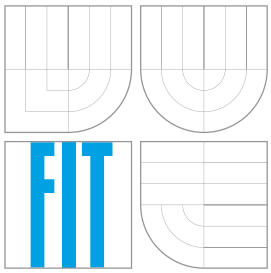
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

HEURISTIKY V OPTIMALIZAČNÍCH ÚLOHÁCH TŘÍDY RCPSP

META-HEURISTIC SOLUTION IN RCPSP

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR ŠEBEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN HRUBÝ, Ph.D.

BRNO 2015

Abstrakt

Tato práce se zabývá popisem stavu resource-constrained project scheduling problem. Definuje základní problém a jeho složitost. Také popisuje varianty tohoto problému. Jsou představeny algoritmy pro řešení RCPSP. V práci je důkladně rozebrán heuristický genetický algoritmus GARTH. Je také naznačena implementace dvou prototypů řešících RCPSP pomocí algoritmu GARTH. Je navrhnuto několik vylepšení originálního algoritmu a ty jsou vyhodnoceny.

Abstract

This thesis deals with the description of the state of resource-constrained project scheduling problem. It defines the formal problem and its complexity. It also describes variants of this problem. Algorithms for solving RCPSP are presented. Heuristic genetic algorithm GARTH is analyzed in depth. The implementation of prototypes solving RCPSP using GARTH is outlined. Several improvements to the original algorithm are designed and evaluated.

Klíčová slova

Rozvrhování, RCPSP, heuristický algoritmus, genetický algoritmus, GARTH

Keywords

Scheduling, Resource-constrained project scheduling problem, RCPSP, heuristic algorithm, genetic algorithm, GARTH

Citace

Petr Šebek: Meta-Heuristic Solution in RCPSP, diplomová práce, Brno, FIT VUT v Brně, 2015

Meta-Heuristic Solution in RCPSP

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Martina Hrubého, Ph.D.

.....
Petr Šebek
May 26, 2015

Poděkování

Chtěl bych poděkovat vedoucímu této práce panu Ing. Martinu Hrubému, Ph.D. za uvedení do problematiky, četné konzultace a cenné rady. Dále bych chtěl poděkovat Ondřeji Pelechovi za jazykovou korekturu práce. A v neposlední řadě děkuji mým nejbližším za podporu a porozumění.

Contents

Chapter 1

Introduction

Decisions how to plan a project are done every day and mistakes or just misjudgments can cost incredible amounts of money. If companies optimize their projects plans they can achieve a competition advantage and seize control over the market.

In this work I focus on *resource-constrained project scheduling problem* (RCPSP) [3] which is and long time will be one of the most important problems in operational research. In its basic shape it creates schedules of activities that have durations and demands on resources. Activities have to be executed in constrained order and resources are restricted too. All activities have to be finished and no activity can be preempted. Our main objective is to finish a given project as soon as possible, i.e. to minimize the *makespan* of the project schedule.

The simple definition reveals the problem belonging to the class of NP-complete problems and is one of the most intractable classic problems in practice [4]. You can get an impression of how hard it is from the fact that the set of artificial instances of RCPSP problems called PSPLIB [20] was published in 1996 and after nearly 20 years we still have not solved all problems. PSPLIB contains the biggest problems of 120 activities, which is not that much if you imagine the size of contemporary projects, and after many years of hard work we still compute success as how close to unknown theoretic optimum we can reach.

In this work I describe basic notation of RCPSP problem 2.1, define necessary tools to operate with the problem 2.4 and deduce its complexity 2.2. I also describe this field in its breadth by introducing variants that are derived from basic RCPSP 2.5. I outline exact algorithms 3.3 and meta-heuristics 3.4 to solve this NP-complete problem. Then I introduce GARTH 4, a novel genetic algorithm with great results. In following sections I present my main contribution to GARTH. First I design several improvements to the original algorithm 5. Then I present my two prototypes of GARTH 6. In the last section 7 I provide an analysis of schedule characteristics 7.2, a comparison of the results of the original paper 7.3 and an evaluation of my improvements.

Chapter 2

Resource-Constrained Project Scheduling Problem

2.1 Problem Formulation

Resource-constrained project scheduling problem (RCPSP) is a combinatorial optimization problem that searches the shortest schedule of activities with known durations and resource demands to resources of limited capacity. Activities are in precedence relation thus we can model what has to be done before an activity. Activities cannot be interrupted. Main objective of RCPSP is to find a schedule with the minimal *makespan*, i.e. the lowest duration of whole schedule possible, that holds precedence and resource constraints.

Definition 1 (RCPSP)**.** RCPSP of N activities and K resources is a structure $G = (J, P, R, c, d, r)$ where:

- $J = \{j_0, j_1, \ldots, j_N, j_{N+1}\}$ is a set of jobs.
- $P \subseteq J \times J$ is a precedence relation.
- $R = \{r_1, \ldots, r_K\}$ is a set of renewable resources.
- $c : R \to \mathbb{N}$ denotes a capacity of resource.
- $d : J \to \mathbb{N}$ assigns a non-negative production duration to each job.
- $r : J \times R \to \mathbb{N}$ are resource requirements of job on resource.

Dummy activities j_0 and j_{N+1} are called a start activity and an end activity respectively. They hold these conditions:

- j_0 is the predecessor of all other activities.
- j_{N+1} is the successor of all other activities.
- $d(j_0) = d(j_{N+1}) = 0$.

Further, let $P_i = \{j \in J | (j, i) \in P\}$ denote the set of predecessors of activity i and let P_i^* denote all predecessors of activity i up to the start activity.

Let us note that all number domains are natural numbers, therefore we can restrict ourselves to integer computation.

By saying to solve RCPSP, we mean to find a *feasible schedule* with the minimal *makespan* defined as follows.

Definition 2 (Schedule)**.** Assume problem instance G. Schedule S is a vector of start times of activities $s(S) = (s_j)_{j \in J}$. Let vector f denote finish times, $\forall j \in J : f_j(S) = s_j(S) + d(j)$. Schedule S is feasible if it meets precedence 2.1 and resource 2.2 constraints:

$$\forall j \in J, \forall i \in P_j : s_j \geq s_i + d(i) \tag{2.1}$$

$$\forall k \in R, \forall t \in \langle 0, f_{N+1} \rangle : \sum_{j \in J, t \in \langle s_j, f_j)} r(j,k) \leq c(k) \tag{2.2}$$

Definition 3. Let $M(S) = f_{j_{N+1}}(S) - s_{j_0}(S)$ denote the makespan of a solution S.

Example 1. Table 2.1 depicts an example of a schedule with $N = 10$ activities and $K = 2$ resources. Capacities of resources are $c(1) = 7$ and $c(2) = 4$. This example comes from [3, p. 23].

j_i	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}
$d(i)$	0	6	1	1	2	3	5	6	3	2	4	0
$r(i,1)$	0	2	1	3	2	1	2	3	1	1	1	0
$r(i,2)$	0	1	0	1	0	1	1	0	2	2	1	0

Table 2.1: A project with 10 real activities and 2 resources.

Precedence constraints can be displayed as an activity-on-node graph in Figure 2.1, where edge from node i to node j means that $(i, j) \in P$.

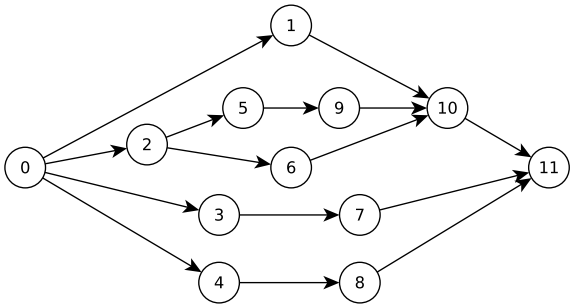


Figure 2.1: Precedence graph for example 1.

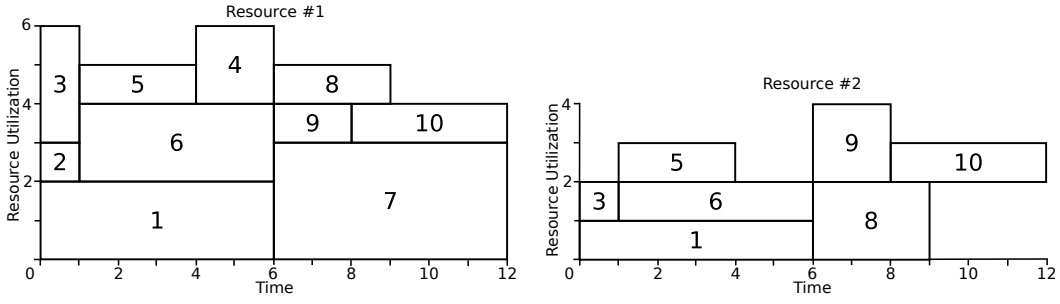


Figure 2.2: The feasible schedule with the makespan 12 for example 1.

In Figure 2.2 we can see a feasible schedule with the makespan 12. Each activity has an assigned color and its number is printed in bottom left corner. It can be checked that precedence and resource constraints are met.

2.2 Computational Complexity of Solving RCPSP

Complexity of RCPSP is one of the reasons why researchers are so interested in this problem. It was proclaimed that RCPSP is one of the most intractable combinatorial optimization problems [3, p.25-26]. It was proven [9] that RCPSP is *NP-complete in the strong sense*. Problem is NP-complete in the strong sense if it is in NP class, is NP-hard and it remains NP-complete even if we restrict all numerical parameters by polynomial in the length of input.

Definition 4 (Decision variant of RCPSP)**.** The decision variant of the RCPSP is the problem of determining whether schedule S of makespan $M(S)$ not greater than H subject to precedence constraints 2.1 and resource constraints 2.2 exists or not [3, p.25-26].

Theorem 1. *RCPSP* $\in$ *NP.*

Further I present some intuition behind proofs of given theorems. Let us have a non-deterministic Turing Machine $\mathcal{M}$. $\mathcal{M}$ guesses schedule S in polynomial time and accepts it if S is feasible and the makespan of S is lower than some value H. We can check precedence constraint in $O(|P|)$. The makespan constraint can be trivially checked in $O(1)$. Resource constraint can be checked by Algorithm 1 in $O(N^2K)$. $\mathcal{M}$ therefore answers the question stated in Definition 4 in non-deterministic polynomial time.

Algorithm 1 Checking of resource constraints [3]

```
1:  L = {f_i | j_i ∈ J}
2:  sort L in increasing values
3:  for t ∈ L do
4:      for r_k ∈ R do
5:          o ← 0, F ← ∅
6:          for j_i ∈ J do
7:              if s_i ≤ t and f_i > t and r(j_i, r_k) > 0 then
8:                  o ← o + r(j_i, r_k), F ← F ∪ j_i
9:              end if
10:             if o > c(r_k) then
11:                 return False
12:             end if
13:         end for
14:     end for
15: end for
16: return True
```

Theorem 2. *RCPSP is NP-hard.*

We use a polynomial reduction of coloring problem to the RCPSP. Let $G = (V, E)$ be an undirected graph and let c be an arbitrary integer. Deciding whether there is a feasible coloring of the nodes in G with c colors such that two adjacent nodes do not share the same color is NP-complete in the strong sense. This graph G is a particular case of the decision variant of RCPSP with a unit duration of jobs, no precedence constraints, a disjunctive resource per arc in E and an activity per node in V with a unit requirement on each resource of its incident arcs. The coloring problem is feasible if and only if the RCPSP has the makespan not greater than c [3].

Theorem 3. *RCPSP is NP-Complete in the strong sense.*

Theorem 3 can be proven by using reduction method with the result of theorems 1 and 2. The proof of strong sense as well as of other theorems is stated in [4].

2.3 Problem Instance Indicators

To describe a problem instance features and hardness of solving it there were described many indicators of problem instances. They are describing average demands on resources, a complexity of precedence constraints and many more schedule features. We can split them into four categories: precedence-oriented, time-oriented, resource-oriented and hybrid. I will only cover three indicators used in classification of a problem instance in the library PSPLIB [20]. For complete overview of instance indicators see [3, chapter 7].

2.3.1 Network Complexity

The network complexity (NC) is the precedence-based indicator. It corresponds to the average number of arcs per activity.

$$NC = \frac{|P|}{N+2}$$

It was observed [21] that problem instances with lower NC are harder to solve than those with greater NC. More arcs in a problem instance means more constraints to precedence order of activities that give us fewer feasible solutions, therefore we have to explore a smaller solution space and the probability of finding an optimal solution is higher.

2.3.2 Resource Factor

The resource factor (RF) belongs to the class of resource-oriented indicators. Let $z(i,k) = 1$ if $r(i,k) > 0$ and $z(i,k) = 0$ otherwise then resource factor is described by equation:

$$RF = \frac{\sum_{j_i \in J} \sum_{r_k \in R} z(i,k)}{K \cdot N}$$

It holds that $0 \leq RF \leq 1$. In this case the instance hardness increases as the RF increases. This is quite intuitive because an activity that has demands on more resources is in conflict over resource with an other activity more often. These activities are harder to place in parallel because together their resource demands often exceeds the resource capacity.

2.3.3 Resource Strength

Last indicator Resource Strength (RS) belongs to the category of hybrid indicators. It combines resource and time features. RS is defined for each resource $r_k \in R$:

$$RS_k = \frac{c_k - c_k^{min}}{c_k^{max} - c_k^{min}} \tag{2.3}$$

where c_k^{min} is the largest activity requirement for r_k and c_k^{max} is the peak demand for resource k in the precedence based early start schedule [3]. Again it holds $0 \leq RS_k \leq 1$.

If $RS_k = 0$ a problem instance has as little resource availability as possible while $RS_k = 1$ equals to the absence of resource constraints. Thus, a problem instances with $RS_k = 0$ are the hardest because it often makes full use of a resource.

2.4 Schedule Classification

In this section I present basic classification of schedules according to their possibility to shift with activities.

Definition 5. *Global left shift operator* $LS(S, j_i, \Delta)$ transforms schedule S into identical schedule S', except for $s'_i = s_i - \Delta$ with $\Delta > 0$. Left shift is *local* when all schedules obtained by $LS(S, j_i, \rho)$ with $0 < \rho \leq \Delta$ are feasible [3].

We could analogically define *global right shift operator* RS.

Definition 6. A schedule is *semi-active* if it admits no feasible activity local left shift.

Definition 6 says that a schedule is semi-active if for each activity $j \in J$ there is no feasible activity left shift of 1 time unit.

Definition 7. A schedule is *active* if it admits no feasible activity global left shift.

We can rephrase definition 7 as a schedule is active if and only if for each activity $j \in J$, there is no feasible left shift of $\Delta \in \mathbb{N}^*$ time units.

Definition 8. *Partial* left shift, which is applicable to preemptive activities, is denoted as $PLS(S, j_i, \Delta, \Gamma)$ and shifts originally non-interrupted activity j_i consists of left shift of a $\Delta > 0$ time units of the first $\Gamma > 0$ units of the activity.

Definition 9. A schedule is *non-delay* if it admits no feasible activity partial left shift.

Non-delay schedules are in other words schedules where no resource is left idle while it could process an activity. Here is important to note that schedules from the set of non-delay schedules may have an empty intersection with the set of optimal solutions, therefore a set of non-delay schedules can contain only sub-optimal solutions [3].

You can see the set relationship between previously described sets of schedules in Figure 2.3. Note that any feasible schedule can be transformed into a semi-active schedule by applying a series of local left shifts. Any semi-active schedule can be transformed into an active schedule by performing a series of global left shifts.

