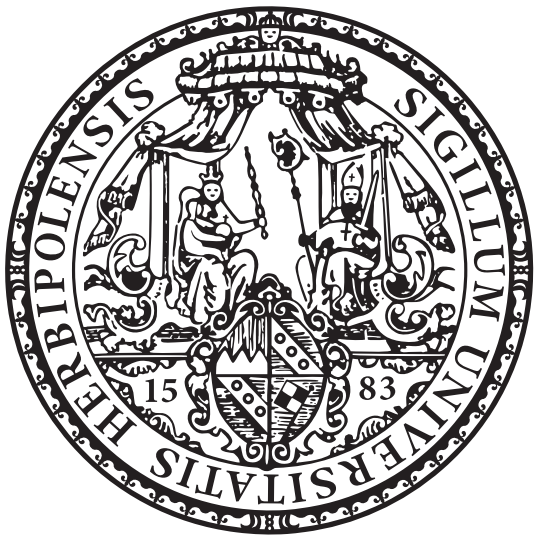
Master Thesis

Optimized Work Schedule Generation in Construction Projects

Yasin Raies

Date of Submission: 04.06.2024
Advisors: Prof. Dr. Marie Schmidt
Prof. Dr. Sebastian von Mammen



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Zusammenfassung

Diese Arbeit formuliert das Project Scheduling Problem with Marked Activities (PSP/mark), eine Erweiterung des klassischen Project Scheduling Problems, bei der Aktivitäten mit umweltbezogenen Eigenschaften wie Lautstärke oder Standort markiert werden, um die Einhaltung eines Arbeitsplans zu gewährleisten. Arbeitspläne geben vor, wann Aktivitäten mit bestimmten Eigenschaften erlaubt oder verboten sind. Darüber hinaus definiert diese Arbeit das Work Schedule Generation Problem (WSGP), das eine optimale Rekombination gegebener Arbeitspläne zur Minimierung einer komplexen Kostenfunktion anstrebt. Die Kosten werden mithilfe einer Lösung des PSP/mark berechnet.

Zur Lösung des PSP/mark wird ein Verzeitlichungsschema formuliert, das Aktivitäten „So bald wie möglich“(As Soon As Possible) einplant. Zur Lösung des WSGP werden zwei Encodierungen vorgestellt, welche in einem evolutionären Algorithmus eingesetzt werden. Durch die Anpassung eines Datensatzes realer Projekte wurden vier Benchmarkdatensätze erstellt, mit denen die vorgeschlagenen Encodierungen bewertet wurden. Die Ergebnisse dieser Auswertung zeigten im Schnitt Verbesserungen von 10% und 14% im Vergleich zu trivial errechneten Ausgangswerten. Jedoch ist dadurch noch keine konkrete Aussage über die Anwendbarkeit und Ergebnisse der Encodings an echten Instanzen des WSGP zu treffen. Daher liegt der bedeutendere Beitrag dieser Arbeit mehr im Hervorheben und Erforschen der genannten Probleme an sich, als den angewandten Lösungen.

Abstract

This work introduces the Project Scheduling Problem with Marked Activities (PSP/mark), an extension of the classical Project Scheduling Problem where activities are annotated with environment-related properties, such as loudness or location, to ensure compliance with a work plan. Work plans specify when activities with certain properties are allowed or forbidden. Additionally, this work defines the Work Schedule Generation Problem (WSGP) which seeks an optimal recombination of given work plans to minimize a complex cost function. To calculate the costs, a solution to the PSP/mark is utilized.

An ASAP (As Soon As Possible) scheduling scheme is proposed for the PSP/mark, while two genetic encodings for WSGP solutions are introduced for use with an evolutionary algorithm. By adapting a dataset of real-world projects, four benchmarking datasets for the WSGP were created and subsequently used for evaluation of the genetic encodings. The results show improvements of 10% and 14% in comparison to a trivial baseline. The applicability of these results to real-word instances of the WSGP is pending. Therefore, the more significant contribution of this work lies in the general insights gained into the novel scheduling problems, opening up avenues of future work.

Contents

1. Introduction

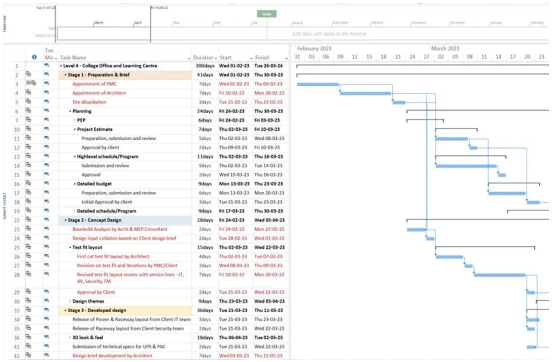
Large construction projects are endeavors of coordination involving hundreds of workers, companies, stakeholders, etc., yielding millions to billions in cost. Infrastructure projects are particularly critical as they may affect the public due to disruptions in availability of roads, rail service or goods. To minimize such disruptions and resulting costs, rigorous and resilient planning is needed. Moreover, it would be highly beneficial if the resulting plan could be adapted to the various outcomes when it meets reality.

Classically, projects are planned by creating a *work breakdown structure (WBS)* which entails all expected outcomes of the project. Next, actions required to achieve these outcomes are derived from the WBS, estimating their duration through rough empirical values and standardized buffer factors [1, Section 4]. Errors in judgement may add up and result in a critically imprecise estimation, as actions often describe high level activities instead of low level tasks, about which one could reason more easily. Adding *precedences*, i.e. dependencies between the actions, results in *precedence diagrams*, often represented as digital Gantt charts or physically through walls of sticky notes. Both representations can be seen in Figures 1.1a and 1.1b. Due to the manual nature of these methods, easy and fast re-planning of projects from the ground up has not been widely possible.

BII GmbH aims to contribute to a solution of these problems through *dProB*, a software environment seen in Figure 1.1c, which integrates into existing *Building Information Modeling (BIM)* workflows. By providing means to efficiently plan low level tasks, whose durations are calculated from utilized equipment and BIM model parameters, dProB helps to create detailed plans, which can be verified through an interactive 3D visualization of models and processes. As tasks expose a series of calculated parameters, custom performance indicators can be derived bottom-up, e.g. estimating CO2 or cost at any point in time during the project.

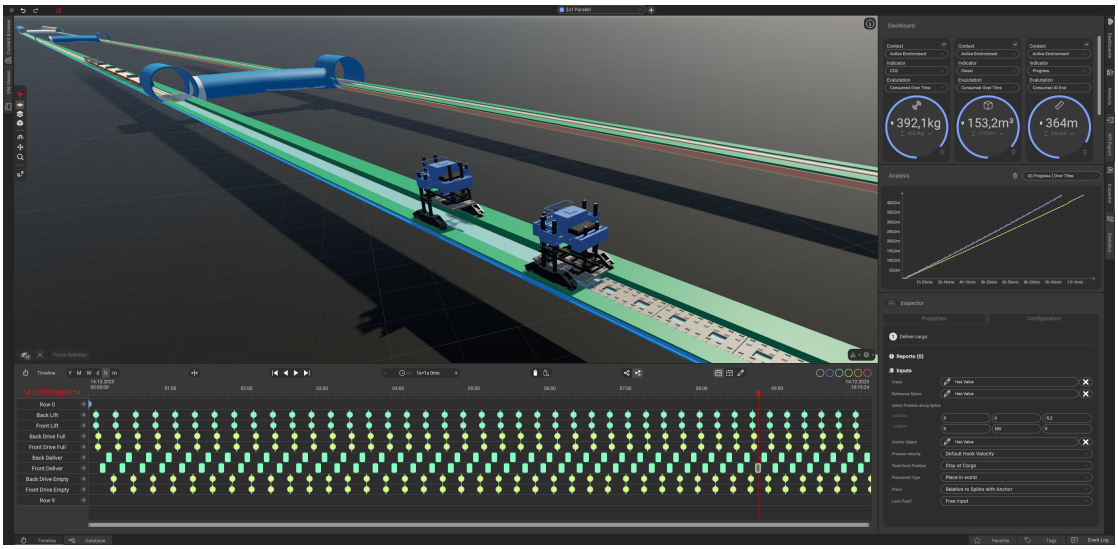
While to this day dProB concerns itself with the creation and precedence of tasks as well as the assignment of resources (equipment and materials) to said tasks, determining when they should happen is rather rudimentary: excluding the compliance with precedence constraints, all start times have to be input manually via delays to other tasks. This may be feasible for projects with few outside constraints, loose deadlines and ample budget, however most construction projects are highly constrained. Complexity in scheduling arises from labor laws, live operating infrastructure which may need to be closed, noise-restrictions and less efficient work at night and so on. Just like with the planning of activities, the scheduling of a project plan in accordance with cost, time and other factors would benefit from quicker manual iteration or automated systems.

A core use-case for dProB has been offered by the railway construction industry in which alterations and maintenance of train paths have to be performed in tight time windows. Those time windows are often planned years in advance, before the contractors or utilized



(a) A screenshot of a scheduled project plan in Microsoft Project [2] with one activity per row, and with names, deadlines, start and end dates the left and a visualization of activities and dependencies on the right. Taken from [3].

(b) A collaborative planning board for a highway construction project, with rows grouped into production areas and activities as bars, color coded per subcontractor. Taken from [TKA17, Fig. 1(a)].



(c) Screenshot of a tunnel construction site in dProB. The project aims to visualize and compare three different methods to transport and install concrete slabs through mobile cranes. The top left shows the 3D viewport which depicts one scenario in which two cranes pick up, drive and install the slabs in parallel. On the lower left, the Timeline depicts all pick-up drive and deliver processes and optionally shows dependencies. The inspector in the lower right shows properties and configuration of selected processes. Finally, the remaining windows on the right depict calculated performance indicators, such as the progress over time or the consumed fuel at the current point in time.

Fig. 1.1.: Three depictions of classical and modern construction project planning tools.

equipment are known. Besides the negative impact on quality of service and branch as well as sector image, delays beyond the planned closures can increase project costs rapidly. In an informal interview, we have been told costs are in the order of hundreds of euros per minute a train could not use its designated track. Furthermore, undercutting given deadlines provides an opportunity for contractors to earn bonuses, as the infrastructure is freed sooner than estimated.

Within this context, this work aims to contribute to questions like "When does a rail track closure provide sufficient benefit to be feasible?" and "Is it cheaper to only work on noisy activities during the day, or to work day and night but offer locals rooms in far-away hotels?". For this, we defined *work plans* as a means to model when activities may be processed, based on their properties like loudness and location. Upon this, we formulated a problem statement for the *work schedule generation problem* (WSGP), which asks which recombination of user-given work plans results in minimal cost for a given project. The result is a work plan, which dictates when costs for environmental factors like permits, closures, or accommodations for locals during overnight work must be paid in order to achieve minimal total cost.

To solve the WSGP, we devised a two tiered system: The upper tier consists of a meta heuristic, specifically a genetic algorithm, which evolves work plans by combining prototypical day or week plans that have been annotated with costs. The fitness of each work plan is calculated through summation of costs after executing the lower tier: An extended ASAP scheduling algorithm, which takes work plan restrictions and activity specific allowed work intervals into account. By this approach we built a proof-of-concept for an optimization method for the given problem that allows the user to decide how much time they want to spend on calculations, and that can easily be extended in future work.

Chapter 2 discusses the broad field of project scheduling and optimization, defining common terms and presenting related work. Building on this, Chapter 3 formalizes the WSGP, describes the proposed method to solve it and discusses implementation details. To evaluate the method and discover the stated problem, Chapter 4 presents the design of four subsequently refined datasets on which we evaluated our approach. It also states learnings about the implementation and the problem in general. Lastly, Chapter 5 recaps this work, highlighting its limitations and giving pointers for future work in this field.

2. Background

To provide all context needed for our approach, the associated reasoning and evaluation in the following chapters, this chapter firstly discusses the mathematical foundations of optimization and project scheduling, followed by an overview of current related works.

2.1. Mathematical Foundations

2.1.1. (Project) Scheduling

In general, *scheduling* could be seen as the task of assigning temporal information to a given set of actions or events to achieve a certain goal. In *Scheduling - Theory, Algorithms and Systems*, however, Pinedo also calls scheduling "a decision-making process that [...] deals with the allocation of resources to tasks over given time periods [...]" [Pin16]. This simple comparison of definitions highlights the importance of perspective in formulating a given scheduling problem depending on its real world use-case. In the end all actions or events, their relative priorities, resource limitations, due dates, throughput of actions, and other variables have to be considered. Yet, depending on the elements of interest, it may prove beneficial to model the problem one way or another.

Machine and Job Scheduling

A framework to describe the most common families of scheduling problems, which models resources as machines and actions or events as jobs, has been presented in Section 2.1 of *Scheduling* [Pin16] with according notation $\alpha \mid \beta \mid \gamma$: α describes the machine environment, including their numbers, interdependencies, and capabilities, β describes scheduling constraints and specifies properties of jobs, and finally γ describes a goal.

A classic scheduling problem in the literature is the *job shop problem* $Jm \mid\mid C_{\max}$, wherein every job has to be processed on a predetermined subset of m machines with a given order ($\alpha = Jm$), aiming to minimize makespan, i.e. the latest finishing time of all jobs ($\gamma = C_{\max}$). A solution to this problem could be used in a woodworking workshop to manufacture a certain set of items, which require individual processing at their lathe, saw, planer, sander, glue-up station, and paint booth, as fast as possible.

Another exemplary problem could be $Pm \mid r_j, M_j \mid \sum w_j T_j$: At a 3D print farm, there are m 3D printers, which can operate in parallel and independently of one another and which have essentially the same capabilities ($\alpha = Pm$). Despite this, every job j can only be processed on a subset of machines M_j, as they provide varying print volumes. Further, the files required for printing will arrive only at a known release time r_j ($\beta = r_j, M_j$). With

these parameters the goal is to minimize the sum of deadline overruns over all jobs weighted by w_j, e.g. the contractual penalty of a given job ($\gamma = \sum w_j T_j$).

Finally, Pinedo propose $P\infty \mid prec \mid C_{\max}$ to describe a basic project, i.e. infinite machines working in parallel ($\alpha = P\infty$) on jobs with precedence constraints ($\beta = prec$) with the goal to minimize makespan ($\gamma = C_{\max}$). While perfectly usable in its basic form, the machine-job-centric approach has not been adopted in the world of project planning and scheduling. We shall assume that this lies in the fact, that projects often involve a higher variety of resources like skills, workers, equipment, and actions or events, which could theoretically be represented as machines and jobs, but lend themselves to other representations.

Project Scheduling Problem

The Project Scheduling Problem (PSP) describes the basis of a variety of problems concerned with the assignment of start and end times of related and interacting activities. The statements following in this subsection are based on a survey by Hartmann and Briskorn [HB22].

Definition 2.1 (Project Scheduling Problem)**.** Given a set of n activities labeled $V := 1, \ldots, n$ having a processing time p_i and a set of immediate predecessors P_i each, find a start time $s_i > 0$ and an end time $e_i = s_i + p_i$ for each activity i, such that the makespan, i.e. the time to the latest end $\max_{i \in V}\{e_i\}$, is minimized, while every activity starts only if all its predecessors have ended, i.e. $\forall i \in V, j \in P_i : s_i \geq e_j$.

One possible formalization of the PSP is given in Definition 2.1. Notably, all activities can be executed in parallel if they do not share any predecessors. In practice, this is only the case, when a project's activities are planned on a high level of detail and when predecessors are defined accordingly. To provide real-world value, many scheduling problems are extended by additional constraints.

The predecessor relationships defined in Definition 2.1 are an instance of so-called finish-start precedence constraints, which are the most common kind of precedence constraints. Other kinds include start-start, finish-finish and start-finish constraints. They are readily available in project management software, such as Microsoft Project. Often, they also feature lag times, which define an offset that shifts the time constraint forward or backward, for example saying that some activity can start one day after another activity ended.

Resource Constrained Project Scheduling

A very active area of research and common extension of the PSP is the Resource Constrained Project Scheduling Problem (RCPSP), which allows each activity to be assigned a set of resources, which are then assigned or consumed during the activity's execution. As seen in a recent survey of RCPSP literature [HB22], there are many kinds of resources, which can be distinguished by their availability, e.g. being (partially) renewable or being available on a calendar schedule, or by having multiple skills, e.g. workers or tools that can stand in for more than one abstract resource requirement.

Building upon availability of resources, activities might also be extended by the possibility of preemption, i.e. the possibility to interrupt an activity and resume it later. This entails

whether preemption causes costs, either in time (setup times) or money. To provide higher flexibility and the possibility to avoid preemption, activities can also be modeled as multimode activities, i.e. activities which can be processed in a variety of modes, each with different resource requirements and processing duration.

With this great variety of problem models, two major goals can be found in the literature [HB22; GZ22]: Minimization of resource usage, like cost, time (makespan) or emissions, and minimization of risk, meaning possible change of resource usage in face of adversity. More recent works, however, often address more intricate optimization targets:

- The Resource Investment / Resource Availability Cost Problem (RIP/RACP) tries to minimize the fixed cost of resources, under the constraint of a project deadline. Resources, when bought, are available for the entire project. This can be used to decide on the minimal number of equipment to buy to get the project done in time.
- The Resource Renting Problem (RRP) extends the RACP by acknowledging, that some resources may have a time-dependent cost component, i.e. some per minute cost can be assigned resource usages. Therefore, this adds the possibility to model the renting of additional equipment during critical bottlenecks of a project.
- The Resource Leveling Problem (RLP) aims to minimize fluctuations in resource usage, e.g. to minimize fire-and-hire, to avoid spikes in spending, or to avoid bottlenecks in logistics.
- Afshar-Nadjafi worked on a variation of the RCPSP, where total project cost was minimized by considering earliness and tardiness penalties, i.e. an abstract cost factor subsuming contractual deviations, extra storage requirements, idle times, and damages to reputation among others [Afs14].

