TU DELFT

MASTER THESIS

Decision diagrams for decomposed mixed integer linear programs

Author:
Koos van der Linden
4133145

Supervisor:
Dr. Mathijs de Weerdt

Other members of the thesis committee:
Dr. Matthijs Spaan
Dr. Ir. Leo van Iersel

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science

in the

Algorithmics Research Group
Department of Software and Computer Technology

August 30, 2017

TU Delft

TU DELFT

Abstract

Faculty Electrical Engineering, Mathematics and Computer Science
Department of Software and Computer Technology

Master of Science

Decision diagrams for decomposed mixed integer linear programs

by Koos van der Linden

A recent development in the field of discrete optimization is the combined use of (binary) decision diagrams (DD) and branch and bound for optimization. This method has been shown to outperform integer linear programming on several classic problems. The performance of DDs in integer optimization raises the question if this method can be extended to solve mixed integer problems (MIP), because of the prevalence of MIP modelling. This thesis is an effort to answer that question.

Currently DD-based optimization does not allow for continuous variables. We propose a method that uses Benders Decomposition to divide MIP problems into a discrete master problem and a continuous subproblem. DD-based optimization is used to solve the master problem. The subproblem can be solved by linear programming.

The results presented in this thesis show that our method can be used for MIP problems whose integer variables play a dominant role, and are hard to solve by MIP solvers. An advantage of our method is that it provides feasible solutions as early as the first iteration.

Acknowledgements

Six years ago I started to study computer science in Delft for this reason. With algorithms a person can use his intelligence and creativity to exploit the speed and power of a computer. Now at the end of my thesis project, I can look back with a grateful heart. My thanks is mostly directed to God, or as one would say in the past: Soli Deo Gloria. Creativity, inspiration and motivation are critical in a project such as this, and for me He is the source of all three. What helped me especially, was His encouragement to work wholeheartedly on the project as if it were for Him.

My motivation during this project may very well be described by a cosine function, with ups and downs. My daily supervisor Mathijs de Weerdt has always been positive during the whole project. His positive attitude was of great help, and quite often changed the tide of the cosine. I want to thank him for that. He has helped me a lot during our frequent meetings, and by all the feedback that he gave to me.

A special thanks is also directed to Willem-Jan van Hoeve, one of the authors of the book on Decision Diagrams that I frequently refer to. His critical feedback and profound knowledge of decision diagrams helped in assessing the contribution of this thesis. I want to thank him for the time he spent in providing feedback.

I also want to thank Jan Elffers, Chris van Vliet and Germán Morales-España for their effort, for the feedback that they gave me and for helping me during the process. In the same way I want to thank Matthijs Spaan and Leo van Iersel for partaking in the thesis committee.

At last I want to thank my family and friends and the Navigators community. They have supported me in all other ways.

Koos van der Linden
Delft, August 2017

Contents

1 Introduction **1**

2 Background **5**
- 2.1 Decision Diagrams for Optimization 5
  - 2.1.1 Formal description 6
  - 2.1.2 Dynamic Programming Formulation 7
  - 2.1.3 Compilation of a Decision Diagram 7
- 2.2 Branch and bound with decision diagrams 9
  - 2.2.1 Restricted Decision Diagrams 9
  - 2.2.2 Relaxed Decision Diagrams 10
  - 2.2.3 Branch and Bound procedure 11
- 2.3 Threshold diagram 13
- 2.4 Benders Decomposition 14

3 Decision Diagrams with Benders Decomposition **17**
- 3.1 Decision Diagrams and Benders Decomposition 18
  - 3.1.1 Cuts using the threshold diagram 20
  - 3.1.2 Finding an exact cutset 21
- 3.2 Cuts on the optimal value 24

4 Evaluation **27**
- 4.1 Threshold diagram 27
  - 4.1.1 Threshold versus Custom 28
  - 4.1.2 Number of constraints 29
- 4.2 Cost tuples 30
- 4.3 Decomposition 31
  - 4.3.1 Scheduling with independent sets 32
  - 4.3.2 Market Share Split (adapted) 35
  - 4.3.3 Other problems 38

5 Discussion **39**
- 5.1 Conclusions 39
- 5.2 Recommendations 41

A Implementation details **43**
- A.1 State description 43
- A.2 Variable ordering 44

Bibliography **45**

Chapter 1

Introduction

Decision diagrams for optimization by using branch and bound is a novel approach in discrete optimization. It has been proved that decision diagram based solvers can outperform state of the art integer programming solvers for several classic problems, such as independent set, vertex cover, the maximum cut problem, and the maximum 2-satisfiability problem [5, 6]. These results in integer optimization raise the question if this method can be extended to solve mixed integer problems.

Mixed integer programming (MIP) can be used to model all kinds of problems. One can think of problems such as the map labeling problem, optimal transmission switching and scheduling problems. These kind of problems occur in numerous fields, for example in logistic companies, chemical process optimization, and power engineering. Optimal solutions for these problems help to increase efficiency, for example in reducing traffic time and reducing system cost.

However, integer programming and, as a consequence, mixed integer programming is an NP-hard problem. Therefore some of the MIP problem instances are hard to solve. A more efficient algorithm in terms of runtime would allow for solving larger instances of a difficult problem, or would allow for more complex models.

A decision diagram (DD) is a graphical representation of a discrete function. The paths from root to terminal in the diagram represent all the feasible assignments in an optimization problem. The weights of the edges are the contribution to the objective. The length of the path is the objective value of a solution, and therefore the shortest path (in case of minimization) is the optimal feasible solution. By using restricted and relaxed versions of a decision diagram, lower and upper bounds on the optimal solution can be computed rapidly. Because of this ability to find bounds, decision diagrams can be used in a branch and bound procedure. This allows for solving large discrete optimization problems.

There are several advantages of DD-based optimization. Firstly, it may provide tighter bounds (than provided by a continuous relaxation). Secondly, these bounds and (a first) feasible solution can be computed rapidly. Thirdly, it does not require an inequality formulation (in some cases the inequality formulation is exponential in the problem size). With DD-modeling the recursive structure of a problem is used instead. Lastly, DD-modeling can also deal with non-linear and non-convex constraints, which means that linearization is not required.

DDs can (currently) only be used to solve integer programs. However, real world applications may contain continuous decision variables which cannot be discretized, or whose discretization provides a bad approximation. Consider the example MIP problems mentioned above. Also, some problems with continuous decision variables are clearly an extension of a typical integer problem, such as the asymmetric travelling salesman problem with time windows.

MIP problems are mostly solved by mixed integer programming using branch and bound. This technique is based on a relaxation of the integrality constraints. This relaxed problem can then be solved as a Linear Program (LP). Solving an LP can be done in polynomial time. The relaxed LP solutions are used in the branch and bound. The performance of this method therefore relies mainly on the quality of the relaxation. The integer variables, and the constraints on the integer variables are the main reason that these problems are hard to solve. Especially for problems where the integer variables play a dominant role and where the LP relaxation of a problem is bad, another method that does not rely on LP relaxation could be useful.

Because of the good performance of DDs in discrete optimization, it is interesting to see what contribution DDs can offer for mixed integer problems. The question, however, that needs to be answered is how to deal with the continuous variables and constraints. A DD can only describe a finite amount of assignments to the variables and therefore can not trivially deal with continuous variables.

One possible option could be to extend the use of Interval Automata (IA) [16] (or multi-data type interval decision diagrams [15]). IAs are a generalization of DDs that can have both discrete and continuous variables. At this moment (at least to our knowledge), IAs can only be used for decision problems, and not yet for optimization. The possibility of this option has been tested during the research for this thesis, but no successful results can yet be reported. IA-based optimization however, is still recommended for further research.

Another promising option, which is explored successfully in this thesis, is to use a decomposition technique that separates the problem into a discrete problem and a continuous problem. This means that the decomposition splits the set of decision variables.

Benders decomposition [3] is a technique that achieves precisely this. Benders decomposition (BD) is often used successfully for decomposing large problems and can also be used to decompose a mixed problem into a discrete master problem and a continuous subproblem (see for example [9]). BD is a constraint generating decomposition that iteratively updates the master problem until the optimal solution to the master problem is also optimal to the subproblem.

However, the contribution of continuous variables to the objective results in a (convex) piecewise linear objective when translated into constraints in the master problem by Benders decomposition. The question that needs to be answered now, is how to deal with a piecewise linear objective in a decision diagram. This same question may also arise when dealing with other problems. Piecewise linear objectives are also often used in practice for example to describe more accurately a cost function.

The method that is proposed in this thesis includes a solution to this problem. To be able to deal with a piecewise objective, the decision diagrams use cost tuple labels, instead of just scalar cost labels. Each element in the cost tuple represents a piece of the objective.

A decomposition for DDs that divides the set of constraints is also proposed in [4]. In this article the Lagrangian relaxation is used to relax a subset of the constraints. These relaxed constraints are considered by solving the Lagrangian dual. A notable difference between this decomposition and BD is that BD also divides the set of variables. Apart from that, the method proposed in this thesis also presents how a DD-based branch and bound can work even when the set of constraints describing the DD is a subset of all the constraints.

The research question is whether decision diagrams in combination with Benders decomposition can be used as a method for solving mixed integer problems, and if so, for what kinds of problems this combination of DD-based optimization and Benders decomposition can be used. Because the method is proposed as a general-purpose solver, it is compared with mixed integer programming solvers. Relevant subquestions are the following: 1) How do the threshold diagrams (used for generated constraints) perform? 2) How does the idea of cost tuples perform for an integer problem with a convex piecewise linear objective? 3) What number and kind of constraints are generated in the decomposition? 4) What elements contribute to the runtime of the proposed method?

This thesis has two main contributions. The first contribution is the idea to combine DD-based optimization with Benders Decomposition (BDDD) to solve hard mixed integer problems. The master problem is solved by DD-based optimization, the subproblem is solved by a linear programming (LP) solver and generated constraints are represented by a so called threshold diagram. The second idea (used in the first) is to use cost tuples in DDs to deal with convex piecewise linear objectives.

The following sections are organized as follows. Chapter 2 presents the required background, namely DD-based optimization and Benders decomposition. Chapter 3 introduces the proposed method. This chapter shows how DDs for optimization can be combined with Benders decomposition and how cost tuples can be used to deal with a convex piecewise linear objective. Chapter 4 introduces the tests and problems that are used to evaluate the model and also presents the computational results of these tests. The test results contribute to the subquestions asked above. Finally, Chapter 5 discusses these results, answers the research question and presents some recommendations for future work.

Chapter 2

Background

This chapter provides the main background information that is required for the method proposed in Chapter 3. The main building blocks are decision diagrams for optimization and benders decomposition. Also a specific kind of decision diagram, namely the threshold diagram is introduced.

Firstly, the general idea of a decision diagram is discussed (called an exact diagram). Secondly, the branch and bound procedure with decision diagrams is explained. Thirdly, the threshold diagram is introduced, and lastly, an explanation of Benders decomposition is given.

2.1 Decision Diagrams for Optimization

Decision Diagrams, originally meant for graphical representation of boolean functions, can be a useful tool in optimization and constraint programming. In this thesis only binary decision diagrams (BDDs) are considered. However, the theory can easily be extended to multi-valued decision diagrams (MDDs).

This section explains (summarized from [5]) how to use DDs in optimization. First a formal description of a decision diagram is given. Then the dynamic programming formulation is introduced. This formulation is required for compiling the decision diagram. After that the compilation of a decision diagram based on such a formulation is explained.

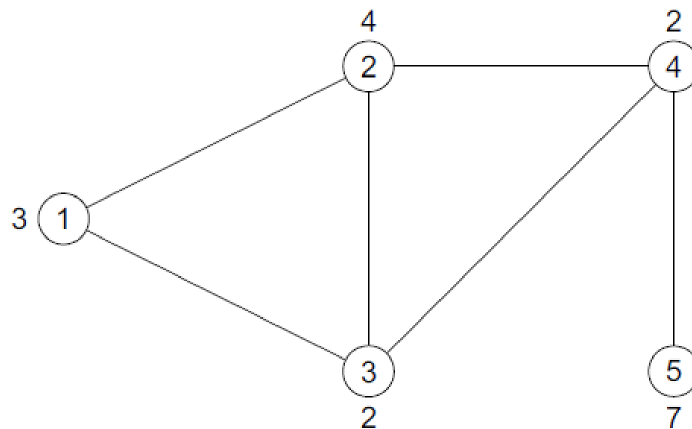

FIGURE 2.1: Example graph (from [5])

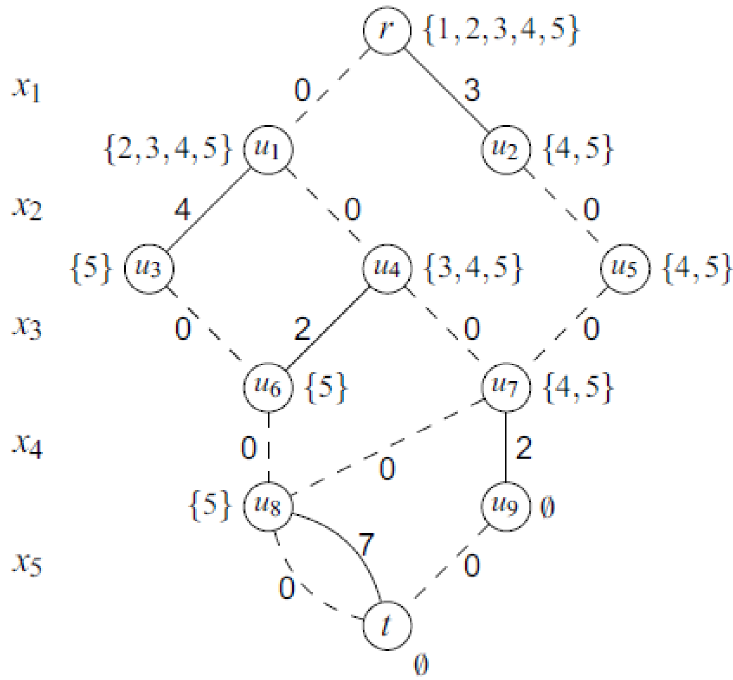

FIGURE 2.2: Example exact decision diagram for MISP (from [5]). A dashed line means a zero-label. A solid line means a one-label.

2.1.1 Formal description

For a discrete optimization problem $\mathscr{P}$ with objective function $f(x)$ and variables $x_1, ..., x_n$ one can define a decision diagram (DD). Formally a decision diagram is a tuple $B = (U, A, d, v)$ describing a directed acyclic graph. U is the node set, A the arc set, d the arc labels and v the arc weights. The node set U is subdivided in layers $L_1, ..., L_{n+1}$. L_1 and L_{n+1} consist of precisely one node, the root node r and the terminal node t respectively. The width $|L_i|$ of layer L_i is the number of nodes in the layer. The width of a DD is $\max_i\{|L_i|\}$.

