December 2021

# Selected Topics in Network Optimization: Aligning Binary Decision Diagrams for a Facility Location Problem and a Search Method for Dynamic Shortest Path Interdiction

Alexey Bochkarev
*Clemson University*, a@bochkarev.io

# Selected Topics in Network Optimization: Aligning Binary Decision Diagrams for a Facility Location Problem and a Search Method for Dynamic Shortest Path Interdiction.

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Industrial Engineering

by
Alexey A. Bochkarev
December 2021

Accepted by:
Dr. Yongjia Song, Committee Chair
Dr. J. Cole Smith
Dr. Brian C. Dean
Dr. Thomas C. Sharkey

# Abstract

This work deals with three different combinatorial optimization problems: minimizing the total size of a pair of binary decision diagrams (BDDs) under a certain structural property, a variant of the facility location problem, and a dynamic version of the Shortest-Path Interdiction (DSPI) problem. However, these problems all have the following core idea in common: They all stem from representing an optimization problem as a decision diagram. We begin from cases in which such a diagram representation of reasonable size might exist, but finding a small diagram is difficult to achieve. The first problem develops a heuristic for enforcing a structural property for a collection of BDDs, which allows them to be merged into a single one efficiently. In the second problem, we consider a specific combinatorial problem that allows for a natural representation by a pair of BDDs. We use the previous result and ideas developed earlier in the literature to reformulate this problem as a linear program over a single BDD. This approach enables us to obtain sensitivity information, while often enjoying runtimes comparable to a mixed integer program solved with a commercial solver, after we pay the computational overhead of building the diagram (e.g., when re-solving the problem using different costs, but the same graph topology). In the last part, we examine DSPI, for which building the full decision diagram is generally impractical. We formalize the concept of a *game tree* for the DSPI and design a heuristic based on the idea of building only selected parts of this exponentially-sized decision diagram (which is not binary any more). We use a Monte Carlo Tree Search framework to establish policies that are near optimal. To mitigate the size of the game tree, we leverage previously derived bounds

for the DSPI and employ an alpha–beta pruning technique for minimax optimization. We highlight the practicality of these ideas in a series of numerical experiments.

# Acknowledgments

I am deeply grateful to Dr. J. Cole Smith for his advice, support, and being a great role model. And separately — for helping me to start, survive in the process, and finish my Ph.D. studies, across states and continents (none of which were trivial).

I would like to thank my advisory committee, professors Yongjia Song, Brian Dean, and Thomas Sharkey. Also, Dr. Jorge Sefair from Arizona State University for providing test instances for Chapter 5 (that is, besides supplying the problem itself in a paper from 2016, together with Dr. Smith) and Dr. Leonardo Lozano from University of Cincinnati, for fun discussions on Binary Decision Diagrams.

To my kids, for being such a great source of joy and inspiration (and also, for sharing my excitement regarding the max independent set problem — before the actual fight over comparing the solutions). To my parents, for doing everything they could to help me make my life and dreams converge.

I am indebted to my wife beyond any measure, for helping me to understand better what is inherently valuable in life, and for doing beyond what she could to keep our world from falling apart.

Finally, I would like to thank Clemson University's Division of Research, Graduate School, and Industrial Engineering Department for financial support in the framework of 2021–2022 Doctoral Dissertation Completion Grant.

# Table of Contents

**Title Page** . . . . . **i**

**Abstract** . . . . . **ii**

**Acknowledgments** . . . . . **iv**

**List of Algorithms** . . . . . **vii**

**List of Figures** . . . . . **viii**

**List of Tables** . . . . . **x**

**1 Introduction** . . . . . **1**

**2 Literature Overview** . . . . . **6**

**3 On Aligning Binary Decision Diagrams** . . . . . **10**
3.1 Formal definition of BDDs . . . . . 10
3.2 Collection of BDDs and the alignment problem . . . . . 14
3.3 The simplified problem: definition . . . . . 17
3.4 An algorithm to solve the simplified problem . . . . . 31
3.5 Numerical experiments . . . . . 42

**4 A BDD-based Approach to a Facility Location Problem** . . . . . **58**
4.1 Building the cover diagram . . . . . 60
4.2 Building the type diagram . . . . . 63
4.3 Solving the CPP representation of t-UFLP . . . . . 66
4.4 Breakdown of the proposed heuristic runtime for t-UFLP . . . . . 68
4.5 On performance of the BDD alignment . . . . . 71

**5 Monte Carlo Tree Search Framework for DSPI** . . . . . **74**
5.1 Problem formulation . . . . . 74
5.2 Monte Carlo Tree Search framework . . . . . 76
5.3 Algorithm presentation . . . . . 85
5.4 On the algorithm correctness . . . . . 87
5.5 Numerical experiments . . . . . 96

**6 Conclusions and Future Research . . . . . 111**

# List of Algorithms

A3.1 Sift-up (BDD). . . . . 18
A3.2 Swap (quasi-reduced BDD) . . . . . 21
A3.3 Align-to (weighted variable sequence) . . . . . 27
A3.4 Align-to (weighted variable sequence), linear time . . . . . 30
A3.5 BB-search for AP($S_A, S_B; T^*_{\mathrm{VS}}$) . . . . . 43
A4.1 Build-cover-BDD . . . . . 62
A4.2 Build-type-BDD . . . . . 65
A5.1 MCTS-next-move . . . . . 88
A5.2 MCTS-roll-out . . . . . 89
A5.3 MCTS-UB . . . . . 90
A5.4 MCTS-LB . . . . . 91
A5.5 MCTS-play-instance . . . . . 91

# List of Figures


3.1 BDD representation for max independent set . . . 13
3.2 BDD collection representation for max independent set . . . 15
3.3 Diagram $B$ (before the sift-up) . . . 19
3.4 Diagram $R(B)$ (after the sift-up) . . . 19
3.5 A swap-up operation (as per Algorithm A3.2) . . . 22
3.6 Transforming a variable sequence (sifts) . . . 28
3.7 Transforming a variable sequence (alternative) . . . 30
3.8 Optimal solution properties (variable sequences) . . . 34
3.9 An example for local optima (simplified problem) . . . 35
3.10 Search tree example (simplified problem) . . . 39
3.11 Align-BDD dataset: layer widths summary . . . 45
3.12 Lower bounds benchmarking (simplified problem) . . . 47
3.13 Runtime and objectives (simplified problem) . . . 49
3.14 Aligned BDD sizes summary . . . 50
3.15 Runtime scaling (original problem) . . . 52
3.16 Example BDD: on variable sequence tightness . . . 54
3.17 Histogram for the number of optima . . . 55
3.18 Histogram for the optimal set "diameter" . . . 55
3.19 Similarity scores . . . 56
3.20 Effect of the initial BDD orders . . . 57

4.1 Example t-UFLP structure . . . 61
4.2 Cover BDD example . . . 61
4.3 Type BDD example . . . 61
4.4 Runtimes for t-UFLP . . . 69
4.5 Breakdown of the runtimes into solution steps for a CPP . . . 71
4.6 Intersection diagram size (tUFLP vs. random diagrams) . . . 73

5.1 Summary of the key steps of MCTS–DSPI algorithm . . . 80
5.2 An illustration: alpha–beta pruning . . . 83
5.3 An illustration: a case with a cycle . . . 94
5.4 MCTS solutions (pre-defined instances) . . . 98
5.5 MCTS solutions and runtimes (randomly generated instances) . . . 101
5.6 MCTS convergence profile: $\varepsilon = 0.05$ . . . 105
5.7 MCTS convergence profile: $\varepsilon = 0.5$ . . . 106
5.8 MCTS convergence profile: $\varepsilon = 1.0$ . . . 107

5.9 MCTS tree sizes before the first move . . . . . 108
5.10 MCTS: play-out vs. first-move strategies . . . . . 110

# List of Tables

3.1 Example: alternative optima (simplified problem) . . . . . 34
3.2 Example: single-sift changes . . . . . 36
3.3 Heuristic performance (simplified problem). . . . . 48

5.1 Runtimes and objectives: MCTS vs. exact algorithms . . . . . 99

# Chapter 1

# Introduction

This dissertation focuses on leveraging decision diagrams to solve combinatorial optimization problems. We consider several specific aspects of this vast topic and illustrate the proposed approaches with a few network optimization problems. The first large part of the work, presented in Chapters 3 and 4, deals with binary decision diagrams (BDDs). We develop a technique that helps consider some of the combinatorial problems in a framework of optimization over BDDs, and demonstrate the possible computational pipeline with a variant of the Facility Location Problem. This approach implies building the diagram first, and then solving an optimization problem to obtain a solution. Then, we take another perspective in Chapter 5 and adapt a method that would rely on building the diagram along with looking for a solution, similar to how a branch-and-bound algorithm would work. We consider a complex variation of the widely studied Shortest-Path Interdiction problem to present an algorithm based on Monte Carlo Tree Search idea. This chapter, along with the literature review presented in Chapter 2, aims to provide the necessary research context for this discussion.

Binary decision diagrams (BDDs) are a special class of layered acyclic networks that can be used to represent combinatorial optimization problems (COPs). BDDs contain a root node, `r`, and two terminal nodes: true (**T**) and false (**F**). The BDD is constructed so that

there exists a one-to-one correspondence between **r**-to-**T** paths and solutions to the COP it represents, such that the COP objective and BDD path length are equivalent. Paths from **r** to node **F** in the BDD correspond to infeasible solutions in the corresponding COP. Thus, the COP instance can be solved by creating a BDD that represents the COP instance, and then solving a shortest- (or longest-) path problem over the BDD.

BDDs are instrumental in representing Boolean functions, which arise in computer-aided design (see, e.g., Meinel and Theobald 1998), and increasingly many studies leverage BDDs to solve COPs. However, the size of a BDD that equivalently represents a COP can be exponentially larger than the size of the original problem. This exponential growth can happen due to two reasons. First, it might be intrinsically impossible to represent the COP by a compact (i.e., polynomial-size) BDD. Alternatively, even if it were possible to derive a compact BDD that represents the COP, doing so may be very difficult.

To overcome these difficulties, some COPs can instead benefit from being represented as a collection of multiple BDDs, where each BDD is fairly small in size, as done by Bergman and Cire (2016). The trade-off is that, as we will show, the original COP is now equivalent to solving a set of shortest-path problems over networks corresponding to these BDDs, subject to a set of side constraints. The problem of optimizing this set of jointly-constrained shortest-path problems is called the consistent path problem (CPP), which is itself NP-hard. However, under certain conditions that we discuss in this work, the CPP can be transformed into a single, tractable shortest-path problem. When this transformation is attainable, the COP not only becomes relatively easy to solve, but it also becomes possible to obtain sensitivity analysis on the problem, as we demonstrate with a detailed example presented in Chapter 4.

The conditions needed to make the aforementioned transformation are that the collection of BDDs must have *aligned variable orders*, as defined in the beginning of Chapter 3, and that the component BDDs must be relatively small (measured by the number of nodes). If the collection of BDDs does not have an aligned variable order, then one interesting strategy is to (a) identify a common variable order for the BDD collection and (b) revise the

component BDDs to align each of their variable orders with the identified common variable order. The step of revising BDD variable orders typically changes the size of those BDDs. Hence, the goal is to identify a variable order that minimizes the total size of the BDD collection. This problem is the focus of Chapter 3.

Certain problems (e.g., those studied in Bergman et al. 2016a and Lozano et al. 2020) allow for a "natural" aligned variable order. That is, for those problems, the creation of the collection of BDDs naturally retains a common variable order within the collection. Sometimes the algorithm used to construct the collection implies a representation of different diagrams, in which they can share nodes. It allows to save space and naturally enforces a shared order of variables. However, many other problems imply no natural variable alignments, and no clear mechanism exists to choose one. In such cases, it might be beneficial to revise a collection of diagrams to enforce the same order of layers, either to build a single BDD, or to apply another method that requires this property, such as the one presented by Lozano et al. (2020). We present a specific example of such problem in Chapter 4. In a variant of the Facility Location Problem, assuming facilities can have different *types*, it seems natural to represent two groups of constraints by two different BDDs. The structure of the problem, generally speaking, will dictate different orders of variables in these diagrams. Therefore, we use the algorithm developed in the previous chapter to align them. As we show in a series of numerical experiments, sometimes this allows to reformulate the problem as a linear program of reasonable size and enjoy the corresponding numerical benefits. To sum up, key contributions of the part concerning BDDs include development of a heuristic to align BDDs, which tries to achieve better scaling by avoiding intermediate BDD transformations, and demonstration of a specific computational pipeline and insights into the effects of the problem structure for a combinatorial optimization problem involving this heuristic.

Finally, it may be not necessarily tractable to consider a full decision tree at all. In fact, a binary decision diagram can be exponentially large regardless of the order of variables.[1] In Chapter 5 we consider a case in which building the full tree is practically

[1]This effect is well known in the study of Boolean functions: some functions just yield exponential BDD sizes in any case. (They are called *"very ugly"* then, see Wegener 2000, Definition 5.1.3.)

impossible. Hence, we try to build only a part of it, ignoring those parts that are ultimately unnecessary. Our focus here is the Dynamic Shortest Path Interdiction (DSPI) problem, introduced by Sefair and Smith (2016). The DSPI is a zero-sum game played over a directed weighted graph, where two players, the *Evader* and the *Interdictor*, take steps in turns. The Evader seeks to take a shortest-path between two given nodes, while the Interdictor tries to maximize the cost of this path by *attacking* (or *interdicting*) the arcs. When an arc is interdicted, its cost increases by a pre-defined amount. The Interdictor starts the game, and can interdict any subset of arcs, subject to a cardinality (budget) constraint. The Evader follows and traverses one arc during each turn. The players take turns until the Evader reaches the given *terminal* node.

Potential applications include counter-terrorist activities, infrastructure reliability problems, and vulnerability analysis, as well as natural disaster response, analysis of social effects, and others. Note that it is not forbidden for the user to visit a node, or an arc, more than once. Moreover, it is sometimes optimal to do so, as discussed in an example presented by Sefair and Smith (2016). This complexity implies that for graphs of modest size (tens of nodes) enumerating all game states while building the decision tree becomes impractical for consumer-grade hardware. Following some ideas of Reinforcement Learning research and strategy employed to design a powerful Go playing machine (Silver et al. 2017), we adapt Monte Carlo Tree Search approach. The decision tree in this case is not necessarily a BDD, and we guide its construction with cost estimates obtained from random simulations of the players' turns. We also incorporate tree pruning mechanism using bounds presented in the literature and using a classical approach for two-agent zero-sum games, called alpha–beta pruning. The resulting procedure is a reinforcement learning algorithm in a sense that it does not require a pre-compiled dataset to devise a good solution, but learns parts of the solution by interacting with the environment model and receiving *rewards* (in the form of costs in this case).

The remainder of this dissertation is organized as follows. We provide a brief overview of the relevant literature in Chapter 2. Then, we introduce a problem of aligning two BDDs

and propose several related algorithms in Chapter 3. The following Chapter 4 presents an example of a computational pipeline to solve a variant of the Facility Location problem, using the results obtained in the previous chapter. Then, Chapter 5 focuses on the DSPI and a heuristic algorithm designed using the Monte Carlo Tree Search ideas. Each of the chapters 3–5 concludes with a set of numerical experiments. Finally, we provide a brief note on future research and concluding remarks in Chapter 6.

# Chapter 2

# Literature Overview

Our work builds upon the vast existing research on the decision diagrams, a few sources on DSPI, and general literature on reinforcement learning.

BDD foundations appear in early works of Lee (1959), Akers (1978), Bryant (1986), Brace et al. (1990), and Bryant (1992). See overviews by Wegener (2004), Drechsler and Becker (1998), Meinel and Theobald (1998), and Bryant (2018) for a more recent treatment of this material. Knuth (2009, Section 7.1.4) provides an overview of basic algorithms concerning the manipulation of BDDs. Minato (2013) presents an overview on BDDs and their modifications with notes on applications. There is a growing body of literature on BDDs in the optimization context (Consistent Path Problem and Market Multisplit being some of many examples), as discussed by Bergman et al. (2016a), Bergman et al. (2016b), Hooker (2013), Lozano et al. (2020), and others.

The problem of aligning two diagrams, which we study in Chapter 3, is a natural generalization of the problem seeking to minimize the size of a *single* BDD. Since any pair of adjacent layers can be swapped, any BDD can be revised to an arbitrary order of variables; however, finding one to minimize the BDD size is NP-hard (Bollig and Wegener 1996). Moreover, Sieling (2002) demonstrated that the problem is not approximable.

Heuristic methods seek to improve the BDD size incrementally utilizing swap and sift

operations. The window permutation algorithm discussed by Fujita et al. (1991), Ishiura et al. (1991), and Rudell (1993) exhaustively searches through all permutations of layers within a small consecutive subset of layers (a "window"), moving this window until no more improvements are found. The sifting method by Rudell (1993) (a variant of which we are using as a baseline) processes variables consecutively by sifting each one to all possible positions, other variables being fixed; ultimately, the variable is sifted to the position corresponding to the smallest BDD. The process is stopped after every variable is examined or when no more improvements are found.

Rudell introduced the idea of running "housekeeping" procedures to periodically deflate the diagram size. Drechsler and Günther (2000) further discuss efficient use of the information from the previous sifts. To accelerate this sifting process, Panda and Somenzi (1995) examine sifting variables in groups. Meinel and Slobodová (1997) introduce the idea of "block-restricted sifting": identifying certain blocks within the BDD and restricting variable reordering process to these blocks. Slobodová and Meinel (1998) and Jain et al. (1998) exploit the idea of sampling to determine a good variable order for the whole BDD. Nevo and Farkash (2006) discussed the parallel implementation of Rudell's sifting algorithm. Lin and Wei (2005) report a significant improvement against a benchmark instances set with a randomized algorithm (utilizing random starting variable orders). Friedman and Supowit (1987) give an exact algorithm for the BDD size minimization problem . The complexity of this algorithm is $O(n^2 3^n)$ for a Boolean function of $n$ variables, or $O(n3^n)$ with a modification mentioned by Bollig and Wegener (1996). Drechsler et al. (1998) propose a branch-and-bound search scheme based on this algorithm. Drechsler et al. (2001) present lower bounds for the BDD size, further improved by Ebendt and Drechsler (2006) (by generalizing the ones proposed by Bryant 1991). Ebendt et al. (2005) and Ebendt and Drechsler (2005) present another branch-and-bound technique (combined with an ordered best-first search) and a quality-runtime trade-off. Drechsler et al. (2001) use lower bounds in the dynamic minimization process to introduce the "lb-sifting" method, which speeds up the "normal" sifting without compromising the solution quality (BDD sizes).

There is a vast literature on minimizing the size of a single BDD (see, e.g., overviews by Knuth (2009), Wegener (2000), or many of the works on BDDs mentioned above). Less research has been conducted on the problem of finding a good variable order for a *collection* of BDDs. For example, Cabodi et al. (1998) discuss greedy heuristics based on the minimization of the necessary number of layer swaps and on preserving the longest common partial order. (Note that the *multiple variable order* problem discussed there is equivalent to our AP$(A, B; T^*)$.) The approach of Scholl et al. (2001) starts with one of the initial BDD orders as a target, adjusting this target if the "reordering limit" is reached. However, both studies (along with the literature on single BDD minimization) directly manipulated BDDs, resulting in high computational costs if a bad solution was examined. By contrast, we construct a heuristic based on a problem simplification that allows for computationally cheap incremental improvements. Some discussion of the operations over BDDs and the corresponding bounds on layer sizes, which were an inspiration for our concept of the simplified problem, presented in many sources, including Bollig et al. (1996), Wegener (2000) in Section 5.7, and Knuth (2009) in Section 7.1.4. (In particular, our *sift-up* and *sift-down* operations discussed in Section 3.3 are also referred to as *jump-up* and *jump-down* in the literature.)

Network interdiction, which we deal with in Chapter 5, is a well-established research area. The foundation for our work is the problem of Shortest-Path Interdiction (SPI): a two-stage game, with the attacker first choosing arcs to be interdicted, and the user then minimizing its path cost given the information on the attacker's decision (Israeli and Wood 2002, Fulkerson and Harding 1977). Many variants of the problem are discussed in the literature, including ones with arcs fortification, asymmetric information, multi-objective versions, etc. See, e.g., the chapter on network interdiction by Pardalos et al. (2013) or Smith and Song (2020) for an overview. We consider a specific complication of SPI, viz., a dynamic version of the game, introduced by Sefair and Smith (2016). As opposed to SPI, the DSPI is a multi-stage game. An exact dynamic-programming based algorithm proposed in the aforementioned paper is able to find an optimal solution in polynomial time for a

directed acyclic graph. However, the authors show that for the general case, no polynomial-time algorithm exists (unless $\mathcal{P} = \mathcal{NP}$). To the best of our knowledge, while Sefair and Smith (2016) proposed bounds on the optimal objective, no heuristic algorithm exists in the literature that yields high quality solutions (valid sequences of moves corresponding to the objective values, supplied by the bounds, or the players' policies). We seek to fill in this gap with the algorithm proposed in Chapter 5.

From a methodological perspective, we draw inspiration from the vast and growing research area of Machine Learning (ML) and, in particular, reinforcement learning (RL). There is growing literature on using ML for combinatorial optimization, in various ways: see, e.g., Bengio et al. (2021) for a general overview and Mazyavkina et al. (2021) for one emphasizing RL. We build our algorithm using the Monte Carlo Tree Search (MCTS) framework, which is described in many sources. In particular, Sutton and Barto (2018) give a thorough introduction to RL. Browne et al. (2012) briefly describe the method and summarize many ideas for improving different components of an MCTS algorithm developed in the literature. (These ideas include a specific way for encouraging exploration during the selection process using the ideas from multi-armed bandit literature, introduced by Kocsis and Szepesvári 2006, Kocsis et al. 2006.) Apart from this, there are numerous examples of using this framework for solving various problems, both within the area of game playing and beyond, including Rimmel et al. (2010), Silver et al. (2017), and Karwowski and Mańdziuk (2019). To implement the tree pruning we used a classical idea of alpha–beta pruning, mentioned in many sources, including, e.g., Knuth and Moore (1975).

# Chapter 3

# On Aligning Binary Decision Diagrams

In this chapter we derive a heuristic to close a specific gap in the research literature regarding application of BDDs for optimization, highlighted by Lozano et al. (2020). We start by presenting the necessary definitions on BDDs in Section 3.1 and introducing the BDD alignment problem in Section 3.2. The latter will be the focus of this chapter. Then, we discuss the changes in BDD layer sizes due to their reordering, and propose the auxiliary problem in Section 3.3, which will constitute the core of our heuristic for the alignment problem. We present the underlying technical results and formally present the algorithm to solve the simplified problem in the next Section 3.4. The Chapter concludes with a series of numerical experiments with random instances of the BDD alignment problem, presented in Section 3.5.

## 3.1 Formal definition of BDDs

A BDD $B$ is an acyclic graph having a node set $\mathcal{N}$ that possesses the following properties.

- All nodes in $\mathcal{N}$ are partitioned into $(N+1)$ layers. We denote the $i$-th layer $L_i$ (or

$L_i^B$, if we need to specify that this layer belongs to $B$). We refer to $i$ as a *layer index*. The *layer width* is the number of nodes in the layer, and the *diagram size* $|B|$ is the sum of layer widths.

- The first layer contains the single root node (denoted by $\mathtt{r}$), and the last layer comprises two terminal nodes, referred to as *true* ($\mathbf{T}$) and *false* ($\mathbf{F}$) nodes.
- Layers are ordered, and layer index $i = 1, \ldots, N$ is associated with a unique *variable*, $\text{var}(L_i)$. We let $\text{var}(B)$ denote the ordered list of variables for $B$. (When $\text{var}(B) = \mathbf{v}$, we say that $B$ "has variable order $\mathbf{v}$.")
- We say that variable $a$ has *position* $i_B(a)$ in $B$ if $a = \text{var}(L_{i_B(a)})$. For convenience, we will write $a \prec_B b$ to mean $i_B(a) < i_B(b)$ ("$a$ appears in $B$ before $b$").
- A node outside the last layer has exactly two outgoing arcs, pointing to nodes in the next layer. One is designated as a "one-arc" and the other as a "zero-arc." Nodes $\mathbf{T}$ and $\mathbf{F}$ have no outgoing arcs.
- A node outside the first layer has at least one incoming arc (emanating from a node on the previous layer); $\mathtt{r}$ has no incoming arcs.
- For convenience, we define a "path" to be a path from $\mathtt{r}$ to $\mathbf{T}$ or $\mathbf{F}$ and a "subpath" to be any consecutive subsequence of nodes in a path.

Associating each layer of a BDD with a binary variable, every path in $B$ encodes a vector of binary variables corresponding to a solution to the underlying problem represented by the BDD (a feasible one if the path ends at $\mathbf{T}$, or an infeasible one otherwise). Optimization problem solutions correspond to BDD paths, such that a path uses a one-arc (zero-arc) at layer $L_i,\ i \in \{1, \ldots, N\}$, if and only if $\text{var}(L_i)$ equals 1 (respectively, 0) in the solution. All feasible (respectively, infeasible) solutions to the optimization problem correspond to BDD paths that terminate at $\mathbf{T}$ (respectively, $\mathbf{F}$).

*Remark* 3.1. In much of this and the next chapter, we will seek to reduce the size of BDDs by merging redundant node pairs where possible. Suppose that there exists a pair of nodes

$u$ and $v$ in $\mathcal{N}$, such that one-arcs of $u$ and $v$ point to the same node and zero-arcs of $u$ and $v$ point to the same node (and respective arc weights for $u$ and $v$ coincide, if applicable). In that case, nodes $u$ and $v$ can be merged. A BDD that has no such pair of nodes $u$ and $v$ is said to be *quasi-reduced* (which is equivalent to the definition by Wegener 2000). Our methods do not require the BDDs we consider to be quasi-reduced; however, all our computations will operate over quasi-reduced BDDs to remove all redundant node pairs. □

**Example 3.1.** Figure 3.1 illustrates the process of modeling a maximum independent set (MIS) instance with a BDD. MIS seeks a largest subset of nodes such that no two nodes in the subset are adjacent. We first associate a binary variable with every node in the MIS graph (Figure 3.1a), order these nodes (arbitrarily), and associate each variable with a BDD layer. The last layer comprises nodes **T** and **F**, with all paths corresponding to feasible (infeasible) solutions ending at **T** (**F**), as desired. Each such path consists of five arcs, with one arc outgoing from each of the first five layers. The arc lengths reflect the solution's objective by assigning a unit length to each one-arc and a zero length to each zero-arc.

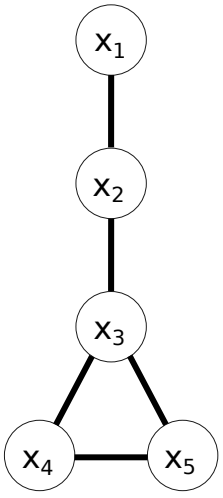
Let binary variables $x_i$ determine if node $i$ of the MIS graph belongs to the independent set, and consider the order $(x_1, x_2, x_3, x_4, x_5)$. Figure 3.1b displays a BDD constructed from this ordering. Note, for example, that BDD node 5 in Figure 3.1b indicates that one-arcs have been selected in layers corresponding to $x_1$ and $x_2$, so $x_1 = x_2 = 1$. Because nodes 1 and 2 are adjacent in the MIS graph, all subpaths starting from BDD node 5 in Figure 3.1b terminate at **F**.

Constructing the BDD with respect to another MIS variable order, $(x_1, x_4, x_3, x_2, x_5)$, yields the larger BDD depicted in Figure 3.1c. Because the MIS can be solved by identifying a longest-path from `r` to **T** in the BDD, requiring effort proportional to the number of nodes in the BDD, we seek a variable order that yields the smallest possible BDD. □

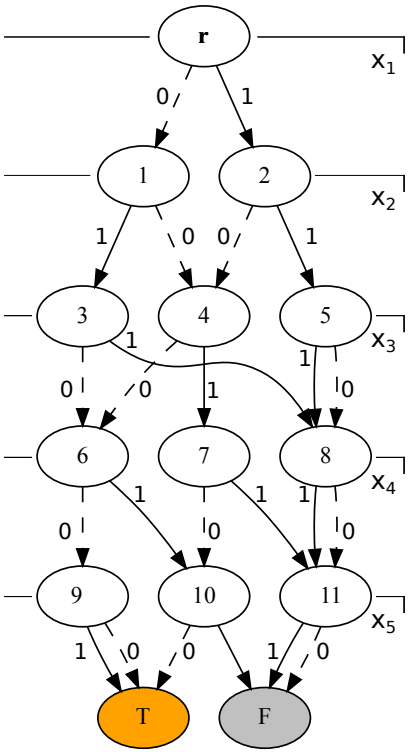
To explore this notion further, we formally introduce the concept of equivalent BDDs.

**Definition 3.1.** BDDs $B$ and $B'$ are *equivalent* if for every path in $B$ (or $B'$) there exists

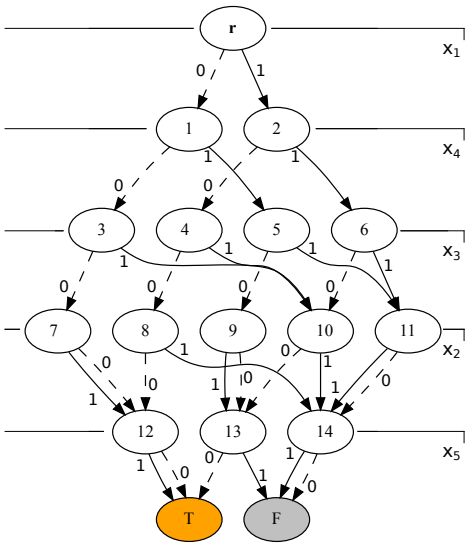
Figure 3.1: BDD representation for max independent set




(a) Initial graph




(b) $\mathbf{v} = (x_1, \ldots, x_5)$ : 14 nodes




(c) $\mathbf{v} = (x_1, x_4, x_3, x_2, x_5)$ : 17 nodes

*Note.* One-arcs (zero-arcs) are depicted with solid (respectively, dashed) lines and given unit (respectively, zero) lengths.

a path in $B'$ (or $B$) that corresponds to the same variable assignment and ends at the same terminal node ($\mathbf{T}$ or $\mathbf{F}$).

*Remark* 3.2. Where possible, we will also ignore BDD arc lengths in this chapter to simplify our exposition. Our subsequent results easily extend to problems having arc weights, and the source code implemented for this dissertation accommodates weighted BDDs as well. □

Finally, note that all definitions presented for BDDs here are compatible with the optimization-related research we build upon, mostly by Lozano et al. (2020). The diagram we define is effectively different from the classical definition of the BDD as a *reduced function graph* by Bryant (1986) in the following aspects:

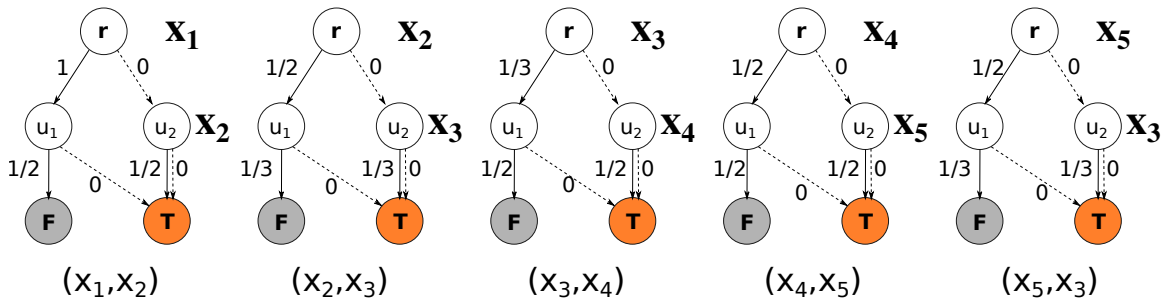
- We do not enforce the condition that one- and zero-arcs must point to distinct nodes. Therefore, our BDDs are not generally *reduced* (as per definition by Bryant 1986).
- We restrict BDD arcs to connect nodes on adjacent layers only. Therefore, we consider only *complete* ordered BDDs in terms of Definition 3.2.1 by Wegener (2000): all paths include $N$ arcs.

While such diagrams are general enough to represent feasible sets for COPs, some problems might benefit from leveraging more specialized decision diagram-type data structures. See, e.g., more general notes on diagrams by Knuth (2009), a more focused discussion of zero-suppressed BDDs (ZDDs) by Minato (2013), or a comprehensive overview by Wegener (2000), to name a few sources.

## 3.2 Collection of BDDs and the alignment problem

The problem of finding a variable order that minimizes the diagram size is NP-hard (Bollig and Wegener 1996); moreover, this minimum size could still be extremely large. An alternative way to limit diagram sizes is based on the idea of representing the problem with

Figure 3.2: BDD collection representation for max independent set




*Note.* Representation of the independent set problem depicted in Figure 3.1a with a collection of BDDs.

a collection of multiple BDDs, as proposed by Bergman and Cire (2016). We illustrate this representation in the following example.

**Example 3.2.** For the MIS example, one could design a collection of BDDs as in Figure 3.2 (adapted from Lozano et al. 2020). For each MIS edge $(x_i, x_j)$, create a separate diagram with two layers associated with variables $x_i$ and $x_j$, plus a terminal layer, enforcing the condition that no two adjacent nodes belong to the independent set. Arc lengths in BDDs are chosen to ensure that setting $x_i = 1$ for any $i$ yields a total contribution of one to the objective (and zero otherwise).

Now, note that these diagrams (one for every edge in the original graph) have overlapping variables. We seek a collection of `r`-to-**T** paths, one per diagram, that has the maximum sum of lengths and is consistent in the following sense: For each variable, all paths that include an arc corresponding to this variable must share the same arc type. For example, if we have a zero-arc in the $x_2$ layer in a path for some diagram, all paths in the collection that involve $x_2$ must include a zero-arc for that layer. This problem is the CPP. Critical to this study, the CPP is NP-hard if the number of diagrams is not fixed, or even with just two BDDs if the diagrams do not share the same order of variables (Lozano et al. 2020).

We address the problem of *aligning* a pair of BDDs that are (without loss of generality) defined over the same set of variables, but have a different variable order. The alignment

problem aims to find a common variable order such that after revising both BDDs to this variable order, the total number of nodes in both diagrams is minimized. Since the problem is NP-hard by extension from Bollig and Wegener (1996), we aim to create a heuristic.

Our heuristic approach creates an auxiliary, simplified problem. As we will show, determining the impact of changing a variable order on the corresponding BDD size might involve significant computational cost, because diagram sizes can grow exponentially. We introduce a way to quickly update upper bounds on layer widths after swapping variables corresponding to adjacent BDD layers, which naturally yields a heuristic of minimizing the upper bound on the objective. We will refer to labeled arrays of such upper bounds as *weighted variable sequences*. The following definitions help to define the BDD alignment problem formally.

**Definition 3.2.** An *optimal transformation* of BDD $A$ to variable order $\mathbf{v}$, denoted by $T^*[A, \mathbf{v}]$, is defined as a smallest BDD equivalent to $A$ that has variable order $\mathbf{v}$.

**Definition 3.3.** Given two BDDs $A$ and $B$ defined over the same variable set, the *alignment problem* solves:

$$s^* = \min_{\mathbf{v}}(|T^*[A, \mathbf{v}]| + |T^*[B, \mathbf{v}]|), \tag{AP}$$

where $\mathbf{v}$ belongs to the set of all possible permutations of the labels in the variable list, $\text{var}(A)$. We refer to this problem as the "original" one, as opposed to a "simplified" version considered later, and parameterize it as $\text{AP}(A, B; T^*)$. Note that we examine the alignment of only two diagrams for the sake of readability; the problem can also be easily generalized to the alignment of several BDDs. Also, as we mentioned before, the requirement to have the same variable set is not restrictive: If a variable appears in BDD $A$, but not in $B$, it is simply ignored in $B$ (and thus, can be trivially introduced at any position).

