Friedrich-Schiller-University Jena
Faculty of Economics and Business Administration
Chair of Business Information Systems, esp. Business Intelligence
Prof. Dr. Simon Emde

Human resource planning as a parallel machine scheduling problem in cooperation with Confovis GmbH

– Master's Thesis –

to be awarded Master of Science (M.Sc.)

Submitted by:
Erik Hersmann
Schneewittchenweg 5
99099, Erfurt
192482

Reviewer:
University Professor
Dr. Simon Emde
Supervisor:
University Professor
Dr. Simon Emde

February 5, 2025

Contents

I List of Figures

II List of Tables

III List of Acronyms, Synonyms

FSU Jena	Friedrich-Schiller-University Jena
Machine	Employee
Job	Task
Seminar	Schooling

1 Introduction

Team leaders and managers are tasked with the difficult problem of allocating personnel to tasks and further optimally growing those employees' proficiency in the relevant areas of knowledge and abilities that are vital to the company's business and value creation. This very scheduling problem, allocating resources to processes or tasks while trying to create a schedule that does not violate any deadlines, might be optimally or nearly optimally solvable for an experienced human resource planner or manager, given that the team size is relatively small. However as the team grows in size and the horizon for a schedule moves further into the future the manually created schedules become less and less optimal, resulting in opportunity costs for the company and at worst a team of unskilled employees and unmet project deadlines. For these reasons this study aims to provide a tool to generate additional insights and help the managers in their decision-making process for creating such a schedule, that is sufficiently close to being optimal, while also matching the skill set of the employees to the ones needed for future tasks.

The partnering company for this study is interested in finding schedules for allocating specialized employees such as programmers or engineers to complex tasks. It is also interested in optimizing the proficiency of the pool employees to match the companies tasks' requirements optimally. An example for this would be a significant amount of tasks in the near future for a software update that all require the executing employee to be knowledgeable about a specific programming language. In this case the manager is now interested in a way to complete all sub-tasks for the software release in a timely manner, without violating any deadlines. However to achieve this, it may be optimal to assign all of the experienced developers to developing, in case this is not enough to meet the deadlines, the schedule should then also train the junior developers or unrelated employees to be able to take over simple tasks like writing documentation or a guide on how to use a certain piece of the software.

In alignment with the requirements set by company, the goals of this study are to provide a tool, "solver" or heuristic approach to solving the problem of assigning a pool of employees of a company, where each employee has a different set of proficiencies and each tasks requires a certain skill and some proficiency in that skill. The tool has to construct a schedule that completes all tasks by assigning any order of tasks to employees, which they should complete and seminars they should attend to further strengthen their proficiency in certain skills. Another important aspect to the company is the aspect that not all employees have the same learning curve and increase their proficiencies at the same rate as other employees. In other words if two employees A and B visited a seminar on algorithm design and employee B is said to be a "bad" learner, then upon finishing the seminar and designing an algorithm for a business application, employee A would outperform employee B. Assuming that both started at a roughly similar initial proficiency in algorithm design. This outperforming would be modeled in terms of time, returning to the prior example employee A could be done in 4 days while employee B might take 5 days. Expanding upon this, if there was a cyclic task of designing algorithms with periodic 4 day deadlines, then the employee A would be able to match that and not miss any deadlines, while employee B or employee A prior to taking part in the seminar would both not be able to keep the deadlines resulting in lateness. The tool should then determine to send employee A to the seminar and let employee A work on the cyclic task, while employee B can be tasked with jobs that require another proficiency.

In the following an example of a problem and a solution is provided to illustrate the studied problem. Considering the following problem instance; 6 different tasks are to be scheduled and only 2 employees are available. Furthermore 4 optional seminars can be assigned. The indexing is as follows; the job indices 0 through 5 belong to jobs and the indices 6 through 9 describe seminars. The numbers in the round brackets are the respective deadlines of each job and None for the seminar, since seminars have no deadline by the used problem definition. The number before the round brackets is the index of the job or seminar. The color-coding represents the type of assigned task, blue and cyan indicate mandatory jobs and orange indicates optional seminars.

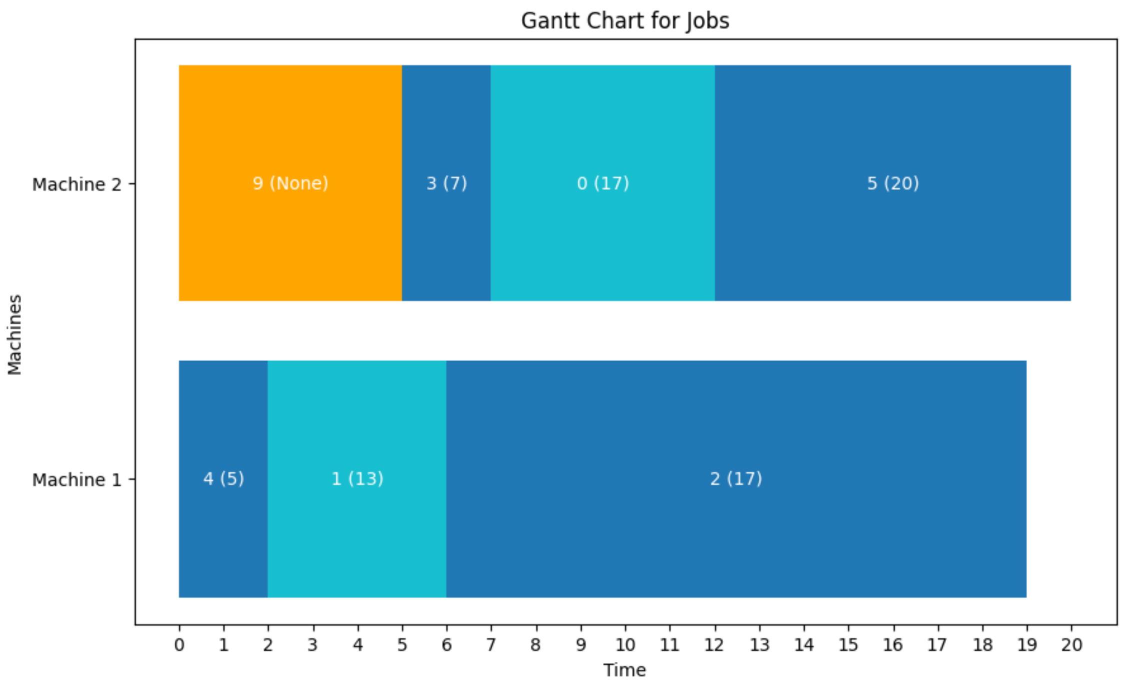

Figure 1.1: Gantt chart for 6 jobs and 2 machines

The figure 1.1 shows a gantt chart for an example problem instance with 6 tasks and 2 employees. The lateness this schedule produces for the given problem is 2, meaning that at least one job is finishing 2 units of time later than the assigned deadline. To verify this one may calculate the difference between the finish time and deadline for all jobs and the maximum of the resulting set; $L^{\text{total}} = \{2-5, 6-13, 19-17, 7-7, 12-17, 20-20\} = \max\{\{-3,-7,2,0,-5,0\}\} = 2$.

The tool itself is implemented in two different ways. One is a perfect solution method that guarantees that for the given problem the solution that is being presented is optimal and no schedule that outperforms the solution exists. This is only practical for very small applications such as 2 employees and 4 tasks for example, which can also easily solved manually. The other implementation is the usage of heuristics and metaheuristics, which may produce a good solution and in most cases offer alternatives to the single best solution (found by the heuristic, not the optimal solution in all cases) in form of other schedules that perform just as well, allowing the decision maker to choose from a set of options instead of just presenting a one-fits-all schedule. This method however does not guarantee perfect solutions and might differ by a large amount from the actual optimal schedule, more on the performance gap in the section on the results of the study.

Using this tool or software is relatively straightforward and follows the procedure of specifying the problem in form of the json file format, one for the employees in which factors such as the learning rate and the initial proficiency levels have to be specified. The second part contains the tasks which have to be completed with their deadlines and approximately required proficiency levels as well as a guess to how long the task processing time might be in a normal setting where the assigned employee roughly fulfills the requirements of the task. One would then interact with the program through the command-line and get the results of the metaheuristics and heuristics as well as the solvers in case the problem instance was small enough. The hyperparameters for the heuristics may be changed in the configuration file, although that is not necessary and the tool will work just fine with the defaults set for all hyperparameters.

The benefits of using such a tool are for one that the planning person, for example a manager has to spend less or almost no time on scheduling large projects and can instead just input the data and choose from the available solutions to the best of their knowledge. This might also yield better results, or any schedules at all for very large problem sizes, such as scheduling a large number of consultants over the time-span of multiple months, which may become very impractical if done manually. While this sounds like a flawless approach, there are multiple shortcomings to consider. For one the assumptions of the model have to hold true, otherwise the results may vary greatly. The following summarizes the assumptions the model makes. Information on tasks and employees has to be at least roughly available, such as proficiency levels of the employees or information regarding the tasks such as deadlines. Furthermore the modeling of learning effects is assumed to be aligned with how learning works in a real business case, which might not be true for some employees. Besides the aforementioned, it is also questionable whether modeling processing times as the factor $\frac{\text{required proficiency level}}{\text{possessed proficiency level}}$ times the average case processing time is a reasonable approach. For example assigning a highly complex programming task to a salesperson might return something like a tenfold or twentyfold average case processing time, however it is not certain whether this is realistic since the salesperson might need to learn programming from zero and might take months to get to a practical proficiency level. Finally these schedules might have no downtimes or in other words leave no room for being tardy with a task. This might prove to be highly unrealistic, for example an employee could run into difficulties and might have to modify the task description to accommodate for more time. The given schedule might prove to be infeasible in that case and the decision-maker would have to recalculate or manually design part of the schedule involving this particular task and employee. This is just to say that the provided tool and schedules are by no means flawless and should only serve as a suggestion or decision-making aid for the decision-maker.

The problem studied in this work is a special instance of the unrelated parallel machine scheduling problem with a deterministic single objective. The task is to assign a number of jobs to a number of machines. Following the $\alpha|\beta|\gamma$ notation for classifying scheduling problems as proposed in Graham et al. 1979 and the extended notation as in Đurasević and Jakobović 2023, section 2 [1], the problem in this study can be represented by the following notation $R|\bar{d_j}, p_c, M_{ded}|Tt$. Where R represents that the machines are not related, $\bar{d_j}, p_c, M_{ded}$ represents the special assumptions, and Tt represents the target performance criterion. The first assumption is that all jobs have a deadline[2], which will be penalized if the finish time of the job is later than the time of deadline of the job. The second assumption is that processing times are dynamic and change over time. The processing times of a job depends on the assigned employee and the time at which the job was assigned, since the proficiency levels of an employee can increase over time and decrease the processing time further. The third and last assumption is that the processing times are faster on some dedicated machines (skilled employees), and the corresponding jobs have a decreased processing time on those machines. The third field describes the performance criterion that is to be optimized. In this study, the total tardiness also represented by Tt should be minimized. Different from the described metric in Đurasević and Jakobović 2023, if a job finishes before the deadline this is rewarded with a negative term in the set. Since this is a minimizing the maximum of sets problem, a good solution tries to balance the jobs to be somewhat evenly finished before their respective deadlines. Since one outlier would force the lateness to be the tardiness of that overly late job.

Usually other performance metrics than the lateness are considered; a counterexample is given in Fry, Armstrong, and Rosen 1990 where mean absolute lateness is considered. In the survey Đurasević and Jakobović 2023, section 4 the authors provide a statistic on which performance criterion is being modeled in which papers. The overarching majority[3] of literature considers the makespan criterion ($C_{\max}$, which is somewhat related to the total tardiness (Tt) or lateness which is only considered

[1] The authors provide an extensive overview of the notation and problem types.

[2] Seminars do not have a deadline.

[3] Over 60% of studies consider makespan as the performance criterion according to Đurasević and Jakobović 2023, section 4.

in around 8% of the studies Đurasević and Jakobović 2023, section 4. Overall total tardiness is the fifth most often considered performance criterion, behind makespan, total weighted tardiness[4], the total weighted flowtime and the special case total flowtime, which sets all job weights to 1. Lateness was chosen because it allows for modeling the fact that deadlines should not be violated if at all possible, if there is a 0 or negative lateness value solution, the exact solution methods will find it. But in case there is no solution that does not at least violate one deadline, this performance criterion still manages to produce a solution to assist the management in decision-finding. If possible the management could also remodel the input parameters to allow for a wider search spectrum, at the cost of a suboptimal representation of reality.

The general structure of this study will be listed as follows; this section serves the purpose of providing a general overview of the goals and the problem of this study. It should serve as an introduction for the reader to get used to the terms and ideas used in the following sections. The next section takes a survey of the existing literature considering the proposed problem instance or loosely similar studies approaching relevant problems. The goal of the literature review is to establish an overview of the current state of research in this general field and also to identify the gaps in research that are to be filled. This should give the reader a good impression of the place the current research is in and also where this work fits in in the current research landscape. The main part of this work is sectioned into the mathematical modeling of the problem with its constraints. A study on the computational complexity of this problem is provided to decide whether this problem is NP-hard. Furthermore another literature review, on the implemented learning effect functions and model behind it, is conducted to justify the choice of the learning curve functions. The solution methods are split into exact methods for solving the mixed integer linear programming formulation and into metaheuristics, which as described earlier will be deployed for larger problem instances. Lastly the results of the solution methods are compared and a final conclusion will be drawn on which methods outperform the others and for which instances this is the case. Other computational studies and their results will also be conducted in this section. Finally after the main part has been concluded by the results of the computational studies, a conclusion together with an outlook for future research will be provided to guide the reader beyond this work.

The main contributions of this work can be summarized as studying and proposing a novel model for the unrelated parallel machine scheduling problem with learning curves and individually decreasing processing times for each machine. In this work 2 exact methods and 2 metaheuristics are proposed to solve the problem and compare the quality of the metaheuristic methods in terms of the optimality gap to exact methods and other criteria. The problem is formally expressed as both a mixed integer linear programming formulation as well as a nonlinear model. Furthermore a functional explanation designed to be accessible to professionals outside the field of

[4]This is similar to the total tardiness in a sense that the total tardiness is the total weighted tardiness with all job weights set to 1.

operations research, including researchers from other disciplines and practitioners in industry (for example the company that is cooperating in this work) is provided. The algorithms itself, or more generally speaking the benchmarking framework allows for an easy reproduction of the results or application of the framework for industry specific input data (for example an actual human resource scheduling problem for a company). The results are statistically compared and a managerial decision-support framework is being provided. Lastly this work aims to uncover further open gaps in research and provide an outlook for future works on this problem or familiar problems.

2 Literature Review

2.1 Introduction

This section should give an insight on the existing literature concerning the problem of the study and also highlight any existing gaps in the current research landscape. Furthermore it should serve as a showcase of any existing debates or disagreements between research results of different studies should they exist. The literature review is structured into an exploration of the existing research and a conclusion thereafter meant for summarizing the results and effects on this work. The reviewed literature is mainly chosen with a focus on heuristics and metaheuristics as opposed to exact solutions, because exact solutions for example in form of a mixed integer linear programming model, might differ greatly in case the problem setup is changed just slightly. Since this approach and model is thought to be quite novel it might be the case in most if not all reviewed studies that the mathematical problem formulation will not be adaptable. In case the identified literature solves a different problem, the differences between the problem in the reviewed study and the problem studied in this work will be made clear to identify whether the results can be directly applied to this study or have to be modified. Given that differences exist, this literature hopes to identify a gap in the existing literature that justifies the purpose of this study by providing novel results and insights for solving a problem that has not been explored in the relevant research area.

The authors of Đurasević and Jakobović 2023 provide an excellent overview of the existing literature on metaheuristics, exact methods and polynomial time approximation schemes concerned with solving, approximating or finding heuristic results for the parallel unrelated machines scheduling problem. The following section will further explore the surveyed papers and compare the problem instances to the one studied in this work.

2.2 Existing Research

Heuristics are a technique employed to solve optimization problems akin to the one being studied in this work. They differ from exact methods in their computational runtime, which has a polynomial runtime or at least can be of polynomial runtime if the heuristic is constructed in a way that allows for it. The heuristic will still return the best found solution out of the ones explored in the given time, meaning that the solution quality heavily depends on the quality of the explored search-space, which is determined by the fit of the heuristic to the problem and data. However a lot of heuristics tend to be constructed greedily and end up optimizing a local area and being stuck in that local maximum with no possibility of getting to a neighborhood that contains the global maximum once stuck. Metaheuristics aim to fix this shortcoming of heuristics, by escaping the local neighborhood and exploring multiple neighborhoods even if they seem less promising (a non-greedy approach). This is usually achieved by introducing a degree of randomness such as a random mutation chance in the genetic algorithm, which changes schedules with a random probability even though that might worsen the schedules' quality. This section explores the existing literature concerning constructing and employing metaheuristics for obtaining high quality solutions for the unrelated parallel machine scheduling problem. The authors of Gandomi et al. 2013 provide an introduction to what a metaheuristic is and some commonly used techniques and elements in them.

Compared to exact methods such as a mixed integer linear programming formulation, the metaheuristics manage to achieve a polynomial runtime as stated above, which renders them viable for larger instance sizes, where a mixed integer linear programming model would become infeasible to compute. The trade-off for the polynomial runtime is that a metaheuristic can not guarantee an optimal solution in polynomial time, nor give an optimality gap or an approximation to optimality gap between the currently best found solution and the theoretical best solution. One framework for evaluating the optimality gap in polynomial time exists and is called polynomial-time approximation scheme (PTAS), however this approach will not be explored in this work and is an outlook for future research concerning the studied problem.

