On Aligning Non-Order-Associated Binary Decision Diagrams

Alexey A. Bochkarev[1] and J. Cole Smith[2]

[1]Department of Industrial Engineering, Clemson University, Clemson, SC 29634, abochka@g.clemson.edu

[2]Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244, colesmit@syr.edu

August 3, 2022

Abstract

Recent studies employ collections of binary decision diagrams (BDDs) to solve combinatorial optimization problems. This paper focuses on the problem of optimally aligning two BDDs, i.e., transforming them to enforce a common order of variables while keeping the total size of the diagrams as small as possible. We address this problem, which is known to be NP-hard, by introducing and studying a simplified problem instead of working with the more complex original diagrams. We discuss some basic properties of the simplified problem, design a corresponding heuristic for the original problem, and show empirically that this approach yields good quality alignments while significantly reducing the complexity of intermediate diagram transformations. We highlight the practicality of this approach in the context of a variation of the uncapacitated facility location problem.

1 Introduction

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 paper, 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.

The conditions needed to make the aforementioned transformation are that the collection of BDDs must have *aligned variable orders*, as defined in Section 2, 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 our paper.

Certain problems (e.g., those studied in Bergman et al. 2016 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. However, as we will illustrate in Section 4.2, many other problems have no natural variable alignments, and no clear mechanism exists to choose one.

The rest of this paper is organized as follows. Section 2 provides necessary background on BDD and CPP concepts. We present algorithms for attacking the variable alignment problem in Section 3. We then illustrate our approach in Section 4 on two sets of random instances: one on random diagrams and another in the context of a variation on a classical facility location problem. Section 5 concludes the paper.

2 Problem description and background

This section provides the requisite technical background for this paper. Section 2.1 formally introduces BDDs (Appendix A presents some further details on the definitions and their relation to the existing literature). Section 2.2 illustrates how large BDDs can be represented by a collection of BDDs having overlapping variables, necessitating the solution of a CPP. We then show in Section 2.3 how representing a COP via a collection of BDDs yields the variable alignment problem that is the focus of this paper. Section 2.4 briefly discusses research on related methods in this context.

2.1 Binary decision diagrams

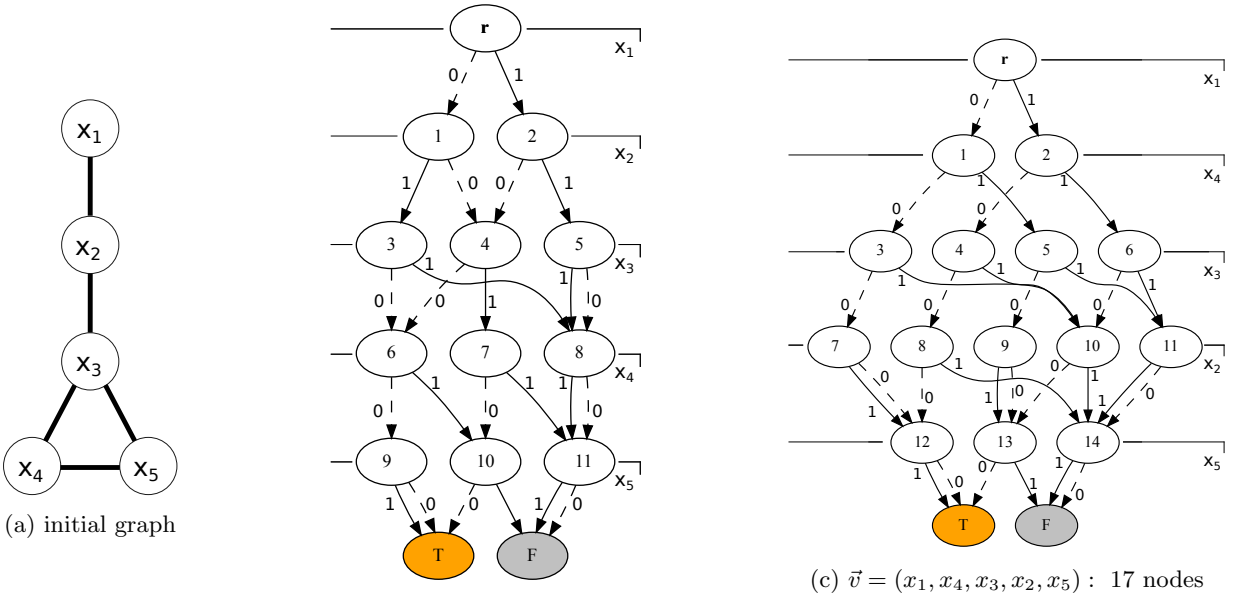
A BDD B is an acyclic graph having a node set $\mathcal{N}$ that possesses the following properties.

- 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 `r`), and the last layer comprises two terminal nodes, referred to as *true* (**T**) and *false* (**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 $\text{var}(B) = \vec{v}$, we say that B "has variable order $\vec{v}$.")
- 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, or above 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 **T** and **F** have no outgoing arcs.
- Each node outside the first and the last layer has at least one incoming arc (emanating from a node on the previous layer). Also, `r` has no incoming arcs, and at least one of nodes **T** and **F** has incoming arcs.
- For convenience, we define a "path" to be a path from `r` to **T** or **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 **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 **T** (respectively, **F**).

Remark 1 *In much of this paper, 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 1 *Figure 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 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*



(a) initial graph

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

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

Figure 1: BDD representation of the maximum independent set problem. One-arcs (zero-arcs) are depicted with solid (respectively, dashed) lines and given unit (respectively, zero) lengths.

at $\boldsymbol{T}$ *(*$\boldsymbol{F}$*), 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 1b displays a BDD constructed from this ordering. Note, for example, that BDD node* 5 *in Figure 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 1b terminate at* $\boldsymbol{F}$*.*

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 1c. Because the MIS can be solved by identifying a longest-path from* $\boldsymbol{r}$ *to* $\boldsymbol{T}$ *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 1 *BDDs* B *and* B' *are* equivalent *if for every path in* B *(or* B'*) there exists a path in* B' *(or* B*) that corresponds to the same variable assignment and ends at the same terminal node (*$\boldsymbol{T}$ *or* $\boldsymbol{F}$*).*

Remark 2 *Where possible, we will also ignore BDD arc lengths in this paper to simplify our exposition. Our subsequent results easily extend to problems having arc weights, and the source code accompanying this paper accommodates weighted BDDs.* □

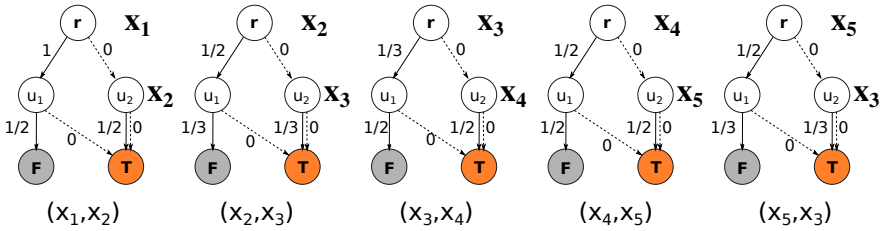


Figure 2: Representation of the independent set problem depicted in Figure 1a with a collection of BDDs.

2.2 Representing problems with multiple BDDs

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

Example 2 *For the MIS example, one could design a collection of BDDs as in Figure 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).

2.3 Aligning variables of a BDD pair

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

Definition 3 *Given two BDDs A and B defined over the same variable set, the alignment problem solves:*

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

where $\vec{v}$ belongs to the set of all possible permutations of the labels in the variable list, $\mathrm{var}(A)$. We refer to this problem as the "original" one, as opposed to a "simplified" version considered later, and parameterize it as $\mathrm{AP}(A, B; T^*)$. 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).

2.4 Background on BDD minimization

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 (CPP and market multisplit being one of many examples), as discussed by Bergman et al. (2016) and Lozano et al. (2020).

The problem of aligning two diagrams, which we study in this paper, 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. 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 $\mathrm{AP}(A, B; T^*)$ defined above.) 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.

3 Problem simplification

We begin in Section 3.1 by describing operations that transform a (single) BDD into an equivalent BDD having a different variable order. These operations can create diagrams having an exponential number of nodes (in the number of variables). As an alternative, we introduce an auxiliary ("simplified") problem in Section 3.2. This simplified problem operates over smaller objects and yields a heuristic solution to the original problem. In Section 3.3 we derive properties of optimal solutions to the simplified problem, which we then leverage in devising algorithms for solving the problem in Section 3.4.

3.1 Revising a BDD based on a revised variable order

We briefly introduce the operations required to transform a BDD, B, into an equivalent BDD, $R(B)$, having a different variable order. We focus primarily on *swap* operations, which exchange positions of two adjacent layers, and *sift-up* operations, which change a single layer's position from s to $d < s$, with the relative positions of other layers being fixed. A *sift-down* operation is analogous to sift-up, with an adjustment to $d > s$. The relevant algorithms are presented in Appendix A; see also Bollig et al. (1996), Wegener (2000) in Section 5.7, and Knuth (2009) in Section 7.1.4 for similar concepts related to the following operations and bounds. (In particular, our *sift-up* and *sift-down* are also referred to as *jump-up* and *jump-down* in the literature.) These operations require polynomial space and time in the size of the underlying diagram (Wegener 2000).

We summarize the idea behind the sift-up operation here (which subsumes the swap operation), with technical details presented in Appendix A (see also Bollig et al. 1996). In sifting a layer up from position s to $d < s$, the transformed network $R(B)$ is 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 $\text{var}(L_s^B)$. We then generate two copies (a one-copy and a zero-copy) of the BDD between layers L_d^B and L_{s-1}^B, and include them in the transformation. The one-copy (respectively, zero-copy) assumes that $\text{var}(L_s^B)$ is set to 1 (respectively, 0). These copies are 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.

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 $\text{var}(L_{s-1}^B)$. An example transformation is given in Figures 3 and 4, and the procedure is formally described in Algorithm 2 of Appendix A.

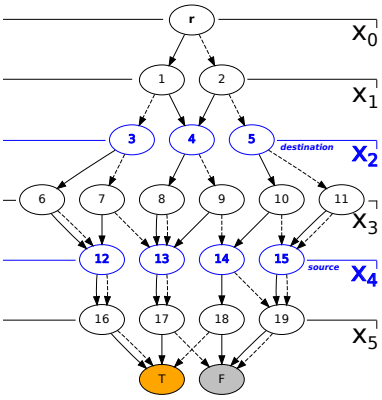


Figure 3: Original diagram B.

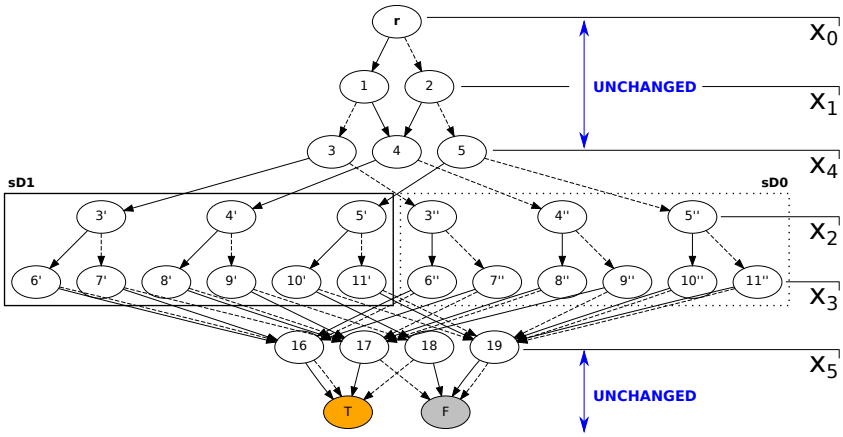


Figure 4: Resulting diagram $R(B)$ (after the sift).

3.2 The simplified problem

We first introduce a data structure that keeps track of the upper bounds on layer widths as we perform layer sifts. Define a *weighted variable sequence* over variables $x_1, \ldots, x_N$, denoted by $S \stackrel{\text{def}}{=} [x_1, \ldots, x_N | n_1, \ldots, n_N]$, as an ordered list of N variable names $\text{var}(S) \stackrel{\text{def}}{=} (x_1, \ldots, x_N)$ 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 $|S| \stackrel{\text{def}}{=} \sum n_i$. 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 $\text{var}(S)$; 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 and 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 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: $[x_1, \overset{\frown}{x_2, x_3}, x_4 | n_1, n_2, n_3, n_4] \to [x_1, x_3, x_2, x_4 | n_1, n_2, 2n_2, n_4]$.
- $\texttt{sift}(S, s, d)$**:** 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 *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* $d < i \le s$ *and* $n'_i = n_i$ *otherwise. For a sift-down* $(d > s)$ *:* $n'_i = 2^{(i-s)} n_s$ *for* $s < i \le d$ *and* $n'_i = n_i$ *otherwise.* □

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

Remark 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, $|T^*_{VS}[S, \vec{v}]| \geq |S|$.* □

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

Definition 4 *Given the original problem $AP(A, B; T^*)$, define variable sequences S_A and S_B with variable lists $var(S_A) = var(A)$ and $var(S_B) = var(B)$. Define element weights to equal the corresponding layer widths: $n^A_i = |L^A_i|$ and $n^B_i = |L^B_i|$ for all $i = 1, \ldots, N$. Overloading the notation, we refer to the problem $AP(S_A, S_B; T^*_{VS})$ as the simplified problem for $AP(A, B; T^*)$.*

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

Lemma 1 *Given BDD A and its corresponding sequence S (constructed as per Definition 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 $\vec{v}$: $|T^*[A, \vec{v}]| \leq |T^*_{VS}[S, \vec{v}]|$.*

Lemma 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 $T^*[A, \vec{v}]$. (The formal proof is omitted for brevity.)

