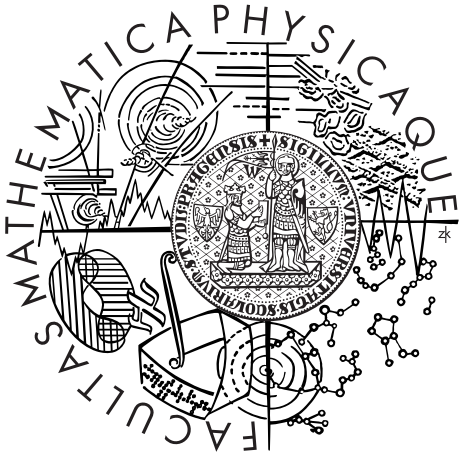
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Miroslav Štola

Combination of Evolutionary Algorithms and Constraint Programming for Scheduling

Department of Theoretical Computer Science
and Mathematical Logic

Supervisor of the master thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science
Study branch: Theoretical Computer Science

Prague 2015

I would like to thank Martin Pilát for supervising this thesis. I found his comments and consultations enlightening. His insights helped me better orient myself in the field of evolutionary algorithms, especially in the multi-objective optimization. I also appreciate his swift reaction times and willingness to spend time on my thesis even on the weekends.

I would also like to thank Jan Gregor who introduced me to the project and also helped me realize the deadline was drawing near.

Finally, big thanks belong to my family who supported me throughout my studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Kombinace evolučních algoritmů a programování s omezujícími podmínkami pro rozvrhování

Autor: Miroslav Štola

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Martin Pilát, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Je známé, že rozvrhovací problémy a problémy splňování podmínek jsou velice těžké. Tato práce nabízí nový přístup, jak pomocí evolučních algoritmů řešit rozvrhování s omezujícími podmínkami. Evoluce probíhá na pořadí proměnných, které solver postupně ohodnocuje. Tento přístup umožňuje jedince zakódovat jako permutace, a tedy je použitelný na širší škálu problémů s omezujícími podmínkami. Na základě analýzy grafu závislostí byly navrženy metody pro inicializaci počáteční populace jedinců. Rovněž byly vymyšleny a úspěšně použity nové genetické operátory. Naše metoda nalezla mnoho rozličných rozvrhů s optimální délkou. Dále byla úspěšně vyzkoušena vícekriteriální optimalizace za pomoci algoritmu NSGA-II.

Klíčová slova: evoluční algoritmus, rozvrhování, CSP, pořadí proměnných

Title: Combination of Evolutionary Algorithms and Constraint Programming for Scheduling

Author: Miroslav Štola

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Scheduling problems and constraint satisfaction problems are generally known to be extremely hard. This thesis proposes a new evolutionary algorithm approach to solve a constrained-based scheduling problem. In this approach, variable orderings are evolved. The variable ordering serves as a parameter for the constraint solver. Its purpose is to determine the order in which variables are labelled by the solver. Hence the evolving individuals may be encoded as permutations. Therefore, our approach can be applied to a wider range of constraint satisfaction problems. Methods for generating the initial population of individuals based on the analysis of the precedence constraints graph are proposed. New genetic operators are presented and successfully applied. Our approach succeeded in finding a range of diverse schedules with the optimal makespan. Furthermore, multi-objective optimization was successfully attempted with the NSGA-II.

Keywords: evolutionary algorithm, scheduling, CSP, variable ordering

Contents

Introduction

Thesis motivation Scheduling is a decision-making problem that has impact on many disciplines. It is primarily concerned with allocation of often limited resources to activities in order to optimize one or more objectives. Resources include a wide spectrum of entities, for example workers in factories, public transport drivers, machines in an assembly plant, CPUs, etc. The activities also comprise a large variety of possibilities: executions of a computer program, assembling a part of a product and so on. There are also many potential objectives to optimize, from minimizing the length of the schedule to minimizing the number of delayed activities. Scheduling has become omnipresent, since its beginnings in the 20^{th} century. It is an extremely hard class of problems. Even for small projects, the number of possible allocations of resources is too great to be fully inspected. It is therefore clear that heuristics and approximations are needed.

One of the techniques applied on a scheduling problem is encoding it as a constraint satisfaction problem and then obtain a solution by running a constraint solver. Constraint-based scheduling is also an extremely difficult class of problems. Often, there is also vast space to search. The outcome of the solution space search is highly dependent on the order in which the solver chooses variables to label. This thesis focuses on finding a good *variable ordering* for the constraint solver which in turn produces a schedule as a solution.

Evolutionary algorithms belong to the class of popular heuristics for solving difficult optimization problems. This thesis focuses on combining the evolutionary algorithms and constraint programming as a means of solving the scheduling problem. The main objective is to evolve the *variable orderings* for the constraint solver. Both single and multi-objective evolutionary algorithms were tried. Several new genetic operators and methods of population initialization are presented. Multiple combinations of operators and initialization methods are experimentally tested and the results discussed.

Goals

Constraint satisfaction and evolution is combined in our approach. The constraint solver, which the evolution uses as a black box, optimizes the minimal makespan criterion. We aim to improve the solutions obtained from the solver by evolving the variable orderings. We strive not only to find good solutions, but to find as many as possible. We set the following goals:

1. to propose a method for initialization of the population of the variable orderings,
2. to propose appropriate genetic operators,
3. to attempt a multi-objective optimization in order to obtain higher number of good solutions and to optimize by other criteria as well.

Document Outline

This thesis is thematically divided into several chapters.

Preliminaries The first chapter includes a description of the project which this thesis aims to enhance. It also brings forth an introduction to multi-agent systems, constraint programming and scheduling as they are all used in the project.

Related work Evolutionary algorithms are introduced in the second chapter. Additionally, various approaches to evolutionary scheduling are described. An overview of different representations as well as genetic operators is offered.

Problem analysis The third chapter is dedicated to the description of the problem at hand. The importance of good inputs for the constrained solver is emphasised and backed up experimentally. Finally, the size of the search space is estimated.

Evolution The fourth chapter focuses on the evolutionary algorithms, in particular original methods of initialization and genetic operators invented for this thesis.

Results The fifth chapter provides the results of our approach to combine evolution and constraint programming to solve the scheduling problem. Genetic operators and initialization methods are compared and results discussed.

Conclusion The thesis is summarised in the concluding chapter. Each of the goals is evaluated and discussed individually.

Future work The last chapter offers improvements that are yet to be implemented, should the project continue.

1. Preliminaries

In this section, we shall take a look at the description of the project which this thesis is trying to enhance. Next, multi-agent systems (MAS), constraint programming (CP) and scheduling are briefly introduced. Finally, the role of this thesis in the project is sketched.

1.1 Project Overview

This thesis is a part of the ARUM[2] project which is going to be described in this section. The goal of the project is to create multiple schedules meeting various criteria for an assembly line. It is achieved primarily by combining a multi-agent system and constraint programming. This section was primarily based on an internal document of CertiCon[12], one of the contributors to the ARUM project.

1.1.1 Use Case Description

The problem at hand is inspired by a real-life production line whose purpose is to assemble planes. The line is divided into several units called workstations. Work at each station should be scheduled. Every workstation has its manager and workers possessing skills required by certain tasks.

Assembly line The Assembly line consists of workstations connected in series. The workstations are occupied by plane sections in various states of completeness. A plane section cannot be moved to the next workstation until all tasks are done on the workstation in question. Whenever all stations complete their tasks, all plane sections shift to the next station at the same time. The time this occurs is called a *cycle time*.

Main goal The main goal of the whole project is to provide schedules that are near-optimal in some criteria. One of those criteria is the ability to recalculate the schedule in case some undesirable events occur. Others are more obvious, i.e. the end time of the last task on given station.

Possible challenges There are many potential challenges that have to be tackled. Since the problem is inspired by a real-world problem, the schedules must be flexible in anticipating certain events like delayed delivery of assembly parts, machine malfunctions, workers unavailability etc. The assembly line is by nature a dynamic environment and the necessity to recalculate the schedules on-the-fly may arise. Also, in order to ensure flexibility and variety, several different schedules have to be calculated. A responsible entity called the *station manager* will decide which schedule is chosen. Another thing that makes the problem even more challenging is the fact that the task assignment is not fixed to a particular station, neither are the workers. It is the responsibility of the scheduler to allocate or migrate them among stations. Furthermore, duration of the stations is not fixed, it may be extended or shortened in the process of recalculation.

Objectives The primary objective is to finish work on all stations in time. The quality of a solution may be measured e.g. by the length of the schedule (difference between the start time of the first task and the finish time of the last task). Due to possible disturbances, measures must be taken to ensure robustness of the schedules. When the primary objective is met, other objectives may be taken into account.

Scheduler input The scheduler expects several necessary parameters as input. First of all it needs a set of tasks to be scheduled with initial assignment to stations. Available resources and stations configuration (start of shift, duration, etc.) are also needed. The criterion to optimize also has to be specified for the solver. Finally, a strategy how to approach tasks that are scheduled beyond station's time span has to be defined.

Problem solving is divided between a multi-agent system (MAS) and a constraint solver. Each station is scheduled by the solver. The agents provide communication among the stations. They try to improve the global solution by moving tasks between stations.

Decomposition Although the global schedule may be calculated, it is more appropriate to decompose the solution to individual stations. This approach has several advantages. This design fits the enterprise's current solution. Each station is supervised by a manager who chooses suitable schedule to follow from a precalculated set of schedules. It also enables parallel computation of schedules, provided that there are no dependencies among the stations. Another advantage is the ability to resolve the disruptive events at station's level, so it does not affect the other stations.

Solving a station Each station is represented by an agent. The agent asks the constraint solver for a schedule of the current station. The agent must provide a criterion to optimize. There are three possible outcomes of running the solver: a solution is found and all tasks are assigned within the station's time boundaries. A solution is found, yet some tasks are scheduled after the station's end time. In this case, agents take over and negotiate tasks among stations. In the third case no solution is found. It may be due to unavailability of some resources or other factors. Again, agents take over and try to fix the schedule by negotiating between stations.

Work division As was mentioned before, the main workload is divided between two parts: a MAS and a constraint solver. The multi-agent system has the following functionality:

- Business strategies if task does not fit to the assigned station:
  - STOP & FIX - the station's duration is extended accordingly.
  - Travelling work - the task is moved to another station.
- Moving workers among the stations.

- Moving tasks among the stations.

The constraint solver, on the other hand, primarily focuses on finding solutions to the scheduling problem. It supports the following:

- Temporal relations - i.e. tasks precedence. Task precedence determines a relation among tasks. It dictates that, for example, task t_1 cannot start until task t_2 is finished. Temporal relations are defined further in 1.4.6.
- Non-consumable resources - used to model e.g. the production space or tools.
- Consumable resource (materials) - tasks that need certain consumable resource have to be scheduled after the delivery time of the materials.
- Human resources possessing certain skills. A human resource can be allocated to only one task at a time.
- Station bound and station unbound tasks. The bound tasks cannot be executed anywhere but the station they are bound to. These tasks have generally higher priority and should therefore be preferred to others in case the scheduler fails to schedule all tasks.
- Minimal change - when placing a task in a schedule, the constraint solver prefers such values from its domain that differ as little as possible from the same task in an already computed schedule.

1.1.2 Architecture

Scheduler The scheduler contains various agents using the JADE platform[13]. The agents need not run on a single machine, thus enabling parallelization. Figure 1.1 depicts the types of agents which are listed below:

- Scheduler agent - a top level agent, it deals with the communication between the system and the outer world. It receives requests for schedules and returns calculated schedules via an ontology service.
- Solution agent - its responsibility is to create a global schedule over all stations. It is supplied with a promising schedule by the scheduler agent for each station and then combines them to a single one.
- Station agent - represents station at a cycle time
- Main solver agent - an agent that keeps the old schedules as well as calculates new ones. It can also assign the schedule computation to another solver agent on another station
- **Solver agent** - runs the actual schedule computation. A part of this agent is dealt with in this thesis.

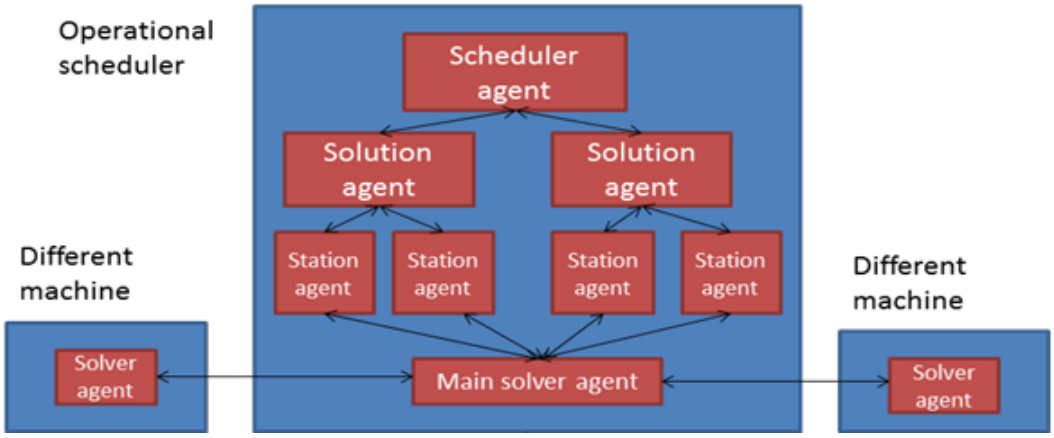


Figure 1.1: Architecture of the scheduler

1.1.3 KPI

A KPI (Key Performance Indicator)[14] is a mechanism to measure the quality of a solution. In other words, a KPI is a criterion to optimize. It is up to the user to define single or multiple KPIs. The scheduler expects a set of KPIs (or a single one) as a part of its input. After schedules are computed, a responsible entity (i.e. the station manager) chooses one schedule according to the selected KPIs.

KPI division KPIs can be divided into local and global. Local KPI serves as input for the constraint solver. Global KPI is used to measure the overall quality of the assembly line performance (e.g. number of manufactured products per month). Alternatively, the KPIs can be viewed as either the performance indicators (e.g. the shortest possible schedule) or robustness indicator (e.g. schedule's ability to recover from disruptions).

The local KPIs are used to measure the quality of a solution on a single station. List of potential KPIs follows.

- Makespan - the finish time of the last task on the station's schedule

$$\text{objective} : \text{minimize}(\max\{\, e(t) \mid t \in Tasks\})$$

  In this scenario, a free interval can come into existence between the schedule finish time and the time the station has to finish. It can be used either to reduce the cycle time of the station or to move tasks from other stations to this free interval. When a solution with sufficiently good makespan is found, the search for alternative schedules (e.g. focusing on robustness) may commence.

- Tasks ASAP - every task should end as early as possible. This is done in order to preserve some space at the finish time of the station.

$$\text{objective} : \text{minimize}(\sum_{t \in Tasks} \text{penalty}(t))$$

The penalty of a task is computed based on its distance from the station start time. This KPI can be reversed if free interval is desired at the beginning of the station. In such case, maximization instead of minimization is used.

- Number of overlapping tasks - number of tasks per an arbitrary time interval. It aims to schedule the same amount of tasks in each interval if possible. This can lead to more flexible rescheduling - it can produce schedules similar to the original which means less effort for the recalculation. This KPI can also cause a more evenly distributed worker utilization.

- Minimal change KPI - in case a schedule is disrupted, new schedule needs to be calculated. This KPI tries to ensure similarity (whenever possible) between the old schedule and the new one. Examples: task count for each station in the schedule, start times of all tasks at all stations and end times of all tasks at all stations.

- Worker utilization - uniform utilization of workers is preferred. This aspect can be effectively measured after a solution is found. It is influenced by the variable ordering used by the constraint solver. It can also be optimized when all tasks in the solver are placed and only the worker assignment to tasks remains to be determined.

- Robustness to missing material - if there is uncertainty about delivery of some material, a schedule that needs the material later may be preferable.

The global KPIs include maximization of the number of manufactured products per time interval, utilization of workers, the fabrication cost of a single product and so on.

1.1.4 Constraint Solver Description

The solver agents provide the station agent with feasible schedules respecting the agent's preferences. The solver agent has many parts, we shall state the most important ones. The open source Choco solver[15] was used as the constraint solver. Java objects representing the scheduling essentials: tasks, workers, resources etc. are also a part of the agent. Each task t is defined by three variables: start time $s(t)$, end time $e(t)$ and duration $p(t)$. Some tasks require resources. To prevent assigning more resources than available, a cumulative constraint over all tasks was added to the model. This constraint states that the number of tasks, using a resource r, may not exceed the capacity of r at any time. The domains of CSP variables consist solely from integer values in this project. Another necessary item is the Choco3 model which is used to map java objects to Choco entities (variables, constraints). Finally, there is the *variable ordering* that specifies the order in which the variables are labelled by the solver.

Solver workflow The process of calculating a schedule consists of the following steps:

1. Information about tasks, their resources demands and their dependencies is received. This information is used to create the corresponding java objects.
2. A constraint model is constructed using the java objects.
3. The solver receives a variable ordering.
4. According to the chosen objective variable and the variable ordering, the solver tries to find assignment for all the variables that satisfy all constraints and to optimize the objective.
5. When a solution is found, the java objects are recreated from the constraint model.

1.1.5 Dealing with Disruptions

The events that corrupt the currently executed schedule are dealt with by the ontology service. As time progresses, the schedule is damaged further by the undesired disruptive events. The need of rescheduling arises. The tasks that can be executed or at least have a chance of being executed in the future, are fed to the operational scheduler. New schedule is created as soon as possible and applied to the work station.

Missing material If a task needs an unavailable material, it has to be moved to the point when the material is expected to arrive. Naturally, every dependent task has to be moved accordingly so that the schedule remains admissible.

Prolonged task duration If a task exceeds its scheduled duration, all dependent tasks may have to be moved in order to satisfy the constraints on them.

1.2 Introduction to MAS

This section brings a brief overview of the multi-agent systems. Firstly, an agent is defined, followed by MAS definition. Finally, agent communication is described. A multi-agent system is a part of the project. There are several types of agents and their responsibilities vary from handling requests for schedules to negotiating tasks among the stations. For more thorough description, please refer to 1.1.2.

1.2.1 Agent Definition

There are many definitions as to what an agent is. In essence, an agent is a computer system that perceives its environment and is capable of autonomous actions affecting its environment. Other definitions are offered by Wooldridge[10] or Russel and Norvig[9] and many more. The need of defining properties of an intelligent agent arises. An intelligent agent should manifest *reactivity*, meaning it should perceive the environment and be capable to react to its changes. It should also be *proactive*, to be capable of performing actions leading to achieving its goals. Lastly, an intelligent agent should exhibit a certain level of *sociality*, which means it should be capable of communication with other agents. Creating

a purely reactive or proactive agent is not difficult, however, finding balance between the two may prove challenging.

An agent can exist in various types of environments. They are divided into several categories. *Fully observable* environment is visible by the agent at all times. On the other hand, an environment can also be *partially observable*. Environment that can be changed solely by actions of an agent is called *static*, otherwise it is said to be *dynamic*. The environment is referred to as *discrete* if the number of percepts and actions performed by an agent are limited and *continuous* otherwise. If each of agent's actions has only one outcome, the environment is *deterministic*. If an action can have several outcomes, then the environment is *non-deterministic*.

1.2.2 MAS Definition

A multi-agent system is a system consisting of agents that interact with each other. As Wooldridge[10] states: "To successfully interact, they will require the ability to cooperate, coordinate, and negotiate with each other, much as people do."A MAS is more than a model for distributed computing, since agents are autonomous and independent and the means of coordination are not set beforehand. A MAS focuses mainly on these issues[10]:

- Which languages do agents use to communicate with one another?
- How can cooperation emerge in societies of self-interested agents?
- How can agents with self-interest reach an agreement?
- How can autonomous agents coordinate their actions to cooperatively reach their goals?

These issues are addressed by other fields as well (economics, sociology), however only in MAS the agents are computational, information processing entities.

1.2.3 Agent Communication

In the OOP approach, communication means invoking methods with parameters. In contrast, agents cannot force other agents to do something in the same way. Agents have to communicate in order to request some action or trade information with other agents. It is achieved by sending messages. Once a message is received, it is up to the recipient agent how to deal with the message, e.g. deny the request, make a counter offer etc.

In the project, the communication among agents is handled primarily by ontology services. In 1993, Gruber provided the following definition of ontology[11].

Definition 1.1. *"An ontology is a formal, explicit specification of a shared conceptualization."*

Ontology in computer science is formal, declarative representation which contains glossary (definitions of terms) and thesaurus (definitions of relations between terms). Ontology is a dictionary that serves to maintain and pass on knowledge concerning certain problems. Ontology allows abstraction away from the underlying data structures and implementation.

