

# Shaping and Trimming Branch-and-bound Trees\*

Philipp M. Christophel<sup>†</sup>      Imre Pólik<sup>‡</sup>

September 5, 2017

## Abstract

We present a new branch-and-bound type search method for mixed integer linear optimization problems based on the concept of *offshoots* (introduced in this paper). While similar to a classic branch-and-bound method, it allows for changing the order of the variables in a dive (*shaping*) and removing unnecessary branching variables from a dive (*trimming*). The regular branch-and-bound algorithm can be seen as a special case of our new method. We also discuss extensions to our new method such as choosing to branch from the top or the bottom of an offshoot. We present several numerical experiments to give a first impression of the potential of our new method.

## 1 Introduction

In this paper we are discussing mixed integer linear optimization problems (MILP), i.e., optimization problems with a linear objective function, linear constraints, and integrality restrictions on some or all of the variables. Most of the techniques described in the paper generalize naturally to problems with non-linear constraints as well (MINLP). The typical approach to solve any optimization problem with an integrality restriction on the feasible domain involves variants of the branch-and-bound algorithm first described for general integer optimization by Land and Doig in [9]. For details about the origins of branch-and-bound, see also [4]. The branch-and-bound method is at the core of every software to solve mixed integer optimization problems and is successfully used to solve a variety of practical problems. But it is also known that in the worst case the branch-and-bound method will enumerate all possible solutions, leading to a disastrous performance. In Example 1 we show such a case.

---

\*This paper is a written version of the talk given by the first author at the MIP 2017 Workshop in Montreal, Canada.

<sup>†</sup>philipp.christophel@sas.com, SAS Institute, Inc.

<sup>‡</sup>imre@polik.net, SAS Institute, Inc.

**Example 1** Consider the following integer optimization problem with three binary variables:

$$\begin{aligned}
 \min \quad & x_1 - 2x_2 - 6x_3 \\
 & -3x_1 - 4x_2 - 2x_3 \geq -8 \\
 & 3x_1 - 4x_2 - 2x_3 \geq -5 \\
 & -3x_1 + 4x_2 - 2x_3 \geq -4 \\
 & 3x_1 + 4x_2 - 2x_3 \geq -1 \\
 & x \in \{0, 1\}^3
 \end{aligned} \tag{1.1}$$

We solve this example with a depth-first branch-and-bound algorithm where in the left nodes variables are fixed to one. In this example, it does not matter which branching variable selection method is used since there is only one fractional variable in each node and a traditional branch-and-bound method chooses the branching variable only from the fractional variables. In Figure 1 we show the branch-and-bound tree resulting from this example. We show the objective value of the LP relaxation below each node and the solution values for the three variables to the right of each node. The number inside the nodes shows the order in which they are processed.

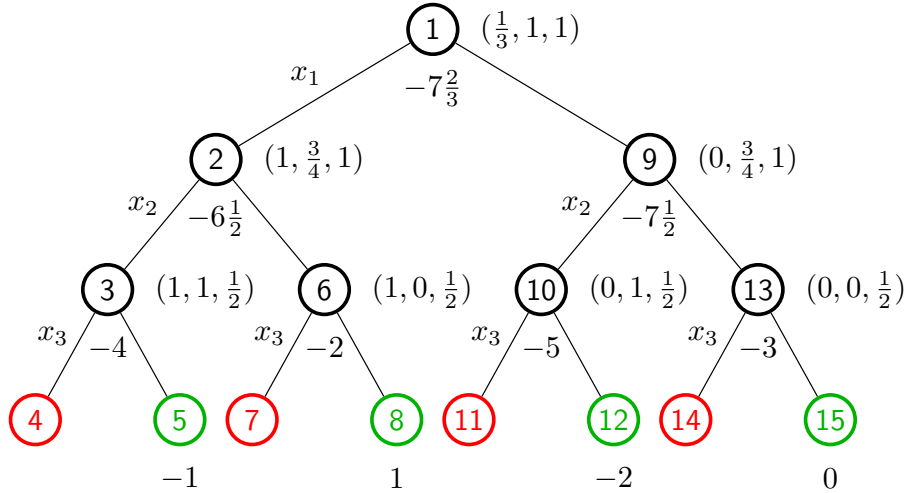


Figure 1: The branch-and-bound tree for the optimization problem 1.1.

Note that the optimal objective value is  $-2$  but it is obtained only after 12 nodes have been processed and proving its optimality requires 15 nodes, the maximal number of nodes possible, which corresponds to enumerating all possible solutions, i.e., the leaf nodes of the tree in Figure 1. Note that the leaf nodes are alternating between infeasible and integral nodes. This is the reason why we have such a bad tree in this example; the enumeration ended up with different

types of leaf nodes next to each other. Hence the branch-and-bound method has no chance to prune several leaf nodes together early on at a common ancestor.

