Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Bachelor's thesis

Scheduling algorithms and their applications

Jiří Stránský

Supervisor: RNDr. Tomáš Valla, Ph.D.

15th May 2014

Acknowledgements

I would like to thank my supervisor RNDr. Tomáš Valla, Ph.D. for his leadership and patience. Next I would like to thank my colleagues for their support and Faculty of Information Technology for the required knowledge.

Declaration

In Prague on 15th May 2014 ....

Citation of this thesis

Stránský, Jiří. *Scheduling algorithms and their applications.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2014.

Abstrakt

Tato práce popisuje algoritmy pro generování studentského rozvrhu na univerzitách. Práce zároveň obsahuje porovnání algoritmů z teoretického i praktického hlediska. Jsou navrženy a analyzovány vlastní unikátní algoritmy a spolu s již existujícímy algoritmy jsou použity k sestrojení komplexního modelu pro generování rozvrhů.

Klíčová slova rozvrh, student, univerzita, generátor

Abstract

This thesis describes algorithms for generation student schedules at universities. Comparison of algorithms is also included from both theoretical and practical point of view. Unique self-designed algorithms are introduced and together with existing algorithms are used to design a complex model for generating the schedules.

Keywords schedule, student, university, generator

Contents

List of Figures

List of Tables

Introduction

In this thesis, we will focus on schedules at universities. Universities from around the world operate many different ways, but all of them have to deal with a very important problem. The problem is scheduling classes between students, teachers, classrooms and available time windows. This problem has been studied for a long time and offers a variety of approaches, each of them leading to different solutions.

Motivation

Since it has been written a great deal of papers about this problem, we will focus a bit more on one of its possible solutions. We claim, that many universities use a certain algorithm, leading to another problem, which needs to be solved. So why would anyone use it? Because the following problem does not need to be solved by a university anymore, as it is a problem for students now.

Let us talk more about how the solution we are talking about works. The university has teachers, rooms, subjects to be taught, students and a defined time quantum per week. Each student has to select subjects he is interested in in advance. Now the university has to make a schedule for everyone, but to make the task a little bit easier, students are not considered individually in the schedule. The university creates so-called study groups. By knowing the number of students interested in each subject and a capacity of a single lesson i.e. maximum size of a study group, the university can determine the required number of lessons per week to meet the demand. Different lessons then divide students of each subject into different study groups. So the task is to make such weekly schedule, where there are enough lessons for each subject, taught by a particular teacher, in a particular classroom and at particular time. And of course, no teacher can teach multiple lessons at the time, similarly as there can not be more than one lesson in each classroom at the same time.

Now, when the university schedule is ready, it is released to the students. At this moment, every student has a guaranteed place in his selected subjects, but what is not guaranteed, is that there is such combination of study groups for each of his subject, that he will not have to be at multiple classes at the same time. At this moment, students have to sit down and try to find a set of study groups which suit them. Now, if the student wants to visit 5 classes, it is not that big of a deal, but if you need 10 classes, each offering 10 lessons, where time collisions between lessons are relatively frequent, it is starting to be actually pretty difficult task.

Goals

In this thesis, we will survey recent papers and books for suitable methods. We will also try to design our own methods to approach the problem. Selected methods will be implemented and evaluated on benchmark datasets. Based on results we will discuss the best possible approach to the problem.

Thesis organization

In Chapter 1 we define the computational model. In Chapter 2 introduce the problems we will be focusing on. Then in Chapter 3 we will describe already existing algorithms that can be used. In Chapter 4, we will describe self-designed solutions. In Chapter 5 we will compare the results from both theoretical and practical view.

Chapter 1

Computational model

To analyze complexity of algorithms, we need to define a computational model. We choose Random Access Machine (RAM).

RAM is a theoretical computational model consisting of the following components.

- *Memory* of RAM consists of integral memory cells addressable by non-negative integers. Each memory cell can hold a single integer.
- *Program* is a finite sequence of sequentially executed instructions of the following types.
  - Arithmetic and logic instructions such as *add*, *subtract*, *multiply*, *and*, *or* and others.
  - Control instructions such as *goto*.
  - Conditions such as *if*.
  - Indirect addressing.

Now we have to specify the number of available memory cells. This is limited by the size of each cell, because with the limited cell size, we have limited number of cells we can address.

We will use a polynomial RAM model. This model declares that for each program there is a polynom $p(N)$, such that input of size N memory cells can store in any memory cell number of size up to $p(N)$. We will call this model a *unit-cost* model, because each instruction in this model takes $\mathcal{O}(1)$ time.

Polynomial RAM model does not allow us to use exponential memory. Because algorithm in Section 4.1 requires exponential memory, we will offer another model for this particular problem. We choose *logarithmic-cost* model. In this model, instructions cost is scaled relatively to the size of its operands. The cost of one instruction is $\left\lceil \frac{b(X)+b(Y)+b(Z)}{\log_2 N} \right\rceil$, where $b(X)$, $b(Y)$ and $b(Z)$

are number of bits in the input and output operands and N is a number of memory cells in the input.

Input is defined as a sequence of memory cells, where a number of memory cells determines the size of the input.

Output is defined as a sequence of memory cells, containing a required result of the program.

Chapter 2

Problem Statements

Before we get to the problem statements, we have to define the following notation.

- *I:* Instance representing a study group.
  Formally a half-open interval $[b(I), e(I))$, where $b(I) < e(I)$
- *A:* Activity representing a subject.
  Formally a non-empty set of instances.
- *S:* Schedule S represents a students schedule.
  Formally a set of instances.
- *Task:* Set of subjects selected by the student, containing sets of study groups.
  Formally a set of activities.
- *$A(I)$:* Activity to which instance I belongs.
- *$w(A)$:* The weight of activity A. $w(A) \in \mathbb{Z}$
- *$w(I)$:* The weight of instance I. $w(I) \in \mathbb{Z}$
- *$w(S)$:* The weight of schedule S. Described in detail later in this chapter.
- *Colliding instances:* Two instances I_1 and I_2 are colliding, if there exists time $t \in N$ inside both intervals I_1 and I_2.
  Formally: Instances I_1 and I_2 are colliding if there exists $t \in \mathbb{N}$ such that $b(I_1) \leq t < e(I_1)$ and $b(I_2) \leq t < e(I_2)$
- *Valid schedule:* Schedule S is valid, if there are no two colliding instances or belonging to the same activity in schedule S.
  Formally: Schedule S is valid if each pair of different instances $I_1 \in S$ and $I_2 \in S$, is not colliding and belong to different activities.

We will divide our problem into multiple subproblems based on the different requirements for the resulting schedule.

2.1 Non-weighted scheduling

For this subproblem, activities and instances are not weighted. In practice this simulates a situation, when a student only wants to know the maximum number of classes he can possibly fit into his schedule. Therefore, $w(S)$ represents the number of instances in schedule S.

Problem name: Non-weighted scheduling

- **Input:** $Task$
- **Output:** Valid schedule S, where $w(S) = |S|$ is maximal.

2.2 Weighted activities

Students usually have subject preferences. By the subject preferences we mean, that the student would prefer to have in his schedule one subject rather than another. This preferences can be represented by weights of corresponding activities. All activities of the task will have specified weight $w(A)$ to determine their importance. Therefore, in this problem, the weight of schedule $w(S)$ will represent a sum of the weights of the activities to which the instances in schedule S belong.

Problem name: Weighted activities

- **Input:** $Task$
- **Output:** Valid schedule S, where $w(S) = \sum\limits_{I \in S} w(A(I))$ is maximal.

2.3 Weighted instances

Most of the students also have time preferences. For example, they do not like to go to school early in the morning, or on Fridays, or they have a part time job on some specific days of the week, or they have some other personal preferences. Hence it is appropriate to use also weights based on time. For each instance I in the task, we specify weight $w(I)$. This weight determines both the importance of the activity to which instance I belongs and time suitability of instance I. In this problem, the weight of schedule $w(S)$ will represent the sum of weights of instances in schedule S.

Problem name: Weighted instances

- **Input:** $Task$
- **Output:** Valid schedule S, where $w(S) = \sum\limits_{I \in S} w(I)$ is maximal.

Note: Discussed algorithms do not handle transformation of real-life time preferences to weights of instances. They expect the instances in the task to already have specified weight. The transformation is left for the student to handle, or for additional application. Also the ratio between subject preferences and time preferences in the weight of an instance is determined ahead and already included in the weights of instances.

2.4 Other notation

We define additional notation used further in this thesis.

- *m:* Number of activities in the $Task$.
  Formally: $m = |Task|$
- *n:* Number of instances in the $Task$.
  Formally: $n = \sum\limits_{A \in Task} |A|$
- *$b(I)$:* The time when instance I begins. $b(I) \in \mathbb{N}$
- *$e(I)$:* The time when instance I ends. $e(I) \in \mathbb{N}$
- *$Last(S)$:* The highest $e(I)$ of all instances in schedule S.
  Formally: $Last(S) = \max\limits_{I \in S} e(I)$
- *Commutable schedules:* Schedules S_1 and S_2 are commutable, if instances contained by them belong to the same set of activities.
  Formally: Schedules S_1 and S_2 are commutable if $\{A(I) : I \in S_1\} = \{A(I) : I \in S_2\}$.
- *r-approximation:* We define solution x to be an r-approximation if $v(x) \geq r \cdot v(x')$, where $v(x)$ is a value of solution x and x' is optimal solution to the given problem. In this thesis we consider only maximalization problems.

2.5 Example

Let us have the following study groups offered by the university.

- *Subject 1 (weight 1):* Monday 9:00–10:30 (weight 1), Tuesday 9:00–10:30 (weight 2), Tuesday 10:45–12:15 (weight 10).

- *Subject 2 (weight 3):* Monday 10:00–11:30 (weight 5), Monday 16:00–17:30 (weight 3).
- *Subject 3 (weight 2):* Tuesday 11:00–12:30 (weight 2).

We can get the following transformation to our notation.

$$I_1 = [540, 630), I_2 = [1980, 2070), I_3 = [2085, 2175)$$
$$I_4 = [600, 690), I_5 = [960, 1050)$$
$$I_6 = [2100, 2190)$$
$$A_1 = \{I_1, I_2, I_3\}, A_2 = \{I_4, I_5\}, A_3 = \{I_6\}$$
$$Task = \{A_1, A_2, A_3\}$$
$$W(A_1) = 1, W(A_2) = 3, W(A_3) = 2$$
$$W(I_1) = 1, W(I_2) = 2, W(I_3) = 10$$
$$W(I_4) = 5, W(I_5) = 3, W(I_6) = 2$$

$$A(I_4) = A_2$$

Instances I_1 and I_4 are colliding.

$S = \{I_1, I_4, I_5\}$ is a schedule.

Schedule S is not valid, because I_1 and I_4 are colliding and I_4 and I_5 belong to the same activity.

Schedules $S_1 = \{I_1, I_5\}$ and $S_2 = \{I_2, I_4\}$ are commutable.

$$n = 6, m = 3$$

$$b(I_2) = 1980, e(I_5) = 1050$$
$$Last(S) = 1050$$

One of optimal solutions for *Non-weighted scheduling problem* is $S = \{I_1, I_5, I_6\}$, where $W(S) = 3$.

One of optimal solutions for *Weighted activities problem* is $S = \{I_2, I_4, I_6\}$, where $W(S) = 6$.

Optimal solutions for *Weighted instances problem* is $S = \{I_3, I_4\}$, where $W(S) = 15$.

Chapter **3**

Existing Algorithms

3.1 Greedy

3.1.1 Non-weighted scheduling

Our problem can be represented as undirected graph G. Vertices of graph G represent instances. Edge between vertices v_1 and v_2 is present if instances represented by v_1 and v_2 are colliding or belong to the same activity. Schedule S is a set of instances, which is equivalent to set of vertices of graph G. Schedule S is valid, if a vertex-induced subgraph G_1 of graph G, induced by vertices representing instances of S is an edgeless graph. In other words, a valid schedule is in graph G represented as independent set. This means, that to get maximum size of schedule S, we need to find maximum independent set of graph G. This is proven to be NP-hard problem as shown in [1], but there is a simple approximation algorithm called *Greedy* described in [2] with approximation factor $\frac{3}{\Delta+2}$, where Δ is maximum degree of graph G. Complexity of the algorithm is linear with the number of vertices and edges of graph G.

The algorithm begins with an empty set S of vertices, representing the schedule, and a given graph G. While the graph G is not empty, the algorithm selects a vertex v of graph G, where the degree of v is minimal among $V(G)$. Than vertex v is added to set S and altogether with his adjacent vertices removed from the graph G.

