = \begin{cases} 30 & \text{if } S_{has-water} \wedge S_{picture-taken} \\ -28 & \text{if } \neg S_{has-water} \wedge S_{picture-taken} \end{cases}$$

  Again, these two values are manually set up so that the point-based algorithms can find a policy to achieve the goal.

In the previous reconnaissance domain, there is only one repair action for each tool. We are adding one more repair action $\{repair-water-detector2, repair-camera2\}$ with different recovery accuracy and action cost. This is to study the effect of more complex recovery actions on the execution monitoring performance.

Factory (54s, 6a, 20)	Gen. & Exec. Time (s)	Total Reward
SARSOP (no replanning)	$400 + 0.01$	286.68 ± 213.56
SARSOP (no replanning)	$200 + 0.01$	280.41 ± 220.88
SARSOP (no replanning)	$100 + 0.01$	292.6 ± 229.9
Random Replan	$100 + 8.56$	452.4 ± 201.9
Heuristic M_{gap}	$100 + 0.32$	261.7 ± 218.6
Heuristic M_{gap}	$100 + 1449.8$	589.5 ± 115.5
Heuristic M_{L_1}	$100 + 5.64$	566.2 ± 81.7
Heuristic M_{val}	$100 + 3.09$	581.2 ± 70.6
Heuristic M3	$100 + 10.21$	602.8 ± 3.0
Heuristic M4	$100 + 3.93$	608.3 ± 48.3
look-ahead (blind strategy)	$0.06 + 3.5$	240.8 ± 192.7
look-ahead (SARSOP offline)	$100 + 343.9$	611.9 ± 28.4

Table 8: Results for the factory domain.

5.3.2 *Results*

The five heuristic measurements and straight SARSOP are tested on the three domains. Two general parameters are considered here. One is the threshold θ, which is used for determining whether the current policy is good enough for the current belief point. The other is the replanning time τ, which is applied as a time constraint for replanning. We use the following function to find the best parameters for each heuristic:

$$(\theta, \tau) = \arg\max_{\theta,\tau} \frac{Reward(\theta,\tau)}{Max(Reward)} - \frac{Time(\theta,\tau)}{Max(Time)} \tag{39}$$

where $Max(Reward)$ is the maximum total reward and $Max(Time)$ is the maximum execution time. The parameter settings used were op-

Reconnaissance (4096s, 14a, 80)	Gen. & Exec. Time (s)	Total Reward
SARSOP (no replanning)	$400 + 1.48$	1730.4 ± 1382.0
SARSOP (no replanning)	$200 + 1.09$	1559.5 ± 1279.0
SARSOP (no replanning)	$100 + 0.58$	1507.3 ± 1244.9
Random Replan	$100 + 21.11$	2981.7 ± 871.8
Heuristic M_{gap}	$100 + 14.73$	3493.9 ± 584.7
Heuristic M_{L_1}	$100 + 17.51$	3597.2 ± 305.2
Heuristic M_{val}	$100 + 11.14$	3325.9 ± 864.6
Heuristic M_3	$100 + 17.28$	3530.0 ± 1045.6
Heuristic M_4	$100 + 7.09$	3357.5 ± 890.8
look-ahead (blind strategy)	$22.75 + 295.9$	469.6 ± 421.2
look-ahead (SARSOP offline)	$100 + 1281.9$	3698.5 ± 309.8

Table 9: Results for the reconnaissance domain.

timised over all three domains and then the same settings were used for all experiments.

In each domain we compare standard SARSOP with 100 seconds, 200 seconds, or 400 seconds available for policy generation. As well as standard SARSOP and SARSOP with our plan repair, we also run SARSOP with replanning triggered randomly. To make this a fair comparison, we first calculate the average number of replannings in each domain, where the average is over all the heuristics tested; Then, for the random replanning variant, we set the probability of replanning at each step so the expected number of replannings is the same. The best number of replanning for the factory domain, reconnaissance domain and modified reconnaissance domain are 2, 4, and 7.

We also compared our results with an on-line look-ahead algorithm, where the current best action is computed by expanding the search tree at each time step. Because on-line POMDP solvers require

an off-line policy to provide heuristic values for each state, we show results using both a blind strategy [47] and a policy generated by SARSOP for the heuristic. A blind strategy is sometimes called a fixed-action , which generates a lower bound on the optimal value function by always choosing the same action regardless of current belief state. Therefore, there will be at most $|A|$ vectors in the set after applying the blind-strategy. The policy generated by SARSOP usually has a tighter bound than blind strategy (also more computation time) because Bellman updates have been applied to the initial bounds. Both policies are used to estimate the values of the fringe nodes at the search tree expanding process and are propagated to the starting node in order to choose the current best action.

The performance comparisons are displayed in Tables 8–10. Initial policies are generated by SARSOP with a time limit of 100 seconds, and execution time is the average CPU time for each problem over 100 trials. As the tables show, SARSOPs with heuristic approaches generally improve standard SARSOP in terms of total reward by more than 100% in all three domains. Even when SARSOP has four times as much computation time, our approach still performs much better, and it is clear from the SARSOP performance, particularly in the Factory domain, that further computation will not improve the policy. Random replanning does surprisingly well, largely because in these domains there is no penalty for carrying on with normal actions after a fault has occurred, so the system can keep acting badly until replanning occurs, without a penalty. The on-line algorithm using the blind strategy does not generate sensible policies and, with the SARSOP policy, produces better results in total reward, but also requires much more execution time. We did not inject faults into the runs, so if no fault occurs at all, all the approaches will perform very well. This is shown in the standard deviations of the rewards, which typically fall

Modified Recon. (4096s, 16a, 80)	Gen. & Exec. Time (s)	Total Reward
SARSOP (no replanning)	$400 + 0.41$	1470.8 ± 1321.4
SARSOP (no replanning)	$200 + 0.97$	1511.1 ± 1138.8
SARSOP (no replanning)	$100 + 0.78$	1704.7 ± 1367.0
Random Replan	$100 + 15.48$	2831.3 ± 861.2
Heuristic M_{gap}	$100 + 20.4$	3185.8 ± 1120.0
Heuristic M_{L_1}	$100 + 44.68$	3152.4 ± 1122.9
Heuristic M_{val}	$100 + 11.76$	3491.7 ± 641.5
Heuristic M_3	$100 + 18.05$	3694.7 ± 332.5
Heuristic M_4	$100 + 12.69$	3362.8 ± 843.4
look-ahead (blind strategy)	$28.7 + 625.7$	400.4 ± 79.5
look-ahead (SARSOP offline)	$100 + 272.0$	3582.9 ± 431.1

Table 10: Results for the modified reconnaissance domain.

as reward increases, reflecting the fact that the replanning improves only the low reward runs.

Comparing the heuristics, we note that while the Gap heuristic works well for the two reconnaissance domains, it performs very poorly in the Factory domain, where the two different entries in the table correspond to two different thresholds for replanning. The poor performance is because it is extremely sensitive to this threshold as the noisy observations mean the belief states change only gradually from a belief that everything is OK to one in which a fault has occurred. This results in a heuristic that either does not replan at all, or replans far too often.

