UNIVERSITY OF BIRMINGHAM

Methods for Efficient, Exact Combinatorial Computation in Machine Learning

by

Ugur Kayas

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

University of Birmingham

July 2022

UNIVERSITY OF
BIRMINGHAM

University of Birmingham Research Archive

e-theses repository

This unpublished thesis/dissertation is copyright of the author and/or third parties. The intellectual property rights of the author or third parties in respect of this work are as defined by The Copyright Designs and Patents Act 1988 or as modified by any successor legislation.

Any use made of information contained in this thesis/dissertation must be in accordance with that legislation and must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the permission of the copyright holder.

Abstract

Combinatorial problems are common in machine learning, but they are often large-scale, exponential or factorial complexity optimization problems, for which exhaustive methods are impractical. Heuristics are typically used instead but these are not provably optimal, although they may produce a workable compromise solution in a reasonable time. On the other hand, dynamic programming (DP) is an efficient and broadly applicable tool that finds exact solutions to combinatorial problems. However, DP lacks systematicity as most algorithms are derived in an ad-hoc, problem-specific manner. In the literature, there are attempts to standardize DP algorithms, but they are either unnecessarily general (constructive algorithmics) or have limited applications to different problems (Emoto's GTA).

In this thesis, we propose a rigorous algebraic approach that systematically solves DP problems either by deriving algorithms from existing ones, or by deriving them from simple functional recurrences. The main contribution is providing novel, exact solutions for combinatorial optimization problems in machine learning and artificial intelligence. Our novel formalism largely bypasses the need to invoke the often quite high level of abstraction present in classical constructive algorithmics, as well as providing algorithms that are provably correct and polymorphic over any semiring. These algorithms can be applied to any combinatorial problem expressible in terms of semirings as a consequence of polymorphism. This approach also contributes to systematicity in embedding combinatorial constraints applying tupling to avoid the need for ad-hoc backtracking.

Acknowledgment

During my PhD, I have been extremely lucky with the people I have stumbled upon. I would like to thank Dr Max A. Little firstly for accepting me as his PhD student; and then for all the support, and guidance that he has given me throughout my PhD journey. I also could not have undertaken this journey without the academic support and great friendship of Dr Yordan Raykov. Of course this journey would not have been possible without the generous funding provided by the Ministry of National Education/Turkiye. Apparently the Turkish tax payer made a good investment.

I have also been blessed with meeting some truly kind and supportive friends. First of all, I would like to thank Adele for being my English advisor, for all the walks that we had together and for making me feel like I am the best chef in the world (as indicated by me never having any food left). Meeting with Ellen was also a great pleasure and thank you Ellen for checking the reference list of this thesis (so you know who to blame if you spot a mistake). I also need to mention Dorine for being an amazing flatmate and for teaching me how to clean the right way.

Reham was an amazing mentor, she provided me with exceptional support throughout my first year (then she disappeared suddenly). It would be rude of me not to mention that I am so thankful for Yazan, Adam and Hakan (although none of them thanked me in their thesis despite all my support and amazing friendship) for answering my random questions during my PhD. I would like to thank Wayfarers hiking society and APRS for making my social life great again. Lastly and most importantly, I would like to thank my family for the support and genetic inheritance they provided.

Contents

List of Figures

List of Algorithms

List of Tables

Notation

In this thesis, we will indicate sets and spaces with the upper case double-strike letters $\mathbb{S}$, $\mathbb{M}$ with their corresponding cardinalities $S = |\mathbb{S}|$ and $M = |\mathbb{M}|$. Algebras such as monoids, groups, semirings and graphs are given as tuples with upper-case calligraphic letters for names, e.g. $\mathcal{S}$ and $\mathcal{M}$. Integer indices and numbers are indicated by both upper and lower case letters in italic font such as n and N etc. Binary algebraic operators $\oplus$, $\otimes$, $\odot$ are given with their identity elements $\imath_{\oplus}$, $\imath_{\otimes}$, $\imath_{\odot}$ where they exist. Functions are indicated as $f : \mathbb{N} \to \mathbb{R}$ or $f \in \mathbb{R}[\mathbb{N}]$. Bold symbols are used to represent vectors such as $\boldsymbol{x}$, $\boldsymbol{y}$, and subscript notation is utilized to index elements of vectors f_n, x_1 or x_n etc. We also use subscript notation, $f_{\mathcal{G},w}$, to denote a polymorphic function f computed using the algebra $\mathcal{G}$ and mapping function w as parameters. Operators are subscripted to indicate lifting, e.g. $\oplus_{\mathcal{M}}$ is the $\oplus$ operator lifted over the algebra $\mathcal{M}$.

Chapter I

Introduction

1 Motivation

The invention of the decimal system in India around AD 600 was a significant development in quantitative reasoning [Dasgupta et al., 2008]. It would be tricky to write the number 3894 in Roman numerals as MMMDCCCXCIV or to add two Roman numerals such as MMMDCCCXCIV + MMDLXVIII. On the other hand, the decimal positional system provided the means for people to write down very large numbers and manipulate them arithmetically in an efficient manner through the use of only 10 symbols [Kleiner, 2007]. In the ninth century, Al Khwarizmi manipulated these 10 symbols with basic methods such as adding, multiplying and dividing, and these precise, unambiguous, mechanical procedures were the foundation of what we know today as *algorithms* [Dasgupta et al., 2008].

An algorithm is defined as "a sequence of computational steps that transform the input into the output" [Cormen et al., 2009]. In other words, algorithms are specific procedures that achieve the desired relationship between the input and output [Cormen et al., 2009]. Since its construction, the decimal system and associated numerical algorithms have accelerated scientific and technological developments throughout human history. Much later, the computer was created, enabling sequenced computations, in which the specific sequence can depend upon the results of previous computations [White and Downs, 1998]. Algorithms are used to generate different applications/programs which have become an important part of daily life.

A user's primary expectation of a program is to obtain a correct result. As comput-

ing applications are relied upon heavily, incorrect results can have adverse consequences. For instance, more than 700 Post Office operators in the UK, were wrongly convicted of theft, fraud and false accounting, based on evidence from company IT systems which were discovered to be faulty, in *the Post Office Horizon Scandal* between 2000 and 2014 [Moorhead et al., 2021]. Another example can be employing advanced algorithms and techniques in healthcare settings such as in the diagnosis and/ or prediction of heart disease [Hazra et al., 2017], diabetes [Iyer et al., 2015], breast cancer [Williams et al., 2015] and many more [Shailaja et al., 2018]. Either raising a false positive for serious disease or failing to detect such a condition, can have detrimental effects. Therefore, it is important that the algorithms we deploy are correct, and provably so.

The second concern relating to programs is how quickly the results can be computed. One famous example of relevance, is the *decryption* of messages from *the Enigma machine,* where rapid actions in response to the decoding of messages led to significant course changing events during World War II [Hodges, 2014]. Computing outputs (in this case, decrypted messages) within a short time was critical in that setting, and it still remains of critical importance today. For instance, advanced algorithms are used in *market investment* [Lee et al., 2019] to make predictions about stock market values. As stock market values change rapidly, computing results at the right time and quickly is important for an investor. Thus, processing time has to be taken into account when we apply advanced algorithms to real-world problems.

Computing a solution which is both correct and obtained in a reasonable time, is not necessarily straightforward. Limited time and resources always need to be considered. Most current methods for optimization in machine learning, can only provide either exhaustive solutions which are not practical or approximate results which cannot be employed for critical problems as is discussed above. In this thesis, we aim to provide a framework which solves some well-known, affecting daily-life problems, in an accurate

and fast way.

2 Contributions

This work is based on a research paper written by Little and Kayas [2021]. This thesis details the following two main contributions to the topic:

1. We derive novel general polymorphic rigorous approach which systematically solves dynamic programming (DP) problems arising in combinatorial machine learning;
    - Deriving algorithms for mixed-continuous combinatorial problems in machine learning, which solve the problem exactly and efficiently;
    - This systematicity also allows the specification of constraints to problems in a standard form, which alleviates the need for ad-hoc approaches to DP;
    - The framework naturally allows the use of the *tupling method* to avoid ad-hoc backtracking.
2. Derivations of algorithms for some widely-occurring mixed-continuous combinatorial machine learning problems, such as segmented linear regression, sequence alignment and clustering.

3 Thesis Structure

Chapter II gives all the necessary background information for the thesis through a comprehensive literature review. We introduce combinatorial machine learning briefly and then describe some relevant, fundamental concepts such as mathematical/combinatorial

optimization, and computational complexity. Subsequently, the main focus of this thesis — combinatorial machine learning problems — is introduced. Having described the main problem, we refer to the literature to discuss what is already known about the topic, and also familiarize ourselves with the mathematical tools that we will use in Chapter III.

Chapter III introduces all the technical details. First we give some fundamental descriptions and then we state the general form of Bellman's recursion. Afterwards, we use semiring homomorphims to gain some effectiveness. After representing problems in this framework, we expand the scope of applicability it by adding constraints via *lifting*. We then simplify the lifting using algebraic arguments, optimizing the efficiency of the derived algorithms. At the end of the chapter, we revisit the *tupling method* to examine what benefits we can obtain from it.

Chapter IV explores the application of the resulting formalism, to develop novel algorithms for some well-known mixed contiuous-combinatorial problems in machine learning.

Chapter V reports on computational tests of the implementation of our derived algorithms on some real and synthetic data, as well as examining the practical performance of these algorithms. We discuss the advantages and disadvantages of these derived algorithms, comparing them with existing heuristic algorithms for the same problems.

Finally, Chapter VI concludes the thesis by discussing future directions in this area posing new questions raised by the novel research presented here. We will also provide some useful supplementary information such as tables of widely used semiring structures and constraints, and their applications, in the Appendices.

Chapter II

Background

1 Related Work

The foundation of modern *machine learning* dates back to 1949 and is inspired by Donald Hebb's model of brain cell network interactions. Hebb's theories on neuron excitation and inter-neuronal communication are presented in Hebb [1949]. In 1957, the psychologist Frank Rosenblatt used Hebb's model in his textbook Rosenblatt [1957] and created the *perceptron*, which was the first neuro-computer that was deployed for *image recognition*. A decade later, *nearest neighbor algorithm* was invented by Cover and Hart [1967] to solve the *traveling-salesman problem* (*TSP*), which was the beginning of *pattern recognition* [Foote, 2022]. Indeed, TSP is originally an *optimization* problem, more specifically a *combinatorial optimization* problem (See Section 3); therefore, machine learning requires *mathematical optimization*.

Mathematical optimization has an established history dating back to 300 BC. Euclid observed the minimum distance between two points on a line; and proved that a square has the maximum area among other rectangles with the same total edge length [Simson, 1838]. In the 17th century, Newton and Leibniz independently developed *calculus* which forms the basis of continuous optimization problems [Thomas and Finney, 1961]. Studies during WW II, which required effective military responses while using limited resources, showed the direct impact and utility of mathematical optimization [Dantzig and Thapa, 2006]. Today, mathematical optimization is widely used to solve real-world problems in areas such as *economics* [Intriligator, 2002], *civil engineering* [Easa and Hossain, 2008] and *electrical engineering* [Angell and Kirsch, 2004]. Mathematical op-

timization is a vast subject and has many sub-fields such as *linear programming* (*LP*), *integer programming,* and *combinatorial optimization.*

Combinatorial optimization is a relatively young research area, which was developed in a variety of different research directions that separately take into account problems such as the TSP, *minimum spanning tree* and *optimal assignment* [Schrijver, 2005]. In 1950s, operations research received a lot of attention, which led putting these problems into one framework and links between them were established [Schrijver, 2005]. The history of combinatorial optimization cannot be held without linear programming as Kantorovich [1960] and Koopmans [1960] built the early conceptualization of LP by the motivation of combinatorial applications. After Dantzig's work on *simplex method* [Dantzig, 1990] and the generic interpretation of LP, the popularity and application areas of combinatorial optimization were extended with LP techniques [Schrijver, 2005]. For instance, Edmonds [1965] developed the *blossom algorithms* which constructs maximum *matchings* (*pairings*) on graphs. Especially for the last three decades, as combinatorial optimization are implemented into complex real world problems, there has been an increasing interest in them [Weinand et al., 2022]. As a result, combinatorial problems are also common in machine learning [Moshkov and Zielosko, 2011, Bengio et al., 2018] in the form of *feature selection,* minimum spanning tree; and the most common method to solve them is *dynamic programming* (*DP*) (See Section 7) .

The "Principle of optimality" was proposed by Richard Bellman to give a general formalism to dynamic programming problems, which breaks an optimization problem into sub-optimization problems, *Bellman's recursion* [Bellman and Dreyfus, 1962]. After Bellman's prescription, there have been many attempts to solve/generalize dynamic programming problems at various levels of abstraction in the literature. Karp and Held [1967] represent Bellman's recursion with another formalism which handles *discrete decision process* as *sequential decision process* (*DDP-SDP*) to solve problems such as

scheduling, where monotonicity justifies optimizing an associated global objective function. Helman and Rosenthal [1985] relax the associativity rule of DDP-SDP with the model which includes separating the ideas of problem structure and computation, and a special kind of homomorphic map over structural operators. Helman and Rosenthal [1985] also interpret dynamic programming with *tree* structures rather than *strings* [Curtis, 1996]. We need to emphasize that both Karp and Held [1967]'s and Helman and Rosenthal [1985]'s formalisms only work for specific *types* and are not allowed to be used with different types, as they are not *polymorphic* (See Chapter III).

Later, Oege de Moor extended Helman and Rosenthal [1985]'s structural approach to initial *data types* [De Moor, 1991, Bird and De Moor, 1996]. In the work of De Moor [1991], an abstraction of DP is based on *category theory* and *relations* such as inequalities, which are typical operations for optimization applications of DP algorithms such as expectations for parameter estimation in *natural language processing* problems [Li and Eisner, 2009]. In order to address significant non-optimization uses of DP, it is unclear how to further adapt this relational framework to arbitrary *semirings* (See Definition 4) despite it being polymorphic. Goodman [1999] and Huang [2008] use implicit different semirings with their polymorphic structures in DP algorithms for specialized application domains such as natural language processing across graphs and hyper-graphs. However, they only address special DP algorithms, which are not general.

Alternative to dynamic programming, *divide-and-conquer* (*divide et impera*) and *greedy* are two other approaches to solve combinatorial problems. Unlike dynamic programming, divide-and-conquer breaks down a problem into smaller pieces and then solves each one separately rather than solving it sequentially [Cormen et al., 2009]. An early implementation of divide-and-conquer algorithms has an old history going back to Galois, which is now used in fast Fourier transform [Heideman et al., 1985]. In 1945, John von Neumann developed the *merge sort* algorithm based on the divide-and-

conquer manner [Kamlesh et al., 2014]. Divide-and-conquer algorithms provide various advantages such as structural simplicity, parallel implementation and diversity of applications from *social network partitioning* [Pujol et al., 2009] to *classification* [Frosyniotis et al., 2003] [Smith, 1985]. However, these algorithms can be computationally demanding and the structure of the algorithm differs depending upon the problem.

The other important method, greedy algorithms were conceptualized by Edsger Dijkstra in 1959 to solve spanning tree problems [Dijkstraa, 1959]. In the same decade, [Kruskal, 1956] and Prim [1957] also developed greedy algorithms to solve shortest path problems. Later, Cormen et al. [2009] proposed a recursive structured version of greedy algorithms. Greedy algorithms are used for different areas such as *DNA sequencing* [Frieze and Szpankowski, 1998, Kececioglu and Myers, 1995] and *computational geometry* [Dickerson et al., 1997, Guibas et al., 1993]. However, the main disadvantage of greedy algorithms is that they may fall into *local optimal* results, and local optimal solutions are not always equal to the *global optima*.

Bird-Meertens formalism (BMF) or Emoto et al. [2012]'s *generate-test-aggregate* (*GTA*) *algorithm* are two different principles of deriving efficient and correct algorithms from correct but inefficient specifications [Bird and Meertens, 1987, Backhouse, 1988]. BMF formalism is also referred to as *constructive algorithmics* as it constructs algorithms from specifications [Hu et al., 1998]. In other words, constructive algorithmics was designed by Richard Bird and Lambert Meertens to develop an efficient program from a correct but inefficient initial specification (*brute-force* for instance) by applying a series of meaning preservation transformations [Gibbons, 2019]. Constructive algorithmics and their properties have been widely studied in the literature [Bird and De Moor, 1996, Jeuring, 1993, Bird, 1987, 1989]. The BMF consists of some fundamental theorems such as *fusion* (combining two recursive computations into a single one) and *Horner's Rule* (leveraging distributivity in polynomial division) to solve recurring

optimization problem structures [Gibbons, 2019].

Another approach, Emoto et al. [2012]'s algorithm, specifies brute-force problems with a constraint under their GTA framework. Similar to brute-force, GTA generates every single possible configuration in the search space (generate), but additionally filters out those that do not match with the constraint (test), and then finally provides the best option within the elements that survived filtering (aggregate). Emoto et al. [2012]'s GTA algorithm applies fusion theory to combine *generator* and *aggregator*, and also embeds a *filter* on the combination to gain computational savings by preventing the generation of unnecessary subsets [Liu et al., 2014, Emoto et al., 2012]. Nonetheless, this algorithm requires the problem to be in a list structure and all the components need to be linked to each other via list homomorphisms.

Regarding the basic motivation, all these frameworks and our thesis aim to increase the efficiency of problems by considering their exactness. Both BMF and GTA for solving DP (and other) problems are based on the idea of generate-test-aggregate. Another way to say this is that they all exploit the same overall pattern of specifying provably exact algorithms through brute-force enumeration, followed by constraint satisfaction (filtering), and then aggregation (accumulation). Subsequently, they all use some form of fusion (and possibly other equational equivalences) and algebraic features to derive efficient algorithms which are exact by construction.

Both BMF and GTA are created on the list structure which limits the extension of the area of application because many problems in ML are not structured as computations over (join) lists. Bird-Meertens is a compact notation for dealing with list computations and equational reasoning in a point-free way for functional programs expressed in this notation. However, we will not need the full generality of computations described in the BMF. While Emoto's GTA framework is based on (free) join list homomorphisms because it is restricted to lists, their goal is to come up with easily

parallelizable algorithms, which explains the restriction to join lists.

We will use already existing algorithms to build up our method like the examples in the literature. More specifically, Morihata et al. [2007] derive efficient algorithms for DP problems from existing naive but inefficient algorithms by using functions, logic and *the strictly monotone property* (preserving the order of sets). Another example is that Mu and Oliveira [2012] derive programs from specifications which take the form of *Galois connections* (functions between ordered sets). Liu and Stoller [2003] propose a method to solve DP problems, which is based on *static incrementalization* (a recursive solving method that uses the *memoization* technique).

In recent years, solving combinatorial problems in machine learning exactly and efficiently is an increasing trend. Mensch and Blondel [2018] propose an approach which uses the continuous relaxation strategy in semirings (smoothing the max operator in the DP recursion). Choi et al. [2020] use probabilistic circuits to reach exact probabilistic inference in linear time. Parallel to our method, Belle and De Raedt [2020] use semiring features to create a semantic framework for generalized sum-product problems. However, none of these studies mention about constraint for combinatorial problems. Here, we will propose a different novel approach to solve combinatorial problems in machine learning exactly and efficiently, which also proposes a generic way to embed constraint. In the next section, we will start delivering the essential background of our framework.

2 Machine Learning

The study that investigates regularities in data through the use of algorithms is known as *pattern recognition* [Bishop, 2006]. An examples of pattern recognition problems are Kepler's astronomical observations to determine the rules behind planetary motions in the 17th century, and recognizing cars from digital number plate images automatically

[Bishop, 2006].

Since today's pattern recognition problems are of increasing complexity, hand-crafted solutions are not able to respond to the demand. From the 1940's, *machine learning* (ML) emerged as the study of finding (or "learning") patterns by using mathematical algorithms implemented in software, which improve themselves automatically from experience (or "knowledge") to give useful predictions [Little, 2019]. Widespread accessibility to the internet has increased the importance and use of machine learning, in various areas such as *digital health monitoring* [Badawy et al., 2017], *genetics-genomics* [Libbrecht and Noble, 2015], *image processing* [Singh, 2019] and *analysis of human behavior* [Qarout et al., 2020].

By contrast to classical programming, a machine learning system is trained rather than programmed. Given a task, an ML system is presented with enough training examples so as to identify the underlying pattern in these examples, which enables the system to determine the rules for automating the task [Chollet, 2021]. Machine learning systems predict the structure (or the pattern) from the relationship between input and output data. As an analogy, ML algorithm determines the recipe from the ingredients list and the prepared dish; whereas classical programming produces the dish with the ingredients and a recipe.

Traditionally, machine learning has been divided into three fundamental kinds of data analysis tasks: *regression, classification* and *dimensionality reduction & clustering* [Mitchell, 1997]. In this thesis, our main focus will be on regression and clustering.

2.1 Regression

Regression is a widely used tool for analysis and prediction, which goals to find the associative mapping $f : \mathbb{S}^P \to \mathbb{S}$ between P-dimensional data $\boldsymbol{X} = \{x_p^{(i)}\}_{i=1,p=1}^{N,P}$ and desired continuous targets $\boldsymbol{Y} = \{y^{(i)}\}_{i=1}^{N}$ where N is the number of observed samples

[Sykes, 1993, Raykov and Saad, 2022]. Once we find the relationship between $\boldsymbol{X}$ and $\boldsymbol{Y}$ we can make predictions about $\boldsymbol{Y}$ with given $\boldsymbol{X}$. In the literature, regression has been studied in different forms such as *linear regression* [Weisberg, 2005], *polynomial regression* [Cheng and Schneeweiss, 1998], *logistic regression* [Wright, 1995] etc..

Linear regression which is a widely utilized form of regression, fits a line through data to establish a relationship between covariates and their effects, as shown in Figure 1(b). Here, a line $\boldsymbol{y} = \alpha\boldsymbol{x}+\beta$ where $\boldsymbol{x}$ a vector with size N, α represents the slope and β is the intercept, fits through data subject to an error function (commonly sum-squared error). Let $\mathbb{P}$ be a set of points in the plane, denoted $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$; and L is the line, and then the estimation error (E) of L with respect to $\mathbb{P}$ is the sum of its squared distances to the points in $\mathbb{P}$:

$$E(L, \mathbb{P}) = \sum_{n=1}^{N} (y_n - \alpha x_n - \beta)^2 \tag{2.1}$$

where $x_1 < x_2 < \ldots < x_N$. The main goal is to minimize this fit error [Kleinberg and Tardos, 2006].

2.2 Clustering

Clustering is a conventional machine learning technique that is mainly used to determine assignments from unlabeled data in many areas [Raykov et al., 2016] such as *social media analysis* [Luo et al., 2014], *neuroscience* [Yang et al., 2014], *bio-informatics* [Saeed et al., 2012] and *image processing* [Bogner et al., 2013]. In other words, clustering is the task of partitioning data into small groups where data points associated with each cluster share more structure between each other then with the rest of the data. There are a very large number of clustering techniques such as *k-means* [Bishop, 2006], *DP-means* [Kulis and Jordan, 2011] and *Maximum a-posteriori DP* (*MAP-DP*) [Raykov et al.,

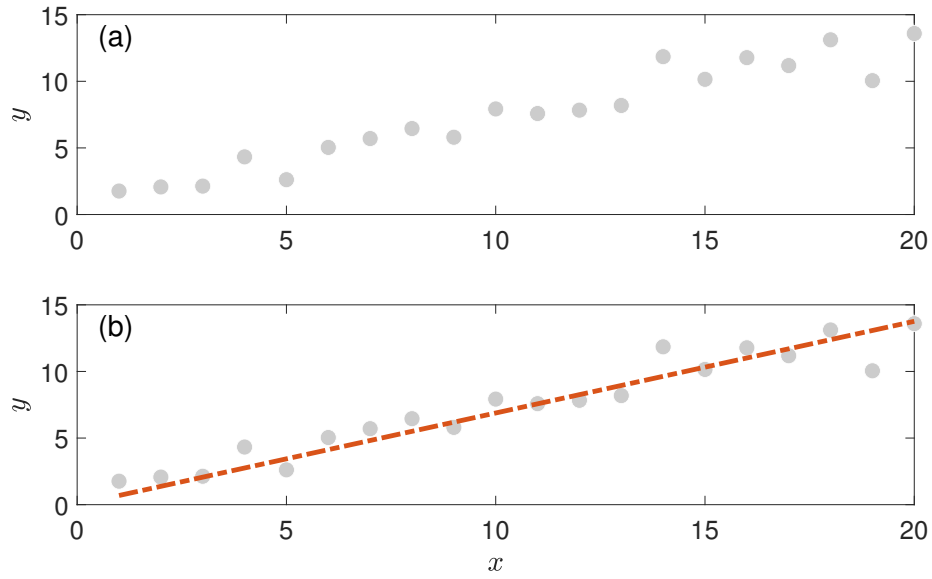

Figure 1: An illustration of linear regression. Arbitrary data in (a), and a line fitting this data in (b).

