CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Online Research Database In Technology

Crane scheduling for a Plate Storage in a Shipyard: Solving the Problem

Jesper Hansen*   Torben F. H. Kristensen§

April 23, 2003

Abstract

This document is the second in a series of three describing an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.

Two gantry cranes move the plates into, around and out of the storage when needed in production. Different principles for organizing the storage and also different approaches for solving the problem are compared. Our results indicate a potential reduction in movements by 67% and reduction in time by 39% compared to current practices. This leads to an estimated cost saving by approx. 1.0 mill. dkr. per year.

This paper describes aspects of solving the model developed and described in Hansen and Kristensen [8]. Conducted experiments and achieved results are reported in Hansen and Kristensen [7].

*Informatics and Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, email:jha@imm.dtu.dk

§Department of Production, Aalborg University, 9220 Aalborg Ø, Denmark, email: tfhk@wanadoo.dk

1 Introduction

This document describes an investigation of possible improvements in methods for handling the storage of steel plates at Odense Steel Shipyard (OSS). Steel ships are constructed by cutting up plates and afterwards welding them together to produce blocks. These blocks are again welded together in the dock to produce a ship.

In Hansen and Kristensen [8] we described the physical environment on the plate storage and how it fits into the overall process of building steel plate ships. Based on the problem description a model was developed including both the physical entities of the system and the planning and processing aspects. In this paper we will discuss the aspects of solving the problem, but we will first briefly repeat the main modelling concepts from [8].

Our model of the storage consists of 8×24 plate stacks, and two gantry cranes can move plates between the stacks. 8 additional stacks are used for arriving plates. Each plate has a due date specifying at which date it must leave the storage and enter the production line. When a plate leaves the storage, it is placed on a conveyer belt referred to as the exit-belt. A maximum of 8 plates can be at the belt at the same time and a plate is drawn from the exit-belt at certain intervals. The two gantry cranes share tracks and collision conflicts must hence be avoided.

The problem adressed is to develop approaches for scheduling the crane operations better than current practices. Three different storage principles are considered:

The block storage is the current storage principle. Two stacks are associated to a specific block of the ship and all plates to be used to produce that block are placed in these designated stacks.

The due date storage was suggested by the shipyard management as an alternative to the block storage. Here each stack is assigned a due date interval and a plate with a due date in that interval can be placed on that stack. The storage is divided into zones and each zone is divided into a number of due date intervals. Several stacks are assigned each due date interval. The user can determine the layout of the storage by specifying the different parameters regarding size of the intervals, stacks per interval, number of intervals and overlap in due dates between zones. Plates are initially placed in zone 3 and are afterwards moved through the zones closer to the exit-belt. Refer to [8] for further details.

The self-regulating storage principle is the last alternative. No specific purpose or due date is assigned to the stacks. The organization of the storage is determined by the planning procedure.

Two approaches have been investigated to solve the scheduling problem. The two approaches are based on different principles. The first approach is an *on-line algorithm* or more precisely a *heuristic discrete event feedback control system* where a movement is chosen in real-time. The other approach uses the on-line algorithm off-line as a greedy construction heuristic to get a good initial solution, which is then improved by local search heuristics. The off-line algorithm makes a schedule for the controller to follow. The schedule specifies the order in which the movements are to be executed. A control module dispatches the movements in the sequences and adjusts the schedule if the initial schedule becomes infeasible because of the underlying stochastic nature of the system.

In section 2 we describe the different local search heuristics implemented, and in section 3 the control system is described. Experiments and results are reported in Hansen and Kristensen [7].

2 Planning Procedure

Given the model of the plate storage, the planning task is now to create a schedule of movements for the 2 cranes without collision, that delivers the plates needed for the given day and minimizes costs. In order to avoid collisions of the two cranes extra positioning and wait movements may have to be inserted in the sequence of movements.

To give a flavour of the size of the problem; consider the scheduling of N movements for 1 crane and M possible stacks. All permutations of N movements are potential, but not necessarily feasible sequences, resulting in $N!$. For each non-exit-movement we have the choice of M possible destination stacks. In total $MN!$ potential solutions. For M and N only 100 we get a 157-digit number! If we consider the case where we only get plates from 10 stacks of 10 plates each, we get at least 10^{10} (10 billion) number of solutions.

We have implemented a construction heuristic (or control method) described in section 3 and local search heuristics for improving the initial plan. After improving the plan, it is returned to the control module to be executed in the simulator with disturbances also described in 3.

A local search heuristic tries to improve the solution by making local or small changes to the plan. A solution achieved by such a local change is called a *neighbour solution*. How we define the changes on a plan is called the *neighbourhood structure*. For a given solution S the set of neighbour solutions reachable with the given neighbourhood structure is simply called the *neighbourhood of* S, $N(S)$. Refer also to Pirlot [12] for an introduction to local search heuristics.

The components in a local search procedure are then the following:

- The *representation* of a *solution*. In our case the initial storage and the sequence movements with movement times for each crane exactly determine the solution.
- *The objective function* is described in section 2.1. The evaluation of the function is in our case done by simulation. This is described in section 2.2. One of the most important points in a successful local search approach is the fast evaluation of the cost function. As we see later this is not trivial in our case. An alternative is trying to estimate the costs, which is described in section 2.6.
- *The neighbourhood structure* is described in section 2.4.
- *Search procedures* are described in section 2.5.

Some of the components can be considered problem *independent* while others are *specific* for the given problem. Generally, the objective function, the representation of a solution and the neighbourhood structure are problem specific.

We have implemented a local search object-oriented framework similar to Andreatta et.al [2] where the problem independent parts of the algorithm is separated from the problem specific. This allows easy reuse of code for other problems. Other local search frameworks and class libraries can be found in Voss and Woodruff [15].

2.1 Objectives

Costs include salary to the crane operators, power for the cranes to move and activation of the magnetic lift as well as maintenance on the cranes. These costs can be mapped to the make-span or finish time of the schedule, the total travelled distance and total number of movements. Minimization of the time

when the last plate is put on the exit-belt is also important to be able to start the following processes as early as possible.

One day of production is planned at a time, so in order to minimize the long-term costs, we measure the quality of the state of the storage after each day. The state is evaluated by 3 different criteria explained in the following.

How well are the stacks sorted by plate due date?

More specifically, how many dig-ups have to be done for each stack with the given plate due-dates. For stack (2,4) in figure 1 plate p_2 is the first to be removed from the storage on November 1. This requires the dig-up of p_1. The next plate to be removed from the stack is p_4, which causes the dig-up of p_3. The result is 2 dig-ups. We have defined the *cost of a dig-up* to be the sum of the following terms:

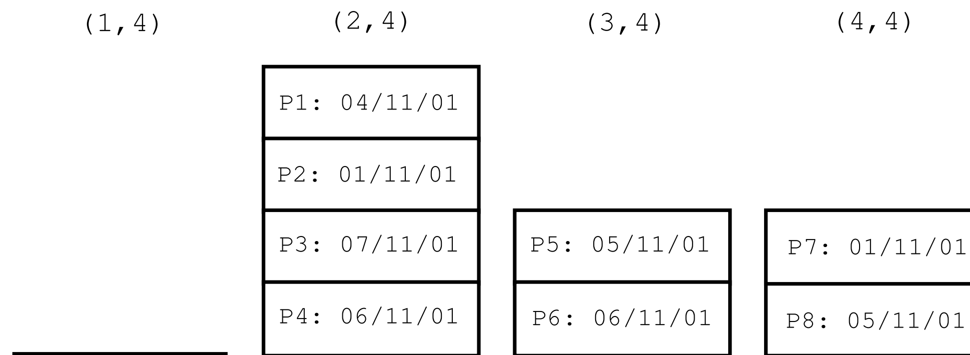

Figure 1: Objective function example.

- Estimated cost of power and maintenance used for lifting and dropping a plate.
- The cost of power for moving the crane to the nearest stack and back.
- The cost in salary for operating the crane in the amount of time according to the dig-up movement.

It is perhaps a bit surprising that according to the management of the yard, the main part of the cost comes from maintenance on the cranes.

How close are the plates to the exit-belt compared to the due date of the plates?

When minimizing travelling distance it is better always to move plates in direction of the exit-belt and when the due-date of a plate is close to the current date, we want it to be located close to the exit-belt. We have chosen to model the cost for each plate p as:

$$\frac{\text{dist}(c_p, c_e)}{\text{diff}(d_p, today) + 1} \tag{1}$$

where $\text{dist}(c_p, c_e)$ returns the distance from the location, c_p, of plate p to the exit-belt, c_e and $\text{diff}(d_p, today)$ returns the difference in days between the due-date d_p of plate p and the current date.

Let us assume that the exit-belt is located at the coordinate (1,4) in figure 1, then stack (2,4) is 1 closer than (3,4). If for instance today is 01/11/2001 then the cost for plate p_1 is 1/4, for p_4: 1/6 and p_6 it is 1/3. The above exit distance can be converted to time and the cost is then achieved by multiplying with the salary of the operators.

How many plates are there in each stack?