One of the first papers considering the very simple problem of scheduling J jobs on M machines with the optimization criterion being the makespan and no additional constraints being considered is Hariri and Potts 1991, where the authors consider multiple heuristics and problem instance sizes. The benchmarked algorithms are a mixture of dispatching rules by itself and dispatching rules together with local optimization of a graph problem that can be solved in polynomial time. Since this is one of the first studies benchmarking different heuristics there were no other heuristics implemented to benchmark and validate the results against. The instance were constructed by benchmarking all configurations with machine counts taken as an element from the set 5, 10, 20 or 50 and job counts taken from the set 20, 50 and 100. Processing times were given as 4 different distributions similar to the benchmarking study conducted in Fanjul-Peyro and Ruiz 2010. This setup results in a largest instance size of 50 machines and 100 machines.

The study conducted in Orts et al. 2020 is one of the most recent studies, as of the time writing this thesis, considering this particular problem without additional constraints. As described above the focus of contemporary research has shifted towards problem formulations with makespan optimization and additional instances Đurasević and Jakobović 2023, section 3.3.1. The authors apply the scheduling to a real world problem of assigning operating system tasks to hardware processors, namely the central processing units (CPU) and the graphical processing units (GPU) with different specifications, yielding different processing durations, leading to the parallel unrelated machine scheduling problem definition. The authors introduce a genetic scheduler that is tasked with finding the best local solution in the explored search space. The research was conducted with a focus on super-computing and not operations research, hence no exact solution is provided for comparison, since on the problem instance sizes that are relevant to the study no exact methods would be able to perform well[5].

This work[6] also introduces a genetic algorithm for solving the considered problem. The operator design and various other choices regarding the hyperparameters and encoding used in Orts et al. 2020 are different from the ones proposed in this work. For this reason the operational summary of the genetic scheduler (also GenS Orts et al. 2020, section 3.1.) serves as a comparison between the 2 different implementations. The general order of tasks to complete in one iteration is the same for both implementations. First the current parent generation is recombined according to the recombination operator and mutated randomly according to the mutation operator. To compare individuals a fitness value has to be defined and the fitness value for all individuals has to be calculated. Then the children are to be repaired to be valid solutions only and via the selection operator the size of the parent generation is picked and becomes the next iterations parent generation, in some implementations the selection operator also keeps the top k fittest individuals across generations, regardless of selection outcome. It should also be noted that the performance criterion which is being optimized is the makespan.

A schedule is being encoded as a matrix of size J (number of jobs) times M (number of machines), where a column j represents a vector that specifies the counts of assigned j to each machine m for each row m. For example if $x_{3,5} = 2$ this would entail 2 assignments of jobs of type 5 on machine 3. The crossover operator is defined as randomly selecting a crossover point between 1 and $J - 1$ and generating 2 children based on this crossover point by selecting all columns up to the crossover point from parent 1 and the columns past the crossover point from parent 2 for children 1 and in reverse order for children 2. This approach and encoding yields the advantage of not needing a repair operator since each column in itself is valid, hence reordering or swapping columns is not violating that integrity. The mutation operator is defined as a 1% probability on every column, in which case 2 entries are swapped. Selection keeps the top k individuals. The algorithm terminates when the top n% of the k individuals average makespan does not improve over a certain amount of iterations or the overall iteration limit is reached Orts et al. 2020, section 3.1.

The authors compare the results of 5 different solution algorithms. These include the previously described genetic scheduler, a simple greedy dispatching rule, a cyclic dispatching rule, a simple task queue and a double-ended task queue. In the ex-

[5] Machine counts range from 8 to 76 (8 corresponding to 8 GPU and the other configurations being a mix of CPU and GPU), task counts range from 216 to 15625 Orts et al. 2020, section 5 Results.

[6] This work refers to the thesis itself throughout this thesis.

periments performed by the authors the genetic scheduler performs best on all configurations of hardware (machines) with the same job setup for each configuration. The simple task queue and the double-ended queue are dominated by the genetic scheduler[7], the other 2 dispatching rules are dominated across all configurations.

The paper contributes the design of the genetic algorithm (GenS), as well as a public implementation, which can be adapted for other problem instances. However since makespan is considered and the machines do not exercise any kind of learning effects this paper differs from the problem studied in this work. Since for those 2 additional constraints to be considered, namely using lateness as a performance criterion and modeling processing durations as dynamically decreasing processing times over the course of time, a different problem and operator definition is necessary. Furthermore the instance sizes and problem definition as stated differs from the one being studied in this work, nevertheless the genetic scheduler remains relevant to the problem studied in this work.

The authors of Cappadonna, Costa, and Fichera 2012 consider a problem similar to the one being studied in this work, namely finding an optimal schedule for assigning jobs to human resources in form of employees. Specifically a set of J jobs has to be assigned to M machines[8], similar to the main problem being discussed in this work. However this study also considers setup times which are dependent on the worker that performs the setup. Meaning that not only machines are a limited resource, but the workers that perform the setup of a machine before a new job can be processed are also limited. Since the problem is the unrelated parallel machine scheduling problem, the processing times depend on the processing machine and not on the worker performing the setup.

They propose three genetic algorithms for solving the problem by means of using a metaheuristic and also formulate a mixed integer linear programming model tasked with deriving an optimal solution in non-polynomial time. This way the authors were able to compute optimality gaps for instance sizes of 8 or 10 jobs and 4 to 5 machines with 2 to 3 workers for the setup times Cappadonna, Costa, and Fichera 2012, section 5. Furthermore larger instance sizes without an optimality gap are provided for up to 100 jobs, 20 machines and 10 workers. The instances were generated by drawing from an uniform probability distribution, giving no insight into the "hardness" of solving any single instance optimally.

The first of the three genetic algorithms also titled "Permutation-based GA" Cappadonna, Costa, and Fichera 2012, section 4.1 and the second genetic algorithm "Multi-encoding GA" work by encoding solutions to computationally efficiently compute neighborhoods and other genetic algorithm operators. The authors employ the roulette wheel selection with inverse proportional makespan. The third algorithm

[7]The same result is achieved for one configuration, in all other configurations the genetic scheduler outperforms the queuing algorithms.

[8]In this work machines and employees/workers are interchangeable terms. In Cappadonna, Costa, and Fichera 2012 the terms are to be understood literally. In other words a machine being an inanimate object and not a human employee.

combines the first two by executing the permutation based genetic algorithm first and afterwards the multi encoding genetic algorithm.

With a statistical significance of $\alpha = 0.05$ the authors find that for the larger instance size configurations the hybrid algorithm outperforms the other two. These are averaged performances over 3600 created problem instances, which were all solved 5 times by each algorithm and the results were then tested for a statistical significant deviation of the results from each other[9].

While skill dependent setup times (similar to skill dependent processing times) are being modeled here, the workers skill stays constant and so do their setup times. Learning effects are not being introduced in this model, which demonstrates the gap in research concerning a variation of the problem, where the unrelated machines can have decreasing processing times over time. Because of the difference in the problem definition, the genetic algorithms can not be easily applied to the modified problem in this work.

In Mokotoff and Jimeno 2002 the unrelated parallel machine scheduling problem is considered without additional constraints. The authors propose a partial enumeration algorithm that yields very good results on their own instances as well as decent results on the benchmarking study conducted by Fanjul-Peyro and Ruiz 2010. The proposed heuristics work by reducing the mixed integer linear programming formulation down to a smaller problem by either relaxing constraints or replacing constraints with less restrictive ones. The algorithms are then tasked with solving this subproblem either to optimality or until an upper bound in form of a time-limit or similar is reached. The optimization criterion is the makespan and as stated before no additional constraints are considered as this is the standard unrelated parallel machine scheduling problem. All algorithms reduce the problem by considering less binary variables than the original problem statement and then solve until the solution becomes valid otherwise the problem has to be reduced further. The different heuristics differ in the way they further reduce and or solve the reduced problem by different strategies, for example ordering the decision variables by some calculated efficiency coefficient. The results obtained are compared alongside each other and against previous metaheuristics and they authors obtain better results than the considered literature's algorithms up to that point.

The authors of Fanjul-Peyro and Ruiz 2010 the authors criticize that previous studies do not compare the performance of the proposed heuristics, metaheuristics, exact methods and further algorithms with modern commercial solvers, such as CPLEX, tasked with solving the mixed integer linear programming formulation of the unrelated parallel machine scheduling problem. The study provides this exact benchmark in their computational studies and also implement a metaheuristic on their own, as well as comparing the previously described method from Mokotoff and Jimeno 2002. The authors proposed algorithm works by finding an initial solution via a simple greedy method such as picking the fastest machine for each job. Next this initial solutions local neighborhood is searched by a modification algorithm that will find

[9] The authors tested via ANOVA.

the local optimum without a given time-limit. To escape from this local optimum another algorithm is applied to change to a different neighborhood, which is then once again locally optimized. This 2-step loop, also called variable neighborhood descent as described in Hansen and Mladenović 2001, is then repeated until the iteration stopping criterion is reached. In their novel algorithm they propose 4 different neighborhood operators to replace the operators in the variable neighborhood descent loop with. The final 2 metaheuristics are generated by combining the neighborhood operators cheaply and then modifying the solution to yield a local optimum. In their benchmarking study the authors compared all of the above described novel algorithms against the partial enumeration algorithm proposed in Mokotoff and Jimeno 2002 and the recovering beam search algorithm as proposed in Ghirardi and Potts 2005. The chosen instances were all possible combinations of machine count and job count settings. The machine counts were in a set that is given by the elements 10, 20, 30, 40 and 50, with the job counts being in the set that is given by 100, 200, 500 and 1000. The authors take note of how in industry specific applications processing times are not distributed according to a 1 to n scale. As this would in turn mean that if the time unit was for example given in hours there would be jobs with a processing time of 1 hour and other jobs with a processing time of n hours which could be on the magnitude of days, weeks or months. To avoid such huge relative distances between processing times and appease to a more industry near benchmarking, the authors test the above machine count, job count configurations across multiple distributions of processing times and compare the results in that way. A preliminary exact solver test was also employed using CPLEX, however only 3% of the instances could be solved to optimality in the given 2 hour time-limit. The authors then provide an average for each solver across all configurations and distributions given their 15 second CPU time-limit. They arrive at the conclusion that their newly proposed algorithm using all 4 techniques and improving the solution, also called NVST-IG+ is the best algorithm across the ones tested, even outperforming an industrial solver over shorter periods of time, namely CPLEX due to the time constraint. The partial enumeration algorithm as proposed in Mokotoff and Jimeno 2002 is performing very poorly across all distributions. In a further experiment they compare CPLEX and NVST-IG+ for higher time-limits and note that up until 4 to 5 minutes CPLEX is being outperformed by their own NVST-IG+ algorithm, that as they note is very simple and uses little computing power.

Utilizing a genetic algorithm for solving the unrelated machines scheduling problem, with the additional constraint of job release times, with different initialization strategies, the authors of Vlašić, Đurasević, and Jakobović 2019 compare the non dispatching rule initialization performance against the initialization using dispatching rules. The evaluated performance criterion is the total weighted tardiness of the generated solution. The authors tested random initialization, which performed the worst against 4 different dispatching rules. They found that using this method leads to less variance in the first explored neighborhood and therefore reduces the need to run the genetic algorithm multiple times to find a good neighborhood by chance, as with the random initialization. The authors test setup was a job count from 12 to 100 and a machine count from 3 to 10. They conclude that their best initialization strategy, utilizing all dispatching rules for initialization, yielded the best results on average given their test setup.

2.3 Conclusion

It is worth noting that the polynomial time approximation scheme and dispatching rule are not being considered in the computational study and the reviewed literature should serve as a guidance for the interested reader. This is due to the fact that as shown in the survey by Đurasević and Jakobović 2023 dispatching rules tend to underperform compared to metaheuristics for specific problem definitions (such as the one being studied in this work). The polynomial time approximation scheme is being left as an outlook for future research.

All of the above surveyed studies consider a variant of the unrelated parallel machine scheduling problem, however none of the surveyed studies examine the exact problem of this work. No study considered learning effects of machines and or workers in this context, nor decreasing processing times over time. But machine specific processing times, for example faster processing of a graphical processing unit intensive job on a graphical processing unit as shown in Orts et al. 2020 models a similarly related matter. Learning effects of workers will be surveyed in a later part of this work, however none of those later surveyed studies model machines as individuals and rather just assume uniformity. Furthermore a lot of studies in the area of learning effects have no direct connection to operations research, hence no scheduling problem is being solved and the results only apply to modeling the learning curves for the model being formulated in this work.

To summarize common findings in the surveyed literature a lot of the surveyed papers either make use of a genetic algorithm[10] or a simulated annealing algorithm (as described in Anagnostopoulos and Rabadi 2002) or benchmark against it, making it a candidate for implementation in this works computational study as well. All the operators can not be directly adapted from existing implementations since solution validity is given by a different rule-set than the one for the standard unrelated parallel machine scheduling problem. Variable neighborhood descent is just implemented in one surveyed study, namely Fanjul-Peyro and Ruiz 2010, it manages to reach excellent results and the authors expand upon it in Fanjul-Peyro and Ruiz 2011, other authors also used variable neighborhood descent namely in the study Cappadonna, Costa, and Fichera 2012. The authors do not compare it to a genetic algorithm however and the design and implementation may need to be adapted for the problem studied in this work. The less common and in benchmarking results usually shown to be underperforming methods of partial enumeration and methods utilizing just dispatching rules bring a lot of implementation complexity with them and just using existing dispatching rules without problem specific adaption might

[10] Besides the ones surveyed in this literature review the following studies also employ a genetic algorithm for variants of the unrelated parallel machine scheduling problem; Jolai et al. 2011, Peng and Liu 2004 and Lin, Pfund, and Fowler 2011

yield bad results. For this reason they have been chosen not to be implemented in this study. Additionally to note is that more often than not the optimized performance criterion was makespan and not the lateness criterion, which might change the order of best performing methods. Metaheuristics were employed in studies considering total weighted tardiness and as such might be able to adapt better to a change in the problem specification.

As stated above the gap in research is clearly the combination of machine specific processing times and learning effects that allow for a further reduction of those processing times, dynamically based on the jobs that were assigned to the machine earlier in the schedule. The combination of learning effects with classic scheduling of jobs on machines will be thoroughly studied in this work. For this a mixed integer linear programming formulation as well as metaheuristics similar to the ones being surveyed in the literature review will be designed and benchmarked to determine an algorithm that is fit to find good or optimal schedules for this problem.

3 Main Part

This part presents the main concerns of this study. Including a novel mixed integer programming model, an approach to fully enumerating the search space, as well as 3 metaheuristics, a comparison of these methods and finally an advice to managerial decision making based on the outcomes of the computational study.

3.1 Modeling

This section provides the mathematical modeling of the problem studied in this work. First formulations are presented, the first is a non-linear but significantly more intuitive variant of the problem formulation. To work with a commercial solver this formulation has then been rewritten to exclusively contain linear constraints and decision variables that are of binary or integer type.

3.1.1 The Nonlinear Model

Objective Function

$$\min_{X} \left\{ L^{\text{total}} \right\} \tag{3.1}$$

Constraints
Lateness is the sum of differences between end time and the Deadline for all jobs

$$\sum_{t \in T} \left(\left(t + p_{m,j,t} - t_j^{\text{dead}} \right) \cdot x_{m,j,t} \right) \leq L^{\text{total}} \qquad \forall\, j \in J, m \in M \tag{3.2}$$

Machines must have at most one job on them for any given time t

$$\begin{gathered} \sum_{t \in T} x_{m,j_1,t} \cdot (t + p_{m,j_1,t}) \leq \sum_{t \in T} x_{m,j_2,t} \cdot t \quad \oplus \\ \sum_{t \in T} x_{m,j_2,t} \cdot (t + p_{m,j_2,t}) \leq \sum_{t \in T} x_{m,j_1,t} \cdot t \\ \forall\, j_1 \in J, j_2 \in J \setminus \{j_1\}, m \in M \end{gathered} \tag{3.3}$$

Each job has to be completed

$$\sum_{m \in M} \sum_{t \in T} x_{m,j,t} = 1 \qquad \forall j \in J \tag{3.4}$$

Seminars can be completed once at most

$$\sum_{t \in T} x_{m,j,t} \leq 1 \qquad \forall j \in S, m \in M \tag{3.5}$$

Processing-durations are dynamically based on competence

$$p_{m,j,t} = f(j, k_{m,t,w}) \qquad \forall\, j \in J, m \in M, t \in T, w \in W \tag{3.6}$$

$$f(j, k_{m,t,w}) = p_j^{\text{base}} \cdot \frac{j_w}{k_{m,t,w}} \tag{3.7}$$

Competence increases based on the job and the machines learning curve

$$k_{m,t,f} = \max \left\{ \sum_{j \in J} g_m(k_{m,t-1,w}, j) \cdot x_{m,j,t};\ k_{m,t-1,w} \right\} \qquad \forall\, m \in M, t \in T, w \in W \tag{3.8}$$

m	Machine indices (Employees)
M	Set of all machines
j	Job indices (Tasks)
J	Set of all jobs (without trainings)
S	Set of trainings (Subset of J)
t	Time indices
T	Set of all points in time (The maximum time equals Q^{cap} from the section on the MILP model)
w	Skill indices
W	Set of all skills
$x_{m,j,t}$	Binary decision variable that is 1 if job j on machine m starts at t, else 0
X	Set of decision variables $x_{m,j,t}$
$p_{m,j,t}$	Processing duration of job j on machine m at time t
t_j^{dead}	Deadline of job j
$k_{m,t,w}$	Skill-level of employee m at time t in skill w
$f(j, k_{m,t})$	Function that generates processing duration from required skill-level, existing skill-level on m and the baseline duration of the job j
$g_m(k_{m,t,w})$	Machine-specific function that generates the increase in skill based on prior skill of that employee m and the completed job j in t

L^{total}	Lateness for a given solution
p_j^{base}	Baseline processing duration for a job j
l_j^{req}	Required skill level for a job j
w_j^{req}	Required skill type for a job j

Table 3.1: Notation for the nonlinear model

3.1.2 The MILP Model

It is notable that proficiency levels are being modeled as rounded-down integers here, instead of as floats as in the nonlinear model (for a heuristic or metaheuristic approach). This may lead to a slightly worse results given the same schedule, since the learning effect carries over fractionally with the nonlinear approach and results in overall lower processing times than the mixed integer linear programming formulation.

