Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

BENJAMIN HILLER TORSTEN KLUG
JAKOB WITZIG

Reoptimization in branch-and-bound algorithms with an application to elevator control

ZIB-Report 13-15 (March 2013)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Reoptimization in branch-and-bound algorithms with an application to elevator control

Benjamin Hiller, Torsten Klug, Jakob Witzig
Zuse Institute Berlin
Takustraße 7
D–14195 Berlin, Germany.
hiller,klug,witzig@zib.de

March 21, 2013

Abstract

We consider reoptimization (i.e., the solution of a problem based on information available from solving a similar problem) for branch-and-bound algorithms and propose a generic framework to construct a reoptimizing branch-and-bound algorithm. We apply this to an elevator scheduling algorithm solving similar subproblems to generate columns using branch-and-bound. Our results indicate that reoptimization techniques can substantially reduce the running times of the overall algorithm.

1 Introduction

Many powerful solution methods for hard optimization problems, e.g., Lagrangian relaxation and column generation, are based on decomposing a problem into a master problem and one or more subproblems. The subproblems are then repeatedly solved to update the master problem that will eventually be solved to optimality. Usually, the subproblems solved in successive rounds are rather similar; typically, only the cost vector changes reflecting updated information from the master problem (i.e., Lagrangian multipliers in the case of Lagrangian relaxation approaches and dual prices in column generation methods). It is obvious that this similarity in subproblems should be exploited by "warmstarting" the solving process of a subproblem using information from the last round in order to reduce the running time. Methods to achieve this are known as *reoptimization techniques* and have been investigated in the context of decomposition methods for some time now, see e.g., [12] as an example for reoptimization in the context of Lagrangian methods and [2] as an example for column generation methods.

There is much literature on reoptimization of polynomially solvable optimization problems like the (standard) shortest path problems, see e.g. [13] and the references therein. However, for theoretical reasons the decomposition is usually done such that the resulting subproblems are NP-hard, though solvable effectively in practice. Recently, there is also growing theoretical interest in reoptimization methods focusing on the case that only the optimal solution from the last problem is known, see e.g. [1] for a survey.

Usually, specialized combinatorial algorithms are used to solve the subproblems, but there are also cases in which the subproblems may be solved using branch-and-bound algorithms, see e.g., [11, 4, 10]. Moreover, there is a recent interest to automatically reformulate mixed-integer programs (MIPs) and then apply decomposition techniques [6, 7, 8, 5, 3]. The subproblems in these reformulated models are solved by standard MIP solvers, which are very sophisticated branch-and-bound algorithms. Thus there is a need for reoptimization techniques for branch-and-bound-type algorithms.

In this paper, we propose a generic way to implement a reoptimizing branch-and-bound algorithm using the typical ingredients of a branch-and-bound algorithm. The essential idea is to "continue" the branch-and-bound search; in order to do that correctly, we do not only have to keep the search frontier, but also the set of pruned nodes and all solutions found so far. The intuition is that once the cost vector has converged to some extent, the branch-and-bound trees generated in successive rounds are basically the same in the higher levels. Thus the effort for creating this part again may be saved by reoptimization. In contrast to e.g., [2], our approach does not require any assumption on the structure of the change of the cost function, although we propose a way to exploit a special common structure. The details of our approach are presented in Section 2.

The main part of the paper is devoted to an application to elevator control, where reoptimization allows to substantially improve the running times of the column generation algorithm ExactReplan presented in [9, 10]. In addition to the extensions suggested by the generic reoptimization scheme, we also adjust the branching rule to facilitate reoptimization. This is necessary to take advantage of additional pruning possibilities that arise from properties of the lower bound used in ExactReplan.

2 Construction of a reoptimizing branch-and-bound algorithm

To formally introduce our concept of a reoptimizing branch-and-bound algorithm, we consider the following abstract setting. The aim is to solve the combinatorial optimization problem

$$\min\{c(x) \mid x \in \mathcal{S}\}, \tag{1}$$

given by a finite set of feasible solutions $\mathcal{S}$ and a cost function $c\colon \mathcal{S} \to \mathbb{R}$, successively for a sequence of cost functions $(c_i\colon \mathcal{S} \to \mathbb{R})_{i\in I}$, $I \subseteq \mathbb{N}$. For the applications we have in mind, c_{i+1} depends on the solutions obtained for c_i, but this is not used in the following.

