TU WIEN Informatics

Scalable Bayesian Network Structure Learning using SAT-based Methods

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften

eingereicht von

Vaidyanathan Peruvemba Ramaswamy, M.Tech., B.Tech.
Matrikelnummer 11928280

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.rer.nat. Stefan Szeider

Diese Dissertation haben begutachtet:

Pekka Parviainen

Sebastian Ordyniak

Wien, 12. Mai 2023

Vaidyanathan Peruvemba
Ramaswamy

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Scalable Bayesian Network Structure Learning using SAT-based Methods

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften

by

Vaidyanathan Peruvemba Ramaswamy, M.Tech., B.Tech.
Registration Number 11928280

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.rer.nat. Stefan Szeider

The dissertation has been reviewed by:

Pekka Parviainen

Sebastian Ordyniak

Vienna, 12th May, 2023

Vaidyanathan Peruvemba Ramaswamy

Erklärung zur Verfassung der Arbeit

Vaidyanathan Peruvemba Ramaswamy, M.Tech., B.Tech.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Mai 2023

Vaidyanathan Peruvemba
Ramaswamy

Acknowledgements

When I started this PhD back in September 2019, there was a small part of me that was fearing that I had bitten off more than I could chew, that I was too naïve to comprehend the behemoth of a job I was signing up for. A PhD is a big deal where I come from. After completing my Bachelors and Masters in India, and faced with the decision of what to do next; a PhD just happened to be the least undesirable option. I applied to few places, with the primary criterion of maximizing the likelihood of me completing the PhD. Consequently, what drew me towards this PhD position at TU Wien was the familiarity of the research area along with the promise of a healthy mix of theoretical and practical work (and I was not disappointed). I was so desperate to secure this position that I sacrificed a good night's sleep to complete an assignment (which was a part of the hiring process) in one day, despite having been granted 2-3 weeks. Fortunately, my efforts paid off, and I got the position.

I've always maintained that I'm not a big fan of *change*, but this move had change written all over it: living in a foreign country for the first time, managing my own living and food situation, financial independence, new culture, new language, und so weiter. I had done what I could do from my side to maximize my chances of success by picking a position that I felt I would be self-motivated in. But, there were several other cogs in the machinery that finally yielded me my doctorate amid all the change and chaos 🦠. These cogs were the numerous people that helped in their own ways, and these are the people that I would like to thank in the rest of this section.

First and foremost, I would like to thank my supervisor Stefan Szeider for being extremely understanding and making the core of this PhD experience an absolute treat. I was particularly moved by how well you handled my dip in productivity at the start of the pandemic—no anger or frustration, just looking to come up with a way to get me back on track. I also appreciate the space you gave me with my vision for this thesis despite it not being the most efficient approach. I would also like to thank my reviewers Pekka Parviainen and Sebastian Ordyniak for their valuable feedback and cooperation.

Secondly, I would like to thank my parents for nurturing my curiosity and problem-solving attitude. Despite the humble background, I rarely had the feeling that I was lacking anything. Further, I would like to thank, my grandparents who also contributed to the childhood that I so fondly reminisce.

I am very fortunate to have found a second family here in Vienna, i.e., the Schlossis (Toby, Sezen, Mali, Greg) and the former Schlossis (Milena, Eli, Neha, Stefan, Klausi). Thank you for making the lockdown bearable, for improving my German, and for the movie and game nights. I am also thankful to Theresienbad for some of my most blissful summer memories.

I would also like to thank the Algorithms and Complexity group for providing a pleasant and relaxing backdrop for me to work in. The group lunches were quite often the highlight of my day, and my weekends were often brightened up by racket sports with colleagues. On a related note, I would like to thank the fallen vegan angel [Venga] for their tireless and efficient service before being wiped out by the pandemic. I would also like to thank Fabian Klute for his dissertation which served as a trusty exemplar.

I would like to thank Mikko Koivisto for being a wonderful host during my research visit at the University of Helsinki. Further, I would also like to thank Alex, my Table Tennis coach in Helsinki, for turbocharging my love for the sport. And thanks to Juha, Trân and Hector for sprucing up my Helsinki experience.

In retrospect, one of the most influential steps that brought me to this juncture was the decision to join IIT Gandhinagar. I am incredibly thankful to the institute and faculty for laying a strong foundation, especially Neeldhara Misra for introducing me to Parameterized Complexity, and Bireswar Das for the enriching teaching opportunities. Additionally, I would like thank my close friends in India, for ensuring that I do not forget my roots with regular gaming nights and untimely video calls. I am also grateful to those friends who proofread the initial chapters of this manuscript.

Moving to a new country and then to another for a research visit means lots of bureaucracy and paperwork. MA35 was the kingpin here, making sure I don't get too engrossed in my research with periodic interruptions. However, thanks to Beatrix Buhl and Anna Prianichnikova (a part of the *Doktorandenkolleg*), I was able to mitigate the impact of MA35. I would also like to thank the other DK students for providing another much-needed social outlet and for the wonderful retreat to Linz in 2019.

Finally, a mixed bag of all the remaining things I am grateful for: Gorilla Kitchen (for helping me pull late-nighters), Eiskaffee, Tischtennis, Conrad das Fahrrad (which was tragically stolen) und das türkise Fahrrad. *Danke schön!*

Kurzfassung

Bayes'sche Netze (BNs) sind grafische Modelle, die weithin erforscht wurden und in der Praxis, insbesondere in der Medizin, häufig Anwendung finden. Diese Popularität hat zu mehreren heuristischen Methoden für das Lernen von BNs geführt, dem „Bayesian Network Structure Learning" (BNSL). Die „SAT-based Local Improvement Method" (SLIM) ist eine Metaheuristik zur lokalen Verbesserung einer globalen Lösung mit Hilfe eines SAT-Lösers. In dieser Arbeit befassen wir uns mit der Anwendung der SLIM-Metaheuristik auf bestehende BNSL-Algorithmen, um qualitativ bessere BNs zu erhalten und gleichzeitig die Skalierbarkeit beizubehalten. Wir betrachten das grundlegende BNSL-Problem mit beschränkter Baumweite sowie zwei weitere Varianten: BNSL mit beschränkter Zustandsraumgröße und BNSL mit Expertennebenbedingungen. Für jede Variante entwerfen wir einen SLIM-Algorithmus und führen eine empirische Analyse durch, in der wir ihn mit modernen BNSL-Algorithmen vergleichen.

In einer Welt, in der Vorhersagemodelle wie neuronale Netze und Autoencoder immer beliebter werden, könnte die Rolle der Bayes'schen Netze trotz ihres bedeutenden Beitrags leicht übersehen werden. Im Bereich der *vertrauenswürdige Vorhersagemodelle* sind die BNs jedoch die vorherrschende Methode. Darüber hinaus spielen sie weiterhin eine entscheidende Rolle bei der Anwendung von Vorhersagemodellen. Ein wichtiges Beispiel für solche Anwendungen, bei denen Vertrauen von größter Bedeutung ist, ist der Bereich der Medizin. Beispiele hierfür sind Diagnose von Krankheiten, die Bestimmung deren Ursachen oder Symptome, die Abschätzung der Wirksamkeit potenzieller Behandlungen und die Angabe von Wahrscheinlichkeiten im Zusammenhang mit genetisch bedingten Krankheiten. Diese Vertrauenswürdigkeit wird manchmal auch als *Erklärbarkeit* bezeichnet und findet als Forschungsgebiet *Explainable AI* (XAI) immer mehr Interesse. Für die am weitesten verbreiteten Modelle (wie neuronale Netze) ist die Erklärbarkeit nicht zufriedenstellend möglich und muß behelfsmäßig im Nachhinein approximiert werden. Bei BNs ist die Erklärbarkeit jedoch in das Modell selbst eingebaut, was es in diesem Zusammenhang viel attraktiver macht.

Ein BN besteht aus mehreren *Zufallsvariablen* „Random Variables" (RVs), die durch gerichtete Kanten miteinander verbunden sind. Jede RV stellt ein atomares Ereignis der realen Welt dar und kann mehrere Zustände annehmen; die Kanten erfassen die Interaktionen zwischen diesen Ereignissen, d.h., die probabilistische Korrelation zwischen den Zuständen der RVs. Der *gerichtete azyklische Graph* „Directed Acyclic Graph" (DAG)

der RVs bildet zusammen mit den Kanten einen Teil des BN, der als *Struktur* des BN bezeichnet wird. Darüber hinaus benötigen wir auch Tabellen der *bedingten Wahrscheinlichkeitsverteilung* „conditional probability distribution“ (CPD), eine für jede (die RVs), die die Wahrscheinlichkeiten bezüglich des Zustands dieses RVs festhält. Diese Tabellen werden zusammenfassend als *Parameter* des BN bezeichnet. Die Lernaufgabe kann direkt in zwei Teilaufgaben unterteilt werden, das Lernen der Struktur des BN und das Lernen der Parameter für eine gegebene Struktur, wobei die CPD-Tabellen ausgefüllt werden. Das algorithmisch BNSL-Problem ist bekanntermaßen schwierig. Da reale Szenarien in der Regel Tausende von RVs beinhalten, hat dies viele Forschungsarbeiten zur Entwicklung heuristischer Algorithmen für die Suche nach approximative Lösungen für das BNSL Problem angezogen. Es gibt einige Arbeiten zur exakten BNSL, aber diese Methoden sind schlecht skalierbar und für BNs mit mehr als 50 Variablen in der Regel nicht anwendbar.

Das Hauptziel unserer Arbeit bestand darin, bessere BNSL-Algorithmen zu entwickeln. Konkret geht es um drei wichtige Eigenschaften, die einzeln leicht zu erreichen sind, deren gleichzeitige Verwirklichung jedoch eine Herausforderung darstellt. In erster Linie müssen die von uns entwickelten Methoden zum Strukturlernen in der Lage sein, auf Tausende von Variablen zu skalieren. Zweitens müssen wir sicherstellen, dass die Schlussfolgerungen anhand des gelernten BN nicht nur handhabbar, sondern auch extrem schnell sind. Daraus ergeben sich gewünschte Eigenschaften der Struktur des gelernten BN, wie die Beschränkung der Baumweite. Schließlich möchten wir neben dem praktischen Fokus der beiden vorangegangenen Punkte sicherstellen, dass die Qualität des BN, unter Berücksichtigung dieser Eigenschaften, nicht leidet, d.h., dass die Eingabedaten gut repräsentiert werden. Dies bildet unser Dreigestirn–*Skalierbarkeit*, *Handhabbarkeit* und *Qualität*–die drei Eigenschaften, die wir gleichzeitig anstreben. Zunächst betrachten wir das klassische beschränkte Baumweite BNSL-Problem, bei dem wir uns auf die *Skalierbarkeit* konzentrieren und den Stand der Technik verbessern. Als nächstes befassen wir uns mit dem beschränkte Zustandsraumgröße BNSL-Problem, bei dem wir die Grenzen der *Handhabbarkeit* erweitern, indem wir eine neue Metrik vorschlagen, die die Inferenzzeit besser widerspiegelt. Schließlich leisten wir einen Beitrag zur Forschung im Bereich des Lernens von BNs, die zusätzliche kausale Beschränkungen berücksichtigen, die von einem Domänenexperten bereitgestellt werden. Dies führt zu *qualitativ* hochwertigeren BNs, die die zugrundeliegenden Daten besser repräsentieren, da sie nicht mehr rein korrelational sind. Wir schlagen die erste skalierbare Methode zum Erlernen solcher BNs vor, die dennoch handhabbare Schlussfolgerungen gewährleistet.

Unser Ansatz zur Erfüllung all dieser Anforderungen ist die *SAT-basierte lokale Verbesserungsmethode* (SLIM). SLIM ist eine Metaheuristik, die von Lodha, Ordyniak und Szeider (SAT 2016) eingeführt wurde. SLIM-basierte Algorithmen kombinieren heuristische und exakte Methoden, um ihre jeweiligen Vorteile zu nutzen und gleichzeitig ihre Nachteile in Grenzen zu halten. Auf einer höheren Ebene sind die Lokale-Suche-Algorithmen, die mit einer heuristischen Ausgangslösung beginnen und dann wiederholt lokale Teile dieser Ausgangslösung isolieren, verbessern und ersetzen. Das Herzstück dieses Verfahrens ist eine SAT-basierte exakte Methode (lokaler Löser), die für die Verbesserung der lokalen Teile zuständig ist. ‘SAT-basiert’ bezieht sich auf die Verwendung von Lösern für das *aussagenlogische Erfüllbarkeitsproblem* „Propositional Satisfiability problem“ (SAT) oder dessen Verallgemeinerung (wie MaxSAT). Diese Löser sind moderne technologische Wunderwerke, die in der Lage sind, das SAT-Problem für Formeln mit Millionen von Variablen zu lösen. Die allgemeine Philosophie bei der Kombination von heuristischen und exakten Methoden besteht darin, Probleme, die für jede Methode geeignet sind, spezifisch anzupassen. Dies ist der Grund dafür, dass der SAT-basierte lokale Löser nur mit mundgerechten Teilproblemen gefüttert wird und nicht mit der gesamten Eingabe in einem Durchgang. Dadurch wird sichergestellt, dass wir die Vorteile der exakten Methode nutzen können, ohne dass wir unzumutbar lange Laufzeiten in Kauf nehmen müssen.

Oberflächlich betrachtet mögen die von uns entwickelten SLIM-basierten Algorithmen ähnlich aussehen, aber jeder Algorithmus erfordert eine sorgfältige Einbettung der exakten Methode, um sicherzustellen, dass die globale Lösung auch dann gültig bleibt, wenn ein lokaler Teil durch eine verbesserte lokale Lösung ersetzt wurde. Dies ist die größte Herausforderung bei der Entwicklung eines SLIM-basierten Algorithmus. In unserer Arbeit über das beschränkte Zustandsraumgröße BNSL-Problem haben wir zum Beispiel einen neuartigen BDD-basierten Zähler verwendet, um mit Logarithmensummen umzugehen. Schließlich haben wir aufgrund des übergreifenden Ziels der Praktikabilität jeden unserer SLIM-basierten Algorithmen implementiert und experimentelle Auswertungen durchgeführt. Diese Auswertungen bestätigten, dass die theoretischen Versprechen auch unter den praktischen Herausforderungen der realen Welt Bestand haben. In jedem Fall haben wir gezeigt, dass die von uns entwickelten Algorithmen im Vergleich zu den modernsten Algorithmen gut abschneiden.

Abstract

In a world where the popularity of predictive models like Neural Networks and Autoencoders is rapidly growing, the role of the Bayesian Networks (BNs) might be easily overlooked, despite their significant contribution. However, when it comes to *trustworthy predictive models*, BNs have dominated the landscape and continue to play a critical role in the real-world use of predictive models. A major example of such real-world use where trust is of paramount importance is the field of *Medicine*. Examples include predictions about human illnesses and their causes or symptoms, estimating effectiveness of potential remedial measures, expressing the probabilities related to genetically transmitted conditions, etc. This trustworthiness is also sometimes referred to as *explainability* and is starting to garner lots of interest as a field of research called 'Explainable AI' (XAI). For most widely used models (like neural networks), explainability is an afterthought meaning that such models need to be retrofitted with additional 'explaining' mechanisms. However, in BNs, explainability is baked into the model itself, which makes them far more appealing.

A BN consists of several Random Variables (RVs) connected to one another by directed arcs. Each RV represents an atomic real-world event and can have multiple states; the arcs capture the interaction between these events, i.e., the probabilistic correlation between the states of RVs. The Directed Acyclic Graph (DAG) of the RVs along with the arcs forms one part of the BN, called the BN's *structure*. In addition to that, we also need conditional probability distribution (CPD) tables, one for each RV, which pins down the probabilities concerning the state of that RV. These tables are collectively called the *parameters* of the BN. Learning a BN can be naturally split into two subtasks, learning the structure of the BN and then learning the parameters for a given structure, by filling in the CPD tables. BN structure learning (BNSL) is known to be notoriously hard. Since real-world scenarios usually involve thousands of RVs, it has attracted lots of research in developing heuristic algorithms for finding approximate solutions to the BNSL problem. There is some work on exact BNSL, but these methods scale poorly and are usually impractical for networks with more than 50 variables.

The primary goal of our work was to develop better BNSL algorithms. More specifically, our work addresses three important properties which are easy to achieve individually but challenging to achieve simultaneously. First and foremost, the structure learning methods we develop must be able to scale up to thousands of variables, as opposed to freezing up

and failing to find any solutions for large networks. Secondly, once a BN is learned, it is used to make several predictions or inferences, thus we must ensure that these inferences are not only tractable but also extremely quick. This, in turn, translates to restrictions on the structure of the learned BNs, like bounding the treewidth of the BN. Lastly, along with the practical focus of the previous two points, we would like to ensure that the quality of the BNs does not suffer in the process, i.e., the input data is well-represented by the BN. This forms our trifecta—*scalability*, *tractability*, and *quality*—the three properties that we strive to achieve simultaneously in all our works. We first look at the classic Bounded Treewidth BNSL problem where we focus on *scalability* and improve the state of the art. Next, we work on the Bounded State Space BNSL problem where we expand the frontiers of *tractability* by proposing a new metric which better reflects the inference time. Finally, we contribute to the line of research in learning BNs which respect additional causal constraints supplied by a domain expert. This results in higher *quality* BNs which better represent the underlying data, as they are no longer purely correlational. We propose the first scalable method for learning such BNs while still ensuring tractable inferences.

Our approach to satisfy all these requirements is the SAT-based Local Improvement Method (SLIM). SLIM is a framework introduced by Lodha, Ordyniak, and Szeider (SAT 2016) and can be used to develop heuristic algorithms. SLIM-based algorithms combine heuristic and exact methods to capitalize on their respective advantages while keeping their drawbacks in check. At a higher-level, these algorithms are local search algorithms that start off with an initial heuristic solution and then repeatedly isolate, improve and patch back local parts of this initial solution. At the heart of this procedure is a SAT-based exact method (local solver) which is in charge of improving the local parts. 'SAT-based' refers to the utilization of solvers for the propositional satisfiability problem (SAT) or its generalization (such as MaxSAT). These solvers are modern-day technological marvels that are capable of solving the SAT problem for formulas with millions of Boolean variables. The general philosophy in combining both heuristic and exact methods is to specifically tailor problems which are suitable for each method. This is the reason that the SAT-based local solver is only fed bite-sized subproblems and not the entire input in one go. This ensures that we can still reap the benefits of the exact method without incurring infeasibly long running times.

At a higher level, the SLIM-based algorithms we develop, might look similar, but each algorithm requires careful fortification of the exact method to be able to ensure that the global solution remains valid even after a local part was replaced by an improved local solution. This is the most challenging part of developing a SLIM-based algorithm. For instance, in our work on bounded state space BNSL, we used a novel BDD-based counter to deal with sums of logarithms. Finally, due to the overarching goal of practicality, we implemented each of our SLIM-based algorithms and performed experimental evaluations. These evaluations verified and confirmed that the theoretical promises still hold under the practical challenge of real-world problems. In each case, we showed that the algorithms we develop perform favorably in comparison to state-of-the-art algorithms.

Contents

CHAPTER 1

Gentle Introduction

Example is the school of mankind, and they will learn at no other.

— Edmund Burke, “Letters on a Regicide Peace”

In this section, we give a gentle introduction to the main problem of interest. This chapter is intended to be fairly accessible and didactic with ample diagrams and examples to make clear the underlying logic and intuition. To the reader who is already comfortable with these concepts, we recommend skipping to Chapter 3 where we back these somewhat informal statements with formal notation and terminology.

Note

Boxes like these provide helpful pointers, clarifications, intuitions, and examples.

Note

The words on the margins guide the reader to the definition of the respective terms and can be used in conjunction with the Definition Index at the end of this manuscript to locate or recall definitions. *Emphasized terms* in the prose denote definitions.

1.1 Graph Theory

node
vertex
edge

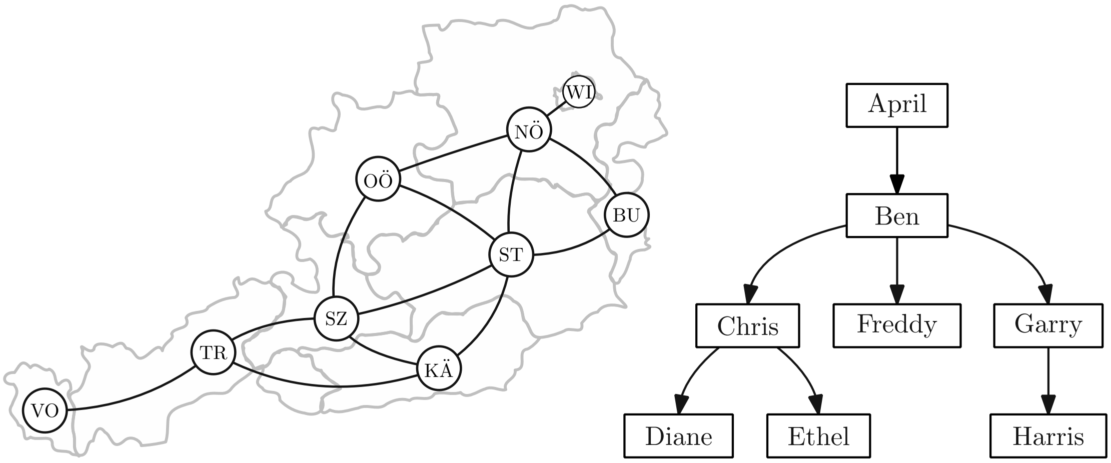
Graphs[1] (also known as networks) are one of the most fundamental objects in Computer Science. Graphs represent relations between entities. This simple definition enables graphs to be used for a wide variety of applications. The entities are called either *nodes* or *vertices*, and the relations between them are called *edges*. Some real world examples which can be represented by graphs are:

- State neighborhood graph, where each state (or *«Bundesland»*) of Austria is an entity and the property of sharing a border is the relation;

- Organization hierarchy diagram, where each employee is an entity and the property of being a superior is the relation.

undirected edge
directed edge
directed graph
undirected graph

The represented relations can either be symmetric relations (called *undirected edges*) like in the neighborhood graph, (if u shares a border with v, so does v share a border with u); or *oriented* in a particular direction (called *directed edges*) like in the hierarchy diagram (if u is v's superior, v cannot be u's superior). In this thesis, we only deal with either *directed graphs* where all relations are directed; or *undirected graphs*, where all relations are undirected. Please see Figure 1.1 for a visualization. Note that, in Figure 1.1a, the state of Tirol (TR) consists of two disjoint land masses.



(a) State neighborhood graph

(b) Organization hierarchy diagram

Figure 1.1: Examples of graphs

[1] not to be confused with graphs of functions, as in plots or charts, which represent how a function's output value changes with respect to the input argument

Note

In visual representations of graphs, the positions of the vertices do not matter and neither do the shapes and curves of edges. The only information the drawing is meant to convey is which vertices are connected to each other, either by undirected edges (plain lines) or by directed edges (lines with arrowheads).

Unless otherwise specified, we use the term 'graph' as a shorthand for undirected graphs. We refer to edges in a directed graph as *arcs* (depicted as arrows $u \rightarrow v$ in figures) and henceforth reserve the term 'edge' as a shorthand for undirected edges (depicted as lines $u — v$ in figures). A graph *over* a set S is a graph where the vertices belong to the set S. An edge between vertices u and v is *incident* on vertex u and vertex v. A *subgraph* of a graph G is a graph obtained by deleting some vertices or edges from G. Note that when we delete a vertex, it also implies the deletion of the edges incident on that vertex because otherwise we would be left with an edge that has only one endpoint.

arc
graph over a set
incident
subgraph

In an undirected graph, a vertex is a *neighbor* of another vertex (or is *adjacent* to another vertex) if there is an edge between them. The *neighborhood* of a vertex v is the set of all other vertices that are adjacent to vertex v. The *degree* of a vertex is the number of neighbors of that vertex. For example, in Figure 1.1a, the neighborhood of Oberösterreich (OÖ) consists of Steiermark (ST), Salzburg (SZ) and Niederösterreich (NÖ) and thus Oberösterreich (OÖ) has degree 3. In a directed graph, if there is an arc going from vertex u to v, then u is a *predecessor* of v and v is a *successor* of u. The *in-degree* and *out-degree* of a vertex is the number of predecessors and number of successors respectively. So, Diane and Ethel are successors of Chris and Ben is his predecessor. Thus, Chris has an out-degree of 2 and an in-degree of 1.

neighbor
adjacent
neighborhood
degree
predecessor
successor
in-degree
out-degree

Another natural concept in the context of graphs is a path. A *path* is a sequence of vertices such that each successive vertex in the sequence is a neighbor (or successor) of the previous vertex. In Figure 1.1a, WI–NÖ–ST–KÄ forms a path. Extending the concept of successors and predecessors, if there exists a path from u to v in a directed graph, u is an *ancestor* of v and v is a *descendant* of u. A path is termed *simple* if no vertex is repeated, and in this thesis, we only deal with simple paths. A *cycle* is a simple path except that the last vertex is the same as the first vertex. For example, BU–ST–OÖ–NÖ–BU is a cycle, but BU–ST–OÖ–NÖ–ST–BU is not a cycle because the intermediate vertex ST appears twice. An *acyclic* graph is a graph devoid of cycles. Combining these concepts, we arrive at the oft-encountered and aptly named *Directed Acyclic Graphs* (DAGs). A graph is *connected* if there exists a path from any vertex of the graph to any other vertex of the graph. Lastly, we only deal with simple graphs in this thesis, i.e., we neither allow multiple edges (or arcs) between the same two vertices, nor an edge (or an arc) from a vertex to itself.

path
ancestor
descendant
simple path
cycle
acyclic graph
DAG
connected graph

1.2 Bayesian Network Structure Learning

In this section we first introduce the main object of study of this thesis, Bayesian Networks. We then describe the base problem concerning Bayesian Networks which we will tackle later in Chapter 4. After that, we consider variations of this base problem in Chapters 5 and 6.

We refer to the book by Neapolitan and Jiang [NJ07] for a more well-rounded introduction to Bayesian Network Structure Learning.

1.2.1 Bayesian Networks

RV
random variable
domain

A *random variable* (RV) is a variable that takes values from a pre-defined set (called *domain*) with some probabilities, and represents the outcome of a random event. For example, let D be a random variable representing the outcome of rolling a dice, and so the domain of D is the set $\{1, 2, 3, 4, 5, 6\}$. Each of this event is equally likely and hence each event has the same probability of 1/6 or about 16%. In this thesis, we only deal with *categorical random variables*, i.e., variables with finitely many possible values. Using categorical random variables, we can capture simple real-world phenomena like a die roll, the grade of a student, or a patient's tendency to contract a disease. However, to really leverage random variables and broaden the scope of the real-world phenomena that we can model accurately, we need some way to look at several random variables simultaneously as well as how they interact with each other. For instance, we need something that helps us answer questions like "How does a student's family income affect their grade?"

categorical RV

BN

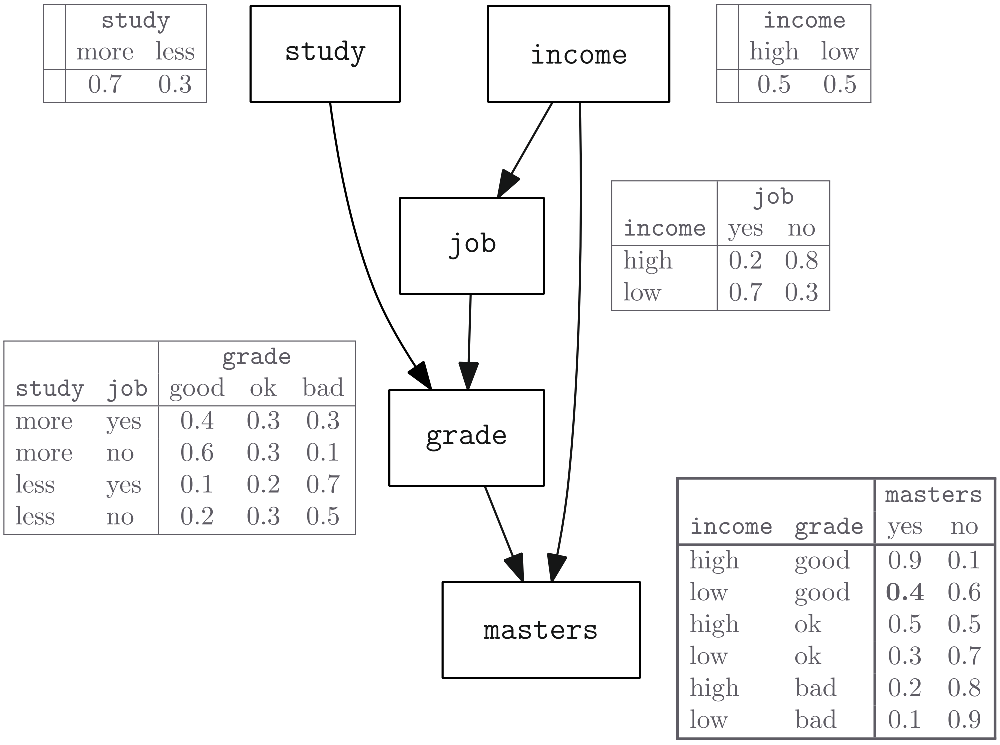
A *Bayesian Network* (BN) allows us to accomplish this. Bayesian Networks were introduced by Pearl in 1985 and later formalized by Pearl [Pea88] and Neapolitan and Jiang [NJ07]. A Bayesian Network consists of a DAG over categorical random variables as nodes, together with local parameters, i.e., *Conditional Probability Distribution* (CPD) tables—or probability tables for short—for each node. An arc between two nodes of the DAG denotes that the random variables corresponding to those nodes are correlated or interdependent, and the probability tables quantify this dependence. The DAG by itself is called the *structure* of the Bayesian Network and all the probability tables collectively are called *parameters* of the Bayesian Network.

CPD
structure
parameters

Consider the example of a Bayesian Network over five random variables shown in Figure 1.2. In this example, we are analyzing the likelihood of a student pursuing a master's program (`masters` node), in the context of various other attributes detailed in Table 1.1. Notice, for instance, how there is an arc from `income` to `job` to capture that the student's family income affects their tendency to take up a part-time job. Similarly, the arc from `income` to `masters` says that the student's family income affects their tendency to pursue a master's program.[2] Further, the (probability) table to the right of `income` specifies this tendency precisely. This table is interpreted as follows: if the family income is high, then there is a 20% chance that the student takes up a job and 80% chance that they do not; on the other hand, if the family income is low, then there is a 70% chance that the student takes up job and 30% chance that they do not. As another representative example, consider the number 0.4 in the bottom right table (corresponding to `masters`), this expresses that there is a 40% chance that a student with low family income and good grades would pursue a master's program.

Name of RV	Domain of RV	Description
`study`	$\{\text{more}, \text{less}\}$	how much the student studies
`income`	$\{\text{high}, \text{low}\}$	the student's family income
`job`	$\{\text{yes}, \text{no}\}$	does the student have a job on the side
`grade`	$\{\text{good}, \text{ok}, \text{bad}\}$	how well the student performs
`masters`	$\{\text{yes}, \text{no}\}$	whether the student pursues a master's program

Table 1.1: Description of random variables used in Example 1.2



	study	
	more	less
	0.7	0.3

	income	
	high	low
	0.5	0.5

	job	
income	yes	no
high	0.2	0.8
low	0.7	0.3

		grade		
study	job	good	ok	bad
more	yes	0.4	0.3	0.3
more	no	0.6	0.3	0.1
less	yes	0.1	0.2	0.7
less	no	0.2	0.3	0.5

		masters	
income	grade	yes	no
high	good	0.9	0.1
low	good	**0.4**	0.6
high	ok	0.5	0.5
low	ok	0.3	0.7
high	bad	0.2	0.8
low	bad	0.1	0.9

Figure 1.2: Example of a Bayesian Network

[2] In this setting, we assume that the master's program needs to be paid for by the student and is not free of cost.

Note

It is subtle but worth noting that we model certain deterministic phenomena (like the student's income or how much they study) as *random* variables in order to capture the stochastic characteristics across a data set. In other words, this Bayesian Network does not represent the story of a single student but rather it tries to capture the underlying patterns and dependence of these attributes in a general sense, as observed for several students.

1.2.2 Structure Learning of Bayesian Networks

Now that the concept of Bayesian Networks is clear, we can introduce the core problem of this thesis in an albeit simplified form:

BN Learning: Given observations of a set X of random variables, find the Bayesian Network over X which is most likely to have produced the given observations.