Algorithm 3.1 Greedy

```
1: function Greedy(Task)
2:     BuildGraph
3:     $S \leftarrow \emptyset$
4:     while $G$ is not empty do
5:         $v \leftarrow \arg\min_{w \in V(G)} deg(w)$
6:         $S \leftarrow S \cup \{v\}$
```

```
7:         $G \leftarrow G \setminus \{v\} \cup N(v)$
8:     return $S$
```

Building graph G First, we need to build graph G. First, we create edges representing activities. Every activity will form an independent clique in the graph G. This takes $\sum\limits_{A \in Task} \frac{|A| \cdot (|A|-1)}{2}$ time. Next, we need to create edges representing collisions between instances. This can be done naively in $\frac{n \cdot (n-1)}{2} = \mathcal{O}(n^2)$ time. Better way to do this is to sort instances ascending by $e(I)$ first and then for each instance I, iterate over instances I' backwards starting from instance previous to I, while creating edges between I and I', until I' is not colliding with I anymore.

Algorithm 3.2 BuildGraph

```
1: procedure BuildGraph
2:     Create a vertex $v_I$ for each instance $I \in Task$.
3:     for all $A \in Task$ do
4:         for all $I_1, I_2 \in A : I_1 \neq I_2$ do
5:             Create an edge between vertices $v_{I_1}$ and $v_{I_2}$.
6:     Sort instances by $e(I)$.
7:     for $i$ from 0 to $\sum_{A \in Task} |A| - 1$ do
8:         $j \leftarrow i - 1$
9:         while $j \geq 0$ and ($I_i$ and $I_j$ are colliding) do
10:            Create an edge between vertices $v_{I_i}$ and $v_{I_j}$.
11:            $j \leftarrow j - 1$
```

Proposition 3.1 *Building graph G via* BuildGraph *takes $\mathcal{O}(n \cdot \log(n) + E)$, where E is the number of edges in graph G.*

Proof Sorting instances takes $\mathcal{O}(n \cdot \log(n))$. Number of collisions can be generalized as $\mathcal{O}(n^2)$, but to be more precise we use $\mathcal{O}(E)$, where E is the number of edges in graph G. As E also includes edges representing cliques of activities, the overall process of graph creation takes $\mathcal{O}(n \cdot \log(n) + E)$. Using E instead of n^2 proves to be significantly more accurate with sparse graphs, where $E \simeq n \cdot \log(n)$ or $E < n \cdot \log(n)$. □

Required operations We claim the greedy algorithm to run in $\mathcal{O}(V + E)$, where V is number of vertices and E number of edges. Since the size of the maximum independent set can be up to V (e.g. edgeless graph), we need to find data structure over the graph, with operation of finding a vertex with minimal degree in constant time. Therefore, we will need to keep the vertices ordered by their degree. When we are removing vertices from the graph, we have to change degrees of other vertices, as we remove edges of the removing

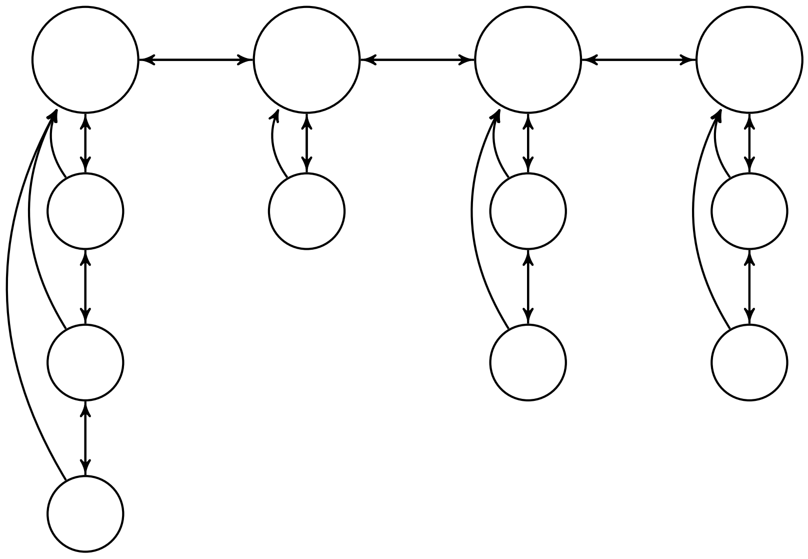
Figure 3.1: Visualization of LOL structure

vertices. We are removing E edges, so to keep the complexity $\mathcal{O}(V + E)$, we need to remove edges in constant time, including the change of degrees. (In the process of building graph G, we are adding edges, i.e. increasing degrees, so we also need to support both increment degree and decrement degree operation in constant time.) We also need a structure to keep list of neighbours for each vertex. We can use singly linked list, or if we settle for amortized $\mathcal{O}(1)$, we can use dynamically reallocated array.

Structure LOL A structure supporting required operations in $\mathcal{O}(1)$ time can be achieved with a structure LOL. The structure LOL contains a double linked list M. Each item of M contains another double linked list S. Each item of S keeps a link to an item in M containing S. In our algorithm, items in S represent vertices. Each S is a list of vertices with the same degree d. A degree of vertices in S is kept in the item of M containing S. The items of M are kept in the increasing order of the degrees. For each vertex, we keep a link to corresponding item of S, to find it in constant time.

Operations in LOL

- *Find a vertex with a minimum degree.*
  We select the first item in S of the first item in M.

- *Decrementing/incrementing a degree of vertex v.*
  We get item S' representing vertex v. Next we get a corresponding item M' in M. We check, if the item M'_n for lower/higher degree is present

in M. If not, we create it. Next, we move S' from M' to M'_n. If M' is now empty, we remove it from M.

Proposition 3.2 *Time complexity of this solution is $\mathcal{O}(n \cdot \log(n) + E) \subseteq \mathcal{O}(n^2)$ and space complexity $\mathcal{O}(n + E) \subseteq \mathcal{O}(n^2)$.*

Proof The whole algorithm consists of building graph G and finding an independent set with greedy algorithm. Building graph G takes $\mathcal{O}(n \cdot \log(n) + E)$. Greedy algorithm runs in $\mathcal{O}(V + E)$. Together we get $\mathcal{O}(n \cdot \log(n) + E + V + E) = \mathcal{O}(n \cdot \log(n) + E + n + E) = \mathcal{O}(n \cdot (\log(n) + 1) + 2 \cdot E) = \mathcal{O}(n \cdot \log(n) + E) \in \mathcal{O}(n^2)$.

We use $\mathcal{O}(n + E)$ space to keep the graph G. The number of items in M is $\mathcal{O}(n)$. The sum of items in all S is $\mathcal{O}(V) = \mathcal{O}(n)$. Together we get $\mathcal{O}(n + E + n + n) = \mathcal{O}(3 \cdot n + E) = \mathcal{O}(n + E) \in \mathcal{O}(n^2)$. $\square$

3.2 Integer programming

Another approach to our problem we can take is through linear programming. Linear programming is a method used to maximize profit in a mathematical model bounded by linear inequalities.

To represent linear problem, we can use canonical form:
maximize $c^T \cdot x$
subject to $M \cdot x \leq b$
and $x \geq 0$
where x is a result vector we want to recieve as a result, c is a profit vector representing weights of variables in vector x to the final profit, b is a vector, that together with matrix M defines restrictions of vector x as linear relations.

In our problem, variables in vector x represent presence of instances in the final schedule. As we do not want to have partial instances in our schedule, we need to use integer programming, i.e. solve the following program.
maximize $c^T \cdot x$
subject to $M \cdot x \leq b$,
$x \geq 0$
and $x \in \mathbb{Z}^n$ is integral.
In fact, because we want the instances to either be in the schedule or not, we have to use binary integer programming to solve the following program.
maximize $c^T \cdot x$
subject to $M \cdot x \leq b$,
$x \geq 0$
and $x \in \{0, 1\}^n$ is binary.

In our problem, we have to ensure the following restrictions.

- *For each activity A, a valid schedule S can contain at most one instance of activity A.*
  To satisfy this restriction, we create the following inequality for each activity A:
  $$\sum_{I \in A} x_I \leq 1$$
- *A valid schedule S can not contain colliding instances.*
  This restriction can be represented in multiple different ways. We will create an inequality for each instance J as follows:
  $$\sum_{J \in I_c} x_J \leq 1$$
  where I_c is a set of instances colliding with J.

Content of the profit vector c will be described for each problem individually.

Proposition 3.3 *Time complexity is at least $\mathcal{O}(n \cdot (n+m))$, space complexity also at least $\mathcal{O}(n \cdot (n+m))$.*

Proof We have to create matrix M according to the restrictions described. Matrix M has n columns, m rows for the first restriction and n columns for the second restriction. Hence both time and space complexity is $\mathcal{O}(n \cdot (n+m))$. Then we have to use binary integer programming to solve the problem. $\square$

3.2.1 Non-weighted scheduling

For a *non-weighted scheduling problem*, we want all instances to have the same weight. Therefore, we want to maximize $\sum\limits_I (x_I \cdot c_I)$, where all $c_I = 1$, hence maximize $\sum\limits_I x_I$.

3.2.2 Weighted activities

For the weighted activities scheduling problem, we want all instances of activity A to have weight $w(A)$. Therefore we want to maximize $\sum\limits_I (x_I \cdot c_I)$, where $c_I = w(A(I))$.

3.2.3 Weighted instances

For the weighted instances scheduling problem, we want each instance I to have weight $w(I)$. Therefore we want to maximize $\sum\limits_I (x_I \cdot c_I)$, where $c_I = w(I)$.

3.3 The Unified Algorithm

In this section, we will first introduce a solution to the *Weighted instances problem. Weighted activities problem* and *Non-weighted scheduling problem* are covered by this solution as special cases of *Weighted instances problem.* Possible simplifications of the algorithm for these problems will be discussed later in the corresponding subsections.

3.3.1 Weighted instances

To speed up the processing time of the task and get some upper bound on the complexity, we can settle for an approximation of the optimal solution. We will use an existing algorithm called *The Unified Algorithm* described in [3] with approximation factor $\frac{1}{2}$ and time comlexity $\mathcal{O}(n \cdot \log(n))$.

We will use the same representation of the problem as in the previous section 3.2 about Integer programming. The Unified Algorithm we will describe has a approximation guarantee of $\frac{1}{2}$, which means that it always produces $\frac{1}{2}$-approximation solutions, where value of solution x is $c \cdot x$.

Local Ratio Theorem Let M be a set of constraints and c, c_1 and c_2 profit vectors such that $c = c_1 + c_2$. If x is an r-approximation with respect to constraints M with profit function c_1 and with respect to constraints M with profit function c_2, the x is also an r-approximation with respect to constraints M and profit function c.

Proof Let x', x'_1 and x'_2 be optimal solutions for constraints M with profit function c, constraints M with profit function c_1 and constraints M with profit function c_2 respectively. Then we can use the following rules:

- $c \cdot x = c_1 \cdot x + c_2 \cdot x \geq r \cdot c_1 \cdot x'_1 + r \cdot c_2 \cdot x'_2$ by definition.
- $r \cdot c_1 \cdot x'_1 + r \cdot c_2 \cdot x'_2 \geq r \cdot c_1 \cdot x' + r \cdot c_2 \cdot x'$, because x'_1 and x'_2 are optimal solutions for corresponding profit functions.
- $r \cdot c_1 \cdot x' + r \cdot c_2 \cdot x' = r \cdot c \cdot x'$ by definition.

□

We use algorithm approximating the optimal solution based on the decomposition of profit function c. The algorithm can be described by the following scheme.

Algorithm 3.3 The Unified Algorithm

```
1: procedure Unified Algorithm
2:     Delete all instances with non-positive profit.
```

```
3:    If no instances remain return the empty schedule. Otherwise proceed
      to the next step.
4:    Select an instance $I'$ and decompose $c$ by $c = c_1 + c_2$. This step will
      be later described in detail.
5:    Solve the problem recursively with $c_2$ as the profit function. Let $S$ be
      the schedule returned.
6:    If $S \cup \{I'\}$ is a valid schedule, return $S \cup \{I'\}$, otherwise return $S$.
```

The instance I' we select in step 3 will be an instance with the lowest $e(I)$ of all remaining instances. Decomposition of c is determined by selecting c_1. A variable in c_1 corresponding to instance I is $w(I')$, if $A(I) = A(I')$, or if instance I is colliding with instance I'. Otherwise the variable is 0.

Proposition 3.4 *If schedule S is a $\frac{1}{2}$-approximation with respect to c_2, then schedule returned in step 5 is a $\frac{1}{2}$-approximation with respect to c_1, for a decomposition of c and selection of I' described above.*

Proof By the definiton of decomposition of c, we know that the variable in c_2 corresponding to the instance I' is 0. This means that whether we include I' into the schedule S or not, the value of profit function will not change. Therefore if the schedule S is a $\frac{1}{2}$-approximation with respect to c_2, then then schedule returned in step 5 is also a $\frac{1}{2}$-approximation with respect to c_2.

Let us determine the maximum possible value of profit function for vector c_1. Instances with a positive profit to the solution are all either mutually colliding – instances I_1, or belong to the activity $A(I')$ – instances I_2. This means, that for a valid schedule, we can select at most one instance from I_1 and at most one instance from I_2. Since all of these instances have the same value $w(I')$ in the profit vector c_1, the value of the profit function for the optimal solution can be at most $2 \cdot w(I')$.