Corollary 1 *The optimal objective s^* of $AP(A, B; T^*)$ is no more than the optimal objective s^*_{VS} for $AP(S_A, S_B; T^*_{VS})$.*

Proof. Assume the contrary is true. If there is an optimal variable order shared between $\mathrm{AP}(A, B; T^*)$ and $\mathrm{AP}(S_A, S_B; T^*_{\mathrm{VS}})$, then the claim immediately follows from Lemma 1. Otherwise, there must exist an optimal variable order $\vec{u}$ for $\mathrm{AP}(S_A, S_B; T^*_{\mathrm{VS}})$ corresponding to the optimal value $s^*_{\mathrm{VS},\vec{u}} = |T^*_{\mathrm{VS}}[S_A, \vec{u}]| + |T^*_{\mathrm{VS}}[S_B, \vec{u}]| < s^*$. But due to Lemma 1, $|T^*_{\mathrm{VS}}[S_A, \vec{u}]| \geq |T^*[A, \vec{u}]|$ and $|T^*_{\mathrm{VS}}[S_B, \vec{u}]| \geq |T^*[B, \vec{u}]|$. Hence, $s^*_{\mathrm{VS},\vec{u}} \geq |T^*[A, \vec{u}]| + |T^*[B, \vec{u}]| \geq s^*$, 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 2 *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 2 *Consider a variable sequence S with $var(S) = \vec{v}$. If an exponentially weighted subsequence spans positions $k, \ldots, l$, then $|S| = |T^*_{VS}[S, \vec{u}]|$ for any $\vec{u}$ such that $u_i = v_i$ for all $i < k$ and all $i > l$.*

We will say that a pair of elements $a, b \in \mathrm{var}(S)$ *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 *For a variable sequence S and an arbitrary permutation of $var(S)$, denoted $\vec{v}$, there exists a non-redundant schedule of swaps that yields $T^*_{VS}[S, \vec{v}]$.*

Proof. First, note that $T_{\mathrm{VS}}[S, \vec{v}] \neq \varnothing$ (since any two adjacent elements can be swapped), and $|T_{\mathrm{VS}}[S, \vec{v}]|$ is finite, because there is a finite number of valid variable sequences with variable order $\vec{v}$. Hence, there is an optimal schedule of swaps that yields $T^*_{\mathrm{VS}}[S, \vec{v}]$. 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 4, the resulting sequence will have size at most $|T^*_{\mathrm{VS}}[S, \vec{v}]|$ (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 1. Without loss of generality, assume elements are labeled as $1, \ldots, N$, and the target order is $\vec{v} = (1, \ldots, N)$. 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 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 $T^*_{\mathrm{VS}}[S, \vec{v}]$.

Algorithm 1 Align-to (weighted variable sequence)

Input: $S = [x_1, \ldots, x_n | n_1, \ldots, n_N]$ – a weighted variable sequence
Output: $T^*_{\mathrm{VS}}[S, \vec{v}]$ for $\vec{v} = (1, \ldots, N)$
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 1 iterates at most N times. This algorithm is illustrated with a specific example in Appendix B.

Remark 5 *It can be shown that all non-redundant schedules of swaps transforming S to variable order $\vec{v}$ yield the same variable sequence, and hence $T^*_{VS}[S, \vec{v}]$ is unique. Appendix C presents a linear-time optimal transformation algorithm.* □

Remark 6 *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 4.) Therefore, the simplified problem could return a solution that is arbitrarily good or bad with respect to the original problem objective.* □

3.3 Properties of an optimal solution to the simplified problem

We now derive the following properties of optimal solutions to the simplified problem, which we will use to design a heuristic. 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 4 (Aligned pair) *Given an instance of $AP(S_A, S_B; T^*_{VS})$ and any pair of elements $a, b \in var(S_A)$ such that $a \prec_{S_A} b$ and $a \prec_{S_B} b$, there exists an optimal solution $\vec{v^*}$ such that $a \prec_{\vec{v^*}} b$.*

Proof. The claim is illustrated in Figure 5a. Consider any optimal alignment $\vec{v}$, and suppose that $b \prec_{\vec{v}} a$. Consider two non-redundant schedules of swaps that transform S_A and S_B to $\vec{v}$. 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 4 the total size of the resulting sequences will be no more than $|T^*_{\mathrm{VS}}[S_A, \vec{v}]|$ and $|T^*_{\mathrm{VS}}[S_B, \vec{v}]|$. Thus, $\vec{v^*}$ 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 $\vec{v}$, and element w of $\vec{v}$, define $C_{S,w} \stackrel{\text{def}}{=} \{x : w \prec_S x\}$ as the set of elements in S *covered* by w.

Corollary 3 (Covered elements) *For any instance of $AP(S_A, S_B; T^*_{VS})$, there exists an optimal variable order $\vec{v^*} = (v^*_1, \ldots, v^*_N)$ such that:*

- $C_{S_A,v^*_N} \cap C_{S_B,v^*_N} = \varnothing$;
- $a \prec_{\vec{v^*}} b$ *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 5b. To see that $C_{S_A,v^*_N} \cap C_{S_B,v^*_N} = \varnothing$ for some optimal $\vec{v^*}$, 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 4 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 5c, consider an optimal alignment $\vec{v}$, and suppose that $a, b \in C_{S_A,v^*_N}$, $a \prec_{S_B} b$, and $b \prec_{\vec{v}} a$. Let $\vec{v^*}$ be a modified version of $\vec{v}$ in which $a \prec_{\vec{v^*}} b$. Note that $|T^*_{\text{VS}}[S_A, \vec{v}]| = |T^*_{\text{VS}}[S_A, \vec{v^*}]|$, because reordered elements belonged to the same exponentially weighted subsequence. Now, observe that we can modify the schedule that builds $T^*_{\text{VS}}[S_B, \vec{v}]$ to build $T^*_{\text{VS}}[S_B, \vec{v^*}]$ by omitting all swaps between elements a and b. Due to Remark 4, $|T^*_{\text{VS}}[S_B, \vec{v^*}]| \leq |T^*_{\text{VS}}[S_B, \vec{v}]|$. Hence, $\vec{v^*}$ 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 5 (Target first element) *Given an instance of $AP(S_A, S_B; T^*_{VS})$, 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 $\vec{v^*}$ such that $v^*_1 \in \{a_1, b_1\}$.*

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

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

3.4 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 4). However, local

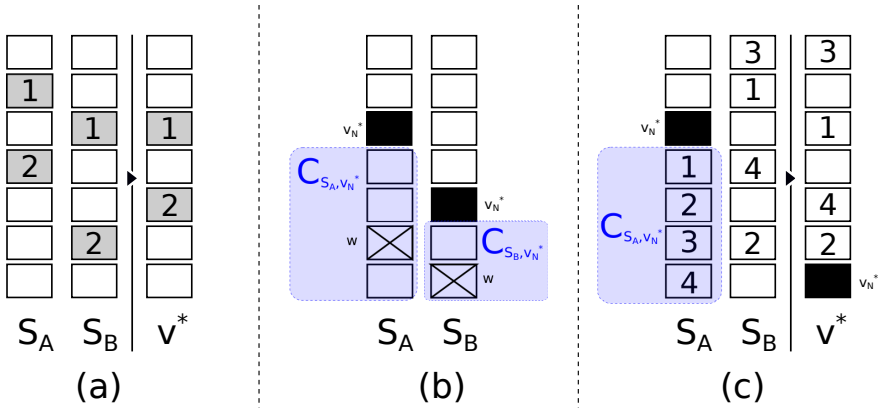


Figure 5: Illustrations for optimal solution properties. **(a):** "Aligned pair" (elements 1 and 2); **(b):** "Covered elements" ($w \in C_{S_A,v_N^*} \cap C_{S_B,v_N^*}$); **(c):** "Covered elements" (1, 2, 3, and 4).

search procedures that require a decreasing objective at each step can yield local optima that are inferior to global optima. To find a global optimum for the simplified problem, we design a branch-and-bound procedure presented in Appendix D. It is built upon Algorithm 1 in the following way. A search tree node represents a pair of partially aligned sequences, $T^*_{\mathrm{VS}}[S_A, \vec{v^1}]$ and $T^*_{\mathrm{VS}}[S_B, \vec{v^2}]$, such that tails of $\vec{v^1}$ and $\vec{v^2}$ coincide (i.e., $v^1_i = v^2_i$ for all $i > N_f$ for some N_f). The key idea is to branch on the next element aligned at position N_f, decreasing it from N down to 1 (choosing the last element in the target order, then the next to last, and so on). Technical details of the implementation are presented by Algorithm 5. The key component of the algorithm was the procedure to calculate lower bounds. We tested several approaches (presented in Appendix E), and chose the one based on the following idea.

Lemma 6 *For $S = [a_1, \ldots, a_N | n_1, \ldots, n_N]$ and arbitrary $\vec{v}$, define $I(S, \vec{v}) \stackrel{def}{=} \{(j, l) : \; a_j \prec_S a_l$ and $a_l \prec_{\vec{v}} a_j\}$. Then:*

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

Proof. Let us consider the optimal schedule of swaps as implemented in Algorithm 1. 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 14). 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 2). The lower bound due to swaps in this first step would be maximal if $(k, l) \in I(S, \vec{v})$ for all $j \leq k < l \leq N$. Then, the contribution to the lower bound would be

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

Denoting $K \stackrel{\text{def}}{=} N - j$, the difference between the actual size increase and the change in the lower bound at the first step of Algorithm 1 is at most:

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

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

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

Substituting $\sum_{k=1}^{K} 2^K n_j = (2^{K+1}-2)n_j$ we can revise (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 $n_{j+k} \leq 2n_{j+k-1}$ for all k, we obtain:

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

Noting that $\sum_{k=1}^{K-1}(K-k)2^k = 2^{K+1} - 2K - 2$ when $K \geq 2$, we get that $\Delta \geq 0$. Therefore, the size increase of the sequence is at least as large as the lower bound contribution at the first step of Algorithm 1.

Now, consider the next step of Algorithm 1, 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 4 *For $AP(S_A, S_B; T^*_{VS})$, the optimal objectives* satisfies:*

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

.

Proof. Consider an optimal alignment $\vec{v}$. 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 $a_l \prec_{\vec{v}} a_k$ and $\delta_{lj} = 0$ otherwise. Applying Lemma 6 twice:

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

Taking the sum of the two inequalities, we conclude that

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

regardless of specific δ_{lj} values. □

4 Numerical experiments

We test the proposed approach against two classes of test instances. First, we analyze the relative performance of different heuristics on randomly generated diagrams. Then, we examine a featured combinatorial optimization application that can benefit from applying the proposed heuristic. As per Remark 1, we ensure that all BDDs are quasi-reduced by merging redundant pairs of nodes where possible. The following two subsections provide details for each of these investigations.

4.1 Aligning random diagrams

First, we generate 10,048 random fifteen-variable BDD pair alignment instances. (Appendix F describes details of this process.) For each instance, we construct a simplified problem instance as per Definition 4, obtain a solution, and revise BDDs to the target variable order derived from the simplified problem. We will discuss the simplified and original problems in Sections 4.1.1 and 4.1.2, respectively.

4.1.1 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.

- **Best of A and B:** calculate the objectives for $\mathrm{var}(S_A)$ and $\mathrm{var}(S_B)$ 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

Table 1: Number of times (out of 10,048) each heuristic achieved the objective performances listed by column.

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

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.

The performance of these strategies in terms of objective values are summarized in Table 1. 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 6 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.

4.1.2 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 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 $\mathrm{var}(B)$ as the initial target variable order, and choose the best solution out of the two.

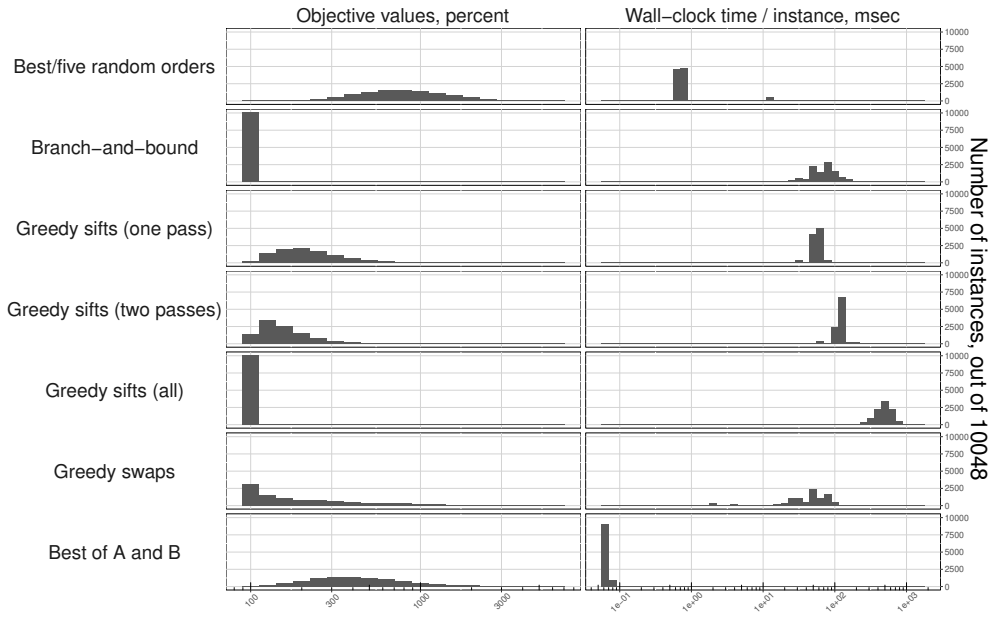


Figure 6: Computational time and relative objective values for various heuristics, simplified problem.