Consider a branch-and-bound algorithm $\mathcal{A}$ that solves (1). We could just invoke $\mathcal{A}$ as-is once for every c_i to solve the sequence of optimization problems. We now design a branch-and-bound algorithm $\mathcal{A}'$ that allows to benefit from computations done for c_i when solving (1) with cost function c_{i+1}. To do that, we denote by v a node in $\mathcal{A}$'s search tree corresponding to a subproblem of (1) and think of $\mathcal{A}$ as being specified by the following subalgorithms:

$\mathsf{lb}(v)$ A function to compute a lower bound for node v in the search tree.

$\mathsf{branch}(v)$ A branching rule that partitions the search space corresponding to v into smaller regions, creating nodes $v_1, \ldots, v_k$, $k \geq 2$, of the search tree.

$\mathsf{heu}(v)$ A function to compute a heuristic solution for node v in the search tree; this function may also fail in the sense that no solution is produced. If, however, v corresponds to a single solution x (i.e., v is a leaf in the search tree), $\mathsf{heu}(v)$ returns x.

$\mathsf{feas}(v)$ A function to check whether the search tree rooted at node v still contains feasible solutions.

Using these subalgorithms, $\mathcal{A}$ basically divides the search space $\mathcal{S}$ into

- the set $\Sigma \subseteq \mathcal{S}$ of feasible solutions found so far,
- a set of subproblems $\mathcal{P}_f$ that have been pruned since the corresponding regions do not contain elements of $\mathcal{S}$ (i.e., for $v \subseteq \mathcal{S}$ $\mathsf{feas}(v)$ is false),
- a set of subproblems $\mathcal{P}_c$ that have been pruned since the corresponding regions do not contain elements with cost less than some upper bound U (i.e., for $v \subseteq \mathcal{S}$ we have $\mathsf{lb}(v) > U$),
- and a set $\mathcal{O}$ of as-yet unexplored subproblems,

such that we have $\mathcal{S} \subseteq \Sigma \cup \mathcal{S}(\mathcal{P}_c) \cup \mathcal{S}(\mathcal{O})$, where $\mathcal{S}(V) \subseteq \mathcal{S}$ denotes the solutions represented by the search tree nodes V.

As an example, consider the well-known LP-based branch-and-bound algorithm to solve mixed-integer programs. In this case, $\mathsf{lb}(v)$ is just the value of the LP relaxation at node v and $\mathsf{branch}(v)$ is a (possibly quite sophisticated) branching rule that selects a fractional variable and creates two child nodes with integer bounds for that variable excluding its current fractional value. $\mathsf{heu}(v)$ corresponds to the set of heuristics applied at v, including the trivial heuristic that returns the current LP solution if it is integer. Finally, $\mathsf{feas}(v)$ is the check whether the LP relaxation at node v is still feasible.

Assuming that $(\Sigma, \mathcal{P}_f, \mathcal{P}_c, \mathcal{O})$ are the corresponding sets after running $\mathcal{A}$ for cost function c_i, we may observe the following:

- The nodes in $\mathcal{P}_f$ do not have to be considered for cost function c_{i+1}.
- An optimal solution for cost function c_{i+1} is contained in $\Sigma \cup \mathcal{S}(\mathcal{P}_c) \cup \mathcal{S}(\mathcal{O})$.
- Any solution $x \in \Sigma$ may be optimal for c_{i+1}.
- A node $v \in \mathcal{P}_c$ might be attractive for cost function c_{i+1}, i.e., $\mathsf{lb}(v) \leq U$ for some given upper bound U.
- A node $v \in \mathcal{O}$ might be pruned due to cost for cost function c_{i+1}, i.e., $\mathsf{lb}(v) > U$.

Based on these observations it is straightforward to construct a branch-and-bound algorithm $\mathcal{A}'$ that uses the subalgorithms of $\mathcal{A}$ and exploits the computations done in the last round. To this end, $\mathcal{A}'$ maintains the sets Σ and $\mathcal{P}_c$ in addition to $\mathcal{O}$[1] for use in the next round and initializes them properly based on the sets from the last round. A pseudo-code for this reoptimizing branch-and-bound algorithm is shown in Figure 1. This version computes the set Σ^*

[1] $\mathcal{O}$ may not be empty in case the branch-and-bound search is stopped early, which is useful when applied to subproblems as a part of a decomposition scheme.

