Computing a toolbit pre-assignment for the AX machine

van Duijnhoven, R.

Award date:
2013

Eindhoven University of Technology

Department of Mathematics and Computer Science

Master's Thesis

Computing a toolbit pre-assignment for the AX machine.

by

R. (Roel) van Duijnhoven

Supervisors

TU/e: prof. dr. M.T. (Mark) de Berg
Assembléon: C.M.W. (Corne) Kuepers and S.H. (Sjoerd) van der Laag

Eindhoven, Thursday 4th April, 2013

Abstract

The AX is a machine developed by Assembléon that uses Surface Mount Technology to pick and place electrical components on panels in a pipelined fashion. It consists of up to 20 robots that work in parallel and can all place components. A robot can only pick up a component if a compatible toolbit is mounted to it. Components of all size and shape can be placed by a robot by exchanging its mounted toolbit with one of the toolbits in the toolbit exchange unit (TEU) that is found on each robot. Assembléon has software that can generate instructions for the AX to assemble panels that lead to low production times. This software computes the toolbits present in the TEU of each robot. There are, however, situations known where the instructions that are generated lead to a large number of toolbit exchanges. These take a relatively long time, and Assembléon believes solutions exist that yield better production times.

In this thesis we present an algorithm that makes use of Simulated Annealing to allocate toolbits to the TEUs of robots. This assignment of toolbits to TEUs can be pre-assigned into Assembléon's software. We show that this software is indeed capable of reducing the production time for these troublesome problem instances when it uses the pre-assignment as computed by our algorithm. Our solution gives a reduced production time for 58% of the inputs, and the average improvement for thoses inputs is 1.8%. If we look at all problem instances we tested (including the ones where our solution was worse) we still get an average improvement of 0.5%, and we believe there is still room for improvement.

Contents

Chapter 1

Introduction

Assembléon is a global supplier of Surface Mount Technology (SMT) pick & place solutions for the electronics manufacturing industry. SMT is a method for constructing electronic circuits in which the components are mounted directly onto the surface of panels (also called printed circuit boards). The pick & place machines developed by Assembléon use SMT to place electrical components on top of panels with high speed and precision.

Figure 1.1: Picture of an AX.

The problem studied in this thesis concerns the *AX*, Assembleon's most sold machine. A picture showing an actual AX is shown in Figure 1.1. The AX machines are used by various manufacturers to assemble motherboards for laptops, for mobile phones, and for many other types of panels. Once fully assembled, these panels contain all sorts of chips, resistors and other electronical components. An AX machine essentially consists of a transport system for transporting the panels through the machine, and a number of robotic arms alongside the transport system that can place components onto the panels. To assemble a panel it is put onto the transport system, and transported through the machine from one position to the next. At each position where the panel halts, one or more robots place a number of components onto the panel. Then the panel is moved to the next position, where each robot places some more components. After the panel has halted at every position, all components have been placed and the panel has been fully assembled. All of this is done in a pipelined fashion: when the first panel moves from the first robot to the second robot, another panel enters so that the first robot can start working on that panel. Thus in a generic time step, the robots are all operating in parallel. Note that the transport system can only transport the panels to their next positions once all the robots have finished. This requires careful scheduling

of which robots places which components, so that the time needed is as small as possible and is distributed evenly among the robots. There are many aspects that influence the time needed by the robots to place their components. One important aspect are the so-called toolbit exchanges, as discussed next.

A robot can pick up a component using a toolbit. The shape and characteristics of the toolbit should be compatible with the component. The components that a machine places vary in size and weight however; from a CPU to a resistor. To be able to place all components on a panel multiple toolbits are thus necessary. In general, it is not possible to schedule the placement of the components in such a way that each robot only has to place components that can be picked with the same toolbit. Hence the robots can change their toolbit during the assembly process. This operation, however, takes a relatively long time.

By smartly deciding which robots places which components (and in what order) the time it takes to assemble a panel can be reduced. Assembléon has developed software for this, for the AX machine. While this software usually performs quite well —that is, the resulting schedule for the given panel leads to a low production time for the panel— there are also cases where significant improvements seem possible. In particular, there are situations where the current software is known to generate lots of these toolbit exchanges.

In this thesis an algorithm will be described that makes use of Simulated Annealing to solve a simplified version of the problem that the AX software is facing. The solution obtained in this way can be fed to the AX software in an attempt to solve the problems involving toolbit exchanges.

We have experimentally evaulated our new algorithm, and compared it to Assembléon's existing *Optimizer* algorithm. For the simplified version of the problem, our algorithm outperforms the solution generated by the *Optimizer* in 74% of the cases, and the average decrease in assembly time for these cases is then 7.3%. When we feed our solution —more precisely, how it distributes toolbits over the robots— as a starting point into the *Optimizer*, then we obtain improved assembly times in 58% of the cases, with then an average improvement for these cases of 1.8%. The improvements are particularly good when the original *Optimizer* generates many toolbit exchanges. Surprisingly there are also cases where our solutions generates more toolbit exchanges than the *Optimizer*, but still has a lower assembly time. Apparently our algorithm is also good at balancing the workload over the robots.

Chapter 2

Background on the AX

2.1 Hardware

High level. The AX is build around a *transport beam*. This transport beam is used to transport *panels* from one side to the other. Multiple panels can be on the transport beam at the same time. As panels move over the transport beam, components get placed on them. A component is placed on a panel by a *robot*. There are multiple robots on an AX, which can work in parallel on the panels lying beneath it. The components that are placed by a robot are taken from a *feeder bank*. The AX consists of a number of *segments*. Each segment consists of four robot slots; robots can be mounted upon one or more consecutive slots. Furthermore each segment has its own feeder bank from which the mounted robots can take components.

There are two types of AX machines. They only differ in the number of segments. The *AX301* has three segments, the *AX501* has five segments. A high level visualization of an AX301 can be seen in Figure 2.1.

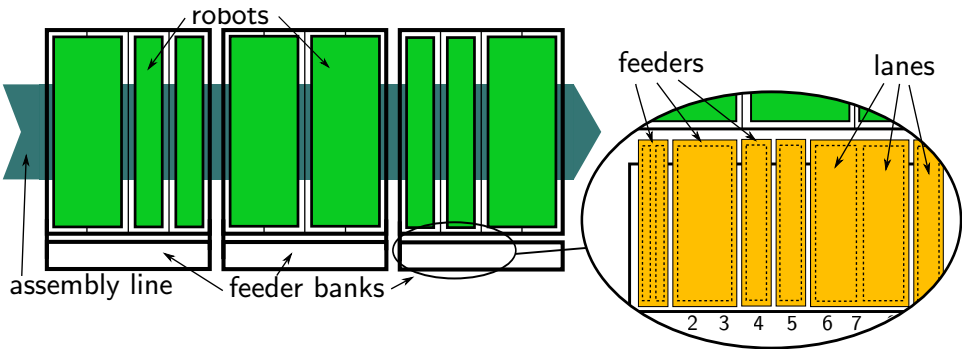

Figure 2.1: High level visualization of an AX301 configured with eight robots. The rightmost feeder bank is zoomed in at. One can see six feeders mounted to the feeder bank slots. Some feeders take up multiple slots. The lanes of the feeders are represented by dashed boxes. Some feeders contain multiple lanes. Inside of each lane a part can possibly be placed.

Feeder bank. A fully assembled panel has lots of components on it. An example of such panel is the motherboard of a smartphone. A picture of a fully assembled panel can be seen in Figure 2.2. A component is characterized by a location on the panel and a part. A part describes the piece of electronics. A part can for example be a resistor or a memory chip. Parts are initially placed in the feeder bank, and it is the task of the robot to pick them up and place them onto the panels.

Parts cannot be placed in the feeder bank directly. First *feeders* need to be mounted on the feeder bank. Each feeder bank consists of 27 slots. A feeder can be mounted on one or more consecutive slots. The number of slots depends on the feeder's type. A feeder consists of one or

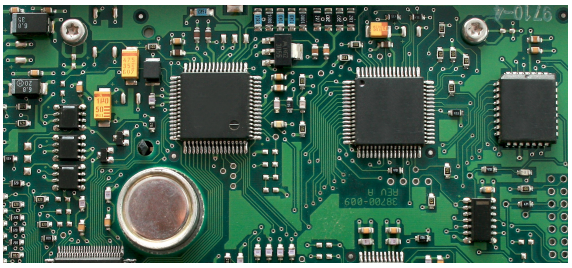

Figure 2.2: Example of an assembled panel.

Figure 2.3: Seven different toolbit types that the AX can use. Note the different tip at the end of each toolbit.

more *lanes*. A lane can house a large number of components of the same part type. However a lane can only hold a part if the part type is compatible with that lane. A feeder bank, configured with feeders, can partially be seen in Figure 2.1.

Robot. Each robot is located in a so-called *module*. A module is essentially a box that contains all hardware that is needed to pick and place components using the robot. It can can be mounted upon one or two slots of an AX. Once mounted, the module partially overlaps both the transport beam and feeder bank. The robot can move horizontally through the space that the module takes up. Since the fact that robots are contained in modules is not relevant to the problem we study we will talk only about robots in the remainder of this document. Attached to the robot is a *placement head*. This placement head contains a *nozzle* that can be used to pickup components. The placement head can move up and down and can rotate around a vertical axis. Remember that the robot partially overlaps the feeder bank; thus by moving the robot a component can be picked up from a feeder using the nozzle of the placement head. By again moving the robot, and possibly rotating the placement head, the component can be placed somewhere on the area of the panel that is reachable by the robot.

Before a component can be picked up, a *toolbit* must be mounted to the nozzle of the placement head. Because of the different size, structure and weight of parts, a single toolbit cannot pick up all types of parts. Therefore multiple toolbits exist. A picture showing some of the toolbits is given in Figure 2.3. Each robot is equipped with a toolbit exchange unit (TEU) that can hold up to eight toolbits. The placement head can exchange the toolbit currently in its nozzle with one out of the TEU. This is called a toolbit exchange.

A component may only be placed once the robot can guarantee the placement on the panel to occur within a certain precision. The precision that is required depends on the part. To obtain a higher placing precision the robot needs to figure out how the panel is positioned. This can be achieved by reading *fiducials*. The exact implementation is far more complex and is outside of the scope of this master's thesis.

An abstract visualization of a robot as found in a module with all the previously described components can be seen in Figure 2.4.

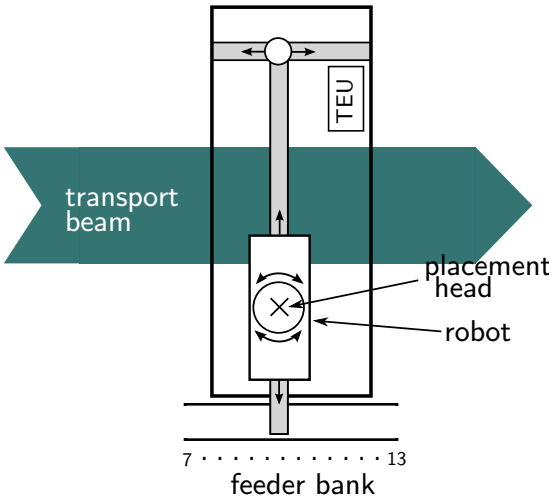

Figure 2.4: Visualization of a robot inside a module.

Transport beam. Panels are transported through the AX via the transport beam. The transport beam can be in one of two modes: either it clamps itself to all the panels in the AX; or it is unclamped from the panels. Only once the transport beam is clamped to the panels can it actually transport the panels. More importantly, robots can only work on a panel once it is clamped to the transport beam. Clamping guarantees that the panels will not move and allows placement of components with the required precision.

However, unlike a conveyor belt, the transport beam can not continuously move in one direction. The transport beam can only move a limited distance in one direction before it runs out of space. This problem is tackled by unclamping the transport beam once it can no longer move forward; by moving it (unclamped, thus without the panels) backwards and finally by clamping the panels back to the transport and continuing the transport of panels. This is called a *return* stroke. An example of this effect is visualized in Figure 2.5.

2.2 Panel specification

The description of a fully assembled panel is given by a *panel specification*. This specification is given by a set of components with their locations that need to reside on any assembled panel. Furthermore the panel specification contains information about fiducials that are found on a panel.

2.3 Placement program

The AX is controlled by a *placement program*. A placement program contains instructions for both transport beam and robots. These instructionts are meant for an AX conforming to a specific machine configuration and setup (see Section 2.3.1). Only once an AX is configured in this way can it actually perform the instructions in the placement program. The instructions are described by a *cycle* (see Section 2.3.2).

2.3.1 Configuration & setup

The operator of an AX should make sure the machine *configuration* and *setup* is met before any of the instructions contained in a placement program are run.

The machine configuration describes essentially all hardware pieces that are difficult or impossible to change. It consists of the AX type and the robots mounted on the robot slots of each section.

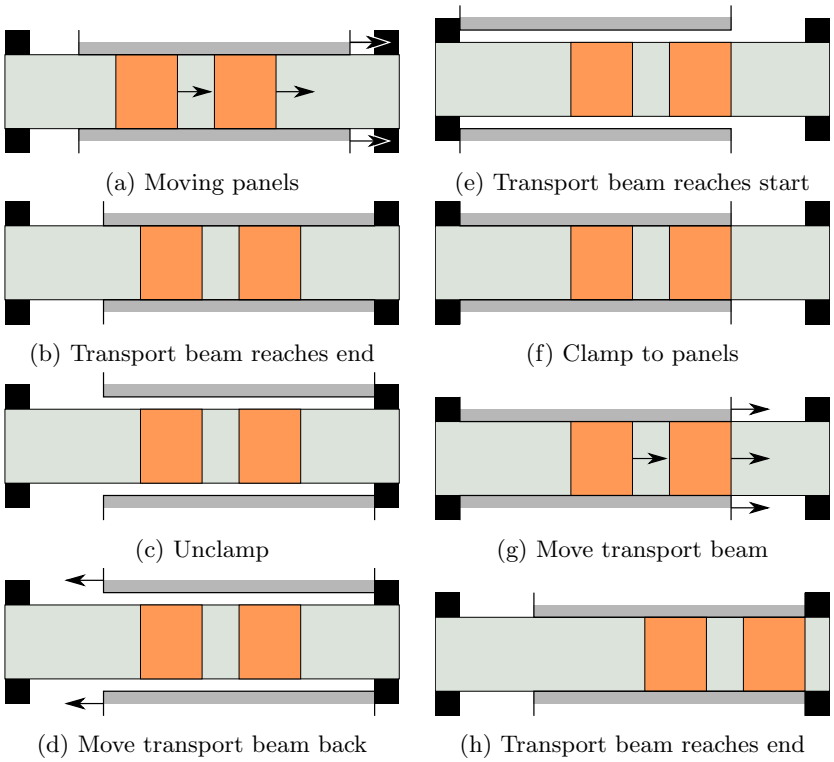

Figure 2.5: How transport beam acts a a conveyor belt with use of return stroke.

