Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Michal Frdlík**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Algorithms and Tests for Human Resource Management Tools**

Guidelines:

1) make the review of the activity scheduling algorithms
2) design appropriate algorithm for incremental activity scheduling
3) implement and evaluate the algorithm
4) choose appropriate testing methodology for the resource management tool
5) implement the tests on different levels of abstraction

Bibliography/Sources:

Cayirli, T. and E. Veral (2003). "OUTPATIENT SCHEDULING IN HEALTH CARE: A REVIEW OF LITERATURE." Production and Operations Management 12(4): 519-549.
Cardoen, B., E. Demeulemeester, et al. (2010). "Operating room planning and scheduling: A literature review." European Journal of Operational Research 201(3): 921-932.

Willy Herroelen and Bert De Reyck and Erik Demeulemeester,
Resource-constrained project scheduling : A survey of recent developments,
Computers and operations research, volume 25, number 4, 279-302,
1998

Diploma Thesis Supervisor: prof.Dr.Ing. Zdeněk Hanzálek

Valid until the end of the summer semester of academic year 2014/2015

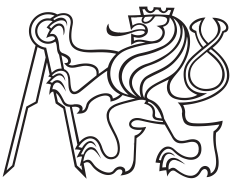
L.S.

doc. Ing. Filip Železný, Ph.D.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, March 3, 2014

Master's thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Algorithms and Tests for Human Resource Management Tools

Bc. Michal Frdlík
Open Informatics, Artificial Intelligence

February 2014
Supervisor: Prof. Dr. Ing. Zdeněk Hanzálek

Acknowledgement / Declaration

I would like to express my gratitude to my supervisor, Prof. Dr. Ing. Zdeněk Hanzálek, for his support, professional guidance, and for his patience.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

/

I hereby declare, that I have completed this thesis on my own and I have properly specified all the information sources used, according to the Guideline about observance of the ethic principles concerning university theses.

V Praze 13.12.2014

Abstrakt / Abstract

Tato práce se zaměřuje na silně omezený rozvrhovací problém nalezení rozvrhu kurzů s neúplnou a časově závislou informací o těchto kurzech. Cílem této práce je navrhnout a implementovat efektivní algoritmus, který by tento problém řešil, respektive softwarový nástroj, který by tento algoritmus využíval.

Na základě rešerše moderních algoritmů týkajících se tohoto problému je navrhnut algoritmus nový, využívající vnitřního optimalizačního algoritmu, a tedy se jedná o meta-algoritmus. Na základě analýzy je několik optimalizačních algoritmů podrobeno důkladnému testování a následně je nejvhodnejší z nich zvolen a implementován jako součást meta-algoritmu. Meta-algoritmus sám je následně podroben testům.

Následně je navržen a implementován softwarový nástroj, který tohoto meta-algoritmu využívá, a jeho zdrojový kód je podroben jednotkovým a integračním testům, jejichž metodologie je v práci také popsána.

Klíčová slova: rozvrhování, metodologie testů, algoritmy

Překlad titulu: Algoritmy a testy pro nástroj na organizaci zaměstnanců

This thesis concerns the problem of strongly constrained course timetabling with imperfect time-dependent information taken into account. The aim of this thesis is to design and implement an effective incremental activity scheduling algorithm which would solve this problem.

After reviewing the state of the art, a novel meta-algorithm is proposed, which encapsulates an inner search algorithm. Several suitable inner search algorithms are subjected to benchmarks and the best one is found and chosen as the inner search algorithm. The meta-algorithm itself is then also subjected to benchmarks in order to find best settings for it.

Then the software tool which makes use of this algorithm is designed, implemented and its code base is covered by unit and integration tests. General software testing and algorithm benchmarking methodology is described.

Keywords: scheduling, testing methodology, algorithms

Contents /

Tables /

Chapter 1
Introduction

1.1 Aim of This Thesis

The main aim of this thesis is to design and implement an incremental activity scheduling algorithm suitable for the problem specified in 1.2, which shall achieve good results in terms of both optimality and effectiveness. Besides the main aim, this work shall also provide a description of general benchmarking and testing methodology, suitable for algorithms similar to the aforementioned kind. Several goals are specified in 1.1.1.

1.1.1 Goal Specification

- Review of the SotA activity scheduling methods and algorithms
- Evaluation of these methods and algorithms
- Design of an incremental activity scheduling algorithm suitable for the given problem
- Implementation and evaluation of the above mentioned algorithm
- Choice and specification of the appropriate testing methodology
- Implementation of tests on different levels of abstraction

1.2 Problem Definition

The problem of our concern is a strongly constrained course timetabling problem with imperfect information taken into account. It features students, teachers and courses. The task is to construct a timetable, which assigns students and teachers to courses, and courses to their respective days and periods, and assures that none of the hard constraints are violated. Moreover, the scheduling process itself has to happen in a serial manner, i.e. as the time goes on (and some of the scheduled activities are already in progress), new students may arrive, immediately demanding timetables for their curricula. When these new timetables are constructed, one must take into account, that previously scheduled activities *cannot* change.

Character of the given percentage of the forthcoming students' curricula is known a priori and the remaining part is supposed to have the same probability distribution.

In a valid timetable, these constraints need to hold (they are reffered to as hard constraints):

- No student may be assigned to two or more classes at the same time
- No teacher may be assigned to two or more classes at the same time
- Classes need to have exactly one student and one teacher assigned
- Classes need to be scheduled at most once per day
- If a class C is scheduled, it needs to be scheduled to exactly k consecutive periods, where k is the length of C
- A teacher must not be assigned to the period of his unavailability (lunch time)

- Every student must be assigned to all the class types C he has in his curriculum and exactly $volume(C)$ times
- No student has to have any class scheduled before the time of his arrival

The aforementioned constraints are enough to ensure the feasibility of the resulting timetable, but it may not be sufficient in terms of optimality. Hence, the following properties, which improve the objective function, but need not to hold, are specified (they are reffered to as soft constraints):

- Length of the daily schedule should approach its minimal possible value
- Classes of the same type (of a given student) should be evenly distributed throughout the days and should be equidistant if possible
- Daily workload of the teachers should approach mean workload
- Average gap between subjects should approach zero

1.2.1 Problem Complexity

In order to justify the time and space complexity of the methods and algorithms proposed in this thesis, it needs to be shown, that the underlying timetabling decision problem (i.e. problem that decides whether a valid timetable exists w.r.t. given resources and constraints) is NP-Complete, and therefore that the corresponding optimisation problem (i.e. problem that finds the valid timetable) is NP-Hard. This shall be done using a sequence of proofs, but firstly, some fundamental concepts ([1]) of complexity theory are briefly reviewed.

1.2.1.1 Definition. ***Turing machine.*** Informally, a Turing machine can be viewed as a physical machine consisting of a *control unit* (which can hold a single state), a *tape* of infinite length (divided to discrete fields) and a *head* (which can read from and write to the tape). According to the actual tape symbol (i.e. a symbol read by the head) and the actual state, the control unit changes the actual state and moves the head either to the left or to the right, according to the transition function. Formally, a Turing machine is a seven-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, Γ is a finite set of tape symbols (where $\Sigma \subseteq \Gamma$), B is an empty symbol (also called *a blank*), δ is a transition function (i.e. a partial mapping $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$, where L and R represent the transition of the *head* to the left or to the right, respectively), $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of terminal states.

1.2.1.2 Definition. ***Nondeterministic Turing machine.*** Nondeterministic Turing machine (NTM) is a Turing machine, which is allowed (at every step) to branch the computation to several branches, that evaluate simultaneously. In other words, an NTM is allowed to be in multiple states at one time.

1.2.1.3 Definition. ***Time complexity of a Turing machine.*** Time complexity of a Turing machine is the maximal number of steps, after which the machine successfully halts. The maximum is evaluated over all possible initial configurations of the tape.

1.2.1.4 Definition. ***Decision problem.*** Decision problem is a problem, for which there exists a Turing machine, which can decide it (i.e. for every instance of the problem (input tape) it can give a yes/no answer).

1.2.1.5 Note. Turing machine can be adapted to simulate the logic of any computer algorithm.

1.2.1.6 Definition. ***Class P.*** We say that a decision problem U lies in class P if and only if there exists a Turing machine, which can decide it and operate in polynomial time.

1.2.1.7 Definition. ***Class NP.*** We say that a decision problem U lies in class P if and only if there exists a Nondeterministic Turing machine, which can decide it and operate in polynomial time.

1.2.1.8 Definition. ***Polynomial reduction.*** We say that a decision problem U is polynomially reducible to a decision problem V if and only if there exists an algorithm (modeled by some Turing machine), which can convert any input of the problem U to the input of the problem V, while operating in polynomial time and assuring that every YES input of U translates to YES input of V, and every NO input of U translates to NO input of V.

1.2.1.9 Definition. ***Class NP-Complete.*** We say that a decision problem U lies in class $NP\text{-}Complete$ if and only if U lies in NP and every decision problem that lies in NP can be polynomially reduced to U.

1.2.1.10 Definition. ***SAT problem.*** A SAT (boolean satisfiability) problem is the problem of deciding whether a formula in general conjunctive normal form is satisfiable by some interpretation.

1.2.1.11 Theorem. ***Cook, 1971.*** SAT problem is NP-Complete[2].

1.2.1.12 Theorem. 3-SAT problem is NP-Complete.

1.2.1.13 Proof. Firstly, 3-SAT lies in NP, because any interpretation can be evaluated in linear time. Let $\phi \equiv C_1 \wedge C_2 \wedge \ldots \wedge C_n$ be a formula in CNF. Then for every $C_i \equiv l_1 \vee l_2 \vee \ldots \vee l_k$, where $k > 3$, construct new formula X_i using the following prescription:

$$X_i \equiv (l_1 \vee l_2 \vee x_1) \wedge (\neg x_1 \vee l_3 \vee x_2) \wedge (\neg x_2 \vee l_4 \vee x_3) \wedge \ldots \wedge (\neg x_{k-3} \vee k_{k-1} \vee l_k)$$

Then $C_i \models X_i$ and $X_i \models C_i$, and by replacing C_i for X_i in the original formula, we get 3-CNF formula, Q.E.D.

1.2.1.14 Definition. ***Graph colouring problem.*** A graph $G = (V, E)$ is 3-colourable if and only if there exists a mapping $colour : V \to \mathbb{N}$, such that for every pair of vertices $\{v_1, v_2\}$ that are adjacent (coonected through an edge) holds that $colour(v_1) \neq colour(v_2)$.

1.2.1.15 Definition. ***Graph 3-colouring problem.*** A graph $G = (V, E)$ is 3-colourable if and only if there exists a mapping $colour : V \to \{1, 2, 3\}$, such that for every pair of vertices $\{v_1, v_2\}$ that are adjacent (coonected through an edge) holds that $colour(v_1) \neq colour(v_2)$.

1.2.1.16 Theorem. 3-SAT is polynomially reducible to 3-colouring[3]

1.2.1.17 Definition. ***General timetabling problem.*** In a general timetabling problem, there are three sets defined: T (set of time slots), R (set of resources) and M (set of meetings). Every meeting consists of a timeslot on which it is supposed to happen, and a set of resources, that will be taken by it (in our case one teacher and one student).

We ask if there is a configuration, in which no resource is used twice or more in the same timeslot.

1.2.1.18 Theorem. 3-colouring is polynomially reducible to General timetabling problem.

1.2.1.19 Proof.[4] Firstly, general timetabling problem lies in NP, because consistency violations can be detected simply by checking all the time slots for multiple presence of the same resource. Now suppose the following graph k-colouring instance: $G = (V, E)$, where $|V| = n$ and $|E| = m$. Construct the folowing general timetabling instance $T = \{t_1 \dots t_k\}$, $R = \{r_1 \dots r_m\}$ and $M = \{M_1 \dots M_n\}$, where $M_i = \{t \in T, \{c_{i,1} \dots c_{i,k_i}\}\}$, where the resources $c_{i,1} \dots c_{i,k_i}$ in the meeting M_i are the resources r_j, such that e_j is adjacent to v_i in G. Suppose that k-colouring $colour : V \to \{1, \dots, k\}$ exists for G. Then assign t_k where $k = colour(v_i)$ to M_i for every i. The condition $colour(v_i) \neq colour(v_j)$ whenever $\{v_i, v_j\} \in E$ guarantees, that meetings which share resources are assigned to different time-slots. Also a valid timetabling configuration guarantees, that $colour$ exists for G, Q.E.D.

1.3 The Challenge

The underlying timetabling problem of 1.2 is well studied. Many optimal and heuristic techniques exist. Unfortunately, that cannot be said about the primary problem (see 2) of dealing with uncertainty and iterative nature.

Generally, the objective functions which are commonly used for timetabling tend to have *superadditive* nature:

$$f(s(C_1 + C_2, \{\})) \geq f(s(C_1, C_2)),$$

where C_1 and C_2 are sets of constraints, function $s(A, B)$ maps sets of constraints to schedules (where set B is applied *after* set A, with respect to the constraints, that arise from application of A), and function f is the objective function. Operator $+$ does union on its operands. Operator $\geq$ represents relation "better".

On the left side of the inequality, constraint set C_2 is known a priori (which is the fully informed case), and on the right side, C_2 is known after C_1 is applied (which is the uninformed or partially informed case). The partially informed case can never be better than the informed one, given that s creates optimal schedules.

Let us show an example using c_{max} objective function (i.e. function that maps schedules to their lengths).

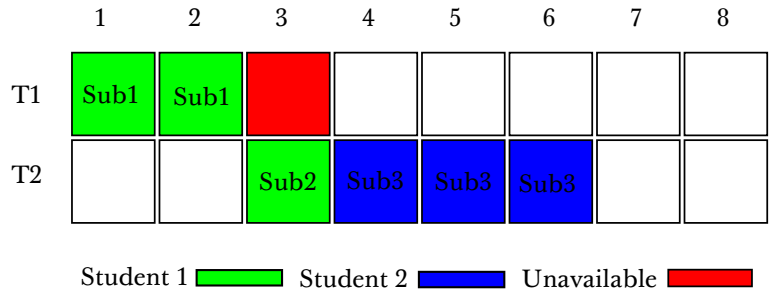


Figure 1.1. Uninformed iterative scheduling.

Let us have a simplified timetabling problem with two teachers ($T1$ and $T2$) teaching the same subjects. Teacher $T1$ is unavailable at time period 3. Student 1 has subjects $Sub1$ and $Sub2$ in his curriculum, whereas student 2 only has subject $Sub3$.

