UCLouvain

epl

École polytechnique de Louvain

Discrete Optimization with Decision Diagrams

Author: **Vianney Coppé**
Supervisor: **Pierre Schaus**
Readers: **Xavier Gillard, Hélène Verhaeghe**
Academic year 2018–2019
Master [120] in Computer Science and Engineering

Acknowledgments

I would like to thank my supervisor Professor Pierre Schaus for his guidance throughout the year. With weekly meetings even when I was abroad, he really helped me set milestones to this thesis, which allowed me to stay in a positive and productive attitude. Moreover, he gave me precious feedback on my intermediate results and drafts.

I would also like to thank my family for the caring and supporting environment, with a special thanks to my father for taking the time to read and comment this document.

Contents

Introduction **3**

1 Decision Diagrams **5**
1.1 Discrete Optimization Problems 5
1.2 Exact Decision Diagrams 6
1.3 Dynamic Programming Formulation 8
1.4 Compiling Exact Decision Diagrams 9
1.5 Relaxed Decision Diagrams 10
1.6 Restricted Decision Diagrams 12
1.7 Branch-and-Bound Algorithm 13
1.8 Heuristic components 18

2 Example Problems **19**
2.1 Maximum Independent Set Problem 19
2.2 Maximum Cut Problem 21
2.3 Maximum 2-Satisfiability Problem 23

3 Minimum Linear Arrangement Problem **25**
3.1 Introduction 25
3.2 Dynamic Programming Formulation 26
3.3 Relaxation Formulation 32
3.4 Computational Results 34

4 Implementation **36**
4.1 Overview 36
4.2 Implementing A New Problem 40

Conclusion **42**

References **43**

Introduction

Discrete optimization problems are everywhere nowadays, from delivery routes planning and flight crews scheduling to choosing the location of a new set of warehouses. Although mathematical programming or constraint programming solvers are very successful for a wide range of these problems, others involve cost functions and constraints which are difficult to formulate with traditional techniques, or contain a combinatorial structure that these techniques currently do not benefit from. A new general-purpose optimization method based on *decision diagrams* was presented in [7, 8], featuring an original modeling approach, a procedure providing tight bounds to the objective function and an adapted branch-and-bound algorithm. These decision diagrams are derived from *binary decision diagrams*, a mathematical tool originally introduced for the representation of Boolean functions [3, 23] and successfully used for circuit design and formal verification [9, 20]. They were then recently applied to constraint programming and optimization [5, 6, 15, 16, 17, 22, 27] and are promising to enhance or even replace existing methods. Note however that this technique relies on a dynamic programming formulation (see Section 1.3) and is therefore not suitable for any given problem.

A decision diagram is a graphical representation of the feasible solutions of a problem, it is composed of nodes connected by arcs. An arc encodes the assignment of a value to a variable of the problem and is associated with a weight, which is the contribution of this transition to the objective function. As a result, if these transitions are correctly designed for a specific problem, an optimal solution will be given by a longest-path in the corresponding decision diagram. In Chapter 1, we will summarize the theoretical concepts presented in [7, 8] to build a solver based on decision diagrams : we first define formally what decision diagrams are, then we introduce *restricted* and *relaxed* decision diagrams which respectively provide feasible solutions (lower bounds) and solutions to a relaxed problem (upper bounds) and finally, we explain how those tools can be used in an adapted branch-and-bound algorithm.

Three well-known problems have been successfully modeled with decision diagrams in [7, 8] : the *maximum independent set problem*, the *maximum cut problem* and the *maximum 2-satisfiability problem*. After showing how these problems were formalized in Chapter 2, we will try to apply this technique to a new problem : the *minimum linear arrangement problem* (MinLA), which consists of ordering a set of vertices on a line as to minimize the total length of the edges. Formally, for a graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ and E a set of weighted edges, where w_{ij} is the weight of the edge connecting

vertices i and j, the objective function is given by

$$LA_\pi(G) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j>i}}^{n} w_{ij} |\pi(i) - \pi(j)|$$

where the bijection $\pi : V \to \{1, 2, \ldots, n\}$ is the *linear arrangement*. This problem has many applications including error-correcting codes, wire length minimization in VLSI design and single machine job scheduling [1, 2, 12, 18, 26]. MinLA is NP-complete [13, problem GT42] and there has been a lot of research made on heuristic techniques designed to find good bounds for this problem [11, 19, 21, 25]. However, we are here interested in finding the exact solution to the problem so we will compare our approach with another exact method : a $0-1$ mixed linear programming model introduced in [4]. We present in Chapter 3 a new formulation for the MinLA using decision diagrams and show that it solves instances with 10 to 20 vertices under 1 hour, outperforming the linear programming model. As it is explained in sections 1.5 and 1.6, the size of relaxed and restricted decision diagrams can be controlled during their generation. We will study the influence of this parameter on the quality and efficiency of our bounding procedure.

The second objective of this thesis is to build a generic solver for discrete optimization with decision diagrams. It translates the theory presented in [7, 8] and summarized in Chapter 1 into a Java library allowing to solve new problems with simple and limited amount of code given a correct decision diagram formulation as described in Section 1.3. We used it to implement and evaluate our MinLA formulation and also included the maximum independent set problem, the maximum cut problem and the maximum 2-satisfiability problem as examples. The complete source code can be found on GitHub at the following link : **https://github.com/vcoppe/mdd-solver**. An overview of the implementation is given in Chapter 4 as well as a short example of usage of the library. Finally, we conclude this thesis by a summary and discuss possible improvements both to the MinLA formulation and to the Java solver.

Chapter 1

Decision Diagrams

In this chapter, we present the *decision diagram* representation of discrete optimization problems and introduce the notations we will use to formalize specific problems. We also explain how we can benefit from *relaxed* and *restricted* decision diagrams in the context of an alternative branch-and-bound scheme. This introduction is based on the theory developed in [7, 8] and follows a similar structure.

1.1 Discrete Optimization Problems

A *discrete optimization problem* $\mathscr{P}$ is defined as

$$\max\{f(x) \mid x \in D, \mathscr{C}\}$$

where :

- $x = (x_1, x_2, \ldots, x_n)$ is a tuple of n variables
- $D = D(x_1) \times D(x_2) \times \ldots \times D(x_n)$, with $x_j \in D(x_j)$ for each j
- $\mathscr{C} = \{C_1, C_2, \ldots, C_m\}$ is a set of m constraints on x
- $f : D \to \mathbb{R}$ is the objective function we want to maximize (without loss of generality as $\min\{f(x) \mid x \in D, \mathscr{C}\} = -\max\{-f(x) \mid x \in D, \mathscr{C}\}$).

A *feasible solution* of $\mathscr{P}$ is a complete assignment of the variables in x such that $x \in D$ and all the constraints in $\mathscr{C}$ are satisfied, $\mathrm{Sol}(\mathscr{P})$ is the set of all feasible solutions. An *optimal solution* x^* for $\mathscr{P}$ is a feasible solution which verifies $f(x^*) \geq f(x)$, $\forall x \in \mathrm{Sol}(\mathscr{P})$.

Example 1.1.1. A classical example of discrete optimization problem is the *Unbounded Knapsack Problem* where we have n items with a weight and a value. The goal is to find the combination of items with the largest value without exceeding the capacity of the knapsack. An instance of this problem is described in Table 1.1, there are 3 items to consider and a total capacity of 7.

We model this instance by creating a variable x_i for each item i, representing the number of times we will select this item, so we ensure that x_i is a natural number. The

Item	Value	Weight
1	7	4
2	2	2
3	5	3
Capacity		7

Table 1.1: Example instance of the knapsack problem.

objective function is the total value of the items chosen. Finally, we need to add a constraint such that the total weight is below the capacity. Below is the discrete optimization problem formulation :

$$\begin{aligned} \max\ & 7x_1 + 2x_2 + 5x_3 \\ & 4x_1 + 2x_2 + 3x_3 \leq 7 \\ & x_i \in \mathbb{N}, i = 1, \ldots, 3 \end{aligned} \tag{1.1}$$

The set of feasible solutions is

$$\mathrm{Sol}(\mathscr{P}) = \left\{ \begin{array}{c} (0,0,0), (1,0,0), (0,1,0), (0,0,1) \\ (1,1,0), (1,0,1), (0,1,1) \\ (0,2,0), (0,0,2), (0,3,0), (0,2,1) \end{array} \right\}$$

and the optimal solution is $x^* = (1,0,1)$ with a value $z^* = 12$.

1.2 Exact Decision Diagrams

A *weighted decision diagram* is a graphical structure which encodes a set of solutions to a discrete optimization problem $\mathscr{P}$. Formally, it is represented by a layered directed acyclic multigraph $B = (U, A, d, v)$ where U is the set of nodes, A is the set of arcs, d are the arc labels and v the arc weights. The set of nodes is partitioned into layers $L_1, \ldots, L_{n+1}$. Layers L_1 and L_{n+1} contain only one node, respectively the root node r and the terminal node t. An arc $a \in A$ is directed from a node in layer L_j, for some j, to a node in the next layer L_{j+1}. Its label $d(a) \in D(x_j)$ represents the assignment of value $d(a)$ to variable x_j. As a result, each path $p = \left(a^{(1)}, \ldots, a^{(n)}\right)$ from r to t is a complete assignment of the variables, with $x_j = d\left(a^{(j)}\right)$ for $j = 1, \ldots, n$, denoted as x^p. The set of all r to t paths of B encodes the set of assignments $\mathrm{Sol}(B)$. Each arc a also has a length $v(a)$, so we can define the length of a path $p = \left(a^{(1)}, \ldots, a^{(k)}\right)$ starting from r as $v(p) = \sum_{j=1}^{k} v\left(a^{(j)}\right)$. In an exact decision diagram, the length of r to t path is equal to the objective function value of the corresponding assignment. Formally, B is *exact* for $\mathscr{P}$ if

$$\begin{aligned} & \mathrm{Sol}(\mathscr{P}) = \mathrm{Sol}(B) \\ & f(x^p) = v(p), \text{ for all } r - t \text{ paths } p \text{ in } B. \end{aligned} \tag{1.2}$$

The *size* $|B|$ of a decision diagram is the number of nodes it contains in all layers. The *width* of a decision diagram is given by $\max_j |L_j|$, where $|L_j|$ is the *width* of layer j. Arcs leaving a same node always have different labels, so every node has a maximum out-degree of $|D(x_j)|$. A *binary decision diagram* encodes binary variables only, as opposed to *multi-valued decision diagrams* (MDD) in the general case.

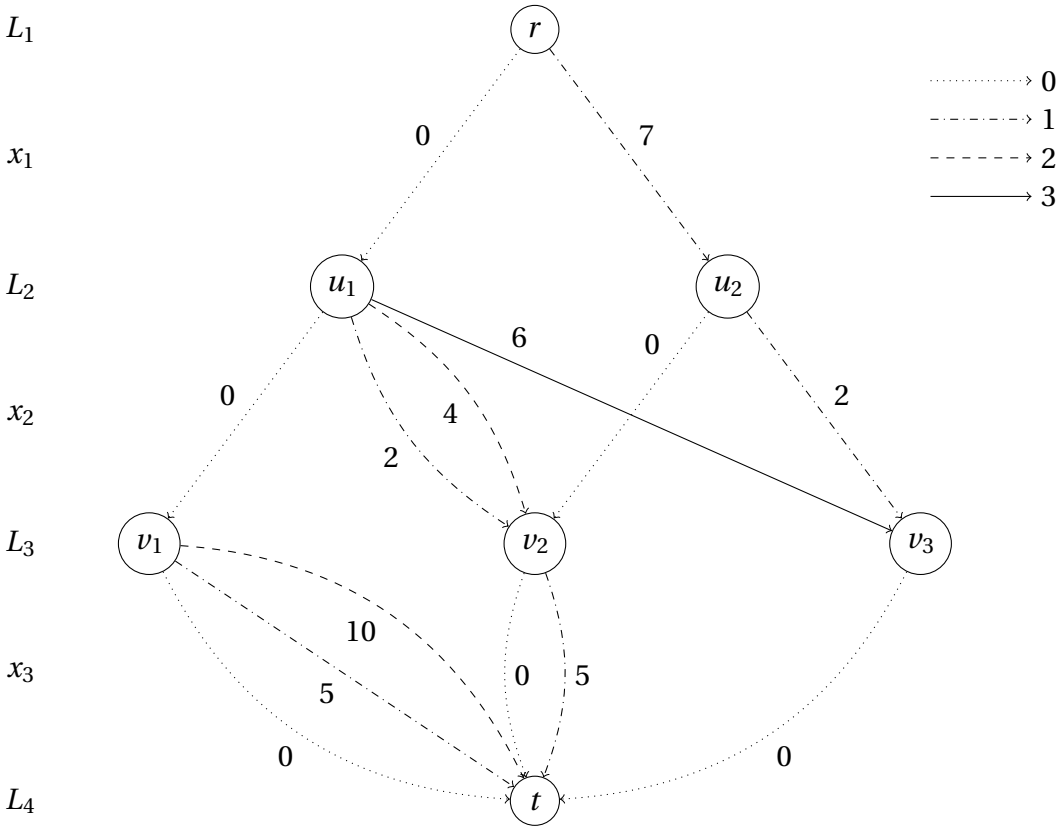


Figure 1.1: Complete decision diagram for the knapsack problem of Table 1.1. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3. Numbers on the arcs are their length.

Due to the equivalence stated in equation (1.2), exact decision diagrams reduce the resolution of discrete optimization problems to a longest-path problem on a directed acyclic graph. If p is a longest path in an exact decision diagram B for $\mathscr{P}$, then x^p is an optimal solution of $\mathscr{P}$, and its length $v(p)$ is the optimal value $z^*(\mathscr{P}) = f(x^p)$ of $\mathscr{P}$.

We say that two nodes of the same layer L_j are *equivalent* if the paths from each of them to the terminal node t correspond to the same set of assignments to $(x_j, \ldots, x_n)$, which implies they are redundant in the representation. A decision diagram is *reduced* if it does not contain any pair of equivalent nodes in any of its layers. Figure 1.1 shows an example of a reduced decision diagram.

Example 1.2.1. We build the exact decision diagram for the knapsack problem example on Figure 1.1. Assignments of the variables x_1, x_2 and x_3 are represented by the arcs respectively ranging from nodes in L_1 to L_2, L_2 to L_3 and L_3 to L_4. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3 and numbers on the arcs are their length. For example, the arc (r, u_2) represents the assignment $x_1 = 1$ and its length of 7 represents the fact that it adds 7 to the objective function. We can observe that all feasible solutions of the problem are encoded in the diagram, and in particular, the path $p^* = (r, u_2, v_2, t)$ (with the dash-dotted arc between v_2 and t) encodes the optimal solution $x^{p^*} = (1, 0, 1)$ with $v(p^*) = 7 + 0 + 5 = 12$. We also have that $|B| = 7$ and $|L_1| = 1, |L_2| = 2, |L_3| = 3, |L_4| = 1$.

1.3 Dynamic Programming Formulation

We now explain how we can formulate a discrete optimization problem $\mathscr{P}$ for the decision diagram approach. We will use a dynamic programming model, which is a recursive optimization method based on the resolution and combination of sub-problems in order to find the global optimal solution. For a discrete optimization problem $\mathscr{P}$ with n variables having domains $D(x_1), \ldots, D(x_n)$, a dynamic programming model usually consists of the following elements :

- A *state space* S partitioned in to the sets $S_1, \ldots, S_{n+1}$ corresponding to the $n+1$ stages. It includes a *root state* $\hat{r}$, a countable set of *terminal states* $\hat{t}_1, \ldots, \hat{t}_k$ and an *infeasible state* $\hat{0}$ which is the result state for transitions leading to infeasible solutions to $\mathscr{P}$. We have in particular that $S_1 = \{\hat{r}\}, S_{n+1} = \{\hat{t}_1, \ldots, \hat{t}_k, \hat{0}\}$ and $\hat{0} \in S_j$ for $j = 2, \ldots, n$.
- A set of *transition functions* $t_j : S_j \times D(x_j) \to S_{j+1}$ for $j = 1, \ldots, n$ which rule the transition between the states of consecutive stages. Transitions starting from an infeasible state always result in an infeasible state : $t_j(\hat{0}, d) = 0$ for any $d \in D(x_j)$.
- A set of *transition cost functions* $h_j : S_j \times D(x_j) \to \mathbb{R}$ for $j = 1, \ldots, n$ which associate a value with each transition.
- A *root value* v_r used to express constants in the objective function, its value is added to the cost of all transition from the root state $\hat{r}$.