It is standing to reason that more complex objectives are still in large parts proxies or heuristics for resource cost or risk, required due to insufficient detail in models or computational complexity of a detailed formulation. However, less tangible goals like ethics, reputation and sustainability that inform optimization goals, like reducing needless hiring and firing, do make appearances.

2.1.2. Optimization

Informally, optimization is the problem of finding a set of input values for which a given goal is solved "the best". The following formal description is a brief summary of [Eng07, Appendix A]: A (single goal) optimization problem consists of an object function f, a search space or domain $\mathcal{S}$ which contains all possible assignments for all variables of the given problem, and a set of constraints that restrict which solutions $x \in \mathcal{S}$ are feasible, resulting in the set of feasible solutions $\mathcal{F} \subseteq \mathcal{S}$. Assuming a maximization problem, we aim to determine the set of all $x^* \in \mathcal{F}$ for which $f(x^*) \geq f(x)$ for all other $x \in \mathcal{F}$, i.e. $\arg\max_{x \in \mathcal{F}} f(x)$. The minimization problem is defined analogously. The definitions and classifications below are built upon [Tal09] unless noted otherwise.

Characteristics of Optimization Problems

The three components of optimization problems, together with the required accuracy and certainty, can contribute to the difficulty of problem instances. In the following paragraphs, we will look at some of these factors and comment on their impact. Note that "difficulty" in the paragraphs below is not intended to imply any formal complexity but is intended to give a rough intuition. As a visual metaphor we will keep a topological map in mind, whose altitudes are derived from the goal function, akin to [Ban12, Chapter 1].

The *goal function* specifies a landscape on which we aim to find the highest or lowest point. The more we know about this function and "the better it behaves", the easier it will be to reach our goal. Such "good behavior" can be seen in functions that are continuous, differentiable, convex, and have homogenous rates of change as well as low interactions between variables, among others. Visually this would translate to the difference between a map with broad landscape features, moderate slopes and a single mountain versus rough, mountainous terrain with many cliffs and peaks. A simple mathematical example would be a goal function $f : \mathbf{x} \mapsto \sum x_i c_i$ with coefficients $\mathbf{c}$. With this knowledge it is comparatively easy to find the solution by moving into direction $\mathbf{c}$ as far as possible, i.e. maximizing each $x_i c_i$ independently. A nonlinear black box goal function, on the other hand, does not allow for any global reasoning, instead requiring informed trial and error. Formal descriptions for the complexity or difficulty of goal function landscapes can be found in [Tal09, Chapter 2.2]. Finally, it shall be noted that dynamic, self-referential and non-deterministic goal functions, i.e. functions that change over time or may produce a range of results with the same inputs independently of time, exist. These kinds of problems are beyond the scope of this thesis' background.

The *domain* has two modes of influencing the difficulty of a given problem: the number of its dimensions and their kind. The number of its dimensions, i.e. the number of variables, trivially increases the problem complexity, if the variables are used in the goal function in a meaningful way. The kind of dimension, however, has a non-trivial impact: Even though the set of all integers $\mathbb{Z}$ is a real subset of the real numbers $\mathbb{R}$, whereby a domain $\mathbb{Z}^n$ could be deemed less complex than a domain of $\mathbb{R}^n$, discrete dimensions implicitly force the goal function to have discontinuities. As discussed above, discontinuities may increase problem difficulty. Visually speaking, it is easier to find the highest peak on a single map of a continuous stretch of land on an A3 sheet of paper, than on eight smaller stretches of land that may or may not overlap on eight pieces of A6 paper.

The same reasoning can be applied on *constraints*, as the domain of a problem could be interpreted as the set of intermediately feasible solutions in the domain of all possible values, i.e. a set of constraints. Two kinds of *constraints* are considered on a high level: (1) Equality constraints h, requiring a function of the variables to result in a certain value, e.g. $h(\mathbf{x}) = 0$, and (2) inequality constraints, requiring a function of the variables to be greater/less than or equal to a certain value, e.g. $g(\mathbf{x}) \leq 0$. Boundary constraints are often considered a third kind of constraint, but are usually a special case of inequality constraints, limiting the search space such that the domain closer in size to the set of practically useful, feasible solutions. The most common boundary constraints are box constraints which put an upper and lower bound on each dimension of the domain individually. Notably, the functions involved in

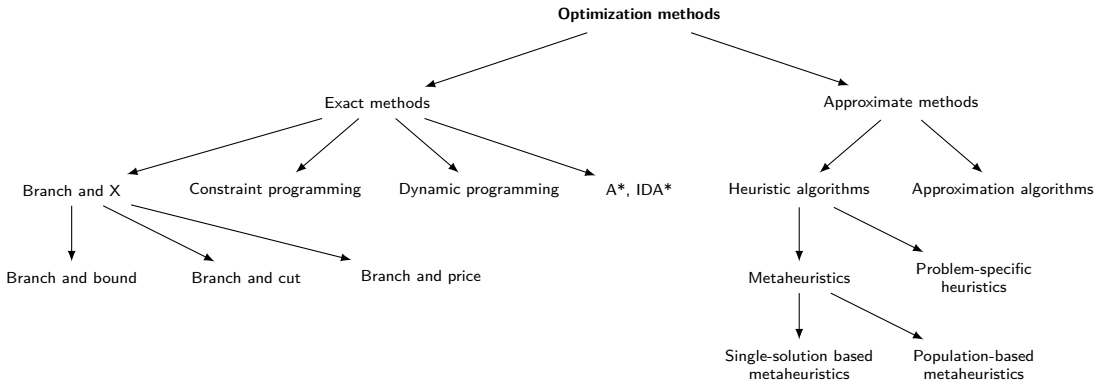


Fig. 2.1.: Hierarchy of classical optimization methods, taken from [Tal09, Figure 1.7]

constraints induce similar complexities as a goal function if they are not "well-behaved". On a map, boundary constraints could be represented by the map's borders. Assuming there exist markings of trees, we could also formulate a complex equality constraint, that solutions are only feasible if they have exactly zero tree markers within 50 meters adjusted for scale.

As seen above, there are many ways in which an optimization landscape can make it difficult to find a solution. While there are *exact and ϵ-approximation optimization methods*, which yield a provably optimal solution or a solution which is at max ϵ times worse than the optimum, they usually lean heavily on certain assumptions about the structure and behavior of goal function, domain or constraints. For an exact approach, brute force enumeration is always applicable but usually practically unhelpful. If possible, it is useful to prune or bound the search space as much as possible: suppose a map is partitioned into grid squares and a heuristic is known that can estimate the highest point in a square while always estimating higher values than really exist. We can therefore safely discard any square for our search, that has a lower estimated maximum height than any point we visited.

For black boxes, highly complex problems, or very large instances, it may be desirable to consider *heuristic search methods*. They do not guarantee that a global optimum is found, possibly providing a local optimum, but they can be much more performant, and can often be generalized to larger problem classes, while being "good enough". Heuristic methods come in many flavors with approaches of both local search, i.e. iteratively exploring neighborhoods of search points, and global search, i.e. trying to find feasible and exclude infeasible subdomains by aggregating knowledge about the domain. To find the highest point on a map, an example could be to choose one or more points at random. Each of them is then repeatedly replaced by the best point in their neighborhood, e.g. by moving a fixed distance up the local slope, until they end up at peaks. The highest of those peaks is then the best local and possibly the global maximum. An overview over the hierarchy of optimization methods can be seen in Figure 2.1.

Metaheuristics

While there are certainly problem-tailored heuristic methods, *metaheuristics* are a family of optimization methods, which aim to assume comparatively little knowledge about a problem's goal function and its domain (search space). Instead, they aim to present general frameworks to traverse the problem domain efficiently towards promising solutions. To achieve this, designers of metaheuristics have to decide where on the spectrum of exploration versus exploitation their method should be positioned. This means they have to decide if the algorithm samples the search space to try and "learn" more about less visited sub-spaces (explore) or if sampling aims to derive an optimum from knowledge it has already obtained. Notably, this decision does not have to be static over the algorithm's runtime and often moves from exploration to exploitation over time, as seen with the methods below.

Before presenting a select few metaheuristic methods, we visit a set of classification criteria, as defined in *Metaheuristics: From Design to Implementation* [Tal09]:

- We differentiate between *iterative and greedy*, also known as constructive metaheuristics. While iterative algorithms modify their proposed solution(s) repeatedly based on evaluation of the goal function, greedy algorithms start with a blank slate and add to their solutions but never modify or remove previous results. Thereby, we note, greedy algorithms lean heavily towards exploitation.

- The number of solutions, notably *single-solution* based algorithms and *population-based* or many-solution algorithms, also provide a classification criterion. Notably, population-based search algorithms naturally lean towards exploration unless the population degenerates into a single-solution.

- The use of *memory* or lack thereof is an important practical criterion. Extracting and aggregating information throughout the runtime may provide the possibility to exploit already visited solutions more efficiently, however memory-less methods may be less complex, thus easier to reason about and also less demanding in hardware requirements.

- Next to last, *deterministic and stochastic methods* are juxtaposed. In contrast to deterministic methods, stochastic methods may provide different results on repeated execution. Talbi points out that performance evaluation of stochastic methods must take this into account.

- Lastly, Talbi lists *"Nature inspired versus nonnature inspired"* as a classification criterion. While it is interesting to see how many methods are inspired by physical, chemical, biological or social processes, and may be beneficial for further development of the methods, we assume this criterion to be less consequential than others for the application of said methods.

Important for metaheuristics, but not listed as a classification criterion is the assumed *representation* or encoding of the problem domain. Usually problems can be modeled and represented in a multitude of ways. Depending on it, the same method may traverse the

$[0, 5, 5, 6]$

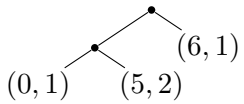
(a) An array of 4 3-bit integers.

$[0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0]$

(b) An array of $4 \cdot 3 = 16$ boolean values.

$[1, 0, 0, 0, 0, 2, 1, 0]$

(c) An array of integers, each representing the number of times its integer valued index occurs in sorted list of values.



(d) A tree, with leafs sorted by the first tuple element. The second tuple elements represent the number of occurrences of the first tuple element.

Fig. 2.2.: Various representations of a sorted list of 4 of 3-bit unsigned integers.

problem domain more or less efficiently. This is especially true for methods that mutate or recombine solutions into new solutions for future iterations, as we will see below. Example representations for a sorted list of 3-bit unsigned integers can be seen in Figure 2.2. While Figures 2.2a and 2.2b both state each integer one after the other but differ in the representation of their atoms, Figure 2.2c chooses to ignore the "identity" of individual integers, instead opting to count how often each integer value occurs. Similarly, Figure 2.2d counts occurrences, but representing them in a nonlinear fashion as a tree. Notably, the latter two variants have to be decoded to retrieve the encoded list, while the former representations are just different views on the normal integer representation in hardware.

In the excerpt of metaheuristic methods below, we will focus on iterative methods, as greedy methods require more knowledge about the problem model for their design. We will, however, feature methods that vary in solution count, use of memory, and determinism.

Tabu Search (TS) is a deterministic single solution optimization method with memory [Tal09, Chapter 2.5]. Starting from the initial solution, each iteration evaluates all neighbors of the current solution to select the best feasible solution among them for the next iteration. Notably, this may include solutions that are worse than the current solution to escape local optima. As this can lead to cycles, where the algorithm switches back and forth between two solutions, a tabu list is introduced. This short term memory is filled with visited solutions, or moves and solutions with certain properties, which will be declared infeasible upon evaluation, such that the algorithm cannot visit or execute them. For better exploitation and higher diversification, extensions with longer term memories exist. While this method has been applied to many optimization problems, it is said to be "very sensitive to [...] the size of the tabu list". Lastly, definition and enumeration of a solution's neighborhood and determination of taboo properties can be highly problem-specific.

Simulated Annealing (SA) is a stochastic single solution optimization method without memory, inspired by controlled heating and cooling of materials in metal and glass processing among others [Tal09, Chapter 2.4]. Starting with an initial solution and a starting temperature, each iteration random neighbors of the current solution are generated. The generated solution replaces the current solution if it is better but also with probability $\exp\left(\frac{-\Delta E}{T}\right)$, where ΔE is a measure of how much worse the generated solution is and T is the current temperature. If after generation and replacement an equilibrium condition is satisfied, e.g.

after a fixed number of iterations, the temperature is updated according to the temperature schedule, e.g. $T = T \cdot 0.5$. SA is a widely used search method which "gives good results for a wide spectrum of optimization problems".

Evolutionary Algorithms (EA) are a family of population based optimization methods, all of which build on a simple loop inspired by Darwinian evolution [Tal09, Chapter 3.2]: (1) Selection: from a given population with known fitness, i.e. with known evaluations of the goal function, individuals, i.e. solutions, are selected with a probability depending on their fitness. (2) Reproduction: The selected individuals are then used to generate new offspring, e.g. through mutation, crossover, sexual or asexual reproduction depending on the concrete method. (3) Replacement: After evaluating their fitness, a new generation of individuals is assembled by replacing some or all individuals from the current population by newly generated individuals, again depending on their fitness. This new generation is the population for the next iteration.

Genetic Algorithms (GA) are a stochastic population based optimization method and are a very popular class of evolutionary algorithms [Tal09, Chapter 3.2.1 and 3.3]. Classically, they utilize a selection rule which increases the probability of selection proportional to the individuals' fitness, a reproduction rule with two parents both with crossover of the parents' genes and probabilistic mutation of the resulting gene, and generational replacement, where all parents are discarded. Furthermore, GAs historically operated on linearly encoded binary representation of models, where mutations are a bit-flips and crossover refers to the exchange of gene segments. There are however many extensions of GAs which modify selection, reproduction or replacement and allow for other representations, blurring the lines to other methods. One other selection method, for example, is "Rank-Based Selection" where all individuals are sorted by their fitness and assigned an increasing rank. Based on this rank and a configurable selection pressure $1 < s \leq 2$ individuals with higher ranks are selected more often. Another extension, which can be wrapped around any replacement or selection method, is *elitism*, where a certain number of previously best individuals of both parent and child generations are added to the next population or can be considered for reproduction, no matter the internal mechanism.

As an example for blurred lines between EAs, *Differential Evolution* (DE) is a stochastic population based optimization method without memory, for problems whose solutions can be represented as vectors of real numbers [Tal09, Chapter 3.4.2]. The distinguishing feature of DE is its unique recombination operator: Given a parent x, a random target t, random mutants m^1 and m^2, each selected from the population, and a scaling factor F, such that $x \neq t \neq m^1 \neq m^2$, each vector component x_k of the parent is replaced by $x_k = t_k + F \cdot (m_k^1 - m_k^2)$ probabilistically with a fixed crossover rate CR. The resulting vector then replaces its parent if it scores better upon evaluation. While there are still ways to customize DE strategy, e.g. by defining how parents, targets and mutants are selected, it has relatively few parameters (F and CR), while we can still observe an integral feature of many EAs: If the pool of chromosomes converges towards an optimum, the algorithm automatically moves from exploration to exploitation over time. However, where GAs would have to specify a dynamic mutation rate to lean into this, DE is self-adapting by relying on the distance of individuals. Notably, this can also result in an increase in exploration, when a subset of the population discovers a new optimum after exploitation of the old one.

Many more metaheuristic methods, including the combination of various algorithms in hybrid metaheuristics and metaheuristics designed for parallel and distributed computing systems, can be found in *Metaheuristics: From Design to Implementation*, Chapters 4 and 5 respectively [Tal09]. More evolution and biology inspired optimization methods, definitions for many evolutionary computation operators and discussions of extensions for genetic algorithms can be found in *Computational Intelligence: An Introduction* [Eng07]. A recent overview of operators used in genetic algorithms can be found in [KCK20]. While no comprehensive quantitative evaluation of operators for a variety of general problems could be found, a listing of qualitative pros and cons can be found in [KCK20; MT24].

Mathematical Programming

The field of mathematical programming aims to model optimization problems in mathematically clear and formal language, such that they can be classified and then solved with off the shelf solvers, for example with branch and bound methods. This depends on deep knowledge about the structure of the problem at hand, as it would be impossible to formulate it otherwise. A very well studied mathematical programming model is that of *Linear Programs* (LPs), where the domain is some real value vector space and both goal functions and constraint functions are linear maps. The mathematical notation can be seen in Equation (2.1). It is known, that LPs can generally be solved efficiently, meaning commercial solvers are able to find optima in problems with millions to tens of millions of variables within minutes to a few hours [4].

$$\begin{aligned} \text{minimize } & f(x) = c_1 x_1 + \cdots + c_n x_n \\ & a_{11}x_1 + \ldots + a_{1n}x_n \leq b_1, \\ \text{subject to } & \quad \vdots \qquad\qquad \vdots \qquad \vdots \\ & a_{m1}x_1 + \ldots + a_{mn}x_n \leq b_m, \\ \text{with all } & x_i, c_i, a_{ij} \in \mathbb{R}, x_i \geq 0 \end{aligned} \tag{2.1}$$

Mixed Integer (Linear) Programs (MI(L)Ps) are very similar to LPs, except that the domain may consist of real or integer valued dimensions. As discussed above such discretization can make problems much harder: Even a problem instance with just 60 binary variables cannot be solved within two hours by many modern solvers [5; Gle+21, markshare2]. The same holds for extensions like *Constraint Integer Programs* (CIPs) [Ach07], which generalize the constraints to any function that returns some boolean value. This does, of course, not mean that MIPs and CIPs cannot be used for practical problems, we will, however, observe in Section 2.2 and Appendix A that their formulation has to be considered well for them to be solved in feasible time.

