Numerical Optimization Methods based on Discrete Structure for Text Summarization and Relational Learning

Masaaki Nishino
Department of Intelligence Science and Technology
Graduate School of Informatics
Kyoto University

Doctoral dissertation, 2014

Contents

List of Tables

List of Figures

Abstract

Numerical optimization problems have an important role in natural language processing (NLP) and many machine learning (ML) tasks, which are accomplished by first making a mathematical model, and then solving certain optimization problems based on the developed model. The difficulty in solving a numerical optimization problem affects the efficiency of a task, and it is therefore important to design a model such that it can formalize the task well, thereby making the optimization problem easy to solve. In this thesis, we propose novel numerical optimization methods for text summarization and relational learning, both of which are typical ML and NLP tasks that can be solved using numerical optimization methods. The key point to our methods is exploiting the discrete structure inherent to the problem domains for decomposing them into small subproblems. This approach has two merits. The first merit is efficiency, i.e., we can use a discrete structure inherent to a problem by decomposing the problem into small subproblems, thereby solving the original problem efficiently. The second merit is flexibility in the design of the optimization problems. We have to be careful when making a new optimization problem for a given NLP or ML task because an easy problem may become a far more difficult problem if we extend it by adding new constraints or objectives. Our technique is flexible from the viewpoint that it can extend existing optimization problems without making the new problem difficult to solve. In the thesis, we treat three different tasks for text summarization and relational learning, and present a concrete optimization method for each of them.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Prof. Akihiro Yamamoto. His helpful advice and our extensive discussions were indispensable to this study. I would also like to state my appreciation to the committee members, Prof. Tatsuya Akutsu and Prof. Sadao Kurohashi, for reviewing this thesis and for their insightful comments.

In addition, I would like to express my gratitude to co-authors, Dr. Norihito Yasuda, Dr. Tsutomu Hirao, Dr. Jun Suzuki, Prof. Shin-ichi Minato, Prof. Toru Kobayashi, Dr. Masaaki Nagata, and Mr. Ryoji Kataoka. The knowledge I have learned from them through our collaborations has greatly helped my research. In particular, I would like to thank Dr. Yasuda for his extensive contribution.

I would also like to give a special thanks to the members of NTT Cyber Solutions laboratories, NTT Communication Science laboratories, the JST ERATO Minato discrete structure manipulation systems project, and Yamamoto-Cuturi laboratory for their useful comments regarding my research.

Finally, I would like to thank my family, including my parents Hatsuhide and Toshie, my parents-in-law Syozo Yokose and Hideko Yokose, my wife Junko, and my daughter Ena, for supporting me in my research. In particular, Junko allowed me to be fully engaged in my research and consistently offered her encouragement. I would like to dedicate this thesis to her.

Chapter 1

Introduction

1.1 Numerical Optimization for Natural Language Processing and Machine Learning

Numerical optimization problems appear in many tasks of natural language processing (NLP) and machine learning (ML), and have important roles in these areas. For example, parameter estimation, a fundamental task of ML, is solved as a numerical optimization problem of finding appropriate parameter values of a mathematical model that maximize the score function given the training data. Another example is translating an input document from one language into another language using a statistical method, which can be solved as a combinatorial optimization problem.

We have to note that such tasks are not numerical optimizations on their own; we accomplish these tasks by first designing a representative mathematical model, and then solving the optimization problems based on this designed model. Suppose that, for example, we want to design a machine translation system. Generating a translation of an input document is not itself solving an optimization problem. We first design a mathematical model for the translation, and using this model, we derive combinatorial optimization problems. By solving these problems, we can obtain a translation as a result.

The difficulty in solving a numerical optimization problem depends heavily on the problem itself. Some problems can be solved exactly in an amount of time linear to the input size, whereas other problems may take an exponential amount of time compared to the size. Hence, designing a model that can describe a task well while simplifying the optimization problem is important for the ML and NLP tasks that we want to accomplish. For example, a parameter estimation of a linear regression model can be modeled as the problem of estimating the parameters that minimize prediction errors using training examples. We have some freedom on how to measure the prediction errors. However, if we use a sum-of-squares error, the score function becomes continuous and convex, and we can solve the minimization problem analytically. Another example is a sequence labeling problem, which is a fundamental NLP problem. A Hidden Markov Model (HMM) [70] or Conditional Random Field (CRF) [50] can be used to solve this problem. Using these models, the task of finding a sequence of labels that maximizes the log-likelihood score, i.e., an optimal solution, can be easily solved using a dynamic programming algorithm. As we show through the above examples, making a model that results in an easy optimization problem is important.

As research in ML and NLP progresses, the methods developed in these fields have started to be applied to a variety of complex problems, such as the modeling of complex networks [34]. Since the optimization problems appearing in these complex problems have tended to become difficult, optimization algorithms that can solve the problems in an efficient manner are in strong demand. Another direction of research in ML and NLP is to extend existing mathematical models by adding new constraints to them. One of the most important extensions is a parameter estimation using ℓ_1 regularization [80, 15], which sets additional penalties defined as the ℓ_1 norm of parameters. This extension can help us obtain sparse parameters. Extending a model by adding new constraints can also make an optimization problem difficult. As a ℓ_1 regularization case, we can optimize a differentiable convex function using its gradients, but if we add a factor that is not differentiable, we cannot optimize the new function using the gradient descent method. We therefore have to take care in designing an extended model.

1.2 Optimization Algorithm based on Discrete-Structure-based Decomposition

In this thesis, we show new approaches for optimization problems that appear in ML and NLP tasks. Our approaches have the following two features:

1. exploiting the discrete structure inherent in the problem domain, and
2. decomposing a complex problem into easy-to-solve subproblems.

We have two objectives for using these features. The first objective is efficiency. Many ML and NLP tasks are founded on a discrete structure, even if the optimization problem related to the task is not a discrete or combinatorial optimization problem. If we can use a discrete structure, we can efficiently solve an optimization problem. For example, the sequential labeling problem mentioned above is founded on the sequence structure, and we can therefore apply dynamic programming. The algorithm exploits the sequential structure of the problem domain. Unfortunately, the discrete structure is simply a component in a complex optimization problem, and we cannot directly exploit the structure. Decomposing a problem into subproblems enables subproblems to be extracted, allowing a discrete structure to be used more efficiently. The second objective is flexibility in designing optimization problems. This is especially useful for designing new optimization problems by extending existing problems. As we previously showed, we have to take care when making a new optimization problem for NLP or ML tasks because an easy problem quickly becomes difficult if we extend it by adding new properties. Our discrete structure-based decomposition method can extend the existing optimization algorithms that exploit a discrete structure. We decompose an extended complex problem into subproblems, some of which can exploit a discrete structure, and others can represent extensions. This method enables us to use a discrete structure efficiently, and the extended problem becomes easy to solve.

Both features, i.e., using a discrete structure and decomposing a problem into subproblems, are by themselves very standard approaches taken for solving problems in computer science. Decomposing a problem into subproblems is a particular standard approach for optimization, and has recently been used in dual decomposition [73, 8, 72] and proximal gradient approaches [68, 17, 18]. In these approaches, a problem is decomposed into simple subproblems, and is solved by solving its subproblems either iteratively or in parallel. However, such decomposition techniques require skill in designing an effective decomposition of a problem because a suitable decomposition depends strongly on the problem. One of our contributions is to combine discrete structures into these decomposing frameworks that have not been used in previous approaches. Using these decomposition-based approaches can broaden the range of solvable problems.

1.3 Text Summarization and Relational Learning

In the following chapters, we treat three concrete tasks and propose new optimization algorithms for each. The first task, treated in Chapter 2 is text summarization, which is an NLP task of generating concise summaries of input documents. The latter two tasks are foundations for relational learning as well as NLP. Relational learning means the class of ML tasks that deals with complex relationships between entities. For example, analysis of the structure of large networks, link prediction, and classification of nodes based on network structure are typical relational learning tasks near to various applications. The tasks we treat in this thesis are to be foundations of these complex tasks. We treat two fundamental tasks useful for relational learning.

In Chapter 2, we treat r summarizing multiple documents. Both in Chapter 3 and 4, we deal with relational learning tasks. The task treated in Chapter 3 is parameter learning for the probabilistic logic programming (PLP) model. PLP is a kind of statistical relational learning model [34] that is used for analyzing relationships between entities. Chapter 4 is for accelerating repeated multiplication of adjacent matrices, which are used in the computation of the Personalized PageRank (PPR) and Non-negative Matrix Factorization (NMF), both of which are used for unsupervised learning in analyzing network structure or relationships between entities.

1.4 The Outline of this Thesis

We briefly introduce these concrete algorithms we will propose in the following chapters.

Text Summarization as a Combinatorial Optimization Problem with Multiple Objectives The first method we describe in Chapter 2 is for text summarization. Text summarization is an NLP task for automatically generating a summary from a given set of documents. Text summarization is very useful and has many practical applications such as generating snippets [57], short fragments of a Web page that are used in most of commercial search engines. There are two main methods for text summarization: extractive methods and abstractive methods. Using extractive methods, a summary is drafted by extracting appropriate textual components from the given documents. In other words, such methods regard text summarization as solving the problem of extracting an appropriate subset of given input documents. The problem is formulated as a combinatorial optimization problem since we use discrete textual components such as sentences, phrases, or words as the elemental textual units that compose a document. We solve this problem with a constraint in which the size (e.g., number of words contained in a subset) of a feasible solution is smaller than the given capacity. Abstractive methods generate a summary by generating concise texts that cover information contained in the original summary. This is much more difficult than an extractive method, and therefore extractive methods are currently receiving greater attention.

For obtaining a good summary in extractive summarization, we have to design an appropriate score function that measures the quality of a summary. In designing a score function, we have to consider the trade-off between the difficulty of the optimization problem and the quality of the summary. If we design a score by setting the weights for each textual unit and defining the score of a summary as the sum of weights of the selected textual units, we can then maximize the score by solving a 0-1 knapsack problem. A 0-1 knapsack problem can be exactly solved in pseudo polynomial time if we exploit a dynamic programming algorithm. However, the quality of the obtained summaries may not be high since the score does not consider the redundancy in a summary. More precisely, if there are two textual units that have large weights but are very similar, a summary containing both of them is then given a high score. However, the two units are obviously redundant as a summary, and a summary containing only one of them may be more concise and informative. Hence, in previous works, objective functions were designed to avoid redundancy. For example, Filatova and Hatzivassiloglou [33] designed an objective function by formulating it as a maximum coverage problem. In this formulation, each textual unit is regarded as a set of conceptual units, and the score of a summary is defined as the summary of weights of conceptual units contained in the summary. Since redundant conceptual units are counted only once, this formulation can avoid a high score for redundant summaries. However, differing from the 0-1 knapsack problem, the objective function cannot be maximized using a dynamic programming algorithm. The maximization problem becomes difficult, and we have to resort to using certain heuristic algorithms such as a greedy search algorithm [33], or formulate the problem as an integer linear programming (ILP) problem and solve it using an ILP solver [78]. This research shows that the trade-off between the expressive power of an objective function and the difficulty of obtaining an optimal solution makes it difficult to design an appropriate objective function that has sufficient representing power and can make the optimization problem easy to solve.

We propose a new scheme for designing an objective function for an extractive summarization. We make an objective function as the sum of objective functions, each of which represents a different aspect of a desirable summary. Our scheme is based on an efficient optimization scheme called Lagrangian relaxation [13]. By exploiting a Lagrangian relaxation, we can solve an optimization problem by repeating two steps: (i) individually solving each sub-problem, and (ii) combining the solutions of the sub-problems to update the solution to the original problem. We formulate three different aspects of a text summarization problem into different small optimization problems, where each problem can be solved exactly using a DP algorithm, or approximately using techniques for submodular function maximization [55, 56]. Our proposed method runs much faster compared with an exact algorithm that uses a commercial ILP solver, and returns high-quality summaries compared with previous approximate summarization algorithms.

Learning Parameters of Probabilistic Logic Programs with Knowledge Compilation The second algorithm we propose in Chapter 3 is for parameter learning in probabilistic logic program (PLP) models. In artificial intelligence research, mathematical logic has been used for a long time as an important language for describing knowledge inherent to problem domains, and as a tool for conducting inferences. Although mathematical logic can describe the knowledge inherent in the target domain in a precise manner, it cannot handle the inherent uncertainty. In contrast, statistical models such as probabilistic graphical models [48] can handle uncertainty, and are widely used in many ML and NLP tasks. However, since these models can describe relationships only at the propositional level, they are unsuitable for describing complex relationships.

PLP models are extensions of first-order logic programs that can handle probabilistic distributions. Because they are also statistical models, they have both the descriptive power of first-order logic and the ability to perform probabilistic inference. As statistical models, PLP models can be regarded for use in statistical relational learning [34], and many PLP models have been proposed. Stochastic Logic Programs (SLP) [62], Independent Choice Logic (ICL) [69], PRISM [75], and ProbLog [28] are popular PLP models. Many of them owe their theoretical background to Sato's distribution semantics, which is the semantics of first-order logic programs that defines the probabilistic distribution in a possible world [74]. PLP models are used in applications that require the handling of both relationships between elements existing in the target domain and uncertainty, such as in link prediction problems or temporal component analysis.

As with other statistical models used in ML tasks, designing an efficient learning algorithm for PLP models is important since we can apply a PLP model to various applications if efficient learning algorithms are provided for the model. Many learning algorithms have also been proposed for PLP models [38, 75, 22]. Similar to general parameter estimation problems for statistical models, a parameter estimation problem for PLP models can be formulated as a numerical optimization problem. These algorithms tend to solve more complex optimization problems compared with those for ordinal statistical models, since PLP models tend to have a complex and discrete structure. Such a complex structure is a byproduct of the high descriptive power of PLP models, and we therefore have to design more efficient algorithms to make PLP models applicable to complex problems.

Previous approaches for the parameter estimation problem of PLP models take either of the following two approaches. The first approach is to maximize an approximate objective function to make the optimization problem much easier. For example, Domingos and Lowd [30] proposed a parameter estimation algorithm for Markov logic network models. They estimate the parameters by maximizing a pseudo negative log-likelihood function instead of a log-likelihood function.

The second approach is to use the discrete structure of a logic program. The computation of the likelihood of a PLP model based on Sato's distribution semantics requires all possible interpretations of facts consistent with the given training examples to be found. This process is known to be #P-complete, which makes the computation of log-likelihood intractable. The parameter learning algorithm for the ProbLog model proposed by Gutmann et al. [38] tackles this problem by exploiting the discrete structure of logic programs by representing the set of possible interpretations using a Binary Decision Diagram (BDD) [9], a data structure that represents a Boolean function as a directed acyclic graph. To represent an n-ary Boolean function using a truth table, the number of entries of the table becomes 2^n. In contrast, if we use a BDD to represent the same function, we can represent it as a DAG whose number of nodes is far smaller than 2^n. Gutmann et al. use a BDD to represent the set of all possible interpretations of a ProbLog program, and use its structure to perform the Expectation-Maximization (EM) algorithm to estimate the parameters of a ProbLog program. Using a BDD, we can compute the log-likelihood in an amount of time proportional to the size of the BDD representing all possible interpretations. Although the size of a BDD strongly depends on the Boolean function, in a general case, we can speed up the parameter learning algorithm using a BDD.

Our parameter estimation algorithm for a PLP model takes the later approach, and uses the decomposable, deterministic negation normal form, (d-DNNF) [25], which is a kind of compiled knowledge representation similar to a BDD. A compiled knowledge representation is a kind of knowledge representation that is more suitable for processing. For example, the first-order logic is suitable for representing knowledge inherent to the problem domain; however, when we directly use knowledge represented by a set of first-order clauses, it may take time to obtain the required results. We therefore convert the set into another form that is more suitable for the computation. A BDD is one of the most frequently used representation forms of compiled knowledge since it can represent a Boolean function as a compact DAG, and can efficiently perform some important Boolean operations such as AND, OR, and XOR between Boolean functions represented by BDDs. The d-DNNF is a subclass of a negation normal form (NNF) [25], and is also a popular compiled knowledge representation. A NNF also takes the form of a DAG and represents a Boolean function.

The difference between our proposed algorithm and previous parameter learning methods is that our algorithm can impose sparsity on the solutions. Using a compiled knowledge representation, we can reduce the computational costs required for performing inference with a PLP model such as ProbLog. However, the amount of computational time increases with the number of probabilistic parameters used in the program. As a worst case, it can become exponential to the number of parameters. Hence, PLP models with a smaller number of parameters are preferable. Our algorithm uses both a penalty term and a projected gradient algorithm [5] to learn a PLP model that has a smaller number of probabilistic parameters. Compiled knowledge is useful, but is inflexible in the sense that it is only efficient if we also perform certain predefined operations. The previous BDD-based approach is useful but only allows the EM algorithm to be performed with it, and we cannot optimize an objective function that has a penalty term. We therefore provide a new projection gradient algorithm that can exploit the structure of a logic program, and can optimize a penalty-added objective function. A projection gradient algorithm is an iterative algorithm for solving numerical optimization problems under the constraint in which feasible solutions must be contained in a given convex set. The algorithm solves this type of optimization problem by iteratively performing both a gradient descent and projection into the convex set.

The main contribution of our algorithm is to show a new way of using a compiled knowledge representation to perform a projection gradient algorithm. Previous parameter learning algorithms used an EM algorithm or gradient projection algorithm for optimization. In contrast, our projection gradient formulation enables a flexible optimization problem that can reflect additional requirements for a PLP model.

Accelerating Repeated Matrix Multiplications using Compressed Sparse Matrix Representation In Chapter 4, we show another algorithm for numerical optimization problems appearing in relational learning tasks. Differing from the two previous algorithms, the new algorithm focuses on solving only an optimization problem, and is not concerned with designing a model. We deal with optimization problems that can be solved using repeated matrix multiplications with an adjacency matrix. Many important ML algorithms that handle relationships between objects can be solved in this way. For example, PageRank [67], one of the most popular algorithms for ranking the nodes of a directed graph, is implemented using a power iteration method, i.e., iteratively performing matrix multiplications between the adjacency matrix of the graph and a vector. Non-negative matrix factorization (NMF) [51] is another important example that exploits repeated matrix multiplications. NMF is an algorithm for factorizing a data matrix into two small matrices, using the property in which all three matrices have no negative elements. NMF is widely used for analyzing the relationships inherent in a dataset represented in a matrix form. If the target matrix is small, these matrix-multiplication based algorithms are generally not slow because a matrix multiplication can be efficiently performed. However, these algorithms tend to be applied to large matrices, and as the size of the matrix begins to increase, these matrix-multiplication based algorithms begin to require additional time.

To accelerate repeated matrix multiplications, we propose a new data structure, called an adjacency forest, which is used for representing an adjacency matrix. Similar with the two previous approaches, in decomposing the problem, an adjacency forest also exploits the problem's inherently discrete structure. An adjacency forest is a data structure that represents a matrix as a set of trees. Each tree represents a component of the adjacency matrix, and can represent the component as a compact tree made by sharing equivalent non-zero elements within the component. Using an adjacency forest, we can perform multiplication between matrices with a number of scalar operations proportional to the size of the adjacency forest. If we can reduce the size of an adjacency forest to much smaller than the number of non-zero elements inherent in the matrix, we can reduce the number of scalar operations and speed up the optimization problems.

Efficient algorithms specialized for NMF or PageRank have been proposed [44, 43, 52]. These algorithms use the characteristic features of each problem. Our adjacency forest is a general approach because it can be used for applications that require repeated matrix multiplications. Fast matrix multiplication algorithms have also been proposed [20, 85]. These algorithms assume that the matrices are dense. Our adjacency forest may be faster than these matrix multiplication algorithms when the matrix is sparse.

