Job-Scheduling for automated Car Parking Systems
A Machine Learning Approach

Jan Lödige
Abstract

Job-Scheduling for automated Car Parking Systems

Jan Lödige

The ever growing amount of cars and their requirement for parking space has led to the development of highly sophisticated public automated car parking systems. The user acceptance criteria for such systems is the car drop-off and retrieval time. In this thesis, a genetic algorithm is developed, that tries to minimize the drop-off and retrieval times compared to established “First-In-First-Out” scheduling techniques. The algorithm is a so called crossbreeding algorithm, that combines machine allocation and job order execution into one chromosome encoding. Allocation and order are evolved by using genetic operators separately. A variety of different operators are tested in a Monte Carlo type simulation and the results are compared to the benchmark algorithm using scheduling strategy as currently in use. On average, the genetic algorithm can improve the job scheduling by 14% for a reasonable job queue.
## Contents

1 Introduction .................................................. 7  
  1.1 Hoist and Shuttle Configuration for Public Car Parking Systems .... 7  
  1.2 Optimization Target ..................................... 10  
  1.3 Implications of Double-Depth-Storage .......................... 14  
  1.4 Introduction to Genetic Algorithms .......................... 16  

2 Queue-Scheduling using a Cross-Breeding Genetic Algorithm .......... 19  
  2.1 Chromosome Encoding ...................................... 19  
    2.1.1 Species: Allocation Chromosome ......................... 21  
    2.1.2 Individuals: Permutation Chromosome .................. 21  
    2.1.3 From Chromosome to Schedule ........................... 22  
  2.2 Fitness & Selection ......................................... 23  
  2.3 Cross-Breeding ............................................. 25  
  2.4 Permutation Operators ...................................... 26  
    2.4.1 Factoradic Encoding Crossover ......................... 26  
    2.4.2 PMX: Partially Mapped Crossover ..................... 28  
    2.4.3 OX1: Order-based Crossover ............................ 28  
    2.4.4 POS: Position-based Crossover ......................... 29  
    2.4.5 Mutation Operators .................................... 31  
  2.5 Allocation Operators ....................................... 32  
  2.6 Benchmark Algorithm ..................................... 33  

3 Simulation & Results .......................................... 35  
  3.1 Results for Permutation Operators ........................... 36  
  3.2 Results for Allocation Operators ........................... 39  
  3.3 Results for combined operators ............................. 39  

4 Conclusions & Future Work ................................... 42  

Bibliography .................................................. 43
Nomenclature

ASAP  As Soon As Possible
CPS   Car-Park-System
DIM   Displacement Mutation (permutation operator)
EXM   Exchange Mutation
FIFO  First-In First-Out
GA    Genetic Algorithm
ISM   Insertion Mutation
IVM   Inversion Mutation (permutation operator)
OX1   Order-Based Crossover (permutation operator)
PMX   Partially Mapped Crossover (permutation operator)
POS   Position-Based Crossover (permutation operator)
RL    Reinforcement Learning
RNG   Random Number Generator
TSP   Traveling Salesman Problem
1 Introduction

Due to the continuing urbanization of modern societies, cities become more and more densely populated. At the same time, the constantly high demand for mobility results in an ever rising number of cars and shortage of parking space is the new norm.

Automated Car Parking Systems (CPS) allow a better usage of space, avoiding areas to maneuver and packing cars closer together than regular parking garages. The different CPS types can be divided into (residential) private, semi-private and public systems.

Residential private systems usually integrate the parking space into the housing. This is achieved by hoist(s), that lift the car up onto the floor where the residents live, and the car is parked right next to the apartments. Semi-private systems are typical for business-buildings, where the majority of cars are parked by the employed staff, but visitors can use the system as well.

Public systems as discussed in this thesis require the most sophisticated job scheduling due to the stochastic behavior of the human users, which poses a challenging problem for the automated control of such systems, as forward planning is almost impossible.

1.1 Hoist and Shuttle Configuration for Public Car Parking Systems

The CPS discussed in this paper use a system design known as shuttle-systems. Shuttle-systems are usually high-rack type storages, where racks of storage lots are placed around aisles. Fig. 1.1 shows an example system (only one side of the aisle for better visualization).

The main motivation behind shuttle-systems is to decouple the movement in different dimensions. Each level of the rack has its own or multiple shuttles to execute the pallet handling within the level (horizontal). The vertical connection between the levels is established by lifts. This decoupling gives the system more processing power (multiple shuttles/lifts work in parallel).

Therefore, shuttle systems are favorable in applications that demand high throughput and fast response to job requests as it is the case for CPS.
As an example CPS, Fig. 1.2 shows the layout for the Bryghus, Copenhagen, CPS, which is currently under construction. The transfer area (bottom) is where the users interface with the system. In 6 transfer cabins, the cars are parked and retrieved. The system is fully bidirectional, and therefore, the cars are not only dropped-off but also picked up in the cabins.\(^1\)

The cabins themselves are hoists, and after a car has been parked in the cabin and the people have left, the car is moved down into the storage area (top of Fig. 1.2). The storage area has three levels. To the left and the right of the hoists are the shuttle aisles (shuttles shown in yellow). There are two shuttles per track, i.e. 12 shuttles in total. The storage itself (red) is double-depth to increase storage capacity.

Due to the parallelization of processing in shuttle systems, different machines have to interact with each other. This makes the job-scheduling and machine allocation more challenging. Especially, because jobs generally have to be executed as soon as possible. If the response of the system is too slow, the acceptance for CPS suffers.

When it comes to the schedule horizon, it is helpful to interpret the input- and output-jobs as \textit{requests}, that present to the system at a certain point in time. Current research in genetic algorithms (GA) based schedulers for logistic processes either use fixed demo/test-sets of jobs (offline-GA) or a rolling-horizon\(^2\) approach (online-GA)

\(^1\)The DOKK1 system in Aarhus, Denmark, is currently Europe's biggest CPS with 1000 parking lots: https://www.youtube.com/watch?v=HPkrGamrd8

\(^2\)A “timeframe” is rolling through a list of jobs and only the jobs within the timeframe are used for
Figure 1.2: Layout for the Bryghus CPS, Copenhagen. Shuttles left and right of the hoist are colored yellow. The double depth storage is in red. (Source: Lődige Industries GmbH)
Both approaches require that the jobs are requested with a certain lead time, such that the job-list to be optimized is complete. If this leading time is long, the system loses flexibility, if it is short, the scheduler loses its optimization potential. In car parking systems, there is no leading time at all, as customers want to park and retrieve their cars immediately. Nevertheless, the other aspects of the scheduler (load-balancing, machine interaction) still need to be managed.