2.2. Related Works

In a 2022 survey, Hartmann and Briskorn presented an overview over many works on extensions of the RCPSP, classifying them by their focus [HB22]. Many extensions have been mentioned in Section 2.1.1, however, the most important ones for this work are those on scheduling of interruptible/preemptible activities, time dependent availability of resources, and those focusing on the RACP/RRP. For future work, those on setup times would also provide a starting point. Below, we will look into three works from this survey. An overview over their and a related work's formal problem and approach can be seen in Table 2.1.

In "Mixed-integer linear programming and constraint programming formulations for solving resource availability cost problems", Kreter et al. addressed the *RACP/max-cal*, a version of the RACP, which features only start-start precedences with minimum and maximum lag, for which each resource has its own availability-calendar [Kre+18]. As seen in Section 3.1, we built on this by proposing a method to define such calendars for environmental factors through "work plans" and "markers". Their approach featured mixed integer linear programs, as well as constraint programming methods using the chuffed solver [6], the latter of which were able to solve problem instances of 200 activities within below one minute on average in case of zero float[1], and within less than three minutes with up to 50% extra project time. Since their approach operates on the basis of a discretization of time it is important to note, that the prescribed maximum project duration in the data appears to be around 5500 time units. We therefore note, that this scheme may not be feasible when activities with durations of seconds are combined with more abstract activities with durations of hours.

Unlike Kreter et al., Polo-Mejía et al. utilized a mixed discrete and continuous time approach in their MILP approach for scheduling of activities in a nuclear research facility [Pol+20]. Formally, they solved the Multi-Skill PSP with partial preemption, i.e. a PSP variant in which some resources, e.g. technicians, have multiple skills, and activities can require multiple skills to be processed. Further, activities can be partially preemptive, meaning some required resources can be released to pause processing, while others cannot be released without starting from zero. This could be used for experiments, where equipment will still be occupied, while technicians can switch to more important tasks. Similarly to the previous MIP approach, their methods worked well enough for their needs, at least after providing the MIP with at starting point through a greedy heuristic ("warm start"), but the size of problem instances may become a problem when utilizing their methods in other contexts: Both their MILP and their CP approach computed in the order of minutes for instances with 30 activities. Even with a warm start, the MILP approach could solve less than half ($\frac{80}{200}$) and the CP approach could solve above three quarters ($\frac{160}{200}$) of the testing instances to optimality within 10 minutes.

Another approach could be taken from "Resource Constrained Project Scheduling Subject to Due Dates: Preemption Permitted with Penalty", where Afshar-Nadjafi addressed the RCPSP with cost optimization extended by penalties for preemption as well as earliness/tardiness [Afs14]. Their approach works as follows: A modified simulated annealing

[1] "Float" or "slack" being the duration an activity can be pushed forward or backward in time without impacting any other scheduled activity.

Tab. 2.1.: Overview of relevant characteristics of reviewed non-survey and background works.

Work	Problem	Viewpoint	Schedule/Time	Solution Approach
[Kre+18]	RACP/max-cal	Scheduling	discrete	MILP and CP
[Pol+20]	MSPSP-PP	Scheduling	discrete	MILP, CP and greedy Alg.
[Afs14]	PRCPSP-WETP	Scheduling	permutation	SA with SSGS
[CT15]	TCUT	Work Plans	permutation	OMODE with SSGS

process, in which the neighborhood traversal is informed by tardy activities, generates an ordering of all activities. The generated order is then used by a serial schedule generation scheme (SSGS), in which the activities are temporalized as early as possible in ascending order. In their benchmarks, they solved problem instances with 90 activities in less than 5 seconds on average. With smaller instances of 30 activities, they compared their results with a MILP based approach and achieved an average deviation of less than 1% in cost.

Moving on to the field of work schedule generation, there do not appear to be many works on generating a timetable or work plan that describes when certain categories of work could happen instead of focusing on scheduling activities. With "Opposition-based Multiple Objective Differential Evolution (OMODE) for optimizing work shift schedules", Cheng and Tran worked on one instance with said focus [CT15]. They are trying to solve the time-cost-utilization work shift tradeoff (TCUT) problem, an extended time-cost trade off (TCT) problem, which in its base form acknowledges that time and cost are highly related in some domains like construction projects and aims to minimize both time and cost. In their extension, they add a third goal which aims to minimize evening and night shifts. Similar to [Afs14] above, a SSGS is used for temporalization. They tested their method with two test cases of 60 activities, though no calculation time is given.

3. Approach

This chapter formally defines the problem this thesis aims to contribute to. Furthermore, it discusses the implementation of our proposed approach. For this, Section 3.1 defines a variation of the project scheduling problem, called the project scheduling problem with marked activities (PSP/mark), to then define our overarching problem, the work schedule generation problem (WSGP), together with various potential cost components. Section 3.2 presents a solution for the PSP/mark and two solution-encodings for the WSGP for use in evolutionary algorithms. Further, it contains the required implementation details.

3.1. Problem Statement

On a non-mathematical level this thesis aims to make a first step towards generating cost optimized work plans, based on an input of activities and precedences which are annotated with information on resource assignments, environmental needs and other general properties, and a series of prototypical work plan templates which encode information about when activities are allowed to be worked on, based on their properties. Further, work plans are annotated with information about how much these opportunities to work at said times costs.

For this first step, we operated under the assumption that it is generally beneficial to complete projects as fast as possible. As this is not always true, we formulated the problem in two parts: on a high level the problem asking for optimization of work plan cost is defined in Section 3.1.2. On a lower level, however, it requires a scheduling scheme to determine project cost. Based on the assumption above, we are using and solving a customized variant of the PSP defined in Section 3.1.1. This scheme can be replaced in the future.

3.1.1. Modified Project Scheduling Problem

To begin with, we formally define activities in Definition 3.1. Note, that this serves as an abstract representation of low-level tasks, high-level groupings of sets of tasks, and any kind of processes that have distinct start and end times. This way, information about both rough and detailed estimations and simulation results can be incorporated.

Definition 3.1 (Activities and Markers)**.** An activity a is some process which has a fixed processing duration p_a. In case processing can be paused at will, we call an activity interruptible with $i_a = 1$, otherwise we say it is non-interruptible with $i_a = 0$. To record scheduling relevant properties, activities have an associated set of markers m_a which can contain arbitrary objects, e.g. text. After scheduling, we refer to the time processing is started, i.e. the start time, as s_a and the time processing is finished, i.e. the end time, as e_a. For a schedule of non-interruptible activities to be admissible $e_a = s_a + p_a$ has to hold, otherwise $e_a \geq s_a + p_a$ has to hold.

In comparison to other PSP extensions, the inclusion of markers instead of resource requirements matches the expectation that resource assignment has already been performed, while still exposing information like occupied location, utilized equipment or activity inherent properties like noisiness. Assuming one time unit equals one hour, we could, for example, model the process of making a wall opening as some a_1 with $p_{a_1} = 2,\ i_{a_1} = 1$ and $m_{a_1} = \{\text{loud}, \text{in basement}, \text{uses drill}\}$, while the drying of a wall could be modeled as a_2 with $p_{a_2} = 6,\ i_{a_2} = 0$ and $m_{a_0} = \{\text{quiet}, \text{in basement}\}$.

Building upon this, we define precedences, i.e. dependencies between activities, in Definition 3.2. Notably this definition only contains a minimum lag, but not a maximum lag, which would seem a natural next step. This has originally been considered by use of a reversed precedence with negative minimum lag akin to [Kre+18], but has been removed due to increase in problem complexity and cyclic dependencies. Further details are discussed below and in Appendix A.

Definition 3.2 (Precedence Constraints)**.** A precedence constraint $\pi = (a_1, a_2)$ between two activities a_1, a_2 with minimum lag l_π describes a dependency of the start or end time of a_2 on the start or end time of a_1. Depending on the precedence type $t_\pi \in \{FS, FF, SF, SS\}$ this constraint implies one of the following inequalities:

$$\text{Finish-Start (FS): } s_{a_2} \geq e_{a_1} + l_\pi \qquad \text{Start-Start (SS): } s_{a_2} \geq s_{a_1} + l_\pi$$
$$\text{Finish-Finish (FF): } e_{a_2} \geq e_{a_1} + l_\pi \qquad \text{Start-Finish (SF): } e_{a_2} \geq s_{a_1} + l_\pi$$

With these definitions we can now define the activity-on-node (AoN) graph in Definition 3.3 which describes the network of interdependencies between all activities of a given project. To ensure precedence constraints do not over-constrain the problems they are involved in, we assume AoN graphs to be directed, acyclic graphs. This assumption forbids two activities to depend on each other, possibly asking states like "a_1 happens before a_2 and a_2 happens before a_1". Throughout this thesis, we presume AoN graphs in inputs to be unscheduled, while the outputs of algorithms will be scheduled AoN graphs.

Definition 3.3 (Activity-on-Node Graph)**.** An activity-on-node (AoN) graph $G = (A, \Pi)$ describes the network of all activities $a \in A$ as nodes and all precedences $\pi \in \Pi$ as edges. It is assumed, that the G is a directed, acyclic graph. We call an AoN graph unscheduled if the start and end times of activities are undefined while fully defined AoN graphs are called scheduled.

To schedule an unscheduled AoN graph, we utilize a calendar similar to [Kre+18] but translated from resource requirements to our activity markers. To this end, we define work plans in Definition 3.4, which define when activities are allowed to be processed. A visual representation of two work plan prototypes can be seen in Figure 3.1.

On a technical note, we are using intervals which include their left bound but exclude their right bound. Formally, we can define a right-open interval as $[a; c[= \{b \mid a \leq b < c\}$. This way, every point in time corresponds to a maximum of one work plan intervals. Further, this also forbids activity starts in the last instant of an interval.

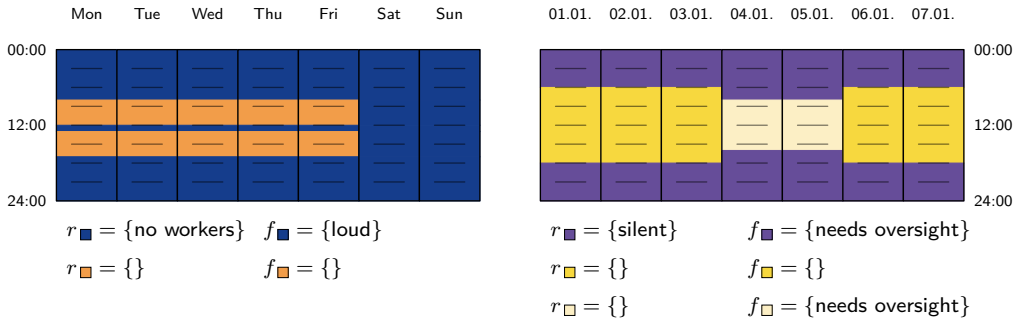


Fig. 3.1.: Two work plans for the duration of a week. The prototypical work plan on the left depicts an 8-hour work week with a 1-hour break at noon. The instanced work plan, starting on 01.01.2020, on the right depicts 12-hour work days during the week with additional 8-hour weekend shifts. The colors denote work plan intervals with the same markers required ($r_{\square}$) and forbidden ($f_{\square}$) in an activity for it to be processable in the intervals.

Definition 3.4 (Work Plans)**.** A work plan W is a partitioning of given right-open time interval into right-open intervals $w \in W$, for each of which a set r_w of required markers and a set f_w of forbidden markers is defined. For the processing of an activity a to be *allowed* during w, the following condition has to hold: $r_w \subseteq m_a \wedge m_a \cap f_w = \emptyset$. Otherwise, the processing of a is *forbidden* in w. For brevity, we say a is allowed or forbidden in w. We call W a *prototype* when it partitions an abstract time frame, like a day or a week. When it partitions a real world calendar interval, e.g. from midnight January 1st 2024 to midnight January 3rd 2024, it is called *instanced*.

Definition 3.5 (Schedule Feasibility)**.** A scheduled AoN graph is called *feasible* with respect to a work plan, when all activities are allowed to be processed for at least their processing duration, while activity feasibility constraints and precedence constraints are held, and when activities start in work plan intervals which allow them to be processed.

We can now formulate an extension of the PSP that fits our definitions of activities, precedences, AoN graphs and work plans. This project scheduling problem with marked activities (PSP/mark) is defined in Definition 3.6 and will be the basis on which the larger work plan generation is built. An overview over all parameters and variables of this problem is also presented in Table 3.1.

Definition 3.6 (PSP/mark)**.** Given an unscheduled AoN graph $G = (A, \Pi)$, and an instanced work plan W, the project scheduling problem with marked activities (PSP/mark) asks which feasible schedules of G w.r.t. W minimize the makespan, i.e. $\max_{a \in A} e_a$.

As one solution to the PSP/mark we propose the *As Soon As Possible* (ASAP) scheduling scheme as defined in Proposition 3.7. It is based on classical ASAP scheduling, e.g. known from processor scheduling and hardware synthesis [WC95], with an extension for work plans and is proposed as a starting solution for the PSP/mark to build on in this work, which would be easily replaceable by an *As Late As Possible* (ALAP) scheme, other critical path method related techniques or more complex scheduling algorithms later on [HHA24, Chapter 10].

Tab. 3.1.: Overview of parameters and variables in the PSP/mark scheduling problem.

activity	a	activity identifier	p_a	processing time of a
	m_a	marker set of a	i_a	interruptibility of a
	s_a	start time of a	e_a	end time of a
prec.	π	precedence identifier		
	l_π	minimum lag of π	t_π	precedence type of π
work plan	w	work plan interval		
	s_w	start of interval w	e_w	end of interval w
	r_w	required markers in w	f_w	forbidden markers in w
project	A	set of all activities	Π	set of all precedences
	W	set of work plan intervals		

Proposition 3.7 (ASAP Scheduling Scheme)**.** *We propose to solve the PSP/mark optimally by greedily scheduling all activities as early as possible, i.e. starting and ending their processing as early as possible.*

This approach has been chosen, as it yields unique and feasible solutions for the PSP/mark and can be computed efficiently, as shown below in Section 3.2 and Algorithm 1.

Lemma 3.8. *ASAP schedules are unique, minimize makespan and solve the PSP/mark.*

Proof. To show that ASAP schedules are unique and minimize makespan by induction, we will use that all $a_i \in A$ can be sorted topologically, because G is a directed acyclic graph. The empty AoN Graph $G_0 = \{\emptyset, \emptyset\}$ is trivially unique and has minimal makespan 0. Assuming we have an ASAP scheduled AoN graph $G_i = \{A_i, \Pi_i\}$, we now want to add the next activity a_{i+1} with its precedences Δ_{i+1} of form $(a_j, a_{i+1}), a_j \in A_i$, from the topological order. As we are only appending nodes without outgoing edges, we are not influencing G_i. It thus remains to be shown that there is only one way to append a_{i+1} as early as possible, i.e. there is exactly one assignment for $s_{a_{i+1}}$ and $e_{a_{i+1}}$.

Constructively, this time can be determined for all activities through their precedence constraints, which pose a lower bound of the latest end time with lag $\beta_{i+1} = \max_{\pi \in \Delta_{i+1}}(e_{a_i} + l_\pi)$, where $\pi = (a_i, a_{i+1})$. Non-interruptible activities have to start at the first point in time after β such that there exists a work plan interval or consecutive series thereof which allow a_{i+1} and which are longer than or equal to $p_{a_{i+1}}$. The earliest end time $e_{a_{i+1}} = s_{a_{i+1}} + p_{a_{i+1}}$ can be derived for non-interruptible activities from the activity feasibility formula in Definition 3.1. Interruptible activities are less restrictive in terms of start time and have to start at the earliest time β that is in a work interval which allows it. Scheduling $e_{a_{i+1}}$ as early as possible also minimizes interrupted time, such that the processing time $p_{a_{i+1}}$ equals the time a_{i+1} intersects work plan intervals that allow it.

As all G_i have minimal makespan and $e_{a_{i+1}}$ is scheduled as early as possible, the makespan of G_{i+1} is also minimal at $\max(\text{makespan}(G_i), e_{a_{i+1}})$. Therefore, Lemma 3.8 holds. □

3.1.2. Work Schedule Generation Problem

In this section, we state definitions, building up to Definition 3.14, which formalizes a broad *work schedule generation problem* (WSGP). It asks which recombination of prototypical work schedules is cheapest for a given scheduling scheme. As a first step to approach the WSGP, this work focuses on instances with an ASAP scheduling scheme.

To recombine prototypical work plans, we define the *splicing* operation. When splicing two work plans W_1, W_2 together at time t, we take all work intervals $w_1 \in W_1$ which end before or at t and all $w_2 \in W_2$ which start after or at t. Should any interval $w_1' \in W_1$ or $w_2' \in W_2$ contain t it gets shortened accordingly. Should, for example, both work plan intervals span the same time interval $w_1' = w_2' = [0; 1[$ for some splicing at $t = 0.5$, they would be turned into $w_1'' = [0; 0.5[$ and $w_2'' = [0.5; 1[$. Any parameters, like required or forbidden markers, are identical to the parameters of the full-length intervals.