Heuristic M_{val} is cheaper to compute than M_{L_1} because we store the best α-vector for each point, so all that needs to be done is calculate the averages, which for domains of this size is cheaper than computing the L1 norm. The best total rewards in factory domains and modified reconnaissance domain were generated by the heuristic

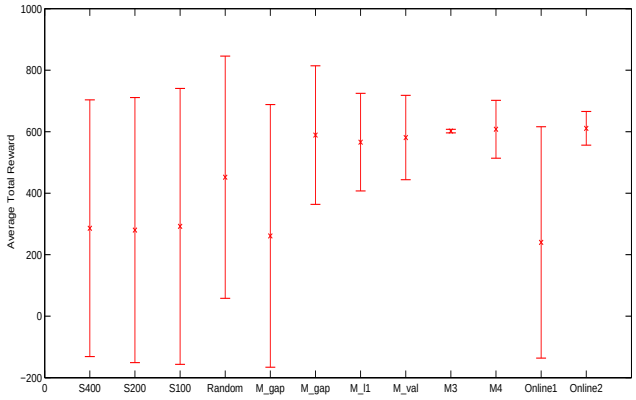

Figure 23: Plotted graph for factory domain with 95% confidence interval

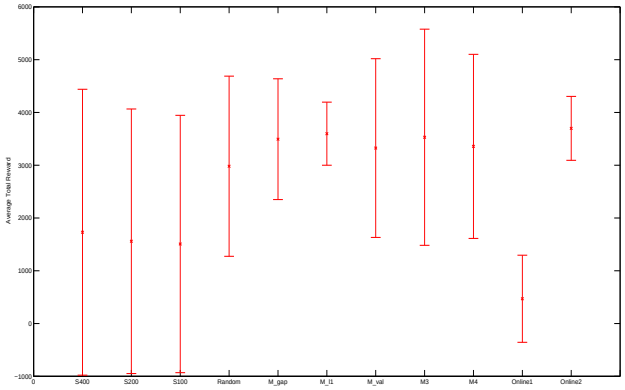

Figure 24: Plotted graph for reconnaissance domain with 95% confidence interval

M3 and M4 which combines three parameters and there is a statistical significant difference between the performance of M3 and M4 and that of M_{val} and M_{L_1} for the Factory and Modified reconnaissance domains. The significant differences of average total reward between the algorithms are shown in Figure 23–25. As you can see from the Figure 23–25, our competitors results have much higher standard deviations than many of our approaches variations, because the policies of the competitors will perform poorly if unlikely action outcomes actually occur at run time, while our approaches account for both likely and unlikely outcomes.

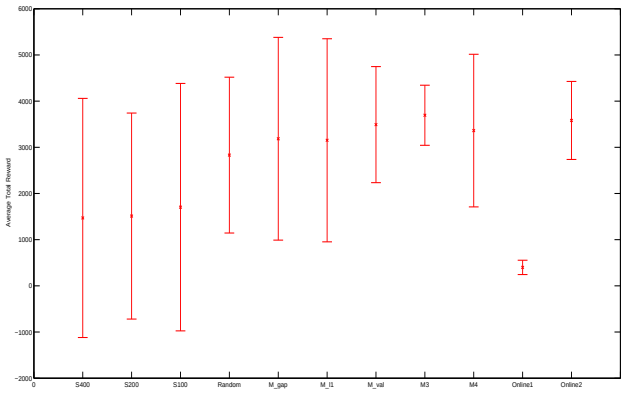

Figure 25: Plotted graph for reconnaissance2 domain with 95% confidence interval

When comparing the two reconnaissance domains, it can be seen that modified version with more recovery actions takes more execution time for all heuristic functions. This is what we ed because more repair actions means more replanning time to perform Bellman backup operations. One thing worth noting here is that standard SARSOP also performs worse in the modified reconnaissance domain. The execution time of the first heuristic function which computes the L1 distance between the current belief point and the existing belief point set grows with the number of states in the domain. As you can see from Table 8 and 9, it only takes an average of 5.64 seconds to finish 1000 steps for the factory domain, which has only 54 states, while it needs an average of 17.51 seconds for the modified reconnaissance domain, which has 4096 states.

To show performance in domains that do not have the properties we expect to benefit our approach, we also compared with SARSOP in the Hallway2 problem (92 states) and the rock sample domain (4096 states). As shown in table 11, SARSOP does a good job of covering the policy for Hallway2 so replanning is rarely needed. On this kind of domain our approach only evaluates the heuristic at each step, so the overhead compared with standard SARSOP is small. On the rock

Hallway2	Gen. & Exec. Time (s)	Total Reward
SARSOP (no replanning)	$100 + 0.01$	6.91 ± 1.95
Heuristic M_{L_1}	$100 + 0.68$	7.02 ± 1.87
RockSample(4,4)		
SARSOP (no replanning)	$100 + 6.43$	1338.4 ± 208.17
Heuristic M_{L_1}	$100 + 22.77$	1343.0 ± 159.02

Table 11: Results for the RockSample and Hallway domain.

sample domain we see a small amount of replanning and a slight improvement in performance.

In conclusion, we have demonstrated that our heuristic measurements will all improve the initial policies, which are generated by standard SARSOP in three different domains by committing computation for replanning at run-time. The second heuristic function out performed the first heuristic function in both total reward and total time for all three domains, while heuristic functions that combine the previous heuristics with additional attributes will generate better policies than the other two but also need more computational time. All the heuristic functions tend to require more execution time when the number of states in the domains becomes larger.

5.4 CONCLUSION

We have shown how additional execution monitoring can be applied to approximate off-line POMDPs solutions at run-time to increase the robustness of the policies. Particularly, we are interested in the problems that arise from point-based POMDPs solvers, which are trying to focus on the most promising belief points and tend to ignore the points with low probability. After detecting the situations where current polices are not performing well for our current belief points, we

use standard backup operations to trigger the replanning procedure on-line in order to produce a better policy for agents to execute at run-time. Because such a replanning procedure is computationally expensive using standard backup operations at execution time, we would like to trigger this procedure as little as possible. We proposed several different heuristic functions that are used to detect such situations.

By doing this incremental modification of the initial policy, we showed in experiments that our approach can have a better overall performance in several POMDP domains.

6

RELATED WORK

6.1 RELATED WORK ON QDET-POMDP MONITORING

Many execution monitoring approaches as described in Chapter 3 try to focus on detecting the discrepancies between the actual world state and the agent's knowledge of the world and incorporate plan modifications or replanning at run-time to recover from any of the faulty states or unexpected states. Many of them do not address the same problem of partial observability as we investigate here, so their approaches are not comparable with ours. The work of Fritz[37] on monitoring MDP policies as mentioned before is the most related to our MDP execution monitoring approach, although he tried to address the problem of preserving the optimality of a plan given a dynamic environment. The dynamic environment means the planning model is not complete, so exogenous events could occur to affect the state at any time step. Once an external event happens at run-time, his monitoring approach will examine the relevant conditions associated with the optimal solution and sub-optimal solutions in order to select the current best action. However, our work is using execution monitoring to fix an approximate plan or policy that is generated because of computational reasons. The noisy stochastic sensing actions in QDET-POMDPs play a large part in the difficulties but we are not concerned withthe dynamic environment here.