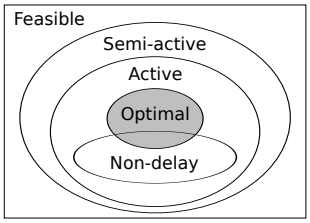


Figure 2.3: Depiction of the set of schedules

Regarding a particular problem instance G we need to distinguish between set of all feasible schedules $\mathbb{S}$ and set of active schedules $\mathbb{S}^A$ so there holds $\mathbb{S}^A \subseteq \mathbb{S}$ [26]. Set $\mathbb{S}$ is infinite because we can generate unlimited number of different schedules that satisfy constraints simply by shifting some activities in the same order to the right and thus increasing a makespan.

2.5 Variants of RCPSP

Problem of project scheduling arises from industry. Basic variant stated above is not flexible enough to model all real world requirements. Industry has more ways to perform an action (multi-mode RCPSP) and has time limitations on activities (maximum time lags) and needs to be more flexible in creating schedules (preemption). Please note that this section serves only as an overview of possible modifications of RCPSP. For details consult [3, part 2]. In the rest of the work we will consider only basic model defined in section 2.1.

2.5.1 Preemptive Activities

Basic RCPSP's activities cannot be interrupted. If we allow this functionality we call them *preemptive*. Of course we can imagine that this interruption brings some penalty to a schedule performance, so preemption is not for free. A project in which all activities are preemptive is also called preemptive. We can derive another forms of preemptive activities: *selective preemption* can occur only in favor of specific activities, in *limited preemption* only restricted amount of interruptions can happen to an activity [3, chapter 8].

We can also distinguish between *integral preemption* and *rational preemption*. Integral preemption allows interruption to occur only at times from domain of natural numbers. Rational preemption, on the other hand, allows an interruption to take place in times from domain of rational numbers and thus breaks integer character of RCPSP. This variant is often solved using mathematical programming techniques, as it seems difficult to predict iteratively at what time the next preemption could occur.

2.5.2 Multi-Mode and Multi-Skill Project Scheduling Problem

Multi-mode resource-constrained project scheduling problem (MM-RCPSP) is a modification of the original RCPSP where we can choose *mode* for an activity in which it will be performed. Another mode of performing an activity also means that an activity will use different amount of resources and processing time. Thus each mode corresponds to a trade-off between resource consumptions and processing time of an activity, e.g. we will complete an activity faster but with more demands on resources. This is one of the most studied extensions of RCPSP because of its obvious application in industry [3, chapter 9].

Multi-skill project scheduling problem (MSPSP) allows resources to be assigned to different kinds of resource requirements, i.e. resource can have multiple roles. Both these modifications lead to a significant increase in the number of feasible solutions. MSPSP can be used to model resources as staff where each person masters several skills.

2.5.3 Non-renewable Resources

Until now we considered only *renewable* resources, i.e. resources that returned after an activity is done (e.g. workforce). *Non-renewable* resource is a resource that we have only limited amount for the whole project and when used it is not returned (e.g. material). There is an extension connected with non-renewable resources and that is the concept of production and consumption of non-renewable resources by activities [3, chapter 10].

2.5.4 Minimum and Maximum Time Lags

When activity i precedes activity j, j cannot start earlier than a start time of i plus duration $s_i + d(i)$. This is called *minimum time lag* [3, chapter 11]. We can define constraint called

maximum time lag or *start-to-start* time, i.e. an activity has to start in the given limit after its predecessor. Interesting about this concept is that even to find a resource-feasible schedule that respects both the minimum and the maximum time lags is NP-hard, without considering a makespan minimization.

Often we speak about release dates and deadlines of activities. Release date is equivalent to a minimal finish-start time lag between j_0 and an activity. Similarly, deadline is a maximal finish-finish time lag between the j_0 and an activity [12].

2.5.5 Resource Requests Varying with Time

Activities in standard RCPSP have constant demands during whole their executions. However, we can generalize this concept to resource varying with each period. This extension was defined because for some activities we often need certain resources only at the end or at the beginning of execution. Requests varying with time can be transformed into constanst requests if maximal time lags are available [12].

2.5.6 Logical Dependencies

Predecessors of a certain activity all have to be finished before the activity can start executing. We can imagine this activity as logical AND node. Standard RCPSP can be extended of OR nodes, where only one or more predecessors have to finish before an activity start, and XOR nodes, where only one preceding activity can be executed not more [12].

2.5.7 Predictive-Reactive Project Scheduling

We have considered project that would never change until now. But the real world is in constant change. An activity may be added or removed, duration of an activity is changed, resource can be suddenly unavailable or have reduced capability. *Dynamic scheduling* and its part *predictive-reactive* approach [3, chapter 12] tries to solve these situations. Briefly, predictive-reactive approach first builds a feasible schedule (predictive) that can be re-optimized when an unexpected event occurs (reactive). In this case we cannot anticipate any disruptions because we do not have any stochastic model of them.

2.5.8 Proactive-Reactive Project Scheduling

This approach to solve dynamic scheduling tries to expect some disruptions and to create a robust schedule that may be more resistant to change [3, chapter 13]. Thus, when any disruption happens reactive phase is easier than in the previous concept. There were efforts to make robust schedules under an uncertainty about an activity duration and resource breakdowns.

2.5.9 RCPSP with Financial Costs

In the basic RCPSP we try to minimize the makespan, i.e. time to accomplish the given project. In a real world project scheduling we often pursue to minimize costs and maximize profit. This variant is a combination of previously described RCPSP modifications [3, chapter 14]. We can have several activity modes, due dates, renewable, non-renewable and partially renewable resource; the way of performing activities impact costs and a lot more. This concepts try to address problems that companies deal with every day.

Chapter 3

Computational approaches

In the previous chapter we defined the RCPSP which is a combinatorial optimization problem. Combinatorial optimization problems are defined by a solution space $\mathcal{X}$, which is discrete, and by a subset of feasible solutions $\mathcal{Y} \subseteq \mathcal{X}$ associated with an objective function $f : Y \to \mathbb{R}$. Task of combinatorial optimization is to find a feasible solution $y \in \mathcal{Y}$, such that $f(y)$ is minimal or maximal [3].

In the particular case of the RCPSP set $\mathcal{X}$ is the set of all schedules with respect to the number of activities, $\mathcal{Y}$ are schedules that satisfy precedence and resource constraints for a given instance of the problem and we seek to minimize objective function M, which returns the makespan of each feasible schedule.

We can optimize a problem instance by two approaches. One is trying to find a schedule with the optimal (minimal) makespan. This means that we need to explore whole state space or use some techniques to reduce it and prove that no better solution than found one exists. I describe some of these algorithms in section 3.3. Exact algorithms are sound and complete but due to RCPSP combinatorial complexity they also could be slow or practically incalculable for large problem instances. Because of this we need to come up with some good enough solution, mostly not optimal, in the reasonable amount of time, i.e. meta-heuristics. Meta-heuristic algorithms is the main topic of my work and I describe them in section 3.4. But before I can describe heuristics and approaches I need to define how to represent a solution in section 3.1 and some of its encoding in section 3.2.

3.1 Solution Representation

In this and the following section I briefly introduce representation and encoding of solutions because it is tightly connected with presented heuristic algorithms [3, chapter 6.1]. Here I present a schedule representation.

3.1.1 Schedule

Obvious way of representing a solution is a schedule itself. A schedule is a vector of starting times for each activity. This is usually output of each algorithm, however, manipulating with a schedule itself can be very hard and inefficient in comparison with other representations, e.g. we could easily get an unfeasible schedule after applying crossover operation. Another reason for not using a schedule as an internal representation is the fact that the set of all schedules is infinite and thus searching in it is more difficult than in a finite set of

active schedules. Thus a schedule representation is considered only as a result and during computation another representation is used.

3.2 Solution Encoding

As we cannot effectively work with schedules in a genetic algorithm we need to use an encoding that we can convert to schedules. I outline main encodings:

Activity List sorts activities in certain order for conversion to a schedule.

Random Key Vector assigns priority to each activity.

Forbidden sets resolve resource conflict in each forbidden set of a project.

Resource Flow Network sets strict order between activities similarly to Forbidden sets without need to enumerate then.

3.2.1 Activity List

As it is introduced further we typically need to choose between more activities when creating a schedule. For that reason we use some sort of priority list. First type of priority list is so called *activity list* (AL). It is a list of all activities in the precedence order. For Example 1 we could write activity list $(A_0, A_4, A_8, A_3, A_7, A_2, A_1, A_6, A_5, A_9, A_{10}, A_{11})$. As you can see an activity list is actually topological sorting of precedence graph. According to [19] the activity list is the most widely used encoding. Let $\mathbb{A}$ denote a space of the feasible activity lists for the given problem and $al(i)$ denote index position of activity $i \in J$ in activity list $al \in \mathbb{A}$.

Some operators, that can be used to modify the solution, can be defined on AL. First operation is called *pairwise interchange* and in practice it swaps two activities in AL, if this swap does not break precedence feasibility. Special case is called *adjacent pairwise interchange*, which swaps only activities that are adjacent in AL.

The second operator is called *shift*. Please note that a shift in context of activity lists is not the same term as in context of schedules. Having an AL we can shift it's one activity to either direction left or right within given distance. In practice it means that if we want to shift activity j to distance 4 to left we will swap j with its neighbor on left side four times, if precedence feasibility is not broken.

In further text we need to combine two ALs together. For this purpose we define binary operator *crossover*. There are three types defined in [11]: *one-point*, *two-point* and *uniform* crossover. I will define only two-point crossover, other can be found in the cited literature. Let have two AL: mother M and father F and two random integers $1 \leq q_1 \leq q_2 \leq N$. When we apply two-point crossover operation we get two children:

$$child_1 = M[0 : q_1], F[q_1 : q_2], M[q_2 : N + 1]$$
$$child_2 = F[0 : q_1], M[q_1 : q_2], F[q_2 : N + 1]$$

If an activity already is a part of child AL, because it was inserted from other parent AL, it is skipped and we continue with a following activity until desired count. It was proven in [11] that this operation does not break precedence feasibility of the AL. You can find an illustration of two-point crossover operation in Figure 3.1.

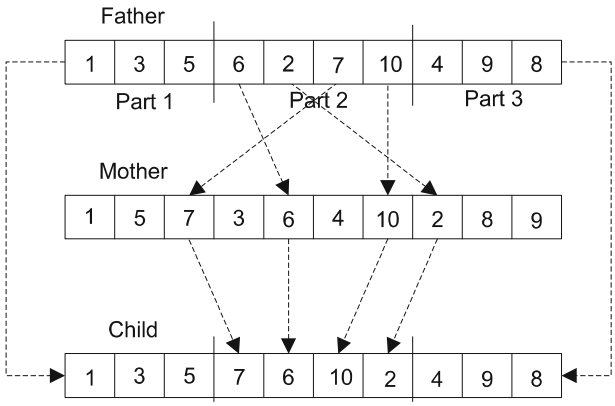


Figure 3.1: Two-point crossover operation [22, Fig. 6]

Interesting modification of two-point crossover which is inspired by [22] is described in [15]. In this modification the points q_1 and q_2 are not generated purely randomly but have certain limits. The first point q_1 is generated from the range $1, \ldots, \frac{2N}{5}$ and the second point is set to $q_2 = q_1 + \Delta$ where Δ is a random number from $\frac{2N}{5}, \ldots, \frac{3N}{5}$. This limits restrict degenerated cases to happen, e.g. where q_1 is too close q_2.

3.2.2 Random Key Vector

Another variant of priority list is the random key vector. In this case we have a list of priorities of each activity. At zero place of the list is a priority of activity A_0, at first place is a priority of A_1 and so forth. A priority is generally a real number. For example we can have this random key vector that corresponds with previously mentioned activity list (11.0, 6.6, 7.2, 8.7, 9.9, 4.4, 5.3, 8.1, 9.3, 3.0, 2.6, 1.8, 0.0). There is an interesting way to standardize random key vector described in [6].

3.2.3 Forbidden Sets

The concept of *forbidden sets* is another way to represent our solution. Forbidden set F is a set of activities between which there is no path in precedence graph and there is a resource conflict on F [3]. If no activity can be removed from that set without removing resource conflict, then F is said to be *minimal* forbidden set. If we add precedence relation between two activities in F, we will solve the resource conflict. Our representation is then a compound of at most one pair of every forbidden set in the problem instance such that all resource conflicts are resolved and there is no cycle in the augmented precedence graph.

3.2.4 Resource Flow Network

The last representation is based on a flow in a network with a capacity from resources. This representation is used to avoid problems with forbidden sets, such as necessity to enumerate all forbidden sets. Using Resource Flow network we set strict order as in Forbidden sets but without need to enumerate them all [3].

3.3 Exact Algorithms

As was already mentioned exact algorithms ensure that found solution is the optimal one. To guarantee such claim we often need to search the whole state space therefore this approach is not feasible on bigger problem instances.

3.3.1 Branch-and-bound

Branch-and-bound methods build a search tree in order to explore implicitly the search space. At each node of the search tree, two situations may occur: a leaf node is reached, therefore a feasible solution can be deduced, or a search space corresponding to the current node is partitioned into subsets in a way that union of these subsets corresponds to the set of solutions in the current node [3, chapter 5].

We decide the way to build a search tree by *branching schemes*. We can use a chronological branching scheme that uses partial schedules and adds one activity per branching step. Using this approach we can control whether a schedule is active/semi-active in every node and cut branches which are not. Second approach is using full schedules, not only partial schedules.

We must note that these branching schemes are used not only as a standalone algorithm for solving RCPSP, but are also used together with heuristic algorithms.

3.3.2 SAT

Reducing RCPSP to satisfiability problem (SAT) [14], which has been historically the first problem proven to be NP-complete, is another exact approach. SAT solver works with propositional formula in conjunctive normal form (CNF). We can express our problem constraints in clauses of this CNF. There are many well optimized SAT solvers, as an example in cited paper is taken MiniSat.

Algorithm for solving RCPSP by reducing it to SAT works with time period of earliest and latest start times and earliest and latest finish times. There are variables that are true if an activity is starting in a given period and other variables that hold true if an activity is processed at given period. With these variables we can build CNF that describes precedence and resource constraints as well as internal rules of RCPSP (no preemption, an activity lasted all its duration, etc.). Because clauses representing resource constraints can be of exponential size and we do not need them all, author in cited paper generates them on the fly during computation of SAT. We need to represent the makespan of given CNF/RCPSP as the last thing. We then ask SAT solver iteratively if CNF is satisfiable with this makespan and if it is we decrease the value of the makespan and start again.

3.3.3 Constraint Programming

Constraint programming (CP) is an optimization technique based on a declarative description of a problem as a set of decision variables with domains, e.g. the start time of an activity with its possible time-windows (the earliest start and the latest finish), and set of constraints restricting the combinations of values the decision variables can take, e.g. precedence constraint between activities [3, p.63]. As we can see its special case is SAT solving.

3.3.4 Linear Programming

Linear programming (LP) is a special case of mathematical programming [3, chapter 3]. It is a method to achieve the best (minimal or maximal) outcome of *objective function* under given constraints which are represented as linear equations. Computing RCPSP depends on chosen mathematical programming formulations. In *sequence-based* formulations we focus on the ordering the activities are processed in. *Time-indexed* formulations focus on the assignment of resources to activities at each time.

3.4 Heuristic Preliminaries

Before we get to actual heuristic algorithms we need to describe various terms used in RCPSP heuristics. First I present schedule generation schemes. These are algorithms that translate a solution in form of a priority list to a schedule representation. We need a schedule to examine a makespan. When we have a schedule generated, we can apply improving technique called *Forward-Backward Improvement*.

3.4.1 Schedule Generation Scheme

The *Schedule Generation Scheme* (SGS) [17] is an algorithm which transforms a priority list to a feasible schedule. A schedule is built iteratively, i.e. only some subset of activities J is scheduled at the given moment. We can distinguish between two major approaches to SGS. The *serial* SGS performs *activity-incrementation* and the *parallel* SGS performs *time-incrementation*.