Since an adjacency forest can be seen as a kind of Zero-suppressed Binary Decision Diagram (ZDD) [60], we can state that an adjacency forest is also a kind of compiled knowledge representation. ZDD is a variant of a BDD, and represents a family of sets as a DAG that is very close to a BDD. The difference between a ZDD and BDD is that a ZDD can represent a sparse family of sets in a very concise form, where we consider a family of sets as sparse if the size of every set contained in the family of sets is small compared with its base. If we represent an adjacency matrix as a family of sets, and represent the set as a ZDD, then the ZDD becomes an adjacency forest.

We next describe the above three algorithms for three concrete ML and NLP tasks. These algorithms were designed for concrete tasks. However, the main motivation of our work is not focused on showing the concrete algorithms for these problems. Rather, we propose a new optimization framework that can exploit the discrete structure inherent to the problem domain. Each of the three methods is flexible in the sense that it can be applied to different optimization problems with small modifications.

Chapter 2

Lagrangian Relaxation for Scalable Text Summarization while Maximizing Multiple Objectives

In this chapter, we propose a text summarization algorithm. A text summarization problem can be seen as a combinatorial optimization problem of selecting the subset of given document. Our optimization algorithm can We use discrete structure of documents to efficiently solve the optimization problem whose score function is composed as a combination of multiple objective functions. Most part of the contents in this chapter comes from [65].

2.1 Text Summarization as a Combinatorial Optimization Problem

Automatic text summarization is the task of generating a concise summary from given documents. To make high quality summaries, it is important to define appropriate measures with which we can evaluate summary quality. For example, the contained words and the positions of textual units in the documents are widely used for measuring summary quality. Upon securing appropriate measures, we can obtain good summaries merely by solving an optimization problem of selecting textual units that maximize these measures under some constraint on summary length.

Previous work on multi-document summarization can be divided into two categories: The first category covers methods based on the maximization of *relevance* and the minimization of *redundancy* [12, 36, 58, 78]; the other one covers graph-based methods [31, 79]. The former approach makes a summary by solving an combinatorial optimization problem of simultaneously maximizing the relevance and minimizing the redundancy of the summary, where relevance is determined by how much important information the summary contains, and redundancy is determined by how duplicative the contents of the summary are. Graph-based methods first construct a graph by regarding each textual unit in the given documents as a vertex and create edges between vertices to construct a graph; they then treat text summarization as the graph cut problem of maximizing an objective score.

Maximization of relevance and minimization of redundancy based methods offer a simple and powerful approach for approximately measuring how well a summary covers all given documents; however, measuring redundancy accurately is a subtle problem. In the human-generated summaries of the document understanding conference 2004 (DUC'04) multi-document summarization dataset, about 20% of the words contained in each summary are used more than twice. This implies that human-generated summaries tend to contain redundant contents to some extent, and severely limiting the redundancy of a summary may degrade its quality. Though it is clear that we need to control summary redundancy, deciding how strictly it must be limited is a difficult problem.

On the other hand, graph-based methods have the advantage that they can directly consider the coverage of the summary, i.e. how well a summary covers all topics contained in the original documents, because they directly represent the covering relationships between textual units. Since the original documents generally contain multiple topics, it is meaningful to consider summary coverage when selecting sentences. Though graph-based methods do not explicitly consider summary redundancy, they eventually yield less-redundant summaries since they select sentences so as to cover all topics in the documents. In other words, they provide implicit control of redundancy, which is not provided by the relevance and redundancy based methods. However, implicit control does not always work well, especially when the topics of the documents are biased. If multiple sentences on the dominant topic are selected, the summary will offer high coverage but be quite redundant.

To improve summary quality by countering the weaknesses of the above methods, we set up an objective function that combines the relevance and redundancy based formulation with the graph-based approach. For the relevance and redundancy based approach, our method introduces a graph-based objective to soften the impact of severely limiting redundancy. For the graph-based approach, our method provides a way of evaluating summary redundancy.

We can obtain the exact solution by formulating it as an *integer linear programming (ILP)* and solving it. However, this optimization problem is known to be NP-hard and we cannot solve it in feasible time if the problem size is large. If we want to use automatic summarization in applications that demand immediate responses, such as information retrieval, ILP is not suitable. We thus further propose a fast approximation method for solving the maximization problem of our objective function. Our solution is a fast and high quality optimization heuristic based on the use of *Lagrangian relaxation (LR)*. Lagrangian relaxation is widely used in the combinatorial optimization field. It makes it possible to solve a complex combinatorial optimization problem by dividing it into sub-problems, each of which is solved individually. Recently, it has been applied to some natural language tasks such as parsing [73] and machine translation [13]. Our Lagrangian relaxation based method runs much faster than ILP solvers, while returning high-quality approximate solutions.

The contributions of this chapter are summarized as follows:

- We formulate multi-document summarization as the optimization problem of maximizing three objectives, relevance, redundancy, and coverage.
- We introduce heuristics for solving the problem by using Lagrangian relaxation; the original problem is divided into sub-problems, solving each sub-problem independently, and then finding the best match of all solutions.
- We use the DUC'04 summarization dataset to challenge our proposal. Our LR-based method yields higher ROUGE scores while running in feasible computation time.

We introduce our objective function and its ILP formulation in Section 2.2. Our LR-based heuristics are detailed in 2.3. We describe the experiments and their results in Section 2.4 and related work in Section 2.5. Finally, we give the summary of this chapter in Section 2.6.

2.2 An Objective Function for Text Summarization

We formalize the text summarization problem as the optimization problem of maximizing an objective function that measures summary quality. We formalize our objective function by combining McDonald's formulation [58], a typical formulation of relevance and redundancy based formulation, with the graph-based one that measures the coverage of a summary. Hence we use the three objectives of *relevance*, *redundancy*, and *coverage*.

Relevance We define relevance as the quality of the information contained in the generated summaries. A summary is relevant if it has a lot of information relevant to the topics of the original documents.

Redundancy Redundancy measures how redundant the topics that a summary contains are. Since generally a summary cannot contain all relevant information in the documents, by minimizing redundancy we can make a summary to cover many topics and so improve its quality.

Coverage Ideally, a summary should be as informative as the original documents, hence its information coverage rate is an important measure of summary quality.

Our method simultaneously maximizes these three objectives.

Before we introduce our formulations, we first describe the notations used in this chapter. We formalize the text summarization problem as the problem of extracting sentences from multiple documents. We use D to denote a document cluster, i.e. a set of multiple documents. We assume that every document contained in D is related to the same topic. A document cluster D is represented as a set of sentences, i.e. $D = \{s_1, \ldots, s_N\}$, where s_i is the i-th sentence in D, and N is the number of sentences contained in D. We use $\mathcal{D}$ as a set of document clusters. The input of text summarization problem is a document cluster D, and the constraint of the maximum total length of summary $L_{\max}$; the output is a set of sentences S, which is a subset of D. In addition to the set notation $S \subseteq D$, we also use N-dimensional binary vectors $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \{0, 1\}^N$ to represent a summary. If the i-th sentence s_i is in the summary represented by $\mathbf{x}$, then $x_i = 1$, otherwise $x_i = 0$.

2.2.1 Relevance Score

We measure the relevance of a summary based on the importance of each sentence in it, i.e. we define the relevance of a summary as the sum of the relevance scores of its sentences. We set w_i as the relevance score of the i-th sentence s_i. Then objective function $f(\mathbf{x})$ for summary $\mathbf{x}$ becomes

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i x_i \,, \tag{2.1}$$

where $x_i \in \{0,1\}$ is a binary variable and $x_i = 1$ if s_i is in the summary represented by $\mathbf{x}$, otherwise $x_i = 0$. Following the *topic signature* [54], a measure used for extracting relevant sentences, we set the relevance score of a sentence to be the sum of the logarithm of the χ^2 score of the words contained in the sentence, multiplied by the inverse of the position of the sentence in a document. The χ^2 score of word t is defined as follows:

$$\chi^2(t) = \frac{O(O_{Dt}O_{\bar{D}t} - O_{D\bar{t}}O_{\bar{D}\bar{t}})^2}{O_t O_{\bar{t}} O_D O_{\bar{D}}} \,, \tag{2.2}$$

where O_{Dt} is the number of sentences in D that contain t, $O_{\bar{D}t}$ is the number of sentences in the set $\bar{D} = \cup_{D' \in \mathcal{D} \setminus D} D'$ that contain t, $O_{D\bar{t}}$ is the number of sentences in D that do not contain t, and $O_{\bar{D}\bar{t}}$ is the number of sentences in $\bar{D}$ that do not contain t. The relationships between these values and O_t, $O_{\bar{t}}$, O_D, $O_{\bar{D}}$, O are described in the contingency table in Tab. 2.1. We set w_i as the sum of the logarithm of the χ^2 score of each word in s_i:

$$w_i = \frac{1}{pos(s_i)} \sum_{t \in s_i} \log\left(\chi^2(t) + 1\right), \tag{2.3}$$

where $pos(s)$ is the position of sentence s_i in a document contained in D; $pos(s_i)$ ranges from 1(the first sentence in the document) to the maximum number of sentences in the document. We add 1 to the sum of $\chi^2(t)$ before taking the logarithm so as to ensure that $w_i \geq 0$.

2.2.2 Redundancy Score

We measure the redundancy of a summary by the number of different bigrams contained in the summary; if there are two summaries of the same length and one

	D	$\bar{D}$	
t	O_{Dt}	$O_{\bar{D}t}$	O_t
$\bar{t}$	$O_{D\bar{t}}$	$O_{\bar{D}\bar{t}}$	$O_{\bar{t}}$
	O_D	$O_{\bar{D}}$	O

Table 2.1: 2×2 contingency table

contains fewer different bigrams than the other, it contains redundant bigrams and thus is the more redundant summary. We set the objective function as

$$g(\mathbf{y}) = \sum_{u_i \in \Gamma(\mathbf{y})} b_i u_i \,, \tag{2.4}$$

where $u_i \in \{0, 1\}$ is a binary variable and $u_i = 1$ if the i-th bigram is contained in the summary, otherwise $u_i = 0$. $\Gamma(\mathbf{y})$ is the set of all unique bigrams contained in the summary represented by N-dimensional binary vector $\mathbf{y}$. b_i is the weight of the i-th bigram. We set b_i as the number of documents containing the i-th bigram, which is normalized so as the maximum value of the sum of the weight of the bigrams contained in a sentence is 1.

2.2.3 Coverage Score

For measuring the coverage of a summary, we use the formulation in [79]; they introduced the asymmetric similarity between every pair of sentences contained in a document set, and then evaluated a summary by how well the sentences contained in the summary cover the sentences in the document set. This objective is defined as follows

$$h(\mathbf{z}) = \sum_{i=1}^{N} \sum_{j=1}^{N} e_{ij} v_{ij} \,, \tag{2.5}$$

where v_{ij} is a binary variable and $v_{ij} = 1$ if the i-th sentence is contained in a summary $\mathbf{z}$ and the j-th sentence is regarded to be "covered" by the i-th sentence.

e_{ij} is the score for the j-th sentence covered by the i-th sentence. We assume the following three constraints; (i) a sentence can cover other sentences if it is contained in the summary, i.e. $\sum_j v_{ij} \geq 0$ if $z_i = 1$, otherwise $\sum_j v_{ij} = 0$, (ii) every sentence must be covered by exactly one sentence, i.e. $\sum_i v_{ij} = 1$ for all j, unless $\mathbf{z}$ is an empty set, and (iii) a sentence contained in the summary must be covered by itself, i.e. $v_{ii} = z_i$. With these three constraints, the problem can be regarded as finding the set of sentences that maximizes the sum of score of covered sentences. We define score e_{ij} as:

$$e_{ij} = \frac{|S(s_i) \cap S(s_j)|}{|S(s_j)|} n_j \, , \tag{2.6}$$

where $S(s_i)$ is the set of words contained in s_i. n_j is the weight of sentence s_i as defined by

$$n_j = \frac{pos(s_j)^{-1} + \cos\left(s_j, \sum_k s_k\right)}{2} \, , \tag{2.7}$$

where $\cos\left(s_j, \sum_k s_k\right)$ indicates the cosine similarity between s_j and the sum of all sentences in the documents.

2.2.4 ILP Formulation

Combining the above three objectives, we formulate text summarization as a combinatorial optimization problem that can be formulated in ILP terms as follows.

$$\begin{aligned}
\text{maximize} \quad & f(\mathbf{x}) + g(\mathbf{x}) + h(\mathbf{x}) \\
= \quad & \sum_{i=1}^{N} w_i x_i + \sum_{i=1}^{M} b_i u_i + \sum_{i=1}^{N}\sum_{j=1}^{N} e_{ij} v_{ij} && (2.8) \\
\text{subject to} \quad & \sum_{i=1}^{N} c_i x_i \le L_{\max} && (2.9) \\
& \forall j : \sum_{i}^{N} a_{ij} x_i \ge u_j && (2.10) \\
& \forall j : \sum_{i}^{N} v_{ij} = 1 && (2.11) \\
& \forall i, j : x_i \ge v_{ij} && (2.12) \\
& \forall i : v_{ii} = x_i && (2.13) \\
\forall i : x_i \in \{0, 1\}, \quad & \forall i : u_i \in \{0, 1\}, \forall i, j : v_{ij} \in \{0, 1\}, && (2.14)
\end{aligned}$$

where c_i is the length of the i-th sentence, M is the number of different bigrams in D; a_{ij} is a binary constant that equals 1 if the i-th sentence contains the j-th bigram, otherwise $a_{ij} = 0$. Constraint (2.9) ensures that the length of a generated summary is less than the length limit $L_{\max}$. Constraint (2.10) addresses bigram occurrence and states that u_i can take value 1 only if at least one sentence that contains the corresponding bigram is in the summary. This constraint corresponds to the use of $\Gamma(\mathbf{y})$ in (2.4). Constraints (2.11) to (2.13) address the covering relation between sentences. Constraint (2.11) ensures that every sentence should be covered by one sentence in the summary, and (2.12) states that v_{ij} can take value 1 only if the i-th sentence is contained in the summary. Constraint (2.13) ensures that a sentence is covered by itself if it is contained in the summary. Constraint (2.14) ensures that x_i, u_i, v_{ij} are binary variables.

With the above ILP formulation and the use of an ILP solver, we can obtain a summary that maximizes the three objectives. Though the above optimization problem is NP-hard, existing ILP solvers can find the optimal solution in feasible running time if the size of the problem is small. The running time, unfortunately, increases exponentially with problem scale, and is impractical if immediate responses are needed. We therefore introduce in the next section a heuristic that exploits *Lagrangian relaxation* to find near optimal solutions in reasonable time.

2.3 Lagrangian Relaxation

Our solution is to use the Lagrangian relaxation technique. Lagrangian relaxation (LR) is a popular technique for solving combinatorial optimization problems. Recently, Lagrangian relaxation has been applied to some NLP tasks, e.g. dependency parsing [73] and phrase-based models for statistical machine translation [13].

If we can maximize the previously noted three objectives independently, i.e. we maximize $f(\mathbf{x}), g(\mathbf{y}), h(\mathbf{z})$ for independent variables $\mathbf{x}, \mathbf{y}, \mathbf{z}$, the problem becomes easier. Unfortunately, the exact solutions for each problem are usually different, making the required solution impossible to find. To find agreement for these three sub-problems we consider the maximization problem

$$\begin{aligned} \text{maximize} \quad & f(\mathbf{x}) + g(\mathbf{y}) + h(\mathbf{z}) && (2.15) \\ \text{subject to} \quad & \mathbf{x} = \mathbf{y} = \mathbf{z}, \mathbf{x} \in \mathcal{X}, \mathbf{y} \in \mathcal{Y}, \mathbf{z} \in \mathcal{Z}, && (2.16) \end{aligned}$$

where $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are N-dimensional binary vectors and represent summaries. If the i-th sentence is contained in a summary, corresponding elements of these vectors take value 1. $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$ are sets of all possible summaries that satisfy the corresponding constraints in (2.9) to (2.13), i.e. they are the sets of candidate summaries whose length is less than $L_{\max}$. This optimization problem is the same as the one in Section 2.2.4, and it is still difficult to solve due to the constraints $\mathbf{x} = \mathbf{y} = \mathbf{z}$. Lagrangian relaxation solves this problem by relaxing it through the easing of the constraint on the equality of variables by setting the objective as defining the Lagrangian:

$$\begin{aligned} L(\boldsymbol{\lambda}, \boldsymbol{\mu}, \mathbf{x}, \mathbf{y}, \mathbf{z}) &= f(\mathbf{x}) + g(\mathbf{y}) + h(\mathbf{z}) \\ &+ \sum_{i=1}^{N} \left(\lambda_i (x_i - y_i) + \mu_i (x_i - z_i) \right), \end{aligned} \tag{2.17}$$

where $\lambda_i, \mu_i \; (1 \leq i \leq N)$ are Lagrange multipliers. Lagrangian relaxation involves minimizing the Lagrangian dual

$$L(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \max_{\mathbf{x} \in \mathcal{X}, \mathbf{y} \in \mathcal{Y}, \mathbf{z} \in \mathcal{Z}} L(\boldsymbol{\lambda}, \boldsymbol{\mu}, \mathbf{x}, \mathbf{y}, \mathbf{z}) \tag{2.18}$$

$$\begin{aligned} &= \max_{\mathbf{x} \in \mathcal{X}} \left\{ f(\mathbf{x}) + \sum_{i=1}^{N} (\lambda_i + \mu_i) x_i \right\} \\ &+ \max_{\mathbf{y} \in \mathcal{Y}} \left\{ g(\mathbf{y}) - \sum_{i=1}^{N} \lambda_i y_i \right\} \\ &+ \max_{\mathbf{z} \in \mathcal{Z}} \left\{ h(\mathbf{z}) - \sum_{i=1}^{N} \mu_i z_i \right\}. \end{aligned} \tag{2.19}$$

It is known that Lagrangian dual $L(\boldsymbol{\lambda}, \boldsymbol{\mu})$ is always larger than the value of the exact solution of the original problem, and the dual problem is to find

$$\min_{\boldsymbol{\lambda}, \boldsymbol{\mu}} L(\boldsymbol{\lambda}, \boldsymbol{\mu}). \tag{2.20}$$

By solving the dual problem, we can get the tightest upper bound of the exact solution of the original problem. To solve the dual problem, we can minimize it by using the subgradient method since $L(\boldsymbol{\lambda}, \boldsymbol{\mu})$ is a convex function. Since $x_i - y_i$, and $x_i - z_i$ are subgradients for λ_i and μ_i, we can update $\boldsymbol{\lambda}, \boldsymbol{\mu}$ by using subgradient as

$$\lambda_i^{(k)} = \lambda_i^{(k-1)} - \delta_k (x_i^{(k)} - y_i^{(k)}), \tag{2.21}$$

$$\mu_i^{(k)} = \mu_i^{(k-1)} - \delta_k (x_i^{(k)} - z_i^{(k)}), \tag{2.22}$$

where δ_k is the step size at the k-th iteration. It is known that for any sequence of step size $\delta_1, \delta_2, \delta_3, \ldots$ such that $\delta_k > 0$ for $k > 1$, the above iteration will converge to the minimum of $L(\boldsymbol{\lambda}, \boldsymbol{\mu})$ if $\lim_{k \to \infty} \delta_k = 0$ and $\sum_{k=1}^{\infty} \delta_k = \infty$ [49].