Objective Function

$$\min \left\{ L^{\text{total}} \right\} \tag{3.9}$$

Linear constraints from the nonlinear model

Each job has to be completed

$$\sum_{m \in M} \sum_{t \in T} x_{m,j,t} = 1 \qquad \forall j \in J \tag{3.10}$$

Seminars can be completed at most once

$$\sum_{t \in T} x_{m,j,t} \leq 1 \qquad \forall j \in S, m \in M \tag{3.11}$$

Modified and additional constraints

Lateness is the sum of differences between end time - Deadline for all jobs

$$\sum_{m \in M} \left(\sum_{t \in T} \left(t \cdot x_{m,j,t} + b_{m,j,t} - \left(t_j^{\text{dead}} \cdot x_{m,j,t} \right) \right) \right) \leq L^{\text{total}} \qquad \forall\, j \in J \tag{3.12}$$

Machines must have at most one job on them for any given time t

$$\begin{gathered} \left(\sum_{t \in T} x_{m,j_1,t} \cdot t + b_{m,j_1,t} \right) - Q^{\text{xor}} \cdot (1 - a_{j_1,j_2,m}) \leq \sum_{t \in T} x_{m,j_2,t} \cdot t \\ \left(\sum_{t \in T} x_{m,j_2,t} \cdot t + b_{m,j_2,t} \right) - Q^{\text{xor}} \cdot a_{j_1,j_2,m} \leq \sum_{t \in T} x_{m,j_1,t} \cdot t \\ \forall\, j_1 \in J \cup S, j_2 \in J \cup S \setminus \{ j \in J \mid j \neq j_1 \}, m \in M \end{gathered} \tag{3.13}$$

Auxiliary variable constraints to model the multiplication $b_{m,j,t} = x_{m,j,t} \cdot p_{m,j,t}$

$$b_{m,j,t} \leq x_{m,j,t} \cdot Q^{\text{cap}} \qquad \forall\, j \in J \cup S, m \in M, t \in T \tag{3.14}$$

$$b_{m,j,t} \geq 0 \qquad \forall\, j \in J \cup S, m \in M, t \in T \tag{3.15}$$

$$b_{m,j,t} \leq p_{m,j,t} \qquad \forall\, j \in J \cup S, m \in M, t \in T \tag{3.16}$$

$$b_{m,j,t} \geq p_{m,j,t} - (1 - x_{m,j,t}) \cdot Q^{\text{cap}} \qquad \forall\, j \in J \cup S, m \in M, t \in T \tag{3.17}$$

Calculation of processing durations of jobs

$$p_{m,j,t} \geq \sum_{l \in L} \left(\frac{l_j^{\text{req}} \cdot p_j^{\text{base}}}{l} \cdot y_{m,t,l,w_j} \right) \qquad \forall\, j \in J, m \in M, t \in T \tag{3.18}$$

$$p_{m,j,t} \leq 0.999 + \sum_{l \in L} \left(\frac{l_j^{\text{req}} \cdot p_j^{\text{base}}}{l} \cdot y_{m,t,l,w_j} \right) \qquad \forall\, j \in J, m \in M, t \in T \tag{3.19}$$

Calculation of processing durations of seminars

$$p_{m,j,t} = p_j^{\text{base}} \qquad \forall\, j \in S, m \in M, t \in T \tag{3.20}$$

Skill level at start constraint

$$y_{m,1,l_{m,w}^{\text{start}},w} = 1 \qquad \forall\, m \in M, w \in W \tag{3.21}$$

An employee should always only possess one level for a given skill

$$\sum_{l \in L} y_{m,t,l,w} = 1 \qquad \forall\, m \in M, t \in T, w \in W \tag{3.22}$$

Calculation of the next periods skill level
Skill level should be bigger or equal the new level or 0 (rounded down)

$$\begin{gathered} \sum_{l \in L} y_{m,t+1,l,w} \cdot l \geq \sum_{l \in L} y_{m,t+1,l,w} \cdot l + \\ \sum_{j \in J \cup S} \left(\frac{l^{\text{cap}} \cdot x_{m,j,t} - \left(\sum_{l \in L} c_{m,j,t,l,w} \cdot l \right)}{l^{\text{cap}}} \cdot \alpha_m - 0.999 \cdot x_{m,j,t} \right) \\ \forall\, m \in M, t \in T \setminus \{t \in T \mid t < t^{\text{last}}\}, w \in W \end{gathered} \tag{3.23}$$

Skill level should be equal to the new level (rounded down) if $d_{m,t,w} = 0$

$$\begin{gathered} \sum_{l \in L} y_{m,t+1,l,w} \cdot l \leq Q^{\max} \cdot d_{m,t,w} + \sum_{l \in L} y_{m,t+1,l,w} \cdot l + \\ \sum_{j \in J \cup S} \left(\frac{l^{\text{cap}} \cdot x_{m,j,t} - \left(\sum_{l \in L} c_{m,j,t,l,w} \cdot l \right)}{l^{\text{cap}}} \cdot \alpha_m \right) \\ \forall\, m \in M, t \in T \setminus \{t \in T \mid t < t^{\text{last}}\}, w \in W \end{gathered} \tag{3.24}$$

Skill level should be equal to the old level if $d_{m,t,w} = 1$

$$\sum_{l \in L} y_{m,t+1,l,w} \cdot l \leq Q^{\max} \cdot (1 - d_{m,t,w}) + \sum_{l \in L} y_{m,t,l,w} \cdot l \\ \forall\, m \in M, t \in T \setminus \{t \in T \mid t < t^{\text{last}}\}, w \in W \tag{3.25}$$

Constraints for multiplication auxiliary variable $c_{m,j,t,l,w} = x_{m,j,t} \cdot y_{m,t,l,w}$
Auxiliary variable can only be 1 if a job is started by the employee

$$c_{m,j,t,l,w} \leq x_{m,j,t} \qquad \forall\, m \in M, j \in J \cup S, t \in T, l \in L, w \in W \tag{3.26}$$

Auxiliary variable can only be 1 if the skill limit is possessed by the employee

$$c_{m,j,t,l,w} \leq y_{m,t,l,w} \qquad \forall\, m \in M, j \in J \cup S, t \in T, l \in L, w \in W \tag{3.27}$$

Auxiliary variable is 1 if both x and y are 1, else 0

$$c_{m,j,t,l,w} \geq x_{m,j,t} + y_{m,t,l,w} - 1 \qquad \forall\, m \in M, j \in J \cup S, t \in T, l \in L, w \in W \tag{3.28}$$

l^{cap}	Specifies the upper limit of a skill
$l^{\text{start}}_{m,w}$	Specifies the starting skill level in skill w that an employee m possesses at time 0
$y_{m,t,l,w}$	Binary variable that is 1 if employee m has a proficiency of l in w at time t, else 0
$a_{j_1,j_2,m}$	Binary auxiliary variable that is 1 if job 1 (j_1) comes before job 2 (j_2) on machine m and 0 if j_2 comes before j_1
$b_{m,j,t}$	Integer auxiliary variable that models the result of $x_{m,j,t} \cdot p_{m,j,t}$
$c_{m,j,t,l,w}$	Binary auxiliary variable that models the result $c_{m,j,t,l,w} = x_{m,t,w} \cdot y_{m,t,l,w}$
$d_{m,t,w}$	Binary auxiliary variable that is 1 if the skill w for employee m increased in t
α_m	The relative growth factor for an employee m
Q^{cap}	Big M for the processing time of jobs that satisfies $p_{m,j,t} < Q^{\text{cap}} \quad \forall\, m \in M, j \in J, t \in T$. Calculate the processing durations at the start for each employee assuming all jobs where scheduled linearly on that employee and assuming no improvement in skill, the maximum sum across all employees + the sum of the baseline processing durations of all seminars is Q^{xor}
Q^{xor}	Big M for modeling the exclusive or. This is simply the upper bound for the skill level + 1

$Q^{\max}$	Big M for modeling the maximum function. This is the maximum of the worst possible processing times + 1, assuming no skill decay no job can have a higher processing time than this value. The value is obtained from the calculations of Q^{cap}, by keeping track of the single job maximum throughout.

Table 3.2: Notation for the MILP model

3.1.3 Cardinality Of The Decision Variable Set

In the following the calculations to determine the amount of decision variables needed, dependent on input parameter values, are presented and should be able to offer an insight to how inefficient using an mixed integer linear programming representation of this problem becomes for bigger instance sizes.

- Count of auxiliary variable $a_{j_1,j_2,m} = J^2 \cdot M - J$
- Count of auxiliary variable $b_{m,j,t} = M \cdot J \cdot T$
- Count of auxiliary variable $c_{m,j,t,l,w} = M \cdot J \cdot T \cdot L \cdot W$
- Count of auxiliary variable $d_{m,t,w} = M \cdot T \cdot W$
- Count of decision variable $p_{m,j,t} = M \cdot J \cdot T$
- Count of decision variable $x_{m,j,t} = M \cdot J \cdot T$
- Count of decision variable $y_{m,t,l,w} = M \cdot T \cdot L \cdot W$
- Total count $= (M \cdot J \cdot T) \cdot (3 + L \cdot W) + J^2 \cdot M - J + (M \cdot T \cdot W) \cdot (1 + L)$

Where J denotes the sum of the number of jobs and the number of seminars, M denotes the number of machines, T denotes the maximum possible makespan, L denotes the proficiency levels upper limit and W denotes the number of skills that there are. Of these $((M \cdot J \cdot T) \cdot (1 + L \cdot W) + J^2 \cdot M - J + (M \cdot T \cdot W) \cdot (1 + L))$ are binary decision variables and $2(M \cdot J \cdot T)$ are integer variables with a lower bound of 0 for the auxiliary variable and 1 for the processing time variable respectively and an upper bound of $Q^{\max}$ as derived in the modeling section.

For a relatively small instance like 10 jobs, 5 seminars ($J = 15$), an upper proficiency level limit of $L = 10$, a set of $W = 10$ skills, a time horizon of $T = 150$ [11] and $M = 4$ employees this yields $(4 \cdot 15 \cdot 150) \cdot (3 + 10 \cdot 10) + 15^2 \cdot 4 - 15 + (4 \cdot 150 \cdot 10) \cdot (1 + 10) = 993885$.

[11] This is derived by example from the 15 jobs all having a base processing duration of below 10 and some skill requirements not being met by the "slowest" employee, leading to an increase in processing time. Averaging a processing time of 10 per job times 15 for all jobs and seminars.

3.1.4 Operational Summary Of The Model

This section should serve as another way of describing the problem and a method for optimally solving it in non-polynomial (please refer to the section on polynomial time analysis) time. The goal of this study is to solve the problem of (repeatedly) assigning tasks to a pool of employees. Furthermore each task may or may not have a deadline, before which the task should be completed if possible. Processing times are given by a function of the required skill of a task and the proficiency of the assigned employee. The proficiency of an employee grows by a function of some parameters (the learning curve parameters are individually assignable for each employee, in other words having an adaptable employee grow their skill-set faster than an employee that is not as adaptable) and the current proficiency. Additionally seminars (for example a schooling or seminar of sorts that is not a task that has to be mandatorily completed but rather an optional means through which the proficiency of an employee could be furthered) are to be considered in the model. Another clarification that is necessary is the difference between a general nonlinear and a linearized model or mixed integer linear programming model. The first introduced and shorter model in this study is nonlinear in its constraints. That in return equates to conventional industrial solvers such as GUROBI not being able to solve the problem in its nonlinear form. This makes linearizing the model necessary if one wants to compute optimal solutions for smaller instance sizes with conventional solvers, the following description will therefore focus on the mixed integer linear programming model instead of the nonlinear model.

The performance criterion that is to be optimized is to minimize the maximum lateness, or in other words to minimize *max{start time of a task + processing time of a task - deadline of that task}* or *max{finish time of a task - deadline of a task}* for each task. This reflects the decision-makers wish to schedule all tasks before their respective deadline. With an additional incentive to leave as big of a gap between early finish time and a deadline as possible. Given that all tasks are not late or in other words finish before their deadline. Without any constraints this would be a valid model, consisting of the application specific parameters: tasks, employees and seminars. The decision variables in this case are just the start times encoded as a binary variable which expresses the starting time of a job at a certain time on a certain machine as 1 for those 3 indices m, j, t and 0 otherwise. The same goes for the processing times with the difference between the two being that processing times are integer decision variables and not binary. That means they are able to take on any value between 1 and the upper limit for processing times which can be computed from the input parameters. The calculation involves computing the processing times for all jobs for each employee and then determining the maximum of those processing times across all employees. This process is explained in detail later in this section on the calculation of the upper bound of the lateness. This upper bound of the processing times will never be violated for any schedule and is thus a valid upper bound for the processing time.

Going back to the simple model without any constraints any solver would set all start times to 0, in other words no job is actually executed. The processing times would all be set to their minimum which is given as 1. Hence the lateness would be the negative of the earliest deadline across all jobs plus 1. Obviously this is not a valid solution to any non-trivial problem instance. First and most obviously all jobs should be completed exactly once across all employees, in other words no task should be left unassigned by the end of the scheduling nor should any task be assigned to multiple employees. This is achieved via a summation over the decision variables of the start times as described earlier. The same constraint has to be applied to all seminars with the difference that each employee should be able to complete a seminar once optionally. That means the sum of starting times for each employee has to be 0 or 1 instead of the sum over all employees having to be 1 as with mandatory tasks. The performance criterion, namely the lateness, is also modeled via an individual decision variable that is an integer and can take on any values between a calculated lower and upper bound. The lower bound is obtained by taking the negative of the earliest deadline. This works because no job can finish earlier than 1 since processing times have a lower bound of 1 and assuming all jobs finish at 1 the lateness is then just 1 minus the earliest deadline or the negative of the earliest deadline plus 1. The upper bound of the lateness can be obtained in two steps. First one must compute the processing times of all jobs without learning for each employee. The second step is to sum those processing times up for each employee, in other words this would mean scheduling all tasks on one employee sequentially. Leading to the worst case longest schedules possible (without seminars), the maximum of those longest schedules is the worst case and can be used as the upper bound for the lateness. To account for seminars, to the previously obtained longest schedule end time, one can just add the sum of all fixed processing times of the seminars. This total time is the worst case latest finish time of any task or seminar. This value minus 0 (or the earliest deadline) would be the maximum deadline violation possible assuming that task is to be scheduled last in this worst case schedule. Hence the lateness has an upper bound as just calculated, which in no case can be violated. To properly set the decision variable for the performance criterion to the correct value, in other words the lateness of the obtained solution, another constraint is necessary. This constraint ensures the same calculation as described earlier by calculating the difference between the finish time of a task and the deadline of the task for each task. This is once again achieved via a summation across the start times decision variable, the processing times decision variable and the fixed parameter deadlines.

These three constraints model some of the core requirements of the problem, however any solver would just set the processing times to 0 and schedule all jobs on the first employee and obtain the lower bound of the lateness decision variable as an objective function to any given problem instance. To prevent the solver from scheduling multiple tasks on the same employee before the previous task has been completed another constraint is necessary. This constraint ensures that given two tasks A and B, that if both tasks are to be scheduled on the same employee and task A is scheduled before B, the difference between the starting time of task A and the starting time of task B is at least the processing time of task A. Alternatively if task

B is to be scheduled before task A, the reverse applies. This ensures that no job is scheduled simultaneously on an employee. However the solvers would still obtain the lower bound of the lateness because, processing times are not constrained to be what they should be: a function of the required skill of a task and the proficiency of the assigned employee.

To fix this another constraint and decision variables are necessary. Currently proficiency levels of an employee are not being reflected in the model at all, except for the starting proficiency levels which are given as an input parameter. The decision variable to model employees proficiency is defined for the four indices time, an employee, a proficiency level and a skill (such as a programming language or communication skills etc.) and is 1 if at the given point in time, the employee holds the proficiency in the given skill, otherwise the decision variable is 0. The proficiency levels have to be constrained in some form to disallow the solver from just setting all proficiencies to 1. This can be achieved by constraining each employee and skill combination to only have exactly one proficiency level set to 1 and all other to 0. Furthermore proficiency levels should have a lower bound of 1 to avoid zero division as explained later, at the start (first time index) the decision variables should be set according to input parameters (starting proficiency input parameter for each employee), this is achieved via a summation of the proficiency levels times to decision variable for each employee and skill combination. Additionally proficiency should not be able to decrease or grow past a set maximum. This is achieved via a constraint that limits the proficiency level at each point in time (except for the starting time) to be greater or equal the proficiency level at the prior point in time. An explanation on how to limit the proficiency to an upper limit is given in the paragraph on learning curves. With this new decision variable and the appropriate constraints, the correct processing times can now be set according to a simple function. For each job a required skill is given by the input parameters as well as a proficiency level and a baseline processing duration. The model first determines the proficiency level in the required skill held by the employee and then calculates the processing duration according to the following simple formula: *ceiling of*{*base processing duration times (required proficiency divided by held proficiency)*}. This rewards over-fulfilling the proficiency requirement for a given task by having the base processing duration multiplied with a factor smaller than 1, on the other hand if a proficiency requirement is not met the factor will exceed 1 and the base processing duration will be lengthened. The ceiling function has to be applied to guarantee that the decision variable for processing times remains an integer. This calculation should obviously only be applied to mandatory tasks and not schoolings. The processing times of schooling are given by an input parameter and should remain constant throughout the model, to remain consistent with real world requirements to the given problem.