We want the plates to be as evenly distributed on the stacks as possible, which will result in less dig-ups when changes in due dates occur. The cost for a stack s is defined to be:

$$\frac{size(s)^2}{maxsize} \tag{2}$$

where $maxsize$ is a user supplied upper limit on the number of plates in the stacks. We observe that the cost per stack is in the interval from 0 to the maximum number of plates in the stack.

If we have an upper limit of 10 plates, then the stacks in figure 1 on the page before will have a cost of $\frac{16+4+4}{10} = 2.4$ while if p_1 was moved to stack (3,4), the cost would be $\frac{9+9+4}{10} = 2.2$, and hence better.

What we are minimizing is in some sense the worst case number of dig-ups if a plate in the bottom of a stack is requested. The cost of a dig-up is therefore used here as well.

Due-date and block principles

For the due date and block principles all criteria are used except for the stack sorting criteria, since that criteria is not relevant for these principles.

2.1.1 Weighting of criteria

We have chosen to convert all the criteria into a common unit, which is then minimized. The user has to supply the operating costs and the wage of the operators. The operating costs of the crane are dividided into costs per lift and per moving second. The majority of the costs are due to maintenance especially on the electromagnetics. The wage of the operators are divided into operators of the cranes and operators of the following production processes and again with the possibility of different wages per shift or for working overtime. The user specifies the length of the three shifts in seconds. The wage is calculated per second, which of course is a simplification of the real world. It is however possible to model a fixed cost of one shift with overtime pay in second shift: Set the wage in the first shift to zero and to the wage plus overtime in second shift. If it is desired to avoid work for instance in shift h a penalty in form of the power, p, can be set:

$$\max(0, tt - t_h)^p * w_h$$

where tt is the total time, t_{h-1} is the user specified time length of shift $h-1$ and w_h is the wage in shift h. Larger $p > 1$ makes it much more expensive to violate the soft constraint. This flexible setting makes it very easy to model different wage structures. This is similar to Lagrangian Relaxation in Integer Programming (IP) where constraints are relaxed and added to the objective function. In IP we usually only allow linear constraints, but in this setting any non-linear constraint can be modelled.

We will later see that the objectives are used both for evaluating solutions and to guide the search for good solutions. In the next section we describe how to calculate the objective value.

2.2 Simulation

Assume that we have given a sequence of movements for each crane. To completely determine the plan we need to find the start time for each movement, determine destination stacks for dig-up and sort movements and perhaps insert

necessary positioning and wait movements in the two sequences. Simulating the crane movements for the given sequences of plate movements does this. Note that we in the system have 2 simulators one for the planning module and one for the control module. The one described here is the planning simulator.

The times for lifting and dropping plates, moving the crane and speed of the exit-belt are not deterministic, but for the purpose of planning they are considered to be. During execution of the plan, changes in times may cause the plan to be infeasible. In that case the plan must be revised. One approach is to adjust the plan according to the changes (control with a plan) and the other approach is to use the construction heuristic as an on-line algorithm (control without a plan). We will consider these issues in detail in section 3.

The complex part of the simulation procedure is handling each type of occuring event:

- Before moving a crane, a potential conflict between the cranes must be identified and taken care of.
- The exit-belt is not ready to receive plates.
- A plate is for some reason not ready to be moved to or from stacks.

In the following we describe in detail how these events are handled in the simulator. Figure 2 on the following page shows an activity diagram of the process of simulating a movement of a plate by a crane.

First the crane must move to the source stack of the plate. Then the plate is lifted, moved to the destination and dropped. Before actually moving the crane, it must be checked whether a collision conflict between the cranes will occur. We will later get back to the handling of conflicts.

The cranes work in parallel. The simulator must therefore keep track of which crane first finishes its current operation and then determine the next operation for that particular crane. When entering the box in figure 2 the next activity of the crane to perform is depending on the previous operation done by the crane. If for example the last operation was lift, then the next is to go to the destination of the movement.

Figure 3 on page 10 elaborates the choices made before moving to the source stack. In situations of choice, arrows with no legend cover the situations not described by any other legend. Text in boxes written in italics are elaborated in other figures. There can be several reasons for the requested plates not being ready to be moved. We return to these issues later. If the

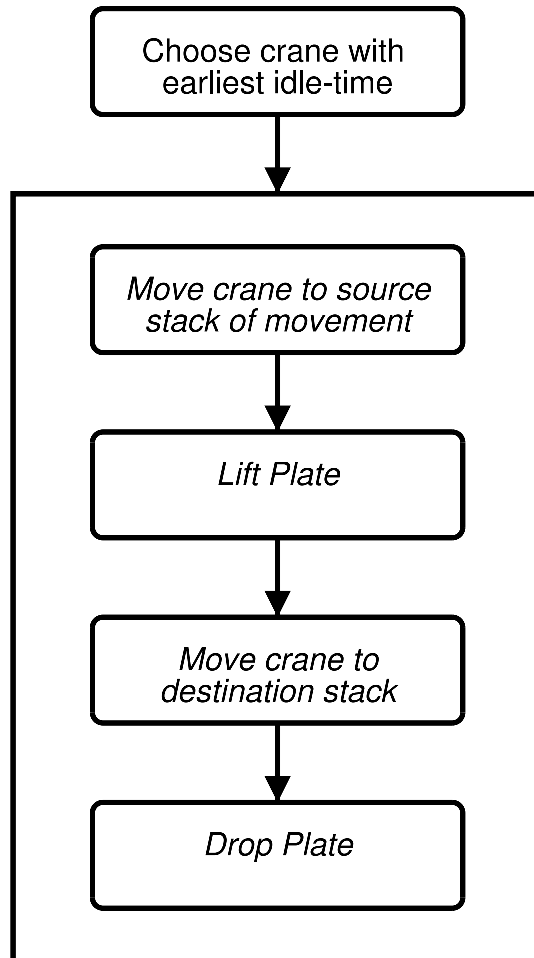

Figure 2: Simulation of a movement.

plate is ready and no crane conflicts will occur the crane is moved to the source stack.

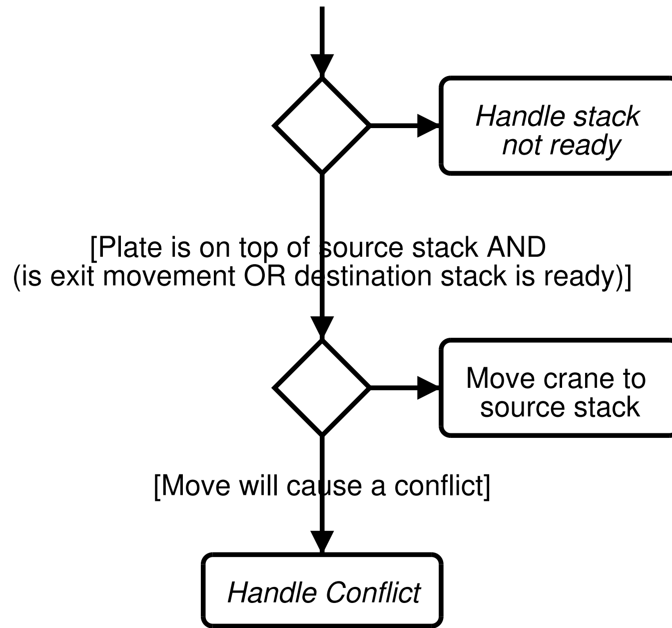

Figure 3: Move crane to source stack of movement.

When the crane arrives at the source stack it is again tested if the requested plate is ready to be moved, since the state of the storage can have changed, while the crane was moving to the source stack. This process is shown in figure 4 on the following page.

After lifting the plate the crane must move to the destination stack as shown in figure 5 on the next page. Again potential crane conflicts must be handled properly.

When the crane arrives at the destination stack, the plate can be dropped immediately. However if the destination is the exit-belt a slot must be available. Otherwise the crane waits until a slot becomes available.

After dropping the plate the crane is ready for a new plate movement as shown in figure 7 on page 13. If a crane has no more movements to execute, it is moved to the end of the tracks in either direction. The tracks end outside the area defined by the stacks and conflicts between the cranes are in that

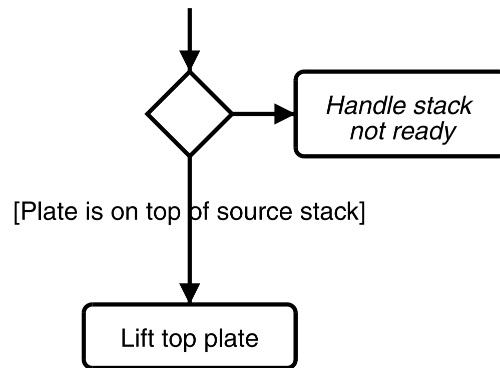

Figure 4: Lift plate.

