UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1225

EVOLUTION OF SCHEDULING HEURISTICS FOR THE RESOURCE CONSTRAINED SCHEDULING PROBLEM

Dominik Šišejković

Zagreb, June 2016.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 9 March 2016

MASTER THESIS ASSIGNMENT No. 1225

Student: **Dominik Šišejković (0036466662)**
Study: Computing
Profile: Software Engineering and Information Systems

Title: **Evolution of scheduling heuristics for the resource constrained scheduling problem**

Description:

Describe the problem of resource constrained project scheduling and existing algorithmic approaches. Design a hyper-heuristic model to evolve appropriate scheduling heuristics for the problem using genetic programming. Develop a system for experimentation with different solution representations and problem constraints. Investigate the effectivenes of the proposed approach considering existing scheduling heuristics, especially in dynamic conditions. Enclose program codes, results with proper explanation and resources to the thesis.

Issue date: 18 March 2016
Submission date: 1 July 2016

Mentor:
Associate Professor Domagoj Jakobović, PhD

Committee Chair:
Full Professor Krešimir Fertalj, PhD

Committee Secretary:
Assistant Professor Igor Mekterović, PhD

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 9. ožujka 2016.

Predmet: **Diplomski rad**

DIPLOMSKI ZADATAK br. 1225

Pristupnik: **Dominik Šišejković (0036466662)**
Studij: Računarstvo
Profil: Programsko inženjerstvo i informacijski sustavi

Zadatak: **Evolucija heuristika raspoređivanja za raspoređivanje s ograničenim sredstvima**

Opis zadatka:

Opisati problem raspoređivanja projekata s ograničenim sredstvima. Oblikovati hiper-heuristički model za evoluciju prikladnih heuristika raspoređivanja za zadani problem uz pomoć genetskog programiranja. Razviti programski sustav za ispitivanje različitih prikaza rješenja i ograničenja problema. Ispitati učinkovitost razvijenog pristupa s obzirom na postojeće heuristike, posebno u dinamičkim uvjetima raspoređivanja. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 18. ožujka 2016.
Rok za predaju rada: 1. srpnja 2016.

Mentor:

Izv. prof. dr. sc. Domagoj Jakobović

Predsjednik odbora za diplomski rad profila:

Prof. dr. sc. Krešimir Fertalj

Dielovođa:

Doc. dr. sc. Igor Mekterović

I would like to express my sincere gratitude to my advisor prof. dr. sc. Domagoj Jakobović for his patience, understanding, motivation and continuous support for this master's thesis and overall studies. You have been an outstanding mentor and friend.

Special thanks to my family. Words can not express how grateful I am for your support, advice and love. Thank you.

CONTENTS

1. Introduction

Scheduling can be defined as a decision-making process that deals with the allocation of resources to tasks under specific conditions with the goal to optimize one or more objectives. Scheduling is widely used on a regular basis in a variety of tasks ranging from large scale problems in manufacturing, industrial and service industries to highly specific resource management especially in computer-based systems. Resources, tasks and objectives can take different forms making scheduling a very interesting and demanding problem field.

During the recent decades a substantial amount of research was directed towards exploring efficient techniques of dealing with scheduling-based problems. Though usually complex and heavy in resource demands, these approaches very often try to find suitable solutions to only concrete problem instances without focusing on generating reusable and efficient tools to cope with scheduling demands from a more general perspective.

In this thesis the problem of scheduling tasks is addressed by means of genetic programming with focus on the resource constrained scheduling problem as a mathematical model. As part of a very large body of research called machine learning, genetic programming is used to learn and evolve suitable scheduling heuristics to be applied efficiently to generate feasible schedules for a larger set of problem instances taking performance and solution reusability into account.

In the first chapter of this work, the resource constrained scheduling problem is presented alongside with properties definitions, constraints, complexity analysis and different solving approaches. The second chapter presents genetic programming as a nature inspired machine learning technique. The application of genetic programming to the concrete problem domain is described in the third chapter including a detailed overview of the devised hyper-heuristic problem solving model. The fourth chapter focuses on the experimental setups and procedures used in the evolution of scheduling heuristics. Achieved results with proper explanation and effectiveness analysis are given in the fifth chapter. At last, a conclusion is given regarding the devised heuristic

approach with possible future work extensions and ideas.

2. RCSP

Before describing the problem this thesis focuses on, it is important to mention the very basics of scheduling theory which is the critical path method (CPM) [8] that emerged in the late 1950s. CPM is an activity network method with activity-on-node representation commonly used with all forms of projects. This method assumes unlimited availability of resources which leads to great underestimation of project duration in total. However, this method is used even today as a basis for making scheduling decisions especially for activities' properties calculations.

One extension of the CPM model can be made by adding resource constraints which define the resource constrained (project) scheduling problem, commonly abbreviated as RCSP or RCPSP.

2.1. Problem Definition

In general, the resource constrained scheduling problem considers activities and resources. Activities have known durations and resource demands, while resources are of limited availability. Additionally, activities are linked by precedence relations. The problem consists of finding a feasible schedule with minimal total duration by assigning start times to each activity such that the precedence and resource constraints are satisfied.

More formally, the RCPSP can be defined as a combinatorial optimization problem in form of a tuple [2]:

$$RCPSP = (A, E, p, R, B, D, c). \tag{2.1}$$

All *activities* constituting the project are defined by the set $A = \{A_0, \ldots, A_{n+1}\}$. By convention the activities A_0 and A_{n+1} represent the start and the end of a schedule. These activities are usually referred to as dummy activities. The set of non-dummy activities is identified by $A' = A \setminus \{A_0, A_{n+1}\}$.

Durations are represented by the vector $p \in \mathbb{N}_0^{n+2}$ where p_i is the duration of activity A_i. A special case applies for both dummy activities: $p_0 = p_{n+1} = 0$.

The *precedence relation* is given by the set E which defines pairs such that $(A_i, A_j) \in E$ means that activity A_i precedes activity A_j. Additionally, we assume that A_0 precedes all other activities and that A_{n+1} succeeds all other activities.

Generally, resources can be categorized as *renewable*, *non-renewable* and *doubly-constrained* [16]. Renewable resources are available at any given time with full replenishment capacity. Non-renewable resources are available only in predetermined amounts without replenishment capability. Doubly-constrained resources are limited per each time period and for the whole project duration. In this thesis we shall work exclusively with renewable resources which can be formalized by the set $R = \{R_1, \ldots, R_r\}$ where the resources' availabilities are defined as vector $B \in \mathbb{N}^r$.

$D \in \mathbb{N}_0^{(n+2)\times r}$ defines activity *demands* on resources with dummy activities having demands of 0 for all resources. In order for an activity to execute, all demands have to be satisfied.

The *objective function* is defined as $c : \chi \to \mathbb{R}$. This function maps from the feasible schedule space to real values. In case the tuple member is omitted, it is assumed that the objective function is the project makespan.

The solution of RCPSP is schedule $S \in \chi$ in $\mathbb{R}^{n+2}$ such that S_i represents the start time of activity A_i. $C_i = S_i + p_i$ denotes the completion time of activity A_i. As we assume that $S_0 = 0$, the schedule makespan equals to S_{n+1}. As already mentioned, a solution S is feasible only if the precedence (2.2) and resource constraints (2.3) are satisfied, where $A_t = \{A_i \in A \mid S_i \leq t \leq S_i + p_i\}$ is the set of concurrently running activities at a given time t.

$$S_j - S_i \geq p_i \quad \forall (A_i, A_j) \in E \tag{2.2}$$

$$\sum_{A_i \in A_t} D_{ij} \leq B_j \quad \forall t \geq 0, \forall R_j \in R \tag{2.3}$$

The above defined constraints and the set A_t state that an activity can not be interrupted once it is started. This property is referred to as not allowing *preemption*. Taking this into account, RCSP can further be defined as follows:

Definition 2.1. The RCPSP is the problem of finding a non-preemptive schedule S of minimal makespan S_{n+1} subject to precedence constraints (2.2) and resource constraints (2.3).

2.1.1. Computational complexity

According to the computational complexity theory, the RCPSP is one of the most intractable combinatorial optimization problems. RCPSP belongs to the class of problems that are *NP-hard in the strong sense* [4] meaning that its decision version is *NP-complete*. Here the decision version is defined as follows:

Definition 2.2. The decision variant of the RCPSP is the problem of determining whether a schedule S of makespan S_{n+1} not greater than H subject to precedence and resource constraints exists or not.

If no final time at which all activities must be finished is given then the RCPSP is not *NP-hard* (in the strong sense) as in this case a feasible solution can be found in polynomial time. One of the approaches would include scheduling all activities according to precedence constraints.

2.2. Properties Definition

For further problem analysis and implementation it is important to mention the basic properties of RCPSP. For each activity in the problem instance there is a variety of properties that can be calculated to define the *time-frame* in which a specific activity can be scheduled.

Let the set of all activities that can be scheduled at a given time t be defined as $E(t)$ - the set of eligible activities at time t. To calculate the set, the following properties are needed:

- ES_j - the earliest start time of activity j
- EF_j - the earliest finish time of activity j
- LS_j - the latest start time of activity j
- LF_j - the latest finish time of activity j

To calculate the above mentioned variables the total project time ($horizon$) T must be known in advance. This value is either given in the problem description or it can be calculated with any appropriate heuristic [9] or with the following simple formula:

$$T = \sum_{j=1}^{n} p_j. \tag{2.4}$$

To calculate the values of ES_j and EF_j we set $ES_0 = EF_0 = 0$, while the other values are calculated as follows:

$$ES_j = max\{EF_i : i \in P_j\} \tag{2.5}$$

$$EF_j = ES_j + p_j, j \in \{1, \ldots, n\}, \tag{2.6}$$

where P_j denotes the set of activities that directly precede the activity with index j.

The procedure for calculating the values of LS_j and LF_j is similar, but with the difference that the recursive calculation starts form the upper bound T with $LF_n = LS_n = T$ by using the following expressions:

$$LF_j = min\{LS_i : i \in F_j\} \tag{2.7}$$

$$LS_j = LF_j - p_j, j \in \{n-1, \ldots, 1\}, \tag{2.8}$$

where F_j denotes the set of activities that directly succeed the activity with index j. One can note that the aforementioned calculation procedures assume unlimited resources and rely only on the precedence relations.

Finally we can define $E(t)$ as follows:

$$E(t) = \{j : j \in A, ES_j + 1 \leq t \leq LF_j\}. \tag{2.9}$$

As can be seen from the above expression, each activity A_j can be scheduled and executed in the given time frame between its earlier start and latest finish time where we assume integer time values.

The concrete difficulty of a RCPSP instance depends upon many different parameters of which the following are mentioned as most important in literature [6]:

1. Network complexity - NC

   Defined as the average number of non-redundant arcs per node (including the super-source and -sink nodes) [13].

2. The effect of resources:

   (a) *Resource factor* - RF_r

   Defines the average activities' needs for resources. This factor is defined as follows:

$$RF_r = \frac{1}{|A|} \cdot \frac{1}{|R|} \sum_{j \in A} \sum_{r \in R} \begin{cases} 1, & u_{jr} > 0 \\ 0, & otherwise. \end{cases} \tag{2.10}$$

If the resource factor equals 1, each activity demands at least some amount of each resource. If the factor equals 0, activities do not demand any amount of any resource, therefore the problem collapses to a scheduling problem without resources.

(b) *Resource strength* - RS_r

The resource strength factor connects the demands for resources with the actual resources' availabilities. This factor is defined as follows:

$$RS_r = \frac{a_r}{\frac{1}{|A|} \cdot \sum_{j \in A} u_{jr}}, \tag{2.11}$$

where a_r is the availability of resource with index r and u_{jr} the demand for the resource r of activity j. If the resource strength equals 0 then there is no available amount of resource r. Otherwise if it equals 1, the amount of available resources allows the problem to be treated as a scheduling problem without resources. It is important to note that some problem instances are generated with the additional property that each resource has the same resource factor. In that scenario RS_r becomes a project-specific property.

It is important to note the correlation between the given factors and the execution time. Experiments conducted by *Kolisch et al.* [13] show that a negative but very weak correlation exists between the network complexity and the program execution time, while a great magnitude of a positive correlation between the resource factor and execution time exists as for a negative correlation between the resource strength and the execution time.

2.3. Schedule Properties

To ease the presentation of schedule generation schemes and the understanding of gained results a number of terms and definitions concerning different types of schedules are introduced in this section. All given definitions are referenced from [9].

Definition 2.3. A *complete* schedule (CS) assigns a schedule starting and finishing time $SS_j(CS)$ ($SF_j(CS) := SS_j(CS) + p_j$) to each activity $j = 1, \ldots, n$, i.e. activity j is performed in the periods $SS_j + 1(CS), \ldots, SF_j(CS)$.

Definition 2.4. CS is called *feasible* if the precedence and resource constraints are satisfied for all scheduled activities included in the schedule.

Table 2.1: Notation Description

Notation	Definition
CS	feasible complete schedule
PS	feasible partial schedule
SS_j	starting time for activity j
SF_j	finish time for activity j
p_j	duration of activity j

Definition 2.5. A (feasible) *partial* schedule (PS) is a schedule in which the starting times are only set for a subset all activities.

Definition 2.6. A *local left shift* of an activity j consists of a sequence of starting time reductions by one period with each intermediate schedule being feasible. If a feasible starting time $SS'_j < SS_j$ for an activity j exists but can not be obtained by a local shift, then assigning SS'_j to activity j represents a *global left shift*.

The defined local and global left shifts yield two classes of schedules:

Definition 2.7. *Semi-active* feasible schedules are schedules for which no further local left shifts are possible. A subset of this set is build by the *active* schedules which are schedules for which no further global left shift is possible. In terms of start times, an active schedule is a schedule where none of the activities can be started earlier without delaying some other activity (activity preemption is not allowed).

Definition 2.8. A *non-delay* schedule is a schedule for which there is no time point t at which an eligible activity j can be started (in addition to the active ones) neglecting the resource constraints for the interval $[t, t + p_j]$. That is, in a non-delay schedule no activity can be started earlier without delaying some other activity even if activity preemption is allowed.

Here it is important to mention that the set of non-delay schedules is a subset of active schedules (2.1), therefore it is on average of smaller cardinality. Also the set of non-delay schedules may not contain an optimal solution. In other words, one can not guarantee to find an optimal solution if considering only non-delay schedules.

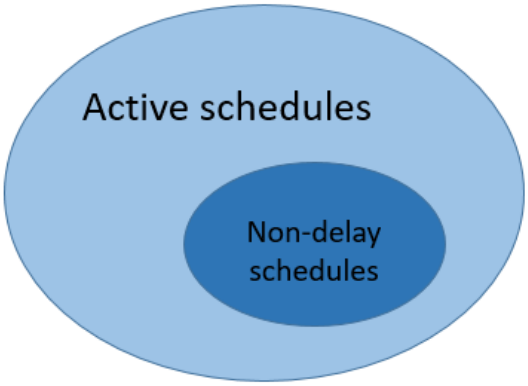

Figure 2.1: Schedule type subsets.

2.4. Schedule Generation Schemes

Schedule generation schemes (SGS) are used to generate feasible schedules out of given project instances and activities' priorities (if applicable). The generation schemes are core ingredients of most heuristic solution procedures. The general idea is that the SGS builds a schedule from scratch taking resource and precedence constraints into account where the construction can take place in three different directions. In this work, only the forward planning direction is used for schedule generation (see 2.4.3). During the building process, the SGS upgrades *partial schedules* until all activities are scheduled and a complete feasible schedule is generated. A partial schedule is therefore a schedule where only a subset of the $n + 2$ activities have been scheduled.

In the literature, two different schedule generation schemes are available: the *serial* SGS and the *parallel* SGS. Both schemes generate feasible schedules but differ in the way activities and time is handled throughout the procedures.

As can be seen in sections 2.4.1 and 2.4.2, both algorithms include a moment where it is necessary to decide which activity will be considered next for scheduling out of the current (eligible) set. In general, the first activity in the set is taken. But if priority scheduling is applied, then the process of picking an activity relies on its relative priority in the given set.

Both the serial and parallel SGS are presented in the following sections each with one concrete example. Both examples are based on the project instance given in figure 2.2.

