Job Shop Scheduling with Metaheuristics for Car Workshops

Hein de Haan
s2068125

Master Thesis

Artificial Intelligence
University of Groningen

July 2016

First Supervisor
Dr. Marco Wiering (Artificial Intelligence, University of Groningen)
Second Supervisor
Prof. Dr. Lambert Schomaker (Artificial Intelligence, University of Groningen)

university of groningen

faculty of mathematics and natural sciences

Abstract

For this thesis, different algorithms were created to solve the problem of Jop Shop Scheduling with a number of constraints. More specifically, the setting of a (simplified) car workshop was used: the algorithms had to assign all tasks of one day to the mechanics, taking into account minimum and maximum finish times of tasks and mechanics, the use of bridges, qualifications and the delivery and usage of car parts. The algorithms used in this research are Hill Climbing, a Genetic Algorithm with and without a Hill Climbing component, two different Particle Swarm Optimizers with and without Hill Climbing, Max-Min Ant System with and without Hill Climbing and Tabu Search. Two experiments were performed: one focussed on the speed of the algorithms, the other on their performance. The experiments point towards the Genetic Algorithm with Hill Climbing as the best algorithm for this problem.

Contents

Chapter 1

Introduction

1.1 Scheduling

Imagine you own an automobile repair shop. You pay fifteen full-time mechanics, each with their own skill levels and certificates, to repair the broken cars of your customers. Of course, you own a few bridges in case a mechanic needs to lift a car to work underneath the car, there are lots of tools for the mechanics to use and you maintain a certain stock of car parts. All cars that are brought in are diagnosed, and the concerning mechanic makes a list of tasks that need to be done in order to have the car fixed. Since you make appointments with your customers as to when their cars are ready, this results, for each day, in a list of tasks on specific cars that need to be done that day. Keeping in mind the different skill levels, certificates, work times etcetera of your mechanics, for each day, you make a schedule, which assigns each task to a specific agent, on a specific time, and using a specific bridge (if it needs one). Of course, the specific times depend on the delivery times of new car parts, the availability of bridges and tools, etcetera. Furthermore, you have to be efficient: you try to have your mechanics perform as many tasks (workshop activities) as possible every day, in order to generate a maximum profit. Given a relatively large number of tasks, you have to assign these tasks wisely, or else you might end up having your mechanics work late. For these reasons, you spend a lot of time trying to create the best possible schedule. However, imagine having created such a schedule for, say, a Tuesday. You have each of your mechanics scheduled to do their first tasks at nine o'clock in the morning. On Tuesday morning, you get a call from one of your mechanics, saying he had an emergency and will be half an hour late. This means you have to create a new schedule which deals with this mechanic being late. However, you do not have enough time to create a good schedule, since it is almost nine o'clock. Since each mechanic already has his hands full, the result is tasks being done too late

and hence there are disappointed customers. And the problem you have happens quite often, since there are multiple causes: for example, sometimes car parts are delivered too late, sometimes a bridge is out of order, etcetera.
Of course, there are ways to solve such a problem. You could, for example, leave some room in your future schedules, giving your mechanics small breaks between their tasks in order to be able to shift tasks to a different time in case of a problem. However, this would severely lower the efficiency of your schedules, and it is hard to estimate how many of such small breaks you need, and how big they should be. What you want is to be able to reschedule very quickly in the case of a problem such as the one just described, in order to keep your schedule as efficient as possible and have your customers as satisfied as possible. The solution the author proposes in this thesis is a computer system which can schedule (and therefore reschedule) autonomously, given a set of tasks, a set of mechanics, a set of bridges and a set of car parts, optimizing for efficiency and customer satisfaction. Since computers can schedule a lot faster than humans, one can have new efficient schedules very fast during the day, in case problems show up and the old schedule is outdated. Furthermore, you do not have to spend a lot of time on a task a computer can do for you; therefore, you can focus on other things.
The situation just described is quite specific, but the scheduling problem is present in many other situations, making the research of this thesis even more important. Examples of course include other kinds of repair shops, but really any situation in which different agents (in this case mechanics) have to be assigned to different tasks. For example, the scheduling of high school or university classes includes assigning different teachers, each with their own qualities, to different classes, with constraints like two classes attended by the same student cannot be assigned to the same time slot. Of course, each situation is different, but the general problem remains.
Now that we have discussed the importance of autonomous scheduling in practice, let us briefly look at scheduling from a theoretic viewpoint. Since the problem of scheduling itself is NP-hard, it is, given the current state of processor power, impossible to find an algorithm that both guarantees to return the best solution and does so in reasonable time. Simple brute force algorithms that just search through all possible solutions finish in a workable time frame only for relatively very small scheduling problems, and already take too much time when given a problem which is just a little bit bigger. Of course, heuristic algorithms, which do not search through all solutions, take much less time to return their solution but are not guaranteed to return the best schedule. It is of course this trade-off between performance and efficiency which makes autonomous scheduling (and most optimization problems) interesting. Furthermore, this same trade-off has led to a lot of algorithms proposed to solve the scheduling problem, each with its own

performance and efficiency levels. Examples of such algorithms are Genetic Algorithms [21], Ant Colony Optimization [7], a collection of algorithms which includes Ant System Optimization, and Particle Swarm Optimization [25].

1.2 Research Question

In this graduation project, different (metaheuristic) algorithms for Job Shop Scheduling are compared. These algorithms are: a Genetic Algorithm (with and without a Hill Climbing component), two Particle Swarm Optimization variants (again, with and without a Hill Climbing component), Max-Min Ant System (also, with and without Hill Climbing), Tabu Search and a basic Hill Climber, which will be described later. More specifically, these algorithms are used for scheduling car workshop activities: the algorithms should assign all tasks planned for a single day to a specific mechanic, each on a specific time, while considering the use of scarce resources, consumables and different skill levels of the mechanics. Resources are entities that can be used over and over again, but can be used by only one car at a time. For this research, the only resources are bridges, which can only lift one car at a time, and the mechanics themselves, who of course can only work on one car at a time. Consumables are defined as entities that can be used only once and thus their stock has to be refilled from time to time; examples of consumables are car parts, such as lights and tires. For this research, one consumable was defined, simply called "car part".
Keeping in mind the previous informal description, the research question for this thesis is: which algorithm, or which algorithms mentioned in this subsection works or work best for the scheduling problem mentioned here, in terms of both speed and performance? In this context, speed means how fast a specific algorithm finds a reasonable schedule, meaning a schedule without obvious mistakes like a task being done after its maximum finish time, or a mechanic working after hours. Performance includes not making those mistakes, but also efficiency: the less total work hours a schedule has (given, of course, the same tasks and mechanics etcetera), the better it is.
The remainder of this thesis is structured in the following way. First, a theoretical background of job shop scheduling will be given, with a more formal description of the problem and some work that has been done to solve similar problems. After this, all methods deployed for the research of this thesis will be described in detail, with pseudocode. Then, in chapter 4, the experimental setup and results will be discussed. Finally, a conclusion will be offered, together with some suggestions for future work.

Chapter 2

Theoretical Background of Job Shop Scheduling

Before the author continues to describe and discuss the algorithms developed for the research described in this thesis, it is first important to give a good theoretical background of Job Shop Scheduling in general, with of course a formal description and some early and some recent work that has been done to tackle the problem.

2.1 Formal Description

As noted before, before continuing to describe any algorithms solving a problem, it is crucial to first precisely define the problem. The Job Shop Scheduling problem as adopted for this research is defined as follows. We have a set T of tasks, or jobs, which need to be executed. We also have a set M of mechanics who can execute tasks. Not all mechanics are qualified to perform each task; therefore, for each mechanic, we have a $Q_{mechanic}$ which contains qualifications: if and only if $q_2 \in Q_1$ is mechanic 1 qualified to do task 2. Furthermore, for each mechanic $m_y \in M$ we have a set D_y of task durations for each task the mechanic is qualified to perform. So, if and only if $\delta_0^y \in D_y$ does it take mechanic y δ_0^y time units to perform task 0. Moreover, each task $t_x \in T$ has a corresponding pair of a minimum start time and a maximum finish time: $\{\tau_x^{min}, \tau_x^{max}\}$. The same holds for each mechanic $m_y \in M$: $\{\theta_y^{min}, \theta_y^{max}\}$. Finally, any task x may have a $t_x^{predecessor} \in T$, where $\exists p(t_x^{predecessor} = t_p \wedge p < x)$, which is a task that has to be finished in order for t_x to be executable. Apart from the mechanics and the tasks, we have bridges which lift a car in order for mechanics to be able to work under the car. Therefore, each problem has a set of bridges B. Each bridge $b_i \in B$, for $i > 0$ (b_0 would indicate no bridge, in combination of a task which uses none) is a tuple of the maximum weight it can lift, w_i^{max} and a list of time units at which the bridge is scheduled

for use. Such a list can never contain the same time unit twice, since this would mean a bridge has to lift two cars at once. We also have car parts, which as discussed earlier are all the same. This set is modeled as C, which simply consists of pairs of time units at which its stock increases or decreases (by delivery or usage, respectively) and the amount (positive or negative) with which this value changes. Apart from this, we have the c_{start}, which is a constant which tells the amount of car parts at the start of the day. It is of course crucial that the actual stock $c_{current}$, which can be calculated for each time unit, can never be below zero.

A *solution* can now be defined as a set S_i of tuples of a task, a mechanic and a bridge (b_0 if the task uses no bridge). Each task must be an element of exactly one such tuple. There is no such constraint for the mechanics. This indeed means that, in principle, a mechanic could end up with no tasks at all if that makes for an efficient schedule. Such a tuple of a mechanic, a task and a bridge looks like this: $\{t_x, m_y, b_z\}$, and means task x is to be performed by mechanic y on bridge z (or no bridge at all in case $z = 0$).

The fitness value of a solution S is inversely related to: $\sum_{\forall\{t_x,m_y\}\in S} \delta_x^y$, simply said the total amount of work time of all mechanics. Specifically, the total amount of hours the mechanics could potentially work is devided by the total amount they do work:
$\frac{MaximumCombinedWorktimeofAllMechanics}{\sum_{\forall\{t_x,m_y\}\in S} \delta_x^y}$
Furthermore, the fitness is negatively affected by agents working later than their θ^{max} and tasks being finished later than their τ^{max}. More specifically, for each hour an agent is working late and for each hour a task is finished late, the fitness value is descreased by 1.

Note that the time at which a task is performed is not specified anywhere in this model. This is because, to save computation time, the algorithms do not optimize the execution times of the tasks: they only assign a specific mechanic and a bridge to each task. After this part of the scheduling is done, a simple algorithm takes over and assigns a start time to each task. All tasks done by the same mechanic are simpy scheduled in order, beginning with the task with the lowest identification number and ending with the one with the highest, placing each task as early as possible in the schedule. Doing the schedule timing this way might result in less-than-optimimal solutions (which will be explained in more detail in the Methods chapter), but the cost of this is far lower than the benefit of the computation time saved.

Also note that the fitness value of a solution does not depend on the bridges

being in double use at a certain time, or the car parts stock reaching a negative value. This is because this simply cannot happen. Solutions are automatically created to avoid this; tasks will simply be scheduled later, when there is a bridge available or when the car parts stock is big enough again. Indirectly, such problems may cause late hours for mechanics or tasks not being finished in time, which is directly reflected in the fitness value.

Figure 2.1 gives a more schematic representation of the scheduling problem. In this figure, it is shown that for each task, the scheduling algorithm first has to choose a mechanic, and after this, a bridge is chosen automatically. The figure only has four tasks, two mechanics and two bridges, and already shows many possible schedules. This gives an appreciation of how big our scheduling problem really is: imagine having 100 tasks, 16 mechanics and 25 bridges.

2.2 Early Work

In one of the very early papers on scheduling problems, Giffler and Thompson [12]) describe a problem similar to the Job Shop Scheduling problem presented in this thesis. Their problems consist of multiple facilities, which correspond to the mechanics in this thesis, and multiple commodities, corresponding to the tasks in this thesis. Giffler and Thompson define what they call an *active schedule* [12]), which in short is a schedule in which no commodities (tasks) can be performed earlier by the facility they are processed by without having to reschedule another task. So, an active schedule is one in which there are no idle times between commodities processed on the same facility. They use the concept of active schedule to define a very interesting algorithm: one that can, given a scheduling problem, find *all* the active schedules solving that problem. An exact specification of the algorithm is beyond the scope of this thesis (and can easily be found in [12]); suffice it to say that the algorithm is an exhaustive one (after the search space is defined in a smart way). The reason this is so interesting, is that in general (and in particular in this thesis) scheduling is seen as an optimization problem, while the algorithm of Giffler and Thompson does not just find the *best* solution in a class of solutions (the class of active solutions), but it finds *all of them*. Alas, creating such an algorithm for the Job Shop Scheduling problem presented in this thesis would be impractical, since such an algorithm would take too much time to find all the "good" solutions.