The effect of iterative scheduling applied to this problem is depicted in figure 1.1. In this situation, student 1 is optimally scheduled (subject 2 cannot be scheduled to timeslots 1,2 of teacher 1 and 2, nor to timeslot 3 of teacher 1, due to consistency constraints or unavailability constraints, respectively) and then student 2 is optimally scheduled to the actual schedule.

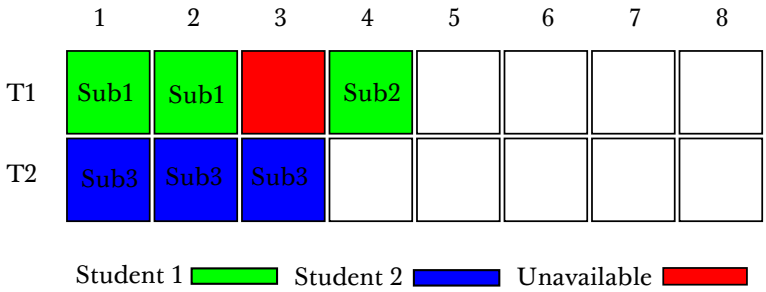


Figure 1.2. Optimal scheduling with complete information.

On the other side, when both students' curriculla are known a priori, as depicted in figure 1.2, the resulting schedule is optimal, having $c_{max} = 4$ instead of 6.

In the problem 1.2, only incomplete information about forthcoming students is known a priori. It is rather obvious, that having some kind of information about forthcoming students is *conditio sine qua non* of successful iterative scheduling. The question is, how to utilise this information.

Chapter 2
Review of Literature

In recent years, a considerable attention has been devoted to the automated timetabling problem and its variations, yet nobody seems to have been investigating the problem of our concern (as defined in 1.2).

In this chapter, publications relevant to 1.2 shall be reviewed.

2.1 Literature Concerning Iterative Scheduling

By "iterative scheduling" it is meant the iterative nature of the problem (as new students arrive, the scheduling procedure is repeated with existing students' schedules taken into account).

2.1.1 Reactive Scheduling

The term "reactive scheduling" is used by Herroelen and Leus in [5] and [6] to describe resource constrained project scheduling problem (RCPSP), where tasks arrive over time and their processing time is unknown. Arrival of the task is known a priori, but the processing time and the exact time of arrival is not. It is the exact opposite of our problem, where times of arrival and processing times are known a priori, but tasks are not. Moreover, in [5] and [6], there is a strong accent on precedence constraints, which are not important in our case. Simplicity of constraints defined in these articles and no implementation description renders the methods mentioned unusable for our problem.

2.1.2 Dynamic Timetabling

The term "dynamic timetabling" is coined in [7], where it is most promisingly stated, that most of the approaches to the timetabling problem focus on the timetable construction as a static process. Authors of the article then propose a reactive constraint agents architecture, which can cope with real-time changes of a timetable with minimal modification. By "changes of the timetable" they mean changes of constraints. Sadly, in our problem, constraints are strictly forbidden to change, which, again, renders the methods mentioned in this article unusable for our problem.

In [8], there is an RCPSP based method proposed, but as in [7], an already scheduled timetable is allowed to change, which is unacceptable in our problem, because it violates the most important constraint.

2.1.3 Online Scheduling

Articles concerning online scheduling, such as [9], are mostly thorough mathematical analyses of basic scheduling instances, such as $P|r_j, pmtn| \sum_j w_j C_j$ and $P|r_j| \sum_j w_j C_j$, without further constraints defined, heavily theoretical and with no practical use in this thesis.

2.1.4 Fuzzy Scheduling

In [6], there is a mention about fuzzy scheduling, but again, only fuzzy processing times are taken into account.

2.2 Literature Concerning Timetabling

Contrary to literature concerning iterative scheduling, there is a vast ammount of articles concerning timetabling (and for our problem in particular—university course timetabling and examination timetabling). The method range is very wide—starting with well known and well studied *integer linear programming* (also known as mixed integer programming), which also has very good software support, and ending with simple specialised local search algorithms.

2.2.1 Integer Linear Programming

There are many integer linear programming formulations in the literature, but in [10], there is an efficient and exceptionally well-described formulation of constraints and objective functions, which suit underlying timetabling problem of 1.2 very well. The model in [10] even contains definition for consecutive multi-period classes, which are rarely present in the literature. This model is adapted and further analysed in 3.

2.2.2 Constraint Satisfaction over Finite Domains

Another method for solving timetabling problem is constraint satisfaction. In [11], there is a description of such method, but it lacks multi-period classes, which are essential for our problem, and explanation is not as exhaustive as it is in [10]. Moreover, integer linear programming and constraint satisfaction problem can be polynomially reduced to one another, so it is possible for an integer linear programming formulation to be also tested on constraint satisfaction problem solvers.

2.2.3 Boolean Satisfiability Problem (SAT)

As it is shown in [12], the timetabling problem can also be solved by using SAT and MaxSAT (explained in appendix B). It is generally known, that SAT solvers are very powerful tools, especially in comparison with CSP solvers. Article [12] supports this statement with exceptional results of the proposed method: Out of 32 standard benchmark instances derived from the Second international timetabling competition held in 2007, techniques proposed in [12] yielded the best known solutions for 21 instances, while improving the previously best known solution for 9 of them, which is impressive.

Sadly, the propositional logic operates with low expressivity and some high-order constructs, such as cardinality constraints, cause SAT formulation to grow exponentially. For example, defining a constraint "at most 2 of 5 variables must hold" will translate to 10 CNF clauses. For the instances we are dealing with, the number of clauses shall drain memory very fast.

Moreover, authors of [12] have developed their own SAT and MaxSAT solvers (called Barcelogic), which are more suitable for the timetabling problem than conventional solvers (such as MiniSAT).

2.2.4 Reduction to Graph Colouring

As it is shown in proof 1.2.1.19, a graph colouring problem can be polynomially reduced to a timetabling problem (making it NP-Complete). Article [13] shows, how a problem of finding a feasible timetable can be reduced to a problem of finding chromatic number (i.e. minimal possible number of colours needed to succesfully colour a graph) of a graph. An exact (and rather easy) algorithm called CHROMA is proposed, along with its alteration, which can even minimise daily c_{max}. However, constraint specification requires severe changes in the algorithm itself, making it non-flexible and unusable for our problem.

2.2.5 Genetic Algorithms

Genetic algorithms (explained in appendix B) are obvious choice for their robustness and previous successes in solving highly constrained problems. There is a vast amount of articles describing various genetic algorithms solving various timetabling problems. Some of them are described in [14] (page 9-11). Most of these differ only in a choice of genetic operators (according to the timetabling problems which they are supposed to solve). Some of the genetic algorithms use exotic representations (such as [15], in which an individual is represented as a sequence of quantum bits), others use sophisticated population models or inner local search procedures. Generally, all randomised search–based methods are suitable for the timetabling problem, especially because of the possibility to choose nearly any fitness function to be optimised.

2.2.6 Simulated Annealing

Simulated annealing (explained in appendix B) is a notable example of a randomised search–based method, which is known for it's ability to overcome local extrema. Articles [16] and [17] describe such methods. One of the advantages of simulated annealing is a fact, that the search itself can begin from any solution, and so it is suitable for interactive timetabling (where manual schedule corrections are expected to be made from time to time).

2.2.7 Tabu Search

Tabu search is so far the most successful heuristic method for solving the timetabling problem (namely the algorithm proposed by Schaerf in [13]).

Chapter 3
Integer Linear Programming

3.1 Method Description

The Integer Linear Programming (ILP) is a mathematical programming method intended to solve problems of the following form: Given matrix $A \in \mathbb{R}^{m \times n}$ and vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, we search for a vector $x \in \mathbb{Z}^n$ such that $A \cdot x \leq b$ and c^T is maximal. ILP is NP-Complete.

The fundamental difference between the common linear programming and the ILP is that the variables in the ILP are restricted to integers (sometimes both real and integral numbers are allowed and then it is called Mixed Integer Programming). Restriction to integral numbers allows us to define many practical problems as ILP and solve them using powerful industrial solvers (for instance IBM CPLEX).

There are several methods to solve ILP, such as *enumeration*, *branch and bound* or *cutting planes*, however the solvers used in this thesis use only the *branch and bound* method.

3.2 Formal Definition

There are two common ILP models of the timetabling problem: A time-indexed model and a relative-indexed one. In the time-indexed model, the time axis of each day is discretised to equidistant atomic time intervals of given length, whereas in the relative-indexed model, only a relative ordering of courses and their durations are specified. The advantage of the time-indexed model is that uniqueness constraints (one teacher teaches only one course at certain time) can be expressed trivially, because of native time-alignment of the model. Thus we shall proceed with the time indexed model.

3.2.1 Variables

Two sets of binary variables are adopted. The main variable $x_{stu,sub,tea,day,per}$, which specifies whether the student *stu* has the subject *sub*, which is taught by the teacher *tea* on day *day* and time period *per* of that day. The auxiliary variable $y_{stu,sub,tea,day}$ specifies whether the student *stu* has the subject *sub*, which is taught by the teacher *tea* on day *day*.

3.2.2 Features

The problem features are divided into the following basic sets:

- I – set of days
- J – set of periods
- K – set of students
- L – set of teachers
- M – set of subjects

and these basic sets are also divided into more specific subset classes, which may help with reducing the size of the model:

- L_k – set of teachers teaching at least one subject for the student k
- L_i – set of teachers available at day i
- L_m – set of teachers teaching the subject m
- M_l – set of courses taught by the teacher l

There is also the relation $PREC \subset M^2$, which specifies precedence relation on the subjects, function $curriculum : K \to \mathcal{P}(M)$ specifiing subjects of student's curriculum, function $volume : (K, M) \to \mathbb{R}$ specifiing desired volume of the curriculum subject. There is also a function specifiing availability of teachers: $available1 : L \to \mathcal{P}(I)$, $available2 : L \to \mathcal{P}(J)$, a function specifiing duration of subjects: $duration : M \to \mathbb{N}$, and a function specifiing teacher proficiency: $proficiency : L \to \mathcal{P}(M)$.

3.2.3 Constraints

C1. Time consistency (teachers): On every day and every period, every teacher teaches at most one student in one subject (the capacity of the courses is treated elsewhere):

$$\forall i \in I, \forall j \in J, \forall l \in L : \sum_{k \in K} \sum_{m \in M} x_{i,j,k,l,m} \leq 1 \tag{1}$$

C2. Time consistency (students): On every day and every period, every student attends at most one subject with one teacher:

$$\forall i \in I, \forall j \in J, \forall k \in K : \sum_{l \in L} \sum_{m \in M} x_{i,j,k,l,m} \leq 1 \tag{2}$$

C3. Subject volume: Every student k needs his curriculum subjects m to be scheduled exactly at the volume $volume(k, m)$ he needs. Additionally, no student may study a subject, which he does not have in his curriculum. Formally: For every student k and every subject m, the sum of variable x over teachers l who teach the subject m, over days i and period of days j, must be either the $volume(k, m)$ (if the subject m is the curriculum subject for the student k) or 0 (otherwise).

$$\forall k \in K, \forall m \in curriculum(K) : \sum_{l \in L_m} \sum_{i \in I} \sum_{j \in J} x_{i,j,k,l,m} = volume(k, m) \tag{3}$$

$$\forall k \in K, \forall m \notin curriculum(K) : \sum_{l \in L_m} \sum_{i \in I} \sum_{j \in J} x_{i,j,k,l,m} = 0 \tag{4}$$

C4. Teacher availability: No teacher may teach anything when he is not available:

$$\forall m \in M, \forall l \in L_m, \forall i \in available1(l), \forall k \in K, \forall j \in available2(l) : x_{i,j,k,l,m} \leq 0 \tag{5}$$

C5. Subject completeness: Whenever a subject is scheduled in a certain day, it must be scheduled fully (scheduled units must equal subject duration).

$$\forall k \in K, \forall i \in I, \forall l \in L_i, \forall m \in proficiency(l) :$$

$$(\sum_{j \in J} x_{i,j,k,l,m}) - duration(m) \cdot y_{i,k,l,m} = 0 \tag{6}$$

C6. No repetitions in one day: Whenever a subject is scheduled in a certain day, it must not be scheduled again in that day.

$$\forall k \in K, \forall m \in M, \forall i \in I : \sum_{l \in L_m} \sum_{j \in J} x_{i,j,k,l,m} \leq duration(m) \tag{7}$$

C7. Consecutiveness: Whenever a subject is scheduled in a certain day, its periods must be scheduled consecutivelly.

$$\forall k \in K, \forall m \in M, \forall l \in L_m, \forall i \in I, \forall t \in J : x_{i,j,k,l,m} - x_{i,t,k,l,m} \leq 0 \tag{8}$$

$$\forall k \in K, \forall m \in M, \forall l \in L_m, \forall i \in I, \forall j \in J, \forall t \in J \setminus \{0, 1\} :$$

$$-x_{i,j,k,l,m} + x_{i,j+1,k,l,m} - x_{i,j+t,k,l,m} \leq 0 \tag{9}$$

C8. Precedence: Whenever subjects a and b are scheduled in a certain day and subject b is supposed to go after subject a, subject b must be scheduled after subject a or must not be scheduled at all.

$$\forall k \in K, \forall i \in I, \forall m_a \in M, \forall j_a \in J, \forall m_b \in M, \forall j_b \in B, \forall l_a \in L, \forall l_b \in L :$$

$$j_a \cdot x_{i,j_a,k,l_a,m_a} - j_b \cdot x_{i,j_b,k,l_b,m_b} + j_a \cdot x_{i,j_b,k,l_b,m_b} \leq j_a \tag{10}$$

3.2.4 Objective Function

O1. Shortest timetable in terms of days: Minimize quadratic penalty function for days.

$$\sum_{i \in I} \sum_{j \in J} \sum_{L \in L_i} \sum_{k \in K} \sum_{m \in M_l} i^2 \cdot x_{i,j,k,l,m} \tag{11}$$

O2. Shortest timetable in terms of periods: Minimize quadratic penalty function for periods.

$$\sum_{i \in I} \sum_{j \in J} \sum_{L \in L_i} \sum_{k \in K} \sum_{m \in M_l} j^2 \cdot x_{i,j,k,l,m} \tag{12}$$

O3. Shortest timetable in terms of both days and periods: Minimize quadratic penalty function for both days and periods.

$$O1 + O2 \tag{13}$$

3.3 Platform Selection

In order to satisfy the request for free ILP solving software, the following three prominent solvers shall be compared in terms of performance (hadrware and software environment used for benchmarking is described in appendix A).