2016].

Clustering methods can be divided into two principal categories: *hierarchical* and *partitional* [Omran et al., 2007]. Here we will examine partitional clustering methods with distance-based similarity measures. As mentioned previously, clustering is the process of identifying groupings or clusters that occur naturally within a set of data. Similarity measures are, therefore, a fundamental component of most clustering algorithms [Omran et al., 2007]. One of the most applied similarity measures is distance measure with *Euclidean distance* being a popular measurement of choice:

$$||x_n - y_n|| = (1/2)\sum_{n=1}^{N}(y_n - x_n)^2 \tag{2.2}$$

which is a special case of *Minkowski metric* [Jain et al., 1999].

Highly dimensional data can be transformed into a useful representation with reduced dimensionality in a process referred to as dimensionality reduction [Van Der Maaten

et al., 2009]. One of the most used dimensionality reduction method is *feature selection*, is used to control the number of input variables for a predictive model [Liu and Motoda, 2007]. Specifically, in real-world applications, there may be an intractably large number of input variables. Using each variable is computationally challenging and reduces the interpretability of the prediction algorithm. Therefore, the number of features needs to be reduced. For instance, a real life example could be that of a company only interested in a specific criterion (i.e. programming skills) within a group of features (from candidate CV) in order to select suitable applicants for a software post at the company. As clustering is employed to group large scale data into certain number of clusters, clustering is a kind of dimensionality reduction

Although there are many different machine learning problems, common to nearly all machine learning algorithms is a *loss function* which is minimized over a training data set [Sra et al., 2012]. In this context, a loss function measures the algorithm's performance by aggregating over mismatches between the expected and actual results [Chollet, 2021]. The measurement can be used as feedback so that algorithms can adjust their parameters to reach an optimal solution. The "learning" part actually comes from these adjustment steps [Chollet, 2021]. As a result, machine learning cannot generally be separated from *optimization,* which is the topic of the next section.

3 Mathematical Optimization

Optimization is the discipline of determining the 'best' result among a set of other available alternatives under given constraints [Dantzig and Thapa, 2006]. Formally, all *constrained optimization problems* can be represented in the following general form:

$$\max \text{ (or } \min) f(\boldsymbol{x}), \quad \boldsymbol{x} = (x_1, x_2, \ldots, x_N) \in \mathbb{X} \tag{3.1}$$

subject to the constraints:

$$\begin{aligned} g_m(\boldsymbol{x}) &\leq 0, \quad m = 1, 2, \ldots, M \\ h_r(\boldsymbol{x}) &= 0, \quad r = 1, 2, \ldots, R \end{aligned} \tag{3.2}$$

where $f(\boldsymbol{x})$ is the *objective function* or *cost function*, $g(\boldsymbol{x})$ and $h(\boldsymbol{x})$ represent the *inequality constraint functions* and *equality constraint functions* respectively [Snyman, 2005]. The objective function $f(\boldsymbol{x})$ can be either linear or nonlinear whereas the type of constraint functions affects the form of the problem directly [Yang, 2008]. Here, x_n of $\boldsymbol{x} \in \mathbb{X}$ are called as *decision variables,* and they can be either *continuous* e.g. $\mathbb{X} = \mathbb{R}$ such that $x_n \in \mathbb{R}^N$, or *discrete* or *combinatorial* when $\mathbb{X}$ consists of finite set, or countably finite as $\mathbb{X} = \{0, 1\}^N$ such that $x_n \in \{0, 1\}$ [Yang, 2008]. $\mathbb{X}$ is called the *search space*, while objective function values form the *solution space* [Yang, 2008]. The optimization problem essentially maps the search space into a solution space. For instance:

$$\min(x^2 + 1), \quad x \in \mathbb{R} \tag{3.3}$$

subject to:

$$x \in (-\infty, -1] \tag{3.4}$$

is a *minimization problem* that aims to find a *feasible value* (one which satisfies the constraint equations) where the objective function is minimal in $\mathbb{R}$. The notation is equivalently represented as

$$\underset{x \in (-\infty, -1]}{\arg\min} \, (x^2 + 1) \tag{3.5}$$

where arg min is argument of the minimum, which expresses the value(s) that minimizes (or maximizes with arg max) the objective function under the constraint. Here, as we go through in the entire search space, the optimal solution is called *global optima* (or *global optimal value*). However, if it was the optimal value for a certain neighborhood (or part) of search space, then the solution would be named as *local optima* (or *local optimal value*). In some cases, the local optimal value can be equal to the global optima [Pedregal, 2006]. An optimization problem which does not require constraints is called an *unconstrained optimization problem* [Snyman, 2005]. In this thesis, we will focus on; as a sub-field of mathematical optimization, mixed continuous-combinatorial optimization as well as its connection with machine learning problems.

3.1 Combinatorial optimization

Combinatorics is the mathematical study of the ordering, grouping, arrangement, or collection of a discrete set of objects, usually of a finite size [Lawler, 2001]. As discussed, combinatorial optimization aims to find the 'best' order, group or arrangement of a discrete set of elements. Well-known combinatorial optimization include the traveling-salesman and *0-1 knapsack* problems. The 0-1 knapsack problem aims to fill a bag with a given set of items, each with a given weight and non-negative value, by subject to a weight limit, so as to maximize the total value of the items. As these combinatorial problems are mathematical optimization problems, (3.1) and (3.2) are reformulated as (3.6) and (3.7) with N number of items and weight limit W:

$$\max \sum_{n \in 1,2,..,N} x_n v_n, \quad x_n \in \{0,1\}, \quad v_n \in \mathbb{R} \tag{3.6}$$

subject to:

$$\sum_{n \in 1,2,\ldots,N} x_n w_n \leq W, \quad w_n \in \mathbb{Z} \tag{3.7}$$

where x_n is the number of instances of item n with assigned a weight w_n and a value v_n. In (3.7), $x_n \in \{0, 1\}$ says that an item can be chosen only once or not at all. We will come back to the knapsack problem later on. We will briefly discuss other combinatorial problems. The following list suggests that combinatorial problems can be grouped according to their *combinatorial object,* e.g. *sorting, permutations, partitions, subsets, calendars* and *schedules.*

- **Sorting** deals with the arrangement of a set of elements in increasing or decreasing order [Skiena, 1998].
- **Searching** investigates methods that locate and recall information stored in data structures [Skiena, 1998]. There are very common problems in computer science, often dealing with tree structures to store data.
- **Median and selection** aims to find the item that has the middle value, or a certain sorted rank, within the list of items [Skiena, 1998].
- **Permutations** permutes N number of items from a list of numbers [Skiena, 1998]. For example, arranging a seating plan of a group of people at a round table.
- **Subsets** refers to the set of all possible selections of elements of a set, of which there are 2^N for a set with N number of elements [Skiena, 1998].
- **Partitions** finds possible ways of separating a set into groups. For example; the partitions of 4 into groups which, when added sum up to 4, are $\{4\}, \{3, 1\}$,$\{2, 2\}$, $\{2, 1, 1\}$ and $\{1, 1, 1, 1\}$.

- **Graphs** are objects with vertices connected by edges, where the edges may have values [Skiena, 1998].

All the kinds of problems above can be "solved" with either *exact* or *heuristic* methods [Lawler, 2001]. The main difference between these two methods is that exact methods guarantee that the solution is globally optimal, whereas heuristic methods do not provide any guarantee that the solution is globally optimal [Rothlauf, 2011]. A good example of an exact method is perhaps the most obvious method, *brute-force*. A brute-force algorithm finds the optimal result by evaluating all possible combinatorial configurations. On the other hand, *greedy algorithms* are an ideal example of a heuristic algorithm, which provide a result by identifying locally optimal solutions at each step, which does not guarantee the global optimality.

To illustrate the above ideas, let us take a toy example of the 0-1 knapsack problem with items x_1 and x_2 which have $(v_1, v_2) = (20, 40)$ values and $(w_1, w_2) = (2, 5)$ weights, subject to the weight constraint $W = 6$. A brute-force algorithm would check every single weight configuration in the search space, e.g. the set of pairs $(x_1, x_2) = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$, and can find the optimal solution by selecting the pair with the largest total value which satisfies the constraint. A greedy algorithm would check the ratio of value and weight and then place items in the bag beginning with the one with the highest ratio. In this case, the ratios are $r_1 = 10$ and $r_2 = 8$, and the algorithm naturally selects only the first item as it has the larger ratio value such that the weight limit can not be exceeded. However, the optimal solution is a bag with the second item only as it has a higher value.

As this example demonstrates, heuristic methods are not guaranteed to find the optimal solution. Exact solution methods have the problem that, checking all possible combinatorial configurations exhaustively, may not be practical for a much larger data set. This raises the question of, "What is practical?" or "Is there any threshold that

we can apply to separate methods which practical from impractical?". Answering this question is the topic of the next section.

4 Concepts in Computational Complexity

A key criterion for evaluating an algorithm's performance is its efficiency. In this context, efficiency can be examined in terms of both *time* and *space*. The number of fundamental computational steps that are needed to execute an algorithm is referred to as time efficiency. Space efficiency refers to the number of fundamental data structures that have to be stored in computer memory to execute a program [Goldreich, 2008]. The required time and space are measured with respect to N, the size of the input data for the algorithm. More data usually demands more computational space and time [Goldreich, 2008].

Time complexity is determined by reference to the *worst-case* scenario. For instance, to find a specific person from a group of N people , we would need to ask each individual; the best-case scenario would be finding the person in the first step. However, there is no guarantee of finding the desired person in any specific turn, which leads us to argue that the worst-case scenario is that we must question every individual. Therefore, the order of the time complexity equals the size of the possible steps, N, which it the number of people in the group.

In general, the specific amount of time or space required to execute an algorithm, is a property of a specific hardware or software implementation. Therefore, instead, we are more interested in the *relative* computational resource requirements under changing algorithm input size [Little, 2019]. For instance, given a specific hardware setting, one algorithm (a) requires $a(N) = 2N^3$ fundamental steps to be executed and another algorithm (b) needs $b(N) = 1/2N^3$ steps. Thus, although algorithm (a) requires four

times more fundamental steps than algorithm (b), as N increases, this 1-4 ratio does not change. We therefore say that both algorithms have the same *time complexity of order* $O(N^3)$.

Definition 1. (Big-O Notation): The order of time complexity is symbolized with big-O notation, which is defined as $f(x) = O\Big(g(x)\Big)$ as $x \to \infty$. This indicates that there is an upper bound which satisfies

$$\Big|f(x)\Big| \leq c \times \Big|g(x)\Big|, \quad \text{for all } x \geq x_0 \quad \text{such that } x_0 \in \mathbb{R} \tag{4.1}$$

where c is a positive constant [Skiena, 1998].

As the degree of a function is determined by the most dominant term, big-O notation measures only the highest degree term. For instance, the time complexity order of an algorithm which takes $2N^3 + 7N$ steps, is $O(N^3)$ as $2N^3$ is the dominant element.

The order of the complexity is determined by the class of the function measuring it. For instance, an algorithm with $O(N)$ complexity has *linear* time complexity. Table 1 gives a hierarchy of computation complexity.

Order	Name
$O(1)$	constant
$O\Big(\log(N)\Big)$	logarithmic
$O(N)$	linear
$O\Big(N\log(N)\Big)$	log-linear
$O(N^k),\ k > 1$	polynomial
$O(k^N),\ k > 1$	exponential
$O(N!)$	factorial

Table 1: Computational complexity order classes, in big-O notation. Most efficient classes at the top, down to the least efficient at the bottom [Little, 2019].

As the table demonstrates, constant time complexity requires the least computa-

tional time effort, by contrast, factorial time complexity demands the highest. Figure 2 gives a visual demonstration of the computational complexity classes listed in Table 1 for small sizes. From the figure, it is clear that exponential and factorial time complexity orders increase dramatically with N, whereas $\log(N)$ rises very slowly.

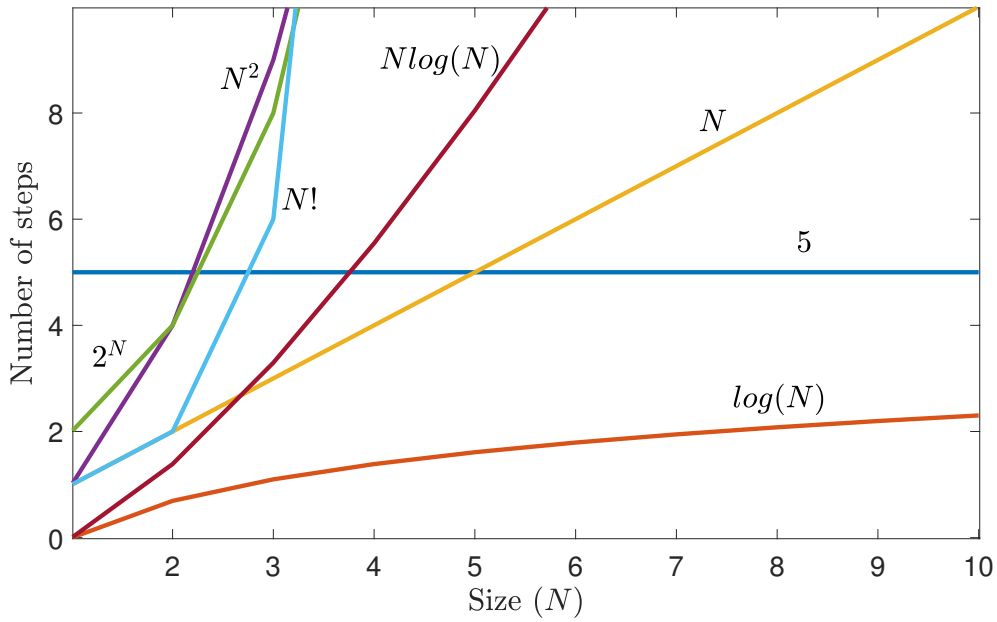

Figure 2: A representation of computational complexity classes listed in Table 1 for small sizes using assigned curves.

Efficiency for small N can be quite specific. For example, consider two different algorithms, one taking $f(N) = \frac{3}{20}N^2$ number of steps and another $g(N) = 20N - 50$ steps. Which one is more efficient? For small N it depends on the value of N. In this case, for $N \leq 130$, algorithm $g(N)$ is more efficient, but for $N > 130$ algorithm $f(N)$ is more efficient [Dasgupta et al., 2008], see Figure 3.

Under more realistic circumstances, there are significant practical differences between the order classes. For example, exponential time computations require extremely large amounts of time. As a specific example, finding the Nth element of the *Fibonacci sequence* with Algorithm 1 requires $2^{0.694N}$ elementary computer steps to solve [Das-

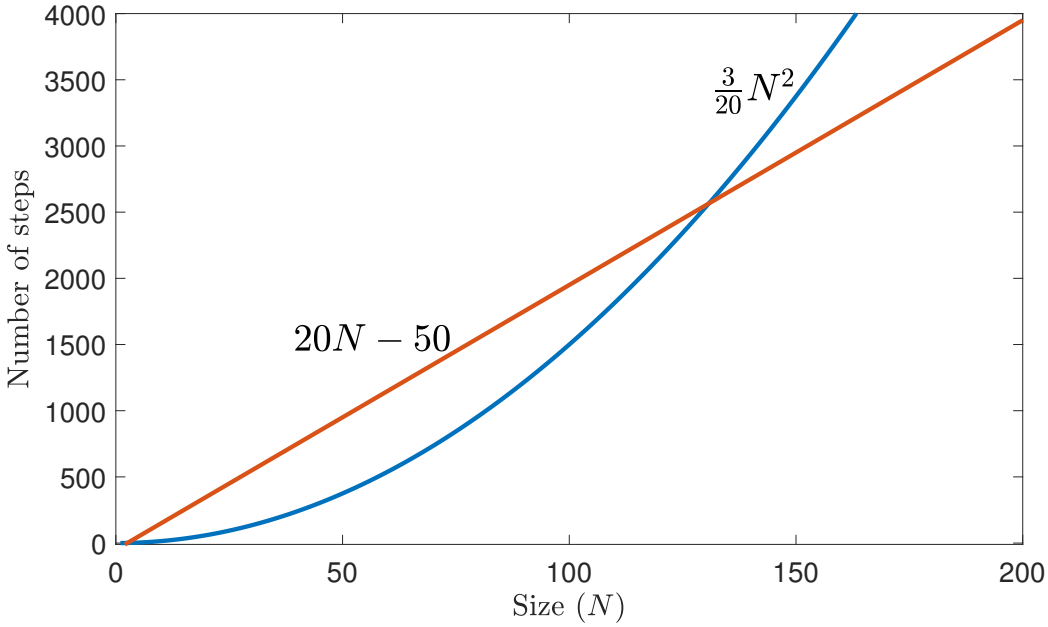

Figure 3: The required computational steps for $f = \frac{3}{20}N^2$ and $g = 20N - 50$. The efficiency performance entirely depends on the size of N e.g. for $N \leq 130$, algorithm $g(N)$ is more efficient, but for $N > 130$ algorithm $f(N)$ is more efficient.

gupta et al., 2008]. The Fibonacci sequence is a sequence of numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \tag{4.2}$$

where each element is the sum of its two immediate predecessors [Dasgupta et al., 2008]. Algorithm 1 requires 2^{27} computational steps to find the 40th element of Fibonacci sequence, which takes 134.572 seconds on a *Intel Core i7 2.80 GHz, 8 GB RAM* machine. Although this seems like a reasonable time for a relatively small example, this would be prohibitive for even $N = 50$, which takes 4.5 hours to perform with the same machine.

Algorithm 1 Exponential-time complexity Fibonacci algorithm [Dasgupta et al., 2008]

function fib1 (N)
  if $N = 0$; **return** 0
  if $N = 1$; **return** 1
  return $f[N-1] + f[N-2]$

Algorithm 1 is not practically feasible has $O(2^N)$ computational complexity. The reason for this is that many computations are repeated. However, it is possible to enhance the performance if the intermediate results are stored and re-used, as shown in the more efficient Algorithm 2. This storage and re-use technique is called *memoization.*

Algorithm 2 Linear Fibonacci algorithm [Dasgupta et al., 2008]

```
function fib2 (N)
    if N = 0; return 0
    f[0] = [0], f[1] = [1]
    for i = 2...N
        f[i] = f[i − 1] + f[i − 2]
    return f[N]
```

In this case, memoization reduces the complexity to linear time, $O(N)$, so that Algorithm 2 executes the same problem ($N = 50$) within a second with the same machine [Dasgupta et al., 2008]. This large difference in computational steps and the time required to execute them illustrates the difference between computational complexity classes. To understand these classes, let us consider decision problems which are identified as "yes, or no questions" of the input values. For instance, finding prime numbers from a list is a decision problem as elements are checked regardless of whether they are prime numbers or not. So far, we have considered optimization problems and the complexity theory is on decision problems. However, we can always recast an optimization problem as a decision problem by adding a constraint and asking if the solution satisfies the constraint [Cormen et al., 2009]. The traveling-salesman problem (TSP) is a good example of this transformation. The TSP requires finding the shortest possible tour of a salesman who can visit each city in the tour only once [Potvin, 1993]. Once we add a constraint e which indicates that the shortest TSP tour must spend less or equal energy than e, the TSP become a decision problem.

Another important point we need to highlight is the difference between *deterministic* and *non-deterministic algorithms.* The main difference between these two algorithms is

how solution steps are executed. The deterministic algorithms execute one step in each iteration, whereas non-deterministic methods can execute multiple steps in each turn. Nonetheless, non-deterministic algorithms are theoretical models which are impractical [Cormen et al., 2009]. Emoto et al. [2012]'s generate-test-aggregate algorithm is a nice illustration of how non-deterministic algorithms work, as GTA is an implementation of brute-force solution search algorithms, and brute-force algorithms have a close conceptual fit to non-deterministic implementation [Floyd, 1967]. After giving the essential background for the complexity classes, we can divide them as below:

1. ***Polynomial-time*** (or ***P***) problems represent decision problems which are solvable in polynomial time with deterministic algorithms. Polynomial class problems are identified as "tractable" problems [Harel and Feldman, 2004].

2. ***Non-deterministic Polynomial-time*** (or ***NP***) problems are identified as the problems solved by non-deterministic algorithms in polynomial time. With deterministic algorithms, solving NP problems may require exponential time complexity, but they are verifiable in polynomial time. What we mean by verifiable is that it is possible to check the accuracy of a given solution for an NP problem. For example, let us consider the TSP again where we want to check if the given solution satisfies the energy constraint. We can easily calculate the amount of energy that the proposed path requires and see if it is below or above the limit, whereas solving the problem with a brute-force algorithm could require factorial time complexity [Cormen et al., 2009].

3. ***Non-deterministic Polynomial-time Complete*** (or ***NP-Complete***) problems are the hardest NP problems. A decision problem L is accepted as an NP-complete problem if it is in NP and all problems in NP are *reducible* to L in polynomial time. Here, the reduction is a transformation of one problem to another to use the related first algorithm in the second one. NP-complete problems are accepted as "intractable" problems [Cormen et al., 2009, Harel and Feldman, 2004].

4. ***Non-deterministic Polynomial-time hard*** (or ***NP-hard***) problems also require reducibility. Same as above, a decision problem M is accepted as an NP-hard problem when all problems in NP-hard are reducible to M in polynomial time [Cormen et al., 2009].

To sum up, the complexity of these classes is ordered as P, NP, NP-complete, and NP-hard, respectively. NP-hard problems do not have to be in NP, but they may be included in NP. NP-complete problems lie at the intersection of NP and NP-hard problems. P problems are considered "easy" as they can be solved efficiently, while NP-complete and NP-hard problems are considered the most difficult or intractable problems. For example, combinatorial problems with exhaustive solutions, such as the solution for the 0-1 knapsack problem, are NP-hard problems, which require exponential time complexity.

Where possible, more efficient algorithms are always preferable for reasons shown above. Even if an initial formulation of an algorithm is not efficient, there are ways to improve its efficiency. This is possible by using special techniques such as memoization described above for Fibonacci numbers. In this same manner, the divide-and-conquer method could be another example. As mentioned, this method simply transforms complex algorithms into manageable components to solve them more efficiently [Roman, 2017]. In detail, divide-and-conquer is a solving technique which divides a problem into sub-problems solving them *independentely* before combining them at the end to obtain computational save [Bentley and Shamos, 1976].

Our last example is the computation of the *discrete Fourier transform*. The default algorithm has time complexity $O(N^2)$, but this can be reduced $O(N \log N)$ by exploiting mathematical symmetries in the problem [Little, 2019]. Using this technique a polynomial time complexity class algorithm can be reformulated as a log-linear complexity algorithm. However, not all mathematical problems have such highly efficient

algorithms.

5 Combinatorial Machine Learning (ML) Problems

The traveling-salesman problem (TSP) an example of a well-known combinatorial problem. To solve the TSP exactly, we need to consider every possible sequence of cities visited, which requires factorial time complexity. For instance, it would require $(N)! = 11!$ different possible sequences if $N = 11$ cities are given [Potvin, 1993]. There are many different heuristic approaches to solving the problem [Nilsson, 2003]. As an example, *the nearest neighbor method* works as follows; firstly, it selects a random city, and then finds the nearest unvisited city and repeats this process until there are no cities remaining unvisited, before finally returning back to the first city [Nilsson, 2003]. Nearest neighbor performs executes in polynomial time, in fact $O(N^2)$ for N cities, but it cannot guarantee optimality as the initial city is chosen completely at random [Nilsson, 2003].

There are many combinatorial problems in ML, which makes substantial use of combinatorial objects, such as subsets in feature selection [Hall and Smith, 1998] and partitions in classification [Chou, 1991]. As we have seen in knapsack and TSP examples; heuristics simplistic and exact methods are unsatisfactory as either the solution is not guaranteed to be optimal, or it is guaranteed optimal but the computations required to find the solution are impractical. As the machine learning algorithms are employed in many critical areas, this lack of provable, efficient optimality is now a big challenge in real-world applications [Murray et al., 2005]. One of the most well known combinatorial machine learning problems is *k-means clustering.*

5.1 k-means Clustering

A widely employed clustering algorithm is that k-means clustering is formalized as given a set of data points $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$; to find labels $\ell_1, \ell_2, \ldots, \ell_k$ to minimize the following objective function:

$$\min_{\{\ell_c\}_{c=1}^{k}} \sum_{c=1}^{k} \sum_{\boldsymbol{x} \in \ell_c} ||\boldsymbol{x} - \boldsymbol{\mu}_c||_2^2 \tag{5.1}$$

where $\boldsymbol{\mu}_c = \frac{1}{|\ell_c|} \sum_{\boldsymbol{x} \in \ell_c} \boldsymbol{x}$. This unsupervised methods start with selecting k number of centroids arbitrary, and then assigns each data points $\boldsymbol{x}_i$, $i \in 1, 2, \ldots, N$ with a centroid through reducing the in-cluster sum of squares $\ell^*(i) = \text{argmin}_c ||\boldsymbol{x}_i - \boldsymbol{\mu}_c||_2^2$. The centroids are kept reassigned and the mean of each clusters $\boldsymbol{\mu}_c$ for all c are updated until reaching a minimum value of the objective function [Kulis and Jordan, 2011]. Each step executed to reach a solution are named as *iteration.* This process performs with restarts to get better solution.