The machine *setup* describes the hardware that can be changed relatively cheaply. This is given by the toolbits found in the TEU of each robot; the feeders attached to the feeder bank slots of each section; and the parts that are located in the lanes of feeders.

2.3.2 Cycle

A *cycle* describes how an AX can produce panels in a pipelined way. The initial state of a cycle is given by a transport beam that is filled with panels. The further a panel is on the transport beam, the more components have been placed onto it. The panel at the start of the transport is empty while the panel at the end is fully assembled. Now during a cycle the panels are transported by the transport beam in such a way that all panels move exactly one position further. At the start of the cycle the fully assembled panel at the end of the AX is transported out, and at the end of the cycle an empty panel is inserted at the start of the AX. Once the cycle ends we are back in a state that is identical to the initial situation at the start of the cycle: there is an identical number of panels inside the AX lying at exactly the same locations.

During the transport of the panels in a cycle, multiple stops occur. Such a stop is called an *index step*. Only during these index steps robots can work on the panels lying in the AX. Let us define a *bucket* to be a collection of components that are placed by a particular robot in a particular index step. A robot has thus a bucket for each index step in this cycle. The total number of buckets is given by the number of robots multiplied by the number of index steps. To make sure that an assembled panel leaves the AX each cycle, all components have to be placed exactly once during a cycle. Thus each component should be allocated to a single bucket, that is going to place it.

A component can not be placed in any bucket. The main reason for this being that a component may not be reachable by a robot in an index step. Recall that an index step is a position where the transport beam halts, so robots can work on panels. The positions of the index steps decide where the transport stops, and thus where the panels are lying in the AX at each index step. Where

the panels are lying decides what components a robot can actually place, namely the components that are located on an area of a panel that is reachable by the robot. Let us define all the buckets for which the robot can reach a component in that index to be the *reachable buckets* of that component.

2.3.3 Quality

For a single panel specification there exist a vast number of placement programs that place all the components correctly onto the panels. A component can be placed by a robot in another index step for example. A part can also be found on multiple feeder banks allowing more robots to place components of it. Obviously, the performance of these programs may differ.

The *cycle time* can be used to compare placement programs. The cycle time is the time it takes for all instructions in the cycle to be executed and is thus equivalent to the time it takes the AX to repeatedly output a new assembled panel.

Now what are items that make up the cycle time? The transport beam transports the panels iteratively to the next index step of the cycle. After the panels are in position the robots can work on the panels. Once they are done the transport beam moves on to the next index step. As these actions happen strictly after each other, the cycle time is given by the sum of the time all these actions takes.

The time it takes the robots to work in an index step can be broken down further. Remember that the robots work in parallel, thus the time it takes all robots in an index step to complete their work is the time the robot that finishes its work last takes.

The actions in a bucket are performed in sequence by the robot, thus the sum of the time these individual actions take is the time this robot needs. An action is either placing a component, reading a fiducial, or exchanging the toolbit. A robot needs to perform a toolbit exchange just before it is about to place a component of a part that is not compatible with the toolbit that is currently attached to the nozzle of the robot.

An example of a solution can be seen in Figure 2.6. The cycle time of this solution is given by the sum of the times index step one and index step two take up. The time each index takes up is represented by the dotted line, and is the maximum time among buckets. The time index step one takes, for example, is clearly determined by the time robot R_1 takes. As a consequence robot R_2 is idle half of the time. The cycle time could well be reduced if some of the load of robot R_1 in this index step is offloaded to robot R_2. However, this may not always be possible.

2.4 The *Optimizer* package for computing placement programs

Even for panel specifications containing a small number of components it is not trivial to make a valid placement program. For each single component a robot that is going to place it should be chosen, as well as the index step in which this robot is going to do so (in other words: a bucket should be chosen for every component). However doing so may influence the toolbits that this robot needs to carry in its TEU and the parts that it should have on the feeder bank. All these can, directly or indirectly, influence the cycle time that one can achieve. Obviously the exercise of obtaining an optimal cycle time for a placement program gets even harder as the complexity of the panel specification increases. To that end the AX comes bundled together with tooling. The *Optimizer* software package is part of that bundle and can be used to generate placement programs. The *Optimizer* will produce a placement program based on the input that it receives. The most important part of the input of the *Optimizer* is given by a panel specification. The *Optimizer* will try to produce a placement program for assembling the input panel specification yielding an optimal cycle time for the input panel specifications.

Recall that the machine configuration is given by the robots that are configured on an AX. Replacing or changing these robots can be very costly or involve a lot of work. Therefore the machine configuration for which one wants to produce a placement program is to be given to the

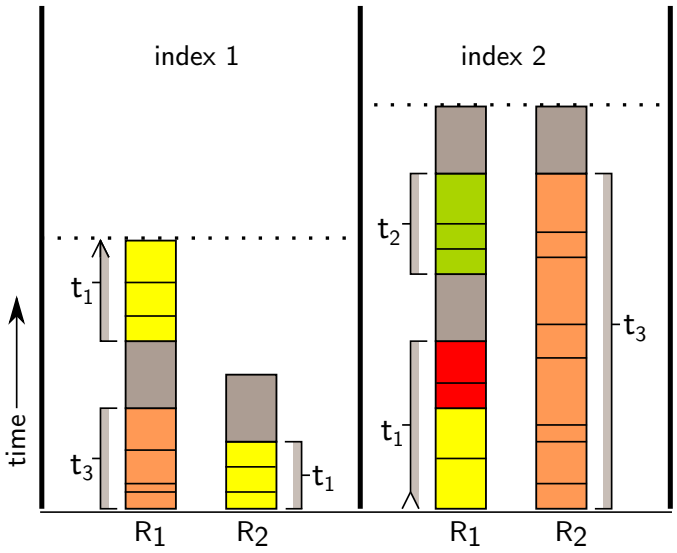

Figure 2.6: Example of a solution with two index steps and two robots. The time it takes to read a fiducial is not shown. A grey coloured box represents a toolbit exchange. Non-gray boxes of the same colour represent placements of components of the same part type. One can see that components of the same part type do not necessarily take the same amount of time. The toolbits that are used to place components are annotated. An arrow on the toolbit annotation marks that a toolbit is carried over by a module from that index to the next. For example toolbit t_1 that is used at the end of index step one of robot R_1, is reused in index step two.

Optimizer as input. The *Optimizer* will then make sure that the given machine configuration is used in the placement program that it generates.

It is up to the *Optimizer* to compute a machine setup: the toolbits in the TEU of each robot, the feeders in the feeder banks and the parts to be found in the lanes of these feeders. The *Optimizer* may only use toolbits, feeders and parts that are owned by the customer. A library of resources that the *Optimizer* may draw from is therefore part of the input of the *Optimizer*. Each resource in this library may have an *inventory* accompanied with it, which specifies how much of this resource is available. Parts, feeders and toolbits may have an inventory. The performance of a placement program is largely influenced by the resources that one chooses to use from this library.

Optionally, the user of the *Optimizer* may choose to preassign some of the resources to a position on the machine. For example a toolbit may be assigned to the TEU of a robot. These pre-assignments must be obeyed by the *Optimizer*.

The *Optimizer* can generate placement programs for multiple panel specification at once that all share the same machine configuration and setup. For simplicitiy this is ignored.

A schema depicting input and output of the *Optimizer* is given in Figure 2.7.

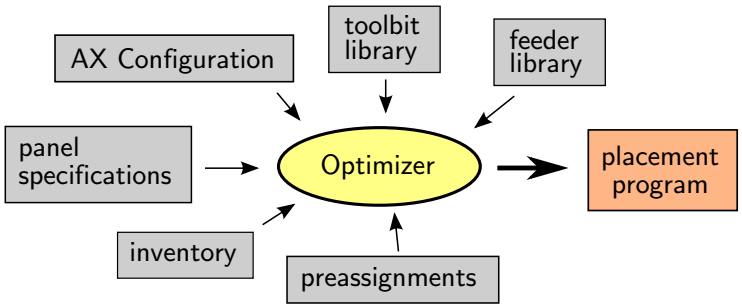

Figure 2.7: Input and output of the *Optimizer*.

Chapter 3

The algorithm of the current *Optimizer*

In this Chapter we will discuss the internals of the current *Optimizer*. The largest part of this algorithm is driven by a *Genetic Algorithm*. First a high level description is given. Then we zoom in at the important steps.

3.1 High level description

A complete placement program is be generated by the *Optimizer* by executing three steps in sequence. Let us briefly describe these three steps.

1. Index steps are generated primarily based on the panel dimensions and the machine configuration. Recall that these index steps decide the reachable buckets of each component.
2. A robot can contain up to eight toolbits in its TEU. In this step the algorithm figures out which toolbit it thinks are most useful for each robot, and places these in the TEU.
3. Using the information computed in the previous two steps a Genetic Algorithm is started. This algorithm will evaluate a large number of solutions and finally report the best solution that it found.

Both step two and three involve toolbits and toolbit exchanges, therefore both steps are described in more detail in the next sections.

3.2 Fill TEUs of robots

This step will fill the TEU of each robot with toolbits that may be relevant later on. These are the toolbits that actually end up in the AX. However, no guarantee is given that these are actually used, as will become apparant later on.

To decide how relevant a toolbit type is, upfront the *workload fraction* is computed for each toolbit type. It is defined as the number of components that are placeable using this toolbit type divided by the total number of components. Note that the sum of the workload fractions of all toolbit types is generally higher than one, because multiple toolbit types are generally able to place the same component. If we now assign a toolbit to a robot we expect that the fraction of the components that this robot can place using that toolbit is given by the workload fraction divided by the number of toolbits of that type that are already assigned plus one.

Now iteratively the toolbit is selected that can place the highest fraction of components. For each toolbit a TEU is chosen to which it should be allocated. This is the TEU containing the

most free spots, that does not already have this toolbit assigned to it. This process repeats until there are no useful toolbits anymore, or all TEUs contain eight toolbits. The latter is usually the case.

As an example assume that two toolbits $_1$ and t_2 exist with a workload fraction of respectively $\frac{5}{6}$ and $\frac{1}{3}$. First toolbit t_1 is chosen, as it can place the highest fraction of components. In the next round toolbit t_2 can still place the same fraction of components, given by its workload fraction. For toolbit t_1 the fraction of placeable components reduces, there is namely already a toolbit of this type assigned. The fraction both toolbits t_1 can place is thus given by half the workload fraction, $\frac{5}{12}$. Hence, assigning toolbit t_1 is still better. However, the third time around assigning toolbit t_2 wins because the fraction of t_1 reduced to $\frac{5}{18}$.

3.3 The genetic algorithm

A Genetic Algorithm (GA for short) is a heuristic algorithm for solving optimization problems, that is problems where the goal is to find a solution that minimizes some cost-function. It works by evolving over time a large collection of possible (valid) solutions. These solutions are encoded as strings, which are called *chromosomes*. The characters in this string are called *genes*, and they represent different "parts" of the solution. Chromosomes evolve over time by *crossover* and *mutation*, and the fittest chromosomes survive. Once a stop critereon is met, the genetic algorithm terminates and the fittest solution found is reported. Pseudo code of a genetic algorithm is given in Algorithm 1.

Algorithm 1 Genetic Algorithm

```
population ← InitialPopulation
repeat
    evaluate obj of each chromosome in population
    population ← Crossover(population) ∪ Mutate(population)
until stopping criterion is met
return chromosome found with best obj
```

A chromosome should encode a solution to the problem one is solving. At the end of a GA the best chromosome is reported, and thus a matching solution is found. A *population* is a set of chromosomes. At the start of the algorithm this set is filled using an algorithm InitialPopulation. In each stage of the loop a new population is created out of the previous one. A CrossOver and Mutate operation are to be defined that create new chromosomes out of the existing population. The CrossOver operation does so by pairing two existing chromosomes. The Mutate operation does so by taking an existing chromosome and changing it slightly. Candidate chromosomes that are used by these operators are normally the chromosomes that performed well. To check the performance of a chromosome an algorithm Obj should be defined. Let us now see how these parameters are chosen in the *Optimizer*.

- A chromosome normally represents a solution to the problem. A chromosome for a solution to the Traveling Salseman Problem —where the shortest tour through a number of cities is to be found— could for example encode the order in which the cities are visited. The chromosome that is used by the *Optimizer* however, does not encode a solution to the problem directly. Rather it contains all sorts of weights and hints that are used to influence heuristics used by an algorithm that computes a solution. In this way, the chromosome *does* influence the solution, but very indirect.

  The algorithm that decodes a chromosome into a solution is described in Section 3.4.

- The performance Obj of a single chromosome is defined to be the cycle time of the solution encoded by that chromosome.

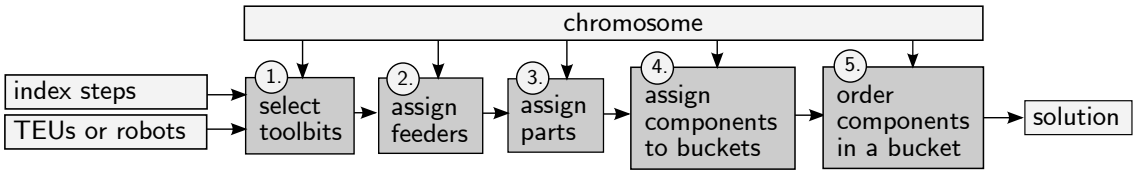

Figure 3.1: A high level schema of the five steps performed by the GA algorithm to decode a chromosome into a solution.

- The algorithm that defines how an initial population is constructed and how a new population is created follows a strategy called *CHC* [1]. This strategy makes no use of mutations. The new population constructed by this strategy always contains the fittest chromosomes found in the previous population. But on top of that it will produce chromosomes using crossover that are maximally different from their parents. The highly disruptive crossover mutation can provide a more effective search according to [1]. The initial population is taken randomly.
- When does the GA algorithm terminate? The *Optimizer* itself terminates after some time has passed. The *Optimizer* software is normally kept running for five minutes. The time it takes to evaluate the performance OBJ of a chromosome is what takes up the most time. On a typical computer for a typical problem the *Optimizer* is solving a single evaluation takes takes about ± 0.3 seconds. Thus there is normally time to evaluate ± 1.000 chromosomes. As each generation has a population of 50, ± 200 generations will approximately be evaluated in total.

3.4 Algorithm to obtain solution represented by chromosome

Recall that before the GA algorithm started, two steps were already performed. In these steps the index steps were computed and the TEU of each robot was filled completely with toolbits. That information, in combination with the chromosome, will form the input of the algorithm described in this section. The algorithm consists of five steps that are performed in sequence. After the last step has finished a solution is found. A schema showing this is given in Figure 3.1. Let us now briefly describe these five steps.