Input: cost function $c\colon \mathcal{S} \to \mathbb{R}$; upper bound U; sets Σ', $\mathcal{P}'_c$, $\mathcal{O}'$
Output: sets Σ, $\mathcal{P}_c$, $\mathcal{O}$; set of solutions Σ^* costing at most U

1: $\Sigma \leftarrow \Sigma'$, $\mathcal{P}_c \leftarrow \emptyset$, $\mathcal{O} \leftarrow \emptyset$, $\Sigma^* \leftarrow \emptyset$ ▷ Initialization
2: **for all** $x \in \Sigma$ **do**
3: Put x in Σ^* if $c(x) \leq U$.
4: **for all** $v \in \mathcal{P}'_c \cup \mathcal{O}'$ **do**
5: Put v in $\mathcal{O}$ if $\mathsf{lb}(v) \leq U$ and in $\mathcal{P}_c$ otherwise.
6: **while** $\mathcal{O} \neq \emptyset$ **do** ▷ Standard branch-and-bound
7: Choose $v \in \mathcal{O}$.
8: **if** $\mathsf{heu}(v)$ is successful and returns solution x **then**
9: Put x in Σ.
10: Put x in Σ^* if $c(x) \leq U$.
11: $v_1, \ldots, v_k \leftarrow \mathsf{branch}(v)$
12: **for** $i = 1, \ldots, k$ **do**
13: **if** $\mathsf{feas}(v)$ **then**
14: Put v in $\mathcal{O}$ if $\mathsf{lb}(v) \leq U$ and in $\mathcal{P}_c$ otherwise.

Figure 1: Pseudocode for reoptimizing branch-and-bound algorithm $\mathcal{A}'$.

of all feasible solutions with cost at most that of a given upper bound U. This formulation of the optimization is due to our application, where (1) corresponds to a pricing problem in a column generation context and U is some small negative constant, i.e., we look for any columns with negative reduced cost. We may as well consider all truely optimal solutions only.

From the preceding discussion and the logic of standard branch-and-bound, we have the following result.

Theorem 1 *Assuming that $\mathcal{S} = \Sigma' \cup \mathcal{S}(\mathcal{P}'_c) \cup \mathcal{S}(\mathcal{O}')$, algorithm $\mathcal{A}'$ defined in Figure 1 correctly computes the set Σ^* and upon termination we have $\mathcal{S} = \Sigma \cup \mathcal{S}(\mathcal{P}_c) \cup \mathcal{S}(\mathcal{O})$.*

Proof Consider a solution $x \in \mathcal{S}$. In the case that x is in Σ', by Step 1 it will be in Σ, too. Moreover, it will also be in Σ^* iff its cost are at most U. If x is represented by a search tree node $v' \in \mathcal{P}'_c \cup \mathcal{O}'$, i.e., $x \in \mathcal{S}(\mathcal{P}'_c) \cup \mathcal{S}(\mathcal{O}') = \mathcal{S}(\mathcal{P}'_c \cup \mathcal{O}')$, after Step 4 node v' is either in $\mathcal{O}$ (if $\mathsf{lb}(v') \leq U$) or in $\mathcal{P}_c$ (if $c(x) \geq \mathsf{lb}(v') > U$).

Assume now that $v' \in \mathcal{O}$ after the initialization phase. The remaining steps of $\mathcal{A}'$ maintain the invariant $x \in \Sigma \cup \mathcal{S}(\mathcal{P}_c \cup \mathcal{O})$. To see this, let v be the node representing x at the beginning of the while loop. In case x is found by $\mathsf{heu}(v)$, it is put in Σ (and in Σ^* if necessary). Otherwise, it will be represented by at least one of the nodes $v_1, \ldots, v_k$ created by $\mathsf{branch}(v)$, say v_1. By definition, $\mathsf{feas}(v_1)$ is true, so v_1 is either put in $\mathcal{O}$ or $\mathcal{P}_c$. Moreover, v_1 is only put in $\mathcal{P}_c$ if $c(x) \geq \mathsf{lb}(v_1) > U$. Thus in case $c(x) \leq U$ it will eventually be found by $\mathsf{heu}()$, and thus be contained in Σ^*.