As the main goal is to minimize the objective function, k-means clustering can be seen as an optimization problem. However, k-means algorithm is also a heuristics of which the optimality is not guaranteed. As the initial centroids are arbitrary, algorithms can not satisfy the clustering repeatability, and each turn could end up with a different result. For instance, we can find an optimal solution after M number of restarts, but the most optimal solution might be found in the $(M+1)$th turn.

In conclusion, existing methods for solving combinatorial machine learning problems do not meet with the modern-day demands on reliability and efficiency. With this in mind, the motivation for this thesis is to seek methods for solving combinatorial machine learning problems exactly, under practical time constraints. To achieve this, we will construct a theoretical framework which aims to create a practical synthesis of successful solutions to this problem. To do this, in the next section, some fundamental

abstract algebraic structures will be considered, which will form a crucial part of this synthetic framework.

6 Abstract algebra: monoids, groups and semirings

To build up the theory, we begin with some fundamental definitions.

Definition 2. (Monoid): Let $\mathbb{M}$ be a set, and $\oplus$ an operation on $\mathbb{M}$. $\mathcal{M} = (\mathbb{M}, \oplus)$ is called as a *monoid* if the following properties are satisfied;

- (*closure*) for any $x, y \in \mathbb{M}$, $x \oplus y \in \mathbb{M}$,
- (*associativity*) for any $x, y, z \in \mathbb{M}$, we have $(x \oplus y) \oplus z = x \oplus (y \oplus z)$,
- (*identity left and right*) there exists identity element $\imath \in \mathbb{M}$ such that for any $x \in \mathbb{M}$, we have $\imath \oplus x = x \oplus \imath = x$ (also symbolized as $\imath_{\oplus}$) [Lang, 2012].

For instance, $(\mathbb{R}\backslash\{0\}, \times)$ is a monoid as it is closed, associativity applies and 1 is the identity element.

Definition 3. (Group): Let $\mathbb{M}$ be a set, and $\otimes$ an operation on $\mathbb{M}$. $\mathcal{M} = (\mathbb{M}, \otimes)$ is called as a *group* if the following properties are satisfied;

- (*closure*) for any $x, y \in \mathbb{M}$, $x \otimes y \in \mathbb{M}$,
- (*associativity*) for any $x, y, z \in \mathbb{M}$, we have $(x \otimes y) \otimes z = x \otimes (y \otimes z)$,
- (*identity left and right*) there exists identity element $\imath \in \mathbb{M}$ such that for any $x \in \mathbb{M}$, we have $\imath \otimes x = x \otimes \imath = x$ (also symbolized as $\imath_{\otimes}$),
- (*inverse*) for any $x \in \mathbb{M}$, there exists $y \in \mathbb{M}$ such that $x \otimes y = y \otimes x = \imath$.

For example, $(\mathbb{R}\backslash\{0\}, \times)$ is also a group as each element has an inverse e.g. $1/a \in \mathbb{R}\backslash\{0\}$ holds for all $a \in \mathbb{R}\backslash\{0\}$[Lang, 2012].

Definition 4. (Semiring): $(\mathbb{M}, \oplus, \otimes)$ is a *semiring* if $(\mathbb{M}, \oplus)$ is a monoid with $\imath_\oplus$ identity element, which satisfies for any $x, y, z \in \mathbb{M}$:

- (*commutativity*) $x \oplus y = y \oplus x$,

$(\mathbb{M}, \otimes)$ is a monoid with $\imath_\otimes$ identity element and $\otimes$ distributes over $\oplus$:

- (*distributivity left*) $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$,
- (*distributivity right*) $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$,

and then $\imath_\oplus$ is the zero element of $\otimes$, which annihilates over $\otimes$:

- (*right annihilator*) $x \otimes \imath_\oplus = \imath_\oplus$,
- (*left annihilator*) $\imath_\oplus \otimes x = \imath_\oplus$ [Lothaire, 2005].

A semiring can also be identified as $\mathcal{M} = (\mathbb{M}, \oplus, \otimes, \imath_\oplus, \imath_\otimes)$ where $\imath_\oplus$ and $\imath_\otimes$ are the identity elements for $\oplus$ and $\otimes$ respectively. For example, $\mathcal{R} = (\mathbb{R},+,\times,0,1)$ is a semiring as $(\mathbb{R},+)$ is commutative, $\times$ distributes over $+$, and $a \times 0 = 0$ for all $a \in \mathbb{R}$.

In abstract algebra, a problem can be handled in different structures/operators, which is possible with a structure-preserving map between two identical algebraic structures (e.g. monoids, groups), *homomorphism* [Beachy and Blair, 2019].

Definition 5. (Monoid homomorphism): Given two monoids $(\mathbb{M}, \oplus)$ and $(\mathbb{M}', \oplus')$, a function $hom : \mathcal{M} \to \mathcal{M}'$ is called a *monoid homomorphism* from $\mathcal{M} = (\mathbb{M}, \oplus)$ to $\mathcal{M}' = (\mathbb{M}', \oplus')$ if and only if:

$$\begin{aligned} hom\, \imath_\oplus &= \imath_{\oplus'} \\ hom(x \oplus y) &= hom\, x \oplus' hom\, y \end{aligned} \tag{6.1}$$

For example; the log function which takes the logarithmic value of all elements in a list are a monoid homomorphism from $(\mathbb{R}^+, \times)$ to $(\mathbb{R}, +)$:

$$\begin{aligned} \log(1) &= 0 \\ \log(x \times y) &= \log(x) + \log(y) \end{aligned} \tag{6.2}$$

such that $x, y \in \mathbb{R}^+$.

Definition 6. (Semiring homomorphism): Given two semirings $(\mathbb{M}, \oplus, \otimes)$ and $(\mathbb{M}', \oplus', \otimes')$, a function $hom : \mathbb{M} \to \mathbb{M}'$ is called semiring homomorphism from $(\mathbb{M}, \oplus, \otimes)$ to $(\mathbb{M}', \oplus', \otimes')$ if the following conditions are satisfied:

$$\begin{aligned} g(\imath_{\oplus}) &= \imath_{\oplus'} \\ g(x \oplus y) &= g(x) \oplus' g(y) \\ g(\imath_{\otimes}) &= \imath_{\otimes'} \\ g(x \otimes y) &= g(x) \otimes' g(y) \end{aligned} \tag{6.3}$$

for all $x, y \in \mathbb{M}$ [Allen, 1969].

Such abstract algebraic ideas have been explored to improve the effectiveness of established algorithms or to create more efficient novel algorithms [Kaltofen, 1989]. If a question is defined in an algebraic structure e.g. semiring, then it can be manipulated with algebraic features thanks to various similarities and properties of each algebra. In this case, the question relates to how algebraic features can contribute in a computational manner. For instance, we can consider the following toy example; let $a_1, a_2, \ldots, a_N$ and $c \in \mathbb{N}_0$, with which we want to evaluate the following expression:

$$c \times a_1 + c \times a_2 + \ldots + c \times a_N \tag{6.4}$$

we would need to use n multiplication and $(N-1)$ addition operations, which costs

$(2N-1)$ computational steps. However, as $\mathcal{N} = (\mathbb{N}_0, +, \times, 0, 1)$ is a semiring, using distributivity we can reformulate the above as

$$c \times a_1 + c \times a_2 + \ldots + c \times a_n = c \times (a_1 + a_2 + \ldots + a_n) \tag{6.5}$$

so that, using this feature of the algebra (distributivity) we can decreases the computational to N steps. Distributivity is an exchange in the order of operations. In simple terms, we use the addition operator first and then multiplication rather than applying them in the reverse order. This simple abstract algebraic rule and others can be applied to more complex algorithms to save computational time and space.

7 Dynamic programming (DP) and Directed Acyclic Computation Graphs (DAG)

In early 1950s, Richard Bellman introduced a new approach to solving combinatorial optimization problems, known as *dynamic programming* (DP) [Bellman, 1952].

Definition 7. (Dynamic Programming): Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems; which solves problems in a divide-and-conquer logic recursively and uses the overlapping solution to prevent recalculation [Huang, 2008, Bradley et al., 1977].

DP has numerous areas of application from *speech recognition* [Silverman and Morgan, 1990] to *finance* [Elton and Gruber, 1971]. For instance, Algorithm 2 is a useful example to show how DP and its essential memoization technique can be utilized. Traditionally, DP problems have three important characteristics; *stages, states* and *recursive optimization.*

Conventional DP approach structures an optimization problem into multiple stages,

which are solved sequentially one stage at a time. Each one-stage problem in the sequence is solved as an ordinary optimization problem and then its solution is used to define the characteristic of the next one-stage problem [Bradley et al., 1977]. For instance, let us consider the traveling-salesman problem (TSP) once again. Here, each city represents a stage.

The states of the process are associated with each stage of the optimization problem. The information needed to fully assess the implications of the current decision on future actions is reflected by the states [Bradley et al., 1977]. The structure of a DP algorithm is determined by the relation between states [Bradley et al., 1977]. As an example of states in DP, the city of departure in TSP is the initial state of the problem.

The recursive optimization procedure obtains the solution of the overall N-stage problem by first solving a one-stage problem followed by a sequence of remaining stages. Repetition of this process until the overall optimum is found, is the final general characteristic of the DP approach [Bradley et al., 1977].

All dynamic programming problems can be expressed as a special case of *Bellman's recursion* which obtains exact solutions to combinatorial problems by assembling solutions to "self-similar" sub-problems [Bellman, 1957]. In other words, Bellman's recursion provides the most optimal solution at each stage depending upon the solution to already-computed stages. For instance, in TSP, the global optimal result for N^{th} city can be reached by using the optimality result of $(N-1)^{th}$ city. Formally, TSP can be formalized as below;

$$C(n, V) = \min \Big\{ C\Big(m, V - \{m\}\Big) + d[n, m] \Big\}, \quad m \in V \quad \text{and } n \notin V \tag{7.1}$$

where $d[n, m]$ is the distance between cities n and m, V is the cities; $C\Big(m, V - \{m\}\Big)$ is

the cost of the path that starts from city m [Bellman, 1962]. DP creates 2^N subproblems for each city where N is the total number of city, which means $N \times 2^N$ subproblems to reach a solution. DP for TSP then requires $O(2^N)$ time complexity; although this is an exponential, it is still far better than exhaustive solution with $O(N!)$ complexity.

One major drawback of dynamic programming is that the level of systematicity is low as most algorithms are derived in an ad-hoc, problem-specific manner. Huang [2008] proposes a framework which describes DP problems, which can be solved *Viterbi algorithms* or *Dijkstra's algorithms;* with the sub-problem structure using a *graph* formalism over semirings. Here, Viterbi algorithms are topological order algorithm (e.g. finding the shortest path in the US from East Coast to West Coast) and Dijkstra's algorithms are the graph algorithms that aim to find shortest path within a graph [Huang, 2008].

Definition 8. (Graph/Directed Acyclic Graph): A graph G (or *network*) is a mathematical structure which is formed by a finite set or *vertices* (*nodes*) $\mathbb{V}$ and *edges* (*links*) $\mathbb{E}$, which is represented as $G = (\mathbb{V}, \mathbb{E})$. Vertices are connected via edges. The graph is called *directed* when there is a specific directionality to the edge [Diestel, 2000]. A common form of directed graph is the *directed acyclic graph* (DAG) which has no cycles (or loops), as represented in Figure 4.

The DAG is a conceptual way to represent dependencies of events [Kleinberg and Tardos, 2006]. The order of the dependencies is depicted by a graph, and events are represented by vertices, which are linked via edges. For instance, a DAG structure can represent a distribution chain of a delivery company from the main hub to small distribution points.

DAG is named as *weighted directed acyclic graph* when a homomorphism $w : \mathbb{E} \to \mathbb{S}$ assigns each edge to a weight from the semiring $(\mathbb{S}, \oplus, \otimes, \imath_{\oplus}, \imath_{\otimes})$ [Huang, 2008]. A weighted DAG works recursively as each of the nodes indicate the value of the solution at

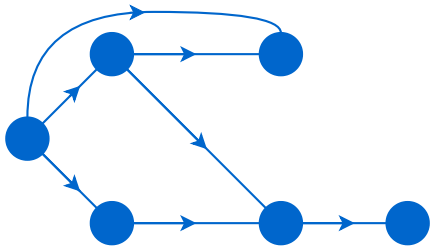

Figure 4: A representation of directed acyclic graph where the nodes (dots) are vertices, and connectors (arrows) are edges. An acyclic graph is directed when no cycles (or loops) are presented.

each stage, and the weighted dependency of each stage on previous stages is represented by the graph edges.

Let us focus on *an unbounded knapsack problem* to illustrate how DAG can be used to represent a brute-force solution first, and then see how DAG can be used to solve a dynamic programming problem. Here, in comparison with a 0-1 knapsack problem, items can be used multiple times in an unbounded knapsack problem [Salkin and De Kluyver, 1975]. For the problem, we have three different items with the values $\{40, 5, 45\}$ and weights $\{3, 1, 4\}$ and weight limit $W = 6$. We can represent the problem in mathematical optimization framework with the object and subject functions as below:

$$\max(40x_1 + 5x_2 + 45x_3), \quad x_1, x_2, x_3 \in \mathbb{N}_0 \tag{7.2}$$

subject to:

$$3x_1 + x_2 + 4x_3 \leq 6 \tag{7.3}$$

and this problem can be transformed into a DAG representation, see Figure 5. Here, the DAG nodes represent states with different bag configurations corresponding to the weight capacity of the bag. For instance, the node assigned with $w = 0$ represents

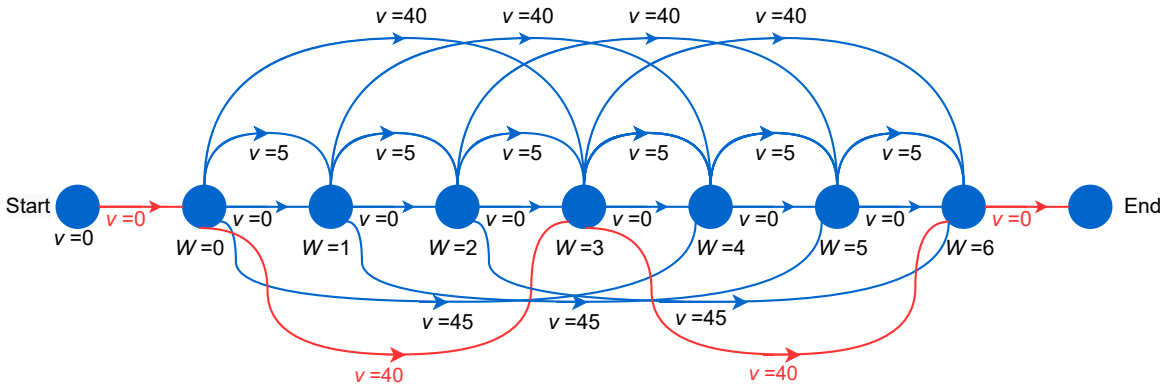

Figure 5: Unbounded knapsack problem as a longest path; weight, value and the weight limit are the fundamental knowledge for the problem. An edge tells the weight of an item with the magnitude, and shows the value of the item as assigned on it. Nodes represent different weight constraints from 0 to 6. Brute force solution of this problem generates every single path (blue paths) from Start to End, and then choose the path with maximum value. The red path from Start to End indicates the optimal solution which requires $x_1 = 2$.

an empty bag while node assigned with $w = 3$ represents the bag with weight 3. The magnitude of a single edge gives the weight of an item, and the value of the item is indexed on the edge. The purpose is to maximize the sum of the edge length. In other words; as Huang's semiring graph DP formalism (which is illustrated above) requires, the knapsack problem becomes a *longest path problem* [Huang, 2008].

Exhaustively, we can fill an empty bag with different combination of these items with a weight limit up to 6. Each path represents the addition of an item to the bag or just skipping the item (horizontal arrows between adjacent nodes). The solution to this problem is obtained via any path from *Start* to *End* node [Zhang et al., 2013]. Start node represents the empty bag while End node gives the bag with the items giving the best result. All the possible configurations of the blue lines (edges) from Start to End is calculated to perform a brute-force solution, and at the end the path with the highest value is chosen. Among the all possible paths, the red path is the one with highest value e.g. $v = 80$ with respect to the weight limit $W = 6$.

Dynamic programming also solves this problem, by finding the longest path to the

node with $W = 1$, and then the longest path to the node $W = 2$, and so on up to the final node. In this recursive algorithm, the DP stage is given by;

$$n \in \mathcal{N} = \{1, 2, \ldots, N\} \tag{7.4}$$

and the DP state

$$m \in \mathcal{M} = \{0, 1, 2, \ldots, W\} \tag{7.5}$$

where $N = 3$ is the number of elements, $W = 6$ is the weight limit for this problem specifically. The recursion for the unbounded knapsack problem is stated below;

$$T(m) = \max(T(m), T[m - w[n]] + v[n]) \quad \text{when } w[n] < m \tag{7.6}$$

where T is the total value, $v[n]$ is the value of, and $w[n]$ is the weight of item n. To solve this problem, we have to generate a table that starts with the first item, and the recursion checks possible solutions that the first item is used as the first row indicates in Table 2. Afterwards, the same procedure is held for the first and second items; and then it continues until all the items are considered. Therefore, the DP solution for the unbounded knapsack problem requires $O((W + 1) \times N)$ computational complexity.

			m						
Index (n)	Value	Weight	0	1	2	3	4	5	6
1	40	3	0	0	0	40	40	40	80
2	5	1	0	5	10	40	45	50	80
3	45	4	0	10	15	40	45	50	80

Table 2: Dynamic programming solution for the unbounded knapsack problem.

Chapter III

Methods and Techniques

1 Preliminaries

Let us start the chapter with some fundamental definitions and theories that we will use later.

Definition 9. (Type): A *type* is the collection or class of data [Shaffer, 1997].

For instance, *False* and *True* are the values of the *Boolean type* and strings consists of any sequence of *characters* in a particular written language e.g. the string "Hello world". Data types (or types in short) are manipulated by functions, for example, addition on the integer data type differs from addition on the type of real numbers

Definition 10. (Parametric Polymorphism): A function acts on all types *uniformly* (or exactly same) when *parametric polymorphism* holds, such a uniformly-acting function is called a *polymorphic function* [Milewski, 2019].

The identity function is the simplest example of a polymorphic function as it just returns the input element for all types. Another example could be the *list length* function, as it maps a list with elements of any type, into the integer corresponding to the list size [Cardelli and Wegner, 1985].

Definition 11. (Topological Order): In a directed graph $G = (\mathbb{V}, \mathbb{E})$, we say that a total ordering $\prec$ on vertices $\mathbb{V}$ is a topological ordering if for every edge $(u, v) \in \mathbb{E}$, we have $u \prec v$ [Bender et al., 2009].

Namely, in a DAG structure, for every directed edge from vertex u to vertex v, u comes before v in the ordering.

Theorem 12. *Given G is a directed a cyclic graph, then G has a topological ordering.*

Proof. It is true for the base case when $N = 1$.

For the induction step, let G' be a DAG of size $\leq N$, and then we assume G' has a topological ordering.

Given DAG G with $N + 1$ vertices. A DAG must have a vertex v without any parents (otherwise, it contradicts the directed graph being acyclic), then $G\backslash\{1\}$ is a DAG since removing v does not create any cycles. From the induction, $G' = G\backslash\{1\}$ is topologically ordered. To create a topological ordering in G, first, we can put v and then append the topological ordering of $G\backslash\{1\}$. This is valid as v does not have any incoming edges. By induction, the theorem is proven [Kleinberg and Tardos, 2006]. □

2 A Generalized Bellman's Recursion

As discussed in the previous chapter, weighted directed acyclic graphs can represent dynamic programming problems, which also implies Bellman's recursion with regards to the graph edges represent weighted dependency of each stage on previous stages. In Figure 5, we have shown that a knapsack problem can be represented as a longest path problem in an appropriate DAG.

Here, we will give a presentation of this general correspondence between longest path problems in a DAG, and a generalized form of Bellman's recursion. With node labels $\mathbb{V} = \{1, 2, \cdots, N\}$, edge labels $\mathbb{E} = \mathbb{V} \times \mathbb{V}$ and the (set-valued) function $\mathbb{P} : \mathbb{V} \to \{\mathbb{V}\}$ giving the DAG structure, the DP problem can be described using functional equations on the DAG. We can represent the longest Start to End path problem as a Bellman's

recursion, given the edge weight map function $w : \mathbb{E} \to \mathbb{R}$:

$$\begin{aligned} f_1 &= 0 \\ f_v &= \max_{v' \in \mathbb{P}(v)} \Big(f_{v'} + w(v, v') \Big) \quad \forall v \in \mathbb{V} \backslash \{1\} \end{aligned} \tag{2.1}$$

where f_v is the length of the longest path between node v and the end node, which is a sub-problem. (2.1) can be defined over the set $\mathbb{R}$, the operators $+$ and max first satisfy Definition 2 with their identity elements as 0 and $-\infty$; and then the monoid structures $(\mathbb{R}, \max)$ and $(\mathbb{R}, +)$ also meet the conditions in Definition 4, which leads us to form a semiring $\mathcal{R} = (\mathbb{R}, \max, +, -\infty, 0)$. Once we represent (2.1) in a semiring form, it is possible to swap the semiring with any other semiring, $\mathcal{S} = (\mathbb{S}, \oplus, \otimes, \imath_{\oplus}, \imath_{\otimes})$ [Huang, 2008]. In the abstract, (2.1) over semiring $\mathcal{S}$ with edge map $w : \mathbb{E} \to \mathbb{S}$ becomes:

$$\begin{aligned} f_1 &= \imath_{\otimes} \\ f_v &= \bigoplus_{v' \in \mathbb{P}(v)} \Big(f_{v'} \otimes w(v, v') \Big) \quad \forall v \in \mathbb{V} \backslash \{1\} \end{aligned} \tag{2.2}$$

which is denoted by $f_{\mathcal{S},w}$. Let us move on by considering knapsack problem once again to see why "semiring substitution" is correct. One exact solution to knapsack problem is exhaustive brute force as it was mentioned in Figure 5. To obtain the exact solution, we can insert the *generator semiring* $\mathcal{G} = \Big(\big\{ [\mathbb{E}] \big\}, \cup, \circ, \emptyset, \big\{ [] \big\} \Big)$ into (2.2). Here, $f_{\mathcal{G},w'}$ is the DP recursion that generates all possible elements, $\big\{ [\mathbb{E}] \big\}$ is the list of lists which is the list of all the possible element combinations, $\cup$ is the union operator, and $\circ$ is the *cross-join* operator which concatenates lists from different sets. For $\mathbb{E} = \mathbb{N}$, the cross-join operator works as below:

$$\big\{ [1, 2], [3] \big\} \circ \big\{ [4], [5] \big\} = \big\{ [1, 2, 4], [3, 4], [1, 2, 5], [3, 5] \big\} \tag{2.3}$$

As all the possible candidates are examined, the optimality must be held for the knap-

sack problem, which proves that semiring substitution is correct for the special case of the DP solver for the knapsack problem. As a result, f is a polymorphic function that satisfies Definition 10.

In the DAG structure, the general form of solving a DP problem over some other semiring $\mathcal{S} = (\mathbb{S}, \oplus, \otimes, \imath_{\oplus}, \imath_{\otimes})$ with associated edge map $w : \mathbb{E} \to \mathbb{S}$ in a brute-force manner follows the procedure below;

- Mapping each element in each generated path into values of type S.
- Combining the values with $\otimes$.
- Accumulating over paths with $\oplus$.

This exhaustive computation is a function $g : \Big\{[\mathbb{E}]\Big\} \to \mathbb{S}$ which is a semiring homomorphism $\mathcal{G} \to \mathcal{S}$ as Definition 6 with the requirement $g\Big(\Big\{[e]\Big\}\Big) = w(e)$ for all $e \in \mathbb{E}$ is held with $\Big(\Big\{[\mathbb{E}]\Big\}, \cup\Big)$ to $(\mathbb{S}, \oplus)$ and $\Big(\Big\{[\mathbb{E}]\Big\}, \circ\Big)$ to $(\mathbb{S}, \otimes)$ monoid homomorphisms. The most crucial point here is that the semiring structure is preserved with homomorphism.