Each arc $a \in A$ is directed from a node in some layer L_i to a node in layer L_{i+1}. Each arc also has a label $d(a) \in D(x_i)$, where i is the layer number, and $D(x)$ the set of all possible values for the variable x. Let a path be denoted by $p = (a^{(1)}, ..., a^{(n)})$ from r to t. This encodes an assignment to the variables $x_1, ..., x_n$: $x_i = d(a^{(i)})$ for each variable. This assignment is denoted by x^p. The length of a path is the sum of the lengths of the arcs so $v(p) = \sum_{i=1}^{n} v(a^{(i)})$. All r-t paths encode precisely the feasible solutions of $\mathscr{P}$ and for each path it holds that $f(x^p) = v(p)$.

When $\mathscr{P}$ is a minimization problem and $\mathrm{Sol}(B)$ gives the shortest path(s) of the DD B and $\mathrm{Sol}(\mathscr{P}) = \mathrm{Sol}(B)$ then B is called *exact* for $\mathscr{P}$ (i.e. all feasible solutions are represented by paths in the diagram and every path in the diagram resembles a feasible solution). So an exact DD reduces a discrete optimization problem to a shortest-path problem (on a directed acyclic graph). The shortest path of the DD is the optimal solution for the optimization problem.

Take the maximum independent set as an example problem. The maximum independent set problem (MISP) is the task to select vertices with maximum total value without selecting any two vertices that share an edge. Let the graph in Figure 2.1 be the input graph. Figure 2.2 is an exact DD that solves MISP for the given graph. The longest path in the diagram makes the assignment $x_2 = 1, x_5 = 1$. This indeed is the maximum independent set in the graph.

In a decision diagram every node has a state. In Figure 2.2 the state describing a node is the set of vertices in the input graph that can still be selected. The weights of the arcs in the graph are the weights of the vertices in the input graph when it is has one-label, and zero when it has a zero-label.

2.1.2 Dynamic Programming Formulation

As can be seen from Figure 2.2 the MISP is solved with help of a dynamic programming formulation. In a dynamic programming formulation (dp-formulation) a problem is described recursively. The formulation consists of a number of states s_j and variables x_j. The states are Markovian, that is, the state s_{j+1} only depends on the variable x_j and the previous state s_j. Each transition has a cost.

Formally, a dp-formulation consist of the four following elements:

1. A state space S with root state $\hat{r}$ and a set of terminal states $\hat{t}_1, ...\hat{t}_k$. For notational ease, let $\hat{0}$ be the infeasible state representing all infeasible solutions.
2. Transition functions t_j representing how variables control the transition between states. An infeasible state always leads to an infeasible state.
3. A transition cost function h_j.
4. A root value v_r (to take care of constants in the objective function), which is added to the transition costs directed out of the root state.

As an example take again the MISP. Let V be the set of vertices in the input graph, and let $N(j)$ be the function that returns all the neighboring vertices of vertex j including j itself, and w_j the weight of vertex j. Then the dp-formulation of MISP has the following components:

1. The state space $S = 2^V$, i.e. the power set of the V. $\hat{r} = V$, and $\hat{t} = \emptyset$.
2. The transition functions: $t_j(s_j, 0) = s_j \setminus \{j\}$,
$$t_j(s_j, 1) = \begin{cases} s_j \setminus N(j), & \text{if } j \in s_j \\ \hat{0}, & \text{if } j \notin s_j \end{cases}$$
3. Cost functions: $h_j(s_j, 0) = 0$, $h_j(s_j, 1) = w_j$
4. A root value of 0.

2.1.3 Compilation of a Decision Diagram

When a dp-formulation for a problem is known, this formulation can be used to build a decision diagram. This is called top-down compilation of a decision diagram. Because the state of each node is Markovian only the state of this node and the value of the decision variable is needed to define the next node. The resulting diagram is an exact diagram representing

the original problem. Building this DD then can be done by algorithm 1. Algorithm 1 relies heavily on the dp-formulation of the problem.

Most of the notation has already been introduced, except for $b_d(u)$. The function $b_d(u)$ gives the destination of the edge leaving node u with label d. The set D_j in this case is short for $D(x_j)$.

Algorithm 1 works as follows: One layer at the time is built. The first layer consists of the root node r with the root state $\hat{r}$. The next layer of the diagram is built by iterating over all nodes in the layer. For every node u in layer j a new node is created for every possible value that variable x_j can take (the set D_j; in the binary case $D_j = \{0, 1\}$) without resulting in the infeasible state. The state of the new node is computed with help of the transition function t_j. The edge costs are set with help of the the transition cost function h_j.

In the last layer, all nodes are merged into one node, the terminal node.

Algorithm 1: Exact DD top-down compilation

Input: A dp-formulation with root state $\hat{r}$, transition functions t_j and transition cost functions h_j.
Output: An exact DD for this dp-formulation.

```
1  Create node r = r̂ and let L_1 = {r};
2  for j = 1 to n do
3      let L_{j+1} = ∅;
4      forall the u ∈ L_j and d ∈ D_j do
5          if t_j(u, d) ≠ 0̂ then
6              let u' = t_j(u, d), add u' to L_{(j+1)}, and set b_d(u) = u',
               v(u, u') = h_j(u, u').;
7          end
8      end
9  end
10 Merge nodes in L_{(n+1)} into terminal node t;
```

Another method for building a decision diagram is compilation by separation. In Algorithm 2 the pseudocode of this procedure is given. Compilation by separation assumes an already existing (relaxed) decision diagram B and a constraint C that is violated or needs to be added. This constraint has transition function t^C and root state $\hat{r}^C$. The DD B is updated layer by layer. First all nodes are labelled as unvisited by setting the state to χ. Then all outgoing edges per layer are visited. For each edge there are four options: 1) The edge leads to node with an infeasible state according to t^C in which case the edge is removed from the diagram. 2) The edge leads to a node that is unvisited and feasible according to t^C. In this case the state of the node is updated according to t^C. 3) The edge leads to a node that is already visited, but the state of the node is not the same as according to t^C. (this can only occur in diagrams that are (partly) reduced). In this case that node is copied. This edge is set to lead to the new node. All outgoing nodes of the node are copied to the new node. 4) The edge leads to a node that is already visited (and the state is as expected according to t^C. In this case nothing happens. This last case is not in Algorithm 2.

Algorithm 2: Exact DD Compilation by Separation

```
   Input: A constraint C with transition functions t^C and root state r̂^C
          and a DD B
   Output: B updated so that it does not violate C.
 1 Let s(u) = χ for all nodes u ∈ B
 2 s(r) = r̂^C
 3 for j = 1 to n do
 4 |  for u ∈ L_j do
 5 |  |  for each arc a = (u, v) leaving node u do
 6 |  |  |  if t_j^C(s(u), d(a)) = 0̂ then
 7 |  |  |  |  Remove arc a from B
 8 |  |  |  else if s(v) = χ then
 9 |  |  |  |  s(v) = t_j^C(s(u), d(a))
10 |  |  |  else if (s(v) ≠ t_j^C(s(u), d(a)) then
11 |  |  |  |  Remove arc (u, v) from B
12 |  |  |  |  Create new node v' with s(v') = t_j^C(s(u), d(a))
13 |  |  |  |  Add arc (u, v')
14 |  |  |  |  Copy outgoing arcs from v as outgoing from v'
   |  |  |  |  L_(j+1) = L_(j+1) ∪ {v'}
15 |  |  end
16 |  end
17 end
```

2.2 Branch and bound with decision diagrams

An exact diagram contains all and only feasible solutions to the original problem. Because the width or size of an exact diagram may increase rapidly in proportion to the problem input size, it is possible that it does not fit in memory. A large diagram also takes longer to build. Because of constraints on memory usage or computation time, it is not feasible to build an exact representation of the problem. Branch and bound is a procedure that reduces the total search space into subproblems that are more easy to solve. The solutions of the subproblems provide lower and upper bounds to the original problem. These bounds can be used for branching decisions in the search tree and to exclude parts of the search tree. With decision diagrams, restricted and relaxed diagrams can be used to find these bounds.

In the subsequent sections, first the compilation of restricted and relaxed diagrams (for upper and lower bounds) is explained. After that the branch and bound procedure itself is introduced. Again, this section is summarized from [5].

2.2.1 Restricted Decision Diagrams

A restricted diagram is a decision diagram that does not contain all feasible solutions. However, all the paths in a restricted diagram still are feasible solutions. This diagram is called restricted, because the diagram is more restricting than the original problem. Apart from this it should also hold that every path length (in the case of minimization) is at least equal to the

value of that feasible solution. So in short, for a restricted diagram B' the following two points hold:

1. $\text{Sol}(B') \subseteq \text{Sol}(\mathscr{P})$
2. $v(p) \geq f(x^p)$ for each path p in B'

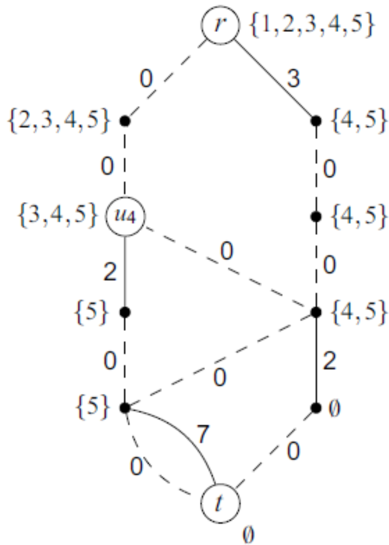

FIGURE 2.3: Example of a restricted decision diagram for MISP (adapted from [5]).

If these two properties hold, the restricted diagram yields an upper bound to the original problem (when minimizing).

In practice one would want the width of the diagram not to exceed some maximum width w_{max}. During the top down compilation of a decision diagram according to Algorithm 1 this can be easily achieved. When the width of a layer exceeds w_{max}, one can select nodes in the layer to be removed. Doing so, all the paths that go through this node are also removed, but no new path is added. Requirement 1 is achieved. Requirement 2 is also achieved trivially because none of the edge weights is changed.

In Figure 2.3 an example of such a restricted diagram is shown. In this figure the MISP from Figure 2.2 is reduced to a diagram with $w_{max} = 2$. In this case only one node (with its edges) is removed. According to the restricted diagram, the independent set with highest value is node 1 and node 5, with a total value of 10. This is a lower bound to the original problem, because MISP is a maximization problem.

The selection of nodes for removal is done by a heuristic. There are several heuristics that can be used for this task. In [5] the performance of several heuristics is shown. MAXLP is shown to be best for general use, and therefore in this thesis the MAXLP heuristic is used. With MAXLP all nodes in a layer are sorted based on the length of the shortest path from the root to that node. Those nodes are selected to be removed that have the maximum shortest path. The intuition of MAXLP is that the partial paths with longest length are less probable to be part of the optimal solution.

2.2.2 Relaxed Decision Diagrams

Relaxed decision diagrams are similar to restricted diagrams. With a minimization problem a restricted diagram provides an upper bound to the solution, and a relaxed diagram provides a lower bound to the solution. A restricted diagram contains only feasible paths, but not all paths, a relaxed diagram contains all feasible paths but not all paths are feasible. Apart from this it should also hold that every path length (in the case of minimization) is at most equal to the value of that assignment. So in short, for a relaxed diagram $\bar{B}$ the following two points hold:

1. $\text{Sol}(\bar{B}) \supseteq \text{Sol}(\mathscr{P})$
2. $v(p) \leq f(x^p)$ for each path p in $\bar{B}$

If these two properties hold, the relaxed diagram yields a lowerbound to the original problem (when minimizing).

Again, the goal is to limit the width of the diagram to w_{max}. With a restricted diagram this was achieved by removing nodes from the diagram. With relaxed diagrams this is not the case because no feasible path may be removed from the diagram. To reduce the width of the diagram nodes in the same layer can be merged during the top down compilation. When the width of a layer exceeds w_{max}, one can select nodes in the layer to be merged. For every dp-formulation a merge function must be defined such that the subtree of the merged node contains the subtree of all merged nodes. For example the merge function for the MISP is the set union. (The transition function only leads to an infeasible state if $j \notin s_j$. The set union does not remove any node from any of the merged states, and therefore no feasible solution is lost). In Figure 2.4 an example of such a relaxed diagram is shown. In this figure the MISP from Figure 2.2 is relaxed to a diagram with $w_{max} = 2$. In this case, only two nodes are merged into one (u'). The longest path in this restricted diagram means selecting node 2, node 3 and node 5, with a total value of 13. This is an upper bound to the original problem, because MISP is a maximization problem. This solution however is not feasible in the original problem because node 2 and node 3 are adjacent to each other. Again, following [5], the choice is made to select the nodes that are merged based on the MAXLP heuristic.

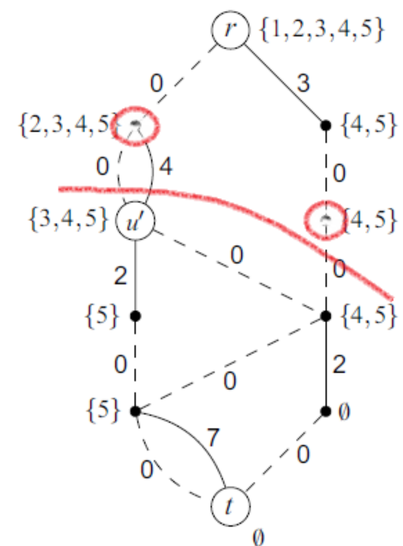

FIGURE 2.4: Example of a relaxed decision diagram for MISP (adapted from [5]) with the frontier cutset highlighted.

2.2.3 Branch and Bound procedure

Branch and bound is a technique that finds the global optimum for a problem by branching and bounding a full enumeration of all feasible solutions. The search space of all possible configurations is displayed as a tree. The root node contains all possible configurations. Every branch in the tree fixes a part of the configuration. Bounds on subtrees in the search space are used to determine where to search (branch). If (with minimization) the lower bound of a subtree is greater than the current incumbent (the upper bound), then that subtree does not contain an improvement on the incumbent and can be skipped. If the lower bound is lower than the incumbent, then that subtree is a good candidate for further investigation. If the subtree still contains too many possible configurations, the subtree can be split by another branch operation. This procedure continues until there is no subtree with a better lower bound than the incumbent. In that case, a global optimum has been found.