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). The resulting objective value distributions are presented in Figure 7.

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 8). We generated 200 instances for each problem size, from 5 to 25 variables (see Appendix F for details). 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 8a. 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 8b 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 problem 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 8, our heuristic outperformed the greedy sifts heuristic

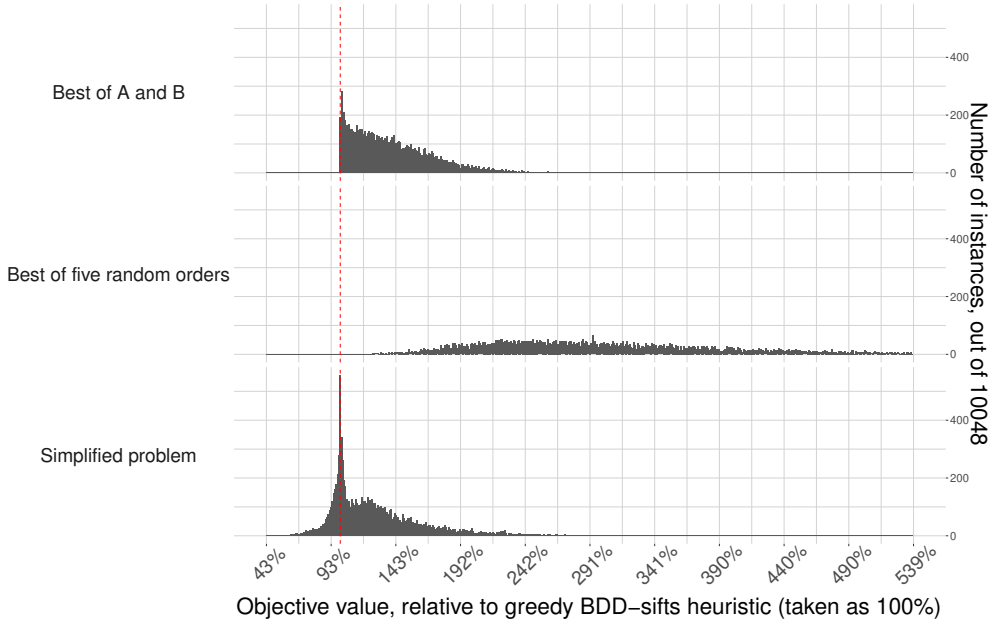


Figure 7: Histogram of the aligned BDD sizes for different heuristics, original problem.

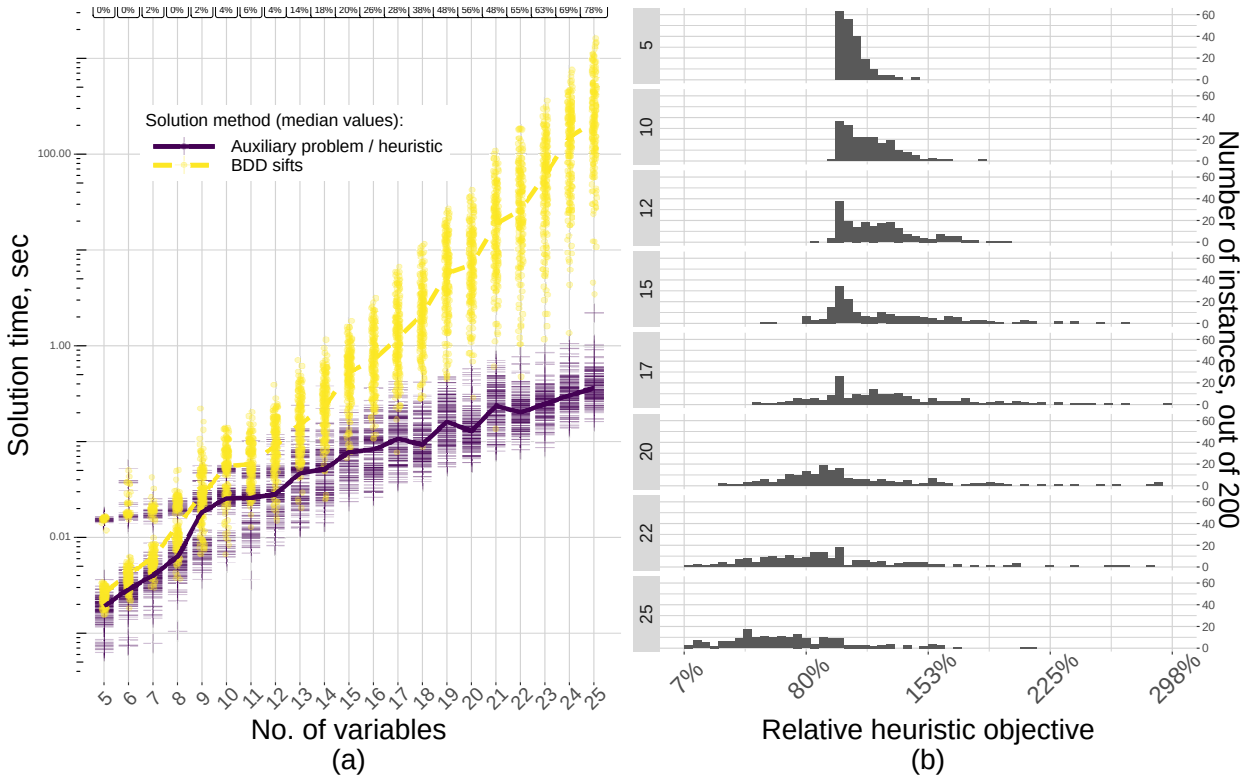


Figure 8: Changes in runtimes and objective values for the proposed heuristic versus the baseline of greedy BDD sifts for varying problem sizes.

on instances containing more than twenty variables both in terms of time and objective on more than half of the instances.

Remark 7 *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 original problems share structural properties. Appendix G analyzes this question in detail. Despite the fact that our heuristic can yield arbitrarily poor solutions as per Remark 6, 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 Appendix G.*

4.2 Joint uncapacitated facility location problem

We next briefly illustrate the performance of our algorithms in the context of the *joint uncapacitated facility location problem* (j-UFLP). The j-UFLP is a variation of the uncapacitated facility location problem (see, e.g., Owen and Daskin 1998, ReVelle et al. 2008), which contains structures that can be readily exploited by our proposed approach. The j-UFLP comprises two sub-instances, each one representing an uncapacitated facility location problem over a separate graph of N points, linked by side constraints. At each point i in sub-instance s, we can locate a facility at cost c_i^s, covering all points in some given set S_i^s. (Set S_i^s may, for instance, refer to customers that are sufficiently close to location i in sub-instance s.) Furthermore, covering point i exactly a times in sub-instance s yields an *overlap cost* of $f_i^s(a)$, where $a \in \{0, 1, \ldots, |S_i^s|\}$ (assuming $i \in S_i^s$). Additionally, linking constraints enforce that each location decision in the first sub-instance coincides with a certain location decision in the second sub-instance. In particular, for each facility in the first instance $i \in \{1, \ldots, N\}$ we define $m(i) \in \{1, \ldots, N\}$ as its linked facility in the second sub-instance. We require that a facility is located at point i in the first sub-instance if and only if a facility is located at point $m(i)$ in the second sub-instance. Therefore, locating a facility at point i in the first sub-instance in a feasible solution requires location of the linked facility in the second one, and yields the total location cost of $c_i^1 + c_{m(i)}^2$. The overlap costs, however, are non-trivial, because the adjacency sets S_i^1 and $S_{m(i)}^2$ are, generally speaking, different.

The j-UFLP minimizes the sum of facility location costs and overlap costs. A baseline mixed-integer programming (MIP) formulation can be expressed as follows:

$$\min \sum_{i=1}^{N} \left(c_i^1 x_i^1 + f_i^1(a_i^1) \right) + \sum_{i=1}^{N} \left(c_i^2 x_i^2 + f_i^2(a_i^2) \right), \tag{2a}$$

$$\text{s.t. } a_i^s = \sum_{j \in S_i^s} x_i^s \qquad \forall i = 1, \ldots, N \text{ and } s = 1, 2, \tag{2b}$$

$$x_i^1 = x_{m(i)}^2 \qquad \forall i = 1, \ldots, N, \tag{2c}$$

$$x_i^s \in \{0, 1\} \qquad \forall i = 1, \ldots, N, \text{ and } s = 1, 2. \tag{2d}$$

The j-UFLP models a situation in which facilities support two different networks at the same time, but in different ways. Note that we assume $m(i)$ to be one-to-one mapping between the

sub-instances nodes for convenience in exposition, but without loss of generality. The situation when some nodes in one sub-instance are not in fact associated with nodes in other sub-instance can be modeled by adding artificial isolated nodes with zero location and overlap costs.

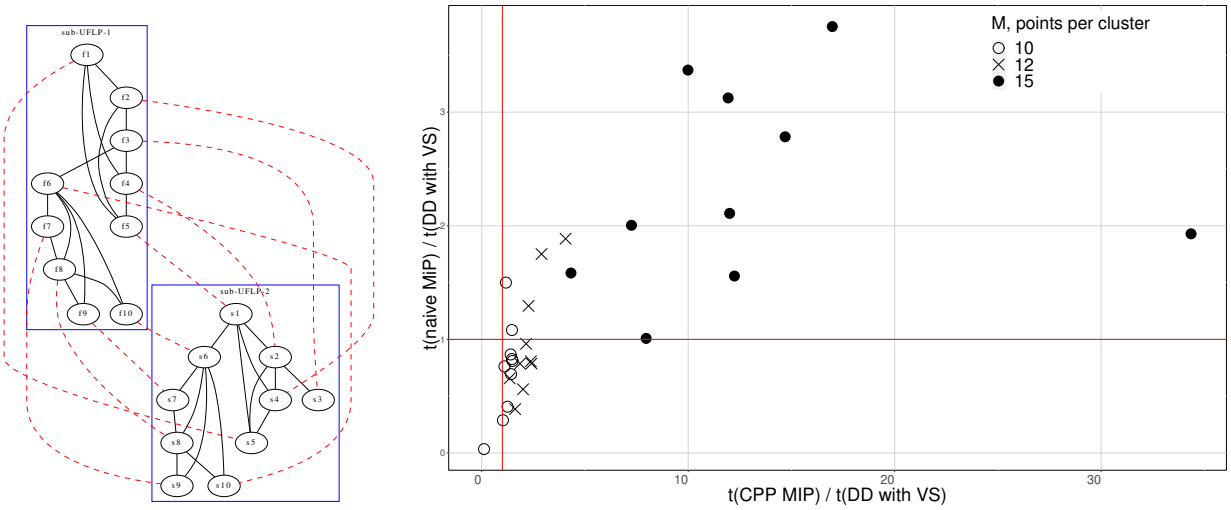
This problem can be reformulated as a CPP instance as follows. Each diagram encodes a single UFLP sub-instance, with layers corresponding to the location decisions x_i^s within the original graphs. Edge costs incorporate both location costs c_i^s and overlap costs $f_i^s(a_i^s)$, and layer labels are set to reflect linking constraints (2c). For example, given a constraint $x_3^1 = x_5^2$, the two layers encoding these linked decisions in different diagrams possess the same label "x_5". We can build each of the two diagrams based on the following ideas. First, we incorporate location cost c_i^s effectively into every one-arc emanating from layer x_i^s. Then, we incorporate overlap costs $f_i^s(a_i^s)$ into every arc of the first layer for which a_i^s is already determined, i.e., the first layer situated below all the layers corresponding to decisions x_j^s for all $j \in S_i^s$ in the diagram.

Each BDD node is associated with a *state* $\sigma = (\sigma_1, \ldots, \sigma_N)$, where component σ_j takes one of three possible values. If the location decision for point j affects any overlap costs associated with layers below the layer corresponding to j in the diagram, then σ_j takes values 0 or 1, depending on whether a facility was located at j or not, respectively. However, when this location is not yet determined, or when the decision ceases to affect any further costs, we set σ_j to a special value "*". When we build the diagram, at the root node we start from state $\sigma^{\mathbf{r}}$ with all components $\sigma_j^{\mathbf{r}}$ set to "*". As we create layers, each component j of the state will receive a value 0 or 1 in some BDD nodes. After we create the necessary layers, each state component will be marked again, and will keep the value "*" in all the subsequent nodes we will create. The terminal node will have state $\sigma^{\mathbf{T}} = \sigma^{\mathbf{r}}$ again (where all state components are marked with "*").