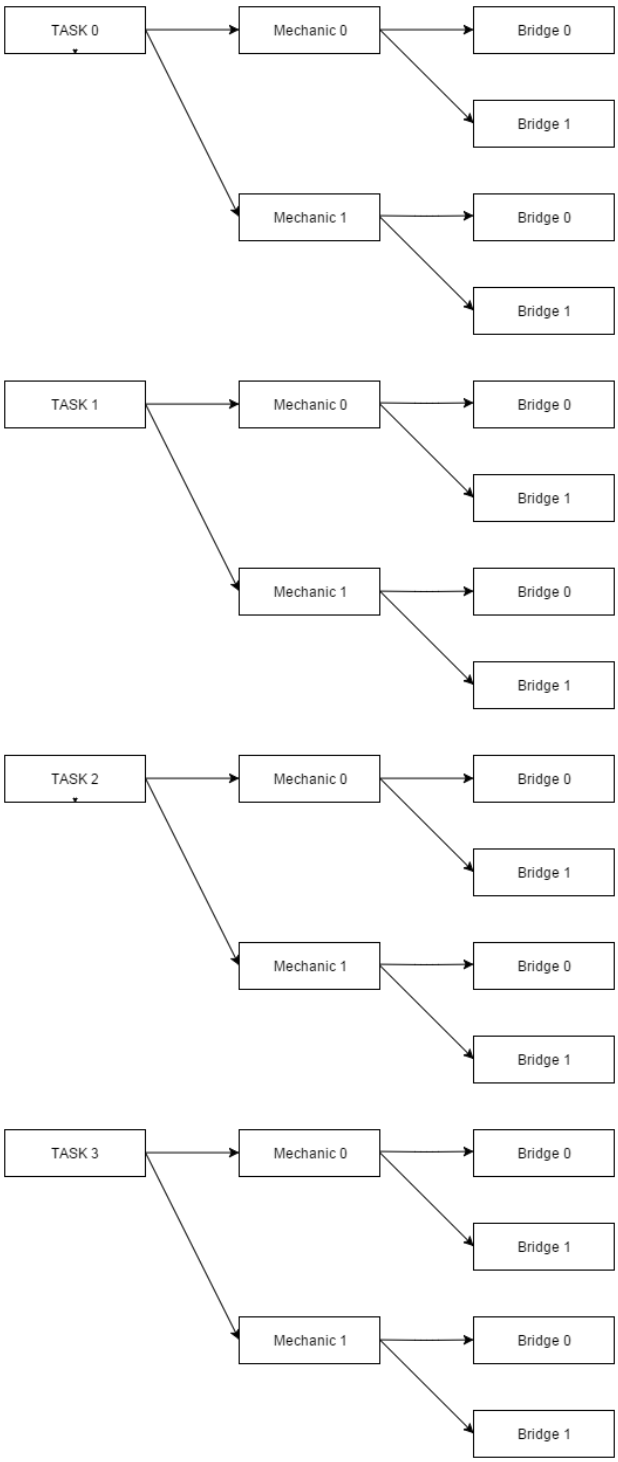

Figure 2.1: Schematic Representation of the Scheduling Problem

2.3 Current Algorithms

As briefly pointed out before, there have been a lot of (successful) recent attempts to tackle the Job Shop Scheduling problem (in one form or another). Of course, throughout time, computer power increased a lot *and* cleverer algorithms were invented, a combination which has led to researchers being able to find good solutions to much bigger scheduling problems. Recent examples of algorithms which have been used to solve the problem are Genetic Algorithms [21], Simulated Annealing [26], Ant Colony Optimization [7] and Neural Network approaches [22]. Since Genetic Algorithms and Ant Colony Optimization will be covered in the next chapter, this section will focus on Simulated Annealing and Neural Networks.

2.3.1 Simulated Annealing

Simulated Annealing is a technique for solving optimization problems which was first described in 1983 [26]. The inspiration for and the name of this algorithm come from a process in metallurgy named annealing, which is done to make metals more workable by decreasing their hardness. It works by heating the metal and then slowly cooling it, which is where the anology with Simulated Annealing exists. Simulated Annealing has four basic principles: the creation of neighbouring solutions to the optimization problem, a decreasing temperature, an energy function and an acceptance probability function [26]. Since this algorithm is not used in the research described in this thesis, only an informal description will be given. All possible solutions to the optimization problem being solved have an energy value, which inversely describes how "good" that solution is. The goal is to find the solution with the lowest energy associated with it. At each iteration, Simulated Annealing creates a solution which differs just slightly from its current solution: the neighbour. The algorithm chooses whether to take this neighbour as its next solution or to stay at the current solution with the acceptance probability function. This function uses the energy of the current solution, the energy of the neighbouring solution being considered and the current temperature to compute the probability of that neighbouring solution being chosen. If the neighbour is not chosen, the algorithm keeps the current solution and creates a new neighbour to repeat the process. The lower the energy of the neighbour, the higher its acceptance probability, all else being equal. However, a high temperature will make the acceptance probability higher, which can lead to the choice of a neighbour whose energy is higher than that of the current solution. The idea behind this is that only chosing neighbours with lower energies, as in greedy algorithms, leads the search into local minima: points in the search space with an energy lower than those of its neighbours but not lower than solutions further away. If, however, the algorithm is close to the global minimum (the best solution possible), one does not want it

to jump away from it because of the temperature still being high. Therefore, the temperature lowers gradually during the execution of the algorithm.
As each algorithm does, Simulated Annealing comes with advantages and disadvantages. As noted in [26], one advantage of the algorithm is its ability to guarantee finding the best solution to an optimization problem. However, as expected, a related disadvantage is its long computation time, which makes it relatively unsuitable for the scheduling problem of this research. Other advantages of Simulated Annealing include ease of coding and its ability to use fitness functions with relatively high levels of nonlinearities, discontinuities and constraints. Especially the ability to process constraints itself would make Simulated Annealing great for our scheduling problem, since this problem has a lot of constraints like scarce resources. However, other algorithms like Genetic Algorithms can handle these constraints too.

2.3.2 Neural Networks

Introduction "Neural networks" are a collection of tools which all work by the same principle: distributed computing. A neural network consists of a number of simple computational units, also called *nodes* (see for example [20]). Nodes are connected to each other with interunit *weights*, and each compute an output value based on the input it receives from other units via those weights and its internal parameters. Note that these weighted connections are directed: one node's output multiplied by the weight of such a connection serves as input for another node, but that same connection does not serve the other way around. This does not mean nodes cannot have connections to each other both ways, which is the case in Hopfield Networks, but more on those later.
Neural networks are often built as layers of nodes, each layer only having connections to the nodes of the next layer. Such networks are called Feedforward Neural Networks. Such a network could for example be used for textual character recognition, in which case an image of a textual character is given to a Feedforward Neural Network, which then has to decide what character it is. A complete description of how one would build such a network is beyond the scope of this thesis. In short, the programmer would have to create a numerical representation of the image (for example, the pixel intensities) to give as input to the first layer of the network. The parameters of the network then abstract patterns from the input, and finally decide to which of a number of characters (26 in case of the alphabet) the input belongs. It would use one node for each character in its final layer. The node with the highest output value determines the character choice of the network. Of course, the weights of the network somehow have to get the right values in order to make the right decision. The process of getting those values is called *learning*, and can be done by giving the network example input-output pairs, from which

it can learn the right weights with an algorithm called backpropagation. Besides textual character recognition, Feedforward Neural Networks have been used in a lot of other domains. One interesting example is the application of such networks in game playing done by a computer. One could train a network to evaluate board positions of, for example, Othello (Reversi), after which such a network can be used by a computer to determine which moves to play [10]. Training such a network (i.e. having it set its weights to the correct values) involves having it play a lot of games from which it learns what board positions are good and which are bad.

Hopfield Networks Briefly described before, Hopfield Networks [22]) are a class of neural networks with weights going from one node to another and vice versa. More specifically, a Hopfield Network consists of a number of binary threshold units (units which have a binary output, 1 or 0, based on whether the input is higher than the threshold), which are all connected to each other. Weights are symmetric, meaning the weight $w_{1,2}$ from a certain node n_1 to another node $n_2 \neq n_1$ equals $w_{2,1}$. Hopfield Networks are a (simple) model of biological associative memory. It works as follows: a network of N nodes has 2^N states it can be in. Depending on the values of the weights between the nodes, some of these states have lower *energy* than others. The energy is calculated as follows [20]):

$$E = -\tfrac{1}{2} * \textstyle\sum_{i,j} w_{ij} * x_i * x_j,$$

where x_i is the state (output value) of node n_i. Note that this assumes all thresholds to be zero. The idea of the energy function, however, is clear: if nodes with positive weights between them both fire, they lower the energy of the network; in contrast, if nodes with negative weights between them fire, the energy increases. If only one of the nodes of a pair fires, nothing happens since one of the factors of the equation then is zero. If we were to run such a Hopfield Network, we could update each node sequentially (asymmetric updating). Updating a node means having it set its state (output value) based on its inputs, the weights corresponding to the inputs and the threshold of the node: the sum of all inputs multiplied with their corresponding weights is compared to the threshold. We do so for every node, and then start over again. It can be easily shown (see for example [20]) that the energy of such a Hopfield Network will decrease or stay the same for every update. The network will converge to a state in which the energy cannot decrease anymore. This is why a Hopfield Network is called associative: we input a state and it converges to another state based on the input and its weights. It therefore in a way remembers the state with the low energy.
Now that a short introduction on the workings of Hopfield Networks are given,

let us see how they could be used for Job-Shop Scheduling, although not for the specific problem described in this thesis. First, however, let us discuss how a quite similar (at least on an abstract level) problem, the Traveling Salesman Problem, might be solved. The objective of the Traveling Salesman Problem is to visit a number of cities all exactly one time, and then to return to the city that was first visited, and find the shortest route doing so. The distances between each pair of cities are given. As is done in [19], one might design a Hopfield Network to solve this problem. In [19], the author does this by creating a Hopfield Network with $N * N$ nodes, where N is the number of cities. This is because each city can be chosen at any point in the route, which of course is N steps long. Since each city has to be visited exactly once, each column and each row have to have exactly one node firing. This can be realized by setting the weights between nodes of the same column and nodes of the same row negative. Furthermore, nodes that are in adjacent rows should have weights negatively proportional to the distance, as to enforce that the resulting path be minimal. One could do almost exactly the same thing for solving the Job-Shop Scheduling problem. After all, each task is done by exactly one mechanic, and we search for the "shortest" schedule, i.e. the most efficient one with respect to time duration. One could create a network of size $T * M$, where T is the number of tasks and M is the number of mechanics. Each task is done by exactly one mechanic, so the nodes of a row (which consists of M nodes) have negative weights between them. The distances of the Traveling Salesman Problem can be directly translated to time durations for specific task-agent combinations here: from row i to row j, the weights to each node (and the weights from these nodes back to row i) could be negatively proportional to the time it takes each specific agent to perform task j. A problem, however, arises. There should be negative weights in the column, since one mechanic can only perform a limited number of tasks. However, this is very difficult to do. If each agent can do only one task, the problem is virtually the same as the Traveling Salesman Problem and the weights in the columns can be set very low, say -1. If each agent can do, say, 5 tasks, this would mean setting the weights to $-\frac{1}{5}$. However, this is too naive: some tasks only take ten minutes, where others take two hours. Tasks of ten minutes should have weights closer to zero than the long tasks. Since weights are symmetric, it seems impossible to determine the weights between short and long tasks in the same column. Furthermore, adjacent nodes in the same column already have weights between them: the time duration weights. Furthermore, our scheduling problem has other constraints which are seemingly impossible to implement in this network: the use of scarce resources and consumables. Solutions to all the problems might be available, but would (at least as far as the author can see) all lead to very large networks, which obviously have high computation times. This is why no neural network was created for our

scheduling problem.