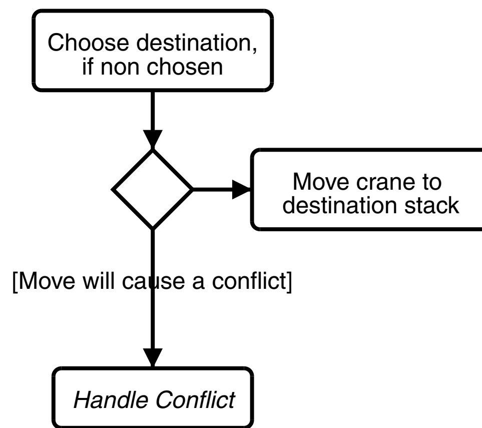

Figure 5: Move crane to destination stack of movement.

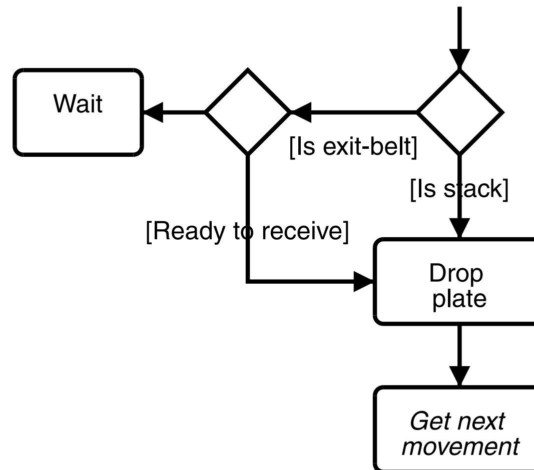

Figure 6: Drop plate.

way avoided.

2.2.1 Handling of Stack not ready

Figure 8 on page 14 shows the process of handling the case when a requested plate is not ready to be moved. Several cases must be taken into account:

The plate is not in the source stack

A plate will need to be moved more than once, if it is moved to a stack in which there are plates that are going to be moved later on the same day. Generally one or more movements are necessary for a given plate.

If at some stage a plate is not in the expected stack, it means that it has not yet arrived from some other stack. If the current movement of the other crane is to move the plate to the stack, then we wait for the other crane to finish its movement. Otherwise we remove the movement from the sequence of the crane as well as any following movements of that plate if any exists. Note that the movement cannot be an exit-movement, since plates involved in exit movements are moved directly to the exit-belt and are therefore always present in the source stack until it is moved.

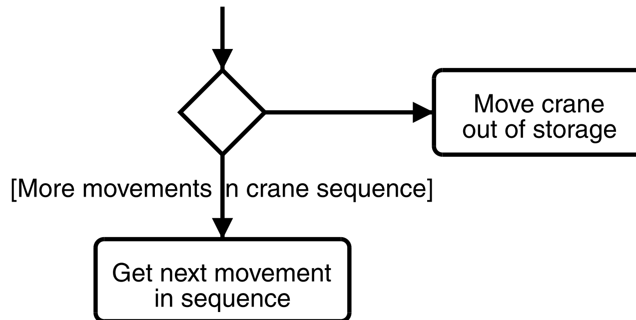

Figure 7: Get next movement in sequence of crane.

The plate is not the top plate of the source stack

If the plate is not the top plate of the source stack, the crane does a dig-up of the top plate, unless the current movement of the other crane is to dig-up the top plate. In that case we wait for the other crane to take the plate.

The destination stack is not ready to receive the plate

If the destination stack is not ready to receive the plate, the reason is that all plates to be removed from the destination stack have not yet been removed. To manage that we first consider moving the plate to another stack, which is ready, and alternatively removing the top plate of the destination stack before the current plate movement. In that way we will in many cases avoid moving a plate more than once on the same day.

This is especially a problem for the block storage where only two stacks are designated to each block, and hence a large number of dig-ups can occur. In the worst case the simulation will enter an infinite loop of dig-ups and positioning movements of the cranes: Assume that crane 1 is going to move several plates from the first block stack and crane 2 from the second stack. First crane 1 lifts a plate from the first block stack and puts it on the second. Meanwhile the crane 2 is requesting the top plate of the second, but has to wait for crane 1 to finish its move. Now another plate is on top and crane 2 needs to do a digup. Crane 2 then moves the plate to the first stack. Again crane 1 has an extra plate to dig-up etc. We deem the solution infeasible if a relatively large number of dig-ups has been done from the same stack. This is actually

the only case not taken directly into account by the simulation. Infeasible solutions are not a problem when using a local search heuristic, since we can simply try other neighbour solution until we find a feasible one. We must however make sure that the first solution is guaranteed to be feasible. The strategies of the online heuristic are somewhat different avoiding this cyclic behavior. Experiments have shown that the cycles only occur for the block storage caused by the many digups. Cycles can hence be avoided by only allowing one of the cranes to move plates from stacks dedicated to a particular block.

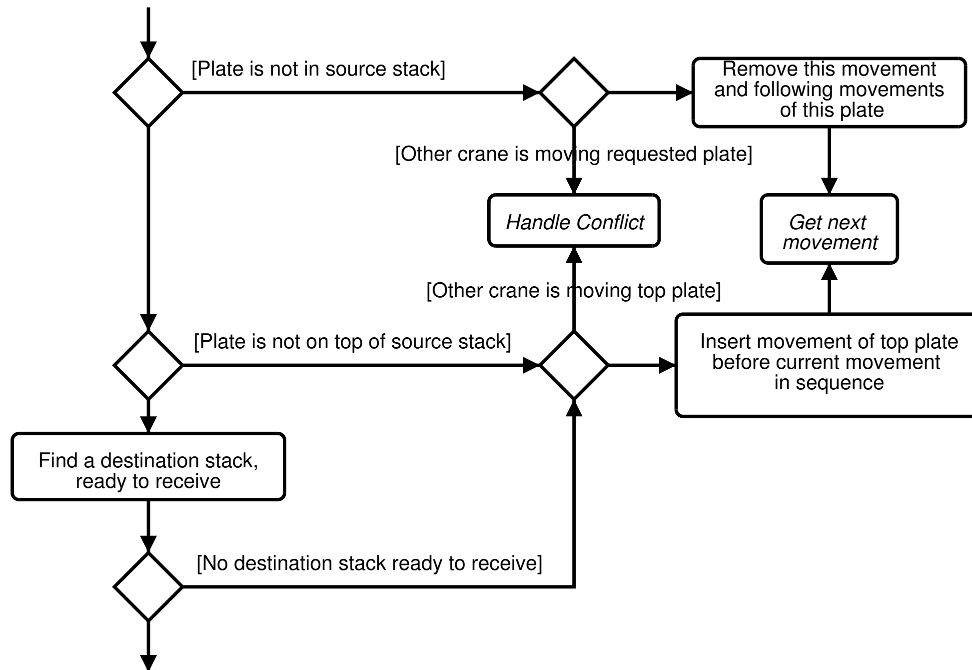

Figure 8: Handle stack not ready.

2.2.2 Collision Handling

It is quite easy to check for collision, when the speed of the cranes is fixed. We only need to check the destination of the crane movement. Assume that the following holds for cranes 1 and 2: $x_1 < x_2$. If crane 1 is moving from x_1

to x_1', we need to check if crane 2 is moving or not. If it is not moving, we check if $x_1' < x_2$, or if it is moving the same for the destination $x_1' < x_2'$.

If a collision of the cranes would occur we handle this as depicted in figure 9. If possible the crane moves closer to the other crane, which will always be closer to the destination of the crane. If the crane is already next to the other crane we either wait or move away for the other crane to finish its operation. This procedure will solve any collision conflict between the cranes.

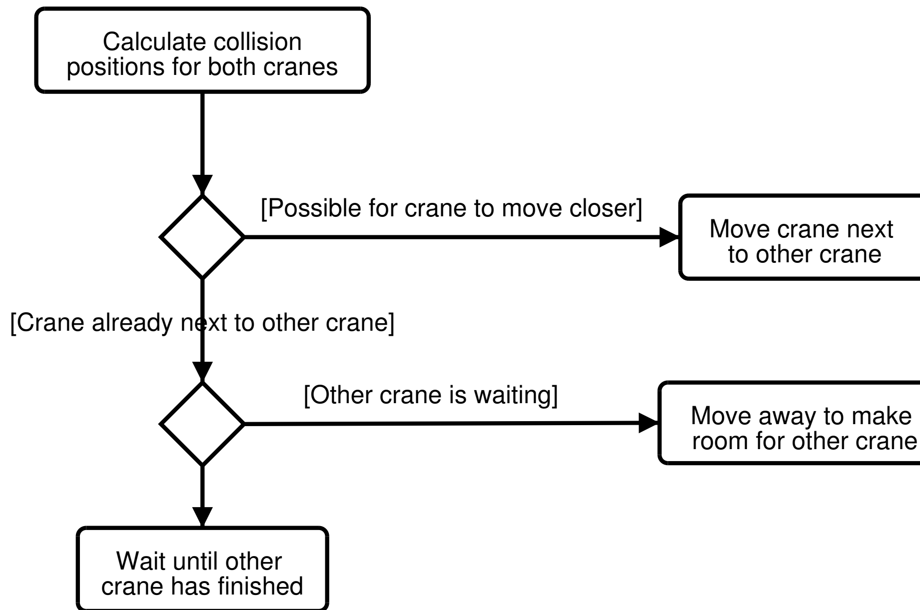

Figure 9: Handle collision conflict between cranes.