When it comes to system response, from the customers point of view, a CPS is similar to an elevator group. Inspired by the successful application of reinforcement learning (RL) on the group-elevator-problem [2, 6], a scheduler model combining GA and RL to address these specific requirements seems like a possible approach. Nevertheless, a genetic algorithm was chosen as the method to use.

In all, a good system controller has to deal with the following topics:

**Load-Balancing** over the entire system is essential, if the parallel processing power is supposed to be put at use effectively. If there is “job-cluttering” on e.g. one of the lifts, the overall performance of the system will lose efficiency.

**Machine-Machine-Interaction** occurs, whenever multiple machines are involved during the execution of a job. The main interaction to be optimized in shuttle systems is between hoists and shuttles. Minimization of unnecessary waiting time is important.

### 1.2 Optimization Target

To get a feeling for how CPS are used by the public, real operational data of the Aarhus DOKK1 system were analyzed. In the timeframe from 2015-07-01 until 2015-09-11 roughly 35,500 cars were parked in the system. This gives a good statistical database to fit a kernel density estimation model onto the data.

This model can be used to generate daily usage profiles, where the car arrivals and parking times are distributed like in the real data, but the total amount of cars that arrive during the day can be freely selected.

This has the advantage, that data can be generated for different CPS sizes. An example generated from the model is shown in Fig. 1.3. The plot on the top shows the average scheduling.

A preliminary literature study yielded primarily results in applications of maritime container yards. Nevertheless, the problems there are similar to shuttle-system scheduling.

A ready to use model for Density Estimation was used from the *scikit.learn* packages for Python. The path to the function within the package is *sklearn.neighbors.KernelDensity*.  

---

3 A preliminary literature study yielded primarily results in applications of maritime container yards. Nevertheless, the problems there are similar to shuttle-system scheduling.

4 A ready to use model for Density Estimation was used from the *scikit.learn* packages for Python. The path to the function within the package is *sklearn.neighbors.KernelDensity*.  

---
number of cars that arrive and depart over one day, when the total amount is 1000 cars per day. The gray shaded area shows a histogram with bin-size of 1 minute. The positive bins represent the cars arriving and the negative bins the cars departing. The red graph is the relative change of number of cars in the system and the black graph shows the total number of jobs that have to be done per minute. The plot on the bottom shows the number of cars stored in the system.

In Fig. 1.4 the relations between utilization, throughput and system response are illustrated. The graphs are not based on real data or calculations, but are provided to exemplify the relations on average.

The objective for the CPS owner is to maximize the utilization (number of cars parked), as parking cars is what is generating the revenue of the system. An example utilization profile throughout a day is given in Fig. 1.4 in the top plot. It is shown as relative utilization compared to parking lot capacity. The revenue depends on the gray shaded area under the profile.

For a given average parking duration of each car, utilization demands throughput (number of cars parked or retrieved per time unit). This is illustrated in the middle plot. The basic problem is, that the actual throughput capacity of the system decreases for higher utilization.

There are a couple of reasons for this. For one, the time it takes to park or retrieve a car depends on the distance the shuttles have to travel to reach the parking lot. With increasing utilization the probability of having to travel a greater distance increases. Furthermore, for systems with double depth storage, shuffle routines become necessary more often, when the system approaches 50% utilization. Those shuffle routines can either mean, to re-park a car in the front row to gain access to the car behind or to shift a car in the front row to the back to free the lot in the front.

Every once in a while, when e.g. a larger group of people coming from some event want to retrieve their cars at the same time, the system might not be able to meet the throughput demand and some people will have to wait longer than desirable. A good control strategy should of course try to minimize this effect.

But the real problem is, when the system is not able to substantially meet the throughput demand anymore. Then, longer and longer waiting queues will form and the system’s response can become arbitrarily bad, depending on how long the systems processing power is already under-capacitated. This “danger zone” must be avoided at all times.

The utilization, where the throughput capacity gets lower than the demand, is the actual maximum utilization $u_{max}$ for the system. It is important to point out, that such systems are not designed to reach the theoretical 100% utilization. To avoid that shuffle routines become unbearably slow, a certain amount of empty storage is absolutely necessary. Compared to processing power (machinery), storage space is relatively cheap.

The general goal of system control optimization is to try to increase $u_{max}$ by shaping the throughput capacity and at the same time ensuring, that the system’s response stays
Figure 1.3: Top: Infeed and Outfeed jobs that have to be done per minute for a day with 1,000 cars being parked. The red graph shows the relative change of cars in the system and the black the total number of jobs. Bottom: The number of cars stored in the system.
Figure 1.4: Illustration of how Utilization, Throughput and Response interact.
short. In the end, the acceptance of CPS by the public solely depends on the user experience and the response the system gives them. If they have to wait too long to retrieve their cars, they will not come back.

1.3 Implications of Double-Depth-Storage

Using Double-Depth-Storage (DDS) is a relatively easy way to increase the storage area for a CPS. But it comes at the cost of having eventually to re-park, or shuffle, cars to get access to cars in the back row. A method addressing the optimization of DDS for high-rack storage is given in [9]. The method used is a genetic algorithm.

A similar problem is the so called Berth-Allocation-Problem for maritime container yards. There, containers are stacked upon each other and an optimal stacking schedule must be determined. For this problem, genetic algorithms are used as well and yield promising results, e.g. in [7, 1].

For CPS, the DDS poses a different challenge. While in the above mentioned literature a complete list of arrival and departure times are given for the goods to be stored, for the CPS the departure times are unknown. The only information that can be used is the probability distribution for the parking time, which is shown in Fig. 1.5.

Since the mean parking time is relatively short compared to the daily duration of high utilization, during which DDS is used the most, a forecasting strategy to minimize shuffles will likely fail by the stochastic nature of the problem.

Instead, at this point, some simple rules are developed, that help to minimize the probability for shuffles without complex storage optimization strategies.

Assume a Double-Depth-Storage lot with two cars \( c_B \) and \( c_F \). The indices \( B \) and \( F \) indicate the car in the back and in the front respectively. Further, assume that none of the cars have been shuffled before, i.e. it is their first storage location since their arrival in the system.

Because DDS behaves like a LIFO queue (stack), the car in the back must have arrived before the car in the front, i.e. \( t^{(a)}_B < t^{(a)}_F \) with \( t^{(a)} \) meaning the time of arrival. Let

\[
\Delta t^{(a)} = t^{(a)}_F - t^{(a)}_B \tag{1.1}
\]

be the time between the two car arrivals.