The dynamic programming formulation has variables $(s, x) = (s_1, \ldots, s_{n+1}, x_1, \ldots, x_n)$, is written

$$\begin{aligned} \min \hat{f}(s,x) &= \sum_{j=1}^{n} h_j(s^j, x_j) \\ s^{j+1} &= t_j(s^j, x_j), \text{ for all } x_j \in D(x_j), j = 1, \ldots, n \\ s^j &\in S_j, j = 1, \ldots, n+1 \end{aligned} \tag{1.3}$$

and is *valid* for $\mathscr{P}$ if, for every $x \in D$, there exists an $s \in S$ such that (s, x) is feasible in (1.3) and

$$s^{n+1} = \hat{t} \text{ and } \hat{f}(s,x) = f, \text{ if } x \text{ is feasible for } \mathscr{P} \tag{1.4}$$

$$s^{n+1} = \hat{0}, \text{ if } x \text{ is infeasible for } \mathscr{P}. \tag{1.5}$$

Example 1.3.1. For our knapsack problem (1.1), let us call the value and weight of item j respectively v_j and w_j, and the capacity of the knapsack C. There exists a classical dynamic programming approach for this problem where each state s^j represents the remaining capacity of the knapsack when the number of items of type 1 to $j-1$ has already been decided.

At the root state, the knapsack is empty and we still have all items to consider, so $\hat{r} = C$. Then, a transition from a state s^j will be defined by x_j, the number items j we put in the knapsack. After choosing x_j, we can find the resulting state by computing the remaining space : $s^{j+1} = s^j - x_j w_j$. The total weight of the items chosen should not exceed the

capacity so we impose $s^{j+1} \geq 0$. The transition costs represent the value brought by the items put in the knapsack, so for a transition from state s^j to s^{j+1} with x_j copies of item j added to the solution, the cost will be $x_j v_j$.

Assuming we have $w_j \in \mathbb{N}$, for $j = 1, \ldots, n$, the complete dynamic programming formulation is the following :

- State spaces : $S_j = \{0, \ldots, C\}$ for $j = 2, \ldots, n+1$ and $\hat{r} = C$
- Transition functions : $t_j(s^j, x_j) = \begin{cases} s^j - x_j w_j & \text{if } s^j \geq x_j w_j, \\ \hat{0} & \text{otherwise.} \end{cases}$
- Cost functions : $h_j(s^j, x_j) = x_j v_j$
- Root value : $v_r = 0$

1.4 Compiling Exact Decision Diagrams

Algorithm 1 specifies the compilation procedure of the exact weighted decision diagram associated with a given dynamic programming formulation. We denote by $b_d(u)$ the node at the opposite end of the arc leaving node u with label d (if it exists). The decision diagram is built layer by layer starting with the root node r which is the only node in layer 1 and represents the root state $\hat{r}$. From a layer j, we then fill L_{j+1} with all nodes corresponding to *distinct* feasible states that can be reached from any state in L_j. For each of these transitions, we add an arc from the node in layer j to the one in layer $j+1$ where the length is given by the transition cost. Finally, we associate all terminal states $\hat{t}_1, ..., \hat{t}_k$ to a single terminal node t. The algorithm identifies each node with its corresponding state since distinct nodes always have distinct states.

Algorithm 1: Exact decision diagram top-down compilation.

```
1 create node r = r̂ and let L_1 = {r}
2 for j ← 1 to n do
3   let L_{j+1} = ∅
4   forall u ∈ L_j and d ∈ D(x_j) do
5     if t_j(u, d) ≠ 0̂ then
6       let u' = t_j(u, d), add u' to L_{j+1}
7       set b_d(u) = u' and v(u, u') = h_j(u, u')
8 merge nodes in L_{n+1} into terminal node t
```

In the algorithm, the ordering of the variables $x_1, \ldots, x_n$ is the one given by the dynamic programming model. However, reordering the variables could lead to a decision diagram with a different size. Moreover, the decision diagram created is not necessarily reduced and in general, identifying if two nodes are equivalent is NP-hard [7].

Example 1.4.1. Figure 1.2 shows the result of Algorithm 1 applied to the knapsack problem (1.1), without the final step of merging all nodes of the last layer in a single terminal node. Numbers in the gray boxes show the state corresponding to each node. We can note that nodes v_1 and v_4 both have a transition, respectively labeled 1 and 0, to node

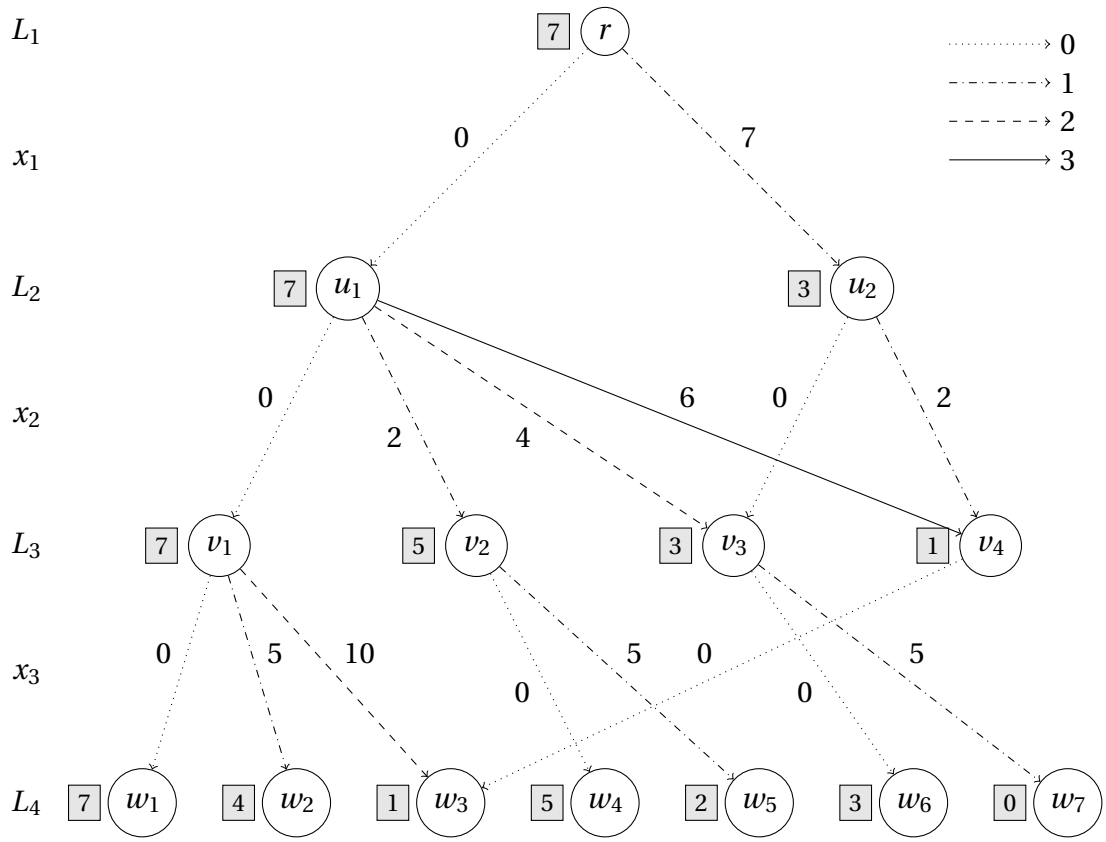


Figure 1.2: Exact decision diagram for the knapsack problem of Table 1.1 as computed by Algorithm 1. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3. Numbers on the arcs are their length. Numbers in the gray boxes show the remaining capacity in each node.

w_3 since they lead to same state. Furthermore, the resulting decision diagram will not be reduced, since v_2 and v_3 are equivalent nodes in the decision diagram of Figure 1.1 but they are not merged in the top-down compilation procedure.

1.5 Relaxed Decision Diagrams

In this section, we explain how we can use decision diagrams to compute upper bounds for discrete optimization problems. We introduce the concept of *relaxed decision diagrams*: a weighted decision diagram B is relaxed for an optimization problem $\mathscr{P}$ if B represents a superset of the feasible solutions of $\mathscr{P}$, and path lengths in B are upper bounds on the value of feasible solutions. Formally, B is relaxed for $\mathscr{P}$ if

$$\begin{aligned} &\mathrm{Sol}(\mathscr{P}) \subseteq \mathrm{Sol}(B), \\ &f(x^p) \leq v(p), \text{ for all } r-t \text{ paths } p \text{ in } B \text{ for which } x^p \in \mathrm{Sol}(\mathscr{P}). \end{aligned} \tag{1.6}$$

In Section 1.2, it is explained how exact decision diagrams reduce the resolution of discrete optimization problems to a longest-path problem : if p is a longest path in an exact decision diagram B for $\mathscr{P}$, then x^p is an optimal solution of $\mathscr{P}$, and its length $v(p)$ is the optimal value $z^*(\mathscr{P}) = f(x^p)$ of $\mathscr{P}$. If B is relaxed for $\mathscr{P}$, then a longest path p gives an

upper bound on the optimal value of the problem $\mathscr{P}$. The corresponding assignment x^p may not be feasible, but we know that $v(p) \geq z^\star(\mathscr{P})$. We will now show how relaxed decision diagrams can be built with a chosen maximum width.

The procedure for building relaxed decision diagrams of limited width extends the top-down compilation of exact decision diagrams presented in Section 1.4. In addition to the dynamic programming model of the problem, we will need a rule specifying how to *merge* nodes of the same layer. This operation should satisfy the condition (1.6) and could need a modification of the transition costs.

When compiling the decision diagram with the top-down construction procedure, we will merge a subset M of the nodes in layer L_j while its size exceeds the maximum width W. The operator $\oplus(M)$ is used to define the state of the merged node. In the decision diagram, all nodes in M are removed and a new node is created with state $\oplus(M)$. Every arc pointing to a node $u \in M$ is redirected to $\oplus(M)$ and its length v is modified to $\Gamma_M(v, u)$. Algorithm 2 formally states this process, where lines 3–8 have been added to Algorithm 1. In the algorithm, $a_d(u)$ denotes the arc leaving node u with label d, and $b_d(u)$ refers to the node at the opposite end of this arc (if it exists).

Algorithm 2: Relaxed decision diagram top-down compilation with maximum width W.

1 create node $r = \hat{r}$ and let $L_1 = \{r\}$
2 **for** $j \leftarrow 1$ **to** n **do**
3 **while** $|L_j| > W$ **do**
4 let $M = \text{node_select}(L_j)$
5 $L_j \leftarrow (L_j \setminus M) \cup \{\oplus(M)\}$
6 **forall** $u \in L_{j-1}$ and $d \in D(x_{j-1})$ with $b_d(u) \in M$ **do**
7 $b_d(u) \leftarrow \oplus(M)$
8 $v(a_d(u)) \leftarrow \Gamma_M(v(a_d(u)), b_d(u))$
9 let $L_{j+1} = \emptyset$
10 **forall** $u \in L_j$ and $d \in D(x_j)$ **do**
11 **if** $t_j(u, d) \neq \hat{0}$ **then**
12 let $u' = t_j(u, d)$, add u' to L_{j+1}
13 set $b_d(u) = u'$ and $v(u, u') = h_j(u, u')$
14 merge nodes in L_{n+1} into terminal node t

Note that the subset M of nodes to be merged is determined by a heuristic function *node_select* which can be based, for example, on the longest-path value of the nodes or on the similarity between a set of states in the layer (see Section 1.8 for more details).

Example 1.5.1. For the *Unbounded Knapsack Problem*, if we are given a set of nodes with different states, a valid merging operator would be to take the maximum remaining capacity among those nodes. By doing so, we ensure that all $u - t$ paths for $u \in M$ in the exact decision diagram are included in the relaxed decision diagram since the remaining capacity of the merged node is larger or equal for each node u. The cost of these paths will stay the same, we will however potentially create infeasible solutions from nodes for which the remaining capacity has been modified. Formally, we define $\oplus(M) = \max_{u \in M} u$ and $\Gamma_M(v, u) = v$.

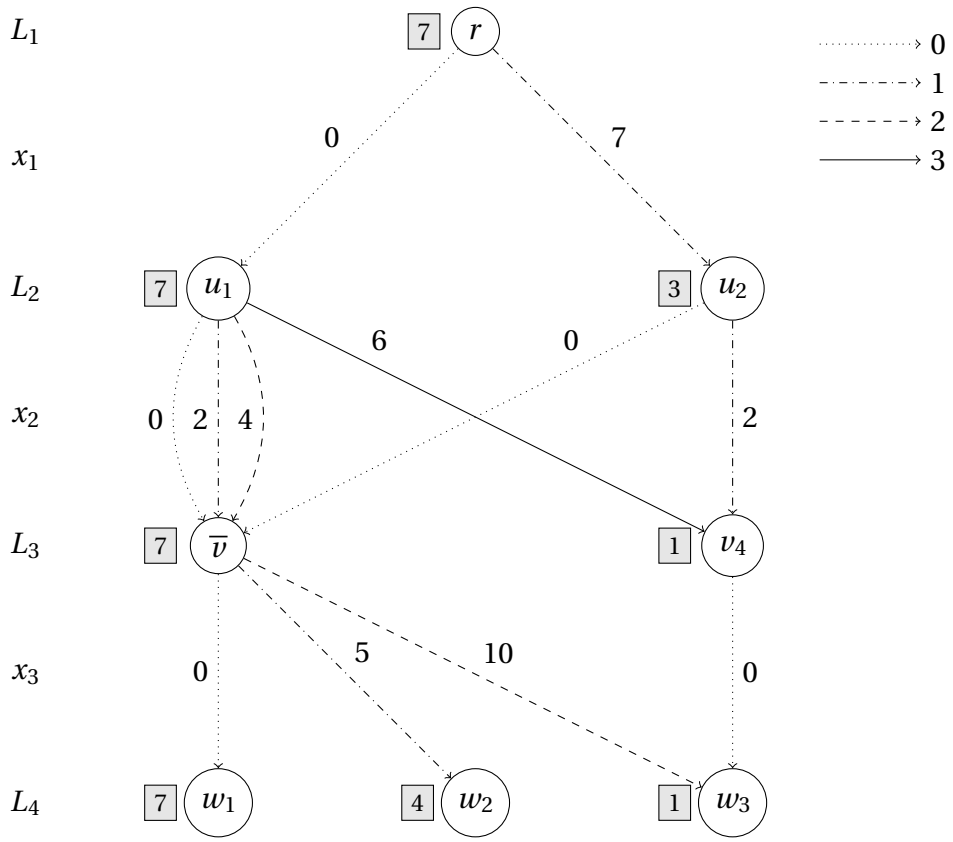


Figure 1.3: Relaxed decision diagram for the knapsack problem of Table 1.1 as computed by Algorithm 2. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3. Numbers on the arcs are their length. Numbers in the gray boxes show the state of each node.