Theorem 13. (DP Semiring Fusion): *Given the generator semiring* $\mathcal{G} = \Big(\Big\{[\mathbb{E}]\Big\}, \cup, \circ, \emptyset, \Big\{[]\Big\}\Big)$ *with the mapping function* $w'(e) = \Big\{[e]\Big\}$ *for all* $e \in \mathbb{E}$*, and another, arbitrary semiring* $\mathcal{S} = (\mathbb{S}, \oplus, \otimes, \imath_{\oplus}, \imath_{\otimes})$ *together with its edge map* $w : \mathbb{E} \to \mathbb{S}$*, if there exists a homomorphism* $g_{\mathcal{S},w}$ *mapping* $g : \mathcal{G} \to \mathcal{S}$ *which additionally satisfy* $g\Big(\Big\{[e]\Big\}\Big) = w(e)$ *for all* $e \in \mathbb{E}$*, then for a function* f*;*

$$g_{\mathcal{S},w} \cdot f_{\mathcal{G},w'} = f_{\mathcal{S},w} \tag{2.4}$$

Proof. Firstly, we will assume without loss of generality, that such DP DAGs have the property that there is only one "start" vertex which must be given a value in order to evaluate all the rest of the DAG vertices, which we denote by $v = 1$. Even if we have

multiple starts, we can always link these start nodes to an initial start vertex via edges that have the values of the "old" start nodes. We also assume that $\mathbb{P}(v) \in 2^{|\mathbb{V}|}$ that is, the parent sets contain only subsets of $\mathbb{V}$.

From Theorem 12, we can always construct a topological ordering of a DP DAG which satisfies Definition 11 [Kleinberg and Tardos, 2006]. This topological ordering of DP DAG starting at $u_n = 1$ with the traversal $u_n \in \mathbb{V}\backslash\{1\}$ for $n \in 2, 3, \ldots, N$. We want to construct an inductive proof of DP semiring fusion using induction on this vertex ordering. Here, to have notational clearance we use upper index to indicate semirings and edge mappings, $f^{\mathcal{S},w}$. Additionally, w edge mapping is arbitrary, therefore, we can ignore it to be clear, $f^{\mathcal{S}}$. Showing the first part of (2.2) is trivial here. $f_1^{\mathcal{G}} = \Big\{[]\Big\}$ from (2.2), which implies that

$$g^{\mathcal{S}}(f_1^{\mathcal{G}}) = g^{\mathcal{S}}\Big(\Big\{[]\Big\}\Big) \tag{2.5}$$

As $g^{\mathcal{S}}$ is a semiring homomorphism, we can conclude that:

$$g^{\mathcal{S}}\Big(\Big\{[]\Big\}\Big) = \imath_{\otimes} \tag{2.6}$$

which requires $\imath_{\otimes} = f_1^{\mathcal{G}}$; therefore, $g^{\mathcal{S}}(f_1^{\mathcal{G}}) = f_1^{\mathcal{S}}$.

For the induction step, we can assume $g(f_{v'}^{\mathcal{G}}) = f_{v'}^{\mathcal{S}}$ for all $v' \in \mathbb{P}(u_n)$ holds in traversal structure which is the process of visiting each vertex in a graph:

$$
\begin{aligned}
g^{\mathcal{S}}(f^{\mathcal{G}}_{u_n}) &= g^{\mathcal{S}}\Big(\cup_{v' \in \mathbb{P}(u_n)} \Big(f_{v'} \circ \{[w(u_n, v')]\} \Big) \Big) && \text{(Generator recursion)} \\
&= \bigoplus_{v' \in \mathbb{P}(u_n)} g^{\mathcal{S}}\Big(f_{v'} \otimes \{[w(u_n, v')]\} \Big) && \text{(Definition 6)} \\
&= \bigoplus_{v' \in \mathbb{P}(u_n)} g^{\mathcal{S}}(f_{v'}) \otimes g^{\mathcal{S}}(\{[w(u_n, v')]\}) \Big) && \text{(Definition 6)} \\
&= \bigoplus_{v' \in \mathbb{P}(u_n)} f_{\mathcal{S}} \otimes w(u_n, v') && \text{(g}\Big(\Big\{[\text{e}]\Big\}\Big)\text{=w(e) and the induction)} \\
&= f_{\mathcal{S}} && \text{((2.2))}
\end{aligned}
\tag{2.7}
$$

This induction is possible with the topological ordering which guarantees that for every u_n, all its parents $v' \in \mathbb{P}(u_n)$ are arranged before u_n in the ordering. Therefore, induction on the topological ordering is valid and we can conclude that all the rest of the vertex values $u_n \in \mathbb{V} \backslash \{1\}$ are mapped into the semiring $\mathcal{S}$ by the homomorphism. □

The proof of Theorem (13) is informally illustrated in Figure 6. Once Bellman's recursion is fit in the right side of (2.4), we can plug computationally heavy but provable generator semiring in to the left side of (2.4), which gives the proof that our approach is exact. There are a couple of very important results of having Bellman's recursion under semiring fusion theorem structure;

1. As mentioned above, any DP problem over semiring $\mathcal{S}$ can be solved in brute-force manner with DAG structures by using the generator semiring $\mathcal{G}$ first, and then applying the semiring $\mathcal{S}$ to result using the homomorphism g, which proves (exhaustively) that DP algorithm is correct as all paths in the DAG is explored. Namely; for any specific problem, all possible combinatorial configurations are tested, one of which must be the solution to the stated problem.

2. As this exhaustive implementation requires the generation and storage of all pos-

sible DP DAG paths, it is computationally inefficient/intractable. This computational intractability is a consequence of the operators in $\mathcal{G}$ being computationally expensive, for example the cross-join $\circ$ operator which is quadratic in the size of each set, and the length of the edge lists they contain. The required memory also grows quadratically with each invocation.

3. However, a consequence of (2.4); we can swap costly $\mathcal{G}$'s operators with computationally and memory-wise less expensive semiring $\mathcal{S}$'s operators (as small as $O(1)$ time and space) like max and $+$.

(2.4) is a shortcut fusion implementation that combines the computation into a single recursion by avoiding the explicit building of the intermediary DAG paths [Hinze, 2010]. Emoto et al. [2012] apply to semiring polymorphic computations over list structures. Indeed, the structural decomposition of the DP problem gives the efficiency of (2.4). However, the proof of (2.4) justifies the decoupling of the type of the quantities computed from the structure of the computation.

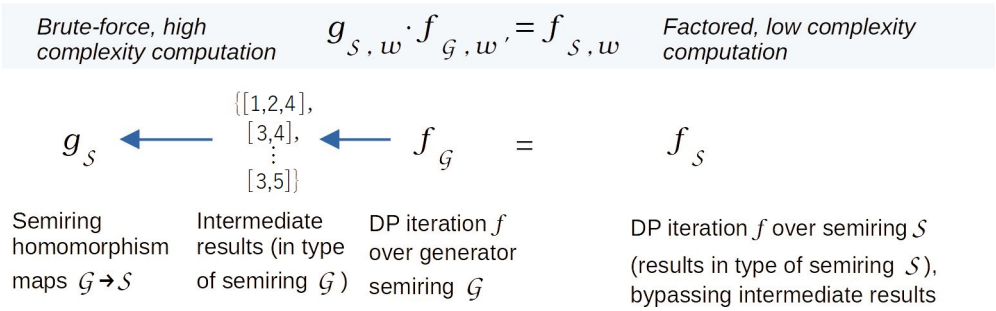

Figure 6: Informal illustration of the dynamic programming (DP) semiring fusion theorem (2.4), the basis on which DP computations in arbitrary semirings is justified.

3 Embedding a Constraint

So far, we have only considered problems which do not require any constraints. It is easy to implement a constraint when we use an exhaustive algorithm (like Emoto

et al. [2012]'s GTA framework). For instance, to solve the 0-1 knapsack problem with a constraint; the only step that we need to add is; after generating all the possible bag configurations, an intermediate operator (a filter) that filters out the bags which do not match with the constraint before moving to the next operator (e.g. aggregation stage). However, applying a constraint to DP problems is not straightforward as we have to rely on custom methods to decompose the problem into sub-problems [Kleinberg and Tardos, 2006]. Once again, exhaustive solutions are not feasible owing to their computational demand, and dynamic programming methods are not systematic meaning they are often quite complex to apply in practice.

As an illustration of how we will systematize DP problems, let us consider the problem of finding the *maximum sum sub-sequence* of a list. An exhaustive solution of the problem starts with generating 2^N possible sub-sequences which requires an exponential time complexity. However, semiring distributivity (max-sum semiring) allows us to solve the same problem in $O(N)$ linear time with the following polymorphic semiring recurrence;

$$\begin{aligned} f_1 &= 0 \\ f_n &= \max\left(f_{n'} + w(n, n')\right) \quad \forall n \leq n' \in \{1, 2, \ldots, N\} \end{aligned} \tag{3.1}$$

where $w : \mathbb{N} \to \mathbb{S}$. Here, (3.1) sums up the elements with a non-negative value. Let us consider the length-constrained maximum sub-sequence sum problem. As discussed above, an exhaustive method (like GTA) is not preferable due to the requirement of expensive computation. However, we can increase the efficiency by using a new semiring that enables us to fuse the constraint with the semiring homomorphism, followed by DP semiring fusion (2.4), in the hopes of eliminating the filtering step and, consequently, the need to generate the intermediate data structures, while taking advantage of the effectiveness of the current recurrence. To fuse the constraint with the generation part,

the function of constraint needs to be in *separable* form.

Here, what we mean by separable form is that the constraint can be expressed in a recursive formula. We will formalize such separable constraint by $\mathcal{M} = (\mathbb{M}, \odot, \imath_{\odot})$ where $\odot$ is a binary operator with an identity $\imath_{\odot}$ (although it is not essential in some applications). The following recurrence $h_{\mathcal{M},v}$ expresses a typical constraint over a list of DAG edges of length L:

$$\begin{aligned} h_0 &= \imath_{\odot} \\ h_l &= h_{l-1} \odot v(e_l) \quad \forall l \in \{1, 2, \ldots, L\} \end{aligned} \tag{3.2}$$

where the *constraint map* $v : \mathbb{E} \to \mathbb{M}$ maps edges into constraint set. To complete the specification of the constraint, we define a Boolean *acceptance condition* $a : \mathbb{M} \to \mathbb{B}$ below, which filters out a list of edges if $a(h_L)$ evaluates false; and the remaining part is kept.

$$a(m) = \begin{cases} T & m = M \\ F & otherwise \end{cases} \tag{3.3}$$

Thus, constraint embedded exhaustive formula (based on 2.4) is represented as:

$$g_{\mathcal{S},w} \cdot \phi_{\mathcal{M},v,a} \cdot f_{\mathcal{G},w'} \tag{3.4}$$

where $\phi : \left\{[\mathbb{S}]\right\} \to \left\{[\mathbb{S}]\right\}$ is the filter function we mentioned. We can write a specific recursive implementation to illustrate as [Bird and De Moor, 1996]:

$$
\begin{aligned}
\phi(\emptyset) &= \emptyset \\
\phi(\{x\}) &= \begin{cases} \{x\} & a\Big(h_{\mathcal{M},v}(x)\Big) = T \\ \emptyset & \text{otherwise} \end{cases} \\
\phi(x \cup y) &= \phi(x) \cup \phi(y)
\end{aligned}
\tag{3.5}
$$

To solve the problem above, we use the semiring which is obtained by *lifting* original semiring over the algebra $\mathcal{M}$ [Emoto et al., 2012, Jeuring, 1993]. Lift operators are a widely used algebraic tool in the literature such that *addition* is lifted *point-wise* to functions, $\Big(f + g\Big)(x) = f(x) + g(x)$ [Hinze, 2010]. In the next subsection, we will cover the technical details of lifting.

3.1 Lifting

A structure-preserving map from one mathematical structure to another of the same type is called a *morphism*. Given morphisms $f : \mathbb{X} \to \mathbb{Y}$ and $g : \mathbb{Z} \to \mathbb{Y}$, a *lifting* of f to Z is a morphism $h : \mathbb{X} \to \mathbb{Z}$ such that $f = g \cdot h$. In category theory, a morphism is an abstraction of homomorphism [Jacobson, 2012]. The following diagram is an illustration of lifting:

$$
\begin{array}{ccc}
 & & O \times A \\
 & {\scriptstyle h \text{ (lifting hom)}} \nearrow & \Big\downarrow {\scriptstyle g \text{ (projection)}} \\
I & \xrightarrow[f \text{ (hom)}]{} & O
\end{array}
\tag{3.6}
$$

Here, (I, O) denotes the input and output of the original homomorphism, and A is the *auxiliary* information that is "lifted" by h lifting operator [Farzan and Nicolet, 2019].

Definition 14. (Lifted Set): $\mathbb{M}$ is a finite and $\mathbb{S}$ is an arbitrary set, a *lifted set* of $\mathbb{S}$ with $\mathbb{M}$ is denoted by $\mathbb{S}[\mathbb{M}]$ such that an element $s \in \mathbb{S}[\mathbb{M}]$ can be seen as an array of size $|\mathbb{M}|$ of which index space is $\mathbb{M}$ [Emoto, 2011].

Emoto et al. [2012] propose an algebraic tool which uses the index space $\mathbb{M}$ to define a generalized product, *lifted semiring* [Emoto, 2011, Emoto et al., 2012].

Definition 15. (Lifted Semiring): Lifting defines a vector of semiring values $f \in \mathbb{S}[\mathbb{M}]$ indexed by $\mathbb{M}$, which can also be seen as a function, $f : \mathbb{M} \to \mathbb{S}$. Let $S[\mathcal{M}] = \Big(\mathbb{S}[\mathbb{M}], \oplus_{\mathcal{M}}, \otimes_{\mathcal{M}}, \imath_{\oplus_{\mathcal{M}}}, \imath_{\otimes_{\mathcal{M}}}\Big)$ be a composite semiring and $(\mathbb{M}, \odot)$ be a finite monoid. The semiring binary operators with parameters x, $y \in \mathbb{S}[\mathbb{M}]$ satisfy the following:

$$\begin{aligned} (x \oplus_{\mathcal{M}} y)_m &= x_m \oplus y_m && (a) \\ (x \otimes_{\mathcal{M}} y)_m &= \bigoplus_{\substack{m', m'' \in \mathbb{M} \\ m' \odot m'' = m}} (x_{m'} \otimes y_{m''}) && (b) \end{aligned} \tag{3.7}$$

is a semiring with associated identities:

$$\begin{aligned} (\imath_{\oplus_{\mathcal{M}}})_m &= \imath_{\oplus} \quad \forall m \in \mathbb{M} \\ (\imath_{\otimes_{\mathcal{M}}})_m &= \begin{cases} \imath_{\otimes} & m = \imath_{\odot} \\ \imath_{\oplus} & \text{otherwise} \end{cases} \end{aligned} \tag{3.8}$$

Let us make this definition more understandable with an example. We start with the lifted space $\mathbb{Z}_3$ which is a vector space with $|\mathbb{Z}_3| = 3$ and $(\mathbb{Z}_3, \max_3, +_3, -\infty_3, 0_3)$ is a semiring with lifted operators $\max_3$ and $+_3$ with the identity elements $-\infty_3 = (-\infty, -\infty, -\infty)$ and $0_3 = (0, 0, 0)$. Here, we have two vector elements $\boldsymbol{p} = (p_0, p_1, p_2)$, $\boldsymbol{q} = (q_0, q_1, q_2) \in \mathbb{Z}_3$. To find a maximum value associated to each component, $\max_3$ utilizes the underlying maximum operator max; as (3.7)(a) requires:

$$\max{}_3\Big(\boldsymbol{p}, \boldsymbol{q}\Big) = \Big(\max(p_0, q_0), \max(p_1, q_1), \max(p_2, q_2)\Big) \tag{3.9}$$

Furthermore, the operator $+_3$ combines associated components, works as in (3.7)(b) :

$$\boldsymbol{p} +^3 \boldsymbol{q} = \Big(\underbrace{(p_0 + q_0)}_{\substack{0,0 \in \mathbb{Z} \\ 0+0=0}}, \underbrace{\max\big((p_0 + q_1), (p_1 + q_0)\big)}_{\substack{0,1 \in \mathbb{Z} \\ 0+1=1 \text{ and } 1+0=1}}, \underbrace{\max\big((p_0 + q_2), (p_2 + q_0), (p_1 + q_1)\big)}_{\substack{0,1,2 \in \mathbb{Z} \\ 0+2=2,\, 2+0=2 \text{ and } 1+1=2}} \Big) \tag{3.10}$$

After having the lifted semiring structure, we also need the lifted edge mapping: $w_{\mathcal{M}} : \mathbb{E}[\mathbb{M}] \to \mathbb{S}$:

$$w_{\mathcal{M}}(x)_m = \begin{cases} w(x) & v(x) = m \\ \imath_{\oplus} & \text{otherwise} \end{cases} \tag{3.11}$$

where $v : \mathbb{E} \to \mathbb{M}$ and $w : \mathbb{E} \to \mathbb{S}$. To obtain the solution to (3.4), we need to lift the monoid $(\mathbb{M}, \odot)$ onto the Boolean acceptance condition $a : \mathbb{M} \to \mathbb{B}$:

$$\pi_{\mathcal{S},a}(x) = \bigoplus_{m' \in \mathbb{M} : a(m') = T} x_{m'} \tag{3.12}$$

which requires a result that we call DP *semiring constrained fusion*:

Theorem 16. (DP semiring constrained fusion): *Given the generator semiring $\mathcal{G}$ with the mapping function w', and another, arbitrary semiring $\mathcal{S}$ with edge mapping function w, the constraint algebra $\mathcal{M} = (\mathbb{M}, \odot, \imath_{\odot})$ with edge mapping function v, acceptance criteria $a : \mathbb{M} \to \mathbb{B}$, constraint filtering function $\phi : (\mathbb{M} \times \mathbb{M} \to \mathbb{M}) \times \mathbb{M} \times (\mathbb{E} \to \mathbb{M}) \times (\mathbb{M} \to \mathbb{B})$ and projection function $\pi : (\mathbb{S} \times \mathbb{S} \to \mathbb{S}) \times \mathbb{S} \times (\mathbb{M} \to \mathbb{B}) \to \mathbb{S}$:*

$$g_{\mathcal{S},w} \cdot \phi_{\mathcal{M},v,a} \cdot f_{\mathcal{G},w'} = \pi_{\mathcal{S},a} \cdot f_{\mathcal{S}[\mathcal{M}],w_{\mathcal{M}}} \tag{3.13}$$

Proof. Being able to fuse the composition of the constraint filtering followed by a semiring homomorphism into a single semiring homomorphism is essential to demonstrating the constrained version of DP semiring fusion. Lifting the constraint filtering over the

set $\mathbb{M}$ will be used to achieve this. Assume the shorthand $g' : \left\{[\mathbb{E}]\right\} \to \mathbb{S}[\mathbb{M}]$ and $\phi'_m = \phi_{\mathcal{M},v,\delta_m}$ where the acceptance function $\delta_m(m') = T$ if $m' = m$ and F otherwise. We write:

$$g'_m(x) = (g_{\mathcal{S},w} \cdot \phi_{\mathcal{M},v,\delta_m})(x) \tag{3.14}$$

As a result, $g'_m(x)$ is the outcome of applying the homomorphism $g_{\mathcal{S},w}$ to the remaining lists after first filtering the set of lists x to keep any lists on which the constraint evaluates to m. Now, g'_m must maintain a semiring structure in order to be a semiring homomorphism. It must also maintain the action of the filtering under $\phi_{\mathcal{M},v,\delta_m}$ in order for it to remain consistent with the filtering.

Turning to the semiring sum, we have:

$$\begin{aligned} g'_m(x \cup y) &= g_{\mathcal{S},w} \cdot \phi'_m(x \cup y) \\ &= g_{\mathcal{S},w} \cdot \left(\phi'_m(x) \cup \phi'_m(y)\right) \\ &= (g_{\mathcal{S},w} \cdot \phi'_m)(x) \oplus (g_{\mathcal{S},w} \cdot \phi'_m)(y) \\ &= g'_m(x) \oplus g'_m(y) \end{aligned} \tag{3.15}$$

To clarify the second step: note that forming the union of sets of lists does not affect on the computation of the constraint value which determines the result of the filtering. Thus, the union of sets of lists is invariant under the action of the filter. The third step follows because $g_{\mathcal{S},w}$ is a semiring homomorphism.

Somewhat more complex is the semiring product, for which we have:

$$\begin{aligned} g'_m(x \circ y) &= g_{\mathcal{S},w} \cdot \phi'_m(x \circ y) \\ &= g_{\mathcal{S},w} \cdot \bigcup_{m',m'' \in \mathbb{M}: m' \odot m'' = m} \left(\phi'_{m'}(x) \circ \phi'_{m''}(y)\right) \\ &= \bigoplus_{m',m'' \in \mathbb{M}: m' \odot m'' = m} g_{\mathcal{S},w} \cdot \left(\phi'_{m'}(x) \circ \phi'_{m''}(y)\right) \\ &= \bigoplus_{m',m'' \in \mathbb{M}: m' \odot m'' = m} (g_{\mathcal{S},w} \cdot \phi'_{m'})(x) \otimes (g_{\mathcal{S},w} \cdot \phi'_{m''})(y) \\ &= \bigoplus_{m',m'' \in \mathbb{M}: m' \odot m'' = m} g'_{m'}(x) \otimes g'_{m''}(y) \end{aligned} \tag{3.16}$$

Clearly, this motivates the definition of the lifted semiring product as

$$(x \otimes_{\mathcal{M}} y_m) = \bigoplus_{m',m'' \in \mathbb{M}: m' \odot m'' = m} x_{m'} \otimes y_{m''} \tag{3.17}$$

A more thorough explanation is necessary for (3.16)'s second phase. In order to define a semiring homomorphism We need to be able to push the filter ϕ'_m inside the cross-join. Recall that the cross-join $x \circ y$ of two sets of lists entails combining each list in x with each list of y. For general lists l', l'' whose constraints evaluate to m' and m'' respectively, then due to the separability of the constraint algebra, the constraint value of their join $l' \circ l''$ is $m' \odot m''$. The cross-join $s' \circ s''$ will consist of sets of lists, all of which have constraints evaluating to $m = m' \odot m''$ if we group together into one set s', all those lists whose constraints evaluate m' and into another set s'', all those lists whose constraints evaluate to m''. In the end, we may determine the values of m', m'' such that $m' \odot m'' = m$ for a given m without knowing further about the characteristics of $\odot$ by exhaustively evaluating all such pairs. Our algebraic simplifications for unique instances like group lifting algebras are based on the fact that this exhaustive search may be reduced if $\odot$ is specialized in some way, particularly with regard to the presence

of inverses.

A semiring homomorphism must map identities. For empty sets which are the identity for $\cup$, we simply require:

$$g'_m(\emptyset) = i_\oplus \quad \forall m \in \mathbb{M} \tag{3.18}$$

Similarly, sets of empty lists act as identities for the cross-join operator. In this case, we must have $g'_m\Big(\Big\{[]\Big\} \circ x\Big) = g'_m\Big(x \circ \Big\{[]\Big\}\Big) = g'_m(x)$. If we set $g'_m\Big(\Big\{[]\Big\}\Big) = \delta_{i_\odot,m}$, then we have:

$$\begin{aligned} g'_m\Big(\Big\{[]\Big\} \circ x\Big) &= \bigoplus_{m',m'' \in \mathbb{M}: m' \odot m'' = m} \delta_{i_\odot,m'} \otimes g'_{m''}(x) \\ &= \bigoplus_{m'' \in \mathbb{M}: i_\odot \odot m'' = m} \delta_{i_\odot,i_\odot} \otimes g'_{m''}(x) \\ &= \bigoplus_{m'' \in \mathbb{M}: i_\odot \odot m'' = m} g'_{m''}(x) \\ &= g'_m(x) \end{aligned} \tag{3.19}$$

and similarly for $g'_m\Big(x \circ \Big\{[]\Big\}\Big)$. This shows the lifted semiring identity to be $i_{\otimes_\mathcal{M}} = \delta_{i_\odot}$.