## 3.3 The simplified problem: definition

To illustrate the motivation behind the core concept of weighted variable sequence (introduced in Section 3.3.1), we briefly introduce the operations required to transform a BDD $B$, into an equivalent BDD $R(B)$, having a different variable order. Since both sift-down and swap can be expressed via sift-up operations, we examine the sift-up of a layer $L_s$ from source position $s$ to destination position $d < s$. Note that these operations require polynomial space and time in the size of the underlying diagram (Wegener 2000).

The transformed network $R(B)$ will be identical to $B$ in layers $L_1, \ldots, L_d$ and in layers $L_{s+1}, \ldots, L_{N+1}$, with the caveat that the variable associated with index $d$ in $R(B)$ changes from $\text{var}(L_d^B)$ to $\text{var}(L_s^B)$. A key challenge is that after the sift-up operation, we must now create arcs that connect nodes in the layer corresponding to $\text{var}(L_{s-1}^B)$ to nodes in the layer corresponding to $\text{var}(L_{s+1}^B)$, which are now adjacent in $R(B)$. However, the destinations of these arcs depend on the choice of value for $\text{var}(L_s^B)$ in $R(B)$.

**Example 3.3.** Consider the BDD depicted in Figure 3.3 and a sift-up operation with source variable $x_4$ and destination variable $x_2$. Consider node 9 in Figure 3.3: Following the zero-arc should lead to node 18 if $x_4 = 1$, and to node 19 if $x_4 = 0$. Therefore, creating such arcs in our transformation must incorporate the previous choice of the value for $x_4$.

Our transformation therefore generates two copies of the BDD between layers $L_d^B$ and $L_{s-1}^B$ (inclusive) in the transformation, with the purpose of retaining the value of $\text{var}(L_s^B)$. The overall process is as follows.

- $R(B)$ is the same as $B$ through layers $1, \ldots, d$ and $s+1, \ldots, N+1$.
- Two copies of $B$ between layers $L_d^B$ and $L_{s-1}^B$ (inclusive), a one-copy and a zero-copy, are generated and placed in parallel in $R(B)$, where for each copy, the nodes corresponding to layer $L_i^B$ of $B$ are now in layer $L_{i+1}^{R(B)}$ in $R(B)$. One-arcs from the nodes in $L_d^{R(B)}$ point to the corresponding nodes in the one-copy, zero-arcs to the nodes in the zero-copy.

- Nodes in layer $L_s^{R(B)}$ are connected to $L_{s+1}^{R(B)}$ in $R(B)$ according to the corresponding choice of $\text{var}(L_s^B)$ (i.e., whether the node is in the one-copy or zero-copy of the network) and $\text{var}(L_{s-1}^B)$, as detailed in Algorithm A3.1.

Figure 3.4 illustrates the final step in which, for example, the zero-arc from $9'$ connects to node 18, because $9'$ belongs to the one-copy of the network, signifying that $x_4 = 1$, and reflecting the path $9 \to 14 \to 18$ in $B$ (Figure 3.3). Similarly, the zero-arc from $9''$ connects to node 19, because $9''$ belongs to the zero-copy of the network ($x_4 = 0$), reflecting the path $9 \to 14 \to 19$ in $B$. □

Importantly, this duplication permits us to retain the value of $\text{var}(L_s^B)$, at the expense of duplicating the layer widths between $s$ and $d$ in the worst case. Nodes in layer $L_s^{R(B)}$ are connected to $L_{s+1}^{R(B)}$ in $R(B)$ according to the corresponding choice of $\text{var}(L_s^B)$ (i.e., whether the node is in the one-copy or zero-copy of the network) and $\text{var}(L_{s-1}^B)$.

**Algorithm A3.1.** Sift-up (BDD).

**Input:** a BDD $B$ with $\text{var}(B) = (v_1, \ldots, v_N)$; source layer index $s \in (1, N]$ and destination layer index $d < s$

**Output:** a transformed BDD $R(B)$ with $\text{var}\Big(R(B)\Big) = (v_1, \ldots, v_{d-1}, v_s, v_d, v_{d+1}, \ldots, v_{s-1}, v_{s+1}, \ldots, v_N)$

**Step 1.** Initialize $R(B)$:

- Create $R(B)$ as a duplicate of $B$.
- Duplicate layer $L_d^{R(B)}$ (including outgoing arcs) and insert it after $L_d^{R(B)}$ as the new layer $L_{d+1}^{R(B)}$.

**Step 2.** Label the group of all nodes in layers $L_{d+1}^{R(B)}, \ldots, L_{s+1}^{R(B)}$ as `sD0`. Reassign zero-arcs emanating from every node in $L_d^{R(B)}$ to point to its corresponding node in layer $L_{d+1}^{R(B)}$ of `sD0`.

**Step 3.** Duplicate the `sD0` group including all the arcs and respecting layer labels. Label the new group as `sD1`. Reassign one-arcs emanating from every node in $L_d^{R(B)}$ to point to its corresponding node in layer $L_{d+1}^{R(B)}$ of `sD1`.

**Step 4.** Reassign the destination layer variable $\text{var}(L_d^{R(B)}) \leftarrow \text{var}(L_s^B)$.

**Step 5.** For every node $n \in L_d \cap$ `sD0`, reassign its outgoing arcs as follows. Identify nodes $j_0(n)$ and $j_1(n)$ reached by starting at $n$ and following:

- for $j_0(n)$: two consecutive zero-arcs;
- for $j_1(n)$: the one-arc, followed by the zero-arc.

Reassign the tail of the zero- (respectively, one-) arc emanating from $n$ to $j_0(n)$ ($j_1(n)$).

**Step 6.** For every node $n \in L_d \cap$ `sD1` reassign the outgoing arcs as follows. Identify nodes $j_0(n)$ and $j_1(n)$ reached by starting at $n$ and following:

- for $j_0(n)$: the zero-arc followed by the one-arc;
- for $j_1(n)$: two consecutive one-arcs.

Reassign the tail of the one- (respectively, zero-) arc emanating from $n$ to $j_1(n)$ ($j_0(n)$).

**Step 7.** Remove layer $L_{d+1}^{R(B)}$ (which now has no incoming arcs).

This procedure is formally presented in Algorithm A3.1. It returns diagram $R(B)$ that has the correct variable order by construction, and $B$ and $R(B)$ are equivalent since

Figure 3.3: Diagram $B$ (before the sift-up)

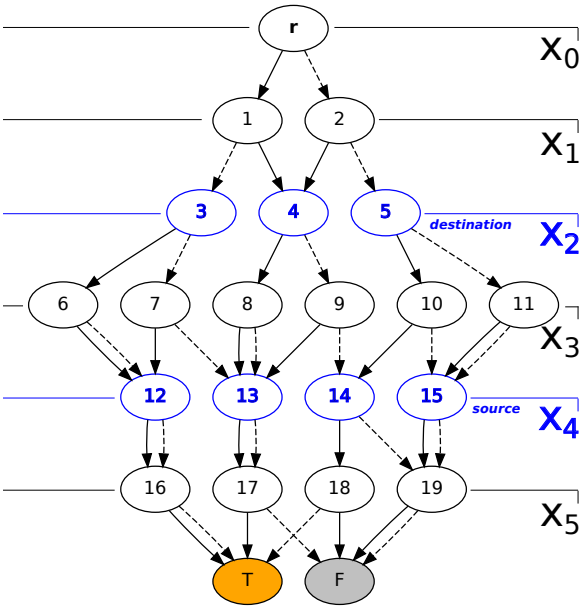
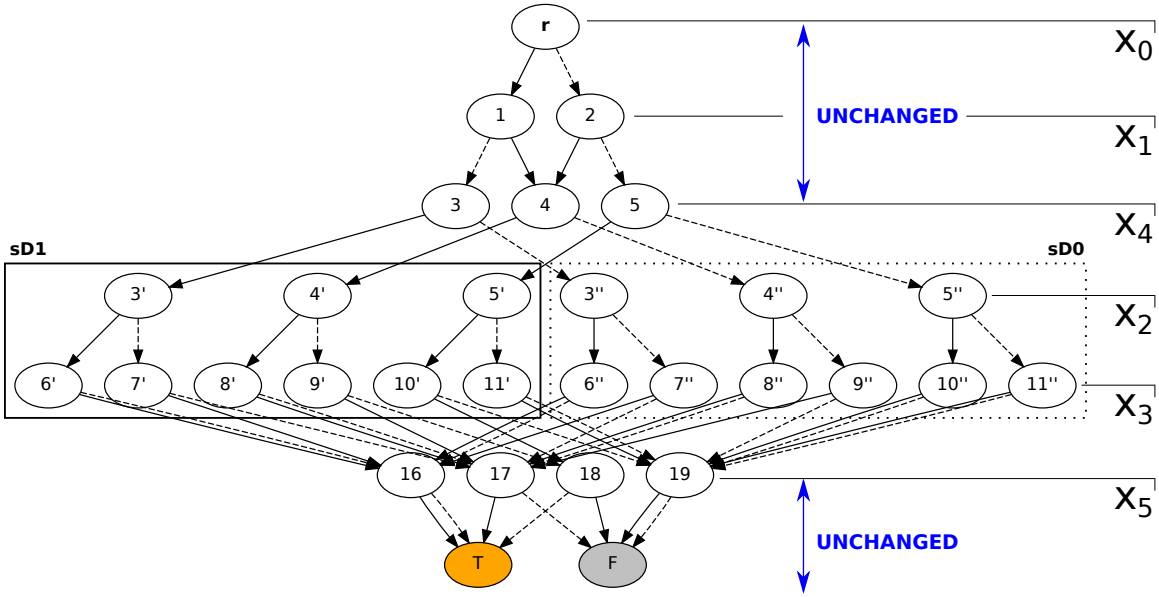



Figure 3.4: Diagram $R(B)$ (after the sift-up)




*Note.* The resulting transformation after sifting of the source layer $x_4$ to the position immediately after $x_2$.

paths corresponding to the same variable assignment pass through the same nodes in the $L_s$ layer. (Reassigning arc lengths to preserve path lengths is also straightforward, if one chooses to implement a weighted version of the algorithm.) Also, if we use hash tables to access nodes by IDs in constant time, it takes $O(|\mathcal{N}|)$ operations to perform a sift-up (in the worst case the whole BDD will be duplicated if we sift-up the last layer to the first position).

Finally, Algorithm A3.1 might result in the creation of redundant node pairs, as defined in Remark 3.1. Continuing with the previous example, both one-arcs and zero-arcs of nodes $11'$, $10''$, $11''$ point to node 19 in Figure 3.4. Hence, the resulting diagram is not quasi-reduced. These three nodes can be merged. In our numerical experiments we remove such redundancy by considering exclusively *quasi-reduced* diagrams in terms of Wegener 2000: minimal size complete diagrams. This is equivalent to forbidding the "redundant nodes" by ensuring that for all $u \in \mathcal{N}$, there is no such $v \in \mathcal{N} \setminus \{u\}$ that one-arcs of $u$ and $v$ point to the same node and zero-arcs of $u$ and $v$ point to the same node (and respective arc weights for $u$ and $v$ coincide, if we implement a weighted version). Note that this diagram will be still not necessarily *reduced*, because it can contain nodes with only one descendant.

Starting from a quasi-reduced diagram, we can preserve this property in a course of the diagram transformations by incorporating this idea into the implementation of *swap* operation. The necessary logic can be derived from the discussion of the reduced diagrams (and canonical arc costs, if we are implementing the weighted version) by Hooker (2013) or adapted from Knuth (2009). The swap operations are then adjusted as presented in Algorithm A3.2 (arc lengths are ignored for readability). We iterate over the nodes of the upper layer, $L_{s-1}$, in lines 3–20. For each outgoing arc, we examine if a target node (with the specified zero- and one-arcs) already exists in layer $L_s$. If it does, we point the outgoing arc to this node (line 9). Otherwise, a new node is created in $L_s$ (line 6). We use a dictionary (`newNodes`) to make sure that only unique nodes are created in the layer $L_s$. An example for this algorithm is given in Figure 3.5, the diagram before the swap is presented in the left panel, and the updated diagram is on the right. Assume we are swapping $x_1$ and $x_2$. Denote one-arc leaving node $u$ as $\texttt{HI}(u)$ and zero-arc leaving node $u$ as $\texttt{LO}(u)$. Consider node

*F1* after the swap: its one-arc must point to a node $n_1$ with $\texttt{HI}(n_1) = 1$ and $\texttt{LO}(n_1) = 2$ (determined just by examining outgoing arcs), and its zero-arc must point to a node $n_2$ with $\texttt{HI}(n_2) = 2$ and $\texttt{LO}(n_2) = 3$. Neither node exists during the processing of node *F1*, so both are created and linked to *F1*. Then we repeat the procedure for node *F2*. $\texttt{HI}(F2)$ must have a one-arc pointing to 1 and a zero-arc pointing to 2. This node ($n_1$) already exists, so *F2* will be linked to it with a one-arc. The process is repeated for the zero-arc of *F2*, resulting in a diagram that is smaller than the original one. Therefore, if the diagram was quasi-reduced before the swap, it keeps this property after the operation is performed according to this implementation.

**Algorithm A3.2.** Swap (quasi-reduced BDD)

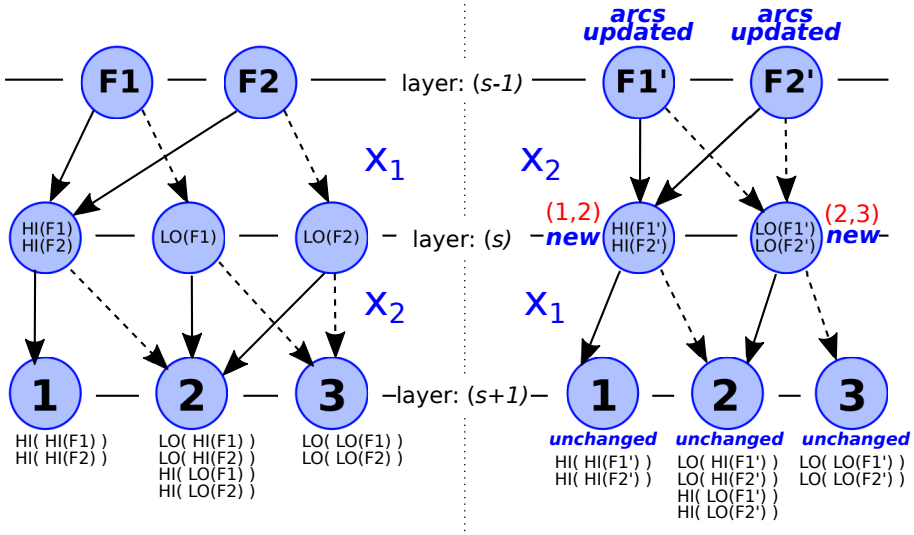
**Input:** a quasi-reduced BDD $B$; a layer number $s$ to swap *up*
**Output:** A (quasi-reduced) transformed diagram with layers $s$ and $s-1$ swapped

```
 1: procedure SWAP-QR(B, s)
 2:     newNodes ← {∅ : ∅}  // A dictionary, with O(1) access operations
 3:     for all nodes F ∈ L_{s-1} do  // iterate over a nodes of the destination (upper) layer
 4:         Adjust one-arc: // Creating a node if necessary
 5:             if (HI(HI(F)), HI(LO(F))) ∉ keys of newNodes then
 6:                 F_HI ← create new node with HI(F_HI) = HI(HI(F)) and LO(F_HI) = HI(LO(F)))
 7:                 add {(HI(F_HI), LO(F_HI)) : F_HI} to newNodes
 8:             else
 9:                 F_HI ← a node from newNodes corresponding to key (HI(HI(F)), HI(LO(F)))
10:             end if
11:         Adjust zero-arc:  // Create a node if necessary for the other arc
12:             if (LO(HI(F)), LO(LO(F))) ∉ keys of newNodes then
13:                 F_LO ← create new node with HI(F_LO) = LO(HI(F)) and LO(F_LO) = LO(LO(F))
14:                 add {(HI(F_LO), LO(F_LO)) : F_LO} to newNodes
15:             else
16:                 F_LO ← a node from newNodes corresponding to (LO(HI(F)), LO(LO(F)))
17:             end if
18:             HI(F) ← F_HI
19:             LO(F) ← F_LO
20:     end for
21: end procedure
```

### 3.3.1 The simplified problem

We first introduce a data structure that keeps track of the upper bounds on layer widths as we perform layer sifts.

Figure 3.5: A swap-up operation (as per Algorithm A3.2)




*Note.* Swapping layers $s$ and $(s-1)$. Changes are marked with bold text outside the nodes: `F1` and `F2` nodes get their outgoing arcs updated; layer $s$ is rebuilt, layer $(s+1)$ remains unchanged (as well as all the layers above $s$). The algorithm guarantees that each node in the newly built layer has a unique (`HI`,`LO`) tuple (depicted as numbers in parentheses outside of newly created nodes). Note that there are only two nodes in the resulting layer $s$: the total number of nodes has decreased.

Define a *weighted variable sequence* over variables $x_1, \ldots, x_N$, denoted by $S = [x_1, \ldots, x_N | n_1, \ldots, n_N]$, as an ordered list of $N$ variable names $\mathrm{var}(S) = (x_1, \ldots, x_N)$ together with associated integer weights $(n_1, \ldots, n_N)$, $n_i \in \{1, \ldots, 2n_{i-1}\}$, $i = 2, \ldots, N$. Similarly to BDDs, we define the sequence size as $|S| = \sum n_i$. A variable-weight pair $(x_i, n_i)$ is a sequence *element* (for convenience, we refer to $i_S(x)$ as the *position* of $x$ in $\mathrm{var}(S)$; e.g., for $S = [x_1, x_3, x_2 | n_1, n_3, n_2]$ the position $i_S(x_2) = 3$).

Our problem simplification captures the worst-case change in the diagram size due to sifts, wherein all layer widths between the source and the destination are duplicated. (For instance, recall the doubling of layers corresponding to $x_2$ and $x_3$ in Figures 3.3 and 3.4.) We focus on the swap operation, which interchanges adjacent variable positions and duplicates the destination element weight. Sift operations can be represented as a series of swap operations. For convenience, we denote the variable sequence before the operation as $S =$

$[x_1, \ldots, x_N | n_1, \ldots, n_N]$, and after the operation as $S' = [x'_1, \ldots, x'_N | n'_1, \ldots, n'_N]$.

The operations over a weighted variable sequence are described as follows.

- $\texttt{swap}(S, i)$**:** exchanges positions of two adjacent variables at positions $i$ and $i+1$. Labels are adjusted as $x'_i = x_{i+1}$ and $x'_{i+1} = x_i$, and weights are updated as $n'_i = n_i$ and $n'_{i+1} = 2n_i$. Other weights and labels are unchanged: $x'_j = x_j$ and $n'_j = n_j$ for all $j \notin \{i, i+1\}$.
  **Example:** $[x_1,\ \overset{\curvearrowleft}{x_2},\ x_3,\ x_4 | n_1, n_2, n_3, n_4] \to [x_1, x_3, x_2, x_4 | n_1, n_2, 2n_2, n_4]$.

- $\texttt{sift}(S, s, d)$**:** changes position of variable $x_s$ from $s$ to $d$ (with the relative positions of all other variables unchanged), defined as a series of consecutive $\texttt{swap}(S, i)$ operations with $i = s-1, \ldots, d$ if $d < s$ and $i = s, \ldots, d-1$ if $d > s$.
  **Examples:**
  $[x_1, \overset{\frown}{x_2, x_3}, x_4 | n_1, n_2, n_3, n_4] \to [x_1, x_4, x_2, x_3 | n_1, n_2, 2n_2, 2n_3]$;
  $[\overset{\frown}{x_1, x_2, x_3}, x_4 | n_1, n_2, n_3, n_4] \to [x_2, x_3, x_1, x_4 | n_1, 2n_1, 4n_1, n_4]$.

*Remark* 3.3. Sequence $S' = \texttt{sift}(S, s, d)$ can be constructed in $O(N)$ time as follows. For a sift-up $(d < s)$ : $n'_i = 2n_{i-1}$ if $d < i \leq s$ and $n'_i = n_i$ otherwise. For a sift-down $(d > s)$ : $n'_i = 2^{(i-s)} n_s$ for $s < i \leq d$ and $n'_i = n_i$ otherwise. □

Similarly to the discussion for BDDs above, we now define an *optimal transformation* $T^*_{\mathrm{VS}}[S, \mathbf{v}]$ as a variable sequence with variable order $\mathbf{v}$ of minimal size that can be obtained from $S$ by applying a series of swap operations.

*Remark* 3.4. Consider a swap operation that exchanges elements $i$ and $i+1$. After the swap, the sum of affected elements' weights in the new sequence becomes $n_i + 2n_i \geq n_i + n_{i+1}$, since $2n_i \geq n_{i+1}$. Hence, the total size of the sequence never decreases with a swap, i.e., $|\texttt{swap}(S, i)| \geq |S|$ for all $i \in \{1, \ldots, N-1\}$. Consequently, $|T^*_{\mathrm{VS}}[S, \mathbf{v}]| \geq |S|$. □

These concepts allow us to introduce a simplified alignment problem as follows.

**Definition 3.4.** Given the original problem AP$(A, B; T^*)$, define variable sequences $S_A$

and $S_B$ with variable lists $\text{var}(S_A) = \text{var}(A)$ and $\text{var}(S_B) = \text{var}(B)$. Define element weights to equal the corresponding layer widths: $n_i^A = |L_i^A|$ and $n_i^B = |L_i^B|$ for all $i = 1, \ldots, N$. Overloading the notation, we refer to the problem $\text{AP}(S_A, S_B; T^*_{\text{VS}})$ as the *simplified problem* for $\text{AP}(A, B; T^*)$.

Weighted variable sequences yield upper bounds on diagram sizes in the following sense.

**Lemma 3.1**

Given BDD $A$ and its corresponding sequence $S$ (constructed as per Definition 3.4), consider BDD $A' = \texttt{swap}(A, i)$ and sequence $S' = \texttt{swap}(S, i)$. Then $|S'| - |S| \geq |A'| - |A|$, which implies that for any variable order $\mathbf{v}$: $|T^*[A, \mathbf{v}]| \leq |T^*_{\text{VS}}[S, \mathbf{v}]|$.

Lemma 3.1 states that the swap operation increases the size of a sequence by at least as much as it increases the size of the corresponding BDD. This fact can be proven by contradiction, since the opposite would yield a schedule of swaps implying a transformation of $A$ that would be strictly smaller than $T^*[A, \mathbf{v}]$. (The formal proof is omitted for brevity.)

**Corollary 3.2**

The optimal objective $s^*$ of $\text{AP}(A, B; T^*)$ is no more than the optimal objective $s^*_{VS}$ for $\text{AP}(S_A, S_B; T^*_{\text{VS}})$.

*Proof.* Assume the contrary is true. If there is an optimal variable order shared between $\text{AP}(A, B; T^*)$ and $\text{AP}(S_A, S_B; T^*_{\text{VS}})$, then the claim immediately follows from Lemma 3.1. Otherwise, there must exist an optimal variable order $\mathbf{u}$ for $\text{AP}(S_A, S_B; T^*_{\text{VS}})$ corresponding to the optimal value $s^*_{\text{VS},\mathbf{u}} = |T^*_{\text{VS}}[S_A, \mathbf{u}]| + |T^*_{\text{VS}}[S_B, \mathbf{u}]| < s^*$. But due to Lemma 3.1, $|T^*_{\text{VS}}[S_A, \mathbf{u}]| \geq |T^*[A, \mathbf{u}]|$ and $|T^*_{\text{VS}}[S_B, \mathbf{u}]| \geq |T^*[B, \mathbf{u}]|$. Hence, $s^*_{\text{VS},\mathbf{u}} \geq |T^*[A, \mathbf{u}]| + |T^*[B, \mathbf{u}]| \geq s^*$, which is a contradiction. □

Note that complexity of the swap and sift-up operations for variable sequences is $O(1)$ and $O(N)$, respectively, while for BDDs these operations are $O(|L_i|)$ and $O(|\mathcal{N}|)$.

(Moreover, both $|L_i|$ and $|\mathcal{N}|$ can be exponential in $N$.) This justifies the idea of exploiting the simplified problem to obtain an upper bound.

To derive several key properties of an optimal solution that will serve a foundation for our branch-and-bound algorithm, we obtain some insight in this section into how an optimal transformation can be built. We first note that a certain structure in variable sequence allows for changes in the variable order at no additional cost. Then we introduce a concept of *non-redundant schedule of swaps*, which is, essentially, a sequence of swaps without any steps that can be safely removed. These two concepts allow us to prove the existence of an optimal solution that possess certain properties that makes it easier to find (by constructing such solution from an arbitrary alternative optimum). In particular, in the next section we establish restrictions on the first element, last element, and potentially on the mutual order of some other elements as well.

We introduce *exponentially weighted subsequence* to denote a continuous subsequence of elements spanning positions $k$ to $l$, $k < l$, such that $n_i = 2n_{i-1}$ for all $i = (k+1), \ldots, l$.

**Lemma 3.3**

Consider an exponentially weighted subsequence of $S$ spanning positions $k, \ldots, l$. An exponentially weighted subsequence spans positions $k, \ldots, l$ in $S' = \texttt{swap}(S, i)$ for any $i \in \{k, \ldots, (l-1)\}$ and positions $k, \ldots, (l+1)$ in $S' = \texttt{swap}(S, l)$.

*Proof.* Consider the case for $k \leq i < l$. Affected elements' weights before the swap are $n_i$ and $n_{i+1} = 2n_i$ (since both elements belong to an exponentially weighted subsequence). After the swap the weights become $n'_i = n_i$ and $n'_{i+1} = 2n_i = n_{i+1}$ (no change), so the subsequence is still an exponentially weighted one. For $i = l$, before the swap we have $n_l = 2^{(l-k)} n_k$. After the swap we have $n'_l = n_l$ and $n'_{l+1} = 2n_l = 2^{(l-k)+1} n_k$, so the exponentially weighted subsequence now includes position $(l+1)$. □

**Corollary 3.4**

Consider a variable sequence $S$ with $\text{var}(S) = \mathbf{v}$. If an exponentially weighted subsequence spans positions $k, \ldots, l$, then $|S| = |T^*_{\text{VS}}[S, \mathbf{u}]|$ for any $\mathbf{u}$ such that $u_i = v_i$ for all $i < k$ and all $i > l$.

We will say that a pair of elements $a, b \in \text{var}(S)$ *appears in a schedule of swaps* if there exists a $\texttt{swap}(S, i)$ in the schedule such that either $x_i = a,\ x_{i+1} = b$ or $x_i = b,\ x_{i+1} = a$. A *non-redundant* schedule of swaps is one in which each pair appears at most once.

**Lemma 3.5**

For a variable sequence $S$ and an arbitrary permutation of $\text{var}(S)$, denoted $\mathbf{v}$, there exists a non-redundant schedule of swaps that yields $T^*_{\text{VS}}[S, \mathbf{v}]$.

*Proof.* First, note that $T_{\text{VS}}[S, \mathbf{v}] \neq \varnothing$ (since any two adjacent elements can be swapped), and $|T_{\text{VS}}[S, \mathbf{v}]|$ is finite, because there is a finite number of valid variable sequences with variable order $\mathbf{v}$. Hence, there is an optimal schedule of swaps that yields $T^*_{\text{VS}}[S, \mathbf{v}]$. If the schedule is non-redundant, then the claim is proven. Otherwise, suppose that pair $\{a, b\}$ appears more than once in the schedule. Now, modify the schedule by ignoring all swaps between $a$ and $b$ (except for the last one, if the number of occurrences is odd), and perform the same schedule of swaps in terms of element indices. The new schedule produces the same variable order with some swaps removed. Due to Remark 3.4, the resulting sequence will have size at most $|T^*_{\text{VS}}[S, \mathbf{v}]|$ (hence, exactly the optimal value). Repeating the procedure for each redundant pair of swaps, we obtain a non-redundant optimal schedule. □

To build a non-redundant schedule of swaps that yields an optimal transformation, examine a specific swapping strategy presented in Algorithm A3.3. Without loss of generality, assume elements are labeled as $1, \ldots, N$, and the target order is $\mathbf{v} = (1, \ldots, N)$. At every iteration of the algorithm, we find the element with the largest label that is not in its final position (line 3). Indexing this element as $N_f$, note that elements in positions $(N_f + 1), \ldots, N$ cannot participate in any further swaps (otherwise, they would belong to a

redundant swap). Therefore, we focus on the subsequence spanning positions $1, \ldots, N_f$.

We perform the corresponding sift in line 4. Note that in any non-redundant schedule this element will not be swapped further with any one having index less than $i_{S'}(N_f)$ or more than $N_f$. Moreover, this element also must be swapped with all elements occupying positions $(i_{S'}(N_f)+1), \ldots, N_f$. Since we do not perform any other swaps with this element, then due to Remark 3.4, the weights of elements in positions $i_{S'}(N_f), \ldots, N$ after the sift represent a lower bound on the corresponding weights in $T^*_{\mathrm{VS}}[S, \mathbf{v}]$.

**Algorithm A3.3.** Align-to (weighted variable sequence)

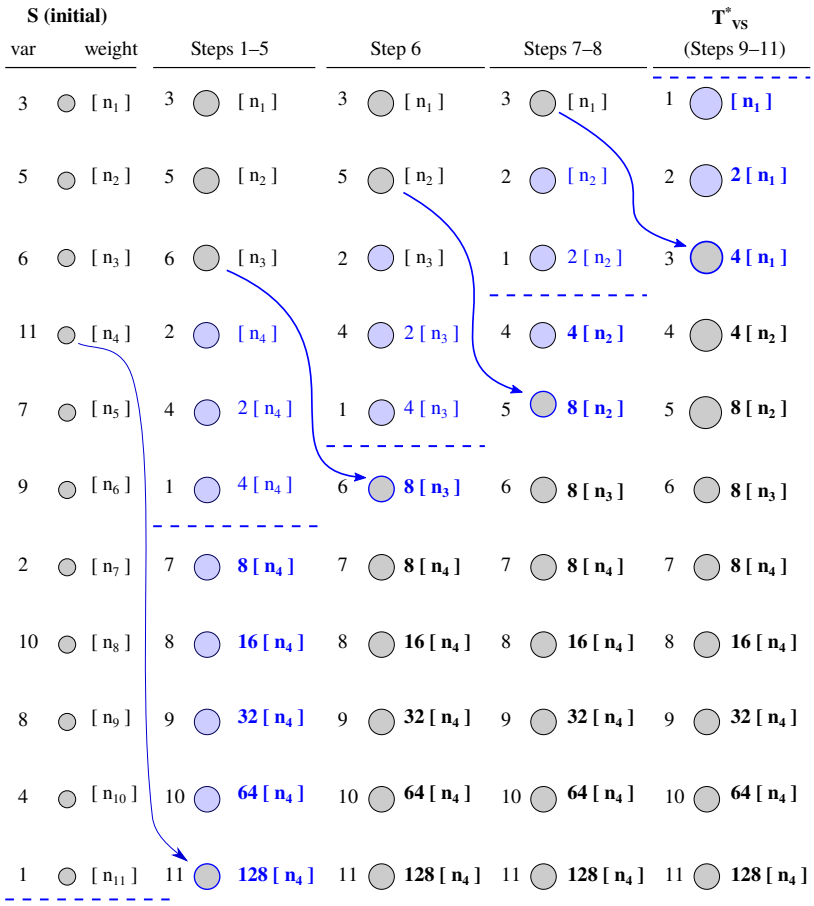
**Input:** $S = [x_1, \ldots, x_n | n_1, \ldots, n_N]$ – a weighted variable sequence
**Output:** $T^*_{\mathrm{VS}}[S, \mathbf{v}]$ for $\mathbf{v} = (1, \ldots, N)$
1: $S' \leftarrow S$, $N_f \leftarrow N$
2: **while** $N_f > 1$ **do**
3: Let $N_f$ be the largest index such that $x'_j = j$ for all $j = (N_f + 1), \ldots, N$ (and $N$ if no such index exists).
4: $S' \leftarrow \texttt{sift}(S', i_{S'}(N_f), N_f)$
5: **end while**

Note that by the above rationale, weights for elements $N_f, \ldots, N$ at any moment represent a lower bound on the corresponding weights for an optimal transformation. Since each cycle decreases $N_f$ by at least one, the while-loop of Algorithm A3.3 iterates at most $N$ times. This algorithm is illustrated with a specific example as follows.

**Example 3.4.** An optimal transformation of $S = [3, 5, 6, 11, 7, 9, 2, 19, 8, 4, 1 | n_1, \ldots, n_{11}]$ to the target order $(1, \ldots, 11)$ is presented in Figure 3.6. We start with $N_f = 11$. The first sift moves element 11 to position 11, which creates an exponentially weighted subsequence spanning positions 4–11. The next four steps move elements 10, 9, 8, and 7 to their respective positions without changing any weights (due to Corollary 3.4, since these are sifts within an exponentially weighted subsequence), resulting in $N_f = 6$. Step six sifts element 6 to the target position, creating an exponentially weighted subsequence spanning positions 3–6 and updating $N_f$ to the value of 5. The next step sifts element 5 to the target position, creating an exponentially weighted subsequence spanning positions 2–5, which allows us to perform the next sift of element 4 without changing weights. Finally, element 3 is sifted to its respective position and 2 is swapped without any weight updates. □

It can be shown that all non-redundant schedules of swaps transforming $S$ to variable

Figure 3.6: Transforming a variable sequence (sifts)




*Note.* Building an optimal transformation of a weighted variable sequence with separate sifts. Circles are elements, with labels to the left, and weights to the right of them. Horizontal dashed lines represent $N_f$.

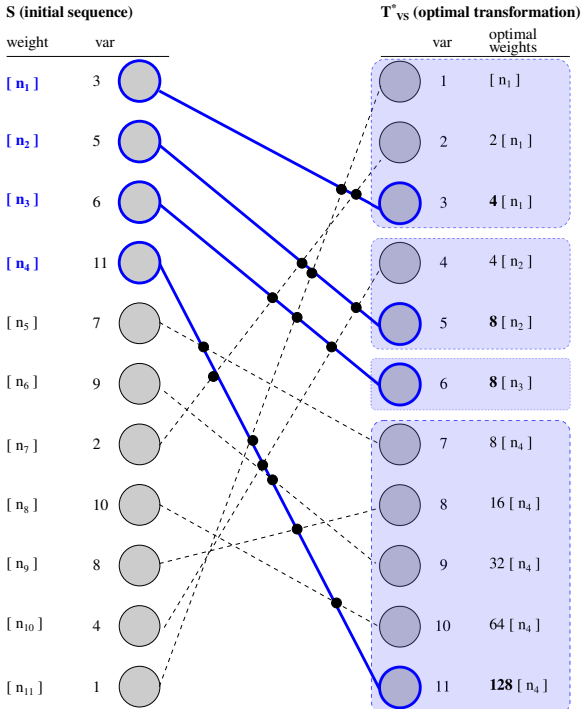
order $\mathbf{v}$ yield the same variable sequence, and hence $T^*_{\mathrm{VS}}[S, \mathbf{v}]$ is unique.

The sift operations will be called many times in the course of our further solution search, but Algorithm A3.3 can require runtime in $O(N^2)$. For example, note that a case when we would like to revert the order from $1, 2, \ldots, N$ to $N, (N-1), \ldots, 1$ would require a quadratic number of swap operations. In fact, we can build the transformation in linear time, as we present now with Algorithm A3.4.