Currently a solver would just set proficiency levels to the maximum for each point in time except for the starting time and schedule jobs all at that point in time and not the starting point. This is because proficiency levels are not being updated according to a learning function right now but rather just constrained by the constraints defined earlier which are limiting the proficiency levels to an upper bound and one level at a time. That is why another set of constraints becomes necessary to actually update the proficiency levels according to a learning curve function which is defined similarly for each employee, but differs in the function parameters which are given by the input parameters for each employee. The function to calculate the new (for each point in time the following point in time) proficiency level for each skill is given as follows

$$\begin{cases} \text{previous level} + \lfloor \alpha \cdot \text{elapsed} \rfloor & \text{if } m \text{ starts a job } j \text{ that uses skill } w \\ \text{previous level} & \text{else} \end{cases}$$

Where α is a part of the input parameters and represent a learning rate that raises the skill level unless the upper bound has been reached. The growth rate (α) which multiplies the current elapsed proficiency level to the maximum level possible expresses a dynamic learning rate, dependent on the current proficiency level. Please refer to the section on the learning curves and effects, for a more detailed explanation on why this specific family of functions was chosen.

To summarize: The constructed model is given the following input parameters, a list of tasks to be completed. This list includes a deadline, baseline processing duration, required skill and a required proficiency level for each task. The second parameter is a list of seminars, this can be an empty list or a list of seminars, which yield the fixed duration of the seminar and the skill which is being improved as information. Furthermore a list of machines or employees is to be provided that contains a list of proficiency levels at the start (proficiency matrix at $t = 1$), as well as the learning curve specific parameter α that will be used for correctly updating the proficiency levels. Additionally maximum proficiency levels have to be provided for each individual skill or all skills uniformly.

To prevent a solver from freely choosing values for the decision variables and instead making sensible decisions, constraints were introduced. The first constraint is that the completion of all tasks is necessary in order for a solution to be valid. Seminars were made to be optional with a maximum of 1 being assignable per employee. The third major constraint is the exclusion of simultaneous task execution for 1 employee. Another constraint is the calculation of processing times according to the formula given in the paragraphs before. The last 2 major constraints are that proficiency levels should not be able to depreciate and be limited by the maximum obtainable proficiency level and only increase according to the learning function in case a task or seminar has been started.

A solver is now tasked with calculating an optimal assignment to the decision variables, assigning the starting time for each task and deciding which employee executes the task. Also processing durations which can be influenced by the proficiency levels have to be chosen in accordance with the constraints and other decision variables by a solver. Furthermore the proficiency levels for each employee at each point in time have to be decided on as well. Additionally a few auxiliary variables that are necessary for modeling the problem as a solvable mixed integer linear programming model have to be set, for example auxiliary variables for the job order on an employee or the auxiliary variables for the multiplication of starting time and processing time.

The optimal assignment does not violate any constraints and results in the minimum possible value for the objective decision variable, namely the lateness.

3.2 NP-Hardness and Problem Size

The authors of Lenstra and Kan 1979 provide the proof that the simple unrelated parallel machine scheduling problem is NP-Hard. Since the given problem definition is a more complex variation on the proven NP-hard unrelated parallel machine scheduling problem (with more than 2 machines) with additional constraints[12], the problem being studied in this study is therefore also NP-hard.

The problem size can be explained as the size of mixed integer linear programming formulations number of variables for an instance size or as the size of the search space. The first of the 2 has been computed in the section on the mixed integer linear programming formulation and is given as a formula of the 5 input parameters number of jobs and seminars, number of machines, number of skills, calculated upper limit for the time and the number of proficiency levels possible. The second of the 2 problem size definitions, namely the cardinality of the search space has at least been roughly bounded by the upper bound given in the section on the full enumeration algorithm. It is to note that this upper bound is not very exact and tends to overestimate the cardinality by a lot for larger instance sizes. This is because the calculations assume a worst case for a few steps that are guaranteed to not always be worst case.

3.3 Learning Effects Theory

A lot of the relevant and prominent literature in the field of learning curves revolves around decreasing processing times over the course of producing a product series. This is due to the fact that in most optimization problems and modeled manufacturing processes in this field, the executed activities or tasks are the same for each bigger manufacturing cycle. For example screwing the same 5 screws in the same 5 locations and then putting the tools back in their respective places. These kind of repetitive activities do not require a more complex modeling of proficiency levels or competency matrices and can be directly modeled as processing times per type of activity, which with more execution decreases in processing time. That is, workers and technicians become more efficient over time and thus need less time per manufactured piece, leading to a function describing the processing time of the nth piece as a function of n. One of the earliest studies conducted on this functional dependency is Wright 1936, in which the authors model the reduction in cost to produce the nth model of an airplane series as $F = N^x$, where x is an industry specific factor determining how effective a learning effect is. N specifies the produced quantity

[12] Please refer to the problem classification in the introduction.

and F a factor of time required to produce 1 airplane (in relation to the average time required to produce N) to produce twice the quantity. A cited example for the aeronautical manufacturing industry in Wright 1936, section "Effect of Quantity" is $x = 0.322$, which yields that to produce a quantity twice the current factory output N, the average needed construction time per airplane is only 80% of what was needed to produce N in the first place, in other words $F = 0.8$.

The authors of Saraswat and Gorgone 1990 consider learning curves for software developers in the 1990's. The authors note that at the time of writing the paper the estimation of used time and personnel is difficult. This was also confirmed in the computational experiments of this work. The company data from the task planning applications API had major inconsistencies and processing time estimations before the task were usually far off from the actual processing duration after the task was completed. The authors find an inverse relationship between number of installations and the time taken for a single installation, in other words the processing times decline over time as the developers become more proficient. Similar to the study conducted in Wright 1936 the authors find learning curves to be spread around the 20% learning curve, specifically from 6% to 36.2% (Saraswat and Gorgone 1990, Statistical Analysis and Conclusions). The study considered statistical significant testing of a linear model that modeled the relation as $Y = a + bx$. The testing was carried out to confirm that $b > 0$ with statistical significance.

In the study Quarterman Lee 2014, which was carried out by Lee Quarterman for Strategos Inc.[13], the author analyzes the learning curves in manufacturing industries. The findings align with the previously reviewed papers and confirm that there is a nonlinear learning curve described by the function $Y = a \cdot x^{-b}$ in accordance with Wright 1936. The industry or task-specific n-% learning curves were also listed from empirical studies. These range from 5% to 25% and consider mostly manufacturing related activities such as welding or wiring.

The models above are all log-linear models and represent a subset of the proposed log-linear models in the reviewed literature. Another interesting log-linear model was explored in Li and Rajagopalan 1998, the authors proposed a plateau model which limits the decrease of processing time to a certain level beyond which the processing time can not further fall. This is in line with real world applications, where the processing time can not possibly fall infinitely. Unlike the proposed curves in an unlimited model such as Wrights, where if enough iterations are present the processing time nears 0.

Another model studied in Nembhard and Uzumeri 2000 is called the Stanford-B model, this model also takes into account that a worker might bring previous experience from another employer or previous activity. This reflects the problem of this work, where skill levels at $t = 0$ are not all 0 and vary greatly between employees. The formula is given as $Y = a(x + B)^b$, where a is the cost to produce the first piece, x is the piece number produced right now and b is employee or industry-specific learning rate parameter. B is just a constant parameter employed to give

[13] A consulting firm.

the model more freedom to simulate different families of learning curves (Nembhard and Uzumeri 2000).

The issue with implementing the above models is that for every machine-skill combination a learning curve would be necessary, which would also necessitate the modeling the number of previously performed time or tasks related to the given skill by the machine, which would complicate the mixed integer linear program by a large margin. Another issue is the nonlinearity of some of the above families of function, which would have to be linearly approximated to be usable in a mixed integer linear programming formulation. Additionally the learning curve is applied to increase an arbitrary proficiency level and not decreasing the processing directly, which could be substituted in the formula on a one-by-one case for each proficiency level and job requirement, base duration case. Therefore a linear implementation that still captures the idea that learning has diminishing returns for larger time horizons is implemented.

$$l + \left\lfloor \frac{l_{\max} - l}{l_{\max}} \cdot \alpha_m \right\rfloor \tag{3.29}$$

The above formula is the same that is being used throughout this work for modeling the learning curves for employee and aims to model the behavior that learning becomes more inefficient at higher levels and is still relatively quick when proficiency is still low.

Figure 3.1 shows the increase in l on the y axis, or in other words the expression $\lfloor \frac{l_{\max}-l}{l_{\max}} \cdot \alpha_m \rfloor$, as a function of the current l and the current l on the x axis. This can then be interpreted as l plus the value of the function at l is the proficiency level of the employee in $t+1$. For higher values of α the initial surge in skill increase grows by an even bigger amount and converges to 0 for $l \to l_{\max}$ for all values of α. This linear behavior should model the actual learning effects that can be observed in the practical industry applications of learning theory.

3.4 Exact solutions

In this section two exact methods for obtaining a guaranteed optimal schedule for sufficiently small instance sizes are presented. The first method revolves around employing a commercial solver, namely GUROBI, to solve the previously showcased mixed integer programming formulation. Furthermore an algorithm to fully enumerate a given instance sizes solution space is described and implemented. These methods, while achieving the same lateness, might specialize with different instance size properties and may thus in combination yield a bigger number of cases where an optimal solutions lateness value can still be obtained in reasonable runtime. If only one of the two methods had been employed the cardinality of the computable set might be lower.

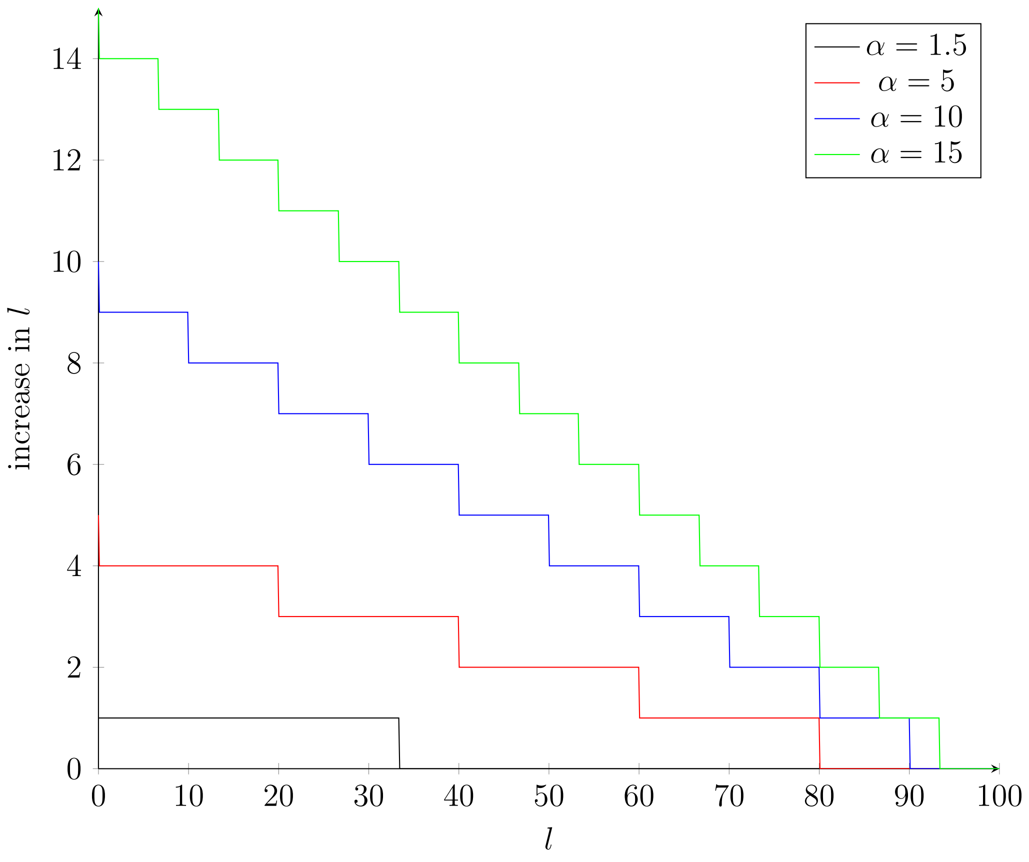

Figure 3.1: Learning curves for different employee parameters

3.4.1 GUROBI

The mixed integer linear programming formulation is being translated into a solver compatible form of constraints, via equations and inequations, with the help of a python software package called pulp. This intermediary form of constraints can the be processed by the commercial solver GUROBI. There are other solvers available but for the purposes of this computational study and given the fact that a majority of the variables are binary variables, as can be seen in the variable count calculations in the mathematical modeling section, GUROBI was chosen over CPLEX. The next step is to prepare an instance, more on this in the computational results section, from which the problem is uniquely identified. Given the jobs, seminars and machines the big M values can be calculated and set as the respective upper and lower bounds of the integer variables and the lateness. With this setup in place GUROBI can then start the actual process of trying to find an optimal schedule by assigning values to the decision variables that do not violate any constraints and produce a schedule that is guaranteed to not be worse than any other feasible schedule in the search space.

To guarantee an optimal solution GUROBI first simplifies the problem formulation and removes unused variables and constraints if possible. Then an initial feasible solution is constructed by various means, including employing heuristics and cutting planes. Once a feasible solution has been found the lateness is no upper bounded by that solutions lateness value and all further explored solutions that, regardless of if they are feasible or infeasible, can be directly discarded without considering them for the computation. GUROBI constructs a tree of candidates and explores it intelligently via calculated metrics and heuristics and also manages to explore unique nodes in parallel, utilizing the hardware. If no time-limit is set GUROBI will then iterate the process of relaxing constraints and or leaving out parts of the problem to construct lower bound estimations and feasible results that, if their lateness value is smaller than the previously best, serve to lower the upper bound. Once the 2 bounds have been brought together, in other words the optimality gap between the best theoretically possible solution and the best feasible solution, GUROBI terminates and yields the found schedule as one possible schedule with an optimal lateness value. The results regarding runtime and being used as a benchmark for the optimal solutions lateness value will be provided in the section on the computational results. The authors of Linderoth, Lodi, et al. 2010 give a mathematical overview of the steps a mixed integer linear programming solver like GUROBI follows and what techniques are commonly used.

3.4.2 Full enumeration of all possible solutions

While this approach might seem very computationally expensive (due to the NP-Hardness of the problem) it is reasonable to compute the set of all possible non-dominated solutions for some smaller instance sizes to compute optimality gaps against heuristics and metaheuristics. The calculations to compute the overall count of solutions for a given instance size will be given as well. The instance size will be denoted as a triple of numbers representing the number of jobs, the number of seminars and the number of machines. For example (3,5,2) or 3JS5M2 denotes all instances with 2 machines, 3 jobs and 5 seminars to be scheduled. One should also note that this full enumeration works independently of the lateness calculation function and without any prior information on what kind of jobs are to be scheduled on which employees with which qualifications. This algorithm rather just returns all mathematical combinations for the problem description. The rules for the combinations can be summarized as follows. In each resulting schedule all jobs should be included exactly once and each seminar can be maximally included once on every employee.

To compute the overall solution count one can either consider the entire search space of all solutions or just the search space of non-dominated solutions. Dominated solutions can be trivially identified by checking the last scheduled element on each machine and if any of the elements is a seminar this solution can be reduced by that surplus seminar without increasing the lateness of that solution. Since the reduced solutions will be contained in our set twice or multiple times, one can discard all these solutions that are reducible. Dominated in this case does not refer to the performance criterion of the lateness in this case, since all seminars are disregarded in the lateness calculation. However the solution is longer, or in other words a variant of the shorter version, which yields a combinatorial amount of "duplicates". For the sake of computational performance of the full enumeration all of these are discarded before being considered for the lateness calculation.

Algorithm 1 Full enumeration of non-dominated schedules

```
 1: $A \leftarrow$ all possible job-machine-assignments                 $\triangleright\ M^J$
 2: $S \leftarrow$ all possible seminar-machines-assignments            $\triangleright\ (2^S)^M$
 3: for $a$ in $A$ do
 4:     $B \leftarrow$ all possible reorderings of $a$                  $\triangleright\ < J!$
 5:     for $b$ in $B$ do
 6:         for $s$ in $S$ do
 7:             $C \leftarrow$ all possible insertions of $s$ into $b$  $\triangleright\ < ((J+1)^S) \cdot S!$
 8:             for $c$ in $C$ do
 9:                 Add $c$ to output if it is not dominated
10:             end for
11:         end for
12:     end for
13: end for
```

The above pseudo-code has been re-implemented in python making use of the built in package *itertools* which provides a large array of out of the box recombination tools which are very useful in this case. One should however note that the implementation in python differs slightly from the order in which the pseudo-code works because this representation is easier to understand. The calculations for the output size should still align regardless of that fact. The algorithm works by first creating two sets through which it later iterates. The first set, denoted as A in the pseudo-code, contains all possible valid job assignments to the machines. This does not mean all ordered assignments, just the pure unordered assignments. For example assigning the set of jobs $J = \{1, 2, 3\}$ to $M = 2$ machines. Could yield the following assignment $[[1, 2], [3]]$, this assignment is exactly the same as $[[2, 1], [3]]$ since the order is being disregarded during the creation of this set. The amount of possible job assignments given the 2 input parameters J (number of jobs to be assigned) and M (the number of machines on which jobs can be assigned on) can be derived as M^J, since each job has to be assigned on one machine, this yields M options for each job, this is then multiplied with the options for the next job and so forth, yielding the first term (M) to the power of the number of jobs (J) as M^J.

The second pre-loop calculated set is the unordered seminar assignment set. This set follows a similar scheme to the first set, with the major difference that the seminar pool only gets exhausted on a machine-basis. That is to say the full set S of all seminars can be assigned to machine 1 and after that still be assigned partly or entirely to machine 2 as well. For a single machine this yields the option of assigning the seminar or not assigning it, meaning per seminar 2 choices are available and this is to be recombined for the number of seminars S. This results in 2^S for a single machine assignment and $(2^S)^M$ for M machines. As with the first set, this does not discriminate between ordered and unordered machines, for both sets the ordering will be resolved in a later step. In the pseudo-code these two sets are denoted as A (job assignment set) and S (seminar assignment set) respectively.