Note that the state component j can be reset from "0" or "1" to "*" only in a layer that is below all the other layers corresponding to the points in distance-two neighborhood of j. With the goal of building small BDDs, we use a simple greedy algorithm that adds the smallest possible number of points until a state component can be marked with "*" for the second time. We omit the algorithmic details for brevity; an implementation in Python along with the code documentation is available on the authors' source code repository (www.bochkarev.io/research). More details are provided in Appendix H.

Therefore, we encode each sub-instance with a BDD having its own variable order. When there are alternative orderings of layers implying the same diagram size, we use this freedom to minimize the number of inversions between the BDD orders at the diagram construction step. By setting the BDD labels according to the linking constraints (2c) as discussed above, the j-UFLP modeled by (2) is equivalent to a CPP instance over these two diagrams.

Having this representation, we consider four possible ways to solve the j-UFLP. Our baseline is a naive MIP representation (2), which we solve as is using Gurobi. Alternatively, we can reformulate this problem as a CPP and solve this reformulated problem in one of three ways. First, we can represent it as another MIP, having two shortest-path sub-problems linked through side constraints, as discussed in Appendix H.1. The other two strategies align the two diagrams, build an intersection BDD, and solve a single shortest-path problem. The strategies differ in how we align the diagrams: either using a naive heuristic (align to the order of the first diagram) or by using the proposed heuristic based on the simplified problem discussed in the first part of the paper.



(a) Example j-UFLP structure.

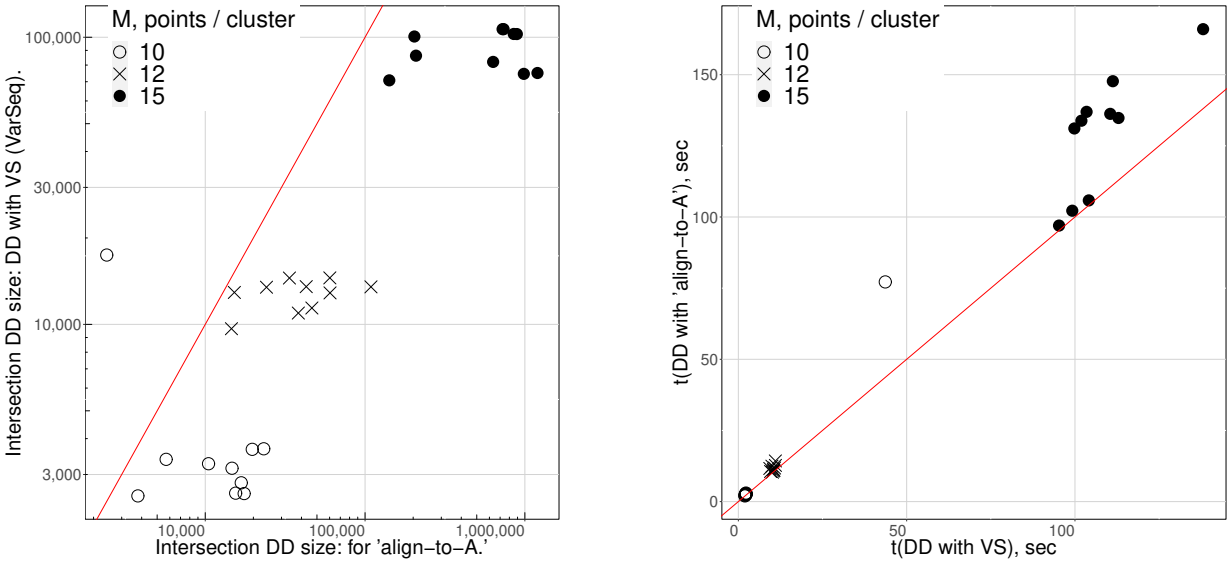
(b) Relative runtimes for different solution methods.

Figure 9: Numerical experiments summary with j-UFLP instances.

Relative runtimes of these four approaches will depend on the underlying graph structure. To provide an example where the proposed heuristic could be beneficial, we design our instances as follows. First, to limit the overall runtime, we assume each sub-instance comprises two components of equal number of nodes M, connected with a single edge. Then, we enforce a certain number of edges within each cluster. This setting corresponds to practical situations in which there are densely connected clusters of points within cities, or within buildings, with a limited connection between the clusters. Finally, to make sure the problem of aligning the diagrams is complex enough, we enforce a special type of linking: within each cluster, we link the points in the reverse order. Real-world networks might only partially reflect this kind of structure, but we deliberately design a complex case to highlight the capabilities of the heuristic we propose. Note that aligning one diagram to the other one would now imply reversing one of the two orders of variables (unless the inversions were already fixed during the BDD construction time). Specific edges are created at random, assuming a set number of edges within each cluster. Figure 9a shows an example instance having $M = 5$ points and $|A| = 7$ edges per cluster. Sub-instances are depicted with two boxes, and linking constraints are represented by dashed lines between the points of the two sub-instances.

We picked three different instance sizes, parameterized by different number of points per cluster $M = 10$, 12, or 15. We adjust the number of edges each time to enforce a fixed connection density, measured as the ratio of the number of edges in the cluster divided by the maximum possible number of edges in a cluster with the given number of nodes. Costs are generated randomly in a manner that promotes challenging instances; see Appendix F.2 for more details on instance generation. For each instance size we generated ten random instances and measured the runtime in seconds for each of the four alternative solution methods discussed above.

The results are summarized in Figures 9b and 10. (The details for each instance are presented in Table 3 in Appendix I.) Along the axes in Figure 9b we depict the runtimes measured in seconds for the alternative MIP-based formulations (a naive MIP, as modeled by (2), and a CPP solved as an MIP), relative to the proposed approach involving the simplified problem, marked



(a) Intersection BDD sizes.

(b) Runtimes for different alignment heuristics.

Figure 10: Comparison of different alignment heuristics.

"DD with VS" in the figure. Each point represents a j-UFLP instance of the size parameterized by M. Positions to the right of the vertical line (above the horizontal line) imply that CPP MIP approach (respectively, naive MIP approach) was slower than the proposed heuristic. Note that while a naive MIP finds a solution quickly for small instances, our aligned BDD approach becomes noticeably faster when the instance size grows. Moreover, solving the CPP directly with an MIP solver is generally slower than our proposed approach, for the type of instances we have generated.

Figure 10 highlights the comparison between the two BDD alignment heuristics: one based on the simplified problem (again, marked "DD with VS" in the figure) and a simple baseline of aligning to the first diagram ("align-to-A" in the figure). Observe that the intersection diagram generated by the proposed heuristic is almost always significantly smaller than the baseline: Most points in the left panel lay below the diagonal line (note that points along this line have the same values along both axes). As demonstrated in the right panel, the computational advantages of using our proposed heuristic only become apparent in larger instances, when the speed up due to the smaller intersection BDD size justifies the computational expense of the simplified-problem based heuristic.

5 Conclusions and future work

The key contribution of this paper 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 attains 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.

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 machine learning 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 paper can be improved by fine-tuning bound estimates, branching strategies, or picking a different principle of branching.

Several generalizations of the approach we discussed constitute potentially interesting further research directions. For example, one might consider minimizing the total amount of memory used to store the BDDs instead of the total number of nodes, which might be interesting if the diagrams are stored in a way that some nodes are shared. (The objective $|T^*[A,\vec{v}]| + |T^*[B,\vec{v}]|$ would be then substituted for $|T^*[A,\vec{v}] \cup T^*[B,\vec{v}]|$.) Also, the simplified problem might be designed for other types of diagrams, such as multi-valued decision diagrams or zero-suppressed decision diagrams.

References

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

Bergman D, Cire AA (2016) Decomposition based on decision diagrams. Quimper CG, ed., *Integration of AI and OR Techniques in Constraint Programming*, volume 9676, 45–54 (Springer International Publishing).

Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2016) *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms (Springer International Publishing).

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

Brace KS, Rudell RL, Bryant RE (1990) Efficient implementation of a BDD package. *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 40–45, DAC '90 (ACM).

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

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

Bryant RE (2018) Binary decision diagrams. Clarke EM, Henzinger TA, Veith H, Bloem R, eds., *Handbook of Model Checking*, 191–217 (Springer International Publishing).

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

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

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

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

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

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

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

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

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

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

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

ReVelle CS, Eiselt HA, Daskin MS (2008) A bibliography for some fundamental problem categories in discrete location science. *European Journal of Operational Research* 184(3):817–848.

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

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

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

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.

A Technical notes on BDD definitions and operations

All definitions presented for BDDs in Section 2 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.

Note that 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.

Next, we describe in detail an algorithm that implements our BDD manipulations (sift and swap) to illustrate the motivation behind the weighted variable sequence concept introduced in Section 3.2. 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 $d < s$. (Some of the material presented in Section 3.1 is reproduced here for the sake of completeness.)

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 $\text{var}(L^B_d)$ to $\text{var}(L^B_s)$. A key challenge is that after the sift-up operation, we must now create arcs that connect nodes in the layer corresponding to $\text{var}(L^B_{s-1})$ to nodes in the layer corresponding to $\text{var}(L^B_{s+1})$, which are now adjacent in $R(B)$. However, the destinations of these arcs depend on the choice of value for $\text{var}(L^B_s)$ in $R(B)$.

Example 3 *Consider the BDD depicted in Figure 11 and a sift-up operation with source variable x_4 and destination variable x_2. Consider node* 9 *in Figure 11: 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^B_d and L^B_{s-1} (inclusive) in the transformation, with the purpose of retaining the value of $\text{var}(L^B_s)$. The overall process is as follows.

1. $R(B)$ is the same as B through layers $1, \ldots, d$ and $s+1, \ldots, N+1$.
2. Two copies of B between layers L^B_d and L^B_{s-1} (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^B_i of B are now in layer $L^{R(B)}_{i+1}$ in $R(B)$. One-arcs from the nodes in $L^{R(B)}_d$ point to the corresponding nodes in the one-copy, zero-arcs to the nodes in the zero-copy.
3. Nodes in layer $L^{R(B)}_s$ are connected to $L^{R(B)}_{s+1}$ in $R(B)$ according to the corresponding choice of $\text{var}(L^B_s)$ (i.e., whether the node is in the one-copy or zero-copy of the network) and $\text{var}(L^B_{s-1})$, as detailed in Algorithm 2.

Example 4 *Figure 12 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 11). 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*.*

Algorithm 2 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).

The algorithm returns diagram $R(B)$ that has the correct variable order by construction, and B and $R(B)$ are equivalent since 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 2 might result in the creation of redundant node pairs, as defined in Remark 1. Continuing with the previous example, both one-arcs and zero-arcs of nodes $11'$, $10''$, $11''$ point to node 19 in Figure 12. Hence, the resulting diagram is not quasi-reduced. These three nodes can be merged.

Starting from a quasi-reduced diagram, we can preserve this property by a simple modification of the *swap* implementation, which 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

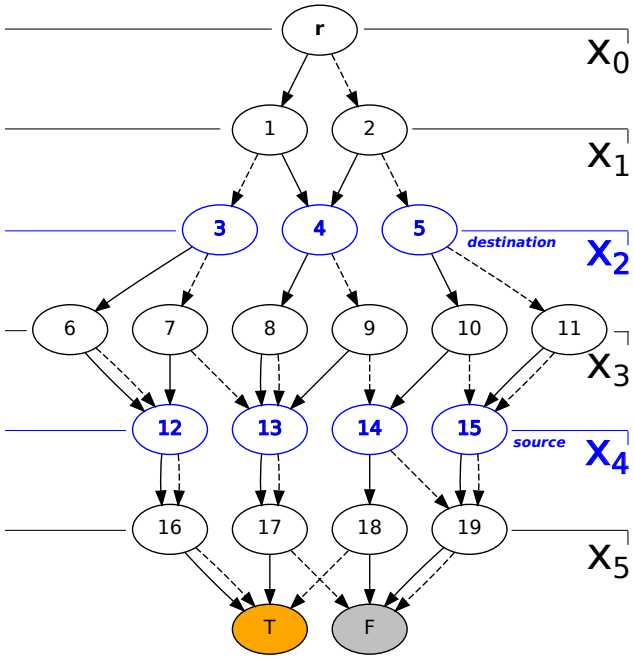


Figure 11: Original diagram B before sifting of the source layer x_4 to the position immediately after x_2.

