36th European Modeling & Simulation Symposium, 031
21st International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2024 The Authors.
doi: 10.46354/i3m.2024.emss.031

# Dynamic transport-lot assignment for the hot-storage area

Sebastian Leitner[1,2,*], Philipp Fleck[1,2], Stefan Wagner[1,2] and Michael Affenzeller[1]

[1]Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria

[2]Josef Ressel Center for Adaptive Optimization in Dynamic Environments, University of Applied Sciences Upper Austria

[*] Sebastian.Leitner@fh-hagenberg.at

## Abstract

In this paper, we present a novel transport-lot optimization problem at the boundary between the continuous casters and the hot-rolling in steel production. The problem and the corresponding solver are part of a system that jointly optimizes the cranes in the hot-storage area and the vehicles feeding the rolling mill. The goal is to group steel slabs into transport lots and assign transporters, handover locations, and due dates. We develop heuristics to iteratively build the full solution by choosing from several promising choices that satisfy a wide variety of safety and performance requirements. We evaluate the performance of different heuristics to search the tree of possible choices on a diverse collection of real-world problem instances. We find that the pilot method significantly outperforms the five other tree-search heuristics we tried.

**Keywords**: Transport-Lots, Hot-storage, Continuous Caster, Integrated Steel Production, Tree-Search

## 1. Introduction

From steel making to continuous casting to hot rolling, there are many interconnected and energy-intensive processes in a steel plant, that if well optimized, can lead not only to cost savings but also to a reduced environmental impact. Tang et al. (2001) gives a good overview of integrated production management systems deployed in steel plants to realize these benefits. For a good overview of optimization techniques used towards the hot rolling end of the entire steel-making process, we recommend Özgür et al. (2021).

This paper is concerned with the optimization of handling of newly cast steel slabs in the so-called hot storage area. The hot storage area is a very dynamic environment consisting of multiple continuous casters, buffer stacks, handover locations, overhead cranes capable of moving a single slab at a time between each of these locations, and vehicles to transport stacks of slabs to processing facilities or storage yards. New slabs are continuously produced in the casters according to a casting program and are finally transported from the handover locations to their next processing step by two types of vehicles. One challenge is that while there is a plan for what the next processing steps should be for each slab, there is no guarantee that the slab will meet all of the criteria for that plan to hold once it is cast. So there is considerable uncertainty about where each slab needs to go next and when. Transportation and storage capacity is limited inside and outside the hot storage area and can be traded against each other to some extent. So even if the next processing step is certain it may make sense to transport slabs to a storage yard if we run out of space in the hot storage area but have a transporter available. However, transport vehicles, are not used exclusively for transport from the hot storage area, so their availability is also uncertain. For a discussion of the various kinds of uncertainties in this environment, see Beham et al. (2019). Roljic et al. (2021) optimized an

integrated routing and stacking problem for slabs in the storage yard, which must also satisfy the same safety constraints. Priorities and bottlenecks can change quickly, and there is a need to respond to human feedback about where problems are occurring and which slabs are needed most urgently. One source of problems is the continuous casters themselves, which can break down and require rescheduling, as discussed by Long et al. (2016).

The slabs are around 700°C hot and weigh tens of tons when they come out of the caster, which means they must be handled carefully to avoid damaging equipment, other slabs, or workers. To achieve this, there are numerous restrictions on how slabs can be stacked and transported based on all of their physical properties. Even when all restrictions are followed, equipment failure can and does occur and must be handled gracefully. Temperature not only dictates where and how slabs can be transported, but also when. This is because most processing steps require a certain temperature, and slabs cool slowly. Complexity is further increased because the cooling rate depends primarily on the temperature of all the surrounding slabs in the stack. Avoiding the need to reheat the slabs before each processing step is essential to the energy efficiency of the overall steel plant.

The goal is to create a plan for the human crane operators to follow that takes all safety and performance considerations into account. This is challenging because of the complex interactions within the systems, the uncertainty, the dependence on human operators, and the large number of safety constraints that must be met. The rest of this paper describes the two-tiered solution approach in Section 2 and then focuses on the Hot-Storage Lot-Assignment Problem. Section 3 tests a variety of tree search variants on a set of 100 real-world problem instances, and Section 4 discusses the results.