- GNU Linear Programming Kit GLPSOL[1])
- LPSolve[2])
- COIN-OR CBC[3])

[1]) `http://www.gnu.org/software/glpk/`
[2]) `http://lpsolve.sourceforge.net/5.5/`
[3]) `https://projects.coin-or.org/Cbc`

3.3.1 Preliminary Benchmarking

In order to select the most suitable ILP solver of the three above mentioned, a small instance of the described problem was created. The parameters of this instance are the following:

- 20 teachers, each having proficiency for exactly 1 subject
- 20 days horizon
- 10 periods (time slots) per day
- 4 courses (3 having duration 2, 1 having duration 3)
- 2 students (one actual, second representing next one to arrive)
- Student 1 has 4 subjects in his curriculum, student 2 has 2.
- 2 precedence rules
- Curriculum subject volumes are {12, 20, 20, 20}, {12, 10} respectively
- Lunch break for teachers is scheduled to period 7
- Objective function O1 is used

Solver (obj: O1)	Runtime [s] (— means over 300)				
Constraints enabled up to	C4	C5	C6	C7	C8
GLPSOL 4.52	0.20	—	1.40	7.10	—
LPSolve 5.5.2.0	0.54	—	0.81	—	—
CBC 2.7.5	0.48	—	1.66	5.95	50.66

Table 3.1. Preliminary solver benchmark (objective O1)

Solver (obj: ZERO)	Runtime [s]				
Constraints enabled up to	C4	C5	C6	C7	C8
GLPSOL 4.52	0.20	15.90	1.10	7.70	58.20
LPSolve 5.5.2.0	0.56	8.54	1.23	3.90	64.84
CBC 2.7.5	0.46	2.56	3.12	10.27	48.61

Table 3.2. Preliminary solver benchmark (only searching for feasible)

	Constraints enabled up to				
	C4	C5	C6	C7	C8
times C4 [1]	1.00	1.27	1.59	2.40	17.21

Table 3.3. Model size

In the table 2.1., we can see that CBC outperforms GLPSOL and LPSolve (for the specified problem), so we shall continue with further analysis of the model with CBC.

	Precedence constraints				
	Disabled	1 simple	2 simple	2 chained	3 chained
Runtime [s]	3.97	34.62	37.83	50.36	52.79

Table 3.4. Precedence constraint (C8) significance

	Precedence constraints	
	Disabled	2 simple
O1 Runtime [s]	3.97	37.83
O2 Runtime [s]	2.78	—
O3 Runtime [s]	17.44	—

Table 3.5. Objective function significance

	Subject length [periods]		
Distinct subjects	**1**	**2**	**3**
1	0.19	0.94	1.42
2	0.21	0.55	2.00
3	0.25	1.50	3.38
4	0.29	1.97	6.60
5	0.34	2.75	43.32
6	0.39	3.16	—
7	0.42	7.36	—

Table 3.6. Subject length significance (runtimes in seconds) with C8 disabled

	Subject length [periods]		
Distinct subjects	**1**	**2**	**3**
2 (1 prec. rel.)	19.83	21.04	19.52
4 (2 prec. rel.)	26.94	28.78	58.78
6 (3 prec. rel.)	41.75	36.06	—

Table 3.7. Subject length significance (runtimes in seconds) with C8 enabled

	Precedence constraints	
Subject volume	**disabled**	**2 simple**
12 (x7 subjects)	6.72	80.24
24 (x7 subjects)	7.34	228.48
36 (x7 subjects)	13.08	—

Table 3.8. Curriculum subjects volume significance (runtimes in seconds)

3.4 Implementation

There are several ways of implementing programatical interfaces to the solvers:

- Using wrappers for solver libraries
- Using third party solver frameworks
- Using custom solver framework

3.4.1 Using Wrappers for Solver Libraries

C# wrappers are managed libraries which provide entry points for functions in unmanaged libraries. They are not intuitive, nor easy to use, as they need to be provided with raw ILP matrices and vectors (in sparse forms). Construction of the matrices and vectors is a very time-consuming process, especially with complex constraints (like the ones our problem has).

3.4.2 Using Third Party Solver Frameworks

Third party solver frameworks provide strongly typed and intuitive mechanisms of constraint definition. There are two most prominent frameworks—Microsoft Solver Foundation and Google OR.

Microsoft Solver Foundation is a commercial product with limited functionality trial version (limited number of variables), but provided there's a plugin, which adapts it to the solver we need, it can become a very powerful tool. For instance, the following listing shows, how variables (in MS Solver Foundation they are called *decisions*) are defined:

```
SolverContext context = SolverContext.GetContext();
Model model = context.CreateModel();

Dictionary<Subject, Dictionary<Teacher, Dictionary<Student,
  Dictionary<int, Dictionary<int, Decision>>>>> x;
x = new Dictionary<Subject, Dictionary<Teacher, Dictionary<Student,
  Dictionary<int, Dictionary<int, Decision>>>>();

foreach (Subject subject in M)
{
  x[subject] = new Dictionary<Teacher, Dictionary<Student,
    Dictionary<int, Dictionary<int, Decision>>>();
  foreach (Teacher teacher in L)
  {
    x[subject][teacher] = new Dictionary<Student,
      Dictionary<int, Dictionary<int, Decision>>>();
    foreach (Student student in K)
    {
      x[subject][teacher][student] = new Dictionary<int,
        Dictionary<int, Decision>>();
      for (int period = 0; period < Problem.periods; period++)
      {
        x[subject][teacher][student][period] = new Dictionary<Int32,
          Decision>();
        for (int day = 0; day < Problem.days; day++)
        {
          Decision d = new Decision(Domain.Boolean, "SUB" +
            subject.number + "_TEA" + teacher.number + "_STU" +
              student.number + "_PER" + period + "_DAY" + day);
          x[subject][teacher][student][period][day] = d;
          model.AddDecision(d);
        }
      }
    }
  }
}
```

We create a solver context, from which we obtain a model and then we add boolean variables to it according to our theoretical model (defined in 3.2), i.e. we have one variable for every student-subject-teacher-day-period. These variables are stored in a system of associative arrays, in order to fetch result from them later in the code.

In the next listing, we show how constraints are defined.

```
// CONSTRAINT UNIQ 1
int cnt = 0;
for (int day = 0; day < Problem.days; day++)
{
  for (int period = 0; period < Problem.periods; period++)
  {
    foreach (Teacher teacher in L)
    {
      SumTermBuilder sum = new SumTermBuilder(1000);
      // SUM BEGINS
      foreach (Student student in K)
      {
        foreach (Subject subject in M)
        {
          sum.Add(x[subject][teacher][student][period][day]);
        }
      }
      // SUM ENDS
      model.AddConstraints("UNIQ1_" + cnt, sum.ToTerm() <= 1);
      cnt++;
    }
  }
}
```

The constraints have a form of equations, in which we use the very variables we defined before. In the listing above, we programatically define constraint 3.2.3.C1.

Objective functions are defined likewise:

```
SumTermBuilder objective1 = new SumTermBuilder(10000);
for (int day = 0; day < Problem.days; day++)
{
  for (int period = 0; period < Problem.periods; period++)
  {
    foreach (Teacher teacher in L)
    {
      foreach (Student student in K)
      {
        foreach (Subject subject in M)
        {
          objective1.Add(Problem.penalty_day(day) *
            x[subject][teacher][student][period][day]);
        }
      }
    }
  }
}

model.AddGoal("objective", GoalKind.Minimize, objective1.ToTerm());
```

Then the model is solved and the results are obtained.

Google OR provides basically the same functionality as MS Solver Foundation, but it is free. Major drawback of both Google OR and MSSF is that they only support a few solvers and they do not allow model export. Another drawback that was found

during testing is that there's a bug in DLL version of COIN-OR CBC, which results in significant performance loss, and because the frameworks are unable to use standalone version of CBC, this loss affects them.

3.4.3 Using Custom Solver Framework

Due to the drawbacks of the third party frameworks mentioned in the previous section, there was a need of creating a custom framework, which would be easy to use, would provide export functionality and would overcome the DLL-related performance loss of CBC. For this purpose, a custom framework was created, which allows following:

- Intuitive and easy definition of variables, constraints and objective functions
- Export to .lp file format
- Ability to call solver standalones on .lp files

3.5 Conclusion on ILP

Due to computational unsustainability of ILP even on extremely small timetabling instances, it will not be used in the final software. Instead of ILP, we shall focus on more conventional and straightforward search methods.

Chapter 4
Proposition of a Solution

Due to conclusion 3.5, a novel approach to solving problem 1.2 shall be proposed. This approach will make use of the search algorithms mentioned later in this thesis (namely chapters 6 and 7); in this chapter, a principle of the proposed meta-algorithm will be explained and the software object model (which will later be used in part 12, concerning testing) will be described.

4.1 The Meta-algorithm

The algorithm we are speaking of, is, strictly speaking, a meta-algorithm. A meta-algorithm is an algorithm designed to manipulate other algorithms which are embedded inside it. This meta-algorithm is designed to manipulate various inner search algorithms, such as local search algorithms and genetic algorithms.

4.1.1 Student Arrival Predictions

As it was stated in 1.3, a certain information about forthcoming students is needed for the meta-algorithm to perform well. In case of the proposed meta-algorithm, this information is a prediction based on previously arrived students, which consists of:

- Estimated number of students to arrive
- Days of arrival of these students
- Estimated curricula of these students
- Estimated subject volumes of these students

The estimation itself is not a part of the meta-algorithm and has to be known *a priori*. For the testing purpose, the proposed meta-algorithm works with randomly generated so called *scenarios*, which are basically randomly generated (fake) estimations. Scenarios also include another information which is vital for the meta-algorithm, such as:

- Number of days
- Number of periods (time slots) of a day
- Number of teachers
- Proficiencies of specific teachers
- Available periods of specific teachers
- List of subjects and their lengths

4.1.2 Lifecycle of a Student

A student enters the meta-algorithm in the state of *prediction* (as seen in figure 4.1). In this state, the student is *subjected to scheduling*, so the meta-algorithm is allowed to reschedule his or her lessons to another day and time, or to change teachers of his or her curriculum subjects. This rescheduling, of course, may only move subjects from the present to the future or vice versa. Moving subjects to the past is inconsistent with the nature of prediction.

In all moments, all the subjects from student's curriculum are scheduled in required volume, despite that there may not be a way to schedule them without inconsistency.

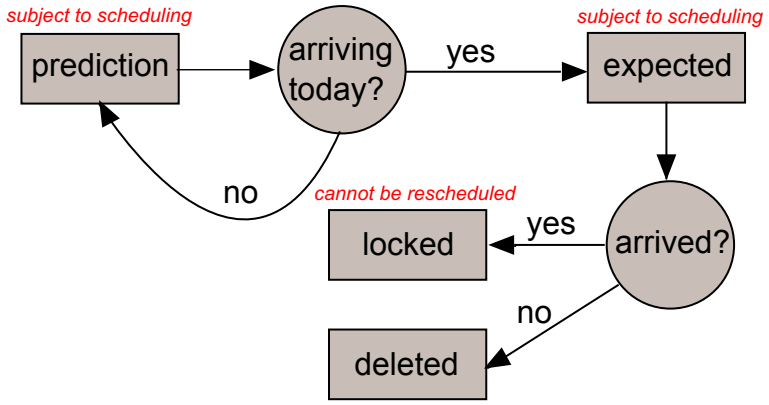


Figure 4.1. Lifecycle of a student

When a predicted day of arrival of a certain student matches the current day, this student's state is then changed to *expected*. At this point of the algorithm, a user needs to provide reflection of a real state to the algorithm—to tell whether a student has actually arrived or whether he or she has not.

A student who has actually arrived is then transfered to *locked* state, which means, that perturbation operators will not be able to select and perturb his or her lessons anymore. Once a student is locked, his timetable becomes *immutable*.

A student (prediction) who has not arrived must be deleted from the timetable.

Transition to a next day is not allowed until all students *expected* to arrive on the actual day have been *locked* or *deleted*. If the transition to a next day was allowed with students in *expected* state, there would exist a chance, that some of the lessons of the *expected* students would be left in the past, with perturbation operators unable to reposition then (because perturbation operators are not allowed to alter the past).

4.1.3 Common Properties of Perturbation Operators

Although, there are many possibilities of how to perturb a timetable, all such operators must share certain common properties (given by this scheduling meta-algorithm); the following must hold:

- Perturbation operators do not alter lessons of locked students.
- Perturbation operators do not operate on the past.

If perturbation operators were allowed to operate on the past, there would be a non-zero probability of never reaching a consistent timetable, which is not acceptable.

4.1.4 Common Properties of Randomisation Functors

Randomisation functors are used to generate random (more or less) timetables, which are then used as intial solutions for various search methods. These functors must also share certain properties:

- Randomisation operators do not alter lessons of locked students.
- Randomisation functors do not place student's lessons before their (student's) days of arrival.

If randomisation functors were allowed to place student's lessons before their days of arrival, perturbation operators would be unable to relocate the lessons assigned to the past (by randomisation functors).

4.1.5 General Flow of the Meta-algorithm

The general flow of this meta-algorithm is depicted in the figure 4.2

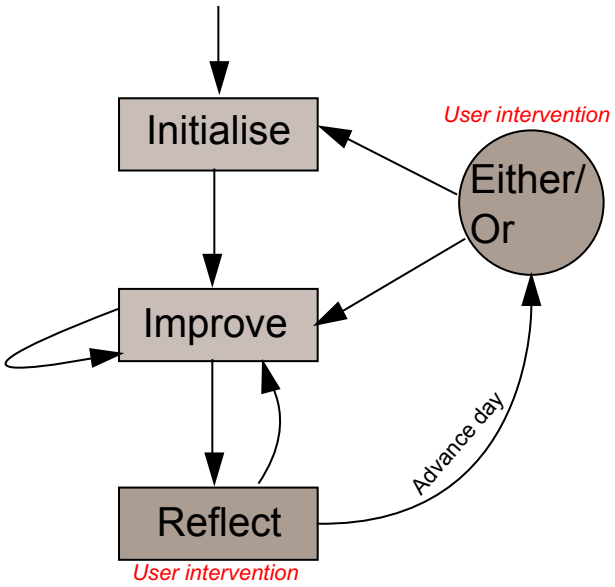


Figure 4.2. General flow of the meta-algorithm

In the first stage, the algorithm is *initialised* (the initialisation procedure itself is controlled by the inner search algorithm—for instance when using a local search, this may mean just a plain randomisation or some form of informed initialisation, and when we use a genetic algorithm, the initialisation stands for generation of an initial population, may it be random or informed).