Having constructed the sets A and S, the algorithm then iterates through each possible assignment a in A and constructs a temporary new set B that is derived from the current iterations a, by generating all possible reordered sets per machine and them recombining these orders for all machines, yielding all possible reorderings for that assignment. The worst case for any given a, would be the assignment of all jobs to one machine which would results in $J!$ possible reorders on that machine times 1 since the other machines only have one option (the empty set). Since this Set B is created for each element in A the current total combinations has an upper bound or worst case that can be expressed as $(M^J) \cdot J!$.

The next 2 loops go through the elements of the previously constructed set B (outer loop) and the set S of the seminar assignments from the start. Given that the algorithm now has b, which is a schedule of assigned order-conscious jobs and a seminar assignment s which is not order conscious, one can now construct all possible insertions that respect order of the seminars into the current schedule b. Considering a single machine the worst case is that all jobs J are assigned to the current machine and the machine also has had all seminars S assigned to it. This single machines possible insertions can be calculated in a similar way that the ordered job assignments were calculated. If each position before a job on the current machine would be viewed as a machine, then the problem becomes inserting the list of seminars S order-conscious into J machines. The calculations for this are that of B with different parameters $(M^J) \cdot J!$ becomes $((J+1)^S) \cdot S!$, since the jobs to be assigned are the seminars and the "machines" to be assigned to are the jobs + 1 (the first job has an insertion possibility before and behind, while all other jobs only allow for insertion behind them). Continuing the worst case scenario, all other machines can not have any jobs assigned to them, since all jobs were already assigned to the previously considered machine. However every machine could have assigned every seminar, as explained in the previous sections the algorithm removes "weakly dominated" schedules. Since no job and S seminar schedules are all weakly dominated, these can all be disregarded and only the empty schedules are to be considered for the other machines. This yields a worst case of $((J+1)^S) \cdot S!$. This can not be the actual amount of Insertions since not all s will have this many seminars assigned to the machine with all jobs.

To calculate the current worst case upper bound the multiplication of the sizes of the sets used in the loops is required. The first set is B which is of size $(M^J) \cdot J!$, the second set is S which is of size $(2^S)^M$ and the calculated insertion set C is of size $((J+1)^S) \cdot S!$. This results in $((M^J) \cdot J!) \cdot ((2^S)^M) \cdot (((J+1)^S) \cdot S!)$ for the current upper bound.

Finally for each element c in C, where c is now a full schedule to be used for the output of the algorithm, since C contains all possible schedules at this point, the algorithm checks each schedule for being weakly dominated. This is done by checking the last element of each machine and if that element is a seminar the schedule can be discarded as weakly dominated. This concludes the calculations and explanations of the generation of a full enumeration of all possible schedules for given input parameters. The formula for calculating the number of possible schedules derived from the pseudo-algorithm is $((M^J) \cdot J!) \cdot ((2^S)^M) \cdot (((J+1)^S) \cdot S!)$ and in reality with removing "weakly dominated" schedules and this being an upper bound approximation, the real world values should always be under this upper bound. The below table visualizes the real count against the calculated upper bound for different instance sizes.

Job Count	**Seminar Count**	**Machine Count**	**Solution Count**[14]	**Upper Bound**	**Runtime (ms)**[15]
1	0	2	2	2	< 1
1	1	2	4	16	< 1
1	2	2	10	256	< 1
1	3	2	32	6,144	1
2	0	2	6	8	< 1
2	1	2	20	96	< 1
2	2	2	94	2,304	< 1
2	3	2	708	82,944	9
3	0	2	24	48	< 1
3	1	2	120	768	< 1
3	2	2	888	24,576	3
3	3	2	10,680	1,179,648	69
4	0	2	120	384	< 1
4	1	2	840	7,680	1
4	2	2	8,856	307,200	25
4	3	2	148,296	18,432,000	867
5	0	2	720	3,840	1
5	1	2	6,720	92,160	12
5	2	2	94,800	4,423,680	331
5	3	2	2,063,520	318,504,960	9,681
6	0	2	5,040	46,080	8

Job Count	**Seminar Count**	**Machine Count**	**Solution Count**[14]	**Upper Bound**	**Runtime (ms)**[15]
6	1	2	60,480	1,290,240	171

Table 3.3: Number of possible combinations for varying input sizes

The table 3.3 presents a detailed breakdown of the calculated schedules for different instance sizes. The 3 columns on the left represent the input parameters, the number of jobs, the number of seminars and the number of machines. The 3 columns on the right represent the resulting values, the number of calculated schedules (excluding the "weakly dominated" ones, as those are being parsed out by the algorithm), the calculated upper bound (the theory behind the calculation of this upper bound is explained in the previous paragraphs) and the runtime in milliseconds on the hardware (details are in the footnote of that column). The calculated upper bound aligns with the actual solution count in a sense that no combination of tested input parameters exceeds the upper bound. However the upper bound seems to be very loose, sometimes exceeding the actual solution count by a hundredfold for the bigger instances.

This algorithm will be used for benchmarking the heuristics and metaheuristics for smaller instance sizes, as can be seen from the table above. At certain cutoff points the amount of combinations exceeds the available $\sim$ 15GB of memory and are therefore not computable with the hardware used for this study. The biggest instances the algorithm is able to generate on the hardware were around 8J1S2M. The combinatorial effect takes full effect for larger instance sizes rendering this algorithm practically unusable for real world instance sizes. However it is still of value to compare the results obtained via non exact methods for smaller instance sizes to determine the optimality gap for those instances and maybe have this be an indicator for the optimality gap of larger instance sizes, although the gap could also significantly drift with instance size. No accurate prediction for the optimality gap of larger instances can be derived from the optimality gap of smaller instances. Furthermore this algorithm will be used to compare the results of the other exact method, namely GUROBI, with each other. The results should be equal for all instances that are computable for both algorithms.

[14] This does not include "weakly dominated" schedules.

[15] Hardware Specifications:
CPU: 6-core AMD Ryzen 5 4500U with Radeon Graphics (-MCP-)
speed/min/max: 1397/1400/2375, MHz Kernel: 6.1.0-27-amd64 x86_64
Mem: 15353.2 MiB

3.5 Metaheuristics

The concept of metaheuristics has been introduced in the literature review section already. This section aims to further explain them and go into detail on how the 2 proposed metaheuristics of this work are constructed. A metaheuristic is a higher level abstraction on top of "normal" heuristics such as dispatching rules. A heuristic operates according to a simple rule-set and constructs or modifies a solution or schedule according to the rule-set until some stopping criterion is matched. A metaheuristic combines this simple concept and iterates through a loop using multiple or a singular heuristic that is combined with another rule-set that allows for escaping local extrema. Modifying a solution obtained by one heuristic with another heuristic until a local extrema is reached is another commonly used method, for example in Arani, Momenitabar, and Priyanka 2024, although this study considers a more intricate version of the unrelated parallel machine scheduling problem. The metaheuristic operates with the same concept of matching a stopping criterion that can be more complex than that of a heuristic but is often in form of total number of completed iterations before the algorithm terminates.

Representing a solution via a list for each employee (list of lists[16]) that contains the job order for the specific employee or is empty in case an employee did not complete any jobs. It also contains seminars in the correct order between or before jobs. An example for 2 employees and 3 jobs is given below.

$$\text{example schedule} = \begin{bmatrix} \begin{bmatrix} s_1 & j_3 & j_2 \end{bmatrix} & \begin{bmatrix} j_1 \end{bmatrix} \end{bmatrix}$$

The issue that arises from this solution encoding is that lateness or other performance metrics aren't trivially obtained from just the order. However this can be computationally solved by simulating the learning effects given the machines, job and seminar specifications and the solution itself. For this reason a section on calculating the lateness from such a solution encoding is provided in this chapter as well.

A common idea in the relevant literature (please refer to the literature review section) seems to be an approach of local extrema escaping neighborhood-search that utilizes a greedy local search (for example a taboo-search or simulated annealing, as shown in Park and Y.-D. Kim 1997). When thinking of a neighborhood-search for the problem at hand, it is to be noted that solutions are almost always valid. An invalid solution is only obtained in case a job is assigned more than once or not all jobs are assigned. This makes finding a starting solution, of which to search the neighborhood of, trivial. For example just scheduling all jobs in order of indices on the first machine is a valid solution. When considering neighborhood-operators, only ordering changes on the same machine, transferring a job from one machine to

[16] Other encodings such as a matrix is being used in Chen 1996 and Orts et al. 2020 to represent a schedule. Besides the matrix encoding others are also employed throughout literature and yield the advantage that certain operations become computationally more "cheap" compared to the simple representation.

another or switching two jobs on two machines with each other are to be considered for mandatory tasks. Seminars actually have 4 operators which can be legal depending on the current solution. Deleting a seminar from any employee does not invalidate a valid solution, inserting a seminar on an employee that has not yet have that exact seminar in its order is also legal, furthermore transferring a seminar is also legal in case the target does not have the seminar in its order. Finally switching the order of the seminar on an employee with another task or seminar is also a non-invalidating operation and has to be considered. Given this array of possible operations and exercising each set of operations on each candidate with every target candidate for each operation yields a very large neighborhood for most solutions even on small instance sizes. That is why for the sake of computational complexity it is advantageous to explore the neighborhood randomly and not choose randomly from the entire neighborhood, which requires generating the entire neighborhood first. More on this in the individual sections describing the design of the heuristics.

Another aspect to keep in mind when designing a metaheuristic for this problem is that the search space is very large and even direct neighborhoods are not fully explorable for larger instance sizes. Hence needing to resort to sparsely exploring the neighborhood. A common approach for this[17] is first using a heuristic that generates a lot of promising candidates that are scattered across the search space, for example a genetic algorithm and then using another metaheuristic or a very simple greedy heuristic that then searches the local neighborhood in detail for example simulated annealing. This yields explored neighborhoods of promising candidates and is a good strategy for designing a metaheuristic to deal with the large search space.

3.5.1 Calculating Lateness

The later described metaheuristics in this work, work largely agnostic to the job, seminar and machine specifications. The only information they require is the lateness value for any given schedule, to compare two different solutions in their quality. For this exact reason an algorithm that determines the lateness of a given schedule is required. This algorithm also needs information on job, seminar and machine specifications and can then via simulation of the actual schedule, determine processing times throughout time and with the processing times and order, also determine the finishing and starting times of all jobs. Given the finish time of each job and their deadlines it is trivial to calculate the lateness.

[17] As is shown in the studies on variable neighborhood descent, like in Fanjul-Peyro and Ruiz 2010.

Algorithm 2 Pseudo-code for calculating lateness from a schedule

```
1: $L \leftarrow -\infty$                                  ▷ Variable that corresponds to lateness
2: for $m \in M$ do                                         ▷ Machines are independent
3:     $t \leftarrow 0$                           ▷ Variable that corresponds to the current time
4:     for $j \in X[m]$ do                                  ▷ $X$ is the full schedule
5:         if type($j$) = job then
6:             $p \leftarrow$ calculate_processing_duration($j$, $m$)
7:         else
8:             $p \leftarrow j_{\text{base\_}p}$
9:         end if
10:        $t \leftarrow t + p$
11:        if $(type(\text{j}) = job) \wedge (t - j_{\text{deadline}} > L)$ then
12:            $L \leftarrow t - j_{\text{deadline}}$
13:        end if
14:        $m_w \leftarrow$ update_skill($m$, $j_w$)                    ▷ $w$ is the skill index
15:    end for
16: end for
```

The above pseudo-code serves to illustrate how the algorithm meant for calculating the lateness value[18] works. The algorithm works on each machine individually since, for calculating the lateness and for other purposes, once the schedule has been decided the individual machines do not influence each other and can therefore be calculated in isolation. First an arbitrarily good value of lateness is assigned to the lateness variable L, in this case $-\infty$. Next as described earlier the algorithm iterates through all available single-machine-schedules and sets the variable t, which holds the current time of the order, to 0. It then iterates through all jobs assigned to the machine in the order they were assigned. For each job the algorithm checks the type of job, indices smaller than the number of jobs correspond to a job and indices bigger or equal to the number of jobs correspond to seminars. If a job is scheduled, the algorithm calculates the processing duration according to the formula described in the section on modeling the mixed integer linear programming model. Otherwise a seminar is scheduled and the base processing duration is assigned as processing duration, this is as per problem statement that seminars should not be able to be completed faster or slower than their base processing duration. After this the current time can be updated to the start of the next job or in case the current job is the last job in the schedule, the finish time of the last job. This is achieved by simply adding the processing duration to the current time. To update the lateness two criteria have to be fulfilled. First the current job type has to be of type job and not of type seminar, once again this is checked by calculating whether the index is below the job count or not. If the first criterion is fulfilled the algorithm then subtracts the deadline of the current job from the finish time or the current time, which are equal at this point. If this calculated value is bigger than the current lateness value stored in L, the variable is updated to the new lateness value. Lastly

[18] In case of the genetic algorithm this is also called the fitness operator.

the algorithm updates the skill that was used in the current job or seminar and reassigns it to the skill array of the current employee m. This is repeated for all jobs j on all machines m. The final value of L corresponds to the lateness value of the given schedule[19].

3.5.2 Dispatching or Priority Rules

Dispatching rules are a more straightforward way of creating a schedule than using a metaheuristic, the trade-off is that many dispatching rules tend to be greedy or perform less well than their metaheuristic counterparts. The "rule" is a function that assigns each job and or machine a priority value in the queue and then processes that constructed queue in an orderly manner to assign the highest priority job to the highest priority currently available machine and so forth. An important factor for the performance of such an algorithm is the quality of the "rule" itself and how it matches the problem, generally speaking a good dispatching rule for one problem does not have perform well on another problem that has different premises.

For the construction of these rules there are 2 different approaches; one is to manually create the dispatching rules for each different problem and find a good rule for each by means of trial-and-error. Because this is tedious and impractical for complex problems there has been approaches to automatically create such dispatching rules on a problem-by-problem basis. An example is given in Zhang, Zheng, and Weng 2007, where the authors create the dispatching rules via a Markov-chain decision-making process and each step in that chain is a different manually designed rule given as an action to the agent. This results in a mix of those manual rules and an automatically generated dispatching rule for the problem the algorithm had been applied to.

The reader is referred to Abdeljaoued, Saadani, and Bahroun 2020, section 3.1 and the sufferage heuristic in Maheswaran et al. 1999, page 16 for actual implementations and benchmarking of dispatching rules for the unrelated parallel machine scheduling problem, albeit not the exact same problem setup has been used. This is for the reasons listed in the literature review, as this work is exploring a novel problem specification. The survey in Đurasević and Jakobović 2023 has shown that dispatching rules tend to be outperformed by more intricate metaheuristics and this is also the reason why they are not implemented for the computation study and are not further explored in this work.

3.5.3 Simulated annealing

Simulated annealing is trying to simulate the annealing process of metals, where the metal is heated to a point that allows for easy reshaping and then set to cool to allow for more subtle changes to the shape as the materials gets naturally harder as

[19] This function is being cached in the practical implementation and just returns the previously calculated lateness value for a schedule that had their lateness value calculated beforehand already.

it cools down. The metaheuristic approach of using simulated annealing is trying to achieve the exact same effect of allowing for easy reshape-ability[20] at the start and then gradually shifting towards a more constricted search space, that is more likely to accept only those solutions that are slightly worse or better than the current one. This effect localizes the search towards the end, with the idea that the broad search at the start yields a promising neighborhood, which can then be more greedily explored towards the end of the algorithm. This concept has been successfully applied in various research[21] and or real world applications. The hyperparameters for the temperature, such as starting temperature and rate of cooling, calculation of acceptance of a candidate have to be carefully calibrated as to not allow for a search space that is too wide from start to finish or one that is too constricted at the start to allow for finding a neighborhood with good solution potential.

To choose a potential neighbor the algorithm first generates a list of potential candidates, of which the algorithm then randomly chooses one. This approach does not generate the full one-move-neighborhood (refer to the genetic algorithm for an explanation of the one-move-neighborhood defined for this problem), as this is computationally expensive for larger problem instances. The partial one-move-neighborhood is generated by first deleting random seminars with a higher probability than inserting new seminars, to prevent bloating of the solution with unnecessary seminars. This probability can be specified as part of the hyperparameters or left at the default value of 1, to have an even amount of removal and insertion neighbors in the pool. This only holds for the case of there being removal operations possible, in the case of no seminars being assigned in the current iterations solution the pool will just be extended by adding 2 insertion neighbors. All of these operations are executed on only one randomly chosen machine out of all of the machines in the current solution, this is to keep the searched neighborhood relatively small as compared to expanding this approach to all machines in the current solution. To yield more neighbors, the random choice is based on the length of the chosen machine, where a higher length of the machine results in a higher probability of being selected by the random number generator. The rest of the pool of neighbors is generated by first selecting two different indices out of the range of possible machine indices, note that for this to work the problem instance must have a machine count of greater or equal 2. The algorithm then iterates over all jobs (excluding seminars) in machine 1 and for each of these jobs over all jobs in machine 2. The pool is then extended by a candidate where the jobs of machine 1 and machine 2 (the two jobs in the current iteration of the loop) have been exchanged. For example if the jobs 1, 2, 3 were scheduled on machine 4 before this operation and job 0, 5, 4 were scheduled on machine 2, the algorithm would then in the first inner iteration pick job 1 and job 0 and switch those two with each other. Resulting in the first neighbor being generated as machine 4 having jobs 0,2,3 and machine 2 having jobs 1,5,4. From this pool the algorithm then randomly chooses a single individual and proceeds with the acceptance probability calculation as described above. This procedure is still

[20] Traversing the search space with less restrictions in terms of worsening solution quality.