Each activity is presented with its $id \in \{0, \ldots, 8\}$, duration p_j and resource demand $r_{j,1}$. For the purpose of simplicity, this project instance contains only one re-

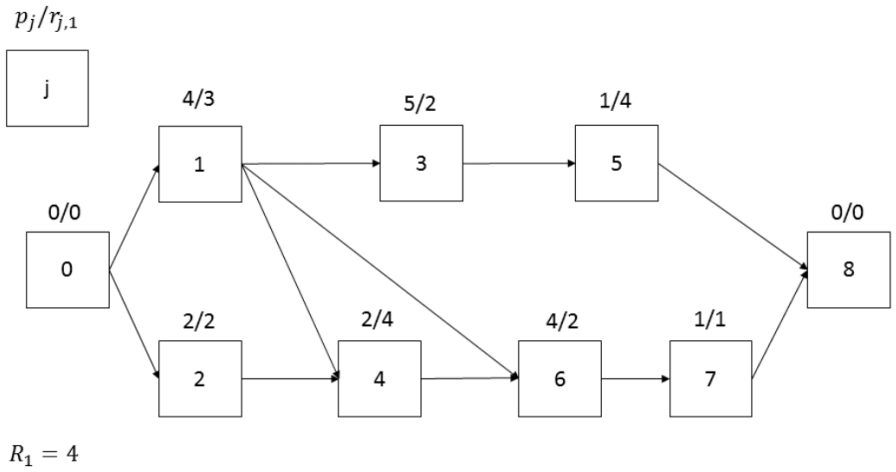

Figure 2.2: Example project instance.

source $R_1 = 4$.

2.4.1. Serial Schedule Generation Scheme

The serial SGS is based on *activity-incrementation* meaning that there is no actual time variable present during the construction of a schedule. The algorithm consists of $g = 1, \ldots, n$ stages where n is the number of activities. In each stage one activity is selected for scheduling out of the set of eligible activities. During the selection process, eligibility concerns only the precedence constraint while the resource constraints are considered later in the algorithm.

Each stage is associated with two disjoint activity sets: S_g and E_g. The set S_g comprises all activities that are already scheduled and part of the current partial schedule in the stage g. The set E_g comprises all activities eligible for scheduling in stage g. Note that the conjunction of S_g and E_g does not always give the set of all activities as some activities are not eligible for scheduling at this stage. Let J be the set of all activities and let $A(t) = \{j \in J \mid F_j - p_j \leq t \leq F_j\}$ be the set of activities which are being processed (active) at the time instant t. Now we can define $\widetilde{R}_k(t) = R_k - \sum_{j \in A(t)} r_{j,k}$ as the remaining capacity of resource type k at the time moment t and $F_g = \{F_j \mid j \in S_g\}$ as the set of all finish times. Let further be $E_g = \{j \in J \setminus S_g \mid P_j \subseteq S_g\}$ the set of eligible activities where P_j is the set of immediate predecessors of activity with index j. Finally, let $K = \{1, \ldots, K\}$ be the set of resource types. The serial SGS is described in algorithm 1 [12].

During the initialization, the dummy activity $j = 0$ is assigned a completion time

Algorithm 1 Serial SGS

Initialize: $F_0 = 0, S_i = \{0\}$

for $g = 1$ to n **do**

Calculate: $E_g, F_g, \widetilde{R}_k(t) \quad (k \in K); t \in F_g$

Select one $j \in E_g$

$EF_j = \max_{h \in P_j}\{F_h\} + p_j$

$F_j = \min\{t \in [EF_j - p_j, LF_j - p_j] \cap F_g \mid r_{j,k} \leq \widetilde{R}_k(\tau),$
$k \in K, \tau \in [t, t + p_j] \cap F_g\} + p_j$

$S_g = S_{g-1} \cup j$

end for

of 0 before being inserted into the partial schedule. At the beginning of each step the set of eligible activities E_g, the set of finish times F_g and the remaining capacities $\widetilde{R}_k(t)$ where $t \in F_g$ are calculated and updated. After the initialization, one activity j is selected from the eligible set. To be able to find the right finish time of the selected activity it is necessary to calculate the earliest precedence finish time EF_j of that activity. Note that this earliest finish time is not a static value as it is determined only based on the set of already scheduled activities. Now it is possible to calculate the earliest precedence- and resource-feasible finish time F_j within the time frame $[EF_j, LF_j]$. The latest finish time is calculated by backward recursion from an upper bound of the project's finish time T [5]. This value must be determined beforehand.

The serial SGS generates active schedules as shown by *Kolisch* [11] meaning that using this approach one is guaranteed to be able to generate an optimal schedule, as described in section 2.3 of this chapter. The time complexity of the serial SGS is $\mathcal{O}(n^2 \cdot |K|)$ [15].

For the purpose of easier understanding the serial SGS given in algorithm 1, a full example is presented in figure 2.3. Activities' priorities are given in table 2.2, while the pre-calculated latest finish values are presented in table 2.3. These latest finish values are calculated by using the expression 2.7.

Table 2.2: Priority values.

Activity index j	1	2	3	4	5	6	7
Priority value	2	7	3	4	6	5	1

Table 2.3: Latest finish times.

Activity index j	1	2	3	4	5	6	7
LF_j	12	12	18	14	19	18	1

#1) g = 1

$S_1 = \{0\}, E_1 = \{1, 2\}$

$j = 2$

$EF_2 = max_{h \in P_2}\{0\} + 2 = 2$

$F_2 = \min\{[2-2, 12-2] \cap \{0\}\} + 2 = 2$

$F_0 = \{0, -, -, -, -, -, -, -\}$

t	R	
0	4	T

$F_1 = \{0, -, 2, -, -, -, -, -\}$

#2) g = 2

$S_2 = \{0, 2\}, E_2 = \{1\}$

$j = 1$

$EF_1 = max_{h \in P_2}\{0\} + 2 = 2$

$F_1 = \min\{[0, 8] \cap \{0, 2\}\} + 4 = 2 + 4 = 6$

$F_1 = \{0, -, 2, -, -, -, -, -\}$

t	R	
0	2	F
2	4	T

$F_2 = \{0, 6, 2, -, -, -, -, -\}$

#3) g = 3

$S_3 = \{0, 1, 2\}, E_2 = \{3, 4\}$

$j = 4$

$EF_4 = max_{h \in 4}\{2, 6\} + 2 = 8$

$F_4 = \min\{[6, 12] \cap \{0, 2, 6\}\} + 2 = 6 + 2 = 8$

$F_2 = \{0, 6, 2, -, -, -, -, -\}$

t	R	
6	4	T

$F_3 = \{0, 6, 2, -, 8, -, -, -\}$

#4) g = 4

$S_4 = \{0, 1, 2, 4\}, E_2 = \{3, 6\}$

$j = 6$

$EF_6 = max_{h \in 6}\{6, 8\} + 4 = 12$

$F_6 = \min\{[8, 14] \cap \{0, 2, 6, 8\}\} + 4 = 8 + 4 = 12$

$F_3 = \{0, 6, 2, -, 8, -, -, -\}$

t	R	
8	4	T

$F_4 = \{0, 6, 2, -, 8, -, 12, -, -\}$

#5) g = 5

$S_5 = \{0, 1, 2, 4, 6\}, E_5 = \{3, 7\}$

$j = 3$

$EF_3 = max_{h \in 3}\{6\} + 5 = 11$

$F_3 = \min\{[6, 13] \cap \{0, 2, 6, 8, 12\}\} + 5 = 8 + 5 = 13$

$F_4 = \{0, 6, 2, -, 8, -, 12, -, -\}$

t	R	
6	0	F
8	2	T

$F_5 = \{0, 6, 2, 13, 8, -, 12, -, -\}$

#6) g = 6

$S_6 = \{0, 1, 2, 3, 4, 6\}, E_6 = \{5, 7\}$

$j = 5$

$EF_5 = max_{h \in 5}\{13\} + 1 = 14$

$F_5 = \min\{[13, 18] \cap \{0, 2, 6, 8, 12, 13\}\} + 1 = 13 + 1 = 14$

$F_5 = \{0, 6, 2, 13, 8, -, 12, -, -\}$

t	R	
13	4	T

$F_6 = \{0, 6, 2, 13, 8, 14, 12, -, -\}$

#7) g = 7

$S_7 = \{0, 1, 2, 3, 4, 5, 6\}, E_7 = \{7\}$

$j = 7$

$EF_7 = max_{h \in 7}\{12\} + 1 = 13$

$F_7 = \min\{[12, 18] \cap \{0, 2, 6, 8, 12, 13, 14\}\} + 1 = 12 + 1 = 13$

$F_6 = \{0, 6, 2, 13, 8, 14, 12, -, -\}$

t	R	
12	2	T
13	2	F

$F_7 = \{0, 6, 2, 13, 8, 14, 12, 13, 14\}$

Figure 2.3: Serial SGS example.

For the above example it is important to notice that for each calculation of the finish time F_j a full "table search" should be done which can be performed in $\mathcal{O}(n)$ time where n is the number of entries. In other words, it is necessary to check for a selected activity j whether there are enough resource available in all time periods covered in the interval $[time, time + p_j]$, where $time$ is a moment resulting from the intersection $[EF_j - p_j, LF_j - p_j] \cap F_g$. The final finish time is the first time (by value) to fulfil all constraints. This part was simplified in the given example but should be taken into account. As listed, the set of finish times also contains finish times of both dummy activities. This is of particular importance as without the finish time of the first dummy activity (source), no activity could be scheduled at $time = 0$.

The complete schedule generated by the serial SGS example is presented in figure 2.4 by using a resource-histogram (see section 4.5).

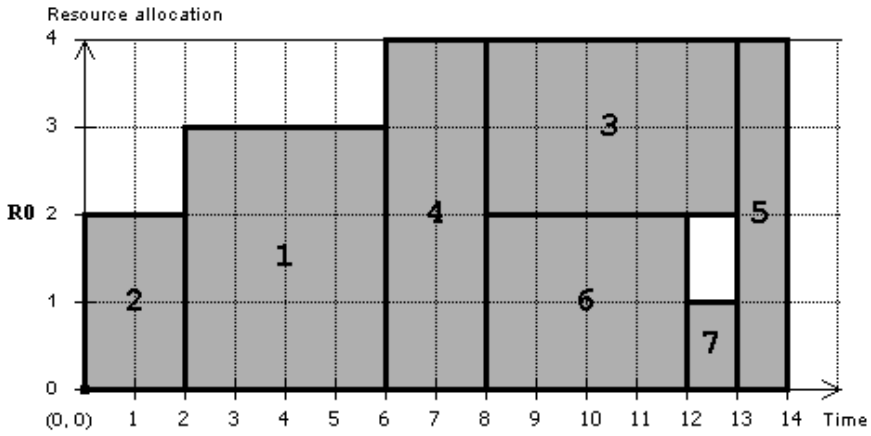

Figure 2.4: Resulting schedule.

2.4.2. Parallel Schedule Generation Scheme

The parallel schedule generation scheme is based on *time-incrementation*. Each iteration g links to one schedule time t_g. Activities scheduled up to iteration g are either part of the complete set C_g or active set A_g. The complete set contains all activities that have been scheduled and have completed up to t_g, therefore we can define the complete set as $C_g = \{j \in J \mid F_j \leq t_g\}$, where J is the set of all activities. The active set contains all activities which are scheduled but have not finished yet, that is $A_g = \{j \in J \mid F_j - p_j \leq t_g < F_j\}$. In each iteration, one activity is selected for scheduling out of the eligible set $E_g = \{j \in J \setminus (C_g \cup A_g) \mid P_j \subseteq C_g \wedge r_{j,k} \leq$

$\widetilde{R}_k(t_g)(k \in K)\}$. In other words, the eligible set contains all activities which can be precedence- and resource-feasibly started at time t_g. For P_j and $\widetilde{R}_k(t_g)$ definitions refer to section 2.4.1.

The parallel SGS is described as follows [12].

Algorithm 2 Parallel SGS

Initialize: $g = 0, t_g = 0, A_0 = \{0\}, C_0 = \{0\}, \widetilde{R}_k(0) = R_k$
while $|A_g \cup C_g| \leq n$ **do**
  $g = g + 1$
  $t_g = \min_{j \in A_g}\{F_j\}$
  Calculate: $C_g, A_g, \widetilde{R}_k(t_g), E_g$
  while $|E_g| > 0$ **do**
    Select one $j \in E_g$
    $F_j = t_g + p_j$
    Calculate: $A_g, \widetilde{R}_k(t_g), E_g$
  end while
end while
$F_{n+1} = \max_{h \in P_{n+1}}\{F_h\}$

The initialization sets the schedule time to 0, schedules the first dummy activity (source) and sets available resource capacities. The first step in each iteration consists of determining the current schedule time t_g. This is simply done by taking the minimum finish time of all currently scheduled activities. Afterwards, the complete, active and eligible set are calculated together with the current resource availabilities. The second step of each iteration schedules a subset of the eligible activities that can be started at time t_g. Even though the parallel SGS might have less than n iterations, there is always exactly n selection decisions which have to be made [12]. The time complexity of the parallel SGS is also $\mathcal{O}(n^2 \cdot |K|)$.

Both the serial and parallel SGS always generate feasible schedules in case of the resource-uncostrained problem. The parallel SGS constructs non-delay schedules meaning that it is possible to never gain an optimal schedule (see 2.3).

A complete run of the parallel SGS for the problem described in section 2.4.1 is given in figure 2.5. In this case, the resulting schedule is identical to the schedule generated by the serial SGS for this project instance (see figure 2.4).

#0) g = 0

$t_0 = 0$, $A_0 = \{0\}$, $C_0 = \emptyset$

$F_0 = \{0, -, -, -, -, -, -, -, -\}$

#1) g = 1

$A_1 = \{0\}$, $C_1 = \emptyset$

$t_1 = \min\{0\} = 0$

$A_1 = \emptyset$, $C_1 = \{0\}$

$E_1 = \{1, 2\}$

$j = 2$; $F_2 = 2$;

$E_1 = \emptyset$

$A_1 = \{2\}$, $C_1 = \{0\}$

$F_1 = \{0, -, 2, -, -, -, -, -, -\}$

#2) g = 2

$A_2 = \{2\}$, $C_2 = \{0\}$

$t_2 = \min\{2\} = 2$

$A_2 = \emptyset$, $C_1 = \{0, 2\}$

$E_2 = \{1\}$

$j = 1$; $F_1 = 6$;

$E_2 = \emptyset$

$A_2 = \{1\}$, $C_2 = \{0,2\}$

$F_3 = \{0, 6, 2, -, -, -, -, -, -\}$

#3) g = 3

$A_3 = \{1\}$, $C_3 = \{0, 1\}$

$t_3 = \min\{6\} = 6$

$A_3 = \emptyset$, $C_1 = \{0, 1, 2\}$

$E_3 = \{3, 4\}$

$j = 4$; $F_4 = 8$;

$E_3 = \emptyset$

$A_3 = \{4\}$, $C_3 = \{0, 1, 2\}$

$F_3 = \{0, 6, 2, -, 8, -, -, -, -\}$

#4) g = 4

$A_4 = \{4\}$, $C_4 = \{0, 1, 2\}$

$t_4 = \min\{8\} = 8$

$A_4 = \emptyset$, $C_4 = \{0, 1, 2, 4\}$

$E_4 = \{3, 6\}$

$j = 6$; $F_6 = 12$;

$E_4 = \{3\}$

$j = 3$; $F_3 = 13$;

$A_4 = \{3, 6\}$, $C_4 = \{0, 1, 2, 4\}$

$F_4 = \{0, 6, 2, 13, 8, -, 12, -, -\}$

#5) g = 5

$A_5 = \{3, 6\}$, $C_5 = \{0, 1, 2, 4\}$

$t_5 = \min\{12, 13\} = 12$

$A_5 = \{3\}$, $C_5 = \{0, 1, 2, 4, 6\}$

$E_5 = \{7\}$

$j = 7$; $F_7 = 13$;

$E_5 = \emptyset$

$A_5 = \{3, 7\}$, $C_5 = \{0, 1, 2, 4, 6\}$

$F_3 = \{0, 6, 2, 13, 8, -, 12, 13, -\}$

#6) g = 6

$A_6 = \{3, 7\}$, $C_5 = \{0, 1, 2, 4, 6\}$

$t_6 = \min\{13, 13\} = 13$

$A_6 = \emptyset$, $C_6 = \{0, 1, 2, 3, 4, 6, 7\}$

$E_6 = \{5\}$

$j = 5$; $F_5 = 14$

$E_6 = \emptyset$

$A_6 = \{5\}$, $C_6 = \{0, 1, 2, 3, 4, 6, 7\}$

$F_3 = \{0, 6, 2, 13, 8, 14, 12, 13, -\}$

#7) g = 7

$A_7 = \{5\}$, $C_7 = \{0, 1, 2, 3, 4, 6, 7\}$

$t_7 = \min\{14\} = 14$

$A_7 = \emptyset, C_7 = \{0, 1, 2, 3, 4, 5, 6, 7\}$

#8) g = 8

$A_8 = \emptyset, C_7 = \{0, 1, 2, 3, 4, 5, 6, 7\}$

$F_8 = \max\{6, 2, 13,8, 14, 12, 13\} = 14$

$F_8 = \{0, 6, 2, 13, 8, 14, 12, 13, 14\}$

Figure 2.5: Parallel SGS example.