The algorithm for branch and bound with decision diagrams is based on the compilation of restricted and relaxed diagrams. Restricted diagrams are used to find feasible solutions to improve the incumbent and the relaxed diagrams are used to find lower bounds (with minimization) to subspaces of the solution space. If the lower bound exceeds the incumbent, then it is not required to explore that part of the solution space any further.

If the lower bound is lower than the incumbent the subspace of the diagram needs to be further explored. This is done by finding an exact cutset

in the relaxed diagram and adding each of the nodes in this exact cutset to a list of nodes that need to be explored further. An exact cutset provides an exhaustive enumeration of sub-problems. Each of the nodes in the subset can be used as the root of a decision diagram. A decision diagram starting from such a root is a subproblem of the original problem.

A cutset to the diagram is a set of nodes that cuts the diagram. This means that any path from root to terminal should go through at least one of the nodes in the cutset. An exact cutset is such a cutset with only exact nodes. A node is exact iff all paths from the root to the node are exact and the node is not a merged node. In any other case the node is called relaxed. Two heuristics for finding such a cutset are explained here.

The first of these two heuristics is called LASTEXACTLAYER. With this heuristic the last layer in the relaxed diagram that does not contain a relaxed node is selected as the exact cutset. This layer can be easily found during compilation because it is the layer preceding the first merged node.

A second heuristic is called FRONTIERCUTSET. For the algorithm, any exact cutset would do, but in this thesis the frontier cutset is used (just as in [5]). A formal definition for the frontier cutset is given in equation 2.1.

$$\text{FC}(\bar{B}) = \{u \in \bar{B} | u \text{ is } \textit{exact} \text{ and } b_0(u) \text{ or } b_1(u) \text{ is } \textit{relaxed}\}. \tag{2.1}$$

An example of such a frontier cutset can be seen in Figure 2.4. In this figure the nodes in the cutset are encircled. The red line divides the diagram in two parts. All nodes in the upper part are exact. All nodes in the lower part are relaxed.

Algorithm 3: Branch and Bound with Decision Diagrams

```
   Input: A dp-formulation with root state r̂.
   Output: The optimal solution z_opt.
 1 Initialize Q = {r} where r = r̂;
 2 Let z_opt = ∞, v*(r) = 0 ;
 3 while Q ≠ ∅ do
 4 |  u = select_node(Q), Q = Q \ {u} ;
 5 |  Create restricted DD B'_ut using Algorithm 1 with root u and
   |  v_r = v*(u) ;
 6 |  if v*(B'_ut) < z_opt then
 7 |  |  z_opt = v*(B'_ut) ;
 8 |  end
 9 |  if B'_ut is not exact then
10 |  |  Create relaxed DD B̄_ut using Algorithm 1 with root u and
   |  |  v_r = v*(u) ;
11 |  |  if v*(B̄_ut) < z_opt then
12 |  |  |  Let S be an exact cutset of B̄_ut ;
13 |  |  |  foreach u' ∈ S do
14 |  |  |  |  Let v*(u') = v*(u) + v*(B̄_uu'), add u' to Q ;
15 |  |  |  end
16 |  |  end
17 |  end
18 end
19 return z_opt ;
```

The algorithm for branch and bound with decision diagrams is shown in Algorithm 3. In this algorithm, $v^*(u)$ is the best (shortest) distance from the root to u. The value v_r in a DD is the start distance of a diagram.

2.3 Threshold diagram

A special kind of decision diagram is the threshold diagram. A threshold diagram is a diagram for a linear constraint (with only integer decision variables). More formally, a BDD representing the set $T = \{x \in \{0,1\}^n : a^T x \leq b\}$ is the threshold diagram of the constraint $a^T x \leq b$. Although it is common to restrict the coefficients of the constraints to integer constraints, in this thesis that is not the case. The coefficients a have real values.

The size of such a diagram is bounded by $O(n(|a_1| + |a_2| + ... + |a_n|))$ [17]. This means that when the weights in a are polynomially bounded in n, then the size of the diagram is polynomially bounded in n. However, when top-down compilation (as in Algorithm 1) is used, this result is not achieved.

A threshold diagram is built as follows (see also [2]). W.l.o.g suppose that $\forall i \; a_i \geq 0$. If not, one can substitute x_i with $1 - \bar{x}_i$ in this constraint. ($\bar{x}$ here means the negation of x). No extra variables need to be added for the negative coefficients.

The state of a node is the slack s of the constraint ($s = b - a^T x$ with all unset decision variables in x set to zero). When $s < 0$ the constraint is violated.

Take for example the constraint $2x_0 - 3x_1 + 4x_2 \leq 3$. This constraint can be rewritten to $2x_0 + 3\bar{x}_1 + 4x_2 \leq 6$. The root node is initialized with $s = b = 6$. While building the DD, the slack s is updated based on the decisions made. So with this constraint s is decreased with 2 if $x_0 = 1$, and with 3 if $\bar{x}_1 = 1$. When deciding on the value of x_2, s is compared with 4. If $s < 4$, then that decision is invalid and not included in the diagram. With previous decisions setting $x_0 = 0$ and $x_1 = 0$, the remaining slack s is $6 - 0 - 3 = 3$. Deciding $x_2 = 1$ therefore is invalid. Setting $x_2 = 0$ of course is valid.

In short (assuming all coefficients are positive) the dp-formulation is:

1. The state space $S = \mathbb{R}^+$. $\hat{r} = b$
2. The transition functions: $t_j(s_j, 0) = s_j$

$$t_j(s_j, 1) = \begin{cases} \hat{0}, & \text{if } s_j < 0 \\ s_j - a_j, & \text{else} \end{cases}$$

If the constraint has an equality sign and not an inequality sign, then one change must be made. The check $s_j < 0$ should be replaced by Equation 2.2.

$$s_j < 0 \lor s_j > \sum_{i=j}^{n} a_i \tag{2.2}$$

When nodes are merged, the state of the new node is the maximum of all states that are merged. If the maximum slack of all selected nodes is taken, then no feasible path is removed from the diagram.

With this dp-formulation, any linear constraint can be described and multiple constraints can be trivially combined by extending the state description with one variable for each constraint. So this dp-formulation can be used for any integer linear program, without requiring any specific problem domain knowledge, or a specific recursive formulation.

2.4 Benders Decomposition

In the method that is proposed in Chapter 3, the DD-based optimization presented above is combined with the Benders decomposition. Benders decomposition is explained in this section. First a general description of the Benders decomposition is given. Then a typical mixed integer programming (MIP) problem is introduced and rewritten into a master and subproblem. Lastly, the pseudocode of Benders decomposition is given that solves the introduced MIP problem.

Benders decomposition [3] is a common used decomposition technique used for solving large (mixed integer) (linear) problems. The decomposition splits the problem into a master and a subproblem. The master problem is a relaxation of the original problem. Solutions to the master problem are used to build a subproblem. The solution to the subproblem then is used to add a constraint to the master problem. This is repeated until the solution to the master problem is feasible to the original problem. The constraints that are returned by the subproblem typically exclude partial assignments, i.e. some combination of variable assignments is forbidden.

A formal explanation of Benders decomposition is as follows (from [12]):

Suppose the following MIP problem:

$$\begin{array}{ll} \text{minimize} & c^T x + f^T y \\ \text{subject to} & Ax + By \geq b \\ & x \in X \\ & y \geq 0 \end{array} \tag{2.3}$$

In Equation 2.3 the set X is the set of all feasible (integer) assignments to x. Therefore x denotes the integer decision variables and y the continuous variables.

With $x = \bar{x}$ fixed to a feasible integer configuration, this results in:

$$\begin{array}{ll} \text{minimize} & c^T \bar{x} + f^T y \\ \text{subject to} & By \geq b - A\bar{x} \\ & y \geq 0 \end{array} \tag{2.4}$$

The complete problem then is:

$$\min_{x \in X} \left[c^T x + \min_{y \geq 0} \{ f^T y | By \geq b - Ax \} \right] \tag{2.5}$$

The inner minimization problem in Equation 2.5 is an LP. Because it is an LP, its dual can be formulated as follows:

$$\begin{array}{ll} \text{maximize} & Z_D = (b - A\bar{x})^T u \\ \text{subject to} & B^T u \leq f \\ & u \geq 0 \end{array} \tag{2.6}$$

Now the problem in Equation 2.6 can be used as the (dual) subproblem in the Benders Decomposition. Its solutions are upper bounds to the original problem. In the master problem the constraints of the subproblem are removed and the objective is replaced by $\min z$. The goal is to optimize the variable z. When the dual subproblem is feasible it returns constraints on the value z. The Benders Decomposition algorithm is then as presented in Algorithm 4.

Algorithm 4: Benders Decomposition

Input: a MIP problem as presented in Equation 2.3
Output: The optimal solution z_{opt}.

1 $\bar{x}$ = Initial feasible integer solution (to the master problem). ;
2 $\text{LB} = -\infty$;
3 $\text{UB} = \infty$;
4 $\text{cuts} = \emptyset$;
5 **while** $\text{UB} - \text{LB} > \epsilon$ **do**
6 $\quad (\bar{u}, z_D) = \max_u\{c^T\bar{x} + (b - A\bar{x})^T u | B^T u \leq f, u \geq 0\}$;
7 $\quad$ **if** *Unbounded* **then**
8 $\quad\quad \bar{u}$ = an unbounded ray ;
9 $\quad\quad$ Add cut $(b - Ax)^T\bar{u} \leq 0$ to cuts ;
10 $\quad$ **else**
11 $\quad\quad$ Add cut $z \geq c^T x + (b - Ax)^T\bar{u}$ to cuts. ;
12 $\quad\quad \text{UB} = \min\{\text{UB}, z_D\}$;
13 $\quad$ **end**
14 $\quad (\bar{x}, z_I) = \min_x\{z | \text{cuts}, x \in X\}$;
15 $\quad \text{LB} = z_I$;
16 **end**
17 return z_I ;

What you can see in Algorithm 4 is that iteratively a feasible integer solution to the relaxed master problem is used to build a dual subproblem. The subproblem returns a feasible solution $\bar{u}$ with value z_D. This subproblem maximizes the objective and the master problem minimizes the objective. This respectively results in the upper and the lower bounds. When these bounds are equal an optimal feasible solution to the original problem is found. Every subproblem that is solved is either unbounded or bounded.

If the dual subproblem is unbounded (and therefore primal infeasible), an unbounded ray is used to add a new constraint to the set cuts. An unbounded model has a solution. However, a multiple of a ray (called an unbounded ray) can be added any number of times to the solution to give another feasible solution with a better objective.

If the dual subproblem is bounded, then the optimal solution is used to add a new constraint on the objective z to the set cuts. With this new cut added, the master problem can be solved again.

One thing to note is that the set of the constraints of the subproblem remains the same throughout the algorithm. Only the objective of the subproblem changes every iteration. Also, during the execution the upper bound represents a feasible solution to the original problem.

In [9] an extra constraint is proposed for the subproblem when the problem's objective relies only on the continuous variables (that is $c = 0$ and

$f \neq 0$). The constraint that is added to the primal subproblem is $f^T y \leq \mathrm{UB}$ (with UB the current incumbent). With this added constraint both the feasibility and the optimality is checked in the subproblem.

Chapter 3

Decision Diagrams with Benders Decomposition

MIP solvers are good at dealing with continuous variables. Integer variables however, often make a problem much harder to solve. It is known that decision diagrams perform especially well with discrete optimization (but are also limited to discrete optimization). Therefore the question arises whether the use of DDs can be extended to also solve hard MIP problems. For most (practical) problems both kind of variables are connected through constraints and therefore the discrete and continuous part of the problem are strongly connected.

One possibility is to split the set of variables (and the problem) into a part that has only discrete variables, and a part that has only continuous variables. A decision diagram can then be used to solve the discrete part. The continuous problem could for instance be solved as an LP. The question that still remains is how to deal with the connection between the two problems.

This section proposes a method that uses Benders Decomposition (BD). Benders Decomposition can be used to split the set of variables and constraints into a master problem and a subproblem. The master problem can contain all the integer variables and constraints. The subproblem is an LP. It is now possible to iteratively 'translate' constraints with continuous and mixed variables into constraints with only integer variables. The previous chapter provides the building blocks for this method.

The proposed method consists primarily of three ideas. The first idea is to combine the iterative procedure of Benders Decomposition with the iterative procedure of branch and bound with decision diagrams. This idea is explained in the first section of this chapter. The second idea, which is strongly linked to the first, is to use threshold diagrams and compilation by separation to add the cuts to the diagram. This is also discussed in the first section of this chapter. The third idea is to use cost tuples in decision diagrams to be able to deal with a piecewise linear objective, because the constraints generated by the subproblem may result in a piecewise objective. This idea is discussed in the second section of this chapter.

Some implementation choices and details have been left out of this section, such as (partial) reduction of diagrams, variable ordering and others. These implementation details can be found in Appendix A.

3.1 Decision Diagrams and Benders Decomposition

Branch and bound with decision diagrams and Benders decomposition have been introduced in Sections 2.2 and 2.4 respectively. In this section a method is proposed to combine these two procedures into one. First pseudocode is provided that gives a general overview of how the combined method works. After that, some important parts of the algorithm are explained in more detail.

The proposed solution is as follows. The problem is decomposed in two sets of variables and constraints by using Benders decomposition. In the decomposition the master problem deals only with integer variables (and all the constraints on only the integer variables) and is represented by a decision diagram. The subproblem deals only with continuous variables (and all constraints with any continuous variable) and is represented by an LP. Just as in the classical Benders Decomposition, solutions to the master problem are used to configure and build subproblems. The solutions of the subproblems are used to derive cuts and these cuts are used to update the master problem until the solution value of the master and subproblem coincide.

Decision diagram based branch and bound can be summarized as follows: Iteratively one node u is popped from a queue Q of nodes. The queue Q is initialized with the root node. A restricted diagram is built starting in u to provide a feasible integer solution. The incumbent is replaced if it is improved. If the restricted diagram is not exact (that is, it does not contain all feasible paths from u to the terminal), a relaxed diagram is built starting in u to provide a lower bound. If the lower bound indicates that the incumbent might be improved, then an exact cutset is sought and added to the queue Q. This procedure is repeated until the queue Q is empty. A longer explanation can be found in Section 2.2.

The method that is presented here is a combination of Benders decomposition and decision diagram based branch and bound (and is called Benders Decomposition for Decision Diagrams, or BDDD abbreviated). Both Benders Decomposition and Branch and Bound are iterative procedures, and in this case the two can be combined into one. BDDD is based on DD-based branch and bound (see the short summary on the side). The compilation of the restricted diagram however is now extended with subsequent subproblem generation and calls. This is required as the decision diagram only enforces a subset of the constraints, namely those of the master problem. A (number of) subproblem(s) is solved and the resulting cuts are used to update the restricted diagram until the restricted diagram and the subproblem yield the same result. Each subproblem is configured with a different fixed assignment of the discrete variables based on the solution of the restricted diagram. The generated cuts are stored in a set called $\mathscr{C}$ and are used in every subsequent compilation of a diagram (restricted and relaxed).