The homomorphic mapping of sets containing single lists of single edges, such as terms like $\Big\{[e]\Big\}$ is something we must finally take into account. Such terms are only kept under the action of the filter $\phi_{\mathcal{M},v,\delta_m}$ if the constraint edge mapping $v(e) = m$, at which point they add a value $w(e)$ to the semiring value of the homomorphism $g_{\mathcal{S},w}$. They do not otherwise add anything to the semiring sum. Thus, it follows:

$$\begin{aligned} g'_m\Big(\Big\{[e]\Big\}\Big) &= \delta_{v(e),m} \otimes w(e) \\ &= \begin{cases} w(e) & m = v(e) \\ i_\oplus & \text{otherwise} \end{cases} \end{aligned} \tag{3.20}$$

which we write as the lifted edge mapping, $w_\mathcal{M}(e)_m$. To summarize then, (3.15)-(3.20)

show that g'_m is a semiring homomorphism performing the lift mapping $\mathcal{G} \to \mathcal{S}[\mathcal{M}]$:

$$g_{\mathcal{S},w} \cdot \phi_{\mathcal{M},v,\delta_m} = g_{\mathcal{S}[\mathcal{M}],w_{\mathcal{M}}} \tag{3.21}$$

The next step is to reconstruct the result of DP constraint filtering $\phi_{\mathcal{M},v,a}$, from the lifted result. This involves computing the effect of the transformation $a : \mathbb{M} \to \mathbb{B}$ mapping the lifting algebra $\mathbb{M}$ into the value in $\mathbb{B}$ of the predicate a, on an arbitrary lifted semiring object $x \in \mathbb{S}[\mathbb{M}]$. The joint product function π on $\mathbb{M} \times \mathbb{B}$ is written using the Boolean-semiring unit function:

The lifted result is then used to recreate the DP constraint filtering $\phi_{\mathcal{M},v,a}$. This entails calculating the impact of the transformation $a : \mathbb{M} \to \mathbb{B}$ mapping the lifting algebra $\mathbb{M}$ into the value in $\mathbb{B}$ of the predicate a on any lifted semiring object $x \in \mathbb{S}[\mathbb{M}]$. Using the Boolean-semiring unit function, the joint product function π on $\mathbb{M} \times \mathbb{B}$ is represented as follows:

$$\begin{aligned} \pi_{m,b}(x) &= x_m \otimes \delta_{b,a(m)} \\ \delta_{b,b'} &= \begin{cases} i_{\otimes} & b' = b \\ i_{\oplus} & \text{otherwise} \end{cases} \end{aligned} \tag{3.22}$$

We then project onto the second parameter of π to obtain:

$$\begin{aligned} \pi_b(x) &= \bigoplus_{m' \in \mathbb{M}} x_{m'} \otimes \delta_{b,a(m')} \\ &= \bigoplus_{m' \in \mathbb{M}: a(m') = b} x_{m'} \\ &= x_{a^{-1}(b)} \end{aligned} \tag{3.23}$$

where the last line holds if a has a unique inverse. We use the notation $\pi_{\mathcal{S},a}$ as a shorthand for π_T over the semiring $\mathcal{S}$ and the acceptance criteria a.

Putting everything above together, we can show the following:

$$
\begin{aligned}
g_{\mathcal{S},w} \cdot \phi_{\mathcal{M},v,a} \cdot f_{\mathcal{G},w'} &= \pi_{\mathcal{S},a} \cdot g_{\mathcal{S},w} \cdot \phi_{\mathcal{M},v,\delta_m} \cdot f_{\mathcal{G},w'} \\
&= \pi_{\mathcal{S},a} \cdot g_{\mathcal{S}[\mathcal{M}],w_{\mathcal{M}}} \cdot f_{\mathcal{G},w'} && \text{(From (3.21))} \\
&= \pi_{\mathcal{S},a} \cdot f_{\mathcal{S}[\mathcal{M}],w_{\mathcal{M}}} && \text{(From Theorem 13)}
\end{aligned} \tag{3.24}
$$

□

This indicates that we can impose a constraint via lifting to create new "compound" semiring, resulting in a new polymorphic DP recurrence that can be computed over any arbitrary semiring. By repeating the lifting procedure above, we can also lift a lifted semiring meaning that we may create innovative, polymorphic DP recurrences with multiple constraints.

Constructing DP algorithms with constraint seems different than DP algorithms in textbooks. However, they are indeed intimately related, in the following way. In the lifting structure, $\odot$ operator breaks down a DP problem with a constraint into sub-problems with the condition $m' \odot m'' = m$ for each $m \in \mathbb{M}$. By using partitioning, this condition determines how the DP sub-problems should be combined. For instance, let us consider a knapsack problem with a weight constraint $W \leq 3$. To solve this problem, the sub-problems should consider first $W \leq 1$, $W \leq 2$ and $W \leq 3$ respectively. As (3.10) shows, we consider the partitions of different values respectively. More precisely; for $W = 3$, we have to consider all possible weight compositions (partitions) such as $1+1+1$, $2+1$, $1+2$ and 3 in the 0-1 knapsack problem. This partitioning provides a more general description of sub-problem structure than dividing into "smaller", self similar problem approach. Such "constraint-driven" DP decomposition is a key step in our systematic construction of practical DP algorithms. However, this method can be computationally inefficient depending upon the size of $\mathbb{M}$.

3.1.1 Group lifting-simplifying the constraint algebra

In lifting, the computational complexity of the binary operator $\oplus_{\mathcal{M}}$ is $O(1)$, and the operator $\otimes_{\mathcal{M}}$ is $O(M^2)$ for each value of $m \in \mathbb{M}$. To compute the result, we need one iteration over the constraint set per iteration of the original recurrence. Therefore; for the worst-case scenario, the computational complexity that we would have is $O(M^3)$, which results an inefficient algorithm for larger data sets.

The reason behind the inefficiency is that $x \otimes_{\mathcal{M}} y$ has a double summation which makes it quadratic in the size of $\mathbb{M}$. On the other hand; in the DP algorithms, we need to compute terms of the form $a \otimes_{\mathcal{M}} w_{\mathcal{M}}(x)$ for some general for some general $a \in \mathbb{S}[\mathbb{M}]$. We can say $m'' = v(x)$ since (3.11) indicates that $w_{\mathcal{M}}(x)$ takes a different value other than identity element $\imath_{\oplus}$ only for one value, and we can simplify the double summation to a single one:

$$\begin{aligned} a \oplus_{\mathcal{M}} w_{\mathcal{M}}(x) &= \bigoplus_{\substack{m',m'' \in \mathbb{M} \\ m' \odot m'' = m}} (a_{m'} \otimes w_{\mathcal{M}}(x)_{m''}) \\ &= \left(\bigoplus_{\substack{m' \in \mathbb{M} \\ m' \odot v(x) = m}} a_{m'} \right) \otimes w(x) \end{aligned} \tag{3.25}$$

Solutions $m' \in \mathbb{M}$ to the equation $m' \odot v(x) = m$ does not have to be unique because the $\odot$ operator may not have inverses. However, we can reverse this and directly compute $m = m' \odot v(x)$ for each $m' \in \mathbb{M}$, which leads us to an obvious iterative algorithm:

$$\begin{aligned} z &\leftarrow\!\shortmid \imath_{\oplus_{\mathcal{M}}} \\ z_{m \odot v(x)} &\leftarrow\!\shortmid z_{m \odot v(x)} \oplus \Big(a_m \otimes w(x) \Big) \quad \forall m \in \mathbb{M} \end{aligned} \tag{3.26}$$

to obtain $a \otimes_{\mathcal{M}} w_{\mathcal{M}}(x) = z$ at the end of the iteration. As the product in (3.25) is an inherently $O(M)$ operation, and; therefore, by using this simplification, DP recurrence

can perform with $O(M^2)$ for the worst-case scenario.

Another case would be the situation when $\mathcal{M}$ has inverses (for instance, $\mathcal{M}$ is a group which was described in Definition 3), as a natural result of having inverses, there is a unique solution to $m' \odot m'' = m$ which we can write as $m'' = (m')^{-1} \odot m$ on fixing m and m'. This allows us to simplify (3.7)(b) to the $O(M)$ computation:

$$x \otimes_{\mathcal{M}} y = \bigoplus_{m' \in \mathbb{M}} (x_{m'} \otimes y_{(m')^{-1} \odot m}) \tag{3.27}$$

Here, we are only interested in those elements $y_{(m')^{-1} \odot m}$ where their index satisfy $(m')^{-1} \odot m \in \mathbb{M}$. The rest of the elements are truncated as $y_{(m')^{-1} \odot m} = \imath_{\oplus}$ as $\imath_{\oplus}$ is an annihilator. By applying that to (3.25) in such group liftings, we can solve $m' \odot v(x) = m$ uniquely to find $m' = v(x)^{-1} \odot m$, and reach the following simplification:

$$(a \otimes_{\mathcal{M}} w_{\mathcal{M}}(x))_m = \begin{cases} \imath_{\oplus} & m \odot v(x)^{-1} \notin \mathbb{M} \\ a_{m \odot v(x)^{-1}} \otimes w(x) & \text{otherwise} \end{cases} \tag{3.28}$$

which is an $O(1)$ time operation. As a result, DP recurrences produced using group lifting constraints can often be computed with a minimal additional multiplicative time complexity $O(M)$.

Let us consider 0-1 knapsack problem once again to illustrate the lifting method over directed acyclic graphs:

$$\max(35x_1 + 20x_2 + 30x_3 + 8x_4), \quad x_n \in \{0, 1\} \tag{3.29}$$

subject to:

$$3x_1 + 2x_2 + 2x_3 + x_4 \leq 5 \tag{3.30}$$

We have nodes and arrows which are placed in the grid order. Each column represents each item with their values from Start node to End nodes. The main purpose is to build a bridge between these two nodes. Rows represent total weights of items from 0 to the weight limit. As (3.11) maps edges according to the value of assigned elements, all the horizontal transitions gain 0 value.

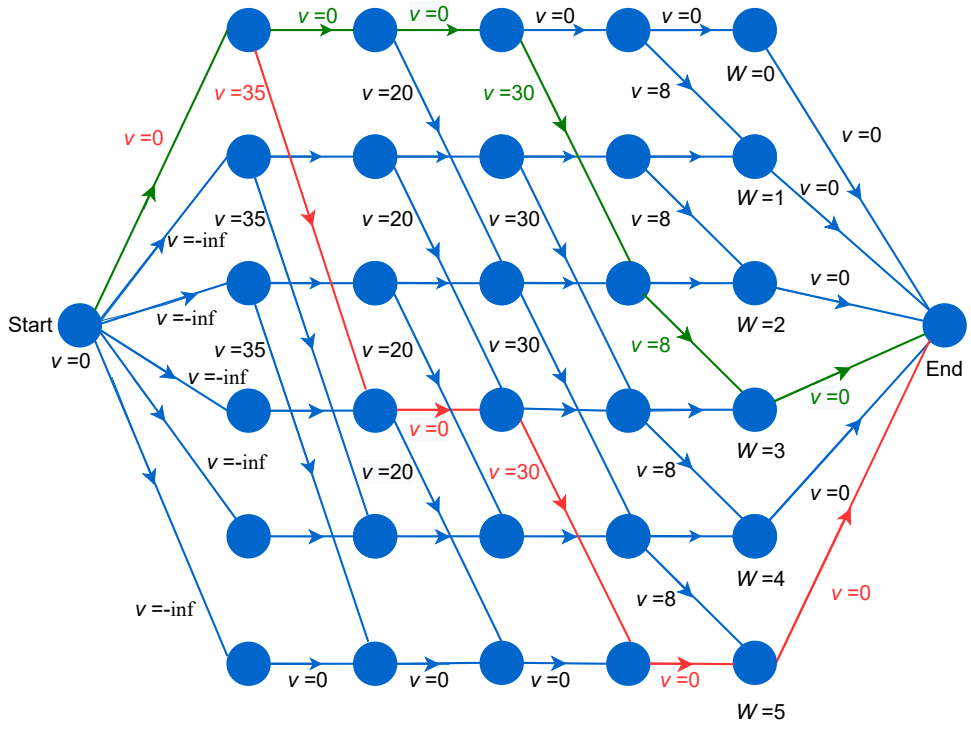

Figure 7: 0-1 knapsack problem with inequality weight constraint; the aim is to find a path from Start node to End nodes to fill a bag with the highest value that has the weight limit up to $W = 5$. An edge tells the weight of an item with the magnitude, and shows the value of the item as assigned on it. The right end nodes represent different weight constraints from 0 to 5. Diagonal move means adding an item to the bag and horizontal move is to pass an item. The path shows the optimal solution for $W = 5$ which requires which requires $x_1 = x_3 = 1$ and $x_2 = x_4 = 0$, and the green paths demonstrates the case $W = 3$.

Under consideration of the weight limit W, the optimal solution is obtained by generating all the possible paths between Start and End nodes, and accordingly evaluating the path with the highest value. As Figure 8 illustrates, the red path indicates the optimal solution $W = 5$ which requires $x_1 = x_3 = 1$ and $x_2 = x_4 = 0$, and the green paths demonstrates the case $W = 3$.

We can also implement this idea to the sum of top K-elements of a list which is a selection algorithm as it is shown below. Finding the sum of top K-elements of a list can also lead us to find the highest K-elements of a list. Here, we apply the same logic as we did for knapsack problem, once we set the weight of each item 1, a list of numbers become a bag of items with the same weight. This problem can be interpreted in combinatorial optimization framework for the example Figure 8 illustrates in as below:

$$\max(8x_1 + 15x_2 - 3x_3 + 27x_4 + 10x_5), \quad x_n \in \{0, 1\} \tag{3.31}$$

subject to:

$$x_1 + x_2 + x_3 + x_4 + x_5 = W \tag{3.32}$$

According to the weight limits, paths are generated between Start and End nodes, and the optimal one is chosen i.e. the red path shows the highest number for $W = 1$. As recursion requires, the latter values with higher weight limits are found by adding up to the previous optimal values as the green path branches from the red path to obtain the optimal value for $W = 2$ and so on.

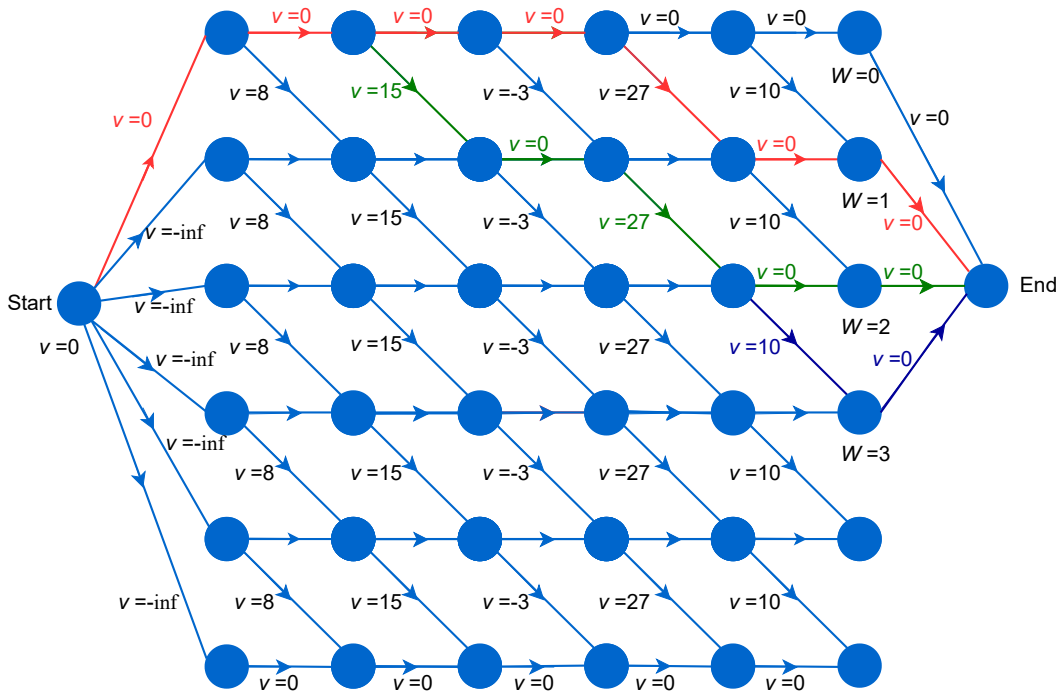

Figure 8: The sum of top K-elements can be determined with DP-DAG formalism. The aim is to find a path from Start node to End nodes to fill a bag with the highest value that has the weight limit up to $W = 5$. An edge tells the weight of an item with the magnitude ($w_n = 1$ for this problem), and shows the value of the item as assigned on it. The right end nodes represent different weight constraints from 0 to 5. Diagonal move means adding an item to the bag and horizontal move is to pass an item. The red path shows how to find the number with the highest value when we set $W = 1$, following that the green path finds the sum of top 2-elements by using the previous optimal solution as recursion requires, and then the dark blue path gives the sum of top 3-elements for $W = 3$.

4 Tupling semirings to avoid backtracking

In most optimization problems, the solution needs to present the value of the decision variables that lead to the most optimal solution. For instance, in Figure 7, we have solved a knapsack problem, and reached an optimal solution 65 when $W = 5$. However, we do not know what decision variables led us to reach this result. The typical solution to this (according to the majority of DP literature) is *backtracking*, which keeps a list of decisions at each step and a series of "back pointers" to the prior choice, and then recovers the unknown decisions by going through the chain of pointers in reverse order

[Ginsberg, 1993].

On the other hand, by utilizing an appropriate semiring, we can prevent the need to implement backtracking, and gain significant flexibility at the same time. Particularly, we will focus on the generator semiring $\mathcal{G}$. Here, we will use a practical trick, *tupling* to apply two different semirings simultaneously. A *tuple* is a data structure which is an ordered collection of elements, and tupling (which is also known as pairing) is a transformation strategy which groups different structures into a tuple [Hu et al., 1997, Pettorossi, 1984, Bird, 1984]. For instance, a tupled semiring is a cartesian product of semirings, and the logic mostly follows applying two different semiring at the same time. We can simultaneously update a semiring total while keeping the values chosen at that stage if we map the semiring values used throughout the DP computations within a pair $\Big(\mathbb{S}, \big\{[\mathbb{S}]\big\}\Big)$. As we already keep the decision variables that lead us to the optimal value, we do not need to use backtracking. For instance, we can use the following *Viterbi* (*arg-max-plus*) semiring [Goodman, 1999, Emoto et al., 2012]:

$$\mathcal{SG} = \Big(\Big(\mathbb{S}, \big\{[\mathbb{S}]\big\}\Big), \oplus, \otimes, \Big(-\infty, \emptyset\Big), \Big(0, \big\{[]\big\}\Big)\Big) \tag{4.1}$$

where

$$\begin{aligned}(a,x)\oplus(b,y) &= \begin{cases}(a,x) & a > b\\ (b,y) & a < b\\ (a, x \cup y) & \text{otherwise}\end{cases}\\ (a,x)\otimes(b,y) &= (a+b, x \circ y)\end{aligned} \tag{4.2}$$

with identities $\imath_{\oplus} = (-\infty, \emptyset)$ and $\imath_{\otimes} = \Big(0, \big\{[]\big\}\Big)$. Furthermore, as we also keep the scenarios where the function has the same value with multiple items, this leads us to multiple solutions at the end. Therefore, the Viterbi semiring maintains multiple

solutions with the same highest score rather than a single solution with the highest score. However, it will cost additional time and space complexity.

Alternatively, using the simplification below, we can get only the optimal single solution:

$$(a, x) \oplus (b, y) = \begin{cases} (a, x) & a \geq b \\ (b, y) & a < b \end{cases} \tag{4.3}$$

In order to compute the value of the best solution as well as the values needed to compute it, the semiring $\mathcal{SG}$ is clearly the tupling of max-plus with $\mathcal{G}$. Regarding time complexity, backtracking and the simple (Viterbi) tupled semiring require similar computational complexity. Backtracking traverses the DP recurrence in the reverse order at the end, which takes $O(N)$ for N made decisions. On the contrary, tupled semirings have the same time complexity as the DP recursion itself. In terms of systematicity, the backtracking implementations often depend entirely on the specific DP recurrence whereas the tupling method only requires changing the semiring. Therefore, we can conclude that tupled semirings are preferable as it provides the systematicity, flexibility and simplicity for finding optimal DP solutions.

Chapter IV

Applications

1 Segmented Linear Regression

Fitting a line through data can be straightforward as we just need to pay attention to the error between the actual data and the line fit. However, it becomes increasingly complex when the linear model does not provide parsimonious representation for the problem as Figure 9(a) demonstrates [Kleinberg and Tardos, 2006]. Of course there are nonlinear options that use curve fitting techniques to represent data such as *log-linear regression* or *polynomial regression* [Ostertagová, 2012, Nassif et al., 2013]. Nonetheless, our initial intention is to stick with the linearity. Then, the next question would be how many lines are needed to represent data. This question leads us another important element in combinatorial machine learning, which is *segmentation* of data items y_n for $n \in \{1, 2, \ldots, N\}$, into contiguous, *non-overlapping* intervals (i, j) such that $i \leq j$ for $i, j \in \{1, 2, \ldots, N\}$. We can utilize segmentation to break the line (with the help of *break-points*) into small pieces and represent data. The points where these small pieces intersect are named as break-points. In the literature, there is a well known technique *segmented linear regression* (*SLR*) which is used to find the best possible piece-wise linear fit with the least-squares error under a penalty which is directly proportional to the number of linear segments [Kleinberg and Tardos, 2006]. Figure 9(b) shows that the data can be represented by $N - 1$ small lines by crossing each of the data points. However, it is neither logical nor efficient.

On the other hand, Figure 9(c) demonstrates that a reasonable representation of the data using two consecutive lines and connecting them at a proper break-point.

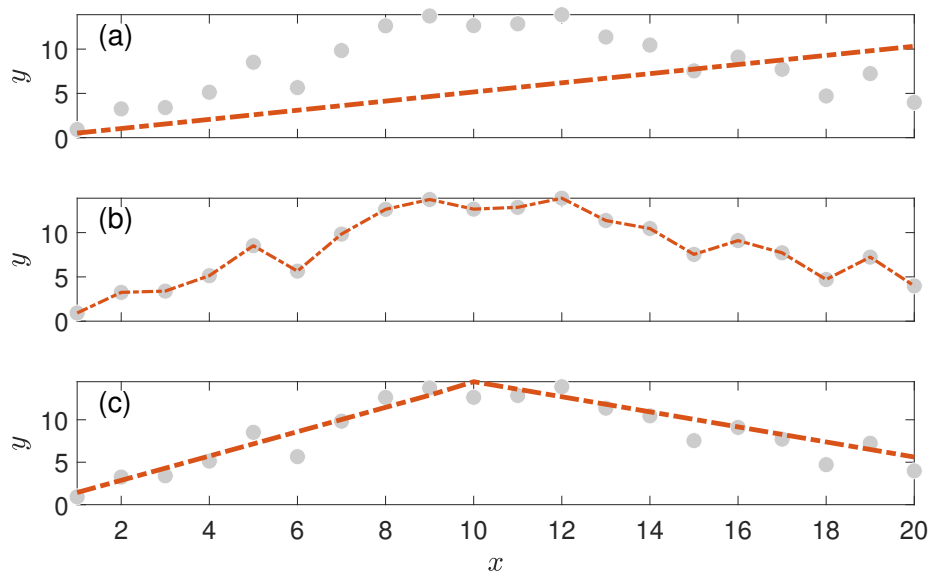

Figure 9: Line fitting through random data in different scenarios. A line under-fits the data in (a), multiple piece-wise linear lines cover the data in (b), and a reasonable piece-wise line fit in (c).

This suggests two crucial aspects which must be considered in order to find an optimal solution; how many linear segments are enough to represent the data, and where the break-points should be located. Such a linear segmented regression problem aims to minimize the estimation error under these two concerns. The general form of segmented regression problem can be represented as minimizing the sum of model fit errors:

$$E(x) = \sum_{j=1}^{N} \sum_{i=1}^{j} x_{i,j} e_{i,j} \tag{1.1}$$

where $e_{i,j} = \frac{1}{p} \sum_{n=i}^{j} \mid y_n - f(n, a_{i,j}) \mid^p$ for $p > 0$ and $x_{i,j} \in \{0, 1\}$ being segment indicators; and $a_{i,j}$ are the optimal model parameters . Before proposing our solution to SLR problems, let us consider one of commonly used solutions to segmented linear problem, ℓ_1 *trend filtering* which proposes an estimation for piece-wise linear time series [Kim et al., 2009].