1. The TEU, as generated in the previous step, contains generally too many toolbits. Therefore in this step a subset of the toolbits in the TEU is selected. These are the toolbits that may later on actually be used to place components with.
2. Feeders must exist on the feeder bank before parts can be allocated to it. In this step therefore feeders are allocated to the feeder banks.
3. At this point we know exactly what feeders are present on the feeder banks. Therefore we know precisely what parts can actually be placed on the feeder bank. In step 3 parts are assigned to the lanes of the feeders assigned in the previous step. Multiple parts of the same type may be assigned if the algorithm believes this will be useful.
4. The index steps and both the toolbits and parts that a robot can use are all known at this point. Using this information we can compute the reachable buckets of a component. In step 6 each component now gets assigned to one such bucket. Now the algorithm may thus have decided that a component is going to be placed by a specific robot and index step. It is however not clear what toolbit is going to be used to do so, as multiple toolbits may be compatible with the component's part. Therefore a toolbit is now chosen that is going to place this component.

5. No attention is given to the order in which components in the previous step are placed. The order in which the components are going to be placed strongly influences the number of toolbit exchanges that are going to be performed. In this step the order in which components are placed is changed with the goal of minimizing the toolbit exchanges.

Step 1, 4 and 5 are related to toolbits. Therefore, these steps are explained in more detail in the following sections. We noted in the previous section that a chromosome does not directly encode a solution. In fact, we explained the algorithm for decoding a solution without speaking about the chromosome at all. The chromosome does however influence the solution, a section is devoted to these influences.

3.4.1 Step 1: Select workable TEU

The toolbits in the TEU have been set already before the GA started. However not all of these toolbits may be used in the solution obtained by this algorithm. Let us call the subset of toolbits out of the TEU of a robot that may actually be used the *workable TEU*. In this step the workable TEU is computed for each robot.

The first step of the algorithm is very simple; a single toolbit is assigned to the workable TEU of each robot. To decide how useful assigning a toolbit is, first some characteristics are computed. The most important characteristic is the number of toolbits that the algorithm expects to need of that kind. Now iteratively a toolbit is selected that is expected to be most useful. For such toolbit a corresponding robot is selected that matches best. Obviously this must be a robot that contains this toolbit in its TEU. This toolbit is now assigned to the workable TEU of this robot.

At this moment each robot has exactly one toolbit in its workable TEU. The TEU of the robot contains eight toolbits however. In this next stage the number of toolbits in the TEU may now be extended. This is done using a list of *rare* toolbits. A toolbit is labeled rare if it is the only toolbit capable of placing a specific part, and only few components of that part need to be placed on this panel. If the toolbit assigned to a robot is contained in this list of rare toolbits then all seven other toolbits from this robot's TEU are assigned to the workable TEU of this robot as well. Thus that robot may now use all eight toolbits for placing components, instead of the single toolbit.

3.4.2 Step 4: Assign components to buckets

In this step the components on the panel get distributed over the buckets. Furthermore for each component the toolbit that is going to place it is decided upon.

A component can be placed by a robot if the robot has the component's part on its feeder bank, a toolbit is found in the *workable TEU* that is compatible with that part, and the component is assigned to an allowed bucket of this robot. Because the contents of the part bank and TEU of each robot, and the index steps are known, we know now precisely where we can place a component.

Much like in previous steps iteratively a component is selected, and that component is then assigned to the bucket that matches best. This must be a a bucket that can actually place the component, as described above. An important reason to assign a component to a bucket is that the component can be placed using one of the toolbits that is already needed by that bucket. This is the case if one of the already assigned components in that bucket is placed using a toolbit that is also compatible with this component. In this case that toolbit is also used to place the newly assigned component. It can however be the case that such a bucket does not exist. In that case another bucket is chosen and a toolbit out of the workable TEU of the robot corresponding to that bucket must be chosen instead. The most suitable toolbit is given by the one that is compatible with the most parts of the feeder bank of this robot. Once all components are distributed the algorithm continues.

3.4.3 Step 5: Ordering the components in a bucket

The components have now been distributed over the buckets, and the toolbits that are going to be used are selected. The order in which the components are ultimately going to be placed is yet

unknown however. By smartly ordering the components faster cycle times can be achieved. In particular, toolbit exchanges can be avoided. Hence groups of components are formed that can be placed by the same toolbit. Components in such group are placed consecutively so no toolbit exchange is necessary in between.

3.4.4 Chromosome's influence on solution

There are four genes of the chromosome that are relevant for the toolbit exchanges. Next we discuss these genes. For reference the actual names used in the source code of the Optimizer are used to represent them.

UseCompleteToolbitExchangeUnitContent. This gene represent a boolean. If that gene has value "true", the workable TEU chosen in Step 1 will contain the entire TEU for all robots. In that case the workable TEU for each robot will contain eight toolbits. If the gene has the value "false" then the method as described in Step 1 is used.

FudgeFactor. Recall the list of rare toolbits that is constructed in Step 1. If a toolbit is on that list, the workable TEU will be the entire TEU of the robot. The number of toolbits in this list will be increased by the number represented by this gene. This is a number between minus two and one.

If this number is for example minus one, one rare toolbit will be dropped from the list. If a positive number is given then an extra toolbit is added to the list, thus increasing the likelyhood that the workable TEU of a robot will contain all toolbits in its TEU. This extra toolbit will be the toolbit that is *closest* to being rare.

Weight. For each toolbit type there is a gene in the chromosome that encodes the weight of that toolbit. This weight is simply used to increase or decrease the likeliness that a toolbit will be selected in Step 1.

TargetNumberOfToolbitsToBeAssignedAddition. For each toolbit type there is a gene in the chromosome that encodes a number. Choosing the most interesting toolbit in Step 1 is based on the number of toolbits the algorithm expects to use. This number is simply increased by the number encoded in the chromosome. This is number between zero and three.

Chapter 4

A new algorithm for pre-assigning toolbits

From Chapter 3 one can learn that the toolbits that make it into the TEU of a robot are computed by the *Optimizer* only once, at the start. This limits the solution space. As the content of the TEU is based on a simple heuristic, an interesting part of the solution space could well be neglected. Either the algorithm should be adapted so the contents of the TEU can change during the algorithm, or more thought should be put into computing a TEU that is really useful.

Halfway the algorithm the workable TEU is constructed. There is a rather harsh distinction in the number of toolbits in the workable TEU: it contains either a single toolbit or the entire TEU. Depending on the components that are placed in a robot, the number of toolbits actually used can lie anywhere between zero and eight. The algorithm for distributing components is not very targeted at minimizing the number of toolbits used by a robot, more toolbit exchanges can ultimately occur then strictly necessary. The other way around, sometimes fewer toolbit exchanges can be incurred if one uses more toolbits, rather than less. As this algorithms does not look at that opportunity, additional toolbit exchanges are believed to occur in general.

The genetic algorithm as used by the *Optimizer* is not a traditional implementation of a genetic algorithm because a solution is not directly encoded by a chromosome. This makes it unclear how precisely the solution space is traversed. Due to this, extending or modifying the structure of the chromosome or internals of the genetic algorithm seems no good option.

The remainder of this thesis will focus on computing an alternative TEU assignment for all robots. Remember that the *Optimizer* software allows one to preassign toolbits to the TEU of a robot. These preassigned toolbits are guaranteed to end up in the TEU. Thus a TEU can be fed into the existing *Optimizer*. The advantage of this is that the algorithm for computing the pre-assignment need not take all details of the machine into account, but that (by feeding its output into the Optimer) it can still be used to generate placement programs that can actually be used in a real-world setting.

We noted earlier that the TEU is currently computed based on some simple heuristics. To make sure a feasible solution is found, the TEU is always filled completely with toolbits. Instead, the alternative TEU should only add toolbits to the TEU that it *believes* are going to be useful for the *Optimizer* to get a good solution. This alternative TEU thus guides the *Optimizer* into a solution direction.

All toolbits in the alternative TEU are thus placed there with the hope the *Optimizer* will use them. The hope is that the workable TEU computed by the *Optimizer* will contain the entire TEU for all robots. Luckily, the *UseCompleteToolbitExchangeUnit* gene exists and does precisely that. One can safely assume that several chromosomes will be evaluated having this boolean set to "true".

As the alternative TEU contains only toolbits that are thought to be necessary, it will generally contain fewer toolbits. Because there are fewer toolbits the step of the algorithm that distributes

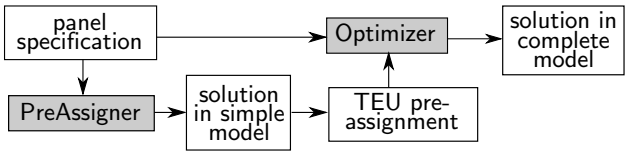

Figure 4.1: Schema showing how *Optimizer* and *PreAssigner* software are combined.

the components will hopefully make fewer errors.

When a pre-assignment is given that does not fill the TEU itself completely, the *Optimizer* will fill the TEU up completely with additional toolbits. This is a problem, but can easily be mitigated by adding dummy toolbits to the TEU pre-assignment that are of no use at all.

How do we generate this TEU? If one thinks about it, actually all aspects of the problem need to be looked at to do so. Whether placing a toolbit in the TEU of a robot is a good idea depends among others on the inventory of that toolbit type, the available parts on the feeder bank, the components that are reachable by the robot that it is allocated to and also the other toolbits in the TEU. The alternative TEU is thus essentially part of a complete solution, that one suspects has a good cycle time.

Thus to obtain such TEU, a complete solution to the problem needs to be generated. Only the TEU of that solution is then taken and inserted into the *Optimizer*. Because only the TEU of the solution is relevant, that part of the solution should match the original problem description. However, other parts of the problem can be simplified were needed. Let us call the algorithm that is going to compute a TEU pre-assignment the *PreAssigner*. A schema of how an improved solution to the original problem is hopefully obtained is given in Figure 4.1.

4.1 Simplified problem

The original problem is very complex and has many features and constraints. One can not expect to come up with an algorithm that produces a complete solution to the original problem in the scope of a single master's thesis project. In order to cope with the complexity, first a simplified model is introduced. As noted in the previous section the only requirement on this simple model is that the toolbits and TEUs should remain identical to the original model, because this is what we are going to preassign. In addition, the simplified model should be close enough to the original problem that the computed pre-assignment is not only good in the simplified model, but that it also leads to good results when used by the *Optimizer* to solve the original problem.

The model that is used by the *PreAssigner* is different from the complete model as introduced in Chapter 2. The model used by the *PreAssigner* is obtained by imposing constraints on the original model or by simply removing concepts:

- The *Optimizer* can generate placement programs for multiple panel specifications that share the machine configuration and setup. The *PreAssigner* will not be able to do so, and is constrained to a single panel specification.

- Each segment has a feeder bank in the original model. The notion of a feeder bank is dropped altogether in the simple model. Instead each robot has a *part bank*. Parts can be assigned directly to the part bank of a robot. A robot can simply use a part if it is assigned to its part bank. The notions of feeders and lanes are now useless and are thus dropped from the simple model. Contrary to the feeder bank, parts do not have an order in the part bank. Similar to the feeder bank, the part bank will also have a number of slots. A part can cover one or more of these slots. A schema of this new model is given in Figure 4.2.

- A cycle in the original model contains a sequence of actions for each bucket. An action is for example placing a component or exchanging a toolbit. In the simplified model only a set of components that need to be placed is maintained per bucket. Actions like reading a fiducial

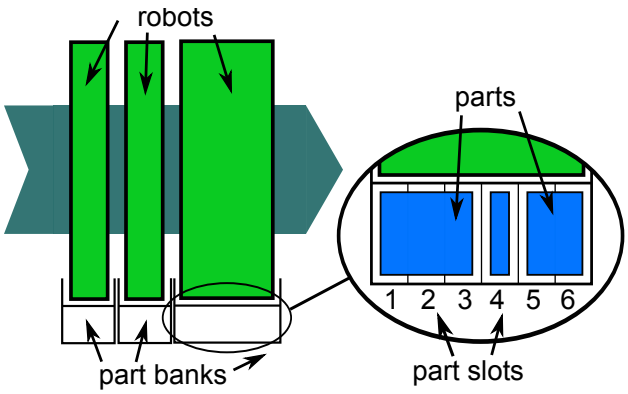

Figure 4.2: Visualization of new model. Note that a part bank is related to a single robot. The rightmost part bank is zoomed in at. One can see that this part bank has six slots, that are occupied by three parts.

and exchanging a toolbit are thus no longer part of this. Also the order in which to place components is removed. Thus in a placement program in the simple model the instructions corresponding to a bucket are given by a set of components that need to be placed somehow.

It seems strange that the time needed for toolbit exchanges is not part of our simplified model, since reducing the number of toolbit exchanges is the goal of our work. However, we *do* take toolbit exchanges into account when we evaluate the quality of a solution in our simplified model, as will be explained later.

- Index steps are part of a placement program. In the complete model they are thus part of the output generated by the *Optimizer*. In the simple model we assume that the index steps are known upfront. To be more precise: the index steps that are put into the simple model are generated using the *Optimizer* software.
- Recall that a return stroke can be performed by the transport beam and takes place at least once, at the end of a placement program. Generally speaking a return stroke takes more time then performing a toolbit exchange does. In the complete model a toolbit exchange can therefore be masked by a return stroke. Hence the time a toolbit exchange takes at the very end of a cycle does not count towards the cycle time. Masking toolbit exchanges is not possible in the simple model.
- Recall that a component may only be placed once a robot can guarantee the placement on the panel to occur within a certain precision in the complete model. As long as this precision is not met the robot must read more fiducials. Reading fiducials takes time however. This time is not accounted for in the simple model.
- We assume that the time it takes a robot to place a component is independent of the other components that are placed by that robot. This is also true in the complete model most of the time. However exceptions do exist. These exceptions are not covered in our simple model.

These simplifications lead to a new problem statement, which we call the *simplified problem* (or sometimes the *simplified model*). Next we describe the simplified problem more precisely. We describe the input and the required output.

The input. The input consist of the following elements.

- The panel is described by a set of components to be placed. For each component it is specified of what part type it is. For each part type the number of slots that it will occupy on a part bank is given. Furthermore an inventory is given for each part type.

- The machine configuration is given by the number of robots. For each robot the number of slots available on its part bank is known. Furthermore, the time it takes a robot to pick and place a component is given for the combination of each component and robot.

  The time that a robot takes to place a component is imported out of the *Optimizer* and called the *lower bound time*. This is the time that it takes the robot to place this component in the most ideal scenario.

- The number of index steps are given. Recall that the exact locations of the index steps determine the reachable buckets of each component. This information is taken as input, instead of the exact index step locations. Thus per component the reachable buckets of the component are given.

- The toolbit library is given. This defines what toolbit types are available. Furthermore the parts that a toolbit is compatible with are given. Furthermore an inventory is given on the toolbits in the library. This defines how many toolbits there are of each toolbit type.