The pseudocode of this algorithm is presented in Algorithm 5. Algorithm 5 is clearly a combination of Algorithm 3 (Branch and bound optimization with DDs) and Algorithm 4 (Benders Decomposition). Its notation is introduced in Chapter 2. One can also clearly see that the BD is inserted in the branch and bound procedure where the restricted diagram was made in Algorithm 3.

More elaborately, BDDD works as follows: In the initialization the set of derived cuts $\mathscr{C}$ is empty. The objective of the decision diagram is set to minimize the variable z. This is done because the objective also relies on the continuous variables. Cuts on the objective variable, that are generated

Algorithm 5: Branch and bound with DDs and Benders Decomposition.

Input: A MIP $\min_{x,y}\{c^T x + f^T y \mid Ax + By \leq b, x \in X, y \geq 0\}$ and a dp-formulation for $x \in X$ with root state $\hat{r}$
Output: The optimal solution z_{opt}.
1 Initialize $Q = \{r\}$ where $r = \hat{r}$
2 Initialize $\mathscr{C} = \emptyset$
3 Let $z_{opt} = \infty$
4 **while** $Q \neq \emptyset$ **do**
5 $u = \text{select_node}(Q)$, $Q = Q \setminus \{u\}$
6 Create restricted DD B'_{ut} with root u s.t. $x \in X$ and $\mathscr{C}$.
7 Initialize $\text{UB} = \infty$
8 **while** $UB > v^*(B'_{ut})$ **do**
9 Let $\bar{x}$ be the optimal solution in B'_{ut}.
10 Let $(z_D, B'_{ut}, \mathscr{C}) = \text{SUBPROBLEM}(B'_{ut}, \bar{x}, \mathscr{C})$.
11 $\text{UB} = \min\{\text{UB}, z_D\}$
12 **end**
13 **if** $v^*(B'_{ut}) < z_{opt}$ **then**
14 $z_{opt} = v^*(B'_{ut})$
15 **if** B'_{ut} *is* not exact **then**
16 Create relaxed DD $\bar{B}_{ut}$ with root u s.t. $x \in X$ and $\mathscr{C}$.
17 **if** $v^*(\bar{B}_{ut}) < z_{opt}$ **then**
18 Let S be the last exact layer according to B'_{ut} and $\bar{B}_{ut}$
19 Let $(S, B'_{ut}, \mathscr{C})$ =MAKEEXACTCUTSET$(S, B'_{ut}, \mathscr{C})$
20 **foreach** $u' \in S$ **do** add u' to Q
21 **end**
22 return z_{opt}

by the results of the subproblems, may constrain the objective variable z. Because $\mathscr{C}$ is initially empty, the variable z is initially unconstrained. In this way the original objective is in time represented by the objective cuts solely in the integer variables. The result is a piecewise convex linear objective.

So the master problem is solved by means of a decision diagram optimizing $\min z$ subject to $\mathscr{C}$ and $x \in X$. The constraints $x \in X$ can be represented by a DD using a custom design (i.e. custom state description, transition function and merge function). The master problem is first solved with a restricted diagram. The solution to the restricted diagram is a feasible solution to the master problem, and in the Benders Decomposition this solution is a lower bound (to the restricted problem, not to the original problem). The subproblem is solved with this solution as a fixed configuration for the x variables. A feasible subproblem gives an objective value z_D which is an upperbound in the decomposition, but is also a lowerbound to the original problem because it is both feasible to the master problem and the subproblem.

The execution of a subproblem can be seen in Algorithm 6. As explained in Section 2.4, the subproblem is either unbounded or bounded. If it is unbounded, the unbounded ray is used to formulate a new constraint on the discrete x variables. If it is bounded, the optimal solution in the subproblem is then used to build a new constraint on the discrete x variables and

Algorithm 6: SUBPROBLEM

Input: A restricted diagram B' and a fixed assignment to $x = \bar{x}$.
Output: The objective value z_D, and an updated diagram B' and $\mathscr{C}$

1 Let $(\bar{u}, z_D) = \max_u \{c^T \bar{x} + (b - A\bar{x})^T u \mid B^T u \leq f, u \leq 0\}$.
2 **if** z_D *is* unbounded **then**
3 | Let $\bar{u}$ be an unbounded ray in the subproblem
4 | Add $(b - Ax)^T \bar{u} \leq 0$ to $\mathscr{C}$
5 **else**
6 | Add $z \geq c^T x + (b - Ax)^T \bar{u}$ to $\mathscr{C}$
7 Update DD B' with the new cut.
8 return $(z_D, B', \mathscr{C})$;

the objective variable z (this is called an objective cut in the rest of this thesis). These constraints are added to the master problem in two ways: by adding the constraint to the set $\mathscr{C}$ and by updating the restricted diagram accordingly by separation (Algorithm 2). This procedure is repeated until the subproblem results in the same solution as the master problem. If the subproblem returns the same objective value as the master problem, then the solution to the master problem satisfies all the constraints of the subproblem.

After this the basic idea of the branch and bound procedure is continued. A relaxed DD is built (constrained by both $x \in X$ and by the set of cuts $\mathscr{C}$) if necessary and the nodes in an exact cutset are added to the list of nodes that are still to be evaluated (this is explained more elaborately in a subsection below). These nodes are handled as new root nodes as described before. A restricted DD is built and updated with all the required cuts, and also solved with the relaxed DD. This procedure is continued until there are no nodes left to evaluate.

Cuts are added by **Separation**. The full procedure is explained in Algorithm 2. In short, the new cut can be described by the transition functions of a threshold diagram. These transition functions are used to update the state of every node in the diagram top-down. If a node becomes infeasible it is removed from the diagram. If two edges lead to the same node in the diagram, but to two different nodes according to the transition function, then a node is separated. A new node is created having the same outgoing edges. The states of the two nodes differ according to the transition functions, and on edge lead to the one and the other to the other.

In the following two subsections, two points that are mentioned before are elaborated. The first point is how to use a threshold diagram to represent the constraint on only the x variables, and how to update the current restricted diagram. (The other kind of constraint is discussed separately in Section 3.2). The second point is about finding the exact cutset.

3.1.1 Cuts using the threshold diagram

The cuts returned by the subproblem can have two forms. The first form (that occurs when the dual subproblem is unbounded) results in a constraint of the form $a^T x \leq b$. If the continuous variables do not contribute to the objective, then all constraints are of the first form. All the cuts of the first form can be described by the constraints $Ax \leq b$ with x the integer (binary)

variables of the master problem. A decision diagram that describes the feasible space subjected to such a constraint can be built using the threshold diagram described in Section 2.3. But instead of building this threshold diagram explicitly, compilation by separation (Algorithm 2) can be used to update the restricted diagram. The transition functions t^C, required to compile by separation, are given by the threshold diagram formulation. The new cut is also added to the set of cuts $\mathscr{C}$ so that decision diagrams that are built later in the iterative procedure do not violate the constraint.

There is an exception that may occur which is not dealt with in the described compilation by separation (Algorithm 2) and in the BDDD algorithm (Algorithm 5) because Algorithm 2 is written for an exact decision diagram, and not for a restricted diagram. This exception occurs when the new cut removes all paths to the terminal from the restricted diagram. This would mean that the problem is not feasible, but this need not be the case. When all paths to the terminal have been removed by the added cut, one has to check if there is indeed no feasible path from the current root to the terminal. An easy method to do this, is to build a new restricted diagram. If this new diagram has no path from the root to the terminal, then the diagram is indeed infeasible. If however the new diagram is feasible, then the iterative Benders decomposition can be continued.

A special case of this exception is when the root node u of the restricted diagram itself is infeasible to the new cut. In that case the iteration can be stopped and the next node from the list Q can be considered. This exception may also occur at a different point in the algorithm. When a new root node u is selected from the list Q, it is possible that new cuts have been introduced in the mean time. The state of the node u needs to be updated according to these cuts. Again, it is possible that there does not exist a feasible path from node u to the terminal. In that case a new node from Q is selected.

Apart from this, one other alteration should be made to Algorithm 2. When a cut is added by separation, it may occur that a decision diagram exceeds its maximum width. A solution to this problem is to simply ignore the step in the algorithm that creates the new node. More precisely in terms of the pseudocode in Algorithm 2: When a new node v' is added to a layer this may have as a result that the maximum width w_{max} of the restricted diagram is exceeded. In that case the new node v' can be left out of the diagram.

3.1.2 Finding an exact cutset

In the presented branch and bound with DDs (Algorithm 3) it can easily be decided if a node is exact or not. This is required for finding the exact cutset. A merged node and all its descendants are always relaxed. Therefore, a node is exact iff it is not merged and all paths from the root to that node only contain exact nodes. An exact cutset is a set of exact nodes that cuts the diagram, that is, all r-t paths go through one of the nodes in the cutset.

A **cutset** of a diagram is a set of nodes that cuts the diagram. This means that any path from root to terminal should go through at least one of the nodes in the cutset. An **exact cutset** is such a cutset with only exact nodes. An exact cutset provides an exhaustive enumeration of sub-problems.

One change to this idea is made with the introduction of the decomposition. Some of the constraints of the problem are not in

the master problem and only in the subproblem. The master problem itself is relaxed. Because the master problem itself is relaxed, the 'exact' nodes in the relaxed diagram and in the restricted diagram may also be relaxed. In the relaxed diagram it is possible that a node u is labelled as 'exact' even though the length of the shortest path to u does not equal the cost of that assignment in the original problem. In the extreme case a relaxed diagram does not contain any relaxed node, although its shortest path does not correspond to a feasible solution. This is possible because not all constraints of the original problem are considered.

The question that is at stake is whether BDDD always returns the optimal solution. Because of the properties of Benders decomposition, the (updated) restricted diagram provides a solution that is feasible to both the master and the subproblem. The compiled relaxed diagram should now be used to check if further investigation of this part of the diagram is needed. If further investigation is needed, then an exact cutset is required to know from which points this further investigation should start. Therefore it is important to be sure that the exact cutset indeed is an exact cutset.

In the BDDD algorithm (Algorithm 5) you can see a small change in finding the exact cutset compared to the normal DD-based branch and bound (Algoritm 3). The nodes in the cutset should now be exact for both the restricted and the relaxed diagram. For the relaxed diagram the same rule can be used as presented before to determine exactness. For a node in the restricted diagram the following rule can be used: every partial path leading to the node should be part of an exact path in the restricted diagram. A path p in the restricted diagram is exact iff the length $v(p)$ is the same as the objective value of the subproblem with x fixed to x^p (Consider the objective value of an unbounded subproblem to be infinite).

In short the new definition for an exact node (according to both the relaxed and the restricted diagram):

Exact node A node u is exact iff it is not a merged node, and every partial r-u path is exact in the relaxed diagram and is part of an r-t exact path p in the restricted diagram.

To determine the exactness of a node in the restricted diagram it needs to be part of an exact path. If a node in the relaxed diagram is not part of the restricted diagram, its exactness cannot be determined. Therefore such a node is considered to be relaxed.

With the top-down compilation presented in Algorithm 1, every node has only one partial path leading to it. If this indeed is the case, as it is in the method presented here, then the rule to determine exactness in the restricted diagram is simplified: a node u is exact in the restricted diagram iff there is an exact path p through u (in the restricted diagram).

By using the method to determine exactness as presented above, an exact cutset can now be found using the LastExactLayer heuristic. Here it is assumed that top-down compilation (as in Algorihtm 1) is used for both the restricted and the relaxed decision diagram. It is also assumed that the compilation is performed with the same maximum width. A result of this assumption is that the last layer in the restricted diagram B'_{ut} from which no node is removed, is the same layer as the last layer in the relaxed diagram $\bar{B}_{ut}$ in which no nodes are merged. This layer is called the last exact

layer. Normally the LASTEXACTLAYER heuristic selects the last layer in the relaxed diagram. Now one should also look at the restricted diagram. The nodes in the cutset should also be exact in the restricted diagram.

For the cutset to be exact, it should meet three requirements. The first requirement is that all nodes in the cutset should be present in both the restricted and the relaxed diagram. This is the case because both diagrams are compiled by the same algorithm and no deletion or merging is done on the layer. The second requirement is that all the nodes are exact according to the relaxed diagram. As can be easily seen this is the case as no node has been merged in this layer or in any layer before, so all the nodes in this layer and in all the previous layers in the relaxed diagram. The third requirement is that all the nodes in the cutset are exact according to the restricted diagram. To ensure that this is the case extra runs of the subproblem are required to make all the nodes in the cutset exact. This is achieved by running Algorithm 7 on the cutset.

Algorithm 7 ensures that every node u in a cutset is exact in the restricted diagram. This is achieved by running a subproblem configured by a path p through u for every path from the root r to u. The execution of this subproblem is repeated until the subproblem and the decision diagram give the same objective value.

Algorithm 7: MAKEEXACTCUTSET

Input: A cutset S in a diagram B' and the set $\mathscr{C}$.
Output: The exact cutset S and an updated B' and $\mathscr{C}$.

```
1 for u ∈ S do
2     for every r-u path q do
3         Let p be any r-u-t path starting with path q.
4         repeat
5             Let (z_D, B', C) = SUBPROBLEM(B', x^p, C).
6         until v(p) = z_D
7     end
8 end
9 return (S, B', C)
```

One special note on the execution of MAKEEXACTCUTSET should be made. When another compilation technique is used than the top-down compilation, a problem may arise. When a new cut is added by SUBPROBLEM, the compilation by separation may cause the width of the last exact layer or any preceding layer to exceed the maximum width. There are two options to solve this problem. The first is to delete the newly created node, reset the last exact layer to the layer preceding the deleted node and restart MAKEEXACTCUTSET on this preceding layer. The other option is to keep the newly created node and let the layer exceed the maximum width. This counts only for the layers up and until the last exact layer. In the case presented in this thesis, reduction is not applied and this problem will not arise.

Using a relaxed cutset

The execution of MAKEEXACTCUTSET can be time consuming, especially when the subproblems are large. This section presents a method that is simpler than the one presented above, and that might provide faster runtimes in some cases.