Note that this heuristic is not guaranteed to result in the best sequence of positioning and wait movements globally or even locally. We have tried different strategies for handling collisions, and clearly there is a trade off between tight and efficient schedules where the cranes are allowed to move very close to each other and robust but inefficient schedules where the crane movements are more restricted. The most “loose” strategy is when a crane can only perform its plate movement if the other crane is completely out of the area defined by the current position of the crane and the source and destination stack of the plate. Our choice is hence much more tight.

The simulation ends when all movements have been executed. The schedule can now be evaluated according to the criteria mentioned earlier.

2.3 Choosing a destination stack

For movements other than exit-movements we have to decide to which stack to move the plate. This is done during simulation just after lifting the plate. How the stack is chosen depends on the storage principle.

For the block principle either the plate is going to leave the storage tomorrow and a corresponding set of due-date stacks exist, or another block stack next to the source stack is selected.

For the due-date principle a set of due-date stacks exists where stacks in zone 1 are preferred to stacks in zone 2 and so on. From the set of preferred stacks we always choose at random.

For the self-regulating storage principle all stacks are evaluated with regard to several criteria similar to the objective function. Preferably we want to put the plate on a stack where the plate with the minimum due date has a due date, which is later than or equal the due date of the plate we are moving. In that case the stack sorting will not deteriorate. On the other hand, we want the difference to the minimum due date in the stack to be as small as possible, because this will make it easier to find suitable stacks for other plates.

Consider for example the case in figure 10 on the next page. If we in our sequence are moving plate p_7 followed by p_8, we have to decide to which stack of (2,4) and (3,4) to move p_7. The plate p_7 will not deteriorate the sorting of either stacks in position (2,4) or (3,4), but it is still better to move it to (2,4): If p_7 is moved to (3,4), then moving p_8 will cause an extra dig-up at a later stage.

We want to minimize travelled distance to the stack and further on to the source stack of the next movement. For this criteria (3,4) is better than (2,4), since the crane afterwards will move back to (4,4) to pickup p_8. When minimizing stack height, again (3,4) is better than (2,4) according to the stack height criteria. For minimizing distance to the exit-belt, (2,4) is better than (3,4) since stack (2,4) is closer to the exit-belt placed at (1,4).

We will now describe in more detail how a new destination is found for a movement m of a plate p_m with due-date dd_p. First stacks different from the source stack and the currently chosen stack (if any) are identified. From that set we select stacks that are ready to receive the plate, i.e. all plates leaving the stacks have been removed. If that set of stacks is empty, the new

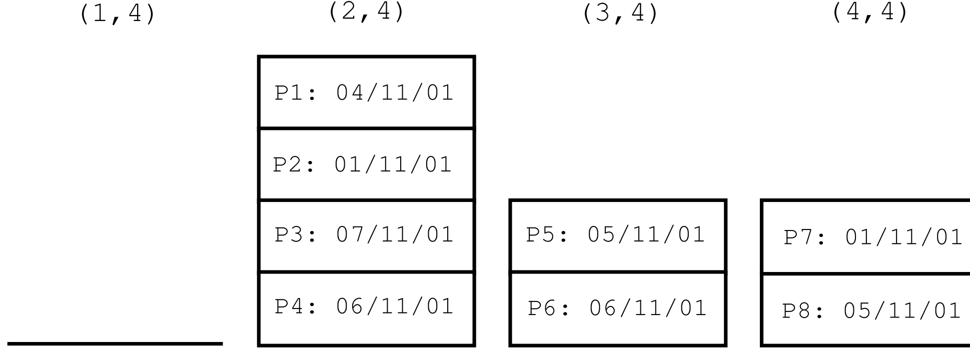

Figure 10: Choosing destination stack example.

destination stack is picked randomly. Let us in the following call the set S' and let md_s be the minimum due date of a plate in stack $s \in S'$:

$$md_s = \min_{p \in s}\{dd_p\}, \ \forall s \in S' \tag{3}$$

From the set S', we identify two subsets:

$$S^{\geq} = \{s \in S' | md_s \geq dd_p\}, \ S^{<} = \{s \in S' | md_s < dd_p\}, \tag{4}$$

If $S^{\geq} \neq \emptyset$ we in the following use that set otherwise we use $S^{<}$ and refer to that set as S''. A subset of the best destination stacks are now chosen and the new destination stack for the movement is a random one from this subset. In the following we describe how the destinations are ranked.

For $s \in S''$ we record $md_s - dd_p$ multiplied by the cost of a dig-up. Let dc_s be that value. We will then have $dc_s \geq 0$ when $dd_p \leq md_s$. Minimizing this cost, we want dc_s to be as small as possible. If $dd_p > md_s$, this cost is disregarded, since no matter the cost we will have to do one dig-up. In addition to the stack sort the following cost changes are calculated:

- Change in cost caused by change in source and destination stack size.
- The cost of moving the crane: The change in cost is based on the time to move from the source to the new destination stack, $dist(s_m, d_m)$, and further on to the start of the next movement in the sequence, $dist(d_m, s_{succ(m)})$, where $succ(m)$ is the successor of m in the sequence of movements.

- The new distance to the exit-belt: The distance is weighted, such that the cost of plates with a due date close in time is larger than for plates due further into the future.

The motivation for these criteria is of course to minimize dig-ups, minimize makespan, travelling time and moving plates due for exit in the near future closer to the exit station. The weighting of these criteria is the same as for the objective function.

2.4 Neighbourhood Structures

It has turned out to be difficult to construct neighbourhood structures where local changes can easily be propagated to global changes in the objective value. Therefore the first attempt was to evaluate the change in objective value by "simply" simulating the crane movements from start to end for every new neighbour solution. This is of course computationally expensive and we have therefore investigated the possibility to estimate the change in objective value instead. This is discussed in section 2.6. Three simple operators define the neighbourhood structure:

Destination operator: Delete a specified destination for a movement and any following movements of that plate. A new destination will then be chosen afterwards while simulating the sequences.

TSP operator: Remove a movement from a crane sequence and insert it at another position in the same sequence.

VRP operator: Delete a movement from one crane sequence and insert it in the sequence of the other crane. The insertion is done at a position in the sequence at approximately the same time in the sequence.

Assume that two plates, p_1 and p_2 in a stack are going to be moved. When using the TSP or VRP operator, it must be checked that plate p_1 above p_2, is earlier in the schedule. We make the following checks for the TSP operator:

- If the movements are in the same sequence, we check if p_1 is earlier in the sequence than p_2.

- If the movements are in different sequences, the solution will always be legal, since the crane doing the succeeding movement just waits until the other crane has finished the preceding movement. This can obviously

result in poor solutions. Therefore we try to estimate the new start and end times of the movements and if waiting is introduced we reject the neighbour solution. Note that this is also handled in the simulation if necessary.

For the VRP operator the two cases are in principle the same. In order to speed up these checks we set up suitable data-structures, such that it is possible to access both predecessors and successors of a movement in constant time. Note that we have two types of predecessors and successors of a movement given by the order of the plates in the stack and in the case that a plate is moved more than once.

In order to reduce the amount of copying object data we simulate the schedule backward to place the plates in the stacks they were at the beginning of the day.

One could come up with more complex neighbourhood structures. Basically all structures suitable for multiple travelling salesman type problems could be used. Note however that more complex structures will also lead to more difficulties in estimating possible savings and trying to check feasibility. This is the reason for our choice of simple operators.

2.5 Local Search Heuristics

A local search heuristic tries to improve the solution by continuously making local or small changes to the plan. We will in the following describe some of the different local search heuristics, which have been implemented.

2.5.1 Steepest Descent

The most simple heuristic is the *Steepest Descent* algorithm shown in algorithm 1. If the neighbourhood is very large we can search for the best neighbour in a subset of the neighbourhood set. The algorithm can be modified to

Algorithm 1 Steepest Descent

1: Select a solution, $s_0 \in S$ with objective $F(s_0)$. $n = 0$.
2: **repeat**
3: $\quad n = n + 1$.
4: $\quad$ Find the best neighbour s_n in the neighbourhood $N(s_{n-1})$.
5: **until** $F(s_n) \geq F(s_{n-1})$

a *Descent* algorithm if the first improving neighbour solution is picked when searching the neighbourhood.

One obvious disadvantage of a simple descent algorithm is it's lack of ability to escape local optima. Other heuristics that do not suffer from this lack are *Simulated Annealing* and *Tabu Search* described in the following.