As suggested by the example presented in Figure 3.6, the only sifts that change sequence weights are the ones dealing with elements that never participate in a swap that would decrease their index (referred to as *sinking* ones). In Figure 3.6, sinking elements are 3, 5, 6, and 11. This allows us to build a linear-time procedure as follows (illustrated in Figure 3.7). We iterate through such elements and keep a running set $P$ of elements that are swapped with the current sinking one. Scanning the initial sequence $S$ top-down (main loop of the algorithm, lines 4–16) we check if the current element is a sinking one in line 5 (which is equivalent to checking if $x_i \notin P$). If $x_i \in P$ then we exclude it from $P$ in line 14, since it will not be swapped with any of the further sinking elements. Otherwise, we build an exponentially weighted subsequence until the next sinking element with lines 6–12 as follows. We set the first element's weight in line 6 and then fill in the remaining weights one by one, doubling the previous weight. Note that by the end of this procedure (line 12) $j$ points to the start of the subsequence that will be defined by the next sinking element. Since we modify each target element's weight exactly once, the algorithm takes linear time.

Algorithm A3.4 yields $T^*_{\mathrm{VS}}[S, \mathbf{v}]$, since it replicates weights that would be obtained by Algorithm A3.3. To see this, first note that algorithms A3.4 and A3.3 generate exponentially weighted subsequences spanning the same positions (viz., between consecutive sinking elements). Then, observe that the first element in each such subsequence receives the same weight regardless of the algorithm. Consider an arbitrary such element $f$. Algorithm A3.4 assigns weight $2^{|P|} n_s$, where $n_s$ is the weight of the current sinking element in $S$, and $P$ contains all such elements $e$ that $e < f$, but $i_S(e) > i_S(s)$. But there are exactly $f - i_S(s)$ such elements (which is the number of elements that needs to be moved out of the

Figure 3.7: Transforming a variable sequence (alternative)




**Algorithm A3.4.** Align-to (weighted variable sequence), linear time

**Input:** $S = [x_1, \ldots, x_n | n_1, \ldots, n_N]$
**Output:** $T^*_{\mathrm{VS}}[S, \mathbf{v}]$ for $\mathbf{v} = (1, \ldots, N)$

```
1: function ALIGN_TO_TARGET(S)
2:     P ← ∅;  // set of elements to be passed by the current sinking element
3:     j ← 1;  // current position in the target sequence
4:     for i = 1,... N do // loop over the initial sequence
5:         if x_i ∉ P  then  // a sinking element detected
6:             n'_j ← n_i × 2^|P|;
7:             while j < x_i do
8:                 P ← P ∪ {j};
9:                 j ← j + 1;
10:                n'_j ← n'_{j−1} × 2
11:            end while
12:            j ← j + 1
13:        else
14:            P ← P \ {x_i}  // x_i is already processed in the resulting sequence
15:        end if
16:    end for
17:    return T*_VS ← [1,..., N | n'_1,..., n'_N]
18: end function
```

subsequence spanning positions $i_S(s), \ldots, s$ for it to attain its final length). The value of $2^{|P|}n_s = 2^{f-i_S(s)}n_s$ coincides with the weight that would be assigned by Algorithm A3.3 to this element after sifting the element $s$ down $f - i_S(s)$ positions. So, algorithms A3.3 and A3.4 construct the same values in each exponentially weighted subsequences, which results in the same (optimal) sequence.

*Remark* 3.5. The variable order obtained from solving the simplified problem is a solution to the original problem; however, its quality depends on the starting variable orders and the problem structure. For instance, consider two BDDs whose variable sequences are already aligned. Then, the simplified problem generated from these two diagrams would imply no changes to the variable order at optimality. (The diagrams are already aligned, and no swap decreases the size of a variable sequence, per Remark 3.4.) Therefore, the simplified problem could return a solution that is arbitrarily good or bad with respect to the original problem objective. □

## 3.4 An algorithm to solve the simplified problem

This section presents an algorithm that will be the core of our heuristic for the BDD alignment problem: a branch-and-bound procedure to solve the simplified problem, AP$(S_A, S_B; T^*_{\text{VS}})$. To design it, we first discuss a few properties of an optimal solution we seek to find in Section 3.4.1. These properties serve a foundation to the algorithm we present then in Section 3.4.2 (which also includes a detailed discussion of the lower bound we use).

### 3.4.1 Properties of an optimal solution to the simplified problem

The key property supports the intuition that if a pair of elements is already aligned in two sequences, then there is no need to change their mutual order in order to achieve an optimum.

**Lemma 3.6: Aligned pair**

Given an instance of $\text{AP}(S_A, S_B; T^*_{\text{VS}})$ and any pair of elements $a, b \in \text{var}(S_A)$ such that $a \prec_{S_A} b$ and $a \prec_{S_B} b$, there exists an optimal solution $\mathbf{v}^*$ such that $a \prec_{\mathbf{v}^*} b$.

*Proof.* The claim is illustrated in Figure 3.8a. Consider any optimal alignment $\mathbf{v}$, and suppose that $b \prec_{\mathbf{v}} a$. Consider two non-redundant schedules of swaps that transform $S_A$ and $S_B$ to $\mathbf{v}$. Note that pair $\{a, b\}$ appears in the schedule both for $S_A$ and $S_B$. If we modify both swap schedules by simply ignoring the swaps between $a$ and $b$, then the resulting sequences will be still aligned. There will be fewer swaps in each schedule, and by Remark 3.4 the total size of the resulting sequences will be no more than $|T^*_{\text{VS}}[S_A, \mathbf{v}]|$ and $|T^*_{\text{VS}}[S_B, \mathbf{v}]|$. Thus, $\mathbf{v}^*$ is an alternative optimal alignment with the desired property. □

This fact immediately allows us to restrict the search space without compromising on the objective. For the next corollary, given a sequence $S$, variable order $\mathbf{v}$, and element $w$ of $\mathbf{v}$, define $C_{S,w} = \{x : w \prec_S x\}$ as the set of elements in $S$ *covered* by $w$.

**Corollary 3.7: Covered elements**

For any instance of $\text{AP}(S_A, S_B; T^*_{\text{VS}})$, there exists an optimal variable order $\mathbf{v}^* = (v^*_1, \ldots, v^*_N)$ such that:

- $C_{S_A, v^*_N} \cap C_{S_B, v^*_N} = \varnothing$;
- $a \prec_{\mathbf{v}^*} b$ for all $a, b$ such that $a, b \in C_{S_A, v^*_N}$ and $a \prec_{S_B} b$.

*Proof.* The first claim is illustrated in Figure 3.8b. To see that $C_{S_A, v^*_N} \cap C_{S_B, v^*_N} = \varnothing$ for some optimal $\mathbf{v}^*$, note that if there exists $w \in C_{S_A, v^*_N} \cap C_{S_B, v^*_N}$, then $v^*_N \prec_{S_A} w$ and $v^*_N \prec_{S_B} w$. Lemma 3.6 states that there exists an alternative optimal alignment with $v^*_N \prec w$. (Iterating this argument establishes the claim.)

To prove the second claim, illustrated by Figure 3.8c, consider an optimal alignment $\mathbf{v}$, and suppose that $a, b \in C_{S_A, v^*_N}$, $a \prec_{S_B} b$, and $b \prec_{\mathbf{v}} a$. Let $\mathbf{v}^*$ be a modified version of $\mathbf{v}$ in which $a \prec_{\mathbf{v}^*} b$. Note that $|T^*_{\text{VS}}[S_A, \mathbf{v}]| = |T^*_{\text{VS}}[S_A, \mathbf{v}^*]|$, because reordered elements belonged

to the same exponentially weighted subsequence. Now, observe that we can modify the schedule that builds $T^*_{\text{VS}}[S_B, \mathbf{v}]$ to build $T^*_{\text{VS}}[S_B, \mathbf{v}^*]$ by omitting all swaps between elements $a$ and $b$. Due to Remark 3.4, $|T^*_{\text{VS}}[S_B, \mathbf{v}^*]| \leq |T^*_{\text{VS}}[S_B, \mathbf{v}]|$. Hence, $\mathbf{v}^*$ is also an optimal order. Applying this procedure for all such pairs of $a$ and $b$ yields an optimal alignment that possesses the desired property. □

In principle, we can also utilize the results from the previous section to restrict the first element, which will yield a way to build a lower bound.

**Lemma 3.8: Target first element**

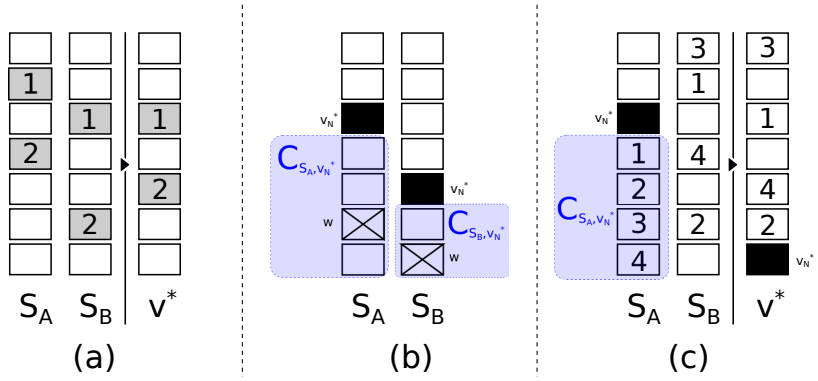
Given an instance of AP$(S_A, S_B; T^*_{\text{VS}})$, with $S_A = [a_1, \ldots, a_N | n^A_1, \ldots, n^A_N]$ and $S_B = [b_1, \ldots, b_N | n^B_1, \ldots, n^B_N]$, there exists an optimal target variable order $\mathbf{v}^*$ such that $v^*_1 \in \{a_1, b_1\}$.

*Proof.* Assume that in an optimal target variable order $\mathbf{v}$ the first element $v_1 \notin \{a_1, b_1\}$. Without loss of generality, assume $i_{\mathbf{v}}(a_1) < i_{\mathbf{v}}(b_1)$. Note that in the transformation from $S_A$ to $T^*_{\text{VS}}[S_A, \mathbf{v}]$ with Algorithm A3.3, the last sift involving $a_1$ will create an exponentially weighted subsequence spanning indices $1, \ldots, i_{\mathbf{v}}(a_1)$ in $T^*_{\text{VS}}[S_A, \mathbf{v}]$ (and weights of these elements will not be changed further). Then, pick $\mathbf{v}^*$ such that $v^*_1 = a_1$, $v^*_i = v_{i-1}$ for $i = 2, \ldots, i_{\mathbf{v}}(a_1)$, and $v^*_i = v_i$ for $i = (i_{\mathbf{v}}(a_1) + 1), \ldots, N$. Due to Corollary 3.4, $|T^*_{\text{VS}}[S_A, \mathbf{v}^*]| = |T^*_{\text{VS}}[S_A, \mathbf{v}]|$.

By the same rationale, an exponentially weighted subsequence spans indices $1, \ldots, i_{\mathbf{v}}(b_1)$ in $T^*_{\text{VS}}[S_B, \mathbf{v}]$ and since $i_{\mathbf{v}}(a_1) < i_{\mathbf{v}}(b_1)$, an exponentially weighted subsequence also spans indices $1, \ldots, i_{\mathbf{v}}(a_1)$ in $T^*_{\text{VS}}[S_B, \mathbf{v}]$. Therefore, the same corollary implies that $|T^*_{\text{VS}}[S_B, \mathbf{v}]| = |T^*_{\text{VS}}[S_B, \mathbf{v}^*]|$, and that $\mathbf{v}^*$ possesses the desired property. □

Therefore, we can restrict our search for $v^*_1$ (the first element in the target variable order) with just set $\{v^A_1, v^B_1\}$. Hence, at least one of the two sifts to position 1 must be made: either $v^A_1$ in $S_B$, or $v^B_1$ in $S_A$, which immediately gives us a lower bound (although, relatively loose).

Figure 3.8: Optimal solution properties (variable sequences)




*Note.* **(a):** "Aligned pair" (elements 1 and 2); **(b):** "Covered elements" ($w \in C_{S_A,v_N^*} \cap C_{S_B,v_N^*}$); **(c):** "Covered elements" (1, 2, 3, and 4).

Table 3.1: Example: alternative optima (simplified problem)

| Sequence $A$ | Weights | $\lvert A\rvert$ | Sequence $B$ | Weights | $\lvert B\rvert$ | $\lvert A\rvert + \lvert B\rvert$ |
|---|---|---|---|---|---|---|
| $[1,2,3,4,5,6]$ | $[1,2,3,4,2,2]$ | 14 | $[6,5,4,3,2,1]$ | $[1,2,2,3,5,1]$ | 14 | 28 |
| $[5,6,1,2,3,4]$ | $[1,2,4,8,12,16]$ | 43 | $[5,6,1,2,3,4]$ | $[1,2,2,4,8,16]$ | 33 | 76 |
| $[5,6,2,1,3,4]$ | $[1,2,4,8,12,16]$ | 43 | $[5,6,2,1,3,4]$ | $[1,2,2,4,8,16]$ | 33 | 76 |
| $[6,5,1,2,3,4]$ | $[1,2,4,8,12,16]$ | 43 | $[6,5,1,2,3,4]$ | $[1,2,2,4,8,16]$ | 33 | 76 |
| $[6,5,2,1,3,4]$ | $[1,2,4,8,12,16]$ | 43 | $[6,5,2,1,3,4]$ | $[1,2,2,4,8,16]$ | 33 | 76 |

### 3.4.2 Algorithms for solving the simplified problem

Given the algorithm discussed above, we can design a variety of neighborhood search procedures considering variable order as a point in the solution space (as we do in Section 3.5). However, local search procedures that require a decreasing objective at each step can yield local optima that are inferior to global optima.

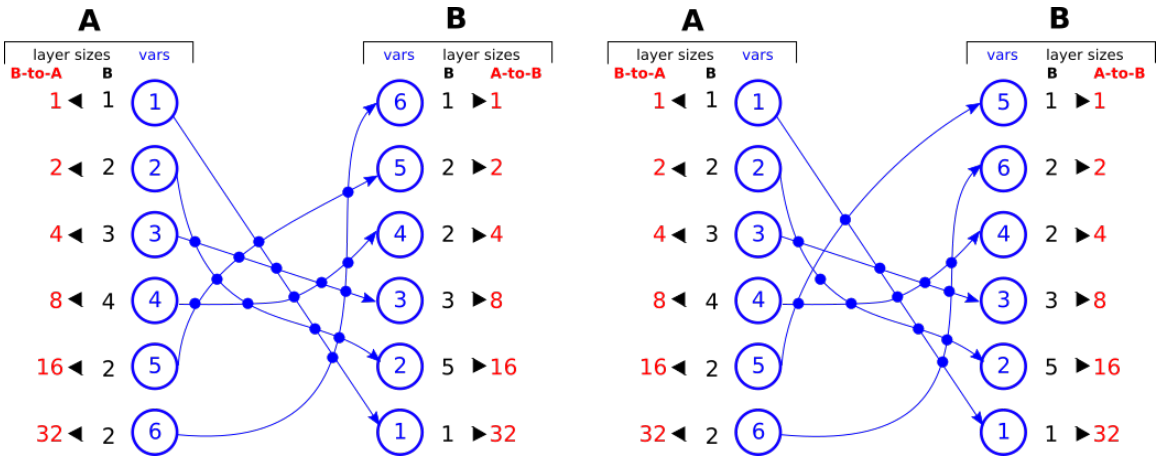
**Example 3.5.** Consider an instance comprising the following variable sequences:

- $A = [1,2,3,4,5,6|1,2,3,4,2,2]$, and
- $B = [6,5,4,3,2,1|1,2,2,3,5,1]$.

Corresponding optimal alignments are presented in Table 3.1, below the first line (which represents the initial sequences).

If we start with the alignment to $A$ or to $B$ as a target, there is no schedule of sifts that would not increase the objective (momentarily) and at the same time would eventually

Figure 3.9: An example for local optima (simplified problem)




*Note.* The figure represents aligning $A$ to var($B$) and $B$ to var($A$). Variable labels are in blue, black numbers correspond to element weights. Red numbers represent weights of the corresponding elements of *another* sequence if it will be aligned to this one. For example, weight of element ④ in sequence `B` is 2 (black number). If `A` will be aligned to `B` as a target, element ④ of this aligned `A'` will get the weight of 4 (red number).

lead to an optimal target. In fact, for each starting target alignment (var($A$) or var($B$)) there is exactly one neighbor alignment within one sift such that it would not increase the total size of two sequences if we decided to align $A$ and $B$ to it. This effect is best illustrated in the context of Algorithm A3.4. The weights resulting from aligning $A$ to var($B$) (or $B$ to var($A$)) are depicted in Figure 3.9.

Let us start with var($B$) as alignment target and see if we can modify the target with a single sift such that the total size of revised diagrams would not increase. Every such change will affect the total size in two ways:

- Increase in size of the revised version of diagram $B$ (which is obviously zero until we change the target). The amount will be equal to the cost of the specific sift in $B$.
- Increase (or decrease) in size of the revised version of diagram $A$, as compared to the revised version of $A$ as aligned to var($B$).

Sum of these two effects must be nonpositive for us to consider a change in the target in a greedy algorithm. Let us consider possible single-sift changes in the target (looking at

Table 3.2: Example: single-sift changes

| **Element** | **Change in** $\lvert A\rvert$ | **Change in** $\lvert B\rvert$ | **Total change,** $\lvert A\rvert + \lvert B\rvert$ |
|---|---|---|---|
| 2 | $(32-32) = -0$ | $(10-1) = 9$ | $+9$ |
| 3 | $(24-32) = -8$ | $(6-5) + (12-1) = +12$ | $+4$ |
| 4 | $(16-32) = -16$ | $(4-3) + (8-5) + (16-1) = +19$ | $+3$ |
| 5 | $(4-32) = -28$ | $(4-2) + (8-3) + (16-5) + (32-1) = +49$ | $+21$ |
| 6 | $(2-32) = -30$ | $(2-2) + (4-2) + (8-3) + (16-5) + (32-1) = +49$ | $+19$ |

*Note.* Each element is sifted to the last position (after element 1).

the left panel of Figure 3.9). Assume we start from aligning $A$ to var($B$). It would result in an exponentially weighted subsequence spanning the whole sequence. Note that any changes between elements $6, \ldots, 2$ in the target does not change the size of the revised $A$. Hence, the only possibility to be considered is when it does not change the revised $B$ either (otherwise the total size would increase). There is exactly one such opportunity: swapping 5 and 6 (to obtain the situation depicted in the right panel). If, for example, we wanted to align $A$ to the sequence similar to $B$, but with elements 1 and 2 flipped, in the revised diagram we would get an exponential-size subsequence ending at the second-to-last position with the size of 16, and the last element 2 would have the same weight of $2 \times 2^4 = 32$ (according to Algorithm A3.4). But the swap of elements 1 and 2 in $B$ would cost $2 \times 5 - 1 = 9$, so the objective would increase. All the rest of the target single-sift alignments (assuming the starting point of var($B$)) are summarized in Table 3.2.

The same analysis for the right panel of the figure suggests that there are no other targets in the single-sift neighborhood of var($B$) reachable without increasing the total size. It can be shown that the situation is similar for the initial point of var($A$). Therefore, the optimum, which implies 5 and 6 as first elements (in any order), followed by elements 1 and 2 (again, in any order), with a tail of $[3, 4]$, can never be reached by a simple greedy search. (There are more complicated examples with larger neighborhoods, which still do not contain an optimal target.) □

To find a global optimum for the simplified problem, we design a branch-and-bound algorithm (Algorithm A3.5) built upon Algorithm A3.3. A search tree node represents a pair of partially aligned sequences, $T^*_{\mathrm{VS}}[S_A, \mathbf{v^1}]$ and $T^*_{\mathrm{VS}}[S_B, \mathbf{v^2}]$, such that tails of $\mathbf{v^1}$ and

$\mathbf{v^2}$ coincide (i.e., $v_i^1 = v_i^2$ for all $i > N_f$ for some $N_f$).

In lines 3–7 we initialize the search tree with a (trivial) root node with $\mathbf{v}^1 = \mathrm{var}(S_A)$ and $\mathbf{v}^2 = \mathrm{var}(S_B)$. We keep a heap of search tree nodes $O$ to be examined, bounds $LB$ and $UB$, and a current incumbent candidate for an optimal solution (given by a variable order $\mathbf{w}$ and aligned sequences $\hat{S}_A = T^*_{\mathrm{VS}}[S_A, \mathbf{w}]$ and $\hat{S}_B = T^*_{\mathrm{VS}}[S_B, \mathbf{w}]$). We generate incumbent solutions via the `heuristic` function, lines 35–44, simply as the best of $\mathrm{var}(S_A)$ and $\mathrm{var}(S_B)$.

In the main loop (lines 8–32) we pick a node from the heap having the smallest lower bound and process it as follows. If the corresponding lower bound of the node is not less than the upper bound $UB$, then we prune the search and report the solution (line 11). Otherwise, in lines 13–31 we consider each element with index no more than $N_f$ as a potential candidate for position $N_f$. If the element does not violate the property described in Corollary 3.7, then we create a new node with this element moved to position $N_f$ and decrease $N_f$ by one, similar to Algorithm A3.3. Corollary 3.7 allows us to fix the relative order for some variables as we sift down, which we do in line 18. We calculate lower and upper bounds for the new node and try to update the global upper bound $UB$ (lines 23–26). We push the newly created node into the heap if the algorithm does not terminate (because the corresponding sequences are not yet aligned) in line 28. We illustrate the algorithm with the following example, involving a specific instance.

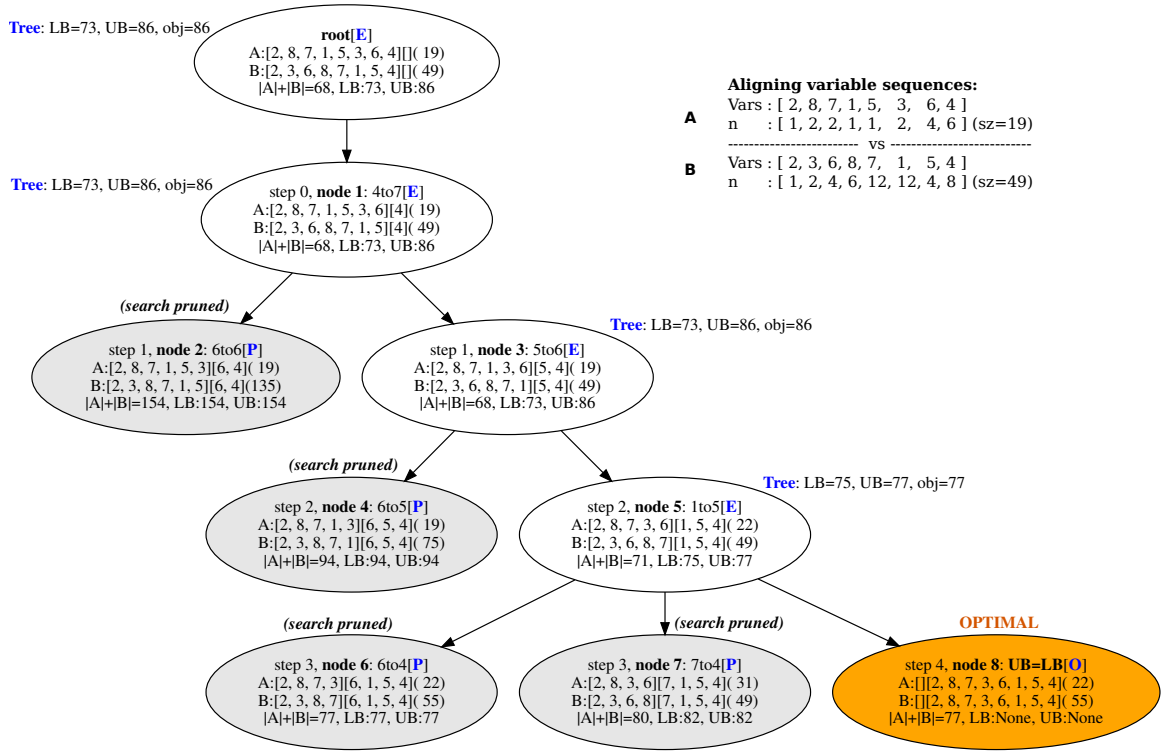
**Example 3.6.** A sample search tree for a random eight-variable instance of the problem $\mathrm{AP}(S_A, S_B; T^*_{\mathrm{VS}})$ is provided in Figure 3.10. Each node represents a pair of sequences, when all the elements with positions after $N_f$ are aligned (aligned and unaligned subsequences are separated by brackets). Nodes are differentiated by types: intermediate nodes processed by the algorithm are marked "[E]"; terminal nodes (with aligned full sequences) are marked with "[T]" and a thick border; pruned nodes are shaded and marked with "[P]"; an optimal node is marked with "[O]". Upper (lower) bound is denoted by `UB` (respectively, `LB`). Bounds within ellipses represent the ones calculated for a particular node, running bounds for the whole search tree immediately before the corresponding node processing are provided in text boxes marked with the word "Tree" outside the node. Current best known solution is

abbreviated as `obj`. Node names are derived from the search tree decision: e.g., "4to7" name means that this node was generated from the previous one by sifting an element labeled 4 to (zero-based) position 7. `step` refers to the current node processing step number (each such step creates all possible first-level descendant nodes from the current one).

Algorithm A3.5 starts from a trivial root node, representing the original sequences, with a lower bound (LB) of 73 and upper bound (UB) of 86. The current objective is 86. Each time a node is created, we calculate lower and upper bounds for that node (depicted within ellipses). The only candidate for the last position is 4, since it is already aligned and any other variable choice would have violated the "aligned pair" requirement (Lemma 3.6). Thus, the only possible node, `node 1`, is created, and a tail with one element, 4, is marked as "aligned" for this node. At the next step, there are two possible candidates for position 6 (in zero-based numbering), elements 5 and 6, so two nodes are created, `node 2` and `node 3`. At this point, we have two nodes to process: `node 2` and `node 3`, with lower bounds 154 and 73, respectively. We choose the one with the smallest lower bound, `node 3`. We create `node 4` and `node 5`, with lower bounds 94 and 75, respectively. We calculate an upper bound for `node 5` and obtain a value of 77, which is less than the current best of 86. Hence, the latter is updated to 77. Now, lower bounds of nodes to explore are 154, 94, and 75. Hence, we update the global lower bound to 75. The next node to process is `node 5` (since 75 is the smallest among 154, 94, and 75). We create `node 6` and `node 7` with lower bounds 77 and 82, respectively. An updated set of lower bounds is $\{154,\ 94,\ 77,\ 82\}$, so the global lower bound is updated with 77. But we already have a solution for 77 (generated during the upper bound calculation for `node 5`). Hence, the optimal value is 77, and we immediately create an optimal `node 8` and prune further search at `node 2`, `node 4`, `node 6`, and `node 7`.

We tested several approaches to calculate lower bounds (see Section 3.5.2 for details). The preferred one is calculated with the function `lowerBound` (lines 45–49), which is based on the following idea.

Figure 3.10: Search tree example (simplified problem)




**Lemma 3.9**

For $S = [a_1, \ldots, a_N | n_1, \ldots, n_N]$ and arbitrary $\mathbf{v}$, define $I(S, \mathbf{v}) = \{(j, l) : \ a_j \prec_S a_l$ and $a_l \prec_{\mathbf{v}} a_j\}$. Then:

$$|T^*_{\mathrm{VS}}[S, \mathbf{v}]| \geq |S| + \sum_{(j,l) \in I(S,\mathbf{v})} (2n_j - n_{j+1}).$$

*Proof.* Let us consider the optimal schedule of swaps as implemented in Algorithm A3.3. We prove the lemma by comparing the maximum possible increase of the lower bound due to swaps at each step of the algorithm, and by showing that this amount does not exceed the increase of the actual sequence size.

At the first step, we move an element from position $j$ to $N$ (such as the one depicted with the first arrow in Figure 3.6). The actual increase in the sequence size is $\sum_{k=1}^{N-j}(2^k n_j - n_{j+k})$, which is the total difference between the new and old element weights on positions

$j+1,\ldots,N$. Because this sift creates an exponentially weighted subsequence spanning positions $j,\ldots,N$, all subsequent sifts of elements $a_{j+1},\ldots,a_N$ do not change any weights (by Lemma 3.3). The lower bound due to swaps in this first step would be maximal if $(k,l)\in I(S,\mathbf{v})$ for all $j\le k<l\le N$. Then, the contribution to the lower bound would be

$$\sum_{k=j}^{N-1}(N-k)(2n_k-n_{k+1}).$$

Denoting $K=N-j$, the difference between the actual size increase and the change in the lower bound at the first step of Algorithm A3.3 is at most:

$$\Delta=\sum_{k=1}^{K}(2^k n_j-n_{j+k})-\sum_{k=0}^{K-1}(K-k)(2n_{j+k}-n_{j+k+1}). \tag{3.1}$$

We seek to prove that $\Delta\ge 0$. Note that for $K=1$ we have $\Delta=0$. For $K\ge 2$:

$$\begin{aligned}\sum_{k=0}^{K-1}(K-k)(2n_{j+k}-n_{j+k+1}) &= \sum_{k=0}^{K-1}2(K-k)n_{j+k}-\sum_{i=1}^{K}(K-(i-1))n_{j+i}\\ &= 2Kn_j+\sum_{k=1}^{K-1}(K-k-1)n_{j+k}-n_{j+K}.\end{aligned}$$

Substituting $\sum_{k=1}^{K}2^K n_j=(2^{K+1}-2)n_j$ we can revise (3.1) as:

$$\begin{aligned}\Delta &= (2^{K+1}-2)n_j-\sum_{k=1}^{K}n_{j+k}-(2Kn_j+\sum_{k=1}^{K-1}(K-k-1)n_{j+k}-n_{j+K})\\ &= (2^{K+1}-2K-2)n_j-\sum_{k=1}^{K-1}(K-k)n_{j+k}.\end{aligned}$$

By noting that $n_{j+k}\le 2n_{j+k-1}$ for all $k$, we obtain:

$$\Delta\ge\Big((2^{K+1}-2K-2)-\sum_{k=1}^{K-1}(K-k)2^k\Big)n_j.$$

Noting that $\sum_{k=1}^{K-1}(K-k)2^k=2^{K+1}-2K-2$ when $K\ge 2$, we get that $\Delta\ge 0$. Therefore,

the size increase of the sequence is at least as large as the lower bound contribution at the first step of Algorithm A3.3.

Now, consider the next step of Algorithm A3.3, from position $l$ to some position $L$, $l < L \leq N-1$. The sequence size increase overestimate stemming from swaps between any elements at positions $j, \ldots, L$ is compensated by the underestimates due to the previous weight-changing sift. Overestimates due to the swaps involving elements $l, \ldots, j$ are compensated by the underestimates due to the current weight-changing sift (by the rationale above). Repeating this logic for the consecutive weight-changing sifts establishes the lemma. □

**Corollary 3.10**

For AP$(S_A, S_B; T^*_{\text{VS}})$, the optimal objective

$$s^* \geq |S_A| + |S_B| + \sum_{(l,j) \in I(S_A, \text{var}(S_B))} \min\{2n_l^{S_A} - n_{l+1}^{S_A},\ 2n_{i_{S_B}(a_j)}^{S_B} - n_{i_{S_B}(a_j)+1}^{S_B}\}.$$

*Proof.* Consider an optimal alignment $\mathbf{v}$. For each pair of indices $(l, j)$ such that $a_l \prec_{S_A} a_j$ and $a_j \prec_{S_B} a_l$, define $\delta_{lj} = 1$ if $a_l \prec_{\mathbf{v}} a_k$ and $\delta_{lj} = 0$ otherwise. Applying Lemma 3.9 twice:

$$|T^*[S_A, \mathbf{v}]| \geq |S_A| + \sum_{(l,k) \in I(S_A, \text{var}(S_B))} (1 - \delta_{lk})(2n_l^{S_A} - n_{l+1}^{S_A}),$$
$$|T^*[S_B, \mathbf{v}]| \geq |S_B| + \sum_{(l,j) \in I(S_A, \text{var}(S_B))} \delta_{lj}(2n_{i_{S_B}(a_j)}^{S_B} - n_{i_{S_B}(a_j)+1}^{S_B}).$$

Taking the sum of the two inequalities, we conclude that

$$\begin{aligned} s^* &= |T^*_{\text{VS}}[S_A, \mathbf{v}]| + |T^*_{\text{VS}}[S_B, \mathbf{v}]| \geq |S_A| + |S_B| \\ &\quad + \sum_{(l,j) \in I(S_A, \text{var}(S_B))} [(1 - \delta_{lj})(2n_l^{S_A} - n_{l+1}^{S_A}) + \delta_{lj}(2n_{i_{S_B}(a_j)}^{S_B} - n_{i_{S_B}(a_j)+1}^{S_B})] \\ &\geq |S_A| + |S_B| + \sum_{(l,j) \in I(S_A, \text{var}(S_B))} \min\{2n_l^{S_A} - n_{l+1}^{S_A}, 2n_{i_{S_B}(a_j)}^{S_B} - n_{i_{S_B}(a_j)+1}^{S_B}\}, \end{aligned}$$

regardless of specific $\delta_{lj}$ values. □

## 3.5 Numerical experiments

We test the proposed approach in a series of numerical experiments carried out using the Clemson University Palmetto high-performance computing cluster infrastructure. The algorithms were implemented in Python and incorporated into a computational pipeline created with `GNU make`.

The core dataset comprises $10,048$ random fifteen-variable BDD pair alignment instances, which is generated as discussed in Section 3.5.1.

In a preliminary experiment presented in Section 3.5.2 we define a metric and numerically compare the performance of different approaches to calculating lower bounds for the simplified problem. This motivates our choice of a specific lower bound that we use in the further experiments.

Then, for each instance, we construct a simplified problem instance as per Definition 3.4, obtain a solution, and revise BDDs to the target variable order derived from the simplified problem. We discuss the experiments related to simplified and original problems in Sections 3.5.3 and 3.5.4, respectively. Finally, we consider structural properties of the solutions to the original problem that were obtained with the proposed heuristic in Section 3.5.5.

### 3.5.1 A dataset of random instances

To generate each instance, we define the desired number of layers and diagram growth parameter $p \in (0, 1]$ (shared by all instances), start from the root node, and process each of the outgoing arcs as follows. If there are no nodes in the next layer, we create the tail node for the arc. Otherwise, we create a new tail node with probability $p$, and use a (uniformly) random existing node in the next layer as a tail with probability $(1 - p)$.