The method described above is designed and analysed to work with any decision diagram, also with reduced diagram. However, in the case presented here DDs are compiled top-down by Algorithm 1 without any reduction. The DDs resulting from this compilation show very specific properties such as that every node (except the terminal and merged nodes) has only one incoming edge and only one partial path leading to it. Because of these properties the extensive method described above can be simplified. The intuition of this method is that the execution of the subproblems by MAKEEXACTCUTSET is postponed. In fact, the execution of MAKEEXACTCUTSET is skipped.

The restricted diagram is still needed to determine the exactness of a node to ensure that a node is not considered as exact based on simply the relaxed diagram. As a conclusion, if the LASTEXACTLAYER is used, even when the execution of MAKEEXACTCUTSET is skipped, the last exact layer is determined based on the restricted diagram. The last exact layer of the restricted diagram is used as the cutset that is added to the list Q.

3.2 Cuts on the optimal value

The constraints that are discussed here can be described by a linear constraint of the form $z \geq a^T x + b$, where z is the variable that is optimized. The presence of the variable z in the constraint makes this constraint type more complicated to handle. Because z is part of this constraint, it cannot be added as a constraint of the first form. The effect of multiple such constraints is that the objective becomes a piecewise linear (and convex) function. The question that needs to be answered is how a decision diagram can deal with a piecewise linear objective.

The idea presented in this section, namely cost tuples, can be used to accommodate for the constraints of the second form. This constraint form is discussed first, and after that the cost tuples are introduced and explained. Finally, a method is described by which the constraints can be used to bound the feasible space.

To allow for this kind of constraints (and for a piecewise linear objective) the objective $\min c^T x + f^T y$ can be replaced by $\min z$. The constraints (or objective cuts) do not reduce the DD, but alter the edge weights in the DD. The edge weights can be described by a tuple with one value for each of the objective cuts.

When $f^T y$ is not a part of the objective, objective cuts never occur. If the continuous variables do not contribute to the objective the subproblem is either infeasible (i.e. the dual is unbounded), or returns the same solution as the master problem. In that case the objective is not replaced by $\min z$ but remains $\min c^T x$.

In the terminal node one can now find the shortest path. The objective cuts provide a lower bound for the length of each path. The lower bound of the path is the maximum value assigned by all the objective cuts, for this

maximum value satisfies all objective cuts. The shortest path is the path which has minimal lower bound.

Cost tuples

The following explains how the decision diagram is compiled and solved with cost tuples. This explanation is for binary decision diagrams, but can easily be extended for multivalued DDs. However, this method works only for diagrams that are not reduced (and are compiled top down by Algorithm 1).

Firstly, the objective of the master problem is to minimize the objective variable z. The variable z is constrained by all the objective cuts in the set $\mathscr{C}$. Denote this subset of $\mathscr{C}$ as $\mathscr{Z}$. In the initialization $\mathscr{Z}$ is initialized as an empty set. When a feasible dual solution to a subproblem has been found an objective cut (constraint) is added to $\mathscr{Z}$. In the master problem a tuple z_n with size $|\mathscr{Z}|$ is part of the state of each node. Also each edge has a cost tuple z_e with size $|\mathscr{Z}|$. The elements of the tuples are determined by the cuts. For example, if the second cut in $\mathscr{Z}$ is $z \geq -4 + 2x_1 + 3x_4$, then the second element of the cost tuple for the edge with label 1 deciding on x_4 is 3. One can also describe this as follows: The set of objective cuts in $\mathscr{Z}$ can be represented as $z \geq G\hat{x}$. The vector $\hat{x}$ here is equal to $[1; x^T]^T$ (the first element is needed for the constant terms in the cuts), and the inequality is element-wise. Each constraint is a row in G. The columns of G determine the cost tuples of the edges. The first column of G (for the constant terms in the cuts) is the value for the initial tuple z_{root} in the root state. The consecutive columns are for the edges deciding the variables $x_1, x_2, ...$ with label 1. At each step the cost state of a node z_n is the cost state of the previous node z_m plus the cost label of the connecting edge z_e ($e = (m, n)$). The cost of the node is equal to the largest element in z_n because z has satisfy all the constraints in $\mathscr{Z}$.

If no reduction is applied, then only when nodes are merged does a node have multiple incoming edges. If there are multiple incoming edges

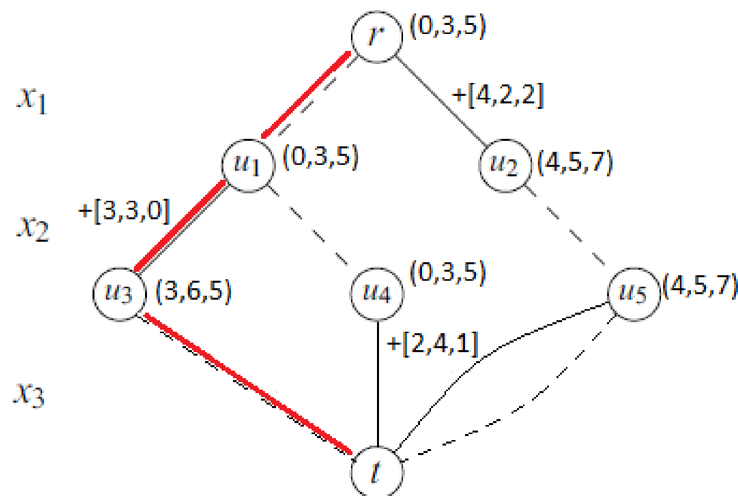

FIGURE 3.1: Example decision diagram with cost tuples and its shortest path.

to a merged node, z_n is the piecewise minimum of $z_m + z_e$ for each edge $e = (m, n)$. The result of this is that the lower bound of the cost of the paths through the merged node is the minimum of the lower bounds.

The terminal is also a merged node, but the value of this node should be exact. The procedure to find the shortest feasible path is as follows: For each node n that is merged into the terminal, find the maximum value in z_n. That value is the lowerbound of the path through node n. The smallest lower bound is the objective value. If the diagram is restricted (and therefore no merge operation is done), then the path through the node belonging to that lower bound is the optimal feasible assignment in that diagram.

Figure 3.1 is an example of how a decision diagram with cost tuples might look like. This graph represent the set of cuts in Equation 3.1.

$$\begin{aligned} z &\geq 0 + 4x_1 + 3x_2 + 2x_3 \\ z &\geq 3 + 2x_1 + 3x_2 + 4x_3 \\ z &\geq 5 + 2x_1 + 0x_2 + 1x_3 \end{aligned} \tag{3.1}$$

The terminal in Figure 3.1 has four incoming edges. They represent four nodes with cost tuple $(3, 6, 5)$, $(2, 7, 6)$, $(6, 9, 8)$ and $(4, 5, 7)$. The leftmost incoming edge (path $x = (0, 1, 0)$) leads to the state $(3, 6, 5)$ with value 6. This is the shortest path satisfying all objective cuts.

Bounding the feasible space

The problem with the objective cuts is that they provide lower bounds on the objective, but do not constrain the feasible space. The objective variable z is bounded from below, but no assignment to the x-variables is made infeasible.

When a feasible solution to the problem has been found, the incumbent z_{opt} can be used to bound the feasible space. If the incumbent is not the optimal solution then for finding the optimal solution the constraint $z < z_{opt}$ should hold (z is, as before the objective variable, and z_{opt} the current incumbent). This constraint bounds the feasible space.

Because the objective variable z is bounded from below by all the objective cuts, the following equation holds: $G\hat{x} < z_{opt}$ (the inequality is element wise). This bound can be added to the diagram as a constraint of the first form ($a^T x \leq b$).

Chapter 4

Evaluation

This section evaluates the performance of the proposed method. This evaluation is done according to the subquestions asked in the introduction. These four questions are 1) How do the threshold diagrams perform? 2) How does the idea of cost tuples perform for an integer problem with a convex piecewise linear objective? 3) What number and kind of constraints are generated in the decomposition? And 4) what elements contribute to the runtime of the proposed method?

The evaluation of these questions in this chapter is divided in three parts. Because the decomposition relies on threshold diagrams, the threshold diagram itself is tested first with typical integer optimization problems without any decomposition. This answers the first question. Secondly the cost tuple method is tested with typical integer optimization problems that are altered in such a way that they have a piecewise linear objective. This answers the second question. Thirdly the decomposition is tested with several mixed integer optimization problems to answer the third and fourth question.

In all cases where a comparison is made with MIP, the Gurobi solver [11] has been used. For a more fair comparison to simple branch and bound, the parameters for gurobi are set such that presolve and cuts are turned off, heuristics are disabled and gurobi runs on at most one thread. The tests are run on a 2.27 GHz Intel Core i5 M430 with 2 cores and 4 GB RAM (of which at most 2GB is used for the tests). In all the tests where the LP subprocedure is called, the LP subproblem is solved by the same Gurobi solver with also the same parameter settings (no presolving, at most one thread and heuristics disabled). For most of the tests presented below the GLPK solver [10] has also been used. Because the results with this solver mostly did not differ considerably, these results have been left out.

For the DD and BDDD solvers the following parameters have been used (except when else is mentioned). Because of the results shown in [5]: the maximum width of every diagram is set to the number of undecided variables; the merge heuristic is MAXLP; no variable reordering is done; no diagram reduction is performed; for BDD the FRONTIERCUTSET heuristic is used; and for BDDD the LASTEXACTLAYER heuristic is used.

4.1 Threshold diagram

To answer the first question, namely how the threshold diagram performs, the threshold diagram as introduced in section 2.3 is evaluated in this section. Two tests are performed. The first test is on the maximum independent set problem (MISP) that was introduced before. With this test the

threshold diagram formulation is tested against a custom dp-formulation. The second test is on the multidimensional knapsack problem (MDKP). This test evaluates mainly the influence of the number of constraints.

4.1.1 Threshold versus Custom

In section 2.1.2 a dp-formulation for MISP was introduced. This formulation is designed specifically for MISP and one may expect that it outperforms the threshold formulation introduced in section 2.3. The main reason to expect this, is that the custom MISP dp-formulation is much smaller in memory, and computationally simpler. The feasibility check is one bit check, and the state update is one substraction opeartion. The threshold diagram formulation however is much larger in memory, and has to do multiple checks (each constraint whose slack is changed, might be violated).

The MISP problem is known to be a problem that can be well handled by DDs. Therefore this problem is chosen as a test candidate. The test presented below is set up to see how the proposed threshold formulation performs for the MISP problem.

Take a graph $G(V, E)$ with $n = |V|$ the number of vertices and w_j the weight of vertex j. The MISP problem for this can be formulated as an integer program with decision variable x_j meaning vertex j is selected:

$$\text{maximize} \quad \sum_{j \in V} w_j x_j \tag{4.1}$$

$$\text{subject to} \quad x_i + x_j \leq 1, \qquad \forall (i, j) \in E \tag{4.2}$$

$$x_j \in \{0, 1\}, \qquad \forall j \in V \tag{4.3}$$

But one can also formulate this problem in another more compact way. This formulation makes use of the cliques that are in the graph. Instead of writing one constraint for each edge, one can write one constraint for each clique in the graph. Let $\mathscr{C}$ be a set of cliques that covers all edges in E. Then equation 4.2 can be replaced by equation 4.4.

$$\sum_{i \in C} x_i \leq 1, \quad \forall C \in \mathscr{C} \tag{4.4}$$

Test instances of this problem were created by generating random graphs with a density $p = 0.8$. This density is chosen because previous results have shown that DDs have good performance on MISP with high density. For each pair of vertices in V an edge is constructed with probability p. For each graph size ten instances are created and the runtime shown in the graph is the average runtime.

In Figure 4.1 the relative performance of MIP and the decision diagram with the custom dp-formulation and the threshold dp-formulation can be seen. Both MIP and the DD threshold formulation have been tested twice: once with the edge constraint formulation and once with the clique formulation. It can be seen from the graph that all five methods show exponential growth in the runtime.

The results show that the performance of the threshold diagram DD is slow compared to the custom state DD, but is still faster than MIP for most instances. The results also show that another constraint formulation

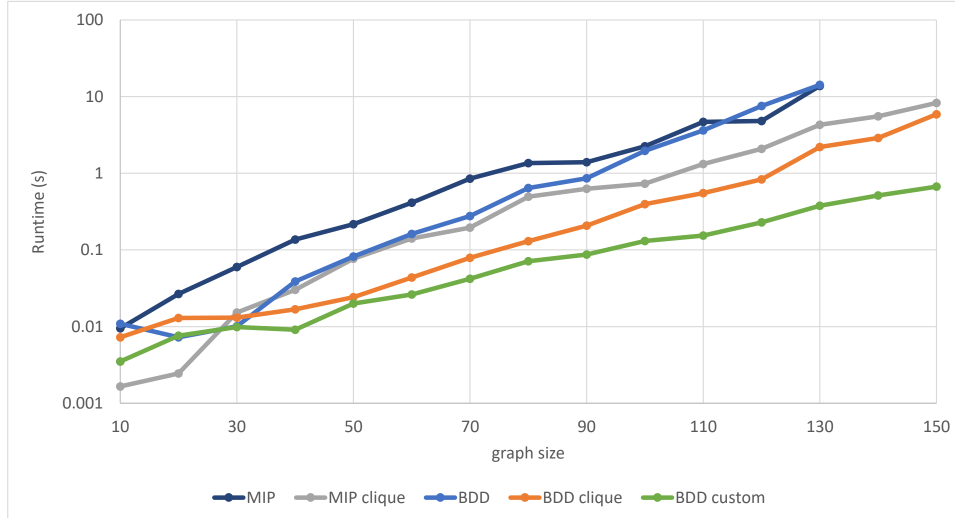

FIGURE 4.1: Runtime performance for MISP of DDs with custom dp-formulation and threshold formulation compared to MIP with edge constraints and clique constraints.

results in a lower runtime for both MIP and BDD solved by a threshold formulation.

One special note should be made here. For the larger MISP instances the BDD using the threshold formulation spent a large and increasing amount of time in garbage collection. For some of the larger instances this was even up to half of the total runtime. This is of course the result of an inefficient implementation. At the moment each compiled diagram is built in newly allocated memory and the old diagram is left for the garbage collector to clean. A more efficient implementation would be to build the new diagram in the same allocated memory as the old diagram. Because of lack of time, this implementation is not yet improved.

4.1.2 Number of constraints

The test presented in this section tests the influence of the number of constraints on the runtime of the threshold diagram. The problem selected for this evaluation is the multidimensional knapsack problem (MDKP). The number of dimensions in this problem is the number of constraints and therefore this problem is well suited to test the influence of the number of constraints.