The output. The output is simply given by a distribution of components over buckets. A solution can only be valid if each component is allocated to an allowed bucket.

The problem we wish to solve is now to compute for the given input a valid output that minimizes the time to assemble a panel. In other words, we wish to compute a valid output that minimizes the cycle time. (In fact, only the TEU assignment of the solution will be used as pre-assignment for the original *Optimizer*. The idea is that this pre-assignment leads to a good result when used by the *Optimizer*, because it leads to a low cycle time in our simplified model.)

The way the cycle time is computed is similar to the original problem. However, because our solution only consists of a distribution of components over buckets, the number of toolbit exchanges is not known. The number of toolbit exchanges can however be deduced from the component distribution using algorithms. Therefore the cycle time of our solution is defined as the output of the algorithm presented in Section 4.2.

Without going into the specifics of this algorithm one can already conclude that solving the problem described above is NP-hard.

Theorem 4.1.1. *The simplified problem is NP-hard*

Proof. Look at the situation where there is only a single index step and a single toolbit. Furthermore assume all parts can be handled by that one toolbit and both parts and toolbit have an infinite inventory. Now the task at hand is to distribute the components over the available robots. As there is only a single index the cycle time is determined by the robot that takes longest to process. Because only one toolbit exists no toolbit exchanges can take place and the cycle time is already well defined without knowing the specifics of the algorithm to be presented in the next section.

The scenario sketched here is identical to the minimum makespan problem. Minimum makespan is known to be NP-hard [2] therefore our simplified problem is NP-hard as well. □

4.2 Computing the cycle time in the simplified model

Recall that the cycle time is given by the sum of the time each index takes. The time an index takes is defined as the maximum time a robot takes processing that index. The time it takes a robot to process an index consists of two pieces: the time it takes to place all components and the time the robot is busy exchanging toolbits. The time it takes a robot to place a component in an index step is given in the input. Hence the time it takes that robot to place all components in an index step, which is given by the sum over all these times, can easily be computed. The time a robot is exchanging toolbits is given by the number of toolbit exchanges multiplied by the time a

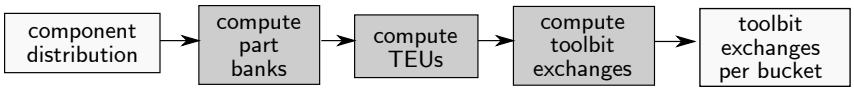

Figure 4.3: Schema depicting steps necessary to compute the number of toolbit exchanges per bucket based on a component distribution.

single toolbit exchanges takes. The time a single toolbit exchanges takes is constant. Computing the number of toolbit exchanges is less trivial, as discussed next.

In the complete model the number of toolbit exchanges does not only depend on the distribution of components, but also on the toolbits that are located in the TEU of each robot, and which toolbit is used to place what component. Because all these are part of a solution in the complete model, the number of toolbit exchanges can be computed. To be able to compute the toolbit exchanges in the simple model we also have to generate both a part bank and TEU for each robot.

As a first step, the part bank is computed for each robot based on the components that are placed by that robot. Next, the TEU of each robot is filled based on the parts that are on its part bank. As a last step the actual number of toolbit exchanges per bucket is computed based on both the part banks and TEUs of all robots. A schema depicting this is given in Figure 4.3. For the last step, computing the toolbit exchanges, two implementations are proposed: a fast but inaccurate one and an exact but slower one. In the following subsections all steps of the process are described.

It can be the case that one of the steps below fails to generate either a part bank or a TEU with whom all components can be placed. In that case a cycle time of INVALID is reported.

4.2.1 Filling the part bank

As a first step we are going to assign parts to part banks of robots, based on the component distribution as contained in a solution of the simple model. A part bank of a robot can trivially be filled based on the following observation: a component that resides in a bucket of that robot can only be placed by that robot if it has the component's part on its part bank. As a valid solution is only found if all components are placeable that part must thus be placed in the part bank of that robot.

Thus per robot we loop over all components that are contained in its buckets and add a part to the part bank as soon as we encounter a part that we did not encounter before.

If the sum of the sizes of the parts allocated to the part bank of a robot exceeds the size of that part bank, INVALID is reported. In this case a solution simply does not exist given the distrubtion of components.

4.2.2 TEU construction from part bank

The part bank of a robot will now contain a part if and only if a component of that part is assigned to a bucket of that robot. Now we are going to fill the TEU of each robot in such a way that all parts located in the part bank are indeed usable. Thus for each part in the part bank a compatible toolbit should be present in the TEU. Recall that toolbits can have an inventory, and thus can be assigned only a limited number of times. Thus assigning a toolbit to a robot can influence the feasibility or performance in another robot. Therefore the assignment of toolbits to TEUs cannot be done independently for each robot. The more toolbits are contained in a TEU the more likely it is that a toolbit exchange will happen. Therefore minimizing the number of toolbits across all TEUs is important.

Let n be the number of robots, and t be the number of toolbits. Recall that for each robot the list of parts in its part bank is known. We are now interested in an assignment of toolbits to robots that covers all parts. An assignment covers all parts if for each part a compatible toolbit is assigned. These assignment should be chosen in such a way that the sum of the toolbit exchanges over all robots is minimal. A visualization of this problem on a small input is given in Figure 4.4.

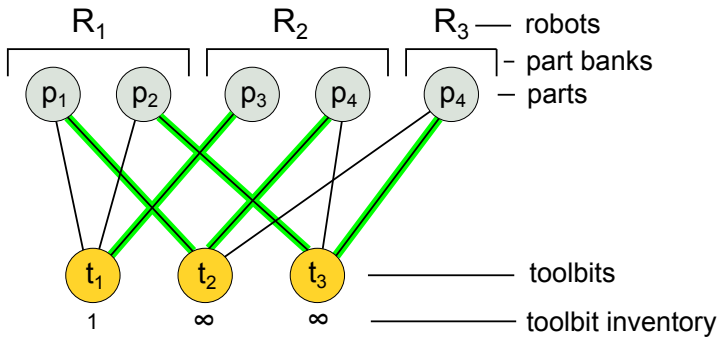

Figure 4.4: Example problem. Marked edges represent a valid solution assignment.

Theorem 4.2.1. *The toolbit-assignment problem defined above is NP-hard.*

Proof. Let there be an arbitrary instance of the *Set Cover* [3] problem. Assume an AX that is configured with only one robot, there is thus only one part bank. Now let there be a part on the part bank of this robot for every vertex in our Set Cover instance. Now introduce a toolbit for every set in the Set Cover instance. A toolbit is compatible with a part if and only if the set corresponding to this toolbit contains the vertex corresponding to that part. Because there is only one robot, we are interested in the assignment with the minimum number of assignments to this robot. That number is identical to the minimal number of sets found in the Set Cover problem.

We thus reduced Set Cover to our toolbit assignment problem. Because Set Cover is known to be NP-Hard [3], this will prove our toolbit assignment problem to be also NP-Hard. □

As this subproblem is NP-hard, one can safely presume no fast algorithm exists. Therefore instead an algorithm is proposed that does not guarantee the minimal number of assignments to be found. This algorithm is a greedy algorithm. The greedy choice is: make the assignment of toolbit to robot that covers the most parts that are yet uncovered. This greedy step is iteratively used until a valid solution is found, and then reported as the solution.

A problem with this algorithm is that iteratively using this greedy step can yield an infeasible solution, when a feasible solution does exists (this is the case in Figure 4.4 where assigning t_1 to R_1 will make p_3 of R_2 uncoverable). Because toolbit inventories are not tight in practise —usually there are more toolbits available then one could possibly need—, this greedy algorithm will usually report a solution if one exists. In these rare cases that the greedy algorithm does fail an extra algorithm is run to test if a feasible solution does indeed not exist. This algorithm simply tests all possible assignments of toolbits to robots (note that the actual implemented algorithm smartly combines these two algorithms).

If the algorithm finishes and has not found a TEU covering all parts then INVALID is reported. In this situation TEUs simply do not exist that cover all parts.

4.2.3 Computing toolbit exchanges

At this point we have a solution in the simple model, augmented with both a part bank and a TEU for each robot. Now enough information is available to actually compute the number of toolbit exchanges that are used in a single bucket.

Two methods are described that can be used to obtain a number of toolbit exchanges. A simple method is shown to estimate the number of toolbit exchanges and one is shown that can compute the minimum number of toolbit exchanges over all the buckets.

Estimating the number of toolbit exchanges

Assume that in a single bucket four components get placed of four different parts, and a minimum of two toolbits are necessary to cover the four parts. It is easy to see that to be able to place all

four components at least one toolbit exchange must be performed. As a more general rule: if the minimum number of toolbits to place all components in a bucket is n, then $\text{MAX}(n-1, 0)$ toolbit exchanges are needed at least.

So now how does one compute the minimum number of toolbits needed to place all components in that bucket? As the number of toolbits found in the TEU of a robot is limited by eight, brute-force suffices. This algorithm will iterate over all possible subsets that can be constructed using the toolbits in the TEU. Worst-case all $2^8 = 256$ possible subsets are tested. The subsets are iterated over such that smaller subsets are always considered before larger ones. If a subset already covers all parts than all subsets that contain this subset can be pruned.

Exact computation of the number of toolbit exchanges

The previous technique is an underestimation on the number of toolbit exchanges. It assumes that the first toolbit that is needed in a bucket is already attached to the nozzle of the placement head at the start of that bucket. This is however not always the case. For an input with n index steps each robot can incur at most n toolbit exchanges that are not accounted for in the previous technique. This is the case if a robot must exchange toolbit at the start of each index. If the AX is configured with m robots the previous method can make an error of at most nm toolbit exchanges.

Note that the choice for the toolbits to be used in a single bucket of a robot can influence the toolbits exchanges between index steps that occur elsewhere in this robot. Therefore, instead of the previous technique, one can not compute the number of toolbits needed in a single bucket in isolation: the toolbits used for all buckets of that robot will need to be computed at the same time.

The problem of computing the minimum number of toolbit exchanges for a given robot can be formulated as a graph problem. An example of a graph that is used to compute the minimal number of toolbit exchanges is given in Figure 4.5. Each vertex in the graph is associated with a toolbit, which represents the use of that toolbit by the robot at the beginning of an index step or at the end of an index step. An edge denotes a possible sequence of toolbit exchanges. The weight of an edge denotes the number of toolbit exchanges that occur between the source and target vertex.

Next we describe the construction of the graph in more details. First introduce a start vertex $v_{0,j}$ for every toolbit t_j in the TEU of the robot. Let us now encode the toolbit exchanges that occur strictly within an index. For an index i of this robot one can trivially find the parts that are necessary to place components assigned to i. Let S_i be the maximum set of subsets of toolbits in the TEU of this robot such that the toolbits in each subset cover the previously found parts. A set of toolbits covers a set of parts if for each part in the set a compatible toolbit exists in the set of toolbits. Introduce vertices $v_{2i+1,j}, v_{2i+2,j}$ for each toolbit t_j that occurs somewhere in the set S_i. These vertices represent respectively the toolbit that is allocated at the start and the end of index i. Introduce an edge between $v_{2i+1,j}$ and $v_{2i+2,k}$ if either

- $t_j = t_k$ and t_j is a singleton element in S_i, or
- $t_j \neq t_k$ and there is a subset of toolbits in S_i that contains both t_j and t_k.

The weight of an edge is given by the minimum number of toolbit exchanges necessary using these toolbits, and is, just as before, given by the number of toolbits minus 1.

Next we model the toolbit exchanges between index steps. Introduce an edge $(v_{2i,j}, v_{2i+1,k})$ if both vertices exist. The weight of the edge is zero if $t_j = t_k$ and one if $t_j \neq t_k$.

Now a graph is constructed for the problem input, how can one utilize it to find the minimum number of toobit exchanges? Well, the minimum number of toolbit exchanges is defined as the girth of the graph. The girth is defined as the length of a smallest cycle in the graph.

A cycle of minimum length among all cycles that pass through start toolbit t can be found using Dijkstra's algorithm [4]. Namely this cycle is given by the shortest path from start toolbit t

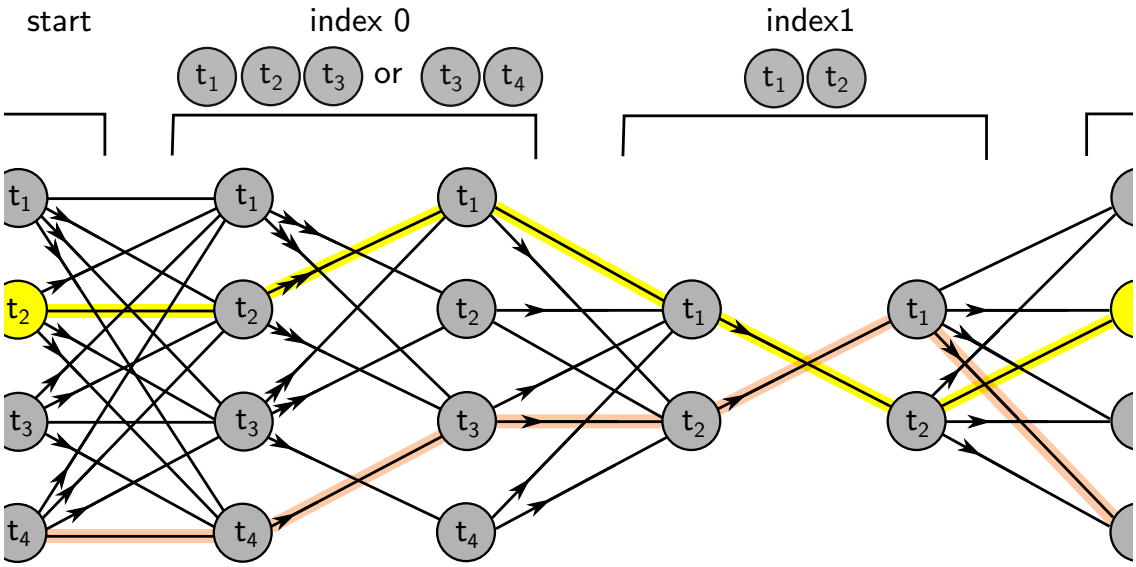

Figure 4.5: Example of a graph used for computing the minimum number of toolbit exchanges. The number of arrows on each edge represents the weight of that edge. Start nodes on left and right node represent identical nodes. The yellow marked cycle represents a minimal solution of three toolbit exchanges whereas the brown cycle represents a solution that yields four toolbit exchanges (as would be generated using the estimation method).

to itself. Dijkstra's can be used as all edges are non-negative. Dijkstra's algorithm can be run for every toolbit in the TEU. The smallest cycle found over all runs is now the smallest cycle among these that go through a start toolbit. However, by construction, a cycle in this graph will always contain precisely one of the start toolbits, and therefore the smallest cycle found is indeed a cycle of minimal length in the entire graph.