After the initialisation stage, there is a stage of *solution improvement*. Again, the procedure itself is solely in hands of the inner search algorithm. Termination state of this stage is not defined—it happens until certain number of iterations have been reached, but the user can decide to continue improving until the solution is good enough.

Upon emergence of a temporaly-dependent information which was unknown before (arrival or non-arrival of a student expected to arrive on the actual day), a user can trigger a transition to the next state called the *reflection* state. In the reflection state, the user is supposed to provide this new information to the algorithm, so the user either turns a predicted student into an actual one (which leads to its *lock-off*) or delete the predicted student.

After the reflection, the user may continue with improvement of a current solution or, in the case all the predicted students are either locked or deleted, advance the current day.

After advancing to a next day, the previous day becomes the past, which is no more subjected to scheduling. The user now may decide between re-randomizing the solution (with keeping settings of all the locked students, of course) and improving the current one.

4.2 The Implementation

In compliance with the assignment, an object model was designed and implemented with accordance to the *SOLID* principles of object programming and design, which lead to good maintainability, extensibility and testability of a code and which will now be briefly explained.

Single Responsibility Principle

This principle states that a class should only serve one purpose and all its functionality should be aligned with that purpose. The term was coined by Robert C. Martin in his book *Agile Software Development, Principles, Patterns, and Practices* [18]. In this book, Martin states that a class should only have one reason to change and he gives an example of a class, which represents a report, but it also has a function which prints the report on a standard output. Assuming that responsibility of the class is to hold a report data, it should only change when the strucure of the data has changed. But in this case, it would also have to change when there is a change in the output method, so, clearly, the single responsibility principle does not hold for this class.

Open/Closed Principle

This principle, coined by Bertand Meyer in his book *Object Oriented Software Construction*[19] states that a class should be closed for modification, but opened for extension. In practice, this means, that a modification of a class should not be accomplished through modification of the code of the class itself, but rather by extension of the class using inheritance.

Liskov Substitution Principle

This principle, formulated by Barbara Liskov in [20], is considered a cornerstone of object modelling:

- *If S is a subtype of T, then objects of type T may be substituted with objects of type S without altering any of the desirable properties of that program.*

This principle is inherent to the most of modern object-oriented programming languages (in our case to *C#*).

Interface Segregation Principle

This principle, formulated by Martin in [18], states that a class should not depend on a fuctionality it does not use. For instance, when a class if forced to implement an interface from which it will only use one method, the rest of the methods still must be implemented, though they will not be used (which violates the interface segregation principle). By using this principle, we are forced to write specific interfaces instead of general ones.

Dependency Inversion Principle

This principle, also formulated by Martin in [18], states that:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

In practice it means, that a class should not depend on concrete types, but rather on abstract ones. This also goes hand in hand with the single responsibility principle, because such class would be unable to instantiate its object components by itself (it is not its responsibility), beacuse that would make it dependent on concrete (instantiable) data types. The responsibility of instantating is delegated to a separate class, sometimes called an IoC (inversion of control) container.

Because correct and generic implementation of this principle is non-trivial, this principle might be the most violated one, but there are several libraries (such as *Unity*, *Autofac* and *SimpleInjector*), which provide quality implementation of IoC and make observance of this principle very straightforward.

4.2.1 Individual Representation

The form of individual representation has significant impact on performance of individual algorithms and it must be designed in a way which would allow fast and efficient computation of its objective function value. The form must also allow fast and efficient perturbation. There are several forms of representation which were considered:

- Full-form multidimensional matrix
- Sparse-form multidimensional matrix
- Custom associative array

Full-form Multidimensional Matrix

This representation basically corresponds to the ILP boolean variable $x_{stu,sub,tea,day,per}$ from the section 3.2.1, which specifies whether the student *stu* has the subject *sub*, which is taught by the teacher *tea* on day *day* and time period *per* of that day.

Advantage of this form is its extraordinary simplicity, be it implemented as an actual array or as a some form of dynamic array, which can be expanded over time, and constant worst-time complexity of random access.

Main drawback of this form is that with every additional dimension, the matrix gets exponentially sparser and exponentially more difficult (in terms of time and memory) to operate with.

Sparse-form Multidimensional Matrix

In this representation, only non-zero elements of the matrix are stored along with their coordinates. There are three prevalent methods of storing sparse matrices:

- Dictionary of Keys: This method uses an associative array to map coordinates to their values. It has amortized constant time complexity of random access and amortized linear time complexity of search, but there is no way to efficiently perturb a timetable stored in this form.
- Coordinate List: A sorted list, which stores coordinates along with their values. This method has $O(log(n))$ time complexity for random access and $O(n)$ for search.

Custom Associative Array

This form of representation was created especially for the problem of our concern. It exploits some specific features of the problem:

- Curriculum of a student cannot change during computation: This means that instead of capturing every possible student-to-subject assignment combination, we may conserve memory by storing only a list of student-subject pairs. And because these pairs never change during computation, we can use them as keys in an associative array.
- Only things that change during computation are teachers and times assigned to student-subject pairs: This means, that teacher-time pairs can be used as values of an associative array (indexed by the student-subject pairs mentioned above).

- Duration of subjects also does not change during computation: This means that collisions in the timetable can be detected in linear time using just simple arithmetics.
- Days always have the same amount of periods (slots): This means that we don't have to physically separate days, because simple one-dimensional encoding of two dimensional data can be used. This significantly simplifies complexity of perturbation. For instance, in figure 4.3 the following holds:

$$R_i = \left\lceil \frac{X_i}{W} \right\rceil$$

$$L_i = mod(X_i - 1, W) + 1$$

or for index base zero:

$$R_i = \left\lfloor \frac{X_i}{W} \right\rfloor$$

$$L_i = mod(X_i, W)$$

where R_i is a row index corresponding to item X_i, C_i is a column index corresponding to item X_i and W is width of a row (in our case number of periods in a single day).

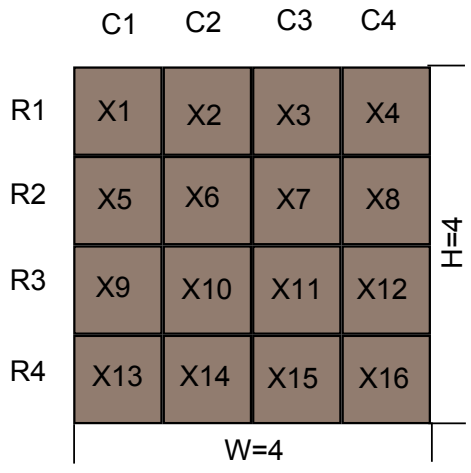


Figure 4.3. 1D encoding of 2D data

By exploiting these features, we can spare ourselves writing computationally inefficient procedures for obtaining objective function values or for perturbations.

Structure of the representation is shown in figure 4.4.

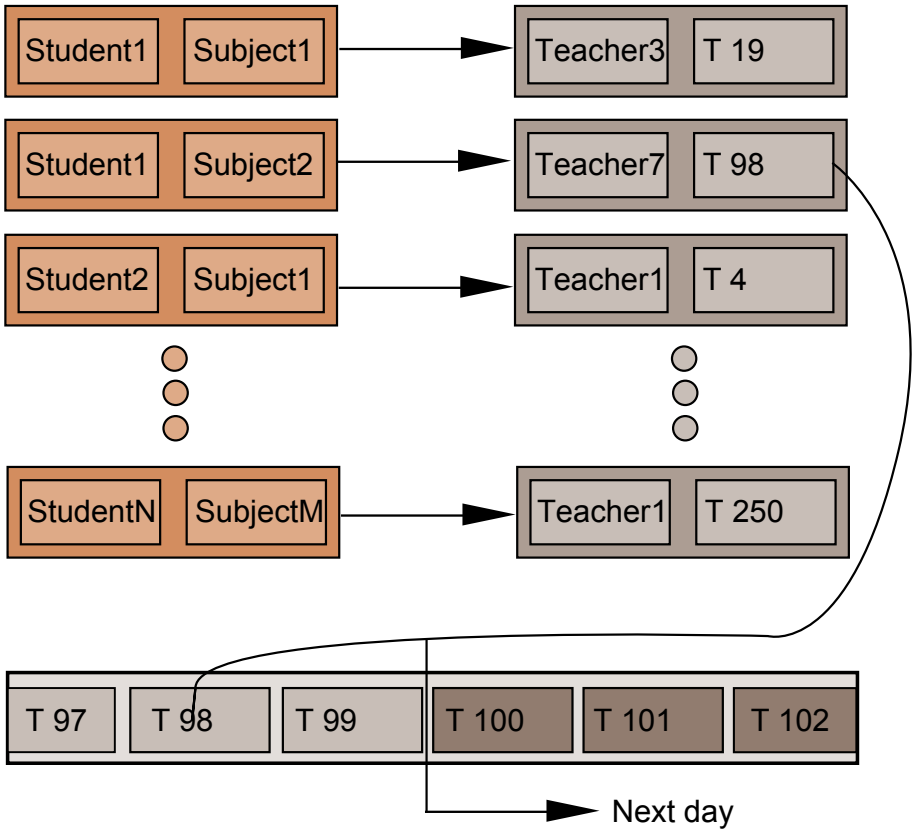


Figure 4.4. Custom associative array

Clearly, this form of representation suits our needs the best.

Implementation

Regarding the implementation, we need several classes to implement this data structure, namely:

- **class Student**, which represents a single student. Fields and properties of this class are the following:

```
public int number; // ID of the student
public string name; // Name of the student
public int arrival; // Day of arrival
public HashSet<Subject> curriculum; // Student's curriculum
public Dictionary<Subject, int> volume; // Subject volume
public bool Locked { get; set; } // Lock status
```

  Student's curriculum is a set of all the subjects student wants to enroll (where *Subject volume* specifies how many lessons of these subjects he wants to take). *Lock status* tells whether a timetable of a student can or cannot change (i.e. whether the student is a prediction or whether he is actually scheduled).

- **class Subject**, which represents a single subject, with the following fields:

```
public int number; // ID of the subject
public string name; // Name of the subject
public string abbr; // Abbreviation of the name
public int units; // How many periods the subject takes
```

When these classes are defined, we can define an aggregation class which would serve as our hashmap's key type:

- **class StudentSubject**, with these properties:

```
public Student Student { get; set; } // Involved student
public Subject Subject { get; set; } // Involved subject
public int VolumeIdx { get; set; } // Volume index
public bool Locked { get; set; } // Lock status
```

This class aggregates a single student and a single subject, creating an object representation of a single lesson of that subject. Student usually wants to take more lessons of the same subject, so there is a *Volume index* property, which specifies which particular lesson is represented by this object.

- **class Teacher**, which represents a single teacher along with properties that characterize him or her. Fields and properties are the following:

```
public int number; // ID of the teacher
public string name; // Name of the teacher
public HashSet<Subject> proficiency; // Teacher's proficiency
// Set of unavailable periods for each day
public Dictionary<int, HashSet<int>> units_unavailable;
```

Teacher's proficiency is a property, which specifies subjects that a teacher can teach. `units_unavailable` maps days to set of units, on which a teacher cannot teach (for that particular day).

Having defined class Teacher, we can define an aggregation class, which links teachers to time periods:

- **class TeacherTime**, with the following properties:

```
public Teacher Teacher { get; set; }
public int Time { get; set; }
```

Now a final data structure can be implemented as follows:

```
public Dictionary<StudentSubject, TeacherTime> Table { get; set; }
```

4.2.2 The Problem Context

In order to create a scalable design, a concept of so called *problem context* was introduced. A problem context is a data structure, whose purpose is to:

- Provide data
- Provide methods for data alteration
- Hold the temporal context (i.e. current day)
- Define functors for inner algorithms to use:
  - Randomisation functor (which provides methods for randomizing individuals or their concrete parts)
  - Fitness functor (which specifies an objective function)
  - Mutation and recombination functors (for population based inner search algorithms)
  - Position and teacher mutation functors (which specify perturbation functions for position a teacher perturbation, respectively)