In general, the cost could be an arbitrary function. We, however, propose to additively compose it from multiple *cost components*, each with its own focus. There is one component most central to our original motivation for the WSGP: The cost of opportunity C_{op}, as defined in Definition 3.9. It models the cost of allocating certain environmental conditions, by assigning each work plan interval a cost. That cost is invoked each time the work plan interval is instanced during the scheduled project, i.e. not after the last or before the first activity. While costs of any intervals starting after the last activity ends do not have to be paid, costs of intervals occurring during the project have to be paid even if no work happens in them. If, for example, we have a work plan which models the cost to provide locals with hotels, such that loud activities can happen at night, that cost has to be paid whether any loud work happens or not.

Definition 3.9 (Cost of Opportunity)**.** For a given instanced work plan W and an AoN graph $G = (A, \Pi)$ feasible w.r.t. W, and a given cost of opportunity for each work plan interval c_w for each $w \in W$, the cost component of opportunity is the sum $C_{op} = \sum_{w \in W^*} c_w$, where $W^* \subseteq W$ is the set of work plan intervals which intersect the interval of the start of the first to the end of the last activity, i.e. $[\min_{a \in A} s_a; \max_{a \in A} e_a[$.

There are, however, other components which are relevant for the actual real work cost of a project: the cost of lateness C_{late}, which models a soft deadline for the project as a whole, penalizing through an hourly rate; the cost of splices in the work plan C_{splice}, which models the cost of switching to another prototypical work plan and discouraging frequent switches, e.g. stemming from inefficiencies from adjusting to the new plan or the exchange of signage; the cost of renting for discrete time intervals C_{rent}, modeling the fact that some resources represented by markers cannot be rented for arbitrarily short amounts of time, e.g. crane #1 costs 500 per day, midnight to midnight; and the hourly and shift dependent cost of labor C_{labor}, which models that activities may be more expensive to process during certain intervals, e.g. at night or on weekends. An example for all stated cost components can be seen in Figure 3.2. In the evaluation of this work, we will focus on C_{op}, C_{late} and C_{splice}.

Definition 3.10 (Cost of Lateness)**.** The cost of lateness penalizes completion after a soft deadline $\overline{d}$ with a rate of c_{late}, such that $C_{late} = c_{late} \cdot \max(0, \max_{a \in A} e_a - \overline{d})$.

Definition 3.11 (Cost of Splicing)**.** The cost of splicing models the fact, that each change from one prototypical work plan to another might incur some cost upon realization. Assuming k_{W_1,W_2} switches from work plan W_1 to W_2, with $W_1, W_2 \in \mathbb{W}$ and a cost c_{W_1,W_2} per switch the cost of splicing is $C_{splice} = \sum_{W_1,W_2 \in \mathbb{W}} k_{W_1,W_2} \cdot c_{W_1,W_2}$.

Definition 3.12 (Cost of Renting)**.** The rental cost component C_{rent} is based on division of time, where each marker, that describes a resource which can be rented, is associated with a grid of time intervals. This grid, specific to each marker m, is defined by a duration d_m and an offset o_m. For every interval in this grid, if any activity a utilizes marker m, i.e. where $m \in m_a$, for any duration within that interval, a corresponding rental cost c_m is incurred. The total rental cost C_{rent} is the sum of these individual costs across all intervals and markers.

Definition 3.13 (Cost of Labor)**.** Assuming a baseline cost c_a is given for each activity a, the cost of labor C_{labor} models varying labor costs over time by allowing each work plan interval w to specify a cost multiplier λ_w. Thereafter, $C_{labor} = \sum_{a \in A} c_a \cdot (\sum_{p_{a,w}} \lambda_w \cdot \frac{p_{a,w}}{p_a})$ is the sum of all activities' baseline costs multiplied by the weighted mean of all λ_w, proportional to $p_{a,w}$, the duration a is processed in w.

For use in the WSGP, we assume the cost annotation of work plan intervals happens for prototypical plans before splicing. As stated above, parameters specific to work plan intervals like opportunity costs are copied. Notably, this may overestimate costs, because both costs of two trimmed spliced intervals are paid in full. Future work could introduce merging algorithms, which modify parameters intelligently upon splicing, however, we chose to allow for no such mechanism, as the problem complexity and parameter count needed for evaluation is already very high.

Definition 3.14 (Work Schedule Generation Problem)**.** Given an unscheduled AoN graph G, a set of prototypical work plans $\mathbb{W}$, the earliest project start date s, the latest end date e and a cost evaluation function C, the optimized work schedule generation problem (WSGP) asks which splicing and temporalization of the given prototypical work plans into an instanced work plan W, that partitions $[s; e[$ results in the lowest cost C, for a PSP/mark optimal scheduled AoN graph.

Contemplating the WSGP with ASAP scheduling scheme and only cost of opportunity, it can be observed, that the problem is easier when activities are interruptible, as the length of work plan intervals does not influence the schedule by being too short for an allowed activity to fit. While the sub-problem of WSGPs with purely interruptible activities will not be studied further in this work, we expect the existence of a solution which is exact and performs well.

Assumption 1. *In accordance with our motivation from Chapter 1, we assume real world instances of the WSGP to feature a detailed project plan, whose activities' processing times are shorter than the average work interval duration. Concretely we expect work interval durations of multiple hours and processing times on the order of minutes to hours. Further, we expect that work plans will be defined through day plays with 10 to 20 intervals, which are then reused for multiple days of week.*

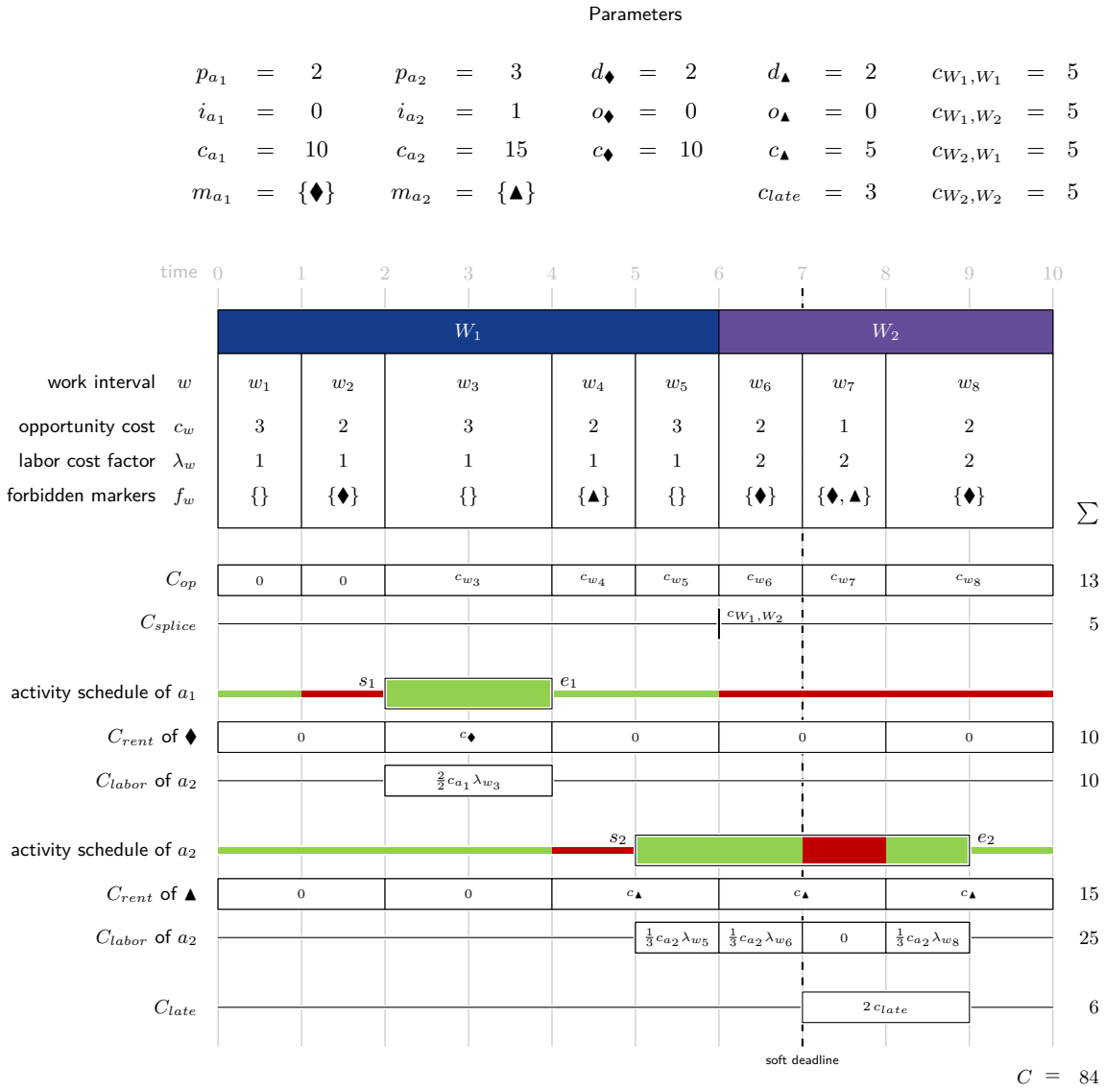


Fig. 3.2.: A visual representation of a spliced work plan annotated with the scheduled project's cost breakdown for each cost component. The project has two activities a_1 and a_2, which have a finish-start precedence (a_1, a_2) without lag. The intervals during which a_1 and a_2 are allowed and forbidden to be processed in the spliced work plan are shown for each in green and red respectively. Notably, opportunity costs of w_1 and w_2 are not incurred, as they lie before the first started activity. In comparison, the opportunity cost of w_4 is incurred even though no activity is processed during w_4, because activities are scheduled before and after it. Further, the opportunity cost of w_8 is incurred in full even though a_2 is only processed for half of it. The same holds for the renting cost of ▲ at the beginning and end of a_2.

3.2. Implementation

To solve the WSGP with ASAP scheduling and opportunity cost, as well as provide an extensible foundation for solutions of closely related problems, we designed a basic data structure for modeling data associated with partitions of time, we built an ASAP scheduling algorithm, we implemented an evolutionary computation (EC) framework, and we developed two encodings for the splicing of work plans within said framework. This section presents each of these implementations. They have been written in `C#`, this section will however discuss algorithms in pseudocode listings or text to focus on semantics.

3.2.1. Timelines

Both for the implementation of work plans and for one of the proposed genetic encodings, we required an efficient way to traverse a series of time intervals which have associated data. Specifically, we required a way to efficiently query the time interval containing a given timestamp, with the possibility to then iterate forwards or backwards over adjacent intervals. For this, we built a *Timeline* data structure by wrapping `SortedList`, a data structure which associates arbitrary values to a sorted and indexed set of keys. An illustration of both the data and the interval interpretation of a Timeline can be seen in Figure 3.3.

When constructing a Timeline, a global start time s, a global end time e, and data for the first implicit interval $[s; e[$ have to be set. Each key in the sorted list can be interpreted as the start of an interval while the next key represents its end. The global end time provides an upper bound for all intervals, particularly the last key. New intervals can be inserted by providing a start time and a data point. To retrieve the data associated with a query timestamp t, e.g. to query which work plan is active, the largest key smaller than t is retrieved by binary search, while iteration is implemented by traversing the key index. By usage of Timelines we are able to query intervals in $\mathcal{O}(\log n)$ and step to neighboring ones thereafter in $\mathcal{O}(1)$, where n is the number of keys or intervals.

3.2.2. ASAP scheduling

Upon this basis, we built a solution to the PSP/mark through an ASAP scheduling scheme, which operates akin to the inductive proof of Lemma 3.8: after sorting all nodes topologically, we iterate through them, calculating a lower bound for each from all its predecessors and determining the earliest start time afterwards. The pseudocode for this can be seen

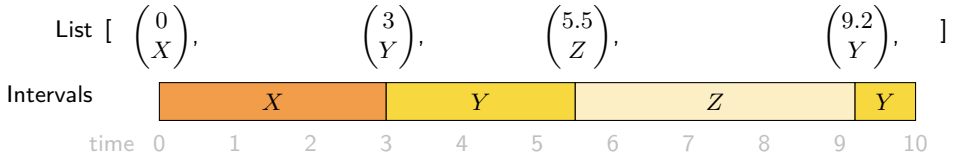


Fig. 3.3.: Example of a Timeline starting at $s = 0$ and ending at $e = 10$. The underlying data representation in form of a sorted list can be seen at the top. The bottom shows the implicit interval interpretation, with colors representing associated data.

in Algorithm 1. The subroutines are relatively similar to each other, iterating over intervals. As an example, the subroutine `EndFromStartAt` is also shown in Algorithm 2.

Regarding runtime, the main algorithm sorts the input AoN graph $G = (A, \Pi)$ topologically, which can be done in $\mathcal{O}(|A| + |\Pi|)$ for which we can assume the upper bound $\mathcal{O}(|A|^2)$ [7]. Afterwards, each node is visited once, iterating over all predecessors. For a directed acyclic graph, this results in a maximum of $\frac{|A|(|A|-1)}{2} \in \mathcal{O}(|A^2|)$ calls of the inner switch statement. The runtime of the switch statement lies in $\mathcal{O}(1)$ for FS, SS precedences. For FF, SF precedences the runtime lies in $\mathcal{O}(\log |T| + |T|) = \mathcal{O}(|T|)$, where $|T|$ is the number of elements in the Timeline, as the subroutines may have to iterate through the whole timeline. This assumes that the check whether a is allowed in w is done in $\mathcal{O}(1)$, e.g. through hash sets. The same complexity of holds for the later subroutines. This results in a worst case runtime complexity of $\mathcal{O}(|A|^2 \cdot |T|)$.

In accordance with Assumption 1, we expect real world work plan intervals which are longer than most activities, meaning activities will fit in at most two consecutive intervals. Further, we expect that most work plans will allow for any activity to be scheduled within a fixed low number of consecutive intervals, e.g. by being cyclical after 6 intervals. Thus lowering the complexity of the subroutines to $\mathcal{O}(\log |T|)$. Further, we assume real world AoN graphs to have a low number of predecessors for any activity. This lowers the complexity of sorting and switch calls to $\mathcal{O}(|A|)$. Therefore, we assume the average real world complexity of the ASAP scheduling algorithm to be in $O(|A| \cdot \log |T|)$.

Algorithm 1: ASAP scheduling of Activity-on-Node graphs

Input : Unscheduled AoN graph $G = (A, \Pi)$ with activities $a_i \in A$, start time s_{a_i}, end time e_{a_i}, and precedences $(a_i, a_j) \in \Pi$ with precedence type $t_{(a_i,a_j)}$and lag $lag_{(a_i,a_j)}$, Timeline T, earliest project start s

Output: ASAP scheduled G, solution to the PSP/mark

```
1  for a_i ∈ NodesInTopologicalOrder(G) do
2      bound ← s
3      for a_j ∈ Predecessor(a_i) do
4          switch t_(a_j,a_i) do
5              case FS do
6                  bound ← Max(bound, e_j + lag_(a_i,a_j))
7              case SS do
8                  bound ← Max(bound, s_j + lag_(a_i,a_j))
9              case FF do
10                 bound ← Max(bound, StartFromEndAt(T, a_i, e_j + lag_(a_i,a_j)))
11             case SF do
12                 bound ← Max(bound, StartFromEndAt(T, a_i, s_j + lag_(a_i,a_j)))
13     s_i ← EarliestAllowedAfter(T, a_i, bound)
14     e_i ← EndFromStartAt(T, a_i, s_i)
```

Algorithm 2: End time of activity from given start time (EndFromStartAt)

Input : Timeline T of work plan intervals $w \in T$ with start time s_w, end time e_w and required markers r_w and forbidden markers f_w, Activity to be scheduled a with processing time p_a, interruptibility i_a and markers m_a, lower bound $start$ for start time of a

Output: Earliest end time for the given parameters $time$

```
1  (rem, time) ← (p_a, start)    // remaining processing duration, time sweep
2  w ← IntervalAt(T, time)
3  while rem > 0 do
4      w' ← IntervalAfter(T, w)
5      if r_w ⊆ m_a and (m_a ∩ f_w) = ∅ then                    // a allowed in w
6          dur ← Min(rem, e_w − s_w)                          // used duration of w
7          time ← time + dur                                    // advance time
8          rem ← rem - dur          // advance processing time of allowed a
9      else if i_a = 0 then      // a is non-interruptible and forbidden in w
10         time ← e_w                                          // skip w entirely
11         rem ← p_a                          // reset processing time of a
12     else                          // a is interruptible and forbidden in w
13         time ← e_w                                          // skip w entirely
14     w ← w'
```

3.2.3. EC Framework

The literature on evolutionary computation and genetic algorithms provides a vast variety of optimization methods [Eng07; Tal09]. Usually, it lists few select applications of a specific method, or states broad qualitative findings [KCK20; MT24]. On the other hand, there are also a great variety of works, which go into specific applications for specific problems, discussing general trends and methods [LO07]. However, it appears that there are no cited works about how to choose the EC operators for a given problem and how to tune them. Therefore, we chose to use an EC framework, such that this and subsequent works can easily try out various operators and methods, to evaluate what works best.

To this end, the EC framework should, in general, provide maximal extensibility with minimal dependencies. The easy use and replacement of various evolutionary and genetic operators was a core requirement. First class control over the randomization provider for the evolutionary process, especially for reproducible testing and benchmarking, was also identified as an auxiliary requirement. A permissive licensing model, and the ability to interface with the *Dot.Net* ecosystem, especially with *netstandard 2.1* to be able to interface with the Unity game engine, were sought for, in hopes of one day integrating the framework into dProB. Finally, strong use of the type system to disallow the use of operators which do not work with a given genome, was a soft requirement to improve developer experience.