Note

We assume that there always exists an underlying Bayesian Network that was responsible for producing the given observations. This is a reasonable assumption considering the expressive power afforded to us by Bayesian Networks in terms of capturing probabilistic relations between variables.

instance

An observation or an *instance* refers to an instantiation or assignment to the complete set of random variables X. For example, in Figure 1.2, an instance could be

$$\texttt{study} = \text{more}, \texttt{income} = \text{low}, \texttt{job} = \text{no}, \texttt{grade} = \text{good}, \texttt{masters} = \text{yes}.$$

BNSL
structure learning
parameter learning

A typical strategy to tackle the BN Learning problem is to split the task into two steps, namely structure learning and parameter learning. *Bayesian Network Structure Learning* (BNSL) or simply *structure learning* is the problem of determining just the structure (i.e., the DAG) of the Bayesian Network. Since the structure by itself isn't sufficient to evaluate the quality of a Bayesian Network, one usually employs an auxiliary score function as a proxy for the quality of the Bayesian Network. Here, 'higher quality' means that the Bayesian Network represents the given observations more accurately. Thus, the precise problem in structure learning is to find the structure that maximizes said score function. On the other hand, *parameter learning* is the problem of determining the probability tables given the structure of the Bayesian Network. In this thesis, we only concern ourselves with structure learning. In general, parameter learning is much simpler than structure learning as it does not require guessing whether two random variables are related or not, rather, it just involves quantifying the dependencies specified in the given structure.

1.2.3 Inference on Bayesian Networks

Once we have learned a Bayesian Network, both the structure and the parameters, we can use them as machines to make predictions about the random variables involved. This is known as performing *inference* or *reasoning* on the Bayesian Network. Following are examples of probabilistic queries (along with their answers) that one can answer using a Bayesian Network (in this case we use the Bayesian Network from Figure 1.2):

inference
reasoning

- What is the probability that a student with low `income` and no `job` who pursued `masters`, had `studied` more? (77%)
- What is the probability that a student pursues `masters`? (42%)
- What is the most likely `grade` for a student with a `job`? (`grade` = bad with 42% chance)

The task of performing inference on Bayesian Networks is quite difficult in general. There exist several techniques for answering such queries exactly, but all of them require time exponential in the size of the network (i.e., if the network grows in size by 10-fold, the running time grows by 2^{10}-fold). As a result, several approximate methods have also been developed which avoid the impractical runtimes by compromising accuracy. Exploring the inner workings of these techniques (be it exact or approximate) is beyond the scope of this thesis. However, what is relevant in this context is that, if we restrict ourselves to exact inference techniques, then the difficulty of performing inference can be mitigated if the Bayesian Networks possess a special property which we uncover in the next section.

1.3 Tree Decompositions

This section introduces the concept of tree decompositions and the related metric called treewidth. The importance of treewidth for inference in Bayesian Networks is discussed towards the end of the section.

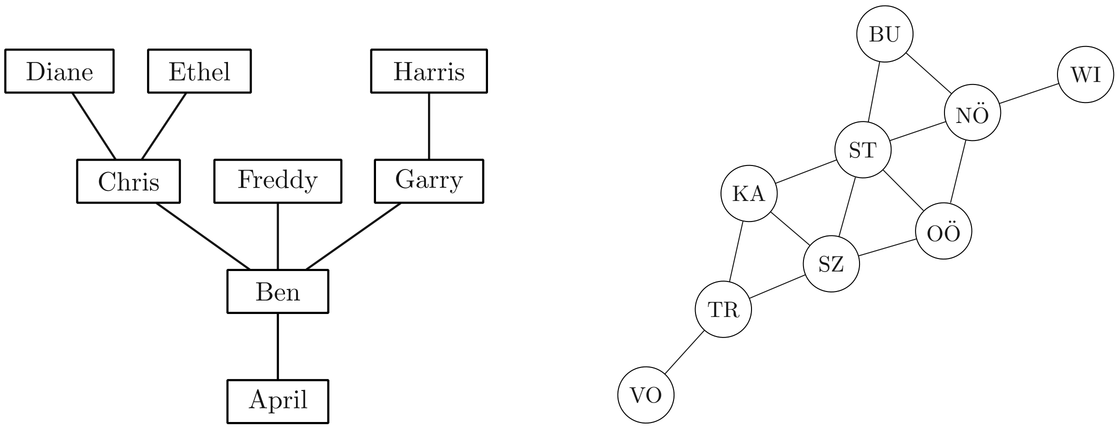
We refer to the book by Cygan et al. [Cyg+15, Chapter 7] for a gentle introduction to tree decompositions.

> **Note**
>
> It suffices to develop a rough intuition of the concept of treewidth and what it correlates to, in order to understand the connection to Bayesian Networks and the core problem of this thesis (introduced at the end of this section). An in-depth understanding of tree decompositions, although helpful, is not strictly necessary.

1.3.1 Definition

Algorithmic problems related to graphs have been widely studied in Computer Science. A general pattern observed in the process is that some problems are much easier to solve when the involved graphs are so-called trees as opposed to general graphs. A *tree* is a connected undirected graph without cycles (see Figure 1.3 for an example). One such example of a problem that is easier on trees is performing inference on Bayesian Networks. Quite often real-world problems do not come in the form of trees, however they may still turn out to be easy to solve. This hints at the potential crudeness in simply binning graphs into 'trees' and 'not trees', i.e., could there be a way to express the tree-likeness of a graph which is more fine-grained than a yes/no answer?

tree



(a) Undirected version of the hierarchy diagram (tree)

(b) Neighborhood graph of the states of Austria (not a tree)

Figure 1.3: Examples of a graph which is a tree and one which is not

An answer to this question came in the form of tree decompositions, first investigated by Bertele and Brioschi [BB73] and Halin [Hal76]. Later, Robertson and Seymour [RS84] reintroduced this notion in the form that it is studied today. Tree decompositions provide a granular answer to the question "How close is a given graph to being a tree?" consequently answering the question of how hard certain problems are for the given graph.

Note

Contrary to how their real-life counterparts look, in Computer Science, trees are typically drawn with the narrower end on the top and the broader end on the bottom. Figure 1.3a, however, has been exceptionally drawn upside-down to elucidate the reason such structures are called trees.

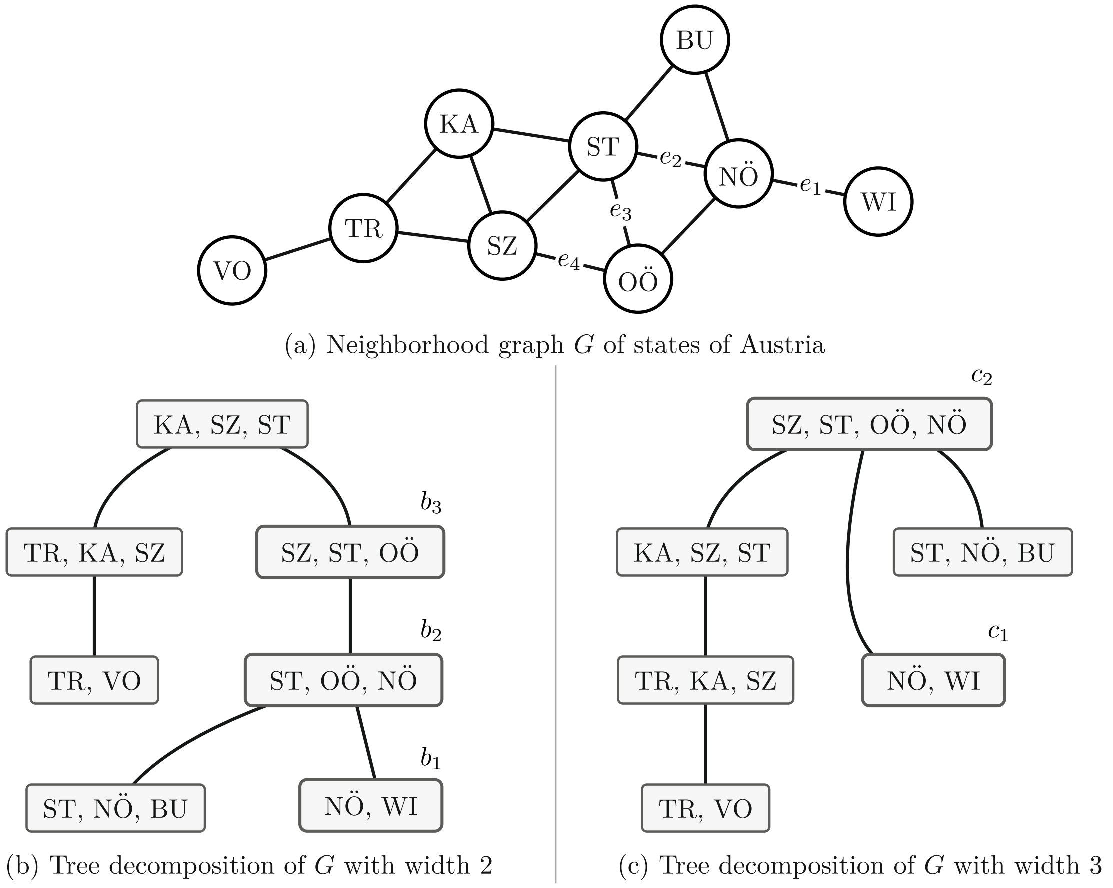
tree decomposition
bag

A *tree decomposition* of a graph G is an auxiliary graph (more specifically, an auxiliary tree), where each node is a *bag* consisting of vertices from the original graph G (see

Figure 1.4 for an example). We then impose two more conditions:

T1. for every edge, there should be at least one bag which contains both the endpoints of this edge,

T2. for every vertex, if we delete every bag that doesn't contain that that vertex, along with all the edges incident on these bags, we should be left with a connected subtree.



(a) Neighborhood graph G of states of Austria

(b) Tree decomposition of G with width 2

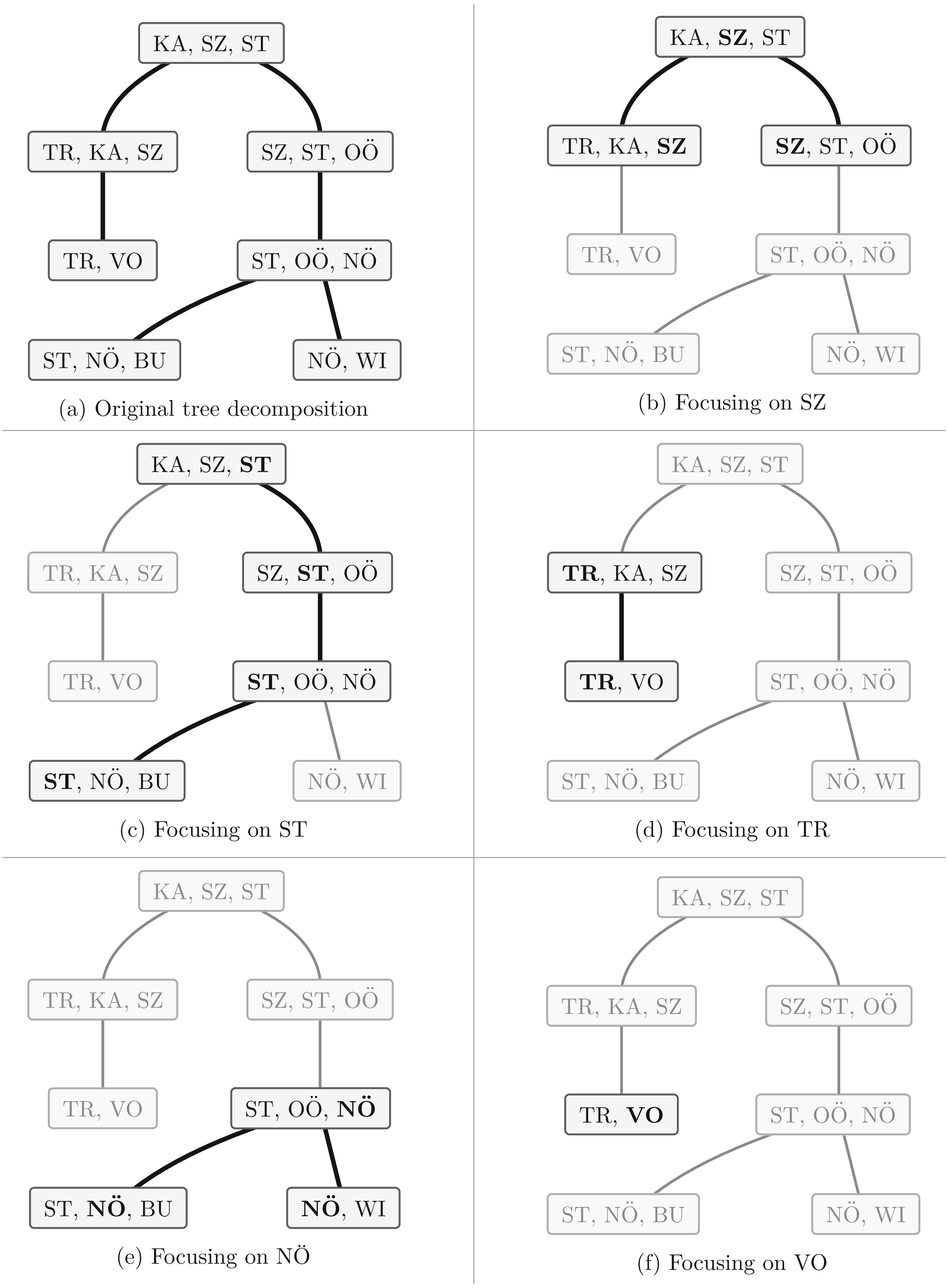
(c) Tree decomposition of G with width 3

Figure 1.4: Examples of tree decompositions

There are several possible tree decompositions for a given graph. For each such tree decomposition, we define a metric called *width* which is one less than the number of vertices in the largest bag. Finally, we define the *treewidth* of a graph to be the lowest possible value of 'width' that can be achieved by some tree decomposition. The definition of treewidth might seem obscure at first, but, as we will see, its applications help cement the fact that it has all the characteristics of the fine-grained measure we were seeking.

width

treewidth



(a) Original tree decomposition

(b) Focusing on SZ

(c) Focusing on ST

(d) Focusing on TR

(e) Focusing on NÖ

(f) Focusing on VO

Figure 1.5: Demonstrating property T2

1.3.2 Example

Figure 1.4 shows two possible decompositions of the Neighborhood graph G from before. We demonstrate that these are indeed valid tree decompositions.

In Figure 1.4b,

- the maximum number of vertices in any bag is 3, and hence this decomposition has width 2,
- edge e_1 is contained in bag b_1,
- edges e_2 and e_3 are contained in bag b_2,
- edge e_4 is contained in bag b_3.

In Figure 1.4c,

- the maximum number of vertices in any bag is 4, and hence this decomposition has width 3,
- edge e_1 is contained in bag c_1,
- edges e_2, e_3 and e_4 are all contained in bag c_2.

And to demonstrate the validity of the second condition T2, we highlight the subtrees that remain upon deletion of all the bags not containing a particular vertex in Figure 1.5. Since this always yields connected subtrees, condition T2 is satisfied.

As further general examples, if we consider graphs with a fixed number n of vertices then

- connected acyclic graphs, i.e., trees always have a treewidth of 1,
- cycle graphs always have treewidth 2,
- $\sqrt{n} \times \sqrt{n}$ grids have treewidth $\sqrt{n}$, and
- completely connected graphs (or *cliques*) have treewidth $n - 1$.

clique

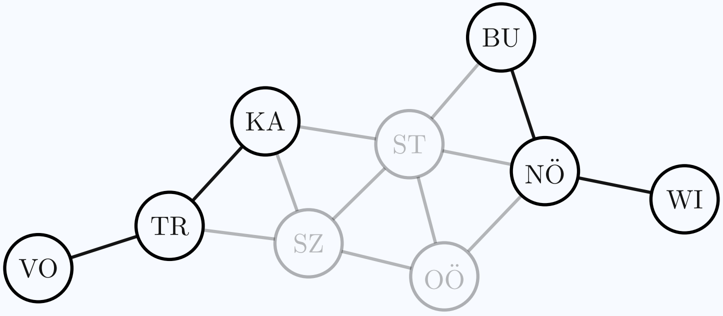
We refer to Figure 1.6 to see how these different types of graphs look and also to develop an intuition of what it means for a graph to have high treewidth.

Note

Roughly speaking, sparse graphs tend to have low treewidths and denser graphs tend to have high treewidths. Using this newly acquired intuition, it should be easy to see that removing a vertex from a graph can not increase its treewidth.

We notice how if the vertices occurring in a bag are deleted from the graph, we are always left with two disconnected subgraphs. And the disconnectedness is guaranteed due to property T2, which ensures that it is impossible for a vertex to appear in both subgraphs while simultaneously being absent from the deletion set.

As an example of the disconnectedness property, in Figure 1.4b, let's arbitrarily pick the bag $\{\mathrm{SZ}, \mathrm{ST}, \mathrm{OÖ}\}$. Deleting these vertices from the graph leaves behind two disconnected subgraphs with vertices KÄ, TR, VO on one side and vertices NÖ, BU, WI on the other side. It is then easy to see intuitively why dense and well-connected graphs would have higher treewidth in comparison to sparse and weakly-connected graphs, since they would need larger bags (i.e., more vertices) to be able to disconnect the original graph.



1.3.3 Relevance to Bayesian Networks

Returning to the topic of Bayesian Networks, to understand the link between tree decompositions and Bayesian Networks, we require one more concept—treewidth of Bayesian Networks. Since tree decompositions are only defined for undirected graphs and Bayesian Networks are inherently directed, we define the treewidth of a Bayesian Network in terms of its so-called moralized graph. The time required to perform inference on a Bayesian Network heavily depends on the sizes of the probability tables at each node. Thus, a desirable attribute in the definition of treewidth of a Bayesian Network is that

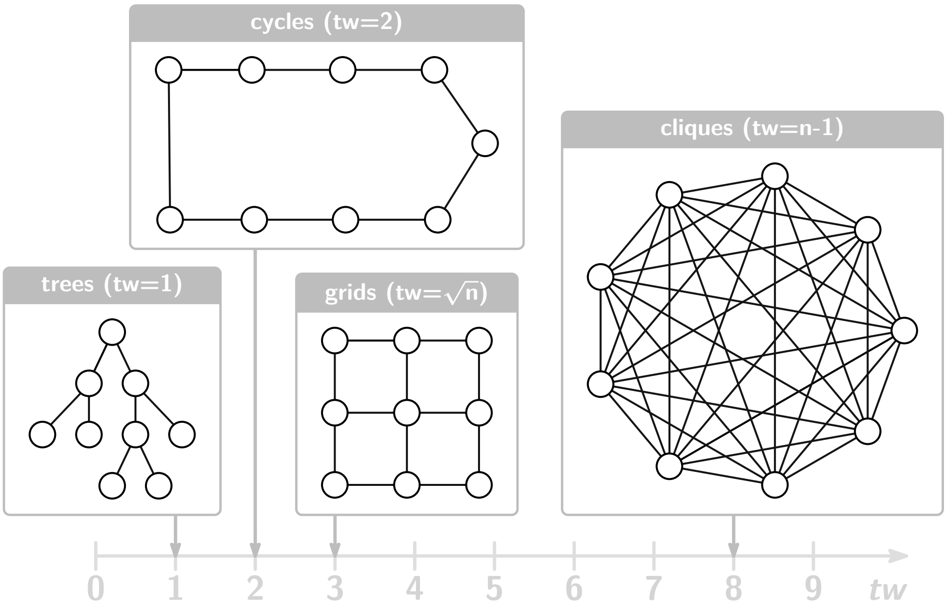
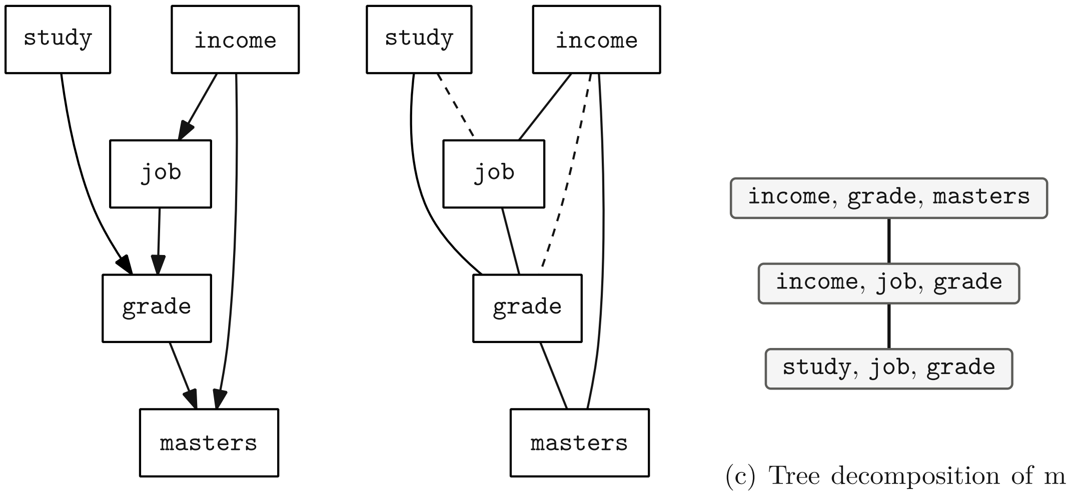


Figure 1.6: Examples of common types of graphs (all with $n = 9$ vertices) along with their respective treewidths

the treewidth should be proportional to the sizes of the probability tables. Moralization transforms the directed graph into an undirected while also embedding the sizes of the probability tables that appear at each node as cliques. To *moralize* a Bayesian Network, we first convert all directed arcs (u, v) into undirected edges $\{u, v\}$ and then insert edges (called *moral edges*) between any two vertices that have a common successor (shown as dashed lines in Figure 1.7b). Notice how if a node has lots of parents, the size of the probability table at that node is also large, and accordingly there is a large clique at that node in the moralized BN due to moral edges between each of the parents of that node. Please see Figure 1.7 for an example of a tree decomposition of a Bayesian Network.

moralization

moral edge



(a) BN before moralization

(b) Moralized graph of BN

(c) Tree decomposition of moralized graph with width 2

Figure 1.7: Moralized Graph of the example BN and (one of) its tree decomposition

As we have already seen, there are two main tasks related to Bayesian Networks, learning a Bayesian Network and performing inference on a Bayesian Network. In a typical real-world workflow, the learning task is only performed once at the beginning, after which the learned Bayesian Network is used to churn out predictions by performing inferences using available partial data. For instance, consider the scenario of learning the relation between a student's income and their tendency to pursue masters. One needs to learn a Bayesian Network over relevant variables from the dataset just once, after which it can be deployed to make predictions over several students with differing incomes. Thus, it makes sense to invest a little more time in the learning phase if it ensures better throughput during the inference phase.

In this thesis, we disallow approximate inference solutions and only consider exact inference solutions. Unfortunately, for a general Bayesian Network, in the worst-case, it may take exponentially long to perform a single exact inference, thereby foiling the standard workflow described above. This is where the treewidth of Bayesian Networks comes in. The time required to perform inference on a Bayesian Network with n random variables and treewidth w is proportional to $2^w \cdot \text{poly}(n)$, i.e., there is an exponential dependence on the treewidth and only polynomial dependence on the number of random variables. The number of random variables is governed by the input and hence out of our control. But, if we limit the treewidth of the Bayesian Networks to small values, we can guarantee quicker inferences. We show the updated problem definition below, and in the next chapter, we lay out the toolkit that will help us solve this problem.

***Bounded Treewidth* BN Learning:** Given observations of a set X of random variables, *and a bound* w for the treewidth, find a Bayesian Network over X *with treewidth* w which is most likely to have produced the given observations.

CHAPTER 2

Algorithms for BNSL

Truth is much too complicated to allow anything but approximations

— John von Neumann, “The Mathematician”

In this chapter we take a gentle look at the algorithms and techniques that pop up frequently throughout the thesis, either as means to solve the problem at hand or as stepping stones to build upon. Similar to the previous chapter, we cover these topics in more technical detail in Chapter 3.

2.1 Maximum Satisfiability

This section introduces a tool called MaxSAT solver which sits at the center of the framework we employ for solving the BNSL problem and its variants.

As is common in the field of Computer Science, it is preferable to reduce any data and bring it into a binary form of 1 or 0, yes or no, on or off. Each bit of this data can then be represented by a so-called *Boolean variable* which has 2 possible states, `TRUE` and `FALSE`. Similar to operations such as addition, multiplication, and subtraction of numbers, we can define the *conjunction*, *disjunction*, and *negation* operations (denoted $\wedge, \vee$ and $\neg$ respectively) for Boolean variables. $p \wedge q$ is only `TRUE` if both p and q are `TRUE`, $p \vee q$ is `TRUE` if either one of p or q (or both) are `TRUE` and $\neg p$ is only `TRUE` if p is `FALSE`.

Boolean variable

conjunction
disjunction
negation
$\wedge, \vee, \neg$

Now we can construct formulas, called *propositional formulas*, using Boolean variables and the operators defined above. For instance, to describe the days on which I go to the

propositional formula

university, I would use the expression

$$(\text{weekday} \wedge \neg\text{holiday}) \vee (\text{saturday} \wedge \text{deadline}).$$

SAT

Translated to plain English, this expression says that I work on weekdays when it's not a holiday or on Saturdays only if there's a deadline approaching. With the right variables and abstractions, such formulas are capable of expressing even complex conditions like the acyclicity of a graph, or the possibility of successfully allocating the lecture rooms in university to the courses, or if the treewidth of a graph is below a certain bound. It then makes sense to pose questions about whether the variables in a given expression can be assigned in a certain way such that the expression evaluates to `TRUE`. This problem is called the *Propositional Satisfiability* (SAT) problem. Despite its innocent appearance, SAT forms a cornerstone of Theoretical Computer Science.

satisfied
assignment
encoding

An expression or formula is *satisfied* by an *assignment* if setting the variables according to the assignment results in the formula evaluating to `TRUE`. An *encoding* of a particular instance of a YES/NO problem is a propositional formula such that the formula is satisfiable if and only if the answer to that instance of the problem is YES. For example, say we have a particular graph G, and we are interested in the YES/NO question of whether G is acyclic. Further, let $\mathcal{F}$ be a formula encoding this question. This means that formula $\mathcal{F}$ is satisfiable if and only if G is acyclic. Note that the formula $\mathcal{F}$ is not a general formula for the question but rather a tailor-made formula to test the acyclicity for the particular graph G.

CNF
clause

Propositional formulas formed by taking a conjunction over disjunctions, are called *Conjunctive Normal Form* (CNF) formulas. Each individual disjunction is called a *clause*. An example of such a formula with five variables and four clauses is shown below

$$(x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_2 \vee x_5).$$

SAT solver
MaxSAT
soft clause
hard clause
clause weight
MaxSAT solver

There exist tools/software, called *SAT solvers*, which are capable of solving the satisfiability problem for CNF formulas encoding real-world problems with millions of propositional variables and tens of millions of clauses in under 2 hours, i.e., given a CNF formula $\mathcal{F}$ these tools can determine rather quickly if there exists some assignment of the variables that results in $\mathcal{F}$ evaluating to `TRUE`. In some cases, it is very difficult to satisfy *all* the clauses. Thus arose the need for the *Maximum Satisfiability* (MaxSAT) problem, where the input consists of two types of clauses: *soft clauses* and *hard clauses*. Any valid assignment must satisfy all the hard clauses and some subset of the soft clauses. The soft clauses are tagged with a numerical *weight* (or profit). The goal is to find an assignment that maximizes the sum of weights (or total profit) of the satisfied soft clauses. Similar to SAT solvers, there exist tools called *MaxSAT solvers* which can solve the MaxSAT problem for formulas involving millions of variables.

MaxSAT solver timeout

In our particular use case, we also make use of the *timeout* functionality supported by some MaxSAT solvers. For such solvers, it is possible to interrupt the solver in the middle of the solving process and the solver spits out the best assignment it has found so far. It is possible that this solution does not achieve the optimum profit, however, as we discover in the next section, our framework gets much more mileage from quicker and slightly suboptimal solutions as compared to slow yet optimum solutions.

2.2 SAT-based Local Improvement Method

In this section we describe the general framework which we use as the weapon of choice to tackle the base BNSL problem and its variants in the forthcoming chapters. The framework in itself does not constitute an algorithm but is rather a template to design algorithms.

2.2.1 Introduction

Let us first take a look at the typical lifecycle of a problem (say P) in Computer Science (shown in Figure 2.1). We first ask "Is it possible to develop an efficient algorithm to solve P?" If the answer is yes, it is proven by coming up with an algorithm. Here, 'efficient algorithm' means an algorithm that runs in polynomial time in the size of it's input. If the answer to the question is 'No', we call the problem *intractable*. In this case, there are several potential follow-up directions. We discuss the two most relevant directions. One can either develop a heuristic algorithm that is efficient but is only likely to produce suboptimal solutions (i.e., 'close-enough' solutions), as opposed to optimal solutions. Alternatively, one can express the problem in the form of a SAT/MaxSAT encoding and then pass on the encoding to a SAT/MaxSAT solver.

intractable problem

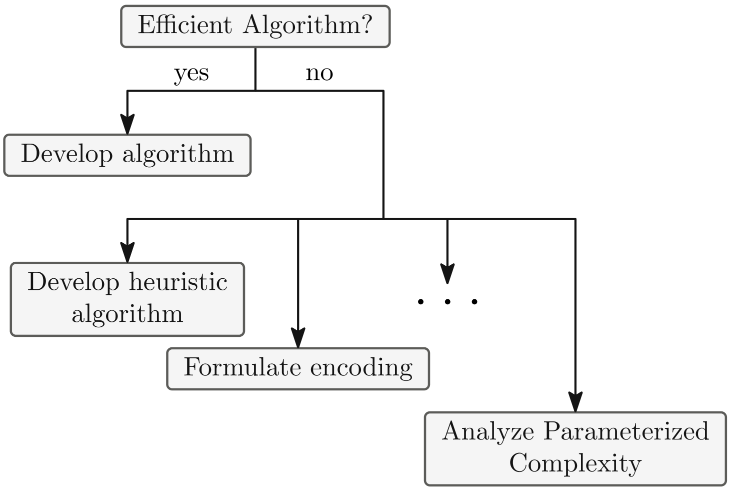


Figure 2.1: Lifecycle of a problem

Both these approaches have their own pros and cons. Heuristic algorithms are usually quite *scalable*, i.e., they perform reasonably well even for huge inputs, however, this comes at the expense of lower solution quality. The encoding-based approach, on the

scalability

other hand, guarantees optimum solutions but usually only produces solutions for very small instances.

SLIM

The primary driving force behind the conception of the *SAT-based Local Improvement Method* (SLIM) framework is to capitalize on the advantages of both these techniques while somewhat balancing out their drawbacks. The end product is an algorithm which retains the scalability of the heuristic approach while still providing better quality solutions by leveraging the encoding-based approach.

> **Note**
>
> The following paragraphs aim to establish a high-level intuition about the SLIM framework in its most general form and hence, we refrain from exemplifying the BNSL problem. We will, however, come across concrete instantiations of the framework for the BNSL problem and its variants in Chapters 4–6. Further, it might be helpful to think of SLIM as a post-processing algorithm, which takes as input the heuristic solution and polishes it to return a better solution.

global solver
global solution
budget, β
subproblem
local solver
replacement consistency

The general idea is to first use the *global solver* (usually the heuristic algorithm) to come up with a (possibly low-quality) initial solution to an input instance. Since we consider small parts of this solution, we refer to the full solution of the original input instance as the *global solution*. We then repeatedly extract small local parts (of size at most some *budget* β) from the global solution and find better solutions to these *subproblems* (within a fixed timeout) using the *local solver* (usually the encoding). Then, we replace the local solution in the global solution with its improved counterpart. We demonstrate this idea with a toy example in Figure 2.2 (walkthrough provided below). It is imperative to take care that the improved counterpart can be patched back safely into the global solution. The property of maintaining the validity of the global solution after replacement is called *replacement consistency*. To accomplish this, one needs to ensure that the newly improved local solution mimics the extracted local solution in terms of its interaction with the rest of the global solution.

anytime algorithm
global timeout
local timeout

Throughout this thesis, we will constantly come across so-called *anytime algorithms*, i.e., algorithms that don't have a fixed termination, rather, they can keep running indefinitely, finding better and better solutions along the way. All algorithms based on the SLIM framework are anytime algorithms. It is preferable to use an anytime algorithm as the global solver, as it allows for a fair and straightforward comparison of the performance of the SLIM algorithm to the heuristic algorithm by itself. For practical use cases, usually one decides a *global timeout*, i.e., a total execution time limit for anytime algorithms. The moment this time limit is reached, the anytime algorithm returns the best solution it found so far. Thus arise two types of timeouts, the global timeout or total time limit and the *local timeout* which limits how long we run the local solver on a local subproblem (internally, this typically translates to the MaxSAT solver timeout). Common values for these timeouts in our algorithms are 30 to 60 minutes for the global timeout and 2 to 20 seconds for the local timeout.