## 2. Problem

All slabs currently in the hot storage are given, including their positions, physical properties, and next processing steps. We also include the next slabs to be cast according to the current casting schedule. We have information on every relevant location vehicle and crane, including stacking restrictions and availability. We have two basic goals which are to ensure that the hot storage area runs smoothly and that all downstream processing can operate efficiently. The first goal boils down to these four points:

1. **Never block casters**, as this can result in costly manual repairs.
2. **Adhere to constraints**, so as not to endanger workers, equipment, or the quality of the slabs.
3. **Use cranes efficiently**, by minimizing their travel time and avoiding relocations as much as possible.
4. **Do not overfill the hot storage**, as this makes it much more difficult to achieve all of the previous objectives.

To ensure good downstream performance, we need to perform the following steps:

1. **Select the right slabs to deliver** based on both downstream processing requirements and the fill level of the hot storage area.
2. **Prioritize deliveries well** to avoid downstream waiting times.
3. **Use vehicles efficiently** by maximizing capacity utilization and minimizing travel time.
4. **Group intelligently for later use** to minimize handling costs at the destination. Some destinations require slabs in a specific order, while others simply want to maximize throughput.

While it would be theoretically possible to define one large optimization problem, we found that we had to split the optimization into two parts to make it tractable. The first part is the transport lot assignment problem, where we decide which slabs will be transported together by a given vehicle, and when and where to hand them off to the vehicle. The output of the lot assignment is part to the input for the stacking problem. There, we optimize the crane moves required to serve both the incoming slabs from the caster and the outgoing slabs according to the lot assignment. Both algorithms are used together in a framework that maintains an up-to-date model of the hot-storage area as well as the current plans generated by our two solvers. It receives real-world events via a message-passing system and reacts to them by updating its models checking whether any plans are invalidated by what just happened, and re-optimizes if they are.

### 2.1. Hot-Storage Lot-Assignment Problem

The central entity in this problem is the transport lot. Each transport lot consists of a possibly ordered set of slabs, a transport vehicle, a handover location, a target location, and two time intervals. The first interval defines when we can use the crane to move the slabs to the handover location, and the second interval defines when we can use the vehicle to bring the slabs to their destination. When we create transport lots, we make sure that these intervals do not overlap between transport lots for any vehicle or section of the crane runway, otherwise, the solution would be infeasible.

The goal is to deliver all the slabs that need to be delivered as quickly as possible. We call this goal the total completion time, and it is calculated as

$$T = \sum_{x \in \text{lots}} \text{delivery}_x - \min_{s \in x} \text{handover}_s. \tag{1}$$

The earliest possible handover time of a lot depends on when the first slab can be put on the handover. This can be now, when the slab is at the top of a stack, the crane has nothing else to do, and the handover is ready; or it can be in the future, after the slab has been cast, the slabs above have been relocated and the handover became available.

The delivery date depends on when the last slab can be handed over and when the vehicle is available. So both the choice of slabs and vehicles and the order of delivery are important to the quality of the solution.

There are, of course, several restrictions on which slabs can be transported together and which transporter can be used. In the case of hot slabs, these restrictions even depend on the time when the transport should take place. A solution that violates safety constraints is considered infeasible. For all other constraints, we allow a configurable number of allowed violations before the solution is considered infeasible.

#### 2.1.1. *Problem Tree-Encoding*

To facilitate the search for the best lot assignments, we employ various tree search heuristics, which are described in more detail later in Sec. 3. For the tree search, the problem is defined in a tree structure, where each non-terminal node is a partial solution, and the terminal nodes are solutions to the lot assignment problem. Starting with the root node, which has a list of "unassigned slabs" and no *lots assignments*, each node lists potential choices of *lot-assignments*. Each lot assignment then defines the list of slabs for a lot, the target handover location, and the vehicle. Because lots are limited by various constraints, such as vehicle capacity, lot-assignments contain only a finite number of slabs. Therefore, for a given list of slabs, there are many possible lot-assignments in which different slabs are grouped together, or the same slabs are grouped at different handover locations.