Serial Schedule Generation Scheme

The *Serial Schedule Generation Scheme* (SSGS) is iterating through activities so it consists of $g = 1, \ldots, N+1$ stages. In every stage one activity is selected and scheduled at earliest precedence and resource feasible time. At each stage g we can define set of scheduled activities S_g, eligible set $D_g = \{j \in J \setminus S_g | P_j \subseteq S_g\}$ are all activities that can be scheduled with respect to precedence constraint at given stage. Let $\tilde{R}_k(t) = r_k - \sum_{j \in \mathcal{A}_t} r(j,k)$ be the remaining capacity of resource k at time t. $F_g = \{f_j | j \in S_g\}$ is set of of all finish times. Algorithm 2 is then SSGS [18].

Algorithm 2 Serial Schedule Generation Scheme

1: $F_0 \leftarrow 0, S_0 = \{0\}$
2: **for** $g \leftarrow 1$ **to** n **do**
3: Calculate D_g, F_g, $\tilde{R}_k(t)$ for $k \in R$, $t \in F_g$
4: Select one $j \in D_g$
5: $ES_j \leftarrow \max_{i \in P_j}\{f_i\}$
6: $f_j = \min\{t \in [ES_j, LF_j - p_j] \cup F_g | r(j,k) \leq \tilde{R}_k(\tau), k \in R, \tau \in [t, t+d(j)] \cap F_g\} + d(j)$
7: $S_g \leftarrow S_{g-1} \cup \{j\}$
8: **end for**
9: $f_{N+1} = \max_{i \in P_{N+1}}\{F_i\}$

All auxiliary sets are computed at the beginning of each stage. We then choose one activity for which we compute the earliest precedence feasible start ES and the latest finish

time LF computed by backward recursion. Then we put activity j into this interval to the first spot where it holds resource constraints.

There is also the possibility not to use set D_g at line 4 and use priority list, i.e. an activity list or a random key vector.

It was proven in [17] that Algorithm 2 yields always feasible, active schedules. This means that SGS with right order of activities (right activity list) will yield the optimal schedule. It was shown that this algorithm belongs to complexity class $O(n^2K)$.

Parallel Schedule Generation Schema

This work uses mainly algorithm SSGS. For completeness I present dual approach to SSGS which is *Parallel Schedule Generation Scheme* (PSGS). PSGS, contrary to SSGS, does time incrementation. Kolisch provided experimental comparison of SSGS and PSGS in [17]. Each stage g of the algorithm corresponds to some time t_g. Activities which have been scheduled up to g are either element of the complete set $C_g = \{j \in J | f_j \leq t_g\}$, i.e. activities which have been completed up to t_g, or the active set $A_g = \mathcal{A}(t_g)$, i.e. activities performed at time t_g. The eligible set $D_g = \{j \in J \setminus (C_g \cup A_g) | P_j \in C_g \wedge \forall k \in K : r(j,k) \leq \tilde{R_k}(t_g)\}$.

Algorithm 3 Parallel Schedule Generation Scheme

1: $g \leftarrow 0$, $t_g \leftarrow 0$, $A_0 = \{0\}$, $C_0 \leftarrow \{0\}$, $\tilde{R_k}(0) \leftarrow R_k$
2: **while** $|A_g \cup C_g| \leq n$ **do**
3: $\quad g \leftarrow g + 1$
4: $\quad t_g \leftarrow \min_{j \in A_g}\{f_j\}$
5: $\quad$ Calculate C_g, A_g, $\tilde{R_k}(t_g)$, D_g
6: $\quad$ **while** $D_g \neq \emptyset$ **do**
7: $\quad\quad$ Select one $j \in D_g$
8: $\quad\quad f_j = t_g + d(j)$
9: $\quad\quad$ Calculate A_g, $\tilde{R_k}(t_g)$, D_g
10: $\quad$ **end while**
11: **end while**
12: $f_{N+1} = \max_{i \in P_{N+1}}\{f_i\}$

Algorithm 3 consists of two major steps. First step on lines 2 - 5 finds next time t_g and computes all auxiliary sets. Second step on lines 6 - 10 schedules subset of eligible activities to start at t_g.

PSGS also always generates feasible solution, but if we allow preemption it generates only a non-delay schedules. A set of non-delay schedules, as was shown in Figure 2.3, not always contains optimal schedules, therefore when we use PSGS we can reach only sub-optimal solutions. Time complexity of PSGS is the same as SSGS $O(n^2K)$.

It is important to note that schedule generation schemes are not injective between priority lists and schedules, in other words we can have two different priority lists that after applying SGS to them we get exactly the same result. There were efforts [6] to standardize random key vector representation so this would not happen again.

Multi Pass Methods

SSGS and PSGS are called *single pass methods* because they run once and generate one schedule. However, we can use different priority rules and obtain more schedules [18]. If we

consider random key vector representation there is virtually unlimited number of priority rules. From one random key vector we can use convex combination of points to generate new random key vectors. This approach is called *Multi Pass Methods*. Again, I present this approach just for completeness as it is not used further.

Another family of algorithms that belongs to multi pass methods is *forward-backward scheduling methods*. Using these we create a schedule in forward way as usual using some SGS. Backward scheduling then means that we reverse precedence graph and create a schedule for an inverse problem. The priority values are usually obtained from the start or finish times of the lastly generated schedule.

Third way to incorporate multi pass methods is *sampling*. It generally uses one SGS and one priority rule. Different schedules are obtained by biasing the selection of the priority rule through a random device. In this case we do not take value as a priority but a probability instead and run an algorithm multiple times, hopefully with different results.

3.4.2 Forward-Backward Improvement

Forward-backward improvement (FBI) [27] or *Double justification* [29] is a technique that helps to improve a generated schedule. We can perform *right justification* in the way that we shift every activity to the right taken in decreasing order (from the end of the schedule) so that the makespan of the whole schedule is not increased. *Left justification* is defined analogically: we shift every activity taken from start of the schedule as much to the left as we can. It was proven [30] that after performing right and left justification we get a schedule with a makespan at worst same as original one and often lower. It may be attached practically to every algorithm right after a schedule generation. This technique is in current research papers plentifully used with promising results [19].

3.5 Meta-heuristic Methods

In this section I introduce main approaches to solve RCPSP with meta-heuristic. You can find current progress in the field of RCPSP heuristics in periodically published articles [19, 13]. It also provides a comparison of these heuristics on the set of problem instances PSPLIB[20].

3.5.1 Simulated Annealing

The *simulated annealing* (SA) is a probabilistic meta-heuristic [16] for the global optimization. The origin of this algorithm comes from annealing in metallurgy, which is a technique of controlled heating and cooling metal to create bigger and better crystals in a material. In the beginning of the algorithm an initial solution is selected. Then the neighborhood of current solution is examined. If some neighbor has better value than current one it is accepted and the algorithm starts again. If it's value is not better neighbor is accepted with probability monotonically decreasing with number of iterations. At start of the algorithm probability to choose inferior state is higher than at the end of computation.

In RCPSP domain the state is obviously a schedule [3]. Its neighbors are then created by shifting some activities. In every iteration is generated few schedules from current one and checked if the makespan of current state is lower than the makespan of a neighbor. If it is, current one is changed, if it is not, we try to change the schedule to an inferior one with

some probability. After we have found an acceptable schedule or time run out we output current solution.

3.5.2 Tabu Search

The *tabu search* (TS) is another improving heuristic with local search [10]. The list of few last states (tabu list) is remembered and avoided when deciding where to move next. This idea of tabu list should prevent from cycling in already examined states. As with SA, TS uses operations like swap and shift to generate neighbors.

3.5.3 Population-based Metaheuristics

Population-based meta-heuristics are generally meta-heuristics where we have a set of solutions called population and each step we create new individuals either from the population or we generate them [5]. Very promising subarea is *genetic algorithms* (GA). GA try to mimic process of natural selection. Individuals in the population are considered according to their fitness function (the makespan in our case). If an individual is found beneficial for the population, his genes (some or all values of his representation) will appear also in the upcoming population. There are operations known from real world used to a new population creation like two individual crossover and mutation.

A genetic algorithm generally works in a loop. In the initialization step we generate a population of individuals. In the second step each individual of a population is evaluated. According to this evaluation we can sort them from the best to the worst. If we meet ending criteria, e.g. an individual with a certain value of objective function was found, we return the best found solution and end the algorithm. We generate a new population otherwise. There are many ways to generate a new population. We can use *elitist selection* for the best individuals to advance to the new population without any change. Other individuals from the current population can be combined by a *crossover* operation and they generate individuals to the new population. This technique combines positive features to obtain even better solution. Last operation we can do on some individuals of the new population is a *mutation*. We randomly change part of their representation to introduce new genes in the population. With the new population we get back to the step of evaluation and repeat until ending criteria are met.

Chapter 4

Genetic Algorithm Driven with Run-Time Hypothesis

Above I presented known algorithms and heuristics to solve the RCPSP. As was stated, exact algorithms are not much helpful for large problem instances, time of computation is often unacceptable. Therefore, extensive research is performed in heuristic algorithms periodically compared in articles [19, 13]. These approaches try to address the problem with different angle of view, using new improved representations, with algorithms inspired by nature [6], using more populations [7] and new scheme generators [23].

One of these new approaches was designed by Martin Hrubý, my supervisor. It is called *Genetic Algorithm Driven with Run-Time Hypothesis* (GARTH) [15] and the goal of this work is to experiment with it and explore possibilities revealed by this new idea. This chapter is based on the original paper.

GARTH is based on the hypothesis that some activities' time relationships in a schedule called *characteristics* prevent us from reaching a schedule with a better makespan. Characteristic is for example parallel start of two activities. Characteristics that appear only in schedules with a high makespan and not in a near to optimal schedules are excluding a solution with a better makespan and therefore we want to rid of them. If we knew what characteristics are excluding an optimal solution we could easily generate the best schedule, however, we cannot determine those characteristics without full enumeration of a state space (and then it would be pointless because we would already have an optimal solution). Therefore, the paper introduces the *Run-Time Hypothesis* that stores all seen characteristics and estimates which are excluding better solution.

4.1 Schedule Characteristics

As was already mentioned in the introduction to this method, GARTH uses a set of time relations between activities called schedule characteristics to depict a particular schedule. Assume problem instance G and two distinct jobs $i, j \in J \setminus \{j_0, j_{N+1}\}$. Let the symbol ξ_{ij} denote a schedule characteristics of type $\xi \in \{PSE, FLE, SLT, INT, SLF\}$

In the work [15] three types of characteristics are defined: PSE (Parallel, Start Equal), FLE (Finish time Lower or Equal) and SLT (Start Lower Than). In this work I add two more: SLF (Start Later than Finishes) and INT (overlaps).

The schedule characteristics PSE_{ij} and INT_{ij} are valid in the context of problem instance G, if parallel run of i and j does not violate their precedence and resource constraint.

The characteristics FLE_{ij}, SLT_{ij} and SLF_{ij} are valid if $j \notin P_i$. Let $\mathbb{CH}$ denote a set of all valid schedule characteristics in G.

Definition 10. Assume schedule $S \in \mathbb{S}$. $\mathbb{CH}_S \subseteq \mathbb{CH}$ is a set of S's characteristics, if for all distinct $i, j \in J \setminus \{j_0, j_{N+1}\}$:

- $s_i(S) = s_j(S) \Leftrightarrow PSE_{ij} \in \mathbb{CH}_S$,
- $f_i(S) \leq f_j(S) \Leftrightarrow FLE_{ij} \in \mathbb{CH}_S$,
- $s_i(S) < s_j(S) \Leftrightarrow SLT_{ij} \in \mathbb{CH}_S$.
- $f_i(S) \leq s_j(S) \Leftrightarrow SLF_{ij} \in \mathbb{CH}_S$
- $(s_i(S) \leq s_j(S) \wedge f_i(S) > s_j(S)) \bigvee (s_j(S) \leq s_i(S) \wedge f_j(S) > s_i(S) \Leftrightarrow INT_{ij} \in \mathbb{CH}_S$

Further I present theorems and lemmas concerned with characteristics excluding solution. Proofs can be found in [15].

Lemma 1 (Feasibility)**.** *For each $\xi_{ij} \in \mathbb{CH}$, there is schedule $S \in \mathbb{S}$ so that $\xi_{ij} \in \mathbb{CH}_S$*

Lemma 1 claims that we can find a schedule, generally non-active, each of valid characteristic of a problem instance.

For following theorems I need to split schedules and characteristics according to a makespan.

Definition 11. Decomposition $[\mathbb{S}]_m$ and $[\mathbb{S}^A]_m$ of a set of schedules according to a makespan defined on the makespan range $m_{opt}, \ldots, m_{max}$ where m_{opt} denotes the optimal makespan and m_{max} denotes an arbitrary finite makespan so that there is no active schedule S with $M(S) \leq m_{max}$ is defined as follows:

$$[\mathbb{S}]_m = \{S \in \mathbb{S} | M(S) = m\}$$
$$[\mathbb{S}^A]_m = \{S \in \mathbb{S}^A | M(S) = m\}$$

We also need decomposition of characteristics.

Definition 12. Decomposition of valid characteristics according to a makespan:

$$[\mathbb{CH}]_m = \bigcup_{S \in [\mathbb{S}]_m} \mathbb{CH}_S$$

Now we can approach to the definition of the excluding characteristic that is crucial part of both this and the original paper.

Definition 13. Assume problem instance G and a particular schedule characteristic $\xi_{ij} \in \mathbb{CH}$. Then ξ_{ij} is said to be **excluding a solution** at a makespan m, if there is no schedule $S \in [\mathbb{S}]_m$ so that $\xi_{ij} \in \mathbb{CH}_S$.

The notion of a characteristic excluding a solution is crucial to understand GARTH. You can find an example of such characteristic in Figure 4.1.

Lemma 2 (Extension)**.** *Let us assume a schedule $S \in [\mathbb{S}]_m$ and its characteristic $\xi_{ij} \in \mathbb{CH}_S$. Then there also exists a schedule $S' \in [\mathbb{S}]_{m'}$, so that $\xi_{ij} \in \mathbb{CH}_{S'}$, for all $m' > m$.*

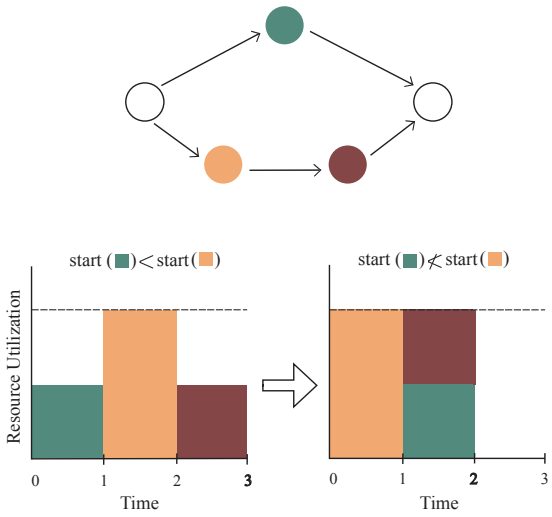


Figure 4.1: In the left schedule characteristic „■ starts before ■“ excludes a solution with makespan 2 provided precedence ordering in the top of the image and displayed resource constraints and activity durations. On the right you can see that once we remove this characteristic we get schedule with the lower makespan.

Extension Lemma 2 claims that once we find a certain characteristic in a schedule at a makespan we find such characteristic in schedules with higher makespans.

Lemma 3 (Inclusion)**.** *If* $m_1 < m_2$*, then* $[\mathbb{CH}]_{m_1} \subseteq [\mathbb{CH}]_{m_2}$*.*

In other words we can sort decompositions of set of characteristics according to a makespan and state following inclusions between them:

$$[\mathbb{CH}]_{m_{opt}} \subseteq [\mathbb{CH}]_{m_{opt}+1} \subseteq \cdots \subseteq [\mathbb{CH}]_{m_{max}} = \mathbb{CH} \tag{4.1}$$

Equation $[\mathbb{CH}]_{m_{max}} = \mathbb{CH}$ holds because of combination Feasibility 1 and Extension 2 lemma.