Now we show the schedule returned in step 5 to be $\frac{1}{2}$-approximation with respect to c_1. As stated in step 5, there are only 2 possible options. If $S \cup \{I'\}$ is a valid schedule, we add I' to the solution, which makes the value of profit function $w(I')$, for profit vector c_1. If $S \cup \{I'\}$ is not valid, S must contain either an instance colliding with I', or an instance belonging to activity $A(I')$. Each of these instances increases the profit function with respect to profit vector c_1 by $w(I')$. Therefore, the value of profit function for schedule returned in step 5 with respect to vector c_1 is always at least $w(I')$, which is $\frac{1}{2}$ of the maximal possible value shown above.

□

Quality of final schedule We showed the schedule returned in step 5 to be a $\frac{1}{2}$-approximation with respect to both vector c_1 and c_2. Therefore the schedule returned is also a $\frac{1}{2}$-approximation with respect to vector c, using the Local Ratio theorem. By induction, the final schedule is also an $\frac{1}{2}$-approximation, if we prove the first step of the induction to apply. The first schedule returned is an empty schedule, returned when we have no more instances with positive profit left. If there is no positive variable in c, the value of profit function for optimal solution is 0, for positive x. Empty schedule has a value of profit function 0, so we return an optimal schedule, hence also an $\frac{1}{2}$-approximation.

Implementation The algorithm can be simply implemented by recursive procedure following the scheme. This is the naive solution resulting in $\mathcal{O}(n^2)$ complexity. We will describe and use a better implementation with complexity $\mathcal{O}(n \cdot \log(n))$.

First, we will define parameters dW, $dA(A)$ for each activity A and $dI(I)$ for each instance I. Parameter dW and every dA are initially set to 0. Next will we define an endpoint. Each instance has a starting endpoint $2b(I) + 1$ and ending endpoint $2e(I)$. Because $b(I)$ and $e(I)$ are integers, we use the $+1$ to sort starting endpoints after the ending endpoints for same $b(I)$ and $e(I)$.

We use procedure described by the following pseudocode:

Algorithm 3.4 FastUnifiedAlgorithm

```
function FastUnifiedAlgorithm($Task$)
    Sort endpoints in ascending order.
    for all $p \in$ endpoints do
        $I$ is an instance to which endpoint $p$ belongs.
        if $p$ is a starting endpoint then
            $dI(I) \leftarrow dW - dA(A(I))$
        else
            $W \leftarrow w(I) - dW + dI(I)$
            if $W \leq 0$ then
                delete $I$
            else
                $dA(A(I)) \leftarrow dA(A(I)) + W$
                $dW \leftarrow dW + W$
    $R \leftarrow \emptyset$
    for all $p \in$ endpoints in reverse order do
        $I$ is an instance to which endpoint $p$ belongs.
        if $p$ is an ending endpoint then
            if $R \cup \{I\}$ is a valid schedule then
                $R \leftarrow R \cup \{I\}$
    return $R$
```

Now we will show the procedure *FastUnifiedAlgorithm* to follow the steps of the Unified Algorithm.

- Step 1 is not necessary to be executed immediately. It's sufficient, that we determine the profit and eventually delete the instance when we process it.

- Step 2 is followed by creating an empty schedule after first iterating over the instances.

- We show how a current profit W of an instance I is calculated when processing instance I. Since W represents the current profit of the instance in profit vector c, it allows us to follow the steps 3 and 4.
  For every ending endpoint p in the iteration, we count the current profit of instance I, to which p belongs, as $W = w(I) - dW + dI(I) = w(I) - dW + dW' - dA(A(I))' = w(I) - (dW - dW') - dA(A(I))'$. Let us explain the formula. The initial weight $w(I)$ of instance I needs to be decremented by profits of the previous non-deleted instances I_x colliding with I or belonging to activity $A(I)$. Parameter dW is incremented by W of each visited ending endpoint with a positive profit. Parameter dW' is a value of parameter dW at time of visiting starting endpoint of I. Therefore $dW - dW'$ is a sum of current profits of all instances I_x, colliding with I. Next, we need to substract the profit of instances I_x belonging to activity $A(I)$. As we may have already substracted some of them in the previous step, we only select those, whose ending endpoints are lower than $b(I)$. This value is already held as $dA(A(I))'$, which is $dA(A(I))$ at time $b(I)$.

- Step 5 is simulated by final traversing all ending endpoints in reverse order and adding non-deleted instances to the schedule, if they keep the schedule valid.

Proposition 3.5 *Time complexity of the algorithm is $\mathcal{O}(n \cdot \log(n))$ and space comlexity is $\mathcal{O}(n)$.*

Proof Initialization of parameters dA takes $\mathcal{O}(m)$. Sorting endpoints takes $\mathcal{O}(n \cdot \log(2 \cdot n))$. Every step in the first iteration over endpoints takes $\mathcal{O}(1)$, which makes $\mathcal{O}(2 \cdot n)$ for all iterations. Every step in the second iteration also takes $\mathcal{O}(1)$, because to test validity of the schedule, we only need to test collision between I and last instance added to the schedule. This is because $e(I)$ is the lowest of all I_1 in S, so we only need only to compare $e(I)$ with the lowest $b(I)$ in the schedule. So we get again $\mathcal{O}(2 \cdot n)$. Altogether we have $\mathcal{O}(m + n \cdot \log(2 \cdot n) + 2 \cdot 2 \cdot n) = \mathcal{O}(m + n \cdot \log(n))$. Because we do not allow empty activities, $m \leq n$, so we get $\mathcal{O}(n + n \cdot \log(n)) = \mathcal{O}(n \cdot \log(n))$.

We need to keep in memory all instances, endpoints, and parameters such as dI. Space complexity can be calculated as is $\mathcal{O}(n + 2 \cdot n + n) = \mathcal{O}(n)$. $\square$

3.3.2 Our contribution

Selecting I' As we have shown in Proof 3.3.1, to achieve $\frac{1}{2}$-approximation factor, we need to guarantee, that at most one instance from instances colliding with I' can be present in a valid schedule. This property is satistied, if instances colliding with I' are all mutually colliding.

Therefore, we can select any instance I' such that instances colliding with I' form a clique in a graph, where vertices represent instances and edges represent collisions.

Improving approximation factor We will show how to improve approximation factor to $\frac{1}{2} + \varepsilon$, where $\varepsilon > 0$.

When creating a schedule in final reverse iteration over endpoints, we will execute a procedure *FindOptimal* as soon as we find first non-deleted endpoint p.

Procedure *FindOptimal* will try to find an optimal solution for the endpoints starting from p. This is done by reverting updates done to dA and dW form endpoint p. We are now searching for a non-colliding pair of instances I_1, I_2, where $I_1 = I_2$, I_1 is colliding with I, $A(I_1) \neq A(I)$, $A(I_2) = A(I)$ and $w(I_1)+w(I_2) > w(I)$. If such pair exists, then we add it to the schedule RET. Otherwise, we add instance I to schedule RET.

Clearly, $b(I_1) \leq e(I_2)$, therefore procedure *FindOptimal* runs in linear time with n.

Algorithm 3.5 FindOptimal

```
procedure FindOptimal($p$)
    $I$ is an instance to which endpoint $p$ belongs.
    $W \leftarrow w(I) - dW + dI(I)$
    $dA(A(I)) \leftarrow dA(A(I)) - W$
    $dW \leftarrow dW - W$
    $R \leftarrow \{I\}$
    $I_1 \leftarrow NULL$
    $W_1 \leftarrow 0$
    for all $p_x \in$ endpoints starting from $p$ do
        $I_x$ is an instance to which endpoint $p_x$ belongs.
        if $p$ is a starting endpoint then
            if $A(I_x) = A(I)$ then
                $W \leftarrow w(I_x) - dA(A(I_x))$
                if $W + W_1 > w(R)$ then
                    $R \leftarrow \{I_1, I_x\}$
            else
                $dI(I_x) \leftarrow dW - dA(A(I_x))$
        else
```

```
19:             if $A(I_x) \neq A(I)$ then
20:                 $W \leftarrow w(I_x) - dW + dI(I_x)$
21:                 if $W > W_1$ then
22:                     $I_1 \leftarrow I_x$
23:                     $W_1 \leftarrow W$
```

Unified algorithm will be updated in the following way.

Algorithm 3.6 FastUnifiedAlgorithm

```
1: function FastUnifiedAlgorithm($Task$)
2:     Sort endpoints in ascending order.
3:     for all $p \in$ endpoints do
4:         $I$ is an instance to which endpoint $p$ belongs.
5:         if $p$ is a starting endpoint then
6:             $dI(I) \leftarrow dW - dA(A(I))$
7:         else
8:             $W \leftarrow w(I) - dW + dI(I)$
9:             if $W \leq 0$ then
10:                delete $I$
11:            else
12:                $dA(A(I)) \leftarrow dA(A(I)) + W$
13:                $dW \leftarrow dW + W$
14:    $R \leftarrow \emptyset$
15:    for all $p \in$ endpoints in reverse order do
16:        $I$ is an instance to which endpoint $p$ belongs.
17:        if $p$ is an ending endpoint then
18:            if $R = \emptyset$ then FindOptimal($p$)
19:            else
20:                if $R \cup \{I\}$ is a valid schedule then
21:                    $R \leftarrow R \cup \{I\}$
22:    return $R$
```

The first non-empty schedule returned will be optimal, so we can improve the theorem.

Sharp Local Ratio Theorem Let M be a set of constraints and c, c_1 and c_2 profit vectors such that $c = c_1 + c_2$. If x is an r-approximation with respect to constraints M with profit function c_1 and r'-approximation with respect to constraints M with profit function c_2, the x is an r'-approximation with respect to constraints M and profit function c, where $r' > r$.

Proof Let x', x'_1 and x'_2 be optimal solutions for constraints M with profit function c, constraints M with profit function c_1 and constraints M with profit function c_2 respectively. Then we can use the following rules:

- $c \cdot x = c_1 \cdot x + c_2 \cdot x \geq r \cdot c_1 \cdot x_1' + r' \cdot c_2 \cdot x_2'$ by definition.
- $r \cdot c_1 \cdot x_1' + r' \cdot c_2 \cdot x_2' > r \cdot c_1 \cdot x_1' + r \cdot c_2 \cdot x_2'$ as stated in proposition.
- $r \cdot c_1 \cdot x_1' + r \cdot c_2 \cdot x_2' \geq r \cdot c_1 \cdot x' + r \cdot c_2 \cdot x'$, because x_1' and x_2' are optimal solutions for corresponding profit functions.
- $r \cdot c_1 \cdot x' + r \cdot c_2 \cdot x' = r \cdot c \cdot x'$ by definition.

We proved $c \cdot x > r \cdot c \cdot x'$, which proves the proposition. □

Weighted activities problem and *Non-weighted scheduling problem* can be solved by simulating *Weighted instances problem* by setting $w(I)$ to $w(A(I))$ or 1 respectively.

Chapter **4**

Self-designed algorithms

4.1 Interval Search

Computational models The algorithm described in this section uses exponential memory, which means that we should use different computational model, as described in chapter 1. We will establish complexity for both *unit-cost* and *logarithmic-cost* models. Because the size of the input for the problems is $\mathcal{O}(n)$, we penalize operations with numbers larger than polynomial to n.

This algorithm is based on creating a set P consisting of valid schedules. New schedules are created by successive adding instances to the schedules already present in set P. In the end the best schedule in P is returned.

For this algorithm, we define two new terms.

- P*:* Set of valid schedules.
- J*:* Set of all instances in the task.
  Formally: $J = \{I : I \in A, A \in Task\}$

4.1.1 Non-weighted scheduling

The algorithm begins with an empty schedule in P. Next, we select instances I from J one by one and try to create new valid schedules by adding I to valid schedules already in P.

Algorithm 4.1 IntervalSearch

```
1: function IntervalSearch($Task$)
2:     $P \leftarrow \{\emptyset\}$
3:     for all $I \in J$ do
4:         for all $S \in P$ do
```

```
 5:            if $S \cup \{I\}$ is a valid schedule then
 6:                $P \leftarrow P \cup \{S \cup \{I\}\}$
 7:    $R \leftarrow \emptyset$
 8:    for all $S \in P$ do
 9:        if $|S| > |R|$ then
10:            $R \leftarrow S$
11:    return $R$
```

Proposition 4.1 *Time complexity of the algorithm 4.1 is $\mathcal{O}(n^2 \cdot 2^n)$ for* unit-cost *model and $\mathcal{O}(\frac{n^3 \cdot 2^n}{\log(n)})$ for* logarithmic-cost *model.*

Proof

- *unit-cost* The size of set J is $\mathcal{O}(n)$. The size of set P is $\mathcal{O}(2^n)$, corresponding to the size of powerset of J. Testing validity of schedule $S_1 = S \cup \{I\}$ can be done in time $\mathcal{O}(|S_1|^2)$. Since we already know the schedule S to be valid, we only need to test collisions between the instance I and the instances in the schedule S. Testing collision between two instances takes a constant time, so testing validity of S_1 is linear with the size of set S. The size of the schedule S is $\mathcal{O}(n)$. Finding the largest schedule in P takes $\mathcal{O}(|P|) = \mathcal{O}(2^n)$, if we can get the size of the schedule in constant time. Hence together we get $\mathcal{O}(n \cdot 2^n \cdot n + 2^n) = \mathcal{O}(n^2 \cdot 2^n + 2^n) = \mathcal{O}((n^2+1) \cdot 2^n) = \mathcal{O}(n^2 \cdot 2^n)$.