A shuffle has to be made, if

\[
t^{(p)}_B < t^{(p)}_F + \Delta t^{(a)}, \tag{1.2}
\]

where \( t^{(p)} \) stands for the parking time. Figure 1.6 shows a Monte Carlo Estimate of the probability that a shuffle has to be made, as a function of \( \Delta t^{(a)} \).
The underlying model is the kernel density model described in section 1.2. The parking time estimates change throughout the day as is shown in Fig. 1.5. Especially the cars arriving early in the morning (between 6 and 9 AM) have a much longer parking time. This is because those cars mainly represent staff and employees of DOKK1, who park their cars during work hours.

For the Monte Carlo simulation the parking time distribution for 14 o’clock was used, as this is also the time of maximum utilization, i.e. with the highest probability for shuffles (see Fig. 1.3, bottom plot).

To minimize the probability for shuffles, it is best to store two cars that arrive shortly after another together in one DDS.

The only information about a parked car available is its arrival time. A couple of simple rules can be derived to stochastically try to minimize the number of shuffles:

1. If a new car arrives, and DDS is necessary, park it in front of the last car that arrived (minimize the time difference).

2. If a car has to be shuffled, park it in front of a car that recently arrived. This will give a negative time difference $\Delta t^{(a)}$, as the car in the front now is longer in the system than the car in the back. This yields a probability lower than 0.5, that the car in the front has to be shuffled a second time.

3. If the system is relatively idle, check whether there are bad combinations of cars in the DDS and preemptively shuffle them to minimize shuffles when the corresponding cars are requested.
1.4 Introduction to Genetic Algorithms

The method to optimize the job schedule in this thesis is based on Genetic Algorithms (GA), which belongs to the class of Evolutionary Algorithms. The basic idea is to translate the concepts of Charles Darwin’s evolution theory into a mathematical model, that can be used to solve optimization problems.

To give a short introduction to GA, consider the following example problem.

Given a set $S$ of integers, partition $S$ into two subsets $S_0$ and $S_1$, such that the difference of the sums of the subsets is minimized.

This is the optimization version of the so called “Partition Problem”\(^5\). To stay in the terminology of CPS, the set $S$ can be interpreted as a job queue and the jobs are allocated to two different machines (subsets $S_0$ and $S_1$). The integer values in $S$ represent the amount of time it takes to process the jobs respectively. Therefore, the sums of the subsets are the total time for each machine to finish their job queues. By minimizing the difference in those sums, the workload is balanced between the machines and the total make-span (the time it takes until both machines are finished and all jobs are processed) is minimized as well.

The first step for designing the GA is to define the chromosome encoding, which means

---

how a candidate solution for the optimization problem, referred to as an individual, is represented in terms of the GA. In this example, for a set \( S \) with \( N \) elements a binary vector of length \( N \) can be used. Each value of 0 or 1 in the vector assigns the corresponding job to the sets \( S_0 \) and \( S_1 \) respectively.

In the second step, a fitness function is defined. The fitness function assigns a fitness value to an individual, such that a higher fitness value quantifies a better solution to the problem than a lower one. The objective function in this case is the difference of the sums of the subsets and a possible fitness function is

\[
f(S_1, S_2) = -\left| \sum_{s_0 \in S_0} s_0 - \sum_{s_1 \in S_1} s_1 \right|.
\]

Note, that taking the negative of the difference converts the original minimization problem into a maximization problem for the GA, so a higher fitness means a better solution.

With the chromosome encoding and fitness function defined, the actual Genetic Algorithm can be described.

At first, an initial population of chromosomes (candidate solutions) is randomly generated and the corresponding fitness values are evaluated. The number of chromosomes is called the population size. Then, the algorithm loop begins:

1. Using a “Selection Operator” sets of parents are selected from the population.
2. Using a “Crossover Operator” the parents reproduce to create the offspring.
3. Using a “Mutation Operator” a (usually small) number of individuals are mutated.
4. A new population is selected from either the offspring alone or offspring and parents together.
5. Repeat Steps 1. to 4. until a stopping criteria is fulfilled.
The selection operator is meant to represent the evolutionary concept of “Survival of the fittest”. While purely random selection is possible, selection using the fitness values gives fitter individuals a higher probability to reproduce and evolve. Many Operators were developed. Examples are fitness proportional selection, where the probability for an individual to be selected is directly proportional to its fitness compared to the overall fitness of the population (the fitness function must ensure non-negative fitness values). Another approach is not to use the absolute fitness but the rank of the individual within the population, called rank-based selection. The fittest individual has rank one, the second fittest has rank two and so on.

After the sets of parents have been selected, the next step is to evolve offspring off of them by using a crossover operator, a process also called reproduction. In Fig. 1.7 one-point crossover is shown. At first, a crossover point (dotted line after the fifth element of the vector) is chosen at random. Then, the elements after that point are interchanged for the offspring. Therefore, the offspring contains parts from each of the parents. Many different crossover operators were developed. For example two or more crossover points can be used, but there are more specialized operators as well, for example ensuring special requirements for a solution to a given problem.

After crossover, usually a mutation operator is used on the population. Individuals are selected with a low probability for mutation and a mutation operator changes one or more of the values in an individual at random. Mutation is used to add diversity to the population. The above mentioned selection based on the fitness of the individuals, favors fitter solutions to be chosen for reproduction more often. Therefore, over time the individuals become more uniform and the diversity in the population decreases. This is called selective pressure and it depends on the type of selection operator. Mutation helps to keep diversity in the population. This also means, that the GA has a higher level of “exploration” in the solution space.

After crossover and mutation, the next generation is selected. Either the parents and the offspring can be used, or the offspring alone. In the GA developed in this thesis, a crossover probability $p_c$ determines, whether offspring is created and passed into the next generation, or no crossover takes place and the parents are taken over unaltered.

Using this new generation, the algorithm loop is repeated until some stopping criteria is met. This can be a fixed number of generations or for example a certain fitness level is reached.

This was only a very basic introduction to genetic algorithms. A broader overview of the field can be found in [3, 5].
2 Queue-Scheduling using a Cross-Breeding Genetic Algorithm

For scheduling the jobs of the CPS, two things must be achieved. For one, the jobs must be allocated to the different machines (hoist and shuttles) and the execution order of the jobs must be determined, which will be referred to as the permutation.

To incorporate both of this required information for building a schedule, a “cross-breeding” algorithm is introduced.

The idea behind it can be exemplified by the zebroid named “Eclyse”, which lives in the “Safari Park Stutenbrock”, near Paderborn, Germany and is shown in Fig. 2.1. Eclyse is a so called hybrid animal, that was born by cross-breeding a zebra and a horse.