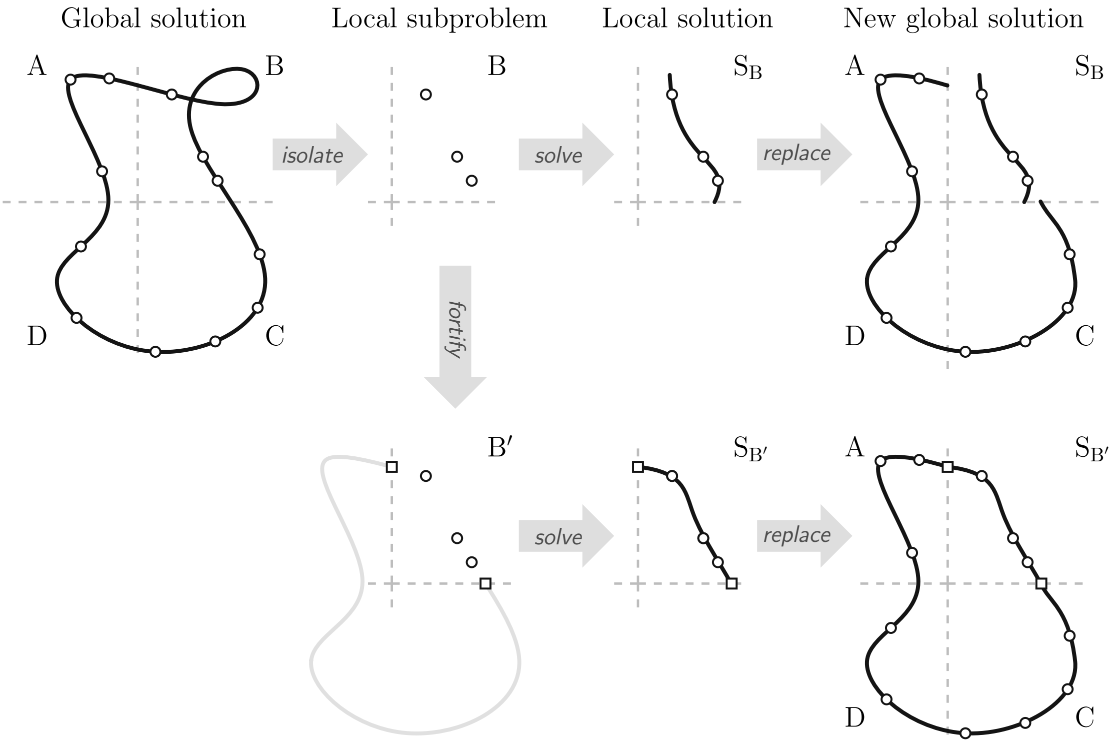


Figure 2.2: Higher-level overview of SAT-based Local Improvement Method (SLIM)

2.2.2 A Toy Example

Now, we analyze the example from Figure 2.2 in detail. In the dummy problem here, we are given a set of points (depicted as ○) and asked to draw a continuous line through all the points and return to the start. Additionally, we are required to minimize the length of the resulting line.

We begin with an initial global solution provided by the global solver. From visual inspection, one can see that there is some scope for improvement (by getting rid of the loop in the top right). We arbitrarily partition the points into 4 quadrants A, B, C, and D. We then select quadrant B (consisting of 3 points) as the local part and treat this quadrant in isolation as a new subproblem. We then pass on this subproblem to the local solver. There are several possible solutions for this subproblem. Say, we first obtain a solution S_B, which we then try to patch back into the global solution. But this results in an invalid solution altogether because the line is no longer continuous, i.e., replacement consistency is violated. This is because the new subproblem did not properly capture the interaction of the old local solution with the rest of the solution, more specifically in this case, the new subproblem is unaware of the points at which the old local solution met the rest of the global solution. We thus come up with an alternate fortified subproblem B' which incorporates information about such interaction (depicted as □ at the boundaries of the local subproblem) to ensure replacement consistency. So, any solution to the fortified instance $\mathrm{S}_{\mathrm{B}'}$ is guaranteed to be compatible with the rest of the global solution. Computing solutions to such fortified instances usually requires upgrading the local solver as well, which is one of the key challenges when coming up with a SLIM algorithm.

2.2.3 Past Work

SLIM was introduced by Lodha, Ordyniak, and Szeider [LOS16] and has been applied to a wide range of problems [PS21c; PS22; LOS19; LOS17; FLS17; PS20; SS21; Sch22; SS22; RSS23]. The diversity of these applications speaks volumes to the versatility and expressiveness unlocked by the SAT-based local solver at the heart of SLIM. A CNF formula, although seemingly simple, is very powerful at capturing and encoding the various constraints one encounters in the problems listed above. The main challenge in each of these papers was devising the fortified encoding so as to ensure replacement consistency, in addition to (optionally) formulating the encoding for the basic version of the problem. A minor remark here is that most of the past applications used SAT encodings and plain SAT solvers while in our applications we use MaxSAT encodings and MaxSAT solvers. At times, we construct a generalized MaxSAT encoding based on several SAT encodings thus nullifying the need for multiple invocations of the SAT solver in favor of a single invocation of the MaxSAT solver.

2.3 Heuristics for the BNSL Problem

In this section, we describe the algorithms which act as candidate heuristic algorithms for computing the initial solution to the BNSL problem (detailed in Chapter 4). Since we modify and extend these algorithms in Chapters 5 and 6, a good understanding of the brass tacks is helpful.

For the original discourse on these algorithms along with the empirical findings, we refer to the papers by Scanagatta et al. [Sca+16; Sca+18].

2.3.1 k-tree Construction

Both the heuristic algorithms rely on a common subroutine to ensure that the resulting Bayesian Networks respect a given treewidth bound. A subtle point worth noting here is that we are not concerned with computing a tree decomposition or calculating the treewidth of a given graph, instead, we are given a treewidth bound k and are tasked with constructing a graph whose treewidth (or constructing a BN whose moralized graph's treewidth) is guaranteed to be at most k by design. This is achieved by constructing the graph sequentially such that at each step the partially constructed graph is a subgraph of a k-tree. A *k-tree* is an edge-maximal graph of treewidth k, i.e., the addition of any edge to the graph would increase the treewidth of the graph [Sca+18]. k-trees are useful

k-tree

to us in this scenario because any subgraph of a k-tree has treewidth at most k[1], and a k-tree on n vertices can be easily 'grown' into a k-tree on $n+1$ vertices. To *grow a k-tree*, we pick a clique of size at most k from the original k-tree and add the new vertex as being adjacent to all the vertices from this clique.

growing k-trees

2.3.2 k-greedy Algorithm

We now explain the k-greedy algorithm [Sca+16] which greedily constructs a structure with treewidth at most k for the BN over random variable set X. We start by sampling a random ordering σ over all the random variables. Intuitively, this random ordering acts as the topological ordering of the resulting DAG (and also as the elimination ordering of its moralized graph). Recall that we also have access to a score function f, which we use implicitly when we say 'the best DAG'. Since any graph with $k+1$ vertices has treewidth at most k, we pick the first $k+1$ vertices from σ and construct the best DAG possible on these vertices (either approximately or exactly). Notice that this is a valid k-tree. After this, we insert the remaining vertices from σ one by one into the DAG. To insert a variable X_i, we grow the k-tree by (greedily) picking as parent set of X_i, a clique $C \subseteq X \setminus \{X_i\}$ of size at most k such that $f(X_i, C)$ is maximized among all such cliques. Once all variables have been inserted, we check the score of the final BN and compare it with the current best BN, replacing if the score of the newly constructed BN is higher. This is an anytime algorithm. We then keep repeating the same procedure, from a newly sampled random ordering each time, until we run out of time, i.e., reach the global timeout, upon which we return the best recorded BN encountered so far.

k-greedy was proposed by Scanagatta et al. [Sca+16] and was the state-of-the-art algorithm for the heuristic BNSL problem at the time of its introduction. k-greedy scales quite well to huge BNs with as many as 10,000 random variables. This performance is due to the fact that the crux of the algorithm is an extremely quick greedy procedure. But, relying on this alone would severely affect the quality of the resulting BNs. However, this is addressed by the random sampling of the linear orderings. The efficiency of the greedy procedure allows the algorithm to comb through a huge number of random orderings. This dual combination improves the likelihood that the algorithm stumbles upon a 'lucky' ordering which yields an almost-optimum BN. An informal analogy would be that of putting a million monkeys to work, each handed a different random ordering and tasked with the simple job of constructing a BN from the random ordering.

2.3.3 k-MAX Algorithm

A couple of years later, Scanagatta et al. [Sca+18] proposed a newer, slightly smarter algorithm, which dethroned k-greedy as the state-of-the-art. The new algorithm, called k-MAX, uses less randomization and more informed guessing to decide the order in which to insert the vertices. It still uses the same tactic of picking the clique which maximizes $f(X_i, C)$ among all cliques. But, instead of randomly sampling a linear

[1] using the fact that removing a vertex cannot increase the treewidth of a graph

ordering at the outset, it only relies on randomization to pick a handful of variables. First, it builds a set of $k + 1$ variables by randomly picking some variables and including all the other variables that could potentially be parents of these variables into this set as well. After this, it uses an estimate $m(X_i)$ to guess the potential increase in score if variable X_i is inserted next. Then it picks the remaining variables one by one, in decreasing order of $m(X_i)$, i.e., most promising variable first. In this way, it uses a cleverer estimate to decide the variable insertion order instead of working through a random yet pre-determined ordering that is completely agnostic of the input instance. The rest of the algorithm is identical to k-greedy, we repeat the previous step, keeping track of the best scoring BN encountered so far, until we reach the global time limit. k-MAX is also an anytime algorithm.

We conclude the description of the potential tools at our disposal to attack the BNSL problem. In the next section, we discuss how we solved the BNSL problem and its variants.

2.4 Solving the BNSL Problem

Our approach to solve the BNSL problem builds on top the SLIM framework. In order to obtain an algorithm from this framework, we need two ingredients: the global solver and the local solver, and then combine them together in a clever way, which is where the real difficulty lies. We use state-of-the-art k-greedy and k-MAX algorithms as our global solvers. We also have access to a MaxSAT encoding for the BNSL problem which is similar to the encoding by Berg, Järvisalo, and Malone [BJM14]. However, this encoding needs to be fortified before it can function as a local solver. Recall that the solution to the BNSL problem is a DAG. In order to fortify the encoding, we need to understand and capture the interaction of the local solution DAG with the remaining of the global solution DAG, to ensure replacement consistency.

This interaction comes in two forms. Firstly, once we replace a local solution with an improved solution, the final DAG must still remain a Directed *Acyclic* Graph, i.e., the local subproblem must capture acyclicity. The second requirement is that the treewidth of the DAG after replacement still respects the required treewidth bound. To satisfy this requirement, it is sufficient to ensure that we can combine the tree decomposition of the local solution with the tree decomposition of the remaining DAG. Thus, the local subproblem must also capture the compatibility criterion between the local and global tree decompositions. We thus fortify the original MaxSAT encoding as follows:

- incorporate information about possible ways to form cycles thereby violating the acyclicity requirement,
- pass on information about the bags in the overlapping zone between the local solution and the global solution, so that the local solver can ensure compatibility with them.

Once we have a global and local solver, we can plug them into the SLIM framework to obtain an algorithm to solve BNSL. We call this algorithm BN-SLIM. We then perform experiments and show that BN-SLIM outperforms k-greedy and k-MAX and can compute higher scoring BNs in the same amount of time.

After that, we looked at two variants of the base BNSL problem. In the first one instead of limiting the treewidth, i.e., the maximum number of variables in a bag, we limit the maximum product of domain sizes of the variables contained in a bag. This measure is more closely related to the inference time than treewidth. In the second variant, we consider Bounded Treewidth BNs which also respect additional externally provided constraints such as ancestries or adjacencies.

We address these variants of the BNSL problem in a similar manner. Since k-greedy and k-MAX are relatively simple algorithms, we upgrade them such that they can handle the BNSL variant. After that, we fortify the MaxSAT encoding from Berg, Järvisalo, and Malone [BJM14] tailoring the encodings to the conditions of required variant. The expressive power of MaxSAT comes in handy for this purpose. Once we have a global solver and a local solver for the variants of BNSL, we can once again plug them into the framework to obtain algorithms to solve the variants. For these variants as well, we show empirically that the SLIM algorithm yields better results.

CHAPTER 3

Technical Preliminaries

The fact that all Mathematics is Symbolic Logic is one of the greatest discoveries of our age.

— Bertrand Russell, "The Principles of Mathematics"

In this chapter, we go through the basic concepts and definitions from Chapters 1 and 2 again but with a rigorous demeanor and in a more formal tone. Consequently, this chapter is chock-full of symbols, notation, and terminology and is meant to serve as a reference guide for the more technical sections of the thesis. We end this chapter with the formal definition of the base version of the core problem of this thesis.

3.1 Theoretical Background

3.1.1 Graph Theory

Although, we cover most of the required notation and concepts in this chapter, for a more comprehensive introduction to Graph Theory, we refer the reader to the book by Diestel [Die00]. Additionally, for the theory on directed graphs, we refer to the initial chapters of the book by Harary, Norman, and Cartwright [HNC65]

A graph $G = (V, E)$ is a tuple where V is the set of *vertices* or *nodes* and E is the set of *edges*. We also use the notation $V(G)$ and $E(G)$ to refer to the set of vertices and set of edge of a graph G respectively. The edges may either be *undirected* (represented by an unordered tuple or a set) or *directed* (represented by an ordered tuple). We only deal with graphs in which either all the edges are undirected (called *undirected graphs*) or all the edges are directed (called *directed graphs*). We use the shorthand

vertex, node
edge
$V(G)$, $E(G)$
undirected edge
directed edge
undirected graph
directed graph

edge, arc
graph over a set
incident
$G[S]$

edge for undirected edges and *arc* for directed edges. An edge between nodes u and v is mathematically represented as an unordered pair $\{u, v\}$ in an undirected graph and as an ordered pair (u, v) in a directed graph. A graph *over* a set S is a graph G such that $V(G) = S$. Edge $\{u, v\}$ or arc (u, v) is *incident* on both u and v. A subgraph of graph G is a graph obtained by deleting some vertices and edges from G, where deleting a vertex implies the deletion of all the edges incident on that vertex. $G[S]$ is the subgraph of G obtained by deleting vertices $V(G) \setminus S$.

neighbor
adjacent
$N(\cdot)$, neighborhood
degree
successor
predecessor
in-degree
out-degree
path
path length
ancestor
descendant
simple path
cycle
acyclic graph
DAG
connected graph

In an undirected graph, a vertex v is said to be a *neighbor* of (or *adjacent* to) another vertex u if there exists an edge $\{u, v\} \in E$. We denote by $N(u) = \{\, v \in V : \{u, v\} \in E \,\}$ the *neighborhood* of u, and $|N(u)|$ is the *degree* of u. Similarly, in a directed graph, for every arc (u, v), v is a *successor* of u and u is a *predecessor* of v. The number of successors and predecessors of a vertex are called *in-degree* and *out-degree* respectively. In an undirected graph G, a *path* is a sequence $v_0, \ldots, v_\ell$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $0 \leq i < \ell$ and ℓ is the *length of the path*. In a directed graph D, a path is similar except that $(v_i, v_{i+1}) \in E(D)$ instead. Analogous to the concept of successors and predecessors, if there exists a path from u to v, u is an *ancestor* of v and v is a *descendant* of u. A *simple path* is one in which all v_i are distinct, and in this thesis, we only deal with simple paths. A *cycle* is a path with $\ell \geq 2$ (or $\ell \geq 3$ for undirected graphs), $v_0 = v_\ell$ and $v_i \neq v_j$ for all other combinations of i and j. An *acyclic* graph is a graph devoid of cycles. Combining these concepts, we arrive at the oft-encountered and aptly named *Directed Acyclic Graphs* (DAGs). Lastly, a graph is *connected* if there exists a path from any vertex of the graph to any other vertex of the graph.

Some common graph structures that we refer to are listed below:

tree
- *trees* are connected acyclic graphs
- cycles are simple paths with coincident endpoints (mentioned above)

clique
- *cliques* are fully connected graphs G such that $E(G) = \{\, \{u, v\} \subseteq V(G) : u \neq v \,\}$

star
- *stars* are graphs $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$ and $E = \{\, \{v_1, v_j\} : 1 < j \leq n \,\}$

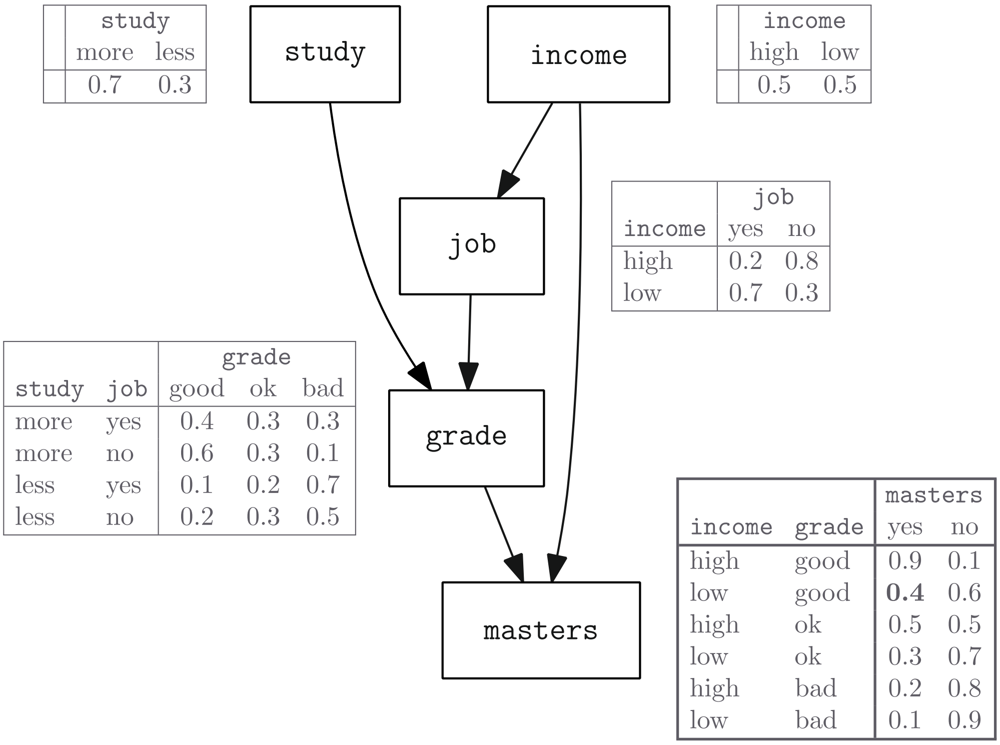
3.1.2 Bayesian Network Structure Learning

RV
random variable
domain, $\mathrm{dom}(\cdot)$
categorical RV
BN
CPD
structure
parameters

A *random variable* (RV) X is a variable that takes values from a pre-defined *domain* $\mathrm{dom}(X)$ with some probability. We only work with *categorical RVs*, which are RVs having a countably finite domain. A *Bayesian Network* (BN) consists of a DAG over n categorical RVs, together with local parameters, i.e., *Conditional Probability Distribution* (CPD) tables, one for each node. The DAG by itself is called the *structure* of the BN and all the CPD tables collectively are called *parameters* of the BN. Bayesian Networks were introduced by Pearl in 1985 and later formalized by Pearl [Pea88] and Neapolitan and Jiang [NJ07]. Please see Figure 3.1 for an example BN showing both the structure and the parameters.

structure learning

The problem of learning a BN is divided into two tasks, *structure learning*, i.e, learning



	study	
	more	less
	0.7	0.3

	income	
	high	low
	0.5	0.5

	job	
income	yes	no
high	0.2	0.8
low	0.7	0.3

		grade		
study	job	good	ok	bad
more	yes	0.4	0.3	0.3
more	no	0.6	0.3	0.1
less	yes	0.1	0.2	0.7
less	no	0.2	0.3	0.5

		masters	
income	grade	yes	no
high	good	0.9	0.1
low	good	**0.4**	0.6
high	ok	0.5	0.5
low	ok	0.3	0.7
high	bad	0.2	0.8
low	bad	0.1	0.9

Figure 3.1: Example of a Bayesian Network (BN)

just the DAG and *parameter learning*, i.e., learning the parameters for a given DAG. We are interested in the *Bayesian Network Structure Learning* (BNSL) problem of learning the structure of a BN from a data set of N instances $D_1, \dots, D_N$ over a set of n categorical RVs $X_1, \dots, X_n$, with no missing entries. An *instance* refers to an observation or an instantiation of the complete set of RVs. For example, in Figure 3.1, an instance could be

parameter learning
BNSL
instance

$$D_1 = \{X_s = \text{more}, X_i = \text{low}, X_j = \text{no}, X_g = \text{good}, X_m = \text{yes}\}$$

which says

$$\texttt{study} = \text{more}, \texttt{income} = \text{low}, \texttt{job} = \text{no}, \texttt{grade} = \text{good}, \texttt{masters} = \text{yes}.$$

Since only the structure by itself without the parameters does not completely specify a BN, we make use of a scoring function as a proxy to determine how well a DAG D fits the data. In other words, the score gives an estimate of how close the computed BN is to the BN underlying the input instances. Thus, the goal is to find a BN which maximizes the score. This problem was shown to be NP-hard by Chickering [Chi96] even when the maximum in-degree for each variable is bounded [CHM04].

We assume that the score is *decomposable*, i.e., the score for the entire BN is the sum of scores for the individual RVs. Hence, we can assume that the score is given in terms

decomposable score function

score function *$f(\cdot,\cdot)$* *parent set* *parent*

of a *score function* f that assigns each node $v \in V$ and each subset $P \subseteq V \setminus \{v\}$ a real number $f(v, P)$, the score of P for v. The score of the entire DAG $D = (V, E)$ is then $f(D) := \sum_{v \in V} f(v, P_D(v))$ where $P_D(v) = \{\, u \in V : (u, v) \in E \,\}$ denotes the *parent set* of v in D and each $u \in P_D(V)$ is a *parent* of v (here we overload the score function f). This setting accommodates several popular scores like AIC, BDeu, and BIC [Aka74; HGC95; Sch78]. Because there is a one to one correspondence between the RVs and the nodes or vertices in the DAG, we refer to them interchangeably. If P and P' are two potential parent sets of an RV v such that $P \subsetneq P'$ and $f(v, P') \leq f(v, P)$, then we can safely disregard the potential parent set P' of v. Consequently, we can disregard all nonempty potential parent sets of v with a score of at most $f(v, \emptyset)$. Such a restricted score function is a *score function cache*.

score function cache

inference *reasoning*

Once we have learned a network, we can use it to predict the probabilities of some sets of variables give observations of some other variables. This task is called *inference* or *reasoning*. A series of works showed that even approximate inference on BNs, is #P-complete in general [Rot96; DL93; Coo90] and requires supposedly exponential time. However, if we impose some restrictions on the structure of the BN, we can perform inference in polynomial time. To understand this restriction, we first need to take a look at the concept of tree decompositions.

3.1.3 Tree Decompositions

tree decomposition

A *tree decomposition* $\mathcal{T}$ of an undirected graph G is a pair (T, χ), where T is a tree and χ is a function that assigns to each tree node t, a set $\chi(t)$ of vertices of G such that the following conditions hold:

T1 For every edge e of G there is a tree node t such that $e \subseteq \chi(t)$.

T2 For every vertex v of G, the set of tree nodes t with $v \in \chi(t)$ induces a subtree of T.

bag *width* *treewidth*

The sets $\chi(t)$ are called *bags* of the decomposition $\mathcal{T}$, and $\chi(t)$ is the bag associated with the tree node t. The *width* of a tree decomposition (T, χ) is the size of a largest bag minus 1. The *treewidth* of G, denoted by $\mathrm{tw}(G)$, is the minimum width over all tree decompositions of G. Computing the treewidth of a graph is known to be NP-hard [ACP87].

elimination ordering

An alternate way of characterizing tree decompositions is with elimination orderings. An *elimination ordering* σ for a graph G is a total ordering of all the vertices $V(G)$. Given an elimination ordering σ, we can construct a corresponding tree decomposition whose width can be computed as follows. We iterate through the vertices as per the ordering σ and for each vertex u perform the following steps,

fill-in edge

1. for $v, w \in V(G)$, insert a *fill-in edge* $\{v, w\}$ if both v and w are adjacent to u (either by normal edges or previously filled-in edges),

2. record the degree d_u of u and then delete vertex u.

After going through all the vertices, $\max_{u \in V(G)} d_u$ yields the width of the corresponding decomposition. Lastly, it can be shown that for every tree decomposition there exists an elimination ordering which yields a decomposition of the same width. Thus, we often work with elimination orderings in lieu of tree decompositions.

Another relevant concept is that of k-trees, which we encounter when dealing with heuristic algorithms tasked with constructing bounded treewidth graphs. A *k-tree* is an edge-maximal graph of treewidth k. k-trees are constructively defined as follows. A clique on $k+1$ vertices is a k-tree. Let C be a k-clique in a k-tree, inserting a new vertex adjacent to all the other vertices of C yields a new k-tree (called *growing a k-tree*). Naturally, any subgraph of a k-tree also has treewidth at most k.

k-tree
growing k-trees

Finally, returning to BNs, the treewidth of a BN is defined via its moral graph. The *moralized* graph of BN $\mathcal{B}$ (written as $M(\mathcal{B})$) is obtained by converting all directed arcs (u, v) into undirected edges $\{u, v\}$ and then inserting so-called *moral edges* $\{v, w\}$ between any two variables v, w if both $v, w \in P_D(u)$ for some u. Kwisthout, Bodlaender, and Gaag [KBG10] showed that bounding the treewidth of BNs is a necessary condition to ensure polynomial time inference. Given a BN on n variables with treewidth w, we can perform exact inference in time $\mathcal{O}(2^w \text{poly}(n))$. Thus, we arrive at the problem of interest:

moralization
$M(\cdot)$
moral edge

Bounded Treewidth Bayesian Network Structure Learning

Input: A set X of random variables, a score function cache f, an integer k.

Question: Find a DAG D over X with treewidth at most k such that $f(D)$ is maximized.

Solving Bounded Treewidth BNSL exactly was proven to be NP-hard by Korhonen and Parviainen [KP13]. Hence, we divert our attention to heuristic algorithms that provide approximate solutions but are still practical for large inputs with thousands of RVs. Some of these design choices are based on the typical workflow when working with BNs. One usually learns a BN only once at the beginning and then uses that learned BN to make several predictions or inferences. Hence, we set the accuracy of the inferences as the primary priority and the time required to learn the BN as only secondary.

3.2 Algorithmic Background

3.2.1 Maximum Satisfiability

The *Propositional Satisfiability* (SAT) problem forms a cornerstone of Computer Science, proving to be a reliable anchor for the theory of Computational Hardness. However, the hardness of the satisfiability problem itself was not sufficient to discourage research in the direction of finding practical solutions. In the SAT problem, we are given a *propositional formula* involving a set X of n Boolean variables and *conjunction*, *disjunction* and *negation* operations (denoted $\wedge, \vee$ and $\neg$ respectively). For convenience, we also define

SAT
propositional formula
conjunction
disjunction
negation
$\wedge, \vee, \neg, \rightarrow$

assignment *satisfiable* *unsatisfiable*

$p \longrightarrow q$ to be equivalent to $(\neg p) \vee q$. Given an *assignment* of the variables, i.e., a mapping $X \to \{\texttt{TRUE}, \texttt{FALSE}\}$, one can evaluate the formula to obtain the truth value of the formula for that assignment as either a TRUE or a FALSE. The goal is then to find an assignment of the variables such that the given formula evaluates to TRUE. A formula is *satisfiable* if there exists such an assignment and *unsatisfiable* otherwise.

CNF *clause* *literal* *satisfied* *encoding*

A propositional formula is in *Conjunctive Normal Form* (CNF) if it consists of a conjunction over disjunctions. Each disjunction is called a *clause* and consists of variables or negated variables (called *literals*). We represent a clause as a set of literals. A clause C is *satisfied* by an assignment τ, if $\tau(x) = \texttt{TRUE}$ for some $x \in C$ or if $\tau(x) = \texttt{FALSE}$ for some $\neg x \in C$. The CNF-SAT problem, similar to SAT, asks to find an assignment of variables that satisfies the given CNF formula, i.e., satisfies all the clauses. Even though CNF formulas seem restrictive, they are expressive enough to capture conditions such as bounded treewidth, acyclicity, bounded state space size and more. An *encoding* of an instance of a decision problem, is a CNF formula such that, the instance of the decision problem is YES-instance if and only if the formula is satisfiable.

SAT solver *MaxSAT* *hard clause* *clause weight* *soft clause* *MaxSAT solver* *MaxSAT solver timeout* *anytime algorithm*

Despite the CNF-SAT problem being exactly as hard as the SAT problem, there has been considerable research in trying to develop software to solve it. This lead to the development of *SAT solvers* which can solve CNF formula involving millions of variables and tens of millions of clauses in under 2 hours. These solvers can determine whether the given formula is satisfiable or unsatisfiable, and return a satisfying assignment as proof of satisfiability. In our case, we make use of a generalization of the CNF-SAT problem called *Maximum Satisfiability* (MaxSAT), where the input consists of *hard clauses* and *weighted soft clauses*. The solution must satisfy all the hard clauses and maximize the weight of the satisfied soft clauses. *MaxSAT solvers*, similar to SAT solvers, are tools that can solve MaxSAT for formulas involving millions of variables and tens of millions of clauses. Many MaxSAT solvers also support a *timeout* functionality through which it is possible to interrupt the solver at any point in time and the solver returns the best (possibly suboptimal) solution found so far. Such an algorithm is called an *anytime algorithm*, i.e., an algorithm which can be interrupted at any point of time, and the algorithm spits out the current best solution.

3.2.2 SAT-based Local Improvement Method

SLIM

The *SAT-based Local Improvement Method* (SLIM) framework was introduced by Lodha, Ordyniak, and Szeider [LOS16] and has since been successfully applied to several diverse problems such as

- computing width measures for graphs like
  - treewidth [FLS17; PS21c; PS22],
  - branchwidth [LOS16; LOS17; LOS19], and
  - treedepth [PS20];

- inducing low-complexity decision trees [SS21];
- graph coloring [Sch22; SS22]; and
- circuit minimization [RSS23].