[21] For example in Anagnostopoulos and Rabadi 2002 or D.-W. Kim et al. 2002.

computationally expensive and renders the algorithm basically unusable for very large instances that have in excess of multiple hundred jobs.

The current temperature in each iteration has to be reduced in order to limit the algorithm to a more local search towards the end (the iteration counter approaching the specified maximum of iterations) and to allow for a wider search in the beginning to escape local extrema. For these reasons one has to design and implement a function that given the current iteration count, the maximum iteration count and the current temperature, calculates an update to the current temperature. This will in most cases be a reduction of the temperature. In this study two different implementations have been tested and the following has been chosen. The first computes the current temperature as $T_{\text{cur}} = T_{\text{start}} \cdot n^{\text{iteration}}$, where iteration is the current iteration counter and n is a value smaller 1 and bigger 0. This leads to larger values for earlier iterations, for example in iteration 5 with an n of 0.99 it results in $T_5 = T_{\text{start}} \cdot 0.951$. This approach is obviously fixed in the temperature reaching 0 independent of the maximum iterations set. The second implemented approach is $T_{\text{cur}} = T_{\text{start}} \cdot \left(1 - \frac{\text{iteration}}{\text{iteration}_{\text{max}}}\right)$, this calculates the remainder of the iterations as a fraction and subtracts it from 1. This yields a value close to 1 for early iterations, hence multiplying the starting temperature with almost 1 yields the starting temperature for the early iterations. For later iterations the fractions value approaches 1 and hence the starting temperature is multiplied by a factor that is nearing 0 and the result is therefore also nearing 0.

To compare the current solution with the solution from the previous iteration, the algorithm first computes the lateness value for both and then makes a decision based on the the temperature and the difference with the help of a function. If the old solutions lateness value is worse than the current solutions, in the case of the lateness this would mean that the new solutions lateness is less than the old solutions, since the problem is a minimization problem, the function immediately returns 1. This means the algorithm unconditionally accepts the new solution because it is better. In the case of the new solution being worse or slightly worse than the current solution the function computes the following $e^{-\frac{\text{lateness}_t - \text{lateness}_{t-1}}{T_{\text{cur}}}}$, this expression achieves the following. First since the current solutions lateness (lateness_t) will always be larger than the previous iterations solution (lateness_{t-1}), the numerator will always be positive and since the temperature also only takes on values above 0 the entire fraction will positive, leading to $e^{-\text{fraction value}}$. This in return will yield 1 for the fraction value being 0 and values between 0 and 1 for all other cases. The temperature decreases with increasing iteration count which leads to the fraction value getting bigger and the entire terms value getting closer to 0 for later iterations. This symbolizes the fact that with increasing time the algorithm "cools down" and is more aversive to picking solutions that worsen the solution value. In this context alternative functions that follow this same general idea are interchangeable and may yield better results for specific problems.

The starting temperature has to be specified, although this can be left at the default value since the update temperature function determines how many iterations the algorithm has to complete in order to reduce the temperature. The same can be said for accepting new solutions, independent of the initial temperature, this can be altered by changing the function itself and how the current temperature is incorporated into the decision process. The stopping criterion in form of the maximum iteration count and maximum iterations without an improvement (if that criterion is used) are to be specified and have an impact on runtime and solution quality. Furthermore the factor for favoring shorter solutions, as described in the paragraph on selecting a random neighbor, can be specified. This value defaults to 1 and represents the factor of "shorter" neighbors to "longer" neighbors. If set to 4 for example, for each neighbor that adds a seminar over the current solution, the algorithm will add 4 neighbors that remove a seminar to the pool, making the probability of the algorithm picking a neighbor that has been generated by removing a seminar 4 times as likely as picking a neighbor that was created by means of adding a seminar.

Algorithm 3 Pseudo-code for the simulated annealing metaheuristic

```
 1: $a \leftarrow$ first individual
 2: $iter \leftarrow 0$
 3: $best \leftarrow \infty$
 4: $T \leftarrow$ starting temperature                          ▷ Hyper-parameter
 5: while stopping criterion not reached do
 6:     $T \leftarrow$ update($T$)
 7:     $B \leftarrow$ 1_move_neighborhood($a$)
 8:     $b \leftarrow$ select_random($B$)
 9:     if $L(b) < best$ then                 ▷ $L()$ is the lateness of an individual
10:         $best \leftarrow L(b)$                            ▷ Update global minimum
11:         $a \leftarrow b$
12:     else if acceptance_probability($b$) $< p$ then          ▷ Hyper-parameter
13:         $a \leftarrow b$
14:     end if
15:     $iter \leftarrow iter + 1$                     ▷ Checking for a stopping criterion
16: end while
```

The algorithm starts by assigning the first individual[22] to the variable a. The iteration counter *iter* is also initialized to 0. To keep track of the globally best solution the variable *best* is initialized to ∞. The starting temperature T is initialized to the value of the hyper-parameter, which has to be chosen by the algorithm designer. This starting value together with the other hyperparameters will later by benchmarked to find good values for general applications.

In each iteration the following procedure has to be followed in order to provide a solution after the algorithm finishes. First the current temperature is updated according to the function previously described. Next a random neighbor is being selected from the neighborhood as described in the paragraph on the neighborhood

[22] This could be a randomly generated schedule or the end result of another metaheuristic, such as in the hybrid algorithm.

exploration. The lateness of the solution is computed and compared with the previous best or tied bests and if the new lateness is lower than any previously found ones, the global best lateness and solution is updated to the current solution. If the current solutions lateness ties the global best, the solution is appended to the list of the global bests. To decide whether the current solution is to be considered as the originating candidate for the next iterations neighborhood search, the acceptance probability is calculated and compared with a random number generator and if it passes that check[23], it will serve as the individual for the next iteration. If the check fails the current solution remains as the start of the next neighborhood search. The iteration counter is then increased by one and the next iteration step is carried out unless a stopping criterion is fulfilled, for example exceeding the maximum iteration count, or a number of iterations having passed without an improvement in the best found solutions lateness value.

3.5.4 Genetic Algorithm

Using a metaheuristic (for example the genetic algorithm) instead of a priority rule (also dispatching rule) yields better results in the reviewed literature, see Đurasević and Jakobović 2023, for this reason implementing a genetic algorithm for the problem is an examined approach in this study. First the required operators and algorithm parameters have to be defined. A typical genetic algorithm makes use of a solution encoding, a fitness value together with an evaluation function that provides the fitness value for a given solution. A selection operator that defines how individuals are compared and eliminated. One or multiple crossover operators that define how 2 individuals (parents) are to be recombined into children solutions. A mutation operator that defines how random mutation on children is applied and a repair operator that defines how invalid solutions, which were constructed by recombination and or mutation are to be changed into valid solutions.

The solution encoding follows the same style as proposed in the section on heuristics, where the indices 1 to $|J|$ represent the jobs and the indices from $|J|+1$ to $|J|+1+|S|$ represent the seminars. The fitness value is the absolute lateness of a given solution, this is the same performance metric as used throughout this study.

As described in the section regarding the design concerns for heuristics and metaheuristics for the given problem, one has to be careful to not explore full neighborhoods as this will lead to non-polynomial runtime for non-trivial cases and instances. With that in mind the next operator of the genetic algorithm is the recombination operator tasked with creating a generation of children, given a generation of parent individuals. In this study random crossover between a pair of parents is chosen as the operator, that means for each employee n_1 and n_2 for parent 1 and parent 2 respectively, the children employee n_3 is computed as follows. First compute the minimum length of the two employees n for parent 1 and parent 2, second take all scheduled jobs and seminars up to the split point from parent 1 and all jobs and

[23]In this case being bigger than the generated number between 0 and 1.

seminars after the split point from parent 2. The resulting order for the employee is the children's employee, which might not be a valid order. Hence requiring a repair operator that is applied to all children after the recombination and mutation operators have been executed. To prevent stagnation and early convergence the parent generation is shuffled before recombination.

The repair operator is naively removing duplicate tasks (global duplicates) or seminars (local duplicates, for example the same seminar twice on a single machine) and adding tasks if the global pool of tasks has not been fully assigned so far. Given a solution the operator proceeds as follows, first keep track of globally assigned jobs. That means across all employees for that solution, parse the current employee and check whether the index is indeed a task and if so, if it has already been assigned previously in this solution. If so, remove this task from the current employee, this of course always leads to the earlier machines being given the tasks since the order is index based. Secondly keep track of the currently assigned seminars and reset this for each new employee in the current solution. Same as with the tasks, if a task index is identified as being a seminar, check whether the current employee already had completed this seminar and if so, remove the current instance of the seminar from the solution. Lastly, to ensure that all mandatory tasks have been completed, after all machines have been iterated over as described before. The list of already assigned tasks is to be compared with the list of all mandatory tasks and if any task is missing from the assigned tasks, that task is inserted at the end of the last employee. Now obviously this does not lead to good solutions across the board, it is however computationally cheap and since the repair operator has to be called on a lot of individuals this trade-off makes sense to allow for the algorithm exploring a bigger neighborhood.

Next is the mutation operator, which is needed to introduce some degree of randomness into the genetic algorithm to escape local maxima and explore a wider neighborhood. Given a mutation probability p, if a random number generator returns a number respective to landing on a p-sized segment of 1, the entire solution is shuffled. That means the order of machines is randomly changed. After this mutation another mutation is possible and will be decided one by one for each individual machine. Each time the above shown random number generator checks whether p has been exceeded and if so, the current machines order will be randomly shuffled. These operations are irrelevant in regard to a solution being valid or invalid, since none of the above described operations add or remove local or global tasks or seminars. Meaning a valid solution remains valid after mutation and an invalid solution remains invalid. An alternative way to mutate would be to just consider seminars. For each employee in a given solution compute the set of not already assigned seminars and then add these with decreasing probability from index 0 to the last index, to utilize the fact that the processing time benefit affects more tasks if scheduled earlier and less tasks if scheduled later. For example if the given employee has 3 tasks assigned in the current solution, the insertion probability for index 0 (before all tasks) could be 0.7, at index 1 (before the last 2 tasks) it could be 0.2 and at index 2 (before the last task) it would result as the remainder of the probability space $1 - 0.7 - 0.2 = 0.1$. As explained in the full enumeration section, the insertion at index 3 is technically not invalid, however the produced solution will be weakly dominated by the same solution not inserting the seminar after all tasks.

The selection operator operates according to the fitness value of individuals, in this case the fitness is the value of the objective function in the mixed integer linear programming model or simply put the lateness. First all children in the current generation obtained from recombination of the parent generation are evaluated with the calculate lateness function, as described in the section on calculating lateness from schedules. Furthermore all duplicate solutions are removed. The resulting list of solutions with their corresponding lateness is then sorted by increasing lateness, since the fittest individuals correspond to the lowest lateness value. The top k values of this sorted list are then chosen as the next parent generation, in this case k is the same parameter as the parent generation size used for initially creating a random parent generation.

To obtain an initial starting population of parent individuals one could simple employ a trivial assignment operator, which would lead to no randomness in the early stages of the algorithm. Hence the need for a semi random algorithm to set up the first generation of k parents. Considering each machine individually a $p = 0.5$ random number generator is queried for success until failure. If the 50% chance is a success, a random element out of the list of all seminars and tasks is picked and added to the end of that machine. If the random number generator returns a failure, the algorithm moves on to the next machine. If the last machine ends on a failure without having added all mandatory tasks, all of those mandatory tasks will be scheduled at the end of a random machine. This process will be repeated k times to generate each parent until the specified parent generation size is reached.

This paragraph summarizes the generation of a solution given the problem specific input parameters with the genetic algorithm. The input parameters are the same throughout this study so the reader can refer to the section on the mixed integer linear programming model, which explains them in detail, a list of jobs, seminars and employees at the start of the calculation (for example t=1) have to be given as parameters. Furthermore the following hyperparameters have to be specified and can be varied to change the behavior of the heuristic. The size of a singular parent generation, note that this number will directly affect the width of the explored neighborhood, since every parent is combined with every other parent. Increasing this will increase the size of the explored neighborhood by the new generation size minus 1. The formula for a given size is $\frac{n(n-1)}{2}$, for example going from a parent generation size of 6 (the size of the explored neighborhood is $\frac{6 \cdot 5}{2} = 15$) to 7 would result in a size of $\frac{7 \cdot 6}{2} = 21$. Since each individual has to be to be evaluated, bigger parent generation sizes slow down the computations. Furthermore a stopping criterion has to be specified, in the case of the genetic algorithm this is the value for the maximum amount of epochs before the algorithm terminates. Similar to the parent generation size parameter, setting this to an arbitrarily high value will cause the algorithm to not terminate in reasonable time, while the quality of the found solutions may not necessarily increase over time. An alternative stopping criterion could be considered, for example the amount of periods since the best solution has not been improved upon, as a percentage of the total iteration stopping criterion. The mutation probability also has to be specified and is vital for how stationary the search will be. The lower this probability is set to, the less mutation will be occurring and the heuristic will explore the local neighborhood in more detail. Setting this parameter to a higher value will cause a higher rate of mutations and therefore causing the produced solutions to not be very local and instead initiate a search that is spread out across the global search space.

Algorithm 4 Pseudo-code for the genetic algorithm

```
1:  $A \leftarrow$ generate first parent generation randomly          ▷ $|A| = k$
2:  $iter \leftarrow 0$
3:  $best \leftarrow \infty$
4:  while stopping criterion not reached do
5:      $C \leftarrow \{\}$                                          ▷ Potential children pool
6:      $B \leftarrow$ all possible non-duplicate pairs of $A$
7:      for $b \in B$ do
8:          $c \leftarrow$ recombine($b$)
9:          $c \leftarrow$ mutate($c$)
10:         $c \leftarrow$ repair($c$)
11:         $C \leftarrow C \cup c$                     ▷ Adding the individual to the pool
12:     end for
13:     $A \leftarrow$ selection($C$)                   ▷ Reducing the pool to the top-k best
14:     for $a \in A$ do
15:         if $L(a) < best$ then                     ▷ $L()$ is the lateness of an individual
16:             $best \leftarrow L(a)$                                 ▷ Update global minimum
17:         end if
18:     end for
19:     $iter \leftarrow iter + 1$                    ▷ Checking for a stopping criterion
20: end while
```

The pseudo-code above describes a full procedure for the genetic algorithm to compute a schedule for the given input values. As described in the paragraph on generating a first parent generation, per semi random generator function, k individuals are produced and result in the first parent generation of iteration 0. This set of individuals is assigned to the variable A. Furthermore the iteration counter *iter* is initialized to 0 and the globally best lateness value *best* is initialized to ∞. Afterwards the while loop is entered, where one iteration of the loop equates to one iteration of the genetic algorithm. In each iteration the algorithm first creates the set C as an empty set and also generates the set B as a set of all possible pairs of parents[24]. The algorithm then iterates through that set B, where each iteration a pair of parents b is recombined into the children candidate individual c. The mu-

[24] Pairs of the same parent, in other words the example pair (parent 3 + parent 3), are not added to B.

tation operator may be applied on c if the mutation probability is triggered. After this the individual is also repaired or trimmed.[25] The individual c, that is now a feasible schedule, will then be added to earlier declared set C, which will contain all the recombination results. Once this set C has been filled with all recombinations in the set B, the algorithm selects the top-k individuals based on the lateness value of the individual. Lastly the algorithm checks for a new global minimum and if found, updates the *best* variable to reflect the new best solutions lateness value. The iteration counter is also incremented by 1 and checked for a stopping criterion before entering the next loop iteration.

3.5.5 Hybrid algorithm

The hybrid algorithm aims to combine the other 2 introduced metaheuristics, namely the genetic algorithm and the simulated annealing algorithm, into a 2-step metaheuristic. The concept is not novel and has been applied by other authors in similar fashion[26] and shown to yield performance comparable or better to the standalone versions of the step algorithms itself.

The idea behind the algorithm is to exploit the search space properties of the genetic algorithm and the simulated annealing algorithm. The genetic algorithm explores a wider neighborhood, due to the fact that many operators were designed with a random element. This allows for non-direct neighbors to be accessed and a more global search-space. With this in mind the idea is to let the genetic algorithm find multiple promising candidates that are not in the same neighborhood and then locally explore those candidates neighborhoods to find the local minimum for each promising neighborhoods. This is accomplished by setting the starting candidate of the simulated annealing algorithm to the global top-k candidates found by the genetic algorithm. The performance of this algorithm is evaluated in the section on the computational results of this work together with the performance of the other metaheuristics.

[25] The process and reasoning for removing trailing seminars is explained in the section on full enumeration of the solution space.

[26] For example in studies considering different optimization problems to the one studied in this work: Smaili 2024 or Xu et al. 2011.

Algorithm 5 Pseudo-code for the hybrid algorithm

```
 1: $A \leftarrow$ top-$k$ individuals generated by the genetic algorithm
 2: $best \leftarrow \infty$
 3: for $a \in A$ do
 4:     $iter \leftarrow 0$
 5:     $T \leftarrow$ starting temperature                          ▷ Hyper-parameter
 6:     while stopping criterion not reached do
 7:         $T \leftarrow$ update($T$)
 8:         $B \leftarrow$ 1_move_neighborhood($a$)
 9:         $b \leftarrow$ select_random($B$)
10:         if $L(b) < best$ then                ▷ $L()$ is the lateness of an individual
11:             $best \leftarrow L(b)$                          ▷ Update global minimum
12:             $a \leftarrow b$
13:         else if acceptance_probability($b$) $< p$ then           ▷ Hyper-parameter
14:             $a \leftarrow b$
15:         end if
16:         $iter \leftarrow iter + 1$                ▷ Checking for a stopping criterion
17:     end while
18: end for
```

The pseudo-code shows the general procedure of the hybrid algorithm. First a generation of potential candidates is generated by executing the genetic algorithm on the problem instance and taking the top-k candidates and storing these as the set A. Then the global optimum *best* is initialized to the worst possible value ∞. In the next step (second step of the 2-step algorithm) for every candidate a in A the simulated annealing algorithm is executed as described above, with the difference that the global best is kept across instance of the simulated annealing algorithm. After every element in A has had their local neighborhood searched, the lateness of the candidate in *best* represents a candidate that is at least as good as what had been found by the genetic algorithm.