The flow of our summarization method is shown in Algorithm 2.1. Constant K is the maximum number of iterations. We first initialize the Lagrange multipliers $\boldsymbol{\lambda}^{(0)}$ and $\boldsymbol{\mu}^{(0)}$ to 0, then repeatedly update them until the k-th round or the process converges. Lines 3 to 5 are the maximization steps of each objective. Three objectives with corresponding Lagrange multipliers are maximized independently. We will show how each maximization works in the following sections. Line 6 checks the agreement between the solutions. If all three solutions are the same, we have reached an agreement solution and thus we return it and finish the procedure. If the condition is not satisfied, we update $\boldsymbol{\lambda}^{(k)}$ and $\boldsymbol{\mu}^{(k)}$ using the subgradient method and then repeat the maximization steps of lines 3 to 5 with updated $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$. If we cannot reach agreement after the K-th iteration, we finally return one of the three candidate solutions $\mathbf{x}^{(K)}$ as the final solution. Since all of these solutions satisfy the constraint on the length, they are valid summaries given length limit $L_{\max}$. As we will experimentally show in Fig. 2.1, the score of the objective function tend to increase as the iteration proceeds. We thus can obtain a solution with a high objective score even if we cannot reach full agreement.

2.3.1 Maximizing the Relevance Score

In the following three subsections, we describe the process of maximizing the objectives in Algorithm 2.1, lines 3 to 5. We first show how to maximize the relevance score. Since we define relevance score $f(\mathbf{x})$ as a linear function of $\mathbf{x}$, $f(\mathbf{x}) + \sum_{i=1}^{N}(\lambda_i^{(k-1)} + \mu_i^{(k-1)})x_i$ is also a linear function of $\mathbf{x}$. We use w_i' to represent $w_i + (\lambda_i^{(k-1)} + \mu_i^{(k-1)})$. Thus we can formulate it as an 0-1 knapsack problem that can be solved efficiently by a dynamic programming algorithm. We show the process in Algorithm 2.2. A and G are two dimensional arrays, each with $(N+1) \times (L_{\max}+1)$ entries. A is used for storing the maximum scores, while G stores the information needed for extracting the best solution after obtaining the maximum score. We can find the exact solution by sequentially updating

Algorithm 2.1. The text summarization algorithm

```
$\lambda_i^{(0)} \leftarrow 0, \mu_i^{(0)} \leftarrow 0$ for $i = 1$ to $N$
for $k = 1$ to $K$ do
    $\mathbf{x}^{(k)} \leftarrow \arg\max_{\mathbf{x} \in \mathcal{X}} \left\{ f(\mathbf{x}) + \sum_{i=1}^{N} (\lambda_i^{(k-1)} + \mu_i^{(k-1)}) x_i \right\}$
    $\mathbf{y}^{(k)} \leftarrow \arg\max_{\mathbf{y} \in \mathcal{Y}} \left\{ g(\mathbf{y}) - \sum_{i=1}^{N} \lambda_i^{(k-1)} y_i \right\}$
    $\mathbf{z}^{(k)} \leftarrow \arg\max_{\mathbf{z} \in \mathcal{Z}} \left\{ h(\mathbf{z}) - \sum_{i=1}^{N} \mu_i^{(k-1)} z_i \right\}$
    if $x_i^{(k)} = y_i^{(k)} = z_i^{(k)}$ for all $i \in \{1, \dots, N\}$ then
        return $\mathbf{x}^{(k)}$
    else
        for $i = 1$ to $N$ do
            $\lambda_i^{(k)} \leftarrow \lambda_i^{(k-1)} - \delta_k (x_i^{(k)} - y_i^{(k)})$
            $\mu_i^{(k)} \leftarrow \mu_i^{(k-1)} - \delta_k (x_i^{(k)} - z_i^{(k)})$
return $\mathbf{x}^{(K)}$
```

every entry in A and G in order, and then backtracking G. We can solve it in $O(NL_{\max} + N)$ computational time.

2.3.2 Maximizing the Redundancy Score

Maximization of the redundancy score defined in Section 2.2.2 is *a maximum coverage problem with knapsack constraint (MCKP)* and it is known to be NP-hard, see [78]. Unlike the relevance score, we cannot find the exact solution efficiently and so resort to an efficient approximation method.

We use the fact that $g(\mathbf{y})$ is a *monotone submodular function* of $\mathbf{y}$; a set function $g : 2^V \to \mathbb{R}$, which maps subset S of finite set V to a real number, is said to be a *submodular function* if the following condition holds for any subset $S, T \subseteq V$

$$g(S \cup T) + g(S \cap T) \leq g(S) + g(T) , \tag{2.23}$$

Algorithm 2.2. Summarization as the Knapsack problem.

```
$A[i][0] \leftarrow 0, G[i][0] \leftarrow 0$ for $i = 0$ to $N$
$A[0][l] \leftarrow 0$ for $l = 0$ to $L_{\max}$
for $i = 1$ to $N$ do
    for $l = 1$ to $L_{\max}$ do
        if $c_i \leq l$ and $A[i-1][l-c_i] + w'_i > A[i-1][l]$ then
            $A[i][l] \leftarrow A[i-1][l-c_i] + w'_i, G[i][l] \leftarrow 1$
        else
            $A[i][l] \leftarrow A[i-1][l], G[i][l] \leftarrow 0$
$l \leftarrow L_{\max}, \ \mathbf{x} \leftarrow \mathbf{0}$
for $i = N$ to $1$ do
    if $G[i][l] = 1$ then
        $x_i \leftarrow 1, \ l \leftarrow l - c_i$
return $\mathbf{x}$
```

There is an equivalent condition

$$g(S+v) - g(S) \geq g(T+v) - g(T)\,, \tag{2.24}$$

where $S \subseteq T \subseteq V \setminus v$. Set function $g(\cdot)$ is called *monotone non-decreasing* if $g(S) \leq g(T)$ for any $S \subseteq T$. It is known that the maximization problem of a monotone submodular function under knapsack constraints has a constant-factor approximation if we solve the problem by greedy search [55]. $g(\mathbf{y})$ is monotone non-decreasing and satisfies the definition of a submodular function. We thus use the greedy procedure to obtain the approximate solution.

We introduce a greedy procedure; it successively chooses sentences that raise the gain of unique bigrams. Algorithm 2.3 is the pseudo-code of this greedy search problem. We use $g'(\mathbf{y})$ to represent the redundancy objective with the terms of Lagrangian multipliers, i.e. $g'(\mathbf{y}) = g(\mathbf{y}) - \sum_{i=1}^{N} \lambda_i y_i$. From lines 3 to 6, we successively find sentence l that maximizes $(g(s \cup \{l\})' - g(s)')/c_l$ and add it to s if the total length is smaller than $L_{\max}$. We finally compare the obtained summary

Algorithm 2.3. A greedy algorithm for maximizing the redundancy objective

```
1: $s \leftarrow \mathbf{0}, C \leftarrow \{1, \ldots, N\}$
2: while $C \neq \emptyset$ do
3:     $k \leftarrow \arg\max_{l \in C} \frac{g'(s \cup \{l\}) - g'(s)}{c_l}$
4:     if $\sum_{i \in G} c_i + c_k \leq L_{\max}$ and $g'(s \cup \{k\}) - g'(s) \geq 0$ then
5:         $s \leftarrow s \cup \{k\}$
6:     $C \leftarrow C \setminus \{k\}$
7: $i^* \leftarrow \arg\max_{i \in \{1,\ldots,N\}, c_i \leq L_{\max}} g'(\{i\})$
8: return $\mathbf{y} = \arg\max_{\mathbf{y}' \in \{\{i^*\}, s\}} g'(\mathbf{y}')$
```

with the summaries consisting of only one sentence (lines 7, 8). We then return the summary that gives the highest score.

2.3.3 Maximizing the Coverage Score

Maximizing the coverage score objective is also an NP-hard problem and it is difficult to obtain the exact solutions. We thus resort to approximation as in the case of the redundancy objective. We can easily see $h(\mathbf{z})$ is also a monotone submodular function, and we can obtain reliable approximate solution by applying the greedy approach similar to Algorithm 2.3.

2.4 Evaluation

2.4.1 Settings

We conduct experiments using the document understanding conference 2004 (DUC'04) dataset for evaluating the quality of summaries generated by proposed methods. All of the experiments are executed on a Xeon 7400 2.66GHz CPU with 256GB RAM running CentOS 5.4 Linux. The Lagrangian relaxation method was implemented in C++ and we set the number of max rounds to $K = 500$, and set subgradient step $\delta^{(k)}$ to $0.5/k$, where k is the number of iterations. For solving integer linear programming, we used ILOG CPLEX version 12.1.0.

DUC'04 dataset is a dataset distributed at the document understanding conference of 2004 and is used for evaluating multi-document summarization algorithms. We use the setting of task 2, which is a multi-document summarization task on English news articles. The dataset consists of 50 document clusters, each of which contains about 10 documents. The task is to make a summary for each cluster. We set the summary length limit to 665 bytes, the settings are used in [56].

We evaluated the summary of DUC'04 dataset using ROUGE version 1.5.5[1] . ROUGE [53] is widely used in studies of summarization for evaluating the quality of system-generated summaries. We used both the recall(R) and F-measure(F) score of ROUGE-1, 2 to evaluate our approach.

We created two variants of the proposed method for both datasets. The first uses the combination of all objectives, i.e. relevance, redundancy, and coverage, by applying ILP solver (denoted as Proposed (ILP)). The second maximizes the same objective with the Lagrangian relaxation base method (denoted as Proposed (LR)). We also evaluated the result of maximizing the pair of relevance and redundancy using Lagrangian Relation (Rel+Red(LR)), and the result of maximizing the coverage score with a greedy algorithm (Cov(Greedy)). These two settings can be seen as variants of previous summarization techniques, like the relevance and redundancy based method, and the graph-based method, respectively. (Rel+Red(LR)) also can be solved by Lagrangian relaxation, and Cov(Greedy) can be obtained with a greedy algorithm.

We compared the results from the DUC'04 dataset with three state-of-the-art methods that do not resort to an ILP solver; submodular function maximization (Lin) [56], McDonald's dynamic-programming-like method (McDonald) [58],

[1] Options used: -n 2 -f A -p 0.5 -b 665 -m -t 0 .

Table 2.2: ROUGE scores of the DUC 2004 dataset. We use bold font when the ROUGE score is significantly different from CLASSY .

	ROUGE-1		ROUGE-2	
Method	F	R	F	R
Proposed (LR)	0.390	**0.397**	0.098	0.096
Rel+Red (LR)	0.374	0.381	0.095	0.094
Cov (Greedy)	0.382	0.385	0.088	0.088
Lin	0.389	0.394	-	-
CLASSY	0.377	0.382	0.092	0.091
McDonald	0.362	0.338	0.081	0.086
Proposed (ILP)	**0.396**	**0.401**	**0.099**	**0.101**

Table 2.3: Comparison of ILP and LR: objective function value and computation time. The numbers in parenthesis are standard deviations.

Method	Objective	Time(sec.)
Proposed(ILP)	14.8	109 (140)
Proposed(LR)	14.0	3.46 (2.56)

and CLASSY [19]. CLASSY is the method that showed the best ROUGE-1 score in the DUC'04 conference. We performed Wilcoxon signed rank tests for paired samples with significance level of 0.05.

2.4.2 Results and Discussions

We show the ROUGE scores for the DUC'04 dataset in Tab. 2.2 [2] . Though they are approximate results, Proposed (LR) shows higher scores than the state-of-the-art methods. We can see Proposed (ILP) offers the highest ROUGE scores.

[2] Due to differences in the preprocessing of the reference summaries, the scores of CLASSY is slightly differ from those reported in [56].

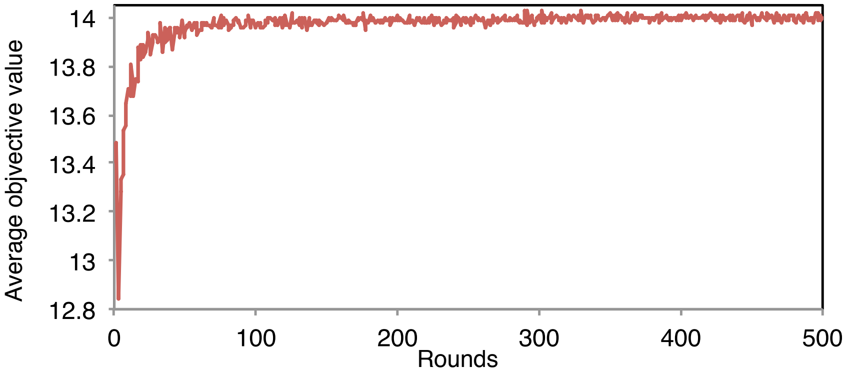


Figure 2.1: The relation between the number of iterations and objective values for the DUC'04 dataset.

This result suggests the effectiveness of our combined objective. By comparing the results of Proposed (LR) to Rel+Red (LR) and Cov(Greedy), we can see that Proposed (LR) achieved the highest ROUGE (F/R) scores in these three settings. This result also confirms that the proposed combination of objective functions can improve summary quality.

Figure 2.1 shows the relation between the number of rounds and the average value of the objective for Proposed (LR). We can see the average objective value increased with round number.

We show the values of objective function and computation times of ILP and LR of DUC'04 dataset in Tab. 2.3. We can see that LR offers high objective value, about 94% of the exact solution obtained by ILP. To see the cause of approximation errors, we further compared the average objective values of converged and not converged cases of Proposed (LR). Proposed (LR) converged on 18 tasks over 50 tasks after 500 iterations, and the average value of them was 93.3% of Proposed (ILP), while that of the not converged settings was 94.6%. This counterintuitive result suggests the convergence of Algorithm 1 did not contribute to increase the objective values. This implies the approximation errors of Proposed (LR) were mainly caused by the use of greedy search, since it is the only approximation factor of Proposed (LR) other than the use of Lagrangian relaxation.

The average computation time of Lagrangian relaxation is much shorter than that of ILP. The computation time of ILP strongly depends on the size of the problem; in the best case, the computation finished in a second, in the worst case, however, it took about 10 minutes. The LR based method finished, at worst, in 10 seconds, and its variance is much smaller.

2.5 Related Work

Many proposed multi-document summarization methods measure the relevance and redundancy of the summary. MMR (Maximal Marginal Relevance) [12, 36] is the first one. MMR is a greedy algorithm that forms the summary by sequentially selecting the most relevant sentences that are not similar to already selected sentences. MMR is simple and efficient but the algorithm can not target the global optimal solution. [58] extended MMR and proposed an ILP (Integer Linear Programming) based approach to obtain globally optimal solutions. ILP solvers can find the exact solution but take too long for large scale data. As an approximation of the model, he also proposed a method for solving Knapsack-like problems by using dynamic programming. If we can ignore redundancy, the method is identical to the Knapsack problem and it will obtain the global optimum. Otherwise, it is not the Knapsack problem, and cannot obtain the exact solution.

Formulating the multi-document summarization problem as a maximum coverage problem can be regarded as a kind of relevance and redundancy based summarization. [33] first regarded text summarization as the maximum coverage problem, and solved it by using a greedy algorithm [46]. [88] employed a stack decoder [42] for solving the problem. [78] and [35] formulated the maximum coverage problem as ILP and employed the branch-and-bound algorithm. The method can achieve globally optimal solutions. As we noted in Section 2.1, how to measure summary redundancy is a subtle problem that must be addressed.

The first graph-based summarization method was proposed by [31]. Their method, LexRank, calculates the centrality of each sentence contained in the given documents according to the similarity between each pair of sentences; sentences with high centrality scores are then extracted to make a summary. [79] proposed an extractive summarization method based on the budgeted median problem. Their method can also be viewed as a kind of graph-based method. Graph-based methods cannot explicitly measure summary redundancy, so adding a function for controlling redundancy might improve summary quality.

Other than the settings of the objective function, previous methods also can be classified by how they maximize the objective functions. [58], [78], and [79] show how to solve the problem exactly using an ILP solver. However, exact solutions take too long if the problem is large. Other works describe methods for maximizing an objective function approximately, but they cannot guarantee the quality of the solutions. [55, 56] showed that the text summarization task can be solved as a maximization problem of a submodular function and showed an efficient greedy algorithm. Their algorithm can find a solution whose score is guaranteed to be, at worst, $1 - e^{-1/2}$ of the exact solution. Since our proposed method constructs the objective function by combining submodular functions, it achieved higher performance in the evaluation described above.

Recently, [86] proposed a multi-document summarization method based on ILP settings. They made an objective function by combining multiple aspects containing sentence compression. Though our focus in this chapter is on designing a good objective function, based on relevance and redundancy, our LR-based method can be easily extended by the addition of other aspects.

2.6 Chapter Summary

We proposed an automatic text summarization method that generates summaries by solving an optimization problem that consists of three sub-problems; maximization of the information contained in the summary, minimization of redundant information, and maximization of coverage. We formulated the problem in integer linear programming terms, and introduced a Lagrangian relaxation method to solve the problem in feasible computation time. We used the DUC'04 multi-document summarization dataset to evaluate the performance of the proposed method, and found that it can attain the highest ROUGE score by maximizing our objective with ILP. Moreover, our LR based heuristics also show higher ROUGE scores than existing text summarization schemes in much shorter running time.

Chapter 3

A Sparse Parameter Learning Method for Probabilistic Logic Programs

In this chapter, we show a parameter estimation algorithm for Probabilistic Logic Programs. Probabilistic Logic Program is a statistical model and it is an extension of first-order logic. PLP models can represent complex relationships inherent in the problem domain. Our algorithm can estimate sparse PLP model. Since the complexity of a PLP model depends on the number of parameters contained in the model, PLP models with less number of parameters are preferable since it takes much time to perform inference with a complex PLP model. Our algorithm exploits discrete structure inherent in the PLP model by using knowledge compilation, and it can be used in combination with penalization technique. Most part of the contents in this chapter comes from [64].

3.1 Parameter Estimation for Probabilistic Logic Program

Probabilistic logic program (PLP) is an extension of a logic program that can perform probabilistic inferences. A PLP is a kind of statistical relational model [34], and was developed for modeling complex and uncertain relationships. Many PLP models have been proposed (e.g., [62, 75, 28]), and most of existing PLP models are based on Sato's distribution semantics [74], which defines a probability distribution over possible worlds by introducing probabilistic ground facts into a logic program.

A problem with these PLP models is the difficulty related to inferences. In the worst case, a PLP model that is based on the distribution semantics may require computational time exponential to the number of probabilistic parameters contained in the model. It prevents PLP models from being applied to large problems. Although some efficient exact and approximate inference algorithms have been proposed for these models [28], inference is still difficult and hence PLP models with fewer probabilistic parameters are preferable.

In this chapter, we propose a parameter learning algorithm for probabilistic logic programs. The proposed algorithm can reduce the number of probabilistic factors contained in the estimated model. Parameter estimation algorithms for PLP models proposed in previous research (e.g, [38, 75]) are not intended for estimating compact models, i.e., a model with fewer probabilistic parameters. In order to estimate a compact model, we add penalty terms to the negative log-likelihood function and then minimize it to estimate probabilistic parameters. Penalty terms are often introduced into machine learning algorithms to impose some restrictions on the estimated parameters. A well-known penalty term is ℓ_1 norm, which is applied for obtaining sparse solutions [1], but the ℓ_1 or other sparsity inducing norms cannot be directly applied to the parameter learning problem of PLP models. The new penalty term we propose in this chapter induces the learned parameters to take either 0 or 1. When a probabilistic parameter is 0 or 1, we can remove it or treat it as deterministic to obtain a PLP model with fewer number of probabilistic parameters. We also give an efficient optimization algorithm that can run on a compiled knowledge representation. Given an objective function with penalty terms, we minimize it by applying a projected gradient algorithm [5]. A projected gradient algorithm is an efficient method for solving an optimization problem with constraints that a solution must be contained in a convex set, and we present a method to run it with a compiled knowledge representation, which is obtained by transforming a logical model into another form, and reduces the computational time. As the transformation, we use a deterministic and decomposable negation normal form (d-DNNF) [25] so that we make optimization problems tractable.