Consider this zebroid as a species and there can be a variety of individuals of this species. While they all share a certain set of DNA, which makes them a zebroid, they all have individual differences for example defining their color or their stripe patterns.

The cross-breeding algorithm is similar. There are different species, which represent and encode the allocation, and for each of this species there is a set of individuals, which encode the permutation.

2.1 Chromosome Encoding

A job $J = \{T_H, T_S\}$ consists of two tasks. The hoist task $T_H$ and the shuttle task $T_S$. Depending on whether the job is an infeed or an outfeed operation, the machines have a different protocol of steps, which are shown in Fig. 2.2.

For an infeed job, first the assigned hoist has to move to the transfer area ($H$ to lobby), where the system user can park his car on the hoist ($C$ to $H$). After the the hoist has moved to the desired shuttle ($H$ to $S$) the car is shifted onto the shuttle ($C$ to $S$). The shuttle now brings the car to its parking lot ($S$ to $L$) and parks it there ($C$ to $L$).

The two task are directly couple to each other, as the shift between the hoist and the shuttle forces both machines to be at the take-over position and work in sync. Unlike in regular job-shop scheduling there may be no time gap between the two tasks.

Furthermore, tasks will be considered in a “start as late as possible” fashion. For the infeed job in Fig. 2.2, the shuttle could start to move towards the hoist earlier and wait
Figure 2.1: Zebroid named “Eclyse” looks like a 2-point crossover from a horse and a zebra. Source for the image: Website of the Safari-Park Stutenbrock near Paderborn, Germany: https://www.safaripark.de/safari/tiere-im-safaripark/zebrapferd.

Figure 2.2: Infeed and Outfeed task to be done by the hoists and shuttles.
in position for the hoist to arrive. But this would not make the job finish earlier.

If the machines start their task as late as possible, both tasks can simply be considered as a single job with a joint execution time $t_k$ for job $k$. The time $t_k$ is given by the time, when the transfer from hoist to shuttle (or the other way around) begins, i.e. at the beginning of ($C$ to $S$) or ($C$ to $H$) steps.

### 2.1.1 Species: Allocation Chromosome

The allocation chromosome is used to allocate a hoist and a shuttle for every job. Usually in genetic algorithms, the chromosome is a simple binary, integer or real valued vector that encodes a solution to the problem.

As shown in Fig. 2.3, the method proposed in this work uses a chromosome, which has multiple row vectors. The first row encodes the hoists, the second the shuttles. The third row is the permutation chromosome, which determines the order in which the jobs are executed. This will be discussed in section 2.1.2.

The values for the hoists and shuttles are integer values representing the index $i \in \{0, 1, \ldots, N_H - 1\}$ or $j \in \{0, 1, \ldots, N_S - 1\}$ for hoist $H_i$ or shuttle $S_j$ respectively in a system with $N_H$ hoist and $N_S$ shuttles overall.

For outfeed jobs, the shuttle allocation is actually not necessary, as the shuttle is given by the position, where the requested car is stored in the system. The simplest way to handle this limitation of values in the chromosome, is simply to ignore these allocations and using the correct shuttles while calculating the fitness. Nevertheless, this introduces redundancy into the encoding.

### 2.1.2 Individuals: Permutation Chromosome

Assume a given queue $J^Q = \{ J_0, J_1, \ldots, J_{N^Q - 1}\}$ with $N^Q$ jobs. Let $J^Q$ be ordered by the arrival times in a “first-in-first-out” sense. For easier notation, consider the queue being described as a tuple

$$\left( 0, 1, \ldots, N^Q - 1 \right)_Q$$

(2.1)
only holding the indices of the jobs in the queue, where the subscript $Q$ indicates the “time-of-arrival ordering”.

There are $N^Q!$ permutations of possible execution schedules for the queue and the objective for the genetic algorithm is to find the optimal permutation in regard of the overall execution time for all the jobs.

A permutation is the bijection

$$\pi : J^Q \rightarrow J^Q$$  \hspace{1cm} (2.2)

and therefore must hold every element of $J^Q$ exactly once. Using the same index based tuple notation, a correct permutation for a 4-job queue would for example be

$$\pi = (2, 0, 3, 1)$$ \hspace{1cm} (2.3)

and for the corresponding job schedule follows

$$J^\pi = \{J_{\pi(0)}, J_{\pi(1)}, J_{\pi(2)}, J_{\pi(3)}\} = \{J_2, J_0, J_3, J_1\}.$$ \hspace{1cm} (2.4)

Using such permutation tuples $(\cdot)_{\pi}$ as the chromosome encoding requires specialized crossover and mutation operators. The condition, that a valid permutation must hold each element of the original set exactly once, is a drawback.

### 2.1.3 From Chromosome to Schedule

The actual schedule for the machines is not given be the chromosome directly, but has to be constructed from it. This is because every job consists of two tasks, which may result in the necessity to wait for another job to finish, before a new one can be started.

This can be best explained by example. In Fig. 2.4 an example solution for a system with 3 hoists and 3 shuttles is shown. Consider the solution for a problem with 4 jobs as given below:

<table>
<thead>
<tr>
<th></th>
<th>$J_0$</th>
<th>$J_1$</th>
<th>$J_2$</th>
<th>$J_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hoist</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Shuttle</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Order</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The green colored jobs are infeed jobs and the red colored jobs are outfeed jobs.

The schedule is created by filling the timetable job by job according to the order given in the solution. Therefore, $J_1$ is filled first and can be executed immediately. It is best to think of it, as if the job was falling into the schedule from right to left, until it hits another job either for the hoist or for the shuttle.

In the second step, $J_3$ is filled. As $J_1$ and $J_3$ both use hoist $H_1$, the start for $J_3$ is delayed.
Step 3 reveals two important facts about this type of encoding. First, the order chromosome has to be thought of in terms of “schedule construction” not in a strictly temporal sense. The new job scheduled in step 3 is executed earlier than the one from step two.

This does, again, introduce redundancy into the encoding. The order \( (2, 0, 3, 1) \) as in Fig. 2.4 could be replaced by \( (1, 0, 3, 2) \) and the resulting schedule would be the same.

### 2.2 Fitness & Selection

In scheduling problems, the total make-span\(^1\) for a set of jobs is commonly used as subject to minimization. In case of the CPS this is not the case.

For one, since outfeed jobs have a given assigned shuttle, there can be “critical paths”, when there are several outfeed jobs for the same shuttle. This is depicted in Fig. 2.5.