Figure 1.3 shows a relaxed decision diagram for the knapsack problem (1.1) with a maximum width of 2, without the final step of merging all nodes of the last layer in a single terminal node. We can compare it to the exact decision diagram produced for the same problem by Algorithm 1 displayed on Figure 1.2. Layers L_1 and L_2 are identical but since L_3 has a width of 4 in the exact decision diagram, we need to apply our merging operator. We select $M = \{v_1, v_2, v_3\}$, compute the new state $\overline{v} = \oplus(M) = \max(7,4,3) = 7$ and modify the decision diagram accordingly. After generating the last layer, we find the longest path $p = (r, u_2, \overline{v}, w_3)$ with a value $v(p) = 7 + 0 + 10 = 17$. It corresponds to the solution $x^p = (1,0,2)$ which violates the capacity constraint. Thus, this solution is infeasible but provides an upper bound of 17.

1.6 Restricted Decision Diagrams

We have seen in Section 1.5 how relaxed decision diagrams can provide upper bounds for discrete optimization problems. In a similar way, we will now explain how lower bounds can be obtained with *restricted decision diagrams*. A weighted decision diagram B is restricted for an optimization problem $\mathscr{P}$ if B represents a subset of the feasible solutions of $\mathscr{P}$, and path lengths are lower bounds on the value of feasible solutions. That is, B is

restricted for $\mathscr{P}$ if

$$\begin{aligned} &\mathrm{Sol}(\mathscr{P}) \supseteq \mathrm{Sol}(B), \\ &f(x^p) \geq v(p), \text{ for all } r-t \text{ paths } p \text{ in } B \text{ for which } x^p \in \mathrm{Sol}(\mathscr{P}). \end{aligned} \tag{1.7}$$

In Section 1.2, it is explained how exact decision diagrams reduce the resolution of discrete optimization problems to a longest-path problem : if p is a longest path in an exact decision diagram B for $\mathscr{P}$, then x^p is an optimal solution of $\mathscr{P}$, and its length $v(p)$ is the optimal value $z^*(\mathscr{P}) = f(x^p)$ of $\mathscr{P}$. If B is restricted for $\mathscr{P}$, then a longest path p gives an lower bound on the optimal value of the problem $\mathscr{P}$ and the corresponding assignment x^p is always feasible, with $v(p) \leq z^\star(\mathscr{P})$. As for relaxed decision diagrams, we will now show how restricted decision diagrams can be built with a chosen limited width.

The procedure is simpler than for relaxed decision diagrams : we only need to remove nodes of a layer when its size exceeds W. Algorithm 3 formalizes this process, and only differs from Algorithm 1 for lines 3–5. Conditions (1.7) are always respected since we only delete solutions and never modify the states or length of the arcs which remain in the decision diagram. As in Algorithm 2, the set of nodes to be deleted is selected with a heuristic function *node_select* (more details in Section 1.8).

Algorithm 3: Restricted decision diagram top-down compilation with maximum width W.

```
1  create node r = r̂ and let L_1 = {r}
2  for j ← 1 to n do
3     while |L_j| > W do
4        let M = node_select(L_j)
5        L_j ← (L_j \ M)
6     let L_{j+1} = ∅
7     forall u ∈ L_j and d ∈ D(x_j) do
8        if t_j(u, d) ≠ 0̂ then
9           let u' = t_j(u, d), add u' to L_{j+1}
10          set b_d(u) = u' and v(u, u') = h_j(u, u')
11 merge nodes in L_{n+1} into terminal node t
```

Example 1.6.1. We show on Figure 1.4 a restricted decision diagram for the knapsack problem (1.1) and a maximum width of 2, without the final step of merging all nodes of the last layer in one terminal node. As for the relaxed decision diagram, layers L_1 and L_2 are exact since their width does not exceed 2 in the exact decision diagram (see Figure 1.2). Layer L_3 has 4 nodes in the exact decision diagram, so we decide to remove nodes v_2 and v_3 in the restricted decision diagram. The longest path of the resulting decision diagram is $p = (r, u_2, v_3, w_2)$ with a length of $v(p) = 7 + 0 + 5 = 12$. It corresponds to the solution $x^p = (1, 0, 1)$ and is in fact the optimal solution of the problem.

1.7 Branch-and-Bound Algorithm

In Sections 1.5 and 1.6, we showed that limited width decision diagrams can be used to compute lower and upper bounds for discrete optimization problems. We will now

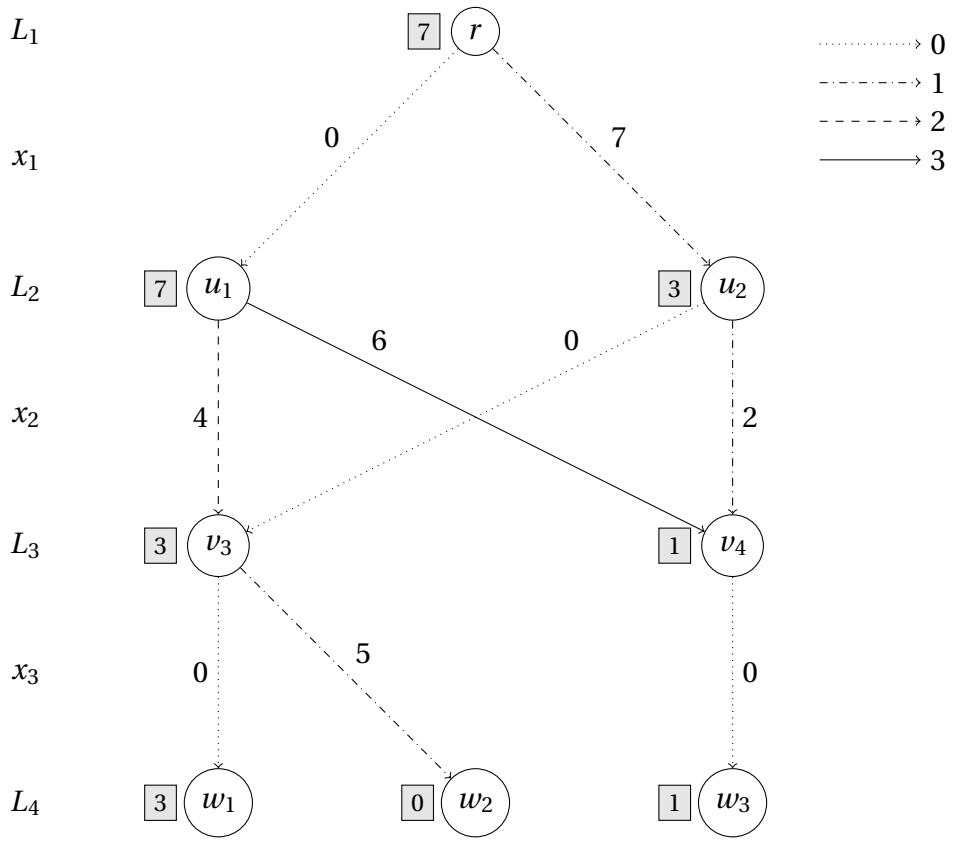


Figure 1.4: Restricted decision diagram for the knapsack problem of Table 1.1 as computed by Algorithm 3. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3. Numbers on the arcs are their length. Numbers in the gray boxes show the state of each node.

see an alternative *branch-and-bound* method based on relaxed and restricted decision diagrams. While in a traditional branch-and-bound we branch on the values of a variable, this approach branches on a particular subset of nodes in a relaxed decision diagram. Each node of this subset leads to a subproblem for which we can build a relaxed decision diagram, so the process can be repeated recursively until the whole search space has been explored (or pruned).

First, we give a few definitions and introduce notations which will ease the explanation of the branch-and-bound procedure. Let $\overline{B}$ be a relaxed decision diagram built by Algorithm 2 according to a valid dynamic programming model of a discrete optimization problem $\mathscr{P}$. A node $\overline{u}$ in $\overline{B}$ is said *exact* if all $r - \overline{u}$ paths in $\overline{B}$ lead to the same state s^j. A *cutset* of $\overline{B}$ is a subset S of nodes of $\overline{B}$ such that any $r - t$ path of $\overline{B}$ crosses at least one node in S. A cutset is *exact* if all nodes in S are exact.

For a given decision diagram B and nodes $u, u' \in B$ with $L(u) < L(u')$, we denote by $B_{uu'}$ the decision diagram induced by the nodes contained in directed paths from u to u' (where the arc labels and costs are the same as in B). In particular, we have $B_{rt} = B$. If B is an exact decision diagram for a discrete optimization problem $\mathscr{P}$, let $v^\star(B_{uu'})$ be the length of a longest $u - u'$ path in $B_{uu'}$. For a node u in B, we define $\mathscr{P}|_u$ to be the restriction of $\mathscr{P}$ whose feasible solutions correspond to all $r - t$ paths of B that cross u.

Lemma 1.7.1. *If B is an exact decision diagram for $\mathscr{P}$, then for any node u in B,*

$$v^\star(B_{ru}) + v^\star(B_{ut}) = z^\star(\mathscr{P}|_u).$$

Proof. $z^\star(\mathscr{P}|_u)$ is the length of a longest $r - t$ path of B that contains u, and any such path has length $v^\star(B_{ru}) + v^\star(B_{ut})$. □

Theorem 1.7.2. *Let $\overline{B}$ be a relaxed decision diagram created by Algorithm 2 using a valid dynamic programming model of the discrete optimization problem $\mathscr{P}$, and let S be an exact cutset of $\overline{B}$. Then*

$$z^\star(\mathscr{P}) = \max_{u \in S} \left\{ z^\star(\mathscr{P}|_u) \right\}.$$

Proof. Let B be the exact decision diagram for $\mathscr{P}$ created using the same dynamic programming model. Because each node $\overline{u} \in S$ is exact, it has a corresponding node u in B (i.e., a node associated with the same state), and S is a cutset of B. Thus

$$z^\star(\mathscr{P}) = \max_{u \in S} \left\{ v^\star(B_{ru}) + v^\star(B_{ut}) \right\} = \max_{u \in S} \left\{ z^\star(\mathscr{P}|_u) \right\}$$

where the second equation is due to Lemma 1.7.1. □

We now expose how we can solve a discrete optimization problem $\mathscr{P}$ by a branching procedure in which we enumerate a set of subproblems $\mathscr{P}|_u$ each time we branch, where u ranges over the nodes in an exact cutset of the current relaxed decision diagram. For each of these subproblems, we build a restricted and a relaxed decision diagram to respectively obtain lower and upper bounds.

Let u be one of the nodes on which we branch. Since u is an exact node, we have already constructed the exact decision diagram B_{ru} and we know the length $v^\star(u) = v^\star(B_{ru})$ of a longest path in B_{ru}. Thus, we can compute an upper bound on $z^\star(\mathscr{P}|_u)$ by evaluating a longest path length $v^\star(B_{ut})$ in a relaxed decision diagram $\overline{B}_{ut}$ with root value $v^\star(u)$. To build the relaxation $\overline{B}_{ut}$, we start the execution of Algorithm 2 with $j = L(u)$ and root node u, where the root value is $v_r = v^\star(u)$. We can obtain a lower bound on $z^\star(\mathscr{P}|_u)$ similarly by using a restricted decision diagram instead.

Algorithm 4: Branch-and-Bound Algorithm

```
initialize Q = {r}, where r is the initial DP state
let z_opt = −∞, v*(r) = 0
while Q ≠ ∅ do
    u ← select_node(Q), Q ← Q \ {u}
    create restricted decision diagram B'_ut with root u and v_r = v*(u) (Algorithm 3)
    if v*(B'_ut) > z_opt then
        z_opt ← v*(B'_ut)
    if B'_ut is not exact then
        create relaxed decision diagram B̄_ut with root u and v_r = v*(u) (Algorithm 2)
        if v*(B̄_ut) > z_opt then
            let S be an exact cutset of B̄_ut
            forall u' ∈ S do
                let v*(u') = v*(u) + v*(B̄_uu')
                add u' to Q
return z_opt
```

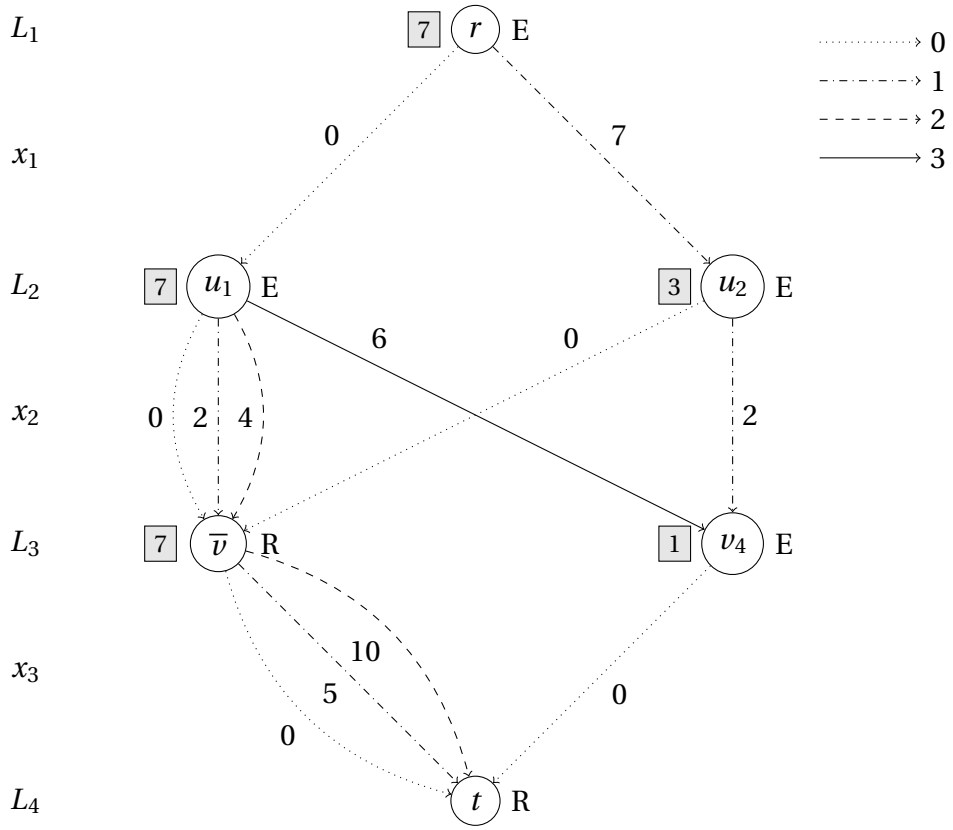
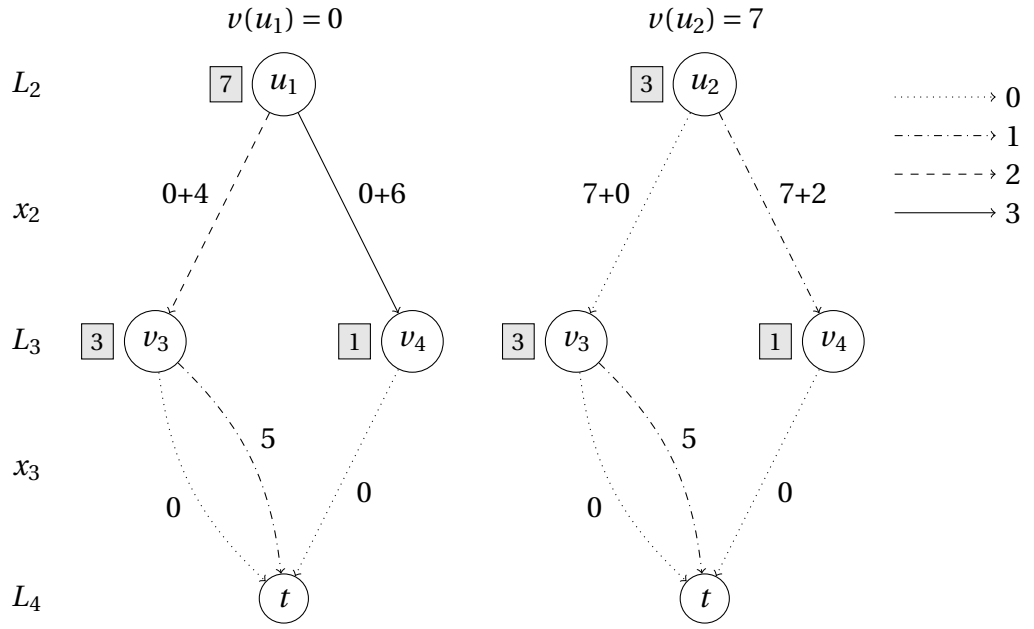


Figure 1.5: Relaxed decision diagram for the knapsack problem of Table 1.1 as computed by Algorithm 2. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3. Numbers on the arcs are their length. Numbers in the gray boxes show the state of each node. Letters E and R indicate whether the node is exact or relaxed.