This procedure is repeated for each node in each consecutive layer, until we reach the layer corresponding to the last variable. Then the terminal nodes, $\mathbf{T}$ and $\mathbf{F}$, are created, and all nodes in the previous layer are connected randomly to $\mathbf{T}$ or $\mathbf{F}$. After the diagram

**Algorithm A3.5.** BB-search for AP$(S_A, S_B; T^*_{\mathrm{VS}})$

**Input:** weighted variable sequences $S_A$ and $S_B$
**Output:** $\mathbf{v}$ – an optimal variable order, $T^*_{\mathrm{VS}}[S_A, \mathbf{v}]$, and $T^*_{\mathrm{VS}}[S_B, \mathbf{v}]$

1: **function** SEARCH($S_A, S_B$)
2: **Initialization:**
3: Initialize $O$ to be an empty heap of open search tree nodes (lower bounds as keys)
4: `rootNode` ← **create** node($S_1 \leftarrow S_A, S_2 \leftarrow S_B, N_f \leftarrow |S_A|$)
5: $O$.`push`(`lowerBound`(`rootNode`), `rootNode`)
6: $\hat{S}_A, \hat{S}_B \leftarrow$ `heuristic`(rootNode) *// current solution candidate*
7: $UB \leftarrow |\hat{S}_A| + |\hat{S}_B|$ *// global upper bound*
8: **while** $|O| > 0$ **do**
9: $LB$, `nextNode` = O.`pop`() *// Pick a node with the smallest objective lower bound*
10: *// process* ***`nextNode`*** *as follows:*
11: **if** $LB \geq UB$ **then goto** line 33
12: $S_1, S_2, N_f \leftarrow$ `nextNode`
13: **for** $j = N_f, \ldots, 1$ **do**
14: $a \leftarrow x_j^{S_1}$
15: **if** $C_{S_1,a} \cap C_{S_2,a} \cap \{x_1^{S_1}, \ldots, x_{N_f}^{S_1}\} = \varnothing$ **then**
16: $S'_1 \leftarrow$ `sift`$(S_1, i_{S_1}(a), N_f)$
17: $S'_2 \leftarrow$ `sift`$(S_2, i_{S_2}(a), N_f)$
18: reorder elements of $S'_2$ and $S'_1$ with labels in $C_{S_1,a}$ and $C_{S_2,a}$ (respectively) as per Corollary 3.7
19: `newNode` ← **create** node($S_1 \leftarrow S'_1, S_2 \leftarrow S'_2, N_f \leftarrow (N_f - 1)$)
20: $Q_A, Q_B \leftarrow$ `heuristic`(`newNode`)
21: `node_UB` ← $|Q_A| + |Q_B|$
22: `node_LB` ← `lowerBound`(`newNode`)
23: **if** $UB >$ `node_UB` **then**
24: $\hat{S}_A \leftarrow Q_A$, $\hat{S}_B \leftarrow Q_B$
25: $UB \leftarrow$ `node_UB`
26: **end if**
27: **if** `node_LB` $< UB$ **and** $\mathrm{var}(S'_1) \neq \mathrm{var}(S'_2)$ **then**
28: $O$.`push`(`node_LB`, `newNode`) *// add the new node to the heap*
29: **end if**
30: **end if**
31: **end for**
32: **end while**
33: **return** $(\mathrm{var}(\hat{S}_A), \hat{S}_A, \hat{S}_B)$;
34: **end function**

35: **function** HEURISTIC(node)
36: $S_1, S_2, N_f \leftarrow$ node
37: $S'_1 \leftarrow$ `align_to_target`$(\mathrm{var}(S_2))$
38: $S'_2 \leftarrow$ `align_to_target`$(\mathrm{var}(S_1))$
39: **if** $|S_1| + |S'_2| < |S'_1| + |S_2|$ **then**
40: **return** $(S_1, S'_2)$
41: **else**
42: **return** $(S'_1, S_2)$
43: **end if**
44: **end function**

45: **function** LOWERBOUND(node)
46: $A, B, N_f \leftarrow$ node
47: Compute $LB$ based on Corollary 3.10.
48: **return** $LB$
49: **end function**

is generated, we process the layers from the last to the first one to make the BDD quasi-reduced: we perform two swaps for each layer, starting from the last one (two swaps for the same index do not change the order of variables, but due to the implementation of swap removes all redundant nodes). Finally, unless stated otherwise, the layer labels are assigned randomly. We also ensure that instances are unique (re-creating the diagram from scratch if it is not the case).

The parameter effectively determines the diagram size: Setting $p = 0$ would result in a single deterministic diagram with one node per layer, and setting $p = 1$ yields a deterministic diagram of exponentially growing layer sizes. For intermediate values of $p$, we would have a random diagram with layer sizes growing depending on $p$. Figure 3.11 illustrates the resulting distribution of layer sizes for different values of $p$. Each panel in the figure corresponds to a BDD layer, each column within a panel corresponds to a specific value of $p$: 0.2, 0.5, or 0.8. Note how the lower values produce relatively compact BDDs, while for $p = 0.8$ layer widths grow exponentially up to a point when the enforced quasi-reduced property becomes binding (e.g., enforcing the last layer to have effectively always four nodes, because there are $2^2 = 4$ distinct ways to point two types of outgoing arcs to two terminal nodes).

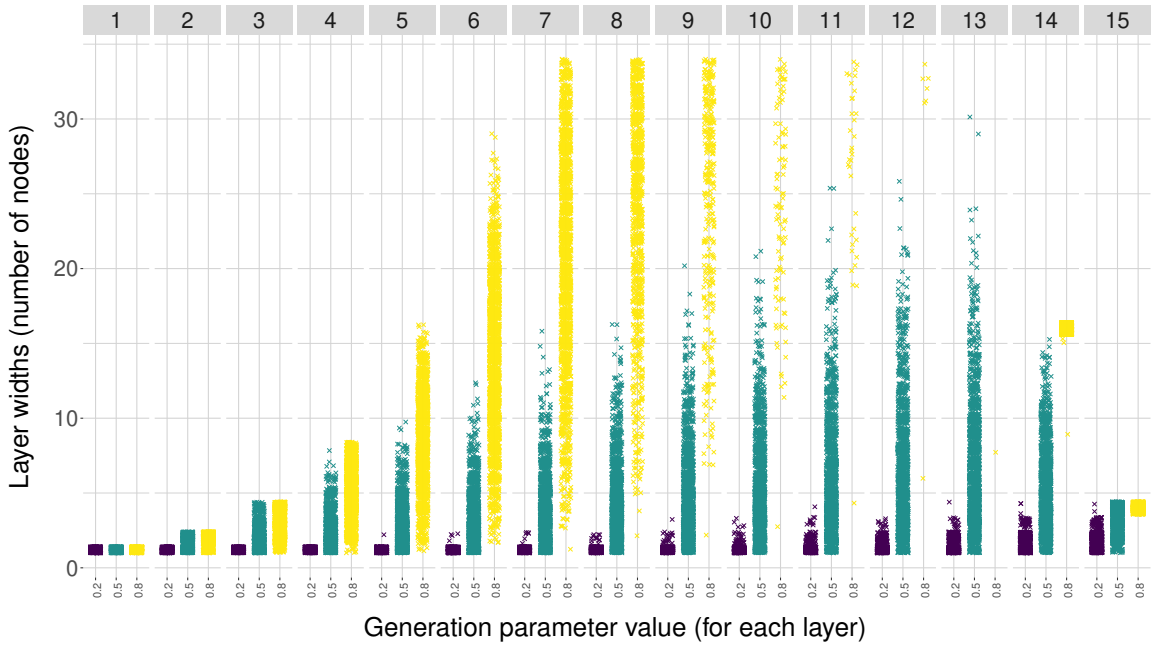
Unless stated otherwise, we considered instances of two BDDs generated with $p = 0.6$ comprising 15 variables, labeled $1, \ldots, 15$ for the first diagram, and a random permutation of the labels for the second one.

### 3.5.2 Analyzing lower bounds for the simplified problem

We examined several choices for computing lower bounds on the simplified problem objective. The competing methods are described below, where $A$ and $B$ are the current states of sequences $S_A$ and $S_B$ in the course of alignment.

- **Current size:** a naive bound of current size before the alignment, $\underline{s^0} = |A| + |B|$.
- **Minimum size / first element aligned:** a bound based on Lemma 3.8. Given that

Figure 3.11: Align-BDD dataset: layer widths summary




*Note.* Layer widths for a set of 1,000 random quasi-reduced 15-variable binary decision diagrams for each value of $p$.

$v_1 \in \{a_1, b_1\}$ in an optimal target alignment, we derive the bound $\underline{s^1} = \min\{|A| + |\texttt{sift}(B, i_B(a_1), 1)|, |\texttt{sift}(A, i_A(b_1), 1)| + |B|\}$.

- **Minimum size / last element aligned:** a bound based on aligning the last element. We simply iterate through all possible elements as candidates for the $N$-th position and pick the smallest size after the alignment of the last element only. That is, $\underline{s^N} = \min_i\{|\texttt{sift}(A, i, N)| + |\texttt{sift}(B, i_B(a_i), N)|\}$.

- **Inversions-driven bound:** a bound based on Corollary 3.10, computed as $\underline{s^I} = \sum_{(i,j)\in I(A,\mathrm{var}(B))} \min(2n_i^A - n_{i+1}^A, 2n_{i_B(a_j)}^B - n_{i_B(a_j)+1}^B)$.

We chose our lower bound calculation approach based on a separate numerical experiment that compared the options above. In particular, we calculated the current size before the alignment, $\underline{s^0}$, and the true optimal objective, $s^*$. For each of the lower bounds $\underline{s'}$, we calculate the following characteristic:

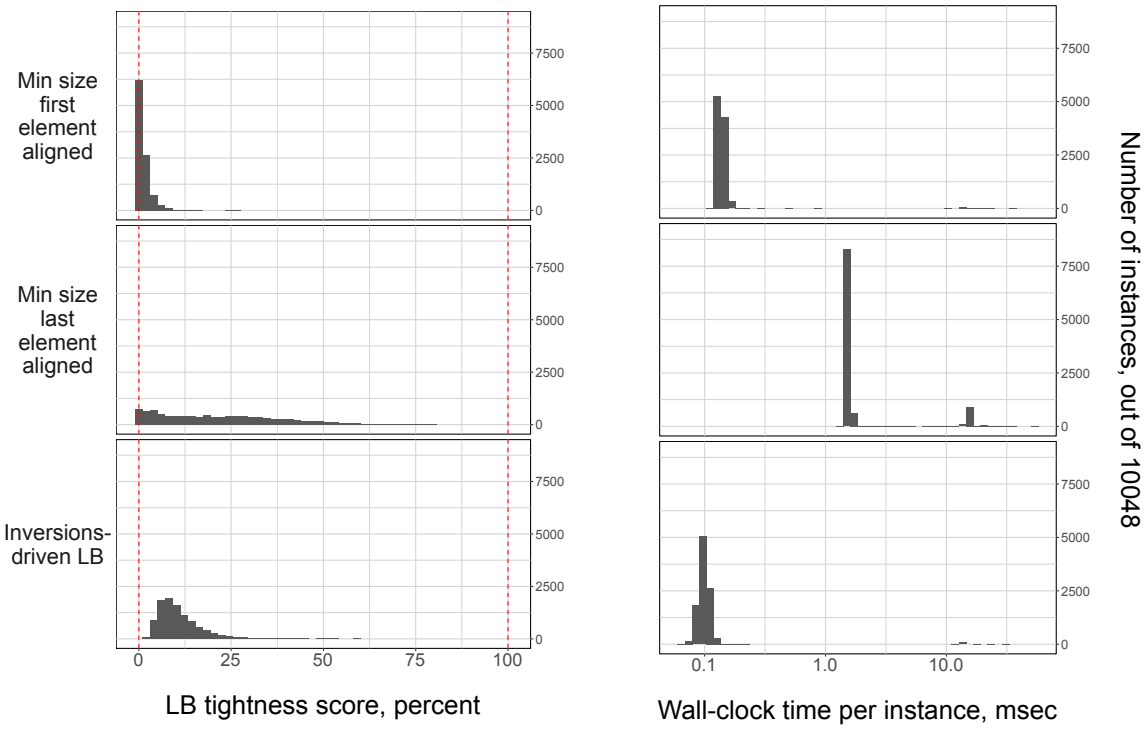
$$\text{LB score}(\underline{s'}) = \frac{\underline{s'} - \underline{s^0}}{s^* - \underline{s^0}}.$$

A score of 100% would correspond to the exact optimal objective (a perfect lower bound), while 0% would indicate a naive bound performing no better than the "current size" estimate. We present histograms of scores for lower bounds corresponding to our main dataset of random align-BDD instances in Figure 3.12. The first-element-aligned heuristic performs poorly, barely exceeding the current size bound. The last-element-related bound performs better that the inversions-driven one, but due to significantly lower runtimes we use the latter.

### 3.5.3 Solving the simplified problem

We solved the simplified problem with several heuristics and compared them to the exact branch-and-bound method described in Section 3.4.2.

Figure 3.12: Lower bounds benchmarking (simplified problem)




- **Best of A and B:** calculate the objectives for $\text{var}(S_A)$ and $\text{var}(S_B)$ as the target orders, and pick the best one.

- **Greedy swaps:** start with the target order obtained by the "Best of A and B" heuristic. At each step, calculate the objective resulting from swapping each pair of adjacent labels in the target order. Implement a swap that maximally decreases the objective, and reiterate. Stop when no swaps improve the objective.

- **Greedy sifts** (inspired by Rudell 1993): adjust the previous procedure introducing sifts instead of swaps. At each iteration (which we call a "pass"), consider moving a variable to all possible positions (without affecting the relative positions of other variables). We implemented variants of this heuristic that terminate after one pass, two passes, or when no further improvements are possible (which we call "all passes").

- **Best of five random orders:** pick a random order and calculate the objective value from aligning the sequences to this target. Repeat five times and choose the best one.

Table 3.3: Heuristic performance (simplified problem).

| **Heuristic** | **Optimal solution** | **Within** 10% |
|---|---|---|
| Best/five random orders | 0 | 1 |
| Best of A and B | 0 | 7 |
| Greedy sifts (one pass) | 4 | 191 |
| Greedy sifts (two passes) | 44 | 1,075 |
| Greedy swaps | 891 | 2,933 |
| Greedy sifts (all) | 8,960 | 10,045 |
| Branch-and-bound | 10,048 | 10,048 |

*Note.* Number of times (out of 10,048) each heuristic achieved the objective performances listed by column.

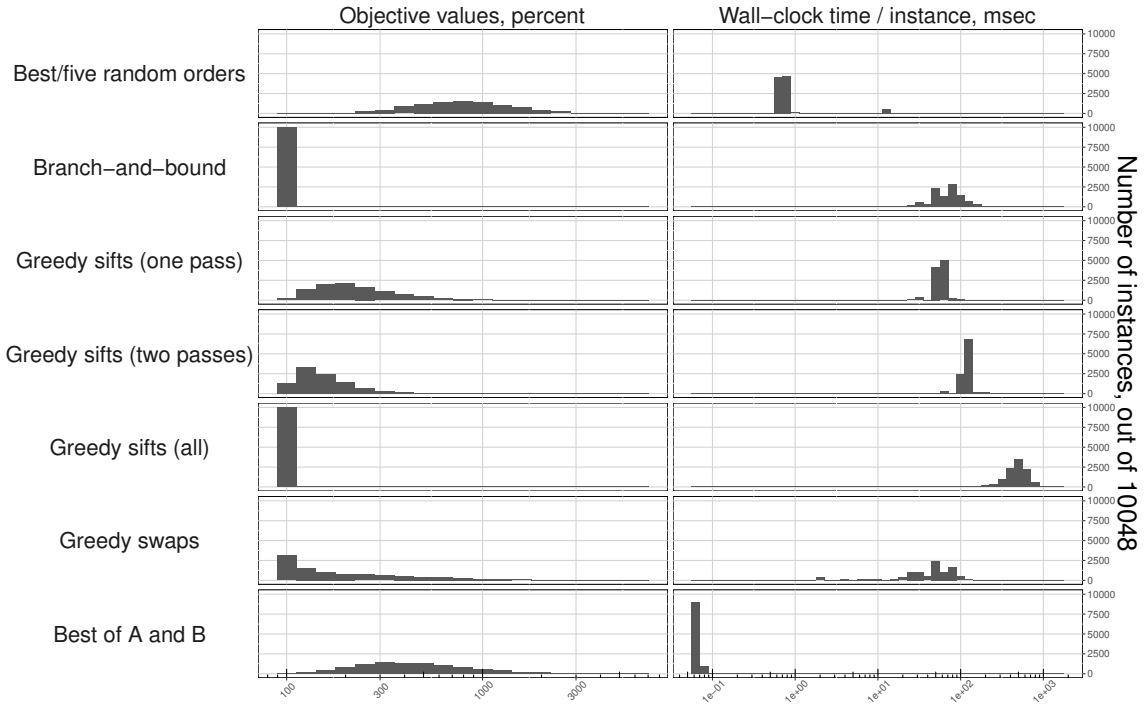
The performance of these strategies in terms of objective values are summarized in Table 3.3. We calculate the number of instances when the objective provided by each heuristic was optimal or exceeded the optimal value by at most 10%. The two naive heuristics ("Best of A and B" and "Best of five random orders") performed poorly. The two-passes version of the greedy sifts heuristic yielded an objective value within ten percent of optimal in about 10% of the instances, while the all-passes version gave near-optimal solutions on a consistent basis. However, the runtimes of these approaches are significantly different. Figure 3.13 depicts the runtime versus objective quality trade-off. Although the all-passes greedy sifts heuristic provided near-optimal solutions, its runtime was excessive, sometimes exceeding that of the exact method. Greedy swaps and restricted greedy sifts heuristics provided a reasonable compromise between objective quality and runtime.

### 3.5.4 Solving the original problem

For each align-BDD instance, we generate and solve a simplified problem instance (to optimality, with the branch-and-bound method). Then the original BDDs are revised to the variable order obtained from the simplified problem to calculate the objective to $\mathrm{AP}(A, B; T^*)$. We benchmark the proposed approach against the following heuristics.

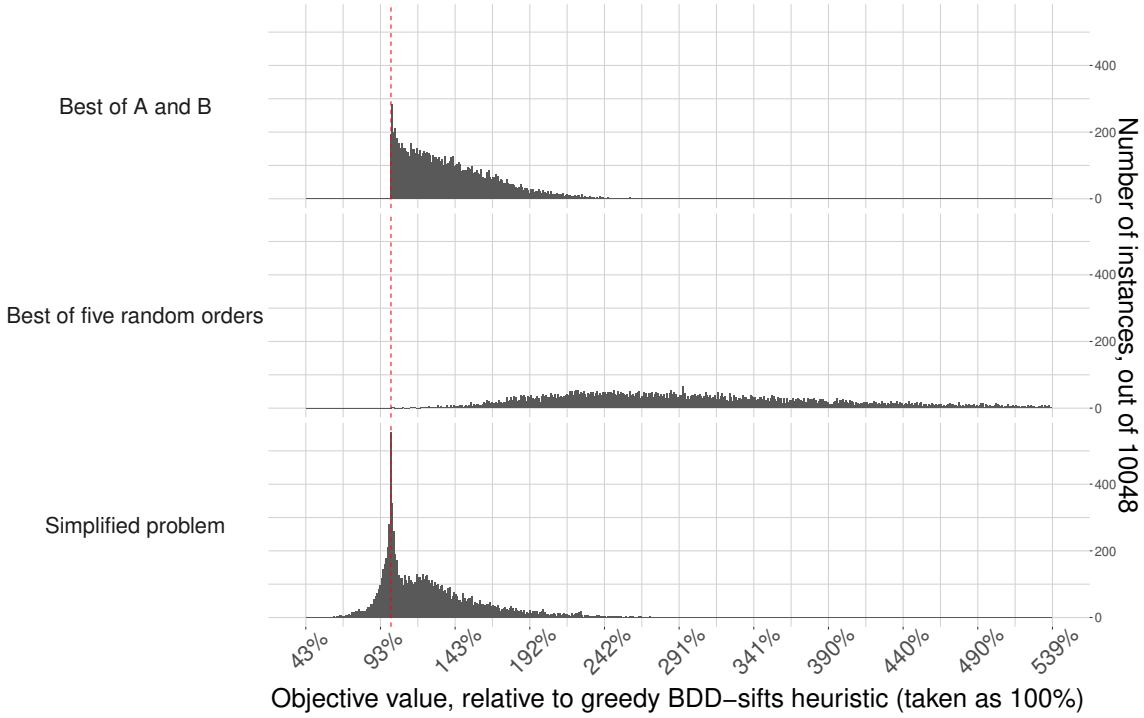
- **Best of five random orders:** generate a random order, align both diagrams. Repeat five times and pick the best objective.
- **Best of A and B:** align both BDDs to $A$ and to $B$, and pick the best objective.
- **Greedy BDD sifts** (inspired by Rudell 1993): start with $\mathrm{var}(A)$ as a target variable

Figure 3.13: Runtime and objectives (simplified problem)




*Note.* Computational time and relative objective values for various heuristics, simplified problem.

Figure 3.14: Aligned BDD sizes summary




*Note.* Histogram of the aligned BDD sizes for different heuristics, original problem.

order, and try to improve the objective as follows. For each variable, try sifting it to each possible position and pick one with the best objective. Repeat with the next variable, and stop when all variables are processed. Re-run the procedure using var($B$) as the initial target variable order, and choose the best solution out of the two.

To make the results comparable across multiple instances with different optimal objectives, we divide the objective values by the baseline objective value obtained by the greedy BDD sifts heuristic (for each particular instance). An objective of 100% would imply performance similar to the baseline heuristic, and 50% would suggest that the heuristic found a solution with the objective being half of the one obtained by the baseline. The resulting objective value distributions are presented in Figure 3.14.

Our simplified-problem based heuristic outperformed the greedy BDD sifts (baseline) on about 20% of all instances, while being at most 30% worse than the baseline at

about 60% of instances. However, a problem involving $O(N)$-sized variable sequences scales differently than the original problem (see Figure 3.15). We generated 200 instances for each problem size, from 5 to 25 variables. We solved each instance with the simplified-problem based heuristic and the baseline, greedy BDD sifts heuristic. Runtimes in logarithmic scale are presented in Figure 3.15a. Each solved instance is denoted by a point (for BDD sifts heuristic) or a cross (for the simplified problem-based heuristic). Logarithms of median values are depicted with line plots. Labels in the boxes indicate the share of instances when the proposed heuristic outperformed the baseline both in terms of time and objective. Figure 3.15b presents histograms of objective values for the proposed heuristic, relative to the BDD sifts, for different problem sizes. (Instances within the top 1% of their relative objective values are omitted for figure readability.) The number of variables is indicated in gray boxes to the left of each panel.

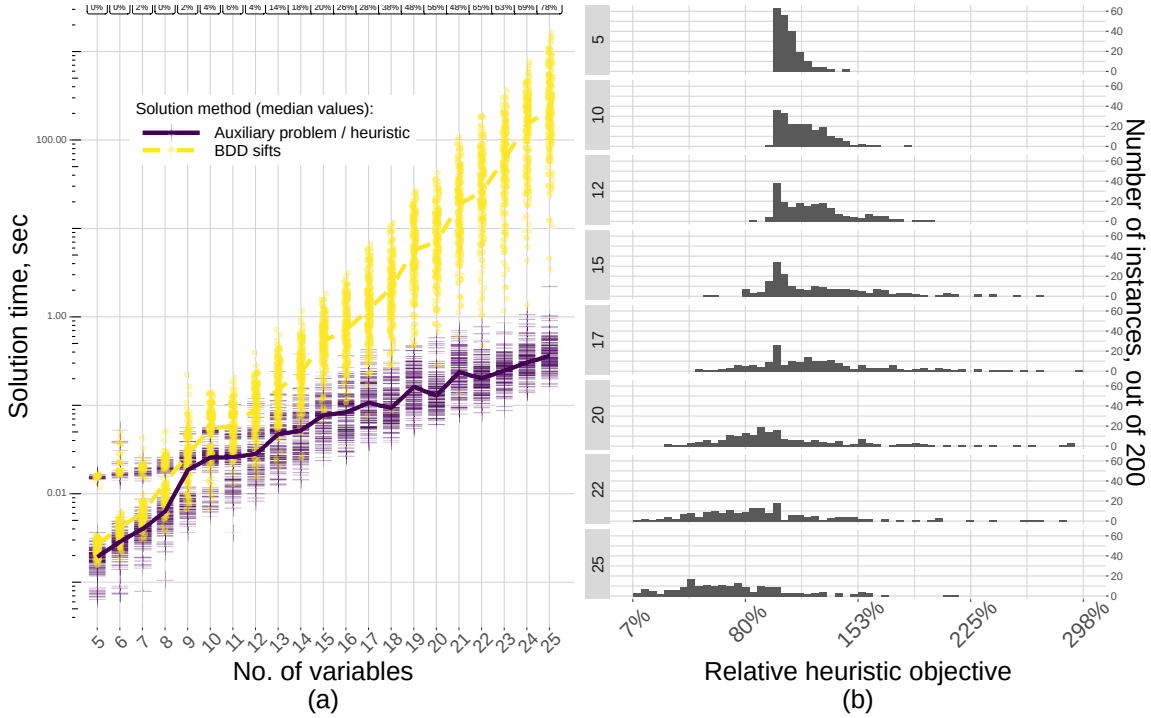
Note that the heuristic we propose often escapes local optima. Thus, it yields better objectives than the baseline, and this effect is stronger as the problem size grows. For five-variable instances the simplified is worse than the greedy BDD sifts heuristic both in terms of time and the objective value. However, as the instance grows, the greedy BDD sifts heuristic becomes far slower and possibly impractical to use. We therefore recommend the proposed simplified-problem based heuristic on instances containing more than twenty variables. In our numerical experiment depicted in Figure 3.15, our heuristic outperformed the greedy sifts heuristic on instances containing more than twenty variables both in terms of time and objective on more than half of the instances.

### 3.5.5 Simplified problem solution structure

Note that our proposed heuristic solves the simplified problem to obtain a variable order for the original problem. The foregoing analysis examines the objective quality of these solutions to the original problem, compared to the optimal objective function value of the original problem.

It is also interesting to consider whether optimal solutions to the simplified and the

Figure 3.15: Runtime scaling (original problem)




*Note.* Changes in runtimes and objective values for the proposed heuristic versus the baseline of greedy BDD sifts for varying problem sizes.

original problems share structural properties. Despite the fact that our heuristic can yield arbitrarily poor solutions as per Remark 3.5, we executed a numerical experiment showing that our solutions are usually more than 75% similar to an optimal solution for the original problem. We describe this experiment and our measure of solution similarity in this section.

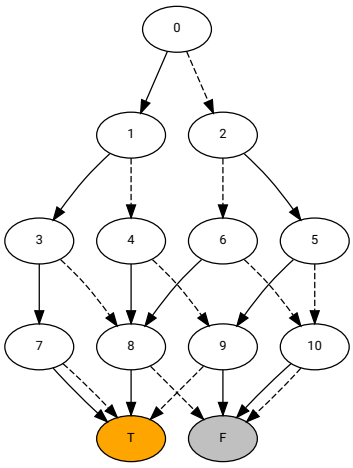
First, note that the weighted variable sequence model can yield bounds on the BDD size that are arbitrarily tight. For instance, consider two arbitrary BDDs with a shared set of variables (that allow for some variation in sizes depending on the order of variables), and align them both to a *worst* possible shared order of variables. Then, the simplified problem generated from these two diagrams after the alignment would imply no changes in an optimal solution. (The diagrams are already aligned, and no swap decreases the size of a variable sequence, per Remark 3.4.) So, even if we solve the simplified problem $\mathrm{AP}(S_A, S_B; T^*_{\mathrm{VS}})$ to optimality, no (strictly) worse solution to $\mathrm{AP}(A, B; T^*)$ is possible for such instance, by construction.

At the same time, the heuristic can be arbitrarily tight in some cases, depending on the problem structure. As pointed out in Remark 3.5, we might just start with an optimal alignment. Less trivial cases are possible as well. Consider, for instance, the diagram in Figure 3.16. Assume we are seeking to align two (quasi-reduced) diagrams having this exact structure, but different layer labels: $\mathrm{var}(A) = (1, 2, 3, 4)$ and $\mathrm{var}(B) = (3, 2, 1, 4)$. In fact, any change in the order of the first three layers does not change the diagram size, which is 13 nodes, and layers indexed 1–3 already induce an exponentially weighted subsequence in the corresponding weighted variable sequences. Hence, the weighted variable sequence will yield a tight bound of 13 for this instance.

Although the heuristic might perform well for the instances we discuss, this performance depends on the structural properties of the problem. We provide several insights in this regard.

Here we seek to analyze how similar solutions that optimize the simplified problem are to solutions that optimize the original problem. To conduct this analysis, we introduce a measure of "distance" of the solution obtained from the simplified problem to an optimal

Figure 3.16: Example BDD: on variable sequence tightness




solution to the original problem. Define a *similarity score* between two length-$N$ sequences, $A$ and $B$, as follows:

$$\text{simscore}(A, B) = 1 - |\{(a, b) : a \prec_A b,\ b \prec_B a\}| / \binom{N}{2}.$$

This score equals 100% if $A$ and $B$ are aligned, 0% if one is the reverse of the other, and 50% if half of all pairs of elements in $A$ are inverted in $B$. (This is a property of $\text{var}(A)$ and $\text{var}(B)$, so it does not depend on the element weights.)

To discuss the deviation of the solution obtained by the proposed heuristic and a true optimum to the original problem, we generated 1,000 random seven-variable align-BDD instances, enumerated all optima of the original problem by a brute-force algorithm, and calculated the minimal and maximal similarity scores between the obtained solution and any true optimum. The results are presented in Figures 3.17–3.20. Note that there are usually several optima for our instances (e.g., out of 1,000 instances included in Figure 3.17, 979 had more than one optimum and 607 had more than five). These optima are usually quite different: 60-70% of the instances have the smallest similarity score between optima of no more than 25% (see Figure 3.18).

We see from Figure 3.19 that in more than half of the cases, the obtained solution

Figure 3.17: Histogram for the number of optima

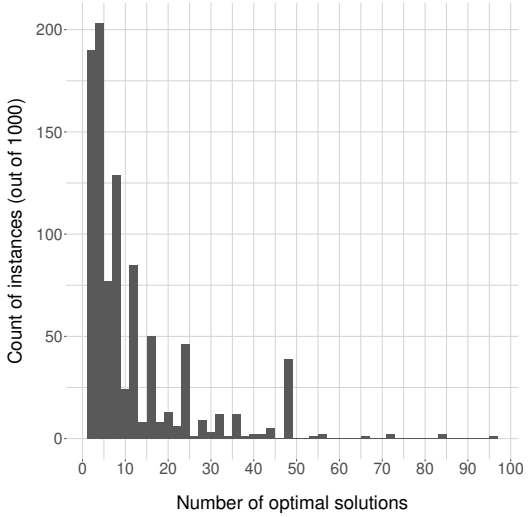
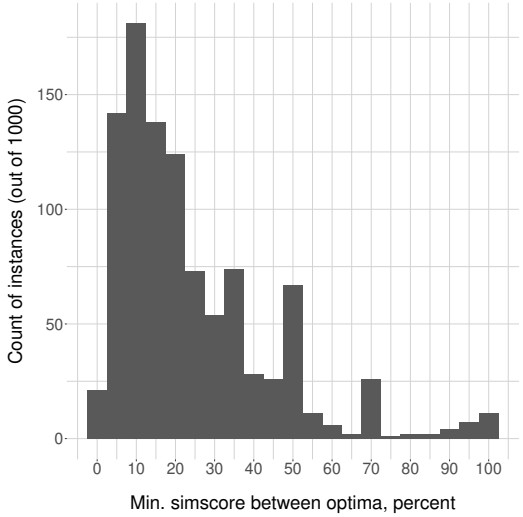
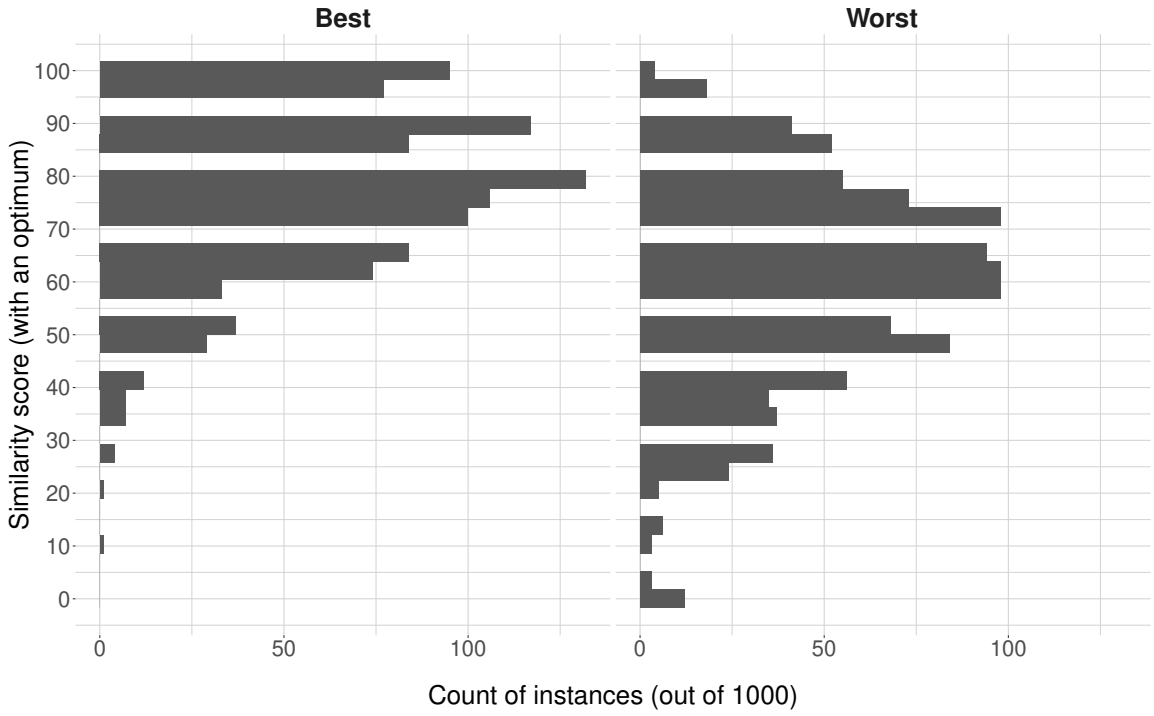



Figure 3.18: Histogram for the optimal set "diameter"



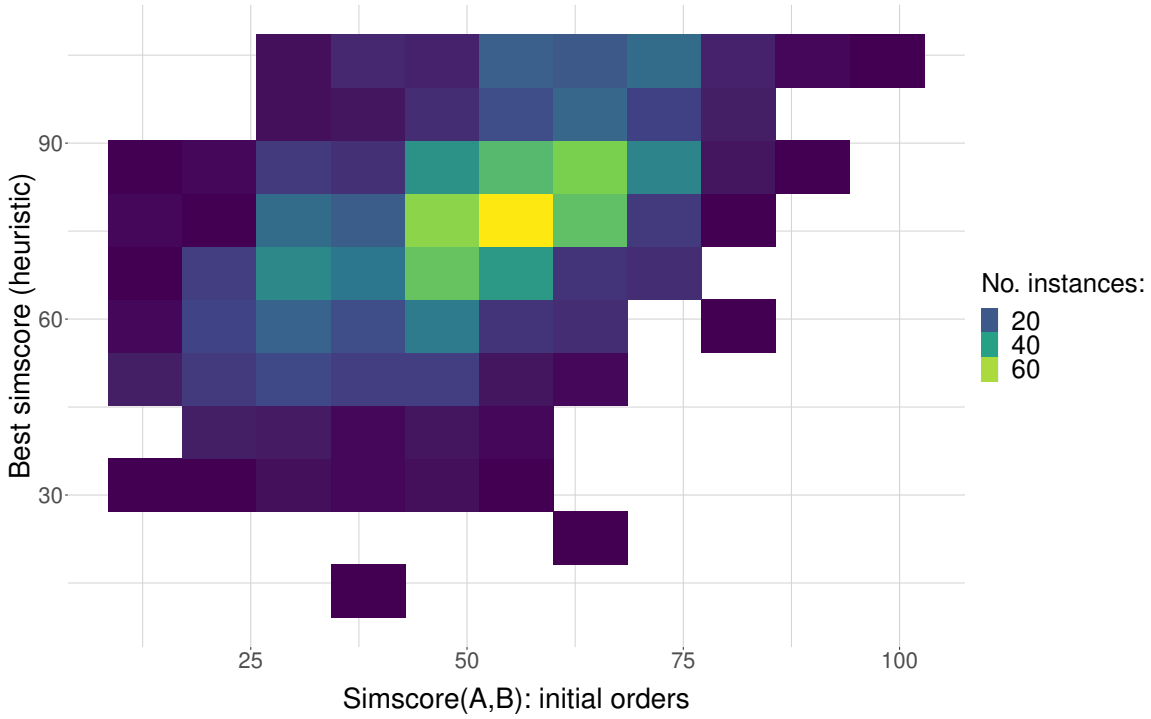

was similar to a true optimum (with simscore of 75% or more). Also, the more similar the initial sequences are, the more likely the heuristic will find a solution close to a true optimum, as suggested by Figure 3.20.