1.3 Introduction to Scheduling

According to Pinedo[1]: "Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives."

Scheduling problem We shall start by an informal description of a scheduling problem. Each scheduling problem is defined by the following features: a set of *tasks* (or *actions*) that need to be carried out, a set of *resources* (machines, human resources, materials) and their availability in time, *resource demands* for tasks, *constraints* (e.g. precedence between tasks) and one or more *objective* to optimize. Some instances of the scheduling problem also define cost functions (for example the cost of machine reconfiguration). The majority of scheduling problems are NP-hard[17].

Every task is identified by several traits. We have already stated the resource requirements. Another important feature is the start time $s(t) \in [s_{min}(t), s_{max}(t)]$ and the end time $e(t) \in [e_{min}(t), e_{max}(t)]$. There can also be an optional recommended end time $\delta(t)$ which may be different from the deadline $e_{max}(t)$. Finally, there is the duration $p(t)$ which is typically a constant. Tasks may be divided into *non-preemptive* and *preemptive*. A non-preemptive task cannot be interrupted during the execution: $p(t) = e(t) - s(t)$. On the other hand, it is possible to interrupt a preemptive task and then resume its execution: $p(t) = \sum_i p_i \leq e(t) - s(t)$.

Another important feature of a scheduling problem is the objective. The most commonly used are the makespan $C = \max\{\, e(t) \mid t \in Tasks\}$ and the lateness $L = e(t) - e_{max}(t)$.

1.3.1 Types of Scheduling Problems

There are several ways to classify scheduling problems. We will briefly describe machine scheduling. It is further divided by complexity of jobs (explained below).

Machine scheduling Machine scheduling is concerned with allocating machines to given jobs. A job is a partially ordered set of tasks, the tasks in different jobs are independent. Tasks of the same job must not overlap. The resulting schedule consists of allocated machine for each task and the time interval in which the machine is to carry out the task in question.

Single-stage machine scheduling In the case of the single-stage machine scheduling, every job is a single task which can be processed on any machine. There are several types of parallel machines:

- Identical - all of the machines are equivalent.
- Uniform - every machine has its speed coefficient. However, the ratio of tasks execution time remains constant. For example, if task A takes twice as long as task B on machine i, it also takes twice as long on machine j.

- Unrelated - possibly variable execution times for all task combinations. For example, task A is processed faster than task B on machine i, but slower on machine j.

Multiple-stage machine scheduling In the multiple-stage machine scheduling scenario a job consists of several tasks, each of which may require different machine allocation. The most common problem types include:

- Flow-shop - all tasks have to be processed in one specific order. The machines are usually connected in series, hence the name. A typical example is an assembly line.
- Open-shop - similar to flow-shop, but the tasks are not linearly ordered.
- Job-shop - more general case, tasks within the job are linearly ordered. Different jobs may have different tasks with different orders.

Resource-constrained project scheduling problem A resource-constrained project scheduling problem (RCPSP) contains tasks with known durations, resource demands and the precedence constraints. The objective is to assign start time to each task, such that no precedence and resource constraints are violated. More formal definition is given in Chapter 2. A scheduling problem of this type is also tackled by this thesis.

1.4 Introduction to Constraint Programming

Constraint programming is a part of declarative or non-procedural programming. Instead of following instructions sequentially as it is done in procedural programming, constraint programming relies on declarative description of a problem and solving it by CP techniques (search, constraint propagation, etc.).

1.4.1 Constraint Satisfaction Problem

A constraint satisfaction problem is defined by a triple (X, D, C)[9], where $X = \{X_1, ..., X_n\}$ is a finite set of variables, $D = \{D_1, ..., D_n\}$ is a set of the domains of the variables and $C = \{C_1, ..., C_m\}$ is a set of constraints over variables. Each variable X_i can be assigned the values from its nonempty domain D_i. Every constraint $C_j \in C$ is a pair $\langle Y, R_j \rangle$, where $Y \subseteq X$ such that $|Y| = k$ and R_j is a k-ary relation among the variables of Y. Constraints can be defined both extensionally (by enumeration of compatible tuples) or by a formula. A popular game Sudoku is one example of a CSP. Every cell's domain has values $\{1, ..., 9\}$ with the constraints that the cells in the same row, column and 3×3 square cannot share the same values.

Solving a CSP consists mainly of assigning values to variables. We say that a solution to a CSP is *feasible* if every variable is assigned a value from its domain and all the constraints are satisfied. We call it *optimal* if it is feasible and optimizes some objective function of the problem.

1.4.2 Maintaining Consistency

When trying to solve a CSP problem, it is desirable to filter out the domains of the variables in order to shrink the search space. We shall see how constraints can be used to rule out from domains such values that would only lead to inconsistent assignments. It is highly advisable as it may dramatically reduce traversing dead-end search branches. All described consistency techniques[7] are performed on graphs where nodes represent variables and edges connect variables that share a constraint.

The simplest type of consistency is called *node consistency*. First of all, unary constraints are converted into variables' domains.

Definition 1.2. *Node Consistency*
The vertex representing variable X is node consistent if and only if every value in the variable's domain D_x satisfies all the unary constraints imposed on the variable X.

CSP is node consistent if and only if all the vertices are node consistent.

Arc consistency Since all CSP problems may be converted into binary ones (all constraints are binary), we will assume that we only deal with those.

Definition 1.3. *Arc Consistency*
The arc (V_i, V_j) is arc consistent if and only if for each value x from the domain D_i there is a value y in the domain D_j that the assignment $V_i = x$ and $V_j = y$ satisfies all the binary constraints of (V_i, V_j)

CSP is arc consistent if and only if every arc (V_i, V_j) is arc consistent (in both directions).

Arc-B consistency There are special cases of CSPs called numeric CSPs where the domain of each variable is represented only as the smallest value $lb(V_i)$ and the largest value $ub(V_i)$. The domain of the variable V_i is the whole interval $[lb(V_i), ub(V_i)]$. This representation is very compact and can therefore be applied to problems where maintaining the whole set of values explicitly might be impossible. Constraints are propagated by Arc-B Consistency[8] techniques.

Definition 1.4. *Arc-B Consistency*
The arc (V_i, V_j) is arc consistent if and only if for each of the bound values $x = lb(V_i)$ and $x = ub(V_i)$ from the domain D_i there exists a value y in the domain D_j that the assignment $V_i = x$ and $V_j = y$ satisfies all the binary constraints of (V_i, V_j)

Making CSP arc consistent There are many algorithms for arc consistency. We shall not describe them here, keen readers are encouraged to learn more in other publications[7]. Basically, every arc in the graph needs to be revised. In revision of arc (V_i, V_j), every value x from D_i, that does not have at least one y in D_j satisfying all the constraints on V_i and V_j , is removed from D_i. It should be noted that revising all the edges is not sufficient. It is due to the fact that revising an edge may influence already revised edges.

Other consistency techniques There are more consistency techniques, several of which quite sophisticated. Each has some merits (strength, efficiency) and some drawbacks (memory consumption). For the sake of completeness, let us name some of them: directed arc consistency, path consistency, restricted path consistency, k-consistency, strong k-consistency, neighbourhood inverse consistency, inverse consistency etc. These are described in detail by Dechter[7].

1.4.3 Solving CSP

CSP can be solved in two traditional manners. On one hand, there is search which explores the whole space. Therefore, it either finds a solution or proves it cannot be found. Obviously, it can work quite slowly and furthermore it continues searching in directions that clearly cannot lead to finding a solution (a constraint was violated). On the other hand, we have consistency techniques described above. They are not complete, but they reduce the size of the search space and are quite efficient. The best way to solve a CSP is to combine both approaches[9]. It is achieved by assigning values to variables via backtracking and consistency is checked after each assignment.

Definition 1.5. *Labelling*
Assigning a value to a variable is called labelling.

Look-back The look-back technique ensures that the already labelled variables are consistent with all the constraints. It looks back at the instantiated variables and looks for a conflict and its source. Algorithms in this category are e.g. backtracking or backjumping.

Constraint propagation Checking if labelled variables do not violate any constraints is useful, but it is also possible to propagate the constraints to variables that have not been checked yet. It is called the constraint propagation. It enables us to prevent failure of assignments in the future.

Forward-checking The forward-checking[3] technique checks the immediate neighbourhood of a variable. After a variable X is labelled with value from D_X, all variables Y that share a constraint with X are checked. Every value from D_Y that is inconsistent with any constraint shared with X is removed from D_Y for each Y. As soon as any domain of an unassigned variable becomes empty, backtracking is in order.

Look-ahead As opposed to forward-checking, the *partial look-ahead*[4] performs consistency checks on all of the future variables. The *full look-ahead*[5][6] technique performs a full arc-consistency check. It is common to use even stronger consistency algorithms and run them only once before starting the search.

1.4.4 Variable Ordering

An other key element in solving CSP is called *variable ordering*[4]. It is a guidance how to choose the order of variables for labelling. It can have a significant impact

on the constraint solver efficiency. There are two categories of variable orderings: static and dynamic.

Static In the static case the order of variables is specified in advance and is not changed later. The solver can assign a variable if all variables that precede it in the order are already assigned. This technique is used in this thesis and the order is obtained by means of evolutionary algorithms. We chose this approach, because it enables us to efficiently encode the individuals as permutations.

Dynamic In the dynamic case the decision which variable is to be labelled next is not fixed, but can be dependent on the current state of search (i.e. already assigned variables, domain size). Dealing with the hard cases first is generally considered a good idea, if the current assignment will lead to failure the sooner it is found, the better. One of the most common principles is called *fail-first*. In this case, a variable with the fewest values left in its domain is chosen for labelling. In other words, the one most likely to fail. It might be counter-intuitive, because we do not want the search to fail, but it is generally better to fail early if the failure is inevitable. Another popular strategy is to choose the variable that participates in most constraints.

Value ordering Determining the order in which values are assigned has the same level of importance as the ordering of variables. One of the representatives is called *succeed-first*. It prefers values that will less likely cause failure. Value ordering will not be described further as it is not explored in this thesis.

1.4.5 Constraint Optimization

Up until now, we have dealt solely with satisfying constraints. However, there may be many solutions and we might be interested in the optimal one.

Definition 1.6. *Constraint Satisfaction Optimization Problem*
CSOP is defined by a CSP and an objective function that maps solutions of the CSP to real numbers. A solution of CSOP optimizes the objective function. In order to be able to solve a CSOP, a mechanism for obtaining multiple solutions is needed.

Now, we shall take a look at a search algorithm which can also be applied to the constraint satisfaction optimization problem. It is called *branch and bound*.

Branch and bound The branch and bound method is used in search to ignore the branches where there is no optimal solution. It uses a heuristic function h that estimates values of the objective function f. Admissible heuristic function for minimization must satisfy $h(x) \leq f(x)$. The more precise the approximation of the objective function, the better. When traversing a search tree, the current branch can be cut off if there is no solution in the subtree or the current best solution cannot be improved by any solution in the subtree, meaning that the bound is better than value of the heuristic function for the subtree. The bound may be the objective value of the best solution so far, for example. In CSOP, the

objective function is encoded in the constraints. The first solution is found as in the regular CSP. Then a constraint $f(x) < Bound$ that ensures the next solution must be better than the current best is added. This process is repeated until no more feasible solutions are found.

1.4.6 Constrained-Based Scheduling

Since scheduling is a static problem (all tasks, domains etc. are known), it can be encoded as a CSP problem. Constrained-based scheduling explores how to solve scheduling problems using CP. We will show how to model a scheduling problem as a CSOP. We will consider a non-preemptive scheduling problem.

Encoding the problem There are three variables introduced for each task: $s(t), e(t), p(t)$ (see 1.3). Additionally, each task t_i has a lower bound (release date $e_{min}(t)$) and upper bound (deadline $e_{max}(t)$). Together they define a time window in which the task has to execute. Linear constraints may be used to model the *temporal relations* between tasks. There are several types of possible constraints: task t_i does not start later than task t_j ($s(t_i) \leq s(t_j)$), t_j does not end before the start of t_i ($s(t_i) \leq e(t_j)$), t_j does not start before the end of t_i ($e(t_i) \leq s(t_j)$) and t_j does not end before t_i ($e(t_i) \leq e(t_j)$). These constraints are propagated using arc-B consistency algorithms or some other techniques.

Resources Tasks may require some resources for their execution. These requirements are modelled by resource constraints. The resources can be divided into three categories: the *unary* resources which can be used only by one task at a time, the *cumulative* which can be used by several tasks at once, provided that their count does not exceed the resource capacity and finally the *consumable* resource. A task may either consume or produce a consumable resource.

1.5 Role of this thesis

This thesis aims to provide multiple near-optimal solutions to a single workstation. The station is scheduled by the constraint solver, which also needs the order of tasks (**variable ordering**) as input. The order is found by means of artificial evolution. In this thesis, the constraint solver is only used as a black-box in order to obtain schedules. Therefore, the focus is solely on the evolution and its capability to find variable orderings leading to high-quality schedules. Details of our approach are described in next chapters, mainly in Chapter 4.

2. Related work

This chapter starts by describing evolutionary algorithms, genetic operators and multi-objective evolution. Next, other work that addresses the scheduling problem using genetic algorithms or constraint programming is described. However, no articles that engage the problem by combining both approaches in manner similar to ours were found. Various representations of individuals and the genetic operators that are frequently used may be found here.

2.1 Evolutionary Algorithms

Before we focus on methods often used to solve the scheduling problems by artificial evolution, we should describe what evolutionary algorithms (EA) are. Evolutionary algorithms are an optimization method inspired by the concept of biological evolution. A group of individuals, called a population, is evolved during the process. The EA (See Algorithm 1) runs in a loop, the population in each iteration is called a generation. An individual is typically a sequence of fixed number of genes (e.g. array of bits). Individuals encode a (possibly infeasible) solution to a given problem. The algorithm starts by generating the initial population of individuals (typically randomly). In every iteration the population undergoes a mating selection, crossover and mutation. The crossover and mutation operators create new individuals. The whole population is evaluated using a fitness function. As a result, each individual is assigned a fitness value. The individuals' fitness is used for environmental selection. The environmental selection will determine which individuals survive and go to the next generation. The more fit the individual is, the bigger chance it has to survive (survival of the fittest). However, even the unfit may get to the next generation. The algorithm terminates when a sufficiently fit individual is found or after a predefined number of iterations.

Algorithm 1 Evolutionary algorithm

```
1: P ← Initialize population
2: Evaluate(P)
3: while not termination do
4:     P' ← Crossover(P)
5:     P' ← Mutate(P')
6:     Evaluate(P')
7:     P ← Select(P ∪ P')
8: end while
```

History Evolutionary algorithms are a part of a field called *evolutionary computing*. The beginnings of evolutionary computing date back to 1960s, when three separate branches arose. Fogel presented *evolutionary programming*[46], which evolves finite automata and aims to optimize their parameters. Rechenberg and Schwefel introduced *evolution strategies*[47][48], which primarily deal with optimization problems with continuous search space. The most interesting for us are the *genetic algorithms* invented by Holland[49]. The GA were inspired

by Darwinian natural selection. The field was later broadened by Koza, who introduced *genetic programming*[50]. Essentially, programs are evolved. As opposed to evolutionary programming, the GP evolves not only parameters, but also the program's structure.

Ever since its origins, this interesting field of nature-inspired computation has been evolving rapidly. It has been a popular method for solving problems that cannot be solved by conventional complete methods. The cost of applicability is that EA do not guarantee optimal solutions.

The following paragraphs are dedicated to explaining some key terms of evolution, namely the initialization, evaluation, selection and genetic operators – crossover and mutation.

Initialization Initialization fills the population with newly created individuals. The generation of individuals typically contains randomness. There is a whole section dedicated to this subject in Chapter 4.

Evaluation Population is evaluated by evaluating each individual in turn. As a result, every individual is usually assigned fitness – a measure of quality. Most of the times, fitness is represented as a value, e.g. integer or double. However, there are cases when defining an ordering on individuals is sufficient rather than counting the exact fitness.

Selection There are two types of selection: environmental and mating. The *environmental selection* (also called survivor selection or replacement) defines how to select individuals that survive to next generations. There are several approaches, e.g. selecting from both parent and offspring generations or selecting solely from the offspring generation. If survival of the best individuals from parent generation is desired, it can be ensured by automatically copying them to the offspring generation. This is called *elitism*. The *mating selection* chooses individuals for reproduction. We will mention two of the most popular selection operators, nevertheless there are more.

The *roulette wheel selection*, also known as fitness proportionate selection, selects an individual with a chance proportionate to its fitness with respect to the whole population's fitness: $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, where p_i is the probability of individual i being chosen, f_i its fitness and N the size of the population.

In the *tournament* selection, several individuals are picked randomly and play tournament afterwards. The winner of the tournament survives and is selected. In the deterministic version, the individual with the best fitness is the winner. In the stochastic one, it wins with probability p_i. Selection pressure is easily controlled by modifying the tournament size and p_i.

Crossover and mutation Crossover is a genetic operator that takes two or more parents (obtained by the mating selection) and recombines them into one or

more offspring. The recombination occurs with probability p_C, otherwise the individuals are left unaltered. Crossover strives to conserve building blocks that are shared by fit individuals. Mutation slightly alters an individual with probability p_M, otherwise it does nothing. Mutation's role is the maintenance of diversity. Particular implementations of both crossover and mutation operators will be described later.

2.1.1 Genetic Operators For Permutations

This section provides an overview of the genetic operators that were designed to operate on permutations. The reason we focus on the permutation encoding is that the individuals are encoded in the same way in this thesis. Not all of the relevant operators are described here, some are added later in the section about evolutionary scheduling along with their application to scheduling.

Crossover Operators

The following paragraphs are dedicated to description and examples of the crossover operators traditionally used to operate on permutations. All stated crossover operators need two parents and produce two offspring. We will denote the parents as p_1 and p_2 and the offspring as o_1 and o_2 in every method.

Partially mapped crossover The partially mapped crossover (PMX)[52] strives to keep the most possible elements at their original positions. The procedure is described in Algorithm 2. We will show creation of only one offspring. The process of making the other is symmetrical.

Algorithm 2 Partially mapped crossover

1: a, $b \leftarrow$ random crossover points. Let us call the half closed interval $[a, b)$ the middle section
2: Copy the middle section from p_1 to o_1
3: Mark those values from the middle section of p_2, that have not been copied to o_1
4: **for each** marked value V **do**
5: Denote the index of this value in p_2 as i. Look at the value at index i in p_1
6: Find the same value in p_2, let the index be j
7: **if** j is within the middle section **then**
8: **goto** 5: using this value
9: **else**
10: Copy V to o_1 to position j
11: **end if**
12: **end for**
13: Copy the remaining elements from p_2 to o_1

Example Let

$$a = 3, \quad b = 7,$$
$$p_1 = (1, 5, 6 \mid 2, 4, 9, 8 \mid 3, 7),$$
$$p_2 = (3, 9, 2 \mid \mathbf{7}, \mathbf{5}, 4, 8 \mid 1, 6).$$

The child o_1 after copying the first section will be: $o_1 = (?, ?, ? \mid 2, 4, 9, 8 \mid ?, ?)$. The missing values from middle section are 7 and 5. First, we try to place value 7: looking at the same index in p_1, we search for location of value 2 in p_2. Since the index is outside the middle section, we may place value 7 there: $o_1 = (?, ?, \mathbf{7} \mid 2, 4, 9, 8 \mid ?, ?)$. We repeat the process for value 5. It leads to value 4 and because it is within the middle section, we continue to value 9. Its position is p_2 is free, so a place for value 5 was found: $o_1 = (?, \mathbf{5}, 7 \mid 2, 4, 9, 8 \mid ?, ?)$ The rest is copied from p_2. Result $o_1 = (\mathbf{3}, 5, 7 \mid 2, 4, 9, 8 \mid \mathbf{1}, \mathbf{6})$.

Edge recombination crossover The *edge recombination crossover*[53] tries to combine parent individuals in such a way that it creates as little new edges (neighbouring genes) as possible in the offspring. The algorithm is described below (Algorithm 3).

Algorithm 3 Edge recombination crossover

```
Neighbourhoods ← List of neighbours for each gene.
o1 = EMPTY
i = the first element from a random parent.
while o1 is not complete do
    Append i to o1
    Remove i from every neighbourhood ∈ Neighbourhoods
    if Neighbourhood list for i is empty then
        j ← random element not already in o1
    else
        j ← arbitrary neighbour of i that has fewest neighbours.
    end if
    i = j
end while
```

Example Let

$$p_1 = (1, 5, 6, 0, 2, 4, 3),$$
$$p_2 = (3, 2, 5, 4, 0, 1, 6).$$

Table 2.1 shows initial state of Neighbourhoods lists.