For a problem with D dimensions, the MDKP can be formally described as follows:

$$\text{maximize} \quad \sum_{i \in I} v_i x_i \tag{4.5}$$

$$\text{subject to} \quad \sum_{i \in I} w_{di} x_i \leq W_d, \qquad d = 1 \ldots D \tag{4.6}$$

$$x_i \in \{0, 1\}, \qquad \forall i \in I \tag{4.7}$$

In Equations 4.5-4.7 W_d is the weight limit in dimension d, w_{di} is the weight of item i in dimension d, v_i is the value of item i, I is the set of all items and $n = |I|$ the number of items.

The test setup is as follows. Every weight w_{id} is chosen randomly from the interval $[0, 6)$, and the capacity W_d is chosen randomly from the interval $[n/3, 4n/3)$. If as a result the weight of all items does not exceed the capacity ($\sum_i w_{id} \leq W_d$), then a new capacity is randomly chosen from the same interval until the capacity is exceeded.

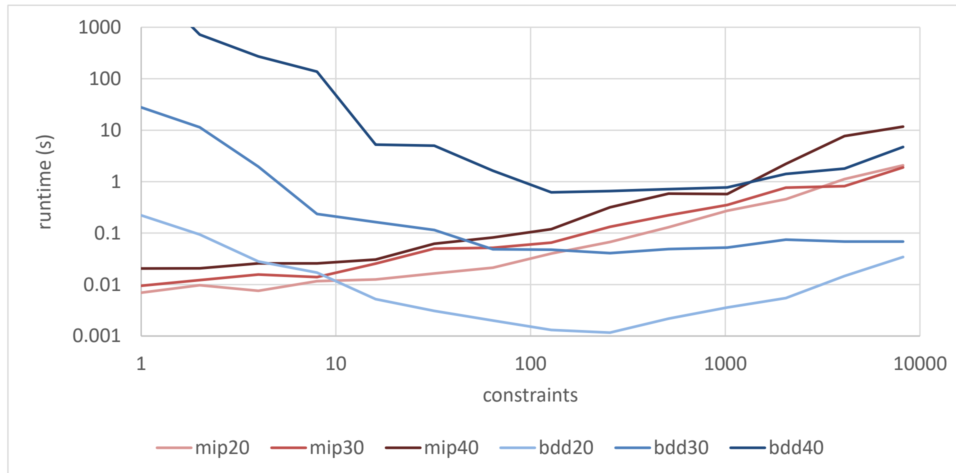

FIGURE 4.2: Runtime for MIP and BDD for MDKP for an increasing amount of dimensions/constraints.

In Figure 4.2 the results of a test are shown with 20, 30 and 40 items with an increasing number of dimensions/constraints. The results show that solving the knapsack problem with only few dimensions is inefficient with a BDD. The feasible space is large and the relaxation is not good. An increasing number of dimensions (and constraints) reduce the feasible space and number of branch operations required to solve the problem. This is not shown in the graph, but for the MDKP with 20 items and a large number of dimensions the number of required branch operations reduces even to only one.

When the number of constraints is increased even more, the runtime of BDD increases again, although the results show that the number of branch operations still decreases or stay constant. This increase in runtime is caused by the increased number of constraints. Each constraint is represented by one variable in the BDD state description. With as many as 1000 or more constraints it is easy to imagine that the transition function takes more and more time to be executed.

4.2 Cost tuples

In section 3.2 the idea of assigning cost tuples to the edges is introduced. This is done to be able to deal with constraints of the form $z \geq ax + b$ with z the variable in the objective (objective cuts). In this section this method is evaluated separated from the decomposition. By this evaluation the second subquestion is answered. This question is 'How does the idea of cost tuples perform for an integer problem with a convex piecewise linear objective?'.

First a test is introduced to test the contribution of the objective cuts to the runtime. Then the test results are shown.

For testing the contribution of the objective cuts to the runtime, the MISP introduced before is extended in such a way that the new problem has multiple constraints on the objective value z. The objective in equation 4.1 is replaced by $\max z$. New constraints are added as formulated in equation 4.8 with m the number of objective cuts. The test setup is the same as in Section 4.1.1. The BDD solver uses the custom state description introduced in Section 2.1.2. The diagrams are not reduced in this test. The density is set to $p = 0.8$.

$$z \leq a_i x + b_i \quad \forall i \in \{1, ..., m\} \tag{4.8}$$

The test described above is executed with graphs of size $n = 75$, 100, and 125. The test results for MIP and BDD can be seen in Figure 4.3.

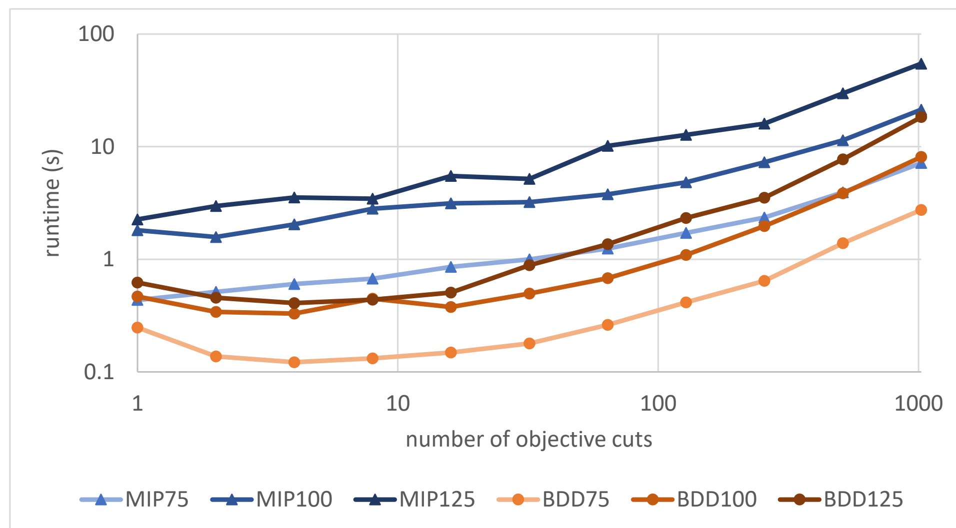

FIGURE 4.3: Runtime of MIP and BDD (with custom state) for MISP with increasing number of objective cuts (Equation 4.8) for $n = 75$, 100, and 125.

What can be seen in Figure 4.3 is that an increase in the number of objective cuts also means an increase in runtime for both MIP and BDD. The increase in runtime for bdd was to be expected. When a new node is made, the state and cost tuple of the previous node is copied and altered. The size of the cost tuple is the number of objective cuts. With an increasing number of objective cuts and everything else constant, it is to be expected that the number of objective cuts at some point becomes the dominant factor. What is interesting is that the increase in runtime of MIP is almost as much as the one of bdd.

4.3 Decomposition

The Benders decomposition combined with the branch and bound introduced in Section 3.1 is tested in this section. With the evaluation in this section two questions are answered: 1) What number and kind of constraints are generated in the decomposition? And 2) what elements contribute to

the runtime of the proposed method? Two problems are chosen for this evaluation. The first problem is a scheduling problem with the independent set problem as its basis. The second problem is a problem related to the multi dimension knapsack problem. In the final section of this chapter some other problems are mentioned that have also been tested.

To answer the two mentioned questions, several subquestions can be asked:

1. What is the effect of constraint cuts on the runtime?
2. What number of generated cuts are generated?
3. At what moment do these cuts occur?
4. Which elements contribute for what amount to the runtime of the algorithm?
5. What number of constraints are of the first and second form?
6. What is the effect of the cost tuples in combination with the decomposition?

(The fifth and sixth items are only tested with the second problem.)

4.3.1 Scheduling with independent sets

The problem in this section is loosely connected to the resource constrained project scheduling problem (RCPSP). RCPSP is a scheduling problem under the constraints of a limited number of resources, precedence constraints and operation durations. More formally, a RCPSP is a optimization problem defined by a tuple (A, R, P, d, u, c). A is the set of activities $(a_1, ..., a_n)$, R is the set of resources $(r_1, ..., r_m)$. P is the set of precedence constraints of the form $a_i \prec a_j$. This constraint means that a_j may start only if a_i is already completed. $d : A \rightarrow \mathbb{R}^+$ is a function that returns the duration of an activity. $u : (A \times R) \rightarrow \mathbb{N}$ returns the resource usage for each activity of a specific resource. The function $c : R \rightarrow \mathbb{N}^+$ specifies the capacity of each resource. In the specific case that is dealt with in this section, the capacity of each resource is always 1. In the same way the resource usage is also only either one or zero, i.e. $u : (A \times R) \rightarrow \{0, 1\}$. Now also suppose that the resource is not from a renewable source, so during the operation of the schedule only one task can make use of an operation.

In this section the problem is not to schedule all tasks while minimizing some objective. Rather the objective is to schedule in such a way that the value of the scheduled tasks is maximized. Let therefore $v : A \rightarrow \mathbb{R}^+$ be a function that returns the value of an activity.

In [14] the idea is proposed to build a graph for all the task in a scheduling problem with edges between task that cannot both be executed. For every two tasks a_i and a_j $(a_i, a_j) \in \text{NO}$ iff task i and j cannot both be executed (for example when they need the same resource). This idea is exploited in this test. The maximum independent set problem is the basis for this problem. Each node now is a task that can be executed. Release dates $\bar{r}$, due dates $\bar{d}$, processing times $\bar{p}$ and precedence constraints P are added to measure the effect of the decomposition. For reference this problem is called the independent set scheduling (ISS).

Consider the mixed integer model for ISS below:

$$\text{maximize:} \quad \sum_{i=1}^{n} x_i v(a_i) \tag{4.9}$$

$$\text{subject to:} \quad x_i + x_j \leq 1 \qquad \forall (a_i, a_j) \in \text{NO} \tag{4.10}$$

$$s_i \geq \bar{r}_i \qquad i = 1, ..., n \tag{4.11}$$

$$s_i + \bar{p}_i \leq \bar{d}_i \qquad i = 1, ..., n \tag{4.12}$$

$$s_i + \bar{p}_i \leq s_j + (1 - x_i)M \qquad \forall (a_i \prec a_j) \in P \tag{4.13}$$

$$x_i \in \{0, 1\} \qquad i = 1, ..., n \tag{4.14}$$

$$s_i \geq 0 \qquad i = 1, ..., n \tag{4.15}$$

The coupling of the integer and continuous variables occurs in the precedence constraints (equation 4.13). Notice that without any precedence constraints the variable s_i can be assigned any value between $\bar{r}$ and $\bar{d}$. In the test below the effect of the coupling is measured by increasing the number of precedence constraints.

For the decision diagram state the custom transition functions for MISP given in Section 2.1.2 are used. For every cut a variable is added to the state according to the threshold formulation (introduced in Section 2.3). One should also note that the objective only relies on the integer variables. The result is that all the generated cuts are of the first form, that is, they can be modeled by a threshold diagram. No cost tuples are used in this problem.

To test the effect of the decomposition on the runtime, a test is set up with different number of precedence constraints. The number of precedence constraints define the amount of coupling between the integer and continuous variables. This test setup is as follows: NO is built in the same way as the edges were set up in the independent set test. The 'edge'-density in this test is set to 85%. This density is chosen because previous results have shown that DDs have good performance on MISP with high density. The number of tasks n is set to 120. (The same test has also been performed with $n = 100$ and $n = 150$. The tests showed similar results and have therefore been left out of this evaluation.) A time horizon is set as $H = 2\log_{10}(n)$.

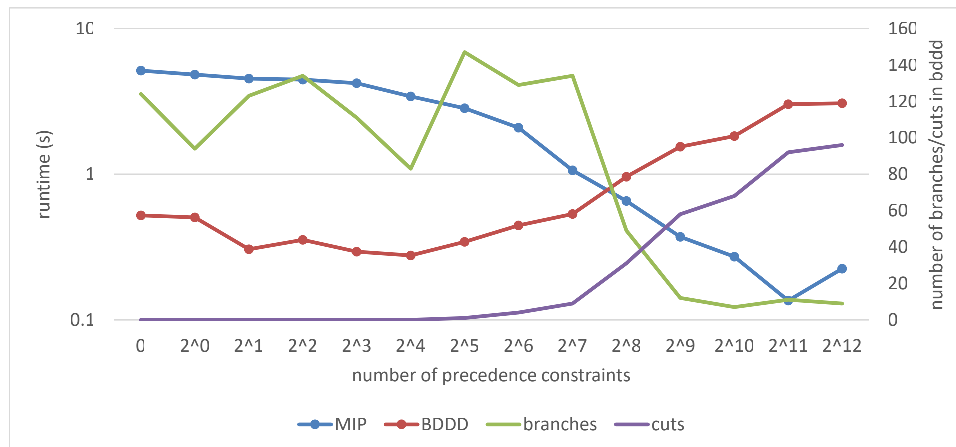

FIGURE 4.4: Performance of MIP and BDDD for ISS. $n = 120$

Release dates are uniformly set in the interval $[0, 0.75H)$, due dates uniformly in the interval $[\bar{r}_i, H)$ and processing times in uniformly in the interval $[0.75(\bar{d}_i - \bar{r}_i), (\bar{d}_i - \bar{r}_i))$. Precedence constraints are created randomly but only in such a way that no circular precedence constraints occur.

The results of this test can be seen in Figure 4.4 (The label `bddd` refers to the benders decomposition with decision diagrams method). Each dot in this figure is the average of ten runs. The number of cuts in the benders decomposition and number of branches in the branch and bound is also plotted. One can see that not every precedence constraint results in a cut. The number of cuts remain small. Furthermore, one of the first things that can be noticed is the large reduction in runtime of MIP with an increased number of precedence constraints. The added precedence constraints at first do not influence the runtime of BDDD that much. When the number of cuts increase one can see that the runtime of MIP decreases while at the same time the runtime of BDDD increases. From the decreasing number of branches one can see that the introduction of the cuts does make the size of the decision diagram smaller. It can be concluded that the increase in runtime of BDDD is caused by the cuts. With even larger number of precedence constraints the number of branches grows even close to one. This also means that cuts are already introduced in or near to the root node of the branch and bound.

As said before, all the cuts are of the first form. A closer look at the cut shows that (most of) the cuts prohibit a partial assignment. What is interesting to see is that a large portion of the constraints involves only one decision variable. This means effectively that this variable is fixed. Most other constraints are also on only a small number of variables. Some of these constraints are also fixing the variables. As an example take a constraint of the form $a_1x_1 + a_2x_2 \leq b$ with both $a_1, a_2 > b$. Other constraints have the form $a_1x_1 + a_2x_2 \leq b$ with $a_1, a_2 \leq b$ and $a_1 + a_2 > b$, which is the same constraint as $x_1 + x_2 \leq 1$ (another independent set constraint). Only very rarely a constraint is generated that is more complex and cannot be written as an independent set constraint.