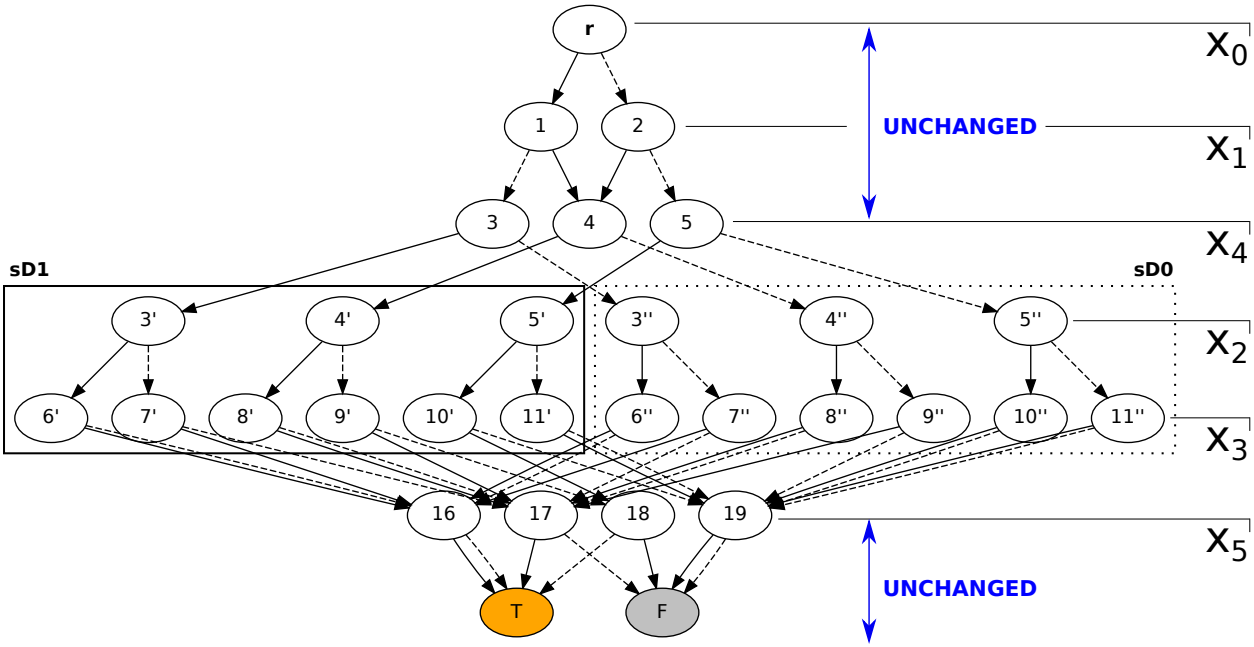


Figure 12: Resulting diagram transformation $R(B)$ after the sift in Figure 11.

Algorithm 3 (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). Similar to Algorithm 3, 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 13. 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.

Algorithm 3 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-LR(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
```

B Example: building an optimal transformation for a weighted variable sequence

A transformation of $S = [3, 5, 6, 11, 7, 9, 2, 10, 8, 4, 1 | n_1, \ldots, n_{11}]$ to $T^*_{\text{VS}}[S, (1, \ldots, 11)]$ is presented in Figure 14. 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 2, 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

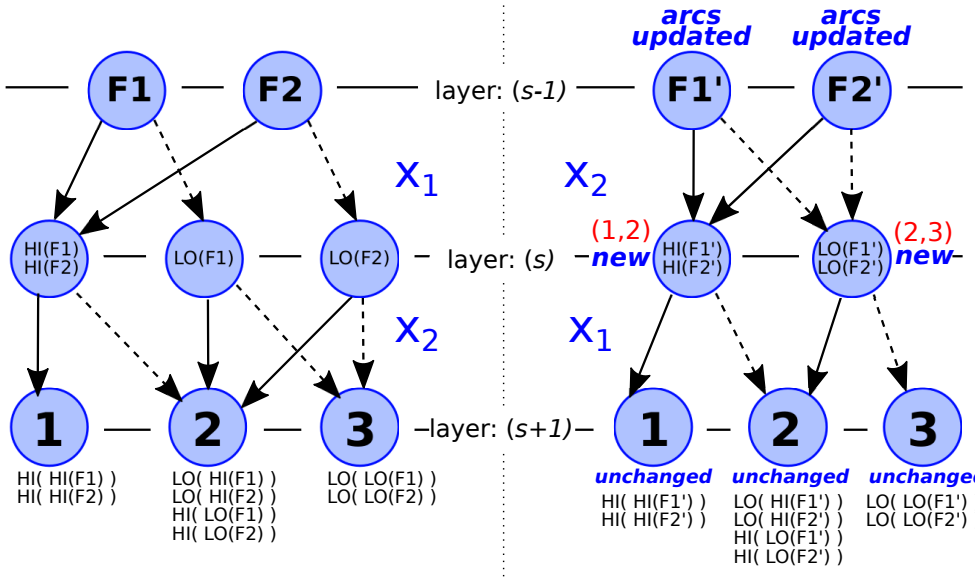


Figure 13: Illustration for Algorithm 3, swapping layers s and $(s-1)$.

Notes. 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.

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.

C A linear-time procedure to build the optimal transformation

In this appendix we present Algorithm 4, which builds the optimal transformation for a weighted variable sequence and target variable order.

As suggested by the example presented in Figure 14, 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 14, sinking elements are 3, 5, 6, and 11. This allows us to build a linear-time procedure as follows (illustrated in Figure 15). 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

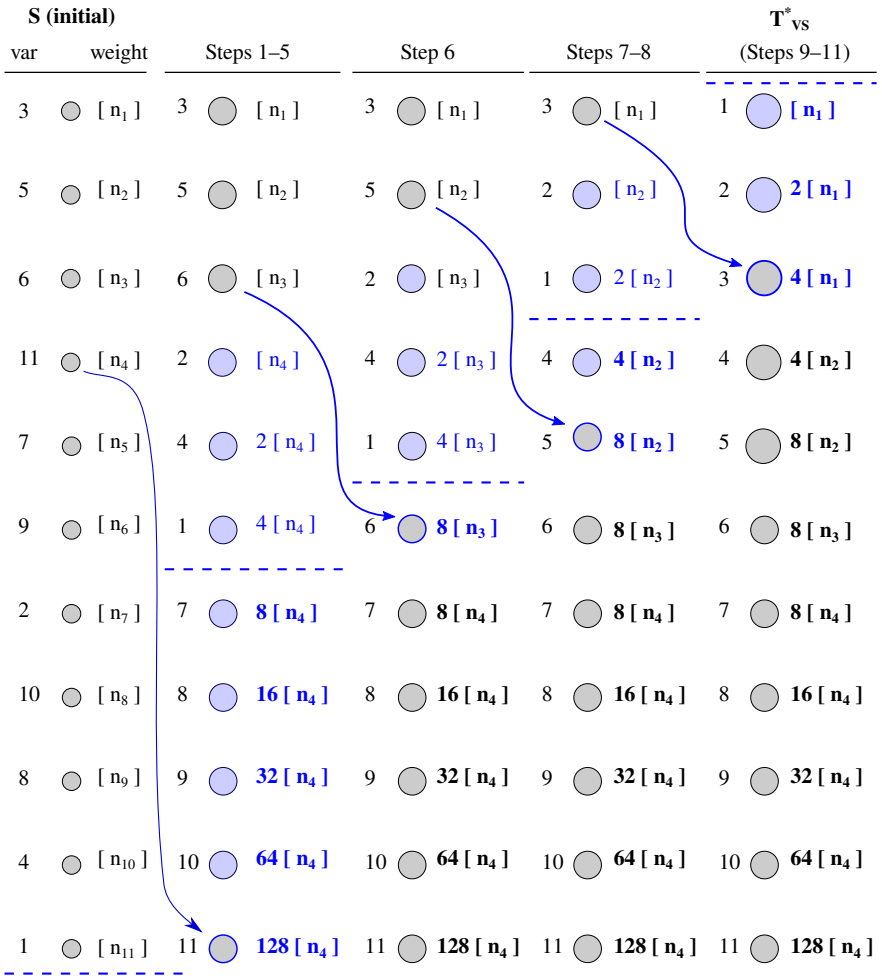


Figure 14: Building an optimal transformation of a weighted variable sequence with separate sifts. Circles are elements, with labels to the left, and weights to the right of them. Horizontal dashed lines represent N_f.

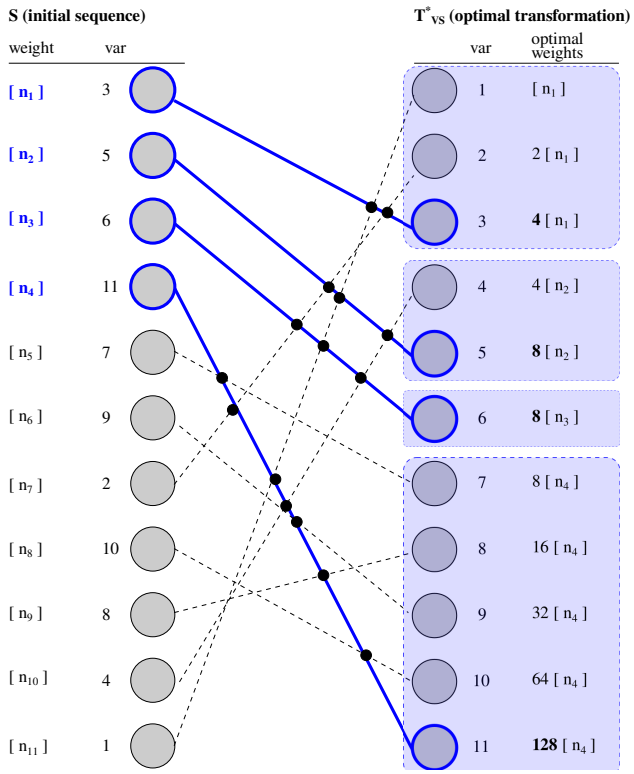


Figure 15: Building an optimal transformation of a variable sequence in one pass.

algorithm takes linear time. Algorithm 4 yields $T^*_{\mathrm{VS}}[S, \vec{v}]$, since it replicates weights that would be obtained by Algorithm 1.

D A branch-and-bound algorithm for the simplified problem

In this appendix we discuss the branch-and-bound procedure for the simplified problem, formally presented by Algorithm 5. A search tree node represents a pair of partially aligned sequences, $T^*_{\mathrm{VS}}[S_A, \vec{v^1}]$ and $T^*_{\mathrm{VS}}[S_B, \vec{v^2}]$, such that tails of $\vec{v^1}$ and $\vec{v^2}$ coincide (i.e., $v^1_i = v^2_i$ for all $i > N_f$ for some N_f). In lines 3–7 we initialize the search tree with a (trivial) root node with $\vec{v}^1 = \mathrm{var}(S_A)$ and $\vec{v}^2 = \mathrm{var}(S_B)$. 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 $\vec{w}$ and aligned sequences $\hat{S}_A = T^*_{\mathrm{VS}}[S_A, \vec{w}]$ and $\hat{S}_B = T^*_{\mathrm{VS}}[S_B, \vec{w}]$). 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, then we create a new node with this element moved to position N_f and decrease N_f by one, similar to Algorithm 1. Corollary 3 allows us to fix the relative order for some variables as we sift down, which we do

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

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