The *Genetic Sharp* library offers a wide variety of genetic operators and does allow for extension through user created operators and chromosomes [8]. It does, however, not provide any of our soft requirements and focuses only on genetic algorithms, whereby differential evolution operators could for example not be implemented in a modular *and* type-safe manner. *ParadisEO* is a mature framework which features many EC methods and operators in a modular way [Dre+21]. Beyond this, it also features an automatic algorithm design module, which allows for automatic configuration and evaluation of evolutionary algorithms [ADD21]. It is, however, written in `C++`, has not yet been packaged for use in Dot.Net ecosystem, and was discovered late in the progress of this thesis. Therefore, we implemented a library similar to Genetic Sharp with slightly broader scope.

Core to our framework is the general evolutionary algorithm seen in Algorithm 3. Within it, we utilize the following modularly replaceable strategies: (1) a *termination* strategy specifies when the evolutionary process terminates; (2) the *reproduction* strategy decides how parents are to be combined to generate children, additionally prescribing how many children to generate; (3) a *parent selection* strategy is used by the reproduction strategy and defines how to select parents from the population for reproduction; (4) the *mutation* strategy dictates how the children are probabilistically modified to explore the domain further; (5) a *repair* strategy can specify how to ensure invariants of the evolved objects can be restored should they not hold after reproduction and mutation; and (6) the *survivor selection* strategy defines which individuals of the last and current population make up the next generation.

Algorithm 3: The general evolutionary computation algorithm of our framework.

Input : Initial population initPop, Environment env,
Randomization provider rand, Goal Function `Eval`
Output: Final generation of the evolutionary process with fitnesses

```
1  population ← []
2  foreach genome in initPop do
3      fitness ← Eval(genome, env)
4      population ← population + (genome, fitness)
5  while not ShouldTerminate () do
6      newPopulation ← []
7      foreach genome in Reproduce(population, env, rand, SelectParents) do
8          mutated ← Mutate(reproduced, env, rand)
9          repaired ← Repair(mutated, env, rand)
10         fitness ← Eval(mutated, env)
11         newPopulation ← newPopulation + (genome, fitness)
12     population ← SelectSurvivors(population, newPopulation, env, rand)
13 return population
```

This differs from the minimal EC algorithm seen in [Eng07, p. 128] and [Tal09, p. 200] in that we split the reproduction step into reproduction, mutation and repair with the goal of building more generally applicable operators with less required configuration. Further, we renamed selection to parent selection and replacement to survivor selection, as both steps generally accept the same operators.

The implementations of strategies are generally intended to be as generic as possible, stateless or pure functions and potentially composable, such that users which aim to evolve more complex and composed objects do not have to redefine all operators for the composed parts. Parameters can be either saved in the operator or read from the shared environment of the evolutionary process. This environment also holds all data of the specific problem instance for use in the evaluation function. Finally, it could also be used for state management of stateful strategies like elite selection or self-adaptive processes. For control over randomization, strategies are given a randomization provider from which floating point numbers, integers and boolean values can be generated, e.g. with deterministic values for testing.

To allow for use of generic operators on objects with certain traits, the framework features some interfaces for genomes to implement. Examples are *Randomizable*, which indicates that the gene-domain can be uniformly sampled by use of a randomization provider, *Differentiable*, which indicates that a difference between two genomes can be calculated and later added onto a genome, or *ArrayGenome*, which allows for translation of the genome object into a linear series of genes, which in turn can be randomizable or differentiable again.

To provide insight into the strategies used later in this thesis, we will define them here. *TimeoutTermination* tracks the duration the evolutionary process has been running for. It stops the algorithm at the earliest end of a generation after a configured timeout duration. The *TwoPointCrossover* reproduces ArrayGenomes by splitting the linear encoding of two parents at uniformly random indices, swapping out the middle segments and turning the resulting linear encodings back into genomes. It is configurable whether the strategy returns one or both resulting genomes. *UniformRandomMutation* of an ArrayGenome with randomizable genes iterates over the linear encoding, replacing genes with randomly generated ones with a given mutation rate. *RankBasedSelection* selects a given count of elements from a list with associated fitnesses. For this the elements are first ordered by descending fitness and associated with a rank. Thereafter, elements are randomly selected with a weight proportional to their rank. For a configured selection pressure $1 \leq \lambda \leq 2$, the best element has a relative probability of λ for being chosen while the worst element is chosen with relative probability $2 - \lambda$. *EliteSelection* or n-elitism as a survivor selection strategy ensures that the best n genomes of the union of the current and last generation survive. The remaining population is thereafter selected based on a sub-strategy.

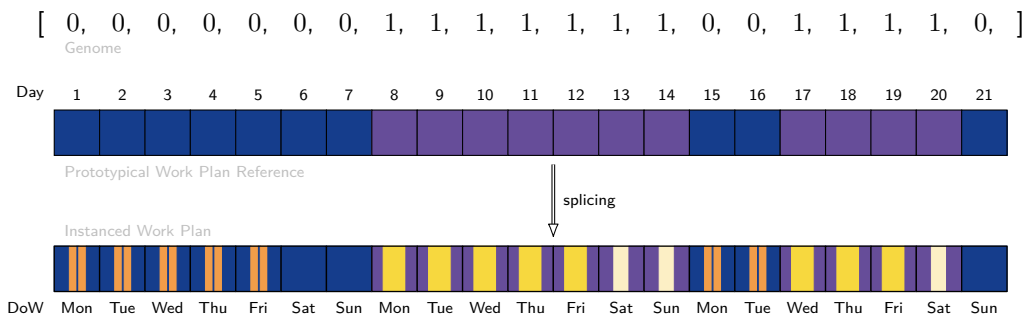
It has to be noted, that the pre-implemented strategies and the framework prioritize flexibility in use, and in general assume the evaluation of the goal function to be a non-trivial calculation, consequently posing a bottleneck. Therefore, the provided means are not built for highest performance, e.g. by focusing on efficient integer or float array encodings and operations.

3.2.4. GA Encodings

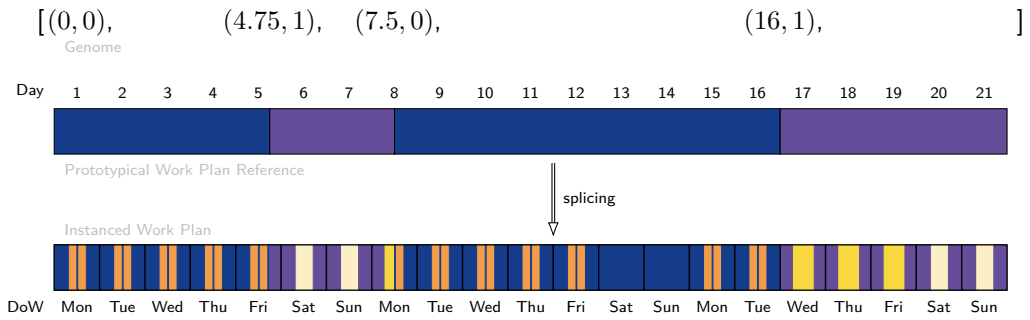
For the encoding of solutions of the WSGP we propose two variants: First, we propose a *calendar encoding* (CAL), where time is discretized into equal sized time intervals, e.g. days, such that every time interval can be assigned a reference to one of the prototypical work plans. An illustration of a day encoding with its instanced counterpart can be seen in Figure 3.4a. The second proposed encoding we call the *timeline* or *interval encoding* (INT). It is essentially a Timeline data structure as described in Section 3.2.1 with references to the prototypical work plans as data. An example of the timeline encoding can be seen in Figure 3.4b.

The CAL encoding is straight forward to implement as an array of integers, which encode the indices of the list of given prototypical work plans. It works well with all classical genetic algorithm operators, as it is intrinsically a linear encoding of values, i.e. an ArrayGenome. Randomization has been implemented by generating a random positive integer modulo the number of work plan indices. The encoding is not able to represent all solutions to the WSGP, as it only allows for work plan switches at discrete points in time. While this may lead to worse solutions, e.g. when the optimal splicing would occur at 3 o'clock in the morning, we expect this encoding to be easy to communicate to users, e.g. through a visual and colored representation of a calendar. As a consequence of their structure, CAL genomes are required to have a length equal to the number of days in a project.

The INT encoding, on the other hand, can be implemented either akin to the Timeline implementation as an array of tuples, or as a tuple of two arrays: one for the interval start points and one for work plan references. We chose to use the former. Either way, this structure is unconventional for a genome. While the classic operators could be applied to the tuples, it is standing to reason that the two different data-types should be handled independently of another to apply data-type specific operators. For timestamps, for example, we propose the *ShiftingMutation* strategy which does not replace a timestamp by a random one but adds an offset sampled from a normal distribution. This opens up a broad field of experimentation by combining and composing GA operators. As a model of WSGP solutions, we expect the INT encoding to always be able to represent the global optimum, unlike the CAL encoding. This does, however, come with the downside, that switches can occur at arbitrary times and with arbitrary frequency, e.g. switching back and forth between various work plans within minutes. To limit computational complexity and improve legibility for users, we set INT genomes to have a fixed length and thus a maximum number of switches per project.



(a) A genome with calendar encoding, which discretizes the project into days and assigns each one a reference to a prototypical work plan. Notably, the genome has the same amount of data, no matter if the work plan reference is switched daily or not.



(b) A genome with timeline encoding, which partitions the project into intervals of arbitrary length and assigns each one a reference to a prototypical work plan. Notably, this can lead to switches midday.

Fig. 3.4.: Illustration of two different encoded example genomes, together with their interpretation of consecutive work plan references and the instantiation thereof. The colors represent prototypical week work plans from Figure 3.1 as well as their intervals when instanced. The figures assume the earliest project start on Monday.

4. Evaluation

To evaluate our approach we performed experiments with the goal of answering two broad questions: 1. How well does an evolutionary algorithm solve WSGP instances with our encodings? 2. What keeps the evolutionary process from providing better solutions? The latter entails both the question of what makes the WSGP hard at it's core, and of what the limitations of our method are. To limit the scope and provide a focus, we assumed usage in a digital environment like dProB, where users make modifications to a project plan and want to see the new optimal work schedule and the resulting cost within a few seconds.

To answer the stated questions, we ran our genetic algorithms on various adapted datasets, first to determine feasible parameters, then to compare the two encodings and to observe the evolutionary process. The chosen base dataset, applied modifications and the reasoning behind the choices are presented in Section 4.1. A description of the performed experiments, results and a discussion thereof is found in later sections.

4.1. Datasets and Test Instances

To evaluate the proposed method we require problem instances of the WSGP, ideally with AoN graph structure similar to that seen in the real word. As the WSGP and the PSP/mark have not been studied before, no benchmarking datasets exist. Therefore, we chose to adapt the DSLIB dataset from [9], first referred to in "Construction and evaluation framework for a real-life project database" [BV15]. This dataset has been chosen as it contains the work breakdown structure, schedule and other documentation of 181 real world projects from the construction, IT and engineering sectors among others. The contained projects span days to years in duration and are described through scheduled activities, including their precedences. Activity counts are in the range of tens to two thousand activities and all four precedence types are featured sometimes including lag. The additional information includes resource requirements, risk analyses, cost breakdowns and controlling information among others. Since a lot of this data is of varying quality and sometimes outright missing, we only utilized additional information about agendas, which describe the work schedule of the project on an hourly basis for each day of the week. They were available for every project.

While the DSLIB dataset contains a wide variety of projects, it does not conform to our assumptions on project plans stated in Assumption 1. In general, the project plans have been formulated on a high level, with activity durations on the order of days to weeks to months even. We assume that their activity networks have been created manually, as no fully integrated tool like dProB has been widely known in the years past. Further we assume the precedences have been added with the goal of visualizing the project plan, not to provide machine input for further processing. We base this assumption on the fact that the DSLIB

contains some projects which have, in our opinion, questionable graph structures. One such example can be seen in instance *C2019-03 Emergency Department* which consists of 17 activities without any precedences. Finally, activities do not have any markers or easily machine-readable information about interruptibility, loudness, or location.

Adapting DSLIB to LIB Even though our approach to the WSGP assumes that projects consist of comparatively short activities on the order of hours to days, which form a weakly connected graph through precedences, we took the whole dataset as a starting point. To generate a dataset that we can use for testing of our method we took the following steps to adapt the DSLIB projects:

First, we parsed all activities. If an activity has been scheduled for over 12 weeks, has both no predecessors and no successors or when the activity name contains the substring "harden", we marked the activity as *passive* and made it non-interruptible. Otherwise, activities are interruptible. This rule applies to many activities that are naturally uninterruptible, like the hardening of concrete, activities which should not be interrupted for the whole project duration, like the removal of water from excavated sites, and pseudo-activities like milestones. The latter are not expected in real world applications of the WSGP, but we left them in our dataset as to not accidentally remove actual activities through imprecise removal rules.

Second, we calculated the processing duration of activities. For passive activities the processing duration is simply the originally scheduled duration in the project plan. We called this interpretation of time "wall time". For the remaining activities we calculated the duration of actual work in the original schedule, by use of the agendas provided. We call this interpretation of time "work time". As an example, one week of wall time equals $7 \cdot 24$ hours, while one week work time with 8-hour days and free weekends equals $5 \cdot 8$ hours.

Third, precedences and their types were transferred unaltered. Potential lag is interpreted as wall time. If there was no lag it is set to zero.

Fourth, we added passive dummy-activities with finish-start precedences before activities without predecessors. This is due to the observation that some projects had multiple activities start long after the project start without any reason in the form of precedences. We assumed some external reason to which we would have to submit and set the duration of the dummy-activities to the duration of the difference from the project start. This concludes the basic adaption of the DSLIB, yielding the AoN graphs for a dataset we called *LIB*.

Augmenting LIB to AUG When experimenting with instances we observed, that some instances with similar properties would perform radically different: instance *C2015-29* in the LIB dataset has 205 activities, 246 precedences and the project spans a maximum of 910 days, while instance *C2019-03* has 34 activities, 17 precedences and spans a maximum of 1161 days. Even though the former instance might appear harder to optimize from the given properties, our tests showed that a genetic algorithm working on *C2015-29* achieved 30 generations per second, while it only achieved 2.4 generations per second while working on *C2019-03*. Upon inspection, we observe that 35, 1 and 0 of 205 activities in *C2015-29* are longer than a day, week and month. In comparison, 27, 25 and 21 of 34 activities of *C2019-03* were longer than a day, week and month. This suggests, that the total sum of all

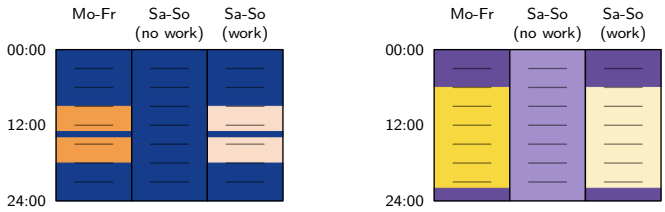


Fig. 4.1.: Overview over the components of the four prototypical work plans used for evaluation of the GA method. The plans on the left apply to the "9 to 6" scheme, those on the right to the "all-day" scheme. The parameters of work intervals are described in Table 4.1

activities' processing durations (TPD) could be an interesting parameter in determining an instance's hardness. Due to this discovery, we generated a second, augmented dataset from LIB: by splitting up all non-passive activities with a duration of over four hours into a chain of non-preemptible activities, whose individual durations are randomly chosen from a symmetric triangle distribution with a mean of $\mu = 0.75$ hours and limits of ± 0.25 hours. This way we generated new AoN graphs which better fit Assumption 1. We called this dataset *AUG*.

Work Plan Definitions In addition to project plans (AoN graphs) our method also requires prototypical work plans, as well as cost annotations. The following descriptions are made part of the LIB and AUG datasets. We used four prototypical work plans: Two with work from 9 am to 6 pm with a 1-hour lunch break at 1 pm, and two with 24-hour days. Both had a variation without work on weekends and with work on weekends. They have been chosen, as both base types represent reasonable work schedules (one 8-hour shift and three consecutive 8-hour shifts), and the weekend variations provide ways to shorten the project duration in exchange for money. An illustration of the days of the week of the work plan types can be seen in Figure 4.1.

Cost-Annotation for LIB and AUG The costs for the work plan intervals have been derived by calculating an hourly rate from the following schema: Each interval type has a base rate of 20 per hour to which a work rate is added, if the interval does not forbid any markers. The work rate has been calculated by multiplying a base work rate of 20 per hour with a factor of 2.5 on weekends and a factor of 2.5 for the "all-day" work plans. The resulting rates can be seen in Table 4.1. The splicing costs have been set to $c_{W_i,W_j} = 1000$ when $i \neq j$ or to $c_{W_i,W_i} = 0$ otherwise. The lateness penalty was $c_{late} = 1000$ per hour and the soft deadline was set to $2/3$ of the scheduled project duration in the original DSLIB schedule.

Interval Type	9 to 6			All Day			
$r_{\square}$	$\{passive\}$	$\{\}$	$\{\}$	$\{\}$	$\{passive\}$	$\{\}$	$\{\}$
$f_{\square}$	$\{loud\}$	$\{\}$	$\{\}$	$\{loud\}$	$\{loud\}$	$\{\}$	$\{\}$
For LIB and AUG $c_{\square}$	20	40	70	70	20	70	125
For MIB and MUG $c_{\square}$	5	25	55	205	5	205	512

Tab. 4.1.: Overview of the work plan parameters and cost rates of the prototypical work plans used for evaluation of our method. The colors refer to those of the day structures in Figure 4.1.