The branch-and-bound algorithm is formally stated in Algorithm 4. Initially, we have a set $Q = \{r\}$ of open nodes with a single node, being the root state r of the dynamic programming model. While open nodes remain, we select a node u from Q with a heuristic function *node_select* (see Section 1.8). Then, we compute a lower bound on $z^\star(\mathscr{P}|_u)$ by building a restricted decision diagram B'_{ut} as we just described and update the current optimal solution z_{opt} if needed. If B'_{ut} is exact (that is, $|L_j|$ never exceeded the maximum width W in Algorithm 3), then $\mathscr{P}|_u$ has been fully explored and we can stop branching at node u. Otherwise, we compute an upper bound on $z^\star(\mathscr{P}|_u)$ by constructing a relaxed decision diagram $\overline{B}_{ut}$ as described previously. We can prune the search if the upper bound of this subproblem is smaller than our current optimal solution z_{opt}. If it is not the case, we identify an exact cutset S of $\overline{B}_{ut}$ and add the nodes in S to Q. Because S is exact, for each $u' \in S$ we know that $v^\star(u') = v^\star(u) + v^\star(\overline{B}_{uu'})$. The search is completed when Q is empty and the solution z_{opt} is optimal by Theorem 1.7.2.

Example 1.7.1. Recall the relaxed decision diagram from Figure 1.3, we show on Figure 1.5 the same decision diagram where this time we merged the nodes of the last layer. Letters E and R next to the nodes respectively indicate *exact* and *relaxed* nodes. Node $\overline{v}$ is relaxed because it is the result of the merging of nodes with different states (nodes v_1, v_2 and v_3 on Figure 1.2).



(a) Starting at $j = 2$ with node u_1 as root.

(b) Starting at $j = 2$ with node u_2 as root.

Figure 1.6: Restricted decision diagram for the knapsack problem of Table 1.1 as computed by Algorithm 3. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3. Numbers on the arcs are their length. Numbers in the gray boxes show the state of each node.

We now demonstrate a possible outcome of the branch-and-bound algorithm. Figure 1.4 shows a restricted decision diagram built from the root state, where we have a longest path length of 12. We then build a relaxed decision diagram from the root state, displayed on Figure 1.5, which gives us an upper bound of 17. A valid cutset for this relaxed decision diagram is $S = \{u_1, u_2\}$, with $v(u_1) = 0$ and $v(u_2) = 7$. Let us build a restricted decision diagram for each of these two subproblems, with again a maximum width of 2.

Figure 1.6a shows a restricted decision diagram created from node u_1. It is not exact since we had to remove nodes obtained with the transition from u_1 labeled 0 and 1 (v_1 and v_2 on Figure 1.2). The longest path value is 9 so we do not update the current best solution. Then, we compute a relaxed decision diagram, if we chose $\overline{v}_1 = \oplus(\{v_1, v_2\})$ and $\overline{v}_2 = \oplus(\{v_3, v_4\})$ (where v_1, v_2, v_3, v_3 are nodes in the exact decision diagram showed on Figure 1.2), we obtain the decision diagram showed on Figure 1.7. The longest path has a length of 12, which is not greater than our current best solution, so we can prune the search in this subproblem.

From node u_2, we obtain the restricted decision diagram showed on Figure 1.6b, which is exact so the branching is stopped in this subproblem. The longest path value is 12 and corresponds to the same solution we found previously.

The search terminates since no open nodes remain, the optimal value is 12, associated with the solution $(1, 0, 1)$.

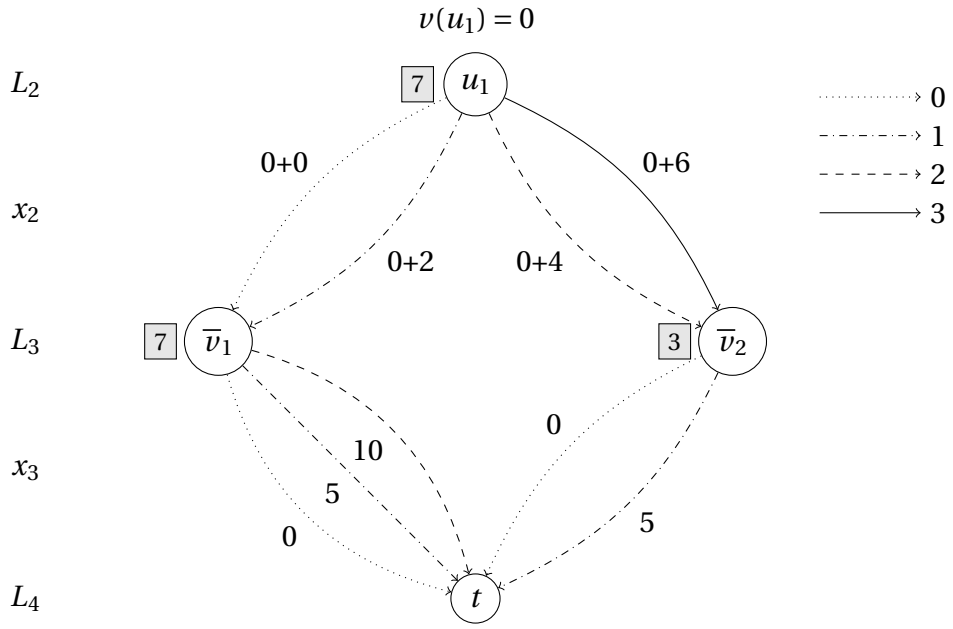


Figure 1.7: Relaxed decision diagram for the knapsack problem of Table 1.1 as computed by Algorithm 3 starting at $j = 2$ with node u_1 as root. Dotted, dash-dotted, dashed and solid arcs respectively denote arc labels 0, 1, 2 and 3. Numbers on the arcs are their length. Numbers in the gray boxes show the state of each node.

1.8 Heuristic components

In the branch-and-bound algorithm, there are several details which were not discussed yet. First, the function *select_node* at line 4 of Algorithm 4 determines the order in which the open nodes will be selected in the branch-and-bound. We select the node u with the minimum longest-path value $v^\star(u)$.

In Algorithms 2 and 3, the function *select_node* decides which nodes will be respectively merged and removed. A simple heuristic is used : after building a layer L_j of a relaxed or restricted decision diagram, we sort the nodes in L_j according to a rank function $rank(u)$. The subset of nodes M is then computed as the $|L_j| - W$ lowest-ranked nodes of L_j. This rank function can be adapted to each problem if particular features can be good indicators of the quality of a partial solution. By default, we use the longest-path value $v^\star(u)$ as rank function because a node with low longest-path value is less likely to lead to the optimal solution.

We also have multiple options for choosing an exact cutset of the relaxed decision diagrams (see line 11 of Algorithm 4). One of them is the *last exact layer* defined as the set of nodes in the layer L_j of a relaxed decision diagram $\bar{B}$ such that all nodes in L_j are exact and j is maximum. Another one is the *frontier cutset* which contains all the deepest exact nodes. Formally, for a relaxed decision diagram $\bar{B}$ the frontier cutset is

$$\mathrm{FC}(\bar{B}) = \{u \text{ in } \bar{B} \mid u \text{ is exact and } \exists d : b_d(u) \text{ is relaxed}\}$$

Finally, note that for some problems, variables can be ordered in such a way that the size of the decision diagram is reduced or the bounds obtained are tighter (e.g. start with the items with the largest value-weight ratio in a knapsack problem).

Chapter 2

Example Problems

In order to build a good understanding of how problems can be efficiently formalized in the framework presented in the previous chapter, we will now look at three different problems successfully solved with the decision diagram approach in [7, 8].

2.1 Maximum Independent Set Problem

Let $G = (V, E)$ be a graph with set of vertices $V = \{1, \dots, n\}$. An *independent set* I is a subset of vertices $I \subseteq V$ such that no two vertices in I share an edge in E. Given a weight $w_j \geq 0$ for each vertex $j \in V$, the *maximum independent set problem* (MISP) is the problem of finding a maximum-weight independent set of G.

We formulate this problem as a discrete optimization problem by letting variable x_j for each vertex $j \in V$ denote whether vertex j is selected in the independent set I, so we have $D(x_j) = \{0, 1\}$. The objective function is the sum of the weights of selected vertices and we need constraints to prevent selecting connected vertices. Formally, we can write

$$\begin{aligned} \max \ & \sum_{j=1}^{n} w_j x_j \\ & x_i + x_j \leq 1, \text{ for all } (i, j) \in E \\ & x_j \in \{0, 1\}, \text{ for all } j \in V \end{aligned} \tag{2.1}$$

Example 2.1.1. Figure 2.1 shows a graph with weights associated to the vertices. In this case, the optimal solution is $x^\star = (1, 0, 0, 0, 1)$ with value $z^\star = 11$ and corresponds to selecting nodes 1 and 5 in the independent set.

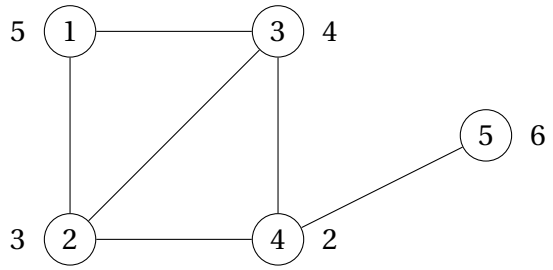


Figure 2.1: Example graph for the MISP.

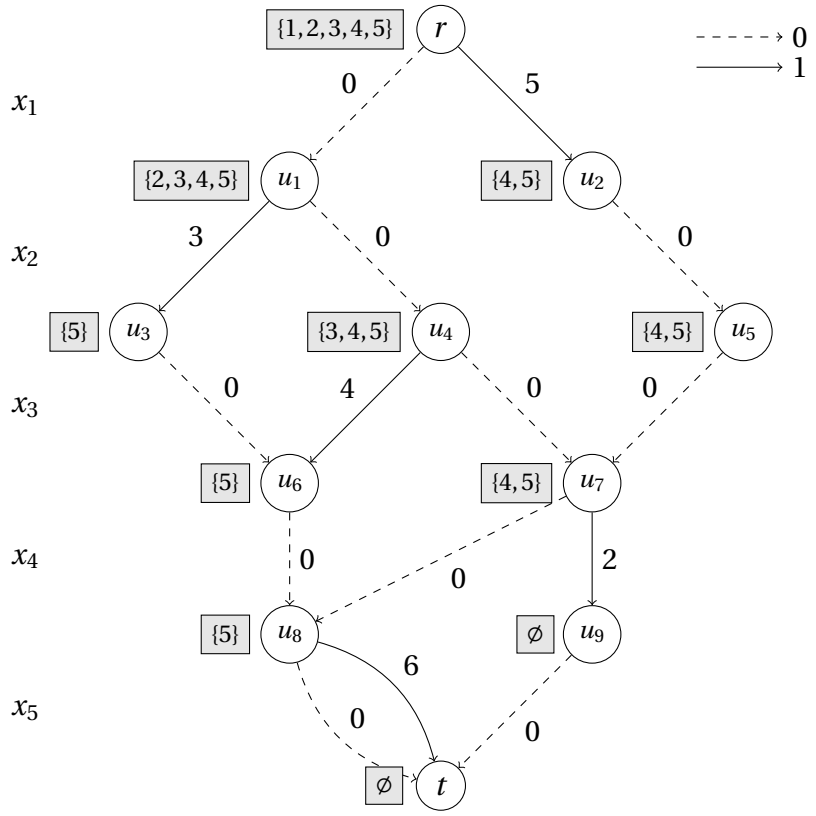


Figure 2.2: Exact decision diagram for the MISP example.

2.1.1 Dynamic Programming Formulation

In the dynamic programming model, we will decide successively for each vertex j if we include it in the independent set or not. If we add it to the independent set, we cannot consider vertices in the neighborhood $N(j)$ of j, defined as $N(j) = \{j' \in V \mid (j, j') \in E\}$. The states s^j represent the set of vertices which can still be added to the partial independent set. With $V_j = \{j, \dots, n\}$, the complete formulation is as follows :

- State spaces : $S_j = 2^{V_j}$ for $j = 2, \dots, n$, $\hat{r} = V$ and $\hat{t} = \emptyset$
- Transition functions : $t_j(s^j, 0) = s^j \setminus \{j\}$, $t_j(s^j, 1) = \begin{cases} s^j \setminus \big(N(j) \cup \{j\}\big) & \text{if } j \in s^j, \\ \hat{0} & \text{if } j \notin s^j. \end{cases}$
- Cost functions : $h_j(s^j, 0) = 0, h_j(s^j, 1) = w_j$
- Root value : $v_r = 0$

Example 2.1.2. Figure 2.2 presents the exact decision diagram for the graph of Figure 2.1 built following the model we just explained, excluding the infeasible node $\hat{0}$ for clarity.

2.1.2 Relaxation Formulation

In this problem, we reach the infeasible state only when a vertex j is not in s^j. Therefore, if we merge states by taking their union, we ensure that no solutions are lost. Formally, we write the merging operator as $\oplus(M) = \bigcup_{u \in M} u$.

2.2 Maximum Cut Problem

Given a graph $G = (V, E)$ with set of vertices $V = \{1, \dots, n\}$, a *cut* (S, T) is a partition of the vertices in V. Given edge weights, the value $v(S, T)$ of a cut is the sum of the weight of the edges connecting a vertex in S and another in T. We assume without loss of generality that the graph is complete, since missing edges can be added with zero weight. The *maximum cut problem* (MCP) asks for a cut (S, T) with maximum value.

In the discrete optimization formulation, we let variable x_j represent the side of the cut where we place vertex j, so we have that $D(x_j) = \{\mathrm{S}, \mathrm{T}\}$. We define $S(x) = \{j \mid x_j = \mathrm{S}\}$ and $T(x) = \{j \mid x_j = \mathrm{T}\}$ and since all partitions are feasible, we can write

$$\begin{aligned} \max \ & v\left(S(x), T(x)\right) \\ & x_j \in \{\mathrm{S}, \mathrm{T}\}, \text{ for all } j \in V \end{aligned} \tag{2.2}$$

Note that we can impose $x_1 = \mathrm{S}$ without loss of generality.

Example 2.2.1. We show on Figure 2.3 an instance of the problem, where an optimal solution is $x^\star = (\mathrm{S}, \mathrm{T}, \mathrm{S}, \mathrm{S})$ with value $z^\star = 6$.

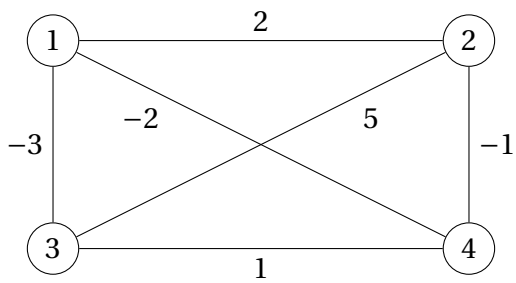


Figure 2.3: Example graph for the MCP.

2.2.1 Dynamic Programming Formulation

As for the MISP, we consider each vertex successively and decide on which side of the cut we place it. A possible choice for the state variables s^j would be the current set of vertices in S because it allows to compute the transition cost of the next vertex to be placed. Here, we consider an alternative formulation which focuses on merging nodes leading to similar objective function values. Therefore, a state s^j will contain, for vertex $j, \dots, n$, the net marginal benefit of placing that vertex in T, given the previous choices. Using the notations $(\alpha)^+ = \max\{\alpha, 0\}$ and $(\alpha)^- = \min\{\alpha, 0\}$, we now write the dynamic programming formulation.