As for the running time, initializing Σ^* in Step 2 takes $|\Sigma'|$ evaluations of c, which is usually cheap. To initialize $\mathcal{O}$ and $\mathcal{P}_c$ in Step 4 requires $|\mathcal{P}'_c \cup \mathcal{O}|$ evaluations of $\mathsf{lb}()$, which may be rather expensive. It is, however, possible to avoid the recomputation of the lower bounds if the cost functions c_i and c_{i+1}

and the lower bound method exhibit special structure. Assume that the cost functions c_i and c_{i+1} have the same (separable) structure of the form

$$c(x) = c_0(x) + \sum_{j=1}^{m} c_j(x)\pi_j, \tag{2}$$

where only the coefficients π_j change from i to $i+1$. Denote by $c(v)$ the minimum cost of a solution represented by node v. If $\mathsf{lb}(v)$ actually provides (additionally) lower bounds $\underline{c}_j(v)$ for $c_j(x)$, $0 \leq j \leq m$, for any $x \in \mathcal{S}$ represented by v, we can compute a lower bound for v as

$$c(v) \geq \underline{c}_0(v) + \sum_{j=1}^{m} \underline{c}_j(v)\pi_j, \tag{3}$$

which takes time $O(m)$ for each node v if $\underline{c}_j(v)$, $0 \leq j \leq m$, are stored with v. A cost structure like (2) arises in the contexts of column generation and Lagrangian relaxation, where the π_j are dual prices or Lagrangian multipliers, respectively.

3 Elevator control as an application of a reoptimizing branch-and-bound algorithm

We now apply our framework for reoptimizing branch-and-bound algorithms to the column-generation-based elevator scheduling algorithm ExactReplan from [9, 10]. The ExactReplan algorithm is designed to schedule elevators in destination call systems, where a passenger registers his destination floor upon his arrival at the start floor. Let $\mathcal{E}$ be the set of elevators. A (destination) *call* is a triple of the release time, the start floor and the destination floor corresponding to this registration. Note that the elevator control knows only calls, not about passengers. At any point in time we can build a *snapshot problem* describing the current system state. ExactReplan determines an optimal solution for each snapshot problem, giving the schedule to follow until new information becomes available. In a snapshot problem, the calls are grouped to *requests* according to certain rules reflecting the communication between the passenger and the elevator control. A request has a start floor and a set of destinations floors. We distinguish between *assigned requests* $\mathcal{R}(e)$ for each elevator, for which it has already been decided that elevator e is going to serve them, and *unassigned requests* $\mathcal{R}_u$, which still may be assigned to any elevator. In fact, determining the elevator serving each request $\rho \in \mathcal{R}_u$ is the main task of an elevator control algorithm. Solving a snapshot problem requires to schedule the elevators such that each request is *served*, i.e., there is an elevator traveling to the corresponding start floor (to pick up the calls/passengers) and visiting its destination floors (to drop the calls/passengers) afterwards. In particular, a feasible schedule for elevator e needs to serve all assigned requests $\mathcal{R}(e)$ and may serve any subset of the unassigned requests $\mathcal{R}_u$. We call a selection of feasible schedules, one for each elevator, that together serve all requests, a *dispatch*.

3.1 The original ExactReplan algorithm

Let $\mathcal{S}(e)$ be the set of all feasible schedules for elevator e and define $\mathcal{S} := \bigcup_{e \in \mathcal{E}} \mathcal{S}(e)$. For each $S \in \mathcal{S}$ we introduce a decision variable $x_S \in \{0, 1\}$ for

including a schedule in the current dispatch or not. Denoting by $c(S)$ the cost of schedule S, the following set partitioning model describes the problem:

$$\min \sum_{S \in \mathcal{S}} c(S) x_S \tag{4}$$

$$\text{s.t.} \sum_{S \in \mathcal{S} : \rho \in S} x_S = 1 \quad \forall \rho \in \mathcal{R}_u \tag{5}$$

$$\sum_{S \in \mathcal{S}(e)} x_S = 1 \quad \forall e \in \mathcal{E} \tag{6}$$

$$x_S \in \{0,1\} \quad \forall S \in \mathcal{S} \tag{7}$$

Equations (5) and (6) ensure that each request is served by exactly one elevator and each elevator has exactly one schedule, respectively. Note that the model only decides assignment for the unassigned requests and the assigned requests are treated implicitly by the sets $\mathcal{S}(e)$. The number of variables of this Integer Programming (IP) problem is very large, because each permutation serving a request subset $R \subseteq \mathcal{R}_u$ corresponds to a feasible schedule. We therefore use column generation to solve the LP relaxation of the model above using a branch-and-bound algorithm to solve the following pricing problem.

