EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Random Generation of Markov Random Fields

Ding, Ke

Award date:
2022

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Mathematics and Computer Science
Uncertainty in AI Research Group

Random Generation of Markov Random Fields

Ke Ding

Supervisors: Cassio de Campos

Eindhoven, May, 2022

Abstract

Probabilistic graphical models are a class of frameworks in machine learning, including representation, inference, and learning. It combines the graphical models with the probabilities of observations. Markov random fields are a case of probabilistic graphical models, which are undirected graphs with factors representing relationships among random variables within the graph. This thesis proposes an algorithm for the random generation of Markov random fields. Our algorithm can generate undirected graphs as benchmarks of the treewidth task and pairwise Markov random fields as benchmarks to evaluate the inferences of Markov random fields. The Markov random fields datasets are created to integrate the partial k-trees technique as the structure of Markov random fields with several sampling methods to generate the factors on the graphs. The method used to generate the partial k-trees has two steps: uniformly generating k-trees by the bijective code technique and removing some edges. For benchmarking, the precise results of queries can be produced efficiently by the variable elimination inference with the perfect orderings known by us but unknown to others. This property results from the perfect ordering being recorded in the construction process but being hard to retrieve from the graph itself. The tasks of answering queries and finding the best elimination orders are NP-hard. Hence, we can build complicated benchmarks to assess the inferences of Markov random fields proposed by others. Besides, increasing the treewidth and the size of graphs improves the difficulty of answering queries on the graphs.

Keywords: partial k-trees, Markov random fields, the perfect ordering, variable elimination inference

Preface

As my graduate studies come to an end, I would like to thank all the professors, teammates and friends who have supported me during this process. Two years have passed by. Although the epidemic and other external factors have brought many barriers to the study process, I am still very grateful for this experience that taught me to be consistent in a changing environment.

First, I would like to thank my supervisor, Dr. Cassio De Campos. He is the professor with whom I communicated the most during my two years of study. I am very appreciative of his patient, guidance, ideas, recommended papers, and introduction of new technologies during the thesis process. I benefited a lot from every meeting with him.

Secondly, I would love to thank my parents for supporting me both spiritually and financially over the past two years and for encouraging me to complete my studies from 7,500 kilometers away. Besides, I hope to express my gratitude to my friends for their accompany and for the happy times I shared with them.

Finally, thank myself for never giving up.

Contents

Abstract **iii**

Preface **v**

Contents **vii**

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Motivation . . . 2
- 1.3 Outline . . . 3

2 Preliminaries **5**
- 2.1 MRFs . . . 5
- 2.2 Variable Elimination Inference . . . 7
  - 2.2.1 Partition Function . . . 7
  - 2.2.2 Most Probable Explanation . . . 7
  - 2.2.3 An Example . . . 8
- 2.3 K-trees and Treewidth . . . 10

3 Literature Review **13**
- 3.1 The Probabilistic Graphical Models . . . 13
- 3.2 Random Generation of Markov Random Fields . . . 14
- 3.3 Inference . . . 14
- 3.4 Benchmark . . . 16

4 Implementation **17**
- 4.1 Preparation . . . 17
- 4.2 Methodology . . . 18
  - 4.2.1 Construction of K-trees . . . 18
  - 4.2.2 Sum-Product Variable Elimination . . . 19
  - 4.2.3 Max-Product Variable Elimination . . . 20
- 4.3 The Benchmark Algorithm . . . 20
- 4.4 Results . . . 23
- 4.5 Evaluation . . . 25
  - 4.5.1 Structure of K-trees . . . 25
  - 4.5.2 Sampling Methods . . . 29
  - 4.5.3 The Removal Rate . . . 29

5 Experiments **33**
5.1 Treewidth Heuristic Algorithms . . . 33
5.2 Elimination Orders Comparison . . . 35
5.3 Randomly Generate MRFs by Adding Edges . . . 42
5.4 Factors on K-cliques . . . 43
5.5 Benchmark the Exact Inferences . . . 45
5.5.1 Junction Tree . . . 45
5.6 Benchmark the Approximate Inferences . . . 46
5.6.1 Approximate Inference using Sampling . . . 46

6 Discussion **49**

7 Conclusions **51**
7.1 Contributions . . . 51
7.2 Future work . . . 52

Bibliography **53**

Appendix **56**

A Code of the benchmark algorithm in Python **57**
A.1 Create a Markov random field as a benchmark for the UAI competition . . . 57
A.2 Create a Markov random field as a benchmark for the tree-width competition 70
A.3 Find the elimination orders from Markov random fields by the straightforward intuition . . . 72
A.4 Translate a Markov random field to a Bayesian Network . . . 74
A.5 Read a Markov random field from UAI file . . . 75

Chapter 1

Introduction

1.1 Background

Markov random fields (MRFs) are a class of probabilistic graphical models (PGMs) widely used to represent and rationalise uncertainty in the real world. For example, handwriting recognition, telecommunication network diagnosis, and object recognition in images [1, 2, 3]. The problems are generally solved by answering queries over multiple random variables in MRFs. The two common types of queries are probability and maximum a posteriori (MAP) queries [4]. The methods applied to answer queries are called inference. There are two categories of inferences: the exact inference, such as variable elimination, belief propagation and junction tree algorithm, and the approximate inference, such as sampling-based inference and variational inference [5].

The queries and inferences are intriguing to researchers and the hot topics in uncertainty in artificial intelligence (UAI) competitions [6]. For example, the conference on UAI holds a UAI competition every two years. These competitions focus on computing the partition function, the marginal probability distribution over a variable given or not given evidence, the cliques marginals, the most likely assignment to all variables, and the most likely assignment to a subset of variables given or not given evidence. The first three tasks are probability queries, and the rest are MAP queries. The use cases selected for these competitions are from diverse domains, such as medical diagnosis and protein-protein interaction. During the challenges, competitors proposed capable solvers and the performance of these solvers were measured and compared within 20 seconds, 20 minutes, and 1 hour time limits.

However, query tasks are NP-hard for the inferences on PGMs [7]. This study is interested in variable elimination inference. It takes $O(nk^{d+1})$ time to answer queries exactly, where n is the number of variables in the MRF, k is the maximum number of values taken by random variables — or the maximum cardinality of random variables — and d is the maximum number of variables in a factor during the process of variable elimination [8]. Since variable elimination is a specific case of dynamic programming, the elimination order matters; therefore, some elimination orders are more efficient than others. Unfortunately, finding the best elimination order is also an NP-hard problem. If MRFs are constructed utilising a particular routine, it would be possible to obtain the best orderings of vertices in the graphs during construction.

Technically speaking, the probabilistic graphical models, consisting of probabilities and graph structures, are graphs. The d in the time complexity of the variable elimination inference is also the treewidth of this graph. The treewidth is an essential parameter of graphs and a crucial question in parameterised algorithms and computational experiments (PACE) [9, 10]. This challenge is held annually with one or two tracks about graphs; it aims to enhance the relationship between parameterised algorithms and practical issues. However, finding the treewidth of a graph is NP-hard [11]. Additionally, graphs with treewidths of exactly k are called k-trees and can be created recursively; therefore, they have the perfect elimination orderings.

1.2 Motivation

Many inference algorithms for answering queries on MRFs are introduced yearly. The performances of these inference algorithms must be evaluated; therefore, benchmarks are needed. This study aims to benchmark two tasks of queries on the MRFs. Specifically, it focuses on computing the partition function and the most probable explanation of MRFs. With the variable elimination inference, solving the partition function problem and the most probable explanation function takes the same time and space complexity. A detailed explanation of the partition function and the most probable explanation can be found in Chapter 2.

This study is only interested in computing the partition function and the most probable explanation of MRFs out of the five query types because they are the basis of other queries. In probability queries, computing the partition function is equivalent to a sum-product problem. The marginal probability distribution over a variable given or not given evidence and the cliques marginal are the ratio of two sum-product problems. For the MAP queries, computing the most probable explanation is identical to an optimisation problem; calculating the most likely assignment to a subset of variables given or not given evidence is the combination of an optimisation problem and a sum-product problem. Overall, the partition function task and the most probable explanation task are the essential parts of queries.

This study proposes an algorithm to randomly generate binary pairwise MRFs. The structure of the MRFs is based on partial k-trees - or subgraphs of k-trees [12], and the factors are sampled by certain distributions on edges. The perfect elimination orders of these MRFs are recorded during the construction process of k-trees. Then the algorithm removes a subset of edges from MRFs, making retrieving the ideal elimination orders challenging. Therefore, the perfect elimination orders of generated MRFs are known, and the tasks of computing the partition function and the most probable explanation can be solved efficiently as benchmarks. On the contrary, for people who do not know the perfect elimination order in advance, computing the partition function and the most probable explanation of these generated MRFs is still complicated. The MRFs generated by our algorithm can be available for the UAI competition. They mentioned in their post-competition summary that they would require more challenging networks and standardised benchmarks. Since they reported that the Markov networks are harder than Bayesian networks, our algorithm that generates MRFs is valuable for creating the datasets and benchmarks satisfying their demand.

In addition to the UAI competition, the generated MRF without factors can also be applied as

benchmarks in the PACE challenge. Even though the treewidth of a k-tree is accessible, when utilising beneficial strategies of removing edges to k-trees, the treewidth of the partial k-tree becomes problematic to obtain. Besides, uniformly generating k-trees enables the unbiased validation of algorithms that compute treewidth.

1.3 Outline

The remainder of this thesis is as follows: In chapter 2, the definition of MRFs, variable elimination, k-trees, and treewidth are discussed. Chapter 3 reviews the current research situation in the domains of PGMs, MRFs, inference, treewidth, and benchmarks. Our algorithm is described in detail, including the methods, process, results, and performance in chapter 4. Then, chapter 5 examines various experiments, such as comparing different elimination order generation approaches, factors on k-cliques, and benchmark applications. The advantages, disadvantages, and other implications of this algorithm are presented in chapter 6. Finally, the contributions and future work are summarised in chapter 7.

Chapter 2

Preliminaries

2.1 MRFs

MRFs are also called Markov networks or undirected graphical models (UGMs), an alternative to Bayesian networks (BNs) - or directed acyclic graphical models. Both are probabilistic graphical models, and the graph formalism for compactly modelling joint probability distributions and dependence or independence relations over a set of random variables. One advantage of MRFs compared to Bayesian Networks is that they are symmetric, thus making it easy to represent the mutual influence relationship. However, the parameters are consequently less interpretable and less modular [4, 13, 14, 15].

This section provides the definition and properties of MRFs.

Definition 1 *Given an undirected graph G = (V, E), where V is a set of vertices and E is a set of edges in the graph G, a set of random variables* $X = (X_v)_{v \in V}$ *indexed by V form a MRF with respect to G if they satisfy the local Markov properties.*

Property 1 *Pairwise Markov property*
Any two non-adjacent variables u, v are conditionally independent given all other variables:

$$X_u \perp\!\!\!\perp X_v | X_{V \setminus \{u,v\}}.$$

Property 2 *Local Markov property*
A variable v is conditionally independent of all other variables given its neighbours:

$$X_v \perp\!\!\!\perp X_{V \setminus N[v]} | X_{N(v)},$$

where N(v) is the set of neighbours of v and $N[v] = v \cup N(v)$.

Property 3 *Global Markov property*
Any two subsets of variables A and B are conditionally independent given a separating subset $V \setminus A \cup B$*:*

$$X_A \perp\!\!\!\perp X_B | X_{V \setminus A \cup B}.$$

Figure 2.1 gives an example from [15] to illustrate the differences between Bayesian networks and Markov networks. There are seven nodes in the graphs presenting seven variables $X_1, ..., X_7$.

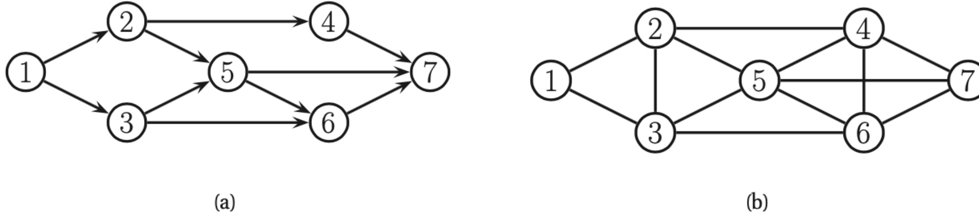


Figure 2.1: (a) A Bayesian network; (b) the corresponding MRF

The joint distribution represented by the Bayesian network (BN) in Figure 2.1(a) is

$$p(X_1, X_2, X_3, X_4, X_5, X_6, X_7) =$$
$$p(X_1)p(X_2|X_1)p(X_3|X_1)p(X_4|X_2)p(X_5|X_2, X_3)p(X_6|X_3, X_5)p(X_7|X_4, X_5, X_6), \quad (2.1)$$

where $p(X_i)$ is the distribution of the random variable X_i and $p(X_i|X_j)$ is the conditional distribution of the random variable X_i given the random variable X_j (the parents in the graph for BNs). Here, $p(X_i) \in [0, 1]$ and $p(X_i|X_j) \in [0, 1]$. We refer to textbooks for more details on BNs [4].

The joint distribution represented by the MRF in Figure 2.1(b) is

$$p(X_1, X_2, X_3, X_4, X_5, X_6, X_7) =$$
$$\frac{1}{Z}\phi(X_1, X_2, X_3)\phi(X_2, X_3, X_5)\phi(X_2, X_4, X_5)\phi(X_3, X_5, X_6)\phi(X_4, X_5, X_6, X_7), \quad (2.2)$$

where $Z = \sum_{X_1,\ldots,X_7} \phi(X_1, X_2, X_3)\phi(X_2, X_3, X_5)\phi(X_2, X_4, X_5)\phi(X_3, X_5, X_6)\phi(X_4, X_5, X_6, X_7)$ and $\phi(X_i, X_j)$ is a factor describing the relation between X_i and X_j; or

$$p(X_1, X_2, X_3, X_4, X_5, X_6, X_7) = \frac{1}{Z}\phi(X_1, X_2)\phi(X_1, X_3)\phi(X_2, X_3)\phi(X_2, X_4)\phi(X_2, X_5)$$
$$\cdot\phi(X_3, X_5)\phi(X_3, X_6)\phi(X_4, X_5)\phi(X_4, X_6)\phi(X_5, X_6)\phi(X_5, X_7)\phi(X_4, X_7)\phi(X_6, X_7), \quad (2.3)$$

where $Z = \sum_{X_1,\ldots,X_7} \phi(X_1, X_2)\phi(X_1, X_3)\phi(X_2, X_3)\phi(X_2, X_4)\phi(X_2, X_5)\phi(X_3, X_5)\phi(X_3, X_6)$ $\cdot\phi(X_4, X_5)\phi(X_4, X_6)\phi(X_5, X_6)\phi(X_5, X_7)\phi(X_4, X_7)\phi(X_6, X_7)$ as a pairwise MRF, and $\phi(X_i, X_j)$ is a factor describing the relation between X_i and X_j. To be noted, different from Bayesian Networks, here, $\phi(\mathcal{X}) \in [0, +\infty)$.

An example to indicate the different relations described by a BN and a corresponding MRF is in Figure 2.1(a), $X_1 \perp\!\!\!\perp X_4|X_2$, while in Figure 2.1(b), $X_1 \perp\!\!\!\perp X_4|X_2, X_3$ (which also holds for the BN).

Moreover, this study mainly focuses on the binary pairwise MRFs, because this kind of MRFs is powerful and high-order MRFs can be transformed into binary pairwise MRFs for further analysis. Additionally, the case that the binary MRFs with factors on cliques is investigated in Section 5.4.

2.2 Variable Elimination Inference

2.2.1 Partition Function

The partition function (PR) of an undirected graph $G = (V, E)$ is,

$$Z = \sum_{\mathcal{X}} \prod_{\phi_i \in \Phi} \phi_i, \tag{2.4}$$

where $\mathcal{X}$ is the set of random variables on the vertices V,$\phi_i \in [0, +\infty)$ is a factor, and Φ is the set of factors of the graph G. It is also known as the normalizing constant and guarantees that the distribution of the MRF sums to 1. In Formula (2.2) and in Formula (2.3), Z is the partition function of the MRF in Figure 2.1(b) with different factor sets, respectively.

In practice, to alleviate the underflow and overflow problems caused by the finite precision of the computer, the log-sum-exp trick is applied [4] that all computation is performed in log-space. Therefore, the main operations in the variable elimination inference, multiplication and addition, are replaced by addition and log-sum-exponential operations. Here is an example. Let $u_i = \log(\phi_i)$. Then

$$Z = \sum_{\mathcal{X}} \prod_{\phi_i \in \Phi} \phi_i = \sum_{\mathcal{X}} \prod_{u_i \in \log(\Phi)} e^{u_i} = \sum_{\mathcal{X}} \exp(\sum_{u_i \in \log(\Phi)} u_i). \tag{2.5}$$

The concept of the log-sum-exp operation is that For $e^a + e^b = e^c$ and the values of a and b are known, the value of c can be calculated by the following formula

$$c = log(e^a + e^b) = a + log(1 + e^{b-a}). \tag{2.6}$$

Combining Formula (2.5) and Formula (2.6), the final result of the partition function is $log(Z)$.

2.2.2 Most Probable Explanation

The most probable explanation (MPE) is one special case of the maximum a posterior inference (MAP). Given $\mathbf{W} = \mathcal{X} \setminus \mathbf{E}$ and the evidence $\mathbf{E} = e$, the most likely assignment to the variables in $\mathbf{W}$ is

$$MAP(\mathbf{W}|e) = \underset{w}{\operatorname{argmax}}\, P(\mathbf{W}|\mathbf{E} = e), \tag{2.7}$$

$$\begin{aligned} MPE(\mathcal{X}) &= \underset{x}{\operatorname{argmax}}\, P(\mathcal{X}) \\ &= \underset{x}{\operatorname{argmax}} \frac{1}{Z} \prod_{\phi_i \in \Phi} \phi_i \\ &= \underset{x}{\operatorname{argmax}} \exp(\sum_{u_i \in \log(\Phi)} u_i) \\ &= \underset{x}{\operatorname{argmax}} \sum_{u_i \in \log(\Phi)} u_i. \end{aligned} \tag{2.8}$$

The log-sum-exponential strategy also works for calculating the most probable explanation, as Formula (2.8).

2.2.3 An Example

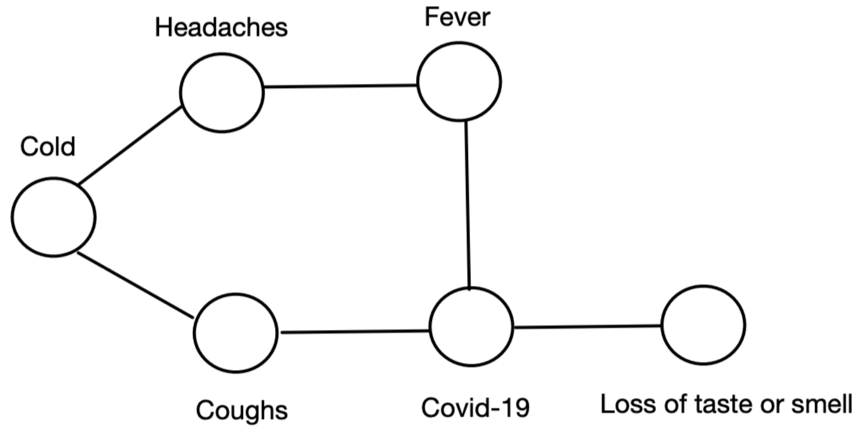


Figure 2.2: The Markov random field example regarding getting cold or getting COVID-19

The following example illustrates the meanings of the partition function and the most probable explanation. Figure 2.2 presents a binary pairwise MRF with six vertices to model a simple case according to the following facts: the most common symptoms of Covid-19 are fever, coughs, and loss of taste or smell; having coughs and headaches probably indicates getting cold; headaches are one of the symptoms of fever. The probabilities are shown in Table 2.1-2.6. The simplified relations and the distributions over random variables are designed only for demonstration purposes.

ϕ(Cold, Headaches)	Headaches = 0	Headaches = 1
Cold = 0	10	1
Cold = 1	0.5	0.1

Table 2.1: The factor over random variables Cold and Headaches

ϕ(Cold, Coughs)	Coughs = 0	Coughs = 1
Cold = 0	10	2
Cold = 1	3	0.1

Table 2.2: The factor over random variables Cold and Coughs

ϕ(Headaches, Fever)	Fever = 0	Fever = 1
Headaches = 0	10	0.01
Headaches = 1	2	0.05

Table 2.3: The factor over random variables Headaches and Fever

ϕ(Coughs, Covid-19)	Covid-19 = 0	Covid-19 = 1
Coughs = 0	10	1
Coughs = 1	2	0.1

Table 2.4: The factor over random variables Coughs and Covid-19

ϕ(Fever, Covid-19)	Covid-19 = 0	Covid-19 = 1
Fever = 0	10	1
Fever = 1	2	0.1

Table 2.5: The factor over random variables Fever and Covid-19

ϕ(Covid-19, Loss of taste or smell)	Loss of taste or smell = 0	Loss of taste or smell = 1
Covid-19 = 0	10	0.02
Covid-19 = 1	1	0.1

Table 2.6: The factor over random variables Covid-19 and Loss of taste or smell

Here is a short explanation of Table 2.6. The value of the factor when a person has taste and smell and does not get Covid-19 is 10, which is much greater than the rest three cases. It indicates a strong relationship between not getting Covid-19 and not losing taste or smell.

Given $\mathcal{X} = \{\text{Cold, Headaches, Fever, Coughs, Loss of taste or smell, Covid-19}\}$, in our example, the probability of getting Covid-19 is

$$
\begin{aligned}
&p(\text{Covid-19} = 1) \\
&= \frac{1}{Z} \sum_{\mathcal{X} \setminus \text{Covid-19}} \phi(\text{Cold, Headaches, Fever, Coughs, Loss of taste or smell, Covid-19} = 1) \\
&\approx 0.0010,
\end{aligned}
$$

where $Z = \sum_{\mathcal{X}} \phi$(Cold, Headaches)$\phi$(Cold, Coughs)$\phi$(Headaches, Fever)$\phi$(Coughs, Covid-19)$\phi$(Fever, Covid-19)$\phi$(Covid-19, Loss of taste or smell) $\approx$ 1080137.5129.