- *logarithmic-cost* Accessing instance I in schedules S in set P takes $\frac{\log(|P|)}{\log(n)} = \frac{\log(2^n)}{\log(n)} = \frac{n}{\log(n)}$. Therefore we get the total complexity of $\mathcal{O}(\frac{n^3 \cdot 2^n}{\log(n)})$.

□

We will try to improve the complexity (for both models). Until now, the validity test for $S \cup \{I\}$ at line 5 of procedure *IntervalSearch* took $\mathcal{O}(|S|) = \mathcal{O}(n)$ time. We will use a little trick to make the validation run in constant time. Before we start iterating over instances of J, we sort them ascending by $e(I)$. Then we iterate over them in this order. When we are adding an instance I to the schedule S, we only need to test collision of I with the last instance added into the schedule S, as shown in the following proposition.

Proposition 4.2 *We are adding the instance I to the schedule $S = \{I_1, I_2, ..., I_k\}$, where the instances of schedule S are ordered in the order they were inserted. If there exists an instance $I_j \in S$ colliding with I, then I_k is also colliding with I.*

Proof I_{k+1} is an instance we want to add into the schedule S. I_k is the last instance added into the schedule S. I_j is an arbitrary instance of schedule S. By definition the following applies.

$$b(I_{k+1}) < e(I_{k+1})$$

$$b(I_k) < e(I_k)$$

$$b(I_j) < e(I_j)$$

Because the instances are sorted by $e(I)$, the following must apply.

$$e(I_j) \leq e(I_k) \leq e(I_{k+1})$$

Because we claim that instances I_{k+1} and I_j are colliding, the following applies. There exists t such that $b(I) \leq t < e(I_{k+1})$ and $b(I_j) \leq t < e(I_j)$. We need to prove the following. There exists t_2 such that $b(I_{k+1}) \leq t_2 < e(I_{k+1})$ and $b(I_k) \leq t_2 < e(I_k)$

We choose t_2 such that $t_2 \geq t$ and $b(I_k) \leq t_2 < e(I_k)$. Such t_2 always exists. If $b(I_k) \leq t < e(I_k)$, we can choose $t_2 = t$. Otherwise, $t < b(I_{k+1})$, because by definition $e(I_k) \geq e(I_j)$. In this case we can choose $t_2 = b(I_k)$.

Because $t_2 < e(I_k)$ and $e(I_k) \leq e(I_{k+1})$, then $t_2 < e(I_{k+1})$. Because $t \geq b(I_{k+1})$ and $t_2 \geq t$, then $t_2 \geq b(I_{k+1})$. □

Now we only need to keep parameter $Last(S)$ for each schedule, because $Last(S) \leq e(I)$ and $S \cup \{I\}$ is not valid only if $Last(S) > b(I)$.

Proposition 4.3 *With faster validity test described above, time complexity is $\mathcal{O}(n \cdot 2^n)$ for* unit-cost *model and $\mathcal{O}(\frac{n^2 \cdot 2^n}{\log(n)})$ for* logarithmic-cost *model.*

Proof

- *unit-cost* Using the knowledge from proposition 4.2, we can test the validity of $S \cup \{I\}$ in constant time, hence with the initial sorting the complexity is $\mathcal{O}(n \cdot \log(n) + n \cdot 2^n + 2^n) = \mathcal{O}(n \cdot \log(n) + (n+1) \cdot 2^n) = \mathcal{O}(n \cdot \log(n) + n \cdot 2^n) = \mathcal{O}(n \cdot (\log(n) + 2^n)) = \mathcal{O}(n \cdot 2^n)$.
- *logarithmic-cost* Accessing instance I in schedules S in set P takes $\frac{n}{\log(n)}$. Therefore we get total complexity of $\mathcal{O}(\frac{n^2 \cdot 2^n}{\log(n)})$.

□

Definition Let S be a schedule. Then the *potential* $\Phi(S)$ of schedule S is defined as $\Phi(S) = -Last(S)$.

Potential $\Phi(S)$ of schedule S can be interpreted as an ability of schedule S to generate new schedules by adding new instances. As we have already proved above, the validity of new schedule $S \cup \{I\}$ for unknown I is dependent only on $Last(S)$. Therefore we claim that schedules with lower $Last(S)$ have a better chance of generating new schedules.

We will try to improve the size of set P. By definition, a valid schedule can not contain multiple instances of the same activity. Let us have commutable schedules S_1 and S_2. We will not consider S_1 and S_2 as unique for set P. Because we are solving a non-weighted problem, weights of S_1 and S_2 are the same. We have to decide, if we want to keep S_1 or S_2 in P. We keep the schedule with better potential. When creating a new schedule S, which already has a commutable schedule in P, we can drop S. This means, that size of P can not be larger than a powerset of activities in the task.

Proposition 4.4 *After the update of the set P, the time complexity of the algorithm is $\mathcal{O}(n \cdot 2^m)$ for* unit-cost *model and $\mathcal{O}(\frac{m \cdot n \cdot 2^m}{\log(n)})$ for the* logarithmic-cost *model.*

Proof

- *unit-cost* Complexity is calculated as $\mathcal{O}(n \cdot \log(n) + n \cdot 2^m + 2^m) = \mathcal{O}(n \cdot \log(n) + (n+1) \cdot 2^m) = \mathcal{O}(n \cdot \log(n) + n \cdot 2^m) = \mathcal{O}(n \cdot (\log(n) + 2^m)) = \mathcal{O}(n \cdot 2^m)$.
- *logarithmic-cost* Accessing instance I in schedules S in set P takes $\frac{\log(|P|)}{\log(n)} = \frac{\log(2^m)}{\log(n)} = \frac{m}{\log(n)}$. Therefore we get the total complexity of $\mathcal{O}(\frac{m \cdot n \cdot 2^m}{\log(n)})$.

□

Proposition 4.5 *Time complexity of the* Interval Search *algorithm is $\mathcal{O}(n \cdot \log(n) + (n-m) \cdot 2^m)$ for a* unit-cost *model and $\mathcal{O}(n \cdot \log(n) + \frac{m \cdot (n-m) \cdot 2^m}{\log(n)})$ for a* logarithmic-cost *model.*

Proof We will examine the growth of the size of the set P. The initial size of P is 1. With each instance we are adding to the schedules, the size of set P can double, because we can create one new schedule from each schedule already in the set P. Therefore, the size of set P can not grow faster than geometric progression with ratio 2. Set P can grow to a maximum possible size sooner at first after m iterations, because for geometric progression the n-th item is equal to $a_1 \cdot r^n = 1 \cdot 2^n$, $2^n = 2^m$. Lets count the number of validations x in the first m iterations in J using the formula for the sum of the first n items of geometric progression.

$$x = a_1 \cdot \frac{1 - r^n}{1 - r} = 1 \cdot \frac{1 - 2^m}{1 - 2} = 2^m - 1$$

- *unit-cost* Complexity is calculated as $\mathcal{O}(n \cdot \log(n) + 2^m + (n-m) \cdot 2^m + 2^m) = \mathcal{O}(n \cdot \log(n) + (n-m+2) \cdot 2^m) = \mathcal{O}(n \cdot \log(n) + (n-m) \cdot 2^m)$.
- *logarithmic-cost* Accessing instance I in schedules S in set P takes $\frac{\log(|P|)}{\log(n)} = \frac{\log(2^m)}{\log(n)} = \frac{m}{\log(n)}$. Therefore we get total complexity of $\mathcal{O}(n \cdot \log(n) + \frac{m \cdot (n-m) \cdot 2^m}{\log(n)})$.

$\square$

Algorithm 4.2 IntervalSearch

```
function IntervalSearch(Task)
    Sort $J$ ascending by $e(I)$
    $P \leftarrow \{\emptyset\}$
    for all $I \in J$ do
        for all $S \in P$ do
            if $S \cup \{I\}$ is a valid schedule then
                if There is no schedule commutable with $S \cup \{I\}$ in $P$ then
                    $P \leftarrow P \cup \{S \cup \{I\}\}$
    $R \leftarrow \emptyset$
    for all $S \in P$ do
        if $|S| > |R|$ then
            $R \leftarrow S$
    return $R$
```

We can not improve the asymptote of the time complexity anymore, but we can still employ several optimalizations.

Observation 4.6 *If we store the set P as a collection, where items are ordered in ordered they were inserted, then the set P is ordered ascending by $Last(S)$.*

Proof We will always insert a new schedule S to the end of the ordered set P. Because J is ordered by $e(I)$ and S contains last visited instance I in set J, then $Last(S) = e(I)$. Therefore there is no schedule S_1 in P such that $Last(S_1) > Last(S)$. $\square$

When creating a new schedule $S_1 = S \cup \{I\}$, we need $Last(S) \leq b(I)$ for S_1 to be valid, as we have shown before. Since P is an ordered set ordered by $Last(S)$, we can iterate over P and when $Last(S) > b(I)$, stop the iteration and save testing validity for the rest of the set P, because we already know the schedule $S \cup \{I\}$, created from the rest of schedules in P, will not be valid.

Note: If we use binary search to find the number of iterations in advance, we can even get rid of all validation tests in exchange for $\mathcal{O}(\log(|P|))$ tests in binary search. Binary search is especially efective when $|P| \gg \log(|P|)$.

Algorithm 4.3 OptimizedIntervalSearch

```
1:  function OptimizedIntervalSearch(Task)
2:      Sort $J$ ascending by $e(I)$
3:      $P \leftarrow \{\emptyset\}$
4:      for all $I \in J$ do
5:          $i \leftarrow 0$
6:          while $i < |P|$ and $Last(P_i) \leq b(I)$ do
7:              if There is no schedule commutable with $P_i \cup \{I\}$ in $P$ then
8:                  $P \leftarrow P \cup \{P_i \cup \{I\}\}$
9:              $i \leftarrow i + 1$
10:     $R \leftarrow \emptyset$
11:     for all $S \in P$ do
12:         if $|S| > |R|$ then
13:             $R \leftarrow S$
14:     return $R$
```

Proposition 4.7 *Space complexity of the algorithm 4.3 is $\mathcal{O}(n + 2^m)$.*

Proof We need $\mathcal{O}(n)$ to keep the list of instances and $\mathcal{O}(m \cdot 2^m)$ for set P, which has a size up to 2^m and contains schedules of the size up to m. Altogether we need $\mathcal{O}(n + m \cdot 2^m)$.

This can be improved by keeping only a limited amount of information for each schedule in set P. Since schedules are never removed from P, we can keep only the last instance I added to the schedule S and a link to schedule S_p, from which the schedule S was created by adding instance I. A complete list of instances in the schedule can be obtained by recursive calls to S_p while collecting last instance I of each schedule. This reduces the space complexity to $\mathcal{O}(n + 2^m)$. □

4.1.2 Weighted activities

For the weighted activities problem, we will use the same procedure of creating set P as for the *Non-weighted scheduling problem.* To be able to do that, we have to show that commutable schedules in this problem still have the same weight.

Lemma 4.8 *Let S_1 and S_2 be commutable schedules. Then $w(S_1) = w(S_2)$.*

Proof Now the $w(S_1) = \sum_{I \in S_1} w(A(I))$ and $w(S_2) = \sum_{I \in S_2} w(A(I))$. Because both S_1 and S_2 consist of the same activities, then $w(S_1) = w(S_2)$. □

We will now have to change the final procedure of selecting the best schedule from set P. For a *Non-weighted scheduling problem*, we only needed the size of each schedule, but in the *Weighted activities problem*, we need to get the sum of the weights of the activities to which instances in schedule S belong, as stated in the definition of the problem.

Algorithm 4.4 FindBest

```
procedure FindBest
    $R \leftarrow \emptyset$
    $bestW \leftarrow 0$
    for all $S \in P$ do
        $W \leftarrow 0$
        for all $I \in S$ do
            $W \leftarrow W + w(A(I))$
        if $W >$ bestW then
            $R \leftarrow S$
            $bestW \leftarrow W$
```

Proposition 4.9 *Time complexity of* Interval Search *using the procedure described in algorithm 4.4 is* $\mathcal{O}(n \cdot \log(n) + n \cdot 2^m)$ *for* unit-cost *model and* $\mathcal{O}(n \cdot \log(n) + \frac{m \cdot n \cdot 2^m}{\log(n)})$ *for* logarithmic-cost *model.*

Proof

- *unit-cost* We will use time complexity from the previous section and update only the complexity of procedure *FindBest*. $\mathcal{O}(n \cdot \log(n) + (n-m) \cdot 2^m - 2^m + 2^m \cdot m) = \mathcal{O}(n \cdot \log(n) + (n-m-1+m) \cdot 2^m) = \mathcal{O}(n \cdot \log(n) + n \cdot 2^m)$.
- *logarithmic-cost* Accessing instance I in schedules S in set P takes $\frac{\log(|P|)}{\log(n)} = \frac{\log(2^m)}{\log(n)} = \frac{m}{\log(n)}$. Therefore we get the total complexity of $\mathcal{O}(n \cdot \log(n) + \frac{m \cdot n \cdot 2^m}{\log(n)})$.