In the following, we use ProbLog [28] as a concrete example of PLP model and we propose a parameter learning algorithm on it. However, with slight modification our method may also be applicable to other PLP models. We give a class of PLP models to which our parameter learning algorithm can be applied.

3.2 Preliminaries

We first briefly introduce some basic notations used in this chapter. A term is a variable, a constant, or a function applied to terms. Let $\mathtt{q}(\mathtt{t_1}, \ldots, \mathtt{t_k})$ be an atom, where $\mathtt{t_1}, \ldots, \mathtt{t_k}$ are terms and $\mathtt{q}$ is a predicate of arity k. Definite clauses are universally quantified expressions of the form $\mathtt{h}$ `:-` $\mathtt{b}_1$`, ...,` $\mathtt{b}_n$, where $\mathtt{h}, \mathtt{b_1}, \ldots, \mathtt{b_n}$ are all atoms, and $\mathtt{h}$ is the head of a clause and $\mathtt{b_1}, \ldots, \mathtt{b_n}$ are the body. A clause without a body is a fact. A substitution θ is an expression of the form $\{V_1/\mathtt{t_1}, \ldots, V_m/\mathtt{t_m}\}$ where V_i are different variables and $\mathtt{t_i}$ are terms. If a substitution θ is applied to an expression e, then the instantiated expression $e\theta$ is made by simultaneously replacing the variables V_i in e with $\mathtt{t_i}$. An expression is called ground if it has no variables. The semantics of a set of definite clauses is given by its least Herbrand model, i.e., the set of all ground facts entailed by the theory.

A ProbLog theory $\mathcal{T}$ consists of both a set of labeled facts F and a set of definite clauses KB. Let f_i be a fact contained in F and $w_i \in [0,1]$ be the label of f_i. w_i represents the probability that each ground substitution $f_i\theta$ is true in the theory. We refer to an annotated fact $w_i :: f_i$ as a *probabilistic fact*.

Example 3.1. *The following is an example of ProbLog program.*

```
0.1::burglary.    0.2::earthquake.
0.7::hears_alarm(X) :- person(X).
person(mary).    person(john).
alarm :- burglary.    alarm :- earthquake.
calls(X) :- alarm, hears_alarm(X).
```

In this program, `burglary.` *and* `earthquake.` *are facts and their probabilities are* 0.1 *and* 0.2*, respectively.* `0.7 :: hears_alarm(X) :- person(X).` *is a notation that represents two probabilistic facts,* `0.7::hears_alarm(mary)` *and* `0.7::hears_alarm(john)`.

Given a finite number of possible ground substitutions $\{\theta_{i,1},\ldots,\theta_{i,K_i}\}$ for each probabilistic fact $w_i :: f_i$, a ProbLog program $\mathcal{T}$ defines probability distribution over total choices L, where L is a subset of the set of all ground facts $L_{\mathcal{T}} = \{f_1\theta_{1,1},\ldots,f_1\theta_{1,K_1},\ldots,f_N\theta_{N,1},\ldots,f_N\theta_{N,K_N}\}$.

$$P(L|\mathcal{T}) = \prod_{f_i\theta_{i,k}\in L} w_i \prod_{f_i\theta_{i,k}\in L_{\mathcal{T}}\setminus L} (1-w_i)\ . \tag{3.1}$$

Using the above definition of probability $P(L|\mathcal{T})$, we define the success probability of a query literal q as

$$\begin{aligned} P(q|\mathcal{T}) = &\sum_{L\subseteq L_{\mathcal{T}}, L\cup KB\models q} P(L|\mathcal{T}) \\ &\sum_{L\subseteq L_{\mathcal{T}}} \delta(q, KB\cup L)\cdot P(L|\mathcal{T}) \end{aligned}$$

where $\delta(q, KB\cup L) = 1$ if there exists a substitution θ such that $KB\cup L \models q\theta$, and 0 otherwise.

Example 3.2. *For the program in Example 3.1, $L_{\mathcal{T}}$ contains four probabilistic facts, and hence there are $2^4 = 16$ possible $L \subseteq L_{\mathcal{T}}$. If q =* `alarm`*, $\delta(q, KB \cup L) = 1$ if either* `burglary` *or* `earthquake` *is contained in L, and its probability is $P(\texttt{alarm}|\mathcal{T}) = 1 - P(\neg\texttt{burglary} \wedge \neg\texttt{earthquake}|\mathcal{T}) = 1 - 0.9 \times 0.8 = 0.28$.*

3.3 Parameter Learning

3.3.1 Motivating Examples

Before presenting our parameter learning algorithm, we first show what it aims to do. What we want is a compact ProbLog program, but a compact program is not just a program with fewer clauses. At this point, our algorithm differs slightly from sparse learning algorithms [1]; sparse learning algorithms try to obtain sparse models by letting many parameters take zero. By contrast, our learning algorithm induces many parameters to take either 0 or 1. This setting is motivated by the following two examples.

$$w_1\texttt{:: q.} \quad w_2\texttt{:: r.} \quad \texttt{p :- q.} \quad \texttt{p :- r.}$$

Suppose that we are given training examples $\mathcal{D}$ that only contains literal `p`, and we want to set parameters w_1, w_2 so as to maximize the log-likelihood of $\mathcal{D}$. If training examples follow probabilistic distribution $P(\texttt{p}) = 0.5$, then any combination of parameters w_1 and w_2 that satisfies $1.0 - (1 - w_1)(1 - w_2) = 0.5$ will maximize log-likelihood. However, if we set $w_1 = 0.0$ and $w_2 = 0.5$ (or equivalently, set $w_1 = 0.5$ and $w_2 = 0.0$), then we can remove one probabilistic fact from the program. This clearly reduces the size of the obtained program. The above approach is equivalent to theory compression [27], which allows some parameters of a ProbLog program to be zero to obtain a more concise logic program.

We give another example.

$$w_1\texttt{:: q.} \quad w_2\texttt{:: r.} \quad \texttt{p :- q, r.}$$

With this program, the probability $P(\mathtt{p})$ is represented as $P(\mathtt{p}) = w_1 w_2$. As same as the previous example, suppose that we are given training examples $\mathcal{D}$ that only contains literal $\mathtt{p}$, and we want to set parameters w_1, w_2 so as to maximize log-likelihood of $\mathcal{D}$. If training examples follow probabilistic distribution $P(\mathtt{p}) = 0.5$, many combinations of parameters are possible. However, if we set $w_1 = 1.0$ and $w_2 = 0.5$ (or $w_1 = 0.5$ and $w_2 = 1.0$), it means we treat the probabilistic fact $\mathtt{q}$ as a deterministic (i.e., non probabilistic) fact. Since the complexity of probabilistic inference with a ProbLog program depends on the number of probabilistic facts, treating some probabilistic facts as deterministic facts also helps to reduce the cost of probabilistic inference.

Our algorithm uses a penalty function to obtain these two types of reduction of probabilistic parameters, namely, (i) removing probabilistic facts and (ii) substituting probabilistic facts with deterministic ones.

3.3.2 Learning Algorithm

Our parameter learning algorithm follows the learning from interpretation setting that has been proposed in [38] since it is more general than the learning from entailment settings as used in other learning algorithms [37, 75, 22]. Let I be a partial interpretation of ground atoms in $L_{\mathcal{T}}$, which determines the truth values of some atoms in $L_{\mathcal{T}}$. We represent a partial interpretation as $I = (I^+, I^-)$ where I^+ contains all true atoms and I^- all false atoms. We define the probability of a partial interpretation I in a way that is similar to that for an atom q:

$$P(I|\mathcal{T}) = \sum_{L \subseteq L_{\mathcal{T}}} \delta(I^+, KB \cup L)\bar{\delta}(I^-, KB \cup L)P(L|\mathcal{T}),$$

where $\delta(I^+, KB \cup L) = 1$ if $KB \cup L \models q$ for all atoms $q \in I^+$, and $\bar{\delta}(I^-, KB \cup L) = 1$ if $KB \cup L \not\models q$ for all atoms $q \in I^-$.

Parameter learning is formalized as the task of finding a set of parameters $\hat{\mathbf{w}} = \{\hat{w}_1, \ldots, \hat{w}_N\}$ that minimizes the objective function given training examples.

Let a set of interpretation $\mathcal{D} = I_1, \ldots, I_M$ be training examples. We make the objective function as a combination of the negative log-likelihood and a penalty function that encourages parameters to take either 0 or 1. We therefore define the objective function $g(\mathcal{T}(\mathbf{w}), \mathcal{D})$ as

$$g(\mathcal{T}(\mathbf{w}), \mathcal{D}) = \ell(\mathcal{T}(\mathbf{w}), \mathcal{D}) + \lambda h(\mathbf{w}) \ ,$$

where $\mathcal{T}(\mathbf{w})$ represents a ProbLog program whose parameters are $\mathbf{w}$, and $\ell(\mathcal{T}(\mathbf{w}), \mathcal{D})$ represents a negative log-likelihood function and $h(\mathbf{w})$ is a penalty function. A parameter λ controls the effect of the penalty term. We minimize the above objective function by considering that $0 \leq w_i \leq 1$ for all $1 \leq i \leq N$. We define the negative log-likelihood function $\ell(\mathcal{T}(\mathbf{w}), \mathcal{D})$ as

$$\ell(\mathcal{T}(\mathbf{w}), \mathcal{D}) = -\sum_{j=1}^{M} \log P(I_j | \mathcal{T}(\mathbf{w})) \ .$$

It is simply defined as the logarithm of the product of the probabilities $P(I_j | \mathcal{T}(\mathbf{w}))$ for $j = 1, \ldots, M$. We define the penalty function $h(\mathbf{w})$ as

$$h(\mathbf{w}) = \sum_{i=1}^{N} \{\log(w_i + \varepsilon) + \log(1 - w_i + \varepsilon)\} \ , \tag{3.2}$$

where ε is a small positive value that is added to avoid the function becoming $-\infty$ at $w_i = 0$ or $w_i = 1$. The penalty term takes smaller values when w_i is near 0 or 1, hence we can encourage w_i to take 0 or 1 by adding $h(\mathbf{w})$ to the negative log-likelihood $g(\mathcal{T}(\mathbf{w}), \mathcal{D})$. Hence this penalty term can contribute to reduce the number of parameters. We show an example of $h(\mathbf{w})$ when $N = 1$ in Fig. 3.1.

The minimization problem of only negative log-likelihood function $\ell(\mathcal{T}(\mathbf{w}))$ is the same as the problem solved in the LFI-ProbLog algorithm shown in [38], and it can be solved efficiently by using the Expectation Maximization (EM) algorithm. The difference between the LFI-ProbLog algorithm and ours is the use of the penalty term $h(\mathbf{w})$. This addition makes EM algorithm inapplicable for our problem. We therefore introduce a new optimization method that is based on the projected gradient algorithm.

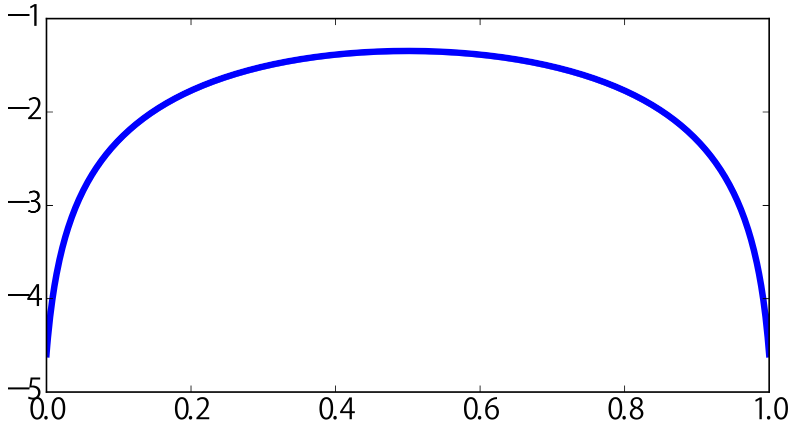


Figure 3.1: Shape of the penalty term $h(\mathbf{w})$, where $N = 1$ and $\varepsilon = 0.001$.

3.3.3 Projected Gradient Algorithm

The minimization problem we introduced in the previous section cannot be solved with the EM algorithm, we therefore propose a new method that is based on the projected gradient algorithm. The projected gradient algorithm [5] is used for minimization problems with the constraint that the variables must be contained in a convex set. In the present case, w_i must satisfy $0 \leq w_i \leq 1$, and hence the region in which $\mathbf{w}$ is contained is convex. We can therefore employ the projected gradient method for our problem.

The projected gradient algorithm is an extension of a gradient descent algorithm, and it solves optimization problems by repeating gradient computation and projection onto a convex set. Algorithm 3.1 shows the projected gradient algorithm. After initializing $\mathbf{w}^0$ (line 1), it repeatedly updates $\mathbf{w}^k$ until it converges (lines 2 to 7). First we compute $\mathbf{x}^k$ from the current $\mathbf{w}^k$ and the gradient $\nabla g(\mathbf{w}^k)$ (line 3). This step is the same as an ordinary gradient descent algorithm, then we project $\mathbf{x}^k$ into the domain that satisfies $0 \leq w_i \leq 1$ $(1 \leq i \leq N)$ to obtain $\mathbf{w}^{k+1}$. Here, function $\mathbf{proj}(\mathbf{x})$ is a projection function that maps $\mathbf{x}$ to $\mathbf{w}$ such that satisfies $0 \leq w_i \leq 1$ for all $1 \leq i \leq N$ and minimizes $||\mathbf{x} - \mathbf{w}||^2$. In this case, the i-th

Algorithm 3.1. A projected gradient parameter learning algorithm

```
Initialize $\mathbf{w}^0$, set $k \leftarrow 0$.
while $k$ is less than the iteration limit do
    $\mathbf{x}^k \leftarrow \mathbf{w}^k - \alpha_k \nabla g(\mathbf{w}^k)$
    $\mathbf{w}^{k+1} \leftarrow \mathbf{proj}(\mathbf{x}^k)$
    if converged($\mathbf{w}^k$, $\mathbf{w}^{k+1}$) then
        return $\mathbf{w}^k$
    $k \leftarrow k + 1$
return $\mathbf{w}^k$
```

element $[\mathbf{proj}(\mathbf{x})]_i$ is defined as

$$[\mathbf{proj}(\mathbf{x})]_i = \begin{cases} 0 & \text{if } x_i \leq 0 \\ x_i & \text{if } 0 < x_i < 1 \\ 1 & \text{if } x_i \geq 1 \end{cases} , \tag{3.3}$$

i.e., we simply map x_i to 0 or 1 if it is not in the $0 \leq x_i \leq 1$ range.

α_k is the step size used for the k-th iteration. Setting an appropriate α_k is important since it determines the convergence of the projected gradient algorithm. We set α^k by using a simple line search based procedure called *the Armijo rule along the projection arc*, as described in [5, 4]. It defines α_k as $\alpha_k = \beta^{t_k}$, where $\beta \in (0, 1)$, and t_k is the first non-negative integer t that satisfies

$$g(\mathbf{w}^k) - g(\mathbf{w}^k(\beta^{t_k})) \leq \sigma \nabla g(\mathbf{w}^k)^T (\mathbf{w}^k - \mathbf{w}^k(\beta^{t_k})) , \tag{3.4}$$

where $\mathbf{w}^k(\beta^{t_k})$ is $\mathbf{proj}(\mathbf{w}^k - \beta^{t_k} \nabla g(\mathbf{w}^k))$, and σ is a parameter that satisfies $\sigma \in (0, 1)$. In the experiments, we use parameters $\sigma = 0.2$ and $\beta = 0.5$, as these values are suggested in the textbook [5]. By using α_k selected with this rule, it is proved that the projected gradient algorithm converges to a stationary point after several iterations[5, 11] even if the objective function is nonconvex. Hence Alg. 3.1 converges after finite numbers of iterations.

3.3.4 Computation of gradient

To compute gradient $\nabla g(\mathcal{T}(\mathbf{w}), \mathcal{D})$, we must compute $\nabla \ell(\mathcal{T}(\mathbf{w}), \mathcal{D})$ and $\nabla h(\mathbf{w})$. Let $[\nabla \ell(\mathcal{T}(\mathbf{w}), \mathcal{D})]_i$ be the i-th element of the gradient $\nabla \ell(\mathcal{T}(\mathbf{w}), \mathcal{D})$, then it becomes

$$[\nabla \ell(\mathcal{T}(\mathbf{w}), \mathcal{D})]_i = -\sum_{j=1}^{N} \frac{\sum_{L \in \mathcal{L}_j} [\nabla P(L|\mathcal{T}(\mathbf{w}))]_i}{P(I_j|\mathcal{T}(\mathbf{w}))} . \tag{3.5}$$

Here we use $\mathcal{L}_j$ as the set that contains all possible assignments $L \subseteq L_{\mathcal{T}}$ that are consistent with I_j. We define $\nabla P(L|\mathcal{T}(\mathbf{w}))$ as

$$\begin{aligned}[\nabla P(L|\mathcal{T}(\mathbf{w}))]_i = &\sum_{k=1}^{K_i} (2\delta(f_i\theta_{i,k} \in L) - 1) \\ &\prod_{f_n\theta_{n,k} \in L^{-i,k}} w_n \prod_{f_n\theta_{n,k} \in L_{\mathcal{T}}^{-i,k} \setminus L} (1 - w_n),\end{aligned}$$

where $\delta(f_i\theta_{i,k} \in L) = 1$ if the condition is satisfied, otherwise 0, and $L^{-i,k}$ is $L \setminus \{f_i\theta_{i,k}\}$. $\nabla h(\mathbf{w})$ is easy to compute and $[\nabla h(\mathbf{w})]_i$ becomes

$$[\nabla h(\mathbf{w})]_i = \frac{1}{w_i + \varepsilon} - \frac{1}{1 - w_i + \varepsilon} .$$

The computation of gradient $\nabla \ell(\mathcal{T}(\mathbf{w}), \mathcal{D})$ involves summation over $L \in \mathcal{L}_j$. Since $\mathcal{L}_j$ consists of all the possible combinations of ground probabilistic facts $L_{\mathcal{T}}$, its size becomes $2^{L_{\mathcal{T}}}$ in the worst case and the naive computation of gradients is intractable with a large $L_{\mathcal{T}}$. Hence we use the knowledge compilation technique to compute them.

Knowledge compilation [25] is an approach that can be used for efficient computation involving propositional models. It first compiles a propositional model so as to represent it in a form that is suitable for specific operations such as probabilistic inference. Although knowledge compilation incurs the additional cost of compiling a model, once a compiled model is obtained, we can efficiently perform several operations by using it. In previous work, a ProbLog program was compiled into *binary decision diagrams (BDD)* [9], and *a deterministic, decomposable negation normal form (d-DNNF)* [25, 24]. With a ProbLog program compiled into a BDD or a d-DNNF, we can compute the probability $P(I|\mathcal{T}(\mathbf{w}))$ in time proportional to the size of the compiled representation. This computation is also used when employing EM style algorithms [32, 38].

We use compiled knowledge representations for computing the gradients of the negative log-likelihood function $\nabla\ell(\mathcal{T}(\mathbf{w}),\mathcal{D})$. Although both BDD and d-DNNF can be used for computing gradients, we chose d-DNNF as a compiled knowledge representation since it performed well in the previously reported parameter learning algorithms [32].