2.4.3. Planning Directions

Even though it is not focus to this work, it is worth mentioning the basic planning concepts. A concrete schedule can be generated incrementally from multiple directions: *backward*, *forward* and *bidirectional planning* [10].

Backward planning simply builds a schedule by applying a schedule generation scheme on a reversed project network where each activity is scheduled as late as possible. Usually this approach results in a schedule not beginning at the start of the planning horizon. Therefore a simple left shift applied on all activities' starting times must be done.

Forward planning is the natural planning procedure where each schedule is constructed by scheduling activities from the start of the planning horizon. This procedure is focus to this work and reflects the schedule generation schemes described in section 2.4.

Bidirectional planning relies on constructing schedules in forward and backward direction simultaneously, where forward and backward priority values must be computed. Here no more information is provided as planning directions are out of the scope of this work. More details can be found in [10].

2.5. Schedule Representations

One of the main challenges in using metaheuristics lies in the inherent difficulty of choosing adequate representations for a given problem domain. Fortunately as this thesis is based on a genetic programming approach, the representation of a schedule does not influence the process of evolution as the main solution of GP is not a schedule (see chapter 3). For that reason, this section briefly describes the main schedule representation reported in literature [12].

To avoid confusion, it is important to notice that the below described representations define schedule representations in forms that yield concrete schedules only when used as input for schedule generation schemes (see section 2.4). Actual schedules which are results of schedule generation schemes are represented by a vector of finish times $(F_1, F_2, \ldots, F_n)$. This type of representation is beneficial in its simplicity and the ability to calculate all necessary values based on its content.

2.5.1. Activity List

In the activity list representation, a precedence feasible activity list is given:

$$\lambda = [j_1, j_2, \ldots, j_n]. \tag{2.12}$$

In this representation, each activity must have a higher index than each of its predecessors. As can be noticed, the serial SGS can easily be used in conjunction with

this representation by simply choosing the next activity by index in each iteration. For usage in the parallel SGS, additional transformations need to be applied.

2.5.2. Random Key Representation

The random key representation makes use of an array in which each activity j is assigned (typically a real-valued) number r_j:

$$\rho = (r_1, r_2, \ldots, r_n). \tag{2.13}$$

Both the serial and parallel SGS can easily be used to derive concrete schedules using this representation. In each iteration of both algorithms we simply select the activity with the highest or lowest value (depending on interpretation) in the eligible set. From this approach one can see that the random values represent priority values.

2.5.3. Priority Rule Representation

The priority rule representation defines a list priority rules:

$$\pi = [\pi_1, \pi_2, \ldots, \pi_n], \tag{2.14}$$

where each π is one priority rule. Both the serial and parallel SGS can be used to generate schedules using this representation. In the decoding procedure, the priority of the i-th activity is determined by the rule π_i. This representation differs from the random key representation in the fact that the priority values are calculated directly during the decoding process and not beforehand.

For more information on different schedule representation refer to [12].

2.6. RCSP Solving Methods

Depending upon the characteristics of the resource constrained scheduling problem, RCSP solving methods can generally be divided into exact and heuristic approaches [6].

Exact solving approaches search the complete space of feasible solutions and therefore guarantee optimality. But the search space is often of impractical size which makes such approaches almost useless for a very large number of problem instances. Nevertheless, it is important to mention the main representatives of this category:

- **Integer Programming (IP)** - In order to use IP as a solving approach, it is necessary to introduce a large number of binary decision variables upon which a

feasible schedule can be generated. This is generally impractical as the number of variables grows in relation to the number of activities.

- **Implicit Enumeration** - This technique relies on implicit enumeration trees and specific borders to conclude feasible solutions. Under good formulation, this method is said to be useful.
- **Branch-and-bound (B&B)** - By using a tree structure and branch conditions on relaxed problems this method can reduce the space of feasible solutions.
- **Dynamic Programming** - The initial problem is recursively divided into smaller problems which are solved separately and later connected to generate the final solution.

As exact methods are generally not applicable for larger problem instances, many different heuristic approaches have been developed. Heuristics differ from exact methods by searching only a part of the solution space which offers possible better performance in a given time frame but not optimality. Nevertheless, in most cases generating a feasible and good enough solution is far more important than optimality. Therefore heuristic approaches are a popular and useful option for solving the RCSP.

As this work is focused on a heuristic approach, further heuristic methods analysis is given in the next subsection.

2.6.1. Scheduling Heuristics Classification

Known scheduling heuristics for solving the resource constrained project scheduling problems can be further be classified into two categories: *priority rule-based methods (or constructive heuristics)* and *metaheuristic-based approaches* [10].

The first class of methods always start with no scheduled activity. A single schedule is incrementally constructed by selecting a subset of activities in each step until all available activities are scheduled and a complete schedule is created. This process is controlled by the schedule generation scheme (see 2.4) in combination with *priority rules* (functions), where the latter is applied to calculate activities' priorities.

The second class of methods are applied to initial complete solutions with the goal of achieving improvement in terms of a selected criterion. This procedure consists of successively applying a set of operators to transform one solution into another. The main representatives of this category are genetic algorithms, tabu search, simulated annealing, ant colony optimization and others. These approaches heavily depend on selecting a suitable schedule representation (see 2.5).

It is interesting too see that genetic programming offers a bridge between the mentioned two heuristics classes as GP is a metaheuristic-based approach for evolving priority rules used in incremental schedule generation. This makes the GP approach particularly interesting for research.

2.7. Scheduling Environments

As mentioned in the introduction part of this thesis, most traditional RCSP solving approaches focus on solving a specific problem instance, meaning that the problem being solved is analysed under static conditions. In those conditions all activities' values e.g. due-date, duration and resource demands are fixed and do not change throughout the scheduling process.

A far more interesting and demanding environment is the dynamic scheduling environment in which activities' properties' values are subject to changes at any time before being scheduled. That is, changes are possible for all unprocessed activities. Properties of activities that are already scheduled are, of course, not being changed. This environment offers a challenging problem as it is not possible to apply all methods (exact or heuristic) to find reasonable solutions. One of the main approaches used in generating dispatching (scheduling) rules that handle dynamic environments is genetic programming. As this approach is an important part of this thesis, an example with explanation is given in section 6.3.

3. Genetic Programming

"How can computers be made to do what needs to be done, without being told exactly how to do it?"
Arthur Samuel, 1959

As argued before, evolutionary computation (EC) techniques are in general able to find high quality solutions for concrete problem instances, but evolutionary algorithms by themselves are not able to handle one of the most challenging tasks in computer science, namely getting a computer to solve problems without the need of programming it explicitly [1]. This central and interesting task can be formulated by the given quotation of *Arthur Samuel*.

Genetic programming (GP) has been around for more than 30 years and has been widely received by the computer science community.

Functionally, GP is an evolutionary algorithm inspired by biological evolution and aims to find *computer programs* that perform a user-defined computational task. It is therefore a machine learning technique that searches the solution space of programs instead of actual solutions to problem instances. This idea differentiates GP to other evolutionary-based metaheuristics. GP is able to evolve *higher-level* solutions (programs) that can be applied to generate solutions to a variety of problem instances. Therefore, GP offers a solid ground for evolving solutions in terms of the RCSP.

GP shares the same initial decision demands as other EC techniques, including selecting the appropriate problem representation, genetic operators (crossover, mutation, selection), fitness function and GP primitives. Each of these components are crucial for a successful application of GP, therefore it is worth mentioning the basic concepts.

3.1. Fitness Function

The fitness function represents a crucial procedure used by GP during evolution to measure the quality of an individual solution (program). The fitness function is domain-

specific and therefore reflects the inherent properties of the concrete optimization problem.

By applying the fitness function, each individual is associated with a value representing its successfulness in learning to solve a given task. This value represents a basic concept used in other operators e.g. fitness-based selection, guiding the search and learning process of GP into the right direction.

3.2. Genetic Operators

Each GP evolution starts with a population of usually low fitness. Throughout the evolution, the application of genetic operators transforms the initial population into a fitter one. In terms of machine learning, these operators are defined as search operators. Even though a variety of operators exists, the three principal genetic operators are [3]:

- **Crossover** - The crossover operator combines the genetic material of two (or more) parents by swapping one or more parts of one parent with one or more parts of the other. In terms of tree-based structures (see section 3.4) in the crossover procedure, randomly selected subtrees of each parent are selected and swapped.
- **Mutation** - Mutation operates on only one individual. Usually, after crossover has occurred, each resulting child is additionally mutated with low probability (parameter) in order to introduce new genetic material. In tree structures, a mutation randomly selects a root node of a subtree and replaces it with a randomly generated subtree.
- **Selection** - In general, this operator is applied to select individuals out of a given population. The selected individuals are subject to other operators (usually crossover).

3.3. GP Primitives

The GP approach relies on building complex learning structures out of a set of more simpler primitives. These primitives are divided into two sets: the function and terminal set.

3.3.1. The Terminal Set

Definition 3.1. The *terminal set* is comprised of the inputs to the GP program, the constants supplied to the GP program and the zero-argument functions with side-affects executed by the GP program [3].

From the very meaning of the word *terminal* one can see that the terminal set contains elements which terminate a branch of a tree. No matter if inputs, constants or zero-argument functions, terminals always return an actual numeric value without taking an input. Therefore, terminal nodes have an *arity* of zero. In this context, arity is simply the number of arguments (inputs) of a function.

The selection of an appropriate terminal set is not an easy task and needs to be executed with special care, as terminals are in general very domain-specific and reflect the actual problem state.

3.3.2. The Function Set

Definition 3.2. The *function set* is comprised of statements, operators and functions available to a GP program [3].

The function set may or may not be domain-specific. Generally speaking, the range of available functions is very large and may contain any available programming construct. Some often used examples include:

- **Boolean Functions** - AND, OR, NOR, XOR.
- **Arithmetic Functions** - PLUS, MINUS, DIVIDE, MULTIPLY.
- **Conditional Statements** - IF, ELSE, THEN.
- **Others** - Memory functions, variable assignments, control flow statements, loops etc.

Choosing the right function and terminal set can be difficult. As general rule, the given sets should be powerful enough to be able to represent a solution to the problem at hand. But one should not spend to much time crafting domain-specific functions and terminals as GP is very creative at combining simple constructs into much more expressive statements.

3.4. Executable Program Structure

One of the most important steps in every EC application is selecting the appropriate problem representation. Terminals and functions are not programs, but can be used to create them by assembling executable program structures. Even though a handful of different representation exist (graph, linear, tree and others), we will mention the most important one: the tree representation.

3.4.1. Tree Structure

The figure 3.1 represents a simple tree-based solution. The tree can be executed in many different orders, but there is a convention for the execution.

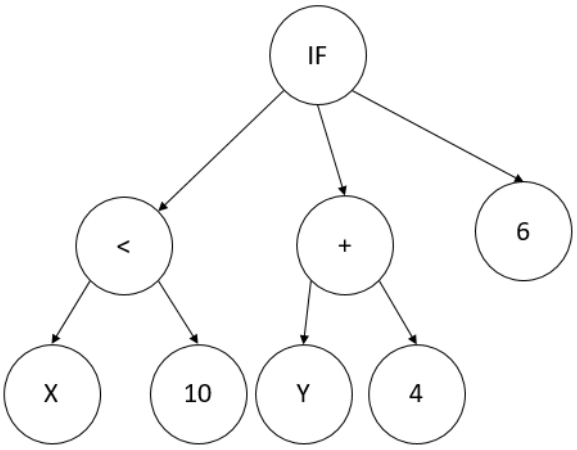

Figure 3.1: Tree structure example.

The two standard conventions for tree execution are prefix and postfix order execution. The postfix notation executes a tree by evaluating the leftmost node for which all inputs are available. This order is referred to as postfix notation because the operators appear after the operands. The second commonly used execution scheme is the prefix order execution. It is the precise opposite of postfix notation as the operators appear before their operands. In this order, nodes that are closer to the root node are evaluated before the terminal nodes. The prefix notation has the advantage that execution time can be saved in trees containing IF/THEN branches.

As the prefix notation is used in this work, an example execution of the tree in figure 3.1 is given as follows:

```
IF < X 10 + Y 4 6
```

Figure 3.2: Prefix notation example.

```
Internal states: X = 2, Y = 5
IF(<(X, 10), +(Y, 4), 6)
=> IF(<(2, 10), +(5, 4), 6)
=> IF(FALSE, +(5, 4), 6)
=> 6
```

Figure 3.3: Prefix evaluation example.

Here it is interesting to see that the tree structure uses only local memory as it is build in inherently. For example, the values X and Y are only local to their parent operator, meaning that they are not available to any other part of the program during execution. This is also true for all other values.

The given example also presents the mentioned prefix notation advantage. As the *IF* statement is resolved to *FALSE*, only the third argument is evaluated, while the second argument does not need to be executed at all.

4. The Application of GP

As discussed in the previous chapter, in terms of RCSP genetic programming generates higher-level solutions that can be used to find actual schedules for any problem instance. As GP is a machine learning technique it is necessary to provide a learning set upon which GP can evolve functions that can be used in combination with a schedule generation scheme to schedule a set of activities and provide a feasible complete schedule of good quality.

The mentioned function is used to calculate activities' priorities which are used to prioritize certain activities in the selection process of a given SGS (see section 2.4). This function is called a *priority function* and represents the actual solution of GP in terms of scheduling.

The main goal is to evolve appropriate priority functions on a selected learning set and successfully apply them to any project instance, even outside the learning set. Even though the process of evolving a qualifying priority function might take a longer time than actually finding a solution to a concrete project, the result afterwards can be used for generating schedules for any instance in a matter of seconds.

4.1. Implementation Model

In order to create a flexible and customizable development-experimental environment a *hyper-heuristic* model was devised and implemented to evolve appropriate scheduling heuristics for the RCSP using genetic programming.

To ease the usage of available GP implementations, the concrete implemented model is coupled with ECF (Evolutionary Computation Framework) [7] and written using the *C++* programming language. This framework offers interfaces to simply plug-in the necessary operators and run the evaluation with desired parameters. In case of RCSP, the only operator needed to be implemented is the evaluation operator modelled as the *RCSPEvalOp* class. This connection is presented with a class diagram in figure 4.1.

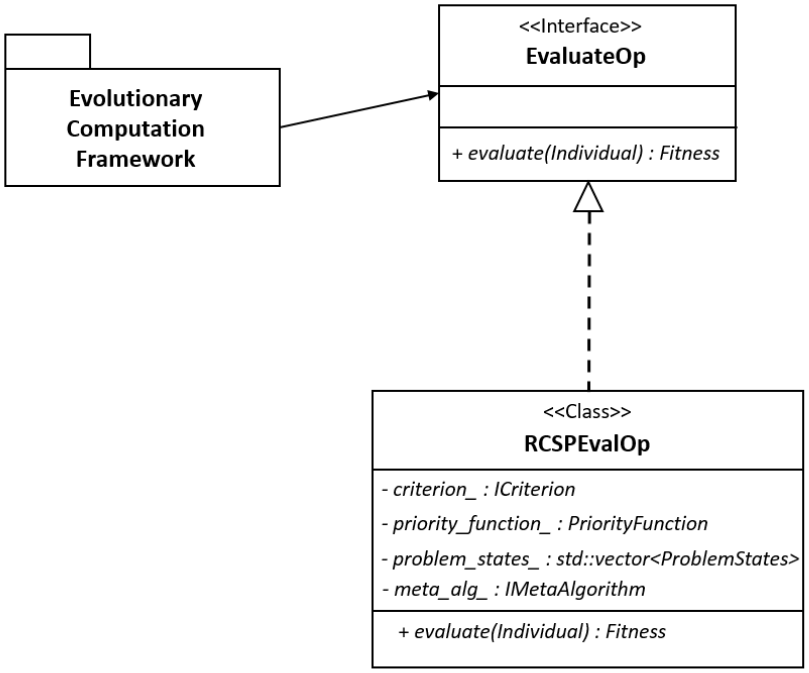

Figure 4.1: Association with ECF.