Let us assume that we take the first element from p_1, that is 1 and append it to the child. We remove value 1 from all the neighbourhood lists. Next, we take a look at its neighbours $(0, 3, 5, 6)$. We should pick the one that has the fewest neighbours, but since all have the equal number, let us assume number 6 was picked. We repeat this procedure until we get $o1 = (1, 6, 3, 4, 0, 2, 5)$.

0:	1	2	4	6
1:	0	3	5	6
2:	0	3	4	5
3:	1	2	4	6
4:	0	2	3	5
5:	1	2	4	6
6:	0	1	3	5

Table 2.1: Initial neighbourhood lists

Cycle crossover The cycle crossover[54] aims to preserve the absolute order of permutation elements. It has two phases. In the first one, a cycle between parents p_1 and p_2 is found. In the second, offspring is created by copying the cycle elements from p_1 and the remaining elements from p_2. Creating another offspring may be achieved by switching parents in the second phase. Details may be viewed in Algorithm 4.

Algorithm 4 Cycle crossover

```
1: p ← random parent p1 or p2
2: x ← random element from p
3: x_i ← x
4: cycle ← ∅
5: repeat
6:     cycle ← cycle ∪ {x_i}
7:     p ← other parent
8:     x_i ← p[x_i]
9: until x_i = x
10: copy elements of cycle from p1 to o1 to the same positions
11: for each value v ∈ p2 \ cycle do
12:     i ← first free position of o1
13:     o1[i] ← v
14: end for
```

Example Let

$$p_1 = (1, 5, 6, 0, 2, 4, 3),$$
$$p_2 = (3, 2, 1, 5, 4, 6, 0).$$

Let us also assume that we start with value 1 in p_1. At index 1 in p_2 we find value 2. Returning to p_1 at index 2 yields value 6 and finally, the cycle is completed at p_2 index 6. Cycle is therefore $(1, 2, 6, 0)$. These values are copied to the first offspring from p_1 on the same positions: $o_1 = (\mathbf{1}, ?, \mathbf{6}, \mathbf{0}, \mathbf{2}, ?, ?)$. The individual is filled with the remaining values from p_2: $o_1 = (1, \mathbf{3}, 6, 0, 2, \mathbf{5}, \mathbf{4})$.

Mutation Operators

Swap mutation This operator cycles through all the elements of a permutation. Each gene is swapped with another random gene with probability p_G (gene mutation probability).

2.1.2 Multi-Objective Optimization

Firstly, the origins of the multi-objective evolution are mentioned in this section. Next, related definitions are listed. Afterwards, the *Non-Dominated Sorting Genetic Algorithm*[57] is described. More precisely, its second version. Optimizing more than one objective at once brings several difficulties. For instance, there is generally no linear order between the individuals, making some individuals incomparable. This occurs, if individual i is better at one objective than individual j and vice-versa at other objectives. More formally, for objectives k, l:

$$k \neq l : f_k(i) < f_k(j) \ \wedge \ f_l(j) < f_l(i) \implies i \not\prec j \wedge j \not\prec i,$$

where $f_k(i)$ means objective value for individual i in objective k. And $i \prec j$ denotes that individual i is better than individual j. Without loss of generality, let us assume that we want to minimize all the objectives. Comparing individuals in the multi-objective case is generally done by the *Pareto dominance*.

Definition 2.1. *Pareto dominance*
Individual i dominates individual j ($i \prec j$) if and only if for each objective k, $f_k(i) \leq f_k(j)$ and there is at least one objective l, $f_l(i) \neq f_l(j)$. Otherwise ($i \not\prec j \wedge j \not\prec i$) individuals are said to be mutually non-dominated.

Definition 2.2. *Pareto set*
The set of all individuals that are dominated by no other individual in the population is called the Pareto set.

Fitness Aggregation

The intuitive approach to multi-objective evolution is to aggregate all objectives f_i to a single f via weighted sum. This enables the use of classical single criterion EA. There is a shortcoming of how to set the weights of the fitness functions.

Vector Evaluated GA

Another algorithm is called *Vector evaluated genetic algorithm*[56]. It was one of the first attempts to address multi-objective evolution. Let N be the population size and n the number of objectives. For each f_i, the population is sorted by f_i and the best $\frac{N}{n}$ individuals are selected for genetic operations. The separate optimization by the objective functions tends to converge to the extremes of the individual objectives. Furthermore, it is difficult to maintain population diversity.

NSGA-II

The second version of the non-dominated sorting genetic algorithm (NSGA-II)[57] uses a different approach to overcome the shortcomings of the previous methods. Its idea is mainly based on sorting individuals by non-dominance and on a mechanism called *crowding distance*. Both key aspects are explained below. Another advantage of this algorithm is the possibility to use elitism.

Non-dominated sort During selection, the population is first sorted based on non-dominance. Individuals are assigned to non-dominated Pareto fronts. The first front contains only the non-dominated individuals. The individuals in the second front are dominated only by the individuals in the first front etc. The individuals are assigned a rank according to their front, for example the non-dominated individuals get rank 1, the ones from the next front get rank 2, and so on.

Crowding distance Another value is computed in addition to rank for every individual. It is called the *crowding distance*. It is a measure of how close an individual is to its neighbours. Its main purpose is to maintain diversity in the population, in other words, to choose individuals that are more evenly spread across the Pareto front. The crowding distance $d(i)$ for individual i is computed for each front separately. See Algorithm 5.

Algorithm 5 Crowding distance

1: $n \leftarrow$ number of individials in the current Pareto front.
2: **for each** individual $i \in$ Pareto front **do**
3: $\quad d(i) = 0$
4: **end for**
5: **for each** objective function f **do**
6: $\quad P \leftarrow$ sort population by f
7: **end for**
assign infinity to the best and the worst individuals:
8: $d(0) \leftarrow \infty$, $d(n) \leftarrow \infty$
9: **for** $j = 1$ **to** $n - 1$ **do**
10: $\quad d(P[j]) = d(P[j]) + \frac{f(P[j-1]) - f(P[j+1])}{f(P[0]) - f(P[n])}$
11: **end for**

Environmental selection The environmental selection operator combines both parent and offspring populations, thus allowing elitism. This way, the best individuals from the parent population are preserved and together with the best individuals from the offspring population form a new generation. See Algorithm 6 for details.

Mating selection The individuals for mating are selected by using a binary tournament selection. Individuals with lower ranks are deemed superior to others and are selected first. In case the rank is equal for both individuals, the individual with greater crowding distance is used, thus not damaging diversity.

2.2 Evolutionary Scheduling

Scheduling problems are known to belong to the NP-hard class, which makes them too difficult to solve using complete methods. It is therefore logical to find out whether evolutionary algorithms are a suitable method. In the next section, we shall describe traditional ways to model scheduling problems as an

Algorithm 6 Environmental selection

```
1: n ← population size
2: P_i ← parent population in iteration i
3: O_i ← offspring population in iteration i
4: F ← fast-nondominated-sort(P_i ∪ O_i)
5: P_{i+1} ← ∅
6: for each Pareto front F_j ∈ F do
7:     compute-crowding-distance(F_j)
8:     P_{i+1} ← P_{i+1} ∪ F_j
9:     if |P_{i+1}| ≥ n then
10:        break
11:    end if
12: end for
13: P_{i+1} ← n best individuals from P_{i+1}
```

individual, how to define operators and research outcomes from various articles. Typically, the problem is modelled as a permutation of actions. During the population evaluation, schedules are often obtained by some heuristic decoding of the individuals. We shall deal with the Resource-Constrained Project Scheduling Problem, Single Machine Scheduling Problem and Job-Shop Scheduling Problem.

2.3 Approaches to Scheduling Problems

Choosing suitable representation is crucial for the evolutionary algorithms in order to find a good solution. The question of encoding was thoroughly studied[19]. The results show, that even slightly different problems may require completely different representations. The encoding may be something as simple as one-dimensional binary array or as complicated as a sequence of neural networks or cellular automatons. There are two categories of encoding: direct and indirect. In scheduling, the direct methods work with schedules as individuals. This leads to the need to design potentially complicated genetic operators. In the latter method, individuals need a decoding procedure. The result of the decoding is a schedule. Genetic operators may lead to infeasible or illegal individuals, so measures must be taken to repair them.

2.3.1 Resource-Constrained Project Scheduling Problem

The resource-constrained project scheduling problem(RCPSP)[20] deals with assigning jobs or tasks to a set of resources with limited capacity trying to optimize some predefined objectives (typically makespan). More formally[21], the RCPSP is defined as a combinatorial optimization problem, which aims to optimize an objective function $f\colon \mathcal{Y} \to \mathbb{R}$, where $\mathcal{Y}$ identifies a discrete space of feasible solutions. The resource-constraint scheduling problem is defined by a triplet $(J, p, E, \mathcal{R}, B, b)$.

Jobs of the project are denoted as a set $J = \{j_0, j_1, ..., j_n, j_{n+1}\}$. Artificial jobs j_0 and j_{n+1} represent the schedule's start and end, respectively. Job durations are

represented by a vector $p \in \mathbb{N}^{n+2}$, where p_i is the duration of job j_i. The artificial jobs have a special value $p_0 = p_{n+1} = 0$. Precedence constraints are defined by a relation E, a pair $(j_k, j_l) \in E$ means that job j_k precedes job j_l ($j_k \prec j_l$). Precedence is a transitive relation, therefore $j_i \prec j_k \wedge j_k \prec j_l \implies j_i \prec j_l$. Bearing that in mind, let us assume that j_0 precedes all other jobs and j_{n+1} succeeds all other jobs. There are q renewable resources formalized by set $\mathcal{R} = \{R_1, ..., R_q\}$. Resource availabilities are represented by a vector $B \in \mathbb{N}^q$, where B_k represents availability of R_k. In case $B_k = 1$, the resource R_k is called unary and may be used only by one job at a time. Jobs resources requirements are formalized by matrix $b \in \mathbb{N}^{(n+2)\times q}$, such that b_{ik} represents the amount of resource R_k required by job j_i per a unit of time. The solution to the RCPSP is a vector of job starting times $s = (s_0, s_1, ..., s_n, s_{n+1})$ that do not violate precedence or resource constraints:

$$s_k - s_i \geq p_i \qquad \forall (j_i, j_k) \in E$$
$$\sum_{i=1}^{n} b_{ik} \leq B_k \qquad \forall R_k \in \mathcal{R}, \forall t \geq 0$$

Priority list encoding In evolution, schedules may be represented as a priority list of jobs[22]. The priority list of job indices $b = (b_0, b_1, ..., b_n, b_{n+1})$ might be constructed from schedule's starting times s, where $s_{b_i} \leq s_{b_j}$ for $i < j$. A schedule is constructed from given priority list in the following way:

1. $s_0 = c_0 = 0$. c_j stands for the completion time of job j. Let $i = 1$.
2. Let us assume that we already know the starting times of the first $i - 1$ jobs. Determine the starting time s_{b_i}. In case the resource constraints are met:
$$s_{b_i} = \max_{0 \leq l \leq i-1} \left\{ \{ c_{b_l} \mid j_l \prec j_i \} \cup \{ s_{b_l} \mid j_l \not\prec j_i \} \right\}$$
Otherwise, s_{b_i} is assigned the completion time of the first job when the resource constraints are met.
3. Terminate if $i = n$, otherwise $i = i + 1$, go to step 2.

Crossover The priority lists are permutations of jobs. There are many ways to crossover permutations. However not all of them preserve precedence constrains. We shall describe one of them[23]. Some more will be shown later. Given two feasible priority lists $b^1 = (b^1_1, ..., b^1_c, b^1_{c+1}, ..., b^1_n)$ and $b^2 = (b^2_1, ..., b^2_c, b^2_{c+1}, ..., b^2_n)$, let us choose a crossover point c at random. Offspring $b^{1'} = (b^1_1, ..., b^1_c, b^2_{k_1}, ..., b^2_{k_{n-c}})$ is obtained by taking the first c symbols from the first parent and the rest from the second parent preserving relative positions of elements. The elements $b^2_{k_1}, ..., b^2_{k_{n-c}}$ are precisely those not contained in the sequence of the first c elements. The other offspring is generated symmetrically. Note that the offspring are also feasible, because the parents were also feasible and no precedence constraints could be violated (relative order from parents was preserved).

Mutation An example of mutation may be swapping two elements or setting the value on index i to b_k and then shifting the elements from the range (i, k) one position to the right. This may produce infeasible offspring. We can either reject or repair them.

Hybrid GA-CSP Rigi and Mohammadi approached the RCPSP problem by combining GA and CSP[42]. They used binary representation of individuals. It consists of n substrings, where n is the number of tasks. Each substring $l = (l_1, ..., l_n)$ encodes precedence of the given task. For example $l_3 = 1$ means that the task represented by l takes precedence over the third task. As a result of the simple representation, simple one point crossover was used. Mutation incorporated local search: random substring s is chosen and all neighbours acquired by flipping variables in s are evaluated. The best one is chosen as the result of the mutation. Individuals might be illegal after application of the operators, hence repair operations are needed. The constraint solver tries to resolve the potential conflicts created by the genetic operators. It uses a heuristic for minimizing conflicts in the precedence constraints.

2.3.2 Single Machine Scheduling Problem

The single machine scheduling problem (SMSP)[24] is a class of problems that is concerned with assigning a set of jobs on a single processor or machine.

Permutation Individuals are commonly represented as a permutation of jobs. One example of crossover is the uniform order-based crossover, which generates offspring of two parents in the following manner: It copies a random number of genes of parent 1 to the same locations and then fills the rest from parent 2 (order from the second parent is preserved). Mutation is usually gene swapping.

2.3.3 Job-Shop Scheduling Problem

In the general job-shop scheduling problem (JSSP)[24], there are j jobs and m machines. Each job contains several tasks, which have to be carried out on a different machine with different processing times. Jobs have precedence constraints as in the SMSP. Also the tasks have to be processed in a specific order. Schedule for job j_i is a sequence of pairs (m, t), where the i-th pair denotes that task i will run on machine m and it takes t units of time. Schedule for the problem comprises the schedules for each job. Schedule is said to be feasible if no precedence constraints are broken and each machine processes at most one task at a time. Now, we will take a look at possible representations and genetic operators.

Chunks For $j \times m$ problem, individuals may be encoded as a string of $j \times m$ chunks. A legal schedule is constructed in this way: chunk *abc* instructs to put the first untackled task of a-th unfinished job and place it at the earliest time in the schedule, then continue with the task of b-th job, c-th job and so on.

Disjunctive graph We can describe the JSSP by a disjunctive graph $G = (V, C \cup D)$, where V is a set of vertices representing tasks of the jobs. Two special nodes are added to V: source (beginning of the schedule) and sink (end of the schedule). C is a set of directed arcs representing precedence constraints of the tasks. D is a set of undirected arcs representing pairs of tasks that must be processed on the same machines. The job-shop scheduling problem might be perceived as defining an ordering between all tasks that must be processed on the same machine. This is done by turning all the undirected edges to directed ones in the disjunctive graph representation. The resulting graph is called *consistent* if it does not contain cycles.

Binary sequence Schedule might be obtained from the undirected graph defined above by turning all undirected edges into directed ones. If all directed edges of the graph are labelled 0 or 1, the schedule can be represented as a binary string. Conventional GA was proposed by Nakano and Yamada[25]. Its advantages include the possibility to use common genetic operators (e.g. 1-point crossover). However, illegal schedules may be created in this way. Schedules are repaired by a harmonization algorithm[25], which has two parts. The first repairs inconsistencies within each machine, the other repairs precedence constraints.

Set of permutations Another way to represent the JSSP is by permutations of jobs on each machine. This set of permutations is called a *job sequence matrix*. The *subsequence exchange crossover (SXX)* was proposed by Kobayashi, Ono and Yamamura[26]. Let i_0 and i_1 be two individuals represented by job sequence matrices. A pair of subsequences, one from i_0 and the other from i_1, is called *exchangeable* if and only if they consist of the same set of jobs. The operator searches for such subsequences and creates new individuals by interchanging them. Schedules may have to be repaired after the crossover.

Permutation with repetition Bierwirth[27] proposed representation by an unpartitioned permutation with m-repetitions of jobs. Each job index appears m times in the permutation, once for each of its task. Tasks in this permutation are linearly ordered. A new *precedence preservative crossover (PPX)* was proposed[28]. Offspring is generated from its parents p_0 and p_1. The algorithm starts by selecting a random parent, then it appends its first element to the offspring. This element is deleted from both parents. This is repeated until the parents are empty. No new precedence edges are introduced here.

GT crossover The GT algorithm[29] for building schedules was modified into the GT crossover[30]. It uses both parents as guides how to build the offspring schedule. Let H be a binary matrix of size $j \times m$. $H_{ir} = 0$ denotes that the i-th task on machine r is determined by information from the first parent. Otherwise, the second parent is used. Let us label these parents as p_0 and p_1. This crossover produces only one offspring. In case another is needed, parents' roles are switched and the algorithm is run again. Note that schedules obtained by this method are valid and thus need no repairing. The crossover consists of the following steps:

1. Let D be a set of all the earliest tasks in a job sequence not yet scheduled and T_{jr} be a task with the minimum earliest completion time *ect* in D: $T_{jr} = \text{argmin}\{T \in D \mid ect(T)\}$.

2. Assume $i-1$ operations have been scheduled on machine M_r. A conflict set $C[M_r, i]$ is defined as:

$$C[M_r, i] = \{T_{kr} \in D \mid T_{kr} \text{ on } M_r \wedge est(T_{kr}) < ect(T_{jr})\}$$

3. Select one of the parent schedules $\{p_0, p_1\}$ according to the value of H_{ir} as $p = p_{H_{ir}}$. Select $T \in C[M_r, i]$ that has been the earliest scheduled task in $C[M_r, i]$ in p.

4. Schedule T as the i-th task on M_r with its completion time equal to $ect(T)$.

The only difference from the original GT algorithm is in step 3, Thompson and Giffler chose the task arbitrarily.

Priority rule based GA Priority rules[31] are a simple yet powerful heuristic for solving the JSSP. These are used in the step 3 of the GT algorithm to choose a task from the conflict set. Individuals in the priority rule based GA[32] are represented as strings of length $j \times m - 1$. At position i, there is one of twelve priority rules, how to choose a task in the i-th iteration of the algorithm.

Shifting bottleneck based GA In 1988 Adams et al.[33] proposed a powerful heuristic for solving the JSSP called the *shifting bottleneck*. This method decomposes the JSSP into a set of SMSPs, one for each machine. The algorithm solves the machines one by one. The machine with the maximal lateness is identified as the bottleneck machine. Each time new machine is sequenced, all previously sequenced machines might be resequenced. It terminates once all machines are sequenced. The problem is represented by a disjunctive graph defined earlier in this section. At the end, all edges are directed, hence a schedule can be built. See Algorithm 7 for more details. The *shifting bottleneck based genetic algorithm*[32] encodes individuals as permutations of machines. This permutation serves as a machine selection mechanism for the bottleneck heuristic.

Genetic local search

Genetic algorithms can be greatly enhanced by applying local search methods. Newly created individuals are not immediately accepted but are substituted by a locally optimal individual. This technique is called *Genetic Local Search*[34].

Neighbourhood search crossover One of the first approaches to integrate local search into a genetic algorithm was proposed by Reeves[35]. Individuals are encoded as binary strings and the distance between individuals x and y denoted $d(x, y)$ is measured as Hamming distance between them. We say an individual y is intermediate between x and z, denoted as $x \diamond y \diamond z$ if and only if $d(x, z) = d(x, y) + d(y, z)$. The *k-th order neighbourhood* of x and z is defined as all the intermediate individuals that have Hamming distance of k to either x or y:

$$N_k(x, y) = \{y \mid x \diamond y \diamond z \ \wedge \ (d(x, y) = k \vee d(y, z) = k)\}$$

Algorithm 7 Shifting bottleneck heuristic

1: $M_0 = \emptyset$ (Scheduled machines)
2: Let M denote the set of all machines
3: Initialize disjunctive graph G
4: **for each** $M_i \in M \setminus M_0$ **do**
5: Generate single machine schedule for M_i from G
6: Compute $L_{max}(i)$ (minimizing maximal lateness)
7: **end for**
8: Let M_k be the machine that maximizes $L_{max}(i)$ (bottleneck)
9: Schedule jobs on M_k according to the computed $L_{max}(i)$
10: Direct the undirected edges between tasks on M_k according to the schedule.
11: $M_0 = M_0 \cup \{M_k\}$
12: **for each** $M_i \in M_0 \setminus \{M_k\}$ **do**
13: Make the edges between tasks M_i undirected
14: Generate single machine schedule for M_i
15: Reschedule jobs on M_i and direct the edges accordingly
16: **end for**
17: **if** $M = M_0$ **then**
18: Stop
19: **else**
20: **goto** 4:
21: **end if**

The *neighbourhood search crossover* (NSX) inspects the neighbourhood of two parent individuals and chooses the fittest as the offspring.