Maximum Project Duration Both the CAL encoding and the Timeline backing the INT encoding require an upper bound for the project duration. Therefore, we scheduled each instance of each dataset once for each prototypical work plan, only using said work plan. We then set the upper project duration bound to the highest calculated makespan. Similarly, we calculated the cost of each of the four schedules to determine a fitness baseline to compare the results of our method against, as no optimal solutions are known for our datasets instances.

MIB and MUG Datasets After running some experiments with the LIB and AUG datasets, in which our GA did not perform well against the baseline in most instances, as described in Section 4.2, we decided to create two more datasets *MIB* and *MUG*. The goal was to create "harder" instances, in which our trivial fitness baseline was further from the optimum. For this we took the LIB and AUG datasets and marked 5% of all non-passive activities with a *loud* marker. For costs, we tweaked the base rate to 5 and set the "all-day" factor to 10. The switching cost has been set to $c_{W_i,W_j} = 2000$ when $i \neq j$ or to $c_{W_i,W_i} = 0$ otherwise and the lateness penalty has been set to 500 per hour. An overview over the work plan parameters can be seen in Table 4.1 and a summary of the four datasets can be found in Table 4.2.

Dataset	Augmented	Passive Markers	Loud Markers
LIB		×	
MIB		×	×
AUG	×	×	
MUG	×	×	×

Percentile		0th	25th	50th	75th	100th
Project Days		7.0	3.4×10^2	6.4×10^2	1.1×10^3	9.2×10^3
TPD		4.5×10^1	1.1×10^3	3.8×10^3	3.2×10^4	1.4×10^7
A. Count	LIB	9.0	2.5×10^1	3.4×10^1	6.7×10^1	2.2×10^3
	MIB	9.0	2.5×10^1	3.4×10^1	6.7×10^1	2.2×10^3
	AUG	2.6×10^1	1.0×10^3	2.2×10^3	4.9×10^3	1.2×10^5
	MUG	2.6×10^1	1.0×10^3	2.2×10^3	4.9×10^3	1.2×10^5
A. Duration	LIB	2.2	4.0×10^1	1.3×10^2	4.6×10^2	7.2×10^3
	MIB	2.2	4.0×10^1	1.3×10^2	4.6×10^2	7.2×10^3
	AUG	7.1×10^{-1}	7.4×10^{-1}	1.5	1.0×10^1	3.5×10^3
	MUG	7.2×10^{-1}	7.4×10^{-1}	1.5	1.0×10^1	3.5×10^3
Preempt. (%)	LIB	5.0×10^1	8.2×10^1	9.3×10^1	9.6×10^1	10.0×10^1
	MIB	5.0×10^1	8.2×10^1	9.3×10^1	9.6×10^1	10.0×10^1
	AUG	0.0	0.0	7.3×10^{-2}	4.7×10^{-1}	6.9×10^1
	MUG	0.0	0.0	7.3×10^{-2}	4.7×10^{-1}	6.9×10^1
Loud (%)	LIB	0.0	0.0	0.0	0.0	0.0
	MIB	0.0	0.0	8.5×10^{-3}	2.3×10^{-2}	4.9×10^{-2}
	AUG	0.0	0.0	0.0	0.0	0.0
	MUG	0.0	0.0	1.0×10^{-3}	2.5×10^{-2}	1.8×10^{-1}
Passive (%)	LIB	4.9×10^{-3}	4.0×10^{-2}	8.0×10^{-2}	2.9×10^{-1}	1.0
	MIB	4.9×10^{-3}	4.0×10^{-2}	8.0×10^{-2}	2.9×10^{-1}	1.0
	AUG	1.6×10^{-4}	7.5×10^{-4}	1.4×10^{-3}	6.0×10^{-3}	1.0
	MUG	1.6×10^{-4}	7.5×10^{-4}	1.4×10^{-3}	6.0×10^{-3}	1.0

Tab. 4.2.: Short overview over the four datasets used for evaluation at the top. Statistical summary of instance parameters maximum project days, total sum of processing times (TPD), activity count, mean activity duration in hours, and percentage of activities that are preemptible, loud, and passive. The rows are split by dataset where differences exist. Notably all differences are between the LIB+MIB and AUG+MUG groups, whereas hardly any differences can be seen inside the groups. The percentage of loud activities does, however, differ in that only M* datasets have them, and they show a significant difference in percentile values.

4.2. EA Configuration and Parameter Study

To evaluate the proposed encodings, we first had to decide on a configuration of an evolutionary algorithm within our framework. This includes strategies, parameters and the concrete fitness function. As they differ in our various experiments, the following paragraphs provide a baseline configuration chosen if not stated otherwise.

Configuration The fitness function used was the logarithm with base 10 of a scheduled project's cost $-\log_{10}(C)$. It has been chosen to give differences of the same value progressively more weight as a project gets cheaper. We assume this to yield better results than simply using the linear cost $-C$, should a proportional selection operator be used. This assumption has, however, not been verified as we chose to use rank-based selection.

In accordance with our assumption from the beginning of this chapter, expecting use in an interactive planning and scheduling context, we chose to terminate the evolutionary processes through TimeoutTermination after 10 seconds. For parent and survival selection operator we chose Rank-Based Selection with $\lambda = 2$, as we did not want to risk premature convergence in local minima which the literature suggests of TournamentSelection [Jeb13]. In addition, the survival selection makes use of 2-elitism as to not "lose" the best solutions. For the reproduction strategy we chose Two-Point-Crossover in hopes that inserting segments from and into the baselines would yield good results quickly in the first generations. The chosen mutation operators for both the CAL encoding and the work plan references in the INT encoding was UniformRandomMutation. The timestamps of the INT encoding have been mutated by ShiftingMutation with $\mu = 0$ and $\sigma = 7$ days. The mutation processes of the work plan reference and the timestamp of any given INT gene were independent. Finally, for repair operators, we enforced that the values were in the range of the work plan reference indices. Mutation of start times for INT could, however, yield shorter genomes when two intervals start at the same time due to implementation details. We therefore added new random genes if any INT genome had fewer genes than their intended genome length of 10.

To initialize the populations four genomes, one for each prototypical work plan, were generated which only featured that work plan. The remaining individuals were uniformly sampled from the domain and therefore had no special internal structure. In combination with the elitism in survival selection, we ensured that the evolutionary process would not yield results worse than the baselines.

Parameter Study (Experiment 1) Finally, we had to select a mutation rate and a population size. For this we performed a small parameter study as a first experiment. To this end, we ran ten iterations with incrementing randomization seeds of our genetic algorithm for each combination of encoding (INT, CAL), mutation rate ($\{0.01, 0.05, 0.1, 0.2\}$) and population size ($\{10, 25, 50\}$), for the first ten instances of both initial datasets (LIB, AUG). This resulted in 4800 EA runs. To evaluate the runs, we used the relative improvement in fitness in relation to the fitness baseline of each instance. A visualization of the results can be seen in Figure 4.2.

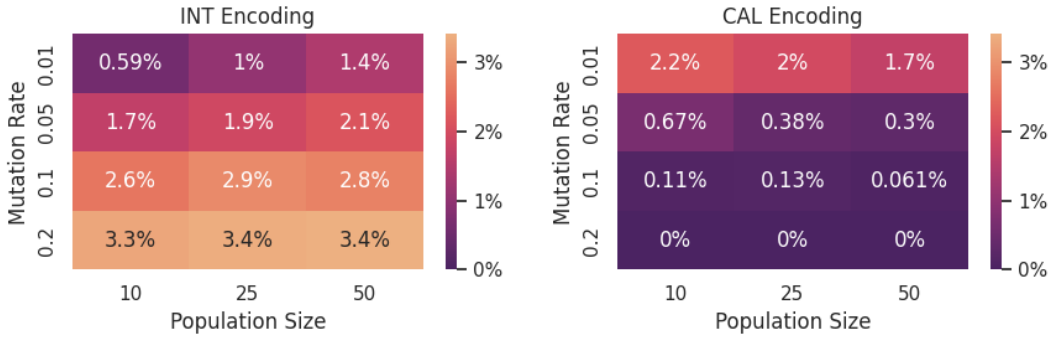


Fig. 4.2.: Visualization of mean improvement depending on mutation rate and population size parameter tests and split by encoding. The heatmaps show, that CAL and INT have vastly different optimal parameters. Further INT appears to yield better results on average.

The results show, that the experiments with CAL encoding yielded better results with a low mutation rate of 1%, while INT encodings yielded best results with high mutation rates of 20%. The population size appears to have a significantly lower impact on improvement than the mutation rate. While the data of the CAL subset might suggest that lower population sizes for, the INT subset does not immediately suggest any impact. To verify these observations, we performed an analysis of variance (ANOVA) to statistically show which parameter has a statistically significant impact on the mean of improvement. We also performed post-hoc tests with the conservative Bonferroni correction, to show which parameter values differ significantly. The results of the analyses are available in Appendix C.

The statistical analysis showed, that for both subsets the influence of mutation rate is in fact highly significant ($p < 0.1\%$) and that the influence of mutation rate values on the mean of improvement differs highly significantly ($p_{bonf} < 0.1\%$), except for 10% and 20%. For the INT subset the two sub-subsets can be distinguished with less but still high significance ($p_{bonf} = 0.3\%$), for the CAL subset their impact cannot be distinguished. The influence of population size on mean improvement was significant for the INT subset ($p \leq 2.5\%$) and highly significant for the CAL subset ($p \leq 1\%$). However only the population sizes of 10 and 50 could be distinguished with statistical significance ($p_{bonf} < 2.5\%$).

We assume, that the strong difference between optimal mutation rate for INT and CAL genomes might stem from the fact that INT genomes are comparatively short, while CAL genomes are very long. This gives changes in INT genes more significant impact on fitness. The fact, that some timestamps in INT genomes lie after the makespan of the given project, effectively shortening the genomes even further, amplifies this effect. In addition to the lower impact of each gene, we assume another effect with CAL genomes: as mutation rate is not applied per genome but per gene, too many changes might happen at once without any feedback in form of an evolutionary cycle. Instead, the impact on cost and makespan of the modified genes might cancel each other out. We concluded the parameter study by choosing a mutation rate of 1% for CAL genomes and 20% for INT genomes, as well as a population size of 25 for all further experiments.

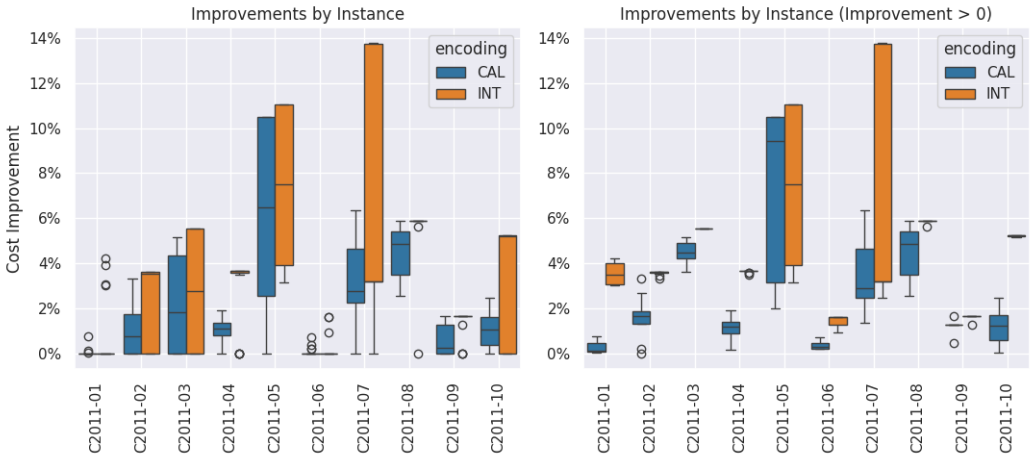


Fig. 4.3.: Per-instance improvements of a subset of the parameter study results with a population size of 25, a mutation rate of 20% for INT genomes and a mutation rate of 1% for CAL genomes. While the left plot shows the whole subset as described, the plot on the right shows only the sub-subset of instances that yielded any improvement greater zero. The variance in improvement is low in some instances, implying reliable convergence behavior, but very large in others. The data suggests that INT generally performs better than CAL.

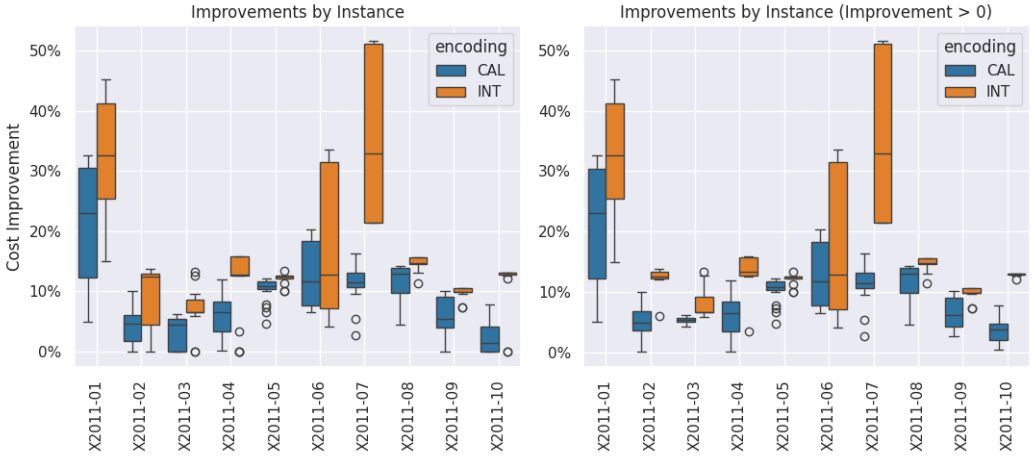


Fig. 4.4.: Per-instance improvements of ten runs of the first ten MIB and MUG instances. While the left plot shows the whole subset as described, the plot on the right shows only the sub-subset of instances that yielded any improvement greater zero. In comparison to Figure 4.3 the significantly larger y-axis has to be noted. The data suggests that INT generally performs better than CAL.

4.3. Experimental Findings

After finalizing the EA configuration and determining its parameters, we first engaged in further exploratory analysis of the results of parameter study, yielding knowledge about our limitations in estimating a WSGP instance's hardness and issues of our mathematical model. Afterwards we performed a final comparison of the CAL and INT encodings.

Hardness of Instances (Experiment 2) Exploratory analysis of the data of the parameter study showed, that even with the best choice in parameters a third of the runs did not yield any improvement. Looking at the statistical distribution of improvement per instance in Figure 4.3, we see that for some instances almost no runs yielded an improvement. Under the assumption, that this was due to "too good" of a baseline, this led to the development of the MIB and MUG datasets. They had the goal to have instances for which our method of generating baselines would yield worse results.

To validate that the MIB and MUG datasets do indeed get higher improvements, we ran a second experiment on them with both encodings, ten times on each of the two datasets instances. In total this yielded 7240 runs. To ensure comparability to the results of the parameter study, only the first 10 instances were inspected here. The data shows a significant improvement over the LIB and AUG datasets. A view on the per-instance distribution suggested a large improvement, as can be seen by comparing Figure 4.3 to Figure 4.4. Whereas in LIB and AUG about a third of runs did not yield an improvement, for MIB and MUG only about a tenth of runs on the first ten instances did not yield an improvement, as can be seen in Table C.7. Furthermore, the mean cost improvement increased from 2.6% to 8.9% with the CAL encoding and from 3.4% to 16.3% with the INT encoding.

This confirms our assumption, that the fitness baseline of the LIB and AUG datasets yielded too good results, such that it was comparatively hard to improve them. The finding also highlights the fact, that the "hardness" of an instance is a vague measure, as it is not immediately clear what it means and how to measure it. This is especially the case when experimenting on synthetic datasets without real world data or at least verified data to compare against. To try to quantify what contributes to better improvements within the MUG dataset, we tried to apply a linear regression analysis. However, no model could be found which would explain more than 5% of the variance in the dataset based on the instance properties *Activity Count*, *Maximum Project Days*, *Total Processing Duration* and *Percentage of Preemptible Activities*, as seen in Appendix C. While statistically significant correlations can be found, as seen in Table C.9, it was to be expected that of the properties above only the percentage of interruptible activities correlated positively with improvement.

Experiment 3 To further study convergence behavior, we performed a third experiment. It contained five runs for each of the first five MIB and MUG instances. The results, seen in Figure 4.5, show that CAL runs had continual improvement but significant spread in fitness within each generation, while INT runs usually show fast convergence within 10 seconds and show only occasional improvements afterwards. INT runs also showed low inner-generation spread. This implies, that selection pressure and mutation should be tweaked further.