The *RCSPEvalOp* class implements the *evaluate()* function defined by the *EvaluateOp* interface that is exposed by ECF. This function takes as its only argument a reference to a generic *Individual* object. This individual represents one solution of the evolution. The idea of the evaluation class (*RCSPEvalOp*) is to evaluate the given solution and return the evaluation measure value as a *Fitness* object. This object is then assigned to the individual by the framework and later used in the proceeding evolution. This is the only association needed for the framework to successfully run.

The evaluator class references four core component class instances: *IMetaAlgorithm*, *ProblemState*, *PriorityFunction* and *ICriterion*.

The *IMetaAlgorithm* interface represents a generic scheduling algorithm inspired by a concrete schedule generation scheme (see 2.4). As can be seen in figure 4.2, the interface demands the implementation of the *createSchedule()* function which must receive references to an initialized problem state and a chosen priority function.

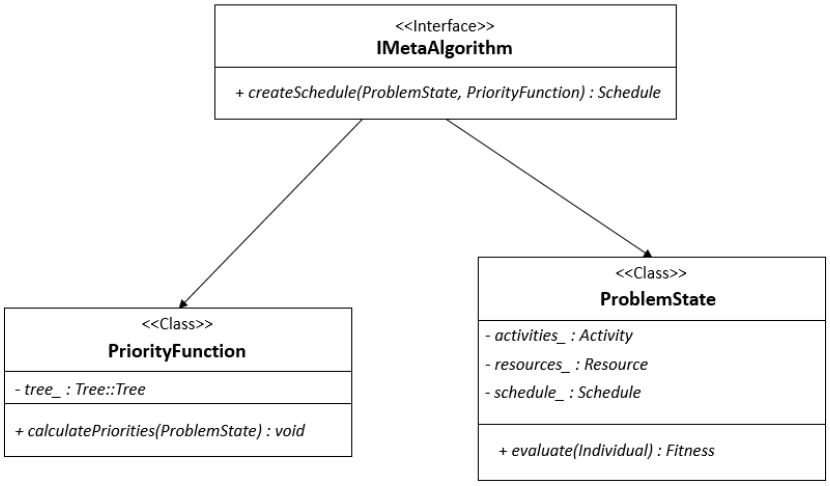

Figure 4.2: IMetaAlgorithm interface.

The priority function can be used for calculating activities' priorities during or before the scheduling process, depending on the terminals (static and/or dynamic) and the implemented SGS type. As the main representation of GP is a tree, the *PriorityFunction* class references the *Tree* class defined by ECF. The idea of the priority function is to evaluate the current referenced tree and set priority values for all eligible activities exposed by the problem state. If custom priority functions are needed, one can simply extend the *PriorityFunction* class and override the *calculatePriorities* function accordingly.

The concrete project is modelled as a *ProblemState* instance (see figure 4.3). Each problem state represents one concrete project containing all needed information, including activities definitions and resource demands. Each activity definition includes basic activity information as for example activity id, duration and references to predecessor and successor activities. The resource demands are modelled as separate object instances of the *ResourceRequest* class which references a concrete resource and stores the needed amount. A specific resource is implemented as the *Resource* class. This class stores basic resource information and exposes functions for taking and freeing a resource as well as checking if the resource is available for a certain amount.

Apart from storing necessary information, the problem state exposes a variety of functions for manipulating the activity and resource set with minimal effort. Here it is also possible to customize or upgrade a problem state by extending the *ProblemState* class. As shown in the class diagram, a concrete problem state also references the

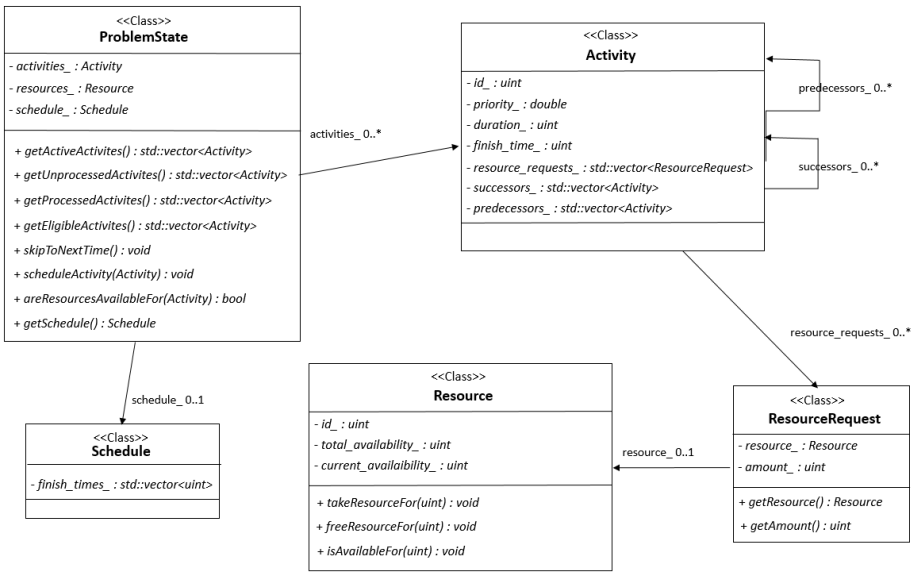

Figure 4.3: ProblemState class references.

Schedule class. As activities are manipulated through the problem state, it is generally simple to update a schedule instance in the problem state class. This can, of course, be changed if needed.

The last component referenced by the *RCSPEvalOp* class is the *ICriterion* interface. This interface simply uses a complete schedule instance and calculates an arbitrary measure. In this work the main optimization criterion used is the makespan which is simply the finish time of the last dummy activity which is equivalent to the total duration of the scheduled project instance. A complete class diagram of the *RCSPEvalOp* class with references is given in figure 4.4.

All components of the implemented hyper-heuristic model are generic and can be extended and reimplemented as needed. The current implementation is oriented towards the domain specifications of the used problem instances (see chapter 5) and enables the usage of any appropriately described problem definition with an arbitrary number of activities and resources.

Also it is important to note that the shown class diagrams contain only the most crucial components and relations to avoid complex diagrams.

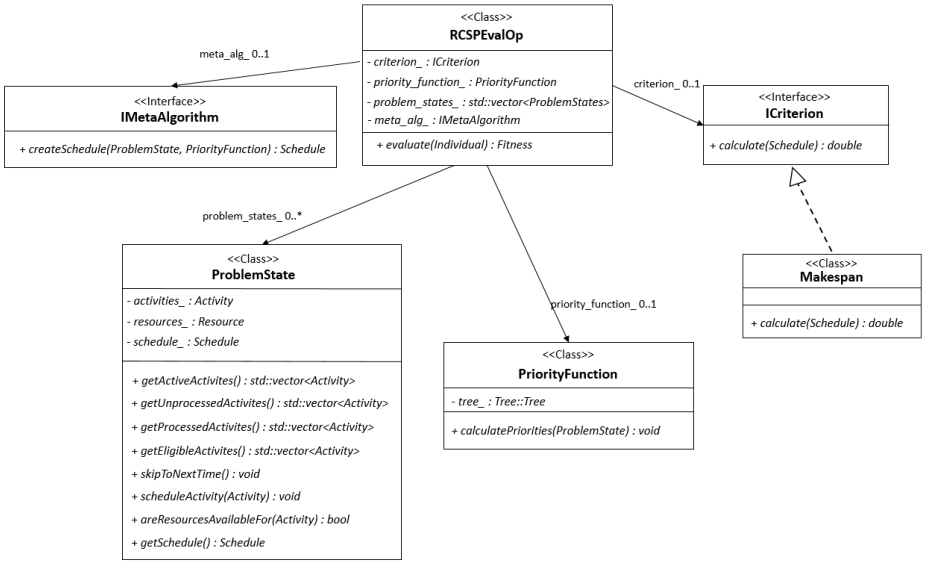

Figure 4.4: RCSPEvalOp class references.

4.2. Custom Schedule Generation Schemes

As mentioned before, schedule generation schemes are modelled with the *IMetaAlgorithm* interface which enables any generic implementation as long as the demanded *createSchedule* function is implemented. In this work three custom SGS versions are devised and implemented (see figure 4.5). Two versions are based on the parallel SGS while the third is an implementation of the serial SGS.

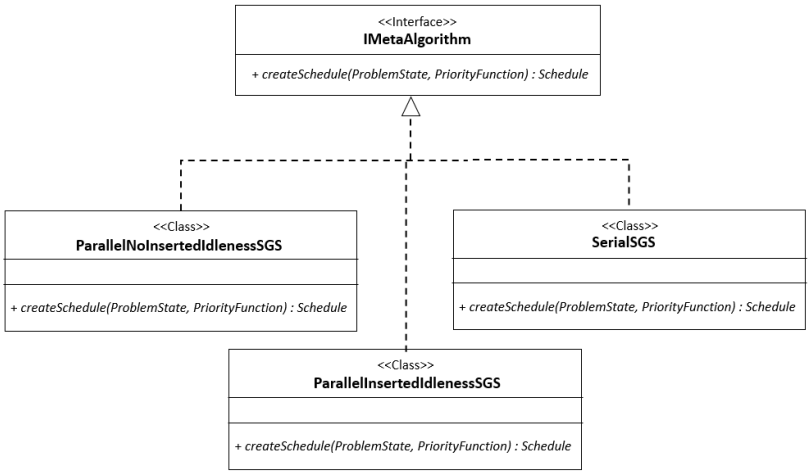

Figure 4.5: Custom SGS implementations.

4.2.1. Custom Parallel SGS

As time-incrementation is a natural approach to scheduling activities, two versions of the parallel SGS are given. The main property that differentiates the two versions is *idleness*.

The first implementation features a parallel SGS with inserted idleness. This version selects the highest priority activity and schedules all other activities only if they do not delay the start of the selected activity. This is true only if the high-priority activity can be scheduled at a given time as the eligible set only takes the precedence constraint into account while the resource constraint is resolved dynamically during the scheduling process. This property that allows for a certain activity to be delayed and scheduled at a later point is called *inserted idleness*. This approach is presented as algorithm 3 (*ParallelInsertedIdlenessSGS*).

Algorithm 3 Custom Parallel SGS with inserted idleness

if $has\_no\_dynamic\_terminals$ **then**
  $priority\_func.calculatePriorities(problem\_state)$
end if
while $hasUnprocessedActivities$ **do**
  $E = getEligibleActivities()$
  if $has\_dynamic\_terminals$ **then**
    $priority\_func.calculatePriorities(E)$
  end if
  Sort E by priority
  $t_{next} = getNextPossibleSchedulingTime(E[0])$
  for i to $|E|$ **do**
    $A_i = E[i]$
    if $i > 0$ and $createsDelay(t_{next}, A_i)$ **then**
      continue
    end if
    if $areResourceAvailableFor(A_i)$ **then**
      Schedule A_i
    end if
  end for
  Skip to next time
end while

The second version of a parallel SGS is very similar but does not include idleness. Therefore, activities are schedules (if possible) by priority without special focus on the activity with the highest priority. This version is shown as algorithm 4 (*ParallelNoInsertedIdlenessSGS*).

Algorithm 4 Custom Parallel SGS with no inserted idleness

if $has\_no\_dynamic\_terminals$ **then**
  $priority\_func.calculatePriorities(problem\_state)$
end if
while $hasUnprocessedActivities$ **do**
  $E = getEligibleActivities()$
  if $has\_dynamic\_terminals$ **then**
    $priority\_func.calculatePriorities(E)$
  end if
  Sort E by priority
  $t_{next} = getNextPossibleSchedulingTime(E[0])$
  for i to $|E|$ **do**
    $A_i = E[i]$
    if $areResourceAvailableFor(A_i)$ **then**
      Schedule A_i
    end if
  end for
  Skip to next time
end while

The parallel SGS versions are of particular importance as they are build upon a time-incrementation approach which is easily applicable in dynamic conditions.

4.2.2. Custom Serial SGS

Another approach concerns the serial SGS. Here no special customizations are made and the concrete serial SGS described in chapter 2.4.1 is implemented alongside with support for dynamic activity calculation. Here it is important to say that the performance of the parallel SGS outperforms the serial SGS by far. This is due to the need of the serial SGS to check resource and precedence constraints for each time moment of an activity's duration, which is a very demanding operation.

Also, the serial SGS is based on activity-incrementation and therefore not suitable for dynamic conditions, at least not in its primal form.

Both versions support dynamic and static terminals. If only static terminals are present the priorities calculation can be done only once before the SGS executes. If at least one dynamic terminal is used, the SGS will recalculate the priorities of the eligible set in every iteration automatically.

4.3. RCSP Terminal and Function Set

In order for GP to be expressive and have the tools to evolve intelligent solutions, it is necessary to define an appropriate and domain-specific terminal and function set.

The defined function set comprises basic arithmetic and logic operators as well as some custom ones as shown in figure 4.1. It is up to GP to use these operators to build more complex and meaningful operations.

Table 4.1: Function Set

Function name	Definition
+, -, *	addition, subtraction and multiplication
/	Protected division: $DIV(a,b) = \begin{cases} 1, & \lvert b\rvert < 0.000000001 \\ \frac{a}{b}, & otherwise. \end{cases}$
MAX	$MAX(a) = \begin{cases} a, & a > 0 \\ 0, & otherwise. \end{cases}$
POS	$POS(a) = \begin{cases} a, & a > 0 \\ -a, & otherwise. \end{cases}$
NEG	$NEG(a) = -a$
IF	$IF(a,b,c) = \begin{cases} b, & a > 0 \\ c, & otherwise. \end{cases}$

Choosing an appropriate terminal set is a far more comprehensive task, especially in terms of RCSP. In general, RCSP domain-specific terminals can be divided into two categories: static and dynamic terminals. Each of these categories can further be divided into project-specific and activity-specific terminals. Project-specific terminals include terminals which are specific for a project instance and depend only on the

general project properties and not on specific activities. Activity-specific terminals are terminals which reflect properties specific to a concrete activity.

Static terminals do not change during execution which means it is possible to calculate their values only once before the scheduling begins. The complete set of static terminals is presented in table 4.2.

Table 4.2: Static Terminal Set

Category	Terminal	Description
Project-specific	RF	resource factor
	RS	resource strength
	TNA	total number of activities (not including dummies)
	TD	total project duration (horizon)
Activity-specific	D	activity duration
	RR	number of required resources
	RRT	RR times quantity required for each resource
	ARU	average resource usage
	DPC	number of direct predecessors
	DSC	number of direct successors
	TPC	total number of predecessors
	TSC	total number of successors
	SPC	number of stages (levels) in predecessors' tree
	SSC	number of stages (levels) in successors' tree
	GRPW*	greatest rank positional weight all
	ES	earliest activity start
	EF	earliest activity finish
	LS	latest activity start
	LF	latest activity finish

Dynamic terminals are more flexible and their value can change according to the current state of activities and resources. Therefore, if dynamic terminals are used, it is necessary to recalculate priorities of certain activities during execution. The set of used dynamic terminals is given in table 4.3.

Table 4.3: Dynamic Terminal Set

Category	Terminal	Description
Project-specific	NUA	number of unprocessed activities
	NAA	number of active activities
	NPA	number of processed activities
	SUD	sum of durations of unprocessed activities
	SAD	sum of durations of active activities
	SPD	sum of durations of processed activities
Activity-specific	NSP	number of scheduled predecessors
	SL	slack: $max(EF - D - time, 0)$

The proposed sets are quite large and hold different types of information. The question is, which terminals are most important and store information that is crucial for scheduling? As in RCSP activities are scheduled, one should give more focus to activity-specific terminals. These terminals hold valuable information and can be used to extract decisions of whether or not an activity should be scheduled at some point. The process of filtering the most useful terminals in terms of evolution is described in chapter 5.

4.4. Optimization Criterion

A crucial step in using evolutionary computation techniques is modelling the fitness function. Even though many different criteria can be analysed in the RCSP as TWT (total weighted tardiness), ET_w (weighted earliness and tardiness), total weighted completion time and others, this work focuses on the *project makespan*.