Research on branch-and-bound algorithms has put a huge emphasis on making the decisions in the algorithm in such a way as to avoid enumeration of large parts of the solution space. Especially selecting the branching variables has been studied extensively (see, for example, [1]), because which branching variable is chosen determines the shape of the tree of a branch-and-bound method and thus how many nodes need to be processed. In this paper we will present an implementation for a branch-and-bound method that follows a different approach in which the shape of the tree can be changed. This means that we want to have a branch-and-bound algorithm where the decisions on the branching variables can be deferred to a time when potentially more information is available to make these decisions. Example 2 demonstrates that the same problem from Example 1 can be solved more efficiently if the branch-and-bound tree has a different shape.

**Example 2** In Figure 2 we show the branch-and-bound tree for the same mixed integer optimization problem as in Example 1 with the only difference that we reversed the order in which we branched on the variables, i.e., we branched on  $x_3$  first although it is not even fractional. In this case we only need to process 9 nodes; we show the remaining nodes (without a number) just to highlight the structure of the tree.

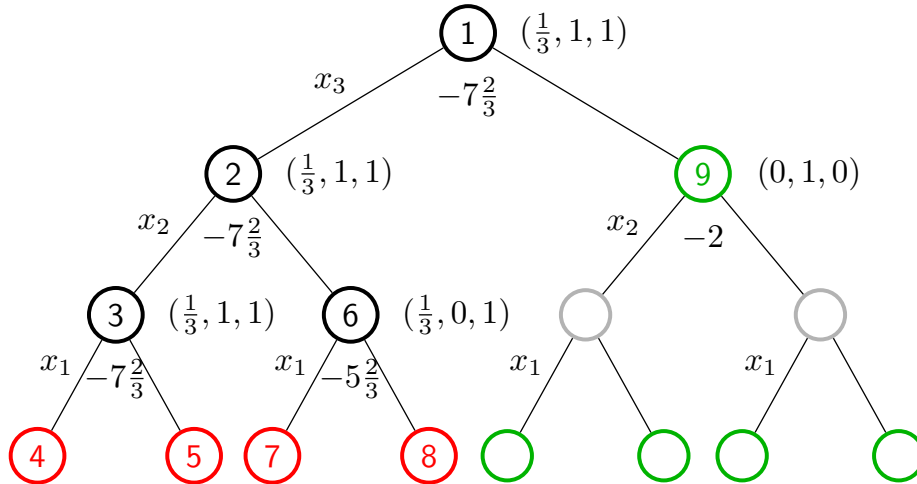


Figure 2: Problem (1.1) solved with a different branch-and-bound tree.

Note that now the infeasible leaf nodes are all on the left side of the tree and all the integral nodes are on the right side of the tree. The integral leaf nodes do not all have to be processed because the common ancestor, node 9, covers them all. Also, if we can detect the integer infeasibility of node 2 (for example

*with probing or some other node presolver technique), then we can prune all the nodes below it and solve the problem even faster. The lesson is that it is better to have a tree in which nodes with similar properties are the leaf nodes of subtrees so that they can be dealt with at a common ancestor.*

The question here is twofold. First, we need to reshape the tree if we believe that the current tree structure is inefficient. The second question is equally important: we need to do this in an efficient way, so that we can preserve most of the advanced techniques that make branch-and-bound implementations perform well in practice. While it is possible to simply throw away most of the tree and roll back to the last known good decision point (or variants thereof called restarting as, for example, discussed in [1]), we want to explicitly look into other possibilities here.

There has been some research in branch-and-bound methods with an adjustable (sometimes called dynamic) tree. The earliest we are aware of is by Glover and Tangedahl [6]. Chvátal in [3] and then Hanafi and Glover in [7] revisited the topic. These papers give valuable insights into alternative methods for solving mixed integer optimization problems but unfortunately do not discuss the implementational challenges. Furthermore, resolution search from [3], for example, is not similar enough to a classic branch-and-bound method such that many of the methods modern solvers successfully use to solve problems today are not directly applicable.

## 2 Diving, Shaping, and Trimming

In this section we discuss three very important concepts for the remainder of this paper: diving, shaping, and trimming. We do so using a depth-first branch-and-bound method because these concepts are easier to explain in this method and are also a natural extension to it.

Depth first branch-and-bound (sometimes also called last-in-first-out, i.e. LIFO, branch-and-bound) is a variant of branch-and-bound where the next node processed is always the most recent node added to a stack of open nodes. In practice it is possible to store the open nodes with a stack of bound changes. The depth-first branch-and-bound method also minimizes the number of open nodes. The result is a very memory-efficient branch-and-bound method.