2.5.2 Simulated Annealing

Simulated Annealing originates in thermodynamics and metallurgy. Basically the problem is that of slowly cooling down metal to a state of minimal energy. In the local search version we have a temperature T as well, the state of the metal is our solution and the energy is our objective function value, which we want to minimize. Kirkpatrick et.al. [10] was the first to "discover" this analogy. The procedure is shown in algorithm 2. We see in that it is possible

Algorithm 2 Simulated Annealing

```
1: Select a solution, s_0 ∈ S with objective F(s_0). n = 0.
2: F* = F(s_0). s* = s_0.
3: T(0) = T_init.
4: repeat
5:    Draw a random neighbour s from N(s_n).
6:    if F(s) ≤ F(s_n) then
7:       s_{n+1} = s.
8:       if F(s) < F* then
9:          s* = s.
10:      end if
11:   else
12:      Draw a random number p ∈ [0; 1].
13:      if p < exp((F(s_n) − F(s)) / T(n)) then
14:         s_{n+1} = s.
15:      end if
16:   end if
17:   n = n + 1.
18: until Stopping criteria fulfilled.
```

to move to solutions, which are not better than the current solution. If the neighbour is worse, we still accept it with a probability depending on the temperature and how much worse the solution is.

The point is that we start with a relatively large temperature T_{init} where almost every solution is accepted, the temperature is gradually decreased until only improving solutions are accepted. In theory this procedure guarantees convergence to the global optimum with probability 1, but unfortunately in an exponential number of iterations, which is generally not feasible in practice. Instead we accept good solutions in a reasonable amount of time.

A number of decisions have to be taken. Choice of initial temperature, T_{init}, the procedure by of which the temperature is decreased (often referred to as the *cooling schedule*), and the stopping criteria.

Initial Temperature

Generally we want the temperature to be high in the beginning, but if the initial solution is reasonably good, we risk destroying the good features of the solution and waste time in the beginning of the search moving to increasingly worse solutions. If the temperature on the other hand is too low, the search will behave like a descent algorithm. Note also that the probability of accepting solutions depends on the objective function. The initial temperature should therefore depend on the objective function as well.

We want in the beginning to accept worse solutions with a probability p_0 which is relatively large: Experiments of Johnson et. al. [9] indicate that a value between 0.4 and 0.9 is appropriate, depending on the quality of the initial solution and how much time is available. We have found that much lower values around 0.1 and 0.2 are required in order not to destroy a good initial solution.

Let ΔF be the change in objective function value between solutions. T_{init} should then for neighbours with increasing costs be set such that

$$e^{\frac{\Delta F}{T_{init}}} \approx p_0 \tag{5}$$

The most widely used procedure to find T_{init}, is to run the simulated annealing in an initial phase where the temperature is adjusted to approach the probability p_0 of accepting worse solutions. Johnson et. al. [9] were the first to suggest this procedure. Let $\overline{\Delta F}^+$ be the average increase in objective value for neighbour transitions to worse solutions. The temperature is then adjusted in the following way:

$$T_{n+1} = -\frac{\overline{\Delta F}^+}{\ln(p_0)} \tag{6}$$

After a number of iterations the initial phase is stopped and the decrease of the temperature is started according to the cooling schedule. T_{n+1} will approach the probability p_0 in (5).

The cooling schedule

The most widely used cooling schedule is the *geometric schedule.* Starting with the initial temperature T_0, the temperature is kept constant for L consecutive moves. Then after L moves it is decreased by multiplication with a constant α, $0 < \alpha < 1$. After nL iterations the temperature is

$$T(nL) = \alpha^n T_0 \tag{7}$$

No definite rules can be given on how to choose L and α, but generally α should be close to 1, e.g. $\alpha \approx 0.99$. L is more difficult to determine. When the temperature is low it should in principle be possible to try all possible neighbour solutions at least once, which suggests a correlation of the size of the neighbourhood and L. For some problems with huge neighbourhood sizes it is unrealistic to have L even close to this value. We decided to set a low L of 10. Assume that a given number of iterations are to be used to get from the initial temperature to the final temperature. Then if L is increased, α must be decreased accordingly and vice versa. In that sense the choice of L and α cannot be separated.

We have implemented a more complex cooling schedule introduced by Aarts and van Laarhoven [1] and also discussed in van Laarhoven and Aarts [14]. Here we will give a brief description of the schedule. It is suggested that the search should be able to reach some *quasi-equilibrium* in the objective values for the given temperature before adjusting it. To ensure a fast convergence to a quasi-equilibrium and hence a small L the decrements of the temperature must be small. During the n'th value of the temperature, we record the objective values and calculate the mean, μ_n, and variance, σ_n, assuming that the values are normally distributed. The decrement rule is then specified as:

$$T_{n+1} = \frac{T_n}{1 + \frac{T_n \ln(1+\delta)}{3\sigma_n}} \tag{8}$$

The *distance* parameter, δ, set by the user, specifies the rate of decrease of T. Larger δ generally result in faster convergence, but worse solutions. After initial experiments the value was set to 0.5. A small value of σ basically

means that the search has reached a quasi-equilibrium and a larger decrease in T is justified. Refer to van Laarhoven and Aarts [14] for an overview of other similar cooling schedules.

Several runs with a faster decrease of the temperature can be superior to one run with a very slow decrease. In particular if the initial solution has been generated with a good construction heuristic. Another possibility we have implemented as well, is a *re-heating* procedure, which increases the temperature, if the search has been trapped in a local optima or has not improved the best solution in a number of iterations. Rules for when, how much and how often to re-heat proved difficult to determine. Since complete restart of the search was just as good, we decided to recommend multiple runs instead.

Stopping criteria

We want the algorithm to stop when future improvements are expected to be small. Usual criteria are therefore maximum number of iterations without improvement or temperature less than a threshold value. In other situations the search has to be stopped because of limited time: maximum running time or maximum number of iterations.

2.5.3 Tabu Search

Tabu Search is like Simulated Annealing a general local search heuristic constructed to be able to escape local optima. Glover et. al. [6] gives an extensive guide to the use of Tabu Search.

The basic idea of Tabu Search is to move to the best solution in the neighbourhood as in Steepest Descent, but where moving to worse solutions is possible in order to escape a local optimum. In case of a symmetric neighbourhood the following cyclic behaviour will occur: When reaching a local optimum there is an immediate risk that moving away from the optimum will result in a move back to the local optimum. To avoid the cyclic behaviour a tabu list, TL, is introduced: We want it to be tabu to move back to the solution we came from, so in principle we could save the entire solution and check whether or not we had visited this solution earlier in the search. It can be both time and memory consuming to do this. Instead we store certain *attributes*, $a(s)$, in the list representing the movement from one solution to another solution. In the following $|TL|$ iterations, it is then forbidden to make

any changes to the solution according to the attribute preventing a return to previously visited solutions.

Let us review the tabu list issue for our crane scheduling problem. For example when changing the destination of a plate movement m from stack (x_1, y_1) to (x_2, y_2), the following different attributes are possible to save in the tabu list:

1. Save the combination move, m, and stack (x_1, y_1), which means that in a certain number of iterations we cannot change the destination of movement m back to (x_1, y_1).

2. Make it tabu to change the destination to *any* other stack.

3. Make it tabu to change m at all, for instance by assigning it to the sequence of the other crane.

We see that the first is less restricting than the second one and again less restricting compared to the third. Later we discuss different ways of representing the tabu list. The pseudo-code of the algorithm is shown in algorithm 3 on the following page. Note that we accept a solution if it is better than the best solution found so far even if it is tabu. This is often called an *aspiration criterion*.

Defining a move to be tabu in the above algorithm is done by adding the attribute to a FIFO list. Alternatively we can define an attribute to be tabu in a number of future iterations. In our application we have an array for each neighbourhood operator and an entry in the array for each plate movement. When a neighbour solution is selected, we record in the appropriate array-entry the iteration number when the plate movement can be changed again by the given operator. We are in other words using the second attribute alternative discussed earlier on the current page.

The most difficult aspects of implementing good Tabu Search heuristics are the questions of which attributes to set tabu and how long the attributes should be tabu. The first is very much depending on the chosen neighbourhood structure and how much cyclic behaviour is observed, but this is however not known in advance.

2.5.4 Reactive Tabu Search

Concerning how long attributes should be tabu, a good choice is to implement a *reactive tabu search* introduced by Battiti and Tecchiolli [4, 3]. The tabu

Algorithm 3 Tabu Search

1: Select a solution, $s_0 \in S$ with objective $F(s_0)$.
2: $F^\star = F(s_0)$. $s^\star = s_0$. $n = 0$.
3: $TL = \emptyset$.
4: **repeat**
5: $\quad \overline{F} = \infty$.
6: $\quad$ **for all** $s \in N(s_n)$ **do**
7: $\quad\quad$ **if** $\Big(F(s) < \overline{F}$ **and** $a(s) \notin TL\Big)$ **or** $F(s) < F^\star$ **then**
8: $\quad\quad\quad \overline{s} = s$. $\overline{F} = F(s)$.
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad s_{n+1} = \overline{s}$.
12: $\quad TL = TL \cup a(s_{n+1})$. If TL is filled up, remove the oldest element.
13: $\quad$ **if** $\overline{F} < F^\star$ **then**
14: $\quad\quad s^\star = \overline{s}$. $F^\star = \overline{F}$.
15: $\quad$ **end if**
16: $\quad n = n + 1$.
17: **until** Stopping criteria fulfilled.

length is changing dynamically in order to intensify the search in promising regions and to diversify the search to investigate other regions of the solution space. In this way the issue of tuning parameters is partially avoided, since parameters controlling the dynamic algorithm still have to be tuned. These parameters are however in our experience easier to adjust and more robust.