Chapter 3

Methods

3.1 Introduction

In this chapter, all algorithms that were used for scheduling in this research are described. However, before this, it is important to discuss what the algorithms exactly need to do. First of all, let us look at the input the algorithms receive.
For this research, a problem generator was created: a script that creates an array of tasks, an array of agents (the mechanics), an array of cars, an array of car parts and an array of bridges. A task has a number of parameters: a minimum start time, a maximum finish time, a boolean variable indicating whether or not a bridge is needed to perform the task, a variable indicating whether another task has to be performed before this task, and if so, which one, and a variable indicating whether it needs its executing agent to have a certificate, and if so, which one, and two arrays indicating which car parts the task needs and how many of each, respectively. Just like a task, a agent has parameters as well: most importantly, it has an array with duration values, indicating for each task how long it takes the agent to perform that task. Furthermore, an agent has a time at which he should be finished with his tasks to go home. A car only has one parameter: its weight, which is important if it needs to be lifted by a bridge. Related to this, a bridge has as its only parameter the maximum weight it can lift. As said before, the problem generator creates problems with only one type of car part; therefore, the array of car parts has only one entry. A car part has three parameters: one specifying the initial amount of that car part at the start of the day, a list of delivery times (times at which a new amount of this car part arrives at the workshop) and a list of values indicating how many car parts arrive at those delivery times.
Now, let us discuss what the algorithms have to do once they receive a problem created by the problem generator. The goal is to have each task assigned to exactly one agent and, if needed, exactly one bridge, at a specific time, of course without

violating the constraints (the times at which the agents go home, the stock of the car parts, etc.). It is very important to note here that the algorithms do not have to plan the execution times of the tasks explicitly: these values are calculated automatically once an algorithm has finished planning. All the algorithms need to do is assigning each task to a specific agent and bridge (if a bridge is needed). After this, all tasks assigned to the same agent will be planned directly behind each other in time, the order being determined by which tasks have to be finished first in order for other tasks to be executable. Tasks being performed at the same time by different agents will automatically be assigned to different bridges, if possible. If this is impossible, one of the tasks has to be scheduled later in time than the other. This specific bridge problem can lead to suboptimal schedules. For example, Task 0 and Task 1 are both assigned to Agent 0, and Task 2 and Task 3 to Agent 1. All four tasks take the same amount of time given this assignment, and no other tasks have to be scheduled. Task 3 has to be performed after Task 2, but Task 0 and Task 1 can be executed in whatever order. Also, there is one bridge, Bridge 0, and Task 0 and Task 2 both need a bridge. Now, Task 0 will at first be scheduled before Task 1, since its identifier (0) is lower (and there is no specified order). Task 0 will of course be assigned to Bridge 0. Task 2 will obviously be scheduled before Task 3 and will also be assigned to Bridge 0. This results in Task 2 being performed after Task 0 since these tasks use the same bridge, which results in Agent 1 doing nothing while Agent 0 is executing Task 0, and finishing late. This could have been avoided by having Agent 1 perform Task 1 first and then Task 0. In practice, however, this does not seem to be a frequently occuring problem. Therefore, the cost of this problem is far lower than the benefit of the computation time saved.

3.2 Hill Climbing

3.2.1 Introduction

Hill Climbing is the simplest algorithm used in the research of this thesis. Although its results may not be as good as the other algorithms described in this thesis, Hill Climbing is surpisingly good considering how simple it is and how fast the algorithm returns a solution. The authors of [32] compare the workings of Hill Climbing (or steepest-ascent) with trying to find the top of Mount Everest, while having amnesia and walking in a thick fog: you have no memory of the route you took so far, and you can only see what is very near to you. Hill Climbing works very much like this: it is a so-called Local Search algorithm. It starts with a random solution to the problem it is trying to solve. It then generates all neighbouring solutions: states that can be created out of the original solution with only

a small change. The algorithm then uses its objective function to determine the best of all those neighbouring solutions. If the best one is better than the original solution, the original is replaced by that new solution (this makes Hill Climbing a *greedy algorithm*). Hill Climbing then repeates the previous steps until it gets stuck: that is, if no neighbouring solution is better than the current best solution. If this happens, the algorithm returns the current solution.
As noted before, Hill Climbing is computanionally cheap: it returns a solution very fast and only requires a constant and small amount of memory. In relation to these points, the authors of [11] conclude that Hill Climbing scales well with larger problem spaces, which causes this algorithm to have a big advantage over the other algorithms described in this thesis. There is, however, a big problem with Hill Climbing: being a pure Local Search algorithm, it uses no memory or Global Search techniques whatsoever, often causing it to not find the global optimum and instead having it get stuck in a local optimum. These characteristics of Hill Climbing (high speed, low memory, local search, and low probability of finding the optimal solution) contrast strongly with some of the more complicated algorithms, like a Genetic Algorithm. Genetic Algorithms work slower, use much more memory but have a higher probability of finding the optimal solution to a problem. Genetic Algorithms will be discussed in detail in the next section; for now, it is interesting to note that Hill Climbing and Genetic Algorithms seem to be opposite extremes in problem solving [31]. Renders and Bersini [31] had the idea to use the powers of both algorithms and create hybrid algorithms which should find a global optimum relatively efficiently. This idea and the algorithms it led to will be explained in more detail at the end of the section on Genetic Algorithms.

3.2.2 Hill Climbing for Job Shop Scheduling

Now that we have discussed the theoretical background of Hill Climbing, it is time to look at how this algorithm was used to solve the Job Shop Scheduling problem presented in this research. As discussed before, Hill Climbing works with a current solution, from which it creates a set of neighbouring solutions. This current solution is initially random ($createRandomSolution()$). Those neighbouring solutions look a lot like the current solution: they are different at only one point. So, what we have at the start of each iteration of the algorithm is a current solution in the form of a task assignment: an array of numbers, each indicating the specific agent carrying out the task of which the identifier equals the entry number of the agent number in the array. Creating a neighbouring solution is simple: simply change the agent number of one of the tasks, which comes down to changing one number in the task assignment. However, this way, creating *all* the neighbouring solutions could become impractical, since this number increases quite fast if the

problem size grows. Therefore, it is more practical to select a number of these neighbours. In this research, this is done as follows. Imagine the algorithm tries to find a neighbour by changing the first number in the task assignment. Also imagine we have to make a schedule for 10 workers. Now, the algorithm iterates over all possible agent numbers (9, not counting the current one used) and simply picks the first one that results in a solution better than the current solution. After all, we are looking for a solution better than the current one. If the algorithm finds such a number, it selects that mechanic as the new mechanic for that task and changes the current solution. The algorithm then moves on to the next task, repeating the previous process for this task. After the algorithm has gone through all the tasks, it simply starts over, with the first task. Indeed, a mechanic was already selected for this task, but a lot may have changed in the solution, so now, another mechanic might be the best fit. If there are no better mechanics to find for any of the tasks, the algorithm is in a local (and hopefully global) maximum, stops optimizing and returns the current solution. For a description of this Hill Climbing algorithm in pseudocode, see Algorithm 1.

Algorithm 1 Hill Climbing

```
Solution ← CreateRandomSolution()
while Stopconditions Not Met do

  CurrentFitness ← Fitness(Solution)
  for Each Task in Solution do

    CurrentMechanic ← Solution[Task]
    for Each Mechanic in Solution do

      Solution[Task] ← Mechanic
      if Fitness(Solution) ≤ CurrentFitness then
        Solution[Task] ← CurrentMechanic
      else
        {Better Solution Found}
        End For Loop
      end if
    end for
  end for
end while
```

3.3 Genetic Algorithm

3.3.1 Introduction

Greatly inspired by natural evolution, a Genetic Algorithm (pioneered by John Holland, [21]) uses a population of initially random solutions to a problem it is trying to solve [32]. Those solutions, being different from each other, all have their own fitness value, given by the objective function (or fitness function) of the Genetic Algorithm. At each iteration, the algorithm selects the n best solutions (i.e. the solutions with the highest fitness) and throws away the other ones. The n best solutions are allowed to generate offspring, for example by copying themselves and mutating the new solution, or by making a new solution by merging two existing ones. The original n best solutions are kept in the population. Newly created solutions will sometimes have higher fitness values than the original solutions (and often, they will be worse). This way, iteration after iteration, the population evolves, untill a plateau is reached (or a time limit is reached). Then, the best solution of the population is returned. This relatively simple algorithm has proven to be a very powerful strategy to solve very different optimization problems. This is at least partially because a Genetic Algorithm makes no assumptions about the problem it has to solve: one can add as many constraints and non-linearities as one likes, because given a big enough population, the algorithm will spawn some improved individuals. Part of the power comes from keeping multiple best solutions, not just one: this way, if the absolute best solution cannot create better ones, one of the other best solutions might. Of course, like with every algorithm, there is no free lunch: larger populations increase the algorithm's chance of finding the best solution, but make the algorithm slower. This is a problem since we want to have a solution fast. As we will see later, Memetic Algorithms, a variant of Genetic Algorithms, handle this problem quite well while keeping the ability to find good solutions.

3.3.2 Previous Work

Before we move on to how to create a Genetic Algorithm for our Job-Shop Scheduling problem, let us see what kind of optimization problems Genetic Algorithms have been used for before, just to give a general idea of their power. For our scheduling problem, multiple objectives have to be realized: the schedule has to be efficient, the mechanics should not have to work after hours, tasks should be finished in time, etcetera. It was chosen to create a single fitness function which takes all objectives into account. This results in the algorithms being more easy to create than in case it was chosen to keep the multiple objectives separate and create as many fitness functions. However, to demonstrate the power of Genetic

Algorithms, it is interesting to discuss previous research that did keep multiple objectives separate. This option was, for example, chosen in [5]: these researchers created a fast Genetic Algorithm that finds very good solutions in multiple dimensions. As they discuss, such multi-objective algorithms are in search of so-called Pareto-optimal solutions. Pareto-optimality is a state in which one cannot improve the solution in the dimension of one objective without making it worse in the dimension of another objective. Of course, as the reader can imagine, multiple Pareto-optimal solutions might exist to a problem, and without having more information, the algorithm cannot know which of those solutions is the "best" and thus has to return all of them [5].

3.3.3 Genetic Algorithm for Job Shop Scheduling

After the general theoretical background of Genetic Algorithms, let us now discuss how such an algorithm can be applied to the problem of Job Shop Scheduling. As said before, we start with a number (in this case 500) of initial solutions, produced randomly, called the population. These solutions again exist in the form of task assignments. We let the fitness function evaluate each of these solutions, after which we can keep the best ones. For this research, the number of task assignments kept is two, but more on this later. The algorithm deletes the other 498 solutions, and creates new ones. This is done as follows. Earlier in this thesis, we discussed the fact that sometimes Genetic Algorithms create new individuals by copying one of the best solutions, and sometimes, they take two of the best and merge them together to make a new one (by taking about half of the task assignment of the first solution, and half of the second). For this research, it was chosen to only do the first method: copying one solution. This choice was made because in general, the best solutions possible will have the workload divided approximately equally among all workers. That is one of the reasons the initial solutions are created randomly, as this will give such a distribution (most of the time). Taking two solutions and merging them together could easily create a new solution in which such an approximately equal distribution is destroyed. Therefore, it was chosen to always take one of the best solutions, copying it, and then mutate the newly "born" individual. Of course, this mutation process is the most important feature of the Genetic Algorithm: it is this process that creates the ultimately winning solution. A mutation can be one of the following two processes in this research. The first one is a simple random change: one of the tasks gets a randomly chosen agent (other than the one it has now), which simply means randomly changing one of the numbers in the task assignment. However, the task to which this is done is not completely randomly chosen: it is one of the tasks currently causing the schedule not to be optimal, for example by being finished too late. Choosing such tasks for the mutation process (and not tasks which cause no problems) makes the

algorithm a bit faster. The second mutation process is a swap: two tasks swap their executing agent. In such a swap, again at least one of the tasks is a task which currently causes a problem in the schedule. Forcing both taks of a swap to have that property would be too much, since there may be only one task in the whole task assignment that causes a problem. If the algorithm decides to try a swap, it tries to find two suitable tasks for a maximum of ten times before it gives up. Tasks not suitable for a swap are tasks which have at least one mechanic not qualified to do the other task. Each time a solution has to be mutated, there is a 20% probability the first mutation process is used and an 80% probability the second one is used.
The mutations just described cause the initial population of solutions to evolve. Each individual that needs to be mutated in a cycle is mutated once or twice, which is randomly decided (with fifty-fifty odds). The Genetic Algorithm described in this section is described in pseudocode in Algorithm 2 and Algorithm 3.