As mentioned in chapter 2, the project makespan equals the project duration which is simply the finish time of the last dummy activity. It is a common goal in a variety of scheduling problems to create schedules of minimal makespan in order to save time and resources. To achieve minimal total project duration it is necessary to schedule activities for simultaneous execution (if possible) and avoid serialization. This is apparently a very complicated task considering resource and precedence constraints.

In terms of GP, a learning set must be established (see chapter 5) which means that

the optimization criterion must take many different project instances into account and guide the learning and evolution process into the right direction.

As project instances differ in the number of activities and their hardness (in terms of finding good solutions) it is profitable to define a fitness function that is normalized accordingly, meaning that it gives the same range of values for any kind of project. Even though a variety of functions can be defined, all of them can be generalized into two different approaches. The first approach concentrates exclusively on the number of projects and not on the project hardness, yielding the following function:

$$f_i = \frac{C_i}{p_i^{avg} \cdot \sqrt{n_i}}, \tag{4.1}$$

where f_i is the fitness value, C_i the achieved makespan, p_i^{avg} the average activity duration and n_i the number of activities for project instance i. One could argue about the selected division with $\sqrt{n_i}$ as being a subjective measure. But a handful of tests for a variety of project instances and measures proved this decision valid as good normalized values are achieved for the whole range of used instances.

The second approach additionally takes the project hardness into account. The influence of a project's hardness can be modelled using the resource factor RF and resource strength RS described in section 2.2. Here the following function is devised:

$$f_i = \frac{C_i \cdot RF_i}{p_i^{avg} \cdot \sqrt{n_i} \cdot RS_i}, \tag{4.2}$$

where the resource factor and strength are defined on the project level. Even though this function achieves good measures when comparing easier to hard problems, it is not good enough in normalizing values according to the number of activities, yielding a large range of values for different projects. This property could possibly lead to forcing GP evolution to focus on harder problems, neglecting the more easier ones. Because of this, the function 4.1 was selected to represent a solution's fitness value in all further experiments.

As GP learns on base of a learning set containing more than one project instance, the final fitness value is determined on behalf of the achieved values of all projects as follows:

$$f_k = \frac{f_i}{N} = \frac{\sum_i^N \frac{C_i}{p_i^{avg} \cdot \sqrt{n_i}}}{N}, \tag{4.3}$$

where f_k is the fitness value for one individual solution for all projects in a given set with N instances. As can be seen from the selected equation, in each evaluation step

only the value C_i is subject to change while all other values are constants. Therefore, the goal to find the minimal f_k reflects in favouring ever smaller values of C_i.

4.4.1. Variable Optimization Criteria

Another beneficial aspect of the GP approach to solving the RCSP is the ability to easily target any optimization criterion by simply changing the implemented schedule evaluation procedure. In terms of the devised hyper-heuristic model, one must simply add a custom *ICriterion* class and plug it into the evaluation procedure. No further changes are necessary for a successful application.

Even though only the makespan criterion is targeted in this work, here we present two other possible optimization criteria which can simply be inserted into the existing framework: the total weighted completion time (4.4) and the net present value (4.5).

$$\sum w_i C_i, \tag{4.4}$$

$$\sum_{i=1}^{n} c_i^F e^{-\alpha C_i}. \tag{4.5}$$

This optimization criterion variability contributes to the various benefits of the genetic programming solving method.

4.5. Schedule Visualization

In lack of available software solutions for easier solution understanding and visual feasibility checking, throughout this work an additional resource histogram visualization tool was implemented using the *Java* programming language.

The tool renders a resource histogram for any valid input definition file. The input file should contain basic information for each activity following the prescribed definition rules.

The general input file definition is simply a text file with the following structure:

```
# this is a comment
# project info: this part is optional!
# if you don't use it, comment or remove it
# first line: project name
PR_INFO
<name>

# resource info
# first line: number of resources
# other lines: <resource id> <amount>
RES_INFO
<num>
<other>

# activity info
# first line: number of activities without dummies
# other lines:
# <activity id> <duration> <finish time> <<resource requirements>>
# <resource requirements>: <id0 amount> <id1 amount> ... <idR-1 amount>
ACT_INFO
<first line>
<other lines>
```

Figure 4.6: Resource histogram input file structure.

The tool supports any number of activities and resources, rendering a responsive graph whose size can be changed dynamically. An example input file with two resources is presented in figure 4.7.

```
PR_INFO
example1

RES_INFO
2
0 4
1 4

ACT_INFO
6
1 2 2 1 3
2 4 6 1 3
3 1 1 0 1
4 2 8 2 2
5 3 4 2 1
6 1 9 4 0
```

Figure 4.7: Example resource histogram input file.

The given example yields the following output graph:

The rendered resource histogram presents each activity with an id-signed rectangle

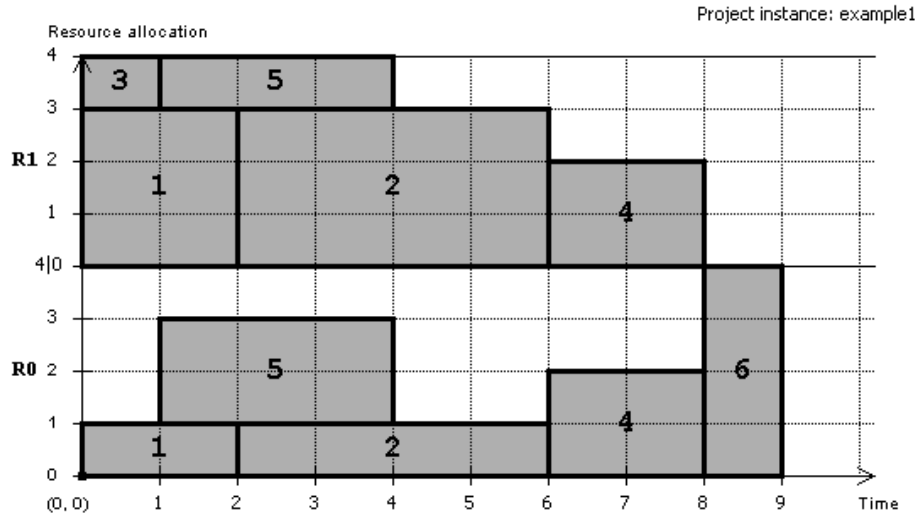

Figure 4.8: Example resource histogram graph.

for each resource allocation, where the x-axis represents the time flow, while the y-axis is divided into partial graphs for each resource. Therefore, the activity with $id = 1$ starts in $t = 0$ and finishes in $t = 2$, taking two units of resource $R0$ and three units of resource $R1$. The project makespan is given with the finish time of the last activity which is, in case of example in figure 4.7, the activity with $id = 6$ with its finish time in time moment $t = 6$. As dummy activities have no duration and resource demand, they are not shown in the graph and should not be part of the definition file.

In order to start the tool, one must provide the input file as the first and only argument and run the rendering process. If the *Java* code is packed into a *jar* file, the run command is:

```
java -jar resource-histogram.jar input.txt
```

Figure 4.9: Start tool (jar) command.

After the tool starts, a graph is presented in the given program window and a *.png* file is automatically generated and stored into the root folder of the executable file.

5. Experimental Setup

As in other EC techniques, the GP working environment is also coupled to the necessity of finding good parameters for the underlying algorithms and domain-specific evaluator where every decision must be supported by an extensive round of experiments. The experimental flow applied in this work contains the following major steps:

1. Initial parameters
2. Defining the learning and test set
3. Convergence analysis
4. Feature selection
5. Parameter optimization
6. Conducting final experiments

Each of these steps depends on the previous ones. Because of this, every decision brought as a result of concluding a specific step is done on behalf of a round of experiments and a handful of experience. In the next sections, each step is described in more details, while the final experiments are given in chapter 6.

5.1. Initial Parameters

Before any experiments can be conducted it is necessary to define an initial set of parameters to start with. In terms of GP, a set of used terminals and functions must be decided upon and the basic algorithmic parameters must be set.

Initially, all available functions are used for the function set, while a subset of activity-specific and project-specific terminals are used to form the terminal set (see table 5.1).

Table 5.1: Initial function and terminal set.

Functions	+, -, *, /, MAX, POS, NEG, IF
Terminals	GRPW*, DSC, RS, RR, ARU, TPC, NSP

Additionally, the basic evolution parameters have to be set. Here a *Steady State Tournament* genetic algorithm is used with the following parameters:

Table 5.2: Initial EC parameters.

Tournament size	3
Tree max depth	7
Mutation probability	0.3
Population size	500
SGS type	PSGS (inserted idleness)

This configuration is used for the first round of experiments in dataset selection and convergence analysis.

5.2. The Learning and Test Set

As a machine learning technique, GP needs a selected learning set to learn and evolve results of expected quality. In this work existing problem sets were used from the *project scheduling problem library* (PSPLIB) [14] which contains various types of synthetic resource constrained project scheduling problems as well as optimal and heuristic solutions.

Each project definition file contains a well defined structure. This structure was not changed in any way and the implemented evaluator was adapted to easily parse input files of the given type, offering a good and large data set for the purpose of this work.

The existing project instances are of different properties differing in the number of activities, network complexity, resource factors, resource strengths and project horizon. The range and various values of the used instances is presented in table 5.3.

Here the first step is to decide upon a large enough learning set that will cover a variety of instances with different properties. The total number of available project instances is greater than 2000. Therefore, a learning set of 56 instances has been established, covering various types of instances.

Table 5.3: Data set values range.

Property name	**Values**
Number of activities	{30, 60, 90, 120}
Network complexity	{1.5, 1.8, 2.1}
Resource factor	{0.25, 0.5, 0.75, 1.0}
Resource strength	{0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 1.0}

The second step includes establishing a large enough test set containing instances the algorithm has not seen before during the learning phase. As the test set all available instances without the selected learning set are used, building a test set of 1984 instances. This means that the selected learning set is more than 35 times smaller than the test set, or precisely it is roughly 2.82 % of the test set.

Here a larger learning set of 112 instances was also devised, but according to a handful of preliminary experiments, the smaller set showed better overall results. This was tested by comparing the results for both learning sets for the best and average solutions throughout 50 points for 12 runs where each run had the terminal evaluations number set to 10^6. The first result presented in figure 5.1 is generated by taking the best solutions according to the selected learning set and evaluating them on the test set.

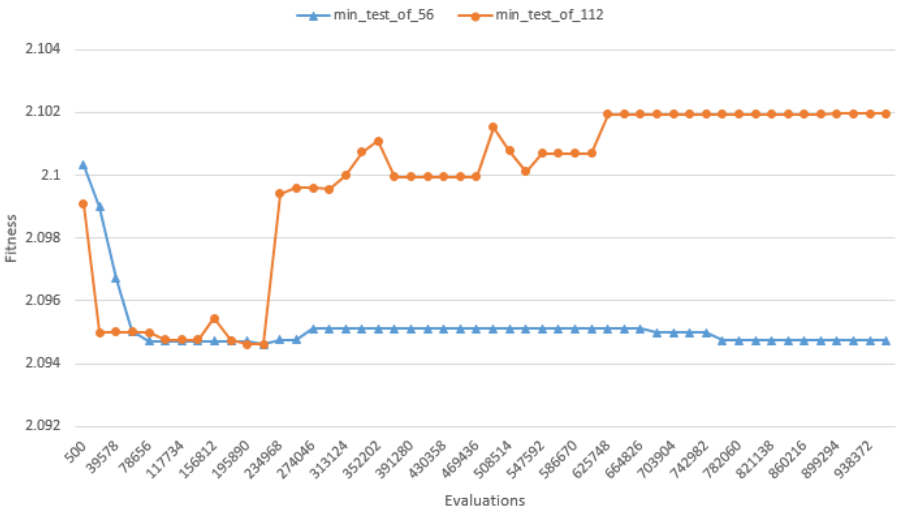

Figure 5.1: Best solution success.

As shown, the smaller learning set generates better solutions and is of smoother convergence. Here one can also note that after a certain numbers of evaluations, the

larger learning set generates solutions that are even worse than the initially randomly generated ones when tested. This is not the case with the smaller learning set, as all solutions tend to be of approximately the same quality.

Additionally, the average success of solutions generated by both learning sets is presented in figure 5.2.

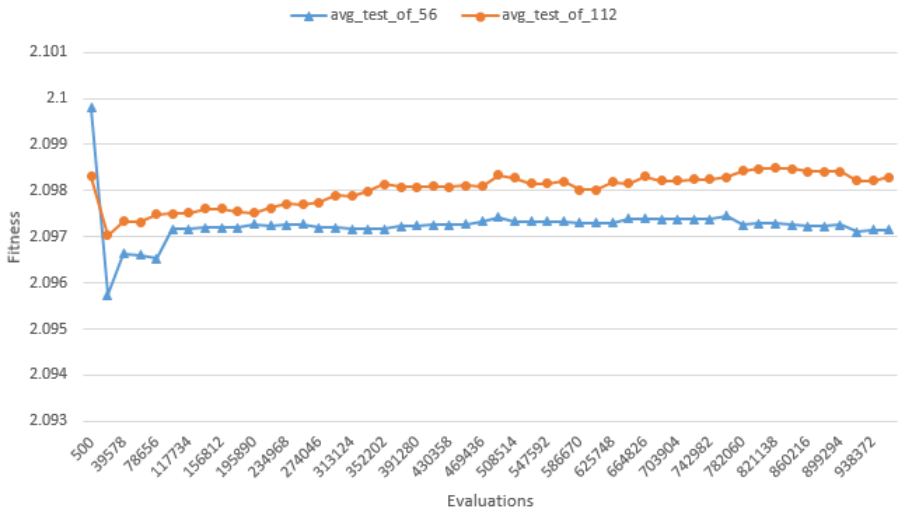

Figure 5.2: Average solution success.

The results also supports the smaller set as it evolves overall fitter solutions.

Both results are tested on a separate test set containing 1872 instances. This set was established by taking all instances and filtering out those existing in the 56 and 112 instances learning sets, offering the same conditions for both options.

The presented results support the selection of the smaller learning set, which is later used in all further experiments. A full list of selected projects building the 56 instances learning set is presented in appendix A with all major properties. One can see that a variety of properties' values is covered, building a meaningful and qualifying learning set.

5.3. Convergence Analysis

The first step in the GP experimental flow is to determine the appropriate number of evaluations needed for GP to converge. This is necessary to set the maximal number of evaluations to be used in all following experiments. Ideally, this number should be as low as possible to save time and resources.

Convergence analysis is done by running a set of N experiments for which the terminal condition is set to be an excessive number of evaluations. This terminal number usually depends on the task at hand and can be as large as 10^6 evaluations or more. In each generation of each run the minimum minimum, minimum average, median minimum and maximum minimum of N runs are recorded to a log file. After all runs terminate, all N log files are parsed and the algorithmic state of the recorded variables is stored into a certain number of points, e.g. 100. This process yields a predefined number of state points reflecting the algorithmic convergence throughout all generations. After plotting the generated values, one can easily determine the appropriate number of evaluations to be used in further experiments. The result of the convergence step should indicate the minimal needed number of evaluations which produce the greatest change in minimal and overall fitness values.

In this work for the selected learning set of 56 instances two rounds of experiments have been conducted, one for each type of parallel SGS (with and without inserted idleness). Each round consists of 30 separate runs with the terminal evaluation number set to 10^6. Here the serial SGS is not analysed as the total evaluation would take too much time to finish and at this stage the parallel versions offer a much more promising environment for further experiments. The convergence results for both algorithm are shown in 5.3 and 5.4 using 100 state points.

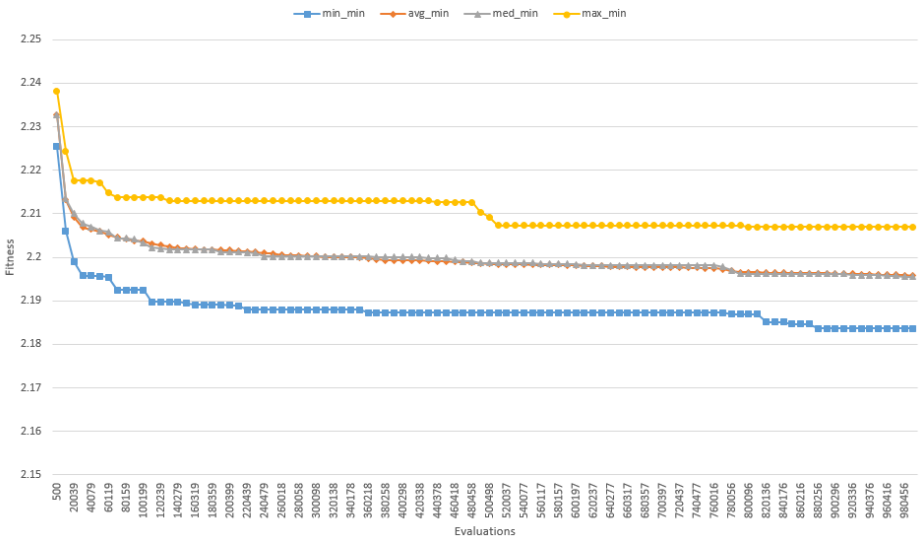

Figure 5.3: Convergence for the parallel SGS with inserted idleness.