In the depth-first branch-and-bound method we repeatedly go down the tree only changing one variable at a time. We call this process of going down a tree *diving*. Another advantage of the depth-first branch-and-bound method is that the LP relaxations during diving can be solved very efficiently using a dual simplex algorithm where most data structures (most importantly the factorization of the basis matrix) can be kept up to date. We call this *hotstarting* the dual simplex to express that it is even better than *warmstarting*, which typically implies that a known dual feasible basis is used to initialize the dual simplex algorithm. When backtracking in the depth-first branch-and-bound we cannot use hotstarting, but since the difference between nodes is typically

small we can warmstart from the last basis instead of resolving from scratch. Since diving is much more efficient, current implementations of non-depth-first branch-and-bound methods also use it to process nodes quickly and only do a full node selection if the current dive does not seem promising anymore.

The disadvantage of the depth-first branch-and-bound method is that the problem described in the introduction is aggravated: a bad decision early on can result in a very bad enumeration tree and thus long running time or a failure to solve the problem within some resource limitation. But it is also much easier to revise earlier decisions and change the order of bound changes in a dive. Notice that for the status of the final node in a dive the order in which the variables were fixed does not matter. The order of the bound changes in a dive in some sense defines the shape of the branch-and-bound tree. Hence we call changing the order of the bound changes in a dive *shaping*. Since in a depth-first branch-and-bound we store the bound changes in a stack we can decide to undo them in a different order than we did them during the dive. The only thing we have to keep in mind is that we can only change the order of the bound changes up to the last node where we have already explored the other side of the bound change.

In a depth-first branch-and-bound algorithm a dive has to end in a pruned node. A node is pruned either because the LP relaxation is infeasible or because the objective value exceeds the cutoff <sup>1</sup>. It is possible that a situation occurs where a dive contains more bound changes than are strictly necessary to prune a node. In this case it is possible to remove the unneeded bound changes from the dive before backtracking. Since this trims the dive down to a smaller set of bound changes we call this *trimming*.

There are a number of ways to trim a dive. For problems with general integer variables it is possible to remove multiple bound changes on the same bound of the same variable and keep only the tightest one. It is also possible to use reduced cost or Farkas certificate values to trim dives. In fact, this problem is identical to the one we are facing when trying to identify an irreducible infeasible system (IIS), so all the reduction techniques in that domain apply readily to our setup; see [2] for details. In the following sections we will sample a few methods.

Shaping and trimming clearly can improve a depth-first branch-and-bound implementation a lot, and the implementational complexity is very low. For shaping, the obvious difficulty is to come up with good rules on which bound change should be undone first. But our experience has been that even simple rules already lead to an improvement. For trimming, the trade-off is between time spent trimming the tree and simply processing nodes. But here as well simple strategies already yielded benefits so that it should be possible to improve any depth-first branch-and-bound implementation not making use of trimming significantly.

The only downside is that if node presolving techniques are used in a depth-first branch-and-bound method it is necessary to keep track of implied bound

---

<sup>1</sup>The case of an integral solution can be seen as first establishing a new cutoff and then pruning the node.

changes separately from the actual branching decisions. As a result, during backtracking some tightenings from node presolve have to be redone.

The concepts of shaping and trimming the tree already appear in principle in [6], but that paper does not include any implementational considerations.

### 3 A New Branch-and-bound Method

In this section we present the basic idea of a new branch-and-bound method that allows for shaping and trimming but is not a depth-first method. The fundamental idea is to perform a branch-and-bound method on objects we call *offshoots* instead of performing it on individual nodes. An offshoot (see Figure 3) is an object that represents a collection of nodes in a tree. It consists of a top node  $s$ , represented by a set  $F$  of initial bound changes, with an attached set  $D$  of bound changes representing a dive in the branch-and-bound tree. Applying both the initial bound changes in  $F$  and the bound changes in the dive  $D$  has to result in a node  $t$  that can be pruned, either because it is infeasible or because its objective value exceeds the current cutoff.<sup>2</sup> Note that the order of the bound changes in  $D$  is not determined,<sup>3</sup> only the set of all the bound changes needed to reach a terminal node.

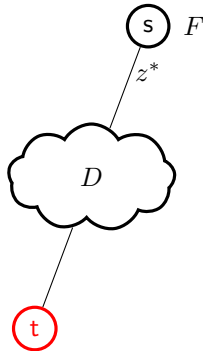


Figure 3: The structure of an offshoot.

Instead of storing a set of open nodes that still need to be processed we store a set of open offshoots. An offshoot is considered open if it has bound changes in its dive that have not been processed. Once the list of open offshoots is empty the problem is solved.

This new method begins with creating a first offshoot for which the set  $F_0$  of initial bound changes is empty. Then it performs a dive until it reaches a node that can be pruned and stores the bound changes of this first dive in the set  $D_0$  of the first offshoot. Then the first offshoot is added to the list of open offshoots.