Algorithm 2 Genetic Algorithm

```
Solutions ← 500 random solutions
BestSolutions ← FindBestTwoIndividuals()
while Stopconditions Not Met do

  GenerateNewIndividuals(Solutions, BestSolutions)
  FindBestIndividuals(Solutions, BestSolutions)
  BestSolutions ← FindBestTwoIndividuals()
end while
return BestSolutions[0]
```

3.4 Memetic Algorithm: Genetic Algorithm with Hill Climbing

3.4.1 Introduction

As noted in for example [35], modern optimization algorithms combine exploration and exploitation: exploration means finding new regions in your search space, while exploitation is intensifying the search in a known region. Therefore, they say, it has been suggested to combine meta-heuristic strategies (for exploration) with local search algorithms (for exploitation). As we will see later in this thesis, such a combination leads to a class of very powerful algorithms.
As discussed before, the researchers of [31] decided to combine the powers of both

Algorithm 3 GenerateNewIndividuals(Solutions, BestSolutions)

```
for Each Solution in Solutions do

  if Solution not in BestSolutions then

    repeat

      if Random() < 0.2 then
        Mutate1(Solution)
      else
        Mutate2(Solution)
      end if
    until Repeated Twice or Random() < 0.5
  end if
end for
```

Genetic Algorithms and Hill Climbing, to create a hybrid algorithm. More specifically, hybrid algorithms created of Genetic Algorithms and local search methods (such as Hill Climbing) are called Memetic Algorithms. The authors of [31] succeeded in creating Memetic Algorithms with higher speed and no loss in performance compared to their Genetic Algorithm (indeed, their ultimate hybrid performed better). Let us look a bit at the theoretical background of Memetic Algorithms.

As noted in [30], the word "memetic" has the word "meme" in it, which is invented by the famous Richard Dawkins [4]. Dawkins [4] explains that a meme can be, for example, a catch-phrase, or an idea, which propagates itself from human to human via imitation much like genes propagate themselves via sperm and eggs. Memetic Algorithms have the word "meme" in them because they use heuristics [30], which are ideas to improve performance, the heuristic in the case of this thesis being a local search method.

3.4.2 Previous Work

To discuss the strength of Memetic Algorithms, let us first look at research performed by Zitzler and Thiele [36]. In this research, the authors developed Evolutionary Algorithms to solve a special kind of optimization problem: the Multi-Objective 0/1 Knapsack Problem. As noted in [36], the problem is NP-hard, though easy to describe. What we have is a generalization of the standard 0/1

Knapsack Problem, which works as follows. We have a knapsack which can hold a number of items, which together must not exceed a weight maximum. Furthermore, we have a number of items, each with a weight and a profit. The purpose is to fill the knapsack with items in such a way that the maximum weight is not exceeded and the total profit is maximized. To extend this 0/1 Knapsack Problem to the Multi-Objective 0/1 Knapsack Problem, Zitzler and Thiele [36] simply allow an arbitrary number of knapsacks. This results in multiple total profits to be maximized, hence the multi-objective property. Zitzler and Thiele [36] developed an Evolutionary Algorithm called SPEA to solve the Multi-Objective 0/1 Knapsack Problem. The details of the inner workings of SPEA are beyond the scope of this thesis; more important is their conclusion that SPEA outperformed other existing Evolutionary Algorithms, while even those algorithms are quite suitable for multi-objective optimization problems [36].
Now that we have discussed the Multi-Objective 0/1 Knapsack Problem, let us discuss the research performed by Knowles and Corne, described in [28]. These authors took the Multi-Objective 0/1 Knapsack Problem described in [36], and developed their own Memetic Algorithm to solve it. This Memetic Algorithm, called M-PAES (Memetic Pareto Archived Evolutionary Algorithm), is a combination of their earlier developed (1+1)-PAES [27] and population and recombination techniques. (1+1)-PAES is a relatively simple to create local search algorithm [27], intended to be used as a baseline to compare more complicated algorithms with. However, it was found to be quite competitive with respected Evolutionary Algorithms, altough the experimental settings required further investigation [27]. Again, the details of (1+1)-PAES are not relevant for this thesis. What is relevant is the Memetic Algorithm, M-PAES, that incorporated it. M-PAES was found to be competitive with and on some problems even better than SPEA [27]. It is noted [27], however, that comparison between the algorithms is difficult and that further investigation is needed. That makes the comparison between Memetic Algorithms and Genetic Algorithms in this thesis (even) more relevant.

3.4.3 Memetic Algorithm for Job Shop Scheduling

Of course, both speed and performance are very interesting for the research described in this thesis, but it is mostly the speed part that is of importance. Genetic Algorithms are by themselves not very fast algorithms, and it is of great importance that an algorithm is created that will not only find a good solution to quite large scheduling problems, but does so in a matter of seconds, and not minutes. When things do not work out as planned in a car workshop, a new planning needs to be generated as fast as possible, especially when mechanics are waiting for a job to be assigned to them. Because of this, for this research too, a Memetic Algorithm was implemented. The original Genetic Algorithm, described in the

previous paragraph, was used and adapted. This was done as follows. After the Genetic Algorithm has found the best two candidate solutions in its population, those two (and only those two) are optimized using Hill Climbing. Optimizing all the solutions in the population would take too much time, and would take away the advantage we are trying to establish by using Hill Climbing in the first place. What is more, the Hill Climbing part does not (at least not necessarily) fully optimize the solutions it works with: it only does two iterations. Again, the reason is simply time constraints. Moreover, using Hill Climbing on solutions worse than the two best solutions will generally result in new solutions worse than the solutions generated by Hill Climbing on the two best solutions, and will thus not be very useful.
The Genetic Algorithm which forms the basis for the Genetic Algorithm with Hill Climbing is described with pseudocode in Algorithm 2 and Algorithm 3. For the Genetic Algorithm with Hill Climbing, Algorithm 3 is replaced with Algorithm 4. Also, Algorithm 5 is added to show the procedure of optimizing the best two solutions with Hill Climbing in pseudocode.

Algorithm 4 GenerateNewIndividualsMemetic(Solutions, BestSolutions)

```
for Each Solution in Solutions do

  if Solution not in BestSolutions then

    repeat

      if Random() < 0.2 then
        Mutate1(Solution)
      else
        Mutate2(Solution)
      end if
    until Repeated Twice or Random() < 0.5
  else
    Solution ← OptimizeWithHillClimbing(Solution)
  end if
end for
```

Algorithm 5 OptimizeWithHillClimbing(Solution)

```
while Stopconditions Not Met do

  CurrentFitness ← Fitness(Solution)
  for Each Task in Solution do

    CurrentMechanic ← Solution[Task]
    for Each Mechanic in Solution do

      Solution[Task] ← Mechanic
      if Fitness(Solution) ≤ CurrentFitness then
        Solution[Task] ← CurrentMechanic
      else
        {Better Solution Found}
        End For Loop
      end if
    end for
  end for
end while
```

3.5 Ant Colony Optimization

3.5.1 Introduction

Invented by Marco Dorigo for his PhD-thesis [7], Ant Colony Optimization can be seen as a member of the swarm intelligence methods [9]. According to [24], the defining properties of any swarm intelligence algorithm are that local rules and interactions between self-organizing agents cause the emergence of a collective intelligence. The local rules have no relation to the global pattern. These properties, as we will see, clearly hold in Ant Colony Optimization. As the name suggests, this algorithm is inspired by the behavior of ants. As described in for example [9], ants who are looking for the shortest path to a food source use a process called stigmercy (first described by [18]). Stigmercy is a communication process in which ants communicate with each other with locally released (and only locally receivable), non-symbolic information; this information is released in the form of pheromones, which is smelled by ants and constantly released by each ant. Pheromones evaporate over time. Ants choose where to go based partially on the pherome concentrations around them: the higher the concentration, the more likely it is ants will go there. Imagine there are two paths to a food source, a long path and a short one (see for such an experiment [17]), which split at a

certain obstacle, at which each ant has to choose to go left or to go right. Since the ants have no prior knowledge of which path will be the shortest, initially, each ant will choose left or right at random. This results in both paths initially being walked by roughly the same amounts of ants, and thus both paths will (again, initially) receive the same amount of pheromones. However, over time, this results in the shorter path having a higher pheromone concentration: because the path is shorter (and the number of ants walking it is inititially the same), it is walked by more ants per time unit. Because of this, more and more ants will walk the shorter path, and thus the group of ants as a whole demonstrates optimization. It is important to note, however, that even in case the two paths in the previously described problem are of equal length, the ants will still all follow the same path after a while, due to early random fluctuations in the pheromone concentrations [6].

As noted before, the ants together show intelligence when finding paths to a food source: they are optimizing with the use of stigmercy. Because of this optimizing, they serve as inspiration for the class of algorithms known as Ant Colony Optimization algorithms. What these algorithms all share is the use of swarm intelligence and the release and reception of virtual pheromones to simulate stigmercy, which is used to optimize the solutions to optimization problems such as the scheduling problem discussed in this thesis. As [3] notes, it is (partly) this swarm intelligence characteristic that gives Ant Colony Optimization its strength in solving optimization problems. The domain to which Ant Colony Optimization applies is huge: the algorithms are applicable to all discrete optimization problems, if there is some kind of solution-creating procedure possible [8]. The general optimization process of Ant Colony Optimization may be best explained when applying it to an example problem. For this, the famous Traveling Salesman Problem is chosen. As explained before, this problem has a number of cities with specific distances between each pair of cities, and a salesman who has to visit each city exactly one time and has to return to the first city he visited afterwards, creating the shortest possible route doing so. As the general form of the Ant Colony Optimization algorithms is described in [9], such an algorithm would repeatedly follow the following steps: construct ant solutions, apply local search, such as Hill Climbing (this is optional), and finally update the pheromone values. Technical details will be discussed in the paragraph about Max-Min Ant System, one of the variants of Ant Colony Optimization (indeed, the details depend on the exact algorithm), but let us look at the first and last steps in a little bit more detail. Constructing ant solutions happens in the following way. Each virtual ant creates a solution to the Traveling Salesman Problem, traveling each city and returning to the first one. While creating such a tour, at each city, an ant has to choose which city to visit next. It does so probabilistically, the probability to visit each city depending on

the pheromone value on the path to that city: the higher that value, the higher the probability. The final step of an iteration of Ant Colony Optimization is the updating of the pheromone values between the cities. Again, the technical details will be discussed later; for now, the update of a pheromone value depends on how good the solutions were containing the path of that pheromone value. This way, over time, good solution parts will receive more and more pheromone value, and will be used more and more in new solutions, optimizing in the process.

3.5.2 Ant System

Having discussed the general way in which Ant Colony Optimization algorithms solve a discrete optimization problem, it is time to turn to a more precise and more technical description of a specific algorithm which is the father of the algorithm used in the research described in this thesis: the algorithm called Ant System. This is the first algorithm in the Ant Colony Optimization class [9]. An important feature of Ant System is the fact that it lets all of its ants update the pheromone values, and not just for example the ant with the best tour [9]. The pheromone update is governed by the following update rule [9]:

$$\tau_{ij} \leftarrow (1-\rho) * \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

In this equation, τ_{ij} is the pheromone on the edge joining cities i and j, ρ is the evaporation factor (causing pheromone values to decline) and $\Delta\tau_{ij}^{k}$ is the amount of pheromone per unit length ant k deposits on the edge joining cities i and j. Thus, this equation simply tells us that the pheromone values on each edge in the graph of the Traveling Salesman Problem are increased by all the amounts of pheromone dropped on it by the ants (and decreased slightly by the evaporation factor ρ). Now, let us define the pheromone update $\Delta\tau_{ij}^{k}$ (as in [9]):

$$\Delta\tau_{ij}^{k} = \begin{cases} \text{If ant } k \text{ used edge } ij \text{ in its tour} & \frac{Q}{L_k} \\ \text{Else} & 0 \end{cases}$$

In this equation, L_k is the length of the tour made by ant k, and Q is simply a constant defined by the programmer. In summary, all the edges that were used by one or more of the ants in its construction of its solution get a positive pheromone update which is inversely proportional to the lengths of the tours that edge is in.