1.1 ℓ_1 trend filtering

ℓ_1 trend filtering is a non-combinatorial method which is a natural *convex relaxation* to the combinatorial SLR. A *trend* is a pattern/character found in time series data sets in long run [Proietti, 2011]. Let y_n be a scalar time series which satisfies $y_n = x_n + r_n$ for $n = 1, 2, \ldots, N$. The underlying trend of data is represented by x_n and r_n is random component (sometimes called the *residual*) which represents noise. The goal is to estimate the trend component x_n from the measurements y_n, which is also called as *smoothing* or *filtering* [Kim et al., 2009]. The penalty term, and then the trend filtering, varies with what *norm* is performed. Kim et al. [2009] define trend filtering (called ℓ_1 trend filtering) when the *norm* equals to 1 as below:

$$\hat{E} = (1/2)\sum_{n=1}^{N}(y_n - x_n)^2 + \lambda \sum_{n=2}^{N-1} |x_{n-1} - 2x_n + x_{n+1}| \tag{1.2}$$

where the first term of (1.2) is the objective function measures the size of the residual, and the second term is the regularizer term, with $\lambda > 0$ is the regularizer parameter balances off the fit term against smoothness. The regularizer term equals to zero when any three consecutive points are on a line. Here, the norm is defined in general form of p-norm of vector $\boldsymbol{x} = (x_1, x_2, ..., x_N) \in \mathbb{R}^{\mathrm{N}}$ as:

$$\|\boldsymbol{x}\|_p = \left(\sum_{n=1}^{N} |x_n|^p\right)^{1/p}, p \in \mathbb{Z}^+ \tag{1.3}$$

ℓ_1 trend filtering is a variant of *H-P filtering* which was introduced by Hodrick and Prescott [1997] that uses 2-norm/Euclidean norm ($p = 2$ in (1.3)) for the regularizer term. Although the difference seems minor, H-P gives filtering smooth trend whereas ℓ_1 trend filtering provides continuous piece-wise linear trend estimations [Yamada and Bao, 2021]. H-P filtering and ℓ_1 trend filtering are methods that estimate the trend component from previous observations [Kim et al., 2009]. The reader can find all the

details of ℓ_1 trend filtering and H-P filtering in Kim et al. [2009].

ℓ_1 trend filtering could provide desirable heuristic results in many ways; like *interior-point optimization*; that are convex and have been widely studied in literature [Nemirovski and Todd, 2008]. However, in this approach, it is not possible to control the number of segment and the location of break-points are not clear. In the next section, we will propose our segmented linear regression framework which fills the gap where ℓ_1 trend filtering is lacking. Later, we will visualize the difference between ℓ_1 trend filtering and our SLR framework in some synthetic and real data.

1.2 Exact segmented linear regression (SLR)

To solve segmented linear regression problem exhaustively, we need to generate all possible segments and then pick the segment with the minimum value with $O(2^N)$ time complexity. For instance, $segment : \mathbb{N} \rightarrow \Big[\Big[(\mathbb{N},\mathbb{N})\Big]\Big]$ is a recursive function that generates all contiguous segmentations of length N. Here, $\Big\{\Big[(\mathbb{N},\mathbb{N})\Big]\Big\}$ represents the data type of lists of lists of pairs of start/end indices. For $N = 3$ elements:

$$segment\, 3 = \Big\{\Big[(1,3)\Big], \Big[(1,1),(2,3)\Big], \Big[(1,2),(3,3)\Big], \Big[(1,1),(2,2),(3,3)\Big]\Big\} \tag{1.4}$$

As Equation (1.4) illustrates $segment\, 3$ generates every single possible $2^{3-1} = 4$ different segmentations such as $\Big[(1,3)\Big]$ is the one-piece segment which starts at the initial point and ends at the end or $\Big[(1,1),(2,2),(3,3)\Big]$ is another segment which fits the data in 3 pieces. We have generated every single possible segments, to find the best fit we can use the function $minsum : \Big\{\Big[(\mathbb{N},\mathbb{N})\Big]\Big\} \rightarrow \Big\{\Big[(\mathbb{N},\mathbb{N})\Big]\Big\}$ which picks the list of segments and the returns the segment with the lowest error value. There are possible results such that we can over-fit or under-fit the data.

To optimize this problem, we need to minimize an objective function to reach the final result under the consideration of two parameters such that an error term which is the sum of linear fit errors from each segment and a penalty term that penalizes the number of segments. In dynamic programming form, it can be represented as:

$$\hat{E} = \min_{x_{i,j} \in \{0,1\}} \Big[E(\boldsymbol{x}) + \lambda C(\boldsymbol{x}) \Big] \tag{1.5}$$

where the regularizer parameter λ is added up when a new segment occurs with $C(\boldsymbol{x}) = \sum_{i,j \in \{1,2,\ldots,N\}} x_{i,j}$.

Bellman has developed an algorithm in dynamic programming, which solves the problem in $O(N^2)$ [Kleinberg and Tardos, 2006]. As DP requires, the optimal segmentation is obtained by combining all the previous "smaller" optimal segmentations:

$$\begin{aligned} f_0 &= 0 \\ f_j &= \min_{i \in \{1,2,\ldots,j\}} [f_{i-1} + e_{i,j}] \quad \forall j \in \{1, 2, \ldots, N\} \end{aligned} \tag{1.6}$$

where so that $\hat{E} = \pi_{\mathcal{S},a}(f) = f_N$. We can implement the polymorphic version of the DP segmentation algorithm as:

$$\begin{aligned} f_0 &= \imath_{\otimes} \\ f_j &= \bigoplus_{i \in \{1,2,\ldots j\}} \Big[f_{i-1} \otimes w(i,j) \Big] \quad \forall j \in \{1, 2, \ldots, N\} \end{aligned} \tag{1.7}$$

This can be fit in (1.5) by setting $w(i,j) = e_{i,j} + \lambda$.

The first benefit of having polymorphic version of Bellman's recursion is that we can prevent from backtracking by implementing the tupled semiring as we have seen in the previous part. For the pre-computation, the least square error e_{ij} is calculated between the actual data and every possible linear fit for all possible segments $x_{i,j}$ such that $i \leq j \in N$. The next step is to determine the segment with the minimum error

$E[j], j \in \{1, 2, \ldots, N\}$ by starting from the first point of the data with the error $E[0] = 0$, and adding up every next point one by one, and finally checking whether existing segmentation is the best or not. Each step may lead to a change in the segmentation modification. This algorithm is shown in Algorithm 3.

Algorithm 3 Procedural pseudo-code implementation of Bellman's recursion to find minimum error segmentation, $O(N^2)$ time and $O(N)$ space complexity.

```
function SLR (e[1...N, 1...N], λ)
    E[0] = 0
    for j = 1...N
        for i = 1...j
            E[j] = min(E[i − 1] + e[i, j] + λ)
    return E[N]
```

To perform Algorithm 3, we first need to compute $N \times N$ squares-error matrix for each $e_{i,j}$ pairs (i, j), which requires $O(N^2)$ time complexity for a recursive approach. Additionally, for each value of $j = 1, 2, \ldots N$ we have to determine the value of $E[j]$; this takes time $O(N)$ for each, which is for a total of $O(N^2)$ [Kleinberg and Tardos, 2006]. We need to fill an N-sized array with tuples to reach the solution, which requires $O(N)$ space complexity.

1.3 Fixed segmented linear regression (F-SLR)

After this point, we can go further with another question is that "Can we control the number of segments directly?" Indeed, ℓ_1 trend filtering can indirectly control the number of change points (break-points) with a regularizer, but this also causes a change in the magnitude of the gradient [Kim et al., 2009]. In the same way, Bellman's algorithm can also not control the number of segments directly the appropriate choice of the single parameter λ can be difficult to obtain, which may cause under and over fitting in different parts of the same signal (for instance, Figure 11(a)). However, we can control the number of segments with more efficient way by directly constraining

the segmentation to solve segmentation error problem with a fixed number $L > 0$ of segments:

$$\begin{aligned} \hat{E}_L &= \min_{x_{i,j} \in \{0,1\}} E(\boldsymbol{x}) \\ \text{subject to } & \sum_{j=1}^{N} \sum_{i=1}^{j} x_{i,j} = L \end{aligned} \tag{1.8}$$

To solve this problem, first we need a constraint which counts the number of segments up to the fixed number of segments L, which implies that we need the lifting algebra $\mathcal{M} = \Big(\{1, 2, \ldots, L\}, +, 0\Big)$ and lifting mapping function $v(i, j) = 1$, with acceptance condition $a(m) = T$ when $m = L$. Afterwards, we need to insert this lifting semiring to (1.7), which gives us:

$$\begin{aligned} f_{0,m} &= (\imath_{\otimes_{\mathcal{M}}})_m \\ f_{j,m} &= \Big(\Big[\oplus_{\mathcal{M}} \Big]_{i \in \{1,2,\ldots j\}} \Big[f_{i-1} \otimes_{\mathcal{M}} w_{\mathcal{M}}(i,j) \Big] \Big)_m \quad \forall j \in \{1, 2, \ldots, N\} \end{aligned} \tag{1.9}$$

As above, the first line indicates that $f_{0,0} = \imath_{\otimes}$ and $f_{0,m} = \imath_{\oplus}$ when $m \neq 0$, and the second line becomes:

$$\begin{aligned} f_{j,m} &= \Big(\Big[\bigoplus \Big]_{i \in \{1,2,\ldots j\}} \Big[f_{i-1} \otimes_{\mathcal{M}} w_{\mathcal{M}}(i,j) \Big] \Big)_m \\ &= \bigoplus_{i \in \{1,2,\ldots j\}} \begin{cases} \imath_{\oplus} & m - 1 \notin \mathbb{M} \\ f_{i-1,m-1} \otimes w(i,j) & \text{otherwise} \end{cases} \quad \forall j \in \{1, 2, \ldots, N\} \\ &= \begin{cases} \imath_{\oplus} & m - 1 \notin \mathbb{M} \\ \bigoplus_{i \in \{1,2,\ldots j\}} f_{i-1,m-1} \otimes w(i,j) & \text{otherwise} \end{cases} \end{aligned} \tag{1.10}$$

using the group lifting simplification (3.28) in the second step. After applying the acceptance condition, we get $\hat{E}_L = \pi_{\mathcal{S},a}(f) = f_{N,L}$ which obtained in $O(N^2 L)$ time with $O(NL)$ space complexity. The pseudo-code of the algorithms is as shown in Algorithm

4:

Algorithm 4 Procedural pseudo-code implementation of the fixed (L) length segmentation with minimum error algorithm, $O(N^2L)$ time and $O(NL)$ space complexity.

function F-SLR $(e[1...N, 1...N], L)$
  $E[0, 0] = 0$
  $E[0, 1...L] = \inf$
  for $m = 1...L$
    $E[j, 0] = \inf$
    for $j = 1...N$
      $E[j, m] = \min(E[i-1, m-1] + e[i, j], i = 1...j)$
  return $E[N, L]$

As calculating $E[j]$ for all $j = 1 \dots N$ requires $O(N^2)$ complexity, and we need to run inside the outer loop m, L times; the total time complexity is then $O(N^2L)$. The same applies to space complexity for each turn, which makes the same complexity $O(NL)$.

We can simplify Algorithm 4 by following the instructions below:

- Swapping nested two loops as the inner $m = 1 \dots L$ loop over the outer loop $j = 1 \dots N$ as $L \leq N$.
- The inner loop computations are recursive in terms of stage m and we only need results from stage $m-1$ and information at $\textit{F-SLR}\,\{0, 1, \dots, j-1\}$ to compute the latest stage $(\textit{F-SLR}\,j)_m$

These steps provide an opportunity to avoid storing intermediate computation and only keep results from the $m-1$ stage, which decrease the space complexity from $O(NL)$ to $O(N)$ in Algorithm 5.

1.4 Minimum length segmented linear regression (ML-SLR)

Another method to control segmentation could be constraining the size of each segments. We can control the minimum size of each segment with the *length*, $\#(i, j) =$

Algorithm 5 Procedural pseudo-code implementation of the fixed (L) length segmentation with minimum error algorithm, $O(N^2L)$ time and $O(N)$ space complexity.

function F-SLR $(e[1...N, 1...N], L)$
  $E[0] = 0$
  $E[1...L] = \inf$
  for $m = 1...L$
    for $j = N...1$
      $E[j] = \min(E[i-1] + e[i,j], i = 1...j)$
    $E[0] = \inf$
  return $E[L]$

$j - i + 1$ such that:

$$\begin{aligned} \hat{E}_{\min \#=L} &= \min_{x_{i,j} \in \{0,1\}} E(\boldsymbol{x}) \\ \text{subject to } & \min \#(x) = L \end{aligned} \tag{1.11}$$

where $\#(x) = \left\{ \#(i,j) : (i,j) \in \{1,2,\ldots,N\}^2, x_{i,j} = 1 \right\}$ is the set of lengths of all selected segments. We will call this problem as *minimum length segmented linear regression* (*ML-SLR*).

Same as above, we use the lifting algebra $\mathcal{M} = \left(\{1,2,\ldots,N\}, \min, N\right)$ and lifting mapping function $v(i,j) = j - i + 1$. Changing the algebra here requires that the first line of (1.9) becomes:

$$f_{0,m} = \begin{cases} \imath_{\otimes} & m = N \\ \imath_{\oplus} & \text{otherwise} \end{cases} \tag{1.12}$$

and also (3.25) becomes:

$$\Big(a \oplus_{\mathcal{M}} w_{\mathcal{M}}(i,j)\Big)_m = \Big(\bigoplus_{\substack{m' \in \{1,2,\ldots,N\} \\ \min(m', \#(i,j)) = m}} \Big) a_{m'} \otimes w(i,j) \tag{1.13}$$

Here, we have to consider all the possible scenarios of min as the lifting algebra is a monoid (no analytical inverses is required):

$$\left\{m' : \min\left(m', \#(i,j)\right) = m\right\} = \begin{cases} \{m\} & m < \#(i,j) \\ \{m, m+1, ..., N\} & m = \#(i,j) \\ \emptyset & m > \#(i,j) \end{cases} \tag{1.14}$$

Inserting this into (1.13), we obtain:

$$(a \oplus_{\mathcal{M}} w_{\mathcal{M}}(i,j))_m = \left(\begin{cases} a_m & m < \#(i,j) \\ \bigoplus_{m'=m}^{N} a_{m'} & m = \#(i,j) \\ \imath_{\oplus} & m > \#(i,j) \end{cases} \right) \otimes w(i,j) \tag{1.15}$$

and then the second line of (1.9) can be simplified:

$$\begin{aligned} f_{j,m} &= \bigoplus_{i \in \{1,2,...,j\}} \left[f_{i-1} \otimes_{\mathcal{M}} w_{\mathcal{M}}(i,j) \right]_m \\ &= \bigoplus_{i \in \{1,2,...,j\}} \left(\begin{cases} f_{i-1,m} & m < \#(i,j) \\ \bigoplus_{m'=m}^{N} f_{i-1,m'} & m = \#(i,j) \\ \imath_{\oplus} & m > \#(i,j) \end{cases} \right) \otimes w(i,j) \qquad \forall j \in \{1,2,\ldots,N\} \\ &= \bigoplus_{i \in \{1,2,...,j\}} \begin{cases} f_{i-1,m} \otimes w(i,j) & m < \#(i,j) \\ \bigoplus_{m' = \{m, m+1, ..., N\}} f_{i-1,m'} \otimes w(i,j) & m = \#(i,j) \\ \imath_{\oplus} & m > \#(i,j) \end{cases} \end{aligned} \tag{1.16}$$

Using the acceptance condition $a(m) = T$ when $m = L$, we have an $O(N^3)$ time DP algorithm to find the optimal solution, $\hat{E}_{\min \#=L} = \pi_{\mathcal{S},a}(f) = f_{N,L}$. The pseudo code of the algorithm is given in Algorithm 6. We will check the results and compare the

methods in the next part. Now , we are moving to examine another application of our framework.

Algorithm 6 Procedural pseudo-code implementation of the minimum length (L) length segmentation with minimum error algorithm, $O(N^3)$ time and $O(N^2)$ space complexity.

function ML-SLR $(e[1...N, 1...N], \lambda, L)$
  $E[0, 0...N-1] = \inf$
  $E[1, N] = 0$
  for $j = 1...N$
    for $m = 1...N$
      $E[j, m] = \inf$
      for $i = 1...j$
        if $(m < j - i + 1)$
          $E[j, m] = \min(E[j, m], e[i, j] + \lambda + E[i - 1, m])$
        else if $(m = j - i + 1)$
          $E[j, m] = \min(E[j, m], e[i, j] + \lambda + \min(E[i - 1, l], l = m...N))$
  return $E[N, L]$

Same as above, we need $O(N^2)$ time complexity to calculate $E[j]$ for all $j = 1 \dots N$, we need to calculate this for all $m = 1 \dots N$. Therefore, the worst case scenario could have $O(N^3)$ time complexity for $E[j, m]$. We would need to create a $N \times N$ array to save the result, which would require $O(N^2)$ space complexity.

2 Sequence Alignment

Sequence alignment is a useful method for comparing two (or more) sequences by searching for similar, overlapping patterns between them [Mount, 2004]. This is widely performed for machine learning applications in *bio-informatics* [Katoh and Toh, 2010], *natural language processing* [Nadkarni et al., 2011] and *linguistics* [List, 2012]. In response to the COVID-19 pandemic, sequence alignment was applied to identify potential similarities in RNA/ protein sequences of the novel virus and of those already characterized [Ibrahim et al., 2020]. Relating to this, it has been further used to identify new variants of the COVID-19 virus; aligning two RNA sequences to determine how closely related

a set of viral sample pairs are, and estimating mutations such as *insertion* and *deletion* [Kumar et al., 2020].

The main goal in this problem is to minimize the total cost of transforming one sequence into another by changes to individual characters. Such changes in the sequence, termed *edits*, may involve the insertion of a new character, or deletion of an existing one [Pachter and Sturmfels, 2005]. The sequence alignment problem aims to identify the shortest sequential of edits relating multiple sequences. For instance, let us consider two DNA sequences given in Table 3. DNA molecules are formed by chains of nucleotides such as *adenine* (A), *cytosine* (C), *guanine* (G) and *thymine* (T). Mutations within DNA sequences occur upon insertion or deletion of one or several adjacent nucleotides [Rosen, 2017]. Here, original sequence 1=ATCAGCAAC and sequence 2=ATGCGCTA, would pose the question how many edits are needed to align these two sequences. As each insertion or deletion have different cost in different nucleotides, we have to consider minimizing the total cost to reach the optimal output.

	H	*H*	*I*	*H*	*D*	*H*	*H*	*I*	*D*	*H*	D
sequence 1	A	T	-	C	A	G	C	-	A	A	C
sequence 2	A	T	G	C	-	G	C	T	-	A	-

Table 3: An optimal solution for a sequence alignment problem for two DNA sequences with nucleotides *adenine* (A), *cytosine* (C), *guanine* (G) and *thymine* (T). {*H,I,D*} are abbreviations of homology, insertion and deletion respectively.

Table 3 shows us how the optimal solution could be. In the table, {*H,I,D*} stand for homology, insertion and deletion respectively. One solution for this problem could be possible with brute-force method, which requires to generate every single possible assignment settings:

$$\binom{n}{m} \approx \frac{2^{2n}}{\sqrt{\pi n}} \tag{2.1}$$

for $n > m$ alignments to find the optimal minima, which is not feasible as $n = 100$

requires to check more than 10^{60} possible alignments which requires exponential time complexity [Di Fatta, 2019].

2.1 Needleman-Wunsch algorithm

In line with Bellman's recursion, *Needleman-Wunsch* (NW) *algorithm* provides a dynamic programming solution for sequence alignment by building up the best alignment from the optimal alignments of smaller sub-sequences [Likic, 2008]. The general representation of the algorithm with min-sum semiring as below:

$$\begin{aligned} f_{0,0} &= 0 \\ f_{i,0} &= f_{i-1,0} + w(i,0) \\ f_{0,j} &= f_{0,j-1} + w(0,j) \\ f_{i,j} &= \min\Big(f_{i-1,j-1} + w(i,j), f_{i-1,j} + w(0,j), f_{i,j-1} + w(i,0)\Big) \end{aligned} \tag{2.2}$$

for all $i \in \{1, 2, \ldots, N\}$ and $j \in \{1, 2, \ldots, M\}$ where N and M are the length of the sequences and $w(i,j)$ is the cost of the alignment with ith element of the first sequence and jth element of the second sequence [Pachter and Sturmfels, 2005]. The polymorphic abstraction of NW is as below:

$$\begin{aligned} f_{0,0} &= \imath_{\otimes} \\ f_{i,0} &= f_{i-1,0} \otimes w(i,0) \\ f_{0,j} &= f_{0,j-1} \otimes w(0,j) \\ f_{i,j} &= \Big(f_{i-1,j-1} \otimes w(i,j)\Big) \oplus \Big(f_{i-1,j} \otimes w(0,j)\Big) \oplus \Big(f_{i,j-1} \otimes w(i,0)\Big) \end{aligned} \tag{2.3}$$

with the result obtained at $f_{N,M}$. Therefore, it is possible to insert Viterbi semiring into (2.3) and prevent from backtracking which lacks in the literature. In comparison with brute-force, DP algorithm provides a tractable solution in $O(N \times M)$ computational

time.

2.2 Constrained sequence alignment

The standard NW approach has a practical problem in that it has no limit on how far the sequences can move out of alignment.

$$(a \otimes_{\mathcal{M}} w_{\mathcal{M}}(x))_m = \begin{cases} \imath_{\oplus} & m <\mid i-j \mid \\ a_{m-|i-j|} \otimes w(i,j) & \text{otherwise} \end{cases} \tag{2.4}$$

which we write as $(a \circledast w(i,j))_m$ for convenience. Inserting this into (2.3), we get:

$$\begin{aligned} f_{0,0} &= \begin{cases} \imath_{\otimes} & m = 0 \\ \imath_{\oplus} & \text{otherwise} \end{cases} \\ f_{i,0,m} &= \Big(f_{i-1,0} \circledast w(i,0)\Big)_m \\ f_{0,j,m} &= \Big(f_{0,j-1} \circledast w(0,j)\Big)_m \\ f_{i,j,m} &= \Big(f_{i-1,j-1} \circledast w(i,j)\Big)_m \oplus \Big(f_{i-1,j} \circledast w(0,j)\Big)_m \oplus \Big(f_{i,j-1} \circledast w(i,0)\Big)_m \end{aligned} \tag{2.5}$$

for all $i \in \{1,2,\ldots,N\}$ and $j \in \{1,2,\ldots,M\}$. The length of the alignments, lying between $\max(N,M)$ and $N+M$, should be considered when the constraint is set. The lifted sequence alignment requires $O(N \times M \times L)$ computational time complexity for maximum sum of absolute alignment differences L.

3 Clustering

3.1 DP-means clustering

Kulis and Jordan [2011] propose a new algorithm based on k-means algorithm:

$$\min_{\{\ell_c\}_{c=1}^{k}} \sum_{c=1}^{k} \sum_{\boldsymbol{x}\in\ell_c} ||\boldsymbol{x} - \boldsymbol{\mu}_c||_2^2 + \lambda k \tag{3.1}$$

where $\boldsymbol{\mu}_c = \frac{1}{|\ell_c|} \sum_{\boldsymbol{x}\in\ell_c} \boldsymbol{x}$. Unlike k-means, *DP-means* does not specify the number of clusters k in advance. Instead, DP-means forms a new cluster when the distance from a data point to the nearest cluster exceeds a predefined threshold λ [Jiang et al., 2017]. In other words, in DP-means, number of clusters are controlled by a penalizer term λ while k-means predefines them.

Another significant difference between two methods is that the result depends on chosen centroids in k-means clustering whereas, for DP-means, the order of processed data points effects the output [Kulis and Jordan, 2011]. As a result, the optimality can not be guaranteed either in DP-means algorithm.

3.2 MAP-DP clustering

Both k-means and DP-means are distance based clustering methods, and they have limitations when the actual clusters overlap. Raykov et al. [2016] propose a new clustering method which can also be used when nonlinear separation is needed. MAP-DP algorithm minimizes the following objective function which is derived as the negative log likelihood of a *Dirichlet process* mixture model:

$$\min_{\{\ell_c\}_{c=1}^{k}} \sum_{c=1}^{k} \Big[\sum_{\boldsymbol{x}\in\ell_c} \frac{1}{2(\sigma_c+\hat{\sigma}^2)} \|\boldsymbol{x}-\boldsymbol{\mu}_c\|_2^2 + \frac{D}{2}\ln(\boldsymbol{\sigma}_c+\hat{\boldsymbol{\sigma}}^2) \Big] - k\ln N_0 - \sum_{c=1}^{k}\log\Gamma(N_c) + C \tag{3.2}$$

where $\boldsymbol{\sigma}_c = (\frac{1}{\sigma_0^2} + \frac{1}{\hat{\sigma}^2}N_c^i)^{-1}$such that $i \in 1, 2, \cdots, N$, $\boldsymbol{\mu}_c = \boldsymbol{\sigma}_c(\frac{\mu_0}{\sigma_0^2} + \frac{1}{\hat{\sigma}^2}\sum_{\boldsymbol{x}\in\ell_c}\boldsymbol{x})$ and $C = -\ln\left(\frac{\Gamma(N_0)}{\Gamma(N_0+N)}\right)$. Here N_0 is prior count, D is the dimension of the data, $\hat{\sigma}^2$ is spherical cluster variance, σ_0^2 is prior centroid variance, μ_0 is prior centroid location and Γ is the gamma/factorial function [Raykov et al., 2016].

Unlike k-means and DP-means clustering, MAP-DP is a model based clustering algorithm which benefits from the flexibility of the Dirichlet process mixture model. Although the algorithm is more advantageous than Euclidean distance based clustering algorithms; nevertheless, MAP-DP algorithm does not guarantee finding a globally optimal parameter fit to the model likelihood, and it gives different results in each restart.