<sup>2</sup>The cutoff is derived from the currently best known primal feasible solution.

<sup>3</sup>In a practical implementation we can remember the original order of bound changes so that we can use the intermediate objective values to prune undisturbed nodes inside an offshoot.

From now on, in each iteration, the method selects an offshoot from the list of open offshoots as parent offshoot  $p$  for a new offshoot  $k$  to create. The method also needs to select a bound change to process associated with an offshoot variable  $i$  from the list  $D_p$  of unprocessed dive bound changes of its parent. The initial set of bound changes for the new offshoot  $k$  is  $F_k = F_p \cup (D_k \setminus \{x_i \leq b\}) \cup \{x_i \geq b + 1\}$  if the bound change for the selected variable was branching down or  $F_k = F_p \cup (D_k \setminus \{x_i \geq b\}) \cup \{x_i \leq b - 1\}$  if it was branching up. The new offshoot starts with a node that corresponds to a right node of the dive but since we can freely choose from all bound changes in the dive it might be a right node that does not correspond to any of the dive nodes that were processed when the offshoot was created. This choosing of the variable from the dive corresponds to shaping the tree.

After creating the initial node of the new offshoot we solve the LP relaxation of the top node in the new offshoot. If the top node can be pruned we proceed by selecting a new parent offshoot right away. Otherwise we store the objective value as the top bound  $z_k^*$  of the new offshoot. Then we perform a dive until we reach a node that can be pruned either because it is infeasible or because its objective value exceeds the current cutoff. If we encounter a new primal feasible solution we update the cutoff. When updating the cutoff we can also remove all open offshoots for which the top bound exceeds the cutoff.<sup>4</sup>

In this setup we can also easily perform trimming. As mentioned before, this can be done, for example, by removing multiple bound changes on the same bound of a variable (only in the case of general integer variables) or by inspecting the dual information of the pruned node. To specify in more detail: the dual information vector  $r$  is either the reduced cost vector for cutoff nodes or the Farkas certificate for infeasible nodes. An upper bound change on variable  $i$  can be removed if  $r_i \geq 0$ , and a lower bound change on variable  $i$  can be removed if  $r_i \leq 0$ .

After trimming the dive we can store the new offshoot in the list of open offshoots and remove the parent offshoot if all the bound changes in its dive have been processed.

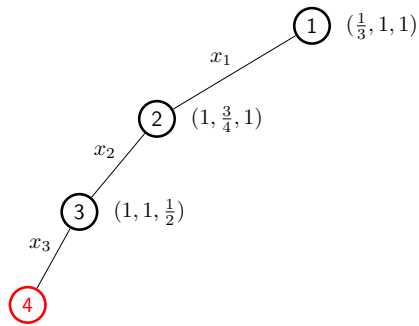
This continues until the list of open offshoots is empty. Figure 4 shows an example where the new method is applied to Example 1.

## 4 Improvements and Extensions

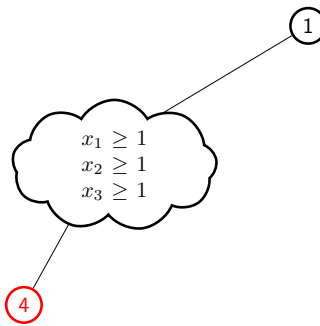
As with many similar methods it is necessary to improve and extend our new method to get the best possible performance. In this section we list some more or less obvious ways to overcome some of the weaknesses of the new method.

---

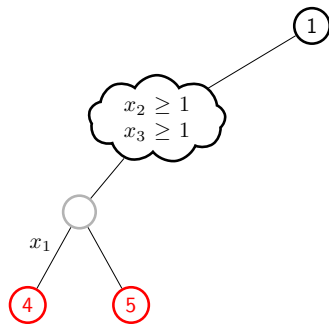
<sup>4</sup>In addition, we can also remove those nodes inside offshoots that have not been disturbed yet if their objective value exceeds the cutoff.



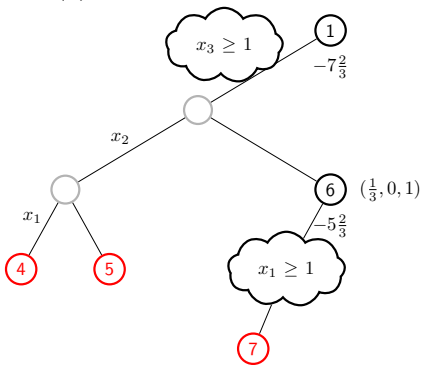
(a) The initial dive ...