```
 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
```

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.

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 16. 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 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 4). 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

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

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

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` $\leftarrow$ **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 with labels from $C_{S_1,a}$ and $C_{S_2,a}$ (from S'_2 and S'_1, respectively) as per Corollary 3
19: `newNode` $\leftarrow$ **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` $\leftarrow |Q_A| + |Q_B|$
22: `node_LB` $\leftarrow$ `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 4.
48: **return** LB
49: **end function**

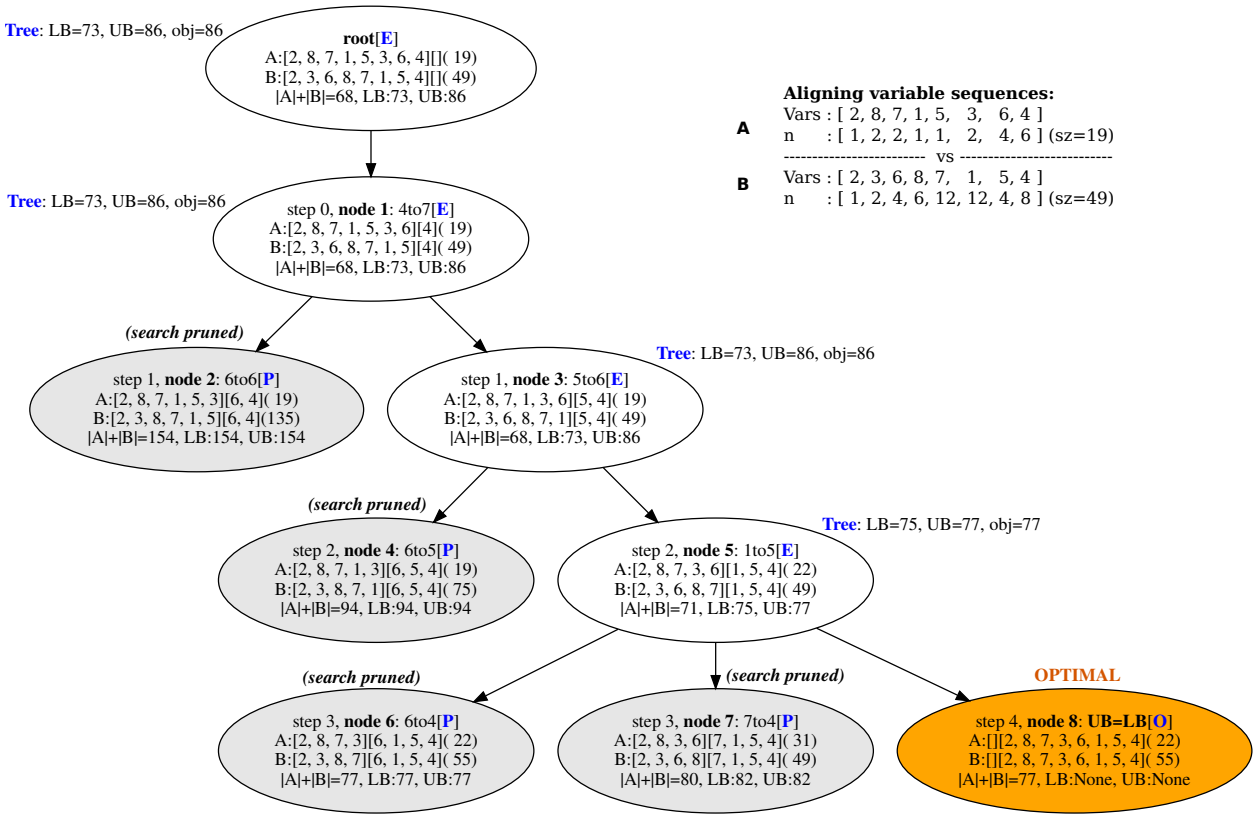


Figure 16: Example of the final search tree (simplified problem).

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`.

E Different 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 5. Given that $v_1 \in \{a_1, b_1\}$ 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 po-

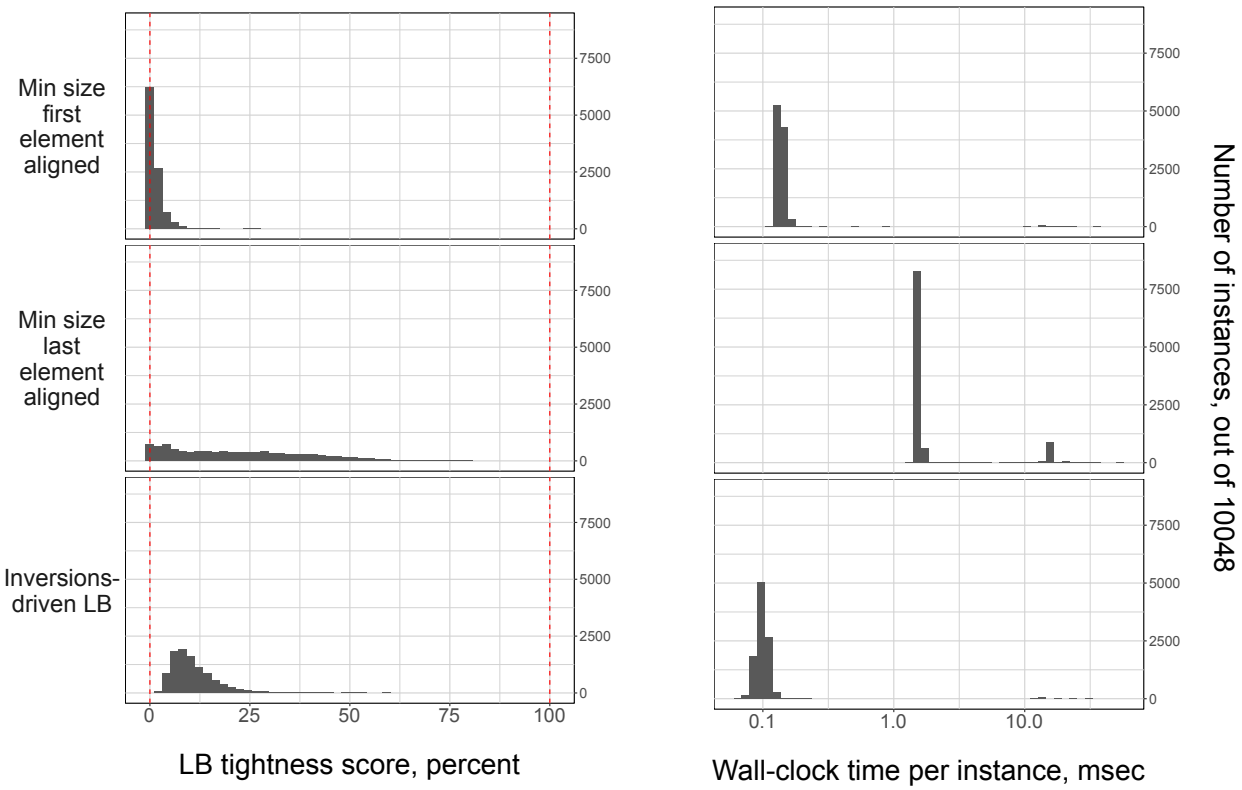


Figure 17: Benchmarking lower bounds for the simplified problem.

sition 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 4, 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 17. 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.

F Random generation of instances

There are two different types of test instances we generate within this paper for computational experiments: random align-BDD and joint uncapacitated facility location problem (j-UFLP).

F.1 Random align-BDD instances

To generate each instance, we define 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, **T** and **F**, are created, and all nodes in the previous layer are connected randomly to **T** or **F**. After the diagram 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). 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 an intermediate values of p, we would have a random diagram with layer sizes growing depending on p. Figure 18 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.

F.2 Random j-UFLP instances

To generate a random j-UFLP instance, we generate two UFLP sub-instances independently. A sub-instance is parameterized by the number of clusters n, number of points per cluster N, and the connection sparsity parameter L. To create a sub-instance, we first create the necessary number of clusters and ensure their connectivity (we connect each new point to a random one from an already connected component). Then, the total number of edges A in a cluster is determined by the sparsity parameter L and the number of nodes. In particular, we are adding edges between random pairs of nodes within the cluster until we obtain the smallest number of edges a satisfying:

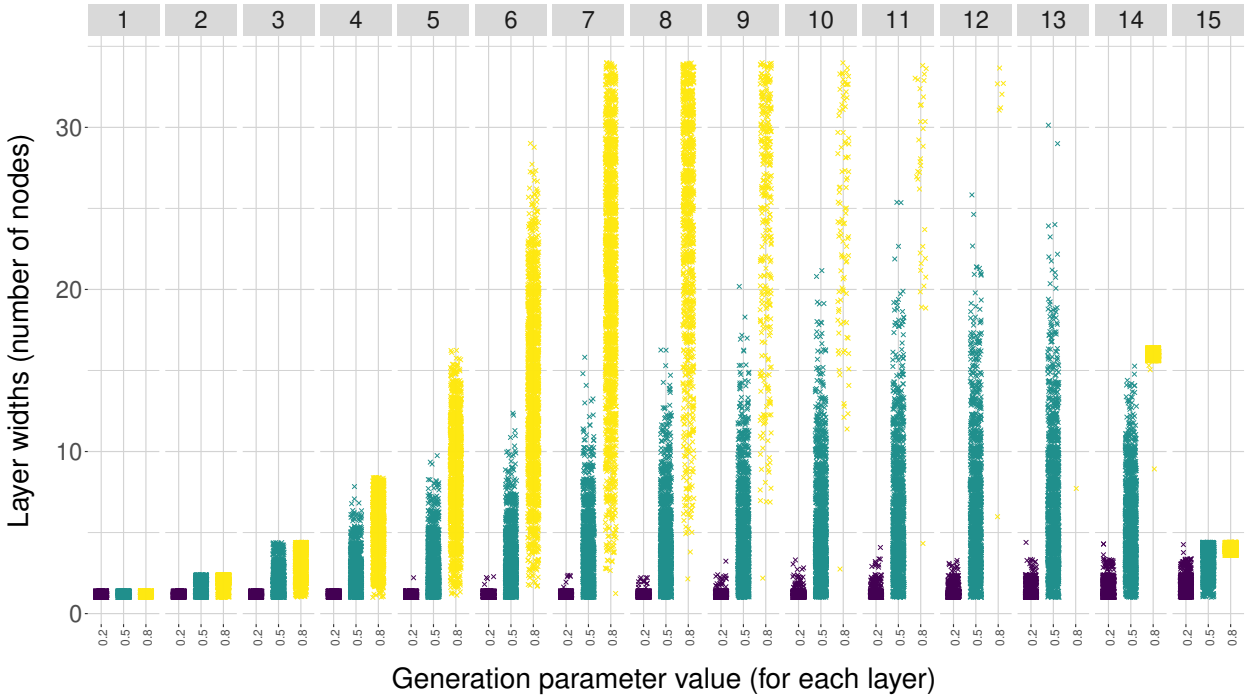


Figure 18: Layer widths for a set of 1,000 random quasi-reduced 15-variable binary decision diagrams for each value of p.

$$L \leq 1 - \frac{a}{n(n-1)/2}.$$

For example, for $n = 12$ points per cluster, we can have $n(n-1)/2 = 66$ edges for a fully connected subgraph. A threshold number of edges in a cluster a will be determined by:

$$L = 0.35 = 1 - \frac{a}{66},$$

which yields $a = 0.65 \times 66 \approx 43$ edges per cluster, and hence $43 \cdot 2 + 1 = 87$ edges per sub-instance and $87 \cdot 2 = 174$ edges per instance of $n = 2$ clusters.

Costs are generated as follows. First, location costs c_j^s are generated uniformly random from $[4.5,\ 5.5]$. Because every solution is feasible, we make sure that location costs are relatively low, to prevent trivial solutions, where it would be optimal not to locate any facilities. For the overlap costs, we assume $f_i(0) = 200$, which is a relatively large positive value, so that not covering a point would yield significant cost. $f_i(1) = 0$, so that covering a point exactly once does not yield any additional costs. We do not impose any further requirements: For example, $f_i(a_i)$ for $a_i > 1$ can be both positive (costs) or negative (benefits for redundant coverage), and not necessarily monotone in a. In particular, we generate the rest of overlap costs $f_i^s(a_i^s)$ for $a_i^s > 0$ uniformly random from $[-100, 100]$.

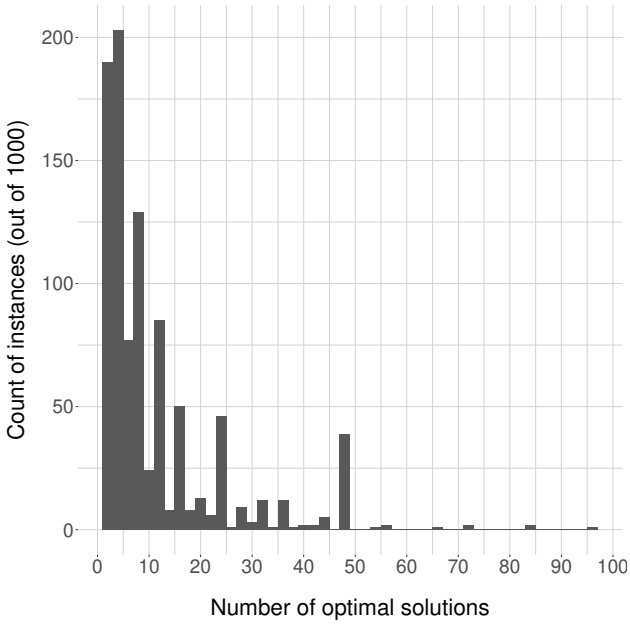


Figure 19: Histogram for the number of optima.

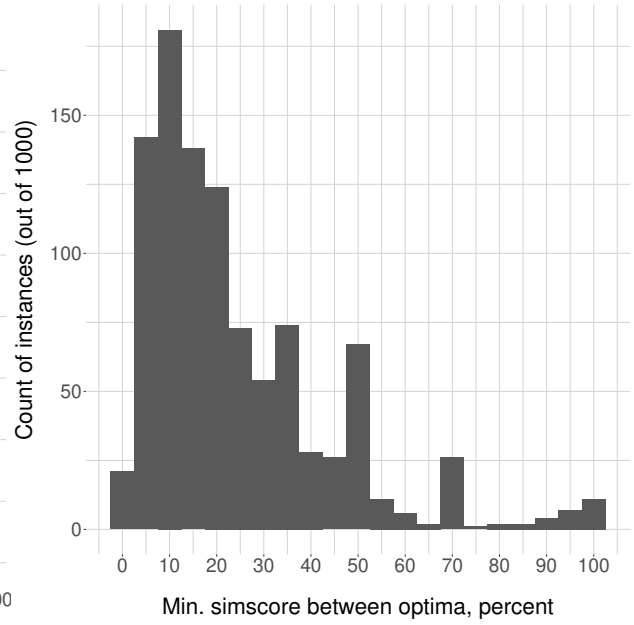


Figure 20: Histogram for the optimal set "diameter."

G Simplified problem solution structure

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

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

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 19–22. Note that there are usually several optima for our instances (e.g., out of 1,000 instances included in Figure 19, 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 20).

We see from Figure 21 that in more than half of the cases, the obtained solution 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

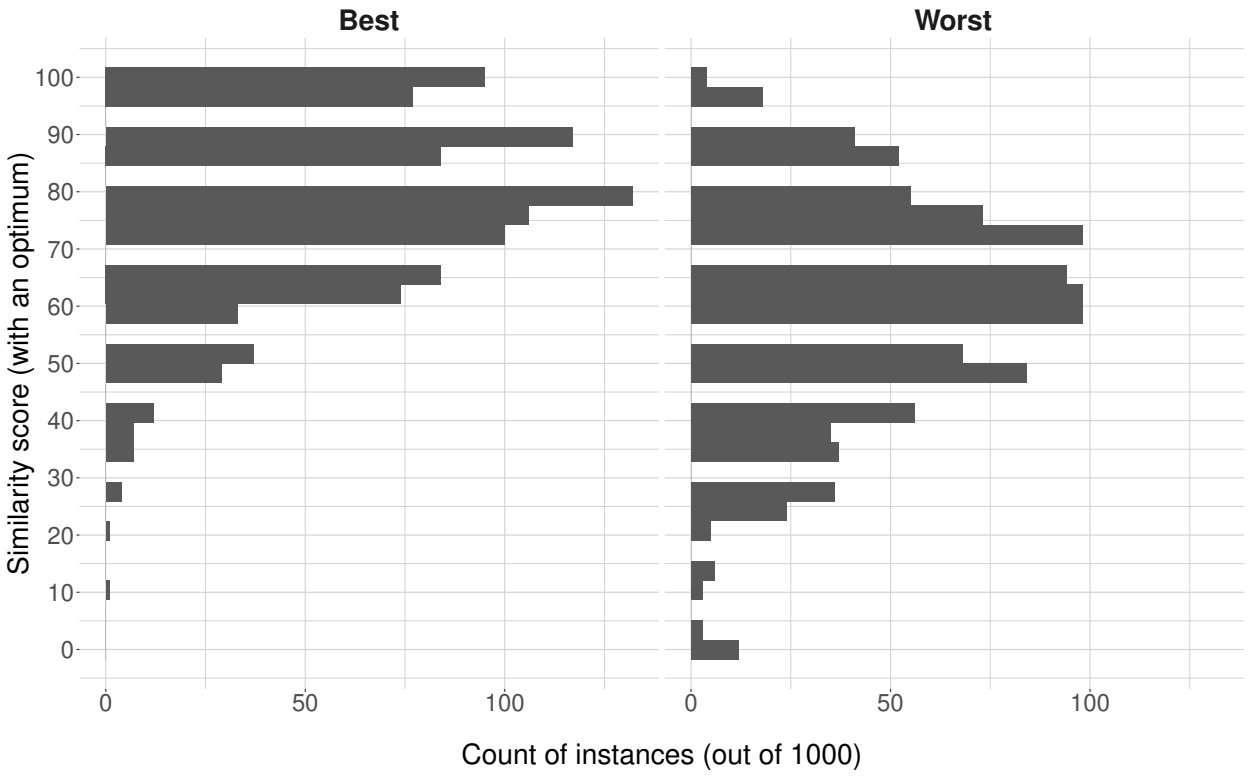


Figure 21: Similarity score between the solution obtained by the proposed heuristic and a true optimum: min and max values across all optima.

Figure 22.

H Implementation details for the j-UFLP

In this appendix we discuss specific implementation details for the j-UFLP, namely a procedure to create a CPP representation of problem (2) and a formulation of j-UFLP as a CPP MIP. We start with an algorithm to construct a BDD for a sub-instance assuming that the order of variables is already fixed. This allows us to later find a natural way to order variables and create small BDD representations.

First, consider a single UFLP instance, which is equivalent to the following MIP:

$$\min \sum_{i=1}^{N} \Big(c_i x_i + f_i(a_i) \Big) \tag{3a}$$

$$\text{s.t. } a_i = \sum_{j \in S_i} x_i \qquad \forall i = 1, \dots, N, \tag{3b}$$

$$x_i \in \{0, 1\} \qquad \forall i = 1, \dots, N, \tag{3c}$$

where S_i is a list of points adjacent to $\textcircled{i}$ (including itself, by assumption). We seek to build a BDD where every layer would represent a single facility location decision, x_i. Each node of the

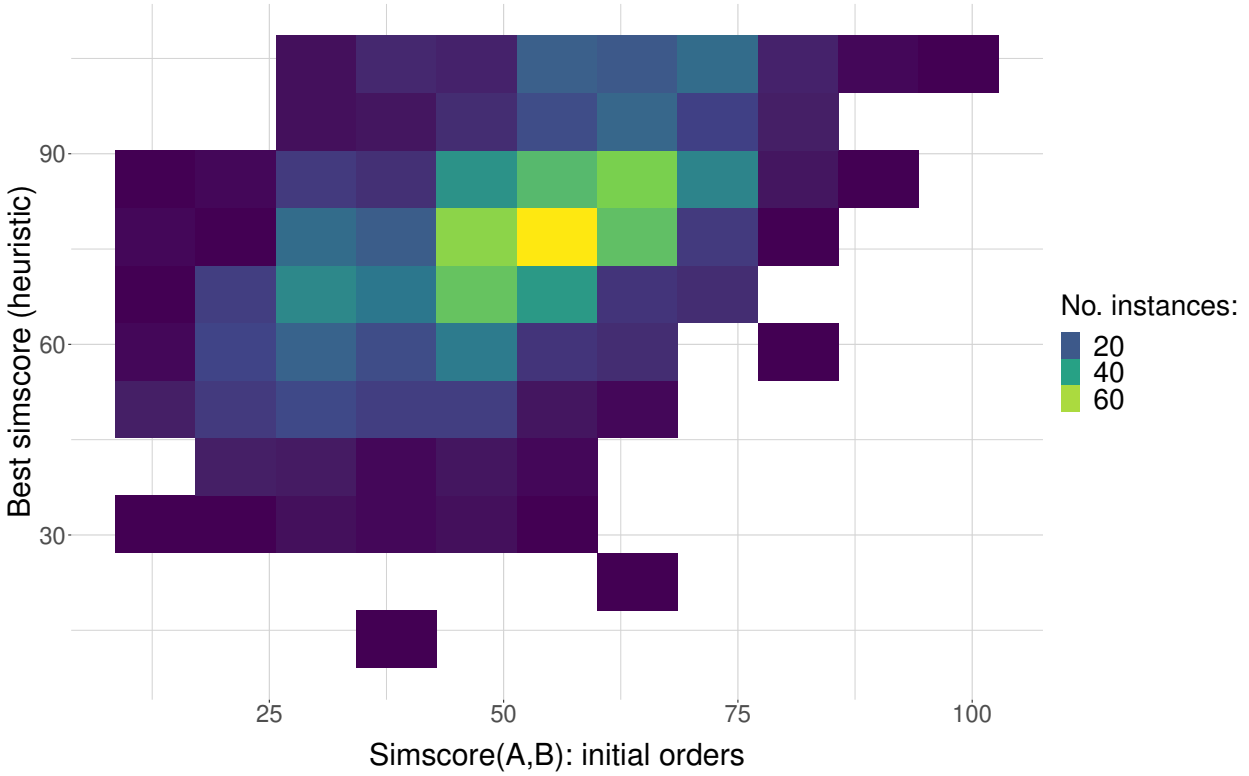


Figure 22: Dependence of the solution on the initial orders.

BDD will be associated with a *state* $\sigma = (\sigma_1, \ldots, \sigma_N)$, where $\sigma_i \in \{1, 0, *\}$ represents whether a facility has been located ($\sigma_i = 1$) at ⓘ or not ($\sigma_i = 0$). The third value, $\sigma_i = *$, for a variable that has already been added to the diagram during the construction process indicates that the location decision on point ⓘ does not affect costs further. Each node state is unique within the respective layer. As we incorporate more decisions and the corresponding costs to the diagram, some previous decisions become irrelevant and the layer width can both increase or decrease.