When a lot assignment is applied to the current partial solution (i.e. a tree search node), a new (partial) solution (i.e., a new node) is created, adding the selected lot assignment to the list of lot assignments and removing the associated slabs from the list of unassigned slabs. This continues until all slabs are assigned to a lot and a node forms a terminal node.

The tree search itself is responsible for selecting one of the possible lot assignments for a given partial solution, and thus traversing through the search space. Efficient guidance relies on the order of choices, as the tree search assumes earlier choices are optimal. Therefore, when we create the list of potential lot assignments, we order them according to domain experts rules, which should favor good lot assignments that contribute to an overall efficient grouping of lots. Invalid or undesirable lot assignments are not even returned and are therefore not considered by the tree search algorithm.

In the following, we will discuss in more detail the steps to create the (ordered) list of choices for the lot assignments, since this process defines the search space and is thus the heart of the optimization, defining all the rules and heuristics to guide the algorithms to proper solutions. An overview of the steps is shown in Fig. 1.

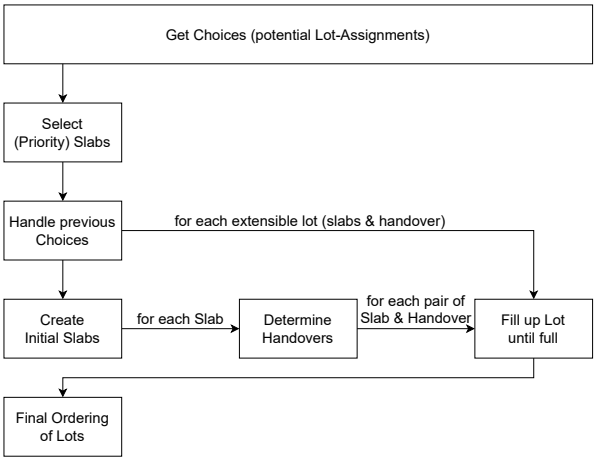


**Figure 1.** An overview of how to create potential lot assignments for a given partial solution.

#### 2.1.2. *(Priority-)Slab Selection*

In the first step, we evaluate which slabs should be put on a handover, i.e., which slabs should be transported out of the storage location, by defining the initial list of unassigned slabs for the root node of the tree search problem. In other words, this filter defines which slabs remain in the hot storage, and which slabs should be transported and therefore assigned to a lot, a vehicle, and a handover. This filter already reduces the search space by limiting the number of slabs for which an algorithm must build lots.

The filters are defined by an ordered list of *selection rules*. When evaluating a single selection rule for a slab, it can choose to exclude a slab, include a slab, or remain undecided and pass on the decision to the next rule. For example, an early exclusion rule checks if a slab is already on a handover, and therefore does not need to be put in a new lot. If the slab is already on a handover, it is excluded and the selection of that slab is determined. Another early rule specifies that if a slab is marked by a human operator for removal from the hot storage, it is included and no further rules need to be evaluated. In this way, rules are evaluated sequentially, with the first rule that is not undecided either including or excluding the slab. If all rules are undecided, the slab is not included.

The list of rules and their order is mainly given by domain experts and can be changed depending on the current optimization goals. The following is an example of a rule set:

1. Excl. slabs where transport is already issued.
2. Excl. slabs that are already on handovers.
3. Excl. slabs with no feasible handover.
4. Incl. slabs marked for transport by the operator.
5. Excl. slabs marked for keeping by the operator.
6. Excl. slabs cooling before transportation.
7. Incl. slabs with known destinations (e.g. slab yard).
8. Incl. slabs marked as "urgent".
9. Incl. slabs to be moved to a warm-keeping box.

10. Incl. slabs that are about to be milled.

#### 2.1.3. Handle Previous Results

Since the hot-storage area is a dynamic environment and solutions can become obsolete due to unplanned events, parts from previous solutions may already be in motion and need to be considered when updating a solution. For example, when a lot assignment is created, and the resulting crane movement requests are sent to the crane operator, we do not want to update or delete these movement requests to give the crane operator a stable preview of the next movements he or she has to perform Therefore, in the data that describes the current state of the hot storage area, which includes all the positions of the slabs and vehicles, we also include a list of *frozen lots*, which are lot assignments where this lot has already been sent to the crane operator system and the corresponding movement requests have already been issued. Similarly, if we have lot assignments from previous optimization runs that are still valid and no movement requests have been issued, we try to reuse those *extensible lots* for efficiency and also to provide a robust preview for the operators.