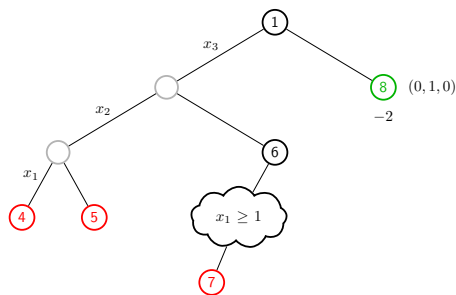
(b) ... becomes the first offshoot.



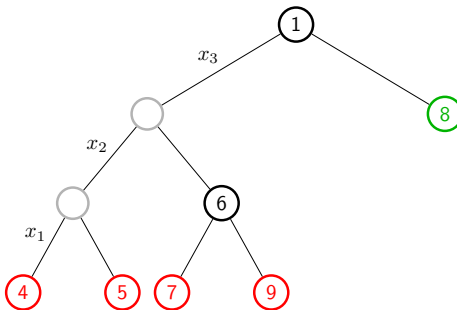
(c) We choose  $x_1$  as the first offshoot variable but the top node of the new offshoot is immediately infeasible.



(d) As the second offshoot variable we choose  $x_2$  and create a second open offshoot with the top node labeled 6.



(e) We have two offshoots to choose from. Since its top bound is better we choose the first offshoot and choose the last remaining bound change. The resulting top node of the new offshoot (8) is integer so we do not create a new offshoot.



(f) We choose the only bound change in the only open offshoot and the resulting offshoot is infeasible in the top node. The method stops with optimal objective value  $-2$ .

Figure 4: A step-by-step example of the new method.



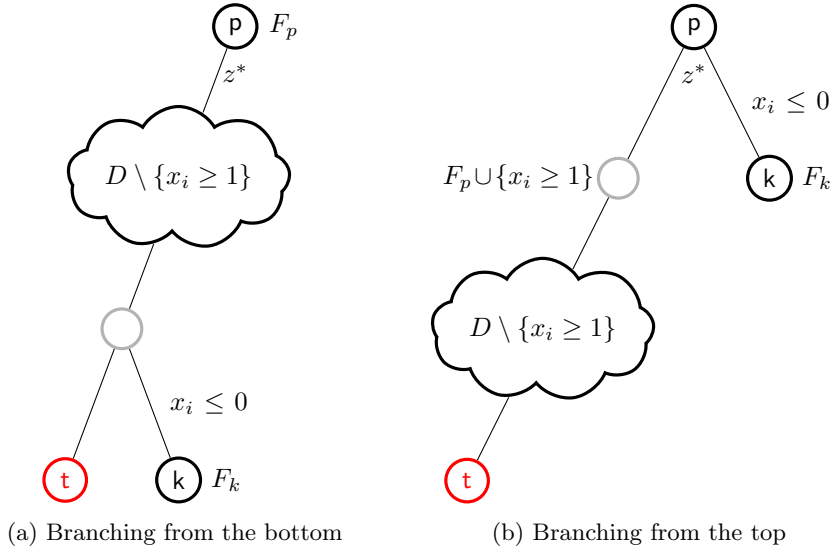


Figure 5: The two ways to branch illustrated.

#### 4.1 Branching From the Top

In the description of the method in the previous section we only added new offshoots below their parent. This can be seen as *branching from the bottom* of an offshoot. It is also possible to *branch from the top* of an offshoot. Then the new offshoot inherits only the bound changes its parent had at the top and additionally exactly one bound change from the dive flipped to the other side. The parent is then adjusted as well and one of the bound changes is moved from the dive to the top. To be precise, the initial set of bound changes for the new offshoot  $k$  is  $F_k = F_p \cup \{x_i \geq b + 1\}$  if the bound change for the selected variable was branching down or  $F_k = F_p \cup \{x_i \leq b - 1\}$  if it was branching up. Figure 5 illustrates both types of branching next to each other.

The big advantage of this additional level of flexibility is that we can decide which type of branching to use based on how sure we are that an offshoot variable is a good choice. If we are not sure whether a bound change will have large impact and hence should be at the top of the tree we can choose to branch from the bottom to minimize the effect if we made a bad choice. If, on the other hand, we have a strong indication that a bound change will have a huge impact and should be at the top of the tree, then we can branch from the top.

#### 4.2 Creating Offshoots and Advanced Trimming

For the correctness of the method it is not necessary to create offshoots by diving. Any method that creates a set of bound changes that results in a pruned node can be used for the diving set in an offshoot.

One slight modification to the method is to apply several bound changes at once before solving an LP relaxation. We call this *plunging*. This can go as far as fixing all integer variables since the result is guaranteed to be pruned and trimming can then be used to reduce the set of bound changes.