- State spaces : $S_j = \left\{ s^j \in \mathbb{R}^n \mid s^j_l = 0, l = 1, \dots, j-1 \right\}$ and in particular $\hat{r} = \hat{t} = (0, \dots, 0)$
- Transition functions : $t_j(s^j, x_j) = (0, \dots, 0, s^{j+1}_{j+1}, \dots, s^{j+1}_n)$, where

$$s^{j+1}_l = \left\{ \begin{array}{ll} s^j_l + w_{jl}, & \text{if } x_j = \mathrm{S} \\ s^j_l - w_{jl}, & \text{if } x_j = \mathrm{T} \end{array} \right\}, l = j+1, \dots, n$$

- Cost functions : $h_1(s^1, x_1) = 0$ for $x = \{\text{S,T}\}$, and

$$h_j(s^j, x_j) = \left\{ \begin{array}{ll} (-s_j^j)^+ + \sum\limits_{\substack{l>j \\ s_l^j w_{jl} \le 0}} \min\left\{|s_l^j|, |w_{jl}|\right\}, & \text{if } x_j = \text{S} \\ (s_j^j)^+ + \sum\limits_{\substack{l>j \\ s_l^j w_{jl} \ge 0}} \min\left\{|s_l^j|, |w_{jl}|\right\}, & \text{if } x_j = \text{T} \end{array} \right\}, j = 2, \ldots, n$$

- Root value : $v_r = \sum\limits_{1 \le j < j' \le n} (w_{jj'})^-$

Example 2.2.2. On Figure 2.4, we have the exact decision diagram for the graph of Figure 2.3. The states and transition costs are quite tedious to check mentally but we can verify the path length of the optimal solution $x^\star = (\text{S,T,S,S}) : v\left(x^\star\right) = -6 + 0 + 3 + 9 + 0 = 6$. Remark that at the last transition, we could have chosen $x_4 = \text{T}$ with the same path length so $x^\star = (\text{S,T,S,T})$ is another optimal solution.

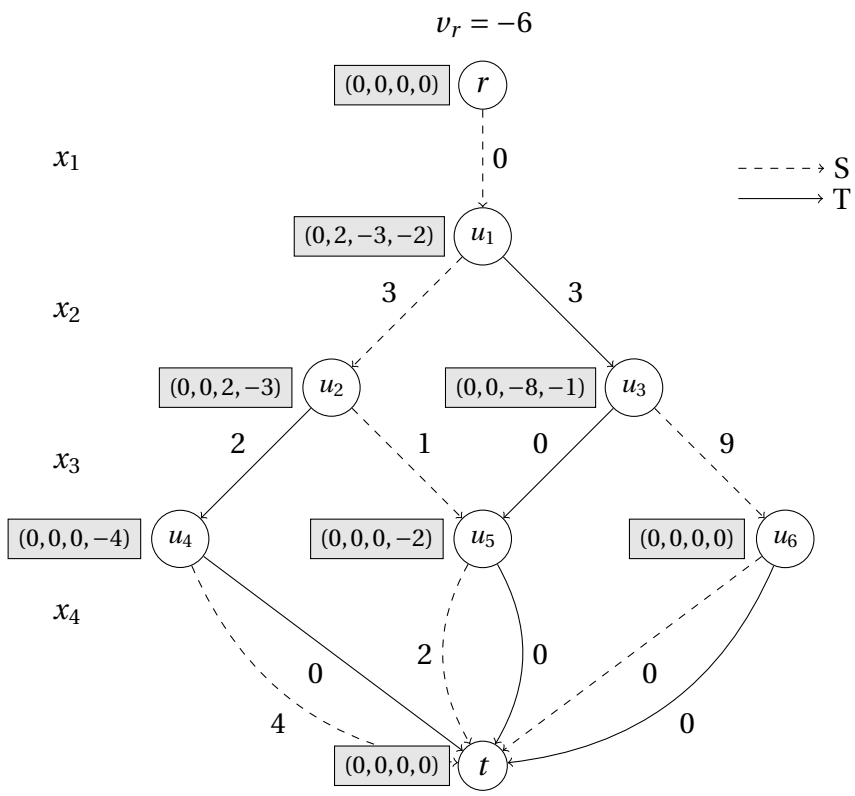


Figure 2.4: Exact decision diagram for the MCP example.

2.2.2 Relaxation Formulation

When merging nodes, we would like the resulting node to have values as close as possible to those of the original nodes but at the same time, path lengths should not decrease. Intuitively, we could think that $\oplus(M)_l = \max_{u \in M}\{u_l\}$ for each l is a valid relaxation operator, because increasing state values would only increase path lengths. However, this

can reduce path lengths as well. Thus, we will offset any reduction in path lengths by adding the absolute value of the state change to the length of incoming arcs. It leads to the following relaxation operators, with $M \subset L_j$:

$$\begin{aligned} \oplus(M)_l &= \left\{ \begin{array}{ll} \min\limits_{u \in M} \{u_l\}, & \text{if } u_l \geq 0 \text{ for all } u \in M \\ -\min\limits_{u \in M} \{|u_l|\}, & \text{if } u_l \leq 0 \text{ for all } u \in M \\ 0, & \text{otherwise} \end{array} \right\}, l = j, \ldots, n \\ \Gamma_M(v, u) &= v + \sum_{l \geq j} (|u_l| - |\oplus(M)_l|), \text{ for all } u \in M. \end{aligned} \tag{2.3}$$

2.3 Maximum 2-Satisfiability Problem

Let $x = (x_1, \ldots, x_n)$ be a tuple of Boolean variables, which can take values T or F corresponding respectively to *true* and *false*. A *literal* is either a variable x_j or its negation $\neg x_j$. In this problem, a *clause* c_i is defined as a disjunction of literals, it is thus satisfied if any of the literals is true. Given a set of clauses $C = \{c_1, \ldots, c_m\}$, each constituted of exactly two literals, and associated with a weight w_i, the *maximum 2-satisfiability problem* (MAX-2SAT) asks for an assignment of the Boolean variables such that the sum of the satisfied clauses in C is maximal.

The discrete optimization formulation simply uses the variables x_j with $D(x_j) = \{\mathrm{T}, \mathrm{F}\}$. Any assignment is valid so we have that

$$\begin{aligned} \max \sum_{i=1}^{m} w_i c_i(x) \\ x_j \in \{\mathrm{T}, \mathrm{F}\}, j = 1, \ldots, n \end{aligned} \tag{2.4}$$

where $c_i(x) = 1$ if x satisfies clause c_i, and $c_i(x) = 0$ otherwise.

Example 2.3.1. Given the clauses shown in Table 2.1, the optimal solution is $x^\star = (\mathrm{F}, \mathrm{T}, \mathrm{T})$ with value $z^\star = 19$ since all clauses are satisfied except c_5.

Index	Clause	Weight
1	$x_1 \vee x_3$	3
2	$\neg x_1 \vee \neg x_3$	5
3	$\neg x_1 \vee x_3$	4
4	$x_2 \vee \neg x_3$	2
5	$\neg x_2 \vee \neg x_3$	1
6	$x_2 \vee x_3$	5

Table 2.1: Example clauses for the MAX-2SAT.

2.3.1 Dynamic Programming Formulation

First, we assume without loss of generality that the problem contains all $4\binom{n}{2}$ possible clauses of two literals, since missing clauses can be included with zero weight. So formally, the set of clauses C contains $x_j \vee x_k$, $x_j \vee \neg x_k$, $\neg x_j \vee x_k$ and $\neg x_j \vee \neg x_k$ for each

pair $j, k \in \{1, \dots, n\}$ with $j \neq k$ and with respective weights w_{jk}^{TT}, w_{jk}^{TF}, w_{jk}^{FT} and w_{jk}^{FF}. Similarly to the formulation of the maximum cut problem, we define each state variable s^j as an array $(s_1^j, \dots, s_n^j)$ where each s_l^j is the net benefit of setting x_l to true, given the previous choices. The net benefit is the advantage of setting $x_l = \mathrm{T}$ over setting $x_l = \mathrm{F}$. The dynamic programming formulation is as follows :

- State spaces : $S_j = \left\{ s^j \in \mathbb{R}^n \mid s_l^j = 0, l = 1, \dots, j-1 \right\}$ and in particular $\hat{r} = \hat{t} = (0, \dots, 0)$
- Transition functions : $t_j(s^j, x_j) = (0, \dots, 0, s_{j+1}^{j+1}, \dots, s_n^{j+1})$, where

$$s_l^{j+1} = \left\{ \begin{array}{ll} s_l^j + w_{jl}^{\mathrm{TT}} - w_{jl}^{\mathrm{TF}}, & \text{if } x_j = \mathrm{F} \\ s_l^j + w_{jl}^{\mathrm{FT}} - w_{jl}^{\mathrm{FF}}, & \text{if } x_j = \mathrm{T} \end{array} \right\}, l = j+1, \dots, n$$

- Cost functions : $h_1(s^1, x_1) = 0$ for $x = \{\mathrm{S,T}\}$, and for $j = 2, \dots, n$,

$$h_j(s^j, x_j) = \left\{ \begin{array}{ll} (-s_j^j)^+ + \sum_{l>j} \left(w_{jl}^{\mathrm{FT}} + w_{jl}^{\mathrm{FF}} + \min\left\{ (s_l^j)^+ + w_{jl}^{\mathrm{TT}}, (-s_l^j)^+ + w_{jl}^{\mathrm{TF}} \right\} \right), & \text{if } x_j = \mathrm{F} \\ (s_j^j)^+ + \sum_{l>j} \left(w_{jl}^{\mathrm{TF}} + w_{jl}^{\mathrm{TT}} + \min\left\{ (s_l^j)^+ + w_{jl}^{\mathrm{FT}}, (-s_l^j)^+ + w_{jl}^{\mathrm{FF}} \right\} \right), & \text{if } x_j = \mathrm{T} \end{array} \right\}$$

- Root value : $v_r = 0$

Example 2.3.2. Figure 2.5 shows the exact decision diagram for the instance described in Table 2.1 according to the dynamic programming model.

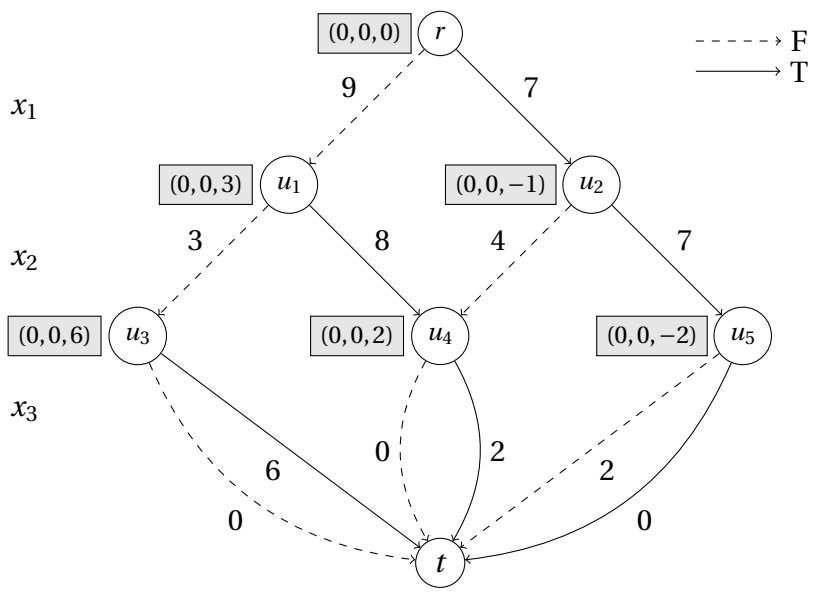


Figure 2.5: Exact decision diagram for the MAX-2SAT example.

2.3.2 Relaxation Formulation

Since the state variables are very similar to those of the maximum cut problem, we use the same relaxation operators.

Chapter 3

Minimum Linear Arrangement Problem

The decision diagram approach for the *minimum linear arrangement problem* (MinLA) has already been discussed in [7] but using a brute-force model. We will now present an alternative formulation for this problem starting from the dynamic programming model presented in [21] and providing a novel relaxation operator.

3.1 Introduction

Let $G = (V, E)$ be a connected graph, with $V = \{1, 2, \ldots, n\}$ the set of vertices and E a set of weighted edges, where $w_{ij} \in \mathbb{R}^+$ is the weight of the edge connecting vertices i and j. For the simplicity of the notations, we make the graph fully connected by adding zero weight edges between nodes not connected before. A *linear arrangement* is an ordering of the vertices defined by the bijection $\pi : V \to \{1, 2, \ldots, n\}$.

The *minimum linear arrangement problem* asks for a minimum-cost arrangement π, where the cost is given by :

$$LA_\pi(G) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j>i}}^{n} w_{ij} |\pi(i) - \pi(j)| \tag{3.1}$$

We formulate the MinLA as a discrete optimization problem by defining the $0-1$ variables x_{ij} for $i = 1, \ldots, n$ and $j = 1, \ldots, n$ where

$$x_{ij} = \begin{cases} 1 & \text{if } \pi(i) = j, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

which leads us to the formulation

$$\begin{aligned} \min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{p=1}^{n} \sum_{\substack{q=1 \\ q>p}}^{n} & x_{ip} x_{jq} w_{ij} (q - p) \\ \sum_{j=1}^{n} x_{ij} = 1, \;\; & 1 \le i \le n \\ \sum_{i=1}^{n} x_{ij} = 1, \;\; & 1 \le j \le n \\ x_{ij} \in \{0, 1\}, \;\; & 1 \le i, j \le n \end{aligned} \tag{3.3}$$

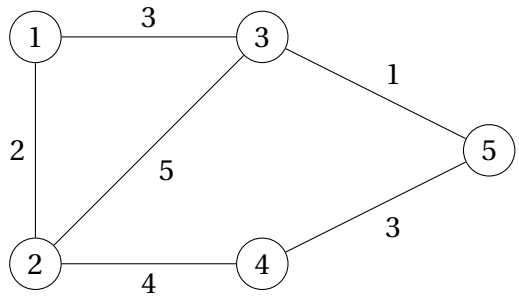


Figure 3.1: Example graph for the MinLA.

where the constraints ensure that each vertex is placed at exactly one position and that each position is used exactly once.

Example 3.1.1. From the graph of Figure 3.1, we build the linear arrangement given by $\pi(1) = 1, \pi(2) = 2, \pi(3) = 3, \pi(4) = 4, \pi(5) = 5$ displayed on Figure 3.2. We will now compute the cost of this solution (we only kept the terms where $x_{ip}x_{jq} = 1$ and $w_{ij} \neq 0$) :

$$\begin{aligned} LA_\pi(G) &= (2-1)\,w_{12} + (3-1)\,w_{13} + (3-2)\,w_{23} + (4-2)\,w_{24} + (5-3)\,w_{35} + (5-4)\,w_{45} \\ &= w_{12} + 2w_{13} + w_{23} + 2w_{24} + 2w_{35} + w_{45} \\ &= 2 + 2\cdot 3 + 5 + 2\cdot 4 + 2\cdot 1 + 3 \\ &= 26 \end{aligned}$$

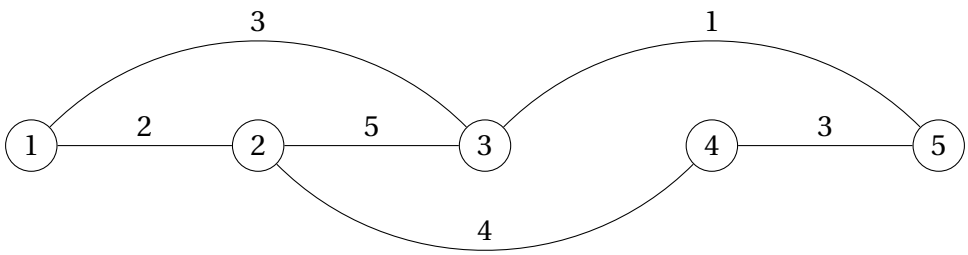


Figure 3.2: Example linear arrangement for the graph of Figure 3.1.