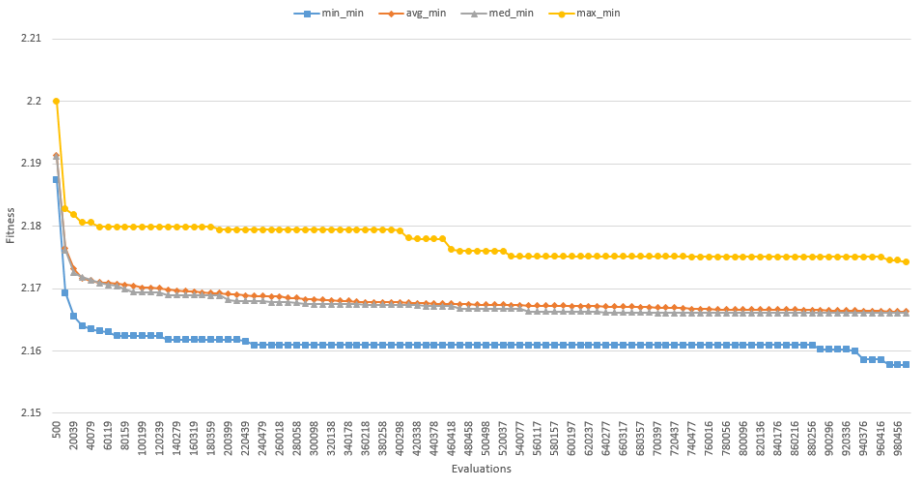

Figure 5.4: Convergence for the parallel SGS with no inserted idleness.

As can be seen in the given figures, the greatest fitness value change happens in the first $2 \cdot 10^5$ to $3 \cdot 10^5$ evaluations. Therefore, the maximum evaluations termination number is set to $2.5 \cdot 10^5$ evaluations. This value is further used in the process of feature selection.

Additionally it is interesting to notice that the given results indicate that the parallel SGS with no inserted idleness generates generally better results. However, the parallel SGS with inserted idleness is used in feature selection in hope that its nature to prefer higher priority activities will reflect on a smarter feature choice.

Even though some major improvement in solution fitness happens for both algorithm versions after approximately $8.5 \cdot 10^5$ evaluations, this evaluation number is not selected as a terminal one in the first round of further experiments as this would lead to huge time consumption needs. Also it is important to note that this behaviour is a characteristic of the learning set and not the test set for which stagnation occurs much earlier.

5.4. Feature Selection

After convergence analysis is done, the next step includes the feature selection, as now the maximal needed number of evaluations for generating good results is determined.

The goal of this process is to filter functions and terminals which are most important and beneficial for the evolution and to eliminate those who do not present any useful information source. Here many approaches can be thought of, but in general one can divide feature selection into two categories: full feature selection and greedy

feature selection. Each of these heuristic approaches can be realized in two counter ways, as a *build-up* and *build-down* heuristic approach.

The build-up feature selection, as its name states, tries to build the optimal set of operators. This selection type uses two sets: the set of used and unused operators. The used operators set contains those operators which are used by the evolutionary algorithm during execution, while the unused operators set represents operators which are currently not used. Usually this heuristic starts with an empty used operators set, even though one can predefine the starting operators set if desired. In each iteration, the selection algorithm selects one operator from the unused set and inserts it into the used operators set. The newly establish operators set is then used in evolution and a final fitness value is determined. After evaluation is done, the inserted operator is removed from the used operators set and added back to the unused set. This process is repeated for all operators in one iteration. When all operators are evaluated after separate injection, the one operators with the highest score is selected for joining the used operators set. In order for the selected operator to finally be added, the overall algorithm success must be greater with the used set including the new operator than without it. If this is not true, the selected operator is not added to the used operators set. This decision point ensures that the an operator can be added only if it enhances the targeted algorithm. The selection process is repeated until no operator is added in one iteration (or all are added).

The described build-up feature selection process is realized as a full selection as all operators are analysed in each iteration. Another approach consists of selecting the first operator that enhances the algorithm. This approach is known as greedy build-up feature selection.

Contrary to the build-up heuristic, the build-down feature selection heuristic tries to enhance the used operators set by filtering out operators that decrease the overall algorithm success. This selection process starts with the used operators set being full. In each iteration, one operator is removed from the set and the used operators set is used in evolution, yielding a fitness value. This is repeated for each operator. After all fitness values for each removed operator are determined, the one operator whose removal results with the highest enhancement is set to be the candidate for being removed from the used operators set. In order to be removed the overall algorithm success must be higher with the operators set not containing the selected operator than containing it. Otherwise, the operator stays in the used operators set.

This build-down feature selection is also described as a full selection. The greedy version of this selection simply removes the first operator whose removal brings any

kind of algorithm enhancement.

Here it is important to mention that after adding or removing an operator, the algorithm evaluation must be repeated a certain number of times, resulting in a list of fitness values, where the overall algorithm success is determined as the median fitness value. In this work, each algorithm evaluation is repeated ten times. Also one can notice that the results of both greedy versions depend on the order of the terminals listed in the configuration file as each terminal is selected in a serial manner which is defined before the process starts.

The evaluation process can potentially demand a great amount of time in terms of RCSP, depending on the evaluation function performance and the number of used project instances. Therefore it is necessary to exploit parallelism in terms of algorithm evaluations. In this work, a generic queue-based thread pool implementation was implemented in order to use all processor resources available on one machine. This implementation offers an input queue where any generic task can be scheduled for execution. The thread pool is automatically initialized to the number of logical processors in a system. All threads pull tasks from the input queue and execute them, leaving the result in the output queue. The main thread can wait for all tasks to be finished and dynamically cast the results to whatever value is predefined. This model can be used in all feature selections.

As these procedures are very time consuming, here only the partial results of the greedy build-up selection are presented in table 5.5 and 5.6, where the predefined input order of all terminals is given in table 5.4.

Table 5.4: Initial terminals definition order.

Terminals	GRPW*, DSC, DPC, NSP, TSC, TPC, SSC, SPC, D, ES, EF, LS, LF, TNA, RF, RS, RR, RRT, ARU, NUA, NAA, NPA, SUD, SAD, SPD, SL

Table 5.5: Selected static terminal set.

Category	Terminal	Description
Project-specific	TNA	total number of activities (not including dummies)
Activity-specific	RRT	RR times quantity required for each resource
	DPC	number of direct predecessors
	DSC	number of direct successors
	TPC	total number of predecessors
	TSC	total number of successors
	SPC	number of stages (levels) in predecessors' tree
	SSC	number of stages (levels) in successors' tree
	GRPW*	greatest rank positional weight all
	EF	earliest activity finish

Table 5.6: Selected dynamic terminal set.

Category	Terminal	Description
Activity-specific	NSP	number of scheduled predecessors

As shown above, the majority of the selected terminals are static activity-specific terminals, while only one dynamic terminal is selected. This is an expected result due to the fact that crucial information is stored in activity related terminals and that the predefined order of all terminals favours static terminals. Most dynamic terminals are given at the end of the predefined range, therefore it is expected that those terminals will be selected at some earlier stage of the selection procedure.

Even though the selection process did not finish completely at this point, the partial results are used in further optimization steps.

5.5. Parameter Optimization

One important step in the optimization of a GP system is the parameter optimization. The goal of this procedure is to find an optimal set of algorithm parameters that are capable of leading the evolution process into the right direction as efficiently as possible. This optimization step is highly domain specific and therefore coupled to the concrete evaluation function.

Full parameter optimization is a potentially very time consuming procedure, especially if a larger number of parameters is considered. Therefore, the optimization step is done by optimizing one by one parameter from the selected parameter set where after each optimization phase one parameter value is fixed before proceeding to the next step. Even though this resembles a greedy approach, it usually results in a good parameter selection that can be generated in a reasonable amount of time.

In each optimization step one parameter is considered. For this parameter a range of values are selected and a set of experimental runs is conducted with the defined parameter configuration. In each run the best fitness values are tracked to offer enough information to make a meaningful decision. After the optimization is done for all selected parameters, one can start to conduct final experiments.

In this work three parameters are considered for the tuning phase: population size, mutation rate and tree depth. Each tuning step considers three values for each parameter where a set of 20 runs is executed. Here it is important to note that a decision is made on behalf of solutions evaluated on the test set, not on the initial learning set.

The first optimization step includes optimizing the population size parameter. Here a range of three different population sizes are considered: 200, 500 and 1000. The achieved results are shown in figure 5.5. One can see that best results are generated for a population size od 200 and 1000 individuals, where the larger population generates slightly better solutions but with a larger dissipation of fitness values. However, since the results are relatively similar and a larger population is proven to be a good option, the population size of 1000 is selected.

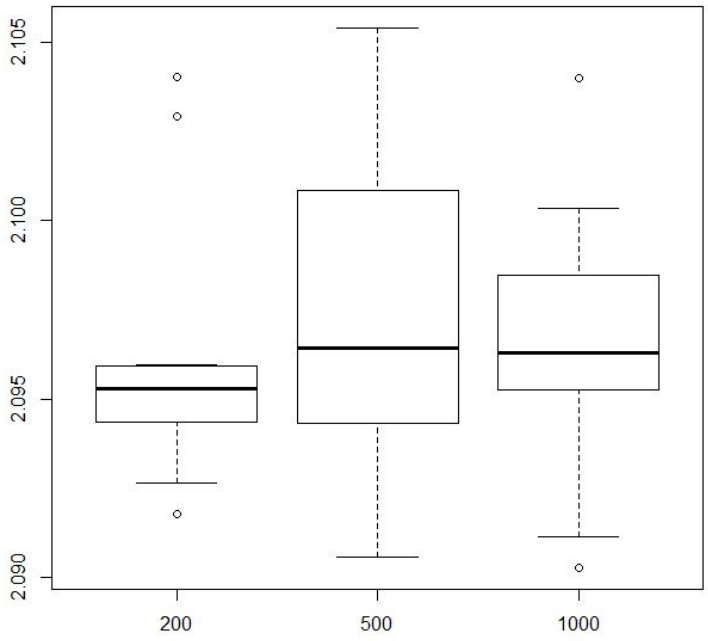

Figure 5.5: Boxplot for population sizes.

The second parameter to be considered is the individual mutation rate. Here three different values are considered: 0.3, 0.6, 0.9. The results are shown in figure 5.6.

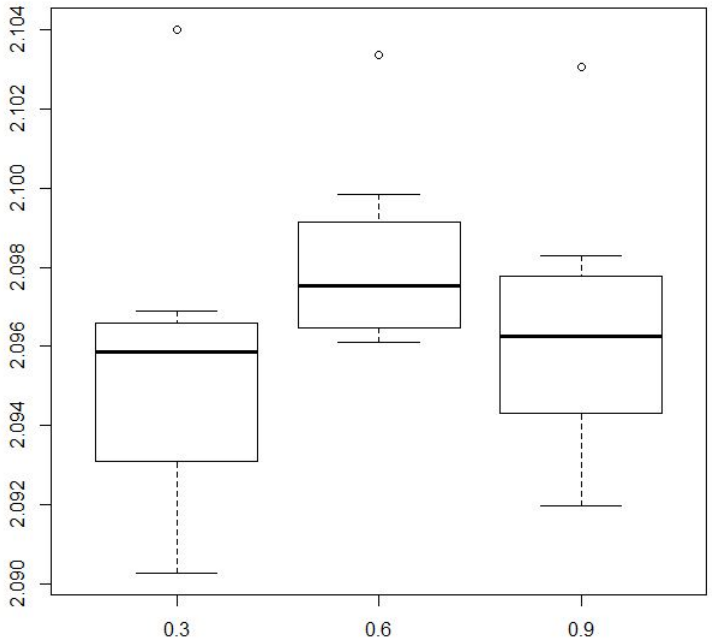

Figure 5.6: Boxplot for individual mutation rates.

Here the best choice is a mutation rate of 0.3, that is 30 % which generates overall best solutions.

The last parameter considered for the tuning phase is the tree depth. The results for three different tree depths are shown in figure 5.7. Even though all three depths have almost the same median values, the three depth of value 7 generates best results.

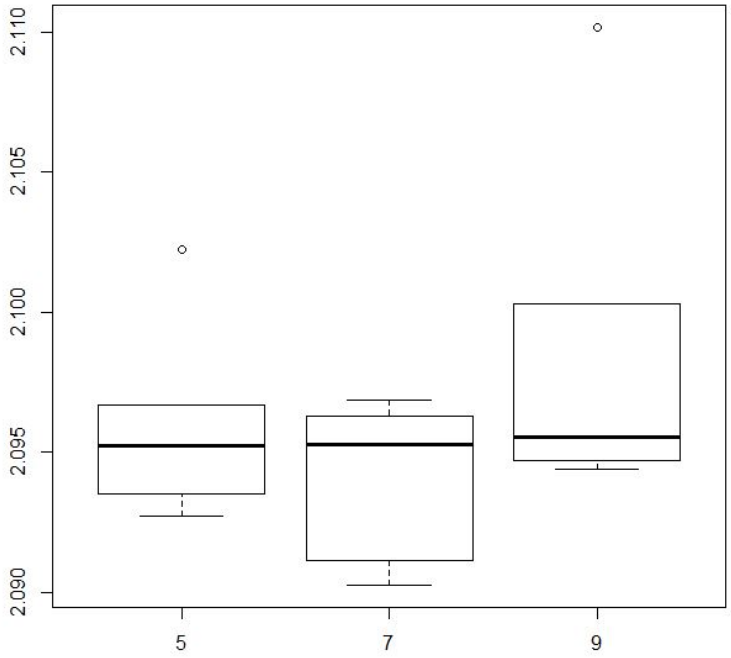

Figure 5.7: Boxplot for tree depths.

The final selected parameters are shown in table 5.7.

Table 5.7: Final selected algorithm parameters.

Population size	1000
Mutation probability	0.3
Tree max depth	7

6. Results

6.1. Final Experiments

For the final stage of experiments, all selected parameter values from the previous stages are taken into account. Here all three different schedule generation schemes are considered (see chapter 2.4).

For each SGS, a set of 20 runs is executed with the additional terminal stagnation value set to 50. This property ensures that the evaluation process terminates after 50 consecutive generations without improvement. For each run of each SGS, the final best solution is taken and evaluated on the test set (see 5.2). The achieved results are shown in figure 6.1.

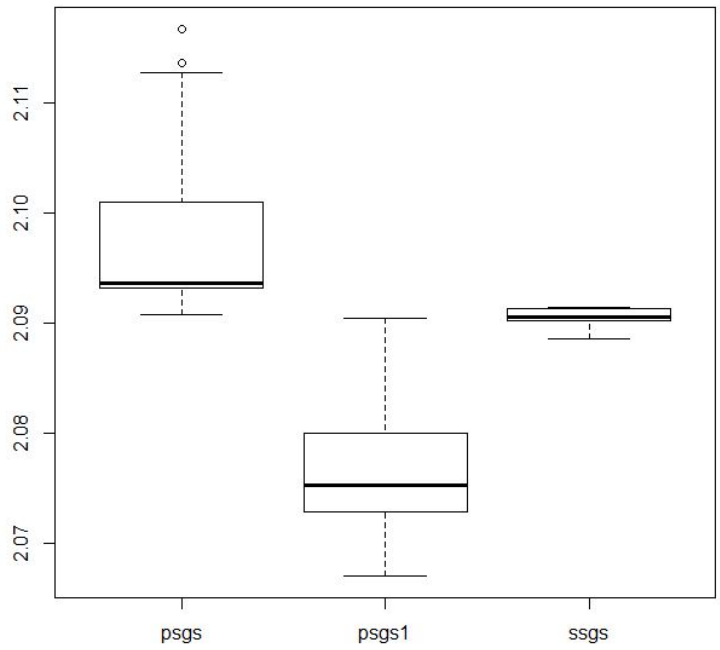

Figure 6.1: SGS final runs.