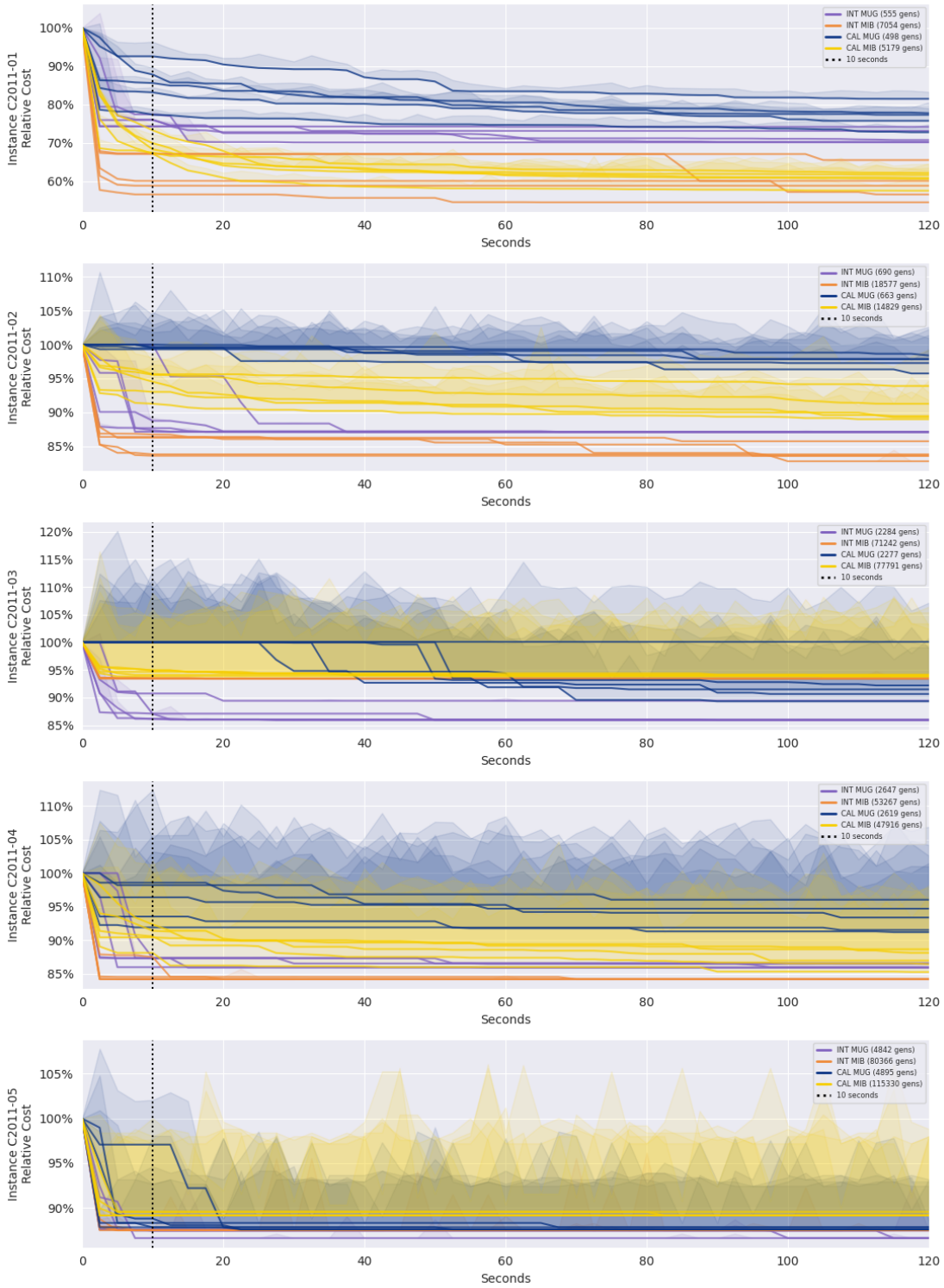
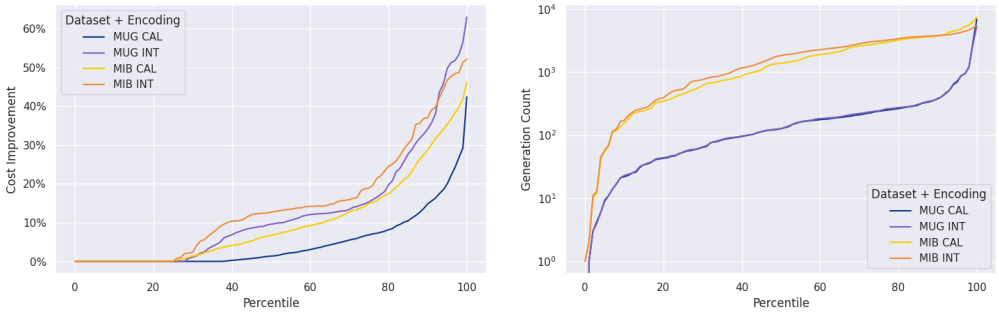


Fig. 4.5.: Overview of the convergence behavior of five runs on each of the first five instances of the MIB and MUG datasets, sampled in 2.5 second intervals. Bold lines indicate the best genome in a generation, the filled areas span from the median to the best genome and thus show a bracket of half the individuals.

Tab. 4.3.: Descriptive Statistics of improvement and generation count for 10 runs of the EA for each instance in the MIB and MUG datasets, and for each encoding.

Encoding	Cost Improvement (in %)				Generation Count			
	CAL		INT		CAL		INT	
Dataset	MUG	MIB	MUG	MIB	MUG	MIB	MUG	MIB
Mean	4.7	10.1	12.6	14.3	242	1758	236	1876
Std. Deviation	7.0	11.2	14.5	14.0	570	1520	481	1393
Minimum	0.0	0.0	0.0	0.0	0	1	0	1
Median	1.4	6.7	9.6	12.7	126	1383	127	1818
Maximum	42.5	46.1	63.1	52.2	6826	7397	5302	5457



(a) Percentiles of improvement in cost. The INT encoding appears to have a more closely related distributions between the MUG and MIB runs than the CAL encoding.

(b) Percentiles of generation count. MUG runs appear to achieve an order of magnitude less generations in the same time as MIG runs.

Fig. 4.6.: Plots of improvement and generations in the results of the second experiment.

Final Analysis of Encodings To properly evaluate the two proposed encodings, we analyzed all 7240 runs of the second experiment. Their descriptive statistics can be found in Table 4.3. The statistics show that the INT encoding appears to yield better results than CAL in general. Nevertheless, both encodings struggle to yield any improvement about 25% of the time. Both methods achieve better improvements on the MIB dataset, however, the MUG difference between the 50th and 75th percentiles appears to be larger with the CAL encoding, indicating that it struggles more with the MUG dataset than the INT encoding. In the best case, i.e. the 100th percentile, the INT encoding even yields better improvements on the MUG dataset. These findings are confirmed by the percentile plots in Figure 4.6a.

Shifting our focus to the generation count achieved per run, the data does not show a major difference between the encodings. It does however show, that runs on instances from the MIB dataset achieved about an order of magnitude more generations within the 10-second time limit than runs on the MUG dataset. We assume this to be due to the higher activity counts, as seen in Table 4.2. Therefore, a more performant scheduling and cost calculation could aid in a higher generation count and thus in better improvements. This causal connection should, however, be verified before a significant time investment is made.

4.4. Miscellaneous Findings

After presenting the statistical evaluation and concrete results, this section aims to share possible paths of improvement based on theories about impediments and general learnings about the PSP/mark and the WSGP.

Opportunities for Improved WSGP Encodings An easily conceivable way to narrow down the vast domain of the CAL encoding could be discrete the interval encoding, i.e. a combination of the time discretization of the CAL encoding with the sparse work plan switch-based intervals of the INT encoding. This, together with an improved population initialization scheme would, in our opinion, probably bring great improvements over the CAL encoding's results. It would also possibly yield better results than the INT encoding. It would, however, suffer from the same central issue its inspirations were held back by: the encoded work plan switches are linked to the time since the project started. After formulating the encodings, it was discovered that every improvement in generated work plans early on during a projects makespan has a significant probability to worsen whatever comes after. As an example, let us assume an AoN graph and a work plan which are perfectly fitted to each other, except for one interval in the beginning. Should the EA now find the perfect assignment for said interval, allowing an activity to be scheduled earlier than before, then there is a high probability of all subsequent activities to be scheduled at other times than before. This would lead to an increase in cost and thus death of the individual in the evolutionary system unless all subsequent work intervals adjust at the same time.

While the "additive" nature of the ASAP solution to the PSP/mark, where topologically later activities cannot influence earlier activities, is a useful property in some situations, in this situation the reverse is true. As every earlier activity may influence later activities when on their critical path, every change in work plans may invalidate the optimality of all subsequent work plan intervals. Therefore, we propose future work to study an encoding that ties work plan switches to the starts or end of activities. This idea is based on the work on serial schedule generation schemes, where the order of activities to be scheduled is determined independent of time.

Challenges in Adapting Classical Schedule Modelling Tools Finally, working on the PSP/mark yielded a fact which had not been discussed in the literature on project scheduling problems: When introducing advanced automated scheduling systems, tools or parameters used for manual scheduling and visualization may change or even lose their meaning. The most concrete example for this discovered while working on the PSP/mark was that of multiple types of precedence constraints.

When developing a solution for the PSP/mark, we first attempted to implement a critical path method based approach, which first calculated an As Soon As Possible and an As Late As Possible schedule. Thereafter, it was our goal to optimize for cost by shifting activities around within their float. This approach has, however, been abandoned, as especially the ALAP schedule yielded results which would not be feasible in real life: when planning interruptible activities, the algorithm opted to schedule the start of some activities at the instant before

work plan intervals that forbid them. Incidentally, we noticed a similar behavior with start-finish and finish-finish precedences, which scheduled infinitesimally small quantities after the weekend. This led to scrutinization of all precedence types except finish-start, since it is unclear what exactly they and their potential lag model.

We assume, that all precedence types but finish-start are a circumvention of the fact that most project plans to date operate on a very high level. We assume, that they are often used, to represent dependencies on sub-activities. For example: the fact, that some plaster can only be post-processed after it dried for some time could be modeled as two long activities with start-start precedences and an offset for the drying time. In this case the activities would stand in for all plastering and post-processing activities on the whole site. If one were to split-up the process per wall, however, only finish-start precedences would be needed. This variant of modeling would also be robust against interruptions, whereas in the first variant the post-processing activity could overtake the plastering activity if they have different markers.

There are many ways to interpret the criticized constraints, for example that the percentage of completion of an activity may not overtake the percentage of completion of another activity. However, we argue that it would be best to model projects in more detail or at least preprocess AoN graphs, such that they only feature precedences with a clear and concise meaning.

5. Conclusion

This thesis contributes to the field of optimized project scheduling, by identifying an under-researched niche: project scheduling with environmental resources, which are either available or unavailable for all processes at the same time, especially concerning optimization of opportunity cost. We formalized this as the Project Scheduling Problem with Marked Activities (PSP/mark) and the Work Schedule Generation Problem (WSGP) aimed at cost-optimized resource availability schedules. In an attempt to solve the stated problems, we built an ASAP scheduling scheme, which is thereafter used in an evolutionary algorithm with two encodings (CAL and INT) that aims to optimize a project's cost. In multiple iterations, we adapted the DSLIB dataset which we first used to tweak the evolutionary parameters in a parameter study and then used to compare the proposed encodings. The results showed that the INT encoding generally outperforms the CAL encoding with a mean improvement in cost of 14.3% versus 10.1% on the MIB dataset. Our evaluations also highlighted difficulties in designing benchmarking instances, in formulating an effective encoding for solutions of the WSGP, and in adapting classical scheduling tools to novel systems.

Our work made first steps towards introducing environmental resources into cost optimized scheduling for large projects, potentially leading to more cost-effective plans. On the theoretical side, we contributed by highlighting that some factors, like the permission to perform loud activities or access to an area, can be modeled as environmental resources. This allows for optimization in a less computationally intensive manner compared to the RCPSP, as activities in the PSP/mark do not compete for resources. On the practical side, we contributed through a solution to the PSP/mark and a first attempt at a WSGP solution.

The contributions are, however, not without limitations. A key limitation is the lack of real-world data to verify the applicability of both the problem and the proposed solutions. As fully integrated and interactive project planning, scheduling, and evaluation tools like dProB are just emerging as of time of writing, industries like the construction industry cannot provide the detailed project and cost plans expected by the WSGP without major effort. Further, it is not yet entirely clear which environmental parameters could and should be modeled as environmental resources. Therefore, the adapted dataset, although useful for a comparative evaluation of encodings and EA parameters, does not allow for any conclusions about real-world optimization potential. To remedy this, a future work should focus on interviewing industry professionals, hand-crafting and verifying WSGP instances. While this work could benefit from the ability to abstract, found in the computer science community, a background in engineering would be beneficial, too.

Another limitation is the underutilization of the framework for evolutionary algorithms due to temporal constraints. As alluded to in Section 3.2.3, the literature did not yield clear guidance in the decision-making process regarding the selection of evolutionary operators and parameters. It did not give the impression that all methods were created equal, i.e.

that it was impossible to have a structured approach in designing an EA. Instead of an overarching structured and empirical analysis, much of the literature appeared to promote a "trial and error" approach informed by highly problem-specific prior instances of success. In hindsight, the time spent researching evolutionary operator literature could have been used more effectively by directly experimenting with operators for this specific problem context. A future work could focus on evaluating the various operators for the WSGP with the MUG dataset, though this does not necessarily need to be done with our EC framework but could be done with ParadisEO [Dre+21].

Similarly, this thesis' findings are limited by the fact that the two proposed encodings for WSGP solutions are very basic. It would for example be easy to propose an INT-like encoding that only allows for discrete time-steps. This could combine the easier legibility for users and the minimum distance between work plan switches of the CAL encoding, with the INT encodings improved performance due to the drastically smaller domain. As alluded to in Section 4.3, we believe that an encoding of work plan switches coupled to the start and end times of activities may yield better results than an encoding coupled to the time since project start.

In conclusion, this work has opened up a subfield of project schedule optimization by proposing two new formal problems and proposing first solutions. While the real-world applicability remains to be validated, the work highlighted points of vantage for future work in academia, to gain a deeper understanding of the problems at hand, and for industry, to provide more and better data for research. The author is optimistic that significant innovations in project planning and scheduling, along with their societal benefits, can be achieved through the recognition of the mutual benefit of regular communication and steady collaboration of industry and all parts of academia.

Bibliography

[Ach07] Tobias Achterberg. "Constraint Integer Programming". PhD thesis. 2007.

[Ach09] Tobias Achterberg. "SCIP: solving constraint integer programs". English. In: *Math. Program. Comput.* 1.1 (2009), pp. 1–41. ISSN: 1867-2949. DOI: 10.1007/s12532-008-0001-1.

[ADD21] Amine Aziz-Alaoui, Carola Doerr, and Johann Dréo. "Towards Large Scale Automated Algorithm Design by Integrating Modular Benchmarking Frameworks". In: *CoRR* abs/2102.06435 (2021). arXiv: 2102.06435. URL: https://arxiv.org/abs/2102.06435.

[Afs14] Behrouz Afshar-Nadjafi. "Resource Constrained Project Scheduling Subject to Due Dates: Preemption Permitted with Penalty". In: *Advances in Operations Research* 2014 (2014), pp. 1–10. DOI: 10.1155/2014/505716. URL: https://doi.org/10.1155/2014/505716.

[AM14] Behrouz Afshar-Nadjafi and Mahyar Majlesi. "Resource constrained project scheduling problem with setup times after preemptive processes". In: *Computers & Chemical Engineering* 69 (2014), pp. 16–25. ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2014.06.012. URL: https://www.sciencedirect.com/science/article/pii/S0098135414001914.

[Ban12] Patrick Bangert. *Optimization for Industrial Problems*. Springer Berlin Heidelberg, 2012. ISBN: 9783642249747. DOI: 10.1007/978-3-642-24974-7. URL: http://dx.doi.org/10.1007/978-3-642-24974-7.

[BB14] Andrew Baldwin and David Bordoli. "An Introduction to Planning and Scheduling". In: *A Handbook for Construction Planning and Scheduling*. John Wiley & Sons, Ltd, 2014. Chap. 1, pp. 3–35. ISBN: 9781118838167. DOI: https://doi.org/10.1002/9781118838167.ch1. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118838167.ch1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118838167.ch1.

[BV15] Jordy Batselier and Mario Vanhoucke. "Construction and evaluation framework for a real-life project database". In: *International Journal of Project Management* 33.3 (2015), pp. 697–710. ISSN: 0263-7863. DOI: https://doi.org/10.1016/j.ijproman.2014.09.004. URL: https://www.sciencedirect.com/science/article/pii/S0263786314001410.

[CT15] Min-Yuan Cheng and Duc-Hoc Tran. "Opposition-based Multiple Objective Differential Evolution (OMODE) for optimizing work shift schedules". In: *Automation in Construction* 55 (2015), pp. 1–14. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2015.03.021. URL: https://www.sciencedirect.com/science/article/pii/S0926580515000618.

[Dre+21] Johann Dreo et al. "Paradiseo: from a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of Paradiseo". In: July 2021, pp. 1522–1530. DOI: 10.1145/3449726.3463276.

[Eng07] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley, Oct. 2007. ISBN: 9780470512517. DOI: 10.1002/9780470512517. URL: http://dx.doi.org/10.1002/9780470512517.

[Gle+21] Ambros Gleixner et al. "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library". In: *Mathematical Programming Computation* (2021). DOI: 10.1007/s12532-020-00194-3. URL: https://doi.org/10.1007/s12532-020-00194-3.

[GZ22] Kai Guo and Limao Zhang. "Multi-objective optimization for improved project management: Current status and future directions". In: *Automation in Construction* 139 (2022), p. 104256. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2022.104256. URL: https://www.sciencedirect.com/science/article/pii/S0926580522001297.

[HB22] Sönke Hartmann and Dirk Briskorn. "An updated survey of variants and extensions of the resource-constrained project scheduling problem". In: *European Journal of Operational Research* 297.1 (2022), pp. 1–14. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2021.05.004. URL: https://www.sciencedirect.com/science/article/pii/S0377221721003982.