- Auxiliary variables

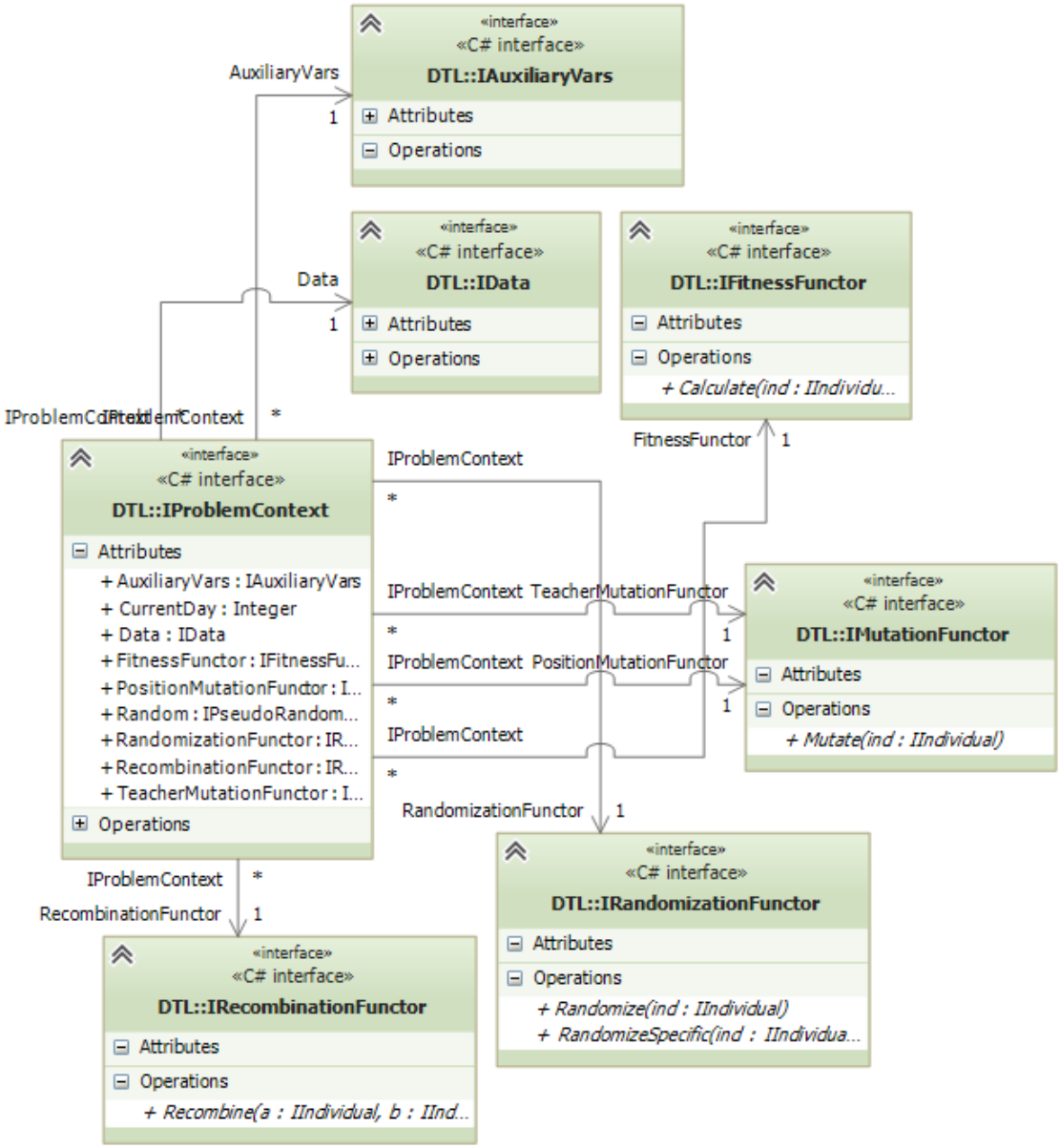


Figure 4.5. UML Class diagram of `DTL::ProblemContext`

Auxiliary variables is a concept, which creates a compromise between good design and computational performance. In a well designed system, every search algorithm would be responsible for creation and management of its vital temporary variables, but in our case, this would mean a significant performance loss. This loss is due to extremely frequent calls to `IFitnessFunctor.Calculate()`. Upon every call, this function would need to allocate large heap space for these variables, compute the objective function and dispose these variables for garbage collection just to reallocate them in the next call, which follows almost immediately. This constant allocation and disposal of large memory chunks presents an unacceptable performance loss.

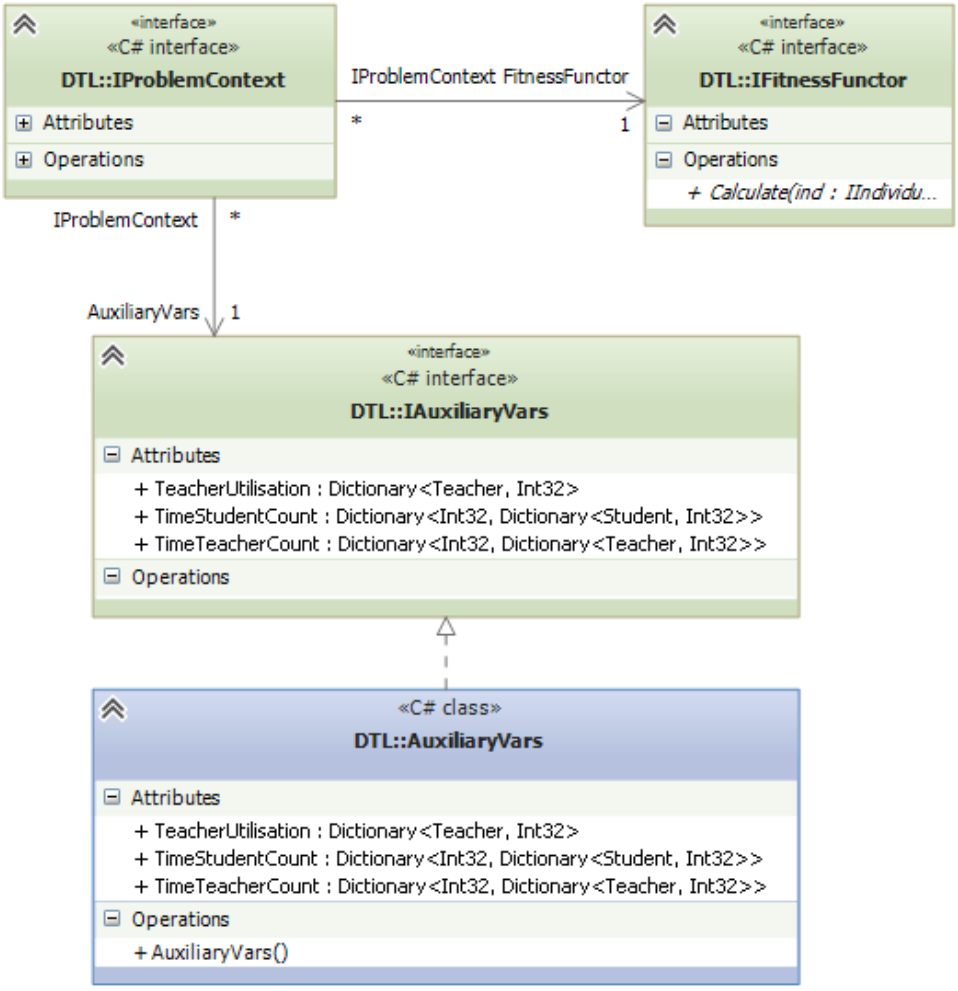


Figure 4.6. UML Class diagram of `DTL::AuxiliaryVars`

In the concept of *auxiliary variables* as presented here, these variables are allocated only once and then constantly reused by the fitness functor. Major drawback of this solution is that there is no way to compute objective functions parallelly in a single context. On the other side, on a single processor, this concept leads to a significant increase in performance.

A solution that performs the best in terms of performance, is to use static variables with no object hierarchy at all. Despite its good performance, it will not be used, because it is not well scalable nor testable.

4.2.3 Functors

A multitude of functions which share the same signature and return type, is a reason to use an abstraction. In C#, there are several ways of representing structures which act as pointers to funtions (i.e. they allow functions to be treated as variables, for instance to be passed as arguments, stored as class members etc.). Whereas traditional C-style function pointers are not object oriented, not type-safe and not secure, C# alternatives are.

First way of doing so, is by using **delegates**—a C# native feature. A delegate is a type, which encapsulates a method. A new delegate type is defined by specifiing signature and return type of an encapsulated method. For instance, for a delegate type

which encapsulates methods with signature `(double, double)` and with `double` return type, the definition would be:

```
public delegate double FitnDelegate(double param1, double param2);
```

Then there would have to be some functions to choose from:

```
public static class FitnessFunctions
{
    public static double FitnVar1(double a, double b)
    {
        return a * b;
    }

    public static double FitnVar2(double c, double d)
    {
        return Math.Sqrt(c * c + d * d);
    }
}
```

And a class that would make use of it:

```
public class Computer
{
    public FitnDelegate FitnessFunctor { get; private set; }
    public Computer(FitnDelegate fitnessFunctor)
    {
        this.FitnessFunctor = fitnessFunctor;
    }
    public double Compute(double x, double y)
    {
        return this.FitnessFunctor(x, y);
    }
}
```

We can see, that the class `Computer` is separated from the computation process itself through an abstraction. The initialisation may happen through an existing function:

```
Computer c1 = new Computer(FitnessFunctions.FitnVar1);
Computer c2 = new Computer(FitnessFunctions.FitnVar2);

double param1 = 9;
double param2 = 13;

Console.WriteLine(c1.Compute(param1, param2));
Console.WriteLine(c2.Compute(param1, param2));
```

Or through a *lambda function*:

```
Computer c3 = new Computer((x, y) => x * x + x * y + y * y);
Console.WriteLine(c3.Compute(param1, param2));
```

Or through *anonymous function* which allows statements:

```
Computer c3 = new Computer
(
    delegate(double x, double y)
    {
```

```
        Console.WriteLine("x^2 + xy + y^2");
        return x * x + x * y + y * y;
    }
);
Console.WriteLine(c3.Compute(param1, param2));
```

Another way of creating an abstract function are custom delegate objects known as **functors**. Functors is our representation of choice, because they are more versatile. Functors are not *sealed*, which means, that they can be extended *ad infinitum* using inheritance mechanics.

First, we have to define the encapsulated method's signature and return type through an interface:

```
public interface I2DToDFunctor
{
    double Call(double a, double b);
}
```

The interface `I2DToDFunctor` is an interface that all the functions which take two `doubles` and return a `double` use. Every time we define a functor, we define a class that implements a signature interface of this functor's encapsulated function and we specify the concrete behaviour in the implementation of the `Call()` method:

```
public class MultiplicativeFunctor : I2DToDFunctor
{
    public double Call(double a, double b)
    {
        return a * b;
    }
}
```

Then we define a class which will make use of the functor:

```
public class FunctorComputer
{
    public I2DToDFunctor FitnessFunctor { get; private set; }
    public FunctorComputer(I2DToDFunctor fitnessFunctor)
    {
        this.FitnessFunctor = fitnessFunctor;
    }
    public double Compute(double x, double y)
    {
        return this.FitnessFunctor.Call(x, y);
    }
}
```

We see that the only change (when comparing to the variant using delegates) is in a method of calling the delegate/functor. The usage itself is also very similar:

```
FunctorComputer c4 = new FunctorComputer(new MultiplicativeFunctor());

double param1 = 9;
double param2 = 13;

Console.WriteLine(c4.Compute(param1, param2));
```

A problem context class can use one of three pseudo random number generator implementations, which are then used mainly by randomization functors:

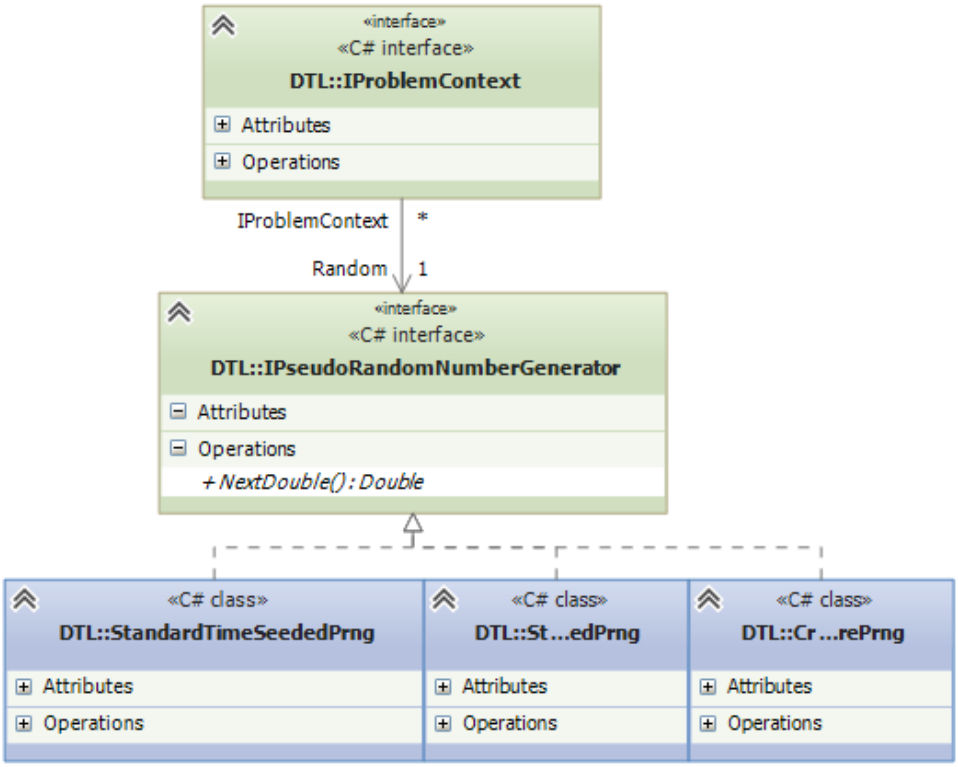


Figure 4.7. UML Class diagram of `DTL::IPseudoRandomNumberGenerator`

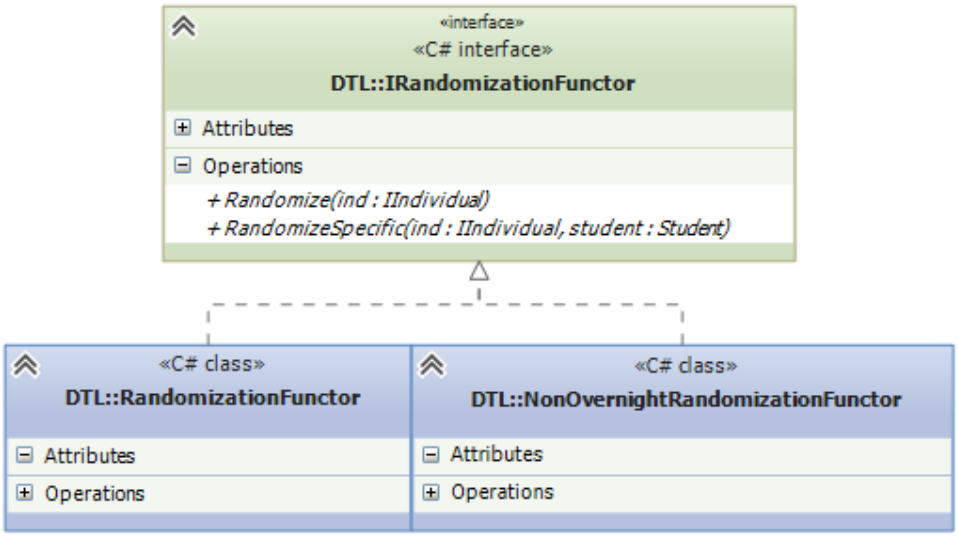


Figure 4.8. UML Class diagram of `DTL::IRandomizationFunctor`

These implementations are:

- **Standard fixed-seed PRNG** – This PRNG is used for testing purposes, for the randomised algorithms always yield the same results when a seed is constant.