We briefly introduce d-DNNF. d-DNNF is a kind of negation normal form (NNF) that satisfies decomposability and determinism. An NNF is a rooted directed acyclic graph in which each leaf node is labeled with a literal, true or false, and each internal node is labeled with a conjunction or disjunction. For any node n in an NNF graph, $vars(n)$ denotes all propositional variables that appear in the subgraph rooted at n, and $\Delta(n)$ denotes the formula represented by n and its descendants. We say an NNF satisfies decomposability if $vars(n_i) \cap vars(n_j) = \varnothing$ holds for any two children n_i and n_j $(i \neq j)$ of an and-node n, and NNF satisfies determinism when $\Delta(n_i) \wedge \Delta(n_j)$ is logically inconsistent for any two children n_i and n_j $(i \neq j)$ for an or-node n.

Given the set of interpretations $\mathcal{D}$, the process of compiling a ProbLog program into d-DNNFs is the same as for the parameter learning method proposed in [32]: We first make a ground ProbLog program and convert it into a conjunctive normal form (CNF), and then convert it into d-DNNF with a d-DNNF compiler that converts a CNF into the corresponding d-DNNF. After obtaining N different d-DNNFs that correspond to each interpretation $I_1,\ldots,I_N$, we use them to compute the probability $P(I_j|\mathcal{T}(\mathbf{w}))$ and $\nabla P(L,\mathcal{T}(\mathbf{w}))$, which is the numerator of (3.5). We use the inference algorithm shown in [24], which was originally used for computing conditional probabilities and gradients for graphical models. These algorithms can compute $P(I_j|\mathcal{T}(\mathbf{w}))$ by traversing all the nodes of a d-DNNF once, and can compute $\nabla P(L|\mathcal{T}(\mathbf{w}))$ by traversing all the nodes twice. As a result, we can efficiently compute $\nabla g(\mathcal{T}(\mathbf{w}), \mathcal{D})$.

3.4 Discussion

Our sparse parameter learning algorithm can be applied to other PLP models whose probabilistic parameters take values in $[0, 1]$. For example, PRISM, SLP, and ICL are in this class of PLP models. A major difference between ProbLog and these models is that these models employ the multinomial distribution, while ProbLog programs give probabilistic distribution based on the Bernoulli distribution defined on probabilistic facts.

With these multinomial distribution PLP models, the set of probabilistic parameters can be represented as $\mathbf{w} = \{\mathbf{w}_1, \ldots, \mathbf{w}_N\}$, where $\mathbf{w}_i$ $(i = 1, \ldots, N)$ is a K_i dimensional vector $\mathbf{w}_i = (w_{i1}, \ldots, w_{iK_i})$ and it satisfies $\sum_{j=1}^{K_i} w_{ij} = 1$ and $w_{ij} \geq 0$ for $j = 1, \ldots, K_i$. Here we make an assumption that we can compute the gradient of the negative log-likelihood function $\ell(\mathbf{w}, \mathcal{D})$, given a set of training examples $\mathcal{D}$. Then we apply our algorithm by only modifying the penalty function $h(\mathbf{w})$ defined in (3.2) as

$$h(\mathbf{w}) = \sum_{i=1}^{N} \sum_{j=1}^{K_i} \log\left(w_{ij} + \varepsilon\right).$$

If $K_i = 2$ for all $i = 1, \ldots, N$, the above definition of $h(\mathbf{w})$ is equivalent to that in (3.2). It is also easy to compute $\nabla h(\mathbf{w})$.

We also need to modify the projection function $\mathbf{proj}(\mathbf{x})$ to perform our projected gradient algorithm. We have to project a K_i dimensional real value vector $\mathbf{x}_i$ onto a probability simplex, i.e., project onto a K_i dimensional vector $\mathbf{w}_i$ that satisfies $\sum_{j=1}^{K_i} w_{ij} = 1$ and $w_{ij} \geq 0$ for all $j = 1, \ldots, K_i$. This type of projection also can be efficiently performed [68].

3.5 Evaluation

3.5.1 Settings

We conducted experiments to evaluate our proposed learning algorithm. Our aim was to answer the following questions:

1. (**Q1**) Can we learn a compressed ProbLog program with our proposed algorithm?

2. (**Q2**) How do the estimated model changes when we change the parameter λ ?

3. (**Q3**) Can the proposed algorithm recover true distributions with a sufficient number of training examples?

We compare our proposed method with the LFI-ProbLog algorithm, which is an EM-style algorithm for estimating parameters from interpretations [38]. We also compared our algorithm with a projected gradient algorithm, which simply minimizes the negative log-likelihood function.

Since our algorithm has much in common with the LFI-ProbLog algorithm, we implemented our algorithm on ProbLog2 [71], a ProbLog implementation that supports several inference methods and parameter learning algorithms. We use grounding, CNF conversion, and a d-DNNF compilation algorithm implemented in ProbLog2, and run our projected gradient algorithm on a compiled d-DNNF to estimate the parameters. We also use the LFI-ProbLog algorithm implemented in ProbLog2.

We use two datasets, WebKB and Smokers for evaluating our parameter learning algorithm. The WebKB dataset[1] [21] is a real dataset that consists of labeled Web pages from the computer science departments of four universities. Every web page is marked with one of the following categories, *student*, *faculty*, *project*,

[1] `http://www.cs.cmu.edu/~webkb/`

course, *staff*, and *other*. The task is to predict the classes of pages given the words contained in each page and the link structure between pages. Following the setting used in [32], we use the following ProbLog program.

```
p::link_class(P,P2,c1,c2) :- links_to(P,P2).
p::word_class(P,w1,c1) :- has_word(P,w1).
p::learnable_prior(P,c1) :- page(P).
0.001::fixed_prior(P,c1):-page(P),class(c1).
has_class(P,C) :- word_class(P,W,C).
has_class(P,C) :- has_class(P2,C2),
                  link_class(P,P2,C,C2).
has_class(P,C) :- fixed_prior(P,C).
has_class(P,C) :- learnable_prior(P,C).
```

Where the probabilistic fact `link_class/4` represents the effect of the link structure, and `word_class/3` represents the effect of words contained in a page. We also added probabilistic facts `learnable_prior/2` to represent the probabilistic distribution on labels that are independent with the link structure and words. Finally we add `fixed_prior/2` for avoiding log-likelihood to become infinity in the test data. For computational reasons, we selected 20 words that show the highest information gain with the class labels. We therefore have in total $6 \times 20 + 6 \times 6 + 6 = 162$ probabilistic parameters to be estimated from the data. We conduct four-fold cross validation by using the dataset for three universities as a training set and use the other university as the test set.

The Smokers dataset [30] represents the relationships between people, and contains the following probabilistic facts and rules.

```
0.2::stress(P) :- person(P).
0.3::influences(P1,P2) :- friend(P1,P2).
0.1::cancer_spont(P) :- person(P).
0.3::cancer_smoke(P) :- person(P).
```

Table 3.1: Negative Log-Likelihood (lower is better) and the number of probabilistic parameters contained on the WebKB learning experiment.

Method	NLL	Num. params
LFI	1387.28	39.0
PG	1299.30	38.0
PG+P (λ = 0.001)	1318.96	25.0
PG+P (λ = 0.01)	1445.37	18.0

```
smokes(P) :- stress(P).
smokes(P) :- smokes(P2), influences(P2, P).
cancer(P) :- cancer_spont(P).
cancer(P) :- smokes(P), cancer_smoke(P).
```

In addition to the above program, we add some ground facts person/1 and friend/2 into the program. We add them by first deciding the number of people in the domain, and then randomly deciding friend relationships between people. We set the number of people to 4.

3.5.2 Results

Table 3.1 shows the average negative log-likelihood and the number of parameters for WebKB dataset. Here we use LFI, PG, PG+P to represent the results of LFI-ProbLog, projected gradient algorithm, and projected gradient algorithm with penalty term (the proposed method), respectively. We can see that PG+P shows comparable performance comparing with the state-of-the-art method LFI when $\lambda = 0.001$, while it can reduce the average number of parameters contained in the learned model from 39 to 25. Following these results, we can answer **Q1** that the proposed method shows the inference performance that is comparable with the state-of-the-art algorithm, while it can reduce the number of parameters.

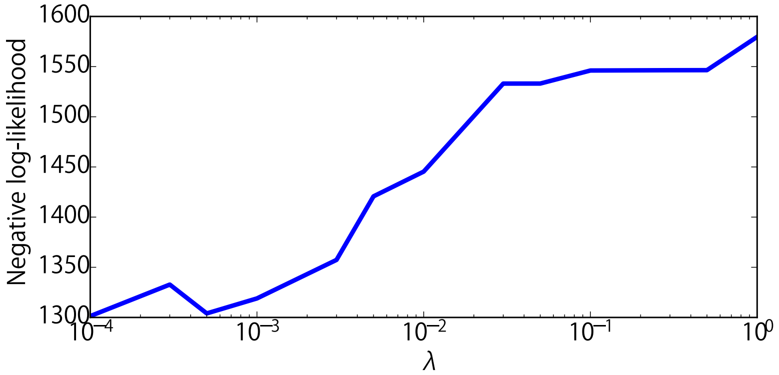


Figure 3.2: Negative log-likelihood for different λ.

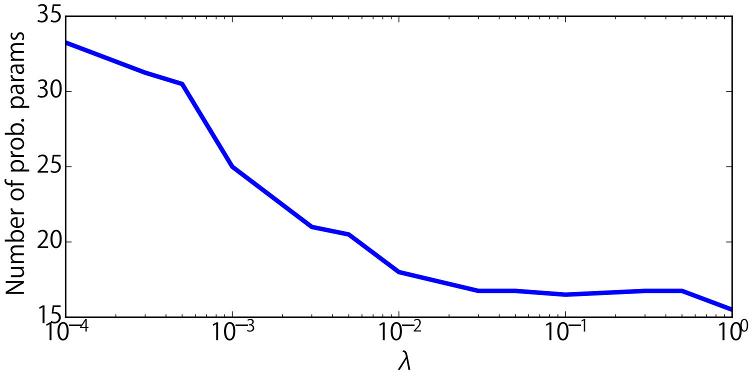


Figure 3.3: The number of estimated probabilistic parameters for different λ.

For answering **Q2**, we show the results of the proposed algorithm while changing the parameter λ in Fig. 3.2 and Fig. 3.3. From the result of Fig. 3.3, we can see that the number of parameters contained in the learned program monotonically decreases when we use a large λ. This result reflects that the parameters tend to take 0 or 1 if we add more weight to the penalty term. Figure 3.2 shows that the performance decreases as we use a large λ. This result suggests the performance may decrease if we penalize too much.

To answer the **Q3**, we measured the KL divergence between the true prob-

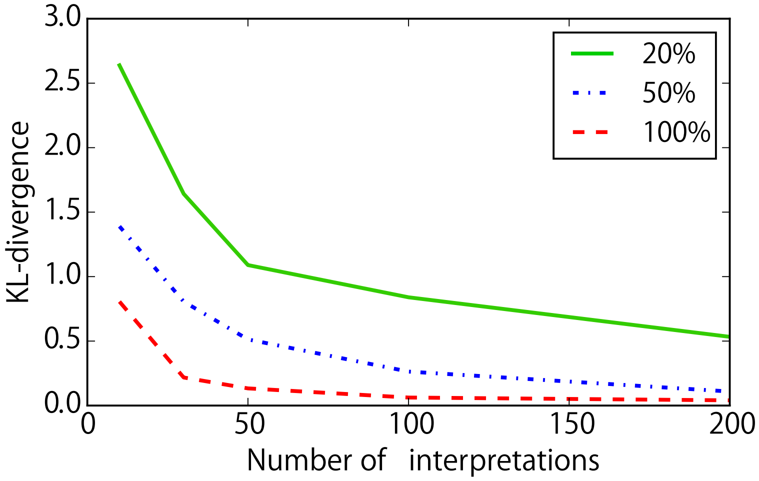


Figure 3.4: KL divergence on Smokers dataset with different numbers of evidences.

abilistic distribution and the distribution estimated by the proposed algorithm. We first make 10, 20, 50, 100, 200 different interpretations on `smokes/1` and `cancer/1` atoms of Smokers dataset, and then sample $20\%, 50\%$, and 100% of them to make training data. We evaluated the KL divergence between the true probabilistic distribution on `smokes/1` and `cancer/1` atoms and the distribution estimated from the data. Figure 3.4 shows the results. We can see that KL divergence decreases as the number of interpretations increases. We therefore can say that the proposed algorithm can estimate the true probabilistic distribution with a sufficient amount of training data (**Q3**).

3.6 Related Work

Many PLP models, such as ProbLog [28], PRISM [75], SLP [62], ICL [69], and parameter learning algorithms for these models have been proposed. Cussens proposed a parameter learning algorithm for SLP [22], Sato [75] proposed an EM learning algorithm for PRISM, and Gutmann et al. proposed two parameter learning algorithms for ProbLog [38, 37]. These algorithms exploit EM-learning or gradient descent methods to optimize an objective function for estimating parameters. Our proposed method differs in that we add penalty terms to induce parameters to take a zero or one probability. This feature resembles the sparse learning algorithms [1] used in many machine learning problems, but we believe that our work is the first to apply a sparse learning method to a parameter learning problem for PLP models. ℓ_1-regularization is used in structure learning for Markov Logic Networks [40], however, the algorithm cannot be directly applied to PLP models like ProbLog.

Our work is also similar to probabilistic theory compression [27] and the theorem revision methods [91] in that it tries to compress a theory into a more concise form. Our algorithm differs in that it simultaneously removes probabilistic facts and infers parameters in one operation.

Our algorithm can be seen as a kind of structure learning algorithm for PLP models, since it outputs a new PLP model given a prototype program and training examples. A previously reported structure learning algorithm for a PLP program is a beam search based algorithm, and it makes it necessary to solve the EM style parameter learning algorithm many times [3]. Obviously we must conduct empirical comparisons, but we believe our algorithm can be more efficient than the previous structure leaning approach since it can find a program by just performing projected gradient based optimization.

3.7 Chapter Summary

We proposed a novel parameter learning algorithm for PLP models that attempts to set the learned parameters so that they take either 0 or 1 by adding a penalty term to an optimization problem. With our algorithm, the learned ProbLog program will have fewer probabilistic components, and inference tasks performed with it become easier. We solved the optimization algorithm by combining the projected gradient algorithm and the computation of gradients in a d-DNNF based knowledge representation.

Chapter 4

Accelerating Graph Adjacency Matrix Multiplications with Adjacency Forest

In this chapter, we propose a data structure for accelerating repeated matrix multiplications performed with a graph adjacency matrix. This procedure appears in many ML applications, and fast algorithm is in strong demand. Our new data structure, *the adjacency forest*, use inherent discrete structure of an graph adjacency matrix, and represent an adjacency matrix as a set of rooted trees. This is also a kind of decomposition technique for exploiting discrete structure. Most part of the contents in this chapter comes from [66].

4.1 Adjacency Matrix Multiplications in Data Analysis

Many data analysis and data mining algorithms need to cope with matrices, and matrix multiplications with an adjacency matrix appear in many popular methods.

Here we define an adjacency matrix as a sparse matrix that represents a graph or relational data (e.g., term document matrix). For example, the computation of PageRank [67] and Personalized PageRank [39, 41] requires multiplications to be performed iteratively with an adjacency matrix, and non-negative matrix factorization algorithms [47, 51, 52] also require repeated multiplications with an adjacency matrix. Since a matrix multiplication needs time proportional to its size, or the number of nonzero elements if it is sparse, iterative matrix multiplications may occupy a large part of the entire computation of these algorithms when the matrix is large.

In this chapter, we propose a specialized method for accelerating repeated matrix multiplications with an adjacency matrix. Here we make two assumptions, namely that an adjacency matrix is sparse and has the *column-scaled nonzeros property*. We say a matrix has the column-scaled nonzeros property if the elements in a column are either zero or have some constant value unique to that column. Many important matrices such as binary matrices and graph random walk matrices, are included in this class of matrix. Our method uses a new data structure, *adjacency forest*, to represent an adjacency matrix. Adjacency forest is an extension of the adjacency list based sparse matrix representation that is made by converting an adjacency list so that it can share equivalent nodes. By sharing nodes, it can reduce the number of scalar operations needed for matrix multiplications. Our method is very simple and easy to implement, but its effect is promising. The following is the main virtues of our method.

Fast and exact computation Our method can reduce the number of scalar operations needed for matrix multiplications since it can share their equivalent intermediate results. Note that our method is not an approximation, i.e., using our method does not change the computational results. We show experimentally that our method can compute a matrix multiplication up to 300% faster than with a standard sparse matrix representation.

Guarantees the worst case The proposed method achieves the fast computation of matrix multiplications by compressing a matrix. How well the compression works depends on the target matrix, however, it is guaranteed that in the worst case the number of scalar additions and multiplications needed for the proposed method is no more than that when using the adjacency list based sparse matrix representation.

Negligible overhead Our method first needs to convert a matrix into an adjacency forest. We show that the time needed for this process is negligible in typical knowledge discovery situations where several matrix multiplications are performed. We also propose a prepossessing method for further reducing the size of an adjacency forest. The time needed for this prepossessing method is also negligible.

In Section 4.2, we first briefly introduce PPR and NMF as examples of widely used algorithms that demand iterative matrix multiplications. We then show the main idea of the adjacency forest along with the multiplication algorithm in Section 4.3. We show a prepossessing stage designed to reduce the size of an adjacency forest in Section 4.4. We describe the experiment and its results in Section 4.5.2 and related work in Section 4.6. Finally, we present our conclusions in Section 4.7.

4.2 Motivating Use Cases

We first show the motivating use cases where iterative multiplications are performed with an adjacency matrix.

4.2.1 Personalized PageRank

The first motivating example of repeated matrix multiplication is personalized PageRank (PPR) [39, 41]. PPR is a popular algorithm that is widely used in several information retrieval and recommendation tasks. Let $\mathbf{A}$ be the adjacency matrix of a directed graph. $\mathbf{A}_{ij} \neq 0$ if there is a link from the i-th node to the j-th node, otherwise $\mathbf{A}_{ij} = 0$. We assume $\mathbf{A}_{ij}$ has the row-scaled nonzeros property, i.e., it can be represented as $\mathbf{A} = \mathbf{BC}$, where $\mathbf{B}$ is a diagonal matrix and $\mathbf{C}$ is a binary matrix. If $\mathbf{A}$ has the row-scaled nonzeros property, then transposed matrix $\mathbf{A}^{\mathrm{T}}$ has the column-scaled non-zeros property. Personalized PageRank is a probabilistic distribution vector $\mathbf{x}$ that is defined on nodes of a graph that satisfies the following equation:

$$\mathbf{x} = (1 - \theta)\mathbf{A}^{\mathrm{T}}\mathbf{x} + \theta\mathbf{u}\,,$$

where $\theta \in (0, 1)$ is the teleportation constant. $\mathbf{u}$ is called a preference vector and is an N-dimensional vector that defines the preference (i.e., initial weights set on nodes) on each node. We can obtain $\mathbf{x}$ by repeatedly updating $\mathbf{x}$ as $\mathbf{x} \leftarrow (1 - \theta)\mathbf{A}^{\mathrm{T}}\mathbf{x} + \theta\mathbf{u}$. This means we need to repeat the multiplication between $\mathbf{A}^{\mathrm{T}}$ and $\mathbf{x}$.

If we want to obtain R different PPR vectors, the personalized PageRank equation for R different personalized vectors, and the power iteration with these R personalized vectors can be written as

$$\mathbf{X} \leftarrow (1 - \theta)\mathbf{A}^{\mathrm{T}}\mathbf{X} + \theta\mathbf{U}\,,$$

where $\mathbf{X}$ and $\mathbf{U}$ are the $N \times R$ matrices that represent sets of R PPR vectors and preference vectors, respectively. To update $\mathbf{X}$, we need to perform a matrix multiplication between $\mathbf{A}^{\mathrm{T}}$ and current $\mathbf{X}$. Let $\mathrm{nnz}(\mathbf{A})$ is the number of nonzero elements of $\mathbf{A}$. If we use a standard method of matrix multiplication with a sparse and a dense matrix, we need $\mathrm{nnz}(\mathbf{A}) \times R$ scalar additions and multiplications. Therefore matrix multiplications occupy a large part of the total time needed for computing PPR.