For any *intractable problem* (NP-hard or harder), the framework allows us to develop a scalable heuristic algorithm using an existing heuristic algorithm and an encoding for the problem. The framework leverages the *scalability* of the existing heuristic algorithm (i.e., the algorithm works reasonably well even for large inputs), as well as the optimality of the encoding to keep finding improvements even when other heuristics are unable to do so. We often use the shorthand *heuristic* to refer to heuristic algorithms, i.e., algorithms that do not guarantee their solution's optimality. This is not to be confused with the notion of heuristic as a static function that maps its inputs to a numeric range.

intractable problem

scalability

heuristic

The key idea behind the framework is to start from an initial heuristic solution S (called the *global solution*) provided by the *global solver* (usually the existing heuristic). We then repeatedly extract from S, improve and patch back into S small local parts using the *local solver* (usually a MaxSAT solver applied to the encoding). For practicality, we enforce that each extracted local part has size at most some *budget* β and also that the local solver respects a *local timeout*.

global solution

global solver

local solver

budget, β

local timeout

It is important that the local solver takes into account the interaction of the local solution with the rest of the solution. The local solver must ensure that replacing the old local solution in S with the newly improved local solution does not invalidate S. This property of maintaining the validity of the global solution is called *replacement consistency*. Any existing encodings of the problem are highly unlikely to respect replacement consistency and hence this entails the *fortification* of the existing encoding by incorporating additional constraints which capture said interaction of the local solution with the rest of the global solution. For example, suppose acyclicity is a required condition for the solution. We know that, at the beginning, the initial heuristic solution is acyclic, and during one iteration of the SLIM algorithm, the solution to the local subproblem computed by the local solver is also acyclic. However, this does not guarantee that after replacing the local solution with the new acyclic local solution, the global solution still remains acyclic. So, we need to supply some additional constraints to the local solver (through the fortifications of the encoding) such that any new local solution which might have caused a cycle upon replacement, is already ruled out by the local solver. The pseudocode of a generic SLIM-based algorithm is shown in Algorithm 3.1. We refer to Figure 2.2 for a demonstration of applying SLIM to a toy example.

replacement consistency

fortification

The SLIM framework by itself is quite modular and agnostic to the choice of the local and global solvers. However, it is quite typical for there to already exist a heuristic algorithm which we can use as the global solver and an encoding which we can fortify and then use as the local solver. The main challenge in developing a SLIM-based algorithm is thus coming up with the right fortifications. It sometimes involves adapting an existing encoding to work for a generalized version of the original problem. Another minor

Algorithm 3.1: Pseudocode of a generic SLIM-based algorithm for problem P

```
Assume : We have access to a global solver $\mathcal{G}$ for P and a local solver $\mathcal{L}$ for the
         fortified version of P
Input  : Instance $I$ of problem P, budget $\beta$, global timeout $T$, local timeout $\tau$
Output : Solution $S$ to instance $I$

begin
    $S \longleftarrow \mathcal{G}(I)$        // initialize $S$ with heuristic solution
    repeat
        Extract local solution $J$ of size at most $\beta$
        Construct fortified subproblem $J'$ from $J$
        $S' \longleftarrow \mathcal{L}(J')$ using time at most $\tau$
        Update $S$ by replacing $J$ with $S'$
    until time $T$ not exceeded

    return $S$
end
```

global timeout

technicality arising out of practical considerations is the *global timeout* which is the total time limit we set for the learning phase. It is preferable if the algorithms we work with are anytime algorithms, this allows for fair and straightforward comparison of their performances. The SLIM-based algorithm by itself is also an anytime algorithm, as it can keep on running the inner loop until it gets interrupted, which is when it returns S as the final (possibly suboptimal) solution.

treedepth

We would like to highlight one specific application of SLIM, computing treedepth [PS20], and provide a quick overview.[1] The treedepth of a graph is defined by means of the treedepth decomposition, which is a tree T with vertex set same as the original graph and a designated root. There is one additional condition: for any two vertices sharing an edge in the original graph, one must be an ancestor the other in the decomposition tree T. The depth of a decomposition is the length of the longest root to leaf path and the *treedepth* of a graph is the minimum possible depth over all possible treedepth decompositions. Please see Figure 3.2 for a classic example of a treedepth decomposition of a path graph. For convenience, we have drawn the implied ancestor-descendant relations as dashed lines.

In order to tackle this problem using SLIM, we start off with a naive DFS heuristic for the initial treedepth decomposition and then try to improve the depth of small local subtrees using a fortified version of the encoding first proposed by Ganian et al. [Gan+19]. Once a local subtree has been processed (either improved or unaltered), we contract it to a single node and move upwards in the global solution, making our way to the root of

[1] Due to being topically disjoint from the core problem of this thesis, this paper is added as an appendix. This problem was among the ones investigated by the author during the course of their doctoral studies and resulted in a publication [PS20].

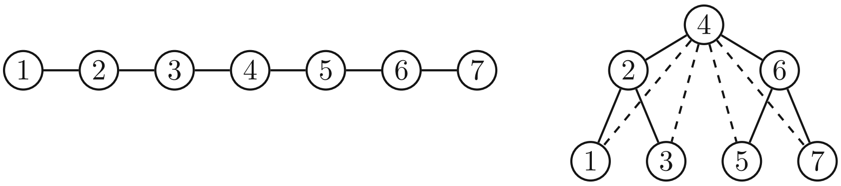


Figure 3.2: Path graph P_7 on the left and its treedepth decomposition (of depth 3) on the right. The dashed lines show the ancestry relations.

the global solution. Upon reaching the root, we *reinflate* all the contractions sequentially to complete one pass and obtain a new global solution with depth at most as much as the previous global solution. We perform as many passes as time permits and in the end return the current global solution as the final solution. In order to ensure replacement consistency, we formulate a generalization of the treedepth problem and also develop an encoding for the same. This generalization allows for depth labels on vertices and additional ancestry constraints. Surprisingly, the SLIM approach was able to find a treedepth decomposition of same width significantly faster compared to encoding the entire problem as a single monolithic SAT encoding.

3.2.3 Heuristics for BNSL

As we saw in the previous section, we need two main ingredients to develop a SLIM-based algorithm. One of which is a heuristic algorithm used to provide the initial solution. At the start of the PhD term, the k-MAX algorithm by Scanagatta et al. [Sca+18] was the state-of-the-art heuristic algorithm for solving the BNSL problem for huge networks with thousands of RVs. Additionally, it was also an anytime algorithm, making it the perfect candidate heuristic for a SLIM-based approach. The k-MAX algorithm was a successor to the formerly state-of-the-art k-greedy algorithm also by Scanagatta et al. [Sca+16]. Since these two algorithms are quite similar, we start with a description of the k-greedy algorithm for the sake of simplicity.

The k-greedy algorithm for Bounded Treewidth BNSL works by repeatedly sampling an ordering over the RVs and then constructing a DAG based on this ordering. The ordering acts as a topological ordering of the resulting DAG and also as an elimination ordering corresponding to a tree decomposition respecting the width bound. Given a randomly sampled ordering σ, the algorithm iteratively inserts RVs one by one, according to σ, greedily fixing the parent sets of each RV as they are inserted (hence the name k-greedy). To ensure that the DAG has bounded treewidth, the algorithm maintains that the partial graph constructed so far is always a subgraph of a k-tree. We refer to Algorithm 3.2 for a pseudocode of the k-greedy algorithm. We denote the partially constructed DAG consisting of the first i variables from σ as D_i. A parent set P is *feasible* if the variables in P occur as a clique or a subgraph of a clique in the moral graph of D_i.

feasible parent set

The k-MAX algorithm was proposed shortly after and outperformed k-greedy to become the new state-of-the-art algorithm for Bounded Treewidth BNSL. It functions similarly,

Algorithm 3.2: Pseudocode of k-greedy [Sca+16]

```
   Input  : Set of $n$ Random Variables $X$, score function cache $f$
   Output : DAG $D$ such that $f(D)$ is maximized
 1 begin
 2     $D' \leftarrow$ empty DAG        // stores the best DAG found so far
 3     repeat
 4         Sample a random order $\sigma$ over $X$
 5         $i \leftarrow k$
 6         Find best scoring DAG $D_{k+1}$ over the first $k+1$ variables from $X$
 7         while $i \leq n$ do
 8             $X_i \leftarrow \sigma[i]$        // $i$th variable as per $\sigma$
 9             Let $P$ be a feasible parent set such that $f(P, X_i)$ is maximized
10             Insert $X_i$ as child of $P$ into $D_i$ to obtain $D_{i+1}$
11             $i \leftarrow i + 1$
12         end
13         if $f(D_n) > f(D')$ then
14             $D' \leftarrow D_n$
15         end
16     until global timeout reached
17     return $D'$
18 end
```

except that it picks the vertex insertion ordering smartly on the fly based on how promising a vertex is. The algorithm estimates the promise of a vertex based on potentially how much that vertex can contribute to the total score. The estimate is based on a heuristic $m(\cdot)$ score $m(X_i)$ which compares in each iteration, for each remaining variable X_i, the best possible score from a feasible parent set against the best possible score from the parent set cache (with the lowest possible cached score as the baseline). The variables for which this gap is huge are deferred to be inserted later in the hopes that more parent sets become feasible later, consequently reducing the gap. To obtain the pseudocode of k-MAX from Algorithm 3.2, we

- delete Line 4,
- change Line 6 to build the initial DAG by randomly picking some vertices and filling in the rest with their respective candidate parents, and,
- replace Line 8 with $X_i \leftarrow \text{argmax}_{x \in X}\, m(x)$.

One of the key reasons for the success of these heuristics is the sheer amount of potential DAGs D_n that they process. For example, for an input with 500 variables, these algorithms go through tens of thousands of candidate DAGs, returning the highest scoring DAG among them in the end. At the beginning of the algorithm, new DAGs are constantly replacing previous lower-scoring DAGs, at the rate of thousands of DAGs every minute. However, as time goes on, this rate slows down as the newer candidate DAGs need to work harder to replace the current best. We use this characteristic to our advantage by applying SLIM to a DAG found after the rate has slowed down, and SLIM is able to squeeze out improvements (due to the MaxSAT solver) even from supposedly good DAGs.

3.3 Overview of Results

During the course of the doctoral studies, we primarily focused on applying the SLIM framework to different problems, starting with treedepth (discussed in the previous section) before moving onto BNSL. In this section, we present the brief overviews of the contributions of the works compiled in this thesis. The reader may use the Expert Index at the end of the thesis for quickly tracking down definitions.

Bounded Treewidth BNSL (detailed in Chapter 4)

Bounded Treewidth Bayesian Network Structure Learning

Input: A set X of random variables, a score function cache f, an integer k.

Question: Find a DAG D over X with treewidth at most k such that $f(D)$ is maximized.

In this work we look at the Bounded Treewidth BNSL problem shown above and develop the BN-SLIM algorithm for it, which also acts as the base version of the algorithm for the other problems we considered. The algorithm starts off with an initial solution provided by k-MAX [Sca+18]. This solution consists of a BN along with a corresponding tree decomposition. We then randomly pick a bag in the tree decomposition and perform a BFS from this starting bag, slowly expanding our selection until the budget is exceeded. At this point, the union of all the bags, gives us the set of RVs that form our local part S.

We now need to use our local solver to find an improved local solution for the selected local part. To this end, we first construct an encoding for the BNSL problem by building on top of the SAT encoding for bounding the treewidth of graphs [SV09] and apply it to the moralized graph. We then add decision variables par_v^P that select for each RV v, a parent set P from the score function cache. We then lift the SAT encoding to a MaxSAT encoding by adding unit soft clauses par_v^P, weighted by the score $f(v, P)$. Finally, we express the edges of the moralized graph in terms of these decision variables. We now have a MaxSAT encoding for the Bounded Treewidth BNSL problem, however this encoding is completely agnostic of the global solution and is not guaranteed to respect replacement consistency. This was the main challenge that we overcame to develop the BN-SLIM algorithm.

To ensure replacement consistency, we first focus on acyclicity. We only need to worry about cycles that pass through both the local solution and the rest of the global solution, to ensure that the global solution remains acyclic upon replacement. To disallow such cycles, we add an arc (w, u) to the local subproblem wherever there exists a path $u, v_1, \ldots, v_\ell, w$, where $\ell \geq 1$, $u, v \in S$, and $v_i \notin S$ for all $1 \leq i \leq \ell$. We then consider the validity of the tree decomposition. For each bag B in the local part which intersects some bag B' not in the local part, we implant a so-called marker clique [FLS17] on the set of variables $B \cap B'$ in the moralized graph of the local part. This ensures that there always exists a bag containing $B \cap B'$ in the tree decomposition of the local part. This bag can then be joined to B' upon replacement, thus maintaining a valid tree decomposition.

We implemented this algorithm and empirically showed that BN-SLIM performs better than running k-MAX alone for the same duration.

Published as “Turbocharging Treewidth-Bounded Bayesian Network Structure Learning”. Peruvemba Ramaswamy and Szeider. In: Proceedings of the AAAI Conference on Artificial Intelligence, (May 2021) [PS21c]

Bounded State Space BNSL (detailed in Chapter 5)

To ensure polynomial time inference, bounding the treewidth of the BN is a very popular technique in the field of BN learning. The motivating question for our work was “is treewidth the best estimate of inference time or does there exist a better measure?” Inference time depends on the maximum over all bags of the number of rows in the CPD table, a measure called the *state space size* which has been considered by various authors [Kas+11; Kjæ92; MJ96; OD08b; OD08a]. The state space size corresponding to a bag B is $\prod_{X \in B} \mathrm{ds}(X)$, where $\mathrm{ds}(X)$ denotes the domain size or the number of states that an RV X can take. When dealing with binary RVs, the state space size is simply $\mathcal{O}(2^{|B|}) = \mathcal{O}(2^k)$ where k is the treewidth. However, when using treewidth as a proxy for the inference time, the accuracy deteriorates as the variance in the domain sizes of the variables grows. Thus, when working with non-binary variables, it is more apt to bound the state space size of the learned BNs rather than bounding the treewidth. We corroborate this fact with practical evidence showing that state space size has a significantly stronger correlation with the inference time.

Bounded State Space Bayesian Network Structure Learning

Input: A set X of random variables, a score function cache f, an integer k.

Question: Find a DAG D over X with *maximum state space size* at most k such that $f(D)$ is maximized.

We formulate the Bounded State Space BNSL problem as shown above and develop a SLIM approach to solve it. We build on top of BN-SLIM and replace the unary cardinality counter which bounds the treewidth in the encoding with a novel BDD-based counter which bounds the state space size. This new BDD counter keeps track of the sum of logarithms of the domain size to keep the product of domain sizes under check. To supplement that, we also modified k-MAX [Sca+16] such that its search is restricted to bounded state space structures. Previous work focused on computing a decomposition that minimizes the state space size for a given structure rather than learning a bounded state space structure from outset. Combining the modified version of k-MAX and the improved encoding using the SLIM approach, we propose the first Bounded State Space BNSL algorithm and provide a scalable implementation. We perform experimental evaluations and highlight the benefits of bounding state space size in place of treewidth.

Published as "Learning Fast-Inference Bayesian Networks". Peruvemba Ramaswamy and Szeider. In: Advances in Neural Information Processing Systems, (2021) [PS21b]

BNSL with Expert Constraints (detailed in Chapter 6)

Lastly, we move our focus to the increasingly popular area of Causality. BNs are inherently correlational structures, an arc from variable u to variable v merely indicates that the two variables are correlated and does not mean that u has a causal effect on v. This justifies the research direction of learning BNs in the presence of additional causal constraints. Since such constraints are typically sourced from an expert, we call them *expert constraints*. Such constraints can either be in the form of arc constraints which indicate direct causation or ancestry constraints which indicate indirect causation. Direct causation between two variables u and v can either be disallowed $u \nrightarrow v$, forced $u \rightarrow v$, or, forced ambivalently $u \leftrightarrow v$. Indirect causation between two variables u and v can either be forced $u \rightsquigarrow v$ (i.e., there must exist a path from u to v) or disallowed $u \not\rightsquigarrow v$ (i.e., there cannot exist a path from u to v).

In our investigation of learning BNs with expert constraints, we further restrict ourselves to learning bounded treewidth BNs in a scalable manner. Several versions of this problem have been studied before for different subsets of qualifiers [Che+16; KKN01; LB18; Cor+13]. However, we initiated the first line of research in the direction of learning bounded treewidth BNs which respect expert constraints while also scaling to thousands of RVs. The formal definition of this problem is shown below.

Constrained Bayesian Network Structure Learning

Input: A set X of random variables, a score function cache f, an integer k, and a set $\mathcal{C}$ of constraints of the form $u \bowtie v$, where $\bowtie \in \{\rightarrow, \nrightarrow, \leftrightarrow, \rightsquigarrow, \not\rightsquigarrow\}$

Question: Find a DAG D over X with treewidth at most k such that $f(D)$ is maximized *and D satisfies all constraints in $\mathcal{C}$*.

We solved this problem by first upgrading the k-greedy heuristic algorithm so that it respects the supplied expert constraints. We also extended the Bounded Treewidth BNSL encoding further, equipping it with the ability to understand and enforce the different types of expert constraints. To achieve this, we first added auxiliary variables and clauses that keep track of presence and absence of paths between different variables. Then, we enforced the presence or absence of arc and ancestry constraints by using the auxiliary variables. Combining this encoding with the upgraded k-greedy algorithm, we obtain the SLIM algorithm for the Constrained BNSL problem. We then implemented this approach and tested it to find that it was able to learn bounded treewidth BNs with thousands of RVs while still satisfying 80-90% of constraints and performing better than running the upgraded k-greedy algorithm by itself.

🗎 Published as "Learning large Bayesian networks with expert constraints". Peruvemba Ramaswamy and Szeider. In: Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, (Aug. 2022) [PS22]

CHAPTER 4

Bounded Treewidth BNSL

All of the chaos seems to be happening at the boundaries between the regions.

— Grant Sanderson, "From Newton's method to Newton's fractal"

Published as "Turbocharging Treewidth-Bounded Bayesian Network Structure Learning". Peruvemba Ramaswamy and Szeider. In: Proceedings of the AAAI Conference on Artificial Intelligence, (May 2021) [PS21c]

In this chapter, we discuss the basic version of the BN-SLIM algorithm aimed to solve the Bounded Treewidth BNSL problem. We use the SLIM framework to develop this algorithm. We use k-MAX as the global solver to produce the initial heuristic solution. For the local solver, we build on top of the SAT encoding for bounded treewidth graphs by Samer and Veith [SV09], and apply it to the moralized graph. The moralized graph itself is expressed in terms of the arcs in the BN. This way, we can keep the treewidth of the BN in check. The main variables in the encoding represent which parent set is assigned to each RV. From this, we derive which arcs must be present in the BN. We have additional variables that represent an ordering of the RVs in the DAG, from source to sink. We then make sure the arcs in the BN only go from left to right with respect to this ordering, thereby guaranteeing acyclicity of the BN. Finally, we add soft clauses for each tuple in the score function cache weighted by the corresponding score. This gives us the basic (unfortified) MaxSAT encoding for the Bounded Treewidth BNSL problem.

To prepare this basic encoding for use as the local solver, we must fortify it. First, we block those arcs that could, upon replacement, form cycles when combined with existing directed paths from outside the local part. We also capture the bags that were in the overlap zone between the local solution and the rest of the solution by embedding so-called marker cliques [FLS17]. These cliques ensure that the tree decomposition of the improved local solution includes specific bags which then allows this tree decomposition to fit back into the unchanged tree decomposition with the rest of the solution. We also perform experiments to show that applying BN-SLIM to k-MAX yields better results than simply running k-MAX for the same duration.

4.1 Introduction and Motivation

Bayesian network structure learning is the notoriously difficult problem of discovering a Bayesian network (BN) that optimally represents a given set of training data [Chi02]. Since exact inference on a BN is exponential in the BN's treewidth [KBG10], one is particularly interested in learning BNs of bounded treewidth. However, learning a BN of bounded treewidth that optimally fits the data (i.e., with the largest possible score) is, in turn, an NP-hard task [KP13]. This predicament caused the research on bounded treewidth BN structure learning to split into two branches:

1. *Heuristic Learning* (see, e.g., Elidan and Gould [EG09], Nie, Campos, and Ji [NCJ15], Scanagatta et al. [Sca+16; Sca+18], and Benjumeda, Bielza, and Larrañaga [BBL19]), which is scalable to large BNs with thousands of random variables but with a score that can be far from optimal, and

2. *Exact Learning* (see, e.g., Berg, Järvisalo, and Malone [BJM14], Korhonen and Parviainen [KP13], and Parviainen, Farahani, and Lagergren [PFL14]), which learns optimal BNs but is scalable only to a few dozen random variables.

In this chapter, we combine heuristic and exact learning and take the best of both worlds.

The basic idea for our approach is to first compute a BN with a heuristic method (the *global solver*), and then to apply an exact method (the *local solver*) to parts of the heuristic solution. The parts are chosen small enough that they allow an optimal solution reasonably quickly with the exact method. Although the basic idea sounds compelling and reasonably simple, its realization requires several conceptual contributions and new results.

For the global solver, we can use any heuristic algorithm for the Bounded Treewidth BNSL problem, such as the recent algorithms k-MAX [Sca+18] or ETL [BBL19]. The local solver's task is significantly more complex than bounded treewidth BN structure learning, as several additional constraints need to be incorporated. Namely, it is not sufficient that the BN computed by the local solver is acyclic. We need *fortified acyclicity constraints* that prevent cycles that run through the other parts of the BN, which have not been changed by the local solver. Similarly, it is not sufficient that the local BN is of bounded treewidth. We need *fortified treewidth constraints* that prevent the local BN from introducing links between a diverse set of nodes that, together with the other parts of the BN, which have not been changed by the local solver, increase the treewidth.

Given these additional requirements, we propose a new local solver BN-SLIM (*SAT-based Local Improvement Method*), which satisfies the fortified constraints. We formulate a fortified version of the Bounded Treewidth BNSL problem. In Theorem 4.1, we show that we can express the fortified constraints with certain *virtual arcs* and *virtual edges*. The virtual arcs represent directed paths that run outside the local instance; with these virtual arcs we can ensure fortified acyclicity. The virtual edges constitute marker cliques and represent essential parts of a global tree decomposition using which we can ensure bounded treewidth.

The new formulation of the local problem is well-suited to be expressed as a Maximum Satisfiability (MaxSAT) problem and hence allows us to harvest the power of state-of-the-art MaxSAT solvers (which received a significant performance gain over the last decade). A distinctive feature of our encoding is that, in contrast to the virtual edges, the virtual arcs are conditional and depend on the local solver's solution.

4.1.1 Related work

The first SAT-encoding for finding the treewidth of a graph was proposed by Samer and Veith [SV09]. Lodha, Ordyniak, and Szeider [LOS16] proposed the first SAT-based local improvement method for branchwidth. Recently, SAT encodings have been proposed for other graph and hypergraph width measures [Fic+18; Gan+19; LOS17; SS20]. So far, there have been four concrete approaches that use the SLIM framework, one for branchwidth [LOS16; LOS19], one for treewidth [FLS17], one for treedepth [PS20] and one for decision trees [SS21]. SLIM is a meta-heuristic that, similarly to Large Neighborhood Search [PR10], tries to improve a current solution by exploring its neighborhood of potentially better solutions. As a distinctive feature, SLIM explores highly structurally constrained neighboring solutions with a complete method (SAT).

Several exact approaches to bounded treewidth BNSL have been proposed. Korhonen and Parviainen [KP13] proposed a dynamic-programming approach, and Parviainen, Farahani, and Lagergren [PFL14] proposed a Mixed-Integer Programming approach. Berg, Järvisalo, and Malone [BJM14] proposed a MaxSAT approach by extending the basic Samer-Veith encoding for treewidth. Our approach for BN-SLIM uses a similar general strategy, but we encode acyclicity differently. Moreover, BN-SLIM deals with the fortified constraints in terms of virtual edges and virtual arcs.

Since the exact methods are limited to small domains, Nie, Campos, and Ji [NCJ15; NCJ16] suggested heuristic approaches that scale up to hundreds of random variables. The k-greedy algorithm proposed by Scanagatta et al. [Sca+16] at NIPS'16 provided a breakthrough, consistently yielding better DAGs than its competitors and scaling up to several thousand of random variables. As mentioned above, k-MAX [Sca+18] is a more recent improvement over k-greedy. More recently, Benjumeda, Bielza, and Larrañaga [BBL19] came up with the ETL algorithms, based on local search within the space of structures called elimination trees. These algorithms perform better than k-MAX and k-greedy in many cases.

4.2 Local Improvement

Consider an instance (V, f, W) of the Bounded Treewidth BNSL problem, and assume we have computed an *initial solution* $D = (V, E)$ heuristically, together with a tree decomposition $\mathcal{T} = (T, \chi)$ of width $\leq W$ of the moralized graph $M(D)$.

S

We select a subtree $S \subseteq T$ such that the number of vertices in $V_S := \bigcup_{t \in V(S)} \chi(t)$ is at most some *budget* B. The budget is a parameter that we specify beforehand, such that the subinstance induced by V_S is small enough to be solved optimally by an exact method, which we call the *local solver*. The local solver computes for each $v \in V_S$ a new parent set, optimizing the score of the resulting DAG $D^{\text{new}} = (V, E^{\text{new}})$.

D^{new}, E^{new}

D_S^{new}, E_S^{new}

Consider the induced DAG $D_S^{\text{new}} = (V_S, E_S^{\text{new}})$, where $E_S^{\text{new}} = \{\, (u, v) \in E^{\text{new}} : \{u, v\} \subseteq V_S \,\}$. The local solver ensures that the following conditions are met:

C1 D_S^{new} is acyclic.

C2 The moral graph $M(D_S^{\text{new}})$ has treewidth $\leq W$.

$\mathcal{S}^{\text{new}}$

S^{new}, χ^{new}

We assume that the local solver certifies C2 by producing a tree decomposition $\mathcal{S}^{\text{new}} = (S^{\text{new}}, \chi^{\text{new}})$ of $M(D_S^{\text{new}})$ of width $\leq W$, which can be used by the global solver.

The two conditions stated above are not sufficient to ensure that D^{new} is acyclic and that treewidth of $M(D^{\text{new}})$ remains bounded by W. Acyclicity can be violated by cycles formed by the combination of the new incoming arcs of vertices in S together with old arcs that are kept from D. The treewidth can increase by a number that is linear in $|V_S|$.

Hence, we need additional side conditions, which we will formulate using the following additional concepts.

boundary vertex

internal vertex

external vertex

adjacent boundary vertices

Let us call a vertex $v \in V_S$ a *boundary vertex* if there exists a tree node $t \in V(T) \setminus V(S)$ such that $v \in \chi(t)$, i.e., it occurs in some bag outside S. We call the other vertices in V_S *internal vertices*, and the vertices in $V \setminus V_S$ *external vertices*. Further, we call two boundary vertices v, v' *adjacent* if there exists a tree node $t \in V(T) \setminus V(S)$ such that $v, v' \in \chi(t)$, i.e., both vertices occur together in some bag outside S. It is easy to see that any pair of adjacent boundary vertices occur together in a bag of S as well.

virtual edge

E_{virt}

extended moral graph

M_{ext}

For any two adjacent boundary vertices v, v', we call $\{v, v'\}$ a *virtual edge*. Let E_{virt} be the set of all virtual edges. These virtual edges form a clique and serve a similar purpose as the marker cliques used in other work [FLS17]. The *extended moral graph* $M_{\text{ext}} = (V_S, E_{\text{ext}})$ is obtained from $M(D_S^{\text{new}})$ by adding all virtual edges.

virtual arc

$E_{\text{virt}}^{\rightarrow}$

For any two adjacent boundary vertices v, v', we call (v', v) a *virtual arc*, if D^{new} contains a directed path from v' to v, where all the vertices on the path, except for v' and v, are external. Let $E_{\text{virt}}^{\rightarrow}$ be the set of all virtual arcs.

We can now formulate the side conditions.

C3 $\mathcal{S}^{\text{new}}$ is a tree decomposition of the extended moral graph M_{ext}.

C4 For each $v \in V_S$, if $P_{D^{\text{new}}}(v)$ contains external vertices, then there is some $t \in V(T) \setminus V(S)$ such that $P_{D^{\text{new}}}(v) \cup \{v\} \subseteq \chi(t)$.

C5 The digraph $(V_S, E_S^{\text{new}} \cup E_{\text{virt}}^{\rightarrow})$ is acyclic.

We note that condition C4 implies that in D^{new}, all parents of an internal vertex are internal.

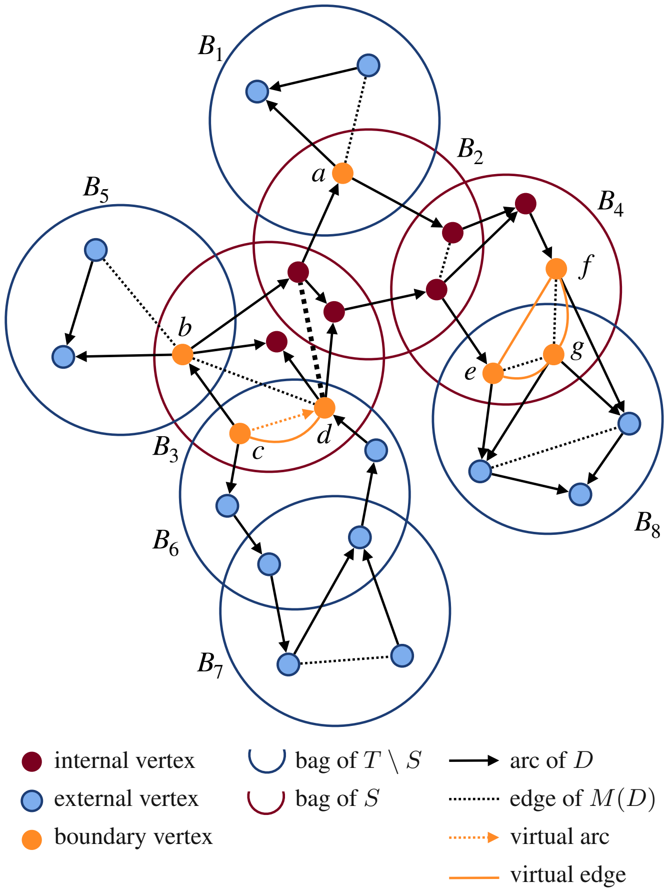


Figure 4.1: Illustration for Theorem 4.1. The large circles $B_1, \ldots, B_8$ represent the bags of T, where B_2, B_3, B_6 belong to S. The boundary vertices are $a, \ldots, g$, where c, d are adjacent, and e, f, g are mutually adjacent. Since there is a directed path from d to c using external vertices from the bags B_7 and B_8, there is a virtual arc from d to c.

Theorem 4.1. *If all the conditions C1–C5 are satisfied, then D^{new} is acyclic, the treewidth of $M(D^{\text{new}})$ is at most W, and the score of D^{new} is at least the score of D.*

Proof. We define a new tree decomposition $\mathcal{T}^{\text{new}} = (T^{\text{new}}, \chi^{\text{new}})$ of $M(D^{\text{new}})$ as follows. Let $T_1, \ldots, T_r$ be the connected components of $T \setminus V(S)$, i.e., the T_i's are the subtrees of T that we get when deleting the subtree S. Let $V_i = \bigcup_{t \in V(T_i)} \chi(t)$, $1 \leq i \leq r$, and observe that each external vertex x belongs to exactly one of the sets $V_1, \dots, V_r$. Let $B_i = V_S \cap V_i$, $1 \leq i \leq r$, be the set of boundary vertices in V_i. We observe that all the vertices in B_i are mutually adjacent boundary vertices and occur together in a bag $\chi(s_i)$ of $s_i \in V(S)$ and in a bag $\chi(t_i)$, for $t_i \in V(T_i)$, as we can take s_i and t_i to be the two neighboring tree nodes of T with $s_i \in V(S)$ and $t_i \in V(T_i)$. We also observe that each B_i forms a clique in the extended moral graph M_{ext}.

Recall that by assumption, the local solver provides a tree decomposition $\mathcal{S}^{\text{new}} = (S^{\text{new}}, \chi^{\text{new}})$ of $D_S^{\text{new}} = (V_S, E_S^{\text{new}})$ of width $\leq W$. Additionally, by condition C3, $\mathcal{S}^{\text{new}}$ is also a tree decomposition of M_{ext}, and hence, by a basic property of tree decompositions (see, e.g., Bodlaender and Möhring [BM93, Lem. 3.1]), there must exist a bag $\chi^{\text{new}}(s_i^*)$, $s_i^* \in V(S^{\text{new}})$, with $B_i \subseteq \chi^{\text{new}}(s_i^*)$. Hence, we can define T^{new} as the tree we get by connecting the disjoint trees $S^{\text{new}}, T_1, \dots, T_r$ with the edges $\{s_i^*, t_i\}$, $1 \leq i \leq r$. We extend χ^{new} from $V(S^{\text{new}})$ to $V(T^{\text{new}})$ by setting $\chi^{\text{new}}(t) = \chi(t)$ for $t \in \bigcup_{i=1}^{r} V(T_i)$.

Claim 4.1.1. *$\mathcal{T}^{\text{new}} = (T^{\text{new}}, \chi^{\text{new}})$ is a tree decomposition of $M(D^{\text{new}})$ of width $\leq W$.*

Proof. To prove the claim, we show that $\mathcal{T}^{\text{new}}$ satisfies the conditions T1 and T2.