For all requests $\rho \in \mathcal{R}$ and $e \in \mathcal{E}$ we denote the dual prices associated with constraints (5) and (6) by π_ρ and π_e, respectively. Moreover, let $\mathcal{R}_u(S)$ be the unassigned requests served by schedule S. For each elevator e we have to find $S \in \mathcal{S}(e)$ with negative reduced cost

$$\tilde{c}(S) := c(S) - \sum_{\rho \in \mathcal{R}_u(S)} \pi_\rho - \pi_e \tag{8}$$

or to decide that no such schedule exists. The cost of S is the sum of the cost $c(\rho)$ for serving each request ρ, i.e.,

$$c(S) = \sum_{\rho \in \mathcal{R}(e) \cup \mathcal{R}_u(S)} c(\rho). \tag{9}$$

Pricing via branch-and-bound

A schedule S is a sequence of stops $(s_0, \ldots, s_k)$ describing future visits to floors. We enumerate all feasible schedules for elevator e by constructing a schedule stop by stop, branching if there is more than one possibility for the next stop. Thus each search tree node v corresponds to a feasible schedule S_v and one of its stops s_v; the schedule up to s_v is fixed and the later stops correspond to dropping off passengers. Moreover, we maintain for each v the set $A_v \subseteq \mathcal{R}(e)$ of not yet picked up assigned requests and the set $O_v \subseteq \mathcal{R}_u$ of not yet picked up optional requests. At v there are the following branching possibilities: Either the next stop is the one following s_v (if there is one) or the next stop is at a starting floor of a request in $A_v \cup O_v$, which is then picked up there. We create a child node for any of these possibilities.

Our branch-and-bound pricing algorithm computes for each node v a lower bound of the reduced costs by

$$\underline{\tilde{c}}(v) = c(S_v) + \sum_{\rho \in A_v} \underline{c}(\rho) + \sum_{\rho \in O_v \,:\, \underline{c}(\rho) - \pi_\rho < 0} (\underline{c}(\rho) - \pi_\rho) \quad - \pi_e, \tag{10}$$

$\mathcal{T}_i$	1	2	3	4	5	6	7
1	1.0000	0.2581	0.1048	0.1044	0.1069	0.1030	0.1039
2		1.0000	0.2591	0.2637	0.2658	0.2604	0.2626
3			1.0000	0.5706	0.5105	0.5537	0.5629
4				1.0000	0.8484	0.9098	0.9286
5					1.0000	0.9187	0.8965
6						1.0000	0.9778
7							1.0000

Table 1: Example for similarity of rooted trees. Entry $(i, j), i \geq j$, represents the similarity between the rooted search tree $\mathcal{T}_i$ at the end of pricing round i and the rooted search tree $\mathcal{T}_j$ at the end of pricing round j. For instance, 91.87 % of all nodes from the rooted search trees $\mathcal{T}_5$ and $\mathcal{T}_6$ are part of both trees.

where $\underline{c}(\rho)$ is a lower bound on the primal-costs of requests ρ. An important observation is that we can prune all optional requests with $\underline{c}(\rho) - \pi_\rho \geq 0$, leading to a much smaller search tree.

Proposition 1 *Consider a node v of the search tree corresponding to an elevator e and dual prices $(\pi_\rho)_{\rho \in O_v}$. If the search tree rooted at v contains a schedule with negative reduced cost, then it also contains one with negative reduced cost that does not serve the requests in $O_v^{\geq} := \{\rho \in O_v \mid \underline{c}(\rho) - \pi_\rho \geq 0\}$.*

An important feature of our pricing algorithm is that we do not solve the pricing problem to optimality, but stop as soon as k schedules with negative reduced cost are found. These schedules are then added to the set partitioning master problem, whose LP relaxation is then resolved to obtain new dual prices. The rationale for this is to avoid to spend too much time due to bad dual prices.

Similarity of search trees

In our computations we observed that the sets of generated nodes in successively generated search trees get more and more similar. To quantify this, we use the following similarity measure for rooted trees [14]. We denote by $\mathfrak{T}$ the set of all rooted trees and define for two rooted trees $\mathcal{T}, \mathcal{T}' \in \mathfrak{T}$ the number

$$\alpha(\mathcal{T}, \mathcal{T}') := |\{v \in \mathcal{T} \mid \text{the unique } (r, v)\text{-path in } \mathcal{T} \text{ is contained in } \mathcal{T}'|\},$$

where r is the root of $\mathcal{T}$. The similarity $\Lambda(\mathcal{T}, \mathcal{T}') \in [0, 1]$ between $\mathcal{T}$ and $\mathcal{T}'$ is then given by

$$\Lambda : \mathfrak{T} \times \mathfrak{T} \to [0, 1],\ (\mathcal{T}, \mathcal{T}') \mapsto \begin{cases} \frac{\alpha(\mathcal{T}, \mathcal{T}')}{|V| + |V'| - \alpha(\mathcal{T}, \mathcal{T}')}, & V \neq \emptyset,\ V' \neq \emptyset \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

An example of the evolution of this similarity measure from pricing round to pricing round is shown in Table 1.