Theorem 4 (Exclusion theorem)**.** *Assume problem instance* G *and a particular* ξ_{ij} *so that* $\xi_{ij} \in [\mathbb{CH}]_m$ *and* $\xi_{ij} \notin [\mathbb{CH}]_{m-1}$ *for some* m*. Then:*

1. $\xi_{ij} \in [\mathbb{CH}]_{m'}$ *for all* $m' \geq m$*.*
2. $\xi_{ij} \notin [\mathbb{CH}]_{m'}$ *for all* $m' < m$*.*

This means that once we proclaim a characteristic excluding at some level m it cannot appear again in a schedule with a lower makespan. Therefore, having a set $[\mathbb{CH}]_{m_{opt}}$ of characteristics of optimal solutions, then $\mathbb{CH} \setminus [\mathbb{CH}]_{m_{opt}}$ are factually all forbidden characteristics if we want to achieve the optimal schedule.

Unfortunately there also holds following lemma which applies to the domain of active schedules.

Lemma 4. *There is no guarantee that for each* $\xi_{ij} \in [\mathbb{CH}]$ *exists an active schedule* $S \in \mathbb{S}^A$*, so that* $\xi_{ij} \in \mathbb{CH}_S$*.*

Lemma 4 refutes Theorem 4 in the domain of active schedules. The $[\mathbb{S}^A]_m$ can posses characteristics $\xi_{ij} \in \mathbb{CH}$, then $[\mathbb{S}^A]_{m-1}$ do not have to and $[\mathbb{S}^A]_{m-2}$ again can have this characteristic.

At least, Excluding Theorem 4 still holds in $[\mathbb{S}]_m$ domain and whenever some ξ_{ij} excludes some m, then ξ_{ij} surely does not exist in all $[\mathbb{S}^A]_{m'}$, $m' \leq m$.

Theorem 5. *Assume ξ_{ij} excluding makespan m. Then there is no active schedule $S \in [\mathbb{S}^A]_{m'}$ such that $\xi_{ij} \in \mathbb{CH}_S$ for all $m' \leq m$.*

4.2 Solution Encoding

In this section I describe solution encoding used in GARTH. First I present schedule generator R-SGS, which is modified SSGS. Then I outline a normalization of each activity list and a normalization of a population. At the end I describe an individual of a population.

4.2.1 Schedule Generator

GARTH has registered benefits resulting from using Forward-Backward Improvement technique 3.4.2 and uses its alternative form during schedule generating. There was described [29] a schedule generator called SLR-SGS that results in three schedules from one activity list. First regular SSGS is used returning schedule S_0, then right justification is applied to S_0 resulting with S_R and onto this schedule we apply left justification with outcome S_L. It was shown that this technique noticeably improves overall results of an algorithm [29]. Obviously multiple of these steps can be taken, then this technique is then called MLR-SGS.

On the other hand schedule generation and shifting are expensive operations. Number of generated schedules is a criterion in comparison of heuristics of Hartmann and Kolisch [19]. SLR-SGS and MLR-SGS are creating a lot of schedules, precisely 3 and $(2n)+1$ respectively where n is a number of steps taken in MLR-SGS. When we are restricted with the number of generated schedules these SGS will lower number of possible evolutionary cycles that could improve our solution better than SLR/MLR-SGS.

Therefore, GARTH introduces a compromise between the scheduler's performance and its computational cost called R-SGS. R-SGS generates basic schedule S_0 with SSGS and then performs just right justification outputting schedule S_R, which is generally non-active. This right-justified schedule is then *normalized* and in the next iteration converted to left-justified S_L again. This is a similar way as in bi-population method [7].

4.2.2 Activity List Normalization

In genetic algorithms we often focus on preserving some positive features of an individual across generations. In the case of a RCPSP schedule these features are order relations of the activities. We affect order of activities in a schedule with an activity lists but, as we mentioned above, order of activities in a schedule can be different from order of activities in an activity list. We try to suppress this effect using the *activity list normalization*. Assume schedule $S \in \mathbb{S}$ and an activity list $al \in \mathbb{A}$. If the al satisfies criteria 4.2 and 4.3 it is called *normalized* activity list for the schedule S, i.e. activities in the al are sorted by their start time and when start times are equal by their number.

$$\forall i, j \in J : s_i(S) < s_j(S) \Rightarrow al(i) < al(j) \tag{4.2}$$

$$\forall i, j \in J : s_i(S) = s_j(S) \wedge i < j \Rightarrow al(i) < al(j) \tag{4.3}$$

4.2.3 Population Normalization

In the population every individual is costly. We evaluate it, shift with it and these operations cost computation time. Thus, we would like to eliminate redundant individuals, i.e. equal individuals except one. *Population Normalization* (PN) addresses this elimination. If any two activity lists results in the same schedule GARTH replace one of them with a randomly generated new activity list. This keeps the population *diverse* and brings positive source of randomness to the algorithm.

4.2.4 Population Individual

Individual consists of an activity list and a schedule because of necessity to use them both in various points of the algorithm. Thus, one individual in GARTH is a pair (al, S) where $al \in \mathbb{A}$ and $S \in \mathbb{S}$. al can be unnormalized because of the mutation and S is often not active as a result of right justification and may be even empty before the first individual evaluation.

4.3 Run-Time Hypothesis of Excluding Characteristics

Consider problem instance G at the start of the optimization. At the beginning we have no knowledge about characteristics excluding any solution. We generate and evaluate the first individual (al_1, S_1) with makespan $m_1 = M(S_1)$. The presence of characteristics $\mathbb{CH}_{S_1}$ suggests that those characteristics are not excluding a solution at makespan m_1. However, we can assume that they possibly exclude a solution with a makespan $< m_1$ because we have not seen a schedule with a better makespan with these characteristics so far.

We continue by generating another individual (al_2, S_2) with makespan $m_2 = M(S_2), m_2 < m_1$ with characteristics $\mathbb{CH}_{S_2}$. We can update our knowledge about characteristics possibly excluding better solutions. Specifically, characteristics $\mathbb{CH}_{S_1} \cap \mathbb{CH}_{S_2}$ improves their hypothesis from excluding the schedules with a makespan $< m_1$ into solution excluding the schedules with a makespan $< m_2$. As you can see it is still only assumption, only our hypothesis that some characteristic excludes better solution. Our algorithm works in an indicated way: it generates many individuals and updates our knowledge about excluding characteristics with their characteristics and makespans. It learns about a problem instance and assumes that if we have not seen a better solution with certain characteristics, this characteristic is excluding a better solution. We can make use of this information in the way that we find out what characteristic spoils a solution makespan and we can eliminate it to hopefully achieve better solution.

The above described structure is called *Run Time (RT) Hypothesis* for a certain characteristic. We store them in the *RT System* structure as we often want to combine different characteristics together. RT System is a set of the matrices RT_ξ for every characteristic ξ of size $|J| \times |J|$. Elements of RT matrices $RT_\xi(i, j)$ are initialized with some maximal value m_∞. Operation $RT_\xi(i, j) \leftarrow m$ updates the current hypothesis, i.e. $RT_\xi(i, j) := m$ if $m < RT_\xi(i, j)$. If we want to update the whole system we find valid characteristics of

current schedule S and for each of them we update relevant matrix. This operation is denoted as $RT \leftarrow S$ and is described in Algorithm 4.

Algorithm 4 RT System update [15]

Require: Schedule S

for $i \in J$ **do**
  for $j \in J \setminus \{i\}$ **do**
    if $s_i(S) = s_j(S)$ **then**
      $RT_{PSE}(i,j) \leftarrow M(S)$
    end if
    if $f_i(S) \leq f_j(S)$ **then**
      $RT_{FLE}(i,j) \leftarrow M(S)$
    end if
    if $s_i(S) < s_j(S)$ **then**
      $RT_{SLT}(i,j) \leftarrow M(S)$
    end if
    if $f_i(S) \leq s_j(S)$ **then**
      $RT_{SLF}(i,j) \leftarrow M(S)$
    end if
    if $(s_i(S) \leq s_j(S) \wedge f_i(S) > s_j(S)) \vee (s_j(S) \leq s_i(S) \wedge f_j(S) > s_i(S))$ **then**
      $RT_{INT}(i,j) \leftarrow M(S)$
    end if
  end for
end for

With the theoretical results presented in the previous section we have to note two simplifications of the RT system:

- Every characteristics ξ_{ij} excluding a solution within $[\mathbb{S}^A]_m$ is interpreted to be excluding also within $[\mathbb{S}]_m$.
- Every $\xi_{i,j}$ with $m = RT_\xi(i,j)$, $m \neq m_\infty$ is interpreted as a characteristic excluding a schedule within $[\mathbb{S}]_{m'}$, $m' < m$

4.3.1 Individual Sorting

Individuals need to be sorted in order to pick the best one of them. This is done using comparator $\leq_c$:

$$\leq_c (p_1, p_2) = \begin{cases} M(S_1) \leq M(S_2) & M(S_1) \neq M(S_2) \\ U(p_1) > U(p_2) & M(S_1) = M(S_2) \end{cases} \tag{4.4}$$

Symbol $U(p)$ denotes the number of characteristics which updated the RT System during the $RT \leftarrow S$ operation. In other words the first sorting criterion is a schedule makespan of an individual and if makespans equal we decide according to which schedule updated RT System more.

4.3.2 Optimization Driven by the RT-Hypothesis

Optimization using RT-hypothesis is trying to find excluding characteristics that hold for a makespan and then rid of these characteristics. Assume we have some history of generated

activity lists that updated our RT system. We pick a characteristic that possibly excludes a better solutions and then we try to generate a schedule where the characteristic is not present.

Firstly, I describe the process of removing a characteristic excluding better solution. We have an individual (al_1, S_1) where characteristic $\xi_{ij} \in \mathbb{CH}$ is present. If we want to remove characteristic from a schedule, we use operator shift on activity lists defined in section 3.2.1. We shift either activity i or j to arbitrary direction within maximal distance. Please note that this does not ensure removal of characteristic ξ_{ij} from newly generated individual (al_2, S_2), but there is a good chance it will. Also notice that we affect only small part of the schedule and most of other characteristics can remain untouched, therefore shift is an operation mutation in the terminology of genetic algorithms.

Secondly, I present the strategy to pick characteristic possibly excluding a better solution. We want to analyze individual (al, S) and to detect some characteristics $\subseteq \mathbb{CH}_S$ that most likely spoil schedule's makespan. The set $J^w_{ch}(S)|_{RTS}$ contains activities that take part in characteristics excluding a better solution $\mathbb{CH}^w_S$.

$$\mathbb{CH}^w_S = \arg \underbrace{\max_{\xi_{ij} \in \mathbb{CH}_S} [RT_\xi(i,j)]}_{M^w} \tag{4.5}$$

$$J^w_{ch}(S) = \{i, j \in J | \xi_{ij} \in \mathbb{CH}^w_S\} \tag{4.6}$$

RT system therefore remembers the best seen makespan for each characteristics and then searches for the worst makespan of characteristics of a schedule.

The value $M^w \leq M(S)$ represents the highest $RT_\xi(i,j)$ through all $\xi_{ij} \in \mathbb{CH}_S$. This implies that if we study the schedule S with the best makespan m so far we get $\mathbb{CH}^w_S = \mathbb{CH}_S$ and $J^w_{ch}(S) = J \setminus \{j_0, j_{N+1}\}$because all characteristics just updated RT System with makespan m and reduce controlled mutation using RT System to random mutation.

4.3.3 Delayed RT-Hypothesis startup

Author in his experiments discovered that in the early phase of the algorithm the RT System is not trained enough and can result in the wrong mutation decisions. Therefore, he introduced the Delayed RT startup mode that splits the evolution process into two phases. In the first phase there is no mutation employed and this leaves time for RT System to train enough. In the second phase we use the RT System as was described in the previous section. The second phase is started after generating 2/5 of the schedule limit.

4.4 Basic RT Searching Algorithm

Basic RT Searching Algorithm (BRTSA) is a simple population-based local search algorithm that uses RT Hypothesis for the mutation and serves as a proof of the contribution of the RT Hypothesis in the simplest way. It is not a genetic algorithm as it does not employ crossover operation. First of all we need to define function for a population evaluation *popEval* (Algorithm 5). Please note that the result of this function is a vector of pairs where the schedule is right justified and the activity list is different than original one because of activity list normalization.

At last we can approach to BRTSA Algorithm 6. In simple words BRTSA keeps the best individual untouched and mutates the rest of the population either using RT Hypothesis or randomly.

Algorithm 5 popEval(Pop) [15]

function POPEVAL(Pop)
  $Pop' = [\,]$
  for $(al, \cdot) \in Pop$ **do**
    $S := R - SGS(al)$
    $al' := S^N$
    $RT \leftarrow S$
    $Pop' := Pop' + [(al', S)]$
  end for
  return Pop'
end function

Algorithm 6 BRTSA [15]

Require: $popSize, nSelJobs, scheduleLimit \in \mathbb{N}, RTS$
Initialize Pop with $popSize$ randomly generated individuals.
$Pop := popEval(Pop)$.
Let $Best$ be the best individual from Pop.
while $scheduleLimit$ is not reached **do**
  $Pop' = [\,]$
  for $pi = (al, S) \in Pop$ **do**
    Generate maximal $J_s \subseteq J$ so that $|J_s| \leq nSelJobs$ following the experiment mode and configuration:

- $BRTSA : J_s \subseteq J^w_{ch}(S)|_{RTS}$,
- $RTS = \emptyset : J_s \subseteq J$.

    for $j \in J_s$ **do**
      Let dir is randomly chosen from $\{Left, Right\}$
      $al \rightarrow_{j,dir} al'; al := al'$
    end for
    $Pop' := Pop' + [(al', \emptyset)]$
  end for
  $Pop := popEval(Pop')$.
  Update $Best$ with Pop.
end while
return $Best$

4.5 Genetic Algorithm Driven with Run-Time Hypothesis

In the previous sections I presented all building blocks of GARTH Algorithm 7. We can affect the algorithm with following parameters: $popSize$ is the size of the population in each step, $schedulesMax$ limits the number of generated schedules, R_c describes a portion of the activity lists to copy to the new population (elitism strategy),R_m is the portion of ALs

Algorithm 7 GARTH [15]

Require: $popSize, schedulesMax \in \mathbb{N}, R_c, R_m, R_cr \in \langle 0, 1 \rangle$

1: Initialize Pop with $popSize$ randomly generated individuals.
2: $Pop := popEval(Pop)$.
3: Let $Best$ is the best individual from Pop.
4: Let $N_c = R_c \cdot popSize; N_m = R_m \cdot popSize; N_{cr} = popSize - N_c - N_m$.
5: **while** $scheduleLimit$ is not reached **do**
6: Let P_c, $|P_c| = N_c$ is a set of the best individudals from Pop.
7: Let P_m, $|P_m| = N_m$ is a random subset of Pop.
8: $Pop' := P_c$
9: **for** $pi \in P_m$ **do**
10: Mutate pi to $(al', \emptyset)$ in the BRTSA style.
11: $Pop' := Pop' + [(al', \emptyset)]$
12: **end for**
13: **for** $i = 1, \ldots, N_{cr}$ **do**
14: Select randomly parents pi_1 and pi_2.
15: Add the recombination product of pi_1, pi_2 to Pop'.
16: **end for**
17: $Pop := popEval(Pop')$.
18: Normalize Pop if PN is enabled in the experiment's configuration.
19: Update $Best$ with Pop.
20: **end while**

to perform mutation on and at last R_{cr} is the portion of individuals to crossover. Naturally, equation $R_c + R_m + R_{cr} = 1$ must hold.