The algorithm will thus yield a cycle of minimum length. The length represents the minimum number of exchanges necessary for the given robot. One can find the exact places where the algorithm decides to exchange toolbit by looking at the edges having a strictly positive weight inside of the cycle. The algorithm should however report a number of toolbit exchanges per index. For index i this is found by the number of toolbit exchanges within this index, plus possibly a single additional toolbit exchange if the correct toolbit is not yet at the nozzle at the start of this index. In terms of the computed cycle this is given by the sum of the weight of the two edges in the cycle that have either vertex $v_{2i,\_}$ as source or target.

4.3 Our approach to solving the simplified problem

It is now clear how the cycle time can be obtained once a solution is found. But how is one going to find a solution to the problem? As we have seen earlier, computing an optimal solution is NP-hard. One can thus not hope to come up with a fast exact algorithm. That limits the possible approaches. Techniques that could possibly be used include: (integer) linear programming, genetic algorithms and simulated annealing.

- Many optimizing problems can be formulated as linear programs, that is, problems where a lineair objective function of a number of variables should be maximized (or minimized) subject to a number of linear constraints on the variables. The function that we want to optimize, however, is not a linear function. In fact, it is partially determined by the outcome of the algorithm for computing the minimum number of toolbit exchanges needed by a given solution. Hence, linear programming is not a viable approach for our problem.

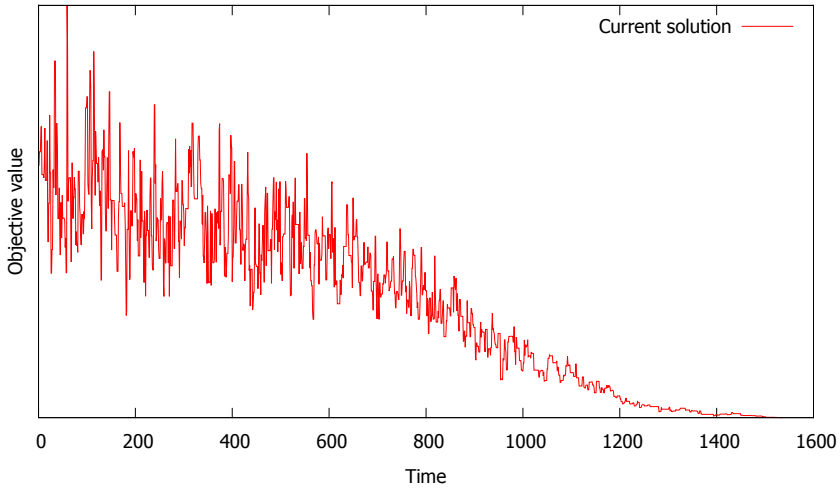

Figure 4.6: Graph depicting the value of the objective function of the single solution over the time of a Simulated Annealing algorithm.

- The theory behind the genetic algorithm was already explained earlier in Section 3.3. A genetic algorithm can be terminated at any time and will then yield the best solution found so far. However, finding a crossover operation for our problem that is not too disruptive turns out to be difficult. The most obvious crossover operations one can think about typically yield invalid solutions.
- Simulated annealing is much similar to a genetic algorithm. Also this algorithm can safely be stopped at any time. It differs from the genetic algorithm in that it has a candidate population of one. That single candidate is iteratively transformed by applying mutations on it. Mutators are operations that alter a given solution slightly. Defining mutators that yield valid solutions is a lot simpler than defining crossover operations that yield valid solutions.

Due to these reasons simulated annealing is chosen as our technique to overcome the NP hardness of the problem.

4.4 Simulated annealing

Simulated annealing (SA for short) is a method for approximating the global optimum of a given function in a large search space. It is a search method algorithm that iteratively chooses a new solution. The quality of a solution is given by an objective function Obj. All solutions having a better objective function value than the current one are accepted, while only some solutions are accepted that have a worse objective function value, based on a probabilistic criterion. This criterion is chosen so that the probability that worse solutions are accepted decreases as the algorithm continues to explore the solution space. New solutions are obtained from the current one by slightly mutating it.

Figure 4.6 shows an example of a graph depicting the value of the objective function over time of the solution maintained by a Simulated Annealing algorithm. One can see that the algorithm can freely accept solutions that are much worse at the start. However, as time passes, the algorithm is less likely to accept such solutions.

Algorithm 2 Simulated Annealing

```
T ← start temperature
solution ← an initial solution
repeat
    T ← NextTemperature(step)
    neighbor ← Mutate(solution)
    if Obj(neighbor) < Obj(solution) then
        solution ← neighbor
    else if P(Obj(neighbor) − Obj(solution), T) ≥ Rand() then
        solution ← neighbor
    end if
until stopping criterion is met
return solution
```

The pseudo code for a standard Simulated Annealing algorithm is given in Algorithm 2. One can see that an initial solution to the problem should be already known. The MUTATE method should be able to mutate an input solution and return a slightly different solution. Their implementations are obviously dependent on the problem at hand.

The variable T is known in the SA literature as the temperature. The value of T is a function of the time. The shape of that function is determined by the the start temperature and the implementation of NEXTTEMPERATURE. The temperature influences the probability P that a solution that has an objective function value that is worse than the current objective function is accepted. If the temperature is high this probability should be high, as the temperature approaches zero this probability should reduce to zero. If the initial temperature and the implementation of NEXTTEMPERATURE are chosen correctly the SA algorithm is known to be able to generate satisfactory results. The start temperature, NEXTTEMPERATURE and method P are independent of the problem. However in order to get good results from the SA algorithm these parameters have to be tuned specifically to the problem. Tuning these parameters is mentioned as future work in Chapter 6.

Now how can we use SA to solve our problem? Therefore we have to fill in all parameters discussed above. These parameters are chosen in the following way:

- A solution is simply a solution to our simple model. Thus a solution is a distribution of components over buckets.
- The value of the objective function of a solution in the simple model is obtained by using our algorithm for computing the cycle time of a solution, as presented in Section 4.2.
- The *start temperature* and the *next temperature* method decide how the temperature cools during the algorithm. The start temperature is chosen as the cycle time of the solution that is initially computed. The next temperature method describes an inverse exponential function. The exponent is chosen in such a way that at the end of the algorithm the temperature has dropped to 0.001.
- The probability function that is used is taken from the paper introducing Simulated Annealing [5]. A solution *neighbor* is accepted over *solution* at temperature T with a probability of:
$$\exp\left(\frac{\text{Obj}(current) - \text{Obj}(neighbor)}{T}\right)$$
- The algorithm terminates after 250.000 evaluations have been performed.
- An algorithm to compute an initial solution is given in Section 4.5.
- A set of mutators is given in Section 4.6. How a mutator is chosen from this set is described in Section 4.7.

- The SA algorithm itself is adapted such that it not always accepts new solutions that yield an equal objective function value. This adaption to the algorithm is explained in more detail in Section 4.8.

4.5 Computing an initial solution

A valid initial solution needs to be computed before the SA algorithm can actually start working. An algorithm that produces such solution is given in this section. The performance of the solutions generated using this algorithm is known to be poor. It is up to our SA algorithm to transform it into a solution that performs well.

To produce a valid solution, each component needs to be allocated to a bucket that can actually place it. However, we also want to distribute the components in such a way that it is unlikely that the algorithm that evaluates solutions in our simple model reports INVALID. The later happens if no valid part bank can be found. To maximize our chances that a part bank for our solution exists we try to place components of the same part on the same robot as much as possible.

The algorithm iterates over all parts. For each part it will find the robot that can place the largest number of components of that part. These components are now distributed over the buckets of this robot. If not all components can get placed by this robot, the remaining components are distributed over a second robot. This continues until all components have been placed.

In theory this algorithm can produce a solution that is INVALID when a valid solution exists. This is for example the case if there is only just enough room to allocate each part once. In that case one should carefully think about where to place each part. However this is not likely to happen in practise.

4.6 Mutating the solution

In each iteration the SA algorithm will mutate the current solution using a mutator. In this section the five available mutators are described.

4.6.1 Move component

This mutator will simply move a component from one bucket to another. The only requirement is that the robot of the bucket that receives the component already places another component of the same part in one of its buckets and the receiving bucket is an allowed bucket of the component. An example of applying such mutator is given in Figure 4.7.

4.6.2 Move part

Applying this mutator will effectively move a part from a part bank of one robot to the part bank of another robot that does not yet have this part. This is done in two steps. First all components of the part are taken out of the buckets of the source robot. Then the components are distributed over the buckets of the receiving robot. This mutator is only allowed if the receiving robot does not already place a component of this part and there is enough space on the part bank of the receiving robot to receive this part. Furthermore all components that are placed in buckets of the source robot should be placeable in buckets of the receiving robot. An example of applying such mutator is given in Figure 4.8.

4.6.3 Duplicate part

This mutator will possibly reduce the time a robot is busy placing components of a part. It does so by moving half of the components of that part to another robot. First, half of the components of this part are taken from each bucket of the source robot. Next these components that got removed are distributed over the buckets of the other robot. This mutator is only allowed if the other robot

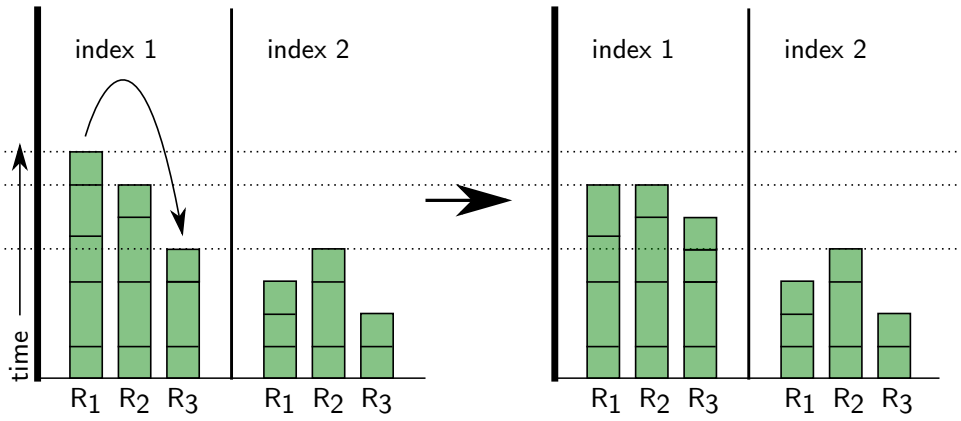

Figure 4.7: Effect of applying a move-component mutator. Each box represents the time it takes to place a single component. A single component is moved from robot R_1 to R_3 within index one. One can see this mutator reduces the time needed for index one, while keeping the time index two takes. The cycle time is thus reduced.

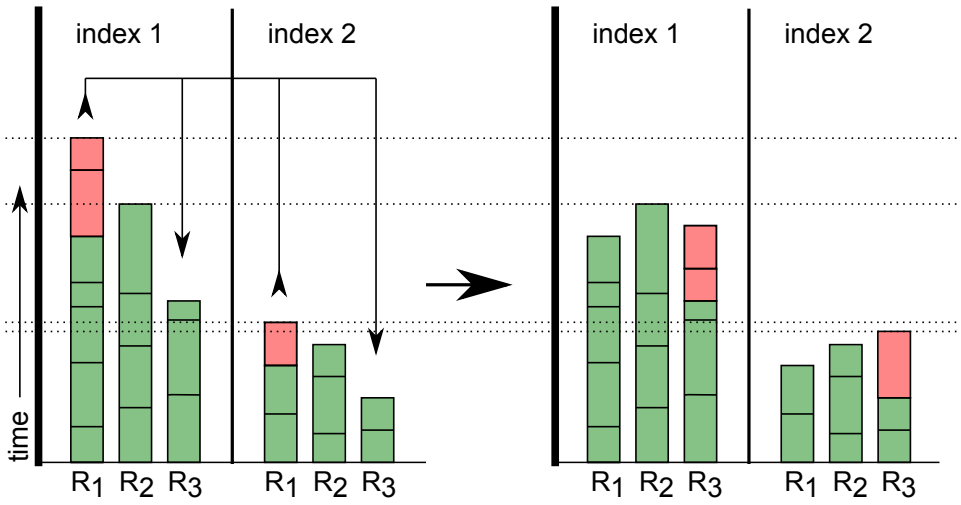

Figure 4.8: Effect of applying a move-part mutator. A box represents the time it takes to place a single component. Boxes colored in the same way represent components of the same part. Components of the red part are moved from robot R_1 to R_3. One can see that this mutator reduces the time each index takes.

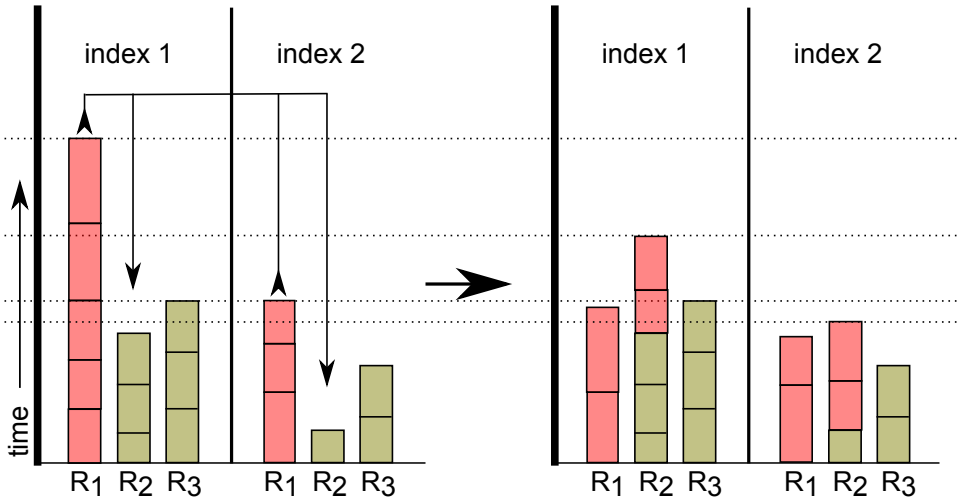

Figure 4.9: Effect of applying a duplicate-part mutator. A box represents the time it takes to place a single component. Boxes colored in the same way represent components of the same part. The red part on robot R_1 is duplicated to robot R_2. The effect is that half of the components previously placed by robot R_1 are now placed by R_2. This reduces the time both indices take.

does not already place a component of this part and the other robot has enough space on the part bank of the receiving robot to receive this part. Also the inventory should allow placing an extra part of this type. Furthermore half of the components that are placed in buckets of the source robot should be placeable in buckets of the other robot. An example of applying such mutator is given in Figure 4.9.

4.6.4 Merge part

A duplicate-part mutator can be reverted by applying the merge-part mutator. All components of the same part will be moved from one robot to another robot. The requirement is that the other robot should already be placing components of that part type and all components taken from the source robot are placeable by buckets of the receiving robot. An example of applying such mutator is given in Figure 4.10.