3.2 Dynamic Programming Formulation

The dynamic programming formulation is based on the following observation : the best ordering of a set of vertices S, placed to the right of the vertices in the set L and to the left of those in the set R, is independent of the internal ordering of the vertices in the sets L and R [21]. This statement is formalized in the theorem below.

Theorem 3.2.1. *Let $G = (V, E)$ be a graph and $L, S \subset V$ where $L \cap S = \emptyset$ and $|L| = l$, $|S| = s$. Let π, ϕ be two permutations of V, such that $\{\pi(i) \mid i \in L\} = \{\phi(i) \mid i \in L\} = \{1, \ldots, l\}$ and $\{\pi(i) \mid i \in S\} = \{\phi(i) \mid i \in S\} = \{l+1, \ldots, l+s\}$.*
Denote by π^*_S the permutation that minimizes the linear arrangement cost, over all permutations that differ from π only with respect to the places of vertices in S (i.e., for all

$i \notin S, \pi(i) = \pi^*_S(i)$).
Define a permutation ϕ^*_S as follows :

$$\phi^*_S(i) = \begin{cases} \phi(i) & \text{if } i \notin S, \\ \pi^*_S(i) & \text{otherwise.} \end{cases}$$

Then, ϕ^*_S minimizes the linear arrangement cost, over all permutations that differ from ϕ only with respect to the places of vertices in S.

As a result, we can apply a dynamic programming approach by recursively finding the optimal local ordering of each subset of V. We will place the vertices one by one on the line from left to right and store partial solutions in a table : MinLA[S] will contain the cost of the optimal local ordering where vertices in S still need to be placed on the right. After defining MinLA[V] = 0, we can then compute optimal orderings for MinLA[$S' \subset V$] with $|S'| = j$ if we know all optimal orderings for MinLA[$S \subset V$] with $|S| = j+1$ by placing every possible vertex at the next position in each of the partial solutions. For each of these transitions, we will add to the cost of the arrangement one time the weight of each edge incident to one vertex in S' and one not in S' because we know that the vertex in S' will be placed in the best case at the next position.

Example 3.2.1. In order to illustrate how the global cost is computed incrementally, we show on Figure 3.3 the same arrangement as on Figure 3.2 but this time, the edges appear the number of times they are multiplied in the cost function, so if we take the sum of all edge costs on this graph, we obtain the arrangement cost. We can also visualize the cost of each transition, which as a reminder is the sum of the weight of edges between one fixed vertex and one still to be used.

For instance, when we place the vertex 1 in first position, we add one time the weight of all edges incident to this vertex i.e. $w_{12} + w_{13} = 2+3 = 5$. After setting vertex 2 in second position, we add the weight of edges leaving vertices 1 or 2 and ending in the remaining vertices, so we have $w_{13} + w_{23} + w_{24} = 3+5+4 = 12$. In Figure 3.3, we can observe that each transition cost is represented by all edges contained between two consecutive dashed lines.

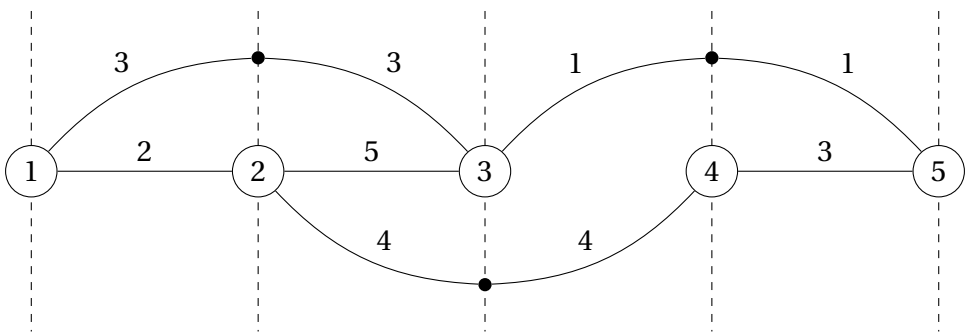


Figure 3.3: Linear arrangement of Figure 3.2 where we represent edges the number of times they are multiplied in the cost function.

Back to our dynamic programming formulation, we can now write formally that

$$\begin{cases} \text{MinLA}[V] = 0 \\ \text{MinLA}[S \subset V] = \min_{k \notin S} \left\{ \text{MinLA}[S \cup \{k\}] + \sum_{i \notin S} \sum_{j \in S} w_{ij} \right\} \\ \text{MinLA}^* = \text{MinLA}[\emptyset] \end{cases} \tag{3.4}$$

We now formalize the components of our dynamic programming model in the framework explained in Section 1.3. As said above, states of the dynamic program are all subsets of V. However, we choose a richer but equivalent representation of the states which will be useful later on. A state $s^j \in S_j$ is defined by a graph G_{s^j} which contains only the vertices and edges impacting the solution in further levels, with $G_{s^j} = (V_{s^j}, E_{s^j})$ where V_{s^j} is the set of vertices which are still not used in the partial solution (that we will refer to as *free vertices*) and a new vertex numbered 0 which represents all vertices already fixed in the linear arrangement. In the following, edges are denoted (x, y, w) where x and y are the vertices connected by the edge (where the order does not matter) and w the weight of the edge. At the root state $\hat{r}$, we have that $V_{\hat{r}} = V \cup \{0\}$ and $E_{\hat{r}} = E \cup \{(0, k, 0) \mid k \in V\}$ and at the terminal state $\hat{t}$, $V_{\hat{t}} = \{0\}$ and $E_{\hat{t}} = \emptyset$. States s with the same set V_s will have the exact same associated graph G_s in the exact decision diagram. Thus, we have 2^n different states corresponding to all subsets of V and $|S_j| = \binom{n}{j-1}$, since at level j, we have exactly $j-1$ vertices already assigned.

The transition functions t_j are defined by $t_j(s^j, u_j) = (V'_{s^j}, E'_{s^j})$, where $u_j \in V_{s^j}$ is the vertex placed in j-th position on the line, $V'_{s^j} = V_{s^j} \setminus \{u_j\}$ and

$$\begin{aligned} E'_{s^j} = &\left\{ (k, l, w) \mid k, l \in V'_{s^j} \setminus \{0\}, (k, l, w) \in E_{s^j} \right\} \\ &\cup \left\{ (0, k, w') \mid i \in V'_{s^j} \setminus \{0\}, (0, k, w) \in E_{s^j}, w' = w + w_{ku_j} \right\} \end{aligned} \tag{3.5}$$

which means that we only keep the edges between two free vertices, and for each free vertex an edge to the vertex 0 with a weight being the sum of the weights of the edges in the original graph from this vertex to vertices already used in the solution.

The transition cost functions h_j are defined by

$$h_j(s^j, u_j) = \sum_{k \in V'_{s^j} \setminus \{0\}} \sum_{l \in V \setminus V'_{s^j}} w_{kl} \qquad \text{with } (k, l, w_{kl}) \in E \tag{3.6}$$

$$= \sum_{k \in V'_{s^j} \setminus \{0\}} w_{0k} \qquad \text{with } (0, k, w_{0k}) \in E'_{s^j} \tag{3.7}$$

and this equality is proved in Lemma 3.2.2.

The root value is $v_r = 0$.

Lemma 3.2.2. *Given a state s^j at stage j in the dynamic programming formulation of the minimum linear arrangement problem, we have for each free vertex $k \in V_{s^j} \setminus \{0\}$ that the edge $(0, k, w_{0k}) \in E_{s^j}$ respects*

$$w_{0k} = \sum_{l \in V \setminus V_{s^j}} w_{kl}$$

with $(k, l, w_{kl}) \in E$.

Proof. According to the dynamic programming model described in Section 1.3, s^j is the result of $j-1$ transitions starting with the root state $\hat{r} = s^1$ and where $s^{i+1} = t_i(s^i, u_i)$ for $i = 1, \ldots, j-1$.

Equation (3.5) states that edges $(0, k, w) \in E_{s^i}$ and $(0, k, w') \in E_{s^{i+1}}$ follow the equality $w' = w + w_{ku_i}$. Furthermore, at the root state $\hat{r}$, we have $(0, k, 0) \in E_{\hat{r}}$ for $k = 1, \ldots, n$. By induction, we have for the edge $(0, k, w_{0k}) \in E_{s^j}$ that $w_{0k} = \sum_{i=1}^{j-1} w_{ku_i}$. Similarly, we have that $V_{s^{i+1}} = V_{s^i} \setminus \{u_i\}$ and for the root state, $V_{\hat{r}} = V \cup \{0\} = \{0, \ldots, n\}$ so we can induce that $V_{s^j} = \{0, \ldots, n\} \setminus \{u_1, \ldots, u_{j-1}\}$ and consequently,

$$\sum_{l \in V \setminus V_{s^j}} w_{kl} = \sum_{l \in \{u_1, \ldots, u_{j-1}\}} w_{kl} = \sum_{i=1}^{j-1} w_{ku_i} = w_{0k}$$

□

Theorem 3.2.3. *The specifications in Section 3.2 yield a valid dynamic programming formulation of the minimum linear arrangement problem.*

Proof. All permutations of the set of vertices are feasible solutions of the problem and for transition $t_j(s^j, u_j)$, the condition $u_j \in V_{s^j}$ ensures that the vertex u_j we are placing has not been used yet. Therefore, we only need to prove that condition (1.4) is respected.

The state transitions imply that $V_{s^{n+1}} = \{0\}$ and $E_{s^{n+1}}$, and thus $s^{n+1} = \hat{t}$. If we let $(\overline{s}, \overline{x})$ be an arbitrary solution, it remains to show that $\hat{f}(\overline{s}, \overline{x}) = f(\overline{x})$. We have $\hat{f}(\overline{s}, \overline{x}) = v_r + \sum_{j=1}^{n} h_j(\overline{s}^j, \overline{x}_j)$ and $v_r = 0$,

$$\begin{aligned}
\hat{f}(\overline{s}, \overline{x}) &= \sum_{j=1}^{n} h_j(\overline{s}^j, \overline{x}_j) \\
&= \sum_{j=1}^{n} \sum_{k \in V_{\overline{s}^{j+1}} \setminus \{0\}} \sum_{l \in V \setminus V_{\overline{s}^{j+1}}} w_{kl}
\end{aligned}$$

As seen in the proof of Lemma 3.2.2, if we know the sequence of transitions leading to a state, we can rewrite its set of free vertices as $V_{\overline{s}^{j+1}} = \{0, \ldots, n\} \setminus \{\overline{x}_1, \ldots, \overline{x}_j\}$ and since we also know the remaining transitions from $\overline{s}^{j+1}$ to $\overline{s}^{n+1}$, we can write $V_{\overline{s}^{j+1}} = \{0, \overline{x}_{j+1}, \ldots, \overline{x}_n\}$ and as a result

$$\begin{aligned}
\hat{f}(\overline{s}, \overline{x}) &= \sum_{j=1}^{n} \sum_{k \in \{\overline{x}_{j+1}, \ldots, \overline{x}_n\}} \sum_{l \in \{\overline{x}_1, \ldots, \overline{x}_j\}} w_{kl} \\
&= \sum_{j=1}^{n} \sum_{k \in \{\overline{x}_1, \ldots, \overline{x}_j\}} \sum_{l \in \{\overline{x}_{j+1}, \ldots, \overline{x}_n\}} w_{kl} \\
&= \sum_{k \in \{\overline{x}_1\}} \sum_{l \in \{\overline{x}_2, \ldots, \overline{x}_n\}} w_{kl} + \sum_{k \in \{\overline{x}_1, \overline{x}_2\}} \sum_{l \in \{\overline{x}_3, \ldots, \overline{x}_n\}} w_{kl} + \ldots + \sum_{k \in \{\overline{x}_1, \ldots, \overline{x}_{n-1}\}} \sum_{l \in \{\overline{x}_n\}} w_{kl} \\
&= \sum_{j=2}^{n} (j-1) w_{\overline{x}_1 \overline{x}_j} + \sum_{j=3}^{n} (j-2) w_{\overline{x}_2 \overline{x}_j} + \ldots + \sum_{j=n}^{n} (j-(n-1)) w_{\overline{x}_{n-1} \overline{x}_j} \\
&= \sum_{j=1}^{n} \sum_{j'=j+1}^{n} (j' - j) w_{\overline{x}_j \overline{x}_{j'}}
\end{aligned}$$

Given the solution $\overline{x}$, the corresponding bijection π is defined by $\pi\left(\overline{x}_i\right) = i$ for $i = 1, \ldots, n$. Thus, if the sums range over the vertices instead of the positions we get

$$\sum_{j=1}^{n} \sum_{j'=j+1}^{n} \left(j' - j\right) w_{\overline{x}_j \overline{x}_{j'}} = \sum_{j=1}^{n} \sum_{j'=j+1}^{n} \left|\pi(j) - \pi(j')\right| w_{jj'}$$

which is equivalent to equation (3.1). Finally, we can obtain an expression similar to equation (3.3) by introducing the same variables as in (3.2). □

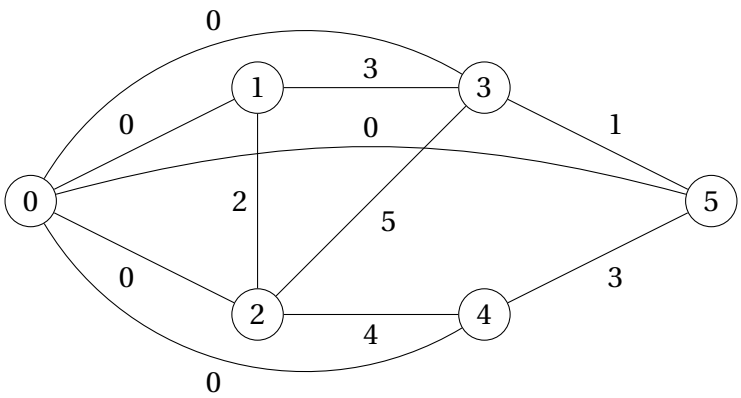


Figure 3.4: State representation for the root of the decision diagram.

Example 3.2.2. We will now compute the cost of the solution on Figure 3.2 incrementally with our dynamic programming formulation.

- At level 1, the state s^1 is represented by the graph displayed on Figure 3.4, which is simply the original graph where all the vertices have been connected to the new vertex labeled 0 by a zero-weight edge, so we have $V_{s^1} = V \cup \{0\} = \{0,1,2,3,4,5\}$ and $E_{s^1} = E \cup_{i=1}^{5} \{(0,i,0)\}$. We apply the transition $t_1(s^1, u_1 = 1)$ and obtain the graph of Figure 3.5a, where $V'_{s^1} = V_{s^1} \setminus \{1\} = \{0,2,3,4,5\}$ and E'_{s^1} has been computed according to equation (3.5) : we kept edges between vertices the free vertices $\{2,3,4,5\}$ and changed the cost of the edges adjacent to the vertex 0. For instance, we have that $w'_{02} = w_{02} + w_{12} = 0 + 2 = 2$ and $w'_{03} = w_{03} + w_{13} = 0 + 3 = 3$. We can now compute the transition cost :

$$h_1(s^1, u_1) = \sum_{\substack{i \in V'_{s^1} \setminus \{0\} \\ (0,i,w_{0i}) \in E'_{s^1}}} w_{0i} = w_{02} + w_{03} = 2 + 3 = 5$$

- At level 2, we apply the transition $t_2(s^2, u_2 = 2)$ to obtain the graph of Figure 3.5b, with $V'_{s^2} = V_{s^2} \setminus \{2\} = \{0,3,4,5\}$ where $w'_{03} = w_{03} + w_{23} = 3 + 5 = 8$ and $w'_{04} = w_{04} + w_{24} = 0 + 4 = 4$.

$$h_2(s^2, u_2) = \sum_{\substack{i \in V'_{s^2} \setminus \{0\} \\ (0,i,w_{0i}) \in E'_{s^2}}} w_{0i} = w_{03} + w_{04} = 8 + 4 = 12$$

- At level 3, we apply the transition $t_3(s^3, u_3 = 3)$ to obtain the graph of Figure 3.5c, with $V'_{s^3} = V_{s^3} \setminus \{3\} = \{0,4,5\}$ where $w'_{05} = w_{05} + w_{35} = 0 + 1 = 1$.