Here the *psgs* denotes the parallel SGS with *inserted idleness*, the *psgs1* the parallel

SGS with *no inserted idleness* and finally the *ssgs* denotes the serial SGS version.

In general, the results show an interesting difference between all three devised SGS versions. The parallel SGS with inserted idleness shows worst results with greater dissipation of best fitness values while the parallel SGS with no inserted idleness generates much better results than all other versions. The serial SGS seems interesting as it generates solutions of very similar fitness values (low dissipation).

6.2. Comparison With Existing Heuristics

In order to see how successful the solutions are, it is necessary to compare the best ones with existing scheduling heuristics (priority rules). For this reason, a few well known scheduling heuristics are selected and applied to the aforementioned test set. For easier comparison, all results are presented in two tables, where the results in table 6.1 show heuristic and GP achievements for the learning set, while the table 6.2 presents achievements for the test set. Here the best available GP solution for each SGS version is used to calculate the given values.

Table 6.1: Heuristic and GP results for the learning set.

Heuristics	**PSGS**	**PSGS1**	**SSGS**
GRPW*	2.239477	2.209870	2.218405
LST	2.240350	2.204331	2.221249
LFT	2.232402	2.192071	2.223641
GRPW	2.378092	2.363225	2.433958
SPT	2.476479	2.372084	2.595813
MSL	2.368033	2.329719	2.366418
MIS	2.305050	2.256045	2.350593
MTS	2.238293	2.208614	2.248315
GP	2.188030	2.147660	2.187520

Table 6.2: Heuristic and GP results for the test set.

Heuristics	PSGS	PSGS1	SSGS
GRPW*	2.096891	2.071174	2.091556
LST	2.086764	2.063453	2.080368
LFT	2.092721	2.063559	2.094752
GRPW	2.208572	2.185470	2.257091
SPT	2.287955	2.220975	2.408350
MSL	2.169497	2.138705	2.203557
MIS	2.175285	2.139966	2.211620
MTS	2.114407	2.081516	2.115462
GP	2.090783	2.067081	2.088606

The comparison clearly shows that the best solutions evolved by GP achieve better results than most of the existing priority values for all SGS versions. For the parallel SGS with inserted idleness and the serial SGS only the LST priority rule achieves better results than the GP solution. In case of the parallel SGS with no inserted idleness, the priority rules LST and LFT achieve slightly better results. The description of all applied heuristics can be found in appendix B.

Additionally it is interesting to see the difference between achieved results for all project instances in the test set and the best known results in terms of makespans. This comparison is given in figure 6.2 where the results are achieved by the parallel SGS with no inserted idleness as this SGS proved to generate best results.

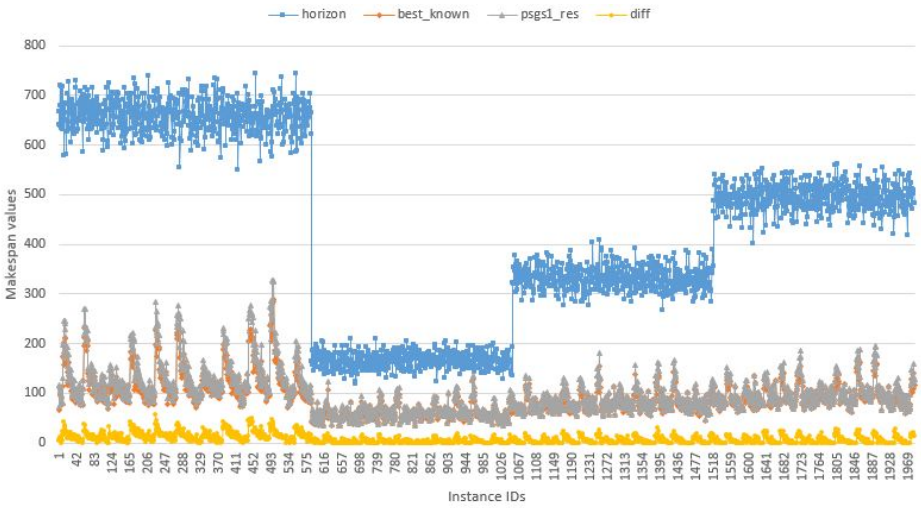

Figure 6.2: Best known and achieved results differences (PSGS1).

In the given figure, the blue line marked with squares represents the projects' horizon values, the orange line marked with rotated squares represents best-known solutions, the grey line marked with triangles represents the results achieved with the GP solution and finally, the yellow line marked with circles represents the absolute difference between best-known and achieved results.

One can notice that the overall difference between best known and achieved results differentiates only in small amounts, proving that the GP solutions give high-quality results for any instance in the test set. Also it is worth mentioning that the gap between achieved makespans and the maximum total project durations (horizons) is relatively large for all project instances without exception. This indicates that genetic programming is capable of evolving solutions which will tend to generate schedules of high-quality for a variety of different project instances.

For easier analysis an additional error histogram is given in figure 6.3. This figure shows the distribution of the aforementioned differences in selected intervals for the same solution as in figure 6.2.

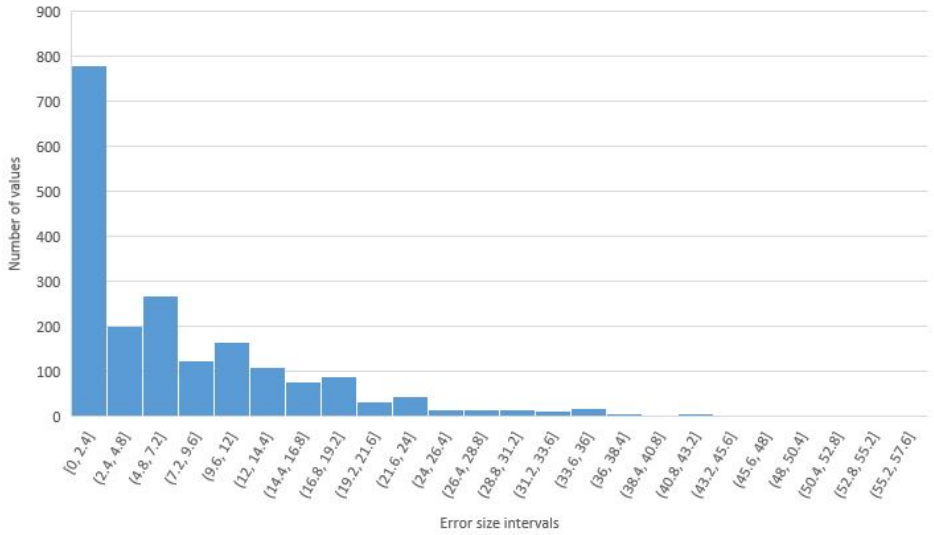

Figure 6.3: Best known and achieved results difference histogram (PSGS1).

This result shows that the smallest differences (errors) are by far the most common ones. As the difference grows, it becomes of much rarer occurrence.

Result differences and histograms for all three SGS variations can be found in appendix C. Concrete solution examples are given in appendix D.

6.3. Dynamic Environment

One of the main advantages of GP evolved priority functions is their applicability in dynamic conditions as described in section 2.7. Here a simplified example is presented with additional explanation.

The figure 6.4 presents the initial project schedule generated in a static environment, meaning that all activities' properties are calculated before the scheduling procedure begins.

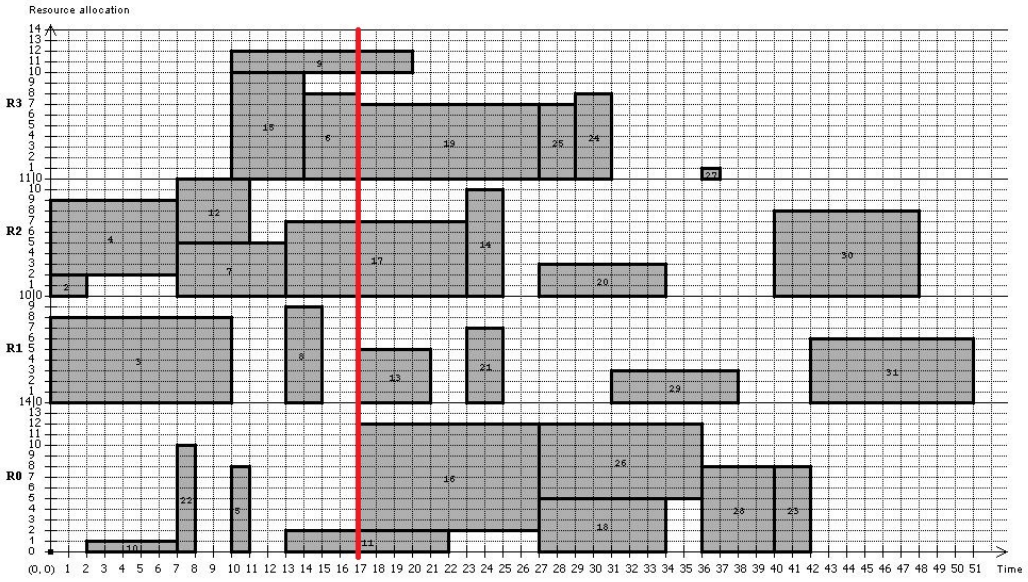

Figure 6.4: Initial project schedule.

The second figure 6.5 presents the generated schedule for the same project instance but in dynamic conditions.

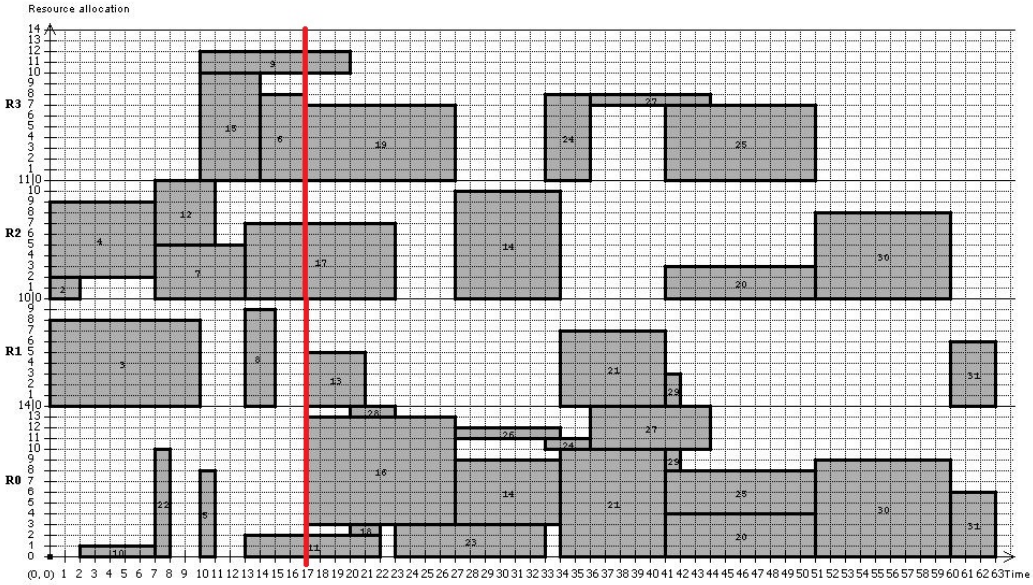

Figure 6.5: Project schedule in dynamic conditions.

In this case, after 17 time units (marked with a red line), the properties of all following activities to be schedule are randomly changed. The properties that are subject to changes are activities' duration and resource demands.

This simple example proves the capabilities of GP evolved solutions to easily cope with dynamic conditions and generate schedules of good quality. Moreover, not all terminals, that is existing priority rules can be calculated in dynamic conditions as for example the terminal ES (earliest start). For this property the relative activity position in the project network must be known before the scheduling procedure executes, which is not the case in dynamic conditions. Therefore, many existing priority rules are not applicable in those conditions and can only be used to create schedules for static projects. However, this is not the case for the GP approach, as one can simply use a specifically selected set of terminals which are able to cope with dynamic conditions. As schedule generation schemes do not depend on the selected terminals, they can always be applied regardless of the scheduling conditions.

7. Conclusion

The results presented in this work show a successful application of genetic programming to the resource constrained scheduling problem. Moreover, the GP generated priority functions show promising results in static and dynamic environments, where many other heuristics and approaches lack in solving capabilities and performance. When compared to existing heuristics, in most cases the achieved results generate much better schedules, presenting a competitive approach to solving the RCSP.

Additionally to the versatile GP solutions' capabilities, the applied solving approach offers a customizable and generically applicable framework to evolve priority rules for any class of scheduling problems targeting any selected optimization criterion. Here the generated solutions can be used to create schedules immediately in both dynamic and static environments, where only one concrete solution can be applied to any problem instance.

Therefore, the simplicity and adaptivity of the GP approach is proven to be a valid choice for solving the RCSP and promises qualifying results in any scheduling problem environment.

BIBLIOGRAPHY

[1] Michael Affenzeller, Stefan Wagner, Stephan Winkler, and Andreas Beham. *Genetic algorithms and genetic programming: modern concepts and practical applications*. Crc Press, 2009.

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

[3] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 270. Morgan Kaufmann San Francisco, 1998.

[4] Jacek Blazewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[5] Salah E Elmaghraby. *Activity networks: Project planning and control by network models*. Wiley New York, 1977.

[6] Khalil S Hindi, Hongbo Yang, and Krzysztof Fleszar. An evolutionary algorithm for resource-constrained project scheduling. *Evolutionary Computation, IEEE Transactions on*, 6(5):512–518, 2002.

[7] D. Jakobovic and et al. Evolutionary computation framework, October 2015. `http://ecf.zemris.fer.hr/`.

[8] James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.

[9] Robert Klein. *Scheduling of resource-constrained projects*, volume 10. Springer Science & Business Media, 1999.

[10] Robert Klein. Bidirectional planning: improving priority rule-based heuristics for scheduling resource-constrained projects. *European Journal of Operational Research*, 127(3):619–638, 2000.

[11] Rainer Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90(2):320–333, 1996.

[12] Rainer Kolisch and Sönke Hartmann. *Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis*. Springer, 1999.

[13] Rainer Kolisch, Arno Sprecher, and Andreas Drexl. Characterization and generation of a general class of resource-constrained project scheduling problems. *Management science*, 41(10):1693–1703, 1995.

[14] Rainer Kolisch, Christoph Schwindt, and Arno Sprecher. Benchmark instances for project scheduling problems. In *Project Scheduling*, pages 197–212. Springer, 1999.

[15] E Pinson, Christian Prins, and F Rullier. Using tabu search for solving the resource-constrained project scheduling problem. In *Proceedings of the 4th international workshop on project management and scheduling, Leuven, Belgium*, pages 102–106, 1994.

[16] Roman Słowinski. Multiobjective network scheduling with efficient use of renewable and nonrenewable resources. *European Journal of Operational Research*, 7(3):265–273, 1981.

Appendix A
Learning Set Selection

Here all projects contained in the selected 56 instances learning set are presented with their major properties' values. The selected learning set consists of four classes of instances: projects with 30, 60, 90 and 120 activities, each presented in its own table (A.1, A.2, A.3, A.4). Each of these classes contain a variety of problems with different values for the following properties:

- NC - Network complexity
- RF - Resource factor
- RS - Resource strength

For more details about the properties see section 2.3 or refer to [14].

Table A.1: Selected project instances with 30 activities.

Parameter Number j	**Instance Number**	**NC**	**RF**	**RS**
1	1	1.5	0.25	0.2
2	3	1.5	0.25	0.5
6	5	1.5	0.5	0.5
10	4	1.5	0.75	0.5
17	5	1.8	0.25	0.2
18	3	1.8	0.25	0.5
22	4	1.8	0.5	0.5
33	2	2.1	0.25	0.2
34	1	2.1	0.25	0.5
5	2	1.5	0.5	0.2
9	7	1.5	0.75	0.2
25	5	1.8	0.75	0.2
41	8	2.1	0.75	0.2
45	3	2.1	1.0	0.2

Table A.2: Selected project instances with 60 activities.

Parameter Number j	**Instance Number**	**NC**	**RF**	**RS**
1	1	1.5	0.25	0.2
2	2	1.5	0.25	0.5
6	3	1.5	0.5	0.5
10	4	1.5	0.75	0.5
17	5	1.8	0.25	0.2
18	2	1.8	0.25	0.5
22	6	1.8	0.5	0.5
33	1	2.1	0.25	0.2
34	4	2.1	0.25	0.5
5	3	1.5	0.5	0.2
9	4	1.5	0.75	0.2
25	1	1.8	0.75	0.2
41	6	2.1	0.75	0.2
45	3	2.1	1.0	0.2

Table A.3: Selected project instances with 90 activities.