3.2 The reoptimizing ExactReplan algorithm

In Section 2 we presented a straightforward way to transform a standard branch-and-bound algorithm to a reoptimizing one. Now we aim to apply this scheme to

the ExactReplan algorithm. Recall that an upper bound is given by U, the set of solutions found so far is denoted by Σ, $\mathcal{P}_c$ is the set of nodes v that have been pruned since $\tilde{\underline{c}}(v) > U$ and finally, $\mathcal{O}$ is the the set of as-yet unexplored nodes. The implementation of these basic structures and the initialization procedure in Algorithm 1 is straightforward.

We already mentioned that, assuming that the cost function has the structure (2) and the lower bound method "is compatible" with this structure, the updates of the lower bounds for all nodes $v \in \mathcal{P}_c \cup \mathcal{O}$ can be done in time $O(m)$. Observe that the schedule cost function (9) is exactly of type (2) and also the lower bound (10) matches this structure. We can thus use Formula (3) to update the lower bounds for each node v, which only requires storing $\underline{c}_0(v) := c(S_v) + \sum_{\rho \in A_v} \underline{c}(\rho)$ and $\underline{c}(\rho)$ for $\rho \in O_v$ with v. Our computational experiments show [14] that using this fast update of the lower bounds reduces the time spent in the initialization phase by 60–85 %.

A disadvantage of the straightforward reoptimizing branch-and-bound algorithm is that we cannot exploit Proposition 1: It might happen that a request ρ with $\underline{c}(\rho) - \pi_\rho \geq 0$ at iteration i will have $\underline{c}(\rho) - \pi_\rho < 0$ at iteration $j > i$, which we would not detect if we just remove ρ from O_v in iteration i. An immediate consequence is an unnecessarily high number of generated nodes in the reoptimizing branch-and-bound algorithm. To avoid that, we use a different branching procedure when reoptimizing that records pruning due to Proposition 1 explicitly. It is thus equivalent to the original one in the sense that it generates the same search tree when used without reoptimization. To describe this, we introduce the following notation.

- The set of all floors which are branching possibilities at node v is denoted by $\mathcal{B}(v)$.
- $O_v^< := \{\rho \in O_v \mid \text{the start floor of } \rho \text{ is in } \mathcal{B}(v) \text{ and } \underline{c}(\rho) - \pi_\rho < 0\} \subseteq O_v$
- $O_v^\geq := \{\rho \in O_v \mid \text{the start floor of } \rho \text{ is in } \mathcal{B}(v) \text{ and } \underline{c}(\rho) - \pi_\rho \geq 0\} \subseteq O_v$
- A node v is called *branched*, if $O_v^\geq = O_v^< = \emptyset$.
- A node v is called *pseudo-branched*, if $O_v^\geq \neq \emptyset$ and $O_v^< = \emptyset$.

In each branching step we branch only on the start floors corresponding to optional requests in $O_v^<$. If $O_v^\geq = \emptyset$, v is branched and can be deleted as in the non-reoptimizing branch-and-bound algorithm. Otherwise we store v in $\mathcal{P}_c$. Additionally, we extend the initialization phase: If a node v from $\mathcal{P}_c$ is not yet branched, we compute the new sets $O_v^\geq$ and $O_v^<$ from the set $O_{\bar{v}}^\geq$ of the last iteration, creating child nodes for each start floor of an request in $O_v^<$. These child nodes are stored in $\mathcal{O}$ for further processing. Moreover, the requests in $O_v^<$ are removed from $O_{\bar{v}}^\geq$, recording the fact that the corresponding branches have been created. We call this modified branching method Pseudo-Branching.

The Pseudo-Branching technique has a positive side effect, namely a reduction of schedules which have to be stored, because we are generating fewer nodes. Thus the time needed for the initialization phases decreases, too. Our computational experiments show that on average, the number of generated nodes decreases by 60 %, the initialization time by 25 % and the number of stored schedules by 24 % due to Pseudo-Branching.