---

\(^1\)In scheduling problems, make-span refers to the total amount of time it takes, until all jobs have been processed.
Figure 2.5: Example Schedule where several outfeed jobs assigned to one shuttle form a critical path.

Shuttle S1 has to retrieve 5 cars and the last job of S1 gives the total make-span. Nevertheless, there is still room for improvement, e.g. looking at shuttle 5 or 6. The corresponding jobs could be finished much earlier. But their execution times are not considered when using the total make-span.

Therefore, to consider all the jobs in the schedule, the euclidean norm

$$f = \|t\|_2 = \left( \sum_{k=1}^{N_Q} t_k^2 \right)^{1/2}$$  \hspace{1cm} (2.5)

is used to find the fitness $f$ from the time vector $t$ containing the job times $t_k$.

The selection should be fitness proportionate, such that individuals with better fitness have a higher probability of being selected for crossover than less fit individuals.

As this is a minimization problem, the fitness values of the individuals are normalized as

$$\tilde{f}_i = \frac{\max (f) - f_i}{\max (f) - \min (f)}$$  \hspace{1cm} (2.6)

onto the closed interval [0, 1]. Note, that if $f_a < f_b$ then $\tilde{f}_a > \tilde{f}_b$. Therefore, under this normalization, the problem becomes a maximization problem and with probability

$$p_i = \frac{\tilde{f}_i}{\sum_{j=1}^{I} \tilde{f}_j}.$$  \hspace{1cm} (2.7)
for individual \( i \) to be chosen, the selection is fitness proportionate.

### 2.3 Cross-Breeding

The cross-breeding algorithm distinguishes from normal genetic algorithms by the structure of the population and in its algorithm procedure.

In Fig. 2.6 the population is shown. It is divided into \( N_S \) species, which consist of \( N_I \) individuals each. The species represent the machine allocations. This means, that all the individuals of one species have the same allocation, or the other way around, for each allocation there are a number of permutations.

The species is creating an instance of the actual problem and then the individuals try to optimize it.

Fig. 2.7 shows a flowchart of the cross-breeding algorithm. After initialization of the population it begins with evolving the species. Each species is assigned a fitness value which corresponds to the fittest individual of the species.

Then regular crossover and mutation is performed on the allocation part of the chromosome alone. The probability \( p_c^S \) determines for each set of parents, whether the crossover is done or if the parents are copied into the next generation unaltered. The mutation is done with probability \( p_m^S \).

Then the fitness for the population is evaluated, by building the schedules for all individuals as described in section 2.1.3 and calculating the fitness.

In the second stage of the algorithm, the individuals are evolved. This is done for each species. That means, that individuals are only selected within their species for
Additionally, there are two bypass probabilities $p_S^g$ and $p_I^g$. They can be used to favor the evolution of either the species or the individuals.

### 2.4 Permutation Operators

As pointed out before, crossover operators for permutation chromosomes encoding the order of job execution have to make sure, that the solution is a valid permutation, i.e. all jobs must be chosen for the order exactly once. These conditions are the same for the Traveling-Salesman-Problem (TSP), which was widely studied. The meta-study [4] gives an extensive overview of genetic algorithm representations and genetic operators for the TSP, as well as a performance comparison for them using a variety of well known TSP test sets\(^2\).

As the CPS problem discussed in this paper distinguishes in some key aspects from the normal TSP, the most promising operators found in [4] are tested on the CPS problem. For clarity they are explained here as well.

#### 2.4.1 Factoradic Encoding Crossover

The factoradic encoding operator is not a specialized permutation operator, but the representation of the permutation is changed, such that normal genetic operators can be used.

---

\(^2\)Grötschels24 and Grötschels48 data sets.
Figure 2.8: Example for a 1-point crossover with a permutation encoding and a factoradic encoding.

The permutation can be represented in the factorial number system (factoradic), which has a factorial base. This approach of chromosome representation is e.g. shown in [8], and used in a GA for flow-shop scheduling.

A factoradic number

\[ A! = a_{n-1} \ldots a_1 a_0 \]  

with \( n \) digits \( \{ a_0, a_1, \ldots, a_{n-1} \} \) is indicated by a subscript \(!\) and can be converted to base 10 by

\[ A_{10} = \sum_{k=0}^{n-1} a_k \cdot k! \]  

As the base is not constant but changing with its place, there is the condition on the digits that \( a_k \leq k \). This condition of maximum digit value is preserved during crossover, as the digits do not change the place (in terms of \( k \)). This is illustrated on the right hand side of Fig. 2.8.

From a programming perspective, the factoradic representation of a permutation can be understood as iteratively in \( k \) “popping the element at index \( a_k \)” from the original set of elements \( (\cdot)_Q \). As an example, the transformation from the factoradic representation to the corresponding permutation for

\[ (3 1 2 1 0)!, \rightarrow (3 1 4 2 0) \pi \]
is shown below:

\[
\begin{align*}
\text{for } k=\ldots,n-1,0 & \rightarrow (3 \ 1 \ 2 \ 1 \ 0) \rightarrow [0 \ 1 \ 2 \ 3 \ 4]_Q \rightarrow 3_\pi \quad \text{pop } Q \text{ at index 3} \\
(3 \ 1 \ 2 \ 1 \ 0) \rightarrow [0 \ 1 \ 2 \ 4]_Q \rightarrow 1_\pi \quad \text{pop } Q \text{ at index 1} \\
(3 \ 1 \ 2 \ 1 \ 0) \rightarrow [0 \ 2 \ 4]_Q \rightarrow 4_\pi \quad \text{pop } Q \text{ at index 2} \\
(3 \ 1 \ 2 \ 1 \ 0) \rightarrow [0 \ 2]_Q \rightarrow 2_\pi \quad \text{pop } Q \text{ at index 1} \\
(3 \ 1 \ 2 \ 1 \ 0) \rightarrow [0]_Q \rightarrow 0_\pi \quad \text{pop } Q \text{ at index 0}
\end{align*}
\]

This example makes it intuitively clear, why \( a_k \leq k \): Whenever an element is popped out of the \( Q \)-array, its length gets shorter by one, i.e. there are less elements to choose from.

One problem with this type of encoding is, that due to the mechanism described above, after e.g. a 1-point crossover the tail of the offspring is basically randomized. It still holds the same values as in the parent, but it now means to “pop elements” out of a different \( Q \)-array.

### 2.4.2 PMX: Partially Mapped Crossover

Partially mapped crossover (PMX) can be considered the 1-point crossover equivalent for permutation chromosomes. In Fig. 2.9 the creation of one offspring from two parents is shown (the other offspring is created accordingly).