□

We have made the complexity a bit worse, but we will try to improve it again. When creating schedules, we can store the $w(S)$ for each schedule S. We use the following equation to keep the weight of new schedules up-to-date. $w(S \cup I)) = w(S) + w(I)$. This allows us to get the weight of schedule in constant time and simplify the procedure *FindBest*.

Algorithm 4.5 FindBest

```
procedure FindBest
    $R \leftarrow \emptyset$
```

```
3:     for all $S \in P$ do
4:         if $w(S) > w(R)$ then
5:             $R \leftarrow S$
```

Proposition 4.10 *Using the procedure described in algorithm 4.5 time complexity of* Interval Search *is* $\mathcal{O}(n \cdot \log(n) + (n-m) \cdot 2^m)$ *for* unit-cost *model and* $\mathcal{O}(n \cdot \log(n) + \frac{m \cdot (n-m) \cdot 2^m}{\log(n)})$ *for* logarithmic-cost *model. Space complexity is* $\mathcal{O}(n + 2^m)$*. Predictibility is good with expected number of operations* $\mathcal{O}(n \cdot \log(n) + (n-m+2) \cdot 2^m)$*.*

Proof

- *unit-cost* Described improvement reduces the complexity of procedure *FindBest* back to $\mathcal{O}(2^m)$, which makes the complexity of the whole algorithm $\mathcal{O}(n \cdot \log(n) + (n-m) \cdot 2^m)$ again.
- *logarithmic-cost* Accessing instance I in schedules S in set P takes $\frac{\log(|P|)}{\log(n)} = \frac{\log(2^m)}{\log(n)} = \frac{m}{\log(n)}$. Therefore we get the total complexity of $\mathcal{O}(n \cdot \log(n) + \frac{m \cdot (n-m) \cdot 2^m}{\log(n)})$.
- Space complexity is the same as in the *Non-weighted scheduling problem*, hence $\mathcal{O}(n + 2^m)$.

□

4.1.3 Weighted instances

In *Weighted instances problem* we can no longer use commutable schedules.

Observation 4.11 *Let S_1 and S_2 be commutable schedules. We claim schedules S_1 and S_2 do not necessarily have to have the same weight.*

Proof Let $w(I)$ of all instances in S_1 be 1 and $w(I)$ of all instances in S_2 is 2. Clearly, $w(S_1) \neq w(S_2)$, therefore we have to consider schedules S_1 and S_2 as unique. □

Without taking an advantage of commutable schedules, we can return to using the set P of the size $\mathcal{O}(2^n)$, which would result in complexity $\mathcal{O}(2^n)$. The linear part $(n-m)$ will disappear, because the set P will grow to full size after n iterations, which would make $(n-n+1)$. Then $\mathcal{O}(n \cdot \log(n) + 2^n = \mathcal{O}(2^n)$.

If we want to keep the exponential part just 2^m, we need to come up with a better solution. We want to keep only one schedule for each set of commutable schedules, so we need to determine which one from commutable schedules to keep.

The first option is to keep schedule with the highest potential $\Phi(S)$, but this can cause wrong results.

Example If we add new schedule S_1 to set P, when we already have schedule S_2 commutable with S_1 in set P, we cannot keep only schedule with higher $w(S)$. Let us have the following task.

$$Task = \{\{I_1, I_2\}, \{I_3\}\}$$

$$e(I_1) < b(I_3) < e(I_2) < e(I_3)$$

$$w(I_1) < w(I_2) < w(I_3)$$

$$J = \{I_1, I_2, I_3\}$$

At the beginning of iteration over I_2, we have the following P.

$$P = \{\emptyset, \{I_1\}\}$$

Because $w(I_1) < w(I_2)$, at the end of the iteration we have the following P.

$$P = \{\emptyset, \{I_2\}\}$$

After all iterations, we have the following P.

$$P = \{\emptyset, \{I_2\}, \{I_3\}\}$$

The best schedule is $\{I_3\}$, but clearly there is a better schedule $\{I_1, I_3\}$.

The second option is to keep the schedule with the highest potential, i.e. the lowest $Last(S)$. But this would also lead to wrong results.

Example If we add new schedule S_1 to set P, when we already have schedule S_2 commutable with S_1 in set P, we cannot keep only schedule with higher $\Phi(S)$. Let us have the following task.

$$Task = \{\{I_1, I_2\}, \{I_3\}\}$$

$$e(I_1) < e(I_2) < b(I_3) < e(I_3)$$

$$w(I_1) < w(I_2) < w(I_3)$$

$$J = \{I_1, I_2, I_3\}$$

At the beginning of iteration over I_2, we have the following P.

$$P = \{\emptyset, \{I_1\}\}$$

Because $e(I_1) < e(I_2)$, at the end of the iteration we have the following P.

$$P = \{\emptyset, \{I_1\}\}$$

After all iterations, we have the following P.

$$P = \{\emptyset, \{I_1\}, \{I_3\}, \{I_1, I_3\}\}$$

The best schedule is $\{I_1, I_3\}$, but clearly there is a better schedule $\{I_2, I_3\}$.

To solve this problem, we will have to make a compromise between the two presented options. We will keep multiple commutable schedules. Let us have a schedule S_1 and a schedule S_2, where $w(S_1) > w(S_2)$. We will keep S_1, but if $\Phi(S_2) > \Phi(S_1)$, then we keep also S_2.

Definition We keep a schedule S, if any of the following rules is satisfied.

- There is no schedule S' commutable with S in P, such that $w(S') \geq w(S)$.
- For every schedule S' in P, commutable with S, where $w(S') \geq w(S)$, $Last(S') > Last(S)$.

At this point we define a new set and slightly redefine the set P.

- *P':* Set P' is an ordered set of schedules, where all schedules are commutable. Schedules are ordered ascending by $Last(S)$ and $w(T)$.
  Formally: For each pair of schedules P'_i and P'_j, where P'_i is i-th schedule in P' and $i < j$ the following inequations apply. $Last(P'_i) < Last(P'_j)$ and $w(P'_i) < w(P'_j)$.
- *P:* Set P is a set of P'.

Let us determine the size of set P'. The number of schedules in set P' is always at most n, because each schedule in P' has to have a unique $Last(S)$ and there is only n unique $e(I)$ in the task.

The size of the set P is again $\mathcal{O}(2^m)$, as we keep subsets P' for commutable schedules.

The algorithm now works in the following way. For every instance I in J, we iterate over P. For each P', we select a suitable schedule S, and try to add $S \cup \{I\}$ to appropriate P'_2.

Algorithm 4.6 IntervalSearch

```
1: function IntervalSearch($Task$)
2:     Sort $J$ ascending by $e(I)$
3:     $P \leftarrow \{\{\emptyset\}\}$
4:     for all $I \in J$ do
5:         for all $P' \in P$ do
6:             $S \leftarrow$ FindSchedule($P'$,$I$)
7:             if $S \neq NULL$ then
8:                 $S' \leftarrow S \cup \{I\}$
9:                 AddSchedule($S'$)
10:    $R \leftarrow \emptyset$
11:    for all $P' \in P$ do
```

```
12:         $S \leftarrow$ Last schedule in $P'$
13:         if $w(S) > w(R)$ then
14:             $R \leftarrow S$
15:     return $R$
```

Let us describe the function *FindSchedule*. This function selects the schedule from P' that will be used to create new schedule S'. As S' will have $\mathit{Last}(S') = e(I)$, we want to maximize $w(S')$. Therefore, we select the schedule S with maximum $w(S)$, such that $\mathit{Last}(S) \leq b(I)$. This can be done with binary search in time $\mathcal{O}(\log(|S|)) = \mathcal{O}(\log(n))$.

Procedure *AddSchedule* adds new schedule S' to the appropriate P'. We select P' in $\mathcal{O}(1)$. Since $\mathit{Last}(S')$ is definitely highest of all schedules in P', schedule S' will be added only if $w(S')$ is also larger than weights of all schedules in P'. This can be verified in $\mathcal{O}(1)$ by comparing with the last schedule in P'.

Proposition 4.12 *Time complexity of the algorithm 4.6 is* $\mathcal{O}(n \cdot \log(n) + \log(n) \cdot (n-m) \cdot 2^m)$ *for a* unit-cost *model and* $\mathcal{O}(n \cdot \log(n) + m \cdot (n-m) \cdot 2^m)$ *for a* logarithmic-cost *model. Space complexity is* $\mathcal{O}(n \cdot 2^m)$.

Proof

- *unit-cost* We used the assumption about growth of set P to get the following time complexity. $\mathcal{O}(n \cdot \log(n) + \log(n) \cdot (n-m) \cdot 2^m)$.
- *logarithmic-cost* Accessing instance I in schedules S in set P takes $\frac{\log(|P|)}{\log(n)} = \frac{\log(2^m)}{\log(n)} = \frac{m}{\log(n)}$. Therefore we get the total complexity of $\mathcal{O}(n \cdot \log(n) + \frac{m \cdot \log(n) \cdot (n-m) \cdot 2^m}{\log(n)}) = \mathcal{O}(n \cdot \log(n) + m \cdot (n-m) \cdot 2^m)$.
- To save memory we can keep only one instance for each schedule again and keep the space usage for a single schedule $\mathcal{O}(1)$. But since the size of each set P' is $\mathcal{O}(n)$ and the size of set P still $\mathcal{O}(2^m)$, we get the total space complexity $\mathcal{O}(n \cdot 2^m)$.

□

4.2 Brute-Force with Heuristics

The most intuitive approach is a brute-force algorithm. Using a simple recursive function is easy to implement and offer a variety of modifications. Using different heuristics we can approach specific problems with slightly modified problem statements effectively and apply all kinds of restrictions. The worst case time complexity is typically very high, but heuristics can often significantly reduce the execution time in real-life.

Definition In this algorithm, we will consider *Task* to be a collection of activities, where $Task_d$ corresponds to d-th activity in the *Task*.

4.2.1 Non-weighted scheduling

Basic algorithm for this problem is very simple. We use a recursive function, where each level of depth corresponds to a single activity of the task. In the function we try all instances of the activity and for each one we call the recursion. Finally, we call the recursion without using an instance of the activity. We keep the best schedule and at the deepest level of recursion we validate the current schedule and eventually update the best solution.

Algorithm 4.7 Recursion

```
function Recursion($d$)
    if $d \geq |Task|$ then
        if $S$ is valid then
            if $|S| > |Best|$ then
                $Best \leftarrow S$
    else
        for all $I \in Task_d$ do
            $S \leftarrow S \cup \{I\}$
            Recursion($d + 1$)
            $S \leftarrow S \setminus \{I\}$
        Recursion($d + 1$)
```

Algorithm calls the recursion in the following way.

Algorithm 4.8 Brute-Force

```
function Brute-Force($Task$)
    $S \leftarrow \emptyset$
    $Best \leftarrow \emptyset$
    Recursion(0)
    return $Best$
```

Proposition 4.13 *Time complexity of the algorithm 4.8 is* $\mathcal{O}(m^2 \cdot (\frac{n}{m} + 1)^m)$.

Proof Validation of schedule S runs in $\mathcal{O}(m^2)$ and number of recursive calls leading to this validation is $\prod_{A \in Task} (|A| + 1)$, which is $\mathcal{O}((\frac{n}{m} + 1)^m)$. Therefore the total time complexity is $\mathcal{O}(m^2 \cdot (\frac{n}{m} + 1)^m)$. □

We will implement two basic heuristics.

- Validity of schedule S will be done before each recursive call, which will remove non-valid branches of the search space.
- Branches already leading to at most the same quality schedule as current best will be skipped.

The second heuristic is ensured by keeping number of activities, for which no instance was added to the schedule.

Algorithm 4.9 Recursion

```
1: function Recursion($d$)
2:     if $d \geq |Task|$ then
3:         if $|S| > |Best|$ then
4:             $Best \leftarrow S$
5:     else
6:         for all $I \in Task_d$ do
7:             if $S \leftarrow S \cup \{I\}$ is valid then
8:                 $S \leftarrow S \cup \{I\}$
9:                 Recursion($d + 1$)
10:                $S \leftarrow S \setminus \{I\}$
11:        $Miss \leftarrow Miss + 1$
12:        if $Miss < m - |Best|$ then
13:            Recursion($d + 1$)
14:        $Miss \leftarrow Miss - 1$
```

Algorithm 4.10 Brute-Force

```
1: function Brute-Force($Task$)
2:     $S \leftarrow \emptyset$
3:     $Best \leftarrow \emptyset$
4:     $Miss \leftarrow 0$
5:     Recursion(0)
6:     return $Best$
```

Proposition 4.14 *Time complexity of the algorithm 4.10 is $\mathcal{O}(m \cdot (\frac{n}{m})^m)$. Space complexity is $\mathcal{O}(n)$.*

Proof Validation of schedule S now takes only $\mathcal{O}(m)$ and gets called for almost every recursive call, which executes $\mathcal{O}((\frac{n}{m})^m)$ times, so we actually also reduced the complexity to $\mathcal{O}(m \cdot (\frac{n}{m})^m)$.