4.6.5 Swap part

This mutator is similar to the move-part mutator. The move part mutator moves a part from one robot to another, where the receiving robot should not already place a component of that part. The swap mutator will at the same time move a part from that receiving robot back to the source robot. Also this source robot should not already place a component of that part. The requirement is that the part banks of both robots should not be overloaded after swapping the parts. An example of applying such mutator is given in Figure 4.11.

4.7 Strategy for selecting mutator

In each iteration of the SA algorithm a mutator is applied to the current solution. There are five mutators available, so how does one choose which mutator to apply? A weight is attached to each of the mutators to do so. The relative weight determines the probability that this mutator is applied to the current solution.

We configured this strategy with weights. These weights were chosen because they yielded good results during the construction of the *PreAssigner*. No experiments have been performed

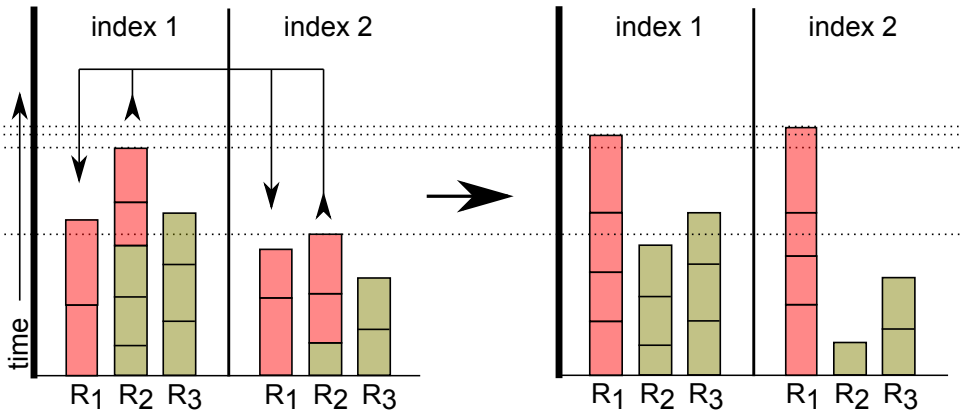

Figure 4.10: Effect of applying a merge-part mutator. A box represents the time it takes to place a single component. Boxes colored in the same way represent components of the same part. The red part on robot R_2 is merged with robot R_1. All components placed by robot R_2 are thus distributed over robot R_1. The time needed increases in both indices, and the cycle time increases.

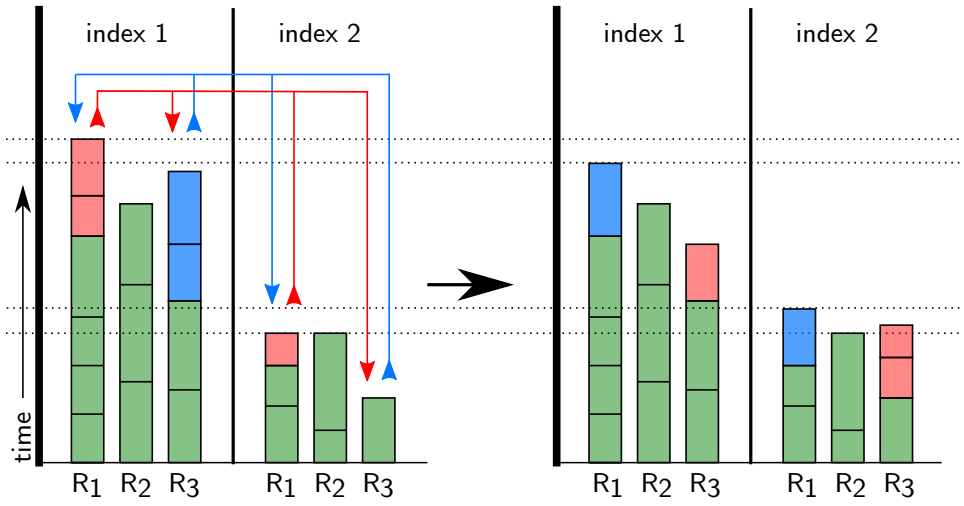

Figure 4.11: Effect of applying a swap-part mutator. The red part on robot R_1 is swapped with the blue part on robot R_3. A box represents the time it takes to place a single component. Boxes colored in the same way represent components of the same part. Note how the components of the blue part are distributed evenly after the mutation. Index one gets faster whereas index three gets slower. Overall the cycle time gets worse.

Mutator	Probability
Move component	0.45
Move part	0.27
Duplicate part	0.09
Merge part	0.05
Swap part	0.14

Table 4.1: The probability that a mutator is chosen by the mutator strategy.

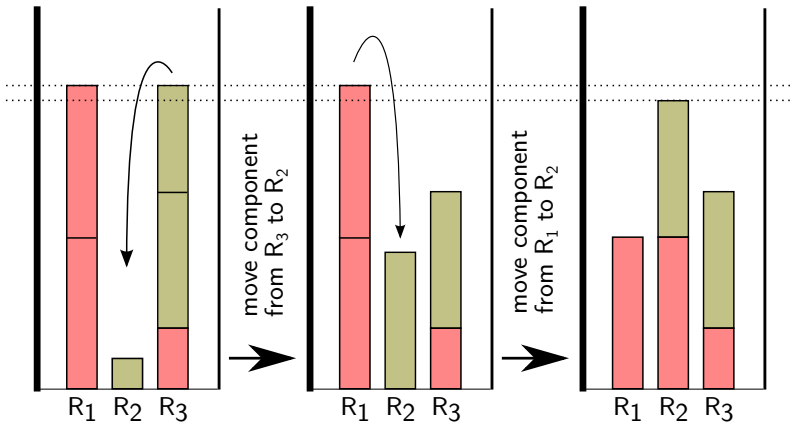

Figure 4.12: Serie of mutators of whom the first does not improve the cycle time, but is beneficiary. A box represents the time it takes to place a single component. Boxes colored in the same way represent components of the same part.

however to find the weights that achieve the best results. Table 4.1 shows the probability that each of the mutators is chosen by this strategy.

4.8 Balance objective function

There are many situations where a mutator will influence the solution but not the cycle time. Recall that the time an index step takes is defined as the maximum over all the buckets in that index. Thus the cycle time changes only if the maximum changes. In the default SA algorithm solutions having identical cycle times are always accepted. In our case, it seems better to use a more refined strategy.

Take a look at the balance of a single index of a solution given in Figure 4.12. Inside of this index step there are two robots that have the exact same value, which is the maximum among robots. As robot R_2 has almost no work scheduled, one would think that a better solution should exist. There is no mutator available, however, that can reduce the cycle time directly. A first step towards improving the cycle time is obviously offloading work from either robot R_1 or robot R_3 to robot R_2. This mutator yields an identical cycle time. Once such mutator has been applied, mutators can be used that do improve the cycle time directly.

The first mutator that offloaded work can be called an improvement even though it did not change the cycle time, because it improved the balance of the solution. Using this information the SA algorithm is now altered. If a new solution has an equal cycle time then the solution will now only be accepted if it improves the balance, or with a constant probability if it did not improve the balance. This constant is taken as 0.2.

A solution improves the balance if the new solution has a lower *balance metric*. The balance metric is defined as the sum of the squared times each bucket takes. This metric will report lower values if the balance among buckets is better distributed (for example $3^2 + 1^2 > 2^2 + 2^2$)

Chapter 5

Experimental evaluation of the new algorithm

5.1 Setup

The previous sections have introduced both the *Optimizer*, the software currently used by Assembléon, and the *PreAssigner* that has been developed for this thesis. The *Optimizer* produces a solution for the complete model, whereas the *PreAssigner* produces a solution for a model. These two solutions can be located in the top-left and bottom-right quadrant of Figure 5.1 respectively.

The TEUs can be extracted from a solution in the simple model. This TEU configuration can be fed into the *Optimizer*. By doing this for a solution generated by the *PreAssigner*, a solution in the complete model is obtained. A solution obtained in such way can be found in the top-right quadrant of Figure 5.1.

The simple model is obviously a simplification of the complete model. Therefore any solution in the complete model can be transformed to a solution in the simple model. We developed a tool such that any solution obtained using the *Optimizer* can be translated into a solution in the simple model. A solution obtained in this way can be found in the lower-left quadrant of Figure 5.1.

There are two comparisons that are of great interest to us. The first is how a solution that is generated by the *PreAssigner* performs compared to a solution as imported from the *Optimizer* (labeled "1" in Figure 5.1). This can show how good our generated solution is in the simple model, and how good potentially the TEUs of that solution are. The second comparison of interest is how well our *PreAssigner* performs in the real-world. This can be shown by comparing a solution obtained by the *Optimizer* with and without a pre-assignment (labeled "2" in Figure 5.1).

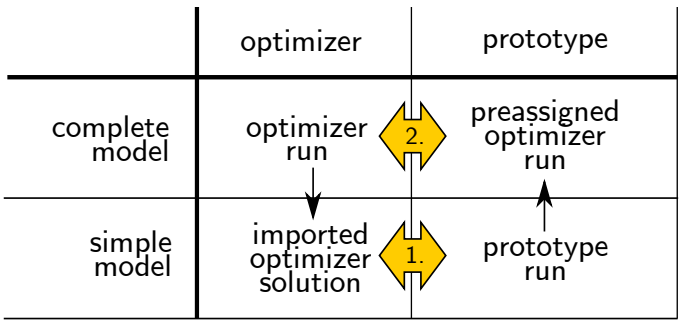

Figure 5.1: The four relevant quadrants for our experiments.

Test set. Assembléon has a test-suite for its *Optimizer* software. Among the tests in this suite, actual problem instances can be found. This set consists of problems taken from actual customers. These customer problems use very diverse configurations and make for an ideal test set for our purposes.

The *PreAssigner* is bound to some constraints. Therefore not all problem instances in this set can be used directly. The following steps are performed upon the test set to prepare it.

- Problems can be meant for multiple machines. For example, a problem can state how two AX machines standing in one assembly line, or even a AX and another pick & place machine, can best assemble a single panel. Our *PreAssigner* only works for AX machines. Problems for multiple machines are thus split into multiple problems for a single machine. Now only the problems for an AX are kept.

- A problem may also contain multiple panel specifications. A constraint on the *PreAssigner* is that the input may only contain a single panel specification. The problem is therefore split into multiple problems, each of only single panel specification. Each of these new problems is inserted into our test set.

- Solutions of the *PreAssigner* are obtained in the complete model via pre-assignments. It can however also be the case that problems in the test suite already have a toolbit pre-assignment configured. This conflict is resolved by simply removing all pre-assignments present in the test suite. Also, the feeder and part pre-assignments are dropped.

After these steps have been performed a total of 145 problem instances is found. These make up our test set.

Runs. Let us now describe how solutions are generated from all four quadrants.

$[opt_{complete}]$ A run of the complete test-set by the original *Optimizer*. Because the *Optimizer* is deterministic it is only run once. Each instance is kept running for 5 minutes. The solutions produced by the *Optimizer* are obviously in the complete model.

$[opt_{simple}]$ The solutions as found in $[opt_{complete}]$ are now transformed into solutions for the simple model. To compute the placement time of the transformed solution, the algorithm that computes the minimum number of toolbit exchanges is used in the objective function. Transforming the solution into the simple model is deterministic and occurs thus only once.

$[pro_{exact,simple}]$ The *PreAssigner* is run for every problem in the test-set. Because the *PreAssigner* is non-deterministic each problem is run three times, the solution yielding the best cycle time is chosen. The time the *PreAssigner* takes to process a single problem instance depends on the complexity of the job, but takes in the order of five minutes. The exact toolbit computation method is used in the algorithm that computes the cycle time. The rest of the SA algorithm is configured as discussed in Chapter 4.

$[pro_{exact,complete}]$ For each solution found by $[pro_{exact,simple}]$ the TEUs are extracted. Solutions in the complete model are found by running the *Optimizer* for each solution with the extracted TEUs preassigned.

$[pro_{estimate,simple}]$ These solutions are obtained similar as described in $[pro_{exact,simple}]$. However instead of the exact method to compute toolbits, the estimation method is used.

$[pro_{estimate,complete}]$ These solutions are obtained similar as described in $[pro_{exact,complete}]$. However the solutions in the simple model as given by $[pro_{estimate,simple}]$ are used instead.

5.2 Experimental evaluation

5.2.1 The effect of the mutators

Our SA algorithm uses a total of five mutators. What is the effect of each mutator, and are all these mutators really necessary? We test this by running the *PreAssigner* for each possible subset over these mutators. For these five mutators there are $2^5 - 1 = 31$ possible non-empty subsets. In such a run the *PreAssigner* may only use mutators out of the subset. The mutator strategy is used for this, and an equal weight is assigned to each mutator. Furthermore the *PreAssigner* is run using this subset for five different problems. These five problems are chosen in such a way that they are believed to be representative for the complete test-set. Each problem is run three times and the best solution is used. The time reported for a subset of mutators is the average cycle time achieved on these five jobs.

The results of this experiment can be seen in Table 5.1. One can see, as expected, that the set containing all the mutators performs best. The set containing all mutators excluding the duplicate part mutator ranks only 13th. The twelve subsets performing better all make use of the duplicate part mutator, apparantly duplicating parts is crucial to obtain well performing solutions. The subset containing all mutators excluding the merge part mutator scores almost as well as the subset containing all mutators. Apparantly the merge part mutator is not really crucial. A sequence of move component steps can imitate the merge part mutator equally well.

5.2.2 Effect of the balance objective function

The SA algorithm is adapted so that it accepts solutions yielding an identical cycle time only if the balance of the solution improves. We are interested if this adaption to the SA algorithm yields an improvement over the SA algorithm without the adaption. We test this by running the *PreAssigner* once with and once without this addition. A run is done over the complete test set, and each problem in the set is run three times. The results can simply be described by saying that it makes no siginificant change if this feature is used or not.

5.2.3 Convergence rate of the SA algorithm

Recall the graph given earlier showing the value of the objective function of the current solution in the SA algorithm over time. This graph shows the behaviour one expects from a SA algorithm in two ways. Firstly, the cycle time reduces to a point where it converges. Secondly, the variance in cycle time reduces as the time progesses. We hope that such effects are also present in our algorithm. To verify this we look at graphs for solutions in $[pro_{exact,simple}]$ that show the cycle time of the solution currently used by our *PreAssigner* over time.

A representative set of four graphs is given in Figure 5.2. Visual inspection of these graphs learns us that the cycle time indeed decreases over time. At the end of the *PreAssigner* the solution is clearly converged. The algorithm could have been terminated earlier. Also one can clearly see that the variance in cycle time decreases over time. At the start of the algorithm the cycle time jumps up and down siginificantly, whereas at the end the cycle hardly changes. This is exactly the behaviour one would hope to get from a SA algorithm.