Multistep crossover fusion Based on the NSX, Yamada and Nakano created a more general crossover. The outcome of their effort is called the *Multi-Step Crossover Fusion* (MSXF)[36]. This crossover has several advantages compared to its predecessor: it handles more general representations, uses stochastic local search and does not rely on fixed neighbourhood condition, but rather biased stochastic replacement. This crossover was successfully applied to the JSSP. For a more detailed description, please refer to [37][38].

Parallel machine encoding Mesghouni et al. proposed an encoding called *parallel machine encoding*[39]. It is a direct representation of feasible schedules. An individual is represented by m lists, one for each machine. Every list contains assignment operations on the given machine. Every item in the list M_k is a tuple (i, j, t_{i,j,M_k}), where i is order of the task of job j and t_{i,j,M_k} is the processing time of the task on machine k. Initial population cannot be obtained randomly as it would result in infeasible schedules. Precise methods like constrained logic programming[40] are used. Crossover[40] is achieved by selecting two parents p_0, p_1 and one machine M_k randomly. List M_k is copied to the offspring from p_0, whereas p_1 provides not yet present elements from all other machines. Finally, the missing elements are inserted into the offspring. This crossover may violate the precedence constraints. Mutation tries to move an element from one machine to another, respecting the precedence constraints.

Parallel jobs encoding Another direct encoding named *parallel jobs encoding* was proposed by Mesghouni et. al.[41]. Since this representation respects the precedence constraints, the crossover does not produce illegal schedules anymore. Individuals are encoded as a set of lists, one for each job. A list for job j has k elements, where k is the number of tasks in j. Element is a tuple (M_k, t_{M_k}), where M_k is the machine that processes the given task and t_{M_k} is the processing time on that machine. Two crossover operators are used, *row* and *column* crossover, both produce feasible offspring. The *row crossover* selects both parents p_0, p_1 and a job j (row) at random. The first offspring receives job j from p_0, the remainder is copied from p_1. The second child is obtained by interchanging p_0 and p_1. The *column crossover* chooses parents in the same manner as the row crossover. In addition it selects task i (column) randomly. Task i of all jobs in the first child is copied from p_0, the rest from p_1. The mutation operator was named *the controlled mutation operator* and was designed to balance the machine loads. The operator chooses a random individual and a task assigned to a machine with high load. This task is moved to another machine, preferably to a machine with small load.

2.3.4 Dynamic Non-Deterministic Scheduling

Up until now, we have only considered problems, where all crucial information is known in advance, for example release dates and due dates of tasks, machine's availability etc. Moreover, these traits remained fixed. This is called static scheduling. In the real-life production, however, there is a more dynamic environment: release dates of tasks are unknown or may be changed, tasks may be added or removed during the processing and so on. The risks of disruption events like resource delays give rise to another criterion to optimize – resistance to these events or effort needed for rescheduling.

Rescheduling When disruptive events occur, it is important to decide when to reschedule. Scheduling of large problems is time-consuming, therefore rescheduling from scratch should be avoided. Instead, information from already computed schedules should be exploited. According to Madureira et. al.[43] there are two types of disruptive events that trigger rescheduling: *partial* and *total*. The *partial events* only change jobs or tasks attributes, such as release dates. The *total events* might contain removal or addition of jobs or tasks. Rescheduling with GA was attempted for the partial events by Fang et. al.[44]. The algorithm receives changed processing times or release dates of some jobs, determines all affected tasks through dependency analysis and runs the GA scheduling only on them. This approach is more economical than full rescheduling. Dealing with the total events is more complicated. Basically, there are two possibilities. The first possibility is to restart the GA with random initial population. The second possibility is more subtle: the initial population of the new GA run is seeded with transformed population of the last GA run[45][44]. Let us assume that rescheduling is required at time t, there are two repairing steps:

1. All tasks that have been started before t are removed from the individuals throughout population

2. Tasks of the new jobs are inserted at random indices in all individuals

After this transformation, the GA may be run again with faster convergence.

3. Problem Analysis

In this chapter, we shall briefly examine the complexity of the task at hand. Most notably, we shall try to count the number of possible inputs for the constraint solver on an actual workstation. It might be interesting to know how small a fraction of the search space we actually explore. The goal of this chapter is to illustrate the significance of a "good" input for the constraint solver.

Sample workstation For measurements and experiments in this section we used the workstation that is serialized to xml and can be found on the included CD (*./station/90.xml*). This station contains exactly 857 tasks. The directed acyclic graph depicting precedence of tasks is too large to fit on a page, therefore it can only be found on the included CD (see Appendix A). The graph visualisation serves only as an illustration of the complexity rather than providing accurate information. It is due to the great number of edges between tasks which renders the resulting image quite chaotic. Vertex numbering in the picture can be ignored, it denotes the order of the given task represented by the vertex. This order serves as the input for the constraint solver.

Workstation scheduling problem definition Formally, the scheduling problem for a workstation is an extension of a resource-constrained project scheduling problem. Let a set of tasks be denoted as $T = \{t_0, t_1, ..., t_n\}$ with processing times $\{p_0, p_1, ..., p_n\}$, $W = \{w_1, w_2, ..., w_k\}$ renewable resources (workers), $Q = \{q_0, q_1, ..., q_l\}$ worker qualifications. Each worker possesses a set of skills $S \subseteq Q$. There is a partial order ($\prec$) defined on T called precedence constraints. Each task is also resource-constrained: let $R_{t,q} = k$ mean that task t requires k workers with skill q. The goal of the scheduling is to determine start and end times for all tasks and to assign workers to them such that all resource and precedence constraints are satisfied. Furthermore, one or more objectives are optimized (typically makespan). The introduction of skills creates a challenge for modelling the corresponding CSP. However, the modelling is not a part of this thesis. The RCPS problems are known to be NP-complete[17]. Not only that, it is one of the most intractable combinatorial problems. It belongs to the strong NP-hard class of problems.

3.1 Experiment

We ran a series of experiments to test the importance of the order of tasks. The permutation of the order of tasks is used as the static variable ordering input for the constraint solver. In each experiment, the constraint solver was run 1,000 times on orderings generated by different methods. If a solution is not found in a given time limit, the solver is cut off and run again with the next input. We are interested in the count of successful runs. Firstly, we tried random permutations with various time limits. In the second experiment, permutations respecting tasks precedence constraints were generated. These were obtained by generating random topological orderings. Note that we are only interested in the number of successful runs. The influence of input on the quality of solutions will be

inspected in the next chapters. The experiments were performed on a machine with an Intel Core i7-4790 3.60 GHz CPU, 16 GB RAM, Windows 7 64-bit, Java 8 and a SSD drive.

3.1.1 Random Permutations

In the first series of experiments, we tried setting the solver limit successively to 2, 20 and 200 seconds. Random permutations were used. None of the experiments yielded any results within the specified time limits. It is not surprising, the search space is enormous as we will demonstrate later on in this chapter and violation of the precedence constraints among tasks is very likely with random permutations. As we can see, using random permutations is quite hopeless. Obviously, the need of exploiting the structure of the problem arises.

3.1.2 Random Topological Orderings

Random permutations often violate the precedence constraints. If the solver receives such variable ordering, it will attempt to schedule a dependent task before its predecessor without considering the necessity to schedule the preceding task before this one. There might be no way of scheduling the preceding task and the search algorithm must backtrack. There can be a huge number of traversing such useless branches in case of random permutations which makes them poor candidates for variable ordering. The topological orderings have no such disadvantage. The topological orderings were obtained from a directed graph of precedence constraints among tasks. The order was created by traversing the graph layer by layer. The vertices representing the tasks without any predecessors belong to the first layer. The second layer is formed by the successors of the vertices from the first layer. Thus, all vertices are assigned to layers. The order is created by retrieving the respective tasks from the graph successively from the first layer to the last. Random topology orderings are created by taking vertices within each layer in a random order. This approach fared much better. In the 2 seconds configuration, all 1,000 solutions were found. This rendered the 20 and 200 seconds experiments redundant. The reason it was so successful is that by generating random topological orderings we ensured the precedence constraints to hold.

3.2 Counting Permutations

The previous experiments have shown that it is desirable to consider only permutations that respect the precedence constraints of the respective tasks. Before we proceed to counting the exact number of topological orderings, let us define some terms.

Definition 3.1. *Poset*
Partially ordered set, or poset, is defined as an ordered pair $P = (X, \leq)$ where X is called the ground set of P and $\leq$ is the partial order of P. We say an order is partial if and only if it is reflexive, antisymmetric and transitive.

We can see that the set of tasks with the binary relation of precedence falls within the definition of a poset.

Definition 3.2. *Linear Extensions*
A linear extension of a partially ordered set P is a permutation of the elements $p_1, p_2, ..., p_n$ of P such that

$$p_i < p_j \implies i \leq j.$$

In other words, it is a permutation that respects the precedence defined by $\leq$. This is exactly what we need – a topological ordering. It was shown that finding the number of the linear extensions of a partially ordered set is #P-Complete[18].

The problem of counting the number of topological orderings may be decomposed by counting the weakly connected components separately as they are independent as there are no tasks from different components connected by an edge.

Definition 3.3. *Weakly Connected Component*
A weakly connected component is a maximal subgraph of a directed graph, such that for every pair of vertices u, v there is a directed path from u to v and an undirected path from v to u.

The resulting count of topological orderings can be therefore written as follows:

$$\prod_{i=1}^{k} t_i \cdot \binom{n - \sum_{j<i} n_j}{n_i} \tag{3.1}$$

where k is the number of weakly connected components, t_i the number of topological orderings for component i, n the total number of vertices representing tasks, n_i number of vertices in the component i. The upper part of the binomial simply stands for the number of free slots in the permutation. So the binomial represents the number of ways to choose slots for vertices in the current weakly connected component. The first component's binomial is $\binom{n}{n_1}$ as there are not any occupied slots. The second is $\binom{n-n_1}{n_2}$ and so on. Each of the binomials is multiplied by t_i in order to obtain the number of orderings of the component with respect to the whole permutation. Thus, all the slots are filled gradually and the number of topological orderings is computed. Note that this formula can use the exact values for the individual t_i as well as estimates. Given that the components are quite large and that the problem is #P-Complete, we shall use estimates instead of the exact number of orderings.

3.2.1 Estimates

The number of topological orderings for the problem at hand is estimated in this section. The computations are applied on the sample workstation described at the beginning of this chapter.

Upper bound The upper bound of the count of topological orderings is obviously

$$n! = 857! \approx 1.7 \cdot 10^{2143}$$

This number is barely conceivable. However, we are more interested in the lower bound.

Lower bound The lower bound is computed using dynamic programming. Let $G = (V, E)$ be a directed acyclic graph, where V is the set of vertices representing the tasks and E the set of edges representing precedence of tasks. We compute the number of ways to traverse the graph from the topologically first vertices to the topologically last ones. The first step is to sort vertices to layers according to topology. In the first layer, there are the vertices that have no incoming edges, in the second one, there are the ones that have incoming edges from vertices from the first layer, etc. Two vertices are added, one preceding all the vertices from the first layer, the other succeeding all the vertices in the last layer. The number of possible ways w_i to each vertex V_i is computed in the following way: the first vertex is assigned $w_1 = 1$ and then for each layer and for each vertex V_i in the layer:

$$w_i = \sum_{(V_j . V_i) \in E} w_j$$

The lower bound is stored in the w_i of the last vertex V_i. It is certainly lower or equal than the exact count, because it represents the number of traversal of the graph and not all ways generally contain all the vertices. Note that we could divide the graph into subgraphs of weakly connected components, estimate the topological orderings separately and then utilize the formula (3.1) to compute the estimate for the whole graph. The lower bound was computed as $e^{3278} \approx 4.1 \cdot 10^{1423}$.

Although substantially smaller, even the lower bound estimate is astronomical which gives us the idea how vast the search space must be. It is clear that clever generation of permutations is needed to guide the search in the right direction. We describe our approach in the chapter about evolutionary algorithms.

Implementation notes The graph was divided into components as shown in the formula (3.1). Furthermore, in order to avoid overflow and infinity values, the result had to be computed as natural logarithm and then transformed to the true estimate by some computational engine. Let b_i denote the binomial from formula (3.1), then the transformation might be written as:

$$\log \prod_{i=1}^{k} (t_i \cdot b_i) = \sum_{i=1}^{k} (\log t_i + \log b_i) = l$$

Therefore the result will be e^l. Using the logarithm reduction of the estimate for weakly connected components t_i and the binomial coefficients proved sufficient.

4. Evolution

Evolution continually innovates, but at each level it conserves the elements that are recombined to yield the innovations.

John Henry Holland

This chapter is dedicated to artificial evolution. While other researchers' approaches were described in the second chapter, this one addresses the way we tackled the problem. First, we take a look at out method of population initialization and our innovative genetic operators. Next, possible issues with diversity and balancing exploration with exploitation are discussed. Finally, we discuss how to measure the quality of individuals through the course of the evolution.

4.1 Framework

For the single-objective evolution, a simple implementation of EA was used. This implementation is also part of an EA course[1]. For the multi-objective one, the open source jMetal[51] implementation of NSGA-II was used.

4.2 Representation

Individual evaluator has available information about the workstation's tasks. Their order is represented as a permutation of their indices. Therefore, we can represent the individuals as permutations of n integers, where n is the number of tasks.

Decoding Individual is decoded in the following manner. The constraint solver is called with the order of tasks obtained from the individual. If it finds a solution, each task is assigned a worker and starting time, hence a schedule is created. After a solution is found, the search is stopped not to slow down the evolution. It is then easy to calculate desired objective of the calculated schedule e.g. makespan by iterating through all tasks and finding the completion time of the task that finishes the last. The solver tries to optimize the makespan, other objectives are only measured and used for evolution. Although the solver is cut off after finding the first solution, it can still find the optimal one in our case. If harder problems are presented, this issue could be resolved by running the solver again without the cut-off restriction on the best individuals obtained from the evolution .

4.3 Initial Population

It was already shown in Chapter 3 that the search space is enormous. Task precedence constraints and resource constraints should be respected. In this sec-

[1] https://github.com/martinpilat/evaTeaching/tree/master/src/evolution

tion, we shall describe several ways to initialize a population. Task precedence constraints and resource constraints should be respected when generating a population. Performance of various techniques of initialization on a particular station is measured using makespan as a metric. The makespan is measured in minutes. The workstation is the same as in the experiments in chapter 3. The solver time limit was set to 2 seconds. The optimal makespan for our problem is known to be 3,899 minutes.

4.3.1 Deterministic Methods

The methods mentioned in this section were already part of the project before the evolutionary approach was attempted. Since they are deterministic, they are not suited for population initialization. They are mentioned not only for the sake of completeness, but they are also the base of the successful stochastic methods, and provide baseline for comparison.

Precedence This method starts by finding a set of tasks that have no predecessor. The tasks are assigned numbers according to order in which they were obtained. The first vertex is labelled with zero, the second with one and so on. Next layer is formed by successors of the tasks from previous set. They are numbered in the same manner, except the starting number is the number of the last vertex in the previous layer incremented by one. This procedure continues until all tasks are labelled. The order in which tasks are assigned numbers in each layer is fixed, therefore this technique always yields the same results. The resulting makespan reached 5,889 minutes.

Dependent tasks count This approach determines the number of dependent tasks for each task. A task t_j is dependent on task t_i, if there is a directed path from t_i to t_j in the precedence graph. This is achieved by recursively obtaining successor tasks and then counting them. The tasks are then sorted in descending order by the number of dependent tasks. Note that the precedence is implicit in this method. This simple heuristic improved makespan to 4,401 minutes.

Precedence and distance This algorithm finds the longest path (measured in tasks' duration) to dependent tasks for each task. Tasks are ordered by the length of the path in descending order. This heuristic proved even more effective, resulting in makespan of 4,009 minutes.

4.3.2 Stochastic Methods

As was already mentioned above, the population needs to be diverse, therefore some randomness should be employed. One thousand individuals were generated by each method and the one with the best makespan is reported as a measure of performance.

Random Completely random initialization yielded no results at all at the given time limit.

Topological order iterator The more sophisticated methods use a topological order iterator[16] to generate the topological orders of the precedence constraint graph. It expects a directed acyclic graph and optionally a comparator of nodes. The comparator is used by a priority queue of nodes. Whenever two or more nodes are topologically equivalent, the comparator decides whichever node takes precedence.

Random topology A natural extension of the *precedence* method is generating random topological orderings of the precedence constraints graph. Whenever two nodes are topologically equivalent, the tie is broken randomly, favouring one or the other. The resulting makespan was 4,348 minutes.

Random topology with dependent tasks count This method also generates topological orderings, but in a case when a set of nodes can be assigned the same number, a node with the most dependent tasks is chosen with probability p. Otherwise, a random node from the set is chosen. The respective comparator is described in Algoritm 8. This yielded the optimal makespan of 3,899 minutes. In total, 2 individuals with the optimal makespan were found.

Algorithm 8 Comparator

```
1: function COMPARE(Task t_1, Task t_2)
2:     if random(0, 1) < p then
3:         return COMPAREBYDEPENDENTTASKSCOUNT(t_1, t_2)
4:     else
5:         return t_1 or t_2 randomly
6:     end if
7: end function
8: function COMPAREBYDEPENDENTTASKSCOUNT(Task t_1, Task t_2)
9:     deps_1 ← ∅
10:    deps_2 ← ∅
11:    DEPENDENTTASKSCOUNT(t_1, deps_1)
12:    DEPENDENTTASKSCOUNT(t_2, deps_2)
13:    if |deps_1| > |deps_2| then
14:        return t_1
15:    else
16:        return t_2
17:    end if
18: end function
19: function DEPENDENTTASKS(Task t, Collection dependentTasks)
20:    for each Task dep in dependent(t) do
21:        if dep ∉ dependentTasks then
22:            dependentTasks ← dependentTasks ∪ dep
23:            DEPENDENTTASKS(dep,dependentTasks)
24:        end if
25:    end for
26: end function
```

Algorithm 9 Distance Comparator

```
1: function compareByDependentDistances(Task t_1, Task t_2)
2:     if tasksDistances(t_1) > tasksDistances(t_2) then
3:         return t_1
4:     else
5:         return t_2
6:     end if
7: end function
8: function tasksDistances(Task t)
9:     distances ← ∅
10:    for each Task dep in dependent(t) do
11:        if dep ∉ distances then
12:            distances.put(dep, duration(dep))
13:        end if
14:        tasksDistances(dep,distances)
15:    end for
16:    return max(distances) + duration(t)
17: end function
18: function tasksDistances(Task t, Map distances)
19:    currentPathLength ← distances.get(t)
20:    for each Task dep in dependent(t) do
21:        oldPathLength ← distances.get(dep)
22:        if oldPathLength < currentPathLength + duration(dep) then
23:            distances.put(dep, currentPathLength + duration(dep))
24:            tasksDistances(dep,distances)
25:        end if
26:    end for
27: end function
```

Random topology with distance This approach uses the same way of generating (the *Compare* function from Algorithm 8), although the heuristic used with probability p is different: it uses the *distance* metric from previous section. Algorithm 9 describes how the distances are computed. This method resulted in 9 individuals with the optimal makespan of 3,899 minutes.

Diversity Care must be taken when choosing the parameter p in the last two stochastic methods. The parameter should not be too high to avoid premature convergence or loss of diversity. It should not be too low either, to exploit the heuristic. The best results were achieved, if the parameter was set to values between 0.8 and 0.9.

4.4 Fitness

Single-objective fitness Simple fitness was used in the single-objective case. For each evaluated individual i we have its schedule and therefore its makespan $C(i)$. We subtract it from a constant (has to be larger than the total duration of the station) and obtain fitness for individual i. For example:

$$f(i) = 10,000 - C(i)$$

This or similar operation was needed, because our EA *maximizes* the fitness and the objective is to *minimize* the makespan.

Fitness scaling In case the fitness values differences are too large or too small, the evolutionary algorithms may encounter problems. In case there are large differences in fitness among the individuals in the population, the roulette wheel selection tends to select only those individuals with large fitness and (almost) never selects those with lower values of fitness. This can lead to premature convergence and significantly reduce the performance of the algorithm. This is not desirable, mainly because the seemingly inferior individuals may hold some useful building blocks, which would unfold by applying genetic operators. If, on the other hand, the fitness values differences are too small, the competition is too great and the fitness proportionate selection loses its merit. In our case, the fitness values differences were small, so we used a scaling function g:

$$g\colon \mathbb{R} \to \mathbb{R}$$

$$g(x) = x^2$$

$$f(i) = (10,000 - C(i))^2$$

Multiple objectives In the single-objective algorithm makespan was used as the fitness function. In the multi-objective variant we added the objective we called *tasks ASAP*. This objective was described in the introductory chapter in the paragraph about local KPIs. Basically, it creates pressure on the tasks to start as early as possible. We want to minimize both objectives.