The most probable explanation is that $MPE(\mathcal{X}) = \text{argmax}_x\, \phi(\mathcal{X})$. The results in Table 2.7 show that most of the time, people are healthy without any symptoms of diseases. The probability of this case is $p(MPE(\mathcal{X})) \approx 0.9258$.

Random Variable	Cold	Headaches	Fever	Coughs	Loss of taste or smell	Covid-19
Assignment	0	0	0	0	0	0

Table 2.7: The MPE results of this example

2.3 K-trees and Treewidth

K-trees are a class of graphs with treewidth of k. The informal explanation of the treewidth is a parameter measuring the size of graphs (in comparison to trees). There are several exchangeable formal definitions of treewidth; however, they are out of the scope of this thesis, and can be found in [16, 17, 18] for interested readers. Partial k-trees are graphs with treewidth at most k. Both k-trees and partial k-trees are appealing generalizations of trees since they can solve NP-complete problems in polynomial time [19]. For example, Cédric Bentz proved that determining the existence of a colour list on node-weighted k-trees and partial k-trees, which is an NP-complete problem, can be completed in polynomial time [20]. Moreover, k-trees have perfect orderings [17]. One application of these orderings is computing the numerical solution of linear equations, specifically, in sparse positive definite systems [21]. After associating the parameters' matrices of linear systems with undirected graphs, the decomposition process can be formulated as eliminating vertices in graphs.

Before introducing the formal definition of k-trees, recall the definition of clique in graph theory [22].

Definition 2 *A clique in a graph is a set of pairwise adjacent vertices.*

To be specific, if there is an undirected graph *G=(V, E)*, where V is the set of vertices and E is the set of edges, a clique C in graph G is a subset of the vertices $C \subseteq V$, such that every two distinct vertices are adjacent $e_{ij} = (v_i, v_j) \in E, \forall v_i, v_j \in C, i \neq j$.

A clique of size k is called a k-clique. The definition of k-tree is as follows [23].

Definition 3 *A k-tree is defined in the following recursive way:*
1. A k-clique is a k-tree.
2. If $T_k^{'} = (V, E)$ *is a k-tree,* $K \in V$ *is a k-clique and* $v \notin V$, *then* $T_k = (V \cup \{v\}, E \cup \{(v, x) | x \in K\})$ *is a k-tree.*

The perfect elimination order comes from its recursive construction process. When the perfect ordering and construction of a tree from the k-tree is followed, each tree vertex is a subset

of the k-tree vertices. Therefore, the maximum size of the subset in the tree is k+1, which reveals the treewidth of this k-tree is k. This process is also called tree decomposition.

There are two straightforward approaches to finding the perfect ordering and constructing a tree from a k-tree. One is Min-Neighbours, and the other is Min-Fill. The Min-Neighbours approach finds the vertex with the minimum degree in the current graph and updates the graph by removing the found vertex and connecting its neighbours until all vertices are removed from the graph. The treewidth of the graph is the maximum degree of a vertex established during the process. The Min-Fill approach finds the vertex that, after removing it, the minimum number of edges are added when updating the current graph and then updates the graph until all vertices are removed. The order of vertice removal from the graph using the Min-Fill approach, combined with tracking the degree of vertices during removal, gives the maximum degree, which equals the treewidth of the graph.

Chapter 3

Literature Review

This section provides an overview of the research in the probabilistic graphical models' domains, MRF random generation, inference, and benchmarks. First, the recent studies on the probabilistic graphical models are discussed. Although Bayesian networks are the popular research topic in this domain, MRFs are an exciting and worthy object with vast potential. This budding interest is why this project investigated the MRFs, specifically randomly generating MRFs. The latest research about this subject is presented in Section 3.2, including random generation of graphs, networks, Bayesian networks, and MRFs. In this project, the structure of MRFs is k-trees. The other exciting findings of k-trees are also reviewed. Moreover, the intention of generating MRFs based on k-trees is to compute the partition function and the most probable explanation efficiently. The current approaches to solving these tasks are introduced in Section 3.3. Finally, the last goal is to evaluate the inference of MRFs; therefore, several investigations of general benchmark methods are given in Section 3.4.

3.1 The Probabilistic Graphical Models

The earliest research on the probabilistic graphical models is from Pearl in the 1980s [24, 25]. This class of models combines the knowledge of probability theory and graph theory; therefore, the dependencies and probability relationships can be described by employing the networks. The main research focuses are network representations, inference, and learning [26, 15]; inference remains a hot topic in this domain.

In 2022, Singh et al. [27] proposed an effective inference algorithm to solve filtering problems in the probabilistic graphical models, which are built on aggregated data from many individuals. Their algorithm combines the Sinkhorn algorithm and the standard belief propagation algorithm, guaranteeing global convergence with polynomial computational complexity. It shows excellent performance in the bird migration and human mobility datasets with hidden Markov models.

There is another application of the probabilistic graphical models in sequential credit card fraud detection using deep neural networks. Forough et al. [28] noticed the fraud problem rising with the prevalence of electronic banking. They proposed a two-step model consisting of an LSTM network and a conditional random field. This model achieved outstanding performance compared to baseline models, such as LSTM, ANN and GRU, on two processed

credit card fraud detection datasets using their novel undersampling algorithm.

3.2 Random Generation of Markov Random Fields

Several methods randomly generate graphs; however, the domain of random generation of MRFs is still unexplored. One possible structure of MRFs is the K-tree, which provides several helpful inherent features.

In 2002, Ide et al. [29] built a heuristic algorithm to randomly generate Bayesian networks. Their ideas come from Melancon et al. [30], who proposed a simple algorithm based on a Markov chain to create acyclic digraphs with a given number of vertices uniformly at random. The procedure takes the number of nodes and iterations at the beginning, then builds a simple tree with one parent and begins to iterate. Inside the loop, the process starts with the random selection of two nodes, i and j, then chooses one option from three: add the arc (i, j), delete the arc (i, j) or keep everything the same. After exiting the loop, a Bayesian network is created. The Markov chain backs up the algorithm, and Dirichlet distributions are used as the conditional distributions for each node. The authors also developed a Java program for this algorithm and produced some experimental results to indicate the correctness of the algorithm.

In 2006, Britton et al. [31] gave four approaches for generating simple undirected graphs: the erased configuration model, the repeated configuration model, the generalized random graph, and the directed graph with removed directions (DGRD) theoretically. The following four distributions are discussed to control the degree distribution: power law, Poisson, mixed Poisson, and compound Poisson distributions. Mathematically examining the four approaches reveals that the exact forms of the desired distributions are not yet achievable. However, based on certain assumptions, the asymptotic distributions are reachable. At the end of the work, the time complexity is considered, and the generalized random graph model requires the fewest operations and scales as $O(n)$. The four approaches to generating simple undirected graphs have not been implemented, but there are similar works on developing random graphs [32, 33].

In 2010, Caminiti et al. [19] introduced a new bijective code for labelled k-trees, or undirected graphs, shown in Figure 2.3. The procedure produces the coding and decoding algorithms running in linear time with respect to the size of the k-tree. With theoretical support, four steps are needed in the coding and decoding processes. Generalized Dandelion Code is used in the coding process for the perfect ordering. It provides a method to present k-trees with codes, which makes k-trees easy and efficient to display with strings, which are then easily deciphered into graphs.

3.3 Inference

The estimation of marginal probabilities and the most probable states of variables sets is called inference in the domain of the probabilistic graphical models [34]; it is also a difficult task. The complexity of different tasks and the summary of common inferences are described below.

Most theoretical analyses of the complexity of inference algorithms are done on Bayesian networks since they can be treated as a special version of MRFs while maintaining the same presentation size. The complexity of Bayesian networks is equivalent to that of MRFs.

The inference tasks in the probabilistic graphical models are NP-hard, which means in the worst case, exponential time is needed. However, in practice, the worst case does not always occur. The exponential blowup can be avoided by finding a good elimination order — a better case [4].

In 2005, Campos et al. [35] provided an overview of the inferential complexity of Bayesian and Credal networks with proofs. Table 3.1 summarizes the inferential complexity of Bayesian network tasks from this paper. Five inferences are mentioned, but the focus here is on the first two with bounded induced width. However, the third task can be solved with the result of the first two. Therefore, our algorithm can also address it, but it doubles running time. The PP-Complete refers to probabilistic polynomial time, and the NP-Complete refers to nondeterministic polynomial-time complete.

Problem	Polytree	Bounded induced-width	Multiply-connected
BN-PR	Polynomial	Polynomial	PP-Complete
BN-MPE	Polynomial	Polynomial	NP-Complete
BN-MPEe	Polynomial	Polynomial	PP-Complete
BN-MAP	NP-Complete	NP-Completel	$\mathrm{NP}^{\mathrm{PP}}$-Complete
BN-MmAP	$\sum_2^P$-Complete	$\sum_2^P$-Complete	$\mathrm{NP}^{\mathrm{PP}}$-Hard

Table 3.1: Complexity results of Bayesian networks

In 2011, Ishikawa [36] proposed a method to transform higher-order MRFs into pairwise MRFs without message loss. This method, also known as higher-order reduction, is generally used in computer vision. It helps to address inferences of higher-order MRFs fast and with less space. The reduced MRFs with the fusion-move and QPBO algorithms can solve energy minimization problems in the computer vision domain. Similar methods such as the binary energy reduction and Potts model are compared.

In 2014, Fix et al. [37] improved Ishikawa's method by utilizing the underlying hypergraph structure of the MRFs. Two steps are needed. Firstly, eliminate all higher-order positive terms. Then, reduce negative coefficients term-by-term. In the worst performance case, the number of new variables is $n + O(td)$, and the number of submodular terms is $O(td^2)$. The greatest number of non-submodular terms is n, where n is the number of variables in an MRF, d is the degree of nodes, and t is the number of positive terms.

3.4 Benchmark

One application of our algorithm is to benchmark inference methods of MRFs. There are few works regarding network benchmark design through sampling datasets from deliberately constructed benchmark algorithms.

In 2013, Trabelsi et al. [38] designed a 2-TBN generation algorithm to benchmark dynamic Bayesian networks and suggested a novel metric (i.e., the structural Hamming distance) for evaluating the performance of 2-TBN structure learning algorithms. The 2-TBN generation algorithm using the tiling approach has been implemented by Matlab and can generate large and realistic 2-TBNs, which can then be employed to create new datasets. The structural Hamming distance can compare the structure of dynamic Bayesian networks by integrating temporal knowledge. Additionally, the authors indicated that any background knowledge needs to be considered for the structural Hamming distance metric. However, no case study was conducted to test the performance of their 2-TBN algorithm.

In 2016, Ishak et al. [39] did similar work as [38] based on [29], relating to data mining problems. They provided two algorithms, one for random generation of the relational schema and the other for random generation of the probabilistic relational models (PRMs), to benchmark these PRMs. Database generation contains four steps: creating the database schema, determining data distribution, generating it, and loading all these components into the database system. After these algorithms were designed, the authors implemented them in C++; the complexities of the generation processes have been discussed.

Furthermore, the previous UAI competitions have numerous benchmarks to measure the inference solvers submitted by competitors. The summary of these benchmarks can be found here, including statistical descriptions of these networks and the results of inference tasks. There are almost eleven benchmarks from diverse domains, including MRFs and Bayesian networks.

Chapter 4

Implementation

4.1 Preparation

In order to generate MRFs, the main parts of MRFs, namely, the structures and the parameters, are constructed sequentially. The structures of MRFs in our algorithm are partial k-trees, while the parameters of factors on these MRFs are randomly sampled from different distributions.

There are seven methods in our consideration. All of them are distributions, i.e., the Dirichlet distribution, the exponential distribution, the Beta distribution, the chi-square distribution, the uniform distribution over $[0, 10)$, the uniform distribution over $[0, 1)$ and the logarithmic normal distribution. Table 4.1 shows the property, parameters and interval of these methods [40] used in our algorithm.

Name	Property	Parameters	Interval
Dirichlet distribution	$\sum_{i=1}^{k} x_i = 1$,	$\alpha_i = [1, 1, 1, 1]$,	$x_i \in (0, 1)$
	$p(x) \propto \prod_{i=1}^{k} x_i^{\alpha_i - 1}$	$k = 4$	
Exponential distribution	$p(x; \frac{1}{\beta}) = \frac{1}{\beta} e^{(-\frac{x}{\beta})}$	$\beta = 1$	$x_i \in (0, +\infty)$
Beta distribution	$p(x; a, b) = \frac{1}{B(\alpha,\beta)} x^{(\alpha-1)}(1-x)^{(\beta-1)}$,	$\alpha = 2$,	$x_i \in [0, 1]$
	$B(\alpha, \beta) = \int_0^1 t^{(\alpha-1)}(1-t)^{(\beta-1)} dt$	$\beta = 3$	
Chi-square distribution	$p(x) = \frac{(\frac{1}{2})^{\frac{df}{2}}}{\Gamma(\frac{df}{2})} x^{(\frac{df}{2}-1)} e^{(-\frac{x}{2})}$,	$df = 2$	$x_i \in [0, +\infty)$
	$\Gamma(x) = \int_0^{-\infty} t^{(x-1)} e^{(-t)} dt$		
Uniform distribution over $[10, 100)$	$p(x) = \frac{1}{(b-a)}$	$a = 10$,	$x_i \in [10, 100)$
		$b = 100$	
Uniform distribution over $[0, 1)$	$p(x) = 1/(1-0)$		$x_i \in [0, 1)$
Logarithmic normal distribution	$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{(\frac{-(\ln(x)-\mu)^2}{2\sigma^2})}$	$\mu = 0.0$,	$x_i \in (0, +\infty)$
		$\sigma = 1.0$	

Table 4.1: The summary of selected sampling methods

The idea of the selecting these proper factor generation methods is as diverse as possible. Therefore, the intervals of these sampling methods are almost different. The only requirement for the values of factors on the MRFs is that the values should be greater than or equal to 0. All selected methods are satisfied.

4.2 Methodology

4.2.1 Construction of K-trees

The recursively generated k-trees described in Section 2.3 tend to be star-shaped because the early generated k-cliques have higher probabilities of being selected and connected by the newly added vertex. Therefore, the earlier the k-cliques are generated, the greater their degrees of connection when converted to the tree shape.

This algorithm is inspired by Caminiti et al. [19], who introduced a bijective code to encode and decode k-trees, tailoring it to generate k-trees and MRFs uniformly. Caminiti et al. coded every labelled k-tree as an adapted Generalized Dandelion Code and proved that the code is bijective. Thus, our algorithm implements and utilizes their idea, randomly generating an adapted Generalized Dandelion Code (Q, S). Here, Q is a set of beginning k vertices, or the first k-clique, of the k-tree, and S is a string of the Generalized Dandelion Code regarding the relations of edges in the k-tree. With the decoding process, the k-tree is reconstructed. The set of the adapted Generalized Dandelion Code is

$$\mathcal{A}_k^n = \binom{[1,n]}{k} \times (\{(0,\varepsilon)\} \cup ([1,n-k] \times [1,k]))^{n-k-2}. \tag{4.1}$$

The decoding process with the input (Q, S) is as follows:

Step 1: Compute a mapping that maps the original labels of vertices to the labels of a Rényi k-tree according to Q and find the largest leaf and the smallest vertex that does not belong to the first k-clique.

Step 2: Insert the pair $(0, \varepsilon)$ into S, which indicates the position of the largest leaf after mapping and decoding the string to obtain the directed characteristic tree.

Step 3: Reconstruct the Rényi k-tree by the directed characteristic tree, and record the ordering and the k-cliques of this Rényi k-tree.

Step 4: Remap the Rényi k-tree labels to the original k-trees and the ordering and the set of k-cliques.

Figure 4.1 from this paper shows the process of coding. The code of this k-tree in Figure 4.1(a) is $([2,3,9],[(0,\varepsilon),(2,1),(8,3),(8,2),(1,3),(5,3)])$, where $[2,3,9]$ is the first k-clique in the k-tree, $(0,\varepsilon)$ means for the vertex labelled 3 in Figure 4.1(e), its parent is vertex labelled 0 and the edge label between its parent and itself is ε. The another example, $(2,1)$ represents the vertex labelled 4, its parent is vertex labelled 2 and the edge label between its parent and itself is 1. The details of the bijective code can be found in [19].

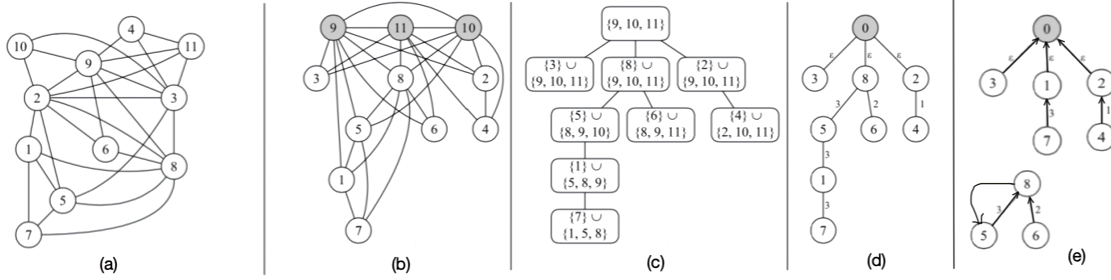


Figure 4.1: (a) A 3-tree with 11 nodes and the first k-clique 2, 3, 9. (b) The Rényi k-tree of this 3-tree. (c) The skeleton of this 3-tree. (d) The characteristic tree of this 3-tree. (e) The transformed directed tree of this 3-tree.

4.2.2 Sum-Product Variable Elimination

The method used to compute the partition function is the sum-product variable elimination. Below is the pseudocode of this method.

Algorithm 1 Sum-Product Variable Elimination

Require: Φ, Xo $\triangleright$ Φ is the set of factors and Xo is the set of ordered variables to be eliminated
1: **for** $x_i \in Xo$ **do**
2: $\quad \Phi_{re} \leftarrow \{\phi \in \Phi : x_i \in Scope\{\phi\}\}$
3: $\quad \Phi_{unre} \leftarrow \Phi \setminus \Phi_{re}$
4: $\quad \psi_{x_i} \leftarrow \prod_{\phi \in \Phi_{re}} \phi$
5: $\quad \tau \leftarrow \sum_{x_i} \psi_{x_i}$
6: $\quad \Phi \leftarrow \Phi_{unre} \cup \{\tau\}$
7: **end for**
8: **return** Φ

The inputs of Algorithm 1 are the set of factors of this MRF and the ordered variables to be eliminated. The order of all eliminated variables is the elimination order. Afterwards, input variables are removed in this order one by one. At first, find all factors related to the variable. Then, multiply all related factors and eliminate the variable by adding the factors according to the value of the variable. Next, update the factor set by uniting the unrelated factors of the variable and the new factor after eliminating this variable from the product of all related factors. Lastly, since all eliminated variables are handled, the factor set contains one value, which is the partition function of the MRF.

As discussed in Section 3, the elimination order of variables is essential in Algorithm 1, which significantly influences this algorithm's complexity. Because the k-trees are built recursively, the elimination order of a k-tree is easy to record and retrieve. The retrieval method is the minimum degree algorithm [21] (i.e., the Min-Neighbors approach). However, if some edges are removed, the elimination order would be unable to retrieve by the minimum degree algorithm. Since we create these k-trees, it is still accessible for us to record the elimination orders, and they are valuable even for these partial k-trees.

4.2.3 Max-Product Variable Elimination

The Max-Product Variable Elimination is also called MAP inference. It can be seen from the name that this method is similar to the previous one. However, the max-product variable elimination has two parts. Foremost, find the highest probability value. Then retrieve the most likely assignment for each random variable, i.e., the most probable explanation. The pseudocode is below.

Algorithm 2 Max-Product Variable Elimination

Require: Φ, Xo ▷ Φ is the set of factors and Xo is the set of ordered variables to be eliminated

1: $\Psi \leftarrow \emptyset$
2: $Assign \leftarrow \emptyset$
3: **for** $x_i \in Xo$ **do**
4: $\quad \Phi_{re} \leftarrow \{\phi \in \Phi : x_i \in Scope\{\phi\}\}$
5: $\quad \Phi_{unre} \leftarrow \Phi \setminus \Phi_{re}$
6: $\quad \psi_{x_i} \leftarrow \prod_{\phi \in \Phi_{re}} \phi$
7: $\quad \tau \leftarrow \max_{x_i} \psi_{x_i}$
8: $\quad \Psi \leftarrow \Psi \cup \{\psi_{x_i}\}$
9: $\quad \Phi \leftarrow \Phi_{unre} \cup \{\tau\}$
10: **end for**
11: $u_n \leftarrow \text{argmax}_{x_n} \psi_{x_n}$ ▷ x_n is the last variable in the elimination order
12: $Assign \leftarrow Assign \cup \{(x_n, u_n)\}$
13: **for** $x_i \in \bar{X}o \setminus \{x_n\}$ **do** ▷ $\bar{X}o$ is Xo in the reverse order
14: $\quad u_i \leftarrow \text{argmax}_{x_i} \psi_{x_i}(x_{i+1} = u_{i+1})$
15: $\quad Assign \leftarrow Assign \cup \{(x_i, u_i)\}$
16: **end for**
17: **return** $Assign$

There are two differences between Algorithm 1 and Algorithm 2. The first one is in Line 7; the process of summing factors over variables is replaced by maximizing factors. Additionally, in Line 8, the factors after the multiplication and the maximization operations are saved for tracing back the assignment of each eliminated variable. The trace back procedure is from Line 11 to the end, based on the reverse elimination order. Since the assignment of the previous variable is decided, the fixed value of the variable determines the next one.