In our implementation we record the solution value, iteration number and the number of occurrences of the solution value in a map-type data structure. In that way we can check if the search repeatedly return to the same solution. The chain of neighbour solutions leading back to an already visited solution is often refered to as a *cycle*. Different solutions can of course result in the same objective function value, but that is not a major problem. For a more detailed discussion on that and other ways to store solution configurations, we refer to [4, 3]. If, during the search, the current solution has been visited earlier or rather the objective value has occured earlier, the tabu length is increased. If solutions are chosen which have not been visited earlier, the tabu length is slowly decreased. The reactive search scheme includes a final *escape mechanism*, when revisiting solutions repeatedly. In that case a random

number of random neighbour movements are performed. The number of moves is depending on the length of the cycle.

2.6 Estimation

Calculating start and end times of the movements and evaluating a solution by simulation has shown to be very time consuming. This is especially a problem for heuristics that investigate the entire neighbourhood like Tabu Search. For some real-life-size instances with approximately 800 movements on a day every solution has up to 1/4 million potential neighbour solutions. For this reason we have investigated the possibility of estimating the change in cost of a local change to the solution. Since the cost is a weighted sum of different criteria, we must estimate the change of all these different criteria. In the following we will describe the calculation of the estimates for the different neighbourhood operators.

2.6.1 Destination Stack Neighbourhood

For the destination stack neighbourhood we evaluate changing the current destination stack of a movement to all other stacks. The evaluation is divided into two parts: The saving in cost of not moving the plate to the current destination stack and the cost of going to another stack instead.

The cost saving is a sum of the following terms. Saving in moved distance by the crane from the source stack to the destination and further on to the source stack of the next movement. This saving can influence several cost criteria: Cost of moving the crane, change in makespan for all movements, change in makespan of exit movements and stack costs.

The potential saving in makespan is not at all guaranteed for several reasons. The movement might not be in the sequence defining the makespan and the fact that the two cranes cannot cross each other may later reduce the saving to nothing or even increase the makespan. To reduce the error we only count the saving in makespan, if the sequence is actually defining the makespan.

The saving in makespan of exit movements can only be decreased if the movement is before the last exit movement and it is actually possible to decrease the exit time. We say that it is only possible to reduce the makespan, if the following is fulfilled:

- The crane at the exit belt does the movement.

- No crane has been waiting at the exit belt to deliver a plate.
- The exit time of the current solution is larger than a bound on the best possible exit time. The bound is the time to get all exit-plates through the exit-belt assuming that the belt is never empty.

Note that the issues above for the total makespan also apply for the exit makespan.

Finally we calculate the saving in stack sort, size and exit distance. These costs are estimated by removing the plate from the stack. For the size and exit distance this estimate is correct. For the stack sorting this is however not necessarily the case. Removing a plate and recalculating the cost is of course correct, but the change in the plan can actually change the order of plates being put on the stack later on the same day and hereby result in an error.

Now the plate is to be moved to another stack. The estimated cost of a new destination stack follows the same procedure as for the saving above, but is a bit different for the stack sorting. Now we are to find the correct position in the stack to put in the plate. This is done by estimating the arrival time at the stack and other plates removal times from and arrival times to the stack and by this estimating the position in the stack. Again this estimate may be crude.

2.6.2 TSP and VRP operators

Now we consider the neighbourhood defined by re-insertion of a movement at another position in the same sequence. When removing a movement from a sequence the saving is the start time of the following movement minus the end time of the previous movement plus the time of moving the crane from the end of the previous movement directly to the following movement. When inserting the movement between movement i and j the opposite applies. Instead of going directly from i to j we go via the inserted movement. Whether or not the saving is effective again depends on the movements of the other crane and the exit-belt. If the movement was inserted earlier in the sequence, the plate might have a lower position in the destination stack, and if the movement is inserted later a higher position may result. This is estimated in the same way as for the destination stack neighbourhood.

The estimate for the VRP operator is calculated basically in the same way as for the TSP operator.

2.6.3 Comparison of estimation and correct evaluation

When comparing estimation and exact evaluation two factors are important, speedup and quality of the evaluation. On an example instance with 1856 movements on a day by two cranes it took 67 seconds to evaluate 1000 solutions exactly while estimating the same number of solutions took only 0.19 seconds. A significant speedup of 352.

Figure 11 shows an indication of the quality of the estimation compared to the correct objective function evaluation. The quality is measured in the

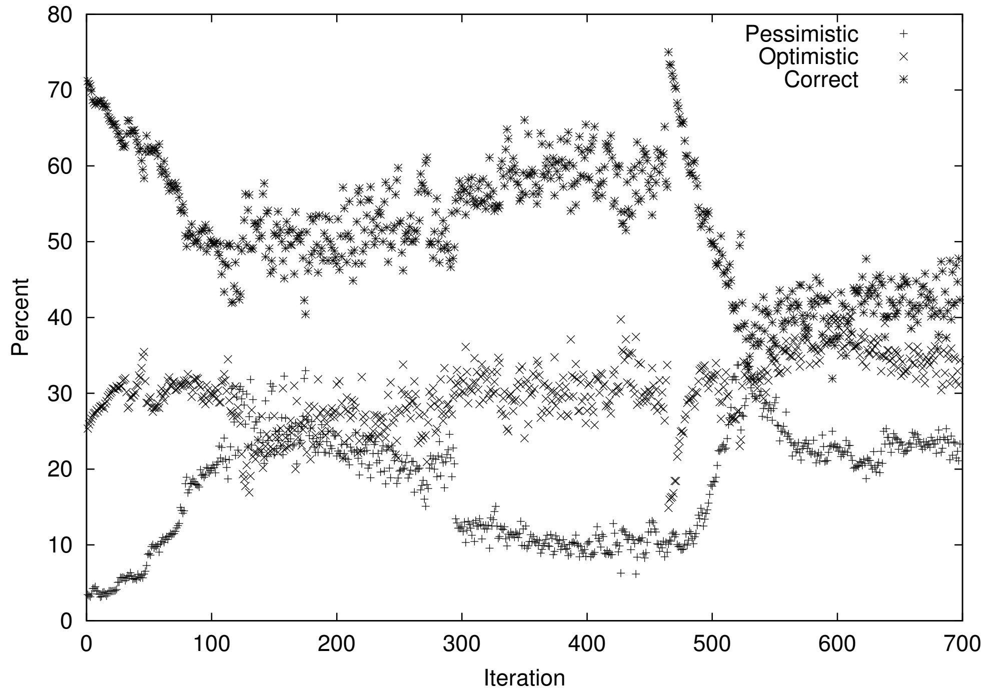

Figure 11: Quality of estimation.

following way. If the estimated change in objective value and the correct change are of the same sign, then we say that the estimate is "correct". If on the other hand the estimate indicates an improvement and the change is actually the opposite, the estimate was "optimistic". The last possibility is the "pessimistic" estimate compared to an improvement of the solution.

The stars (∗) in figure 11 indicate the percentage of neighbour solutions that were estimated correctly. On average only about 50% of the solutions were estimated correctly and it is down to 40%. About 30-40% of the solutions were wrongly estimated to improve the solution, which is too high to be useful. The figure also indicate that after around 500 iterations the quality significantly deteriorate. This is partly caused by the way we measure the quality. At that stage the best changes in objective value are close to zero and hence much more likely to be estimated with an opposite sign.

Figure 12 shows a plot with the first 100 iterations comparing the *best estimated* change and the corresponding *correct* value as well as the neighbour solution with the *best correct* value and the corresponding *estimate*. After each iteration the best estimated neighbour solution is picked. In all 100 iterations the best estimate is too optimistic, but the correct value is still improving the solution.

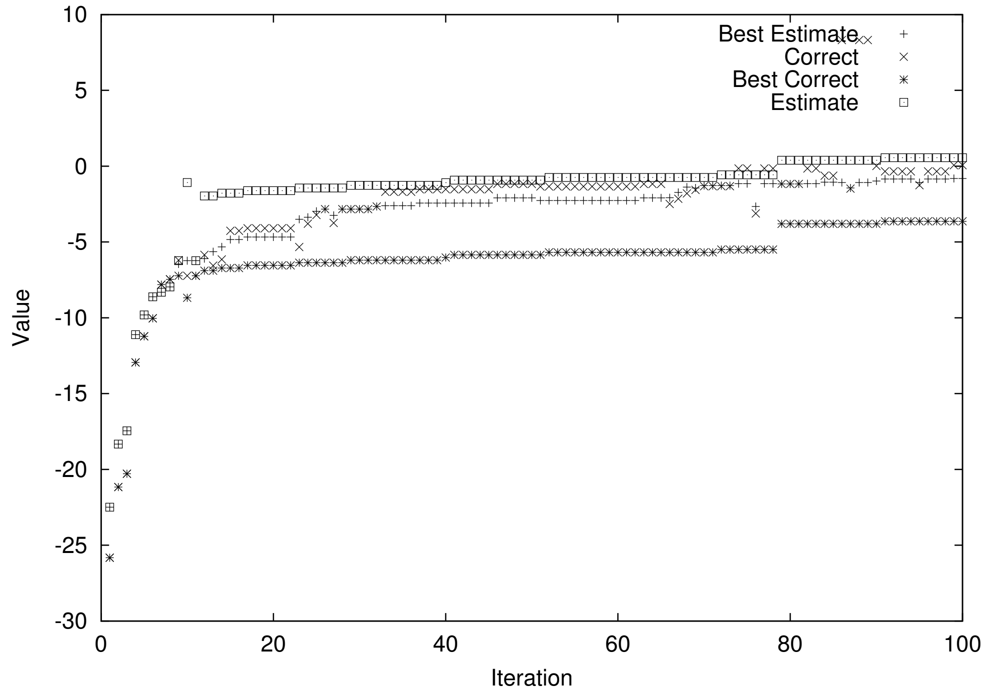

Figure 12: Estimates compared to correct values for iteration 0-100.