First, if we encounter frozen lots in the data, we create unmodified lot assignments only for the existing frozen lots and do not consider any of the unassigned slabs. Second, if no frozen lots exist but extensible lots are available, we use the extensible lots as starting points and try to fill them up if possible, using the same procedure as in the later step described in Section 2.1.7. If a lot assignment based on a frozen or extensible lot is applied during the search, this lot is removed from the list of frozen/extensible lots. Only if there are no frozen or extensible lots left, we will consider creating new lots in the next step.

#### 2.1.4. Choices of Lot-Assignments

In the remaining steps, we build choices of lot assignments from which the tree search algorithm can choose. Since the potential combinations of lots that can be formed from the unassigned slabs grow exponentially, we use a more structured approach to creating the lot assignments.

First, we want to limit the number of lot assignments for a current partial solution. This allows us to force important decisions early in the search process via custom priority rules. This does not mean that we only consider slabs that meet one of the priority rules, but rather that we force the tree search to deal with some slabs earlier, and postpone decisions for lower-priority slabs later, when the decision may not be relevant because the low-priority slabs have already been added to higher-priority lots. Additionally, it limits the number of lot assignments created for a partial solution, which can improve performance in cases where the tree search only considers the first few choices (which is a reasonable assumption), since we do not need to create any lot assignments that the tree search will not consider anyway.

When building lot assignments, we start with a list of *initial slabs*. For each of these initial slabs, we also determine suitable handover locations.

With the initial slab and handover location fixed, we consider all combinations of a second slab from the remaining unassigned slabs to have all potential, valid two-element lots of the initial slab and any other slab from the unassigned list. Next, we try to greedily add more slabs until the lot is full. In this way, building lots scale roughly quadratic with the number of unassigned slabs.

The following is a more detailed description of the steps involved in creating lots.

#### 2.1.5. Create Initial Slabs for Lots

In this step, we create a list of initial slabs for lots. Given an initial slab, we will later try to find other slabs that are good matches for that initial slab.

As a first step, we order all remaining unassigned slabs according to a list of *slab priorities*. Similar to the selection rules in (priority-) slab selection, a slab priority defines an order by comparing two slabs and either ranks one slab higher or is undecided. In the case of an undecided slab priority, the next priority rule is evaluated until an order can be determined, or two slabs simply have the same priority.

Again, the list of priorities is provided by experts and can be tweaked according to the current optimization goals:

1. Prioritize slabs manually marked by the operator to leave the hot storage and deprioritize the ones marked to remain in the hot storage.
2. Prioritize slabs marked as urgent.
3. Prioritize slabs scheduled for hot milling later (i.e., not stored in slab yard prior to milling).
4. Prioritize slabs that come next in the milling plan.
5. Prioritize slabs that are currently in a caster and can be picked up.
6. Prioritize slabs that are currently cast but not ready to be picked up.
7. Prioritize slabs with earlier arrival dates.
8. Prioritize slabs that are currently on a buffer stack.
9. Prioritize slabs that are higher up in a stack.

The next step is to select the initial slabs for the lots based on the ordered slab list. To limit the number of initial slabs and thus keep the number of lot assignments manageable, we select only one initial slab per caster context, i.e. the region around a caster. First, if any slabs are marked as going out of the hot storage by the operator, we take the highest priority slab (as per the previous ordering) per caster context that is marked as going out, and do not consider any other slabs as initial slabs. If no slabs are marked as going out, we select an initial slab for each caster context by selecting the highest priority slab currently stocked at a caster or, if no slab is stocked at the caster, the highest priority slab for that context.

The slabs selected in this step are now the initial slabs for the next steps.

#### 2.1.6. Determine Handovers

For each initial slab that will later form a lot assignment, we determine potential handover locations where the lot can be placed. Again, we want to limit the number of potential handovers to avoid generating a large number of combinations to avoid performance issues when increasing the number of slabs and handovers.