In Figure 4.5 a break down of the runtime of BDDD is shown. In this figure, `lp_sub` is the time spent solving the LP subproblem; `constraint_time`

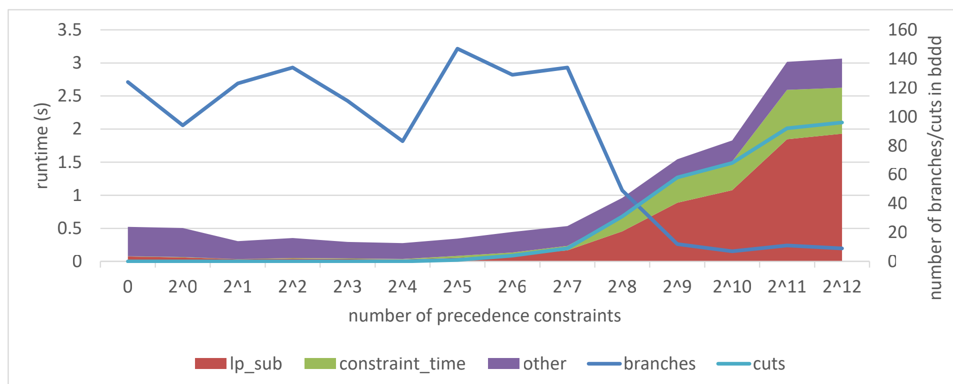

FIGURE 4.5: Break down of the runtime of BDDD for ISS, with $n = 120$ and an increasing number of precedence constraints.

is the time spent applying the cut (changing the restricted diagram by means of the separation algorithm presented in Algorithm 2 and adapted according to Section 3.1.1); `other` is the rest of the running time. This rest consists mainly of the building and solving of the (first) restricted and relaxed diagram per iteration.

What can be seen in Figure 4.5 is that at first `other` is the dominant portion of the runtime of BDDD. When the number of cuts increase `lp_sub` and `constraint_time` increase rapidly. Especially the runtime of the LP subprocedure should be noted. With larger number of precedence constraints the runtime of the LP subprocedure becomes more than half of the total runtime of BDDD.

4.3.2 Market Share Split (adapted)

In the market share problem (first introduced in [18]) retailers are allocated to divisions in such a way that one division controls a fraction f_i of the market share for product i and the other division controls $1 - f_i$. n is the number of retailers, m is the number of products, with $m \leq n$. The demand of retailer j for product i is described by a_{ij}. Let d_i be the amount of product i that represent a market share f_i of the total amount of that product. The feasibility problem checks if such an allocation exists (i.e., such that $Ax = d$, with $x \in \{0,1\}^n$). The optimization problem is what is of interests here. The objective is to minimize the slack required to make a feasible allocation possible. A simplified version is shown below in equations 4.16 - 4.18. The problem is simplified in such a way that the slack is assumed always to be positive.

$$\text{minimize} \quad \sum_{i=1}^{m} y_i \tag{4.16}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij}x_j + y_i = d_i \qquad \forall i \in \{1, ..., m\} \tag{4.17}$$

$$x \in \{0,1\}^n \tag{4.18}$$

This problem has no integer constraints, only continuous constraints. To test the performance of the decomposition random instances of the market share problem are generated with $n = m$ according to the method described in [1]. The coefficients a_{ij} are determined uniformly from the interval $[0, 99]$. And d_i is computes as follows: $d_i = \lfloor \frac{1}{2} \sum_{j=1}^{n} a_{ij} \rfloor$. The results can be seen in Figure 4.6. In this graph the runtime for MIP and DD with Benders Decomposition is shown. In this figure, `count` is the number of branches performed in the branch and bound; and `cuts` is the number of cuts returned by the subproblem. The count and cuts are shown on the secondary axis.

From Figure 4.6 already some conclusions can be drawn. For both MIP and BDDD an exponential increase in the runtime can be seen. The number of cuts remains only small (below 100 cuts). The number of branches however rises quickly. From the results (not in the graph) one can also see that only a small fraction (even for the larger instances below 0.03s) of the time is spent on solving the subproblem and applying the cuts. The conclusion is that most of the runtime is used for branching in the DD-tree.

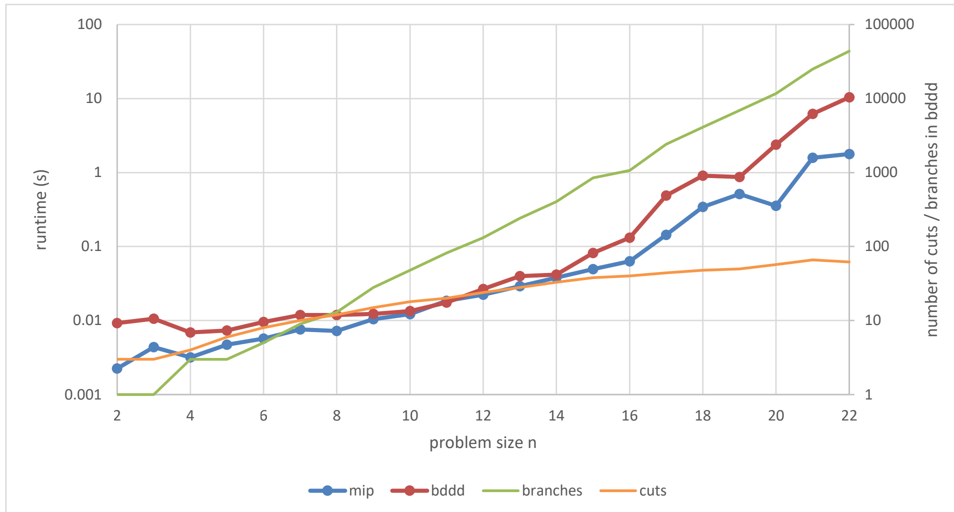

FIGURE 4.6: Performance of MIP and BDDD for the market share split problem with $m = n$

The test results in Figure 4.6 are found by solving the DD without filtering the DD based on the current incumbent. A same test has been performed with filtering based on the incumbent. The results were either the same or worse. The extra effort to check at every need if the incumbent has been exceeded does not pay back in a largely reduced DD. With filtering the DD had roughly the same size. For that reason, incumbent based filtering is also left out of the test presented below.

This test has been performed with running MAKEEXACTCUTSET. Unlike the ISS problem, with this problem the subproblems are easy to solve. The intuition is that the cost of solving extra subproblems is worth the benefit that is gained by the cuts from the subproblems. A comparison of solving AMSS with and without MAKEEXACTCUTSET shows that there is no clear benefit for either method.

In a previous test it was shown that with an increasing amount of dimensions/constraints in the multi dimensional knapsack problem the BDD performed relatively better. For that reason a new test is performed that tests the influence of the number of constraints for the market share split problem. To test this the constraint $n \leq m$ should be omitted. (This means that the problem is no longer a market share split problem.) For ease of notation this problem is called the adapted market share split problem (AMSS). In Figure 4.7 the results are shown with $n = 16$ the number of binary variables, and an increasing number m of continuous variables (or constraints).

From Figure 4.7 one can see that for a fixed number of binary variables the number of branch operations hardly changes. The reason for this is that the larger number of constraints do not result in more cuts on the feasible space (constraints of the first form). Almost all generated cuts are objective cuts (constraints of the second form).

Figure 4.7 also shows that when the number of continuous variables exceeds 2^8 the number of cuts does no longer increase. Both for MIP and BDDD the runtime also stabilizes. The results also show (not in the graph) that for larger values of m the sum of the time spent solving the subproblem

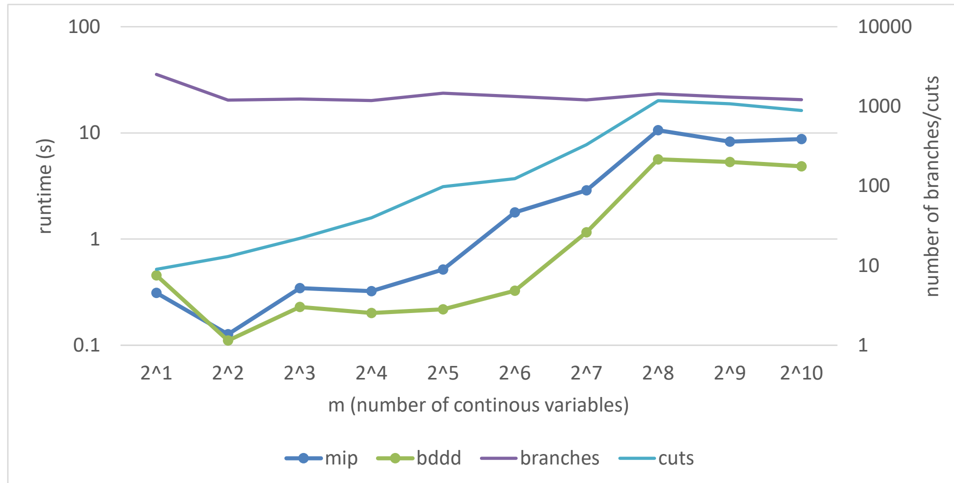

FIGURE 4.7: Performance of MIP and BDDD for AMSS with m on the horizontal axis and $n = 16$.

increases steadily (from 16% at $m = 2^4$ to 57% at $m = 2^{10}$).

Another interesting test is to measure the runtime required to find a first feasible solution. These results with $m = 5n$ can be seen in Figure 4.8. The tests with $m = n$ and $m = 50n$ showed similar results.

What can be seen from this figure is that BDDD finds a first feasible solution much faster than MIP does. The results also show that the number of cuts in the root node do not increase much in comparison to the problem size. The question that remains is the quality of these first solutions. For the instances with $m = 5n$ the relative quality of BDDD's first solution ranges from 5% better than MIP's first (for the smaller instances) to 20% worse than MIP's first (for the larger instances).

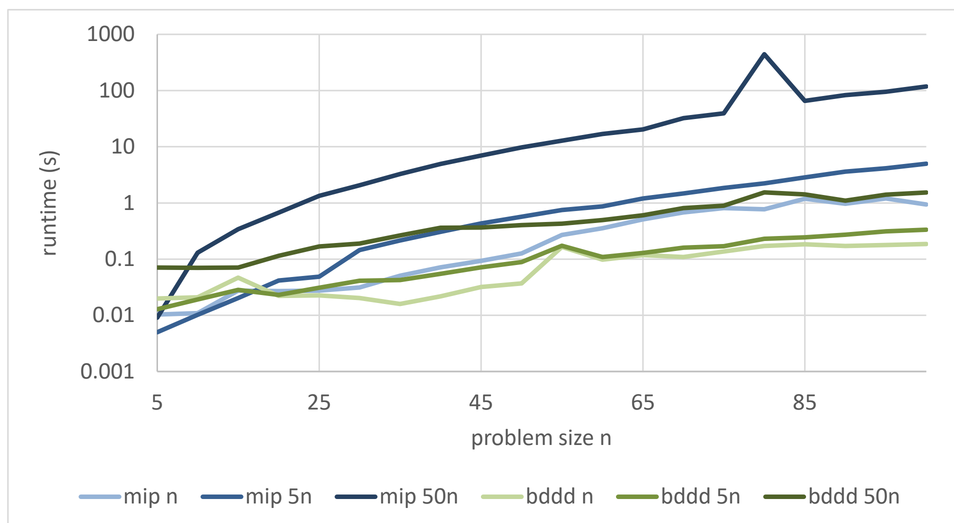

FIGURE 4.8: Runtime required for MIP and BDDD to find a first feasible solution for AMSS with m equal to n, $5n$ and $50n$.

4.3.3 Other problems

Apart from the two problems mentioned above (ISS and AMSS), other problems have been evaluated also. The sections below discuss what can be learned from the evaluation of these problems.

Map labelling problem

The map labeling problem (MLP) was mentioned as an application for the work presented in [9]. Because of its use of Benders decomposition, this problem was also evaluated with BDDD. The method presented in [9] however, is mainly aimed at reducing the negative effect of big-M constraints in large MIP problems by means of Benders decomposition. MLP was evaluated by using the model presented in [13]. This model uses large set of discrete variables. Because of the large number of variables (in comparison) to the number of constraints, the feasible space (of discrete configurations) is large. This makes the problem difficult to solve for DDs. So far, no positive results can be reported.

Traveling salesman problem with time windows

The traveling salesman problem with time windows (TSPTW) was also evaluated, but so far no positive results can be reported. This is in contrast with the positive results presented in [4] for TSPTW solved with a DD-based approach. The model presented in [4] is a multi-valued decision diagram. The current implementation of BDDD however, is only able to deal with binary variables and because of time limits this is not yet extended to multi-valued discrete variables. To solve TSPTW a model was used with binary variables (one for each pair of cities). The resulting DD was too large to provide good results.

Resource constrained project scheduling problem

The ISS problem introduced before, is a simplified version of RCPSP. In [14] several methods are proposed for solving scheduling problems. The MIP model proposed in [14] is quite extensive. It allows for solving a large range of scheduling problems, among which RCPSPs. This MIP model is based on the independent set problem.

Two problems were encountered during the testing with this problem. The first problem is the large number of discrete variables. The second problem is the low density of the non-overlapping graph. Because the graph is repeated for every time step (or priority slot), the density of the non-overlapping graph becomes small. As is known DD-based optimization outperforms MIP with MISP only for graphs with high density.

Chapter 5

Discussion

In this chapter the results of the previous chapter are discussed. In the first section of this chapter some conclusions are drawn from the results. In the second section of this chapter recommendations for future work are given.

5.1 Conclusions

As mentioned in the introduction, the main contribution of this thesis is twofold. The first contribution is the idea to combine DD-based optimization with Benders Decomposition (BDDD) to solve hard mixed integer problems. The second contribution (used in the first) is to use cost tuples in DDs to deal with convex piecewise linear objectives.

The evaluation of these two contributions is discussed here, and is structured as follows: The result in Chapter 4 is divided in three parts: the threshold diagram, the cost tuples and the decomposition. The results of these three sections help to answer the subquestions asked in the introduction. Each of those four questions is answered below, and finally the research question is answered. Can BDDD be used as a method for solving mixed integer problems, and if so, for what kinds of problems can it be used?

In Section 4.1 runtime results are shown for the threshold diagram state formulation. Here these results are summarized. The runtime performance of a BDD with a threshold formulation (BDDT) was compared to MIP and to a BDD with a custom state description (BDDC) for the Maximum Independent Set Problem (MISP). The results show that the BDDC outperforms the other two at great length. BDDT performs better than MIP for the smaller instances, but both show an exponential increase in runtime with increasing graph size. With larger instances the increased size of the state description (on top of the exponential growing search space) becomes the main reason for the increase in runtime and eventually this means that the BDDT solver is slower than MIP for large instances of MISP. This result is confirmed by using a MIP formulation for the MISP with fewer constraints. Both MIP and BDDT performed faster using a MIP formulation with fewer constraints.