The other work closely related to ours from the execution monitoring literature was undertaken by Boutilier[14] (as mentioned in Sec-

tion 3.1.3.5), which similarly used classical planning plus execution monitoring to solve problems that could be represented as POMDPs. In that work, the plans are non-branching and the problem is to decide when to observe the preconditions of actions and determine if they are true, as opposed to using execution monitoring to determine which branch to take. In common with our approach, they use value of information to measure whether monitoring is worthwhile, but then formulate the monitoring decision problem as a set of POMDPs, rather than using value of information directly to select observational actions.

Ong et al.[74] exploited the structure of mixed observability, which means some states are fully observable, while the others are partially observable. Quasi-Deterministic models also fall into this category. However, they apply point-based value iteration to compute an optimal policy which could still be problematic when partially observable states are larger. Our approach can also be viewed as generating MDP policies or contingent plans for partially observable states and using value of information monitoring to modify plans.

An alternative to execution monitoring for solving Quasi-Deterministic problems efficiently is described in Goebelbecker et al. [41]. There a classical planner and a decision-theoretic (DT) planner are used to solve these problems, switching between them as they generate a plan. The approach is similar to ours in that they use FF to plan in a determinisation of the original problem augmented with actions to determine the values of state variables, which they call *assumption actions*. However, they build linear plans with FF and switch to the DT planner to improve the plan whenever an observation action is executed. The DT planner looks for a plan either to reach the goal, or to disprove one of the assumptions. If this occurs, replanning is triggered. The advantage of their approach is that it can find more

general plans using the DT planner, so we might expect it to produce slightly better quality plans overall. However, the DT planner is much more computationally expensive than the simple value of information calculation we use.

Classical planners have also been applied in fully observable MDP domains. By far the most successful of these approaches has been FF-replan [115], from which we have taken the determinisation ideas discussed above. Because these approaches rely on being able to determine the state after each action, they cannot easily be applied in POMDPs. Our approach can be thought of as FF-replan (although we actually build the entire contingency plan rather than a single branch), where we use execution monitoring to determine with sufficient probability the relevant parts of the state.

Translation-based approaches have recently been popular for solving non-deterministic problems. One example is conformant planning where an agent is trying to generate a plan that works for every possible initial state when the initial situation of the world is not fully known in advance. The actions in conformant planning can also have non-deterministic effects [42]. Although there are many initial states that the agent can start in, the solution of conformant planning always leads the agent to the goal, regardless of the initial state. Palacios et al. [75] proposed a translation-based approach to solve conformant problems with a classical planner. They have shown that the translation is sound but not complete, which means all the plans found by the classical planner are conformant plans but the classical planner cannot find every possible conformant plan. They extended the work to produce a more powerful translation scheme so that the approach is both sound and complete and its complexity can be characterized in terms of a parameter in the problem [76]. However, they do simplify the problems by making the actions deterministic so the only

uncertainty comes from the initial state. Although the belief state is represented in conformant planning, it only maintains a set of possible states of the world while in our work a probability distribution over possible states is maintained at run-time.

Another example is contingency planning where both sensing actions and incomplete information about the initial states are available. Work in Albore et al.[1] extended the previous conformant translation approach by encoding sensing actions as non-deterministic actions. However, sensing actions are assumed to have the ability to reveal the truth value of the unknown state variables during execution. In our case, sensing actions can have more complex observation models where observations are not always correct. Shani et al. [96] take a different approach for solving contingency planning where replanning will occur when the current observation is inconsistent with the current belief state. The idea is similar to FF-replan [115]. A concrete initial state is firstly sampled from the uncertain initial state, so a classical planner can be applied to generate an initial plan. When the plan is executed, the belief state will also be updated accordingly. Once the observation received is inconsistent with current belief state, they will replan with an updated state. Shani et al. also assume deterministic actions and perfect sensing actions. Generally speaking, models of contingency planning and conformant planning are not as complex as POMDPs which capture noisy observations, incomplete information about initial states and stochastic actions at the same time.

All the above translation-based approaches tend to put additional constraints on a set of planning problems so that they can be translated into the classical planning. We adapted similar ideas where QDET-POMDPs can be thought of as making state-changing actions deterministic while keeping other properties of the general POMDP the same. One major difference between our approach and other

Algorithm 8 PBVI

Let B be b_0
repeat
  for all $b \in B$ **do**
    $\alpha \leftarrow$ Backup(b,V)
    add(V,α)
  end for
  $B = B \cup \arg\max_{b' \in \text{Successor}(B)} \text{dist}(B,b')$
until V has converged

translation-based approaches is that we work on the reward-based problems the object of which is to find an optimal solution that can maximise the reward collected. That is the reason we proposed execution monitoring to improve the approximate plan's quality at runtime.

6.2 Related work on execution monitoring of point-based policies

The first point-based algorithm PBVI (point-based value iteration) [81] solves POMDPs for a finite set of belief points. The key idea is it only maintains one α-vector per point, so the number of α-vectors is not going to be greater than the number of belief points. Moreover, in the backup stage, updates of α-vectors are only performed on this representative set of belief points. By doing this, Pineau et al. [81] have pointed out a full point-based update only takes polynomial time and the size of the solution remains constant. As for finding the relevant belief point set, PBVI starts from a single belief point and incrementally expands its belief set greedily by choosing the reachable belief point which is furthest away from the existing belief set. This is done by stochastically simulating a forward trajectory from any currently selected point b_0 in B. L_1 distance is used to measure the distance between the simulated points and the currently selected point. Pineau

Algorithm 9 Perseus

```
Let B ← points generated from random walk
repeat
  B′ ← B
  while B′ ≠ ∅ do
    b = random(B)
    α = Backup(b, V)
    if α · b ⩾ V(b) then
      B′ ← b′ where α · b′ ⩽ V(b)
      add(V, α)
    end if
  end while
until V has converged
```

et al.[81] have also pointed out that the number of selected points is at most double the previous set size because each current point is going to contribute one new point at every iteration. A skeleton of PBVI is shown in Algorithm 8.

The Perseus algorithm [105] is built on the main idea of PBVI, but differs from PBVI in both belief point generating strategy and backup strategy. It firstly explores the world with a random walk in order to generate initial belief points set B. This subset remains the same throughout the following backup operations. The other difference is how to choose backup operations. Instead of updating α-vectors for all the representative points one by one. Spaan et al. [105] observed that a single update of the current α-vector for the current belief point is going to be beneficial to other belief points at the same time. At the beginning of each Bellman update iteration, all the belief points are in a belief set B. After randomly selecting a belief point in B and updating its α-vector, all the belief points that are improved by this α-vector will be removed from the set B and backup operations are performed randomly over the points which are still left in B. By doing this, they expected that the number of backup operations could be reduced at each iteration compared to PBVI algorithm. A skeleton of Perseus is shown in Algorithm 9.