For each initial slab, we first obtain all potential handovers that must satisfy the following constraints:

- The handover is within the same or adjacent caster context as the initial slab.
- There is no manual blocking of the handover defined by an operator.
- Various size, weight, and temperature constraints of the transport vehicles are met (e.g. maximum and minimum lengths for certain vehicles).

Currently, we have two types of handovers: a pallet (which is pulled by another vehicle) and a stack (where the lot is picked up directly by a vehicle), with different restrictions. To propose handovers, we take up to three different handovers per handover-type according to the following scheme:

1. Take the earliest available handover (e.g. because it becomes available after transporting a different lot).
2. Take the handover with the least amount of time it would take to move the initial slab to that handover (e.g. zero if it is already there, otherwise the estimated relocation time).
3. Take the handover that is available earliest if no lot is assigned to that handover yet.

#### 2.1.7. Add Slab to Lots

For a given initial slab of a lot and a given handover, in this step,we generate different combinations of slabs that can form the lot. Before forming the lot, the available slabs (i.e., the unassigned slabs that are not yet part of the lot) are sorted. The order used in this step is different from the order used to select the initial slabs because we also take into account how well the other slabs match the properties of the initial slab. For example, if we are sorting two potential slabs with different destinations, and one of the slabs' destinations matches the destination of the initial slab, we prefer the slab with the matching destination. Again, we have a list of rules, and if a criterion (like the slab destination) is indecisive (e.g. all slabs have the same destination), the next rule is checked. To sort slabs based on the initial slab (and the handover), we currently use the following rules:

1. Prioritize the slab with the matching (to the initial slab) disposition, i.e. the decision whether a slab stays in hot storage or will be moved out.
2. Prioritize slabs that both have the hot-rolling mill as their destination.
3. Prioritize slabs with matching rolling mill types.
4. Prioritize slabs with smaller differences in milling due date.
5. Prioritize slabs with matching target location (e.g. storage yard, processing facility, etc.)
6. Prioritize slabs at the same location (e.g. caster or buffer) as the initial slab.
7. Prioritize slabs from the same caster as the initial slab.
8. Prioritize slabs from the same stack as the initial slab.
9. Prioritize slabs that are closer to the initial slab (by estimated movement and excavation times).

Starting with the initial slab at a given handover, we use this ordered list to create different combinations of slabs that form lots. Each time we try to add an additional slab to the lot, we check the following lot restrictions, all of which must be met in order for the slab to be added to the lot:

- If handover is a stack, check if putting an additional lot on the stack violates the maximum height or weight for a vehicle transporting from stacks.
- Check if general with, depth, and weight restrictions of the handover locations are not violated.
- Check if the milling type is homogeneous.
- Check if the temperature is low enough for transportation.
- Check if the slab is accessible (i.e. not burrowed in a buffer stack of slabs that will not be moved).
- Slabs are sorted correctly if the destination requires a specific order (ascending or descending by milling number).
- All slabs are in the same or adjacent caster contexts.
- No manual blocks by the operator at the current slab location.
- We do not cross a caster other than in which the one the slab was cast.

In the first step, we exhaustively create all possible pairs of the given initial slabs plus a second slab from the previously defined list (the order does not matter in this case), where the lot restrictions defined above are satisfied. This results in all possible two-slab lots with the given initial slab. All of these two-slab lots are immediately used as potential choices.

In the second step, for each of the two-slab lots, we start a greedy process that adds as many slabs as possible to the two-slab lot. In this process, we go through all ordered slabs (ordered by slab priority using the initial slab as reference) after the second slab that forms the two-slab lot. For example, if the second slab is at position 10 out of 20, we go through slabs 11 to 20. During the iteration, if a slab can be used to expand the lot by checking the lot restrictions above, we immediately add it to the lot and continue with the next of the remaining slabs. After the two-slab lots are filled with all the greedily added slabs, such a lot is also returned as a potential choice for the tree-search. If no other slab could be found to extend an initial lot, a single-slab lot is created as a fallback, since as we must deliver all the slabs we identified at the beginning.