Now that we have discussed the pheromone update rules, let us go on to define how the ants construct their solutions. At each step in creating a solution for the Traveling Salesman Problem, an ant has to choose which of the remaining cities it is going to visit next. As discussed before, this is done probabilistically. The probability of ant k visiting city j when currenty it is in city i, p_{ij}^k, is defined as follows [9]:

$$p_{ij}^k = \begin{cases} \text{if } j \in allowed_k & \frac{\tau_{ij}^\alpha * \eta_{ij}^\beta}{\sum_{l \in allowed_k} \tau_{il}^\alpha * \eta_{il}^\beta} \\ \text{Otherwise} & 0 \end{cases}$$

Here, η_{ij} is the heuristic information on edge ij, equal to the reverse of the length of this edge: $\eta_{ij} = \frac{1}{d_{ij}}$. $allowed_k$ is the list of cities that have not yet been visited by ant k. α and β are simply parameters governing the relative influence of the pheromones and the heuristic information, respectively.
Note that Ant System uses a greedy heuristic (the distance between two cities in case of the Traveling Salesman Problem), which by itself usually leads to relatively bad solutions [3]. The Ant System still converges to a good solution (most of the time). This is very interesting indeed; the use of a simple, usually bad working heuristic applied multiple times results in a good solution. Of course, the reason for this is the powerful (though simple) communication between the ants.

Before we continue to the paragraph about the Max-Min Ant System algorithm (which was used for the Job-Shop Scheduling problem described in this thesis), let us look at some problems the Ant System algorithm has been applied to in the past. As described in [3], Colorni et al. have successfully applied the Ant System for a Job-Shop Scheduling problem, very similar to the one described in this thesis. As said before, one of their conclusions was that the strength of the Ant System algorithm comes from the swarm characteristic. More specifically, the algorithm is successful because of the cooperation among the ants, which is done using the release and reception of pheromones. Looking at one ant, the tour it creates is (usually) in part good and in part bad; this is because it uses the greedy policy. Overall, the individual ant's solution is therefore not very good. However, there are multiple ants all creating different tours. All these tours have good parts and bad parts. Overall, good parts will be chosen more often than bad parts since their pheromone trails will be and will *become* the highest (positive reinforcement), because they are more often in good tours. The process just described is called a *probabilistic superposition of effects* [3]. The term probabilistic is already explained; the superposition term comes from the fact that the strength of the Ant System lies not in the individual ant, but in the cooperation

of the group.

3.5.3 Max-Min Ant System

As the name already suggests, Max-Min Ant System [34] is a more specific variant of the more general Ant System algorithm. Max-Min Ant System introduces two changes to the original algorithm. The first change is the fact that only the best ant can update the pheromone trails, in contrast to the property of Ant System that *all* ants update these trails. Secondly, Max-Min Ant System does have, as the name suggest, minimum and maximum pheromone values. The equation for pheromone updates thus becomes [9]:

$$\tau_{ij} \leftarrow (1 - \rho) * \tau_{ij} + \Delta\tau_{ij}^{best}$$

However, it τ_{ij} drops below the minimum value or becomes higher than the maximum value, it will automatically become equal to the minimum or maximum value, respectively. Apart from these few differences, the Max-Min Ant System works the same as the Ant System.

3.5.4 Max-Min Ant System for Job Shop Scheduling

In the previous paragraphs, we have discussed how Ant Colony Optimization in general solves optimization problems, and more specifically how the Max-Min Ant System algorithm, a variant of the Ant System, does this. Now that we have discussed its pheromone update rules and its move selection probability equations, let us look at how the Max-Min Ant System algorithm can be applied to solve the Job Shop Scheduling problem defined in this thesis. As with the other algorithms, solutions are coded as task assignments: arrays of numbers in which the index is symbol for a task and the value stands for the agent (mechanic) performing that task as defined by that specific solution. The construction of such a task assignment by an ant in the Max-Min Ant System is a lot like the creation of a tour in the Traveling Salesman Problem, which we discussed earlier. Instead of traveling from one city to the next to create a tour, as happens in the Traveling Salesman Problem, in the Job Shop Scheduling problem an ant "travels" from one task-agent combination to the next. The "distance" between such task-agent combinations is defined as the time it takes the agent to perform the task. At the start, the ant chooses the agent for task 0. It can choose from all agents qualified to perform that task. The choice of such a task-agent combination happens according to the probability equation described in the previous paragraph, and is exactly

like the choice of a city in the Traveling Salesman Problem. After the choice of an agent for task 0, the ant moves on to choose an agent for task 1, etcetera. After all ants have assigned an agent to each task, their tours are evaluated according to the fitness function. The best created tour since the start is kept in memory. After this, the pheromone updates happen. After this, the whole process starts over, until the solutions converge. There is, however, a problem which arises when we compare the application of a Max-Min Ant System to the Traveling Salesman Problem to the application of a Max-Min Ant System to Job Shop Scheduling. The fitness of a solution to the Traveling Salesman Problem is inversely related to the total distance travelled, and is completely determined by this. In contrast, the fitness of a solution to the Job Shop Scheduling problem has a more complicated calculation: it is indeed related to the total "distance travelled", but it also depends on how much overtime each agent has, the degree to which each task is finished before its deadline, etcetera. This makes the "distance" a weak heuristic in the case of the Job Shop Scheduling problem, while it is more than that in the Traveling Salesman Problem. Since the pheromone updates are done using the fitness function (and are thus related to overtime etcetera), this might negatively influence the optimization process.

The parameters of the Max-Min Ant System are set as follows. First of all, the α and β parameters are both equal to 1.0. The minimum pheromone value equals 0.01, while the maximum value is 1.00. The Q-factor is set to 0.5 and ρ equals 0.08. There are 200 ants constructing solutions. The Max-Min Ant System described in this section is more formally described in pseudocode in Algorithm 6 and Algorithm 7.

Algorithm 6 Max-Min Ant System

```
while Stopconditions Not Met do

  UpdateProbabilities()
  CreateTours()
  BestTour ← FindBestTour()
  UpdatePheromoneValues()
end while
return BestTour
```

Algorithm 7 CreateTours()

```
for Each Ant in Ants do

  for Each Task in Tasks do

    CumulativeProbability ← 0.0
    RandomFloat ← Random()
    for Each Mechanic in Mechanics do

      CumulativeProbability ← CumulativeProbability + Probabilities[Task][Mechanic]
      if RandomFloat < CumulativeProbability then
        Tours[Ant][Task] ← Mechanic
      end if
    end for
  end for
end for
```

3.6 Max-Min Ant System Combined with Hill Climbing

As suggested by the general Ant Colony Optimization algorithm (see for example [9]), the Max-Min Ant System presented in the previous subsection was combined with local search in the form of Hill Climbing. The general idea is to simply improve each ant's tour after it has created one by itself, but, of course, before the pheromone values are updated. However, again, because of time constraints, it was decided to only improve the best tour with Hill Climbing, not all the tours. Again, the Hill Climbing component runs for a maximum of two rounds.

3.7 Particle Swarm Optimization

3.7.1 Introduction

In its standard form, Particle Swarm Optimization [25] uses continuous variables. Of course, for the problem at hand, we need algorithms that use discrete variables; hence, an adapted version of the standard algorithm was needed, which is described later in this section. First, we will look at the basic, continuous variables using algorithm.

Just like Genetic Algorithms and Ant Colony Optimization algorithms, Particle Swarm Optimization, invented by Kennedy and Eberhart [25], is inspired by

nature. This algorithm works in analogy with the way birds or fish find a food source: once one of the birds or fish finds food, shortly after many others come to the same place. In the same way, Particle Swarm Optimization starts with a large number of particles flying around in the problem space, defined along a number of dimensions. While initially being random, the velocities of the particles are updated each cycle using information found so far about where the global best solution might be [33]. In the basic form of the algorithm, this information for each particle comes from two variables: the best found solution so far of the particle itself, and the best found solution so far of all particles. The velocity (more precise, the speed and direction) of each particle is then adjusted in order to have it move more in the direction of those two variables. This velocity update is done for each axis individually. The positions of the particles on each axis are then updated using their new velocities. The equations [33] for the velocities and positions of the particles are as follows:

$$v_{id} = v_{id} + c_1 * rand() * (p_{id} - x_{id}) + c_2 * Rand() * (p_{gd} - x_{id})$$
$$x_{id} = x_{id} + v_{id}$$

Here, v_{id} is the velocity of particle i in dimension d and x_{id} is the position of particle i in dimension x. p_{id} is the best position reached so far in dimension d by particle i, and p_{gd} is the best position reached overall, again in dimension d. c_1 and c_2 are constants regulating the influence of the previous two variables, whereas $rand()$ and $Rand()$ are two functions generating random numbers between 0 and 1.

3.7.2 Discrete Particle Swarm Optimization

As discussed before, the standard Particle Swarm Optimization algorithm solves problems with continuous variables. In the scheduling problem the optimization algorithms use task assignments (where each task has a specific mechanic who performs that task), and these assignments are, of course, discrete. We therefore need an adapted algorithm: Discrete Particle Swarm Optimization. In [23], such an algorithm is used to solve the Grid Job Scheduling problem, a problem similar to Job Shop Scheduling. What is most interesting in their approach is not the position of the particles, but their velocities. The problem is that using standard, continuous domain velocities leads to continuous positions, which then have to be rounded to whole numbers. Even worse, velocities in the classic sense do not make sense here: moving "towards" a previously encountered good solution will usually not give the particle interesting new positions, since in the problems we are discussing, for example resources are randomly ordered in the resource dimension. The author therefore follows the approach of [23] and models velocities as probabilities. In

this approach, each particle has a probability distribution for each dimension of the problem space. The position of a particle on each axis is therefore determined probabilistically, using this probability distribution. Two Discrete Particle Swarm Optimization algorithms were created modeling velocities as probabilities. The first one works as follows. For each particle, its best position in the problem space so far is kept in memory. The same is done for the best position of all particles. The probabilities are calculated as follows. Each particle gets is own probabiliy distribution, initially (in each iteration) giving each coordinate the same probability along each axis (except for those places where the probability is zero, caused by an agent being unqualified for that task). Then, the best position of each particle so far updates the probability distribution of that particle by increasing the probabilities of the coordinates of its best task assignment so far with a factor: if its best assignment so far has value 4 in entry 3, then the probability of value 4 in entry 3 is increased. The same goes for the best task assignment overall so far, with the difference that this one updates the probability distributions of all particles. Furthermore, another difference is that the best task assignment overall increases probabilities with a bigger factor than the particles' own best task assignments do. Of course, the probability distributions are normalized after the updates, as they have to add up to 1.
The second way probabilities are calculated in this thesis works almost the same as the first one, with one major difference: the best task assignments of each individual particle update *all* probability distributions, not just their own. The factor with which this best task assignments update the probability distributions has to be small here, since each probability distribution now gets a great number of updates from such task assignments, this number being equal to the number of particles. If this factor is too big, the overall best solution will hardly have any influence. While calculating the probabilities the second way may be less popular, it does give the word "probability" more meaning per iteration: giving a, say, 0.01 probability update to all particles usually actually results in around one percent of the particles changing their state according the update. This alternative algorithm will be called Particle Swarm Optimization 2, as opposed to the first Particle Swarm Optimization algorithm, called Particle Swarm Optimization 1.

3.7.3 Discrete Particle Swarm Optimization for Job Shop Scheduling

Now that we have discussed the general workings of the Discrete Particle Swarm Optimization algorithm designed for the Job Shop Scheduling problem, it is time to look at how to apply the algorithm to the problem. As discussed before (and as always) task assignments are used to encode solutions. Each particle therefore exists as a task assignment, with each index representing a task and each value representing the agent performing that task. As discussed before, these particles

"fly around" in the problem space, with multiple dimensions. If particles exist as task assignments, each dimension represents a specific task. The agents are modeled along each axis. This way, a particle being five steps away from the origin on the axis belonging to task 2 is a task assignment assigning agent 6 to task 2, as the origin represents agent 0.

For the Discrete Particle Swarm Optimization algorithms, the parameters are set as follows. For Particle Swarm Optimization 1, the best task assignments of each individual particle multiply the probabilities corresponding to that task assignment with 1.003. This factor is set to 1.0003 for Particle Swarm Optimization 2. The corresponding factor of the best overall task assignment for the two algorithms is set to 1.3 and 1.03, respectively. The swarm sizes are set to 100 for each algorithm. Furthermore, it should be noted that in Particle Swarm Optimization 2, all probabilities are reinitialized at the beginning of every cycle, while this is not the case in Particle Swarm Optimization 1. Furthermore, Particle Swarm Optimization 2 sets the probability for mechanics not qualified for a task to zero, which Particle Swarm Optimization 1 does not. Therefore, Particle Swarm Optimization 2 could be seen as an updated version of Particle Swarm Optimization 1.
Algorithm 8 describes both Particle Swarm Optimization 1 and Particle Swarm Optimization 2 in pseudocode.