4.5 Genetic Operators

This section contains genetic operators devised for this thesis. Apart from them we used: *cycle crossover, partially mapped crossover, edge recombination crossover, precedence preservative crossover* and *swap mutation*. Those operators were described in the second chapter.

4.5.1 Mutation Operators

Precedence preserving shift mutation The *precedence preserving shift mutation* (PPSM) chooses a random element x. All positions lower than the index of x are inspected. In order for an index to pass, moving x to its position must not violate any precedence constraints. If there are any such indices, index i is chosen from them randomly. Element x is moved there, causing all elements beginning with the original element at position i and ending with the former predecessor of x to shift one position to the right. Thus is the individual mutated without breaking any precedence constraints. This procedure is repeated for predefined number of times. The parameter is called *precedence preserving shift mutation trials*.

Segment proportionate mutation In the *segment proportionate mutation* (SPM), the individual is divided into k segments of equal length and one is chosen randomly. The earlier segments have greater chance of being selected. After a segment is chosen, a random element is selected from the segment. This element is then swapped with a random element with greater index. This procedure repeats n times, where n is the number of genes if probability conditions are met: for each gene a random value must be smaller than the parameter called *gene mutation probability*. If it is not smaller the individual remains unchanged. We proposed this operator to exploit the fact that changes at the lower indices of the individuals have greater impact on their fitness.

Complete mutation An operator called *complete mutation* was introduced in order to boost diversity in the population. It creates a completely new individual in the same manner as in the initial population. However, the probability of this operator should be chosen with care. We do not want to overwrite potentially superior individuals. Furthermore, high probability of this mutation would probably result in the lack of exploitation. Exploitation versus exploration principle is mentioned later on in this chapter.

Topology mutation Although all of the presented mutation and crossover operators provide legal permutations, the results often violate precedence constraints. This leads to a population that has only a fraction of individuals successfully evaluated. The rest exceeds the solver's time limit, which drastically slows down the evolution. An even greater drawback is the impossibility to compare unevaluated individuals. This had to be resolved. We introduced a new operator called the *topology mutation*. New topological order is created using the topological iterator described above (4.3.2). This time, the comparator is defined by the linear order of the individual's permutation. Thus, if there are two

topologically equivalent tasks, the one with the earlier occurrence in the permutation is selected. This results in preserving as much information from the original individual as possible without violating the precedence constraints. Although replacing an infeasible individual with a repaired one can damage diversity, it usually leads to better quality solutions and faster convergence.

4.5.2 Exploitation / Exploration Trade-Off

As with many AI techniques, finding the balance between exploitation and exploration is very important. Exploitation makes use of the current knowledge, whereas exploration searches for new regions of the search space. If only exploitation is used, there is a great risk of getting stuck at local optima. It is also a very rigid search. Employing only exploration, on the other hand, is basically a random walk without exploiting the knowledge at hand. This trade-off should be kept in mind, when setting the parameters of the genetic operators. More general and detailed description of the trade-off may be found here[55].

4.5.3 Maintaining Diversity

Maintaining diversity in the population is crucial, loss of diversity may lead to stucking at local optima. One good precaution is the fitness scaling, but by itself it is not sufficient. Another, more obvious way, is the use of mutation operators. However, several different individuals may be transformed to identical ones after *topology mutation* is applied. New information may be brought to the population by the *complete mutation*. Additional measure tries to apply the genetic operators multiple times. If the produced individual is already in the generation, it is discarded. This continues, until a unique individual is created or the maximum number of repeats is reached. We call this number *diversity trials*.

4.6 Performance measures

The objective of evolution in our case is to provide several good individuals. The individual archive serves that purpose. Every time an individual is evaluated, it is added to the archive. After each generation of the EA, information is gathered from the archive. First, the archive is sorted by the first criterion, then by second criterion in case of a tie. As makespan is the most important measure, it is the first criterion. After that, n best individuals are considered, where n is for instance the population size. Afterwards, the best, average and worst values for every objective are logged along with the count of individuals (may exceed n) with the best objective values. These generation-wise logs are used to measure the EA performance. The respective graphs will be displayed in the next chapter.

5. Results

This chapter first describes the Mann-Whitney U test and its application to compare evolution runs. Next, the performance of genetic operators, initialization methods and evolutionary algorithms is measured.

All presented experiments were performed on the sample workstation mentioned in Chapter 3. This workstation with known optimal makespan of 3,899 minutes contains over 850 tasks. Although it would be optimal to perform the experiments on multiple scheduling problems, time restrictions enabled us to use only one workstation as input.

5.1 Mann-Whitney U test

Since the optimal solutions (in the terms of makespan) were often found in the initialization process, it is valid to ask, if it is necessary to run the evolutionary algorithm. The answer is positive, because a number of good-quality solutions is demanded. Also, the evolution produces overall better individuals than the initialization process. We will back up our claim by the Mann-Whitney U test[58]. Even if the test failed, usage of the evolution process might still be valid for optimizing several criteria.

The Mann-Whitney U test is a non-parametric test that allows to test the null hypothesis that the two samples come from the same population against the alternative hypothesis that one population tends to have larger values than the other. We use this to test, whether individuals taken from one evolutionary run tend to have similar fitness to those initialized by the best method described in the previous chapter. In other words, if the evolution is necessary. The outcome of the Mann-Whitney test include the **U** statistic, whose distribution is known under the null hypothesis. **P-value** is also part of the outcome. P-value is used to determine whether to reject the null hypothesis or not. In case the p-value $< \alpha$, where α is the level of significance, we can reject the null hypothesis. Otherwise, the differences between the samples cannot be significantly attributed to different distributions.

Computation of U There are two common ways how to compute U, a *direct* and an *indirect* method. The direct method is rather straightforward. All elements from both samples are compared pairwise. A win is awarded 1 point, a tie by 0.5 point. U for the first sample (denoted U_1) is the sum of points for wins and ties against elements from the other sample. U for the second sample is computed symmetrically. Note that $U_1 + U_2 = n_1 n_2$, where n_1, n_2 are the sample sizes. The indirect method relies on ranks:

1. Sort all observations by size and assign a rank to every element in ascending order starting by 1. Tied elements should have their ranks averaged to the same value. E.g. sequence $(4, 8, 8, 9)$ will be ranked as $(1, 2.5, 2.5, 4)$.

2. Let R_1 denote the sum of all ranks from the first sample.

3. $U_1 = R_1 - \frac{n_1(n_1+1)}{2}$.

Again, computation of U_2 is symmetrical.

Approximation of U For great sample sizes ($n > 10$), the distribution of U can be approximated by the normal distribution. Therefore the standardized score of U is:

$$z = \frac{U - \mu_U}{\sigma_U}$$

where μ_U is the mean of U and σ_U its standard deviation. Critical values of z for various levels of significance are tabulated. Let n_1, n_2 be the sample sizes as in the previous paragraph. Then the mean and standard deviation of U are computed as follows:

$$\mu_U = \frac{n_1 n_2}{2}$$

$$\sigma_U = \sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12}}$$

Test assumptions

1. Samples are taken randomly from both populations.
2. All observations must be comparable.
3. Each measurement must correspond to a different individual (independence within groups).

Hypothesis test We compare the following populations: the first was obtained by running 10 independent multi-objective evolutions with population size of 100 and 30 generations. Archives from all runs were merged. In total, at most 30,000 individuals (archive contains only distinct individuals). The second population was created using only the successful initialization method. This also yields at most 30,000 individuals. To test the null hypothesis that states that the two populations tend to have the same values, both groups were tested. The test assumptions hold:

1. Samples are random.
2. All observations are comparable by the makespan objective values.
3. Only distinct individuals are added to the archives.

We were interested if the EA performs better than the initialization, so we ran the one-tailed version of the test. The level of significance α parameter was chosen as 0.01.

Results The resulting Z-score was 385.92 and p-value $< 10^{-6}$. The Z-score was much larger than the critical value at 0.01 significance level, thus lies well in the rejection region. The p-value was smaller than α, so we can reject the null hypothesis and accept the alternative hypothesis that the evolution produces higher number of the optimal to suboptimal individuals.

5.2 Experiments

This section is dedicated to the outcome of experiments and discussion of the results. First, we compare crossover operators, while all other parameters will be fixed. We compare mutation operators, evolutionary algorithms and initialization methods in similar manner. The values of the fixed parameters were set based on empirical knowledge or the outcome of previous comparisons. Apart from the quality and quantity of found solutions, we bring another performance measure. It is the average generation of the first occurrence of an optimal individual. The index of the initial population is 0. If no optimal solution is found, the run is assigned the number of the generations in total, e.g. in our experiments, there is always 30 generations indexed from 0 to 29, the run without optimal solutions is awarded 30.

Graph description Every experiment's output was recorded in the form of a graph. Each experiment consists of 10 independent runs of the EA. The x-axis defines generations and the left y-axis the objective. For every generation, the objective value of the best individual is recorded, as well as the count of the best ones. The blue curve takes the average of the objective over all runs. The red curve denoted, Q_3 or the third quartile, represents the objective value achieved in the top 25% runs and the green curve, denoted Q_1 or the first quartile, represents top 75% of the runs. The dotted lines represent the count of best individuals and they relate to the right y-axis. Colours of the lines have the same semantics as the solid lines. In case of a makespan graph, there is a horizontal dashed line which stands for the optimal makespan.

5.2.1 Crossover Comparison

Crossover operators are compared in this section. Only the traditional crossover operators (2.1.1) were employed. All parameters for every crossover operator are fixed. The fixed values were chosen based on empirical knowledge. We arranged the results from the operator with the worst results to the one with the best. We used our proposed segment proportionate mutation (4.5.1) to test each of the traditional operators.

Parameters

- Initialization: random topology with distance, probability 0.8
- Crossover: probability 0.8, 10 diversity trials
- Segment proportionate mutation: probability 0.3, gene mutation probability 0.01, 20 segments
- Topology mutation: probability 1.0
- Population size: 100
- Generations: 30
- Algorithm: NSGA-II

Edge recombination crossover

The edge recombination crossover proved to have poor performance on our problem. Although it reached the optimal makespan in the majority of runs, the number of individuals with the best makespan was very low. Figure 5.1 depicts the makespan and Figure 5.2 shows a graph of the tasks ASAP objective. The curves bring evidence that the separate evolutionary runs were very different from each other. Furthermore, the average generation which reached the optimal individual was 17.4.

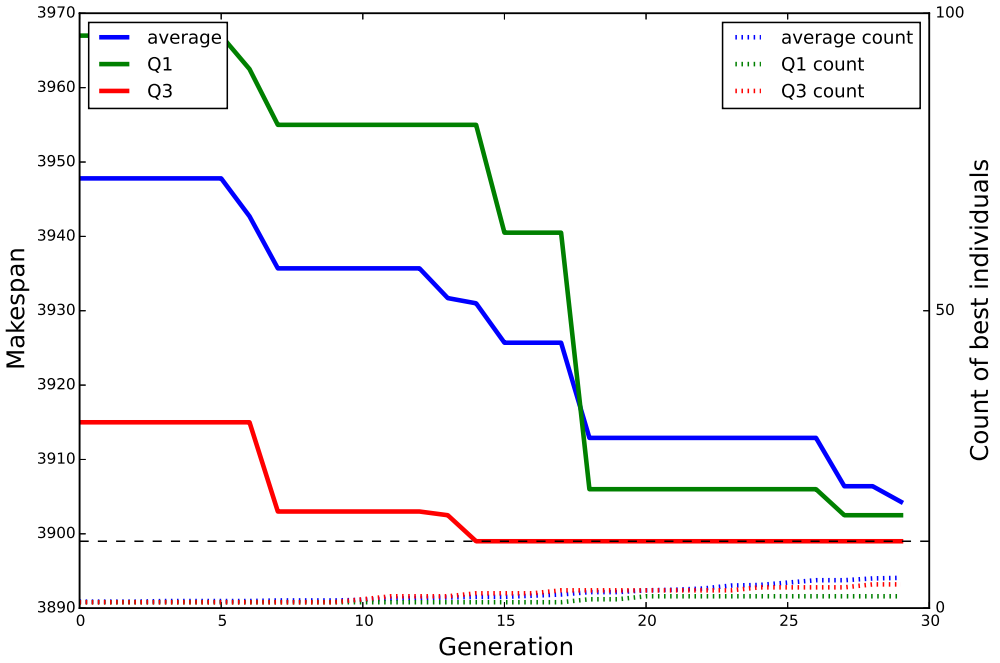


Figure 5.1: Edge recombination crossover - makespan

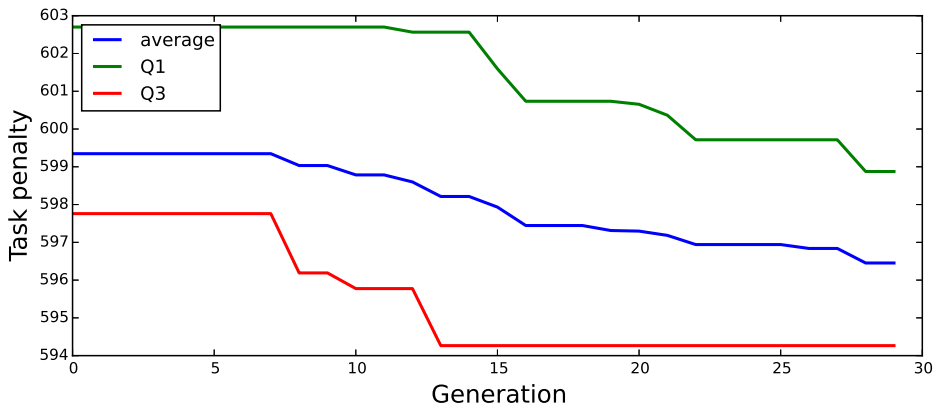


Figure 5.2: Edge recombination crossover - tasks ASAP

Partially mapped crossover

The partially mapped crossover fared much better. It reached the optimal makespan in all runs and the number of optimal individuals was much higher (Figure 5.3). Additionally, the optimal individual was found on average in generation 1.8. Also, the progress of the tasks ASAP objective was much smoother (Figure 5.6) and it reached significantly better values than the edge recombination crossover.

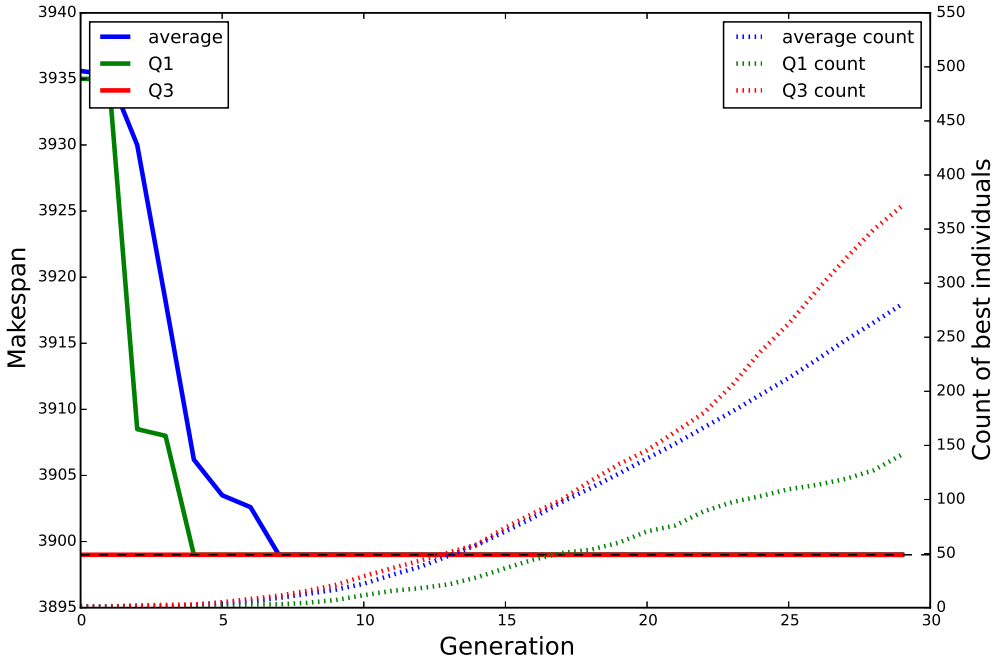


Figure 5.3: Partially mapped crossover - makespan

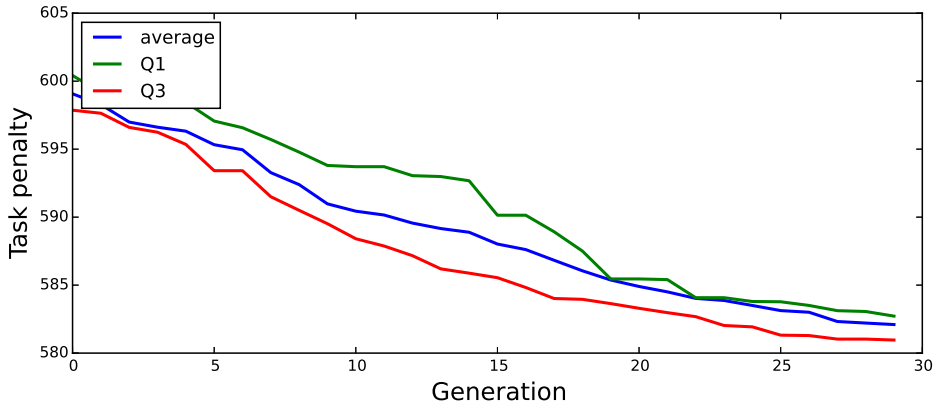


Figure 5.4: Partially mapped crossover - tasks ASAP

Precedence preservative crossover

The precedence preservative crossover reached similar results as the partially mapped crossover. Nevertheless, the PPX generated more makespan-optimal individuals and reached overall better tasks ASAP values. This operator seems to have the fastest convergence, the optimum was found on average in generation 0.9. Furthermore, the tasks ASAP curve was smoother and very similar in all runs. See Figure 5.5 and 5.6 for details.

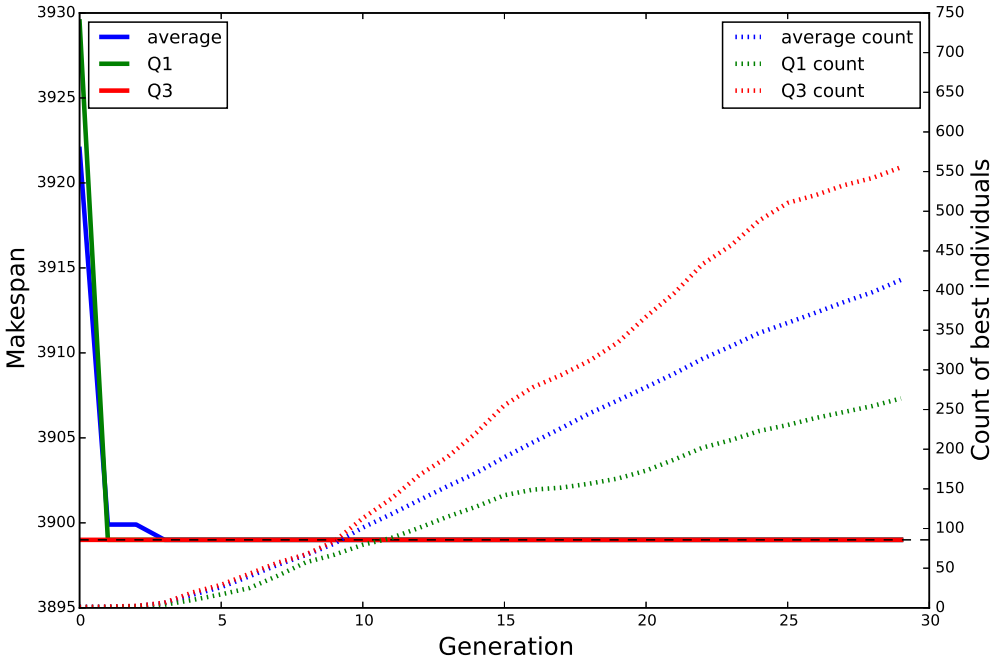


Figure 5.5: Precedence preservative crossover - makespan

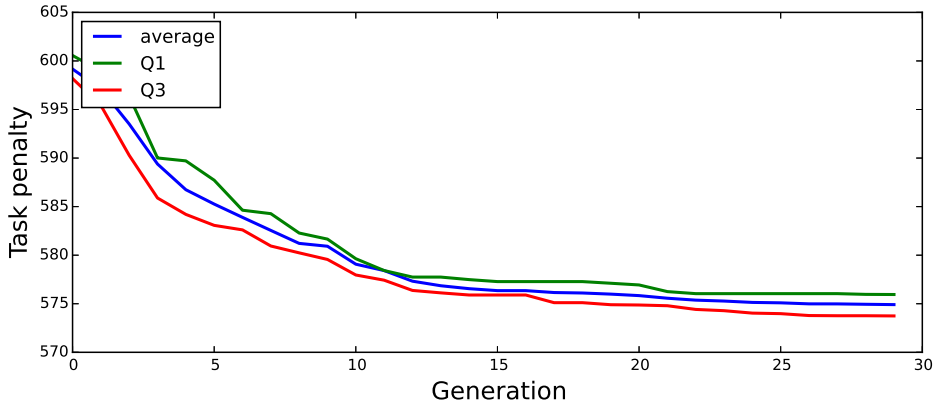


Figure 5.6: Precedence preservative crossover - tasks ASAP

Cycle crossover

The cycle crossover proved to be the most effective. The tasks ASAP objective had similar results as the precedence preservative crossover (Figure 5.8). However, the number of optimal individuals was higher and more importantly there were lesser differences among individual runs (Figure 5.7). Optimal schedules were reached on average in generation 1.4. We can see that there were only small differences among individual runs, which cannot be said about the PPX. On the other hand, the PPX does not need any repairing mechanism. If the parents are feasible, so are the offspring.