#### 2.1.8. Final Ordering of Lot-Assignments

From the previous stage, we obtained several potential lot assignments, starting with creating the initial slabs, determining handover, and adding additional slabs to the lot. In this last step, we perform additional ordering to prioritize some lots over others by considering the following lot properties:

1. Consider all lots that have the earliest handover time within a margin of 30 minutes. This usually means all lots that are available now will be available at about the same time later.
2. Prefer lots with three or more slabs.
3. Prefer lots where deliveries can start earlier.
4. Prefer lots with less active crane time.

After this final sorting, these lot assignments can be traversed by the tree-search algorithm, allowing it to gradually form lots according to the various prioritization rules and constraints that exist for the hot storage lot assignment problem.

### 2.2. Stacking

The stacking problem is modeled as a dynamic variant of the block relocation problem with continuous pickup and delivery. Its goal is to fulfill all the transportation requests defined in our lot assignments while minimizing the distance the cranes have to travel. An earlier limited version of this is described in detail by Raggl et al. (2018). Unlike the previous work, the stacking now supports multiple cranes and takes transport lots as input instead of generating them on the fly using heuristics. All stacking restrictions including temperature are handled. This even includes the simulation of the slow cooling of slabs. Since casters are operating continuously, we have to limit the horizon we consider for optimization. We typically choose a horizon of two hours and consider the stacking problem solved when we have stored all the slabs cast within that time and delivered all the transport lots due before that time. We search for the best solution using a variant of tree search called iterated rake search.

## 3. Experiments

For testing, we use 100 real-world problem instances collected every 10 hours over 42 days, capturing a wide range of operational conditions. We used our open-source TreesearchLib [1], which enabled us to test a variety of algorithms, namely Depth-First Search (DFS), Beam Search (BS), Limited Discrepancy Search (LDS), Rake Search (RS), Iterated Rake Search (IRS), and Pilot Method (PM) and compare them to the greedy baseline. Greedy, DFS, and LDS rely on the choices to be sorted so that the most promising candidates appear first. BS uses a heuristic to evaluate how good a given child is and filters only those children that are not in the top n. Finding a good heuristic and picking the right beam width can be tricky. The pilot method introduced in Voß et al. (2005) gets around this problem by evaluating the quality of a branch in the search tree using a look-ahead, and then greedily picking the branch with the best look-ahead result. As a look-ahead we can just greedily pick the first choice until we have a solution. Rake search does a breath first search until a limit is reached and then uses a look-ahead to find solutions, while iterated rake search does this repeatedly and is, therefore, a generalization of PM.

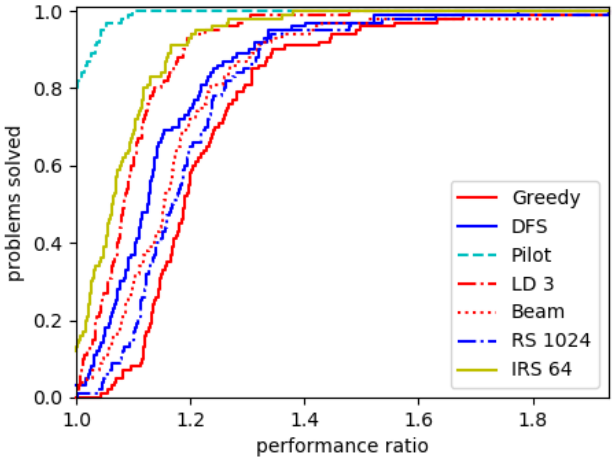


**Figure 2.** Performance profile of selected algorithms.

All of our runs have a time limit of one minute, as this was found to be the upper limit of what is acceptable in the real-world use case due to the uncertainty and ever-changing environment described above. Where there are parameters in the algorithms that can be selected, we have chosen them so that the search takes approximately one minute of time. All benchmarks are performed on a Windows 10 desktop machine with an Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz 8 cores and 32 GB of RAM. Everything is implemented in C# and uses multithreading where appropriate.

Figure 2 shows a performance profile comparing the algorithms. The x-axis shows the performance targets in terms of multiples of the best-known solution, while the y-axis shows the proportion of runs that achieved a given performance target, as described by Dolan and Moré (2002). So the further an algorithm is to the top left, the better.