Condition T1. There are two reasons for an edge $\{u, v\}$ to belong to $M(D^{\text{new}})$: first, because of an arc $(u, v) \in E^{\text{new}}$ and second, because of two arcs $(u, w), (v, w) \in E^{\text{new}}$. *First case:* $(u, v) \in E^{\text{new}}$. If u and v are both external, then $\{u, v\} \subseteq \chi(t) = \chi^{\text{new}}(t)$ for some $t \in V(T^{\text{new}}) \setminus V(S^{\text{new}}) = V(T) \setminus V(S)$. If neither u nor v is external, then $(u, v) \in E_S^{\text{new}}$, and since S^{new} is a tree decomposition of D_S^{new}, $\{u, v\} \subseteq \chi^{\text{new}}(s)$ for some $s \in V(S^{\text{new}})$. If v is external but u isn't, then the arc (u, v) was already present in D, as the parents of external vertices didn't change. Hence, since $\mathcal{T}$ is a tree decomposition of $M(D)$, it follows that $\{u, v\} \subseteq \chi(t) = \chi^{\text{new}}(t)$ for some $t \in V(T^{\text{new}}) \setminus V(S^{\text{new}}) = V(T^{\text{new}}) \setminus V(S)$. If u is external but v isn't, it follows from C4 that $\{u, v\} \subseteq \chi(t) = \chi^{\text{new}}(t)$ for some $t \in V(T^{\text{new}}) \setminus V(S^{\text{new}}) = V(T^{\text{new}}) \setminus V(S)$. *Second case:* $(u, w), (v, w) \in E^{\text{new}}$. If $u, v, w \in V_S$, then $\{u, v\} \in E(M(D_S^{\text{new}}))$, and so $\{u, v\} \subseteq \chi^{\text{new}}(s)$ for some $s \in V(S^{\text{new}})$, since $\mathcal{S}^{\text{new}}$ is a tree decomposition of $M(D_S^{\text{new}})$. If $w \in V_S$ but $u \notin V_S$ or $v \notin V_S$, then C4 implies that $\{u, v\} \subseteq \chi(t) = \chi^{\text{new}}(t)$ for some $t \in V(T^{\text{new}}) \setminus V(S^{\text{new}}) = V(T^{\text{new}}) \setminus V(S)$. If $w \notin V_S$, then u, v are two adjacent boundary vertices, hence $\{u, v\}$ is a virtual edge which, by C3, means $\{u, v\} \subseteq \chi^{\text{new}}(s)$ for some $s \in V(S^{\text{new}})$. We conclude that T1 holds.

Condition T2. Let $v \in V$. If v is external, then there is exactly one $i \in \{1, \dots, r\}$, such that $v \in V_i = \bigcup_{t \in V(T_i)} \chi(t)$. Since we do not change the tree decomposition of T_i, condition T2 carries over from $\mathcal{T}$ to $\mathcal{T}^{\text{new}}$. Similarly, if v is internal, then v does not appear in any bag $\chi^{\text{new}}(t)$ for $t \in V(T^{\text{new}}) \setminus V(S^{\text{new}})$, hence condition T2 carries over from $\mathcal{S}^{\text{new}}$ to $\mathcal{T}^{\text{new}}$. It remains to consider the case where v is a boundary vertex. The tree nodes $t \in V(S^{\text{new}})$ with $v \in \chi(t)$ are connected, because $\mathcal{S}^{\text{new}}$ satisfies T2, and for $B_i : v \in B_i$, the tree nodes $t \in V(T_i)$ for which $v \in \chi(t)$ are connected, since $\mathcal{T}$ satisfies T2. By construction of T^{new}, if $v \in B_i$, then there are neighboring tree nodes $s_i^* \in V(S^{\text{new}})$ and $t_i \in V(T_i)$ with $v \in \chi^{\text{new}}(s_i^*) \cap \chi^{\text{new}}(t_i)$. Hence, all the tree nodes $t \in V(T^{\text{new}})$ with $v \in \chi(t)$ are connected, and T2 also holds for boundary vertices.

To conclude the proof of the claim, it remains to observe the width of $\mathcal{T}^{\text{new}}$ cannot exceed the widths of $\mathcal{T}$ or $\mathcal{S}^{\text{new}}$, hence the width of $\mathcal{T}^{\text{new}}$ is at most W. ◀

Claim 4.1.2. *D^{new} is acyclic.*

Proof. To prove the claim, suppose to the contrary that D contains a directed cycle $C = (V(C), E(C))$. The cycle cannot lie entirely in D_S^{new}, nor can it lie entirely in $D^{\text{new}} - V_S = D - V_S$, because D_S^{new} and D are acyclic. Hence, C contains at least one arc from $V_S \times (V \setminus V_S)$ and at least one arc from $(V \setminus V_S) \times V_S$. Let $(v_j, x_j) \in E(C) \cap (V_S \times (V \setminus V_S))$ and $(x'_j, v'_j) \in E(C) \cap ((V \setminus V_S) \times V_S)$, for $0 \leq j \leq p$, be these arcs, such that they appear on C in the order (v'_0, x'_0), (x_0, v_0), $(v'_1, x'_1), \ldots, (v'_p, x'_p)$, (x_p, v_p). It is possible that $x'_j = x_j$ or $v_j = v'_{j+1}$. We observe that the vertices on the path from x'_j to x_j on C all belong to some $V_i = \bigcup_{t \in V(T_i)} \chi(t)$. Hence, v_j and v'_j are adjacent boundary vertices, and $E_{\overrightarrow{\text{virt}}}$ contains all the arcs (v'_j, v_j), $1 \leq j \leq p$. However, the cycle C contains also the paths from v_j to $v'_{j+1 \pmod p}$, for $1 \leq j \leq p$, which only run through vertices in V_S. These paths, together with the virtual arcs (v'_j, v_j) form a cycle C' which lies in $(V_S, E_S^{\text{new}} \cup E_{\overrightarrow{\text{virt}}})$. This contradicts C5 which requires that this digraph be acyclic. Hence the claim holds. ◀

Claim 4.1.3. *The score of D^{new} is at least the score of D.*

Proof. We observe that by taking $D^{\text{new}} = D$ we have a solution that satisfies all the required conditions and maintains the score. ◀

This concludes the proof of the theorem. □

4.3 Implementing the Local Improvement

In this section, we first discuss how the set S representing the subinstance is constructed. Then we provide a detailed explanation of the MaxSAT encoding that is responsible for solving the subinstance.

4.3.1 Constructing the subinstance

For this section, we follow the same notation as used in the previous section. To construct the subinstance, we initialize the subtree S with a tree node r picked at random from $V(T)$. We then expand S by performing a bread-first search from $V(S)$ and adding a new tree node to S as long as the size of V_S does not exceed the budget. Next, we compute E_{virt} for the chosen S. Finally, we prune the parent sets of each vertex so as to only retain those parent sets which satisfy conditions C3 and C4. This can be done by first checking, for each parent set, if the required tree node t is present $V(T) \setminus V(S)$, and if it does, we record the set of virtual arcs (excluding self-loops) that are imposed by this parent

$A^{\to}_{\text{virt}}(\cdot,\cdot)$ set. For each $v \in S$ and $P \in \mathcal{P}_v$, we denote by $A^{\to}_{\text{virt}}(v, P)$ the set of imposed virtual arcs when v has the parent set P in D^{new}. We denote by $\mathcal{P}_v$, the collection of parent sets of node v that remain after this pruning process. Notice that, under this pruning, all remaining parent sets $P \in \mathcal{P}_v$ satisfy C4. Also note that, since $E^{\to}_{\text{virt}}$ is conditional on the chosen parent sets, it cannot be precomputed.

Further, since we intend to solve the subinstance using a MaxSAT encoding, we need to ensure that the score of each parent set is non-negative. Recall that $\mathcal{P}_v$ only contains those non-empty parent sets whose score is at least that of the empty parent set. Thus, we may assume that the empty parent set has the lowest score among all the parents of a certain vertex. Consequently, we can adjust the score function by setting $f'(\cdot,\cdot)$ $f'(P, v) = f(P, v) - f(\emptyset, v)$ for $v \in S$ and $P \in \mathcal{P}_v$, which implies that $f'(P, v) \geq 0$ for all $v \in S$ and $P \in \mathcal{P}_v$.

4.3.2 MaxSAT encoding

We now describe the weighted partial MaxSAT instance that encodes conditions C1–C5. We build on top of the SAT encoding proposed by Samer and Veith [SV09]. The only difference in our case is that there are no explicit edges and hence we do not require the corresponding clauses. Instead, the edges of the moralized graph are dependent on and decided by other variables that govern the DAG structure. For convenience, let n denote the size of the subinstance, i.e., $n := |S|$. A part of the encoding is based on the elimination ordering of a tree decomposition (see, e.g., Samer and Veith [SV09, Sec. 2]).

The main variables used in our encoding are

- variables par_v^P represent for each node $v \in S$ the chosen parent set P,
- $n(n-1)/2$ variables $\text{acyc}_{u,v}$ represent the topological ordering of D_S^{new},
- $n(n-1)/2$ variables $\text{ord}_{u,v}$ represent the elimination ordering of the tree decomposition,
- n^2 variables $\text{arc}_{u,v}$ represent the arcs in the moralized graph M_{ext}, along with the *fill-in edges* (see Samer and Veith [SV09]).

Since $\text{acyc}_{u,v}$ and $\text{ord}_{u,v}$ represent linear orderings, we enforce transitivity of these variables by means of the clauses

$$\left.\begin{aligned} (\text{acyc}^*_{u,v} \wedge \text{acyc}^*_{v,w}) &\rightarrow \text{acyc}^*_{u,w} \\ (\text{ord}^*_{u,v} \wedge \text{ord}^*_{v,w}) &\rightarrow \text{ord}^*_{u,w} \end{aligned}\right\} \text{ for distinct } u, v, w \in S.$$

To prevent self-loops in the moralized graph, we add the clauses

$$\neg\text{arc}_{v,v} \qquad \text{for } v \in S.$$

For each node $v \in S$, and parent set $P \in \mathcal{P}_v$, the variable par_v^P is true if and only if P is the parent set of v. Since each node must have exactly one parent set, we introduce the cardinality constraint

$$\sum_{P \in \mathcal{P}_v} \mathrm{par}_v^P = 1 \text{ for } v \in S.$$

Next, for each node v, parent set P, and $u \in P$, if P is the parent set of v then u must precede v in the topological ordering. Hence we add the clause

$$\mathrm{par}_v^P \to \mathrm{acyc}_{u,v} \quad \text{for } v \in S, P \in \mathcal{P}_v, \text{ and } u \in P.$$

Similarly, for each node v, parent set P, and $u \in P$, if P is the parent set of v then we must add an arc in the moralized graph respecting the elimination ordering between u and v, as follows:

$$\left.\begin{array}{l}(\mathrm{par}_v^P \wedge \mathrm{ord}_{u,v}) \to \mathrm{arc}_{u,v} \\ (\mathrm{par}_v^P \wedge \mathrm{ord}_{v,u}) \to \mathrm{arc}_{v,u}\end{array}\right\} \quad \begin{array}{r}\text{for } v \in S, P \in \mathcal{P}_v, \\ \text{and } u \in P.\end{array}$$

Next, we encode the moralization by adding an arc between every pair of parents of a node, using the following clauses

$$\left.\begin{array}{l}(\mathrm{par}_v^P \wedge \mathrm{ord}_{u,w}) \to \mathrm{arc}_{u,w} \\ (\mathrm{par}_v^P \wedge \mathrm{ord}_{w,u}) \to \mathrm{arc}_{w,u}\end{array}\right\} \quad \begin{array}{r}\text{for } v \in S, P \in \mathcal{P}_v, \\ \text{and } u, w \in P.\end{array}$$

Now, we encode the fill-in edges, with the following clauses

$$\left.\begin{array}{l}(\mathrm{arc}_{u,v} \wedge \mathrm{arc}_{u,w} \wedge \mathrm{ord}_{v,w}) \to \mathrm{arc}_{v,w} \\ (\mathrm{arc}_{u,v} \wedge \mathrm{arc}_{u,w} \wedge \mathrm{ord}_{w,v}) \to \mathrm{arc}_{w,v}\end{array}\right\} \quad \text{for } u, v, w \in S.$$

Lastly, to bound the treewidth, we add a cardinality constraint on the number of outgoing arcs for each node as follows

$$\textstyle\sum_{w \in S, w \neq v} \mathrm{arc}_{v,w} \leq W \quad \text{for } v \in S.$$

To complete the basic encoding, for every node $v \in S$, and every parent set $P \in \mathcal{P}_v$ we add a unit soft clause weighted by the score of the parent set as follows

$$(\mathrm{par}_v^P) : \text{weight } f'(P, v) \quad \text{for } v \in S, P \in \mathcal{P}_v.$$

To speed up the solving, we encode that for every pair of nodes, at most one of the arcs between them can exist. We add the following redundant clauses

$$\neg\mathrm{arc}_{u,v} \vee \neg\mathrm{arc}_{v,u} \quad \text{for } u, v \in S.$$

Now, we describe the additional clauses required to satisfy the fortified constraints, and thus conditions C3 and C5. For every virtual edge $\{u, v\} \in E_{\text{virt}}$, we introduce a forced arc depending on the elimination ordering using the following pair of clauses

$$\text{ord}^*_{u,v} \to \text{arc}_{u,v} \wedge \text{ord}^*_{v,u} \to \text{arc}_{v,u} \quad \text{for } \{u, v\} \in E_{\text{virt}}.$$

This takes care of the fortified treewidth constraints, satisfying C3 and ensuring that the edge $\{u, v\} \subseteq \chi(s)$ for some $s \in V(S)$. Finally, we add the clauses that encode the forced arcs $E^{\to}_{\text{virt}}$. For each $v \in S$, $P \in \mathcal{P}_v$, and $(u, v) \in A^{\to}_{\text{virt}}(v, P)$, we add the clause

$$\text{par}^P_v \to \text{acyc}^*_{u,v},$$

which forces the virtual arc (u, v) if P is the parent set of v in D^{new}, thereby handling the fortified acyclicity constraints and ensuring that C5 is satisfied.

$\Phi_{D,f}(S)$ This concludes the definition of the MaxSAT instance, to which we will refer as $\Phi_{D,f}(S)$. We refer to the weight of a satisfying assignment τ of $\Phi_{D,f}(S)$ as the sum of weights of all the soft clauses satisfied by τ. Let $\alpha(S) := \sum_{v \in S} f(\emptyset, v)$. To each satisfying assignment τ of $\Phi_{D,f}(S)$ we can associate for each $v \in V$ the corresponding parent set, which in turn determines a directed graph D^{new}. Due to Theorem 4.1, the treewidth of $M(D^{\text{new}})$ is bounded by W, and D^{new} is acyclic. By construction of $\Phi_{D,f}(S)$, the weight of τ equals $\sum_{v \in S} f'(P, v) = f(D^{\text{new}}_S) - \alpha(S)$. Conversely, if we pick new parent sets for the vertices in S such that all the conditions C1–C5 are satisfied, then by construction of $\Phi_{D,f}(S)$, the corresponding truth assignment τ satisfies $\Phi_{D,f}(S)$, and K_0 its weight is $\sum_{v \in S} f'(P, v) = f(D^{\text{new}}_S) - \alpha(S)$. In particular, let K_0 be the weight of the truth assignment which corresponds to the parent sets of S as defined by the input DAG D. We summarize these observations in the following theorem.

Theorem 4.2. *$\Phi_{D,f}(S)$ has a solution of weight K if and only if there are new parent sets for the vertices in S giving rise to a DAG D^{new} with $f(D^{\text{new}}) - f(D) = K - K_0$.*

4.4 Experimental Evaluation

In this section, we describe the experiments conducted to analyze the performance of the local improvement algorithm. The current state-of-the-art heuristic algorithms for solving the Bounded Treewidth BNSL problem are the k-MAX algorithm by Scanagatta et al. [Sca+18] and the ETL algorithms by Benjumeda, Bielza, and Larrañaga [BBL19] (available as two variants–the default variant ETL_d and the poly-time variant ETL_p). Therefore, we analyze the benefit of applying BN-SLIM on top of these algorithms. It is worth noting that both k-MAX and BN-SLIM are anytime algorithms, i.e., they run indefinitely long and can be halted at any instant to output the best solution found so far; ETL, on the other hand, as per the available implementation, is deterministic and terminates when it fails to find any new improvements. This distinction affects the nature of the experiments conducted to draw a comparison between the different algorithms. However, for the most part, we closely follow the experimental setup (including data sets, timeouts, comparison metrics) used by Scanagatta et al. [Sca+18] to compare k-MAX with previous approaches.

Since BN-SLIM needs an initial heuristic solution, we enlist either k-MAX, ETL_d, or ETL_p for this purpose. We denote by BN-SLIM (X), the algorithm which applies BN-SLIM on an initial solution provided by X where X $\in$ {k-MAX, ETL_d, ETL_p}. We run all our experiments with treewidth bounds 2, 5, 8 for each data set following Scanagatta et al. [Sca+18]. All reported BN-SLIM results are averages over three random seeds.

4.4.1 Setup

We run all our experiments on a 4-core Intel Xeon E5540 2.53 GHz CPU, with each process having access to 8 GB RAM. We use UWrMaxSat as the MaxSAT-solver primarily due to its anytime nature (available at the 2019 MaxSAT Evaluation webpage[1]). After trying other solvers we found that UWrMaxSat works best for our use case. We use the BNGenerator package [Ide15] in conjunction with the BBNConvertor tool [Guo] to generate and reformat random Bayesian Networks. We also use the implementation of the k-MAX algorithm available as a part of the BLIP package [Sca15]. For the ETL algorithms we use the software made available[2] by Benjumeda, Bielza, and Larrañaga [BBL19]. We implement the local improvement algorithm in Python 3.6.9, using the NetworkX 2.4 graph library [HSS08].

The source code along with the experiment data is available publicly at https://github.com/aditya95sriram/bn-slim.

We first conducted a preliminary analysis on 20 data sets to find out the best values for the *budget* (maximum number of random variables in a subinstance) and the *timeout* (per MaxSAT call) of BN-SLIM. We tested out budget values 7, 10, and 17, and timeout values 1 second, 2 seconds, and 5 seconds; and finally settled on a budget of 10 and a timeout of 2 seconds for our experiments.

4.4.2 Data sets

We consider 99 data sets for our experiments. Of these, 84 data sets come from *real-world* benchmarks. These are based on the benchmarks introduced by Lowd and Davis [LD10], Van Haaren and Davis [VD12], Bekker et al. [Bek+15], and Larochelle, Bengio, and Turian [LBT10], a subset of which has been used by Scanagatta et al. [Sca+18]. These benchmarks are publicly available[3] in the form of pre-partitioned data sets. There are three data sets corresponding to each of the 28 benchmarks (see Table 4.1).

The remaining 15 data sets are classified as *synthetic* as they are obtained by drawing 5000 samples from known BNs (see Table 4.2). Five of these BNs are commonly used in

[1] https://maxsat-evaluations.github.io/2019/descriptions.html

[2] https://github.com/marcobb8/et-learn

[3] https://github.com/arranger1044/DEBD

the literature as benchmarks[4], and we generated the remaining 10 BNs randomly using the BNGenerator tool with more random variables than the previously mentioned data sets. Overall, the collection of data sets provides a wide variety of the data's nature and the different parameters.

Both k-MAX and BN-SLIM take a score function cache as input, while ETL requires the samples themselves and computes the required scores on-the-fly. We thus compute the score function cache using the scoring module provided as a part of ETL's source code. More specifically, we first obtain the parent set tuples using independence selection (available in the BLIP package), and then we recompute the scores for these tuples using ETL's scoring module. This cache is used as input to both BN-SLIM and k-MAX. This provides a level playing field and improves comparability between the different algorithms.

While computing these score function caches, the scoring function module was unable to process two data sets and hence we discarded these two data sets. The final list of data sets is shown in Tables 4.1 and 4.2. Further, k-MAX crashes for 3 data sets and hence we disregard these for any experiments involving k-MAX or BN-SLIM(k-MAX).

Name	n	Name	n	Name	n	Name	n
Kdd	64	Accidents	111	MSWeb	294	C20NG	910
Plants	69	Retail	135	Book	500	BBC	1058
Audio	100	Pumsb-star	163	EachMovie	500	Ad	1556
Jester	100	DNA	180	WebKB	839		
Netflix	100	Kosarek	190	Reuters-52	889		

Table 4.1: Real-world data sets (n is the number of random variables, the number of samples ranges from 100 to 291326)

Name	n	Name	n	Name	n	Name	n
andes	223	r0	2000	r5	4000	r10	10000
diabetes	413	r1	2000	r6	4000	r11	10000
pigs	441	r2	2000	r7	4000	r12	10000
link	724	r3	2000	r8	4000	r13	10000
munin	1041	r4	2000	r9	4000	r14	10000

Table 4.2: Synthetic data sets (n denotes the number of random variables, 5000 samples from each network)

4.4.3 Evaluation metric

For evaluating our algorithm's performance, we use the same metric as Scanagatta et al., i.e., ΔBIC, which is the difference between the BIC scores of two solutions. Given a ΔBIC

[4] https://www.bnlearn.com/bnrepository/

DAG D, the BIC score approximates the logarithm of the marginal likelihood of D. Thus, given two DAGs D_1 and D_2, the difference in their BIC scores approximates the ratio of their respective marginal likelihoods which is the Bayes Factor [Raf95]. A positive ΔBIC score signifies positive evidence towards D_1 and a negative ΔBIC score signifies positive evidence towards D_2. The ΔBIC values can be mapped to a scale of qualitative categories [Raf95] as shown in Table 4.3.

Category	ΔBIC
🟥 extremely negative	$(-\infty, -10)$
🟥 strongly negative	$(-10, -6)$
🟥 negative	$(-6, -2)$
🟦 positive	$(2, 6)$
🟦 strongly positive	$(6, 10)$
🟦 extremely positive	$(10, \infty)$

Table 4.3: ΔBIC category scale

4.4.4 Experimental results

The primary focus of our experimentation is to analyze the benefit gained by applying BN-SLIM on top of other heuristics and not to compare between the different heuristics. To this end, we run BN-SLIM for 60 minutes on top of the initial solution provided by k-MAX, $\mathrm{ETL_d}$, and $\mathrm{ETL_p}$ and measure the time required for BN-SLIM to obtain a solution that counts as extremely positive evidence with respect to the initial solution. The initial solution by k-MAX is the solution captured at the 30-minute mark, whereas the initial solution by ETL is the final solution obtained upon termination. The maximum time required for computing the initial solution on any individual instance, by both $\mathrm{ETL_d}$ and $\mathrm{ETL_p}$, is around 3.5 hours. For comparison, we let k-MAX continue running for 60 more minutes after it has produced the initial solution.

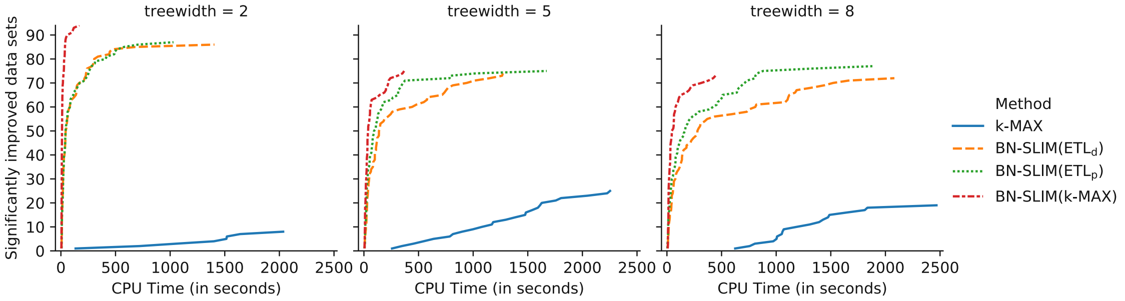


Figure 4.2: CDF plots showing the number of significantly improved data sets (ΔBIC ≥ 10) across 94 data sets

Fig. 4.2 shows the results of this analysis. We consider a data set to be *significantly improved* if BN-SLIM is able to improve by at least 10 BIC points over the initial heuristic

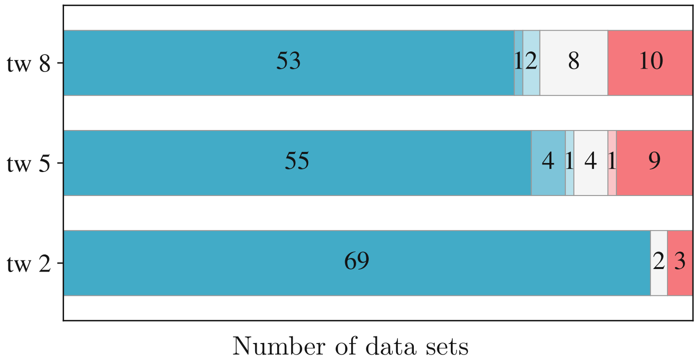
significantly improved

solution. We observe that BN-SLIM improves over k-MAX much more efficiently as over ETL. Giving k-MAX more time for computing the initial solution increases this discrepancy even further, as the improvement rate of k-MAX rapidly slows down after 30 minutes. Averaging over all the heuristics, BN-SLIM can produce a solution with extremely positive evidence for 95%, 79%, and 78% of instances for treewidth bounds 2, 5, and 8, respectively.

Fig. 4.4 shows the ΔBIC values from comparing the BN-SLIM(ETL) solution after 30 minutes to the corresponding initial solution by ETL. We can see that BN-SLIM(ETL) can secure extremely positive evidence for a significant number of data sets across all tested treewidth bounds, with a smaller treewidth being more favorable.

Due to the anytime nature of k-MAX, we can compare it against BN-SLIM(k-MAX) in a "race." We run both simultaneously for one hour, where out of the time allotted to BN-SLIM(k-MAX), 30 minutes are used to generate the initial solution, and the remaining 30 minutes are used to improve this initial solution. Fig. 4.3 shows the ΔBIC values of comparing k-MAX and BN-SLIM(k-MAX) at the one-hour mark. Similar to BN-SLIM(ETL) we observe that BN-SLIM(k-MAX) outperforms k-MAX on a significant number of instances, and on all instances for treewidth 2.

The experimental evaluation demonstrates BN-SLIM approach's effectiveness and the combined power as a heuristic method of BN-SLIM(k-MAX) and BN-SLIM(ETL).



ΔBIC	treewidth 2	5	8
🟦 extreme positive	69	55	53
🟦 strong positive	0	4	1
🟦 positive	0	1	2
⬜ neutral	2	4	8
🟥 negative	0	1	0
🟥 strong negative	0	0	0
🟥 extreme negative	3	9	10

BN-SLIM(k-MAX) vs k-MAX

Figure 4.3: Comparison between BN-SLIM(k-MAX) and k-MAX over 94 data sets

4.5 Conclusion/Related Work

With BN-SLIM, we have presented a novel method for improving the outcome of bounded treewidth BNSL heuristics. We have demonstrated its robustness and performance by applying BN-SLIM to the solution provided by the state-of-the-art heuristics k-MAX, ETL_d, and ETL_p. The approach of BN-SLIM is based on exact reasoning via MaxSAT, which is fundamentally different from the mentioned heuristics. Consequently, both approaches complement each other, and their combination provides significantly better solutions than any of the heuristics alone. Simultaneously, the combination still scales to

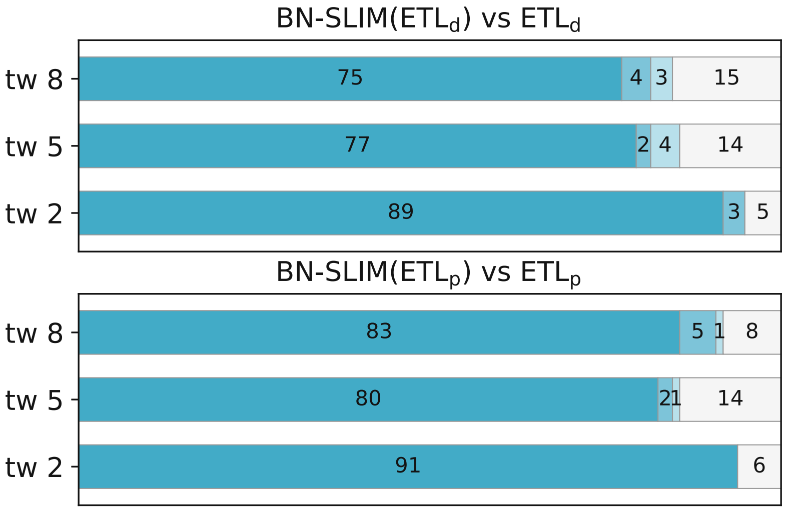


Figure 4.4: Comparison between BN-SLIM(ETL) and ETL over 97 data sets

large instances with thousands of random variables, which are far out of reach for exact methods alone. Thus, BN-SLIM combines the best of both worlds.

The highly encouraging experimental outcome suggests several avenues for future work, which include the development of more sophisticated subinstance selection schemes, the inclusion of variable fidelity sampling (crude for the global solver, fine-grained for the local solver), as well as more complex collaboration protocols between local and global solver in a distributed setting.

CHAPTER 5

Bounded State-Space BNSL

Everything isn't black or white, yes or no. Sometimes it's not a switch, it's a dial.

— Jeff Garvin, "Symptoms of Being Human"

Published as "Learning Fast-Inference Bayesian Networks". Peruvemba Ramaswamy and Szeider. In: Advances in Neural Information Processing Systems, (2021) [PS21b]

We started our journey with the widely popular Bounded Treewidth BNSL problem. The main motivation for bounding the treewidth was to guarantee quick inference. Usually, a BN is learned only once and then used to make several predictions in the form of inferences. Thus, it is critical to ensure extremely quick inference times. The main factor governing the inference time is the number of rows in the probability tables of the learned BN. Bounding the treewidth usually also bounds this underlying measure and hence guarantees quick inference. However, treewidth is a good proxy only as long as the BN contains RVs whose domain sizes are all the same.

In this chapter, we look at a more fine-grained measure called the state space size and why it is a better candidate than treewidth to ensure quick inference. The state space size of a BN takes into account the domain sizes of the variables and hence is able to provide a much better handle on the inference time. We first show empirically that indeed, the state space size correlates to inference time more tightly than treewidth (Figure 5.2). Then, we develop a SLIM-based algorithm BN-SLIM$^{\text{bss}}$ for learning bounded state space BNs by extending BN-SLIM. To develop BN-SLIM$^{\text{bss}}$, we modify k-greedy and k-MAX such that they only consider parent sets which respect the state space bound, and we upgrade the encoding to express the state space size instead of the treewidth. Since computing the state space is more involved than computing the treewidth, we use a sophisticated counting mechanism (illustrated in Figure 5.3) which is capable of keeping track of sums of non-integer values as well.

5.1 Introduction and Motivation

The time complexity of probabilistic reasoning on a Bayesian Network (BN) is dominated by the *maximum state space size* of clusters (i.e., bags of the BN's tree decomposition) [LS88; Dec99; Kas+11]. A bag's state space size is the product of the domain sizes of all the variables it contains. This metric has been previously studied under different names such as "total state space", "maximum complexity", and "complexity width" [Kas+11; Kjæ92; MJ96; OD08b; OD08a]. It is a more fine-grained metric as compared to the extensively studied treewidth metric [EG09; NCJ15; Sca+16; Sca+18; BBL19; BJM14; KP13; PFL14; PS21c]. We propose algorithms for learning BNs from data, keeping the state space size within a user-specified bound. This results in *fast-inference* BNs, i.e., BNs reliably admitting fast probabilistic reasoning. We compare our algorithms to the baseline of state-of-the-art bounded treewidth BN learning algorithms, on real-world benchmark data sets, with up to over a thousand variables. The results show a clear advantage for our bounded state space (*bss*) BNSL algorithms.

It is common to encounter non-binary variables in real-world data. Moreover, during our preliminary analysis, we noticed that even variables with domain sizes as small as 4 were sufficient to impact the reasoning times significantly. This is in agreement with the fact that the reasoning time has an exponential dependence on the domain sizes. For instance, consider some of the networks learned for `alarm` and `hepar2` having 37 and 70 variables, respectively. Both these data sets were learned with small values of treewidth, and the maximum domain size of the variables is 4. Despite this, they exhibited reasoning times in the order of magnitude of 2.5 seconds.

For our bss BNSL algorithms, we build upon recent work on bounded treewidth BNSL, particularly on the heuristic algorithms k-greedy and k-MAX by Scanagatta et al. [Sca+16; Sca+18] as well as our BN-SLIM algorithm [PS21c], see Chapter 4. The latter is a post-processing algorithm that uses MaxSAT to improve BNs generated by the heuristics. All these algorithms assume a user-specified upper bound k for the treewidth of the learned BN and optimize the BN's score under the given treewidth bound. The learning algorithms are highly optimized for dealing with large instances, and so the generalization of treewidth bounds to state space bounds isn't straightforward. The main challenge for extending BN-SLIM to bss learning is to replace BN-SLIM's simple cardinality constraints with a MaxSAT encoding that bounds the state-space of a bag, i.e., a product of integers. We achieve this by switching to logarithms and bounding the sum of real numbers, utilizing a MaxSAT encoding based on *binary decision diagrams (BDDs)*.

We consider several variants of bss BNSL algorithms, tested them on 16 real-world benchmark data sets with up to 1041 variables, and compare them with bounded treewidth BN learning algorithms. For the comparison, we put pairs of scatter plots side by side, which show the trade-off between reasoning speed and data-fitting (score), one for the baseline methods and one for the bounded state space methods. The bounded state space methods show better performance throughout, with significantly higher reliability (small variance).