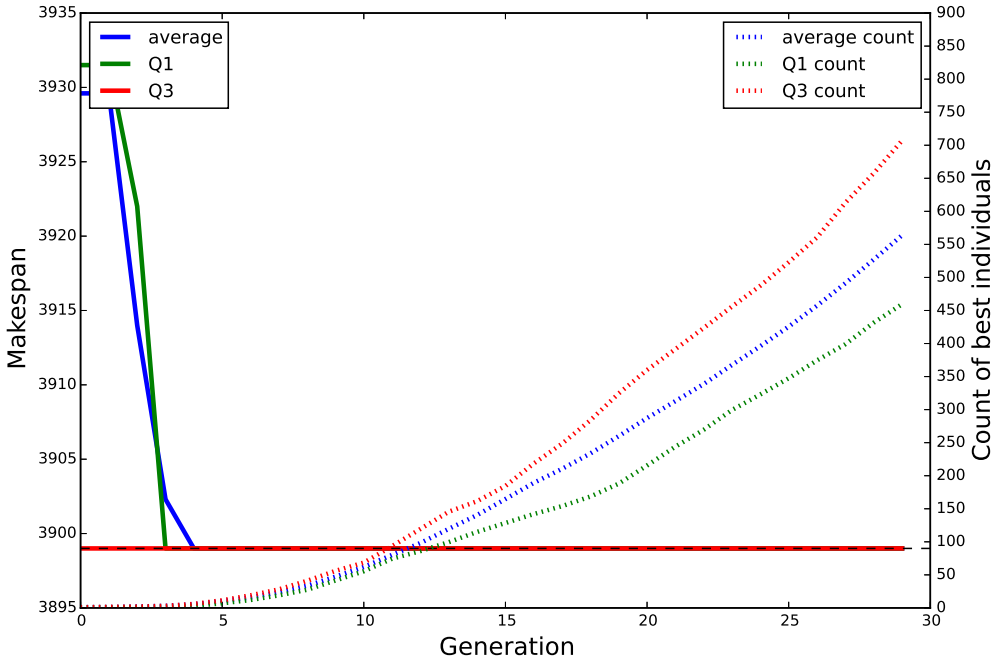


Figure 5.7: Cycle crossover - makespan

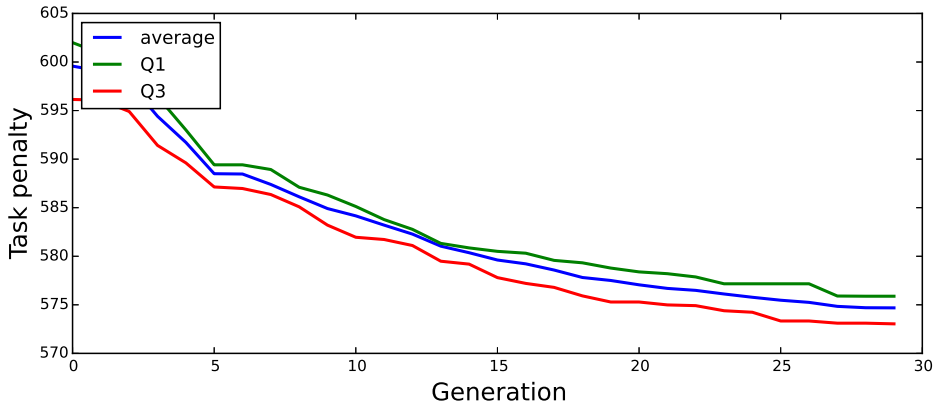


Figure 5.8: Cycle crossover - tasks ASAP

5.2.2 Mutation Comparison

Mutation operators are tested in this section. Again, all parameters except for the mutation are fixed. Based on the crossover experiments, the cycle crossover is used in the mutation tests. It is clear that different mutation operators' performance may vary with different crossover operators, however, it is time-consuming to test all combinations.

Parameters

- Initialization: random topology with distance, probability 0.8
- Cycle crossover: probability 0.8, 10 diversity trials
- Topology mutation: probability 1.0
- Population size: 100
- Generations: 30
- Algorithm: NSGA-II

Segment proportionate mutation

The segment proportionate mutation (4.5.1) was run with parameters identical to the experiments in the previous section. We have already seen the results in Figure 5.7 and 5.8.

Swap mutation

Next, the simple swap mutation (2.1.1) was tried. The parameter *mutation probability* was set to 0.3, *gene mutation probability* to 0.01 and *diversity trials* to 10. The results were similar to the experiments with the segment proportionate mutation. We compared the number of makespan-optimal individuals for each run for both operators using the Mann-Whitney U test at 0.05 level of significance. The resulting p-value was very high (0.79), therefore we cannot reject the null hypothesis which states that both mutation operators produce similar number of makespan-optimal individuals. However, the experiment with this mutation seems to converge slower (generation 2.6) than the proposed SPM (4.5.1) (generation 1.4). Figures 5.9 and 5.10 depict the results for the swap mutation experiment.

Precedence preserving shift mutation

The precedence preserving shift mutation (4.5.1) was tested next. The parameter *mutation probability* was set to 0.3, *precedence preserving shift mutation trials* to 10 and *diversity trials* to 10. See Figure 5.11 and Figure 5.12 for results. Although the average and the absolute maximum numbers of makespan-optimal individuals were higher than in the previous methods, the differences amongst individual runs were not statistically significant according to the The Mann-Whitney U test at the level of significance of 0.05. This method also proved to be quicker (generation 1.4) than the traditional swap mutation (2.1.1).

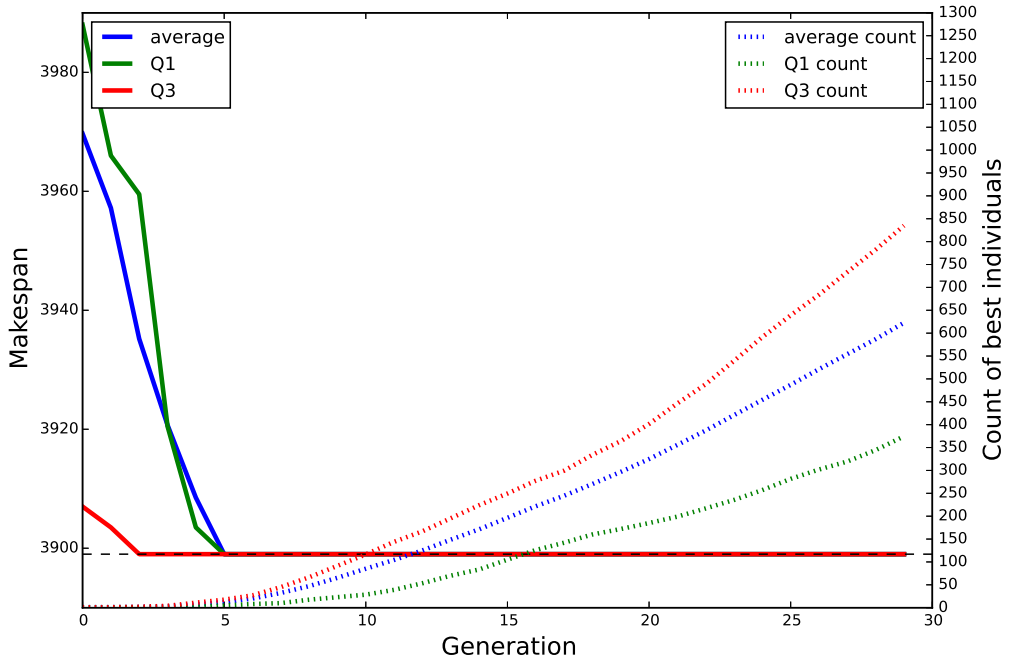


Figure 5.9: Swap mutation - makespan

No mutation

Finally, evolution was tested with no mutation at all (except for the necessary topology mutation of course). All other parameters remained the same. Figures 5.13 and 5.14 show the results. We can see that the number of individuals with the optimal makespan was even higher than in the previous experiments. Also, it was very quick, the optimal solution was on average found after 1.7 generations. The Mann-Whitney U test comparing the number of optimal individuals hinted that using no mutation is significantly better than using the segment proportionate mutation. However, comparisons with the other mutation operators remained inconclusive.

5.2.3 Algorithm Comparison

The experiments described in the previous section suggest that the cycle crossover applied to our problem performs the best without any mutation at all. However, the differences to other mutation operators proved statistically insignificant. Therefore using mutation operators should not be dismissed. Up until now, we ran experiments using the NSGA-II. In this section, we used the best parameter settings from previous experiments and apply it on the single objective evolutionary algorithm. Since there was no significant difference between using the precedence preserving shift mutation and no mutation, we decided to use the PPSM (4.5.1), mainly because it improves the chances of escaping from potential local optima.

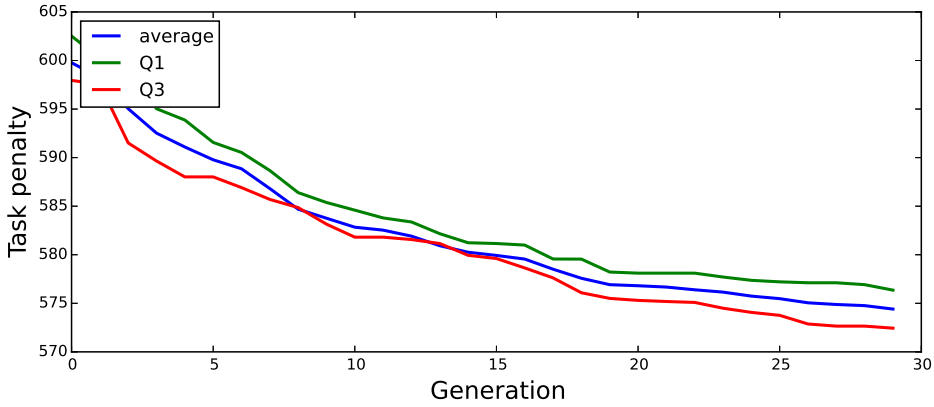


Figure 5.10: Swap mutation - tasks ASAP

Single-objective EA

Single-objective evolutionary algorithm is tested in this section. The parameters are the same as in the previous section. The mutation operator is the precedence preserving shift mutation. Its parameters were set to the same values as in the section about testing mutation operators. Figure 5.15 depicts the experiment with the regular single-objective EA. The optimal solution was on average found after 3.7 generations. Next, evolutionary algorithm with fitness scaling was tried. The results are shown in Figure 5.16. As we can see, the experiment without scaling reached the optimal solutions in all runs. On the other hand, the number of the optimal individuals was quite low in all of them. The version with scaling produced much higher number of optimal individuals in some runs, but a few runs did not reach the optimum at all. This results in worse convergence statistics (generation 6.4).

It is obvious from the presented experiments that the NSGA-II substantially outperforms the single-objective algorithm. Not only in the number of optimal individuals, but also in the second objective and the rate of convergence. It is probably due to the selection mechanism and the fact that the single objective EA deems the individuals with the same makespan as equals, whereas NSGA II can compare them using the tasks ASAP objective.

5.2.4 Initialization Comparison

In this section, initialization methods are tested. We used the NSGA-II again. The cycle crossover and precedence preserving shift mutation were used with the same parameters as before. Three methods of initialization are tested: random topology with distance, deterministic precedence and distance and finally random topology.

Random topology with distance

In the previous sections, the initialization was fixed to random topology with distance with probability 0.8. Results are for example in Figure 5.11. Setting the initialization probability to higher values was tried, while other parameters

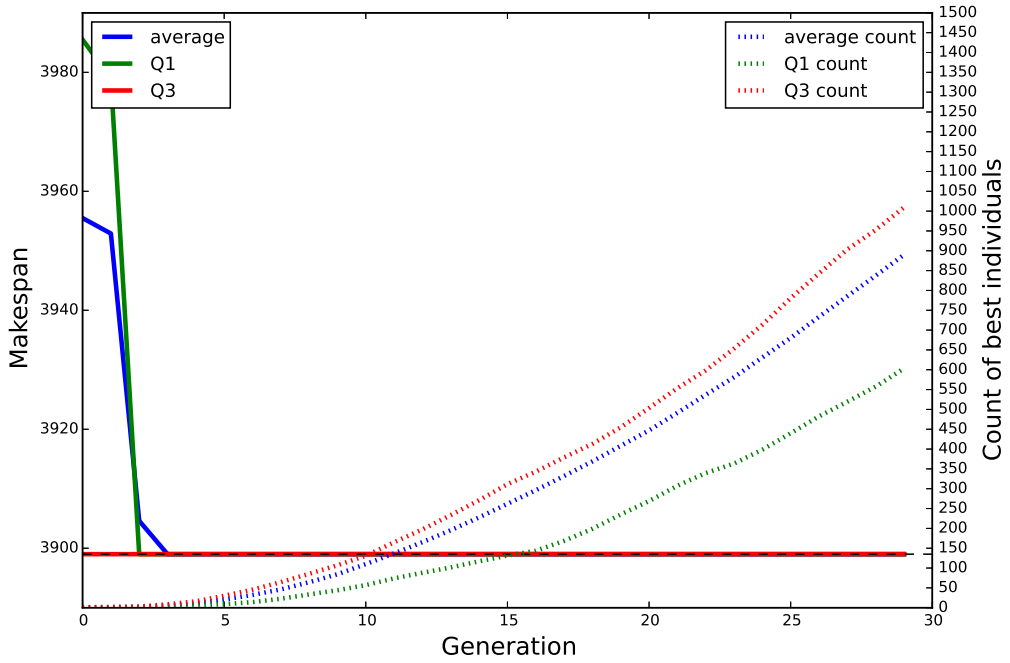


Figure 5.11: Precedence preserving shift mutation - makespan

remained the same (cycle crossover and precedence preserving shift mutation). The results are shown in Figures 5.17 and 5.18. We can see that it works very well, the number of individuals with the optimal makespan increased. Also, the convergence was faster (generation 0.67). The tasks ASAP objective achieved similar values to those with the previous initialization settings.

Precedence and distance

In this section, the operators' ability to recover from homogenous population is tested. The initial population consists solely from the individuals generated by the deterministic *precedence and distance* initialization method. All other parameters remain fixed. Figure 5.19 shows that despite the lack of diversity, the optimal individuals were reached very quickly in all runs (generation 6.2). Furthermore, a large number of optimal individuals was generated. However, the tasks ASAP objective was negatively influenced (Figure 5.20) by a small measure.

Random topology

In the last series of experiments, the initial population was generated using *random topology* initialization method. The successful parameters from previous tests were used again. The optimal makespan was found almost every time at the beginning of the evolution in the previous experiments. This is not the case. Figure 5.21 shows that the makespan was well above the optimum value in this experiment. This result suggests that more exploration of the search space is needed. Therefore, other mutation operators were tried with elevated probabilities. The swap mutation with the parameter *mutation probability* set to 0.8,

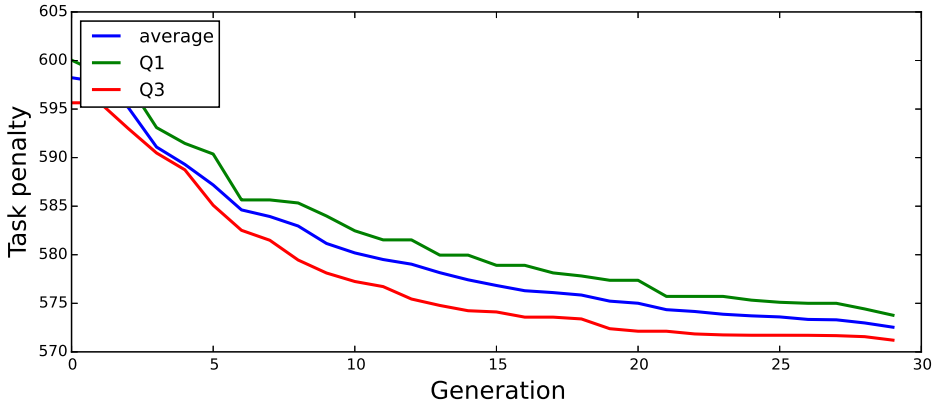


Figure 5.12: Precedence preserving shift mutation - tasks ASAP

gene mutation probability to 0.01 and *diversity trials* to 10 proved to be the most effective. Figure 5.22 shows that some runs reached the optimal makespan. Additionally, all of the other runs got very close to the optimum. Optimal solutions were on average reached after 24.8 generations.

5.3 Topology Mutation

In this section, we shall elaborate on the importance of the proposed topology mutation. In all of the previous experiments, the topology mutation (4.5.1) served as a repair mechanism of the infeasible individuals. The topology mutation was applied after all the other specified operators, thus preventing existence of infeasible individuals in the population. We ran another series of experiments without this operator to demonstrate its significance.

In the first experiment, we ran the NSGA-II with the cycle crossover (2.1.1) and PPSM (4.5.1). All parameters were the same as with the majority of experiments in this chapter. See 5.2.2 for details. The only difference was turning off the topology mutation. See Figure 5.23 and 5.24 for results. The optimal solution was found on average in generation 12.6. The corresponding experiment with the topology mutation found it much sooner - in generation 1.4. It is clear from comparing with Figure 5.11 and 5.12 that the setting with topology mutation outperforms the other one in all measures.

In the second experiment, we changed the PPSM for the swap mutation. We expected a deterioration in results, because the proposed PPSM operator does not create infeasibility if applied on a feasible individual. Figure 5.25 shows the experiment outcome. We can see that a substantial number of runs did not reach the optimum. The average generation of occurrence of the optimal individual was 16.

In the last experiment, the single-objective evolutionary algorithm without the topology was tested. The fitness of the best individual remained the same throughout the evolution in almost every run. In other words, if the optimal solution was not found in the initial population, it would almost never be found. The algorithm apparently did not recover from the large number of infeasible individuals in the population. The reason the NSGA-II did, is probably due to

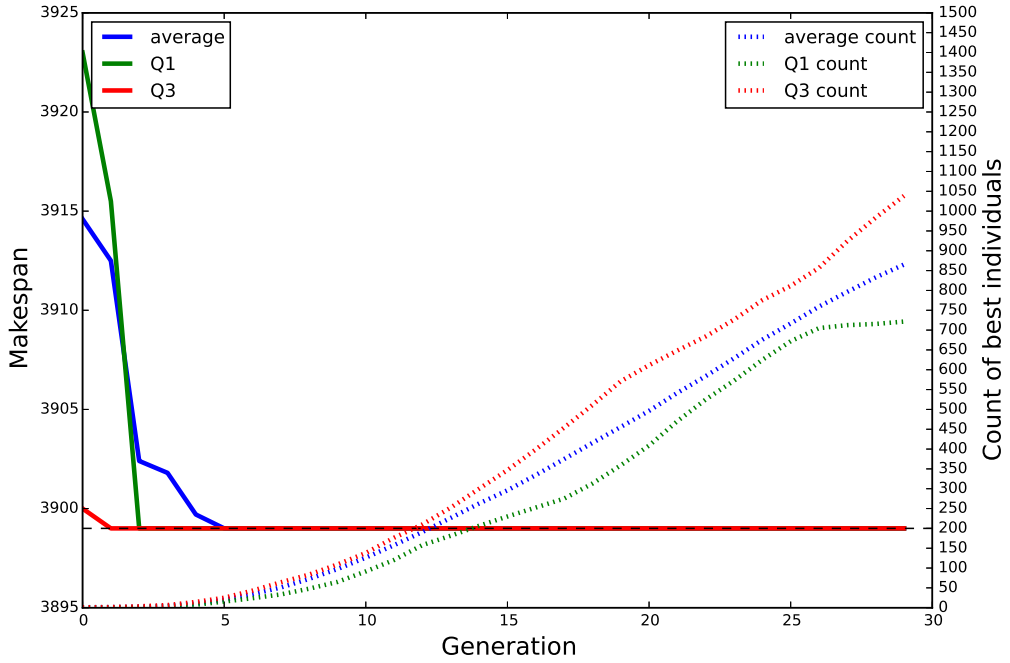


Figure 5.13: No mutation - makespan

elitism which preserved the feasible individuals.