4.2.2 Non-negative Matrix Factorization

Non-negative matrix factorization is a group of algorithms whose purpose is to factorize a non-negative data matrix into two non-negative matrices. NMF is frequently applied to binary matrices such as term-document matrices. Given a non-negative $N \times M$ data matrix $\mathbf{A}$, NMF finds $\mathbf{W}$ and $\mathbf{H}$ such that $\mathbf{A} \approx \mathbf{WH}$, where $\mathbf{W}$ is an $N \times R$ non-negative matrix, and $\mathbf{H}$ is an $R \times M$ non-negative matrix, and $\approx$ indicates an approximation. Under an appropriate metric, the problem is solved as an optimization problem that involves finding $\mathbf{W}$ and $\mathbf{H}$ with which the distance between $\mathbf{A}$ and $\mathbf{WH}$ is minimized. Seung and Lee [51] proposed one of the most famous and simple algorithms under the Euclid distance. It iteratively updates $\mathbf{W}$ and $\mathbf{H}$ as

$$\mathbf{W}_{ij} \leftarrow \mathbf{W}_{ij} \frac{(\mathbf{A}^{\mathrm{T}}\mathbf{H})_{ij}}{(\mathbf{WHH}^{\mathrm{T}})_{ij}}, \ \mathbf{H}_{ij} \leftarrow \mathbf{H}_{ij} \frac{(\mathbf{W}^{\mathrm{T}}\mathbf{A})_{ij}}{(\mathbf{W}^{\mathrm{T}}\mathbf{WH})_{ij}},$$

to find a solution. Since NMF is used for approximating $\mathbf{A}$ with low-rank matrices $\mathbf{W}$ and $\mathbf{H}$, it is usually used with a small R compared with N and M. With this setting, the computational time needed for multiplications $\mathbf{W}^{\mathrm{T}}\mathbf{A}$ and $\mathbf{A}^{\mathrm{T}}\mathbf{H}$ occupies a large portion of an iteration. Several efficient algorithms have been proposed for NMF in addition to the above iterative algorithm [47, 52], however, these algorithms also require repeated matrix multiplication with $\mathbf{A}$. For example, Kim and Park [52] proposed the projected gradient method that computes the gradient of $\mathbf{W}$ and $\mathbf{H}$ for each iteration. Their method also requires the matrix multiplications of $\mathbf{W}^{\mathrm{T}}\mathbf{A}$ for computing gradients.

4.3 Adjacency Forest

We introduce our new data structure, *adjacency forest*. For ease of presentation, we first describe *a single tree adjacency forest (STAF)*, a very simple and limited version of adjacency forest. After that we introduce a general adjacency forest.

Here we introduce a few notations used in this section. Let $r_i(\mathbf{A})$ be the i-th row vector of $\mathbf{A}$, and $c_j(\mathbf{A})$ be the j-th column vector of $\mathbf{A}$. We also use r_i and c_j instead of $r_i(\mathbf{A})$ and $c_j(\mathbf{A})$ if it is apparent from the context.

4.3.1 Single Tree Adjacency Forest

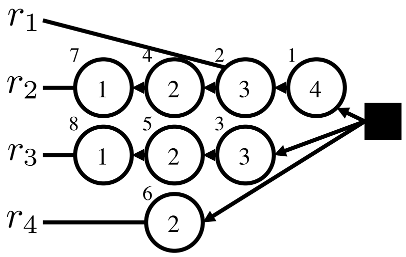
The STAF is an extension of the adjacency list based representation of a matrix. An adjacency list based representation of a matrix represents a matrix as a set of row vectors, and each row vector is represented as a list that stores nonzero elements contained in that row. Each element of the list consists of a pair consisting of the column number and its value. Hence an $N \times M$ matrix $\mathbf{A}$ is represented by N lists, and the sum of the size of all lists is $\mathrm{nnz}(\mathbf{A})$. While an adjacency list represents a matrix as a set of lists, a STAF represents a matrix as a tree, which is made by sharing the equivalent suffixes of the row vectors. For example, the matrix in Fig. 4.1 (a) is represented by the STAF in Fig. 4.1 (b). The rightmost rectangle node of this STAF is a *root node*, and circle nodes are *intermediate nodes*, whose label represents the corresponding column number of the adjacency matrix, e.g., a node with label 1 corresponds to the first column of the matrix. The leftmost nodes with r_i labels are *leaf nodes*, and they represent the corresponding rows. From the definition of the STAF, a path from a leaf node to the root node corresponds to a row vector. For example, the path from leaf node r_1 to the root node is r_1 – 3 – 4 – $root$ and it corresponds to the first row vector of the matrix in Fig. 4.1 (a)

In adjacency list based representation, the matrix in Fig. 4.1 (b) is represented by the following four lists: $r_1 = \langle 3, 4 \rangle$, $r_2 = \langle 1, 2, 3, 4 \rangle$, $r_3 = \langle 1, 2, 3 \rangle$, and $r_4 = \langle 2 \rangle$. In contrast, with the STAF in Fig. 4.1 (b), since r_1 and r_2, have a common suffix $\langle 3, 4 \rangle$, they are shared. Node sharing means that the number of intermediate nodes of a STAF in Fig. 4.1 becomes 8 and is less than $\mathrm{nnz}(\mathbf{A}) = 10$.

Algorithm 4.1 shows the process for constructing a STAF. Since we can see it

$$\begin{pmatrix} 0 & 0 & 0.33 & 0.5 \\ 0.5 & 0.33 & 0.33 & 0.5 \\ 0.5 & 0.33 & 0.33 & 0 \\ 0 & 0.33 & 0 & 0 \end{pmatrix}$$

(a)



(b)

Figure 4.1: (a) Example matrix, and (b) a STAF that represents the matrix.

Algorithm 4.1. Constructing a STAF.

```
Input: A $N \times M$ adjacency matrix $\mathbf{A}$
Output: STAF representing $\mathbf{A}$
Make an empty trie.
Insert every row sequence of $\mathbf{A}$ into the trie.
```

as a trie that stores all of the reversed row vectors, trie construction algorithms can be used to construct a STAF. It takes $O(\mathrm{nnz}(\mathbf{A}) + N)$ time, which is proportional to the sum of the lengths of the all sequences inserted into a trie.

A STAF has the following important property:

property 4.1. *If a matrix* $\mathbf{A}$*, which has column-scaled nonzeros property, is represented with a STAF, the number of intermediate nodes is always equal to or less than* $\mathrm{nnz}(\mathbf{A})$.

This property is obvious from the definition of a STAF. If row vectors have no equivalent suffixes, the number of intermediate nodes of the STAF is $\mathrm{nnz}(\mathbf{A})$ [1] .

[1] Even if a matrix does not have the column-scaled nonzeros property, we can represent a matrix as a STAF whose number of nodes is equal to or less than $\mathrm{nnz}(\mathbf{A})$, and use it for matrix multiplications. However, its compression effect would be limited since STAF can share fewer substructures.

Algorithm 4.2. Matrix multiplication with a STAF.

```
Input: A STAF of an $N \times M$ matrix $\mathbf{A}$, an $M \times R$ real value matrix $\mathbf{X}$
Output: The $N \times R$ matrix $\mathbf{Y}$ such that $\mathbf{Y} = \mathbf{AX}$
1: Prepare length $R$ vectors $l_i$ for each intermediate node.
2: Traverse every intermediate node in a root-to-leaf order and compute $l_i$ as
       $l_i = l_{\mathrm{pa}(i)} + v_{\mathrm{col}(i)} r_{\mathrm{col}(i)}(\mathbf{X})$
3: For every leaf node, set $r_j(\mathbf{Y})$ as the vector of its parent node ($1 \leq j \leq N$).
   Returns $\mathbf{Y}$.
```

4.3.2 Matrix Multiplication with a STAF

With a STAF, we can perform a matrix multiplication with a number of scalar operations that is proportional to the number of nodes. We show the procedure in Algorithm 4.2. To perform a matrix multiplication between an $N \times M$ adjacency matrix $\mathbf{A}$ and an $M \times R$ real value matrix $\mathbf{X}$, we prepare additional memory space for storing an R dimensional real value vector for each intermediate node. Let l_i be such a vector attached to the i-th intermediate node. After preparing the vectors (line 1), the algorithm updates each l_i in a root-to-leaf order following the update equation of line 2. Here $\mathrm{pa}(i)$ is the order of the i-th node's parent node, and $\mathrm{col}(i)$ is the column number corresponding to the i-th intermediate node, and v_j is the constant value of the j-th column. For example, the constant values of the matrix in Fig. 4.1 (a) are $v_1 = 0.5$, $v_2 = 0.333$, $v_3 = 0.333$, and $v_4 = 0.5$.

Example 4.2. *Consider a matrix multiplication with the STAF of the matrix in*

Fig. 4.1 (a), and a 4×2 *matrix*

$$\begin{pmatrix} 0.1 & 0.5 \\ 0.2 & 0.15 \\ 0.3 & 0.15 \\ 0.4 & 0.2 \end{pmatrix}.$$

We use the order of nodes written at the upper left of the intermediate nodes in Fig. 4.1. This order satisfies the condition that an intermediate node always comes after its parent node. We therefore traverse intermediate nodes in this order to perform the update of l_i *in a root-to-leaf order. For example, the computation for the leaf node* r_1 *using this order requires the following computations:*

$$\begin{aligned} l_1 &= 0.5 \times (0.4, 0.2)^{\mathrm{T}} = (0.2, 0.1)^{\mathrm{T}} \\ l_2 &= l_1 + 0.3333 \times (0.3, 0.15)^{\mathrm{T}} = (0.3, 0.15)^{\mathrm{T}} \\ r_1(\mathbf{Y}) &= l_2 = (0.3, 0.15)^{\mathrm{T}}. \end{aligned}$$

Similarly, to compute $r_2(\mathbf{Y})$*, we need to compute* l_1*,* l_2*,* l_4*, and* l_7 *on this order. But* l_1 *and* l_2 *have already been computed by the computation for* $r_1(\mathbf{Y})$ *and we can re-use them to avoid computing* l_1 *and* l_2*.*

To understand why Algorithm 4.1 works, it might be helpful to consider the matrix multiplication of a $1 \times M$ matrix $\mathbf{A}'$ and an $M \times R$ matrix $\mathbf{X}$. The STAF of $\mathbf{A}'$ is a tree that has only one path, and is equivalent to the adjacency list representation. The multiplication is performed in the following way: (i) select row vectors of $\mathbf{X}$ whose row numbers are equal the column numbers of $\mathbf{A}'$ whose elements are nonzero, (ii) multiply each row vector with the element of $\mathbf{A}'$ whose column number equals to the row number of the vector, and (iii) take the summation of the vectors. Algorithm 4.2 corresponds to perform this procedure for each row vector.

We analyze the number of scalar additions and multiplications of Algorithm 4.2.

Lemma 4.3. *To perform a matrix multiplication with an $N \times M$ adjacency matrix $\mathbf{A}$ and $M \times R$ matrix $\mathbf{X}$, Algorithm 4.2 requires to perform KR scalar additions and MR scalar multiplications, where K is the number of intermediate nodes whose parent node is not the root.*

Proof. Because $v_j r_j(\mathbf{X})$ $(1 \leq j \leq M)$ is used multiple times, we can save a maximum of MR multiplications by storing them. As for the number of vector additions, since one addition of two R dimensional vectors is performed for every intermediate node except the nodes whose parent is the root node. In total KR scalar additions are performed. □

This approach should be contrasted with an adjacency list based representation, or other standard sparse matrix representations such as compressed raw storage (CRS) [2], which have to perform $\mathrm{nnz}(\mathbf{A}) \times R$ scalar additions and MR scalar multiplications. From property 4.1, the number of intermediate nodes of a STAF does net exceed $\mathrm{nnz}(\mathbf{A})$, and so we can reduce the number of scalar additions by using STAF.

4.3.3 Properties of the Matrix Multiplication Algorithm

Memory Usage To perform a matrix multiplication with a STAF, we need to prepare memory for storing a STAF, vectors l_i and v_j. A STAF can be represented as a set of nodes, and a intermediate node can be represented as a pair of the corresponding column number and a pointer that indicates the parent of the node. The set of leaf nodes can be represented as an N-dimensional array that each element stores a pointer that indicates the corresponding intermediate node. CRS can represent a matrix with column-scaled nonzeros property as a combination of an array that stores column numbers of each nonzero element, and an array that stores pointers for each row. Comparing with CRS, a STAF requires more than twice memory to represent one nonzero element.

The amount of memory required for storing l_i corresponds to (the number of intermediate nodes) $\times R$ real numbers if we naively implement Algorithm 4.2. This can be improved by releasing memory used for l_i that will not be used in future computations. Since l_i is not used again after the values of its child nodes have been computed, we release it to save the maximum amount of memory. With this strategy, we can perform a matrix multiplication while storing a maximum of NR real numbers.

CPU cache hit ratio The CPU cache hit ratio is also important in terms of the performance since a cache miss may result in the delay of hundreds of CPU clocks for modern computers. Compared with standard sparse matrix representation such as CRS, a STAF consists of many pointers and the CPU cache hit ratio may be lower. We examine the adjacency forest performance empirically in experiments.

4.3.4 General Adjacency Forest

We can reduce the number of additions by using a STAF to re-use intermediate computation results. However, the effect may be limited because we can share intermediate results only when the suffixes of row vectors are the same. We therefore extend the STAF to induce more sharing.

As a STAF represents a matrix as a tree, we extend it to represent a matrix as a set of trees. We first divide an $N \times M$ adjacency matrix $\mathbf{A}$ into sub-matrices $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$, …, $\mathbf{A}^{(m)}$, where each $\mathbf{A}^{(i)}$ $(1 \le i \le m)$ is an $N \times M$ matrix that has the column-scaled nonzero property. We form sub-matrices that satisfy $\mathbf{A} = \sum_{k=1}^{m} \mathbf{A}^{(k)}$ and if $\mathbf{A}_{ij} \ne 0$, then for some k', $\mathbf{A}_{ij}^{(k')} = \mathbf{A}_{ij}$ and $\mathbf{A}_{ij}^{(k)} = 0$ for other $k \ne k'$. Then we represent each matrix $\mathbf{A}^{(k)}$ as a STAF, and we concatenate all of the STAFs to make a forest. We call this structure *adjacency forest*. While a STAF can only share the equivalent suffixes of row vectors, an adjacency forest can share equivalent suffixes for each sub-matrix and can represent a matrix in an even smaller form. Figure 4.2 shows an example of an adjacency forest representing the adjacency matrix in Fig. 4.1 (a), where we divide the matrix into two matrices

$$\mathbf{A}^{(1)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0.5 & 0.33 & 0 & 0 \\ 0.5 & 0.33 & 0 & 0 \\ 0 & 0.33 & 0 & 0 \end{pmatrix} \tag{4.1}$$

$$\mathbf{A}^{(2)} = \begin{pmatrix} 0 & 0 & 0.33 & 0.5 \\ 0 & 0 & 0.33 & 0.5 \\ 0 & 0 & 0.33 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{4.2}$$

Here $\mathbf{A}^{(1)}$ consists of the first two columns of $\mathbf{A}$ and $\mathbf{A}^{(2)}$ consists of its last two columns. Dashed edges in Fig. 4.2 are *jumping links*. They connect STAFs and make an adjacency forest.

The process for conversion from a set of STAFs into an adjacency forest has the following three steps. First, set order on the STAFs that correspond to each sub-matrix. Then, we substitute each leaf node of all of the STAFs except the first one with a jumping link that comes from the previous intermediate node that is indicated by the same leaf node. Finally, we duplicate any node that has a jumping link. Figure 4.3 shows two STAFs that represent two sub-matrices. When we compare Fig. 4.3 and Fig. 4.2, we can see that the node with label 1 is duplicated since it has jumping links.

The matrix multiplication algorithm for adjacency forest is also a straightforward extension of that of the STAF (Algorithm 4.2). Algorithm 4.3 is the procedure for matrix multiplication with an adjacency forest. The difference from the previous algorithm is that it copes with jumping links. The number of scalar additions and multiplications is proportional to the number of intermediate nodes plus the number of jumping links. The validity of the algorithm is verified as following. If we select a leaf node r_i and enumerate all the nodes that are reachable from the leaf node by traversing links backward, we can reach all of the intermediate nodes whose labels are columns of nonzero elements contained in that row. Since $r_i(\mathbf{Y})$ is the sum of $v_j r_j(\mathrm{X})$ of the all of column number j that A_{ij} has a nonzero value, it is computed by the algorithm.

Lemma 4.4. *Algorithm 4.3 must perform NR scalar multiplications and*

$$(K + \textit{the number of jumping links}) \times R$$

scalar additions, where K is the number of intermediate nodes whose parent is not a root node.

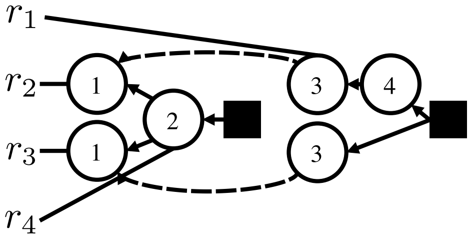


Figure 4.2: Example of adjacency forest representing the matrix in Fig. 4.1 (a).

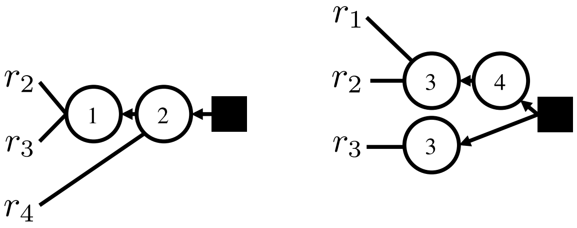


Figure 4.3: Two STAFs that represent sub-matrices (4.1) and (4.2).

Introduction of matrix division, however, not always reduce the number of computations needed for a matrix multiplication. For example, if we divide a matrix into two matrices, the number of intermediate nodes may decrease but we also need to add jumping links. If we add more jumping links than removed intermediate nodes, the number of needed operations will increase. We therefore need to look for appropriate sub-matrices that minimize the total number of operations. We present an algorithm for dividing an adjacency matrix into sub-matrices in the next section.

Algorithm 4.3. Algorithm of multiplications between matrices with adjacency forest.

Input: An adjacency forest of $N \times M$ matrix $\mathbf{A}$, $M \times R$ real value matrix $\mathbf{X}$

Output: $N \times R$ matrix $\mathbf{Y}$ such that $\mathbf{Y} = \mathbf{AX}$

```
Prepare length $R$ vectors $l_i$ for each intermediate node.
Select every tree in right-to-left order.
Traverse every intermediate node of the tree in root-to-leaf order and compute $l_i$ as
    $l_i = l_{\mathrm{pa}(i)} + v_{\mathrm{col}(i)} r_{\mathrm{col}(i)}(\mathbf{X})$
If the current node is indicated by a jumping link, then add the value of the source node of the link to $l_i$.
For every leaf node, set $r_j(\mathbf{Y})$ the value of its parent node. Returns $\mathbf{Y}$.
```

Memory Usage As with a STAF, we can reduce the memory required for computation.

Lemma 4.5. *The additional memory needed for performing a matrix multiplication with an adjacency forest is less than* $2NR$.