Algorithm 8 Discrete Particle Swarm Optimization

```
Particles ← 500 random solutions
while Stopconditions Not Met do

  CalculateParticleFitnesses()
  for Each Particle in Particles do

    UpdateIndividualBestPosition(Particle)
  end for
  CalculateOverallBestPosition()
  UpdateParticleVelocities()
  UpdateParticlePositions()
end while
```

3.8 Discrete Particle Swarm Optimization Combined with Hill Climbing

3.8.1 Introduction

As noted before, combinations between meta-heuristic algorithms and local search techniques often lead to powerful new algorithms (see for example [35]), such as Memetic Algorithms, discussed in a previous section. Since this approach seems so powerful, it was decided to combine the Particle Swarm Optimization variants used in this research with the Hill Climbing algorithm discussed earlier, just as was done with the Genetic Algorithm. A (brief) description of how this was done will be given in the next paragraph; for now, let us have a look at the research performed by [35]. They created a hybrid algorithm, Particle Swarm Optimization combined with Hill Climbing, for a problem in multiprocessing in computation systems. More specifically, this problem is called the Task Assignment Problem and is about optimizing the assignment of tasks that need to be executed to the distributed processors in order to minimize the costs associated with the execution of the task assignment and the communication between the tasks. The hybrid algorithm [35] proposed to solve the Task Assignment Problem is just as one would expect: what they have is a Particle Swarm Optimizer, and after each iteration, each particle's position is improved using a Hill Climbing optimizer. Interestingly, [35] compared their hybrid optimizer to a Genetic Algorithm and found that their hybrid algorithm was faster and more scalable, i.e. the difference in speed increased with the problem size.

3.8.2 Discrete Particle Swarm Optimization Combined with Hill Climbing for Job Shop Scheduling

After the general introduction on how to combine Particle Swarm Optimization with Hill Climbing, it is time to discuss how this was done in this research. This approach is basically the same as the one discussed in [35], except that again, only the best particle is updated with Hill Climbing. At the end of each iteration, after each particle has its position updated, its position is improved with the Hill Climbing algorithm described earlier. Note that Hill Climbing again runs for only two cycles, due to time constraints.

3.9 Tabu Search

3.9.1 Introduction

With the obvious exception of Hill Climbing, Tabu Search is the only algorithm discussed in this thesis that is classified as a local search algorithm. Invented in 1986 by Fred Glover (see for the original paper [13]), it was formalized three years later [14] and [15]). As noted in [14], Tabu Search has the ability to use specialized heuristics and to direct them in order to overcome the limitations of local search. This is, of course, an extremely interesting property. In Artificial Intelligence, algorithms are often classified as a local search *or* a global search algorithm, and this classification, in its pure form, is still applicable to Tabu Search. However, classifying this algorithm as local search (which was done at the start of this chapter and is technically correct) might suggest it has the same limitations as other local search algorithms, such as Hill Climbing, often have, the most important limitation probably being their tendency to get stuck in local optima. However, as Glover [14] states, Tabu Search overcomes the limitations other local search algorithms have. As we will see later, it still has the powers of a local search algorithm: high optimization speed and low memory usage (although it needs a little bit more memory than Hill Climbing).
To give an impression of what Tabu Search is capable of, here is a list of examples of problems Tabu Search has been applied to in previous research [16]: employee scheduling, maximum satisfiability problems, space planning and architectural design, Job Shop Scheduling, probabilistic logic problems, neural network pattern recognition, quadratic assignment problems, the Traveling Salesman Problem, graph coloring and Flow Shop Sequencing.
After the very general introduction on Tabu Search that is just given, let us discuss how the algorithm solves a problem. As all the other algorithms in this research, Tabu Search is an optimization algorithm, meaning it tries to find the solution with the highest value according to some objective function, which in this thesis is called the fitness function. Let us discuss a simple form of Tabu Search, which starts with a simple heuristic we have seen before, be it in another form: the Hill Climbing heuristic. Glover [14] describes the first steps of the heuristic as follows. The algorithm starts with an initial solution to the problem. Using this solution, it creates a neighbourhood of new solutions. Note that Glover does not describe how to create such a neighbourhood, which is apparently up to the creator of the simple Tabu Search variant. Now, after the creation of the neighbourhood, the next step is to select the neighbour whose penalty is the lowest (and thus whose fitness is the highest) of all the neighbours [16]. In [14], a version of Hill Climbing that does not require the *best* neighbour to be selected is described; just *a* neighbour with a better fitness than the current solution is required. For the research

in this thesis, the first variant was chosen. Back to the algorithm itself, if the best neighbour is not better than the current solution, then the algorithm is in a local (and hopefully global) maximum and terminates. If such a neighbour does exist, the current solution is replaced by one of those neighbours with better fitness values, after which the next iteration of the algorithm can begin.
Now that we have discussed a simple heuristic with which Hill Climbing can work, let us go on to discuss what the Tabu Search algorithm actually looks like using this heuristic. To do this, it may be best to start with the previously discussed problem of Hill Climbing: its ability to get stuck in local optima. The strength of Tabu Search partially comes from the fact that it does not stop if it has arrived in such a place in the fitness landscape: in each iteration, the algorithm simply chooses its best neighbour, whether this neighbour is better than the current solution or not. This alters the algorithm's termination criterium, since it can in principle infinitely continue to look for new solutions. Tabu Search stops after a determined number of steps or when a certain penalty or fitness threshold is met.
The sharp reader might have noticed another problem which arises when the best neighbour is not required to be better than the current solution in order to be chosen as the next current solution: the possibility of infinite switching between two (or more) solutions. Imagine a landscape with solutions s_0 and s_1, which are neighbours of each other. Furthermore, s_0 has neighbours n_0^0 and n_1^0, which are not neighbours of s_1. s_1 has neighbours n_0^1 and n_1^1, which are not neighbours of s_0. s_0 and s_1 both have fitness value 1, while n_0^0, n_0^1, n_1^0 and n_1^1 all have fitness value 0. If the current solution is s_0, the algorithm will move to s_1 since this is the best neigbhour of s_0. After this, Tabu Search will move back to s_0, because this is the best neighbour of s_1. Then it will again move to s_1, then back again to s_0, indefinitely looping and never finding new and possibly better solutions. To prevent this, Tabu Search has its most important feature: the tabu moves. As the term suggests, tabu moves are moves the algorithm is not allowed to take (unless in some specific cases, in which aspiration criteria hold, which we will see later) [16]. Having moves marked as tabu can be realized in the form of a tabu list, a kind of short term memory which stores certain solutions that may not be visited. Such solutions are stored because they meet the tabu conditions [14], and are kept till new "memories" push them out of the finite tabu list. The tabu conditions can simply be whether the solution is visited recently or not. Going back to the infinite looping between two (or more) solutions, such tabu conditions would prevent this from happening. Visiting s_0 would put s_0 in the tabu list, so when the algorithm visits s_1 after, it will not move back to s_0 since s_0 is marked tabu. That is how Tabu Search is able to navigate the search space efficiently, and has a relatively low probability of getting stuck in a local maximum.
Coming back to the aspiration criteria, these can be interesting when tabu condi-

tions prevent a move to a solution which would be the best one found so far. In such a case, for example, aspiration criteria could overrule the tabu restrictions [16]. As noted in [16], the core of Tabu Search is its short term memory. This component can be summarized as follows, which immediately gives us a simple Tabu Search algorithm:

1. Begin with a starting current solution

2. Create a candidate list of solutions from the current solution: the neighbourhood

3. Choose the best admissable neighbour as the new current solution

4. If the stopping criterium holds, stop;
   else, update the Tabu Restrictions and Aspiration Criteria and start again

3.9.2 Tabu Search for Job Shop Scheduling

In the previous section, an overview of the Tabu Search algorithm was given, along with a general description of how the algorithm works. Now that this is done, as with the other algorithms, it is time to discuss how Tabu Search can be applied to the Job Shop Scheduling problem described in this thesis. However, before we go any further, note that with the Tabu Search algorithm applied here, the aim has been to create an algorithm as simple as possible, to contrast it with other more complex algorithms described in this thesis, like the Max-Min Ant System and the Discrete Particle Swarm Optimizers. Simplicity can lead to good results, and it is always important to question the usefulness of using a more complex algorithm. Hence, in this research, some more complex algorithms are compared to a simple (though quite powerful) variant of Tabu Search. Now that the reader understands the motive behind this algorithm, we can continue to describe it.

Basically, the Tabu Search variant used here is a Hill Climber with a short term memory (storing ten of the most recent solutions) and the ability to choose positions with a fitness value *not* higher than that of its current position. The algorithm starts with a random solution, again in the form of a task assignment: an array of numbers representing the mechanics performing the task corresponding to the array index. It then creates the neighbourhood: solutions similar to the current one, but differing a little bit. The neighbours it creates are all solutions with differ in one entry of the task assignment (meaning one task is performed by another mechanic) and all solutions which have two entry values swapped (meaning two tasks have swapped their mechanics). The algorithm moves to the best neighbour,

Algorithm 9 Tabu Search

```
CurrentSolution ← CreateRandomSolution()
while Stopconditions Not Met do

  CurrentSolution ← FindBestNeighbour(Tabulist)
  if CurrentSolution not in Tabulist then
    Append CurrentSolution to TabuList
  end if
  if Size(Tabulist) > TabulistMaxSize then
    Delete(Tabulist[0])
  end if
end while
return CurrentSolution
```

and stores the previous solution in the short term memory. Algorithm 9 describes this Tabu Search algorithm in pseudocode.

Chapter 4

Experiments and Results

4.1 Experiments

In the previous chapters, a broad description of the scheduling problem was given, as well as descriptions of a number of algorithms and how they are used to solve this scheduling problem. Now, it is time to discuss what experiments were performed to determine which of these algorithms is the best fit for our scheduling problem. Of course, "best" is not that easily defined: both the schedule quality *and* the time it takes an algorithm to make a good schedule are important. To this end, two experiments were designed. In the first experiment, 4 sets of 50 problems are created by the problem generator: one set with problems of 25 tasks, one with problems of 50 tasks, one with problems of 75 tasks and one with problems of 100 tasks, for a total of 200 problems for each algorithm. The algorithms have a maximum of two minutes to create a reasonable solution for each problem, and their average duration and standard deviation over all 50 problems, for each problem size, are compared, as well as the intra-algorithm scoring difference for different problem sizes. "Reasonable" means "containing no mistakes", which are tasks being performed after their maximum finish time and mechanics having to work after hours. Efficiency (total work hours of a solution) does not count in this experiment. As a baseline algorithm, a random schedule generator was created. All this algorithm does is creating a thousand random schedules and return the one with the highest fitness value. This algorithm is included to compare the more complicated algorithms to.

The second experiment has more focus on performance than on speed. Here, 10 problems are created and given to each algorithm, but now the algorithms have 10 minutes for each problem. Their score (fitness) after these 10 minutes is compared. Speed does not count here: it is all about creating the most efficient schedule containing no mistakes.

In both experiments, each problem description contains a number of bridges, with is dependent on the number of tasks in that problem: the number of bridges is equal to one sixth of the number of tasks, and one half of the tasks actually needs a bridge. Each task needs exactly one car part, the stock of which is refilled in the middle of the day. Mechanics are qualified for at most half of the tasks of the problem (the exact number being randomly decided). Furthermore, each mechanic has its own time duration specifications for each task. These durations are taken from a dataset provided by Bungert GMBH & CO. KG, and are values ranging between 12 minutes and 174 minutes and are mutated a little (minus zero minutes, ten minutes, or twenty minutes for task durations longer than twenty minutes) for each mechanic in order to get differences between them. The tasks each have a 80% probability of having another task which has to be finished before it can start. Finally, there is only one type of car part, of which the stock is initially 100. Each task uses exactly 1 car part, and the stock is increased by 100 at the exact middle of the day.