4.3 The Benchmark Algorithm

The methods and configurations are discussed in the previous sections. Next, the procedure of our algorithm will be presented. Figure 4.2 provides an overview of our algorithm, and the code is attached in Appendices A.

The benchmarking algorithm can be split into two parts. The first part is benchmarking treewidth algorithms, including creating k-trees, removing edges and generating benchmarks. The second part is to benchmark inferences regarding MRFs, which has three steps. The left part in Figure 4.2 is the first step, random generation of MRFs; the second step in Figure 4.2

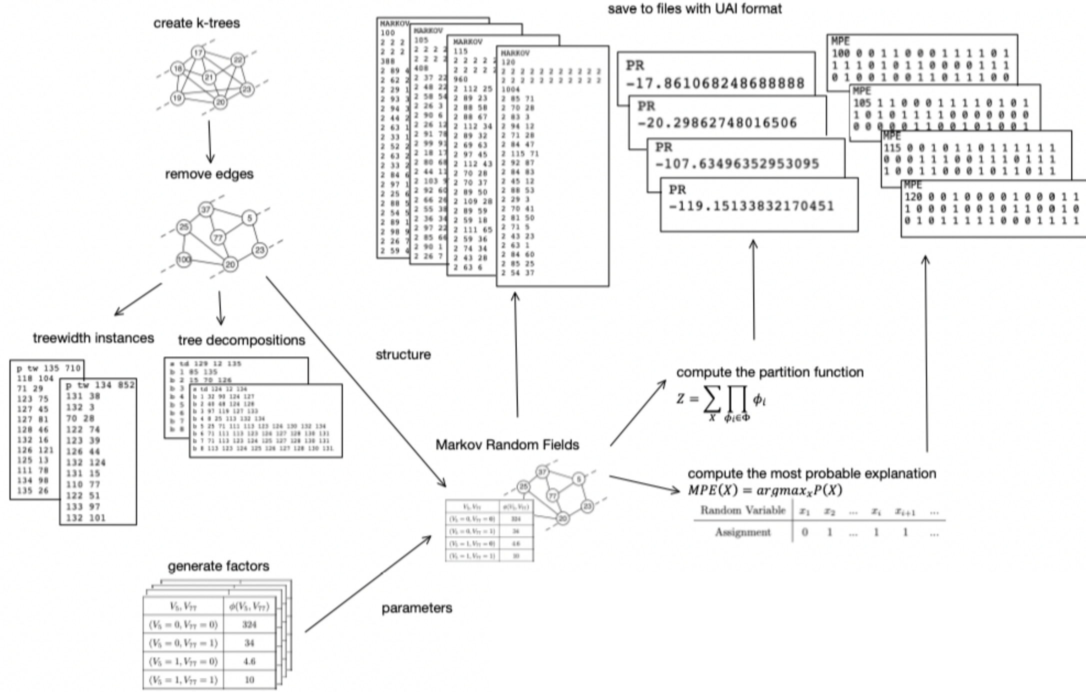


Figure 4.2: The procedure of the algorithm

below is to benchmark the partition function and the most probable explanation. Ultimately, MRFs, their partition functions and the most probable explanations are saved to files.

The interpretation of the benchmark algorithm in detail is as follows.

Create a k-tree. According to Section 4.2.1, given the number of vertices and the value of k, a k-tree and its perfect ordering will be returned by deciphering an adapted Generalized Dandelion Code. With the specific number of vertices n and the treewidth k, the number of k-trees is $|\mathcal{T}_k^n| = \binom{n}{k}(k(n-k)+1)^{n-k-2}$. The difference between these k-trees comes from that the k-clique connecting to the inserted vertex being randomly selected, as in Definition 3. However, for the bijective code, our algorithm that randomly generates the adapted Generalized Dandelion Code can uniformly generate k-trees over the k-tree space with a specific treewidth k and the number of vertices n in the graph.

Remove edges. As mentioned before, the Min-Neighbours approach can effortlessly find the perfect ordering of k-trees. Therefore, some edges are removed from k-trees and partial k-trees are created to hide the perfect ordering. In our code, the users can select either randomly removing edges or removing edges such that their constituents have larger degrees. The first edge removal strategy reveals that each edge has the same probability of being removed, while the second changes the removed probability of each edge in different cases. Besides, the number of removed edges is controlled by the removal rate. The removal rate depends on the density of graphs, section 4.5 assesses how different choices of the removal rate affect the performance of the benchmark algorithm.

Treewidth instances. This step generates instances for treewidth tasks. Since the partial k-trees are created in the previous step, they can be saved in .gr files as instances of treewidth tasks. The first line in the .gr file states: "p" means this is a problem; "tw" means treewidth; the following two numbers are the number of vertices and the number of edges, respectively. The rest part is two columns where each row represents an edge.

Tree decomposition. The solutions to the instances generated in the previous step are saved in the .td file with the same file name. The first line in the .td file states: "s" means this is a solution; "tw" means treewidth; the following three numbers are the number of vertices in the tree mapped from a k-tree, the treewidth of this graph, and the number of vertices in the graph, respectively. Then, each line is a bag of vertices. The last is the decomposed tree with bags as vertices.

Create factors. After the graph's structure is ready, the factors of the graph will be sampled for each pair of edges. There are seven sampling methods in Table 4.1. Generally, a chosen method produces all values of factors in a graph, though applying different sample methods for certain factors in a graph is feasible. Our code employs the former approach. The influence of different sample methods is tested in Section 4.5.

Compute the partition function. With the structure of the graph and parameters, an MRF is generated. Applying Algorithm 1, the partition function will be computed. In our code, we adapted the source code of the *query* function from a Python package *pgmpy*[41] because it does not use the log-sum-exponential strategy. With the inherent elimination order, such as the perfect ordering of the k-trees, the computing process of the partition function is fast compared to other elimination orders. The comparison is presented in Section 5.

Compute the most probable explanation. The method used to compute the most probable explanation is in Algorithm 2. As discussed, there are two steps in computing the most probable explanation. The function *max_marginal* in *pgmpy* helps to calculate the maximum probability in the joint distribution over all variables, which is the first step of computing the most probable explanation. In comparison, the second step is implemented in-house. It has the same question as the *query* function; therefore, the log-sum-exponential strategy is added here also. Overall, the source code of function *max_marginal* is modified and these two steps are combined in a function called *cal_mpe*.

Save MRFs and corresponding partition functions and most probable explanations to files. All files are saved in UAI format, which is also done by *pgmpy*. An MRF saved in a file with the ".uai" suffix obeys the following instructions. The first line is the type of the graph, "MARKOV" or "BAYESIAN". The second line indicates the number of vertices in the graph, followed by the cardinality of each variable. After that, the number of factors is presented, followed by lines with the number of random variables in this factor and the indices of these variables. The last part is the values of factors. The corresponding partition functions are saved in the file with the same name as the MRFs but added ".PR" suffix. The first line states the content of this file, which is "PR". The second line is the $\log 10$ value of the partition function for evaluation purposes. Last is the most probable explanation file. The file's structure begins with "MPE" and follows the number of vertices in the graph and

their assignments in the following line. The suffix of the most probable explanation files is ".MPE".

4.4 Results

There are five datasets with different kinds of MRFs and their partition functions and the most probable explanations, which have been uploaded to GitHub. The total number of MRFs is 392. The summary of some settings of the datasets is listed in Table 4.2. One requirement to generate MRFs in the datasets is that the running time of computing both the partition function and the most probable explanation of one network is at most 30 seconds (for the sake of time in this project).

	#N	Value of K	Sample method	Removal rate	#MRFs
Dataset 1	{100, 110, ..., 200}	[14, 18]	1	0	55
Dataset 2	{200, 400, ... 2000}	[6, 10]	1	0	50
Dataset 3	{300, 500, ..., 1300}	13	[1, 7]	0	42
Dataset 4	{95, 105, ..., 135}	[15, 19]	2	[0.1, 0.5]	125
Dataset 5	{900, 1100, ..., 1900}	[8, 11]	2	[0.1, 0.5]	120

Table 4.2: Datasets summary

The first dataset is a dataset with dense graphs. There are 55 MRFs in this dataset, with the number of vertices from 100 to 200. The range of treewidth is from 14 to 18. The sampling method is Dirichlet distribution, and the removal rate is set to be 0, which means this is a dataset of k-trees.

The second dataset is a dataset with sparse graphs. Compared to the first dataset, there are 50 larger MRFs in this dataset, with the number of vertices from 200 to 2000. The range of treewidth is from 6 to 10. Large graphs with more vertices and smaller treewidth result in sparse graphs. The sampling method and the removal rate are identical to the first dataset, which is Dirichlet distribution and 0, respectively.

The third one is designed to compare MRFs with different sampling methods. There are 42 MRFs with several vertices from 300 to 1300. The treewidth is fixed as 13, and no edges are removed. It consists of MRFs with the same k-tree structure but different sampling methods.

The fourth dataset is a collection of dense graphs with 95 to 135 vertices and treewidths from 15 to 19. As opposed to the dense graphs collection in Dataset 1, the MRFs in this dataset are not based on the k-trees but the partial k-trees because edges are removed. The removal rate is changed from 0.1 to 0.5. Besides, the sampling method is the exponential distribution. There are 125 graphs in total.

The last dataset is similar to Dataset 2 with sparse graphs. There are 120 MRFs with 900 to

1900 vertices and treewidths from 8 to 11. The sampling method is exponential distribution, and the range of the removal rate is from 0.1 and 0.5, which are the same as in Dataset 4.

The density of graphs is shown in Figure 4.3. Since there are many more graphs in Dataset 4 and Dataset 5 compared to Dataset 1 and 2, Figure 4.3 merely presents the density of graphs before removing edges. The darker colour represents the denser graphs, and the lighter colour represents the sparser graphs. The heatmaps of Datasets 1 and 4 have a more prominent dark blue area, while the heatmaps of Datasets 2 and 5 look almost all celadon. It reveals that Datasets 1 and 4 have dense graphs with fewer vertices and larger treewidth. In contrast, Datasets 2 and 5 have sparse graphs with a higher number of vertices and lower treewidth.

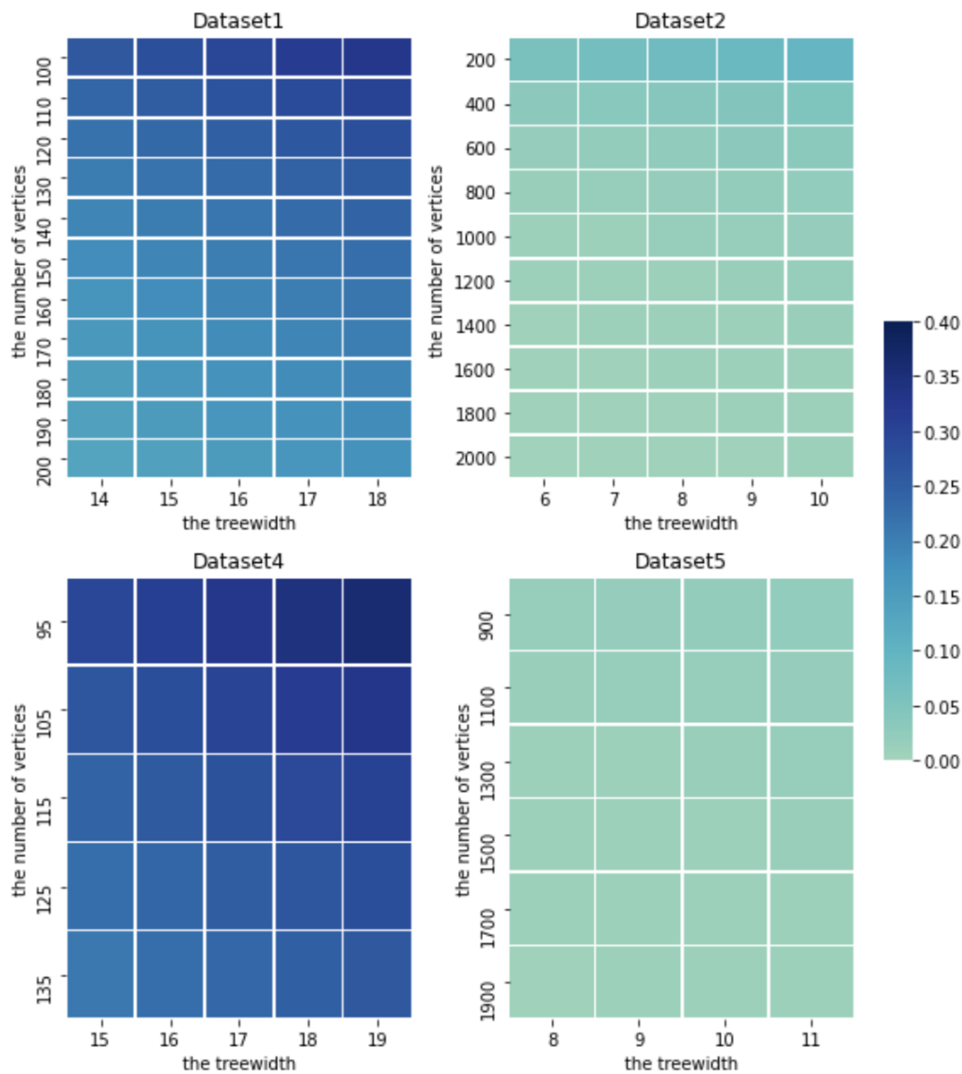


Figure 4.3: The graph density in Datasets 1, 2, 4 and 5

4.5 Evaluation

This section analyses the performance of our algorithm using the structure of k-trees, the sampling methods, and the removal rate. The datasets described in Section 4.4 will be used to execute and measure the running time change for different cases.

4.5.1 Structure of K-trees

4.5.1.1 Construction Comparison

As mentioned in Section 2.3, based on Definition 2 and 3, the k-trees can be generated recursively. The recursive process of generation of k-trees is described in Algorithm 3.

Algorithm 3 An algorithm for the Generation of k-trees

Require: $k > 0, N \geq k$ ▷ $N = \#vertices$

1: $V \leftarrow \emptyset$
2: $E \leftarrow \emptyset$
3: $k_cliques \leftarrow \emptyset$
4: $i \leftarrow 0$
5: $K \leftarrow \emptyset$
6: $ordering \leftarrow \emptyset$
7: **while** $i < k$ **do**
8: $\quad V \leftarrow V \cup \{N\}$
9: $\quad K \leftarrow K \cup \{N\}$
10: $\quad N \leftarrow N - 1$
11: $\quad ordering \leftarrow ordering \cup \{i\}$
12: $\quad i \leftarrow i + 1$
13: **end while**
14: $E \leftarrow E \cup \{(ei, eo) | ei, eo \in K, ei > eo\}$
15: $k_cliques \leftarrow k_cliques \cup \{K\}$
16: **while** $i \neq N$ **do**
17: $\quad V \leftarrow V \cup \{N\}$
18: $\quad K \leftarrow K \in k_cliques$
19: $\quad E \leftarrow E \cup \{(N, x) | x \in K\})$
20: $\quad ordering \leftarrow ordering \cup \{i\}$
21: $\quad$ **for** $x \in K$ **do**
22: $\quad\quad k_cliques \leftarrow k_cliques \cup \{K \setminus \{x\} \cap \{N\}\}$
23: $\quad$ **end for**
24: $\quad i \leftarrow i + 1$
25: **end while**
26: **return** $G = (V, E), k_cliques, ordering$

Before creating a k-tree, the value of parameter k and the number of vertices in this k-tree should be given. For a reasonable k-tree, the number of vertices in the graph should be greater than or equal to the value of parameter k. Then, from Line 1 to Line 5, the vertex set, the edge set, the set of all k-cliques, an index, the first k-clique of the graph, and the ordering are initialized, respectively. Lines 6 to 11 indicate that k vertices are chosen in the first loop

to form the first k-clique. In Line 12, add all edges in the first k-clique to the edge set. Since the final output is an undirected graph, the edges created in this graph are undirected. The edges are saved as the vertex with a smaller label, followed by the vertex with a greater label. Next, the first k-clique is combined with the set of all k-cliques. The following loop is to insert the rest vertices into the graph. In Line 16, randomly select a k-clique from the k-cliques set and connect the selected k-clique to the inserted vertex. Meanwhile, other k k-cliques are generated and combined with the k-cliques set. Eventually, until all vertices are inserted in the graph, return the vertex set and the edge set as a created k-tree and the ordering.

Table 4.3 illustrates the differences between k-trees generated by the recursive way and the bijective code. There are nine kinds of k-trees. They are compared based on the graph's maximum vertex degree, minimum vertex degree, longest path in the k-tree, shortest path in the k-tree, longest path in the skeleton of the k-tree, maximum clique, largest circle, and smallest circle. The last three features of the graphs are the same for the k-trees generated by both methods; therefore, they are excluded from the table. For each kind of k-tree, 100 k-trees are generated by a specific method. The values of features are average over the 100 k-trees. The maximum degree of a vertex is identical to the treewidth over all kinds of k-trees with two methods. In contrast, the k-trees generated by the recursive method have a higher average of the maximum degree of a vertex, a smaller average of the longest path, and a smaller average of the short path compared to the k-trees generated by the bijective code. Moreover, comparing the last column, since the recursive approach produces star-shaped graphs, the longest path in the skeleton of these k-tree is clearly shorter than that of graphs generated by the bijective code. In sum, the star shape of k-trees generated can be statistically verified by the features of graphs, and the bijective code can generate k-trees randomly and uniformly.

4.5.1.2 The Density of Graphs

The first evaluation is regarding the structure of k-trees and is performed on the first and second datasets. As mentioned before, the first dataset contains dense graphs, and the second dataset consists of sparse graphs.

Generally, in Figure 4.4 and Figure 4.5, the x-axis represents the number of vertices in a MRF, the y-axis represents the overall running time of solving both the partition function problem and the most probable explanation problem and different colored lines denote the k values of the k-trees, which is also termed the treewidth of graphs. It is easy to see the increasing trend of the running time of computing the partition function and the most probable explanation, which indicates that the running time is positively correlated to the number of vertices and the treewidth of MRFs. Recall that the time complexity of the variable elimination inference is $O(nk^{d+1})$. The larger the treewidth of the MRF is, the larger intermediate factor will be created during the elimination process and d will grow which leads to a significant increase in the running time. The influence of the treewidth is exponential, while the influence of the number of vertices is linear. Comparing the two patterns of the running time in two datasets, the gap between every two lines is clear in Figure 4.4 while in Figure 4.5, the colored lines are closely contiguous to each other. Therefore, we could find that the treewidth has a greater effect on the running time in dense graphs while having less effect in sparse graphs.
Next, analyze the running time behavior in dense graphs individually. In Figure 4.4, the computation time of MRFs with treewidth from 14 to 16 and the number of vertices from

#n	k	Method	Max d(v)	Min d(v)	Max p(G)	Min p(G)	Max p(S)
100	10	1	90.10	10	2.15	2.00	14.37
		2	83.82	10	2.97	2.00	27.99
100	20	1	98.02	20	2.00	1.68	13.41
		2	97.71	20	2.00	1.66	26.05
100	30	1	98.98	30	2.00	1.02	12.85
		2	98.99	30	2.00	1.01	23.77
500	10	1	390.60	10	3.00	2.00	21.99
		2	234.10	10	4.77	2.64	70.31
500	20	1	462.79	20	2.00	2.00	21.74
		2	409.89	20	3.01	2.00	70.62
500	30	1	481.50	30	2.00	2.00	21.79
		2	462.34	30	2.30	2.00	70.59
1000	10	1	722.22	10	3.02	2.00	25.19
		2	337.96	10	5.66	3.10	109.25
1000	20	1	893.29	20	2.00	2.00	25.04
		2	680.52	20	3.48	2.01	102.74
1000	30	1	944.08	30	2.00	2.00	25.15
		2	836.20	30	2.98	2.00	103.95

Table 4.3: The comparison of k-trees generated recursively (Method 1) or by bijective code (Method 2)

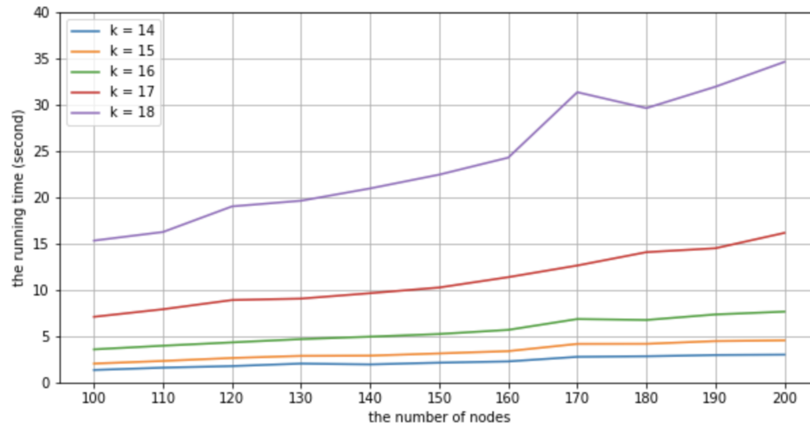


Figure 4.4: The running time changes according to the structure of k-trees in dense graphs

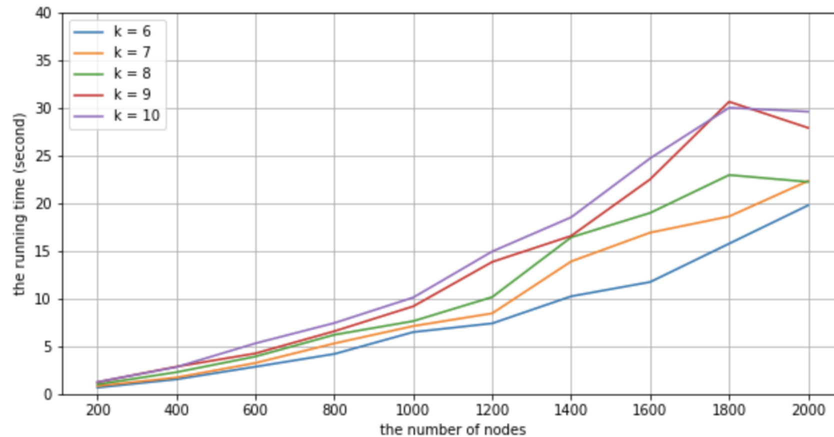


Figure 4.5: The running time changes according to the structure of k-trees in sparse graphs

100 to 200 rises slowly, in contrast to the lines with treewidth 17 and 18, which implies that the number of vertices has a greater influence on denser graphs such as graphs with higher treewidth. Moreover, there is a sharp spike in the line with k equals 18, while the difference in the running time is only 2 seconds. It could be explained by randomness.