Proof. We run algorithms on each trees in a row. To execute an algorithm on a tree, we need NR memory space, which is same as for a STAF. As with a STAF, we need to keep the values of the nodes that are indicated by jumping links. Since the number of nodes indicated by jumping links is at most N for a tree, the total number of nodes is $2NR$. □

The amount of memory required for representing an adjacency forest is larger than that of a STAF that has the same number of intermediate nodes since we have to store jumping links. However, since the number of jumping links is always equal or less than that of the intermediate nodes, the amount of memory needed for storing an intermediate node is less than thrice of that is required for CRS.

4.4 Reduce Size of Adjacency Forest

How much we can reduce the number of calculation relies on the number of intermediate nodes and jumping links of the adjacency forest. These metrics depend on how a matrix is divided into sub-matrices. However, it is difficult to find the matrix division that minimizes the number of operations. We therefore introduce a heuristic that divides a matrix into sub-matrices to realize a small adjacency forest.

Our matrix division algorithm first construct lists of column vectors so that the sub-matrices made from these lists have many similar row vectors while having fewer rows with nonzero elements. The aims of these two conditions are to reduce the numbers of intermediate nodes and the number of jumping links, respectively. We show our algorithm in Algorithm 4.4. We prepare m lists $L_1, \dots, L_m$ (line 1), and then push each column vector $c_i(\mathbf{A})$ into an appropriate list (line 2) that minimizes the score. Finally we convert each list L_j to a sub-matrix $\mathbf{A}^{(j)}$ (line 3) if L_j is not empty.

The most important point in the algorithm is the score function $\mathrm{score}(L_j, c_i(\mathbf{A}))$, which returns the score for appending $c_i(\mathbf{A})$ to list L_j. We define the score function by considering two points (i) how many intermediate nodes will be formed if we add c_i, and (ii) how many new jumping links will be made. With these two elements, we define the score function as

$$\mathrm{score}(L_j, \mathbf{A}_i) = \lambda(\text{NewIMNodes}) + \text{NewRows}\,. \tag{4.3}$$

NewIMNodes is the number of intermediate nodes that will be made if we add c_i to L_j, and NewRows is the number of rows that have nonzero elements in c_i and do not have in any column vectors in L_j. λ is the parameter that adjusts the effect of the two elements. We compute NewIMNodes by creating a STAF for a matrix that is made with the current elements of L_j and c_i. Actually, this can be easily computed by storing the groups of row numbers that have the same row vector in L_j. We compute NewRows by first storing rows with nonzero elements in L_j

Algorithm 4.4. Matrix division algorithm.

```
Input: An $N \times M$ adjacency matrix $\mathbf{A}$, maximum number of sub-matrices $m$.
Output: Sub matrices $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(m)}$
1: Initialize empty lists $L_1, \ldots, L_m$.
2: Traverse every column vector in order, and add $c_i(\mathbf{A})$ to $L_j$ $(1 \leq j \leq m)$ that maximizes the score $\mathrm{score}(L_j, c_i(\mathbf{A}))$.
3: Convert each $L_j$ into $\mathbf{A}^{(j)}$ $(1 \leq j \leq m)$ if $L_j$ is not empty.
```

and comparing it with the nonzero elements of c_i for computation. On converting L_j to $\mathbf{A}^{(j)}$, we simply place the column vectors in an order whereby that the first element becomes the rightmost column of $\mathbf{A}^{(j)}$.

Lemma 4.6. *The running time of Algorithm 4.4 is* $O(\mathrm{nnz}(\mathbf{A}) \times m)$.

Proof. Line 1 of the algorithm takes $O(m)$ time. Line 2 needs to compute $\mathrm{score}(L_j, c_i)$ m times, and to compute $\mathrm{score}(L_j, c_i)$ requires the computation of (i) how many new intermediate nodes will be made, and (ii) how many new rows have nonzero values. Suppose that column vector c_i is a sparse vector, these computations take time proportional to the number of nonzero elements in c_i. Hence the loop of line 2 to 4 takes $O(\mathrm{nnz}(\mathbf{A}) \times m)$ time. Finally, line 4 simply converts lists of column vectors into matrices, which takes $O(\mathrm{nnz}(\mathbf{A}))$ time. Hence Algorithm 4.4 is $O(\mathrm{nnz}(\mathbf{A}) \times m)$. □

4.5 Evaluation

4.5.1 Settings

We conducted experiments to evaluate the proposed algorithm taking account of two typical situations where iterative matrix multiplications are performed. The first situation is the computation of PPR with a square adjacency matrix, and the other is the computation of NMF with a sparse rectangular matrix that corresponds to a word-document association matrix. This class of matrices is used in document clustering, or for the user-item association used in recommender systems. We used two datasets, Amazon[2] and Web[3] , for PPR, and one dataset, news[4] , for NMF. Amazon is a directed graph extracted from Amazon's co-purchased items. Each node represents an item and each direct edge represents a co-purchased relationship. It contains 403,394 nodes and 3,387,388 edges. Web is a directed graph consisting of the web pages of the University of Notre Dame. Each node represents a page and the direct edges represent hyperlinks between them. It contains 325,729 nodes and 1,497,135 edges. news is a dataset consisting of news group data, which forms a rectangular binary matrix of $100 \times 16,242$ with 65,451 nonzero elements. Since the matrices we used for the PPR computations are graph random walk matrices, and the matrix used for MMF is a binary matrix, our adjacency forest based representation is applicable.

For both settings, we conducted iterative matrix multiplications with an adjacency matrix. All of the experiments were implemented in C++ and compiled with gcc 4.7.2 (-O3), and executed on a Linux server with a 3.33GHz Intel Xeon CPU and 48GB RAM. We evaluate the proposed method of using an adjacency forest. As a baseline method, we employed a power iteration method that uses the compressed row storage (CRS) format matrix representation [2], which is a popular sparse matrix format used in many linear algebra libraries. We tried two C++ matrix algebra packages that support sparse matrix representation, Eigen version 3.1.2 and Armadillo version 3.800.2, and we selected Eigen as the baseline since it performs better in our settings. We use the SparseMatrix class of Eigen to represent an adjacency matrix, and then used the multiplication method of the class to compute PPR and NMF. Since both the baseline method and the proposed method

[2] http://snap.stanford.edu/data/amazon0601.html

[3] http://snap.stanford.edu/data/web-NotreDame.html

[4] http://www.cs.nyu.edu/~roweis/data/20news_w100.mat

Table 4.1: Preprocessing time with different m.

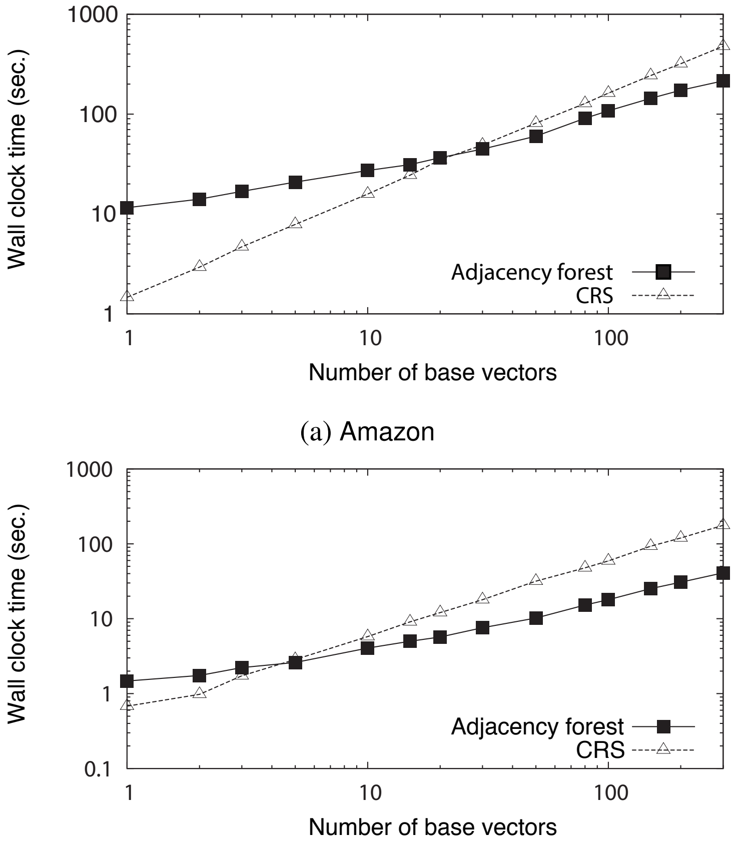
	Matrix Division (Sec.)			Construct(Sec.)		
m	1	10	100	1	10	100
Amazon	0.94	2.60	14.95	0.96	0.74	0.65
Web	0.18	0.81	3.13	0.39	0.15	0.20
News	0.03	0.04	0.16	0.04	0.02	0.02
News (Tr)	0.02	0.05	0.38	0.03	0.03	0.03

returned the same result for these experiments, we fixed the number of iterations at 100 for both implementations. The proposed method used reference counters to minimize the memory consumption. We applied preprocessing algorithm (Algorithm 4.4) to an adjacency forest, and set the number of lists of the matrix division algorithm at m = 10 and parameter λ to λ = 2 as it performs the best for all of the datasets[5] .

4.5.2 Results

Figure 4.4 shows the computation time of PPR on the two datasets when changing the number of base vectors, i.e., the number of columns of the multiplied matrix, and Figure 4.6 shows the computation time needed for NMF while changing the dimension R of factorized matrices. Since when computing NMF, we also need to perform multiplications with the transposed matrix, we show the result with a transposed matrix as News (Tr). Comparing the baseline with the proposed methods shows that the adjacency forest is faster in almost all cases except when the multiplied dense matrix has very few columns. In particular with Web dataset the adjacency forest is more than 300% faster than the baseline. With the Amazon dataset the speed is slightly faster than with the Web dataset. These results can

[5] We tried $\lambda \in [0.5, 3.0]$, and found that the compression ratio was insensitive to changes of λ, i.e., the changes in the compression ratio were around 2%, when $\lambda \geq 2$.



(a) Amazon

(b) Web

Figure 4.4: Comparison of computation times needed for computing PPR.

be related to how well the dataset is compressed. Figure 4.5 shows the compression ratio of an adjacency forest with different parameters m. Here we define the compression ratio of an adjacency forest as

$$\text{compression ratio} = \frac{K + (\text{\# of jumping links })}{\mathrm{nnz}(\mathbf{A})}, \tag{4.4}$$

where K is the number of intermediate nodes whose parent node is not a root node. When $m = 10$, we can see that the compression ratio of the Amazon dataset is high compared with Web, and it means the proposed method is more effective for the Web dataset. The results for the News dataset are similar. The compression

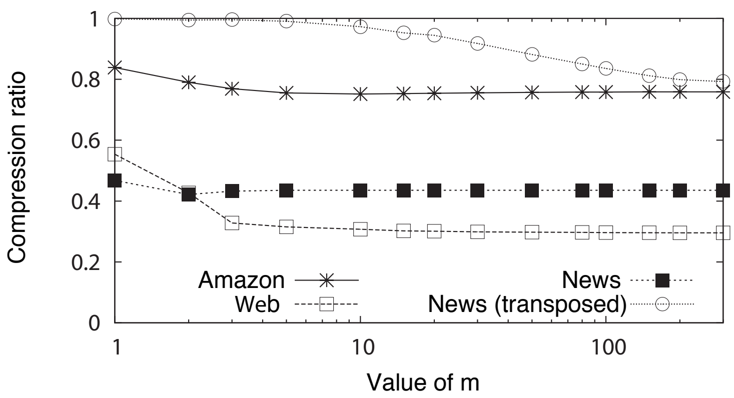
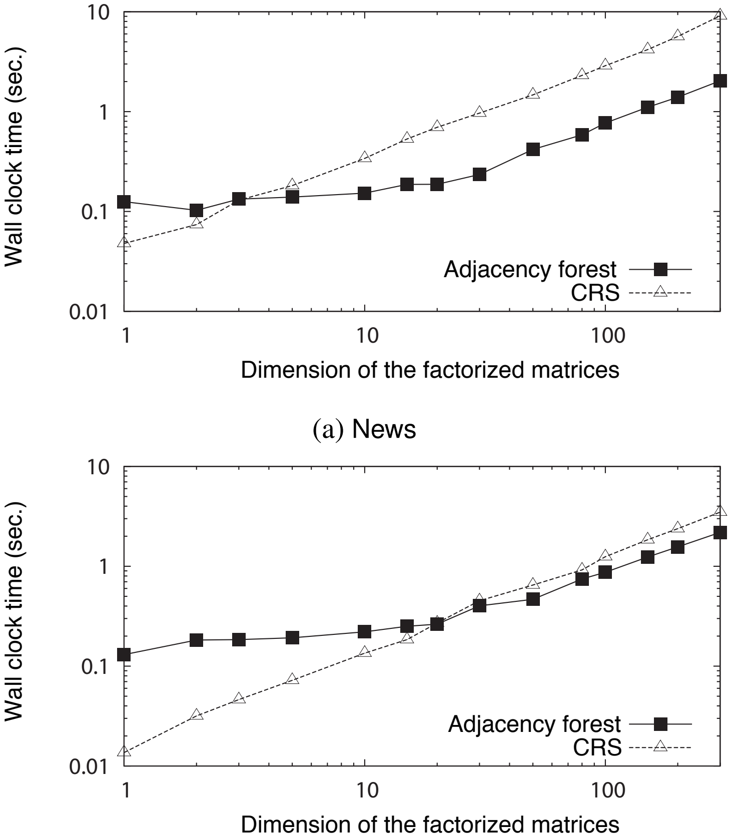


Figure 4.5: Compression ratio with different m.

ratio of News is smaller than News (Tr), and this results in the increased speed in Fig. 4.6. If the number of base vectors is small, our method is slower than the baseline method for all datasets. This is caused by overheads in the reference counting, and cache miss caused by the use of many pointers.

Table 4.1 shows the time needed for preprocessing. Since we used $m = 10$ for our experiments, we can see the proposed algorithm is still faster even when we add the preprocessing time. Moreover, once the adjacency forest is constructed, we can use it several times. PPR can be computed with several different hub vectors, and in such cases, the time taken by preprocessing secures an adequate return.

Figure 4.5 shows how the compression ratio changes with parameter m. We can see that the adjacency forest becomes more compressed as m increases. This means we do not need to make the effort to set an appropriate value for m. As shown in Tab. 4.1, however, a large m makes the time needed for matrix division long. There is a trade off between the preprocessing time and the size of the adjacency forest. Although the optimum m depends on the dataset, $m = 10$ is sufficient for many situations. Figure 4.7 shows the amount of heap memory used with the Web dataset. As we noted in Section 4.3.3, the proposed algorithm needs



(a) News

(b) News (Tr)

Figure 4.6: Comparison of computation times needed for NMF.

a large amount of temporal buffer if it is implemented naively. However, this figure shows that the use of a reference counter enables us to reduce the required memory to less than that needed for the CRS data structure with Eigen.

Since the effectiveness of our proposed method depends on the compression ratio, it is practically important to provide an indicator that shows whether we can predict the compression ratio. After trying several metrics, we found that the Jaccard index of $|A \cap B|/|A \cup B|$ with two sets A and B, used for measuring the similarity of row vectors represented as sets of nonzero elements, can be used as

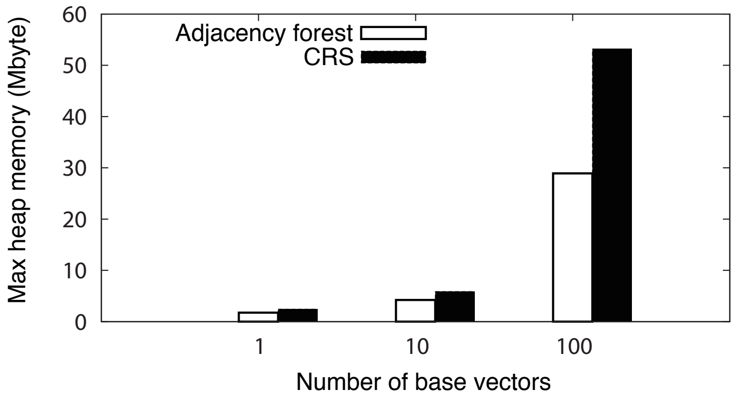


Figure 4.7: Used heap memories.

such an indicator. Figure 4.8 plots the relation between the average of the top 10 Jaccard index of row vectors and the compression ratios of 6 matrices of the datasets and their transportation. We can see that the compression ratio tends to be small if the average Jaccard index is large.

4.6 Related Work

Many fast matrix multiplication algorithms have been proposed. Strassen first proposed the $O(N^{2.81})$ algorithm for the multiplication of two $N \times N$ matrices [77], where a naive algorithm takes $O(N^3)$ time. The fastest algorithm was Coppersmith and Winograd's $O(N^{2.376})$ algorithm [20] for decades, and it was recently updated to $O(N^{2.3727})$ by Williams [85]. Although these cannot exploit the sparsity of matrices, they are efficient if the number of nonzero elements $\mathrm{nnz}(\mathbf{A})$ satisfies $\mathrm{nnz}(\mathbf{A}) > N^{1.38}$. A fast multiplication algorithm for sparse matrices has been proposed by Yuster [90]. It can perform multiplications between two $N \times N$ sparse matrices in $O((\mathrm{nnz}(\mathbf{A}))^{0.7}N^{1.2} + N^{2+o(1)})$ time. However, many knowledge discovery algorithms including PPR and NMF need to multiply a sparse matrix with dense matrices, Yuster's algorithm cannot be directly applied to these

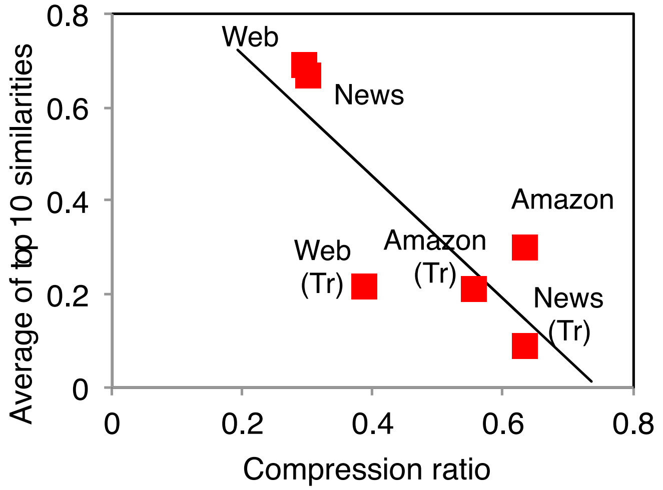


Figure 4.8: Relation between similarities (Jaccard coefficient) of rows and compression ratio.

settings. A simple $O(\mathrm{nnz}(\mathbf{A}) \times N)$ algorithm can be executed by using well-known data structures for sparse matrices, such as the Compressed Row Storage (CRS) format [2]. With the CRS format, a matrix multiplication can be executed by performing additions and multiplications of vectors of length N $\mathrm{nnz}(\mathbf{A})$ times. Our proposed algorithm can further reduce the number of operations by sharing equivalent intermediate results.

Research on graph compression [6, 16] is similar to our work. Graph compression is a technique for compressing large graphs, such as web graphs or social networks, to make them fit into a memory and it can perform certain graph operations without expansion. Boldi and Vigna [6] exploit the locality and similarity of web graphs to make them in a compact form. Karande et al. [45] proposed an algorithm for multiplication with a compressed graph that is made by adding virtual nodes to a graph to reduce the number of edges [10]. Although the interpretation of the proposed method in the context of a graph algorithm is future work, since our method does not impose any assumptions on the structure of a graph, it may be used with virtual node techniques.