Another possibility is to use conflict analysis as described in [1] to obtain a clause. For offshoots that end in an infeasible node the dive set of bound changes is precisely a clause. Hence it is also possible to apply the method described by Karzan et al. in [8] to obtain a minimal clause using a MIPing approach.

### 4.3 Improved Pruning

One of the disadvantages of the new method is that pruning by bound after a new primal feasible solution has been found is complicated. Obviously, we can prune whole offshoots as soon as their top bound exceeds the new bound. But it can happen that for some offshoots the top bound is not large enough although applying some of the bound changes from the dive would result in an LP bound that would lead to a pruning.

This issue can be overcome partially by storing the objective values obtained during a dive. As long as no new offshoot is created from the dive (or new offshoots are created only from the bottom and in the order of the original dive) we can use the objective values to trim the dives after a new bound has been found. Since trimming also invalidates the bounds from the dive it is advisable to delay trimming until we first want to create an offshoot. This requires slightly more memory since more bound changes and dual information might have to be stored, but it could result in significantly better performance.

### 4.4 Bounding Offshoots

When branching from the top, the top bound of an offshoot remains a valid bound on all the nodes below this offshoot. But since we add a bound change to the top of the offshoot the bound is obviously not as strong as it could be. Hence it might be worthwhile updating the top bound after branching from the top.

We propose three methods of increasing computational effort to strengthen the bound. The first method is to derive a bound on the top node of the parent offshoot by using the reduced cost of the just-solved top node of the new offshoot.

The second method is a bit more general but also requires more computational effort. It involves simply evaluating the dual solution of the new offshoot's top node for the bounds of the parent offshoot.

The third method is to solve the LP for the new top node of the parent offshoot. Since we have a warmstart basis from the top node of the new offshoot this can be done using a very good warmstart basis.

Obviously, the first two methods provide only a lower bound on the new optimal objective value.

## 4.5 Shortening Dives

It is possible to implement the new method in a way that traditional branch-and-bound is just a special case. To this end we only need to ensure that in addition to storing open offshoots we can also store open nodes. This can be achieved, for example, by treating offshoots without a dive as normal nodes, which means when we select them we do not select an offshoot variable. Instead we treat it as the top node of a new offshoot directly. With this in place we can also have a limit on the number of bound changes in a dive. When the limit is hit, we store the last node as an open node in addition to storing the offshoot. The offshoot in this case does not end in a pruned node, but the method works regardless. If we set the limit of bound changes in a dive to zero, the method reverts to a traditional branch-and-bound method.

## 4.6 Splitting offshoots

Branching from the top on a bound change that was not the initial bound change of an offshoot invalidates all the internal objective values of the original nodes of an offshoot. This prevents us from pruning them and hurts the performance for very deep dives. Therefore it seems advantageous to split very long dives. However, this creates a non-terminal offshoot, so it extends our depth-first framework a little. For maximum efficiency, we need to resolve the linear relaxation of the bottom offshoot, so that we can have a valid objective value useful for pruning by bound. Which bound changes should be in the top or bottom half of the split is an interesting research question.

# 5 Computational Evaluation

The new method was prototyped using the MILP solver in SAS/OR. The prototype was meant as a way to evaluate the correctness and practicability of the method described and as such does not contain all the features and tricks of a full MILP solver. Nevertheless we present results using this prototype to give an impression of the capabilities of the new method.

The prototype plugs into the MILP solver after its root node when the actual branch-and-bound phase begins. It features a standard reliability branching strategy with a dynamic strong branching limit and a reliability limit of 5. For selecting the next offshoot we choose the best top bound first without explicit tie breaking. The prototype also features basic node presolve and reduced cost fixing techniques (only at the top of an offshoot), and also using the root reduced cost to fix columns globally as new incumbent solutions are found. What it notably lacks are more advanced node presolver techniques, local or global cuts in the nodes of the branch-and-bound tree, and primal heuristics.

We conduct our experiments on 96 machines running 2 jobs each on 16-core/2-socket Intel® Xeon® E5-2630 v3 @ 2.40GHz CPUs. All experiments are done with default settings, a memory limit of 62 GB, and a time limit of 2 hours. We use 798 instances that are the internal test set used to develop

the SAS MILP solver. To evaluate our results we use performance profiles as described in [5].

## 5.1 Offshoot variable selection

The first experiment evaluates several methods we implemented for choosing the offshoot variable, i.e., it is meant to judge the importance of shaping in the new method. We implemented four different methods:

**bottom:** always branch from the bottom of an offshoot without changing the order of the variables;

**top:** always branch from the top of an offshoot without changing the order of the variables;

**pseudo:** choose the offshoot variable with the best reliable pseudocost score and branch from the top. If there are no offshoot variables with reliable pseudocost available, then choose the variable with the worst pseudocost score and branch from the bottom.