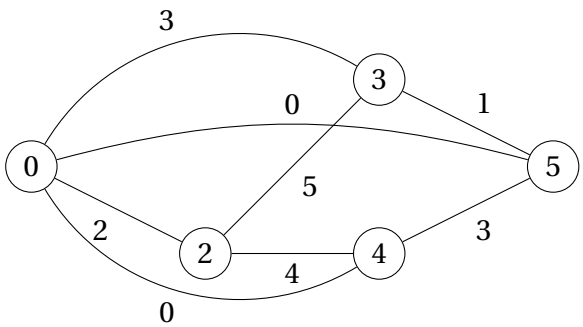
$$h_3(s^3, u_3) = \sum_{\substack{i \in V'_{s^3} \setminus \{0\} \\ (0,i,w_{0i}) \in E'_{s^3}}} w_{0i} = w_{04} + w_{05} = 4 + 1 = 5$$

- At level 4, we apply the transition $t_4(s^4, u_4 = 4)$ to obtain the graph of Figure 3.5d, with $V'_{s^4} = V_{s^4} \setminus \{4\} = \{0,5\}$ where $w'_{05} = w_{05} + w_{45} = 1 + 3 = 4$.

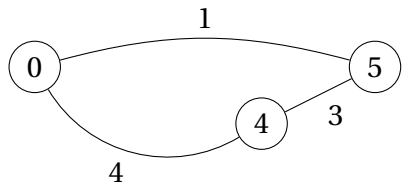
$$h_4(s^4, u_4) = \sum_{\substack{i \in V'_{s^4} \setminus \{0\} \\ (0,i,w_{0i}) \in E'_{s^4}}} w_{0i} = w_{05} = 4$$

- At level 5, we apply the transition $t_5(s^5, u_5 = 5)$ and obtain a single node : $V'_{s^5} = V_{s^5} \setminus \{5\} = \{0\}$, so we have :

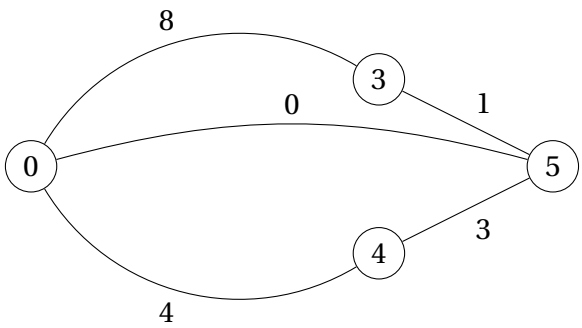
$$h_5(s^5, u_5) = \sum_{\substack{i \in V'_{s^5} \setminus \{0\} \\ (0,i,w_{0i}) \in E'_{s^4}}} w_{0i} = 0$$



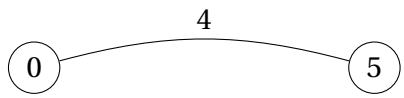
(a) State graph after $t_1(s^1, u_1 = 1)$.



(c) State graph after $t_3(s^3, u_3 = 3)$.



(b) State graph after $t_2(s^2, u_2 = 2)$.



(d) State graph after $t_4(s^4, u_4 = 4)$.

Figure 3.5: Evolution of the state graph for the arrangement of Figure 3.2.

We can now add all transition costs together and verify that we obtain the same result as before.

$$\begin{aligned}\sum_{j=1}^{5} h_j(s^j, u_j) &= h_1(s^1, u_1) + h_2(s^2, u_2) + h_3(s^3, u_3) + h_4(s^4, u_4) + h_5(s^5, u_5) \\ &= 5 + 12 + 5 + 4 + 0 = 26\end{aligned}$$

3.3 Relaxation Formulation

We are facing a minimization problem so our goal is to compute a lower bound to the objective function. By only adding weights of edges that all states to be merged share, we ensure that the path lengths will be at most equal to the corresponding path lengths in the exact decision diagram. A valid merging operator is to take the intersection of the free vertices of all states to merge so that in further transitions, we only account for edges and vertices that all states shared. However, they will most of the time have different sets of free vertices so we will quickly lose a lot of the additional costs. We can try to do better by forgetting about the labels of the free vertices and try to preserve the structure of the graph. We can observe in equations (3.5) and (3.7) that transition costs are only affected by the weight of the edges in the state graph. Thus, our goal is to find a subgraph of all state graphs such that we maximize the total edge weight (some edges will have a higher impact on the objective function (i.e. a bigger coefficient) but we do not know in advance which edge will be useful in interesting configurations). For two graphs, this problem is called the *maximum weight common subgraph*, which is a generalization of the NP-complete *maximum subgraph matching* problem [13, problem GT50]. We will therefore use a heuristic approach to find good a subgraph.

We will now define the merging operator $\oplus(M)$, where $M = \{s_1, s_2, \ldots, s_m\}$ is the subset of states to be merged. As said earlier, we will find a common subgraph of all state graphs. Formally, we define the bijective functions $\phi_l : V_{s_1} \to V_{s_l}, l = 2, \ldots, m$ which represent the mapping from the free vertices of state s_1 to the free vertices of state s_l. We impose $\phi_l(0) = 0, l = 2, \ldots, m$ since the vertex 0 plays a special role in the state graphs. It follows that

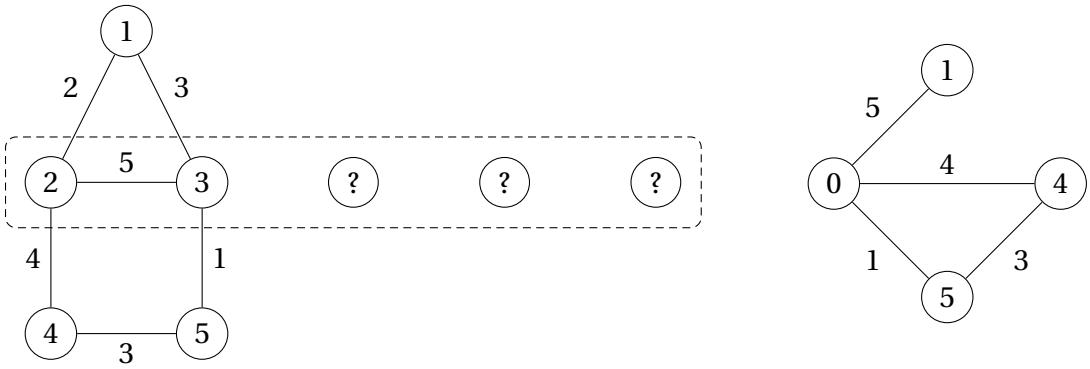
$$\oplus(M) = s_M \text{ with } G_{s_M} = \left(V_{s_M}, E_{s_M}\right) \text{ and } V_{s_M} = V_{s_1}$$

and where

$$E_{s_M} = \left\{ (i, j, w) \;\middle|\; \begin{array}{l} (i, j, w_1) \in E_{s_1}, \\ (\phi_l(i), \phi_l(j), w_l) \in E_{s_l} \;\forall l \in \{2, \ldots, n\}, \\ w = \min_{l=1}^{m} \{w_l\} \end{array} \right\}$$

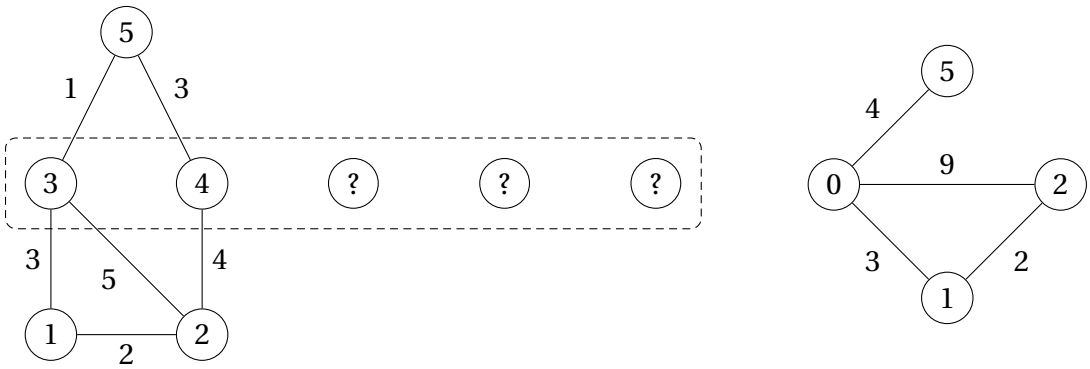
The transition cost is not changed, such that $\Gamma_M(v, u) = v$ for all v, u.

We explained how the resulting state graph is built but not how we find the mappings ϕ_l. We use a very simple heuristic : for each state graph, we sort the vertices by degree in decreasing order. Between vertices with the same degree, we sort them in decreasing order according to the total weight of their incident edges. We then map vertices of each state that are at the same position in their respective sorted list.



(a) Partial linear arrangement corresponding to s_1.

(b) State graph of s_1.



(c) Partial linear arrangement corresponding to s_2.

(d) State graph of s_2.

Figure 3.6: Two different states for the graph of Figure 3.1.

Example 3.3.1. We illustrate the concepts introduced in this section by showing the result of a merge operation on two partial arrangements displayed on Figure 3.6a and 3.6c, respectively referred as s_1 and s_2. The dashed rectangle represents the linear arrangement, in this case two vertices are already placed.

Table 3.1 shows the free vertices of each state, sorted as explained before. We then create the mapping by pairing vertices on the same line in the table, so we have $\phi_2(4) = 2, \phi_2(5) = 1, \phi_2(1) = 5$.

s_1	Degree	Weight	Index
	2	7	4
	2	4	5
	1	5	1

s_2	Degree	Weight	Index
	2	11	2
	2	5	1
	1	4	5

Table 3.1: Sorted list of the vertices of the state graphs from Figure 3.6a and 3.6c.

Finally, we compute the set of edges of the resulting state s_M as shown in Table 3.2 and derive the resulting state graph G_{s_M} showed on Figure 3.7.

$(i, j, w_1) \in E_{s_1}$	$(\phi_2(i), \phi_2(j), w_2) \in E_{s_2}$	$(i, j, \min\{w_1, w_2\}) \in E_{s_M}$
(0,1,5)	(0,5,4)	(0,1,4)
(0,4,4)	(0,2,9)	(0,4,4)
(0,5,1)	(0,1,3)	(0,5,1)
(4,5,3)	(2,1,2)	(4,5,2)

Table 3.2: Computation of the edges of the merged state.

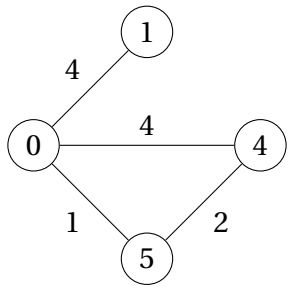


Figure 3.7: Resulting state graph after merging the states of Figure 3.6a and 3.6c.

3.4 Computational Results

We will now evaluate the performances of our formulation for the MinLA by comparing them with a recent MIP model. The instances are generated using the Nugent distance matrices [24] available from QAPLIB [10]. From the Nugent distance matrix of size 20, we remove the last rows and columns to obtain matrices of size 10 to 20 (instances with more vertices can be obtained with the Nugent distance matrix of size 30). Then, for a given parameter t, we replace each element by 0 if it equals the value t and by 1 otherwise. This leads to adjacency matrices representing dense graphs.

The MIP model was presented in [4, MIP2] and is solved here with Gurobi Optimizer 8.1.0 callable library [14] using Java language. Our MDD model is solved using our decision diagram library implemented in Java and described in Chapter 4. The tests were performed on an Intel Core i5, 3GHz PC with 8Go of RAM and the Windows 7 operating system with a single thread (otherwise using default settings for Gurobi).

In Table 3.3, we show the number of vertices n and the value t used to generate the adjacency matrix from the Nugent distance matrix. We also show the width W_B of the exact decision diagram for the instance and the optimal value $z^\star$. Then, we have the total time required to solve each instance to optimality with a time limit of 1 hour. For the MDD, we used four different maximum widths strategies : for each instance, the maximum width is set to a fraction of W_B. We used the last exact layer cutset, which proved to be more efficient on this problem, and used the default rank function based on the longest-path value of the nodes. We tested a modified rank function including the sum of the weights of the remaining edges in the state graph but it did not lead to substantially better results.

From the results showed in Table 3.3, we can clearly state that the decision diagram approach outperforms the MIP model, given that we use decision diagrams with a sufficient maximum width. It is impressive given that Gurobi Optimizer is a state-of-the-art mathematical programming solver while our implementation of the MDD-based solver is quite basic.

Instance				MIP	MDD Time (s)			
n	t	W_B	$z^\star$	Time (s)	$W_B/2$	$W_B/4$	$W_B/8$	$W_B/16$
10	5	252	149	9	1	1	1	1
11	5	462	186	12	1	1	1	2
12	5	924	241	100	1	1	2	4
13	5	1716	314	223	1	1	7	10
14	5	3432	391	631	3	2	7	34
15	5	6435	474	1717	12	9	71	98
16	6	12870	629	*	34	29	219	565
17	6	24310	748	*	105	87	85	2382
18	6	48620	896	*	322	320	248	*
19	6	92378	1049	*	1116	1008	727	*
20	5	184756	1076	*	3050	3507	3028	*

* Execution aborted after 1 hour was reached.

Table 3.3: Comparison of MDD and MIP models for dense graphs with $10 \leq n \leq 20$.

Although the decision diagrams with maximum width $W/16$ are clearly slower, the performances of the three other maximum width strategies are surprisingly similar. To better understand the influence of the maximum width, we will compare the gap and time spent after the first lower and upper bound computation at the root state (that is, after having built one restricted and one relaxed decision diagram) for different maximum widths and graph sizes. We performed these tests on random graphs with 16 to 21 vertices with a similar density to the instances of Table 3.3 (around 0.9). For each graph size, we generated five different random instances and we show on Figure 3.8 the average gap and run time recorded on those five instances for each maximum width strategy.

From the left plot, we can observe that the quality of the bounds seem to remain fairly constant as the number of vertices increases. On the right plot (note that the y-axis is in log-scale), we discover an empirical relation where the time needed to compute these bounds is multiplied roughly by 2.5 for each vertex that we add. In conclusion, we can say that the quality of the bounds of our formulation scale well with the number of vertices, the efficiency of their computation should however be improved to avoid an exponentially increasing execution time.