In memory we only need to keep the task, schedule S and schedule $Best$, plus frames of recursive calls, which is $\mathcal{O}(m+n+m+m+m) = \mathcal{O}(m+n) = \mathcal{O}(n)$ in total. □

4.2.2 Weighted activities

For this problem, we can use the same recursive procedure as for the *Non-weighted scheduling problem.* We just need to change selection of best schedule.

Algorithm 4.11 Recursion

```
function Recursion($d$)
    if $d \geq |Task|$ then
        if $S$ is valid then
            if $w(S) > w(Best)$ then
                $Best \leftarrow S$
    else
        for all $I \in Task_d$ do
            $S \leftarrow S \cup \{I\}$
            Recursion($d + 1$)
            $S \leftarrow S \setminus \{I\}$
        Recursion($d + 1$)
```

Proposition 4.15 *Time complexity of the algorithm 4.11 is* $\mathcal{O}(m^2 \cdot (\frac{n}{m}+1)^m)$.

Proof Time complexity is the same as the time complexity of algorithm 4.8.
□

We can also use the same heuristics, with a small update to the second one. Insted of keeping the number of activities not in the schedule, we will keep a sum of the weights of the instances not in the schedule.

Algorithm 4.12 Recursion

```
function Recursion($d$)
    if $d \geq |Task|$ then
        if $w(S) > w(Best)$ then
            $Best \leftarrow S$
    else
        for all $I \in Task_d$ do
            if $S \leftarrow S \cup \{I\}$ is valid then
                $S \leftarrow S \cup \{I\}$
                Recursion($d + 1$)
                $S \leftarrow S \setminus \{I\}$
        $Penalty \leftarrow Penalty + w(Task_d)$
        if $Penalty < Optimal - w(Best)$ then
            Recursion($d + 1$)
        $Penalty \leftarrow Penalty - w(Task_d)$
```

Algorithm 4.13 Brute-Force

```
function Brute-Force($Task$)
    $S \leftarrow \emptyset$
    $Best \leftarrow \emptyset$
    $Penalty \leftarrow 0$
    $Optimal \leftarrow 0$
    for all $A \in Task$ do
        $Optimal \leftarrow Optimal + w(A)$
    Recursion(0)
    return $Best$
```

Proposition 4.16 *Time complexity of the algorithm 4.13 is $\mathcal{O}(m \cdot (\frac{n}{m})^m)$. Space complexity is $\mathcal{O}(n)$.*

Proof Value of $w(S)$ can be updated with every recursive call, keeping the complexity at $\mathcal{O}(m \cdot (\frac{n}{m})^m)$. The space complexity also stays $\mathcal{O}(n)$ as in algorithm 4.10. □

4.2.3 Weighted instances

We will again use the same recursive procedure as we did for *Weighted activities problem*, only where $w(S)$ is calculated in the correct way for this problem. Pseudocode is exactly the same as algorithm 4.11.

The second heuristic will again require a little modification. As weights of instances differ, the heuristic actually weakens, so we add a third heuristic, where we sort instances of each actitity descending by $w(I)$. This should enable the algorithm to get to better schedules more quickly and strenghten the heuristic cutting through the branches of search space.

Algorithm 4.14 Recursion

```
function Recursion($d$)
    if $d \geq |Task|$ then
        if $w(S) > w(Best)$ then
            $Best \leftarrow S$
    else
        $A \leftarrow Task_d$
        for all $I \in A$ do
            if $S \leftarrow S \cup \{I\}$ is valid then
                $Penalty \leftarrow Penalty + (w(A_0) - w(I))$
                if $Penalty < Optimal - w(Best)$ then
                    $S \leftarrow S \cup \{I\}$
```

```
12:                 Recursion($d + 1$)
13:                 $S \leftarrow S \setminus \{I\}$
14:             $Penalty \leftarrow Penalty - (w(A_0) - w(I))$
15:         $Penalty \leftarrow Penalty + w(A_0)$
16:         if $Penalty < Optimal - w(Best)$ then
17:             Recursion($d + 1$)
18:         $Penalty \leftarrow Penalty - w(A_0)$
```

Algorithm 4.15 Brute-Force

```
1: function Brute-Force($Task$)
2:     $S \leftarrow \emptyset$
3:     $Best \leftarrow \emptyset$
4:     $Penalty \leftarrow 0$
5:     $Optimal \leftarrow 0$
6:     for all $A \in Task$ do
7:         Sort $A$ descending by $w(I)$.
8:         $Optimal \leftarrow Optimal + w(A_0)$
9:     Recursion(0)
10:    return $Best$
```

Proposition 4.17 *Time complexity of the algorithm 4.15 is $\mathcal{O}(n \cdot \log(\frac{n}{m}) + m \cdot (\frac{n}{m})^m)$. Space complexity is $\mathcal{O}(n)$.*

Proof The initial sorting takes $\mathcal{O}(\sum\limits_{A \in Task} |A| \cdot \log(|A|)) = \mathcal{O}(m \cdot \frac{n}{m} \cdot \log(\frac{n}{m})) = \mathcal{O}(n \cdot \log(\frac{n}{m}))$. In total we get $\mathcal{O}(n \cdot \log(\frac{n}{m}) + m \cdot (\frac{n}{m})^m)$. The space complexity stays the same as in algorithm 4.13 again, hence $\mathcal{O}(n)$. □

Chapter 5

Results

5.1 Theoretical comparison

In this chapter, we will compare algorithms based on their their theoretical performance.

5.1.1 Non-weighted scheduling

Algorithms are compared in table 5.1.

Algorithm	Time complexity	Space complexity	Result
Greedy	$\mathcal{O}(n \cdot \log(n) + E)$	$\mathcal{O}(n + E)$	$\frac{3}{\Delta+2}$-approx.
Integer programming	$\mathcal{O}(n \cdot 2^n)$	$\mathcal{O}(n \cdot m)$	Optimal
Unified Algorithm	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n)$	$\frac{1}{2}$-approx.
Interval Search *(unit-cost)*	$\mathcal{O}(n \cdot \log(n) + (n - m) \cdot 2^m)$	$\mathcal{O}(n + 2^m)$	Optimal
Interval Search *(logarithmic-cost)*	$\mathcal{O}(n \cdot \log(n) + \frac{m \cdot (n-m) \cdot 2^m}{\log(n)})$	$\mathcal{O}(n + 2^m)$	Optimal
B-F with Heuristics	$\mathcal{O}(m \cdot (\frac{n}{m})^m)$	$\mathcal{O}(n)$	Optimal

Table 5.1: Comparison of algorithms.

Both *Greedy* and *Unified Algorithm* seem very usable for approximation solution. *Greedy* might get slower for dense graphs and return worse solutions according to it's approximation factor.

For optimal solution we have multiple choices. For tasks, where predicted complexity of *Interval Search* is acceptable, *Interval Search* is a smart choice. Otherwise, we can try heuristics of *Brute-Force with Heuristics*, especially if n is close to m. If there still seems to be no reasonable solution, we can try *Integer programming.*

5.1.2 Weighted activities

Algorithms are compared in table 5.2.

Algorithm	Time complexity	Space complexity	Result
Greedy	-	-	-
Integer programming	$\mathcal{O}(n \cdot 2^n)$	$\mathcal{O}(n \cdot m)$	Optimal
Unified Algorithm	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n)$	$\frac{1}{2}$-approx.
Interval Search *(unit-cost)*	$\mathcal{O}(n \cdot \log(n) + (n-m) \cdot 2^m)$	$\mathcal{O}(n + 2^m)$	Optimal
Interval Search *(logarithmic-cost)*	$\mathcal{O}(n \cdot \log(n) + \frac{m \cdot (n-m) \cdot 2^m}{\log(n)})$	$\mathcal{O}(n + 2^m)$	Optimal
B-F with Heuristics	$\mathcal{O}(m \cdot (\frac{n}{m})^m)$	$\mathcal{O}(n)$	Optimal

Table 5.2: Comparison of algorithms.

There turns out to be very little difference between *Weighted activities problem* and *Non-weighted scheduling problem.* The main difference is, that we can not use *Greedy* algorithm for this problem. Hence if we require fast solution for hard tasks, we have to go with *Unified Algorithm.*

5.1.3 Weighted instances

Algorithms are compared in table 5.3.

Algorithm	Time complexity	Space complexity	Result
Greedy	-	-	-
Integer programming	$\mathcal{O}(n \cdot 2^n)$	$\mathcal{O}(n \cdot m)$	Optimal
Unified Algorithm	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n)$	$\frac{1}{2}$-approx.
Interval Search *(unit-cost)*	$\mathcal{O}(n \cdot \log(n) + \log(n) \cdot (n-m) \cdot 2^m)$	$\mathcal{O}(n \cdot 2^m)$	Optimal
Interval Search *(logarithmic-cost)*	$\mathcal{O}(n \cdot \log(n) + m \cdot (n-m) \cdot 2^m)$	$\mathcal{O}(n \cdot 2^m)$	Optimal
B-F with Heuristics	$\mathcal{O}(m \cdot (\frac{n}{m})^m)$	$\mathcal{O}(n)$	Optimal

Table 5.3: Comparison of algorithms.

For this problem, *Interval Search* algorithm becomes less predictable and more time and space demanding. This makes it less usable. It's main advantage over other algorithms returning optimal solution was a good predictibility and thus an opportunity to ensure optimal result for in convenient time for certain tasks. Now we can consider *Brute-Force with Heuristics* to be a very good alternative especially if we cannot afford enough memory for *Interval Search. Integer programming* can again offer suprising performance.

5.2 Practical comparison

5.2.1 Real datasets

For the real-life data we used recent timetables at Faculfy of Informatics at Czech Technical University. It was neccessary to make a few alternations to the dataset.

Subjects at the university usually consist of weekly lectures, laboratory and tutorials. We consider letures, laboratory and tutorials as a separate subjects (activities).

Some of the subjects are teached only on even or odd weeks. This leads to the possibility of having different subjects at the same time of the week, but alternating on odd and even weeks. To handle this situation, we exploit so-called teaching blocks at the university. Teaching block is a 45-minute time interval. For each day, there are 15 teaching blocks available for the classes. Most classes require 2 teaching blocks, so the blocks are further coupled in paires. Now we reorder the blocks of odd and even weeks in the following order. For every day of the week, the pairs of blocks alternate over even and odd weeks. For the blocks numbering see table 5.4.

Odd week															
Mo	0	1	4	5	8	9	12	13	16	17	20	21	24	25	28
Tu	30	31	34	35	38	39	42	43	46	47	50	51	54	55	58
We	60	61	64	65	68	69	72	73	76	77	80	81	84	85	88
Th	90	91	94	95	98	99	102	103	106	107	110	111	114	115	118
Fr	120	121	124	125	128	129	132	133	136	137	140	141	144	145	148

Even week															
Mo	2	3	6	7	10	11	14	15	18	19	22	23	26	27	29
Tu	32	33	36	37	40	41	44	45	48	49	52	53	56	57	59
We	62	63	66	67	70	71	74	75	78	79	82	83	86	87	89
Th	92	93	96	97	100	101	104	105	108	109	112	113	116	117	119
Fr	122	123	126	127	130	131	134	135	138	139	142	143	146	147	149

Table 5.4: Numbering of teaching blocks.

Some of the classes require odd number of blocks. This again leads to a problem. We deform such classes by moving the non-continuous block closer to the others to create a continuous sequence of blocks. Tasks containing instances of other non-continuous sequences of blocks are removed from the dataset.

Agorithms are rated by execution time they need to process the whole dataset and the score they achieve. Score on the dataset is calculated as a sum of weights of the returned schedules for each task in the dataset.
Formally: $score = \sum_{Task \in Dataset} w(S)$

NW For the *Non-weighted scheduling problem*, we create a dataset *NW*, consisting of real data with non-weighted instances.

WA For the *Weighted activities problem*, we create a dataset *WA*, consisting of real data with weighted activities. Activities are weighted in the following way. For all lectures $w(A) = 1$. For all laboratories $w(A) = 2$. For all tutorials $w(A) = 4$.

Because tutorials are obligatory at CTU and on the other hand lessons are often skipped by students, we will offer more drastically weighted dataset, which may be closer to the requirements of some students.

WA2 For the *Weighted activities problem*, we create a dataset *WA2*, consisting of real data with weighted activities. Activities are weighted in the following way. For all lectures $w(A) = 1 + r$, where r is a pseudorandom integer $r \in [0, 1000)$. For all laboratories $w(A) = 1000 + r$, where r is a pseudorandom integer $r \in [0, 100)$. For all tutorials $w(A) = 1000000 + r$, where r is a pseudorandom integer $r \in [0, 10)$.