3.3 Exact clustering

In over all, all the algorithms that we have mentioned can provide results depending upon how they are initialized. As a consequence of this, the results are not provably correct. Other traditional clustering methods such as distribution based (*DBCLASD* [Xu et al., 1998]), model based (*GMM* [Rasmussen, 1999]) and grid based algorithms (*STING* [Wang et al., 1997]) also cannot provide any solution with guaranteed accuracy [Xu and Tian, 2015].

We have discovered that DP-means clustering in 1-D naturally lays down in Equation (2.2) when data is sorted as the following indicates:

$$
\begin{aligned}
f_0 &= \imath_{\otimes} \\
f_j &= \bigoplus_{i \in \{1,2,\ldots j\}} \Big[f_{i-1} \otimes w(i,j) \Big] \quad \forall j \in \{1,2,\ldots,N\}
\end{aligned}
\tag{3.3}
$$

where $w(i,j) = e_{i,j} + \lambda$ such that where $e_{i,j} = \frac{1}{p} \sum_{n=i}^{j} \left| y_n - f(n, a_{i,j}) \right|^p$ for $p > 0$ and $x_{i,j} \in \{0,1\}$ being cluster indicators; and $a_{i,j}$ are the optimal model parameters and the regularizer parameter λ is added up when a new cluster occurs [Wang and Song, 2011]. For the pre-computation, the least square error e_{ij} is calculated between the actual data and every possible fit for all possible clustering $x_{i,j}$ such that $i \leq j \in N$. The next step is to determine the cluster adjustments with the minimum error $E[j]$, $j \in 1, 2, \ldots, N$ by starting from the first point of the data with the error $E[0] = 0$, and adding up every next point one by one, and finally checking whether existing clustering is the best or not. Each step may lead to a change in the clustering modification. This algorithm is shown in Algorithm 7.

Algorithm 7 Procedural pseudo-code implementation of exact DP-means algorithm with $O(N^2)$ time and $O(N)$ space complexity.

function exact clustering dp-means $(e[1...N, 1...N], \lambda)$
  $E[0] = 0$
  for $j = 1...N$
    $E[j] = \min(E[i-1] + e[i,j] + \lambda, i = 1...j)$
  return $E[N]$

In the special case of 1-D clustering with known component variances, MAP-DP clustering can also be achieved using Bellman recursion as each new clusters add a penalty to the objective value, then naturally (3.3) works for MAP-DP as Algorithm 8 implies.

As Algorithm 7 and Algorithm 8 are an implementation of Bellman's recursion, the computational requirements are the same as what Algorithm 3 requires. Namely, for each value of $j = 1, 2, \ldots N$ we have to determine the value of $E[j]$; this takes time

Algorithm 8 Procedural pseudo-code implementation of exact MAP-DP algorithm with $O(N^2)$ time and $O(N)$ space complexity.

function exact clustering MAP-DP $(e[1...N, 1...N], d[1...N], \alpha)$
    $E[0] = 0$
    for $j = 1...N$
        $E[j] = \min(E[i-1] + e[i,j] - d[j-i+1] - ln(\alpha), i = 1...j)$
    return $E[N]$

$O(N)$ for each, which is for a total of $O(N^2)$ and $O(N)$ space complexity requires to fill an N-sized array with tuples.

Similarly, we can do distance based clustering for known number of clusters with k-means clustering. Another practical advantage in proposed 1-D clustering/segmentation is that we can apply tractable constraints such as the minimum number of items to be associated with a cluster. To solve this problem; as we did in F-SLR, first we need a constraint which counts the number of clusters up to the fixed number of clusters k, which implies that we need the lifting algebra $\mathcal{M} = \Big(\{1, 2, \ldots, k\}, +, 0\Big)$ and lifting mapping function $v(i,j) = 1$, with acceptance condition $a(m) = T$ when $m = k$. using the group lifting simplification (3.28) in the second step. After applying the acceptance condition and the simplification we had in F-SLR, we can then get in $O(N^2k)$ time with $O(N)$ space complexity, which is the same as Algorithm 5 as As calculating $E[j]$ for all $j = 1 \ldots N$ requires $O(N^2)$ complexity, and we need to run inside the outer loop m, k times; the total time complexity is then $O(N^2k)$. The pseudo-code of the algorithms is as follows:

Further more, as we are handling 1-D clustering segmentation problem, we can apply a constraint so that the minimum number of elements of a cluster can be determined. Therefore, we can simply fit a clustering problem in to (1.16) where $\#(i,j)$ is the number of elements in a cluster. As a result, Algorithm 6 can be used directly.

Algorithm 9 Procedural pseudo-code implementation of exact k-means algorithm with $O(N^2k)$ time and $O(N)$ space complexity.

function exact clustering k-means $(e[1...N, 1...N], k)$
    $E[0] = 0$
    $E[1...k] = \inf$
    for $m = 1...k$
        for $j = N...1$
            $E[j] = \min(E[i-1] + e[i, j], i = 1...j)$
        $E[0] = \inf$
    return $E[N]$

Chapter V

Results and Discussion

In this part, we will apply our theory to some synthetic or real data experiments to see how practical/useful our framework is. We are starting with a systematic comparison between ℓ_1 trend filtering and Bellman's segmented linear regression. As we discussed, although these two methods have the same aim to estimate the original data or a piece-wise linear fit that represents the data points, they use different methodologies such that one penalizes the gradient points and the other does for the segments.

In the first experiment, we will compare DP segmentation algorithms with ℓ_1 trend filtering with synthetic data. The original data; piece-wise linear constant signal, is represented with grey lines, and noise added to input data y_n is illustrated with grey dots. The noise we used here is Gaussian noise with a different standard deviation σ. Different levels of noise have been added to the same original data for (a) $\sigma = 15$, (b) $\sigma = 30$ and (c)-(d) $\sigma = 60$ respectively as Figure 10 demonstrates. The red lines represent the segmentation results of our DP algorithms and ℓ_1 trend filtering.

In Figure 10, red lines in (a) give a solution for unconstrained segmented linear regression (SLR) (Bellman's recursion), (b) is the solution for SLR with a constrained number of segments, F-SLR, (c) is when a segment has a minimum number of elements constraint, ML-SLR and (d) is the solution that ℓ_1 trend filtering is provided.

For the second experiment, we used real data (Standard and Poor's 500 (S&P 500)) that Kim et al. [2009] also used. S&P 500 is an indexing method of 500 large companies' stock market exchange in the United States [Blume and Edelen, 2004]. In the data, we have 2000 consecutive daily closing values of the S&P 500 Index after the logarithmic transform from March 25, 1999, to March 9, 2007 [Kim et al., 2009]. The same as the

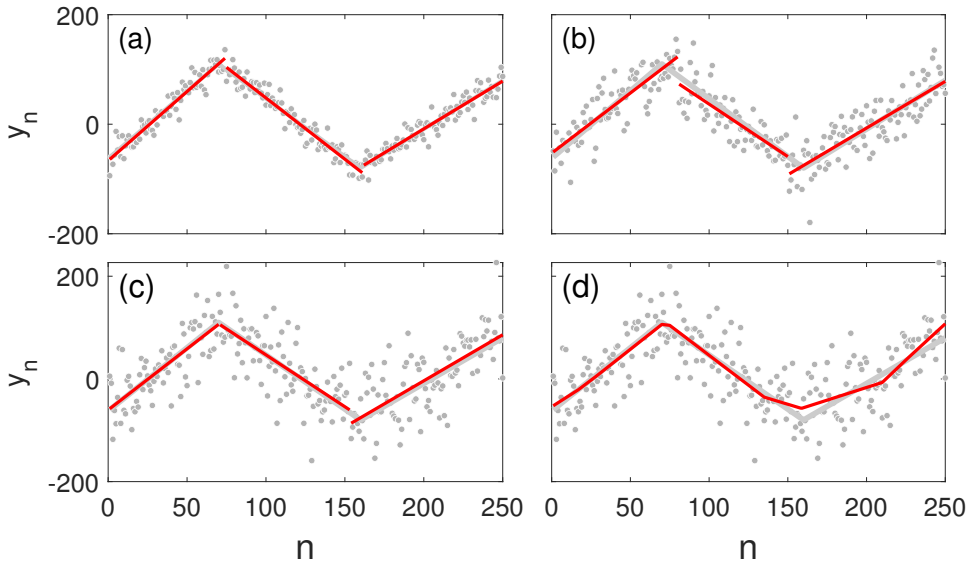

Figure 10: DP segmentation algorithms for a synthetic example with sum-squared error and Gaussian noise (standard deviation σ). The original data; piece-wise linear constant signal (grey line) and noise added input data y_n (grey dots) segmentation result (red line). (a) unconstrained SLR with regularization $\lambda = 15$, noise $\sigma = 15$, (b) F-SLR with $L = 3$ and noise $\sigma = 30$, (c) ML-SLR with $ML = 70$ and noise $\sigma = 60$ and (d) ℓ_1 trend filtering with regularization $\lambda = 10^3$, noise $\sigma = 60$.

previous one, we have got the same experiment set-up in Figure 11; the original data is represented with a grey line, and the segmentation results are illustrated with red lines.

Figure 10(d) and Figure 11(d) show where ℓ_1 trend filtering fails to identify break-points whereas our SLR methods return precise results, which clearly shows the advantage of constrained, exact combinatorial optimization in ML applications for time series analysis. This comparison is one clear line of evidence in support of our approach to DP algorithm derivation.

For the next experiment, we will compare the time complexity of the Needleman-Wunsch algorithm against our constraint sequence alignment algorithm. More precisely, we will check the time complexity of (2.3) and (2.5). To execute the experiment, we have 2 randomly generated sequences with the same length $N = 500$. Afterwards, we

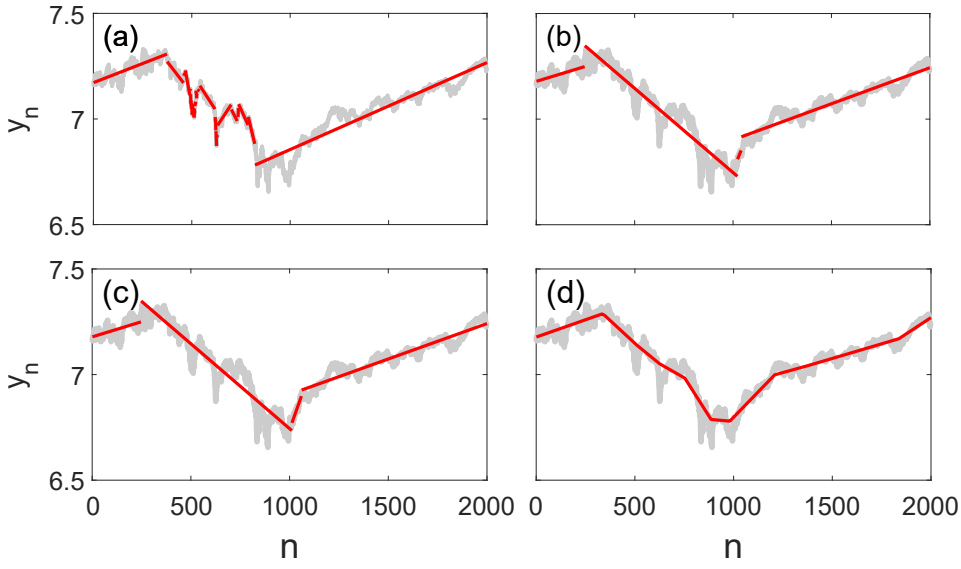

Figure 11: DP segmentation algorithms for a sample of logarithmically-transformed S&P500 financial index daily values [Kim et al., 2009]. Input data y_n (grey lines) and segmentation result (red line). (a) unconstrained SLR with regularization $\lambda = 1.78 \times 10^{-5}$, (b) F-SLR with $L = 4$, (c) ML-SLR with $ML = 50$ and (d) ℓ_1 trend filtering with regularization $\lambda = 100$.

run the Python implementation of algorithms by increasing the length of the sequences by 50 to check how execution time differs. Figure 12 demonstrates the difference between NW $\left(O(N^2)\right)$ and an arbitrary constraint on the maximum absolute difference of misalignments $\left(O(N^3)\right)$.

Our last experiment is to see the importance and accuracy of our exact algorithm in clustering problems. We start our experiment with obvious (separable) data. Figure 13 illustrates that we have a data set which includes 3 clusters with different densities. The crucial point here is that these three clusters have different central points and their variances are small enough to prevent them from overlapping.

We run each approximate method 100 times with different initial starts to replicate the randomness of initialization of approximate methods. For both exact and approximate DP-means and MAP-DP, we have used the same hyper-parameters such that $\lambda = 40$ for each DP-means, and $\hat{\sigma} = 0.1$, $\sigma_0 = 0.5$, $N_0 = 1$ and $\mu_0 =$(mean value of

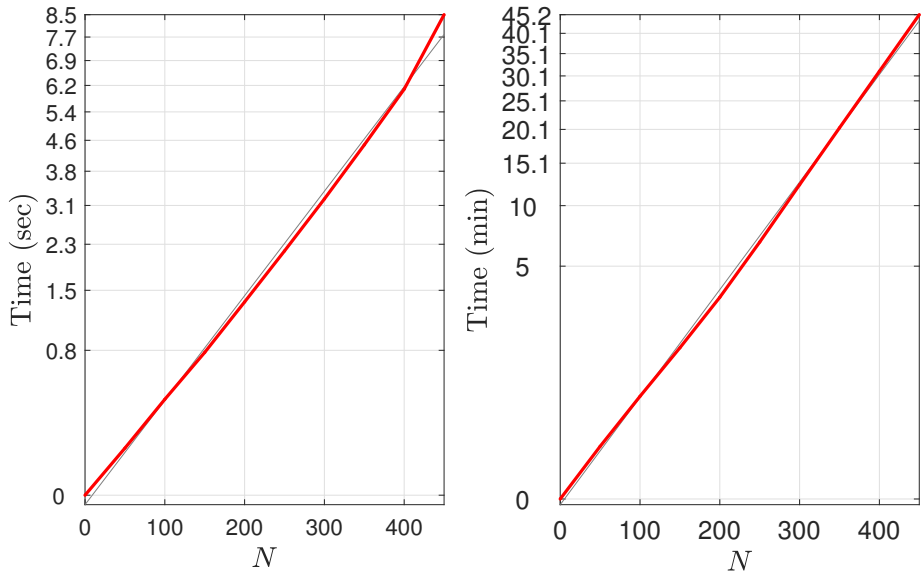

Figure 12: Computational time (red line) required to solve the Needleman-Wunsch DP sequence alignment algorithm (left) without constraints and (right) with lifted constraint. The horizontal axis is the length of both sequences and also the size of the constraint algebra (e.g. $N = M = |\mathcal{M}|$). The vertical axis is on a quadratic (left) and cubic (right) scale such that exact $O(N^2)$ and $O(N^3)$ complexities correspond to a straight line (grey line). Python language implementation on a *Intel Core i7 2.80 GHz, 8 GB RAM.*

data) for MAP-DP. Table 5 indicates the result of our exact algorithm and the minimum and maximum value for approximate methods from the total run. As a result, it can be seen that our exact algorithm and approximate methods perform the same or very close.

On the other hand, the scenario differs when we use a data set that is shown in Figure 14, which has the same number of clusters with more variant density and variance. As it can be seen from the figure, the far left cluster intersects with the middle one, and the far right cluster is completely compassed by the middle one. These overlaps in data also affect the results from approximate algorithms.

Like the previous example, we have used the same hyper-parameters for both methods such that $\lambda = 70$ for each DP-means, and $\hat{\sigma} = 0.5$, $\sigma_0 = 0.25$, $N_0 = 1$ and $\mu_0 =$(mean value of data) for MAP-DP. As Table 5 shows, there is a significant gap between the

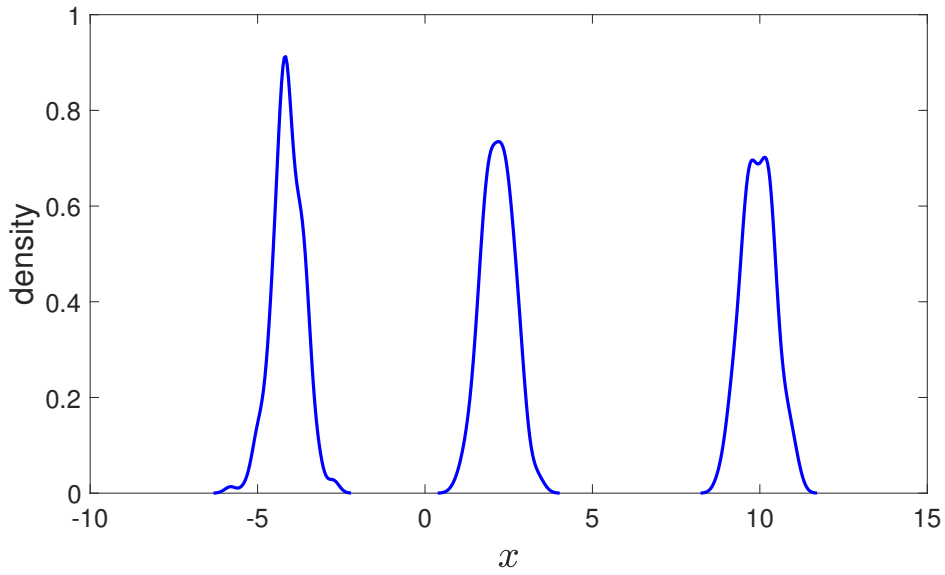

Figure 13: The representation of the synthetic data set for the clustering experiment; Each cluster of data has a small variance so that there is not any overlapping between them. The expected performance from each clustering method is high.

Problem	Exact Alg. Value	Approx. Alg. Min	Approx. Alg. Max
k-means	136	136	136
DP-means	256	256	256
MAP-DP	543	545	545

Table 5: A comparison of clustering methods for the data with separable clusters.

minimum and maximum value in which approximate methods produce, which indicates the importance and reliability of our exact method. For a specific example, k-means specifically, the minimum value from 100 runs matches with what the exact methods produce. However, there is no guarantee that we can get the global optima in which run, or we cannot be sure about how many runs are enough to obtain the optimal value. Therefore, we can conclude that our exact algorithm can be preferable as the exactness in solutions is provided.

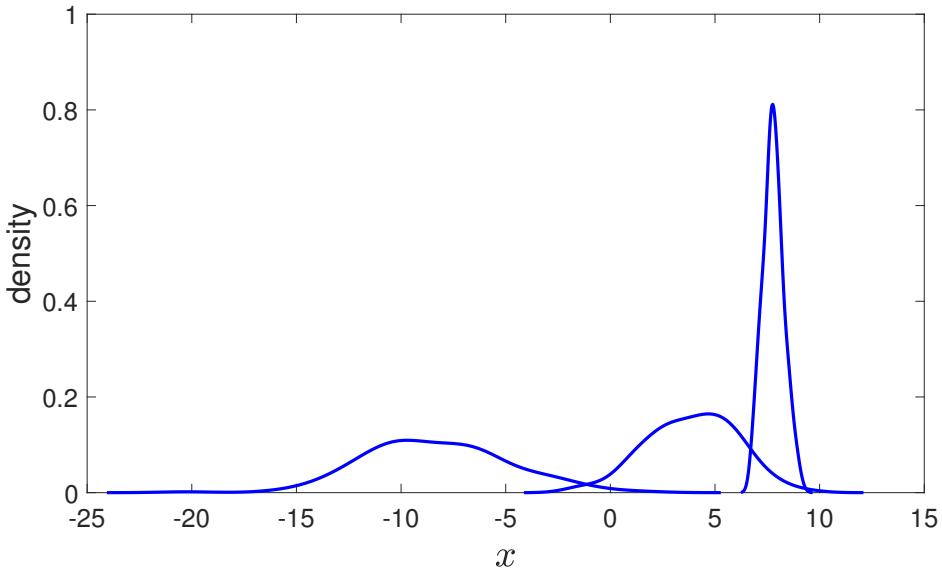

Figure 14: The representation of the synthetic data set for the clustering experiment; the variance of each cluster is high enough to have overlaps, which makes the partitioning difficult. .

Problem	Exact Alg. Value	Approx. Alg. Min	Approx. Alg. Max
k-means	2258	2258	3145
DP-means	919	3494	4578
MAP-DP	1300	1311	1371

Table 7: A comparison of clustering methods for the data with inseparable.

Chapter VI

Conclusion

In conclusion, large-scale combinatorial problems of exponential or factorial complexity that are present in machine learning cannot be optimally solved by exhaustive methods because of their high computational time demand. Heuristic techniques are one approach to solving these issues; however, while they may give a viable compromise solution in a reasonable time, they cannot provide proven optimality. As previously stated, a significant difficulty in current real-world applications is the absence of verifiable optimality.

In the literature, there are attempts to solve combinatorial machine learning problems such as Emoto et al. [2012]'s structure or constructive algorithms approaches [Bird and De Moor, 1996]. As discussed, they are restricted in their presentation as list structures, which are not widely applicable. Combinatorial problems are also common in dynamic programming, but DP algorithms are problem-oriented and rely on ad-hoc solutions; therefore, they cannot be solved systematically.

Indeed, our work is closely related to Emoto et al. [2012]'s semiring filter fusion model, while not explicitly aimed at DP, covers some algorithms which our framework addresses. Emoto et al. [2012]'s work also uses semirings and lifting, but the similarity ends there. Our work applies to a much wider class of DP problems than those which can be expressed using join list homomorphisms. Our contribution also includes an entirely novel approach to using algebraic simplifications to derive much more efficient algorithms, than can be expressed using Emoto et al. [2012]'s framework. According to our understanding, this paper was the first to present algebraic lifting in a constrained form that was only applicable to monoids. Our work describes an entirely novel, modular

approach to constructing new, optimally-efficient combinatorial generators by applying lifting and algebraic simplification to existing ones. Due to these restrictions of Emoto et al. [2012], non-sequential DP algorithms like sequence alignment, edit distance, and dynamic temporal warping appears to be inapplicable, as well as algorithms needing constraints based on more "exotic" lifting algebras, such sorted subsequences, appear to be excluded.

Another approach that we can relate to our work is what Huang [2008] proposes a framework for DP problems under the Viterbi-style topological algorithms and the Dijkstra-style best-first algorithms. Huang [2008] shows that we can formulate some DP problems as semiring computations over graphs. This framework requires *monotonicity condition* in a semiring, which is a partial order with respect to $\leq$. The monotonicity condition divides a problem into sub-problem and finds the optimality of the problem from the optimal sub-problems. Unlike our method, Huang [2008] doesn't rely on recursions and is based on semiring computations over hyper-graphs, although it is pretty general, but does not describe the full generality of all DP algorithms. Huang [2008] only describes the classic algebraic path problem and methods for carrying out the necessary computations and does not offer any new contributions beyond this. Such algebraic path problems come "fully formed" in the sense that, the DAG is fixed, and semiring computations are simply carried out in this fixed DAG. By contrast, our work provides proof of the correctness of this algebraic path problem for the generate-test-aggregate approach to algorithm design. Our work also includes lifting and algebraic simplifications to augment existing DAGs to make them applicable to new problems, and as above, our work describes a new way of constructing efficient generators (which will have a new, lifted DAG structure, as we have shown in Figure 7 and Figure 8).

As a result, in this thesis, we have proposed a rigorous, correct-by-design framework which solves combinatorial machine learning problems exactly and efficiently and does

not require high-level mathematics. Our approach is polymorphic over semirings which provides flexibility to the structure of our algorithm and applies to a specific problem over an existing recursion or simply transform from another algorithm as we have applied segmented linear regression algorithm to solve 1-D clustering problems.

Our approach is simply based on the use of semiring homomorphisms of functional recursions (e.g. Bellman's recursion) represented by paths in the computational DAG constructed from the specification of the DP problem as a semiring-based mathematical 'optimization' problem (it is not always an 'optimization' problem, depends on the particular semirings chosen). We can then use the semiring fusion theorem to gain some computational savings. Specifically, our method is broader than Emoto's framework as it gives a general formula to solve DP problems.

1 Limitations and Future Work

We have implemented our framework for some well-known problems such as segmented linear regression, sequence alignment and clustering in 1-D. However, we were not able to implement our method to feature selection. As the objective value of a feature selection problem depends on the correlation of sub-features, we could not fit this problem into a recursion. It is because correlation is calculated independently from previous calculations. Solving a feature selection problem exactly could require examining 2^N sub-feature sets for N number of features. Therefore, solving intractable feature selection problems efficiently and exactly is future work for us.

Although our formalism is quite broad, it still requires the constraint to be in a separable form. Other future directions may allow the removal of this requirement by using similar algebraic structures. Here, we have shown the constraint can be written in a "local" recursive form in (3.2). This separability form needs to be studied more, it

will stay as a future work.