4.2 Results

4.2.1 Experiment 1: Speed

As explained before, the first experiment was designed to see how fast the different algorithms can produce a reasonable schedule. The results of Experiment 1 can be found in Table 5.1 and Table 5.2. Because not all the algorithms were able to create a reasonable solution for each of the problems, the percentage of times they did find one is included in the table as the fourth column. For a more visual representation, each algorithm's performance is visualized in a graph (Figures 5.1-5.6). The rounds on the x-axis of these figures refer to the cycles of the algorithms, which are described in their pseudocode.
The first conclusion is that the Random Scheduler does not perform well. For the 25 tasks problems it performs more or less comparable to Hill Climbing, but in the other three categories it is far worse worse than Hill Climbing (which itself does not perform well). These results tell us that more sophisticated algorithms are necessary to tackle the scheduling problem. Looking at the more complicated algorithms, the Genetic Algorithm with Hill Climbing is quite clearly the best in this experiment. Not only does it produce reasonable schedules in almost all of the cases (missing only 4% in the two largest problem classes), it is also one of the fastest algorithms on all problem sizes, and the absolute fastest on the two largest problem classes. Furthermore, it is surprising how much better the Genetic Algorithm performs if the Hill Climber is added. On the 100 tasks problems, the

Genetic Algorithm only produces a reasonable schedule 20% of the time, a number that increases to 96% if the Hill Climber is added. This performance increase due to an added Hill Climbing component is true for every algorithm with this option, except for Particle Swarm Optimization 2 on the 50 tasks problems. Also, adding a Hill Climber does not increase the performance of the Max-Min Ant System on the larger problem classes, but this is a clear bottom effect.
Another conclusion is that Particle Swarm Optimization 2 clearly performs better than Particle Swarm Optimization 1, although the difference is less clear when both algorithms add a Hill Climber. A last conclusion is that Tabu Search performs surprisingly well for such a simple algorithm; the performance does drop, however, when the problems become larger.

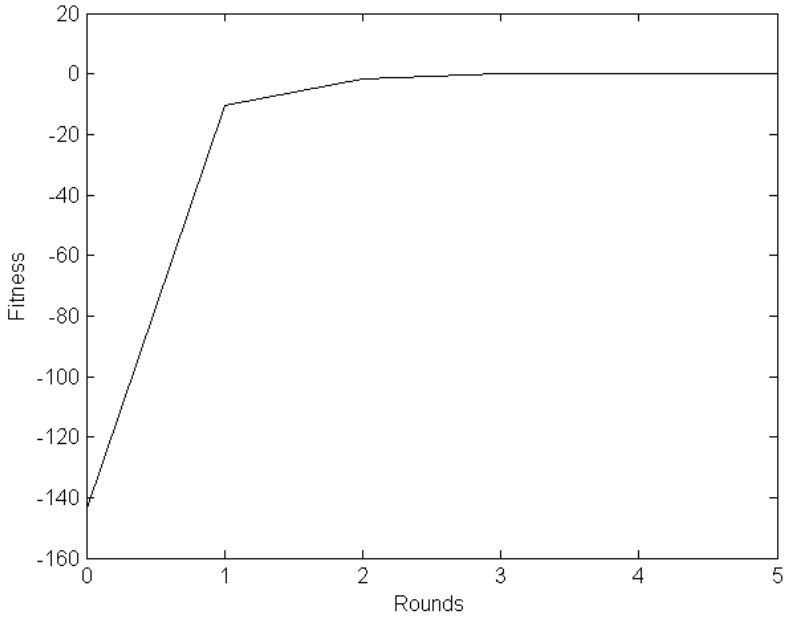

Figure 4.1: Experiment 1: Performance of Hill Climbing

4.2.2 Experiment 2: Performance

After the discussion of the results of Experiment 1, it is time to analyse the results of Experiment 2 before moving on to the conclusion of this thesis. The results can be found in Table 5.3. Three algorithms positively stand out from the group here: Genetic Algorithm with and without Hill Climbing and Tabu Search, with the third of these algorithms performing the best of all algorithms. Just as in the first experiment, it is clear that adding a Hill Climber improves the performance of all

Table 4.1: Results of Experiment 1 - First Part. The Mean and SD values are in seconds, and list the time values corresponding to the 25, 50, 75 and 100 tasks problem classes, in that order.

Algorithm	Mean	SD	% Good Schedules
Random Scheduler	24.30	47.85	80
	117.73	15.92	2
	120.00	0.00	0
	120.00	0.00	0
Hill Climbing	31.41	52.51	74
	32.34	49.34	76
	47.96	48.12	70
	47.08	30.36	88
Genetic Algorithm	6.20	16.36	98
	41.34	22.50	96
	90.97	22.49	88
	117.95	6.96	20
Genetic Algorithm with HC	3.09	0.81	100
	15.65	6.76	100
	34.05	21.55	96
	61.17	26.24	96
Particle Swarm Optimization 1	1.06	1.30	100
	64.16	51.69	54
	116.90	15.41	4
	120.00	0.00	0
Particle Swarm Optimization 1 with HC	1.18	1.36	100
	30.81	22.73	96
	91.48	26.95	62
	118.29	7.05	6

Table 4.2: Results of Experiment 1 - Second Part. The Mean and SD values are in seconds, and list the time values corresponding to the 25, 50, 75 and 100 tasks problem classes, in that order.

Algorithm	Mean	SD	% Good Schedules
Particle Swarm Optimization 2	0.60	0.63	100
	17.43	21.35	100
	101.16	32.36	32
	120.00	0.00	0
Particle Swarm Optimization 2 with HC	1.06	1.72	100
	25.44	31.91	92
	70.21	36.85	72
	115.70	15.60	8
Max-Min Ant System	9.31	24.84	98
	117.12	14.99	4
	120.00	0.00	0
	120.00	0.00	0
Max-Min Ant System with HC	2.52	6.89	100
	107.53	28.40	20
	120.00	0.00	0
	120.00	0.00	0
Tabu Search	6.06	23.87	96
	11.05	22.54	96
	33.18	29.05	94
	65.70	32.39	88

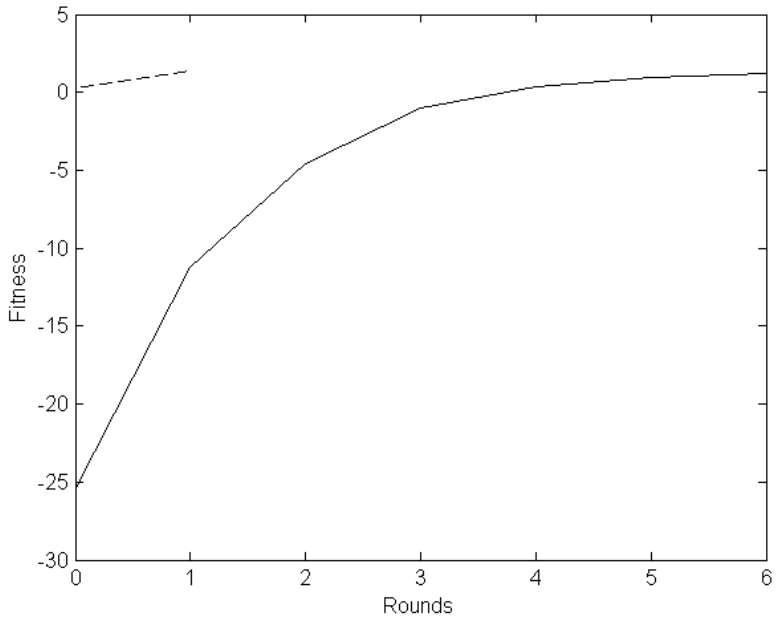

Figure 4.2: Experiment 1: Performance of Genetic Algorithm and Genetic Algorithm with Hill Climbing (dashed)

algorithms with this ability. While Particle Swarm Optimization 2 performs better than Particle Swarm Optimization 1, this effect reverses when the Hill Climbing component is added to these algorithms.

Again, the Max-Min Ant System performs very poorly, even worse than the basic Particle Swarm Optimizers, which already perform very badly. It seems that the Max-Min Ant System and the Particle Swarm Optimizers are just not suited for the scheduling problem presented in this thesis, at least not for the larger problems. The bad performance of the Max-Min Ant System might be due to the heuristic it uses: the time it takes a mechanic to do a task. Since there are many constraints in our scheduling problem, this heuristic may be too simple. However, more research is needed to conform this.

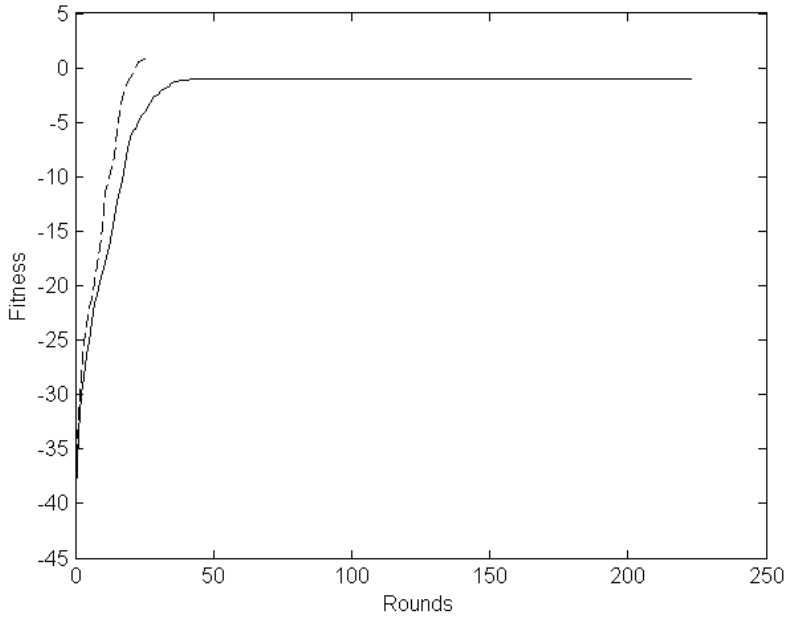

Figure 4.3: Experiment 1: Performance of Particle Swarm Optimization 1 and Particle Swarm Optimization 1 with Hill Climbing (dashed)

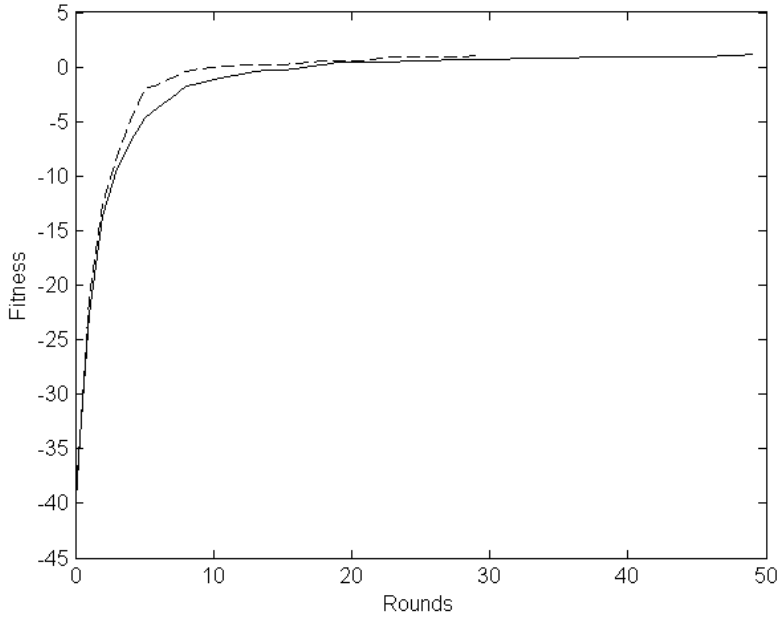

Figure 4.4: Experiment 1: Performance of Particle Swarm Optimization 2 and Particle Swarm Optimization 2 with Hill Climbing (dashed)