In figure 13 the following 100 iterations are shown. As in the previous figure, the *best correct* value is disregarded since the *estimate* is too pessimistic – now above zero. In a lot of cases the *best estimate* is completely wrong leading to very poor solutions.

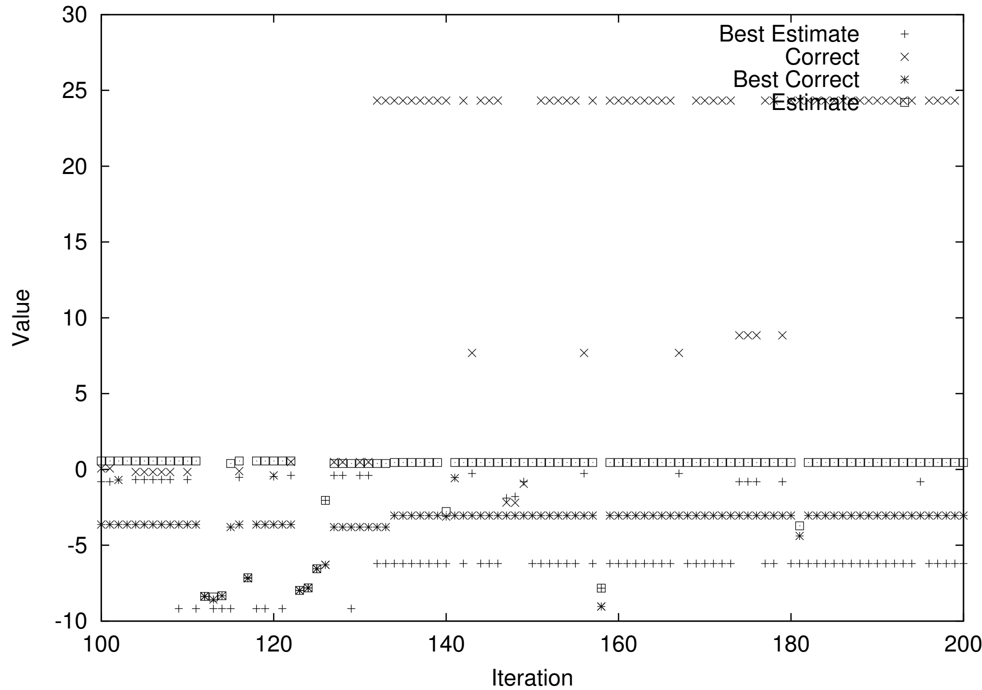

Figure 13: Estimates compared to correct values for iteration 100-200.

To avoid this behaviour we sort the neighbour solutions in increasing order of the change in objective value. Before picking a neighbour as a new solution, the correct objective value is found. If the value is improving the current solution we pick the solution otherwise the next best estimate is evaluated. This is done until all neighbours are evalutated. At that stage we select the neighbour with the best correct value which is then worse than the current.

In a trial run of 3000 iterations, the first solution was picked in 1997 cases, one of the first 4 solutions were picked in more than 90% of the iterations and the maximum number of solutions evaluated was 14. This simple procedure significantly improves performance.

2.7 Hybrid Search

The entire neighbourhood size was 458.012 for the example in the beginning of section 2.6.3. It takes 86 seconds to evaluate all neighbours even with estimation and hence it is too slow to be useful in practice. Instead of evaluating the entire neighbourhood a random subset is evaluated. We have experimented with an initial subset size of 100 ($subsize = 100$), which is slowly increased according to different strategies in order to allow a more thorough investigation of the neighbourhood when it becomes more difficult to find improving neighbour solutions. Specially we increase the subset by one, if the picked neighbour was not the best estimated or if the best estimated was worse than the current solution. In addition to that is a more radical change of the size: If no improvement of the best solution so far has occurred in the last $subsize$ iterations, the $subsize$ is doubled. Trial runs were also done with the descent algorithm. Here a neighbour were only picked if the solution was strictly better than the current. The subset size was incremented in the same way.

Voudouris and Tsang [16] introduces a more intelligent way of restricting the size of the neighbourhood called *Fast Local Search*. Another possibility is *Variable Neighbourhood Search* by Mladenovic and Hansen [11] where the algorithm shifts from one neighbourhood structures to the next at certain intervals or each time a local optima is found.

In Reactive Tabu Search the tabu length determines the degree of intensification and diversification during the search. In Simulated Annealing the temperature and randomness take this role. As we have seen above, randomization can be successfully combined with Tabu Search as well and more intensive neighbour search can be used in Simulated Annealing.

In the literature a lot of other interesting ideas have been suggested to diversify and intensify the search. We will briefly comment on some of them in the following, although we have not implemented them in our system.

Frequency functions are widely used in combination with Tabu Search [6]. The frequency of specific movement attributes are recorded during the search. A high frequency is perhaps an indication of long cycles, which could then be avoided by penalizing the move attribute in the objective function and hence diversify the search. Another strategy is to intensify the search by recording solution features occurring in good solutions and fixing them in a number of iterations.

Augmenting the cost function with penalty functions is also the main idea in *Guided Local Search* by Voudouris and Tsang [16]. The method works on

top of another local search heuristic guiding the search to unexplored regions by penalizing solution features occurring in local optima. Solution features in the plate storage case could for instance be of the following types:

- Destination stack d for a movement m.
- Movement m_i followed directly by m_j.
- Movement m assigned to crane c.

The question is how to represent the features in the objective function: One way is directly, by setting up penalty matrices with elements, p^1_{md}, p^2_{ij} and p^3_{mc}. We get the total penalty by multiplying the penalty matrices with matrices indicating occurring features. Another possibility is to adjust the distance between stacks, which is similar to [16] for the Travelling Salesman Problem. Note that this should only influence the cost – not feasibility.

Our problem is a multi criteria problem where the criteria have been weighted to form one single objective. If we were interested in a *pareto efficient* set of solutions, forcing the search to visit different parts of the solution landscape by changing the weights in objective function could be applied. Ehrgott and Gandibleux [5] gives a good overview of multi-objective combinatorial optimization related papers.

3 The Control System

The control modules are developed in order to handle the stochastic phenomena at the storage and deadlocks or blocking situations.

Two different types of control modules are developed. Both are heuristic discrete event feedback control methods. One of the control modules simply execute the plan received from the planning module. The method is described in section 3.6.

The other module discussed in the following choose the "optimal" job online for the crane that "asks" for a new job. Basically the state of the storage plant is fed back to the control module and it chooses the optimal job from the list of movement jobs and the state of the storage plant in order to fulfil the control-rules and respect the constraints. The concept of the module is depicted in figure 14 on the next page.

Before the control module is called the movements are generated that are to be executed the current day. These movements are kept in an unsorted

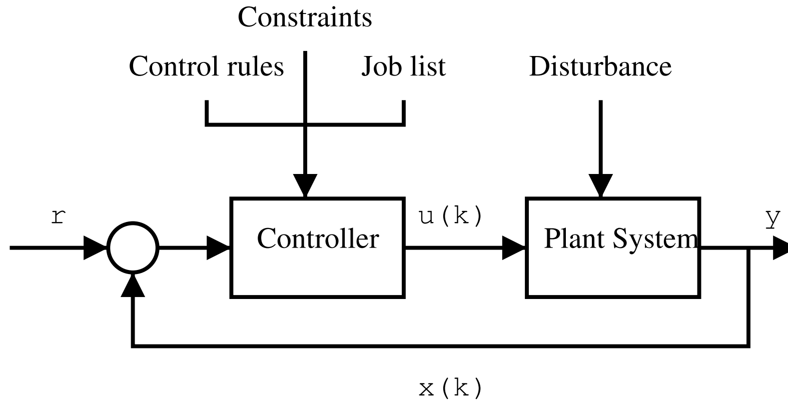

Figure 14: The concept of the controller.

movement list. When the control module is called it splits the movement list into two lists. One of the lists contains all executable movements and the rest of the movements are contained in the other. This is done to reduce the computation time when the movement to be executed next is found, because only the executable movement list has to be searched. A movement is put into the executable movement list when the plate for the movement is located at the top of the stack and the destination stack for the plate is ready to receive the plate. The destination stack is ready to receive a plate when all plates that are supposed to leave the stack the current day have been removed. If the destination is the exit-belt then at least one slot has to be empty.