**pseudodual:** like **pseudo**, except if there are no variables with reliable pseudocost then use the variable with the worst dual information score and branch from the bottom. The dual information score is the reduced cost or the Farkas certificate of the pruned node when the offshoot was first processed.

The first two strategies do not shape the tree so they can be seen as a baseline for the performance of the method. The default method is **pseudodual**.

Figure 6 shows the performance profile comparing the different offshoot variable selection strategies. In addition we would like to mention that the **pseudodual** strategy is about 13% faster in the geometric mean of the solve times than the **bottom** strategy and solves 9 instances more within the time limit. We argue that this shows that shaping, at least in the context of this new method, has a clear impact on the performance. More advanced selection strategies can probably be developed that will demonstrate this even more profoundly.

## 5.2 Trimming and pruning

Our second experiment is designed to show the combined importance of trimming and pruning. In our prototype implementation we delay trimming an offshoot until we need to choose an offshoot variable for the first time. Since we can either prune the bottom of the offshoot using the current cutoff or apply trimming using the dual information, we analyze how many reductions we get from either and choose the method that yields the most. In this experiment we compare the default version of our prototype that does this delayed pruning or trimming with a version where this feature has been disabled. The performance profile can be seen in Figure 7. The version with trimming and pruning is about 5% faster in the geometric mean of solve times and solves 1 instance

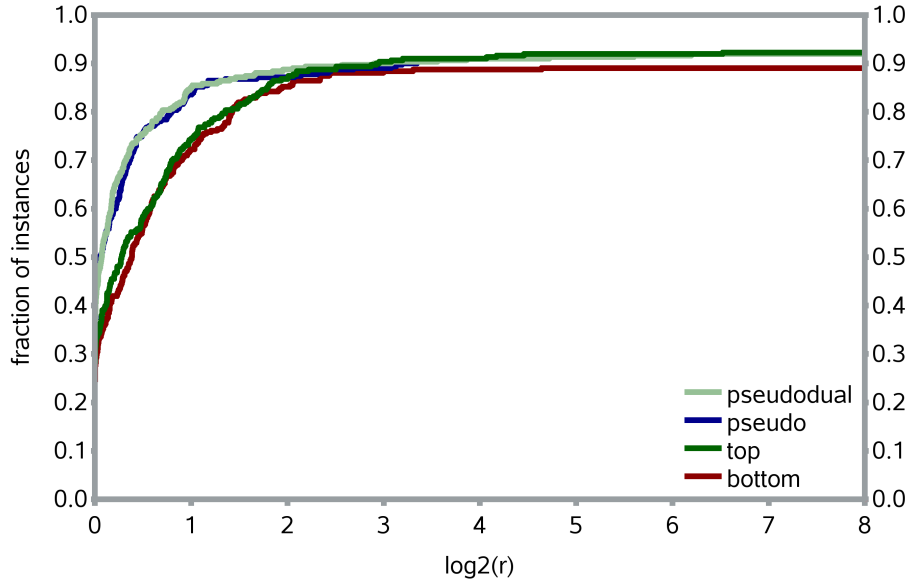


Figure 6: Performance profile comparing offshoot variable selection strategies.

fewer within the time limit. Since this effect seems to be rather small we think that it would be necessary to look into better ways to trim dives. Some ideas are described in Section 4.2.

### 5.3 Comparison against branch-and-bound

In our final experiment we compare our default method that does not limit the depths of the dives to a version where the limit is 0. This means that the method with the limit is essentially a traditional branch-and-bound method. The comparison is not completely fair since a pure branch-and-bound method could be implemented more efficiently, especially regarding memory requirements. But it gives a first impression of how much could be gained by using our new method instead of a traditional branch-and-bound method. Figure 8 shows the performance profile. Our new method is 38% faster in the geometric mean of the solve times and solves 47 instances more within the time limit. We consider this an encouraging result.

## 6 Conclusions

It will obviously take more research and a more elaborate implementation to see if our new method is superior to a traditional branch-and-bound method. From a theoretical perspective and from our preliminary experiments it seems likely