Mutation described at line 6 in Algorithm 7 works in the same way as in BRTSA.

Crossover operation selects two parents and then applies two-point crossover as defined in section 3.2.1. The first parent is selected randomly from the new population Pop'. Second parent is selected from the current population Pop. Selecting first parent from the Pop' gives the algorithm a better chance for specific genomes (the best and the mutated activity lists) to influence the next generation of the population. Only one child is inserted into the Pop'. According to the author's experiments inserting both children did not improve the algorithm's behavior and reduced diversity of the population.

4.6 Experimental Results

The author reached some interesting discoveries during experimenting with the algorithm.

Author performed experiments on the standard library of RCPSPs PSPLIB [20]. For the specific results and the experiment setup please discuss [15]. The most successful configuration of GARTH was using a population size equal 200, $nSelJobs = 5$, all three characteristics $RTS = \{PSE, FLE, SLT\}$ employing Population Normalization (PN), Delayed RT startup (D-RT) and with Small Mutation (SM) enabled in the first phase of the algorithm.

During experimenting the author achieved improvement of 9 solutions in the hardest dataset J120. He also presented that his approach was on the average better than some state of art RCPSP heuristics.

Chapter 5

My contributions to GARTH

GARTH is a very advanced method for solving RCPSP, but it is also quite novel and it certainly contains few places that can be improved or examined. Goal of this thesis is to study the behavior of GARTH, schedule characteristics and Run Time Hypothesis, in order to suggest new improvements and to evaluate them. I describe these improvements in further text:

Evaluation on the Fly improves evaluation strategy to ensure normalization and to save the schedule generation limit.

Problem Instance Bonding effectively reduces the search space of a problem instance.

Higher Order Characteristics comes with an idea of a characteristic over characteristics.

Crossover Parent Selection improves strategy of picking parents for crossover to favour more successful individuals.

5.1 Evaluation on the Fly

GARTH's program flow is similar to other genetic algorithms. Its generation step consists of copy of best individuals, mutation of some other ones and recombination of individuals both from the old and the new population. But we need to have in mind the fact that we employ individual normalization to the individual during evaluation. Only by using individual normalization we can ensure that every schedule has only one activity list representation. But as we can see in Figure 5.1 operation crossover operates on an unnormalized activity list when one of parents comes from the new unormalized population that can lead to carrying unwanted genes to the next generation.

Evaluation on the Fly (EOTF) addresses this problem. Instead of evaluating each individual in a batch at the end of the generation step, it evaluates them right after their generation, i.e. after mutation and crossover operation. With EOTF we can be sure that crossover works always on normalized activity lists.

The second spot where EOTF can influence GARTH positively is during copying best individuals. In the original GARTH we evaluate them at the end of the generation step together with other individuals. But best individuals come through the step unchanged so we often evaluate the same individuals repeatedly with no improvement. Therefore, we can omit evaluation of some individuals but we have to remember that we can achieve a

better result because of justification while evaluating the same schedule over and over again. Thus, we watch a makespan of best individuals and at the moment when a makespan of a left justified schedule to its left justified ancestor does not improve we mark it to not evaluate again. We cannot watch only changes in one iteration of justification, i.e. from a left justified schedule to a right justified schedule, because schedule's makespan can stay unchanged in forward direction and can improve only in backward direction.

Using EOTF we always operate on normalized activity lists and in addition we save computation time and schedules from schedule generation limit up.

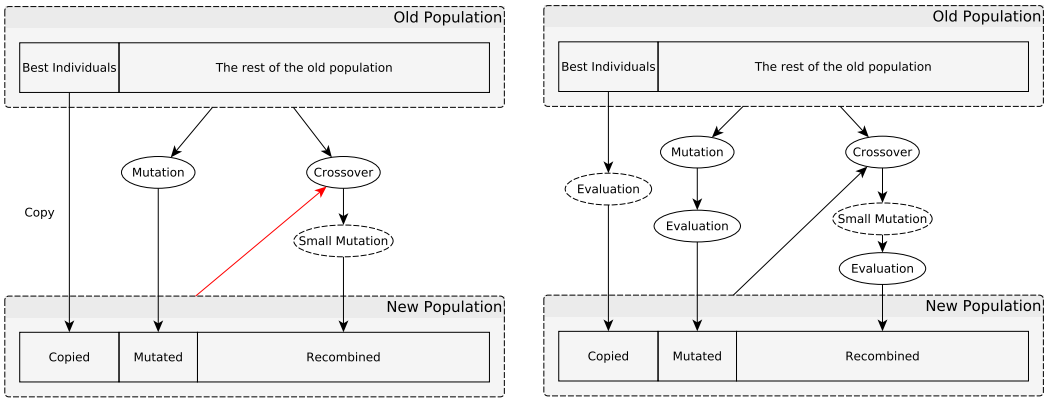


Figure 5.1: Diagram of GARTH genetic step wihout EOTF (left picture) and with EOTF (right picture). The red arrow marks unnormalized individual that enters crossover. You can see in the right picture that the algorithm not always evaluate best individuals.

5.2 Problem Instance Bonding

Another potential improvement that is my contribution to GARTH is the *Problem Instance Bonding* (PIB). In regular GARTH we try to eliminate a certain characteristic that we believe excludes a better solution by shifting those activities in a random direction. This sometimes does not break the unwanted characteristic and the characteristic can return to a schedule in the next generation because of mutation or crossover.

To forbid a characteristic permanently from all future schedules we need to change an original problem instance, we need to add a new precedence constraint. This is the way PIB works. It lets RT system with SLF and INT characteristics to gain needed knowledge about the problem instance. Then it chooses several SLF characteristics and insert edge between activities of such characteristics in reverse order to the original problem instance. To be precise PIB chooses characteristics with the highest makespan that holds equation 5.1 where *threshold* is one of the parameters.

$$\begin{aligned} RT_{SLF}(i,j) \geq threshold \wedge \\ RT_{SLF}(j,i) \leq threshold \wedge \\ RT_{INT}(i,j) \geq RT_{SLF}(i,j) \wedge \\ i \notin P_j^* \wedge j \notin P_i^* \end{aligned} \tag{5.1}$$

In other words PIB searches for activities that have poor results in one ordering and also while running in parallel, and not in the reverse ordering. These activities give us place for a new edge (j, i). Edge is added in a way that it does not create a cycle in a precedence graph, i.e. only between activities that can run in parallel. You can see an illustration of new edges added to an artificial project in Example 2.

Definition 14. Assume problem instance G and two activities $i, j \in J, i \notin P_j^*, j \notin P_i^*$. Adding edge (i, j) to G creates problem instance G' equal to G except $P' = P \cup \{(i, j)\}$.

Theorem 6. *Assume problem instance G and two activities $i, j \in J, i \notin P_j^*, j \notin P_i^*$. After adding edge (i, j) to G we obtain problem instance G' where $\xi_{ji} \notin \mathbb{CH}^{G'}$ for $\xi \in \{PSE, FLE, SLT, INT, SLF\}$.*

Proof. Characteristic $\xi_{ji} \in \{PSE, FLE, SLT, INT, SLF\}$ is not valid in G' because $i \in P'_j$ (section 4.1). □

After adding these new edges to the problem some of activity lists in the population can become invalid because they have activities in order that breaks newly added constraints. To fix them we need to employ the procedure that goes in an activity list from left, takes an activity at the current index and places it only when it holds precedence constraints. If it does not hold precedence constraints it is remembered and tried after next index and so forth until all activities are placed.

The hypothesis behind the PIB is that by restricting a state space of a problem instance we should be able to reach the optimal solution faster. Theorem 8 claims that by removing valid characteristics from a problem instance we decrease the number of schedules at a certain makespan level. Unfortunately, with RT System we can never be sure that a characteristic is indeed excluding a solution at the makespan level. Therefore, this method can be potentially dangerous because by adding constraints we can eliminate optimal schedules from a state space. We try to minimize this effect by letting RT system to learn enough about a problem instance like in the Delayed RT startup, but we cannot ensure that this will not happen.

In the context of problem instance indicators (section 2.3) we are increasing network complexity NC, which decreases with problem instance hardness. PIB is implemented in a way that it adds a number of edges every few iterations as the system is still refining information about a problem instance. Records of forbidden characteristic are erased.

Theorem 7. *Assume problem instance G and two activities $i, j \in J, i \notin P_j^*, j \notin P_i^*$. Adding edge (i, j) to G does not introduce any new characteristic: $\mathbb{CH}^{G'} \setminus \mathbb{CH}^{G} = \emptyset$.*

Proof. Activities i and j can be placed in whatever order even in parallel therefore can generate a number of characteristics they participate in. By adding edge (i, j) we reduce the set of valid characteristics because we add more precedence constraints. □

Theorem 8. *Assume problem instance G, makespan m so that $m_{opt} \leq m \leq m_{max}$, characteristic $\xi_{ij} \in [\mathbb{CH}]_m$. Adding edge (j, i) to G creates problem instance G' where:*

$$|[\mathbb{S}']_m| < |[\mathbb{S}]_m|$$
$$|[\mathbb{S}^{A\prime}]_m| \leq |[\mathbb{S}^A]_m|$$

Proof. Set $[\mathbb{S}]_m$ is finite because only finite number of schedules has makespan m. By adding edge (j, i) we make characteristics ξ_{ij} invalid (Theorem 6) and we do not create any new characteristics (Theorem 7). From the definition of $[\mathbb{CH}]_m$ we know that ξ_{ij} is present in a schedule $S \in [\mathbb{S}]_m$. In G' schedule S is certainly not feasible $S \notin [\mathbb{S}']_m$ because ξ_{ij} is invalid in G'. We have only non-strict inequality in the domain of active schedules because in this domain Lemma 4 holds. □

Example 2. I want to demonstrate function of PIB on this small artificial example. Assume we have problem instance G_2 containing 6 activities and 1 resource as is depicted in Figure 5.2 on the left picture.

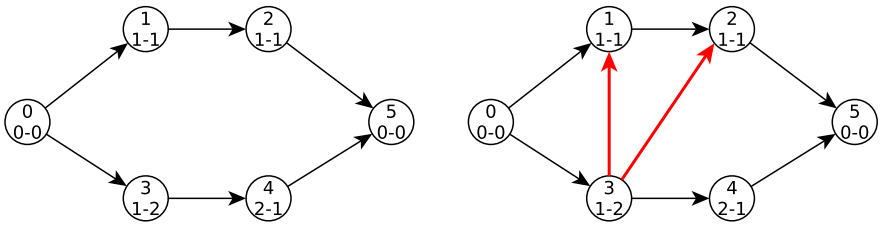


Figure 5.2: On the left is the activity on node graph of the original problem instance. Please note that constraints are displayed in order: duration - demand on the single resource. On the right picture edges (3,1) and (3, 2) are added to the problem instance.

The set of active schedules $\mathbb{S}^A$ of G_2 contains three schedules depicted in Figure 5.3.

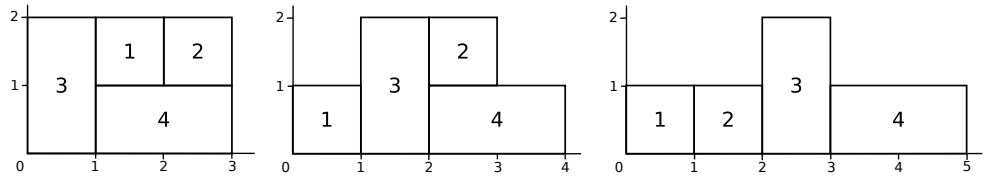


Figure 5.3: All possible schedules for problem instance G_2. On the y axis is depicted resource utilization of single resource and on x axis is time.

To perform informed decision what edge to add we need to have RT System with information about characteristics possibly excluding better solution. Table 5.1 contains complete RT System thus characteristic ξ_{ij} is indeed excluding solutions with a makespan $< RT_\xi(i, j)$ for $\xi \in \{INT, SLF\}$.

We can notice that equation 5.1 with $threshold = 4$ satisfies two pairs: (1, 3) and (2, 3). Thus by adding edges (3, 1) and (3, 2) to G_2 we perform one iteration of Problem Instance Bonding. You may see added edges in the right picture of Figure 5.2. By adding these two edges (actually only (3, 1) would be sufficient) we invalidate schedules with makespans 4 and 5 and therefore only one possible solution remains: the best one with makespan 3.

5.3 Crossover Parent Selection

The third improvement of the original GARTH is the different crossover parent selection. In the original GARTH parents are selected randomly from the new and the old population.

Table 5.1: Complete RT System with characteristics INT and SLF of problem instance G_2.

INT	1	2	3	4
1				3
2				3
3	3	3		
4				

SLF	1	2	3	4
1		3	**4**	4
2			**5**	5
3	3	3		3
4				

However, in many genetic algorithms we employ some strategy to choose better individuals more often for crossover, i.e. we try to add *selection pressure* to the evolutionary cycle. Selection strategy brings a new impulse to the evolutionary process of GARTH. It does not only strive to eliminate the worst characteristics of the a but also to breed the better individuals together to hopefully combine their best characteristics.

In [24] there are described three main selection strategies: tournament, roulette wheel and rank-based roulette wheel selection. I decided to implement tournament selection and rank-based roulette wheel because they were evaluated as noticeably better than basic roulette wheel.

5.3.1 Tournament Selection

Tournament selection is the easiest selection strategy with good features. In tournament selection n individuals are selected randomly from a larger population and they compete with each other, i.e. we choose the one with the best makespan as you can see in Figure 5.4. The usual number of chosen individual is 2 and this variant is called the binary tournament. Higher number of individuals are increasing speed of convergence but for the price of diversity of the population. Tournament among fewer individuals gives a higher chance to all individuals. Tournament selection is easy to implement and does not require fitness scaling or sorting.

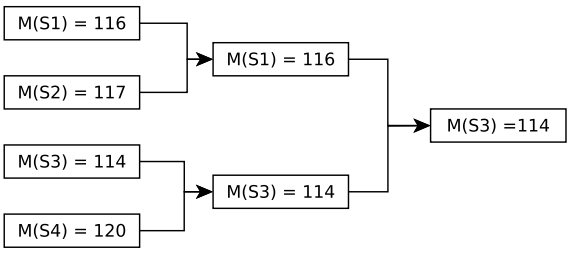


Figure 5.4: Tournament selection among 4 individuals.

5.3.2 Rank Based Roulette Wheel Selection

In a Rank based roulette wheel selection individuals are selected with probability that is directly proportional to their rank, i.e. order in the sorted population. The probabilities can be seen as spinning a portion of a roulette wheel. Better individuals get bigger portions of the wheel. We can imagine selection of an individual as spinning the wheel and returning an individual which portion ends with the ball. Obviously, those with the largest rank are more likely to be chosen.

We can have different mapping functions that map a rank to a number of portions of a wheel. Those can be linear or nonlinear. I chose linear function that assigns $popSize$ portions to the first individual, $popSize-1$ to the second and so forth to 1 portion to the last individual in the population. One important difference between tournament selection and rank based selection is that tournament selection gives the same probability to individuals with the same makespan while rank based selection gives them probability according to the rank that depends on comparator $\leq_c$ (section 4.2.4).

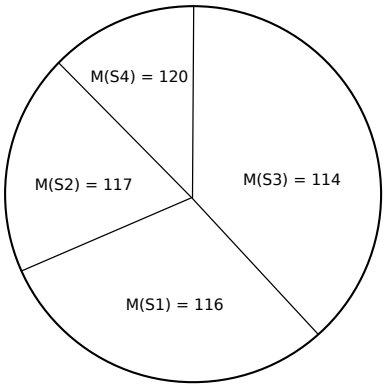


Figure 5.5: Roulette wheel for rank selection in a population of size 4.