First, a 1-point sub-tour is randomly selected. Now, instead of simply copying the values of the sub-tour of parent two into parent one, value by value they are swapped into position within parent one. Exchanging two values always forms a new, valid permutation.

In the example of Fig. 2.9, the sub-tour \((2, 0, 5)\) in parent 1 is supposed to be replaced by the sub-tour \((4, 1, 7)\) of parent 2. To change the first value from 2 to 4, those two values are simply interchanged in the solution. The same principle is repeated until the sub-tour is complete.

### 2.4.3 OX1: Order-based Crossover

The order-based crossover (OX1) draws resemblance to the general 2-point crossover operator. Avoiding a swapping technique as the PMX does, allows it to preserve more of the original order given by the parents.

An example is given in Fig. 2.10. First, a random sub-tour is selected from the parents and copied into the offspring. Now, the values that already are in offspring 2 are removed from parent 1. The remaining values are then copied beginning after the sub-tour. At
1.) select 1-point subtour

Parents

2.) find same values

4.) repeat 1-3 for all genes in subtour

Offspring 1 holds subtour of Parent 2

3.) swap the genes

Offspring 1 after first swap

Figure 2.9: Example for partially mapped crossover (PMX). Only the creation for offspring 1 is shown here. Offspring 2 is created the same way, but interchanging the parents.

the end of the chromosome the values flow over to the beginning and fill the rest of the chromosome.

As pointed out in [4], the OX1 operator exploits the property of the Traveling Salesman Problem, that the order of the cities is important, not at which position they are in placed the chromosome. Furthermore, the solutions to the TSP are cyclic, meaning that the salesman starts and ends at the same city. This implies, that cycling through a given permutation of the TSP, will yield different starting cities, but the result (the distance traveled by the salesman) is always the same.

For the CPS problem, this is not the case, as the solution is not cyclic. Therefore it might be favorable, to fill the values not after the sub-tour, but from the beginning of the chromosome. This will give a closer resemblance to the original positions of the values.

2.4.4 POS: Position-based Crossover

The position-based crossover (POS) works similar to OX1, but now as a uniform n-point crossover. As shown in Fig. 2.11, n genes are selected uniformly and copied into the offspring. Now again, from parent 1 the values already copied into offspring 2 are removed. The remaining values in parent 1 are then filled into the empty spots in offspring 2, starting at the beginning of the chromosome. Offspring 1 is created accordingly.
1.) select 2-point subtour

Parents

\[ \begin{array}{cccccccc}
2 & 0 & 5 & 7 & 3 & 6 & 4 & 1 \\
2 & 1 & 6 & 4 & 3 & 5 & 0 & 7
\end{array} \]

2.) copy into offspring

Offspring

\[ \begin{array}{cccc}
5 & 7 & 3 & \\
6 & 4 & 3
\end{array} \]

3.) remove values already in offspring 2 from parent 1

Parents

\[ \begin{array}{cccccccc}
2 & 0 & 5 & 7 & 3 & 6 & 4 & 1 \\
2 & 1 & 6 & 4 & 3 & 5 & 0 & 7
\end{array} \]

for parent 2 correspondingly

Offspring

\[ \begin{array}{ccccccc}
4 & 0 & 5 & 7 & 3 & 2 & 1 & 6 \\
7 & 1 & 6 & 4 & 3 & 2 & 0 & 5
\end{array} \]

4.) fill offspring after subtour

Figure 2.10: Example for order-based crossover (OX1).

1.) select \( n \) random genes

Parents

\[ \begin{array}{cccccccc}
2 & 0 & 5 & 7 & 3 & 6 & 4 & 1 \\
4 & 1 & 7 & 2 & 0 & 5 & 3 & 6
\end{array} \]

2.) copy genes into offspring

Offspring

\[ \begin{array}{cccc}
5 & 3 & 4 & 1 \\
7 & 0 & 3 & 6
\end{array} \]

3.) remove values already in offspring 2 from parent 1

Parents

\[ \begin{array}{cccccccc}
2 & 0 & 5 & 7 & 3 & 6 & 4 & 1 \\
4 & 1 & 7 & 2 & 0 & 5 & 3 & 6
\end{array} \]

for parent 2 correspondingly

Offspring

\[ \begin{array}{cccccccc}
7 & 2 & 5 & 0 & 3 & 6 & 4 & 1 \\
2 & 5 & 7 & 4 & 0 & 1 & 3 & 6
\end{array} \]

4.) fill offspring with remaining values

Figure 2.11: Example for position-based crossover (POS).
2.4.5 Mutation Operators

Not only crossover operators have to be specially designed for permutation encodings, same hold for mutation. The meta-study [4] gives a comprehensive overview in this regard as well. The 4 most promising mutation operators are tested for this problem.

They could be roughly categorized as:

- Sub-tour mutation:
  - Displacement Mutation (DIM)
  - Inversion Mutation (IVM)

- Gene Mutation:
  - Exchange Mutation (EXM)
  - Insertion Mutation (ISM)

In fig. 2.12 all four are depicted. The sub-tour operators DIM and IVM are very similar and shown on top. At first, a 2-point sub-tour is selected and simply shifted to a new position within the chromosome. The only difference is, that for the inversion mutation the sub-tour is reversed in order.

Insertion mutation is basically the same as DIM, but only a single gene is shifted to a new position. Exchange mutation not just shifts a gene, but swaps two selected genes.
### 2.5 Allocation Operators

Crossover and mutation for the allocation chromosome can generally be done using the well known standard operators like 1-point, 2-point or \( n \)-point crossover. Uniform mutation does work as well.

In either case, the fact that the allocation encoding is designed as a two-row vector, leaves two options for choosing the crossover points. This is shown in fig. 2.13 using 2-point crossover as an example.

On the left-hand side, the crossover points are synchronized for both rows. This means, that crossover preserves the allocation of hoist and shuttle for every job. Looking at the allocation in the third column of parent one, this job is allocated for \( H_2 \) and \( S_3 \). Although, this gene is crossed into offspring 2 and is (especially in conjunction with the corresponding permutation) part of a different solution, the combination of \( H_2 \) and \( S_3 \).
for this job remains.

In contrast, the 2-point crossover on the right hand side uses different crossover points for each row. This means, that new combinations for the allocation in regard of hoist and shuttle are created during crossover.

### 2.6 Benchmark Algorithm

The benchmark algorithm uses a strategy, that could be described as “First-In-First-Out” (FIFO) and “As-Soon-As-Possible” (ASAP).