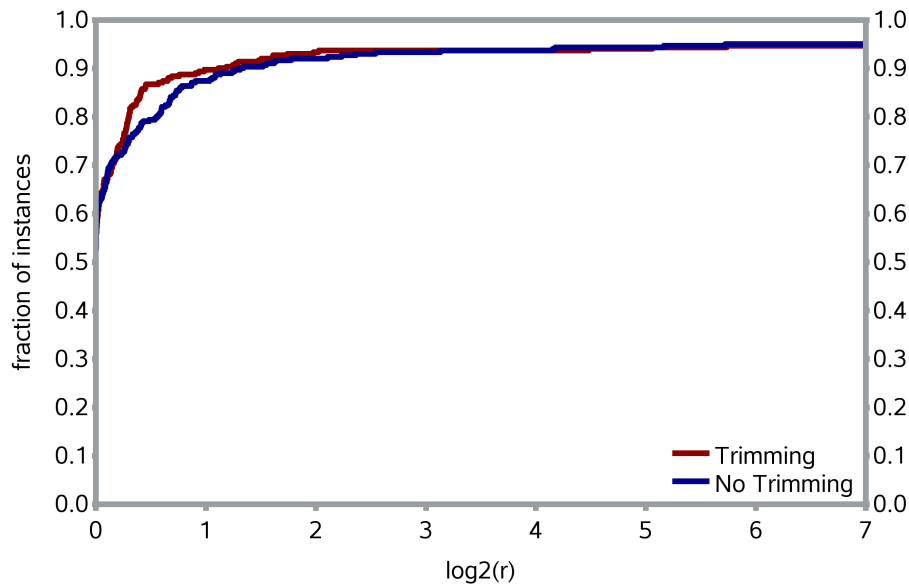


Figure 7: Performance profile comparing a version of the prototype that does pruning and trimming with a version that does not.

that shaping and trimming the tree will result in improved run times. Even if the performance gains end up being very small there is also hope that our new method will result in a more stable performance.

So far we have not investigated other areas of application for our new method such as mixed integer non-linear optimization problems or branch-and-price algorithms. Since in these areas more flexibility in the tree might be even more advantageous we hope that it will find application there as well.

## References

- [1] T. Achterberg, T. Koch, and A. Martin, *Branching rules revisited*, Operations Research Letters **33** (2005), no. 1, 42–54.
- [2] J. W. Chinneck, *Feasibility and infeasibility in optimization: Algorithms and computational methods*, International Series in Operations Research and Management Sciences, vol. 118, 2008.
- [3] V. Chvátal, *Resolution search*, Discrete Applied Mathematics **73** (1997), no. 1, 81 – 99.
- [4] W. Cook, *Markowitz and Manne+ Eastman+ Land and Doig= branch and bound*, Optimization Stories (2012), 227–238.

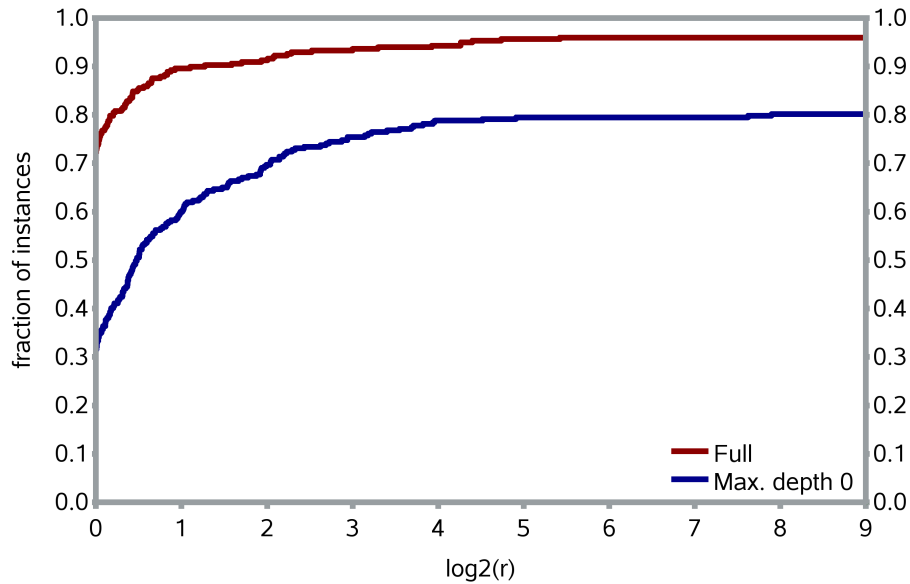


Figure 8: Performance profile comparing the default version with version with a maximum depth of 0, i.e., that resembles a traditional branch-and-bound.

- [5] E. Dolan and J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming Series A **91** (2002), 201–213.
- [6] F. Glover and L. Tangedahl, *Dynamic strategies for branch-and-bound*, OMEGA - The International Journal of Management Science **4** (1976), no. 5, 571–576.
- [7] S. Hanafi and F. Glover, *Resolution search and dynamic branch-and-bound*, Journal of Combinatorial Optimization **6** (2002), no. 4, 401–423.
- [8] F. Kılınç Karzan, G. Nemhauser, and M. Savelsbergh, *Information-based branching schemes for binary linear mixed integer problems*, Mathematical Programming Computation **1** (2009), no. 4, 249–293.
- [9] A. H. Land and A. G. Doig, *An automatic method of solving discrete programming problems*, Econometrica **28** (1960), no. 3, 497–520.