- **Standard time-seeded PRNG** – This PRNG is used in the final appliaction and in statistical testing, because the seed is based on system time and therefore, results from algorithms are always different (as intended).
- **Cryptographically secure PRNG** – This PRNG is intended for future use in scenarios, where cryptographical security is preferred over speed (generation of salt, for instance).

Randomization functors

Randomization functors are used to randomize whole individuals or to randomize their parts (single students or teachers).

There are two implementations of `IRandomizationFunctor`:

- **Standard randomization functor** – Assigns completely random time and completely random teacher to every student-subject.
- **Non-overnight randomization functor** – Assigns completely random teacher to every student, but makes sure that assigned random time never crosses boundaries of days (i.e. lessons are not separated by night). In other words, it makes sure that the following equation holds:

$$\left\lfloor \frac{P_n}{N_p} \right\rfloor = \left\lfloor \frac{P_n + N_s}{N_p} \right\rfloor,$$

  where P_n is randomly generated period for a subject to start on, N_p is the number of periods per day and N_s is the number of periods taken by the subject.

The two methods of the interface are meant to do the following:

- `Randomize(IIndividual ind)` – This method is meant to randomize complete individual (i.e. the complete timetable). Every student-subject (apart from locked ones) is assigned a new starting period and a new teacher (with appropriate proficiency).
- `RandomizeSpecific(IIndividual ind, Student student)` – This method is meant to randomize only a student (i.e. the student-subjects that refer to him or her) defined by the second argument. Other students are left untouched.

Mutation and recombination functors

Mutation and recombination functors are used for mutation/perturbation of a single individual and for recombination of a pair of individuals, respectively.

These functors must implement one of the following interfaces, according to their type:

- `IMutationFunctor`
- `IRecombinationFunctor`

Concrete implementations of these interfaces are described in sections 6 and 7. Inner search algorithms typically use two mutation functors—one for altering student-subject positions and one for altering teachers of these student-subjects.

Fitness functors

Fitness functors are used for computation of an objective function value of a specified individual. The concrete implementation is described in section 4.3.

4.2.4 Search Algorithm Object Model

An inner search algorithm of the dynamic timetabling meta-algorithm is represented by the interface type `ISearchAlgorithm`. Because a search algorithm needs to inform external entities about its progression (for visualisation, debugging and for statistical purposes), its model has to implement some form of *observer design pattern*.

```
public interface ISearchAlgorithm
{
    IIndividual Search(IProblemContext ctx);
    IIndividual Improve(IProblemContext ctx, IIndividual ind,
                        int iterations);

    // Signal from external entity to stop the main loop.
    void SignalStop();

    // Signals to registered observers (external entities).
    event EventHandler BetterSolutionFound;
    event EventHandler SearchCompleted;
    event EventHandler StatisticalHit;
}
```

The *observer design pattern* is used, when an object needs to notify some other entities about some changes in its internal state or about some event, but the object itself is not responsible for creating or maintaining these entities (called *observers*). The pattern specifies, that the object, which notifies (called an *observed object*), extends a supertype, which makes external observers able to register on it. Upon the given event, the observed object calls the notification method of the supertype, which notifies all the registered observers about the specified event. All observers are responsible for their registration on the observed object.

In our case, the search algorithm needs to notify, for instance, a statistical module about the best solution after every fitness function call. For this functionality, we use a native C# feature called an *event*. An event is, in fact, a language integrated observer design pattern. In the interface, we specify, that every class that implements it, must be able to send notifications:

```
event EventHandler BetterSolutionFound;
```

Then we create an abstract observer, which will contain methods for registering and unregistering particular event handlers of particular events. In our case, an event handler is represented by a method of a specific signature:

```
public abstract class BetterSolutionEventListener
{
    public void ListenOn(DynamicTimetabling dt)
    {
        if (dt.SearchAlgorithm != null)
        {
            dt.SearchAlgorithm.BetterSolutionFound +=
                     SearchAlgorithm_BetterSolutionFound;
        }
    }

    public void StopListeningOn(DynamicTimetabling dt)
    {
```

```
            dt.SearchAlgorithm.BetterSolutionFound -=
                    SearchAlgorithm_BetterSolutionFound;
        }

        abstract public void SearchAlgorithm_BetterSolutionFound(
                    object sender, EventArgs e);
    }
```

A program, which uses the dynamic timetabling library, then may create an extension of `BetterSolutionEventListener` class to handle the event of discovering a better solution in a specific way:

```
public class StdoutDumpBSEListener : BetterSolutionEventListener
{
    public override void SearchAlgorithm_BetterSolutionFound(
           object sender, EventArgs e)
    {
        // Print solution to stdout.
    }
}
```

And then register the listener to the oberved object:

```
BetterSolutionEventListener bsel = new StdoutDumpBSEListener();
bsel.ListenOn(dt);
```

In the observed object, the event is raised by calling the handler:

```
if (this.SearchCompleted != null)
{
    this.SearchCompleted(best, new IterationEventArgs(iter));
}
```

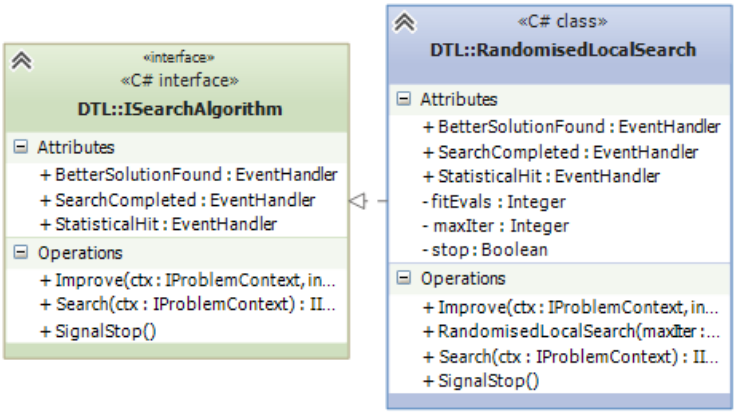


Figure 4.9. UML Class diagram of `DTL::ISearchAlgorithm`

The interface also contains method `SignalStop()`, which is called, for instance, by a visualisation thread, when user demands stopping the algorithm loop immediately (without achieving given number of iterations).

Method `Improve(IProblemContext ctx, IIndividual ind, int n)` improves the provided solution (individual) for another n iterations, and method `Search(IProblem Context ctx)` creates a random solution, which is then improved.

4.2.5 Dynamic Timetabling Object Model

The dynamic timetabling meta-algorithm itself is represented by the class `DynamicTimetabling`. This class implements no interface, because it is represents the top-layer access point to the library and is not supposed to be extended in any way.

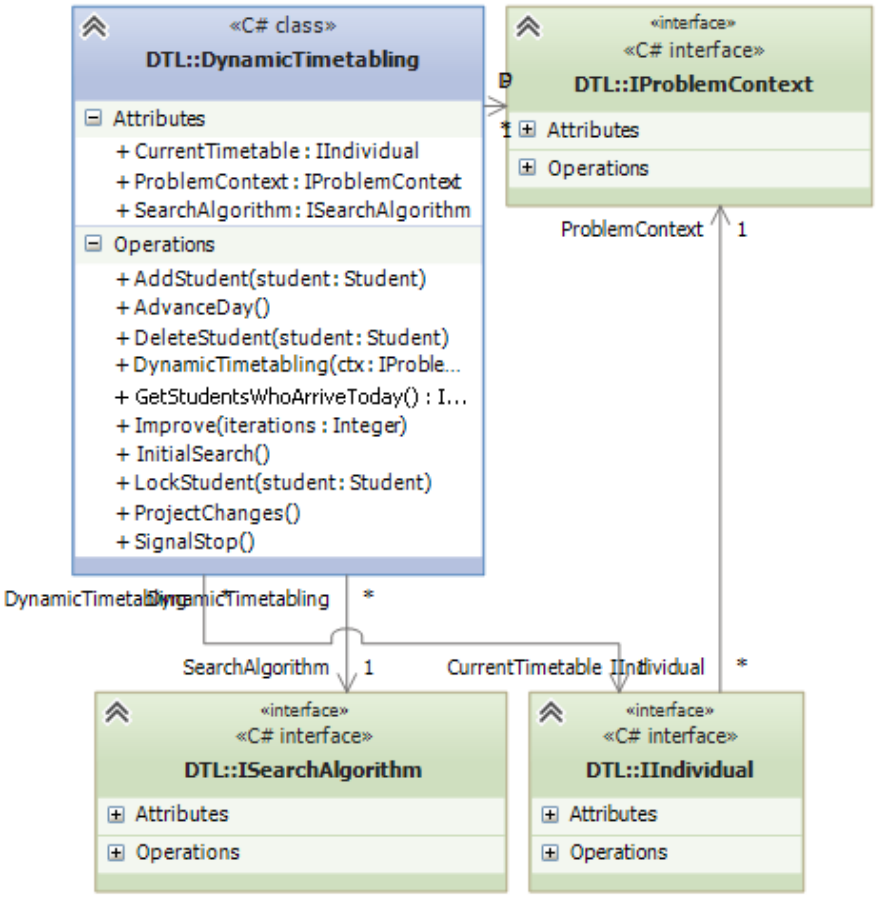


Figure 4.10. UML Class diagram of `DTL::DynamicTimetabling`

The `DynamicTimetabling` class is responsible for the meta-algorithm actions, such as:

- Advancing day (method `AdvanceDay()`)
- Adding students (method `AddStudent()`)
- Deleting students (method `DeleteStudent()`)
- Locking students (method `LockStudent()`)

It also provides a method for projecting the change of context onto the existing individuals.

4.2.6 The Input Format

An input scenario (scenarios are described in 4.1.1) is stored in a text, which has the following format: First line contains total numbers of expected students, total number of teachers, days, subjects, number of days and numbers of periods in a single day:

```
TT DATA | STUDENTS 20 | TEACHERS 40 | SUBJECTS 20 | DAYS 20 | PERIODS 30
```

this line must be followed by exactly n lines, where n is the number of subjects. Each line specifies name of a particular subject in its duration:

```
SUBJECT | NAME C0 | DURAT 3
```

These lines must be followed by exactly m lines, where m is the number of teachers. Each line specifies respectively:

- Teacher's name
- Proficiency of the teacher (subject names separated by space)
- List of unavailable days
- List of unabavilable periods in form of `X:A,B,C;Y:D,E,F;`, where `X` and `Y` are day indices and `A,B,C,D,E,F` are unavailable periods (slots) of particular days

```
TEACHER | NAME T0 | PROFICIENCY C0 | UNAV_DAYS | UNAV_PERIODS
0:13,14,15,16;1:13,14,15,16;2:13,14,15,16;3:13,14,15,16;4:13,14,15,
16;5:13,14,15,16;6:13,14,15,16;7:13,14,15,16;8:13,14,15,16;9:13,14,
15,16;10:13,14,15,16;11:13,14,15,16;12:13,14,15,16;13:13,14,15,16;
14:13,14,15,16;15:13,14,15,16;16:13,14,15,16;17:13,14,15,16;18:13,
14,15,16;19:13,14,15,16
```

These lines must be followed by exactly p lines, where p is the number of students. Each line specifies respectively:

- Student's name
- Day of student's arrival
- Students curriculum and lesson volume in form of `SUB1:X,SUB2:Y,...`, where `SUB1` and `SUB2` are names of subjects and `X,Y` are their respective volumes.

```
STUDENT | NAME S0 | ARRIVAL 0 | CURRICULUM C16:5,C6:3,C1:2,C17:3,C15:2,
C11:3
```

File in this format is then parsed by the `DTL::Data.LoadFromFile(string path)` method into the `DTL::Data` type object, which holds the data in a form of simple lists:

```
public List<Student> Students { get; set; }
public List<Teacher> Teachers { get; set; }
public List<Subject> Subjects { get; set; }

public int StudentCount { get; set; }
public int TeacherCount { get; set; }
public int SubjectCount { get; set; }
public int DayCount { get; set; }
public int PeriodCount { get; set; }
```

This `Data` object is then fed to the `ProblemContext` through its constructor.

These scenario files can be randomly generated by the **DP_DATAGEN** project, which allows user to control several parameters of random variables, namely:

- Student count, teacher count, subject count [exact]
- Teacher proficiency [range]
- Subject duration [range]
- Day count [exact]
- Periods of a single day [exact]
- Lessons per subject of a student [range]
- Number of subjects enrolled [range]
- Teacher unavailability periods [exact]
- Student arrival times [range]

4.3 Objective Function

An *objective function* (called also a *fitness function* in the terminology of genetic algorithms) is used to determine quality of a timetable (called an *individual* in the terminology of genetic algorithms) and is supposed to be minimised or maximised (minimised in our case).

An objective function may only take soft constraints into account (when hard constraints are dealt with by state transition operators) or it may also take hard constraints into account (when state transtition operators allow hard constraint violation). In our case, both soft and hard constraints are taken into account.

4.3.1 Formalisation

Let:

- $\mathbb{S}$ be the set of all student-subjects,
- $\mathbb{T}$ be the set of all teachers,
- $\mathbb{S}_t$ be the set of all student-subjects taught by teacher t,
- $time : \mathbb{S} \mapsto \mathbb{N}$ be the mapping of student-subjects to their scheduled times
- N_p be the number of periods (slots) of a single day
- $tc(x, y), x \in \mathbb{S}, y \in \mathbb{S}$ be the predicate which is true exactly when teachers of student-subjects x and y are the same and are participating in a *teacher clash* (i.e. one teacher is supposed to teach two different lessons in one time),
- $sc(x, y), x \in \mathbb{S}, y \in \mathbb{S}$ be the predicate which is true exactly when students of student-subjects x and y are the same and are participating in a *student clash* (i.e. one student is supposed to take two different lessons in one time),
- $on(x), x \in \mathbb{S}$ be the predicate which is true exactly when the student-subject x is scheduled in such way that it crosses from one day to another (for instance, the first half of the lesson is scheduled to monday evening and the second half of the lesson is scheduled to tuesday morning). This situation is called an *overnight violation*,
- $av(x), x \in \mathbb{S}$ be the predicate which is true exactly when the teacher of the student-subject x is unavailable at the time x is scheduled. This situation is called an *availability violation*,
- $\mathbb{T}_c = \{x \in \mathbb{S} \mid y \in \mathbb{S}, x \neq y, tc(x, y)\}$ (the set of all student-subjects which participate in teacher clashes)
- $\mathbb{S}_c = \{x \in \mathbb{S} \mid y \in \mathbb{S}, x \neq y, sc(x, y)\}$ (the set of all student-subjects which participate in student clashes)
- $\mathbb{O}_c = \{x \in \mathbb{S} \mid on(x)\}$ (the set of all student-subjects which participate in overnight violations)
- $\mathbb{A}_c = \{x \in \mathbb{S} \mid av(x)\}$ (the set of all student-subjects which participate in availability violations)

4.3.2 Hard Constraints

This component of the objective function has the biggest priority and it is required to be zero before locking any students or advancing to a next day. Advancing to a next day without $C_h = 0$ may cause severe structural violations in a timetable.

$$C_h = |\mathbb{T}_c| \ + \ |\mathbb{S}_c| \ + \ |\mathbb{O}_c| \ + \ |\mathbb{A}_c|$$

4.3.3 Maximal Schedule Length

This component computes an average start time of all student-subjects with respect to days they are scheduled on. It sums every student-subject start times in class $\mathbb{Z}_{N_p}$ and divides the sum with the number of all student-subjects. The result is therefore normalised.

$$C_{max} = \frac{\sum_{x \in \mathbb{S}} \frac{mod(time(x), N_p)}{N_p}}{|\mathbb{S}|}$$

4.3.4 Standard Deviation of Teacher Utilisation

Utilisation of a certain teacher specifies how many lessons that particular teacher teaches. Minimalisation of a standard deviation of this number over all teachers should guarantee that all the teachers teach roughly the same amount of lessons.