5.4 Higher Order Characteristics

The last extension is more theoretical effort to understand schedule characteristics than an actual improvement. It discuses the idea of having characteristics not between activities but between another characteristics, e.g. assume we have characteristics SLT_{ij}, SLT_{kl} in problem instance G, our new characteristic examines makespans of schedules where are SLT_{ij}, SLT_{kl} present at the same time. We call this new characteristic *Higher Order Characteristic* as it is characteristic of characteristics.

Formally we can define new characteristic as follows. Higher order characteristic $\xi^2_{I_{i,j},J_{k,l}}$ of type $\xi \in \{PSE, FLE, SLT, INT, SLF\}$ where I and J are characteristic of the same type ξ is valid in problem instance G if $I_{i,j}, J_{k,l} \in \mathbb{CH}$. Technically we are holding information about such characteristics in the RT Hypothesis matrix as with the basic characteristics. Only thing that differs is the size of the matrix: $N^2 \times N^2$ which represent quite high computational overhead. Because of the size of RT Hypothesis we do not intend to use this improvement in the real algorithm for all problem instances but we only want to use it to examine this approach. By experimenting with higher order characteristics we would like to know if the higher level information allows us understand an examined problem instance more.

Chapter 6

Implementation

To conduct all experiments that are the essential part of this work I needed to create a program that implements GARTH. At first I decided to implement the algorithm in Python language. Python has very strong scientific base and many libraries that could help to easily implement GARTH and focus to the development of the method. This first prototype was named Ukko[1].

Unfortunately, after programming this prototype I ran a speed test and it was obvious that this program was too slow to experiment seriously. Therefore, I decided to reimplement the whole algorithm in C++ language which is compiled and can provide more optimization than program in interpreted language. This second try, which I named Perkele[2], was indeed faster than Ukko and I could start experimenting with it.

In next sections I shortly describe the implementation of Ukko and then more deeply the implementation of Perkele. Then I provide a brief user guide.

6.1 Ukko

Ukko is implemented in Python version 2.7. I used scientific library NumPy that is written in C which promises better tools for numerical computing and in general faster command execution that pure Python. Ukko can read a given problem instance, create internal representation and run basic GARTH on this instance. I ran my speed test as early as was possible so Ukko is not complete and even can contain bugs, although I tried to program with the method Test Driven Development that should reveal bugs that normally would occur much later.

When I discovered how slow Ukko is, I first tried to optimize it. At first I revealed few redundant code execution. Then I started to profile code using standard Python profilers `cProfile` for function scoped profiling and `kernprof` for line scoped profiling. I was able to speed up my program by the factor of three but it was still not enough for experimenting in the scale I needed. I decided to rewrite most used class `Schedule` to Cython, thus translate Python code with some annotations to C and then compile it. I hoped that this rapidly increases speed of the implementation but it did not happen. Program was as fast as the previous version because of frequent call to Python interpreter that I could not eliminate. The next way to increase speed was to rewrite `Schedule` class to C++ and provide interface

[1] The name Ukko comes from Finnish mythology. Ukko is the god of the sky, weather, harvest and thunder. Also we can translate it as old man.

[2] Perkele also comes from Finnish and means exactly same thing: the god of the sky, weather, harvest and thunder.

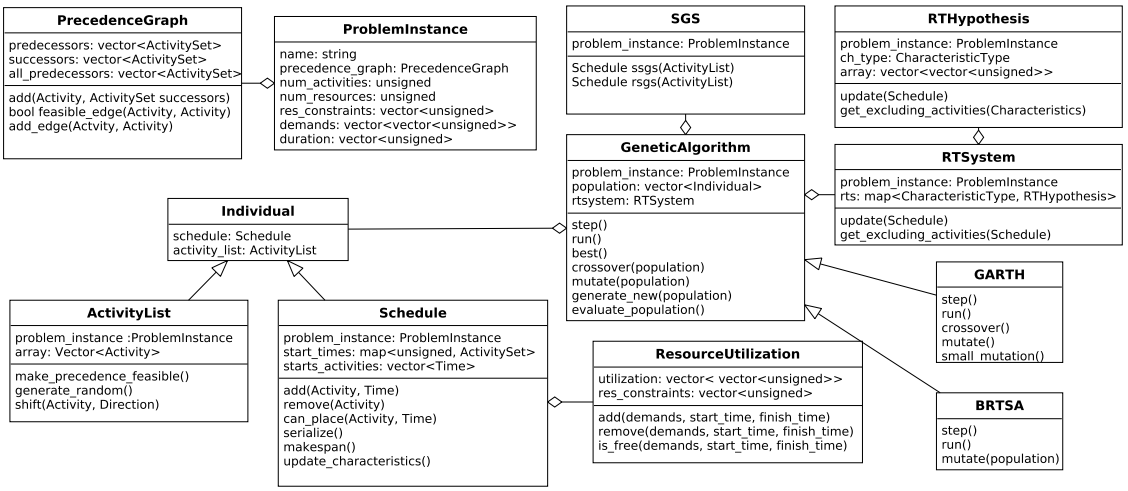


Figure 6.1: UML class diagram showing relationships between Perkele classes

between Python and C++. Using this way I would have to convert arrays from NumPy to C++ arrays and that would be certain computational overhead. At this point I decided to drop Ukko and reimplement it in C++.

6.2 Perkele

As I already had my prototype I could use it as a blueprint for my new program. I reimplemented classes and functions one to one from Python to C++. Of course some Python idioms and libraries were not possible to be used in C++ so I implemented them from scratch. The result of this effort was Perkele, a program with same functionality as Ukko (at the time), but that ran up to 100 times faster than Ukko. With this speed I was able to start serious experimenting.

Perkele is a command line application written in C++ using modern language standard C++11. It stores results in PostgresSQL database for easy evaluation. Because C++ allows writing in various paradigms I decided to use mainly object oriented approach because of legacy design from Ukko.

In Figure 6.1 you can see the most important classes of Perkele. Bellow I provide basic description of each class:

`ActivityList` contains array representing an activity list and operations like an activity shift and operations regarding an activity list feasibility.

`BRTSA` implementation of BRTSA algorithm for basic comparison with GARTH.

`GARTH` implementation of GARTH algorithm, essential part of Perkele.

`GeneticAlgorithm` superclass of `BRTSA` and `GARTH` providing container for the population and operations on them like evaluation and generating.

`Individual` wraps `ActivityList` and `Schedule` to one object as they are tightly connected in the algorithm.

`PrecedenceGraph` holds information about an activity-on-node graph.

`ProblemInstance` structure containing information about a given instance of RCPSP.

`ResourceUtilization` class representing the capacity of resources of each schedule in any moment of the algorithm.

`RTHypothesis` array holds information about characteristics and makespans of best seen schedule with those characteristics.

`RTSystem` class grouping several `RTHypothesis` together.

`Schedule` class that models schedule and operations on them.

`SGS` provides algorithms SSGS and RSGS.

The program also contains few files with functions that are not represented by any class like `rcp_parser.hpp` for parsing input files, `database.hpp` to communicate with PostgresSQL database and `tests.hpp` for unit testing.

6.2.1 Implementation of Essential Structures

In this section I describe the most used and the most interesting structure from the implementation point of view. Specifically I present:

- `Schedule`
- `RTHypothesis`
- `RTSystem`
- `PrecedenceGraph`
- `ResourceUtilization`
- `GeneticAlgorithm`

The most interesting structure of Perkele is `Schedule`. `Schedule` together with `Activity List` forms `Individual` which is a crucial part of every genetic algorithm. In `Schedule` we need to represent activities' start and finish times. This provides C++ `std::map` container with the time as key and a set of activities as value. `std::map` is implemented as binary search tree [1] which provides easy iterating in sorted order which is needed in a schedule serialization. `Schedule` also contains characteristics in form of vector of activity pairs.

`RTHypothesis` and `RTSystem` are another representative structures of Perkele. The essential part of `RTHypothesis` is characteristic matrix RT_ξ defined as vector of vector of times where the size of each vector is equal to the number of activities in a problem instance, i.e. `RTHypothesis.array[i][j]` equals to $RT_\xi(i,j)$. To get the set $J^w_{ch}(S)$ we call function `RTHypothesis.get_excluding_activities(Characteristics)` that assigns the best seen makespan to vector of characteristics of an examined schedule and then sorts them according to the makespan. It returns a predefined number of activities that takes place in the characteristics with the worst makespans. `RTSystem` only stores all `RTHypothesis` that are used in current program run. It also retrieves excluding activities of all `RTHypothesis` and returns their intersection.

`PrecedenceGraph` serves as model of precedence constraints between activities. We need to check only immediate successors and predecessors in the algorithm. The only exception

is adding an edge because in PIB where we need all predecessors of a certain activity to be sure that only edges between parallel activities are added. All three structures are modeled as a vector of a set of activities where index of a vector denotes an activity.

`ResourceUtilization` is a part of a schedule that models resource constraints of a problem instance. For each resource it contains vector initialized to sequence of resource capacity. Whenever new activity i is added to a schedule from start time $s(i)$ to finish time $f(i)$ values of resource capacity vector is decreased of the activity demand to the resource in given interval (interval is closed from left and open from right). Naturally an activity is added only if it does not violate resource constraints.

`GeneticAlgorithm` is superclass of `GARTH` and `BRTSA`. It provides basic structures and methods needed in the genetic algorithms. It stores `Population` as vector of individuals and `RTSystem` to make informed decision using it. It has two methods affecting a run of the algorithm: `step()` for one generation step and `run()` that repeats `step()` until ending criterion is satisfied. As a part of a generation step it provides `generate_new()`, `crossover()`, `mutate()` and `evaluate_population()` methods.

6.2.2 Database

I use PostgreSQL database to store results of each long running experiment. The schema of tables is depicted in Figure 6.2. In table `experiments` is stored name of each experiment, when it started and ended, resulting average deviation and parameter setting for easy evaluation. Each test run carries information about solved problem name and the result , i.e. a makespan, a deviation from critical path lower bound, an activity list and a count of generated schedules. Table `psplib_results` serves only for ad hoc queries about problem instances.

Figure 6.2: Database schema of the experiment results storage.

6.3 User Guide

In this section I provide setup of Perkele and brief description of usage options.

6.3.1 Setup

Perkele uses CMake build system so it can be compiled on various platforms. Development and testing, however, were performed only on Linux OS specifically Fedora 21. The PostgreSQL library `libpq` is the only build time requirement. Program can be compiled with these commands:

```
perkele/ $ cd build
perkele/build/ $ cmake ..
perkele/build/ $ make
```

This will create executable binary file `perkele`.

6.3.2 Usage

To control program computation I designed command line arguments. To have an overview of program capabilities please run

```
$ ./perkele -h
```

that will write out simple help and usage string. Perkele provides more than 30 options to control its run.

To solve a problem instance the user has to select a path to a RCP file containing description of a problem instance and optionally parameters controlling behavior of the program. A launch of the program may look like this:

```
$ ./perkele -f ../psplib/psplib/30-13-1 -A garth -C PSE,SLT
Best makespan: 58
Generated schedules: 13104
0 2 3 1 5 4 6 10 14 11 17 20 7 13 23 8 9 18 30 26 16 22 12 15 27 19 21 24
25 28 29 31
```

This command started Perkele on problem instance 13-1 from the set J30 using GARTH and two characteristics PSE and SLT. Output of program contains information about the resulting makespan, number of generated schedules until the solution was found and a normalized activity list of that solution.

Algorithm has several settings that are described in section 7. I present basic setting of some experiments.

BRTSA

```
$ ./perkele -A brtsa -p 25
```

GARTH-3

```
$ ./perkele -S -m 0.0 -r 0.9
```

GARTH-7

```
$ ./perkele -S -R
```

My improvements are activated by flags or parameters described in `./perkele -h`. Here follows an example where I run the algorithm with as many added features as possible.

```
$ ./perkele -f ../psplib/psplib/60-13-1 -S -R -C PSE,FLE,SLT,INT,SLF -l 500000
 -E -K -b -H 5 -I 10 -u 20
```

Chapter 7

Experiments Evaluation

With Perkele program from last chapter I finally can approach to experimental evaluation of GARTH. You can get an impression of the best results in the field from Table 7.1. Before I present my results please note that I am trying to improve the third best algorithm among RCPSP solvers [15]. Every one hundredth of improvement in J60 and five hundredths in J120 should be counted as success. Please note that my results are not the same as in the original paper. This can be explained by two reasons. Firstly the original implementation is developed and tuned for much longer time than Perkele and secondly this kind of program is very dependent on the small implementation details, e.g. order of elements in a container, and these are not and cannot be completely present in the paper about GARTH. I provide the comparison of my results and the results from the original paper but other experiments will be considered relatively to my results.

Table 7.1: Overview of the results of best RCPSP solvers with 50000 schedule limit, sorted by the results on J120.

Author	J60	J120
Lim et al. (2013) [22]	10.63	30.66
Debels and Vanhoucke (2007) [8]	10.68	30.82
Hrubý (2015) [15]	10.67	30.85
Valls et al. (2008) [28]	10.73	31.24
Mendes et al. (2009) [23]	10.67	31.44

Firstly I describe the problem instance set and the environment for experiments. Then I conduct a basic schedule characteristics evaluation to show some interesting attributes of them. In the third section I compare the results of Perkele with the results originally published in [15]. After that follows the evaluation of possible GARTH improvements which was described in section 5. This chapter extends and improves results presented in [25].

7.1 Experiments Setup

The struggle to find or generate a sufficient unbiased set of testing data is often a necessary evil to a researcher before the actual experimenting. Fortunately, in the area of RCPSP solvers there exists widely used set of testing problem instances called PSPLIB [20] used in the most significant algorithms as objective function of the algorithm performance [19]. Because the original paper [15] uses PSPLIB I use it in my experiments too.

Table 7.2: Values of problem instance indicators for J30 and J60

Indicator	Levels			
NC	1.50	1.80	2.10	
RF	0.25	0.50	0.75	1.00
RS	0.20	0.50	0.70	1.00

Table 7.3: Values of problem instance indicators for J120

Indicator	Levels				
NC	1.50	1.80	2.10		
RF	0.25	0.50	0.75	1.00	
RS	0.10	0.20	0.30	0.40	0.50

PSPLIB contains sets for single mode RCPSP and multi mode RCPSP. GARTH is solving single mode RCPSP only thus we restrict ourselves to this dataset. Single mode RCPSP PSPLIB contains three sets of problem instances with 30, 60 and 120 number of activities, these sets are labeled J30, J60 and J120 respectively. J30 and J60 contains 480 instances and J120 contains 600 instances. Each instance was generated using PROGEN [21], the instance generator for RCPSP. Instances are generated with varying instance indicators (section 2.3): network complexity NC, resource factor RF and resource strength RS. Indicator levels for J30 and J60 are in Table 7.2 and for J120 are in Table 7.3. Each problem instance has 4 resources. Every setting of indicator levels then generates 10 instances of RCPSP. You can see that combinations of indicator levels give us number of instances $3 \cdot 4 \cdot 4 \cdot 10 = 480$ and $3 \cdot 4 \cdot 5 \cdot 10 = 600$. Indicator level permutations are in order that RS is changing most often and NC least often, e.g. instance 13 of J30 has $NC = 1.50$, $RF = 1.00$ and $RS = 0.20$.

The result of running Perkele on a problem instance is a schedule with a certain makespan. The success rate of the schedule is given as the percentage deviation from optimal solution in the case of J30 and percentage deviation from the critical path lower bound for J60 and J120 denoted as M_u^{opt}. Let M_u^{avg} stand for the average of achieved makespans and $U \in \{J30, J60, J120\}$. The equation for the evaluation experiment run is then:

$$D_U = \frac{1}{|U|} \sum_{u \in U} \frac{M_u^{avg} - M_u^{opt}}{M_u^{opt}}$$

Critical path lower bound is the length of the longest path in the activity-on-node graph which is easily computed using topological ordering, as we can see in Algorithm 8, because an activity-on-node graph is a directed acyclic graph. Weight of DAG is a duration of an activity.