[HHA24] Chris Hendrickson, Carl Haas, and Tung Au. *Project Management for Construction (and Deconstruction) - Fundamental Concepts for Owners, Engineers, Architects and Builders*. Carl Thomas Michael Haas, Mar. 1, 2024. ISBN: 978-1-7383557-0-9.

[Jeb13] Khalid Jebari. "Selection Methods for Genetic Algorithms". In: *International Journal of Emerging Sciences* 3 (Dec. 2013), pp. 333–344. URL: https://www.researchgate.net/publication/259461147_Selection_Methods_for_Genetic_Algorithms.

[KCK20] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. "A review on genetic algorithm: past, present, and future". In: *Multimedia Tools and Applications* 80 (2020), pp. 8091–8126. URL: https://api.semanticscholar.org/CorpusID:226227415.

[Kre+18] Stefan Kreter et al. "Mixed-integer linear programming and constraint programming formulations for solving resource availability cost problems". In: *European Journal of Operational Research* 266.2 (2018), pp. 472–486. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2017.10.014. URL: https://www.sciencedirect.com/science/article/pii/S037722171730927X.

[Li+14] R. Li et al. "Estimating railway infrastructure project cost from transferring nominal price to real price by considering the working time possessions". In: *Computers in Railways XIV Special Contributions*. CRS14. WIT Press, Oct. 2014. DOI: 10.2495/crs140011. URL: http://dx.doi.org/10.2495/CRS140011.

[LO07] John Lancaster and Mustafa Ozbayrak. "Evolutionary algorithms applied to project scheduling problems-a survey of the state-of-the-art". In: *International Journal of Production Research* 45.2 (2007), pp. 425–450. DOI: 10.1080/00207540600800326. eprint: https://doi.org/10.1080/00207540600800326. URL: https://doi.org/10.1080/00207540600800326.

[MT24] Büsra Meniz and Fatma Tiryaki. "Genetic Algorithm Optimization with Selection Operator Decider". In: *Arabian Journal for Science and Engineering* (May 2024). DOI: 10.1007/s13369-024-09068-5.

[Net+07] Nicholas Nethercote et al. "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming – CP 2007*. Ed. by Christian Bessière. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543. ISBN: 978-3-540-74970-7.

[Pin16] Michael L. Pinedo. *Scheduling - Theory, Algorithms and Systems*. Springer International Publishing, 2016. ISBN: 9783319265803. DOI: 10.1007/978-3-319-26580-3. URL: http://dx.doi.org/10.1007/978-3-319-26580-3.

[Pol+20] Oliver Polo-Mejía et al. "Mixed-integer/linear and constraint programming approaches for activity scheduling in a nuclear research facility". In: *International Journal of Production Research* 58.23 (2020), pp. 7149–7166. DOI: 10.1080/00207543.2019.1693654. eprint: https://doi.org/10.1080/00207543.2019.1693654. URL: https://doi.org/10.1080/00207543.2019.1693654.

[Rei+24] Julian Reisch et al. "Eine Potentialabschätzung zur Reduktion der von Baustellen betroffenen Zugfahrten". In: *HEUREKA* (2024).

[Tal09] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley, June 2009. ISBN: 9780470496916. DOI: 10.1002/9780470496916. URL: http://dx.doi.org/10.1002/9780470496916.

[TKA17] Algan Tezel, Lauri Koskela, and Zeeshan Aziz. "Lean thinking in the highways construction sector: motivation, implementation and barriers". In: *Production Planning & Control* 29 (Dec. 2017), pp. 1–23. DOI: 10.1080/09537287.2017.1412522.

[WC95] R.A. Walker and S. Chaudhuri. "Introduction to the scheduling problem". In: *IEEE Design & Test of Computers* 12.2 (1995), pp. 60–69. DOI: 10.1109/54.386007.

Miscellaneous References

[1] Sven Wroblewski Petya Tsvyatkova. *I NV Leitfaden Bau- und Sperrzeitenkatalog*. Tech. rep. DB Netz AG I.NIG 41, 2021. URL: https://www.deutschebahn.com/resource/blob/6892074/5d89ccc5d7732f21653024ca1a27c1fa/Leitfaden-Bau-und-Sperrzeiten_Rev-01-data.pdf (visited on 03/19/2024).

[2] Microsoft Corporation. *Microsoft Project*. URL: https://www.microsoft.com/en-us/microsoft-365/project/project-management-software.

[3] Leopard Project Controls. *How to build a construction schedule with MS Project from Project Documents?* URL: https://consultleopard.com/how-to-build-a-construction-schedule-with-ease/ (visited on 05/29/2024).

[4] Hans Mittelmann. *Benchmarks for Optimization Software - LPopt Benchmark*. Mar. 27, 2024. URL: https://plato.asu.edu/ftp/lpopt.html (visited on 04/10/2024).

[5] Hans Mittelmann. *Benchmarks for Optimization Software - The MIPLIB2017 Benchmark Instances Table for 8 Threads*. Apr. 5, 2024. URL: https://plato.asu.edu/ftp/milp_tables/8threads.res (visited on 04/10/2024).

[6] Geoffrey Chu et al. *Chuffed - a lazy clause generation solver*. URL: https://github.com/chuffed/chuffed (visited on 04/10/2024).

[7] Brent Yorgey. *Topological sorting: pseudocode and analysis*. Sept. 20, 2019. URL: http://ozark.hendrix.edu/~yorgey/382/static/topsort.pdf (visited on 05/30/2024).

[8] Diego Giacomelli. *GeneticSharp - GitHub*. May 30, 2024. URL: https://github.com/giacomelli/GeneticSharp/ (visited on 05/30/2024).

[9] M. Vanhoucke, J. Coelho, and J. Batselier. *DSLIB Dataset - Emperiacl Project Data*. URL: https://www.projectmanagement.ugent.be/research/data (visited on 05/22/2024).

[10] JASP Team. *JASP (Version 0.18.3)[Computer software]*. 2024. URL: https://jasp-stats.org/.

A. CIP for ASAP Scheduling Problem

Before utilizing an exact algorithm for solving the ASAP scheduling Problem with the given formal definitions, we tried a CIP approach for a slightly modified problem with the following modifications: precedences could feature a maximum lag sub-constraint and could only be of type finish-start, and work plans were defined through restriction calendars akin to calendars in [Kre+18]. Different to their implementation however we did not discretize time.

The CIP below solves the modified ASAP scheduling problem. Given parameters are

- a restriction calendar of k restriction intervals $r \in R$ each with restriction type $t_r \in T$,
- a set of n activities $a \in A$ each with a set of restriction types $\rho_a \subseteq T$ which forbid a from being processed during some restriction r' if $t'_r \in \rho_a$, and
- a set of m precedences $(a_1, a_2) = \pi \in \Pi$ with a defined lag l_π,
- an absolute earliest start time and end time s, e.

To translate the calendars above into the work plans from our proposed formulation, each combination of markers would have to be translated into a distinct restriction type. In the worst case, this would mean T is the power set of all used markers.

The CIP utilizes variables s_a and e_a to indicate the scheduled start and end times of a, a set of variables i_{ra} which indicates whether restriction r and variable a are intersecting, and a set of variables b_a which indicates how long a is forbidden within s_a and e_a.

minimize	$n \cdot 10^3 \cdot \max e_a + \sum b_a$		
subject to	$s_a \geq s$	$a \in A$	activity box constraint
	$e \geq e_a$	$a \in A$	activity box constraint
	$e_a \geq s_a$	$a \in A$	activity feasibility
	$l_\pi \geq l_\pi^{\min}$	$\pi \in \Pi$	lag box constraint
	$l_\pi^{\max} \geq l_\pi$	$\pi \in \Pi$	lag box constraint
	$s_{a_2} \geq e_{a_1} + l_\pi$	$(a_1, a_2) = \pi \in \Pi$	precedence constraint
	$e_a \geq s_a + p_a + b_a$	$a \in A$	end calculation
	$i_{ra} \leftrightarrow (e_a > s_r \wedge e_r \geq s_a \wedge t_r \in \rho_a)$	$a \in A, r \in R$	interrupting intersections
	$b_a \geq \sum_{r \in R:\, i_{ra}} e_r - s_r$	$a \in A$	break calculation

Notably the addition of maximum lag provides the ability to over constrain the problem, meaning it allows problem instances to be unsolvable. The definitions in Section 3.1 are built in such a way, that problem instances can not be over-constrained through precedence constraints, as the AoN graphs are required to be directed acyclic graphs. Additional constraints may lead to activities be pushed later in time, but they cannot make the problem unsolvable by themselves.

The code has been written in the MiniZinc language [Net+07] and has been solved with the SCIP 8 [Ach09] constraint integer program solver.

```
type floatplus = 0.0..1000000000.0;         % basic box constraint for all variables
enum restrictions = { Quiet, All };         % definition of restriction

int: n;                                     % Activities
set of int: ACTIVITY = 1..n;
array[ACTIVITY] of floatplus: duration;
array[ACTIVITY] of set of restrictions: restricted_by;
array[ACTIVITY] of var floatplus: start;
array[ACTIVITY] of var floatplus: end;
constraint forall(i in ACTIVITY) (end[i] > start[i]);

int: m;                                     % Precedences
set of int: PRECEDENCE = 1..m;
array[PRECEDENCE] of ACTIVITY: source;
array[PRECEDENCE] of ACTIVITY: target;
array[PRECEDENCE] of floatplus: min_lag;
array[PRECEDENCE] of floatplus: max_lag;
constraint forall(i in PRECEDENCE, j in PRECEDENCE) (source[i] != target[i]);
constraint forall(i in PRECEDENCE) (min_lag[i] >= 0);
constraint forall(i in PRECEDENCE) (max_lag[i] >= min_lag[i]);
array[PRECEDENCE] of var floatplus: lag;
constraint forall(i in PRECEDENCE) (lag[i] >= min_lag[i]);
constraint forall(i in PRECEDENCE) (lag[i] <= max_lag[i]);
constraint forall(i in PRECEDENCE) (start[target[i]] >= end[source[i]] + lag[i]);

int: k;                                     % Restriction Calendars
set of int: RESTRICTION = 1..k;
array[RESTRICTION] of restrictions: r_type;
array[RESTRICTION] of floatplus: r_start;
array[RESTRICTION] of floatplus: r_end;
array[ACTIVITY] of var floatplus: pause;
array[RESTRICTION, ACTIVITY] of var bool: intercepting;
constraint forall(i in ACTIVITY, r in RESTRICTION)
  (intercepting[r,i] =
    (r_type[r] in restricted_by[i] /\ end[i] > r_start[r] /\ r_end[r] >= start[i]));
constraint forall(i in ACTIVITY) (pause[i] >= sum(
  [if (intercepting[r, i])
   then (r_end[r] - r_start[r])
   else (0)
   endif | r in RESTRICTION]));
constraint forall(i in ACTIVITY) (end[i] >= start[i] + duration[i] + pause[i]);

solve minimize max(end) * n * 1000 + sum(pause);
```

B. Implemented GA strategies

Tab. B.1.: Overview of Implemented Strategies in Genetic Algorithms

Termination	Max Generation	Terminate after reaching a set number of generations
	Timeout	Terminate after a specific time duration
	Min Fitness	Terminate when a fitness threshold is reached
	x-Fitness Stagnation	Terminate if no improvement in fitness for x generations
Selection	n-Elite	Select the n best individuals based on fitness, defer the rest to a sub-strategy
	Random	Select individuals uniformly
	Roulette Wheel	Probabilistic selection proportional to fitness
	Rank Based	Selection probability scaled by fitness rank
Mutation	Uniform Random	Mutate each gene independently with a set probability
	Inorder Random	Mutate genes between two random indices
	Scaled Uniform Random	Mutate each gene independently with scaling
	Scaled Inorder Random	Mutate and scale genes between indices
	Shifting Mutation	Shift value based on sampling a normal distribution
Reproduction	Uniform Crossover	Swap independently chosen genes from two parents
	One Point Crossover	Split genomes at a point and recombine segments
	Two Point Crossover	Split at two points and recombine segments
	Differential Reproduction	Create offspring using scaled difference vectors
	Idempotent Repair	No modification, placeholder for potential repair
	Length Repair	Fill genome with randomized genes

C. Evaluation Tables

Statistical evaluations and tables have been created with JASP [10].
Some tables have been manually reformatted.

Parameter Study - CAL Encoding

Tab. C.1.: ANOVA - Improvement

Cases	Sum of Squares	df	Mean Square	F	p
Mutation Rate	0.149	3	0.050	234.135	$< .001$
Population Size	0.002	2	0.001	5.046	0.007
MR * PS	0.002	6	3.319×10^{-4}	1.561	0.154
Residuals	0.508	2388	2.126×10^{-4}		

Tab. C.2.: Post Hoc Comparisons - Mutation Rate

		Mean Difference	SE	t	p_{bonf}
1	5	0.015	8.418×10^{-4}	17.973	$< .001$
	10	0.019	8.418×10^{-4}	22.144	$< .001$
	20	0.020	8.418×10^{-4}	23.324	$< .001$
5	10	0.004	8.418×10^{-4}	4.171	$< .001$
	20	0.005	8.418×10^{-4}	5.351	$< .001$
10	20	9.933×10^{-4}	8.418×10^{-4}	1.180	1.000

Tab. C.3.: Post Hoc Comparisons - Population Size

		Mean Difference	SE	t	p_{bonf}
10	25	0.001	7.290×10^{-4}	1.813	0.210
	50	0.002	7.290×10^{-4}	3.166	0.005
25	50	9.865×10^{-4}	7.290×10^{-4}	1.353	0.528

Parameter Study - INT Encoding

Tab. C.4.: ANOVA - Improvement

Cases	Sum of Squares	df	Mean Square	F	p
Mutation Rate	0.187	3	0.062	72.056	$< .001$
Population Size	0.007	2	0.003	3.866	0.021
MR * PS	0.004	6	5.955×10^{-4}	0.689	0.658
Residuals	2.064	2388	8.642×10^{-4}		

Tab. C.5.: Post Hoc Comparisons - Mutation Rate

		Mean Difference	SE	t	p_{bonf}
1	5	−0.009	0.002	−5.286	$< .001$
	10	−0.017	0.002	−10.279	$< .001$
	20	−0.023	0.002	−13.770	$< .001$
5	10	−0.008	0.002	−4.994	$< .001$
	20	−0.014	0.002	−8.485	$< .001$
10	20	−0.006	0.002	−3.491	0.003

Tab. C.6.: Post Hoc Comparisons - Population Size

		Mean Difference	SE	t	p_{bonf}
10	25	−0.003	0.001	−1.778	0.227
	50	−0.004	0.001	−2.741	0.019
25	50	−0.001	0.001	−0.963	1.000

Encoding Comparison

Tab. C.7.: Frequencies for Any Improvement
All Param means the subset of results of the parameter study with population size of 25
Best Param means the subset of runs of the parameter study with population size of 25 and with mutation rate 0.01 (CAL) and 0.2 (INT)
MIB+MUG means the results of our second experiment

		All Param		Best Param		MIB+MUG	
Encoding	Any Improvement	Frequency	Percent	Frequency	Percent	Frequency	Percent
CAL	FALSE	608	76.0	70	35.0	22	11.0
	TRUE	192	24.0	130	65.0	178	89.0
INT	FALSE	412	51.5	67	33.5	12	6.0
	TRUE	388	48.5	133	66.5	188	94.0
	Total	800	100.0	200	100.0	200	100.0

Tab. C.8.: Descriptive Statistics of Cost Improvement
All Param means the subset of results of the parameter study with population size of 25
Best Param means the subset of runs of the parameter study with population size of 25 and with mutation rate 0.01 (CAL) and 0.2 (INT)
MIB+MUG means the results of our second experiment

	All Param		Best Param		MIB+MUG	
	CAL	INT	CAL	INT	CAL	INT
Median	0.000	0.000	0.013	0.032	0.077	0.128
Mean	0.006	0.023	0.020	0.034	0.089	0.163
Std. Deviation	0.016	0.031	0.026	0.035	0.071	0.119

Hardness Factor Analysis

Tab. C.9.: Pearson's Correlations

			Pearson's r	p
Cost Improvement	-	Activity Count	-0.084	$< .001$
	-	Max Project Days	-0.084	$< .001$
	-	Total Processing Duration	-0.129	$< .001$
	-	Interruptible Percent	0.160	$< .001$

Tab. C.10.: Regression Model of Cost Improvement - Summary, ANOVA and Coefficients

Model	R	R^2	Adjusted R^2	RMSE
H_0	0.000	0.000	0.000	0.126
H_1	0.213	0.045	0.045	0.123

Model		Sum of Squares	df	Mean Square	F	p
H_1	Regression	5.213	4	1.303	85.957	$< .001$
	Residual	109.689	7235	0.015		
	Total	114.902	7239			

Model		Unstandardized	Standard Error	Standardized	t	p
H_0	(Intercept)	0.104	0.001		70.438	$< .001$
H_1	(Intercept)	0.088	0.003		34.669	$< .001$
	Activity Count	9.014×10^{-7}	2.009×10^{-7}	0.068	4.487	$< .001$
	Max Project Days	-6.315×10^{-6}	1.567×10^{-6}	-0.052	-4.030	$< .001$
	Total Processing Duration	-1.350×10^{-8}	1.354×10^{-9}	-0.143	-9.971	$< .001$
	Interruptible Percent	5.022×10^{-4}	3.492×10^{-5}	0.176	14.382	$< .001$