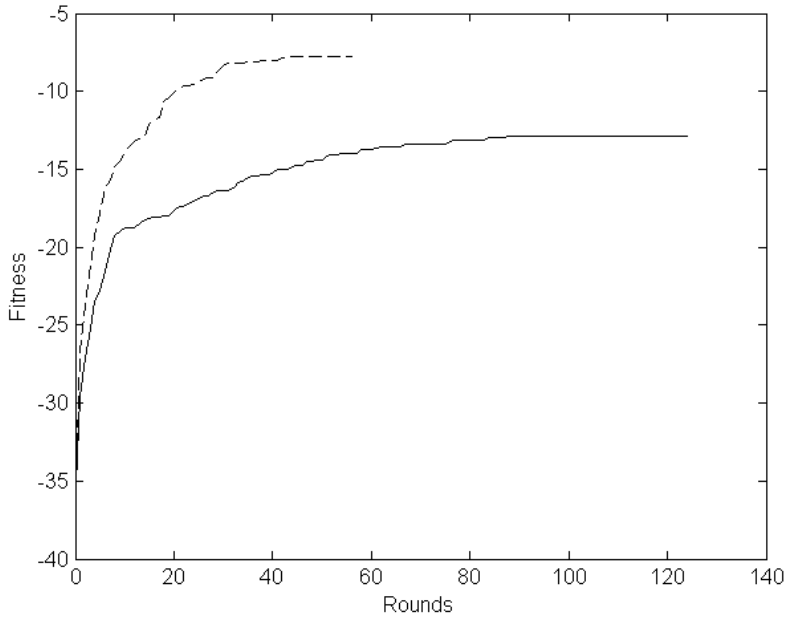

Figure 4.5: Experiment 1: Performance of Max-Min Ant System and Max-Min Ant System with Hill Climbing (dashed)

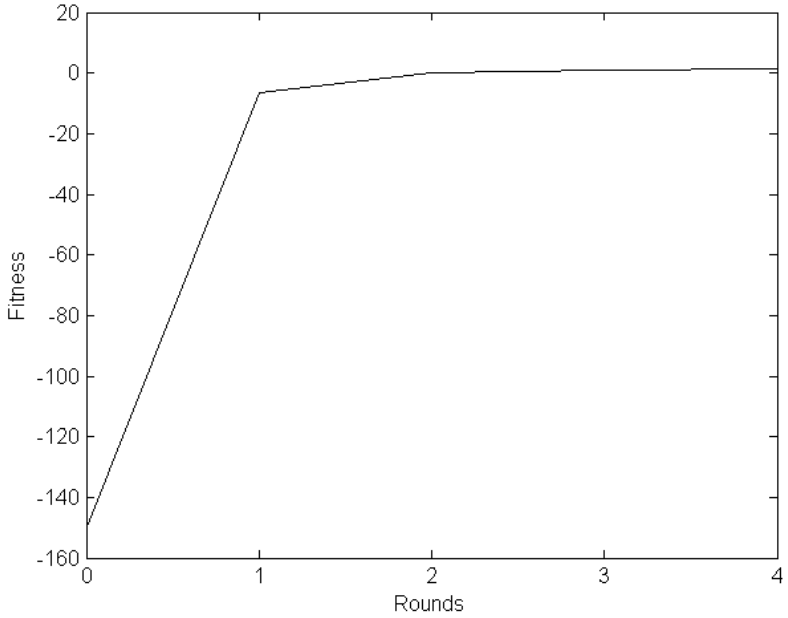

Figure 4.6: Experiment 1: Performance of Tabu Search

Table 4.3: Results of Experiment 2. The Mean is the average fitness value over 10 problems with each 100 tasks, which has to be at least 1.0 in order for the schedule to be reasonable. Higher than 1.0 means a more efficient schedule.

Algorithm	Mean	SD	% Good Schedules
Hill Climbing	0.14	3.02	96
Genetic Algorithm	1.33	0.05	100
Genetic Algorithm with HC	1.44	0.05	100
Particle Swarm Optimization 1	-36.18	23.13	80
Particle Swarm Optimization 1 with HC	-2.55	8.42	86
Particle Swarm Optimization 2	-20.58	21.37	80
Particle Swarm Optimization 2 with HC	-7.87	12.59	82
Max-Min Ant System	-90.01	29.49	80
Max-Min Ant System with HC	-65.34	35.92	80
Tabu Search	1.46	0.06	100

Chapter 5

Conclusion

The research described in this thesis aimed to find an algorithm that can solve a job shop scheduling problem with multiple constraints. More specifically, the setting of a simplified car workshop was chosen. The results of Experiment 1 and Experiment 2 together clearly indicate that of all the algorithms tested, the Genetic Algorithm with Hill Climbing is best suited for the scheduling problem presented in this thesis. In terms of performance, only Tabu Search beats this one, and only slightly. In terms of speed, it is only slightly outranked by a few other algorithms on the three smaller problem classes (25 and 50 and 75 tasks), but still wins on the 100 tasks problems. This directly and clearly answers the research question.
There are more conclusions to be drawn. A very important one seems to be that some algorithms, while elegant, are just not suited for this kind of problem. The Max-Min Ant System is the most obvious example here, but the Particle Swarm Optimizers also perform very bad, also (but less) when enhanced with Hill Climbing. This might be due to the number of constraints present in the schedules; however, to know this for sure requires more (specific) research. For example, Max-Min Ant System uses the time it takes a mechanic to perform a task as heuristic to choose that mechanic for that task; this might be too simple considering all the constraints. Another conclusion is that for all the algorithms tested here, it almost always helps to incorporate a Hill Climbing component. Finally, Particle Swarm Optimization 2 outperforms Particle Swarm Optimization 1 on both speed and performance. Adding a Hill Climber makes Particle Swarm Optimization 1 perform better in experiment 2, but the inverse is true in experiment 1.
Now that a conclusion is offered, it is important to note that the design goals of the algorithms can have a clear impact on the results of the experiments. The Genetic Algorithms (with or without the Hill Climber) were designed to quickly work towards a reasonable schedule: one that has no tasks performed too late or

agents working too late. In order to do this, mutations were primarily aimed at fixing such mistakes, and less at making a more efficient schedule. Shifting the focus towards creating more efficient schedules might have had a positive influence on the performance, and a (small) negative influence on the speed. This is the speed/performance dilemma mentioned in the introduction of this thesis, and of course is always a problem in this research area; it is however worth mentioning again, since the impact of such design choices can be quite large.

Because of the great performance of the Genetic Algorithm with Hill Climbing, future work could investigate more variants of Genetic Algorithms to possibly find even better algorithms for our scheduling problem. But more importantly, the presented algorithms were tested on simulated car workshop problems, but could be more generally applied to similar problems. As said before, one can imagine applying these algorithms to other kinds of repair shops, or even university scheduling. Of course, for such problems, other algorithms than the ones winning here might prevail, and some algorithms might have to be adapted to be suited to such problems. Without a doubt, it would be interesting to try to create an algorithm that would be suited for all scheduling problems. However, future work will have to decide whether this is a realistic option.

Bibliography

[1] C. Blum and A. Roli, 2003. *Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison*, ACM Computing Surveys (CSUR), volume 35, pages 268–308.

[2] B. Bullnheimer, R. Hartl and C. Strauss, 1999. *An Improved Ant System Algorithm for the Vehicle Routing Problem*, Annals of Operations Research, volume 89, pages 319–328.

[3] A. Colorni, M. Dorigo, V. Maniezzo and M. Trubian, 1994. *Ant System for Job-Shop Scheduling*, Belgian Journal of Operations Research, Statistics and Computer Science, volume 34, pages 39–53.

[4] R. Dawkins, 1976. *The Selfish Gene*, Oxford University Press.

[5] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, 2002. *A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II*, IEEE Transactions on Evolutionary Computation, volume 6, pages 182–197.

[6] J. L. Deneubourg, S. Aron, S. Goss and J. M. Pasteels, 1990. *The Self-Organizing Exploratory Pattern of the Argentine Ant*, Journal of Insect Behavior, volume 3, pages 159–168.

[7] M. Dorigo, 1992. *Optimization, Learning and Natural Algorithms*, Ph. D. Thesis, Politecnico di Milano, Italy.

[8] M. Dorigo and M. Birattari, 2010. *Ant Colony Optimization: Overview and Recent Advances*, Handbook of Metaheuristics, pages 227–263.

[9] M. Dorigo and K. Socha, 2006. *An Introduction to Ant Colony Optimization*, Handbook of Approximation Algorithms and Metaheuristics, pages 26–52.

[10] S. van den Dries and M. A. Wiering, 2012. *Neural-Fitted TD-Leaf Learning for Playing Othello with Structured Neural Networks*, IEEE Transactions on Neural Networks and Learning Systems, volume 23, pages 1701–1713.

[11] I. P. Gent and T. Walsh, 1993. *Towards an Understanding of Hill-Climbing Procedures for SAT*, AAAI, volume 93, pages 28–33.

[12] B. Giffler and G. L. Thompson, 1960. *Algorithms for Solving Production-Scheduling Problems*, Operations Research, volume 8, pages 487–503.

[13] F. Glover, 1986. *Future Paths for Integer Programming and Links to Artificial Intelligence*, Computers & Operations Research, volume 13, pages 533–549.

[14] F. Glover, 1989. *Tabu Search-Part I*, ORSA Journal on Computing, volume 1, pages 190–206.

[15] F. Glover, 1990. *Tabu Search-Part II*, ORSA Journal on Computing, volume 2, pages 4–32.

[16] F. Glover, 1990. *Tabu Search: A Tutorial*, Interfaces, volume 20, pages 74–94.

[17] S. Goss, S. Aron, J. Deneubourg and J. Pasteels, 1989. *Self-Organized Shortcuts in the Argentine Ant*, Naturwissenschaften, volume 76, pages 579–581.

[18] P. P. Grassé, 1959. *La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs*, Insectes Sociaux, volume 1, pages 41–80.

[19] D. Graupe and R. Ghandi, 2001. *Implementation of Traveling Salesman's Problem using Neural Network*, International Journal of Computer & organization Trends (IJCOT), volume 1, pages 161–164.

[20] K. Gurney, 1997. *An Introduction to Neural Networks*, CRC Press.

[21] J. H. Holland, 1974. *Adaptation in Natural and Artificial Systems*, University of Michigan Press.

[22] J. J. Hopfield, 1982. *Neural Networks and Physical Systems with Emergent Collective Computational Abilities*, Proceedings of the National Academy of Sciences, volume 8, pages 2554–2558.

[23] H. Izakian, B. T. Ladani, A. Abraham, and V. Snášel, 2010. *A Discrete Particle Swarm Optimization Approach For Grid Job Scheduling*, International Journal of Innovative Computing, Information and Control, volume 6, pages 4219–4233.

[24] D. Karaboga and B. Akay, 2009. *A Survey: Algorithms Simulating Bee Swarm Intelligence*, Artificial Intelligence Review, volume 31, pages 61–85.

[25] J. Kennedy and R. Eberhart, 1995. *Particle Swarm Optimization*, Proceedings of IEEE International Conference on Neural Networks, pages 1942–1948.

[26] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, 1983. *Optimization by Simulated Annealing*, Science, volume 220, pages 671–680.

[27] J. D. Knowles and D.W. Corne, 1999. *The Pareto Archived Evolution Strategy: A New Baseline Algorithm for Pareto Multiobjective Optimisation*, Proceedings of the 1999 Congress on Evolutionary Computation.

[28] J. D. Knowles and D. W. Corne, 2000. *M-PAES: A Memetic Algorithm for Multiobjective Optimization*, Proceedings of the 2000 Congress on Evolutionary Computation, pages 325–332.

[29] A. Lim, J. Lin, B. Rodrigues and F. Xiao, 2006. *Ant Colony Optimization with Hill Climbing for the Bandwith Minimization Problem*, Applied Soft Computing, volume 6, pages 180–188.

[30] P. Moscato and C. Cotta, 2003. *A Gentle Introduction to Memetic Algorithms*, Handbook of Metaheuristics, pages 105–144.

[31] J-M. Renders and H. Bersini, 1994. *Hybridizing Genetic Algorithms with Hill-Climbing Methods for Global Optimization: Two Possible Ways*, Proceedings of the First IEEE Conference on Evolutionary Computation, pages 312–317.

[32] S. Russell and P. Norvig, 2010. *Artificial Intelligence: A Modern Approach.* Third Edition.

[33] Y. Shi and R. Eberhart, 1999. *Empirical Study of Particle Swarm Optimization*, Proceedings of the 1999 Congress on Evolutionary Computation, volume 3.

[34] T. Stützle and H. H. Hoos, 2000. *Max-Min Ant System*, Future Generation Computer Systems, volume 16(8), pages 889–914.

[35] P.-Y. Yin, S.-S. Yu, P.-P. Wang and Y.-T. Wang, 2006. *A Hybrid Particle Swarm Optimization Algorithm for Optimal Task Assignment in Distributed Systems*, Computer Standards & Interfaces, volume 28, pages 441-450.

[36] E. Zitzler and L. Thiele, 1999. *Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach*, IEEE Transactions on Evolutionary Computation, volume 3, pages 257–271.