Naturally, the execution time of all the evolutionary runs without the topology mutation substantially increased. It is due to a large number of constraint solver timeouts.

5.4 Running the Experiments

The experiments can be reproduced by running the binaries located on the included CD. The manual how to set up and run an experiment may be found in the Appendix A.

5.5 Summary

We have shown results of several experiments. Optimal or suboptimal solutions were reached even with the random topology initialization. Still, the initialization with heuristics (4.3.2) is useful - it finds the good solutions more quickly and furthermore it finds them in a great number. The heuristic was so successful that the optimal solutions were often found in the initial population. The well-known cycle crossover (2.1.1) alone proved the most effective along with our proposed precedence preserving shift mutation (4.5.1), especially in producing number of distinct individuals with optimal makespan. This combination of genetic operators was also able to find the optimal solutions when the population was initialized homogenously. We have also shown that when initialized by the *random topology*, the best mutation operator is the popular *swap mutation* (2.1.1). The proposed

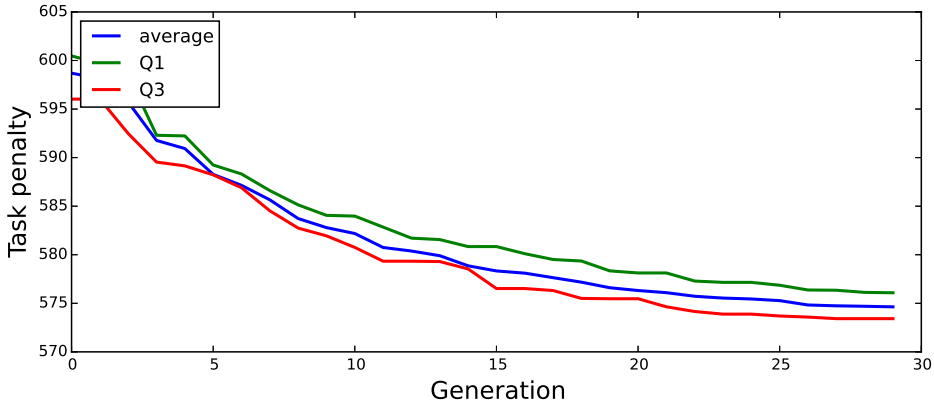


Figure 5.14: No mutation - tasks ASAP

topology mutation (4.5.1) used as a repair mechanism was also tested. It proved to have a significant boost on performance of the evolutionary algorithm. Not only does the evolution find the optimal solutions sooner, if this operator is applied, it also finds greater number of superior individuals. Furthermore, the second objective also benefits from utilization of the topology mutation. Comparing the evolutionary algorithm showed that the simple single-objective evolutionary algorithm was outperformed by the NSGA-II in all experiments. Another outcome of the experiments in the both proposed mutation operators (SPM and PPSM) (4.5.1) seem to converge faster than the traditional swap mutation (2.1.1).