On the other hand, for the dense graphs with the number of vertices as 200, the running time has raised over 30 seconds from the MRF with 14 treewidth to the MRF with 18 treewidth in Figure 4.4. However, in the case of the sparse graphs, for example, keeping the number of vertices as 2000, the difference between the running time of the MRF with 6 treewidth and with 10 treewidth is only increasing by 10 seconds in Figure 4.5. Therefore, we could conclude that addressing problems in dense graphs is harder than addressing problems in sparse graphs because expanding the same amount of the treewidth of the graphs, the dense graphs need more effort to solve queries than the sparse graphs.

4.5.2 Sampling Methods

The second part is to verify whether using different sampling methods to generate the parameters of the MRFs affects the computation time of handling the partition function task and the most probable explanation task. When the tasks are solved on the outside of the log domain, and the parameter values are less than 1 or too large, a range error of the values of the intermediate factors will be caused in large graphs because of the multiplication during the elimination process. Our algorithm adopted the log-sum-exp trick to solve this problem. As can be seen in Figure 4.6, there is no trend in these lines of the running time. The conclusion that the value of the parameters does not affect the calculation time is reached. It also implies the ability of MRFs to depict various scenarios with all kinds of relations.

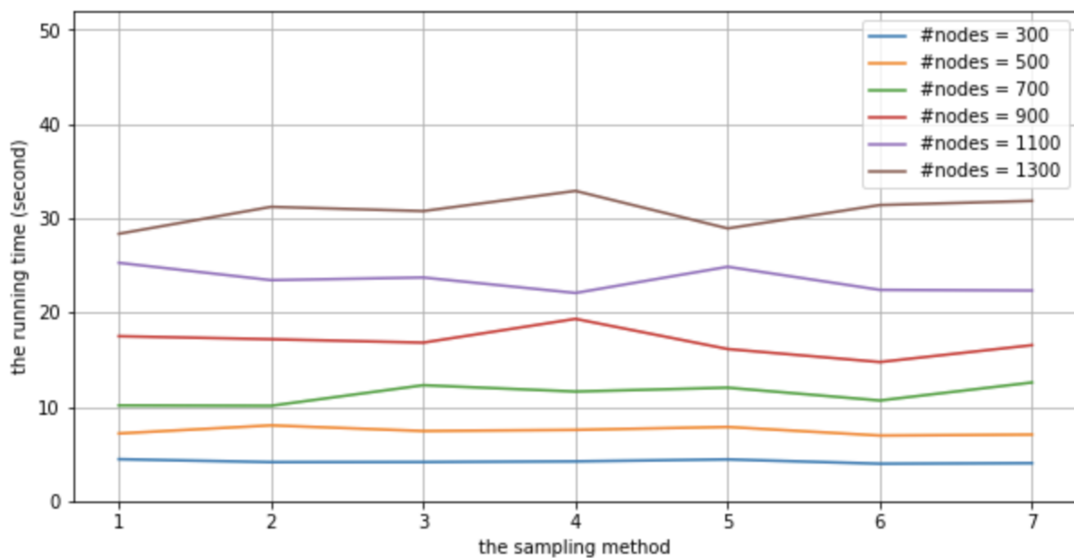


Figure 4.6: The comparison of the running time of MRFs using different sampling methods

4.5.3 The Removal Rate

The last evaluation is about the removal rate, and the experiments are done on Dataset 4 and Dataset 5. The difference between Dataset 4 and Dataset 5 is the density of graphs they contain, which are dense partial k-trees and sparse partial k-trees, respectively. For the removal strategy, there are two strategies in our code, which have been explained in Section 4.3. In this part, we used the strategy that removing edges depends on the degrees of their constituents. For vertices with large degrees, the edges connected to these vertices have a higher probability of being removed. The objective of applying this strategy is to challenge the Min-Neighbors approach to find the perfect orderings in the partial k-trees. Moreover, single vertices produced after removing edges will be excluded from the graphs because they have no relation with and impact on any other vertices in the graphs.

Overall, in Figure 4.7 and Figure 4.8, with the removal rate increasing, the running time of computing the partition function and the most probable explanation of the MRFs decreases. Recall the time complexity of the variable elimination inference. There are two possible reasons for the reduction in running time: either the number of vertices of the graph is

reduced, or the treewidth of the graph is reduced. Given that the removal rate has a limited impact on running time, it is more likely that the graph has fewer vertices. In addition, the amount of edges removed is related to the number of edges in the graph. The denser the graph, the more edges there are, and the number of edges removed will also increase accordingly. When the factor is reduced, the amount of computation in the elimination process is also reduced accordingly.

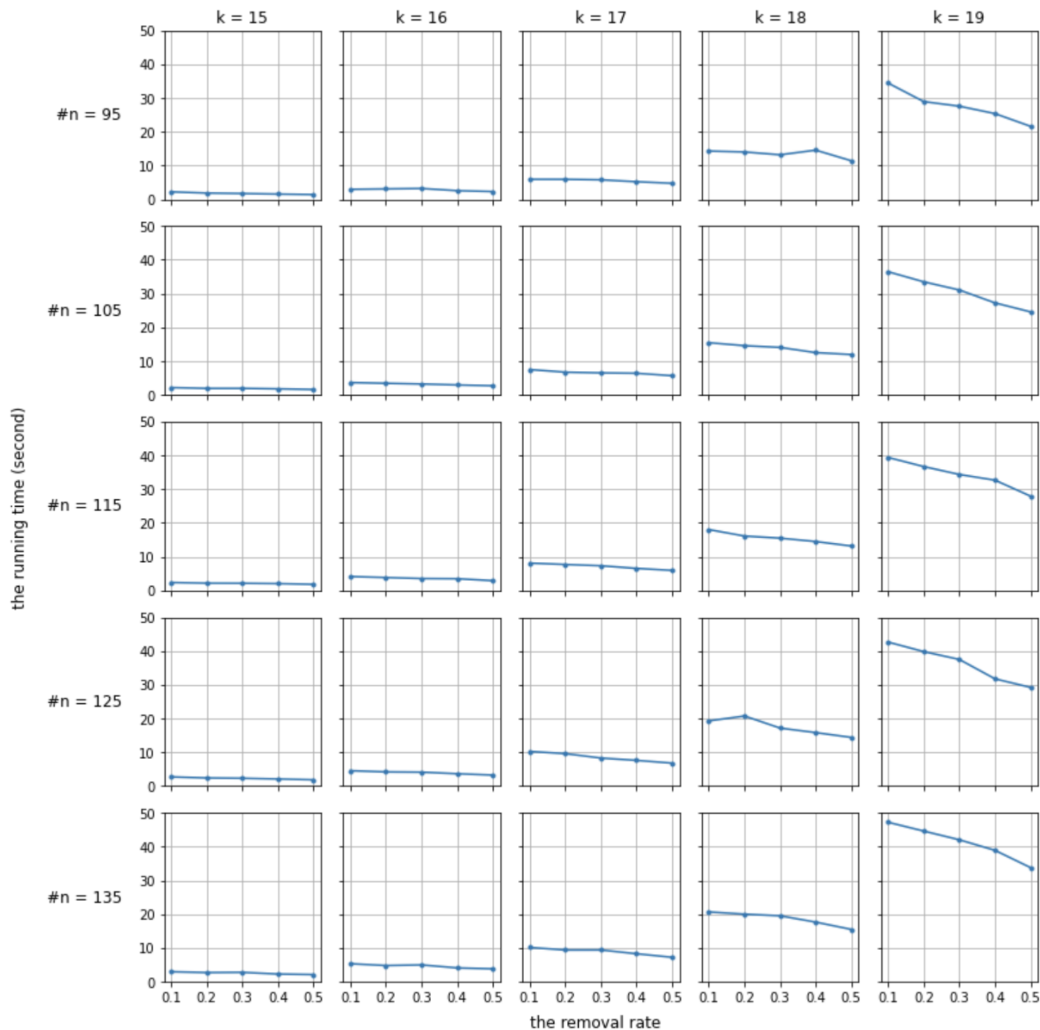


Figure 4.7: The running time with varied removal rate in dense graphs

Looking closely at Figure 1, for each row, when the number of vertices of the graph is the same, the effect of the removal rate on the running time is gradually visible as the treewidth increases. From the perspective of each column, when the treewidth is fixed and the number of vertices of the graph increases, we can find that when the tree width is 15, 16, 17, the change of running time is negligible as the removal rate increases; when the tree width is 18, the change of running time is slight; when the tree width is 19, the change of running time is obvious. Therefore, we can show that the removal rate has an effect on the runtime only when the treewidth is large. In comparison, the effect of treewidth on the running time will

be greater than the effect of the removal rate.

However, the running time demonstrates a different pattern in sparse graphs. Looking at each column in Figure 4.8, when the treewidth is fixed, the effect of removal rate on the running time increases as the number of vertices of the graph increases. For each row, when the number of vertices of the graph is constant, the fluctuation of the running time does not increase with the increase of the treewidth. This is similar to the observation we obtained in Section 4.5.1.2. In sparse graphs, the number of vertices of the graph has a greater impact on the final running time, compared to the treewidth.

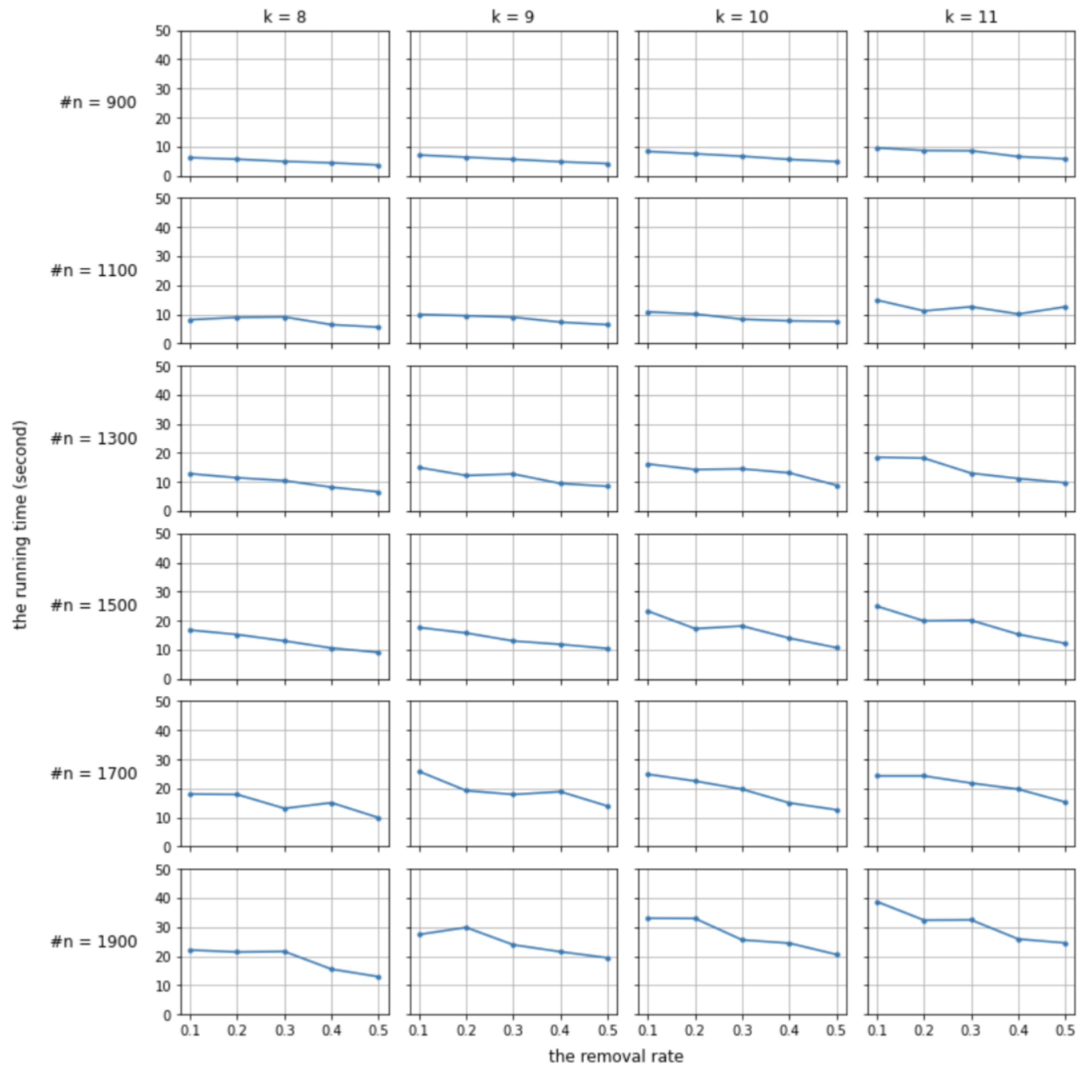


Figure 4.8: The running time with varied removal rate in sparse graphs

Chapter 5

Experiments

5.1 Treewidth Heuristic Algorithms

There are two heuristic algorithms to compute the treewidth of graphs mentioned in Section 2.3. They are Min-Neighbors and Min-Fill. In this experiment, we compared the performance of the two heuristic algorithms of treewidth on the partial k-trees generated by the bijective code and by the recursive way. The objective of this experiment is to estimate and compare the complication of the partial k-trees generated by the two methods empirically.

The partial k-trees used in this experiment are with the number of vertices from 100 to 600, the treewidth from 8 to 16, and the removal rate from 0.1 to 0.8. Overall, there are 480 graphs, and each k-tree generation method provides half of the graphs. For the removal strategy, we followed a similar setting in Section 4.5.3 for the same reason. However, different from the evaluation of the removal rate, in this experiment, we considered greater removal rates such as 0.6, 0.7, and 0.8. With such removal rates, the treewidth of the partial k-tree maybe decrease compared to the k-tree before removing edges. Hence, we configured that if the removal rate is greater than or equal to 0.6, the removal strategy will keep a (k+1)-clique in this graph. With this procedure, we can guarantee the treewidth of the partial k-tree with any removal rate. We keep a (k+1)-clique in the partial k-tree rather than a k-clique because the treewidth of a k-clique graph is $k - 1$.

The comparison of Figure 5.1 and Figure 5.2 shows that the two heuristics are comparable. Moreover, both heuristics are better at detecting the treewidth of the graph when the removal rate is higher than 0.6. However, when the removal rate is smaller than 0.6, both heuristics make certain mistakes, but the mistakes are more severe on the graphs generated by the bijective code. It indicates that the graphs generated by the bijective code are more complex than the graphs generated by the recursive way. Besides, it can be seen that when the treewidth is constant, expanding the number of vertices of the graph does not increase the difficulty of finding the treewidth for both heuristics. While the number of vertices of the graph is fixed, the larger the treewidth is, the greater the error committed by both heuristics on graphs generated by both methods. Therefore, in order to make our benchmark challenging, we would suggest using the bijective code to generate graphs with as large treewidth as possible.
Look at Figure 5.1 individually. When the removal rate is higher than 0.5, the performance of

the Min-Neighbors heuristic algorithm is outstanding in detecting the treewidth of the graphs generated by the recursive way. Even though the performance of the Min-Fill heuristic is not as good as the performance of the Min-Neighbors, the differences between the treewidth found by the Min-Fill heuristic and the actual treewidth are minor. In addition, when the removal rate is smaller, such as 0.1 and 0.2, the recursively generated graphs are relatively complicated for both heuristics.

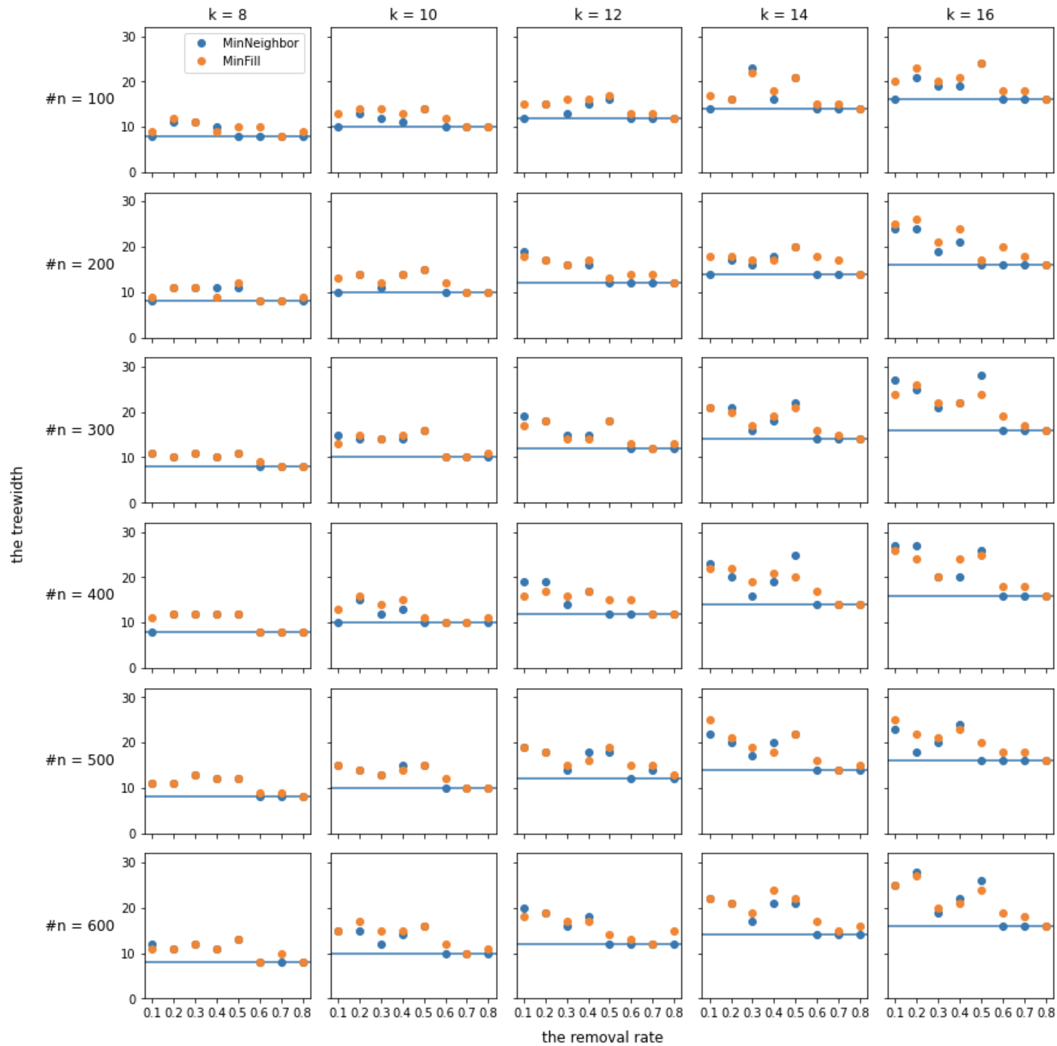


Figure 5.1: The performance of two treewidth heuristic algorithms on the partial k-trees generated by the recursive way

Unlike the patterns of the performance of the two heuristics in the graphs generated by the recursive way, in Figure 5.2, neither heuristic algorithms can detect the graph treewidth well when the removal rate is in [0.2, 0.7], in particular, when the removal rate is around 0.4 and 0.5. At this time, both heuristics calculate a larger value than the actual treewidth, even beyond twice the treewidth.

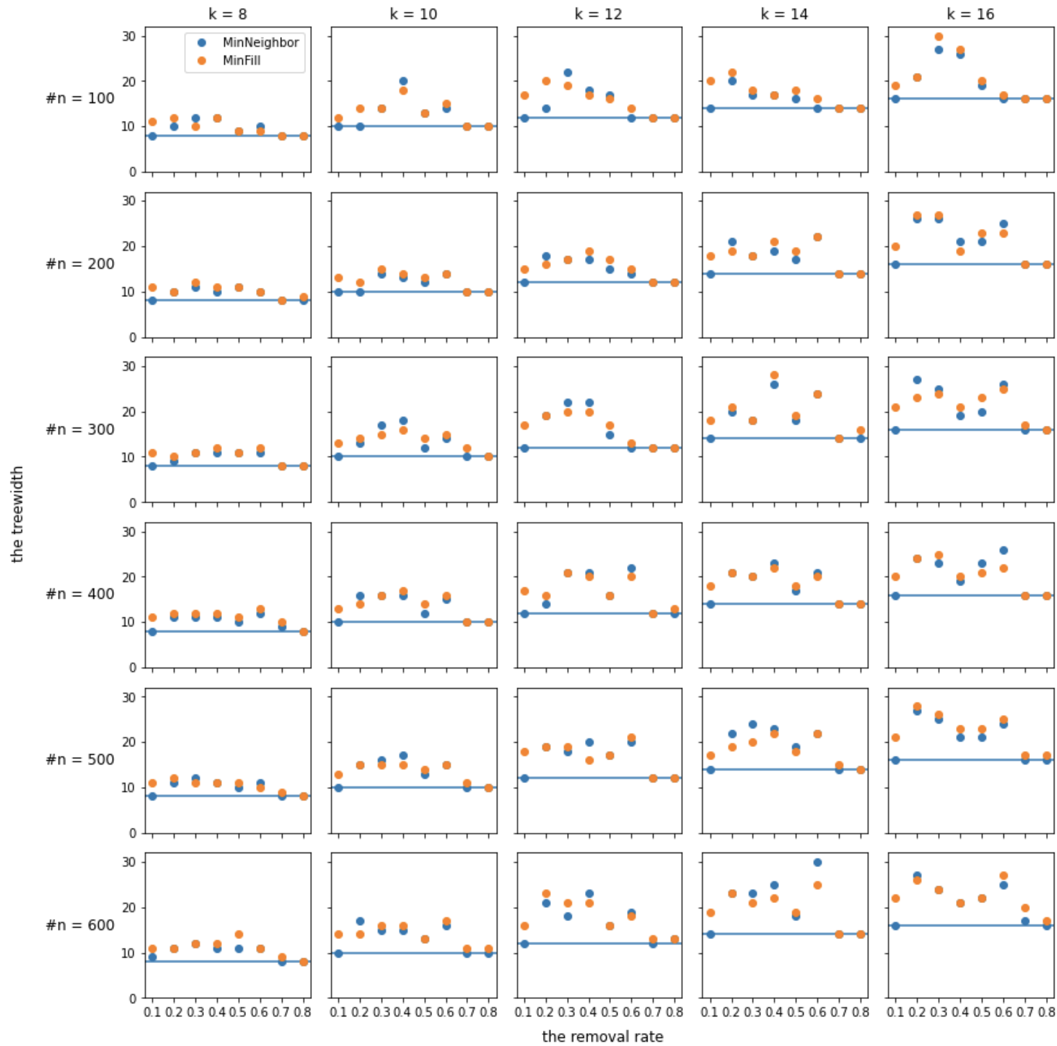


Figure 5.2: The performance of two treewidth heuristic algorithms on the partial k-trees generated by the bijective code

In the analysis, we did not count the exact error rate and the average error because of the randomness of the graphs generated by the two methods. The generated graph is different every time, and the exact error rate and the average error counted on one experiment cannot generalize to the next time.