Naturally, greedy performs the worst, followed by rake search, which is just the best greedy result for the first 1024 nodes in breadth-first order. Beam search does a little better because it can look at choices other than the first one, but it is still pretty bad because we do not have a high-quality heuristic due to the complexity of the problem. Another challenge with beam search is the choice of beam

1 https://github.com/heal-research/TreesearchLib

width, because counter-intuitively a larger beam width does not necessarily lead to better results, as discussed by Lemons et al. (2022). Empirically we found a beam of just 5 to work best. Depth-first search is the next best, but it focuses too much on decisions deep in the search tree. All of the top three methods aim to spread the effort spent more evenly throughout the whole depth of the search tree. In third place is the limited discrepancy search introduced by Harvey and Ginsberg (1995), which limits the number of times the algorithm can deviate from the ordering heuristic. We use a maximum discrepancy of 3 to keep the runtime under control.

It is unclear why the pilot method outperforms the iterated rake search to such an extend, since it is essentially a special case of the latter, with the rake width always set to the number of choices.

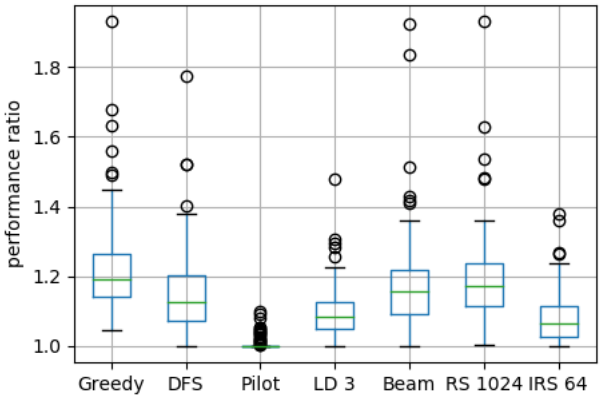


**Figure 3.** A comparison of the qualities achieved relative to the best.

The figures 3 and 4 show the ratio of the found solution to the best-known solution and the runtime required to get there, respectively. Interestingly, the best-performing pilot method has the third shortest average runtime, beaten only by greedy and beam search, both of which produce much worse solutions. Since the pilot method did not need the full minute we set as the time limit, we also experimented with allowing a limited amount of backtracking in the look-ahead by switching to using a DFS. Figure 5 shows that allowing 256 backtracking steps in the look-ahead outperforms 0 backtracking, but allowing 512 backtracking steps underperforms. While this seems counter-intuitive at first, it is explained by the fact that with more backtracking steps we run into the one-minute time limit more often and the PM cannot run to completion.