First, we need a mean value of utilisation U_m:

$$U_m = \frac{\sum_{t \in \mathbb{T}} \sum_{x \in S_t} 1}{|\mathbb{T}|}$$

And then the standard deviation U_{std} is computed. In order to properly normalise the standard deviation, a maximal standard deviation would have to be found, but because it presents an additional computational overhead, a precomputed mean value is used instead.

$$U_{std} = \frac{1}{U_m} \sqrt{\frac{\sum_{t \in \mathbb{T}} \left(U_m - \sum_{x \in \mathbb{S}_t} 1\right)^2}{|\mathbb{T}|}}$$

4.3.5 Feasibility Objective Function

This objective function is used in cases when we only want to get feasible schedule.

$$F = |\mathbb{T}_c| \ + \ |\mathbb{S}_c| \ + \ |\mathbb{O}_c| \ + \ |\mathbb{A}_c|$$

4.3.6 Objective Function with Maximal Schedule Length

This objective function (1.05× slower than the feasibility objective function) is used in cases when we also want maximal schedule length as a soft constraint.

$$F = |\mathbb{T}_c| \ + \ |\mathbb{S}_c| \ + \ |\mathbb{O}_c| \ + \ |\mathbb{A}_c| + \frac{\sum_{x \in \mathbb{S}} \frac{mod(time(x), N_p)}{N_p}}{|\mathbb{S}|}$$

4.3.7 Combined Objective Function

This objective function combines both soft constraints (maximal schedule length and standard deviation of teacher utilisation) and gives them equal convex coefficients (0.5). In theory, the sum of soft constraint values should never get bigger than 1 (so the improvement in one hard constraint is prefered over improvement in all the soft constraints), but due to rough normalisation of U_{std}, this might actually happen (although it never happened during experiments).

$$F = |\mathbb{T}_c| \ + \ |\mathbb{S}_c| \ + \ |\mathbb{O}_c| \ + \ |\mathbb{A}_c| + \frac{\sum_{x \in \mathbb{S}} \frac{mod(time(x), N_p)}{N_p}}{2\,|\mathbb{S}|} + \frac{1}{2\,U_m} \sqrt{\frac{\sum_{t \in \mathbb{T}} \left(U_m - \sum_{x \in \mathbb{S}_t} 1\right)^2}{|\mathbb{T}|}}$$

4.3.8 Fitness Functor

Fitness functors, apart from providing objective function value for individuals, also seek and mark hard constraint-violating student-subjects of inviable individuals. These marks are then utilised in informed perturabation operators.

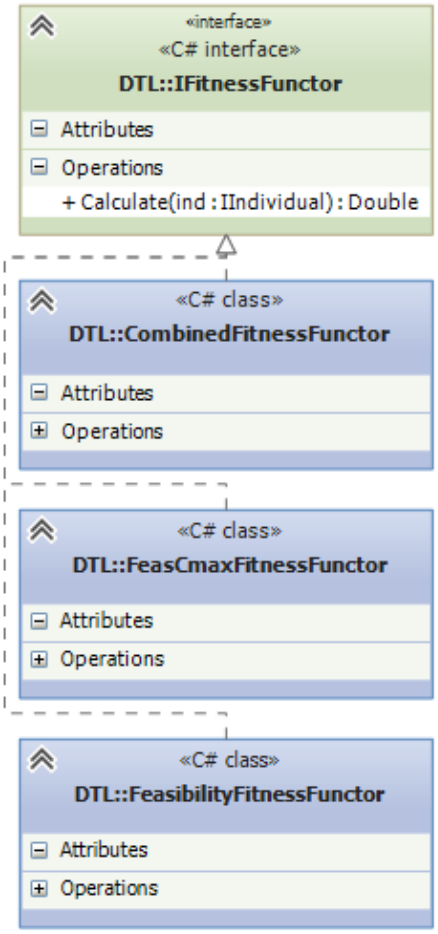


Figure 4.11. UML Class diagram of `DTL::IFitnessFunctor`

Class diagram 4.11 shows, that there are three concrete implementations which exactly correspond to the functions defined in 4.3.5, 4.3.6 and 4.3.7.

Chapter 5
Benchmarking Methodology

This chapter describes statistical testing ([21]) and visualisation methods which are used to evaluate results of benchmarks. In our case, statistical tests are used for concluding about equality of underlying distributions of benchmark results (for instance, to see if resulting objective function values differ when multiple parameter settings are used).

5.1 F-test of Equality of Variances

F-test or Fischer's test ([21]) is used to conclude about equality of variances of two sample sets, whose probability distributions are normal (which is very reasonable assumption).

We assume two random independent sample sets sampled from distributions $N(EA, DA)$ and $N(EB, DB)$ and we want to test *null hypothesis* about equality of variances. The test statistic is:

$$T = \frac{S_A}{S_B}$$

This statistic has Fisher-Snedecor distribution $F(\xi, \eta)$ wit ξ and η degrees of freedom. Figure 5.1 shows Fisher-Snedecor distribution for $(5, 5)$ degrees of freedom.

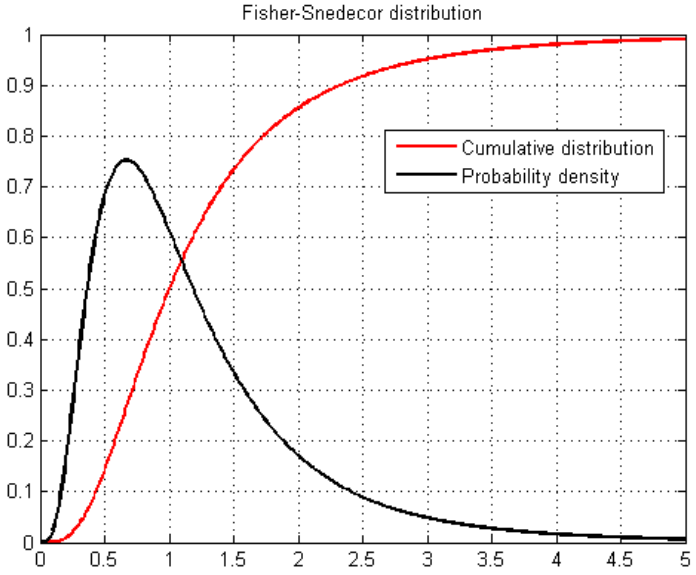


Figure 5.1. Fisher-Snedecor probability distribution for $(5, 5)$ degrees of freedom

In the F-test, the F-distribution has $(n-1, m-1)$ degrees of freedom, where n is the number of samples in the first set and m is the number of samples in the second set.

Given the level of accuracy α, we can obtain critical points:

$$k_1 = u_{F(n-1,m-1)}\left(\frac{\alpha}{2}\right)$$

and

$$k_2 = u_{F(n-1,m-1)}\left(1 - \frac{\alpha}{2}\right)$$

If the realisation t of the test statistic T lies inside the interval (k_1, k_2), null hypothesis holds.

5.2 Student's T-test of Equality of Means

T-test or Student's T-test ([21]) is used to conclude about equality of means of two sample sets, whose probability distributions are assumed to be normal.

Again, we assume two random independent sample sets sampled from distributions $N(EA, DA)$ and $N(EB, DB)$ but this time we want to test null hypothesis about equality of means. The test statistic is:

$$T = \frac{\overline{A} - \overline{B}}{\sqrt{(n-1)S_A^2 + (m-1)S_B^2}} \cdot \sqrt{\frac{mn(m+n-2)}{m+n}}$$

This statistic has Student's T-distribution with $m+n-2$ degrees of freedom. Figure 5.2 show Student's T-distribution with 5 degrees of freedom.

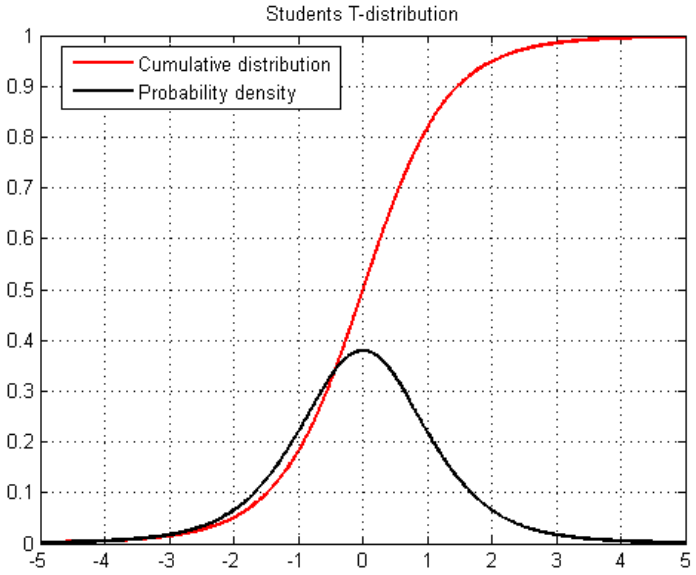


Figure 5.2. Student's T-distribution for 5 degrees of freedom

We need to compare realisation of the test statistic with an appropriate quantile:

$$q_{T(m+n-2)}\left(1 - \left(\frac{\alpha}{2}\right)\right)$$

If the realisation t of the statistic T lies beneath this quantile, the null hypothesis holds.

5.3 Concluding about Algorithm Stability

The common way of visualising results of multiple runs of a randomised algorithm is to plot mean results, but this way of visualising does not say anything about variance and thus about stability.

More proper way of visualising results is by using *boxplots*.

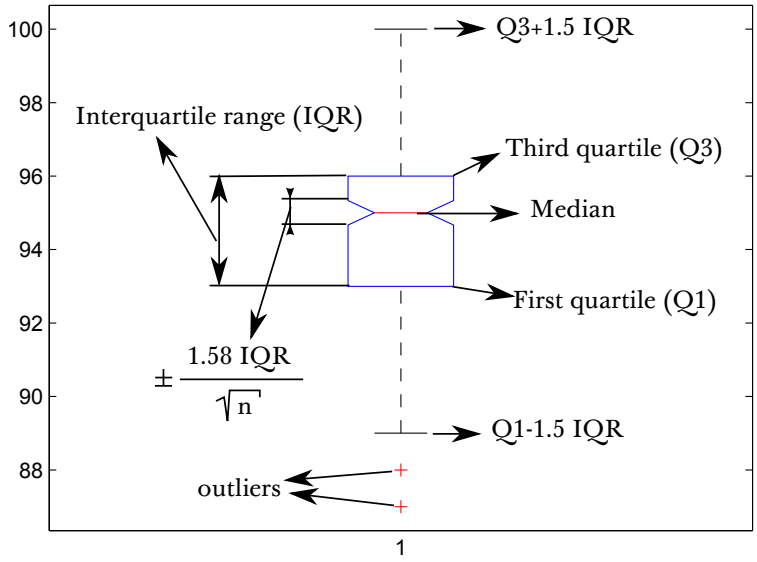


Figure 5.3. Description of boxplot

Boxplots depict data medians, as well as remaining two quartiles (to show an interval inside which the majority of data lies) along with some additional statistical information (interquartile range and outliers).

5.4 Implementation of Statistical Functionality

It is reasonable to use number of objective function calls as an independent variable (instead of time), because this measure is hardware-independent and equidistance of samples can be obtained without any special effort. For this reason, the `ISearchAlgorithm` interface contains the event called `StatisticalHit`.

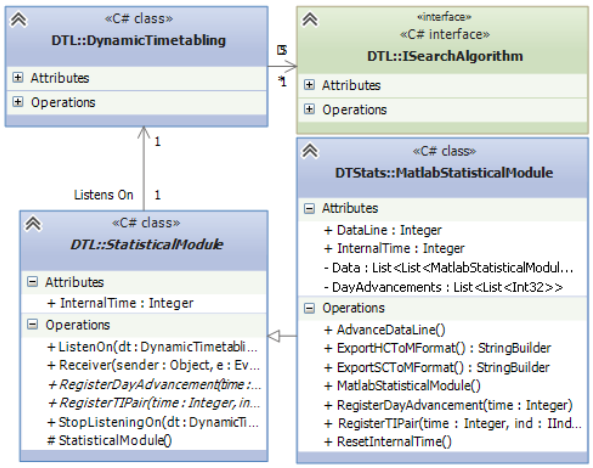


Figure 5.4. UML Class diagram of `DTL::StatisticalModule`

`StatisticalHit` event is raised by an inner search algorithm whenever an objective function is called. If there is an approriate listener registered on the event, it will be called and provided with statistical data. The dynamic timetabling library provides `StatisticalModule` abstract class, which can be extended by external statistical software. In our case, this software is a statistical project, which exports the data to a ready-to-execute MATLAB files, which produce various graphs and boxplots upon their execution.

5.4.1 Matlab-exporting Statistical Program

In order to visualise results of the algorithm, a program which exports statistical data from the algorithm to MATLAB M-files was created. This program executes so-called *execution plan* on a `DynamicTimetabling` instance and repeats the execution several times with different seeds, while obtaining statistical data by using `MatlabStatisticalModule` observer, which listens on the `DynamicTimetabling` instance.