Algorithm 10 HSVI

Initialize $\underline{V}$ and $\bar{V}$
while $\bar{V}$ - $\underline{V} > \epsilon$ **do**
  explore(b_0, $\bar{V}$,$\underline{V}$)
end while

Algorithm 11 explore(b, $\bar{V}$,$\underline{V}$)

Initialize $\underline{V}$ and $\bar{V}$
if $\bar{V}$ - $\underline{V} < \epsilon\gamma^{-t}$ **then**
  return
end if
$a^* \leftarrow \arg\max_a Q_{\bar{V}}(b, a')$
$o^* \leftarrow \arg\max_o \bar{V}(\tau(b, a^*, o)) - \underline{(V)}(\tau(b, a^*, o))$
explore($\tau(b, a^*, o^*)$,$\bar{V}$,$\underline{V}$)
add$\{\underline{V}, Backup(b, V)\}$
add$\{\bar{V}, (b, H\bar{V}(b))\}$

The previous two point-based algorithms are using α-vectors to represent value functions which can be seen as lower bounds on the optimal value function. The process of value iteration incrementally increases this lower bound $\underline{V}$ to approximate the optimal value V^*. Heuristic search value iteration algorithms (HSVI) [102, 103] choose another way of representing value functions by using both lower bounds $\underline{V}$ and upper bounds $\bar{V}$ on the value functions. Lower bounds are again represented as α-vectors as represented in PBVI and Perseus. Upper bounds are represented as a point set where each point stores a belief value and its upper bound value. The upper bound is updated by adding a point into the set. In order to evaluate a belief point's upper bound $\bar{V}(b)$ from the upper bound set, HSVI1 [102] needs to compute the exact projection of b onto the convex hull of the points in the set which involves solving a linear program. HSVI2 [103] introduces an approximate projection onto the convex hull in order to avoid expensive linear programming. The goal of HSVI is trying to minimise the gap between the lower bounds and the upper bounds of value functions for the root belief point in order to approximate the optimal value functions. Because the upper bounds only

have information about the expected value of belief points, α-vectors are used as the final output for the agent to execute. A skeleton of HSVI is shown in Algorithm 10. Instead of updating upper bounds and lower bounds for each belief point at each iteration, HSVI first needs to perform a heuristic search (Algorithm 11), which explores new belief points by using the same action and observation choosing strategies as in SARSOP. When the search is completed by satisfying a termination condition, such as the difference between the lower bound and the upper bound of the leaf node is less than a certain threshold, backup operations are executed backward from the newly generated belief points to the initial belief point. This process will continue until the difference between the lower bound and the upper bound of the initial node satisfy the threshold condition. The main idea of HSVI is that those belief points are generated according to the action and observation selecting strategy so they can contribute more in improving the lower bounds and upper bounds for the root belief point. Backup operations are executed for both lower bounds and upper bounds when a new point is added into the representative set.

In general, our approach of execution monitoring on point-based policies can be viewed as interleaving between on-line algorithms and off-line algorithms of POMDPs solvers. On-line POMDP solvers compute the current best action from the current belief state at every step (see [90] for a survey of on-line techniques). On-line solutions generally generate a rough policy off-line, for example using PBVI, as in [88], and then use local search at run-time to improve this. Most recent algorithms use the off-line policy to generate a heuristic that is then used to guide heuristic search on-line. This approach has two disadvantages. First, the same amount of computation is used at run-time whether the belief state is close to those used to generate the

initial policy or is far away from them. Second, the heuristic remains fixed, so the computation used in calculating the best action from one belief state provides no benefit when calculating the best action at future belief states. Hybrid POMDP [64] attempted to solve the second problem at each step by deciding whether to spend a small portion of the on-line computation on improving the initial policy so later search can benefit. However, they base their decisions on very approximate lower and upper bounds on the value function. The approach we describe below can be thought of as a way to overcome both the above disadvantages. We spend more time on the initial policy than an on-line algorithm might, so our initial policy is hopefully better, and we then use heuristics to decide when to simply rely on the current policy and when to do extra computation to improve it, but unlike the on-line approaches we re-use the extra computation in future by incorporating its results into the off-line generated policy.

One final piece of related work, which proposes an online POMDP learning algorithm which can be adapted for slow environment change was carried out by Shani et al. [45]. Even though they did not characterise their work as execution monitoring, the paper tried to address the problem of a slowly changing dynamic environment occurring on line, such as changes in reward functions or observation probability. One similarity between their work and our work is incrementally improving POMDP policies at run-time in order to increase plan quality. They adapted an on-line version of HSVI algorithm to react to this change in environment. More specifically, traditional HSVI is modified to become an on-line version of POMDP solver where both upper bounds and lower bounds of value functions can be improved incrementally at run-time. This is different from other on-line POMDP solvers, which only use heuristic tree search to select the best action for the current belief point. They realized that recomputing the pol-

icy whenever there is a change in the environment is costly, so they checked whether the current best α-vector could actually achieve the expected value during backup operations and remove the α-vectors which are overoptimistic so that all remaining α-vectors are valid as lower bounds of the value functions even after environment changes. While this does modify the off-line policy in a way similar to ours, its different motivation (a changing model, rather than poor approximation) leads to a rather different solution.

Authors in [61] claim that the benefits of keeping belief point set B small usually out-weight the loss in the approximation quality due to over-pruning. So they introduce a more robust pruning technique called δ-dominance. Pruning α vectors for exact POMDP solvers requires this α-vector to be dominated by another α-vector over entire belief space, and pruning in point-based POMDP solvers only requires the dominance over the sampled belief space B. They noticed that computed approximately optimal policy might be poor at certain regions, so that a more robust pruning requirement is needed. This has the same motivation as our work in execution monitoring of point-based policies but they try to address the problem at planning stage. A α vector α_1 dominates another α vector α_2 at a belief point b only if $\alpha_1 \times b' \geqslant \alpha_2 \times b'$ at every point b' whose distance to b is less than δ. This requirement of dominance forces the better α-vector not only to dominate another one on the current sampled belief point but also for its neighbourhood defined by constraint δ. One can imagine if this constant δ is large enough, pruning in point-based algorithms is the same as in general POMDP solvers, because it will cover entire belief space when determining the dominance of the vectors.

While up until now, we have only been dealing with the problem that point-based POMDP solvers have not been able to cover belief region which is unlikely to occur. In future work, we would like to

find a more general execution monitoring approach that can address the issue of dynamic environment where action effects can be different from what we expect due to the parameters changing in POMDP models or exogenous events.