There are two key observations relevant to the BDD construction. First, we can **incorporate costs** associated with point i, $c_i x_i + f_i(a_i)$, as soon as we incorporate all decisions $x_j \in S_i$. For the situation depicted in Figure 23, we can calculate costs associated with ⓘ only when we incorporate all decisions regarding the light shaded points and ⓘ itself. After that we can derive x_i and a_i to compute $f_i(a_i)$ from each BDD node state. However, incorporating all points from S_i is not enough to discard the decision regarding point ⓘ (marking it with "*" in all further node states). For example, the cost associated with point ⓙ in Figure 23 depends on the decisions regarding points ⓘ, ⓙ, and all the points shaded with dark color. Therefore, we will have to keep the decision on ⓘ in the state until we incorporate all the points depicted in the figure (assuming no other points are present). Therefore, the second key observation regarding the BDD construction is that any point ⓘ can be marked with σ_i set to "*" (as "further irrelevant") as soon as we have incorporated the decisions regarding all points from the distance-two neighborhood of ⓘ into the diagram.

These two observations yield an algorithm to construct the diagram given an order of points to incorporate, and suggest a way to find a good order of decisions. Let us start with the former.

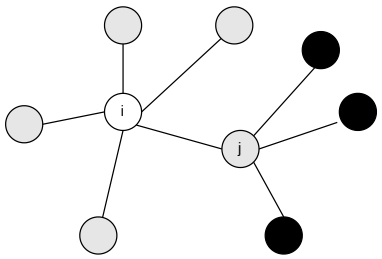


Figure 23: A few points of the original graph.

Algorithm 6 constructs the diagram encoding problem (3), assuming pre-defined order of variables O. For each point $\textcircled{i}$, we track two values. First, its *cost uncertainty* Q_i is the number of points we would need to incorporate into the diagram to be able to calculate the cost associated with $\textcircled{i}$. Second, its *value* V_i is the number of points which costs depend on the decision on point $\textcircled{i}$.

We create layers in the specified order one by one, in the main loop in lines 5–34. Every time we incorporate new point $\textcircled{i}$ into the diagram, we decrement Q_j for all points from its neighborhood S_i (line 8). After this step we can calculate costs associated with all points j such that $Q_j = 0$, we keep the set of such points in variable P (line 9). Further, each time we calculate cost for some point $\textcircled{j}$ we can decrement V_k for all points k from its neighborhood S_j (line 10).

We create each next layer by looping over the set of nodes in the current layer in lines 17–32. We associate nonzero costs to arcs every time a cost uncertainty Q_j reaches zero for some point $\textcircled{j}$, by calculating costs for all points in P (in lines 22–23 and 29–30 depending on the arc type). Also, every time some point value V_k drops to zero, we mark such point as further irrelevant by setting the corresponding component of the state σ_k to "*" (in line 18).

This procedure provides some insight into the size of the resulting diagram. Observe that the layer width increases by a factor of two every time we incorporate a new point, and decreases by a factor of two every time we mark a point as irrelevant. This allows us to claim that the $k+1$-th layer width W_k can be expressed via the previous layer width as follows:

$$W_{k+1} = W_k \times 2^{1-F_k} = 2^{k-\sum_{j=1}^{k} F_j},$$

where F_j is the number of points marked as further irrelevant after step j of Algorithm 6. Therefore, the logarithm of width will look like the one presented in Figure 24. The top line represents step numbers k, the bottom one represents cumulative number of nodes marked irrelevant. Both lines start at $(0, 0)$ and end at (N, N), and $\sum_{j=1}^{k} F_j \leq k$, because at each step only one point is added, and hence no more than k points can be marked further irrelevant after k steps.

This representation suggests a simple greedy algorithm to find a good variable order to yield a small BDD representation. Such algorithm would focus on marking the next node further irrelevant as fast as possible, creating each next layer by incorporating a point from a smallest set $S_j^* \setminus X$, where S_j^* is the distance-two neighborhood of point $\textcircled{j}$, and X is the set of points already added to the BDD. We illustrate this procedure with Example 5.

Example 5 *Consider a graph depicted in Figure 26. We summarize the steps we take in Table*

Algorithm 6 create-UFLP-BDD

Input: Graph G, given by $S_i, i = 1, \ldots, N$; costs $f_i(\cdot), c_i$ for $i = 1, \ldots, N$; points order O.
Output: BDD encoding the UFLP sub-instance.

$Q_j \leftarrow |S_j|$ for all $j = 1, \ldots, N$ *// Costs "uncertainty" for each node*
$V_j \leftarrow |S_j^*|$ for all $j = 1, \ldots, N$ *// Node "value".*
`next-layer` $\leftarrow \{(*, \ldots, *)\}$ *// Root node only.*
$k \leftarrow 1$ *// Current layer number.*
while $k < N$ **do**
    $i \leftarrow Q_k$ *// The next point being incorporated into the diagram.*
    $P \leftarrow \varnothing$ *// Set of points to calculate costs for.*
    decrement Q_j for all $j \in S_i$
    $P \leftarrow P \cup \{j : \ Q_j = 0, \ j \in S_i\}$
    decrement V_j for all $j \in \cup_{t \in P} S_t$
    `current-layer` $\leftarrow$ **copy**(`next-layer`)
    if $k = N$ **then**
        `next-layer` $\leftarrow \{(*, \ldots, *)\}$ *// True terminal.*
    else
        `next-layer` $\leftarrow \varnothing$.
    end if
    for $\sigma = (\sigma_1, \ldots, \sigma_N) \in$ `current-layer` **do**
        set `next-state`: $s_k \leftarrow *$ if $V_k = 0$, and σ_k otherwise for $k = 1, \ldots, N$.
        if `next-state` $\notin$ `next-layer` **then**
            create node `next-state` in `next-layer`
        end if
        calculate $a_j \leftarrow \sum_{k \in S_j} s_k$ for all $j \in P$
        calculate arc cost $C \leftarrow \sum_{j \in P} \left(c_j s_j + f_j(a_j) \right)$
        add arc `LO(`σ`)` $\leftarrow$ `next-state` (with cost C)
        update `next-state`: $s_i \leftarrow 1$ if $V_i \neq 0$.
        if `next-state` $\notin$ `next-layer` **then**
            create node `next-state` in `next-layer`
        end if
        calculate $a_j \leftarrow \sum_{k \in S_j} s_k$ for all $j \in P$
        calculate arc cost $C \leftarrow \sum_{j \in P} \left(c_j s_j + f_j(a_j) \right)$
        add arc `HI(`σ`)` $\leftarrow$ `next-state` (with cost C)
    end for
    $k \leftarrow k + 1$
end while

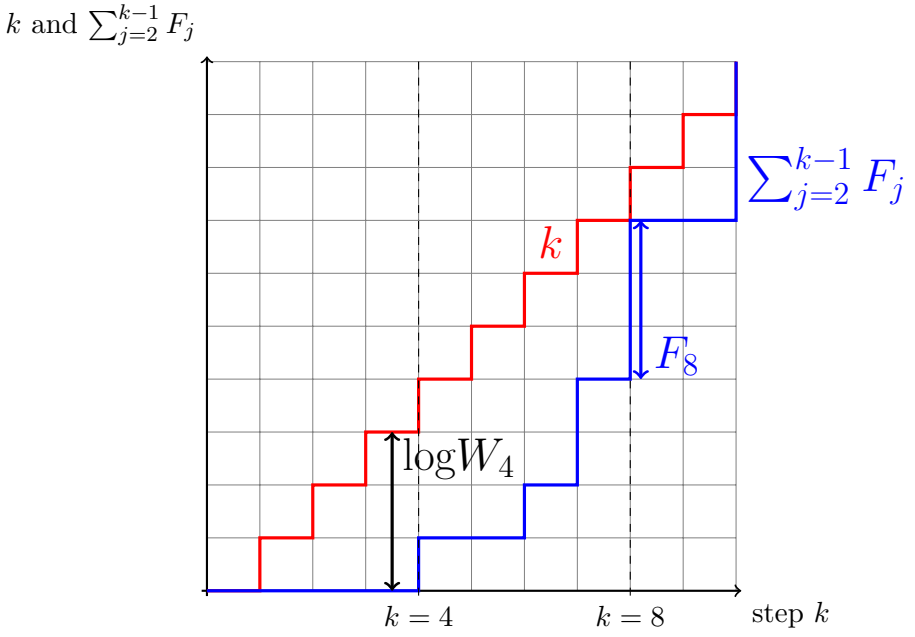


Figure 24: An illustration for the layer width.

2. Rows represent distance-two neighborhoods of respective points. When we incorporate a point (add it to the BDD), we remove it from further consideration by crossing it out from all the rows of the third column (S_i^*) of the table. We keep track of the number of points left in each subset in the right-most five columns of the table. The algorithm runs as follows:

- *The smallest S_i^* corresponds to point 11 (with the single element, 11). Therefore, we add point 11 and mark it irrelevant immediately.*
- *The next candidate subset is S_1^* (of the smallest size 4, ignoring point 11 now). Hence we add points 1, 2, 3, and 4, and mark ① irrelevant.*