5.2 Elimination Orders Comparison

The MRFs generated in the benchmark algorithm in Figure 4.2 have perfect orderings. With the perfect orderings, the variable elimination inference can provide the partition functions and the most probable explanations of these MRFs in a short time. Although finding the perfect elimination order is NP-hard, as we discussed in Section 3, there still are four heuristic methods commonly used for finding the best elimination order. Next, this second experiment is to verify the effect of these four methods on our generated MRFs, which are Min-Neighbors,

Min-Weight, Min-Fill and Weighted-Min-Fill. The Min-Neighbors and the Min-Fill are also used as the heuristic methods to find the treewidth of graphs in the previous section. The explanation of each criterion is as follows.

- **Min-Neighbors:** The cost of a vertex is the number of neighbors it has in the current graph
- **Min-Weight:** The cost of a vertex is the product of weights, domain cardinality, of its neighbors
- **Min-Fill:** The cost of a vertex is the number of edges that need to be added (fill in edges) to the graph due to its elimination
- **Weighted-Min-Fill:** The cost of a vertex is the sum of weights of the edges that need to be added to the graph due to its elimination, where a weight of an edge is the product of the weights, domain cardinality, of its constituent vertices

These four methods are implemented in the python package *pgmpy*. But they only work for Bayesian networks. Here introduces an approach to translate Bayesian networks to MRFs, and the two types of graphs have special relations in computing the partition function and the most probable explanation.

The steps to translate a MRF $MRF(V, E, \Phi)$ to a Bayesian network $BN(V', E', \Phi')$ are:

Step 1: $\forall \phi_i(X_i) \in \Phi$, create a new vertex $s_i \in \{0, 1\}$;

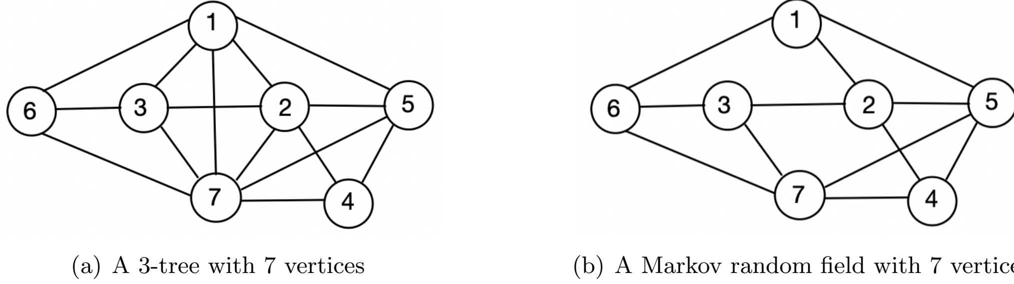
Step 2: Start with an empty graph $E' = \emptyset, \Phi' = \emptyset, V' = V \cup S$, where $S = \{s_1, ..., s_i\}$;

Step 3: $\forall x_j \in V, x_j$ has no parents. Let $p(x_j)$ be uniform and $\Phi' = \Phi' \cup \{p(x_j)\}$;

Step 4: $\forall \phi_i(X_i)$, insert each $x_j \in X_i$ as a parent of s_i and $E' = E' \cup \{(x_j, s_i)\}$;

Step 5: $\forall \phi_i(X_i)$, let $p(s_i = 0|X_i) = \frac{\phi_i(X_i)}{\sum_{X_i} \phi_i(X_i)}, p(s_i = 1|X_i) = 1 - \frac{\phi_i(X_i)}{\sum_{X_i} \phi_i(X_i)}$ and $\Phi' = \Phi' \cup \{p(s_i|X_i)\}$.

There is an example of a partial 3-tree MRF with 7 vertices in Figure 5.3(b) and the initial 3-tree is in Figure 5.3(a). With the translation approach, the corresponding Bayesian network is shown in Figure 5.4. The red vertices are s nodes referred before. Since we only consider binary random variables and the weight of each vertex is the same, the elimination orders generated based on the Min-Neighbors method and the Min-Weight method are identical. This is the same case for the Min-Fill method and the Weighted-Min-Fill method. Therefore, in this experiment, only the Min-Neighbors method and the Min-Fill method are taken into account. The elimination orders produced by the above two methods of the MRF in Figure 5.3(b) and the Bayesian network in Figure 5.4 are listed in Table 5.1. Even thought, these orderings are different, they are equivalent because the example is relatively simple. In fact, in the pilot experiment, with larger graphs, the methods from *pgmpy* provide worse elimination orders, which made variable elimination inference takes much more time in computing the partition function and the most probable explanation. Hence, the actual experiment was carried with the finding elimination orders methods implemented by ourselves.



(a) A 3-tree with 7 vertices

(b) A Markov random field with 7 vertices

Figure 5.3: A 3-tree and its partial 3-tree Markov random field with the removal rate as 0.2

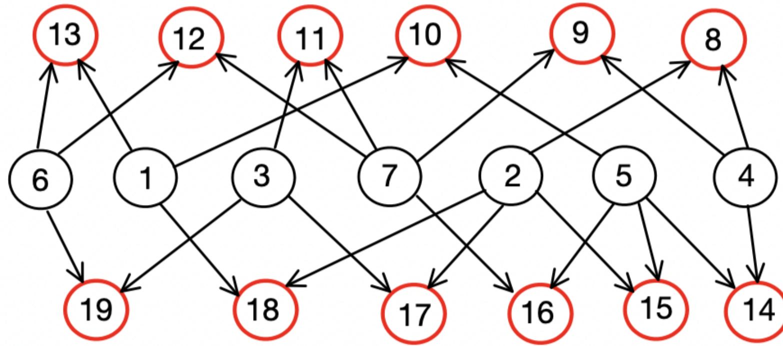


Figure 5.4: The corresponding Bayesian network according to the translation approach

The network	The method	The elimination order
The Bayesian network	Min-Neighbors	4, 3, 6, 1, 2, 5, 7
	Min-Fill	4, 3, 1, 6, 2, 5, 7
The Markov random field	Min-Neighbors	1, 3, 4, 2, 5, 6, 7
	Min-Fill	4, 3, 5, 1, 2, 6, 7
The perfect ordering	Default	6, 4, 5, 7, 3, 2, 1

Table 5.1: The comparison of elimination orders found by different methods

Next, we illustrated the connection between the partition function and the most probable explanation of the MRF and the probability of its parallel Bayesian network in equations. From Equation 5.1, the partition function of MRFs can be calculated as,

$$Z = \sum_{V} \prod_{j=1}^{|\Phi|} \phi_j(X_j) = p(s_i = 0 \quad \forall s_i \in S) \cdot |\Omega_V| \prod_{j=1}^{|\Phi|} \sum_{X_j} \phi_j(X_j),$$

where $|\Omega_V|$ represents the size of the joint distribution over all variables in the MRF.

$$
\begin{aligned}
p(s_i = 0 \ \ \forall s_i \in S) &= \sum_V \prod_{v \in V} p(v) \prod_{j=1}^{|\Phi|} p(s_j = 0|X_j) \\
&= \frac{\sum_V \prod_{j=1}^{|\Phi|} p(s_j = 0|X_j)}{|\Omega_V|} \\
&= \frac{1}{|\Omega_V|} \sum_V \prod_{j=1}^{|\Phi|} \frac{\phi_j(X_j)}{\sum_{X_j} \phi_j(X_j)} \\
&= \frac{\sum_V \prod_{j=1}^{|\Phi|} \phi_j(X_j)}{|\Omega_V| \prod_{j=1}^{|\Phi|} \sum_{X_j} \phi_j(X_j)}
\end{aligned}
\tag{5.1}
$$

The link between the most probable explanation of the MRF and the translated Bayesian network is proved in Equation 5.2. It indicates that the most probable explanation over all variables in the MRF is equivalent to the maximum a posterior inference over the group of shared variables in the translated Bayesian network.

$$
\begin{aligned}
&MAP(v, s_i = 0 \ \ \forall v \in V \text{ and } s_i \in S) \\
&\qquad = \operatorname*{argmax}_V \prod_{v \in V} p(v) \prod_{j=1}^{|\Phi|} p(s_j = 0|X_j) \\
&\qquad = \operatorname*{argmax}_V \frac{\prod_{j=1}^{|\Phi|} p(s_j = 0|X_j)}{|\Omega_V|} \\
&\qquad = \operatorname*{argmax}_V \prod_{j=1}^{|\Phi|} \frac{\phi_j(X_j)}{\sum_{X_j} \phi_j(X_j)} \\
&\qquad = \operatorname*{argmax}_V \prod_{j=1}^{|\Phi|} \phi_j(X_j) \\
&\qquad = MPE(V)
\end{aligned}
\tag{5.2}
$$

Then, the datasets used in this experiment are the subsets of four different datasets from Section 4.4. The details of these subsets are shown in Table 5.2. They are coming from Dataset 1 (dense graphs without removing edges), Dataset 2 (sparse graphs without removing edges), Dataset 4 (dense graphs with the removal rate from 0.1 to 0.5) and Dataset 5 (sparse graphs with the removal rate from 0.1 to 0.5), respectively. Each subset has exactly 30 graphs. There are 120 graphs in total.

	#N	Value of K	Sample method	Removing rate	#MRFs
Subset 1	{100, 110, ..., 150}	[14, 18]	1	0	30
Subset 2	{1000, 1200, ... 2000}	[6, 10]	1	0	30
Subset 3	{95, 105, 115}	[18, 19]	2	[0.1, 0.5]	30
Subset 4	{1500, 1700, 1900}	[8, 9]	2	[0.1, 0.5]	30

Table 5.2: The summary of datasets for experiments

At the beginning, the first testing focuses on the case that no edge removes. Figure 5.5 and Figure 5.6 show the running time change with the three elimination orders in different graphs. The three bars in each subplot of the two figures represent the running time of the variable elimination inference with the Min-Neighbors elimination order (MN), the Min-Fill elimination order (MF) and the perfect ordering (PO), respectively. It can be seen that the three bars have almost the same height, which means the Min-Neighbors method and the Min-Fill method are able to find the perfect orderings both in dense k-trees and in sparse k-trees. This is why the benchmarks generated by our algorithm are the partial k-trees rather than k-trees.

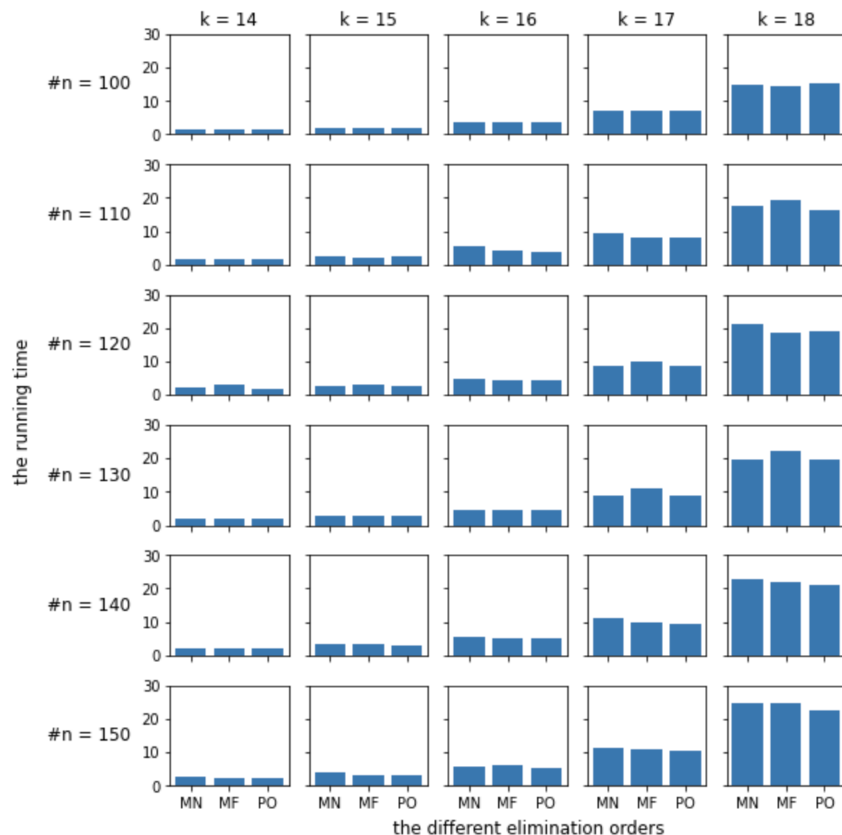


Figure 5.5: The running time comparison on Subset 1

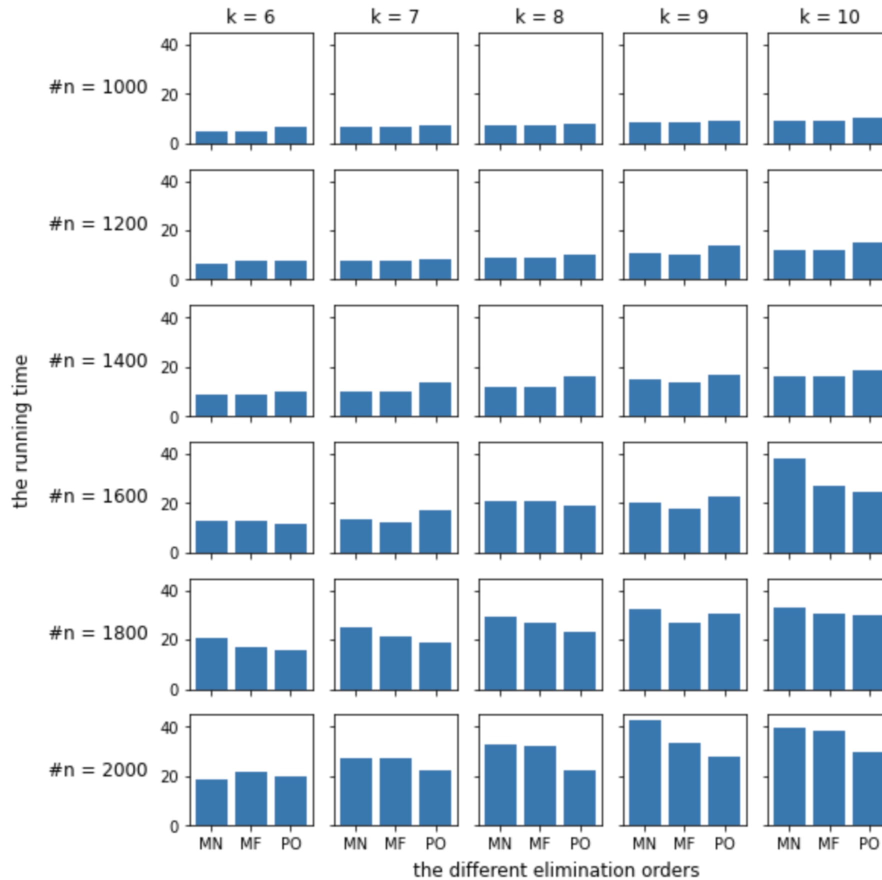


Figure 5.6: The running time comparison on Subset 2

The second part of this experiment concentrates on more challenging graphs. The performance of different elimination orders is shown in Figure 5.7 and Figure 5.8. When we took the testing on Subset 3, we set a running time bound as 60 seconds. If computing both the partition function and the most probable explanation of a MRF takes over 60 seconds, the computation process will be terminated automatically. The yellow bars in Figure 5.7 represent the terminated cases.

In Figure 5.7, it can be seen that the perfect orderings perform better than the other two methods, especially when the removal rate is 0.2 or 0.3. However, when the removing rate is 0.1, the Min-Neighbors method and the Min-Fill method can comparatively easily find the perfect orderings. The reasons might be that fewer edges are removed and the partial k-tree is highly similar to the k-tree.

However, analyzing the results of the testing on Subset 4, in Figure 5.8, it reveals that the large and sparse graphs are not difficult to solve. Because no difference in the performance of these three elimination orders was observed. Recall the time complexity of the variable elimination inference. In a graph, the running time of the variable elimination inference is determined by the size of the maximum factor generated during the elimination process exponentially and the number of vertices in polynomial. Checking the size of the maximum

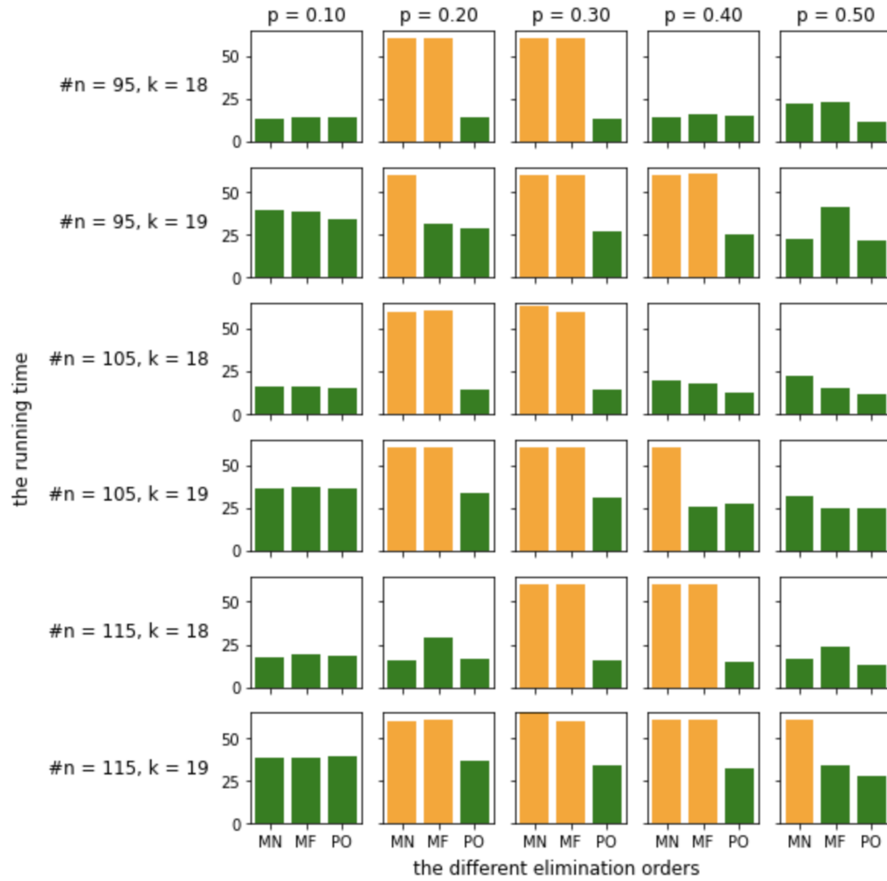


Figure 5.7: The running time comparison on Subset 3

factor generated during the elimination process, we found that with the perfect ordering, the value is the tree-width, while with the Min-Neighbors elimination order, the value is the same or slightly greater than the tree-width and with the Min-Fill elimination order, the value is one or two greater than the tree-width. Even with a larger size of the maximum factor, the total running time is nearly equal. Besides, computing the Min-Neighbors elimination order and the Min-Fill elimination order in sparse graphs needs considerable time compared to in dense graphs.

Overall, with the smaller number of vertices, the larger treewidth and 0.1 or 0.2 removal rate, our algorithm can generate complicated MRFs to benchmark certain inferences.