From another viewpoint, our proposed method can be seen as a variant of the sparse matrix-vector multiplication (SpMV) methods. While there is a wealth of literature on SpMV to exploit the potential power of hardware including memory bandwidth [83] or multi-cores [84], we have not seen any published work that reduces the number of required multiplications and additions themselves. Since our method does not assume any special hardware architecture, we believe we still have some room to improve the performance by considering the hardware architecture, especially caches.

We dealt with PPR and NMF as example tasks, and many accelerating algorithms have been proposed for both problems. For example, PageRank and PPR can be efficiently computed by using extrapolation [44], or dividing an adjacency matrix into groups [43]. As for NMF, several methods including the projected gradient descent method [52] and the active-set method [47] have been proposed. Since our method is not problem specific, it may be less efficient than these methods. However, the proposed method can be used in combination with these specialized methods that perform matrix multiplications, and can help to speed them up.

The adjacency forest is closely related to a *zero-suppressed binary decision diagram (ZDD)* [60]. ZDD is a data structure that represents a family of sets as a directed acyclic graph. If we represent an adjacency matrix as a set of row vectors each represented as a set of nonzero elements in the row, a ZDD representing this set of row vectors is equivalent to a STAF that represents the matrix.

4.7 Chapter Summary

We proposed a method for accelerating the computation of repeated matrix multiplications with a graph adjacency matrix. This operation appears in many knowledge discovery tasks such as PPR and NMF. By using an adjacency forest, we can reduce the number of multiplications and additions. The additional cost incurred by converting the data structure is small thanks to a new efficient construction and preprocessing method, and our proposals can be easily combined with existing power iteration based algorithms.

Chapter 5

Conclusion

5.1 Summary of the Results

We provided three algorithms for efficiently solving numerical optimization problems that appear in text summarization and relational learning tasks. Our methods exploit the discrete structure inherent to the problem domains, and solve problems by decomposing them in such a way that we can efficiently exploit their inherently discrete structure. This approach has two impacts: first, we can solve complex optimization problems more efficiently and obtain better solutions. Second, we can flexibly design optimization problems for solving a broad range of complex tasks that appear in ML and NLP. Our algorithm can also be applied to accomplishing tasks other than those treated in this thesis.

The combinatorial optimization algorithm used for the text summarization introduced in Chapter 2 was designed to maximize a combined objective function. Such a function measures the quality of a summary as a combination of different multiple objective functions. We mainly used two techniques. The first technique is sub-modular function maximization, and the second is a Lagrangian relaxation. The combination of these two techniques enables an optimization problem to be efficiently solved using a complex objective function. Our experimental results show that our Lagrangian relaxation method generates summarizations with high ROUGE scores compared with state-of-the-art extractive summarization methods. The experiment results also show that our method is more than 30-times faster than a commercial ILP solver.

In Chapter 3, we described a new parameter estimation algorithm for a PLP. PLP theories tend to yield more complicated statistical models compared to other statistical models used for machine learning tasks. Hence, a parameter estimation algorithm that can estimate sparse models, i.e., models with fewer parameters, is preferable. Our proposed algorithm can efficiently estimate such sparse models by combining the knowledge compilation technique with optimization algorithms that work with ℓ_1, such as regularization techniques and the projected gradient method. Our algorithm can exploit the discrete structure inherent to a logic program, and can use a regularization technique to create constraints on the estimated parameters. We experimentally showed that our algorithm can estimate PLP models with a smaller number of parameters compared with a state-of-the-art parameter estimation algorithm that uses an EM style algorithm.

In Chapter 4, we proposed a new data structure called an adjacency forest. An adjacency forest can represent a sparse graph adjacency matrix in a compressed form, and can speed up repeated matrix multiplications using an adjacency matrix. It also exploits the inherently discrete structure of an adjacency matrix, and divides a matrix into a set of trees to compress the equivalent substructure of the matrix. This is also an important application of discrete-structure based decomposition techniques because repeated matrix multiplications with an adjacency forest appear in many relational learning tasks. Our algorithm is up to 300% faster than the use of an existing sparse matrix representation.

5.2 Future Research Directions

Here, we describe some limitations and future research directions for the methods proposed in this thesis. The text summarization algorithm described in Chapter 2 will be applied to other types of summarizations because it is in the development stage in which we can combine arbitrarily different objective functions. If we want to consider different aspects when making a summary, we can easily extend the proposed method by adding new objective functions. If we can easily maximize a newly added objective function itself, our Lagrangian relaxation formulation will also work efficiently.For example, a query-focused summarization [26, 87] is a task of generating a summary that can answer a given query. Our method can be easily extended to a query-focused summarization by adding another objective function that measures the relevance with the given query. Extractive text summarization methods have recently been used in combination with sentence compression techniques [86, 61]. A sentence compression is a technique for shortening input sentences, and we can obtain a considerably concise summary by combining this technique with an extractive summarization algorithm. Our summarization method will also be used for this setting.

A shortcoming of our Lagrangian-relaxation based summarization method introduced in Chapter 2 is that its convergence is not guaranteed. In some Lagrangian-relaxation based algorithms used in NLP tasks, several techniques are used for promoting convergence [13, 14]. In these methods, once-relaxed constraints are added back into the relaxed problem to promote convergence. Adding constraints makes the Lagrangian dual problem difficult to solve, but helps find feasible solutions. Although our method converges at a relatively high rate [1] , applying certain techniques can contribute to a further improvement in performance. For example, the alternating direction method of multipliers (ADMM) is an extension of a

[1] For example, it is reported in [29] that the convergence rate of Lagrangian relaxation based method was only 6% when it is applied to bidirectional word alignment problem.

Lagrangian relaxation method that promotes robustness and fast convergence [8].

The parameter estimation algorithm for the PLP models shown in Chapter 3 has several promising research directions. We used a penalty term to encourage many parameters to take a value of 0 or 1. This penalty term was inspired by ℓ_1 regularization methods. There are many extensions of the ℓ_1 regularization method, and it would be interesting to apply such extensions to our parameter estimation algorithms. For example, the grouping of regularization techniques has been proposed as an extension of the ℓ_1 regularization method [59, 89]. These methods impose sparsity on groups of parameters. This is useful for removing unused feature groups together to promote the sparsity of the estimated models. The grouping of parameters is also suitable for parameter learning of PLP models because a complex logic program usually consists of several sub-components. If we can remove unnecessary components in a component-wise manner, the resulting program may be easy to understand.

Many lifted inference algorithms have recently been proposed for probabilistic logic programs [81, 76, 82]. Lifted inference algorithms can accelerate the probabilistic inference with PLP models by performing inference at a first-order level. Ordinal inference algorithms have to first generate a ground program, and the inference with the ground program takes an amount of time proportional to the size of the program. Because the size of a ground program may become extremely large, the inferences may be considerably slow. Lifted inference algorithms perform only necessary groundings to avoid conducting inferences with a fully grounded model. Applying techniques developed for a lifted inference to a parameter learning task is a promising research direction.

The efficiency of our parameter estimation algorithm depends on the size of d-DNNFs. The size of the d-DNNF depends on the order variables, and finding the optimal order that minimizes the number of nodes is a difficult problem. We used existing d-DNNF compilers [63, 23] that convert a CNF into a d-DNNF. However, it may be possible to make considerably smaller d-DNNFs by exploiting features specific to our problem setting. Both theoretical and practical contributions to the size of a d-DNNF are in high demand.

We illustrated the power of an adjacency forest by applying it to repeated matrix multiplications. We believe that using an adjacency forest itself has great value, but it will be more useful if we can apply it to some other matrix operations such as an inverse matrix computation or singular value decomposition. These matrix operations are also used in many ML and NLP applications. Another possible extension is to apply an adjacency forest to multi-valued adjacency matrices. An adjacency forest can speed up the computations by sharing equivalent sub-structures of the matrices. If an adjacency matrix is multi-valued, the amount of equivalent sub-structures will decrease, and the proposed method cannot considerably increase the speed. Therefore, we have to consider other compression rules that encourage a compact representation.

Similar with the case of d-DNNF, the efficiency of the proposed matrix multiplication algorithm depends on the size of the adjacency forest. As we mentioned in Chapter 4, an adjacency forest can be regarded as a special type of ZDD. The size of a ZDD depends on the order of variables appearing in it, and the problem of finding the optimal order is known to be NP-complete[7]. Therefore, finding better heuristics is a preferable research direction. The heuristic algorithm for creating a small adjacency forest introduced in Chapter 4 was designed for dividing matrices in less time compared with the amount of time required for matrix multiplications. If a compiled adjacency forest is used many times, we can take more time to create a smaller adjacency forest. It is possible that a matrix division algorithm exists that, while requiring a large amount of time, can make the size of the adjacency forest considerably smaller. When making an adjacency forest, we divide an adjacency matrix into sub-matrices whose number of rows is the same as the original matrix. This method simplifies the division algorithm and shows a high performance, but other types of divisions for making an adjacency forest are also applicable. We should consider these relaxed matrix division methods in further detail.

Bibliography

[1] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski. Optimization with sparsity-inducing penalties. *Foundations and Trends in Theoretical Computer Science*, 3(2-3), 2009.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[3] E. Bellodi and F. Riguzzi. Learning the structure of probabilistic logic programs. In *Proc. ILP*, pages 61–75, 2012.

[4] D. Bertsekas. On the goldstein-levitin-polyak gradient projection method. *Automatic Control, IEEE Trans. on*, 21(2):174–184, 1976.

[5] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[6] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. WWW*, pages 595–602, 2004.

[7] B. Bollig and I. Wegener. Improving the variable ordering of obdds is NP-complete. *IEEE Trans. Comput.*, 45(9), 1996.

[8] S. Boyd, N. Parikh, E. Chu, P. B., and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1 – 122, 2011.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, 1986.

[10] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proc. WSDM*, pages 95–106, 2008.

[11] P. Calamai and J. Moré. Projected gradient methods for linearly constrained problems. *Mathematical Programming*, 39(1):93–116, 1987.

[12] J. Carbonell and J. Goldstein. The use of mmr,diversity-based reranking for reordering documents and producing summaries. In *Proc. SIGIR*, 1998.

[13] Y. W. Chang and M. Collins. Exact decoding phrase-based translation model through lagrangian relaxation. In *Proc. EMNLP*, 2011.

[14] Y. W. Chang, A. M. Rush, J. DeNero, and M. Collins. A constrained viterbi relaxation for bidirectional word alignment. In *Proc. ACL*, pages 1481–1490, 2014.

[15] S. S. Chen, D. Donoho, and M. Saunders. Atomic decomposition by basis pursuit. *SIAM journal on scientific computing*, 20(1):33–61, 1998.

[16] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. KDD*, pages 219–228, 2009.

[17] P. Combettes and J. C. Pesquet. Proximal splitting methods in signal processing. In *Fixed-point Algorithms for Inverse Problems in Science and Engineering*, pages 185–212. Springer, 2011.

[18] P. L. Combettes and V. R. Wajs. Signal recovery by proximal forward-backward splitting. *Multiscale Modeling & Simulation*, 4(4):1168–1200, 2005.

[19] J. M. Conroy, J. D. Schlesinger, J. Goldstein, and D. P. O'leary. Left-brain/right-brain multi-document summarization. In *Proc. the Document Understanding Conference (DUC)*, 2004.

[20] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. symbolic comput.*, 9(3):251–280, 1990.

[21] M. Craven and S. Slattery. Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, 43(1-2):97–119, 2001.

[22] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.

[23] A. Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proc. ECAI*, pages 328–332, 2004.

[24] A. Darwiche. *Modeling and reasoning with Bayesian networks*. Cambridge University Press, 2009.

[25] A. Darwiche and P. Marquis. A knowledge compilation map. *JAIR*, 17:229–264, 2002.

[26] H. Daumé and D. Marcu. Bayesian query-focused summarization. In *Proc. ACL*, 2006.

[27] L. De Raedt, K. Kersting, A. Kimmig, K. Revoredo, and H. Toivonen. Compressing probabilistic prolog programs. *Machine Learning*, 70(2-3):151–168, 2008.

[28] L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proc. IJCAI*, pages 2462–2467, 2007.

[29] J. DeNero and K. Macherey. Model-based aligner combination using dual decomposition. In *Proc. ACL*, pages 420–429, 2011.

[30] P. Domingos and D. Lowd. Markov logic: An interface layer for artificial intelligence. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 3(1), 2009.

[31] G. Erkan and D. R. Radev. LexRank: Graph-based lexical centrality as salience in text summarization. *JAIR*, 22:457–479, 2004.

[32] D. Fierens, G. V. d. Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *arXiv preprint arXiv:1304.6810 (to appear in Theory and Practice of Logic Programming)*, 2013.

[33] E. Filatova and V. Hatzivassiloglou. A formal model for information selection in multi-sentence sentece extraction. In *Proc. COLING*, 2004.

[34] L. Getoor and B. Taskar. *Introduction to statistical relational learning*. The MIT press, 2007.

[35] D. Gillick and B. Favre. A scalable global model for summarization. In *Proc. the NAACL/HLT Workshop on Integer Linear Programming*, 2009.

[36] J. Goldstein, V. Mittal, J. Carbonell, and M. Kantrowitz. Multi-document summarization by sentence extraction. In *Proc. of ANLP/NAACL Workshop on Automatic Summarization*, 2000.

[37] B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. Parameter learning in probabilistic databases: A least squares approach. In *Proc. ECML/PKDD*, pages 473–488, 2008.

[38] B. Gutmann, I. Thon, and L. De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In *Proc. ECML/PKDD*, pages 581–596, 2011.

[39] T. H. Haveliwala. Topic-sensitive PageRank. In *Proc. WWW*, pages 517–526, 2002.

[40] T. N. Huynh and R. J. Mooney. Online structure learning for markov logic networks. In *Proc. ECML/PKDD*, 2011.

[41] G. Jeh and J. Widom. Scaling personalized web search. In *Proc. WWW*, pages 271–279, 2003.

[42] F. Jelinek. Fast sequential decoding algorithm using a stack. *IBM Journal of Research and Development*, 13:675–685, 1969.

[43] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Exploiting the block structure of the web for computing PageRank. Technical report, Stanford University, 2003.

[44] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating PageRank computations. In *Proc. WWW*, pages 261–270, 2003.

[45] C. Karande, K. Chellapilla, and R. Andersen. Speeding up algorithms on compressed web graphs. *Internet Mathematics*, 6(3):373–398, 2009.

[46] S. Khuller, A. Moss, and J. Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999.

[47] J. Kim and H. Park. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM J. on Scientific Computing*, 33(6):3261–3281, 2011.

[48] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[49] B. H. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Verlag, 2008.

[50] J. Lafferty, A. McCallum, and F. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proc. ICML*, pages 282–289, 2001.

[51] D. D. Lee and H. S. Seung. Algorithm for non-negative matrix factorization. In *Proc. NIPS*, 2000.

[52] C. Lin. Projected gradient methods for nonnegative matrix factorization. *Neural computation*, 19(10):2756–2779, 2007.

[53] C.-Y. Lin. ROUGE: A package for automatic evaluation of summaries. In *Proc. Workshop on Text Summarization Branches Out*, 2004.

[54] C.-Y. Lin and E. Hovy. The automated acquisition of topic signatures for text summarization. In *Proc. COLING*, 2000.

[55] H. Lin and J. Bilmes. Multi-document summarization via budget maximization of submodular functions. In *Proc. NAACL/HLT*, 2010.

[56] H. Lin and J. Bilmes. A class of submodular functions for document summarization. In *Proc. ACL/HLT*, 2011.

[57] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge university press Cambridge, 2008.

[58] R. McDonald. A study of grobal inference algorithm in multi-document summarization. In *Proc. ECIR*, 2007.

[59] L. Meier, S. Van De Geer, and P. Bühlmann. The group lasso for logistic regression. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(1):53–71, 2008.

[60] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. DAC*, pages 272–277, 1993.

[61] H. Morita, R. Sasano, H. Takamura, and M. Okumura. Subtree extractive summarization via submodular maximization. In *Proc. ACL*, pages 1023–1032, 2013.

[62] S. Muggleton. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996.

[63] C. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. Dsharp: fast d-dnnf compilation with sharpsat. In *Advances in Artificial Intelligence*, pages 356–361. Springer, 2012.

[64] M. Nishino, A. Yamamoto, and M. Nagata. A sparse parameter learning algorithm for probabilistic logic program. In *Proc. AAAI workshop StarAI*, 2014.

[65] M. Nishino, N. Yasuda, T. Hirao, J. Suzuki, and M. Nagata. Lagrangian relaxation for scalable text summarization while maximizing multiple objectives. *Trans. of the Japanese Society for Artificial Intelligence*, 28(5):433–441, 2013.

[66] M. Nishino, N. Yasuda, S. Minato, and M. Nagata. Accelerating graph adjacency matrix multiplications with adjacency forest. In *Proc. SDM*, pages 1073–1081, 2014.

[67] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1999.

[68] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):123–231, 2014.

[69] D. Poole. The independent choice logic and beyond. In *Probabilistic inductive logic programming*, pages 222–243. Springer, 2008.

[70] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[71] J. Renkens, D. Shterionov, G. Van den Broeck, J. Vlasselaer, D. Fierens, W. Meert, G. Janssens, and L. De Raedt. Problog2: From probabilistic programming to statistical relational learning. In *Proc. of the NIPS Probabilistic Programming Workshop*, 2012.

[72] A. M. Rush and M. Collins. A tutorial on dual decomposition and lagrangian relaxation for inference in natural language processing. *JAIR*, 45:305 – 362, 2012.

[73] A. M. Rush, D. Sontag, and M. Collins. On dual decomposition and linear programming for natural language processing. In *Proc. EMNLP*, 2010.

[74] T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proc. ICLP*, pages 715–729, 1995.

[75] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *JAIR*, 15:391–454, 2001.

[76] P. Singla and P. Domingos. Lifted first-order belief propagation. In *Proc. AAAI*, volume 8, pages 1094–1099, 2008.

[77] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[78] H. Takamura and M. Okumura. Text summarization model based on maximum coverage problem and its variant. In *Proc. EACL*, 2009.

[79] H. Takamura and M. Okumura. Text summarization model based on the budgeted median problem. In *Proc. CIKM*, 2009.

[80] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[81] G. Van den Broeck, N. Taghipour, W. Meert, J. Dvis, and L. De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *Proc. IJCAI*, pages 2178–2185, 2011.

[82] D. Venugopal and V. Gogate. On lifting the gibbs sampling algorithm. In *Proc. NIPS*, pages 1655–1663, 2012.

[83] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proc. ICS*, pages 307–316, 2006.

[84] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ICS*, pages 1–12, 2007.

[85] V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proc. STOC*, pages 887–898, 2012.

[86] K. Woodsend and M. Lapata. Multiple aspect summarization using integer linear programming. In *Proceedings of the EMNLP/CoNLL 2012*, 2012.

[87] N. Yasuda, M. Nishino, T. Hirao, J. Suzuki, and R. Kataoka. A query-focused summarization method that guarantees the inclusion of query words. In *Proc. TIR*, pages 126–130, 2012.

[88] W.-T. Yih, J. Goodman, L. Vanderwende, and H. Suzuki. Multi-document summarization by maximizing informative content-words. In *Proc. IJCAI*, 2007.

[89] M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[90] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms (TALG)*, 1(1):2–13, 2005.

[91] J. M. Zelle and R. J. Mooney. Inducing deterministic prolog parsers from treebanks: A machine learning approach. In *AAAI*, pages 748–753, 1994.

The contents in Chapter 2 were first published in Transactions of the Japanese Society for Artificial Intelligence, 28(5).

The contents in Chapter 4 were first published in Proceedings of 2014 SIAM International Conference on Data Mining, published by the Society of Industrial and Applied Mathematics (SIAM).