5.2.4 Variance of the *PreAssigner*

The *PreAssigner* is non-deterministic. It is interesting to see how much the results vary if the same problem instance is run more than once. To test this we are going to run the *PreAssigner* on five problem instances a total of 20 times. These five problems are chosen in such a way that they are believed to be representative for the complete test-set. The results of this experiment can be seen in Figure 5.3. The horizontal axis contains the number of times the *PreAssigner* is run. A line is shown for each problem. Each line represents the relative improvement of the best solution, found after having runned the *PreAssigner* a given number of times, over the solution found in

Move component	Move part	Duplicate part	Merge part	Swap part	Cycle time
√	√	√	√	√	25.99
√	√	√		√	26.03
√		√		√	26.30
√		√	√	√	26.39
		√		√	29.82
	√	√		√	30.47
√		√			31.06
√		√	√		31.08
		√	√	√	31.12
√	√	√			31.14
	√	√	√	√	31.98
√	√	√	√		32.61
√	√		√	√	46.22
√	√			√	46.35
	√		√	√	47.34
	√			√	48.00
√	√				49.12
		√			49.49
		√	√		49.89
	√	√			50.36
√	√		√		50.61
√				√	51.76
	√		√		52.04
	√				52.17
	√	√	√		52.25
√			√	√	52.91
				√	53.55
			√	√	53.83
√					91.38
√			√		91.68
			√		94.73

Table 5.1: The average cycle times that are obtained by running the SA algorithm with only using the checked mutators.

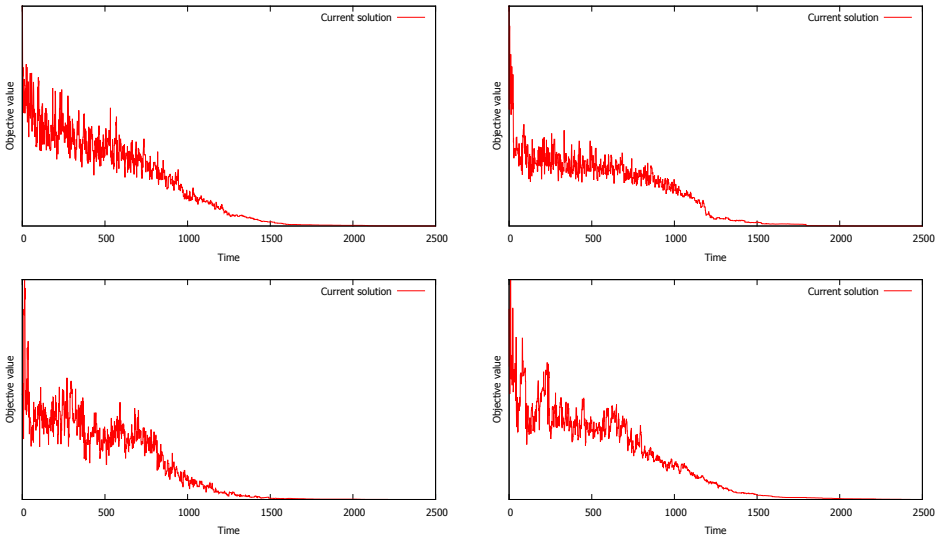

Figure 5.2: Four graphs each showing the cycle time of the solution as maintained by the SA algorithm over time.

the very first run. The red line (of which points are denoted as plusses) represents the average of the five other lines. One can see that two of the five problems do not have much variation in their results. However the other three problems do. For these three problems the solution found after 20 runs has improved by almost 10% on average. For all the five problems together an improvement is found of 7%.

As usual, there is a trade-off between time and performance. Visual inspection shows that running the *PreAssigner* six times gives a good balance between time and performance. Note however that all experiments performed using the *PreAssigner* in this Chapter are run only three times. Unfortunately at the time these experiments were performed we did not yet have this knowledge. One can thus expect that the improvement in the simple model can be increased even more. The first indication based on these five cases is that an increase in performance of 3% can be gained if the *PreAssigner* was run 6 instead of 3 times.

5.2.5 Comparison in the simple model

Let us now compare the solution obtained in the simple model as produced by our *PreAssigner* ($[pro_{exact,simple}]$) with solutions in the simple model obtained by the *Optimizer* ($[opt_{simple}]$). One would hope that the results obtained by the *PreAssigner* will outperform the imported solutions of the *Optimizer*. If this it not the case, then it is not very likely that a solution of our *PreAssigner* in the complete model, as obtained by pre-assigning TEUs, will yield an improvement over the actual *Optimizer*.

The relative improvement that the *PreAssigner* obtains can be seen in Figure 5.4. On average, the improvement of our *PreAssigner* in the simple model is approximately 4.5%. An improvement is found in 77% of the cases. For these cases an improvement of 7.3% is found on average.

Allthough the *PreAssigner* achieves an improvement for most of the jobs, there still exist some problem instances that do not beat the *Optimizer*. Inspection of these problem instances by hand lets us categorize these instances in three groups.

- The feeder bank as found in the complete model cannot be realized as a part bank in the simple model. Recall that the part bank is a simplification of the feeder bank as found in the complete model. To make sure that all parts that are placed on the part bank in the simple model can actually be placed on the feeder bank in the complete model, the size of

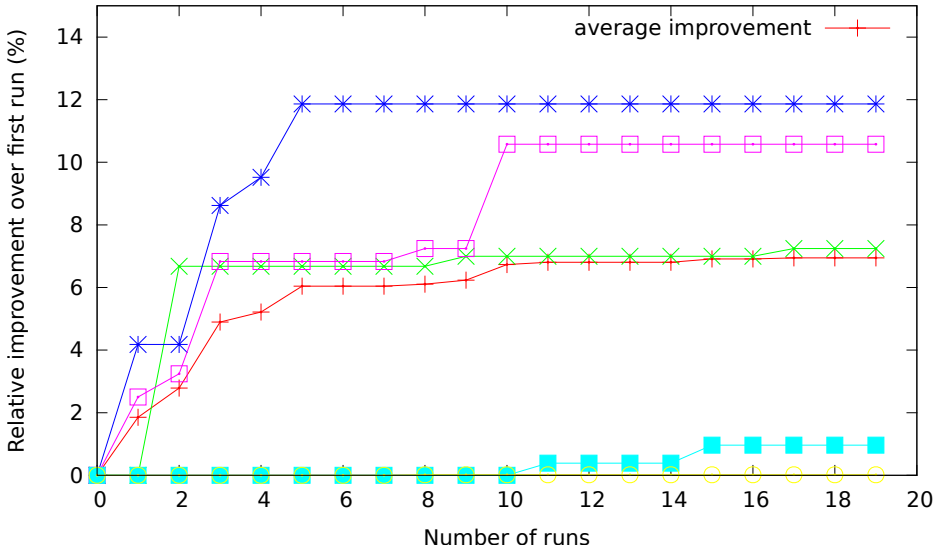

Figure 5.3: Line chart showing the variance of the *PreAssigner*. A point in the graph at r runs depicts the improvement of the best performing solution found after having runned the *PreAssigner* r times over the very first found solution. The red line represents the average of the five other lines.

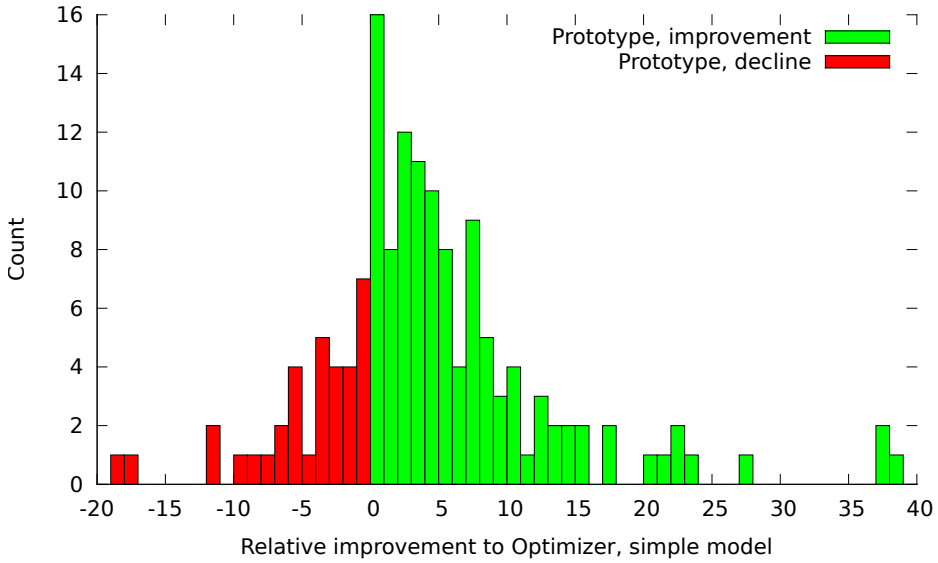

Figure 5.4: Histogram showing the relative improvement of the *PreAssigner* over the *Optimizer* in the simple model.

Toolbit exchanges	Size
$[0, 5)$	109
$[5, 10)$	19
$[10, 15)$	11
$[15, \infty)$	6

Table 5.2: Clustering of problem instances based on number of toolbit exchanges

the part bank in the simple model is underestimated. In general, fewer parts will fit on the AX in the simple model than in the complete model.

Obviously a solution as obtained by the *Optimizer* may use the entire feeder bank. If such solution is now imported in the simple model, the size of the part bank is typically violated for some of the robots. A solution as imported from the *Optimizer* is thus in general able to use more parts. This effect puts the *PreAssigner* in a disadvantage. For a couple of problem instances one can clearly see that due to this a better solution cannot be obtained in the simple model.

- For another set of problem instances it holds that their solutions lie in a deep local minimum that is hard to escape from. To improve such solution a large number of mutations need to be performed that all achieve a worse cycle time. In order to overcome such local minimum the temperature should probably cool at a much slower rate, improving the probability that concequtive worse solutions are accepted. Another solution is to provide a more direct way from this local minimum to a better solution. As a last solution the initially generated solution could be randomized or multiple intial solutions should be used.

- There are also problem instances known where the solution generated by the *PreAssigner* uses more toolbit exchanges and this clearly leads to worse cycle times. A cooling schedule that would cool more slowly could hopefully find better solutions.

An interesting observation is that a lot of solutions in the simple model actually use more toolbits than the imported solutions from the *Optimizer*. Still these solutions have a better cycle time on average. The most extreme example is for a problem for which the *Optimizer* uses 47 toolbit exchanges. For this problem the *PreAssigner* generates a solution that uses 95 toolbit exchanges, but that is still approximately ten percent faster.

The initial belief was that particularly jobs performing lots of toolbit exchanges could be improved. To test this belief we cluster the problem instances into four groups, based on the number of toolbit exchanges that the *Optimizer* originally uses. These four groups, as well as the number of problem instances that are contained in them, are given in Table 5.2. The improvement that the clusters achieve are given in Figure 5.5. Interestingly one can see that, indeed, the improvement of the *PreAssigner* in the simple model improves as the number of toolbit exchanges used by the *Optimizer* increases. The improvement for the 36 jobs that make up the three clusters having 5 or more toolbits, is 9% on average.

5.2.6 Relation simple model to complete model

As the simple model is a simplification of the complete model we suspect the cycle time of a solution in the simple model to be an underestimation of the complete model. A solution in the simple model is thus suspected to have a faster cycle time. Let us see if this is the case. Also it is interesting to see how large the difference between the simple and complete model actually is. To get an idea we compare the solutions obtained by the *Optimizer* ($[opt_{complete}]$) with the analog imported solutions ($[opt_{simple}]$). A histogram showing the relative difference in cycle time of an imported solution over the solution generated by the *Optimizer* is given in Figure 5.6. We observe that in 90% of the cases the simple solution is actually an underestimation. There are however cases that are an overestimation. These cases can possibly be explained by the fact that toolbit

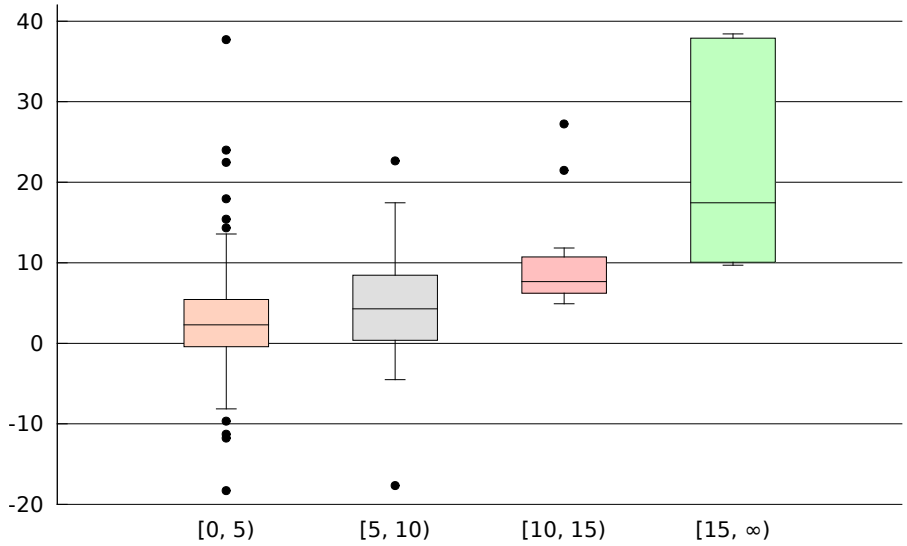

Figure 5.5: Boxplots showing the relative improvement of the *PreAssigner* over the *Optimizer* in the simple model. A boxplot is found for each cluster defined in Table 5.2.

exchanges are not masked at the end of a cycle in the simple model. On average a solution in the simple model is 7% faster.

The difference between the simple and complete model is less than 10% for 65% of the cases. There are however cases that have larger differences. We suspect that these are cases for which fiducial readings are important. Fiducial readings are not part of our simple model.

5.2.7 Comparison in the complete model

Let us now look at the actual results if we compare the results obtained from preassigning the TEU generated by the *PreAssigner* ($[pro_{exact,complete}]$) versus the results obtained by the *Optimizer* ($[opt_{complete}]$). An improvement here would mean an actual improvement in the real world.

The result of running all jobs can be seen in Figure 5.7. On average, the improvement of our *PreAssigner* in the complete model is approximately 0.5%. In 6% of the cases the exact same result is found. An improvement is found in 58% of the cases. For these cases an improvement of 1.8% is found on average.

Let us now again cluster the results according to the clusters as defined by Table 5.2. The results can found in Figure 5.8. Also in the complete model one can see that the obtained improvements increase as the number of toolbit exchanges increases. However this effect is not as strongly as seen before. In particular, a decrease in performance is observed in the second cluster. The 36 jobs that make up the three clusters having 5 or more toolbits still makes an improvement of 1.3% on average. The 17 jobs that make up the two clusters having 10 or more toolbits make an improvement of 2.9% on average.

We give three reasons that can explain why the improvements obtained in the simple model are not obtained in the complete model.