In total, we consider six algorithm schemes for bounded state space BNSL. We tested them on 16 real world benchmark data sets, with up to 1041 variables, and compared them with bounded treewidth BN learning algorithms. The results are overwhelmingly positive, showing a clear advantage for the bounded state space based methods. For the comparison, we put pairs of scatter plots side by side, which show the trade-off between responsiveness (inference speed) and data-fitting (score), one for the baseline methods and one for the bounded state space methods. The bounded state space methods show a better performance throughout, with a significantly better reliability (small variance).

5.1.1 Related Work

We discuss related work in terms of Figure 5.1. In approaches (a) and (b), the BN has

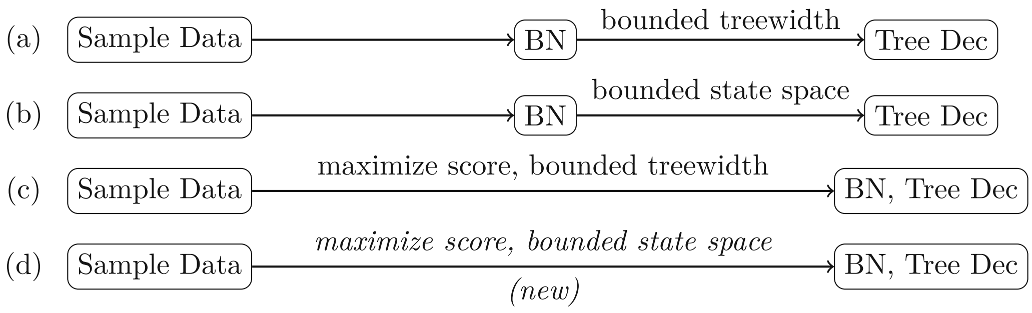


Figure 5.1: Various approaches to structure learning of BNs

already been learned by some other method, and one tries to find a tree decomposition that minimizes either the treewidth or the maximum state space size, respectively. For approach (a), general-purpose tree decomposition algorithms can be applied, such as the one by Gogate and Dechter [GD04]. As the significance of the state space of BNs was recognized [LS88], the research focused on approach (b) [Kas+11; Kjæ92; MJ96; OD08b; OD08a]. However, once the BN has been fixed, the impact of the decomposition method is limited. Therefore, the research in the last decade focused on approach (c), where a treewidth bound is already considered during the BN learning; a suitable decomposition is produced simultaneously. On the one hand, exact learning algorithms have been proposed that scale only to small instances but find score-optimal BNs [KP13; PFL14; BJM14]. On the other hand, heuristic algorithms have been proposed that scale to large instances but do not guarantee score optimality [NCJ15; NCJ16; Sca+16; Sca+18; BBL19]. In Chapter 4, we proposed the hybrid approach BN-SLIM, which improves the score of a heuristically computed BN by multiple applications of a MaxSAT-based exact method.

In this chapter, we follow approach (d) for the first time and implement it via various scalable algorithms. Our experiments compare approaches (c) and (d) in terms of the achieved score and inference speed.

5.2 Treewidth and Maximum State Space Size

In this section, we discuss the different metrics that can be used to estimate the inference speed of a BN along with some empirical findings.

Consider a tree decomposition $\mathcal{T} = (T, \chi)$ of a DAG D, where $V(D)$ consists of random variables, each $v \in V(D)$ ranging over a set of $\mathrm{ds}(v)$ many discrete values.

$\mathrm{ds}(\cdot)$

maximum state space size

The *width* of $\mathcal{T}$ is

$$\max_{t \in V(T)} |\chi(t)| - 1,$$

i.e., is the size of a largest bag minus 1. The *treewidth* $\mathrm{tw}(D)$ of D is the minimum width over all tree decompositions of $M(D)$.

The *maximum state space size* of $\mathcal{T}$ is

$$\max_{t \in V(T)} \prod_{v \in \chi(t)} \mathrm{ds}(v),$$

i.e., largest state space size of all bags, where the state space of a bag is the product of the domain sizes of the variables it contains. The *maximum state space size* $\mathrm{msss}(D)$ of D is the minimum msss over all tree decompositions of $M(D)$.

The maximum state space of a binary BN of treewidth t is 2^{t+1}. The *bounded treewidth (state space, respectively) BN structure learning problem* takes as input a set V of nodes (i.e., random variables), a decomposable score function f on V, and an integer k, and asks to compute a DAG D with $V(D) = V$ of treewidth (maximum state space size, respectively) at most k, with a maximal score $f(D)$.

5.2.1 Empirical Influence on Inference Speed

Complexity results for probabilistic reasoning (inference) suggest that for BNs containing non-binary variables, maximum state space size provides a more accurate prediction for inference speed than treewidth [LS88; Dec99; Kas+11]. Our initial experiments aimed to verify this theoretical assumption empirically. For this purpose, we generated several BNs with varying treewidth and maximum state space size and analyzed the impact of these two measures on the BN's inference speed. We define a BN's reasoning time as the time required for computing the probability of evidence of 5 random variables set to random states, averaged over 100 runs (same as [Sca+16]). For more details on the experimental setup, we refer to Section 5.4.

Figure 5.2 depicts the distribution of the observed reasoning times for different treewidth and maximum state space size ranges utilizing box plots, with dots signifying outliers. We observe that the correlation between reasoning time and maximum state space size is much stronger than the correlation between reasoning time and treewidth. The number of outliers is much higher in the treewidth plot as compared to the maximum state space size plot. Furthermore, for gradually increasing reasoning time thresholds, the

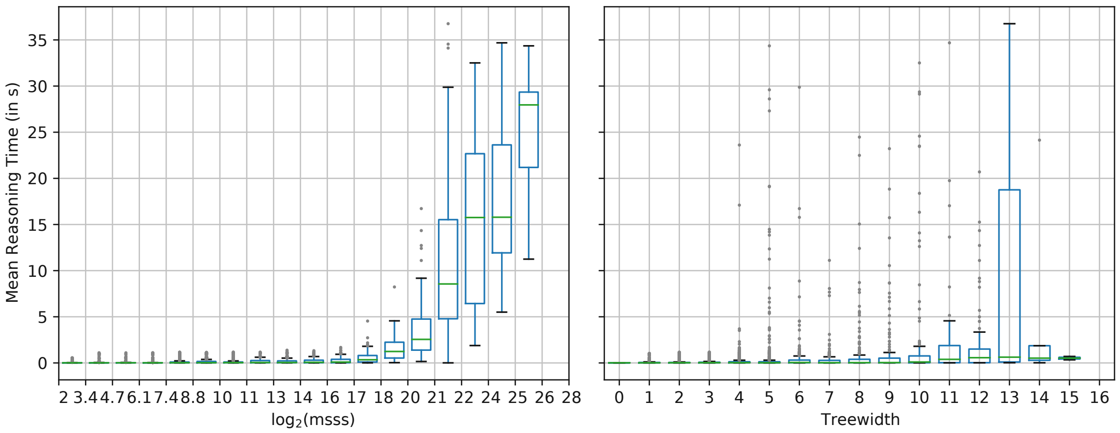


Figure 5.2: Comparison of correlation of treewidth and maximum state space size with reasoning time

corresponding maximum state space size, for which none of the BNs exceed the reasoning time, grows much more gradually than the corresponding treewidth.

These results provide a solid basis for our objective to develop algorithms that already bound maximum state space size during BN structure learning, as we will lay out in the next sections.

5.3 BN Learning of Bounded State Space Size

5.3.1 Modified Heuristics

We now describe the modifications we made to the k-MAX and k-greedy heuristics to operate with a bound on the state space.

Now, we give a brief outline of the k-greedy algorithm. Let us assume a treewidth bound of k. The algorithm starts from a random ordering of the random variables. *Initialization:* It then initializes the first bag with the first $k+1$ variables. It computes the best DAG over these variables either by an exact method or by an approximate method depending on the value of $k+1$. *Addition:* Then, the algorithm iteratively adds variables to this DAG with parent sets that maximize local score. While doing so, the algorithm searches through the existing k-cliques as potential parent sets. *Termination:* The iteration continues until there are no variables left to add.

To make the k-greedy algorithm work for bounded state space, we first modify the Initialization step. Let $\sigma = v_1, v_2, \ldots, v_n$ be the randomly sampled ordering. Instead of simply picking the first k variables, we now pick the first p variables such that p is the largest integer satisfying the condition $\prod_{i=1}^{p} v_i \leq k$. In the Addition step, we no longer treat all existing k-cliques as potential parent sets. Instead, when adding variable v, we only consider those sets S for which $\prod_{u \in S \cup \{v\}} \mathrm{ds}(u) \leq k$.

The k-MAX algorithm is similar to the k-greedy algorithm but picks the variables based on a scoring system instead of following a particular variable ordering. The modifications we propose for the k-greedy algorithm can, however, be easily adjusted to work for the k-MAX algorithm by incorporating domain-size checks in place of cardinality checks.

We thus obtain the following algorithms:

k-greedy and k-MAX refer to the original two heuristics proposed by Scanagatta et al. [Sca+16; Sca+18] that return bounded treewidth BNs.

k-greedy$^{\text{bss}}$ and k-MAX$^{\text{bss}}$ refer to the modified versions of k-greedy and k-MAX with the proposed modifications that return bounded state space BNs.

As a side effect, all these algorithms produce a tree decomposition that witnesses the treewidth or state space bound of the obtained BN.

5.3.2 Local Improvement

In this section, we explain how the local improvement framework from Chapter 4 can be extended and utilized for the Bounded State Space BNSL problem.

Throughout this section, consider an instance of the Bounded State Space BNSL problem, consisting of a set V of random variables, a score function f, and a state space bound k. *Initialization:* We first use a heuristic (such as described in Subsection 5.3.1) to compute an *initial solution* D, together with a tree decomposition $\mathcal{T} = (T, \chi)$ of the moralized graph $M(D)$ with maximum state space size $\leq k$. The local improvement step uses the parameter *budget* β which controls the size of the local instance. *Subtree selection:* We select a subtree S of T such that $V_S := \bigcup_{t \in V(S)} \chi(t)$ does not exceed the budget β. Our aim is to compute for each $v \in V_S$ a new parent set, optimizing the score of the resulting DAG D^{new} with $V(D^{\text{new}}) = V$. We define D_S^{new} as the DAG induced by V_S where $E(D_S^{\text{new}}) = \{\, (u, v) \in E(D^{\text{new}}) : \{u, v\} \subseteq V_S \,\}$. We distinguish between different kinds of nodes:

S

D^{new}

D_S^{new}

- *boundary vertex* $v \in V_S$ a *boundary vertex* if there exists a tree node $t \in V(T) \setminus V(S)$ such that $v \in \chi(t)$;
- *internal vertex* $v \in V_S$ is an *internal vertex* if v is not a boundary vertex;
- *external vertex* $v \in V \setminus V_S$ is an *external vertex*.

adjacent boundary vertices

virtual edge

extended moral graph

M_{ext}

Two boundary vertices v, v' are *adjacent* if both occur together in some bag outside S. In that case we call $\{v, v'\}$ a *virtual edge*. We let E_{virt} be the set of all virtual edges. The *extended moral graph* M_{ext} is obtained from $M(D_S^{\text{new}})$ by adding all virtual edges. If v, v' are two adjacent boundary vertices such that D^{new} contains a directed path from v' to v, where all the vertices on the path, except for v' and v, are external, then (v', v) is a *virtual arc*. $E_{\text{virt}}^{\rightarrow}$ denotes the set of all virtual arcs.

virtual arc

$E_{\text{virt}}^{\rightarrow}$

We call D_S^{new} a *well-behaved* DAG with respect to DAG D and its tree decomposition (T, χ) if the following conditions are satisfied:

well-behaved

1. D_S^{new} is acyclic.
2. For each $v \in V(M_{\text{ext}})$, if $P_{D^{\text{new}}}(v)$ contains external vertices, then there is some $t \in V(T) \setminus V(S)$ such that $P_{D^{\text{new}}}(v) \cup \{v\} \subseteq \chi(t)$.
3. The digraph with vertex set $V(M_{\text{ext}})$ and arc set $E(D_S^{\text{new}}) \cup E_{\overrightarrow{\text{virt}}})$ is acyclic.

Let $\mathcal{T} = (T, \chi)$ be a tree decomposition of DAG D. We call $\mathcal{T}^{\text{new}} = (T^{\text{new}}, \chi^{\text{new}})$ a *conservative* tree decomposition of DAG D^{new} with respect to set $S \subseteq T$ if

conservative

1. $\mathcal{T}^{\text{new}}$ is a tree decomposition of DAG D^{new},
2. T^{new} can be partitioned into $S^{\text{new}}, T_1, \dots, T_r$, where $T_1, \dots, T_r$ are the connected components of $T \setminus V(S)$, and
3. $\chi^{\text{new}}(t) = \chi(t)$ for $t \in \bigcup_{i=1}^{r} V(T_i)$.

Lemma 5.1. *If D_S^{new} is well-behaved, then D^{new} is acyclic and $f(D^{\text{new}}) \geq f(D)$.*

Proof. The correctness follows from Theorem 4.1, as the statement of the lemma is identical except the treewidth constraints. We note that any tree decomposition of such a D_S^{new} is conservative with respect to S. □

Introducing a state space bound to Lemma 5.1, we obtain the following theorem.

Theorem 5.2. *If D_S^{new} is well-behaved and admits a tree decomposition with* mssss *at most k, then*

1. *D^{new} is acyclic,*
2. *the score of D^{new} is at least the score of D, and*
3. $\text{msss}(D^{\text{new}}) \leq k$.

Proof. The first two properties follow from Lemma 5.1. Now, let $\mathcal{T} = (T, \chi)$ be the tree decomposition of DAG D, $\mathcal{T}^{\text{new}} = (T^{\text{new}}, \chi^{\text{new}})$ be the tree decomposition of DAG D^{new} and $\mathcal{S}^{\text{new}} = (S^{\text{new}}, \chi_S^{\text{new}})$ be the tree decomposition of D_S^{new}. Since $\mathcal{S}^{\text{new}}$ is conservative with respect to S (from Lemma 5.1), we can express χ^{new} as $\chi_S^{\text{new}}(t)$ if $t \in S$ and $\chi(t)$ otherwise. In order to prove the third property, we need to bound the state space size of the bags of $\mathcal{T}^{\text{new}}$, i.e., we need to bound

$$\begin{aligned}
\max_{t \in T^{\text{new}}} \prod\nolimits_{v \in \chi^{\text{new}}(t)} \text{ds}(v) &= \max\left(\max_{t \in S} \prod\nolimits_{v \in \chi_S^{\text{new}}(t)} \text{ds}(v), \max_{t \in T \setminus V(S)} \prod\nolimits_{v \in \chi(t)} \text{ds}(v) \right) \\
&\leq \max\left(k, \max_{t \in T} \prod\nolimits_{v \in \chi(t)} \text{ds}(v)\right) \qquad [\text{msss}(D_S^{\text{new}}) \leq k] \\
&\leq \max(k, k) \qquad [\text{msss}(D) \leq k].
\end{aligned}$$

Thus, the maximum state space size of D^{new} is at most k. □

5.3.3 MaxSAT Encoding

Φ_S

We now describe how we construct the (weighted, partial) MaxSAT instance that encodes the conditions required on the local instance S. The instance is a propositional formula in conjunctive normal form (CNF), with hard clauses and soft clauses; each soft clause has a weight. The MaxSAT solver tries to find a truth assignment that satisfies all the hard clauses and maximizes the sum of weights of satisfied soft clauses. We take as input the local instance S, the set of virtual edges E_{virt}, the set of virtual arcs $E^{\rightarrow}_{\text{virt}}$, and the bound k on the maximum state space size and produce a MaxSAT instance Φ_S. We reuse parts of the encoding from Chapter 4 (which we will refer to as the *BN-SLIM encoding*).

Φ'_S

Let $n = |S|$ denote the size of the subinstance. We now reiterate the hard clauses from the BN-SLIM encoding and refer to the conjunction of these clauses as the formula Φ'_S. For an explanation of the semantics of these variables and clauses, we refer to Section 4.3.2.

$$\begin{array}{rl}
\left.\begin{array}{r}(\mathrm{acyc}^*_{u,v} \land \mathrm{acyc}^*_{v,w}) \to \mathrm{acyc}^*_{u,w} \\ (\mathrm{ord}^*_{u,v} \land \mathrm{ord}^*_{v,w}) \to \mathrm{ord}^*_{u,w}\end{array}\right\} & \text{for distinct } u, v, w \in S. \\
\neg\mathrm{arc}_{v,v} & \text{for } v \in S. \\
\sum_{P \in \mathcal{P}_v} \mathrm{par}_v^P = 1 & \text{for } v \in S. \\
\mathrm{par}_v^P \to \mathrm{acyc}_{u,v} & \text{for } v \in S, P \in \mathcal{P}_v, \text{ and } u \in P. \\
\left.\begin{array}{r}(\mathrm{par}_v^P \land \mathrm{ord}_{u,v}) \to \mathrm{arc}_{u,v} \\ (\mathrm{par}_v^P \land \mathrm{ord}_{v,u}) \to \mathrm{arc}_{v,u}\end{array}\right\} & \text{for } v \in S, P \in \mathcal{P}_v, \text{ and } u \in P. \\
\left.\begin{array}{r}(\mathrm{par}_v^P \land \mathrm{ord}_{u,w}) \to \mathrm{arc}_{u,w} \\ (\mathrm{par}_v^P \land \mathrm{ord}_{w,u}) \to \mathrm{arc}_{w,u}\end{array}\right\} & \text{for } v \in S, P \in \mathcal{P}_v, \text{ and } u, w \in P. \\
\left.\begin{array}{r}(\mathrm{arc}_{u,v} \land \mathrm{arc}_{u,w} \land \mathrm{ord}_{v,w}) \to \mathrm{arc}_{v,w} \\ (\mathrm{arc}_{u,v} \land \mathrm{arc}_{u,w} \land \mathrm{ord}_{w,v}) \to \mathrm{arc}_{w,v}\end{array}\right\} & \text{for } u, v, w \in S. \\
\neg\mathrm{arc}_{u,v} \lor \neg\mathrm{arc}_{v,u} & \text{for } u, v \in S. \\
\mathrm{ord}^*_{u,v} \to \mathrm{arc}_{u,v} \land \mathrm{ord}^*_{v,u} \to \mathrm{arc}_{v,u} & \text{for } \{u, v\} \in E_{\text{virt}}. \\
\mathrm{par}_v^P \to \mathrm{acyc}^*_{u,v} & \text{for } v \in S, P \in \mathcal{P}_v, \text{ and } (u, v) \in A^{\rightarrow}_{\text{virt}}(v, P).
\end{array}$$

We also add the following unit soft clauses to Φ'_S, setting $f'(P, v) = f(P, v) - f(\emptyset, v)$ as their weight

$$(\mathrm{par}_v^P) : \text{weight } f'(P, v) \quad \text{for } v \in S, P \in \mathcal{P}_v.$$

The weight of a solution to Φ'_S is given by the sum of weights of the satisfied soft clauses. Let $W = \sum_{v \in S} f'(v, P_D(v))$ be the core of the unmodified local instance S.

Lemma 5.3. *Φ'_S admits a solution of weight W^{new} if and only if there exists a well-behaved DAG D_S^{new} with respect to D, such that $f(D^{\text{new}}) - f(D) = W^{\text{new}} - W$.*

Proof. We refer to Theorem 4.2 to establish the lemma, since the only difference is the clauses that bound treewidth of D_S^{new} from above. Therefore D_S^{new} is still acyclic. Further, an assignment that corresponds to setting $D_S^{\text{new}} = S_S$ achieves the lower bound on the score. □

5.3.4 BDD-based Counter

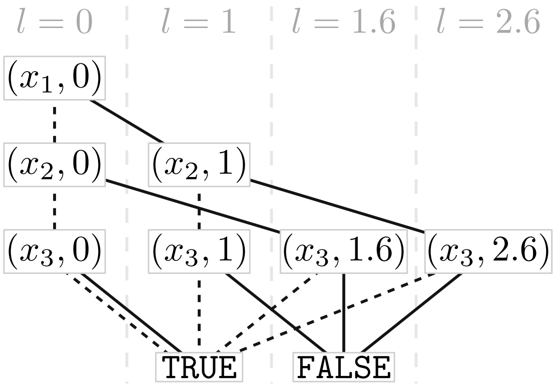


Figure 5.3: Example BDD constructed for three variables v_1, v_2, v_3 with domain sizes $2, 3, 4$, respectively, and with $k = 6$. The logarithms of the domain sizes are $1, 1.6, 2$, respectively. The solid and dashed edges represent the 'if' and 'else' arcs respectively. The path $(x_1, 0)$ — $(x_2, 1)$ — $(x_3, 2.6)$ - - TRUE represents the bag containing x_1 and x_2 with a state space size of $6 \leq k$, hence reaching TRUE. The path $(x_1, 0)$ — $(x_2, 1)$ - - $(x_3, 1)$ — FALSE represents the bag containing x_1 and x_3 with a state space size of $8 > k$, hence reaching FALSE. Note that both these bags have cardinality 2 and hence are treated the same when it comes to treewidth.

In this section, we elaborate the counting mechanism that we use to encode the condition

$$\sum_{w \in S, w \neq v, \text{arc}_{v,w}} \log(\text{ds}(w)) \leq \log(k) \quad \text{for } v \in S.$$

The technique we use was first proposed by Eén and Sörensson [ES06]. Intuitively speaking, we construct an (Ordered) Binary Decision Diagram (BDD), where following a path from an input node to a terminal node corresponds to summing up weights. Each layer of nodes in the BDD is associated with a variable $v \in S$, and there are two outgoing edges from each node in the BDD corresponding to the existence and absence of $\text{arc}_{v,w}$.

More formally, let us assume that the set S consists of the variables $v_1, \ldots, v_m$, where $m := |S|$. We construct a BDD, which is a directed graph with nodes of the form (x_i, l) where $i \in \{1, d \ldots m\}$ is an integer, and $l \in [0, k]$ is a real number. Here, x_i signifies that we branch on variable v_i next, and l denotes the level of the current sum of weights. Additionally, there are two special terminals (or sink nodes) — TRUE and FALSE. From each node (x_i, l) there are two outgoing arcs — the 'if' arc connecting it to the node $(x_{i+1}, l + \log(\mathrm{ds}(v_i)))$, and the 'else' arc connecting it to the node (x_{i+1}, l). In case $l + \log(\mathrm{ds}(v_i)) > k$, we instead connect it to the `FALSE` terminal. Finally, for a node of the form (x_m, l), we connect it to the `TRUE` terminal iff $l + \log(\mathrm{ds}(v_m)) \leq k$. We denote by $\mathrm{if}(x, l)$ and $\mathrm{else}(x, l)$, the nodes connected to (x, l) via the 'if' and 'else' arcs, respectively. We refer to Figure 5.3 for an example. There are at most $2^m + 1$ nodes in the BDD.

For each variable $v \in S$, we have one such BDD to ensure that the product of the domain sizes of the endpoints of the outgoing arcs does not exceed the bound k. It is straightforward to express this BDD in the form of a CNF formula $\mathcal{B}_v$. We introduce a variable $\mathrm{bdd}^v_{w,l}$ for each $w \in S \setminus \{v\}$ and each level l in the BDD. We then add the following clauses to $\mathcal{B}^v$

$$\left.\begin{array}{r}(\mathrm{bdd}^v_{w,l} \wedge \mathrm{arc}_{v,w}) \rightarrow \mathrm{bdd}^v_{\mathrm{if}(w,l)} \\ (\mathrm{bdd}^v_{w,l} \wedge \neg\mathrm{arc}_{v,w}) \rightarrow \mathrm{bdd}^v_{\mathrm{else}(w,l)}\end{array}\right\} \quad \text{for } (w, l) \in V(\mathcal{B}_v).$$

Next, we add the clause $\neg\mathrm{bdd}^v_{\mathtt{FALSE}}$ to $\mathcal{B}^v$ that falsifies $\mathcal{B}^v$ if the `FALSE` sink node is reached, i.e., if the outgoing arcs of variable v violate the bound. Finally, we conjoin all the formulas $\mathcal{B}^v$ for $v \in S$ with the formula Φ'_S to obtain our final formula Φ_S, giving us the following theorem.

Theorem 5.4. *Φ_S admits a solution of weight K^{new} if and only if there exists a well-behaved DAG D^{new}_S with respect to D, such that $f(D^{\mathrm{new}}) - f(D) = K^{\mathrm{new}} - K$ and $\mathrm{msss}(D^{\mathrm{new}}_S) \leq k$.*

Proof. By construction of the BDD, we see that $\mathcal{B}^v$ is falsified if and only if the outgoing arcs from variable v result in the state space size of that bag exceeding k. Thus, $\bigwedge_{v \in S} \mathcal{B}^v$ is satisfiable if and only if $\mathrm{msss}(D^{\mathrm{new}}_S) \leq k$. Combining this with Lemma 5.3, we obtain the desired result. □

Finally, we would like to draw attention to the fact that since only the local part is encoded into a MaxSAT formula, increasing the total number of variables does not directly affect the solving time for each individual subinstance, and consequently, the time required for each individual improvement. Thus, the overhead of the BDDs doesn't increase when the total number of variables in the network increases. This is a crucial strength of the SLIM approach in general, i.e., it scales well relative to the total number of variables, and the runtime has a stronger dependence on the budget.

5.4 Experimental Evaluation

5.4.1 Setup

We tested the various proposed methods on 4-core Intel Xeon E5540 2.53 GHz CPU (internal cluster), with each process having access to 8 GB RAM. The k-greedy and k-MAX algorithms are available as a part of the BLIP package [Sca15] implemented in Java. We modified and extended these to obtain the implementations for k-greedy$^{\text{bss}}$ and k-MAX$^{\text{bss}}$. We implemented BN-SLIM$^{\text{bss}}$in Python making use of the NetworkX library [HSS08]. We used UWrMaxSat[1] as the MaxSAT-solver, because of its reliable response to timeouts. For evaluating the reasoning time we used the Merlin package by Radu Marinescu.[2] We provide the source code as a public GitHub repository [PS21a].

We tested the algorithms on a subset of the `bnlearn` repository.[3] These networks are commonly used as benchmarks in the literature. Out of the 22 networks available in the repository, we only consider the 16 networks that contain non-binary variables. These networks range in size from 6 to 1041 random variables. Due to the smaller networks' behavior being susceptible to random noise, we focus more on the larger networks.

5.4.2 Method

Next, we describe the method used to evaluate the performance of the BNSL algorithms enumerated in Section 5.3.1. We run the algorithms for a total time of 90 minutes and record the reasoning time and score at the end. We denote by BN-SLIM$^{\text{bss}}$(X) the algorithm composed of running the heuristic X for 30 minutes and then running the bounded state space local improvement algorithm on top of the heuristic solution for another 60 minutes. We run the algorithms with multiple bounds and multiple random seeds. Finally, for each instance, we visualize the distribution of scores and reasoning times on a scatter plot to capture the trade-off achieved between the two metrics. We set k-greedy and k-MAX as the *baseline* algorithms and always compare the newly proposed algorithms against these baseline algorithms. The baseline algorithms represent the state-of-the-art bounded treewidth BNSL methods while the heuristics represent the bounded state space based BNSL methods.

5.4.3 Results

We performed some preliminary tests to determine the most promising bounded state space (bss) methods for comparison against the baseline of tw-methods k-MAX and k-greedy. On the basis of these tests, we choose BN-SLIM$^{\text{bss}}$(k-MAX$^{\text{bss}}$) as the bss method. In Figure 5.4 we see the scatter plots for a representative subset of instances. The plots are presented in pairs with the left subplot depicting the tw methods' performance and the right subplot depicting BN-SLIM$^{\text{bss}}$(k-MAX$^{\text{bss}}$)'s performance.

In general, from Figure 5.4, we observe that BN-SLIM$^{\text{bss}}$(k-MAX$^{\text{bss}}$) achieves much faster reasoning times at the expense of slightly worse scores in some cases. Whereas in some cases, BN-SLIM$^{\text{bss}}$(k-MAX$^{\text{bss}}$) manages to match the score while reducing the reasoning time significantly. In most cases, we observe a reduction by an order of magnitude. Another point worth noting is that the clustering of the BNs output by

[1] https://maxsat-evaluations.github.io/2019/descriptions.html

[2] https://github.com/radum2275/merlin

[3] https://www.bnlearn.com/bnrepository/

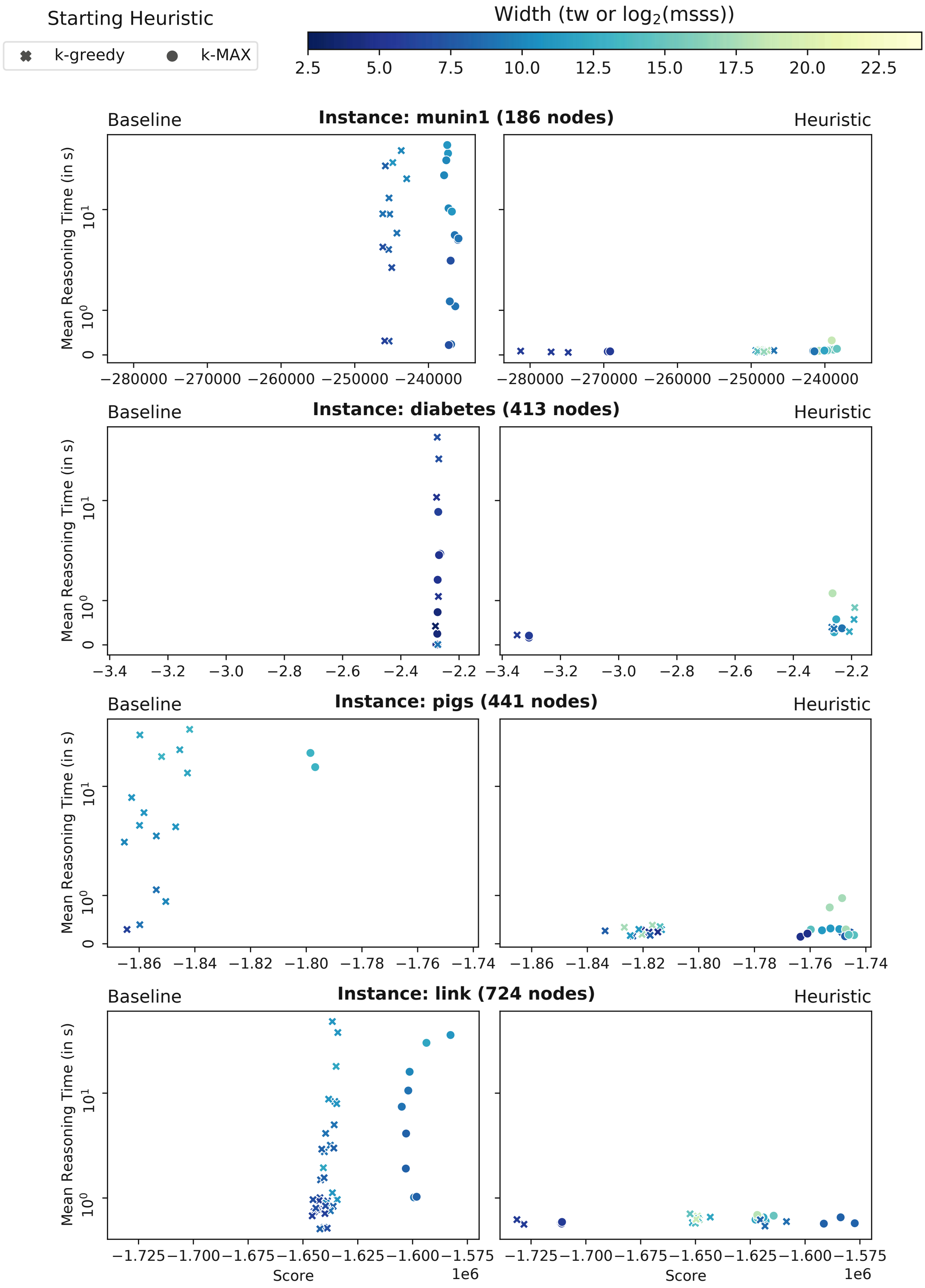


Figure 5.4: Scatter plots comparing $X \in \{\text{k-greedy}, \text{k-MAX}\}$ with BN-SLIM$^{\text{bss}}(X)$

BN-SLIM$^{\text{bss}}$(k-MAX$^{\text{bss}}$) along the reasoning time axis is much tighter than with the tw methods. This highlights the reliability aspect of the bss methods.