Figure 3.19: Similarity scores




*Note.* Scores calculated between the solution obtained by the proposed heuristic and a true optimum: min and max values across all optima.

Figure 3.20: Effect of the initial BDD orders



# Chapter 4

# A BDD-based Approach to a Facility Location Problem

We next illustrate the performance of the algorithms presented in the previous chapter in the context of a specific application. This chapter is organized as follows. We start by introducing the problem and show the corresponding mixed integer program. Sections 4.1 and 4.2 present an alternative formulation of the same problem, in which each group of constraints corresponds to a BDD, and the whole problem is reformulated as a Consistent Path Problem (CPP). We will design algorithms that build two diagrams that together represent the original problem instance. The remainder of the chapter is devoted to numerical experiments. In particular, Section 4.3 presents several ways to solve the problem (including one based on the results of Chapter 3), and benchmarks the solution times for different methods against a dataset of random instances. Section 4.4 demonstrates the runtime breakdown into the key steps of the proposed computational pipeline, and Section 4.5 concludes with an empirical discussion of the performance of the algorithms presented in the previous chapter depending on the problem structure.

We will consider the following variant of the uncapacitated facility location problem (see, e.g., Owen and Daskin 1998, ReVelle et al. 2008), which we call the "typed uncapacitated facility location problem" (t-UFLP). The t-UFLP considers a set of $M$ points. At each point

$i$, we can locate a facility at a cost given by $f_i$, covering all points in a set given by $S_i$, where $i \in S_i$. (Set $S_i$ might refer to customers that are sufficiently close to location $i$ according to some specified metric, like distance or travel time.) Furthermore, there exists a set of $K$ "types," such that a point is classified by exactly one type. Let $T_t$ be the set of points of type $t$, for $t = 1, \ldots, K$. The set of all points is partitioned into non-empty sets $T_1, \ldots, T_K$. For each type $t$, there exists a type budget constraint, which states that the maximum number of facilities located at type-$t$ points must be no more than some positive integer parameter $k_t$. The t-UFLP minimizes the sum of facility location costs required to cover every point at least once, subject to the given budget constraints. (The t-UFLP subsumes the UFLP and is thus NP-hard.)

We introduce the decision variables $x_i \in \{0, 1\}$ corresponding to locating a facility at point $i = 1, \ldots, M$. The objective minimizes the sum of the corresponding costs, and the following binary program models the t-UFLP.

$$\min \sum_{i=1}^{M} f_i x_i \tag{4.1a}$$

$$\text{s.t.} \sum_{j \in S_i} x_j \geq 1 \qquad \text{for all } i = 1, \ldots, M, \tag{4.1b}$$

$$\sum_{j \in T_t} x_j \leq k_t \qquad \text{for all } t = 1, \ldots, K, \tag{4.1c}$$

$$x_i \in \{0, 1\} \qquad \text{for all } i = 1, \ldots, M. \tag{4.1d}$$

Constraints (4.1b) ensure every point is covered at least once, while (4.1c) ensure that the available budget $k_t$ for each type $t$ is not exceeded.

Alternatively, we could formulate the problem as a CPP with two diagrams: a *cover* diagram enforcing constraints (4.1b), and a *type* diagram enforcing constraints (4.1c). Both diagrams have variables $x_i$ associated with their layers (perhaps, in different order). We first discuss construction of the diagrams and the difference in their order of variables, and then move on to the solution of the resulting CPP instance.

## 4.1 Building the cover diagram

We build the cover diagram layer by layer, each one corresponding to a point in the original t-UFLP and the decision whether to locate a facility at this point. We associate an $M$-dimensional binary *state* vector with each node of the diagram (indicating whether each corresponding point is covered). Covering a point $j$ necessitates locating a facility from $S_j$. Therefore, if we have made all the decisions regarding points from $S_j$ and $j$ is not covered, the solution must be infeasible. We start with a simple illustration of the procedure, followed by the formal presentation of an algorithm based on this idea.

**Example 4.1.** Consider the construction of a cover diagram starting with some point $j$ as the first candidate for locating a facility, with the following relevant adjacency lists: $S_j = \{j, 1, 2, 3\}$, $S_1 = \{j, 1, 3, \ldots\}$, $S_2 = \{j, 2, 3, \ldots\}$, $S_3 = \{j, 1, 2, 3, \ldots\}$. (Here the ellipses stand for arbitrary points excluding any combinations of $j, 1, 2$, and 3.) Figure 4.1 illustrates the corresponding network. An edge between points indicates that a facility at either of the two points can cover the other one. The resulting part of the cover diagram is presented in Figure 4.2. To build it, we associate a state with each BDD node, denoted by $(\ldots, c_j, c_1, c_2, c_3, \ldots)$, where $c_i$ is a binary variable accounting for whether or not point $i$ was covered by any facility ($i = j,\ 1,\ 2$, or 3). We start from a root-node state, marked $(\ldots, 0, 0, 0, 0, \ldots)$, where none of the points are covered. Then, if we decide to locate a facility at $j$ (corresponding to a one-arc emanating from the root node), we cover all four of those points, leading to state $(\ldots, 1, 1, 1, 1, \ldots)$. Otherwise, we preserve the state $(\ldots, 0, 0, 0, 0, \ldots)$ in the next layer. Assuming $x_j = 0$, if we further decide to locate a facility at point 1, then we will traverse the solid arc to state $(\ldots, 1, 1, 0, 1, \ldots)$, because we covered points $j$, 1, and 3. After the decision related to $x_3$ is taken, it is certain that there can be no subpath starting at state $(\ldots, 0, 0, 0, 0, \ldots)$ and ending at the **T** terminal (because there is no way to cover $j$ anymore). We call such states *infeasible*. The other states (marked by a box in Figure 4.2) are compatible with some set of feasible location decisions. Adding layers corresponding to points in the adjacency list in arbitrary order and merging the very last

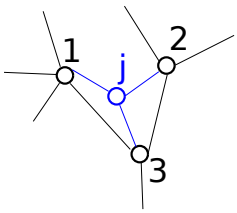



Figure 4.1: Example t-UFLP structure

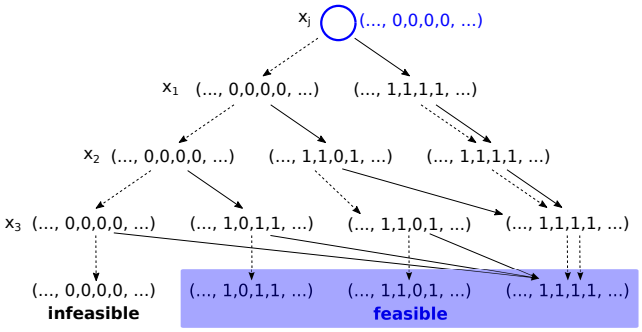



Figure 4.2: Cover BDD example

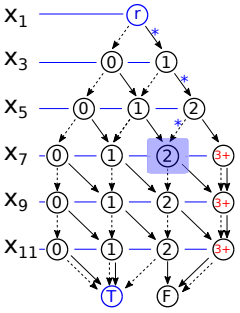



Figure 4.3: Type BDD example

layer into two nodes (feasible and infeasible), we can build the whole diagram encoding a single constraint from constraint (4.1b). We then pick another cover constraint and proceed by building the diagram using the previous feasible node as the new root. Note that once we fix point $j$, we process its entire adjacency list (we describe our ordering strategy in detail further). The resulting diagram has a certain fixed order of variables, which we call a *natural variable order* for the cover diagram. □

We denote the creation of a new BDD node as: "**add node** to $D$: X $\xrightarrow{\text{hi/lo}}$ *(new)* Y," which would mean that a new node, $Y$, is created as a child node of $X$, and that they are connected with an arc of type `HI` (one-arc), or `LO` (zero-arc), respectively. Creation of an arc between nodes $X$ and $Y$ is denoted by "**add arc** to $D$: X $\xrightarrow{\text{hi/lo}}$ Y ."

Using this notation, construction of the cover BDD is presented in Algorithm A4.1. We start by initializing the diagram with a single node (`r`) and the associated state $(0, \ldots, 0)$ (meaning no points are covered in the beginning). For each point $i$, we also track the number $F_i$ of adjacent points that have not yet been added to the diagram as decisions (initialized in line 3). When this number drops to zero for some point (we will call such point *critical*), there will be no further opportunities to cover it. Hence, if it has not been already covered, then no feasible solution is reachable from this state. The set of points to add to the diagram is denoted by $P$ (initialized in line 2 with the set of all points). In the main loop (spanning lines 5–47) we process the next point as follows. We pick the point that has fewest adjacent points to be added to the diagram (in line 5), and iterate over its adjacency list (ordered by

**Algorithm A4.1.** Build-cover-BDD

**Input:** Adjacency lists $S_j$ and facility location costs $f_j$ for $j = 1, \ldots, M$
**Output:** Cover diagram, $D$.

**create** $D$ with a single node, `r`
**initialize** $P \leftarrow \{1, \ldots, M\}$
**initialize** $F_i \leftarrow (|S_i| + 1)$ for all $i = 1, \ldots, M$.
**initialize** `next-layer` $\leftarrow \{$`state` $= (0, \ldots, 0) :$ `r`$\}$
**for** $i \in P$ **do**
    **for** $j \in \{i\} \cup S_i \cap P$ **do**
        $P \leftarrow P \setminus \{j\}$
        `current-layer` $\leftarrow$ `next-layer`
        **create** an empty layer `next-layer` and assign label $x_j$ to it.
        `critical-nodes` $\leftarrow \varnothing$
        **for** $q \in S_j$ **do**
            $F_q \leftarrow F_q - 1$
            **if** $F_q = 0$ **then**
                `critical-nodes` $\leftarrow$ `critical-nodes` $\cup \{q\}$
            **end if**
        **end for**
        **if** exists `infeasible-node` **then**
            **add node** to $D$: `infeasible-node` $\xrightarrow{\text{hi}}$ *(new)* `infeasible-node`$'$
            **add arc** to $D$: `infeasible-node` $\xrightarrow{\text{lo}}$ `infeasible-node`$'$
            `infeasible-node` $\leftarrow$ `infeasible-node`$'$
        **end if**
        **for** `state` $\in$ `current-layer` **do**
            **if** `state` $\in$ `next-layer` **then**
                **add arc** to $D$: `current-layer(state)` $\xrightarrow{\text{lo}}$ `next-layer(state)`
            **else**
                **if** any $\texttt{state}_q = 0$ for $q \in$ `critical-nodes` **then**
                    **if** exists `infeasible-node` **then**
                        **add arc** to $D$: `current-layer(state)` $\xrightarrow{\text{lo}}$ `infeasible-node`
                    **else**
                        **add node** to $D$: `current-layer(state)` $\xrightarrow{\text{lo}}$ *(new)* `infeasible-node`
                    **end if**
                **else**
                    **add node** to $D$: `current-layer(state)` $\xrightarrow{\text{lo}}$ *(new)* `next-layer(state)`
                **end if**
            **end if**
            `next-state` $\leftarrow (0, \ldots, 0)$
            **for** $q = 1$ to $M$ **do**
                `next-state`$(q) \leftarrow 1$ **if** $q \in S_j$, or `state`$(q)$ otherwise
            **end for**
            **if** `next-state` $\in$ `next-layer` **then**
                **add arc** to $D$: `current-layer(state)` $\xrightarrow{\text{hi}}$ `next-layer(next-state)` (with edge length $f_j$)
            **else**
                **add node** to $D$: `current-layer(state)` $\xrightarrow{\text{hi}}$ *(new)* `next-state` (with edge length $f_j$)
            **end if**
        **end for** *// iterating over states in* ***`current-layer`***
    **end for** *// iterating over the adjacency list* $S_j$
**end for** *// iterating over* $i \in P$
**merge** nodes in `next-layer` into a single node and **mark** it as **T** node in $D$.
**mark** `infeasible-node` (if exists) as **F** node in $D$.
**return** $D$.

point number) creating a new layer for each point in the list. Then we calculate the set of critical points (lines 10–16), i.e., those points that necessarily need to be covered after we have added the current point to the diagram. If we had an infeasible state in the previous layer, we create it in the current layer as well (lines 17–21). Then, we process each node corresponding to a feasible state in the current layer, and create its children nodes in the next layer as necessary. For zero-arcs, we make sure all critical points are covered (lines 26–31; otherwise we link the node to an infeasible state in the next layer). For one-arcs we just create children nodes as necessary. Finally, in the last layer we merge all feasible nodes into a single **T** node, and mark `infeasible-node` as the **F** node (if it exists). Note that all one-arcs are associated with costs of the respective facility locations, and zero-arcs have zero costs.

## 4.2 Building the type diagram

After the cover diagram is created we build the type diagram. We will again start the presentation with an example that will serve a foundation of the algorithm. We structure the diagram in $K$ blocks, one for each facility type. Consider the first such block, which encodes a constraint $\sum_{j \in T_t} x_j \leq k_t$ for some $t$. Starting from the root node, we associate each node of the diagram within a layer with the number of facilities of type $t$ located at the corresponding state.

**Example 4.2.** Consider an example diagram encoding a condition $\sum_{j \in T} x_j \leq k$ for $T = \{1, 3, 5, 7, 9, 11\}$ and $k = 2$. For an arbitrarily chosen order $(1,\ 3,\ 5,\ 7,\ 9,\ 11)$, the resulting diagram is presented in Figure 4.3. Every path corresponds to some assignment of variables $x_i$, for $i \in T$. Every node is associated with the number of facilities of the given type located so far. For example, choosing $x_1 = x_3 = 1$ and $x_5 = 0$ implies that two facilities have been located; since $k = 2$, we cannot locate any more facilities this type. These decisions correspond to a part of the path starting from the root node (marked with stars in the figure), and the resulting node is marked with a rectangle. Nodes marked "3+" correspond

to *infeasible states* (no subpaths originating from them and ending at **T** exist). Then, we can choose another budget constraint, and continue building the diagram starting from the **T** node of the previous one, and so on. Note that while the order of variables within each block does not matter, it is desirable to process all variables in some set $T_t$ at once. In our implementation we use this freedom to minimize the number of inversions between the type and cover diagrams (making the alignment problem easier), and call the resulting order of variables a *natural variable order* for the type diagram. □

The main logic of building the type diagram is formally presented in Algorithm A4.2. We create the diagram layer by layer, in blocks. Each block (created with a loop in lines 3–35) corresponds to a single facility type. Within each one, we assign an integer *state* to each BDD node, representing the number of facilities of the given type located by the moment. We start with a layer containing the single "0" state and the associated root node, `r` (initialization in line 2). We also keep track of the infeasible state: Once created, we update the corresponding node in each layer (lines 7–11) in a way so that there is no subpath from any such node to the **T** terminal. Any arcs emanating from an infeasible node end either in the infeasible node of the next layer, or in the **F** terminal. The core procedure is implemented in lines 12–32. We iterate through each state in the current layer, and for each state we make sure the tail nodes are created for a zero-arc (lines 13–17), and for a one-arc (lines 18–30). That is, we create them if they do not exist, or link to existing nodes if they do. Note that if the budget for the current type is exceeded, we do not create another state in the next layer, but link to the infeasible state (by either creating an infeasible node, or linking to an existing one if the infeasible state has been created before, lines 21–30). After processing each layer, we assign it a corresponding label in line 31.

After we process a type, we merge all nodes corresponding to a feasible state into a single node (line 34) to either repeat the same procedure for the next type, or mark the resulting two nodes as **T** and **F** (line 36) if all facility types are processed. Note that all arcs assume zero costs in a type diagram.

As a final remark, note that we have a freedom to choose the order of facility types

**Algorithm A4.2.** Build-type-BDD

**Input:** Facility types $T_t$ and budgets $k_t$ for $t = 1, \ldots, K$
**Output:** The type diagram, $D$.

**create** $D$ with a single node, `r`.
`next-layer` $\leftarrow \{$`state` $= 0 :$ `r`$\}$
**for** $t = 1$ to $K$ **do**
  **for** $j \in T_t$ **do**
    `current-layer` $\leftarrow$ `next-layer`
    `next-layer` $\leftarrow \varnothing$
    **if** exists `infeasible-node` **then**
      **add node** to $D$: `infeasible-node` $\xrightarrow{\text{hi}}$ *(new)* `infeasible-node`$'$
      **add arc** to $D$: `infeasible-node` $\xrightarrow{\text{lo}}$ `infeasible-node`$'$
      `infeasible-node` $\leftarrow$ `infeasible-node`$'$
    **end if**
    **for** `state` $\in$ `current-layer` **do**
      **if** `state` $\in$ `next-layer` **then**
        **add arc** to $D$: `current-layer(state)` $\xrightarrow{\text{lo}}$ `next-layer(state)`
      **else**
        **add node** to $D$: `current-layer(state)` $\xrightarrow{\text{lo}}$ *(new)* `next-layer(state)`
      **end if**
      **if** `state` $+ 1 \in$ `next-layer` **then**
        **add arc** to $D$: `current-layer(state)` $\xrightarrow{\text{hi}}$ `next-layer(state+1)`
      **else**
        **if** `state` $+ 1 > k_t$ **then**:
          **if** exists `infeasible-node` **then**
            **add arc** to $D$: `current-layer(state)` $\xrightarrow{\text{hi}}$ `infeasible-node`
          **else**
            **add node** to $D$: `current-layer(state)` $\xrightarrow{\text{hi}}$ *(new)* `infeasible-node` within `next-layer`
          **end if**
        **else**
          **add node** to $D$: `current-layer(state)` $\xrightarrow{\text{hi}}$ *(new)* `next-layer(state+1)`
        **end if**
      **end if**
      **assign label** $x_j$ to layer `current-layer` in $D$.
    **end for** *// iterations over* ***`state`*** $\in$ ***`current-layer`***
  **end for** *// iterations over* $j \in T_t$
  **merge** nodes with `state` $\leq k_t$ in `next-layer`.
**end for** *// iterations over* $t = 1, \ldots, K$
**mark** `infeasible-node` (if exists) as **F** node, and the other node in `next-layer` as **T** node in $D$.
**return** $D$.

(in the loop starting at line 3) and facilities within a type (in the loop starting at line 4). In our implementation we choose these orders to minimize the number of inversions between the order of variables of the type and cover decision diagrams, and call this a *natural variable order* for the type diagram.

## 4.3 Solving the CPP representation of t-UFLP

Having two diagrams encoding the feasibility set allows us to state the t-UFLP as a CPP, which can then be formulated as another mixed integer program, as follows. We introduce (continuous) variables $v_{ab}^D$ to denote the flow between nodes $a$ and $b$ of diagram $D$ and binary variables $x_i$ to correspond to the decisions of the original t-UFLP. Using this notation, (4.1) can be reformulated as follows.

$$\min \sum f_i v^{\mathtt{C}}_{i,\mathtt{HI(i)}}, \tag{4.2a}$$

$$\text{s.t.} \sum_{i:\mathtt{HI(i)}=u \text{ or } \mathtt{LO(i)}=u} v^{\mathtt{C}}_{iu} = \sum_{j:\mathtt{HI(u)}=j \text{ or } \mathtt{LO(u)}=j} v^{\mathtt{C}}_{uj} \text{ for all } u \in L^{\mathtt{C}}_2 \cup \ldots \cup L^{\mathtt{C}}_{(M-1)}, \tag{4.2b}$$

$$\sum_{j:\mathtt{HI(r)}=j \text{ or } \mathtt{LO(r)}=j} v^{\mathtt{C}}_{\mathtt{r}j} = 1, \tag{4.2c}$$

$$\sum_{j:\mathtt{HI(j)}=u \text{ or } \mathtt{LO(j)}=u} v^{\mathtt{C}}_{ju} = 1 \text{ for } u \in \{\mathbf{T}^{\mathtt{C}}, \mathbf{F}^{\mathtt{C}}\}, \tag{4.2d}$$

$$\sum_{i:\mathtt{HI(i)}=u \text{ or } \mathtt{LO(i)}=u} v^{\mathtt{T}}_{iu} = \sum_{j:\mathtt{HI(u)}=j \text{ or } \mathtt{LO(u)}=j} v^{\mathtt{T}}_{uj} \text{ for all } u \in L^{\mathtt{T}}_2 \cup \ldots \cup L^{\mathtt{T}}_{(M-1)}, \tag{4.2e}$$

$$\sum_{j:\mathtt{HI(r)}=j \text{ or } \mathtt{LO(r)}=j} v^{\mathtt{T}}_{\mathtt{r}j} = 1, \tag{4.2f}$$

$$\sum_{j:\mathtt{HI(j)}=u \text{ or } \mathtt{LO(j)}=u} v^{\mathtt{T}}_{ju} = 1 \text{ for } u \in \{\mathbf{T}^{\mathtt{T}}, \mathbf{F}^{\mathtt{T}}\}, \tag{4.2g}$$

$$\sum_{j \in L^{\mathtt{C}}_q} v^{\mathtt{C}}_{j\mathtt{HI(j)}} = x_q \text{ for all } q = 1, \ldots, M, \tag{4.2h}$$

$$\sum_{j \in L^{\mathtt{T}}_q} v^{\mathtt{T}}_{j\mathtt{HI(j)}} = x_q \text{ for all } q = 1, \ldots, M, \tag{4.2i}$$

$$v_{pq} \geq 0 \text{ for all valid } p,\ q, \tag{4.2j}$$

where index `T` corresponds to the type diagram, and `C` corresponds to the cover diagram. In the objective function we have a sum of all flows along one-arcs of the cover diagram weighted by the corresponding facility location costs. Constraints (4.2b)–(4.2d) are flow continuity constraints at every node of the cover diagram (with the last two groups corresponding to the first and the last layer of the diagram, respectively). Constraints (4.2e)–(4.2g) are similar constraints for the type diagram. The last two groups of binary constraints, (4.2h) and (4.2i), link the two shortest-path instances using shared variables $x_q$.

An alternative to this approach is to align these two diagrams and build an intersection BDD (e.g., see an illustration in Figure 4 by Lozano et al. 2020). The latter constitutes the focus of this chapter. Then, if we have the feasibility set encoded by a single BDD, the solution can be obtained by solving a shortest-path problem through the diagram. As pointed out before, this can be done efficiently (since this is an acyclic layered graph) in terms of the BDD size, and does perform well in practice as we illustrated with numerical experiments, although the size of the diagram itself can be exponential in the parameters of the original t-UFLP.

In terms of aligning BDDs, we will compare the baseline "Greedy sifts" and "Best of A and B" strategies mentioned in Section 3.5.4 to the proposed approach based on the simplified problem. We thus have four approaches to solve the t-UFLP: "Naive MIP" (involving no diagrams), "CPP MIP," "CPP+SP with Greedy sifts," and "CPP+SP with variable sequences (VS)."

To illustrate the relative performance of different approaches to align the diagrams we run the following experiment. We vary the number of points from $M = 5$ to 15, generating 200 instances of t-UFLP for each value. We then solve each of these instances using the different heuristics described above. Note that all methods yield the same objective value: we build an intersection BDD that exactly encodes the feasible set (and solving shortest-path over such diagram gives an exact optimum).

To generate a random t-UFLP instance of the given size $M$, we first generate adjacency lists by creating the underlying network of points using Erdős-Rényi-Gilbert random

graph model (Gilbert 1959) with parameter $p = 0.3$. Location costs are chosen as random integers between 5 and 10. Number of types is generated randomly between 2 and $\min\{10, M\}$, type budget $k_t$ is taken at random from 1 to 5, and we ensure that there is at least one point associated with each type. If the problem happens to be infeasible (which we check using an MIP formulation), we discard the instance and repeat the procedure.

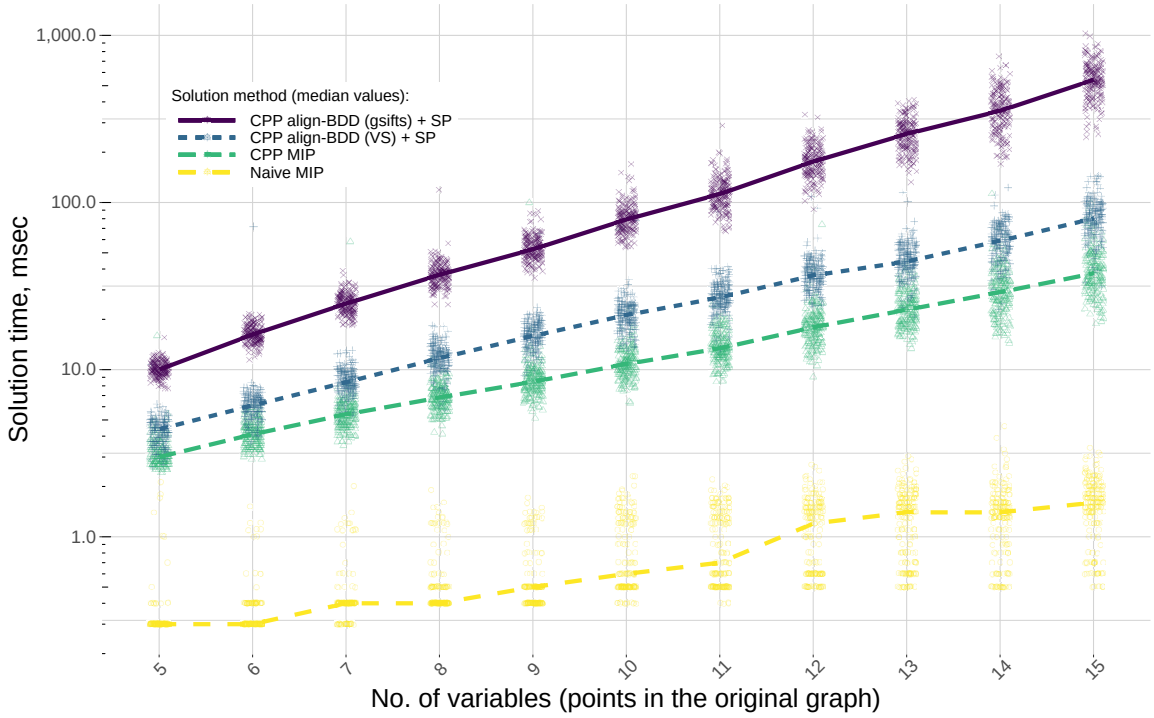
We summarize the solution times in Figure 4.4. First, as expected, solving a simple binary program with a state-of-the-art solver outperforms CPP-based approaches. This may or may not be the case for other problems and specific t-UFLP instances. From the CPP-based methods, CPP MIP and the simplified-problem based approaches are very close in terms of runtime, and both outperform greedy-sifts approach. However, it is worth noting that building intersection BDDs allows for sensitivity analysis, since the shortest-path problem is a linear program. This approach also facilitates easy re-solving of the problem if its structure remains the same, because solving the shortest-path problem over the intersection BDD is comparable in runtime with applying the state-of-the-art solver to tackle the whole problem, which we consider in more detail further.

## 4.4 Breakdown of the proposed heuristic runtime for t-UFLP

As solution approaches to t-UFLP based on CPP (involving BDDs) comprise several steps, we highlight the computational intensity of each step by discussing the corresponding runtimes. In particular, we generate $1,000$ t-UFLP instances and solve each one with the following three methods, reporting the histograms in Figure 4.5 for runtimes in logarithmic scale.

- **CPP+SP with variable sequences (VS)** is described with the first six panels. The first panel corresponds to building the type and cover diagrams. The second panel describes the simplified problem solution (a branch-and-bound process presented in Algorithm A3.5). The next three panels characterize runtimes for aligning the diagrams to the found order of variables, building the intersection BDD, and solving

Figure 4.4: Runtimes for t-UFLP




*Note.* Comparison of different heuristics' runtimes for t-UFLP as the problem size increases.

the shortest-path through this BDD, respectively. The sixth panel corresponds to the total runtime for the mentioned steps. For reference, we also present histograms of runtimes for the following alternative solution approaches.

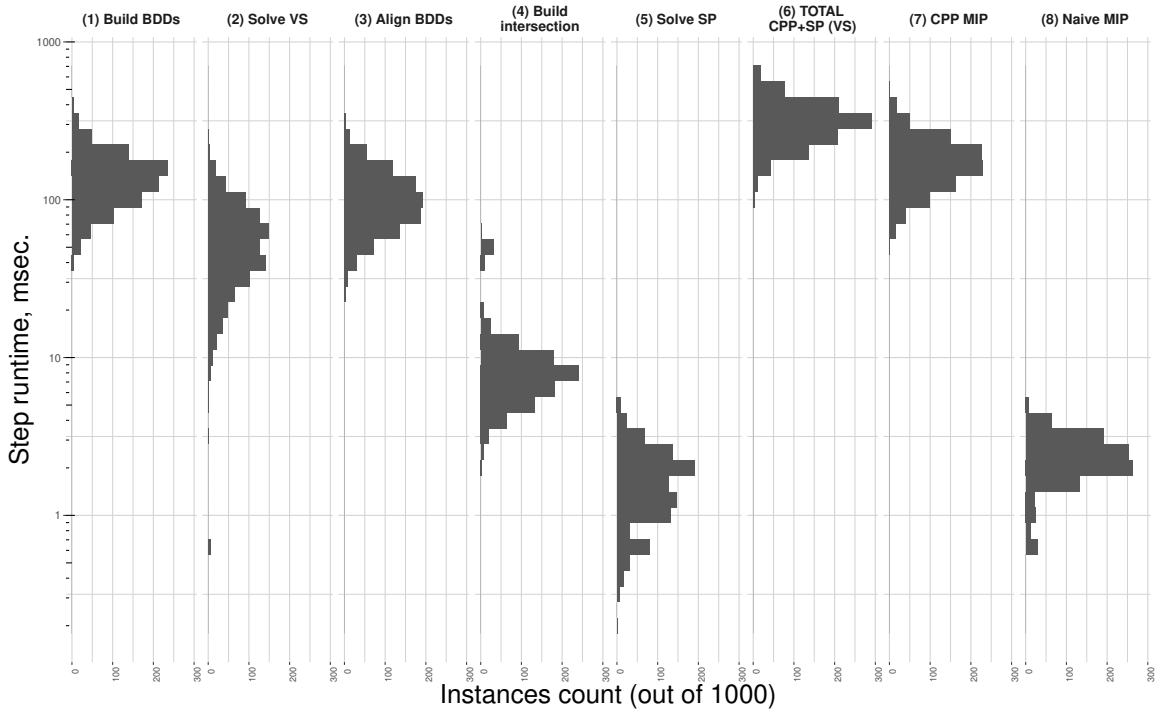
- **CPP MIP** runtimes are characterized by the seventh panel. They comprise building the diagrams (the same as the first step above), building an integer program for the CPP, and solving this MIP with Gurobi solver.

- **naive MIP** runtimes are presented in the last panel, which include building and solving a mixed integer program without constructing any diagrams at all.

We report these results for three of the four methods mentioned in Section 4.3, since the fourth one, based on the "Greedy sifts" alignment strategy, takes significantly more time (as evident from Figure 4.4).

The initial steps of CPP+SP (VS) approach (building the initial diagrams, solving the simplified problem, and aligning the diagrams) take a relatively long time. However, the process of building the intersection decision diagram is faster, and the overall time is comparable to the CPP MIP approach. Moreover, the final step of solving the shortest-path problem over the intersection diagram takes very little time, and is comparable to the baseline of solving a simple MIP formulation. (Unlike the baseline, the shortest-path problem is a convex optimization problem that yields valuable sensitivity information.) This observation reveals that re-solving the t-UFLP with different cost data is very practical using the CPP approach. If only the cost data changes, then the intersection BDD network topology remains the same. Therefore, we can replace cost values directly on the intersection BDD, and the re-solve requires only the very fast step of solving a shortest-path problem over a binary decision diagram.

Figure 4.5: Breakdown of the runtimes into solution steps for a CPP




## 4.5 On performance of the BDD alignment

Finally, the t-UFLP also provides insight into the efficiency of the proposed BDD alignment strategy depending on the structural properties of the problem. The "variable sequence" model we propose assumes an increase in BDD size with every swap (as far as possible in a binary diagram). Its ability to find good variable orders suffers if some swaps do not yield such changes. For example, consider a type diagram used in solving a t-UFLP instance. By construction, that diagram consists of several blocks (one per type), and the position of any two blocks can be exchanged without any changes in the diagram size. Moreover, within each such block the variables corresponding to points of the t-UFLP can be ordered arbitrarily. Therefore, there must be many potential swaps where our heuristic would overestimate the corresponding change in the objective.

To illustrate this dependence on the problem structure, we compared the resulting intersection diagram sizes obtained from the heuristic based on the simplified problem to

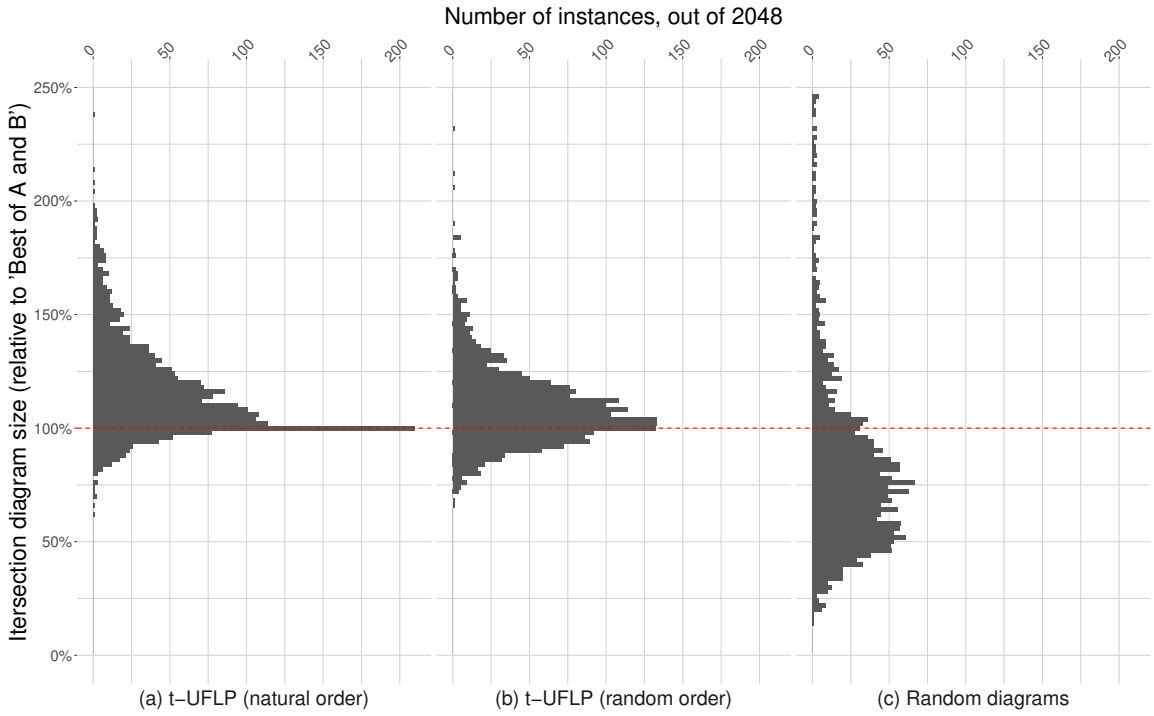
a simple baseline of "Best of A and B" (see Section 3.5.4) over several different types of instances, in a separate numerical experiment. The results are summarized in Figure 4.6.