3.3 Computational results

In our simulations we consider two buildings and six traffic patterns with three different traffic intensities [10]. Building A has a population of 1400 people, 23 floors and 6 elevators; building B has a population of 3300 people, 12 floors and 8 elevators. The traffic patterns are standard for assessing elevator control algorithms and mimic traffic arising in a typical office building. In the morning, passengers enter the building from the ground floor, causing *up peak traffic*. Then there is some *interfloor traffic* where the passengers travel roughly evenly between the floors. During *lunch traffic*, people leave and reenter the building via the ground floor. Finally, there is *down peak traffic* when people leave the building in the afternoon. In addition, we also consider *real up peak traffic* and *real down peak traffic*, which mix the up peak and down peak traffic which 5% of interfloor and 5% of down peak and up peak traffic, respectively. These two patterns are supposed to model the real traffic conditions more closely than the pure ones. One hour of each traffic pattern is simulated for three different intensities: 80 %, 100 % and 144 % of the population arriving in one hour.

We compare the original ExactReplan algorithm to its reoptimizing version ExactReplan-reopt with fast updating of lower bounds and Pseudo-Branching. Both variants of the ExactReplan algorithm solve the LP relaxation of any snapshot problem in the root node to optimality and then solve the resulting IP to optimality without generating further columns. All computations ran under Linux on a system with an Intel Core 2 Extreme CPU X9650 with 3.0 GHz and 16 GB of RAM. We did not use the 64bit facility on this machine. Results are shown in Tables 2 and 3. The number of generated nodes is at least halved on the average. A comparison between the *total time* and the *initialization time* shows that the initialization of the reoptimizing branch-and-bound algorithm is cheap compared to the branching procedure. Moreover, the column *time ratio* shows that it is possible to save up to 85 % of computation time (Up Peak 144 % on building A) when using the reoptimizing variant. Since the branching rules are equivalent w. r. t. generated search trees without reoptimization, the speedup is entirely due to our reoptimization techniques.

4 Conclusion

We proposed a general scheme to use reoptimization in a branch-and-bound algorithm and applied this scheme to the elevator scheduling algorithm ExactReplan based on column generation. Moreover, we adjusted the branching rule of our reoptimizing version of ExactReplan to take advantage of additional pruning possibilities also when using reoptimization. This reoptimizing version of ExactReplan outperforms ExactReplan substanstially up to a factor of 6. As a next step, we want to employ reoptimization also to the branch-and-price version of ExactReplan, which also uses column generation to solve the LP relaxation of nodes below the root. Moreover, we will study reoptimization for LP-based branch-and-bound used in state-of-the-art MIP solvers to improve the performance of automatic decomposition frameworks like GCG [8].

traffic pattern		ExactReplan		ExactReplan-reopt			
name	#snapshots	∅ gen. nodes	total time	∅ gen. nodes	Σ init. time	total time	time ratio
Down Peak 80 %	963	39.57	68	**17.13**	2	**44**	0.642
Down Peak 100 %	1133	73.92	134	**32.45**	5	**86**	0.640
Down Peak 144 %	1544	391.18	824	**168.13**	64	**514**	0.624
Interfloor 80 %	980	71.14	117	**30.35**	5	**76**	0.650
Interfloor 100 %	1174	296.45	515	**129.78**	43	**328**	0.637
Interfloor 144 %	1571	10426.06	26621	**2772.96**	3110	**12124**	0.455
Lunch Peak 80 %	980	26.28	52	**11.33**	1	**36**	0.690
Lunch Peak 100 %	1172	75.47	146	**31.92**	6	**94**	0.641
Lunch Peak 144 %	1568	2774.93	6764	**748.98**	598	**2827**	0.418
Real Down Peak 80 %	979	27.77	55	**12.06**	1	**38**	0.689
Real Down Peak 100 %	1171	61.80	119	**27.75**	4	**79**	0.669
Real Down Peak 144 %	1565	337.59	738	**146.38**	61	**471**	0.639
Real Up Peak 80 %	964	125.26	203	**30.67**	6	**74**	0.364
Real Up Peak 100 %	1164	308.59	585	**73.65**	19	**199**	0.340
Real Up Peak 144 %	1714	3914.48	12690	**553.04**	392	**2401**	0.189
Up Peak 80 %	966	966 65.54	105	**16.91**	2	**45**	0.426
Up Peak 100 %	1159	1305.92	2702	**259.06**	70	**670**	0.248
Up Peak 144 %	1656	29064.20	106597	**3564.73**	2652	**16004**	0.150

Table 2: Building A, computational results. The second and fourth column shows the average of generated nodes per snapshot problem and per elevator. The total time in the third and sixth column is the sum over all snapshot problems. Analog the time needed for the initialization at the beginning of each pricing round (fifth column). All times are represented in seconds. The time ratio is the quotient of (total time ExactReplan-reopt)/(total time ExactReplan).