Broad-scope methods in the literature such as Bird and Moor [1993] and Curtis [1996] can solve greedy algorithms. However, they can only be applied to optimization problems whereas our method works for any problem which can be expressed using a semiring accumulation. Nonetheless, we have not covered mixed-continuous combinatorial problems which can be solved by greedy algorithms and divide-and-conquer, the next motivation could be to extend our method to solve these problems.

Another limitation of our work that we need to address is that it does not make use of some of the more "sophisticated" DP speed-up techniques that have been created for specific scenarios [Galil and Giancarlo, 1989]. Implementation of such an acceleration technique to our method could further enhance the method.

Chapter VII

Appendices

1 Appendix A

A table of some widely used semirings $\mathcal{S} = (\mathbb{S}, \oplus, \otimes, \imath_{\oplus}, \imath_{\otimes})$ and their applications are given below [Golan, 2013, Huang, 2008].

Semiring	Set $\mathbb{S}$	Operators $\{\oplus, \otimes\}$	Identities $\{\imath_{\oplus}, \imath_{\otimes}\}$	Intuition/ Application
Boolean	$\{0, 1\}$	$\{\vee, \wedge\}$	$\{0, 1\}$	Logical deduction
Tropical	$\mathbb{R}^+$	$\{\min, +\}$	$\{+\infty, 0\}$	With non-negative weights
Max Probability	$[0, 1]$	$\{\max, \times\}$	$\{0, 1\}$	Prob. max value
Arithmetic	$\mathbb{N}$	$\{+, \times\}$	$\{0, 1\}$	Solution counting
Real	$\mathbb{R} \cup \{+\infty\}$	$\{\min, +\}$	$\{+\infty, 0\}$	Shortest distance
Softmax	$\mathbb{R}^+$	$\{-\ln(e^{-x} + e^{-y}), +\}$	$\{+\infty, 0\}$	Differentiable minimum negative log likelihood
Relational	$\mathbb{S}R\mathbb{S}$	$\{\cup, \bowtie\}$	$\{\emptyset, 1_R\}$	Database queries
Generator	$\Big\{[\mathbb{E}]\Big\}$	$\{\cup, \circ\}$	$\Big\{\emptyset, \Big\{[]\Big\}\Big\}$	Exhaustive listing
Viterbi	$\mathbb{R} \times \{\mathbb{R}^+\}$	$\Big\{(\min, \arg\min), (+, \cup)\Big\}$	$\Big\{(\infty, \emptyset), (0, \emptyset)\Big\}$	Prob. of the best derivation

Table 8: Some examples of semirings with their applications [Huang, 2008]

2 Appendix B

Some useful lifting algebras:

Example application	Algebra $\mathcal{M} = (\mathbb{M}, \odot, i_{\odot})$	$(a \otimes_{\mathcal{M}} b)_m$ (3.27)	$(a \otimes_{\mathcal{M}} w_{\mathcal{M}}(x))_m$ (3.28)
Subset size	$(\mathbb{N}, +, 0)$	$\bigoplus_{m' \in \mathbb{N}} (a_{m'} \otimes b_{m-m'})$	$\begin{cases} i_{\oplus} & m < v(x) \\ a_{m-v(x)} \otimes w(x) & \text{otherwise} \end{cases}$
Minimum count	$(\{1, \ldots, M\}, \min, M)$	$\bigoplus_{m'=m}^{M} (a_{m'} \otimes b_m) \oplus \bigoplus_{m'=m+1}^{M} (a_m \otimes b_{m'})$	$\begin{cases} a_m \otimes w(x) & m < v(x) \\ \left(\oplus_{m'=m}^{M} a_{m'}\right) \otimes w(x) & m = v(x) \\ i_{\oplus} & m > v(x) \end{cases}$
Maximum count	$(\{1, \ldots, M\}, \max, 0)$	$\bigoplus_{m'=1}^{m-1} (a_{m'} \otimes b_m) \oplus \bigoplus_{m'=1}^{m} (a_m \otimes b_{m'})$	$\begin{cases} a_m \otimes w(x) & m > v(x) \\ \left(\oplus_{m'=1}^{m} a_{m'}\right) \otimes w(x) & m = v(x) \\ i_{\oplus} & m < v(x) \end{cases}$
Existence	$(\mathbb{B}, \vee, F)$	$\begin{cases} a_F \otimes b_F & m = F \\ (a_F \otimes b_T) \oplus (a_T \otimes b_F) \oplus (a_T \otimes b_T) & m = T \end{cases}$	$\begin{cases} a_m \otimes w(x) & v(x) = F \\ (a_F \oplus a_T) \otimes w(x) & (m = T) \wedge (v(x) = T) \\ i_{\oplus} & (m = F) \wedge (v(x) = T) \end{cases}$
For all	$(\mathbb{B}, \wedge, T)$	$\begin{cases} (a_F \otimes b_F) \oplus (a_T \otimes b_F) \oplus (a_F \otimes b_T) & m = F \\ a_T \otimes b_T & m = T \end{cases}$	$\begin{cases} a_m \otimes w(x) & v(x) = T \\ (a_F \oplus a_T) \otimes w(x) & (m = F) \wedge (v(x) = F) \\ i_{\oplus} & (m = T) \wedge (v(x) = F) \end{cases}$
Sequential-value ordering	$((\mathbb{N}, \mathbb{R}), \preceq, z_{\preceq})$	$\bigoplus_{m' \in \mathbb{M}: m' \preceq m} (a_{m'} \otimes b_m)$	$\begin{cases} \left(\oplus_{m' \in \mathbb{M}: m' \preceq m} a_{m'}\right) \otimes w(x) & m = v(x) \\ i_{\oplus} & \text{otherwise} \end{cases}$

Table 9: Some useful example constraints and simplified expressions for the resulting lifted semiring products, see (3.27), along with simplified expressions for the product against the lifted single value, see (3.28).

Chapter VIII

Bibliography

Paul J Allen. A fundamental theorem of homomorphisms for semirings. *Proceedings of the American Mathematical Society*, 21(2):412–416, 1969.

Thomas S Angell and Andreas Kirsch. *Optimization methods in electromagnetic radiation*. Springer Science & Business Media, 2004.

Roland C Backhouse. *An exploration of the Bird-Meertens formalism*. University of Groningen, Department of Mathematics and Computing Science, 1988.

Reham Badawy, Yordan P Raykov, and Max A Little. A unified algorithm framework for quality control of sensor data for behavioural clinimetric testing. *arXiv preprint arXiv:1711.07557*, 2017.

John A Beachy and William D Blair. *Abstract algebra*. Waveland Press, 2019.

Vaishak Belle and Luc De Raedt. Semiring programming: A semantic framework for generalized sum product problems. *International Journal of Approximate Reasoning*, 126:181–201, 2020.

RE Bellman and SE Dreyfus. *Applied dynamic programming*. New Jersey: Princeton Univ. Press, 1962.

Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716–719, 1952.

Richard Bellman. Dynamic programming. *Press Princeton, New Jersey*, 1957.

Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

Michael A Bender, Jeremy T Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 1108–1115. SIAM, 2009.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*, 2018.

Jon Louis Bentley and Michael Ian Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 220–230. ACM, 1976.

Richard Bird and Oege De Moor. The algebra of programming. In *NATO ASI DPD*, pages 167–203, 1996.

Richard Bird and Oege de Moor. From dynamic programming to greedy algorithms. *Formal program development*, pages 43–61, 1993.

Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta informatica*, 21(3):239–250, 1984.

Richard S Bird. An introduction to the theory of lists. In *Logic of programming and calculi of discrete design*, pages 5–42. Springer, 1987.

Richard S Bird. Lectures on constructive functional programming. In *Constructive Methods in Computing Science*, pages 151–217. Springer, 1989.

Richard S Bird and LGLT Meertens. Two exercises found in a book on algorithmics. *Program specification and transformation*, pages 451–458, 1987.

Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

Marshall E Blume and Roger M Edelen. On replicating the s & p 500 index. 30:37–46, 2004.

Christina Bogner, Baltasar Trancón y Widemann, and Holger Lange. Characterising flow patterns in soils by feature extraction and multiple consensus clustering. *Ecological Informatics*, 15:44–52, 2013.

Stephen P Bradley, Arnoldo C Hax, and Thomas L Magnanti. *Applied mathematical programming*. Addison-Wesley, 1977.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

Chi-Lung Cheng and Hans Schneeweiss. Polynomial regression with errors in the variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 60 (1):189–199, 1998.

Y Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. Technical report, Technical report, 2020.

Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

Philip A. Chou. Optimal partitioning for classification and regression trees. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 13(04):340–354, 1991.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2009.

Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

Sharon Curtis. *A relational approach to optimization problems.* PhD thesis, University of Oxford, UK., 1996.

George B Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. 1990.

George B Dantzig and Mukund N Thapa. *Linear programming 1: introduction.* Springer Science & Business Media, 2006.

Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. *Algorithms.* McGraw-Hill Higher Education, 2008.

Oege De Moor. *Categories, relations and dynamic programming.* PhD thesis, University of Oxford, UK., 1991.

Giuseppe Di Fatta. Association rules and frequent patterns. 2019.

Matthew T Dickerson, Robert L Scot Drysdale, Scott A McElfresh, and Emo Welzl. Fast greedy triangulation algorithms. *Computational Geometry*, 8(2):67–86, 1997.

Reinhard Diestel. *Graph theory.* Springer, 2000.

Edsger W. Dijkstraa. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

Said M Easa and Khandaker M Anwar Hossain. New mathematical optimization model for construction site layout. *Journal of construction engineering and management*, 134(8):653–662, 2008.

Jack Edmonds. Maximum matching and a polyhedron with 0, 1 vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

Edwin J Elton and Martin J Gruber. Dynamic programming applications in finance. *The journal of finance*, 26(2):473–506, 1971.

Kento Emoto. An algebraic approach to efficient parallel algorithms for nested reductions. *Technical report METR2011-01, Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo*, 2011.

Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. Filter-embedding semiring fusion for programming with MapReduce. *Formal Aspects of Computing*, 24(4-6):623–645, 2012.

Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 610–624, 2019.

Robert W Floyd. Nondeterministic algorithms. *Journal of the ACM (JACM)*, 14(4): 636–644, 1967.

Keith D Foote. The history of machine learning and its convergent trajectory towards ai. *Machine Learning and the City: Applications in Architecture and Urban Design*, pages 129–142, 2022.

Alan Frieze and Wojciech Szpankowski. Greedy algorithms for the shortest common superstring that are asymptotically optimal. *Algorithmica*, 21(1):21–36, 1998.

Dimitrios Frosyniotis, Andreas Stafylopatis, and Aristidis Likas. A divide-and-conquer method for multi-net classifiers. *Pattern Analysis & Applications*, 6(1):32–40, 2003.

Zvi Galil and Raffaele Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical computer science*, 64(1):107–118, 1989.

Jeremy Gibbons. The school of squiggol. In *International Symposium on Formal Methods*, pages 35–53. Springer, 2019.

Matthew L Ginsberg. Dynamic backtracking. *Journal of artificial intelligence research*, 1:25–46, 1993.

Jonathan S Golan. *Semirings and their Applications*. Springer Science & Business Media, 2013.

Oded Goldreich. Computational complexity: a conceptual perspective. *ACM Sigact News*, 39(3):35–39, 2008.

Joshua Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–606, 1999.

Leonidas J Guibas, John E Hershberger, Joseph SB Mitchell, and Jack Scott Snoeyink. Approximating polygons and subdivisions with minimum-link paths. *International Journal of Computational Geometry & Applications*, 3(04):383–415, 1993.

Mark A Hall and Lloyd A Smith. Practical feature subset selection for machine learning. 1998.

David Harel and Yishai A Feldman. *Algorithmics: The spirit of computing*. Pearson Education, 2004.

Animesh Hazra, Subrata Kumar Mandal, Amit Gupta, Arkomita Mukherjee, and Asmita Mukherjee. Heart disease diagnosis and prediction using machine learning and data mining techniques: a review. *Advances in Computational Sciences and Technology*, 10(7):2137–2159, 2017.

DO Hebb. The organization of behavior; a neuropsychological theory. 1949.

Michael T Heideman, Don H Johnson, and C Sidney Burrus. Gauss and the history of the fast fourier transform. *Archive for history of exact sciences*, pages 265–277, 1985.

Paul Helman and Arnon Rosenthal. A comprehensive model of dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 6(2):319–334, 1985.

Ralf Hinze. Lifting operators and laws. https://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf, 2010. Accessed: 2022-03-16.

Andrew Hodges. *Alan Turing: the enigma*. Princeton University Press, 2014.

Robert J Hodrick and Edward C Prescott. Postwar us business cycles: an empirical investigation. *Journal of Money, credit, and Banking*, pages 1–16, 1997.

Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. *ACM Sigplan Notices*, 32(8):164–175, 1997.

Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–328, 1998.

Liang Huang. Advanced dynamic programming in semiring and hypergraph frameworks. In *Computational Linguistics 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications-Tutorial notes*, pages 1–18, 2008.

Ibrahim M Ibrahim, Doaa H Abdelmalek, Mohammed E Elshahat, and Abdo A Elfiky. Covid-19 spike-host cell receptor grp78 binding site prediction. *Journal of Infection*, 80(5):554–562, 2020.

Michael D Intriligator. *Mathematical optimization and economic theory*. SIAM, 2002.

Aiswarya Iyer, S Jeyalatha, and Ronak Sumbaly. Diagnosis of diabetes using classification mining techniques. *arXiv preprint arXiv:1502.03774*, 2015.

Nathan Jacobson. *Basic algebra I.* Courier Corporation, 2012.

Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

Johan Theodoor Jeuring. *Theories for algorithm calculation.* PhD thesis, Utrecht University, the Netherlands, 1993.

Linhao Jiang, Yichao Dong, Ning Chen, and Ting Chen. Dace: a scalable dp-means algorithm for clustering extremely large sequence data. *Bioinformatics*, 33(6):834–842, 2017.

Erich Kaltofen. Factorization of polynomials given by straight-line programs. *Advances in Computing Research*, 5:375–412, 1989.

Nenwani Kamlesh, Vanita Mane, and Smita Bharne. Adaptive merge sort. *International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868*, pages 21–24, 2014.

Leonid V Kantorovich. Mathematical methods of organizing and planning production. *Management science*, 6(4):366–422, 1960.

Richard M Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.

Kazutaka Katoh and Hiroyuki Toh. Parallelization of the mafft multiple sequence alignment program. *Bioinformatics*, 26(15):1899–1900, 2010.

John D Kececioglu and Eugene W Myers. Combinatorial algorithms for dna sequence assembly. *Algorithmica*, 13(1):7–51, 1995.

Seung-Jean Kim, Kwangmoo Koh, Stephen Boyd, and Dimitry Gorinevsky. l1 trend filtering. *SIAM review*, 51(2):339–360, 2009.

Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

Israel Kleiner. *A history of abstract algebra*. Springer Science & Business Media, 2007.

Tjalling C Koopmans. A note about kantorovich's paper,"mathematical methods of organizing and planning production". *Management Science*, 6(4):363–365, 1960.

Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

Brian Kulis and Michael I Jordan. Revisiting k-means: New algorithms via bayesian nonparametrics. *arXiv preprint arXiv:1111.0352*, 2011.

Swatantra Kumar, Vimal K Maurya, Anil K Prasad, Madan LB Bhatt, and Shailendra K Saxena. Structural, glycosylation and antigenic variation between 2019 novel coronavirus (2019-ncov) and sars coronavirus (sars-cov). *Virusdisease*, 31(1):13–21, 2020.

Serge Lang. *Algebra*, volume 211. Springer Science & Business Media, 2012.

Eugene L Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

Tae Kyun Lee, Joon Hyung Cho, Deuk Sin Kwon, and So Young Sohn. Global stock market investment strategies based on financial network indicators using machine learning techniques. *Expert Systems with Applications*, 117:228–242, 2019.

Zhifei Li and Jason Eisner. First-and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 40–51, 2009.

Maxwell W Libbrecht and William Stafford Noble. Machine learning applications in genetics and genomics. *Nature Reviews Genetics*, 16(6):321–332, 2015.

Vladimir Likic. The needleman-wunsch algorithm for sequence alignment. *Lecture given at the 7th Melbourne Bioinformatics Course, Bi021 Molecular Science and Biotechnology Institute, University of Melbourne*, pages 1–46, 2008.

Johann-Mattis List. Multiple sequence alignment in historical linguistics. In *Proceedings of ConSOLE*, volume 19, pages 241–260, 2012.

Max A Little. *Machine Learning for Signal Processing: Data Science, Algorithms, and Computational Statistics*. Oxford University Press, USA, 2019.

Max A Little and Ugur Kayas. Polymorphic dynamic programming by algebraic shortcut fusion. *arXiv preprint arXiv:2107.01752*, 2021.

Huan Liu and Hiroshi Motoda. *Computational methods of feature selection*. CRC Press, 2007.

Yanhong A Liu and Scott D Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1):37–62, 2003.

Yu Liu, Kento Emoto, Zhenjiang Hu, Y Liu, K Emoto, and Z Hu. A generate-test-aggregate parallel programming library for systematic parallel programming. *Parallel Computing*, 2(40):116–135, 2014.

M Lothaire. *Applied combinatorics on words*, volume 105. Cambridge University Press, 2005.

Chen Luo, Wei Pang, and Zhe Wang. Semi-supervised clustering on heterogeneous information networks. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 548–559. Springer, 2014.

Arthur Mensch and Mathieu Blondel. Differentiable dynamic programming for structured prediction and attention. In *International Conference on Machine Learning*, pages 3462–3471. PMLR, 2018.

Bartosz Milewski. *Category theory for programmers*. Blurb, 2019.

Tom M Mitchell. *Machine learning*. McGraw-hill New York, 1997.

Richard Moorhead, Karen Nokes, and Rebecca Helm. Post office scandal project: Issues arising in the conduct of the bates litigation. *Available at SSRN 3894944*, 2021.

Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Calculus of minimals: Deriving dynamic-programming algorithms based on preservation of monotonicity. Technical report, Technical Report METR, 2007.

Mikhail Moshkov and Beata Zielosko. Combinatorial machine learning. *Studies in Computational Intelligence*, 360, 2011.

David W Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2:19–20, 2004.

Shin-Cheng Mu and José Nuno Oliveira. Programming from galois connections. *The Journal of Logic and Algebraic Programming*, 81(6):680–704, 2012.

Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Machine learning methods for predicting failures in hard drives: A multiple-instance application. *Journal of Machine Learning Research*, 6(May):783–816, 2005.

Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 2011.

Ali Bou Nassif, Danny Ho, and Luiz Fernando Capretz. Towards an early software estimation using log-linear regression and a multilayer perceptron model. *Journal of Systems and Software*, 86(1):144–160, 2013.

Arkadi S Nemirovski and Michael J Todd. Interior-point methods for optimization. *Acta Numerica*, 17(1):191–234, 2008.

Christian Nilsson. Heuristics for the traveling salesman problem. *Linkoping University*, pages 1–6, 2003.

Mahamed GH Omran, Andries P Engelbrecht, and Ayed Salman. An overview of clustering methods. *Intelligent Data Analysis*, 11(6):583–605, 2007.

Eva Ostertagová. Modelling using polynomial regression. *Procedia Engineering*, 48: 500–506, 2012.

Lior Pachter and Bernd Sturmfels. *Algebraic statistics for computational biology*, volume 13. Cambridge university press, 2005.

Pablo Pedregal. *Introduction to optimization*, volume 46. Springer Science & Business Media, 2006.

Alberto Pettorossi. *Methodologies for transformations and memoing in applicative languages*. PhD thesis, The University of Edinburgh, Scotland, 1984.

Jean-Yves Potvin. The traveling salesman problem: A neural network perspective. *ORSA Journal on Computing*, 5(4):328–348, 1993.

Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

Tommaso Proietti. Trend estimation. *In: Lovric M (ed) International Encyclopedia of Statistical Science, 1st Edition*, 2011.

Josep M Pujol, Vijay Erramilli, and Pablo Rodriguez. Divide and conquer: Partitioning online social networks. *arXiv preprint arXiv:0905.4918*, 2009.

Yazan Qarout, Yordan P Raykov, and Max A Little. Probabilistic modelling for unsupervised analysis of human behaviour in smart cities. *Sensors*, 20(3):784–802, 2020.

Carl Rasmussen. The infinite gaussian mixture model. *Advances in neural information processing systems*, 12, 1999.

Yordan Raykov and David Saad. Principled machine learning. *IEEE Journal of Selected Topics in Quantum Electronics*, pages 1–19, 2022. doi: 10.1109/JSTQE.2022.3186798.

Yordan P Raykov, Alexis Boukouvalas, Fahd Baig, and Max A Little. What to do when k-means clustering fails: a simple yet principled alternative algorithm. *PloS one*, 11 (9):e0162259, 2016.

Steven Roman. *An Introduction to the Language of Category Theory*. Springer, 2017.

Kenneth H Rosen. *Handbook of discrete and combinatorial mathematics*. CRC press, 2017.

Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

Franz Rothlauf. *Design of modern heuristics: principles and application*. Springer Science & Business Media, 2011.

Faisal Saeed, Naomie Salim, and Ammar Abdo. Voting-based consensus clustering for combining multiple clusterings of chemical structures. *Journal of cheminformatics*, 4(1):1–8, 2012.

Harvey M Salkin and Cornelis A De Kluyver. The knapsack problem: a survey. *Naval Research Logistics Quarterly*, 22(1):127–144, 1975.

Alexander Schrijver. On the history of combinatorial optimization (till 1960). *Handbooks in operations research and management science*, 12:1–68, 2005.

Clifford A Shaffer. *A practical introduction to data structures and algorithm analysis*. Prentice Hall Upper Saddle River, NJ, 1997.

K Shailaja, B Seetharamulu, and MA Jabbar. Machine learning in healthcare: A review. In *2018 Second international conference on electronics, communication and aerospace technology (ICECA)*, pages 910–914. IEEE, 2018.

Harvey F Silverman and David P Morgan. The application of dynamic programming to connected speech recognition. *IEEE ASSP Magazine*, 7(3):6–25, 1990.

Robert Simson. *The elements of Euclid*. Desilver, Thomas, 1838.

Himanshu Singh. *Practical Machine Learning and Image Processing*. Springer, 2019.

Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

Douglas R Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

Jan A Snyman. *Practical mathematical optimization*. Springer, 2005.

Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

Alan O Sykes. An introduction to regression analysis. *Law School, Univ. of Chicago*, 1993.

George Brinton Thomas and Ross L Finney. *Calculus*. Addison-Wesley Publishing Company, 1961.

Laurens Van Der Maaten, Eric Postma, Jaap Van den Herik, et al. Dimensionality reduction: a comparative. *J Mach Learn Res*, 10(66-71):13, 2009.

Haizhou Wang and Mingzhou Song. Ckmeans. 1d. dp: optimal k-means clustering in one dimension by dynamic programming. *The R journal*, 3(2):29–33, 2011.

Wei Wang, Jiong Yang, Richard Muntz, et al. Sting: A statistical information grid approach to spatial data mining. In *Vldb*, volume 97, pages 186–195, 1997.

Jann Michael Weinand, Kenneth Sörensen, Pablo San Segundo, Max Kleinebrahm, and Russell McKenna. Research trends in combinatorial optimization. *International Transactions in Operational Research*, 29(2):667–705, 2022.

Sanford Weisberg. *Applied linear regression*, volume 528. John Wiley & Sons, 2005.

Ron White and Timothy Edward Downs. *How computers work*. Que, 1998.

Kehinde Williams, Peter Adebayo Idowu, Jeremiah Ademola Balogun, and Adeniran Ishola Oluwaranti. Breast cancer risk prediction using data mining classification techniques. *Transactions on Networks and Communications*, 3(2):1–11, 2015.

Raymond E Wright. Logistic regression. 1995.

Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.

Xiaowei Xu, Martin Ester, H-P Kriegel, and Jörg Sander. A distribution-based clustering algorithm for mining in large spatial databases. In *Proceedings 14th International Conference on Data Engineering*, pages 324–331. IEEE, 1998.

Hiroshi Yamada and Ruoyi Bao. l1 common trend filtering (apr, 10.1007/s10614-021-10114-9, 2021). *COMPUTATIONAL ECONOMICS*, 2021.

Hui-Jun Yang, Young Eun Kim, Ji Young Yun, Han-Joon Kim, and Beom Seok Jeon. Identifying the clusters within nonmotor manifestations in early parkinson's disease by using unsupervised cluster analysis. *PloS one*, 9(3):e91906, 2014.

X Yang. Introduction to mathematical optimization. *From linear programming to metaheuristics*, 2008.

Xiaoge Zhang, Shiyan Huang, Yong Hu, Yajuan Zhang, Sankaran Mahadevan, and Yong Deng. Solving 0-1 knapsack problems based on amoeboid organism algorithm. *Applied Mathematics and Computation*, 219(19):9959–9970, 2013.