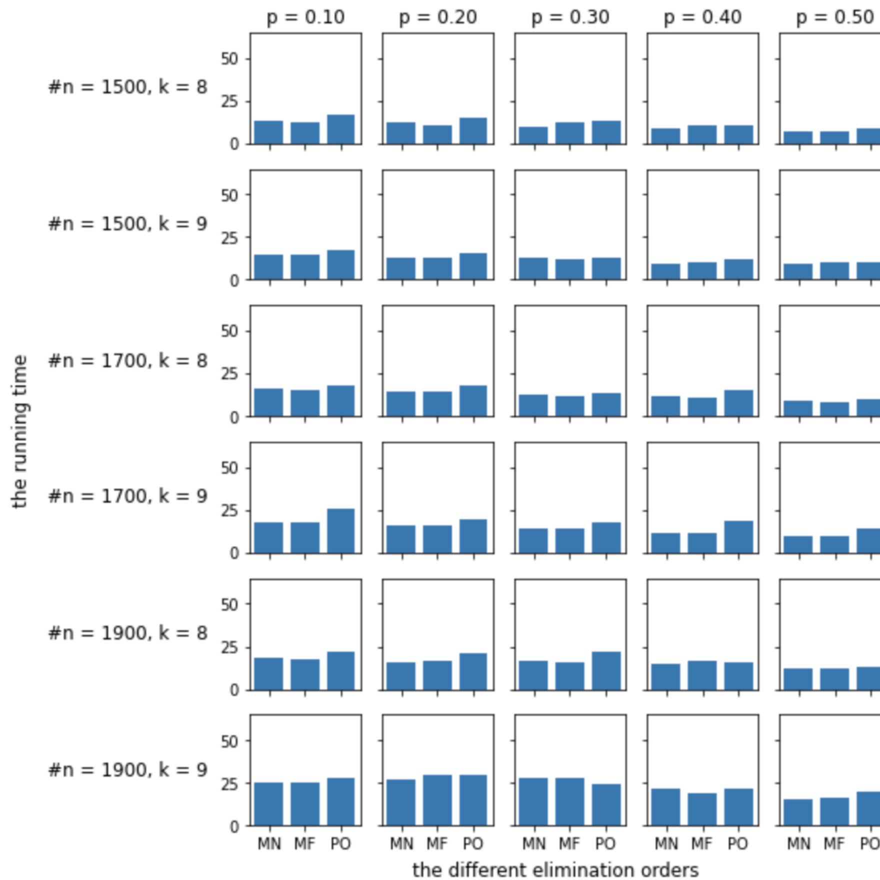


Figure 5.8: The running time comparison on Subset 4

5.3 Randomly Generate MRFs by Adding Edges

Apart from the way that firstly generating k-trees and then removing edges, we can create MRFs by adding edges to the graphs directly the other way around. The probability of adding edges is called the addition rate. If the addition rate is constant over all candidate edges, it is also the density of the graph. Consider a graph with n vertices and the addition rate is p. For each edge, there are only two states, exists or does not exist in the graph. The probability of the existence of an edge is exactly the addition rate p. The maximum number of edges in this graph is $\frac{n(n-1)}{2}$ and the number of graphs with n vertices is $2^{\frac{n(n-1)}{2}}$. The probability of randomly generating a graph with the addition rate p is $p^{\frac{n(n-1)}{2}}$ and the uniform probability of creating a graph is $2^{-\frac{n(n-1)}{2}}$. Therefore, if the addition rate is 50% ($p = \frac{1}{2}$), given the number of vertices, the graphs are uniformly generated. Thus, by modifying the addition rate, the probability of creating a graph by adding edges can be controlled.

However, it is very difficult to solve the partition function and the most probable explanation tasks of MRFs based on randomly generated graphs by adding edges because finding the treewidth of a graph is hard. Figure 5.9 illustrates the results of the experiment that, with the same number of vertices and the same density of graphs, the maximum size of the intermediate

factor produced during the variable elimination inference on k-trees or graphs generated by adding edges. In this experiment, the number of vertices is 100 and the range of the density of graphs is from 0.001 to 0.5. The comparison is among k-trees, partial k-trees and the graphs generated by adding edges. The maximum size of the intermediate factor on k-trees and partial k-trees with the fixed density can be calculated by the relation among the number of vertices, the treewidth, the removal rate and the density of graphs, while the maximum size of the intermediate factor on graphs generated by adding edges with the fixed density are calculated by Min-Neighbors approach and Min-Fill approach.

Figure 5.9 shows the increasing behavior of the maximum factor formed during the variable elimination with Min-Neighbors and Min-Fill elimination orders among denser graphs. While, with the same density, k-trees guarantees smaller factors compared to graphs generated by adding edges. For partial k-trees, to keep the same density of the graph, the treewidth increases. As mentioned in Section 4.5.1, the size of the maximum factor has exponential influence on the running time of the variable elimination inference. Hitherto, k-trees and partial k-trees with bounded treewidth have advantages over randomly generated MRFs by adding edges in the field of benchmark inferences of MRFs.

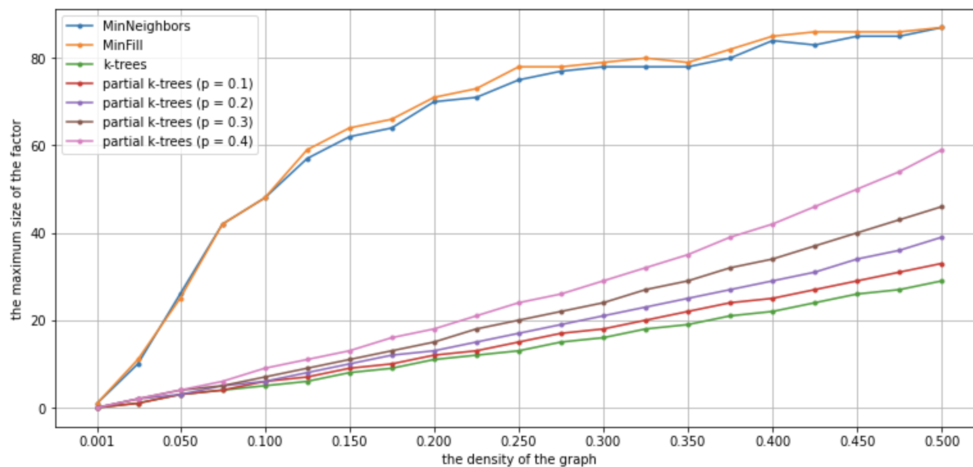


Figure 5.9: the relation between the size of the maximum factor and the density of the graph

5.4 Factors on K-cliques

In the previous sections, we have been discussing pairwise MRFs. In this section, we explored the case that the factors are on the cliques rather than only on the edges.

For a k-tree, it has $(n-k)k+1$ k-cliques. However, for a partial k-tree, since some edges are removed, the corresponding k-cliques are decomposed. Thus, we developed a procedure, Algorithm 4, to handle the cliques in the partial k-trees. The core idea is to check k-cliques whether any clique remains after removing edges separately. In the second loop, the rest edges are verified whether an edge is in a clique with over 2 vertices. If so, this edge should be excluded from the clique set. This is to say, the second loop in this procedure ensures that no repeated information is kept on the edges. After executing Algorithm 4, in most cases, the

number of factors in a non-pairwise graph is moderately smaller than its number of edges, which equals to the number of factors in the pairwise MRF.

Algorithm 4 Cliques of a k-tree after removing edges

Require: $\mathcal{K}, \hat{E}$ ▷ $\mathcal{K}$ is the set of k-cliques and $\hat{E}$ is the set of remaind edges after the edge removal
1: $C \leftarrow \emptyset$ ▷ C is the set of cliques in the partial k-trees
2: $Ce \leftarrow \emptyset$ ▷ Ce is the set of edges in C
3: $Re \leftarrow \emptyset$ ▷ Re is the set of rest edges in the graph but not in Ce
4: **for** $clique \in \mathcal{K}$ **do**
5: **if** $\forall v_i, v_j \in clique$ and $v_i \neq v_j,\ (v_i, v_j) \in \hat{E}$ **then**
6: $C \leftarrow C \cup \{clique\}$
7: $Ce \leftarrow Ce \cup \{(v_i, v_j) : \forall v_i, v_j \in clique \text{ and } v_i \neq v_j\}$
8: **else**
9: **if** $\forall v_i, v_j \in$ subset of $clique$ ($clique_{sub}$) and $v_i \neq v_j,\ (v_i, v_j) \in \hat{E}$ **then**
10: $C \leftarrow C \cup \{clique_{sub}\}$
11: $Ce \leftarrow Ce \cup \{(v_i, v_j) : \forall v_i, v_j \in clique_{sub} \text{ and } v_i \neq v_j\}$
12: $Re \leftarrow Re \cup \{(v_i, v_j) : \forall v_i, v_j \in clique, v_i \neq v_j \text{ and } (v_i, v_j) \in \hat{E} \setminus Ce\}$
13: **end if**
14: **end if**
15: **end for**
16: **for** $clique \in Re$ **do**
17: **if** $cliuqe \notin Ce$ **then**
18: $C \leftarrow C \cup \{clique\}$
19: $Ce \leftarrow Ce \cup \{clique\}$
20: **end if**
21: **end for**
22: **return** C

The experiment was carried out on a dataset with dense graphs, where the number of vertices over the dense graphs ranges from 100 to 150, the treewidth varied from 10 to 14 and the removal rate is set from 0 to 0.5. The total number of graphs is 180. Besides, the sampling method is the exponential distribution, and the inference method is the variable elimination with the perfect elimination ordering. The results are shown in Figure 5.10.

In Figure 5.10, the blue lines represent MRFs with factors on cliques, while the orange lines represent pairwise MRFs. The upper bound of the computation time is 60 seconds. All calculations are done within the requirement time limit. Except the observations of the known relations between the running time and the treewidth, and between the running time and the removal rate, the different discovery is that the blue lines have similar behavior to the orange line after removing edges. However, when the removal rate is 0, computing the partition function and the most probable explanation of MRFs with factors on cliques needs relatively more time, which is different from that of the pairwise MRFs. It might be because of the size of factors in the computation. For the factors on cliques case, in the process of eliminating one variable, there are k factors with k variables, while for pairwise MRFs, there are k factors with 2 variables.

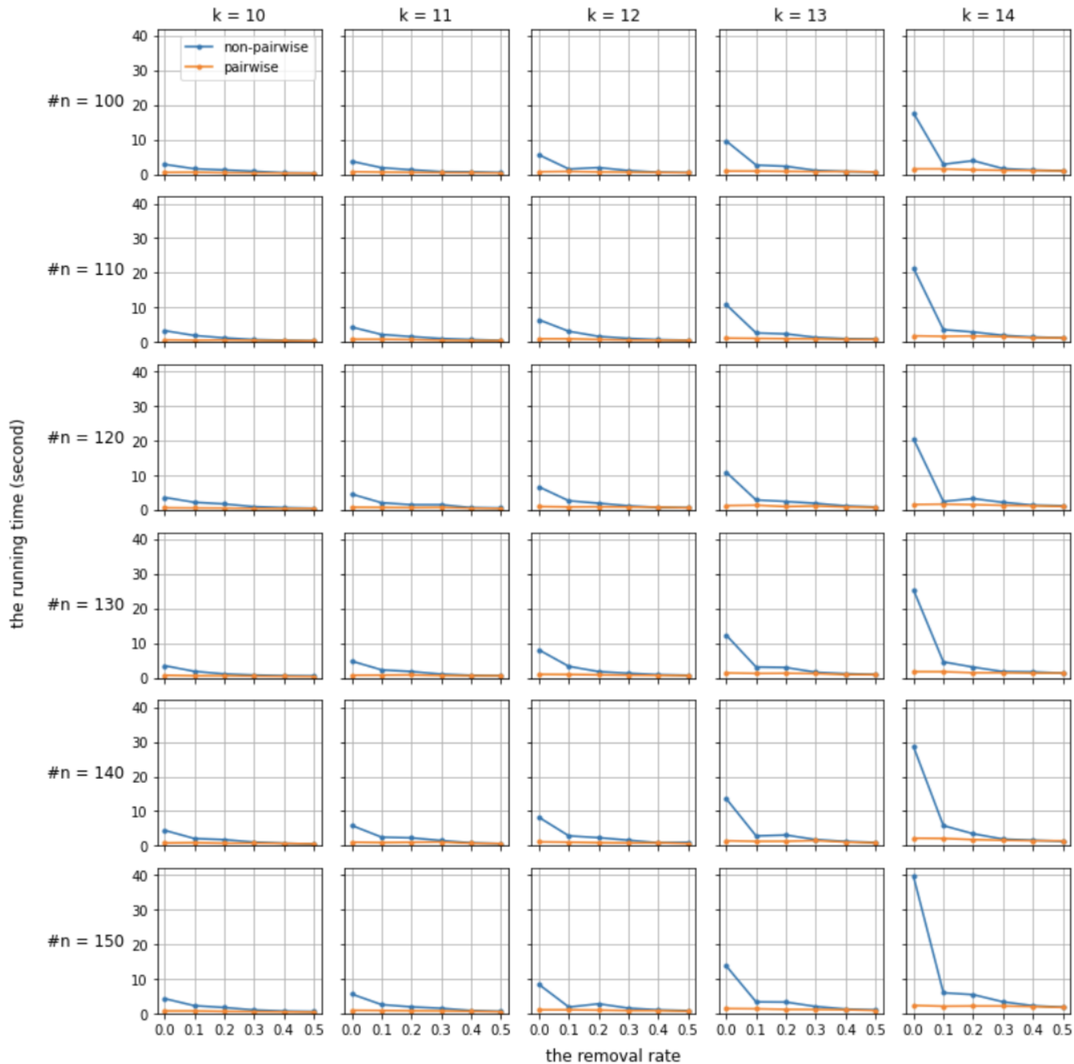


Figure 5.10: The experiment of factors on cliques

5.5 Benchmark the Exact Inferences

5.5.1 Junction Tree

The junction tree algorithm, also known as the clique tree, is an advanced algorithm to perform inferences in tree-type graphs. It includes converting a general graph to a tree and then exacting marginalization from it. The vertices of a junction tree are the clusters of vertices in the original graph. One advantage of the junction tree algorithm is that it overcomes an important shortcoming of the variable elimination inference. In the variable elimination inference, many intermediate factors generated for answering one query are useful for answering another. For example, during the process of answering the conditional marginal queries $p(a|b)$ and $p(c|b)$, the probability of variable b, $p(b)$ will be computed twice. The junction tree algorithm solves this problem by pre-computing intermediate factors and saving them in a tree structure. With the intermediate factors, the marginal quires can be answered

in $O(1)$ time. It is fast and efficient. Besides, reusing the intermediate factors avoids the waste of running time and space and decreases the computational complexity.

The junction tree algorithm consists of 7 stages, namely, moralizing the graph (only for Bayesian networks), triangulating the graph, forming the junction tree, pre-computing potentials and initializing the junction tree, selecting an arbitrary root node, performing message passing and evaluating required marginal potentials [42]. To be noted, there are two essential properties needed to be satisfied for a junction tree, i.e., family preservation and running intersection. The family preservation property requires that for each factor in the original graph, the variables in this factor should be in the same cluster, the same node in the junction tree and the running intersection property requires that for each variable in different clusters c_i and c_j, which are not directly connected, this variable should exist in the clusters in the path between c_i and c_j. The junction trees of graphs in Figure 5.3 are shown in Figure 5.11.

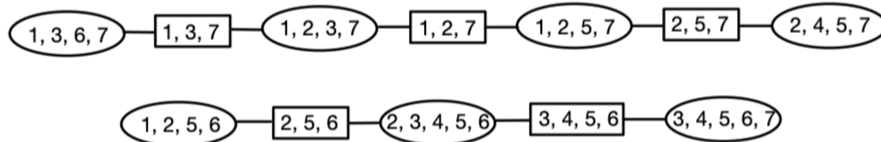


Figure 5.11: The junction tree of the 3-tree and Markov random field in Figure 5.3

The intermediate factors generated in the sum-product variable elimination and the max-product variable elimination are pre-computed by two methods in the junction tree, called the sum-product message passing and the max-product message passing.

The junction tree algorithm used in this experiment is implemented by *pgmpy* as belief propagation. The belief propagation inference has two functions. One is for the MAP query and the other is for the marginal query. They are working on MRFs, while the marginal query function returns the distributions of queried variables. The factors are normalized during the inference process. Therefore, it cannot solve the partition function task. Moreover, even though the MAP query function can return the most probable explanation over all variables, it fails to answer the query with over 10 variables because of the time-out.

Technically speaking, the junction tree algorithm is analogous to the variable elimination inference. The basic operations they used are identical, namely, multiplying factors and summing out variables. In the source code, The belief propagation creates a junction tree for the input graph and calls the variable elimination inference on a sub-junction tree of queried variables. Therefore, the variable elimination order is necessary. The performance of the default methods to determine the variable elimination order in *pgmpy* was tested in Section 5.2.

5.6 Benchmark the Approximate Inferences

5.6.1 Approximate Inference using Sampling

If the precision of the calculation in a controlled range is acceptable, computing the partition function and the most probable explanation by sampling is an alternative. Besides, consider-

ing that the time complexity and the space complexity of variable elimination inference are exponential with relative to the treewidth of the network and in practice the graphs with large treewidth are unavoidable, the approximate inference using sampling is preferable in such situation.

The approximate inference using sampling in *pgmpy* is exclusively for Bayesian networks. Therefore, the approach to translating MRFs to Bayesian networks discussed in Section 5.2 is applied. The concept of this inference is to sample instances from graphs based on the conditional probability distributions and then approximate the probabilities of variables by the frequencies of sampled instances.

This inference has good performance in small graphs. However, during the experiment, we found that the approximate inference using sampling fails to solve the partition function task and the most probable explanation task of large graphs because of the sampling size. In Equation 5.1, before computing the partition function of the MRFs, the joint distribution of all s vertices is required. While the size of the joint distribution of all s vertices is dependent on the number of s vertices and their cardinalities. To be precise, the number of s vertices is the number of factors in the MRF, and the size of the joint distribution is the product of cardinalities of all s vertices, which is exponential in the number of s vertices. Obtaining the joint distribution of all s vertices from the sampling demands that each instance in the joint distribution should at least show up in the sample set once. Since small probabilities of instances exist, the size of samples is required to be enormous.

One reason for using the variable elimination with the perfect ordering is to avert the computation of the joint distribution. But computing the joint distribution is inevitable for the approximate inference with sampling to solve the partition function task and the most probable explanation task. Thus, approximate inference using sampling is defective and not recommended when answering queries.

Chapter 6

Discussion

This algorithm can generate benchmarks for MRF inferences and tree decomposition tasks. The steps to create MRFs are as follows. Initially, the input parameter must be collected, including the number of vertices, the treewidth, and the removal rate of the wanted graph. Additionally, a sampling method must be chosen to generate the parameters of the graph, and an edge removal strategy must be decided on to construct the partial k-trees. With the partial k-tree and factors, a pairwise MRF is randomly generated, and its perfect ordering is provided. The partial k-tree can be saved in a ".gr" file as a treewidth task instance; the tree decomposition will be created and saved with its perfect ordering in the ".td" file for benchmarking. Next, utilize the variable elimination inference and the perfect order to efficiently compute the correct partition function and the most probable explanation of this MRF as a benchmark. Finally, the graph and queries are saved in the UAI file format. This code is readily available and can be used to generate as many MRFs and benchmark datasets as needed.

Two unique parts make up this algorithm. The first is that our k-trees are all decoded with a bijective code. Because bijective codes can randomly construct uniformly in the domain of bijective codes, the k-trees are obtained from uniform distributions given the number of vertices and treewidth; thus, the graphs and benchmarks generated are guaranteed to be unbiased. The second point is about the strategy for removing edges. There are two strategies employed in this algorithm: removing edges randomly and removing edges conditionally. Heuristic algorithms are used for verification, and several experiments are run. When we randomly remove edges, heuristics can easily find the best orderings of what we wish to hide. In this case, the benchmarks we generate are too easy for the inference algorithm to function as a test of the inference algorithm. The degrees of vertices are made as close to the average as possible to challenge these heuristics. Therefore, the edges with vertices of a degree greater than k are removed first, as they are the edges of non-leaf nodes in the k-trees. This strategy has proven effective.

Additionally, the analysis of the *pgmpy* package revealed many areas for improvement. The first is the module that reads the graph. When there are more than 150 edges in a graph, *pgmpy* cannot read it correctly due to a grammar error. The second is that the log-sum-exp trick is not applied in the variable elimination inference module, which leads to the fact that the inner function of the variable elimination inference cannot handle the large graph

with relatively large or small parameters. Moreover, in variable elimination inference, the function that calculates the most probable explanation does not utilize the elimination order but calculates the joint distribution. Therefore, this method cannot return the result for large graphs in a reasonable time-bound. Finally, *pgmpy* does not have a method for calculating the elimination order for MRFs. Therefore, this additional functionality had to be implemented in-house within the structure of *pgmpy*.

As mentioned in the literature review, although only pairwise MRFs are generated, these pairwise Markov networks can represent more complex networks and are used in many practical problems. However, the intention was to construct benchmark datasets with the algorithm to validate inferences for MRFs.

Chapter 7

Conclusions

7.1 Contributions

This project develops an algorithm to randomly generate MRFs based on a partial k-trees structure; this structure includes pairwise factors, the number of vertices, the treewidth, and the removal rate of edges. Besides, the perfect orderings of generated MRFs are known by us to compute the query results effectively. Therefore, the datasets of MRFs generated by our algorithm can be used as benchmarks to evaluate inferences on MRFs. Moreover, seven sampling methods are used to create parameters for the MRFs, such as the values of factors in different ranges.

In this project, the partition function task and the most probable explanation task are chosen to test and compare the performance of inferences on the datasets of MRFs generated by our algorithm. The variable elimination inference is beneficial and efficient since the perfect orderings of the graphs are included in the data. While finding the ideal ordering is still NP-hard, solving the partition function task and the most probable explanation task by the variable elimination inference on our datasets is extremely difficult without the ordering information. Section 5.2 shows that the intuitive methods for finding elimination orders are ineffective on these generated datasets. In addition, the junction tree algorithm, an exact inference, the approximate inference by sampling, and an approximate inference are all unproductive for these datasets.

Furthermore, generating custom shapes of graphs is possible. Configuring the algorithm's inputs, namely, the number of vertices, the treewidth, and the removing rate, can alter the density of the graphs. The queries on the graphs become relatively complex with the smaller number of vertices, the larger treewidth, and the removing rate of 0.2. Hence, setting the removing rate at 0.2 is recommended for challenging benchmarks. All code is presented in the Appendix A. The generation process is fast, and the reader can generate as many graphs as they want. The correct answers of queries utilized to verify the performance of inference algorithms for MRFs can be computed by the variable elimination inference with the perfect orderings rapidly.