7

CONCLUSION AND FUTURE WORK

In this thesis, we have presented an idea of integrating both off-line and on-line planning in order to tackle planning problems with partial observability. In the early stage of planning development, due to the lack of computational power, problems are often assumed to be deterministic and can be observed completely. In order to solve more realistic problems, these assumptions are often relaxed. The problems we addressed in this thesis can naturally be modelled as POMDPs which have the ability to capture uncertainty in both action outcome and sensing ability. However, due to the complexity of solving POMDPs using exact solutions, many approximate POMDP solvers have been developed instead. The approximate solutions can usually be divided into two categories. One is off-line POMDP solvers where a near-optimal policy is generated at the off-line stage. The policy is then executed without any change at each time step during execution time. The other is on-line POMDP solvers where a heuristic search needs to be called in order to find the best action for the current belief state at each time step. On-line POMDPs usually require an approximate policy generated off-line which is used to provide a heuristic value at run-time. Compared to off-line solvers, on-line solvers will spend many more computational resources at execution time than at planning time. Our algorithms can be seen as a combination of both off-line and off-line solvers. At the off-line stage, we still use the traditional off-line POMDP solvers to generate the near-optimal policies but try to improve the policies at run-time by applying execution monitoring approaches. These approaches differ

from on-line POMDP solvers in the way of performing the plan repair processes. Instead of performing plan repair at each time step as on-line POMDP solvers do, our execution monitoring approach only does it when it is decided the initial policy is not good enough at the current time step. The main research question we answered in this thesis is when and how to perform the plan modification procedure at run-time according to the planning structure and information available. There has been a great deal of research that addressed this problem by investigating a variety of execution monitoring approaches on different planning algorithms. Our algorithms can also been seen as another execution monitoring approach that works on POMDP algorithms but the objective of which is to improve approximate policies not to cope with exogenous events or model changes. In the evaluation section, we have compared our approach with standard off-line POMDP solvers and on-line POMDP solvers in several simulated domains. It has been demonstrated that our additional execution monitoring can out-perform other algorithms (on-line and off-line) in terms of the plan generation time, plan quality and plan execution time. In particular, two POMDP off-line solvers were considered here. One is applying translation-based approaches which convert a constrained POMDP problem into a classical planning problem or an MDP and use a classical planner or an MDP solver to generate an initial policy. The other is using point-based POMDP solvers, which are a group of approximate POMDP solvers to generate near-optimal policies.

Translation-based POMDP solvers applied an idea of solving POMDPs using classical planning, which has been very popular recently in the planning community. Classic planners usually scale much better than their non-classical counter-parts, although they can not capture all the uncertainty in the environment. In particular, we considered a

sub class of POMDPs (QDET-POMDPs,) where only observation actions can have stochastic outcomes. Since there is no direct translation between POMDP problems and classic planning problems, the plan generated by the classical planner is not an optimal solution for the original POMDP problems, which is why we deploy our execution monitoring at run-time. At the off-line stage, a contingency plan or an MDP policy is built which depends on the true state of the world, and assumes that observation actions are reliable during execution time. The translation is done by determinising the probability outcomes of observation actions and initial state. At the on-line stage, our execution monitoring approach is then used to select appropriate observation actions by taking account of noise in the observation actions, the current belief state and the information gain of executing the observation action. The major behaviour of our algorithm is inserting additional observation actions at the branch points in order to gain enough confidence about which state the agent is currently in so the associated plan branch can be followed. The timing for triggering execution monitoring procedure is determined before executing the policy, because it will only be triggered when the observation actions, which are the branch points in the decision tree, are encountered. All other state-changing actions are assumed to have deterministic outcome and do not need to be monitored if we assume a static environment (no exogenous events) during execution time. We have compared our approach with pure translation-based POMDP solvers without any execution monitoring and a state-of-art factored POMDP solver Symbolic Perseus. The comparison between translation-based solver and Symbolic Perseus has shown that the former can scale better than a standard POMDP solver therefore much less plan generation time is required for the translation-based solver. The results of comparing translation-based solvers with and without

execution monitoring also demonstrated how additional action assertions at run-time can improve the initial approximate policy in the end.

The second part of this thesis focuses on the point-based POMDP solvers which have shown great success in generating near-optimal policy for large POMDP domains. The quality of policies produced by point-based POMDP solvers largely depends on the set of belief points that was sampled. When the number of possible execution traces grows, collecting sufficient belief points to cover all possible execution traces becomes infeasible. Therefore, it is possible that the generated near-optimal policies are not suitable for the current belief points at run-time. Our execution monitoring approach will compute a policy for the more likely cases off-line and fix the policy on-line when we encounter an unexpected outcome. Several heuristic functions were proposed in this work in order to detect situations where a better policy might be needed for the current belief point. Once we decide it is time to perform plan repair procedures on the initial policies, new belief points will be sampled and a new policy will be computed based on the newly generated belief point set. Therefore the heuristic functions play an important part in these algorithms to trigger plan repair procedure. The main contribution of this work came from the systematic evaluation of different heuristic functions on several POMDP domains. We not only compared our approach with a pure off-line POMDP solver (SARSOP) but also with an on-line look-ahead algorithm. The results have shown that our approach will sacrifice some computation time for replanning at run-time in order to generate a better policy but does not need to re-plan for all the steps as an on-line POMDP solver does. In terms of the POMDP domains, we first investigated a family of domains where low probability transitions exist, such as a factory domain where a component

Table 12: Results for the *RockSample* Domain comparing both execution monitoring approaches

Algorithm	Gen. Time (s)	Exec. Time (s)	Total Reward	Disc. Reward
RS (4,4)				
SARSOP	100	9.15	279	2.8
SARSOP M_{L_1}	100	12.14	251	3.3
POMDP	274	0.6	251	6.9
Macro Actions	13	3.5	161	2.3
EM	13	1.8	141	1.9
Without EM	13	1.1	41	-3.5

can become faulty with a very low probability. We also compared all the algorithms on some POMDP benchmarks where point-based algorithms can perform well, and the results have shown that our execution monitoring will not perform worse than pure point-based algorithms regarding the quality of the policy and only spend more time on computing values of heuristic functions at run-time. In the experiment, we have also made the trade-off between the execution time and the quality of the plans by tuning the parameters that govern the likelihood of re-planning and the time allowed for each re-planning.

We also compared the results between these two execution monitoring approaches on the RockSample domain. As can be seen from Table 12, an MDP policy can be generated with much less computational effort compared with either SARSOP or Symbolic Perseus, while execution monitoring on point-based policy can improve the original policy generated by SARSOP. Both approaches share the same core idea of exploiting the structure and information from the planning stage to guide the monitoring process. As described in Chapter 3, many execution monitoring approaches try to preserve the validity of the plan validity in the face of a dynamic environment or an

incomplete model, while our approach aims to improve the approximate solutions at run-time.

7.1 SUMMARY OF CONTRIBUTIONS

The major contributions to this thesis are as follows

- Two translation-based approaches of solving Quasi-Deterministic POMDP, which is a sub set of general POMDP. One is using a classical planner FF to generate a contingency plan and the other is using an MDP solver to generate an MDP policy. Both contingency plans and policies are based on relaxed domains where the world is assumed to be completely observable at execution time. The interesting part of translation-based approaches is encoding the observation actions into the relaxed domains so that observation actions can appear in the solutions.

- A novel execution monitoring approach which works on approximate solutions generated by translation-based solvers at run-time. The monitoring approach can insert observation actions at the branch point in order to gain more information about the current state of the world. A value of information technique is utilised to determine the occurrence of observation actions.

- A comparison of translation-based approaches with and without execution monitoring on a range of simulated benchmarks. It has been shown in Chapter 4 that our translation-based approaches with additional execution monitoring mechanism can generate a better policy in the end compared to pure translation-based approaches. A comparison of our translation-based approaches with an off-line POMDP solver (Symbolic Perseus),

which has shown that even though our translation-based approaches could not generate a better policy than the one from Symbolic Perseus, it can scale much better than the general POMDP solver for the large domains.