3.6 Results of the computational study

The problem instances are randomly generated with the range of the random parameters being reasonable values that align with the real world requirements of the company's tasks. Various instance sizes were multiple times re-generated to average the results and find a heuristic that is able to generalize the best. [27] Smaller instance sizes that are feasible for the exact methods, the bounds for feasibility with the used hardware are described in the section explaining the exact methods, were generated fewer times and allowed for a calculation of an optimality gap between the lateness value of the solutions the metaheuristics found and the optimal solutions lateness

[27] In the computational experiments usually 10 or more random instances were generated for one instance size.

value. For the bigger instance the best metaheuristic is denoted by a **bold** lateness value in that row. Another aspect of the computational experiment, which is to be considered is the setup of the configuration in terms of minimum and maximum values for employee proficiency levels. These 2 values are largely responsible for the amount of variables used in the mixed integer linear programming model, as described in the modeling section. However the learning curve function is largely ineffective for very small values of proficiency, due to rounding for smaller values leading to a constant rounding down of the same value. In other words either the model becomes very large or infeasible or the learning effects are largely being ignored. For this reason very small instances and very large instances are generated with different configurations for deadlines, proficiency levels and requirements and base processing durations.

The metaheuristics and solvers are all being evaluated in terms of runtime and solution quality, which is equivalent to the schedule with the smallest lateness found by that method. As stated above the exact methods will only be evaluated on feasibly small instance sizes, which are denoted by a $*$ behind the instance size tuple. The evaluated methods are 2 exact methods; namely GUROBI and the full enumeration algorithm described earlier, abbreviated as FE. 5 or technically 2 different, metaheuristics are being evaluated. The genetic algorithm (GA) as described above, the simulated annealing (SA) algorithm as described above, the hybrid algorithm (Hybrid) that combines the two and the hyper-parameter searches for the genetic algorithm as well as simulated annealing, being abbreviated as HPS for hyper-parameter search (HPS GA: Hyper-parameter search genetic algorithm, HPS SA: Hyper-parameter search simulated annealing). Furthermore the optimality gap is being evaluated in a separate table that only contains instances that were feasible to the exact methods.

Instance Size	**GUROBI**	**FE**	**GA**	**SA**	**Hybrid**	**HPS GA**	**HPS SA**
J3S1M2*	-6.2	-6.2	-6.2	-5.1	-6.2	-6.2	-6.2
J5S1M2*	-5.5	-5.5	-5.5	-5.5	-5.5	-5.5	-5.5
J4S2M2*	-10	-10	-10	-10	-10	-10	-10
J4S1M2*	-5.6	-5.6	-5.6	-5.6	-5.6	-5.6	-5.6
J3S2M2*	-4	-4	-4	-3.6	-4	-4	-4
J20S0M2	-	-	75.3	61.2	51.4	**43.2**	46.7
J20S5M2	-	-	40.7	49	39	**33.2**	34.5
J20S14M2	-	-	58.4	42.5	42.3	**31.5**	33.3
J20S0M5	-	-	3.6	4.7	2.7	-1.6	**-3.6**
J20S5M5	-	-	4	4	-0.6	-3.1	**-3.9**
J20S14M5	-	-	4.7	3.8	1.6	-2.8	**-4.1**
J20S0M10	-	-	-2.7	-3	-3.8	-3.8	**-4.4**

Instance Size	**GUROBI**	**FE**	**GA**	**SA**	**Hybrid**	**HPS GA**	**HPS SA**
J20S5M10	-	-	-2.8	-2.8	-4.7	-5.6	**-6.4**
J20S14M10	-	-	-1.5	0.1	-3.3	-4.5	**-4.9**
J50S0M2	-	-	236.3	182.6	185.9	177	**152.8**
J50S5M2	-	-	286.5	268	252.3	243.6	**223.4**
J50S14M2	-	-	271.4	179.4	168.2	178.9	**151.8**
J50S0M5	-	-	62.8	36.5	56	39.6	**20.5**
J50S5M5	-	-	78.4	41.5	56.7	48.6	**22.2**
J50S14M5	-	-	68.9	48.5	55.3	48.5	**26.4**
J50S0M10	-	-	37.6	15.2	23.5	22.6	**0.2**
J50S5M10	-	-	27.8	13.2	17.8	17.6	**-0.5**
J50S14M10	-	-	41.4	15.7	35.6	20.2	**-0.4**
J100S0M2	-	-	535.2	437	423.1	471.7	**392.1**

Table 3.4: Average lateness results across all methods

As shown in table 3.4, the results obtained by the hyper-parameter search itself usually dominate the "standard" variants of the metaheuristics, the genetic algorithm, simulated annealing and the hybrid algorithm. Moreover the hyper-parameter search for the simulated annealing algorithm performs better than the hyper-parameter search for the genetic algorithm and at least in terms of lateness for large instances, the hyper-parameter search of the simulated annealing is the best reviewed algorithm on the reviewed instances. The individual instances with their generated jobs and machines are being provided in the appendix and allow for reproducibility of the computational studies. All instance rows denoted with a star, which are small instances, show that all methods yield very similar and except for one or two instances the same optimal result, due to the simplicity of the instance itself. This also incidentally shows that the full enumeration algorithm and GUROBI arrive at the same results for the same instance.

Figure 3.2 shows the absolute deviation of the lateness values obtained by the metaheuristics from the optimal solution value as obtained by the exact methods. Each instance on the x axis corresponds to a set of 10 tests that were attempted and only aborted if GUROBI exceeded the given 300 second time-limit. For most instances the averages are all the same as the entire search space is easily enumerable and even by chance the metaheuristics will find some solution that yields the optimal lateness. This observation serves as a way of confirming that GUROBI does indeed work as expected and produces the same lateness values for instances as the full enumeration algorithm. Furthermore it shows that even though the instance sizes are all relatively small some of the metaheuristics, namely simulated annealing due

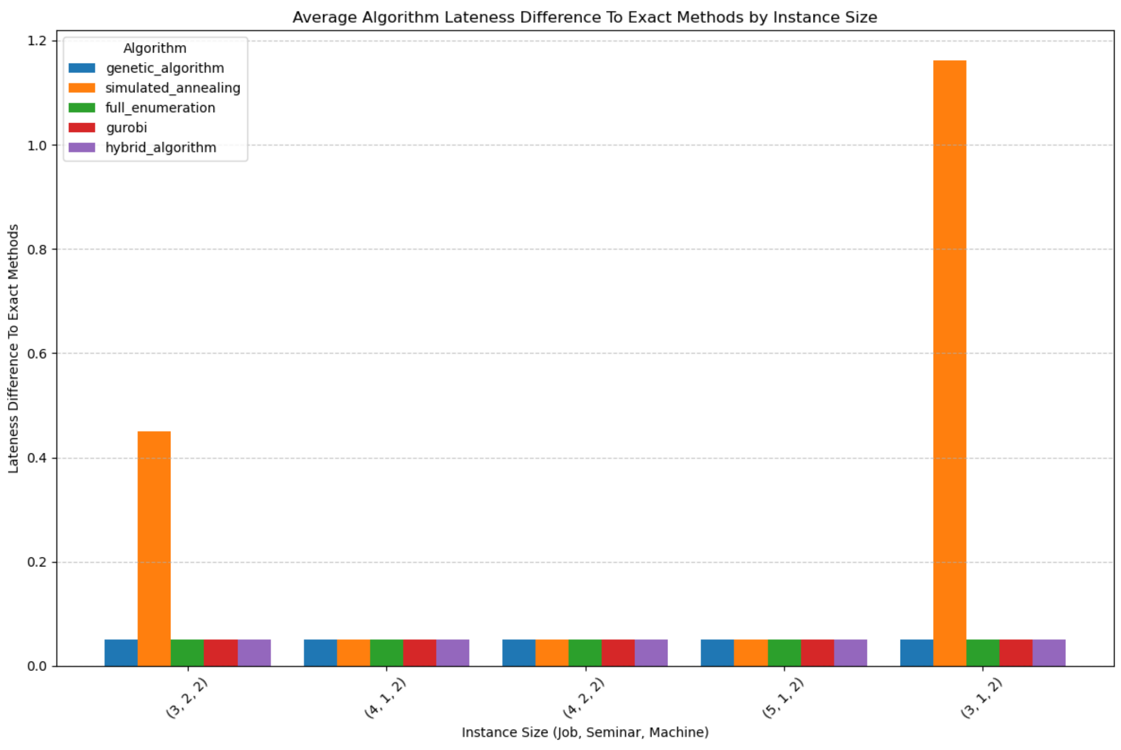

Figure 3.2: Algorithm Lateness Comparison To Exact Methods

to its neighborhood operators, fails to find the optimum in all cases and can not, by construction, explore the full neighborhood. This optimality gap can sadly only be provided for these very small instances since the limit for GUROBI seems to be about 4J2S2M and for the full enumeration algorithm it seems to be around 7J2S2M.

Figure 3.3 shows a histogram of hyper-parameter configurations and the amount of times that each configuration achieved the best results on the instance. Configurations with a value of 1 or 0 were left out for better readability. The tested configurations were all possible combinations of hyperparameters for the genetic algorithm and the simulated annealing algorithm. For the genetic algorithm this included the parent generation size k with possible values of 6, 10 or 15. Second was the maximum possible epoch count before the algorithm terminated, this was a choice between 1000 and 5000 epochs. Lastly the mutation probability is a possible choice between 1%, 10% and 20%. These value choices were chosen as a low, medium, high simulating scenario for each parameter and should provide a good bandwidth of possibilities for the configuration, without having too many possible combinations, which renders the hyper-parameter search ineffective due to combinatorial effects and the runtime requirements associated with it. For the simulated annealing algorithm the parameters to vary are the initial starting temperature T which can take on values of 10, 50 or 100. The second parameter is akin to the genetic algorithm the maximum epoch count before the algorithm terminates. The possible values are 1000, 5000 or 10000, because the simulated annealing algorithm

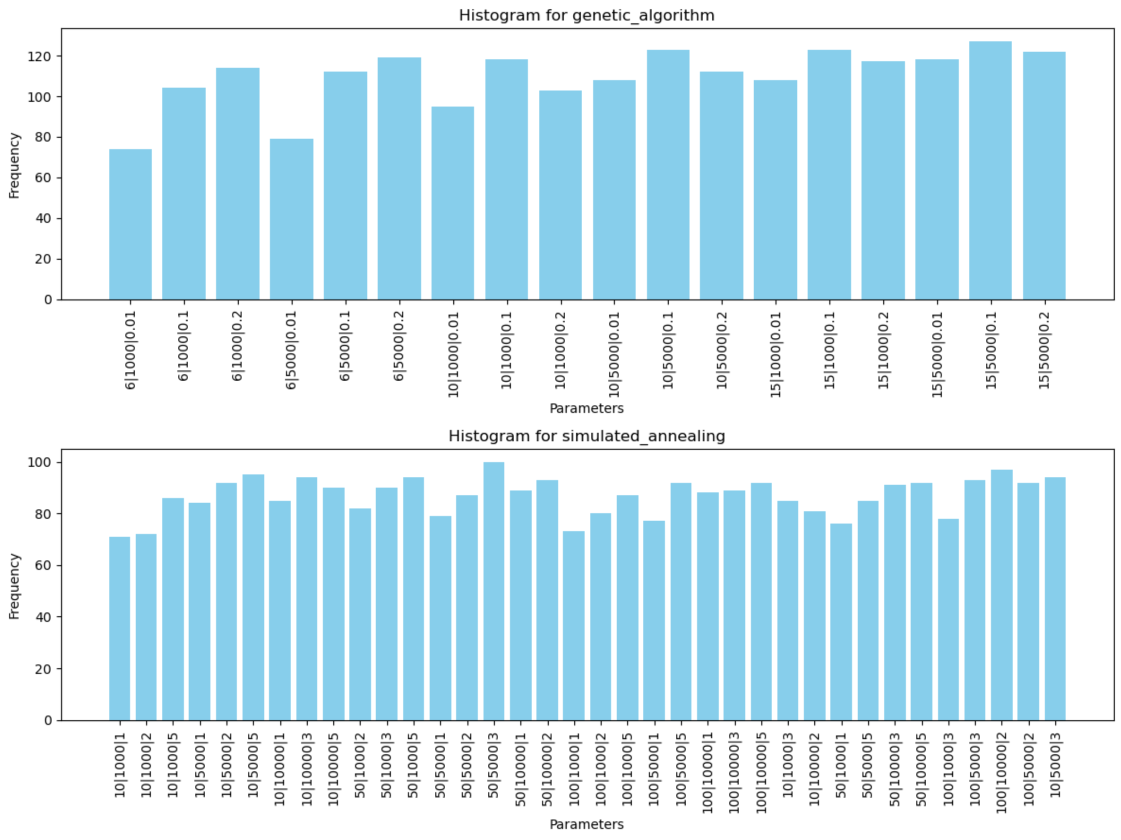

Figure 3.3: Hyper-parameter-search Histogram

implementation of this work tends to by computationally less intensive than the genetic algorithm, allowing for higher epoch counts in the same amount of time. The third parameter is the factor of favoring short solutions by artificially increasing the number of neighbors that were derived by removing seminars. This parameter can take on the values 1, 2, 3 or 5, where 1 corresponds to the standard implementation of the simulated annealing algorithm without favoring any neighbors more or less. However as previously shown the hyper-parameter search does not yield a generally dominant configuration for either algorithm, on the contrary the hyper-parameter search itself is always dominating or equal to any set configuration of the algorithms and is therefore considered as 2 separate metaheuristics in the tables above.

Although not particularly relevant for the application of the methods, for completeness sake the runtime comparison of the different algorithms for all instance sizes is provided in figure 3.4. It can be observed that for the instances, where GUROBI was applicable, GUROBI's runtime was often in the hundreds of seconds, while the other algorithms usually took below a tenth of a second. One fact to note here is that the full enumeration runtime was the computation of best instance from the already generated list of possible solutions for a given problem. The generation of all possible solutions does not have to be performed for every new problem, just for every novel instance size, hence it has been left out of the computation time in this

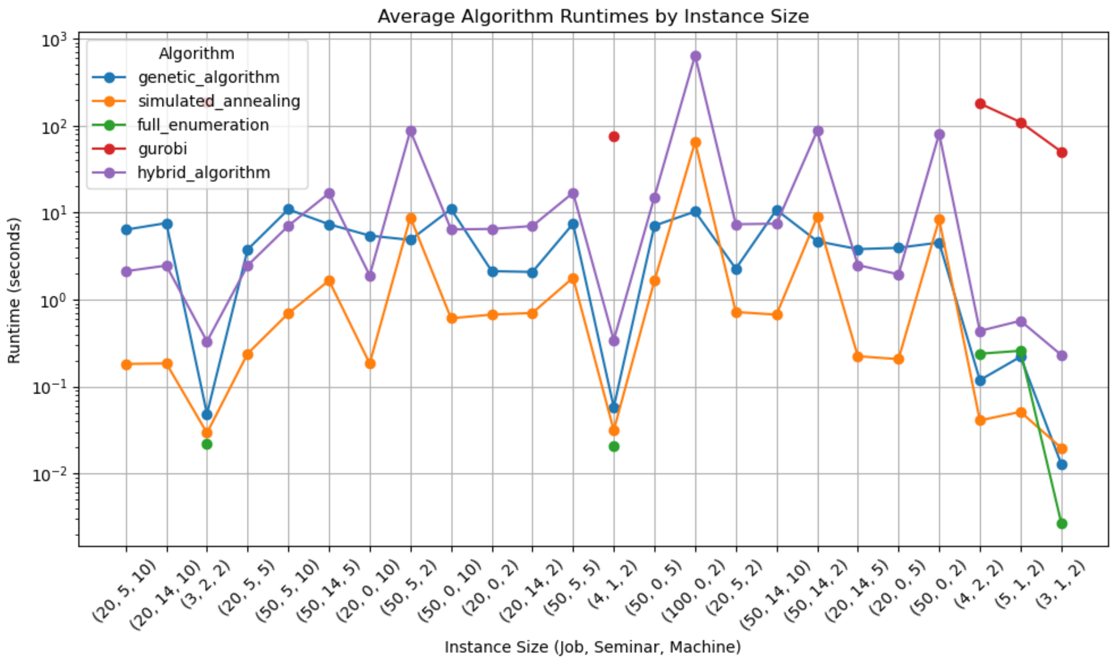

Figure 3.4: Algorithm Runtime Comparison

case. For the hyper-parameter search one can naively multiple the time of the one pass metaheuristic by factor 18 for the genetic algorithm and factor 27 for the simulated annealing algorithm, due to the amount of possible parameter combinations. Since the available time for computing a solution to a short term strategic human resource planning problem will be in the range of multiple hours to multiple days, all shown algorithms except for the exact ones will be feasible to compute. Nevertheless the genetic algorithm tends to be outperformed by the simulated annealing algorithm in terms of runtime, this is due to the fact that the genetic algorithm has to compute more neighbors and the operators are also computationally expensive.