Parameter Number j	**Instance Number**	**NC**	**RF**	**RS**
1	2	1.5	0.25	0.2
2	7	1.5	0.25	0.5
6	7	1.5	0.5	0.5
10	2	1.5	0.75	0.5
17	6	1.8	0.25	0.2
18	5	1.8	0.25	0.5
22	3	1.8	0.5	0.5
33	2	2.1	0.25	0.2
34	1	2.1	0.25	0.5
5	7	1.5	0.5	0.2
9	6	1.5	0.75	0.2
25	1	1.8	0.75	0.2
41	2	2.1	0.75	0.2
45	2	2.1	1.0	0.2

Table A.4: Selected project instances with 90 activities.

Parameter Number j	**Instance Number**	**NC**	**RF**	**RS**
2	3	1.5	0.25	0.2
5	2	1.5	0.25	0.5
10	2	1.5	0.5	0.5
15	5	1.5	0.75	0.5
22	6	1.8	0.25	0.2
25	5	1.8	0.25	0.5
30	3	1.8	0.5	0.5
42	8	2.1	0.25	0.2
45	1	2.1	0.25	0.5
7	5	1.5	0.5	0.2
12	1	1.5	0.75	0.2
32	2	1.8	0.75	0.2
52	3	2.1	0.75	0.2
57	6	2.1	1.0	0.2

Appendix B
Priority Rule-Based Heuristics

All priority rules applied for comparison reasons are presented in table B.1. All rules are given with their description and the additional *sort type* parameter. This parameter denotes which activity's priority is taken as the highest one after sorting is done (see 2.4). More details can be found in [10].

Table B.1: Priority rules definition.

Priority rule	**Description**	**Sort type**	**Priority value**
GRPW*	Greatest rank positional weight all	max	$d_j + \sum_{i \in F_j^*} d_i$
LST	Latest starting time	min	LS_j
LFT	Latest finish time	min	LF_j
GRPW	Greatest rank positional weight	max	$d_j + \sum_{i \in F_j} d_i$
SPT	Shortest processing time	min	d_j
MSL	Minimum slack time	min	$LS_j - ES_j$
MIS	Most immediate successors	max	$\|F_j\|$
MTS	Most total successors	max	$\|F_j^*\|$

Appendix C
Results and Comparisons

Here the differences between achieved and best known results are given for each SGS version, together with their error distribution histograms.

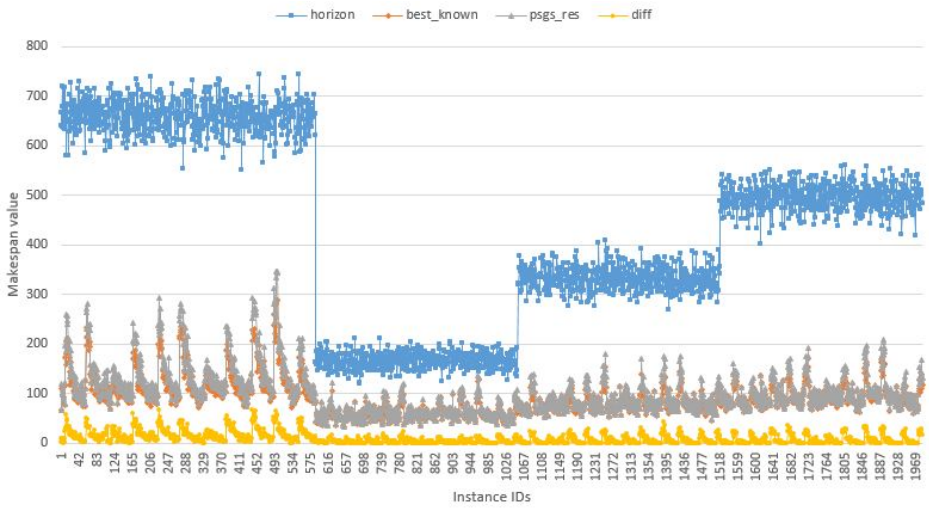

Figure C.1: Best known and achieved results differences (PSGS).

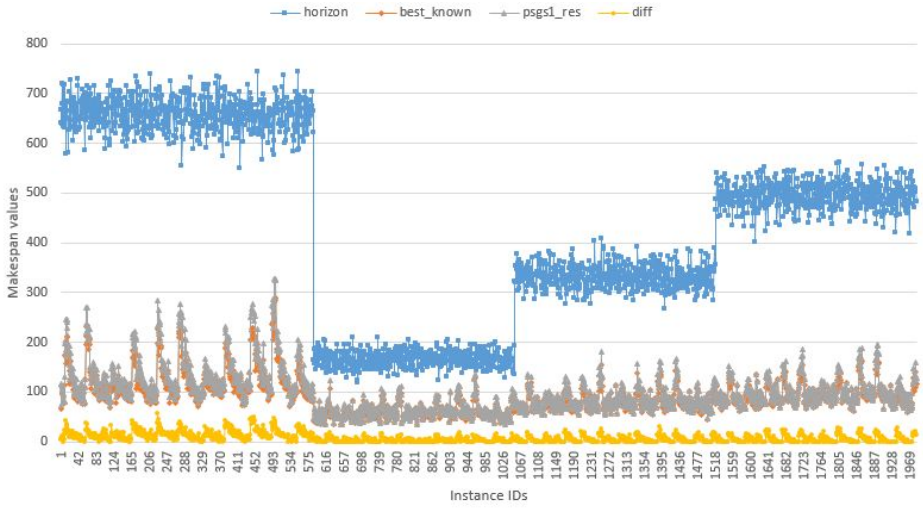

Figure C.2: Best known and achieved results differences (PSGS1).

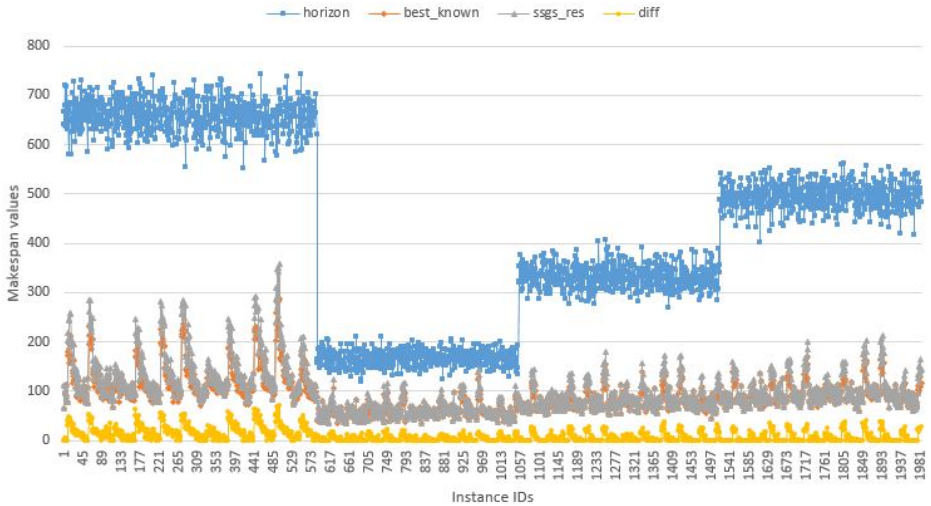

Figure C.3: Best known and achieved results differences (SSGS).

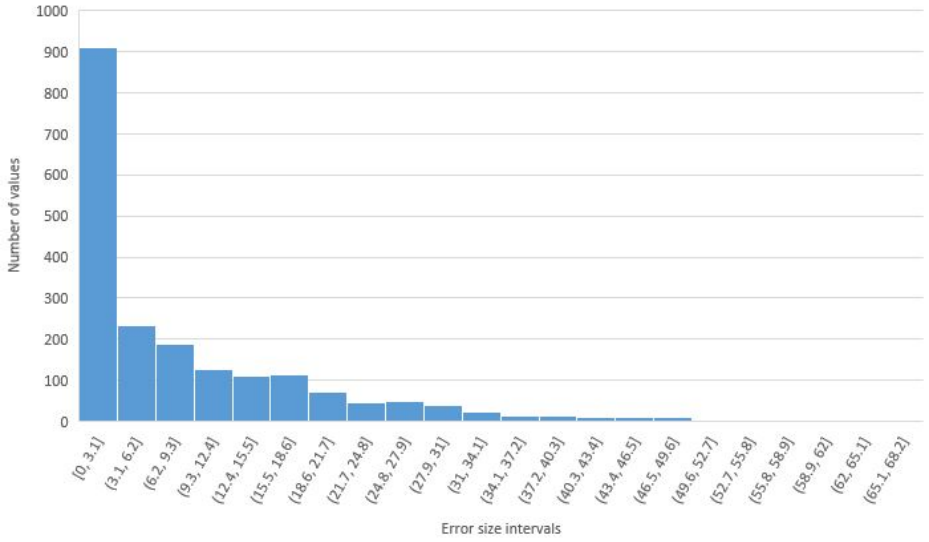

Figure C.4: Best known and achieved results difference histogram (PSGS).

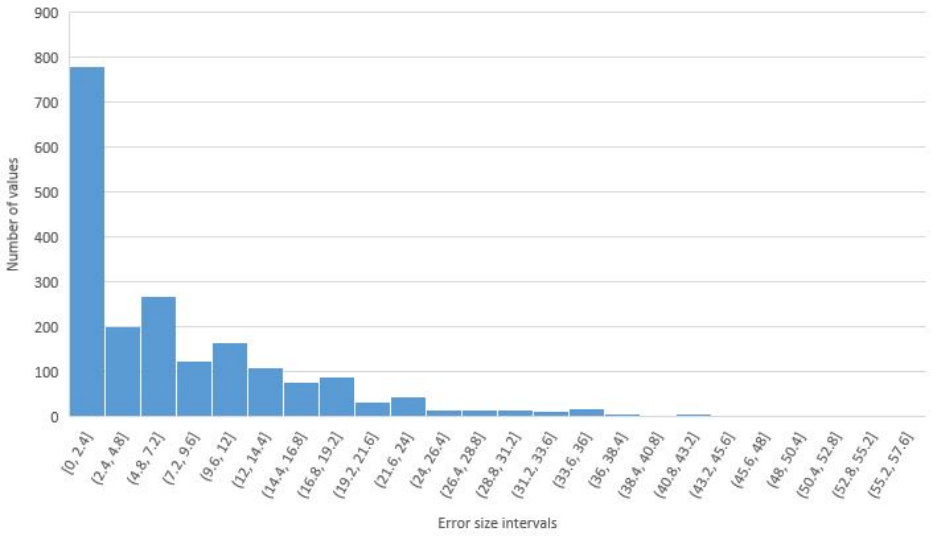

Figure C.5: Best known and achieved results difference histogram (PSGS1).

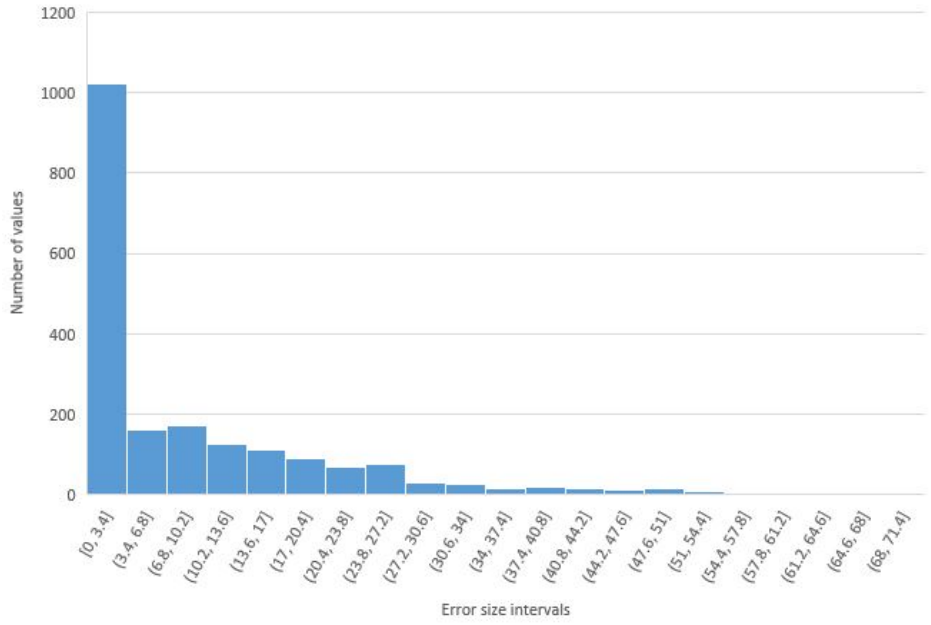

Figure C.6: Best known and achieved results difference histogram (SSGS).

Appendix D
Evolved Solution Examples

In this appendix the concrete best achieved solutions are presented for each SGS version. The results are shown in infix notation.

```
(((SSC * GRPW) * ((MAX((EF - IF(DSC,TSC,DSC))) + RRT) +
MAX(IF((IF(TPC,TSC,SPC) - POS(DPC)), POS(IF(TSC,RRT,GRPW)),
(EF + IF(DSC,TSC,TSC)))))) * ((GRPW * GRPW) * (GRPW - TSC)))
```

Figure D.1: Best PSGS solution (infix).

```
((((POS((MAX(SPC) * IF(TSC,TSC,GRPW))) /
((IF(TNA,DSC,GRPW) + IF(TSC,RRT,SPC)) /
((RRT / SPC) - IF(SSC,DSC,GRPW)))) + RRT) +
(POS(TNA) + ((MAX(SSC) - DSC) /
((TNA + TSC) / ((RRT / SPC) + IF(NSP,DSC,GRPW)))))) * GRPW)
```

Figure D.2: Best PSGS1 solution (infix).

```
((POS(((((POS(SSC) * IF(DPC,NSP,TNA)) +
(POS(SSC) * IF(DPC,NSP,TNA))) *
(IF((TSC - TSC),MAX(DPC),(TSC / TPC)) /
NEG(POS(TPC)))) - POS(((POS(NSP) / POS(DPC)) /
((RRT + SSC) - (TNA + SPC)))))) + (MAX(GRPW) - POS(DPC)))
```

Figure D.3: Best SSGS solution (infix).

Evolution of scheduling heuristics for the resource constrained scheduling problem

Abstract

With the growth of projects in complexity, size and resource demands, finding feasible schedules of good quality becomes a very challenging task within project management. Therefore the resource constrained project scheduling problem (RCPSP) has been studied extensively during the last decades yielding a variety of exact and heuristic solving approaches. In this context, the main issue described within this thesis is the application of genetic programming for the evolution of scheduling heuristics capable of efficiently handling generic project instances in both static and dynamic conditions. For this purpose a hyper-heuristic model was devised and implemented with generic extension capabilities. An extensive round of experiments was conducted for parameter optimization and the generation of best possible solutions. The effectiveness of the proposed approach is analysed considering existing scheduling heuristics and environmental conditions.

Keywords: resource constrained scheduling, RCPSP, genetic programming, priority rules

Evolucija heuristika raspoređivanja za raspoređivanje s ograničenim sredstvima

Sažetak

Pronalazak mogućih rasporeda dobre kvalitete postaje velik izazov u projektnom menadžmentu s porastom kompleksnosti, veličine i zahtjeva projekata. Iz tog razloga je problem raspoređivanja s ograničenim sredstvima iscrpno istraživan kroz prošla desetljeća, rezultirajući u velikom rasponu egzaktnih i heurističkih metoda rješavanja. U tom kontekstu se ovaj rad fokusira na primjenu genetskog programiranja za evoluciju heuristika raspoređivanja sposobnih za efikasno savladavanje generičkih projekata u statičkim i dinamičkim uvjetima. Za tu potrebu implementiran je hiper-heuristički model s mogućnošću nadogradnje. Izvršen je niz iscprnih pokusa s ciljem optimizacije parametara evolucije te generiranja što boljih rješenja. Analizirana je efikasnost predloženog postupka uzimajući u obzir postojeće heuristike raspoređivanja i uvjete okruženja.

Ključne riječi: raspoređivanje s ograničenim sredstvima, RCPSP, genetsko programiranje, prioritetna pravila