- The improvements found before are based on a simplified version of the actual model. Obviously information about the actual problem is lost in this step. For example the time it takes to place a component is approximated. The simple model can thus be a too simple representation of the complete model.

- The TEU of each robot as found by the *PreAssigner* is pre-assigned to the *Optimizer*. The rest of the solution is, however, discarded. The distribution of components is thus not used at

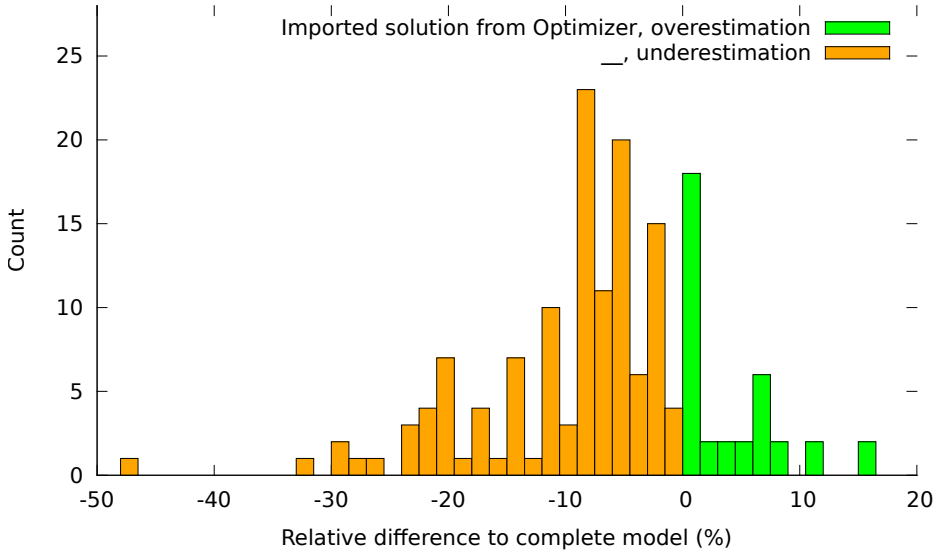

Figure 5.6: Histogram showing the relative difference of an imported solution over a solution generated by the *Optimizer*.

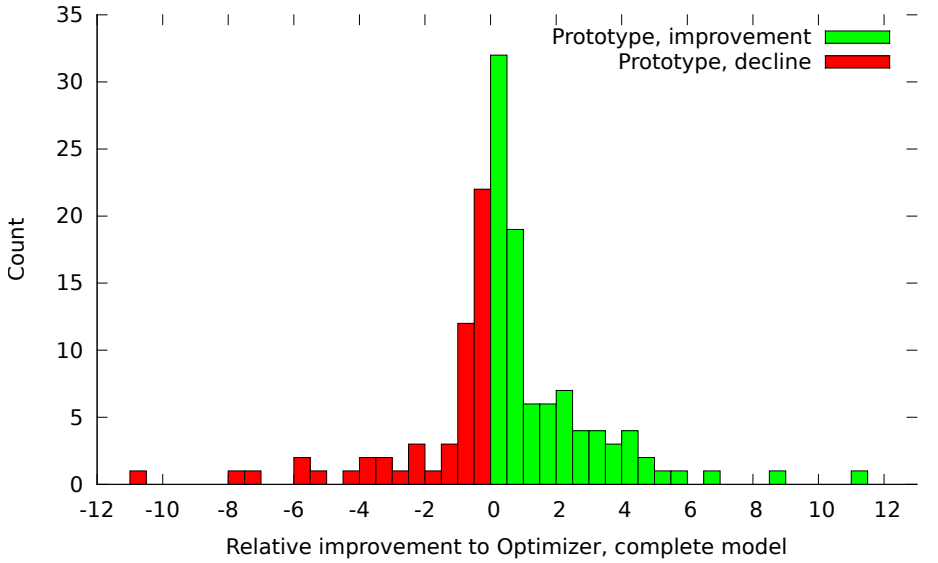

Figure 5.7: Histogram showing the relative improvement of the *PreAssigner* over the *Optimizer* in the complete model.

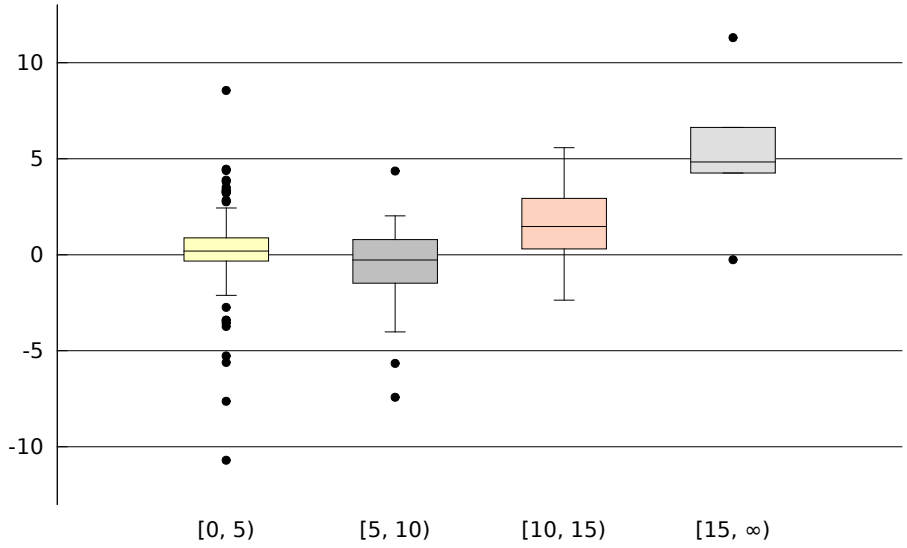

Figure 5.8: Boxplots showing the relative improvement of the *PreAssigner* over the *Optimizer* in the complete model. A boxplot is found for each cluster defined in Table 5.2.

all. Also the part bank, and allocation of toolbits to components as found by the algorithm that computes the cycle time is neglected. In order to obtain an identical solution in the complete model, the *Optimizer* needs to figure out these choices itself based on only the TEU pre-assignment. Because the TEU is generated carefully we hope that this is also the case. However there are solutions that have a completely different feeder bank, and thus a completely different solution. These solutions normally perform worse.

- We reported earlier that solutions are generated by the *PreAssigner* in the simple model that outperform the *Optimizer* but use more toolbit exchanges. However for these problems the *Optimizer* does not seem to be able to generate solutions that yield an improvement. Visual inspection of the solutions obtained in the simple model leads us to believe that obtaining such solution in the complete model is also possible. However, the algorithms the *Optimizer* uses are not tuned to using more toolbit exchanges were that will lead to an improved cycle time.

In general, there can be two reasons that the improvement decreases when going from the simple model to the complete model: either the computed pre-assignment may not be as good as hoped, or the pre-assignment is potentially good, but the *Optimizer* fails to take advantage of it. Is is not clear what the main reason is.

5.2.8 Influence exact toolbit computation

The exact method to compute the number of toolbit exchanges is proposed because we believe that it makes the simple model match better to the complete model. This should improve our chances that an improvement obtained in the simple model can also be obtained in the complete model. Is this actually the case? Earlier in this chapter we showed that the *PreAssigner* yields some improvements over the *Optimizer* in both the simple and complete model. These results are obtained with the *PreAssigner* using the exact method to compute toolbit exchanges. We expect that when using the estimation method, the improvement in the complete model will be less significant. Therefore we compare the improvement using the exact method ($[pro_{exact,complete}]$ over $[opt_{complete}]$) to the improvement using the estimation method ($[pro_{estimate,complete}]$ over $[opt_{complete}]$).

Recall that using the exact method to exchange toolbits there was an improvement over the *Optimizer* of 0.5%. However when the estimation method is used we get a decrease of approximately -0.5 percent over the *Optimizer*. The estimation method thus performs worse. The same holds if we look at the jobs for whom the *Optimizer* used many toolbits exchanges. Using the exact method an improvement of 2.9% is obtained, using the estimation method only an improvement of 2.0% percent is obtained. Hence using the exact method to compute toolbit exchanges indeed helps to obtain a better solution in the complete model.

Chapter 6

Future work

Optimization algorithms for NP-hard problems, as the one described in Chapter 4, will usually not achieve the optimal solution for every problem instance. Therefore tweaking and improving such algorithms is a never-ending process. Because the time for a master's thesis is limited, not all ideas have been implemented or tested. In this Chapter an overview is given of the ideas that could be used to improve the *PreAssigner*.

First of, the *PreAssigner* has some constraints. It can namely only produce solutions for single-panel problems. The algorithm can however be extended quite trivially to multiple panels. To do so, a solution in the simple solution should consist of a distribution of components to buckets for each panel. The cycle time should then be defined as the weighted cycle time of the individual problems.

Recall that toolbits taking place prior to the return stroke can be masked by the transport in the complete model. Thus a toolbit exchange at the end of the program can be performed without it counting towards the cycle time. This is not modelled in the simple model, but it can be implemented rather straightforwardly. The algorithm to compute the minimum number of toolbit exchanges can for example be adapted by simply changing the weight to zero for the edges of which their source is a start vertex. In this way the Simulated Annealing algorithm will most likely prefer to perform a toolbit exchange at the end of the cycle. This straightforward solution has not been incorporated in the *PreAssigner* because we became only aware of this effect at the end of this master's thesis, and not all experiments could be repeated with this adjustment. We did however experiment briefly with an adapted algorithm on a fraction of the problem instances. During these experiments no real improvement in the simple model was found however.

The simulated annealing algorithm as implemented in this thesis has many parameters that can be tweaked. The start temperature, cooling schedule, probability function and weights of the mutators. The choice for the parameters as described is based on observations done during testing the algorithm. However, no tweaking of parameters has been done. By carefully tweaking these parameters most likely improvements can be expected.

We observed that there is a lot of variance in the cycle times of the solutions that are generated by the *PreAssigner*. We mitigate this in this paper by running more instances of the algorithm. Most likely, however, one can get the same increase in performance without running the algorithm multiple times. This can for example be achieved by using multiple initial solutions or by restarting the algorithm during its execution.

A solution in the simple model consists of only a distribution of components to buckets. However, after evaluating the cycle time using our algorithm, also a TEU and part bank are known for each robot. Only the TEU is pre-assigned to the *Optimizer*. The *Optimizer* also supports pre-assigning feeders and parts to a robot. One could also try to pre-assign feeders and parts corresponding to the part bank as found in the simpel solution. This would steer the *Optimizer* much more directly towards the solution as computed in the simple model. It would, however, require either an adaption of the simple model or an algorithm should be placed in between that can convert a part bank to a feeder bank.

Lastly, recall the algorithm of the *Optimizer*. In Step 5 the components were ordered in such a way that the number of toolbits, and hopefully toolbit exchanges, was reduced. This problem looks similar to the problem that is solved by our *PreAssigner* to compute the minimum number of toolbit exchanges. However, the later one solves it exactly and will induce a lower number of toolbit exchanges. This algorithm could well be used in that step.

Chapter 7

Summary and Conclusion

In this master's thesis we studied the *Optimizer*, software used by Assembléon to generate placement programs with whom the AX can assemble panels. That software is known to sometimes produce placement programs using lots of toolbit exchanges, for which Assembléon believes lower cycle times can be obtained.

In Chapter 2 of this thesis we started by describing what hardware makes up an AX. The *Optimizer* software was described, which is as an algorithm to produce such placement programs. In particular, we explained the possibility the *Optimizer* offers to pre-assign toolbits to the TEUs of robots. After we familiarized ourselves with the AX, we turned our attention to the internals of the *Optimizer* in Chapter 3. We found clues as to why the current algorithm can sometimes yield a solution containing too many toolbit exchanges, the most important being that the TEUs of robots are computed using simple heuristics, but remain static during the complete genetic algorithm.

In Chapter 4 we presented the *PreAssigner*, an algorithm that can compute a TEU pre-assignment for each robot. Such a pre-assignment can be fed into the existing *Optimizer*, yielding real-world results. A simplified model was introduced. The simplified model was shown to be NP-hard still and a Simulated Annealing algorithm was presented to compute efficient placement programs in the simplified model. The required ingredients that Simulated Annealing needs were filled in by introducing five mutators, an algorithm to compute an initial solution and an algorithm to compute the fitness of a solution. In Chapter 5 we experimentally tested the performance and characteristics of the *PreAssigner*. In both the simple model and the real world the *PreAssigner* was shown to generate better results. Finally, in Chapter 6, we gave ideas and tasks that could not be completed within the scope of this thesis. These could be used by Assembléon to continue work on the *PreAssigner*.

Assembléon believes the solutions of problem instances can be improved for which the *Optimizer* now uses many toolbit exchanges. The *PreAssigner* described in this thesis has, indeed, shown to be able to generate solutions for these problem instances that outperform the existing *Optimizer* by 2.9%. In fact, the solutions generated by the *PreAssigner* outperforms the *Optimizer* on average over all test instances with 0.5%. The production time of panels can thus be reduced by Assembléon by using the *PreAssigner* as a pre-processing step.

The initial belief of Assembléon was that results could only be improved by reducing the number of toolbit exchanges. Surprisingly there are several cases where our solutions in the simple model generate more toolbit exchanges than the *Optimizer*, but still achieve lower cycle times. This shows that minimizing the number of toolbit exchanges should not always be the goal when trying to increase performance.

Contrary to the 0.5% improvement that the *PreAssigner* achieves over the *Optimizer* in the complete model, the improvement in the simple model is a staggering 4.5%. It is not completely clear why there is a large gap between these two. It can be because the generated pre-assignment is not as good as hoped. In that case most likely the simple model is not a good representation of

the complete model. It can also be because the optimizer fails to take advantage of a generated pre-assignment that is potentially good. For both cases ideas have been listed in Chapter 6 that could mitigate these problems. One option could be to adapt the *Optimizer* in order to cope with a potentially good pre-assignment. However, the internals of the AX have been shown to be implemented rather strangly in a couple of ways. One could therefore also consider replacing the complete *Optimizer* in favor of an algorithm based on the *PreAssigner*.

Bibliography

[1] L. J. Eshelman. *The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination* in Foundations of Genetic Algorithms. Morgan Kaufmann, pp. 265-283 (1991).

[2] D. S. Hochbaum, D. B. Shmoys. *Using dual approximation algorithms for scheduling problems theoretical and practical results.* In ACM volume 34 issue 1, pp. 144-162 (1987).

[3] R. M. Karp. *Complexity of Computer Computations.* In Complexity of Com-puter Computations, New York, Plenum Press, pp. 85-104 (1972).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Section 24.3: Dijkstra's algorithm.* In Introduction to Algorithms. MIT Press and McGraw-Hill, pp. 595-601 (2001).

[5] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. *Optimization by Simulated Annealing.* In science volume 220 no 4598, pp. 671-68 (1983).