We also observe that the bss method BN-SLIM$^{\text{bss}}$(k-MAX$^{\text{bss}}$) allows us to expand the search space much more carefully and predictably. For instance, consider a bound on the maximum state space size of 5×10^5. A reasonable equivalent bound on the treewidth would be $\log_2(5 \times 10^5) \approx 19$. However, networks with treewidth 19 can be expected to have much worse inference speeds as compared to networks with msss $\leq 5 \times 10^5$.

This confirms our hypothesis that maximum state space size is a much better estimator of reasoning time as compared to treewidth, and that one can construct algorithms like BN-SLIM$^{\text{bss}}$(k-MAX$^{\text{bss}}$) to learn such fast-inference BNs.

5.5 Conclusion

We have introduced the concept of bounded state space BNSL, devised it theoretically, implemented it, and tested it rigorously on a set of real-world data sets. We compared the new approach with state-of-the-art bounded treewidth BNSL algorithms on a two-dimensional setting to see the trade-off between inference speed and data fitting. Our results show that the new approach indeed provides overall better and more reliable results. In some cases, the advantage of bounded state space over bounded treewidth methods is significant.

CHAPTER 6

BNSL with Expert Constraints

> *Science and everyday life cannot and should not be separated.*
>
> — Rosalind Franklin, "Rosalind Franklin: The Dark Lady of DNA"

Published as "Learning large Bayesian networks with expert constraints". Peruvemba Ramaswamy and Szeider. In: Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, (Aug. 2022) [PS22]

In the final stop of our journey, we revisit Bounded Treewidth BNSL, but now with an added twist. We introduce so-called *expert constraints* which are additional requirements that the learned BN must satisfy. These requirements are expressed in terms of arcs or ancestries that must be present or absent in the learned BN. Some examples of such constraints are shown in Table 6.1. For each constraint, we also show the restriction that it imposes on the learned BN and also the underlying real-world causal effect identified by the expert that is being modelled by that constraint. From these constraints, Constraints A and B are satisfied by the DAG shown in Figure 3.1 while Constraint C is violated. These constraints are typically provided by an expert based on their domain knowledge and their understanding of the causality of the involved RVs.

The area of Causality was pioneered by Pearl [Pea88] who was concerned that predictive models of the time relied solely on correlation. He introduced causality as a means to make models that are more 'intentional' in terms of their represented dependencies. Models learned from causal data are more trustworthy as compared to models learned from purely correlational data. In this chapter, we consider the problem of learning bounded treewidth BNs with thousands of RVs which also satisfy additional expert constraints. We

	Constraint	Restriction on BN	Interpretation of Expert
A.	`income` $\to$ `job`	there must be an arc from `income` to `job`	`income` has a (direct) causal effect on `job`
B.	`study` $\rightsquigarrow$ `masters`	there must be a path from `study` to `masters`	`study` has a (possibly indirect) causal effect on `masters`
C.	`income` $\not\rightsquigarrow$ `grade`	there can be no path from `income` to `grade`	`income` has no (direct or indirect) causal effect on `grade`

Table 6.1: Examples of expert constraints

develop a SLIM approach for this problem by significantly upgrading both the k-greedy heuristic algorithm and the Bounded Treewidth BNSL MaxSAT encoding.

6.1 Introduction

We are already well-acquainted with the structure learning problem for BNs with the additional requirement of bounded treewidth (from Chapter 4). Another fundamental requirement receiving a growing amount of attention is to learn BNs that fit the data and satisfy additional *expert constraints* [Che+16; KKN01; LB18; Cor+13]. Such constraints can assert, for instance, direct or indirect causation between random variables in terms of whether one variable is a parent or an ancestor of the other in the DAG of the learned BN. Please see Table 6.2 for a list of expert constraints considered in the literature.

	Arc constraints (direct causation)
$u \to v$	the DAG contains the arc (u, v)
$u \not\to v$	the DAG does not contain the arc (u, v)
$u \leftrightarrow v$	the DAG contains either the arc (u, v) or the arc (v, u)
	Ancestry constraints (indirect causation)
$u \rightsquigarrow v$	the DAG contains a path from u to v
$u \not\rightsquigarrow v$	the DAG does not contain a path from u to v
$u \leftrightsquigarrow v$	the DAG contains either a path from u to v or one in the other direction

Table 6.2: Various expert or side constraints considered in literature.

$\to, \not\to, \leftrightarrow, \rightsquigarrow, \not\rightsquigarrow$

In addition to bounded treewidth and expert constraint requirements, one must address the *scalability* of methods for BNSL. For instance, learning a BN of bounded treewidth

	Scalability (# RVs)	Bounded treewidth	Supported constraints	Score optimization
EC Tree [Che+16]	≤ 20	no	$\{\rightsquigarrow, \not\rightsquigarrow\}$	exact
MINOBSx [LB18]	≤ 50	no	$\{\not\rightarrow, \not\rightsquigarrow, \rightarrow, \leftrightarrow, \rightsquigarrow\}$	approx.
CaMML [KKN01]	unknown	no	$\{\not\rightsquigarrow, \rightarrow, \leftrightarrow, \rightsquigarrow\}^{\dagger}$	exact
k-greedy [Sca+18]	≤ 10000	yes	$\varnothing$	approx.
BN-SLIM (Chapter 4)	≤ 10000	yes	$\varnothing$	approx.
Con-k-greedy (here)	≤ 10000	yes	$\{\not\rightarrow, \not\rightsquigarrow, \rightarrow, \leftrightarrow, \rightsquigarrow\}^{\ddagger}$	approx.
Con-BN-SLIM (here)	≤ 10000	yes	$\{\not\rightarrow, \not\rightsquigarrow, \rightarrow, \leftrightarrow, \rightsquigarrow\}^{\ddagger}$	approx.

Table 6.3: Table comparing relevant features of various methods. † CaMML allows weighted constraints with the weight of 1 signifying hard constraints. ‡ Negative constraints are treated as hard constraints, while positive constraints can be violated.

that optimally fits the data is NP-hard [KP13]. The consideration of expert constraints provides an additional source of complexity.

In this chapter, we propose Con-BN-SLIM (Constrained BN-SLIM), the first method for BNSL that addresses all three requirements simultaneously: bounded treewidth, expert constraints, and scalability. Table 6.3 shows how our new method compares to other BN structure learning methods from the literature. Since these methods span decades of research, it was a natural choice to try to reuse their progress as much as possible so as to stand on the shoulders of giants. Thus, we arrived at our 2-phase approach of Con-BN-SLIM, which leverages the scalability of k-greedy and the localized optimization power of BN-SLIM (particularly useful for expert constraints).

In Phase 1, a heuristic algorithm greedily computes a candidate BN from data, thereby trying to satisfy as many expert constraints as possible. The heuristic algorithm is a version of the k-greedy algorithm by Scanagatta et al. [Sca+18] that we modified to consider expert constraints. This method scales very well. However, considering expert constraints significantly deteriorates the algorithm's capability of fitting the BN to the data. This even prevails when we consider the expert constraints as soft constraints, which allows the algorithm to violate some constraints.

We, therefore, add a Phase 2 that takes the candidate BN from the first phase and repeatedly tries to improve the score by optimizing local parts of the BN. The second phase is an extension of the BN-SLIM approach from Chapter 4. BN-SLIM utilizes a MaxSAT solver to locally improve the BN. Crucial for our extension is to express suitable local versions of the desired expert constraints in terms of hard constraints for the MaxSAT solver. This way, the solver may improve the fitting of the BN while maintaining the satisfaction of all the expert constraints satisfied by the first phase solution.

Due to our novel contributions in Section 6.4, like localization of global constraints and the scaffolding of auxiliary variables required to express and incorporate expert constraints into BN-SLIM, the proposed approach is more than just gluing together existing methods.

We evaluated a prototype implementation of Con-BN-SLIM on all discrete sample data from the bnlearn BN repository, sampling expert constraints from the ground truth networks. After the first phase of running the modified heuristic algorithm for about 30 minutes, the rate of improvement deteriorates. Phase 2 begins, and Con-BN-SLIM takes over the candidate network and shows a remarkably high improvement rate. The final network shows a significantly higher score than the one produced by Phase 1, which displays favorably in the ΔBIC metric.

The empirical findings on our prototype implementation are highly encouraging, providing the ground for several avenues of further investigation.

6.2 Additional Background

Since most of the required notation and methodology has already been discussed in Chapter 3, in this section, we only discuss the minimal additional background required.

constraint set

path avoiding a set

In our work, we consider only arc and ancestry constraints. The requirements for satisfaction of the constraints is described in Table 6.2. We use the term *constraint set* to refer to a set of such constraints and a DAG D is said to satisfy a constraint set if it satisfies all constituent constraints. We refer to $\rightarrow$ and $\rightsquigarrow$ as *positive* constraints and $\not\rightarrow$ and $\not\rightsquigarrow$ as *negative* constraints. Note that, $u \not\rightsquigarrow v$ is denoted as $v > u$ by Li and Beek [LB18]. Also note that, some other variants of constraints like $\not\leftrightsquigarrow$ can be expressed as boolean combinations of the elementary constraints from Table 6.2. The path P *avoids* a set $S \subseteq V$ if $v_i \notin S$ for all $1 \leq i < \ell$.

partial order

obey a partial order

We also use the concept of partial orders in our modification of k-greedy (Section 6.3). A *partial order* is a set of pairwise ordering requirements $u \triangleright v$. A linear order $u_1, \ldots, u_n$ is said to *obey* a partial order if, for every $u_i \triangleright u_j$ in the partial order, $i < j$.

6.3 k-greedy with Constraints

In this section, we describe the modifications made to k-greedy to obtain a heuristic algorithm to solve the Constrained BNSL problem. We would like to point out that we chose to modify k-greedy as a proof of concept because of its simplicity. However, theoretically similar modifications are also possible for the more aggressive k-MAX heuristic [Sca+18].

6.3.1 Overview of k-greedy

First, we briefly recall the basic k-greedy heuristic by Scanagatta et al. [Sca+16]. The algorithm takes as input a set X of RVs and a score function cache and returns a DAG D

along with a corresponding (rooted) tree decomposition T. The algorithm repeatedly performs the following steps:

Step 1: Randomly sample a linear ordering σ over the variables X

Step 2: Construct the root bag of T from the first $k+1$ variables of σ. Also, compute a DAG over these variables maximizing the score (either exactly or approximately).

Step 3: Then insert the remaining variables from σ one by one into the DAG, selecting the best parent set for it from the already inserted variables.

After each step, if the newly computed DAG has a higher score than the previous best DAG, it is called an *improvement*.

6.3.2 Modified k-greedy

To upgrade this algorithm to work with expert constraints, we modify each of the steps above to obtain Con-k-greedy (Constrained k-greedy). Algorithm 6.2 shows the pseudocode for Con-k-greedy. In *Step 1*, instead of randomly sampling an order, we first 'compile' the supplied constraints $\mathcal{C}$ into a partial order $\mathcal{P}$. Meaning, we add a partial order pair $u \triangleright v$ to $\mathcal{P}$ for every positive constraint $u \bowtie v$, i.e., $\bowtie \in \{\rightarrow, \rightsquigarrow\}$. This is because it can be easily shown that all topological orderings of all networks that satisfy constraints $\mathcal{C}$ also obey the partial order $\mathcal{P}$. We then randomly sample linear orderings that obey this partial order, which serve as both elimination orderings and topological orderings for the DAG being constructed.

Algorithm 6.1: Pseudocode for `Compile`

```
Input  : Set $\mathcal{C}$ of expert constraints
Output : Set $\mathcal{P}$ of partial order pairs
begin
    $\mathcal{P} \longleftarrow \emptyset$
    foreach $u \bowtie v \in \mathcal{C}$ do
        if $\bowtie \in \{\rightarrow, \rightsquigarrow\}$ then
            $\mathcal{P} \longleftarrow \mathcal{P} \cup \{u \triangleright v\}$
        end
    end
    return $\mathcal{P}$
end
```

In *Step 2*, we now search for a best DAG that does not violate any negative constraint by brute force or using any other local solver (Line 7). In *Step 3*, we select the best parent set from among the parent sets that violates none of the negative constraints and satisfies all the positive constraints involving the currently inserted variable. If there are

Algorithm 6.2: Pseudocode for Con-k-greedy

```
Input  : Score function $f$, set $\mathcal{C}$ of expert constraints, treewidth bound $k$
Output : DAG $D$ satisfying all negative constraints necessarily and positive
         constraints optionally
1 :begin
2:    $\mathcal{P} \longleftarrow$ Compile($\mathcal{C}$)
3:    loop
4:        Sample linear order $\sigma$ obeying $\mathcal{P}$
5:        Construct root bag $B_0 \longleftarrow \{\sigma_0, \ldots, \sigma_{k+1}\}$
7:        Construct a DAG $D$ over $B_0$ maximizing score and not violating any
          negative constraints
8:        for $v$ in $\sigma_{k+2}, \ldots, \sigma_n$ do
9:            $R \longleftarrow$ set of parent sets of $v$ not violating any negative constraints and
              satisfying all positive constraints of the form $u \bowtie v$ for some $u$
10:           if $R$ is nonempty then
11:               $P_D(v) \longleftarrow$ maximum score parent set from $R$
12:           else
14:               $P_D(v) \longleftarrow \emptyset$
15:           end
16:       end
17:       if algorithm terminated then
18:           return $D$
19:       end
20:   end
21 :end
```

no such parent sets, we simply select the empty parent set for the current variable (see Line 14); this ensures that no negative constraints are violated.

This results in an algorithm that can keep generating better and better scoring DAGs with the condition that all generated DAGs respect the negative constraints from $\mathcal{C}$ as hard constraints and the positive constraints as soft constraints.

6.3.3 Practical considerations

Theoretically, it is possible to modify k-greedy similarly so that the resultant algorithm treats all constraints as hard constraints. However, in practice, we noticed that this severely limits the number of improvements and, in many cases, fails to find any networks. We, thus, slightly alter *Step 3* to only reject choices of parent sets that violate the negative constraints, i.e., $\{\not\leftrightarrow, \not\rightarrow\}$. As a result, the heuristic provides solutions which satisfy all the negative constraints but not necessarily all the positive and undirected constraints. In other words, all positive and undirected constraints, i.e., $\{\rightarrow, \leftrightarrow, \rightsquigarrow\}$, are treated as *soft* constraints.

6.3.4 Experimental Evaluation

We experimentally evaluated the heuristic proposed above and found the results unsatisfactory. We noticed that the rate of improvement diminishes quite quickly and essentially reaches saturation by 30 mins (see Figure 6.1). However, the output of the heuristic could serve as a starting point for further improvement. The *SAT-based Local Improvement Method* (SLIM) framework could potentially turbocharge and improve the score of such an intermediate saturated solution. In the next sections, we develop a solution using the SLIM framework for the Constrained BNSL problem.

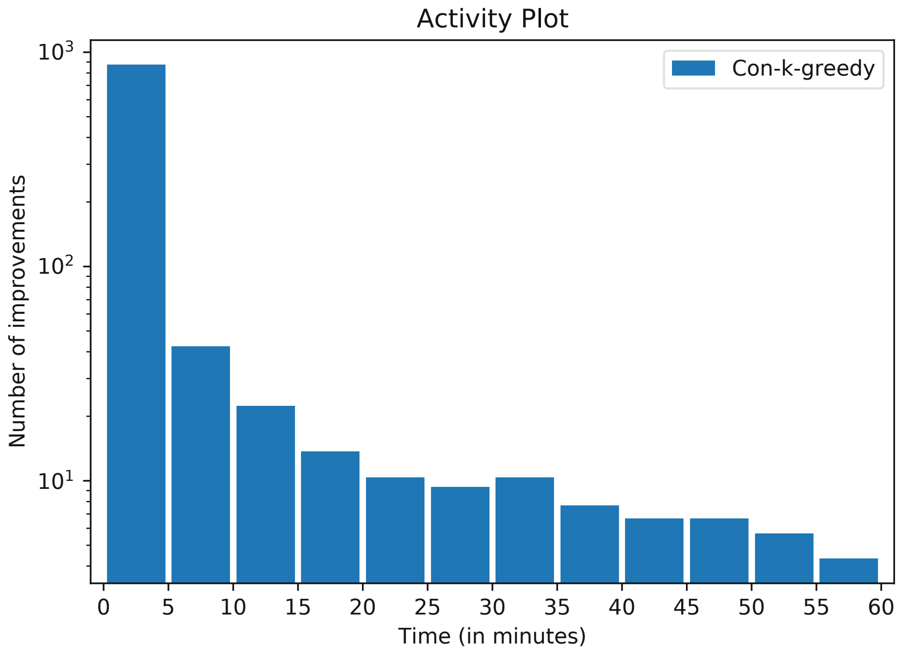


Figure 6.1: Activity plot showing the rate of improvements of Con-k-greedy against time. Note that the y-axis is in log scale.

6.4 BN-SLIM with Constraints

6.4.1 Theory

In this section, we lay the theoretical foundation for solving the Constrained BNSL problem using the SLIM framework. The SLIM framework has been previously demonstrated in Chapter 4 to solve the BNSL problem. We refer to this method as BN-SLIM. We directly extend BN-SLIM to solve the Constrained BNSL problem; as a result we reuse the same notation.

The problem input consists of a set V of random variables, a score function f, a treewidth bound W and a set $\mathcal{C}$ of expert constraints, and each constraint is of the following form

$$u \bowtie v, \quad \text{where } u \neq v \in V \text{ and } \bowtie \in \{\rightarrow, \nrightarrow, \leftrightarrow, \rightsquigarrow, \not\rightsquigarrow\}$$

The goal is to compute a DAG $D^\star$ over V with maximum score such that the treewidth of the moralized graph $M(D^\star)$ is bounded by W and $D^\star$ satisfies all the constraints in $\mathcal{C}$. We assume to have an initial heuristic solution D, a corresponding tree decomposition $\mathcal{T} = (T, \chi)$ of width $\leq W$ of the moralized graph $M(D)$ and that D satisfies the constraint set $\mathcal{C}$. Our aim now, is to compute a DAG D^{new} over V with score at least as much as D while still having bounded treewidth and satisfying constraint set $\mathcal{C}$. Applying this process repeatedly, we can improve the score of the resultant DAG while still satisfying all the requirements.

S
D_S^{new}
$\mathcal{S}^{\text{new}}$
S^{new}, χ^{new}

For convenience, we recall the notation from Section 4.2. We select a subtree $S \subseteq T$ such that the total number of vertices in $V_S := \bigcup_{t \in S} \chi(t)$ is at most some *budget* β. β is a fixed constant limiting the size of the local instances such that instances of this size can be solved reasonably quickly by the local solver. The value of β is decided by means of experimenting and educated guesses. We define D_S^{new} as the DAG induced by D^{new} on V_S, where $E(D_S^{\text{new}}) = \{\, (u,v) \in E(D^{\text{new}}) : \{u,v\} \subseteq V_S \,\}$ and $\mathcal{S}^{\text{new}} = (S^{\text{new}}, \chi^{\text{new}})$ as a tree decomposition of D_S^{new}. For convenience, we use the shorthand E_S^{new} to denote $E(D_S^{\text{new}})$.

We distinguish between different kinds of vertices:

boundary vertex
internal vertex
external vertex

- $v \in V_S$ a *boundary vertex* if there exists a tree node $t \in V(T) \setminus V(S)$ such that $v \in \chi(t)$;
- $v \in V_S$ is an *internal vertex* if v is not a boundary vertex;
- $v \in V \setminus V_S$ is an *external vertex*.

adjacent boundary vertices
virtual edge
E_{virt}
extended moral graph
M_{ext}
virtual arc
$E_{\text{virt}}^{\rightarrow}$

Two boundary vertices v, v' are *adjacent* if both occur together in some bag outside S. In that case we call $\{v, v'\}$ a *virtual edge*edge!virtualvirtual edge. We let E_{virt} be the set of all virtual edges. The *extended moral graph* M_{ext} is obtained from $M(D_S^{\text{new}})$ by adding all virtual edges. If v, v' are two adjacent boundary vertices such that D^{new} contains a directed path from v' to v, where all the vertices on the path, except for v' and v, are external, then (v', v) is a *virtual arc*arc!virtualvirtual arc. $E_{\text{virt}}^{\rightarrow}$ denotes the set of all virtual arcs.

We now collect and reiterate the conditions from Section 4.2 needed to state the main theorem of this chapter.

C1 D_S^{new} is acyclic.

C2 The moral graph $M(D_S^{\text{new}})$ has treewidth $\leq W$.

C3 $\mathcal{S}^{\text{new}}$ is a tree decomposition of the extended moral graph M_{ext}.

C4 For each $v \in V_S$, if $P_{D^{\text{new}}}(v)$ contains external vertices, then there is some $t \in V(T) \setminus V(S)$ such that $P_{D^{\text{new}}}(v) \cup \{v\} \subseteq \chi(t)$.

C5 The digraph $(V_S, E_S^{\text{new}} \cup E_{\text{virt}}^{\rightarrow})$ is acyclic.

Theorem 6.1 (Recalling Theorem 4.1)**.** *If all the conditions C1–C5 are satisfied, then D^{new} is acyclic, the treewidth of $M(D^{\text{new}})$ is at most W, and the score of D^{new} is at least the score of D.*

Now, we discuss how the different types of constraints can be transformed into their respective local versions along with the correctness for the same. We note that the input constraint set can only consist of elementary arc and ancestry constraints (listed in Table 6.2); however, the translation into their respective local versions additionally allows disjunctions over elementary constraints. This is because the local versions of the constraints are directly handed off to the MaxSAT solver which is capable of handling such disjunctions.

To discuss the behavior of the ancestry constraints, we use the concept of first-hit descendants and first-hit ancestors. Given a DAG F over vertices W, subset $Y \subsetneq W$ and vertex $r \in W$, a node $s \in Y$ is said to be a *first-hit descendant* of r in Y if there exists a directed path from r to s avoiding all the other vertices in $Y \setminus \{r, s\}$. We denote by $\text{desc}_Y^r \subseteq Y$, the set of all first-hit descendants of r in Y. Similarly, a node $s \in Y$ is said to be a *first-hit ancestor* of r in Y if there exists a directed path from s to r avoiding all the other vertices in $Y \setminus \{r, s\}$. We denote by $\text{anc}_Y^r \subseteq Y$, the set of all first-hit ancestors of r in Y. We denote by $\top$, the always-true trivial constraint that is always satisfied.

first-hit descendant
desc
first-hit ancestor
anc
$\top$

Arc Constraints ($\rightarrow, \nrightarrow$, or $\leftrightarrow$)

Let c be a constraint $u \bowtie v$, where $\bowtie \in \{\rightarrow, \nrightarrow, \leftrightarrow\}$. If either of $u, v \notin S$, then the constraint remains satisfied, since the presence or absence of the arc between u, v is not affected by D^{new}. The local version of such a constraint is thus $\top$. Alternatively, if both $u, v \in S$, it suffices to ensure that the constraint c holds in D_S^{new}. The local version of such a constraint is c itself.

Positive Ancestry Constraints ($\rightsquigarrow$)

Consider a constraint of the form $u \rightsquigarrow v$. Since the constraint is satisfied in D, we know that there exists a $u - v$ path in D.

Case 1 There is at least one $u - v$ path avoiding V_S. The constraint remains satisfied independent of D_S^{new}. The local version of such a constraint is $\top$.

Case 2 All $u - v$ paths pass through V_S. It suffices to ensure that there exists at least one path in D_S^{new} from some $d_u \in \text{desc}_{V_S}^u$ to some $a_v \in \text{anc}_{V_S}^v$. The local version of such a constraint is $\bigvee_{d_u, a_v} d_u \rightsquigarrow a_v$.

Negative Ancestry Constraints ($\not\rightsquigarrow$)

Consider a constraint of the form $u \not\rightsquigarrow v$. Since the constraint is satisfied in D, we know that there are no $u - v$ paths in D. Any $u - v$ path passing through V_S must be of the form $u - d_u - a_v - v$, for some $d_u \in \text{desc}_{V_S}^u, a_v \in \text{anc}_{V_S}^v$.

Case 1 $\mathrm{desc}^u_{V_S} = \emptyset$ or $\mathrm{anc}^v_{V_S} = \emptyset$. The constraint remains satisfied independent of D^{new}_S. The local version of such a constraint is $\top$.

Case 2 Both sets are non-empty. It suffices to ensure that there is no path in D^{new}_S from any $d_u \in \mathrm{desc}^u_{V_S}$ to any $a_v \in \mathrm{anc}^v_{V_S}$. The local version of such a constraint is $\bigwedge_{d_u,a_v} d_u \not\rightsquigarrow a_v$.

From this discussion, we can assert the following lemma.

Lemma 6.2. *If D^{new}_S satisfies the local versions of each of the constraint in $\mathcal{C}$, then D^{new} satisfies the constraint set $\mathcal{C}$.* □

From Theorem 6.1 and Lemma 6.2, we obtain the following corollary.

Corollary 6.2.1. *If conditions C1–C5 are satisfied and D^{new}_S satisfies the local versions of the constraints in $\mathcal{C}$, then D^{new} is acyclic, the treewidth of $M(D^{\mathrm{new}})$ is at most W, the score of D^{new} is at least that of D and D^{new} satisfies the constraint set C.*

6.4.2 Encoding

In this section, we describe the MaxSAT encoding to compute D^{new}_S. We build on top of the encoding from Section 4.3.2. We recall briefly, the basic variables in the encoding are the par^P_v variables, which are true if and only if P is the parent set of v. These variables appear in the encoding as soft clauses weighted by $f(v, P)$. In addition to that, there are several hard clauses involving $\mathrm{arc}_{u,v}$, $\mathrm{acyc}_{u,v}$ and $\mathrm{ord}_{u,v}$ variables, which encode the edges of the moralized graph, the acyclicity of the DAG and the elimination ordering corresponding to the tree decomposition respectively. The soft and hard clauses of the encoding are passed to the MaxSAT solver to optimize the network's score. The MaxSAT solver then finds solutions satisfying all the hard clauses while also maximizing the weight of the satisfied soft clauses. Eventually, this encoding finds a network with maximum score that satisfies the conditions C1–C5. For the sake of brevity, we skip repeating the entire encoding and only describe the additions.

Now, having Corollary 6.2.1, we describe the addition to the encoding that ensures that D^{new}_S satisfies the local versions of the constraints.

Arc Constraints ($\rightarrow, \not\rightarrow, \leftrightarrow$)

We filter out the infeasible parent sets based on the arc constraints. More specifically, for the constraint $u \rightarrow v$, we discard all parent sets of v that do not contain u, and conversely, for the constraint $u \not\rightarrow v$, we discard all parent sets that contain u.

Ancestry Constraints ($\rightsquigarrow, \not\rightsquigarrow$)

We address the ancestry constraints by introducing the following variables to keep track of the paths within the network:

1. $\mathrm{dagarc}_{u,v}$ represents an arc in the DAG from u to v (does not include the moralized and fill-in edges unlike $\mathrm{arc}_{u,v}$),

2. $\mathrm{tarc}_{u,v}$ captures the transitive closure of the $\mathrm{dagarc}_{u,v}$ variables.

3. $\mathrm{path}_{u,v}$ implies the existence of a path in the DAG from u to v,

4. $\mathrm{pathq}_{u,v,z}$ is a helper variable for $\mathrm{path}_{u,v}$ and implies the existence of a path in the DAG from u to v with z as the penultimate vertex,

5. $\mathrm{virtarc}_{u,v}$ represents the short-circuited directed paths through nodes outside the local instance.

We then introduce hard clauses over these variables to capture their semantics and to allow expressing expert constraints. This implies that these constraints are treated as hard constraints. At times, we write the clauses using the friendlier implication notation. However, all of these clauses can be converted into the standard *Conjunctive Normal Form* (CNF) required by the MaxSAT solver. For this reason, the encoding accepts as input all the elementary constraints as well as disjunctions over elementary constraints.

Phase 2 only considers the set of constraints satisfied by the initial heuristic solution as hard constraints. This ensures that all the constraints satisfied by the Phase 1 solution remain satisfied at the end of Phase 2. Further, there might be some constraints that were previously violated in the heuristic solution but end up being coincidentally satisfied by Phase 2. Thus, the set of satisfied constraints by the Phase 2 solutions is a (not necessarily strict) superset of the set of constraints satisfied by the Phase 1 solution.

To disallow simultaneous arcs in opposite directions in the DAG, we add the clauses

$$\neg\mathrm{dagarc}_{u,v} \vee \neg\mathrm{dagarc}_{v,u} \qquad \text{for all } u \neq v \in S.$$

We then add the following clauses to ensure that $\mathrm{dagarc}_{u,v}$ is true if and only if u is in the parent set of v.

$$\mathrm{par}_v^P \Rightarrow \bigwedge_{u\in P} \mathrm{dagarc}_{u,v} \qquad \text{for all } v \in S, P \in \mathcal{P}_v, \text{ and}$$

$$\mathrm{dagarc}_{u,v} \Rightarrow \bigvee_{P\in\mathcal{P}_v \text{ s.t. } u\in P} \mathrm{par}_v^P \qquad \text{for all } u \neq v \in S.$$

And finally, we propagate the DAG arcs to the arcs of the moralized graph using the clauses

$$\mathrm{dagarc}_{u,v} \Rightarrow \mathrm{arc}_{u,v} \qquad \text{for all } u \neq v \in S.$$

For the $\mathrm{tarc}_{u,v}$ variables, we initialize the transitivity using the $\mathrm{dagarc}_{u,v}$ and $\mathrm{virtarc}_{u,v}$ variables as follows

$$\left.\begin{array}{l}\mathrm{dagarc}_{u,v} \Rightarrow \mathrm{tarc}_{u,v}\\ \mathrm{virtarc}_{u,v} \Rightarrow \mathrm{tarc}_{u,v}\end{array}\right\} \text{for all } u \neq v \in S,$$

and then encode the transitivity using the following clauses

$$\mathrm{tarc}_{u,v} \land \mathrm{tarc}_{v,w} \Rightarrow \mathrm{tarc}_{u,w} \quad \text{for all distinct } u, v, w \in S.$$

To encode the path variables, we first encode the condition that the path can either be a single arc in the DAG, a single external virtual arc or a path through at least one other variable z. For this we add the following clauses for all $u \neq v \in S$,

$$\mathrm{path}_{u,v} \Rightarrow \mathrm{dagarc}_{u,v} \lor \mathrm{virtarc}_{u,v} \lor \bigvee_{z \neq u,v} \mathrm{pathq}_{u,v,z}.$$

Then, we encode the condition for the existence of a path from u to v with z in the penultimate position, by asserting a path from u to z and either a direct arc or a virtual arc from z to v. For this we add the following clauses for all distinct $u, v, z \in S$,

$$\mathrm{pathq}_{u,v,z} \Rightarrow \mathrm{path}_{u,z} \land (\mathrm{dagarc}_{z,v} \lor \mathrm{virtarc}_{z,v}).$$

Finally, we encode the constraints using the predicates described so far. For the arc constraints, we use the $\mathrm{dagarc}_{u,v}$ variables as follows

$$\begin{aligned} &\text{for } u \to v, && \text{we use } \mathrm{dagarc}_{u,v}, \\ &\text{for } u \not\to v, && \text{we use } \neg\mathrm{dagarc}_{u,v}, \\ &\text{for } u \leftrightarrow v, && \text{we use } \mathrm{dagarc}_{u,v} \lor \mathrm{dagarc}_{v,u}. \end{aligned}$$

For the ancestry constraints, we use the $\mathrm{path}_{u,v}$ and $\mathrm{tarc}_{u,v}$ variables as follows

$$\begin{aligned} &\text{for } u \rightsquigarrow v, && \text{we use } \mathrm{path}_{u,v}, \\ &\text{for } u \not\rightsquigarrow v, && \text{we use } \neg\mathrm{tarc}_{u,v}, \end{aligned}$$

It is subtle but worth noting nonetheless, that the clause $\neg\mathrm{path}_{u,v}$ does not ensure the absence of a path from u to v in the DAG, i.e., the $\mathrm{path}_{u,v}$ variables can only be used to assert the existence of paths ($\rightsquigarrow$ constraints), not their absence ($\not\rightsquigarrow$ constraints). This is reason we use the $\mathrm{tarc}_{u,v}$ variables to be able to assert the absence of paths.

6.5 Experimental Evaluation

6.5.1 Setup

We tested the two proposed heuristics on a 4-core Intel Xeon E5540 2.53 GHz CPU cluster, with each process having access to 8 GB RAM. The k-greedy algorithm is available as a part of the BLIP package [Sca15] implemented in Java. The source code for Con-BN-SLIM was built by extending the BN-SLIM codebase and is available as a public GitHub repository [PS21a]. BN-SLIM uses the Python NetworkX library [HSS08], and the UWrMaxSat[1] as the MaxSAT solver.