First, we performed this comparison over $2,048$ random t-UFLP instances, each having 20 points. Figure 4.6a shows that the simplified problem only occasionally helps to reduce the intersection diagram size as compared to a naive heuristic. As discussed above, this is partially due to a special structure of our diagrams. In particular, there are potential consecutive swaps that do not actually change the diagram size; however, corresponding swaps in related weighted variable sequences might yield an increase in the objective. This includes swaps within a block in the type diagram and swaps corresponding to points in the same adjacency list in the cover diagram. We designed another set of instances in a way to break this special structure of the diagrams. This experiment used the same number of t-UFLP instances of the same size, but with the layers of both diagrams shuffled after their construction before formulating the align-BDD problem. The performance of both heuristics decreases as compared to the initial setup, but the relative objective of the proposed alignment strategy improves. Figure 4.6b shows that our heuristic provides better results for about 30% of the instances (instead of 15–20% in the previous set up).

Note, however, that the diagrams still possess some structural properties that are difficult for our simplified problem to capture. Because we only shuffle the order of the layers, but do not actually change the nature of the instances, there still must be swaps for which our heuristic overestimates the effect on the objective. To quantify this effect, we generated the same number of align-BDD instances with the same number of layers, but comprising random diagrams generated as per Section 3.5.1 of the previous chapter, instead of the ones constructed from t-UFLP instances. The relative performance of the proposed heuristic is presented in Figure 4.6c. These results show that the heuristic outperformed the baseline on more than 70% of the instances (as compared to about 30% in the previous setup with t-UFLP instances).

Therefore, the proposed heuristic appears to be sensitive to symmetries in the problem structure. Those symmetries can induce swaps that do not change the BDD size, but

Figure 4.6: Intersection diagram size (tUFLP vs. random diagrams)




*Note.* Number of nodes in the intersection diagram obtained with the proposed heuristic relative to "Best of A and B" heuristic.

are modeled to do so by our weighted variable sequence bound.

While the proposed heuristic does prove to be valuable in some settings, there are also other situations in which there is a certain structure in the problem (when we know a priori that certain swaps do not yield the diagram growth) that might significantly decrease the relative performance of a heuristic based on the variable sequence concept. In these cases leveraging the problem structure might yield significant computational benefits and outperform many general-purpose approaches.

# Chapter 5

# Monte Carlo Tree Search Framework for DSPI

This chapter is devoted to another network optimization problem, and a significantly different approach to building a decision diagram. After briefly introducing the problem in Section 5.1, we present the proposed framework as applied to the DSPI problem in Section 5.2, outlining the key conceptual details of the implementation in Sections 5.2.1–5.2.3. We then formally present our recommended algorithms in Section 5.3 and discuss their correctness in Section 5.4, concluding with the numerical experiments in Section 5.5.

## 5.1 Problem formulation

Dynamic Shortest-Path Interdiction (DSPI) is one of the more complicated variants of the widely studied problem of the Shortest-Path Interdiction. It is a dynamic, two-player game over a directed graph $G$, introduced by Sefair and Smith (2016). An instance is parameterized by a graph with sets of nodes $\mathcal{N}$ and arcs $\mathcal{A}$, two special nodes $s$ and $t$ (called *source* and *terminal*, respectively), and a *budget* $b \in \mathbb{N}$. Each edge $(i, j) \in \mathcal{A}$ possesses a cost $c_{ij}$ and an *interdiction cost increment* $d_{ij}$.

One player, the *Evader* seeks to traverse the graph moving from the source to the

terminal at the minimum possible cost. The other player is the *Interdictor*, who seeks to maximize the Evader's minimum cost by attacking (*interdicting*) the arcs. The game starts with the Evader's position at the source and unfolds in turns. The Interdictor moves first and is allowed to attack any subset of arcs (including the empty one); then the Evader traverses an arc, and the players keep making turns until the Evader reaches the terminal. The Interdictor has a cardinality constraint (the budget $b > 0$) on the number of arcs it can interdict during the game, but any number of arcs within this limit can be interdicted during each particular turn (including none).

The cost for the Evader to traverse arc $(i, j)$ is defined to be $c_{ij}$ if not interdicted, and $c_{ij} + d_{ij}$ otherwise. We will denote the set of interdicted arcs by $S$ and the cost of the arc traversal as:

$$\widetilde{c}_{ij}(S) = \begin{cases} c_{ij} + d_{ij}, \text{ if } (i,j) \in S, \\ c_{ij} \text{ otherwise.} \end{cases}$$

Sefair and Smith (2016) demonstrated that there exists an optimal solution in which the Interdictor only attacks a (possibly empty) subset of arcs incident to the current Evader's position. Therefore, given the Evader's current position $i$ and the interdiction set $S$, and denoting the optimal game cost as $z^*(S, i)$, the Interdictor's problem can be formally described with the following recursive formula (Sefair and Smith 2016):

$$z^*(S,i) = \max_{S' \subseteq \mathrm{FS}(i) \setminus S \,:\, |S \cup S'| \leq b} \Big\{ \min_{j \in \mathrm{FS}(i)} \{ z^*(S \cup S', j) + \widetilde{c}_{ij}(S \cup S') \} \Big\},$$

where $\mathrm{FS}(i) = \{(i, j) : \ (i, j) \in \mathcal{A}\}$ denotes the forward star of node $i$.

Sefair and Smith (2016) have demonstrated that the decision variant of this problem is NP-hard and proposed an exact dynamic programming (exponential-time) algorithm to obtain a solution for the case of a general graph $G$, a polynomial-time algorithm for a Directed Acyclic Graph (DAG), and several approaches to find upper and lower bounds on the optimal objective. We build upon these results by designing a heuristic algorithm based on the Monte Carlo Tree Search (MCTS) approach, and demonstrating its performance on

a set of randomly generated and pre-defined instances. We will parameterize the MCTS with a *computational budget*, in order to obtain feasible solutions (valid sequences of player turns) even for relatively large instances.

## 5.2 Monte Carlo Tree Search framework

The Monte Carlo Tree Search (MCTS) is a randomized algorithm that uses relatively simple *play-out* heuristics to estimate the value for game states (e.g., Monte Carlo simulations in the form of random moves for both players) and to focus the search on more promising states using the obtained information. More detailed discussion of this approach can be found in many sources, e.g., see Sutton and Barto (2018) for a general presentation (along with many other techniques and ideas on reinforcement learning) and Browne et al. (2012) for an overview on the MCTS literature. Here we present the method as applied to the DSPI.

The algorithm builds a *game tree* that incorporates the information known about the different states of the game (corresponding to both players' turns). It can be classified as a *Decision-time* algorithm, as pointed out by Sutton and Barto, in the sense that it is designed to start from some *current state* that happened in the game, and recommend the next action to be taken by the corresponding player. Of course, we reuse the information obtained between the turns, but updating the root node of the tree narrows the search and discards large parts of the state space as the game unfolds. Informally speaking, the procedure mimics a human when playing a game: Starting from the current situation, try to predict good turns for both players until a certain depth is reached, and then assess the situation using some computationally cheap procedure. People tend to use more complicated heuristics to assess the situation than random play-outs (usually described along the lines of developed *game intuition*). However, even such a simple approach yields some success, see, e.g., a discussion by (Sutton and Barto 2018). Improving such heuristics, e.g., via simulation enhancements (Browne et al. 2012) can further improve the quality of this approach. Note

that, for example, AlphaGo Zero used a dedicated neural network to guide its simulations, as discussed by Silver et al. (2017).

More formally, as applied to DSPI, each game tree node (indexed by $j$ here) includes the following information.

**Information associated with a game tree node:**

- $\mathrm{p}(j) \in \mathcal{N}$: Current Evader's position, where the game is concluded if $\mathrm{p}(j) = t$,
- $S_j \subseteq \mathcal{A},\ 0 \leq |S_j| \leq b$: interdiction decisions taken up to this point ($b - |S_j|$ gives the residual budget available to the Interdictor),
- $\tau(j) \in \{\text{Interdictor, Evader}\}$: the player taking the next action at node $j$,
- $\widehat{Q}_j$: an estimate of the cost-to-go of the game (starting from the state implied by node $j$, assuming the first turn by player $\tau(j)$),
- $LB(j),\ UB(j)$: lower and upper bounds on the cost-to-go of the game,
- $N_j \in \mathbb{N}$: number of times the node was visited in the course of analysis,
- `children`$(j)$: a list of child nodes, corresponding to actions taken by player $\tau(j)$ at game state implied by $j$,
- `actions`$(j)$: actions available to player $\tau(j)$ that do not correspond to any existing child nodes.

Strictly speaking, available actions can be determined from the game *state*, which comprises the interdiction decisions $S$, current Evader's position $\mathrm{p}(j)$, and the current turn $\tau(j)$ (along with the graph information). We keep track of this explicitly to simplify the notation.

Note that to avoid enumerating all possible subsets of arcs, we introduce *virtual* turns for the Interdictor. That is, we allow the Interdictor to attack only one arc at each game tree node, but the turn switches to the Evader only after the Interdictor decides to

attack no further arcs (i.e., *passes*). Therefore, all child nodes of an Evader's node are Interdictor's nodes, but there is exactly one child Evader's node for every Interdictor's node (the rest, if present, are also Interdictor's nodes).

The algorithm is parameterized by a computational budget parameter $K$, which limits the number of iterations of the game tree updates. Until this limit is reached we keep updating and refining the the data contained within nodes, adding and removing nodes. Then, the algorithm performs the best action assuming the current game state defined by the root node. When a player makes the next turn (as recommended by MCTS or, in fact, any other algorithm), we update the root node of the game tree, prune the tree as necessary to remove the information regarding the states that are unreachable from the new current one, and run the procedure again to generate the next recommendation. This high-level procedure is formally presented in Algorithm A5.5, which is essentially a loop calling the procedure that performs a single iteration of the game tree update.

Figure 5.1 outlines the core of the algorithm, presented in several connected procedures and comprising four phases, which we together call a *learning episode*.

During the first phase, **Selection**, we start from the root node and recursively choose child nodes that are more promising according to our estimates, until we reach a leaf node of the game tree. Note that such leaf node is not necessarily a final node in the game (in which the Evader has reached the terminal), but instead just represents a point on the frontier of the information about the game the algorithm has collected so far.

After a leaf node is selected, the **Expansion** phase is started, with the aim of creating child nodes corresponding to all possible actions the current player can take starting from the selected node. Since every action restricts the accessible state space, we try to update lower and upper bounds on the objective for each new game tree node (as compared to the parent node).

Bounds alone do not always allow us to assess which of the child nodes are most promising. Accordingly, we build an estimate for the future game cost for each of the new nodes during the **Roll-outs** phase. Starting from each node created during the Expansion

phase, we simulate a simple game by choosing player actions uniformly at random and record the resulting objective as the cost-to-go estimate, $\widehat{Q}$.

Finally, we propagate this new information back from the leaf node we selected towards the root during the **Backpropagation** phase. We traverse the tree backwards and update the cost-to-go estimates along with the bounds for each parent node, to make them consistent with the information contained in the child nodes.

Before we proceed with the formal description of the algorithm in more detail, let us briefly touch two key problems that necessitate the complication of this general and relatively simple approach. The first (in Section 5.2.1) regards balancing exploration and exploitation, and the second (in Section 5.2.2) deals with tree pruning.

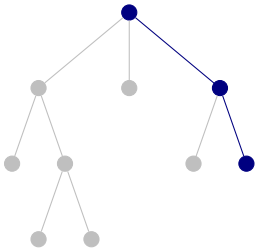
### 5.2.1 Exploration-exploitation trade-off

Building cost-to-go estimates by simple simulated games has inherent drawbacks. For example, consider a tree node that has five child nodes, only one of which is optimal, and the other four entail very poor objective values. Roll-outs passing though this node will converge to the average objective across the child nodes, which is poor by assumption. Because of that, the optimal node might be never selected for expansion. As a result, the algorithm would never refine the corresponding cost-to-go estimates and learn that the node is, in fact, optimal. Hence, we need a mechanism that considers nodes that are not the best ones in terms of our imperfect cost estimates, from time to time. This is an example of well-documented exploration vs. exploitation trade-off (see, e.g., Sutton and Barto 2018). Following the literature, we employ two ideas to alleviate this problem.

The first of these ideas is $\varepsilon$-greedy selection. During the selection phase, with some small probability $\varepsilon$ we just pick a node uniformly at random. However, with (large) probability $(1 - \varepsilon)$ we choose a node having the best cost estimate. The latter incorporates the cost for moving from one node $i$ to another node $j$, i.e., $\widetilde{c}_{ij} + \widehat{Q}_j$, not just $\widehat{Q}_j$. This allows the algorithm to potentially consider any node for exploration within the game tree.

The second idea uses Upper Confidence Bounds (Kocsis and Szepesvári 2006, Kocsis

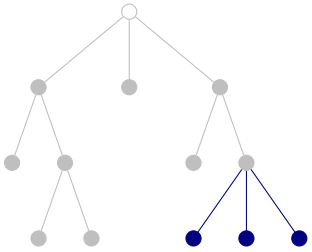
Figure 5.1: Summary of the key steps of MCTS–DSPI algorithm



**Phase 1: Selection.** Starting from the root node, a *tree policy* is used to choose the next node for further consideration (simulating a player's choice), until a leaf node is reached.

**Implementation:** Algorithm A5.1, lines 2–19.

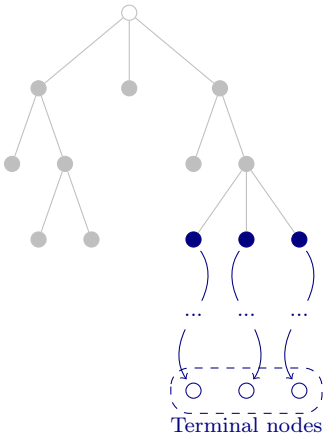
**Tree updates:** Best objective estimates for both players ($\alpha$ and $\beta$), child nodes can be pruned at each selection step (see Section 5.2.2 for details).



**Phase 2: Expansion.** All possible actions from the selected node are considered and instantiated as game tree nodes for further consideration.

**Implementation:** Algorithm A5.1, lines 20–39; Algorithm A5.3 (upper bound), Algorithm A5.4 (lower bound).

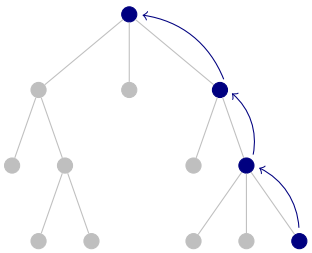
**Tree updates:** New nodes are created, respective upper and lower bounds on the further game cost are calculated for each new node. Some inconsistensies between parent and child nodes bounds can be introduced.




**Phase 3: Roll-outs.** A simple heuristic, random actions for both players, is used to build an estimate for further game cost from each of the new nodes until the end of the game. Game states encountered during the simulation are not recorded (only the resulting total costs are).

**Implementation:** Algorithm A5.2; Algorithm A5.1, line 40.

**Tree updates:** Cost-to-go estimate $\widehat{Q}_m$ for each new node $m$ is determined, and the best node is chosen for the backpropagation.



**Phase 4: Backpropagation.** The new information from the leaf nodes are propagated back to the root node.

**Implementation:** Algorithm A5.1, lines 41–52.

**Tree updates:** Lower and upper bounds of each parent node along the selected path are made consistent with these of the child nodes. The expected cost-to-go $\widehat{Q}$ of each node is updated with the best achievable cost given the child nodes (adjusted for the turn cost for Evader's turns).

et al. 2006). Even in presence of this randomness, our algorithm would ideally explore, at least minimally, some of the nodes that so far have received the least investigation. To incorporate the degree of our certainty about the given objective estimate, we seek to select a node that maximizes the following score (assuming parent game tree node $i$ and child node $j$):

$$R_j = \sigma_j(\widetilde{C}_{ij} + \widehat{Q}_j) + C_p\sqrt{\log(N_i)/N_j},$$

where the first term corresponds to our cost estimate, and the second term captures our uncertainty stemming from the finite number of simulations. The notation used here is as follows.

The cost $\widetilde{C}_{ij}$ is a shorthand for $\widetilde{c}_{\mathrm{p}(i),\mathrm{p}(j)}(S_i)$, i.e., cost of traversing arc $(\mathrm{p}(i), \mathrm{p}(j))$, if moving from node $i$ to $j$ implies any arc traversal at all. An auxiliary variable $\sigma_j$ takes the value of 1 if $\tau(j) =$ "Interdictor", and $-1$ otherwise, and is used to accommodate the fact that the Evader minimizes the game cost, while the Interdictor maximizes it. Therefore, the first term is just our estimate for the additional game cost if we start from node $i$ and pick $j$ as the next one (taking with the appropriate sign, so that maximizing the score would make sense).

The second term is designed to make under-explored nodes more attractive. Note that as the total number of visits for the parent node $N_i$ increases, if children other than $j$ are disproportionately selected, $\log(N_i)/N_j$ will increase, making node $j$ relatively more attractive. The square root is inspired by research concerning the multi-armed bandit problem, as it captures the variance (due to uncertainty) in the estimate of the action's value (see also a brief discussion by Browne et al. (2012) and Sutton and Barto 2018). The constant parameter $C_p$ regulates the degree of exploration (to what extent we are interested in expanding the tree as compared to choosing the nodes with the best cost estimates) and is chosen empirically.

Therefore, the selection procedure can be summarized as follows.

**Node selection criteria**

- At random, with (some pre-defined) probability $\varepsilon$: pick a child node uniformly at random.
- Otherwise, pick a one that maximizes the score including the cost-to-go estimates and the exploration term governed by parameter $C_p$:

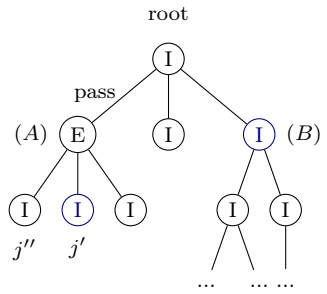
$$j^* \in \operatorname{argmax}_{j \in \texttt{children}(i)} \sigma_j(\widetilde{C}_{ij} + \widehat{Q}_j) + C_p\sqrt{\log(N_i)/N_j}.$$

### 5.2.2 Tree (alpha–beta) pruning

The second problem that significantly deteriorated the performance of our early experiments was the size of the tree for larger instances. It turns out to be relatively easy to adapt practically strong bounds developed by Sefair and Smith (2016) to prune our search tree in many cases, using an idea similar to alpha–beta pruning (see, e.g., Knuth and Moore 1975). To illustrate the concept consider a simple game tree presented in Figure 5.2. Interdictor's nodes are marked with "I", and Evader's ones with "E". Assume we have just expanded node $(A)$ on the left branch, and assume further that we have an extensive subtree developed after $(B)$. Observe that if the best child of $(A)$ from the Evader's perspective implies a game cost no more than that of $(B)$, then it is safe to prune $(A)$. Indeed, in this case the Interdictor is better off choosing $(B)$ as opposed to $(A)$ at the root node, while it has the power to choose, without giving the Evader this choice at $(A)$ in first place. Following the existing literature, we implement this idea as follows. We maintain two running bounds, $\alpha$ and $\beta$: the worst alternative game costs available to the Interdictor and the Evader, respectively. Initializing these values at the root as $\alpha = -\infty$ and $\beta = +\infty$, we perform the update after selection of each next node $n$ (denoting the accumulated game cost up to node $n$, inclusively, as $\pi$):

- Evader's turn: $\beta \leftarrow \min\Big\{\{\beta\} \cup \{\pi + \widetilde{C}_{nj} + UB(j) \mid j \in \texttt{children}(n)\}\Big\}$,

Figure 5.2: An illustration: alpha–beta pruning




- Interdictor's turn: $\alpha \leftarrow \max\Big\{\{\alpha\} \cup \{\pi + LB(j) \mid j \in \texttt{children}(n)\}\Big\}$.

The intuition behind these is just the definition of $\alpha$ and $\beta$ as the guaranteed costs. Returning to the example in Figure 5.2, if $\beta$ gets updated at node $(A)$ due to some child node $j'$ of $n$, it would mean that the best game cost that the Evader can guarantee, including past and future costs relative to $(A)$, corresponds to the choice of $j'$ at $(A)$. Now, in the subtree rooted at $(A)$, any node implying total game cost larger than this value of $\alpha$ we have just calculated can be safely pruned, because we know that it would be optimal for the Evader not to come to that point at all and just choose $j'$ at $(A)$. We can summarize this pruning rule as follows:

**Alpha–beta pruning concept**

Maintain two running numbers, representing bounds, during the selection process:

- $\alpha$: the worst (minimum) alternative cost achievable by the Interdictor,
- $\beta$: the worst (maximum) alternative cost achievable by the Evader.

**Pruning:** any child node $j$ (of $n$) with $\beta \leq \widehat{\alpha}_j$ or $\widehat{\beta}_j \leq \alpha$, where

- $\widehat{\alpha}_j = \pi_n + LB(j)$ for the Interdictor's turns, and
- $\widehat{\beta}_j = \pi_n + \widetilde{C}_{nj} + UB(j)$ for the Evader's turns.

### 5.2.3 A note on backpropagation and the cost-to-go estimate

The specific rule for backpropagating the cost information from the leaf nodes to the root significantly affected the performance of our algorithm. In many sources (e.g., Browne et al. 2012) the MCTS is presented in a way so that the cost-to-go estimate $\widehat{Q}_j$ represents the mean game costs for all play-outs starting from node $j$ or any of its child nodes. However, we found that this approach required relatively many iterations, because discovery of even very promising child nodes was not immediately affecting cost-to-go estimates of the parent nodes. We instead implemented a potentially less stable method (as pointed out by Browne et al. 2012) that propagates the newly obtained estimates faster, even if the quality of such estimates is imperfect. In particular, we update the cost-to-go estimate using the best child node, according to the current information. For instance, for an evader node $i$ we update its cost estimate $\widehat{Q}_i$ as follows:

$$\widehat{Q}_i \leftarrow \min_{j \in \texttt{children}(i)} \left\{ \widetilde{C}_{ij}(S_i) + \widehat{Q}_j \right\},$$

which potentially updates the recommended path through the game tree after a single iteration of the backpropagation phase.

We also update the bounds of all parent nodes during the backpropagation phase using the information in corresponding interdiction sets $S_j$ and the bounds of the child nodes. The intuition here is that for the Evader parent node $m$ the upper bound can be constructed as the smallest upper bound given the child nodes information and arc traversal costs:

$$UB(m) \leftarrow \min_{j \in \texttt{children}(m)} \{\widetilde{C}_{mj} + UB(j)\}, \quad LB(m) \leftarrow \min_{j \in \texttt{children}(m)} \{\widetilde{C}_{mj} + LB(j)\}. \tag{5.1}$$

Respectively, for the Interdictor parent node $m$, the player can execute its power to

choose the child node in order to maximize the lower bound on the game cost:

$$LB(m) \leftarrow \max_{j \in \texttt{children}(m)} \{LB(j)\}, \quad UB(m) \leftarrow \max_{j \in \texttt{children}(m)} \{UB(j)\}. \tag{5.2}$$

Unfortunately, the respective second bounds, lower bound for the Evader in (5.1) and upper bound for the Interdictor in (5.2), are just the least tight ones among the child nodes, adjusted for turn costs as necessary.

> **Backpropagation implementation**
>
> Both cost-to-go $\widehat{Q}$ and bounds of the parent node are updated assuming the choice of the best child node (for the current player).

Therefore our algorithm is closely related to minimax search (from the perspective of backpropagation), uses the ideas of Monte Carlo simulations to estimate game costs and drive the search through the game tree, and leverages alpha–beta pruning to decrease the search tree size when possible.

## 5.3 Algorithm presentation

The core of the implementation is presented in pseudocode in Algorithm A5.1. It takes a tree with root node `root`, performs $K$ episodes of the four phases described above in the outer **for** loop, and then recommends the best action for the root node given the cost-to-go estimates of the child nodes $\widehat{Q}$. In the beginning of each episode, we re-initialize $\alpha$ and $\beta$ to infinite values and point the current node $n$ at `root`. We keep track of the nodes selected with ordered list $P$ (which initially contains only `root`), and a cumulative cost of the selected path in $\pi$, which corresponds to setting $\pi = \sum_{l=1}^{|P|-1} \widetilde{C}_{P_l,P_{l+1}}$.

**Selection** phase spans lines 2–19. We calculate $\alpha$ and $\beta$ for the parent (current) node and then the respective values $\hat{\alpha}$ and $\hat{\beta}$ for candidate child nodes in lines 5–11, which might allow us to prune some of the branches early with line 10. The rest of the process

assumes `children`$(n)$ is free from obviously suboptimal nodes (given the bounds information available at the start of the episode). We then either pick a child node randomly (in a fashion of $\varepsilon$-greedy selection), or utilize the information on the cost-to-go estimates and the number of visits to particular nodes as discussed in Section 5.2.1. We keep track of these choices by updating the selected path $P$ along with its respective cost, and updating the current node $n$ to always point to the last selected node.

**Expansion** phase is given by lines 20–39. This code creates all the child nodes accessible from the selected one. The bounds are calculated by separate procedures presented in Algorithms A5.3 and A5.4. Both use the same scheme: First, an auxiliary DSPI instance is built that incorporates interdiction decisions corresponding to the game tree node into a new instance's arc costs (assuming zero interdiction increments). Then, a bound for this new instance is calculated using the dynamic programming (polynomial-time) algorithms presented by Sefair and Smith (2016). The key idea behind these procedures is restricting the players to make the instance easier to solve, while ensuring that the optimal objective will constitute a valid bound for the original instance. To obtain the lower bound, the Interdictor is restricted by assuming that its attacks expire after the next Evader's turn (therefore, arc costs are reset to their original state). The expiration of these attacks implies that we need not keep track of the specific interdiction decisions $S$ (from an exponential-size collection of all subsets of the arcs), but just the residual budget. We rely on the corresponding polynomial-time algorithm developed in the aforementioned paper and denoted `DP-DSPI-EXP` here. For the upper bound, we follow the idea to restrict the Evader by removing arcs from the graph until it becomes a DAG. The removal of these arcs allows us to use the polynomial-time algorithm (which we denote `DP-DSPI-DAG`) developed by Sefair and Smith to solve the resulting auxiliary instance. The expansion phase results in a list of newly created nodes $E$ with the correct bounds, but so far undefined cost-to-go estimates $\widehat{Q}$.

**Roll-outs** procedure is called in line 40, which is formally described in Algorithm A5.2. This procedure plays out a game with uniformly random turns of both players, keeping track of the changing state of the game during the play-out in temporary variables, including cur-

rent Evader's position $p$, current budget $b$, interdiction decisions $S$ incorporated into variable $\widetilde{C}$, and turn $\tau$. Note that we also discount roll-out costs with factor $\gamma$.

Finally, note that because the algorithm is focused on a single episode, it is possible to set up a competitive play between the MCTS agent and another algorithm or even a human player. We would just need to substitute `MCTS-next-move` in Line 4 of Algorithm A5.5 to a call for another function or reading the user's input.

## 5.4 On the algorithm correctness

The proposed algorithm obviously does not guarantee optimality, but it does possess some asymptotic properties stemming from the Monte Carlo Tree Search logic with Upper Confidence Bound and $\varepsilon$-greedy selection strategy. This section presents a brief note concerning correctness of the algorithm. First, we will show that the tree generated by Algorithm A5.1 is finite even as $K \to \infty$, in absence of zero-cost cycles. Moreover, informally speaking, any game tree produced by the algorithm can be further expanded to comprise a part of the exact minimax game tree containing an optimum. The implication is that Algorithm A5.1 finds an optimal solution with probability one as $K \to \infty$.

**Proposition 5.1**

If $G$ does not contain cycles of zero cost, the size of the tree produced by Algorithm A5.1 (`MCTS-next-move`) is bounded from above at any episode $k \leq K$, and this bound does not depend on parameter $K$ (maximum number of iterations).

*Proof.* First, observe that in a DAG, the algorithm produces a finite tree. Indeed, there are at most $m = |\mathcal{A}| - 1$ turns for the Evader, since every turn implies visiting a node, and no node is used twice in a DAG. This gives at most $O(m!)$ different paths, assuming the nodes can appear in any order in a path (which is already too permissive for a DAG). The Interdictor's turn effectively multiplies the number of nodes in the subtree by some expression depending on $b$ and $m$, so, the game tree size is at most $m!f(b, m)$. For example,

**Algorithm A5.1.** MCTS-next-move

**Input:** `root`; algorithm parameters: $K$, $C_p$, $\varepsilon$, $\gamma$
**Output:** Recommended action. **Side effect:** search tree is modified.

**for** k=1,...,K **do** *// Repeat while the computational budget is not exhausted*
  $n \leftarrow$ `root`, $P \leftarrow$ (`root`), $\pi \leftarrow 0$ *// Current node, selected path, and its cumulative cost*
  $\alpha \leftarrow -\infty$, $\beta \leftarrow +\infty$ *// Max (min) cost achievable for the Evader (respectively, Interdictor)*
  **while** `actions`$(n) = \varnothing$ and `children`$(n) \neq \varnothing$ **do**
    **if** $\tau(n)$ = Evader: $\beta \leftarrow \min\Big(\{\beta\} \cup \{\pi + \widetilde{C}_{nj} + UB(j) \mid j \in$ `children`$(n)\}\Big)$
    **else**: $\alpha \leftarrow \max\Big(\{\alpha\} \cup \{\pi + LB(j) \mid j \in$ `children`$(n)\}\Big)$
    **for** $j \in$ `children`$(n)$ **do**
      $\widehat{\alpha} \leftarrow \max\{\alpha, \pi + \widetilde{C}_{nj} + LB(j)\}$
      $\widehat{\beta} \leftarrow \min\{\beta, \pi + \widetilde{C}_{nj} + UB(j)\}$
      **if** $\widehat{\beta} < \widehat{\alpha}$: **prune** node $j$.
    **end for**
    **with probability** $\varepsilon$
      $j \leftarrow$ random node from `children`$(n)$
    **otherwise**
      **if** $\tau(n)$ = Evader: $j \leftarrow$ random from $\operatorname{argmax}\Big\{\widetilde{C}_{ni} + \widehat{Q}_i + 2C_p\sqrt{2\log N(n)/N(i)} \mid i \in$ `children`$(n)\Big\}$
      **else** $j \leftarrow$ random from $\operatorname{argmax}\Big\{(-1)(\widetilde{C}_{ni} + \widehat{Q}_i) + 2C_p\sqrt{2\log N(n)/N(i)} \mid i \in$ `children`$(n)\Big\}$
    **end**
    $\pi \leftarrow \pi + \widetilde{C}_{nj}$, $P \leftarrow P \cup \{j\}$, $n \leftarrow j$
  **end while**
  $E \leftarrow \varnothing$ *// A set of nodes to roll-out from*
  **while** p$(n) \neq t$ and `actions`$(n) \neq \varnothing$ **do**
    $j \leftarrow$ **create node** for a random action among `actions`$(n)$
    $N_j \leftarrow 1$; set p$(j)$, $b_j$, and $\widetilde{C}_{nj}$ according to the selected action.
    **remove** the action corresponding to $j$ from `actions`$(n)$.
    **if** ($\tau(n)$ = Evader) **or** ($\tau(n)$ = Interdictor **and** $j$ is not *pass*) **then**
      $\tau(j) \leftarrow$ "Interdictor"
      **if** $b_n > 0$ **then**
        `actions`$(j) \leftarrow$ {"pass"} $\cup$ {"interdict (p$(j), m)$" $\mid m \in$ FS(p$(j)$) and (p$(j), m) \notin S_j$}
      **else**
        `actions`$(j) \leftarrow$ {"pass"}
      **end if**
    **else**
      $\tau(j) \leftarrow$ "Evader"
      `actions`$(j) \leftarrow \{m \mid m \in \mathrm{FS}(j)\}$
    **end if**
    $LB(j) \leftarrow$ `MCTS-LB`$(j,$ instance$)$, $UB(j) \leftarrow$ `MCTS-UB`$(j,$ instance$)$
    $E \leftarrow E \cup \{j\}$
  **end while**
  **if** $E = \varnothing$: $E \leftarrow \{n\}$
  **for** $m \in E$: $\widehat{Q}_m \leftarrow$ `MCTS-roll-out`$(m)$
  **while** $P \neq \varnothing$ **do**
    $m \leftarrow$ pick and remove the last element from $P$
    **if** $\tau(m)$ = Evader **then**
      $\widehat{Q}_m \leftarrow \min\{\widetilde{C}_{mj} + \widehat{Q}_j \mid j \in$ `children`$(m)\}$
      $UB(m) \leftarrow \min\{\widetilde{C}_{mj} + UB(j) \mid j \in$ `children`$(m)\}$
      $LB(m) \leftarrow \min\{\widetilde{C}_{mj} + LB(j) \mid j \in$ `children`$(m)\}$
    **else**
      $\widehat{Q}_m \leftarrow \max\{\widehat{Q}_m \mid j \in$ `children`$(m)\}$
      $LB(m) \leftarrow \max\{LB(m) \mid j \in$ `children`$(m)\}$
      $UB(m) \leftarrow \max\{UB(m) \mid j \in$ `children`$(m)\}$
    **end if**
  **end while**
**end for**
**if** $\tau$(`root`) = Evader: **return** random from $\operatorname{argmin}\{\widetilde{C}_{\texttt{root},j} + \widehat{Q}_j \mid j \in$ `children`(`root`)$\}$
**else**: **return** random from $\operatorname{argmax}\{\widehat{Q}_j \mid j \in$ `children`(`root`)$\}$

**Algorithm A5.2.** MCTS-roll-out

**Input:** node $n$, assuming budget $b_n$ and position $\mathrm{p}(n)$, discount factor $\gamma$
**Output:** Discounted roll-out cost.

$\Delta \leftarrow 0$ *// Total (discounted) roll-out cost*
$r \leftarrow 0$ *// Roll-out step number for discounting*
$p \leftarrow \mathrm{p}(n), b \leftarrow b_n, \tau \leftarrow \tau(n)$ *// Current Evader's position, Interdictor's budget, and turn.*
$\widetilde{S} \leftarrow \widetilde{S}_n$ *// Current arcs interdiction state*
**while** $p \neq t$ **do**
  **if** $\tau$ is "Evader" **then**
    $p' \leftarrow \mathrm{random}(\mathrm{FS}(p))$
    $\Delta \leftarrow \Delta + \gamma^r \widetilde{c}_{p,p'}(S)$
    $p \leftarrow p'$
    $\tau \leftarrow$ "Interdictor"
  **else** *// It is an Interdictor's turn*
    $a \leftarrow \mathrm{random}\Big(\{\text{"pass"}\} \cup \{j \mid j \in \mathrm{FS}(p) \text{ and } (p,j) \text{ is not yet interdicted}\}\Big)$
    **if** $a$ is "pass" **or** $b = 0$ **then**
      $\tau \leftarrow$ "Evader"
    **else**
      $S \leftarrow S \cup (p, a)$
      $b \leftarrow b - 1$
    **end if**
  **end if**
  $r \leftarrow r + 1$
**end while**
**return** $\Delta$

one can claim that $f(b, m) \leq (2^b b!)^m$, because for every Evader's position there are at most $2^b$ possible interdiction actions, each of which generate at most $b!$ distinct virtual turns (if we do not employ some technique to alleviate this symmetry). Thus, there is a finite bound on the tree size that does not depend on $K$.