- A novel execution monitoring approach, which works on point-based POMDP algorithms which are approximate off-line POMDP solvers. This approach exploits the fact that point-based POMDP algorithms only compute optimal policies for the belief points with high probabilities but ignore unlikely belief points. The key contribution here is evaluating several heuristic functions which are proposed to detect the situation where current policy is not good enough for the current belief point at run-time. Experiments have been undertaken to show that our approach has both the advantage of off-line and on-line POMDP solvers. Results from Chapter 5 have demonstrated that our execution monitoring on point-based policies can generate a better policy than the ones without any monitoring on the domains with low probability transitions and will not generate a worse policy on POMDP benchmarks where point-based algorithms are doing well. We also compared the algorithms with an on-line look-ahead POMDP solver where each action is computed by looking ahead into a few future step at run-time. It clearly showed that it is not necessary to perform action search for each time step when initial policy has already covered the region with high probability.

7.2 FUTURE WORK

There are a couple of things that we are considering doing in the future:

- Proposing a universal execution monitoring framework which can be embedded into an intelligent agent's planning system. This is quite difficult to achieve because different plan structures usually require different monitor mechanism. What is more, the varieties of assumptions about the problems also make it harder to develop an general execution monitoring approach.

- Addressing the problem of incomplete model or dynamic environment where the world can be changed unexpectedly. For instance, the current state of the world might change without executing any actions in the domain. Sucha dynamic environment might need the agent to have the ability to observe the world completely. If not, things might change without even being noticed so that it will be more problematic to react to such unexpected situations. We can extend our point-based execution monitoring to handle the dynamic environment under the assumption of knowing things have changed unexpectedly. For example, in a robot hijack problem, where a robot might be moved to a totally different position by a person at any time, the first thing we need to do is correct our belief state by doing some exploration in the environment. Once we gain enough information about this new belief state, we can apply our execution monitoring approach to decide whether or not to execute the initial policy. If the original policy is assumed to be not suitable for current new belief point, we can apply plan-repair to improve the initial policy. The difficulty of this problem will be what types of exploration actions are available for unexpected events. Moving around in the nearby environment might give you correct information about current state and possibly recover from the hijack event, but that does not provide useful information for other types of unexpected events. Knowledge about how to

perform plan-repair actions according to different exogenous events might be provided by system expert in advance. Again, the trade-off between exploration about current state and finding appropriate action to execute needs to be made whenever the unexpected events occur. This is slightly different from the trade-off between exploration and exploitation in reinforcement learning of MDP domains where transition model is assumed to be inaccurate but the world is fully observed. Exogenous events in POMDP domain are more difficult to handle because we only maintain this belief state of the world rather than the exact state. Since the belief state summarises the history of the agent including previous executed actions and received observations, exogenous events will break the link between previous belief state and new belief state.

- Some execution monitoring approaches focus on the estimation of the current state of the system. They usually examine directly the low-lever components of the agent, such as a sensor, an actuator or other hardware in the system. Our approach can be seen as an execution monitoring approach that is only concerned with the high-level of plan execution. We consider how to change the plan at run-time to gain more reward in the end but ignore the monitoring of executing individual actions which is highly related to the system's hardware component. These two execution monitoring approaches affect each other in the system. For instance, once we know a sequence of actions to be executed from the high-level planner, we only need to examine the components that are associated with these actions and ignore the components that will not affect current and future execution of the plans at the low-level monitoring procedure. On the other hand, if we determine some physical components are

not working properly at the low-level monitoring process, the actions that need to use these components should be considered as unavailable in the planning or execution phases. In the future, we would like to tackle the relationship between low-level and high-level execution monitoring in an intelligent robot so as to develop a more complete execution monitoring system.

- All the experiments were done in the simulated planning domains, it would be more advantageous to see how the algorithms work on the real robot where off-line computation time and on-line computation time might both be crucial to the performance of the robot. We hope to extend our models to include the action duration at run-time. In this thesis, actions are assumed to be atomic so the execution time for each action to execute is ignored. In reality, we can take into account action duration so that the on-line computation can be done in parallel with the executing of the actions. This requires a scheduling algorithm to order the execution of the actions and on-line computation.

BIBLIOGRAPHY

[1] Alexandre Albore, Héctor Palacios, and Héctor Geffner. A Translation-based Approach to Contingent Planning. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann Publishers Inc., 2009.

[2] Eyal Amir. Planning with Nondeterministic Actions and Sensing. In *Proceedings of National Conference on Artificial Intelligence(AAAI) Workshop on Cognitive Robotics*, 2002.

[3] Brain D.O. Anderson and John B. Moore. *Optimal Filtering*. Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

[4] 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 *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 1993.

[5] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[6] Camille Besse and Brahim Chaib-Draa. Quasi-Deterministic Partially Observable Markov Decision Processes. In *Proceedings of the 16th International Conference on Neural Information Processing*, pages 237–246, 2009.

[7] Avrim L. Blum and Merrick L. Furst. Fast Planning Through Planning Graph Analysis . In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995.

[8] R. Peter Bonasso and David Kortenkamp. Using a Robot Control Architecture to Automate Space Shuttle Operations. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 949–956, 1997.

[9] Blai Bonet. Deterministic POMDPs revisited. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 59–66, 2009.

[10] Blai Bonet and Hector Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In *International Conference on AI Planning and Scheduling (AIPS)*, pages 52–61. AAAI Press, 2000.

[11] A. Botea, M. Müller, and J. Schaeffer. Using Component Abstraction for Automatic Generation of Macro-Actions. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling ICAPS-04*, pages 181–190, Whistler, Canada, June 2004. AAAI Press.

[12] Abdelbaki Bouguerra, Lars Karlsson, and Alessandro Saffiotti. Semantic Knowledge-Based Execution Monitoring for Mobile Robots. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2007.

[13] Abdelbaki Bouguerra, Lars Karlsson, and Alessandro Saffiotti. Handling Uncertainty in Semantic-Knowledge Based Execution Monitoring. In *Proceedings of IEEE International Conference on Intelligent Robots and Systems*, 2007.

[14] Craig Boutilier. Approximately Optimal Monitoring of Plan Preconditions. In *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann, 2000.

[15] Craig Boutilier and David Poole. Computing Optimal Policies for Partially Observable Decision Processes using Compact Representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI)*, pages 1168–1175, 1996.

[16] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence*, 1999.

[17] John Bresina, Richard Dearden, Nicolas Meuleau, David Smith, and Rich Washington. Planning under Continuous Time and Resource Uncertainty: A Challenge for AI. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 77–84. Morgan Kaufmann, 2002.

[18] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting Optimally in Partially Observable Stochastic Domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI)*, pages 1023–1028, 1994.

[19] Leo H. Chiang, Evan L. Russell, and Richard D Braatz. *Fault Detection and Diagnosis in Industrial Systems*. Advanced Textbooks in Control and Signal Processing. Springer, 2001.

[20] A. I. Coles and A. J. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, February 2007. ISSN 11076-9757.

[21] A. I. Coles, M. Fox, and A. J. Smith. Online identification of useful macro-actions for planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 07)*, 2007.