WI For the *Weighted instances problem*, we create a dataset *WI*, consisting of real data with weighted instances. Weighting includes both subject preferences and time preferences. Instances are weighted in the following way. For all lectures $w(A) = 800 + t + r$, for all laboratories $w(A) = 900 + t + r$ and for all tutorials $w(A) = 1000 + t + r$, where r is a pseudorandom integer $r \in [0, 10)$ and t is an integer calculated as an average of values of blocks, required by the class (instance). Values of blocks are described in table 5.5. Weighting model in this dataset significantly penalizes missing subject in the schedule and should be very close to the requirements of students at most universities.

Mo	20	40	60	80	80	80	60	60	80	80	80	80	60	40	20
Tu	20	40	60	80	80	80	60	60	80	80	80	80	60	40	20
We	20	40	60	80	80	80	60	60	80	80	80	80	60	40	20
Th	20	40	60	80	80	80	60	60	80	80	80	80	60	40	20
Fr	20	40	60	80	80	80	60	60	80	80	80	80	60	40	20

Table 5.5: Values of teaching blocks.

5.2.2 Synthetic datasets

To test performance of algorithms with large data, we created synthetic datasets with large inputs.

SNW For the *Non-weighted scheduling problem*, we create a dataset *SNW* in the following way. $m = \max(1, r_m)$ where r_m is a pseudorandom integer $r_m \in [0, 200]$. $n = \max(1, r_n)$ where r_n is a pseudorandom integer $r_n \in [0, 3000]$. Each activity then recieves one instance. The rest of instances are pseudorandomly assigned to the activities. We create two variables *spi* and *spid*. $spi = max(4, r_{spi})$, where r_m is a pseudorandom integer $r_{spi} \in [0, 100]$. $spid = \left\lceil \frac{spi}{4} \right\rceil$. For each instance I, we choose a duration d_I as $d_I = [spi - spid, spi + spid]$. Then $b(I) = r_b$, where r_b is a pseudorandom integer $r_m \in [0, 3000 - d_I)$. Then $e(I) = b(I) + d_I$.

SWA For the *Weighted activities problem*, we create a dataset *SWA* in the following way. We use the same procedure as we used for creating dataset *SNW*. Then we set each $w(A)$ to a pseudorandom number in range $[0, 10000]$.

SWI For the *Weighted instances problem*, we create a dataset *SWI* in the following way. We use the same procedure as we used for creating dataset *SNW*. Then we set each $w(I)$ to a pseudorandom number in range $[0, 10000]$.

5.2.3 Testing environment

To ensure a fair comparison of algorithms by execution times, we used a computational server provided by Czech Technical University.

Computational server STAR

- Based on *blade* architecture by IBM.
- Central computer working as an independent server, providing connection to the computational blades. So-called front-end.
- Total of 14 computational blades (nodes).
  - node-0001 to node-0008 - IBM BladeCenter LS21
    - 2x AMD Opteron 2218 2,6GHz dual-core (total of 4 computational cores)
    - 8GB RAM PC2-5300 CL5 ECC DDR2 SDRAM VLP RDIMM
  - node-0009 to node-0014 - IBM BladeCenter LS22
    - 2x AMD Opteron 6C Processor Model 2435 2.6GHz/6MB L3 (total of 12 computational cores)
    - 26GB RAM PC2-6400 CL6 ECC DDR2 800 VLP RDIMM
- Tasks are distributed to the nodes by Sun Grid Engine.
- Connection to the front-end server is ensured by ssh.
- Multi-core processing is controlled with OpenMP.

For our tasks, we used 12-core nodes. Each task from benchmarked dataset was computed on a separate core. For memory-demanding/memory-intense algorithms, we used only a single core, to eliminate the influence of limited size of cache.

5.2.4 Measuring methodology

All algorithms were implemented in `C++` [4] language and compiled with `g++` [5] with O3 optimalization enabled. Execution time was measured with *omp_-get_wtime()* function provided by OpenMP [6]. Each dataset was executed 10 times to eliminate the influence of operating system. Final execution time for each task was calculated as an average of all execution times.

Formally $t = (\sum_{i=1}^{10} \frac{t_i}{10}$, where t_i is the i-th execution time of the task.

5.2.5 Greedy

We have tested *Greedy* algorithm with datasets NW an SNW. Time performance was very good as expected, but weights of returned schedules turned out to be above our expectations, as the practical score of the algorithm on dataset *NW* (we emphasize this dataset, because we were able to determine the optimal results for this dataset) was much higher than the approximation factor of the algrithm suggested. See comparison tables in subsection 5.2.10 for details.

In section 3.1 we claimed compexity $\mathcal{O}(n \cdot \log(n) + E)$ to be more precise than $\mathcal{O}(n^2)$. This was confirmed in practical tests. In graph 5.1 showing the performance of the algorithm on dataset *NW*, we can observe, that for the same number of instances, the performance becomes worse with lower number of activities. Same phenomenon can be seen in graph 5.2, showing the performance of *Greedy* on dataset *SNW*. This is because with lower number of activities, the average number of instances per activity increases and thus increases also the density of the graph. This leads to worse performance, because the performance depends also on the density of the graph, as we proved in section 3.1.

Advantage of $\mathcal{O}(n^2)$ complexity estimation is, that this value is known as soon as we know the size of n, unlike $\mathcal{O}(n \cdot \log(n) + E)$, which needs to be computed in $\mathcal{O}(n \cdot \log(n))$ time.

5.2.6 Integer programming

Because integer programming is very complicated, we used open-source integer programming solver *lpsolve* [7].

Integer programming was tested with datasets *NW*, *WA*, *WA2* and *WI*. For synthetic datasets, *lpsolve* did not manage to load the input data. Performance on real datasets was surprisingly good. Integer programming achieved the best execution time of all tested algorithms finding an optimal solution. See comparison table in subsection 5.2.10 for details. From the results of the testing, performance of the algorithm seem to be dependent on n, which can be seen in graphs 5.3 showing the performance of the algorithm on dataset *NW*.

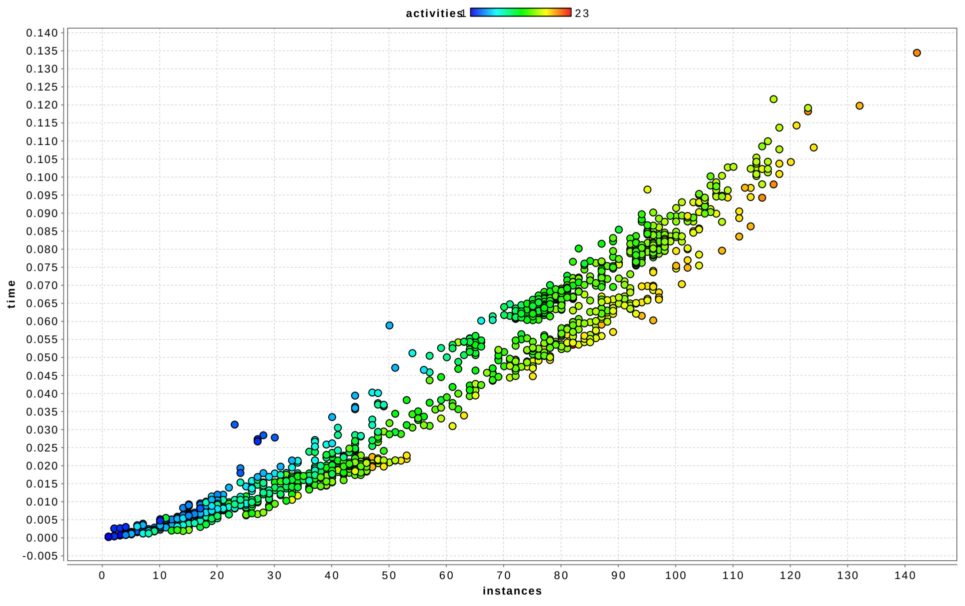

Figure 5.1: *Greedy* performance on dataset *NW*.

5.2.7 The Unified Algorithm

Performace of *The Unified Algorithm* turned out to be the best among all algorithms in terms of execution time. This algorithm handled all datasets and reached the best execution times for all datasets. On the other hand, it recieved the worst score for the returned schedule quality on real datasets. If we look at scores in subsection 5.2.10, we can see that this algorithm still reached over 96% score of the optimal solution, which is significantly more than the performance guarantee. Performance of the algorithm is displayed in graph 5.4 showing the performance on dataset *SWI*.

5.2.8 Interval Search

Interval search algorithm was tested on the real dataset only, because the testing environment did not have enough memory to run this algorithm with synthetic datasets, as this algorithm uses exponential memory space. For the comparison of the optimalization shown in 4.3 see graph 5.5. Detailed results are shown in comparison tables in subsection 5.2.10.

5.2.9 Brute-Force with Heuristics

Performance of this algorithm is strongly dependent on heuristics. Graph 5.6 showing performance of the algorithm on dataset *WI* ilustrates how much can the execution times differ depending on the effectivness of used heuristic. This can be either used as an advantage or disadvantage. We will use it as an

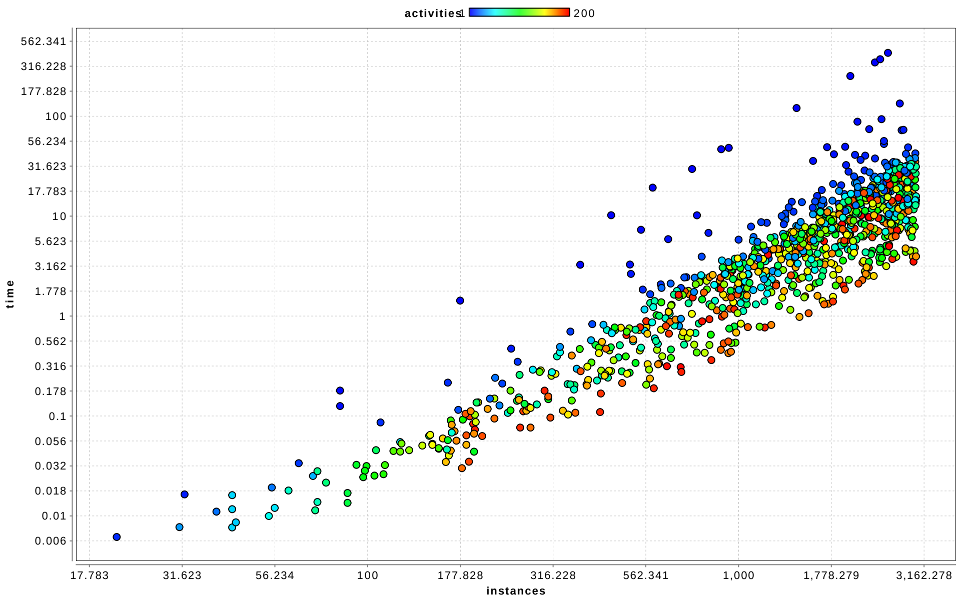

Figure 5.2: *Greedy* performance on dataset *SNW*.

advantage and try this approach when other methods fail to find a solution. On the other hand, most of the tasks in synthetic datasets ended up with a time limit exceeded, hence we did not achieve the optimal solution.

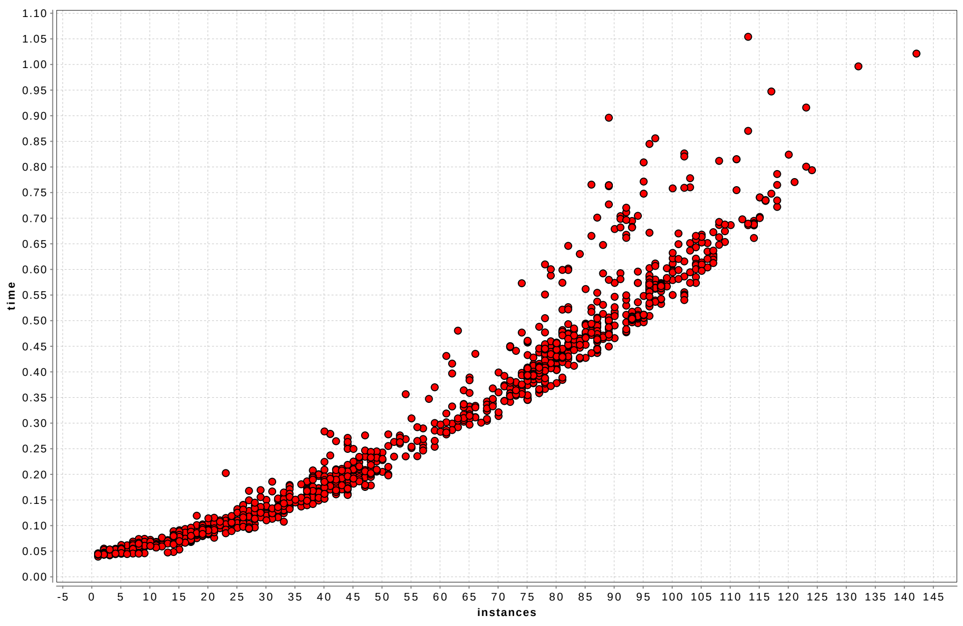

Figure 5.3: *Integer programming* performance on dataset *NW*.