7.2 Future work

When creating MRF datasets as benchmarks for inferences of MRFs, the run time of computing the correct answers for the queries is an important aspect and needs to be considered. However, the time and space complexity of answering the queries is exponential with respect to the treewidth of the partial k-trees, even when the perfect ordering is known via the variable elimination inference. With the time limits, the treewidth of MRFs generated by our algorithm cannot be very large. However, the large graphs are worthwhile in practice. Hence, future focus will be on developing MRFs with large treewidth while the queries from these MRFs can be answered correctly in a limited time. Since the computation of queries is an NP-hard problem, more advanced techniques or tricks must be adopted.

If the computation of queries is unneeded, for example, only randomly generating MRFs, the speed of the generation process is fast. Except for UAI tasks, the PACE competition [10] considers the parameterized algorithms in practice as essential in the domain of graph theory. It is held annually with topics including cluster editing, treedepth, vertex cover, and treewidth. This algorithm can generate graphs with the edge removal rate over 0.8 as benchmarks to evaluate the treewidth algorithms. Some ideas to increase the complexity of our graphs include modifying the structure of k-trees and improving the strategy of removing edges.

Bibliography

[1] Stan Z Li. *Markov random field modeling in image analysis*. Springer Science & Business Media, 2009. 1

[2] Martin J Wainwright, Michael I Jordan, et al. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2):1–305, 2008. 1

[3] George EP Box and George C Tiao. *Bayesian inference in statistical analysis*. John Wiley & Sons, 2011. 1

[4] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009. 1, 5, 6, 7, 15

[5] Li Hongmei, Hao Wenning, Gan Wenyan, and Chen Gang. Survey of probabilistic graphical models. In *2013 10th Web Information System and Application Conference*, pages 275–280. IEEE, 2013. 1

[6] Rina Dechter, Alexander Ihler, Vibhav Gogate, Junkyu Lee, and Bobak Pezeshki. Introduction to uai 2022 competition. `https://uaicompetition.github.io/uci-2022/information/introduction/`. 1

[7] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015. 1

[8] Nathalie Peyrard, Marie-Josée Cros, Simon de Givry, Alain Franc, Stéphane Robin, Régis Sabbadin, Thomas Schiex, and Matthieu Vignes. Exact and approximate inference in graphical models: variable elimination and beyond. *arXiv preprint arXiv:1506.08544*, 2015. 1

[9] Holger Dell, Thore Husfeldt, Bart MP Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A Rosamond. The first parameterized algorithms and computational experiments challenge. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017. 2

[10] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *12th international symposium on parameterized and exact computation (IPEC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 2, 52

[11] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. 2

[12] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete applied mathematics*, 23(1):11–24, 1989. 2

[13] Peter Clifford. Markov random fields in statistics. *Disorder in physical systems: A volume in honour of John M. Hammersley*, pages 19–32, 1990. 5

[14] Luis Enrique Sucar. Probabilistic graphical models. *Advances in Computer Vision and Pattern Recognition. London: Springer London. doi*, 10(978):1, 2015. 5

[15] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012. 5, 13

[16] L. W. Beineke and R. E. Pippert. Properties and characterizations of k-trees. *Mathematika*, 18(1):141–151, 1971. 10

[17] Donald J Rose. On simple characterizations of k-trees. *Discrete mathematics*, 7(3-4):317–322, 1974. 10

[18] Dandan Fan, Sergey Goryainov, Xueyi Huang, and Huiqiu Lin. The spanning k-trees, perfect matchings and spectral radius of graphs. *Linear and Multilinear Algebra*, pages 1–12, 2021. 10

[19] Saverio Caminiti, Emanuele G Fusco, and Rossella Petreschi. Bijective linear time coding and decoding for k-trees. *Theory of Computing Systems*, 46(2):284–300, 2010. 10, 14, 18

[20] Cédric Bentz. Weighted and locally bounded list-colorings in split graphs, cographs, and partial k-trees. *Theoretical Computer Science*, 782:11–29, 2019. 10

[21] Donald J Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In *Graph theory and computing*, pages 183–217. Elsevier, 1972. 10, 19

[22] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001. 10

[23] Frank Harary and Edgar M Palmer. On acyclic simplicial complexes. *Mathematika*, 15(1):115–122, 1968. 10

[24] Cassio P de Campos. Almost no news on the complexity of map in bayesian networks. In *International Conference on Probabilistic Graphical Models*, pages 149–160. PMLR, 2020. 13

[25] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan kaufmann, 1988. 13

[26] Michael I Jordan. An introduction to probabilistic graphical models, 2003. 13

[27] Rahul Singh, Isabel Haasler, Qinsheng Zhang, Johan Karlsson, and Yongxin Chen. Inference with aggregate data in probabilistic graphical models: An optimal transport approach. *IEEE Transactions on Automatic Control*, 2022. 13

[28] Javad Forough and Saeedeh Momtazi. Sequential credit card fraud detection: A joint deep neural network and probabilistic graphical model approach. *Expert Systems*, 39(1):e12795, 2022. 13

[29] Jaime S Ide and Fabio G Cozman. Random generation of bayesian networks. In *Brazilian symposium on artificial intelligence*, pages 366–376. Springer, 2002. 14, 16

[30] Guy Melançon, Isabelle Dutour, and Mireille Bousquet-Mélou. Random generation of directed acyclic graphs. *Electronic Notes in Discrete Mathematics*, 10:202–207, 2001. 14

[31] Tom Britton, Maria Deijfen, and Anders Martin-Löf. Generating simple random graphs with prescribed degree distribution. *Journal of statistical physics*, 124(6):1377–1397, 2006. 14

[32] Mohsen Bayati, Jeong Han Kim, and Amin Saberi. A sequential algorithm for generating random graphs. *Algorithmica*, 58(4):860–910, 2010. 14

[33] Louis-Claude Canon, Mohamad El Sayah, and Pierre-Cyrille Héam. A comparison of random task graph generation methods for scheduling problems. In *European Conference on Parallel Processing*, pages 61–73. Springer, 2019. 14

[34] KiJung Yoon, Renjie Liao, Yuwen Xiong, Lisa Zhang, Ethan Fetaya, Raquel Urtasun, Richard Zemel, and Xaq Pitkow. Inference in probabilistic graphical models by graph neural networks. In *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, pages 868–875. IEEE, 2019. 14

[35] Cassio Polpo De Campos and Fabio Gagliardi Cozman. The inferential complexity of bayesian and credal networks. In *IJCAI*, volume 5, pages 1313–1318. Citeseer, 2005. 15

[36] Hiroshi Ishikawa. Transformation of general binary mrf minimization to the first-order case. *IEEE transactions on pattern analysis and machine intelligence*, 33(6):1234–1249, 2010. 15

[37] Alexander Fix, Aritanan Gruber, Endre Boros, and Ramin Zabih. A hypergraph-based reduction for higher-order binary markov random fields. *IEEE transactions on pattern analysis and machine intelligence*, 37(7):1387–1395, 2014. 15

[38] Ghada Trabelsi, Philippe Leray, Mounir Ben Ayed, and Adel M Alimi. Benchmarking dynamic bayesian network structure learning algorithms. In *2013 5th International Conference on Modeling, Simulation and Applied Optimization (ICMSAO)*, pages 1–6. IEEE, 2013. 16

[39] Mouna Ben Ishak, Philippe Leray, and Nahla Ben Amor. Probabilistic relational model benchmark generation: Principle and application. *Intelligent Data Analysis*, 20(3):615–635, 2016. 16

[40] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006. 17

[41] Ankur Ankan and Abinash Panda. pgmpy: Probabilistic graphical models using python. In *Proceedings of the 14th Python in Science Conference (SCIPY 2015)*. Citeseer, 2015. 22

[42] David Kahle, Terrance Savitsky, Stephen Schnelle, and Volkan Cevher. Junction tree algorithm. *Stat*, 631, 2008. 46

Appendix A

Code of the benchmark algorithm in Python

A.1 Create a Markov random field as a benchmark for the UAI competition

```
from pgmpy.models import MarkovNetwork
from pgmpy.readwrite import UAIWriter
from pgmpy.factors.discrete import DiscreteFactor
from itertools import combinations
from functools import reduce
from tqdm import tqdm
import numpy as np
import time
import math
import copy
import os
import warnings
warnings.filterwarnings(action='ignore')

# https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.
    html
def random(idx, n):
    """
    inputs: idx: select the random method to generate parameters of a factor
            n: the number of parameters needed
    output: the parameters of a factor
    """
    if idx == 1:
        # Draw samples from the Dirichlet distribution.
        return np.random.dirichlet(np.ones(n), 1)
    elif idx == 2:
        # Draw samples from an exponential distribution.
        return np.random.exponential(1, n)
    elif idx == 3:
        # Draw samples from a Beta distribution.
        return np.random.beta(2, 3, n)
    elif idx == 4:
        # Draw samples from a chi-square distribution.
        return np.random.chisquare(2, n)
```

```
    elif idx == 5:
        # Draw samples from a uniform distribution.
        return np.random.uniform(10, 100, n)
    elif idx == 6:
        # Draw samples from a uniform distribution over [0, 1).
        return np.random.rand(n)
    elif idx == 7:
        # Draw random samples from a log-normal distribution.
        return np.random.lognormal(0, 1, n)

def preparation(graph, variables, evidence, elimination_order, joint = True):
    """
    inputs: graph: a pgmpy MarkovNetwork
            variables: variables that need to be eliminated
            evidence: the evidence of this query
            elimination_order: the elimination order
            joint: True: return the query over all input variables;
                False: return the queries of variables individually
    outputs: working_factors: a dictionary of vertices and their
                factors using in the elimination process
             elimination_order: the order the vertices in the elimination
                 process
    """

    # get working factors
    working_factors = {
        node: {(DiscreteFactor(factor.scope(), factor.cardinality, np.log(
            factor.values)), None) for factor in graph.get_factors(node)} # set
        for node in graph.nodes}

    to_eliminate = (
        set(graph.nodes)
        - set(variables)
        - set(evidence.keys() if evidence else [])
    )

    # get elimination order
    # Step 1: If elimination_order is a list, verify it's correct and return.
    # Step 1.1: Check that not of the `variables` and `evidence` is in the
        elimination_order.
    if hasattr(elimination_order, "__iter__") and (not isinstance(
        elimination_order, str)):
        if any(var in elimination_order for var in set(variables).union(set(
            evidence.keys() if evidence else []))):
            raise ValueError(
                "Elimination order contains variables which are in"
                " variables or evidence args"
            )
        # Step 1.2: Check if elimination_order has variables which are not in
            the model.
        elif any(var not in graph.nodes() for var in elimination_order):
            elimination_order = list(filter(lambda t: t in graph.nodes(),
                elimination_order))

        # Step 1.3: Check if the elimination_order has all the variables that
            need to be eliminated.
        elif to_eliminate != set(elimination_order):
```

```
            raise ValueError(
                f"Elimination order doesn't contain all the variables"
                f"which need to be eliminated. The variables which need to"
                f"be eliminated are {to_eliminate}")

    # Step 2: If elimination order is None or a Markov model, return a random
        order.
    elif elimination_order is None:
        elimination_order = to_eliminate
    else:
        elimination_order = None

    # marginal
    if not variables:
        variables = []

    common_vars = set(evidence if evidence is not None else []).intersection(
        set(variables if variables is not None else [])
    )
    if common_vars:
        raise ValueError(f"Can't have the same variables in both `variables`
            and `evidence`. Found in both: {common_vars}")

    # variable elimination
    # Step 1: Deal with the input arguments.
    if isinstance(variables, str):
        raise TypeError("variables must be a list of strings")
    if isinstance(evidence, str):
        raise TypeError("evidence must be a list of strings")

    # Dealing with the case when variables is not provided.
    if not variables:
        all_factors = []
        for factor_li in graphs.get_factors():
            all_factors.extend(factor_li)
        if joint:
            return False, factor_product(*set(all_factors))
        else:
            return False, set(all_factors)

    return working_factors, elimination_order

def factor_product(*args):
    """
    input: factors
    output: the product of input factors
    """

#     if not all(isinstance(phi, BaseFactor) for phi in args):
#         raise TypeError("Arguments must be factors")
#     # Check if all of the arguments are of the same type
#     elif len(set(map(type, args))) != 1:
#         raise NotImplementedError(
#             "All the args are expected to be instances of the same factor
    class."
#         )
#     if len(args) == 0:
```

```
#           print(args)

    return reduce(lambda phi1, phi2: phi1 + phi2, args)

# Compute the partition function (PR) task
def marginalize(phi_o, variables, inplace = True):
    """
    marginalizes the factor with respect to `variables`.

    inputs: phi_o: the intermediate factor
            variables: variables that need to be eliminated from the
                intermediate factor
            inplace: change the orignal factor or create a new factor
    output: the factor after eliminating variables with the marginalization
            operation
    """

    if isinstance(variables, str):
        raise TypeError("variables: Expected type list or array-like, got type
            str")

    phi = phi_o if inplace else phi_o.copy()

    for var in variables:
        if var not in phi.variables:
            raise ValueError(f"{var} not in scope.")

    # get the indices of the input variables
    var_indexes = [phi.variables.index(var) for var in variables]

    # get the indices of the rest variabels
    index_to_keep = sorted(set(range(len(phi_o.variables))) - set(var_indexes))
    # the new factor with the rest variables
    phi.variables = [phi.variables[index] for index in index_to_keep]
    # the new factor with the cardinality of the rest variables
    phi.cardinality = phi.cardinality[index_to_keep]
    # delete the eliminated variables
    phi.del_state_names(variables)

#       phi.values = np.log(np.sum(np.exp(phi.values), axis = tuple(var_indexes))
    )
    values = np.split(phi.values, 2, axis = var_indexes[0])
    phi.values = np.logaddexp(values[0].reshape([2 for i in range(len(phi.
        variables))]), values[1].reshape([2 for i in range(len(phi.variables))])
        )

    if not inplace:
        return phi

def cal_pr(graph, variables, evidence, elimination_order, joint = True,
    show_progress = False):
    """
    inputs: graph: a pgmpy MarkovNetwork
            variables: variables that need to be eliminated
            evidence: the evidence of this query
            elimination_order: the elimination order
```

```
            joint: True: return the query over all input variables;
                False: return the queries of variables
                    individually
            show_progress: use tqdm or not
    output: log10(the partition function)
    """

    # Step 2: Prepare data structures to run the algorithm.
    eliminated_variables = set()
    # Get working factors and elimination order
    working_factors, elimination_order = preparation(graph, variables, evidence
        , elimination_order, joint = True)
    if not working_factors:
        return elimination_order
    # elimination_order = elimination_order

    # Step 3: Run variable elimination
    if show_progress:
        pbar = tqdm(elimination_order)
    else:
        pbar = elimination_order

    for var in pbar:
        if show_progress:
            pbar.set_description(f"Eliminating: {var}")
        # Removing all the factors containing the variables which are
        # eliminated (as all the factors should be considered only once)
        factors = [factor for factor, _ in working_factors[var] if not set(
            factor.scope()).intersection(eliminated_variables)]
#           if len(factors) == 0:
#               print(var)
#               print(factors)
        phi = factor_product_1(*factors)
        phi = marginalize(phi, [var], inplace = False)
        # phi = getattr(phi, operation)([var], inplace=False)
        del working_factors[var]
        for variable in phi.variables:
            working_factors[variable].add((phi, var))
        eliminated_variables.add(var)
#           eliminated_assignments[var] = (var_assignment, phi.variables)

    # Step 4: Prepare variables to be returned.
    final_distribution = set()
    for node in working_factors:
        for factor, origin in working_factors[node]:
            if not set(factor.variables).intersection(eliminated_variables):
                final_distribution.add((factor, origin))
    final_distribution = [factor for factor, _ in final_distribution]

    if joint:
        if isinstance(graph, BayesianNetwork):
            final_distribution = factor_product_1(*final_distribution).
                normalize(inplace=False)
        else:
            final_distribution = factor_product_1(*final_distribution)
    else:
        query_var_factor = {}
```

```
        for query_var in variables:
            phi = factor_product_1(*final_distribution)
            query_var_factor[query_var] = phi.marginalize(list(set(variables) -
                 set([query_var])), inplace=False).normalize(inplace=False)
        final_distribution = query_var_factor

    return np.logaddexp(final_distribution.values[0], final_distribution.values
        [1]) / np.log(10)

# Compute the most probable explanation (MPE) task
def maximize(phi_o, variables, inplace=True):
    """
    Maximizes the factor with respect to `variables`.

    inputs: phi_o: the intermediate factor
            variables: variables that need to be eliminated from the
                intermediate factor
            inplace: change the orignal factor or create a new factor
    output: the factor after eliminating variables with the maximization
            operation
    """

    if isinstance(variables, str):
        raise TypeError("variables: Expected type list or array-like, got type
            str")

    phi = phi_o if inplace else phi_o.copy()

    for var in variables:
        if var not in phi.variables:
            raise ValueError(f"{var} not in scope.")

    # get the indices of the input variables
    var_indexes = [phi.variables.index(var) for var in variables]

    # get the indices of the rest variabels
    index_to_keep = sorted(set(range(len(phi_o.variables))) - set(var_indexes))
    # the new factor with the rest variables
    phi.variables = [phi.variables[index] for index in index_to_keep]
    # the new factor with the cardinality of the rest variables
    phi.cardinality = phi.cardinality[index_to_keep]
    # delete the eliminated variables
    phi.del_state_names(variables)

    var_assig = np.argmax(phi.values, axis = var_indexes[0])
    phi.values = np.max(phi.values, axis = tuple(var_indexes))

    if not inplace:
        return phi, var_assig

def cal_mpe(graph, variables, evidence, elimination_order, joint = True,
    show_progress = True):
    """
    inputs: graph: a pgmpy MarkovNetwork
            variables: variables that need to be eliminated
            evidence: the evidence of this query
            elimination_order: the elimination order
```

```
        joint: True: return the query over all input variables;
            False: return the queries of variables
                individually
        show_progress: use tqdm or not
output: max_prob: the maximum probability in the joint
        distribution over all input variables
        assignments: a dictionary of the assignment of each variable in the
             case that the probability is the maximum
"""

# Step 2: Prepare data structures to run the algorithm.
eliminated_variables = set()
# Get working factors and elimination order
working_factors, elimination_order = preparation(graph, variables, evidence
    , elimination_order, joint = True)
if not working_factors:
    return elimination_order, False
# elimination_order = elimination_order

assignments = {node: None for node in graph.nodes}
eliminated_assignments = {node: (None, None) for node in elimination_order}

# Step 3: Run variable elimination
if show_progress:
    pbar = tqdm(elimination_order)
else:
    pbar = elimination_order

for var in pbar:
    if show_progress:
        pbar.set_description(f"Eliminating: {var}")
    # Removing all the factors containing the variables which are
    # eliminated (as all the factors should be considered only once)
    factors = [factor for factor, _ in working_factors[var] if not set(
        factor.scope()).intersection(eliminated_variables)]
    phi = factor_product(*factors)
    phi, var_assignment = maximize(phi, [var], inplace = False)
    # phi = getattr(phi, operation)([var], inplace=False)
    del working_factors[var]
    for variable in phi.variables:
        working_factors[variable].add((phi, var))
    eliminated_variables.add(var)
    eliminated_assignments[var] = (var_assignment, phi.variables)

# Step 4: Prepare variables to be returned.
final_distribution = set()
for node in working_factors:
    for factor, origin in working_factors[node]:
        if not set(factor.variables).intersection(eliminated_variables):
            final_distribution.add((factor, origin))
final_distribution = [factor for factor, _ in final_distribution]

if joint:
    if isinstance(graph, BayesianNetwork):
        final_distribution = factor_product(*final_distribution).normalize(
            inplace=False)
    else:
```

```
            final_distribution = factor_product(*final_distribution)
    else:
        query_var_factor = {}
        for query_var in variables:
            phi = factor_product(*final_distribution)
            query_var_factor[query_var] = phi.marginalize(list(set(variables) -
                set([query_var])), inplace=False).normalize(inplace=False)
        final_distribution = query_var_factor

    max_assign = np.unravel_index(np.argmax(final_distribution.values, axis =
        None), final_distribution.values.shape)
    for (node, assign) in zip(final_distribution.variables, max_assign):
        assignments[node] = assign
    elimination_order.reverse()
    for node in elimination_order:
        ind = []
        for variable in eliminated_assignments[node][1]:
            ind.append(assignments[variable])
        assignments[node] = eliminated_assignments[node][0][tuple(ind)]
    # max_assign = np.argmax(final_distribution.values)
    max_prob = np.max(final_distribution.values)
    return max_prob, assignments

# create a k-tree recursively
def create_aktree(k, n):
    # input:
    #     k: treewidth
    #     n: the number of nodes
    # output:
    #     k_cliques: all cliques in the k-tree
    #     k_tree: the adjacency lists of the k-tree
    k_tree = {}
    k_cliques = []
    if k > n:
        return k_cliques, k_tree
    else:
        root = list(np.linspace(1, k, k, dtype = int))
        nodes = 1
        for i in range(k):
            k_tree[nodes] = root[0:root.index(nodes)] + root[root.index(nodes)
                +1:]
            nodes += 1
        k_cliques.append(root)
        select_clique = 0
        while nodes <= n:
            k_tree[nodes] = copy.deepcopy(k_cliques[select_clique])
            for i in k_cliques[select_clique]:
                k_tree[i].append(nodes)
                cl = k_cliques[select_clique][0:k_cliques[select_clique].index(
                    i)] \
                    + k_cliques[select_clique][k_cliques[select_clique].index(i
                        )+1:] \
                    + [nodes]
                k_cliques.append(cl)
            if np.random.rand() < 0.2:
                select_clique = 0
            else:
```

```
                select_clique = np.random.randint(0, len(k_cliques)-1)
            nodes += 1
        return k_cliques, k_tree

# disorder the vertices
def disorder(nodes):
    """
    input: nodes: a list of nodes
    outputs: nodes_en: a encoding dictionary with the nodes as keys and the
                corresponding codes as values
             nodes_de: a decoding dictionary with the codes as keys and the
                original nodes as values
    """
    nodes_dis = np.random.permutation(np.arange(len(nodes)))
    nodes_en = {nodes[i]: nodes_dis[i]+1 for i in range(len(nodes))}
    nodes_de = {nodes_dis[i]+1: nodes[i] for i in range(len(nodes))}
    return nodes_en, nodes_de

# check the connectivity of the created Markov random fields
# Disjoint-set data structure
def find(dic, nod):
    """
    inputs: dic: a dictionary with nodes as keys and the smallest node in the
                set their belong to as values
            nod: the node that need to find its father, its father is the
                smallest node in the set it belongs to
    output: the father
    """
    if dic[nod] == nod:
        father = nod
    else:
        father = find(dic, dic[nod])
    return father

def check_subgraphs(nodes, edges):
    """
    inputs: nodes: the list of nodes in the graph
            edges: the list of edges in the graph
    output: the largest father among all subsets
    """
    dic = {}
    for nod in nodes:
        dic[nod] = nod
    for edg in edges:
        fa = [find(dic, edg[0]), find(dic, edg[1])]
        if fa[0] == fa[1]:
            continue
        else:
            idx = fa.index(max(fa))
            dic[fa[idx]] = min(fa)
            dic[edg[idx]] = min(fa)
    for nod in nodes:
        dic[nod] = find(dic, dic[nod])

    return max(dic.values())

def create_aMRF(kk, n, idx, remove_edges = False, prob_re = 0.0):
```

```
"""
inputs: kk: the tree-width
        n: the number of vertices in the Markov random field
        idx: the index of selected sampling method
        remove_edges: whether or not removing edges from the graph
        prob_re: the probability of the removed edges
outputs: G2: a pairwise Markov random field
         nodes_en: a encoding dictionary with the nodes as keys and the
             corresponding codes as values
"""

k_cliques, k_tree = create_aktree(kk, n)

nodes_o = [i+1 for i in range(n)]
edges_o = set()
nodes_en, nodes_de = disorder(nodes_o)
for (k, v) in k_tree.items():
    for l in v:
        edges_o.add((min(nodes_en[l], nodes_en[k]), max(nodes_en[l],
            nodes_en[k])))

connectivity = False

# print(nodes)
# print(edges)

if remove_edges and prob_re > 2e-5:
    while not connectivity:
        num_re = int(len(edges_o) * prob_re)
        len_ed_re = []
        for m in range(6):
            h = 2 - m * 0.2
            edge_re_ca = [edg for edg in edges_o if len(k_tree[edg[0]]) > h
                 * kk and len(k_tree[edg[1]]) > h * kk]
            len_ed_re.append(len(edge_re_ca))
            if len(edge_re_ca) > num_re:
                break
        if len_ed_re[-1] < num_re:
            for m in range(5):
                h = 1.4 - m * 0.1
                edge_re_ca = [edg for edg in edges_o if len(k_tree[edg[0]])
                     > h * kk and len(k_tree[edg[1]]) >= h * kk]
                if len(edge_re_ca) > num_re:
                    break

        prob_edg = [(math.pow(len(k_tree[edg[0]]), 2) + math.pow(len(k_tree
            [edg[1]]), 2)) for edg in edge_re_ca]
        prob_edg = [pro/sum(prob_edg) for pro in prob_edg]
        idx_re = np.random.choice(len(edge_re_ca), num_re, replace = False,
             p = prob_edg)
        edges = edges_o - set([edge_re_ca[i] for i in idx_re])

        dic = {}
        for nod in nodes_o:
            dic[nod] = []
        for edg in edges:
            dic[edg[0]].append(edg[1])
```

```
                dic[edg[1]].append(edg[0])

            nodes_re = []
            for (k, v) in dic.items():
                if len(v) == 0:
                    nodes_re.append(k)

            nodes = [nod for nod in nodes_o if nod not in nodes_re]

            if check_subgraphs(nodes, edges) == 1:
                connectivity = True

    else:
        nodes = nodes_o
        edges = edges_o

    G2 = MarkovNetwork()
    G2.add_nodes_from(nodes)
    G2.add_edges_from(edges)
    # factor_edges = [DiscreteFactor(edge, [2 for i in range(kk)], random(idx,
        int(math.pow(2, kk)))) for edge in k_cliques]
    factor_edges = [DiscreteFactor(edge, [2, 2], random(idx, 4)) for edge in
        edges]
    G2.add_factors(*factor_edges)

    return G2, nodes_en

def create_MRFs(kk, n, idx, remove_edges = False, prob_re = 0.0):
    """
    inputs: kk: the tree-width
            n: the number of vertices in the Markov random field
            idx: the index of selected sampling method
            remove_edges: whether or not removing edges from the graph
            prob_re: the probability of the removed edges
    outputs: G1: a Markov random field with factors on cliques
             G2: a pairwise Markov random field has the same structure as G1
             nodes_en: a encoding dictionary with the nodes as keys and the
                 corresponding codes as values
    """

    k_cliques, k_tree = create_aktree(kk, n)

    nodes_o = [i+1 for i in range(n)]
    edges_o = set()
    # cliques_o = k_cliques
    cliques_o = []
    for kcl in k_cliques:
        kcl_en = [nodes_en[i] for i in kcl]
        cliques_o.append(kcl_en)
    nodes_en, nodes_de = disorder(nodes_o)
    for (k, v) in k_tree.items():
        for l in v:
            edges_o.add((min(nodes_en[l], nodes_en[k]), max(nodes_en[l],
                nodes_en[k])))

    connectivity = False
```

```
    # print(nodes)
    # print(edges)

    if remove_edges and prob_re > 2e-5:
        while not connectivity:
            num_re = int(len(edges_o) * prob_re)
            len_ed_re = []
            for m in range(6):
                h = 2 - m * 0.2
                edge_re_ca = [edg for edg in edges_o if len(k_tree[edg[0]]) > h
                     * kk and len(k_tree[edg[1]]) > h * kk]
                len_ed_re.append(len(edge_re_ca))
                if len(edge_re_ca) > num_re:
                    break
            if len_ed_re[-1] < num_re:
                for m in range(5):
                    h = 1.4 - m * 0.1
                    edge_re_ca = [edg for edg in edges_o if len(k_tree[edg[0]])
                         > h * kk and len(k_tree[edg[1]]) >= h * kk]
                    if len(edge_re_ca) > num_re:
                        break

            prob_edg = [(math.pow(len(k_tree[edg[0]]), 2) + math.pow(len(k_tree
                [edg[1]]), 2)) for edg in edge_re_ca]
            prob_edg = [pro/sum(prob_edg) for pro in prob_edg]
            idx_re = np.random.choice(len(edge_re_ca), num_re, replace = False,
                 p = prob_edg)
            edges = edges_o - set([edge_re_ca[i] for i in idx_re])

            dic = {}
            for nod in nodes_o:
                dic[nod] = []
            for edg in edges:
                dic[edg[0]].append(edg[1])
                dic[edg[1]].append(edg[0])

            nodes_re = []
            for (k, v) in dic.items():
                if len(v) == 0:
                    nodes_re.append(k)

            nodes = [nod for nod in nodes_o if nod not in nodes_re]

            if check_subgraphs(nodes, edges) == 1:
                connectivity = True

        cliques = set()
        cli_edg = set()
        cli_edges = set()
        for clique in k_cliques:
            clique.sort()
            remove_edges = set(combinations(clique, 2)) - edges
            if len(remove_edges) == 0:
                cli_edges.update(set(combinations(clique, 2)))
                cliques.add(tuple(clique))
                continue
            else:
```

```
                temp_dic = {}
                for edge in remove_edges:
                    for i in range(2):
                        if edge[i] in temp_dic.keys():
                            temp_dic[edge[i]].append(edge[1 - i])
                        else:
                            temp_dic[edge[i]] = [edge[1 - i]]
                for key in temp_dic.keys():
                    clique_n = sorted(set(clique) - set(temp_dic[key])) # list
                    if len(clique_n) == 1:
                        if clique_n[0] in nodes:
                            cliques.add(tuple(clique_n))
                        else:
                            continue
                    elif len(clique_n) == 2:
                        cli_edg.add(tuple(clique_n))
                    elif len(set(combinations(clique_n, 2)) - edges) == 0:
                        cli_edges.update(set(combinations(clique_n, 2)))
                        cliques.add(tuple(clique_n))
                    else:
                        cli_edg.update(set(combinations(clique_n, 2)).
                            intersection(edges))

        for clique in cli_edg:
            if clique not in cli_edges:
                cliques.add(clique)
    else:
        nodes = nodes_o
        edges = edges_o
        cliques = cliques_o

    G1 = MarkovNetwork()
    G1.add_nodes_from(nodes)
    G1.add_edges_from(edges)
    # factor_edges = [DiscreteFactor(edge, [2 for i in range(kk)], random(idx,
        int(math.pow(2, kk)))) for edge in k_cliques]
    factor_edges = [DiscreteFactor(clique, [2 for i in range(len(clique))],
        random(idx, int(math.pow(2, len(clique))))) for clique in cliques]
    G1.add_factors(*factor_edges)

    G2 = MarkovNetwork()
    G2.add_nodes_from(nodes)
    G2.add_edges_from(edges)
    # factor_edges = [DiscreteFactor(edge, [2 for i in range(kk)], random(idx,
        int(math.pow(2, kk)))) for edge in k_cliques]
    factor_edges = [DiscreteFactor(edge, [2, 2], random(idx, 4)) for edge in
        edges]
    G2.add_factors(*factor_edges)

    return G1, G2, nodes_en

def benchmark_uai(path, file_name, kk, n, idx, remove_edges = False, prob_re =
    0.0):
    """
    inputs: path: the path of the folder to save the benchmark
            file_name: the name of the benchmark file
            kk: the tree-width
```

```
            n: the number of vertices in the Markov random field
            idx: the index of selected sampling method
            remove_edges: whether or not removing edges from the graph
            prob_re: the probability of the removed edges
    output: no output, but the benchmark will be created in the folder
            according to the input path
    """

    G, nodes_en = create_aMRF(kk, n, idx, remove_edges, prob_re)
    eli_ord = [nodes_en[i] for i in range(n, 0, -1)]
    pr = cal_pr(G, [eli_ord[-1]], evidence = None, elimination_order = eli_ord
        [:-1], joint = True, show_progress = False)
    max_prob, mpe = cal_mpe(G, variables = [eli_ord[-1]], evidence = None,
        elimination_order = eli_ord[:-1], joint = True, show_progress = False)
    completeName = os.path.join(path, file_name)
    writer = UAIWriter(G)
    writer.write_uai(completeName)

    with open(completeName+'.PR', 'w') as f:
        f.write("PR\n")
        f.write(str(pr))
    with open(completeName+'.MPE', 'w') as f:
        f.write("MPE\n")
        ls_mpe = list(mpe.values())
        ls_mpe.insert(0, len(mpe))
        str_mpe = ' '.join(str(v) for v in ls_mpe)
        f.write(str_mpe)
```