[22] Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina. IndiGolog: A High-Level Programming

Language for Embedded Reasoning Agents. In *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, New York, USA, 2009.

[23] Johan de Kleer and James Kurien. Fundamentals of Model-based Diagnosis. In *Proceedings of the Fifteenth International Symposium on the Mathematical Theory of Networks and Systems (MTNS 02)*, University of Notre Dame, 2003.

[24] Johan de Kleer and Brian C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97 – 130, 1987.

[25] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning With Deadlines in Stochastic Domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 574–579, 1993.

[26] Richard Dearden and Craig Boutilier. Integrating Planning and Execution in Stochastic Domains. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1994.

[27] Marie desJardins, Edmund H. Durfee, Charles L. Ortiz Jr., and Michael Wolverton. A Survey of Research in Distributed, Continual Planning. *AI Magazine*, 20(4):13–22, 1999.

[28] Arnaud Doucet and Adam M. Johansen. A Tutorial on Particle Filtering and Smoothing: Fifteen years later. In *Handbook of Nonlinear Filtering*. Oxford University Press, 2009.

[29] Arnaud. Doucet, Nando de Freitas, Neil. Gordon, and A. Smith. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[30] Richard J. Doyle, David Atkinson, and Rajkumar Doshi. Generating Perception Requests and Expectations to Verify the Execution of Plans. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 1986.

[31] Denise Draper, Steve Hanks, and Daniel Weld. A Probabilistic Model of Action for Least-Commitment Planning with Information Gathering. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 178–186. Morgan Kaufmann, 1994.

[32] Matthias Fichtner, Axel Großmann, and Michael Thielscher. Intelligent Execution Monitoring in Dynamic Environments. *Fundamenta Informaticae*, 57:371–392, Oct 2003.

[33] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[34] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3: 251–288, 1972.

[35] Robert James Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, 1989.

[36] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 2003.

[37] Christian Fritz. *Monitoring the Generation and Execution of Optimal Plans*. PhD thesis, University of Toronto, Canada, April 2009.

[38] Christian Fritz and Sheila A. McIlraith. Monitoring Plan Optimality During Execution. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 2007.

[39] Christian Fritz and Sheila A. McIlraith. Monitoring Policy Execution. In *Proceedings of the 3rd Workshop on Planning and Plan*

Execution for Real-World Systems (ICAPS07), Providence, Rhode Island, USA, September 22 2007.

[40] Erann Gat. On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1997.

[41] Moritz Goebelbecker, Charles Gretton, and Richard Dearden. A Switching Planner for Combined Task and Observation Planning. In *Proceedings of the 25th Conference on Artificial Intelligence (AAAI)*, 2011.

[42] Robert P. Goldman and Mark S. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings of Artificial Intelligence Planning Systems*, 1996.

[43] Nicola Muscettola Gregory, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[44] Stephan Gspandl, Ingo Pill, Michael Reip, Gerald Steinbauer, and Alexander Ferrein. Belief Management for High-Level Robot Programs. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 2011.

[45] Ronen I. Brafman Guy Shani and Solomon E. Shimony. Adaptation for Changing Stochastic Environments through Online Pomdp Policy Learning. In *Workshop on Reinforcement Learning in Non-Stationary Environments*, ECML, 2005.

[46] Eric A. Hansen and Zhengzhu Feng. Dynamic Programming for POMDPs using a Factored State Representation. In *Proceed-*

ings of the Fifth International Conference on AI Planning Systems (AIPS), pages 130–139, 2000.

[47] Milos Hauskrecht. Value-Function Approximations for Partially Observable Markov Decision Processes. *Journal of Artificial Intelligence Research*, 13:33–94, 2000.

[48] Jesse Hoey, Robert St-Aaubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic Planning using Decision Diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 279–288. Morgan Kaufmann, 1999.

[49] Jörg Hoffmann and Ronen Brafman. Contingent Planning via Heuristic Forward Search with Implicit Belief States. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*. Morgan-Kaufmann, 2005.

[50] Jörg Hoffmann and Ronen I. Brafman. Conformant Planning via Heuristic Forward Search: A new approach. *Artificial Intelligence*, 170(6šC7):507 – 541, 2006.

[51] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:263–302, 2001.

[52] R.A. Howard. Information Value Theory. *IEEE Transactions on Systems Science and Cybernetics*, pages 22 –26, 1966. ISSN 0536-1567.

[53] Ronald A. Howard. *Dynamic Programming And Markov Processes*. The MIT Press, Cambridge, Massachusetts, 1960.

[54] Simon Julier and Jeffrey K.Uhlmann. A New Extension of the Kalman Filter to Nonlinear Systems. In *Proceedings of AeroSense: The 11th International Symposium on Aerospace/Defense Sensing, Simulation and Controls*, 1996.

[55] Simon Julier and Jeffrey K. Uhlmann. A General Method for Approximating Nonlinear Transformations of Probability Distributions. Technical report, Robotics Research Group, Department of Engineering Science, University of Oxford, 1996.

[56] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *ARTIFICIAL INTELLIGENCE*, 101:99–134, 1998.

[57] L.P. Kaelbling. *Learning in embedded systems*. A Bradford book. 1993 publisher=Mit Press. ISBN 9780262111744.

[58] David N. Kinny. Commitment and Effectiveness of Situated Agents. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 82–88, 1991.

[59] Sven Koenig, David Furcy, and Colin Bauer. Heuristic search-based replanning. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, pages 294–301, 2002.

[60] Roman Van Der Krogt and Mathijs De Weerdt. Plan repair as an extension of planning. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05*, pages 161–170, 2005.

[61] H. Kurniawati, D. Hsu, and W. S. Lee. SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. In *Proceedings of Robotics: Science and Systems*, 2008.

[62] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An Algorithm for Probabilistic Least-commitment Planning. In *Proceedings of the Twelfth National Conference on Artificial intelligence*

(AAAI), Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. ISBN 0-262-61102-3.

[63] Michael Lederman Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, 1996.

[64] Diego Maniloff and Piotr Gmytrasiewicz. Hybrid Value Iteration for POMDPs. In *Proceedings of Twenty-Fourth International FLAIRS Conference*, 2011.

[65] Matthew T. Mason. Automatic Planning of Fine Motions: Correctness and Completeness. In *Robotics and Automation. Proceedings. 1984 IEEE International Conference on*, volume 1, pages 492–503, 1984.

[66] David McAllester and David Rosenblitt. Systematic Nonlinear Planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI)*. AAAI Press/MIT Press, 1991.

[67] Colleen E. McCarthy and Martha E. Pollack. Towards Focused Plan Monitoring: A Technique and an Application to Mobile Robots. *Autonomous Robots*, pages 71–81, 2000.

[68] John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[69] Drew McDermott. PDDL the planning domain definition language. Technical report, the AIPS-98 Planning Competition Committee, 1998.

[70] Drew McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21:35–55, 2000.

[71] Sheila A. Mcilraith. Explanatory Diagnosis: Conjecturing Actions to Explain Observations. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 167–177, 1998.

[72] Kevin P. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, Berkeley, 2002.