However, in presence of cycles in $G$ the number of turns in the game is not necessarily finite, despite the finite number of distinct *game states*, as they can repeat in the course of the game. (Note that negative cycles are not possible, because both $c_{ij} \geq 0$ and $d_{ij} \geq 0$ by assumption.) Therefore, it is not immediately obvious if the tree produced by the proposed MCTS procedure will be finite. The answer is clearly "not necessarily" if there is at least one zero-cost cycle. Note that if the Interdictor does not spend its budget, infinitely many nodes can be created for the same game state, none of which can be pruned, if the first one is not pruned. Let us rather consider the simplest case with a cycle having strictly positive cost. Consider graph $G$ presented in Figure 5.3a. In the following, we denote nodes of $G$ as circled numbers, like ①, to avoid confusion with the game tree nodes, which we will denote with capital Latin letters in this section. A simple cycle is created by arcs ①–② and

**Algorithm A5.3.** MCTS-UB

**Input:** Search tree node $n$, instance data; parameter: `NTRIALS`
**Output:** Upper bound on the DSPI objective (starting from the current state)

**for** $i = 1, \ldots, |\mathcal{N}|$ **do**
  **for** $j = 1, \ldots, |\mathcal{N}|$ **do**
    **if** $(i, j)$ is interdicted as per $n$ **then**
      $c'_{i,j} \leftarrow \widetilde{c}_{i,j}(S_n)$
      $d'_{i,j} \leftarrow 0$
    **else**
      $c'_{i,j} \leftarrow c_{i,j}$
      $d'_{i,j} \leftarrow d_{i,j}$
    **end if**
  **end for**
**end for**
$v^*(b_n, \mathrm{p}(n)) \leftarrow \infty$
**for** $k = 1, \ldots,$ `NTRIALS` **do**
  $G' \leftarrow$ `copy`$(G)$, assuming costs $\{c'_{ij} : (i,j) \in \mathcal{A}\}$ and interdiction increments $\{d_{ij} : (i,j) \in \mathcal{A}\}$
  `mark`$(1) \leftarrow$ `True`, and `mark`$(j) \leftarrow$ `False` for all $j = 2, \ldots, |\mathcal{N}|$.
  $P \leftarrow \{1\}$
  **while** $P \neq \varnothing$ **do**
    $i \leftarrow$ last element of $P$
    **for** $j \in \mathrm{FS}_{G'}(i)$ **do**
      **if** $j \in P$: **remove edge** $(i, j)$ from $G'$
    **end for**
    **let** $A \leftarrow \{j \mid j \in \mathrm{FS}_{G'}(i)\}$ and **not** `mark`$(j)$
    **if** $A \neq \varnothing$ **then**
      $j \leftarrow$ random node from $A$
      `mark`$(j) =$ `True`
      $P \leftarrow P \cup \{j\}$
    **else**
      $P \leftarrow P \setminus \{i\}$
    **end if**
  **end while**
  $u^* \leftarrow$ `DP-DSPI-DAG`$(G', b_n)$
  **if** $u^*(b_n, \mathrm{p}(n)) < v^*(b_n, \mathrm{p}(n))$ **then**
    $v^* \leftarrow u^*$
  **end if**
**end for**
**if** Interdictor's turn at $n$ **then**
  **return** $v^*(b, \mathrm{p}(n))$
**else**
  **return** $\min\{\widetilde{c}_{\mathrm{p}(n),j}(S_n) + v^*(b, j) \mid j \in \mathrm{FS}(n)\}$
**end if**

**Algorithm A5.4.** MCTS-LB

**Input:** Search tree node $n$, instance data
**Output:** Lower bound on the DSPI objective (starting from the current state)

**for** $i = 1, \ldots, |\mathcal{N}|$ **do**
  **for** $j = 1, \ldots, |\mathcal{N}|$ **do**
    **if** $(i, j)$ is interdicted as per $n$ **then**
      $c'_{i,j} \leftarrow \widetilde{c}_{i,j}(S_n)$
      $d'_{i,j} \leftarrow 0$
    **else**
      $c'_{i,j} \leftarrow c_{i,j}$
      $d'_{i,j} \leftarrow d_{i,j}$
    **end if**
  **end for**
**end for**
$G' \leftarrow$ `copy`$(G)$, assuming costs $\{c'_{ij} : (i,j) \in \mathcal{A}\}$ and interdiction increments $\{d_{ij} : (i,j) \in \mathcal{A}\}$
$v^* \leftarrow$ `DP-DSPI-EXP`$(G', b_n)$
**if** Interdictor's turn at $n$ **then**
  **return** $v^*(b_n, \mathrm{p}(n))$
**else**
  **return** $\min\{\widetilde{c}_{\mathrm{p}(n),j}(S_n) + v^*(b, j) \mid j \in \mathrm{FS}(n)\}$
**end if**

**Algorithm A5.5.** MCTS-play-instance

**Input:** a DSPI instance
**Output:** Play-out score

$p \leftarrow 1, \Delta \leftarrow 0$
Create a game tree with root node `root`, initialize $\widetilde{c}$ and $b_{\texttt{root}}$ accordingly.
**while** $p \neq t$ **do**
  $a \leftarrow$ `MCTS-next-move`(`root`), $m \leftarrow$ search tree node corresponding to $a$
  $\Delta \leftarrow \Delta + \widetilde{C}_{nm}$
  $p \leftarrow \mathrm{p}(m)$
  `root` $\leftarrow m$
**end while**
**return** $\Delta$

②–①, having costs $c_{12} > 0$ and $c_{21} > 0$, respectively. Observe that the logic concerning the Interdictor's turns described above is still applicable here, and the tree size will be in class of $O(h(\cdot)f(b,m))$ for some function $h$. Thus, we can restrict our attention to the Evader's turns only, with the goal of showing that function $h(\cdot)$ does not depend on $K$. To simplify the exposition, we will first consider the game tree as if the Interdictor were not present at all. In this case, the alpha–beta pruning mechanism will be dealing with the worst (maximum) alternative cost achievable by the Evader ($\beta$), and the minimum alternative cost achievable by the Interdictor ($\alpha$) will reduce to the best upper bound on alternative paths available to the Evader.

Consider a snapshot of a tree after expansion of the first game tree node corresponding to the Evader's position at node ②, depicted in Figure 5.3b. There are two child nodes, one of which corresponds to traversing the cycle (returning to node ①). We show that expansion of this subtree according to the proposed algorithm will never yield unbounded growth of the tree. First, if node $(A)$ is pruned, then the result follows trivially, so let us consider the other case. Expansion of game tree node $(B)$ yields creation of new game tree nodes describing the same possible actions from node ①, as shown in Figure 5.3c. We show that such repeated blocks will be created only a finite number of times (with the bound not depending on $K$) due to the pruning mechanism. Recall that a node is pruned when the corresponding values $\widehat{\alpha}$ and $\widehat{\beta}$ for this node satisfy the condition $\widehat{\beta} \leq \widehat{\alpha}$. This condition implies that another decision was available before this point that would yield a strictly better solution based on our available bounds. At the creation of nodes $(B)$ and $(C)$ we had $\widehat{\beta}_B > \widehat{\alpha}_B$ and $\widehat{\beta}_C > \widehat{\alpha}_C$ by assumption. However, during the selection phase these values are updated as follows:

$$\widehat{\alpha}_B \leftarrow \max\{\alpha, \pi + c_{21} + \mathrm{LB}_B\}, \text{ and } \widehat{\beta}_B \leftarrow \min\{\beta, \pi + c_{21} + \mathrm{UB}_B\},$$

where $\pi$ is cumulative cost corresponding to the path up to node ② in the graph (before traversing the cycle for the first time), and $\alpha$ and $\beta$ are carried over from the previous

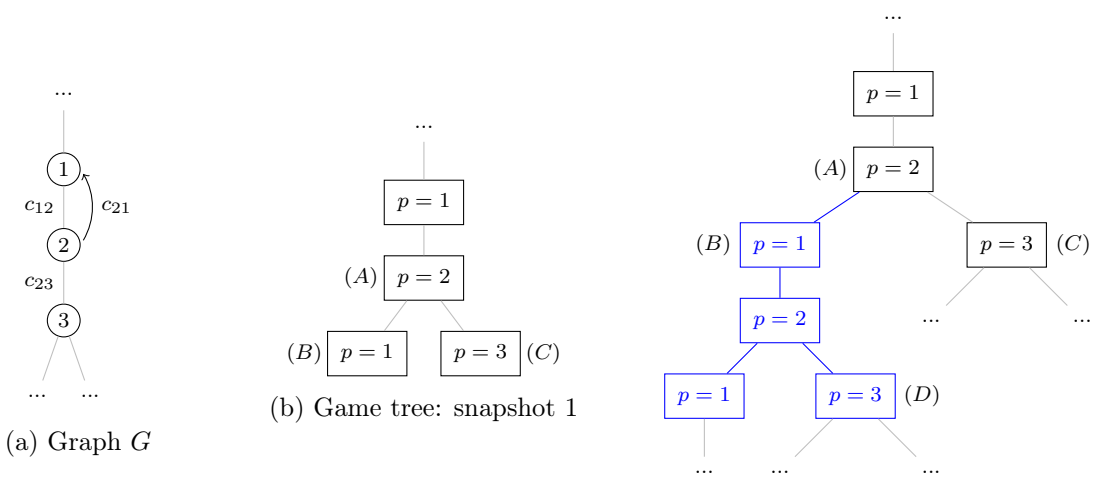
game tree nodes and can be disregarded further without loss of generality. Lower bounds are maintained corresponding to the respective child nodes by backpropagation passes, so that at the beginning of every selection phase, the bounds information for all child nodes is correct and consistent with the respective parent nodes. Observe that after creating at most $M = \lceil((\mathrm{UB}^0_C + c_{23}) - (\mathrm{LB}^0_B + c_{21}))/c_{21}\rceil$ repeated blocks of game tree nodes corresponding to the cycle (Evader's position ①, ②, and ③ in our example), the whole subtree rooted at $(B)$ will be pruned (where $\mathrm{UB}^k_C$ denotes the upper bound at node $(C)$ calculated for the $(k+1)$-th pass through this node). This number $M$ is positive, since we assume node $(B)$ was not pruned in favor of $(A)$. With each such block creation, the lower bound $\mathrm{LB}_B$ increases by at least $c_{21}$. Therefore, after creation of $M$ repeated blocks we would have:

$$\begin{aligned}
&\widehat{\beta}_B \leq \widehat{\beta}_A \leq \pi + c_{23} + \mathrm{UB}^0_C, \\
&\widehat{\alpha}_B \geq \pi + c_{21} + \mathrm{LB}^M_B \geq \pi + c_{21} + (\mathrm{LB}^0_B + Mc_{21}) = \pi + c_{23} + \mathrm{UB}^0_C \geq \widehat{\beta}_B,
\end{aligned} \tag{5.3}$$

which satisfies the pruning condition of $\widehat{\beta}_B \leq \widehat{\alpha}_A$. Informally, if the subtree with repeated actions grows beyond a certain limit (determined by the cost of the arc creating the cycle) the algorithm will prune it in favor of alternative paths through the tree, e.g., node $(C)$ against node $(B)$ in this example. Note that pruning does not eliminate *all* redundant nodes. For example, several repeated blocks might be created before we hit condition (5.3), and due to random roll-outs node $(D)$ could be deemed better than $(C)$. So, a good solution might be found later in a subtree rooted at $(D)$, but we would still have node $(C)$ in the game tree, representing exactly the same Evader's position at ③. (We will show in the next proposition that given enough time $(D)$ would be pruned in favor of $(C)$.)

Therefore, a zero-cost cycle might create a large, but finite multiplicative increase in the game tree size, with the bound not depending on the number of episodes $K$. If the cycle contains more than two arcs, then we can modify this logic to accommodate for a *path cost* instead of *arc* cost $c_{21}$, and introduction of the Interdictor would yield another finite multiplicative increase in the game tree size bound, not depending on $K$ (as discussed

Figure 5.3: An illustration: a case with a cycle




(a) Graph $G$

(b) Game tree: snapshot 1

(c) Game tree: snapshot 2

above). The intuition that infinite paths along cycles of nonzero costs are ultimately pruned in favor of finite alternative paths still applies to all complications of the situation depicted in Figure 5.3. □

Moreover, we can make another step and claim that if we let the algorithm run long enough, the finite tree we obtain will point us to an optimal sequence of moves.

**Proposition 5.2**

Denoting the cost-to-go estimate at the root game tree node after $k$ iterations as $\widehat{Q}^k_{\texttt{root}}$ and the optimal objective as $q^*$, the `MCTS-next-move` procedure implemented as per Algorithm A5.1 satisfies:

$$\lim_{k\to\infty} \mathbb{P}\{Q^k_{\texttt{root}} = q^*\} = 1.$$

*Proof.* First, note that game tree pruning never cuts off all optimal solutions. Then, observe that if all game tree nodes are expanded, then $Q^k_{\texttt{root}}$ is in fact obtained by a procedure equivalent to the minimax search algorithm. Because we never cut off all optima, the probability of $Q^k_{\texttt{root}} = q^*$ equals at least the probability of expanding all possible game

tree nodes, which is sufficient, but not necessary to obtain an exact optimum. Now, given a specific graph, it follows from Proposition 5.2 that there is only finite number of such possible game tree nodes. Recall that with probability $\varepsilon > 0$ we select a node uniformly at random. Consider a random walk from the root to a terminal node in a tree having a finite number of nodes. Each possible node to expand will be selected with some positive probability bounded from below by some $p_0 > 0$. The probability that the algorithm will sample paths to all nodes is bounded from below by the probability of having the necessary (finite) number of successes (which we will denote $X$) in a Bernoulli scheme with success probability $p_0$. As the target number of successful episodes does not depend on $k$, the probability of having at least $X$ successes approaches 1 as $k$ grows. □

Therefore, with a slight modification (making the procedure stop when the tree cannot be further improved), we see that this is a Las Vegas type of a randomized algorithm (see, e.g., Sedgewick and Wayne 2011): It yields the optimal objective as $\widehat{Q}_{\texttt{root}}$, but with random runtime (note that we cannot strictly claim it is finite due to the randomness we employ). We make two modifications to this procedure to improve its practicality.

First, we turn it into a Monte Carlo type of randomized algorithm by the standard trick of stopping the procedure after the computational budget is exhausted (i.e., setting a finite $K$). This budget guarantees a limited runtime, at the expense of guaranteeing that the algorithm will yield optimal objective estimates. This trade-off is reasonable, because no algorithm can provide a fully polynomial-time approximation scheme for DSPI, unless $\mathcal{P} = \mathcal{NP}$, as shown by Sefair and Smith (2016).

Moreover, by actually making moves (as opposed to starting the selection process from the Evader's position at the source every time) we achieve a significant speed up. Also, the convergence due to the random selection mentioned in the proof of the last claim is slow, and guiding the selection process by cost-to-go estimates reduces the runtime in practice. We illustrate these and other effects related to the proposed algorithm with quantitative illustrations presented in the next section.

## 5.5 Numerical experiments

To investigate the computational efficacy of the proposed algorithm in practice, we performed a series of computational experiments over both randomly generated and predefined instances.

We generated the first category of instances using the Erdős-Rényi-Gilbert model (Gilbert 1959) with varying number of nodes and parameter $p = 0.25$. That is, a completely connected graph was created, and then every edge was erased with probability $q = (1 - p)$. Or, conversely, $N$ isolated nodes were created and every edge out of the possible $N(N - 1)/2$ ones was created independently with probability $p$. Such graphs were not necessarily connected, so we checked that every node was accessible from $s$, and that $t$ was reachable from every other node in the graph. Nodes that did not possess these properties were deleted from the graph. Thus, we started with some specific number of nodes $N$, but provided no guarantee that the generated instance would have exactly $N$ nodes (even though few nodes were deleted due to this check). Costs were generated as uniformly random integer numbers between 1 and 10, and interdiction increments for every arc cost $c_{ij}$ were generated uniformly random from $d_{ij} \in [0, 2c_{ij}]$.

The foregoing approach is only one of many ideas for generating a DSPI instance. Perhaps a more elegant one was proposed by Sefair and Smith (2016), which started from a random directed in-tree having $t$ as the root and proceeded with adding arcs randomly. We do not study probabilistic properties of these graphs (or the corresponding solutions) rigorously here, so we have picked the simpler procedure from the implementation perspective. However, it does give rise to an important question of what a "complicated DSPI instance" actually is. For example, consider graphs having the *small world* property (where there is, informally speaking, a short path between every two nodes, and in particular between $s$ and $t$). Now, imagine the process of increasing the number of nodes and budgets $b$ for such instances. We would expect an increase in the state space of the DSPI game, but many new $s$–$t$ paths created would be longer (in terms of the number of arcs) and more expensive than

the ones that existed before, assuming uniform distribution of the arc costs and interdiction increments. Therefore, many game tree nodes corresponding to such paths could be easily pruned. On the other hand, it might be possible to engineer especially difficult instances, where adding nodes would create different $s$–$t$ paths of comparable costs and interdiction increments, making it hard to remove them from consideration. We leave this topic for future research.

For the second category of pre-defined instances, we used a subset of instances generated in (Sefair and Smith 2016): fifteen 10-node instances and fifteen 20-node instances of varying density, with the former having $b = 1$ and the latter implying $b = 2$. We pre-processed these instances in a similar fashion to the randomly generated ones, to ensure all nodes are connected to $s$ and $t$ as necessary.

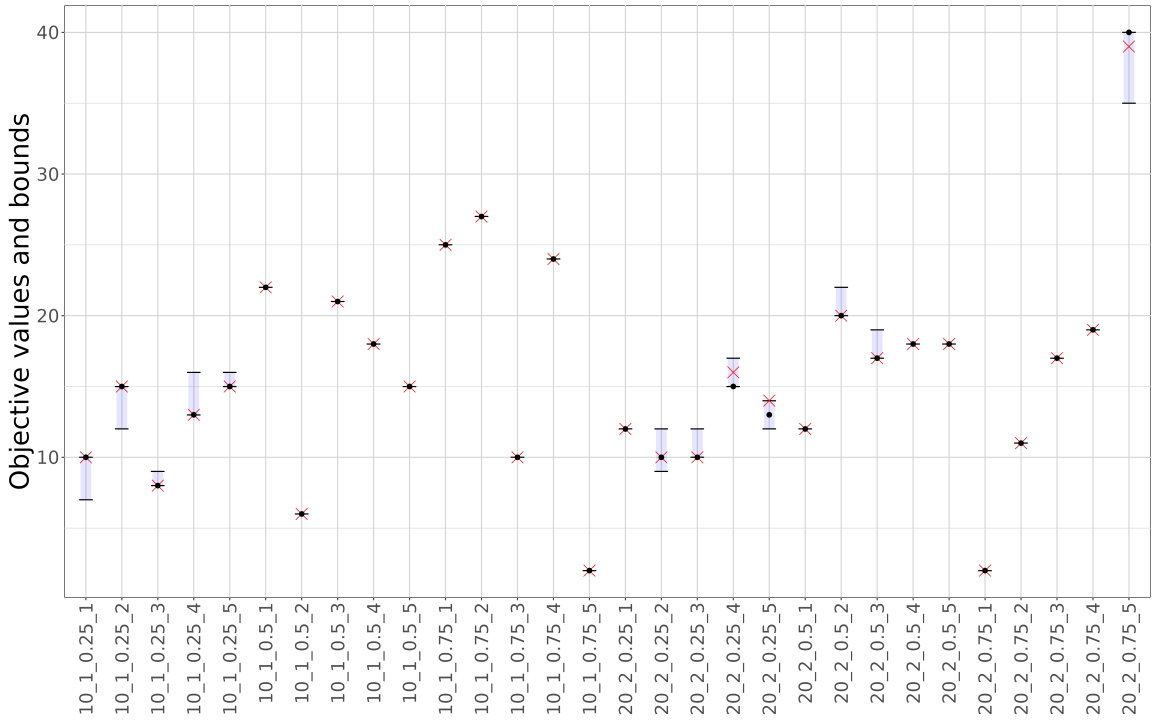
Both the exact dynamic programming approach developed by Sefair and Smith and the MCTS algorithm proposed in this work were implemented from scratch using Julia programming language (Bezanson et al. 2017), the simulations were run on a laptop with a four-core Intel i5 processor (3.1 GHz) and 8 Gb of RAM, with occasional use of the resources available from the Clemson University Palmetto high-performance computing cluster (for longer experiments). The software was exclusively based on GNU/Linux system with `bash` and `GNU make` used to build a computational pipeline and R language ecosystem to produce figures from the generated data (with most of the core components discussed by R Core Team 2021, Wickham 2016, Wickham et al. 2019).

Unless stated otherwise, we assumed $K = 10$ episodes per move, $C_p = 14.0$, $\varepsilon = 0.05$, and $\gamma = 1.0$ (no discounting). These values were chosen after brief preliminary computational experiments.

### 5.5.1 Pre-defined instances and scaling

First, we run the MCTS algorithm against the dataset of pre-defined instances. Figure 5.4 depicts the known optimal value, calculated lower and upper bounds, and the objective obtained by the MCTS play-out for each of the instances. We observe that the

Figure 5.4: MCTS solutions (pre-defined instances)




**Note.** Shaded bars represent the gap between lower and upper bounds. Black dots denote the exact optimum, while crosses represent the objective obtained in a play-out using our MCTS algorithm.

bounds proposed by Sefair and Smith are very tight: in fact, they close the gap immediately for most of the instances. Although their bounds do not reveal the actual optimal policy, they direct our algorithm to find either optimal or near-optimal solutions in just $K = 10$ iterations per move.

However, we note that these instances are easy to solve, and were in fact generated to validate the exact dynamic programming approach proposed for the DSPI. That exact algorithm typically solved each instance in under 10 seconds (under 1 second most of the time), and was actually faster than the proposed heuristic on each instance. However, as one might expect, the exact and heuristic approaches scale in fundamentally different ways. To illustrate this fact, we modified each of the instances discussed above to consider the cases of $b = 1$, 2, and 3. We solved each of these new instances with the exact DP algorithm and using our MCTS procedure, keeping records of the runtime (in seconds) and the objective

values obtained. The results are summarized in Table 5.1.

Table 5.1: Runtimes and objectives: MCTS vs. exact algorithms

| Name | $\lvert\mathcal{N}\rvert$ | $\lvert\mathcal{A}\rvert$ | Runtime, sec. | | | | | | Objective | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $b=1$ | | $b=2$ | | $b=3$ | | $b=1$ | | $b=2$ | | $b=3$ | |
| | | | H | DP | H | DP | H | DP | H | DP | H | DP | H | DP |
| 10_1_0.25_1 | 10 | 61 | 0.08 | 0 | 0.43 | 0.06 | 0.51 | 2.70 | 10 | 10 | 10 | 10 | 10 | 10 |
| 10_1_0.25_2 | 10 | 61 | 0.19 | 0 | 0.24 | 0.07 | 0.77 | 3.47 | 15 | 15 | 19 | 19 | 19 | 19 |
| 10_1_0.25_3 | 10 | 61 | 0.26 | 0 | 0.54 | 0.07 | 0.63 | 3.42 | 8 | 8 | 10 | 10 | 11 | 11 |
| 10_1_0.25_4 | 10 | 61 | 0.29 | 0 | 0.42 | 0.07 | 0.76 | 3.58 | 13 | 13 | 17 | 17 | 22 | 22 |
| 10_1_0.25_5 | 10 | 61 | 0.25 | 0 | 0.41 | 0.08 | 0.93 | 3.65 | 15 | 15 | 19 | 19 | 20 | 20 |
| 10_1_0.5_1 | 10 | 41 | 0.27 | 0 | 0.21 | 0.02 | 0.84 | 1.02 | 22 | 22 | 24 | 24 | 24 | 24 |
| 10_1_0.5_2 | 10 | 41 | 0.08 | 0 | 0.13 | 0.07 | 0.28 | 1.03 | 6 | 6 | 8 | 8 | 8 | 8 |
| 10_1_0.5_3 | 10 | 41 | 0.42 | 0 | 0.18 | 0.03 | 0.59 | 0.79 | 21 | 21 | 28 | 28 | 31 | 31 |
| 10_1_0.5_4 | 10 | 41 | 0.06 | 0 | 0.18 | 0.04 | 0.2 | 0.99 | 18 | 18 | 24 | 24 | 28 | 28 |
| 10_1_0.5_5 | 10 | 41 | 0.06 | 0 | 0.17 | 0.06 | 0.25 | 1.01 | 15 | 15 | 15 | 15 | 15 | 15 |
| 10_1_0.75_1 | 9 | 16 | 0.12 | 0 | 0.16 | 0 | 0.23 | 0.03 | 25 | 25 | 26 | 26 | 35 | 35 |
| 10_1_0.75_2 | 9 | 20 | 0.06 | 0 | 0.11 | 0 | 0.2 | 0.07 | 27 | 27 | 40 | 40 | 44 | 44 |
| 10_1_0.75_3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 10_1_0.75_4 | 8 | 16 | 0.05 | 0 | 0.08 | 0 | 0.1 | 0.04 | 24 | 24 | 26 | 26 | 26 | 26 |
| 10_1_0.75_5 | 9 | 20 | 0.04 | 0 | 0.13 | 0 | 0.22 | 0.07 | 2 | 2 | 2 | 2 | 2 | 2 |
| 20_2_0.25_1 | 20 | 271 | 1.31 | 0.02 | 1.7 | 2.86 | 6.29 | 2004.43 | 11 | 11 | 12 | 12 | 14 | 14 |
| 20_2_0.25_2 | 20 | 271 | 1.27 | 0.01 | 2.89 | 5.08 | 5.6 | 1928.24 | 8 | 8 | 10 | 10 | 12 | 12 |
| 20_2_0.25_3 | 20 | 271 | 0.65 | 0 | 8.3 | 2.72 | 10.57 | 2642.78 | 8 | 8 | 10 | 10 | **14** | **13** |
| 20_2_0.25_4 | 20 | 271 | 4.86 | 0 | 4.56 | 2.7 | 6.08 | 1902.1 | 13 | 13 | 15 | 15 | **16** | **18** |
| 20_2_0.25_5 | 20 | 270 | 9.19 | 0 | 3.55 | 2.73 | 5.66 | 2863.08 | 10 | 10 | **12** | **13** | 14 | 14 |
| 20_2_0.5_1 | 20 | 181 | 0.63 | 0 | 1.3 | 0.92 | 3.99 | 296.59 | 10 | 10 | 12 | 12 | 12 | 12 |
| 20_2_0.5_2 | 20 | 181 | 1.43 | 0 | 5.32 | 1.01 | 2.93 | 337.89 | 18 | 18 | 20 | 20 | 27 | 27 |
| 20_2_0.5_3 | 20 | 181 | 0.65 | 0 | 0.99 | 1.87 | 2.19 | 242.36 | 14 | 14 | 17 | 17 | 20 | 20 |
| 20_2_0.5_4 | 20 | 181 | 1.26 | 0 | 1.54 | 2.3 | 2.22 | 301.79 | 17 | 17 | 18 | 18 | 19 | 19 |
| 20_2_0.5_5 | 20 | 181 | 0.98 | 0 | 2.76 | 1.01 | 3.97 | 347.13 | 15 | 15 | 18 | 18 | 20 | 20 |
| 20_2_0.75_1 | 20 | 91 | 0.17 | 0 | 0.9 | 0.19 | 0.75 | 16.57 | 2 | 2 | 2 | 2 | 2 | 2 |
| 20_2_0.75_2 | 20 | 91 | 0.21 | 0 | 0.62 | 0.34 | 0.68 | 14.78 | 9 | 9 | 11 | 11 | 16 | 16 |
| 20_2_0.75_3 | 20 | 91 | 0.13 | 0 | 0.51 | 0.16 | 1.05 | 15.93 | 17 | 17 | 17 | 17 | 17 | 17 |
| 20_2_0.75_4 | 20 | 91 | 0.45 | 0 | 0.5 | 0.32 | 2.14 | 14.74 | 11 | 11 | 19 | 19 | 19 | 19 |
| 20_2_0.75_5 | 20 | 91 | 1.3 | 0 | 1.97 | 0.2 | 6.14 | 15.96 | 27 | 27 | 40 | 40 | **43** | **47** |

*Note.* MCTS measurements denoted H ("heuristic"), and exact algorithm is denoted DP ("Dynamic Programming"). Instances of different sizes are involved: $\lvert\mathcal{N}\rvert$ nodes, $\lvert\mathcal{A}\rvert$ edges; instance "Name" corresponds to a unique graph topology, varying budget is denoted $b = 1, 2, 3$. Discrepancies between the obtained objective values are highlighted with bold font. Instance names are provided in the format `N_b_D_i` as per Sefair and Smith (2016), where `N` denote number of nodes before the preprocessing, `b` stands for the budget (in the original instance), `D` is the graph density characteristic, and `i` is an instance number.

First, note that for almost all cases the objective values coincide for both methods (except for four larger instances). Therefore, 10 episodes per move was enough for the proposed algorithm to identify an optimal game play for both players most of the time on our dataset. This confirms our intuition that for larger instances, it would be prudent to allow larger computational budgets (number of episodes $K$) to obtain better solutions. Still, surprisingly few iterations are necessary to solve most of these instances to optimality.

Runtimes vary greatly, and exhibit a somewhat expected pattern: For smaller instances ($b = 1$ and 2, first four columns in the "Runtime" section) the MCTS was almost always significantly slower than the exact algorithm, sometimes by orders of magnitude. However, even for $b = 3$ the situation is reversed, as the state space of the dynamic program grows exponentially, while the MCTS limits the game tree size via pruning and strategic move selection. Also as we would expect, these effects grow stronger as we move from 10-node (the top half of the table) instances to 20-node instances (the bottom one).

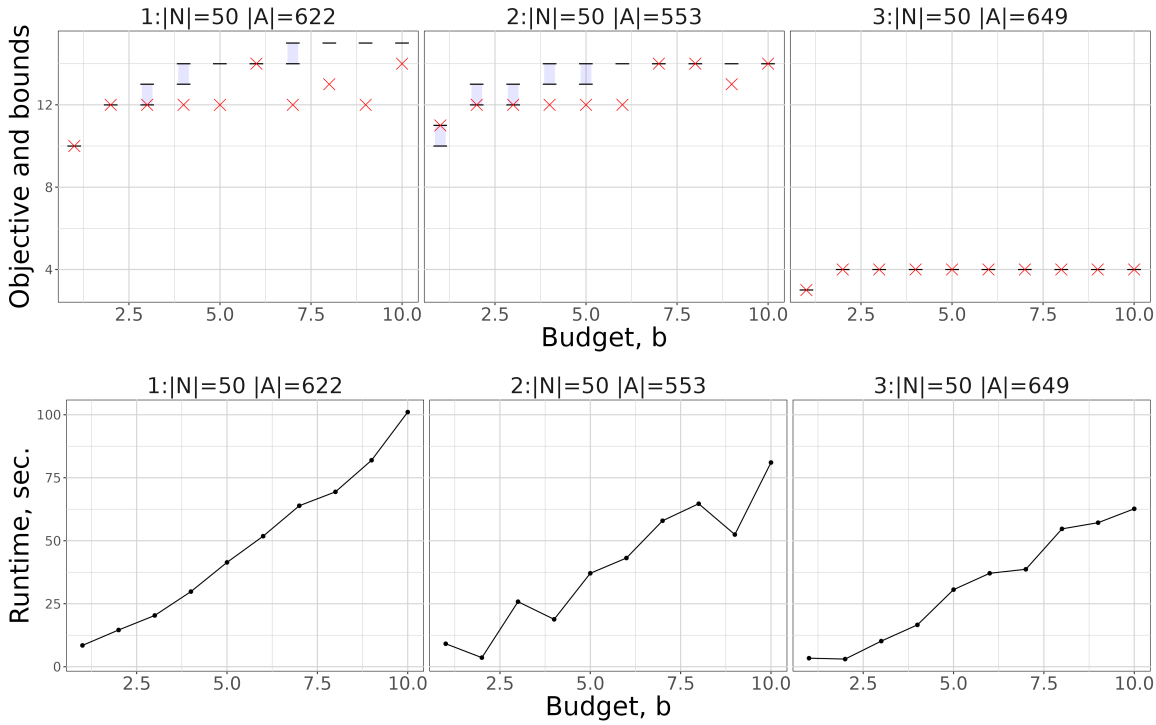
To further illustrate how the MCTS algorithm scales with the instance size, we generated three random fifty-node instances and varied the budget from $b = 1$ to 10. The results are summarized in Figure 5.5. We still see that the proposed approach often finds an actual sequence of turns leading to the optimal (or near-optimal) objective. The runtimes for these larger instances are under 100 seconds, which indicates that the MCTS-based approach is scalable for instances of this size, as opposed to the exact algorithm. The figure also suggests that for larger instances, we could trade off additional runtime for better solutions by increasing the maximum number of iterations $K$, as the algorithm sometimes yielded play-outs outside of the calculated bounds. Note that obtaining objective values outside of the bounds is technically possible, because the algorithm chooses the actions based on our cost-to-go estimates. The bounds are not tight enough to cut off all suboptimal solutions.

Also, note that the runtime does not grow as fast with the increase of $b$ from 1 to 10 as it would be the case with the exact algorithm. The growth of the tree is evidently highly dependent on the instance structure and is connected to the question of the instance difficulty we briefly mentioned above.

### 5.5.2 MCTS convergence profile

Note that a naive implementation of our MCTS algorithm could ultimately construct a full minimax tree, which would inevitably find an optimal strategy given enough time and space. (In this case, there would be little value in the Monte Carlo approach itself.) Accordingly, this section explores the ability of the MCTS heuristic to limit the exploration

Figure 5.5: MCTS solutions and runtimes (randomly generated instances)




**Note.** Shaded bars in the top panel represent the gap between lower and upper bounds, while crosses indicate the objective obtained in a play-out using our MCTS algorithm. Runtimes in the bottom panel correspond to the instances solved for the top one. Runtimes are given per instance. Also, instances with different budget are treated as completely unrelated (no information is transferred between the MCTS runs besides the instance description).

of the game tree while still obtaining high-quality solutions. To accomplish this analysis, we constructed a *convergence profile* for solving a DSPI instance as follows.

We sought to generate one random instance having 50 nodes, $b = 3$, and an optimality gap of zero using the bounds developed by Sefair and Smith (2016). The zero gap ensured that we would know the true optimal objective without having to execute the exact DP algorithm. (The generation scheme was executed as before, but we repeated the procedure until an instance satisfying all our requirements was identified.) After obtaining the DSPI instance, we executed our heuristic as before, but did not commit any moves by either player. Instead, we performed learning episodes from the same root node, recording how the game tree was developed. The results are summarized in Figure 5.6. The top two panels show the dynamics of the tree size, where dots represent the number of nodes in the tree after each learning episode and the bars show the number of node pruning or adding operations. (Note that one pruning might cut more than one node, but one adding always adds a single node.) The vertical dashed line across all panels represent the moment when the cost estimate $\widehat{Q}$ at the root node reached the true optimal value of 5.

We see that tree dynamics changes over time. First, there is a period when the tree grows very actively. Pruning happens frequently (see the panel with bars), but the net effect on the number of nodes is significantly positive. Then, as we collect more information, the pruning starts to dominate and the game tree starts to shrink, until its size hits a plateau. In fact, we observe several plateaus interrupted by the moments when valuable node discovery allows the algorithm to prune several game tree nodes. The tree size stops changing frequently after approximately 300 episodes, and the number of nodes completely stabilizes after the discovery of the true optimum approximately at the 500th episode.