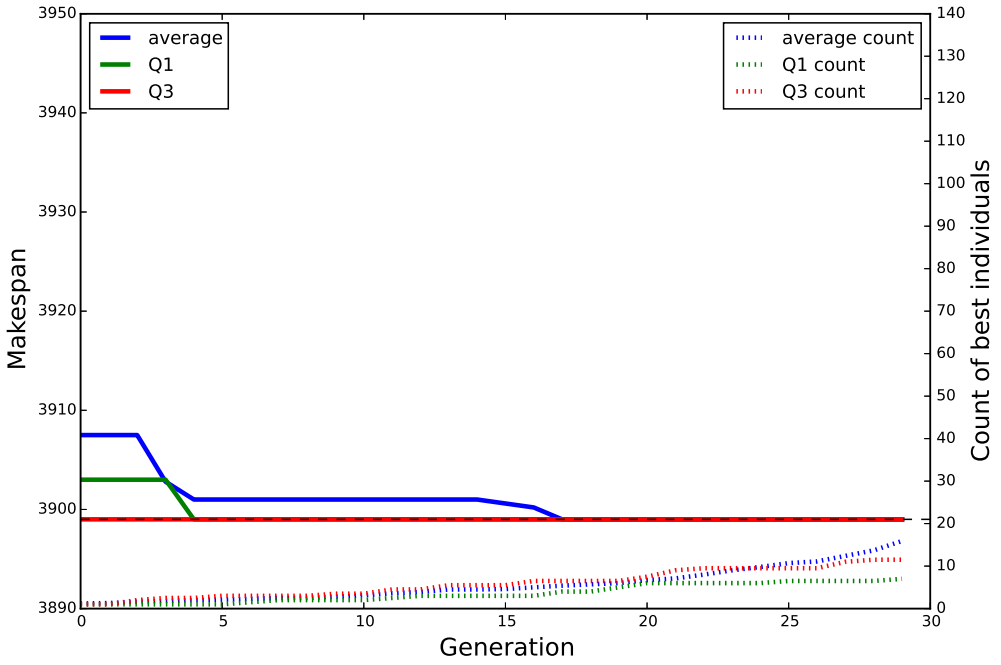


Figure 5.15: Single-objective EA - makespan

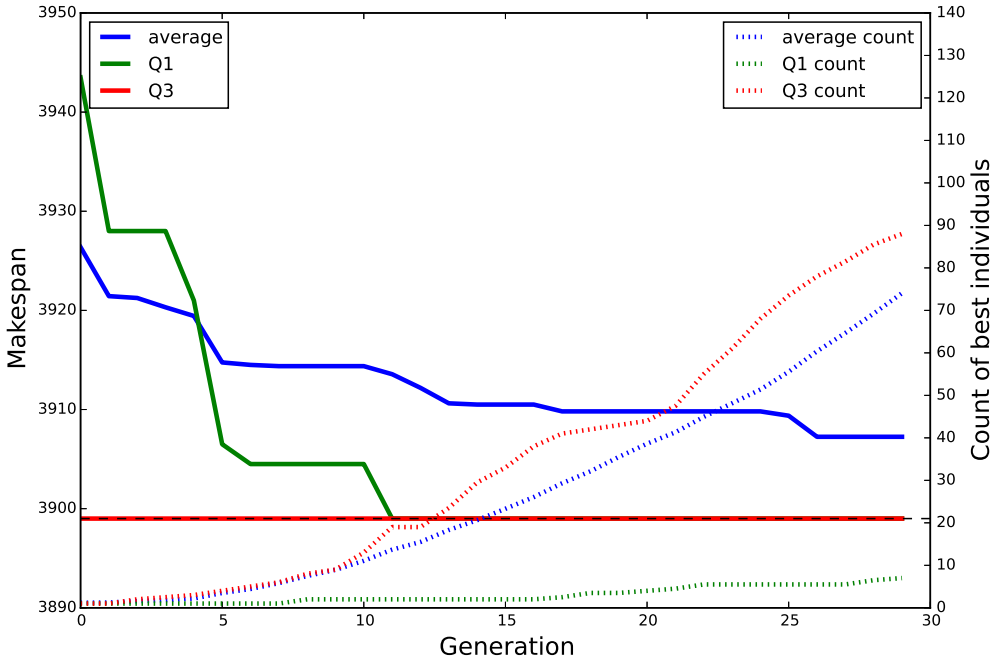


Figure 5.16: Single-objective EA with fitness scaling - makespan

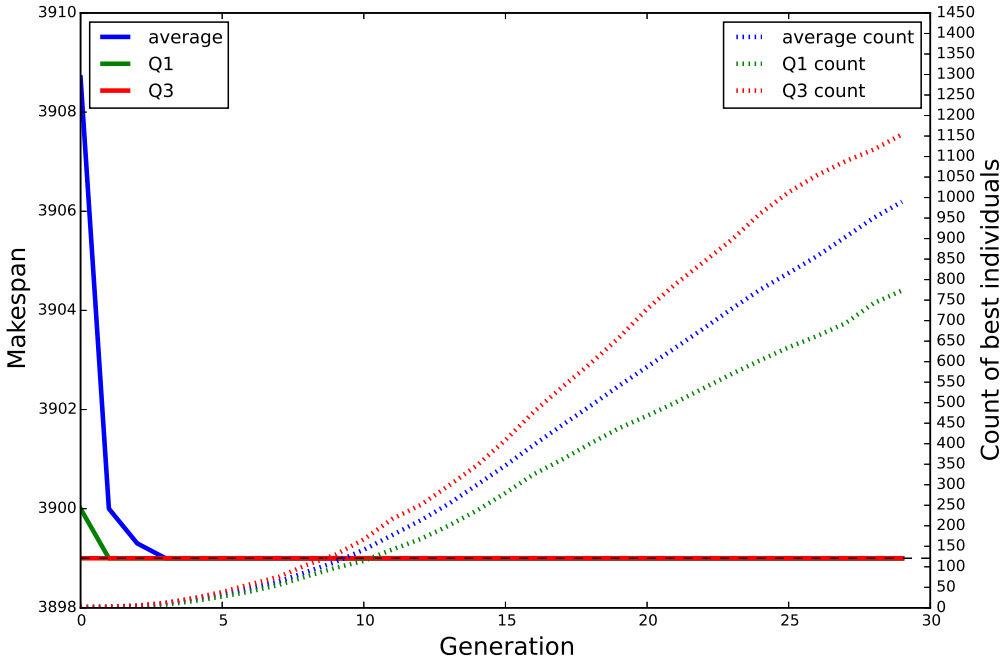


Figure 5.17: Random topology with distance, probability 0.9 - makespan

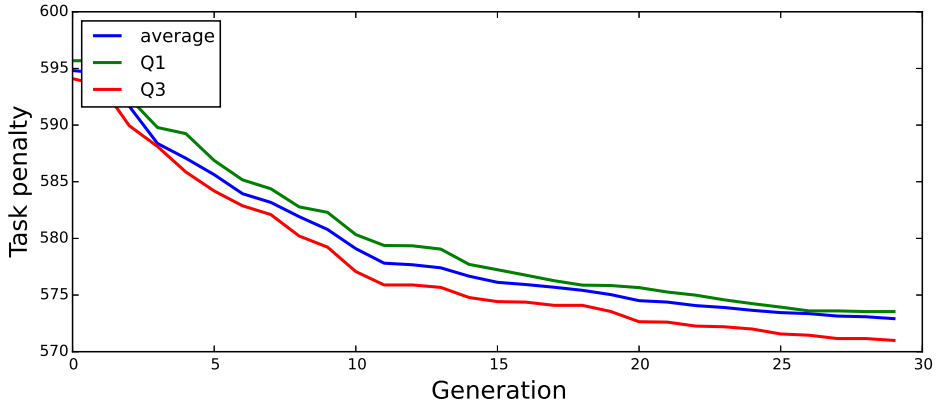


Figure 5.18: Random topology with distance, probability 0.9 - tasks ASAP

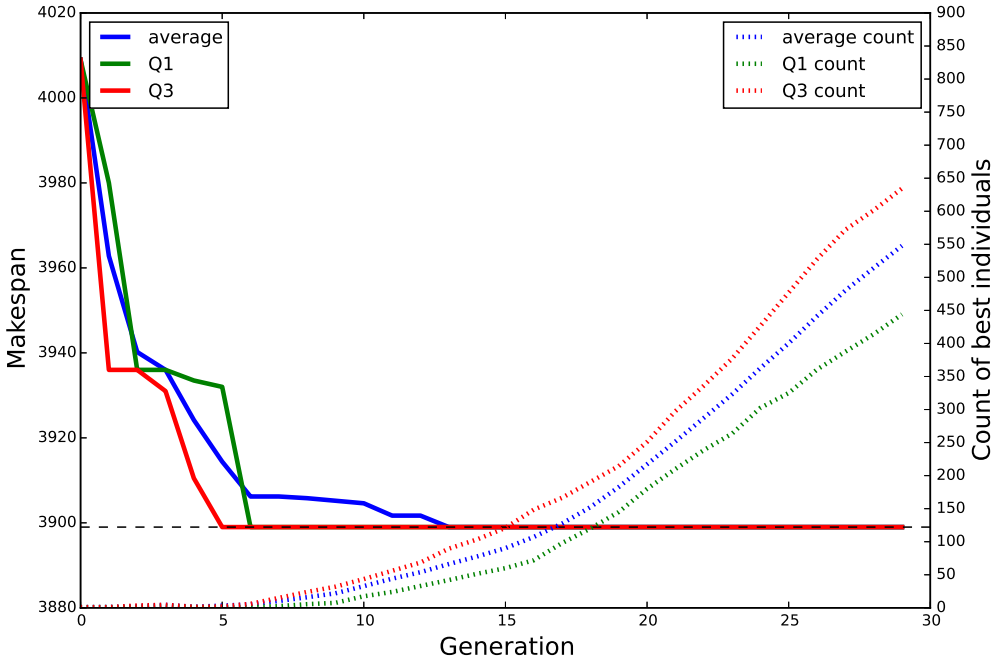


Figure 5.19: Homogenous initialization - makespan

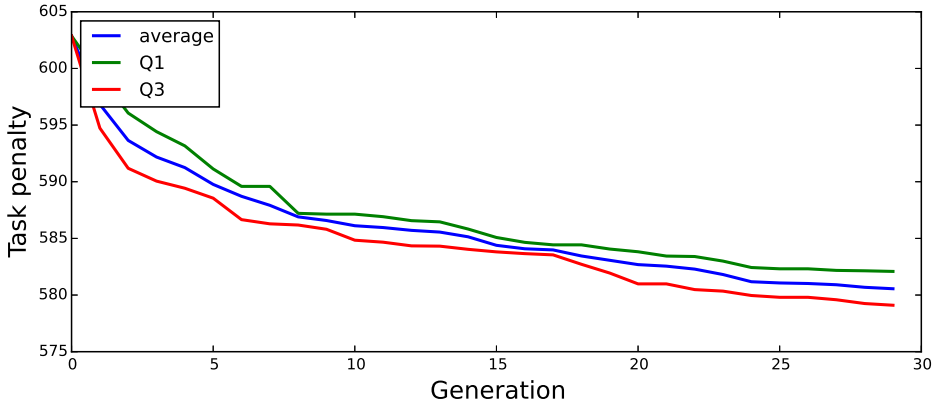


Figure 5.20: Homogenous initialization - tasks ASAP

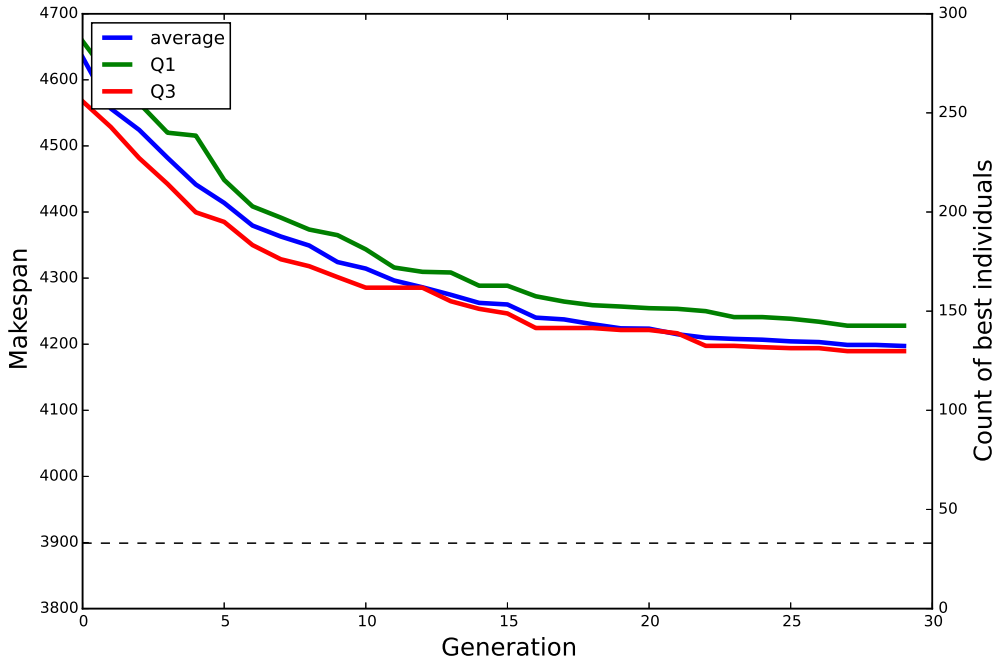


Figure 5.21: Random topology - makespan

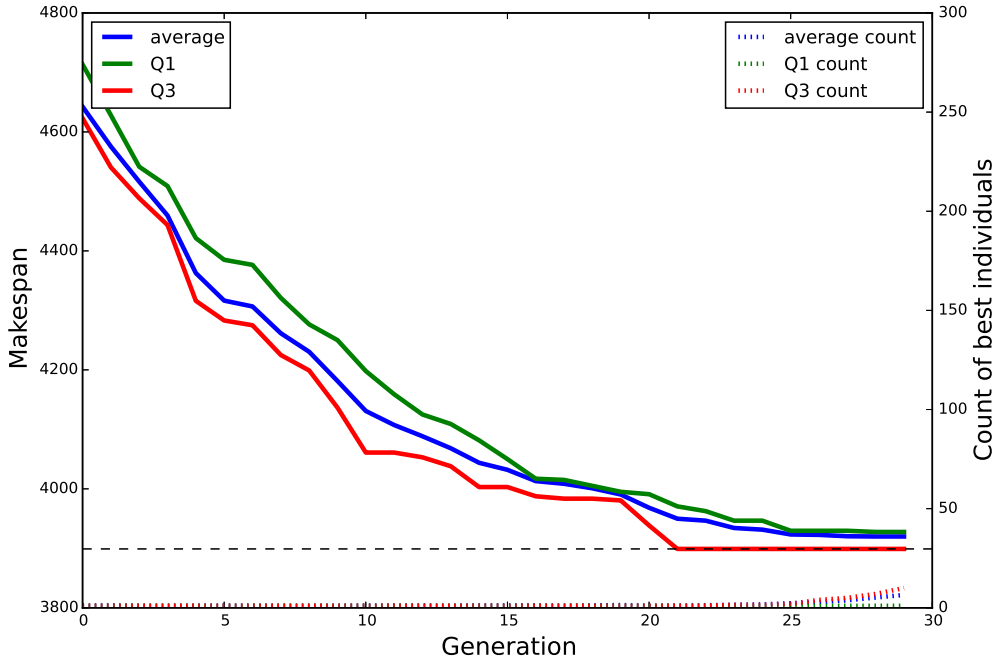


Figure 5.22: Random topology with swap mutation - makespan

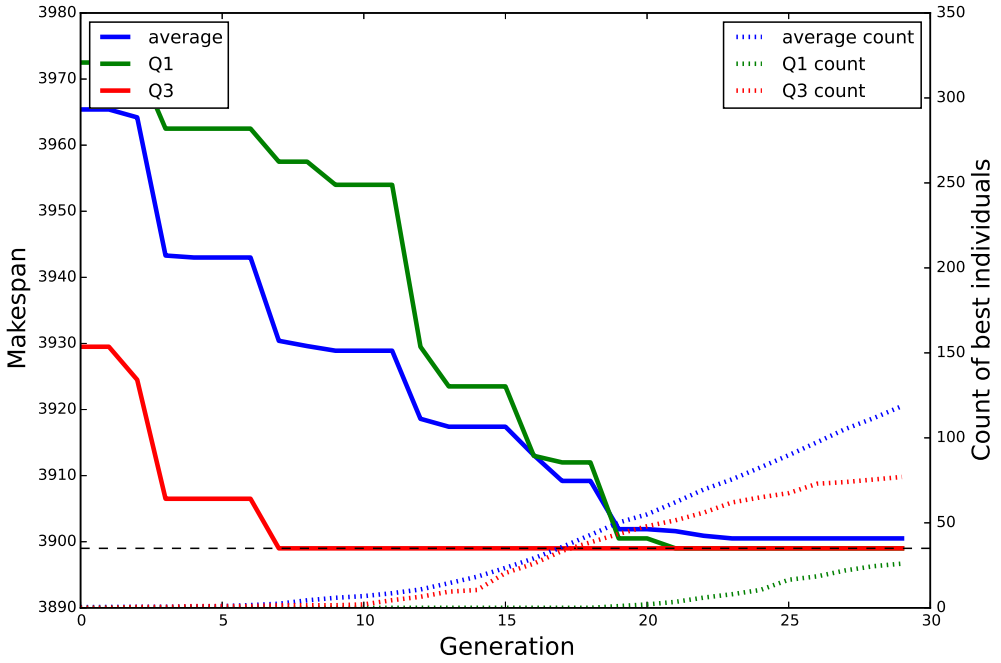


Figure 5.23: No topology mutation with PPSM - makespan

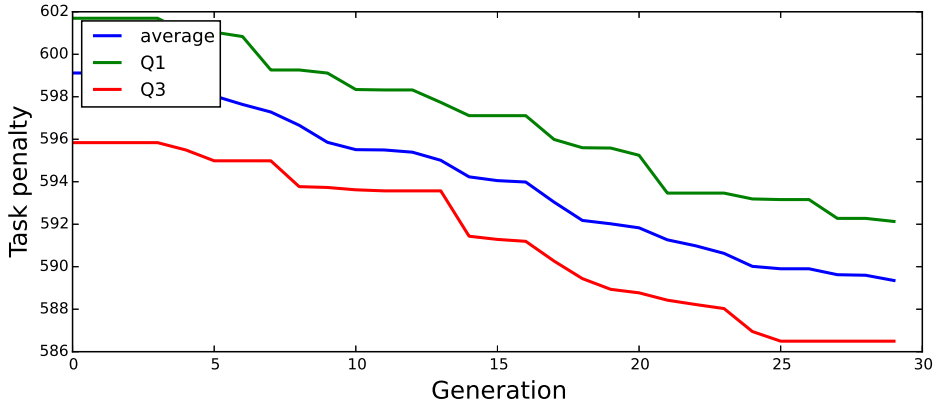


Figure 5.24: No topology mutation with PPSM - makespan

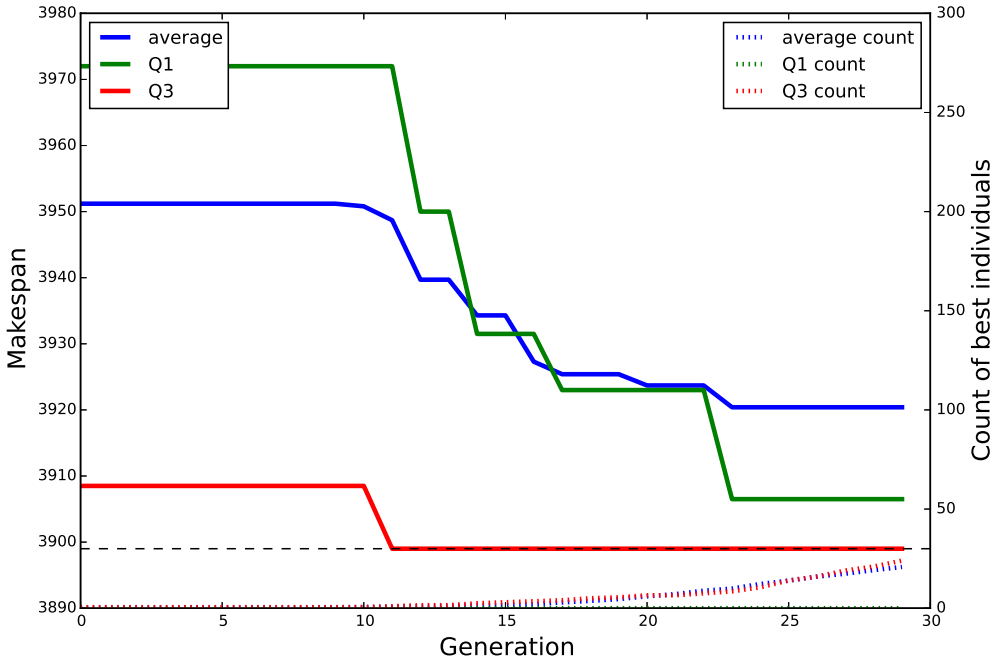


Figure 5.25: No topology mutation with swap mutation - makespan

Conclusion

The scheduling problem was tackled by a combination of evolutionary algorithms and constraint programming in this thesis. We used a novel approach of evolving the variable orderings for the constraint solver. We shall briefly summarize the achieved results.

Initialization methods Multiple methods of initialization were proposed and tested. We showed that the precedence constraints graph has to be analysed in order to produce suitable initial population. Our proposed method of initialization (*random topology with distance*) leads swiftly to optimal solutions in the early generations.

Genetic operators Several new genetic operators were proposed. The most successful ones were the *topology mutation* and *precedence preserving shift mutation*. The topology mutation was used in each generation to repair individuals in each generation, preserving as much original information as possible. It played a major role in the performance of the evolutionary algorithm. The PPSM is a regular mutation operator that shifts genes of the individuals in such a way, than no precedence constraints are broken. The PPSM works particularly well with the cycle crossover. The *segment proportionate mutation* operator was also proposed. It swaps genes of the individual, focusing primarily on the segments at the beginning of the individual. Both operators were successful in finding great numbers of solutions with optimal makespan. Additionally, both operators lead to faster convergence in our experiments than the traditional swap mutation. Furthermore, the proposed genetic operators were able to find the optimal solutions even if inferior initialization methods were used. In the first scenario, random permutations respecting the precedence constraints were used. In the second, we used a homogenous population of individuals with reasonably good makespan. We were successful in both experiments.

Multi-objective optimization We used the NSGA-II for multi-objective evolution. As opposed to the makespan criterion, we optimized another objective which tries to schedule each task to start as early as possible. We were successful in applying the NSGA-II to our problem. It was possible to reach good results in both objectives. Several experiments were performed. In the initialization experiments, for example, there was a visible decline in both objectives if the proposed initialization method was not used.

Future Work

There are several ways which can improve the project presented in this thesis.

Additional Objectives

One of the possible improvements is to add additional objectives to test the NSGA-II capabilities to deal with more criteria. Furthermore, it would likely produce overall better individuals. No more than one or two objectives should be added. The contrary could lead to drastic decrease in the Pareto dominance[59] and the evolution would rely primarily on the crowding distance. One of the candidates for addition is the resource (worker) utilization.

Rescheduling

A real-life production is a very dynamic and non-deterministic environment. It is desirable to make the evolution adapt to these changes. It could be implemented as follows: EA provides an archive of individuals for a single station. This station is then subjected to disruptive events corresponding to the disruptive events statistical models. Several best individuals (according to different objectives) are chosen for the disrupted station. They need to change their genome in order to match the new station's tasks. Indices of tasks that are no longer part of the station are removed from the individual. New tasks indices (if any) are inserted at random positions to the individual. To create a valid individual, the topology mutation should be applied. This way new population is created and the evolution might be run again. If the time is limited or the disruptive events are but a few, it is also possible to simply evaluate a number of transformed individuals and use their resulting schedules.

Schedules Filtering

Another potential improvement is to implement a mechanism that filters vast archives of evaluated individuals and chooses only several representatives. This mechanism should prove useful since many schedules may differ by only a slight shift of a single task. One of the possible ways is to use the critical chain. The critical chain is a set of tasks that results in longest path to project completion. The resources used in critical chain are called critical resources. The idea of filtering is to select only one schedule for each distinct critical resources assignment to critical tasks. The filtered individuals might be used e.g. as rescheduling candidates.

Bibliography

[1] Pinedo Michael L. *Scheduling Theory, Algorithms, and Systems.* Third edition. Springer, 2008. ISBN 978-0-387-78934-7.

[2] ARUM project
`http://arum-project.eu/`

[3] Van Hentenryck, Pascal. *Constraint satisfaction in logic programming.* Vol. 5. Cambridge: MIT press, 1989. ISBN 0-262-08181-4.

[4] Haralick, Robert M., Elliot, Gordon L. *Increasing tree search efficiency for constraint satisfaction problems.* Artificial intelligence, 1980, 14.3: 263-313.

[5] Gaschnig, John. *A constraint satisfaction method for inference making.* In: Proceedings of the Twelfth Annual Allerton Conference on Circuit Systems Theory. 1974. p. 866-874.

[6] Sabin, Daniel, Freuder, Eugene C. *Contradicting conventional wisdom in constraint satisfaction.* In: Principles and Practice of Constraint Programming. Springer Berlin Heidelberg, 1994. p. 10-20.

[7] Dechter, Rina. *Constraint Processing.* First Edition. Elsevier Science, 2003 ISBN-10: 1558608907.

[8] Lomme O. *Consistency Techniques for Numeric CSPs.* Proc. 13th International Joint Conference on Artificial Intelligence, 1993.

[9] Russell, Stuart, Norvig, Peter. *Artificial Intelligence: A Modern Approach.* Third Edition. Pearson, 2009. ISBN 0-13-604259-7.

[10] Wooldridge, Michael. *An Introduction to MultiAgent Systems.* Second Edition. John Wiley & Sons, 2009. ISBN-10: 0470519460.

[11] Gruber, Thomas R. *A translation approach to portable ontology specifications.* Knowledge acquisition 5, no. 2 (1993): 199-220.

[12] CertiCon Company
`http://www.certiconglobal.com/content/company`

[13] JADE framework
`http://jade.tilab.com/`

[14] Fitz-Gibbon, Carol, Taylor. *Performance Indicators.* Multilingual Matters, 1990. ISBN 978-1-85359-092-4.

[15] Choco solver
`http://choco-solver.org/`

[16] JGraphT library
`http://jgrapht.org/`

[17] Garey, Michael R., Johnson, David S. *Computers and intractability: a guide to NP-completeness.* W.H. Freeman, New York, 1979. ISBN:0716710447

[18] Brightwell, Graham R. Winkler, Peter. *Counting linear extensions.* Order 8.3, 225-242, 1991.

[19] Dasgupta, Dipankar, Michalewicz, Zbiniew. *Evolutionary Algorithms in Engineering Applications.* Springer Science & Business Media, 2013. ISBN 978-3-662-03423-1

[20] Klein, Robert. *Scheduling of Resource-Constrained Projects.* Springer US, 2000. ISBN 978-1-4615-4629-0.

[21] Artigues, Christian, Demassey, Sophie Néron, Emmanuel. *Resource-constrained project scheduling: models, algorithms, extensions and applications.* John Wiley & Sons, 2013. ISBN: 9780470611227

[22] Kolisch, Rainer. *Efficient priority rules for the resource-constrained project scheduling problem.* Journal of Operations Management 14.3, 1996, 179-192.

[23] Sakalauskas, Leonidas, Gražvydas, Felinskas. *Optimization of resource constrained project schedules by genetic algorithm based on the job priority list.* Information Technology And Control 35.4, 2015.

[24] Ghallab, Malik, Nau, Dana, Traverso, Paolo. *Automated planning: theory & practice.* Elsevier, 2004. ISBN 9780080490519

[25] Nakano R., Yamada T. *Conventional genetic algorithm for job shop problems* Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, USA, 1991, 474-479.

[26] Kobayashi S., Ono I., Yamamura M. *An efficient genetic algorithm for job shop scheduling problems* Proceedings of the 6th International Conference on Genetic Algorithms. Morgan Kaufmann Publishers Inc., 1995, 506-511.

[27] Bierwirth C. *A generalized permutation approach to job shop scheduling with genetic algorithms.* Operations-Research-Spektrum 17.2-3, 1995, 87-92.

[28] Bierwirth C., Mattfeld D., Kopfer H. *On permutation representations for scheduling problems.* Parallel Problem Solving from Nature–PPSN IV. Springer Berlin Heidelberg, 1996. 310-318.

[29] Giffler B., Thompson G. L. *Algorithms for solving production scheduling problems.* Operations research 8.4, 1960, 487-503.

[30] Yamada T., Nakano R. *A genetic algorithm applicable to large-scale job-shop problems.* 2nd PPSN, 1992, 281-290

[31] Panwalkar S. S., Iskander W. *A survey of scheduling rules.* Operations research 25.1, 1977, 45-61.

[32] Dorndorf U., Pesch E. *Evolution based learning in a job shop scheduling environment.* Computers & Operations Research 22.1, 1995, 25-40.

[33] Adams J., Balas E.,Zawack D. *The shifting bottleneck procedure for job shop scheduling.* Management science 34.3, 1988, 391-401.

[34] Ulder N.L.J., Pesch E., van Laarhoven P.J.M., Bandelt J., Aarts E.H.L. *Genetic local search algorithm for the traveling salesman problem.* Parallel problem solving from nature. Springer Berlin Heidelberg, 1991. 109-116.

[35] Reeves C.R. *Genetic algorithms and neighbourhood search.* Evolutionary Computing. Springer Berlin Heidelberg, 1994. 115-130.

[36] Yamada T., Nakano R. *A fusion of crossover and local search.* Proceedings of The IEEE International Conference on. IEEE, 1996.

[37] Yamada T., Nakano R. *Job-Shop Scheduling by Simulated Annealing Combined with Deterministic Local Search.* Meta-Heuristics. Springer US, 1996. 237-248.

[38] Yamada T., Nakano R. *Genetic Algorithms for Job-Shop Scheduling Problems.* Proceedings of the Modern Heuristics for Decision Support, 1997, 67-81.

[39] Mesghouni K., Hammadi S., Borne P. *On modelling genetic algorithm for flexible job-shop scheduling problem.* Stud. Inform. Contr. J., Vol. 7, No. 1, pp. 37-47

[40] Mesghouni K. *Application des algorithmes évolutionistes dans les problemes d'optimisation en ordonnancement de la production.* (Doctoral dissertation, 1999)

[41] Mesghouni K., Hammadi S.,Borne P. *Evolutionary Algorithms for Job-Shop Scheduling* International Journal of Applied Mathematics and Computer Science 14.1, 2004, 91-104.

[42] Rigi, Mohammad, Amin, Mohammadi, Shariar. *Finding a hybrid genetic algorithm-constraint satisfaction problem based solution for resource constrained project scheduling.* International Conference on Emerging Technologies, 2009, 52-56.

[43] Madureira, Ana and Ramos, Carlos and Silva, do Carmo. *A Genetic Approach for Dynamic Job-Shop Scheduling Problems.* 4 th Metaheuristics International Conference, 2001, 16-20.

[44] Fang, Hsiao-Lan, Ross, Peter, and Corne David. *A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems.* University of Edinburgh, Department of Artificial Intelligence, 1993.

[45] Bierwirth C., Kopfer H., Mattfeld D. C., and Rixen I. *Genetic algorithm based scheduling in a dynamic manufacturing environment.* In Palaniswam, M., editor, Proceedings of the Second Conference on Evolutionary Computation, pages 439-443, IEEE Press, New York, 1995.

[46] Fogel L. J., Alvin J. O., Walsh M. J. *Artificial intelligence through simulated evolution.* John Willy, 1966.

[47] Rechenberg, Ingo. *Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologishen Evolution.* (Doctoral thesis, 1971)

[48] Schwefel, Hans-Paul. *Numerische Optimierung von Computer-Modellen.* (Doctoral thesis, 1974)

[49] Holland, John H. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press, 1992. ISBN: 9780262581110

[50] Koza, John R. *Genetic programming: on the programming of computers by means of natural selection.* MIT press, 1992. ISBN 978-0262111706.

[51] jMetal Framework
`http://sourceforge.net/projects/jmetal/`

[52] Goldberg, David E.,Lingle, Robert. *Alleles, loci, and the traveling salesman problem.* Proceedings of the first international conference on genetic algorithms and their applications. Lawrence Erlbaum Associates, Publishers, 1985.

[53] Whitley, Darrel L., Starkweather, Timothy, Fuquay, D'Ann. *Scheduling problems and traveling salesmen: The genetic edge recombination operator.* ICGA. Vol. 89. 1989.

[54] Oliver I. M., Smith D., Holland, John. *Study of permutation crossover operators on the traveling salesman problem.* Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA. Hillsdale, NJ: L. Erlhaum Associates, 1987.

[55] March, James G. *Exploration and exploitation in organizational learning.* Organization science 2.1, 1991: 71-87.

[56] Schaffer, J. David. *Multiple Objective Optimization with Vector Evaluated Genetic Algorithms.* In: Proceedings of the 1st International Conference on Genetic Algorithms. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1985, pp. 93-100. ISBN: 0-8058-0426-9

[57] Deb, Kalyanmoy, Pratap A., Agarwal S. and Meyarivan T. *A fast and elitist multiobjective genetic algorithm: NSGA-II.* In: Evolutionary Computation, IEEE Transactions on 6.2, 2002, pp. 182-197. ISSN: 1089-778X. doi: 10.1109/4235.996017

[58] Mann, Henry B., Whitney, Donald R. *On a test of whether one of two random variables is stochastically larger than the other.* The annals of mathematical statistics, 1947: 50-60.

[59] Hughes, Evan J. *Evolutionary many-objective optimisation: many once or one many?* Evolutionary Computation, 2005. The 2005 IEEE Congress on. Vol. 1. IEEE, 2005.

List of Tables

List of Abbreviations

- AC - Arc Consistency
- CP - Constraint Programming
- CSP - Constraint Satisfaction Problem
- CSOP - Constraint Satisfaction Optimization Problem
- DG - Disjunctive Graph
- EA - Evolutionary Algorithm
- ES - Evolutionary Strategy
- GA - Genetic Algorithm
- GP - Genetic Programming
- GLS - Genetic Local Search
- HD - Hamming Distance
- JADE - Java Agent Development Framework
- JSSP - Job-Shop Scheduling Problem
- KPI - Key Performance Indicator
- MAS - Multi-agent System
- MSXF - Multi-Step Crossover Fusion
- NSX - Neighbourhood search crossover
- NSGA - Non-dominated Sorting Genetic Algorithm
- OOP - Object-oriented programming
- PME - Parallel Machine Encoding
- PJE - Parallel Jobs Encoding
- PMX - Partially Mapped Crossover
- PPX - Precedence Preservative Crossover
- PPSM - Precedence Preserving Shift Mutation
- RCPSP - Resource-Constrained Project Scheduling Problem
- SB - Shifting Bottleneck
- SMSP - Single Machine Scheduling Problem

- SPM - Segment Proportionate Mutation
- TSP - Travelling Salesman Problem
- SXX - Subsequent Exchange Crossover
- VEGA - Vector Evaluated Genetic Algorithm

A. Appendix

A compact disk is a part of this thesis. It contains source codes, executable files, etc. The source code comprises only the sources written specifically for this thesis. For legal reasons, the code needed for the project was compiled to the executable files.

Contents

- graph - folder containing image of the precedence constraints graph of the sample workstation
- src - evolution related source codes
- station - folder containing problem (workstation) definition
- EvolutionParameters.properties - parameters of the evolution
- MOEA.jar - executable file that invokes the multi-objective evolution run. The corresponding main class *NSGAIIArumRunner.java* is located in the *evolution* package.
- SOEA.jar - executable file that invokes the single-objective evolution run. The corresponding main class *SOEA.java* is located in the *scheduler* package.
- SOEAScaled.jar - executable file that invokes the single-objective evolution. Fitness scaling is used in this version.
- thesis.pdf - electronic version of the text of this thesis
- readme.pdf - a brief document describing the location of the interesting source codes (mainly the genetic operators and initialization methods)

Running the evolution To run the evolution, several condition have to be met:

- Java 8 has to be installed.
- All contents should be copied to a location, where the current user has write permissions.
- The evolution consumes much memory, therefore java should be invoked with the following parameters: `java -d64 -Xms8g -Xmx8g`.

There is a property file associated with the evolution *EvolutionParameters.properties*, most of the properties are self-explanatory. In order to use a genetic operator in the evolution, its parameter *operator_probability* should be set to a positive number less or equal 1. All those with 0 probability are ignored. The boolean *use_human_resources* determines whether the constraint solver considers resource constraints or solely precedence constraints. The property *output_dir* specifies, where the individual archives and

fitness logs are saved. The archive will contain permutations and their filenames have the following semantics: *makespan_asap_hashcode.txt*. The *initialization* parameter serves to determine the initialization method. There are to options: *random* and *hybrid*. The random stands for the *random topology* and hybrid for the initialization method referred to in this text as the *random topology with distance*. The following steps have to completed to run the evolution:

1. Copy the contents to a desired location with write permission
2. Change the current directory to the location
3. Execute i.e. `java -d64 -Xms8g -Xmx8g -jar MOEA.jar`
   optional parameter specifying the parameters might be used. If not, the default properties will be used.

Apart from individuals archive, fitness log is created for each evolutionary run. For each generation, there is one line. Let n be the population size, then the columns semantics is the following: the best makespan in the archive's top n individuals, count of individuals with the best makespan, the average and the worst among the best n individuals. The tasks ASAP objective is described by the other four columns.