3.1 Performance

The control systems developed are making decisions by use of information of the current state of the storage. The decision is made without considering how the decision will affect the performance in the future, but when measuring the performance of the system, it is done over a period of time. This means that the rules used by the control system should be formulated to fulfill the goals over a period of time. Performance and robustness are strongly related. It is important that the control system can handle situations where the number of resources is reduced or the capacity of the plant is reduced because of a break down or other reasons. Furthermore it is important that disturbances that

result in variations in process times do not cause the system to break down.

If a control system is designed to support good performance it often uses simulation into the future based on estimates for the process times.

3.2 Robustness

It is important to remember that humans operate the cranes, so the control system is a so-called *Human-in-the-loop feedback control system*. A control system can control both humans and fully automated systems, e.g. robot welding cells. The main difference between controlling humans and robots is that robots must have a control system while the workers use the control system as a guide to make better decisions and thereby increase the utilization of the production equipment and the attached resources. The difference means that control systems developed for automated systems are more detailed and the constraints are not to be violated, whereas control systems developed to control humans can profit by the flexibility of the human brain. With respect to rules and constraints, humans can handle some special or extreme situations. In other words automated systems have a higher need for robustness.

A couple of examples on this for the plate storage:

- One of the cranes breaks down. If the crane can be moved to the safe-position, then the other crane just executes all the remaining movements.

- The exit-belt breaks down. The two cranes execute all movements except from the exit-movements and the movements that are blocked by the exit-movements.

3.3 Collision check

Critical situations never occur when the cranes are moving in the same direction. The cranes move linearly between two points and therefore it is only necessary to check these end-points. The check is performed by simulating one movement forward. Critical positions and respective times for the idle crane are calculated according to table 1 on the following page. These positions and times are compared with the position of the other crane at the respective times. To make a robust check the check is performed by calculating the combinations of worst/best case scenarios with respect to the lift/drop times. In extreme situations when there is no job to execute and the job list is not empty then new destinations are found for these plates and the control module

Position	Point of time
Current position	Current time; (time = 0)
Current position of the plate	Arrival time for the crane
Current position of the plate	Departure time for the crane
Destination for the plate	Arrival time for the crane
Destination for the plate	Ready to departure time for the crane

Table 1: Collision check points.

tries to find a job to execute. This is just done for a limited number of times and if an executable job is still not found then the idle crane is moved to a *safe position*.

3.4 Deadlocks

A deadlock is a situation in which resources used for the same process are effectively preventing each other from executing any task, resulting in both resources ceasing to fulfill their task. Basically there are two ways of dealing with deadlocks. The control module can either be designed to avoid them or they can be designed to deal with the problem when they occur. One may argue that it is obviously better to avoid the problem than deal with deadlocks when they occur. Avoiding them may be cheapest, but it is not necessarily the case, because it also cost to avoid deadlocks. Furthermore it is not guaranteed that all deadlock situations can be avoided, because of disturbances. Dealing with deadlocks dynamically when they occur also costs. If a deadlock occurs a strategy is needed to decide which of the resources that has "the right of way". The other resource should then allow this resource to start executing a task. The strategy can be more or less simple. A simple strategy can be more expensive, but more robust, while a more complex strategy can be better, but less robust.

At the plate storage a deadlock occur when the only executable movements are on the other side of the other crane or if none of the remaining movements are executable, e.g. when the top plates are exit-moves and the exit-belt is full. In these situations when no movement can be dispatched to the idle crane a movement is generated for the crane ordering it to move out of the storage. These positions are denoted as *safe-positions*. When one of the cranes is in its safe-position the other crane can operate freely in the entire storage area.

When the idle crane arrives to the safe-position it naturally asks for a new movement to execute. If there is still no movement to execute then the crane is told to stay at the safe-position until a movement is executable.

The exit-belt cannot result in a deadlock because the cranes are not allowed to start executing a job that is not possible to finish.

3.5 Priorities

If a set of tasks is possible to execute, a choice have to be made as to which of the tasks to execute first. The tasks can often be divided into types with different priorities. The priorities can indicate the necessity of executing a task. The priority can be static or dynamic. If the priority is dynamic the state of the plant and the available resources determines the priorities. The purpose of having different priorities is to increase the performance of the resources and plant.

Generally exit and dig-up movements have higher priority than doing sort movements or moving arriving plates into the storage. These priorities change dynamically for instance if the exit-belt is filled up.

Other rules ensure that one crane is operating in the area near the quay and the other crane is operating near the exit-belt. Thereby the cranes are separated as much as possible and more movements are possible to execute.

As an alternative one could choose to use just one crane instead of two cranes and hence the coordination problem would be eliminated. Naturally the collision check is then not performed, the priorities are different and the crane is allowed to operate in the entire storage area.

3.6 Control with a reference Sequence

A simple control system has been implemented to execute the optimized sequences of movements returned from the planning module. This control module uses the same method as described above for the collision check. If there is no collision the execution continues. Otherwise the controller adjusts the progress of the execution of the sequences for the two cranes. If a crane is delayed the other crane waits. Alternatively if a crane is ahead in time compared to the plan it is ordered to wait for the other crane.

More advanced control systems include the possibility to change the reference plan or let the planning module dynamically improve the plan to take disturbances into account.

Range and Yde [13] suggest another decomposition where the planning module determines a partial order of the movements and the control module afterwards schedules the movements with respect to the partial order. A partial order is found in the following way. For plates in the same original stack a partial order is naturally defined. For all the movements the destination stacks are determined and an order in which the plates are to arrive at the destinations is found. Two movements can then be scheduled independently if all source and destination stacks are different. Given the partial order of movement a second planning phase can either on-line or off-line fully determine the order in which the movements are to executed by the cranes. The suggested approach have not been implemented and will hence not be discussed further.

4 Conclusion

In this paper we have described different approaches to solving the problem of scheduling cranes at a plate storage. The problem is hard to solve since it is both a question of placing the plates on stacks in order to minimize future movements, but also scheduling the crane movements to minimize moved distance and at the same avoiding collisions.

Two different approaches were suggested to solve the problem. An on-line approach and a control approach executing sequences of movements achieved by meta-heuristics.

The complexity of the problem makes it impossible to evaluate exactly feasibility and change in cost for neighbour solutions without simulating the entire sequence. Experiments on estimating the change instead were reported, showing the difficulty of estimating the cost function.

References

[1] E. Aarts and P. van Laarhoven, *Statistical cooling: A general approach to combinatorial optimization problems*, Philips Journal of Research, 40 (1985), pp. 193–226.

[2] A. A. Andreatta, S. Carvalho, and C. Ribeiro, *A framework for local search heuristics for combinatorial optimization problems*, in Voss and Woodruff [15], 2002, ch. 3.

[3] R. Battiti, *Reactive Search: Toward Self-Tuning Heuristics*, Modern Heuristic Search Methods, John Wiley and Sons Ltd., 1996, ch. 4, pp. 61–83.

[4] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing, 6 (1994), pp. 128–140.

[5] M. Ehrgott and X. Gandibleux, *A survey and annotated bibliography of multiobjective combinatorial optimization*, OR Spectrum, 22 (2000), pp. 425–460.

[6] F. Glover, E. Taillard, and D. de Werra, *A user's guide to tabu search*, Annals of Operations Research, 41 (1993), pp. 3–28.

[7] J. Hansen and T. F. H. Kristensen, *Crane scheduling for a plate storage in a shipyard: Experiments and results*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-12 (2003).

[8] ——, *Crane scheduling for a plate storage in a shipyard: Modelling the problem*, Informatics and Mathematical Modelling, Technical University of Denmark, IMM-TR-4 (2003).

[9] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, *Optimisation by simulated annealing: an experimental evaluation; part i, graph partitioning*, Operation Research, 37 (1989), pp. 865–892.

[10] S. Kirkpatrick, C. Gelatt, and M. Vecchi, *Optimization by simulated annealing*, IBM Research Report, RC9355 (1992).

[11] M. Mladenovic and P. Hansen, *Variable neighbourhood search*, Computers and Operations Research, 24 (1997), pp. 1097–1100.

[12] M. Pirlot, *General local search heuristics in combinatorial optimization : a tutorial*, Belgian Journal of Operations Research, Statistics and Computer Science, 32 (1992).

[13] T. M. Range and S. Yde, *Storage management at odense steel shipyard. simulation, product placement and control.*, Master's thesis, University of Southern Denmark, 2002. (In Danish).

[14] P. van Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications*, Mathematics and its applications, D. Reidel Publishing Company, 1987.

[15] S. Voss and D. Woodruff, eds., *Optimization Software Class Libraries*, Kluwer Academic Publishers, 2002.

[16] C. Voudouris and E. Tsang, *Guided local search and its application to the travelling salesman problem*, European Journal of Operational Research, 113 (1999), pp. 469–499.