A.2 Create a Markov random field as a benchmark for the tree-width competition

```
def benchmark_tw(path, file_name, kk, n, remove_edges = False, prob_re = 0.0):
    """
    inputs: path: the path of the folder to save the benchmark
            file_name: the name of the benchmark file
            kk: the tree-width
            n: the number of vertices in the Markov random field
            prob_re: the probability of the removed edges
    output: no output, but the benchmark will be created in the folder
            according to the input path
    """

    k_cliques, k_tree = create_aktree(kk, n)

    nodes_o = [i+1 for i in range(n)]
    edges_o = set()
    nodes_en, nodes_de = disorder(nodes_o)
    for (k, v) in k_tree.items():
        for l in v:
            edges_o.add((min(nodes_en[l], nodes_en[k]), max(nodes_en[l],
                nodes_en[k])))

    connectivity = False
```

```
    # print(nodes)
    # print(edges)

    if remove_edges and prob_re > 2e-5:
        while not connectivity:
            num_re = int(len(edges_o) * prob_re)
            len_ed_re = []
            for m in range(6):
                h = 2 - m * 0.2
                edge_re_ca = [edg for edg in edges_o if len(k_tree[edg[0]]) > h
                    * kk and len(k_tree[edg[1]]) > h * kk]
                len_ed_re.append(len(edge_re_ca))
                if len(edge_re_ca) > num_re:
                    break
            if len_ed_re[-1] < num_re:
                for m in range(5):
                    h = 1.4 - m * 0.1
                    edge_re_ca = [edg for edg in edges_o if len(k_tree[edg[0]])
                        > h * kk and len(k_tree[edg[1]]) >= h * kk]
                    if len(edge_re_ca) > num_re:
                        break

            prob_edg = [(math.pow(len(k_tree[edg[0]]), 2) + math.pow(len(k_tree
                [edg[1]]), 2)) for edg in edge_re_ca]
            prob_edg = [pro/sum(prob_edg) for pro in prob_edg]
            idx_re = np.random.choice(len(edge_re_ca), num_re, replace = False,
                 p = prob_edg)
            edges = edges_o - set([edge_re_ca[i] for i in idx_re])

            dic = {}
            for nod in nodes_o:
                dic[nod] = []
            for edg in edges:
                dic[edg[0]].append(edg[1])
                dic[edg[1]].append(edg[0])

            nodes_re = []
            for (k, v) in dic.items():
                if len(v) == 0:
                    nodes_re.append(k)

            nodes = [nod for nod in nodes_o if nod not in nodes_re]

            if check_subgraphs(nodes, edges) == 1:
                connectivity = True

    else:
        nodes = nodes_o
        edges = edges_o

    completeName = os.path.join(path, file_name)
    grfile = open(completeName, "w")

    line = "p tw " + str(n) + " " + str(len(edges)) + "\n"
    grfile.write(line)

    for edge in edges:
```

```
        grfile.write(str(edge[0]) + " " + str(edge[1]) + "\n")

    grfile.close()

    txt_path = completeName[:-3] + ".txt"
    with open(txt_path, 'r') as f:
        f.write(file_name + " " + str(kk))
```

A.3 Find the elimination orders from Markov random fields by the straightforward intuition

```
from itertools import combinations

def remove_node(MRF_dic, edges, node):
    """
    inputs: MRF_dic: a dictionary representing the current Markov
            random field that vertices are keys and their edge information is
                values. The edge information includes their neighbors and their
                connected edges.
            edges: the set of edges in the current graph
            node: the vetex that need to be removed from the current Markov
                random field
    output: the updated MRF_dic and the updated edges
    """
    neighbors = MRF_dic[node]['neighbors']
    add_edges = set(combinations(neighbors, 2)) - set(edges)
    for edge in MRF_dic[node]['neigh_edg']:
        if edge in edges:
            edges.remove(edge)
        else:
            continue
    edges.extend(list(add_edges))
    for neighbor in neighbors:
        MRF_dic[neighbor]['neighbors'].remove(node)
        add_neighbors = list(set(neighbors) - set([neighbor]) - set(MRF_dic[
            neighbor]['neighbors']))
#           MRF_dic[neighbor]['neighbors'] = list(set(MRF_dic[neighbor]['
    neighbors']).union(set(neighbors) - set(neighbor)))
#           MRF_dic[neighbor]['neighbors'].extend([x for x in neighbors if x not
    in MRF_dic[neighbor]['neighbors'] and x != neighbor])
        MRF_dic[neighbor]['neighbors'].extend(add_neighbors)
        MRF_dic[neighbor]['neigh_edg'].remove((min(node, neighbor), max(node,
            neighbor)))
        for an in add_neighbors:
            MRF_dic[neighbor]['neigh_edg'].append((min(an, neighbor), max(an,
                neighbor)))
    del MRF_dic[node]

def get_MinNeighbors(MRF_dic, node):
    """
    inputs: MRF_dic: a dictionary representing the current Markov
            random field that vertices are keys and their edge information is
                values. The edge information includes their neighbors and their
                connected edges.
```

```
            node: a vertex in the current graph
    output: the cost of the vertex based on MinNeighbors method          in the
        current graph
    """
    return len(MRF_dic[node]['neighbors'])

def get_MinWeight(MRF_dic, node):
    """
    inputs: MRF_dic: a dictionary representing the current Markov
            random field that vertices are keys and their edge information is
                values. The edge information includes their neighbors and their
                connected edges.
            node: a vertex in the current graph
    output: the cost of the vertex based on MinWeight method          in the
        current graph - log trick used to avoid data overflow
    """
    return np.sum([np.log(MRF_dic[neighbor]['card']) for neighbor in MRF_dic[
        node]['neighbors']])

def get_MinFill(MRF_dic, node):
    """
    inputs: MRF_dic: a dictionary representing the current Markov
            random field that vertices are keys and their edge information is
                values. The edge information includes their neighbors and their
                connected edges.
            node: a vertex in the current graph
    output: the cost of the vertex based on MinFill method                  in
        the current graph
    """
    neighbors = MRF_dic[node]['neighbors']
    exist_edges = set()
    for neighbor in neighbors:
        exist_edges.update(set(MRF_dic[neighbor]['neigh_edg']))
    add_edges = set(combinations(neighbors, 2)) - set(exist_edges)
    return len(add_edges)

def get_WeightedMinFill(MRF_dic, node):
    """
    inputs: MRF_dic: a dictionary representing the current Markov
            random field that vertices are keys and their edge information is
                values. The edge information includes their neighbors and their
                connected edges.
            node: a vertex in the current graph
    output: the cost of the vertex based on WeightedMinFill method          in
        the current graph
    """
    neighbors = MRF_dic[node]['neighbors']
    exist_edges = set()
    for neighbor in neighbors:
        exist_edges.update(set(MRF_dic[neighbor]['neigh_edg']))
    add_edges = set(combinations(neighbors, 2)) - set(exist_edges)
    return np.sum([MRF_dic[edge[0]]['card'] * MRF_dic[edge[1]]['card'] for edge
         in add_edges])

def get_elimination_order(MRF, cost):
    """
    inputs: MRF: a pgmpy MarkovNetwork
```

```
            cost: methods to get the elimination order, the methods are
                get_MinNeighbors, get_MinWeight, get_MinFill,
                get_WeightedMinFill
    output: the elimination order based on the input method
    """
    nodes = list(MRF.nodes)
    edges_o = list(MRF.edges)
    edges = list()

    ordering = []
    dic = {}
    for nod in nodes:
        dic[nod] = {'card': MRF.get_cardinality(nod),'neighbors': [], '
            neigh_edg': []}
    for edg in edges_o:
        edge = (min(edg[0], edg[1]), max(edg[0], edg[1]))
        edges.append(edge)
        dic[edg[0]]['neigh_edg'].append(edge)
        dic[edg[1]]['neigh_edg'].append(edge)
        dic[edg[0]]['neighbors'].append(edg[1])
        dic[edg[1]]['neighbors'].append(edg[0])

    while len(nodes) > 1:
        scores = {node: cost(dic, node) for node in nodes}
        min_score_node = min(scores, key = scores.get)
        ordering.append(min_score_node)
#          print(min_score_node)
        nodes.remove(min_score_node)
        remove_node(dic, edges, min_score_node)

    ordering.extend(nodes)
    return ordering
```

A.4 Translate a Markov random field to a Bayesian Network

```
# convert MRF to BN
from pgmpy.factors.discrete import TabularCPD
from pgmpy.models import BayesianNetwork

def translate(G):
#       input G: a pgmpy MarkovNetwork
#       output BN: a pgmpy BayesianNetwork
    nodes = list(G.nodes)
    edges = list(G.edges)
    BN = BayesianNetwork()
    s_nodes = [str(node) for node in nodes]
    z_nodes = [str(len(nodes) + 1 + i) for i in range(len(edges))]
    BN.add_nodes_from(s_nodes + z_nodes)
    for i in s_nodes:
        card = G.get_cardinality(node = int(i))
        cpd = TabularCPD(i, card, [[1/card] for j in range(card)])
        BN.add_cpds(cpd)

    for (edge, z) in zip(edges, z_nodes):
        BN.add_edges_from([(str(l), z) for l in edge])
```

```
        factor = [fa for fa in G.get_factors(edge[0]) if edge[1] in fa.scope()
            ][0]
        phi = factor.copy()
        phi.normalize()
        phi_reshape = phi.values.reshape((1, -1))
        cpd = TabularCPD(z, 2, np.vstack((phi_reshape, 1 - phi_reshape)),
            evidence = [str(edge[0]), str(edge[1])], \
                          evidence_card = list(factor.get_cardinality(edge).
                              values()))
        BN.add_cpds(cpd)

    return BN
```

A.5 Read a Markov random field from UAI file

```
from pgmpy.factors.discrete import DiscreteFactor
from pgmpy.models import MarkovNetwork
from itertools import combinations

def get_graph(path):
#     input path: path to the graph file
#.    output G: a pgmpy MarkovNetwork
#     reader = UAIReader(path = path)
    with open(path, 'r') as f:

        # Preamble
        lines = [x.strip() for x in f.readlines() if x.strip()]
#         assert lines[0] == 'MARKOV'
        if lines[0] == 'MARKOV':
            G = MarkovNetwork()
        else:
            raise ValueError("This is not a Markov random field.")

        n_vars = int(lines[1])
        nodes = list(range(1, n_vars + 1))
        cardinalities = [int(x) for x in lines[2].split()]
        n_cliques = int(lines[3])
        edges = set()
        factor = []
        G.add_nodes_from(nodes)
        for i in range(n_cliques):
            edge = [y + 1 for y in [int(x) for x in lines[i + 4].split()][1:]]
            edges_in_cliq = set(combinations(edge, 2))
            edges.update(edges_in_cliq)
            factor.append(DiscreteFactor(edge, [cardinalities[node - 1] for
                node in edge], [float(x) for x in lines[i * 2 + 1 + n_cliques +
                4].split()]))

        G.add_edges_from(edges)
        G.add_factors(*factor)

    return G
```