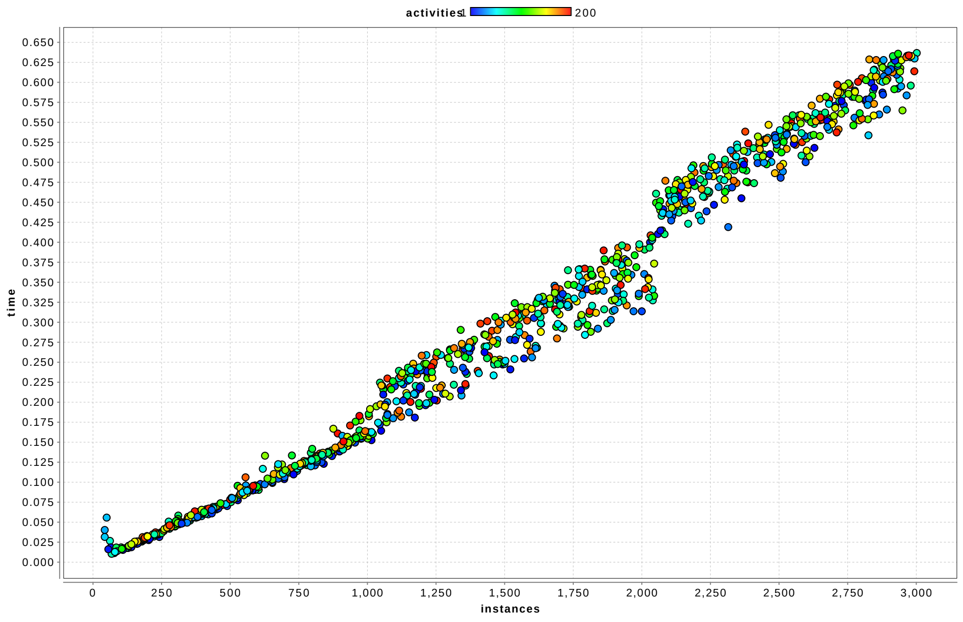

Figure 5.4: *The Unified Algorithm* performance on dataset *SWI*.

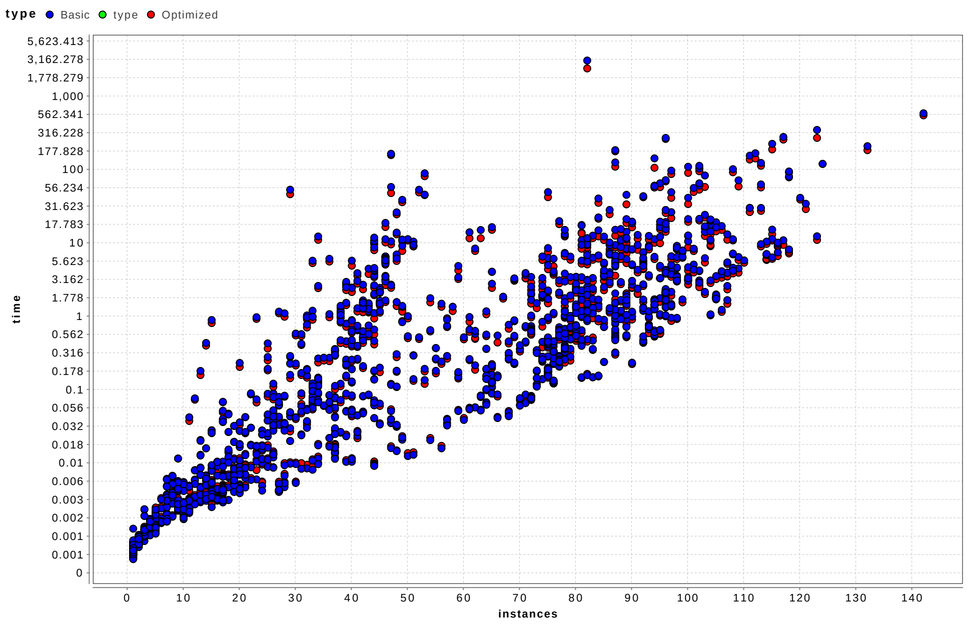

Figure 5.5: Optimized and non-optimized *Interval Search* performance on dataset *NW*.

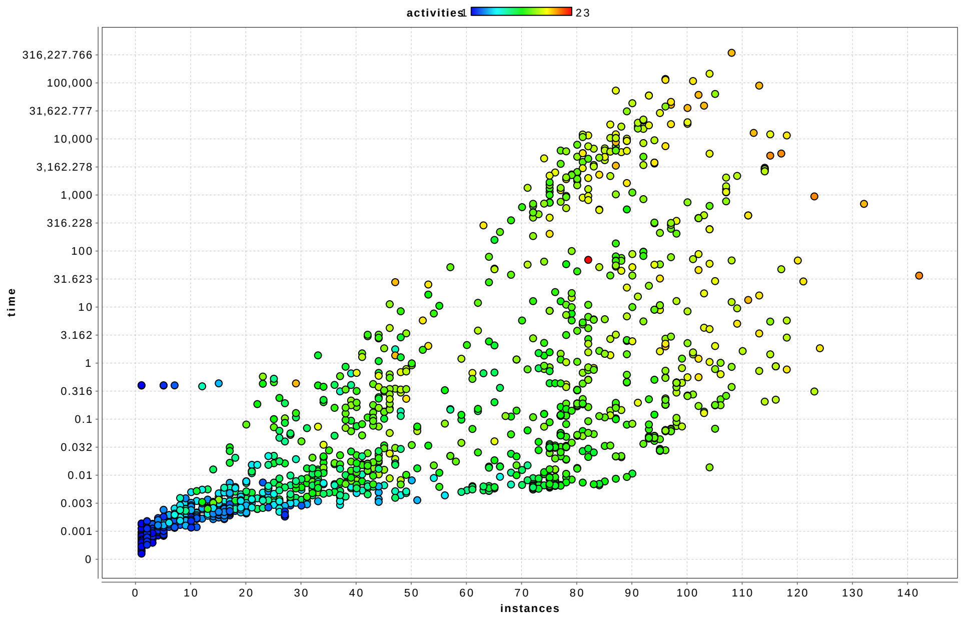

Figure 5.6: *B-F with Heuristics* performance on dataset *WI*. Significant deflections on the left size are caused by loading memory blocks of greater size than required, which is performed always for the first few tasks.

5.2.10 Comparison

We will compare the results of measuring for individual datasets in tables 5.6 to 5.12. Algorithms are limited by execution time. Exceeding the time limit is denoted by TLE.

Algorithm	Time	Score	Maximum time
Greedy	60.093 ms	15117	0.135 ms
Integer programming	448.928 ms	15589	1.055 ms
Unified Algorithm	14.299 ms	15108	0.026 ms
Interval Search	10998.442 ms	15626	3094.077 ms
Interval Search Optimized	9531.083 ms	15626	2414.423 ms
B-F with Heuristics	1987999.776 ms	15626	158371.816 ms

Table 5.6: Comparison of algorithms on dataset *NW*.

Algorithm	Time	Score	Maximum time
Integer programming	831.506 ms	38114	2.505 ms
Unified Algorithm	14.491 ms	37534	0.027 ms
Interval Search	17247.665 ms	38185	4783.041 ms
Interval Search Optimized	15136.104 ms	38185	3949.352 ms
B-F with Heuristics	34540727.510 ms	38185	600000 ms (TLE)

Table 5.7: Comparison of algorithms on dataset *WA*.

Algorithm	Time	Score	Maximum time
Integer programming	905.987 ms	6815081394	3.210 ms
Unified Algorithm	14.498 ms	6775056981	0.027 ms
Interval Search	17225.425 ms	6817082679	4767.409 ms
Interval Search Optimized	15144.037 ms	6817082679	3954.844 ms
B-F with Heuristics	38250488.204 ms	6817082669	600000 ms (TLE)

Table 5.8: Comparison of algorithms on dataset *WA2*.

Algorithm	Time	Score	Maximum time
Integer programming	752.362 ms	15269881	1.969 ms
Unified Algorithm	14.847 ms	14425456	0.027 ms
Interval Search	78341.159 ms	15304872	8200.474 ms
Interval Search Optimized	74736.609 ms	15304872	7339.717 ms
B-F with Heuristics	2321805.572 ms	15304872	351359.854 ms

Table 5.9: Comparison of algorithms on dataset *WI*.

Algorithm	Time	Score	Maximum time
Greedy	10163.976 ms	60592	435.213 ms
Unified Algorithm	28.972 ms	57586	0.633 ms
B-F with Heuristics	20880453.185 ms	49591	30000 ms (TLE)

Table 5.10: Comparison of algorithms on dataset *SNW*.

Algorithm	Time	Score	Maximum time
Unified Algorithm	305.663 ms	325197481	0.665 ms
B-F with Heuristics	29292878.284 ms	264845349	30000 ms (TLE)

Table 5.11: Comparison of algorithms on dataset *SWA*.

Algorithm	Time	Score	Maximum time
Unified Algorithm	298.912 ms	433052982	0.637 ms
B-F with Heuristics	25053962.787 ms	389725573	30000 ms (TLE)

Table 5.12: Comparison of algorithms on dataset *SWI*.

5.3 Summary

We have compared individual algorithms and noticed significant differences between each other. Now, we will try to make a unified model to solve the problem based on our needs and complexity of individual tasks.

1. If complexity of *Interval Search* for the task is suitable for your needs, then use this algorithm.

2. Try *Brute-Force with Heuristics* with time limit suitable for your needs.

3. If heuristis of *Brute-Force with Heuristics* did not work well and the algorithm did not return an optimal schedule, try using *Integer programming* again with time limit suitable for your needs.

4. If none of the previous algorithms performed well enough, use *The Unified Algorithm*, which should run in the best time of all algorithms and return reasonably good schedule. If you are solving *Non-weighted scheduling problem*, you can also try *Greedy* algorithm and choose the better solution.

Conclusion

This work describes algorithms for scheduling classes in student schedules. We described existing algorithms and also designed our own. For testing of algorithms we used both synthetic datasets and real-life data based on student schedules at CTU. Algorithms were compared in both theory and practical tests.

Discussed Problem turned out to be generalizable to known NP-complete problems such as finding a maximum independent set in a graph or solving binary integer programming. Therefore there are no efective solutions available. We found efective algorithms for finding an approximation of optimal solution and tried different approaches to reach optimal solution.

Algorithms aiming for approximation of optimal solution turned out to have very good practical results and huge advantage in execution time. Integer programming have shown great compromise between result quality and execution time. Our methods, aiming on optimal results, have shown a good usability especially for inputs of limited size.

We managed to improve approximation factor of one of existing algorithms and also extended implementation options of this algorithm.

We offered a procedure of finding result using a combination of multiple algorithms, to reach the best result with consideration of individual tasks.

Future work

The current methods do not include genetic programming, which is often used to generate university schedules and solving other scheduling problems. Therefore it should be possible to use it to generate student timetables too.

The Unified Algorithm discussed in this thesis has a performance guarantee of $\frac{1}{2}$, but practical results have shown much better performance. Therefore there might be possibility to improve the approximation factor of the algorithm.

Bibliography

[1] Vazirani, V. V. *Approximation Algorithms.* New York, NY, USA: Springer-Verlag New York, Inc., 2001, ISBN 3-540-65367-8.

[2] Halldórsson, M.; Radhakrishnan, J. Greed is Good: Approximating Independent Sets in Sparse and Bounded-degree Graphs. 1994: pp. 439–448, doi:10.1145/195058.195221. Available from: `http://doi.acm.org/10.1145/195058.195221`

[3] Bar-Noy, A.; Bar-Yehuda, R.; Freund, A.; et al. A Unified Approach to Approximating Resource Allocation and Scheduling. *J. ACM*, volume 48, no. 5, Sept. 2001: pp. 1069–1090, ISSN 0004-5411, doi:10.1145/502102.502107. Available from: `http://doi.acm.org/10.1145/502102.502107`

[4] cplusplus. cplusplus.com. 2000. Available from: `http://www.cplusplus.com/`

[5] Free Software Foundation, I. GCC, the GNU Compiler Collection. 2014. Available from: `https://gcc.gnu.org/`

[6] OpenMP. The OpenMP®API specification for parallel programming. 1998. Available from: `http://openmp.org/wp/`

[7] Free Software Foundation, I. lpsolve. 1991. Available from: `http://lpsolve.sourceforge.net/`

[8] Leung, J.; Kelly, L.; Anderson, J. H. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* Boca Raton, FL, USA: CRC Press, Inc., 2004, ISBN 1584883979.

Appendix A

Acronyms

CTU	Czech Technical University
TLE	Time limit exceeded

Appendix B

Contents of enclosed CD

```
│ readme.txt ...................... the file with CD contents description
├── src .................................. the directory of source codes
│   ├── code ................................... implementation sources
│   ├── data ................................................. datasets
│   └── lpsolve ............................................... lpsolve
├── thesis.pdf ......................... the thesis text in PDF format
└── thesis.tex ......................... the thesis text in TEX format
```