traffic pattern		ExactReplan		ExactReplan-reopt			
name	#snapshots	∅ gen. nodes	total time	∅ gen. nodes	Σ init. time	total time	time ratio
Down Peak 80 %	1830	19.11	105	**8.36**	1	**67**	0.641
Down Peak 100 %	2131	25.27	147	**10.97**	3	**92**	0.626
Down Peak 144 %	2871	42.22	293	**18.35**	8	**181**	0.618
Interfloor 80 %	1868	100.27	403	**48.58**	23	**286**	0.710
Interfloor 100 %	2182	246.01	1078	**136.23**	106	**887**	0.823
Interfloor 144 %	2752	4637.07	26507	**1681.62**	3432	**16225**	0.612
Lunch Peak 80 %	1844	48.40	217	**23.09**	7	**147**	0.680
Lunch Peak 100 %	2183	247.52	1087	**133.77**	105	**875**	0.805
Lunch Peak 144 %	2772	441.79	2417	**249.33**	267	**1991**	0.824
Real Down Peak 80 %	1868	21.29	116	**9.16**	2	**74**	0.642
Real Down Peak 100 %	2182	34.22	195	**14.77**	5	**125**	0.640
Real Down Peak 144 %	2809	81.17	499	**35.93**	20	**317**	0.637
Real Up Peak 80 %	1893	46.19	211	**14.86**	4	**110**	0.522
Real Up Peak 100 %	2226	89.98	447	**26.14**	11	**199**	0.445
Real Up Peak 144 %	11658	1218.45	37309	**257.86**	1180	**10539**	0.282
Up Peak 80 %	1829	46.68	201	**14.28**	3	**99**	0.495
Up Peak 100 %	2264	81.75	427	**22.01**	8	**171**	0.402
Up Peak 144 %	13268	614.23	27857	**121.03**	467	**6834**	0.245

Table 3: Building B, computational results. The second and fourth column shows the average of generated nodes per snapshot problem and per elevator. The total time in the third and sixth column is the sum over all snapshot problems. Analog the time needed for the initialization at the beginning of each pricing round (fifth column). All times are represented in seconds. The time ratio is the quotient of (total time ExactReplan-reopt)/(total time ExactReplan).

References

[1] G. Ausiello, V. Bonifaci, and B. Escoffier. *Computability in Context: Computation and Logic in the Real World*, chapter Complexity and Approximation in Reoptimization. Imperial College Press/World Scientific, 2011.

[2] M. Desrochers and F. Soumis. A reoptimization algorithm for the shortest path problem with time windows. *European J. Oper. Res.*, 35:242–254, 1988.

[3] DIP – Decomposition for Integer Programming. `https://projects.coin-or.org/Dip`.

[4] P. Friese and J. Rambau. Online-optimization of a multi-elevator transport system with reoptimization algorithms based on set-partitioning models. *Discrete Appl. Math.*, 154(13):1908–1931, 2006.

[5] M. Galati. *Decomposition in Integer Linear Programming.* PhD thesis, Lehigh University, 2009.

[6] G. Gamrath. Generic branch-cut-and-price. Diploma thesis, TU Berlin, 2010.

[7] G. Gamrath and M. E. Lübbecke. Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In *Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *LNCS*, pages 239–252. Springer, 2010.

[8] GCG – Generic Column Generation. `http://www.or.rwth-aachen.de/gcg/`.

[9] B. Hiller. *Online Optimization: Probabilistic Analysis and Algorithm Engineering.* PhD thesis, TU Berlin, 2009.

[10] B. Hiller, T. Klug, and A. Tuchscherer. An exact reoptimization algorithm for the scheduling of elevator groups. *Flexible Services and Manufacturing Journal*, to appear.

[11] S. O. Krumke, J. Rambau, and L. M. Torres. Realtime-dispatching of guided and unguided automobile service units with soft time windows. In *Proceedings of ESA 2002*, volume 2461 of *LNCS*, pages 637–648. Springer, 2002.

[12] L. Létocart, A. Nagih, and G. Plateau. Reoptimization in Lagrangian methods for the quadratic knapsack problem. *Comput. Oper. Res.*, 39(1):12–18, 2012.

[13] E. Miller-Hooks and B. Yang. Updating paths in time-varying networks with arc weight changes. *Transportation Sci.*, 39(4):451–464, 2005.

[14] J. Witzig. Effiziente Reoptimierung in Branch&Bound-Verfahren für die Steuerung von Aufzügen. Bachelor thesis, TU Berlin, 2013.