We ran the heuristics on score function caches and constraints sets generated from all the discrete networks available as a part of the bnlearn BN repository.[2] This repository is commonly used for benchmarking Bayesian Networks [LB18; Che+16; Sca+18; Sca+16]. We split up the networks into three groups—small, medium, and large—based on the number of random variables. We then synthesized expert constraints by randomly sampling a fixed number η of constraints of each of the 5 types from the ground truth networks (see Table 6.4). Note that this repository consists of the networks themselves, not the instances or samples drawn from the BNs. Additionally, we also precomputed the treewidths of all the ground truth networks (ranging between 3 and 15) and used those values as the bounds for all the heuristics.

Group	Variables	η
Small	up to 50	$\{5, 10\}$
Medium	50 to 500	$\{10, 25, 50\}$
Large	above 500	$\{25, 50, 75\}$

Table 6.4: Data set characteristics. η denotes the set of possible values for number of constraints of each type in the input.

6.5.2 Method

We now explain the format of the experiments used to compare the proposed heuristics, which is similar to that of Section 4.4. We precompute the score function caches using the available functionality from the BLIP package. All the evaluated methods are supplied with the same score function caches. We then randomly synthesized different constraints using three random seed values. The score function caches along with a corresponding constraint set are together considered to be one input instance. This results in a total of 183 input instances.

We then ran the original k-greedy algorithm and the Con-k-greedy algorithm on these inputs for 60 minutes. For each input, we ran the heuristics with three different random seed values. For evaluating Con-BN-SLIM, we used the intermediate solution produced by Con-k-greedy at the 30-minute mark as the starting heuristic. After which, we run Con-BN-SLIM for another 30 minutes, thereby fixing the total runtime of each method to 60 minutes. For each input, we ran Con-BN-SLIM with 8 different configurations (random seed, timeout, encoding type). For all the experiments, we record the final score, the final satisfied constraint count, and the rate of improvement.

[1] https://maxsat-evaluations.github.io/2019/descriptions.html

[2] https://www.bnlearn.com/bnrepository/

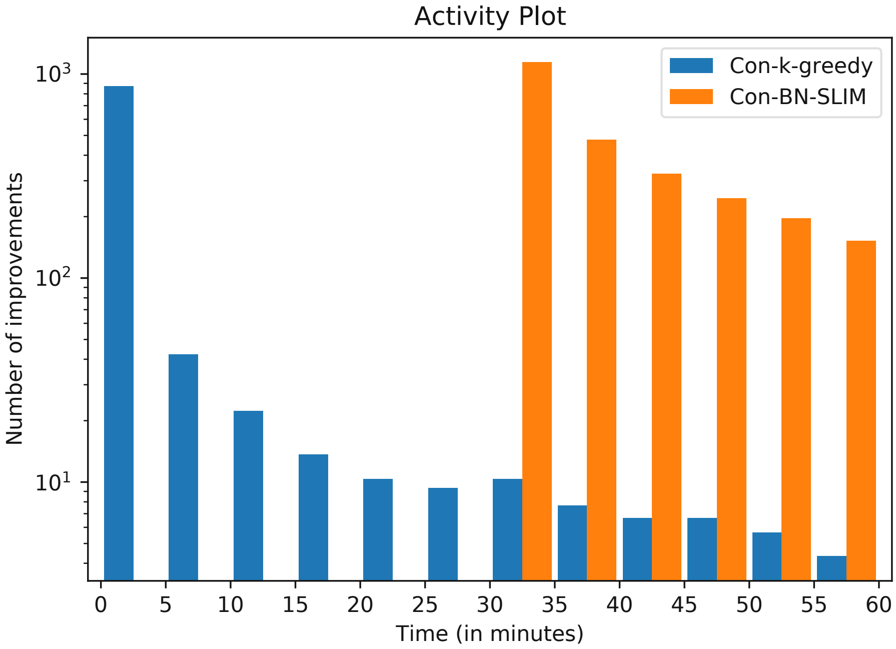


Figure 6.2: Activity plot showing rate of improvement of Con-BN-SLIM and Con-k-greedy against time. Note that the y-axis is in log scale.

6.5.3 Results

As a continuation to Figure 6.1, we first visualize the activity of Con-BN-SLIM compared to Con-k-greedy. Note that, Con-BN-SLIM only starts running at the 30-minute mark (after being handed the heuristic solution from Con-k-greedy) and hence does not record any improvements till that point. As is evident from Figure 6.2, despite the rate of improvements of Con-k-greedy slowing down drastically, when Con-BN-SLIM takes over, it is still able to find many improvements over the exact same networks. This demonstrates the notion of turbocharging quite well.

Next, we compare the scores of the networks produced by Con-k-greedy and Con-BN-SLIM at the 60-minute mark. We use the ΔBIC metric to make this comparison. The difference in BIC scores of two networks approximates the ratio of their marginal likelihoods, which is the Bayes Factor [Raf95; Sca+18]. The ΔBIC score of a pair of networks is mapped to a categorical scale, with positive scores signifying positive evidence towards the first network and vice versa. As can be seen from Table 6.5, Con-BN-SLIM severely outperforms Con-k-greedy.

Finally, we compare the constraint satisfaction by the solutions of k-greedy, Con-k-greedy, and Con-BN-SLIM in Table 6.6. We measure and tabulate the percentage of total constraints satisfied. There are several noteworthy points here.

Category	Δ BIC	Count
extremely positive	(10, ∞)	127
strongly positive	(6, 10)	0
positive	(2, 6)	0
neutral	(-2, 2)	14
negative	(-6, -2)	1
strongly negative	(-10, -6)	0
extremely negative	($-\infty$, -10)	7

Table 6.5: ΔBIC values comparing Con-BN-SLIM against Con-k-greedy

Group	k-greedy	Con-k-greedy	Con-BN-SLIM
Small	77.74%	84.52%	90.24%
Medium	63.80%	74.43%	81.73%
Large	59.44%	88.91%	89.44%
All	67.54%	81.73%	86.53%

Table 6.6: Comparison of average number of satisfied constraints as a percentage of number of total constraints

k-greedy

We see that k-greedy, despite having no knowledge of the constraints, manages to satisfy more than half of them. This could be attributed to the fact that k-greedy still has access to the score function caches whose job is to quantify and reflect the closeness of any network to the ground truth network (just like the expert constraints).

Con-k-greedy

We see a clear improvement in the constraint satisfaction by Con-k-greedy compared to k-greedy. This is to be expected as we modified the heuristic to consider the expert constraints.

Con-BN-SLIM

We see that Con-BN-SLIM ends up satisfying slightly more constraints than Con-k-greedy even though it was not intentionally designed to do so. This, however, is a favorable side effect. Con-BN-SLIM never violates a constraint that was satisfied by the initial heuristic solution. Thus, by random chance, the number of satisfied constraints can only increase.

6.6 Conclusion

We have proposed the first method for BN structure learning that scales to large instances while respecting treewidth bounds and soft expert constraints. At the heart of our method is utilizing a MaxSAT encoding, applied locally, which demonstrates the flexibility of the SLIM framework.

We see several possibilities for improving the portion of satisfied expert constraints. An easy target is improving the Phase 1 heuristics to better handle the root bag construction, which a MaxSAT encoding could provide. Even more potential might be to adapt other heuristics like k-MAX [Sca+18] or Elimination Trees [BBL19] for Phase 1.

The current implementation does not actively try to increase the satisfied constraints in Phase 2. Despite that, it was somewhat surprising for us to still see a significant increase in the number of satisfied constraints (see Table 6.6). This suggests a learning approach where we continuously check during Phase 2 whether any previously violated expert constraint is satisfied and if so, add it as a hard constraint to the Phase 2 engine. This way, Phase 2 could yield a monotonic increase in both the score and the number of satisfied constraints.

The local solver is essentially a CNF formula, and we have not exhausted its whole range of expressiveness with the constraints explored in this work. Thus, another viable future direction could be to explore more sophisticated constraint types. Similarly, one can look into incorporating expert constraints into the heuristic learning algorithms for other probabilistic graphical models.

CHAPTER 7

Conclusion

As our circle of knowledge expands, so does the circumference of darkness surrounding it.

Albert Einstein

We have reached the end our journey through the world of BNSL. We began from the basic bounded treewidth variant and then stopped by the bounded state space variant before finally looking at the constrained variant. In this chapter, we recall and summarize the contributions from the previous chapters, look at potential future avenues of research, and close with general directions for future work and commonly applicable remarks.

7.1 Bounded Treewidth BNSL

In our work on bounded treewidth BNSL (Chapter 4), we studied the problem of learning bounded treewidth BNs from data and devised a scalable SLIM approach to heuristically solve the problem. We built on top of the state-of-the-art heuristic k-MAX and the SAT encoding from Samer and Veith [SV09]. We significantly upgraded the encoding to support virtual arcs and virtual edges which in turn allowed us to maintain replacement consistency. The resultant heuristic algorithm outperformed the state of the art, which at that time, was k-MAX.

This work can be extended in several ways. One of which, we have already seen in Chapter 5, where we bound a different metric of the tree decomposition as opposed to the treewidth. Additionally, we could replace the random traversal-based subinstance selection with more well-guided approach similar to the work by Hickey and Bacchus [HB22]. We also briefly experimented with a variant of BN-SLIM that periodically compares the current heuristic solution (say B_h) with its own current best (say B), and if $f(B_h) > f(B)$, then BN-SLIM adopts B_h as its own current best and proceeds to optimize this solution further. This is similar in spirit to the concept of 'restarts' used by SAT-solvers.

Another interesting line of future work is what we call variable fidelity sampling. For this, we compute different score function caches for the local and global solver. The local solver has access to a finer-grained score function cache with potentially more parents per variable. Naturally, this would come at a higher cost, but since the local solver only works with relatively small number of variables (governed by the budget β), this higher cost isn't particularly deterrent. On the other hand, the global solver, which has to deal with thousands of variables, would be overwhelmed if it operated at a similar level of fidelity as the local solver. Thus, sparser score function caches with fewer parent sets per variable would be more suitable for the scales which the global solver has to handle.

7.2 Bounded State Space BNSL

In our work on bounded state space BNSL (Chapter 4), we extended the basic Bounded Treewidth BNSL problem to a more fine-grained metric as compared to treewidth. This was motivated by empirical evidence of better correlation between the state space size of a BN and its inference time as compared to treewidth of a BN and the inference time. We developed a SLIM-based algorithm, building on top of BN-SLIM, to tackle the Bounded State Space BNSL problem. In the process, we upgraded the encoding, which is a part of the local solver, by replacing the basic linear cardinality counter for treewidth with a much more sophisticated Binary Decision Diagram-based counter. This counter allowed us to represent and bound the sum of the logarithms of the domain sizes of the variable sizes, i.e., bound the state space size of a BN. This is a good showcase for the flexibility due to the MaxSAT encoding-based local solver. As a baseline for comparison, we also upgraded the k-MAX heuristic such that it can search over the space of bounded state space BNs instead of bounded treewidth BNs, and found that BN-SLIM obtains networks with quicker inference times at a slight disadvantage to the score of the learned network.

In the future, it would be interesting to develop a heuristic that supports bounded state space size natively, rather than retrofitting an existing bounded treewidth heuristic. Further, as many of the reviewers of the conference venue *Advances in Neural Information Processing Systems 2021 (NeurIPS'21)* appreciated, there is a huge abundance of bounded treewidth based methods but hardly any bounded state space methods; despite it being the superior metric when it comes to ensuring tractable inference. Thus, the BNSL community as a whole, could benefit from widespread adoption of the bounded state space metric which is perhaps accelerated by the availability of more bounded state space methods.

7.3 BNSL with Expert Constraints

In our work on Constrained BNSL (Chapter 6), we proposed the first algorithm for BNSL that simultaneously supports the requirements of

i. learning a BN of bounded treewidth,

ii. satisfying expert constraints, including positive and negative ancestry properties between nodes, and

iii. scaling up to BNs with several thousand nodes.

The algorithm operates in two phases. In Phase 1, we modify the BN structure learning algorithm k-greedy to obtain Con-k-greedy. k-greedy is modified such that the BN it learns support a portion of the given constraints. In Phase 2, we follow the BN-SLIM framework. We improve the initial heuristic BN produced by Con-k-greedy by repeatedly running a MaxSAT solver on selected local parts. The MaxSAT encoding entails local versions of the expert constraints as hard constraints, thereby ensuring that all the constraints remain satisfied while also maximizing the score of the local part. This is another example of how the expressiveness of MaxSAT proves to be helpful. We evaluated a prototype implementation of our algorithm on several standard benchmark sets. The encouraging results demonstrate the power and flexibility of the BN-SLIM framework. It boosts the score while increasing the number of satisfied expert constraints.

This work has numerous promising follow-ups. Similar to Con-k-greedy, we could also upgrade k-MAX or ETL to respect expert constraints. This would be more involved but could result in initial heuristic solutions with significantly higher scores. It might also be worth investigating if other (not necessarily state-of-the-art) BNSL algorithms can be upgraded to respect expert constraints. Since both k-greedy and k-MAX operate on a randomly sampled linear ordering, it is a common occurrence that they fail to find any candidate networks respecting the expert constraints. This was precisely the reason for treating some expert constraints as soft constraints which meant that SLIM could not guarantee 100% constraint satisfaction despite Phase 2 treating all hitherto satisfied constraints as hard constraints. Another related area for improvement is the construction of the root bag in Con-k-greedy in Phase 1. We can use an exact solver which respects all expert constraints in this place, and the MaxSAT encoding itself would be viable candidate for the same.

There is a lot of potential for future research in Phase 2 as well. At the moment, we noticed that the number of satisfied constraints sometimes increases in Phase 2 despite the encoding not being explicitly asked to optimize for it. This can be easily turned in our favor, if we update the status of these accidentally satisfied constraints to be hard constraints from that point on. This way, we can promise a monotonic increase in both the score and the number of satisfied constraints. Lastly, another natural extension is to consider more types of constraints and possibly more sophisticated constraints. We suspect that the MaxSAT encoding currently in use is not nearly at its limit and can probably handle more different constraint types. One could also investigate the possibility of incorporating expert constraints into probabilistic graphical models other than BNs.

7.4 General Future Directions

There are several potential lines of research that apply to all the SLIM approaches we have seen so far. The most straightforward of those is parallelization. Since each individual improvement works on a local part, it lends itself well to parallelization across different non-overlapping local parts. This, of course, requires access to a multicore processor to reap benefits. One potential strategy for accomplishing this could be to maintain a global *job queue* and several *workers*. Then the job *manager*, pushes local parts as jobs onto the queue, which the workers then sequentially pick up one by one. Then the worker acts as a local solver for the job that they picked up and upon completion, marks the job as done, removing it from the queue. The manager has the responsibility of ensuring that none of the jobs currently present on the queue overlap.

We could also investigate alternate strategies for local instance selection. Currently, we select the local instances somewhat randomly. We pick a random starting bag in the tree decomposition and keep including variables from the neighboring bags as long as the budget is not exceeded. We could instead opt for smarter and more sophisticated approaches. An example of this is defining a 'promise' metric for local subinstances and picking the most promising subinstance first. Such a promise metric would have to be problem-dependent, for instance, for score-based BNSL, inspired by k-MAX, one could compare the current sum of local scores to the maximum possible sum of local scores from the cache for the variables in a local instance. Another idea is to use so-called 'tabu lists' alongside the randomized instance selection. The tabu list ensures that the same variable does not keep getting picked repeatedly and that the local instances do a good job spanning the entire global solution.

A final interesting idea is the technique of 'interleaving', seen in Reichl, Slivovsky, and Szeider [RSS23]. A prerequisite for this technique is that we need a global solver that can itself optionally accept a starting solution and improve it further. Both k-greedy and k-MAX fail to satisfy this requirement as they always learn a new BN from scratch and cannot optimize a provided BN. However, ETL might be promising candidate for this. The key idea is to alternate between the global solver and the local solver always starting from the current best solution, instead of just using the global solver once at the beginning.

7.5 Closing Remarks

Throughout this thesis, we looked at several variants of the BNSL problem and how the SLIM framework is an incredibly effective tool for solving this problem. We attribute this success to the inherent complementary nature of the local solver and the global solver. By virtue of this diversity, the SLIM framework is able to surmount a wide range of problems without changing the skeleton of the framework and with only relatively small modifications.

We would like to present an interesting analogy to demonstrate this. Consider the problem at hand to be a marathon race. The distance covered corresponds to the quality of the solution and the time required to cover that distance is the running time of the algorithm. Some of the contenders at the starting line are:

- heuristic algorithms: fast and nimble, but they run out of steam quickly and struggle to cover good distance,
- exact algorithms: what they lack in speed, they make up for in endurance, and hence are likely to reach the finish line, albeit very slowly.

Now, a SLIM algorithm is like a team of two, composed of a heuristic global solver and an exact local solver. For SLIM, the marathon is more like a relay race, where it capitalizes on the speed of the heuristic solver to gain good distance quickly at the start of the race and then the baton is passed over from the global solver to the exact local solver. SLIM then uses the meticulousness (i.e., endurance) of the local solver to still ensure a steady rate of progress at a point where the heuristic algorithm fails to do so. In this way, SLIM can take advantage of both these approaches and outperform methods that rely on just one.

List of Figures

List of Tables

List of Algorithms

Definition Index

Expert Index

Bibliography

[Aka74] Hirotugu Akaike. "A new look at the statistical model identification". In: *IEEE transactions on automatic control* 19.6 (1974), pp. 716–723.

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. "Complexity of finding embeddings in a k-tree". In: *SIAM J. Algebraic Discrete Methods* 8.2 (1987), pp. 277–284.

[Bek+15] Jessa Bekker et al. "Tractable learning for complex probability queries". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2242–2250.

[BBL19] Marco Benjumeda, Concha Bielza, and Pedro Larrañaga. "Learning Tractable Bayesian networks in the space of elimination orders". In: *Artificial Intelligence* 274 (2019), pp. 66–90.

[BJM14] Jeremias Berg, Matti Järvisalo, and Brandon M. Malone. "Learning Optimal Bounded Treewidth Bayesian Networks via Maximum Satisfiability". In: *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*. Vol. 33. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 86–95.

[BB73] Umberto Bertele and Francesco Brioschi. "On non-serial dynamic programming". In: *J. Comb. Theory, Ser. A* 14.2 (1973), pp. 137–148.

[BM93] Hans L. Bodlaender and Rolf H. Möhring. "The Pathwidth and Treewidth of Cographs". In: *SIAM J. Discrete Math.* 6.2 (1993), pp. 181–188. DOI: 10.1137/0406014.

[Che+16] Eunice Yuh-Jie Chen et al. "Learning Bayesian networks with ancestral constraints". In: *Advances in Neural Information Processing Systems* 29 (2016).

[Chi96] David Maxwell Chickering. *Learning Equivalence Classes Of Bayesian Network Structures*. 1996.

[Chi02] David Maxwell Chickering. "Learning Equivalence classes of Bayesian-Network Structures". In: *J. Mach. Learn. Res.* 2 (2002), pp. 445–498.

[CHM04] Max Chickering, David Heckerman, and Chris Meek. "Large-sample learning of Bayesian networks is NP-hard". In: *Journal of Machine Learning Research* 5 (2004), pp. 1287–1330.

[Coo90] Gregory F. Cooper. “The computational complexity of probabilistic inference using Bayesian belief networks”. In: *Artificial Intelligence* 42.2-3 (1990), pp. 393–405.

[Cor+13] Jukka Corander et al. “Learning Chordal Markov Networks by Constraint Satisfaction”. In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc., 2013. (Visited on 02/23/2022).

[Cyg+15] Marek Cygan et al. *Parameterized algorithms*. Vol. 5. 4. Springer, 2015.

[DL93] Paul Dagum and Michael Luby. “Approximating Probabilistic Inference in Bayesian Belief Networks is NP-Hard”. In: *Artificial Intelligence* 60.1 (1993), pp. 141–153.

[Dec99] Rina Dechter. “Bucket elimination: a unifying framework for reasoning”. In: *Artificial Intelligence* 113.1-2 (1999), pp. 41–85.

[Die00] Reinhard Diestel. *Graph Theory*. 2nd. Vol. 173. Graduate Texts in Mathematics. New York: Springer Verlag, 2000.

[ES06] Niklas Eén and Niklas Sörensson. “Translating Pseudo-Boolean Constraints into SAT”. In: *J. Satisf. Boolean Model. Comput.* 2.1-4 (2006), pp. 1–26. DOI: 10.3233/sat190014. URL: https://doi.org/10.3233/sat190014.

[EG09] Gal Elidan and Stephen Gould. “Learning Bounded Treewidth Bayesian Networks”. In: *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*. Ed. by Daphne Koller et al. Curran Associates, Inc., 2009, pp. 417–424.

[FLS17] Johannes K. Fichte, Neha Lodha, and Stefan Szeider. “SAT-Based Local Improvement for Finding Tree Decompositions of Small Width”. In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer Verlag, 2017, pp. 401–411. DOI: 10.1007/978-3-319-66263-3_25.

[Fic+18] Johannes K. Fichte et al. “An SMT Approach to Fractional Hypertree Width”. In: *Proceedings of CP 2018, the 24rd International Conference on Principles and Practice of Constraint Programming*. Ed. by John N. Hooker. Vol. 11008. Lecture Notes in Computer Science. Springer Verlag, 2018, pp. 109–127. DOI: 10.1007/978-3-319-98334-9_8.

[Gan+19] Robert Ganian et al. “SAT-Encodings for Treecut Width and Treedepth”. In: *Proceedings of ALENEX 2019, the 21st Workshop on Algorithm Engineering and Experiments*. Ed. by Stephen G. Kobourov and Henning Meyerhenke. SIAM, 2019, pp. 117–129. DOI: 10.1137/1.9781611975499.10.

[GD04] Vibhav Gogate and Rina Dechter. “A Complete Anytime Algorithm for Treewidth”. In: *Proceedings of the Twentieth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-04)*. Arlington, Virginia: AUAI Press, 2004, pp. 201–208.

[Guo] Haipeng Guo. *BBNConvertor – Bayesian Networks Formats Convertor*.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring network structure, dynamics, and function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. Pasadena, CA USA, Aug. 2008, pp. 11–15.

[Hal76] Rudolf Halin. “S-functions for graphs”. In: *Journal of Geometry* 8.1 (1976), pp. 171–186.

[HNC65] Frank Harary, Robert Z. Norman, and Dorwin Cartwright. *Structural Models: An Introduction to the Theory of Directed Graphs*. New York: John Wiley & Sons, 1965.

[HGC95] David Heckerman, Dan Geiger, and David Maxwell Chickering. “Learning Bayesian Networks: The Combination of Knowledge and Statistical Data”. In: *Machine Learning* 20.3 (1995), pp. 197–243.

[HB22] Randy Hickey and Fahiem Bacchus. “Large Neighbourhood Search for Anytime MaxSAT Solving”. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. Ed. by Lud De Raedt. Main Track. International Joint Conferences on Artificial Intelligence Organization, July 2022, pp. 1818–1824. DOI: 10.24963/ijcai.2022/253. URL: https://doi.org/10.24963/ijcai.2022/253.

[Ide15] Jaime S. Ide. *BNGenerator – A generator for random Bayesian network*. 2015. URL: http://sites.poli.usp.br/pmr/ltd/Software/BNGenerator (visited on 06/03/2020).

[Kas+11] Kalev Kask et al. “Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models”. In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*. Ed. by Wolfram Burgard and Dan Roth. AAAI Press, 2011, pp. 54–60.

[KKN01] Russell J Kennett, Kevin B Korb, and Ann E Nicholson. “Seabreeze prediction using Bayesian networks”. In: *Pacific-Asia conference on knowledge discovery and data mining*. Springer. 2001, pp. 148–153.

[Kjæ92] Uffe Kjærulff. “Optimal decomposition of probabilistic networks by simulated annealing”. In: *Statistics and Computing* 2 (1992), pp. 7–17.

[KP13] Janne H. Korhonen and Pekka Parviainen. “Exact Learning of Bounded Treewidth Bayesian Networks”. In: *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2013, Scottsdale, AZ, USA, April 29 - May 1, 2013*. Vol. 31. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 370–378.

[KBG10] Johan Kwisthout, Hans L. Bodlaender, and Linda C. van der Gaag. “The Necessity of Bounded Treewidth for Efficient Inference in Bayesian Networks”. In: *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*. Ed. by Helder Coelho, Rudi Studer, and Michael Wooldridge. Vol. 215. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 237–242.

[LBT10] Hugo Larochelle, Yoshua Bengio, and Joseph Turian. “Tractable multivariate binary density estimation and the restricted Boltzmann forest”. In: *Neural computation* 22.9 (2010), pp. 2285–2307.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. “Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems”. In: *J. R. Stat. Soc. Ser. B.* 50.2 (1988), pp. 157–224.

[LB18] Andrew Li and Peter van Beek. “Bayesian network structure learning with side constraints”. In: *International Conference on Probabilistic Graphical Models*. PMLR. 2018, pp. 225–236.

[LOS16] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. “A SAT Approach to Branchwidth”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer Verlag, 2016, pp. 179–195. DOI: 10.1007/978-3-319-40970-2_12.

[LOS17] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. “SAT-Encodings for Special Treewidth and Pathwidth”. In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer Verlag, 2017, pp. 429–445. DOI: 10.1007/978-3-319-66263-3_27. URL: http://www.ac.tuwien.ac.at/files/tr/ac-tr-17-012.pdf.

[LOS19] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. “A SAT Approach to Branchwidth”. In: *ACM Trans. Comput. Log.* 20.3 (2019), 15:1–15:24. DOI: 10.1145/3326159. URL: http://www.ac.tuwien.ac.at/files/tr/ac-tr-19-010.pdf.

[LD10] Daniel Lowd and Jesse Davis. “Learning Markov network structure with decision trees”. In: *2010 IEEE International Conference on Data Mining*. IEEE. 2010, pp. 334–343.

[MJ96] Marina Meila and Michael I. Jordan. “Triangulation by Continuous Embedding”. In: *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*. Ed. by Michael Mozer, Michael I. Jordan, and Thomas Petsche. MIT Press, 1996, pp. 557–563.

[NJ07] Richard E. Neapolitan and Xia Jiang. "Chapter 4 - Learning Bayesian Networks". In: *Probabilistic Methods for Financial and Marketing Informatics*. Ed. by Richard E. Neapolitan and Xia Jiang. Burlington: Morgan Kaufmann, 2007, pp. 111–175. ISBN: 978-0-12-370477-1. DOI: https://doi.org/10.1016/B978-012370477-1.50021-9. URL: https://www.sciencedirect.com/science/article/pii/B9780123704771500219.

[NCJ15] Siqi Nie, Cassio Polpo de Campos, and Qiang Ji. "Learning Bounded Tree-Width Bayesian Networks via Sampling". In: *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 13th European Conference, ECSQARU 2015, Compiègne, France, July 15-17, 2015. Proceedings*. Ed. by Sébastien Destercke and Thierry Denoeux. Vol. 9161. Lecture Notes in Computer Science. Springer Verlag, 2015, pp. 387–396.

[NCJ16] Siqi Nie, Cassio Polpo de Campos, and Qiang Ji. "Learning Bayesian Networks with Bounded Tree-width via Guided Search". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 3294–3300.

[OD08a] Lars Otten and Rina Dechter. "Bounding Search Space Size via (Hyper)tree Decompositions". In: *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*. Ed. by David A. McAllester and Petri Myllymäki. AUAI Press, 2008, pp. 452–459.

[OD08b] Lars Otten and Rina Dechter. "Refined Bounds for Instance-Based Search Complexity of Counting and Other #P Problems". In: *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*. Ed. by Peter J. Stuckey. Vol. 5202. Lecture Notes in Computer Science. Springer Verlag, 2008, pp. 576–581. DOI: 10.1007/978-3-540-85958-1_45.

[PFL14] Pekka Parviainen, Hossein Shahrabi Farahani, and Jens Lagergren. "Learning Bounded Tree-width Bayesian Networks using Integer Linear Programming". In: *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*. Vol. 33. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 751–759.

[Pea85] Judea Pearl. "Bayesian networks: A model of self-activated memory for evidential reasoning". In: *Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, CA, USA*. 1985, pp. 15–17.

[Pea88] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. The Morgan Kaufmann Series in Representation and Reasoning. San Mateo, CA: Morgan Kaufmann, 1988.

[PS20] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “MaxSAT-Based Postprocessing for Treedepth”. In: *Proceedings of CP 2020, the 26th International Conference on Principles and Practice of Constraint Programming*. Ed. by Helmut Simonis. Vol. 12333. Lecture Notes in Computer Science. Springer Verlag, 2020, pp. 478–495. DOI: 10.1007/978-3-030-58475-7_28.

[PS21a] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. *BN-SLIM source code*. Version nips21. Oct. 2021. DOI: 10.5281/zenodo.5598257. URL: https://doi.org/10.5281/zenodo.5598257.

[PS21b] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “Learning Fast-Inference Bayesian Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 17852–17863. URL: https://proceedings.neurips.cc/paper/2021/file/94e70705efae423efda1088614128d0b-Paper.pdf.

[PS21c] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “Turbocharging Treewidth-Bounded Bayesian Network Structure Learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.5 (May 2021), pp. 3895–3903. DOI: 10.1609/aaai.v35i5.16508. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16508.

[PS22] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. “Learning large Bayesian networks with expert constraints”. In: *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence*. Ed. by James Cussens and Kun Zhang. Vol. 180. Proceedings of Machine Learning Research. PMLR, Aug. 2022, pp. 1592–1601. URL: https://proceedings.mlr.press/v180/peruvemba-ramaswamy22a.html.

[PR10] David Pisinger and Stefan Ropke. “Large neighborhood search”. In: *Handbook of metaheuristics*. Springer, 2010, pp. 399–419.

[Raf95] Adrian E. Raftery. “Bayesian Model Selection in Social Research”. In: *Sociological Methodology* 25 (1995), pp. 111–163. ISSN: 00811750, 14679531. DOI: 10.2307/271063.

[RSS23] Franz Reichl, Friedrich Slivovsky, and Stefan Szeider. “Circuit Minimization with QBF-Based Exact Synthesis”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37 (2023). To appear.

[RS84] Neil Robertson and Paul D Seymour. “Graph minors. III. Planar tree-width”. In: *Journal of Combinatorial Theory, Series B* 36.1 (1984), pp. 49–64.

[Rot96] Dan Roth. “On the hardness of approximate reasoning”. In: *Artificial Intelligence* 82.1-2 (1996), pp. 273–302.

[SV09] Marko Samer and Helmut Veith. “Encoding Treewidth into SAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Vol. 5584. Lecture Notes in Computer Science. Springer Verlag, 2009, pp. 45–50.

[Sca15] Mauro Scanagatta. *BLIP – Bayesian Network learning and inference package*. 2015. URL: https://ipg.idsia.ch/software/blip (visited on 06/03/2020).

[Sca+16] Mauro Scanagatta et al. "Learning Treewidth-Bounded Bayesian Networks with Thousands of Variables". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al. 2016, pp. 1462–1470.

[Sca+18] Mauro Scanagatta et al. "Efficient learning of bounded-treewidth Bayesian networks from complete and incomplete data sets". In: *Int. J. Approx. Reason* 95 (2018), pp. 152–166.

[Sch22] André Schidler. "SAT-Based Local Search for Plane Subgraph Partitions". In: *Symposium on Computational Geometry (SoCG)*. 2022, 74:1–74:8.

[SS20] André Schidler and Stefan Szeider. "Computing Optimal Hypertree Decompositions". In: *Proceedings of ALENEX 2020, the 22nd Workshop on Algorithm Engineering and Experiments*. Ed. by Guy Blelloch and Irene Finocchi. SIAM, 2020, pp. 1–11.

[SS21] André Schidler and Stefan Szeider. "SAT-based Decision Tree Learning for Large Data Sets". In: *Proceedings of AAAI'21, the Thirty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, 2021.

[SS22] André Schidler and Stefan Szeider. *SAT-Boosted Tabu Search for Coloring Massive Graphs*. 2022.

[Sch78] Gideon Schwarz. "Estimating the dimension of a model". In: *The Annals of Statistics* 6.2 (1978), pp. 461–464.

[VD12] Jan Van Haaren and Jesse Davis. "Markov network structure learning: A randomized feature generation approach". In: *Twenty-Sixth AAAI Conference on Artificial Intelligence*. 2012.