The performance of BDDT is also tested with the multi-dimensional knapsack problem (MDKP). From this problem can be learned that BDDT performs only good when the feasible space is small enough. When this is the case, BDDT may perform much better than MIP for randomly generated MDKP instances.

From the evaluation of the cost tuple method, the conclusion can be drawn that the method has a good performance. The runtime increases

close to linear in the number of constraints on the objective. These constraints make the objective a convex piecewise linear objective. The increase in runtime is only slightly worse than the increase in runtime that is seen with MIP. A drawback is that the method only works on diagrams that are not reduced.

The number of constraints that are generated during the decomposition depends of course on what kind of problem is solved. With the introduced scheduling problem (Independent Set Scheduling, or ISS) that is closely related to MISP, the number of cuts remains small in comparison to the number of constraints on the continuous variables. Most derived cuts are independent set constraints. Only a small number of constraints is more complex. This means that for ISS the cuts need not be modeled by a threshold diagram. Another solution, for example, would be to model these cuts as extra edges in the graph. The constraints can be modeled using another method than the threshold diagram. This conclusion can probably also be generalized to other problems that have only simple generated cuts.

In a problem related to the market share split problem (AMSS) the number of cuts was much higher. All of these cuts involved all integer decision variables. With AMSS the objective is solely expressed in the continuous decision variables. The result is that almost all cuts are cuts on the objective value. This means that the feasible space of the integer decision variables is barely reduced.

The results in this thesis show that BDDD can be used for solving mixed integer problems. In the tests, two specific mixed integer problems have been solved by means of this method. For the ISS problem BDDD performed better than MIP under specific circumstances. With an increasing amount of (continuous) precedence constraints the runtime of BDDD increases slowly, and the runtime of MIP decreases rapidly. Another way of seeing this, is that as long as the integer constraints are 'dominating' the problem BDDD can outperform MIP. Problems where the integer variables only play a marginal role (for example only to facilitate big-M constraints) are therefore not suitable to be solved with BDDD. From the ISS problem one can also learn that the subproblem size should not be too big.

In the runtime analysis it can be seen that either one of two elements becomes the dominant factor in the runtime of BDDD. With ISS the linear programming (LP) subproblem execution is the most important factor of the runtime. With AMSS this is not the case because the LP subproblem is much easier. With AMSS the feasible solution space is much bigger and therefore the number of branches in the branch and bound much higher. For most instances of AMSS the branch count was the dominant factor in the runtime.

From the AMSS problem the conclusion can be drawn that this problem class is difficult for both MIP and BDDD. Currently the two solvers have approximately the same runtime results (BDDD outperforms MIP slightly for the problems with fewer integer decision variables and MIP outperforms BDDD slightly for the problems with more decision variables). This is a surprising result if you consider that AMSS has no integer variables in the objective and no integer constraints. Another conclusion that can be drawn from AMSS, is that even with the decomposition BDDD is able to find a first feasible integer much faster than MIP. This first solution of BDDD has

roughly the same value as the first solution of MIP . This advantage of DDs over MIP therefore still holds. The long runtime for BDDD is in proving optimality. From the results of BDDD can be seen that almost a full enumeration is required to solve the problem to optimality.

The conclusion is that BDDD can be used to solve mixed integer problems, but is currently mainly useful for problems that satisfy very specific properties. Suitable problems should have integer decision variables that play a dominant role in the problem model. Modeling specifically for DDs is recommended, instead of using a MIP model. DDs still have the advantage that non-linear and non-convex constraints are allowed. Apart from that, BDDD can be useful for MIP problems that do not have a good LP relaxation such as AMSS.

The proposed method BDDD has several advantages. One advantage is, that in contrast to other DD decomposition techniques such as the Lagrangian decomposition [4], BDDD also provides a way to divide not only the set of constraints, but also the set of variables. Another advantage is that BDDD is proven to work in a DD-based branch and bound technique. A possible advantage of BDDD is that it allows for another way of modeling MIP problems. A MIP problem can be modeled 'around' a typical IP problem that is solved by DDs. This is a whole other way of thinking than starting with an LP problem and adding integer variables.

5.2 Recommendations

There are several topics that we suggest as future work. These suggestions are explained here.

Combinatorial Benders Cuts

In the method proposed in this thesis, the cuts generated by the Benders decomposition have the form of a knapsack constraint and are modeled by means of a threshold diagram. In [9] another form of cuts is proposed, namely combinatorial Benders (CB) cuts. This kind of cut does not use coefficients in the constraint. The hypothesis is that for this kind of cut a better custom transition function can be formulated than the threshold diagram.

Also, in [9] a method is proposed to find an Irreducible Infeasible Subsystem (IIS) using a fast heuristic. A benefit of this heuristic is that it can find several violated CB cuts with one run of the subproblem instead of only one cut per run. The hypothesis is that the addition of these two techniques would allow for a smaller state description, a transition function that leads to a more compact diagram, and would also lead to less subproblem invocations.

Scheduling problems with multi-valued decision diagrams

One of the powers of DDs has not been fully utilized in this thesis, namely the power of sequence constraints with multi-valued decision diagrams. In [4, 5] the travelling salesman problem with time windows (TSPTW) (and optionally with precedence constraints) is solved with decision diagrams. In this test the time parameters are integer.

One remaining challenge is to solve TSPTW while minimizing total tardiness. The difficulty with this problem is that arrival time is no longer constrained by the due dates, and can no longer be implicitly incorporated in the decision diagram. Instead, the arrival time is part of the objective. An option is to model the arrival times as continuous variables. Now the assignment of the arrival times is the subproblem. BDDD can be used to derive the cost tuples for the decision diagram.

Dividing an integer variable set

The application of the method presented here is explicitly directed to mixed integer problems, with both continuous and discrete variables. However, there is no specific reason why Benders decomposition should be used to divide the set of variables in a discrete and a continuous set. The decomposition can also be used to decompose a large integer problem.

The subproblem of a decomposed integer problem of course cannot be solved by means of an LP. Neither can its dual be trivially formulated. Another method for solving the subproblem should be used. Using a logic based Benders decomposition is an option, as is done for example in [7, 8].

Depth first branch and bound

The reduction of decision diagrams that are compiled by a top-down compilation is not straight forward. Especially for threshold diagrams, a large reduction in diagram size can be achieved by compiling it depth first instead, as is done in [2]. By compiling the diagram depth first, the theoretical polynomial bound of the size of the diagram in terms of the threshold weights applies. This is not the case with the two algorithms that are used in BDDD, namely top-down compilation, and (top-down) compilation by separation.

A recommended step is to study the depth first compilation of decision diagrams in a branch and bound setting. A requirement for this is a method for depth first compilation of a restricted and a relaxed diagram. Also a depth first compilation by separation method should be formulated. With depth first compilation methods that work with DD-based branch and bound, the threshold diagram formulation can be used more efficiently.

Appendix A

Implementation details

In this appendix some implementation details and choices that are left unmentioned in the main text are further explained.

A.1 State description

One of the drawbacks of the threshold formulation that is mentioned is size of the state description. For each constraint in the model the state description contains one variable. A newly created node gets a state that is computed from the preceding node and the edge label. With a large state description, this can be a costly operation. This operation has to be performed frequently, and therefore it is worthwhile to implement it as efficiently as possible.

A simple technique that can be used to speed up the computation is to pre-compute the state changes per layer. With a fixed variable ordering for each of the diagrams, each layer has a fixed change. This solution works as follows: for each layer $j \in \{1, ..., n\}$ and each possible label $d \in D_j$, build a map M_{jd} of pairs (c, a). In this pair, the variable c is the constraint index and a is the change in the slack of constraint c when $x_j = d$. In the mapping M_{jd} all pairs (c, a) with $a = 0$ are left out, because this means that there is no change in the slack. When the state of a new node is now computed, the state of the preceding node is copied and for every pair in the corresponding M_{jd} map, the state variable with index c is updated by a. Early tests showed that for the Maximum Independent Set Problem (MISP) this technique approximately halved the runtime for dense graphs.

An effort was made to improve on this even more. The large state description was still a problem and therefore a sparser state description was tried. This sparse state description contained only slack variables for the currently active constraints. A constraint is active in a layer when the layer is in between the first and the last decision variable of the constraint. Before a constraint becomes active its slack is still equal to b, the right hand side of the constraint, and storing this in the state description is redundant. When a constraint is inactive again (after its last decision variable has been decided), it is also unnecessary to store the slack of this constraint any longer. The constraint cannot be violated any more.

To implement this spare state method, more pre-calculation is needed. A mapping is created for each layer that maps each active constraint to an index in the state. This index is chosen in such a way that during the active period of a constraint this index does not change. There are two options for a newly active constraint. When another constraint becomes inactive, the

new constraint gets the index of the inactive constraint. When there is no inactive constraint to be replaced, the state size is increased by one and the last index is given to the newly active constraint.

For the sparse state method, the variable ordering is important. For this reason a heuristic was developed meant to minimize the maximum number of active constraints.

In Figure A.1 results are shown for a comparison between the two BDD approaches, both with the threshold diagram formulation. BDD is with a full state description, BDDs with the sparse state description. The random graphs are generated as before in the tests presented in Section 4.1.1. The edge density is 80%. Again two formulations for MISP are tested, the one with one constraint per edge, and the other with one constraint per clique.

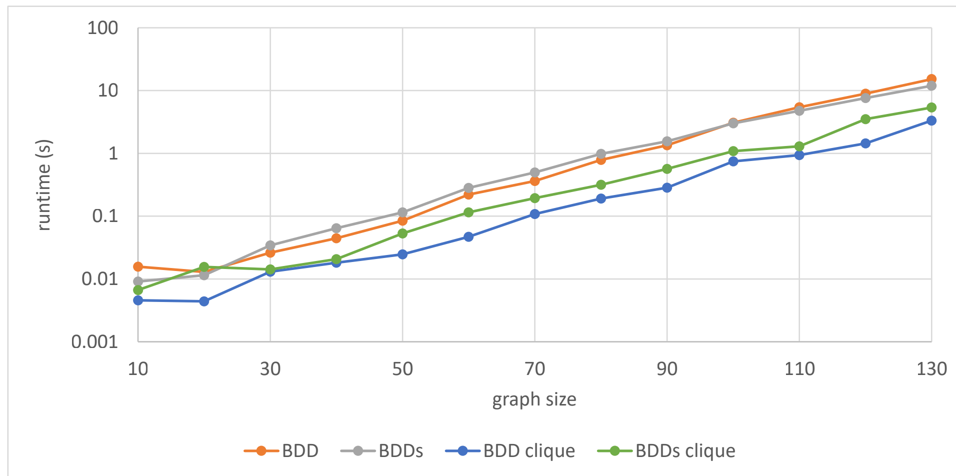

FIGURE A.1: Runtime of BDD and BDDs for MISP for increasing graph sizes.

In these results one can see that in some cases BDDs performs slightly better than BDD, and in others it performs (much) worse. Similar results were achieved for the Vertex Cover problem. Because of these results , and the complexity that BDDs introduces, the choice was made to continue with full state descriptions.

A.2 Variable ordering

The size of a BDD is heavily dependent on the variable ordering. Therefore variable ordering can also not be left out of the discussion here. However, variable ordering is not the main point of concern in this thesis. Finding the optimal variable ordering is an NP-hard problem. In the presented tests, only for the threshold diagram BDD MISP solver was the default variable ordering changed. In all other tests (BDDc for MISP, MDKP, ISS and AMSS) the variable ordering is left unchanged. The change in variable ordering by a heuristic however, showed almost no improvement. Because of this result, and because variable ordering is not the main focus of this thesis, no further effort was made to improve on this.

Bibliography

[1] Karen Aardal et al. "Market split and basis reduction: Towards a solution of the Cornuéjols-Dawande instances". In: *INFORMS Journal on Computing* 12.3 (2000), pp. 192–202.

[2] Markus Behle. "On threshold BDDs and the optimal variable ordering problem". In: *Journal of Combinatorial Optimization* 16.2 (2008), pp. 107–118.

[3] Jacques F Benders. "Partitioning procedures for solving mixed-variables programming problems". In: *Numerische mathematik* 4.1 (1962), pp. 238–252.

[4] David Bergman, Andre A Cire, and Willem-Jan van Hoeve. "Lagrangian bounds from decision diagrams". In: *Constraints* 20.3 (2015), pp. 346–361.

[5] David Bergman et al. *Decision Diagrams for Optimization*. 2016.

[6] David Bergman et al. "Discrete optimization with decision diagrams". In: *INFORMS Journal on Computing* 28.1 (2016), pp. 47–66.

[7] André A Ciré, Elvin Coban, and John N Hooker. "Logic-based Benders decomposition for planning and scheduling: a computational analysis". In: *COPLAS Proceedings* (2015), pp. 21–29.

[8] André A Ciré and John N Hooker. "The Separation Problem for Binary Decision Diagrams." In: *ISAIM*. 2014.

[9] Gianni Codato and Matteo Fischetti. "Combinatorial Benders' cuts for mixed-integer linear programming". In: *Operations Research* 54.4 (2006), pp. 756–766.

[10] *GLPK (GNU Linear Programming Kit)*. 2006. URL: http://www.gnu.org/software/glpk.

[11] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: http://www.gurobi.com.

[12] Erwin Kalvelagen. *Benders decomposition with Gams*. http://web.stanford.edu/class/msande348/papers/bendersingams. [Online; accessed 29-March-2017]. 2004.

[13] Gunnar W Klau and Petra Mutzel. "Optimal labeling of point features in rectangular labeling models". In: *Mathematical Programming* 94.2 (2003), pp. 435–458.

[14] Sylvain Mouret, Ignacio E Grossmann, and Pierre Pestiaux. "Time representations and mathematical models for process scheduling problems". In: *Computers & Chemical Engineering* 35.6 (2011), pp. 1038–1063.

[15] Canh Ngo et al. "Multi-data-types interval decision diagrams for XACML evaluation engine". In: *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*. IEEE. 2013, pp. 257–266.

[16] Alexandre Niveau et al. "Knowledge Compilation Using Interval Automata and Applications to Planning." In: *ECAI*. 2010, pp. 459–464.

[17] Ingo Wegener. *Branching programs and binary decision diagrams: theory and applications*. SIAM, 2000.

[18] H.P. Williams. *Model building in mathematical programming*. John Wiley & Sons Australia, Limited, 1978.