[73] Bernhard Nebel and Jana Koehler. Plan reuse versus plan generation: a theoretical and empirical analysis. *Artificial Intelligence*, 76(1šC2):427 – 454, 1995. ISSN 0004-3702.

[74] Sylvie C. W. Ong, Shao W. Png, David Hsu, and Wee S. Lee. POMDPs for Robotic Tasks with Mixed Observability. In *Robotics: Science and Systems*, volume 5, 2009.

[75] H. Palacios and H. Geffner. Compiling Uncertainty Away: Solving Conformant Planning Problems Using a Classical Planner (sometimes). In *Proceedings of 21st National Conference on Artificial Intelligence (AAAI)*, 2006.

[76] H. Palacios and H. Geffner. From Conformant into Classical Planning: Efficient Translations that may be Complete Too. In *Proceedings 17th International Conference on Planning and Scheduling (ICAPS-07)*, 2007.

[77] Christos Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987. ISSN 0364-765X.

[78] Scott J. Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114. Morgan Kaufmann, 1992.

[79] Mark A. Peot and David E. Smith. Conditional Nonlinear Planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1-55860-250-X.

[80] Ronald P. A. Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, Menlo Park, CA, 2002. AAAI Press.

[81] Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1025 – 1032, August 2003.

[82] Martha E. Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 183–189. AAAI Press, 1990. ISBN 0-262-51057-X.

[83] Pascal Poupart. *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes*. PhD thesis, Department of Computer Science, University of Toronto, 2005.

[84] L Pryer and G Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[85] R. Reiter. Readings in Nonmonotonic Reasoning. chapter On Closed World Databases, pages 300–310. Morgan Kaufmann

Publishers Inc., San Francisco, CA, USA, 1987. ISBN 0-934613-45-1.

[86] R Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, April 1987. ISSN 0004-3702.

[87] Ray Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[88] Stéphane Ross and Brahim Chaib-draa. AEMS: An Anytime Online Search Algorithm for Approximate Policy Refinement in Large POMDPs. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, IJCAI, pages 2592–2598, 2007.

[89] Stephane Ross, Joelle Pineau, and Brahim Chaib-draa. Theoretical Analysis of Heuristic Search Methods for Online POMDPs. In J.c. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, pages 1233–1240, Cambridge, MA, 2007. MIT Press.

[90] Stéphane Ross, Joelle Pineau, Sébastien Paquet, and Brahim Chaib-draa. Online Planning Algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 2008.

[91] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. ISBN 0137903952.

[92] Stuart J. Russell and Eric H. Wefald. *Do the Right Thing : Studies in Limited Rationality*. MIT Press, Cambridge, USA, 1991. ISBN 0262181444.

[93] Zeyn A Saigol. *Automated Planning for Hydrothermal Vent Prospecting Using AUVs*. PhD thesis, School of Computer Science, University of Birmingham, 2011.

[94] Scott Sanner and Sungwook Youn. The Seventh international planning competition, Uncertainty track, 2011. 21st International Conference on Automated Planning and Scheduling, Freiburg, Germany.

[95] M. J. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1039–1046, 1987.

[96] Guy Shani and Ronen I. Brafman. Replanning in Domains with Partial Information and Sensing actions. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[97] Gerald Steinbauer Siegfried Podesser and Franz Wotawa. Selective Belief Management for High-Level Robot Programs. In *Proceedings of the 14th International Workshop on Principles of Diagnosis*, UK, 2012.

[98] Gerald Steinbauer Siegfried Podesser and Franz Wotawa. Improving Belief Management for High-Level Robot Programs by Using Diagnosis Templates. In *Proceedings of the 14th International Workshop on Principles of Diagnosis*, UK, 2012.

[99] Reid Simmons and Sven Koenig. Probabilistic Robot Navigation in Partially Observable Environments. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[100] Richard D. Smallwood and Edward J. Sondik. The Optimal Control of Partially Observable Markov Processes Over a Finite Horizon. *Operations Research*, 21(5):1071–1088, 1973. ISSN 0030364X.

[101] David E. Smith and Daniel S. Weld. Conformant Graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 889–896, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[102] Trey Smith and Reid Simmons. Heuristic Search Value Iteration for POMDPs. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI, pages 520–527, Arlington, Virginia, United States, 2004. AUAI Press. ISBN 0-9749039-0-6.

[103] Trey Smith and Reid G. Simmons. Point-based POMDP Algorithms: Improved analysis and implementation. In *Proceedings of International Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005.

[104] Edward Jay Sondik. *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, 1971.

[105] M.T.J. Spaan and N. Spaan. A point-based POMDP algorithm for robot planning. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 3, pages 2399 – 2404 Vol.3, April-1 May 2004.

[106] Mohan Sridharan, Jeremy Wyatt, and Richard Dearden. HiPPo: Hierarchical POMDPs for Planning Information Processing and Sensing Actions on a Robot. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

[107] Gerald Steinbauer and Franz Wotawa. Enhancing Plan Execution in Dynamic Domains Using Model-Based Reasoning. In *Intelligent Robotics and Applications, First International Conference (ICIRA)*, 2008.

[108] Manuela M. Veloso, Martha E. Pollack, and Michael T. Cox. Rationale-based monitoring for planning in dynamic environments. pages 171–179. AAAI Press, 1998.

[109] David H. D. Warren. Generating Conditional Plans and Programs. In *Proceedings of the Summer Conference on Artificial Intelligence and Simulation of Behaviour (AISB)*, 1976.

[110] Greg Welch and Gary Bishop. An introduction to the Kalman filter. Technical Report TR 95-041, Department of Computer Science, University of North Carolina, 2006.

[111] David E. Wilkins. Recovering From Execution Errors in SIPE. *Computational Intelligence*, pages 33–45, 1985.

[112] D.E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann Publishers, 1988. ISBN 9780934613941.

[113] Brian C. Williams and Robert J. Ragno. Conflict-directed A* and its Role in Model-based Embedded Systems. *Discrete Appl. Math.*, 155(12):1562–1595, June 2007.

[114] Jeremy L. Wyatt, Alper Aydemir, Michael Brenner, Marc Hanheide, Nick Hawes, Patric Jensfelt, Matej Kristan, Geert-Jan M. Kruijff, Pierre Lison, Andrzej Pronobis, Kristoffer Sjöö, Danijel Skočaj, Alen Vrečko, Hendrik Zender, and Michael Zillich. Self-understanding and self-extension: A systems and representational approach. *IEEE Transactions on Autonomous Mental Development*, 2(4):282 – 303, December 2010.

[115] Sung Wook Yoon, Alan Fern, and Robert Givan. FF-Replan: A Baseline for Probabilistic Planning. In *Proceedings of the Four-*

teenth International Conference on Automated Planning and Scheduling, 2007.

[116] Sung Wook Yoon, Alan Fern, and Robert Givan. FF-Replan: A Baseline for Probabilistic Planning. In *International Conference on Automated Planning and Scheduling/Artificial Intelligence Planning Systems*, 2007.

[117] H. L. S. Younes and M. Littman. PPDDL1.0: The language for the probabilistic part of IPC-4. In *Proceedings of the International Planning Competition*, 2004.