The nature of backpropagation with the *minimum* or *maximum* function makes the process unstable in terms of cost estimates: the cost-to-go at the root node (the bottom panel) varies greatly, from 0 to 1000, until around 250th episode. Moreover, recommended actions from the root node ("Best action" panel) also change significantly, until they stabilize on jumping between several optimal moves after about the 500th episode. Note that this

behavior immediately suggests a research line for improving our simple MCTS algorithm. The concept of virtual turns for the Interdictor introduces an unnecessary symmetry to the problem: The same interdiction set can be considered by the algorithm as several distinct turns, if the arcs are interdicted in a different order. This symmetry is one of the reasons why the algorithm recommends multiple different actions at the root node after the discovery of the optimum. (Recall that the first turn is always made by the Interdictor.) This effect also dilutes the valuable information across several branches. There are several ways to alleviate this problem, for example, to artificially merge such turns into one using some sort of hash table, or just pre-ordering the forward stars for the original graph to always consider interdictions in some specific order. On top of this adjustment, one might want to record the information (bounds and cost estimates) into *several* nodes at once, as we perform roll-outs. This is along the lines of the *All-Moves-As-First* idea, which is discussed, along with many other ones, in the survey by Browne et al. (2012).

Thus, for the prior example our proposed algorithm intensively grows the tree, then prunes it effectively, and rapidly obtains a true optimal policy. We next explore a different regime for the tree construction, leveraging the role parameter $\varepsilon$ plays in the efficacy of our algorithm. In particular, as $\varepsilon$ grows, and the probability of selecting a node at random increases, the cost-to-go estimates tend to become less accurate. Figure 5.7 depicts the performance of our approach when we increase $\varepsilon$ from 0.05 to 0.5 on the same instance as discussed before. The maximum tree size dropped, and the process became more uniform. Adding and pruning nodes happened longer: The "added / pruned" bars spread across almost all 1,000 episodes as compared to approximately the first 300 episodes only in the previous experiment. The optimum was discovered later (beyond the 800th episode), and, interestingly, the algorithm did not recognize several of the optimal moves as such. (Note that there are different numbers of dashed lines, representing alternative optimal actions for the first turn, in the "Best action" panels for $\varepsilon = 0.5$ and $\varepsilon = 0.05$.) This behavior does not prevent the Interdictor from playing optimally: in both these runs the cost-to-go estimate stabilized on the true optimal value of 5.

Finally, to provide an extreme case, we made the selection process completely random by setting $\varepsilon = 1.0$ (see Figure 5.8). We do observe the tree growing and somewhat stabilizing, but the algorithm never came close to the optimal value within 1000 episodes, and the process did not converge, as it did after approximately the 500th episode for $\varepsilon = 0.5$ or from the very beginning with $\varepsilon = 0.05$. Therefore, deliberately excluding the mechanism of heuristic selection (and relying exclusively on bounds and pruning to find an optimum) significantly deteriorated the performance of our algorithm.

To illustrate the fact that the difference in these convergence profiles is not mainly due to the randomization employed in the algorithm, we performed three independent runs for each set of parameters. The corresponding tree size dynamics are presented in Figure 5.9. We see that while the trajectories are distinct, the algorithm behaves in different modes depending on the value of $\varepsilon$. There is a clear growth and pruning in the first one, but a steadier and more uniform growth pattern in the next two. The pattern of the cost-to-go estimates also persisted across all three runs.

The previous analysis illustrates the benefits of the Monte Carlo component, but leaves us with one important question. Note that even in the best case, our estimate $\widehat{Q}_{\texttt{root}}$ stabilized around the optimum after approximately 250 iterations. However, in Table 5.1 and Figure 5.4 we have seen that the MCTS with just 10 iterations per move was able to find the optimum quite often. Also, the 100-iteration version was able to quickly solve the instance described in Figures 5.6–5.8. Clearly, total number of iterations per tree was beyond 100; however, making turns and updating the root node itself introduces another benefit of focusing the search process. We illustrate this effect in the following experiment.

### 5.5.3 The value of a play-out

Note that in convergence profiles illustrated in Figures 5.6 and 5.7 optimal actions (corresponding to the dashed horizontal lines after the true optimal solution was found) appeared very early among the recommendations. Actually committing to one of these moves would have caused the algorithm to focus the analysis on a more promising alternative,

Figure 5.6: MCTS convergence profile: $\varepsilon = 0.05$

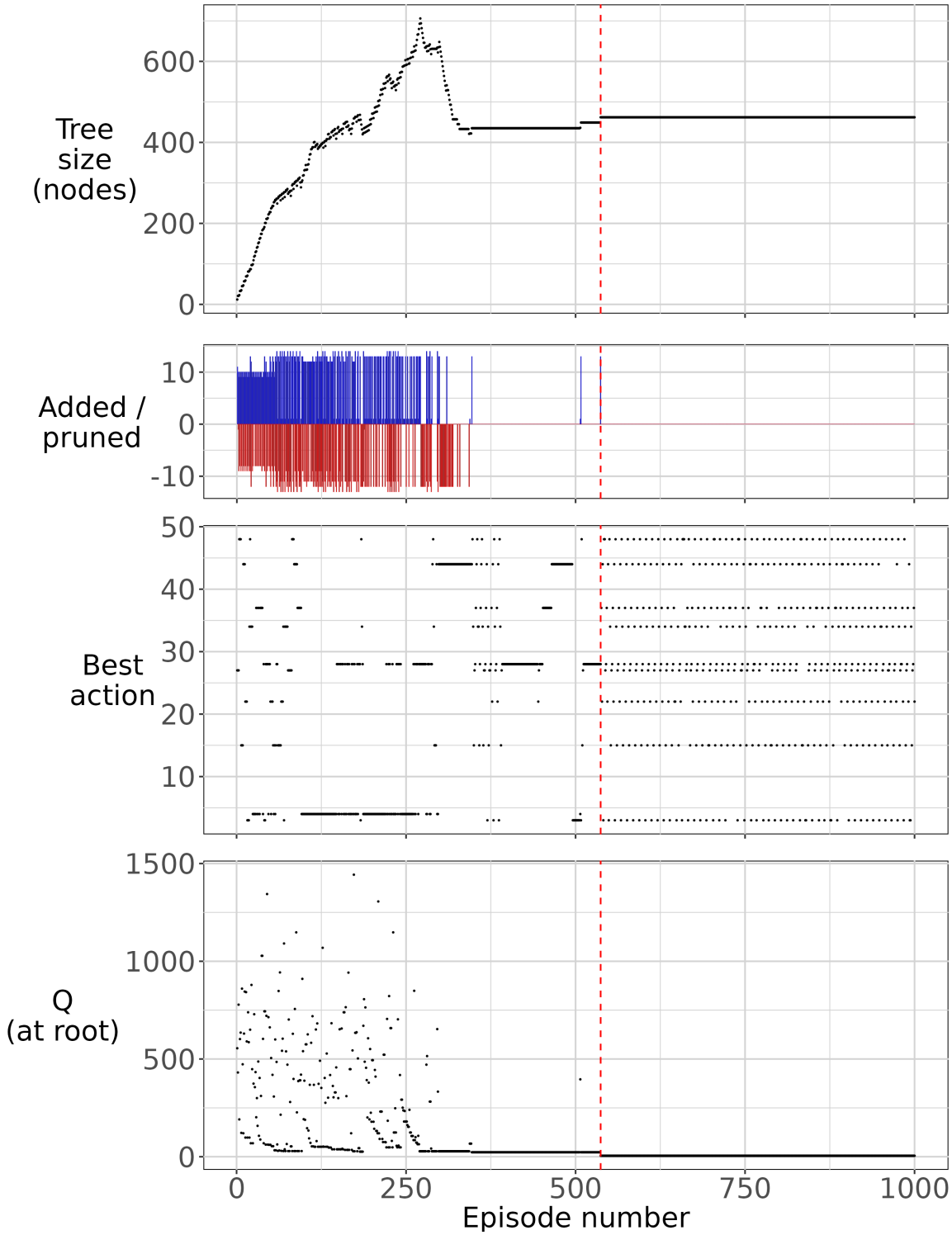


Figure 5.7: MCTS convergence profile: $\varepsilon = 0.5$

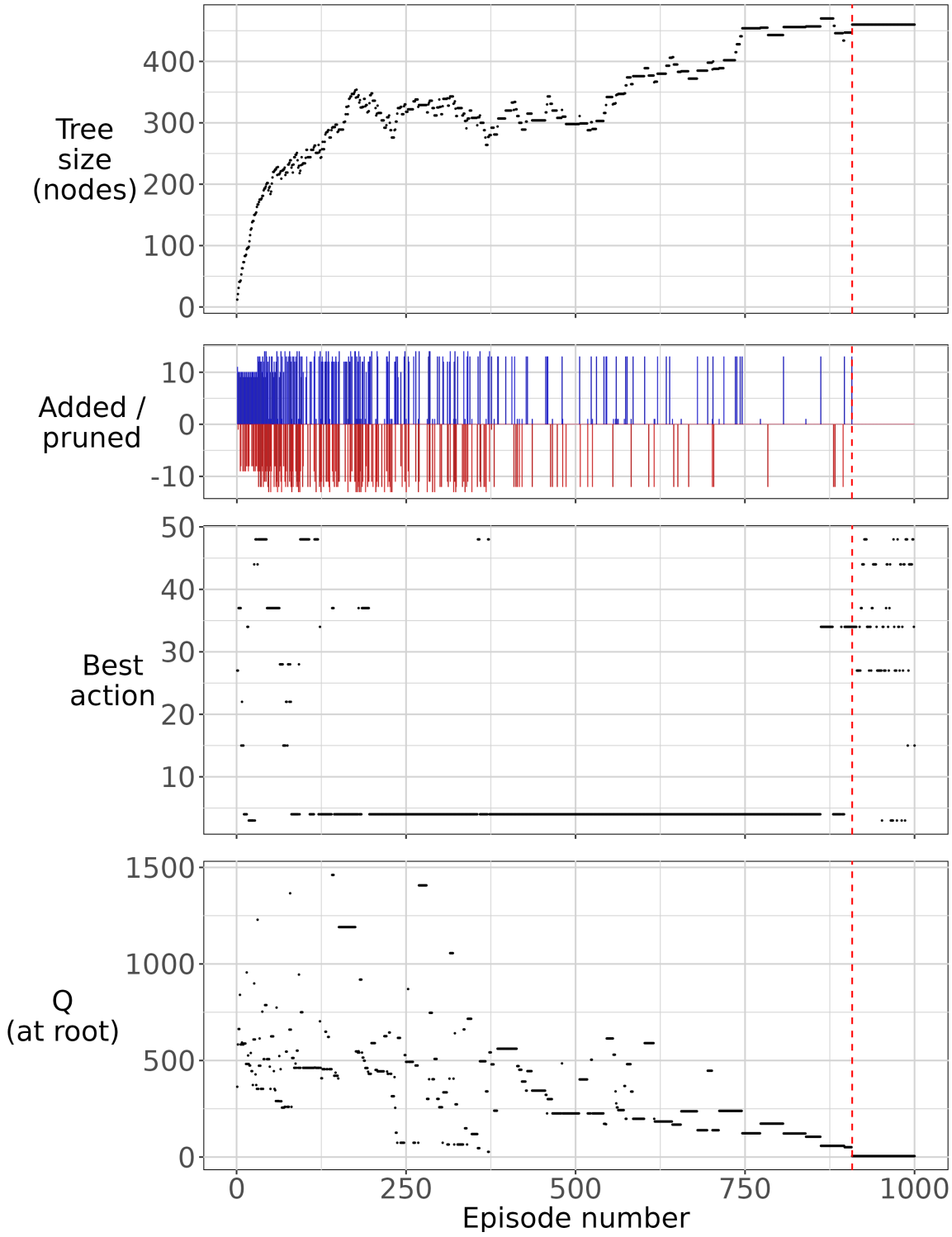


Figure 5.8: MCTS convergence profile: $\varepsilon = 1.0$

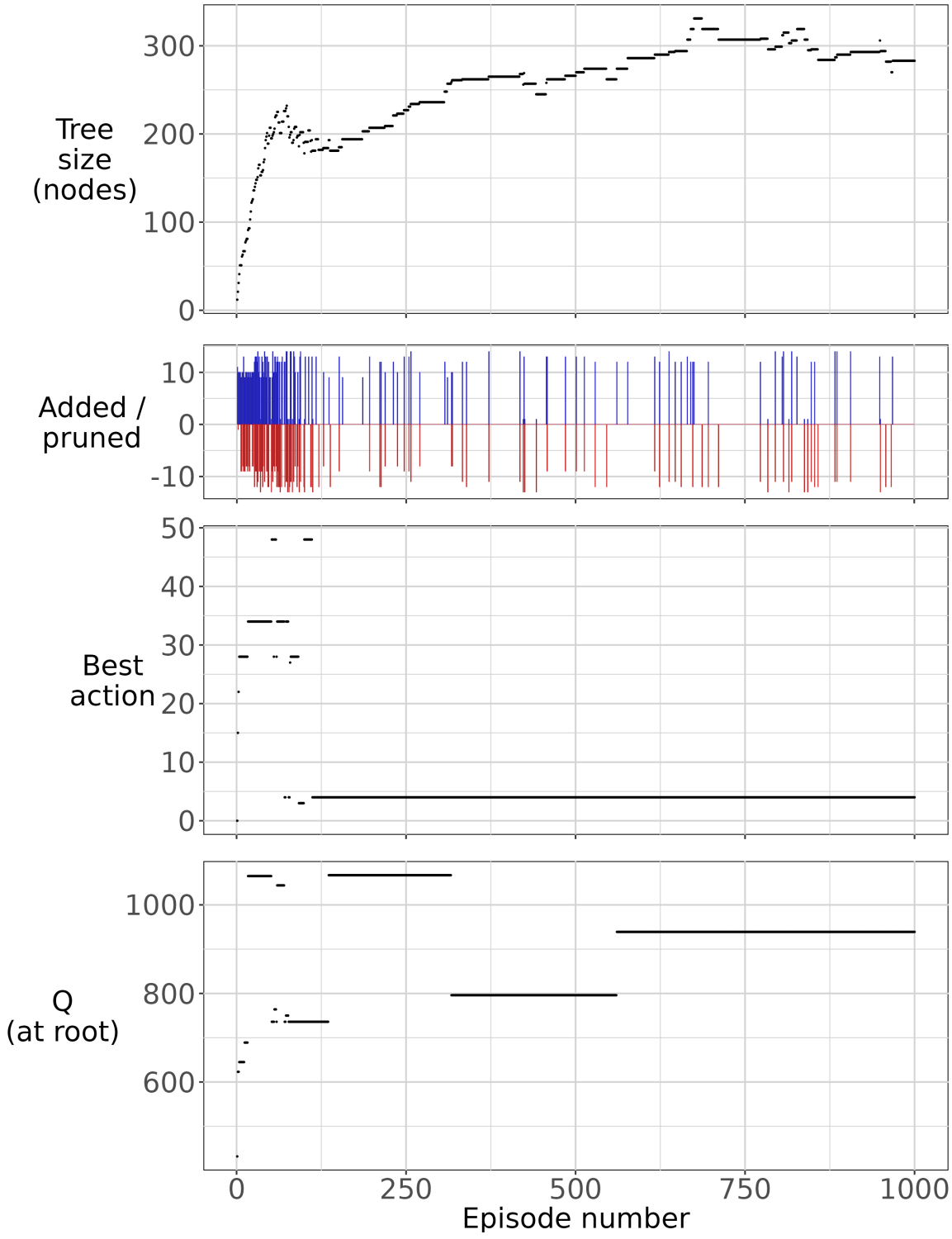
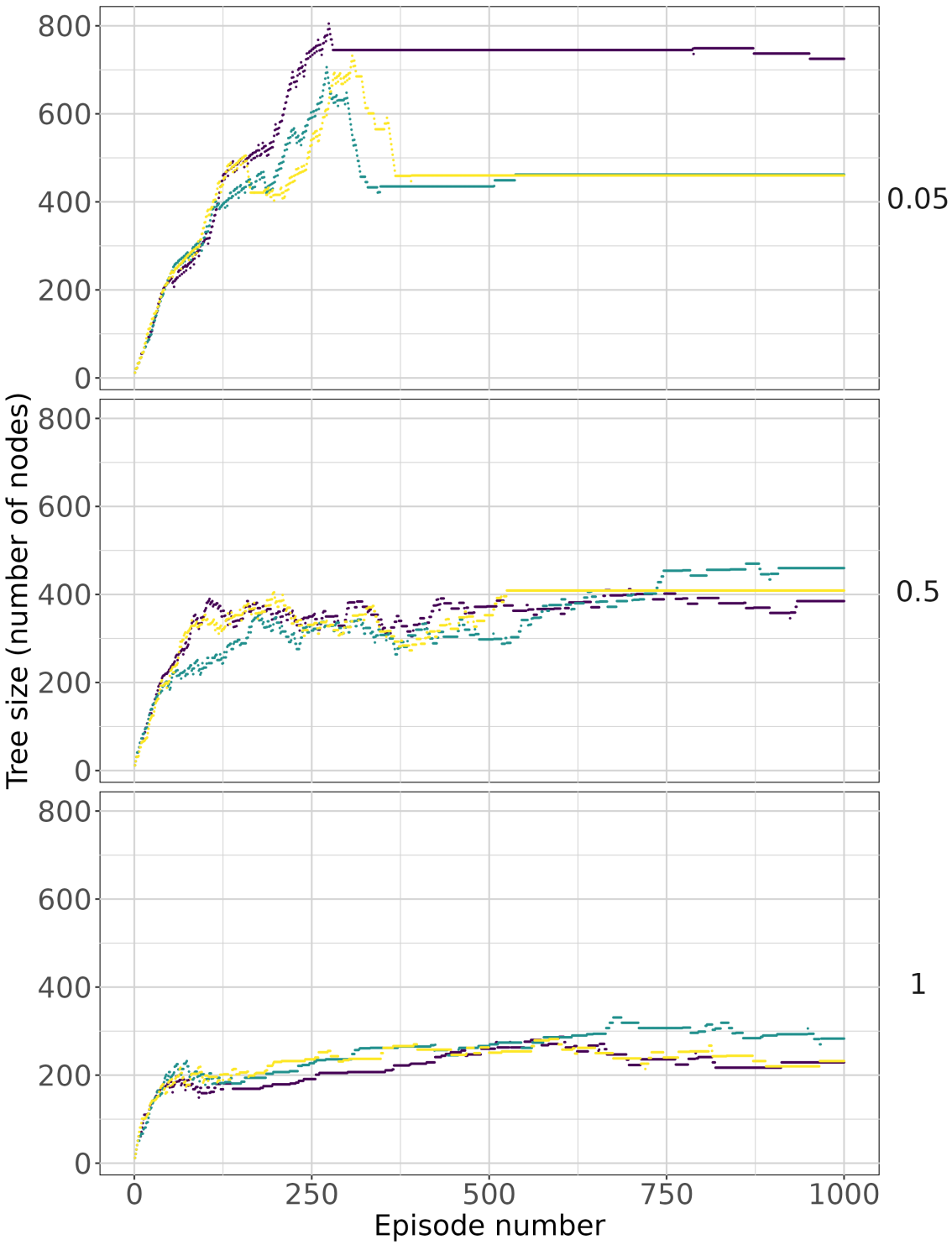


Figure 5.9: MCTS tree sizes before the first move



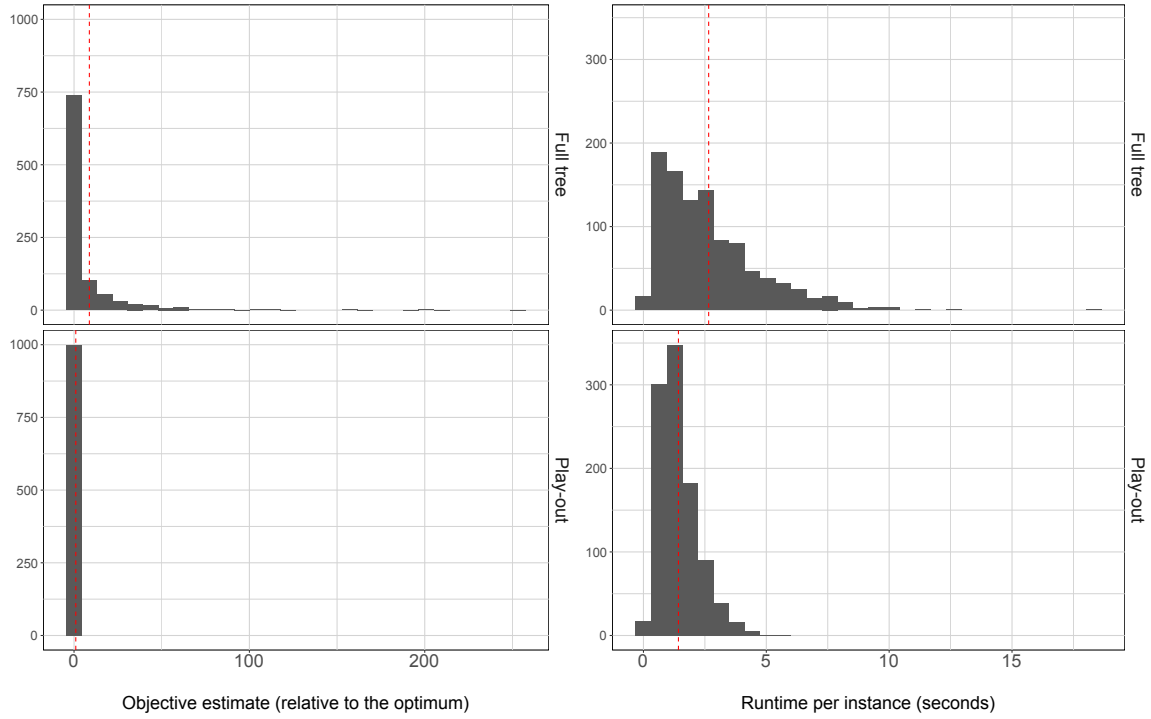

**Note.** Numbers to the right of the panels denote values for parameter $\varepsilon$. Different colors correspond to different runs over the same input.

ignoring the (now impossible) other moves. To investigate this effect, we generated $1,000$ random instances, mostly 17–20 nodes each (i.e., starting the generation from 20 nodes), with $b = 3$. We discarded the 132 instances where the initial optimality gap was positive, so for the remaining instances we would know the optimal objective (but not an optimal sequence of moves). Each such instance was solved by the usual MCTS play-out algorithm, but this time we recorded the total number of episodes (equal to the number of episodes per move multiplied by the number of moves until the end of the game). After that, we re-solved the same instance starting from an empty game tree, allowing the algorithm to run for exactly the same number of episodes, but without making any moves. Therefore, in the latter case the algorithm started the selection every time from the initial root node. The results are summarized in Figure 5.10. The first row represents the *full-tree* approach, i.e., trying to build an estimate for the best cost without making any moves. The bottom row represents the *play-out* strategy, when we run ten learning episodes followed by a turn repeatedly until the end of the game. The first column shows the histograms of the objective values obtained by the algorithms, relative to the true optimum. For these instances both approaches almost always yield the exact optimum. The right column presents the histograms of wall-clock run times per instance, in seconds.

While objective estimates are very comparable, focusing the tree by making turns (and switching the root node) allows the algorithm to reduce computational effort. The runtime is decreased by about one second on average over this dataset. (Vertical dashed lines represent respective mean values.) Also, the revised approach with play-outs results in a maximum computational time of approximately 5 seconds (as opposed to 10 for the full-tree approach)[1].

[1]It is interesting to note how this aligns to the author's experience with the game of Go: one of the Go proverbs allegedly says "Lose your first 50 games as quickly as possible." Apparently, this strategy allows the algorithm to avoid building the whole (prohibitively large) game tree staring at the board in the beginning. Therefore,the computational resource is spent on smaller situations which makes the "selection" and "roll-out" neural networks train better. This is a topic for further research, more connected to machine learning.

Figure 5.10: MCTS: play-out vs. first-move strategies



# Chapter 6

# Conclusions and Future Research

This work contributes to the growing body of literature on using decision diagrams in various forms, and touches a topic of applying Reinforcement Learning techniques to solve optimization problems.

The key contribution related to Chapter 3 is the idea of using simplified problems to align two BDDs. We formulated such a simplified problem based on simple upper bounds on the layer widths. We designed a heuristic that attained solution quality comparable to the one from the baseline heuristic involving BDDs, while operating over $O(N)$ (instead of $O(2^N)$) objects. In the context of optimization over collections of BDDs, this work contributes to the literature a practical approach that enables the use of state-of-the art methods that require aligned variable orders. Chapter 4 in fact provided a detailed case study for applying the proposed computational approach.

These ideas can be extended and improved to create practically efficient domain-specific algorithms. First, one could further explore the simplified problem formulations. For example, we could either seek to improve the weighted variable sequences concept (e.g., introducing the possibility for size decrease during the swap, assuming the diagrams to be quasi-reduced), or propose a fundamentally different model (such as embedding BDDs into $\mathbb{R}^N$ space with ML methods). Then, it may be possible to leverage interconnections

between the simplified and original problems. We could solve several simplified problems with the initial diagrams randomly shuffled in different ways, or even design a divide-and-conquer type of algorithm where BDD transformations would be interleaved with solving auxiliary problems. Finally, the branch-and-bound algorithm provided in this dissertation can be improved by fine-tuning bound estimates, branching strategies, or picking a different principle of branching.

From the methodological perspective, it would be interesting to consider how these ideas of working with collections of BDDs could be mixed with the concepts of relaxed and restricted decision diagrams (which are discussed, e.g., by Bergman et al. 2016a).

In terms of applications, one could look into other types of problems having several groups of binary constraints that would be natural to represent with collections of diagram. Having more types of problems might help us to formulate some constructive description of what kind of structures captured by BDDs seem to yield most benefits from such representations.

In Chapter 5 we focused on the DSPI problem, a complex variant of the Shortest-Path Interdiction problem. A decision variant of the problem is known to be NP-hard, and an exact algorithm presented in the literature reduces to enumerating all the relevant states in a dynamic programming fashion. While existing research discusses bounds for the optimal game cost, no research has yet studied heuristics providing solutions (policies) for the problem.

We have showed how several ideas from the game-playing and RL literature can be applied to this problem, and presented a heuristic algorithm based on the Monte Carlo Tree Search framework. After processing the instance data and formulating the game tree (given a computational budget), the algorithm yields the next recommended action for both players, and can be used to "play out" a DSPI instance. We demonstrate the practicality of the proposed approach over a series of numerical experiments and show that several key algorithmic components significantly contribute to the performance: guiding the search with cost estimates, pruning, and focusing the tree with making actual turns.

There are several lines of attack to improve the algorithm evident from the presented work. One idea is to cache the most frequently used states of the game and share the information between the game tree nodes. Some of the relevant ideas discussed in the literature include "transposition tables" (when the most-visited game states share the information across several nodes) and the ideas of all-moves-as-first roll-outs and its variations (Browne et al. 2012). Then, the very selection and roll-out process can be improved. For the game of Go, (Silver et al. 2017) neural networks were successfully used both for (randomized) node selection and roll-outs. It might be worthwhile to employ the idea of neural networks that are aware of the graph structure (such as Graph Neural Networks (GNN), see, e.g., Scarselli et al. (2009), Wu et al. (2019), Zhang et al. (2018) and Almasan et al. (2020)). It would be especially interesting to learn some patterns that would be persistent between different DSPI instances (in fact, that would mean learning a heuristic *algorithm*). Finally, one could try to generalize these ideas of Monte Carlo Tree Search to other complicated problems that allow for representation as exponential-state dynamic programs, perhaps starting with other variants of network interdiction problems.

As a final note, it would be interesting to look into possible connections between MCTS frameworks and multi-stage stochastic programming, as the latter shares this core component of a huge state space that is virtually impossible to sample in an exhaustive way.

# Bibliography


Akers, S. B. (1978). Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516.

Almasan, P., Suárez-Varela, J., Badia-Sampera, A., Rusek, K., Barlet-Ros, P., and Cabellos-Aparicio, A. (2020). Deep Reinforcement Learning meets Graph Neural Networks: Exploring a routing optimization use case. *arXiv:1910.07421 [cs]*.

Bengio, Y., Lodi, A., and Prouvost, A. (2021). Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421.

Bergman, D. and Cire, A. A. (2016). Decomposition based on decision diagrams. In Quimper, C.-G., editor, *Integration of AI and OR Techniques in Constraint Programming*, volume 9676, pages 45–54. Springer International Publishing, Cham.

Bergman, D., Cire, A. A., van Hoeve, W.-J., and Hooker, J. N. (2016a). *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer International Publishing.

Bergman, D., Cire, A. A., van Hoeve, W.-J., and Hooker, J. N. (2016b). Discrete Optimization with Decision Diagrams. *INFORMS Journal on Computing*, 28(1):47–66.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98.

Bollig, B., Löbbing, M., and Wegener, I. (1996). On the effect of local changes in the variable ordering of ordered decision diagrams. *Information Processing Letters*, 59(5):233–239.

Bollig, B. and Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002.

Brace, K. S., Rudell, R. L., and Bryant, R. E. (1990). Efficient implementation of a BDD package.

In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA. ACM.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

Bryant, R. E. (1991). On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213.

Bryant, R. E. (1992). Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.

Bryant, R. E. (2018). Binary decision diagrams. In Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R., editors, *Handbook of Model Checking*, pages 191–217. Springer International Publishing, Cham.

Cabodi, G., Quer, S., Meinel, C., Sack, H., Slobodová, A., and Stangier, C. (1998). Binary decision diagrams and the multiple variable order problem. In *International Workshop on Logic Synthesis*, pages 346–352, Lake Tahoe, CA.

Drechsler, R. and Becker, B. (1998). *Binary Decision Diagrams: Theory and Implementation.* Kluwer, Boston.

Drechsler, R., Drechsler, N., and Günther, W. (1998). Fast exact minimization of BDDs. In *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, pages 200–205.

Drechsler, R. and Günther, W. (2000). History-based dynamic minimization during BDD construction. In Silveira, L. M., Devadas, S., and Reis, R., editors, *VLSI: Systems on a Chip: IFIP TC10 WG10.5 Tenth International Conference on Very Large Scale Integration (VLSI'99) December 1–4, 1999, Lisboa, Portugal*, IFIP — The International Federation for Information Processing, pages 334–345. Springer US, Boston, MA.

Drechsler, R., Günther, W., and Somenzi, F. (2001). Using lower bounds during dynamic BDD minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(1):51–57.

Ebendt, R. and Drechsler, R. (2005). Quasi-exact BDD minimization using relaxed best-first search. In *IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*, pages 59–64.

Ebendt, R. and Drechsler, R. (2006). Effect of improved lower bounds in dynamic BDD reordering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5):902–909.

Ebendt, R., Günther, W., and Drechsler, R. (2005). Combining ordered best-first search with branch and bound for exact BDD minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1515–1529.

Friedman, S. J. and Supowit, K. J. (1987). Finding the optimal variable ordering for binary decision diagrams. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 348–356. ACM.

Fujita, M., Matsunaga, Y., and Kakuda, T. (1991). On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation.*, pages 50–54, Amsterdam, Netherlands. IEEE.

Fulkerson, D. R. and Harding, G. C. (1977). Maximizing the minimum source-sink path subject to a budget constraint. *Mathematical Programming*, 13(1):116–118.

Gilbert, E. N. (1959). Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144.

Hooker, J. N. (2013). Decision diagrams and dynamic programming. In Gomes, C. and Sellmann, M., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 94–110, Berlin, Heidelberg. Springer.

Ishiura, N., Sawada, H., and Yajima, S. (1991). Minimization of binary decision diagrams based on exchanges of variables. In *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, pages 472–475.

Israeli, E. and Wood, R. K. (2002). Shortest-path network interdiction. *Networks*, 40(2):97–111.

Jain, J., Adams, W., and Fujita, M. (1998). Sampling schemes for computing OBDD variable orderings. In *1998 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (IEEE Cat. No.98CB36287)*, pages 631–638.

Karwowski, J. and Mańdziuk, J. (2019). A Monte Carlo Tree Search approach to finding efficient patrolling schemes on graphs. *European Journal of Operational Research*, 277(1):255–268.

Knuth, D. E. (2009). Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. In *The Art of Computer Programming*, volume 4. Addison-Wesley Professional., 1st edition.

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Priming'. *Artificial Intelligence*, page 34.

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In Fürnkranz, J., Scheffer, T., and Spiliopoulou, M., editors, *Machine Learning: ECML 2006*, Lecture Notes in Computer Science, pages 282–293, Berlin, Heidelberg. Springer.

Kocsis, L., Szepesvari, C., and Willemson, J. (2006). Improved Monte-Carlo Search. *undefined*.

Lee, C. Y. (1959). Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999.

Lin, S.-S. and Wei, C.-J. (2005). A new approach for minimization of binary decision diagrams. *Canadian Journal of Electrical and Computer Engineering*, 30(4):207–214.

Lozano, L., Bergman, D., and Smith, J. C. (2020). On the Consistent Path Problem. *Operations Research*, 68(6):1913–1931.

Mazyavkina, N., Sviridov, S., Ivanov, S., and Burnaev, E. (2021). Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400.

Meinel, C. and Slobodová, A. (1997). Speeding up variable reordering of OBDDs. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 338–343.

Meinel, C. and Theobald, T. (1998). *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*. Springer-Verlag, Berlin Heidelberg.

Minato, S. (2013). Techniques of BDD/ZDD: Brief history and recent activity. *IEICE Transactions on Information and Systems*, E96-D(7):1419–1429.

Nevo, Z. and Farkash, M. (2006). Distributed dynamic BDD reordering. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 223–228, New York, NY, USA. ACM.

Owen, S. H. and Daskin, M. S. (1998). Strategic facility location: A review. *European Journal of Operational Research*, 111(3):423–447.

Panda, S. and Somenzi, F. (1995). Who are the variables in your neighborhood. In *Proceedings of*

*the 1995 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '95, pages 74–77, Washington, DC, USA. IEEE Computer Society.

Pardalos, P. M., Du, D., and Graham, R. L., editors (2013). *Handbook of Combinatorial Optimization*. Springer Reference. Springer, New York, second edition edition.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

ReVelle, C. S., Eiselt, H. A., and Daskin, M. S. (2008). A bibliography for some fundamental problem categories in discrete location science. *European Journal of Operational Research*, 184(3):817–848.

Rimmel, A., Teytaud, O., Lee, C.-S., Yen, S.-J., Wang, M.-H., and Tsai, S.-R. (2010). Current Frontiers in Computer Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):229–238.

Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

Scholl, C., Becker, B., and Brogle, A. (2001). The multiple variable order problem for binary decision diagrams: Theory and practical application. In *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, pages 85–90, Yokohama, Japan. IEEE.

Sedgewick, R. and Wayne, K. D. (2011). *Algorithms*. Addison-Wesley, Upper Saddle River, NJ, 4th ed edition.

Sefair, J. A. and Smith, J. C. (2016). Dynamic shortest-path interdiction. *Networks*, 68(4):315–330.

Sieling, D. (2002). The nonapproximability of OBDD minimization. *Information and Computation*, 172(2):103–138.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359.

Slobodová, A. and Meinel, C. (1998). Sample method for minimization of OBDDs. In Goos, G.,

Hartmanis, J., van Leeuwen, J., and Rovan, B., editors, *SOFSEM' 98: Theory and Practice of Informatics*, volume 1521, pages 419–428. Springer Berlin Heidelberg, Berlin, Heidelberg.

Smith, J. C. and Song, Y. (2020). A survey of network interdiction models and algorithms. *European Journal of Operational Research*, 283(3):797–811.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, Massachusetts, second edition edition.

Wegener, I. (2000). *Branching Programs and Binary Decision Diagrams*. Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics.

Wegener, I. (2004). BDDs — design, analysis, complexity, and applications. *Discrete Applied Mathematics*, 138(1):229–251.

Wickham, H. (2016). *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Yu, P. S. (2019). A Comprehensive Survey on Graph Neural Networks. *arXiv:1901.00596 [cs, stat]*.

Zhang, Z., Cui, P., and Zhu, W. (2018). Deep Learning on Graphs: A Survey. *arXiv:1812.04202 [cs, stat]*.