- *We update the residual sizes of the distance-two neighborhoods (without elements 1, 2, 3, 4, in addition to 11) in column $k = 5$ and observe that ③ can be marked irrelevant as well (points 1,2,3, and 4 are already added).*
- *The next candidate subset to complete (corresponding to a smallest number in column $k = 5$) is ②. We add points 5 and 6, marking ② as irrelevant.*
- *The next candidate subset corresponds to a smallest number in column $k = 8$. This would be the ones corresponding to ④, ⑤, or ⑥. The first one is ④, so we proceed with incorporating points 7 and 10 into the diagram and marking ④ as irrelevant.*
- *The next candidates correspond to smallest numbers in column $k = 11$, which are rows 5 or 6. Either way we incorporate ⑧ and mark both 5 and 6 as irrelevant. Updated residual distance-two neighborhoods are presented in column $k = 12$.*
- *Finally, we add the last point, ⑨ and mark 7, 8, 9, and 10 as irrelevant, concluding the search for the best order.*

This procedure results in the following solution: `11 | 1, 2, 3, 4 | 5, 6 | 7, 10 | 8 | 9`*, where the points between the bars can be re-arranged at no additional cost in terms of the BDD*

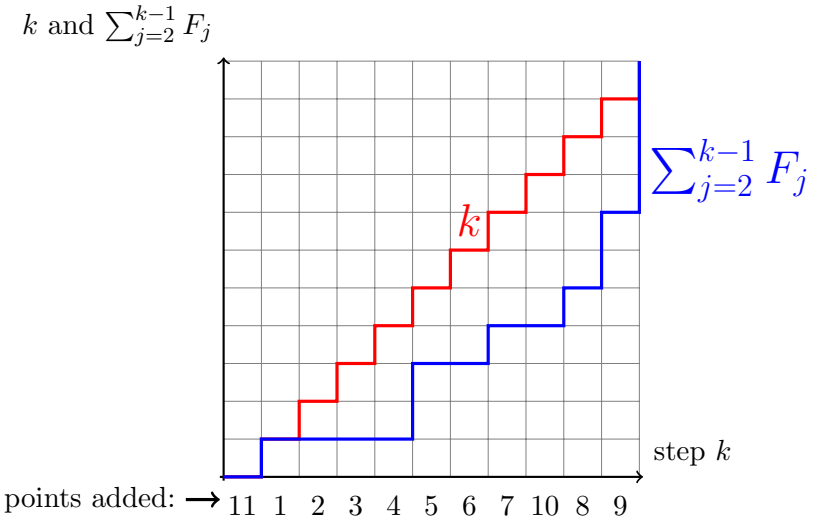


Figure 25: Layer width diagram for the proposed solution.

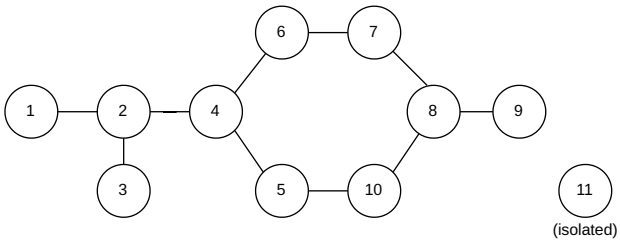


Figure 26: A sample graph.

size. We use this freedom during the BDD construction to minimize the number of inversions between the BDDs we generate (i.e., before we compare our algorithms, therefore, giving the benefit of doubt to the baseline alignment heuristic).

The corresponding layer width diagram (similar to Figure 24) is presented in Figure 25.

H.1 Solving the CPP

Having two diagrams encoding j-UFLP allows us to represent it as a CPP, which in turn can be formulated as an MIP as follows. We introduce (continuous) variables v_a^s to denote the flow along arc a of the diagram associated with sub-instance s (the set $\mathcal{A}^s$ assumes to comprise all such arcs), and binary variables x_i to correspond to the decisions of the original j-UFLP. Note that we omit the sub-instance index s here, because the layer labels already incorporate the linking constraints.

Using this notation, (2) can be reformulated as:

Table 2: Example: variable order for the BDD encoding UFLP.

ⓘ	S_i	S_i^*	$\lvert S_i^*\rvert$	$k=1$	$k=5$	$k=8$	$k=11$	$k=12$
1	{1,2}	{1,2,3,4}	4	4	X	X	X	X
2	{1,2,3}	{1,2,3,4,5,6}	6	6	2	X	X	X
3	{2,3}	{1,2,3,4}	4	4	X	X	X	X
4	{2,4,5,6}	{1,2,3,4,5,6,7,10}	8	8	4	2	X	X
5	{4,5,10}	{2,4,5,6,8,10}	6	6	4	2	1	X
6	{4,6,7}	{2,4,5,6,7,8}	6	6	4	2	1	X
7	{6,7,8}	{4,6,7,8,9,10}	6	6	5	4	2	1
8	{7,8,9,10}	{5,6,7,8,9,10}	6	6	6	4	2	1
9	{8,9}	{7,8,9,10}	4	4	4	4	2	1
10	{5, 8, 10}	{4,5,7,8,9,10}	6	6	5	4	2	1
11	{11}	{11}	1	X	X	X	X	X

Notes. The UFLP corresponds to the graph structure depicted in Figure 26. Underlined are changes as compared to the previous number.

$$\min \sum_{a\in\mathcal{A}^1} C_a^1 v_a^1 + \sum_{a\in\mathcal{A}^2} C_a^2 v_a^2, \tag{4a}$$

$$\text{s.t.} \sum_{a:\text{head}(a)=u} v_a^s = \sum_{b:\text{tail}(b)=u} v_b^s \qquad \forall u \in L_2^s \cup \ldots \cup L_{(N-1)}^s, \text{ and } s=1,2 \tag{4b}$$

$$\sum_{a:\text{tail}(a)=\mathbf{r}} v_a^s = 1, \qquad s=1,2, \tag{4c}$$

$$\sum_{a:\text{head}(a)=\mathbf{T}} v_a^s = 1, \qquad s=1,2, \tag{4d}$$

$$\sum_{a\in Y^s(x_q)} v_a^s = x_q, \qquad \forall q=1,\ldots,N, \text{ and } s=1,2 \tag{4e}$$

$$v_a^s \geq 0 \qquad \forall a\in\mathcal{A}^s, \text{ and } s=1,2, \tag{4f}$$

$$x_q \in \{0,1\} \qquad \forall q=1,\ldots,N, \tag{4g}$$

where we denote $Y^s(x_q)$ to be the set of all yes-arcs in the diagram associated with sub-instance s, emanating from the layer associated with decision x_q.

The problem comprises two standard shortest-path formulations (for $s=1$ and 2) with the usual objective function (given arc costs C_a^1 and C_a^2), flow continuity constraints at every node (4b), border conditions (4c) and (4d) for the root and terminal nodes, respectively, with added linking conditions (4e).

I Numerical experiments for the j-UFLP.

We picked three different instance sizes, parameterized by different number of points per cluster $M = 10$, 12, or 15. We adjust the number of edges each time to enforce a fixed connections density: the share of a possible number of edges actually present in the graph. Therefore, each instance is characterized in a table by a unique instance ID (**Inst ID**), number of points

Table 3: Summary of the numerical experiments results for the j-UFLP.

Exp #	Inst ID	M	N_0	$\|A\|$	t MIP	t CPP-MIP	t DD VS	t DD toA	int DD VS	int DD toA
1	1	10	40	122	1.96	3.52	2.26	3.00	3,155	18,235
2	4	10	40	122	1.91	3.38	2.31	2.64	3,671	14,547
3	7	10	40	122	1.48	2.47	43.65	77.19	17,439	3,379
4	10	10	40	122	3.30	2.87	2.20	2.29	3,388	6,868
5	13	10	40	122	1.58	2.97	1.95	2.00	2,527	4,255
6	16	10	40	122	0.86	2.57	2.11	2.71	2,583	17,839
7	19	10	40	122	0.55	2.33	1.91	2.24	2,807	13,879
8	22	10	40	122	1.70	2.82	2.23	2.55	3,271	15,109
9	25	10	40	122	1.52	3.14	2.19	2.72	2,575	16,959
10	28	10	40	122	2.50	3.04	2.31	2.63	3,687	14,697
11	2	12	48	174	6.65	14.98	10.13	10.33	12,919	18,975
12	5	12	48	174	17.61	37.53	9.34	11.64	9,659	15,707
13	8	12	48	174	5.37	19.33	9.58	10.58	10,951	45,151
14	11	12	48	174	17.46	28.07	9.97	12.44	11,403	72,763
15	14	12	48	174	8.47	24.63	10.76	11.80	12,871	55,799
16	17	12	48	174	4.09	18.89	10.58	10.97	13,471	27,719
17	20	12	48	174	8.66	19.03	11.01	12.72	13,511	75,325
18	23	12	48	174	8.88	27.55	11.00	14.28	14,519	85,071
19	26	12	48	174	10.00	23.69	10.40	10.77	13,535	27,759
20	29	12	48	174	13.49	23.13	10.42	11.35	14,503	43,063
21	3	15	60	278	291.23	1,660.12	138.07	166.00	102,606	1,081,439
22	6	15	60	278	104.97	813.31	104.11	105.83	100,599	137,335
23	9	15	60	278	204.27	736.91	101.91	133.83	82,061	753,727
24	12	15	60	278	175.85	1,376.98	112.93	134.79	106,707	792,313
25	15	15	60	278	388.31	1,773.06	103.45	136.95	75,071	772,270
26	18	15	60	278	265.17	1,410.01	95.27	96.99	70,779	99,339
27	21	15	60	278	213.13	3,799.03	110.48	136.24	106,695	651,351
28	24	15	60	278	176.23	467.19	111.25	147.71	102,667	983,145
29	27	15	60	278	310.03	1,164.51	99.20	102.21	86,263	159,879
30	30	15	60	278	336.43	1,016.69	99.82	131.11	74,559	798,899

per cluster M, number of points in the whole instance N_0 (which is always $4M$ due to the special structure we impose), and the total number of edges $|A|$. For each instance size we generated a few random instances and measured the runtime in seconds for each of the four alternative solution methods: a naive MIP (**t MIP**), a CPP solved as an MIP (**t CPP-MIP**), and the proposed pipeline involving formulating a CPP, building an intersection BDD, and solving a shortest-path: **t DD VS** for the case when the proposed heuristic was used to align the diagrams, and **t DD toA** for the naive baseline of aligning to the first diagram. For the methods involving the intersection diagram, we also keep track of the resulting diagram size (number of nodes), **int DD VS** for the diagram yielded by the proposed heuristic and **int DD toA** for the baseline alignment procedure. Note that all the runtimes comprise full pipelines, needed to process the (same) input data and find the optimal objective value.