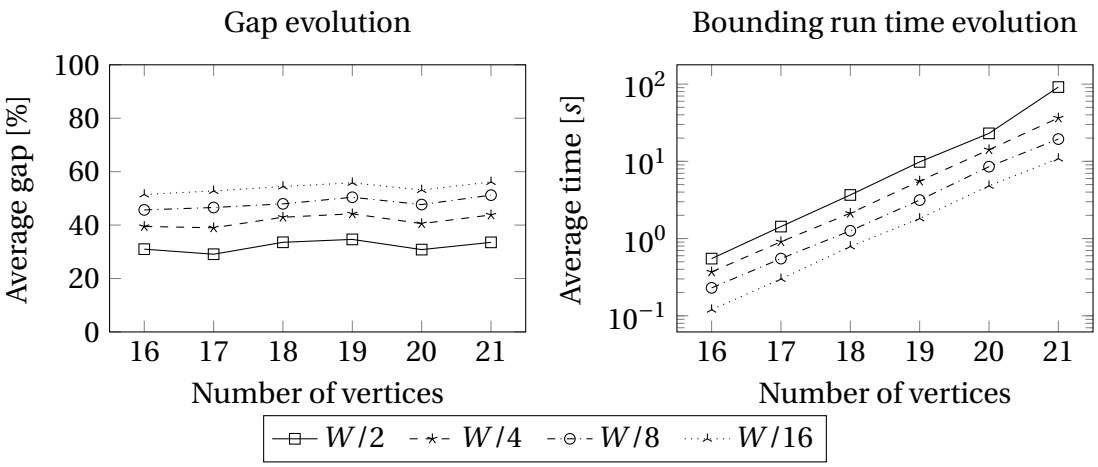


Figure 3.8: Comparison of the average gap and time spent after the first lower and upper bound computation for different maximum widths.

Chapter 4

Implementation

In this chapter, we present our Java implementation of the discrete optimization solver described in Chapter 1. We will first overview the classes representing the components of decision diagrams and then give an example of how one can use this tool to program and solve a new problem. Here is the link of the GitHub repository with the complete code :

https://github.com/vcoppe/mdd-solver

along with the Javadoc, made available at **https://vcoppe.github.io/mdd-solver**.

4.1 Overview

We will start with the simplest classes and then gradually build the more complex ones. Note that the code snippets of this chapter only include methods and details considered relevant for our explanation.

The class `Variable` represents a variable in a minimalist way, with one field `id` corresponding to the index j of a variable x_j in the model of a problem. The second field `value` will contain the value assigned to the variable. Besides the constructors, setter and getter methods, we have a method `copy` that creates another `Variable` object with the same fields. The method `newArray` returns an array of `n` `Variable` objects with `id` ranging from `0` to `n-1`.

```java
public class Variable {

    public int id;
    private double value;

    public Variable( ... ) { ... }

    public Variable copy() { ... }

    public void assign(double value) { ... }

    public static Variable[] newArray(int n) { ... }

    ...
}
```

Listing 4.1: The class `Variable`.

Next, we have the interface `State` showed on Listing 4.2. Classes implementing this interface will contain the representation of the states s^j of the dynamic programming formulation of a problem. Since this representation varies from one problem to another, its implementation is left for each particular use (see Section 4.2 for an example). As we will see later, we use `HashMap` objects to represent the layers of the decision diagrams and identify whether an equivalent node is already in the layer. `State` objects are used as keys of these hash tables so the methods `hashCode` and `equals` should contain key information to detect equivalent states and provoke collisions between them. For equivalent states, the method `hashCode` should return the same result and the method `equals` should return `true`. The method `rank` is used to compare nodes before deleting or merging a subset of them when building restricted or relaxed decision diagrams (see Section 1.8). Finally, the method `copy` should return another object with an equivalent state.

```
public interface State {

    int hashCode();

    boolean equals(Object obj);

    double rank(Node node);

    State copy();

}
```

Listing 4.2: The interface `State`.

The class `Node` represents nodes of the decision diagram. With $p^\star = \left(a^{(1)}, \ldots, a^{(j-1)}\right)$ being the longest-path to a node, the important fields of an object representing this node are : the associated state s^j, an array of variables containing the assignment up to that node $x_k = d\left(a^{(k)}\right)$ for $k = 1, \ldots, j-1$, the longest-path value $v\left(p^\star\right)$, whether the node is exact and the layer number j. When two equivalent states are detected in a layer, we can combine them by using the method `update` which will preserve the information of the node with greater longest-path value. The method `assign` sets the variable identified by `id` to the given `value`. It is used in the method `getSuccessor`, which returns the node at the opposite end of the arc $a^{(j)}$, with state s^{j+1} given by `state`, path length $v(p^\star) + v\left(a^{(j)}\right)$ given by `value`, variable x_j identified by `id` and arc label $d\left(a^{(j)}\right)$ given by `val` representing the assignment $x_j = d\left(a^{(j)}\right)$. It transfers important properties such as the partial assignment of the variables, the layer number and the exact property but does not compute the actual transition.

```
public class Node<R extends State> implements Comparable<Node> {

    public R state;
    public final Variable[] variables;
    private double value;
    private boolean exact;
    private int layerNumber;
    ...

    public Node( ... ) { ... }

```

```
    public void update(Node<R> other) { ... }

    public void assign(int id, int value) { ... }

    public Node<R> getSuccessor(R state, double value, int id, int val) { ... }

    ...
}
```

Listing 4.3: The class `Node`.

We explained previously that the interface `State` needs to be implemented for each problem according to the state representation of its dynamic programming model. The interface `Problem` showed below is the second component which has to be adapted to every problem and it will contain the rest of the information about the problem, mainly the state transitions $t_j(s^j, x_j)$ and $h_j(s^j, x_j)$ and the merging operator $\oplus(M)$. The two first methods are very simple : `root` should return a node with the root state $\hat{r}$ of the problem and `nVariables` should return the number of variables n of the problem. Given a node and an unbound variable, the method `successors` should return a list of nodes that we reach by assigning every possible value to the variable. The method `merge` takes an array of nodes and should return a single node resulting from a valid merging operation.

```
public interface Problem<R extends State> {

    Node root();

    int nVariables();

    List<Node> successors(Node<R> node, Variable var);

    Node merge(Node<R>[] nodes);

}
```

Listing 4.4: The interface `Problem`.

Back to our decision diagram components, the class `Layer` represents their layers L_j by using a `HashMap` where the nodes are identified with the corresponding state, as explained previously. The fields include the number of the layer j in the decision diagram and the exact property. The method `nextLayer` returns the next layer in the decision diagram, choosing the next variable to assign x_j and computing every possible transition $d \in D(x_j)$ from the nodes of the current layer. The parameter `relaxed` indicates whether we are building a relaxed or restricted decision diagram : if the size of the new layer exceeds the parameter `width`, nodes are merged or removed accordingly.

```
public class Layer {

    private HashMap<State, Node> nodes;
    private boolean exact;
    public int number;
    ...

    public Layer( ... ) { ... }

```

```
    public Layer nextLayer(int width, boolean relaxed) { ... }

    ...
}
```

Listing 4.5: The class `Layer`.

The class `MDD` helps building multi-valued decision diagrams by generating layers one by one. With the method called `setInitialNode`, one can start building a decision diagram from a node with a partial assignment and taking into account the longest-path value up to that node as root value v_r. Methods `solveRelaxed` and `solveRestricted` respectively implement Algorithm 2 and 3, returning the terminal node of the decision diagram generated along with its longest-path value. In Section 1.8, we detailed two possible exact cutsets to be used in the branch-and-bound algorithm. These exact cutsets are both implemented and can be used interchangeably in the method `exactCutset` by modifying the variable `LELcutset`.

```
public class MDD {

    private Layer root;
    private boolean exact;
    public boolean LELcutset = true;
    ...

    public MDD( ... ) { ... }

    public void setInitialNode(Node initialNode) { ... }

    public Node solveRelaxed(int width) { ... }

    public Node solveRestricted(int width) { ... }

    public Collection<Node> exactCutset() { ... }

    ...
}
```

Listing 4.6: The class `MDD`.

Finally, the class `Solver` implements the branch-and-bound algorithm (Algorithm 4) in the method `solve` using the class `MDD` to compute bounds with restricted and relaxed decision diagrams. Method `setWidth` can be used to set the maximum width of the decision diagrams generated.

```
public class Solver {

    private Problem problem;
    private MDD mdd;

    public Solver( ... ) { ... }

    public Node solve(int timeLimit) { ... }

    public void setWidth(int width) { ... }

```

```
    ...
}
```

Listing 4.7: The class `Solver`.

4.2 Implementing A New Problem

In Chapter 1, we presented the decision diagram framework for discrete optimization and illustrated it with the *Unbounded Knapsack Problem*. We will now go through the implementation of this particular problem in our Java solver, which is realized through two interfaces.

The first one is the interface `State` showed on Listing 4.2. Listing 4.8 shows how this interface is implemented for the knapsack problem. The states are the remaining capacity so we only need to store an integer `capacity` in each state. In this case, two states (of a same layer) are equivalent if their remaining capacities are equal so the methods `hashCode` and `equals` are very straightforward. Finally, the method `rank` is chosen to return the longest-path value of the corresponding node, as explained in Section 1.8.

```java
private class KnapsackState implements State {

    int capacity;

    KnapsackState(int capacity) { this.capacity = capacity; }

    public double rank(Node node) { return node.value(); }

    public int hashCode() { return capacity; }

    public boolean equals(Object o) {
            return o instanceof Knapsack.KnapsackState
                && capacity == ((KnapsackState) o).capacity;
    }

    public KnapsackState copy() { return new KnapsackState(capacity); }

}
```

Listing 4.8: Implementation of the `State` interface for the knapsack problem.

We now discuss the implementation of the interface `Problem` (see Listing 4.4) for the knapsack problem. The code is displayed in Listing 4.9 where `n` is the number of different items, `c` is the capacity of the knapsack and the arrays `w` and `v` respectively contain the weight and value of the items. In the constructor, we instantiate the root node of the decision diagram with 3 parameters : the associated state which has a remaining capacity equal to `c`, the variables in the number of `n` and the root value which is set to `0`.

In `successors`, we first retrieve the index of the variable we are considering (since variables can be assigned in a different order in order to reduce the size of the decision diagram) and the remaining capacity contained in the state of the given node. Then, we iterate over each feasible quantity of item `i` that we can still put in the knapsack (from `0` to `knapsackState.capacity / w[i]`) and add the corresponding successor to the list.

For each of them, a new state is created with the new remaining capacity and the new value of the knapsack is computed.

```
public class Knapsack implements Problem<Knapsack.KnapsackState> {

    private int n, w[];
    private double[] v;
    private Node<KnapsackState> root;

    Knapsack(int n, int c, int[] w, double[] v) {
        this.n = n;
        this.w = w;
        this.v = v;

        root = new Node<>(new KnapsackState(c), Variable.newArray(n), 0);
    }

    public Node<KnapsackState> root() { return root; }

    public int nVariables() { return n; }

    public List<Node> successors(Node<KnapsackState> node, Variable var) {
        int i = var.id;
        KnapsackState knapsackState = node.state;
        List<Node> successors = new LinkedList<>();

        for (int x = 0; x <= knapsackState.capacity / w[i]; x++) {
            KnapsackState succKnapsackState =
                new KnapsackState(knapsackState.capacity - x * w[i]);
            double value = node.value() + x * v[i];
            successors.add(node.getSuccessor(succKnapsackState, value, i, x));
        }

        return successors;
    }

    public Node merge(Node<KnapsackState>[] nodes) {
        Node<KnapsackState> best = nodes[0];
        int maxCapacity = 0;

        for (Node<KnapsackState> node : nodes) {
            maxCapacity = Math.max(maxCapacity, node.state.capacity);

            if (node.value() > best.value()) best = node;
        }

        best.state.capacity = maxCapacity;
        return best;
    }

    public class KnapsackState implements State { ... }
}
```

Listing 4.9: Implementation of the `Problem` interface for the knapsack problem.

In the method `merge`, we implement the merging operation described in Example 1.5.1. We loop over the nodes to merge, store the largest remaining capacity and keep track of the node with the longest path value. We then replace its capacity by the largest we found and return that node.

Conclusion

In Chapter 1, we summarized the theory behind the decision diagram based discrete optimization solver presented in [7, 8]. We then explained in Chapter 2 how three well-known problems were successfully formalized with this technique. After these opening chapters, we detailed our two contributions.

We presented in Chapter 3 a new formulation of the minimum linear arrangement problem with an original relaxation operator. In Section 3.4, we discussed our computational results and have showed that our model and implementation outperform the linear programming method, solved using a state-of-the-art mathematical programming solver. Our formulation for the MinLA has been designed incrementally throughout the period of research and although its current form led to good results, there is still room for improvement. Namely, the heuristic for finding a good subgraph of the state graphs in the merging operation directly impacts the quality of the bounds but is currently very basic. However, the fact that we are potentially dealing with more than two graphs at the time makes the problem challenging especially as we cannot afford a high computational cost since this operation is used in almost each layer of every relaxed decision diagram generated during the branch-and-bound algorithm. Another possible improvement would be to find a more compact state variable for this problem, they are indeed quite costly to store in the memory as they contain a modified graph. In the code, we take care of using a minimal amount of information to keep track of these modified graphs but in the worst case, we will still have a whole adjacency matrix to save.

Our second contribution is to provide a base implementation of the complete decision diagram based solver with tools allowing to program new problems with simple and limited amount of code. This hopefully brings a convenient way to experiment on decision diagrams for further research. A general presentation of the organization of the code was made in Chapter 4 together with a worked example of how to implement a new problem using the library. Concerning the future developments of this tool, we did some effort to limit the number of unnecessary copies of objects and reduced the amount of nested loops to some extent but it is clear that the efficiency of the code could be improved speed and memory-wise.

References

[1] D. Adolphson. Single machine job sequencing with precedence constraints. *SIAM Journal on Computing*, 6:40–54, 1977.

[2] D. Adolphson and T. C. Hu. Optimal linear ordering. *SIAM Journal on Applied Mathematics*, 25(3):403–423, 1973.

[3] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(06):509–516, 1978.

[4] A. R.S. Amaral. A mixed 0-1 linear programming formulation for the exact solution of the minimum linear arrangement problem. *Optimization Letters*, 3(4):513–520, 2009.

[5] H. R. Andersen, T. Hadžić, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer, 2007.

[6] M. Behle. *Binary decision diagrams and integer programming*. PhD thesis, Max Planck Institute for Computer Science, 2007.

[7] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. N. Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[8] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. Technical report, Carnegie Mellon University, 2001.

[10] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB – a quadratic assignment problem library. *Journal of Global optimization*, 10(4):391–403, 1997.

[11] A. Caprara, A. N. Letchford, and J.-J. Salazar-González. Decorous lower bounds for minimum linear arrangement. *INFORMS Journal on Computing*, 23(1):26–40, 2011.

[12] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34:313–356, 2002.

[13] M.R. Garey, D.S. Johnson, and M. S. Mahoney Collection. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Books in mathematical series. W. H. Freeman, 1979.

[14] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[15] G.D. Hachtel and F. Somenzi. A symbolic algorithms for maximum flow in 0-1 networks. *Formal Methods in System Design*, 10(2):207–219, 1997.

[16] T. Hadžić and J. N. Hooker. Postoptimality analysis for integer programming using binary decision diagrams. Technical report, Carnegie Mellon University, 2006.

[17] T. Hadžić and J. N. Hooker. Cost-bounded binary decision diagrams for 0-1 programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 84–98. Springer, 2007.

[18] L. H. Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 12(1):131–135, 1964.

[19] S. B. Horton. *The Optimal Linear Arrangement Problem: Algorithms And Approximation*. PhD thesis, Georgia Institute of Technology, 1997.

[20] A. J. Hu. *Techniques for efficient formal verification using binary decision diagrams*. PhD thesis, Stanford University, Department of Computer Science, 1995.

[21] Y. Koren and D. Harel. A multi-scale algorithm for the linear arrangement problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 296–309. Springer, 2002.

[22] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):959–975, 1994.

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

[24] C. E. Nugent, T. E. Vollmann, and J. Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Operations research*, 16(1):150–173, 1968.

[25] J. Petit. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics*, 8, 2003.

[26] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single-processor scheduling and interval graph completion. In M. R. J. Leach and B. Monien, editors, *Automata, Languages and Programming*, pages 751–762. Springer, 1991.

[27] I. Wegener. Branching programs and binary decision diagrams: Theory and applications. *Discrete Applied Mathematics*, 2000.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**