The program is launched 10 times for each instance in the set to reduce influence of chance. This gives us 4800 and 6000 computations for J30, J60 and J120 sets respectively. The overall result of this experiment is the average number of percentage deviation as described above.

I run these experiments on school computational grid because such number of test runs is quite computationally demanding. In particular I use Sun Grid Engine (SGE) [2] at the Department of Inteligent Systems at FIT BUT. Each computation of a problem instance is a single task in the terms of SGE. After computation finish the resulting schedule with its makespan and deviation is sent to the PostgreSQL database where results are stored

Algorithm 8 Longest path in a directed acyclic graph

Require: Weighted DAG G = (V, E)
Ensure: Largest path cost in G
1: Topologically sort G
2: **for** each vertext $v \in V$ in linearized order **do**
3: $dist(v) = \max_{(u,v)\in E}\{dist(u) + w(u, v)\}$
4: **end for**
5: **return** $\max_{v\in V}\{dist(v)\}$

and in the end aggregated to the average of all deviations. Results of all experiments are stored in the database together with parameters of Perkele in that particular experiment. This experiment setup allowed me to comfortably test new ideas and evaluate these ideas in minimal time.

7.2 Schedule Characteristics Evaluation

The first of my experiments deals with schedule characteristics, number of their occurrence and their relationship to the problem instance indicators. As far as I know schedule characteristics have never been experimentally evaluated. In my experiment I randomly generate 1000 activity lists, evaluate them using only SSGS and count the number of characteristics in resulting schedules. I run my experiments on set J30 but the results are very similar for J60 and J120 too.

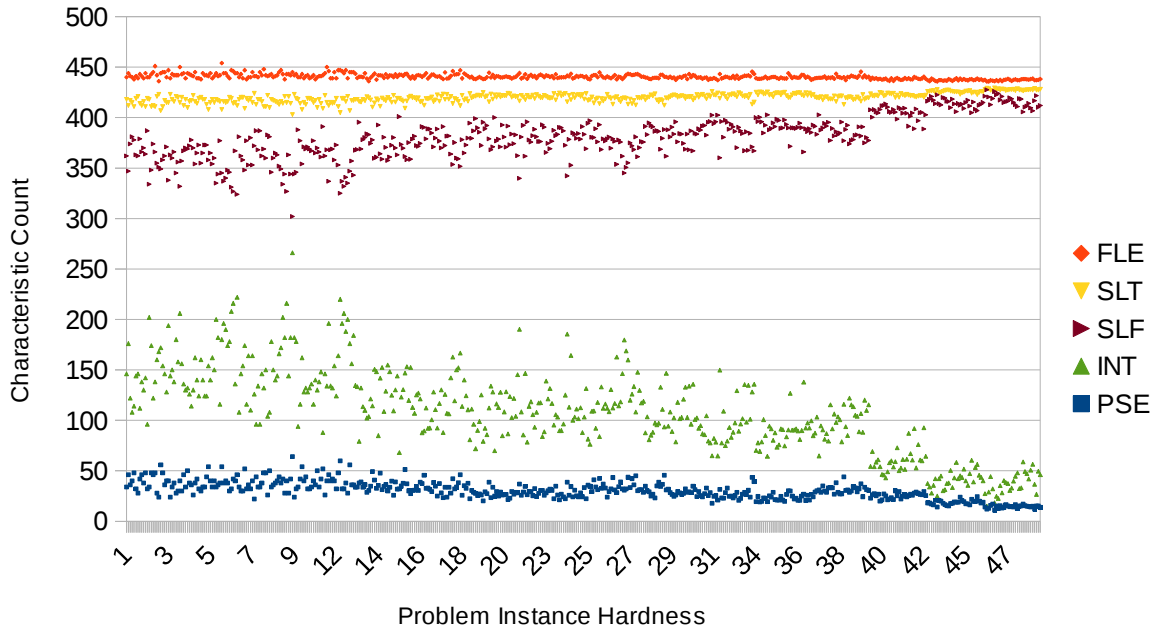


Figure 7.1: The count of schedule characteristics in 1000 randomly generated schedules per schedule to assumed problem instance hardness.

You can see the results of my experiment in Figure 7.1. There I outlined the number of particular schedule characteristic to the problem instance hardness. The hardness is

an ordering of problem instances from easiest (value 1) to hardest (value 48). We order problem instances by indicators where the most important is resource strength, second is resource factor and the least significant is network complexity. At first you can note average number of characteristics. The most frequent characteristic is FLE followed by SLT and SLF so called *asymmetric* characteristics. These characteristics are present in a schedule if some ordering of start or finish times between activities holds. A plenty of activities start before others because of the precedence constraints, e.g. activity 1 has quite possibly lower start time than activity 20 in the most schedules. This finding however decreases the information value of some characteristics as some are always present in all active schedules of the problem instance. On the other hand so called *symmetric* characteristics PSE and INT, that rely on activities time proximity, have much lesser counts of occurrences in the same schedules, i.e. it is easier to find activities where one starts before another than activities which start almost at the same time.

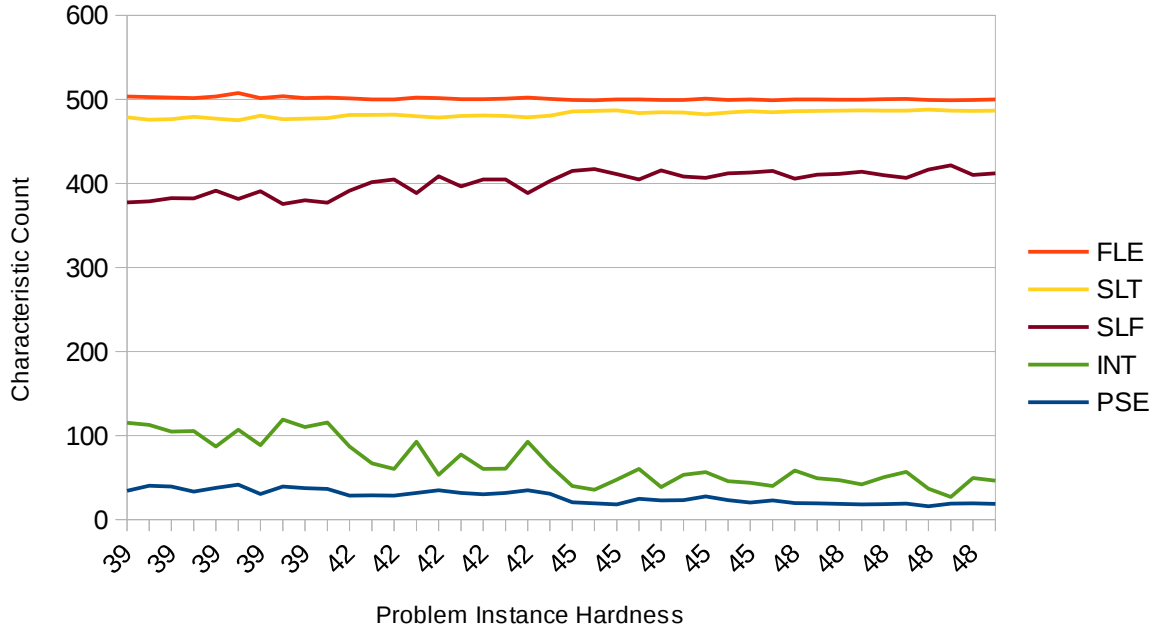


Figure 7.2: The detail to the count of schedule characteristics in 1000 randomly generated schedules per schedule to the resource factor indicator. Only the part of dataset where $RS = 0.2$ and $NC = 1.5$ is considered so we can focus on the effect of the resource factor.

Another piece of knowledge we can reveal is the number of characteristics according to problem instance indicators. As was already mentioned in section 2.3 problem hardness increases with increasing resource factor, with decreasing network complexity and resource strength. Thus, a problem instance is harder if it contains activities with more demands to resources that have not much extra capacity regarding demands and have rather loose precedence constraints. Thus, the hardest problem instance should be an instance with $NC = 1.5$, $RF = 1.0$ and $RS = 0.2$ in the settings of the J30. This assumption can be indeed observed as the problem instance with the number 13 corresponding to mentioned indicators takes often most time to solve with an uncertain result. In Figure 7.1 we can notice that the number of particular characteristics depends on hardness of the problem instance. Let take INT as an example. The harder a problem instance is it allows less ac-

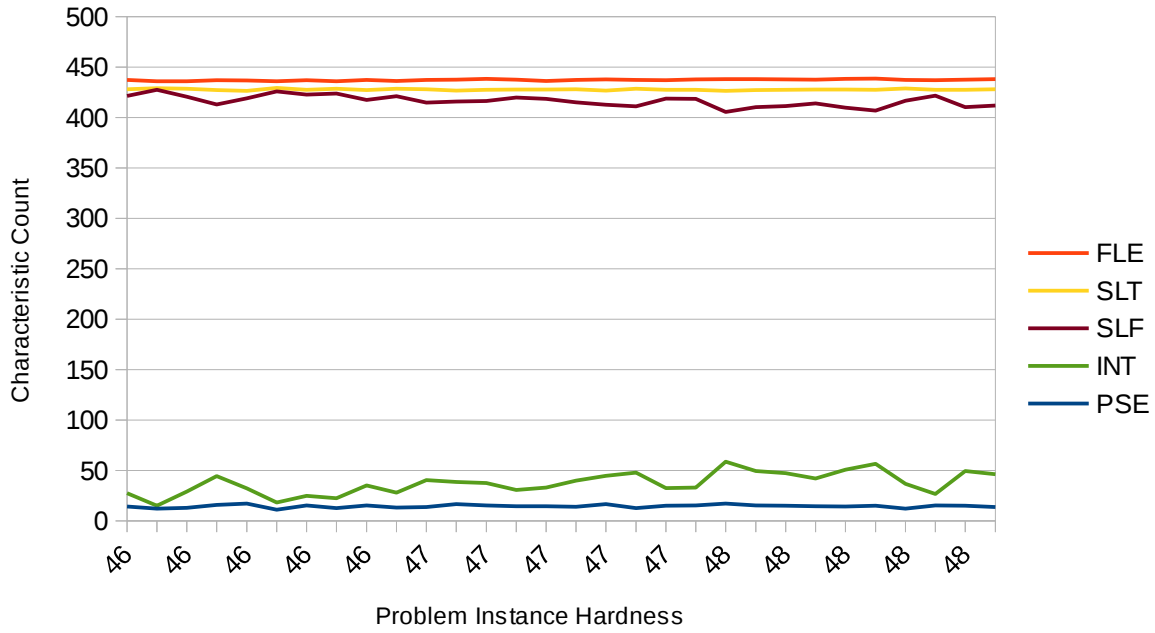


Figure 7.3: The detail to the count of schedule characteristics in 1000 randomly generated schedules per schedule to network complexity indicator. Only the part of dataset where $RS = 0.2$ and $RF = 1.0$ is considered so we can focus on the effect of the network complexity.

tivities to run in parallel because resource capacity is often fully utilized by a single activity. We can notice some kind of duality between asymmetric and symmetric characteristics. A count of asymmetric characteristics increases as a problem instance is getting harder while a count of symmetric characteristics decreases as the problem is getting harder.

The biggest impact on the count of schedule characteristics and a problem instance hardness has the resource strength indicator Figure 7.1 followed by the resource factor indicator Figure 7.2. The network complexity has only little significance as you see in the Figure 7.3. This is unfortunate as the network complexity is the only parameter we can quite safely change by adding new edges in Problem Instance Bonding 5.2.

7.3 Original Paper Experiments

One of the most important parts of my thesis is the confirmation of the results stated in the original paper [15]. Only experiments that were proven in different environment by another researcher can be considered trustworthy. I am going to repeat only the most important experiments that can show whether GARTH in my implementation works as expected.

7.3.1 BRTSA

The first experiment that should show the contribution of the Run-time Hypothesis by itself is running BRTSA 4.4 with various schedule characteristics. In the original paper the author compares two strategies of picking characteristics to mutate. The usual one picks characteristics with the maximal makespan in RT system while the second one picks

characteristics with the minimal makespan in RT system. The second strategy is there only for comparison and is not used further. Thus, I decided to omit it. I am also leaving out the evaluation with schedule limit equal 5000 schedules as it has little informative value.

Table 7.4: The BRTSA experiment results.

Dataset	RTS	Original	This paper
J60	∅	11.21	11.15
	PSE	11.10	11.10
	FLE	10.91	11.04
	SLT	10.95	11.04
	SLF	-	11.02
	INT	-	11.05
J120	∅	33.53	33.46
	PSE	32.92	33.35
	FLE	32.60	33.07
	SLT	32.63	33.09
	SLF	-	33.09
	INT	-	33.17

You can see the comparison of BRTSA results in Table 7.4. Although my implementation does not reach to the results of the original paper, we can see positive effect of using RT System. We always get better results than with random mutation using whichever characteristic. Another interesting fact we can see in Table 7.4 is that characteristic PSE is constantly worse than other characteristics. This probably can be explained by noticeably lower count of PSE in comparison to FLE, SLT and SLF. On the contrary another parallel characteristic INT has better results than PSE. PSE is present only when characteristics have the same start time while INT is present if activities overlap. With INT we apparently can express more about an examined problem instance.

7.3.2 GARTH

More important results lies in the evaluation of GARTH method itself. Author ran GARTH in various configurations to show how each part of GARTH (small mutation, a population normalization, etc.) influences the final result. I pick only the most important test configurations to compare our results. I chose GARTH-3 and GARTH-7, you can find their configurations in the Table 7.5.

Table 7.5: GARTH experiments configuration.

Experiment	R_{cr}	R_m	Configuration
GARTH-3	0.9	0	SM, PN
GARTH-7	0.9,0.7	0,0.2	PN, D-RT

Basically GARTH-3 runs without mutation and ($R_m = 0$) and all new individuals are made entirely by the crossover operation with Small Mutation (SM) activated. We also apply Population Normalization (PN) here. GARTH-7 runs with delayed RT startup activated so first phase is exactly the same to GARTH-3. The second phase turns mutation on ($R_m = 0.2$) and turns Small Mutation off. Algorithm is affected by RT Hypothesis only in the second phase. You can see the results of experiments in Table 7.6.

Table 7.6: The GARTH experiment results.

		J60		J120	
Experiment	RTS	Original	This paper	Original	This paper
GARTH-3	-	10.77	10.84	31.36	32.20
GARTH-7	$\emptyset$	10.69	10.79	31.10	31.78
GARTH-7	PSE, FLE, SLT	10.67	10.76	30.85	31.59

In this experiment you can see that my results are again slightly worse than the original one. But it copies a trend given by the original paper. We can see that GARTH-3 is worst in all configurations because RT Hypothesis is not activated. GARTH-7 with random mutation is a bit better because in this algorithm we employ both operations mutation and crossover. Finally GARTH-7 with $RTS = \{PSE, FLE, SLT\}$ achieves the best result as it fully uses RT Hypothesis.

You can note that I achieved similar results as was stated in [15]. These results confirm usefulness of RT Hypothesis.