Comparing the constructed metaheuristics has been performed in a quantitative sense in the computational studies above. Another more qualitative comparison will be given to illustrate the differences in the search behavior and quality of the produced solution. The genetic algorithm, as specified in the section on the genetic algorithm, makes use of many random genetic operators and is therefore very volatile in its explored neighborhoods and fails to further locally explore good or promising neighborhoods. This becomes especially apparent for the larger instance sizes were the randomness leads to a lot of unexplored neighborhoods and therefore worse results, as compared to the smaller instances. The simulated annealing algorithm combats this shortcoming of the genetic algorithm by more extensively exploring local neighborhoods even for larger instance sizes, this leads to good results across the board but especially for the large instance sizes with 50 or 100 jobs. The hybrid algorithm which just combines the genetic algorithm and the simulated annealing algorithm does seem to be weighed down by the initial candidate quality provided by the genetic algorithm and therefore seems to be outperformed by the simulated annealing algorithm with a random first candidate generation. This could indeed be improved by searching the top-k candidates found by the genetic algorithm and initializing a new simulated annealing search for each candidate. The hyper-parameter search variants are able to achieve major performance improvement effects through 2 characteristics that are innate to the search itself. First is the elimination of a big part of the randomness of the algorithms initialization together with the initial few neighborhoods. Just by sheer quantity the hyper-parameter search is bound to have initial candidates generated that are not as "unlucky" as the single variant of the respective algorithm, this in return leads to better results towards the end of the algorithm. Secondly as shown in the histogram 3.3 no combination of hyperparameters is clearly dominant for either of the algorithms, which leads the hyper-parameter search to usually outperforming any single combination by the sheer amount of combinations that are tried and might match the current instance best.

The major contributions of this work can be thought of as the following. First a novel or unstudied problem description has been proposed for considering individually decreasing processing times on already individual machines for the unrelated parallel machine scheduling problem. For this novel problem a formal notation in form of a mixed integer linear processing formulation has been provided and linearized to a point where GUROBI, a standard industrial solver, was able to solve smaller instances. Furthermore the learning effects of repeating activities under prior proficiency levels has been modeled and relevant literature has been reviewed. Besides GUROBI another exact method in form of a generator function for fully enumerating all solutions for a given instance size has been implemented and can be used for crosschecking instances or calculate exact results slightly bigger than those solvable by GUROBI. Furthermore 2 metaheuristic and 3 adaptions of those have been implemented and been compared in terms of solution quality and runtime for instances sizes up to instance sizes of 100J14S10M. The first of those metaheuristics is the genetic algorithm as described in the section on metaheuristics, the algorithm itself and ideas for the operators are not novel proposals as genetic algorithms are a commonly employed metaheuristic in the relevant literature as it was surveyed in the literature review section. However some operators had to be adapted especially for this problem formulation and this adaptation is the contribution of this work. The simulated annealing algorithm is not as commonly used in the surveyed literature, but regardless of that it performed well and tends to even outperform the genetic algorithm on larger instance sizes. The hybrid algorithm, which in its design is relatively simple, just running the genetic algorithm and the simulated annealing algorithm as a 2 step algorithm with the genetic algorithm first, does not yield that much better results than just simulated annealing by itself. Which leads to believe that the genetic algorithms final or best individual is not in a good neighborhood and therefore that initialization reduces the simulated annealing algorithms solution quality in the second part. To summarize the main contributions are the proposal of a new problem, the implementation and conception of metaheuristics for it, as well as a benchmarking framework and study for comparing those and further metaheuristics on random instance sizes without the metaheuristic needing to be problem specific for the new problem. The benchmarking framework can of course also be seen as the proposed tool for an assistance in decision making for these kinds of problems, where the managerial decision maker only has to supply a json file with the jobs, seminars and employees defined and the tool would then be able to compute a valid and hopefully decent schedule for the problem data.

The managerial advice derived from the computational study is that in most cases the hyper-parameter search variant of the simulated annealing algorithm outperforms the other algorithm, at least for larger instance sizes. However since the computational cost is relatively low one can afford to run all algorithms and pick one of the schedules with the lowest found lateness value. This might include a choice of multiple different schedules, which could then be used to align with the decision makers own ideas in constructing a schedule and a training schedule of the employees proficiencies. As shown in the hyper-parameter histogram (figure 3.3) no hyper-parameter combination is clearly dominant such that a hyper-parameter search seems advantageous in all cases. Furthermore for very small instances it might be reasonable to fully enumerate the problem, if for example the employee count is 2 and the job count is below 10, additionally without many seminars.

4 Conclusion

This final section serves as a summary of the main points explored in this work, as well as the main findings presented. Further it provides a presentation of the limitations that came apparent. Because of the limitations it also provides an outlook for future studies to provide ideas and unexplored directions for future research. The main points of this work were an introduction to the problem statement and its applications in real world businesses. Next a conclusive literature review into modeling similar problem statements and their metaheuristic solution methods was conducted. This lead to a survey that identified commonly used methods and design choices for the unrelated parallel machine scheduling problem. In the main part of this work the problem was formalized and a mixed integer linear programming formulation was given to mathematically specify the novel problem. A classification of the problem as NP-Hard and the problem sizes was given in the following section. Afterwards a second literature review on the effects of repeating similar activities such as the implementation of software and its learning effects was provided. The results were formalized and the used learning curve formula of the model was derived. The next section focused on the exact solution methods and provided the algorithm for the full enumeration algorithm, as well as the process of solving the mathematical model with an industrial solver. Besides the exact methods for solving the given problem the section on metaheuristics defined 2 metaheuristics and 3 adaptations of those 2, namely the genetic algorithm, the simulated annealing algorithm, the hybrid algorithm that combines the 2, a hyper-parameter search variant of the genetic algorithm and the hyper-parameter search variant of the simulated annealing algorithm. The results determine the hyper-parameter search variant of the simulated annealing algorithm to be the dominant algorithm for a lot of the studied instance sizes.

The main contributions of this work can be summarized as providing the mathematical model for this novel problem, one the nonlinear formalization and secondly the mixed integer linear programming formulation. Second this work provided a benchmarking study and metaheuristics for the novel problem that yielded results on common instance sizes. The metaheuristics are quantitatively and qualitatively compared and a recommendation is derived from the results. A survey of thematically close research is provided as well as a literature review for the learning effects, which influenced the way the model itself was constructed. The framework used for benchmarking can itself be used as a decision-making aid and is therefore also one of the main contributions of this study.

This work was limited in multiple aspects and will refer the interested reader to other literature or the outlook for future research below. The first aspect is the modeling of the learning curves which, due to the limitation set by a mixed integer linear programming compatible formulation, does not model a learning curve as the common formula shown in the surveyed literature. This is also due to the fact that an arbitrary proficiency level and the direct reduction of processing times are 2 different matters. Nevertheless the employed linear learning curve function is limited in that aspect and can not fully capture empirically studied curves. The metaheuristic design is relatively simple and could be improved upon by taking in aspects from other constructed heuristics in the relevant literature. A more extensive testing apparatus for finding optimal hyper-parameters, including function definitions, could be helpful in determining better operators for some of the employed metaheuristics. Problem instances itself are limited in the fact that there does not exist a common set of instances that are used for this problem, since the problem itself is being proposed in this work. This also leads to the issue of no comparison with other authors or studies in regards to method performance. A large limitation is that of the mixed integer linear programming formulation, which in its current formulation is not really usable for any instances with more than 5 jobs and 2 machines and GUROBI as a solver. A better formulation which uses less variables and constraints or one that is solvable in less time for an industrial solver is a task for future research.

An outlook for future research will be provided in the following. Future research may consider employing a stochastic learning effect function for each employee instead of a deterministic one. In a sense that the learning effect for a given employee, with a given skill-set and a job that uses any of the skills the employee possesses, could lead to different outcomes despite the same input. Furthermore the function parameters of the learning curve would then specify a "strong-learner" via parameters that lead to a high probability of an increase in proficiency levels, whereas a "weak-learner" would not be able to increase their proficiency levels with such a high probability. However this would impact the way the nonlinear and linear model have been constructed and would also affect the way the model and heuristics work. The linearization of a stochastic learning function might prove to be difficult. Because of the given reasons this additional model extension has not been studied further in this work. A look into relating research of learning effects, beyond the literature surveyed in this work, could be a good starting point to decide whether to employ non-deterministic elements or whether such an approach does not align with actual learning curves. On another note the implementation of the learning curves from the corresponding literature review conducted in this study could also be good approach to modeling real world environments better. Furthermore it may be a good approach to align seminar starting times with each other in order to reflect the fact that in real world applications of this problem, an employer may not have the means to arrange a seminar multiple times (maybe even in parallel) for each employee. Meaning that the model would have to be rewritten to incorporate this fact, this should be easily incorporable by means of adding an additional constraint. Additionally some of the "optimal" schedules contain little or no assignments or large breaks for some employees, which is not desirable in a real world application. Expanding upon this, besides just looking for the minimum lateness achievable, the solver should optimize towards an evenly distributed workload for all employees. This workload means the average processing time for a given time interval length, such as a week. To ensure that morale and workplace ethics are upheld. Such a criterion would have to be formalized and the metaheuristics performance might have to be reevaluated. As described in Orts et al. 2020, section 3.2, a polynomial time approximation scheme (PTAS) might prove as a viable approximation for finding the optimality gap for a given instance in polynomial time. This approach could be explored to further assist in presenting a more conclusive computational study. The computational study could also be repeated with different metaheuristics as candidates or adapting the proposed metaheuristics with a different architecture or function choices.

In conclusion, this work studied the assignment of tasks such as programming projects to employees with the goal of finishing all tasks before their deadlines. This problem has been mathematically formalized and various methods for solving it have been proposed and benchmarked against each other. Relevant literature regarding the methods for solving as well as modeling individual learning effects has been surveyed in addition. The real world application of the proposed methods has been made possible by the developed framework and can be directly applied to problems as a decision-making aid. The problem is therefore formalized and can also be solved for larger instance sizes such as those observable in industry-based applications.

IV References

Abdeljaoued, Mohamed Amine, Nour El Houda Saadani, and Zied Bahroun (2020). “Heuristic and metaheuristic approaches for parallel machine scheduling under resource constraints”. In: *Operational Research* 20, pp. 2109–2132.

Anagnostopoulos, Georgios C and Ghaith Rabadi (2002). “A simulated annealing algorithm for the unrelated parallel machine scheduling problem”. In: *Proceedings of the 5th Biannual world automation congress*. Vol. 14. IEEE, pp. 115–120.

Arani, Mohammad, Mohsen Momenitabar, and Tazrin Jahan Priyanka (2024). “Unrelated Parallel Machine Scheduling Problem Considering Job Splitting, Inventories, Shortage, and Resource: A Meta-Heuristic Approach”. In: *Systems* 12.2, p. 37.

Cappadonna, Fulvio Antonio, Antonio Costa, and Sergio Fichera (2012). “Three genetic algorithm approaches to the unrelated parallel machine scheduling problem with limited human resources”. In: *International Conference on Evolutionary Computation Theory and Applications*. Vol. 2. SCITEPRESS, pp. 170–175.

Chen, Zhi-Long (1996). “Parallel machine scheduling with time dependent processing times”. In: *Discrete Applied Mathematics* 70.1, pp. 81–93.

Ðurasević, Marko and Domagoj Jakobović (2023). “Heuristic and metaheuristic methods for the parallel unrelated machines scheduling problem: a survey”. In: *Artificial Intelligence Review* 56.4, pp. 3181–3289.

Fanjul-Peyro, Luis and Rubén Ruiz (2010). “Iterated greedy local search methods for unrelated parallel machine scheduling”. In: *European Journal of Operational Research* 207.1, pp. 55–69.

— (2011). “Size-reduction heuristics for the unrelated parallel machines scheduling problem”. In: *Computers & Operations Research* 38.1, pp. 301–309.

Fry, Timothy D, Ronald D Armstrong, and L Drew Rosen (1990). “Single machine scheduling to minimize mean absolute lateness: A heuristic solution”. In: *Computers & operations research* 17.1, pp. 105–112.

Gandomi, Amir Hossein et al. (2013). “Metaheuristic algorithms in modeling and optimization”. In: *Metaheuristic applications in structures and infrastructures* 1, pp. 1–24.

Ghirardi, Marco and Chris N Potts (2005). “Makespan minimization for scheduling unrelated parallel machines: A recovering beam search approach”. In: *European Journal of Operational Research* 165.2, pp. 457–467.

Graham, Ronald Lewis et al. (1979). “Optimization and approximation in deterministic sequencing and scheduling: a survey”. In: *Annals of discrete mathematics*. Vol. 5. Elsevier, pp. 287–326.

Hansen, Pierre and Nenad Mladenović (2001). “Variable neighborhood search: Principles and applications”. In: *European journal of operational research* 130.3, pp. 449–467.

Hariri, AMA and Chris N Potts (1991). “Heuristics for scheduling unrelated parallel machines”. In: *Computers & operations research* 18.3, pp. 323–331.

Jolai, Fariborz et al. (2011). “A hybrid memetic algorithm for maximizing the weighted number of just-in-time jobs on unrelated parallel machines”. In: *Journal of Intelligent Manufacturing* 22, pp. 247–261.

Kim, Dong-Won et al. (2002). “Unrelated parallel machine scheduling with setup times using simulated annealing”. In: *Robotics and Computer-Integrated Manufacturing* 18.3-4, pp. 223–231.

Lenstra, Jan Karel and AHG Rinnooy Kan (1979). “Computational complexity of discrete optimization problems”. In: *Annals of discrete mathematics*. Vol. 4. Elsevier, pp. 121–140.

Li, Georgi and S Rajagopalan (1998). “A learning curve model with knowledge depreciation”. In: *European Journal of Operational Research* 105.1, pp. 143–154.

Lin, Yang-Kuei, Michele E Pfund, and John W Fowler (2011). “Heuristics for minimizing regular performance measures in unrelated parallel machine scheduling problems”. In: *Computers & Operations Research* 38.6, pp. 901–916.

Linderoth, Jeffrey T, Andrea Lodi, et al. (2010). “MILP software”. In: *Wiley encyclopedia of operations research and management science* 5, pp. 3239–3248.

Maheswaran, Muthucumaru et al. (1999). “Dynamic mapping of a class of independent tasks onto heterogeneous computing systems”. In: *Journal of parallel and distributed computing* 59.2, pp. 107–131.

Mokotoff, Ethel and JL Jimeno (2002). “Heuristics based on partial enumeration for the unrelated parallel processor scheduling problem”. In: *Annals of Operations Research* 117, pp. 133–150.

Nembhard, David A and Mustafa V Uzumeri (2000). “An individual-based description of learning within an organization”. In: *IEEE transactions on engineering management* 47.3, pp. 370–378.

Orts, F et al. (2020). “On solving the unrelated parallel machine scheduling problem: active microrheology as a case study”. In: *The Journal of Supercomputing* 76, pp. 8494–8509.

Park, Moon-Won and Yeong-Dae Kim (1997). “Search heuristics for a parallel machine scheduling problem with ready times and due dates”. In: *Computers & Industrial Engineering* 33.3-4, pp. 793–796.

Peng, Jin and Baoding Liu (2004). “Parallel machine scheduling models with fuzzy processing times”. In: *Information Sciences* 166.1-4, pp. 49–66.

Quarterman Lee, PE (2014). *Learning & Experience Curves In Manufacturing.*

Saraswat, Satya Prakash and John T Gorgone (1990). “Organizational learning curve in software installation: an empirical investigation”. In: *Information & management* 19.1, pp. 53–59.

Smaili, F (2024). “A hybrid genetic-simulated annealing algorithm for multiple traveling salesman problems”. In: *Decision Science Letters* 13.3, pp. 709–728.

Vlašić, Ivan, Marko Đurasević, and Domagoj Jakobović (2019). “Improving genetic algorithm performance by population initialisation with dispatching rules”. In: *Computers & Industrial Engineering* 137, p. 106030.

Wright, Theodore P (1936). “Factors affecting the cost of airplanes”. In: *Journal of the aeronautical sciences* 3.4, pp. 122–128.

Xu, Qiaoling et al. (2011). “A robust adaptive hybrid genetic simulated annealing algorithm for the global optimization of multimodal functions”. In: *2011 Chinese Control and Decision Conference (CCDC)*. IEEE, pp. 7–12.

Zhang, Zhicong, Li Zheng, and Michael X Weng (2007). “Dynamic parallel machine scheduling with mean weighted tardiness objective by Q-Learning”. In: *The International Journal of Advanced Manufacturing Technology* 34, pp. 968–980.

V Appendix

The computational experiments with the respective code can be found in the gitlab repo at `https://git.uni-jena.de/pi83xax/masters_thesis`.

List of Algorithms

Statutory Declaration:

1. I hereby confirm that this work — or in case of group work, the contribution for which I am responsible and which I have clearly identified as such — is my own work and that I have not used any sources or resources other than those referenced. I take responsibility for the quality of this text and its content and have ensured that all information and arguments provided are substantiated with or supported by appropriate academic sources. I have clearly identified and fully referenced any material such as text passages, thoughts, concepts or graphics that I have directly or indirectly copied from the work of others or my own previous work. Except where stated otherwise by reference or acknowledgment, the work presented is my own in terms of copyright.

2. I understand that this declaration also applies to generative AI tools which cannot be cited (hereinafter referred to as 'generative AI'). I understand that the use of generative AI is not permitted unless the examiner has explicitly authorized its use (Declaration of Permitted Resources). Where the use of generative AI was permitted, I confirm that I have only used it as a resource and that this work is largely my own original work. I take full responsibility for any AI-generated content I included in my work. Where the use of generative AI was permitted to compose this work, I have acknowledged its use in a separate appendix. This appendix includes information about which AI tool was used or a detailed description of how it was used in accordance with the requirements specified in the examiner's Declaration of Permitted Resources. I have read and understood the requirements contained therein and any use of generative AI in this work has been acknowledged accordingly (e.g. type, purpose and scope as well as specific instructions on how to acknowledge its use).

3. I also confirm that this work has not been previously submitted in an identical or similar form to any other examination authority in Germany or abroad, and that it has not been previously published in German or any other language.

4. I am aware that any failure to observe the aforementioned points may lead to the imposition of penalties in accordance with the relevant examination regulations. In particular, this may include that my work will be classified as deception and marked as failed. Repeated or severe attempts to deceive may also lead to a temporary or permanent exclusion from further assessments in my degree programme.

Place, Date	Signature