The FIFO principle is used for the order, in which the jobs are processed. Again, the order refers to the schedule-building algorithm, as described in section 2.1.3. It is therefore equivalent to a permutation chromosome

\[
(0, 1, 2, \ldots, N_Q - 1)_{\pi}.
\]  

The ASAP principle applies to the allocation of machines. The allocation is done on the fly, while building the schedule. In total, there are \(N_H \cdot N_S\) possible allocations per job. All those combinations are tried and the corresponding job time \(t_k\) is calculated. Then, the allocation with minimal \(t_k\) is used. If there are multiple combinations with equal \(t_k\), one allocation is chosen at random.

In Fig. 2.14 the scheduling is shown for the same example as in section 2.1.3.
Figure 2.14: Example, how a schedule is generated by the benchmark algorithm.
3 Simulation & Results

In chapter 2 a variety of crossover and mutation operators were introduced for the allocation as well as for the permutation subproblems.

To achieve reliable results for the analysis of the proposed method, a Monte-Carlo approach is used. The performance of the method is determined by averaging over 100 problem instances for scheduling a CPS. To generate the problem instances, a seeded random number generator (RNG) is used, allowing to reproduce the problem instances by reapplying the seed. The initial conditions for the GA, i.e. the initial population, is generated by the seeded RNG as well. Therefore, each parameter setup starts with the same initial population.

While running the algorithm, the random numbers needed for crossover, mutation or selection probabilities are not seeded and independent from one another.

The control parameters for the algorithm are:

<table>
<thead>
<tr>
<th></th>
<th>probability</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Species</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generations</td>
<td>( p_g )</td>
<td></td>
</tr>
<tr>
<td>Crossover</td>
<td>( p_c )</td>
<td>1-p, 1-p synced, 2-p, 2-p synced</td>
</tr>
<tr>
<td>Mutation</td>
<td>( p_m )</td>
<td>uniform mutation</td>
</tr>
<tr>
<td><strong>Individuals</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generations</td>
<td>( p_i )</td>
<td></td>
</tr>
<tr>
<td>Crossover</td>
<td>( p_c^i )</td>
<td>FAC, PMX, OX1, POS</td>
</tr>
<tr>
<td>Mutation</td>
<td>( p_m^i )</td>
<td>DIM, IVM, EXM, ISM</td>
</tr>
</tbody>
</table>

To find the best combination of operators, they are looked at individually by minimizing the effects of the other elements of the GA.

As an example CPS-setup, a system with 6 hoists and 6 shuttles is used. Each shuttle handles 50 parking lots giving an overall storage capacity of 300 cars. This is very similar to the Bryghus system described in section 1.1.

The test-problems are with \( N = 30 \) jobs unusually hard for such a system setup and such situations are rare in real life. On average this would mean, that per hoist and therefore per transfer cabin there were 2.5 cars waiting to be parked and 2.5 people were waiting to retrieve their cars. This delays would certainly cause aggravation among the drivers. Nevertheless, the idea to chose such a hard setup, is, that the influence of different genetic operators might be more significant.

The jobs are balanced in regard of infeed and outfeed, i.e. 15 jobs each. For calculating the travel distances of the shuttles, the parking spots are uniformly distributed along
the aisles.

For all simulations, the population size is chosen to have $2N$ species with $2N$ individuals per species. This gives a total of population size of $(2N)^2$, in this case 3,600 individuals.

In section 3.3 smaller problem instances are used to show the effect of problem size on the algorithm performance and the populations are smaller accordingly.

No elitism is used. This means, that there is no functionality to preserve the best solution(s) by copying them into the next generation of the population. This means, that the individual with the best fitness might be destroyed during crossover or mutation. Nevertheless, the best solution is kept as a “bookkeeping” value. It will not influence the evolution of the population, but be shown as the best result.

All results are given as fitness

$$f_{rel}(g) = \frac{f(g) - f_{bench}}{f_{bench}} \quad (3.1)$$

relative to the the fitness of the benchmark algorithm described in section 2.6 and as a function of the generation $g$ of the GA.

### 3.1 Results for Permutation Operators

The first part of the simulation focusses on the job permutation only. This means, that $p_g^S = 0$ and only the initial allocations generated by the random number generator are used. The corresponding flowchart for the algorithm is shown in Fig. 3.1. The complete upper half of the algorithm is ignored.
Furthermore, to begin with, only mutation is used. Therefore, $p^I_m = 0$ as well. This means, the algorithm performs a purely random-search, without any fitness-related selection or crossover.

The probability $p^I_m$ for an individual to be chosen for mutation is set to $p^I_m = 0.1$. In this pure random-search setup, where no form of fitness-related selection takes place, searching only in “10% of the time” might not seem logical. Nevertheless, this low setting was used to give a better comparability in terms of convergence-speed with the simulations, where mutation is used with $p^I_m = 0.1$ together with crossover.

In Fig. 3.2 in the left-hand side plot, the performance of the 4 mutation operators described in section 2.4.5 is shown. The algorithm is run for 100 generations and all graphs start at the same point, as the initialization is generated by a seeded RNG.

From there, the displacement mutation operator (DIM) has a clear advantage. DIM is, like the second best, inversion mutation (IVM), a sub-tour operator. Compared to ISM and EXM, which both select single genes only for moving or swapping them within the permutation, sub-tour operators seem to have an advantage searching the solution space.
Parents \( N = 30 \) and \( n = 15 \)  

\[ \text{Parents} \rightarrow \text{Offspring} \]

Parents \( N = 30 \) and \( n = 3 \)  

\[ \text{Parents} \rightarrow \text{Offspring} \]

Figure 3.3: Example for POS operator with \( n \) being large (top) and \( n \) being small (bottom). The parents are colored in blue and red. The \( n \) selected genes are colored yellow.

Now, using DIM as mutation, crossover as described in section 2.4 is added to the simulation.

The results are shown in Fig. 3.2 on the right-hand side. The best results are achieved with partially-mapped crossover (PMX) followed by PO6.

The POS operator is the only one, which has an additional parameter \( n \), with which the number of selected genes for crossover is controlled. PO\( x \), in this case PO6 corresponds to

\[ n = \left\lfloor \frac{N}{x} \right\rfloor \quad (3.2) \]

and \( \lfloor \cdot \rfloor \) meaning the floor operator (i.e. rounding down to closest integer). This means, for \( N = 30 \), that PO6 stands for POS with \( n = 5 \). The red graph labeled with POS uses \( n = 15 \), i.e. half of the genes are selected to be copied into the offspring.