7.4 Decision Tree Experiment

This experiment tries to examine the idea that each problem instance could be prone to be solved faster or better with a particular schedule characteristic. As we have seen in section 7.2 there is a connection between a schedule characteristic count and problem instance indicators. If there was a connection between the speed of the convergence of GARTH and used characteristic we could deduce rules to assign schedule characteristics to a problem instance and achieve better results. Unfortunately, said assumption did not prove right. I ran BRTSA on the dataset with a schedule characteristics because I hoped that in BRTSA the differences between schedule characteristics would become most apparent. As you can see in Table 7.7, which is the excerpt of the result of this evaluation, results differs only minimally so we cannot deduce correct decision tree that would choose a schedule characteristic according to a problem instance. This experiment unfortunately did not bring any improvement. The only lesson we can learn from this experiment is that results of GARTH have a small variance and they are not much dependent on the chosen characteristics.

Table 7.7: Average deviation from critical path lower bound for BRTSA for some problem instances according to a schedule characteristic.

Instance	PSE	SLT	FLE	INT	SLF
60-8-10	0	0	0	0	0
60-9-1	50	50.16	50.16	50.16	50
60-9-2	17.88	18.02	18.02	18.02	18.16

7.5 Evaluation on the Fly

This experiment examines Evaluation on the Fly as was defined in section 5.1. In this experiment I compare results of running GARTH-7 on J60 and J120 with and without EOTF. You can see the results of this comparison in the Table 7.8.

Table 7.8: Results of experiment with and without Evaluation on the Fly.

Set	Characteristic	GARTH-7	EOTF
J60	PSE	10.766	10.758
	FLE	10.775	**10.738**
	SLT	10.769	10.759
J120	PSE	31.695	31.583
	FLE	31.667	**31.507**
	SLT	31.665	31.587

As you can see the improvement is not striking but is consistent as the change of the algorithm is only small. There are two factors that are helping EOTF with improvement. First is strict normalization of all individuals that enter crossover. Second is the possibility not to evaluate the best individuals that was not changed in the last generation step.

7.6 Problem Instance Bonding

This section describes experiments regarding Problem Instance Bonding introduced in section 5.2. As in the previous section I compare run of Perkele on the sets J60 and J120 but this time only on RT System consisting of $RTS = \{SLF, INT\}$ because they are essential to PIB and other characteristics do not influence the result much. I conducted the evaluation of two bonding parameters: number of added edges and number of iterations between PIB activation. Experimentally evaluated parameters of PIB are shown in Table 7.9. The results of the experiment are in Table 7.10.

Table 7.9: Parameters for Problem Instance Bonding. $M(Best)$ is the makespan of the best individual in the current population.

Number of added edges per the PIB activation	20
Number of iterations between the PIB activation	5
Threshold	$M(Best) + 5$

Table 7.10: Problem Instance Bonding evaluation.

Set	Characteristic	GARTH-7	PIB
J60	SLF, INT	10.78	10.75
J120	SLF, INT	31.72	31.71

We get a slight improvement for set J60 but practically the results for J120. We could expect better enhancement after so much space reduction but we need to be aware of the conclusion from section 7.2: network complexity influences a problem instance hardness only minimally. If we could somehow affect rather resource strength or resource factor we would get much better results. However, we cannot change resource capacity and demands to do so.

I conducted a research on the results from the last experiment. I compared an average result of a problem instance with and without PIB on set J60. Then I took those results that performed better with PIB and assigned problem instance indicators to those results. You can find counts of each results of this improving instance in Table 7.11. We can discover that

improved solution are those that are considered as hard. This is obvious in the indicator RS because most improvements happened at the hardest level 0.2. RF indicator has similar trend but not so convincing as RS. NC indicator is uniformly distributed because it has a small influence to a hardness of a problem instance. Improvement only in the hardest problem instances makes sense because only them run long enough to apply PIB. Algorithm achieves an optimal solution faster on easier problem instances.

Table 7.11: Indicator's count of problem instances with the improving results regarding PIB.

RS level	Count
0.2	47
0.5	10
0.7	0
1	0

RF level	Count
0.25	1
0.5	12
0.75	19
1	25

NC level	Count
1.5	19
1.8	18
2.1	20

7.7 Crossover Parent Selection

In this experiment I examine the influence of the crossover selection strategy as was described in section 5.3. This feature should add greater selection pressure to the algorithm because more successful individuals are more likely to become parents of a new child and thus spread their fine genes to the next population. I conducted experiments with tournament and rank selection on GARTH-3 and GARTH-7. I also have EOTF turned on during these tests because the algorithm needs to have individuals from the new population evaluated immediately.

Table 7.12: Crossover Parent Selection, GARTH-7 has $RTS = \{PSE, FLE, SLT\}$

Experiment	Selection	J60	J120
GARTH-3	-	10.84	32.20
	Tournament 2	**10.73**	**31.40**
	Tournament 4	10.78	31.40
	Tournament 8	10.84	31.69
	Rank	10.76	31.67
GARTH-7	-	10.76	31.59
	Tournament 2	10.73	**31.11**
	Rank	**10.72**	31.21

As you can see in Table 7.12 these results are the best results from all presented improvements. The result of GARTH-3 with the Tournament 2 is surprisingly improved to the level of GARTH-7 without selection strategy. Tournaments with 4 and 8 individuals are notably worse than run without any selection strategy. This behavior is most probably due to a loss of diversity in crossover operation because only the best individuals are recombined.

GARTH-7 with Tournament 2 does not bring any improvement in comparison to GARTH-3, but GARTH-7 with Rank selection is even better than Tournament 2 by one hundredth which is so far best result. This holds only in J60. Rank selection on the set J120 has worse

results than tournament selection on the same problem set. The reason why Rank is doing better on J60 and not on J120 with presence of controlled mutation requires more research.

7.8 Higher Order Characteristics

In this experiment I examine Higher Order Characteristics which was described in section 5.4. First experiment is a comparison between characteristics SLT and SLT^2 in the configuration GARTH-7 on the set J60. You can find results in the Table 7.13. You can note that Higher Order Characteristics does not bring improvement. This happens most likely because of the size of RT Hypothesis SLT^2. The size of RT Hypothesis SLT is 3 600 and the size of SLT^2 is 12 960 000. With the matrix so big it is much harder to train RT System enough to provide well informed decisions.

Table 7.13: GARTH-7 with Higher Order Characteristics.

Characteristic	ξ	ξ^2
SLT	10.772	10.788
PSE	10.773	10.791
FLE	10.775	10.788

In the next experiment I try to increase the schedule limit to 500 000. Because this experiment would be too computationally demanding on the whole dataset, I decided to present result only on one problem instance with SLT characteristic. I run instance 60-13-1 10 times. This problem instance is known for its hardness.

Table 7.14: Higher Order Characteristics evaluation with the schedule limit 500,000.

Instance	SLT		SLT^2	
	Makespans	Avg Deviation	Makespans	Avg Deviation
60-13-1	112, 112, 112, 113, 113, 113, 113, 113, 114, 114	63.62	112, 113, 113, 113, 113, 114, 114, 114, 114, 115	64.49

As you can see in Table 7.14 this experiment did not prove a positive effect of Higher Order Characteristics either. This approach is too computationally demanding in comparison with standard characteristics to be used in standard optimization. The size of the RT Hypothesis matrix is too big to be trained enough to give better results than with normal characteristic.

7.9 Combination of Improvements

The last experiment tries to achieve the best result with combining all above mentioned improvements. We do not have to get the best result just by combining all extensions together as some new features can work against each other. You can see the overall results in Table 7.15.

From the results we can see that the biggest impact on the improvement has the crossover selection. It can be considered as the most valuable improvement of this work. We can also note that running EOTF and PIB together does not improve the results much,

Table 7.15: Evaluation of combinations of improvements with the characteristics $\Xi_1 = \{PSE, FLE, SLT, INT, SLF\}$, $\Xi_2 = \{INT, SLF\}$.

Configuration	J60		J120	
	Ξ_1	Ξ_2	Ξ_1	Ξ_2
GARTH-7	10.77	10.78	31.73	31.70
EOTF, PIB	10.77	10.76	31.71	31.69
EOTF, PIB, Tournament	10.74	10.74	**31.15**	**31.14**
EOTF, PIB, Rank	**10.73**	**10.72**	31.21	31.22

it rather eliminates benefits of each improvement. PIB on set J120 actually has no influence on the results in comparison to test only with EOTF and rank selection Table 7.12.

During my tests I ran over 370 experiments which makes 1.8 million computations of a problem instance and over 30 billion evaluated schedules. The results of GARTH with my improvements would take the fourth place in comparison with other RCPSP solvers (Table 7.1), right after the results of the original GARTH implementation.

I managed to find an improving solution for three problem instances as you can see in Table 7.16. The improving activity lists are in Appendix A.

Table 7.16: The improved solutions.

Problem Instance	Original Makespan	My Makespan
120-9-4	87	86
120-19-9	89	88
120-48-5	111	110

Chapter 8

Conclusion

In this work I described state of art of resource-constrained project scheduling problem. I defined basic problem and terms used. I deduced complexity of computing a schedule for the given makespan. I described all main modifications of RCPSP that are currently being discussed. I also outlined main approaches to solve RCPSP with exact algorithms and described the most successful heuristics, along with the representation of solution in them.

I chose a genetic algorithm called GARTH [15] with very promising results for further research. I analyzed this algorithm and picked the most interesting results of the author's experiments. I designed 4 improvements of the basic GARTH: Evaluation on the Fly, Problem Instance Bonding, Crossover Parent Selection and Higher Order Characteristics.

I implemented two prototypes of GARTH, one in Python language and second – more advanced – in C++. With the second prototype I conducted a series of experiments on the PSPLIB, library of problem instances for RCPSP. I experimentally analyzed schedule characteristics and the relationship between count of each type to problem instance indicators and hardness. I also compared results from the original paper with my result and confirmed usefulness of GARTH. I evaluated my improvements to the algorithm and analyzed their results. Evaluation on the Fly, Problem Instance Bonding and Crossover Parent Selection are provable improvements of the original algorithm. My best result with the original GARTH on the problem sets J60 and J120 is 10.76, 31.59 of average deviation from critical path lower bound. With my extensions I improved the result to 10.72 and 31.11. I managed to improve 3 solutions of problem instances in PSPLIB. I attended student conference Excel@FIT [25] and presented my contribution with a poster.

GARTH and schedule characteristics certainly deserve more research. Currently the usage of GARTH on vehicle routing problem is researched. Modified GARTH can be used also in other areas like knapsack problem or nurse scheduling problem. Another direction that could be examined are variants of RCPSP, e.g. maximum time lag and non-renewable resources. The results of this thesis can be integrated to the original GARTH that hopefully brings an improvement of the results. Generally, the idea of eliminating undesirable individuals from the population can become a new trend in genetic algorithms.

Bibliography

[1] C++ reference: std::map. http://www.cplusplus.com/reference/map/map/. Accessed: May 26, 2015.

[2] Sun grid engine. http://www.fit.vutbr.cz/CVT/cluster/sge.php. Accessed: May 26, 2015.

[3] Christian Artigues, Sophie Demassey, and Emmanuel Neron. *Resource-constrained project scheduling: models, algorithms, extensions and applications*, volume 37. John Wiley & Sons, 2010.

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

[5] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[6] Dieter Debels, Bert De Reyck, Roel Leus, and Mario Vanhoucke. A hybrid scatter search/electromagnetism meta-heuristic for project scheduling. *European Journal of Operational Research*, 169(2):638–653, 2006.

[7] Dieter Debels and Mario Vanhoucke. A bi-population based genetic algorithm for the resource-constrained project scheduling problem. In *Computational Science and Its Applications–ICCSA 2005*, pages 378–387. Springer, 2005.

[8] Dieter Debels and Mario Vanhoucke. A decomposition-based genetic algorithm for the resource-constrained project-scheduling problem. *Operations Research*, 55(3):457–469, 2007.

[9] Michael R Garey and David S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[10] Fred Glover. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[11] Sönke Hartmann. A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics (NRL)*, 45(7):733–750, 1998.

[12] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010.

[13] Sönke Hartmann and Rainer Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 127(2):394–407, 2000.

[14] Andrei Horbach. A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, 181(1):89–107, 2010.

[15] Martin Hrubý. A dynamic analysis of resource-constrained project scheduling problems for an informed search via genetic algorithm optimization. *European Journal of Operational Research*, 2015. In review.

[16] Scott Kirkpatrick, MP Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

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

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

[19] Rainer Kolisch and Sönke Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, 2006.

[20] Rainer Kolisch and Arno Sprecher. Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program. *European Journal of Operational Research*, 96(1):205–216, 1997.

[21] 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.

[22] Andrew Lim, Hong Ma, Brian Rodrigues, Sun Teck Tan, and Fei Xiao. New meta-heuristics for the resource-constrained project scheduling problem. *Flexible Services and Manufacturing Journal*, 25(1-2):48–73, 2013.

[23] Jorge Jose de Magalhaes Mendes, Jose Fernando Gonçalves, and Mauricio GC Resende. A random key based genetic algorithm for the resource constrained project scheduling problem. *Computers & Operations Research*, 36(1):92–109, 2009.

[24] Mohd Razali Noraini and John Geraghty. Genetic algorithm performance with different selection strategies in solving tsp. 2011.

[25] Martin Hrubý Petr Šebek. Evaluation of schedule charcteristics in RCPSP. In *Excel@FIT 2015*.

[26] Arno Sprecher, Rainer Kolisch, and Andreas Drexl. Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 80(1):94–102, 1995.

[27] Pilar Tormos and Antonio Lova. A competitive heuristic solution technique for resource-constrained project scheduling. *Annals of Operations Research*, 102(1-4):65–81, 2001.

[28] Vicente Valls, Francisco Ballestin, and Sacramento Quintanilla. A hybrid genetic algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 185(2):495–508, 2008.

[29] Vicente Valls, Francisco Ballestín, and Sacramento Quintanilla. Justification and rcpsp: A technique that pays. *European Journal of Operational Research*, 165(2):375–386, 2005.

[30] Jerome D Wiest. Some properties of schedules for large projects with limited resources. *Operations Research*, 12(3):395–418, 1964.

Appendix A

Improved Solutions

I improved results of three problem instances during experimenting with Perkele. Here I present their activity lists.

Table A.1: The improved solutions.

Problem Instance	Makespan	Activity List
120-9-4	86	0 1 2 3 17 18 20 37 4 7 29 5 19 40 46 28 6 8 9 13 35 56 23 32 22 21 34 48 15 11 12 59 10 24 33 49 79 44 60 62 57 16 25 53 30 42 87 43 72 84 47 90 45 55 31 51 61 63 58 81 69 65 75 14 82 83 86 73 97 68 95 27 89 52 36 70 78 91 93 92 74 77 64 80 111 39 66 102 41 50 76 96 54 38 71 100 112 88 98 99 26 103 104 106 85 113 116 94 105 109 107 108 115 101 110 114 67 117 118 119 120 121
120-19-9	88	0 1 2 3 7 9 11 4 5 6 8 43 81 13 18 29 12 27 23 24 17 20 22 25 28 16 14 15 40 21 26 30 31 54 34 36 49 61 51 38 85 37 41 45 10 32 52 71 78 47 92 57 98 35 62 68 84 53 64 76 39 58 73 89 48 50 67 69 65 86 60 87 94 46 82 70 33 83 93 79 44 66 77 100 55 75 80 108 88 91 110 104 63 103 19 59 97 56 74 90 96 105 111 113 102 101 106 112 114 95 99 109 42 107 120 115 116 117 72 118 119 121
120-48-5	110	0 1 2 3 5 11 17 9 4 18 23 27 45 7 8 12 16 22 24 30 49 25 52 13 21 31 26 10 20 51 37 46 14 36 40 44 28 56 72 6 19 33 58 39 29 32 53 73 60 78 35 43 48 50 41 15 82 42 54 100 34 62 63 79 65 47 59 74 68 88 57 64 55 61 38 66 86 87 71 90 93 94 102 67 69 81 97 75 77 83 91 105 80 92 101 95 107 89 96 110 103 76 70 84 104 98 108 85 111 106 99 112 115 113 109 114 120 117 116 119 118 121