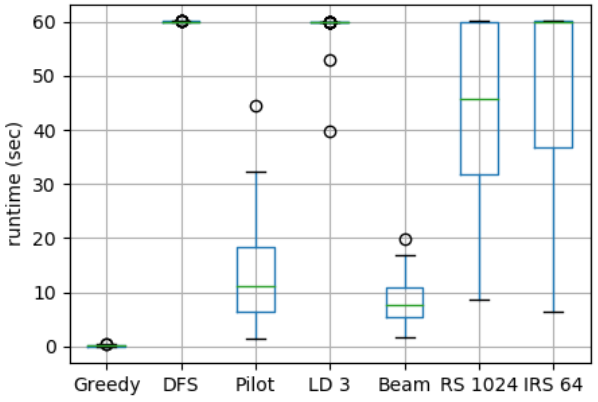


**Figure 4.** A runtime comparison of the algorithms.

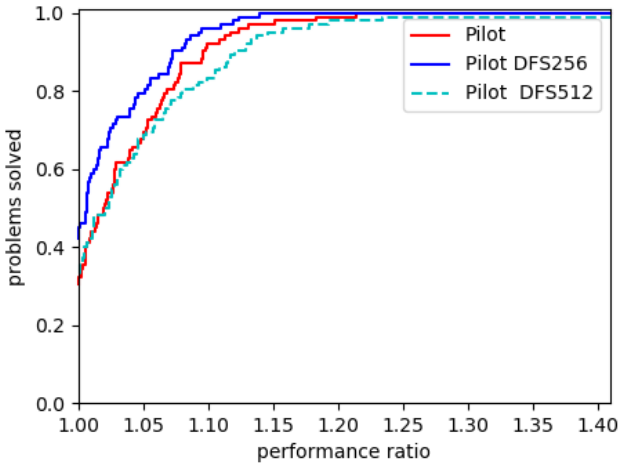


**Figure 5.** Comparison of different look-ahead strategies.

## 4. Discussion

By framing the hot-storage lot-assignment Problem as a tree search, we were able to quickly test a variety of algorithms and heuristics. More importantly, by constructing a solution one lot at a time, were were able to guarantee that every solution found was feasible. This is more difficult to achieve when using any of the population-based metaheuristics that typically operate on full solutions. Additionally, it has the advantage that it is easy to account for the fact that the transport lots have different levels of importance and urgency by sorting them as described in Section 2.1.5. We also tried exact solvers based on a mixed-integer programming formulation of the problem, but implementing the full complexity of the problem proved difficult, and even a variant with key constraints missing was too slow to be useful for real-world use. We found that the pilot method, a tree search heuristic, is very well suited to this problem because it can find good solutions relatively quickly by spreading the search effort evenly over the depth of the search tree. The exact goals and constraints of the lot assignment problem changed several times during development in response to feedback from the operators of the production system. For this reason,

the implementation is written to prioritize extensibility and ease of experimentation over raw performance. However, when we do optimize for execution speed, our experiments show that we have a way to translate this into better optimization results by using a more accurate but costly look-ahead in the pilot method.

## 5. Conclusions

We presented a complex lot-building problem that is part of a larger system to optimize the hot storage area at a steel plant. We have deployed and tested the entire system in the real world. These tests provide invaluable insights into the dynamics of the system, because while both the lot assignments problem and the stacking problem are complex enough on their own, using them together is even more challenging to incorporate the constantly changing conditions of the real world. We are now at a point where the system works very well in general unless there is a new special case we have not yet encountered. The next steps are to make the system robust enough to declare it fully operational. Of course, running such a system 24/7 brings a whole new set of challenges in terms of operator acceptance and maintenance.

## 6. Funding

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

## References

Beham, A., Raggl, S., Wagner, S., and Affenzeller, M. (2019). Uncertainty in real-world steel stacking problems. In *GECCO 2019 Companion - Proceedings of the 2019 Genetic and Evolutionary Computation Conference Companion*, pages 1438–1440.

Dolan, E. D. and Moré, J. J. (2002). Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213.

Harvey, W. D. and Ginsberg, M. L. (1995). Limited discrepancy search. In *IJCAI (1)*, pages 607–615.

Lemons, S., López, C. L., Holte, R. C., and Ruml, W. (2022). Beam search: Faster and monotonic. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 222–230.

Long, J., Zheng, Z., and Gao, X. (2016). Dynamic scheduling in steelmaking-continuous casting production for continuous caster breakdown. *International Journal of Production Research*, 55:1–20.

Raggl, S., Beham, A., Tricoire, F., and Affenzeller, M. (2018). Solving a real world steel stacking problem. *International Journal of Service and Computing Oriented Manufacturing*, 3(2-3):94–108.

Roljic, B., Leitner, S., and Doerner, K. F. (2021). Stacking and transporting steel slabs using high-capacity vehicles. *Procedia Computer Science*, 180:843–851. Proceedings of the 2nd International Conference on Industry 4.0 and Smart Manufacturing (ISM 2020).

Tang, L., Liu, J., Rong, A., and Yang, Z. (2001). A review of planning and scheduling systems and methods for integrated steel production. *European Journal of Operational Research*, 133(1):1–20.

Voß, S., Fink, A., and Duin, C. (2005). Looking ahead with the pilot method. *Annals of Operations Research*, 136(1):285–302.

Özgür, A., Uygun, Y., and Hütt, M.-T. (2021). A review of planning and scheduling methods for hot rolling mills in steel production. *Computers Industrial Engineering*, 151:106606.