It is quite prominent that smaller \( n \) result in better crossover. The reason for this might be, that for high values of \( n \) the crossover has a stronger randomizing effect on the offspring.

To illustrate this, in Fig. 3.3 two examples with different \( n \) are shown. On the top, a large number \( n = N/2 = 15 \) is used. The selected genes are shown in yellow and are copied into the offspring. After filling the gaps with genes from the other parent, there are no significant sub-tours from the parents left in the offspring. For a small \( n = 3 \), much of the original information given by the parents is preserved.

PMX, the most promising crossover operator, is a 1-point crossover equivalent for permutation encodings, for which the selected 1-point sub-tour is copied into the offspring unaltered. Keeping this and the findings for POS in mind, this is an indication, that for this CPS scheduling problem the operators should try to preserve information as much as possible and avoid too much randomization in the offspring.
3.2 Results for Allocation Operators

In a similar fashion as for the permutation, the different allocation operators are simulated independently from the rest of the algorithm. This is depicted in Fig. 3.4. By setting \( p_g^I = 0 \) the evolution of the individuals is turned off and they keep their values determined by initialization.

For mutation, uniform mutation is used with \( p_m^S = 0.1 \). If a species is selected for mutation, each gene will be altered with a probability of 0.4.

The crossover operators are used as described in section 2.5. The results of the simulation are shown in Fig. 3.5.

As the convergence rate is slower as for permutation, the simulation is run for 250 generations. Again, the same problem instances are used as in section 3.1. The results are surprisingly inconclusive. While the 2-point crossover (red) performs worse than the others for 175 generations, it can catch up in the end.

One-point and 2-point-synchronized crossover perform the best, overall. But not by much.

3.3 Results for combined operators

So far, the algorithm was only simulated with parts being deactivated. Now, using 1-point crossover (without synchronizing the crossover points) for the allocation and partially-mapped crossover (PMX) and displacement mutation (DIM) for the permutation, the algorithm can be used with all parts running.
The first question is, if the bypass probabilities $p_I^f$ and $p_S^S$ should be used to make the evolution of allocation or permutation statistically more likely. To find this out, several combinations were tried and the results are shown in Fig. 3.6 on the left-hand side.

The labeling of the graphs as $I_xS_y$ describe the values $p_I^f = \frac{x}{10}$ and $p_S^S = \frac{y}{10}$. So, for example, $I_6S_8$ stands for the combination $p_I^f = 0.6$ and $p_S^S = 0.8$.

The plot shows, that especially $p_S^S$ should be large, as the three best results are achieved for $p_S^S = 0.8$. The blue graph with $p_I^f = 0.8$ and $p_S^S = 0.8$ performs the best and this indicates, that balancing the evolution of allocation and permutation is favorable.

As pointed out before, the problem size of $N = 30$ is very large for this CPS-setup. Therefore, the algorithm was used with the given parameters on problems with smaller size. The performance in comparison to the benchmark improved with decreasing problem size, as shown in Fig. 3.6.

Job queues with $N = 15$ can in real life be considered as high-traffic situations where a CPS of the size simulated here is under high demand. On average the genetic algorithm can improve the job scheduling by ca. 14% (green graph in Fig. 3.6).
Figure 3.6: Left: Simulation results using different bypass possibilities. The labeling of the graphs as $I_xS_y$ describe the values $p_{f}^{I} = \frac{x}{10}$ and $p_{g}^{S} = \frac{y}{10}$. Right: Different problem sizes with number $N$ of jobs. The algorithm gets more effective for smaller sizes.
4 Conclusions & Future Work

Scheduling jobs for an automated car parking system is similar to a variety of job-shop scheduling problems. In the past, many methods using genetic algorithms were developed and published in the literature. The traveling salesman problem, being a special case of job-shop scheduling, has been researched in regard of genetic algorithms for decades.

As a reference car parking system, one that is currently under construction in Copenhagen, Denmark, as part of the harbor front area re-development also called the Bryghus-project, was used. What sets this CPS apart from other job-shop problems, is, that there is no buffering between machines. Therefore, a new methodology to encode solutions to the CPS problem was developed, in conjunction with an algorithm to build a schedule from such a solution.

This results in a two-dimensional chromosome with two row-vectors for the machine allocation and one row-vector for the permutation (execution order). This encoding not only imposes special requirements in regard of the evolutionary operators, but also for the algorithm itself.

Therefore, a two-stage genetic algorithm was developed, which can, as a metaphor, be interpreted as a cross-breeding algorithm. The idea is, that the individuals in the population are divided into species, which represent the machine allocation. For each such species, there is a number of individuals which encode the permutation of the schedule.

The two stages of the algorithm are, that first species evolve using crossover and mutation, and then for each of those species, the individuals evolve using crossover operators that can be used on permutation chromosomes.

The implementation of the algorithm was done in Python and built from scratch, only using standard packages for numerical methods (especially NumPy). For faster execution a python implementation of MPI was used to employ all cores of the computer. The GA easily results in very large population size, causing slow computation.

As there is no prior experience with such an algorithm, a variety of genetic operators were used and tested in a Monte Carlo fashion to determine the best combination of operators for this problem.

The simulation has shown, that the performance of the algorithm compared to a benchmark scheduling method (as is used in existing CPS), depends on the problem size.
number of jobs to be scheduled). For realistic conditions the algorithm can improve the schedule about 15% compared to the benchmark. For very large problem sizes, on average, the algorithm is still better than the benchmark, but only slightly.

Following, there are two ideas that will be tested in future work, that might improve the performance of the GA.

**Introducing Balancing Allocation Mutators**

A solution to the problem will not be good, if the solution is unbalanced in regard of the machine allocation.

As an extreme example: If all jobs are scheduled on one hoist and one shuttle alone and all other machines are idle, the system doesn’t use its power of parallel processing and throughput decreases.

One way, to actively nurture balancing the allocation over all machines, is to use a mutation operator, which does not use a uniform probability for the new allocation values in the chromosome, but chooses the values proportional to how idle a machine is.

**Advanced Cross-Breeding Selection**

The cross-breeding approach allows for a variety of new selection strategies.

At the moment, when two species are crossed-over, one species offspring gets the individuals from one parent and the other offspring the others.

A possibly better approach might be, to merge the individuals from both parents into one bigger gene-pool after the species crossover.

Then, evaluate the fitness for all the individuals for each of the two species, assigning them two fitness values each.

Now, during the crossover of the individuals, they can be selected according to the fitness in regard of the corresponding species. This way, the pool will split up again, but each species had the chance to select from the whole pool.
Bibliography