The execution plan simulates behaviour of a meta-algorithm user. It is represented as an ordered list of enum-type variables, for instance the following plan

```
List<DTAction> plan = new List<DTAction>
{
    DTAction.Improve,
    DTAction.LockRemaining,
    DTAction.AdvanceDay,
    DTAction.Randomize
}
```

runs an improvement cycle (for a number of iterations specified *a priori*), then it locks all the students who were expected at that day, then it advances the day and then it randomizes remaining students' lessons. There are several types of actions which can be part of the execution plan:

- `DTAction.Improve` — Runs improvement cycle for n iterations, where n is specified *a priori*.
- `DTAction.AdvanceDay` — Advances the current day.
- `DTAction.Lock` — Locks the next student in the arrival-expectation queue.
- `DTAction.Delete` — Deletes the next student in the arrival-expectation queue.
- `DTAction.LockRemaining` — Locks the whole arrival-expectation queue.
- `DTAction.DeleteRemaining` — Deletes the whole arrival-expectation queue.
- `DTAction.Randomize` — Randomizes remaining students' lesons.

Chapter 6
Inquiry into Conventional Perturbation-based State Space Search Methods

In this chapter, we shall describe conventional perturbation-based state space search methods (such as exhaustive enumerative search, local search with different search strategies, tabu search and stochastic hill-climbing) and discuss their advantages and drawbacks in terms of applicability on the problem of our concern.

6.1 The State Space

Prior to any inquiry into perturbation-based search methods, an analysis of the state space must be performed, for it is the size and the complexity of the state space, which is crucial to performance of these algorithms.

Our state space S is a space of timetables constrained by their dimensions (number of timeslots, students-subjects and teachers). A timetable always has all the student-subjects scheduled, may it or may it not be consistent.

There are two *atomic operators*: $X_p(l_s), s \in S$, which moves the student-subject l_s of the state s to another time; and $X_t(r_{l_s}), s \in S$, which changes the teacher r_{l_s} of the student-subject l_s to another teacher. The transitive closure of the relation specified by the operators $\{X_p, X_t\}$ is equal to the state space S.

6.1.1 Branching Factor

Branching factor of S specifies how many state space transitions are possible from a single state. Because there are two transition operators, total branching factor equals to the sum of branching factors of particular operators:

$$F = F_{X_p} + F_{X_t}$$

Branching factor of a position operator is:

$$F_{X_p} = n_l \cdot n_d \cdot n_p$$

where n_l is the number student-subjects, n_d is the number of days and n_p is the number of periods (slots) of a single day. We can select any student-subject and put it to any time in any day.

Branching factor of a teacher operator is:

$$F_{X_t} = n_l \cdot n_r$$

where n_r is the number of teachers (in practice, this is reduced to just those teachers who have appropriate profficiency in the subject of l).

So the total branching factor is:

$$F = n_l \cdot (n_d n_p + n_r)$$

6.1.2 State Space Size

Cardinality of S is given by the following equation.

$$|S| = n_l!(n_t n_d n_p)^{n_l}$$

6.2 Exhaustive Enumerative Search

Exhaustive enumerative state space search, as its name suggests, is based on enumeration of the whole state space and selecting the best solution from this enumeration. It is often realised by recursive depth or breadth first search.

This method is only applicable on small discrete search spaces. The size of our state space renders this method unusable.

6.3 First-improving Local Search

First-improving local search is a local search variant, which selects the first solution which improves the temporary best one and continues the search on this new solution. The search is repeated until some termination condition is met.

```
Solution x.
Initialize(x).
Until termination condition Do
    Solution y = Perturb(x).
    If TargetFcn(y) > TargetFcn(x)
        x := y;
    End
End
```

Because often there is no specific reason for deterministic perturbation, this method is often stochastic, i.e. it generates more or less random neighbours of the currently best solution.

Performance of this method is correlated with the size of the state space: When the state space is small, there is a huge risk of getting stuck in a local extreme (but the search is very fast). On the opposite side, when the state space is very large (which holds in our case), the risk of stucking in a local extreme is reduced (it is reduced even more significantly in case of multiple runs with different seeds).

6.4 Best-improving local search

Best-improving local search is the second possible local search variant, which, instead of selecting the first improving solution, enumerates the whole neighbourhood and selects the best transition possible.

```
Solution x.
Initialize(x).
Until termination condition Do
    Solution y = BestFromNeighbourhood(N(x)).
    If TargetFcn(y) > TargetFcn(x)
        x := y.
    End
End
```

This method is very inefficent on large neighbourhoods (which is again, our case).

6.5 Stochastic Hill Climbing

Stochastic hill climbing is a variant of local search which introduces a method for overcoming local extremes. In this algorithm, there is always non-zero probability of selecting a solution from the neighbourhood, which is worse than the best-so-far solution.

```
P = .05.
Solution x.
Initialize(x).
Until termination condition Do
    Solution y = Perturb(N(x)).
    If TargetFcn(y) > TargetFcn(x) OR RandomDouble(0.0,1.0) < P
        x := y.
    End
End
```

6.6 Simulated Annealing

Simulated annealing is a metallurgy-inspired probabilistic search method based on physical properties of metal annealing.

In metallurgy, annealing is a method used to ensure regular atom grid in metals. A piece of metal is treated by extreme heat, so the atoms can move from their non-optimal positions, which cause the metal piece to be non-homogenous, and then slowly cooled down. In the process of cooling down, atoms tend to locate themselves in their equilibrium states.

An optimisation analogy of this process means introducing a time-dependent temperature function, which defines randomness of perturbation.

```
Solution x.
Initialize(x).
Temperature t.
Heat(t).
Until termination condition Do
    Solution y = Perturb(x).
    If TargetFcn(y) > TargetFcn(x)
        x := y.
    Else
        p := e ^ - ( (TargetFcn(y) - TargetFcn(x)) / T).
        If RandomDouble(0.0,1.0) < p
            x := y.
        End
    End
    Cool(t).
End
```

6.7 Tabu Search

Tabu search is very similar to common hill climbing, but it allows transitions whenever the new solution is better than or **equally good** (or possibly some delta) as the best-so-far solution. To avoid cycles, which are very likely to happen due to this non-strict

transition rule, the concept of *memory* is introduced. In this concept, inverses of the last n transitions are stored (i.e. when a transition $A \to B$ is made, $B \to A$ is stored) in a list called a *memory*.

Whenever a transition is about to be made, the algorithm iterates the memory and checks whether there is such transition in it. If there is, no transition is made, because it would lead the algorithm to already recently searched part of the state space.

After every iteration, some of the memories are deleted (according to their age).

```
Solution x.
Initialize(x).
List m.
Until termination condition Do
    Solution y = Perturb(x).
    If TargetFcn(y) >= TargetFcn(x) AND NotContains(m,[x->y])
        x := y.
        AddTo(m,[y->x]).
    End
    DeleteOldItems(m).
End
```

The major drawback of this method is that for large state spaces, the memory would have to be enormous to work effectively.

6.8 Applicability on the Problem of our Concern

Because of very large state spaces and neighbourhoods of the problem of our concern, only some methods of the previously mentioned are suitable and will be implemented, namely:

- First-improving local search
- Stochastic hill climbing
- Simulated annealing

Chapter 7
Inquiry into Population-based State Space Search Methods

Population-based state space search methods are, as the name suggests, search methods, which build their functionality upon maintaining a set of solutions called a population. There are many population-based search methods, such as genetic algorithms, evolutionary algorithms, ant colonies, particle swarm optimisation, scatter search etc., but in this thesis, we will only concern genetic algorithms and their hybrids.

7.1 Standard Genetic Algorithm

A standard genetic algorithm (SGA) is an algorithm which maintains a population of solutions, which is subjected to initialisation, selection, mutation, recombination and renewal (these are also called genetic operators).

The SGA can be inspired by one of the following theories of evolution ([22]):

- Either by **Darwinian** theory, which states that life experience of parents does not affect the child
- or by **Lamarckian** theory, which states exactly the opposite
- or by **Baldwinian** theory, which states that genetic information encodes the ability to learn.

All of these theories share the common idea called *the survival of the fittest,* which states that only the individuals, which show an exceptional ability to survive in their surroundings, get to live, reproduce and possibly project their qualities onto their offsprings.

The SGA tries to implement the following analogy betwen one of the evolution theories and problem optimisation:

- Population of life forms = Population of problem solutions
- Fitness of a single life form = Value of the problem solution objective function
- Natural selection = Stochastic selection functions based on objective function value
- Mating to maintain genetic diversity = Combination of two solutions
- Genetic mutation = Slight alteration of problem solutions
- Dying of age = Population replacement (generations)

By implementing this analogy, we obtain the inherent properties of evolution, but applied to problem optimisation. The basic structure of a genetic algorithm is the following:

```
Population P As List.
Population NewP As List.
Individual Best := null;
1. Initialise(P).
2. Individual A := Select(P).
```

```
3. Individual B := Select(P).
4. Individual C := Recombine(A,B).
5. C := Mutate(C).
6. If Fitness(C) > Fitness(Best)
       Best := C.
   End
7. Add(C, NewP).
8. If Size(NewP) = Size(P)
       P := NewP.
       Clear(NewP).
   End
9. If Not TerminationCondition
       Goto 2.
   End
```

7.1.1 Initialisation

An initialisation operator is used to fill the first population with individuals. These individuals can be either completely random or partially random (improved or generated by some informed procedure).

7.1.2 Selection

A selection operator selects two individuals which will be subjected to recombination. The selection must be based on an objective function value, so the better individuals are selected more often. Also, to achive genetic diversity (and thus the ability to overcome local extrema), there must be a non-zero probabilty, that the worst individual from the population gets selected. Two most common selection operators are:

- **Fitness proportional selection** (also called *roulette selection*) — Each individual has a probability of selection, which is directly equal to the percentage of his share on the total fitness of the whole population. For instance, if there are three individuals in the population, having fitness values 50, 25 and 25, respectively, then the first individual has a probability of 0.5 of being selected.
- **Tournament selection** — A certain number of random individuals is selected from the population. Given the probability p, the best individual of this selection is selected for recombination with probability p. The second individual of this selection is selected for recombination with probability $p(1-p)$ etc.

7.1.3 Recombination

A recombination operator (also called *crossover operator*) constructs one or more new individuals by combining genotypes of selected parents. This operation's effectivity is based on assumption, that it is possible to obtain a child that is better (in terms of its fitness) than both of its parents. There are two common types of recombination:

- **Uniform recombination** — parts of parental genotypes are combined completely at random.
- ***n*-point recombination** — parental genotypes are split by n cuts and the resulting sections are combined to produce one or more different children

7.1.4 Population Renewal

Population can be either completely replaced by offspring in each generation, or it can be continually pruned and updated with children.

One of the important aspects of the SGA is called *elitism*. When using elitism, the most fit individuals from the old population are selected deterministicaly and transfered to the new population without any changes. This behaviour is supposed to guarantee, that the quality of the solution will not decrease from generation to generation.

7.2 Memetic Algorithms

Memetic algorithms ([22]) are genetic algorithms that follow the *Lamarckian evolution scheme*, according to which some life experiences of the parents can be inherited by their children.

Optimisation analogy of this behaviour is implemented by using an additional *inner search* operator. This inner search operator tries to improve a genotype (commonly by using some local search technique) before or after its mutation, or to improve genotypes of individuals in the initial population.

There exists a standard categorisation of memetic algorithms:

- **LTH (Low-level teamwork hybrid)** — Local search embedded in a GA
- **HRH (High-level relay hybrid)** — Initial population of a GA is built by a local search
- **HTH (High-level teamwork hybrid)** — Several GAs are run in a parallel model, cooperating with one another

Chapter 8
Experiments with Conventional Perturbation-based State Space Search Methods

In this chapter, the algorithms mentioned in 6 will be subjected to several benchmarks in order to select the most suitable one for the dynamic timetabling tool. In addition, several perturbation operators will be presented, analysed and their performance will be evaluated.

8.1 Benchmark Scenarios

There are five different benchmarking instances:

- **Small Loose Satisfiable (SLS)** — An instance very easy to solve, with 20 students, 40 teachers, 20 subjects, 20 days, 30 periods, with subject durations being 2 to 5 periods, curriculum size being 4 to 7 subjects and lesson volume being 2 to 5 lessons per subject.
- **Small Tight Unsatisfiable (STU)** — A small instance, which was proven to be unsatisfiable.
- **Small Tight Satisfiable (STS)** — A small instance, which is satisfiable, but the resulting timetable is very tight (with not much space between lessons).
- **Realistic Satisfiable 1 (RS1)** — A large instance with 100 students, 200 teachers, 100 subjects, 25 days, 16 periods (one period equals 30 minutes, thus adding to 8 hour work day), 1 to 3 period subject durations (either 30, 60 or 90 minute lessons), lunchbreaks taking 1 period (30 minutes), students arriving until day 5, each having enrolled 4 to 8 subjects with 5 to 10 lessons of each.
- **Realistic Satisfiable 2 (RS2)** — Another large instance with same parameters as RS1.

8.2 Perturbation Operators

Several position and teacher perturbation operators were designed and will be subjected to benchmarking. In general, there are two basic groups of these operators—informed and uninformed. *Uninformed* perturbation operators do not exploit any of the properties of the problem and are almost completely random. This randomness decreases the risk of getting stuck in a local extreme, but it makes the search process itself significantly slower. On the other side, *informed* perturbation operators exploit the fact, that there are subparts of timetable, which cause worse penalty than any other subparts. These subparts are identified by a fitness function, which sets a predicate called `Bad` on those student-subjects, which cause hard constraint violations.

Perturbation operators can also introduce a stochastic element, which will control the amount of information they exploit. This element is a ratio, which specifies how many times a `Bad` predicate is ignored. These are called *stochastic informed* perturbation operators.

Some operators can also guarantee, that their execution will not cause any new hard constraint violations (or violations of subsets of hard constraints). In our case, operators with prefix `NonOvernight` are guaranteed not to violate 3.2.3.C7 (i.e. *overnight* constraint).

8.2.1 Blind Position Perturbation Operator

This operator selects a student-subject from so called *unlocked subset* of a timetable, which is defined by the following lambda selector:

```
var unlockedSubset = ind.Table.Keys.Where(x => !x.Student.Locked &&
ind.Table[x].Time >= Math.Max(day_offset,
x.Student.arrival * ind.ProblemContext.Data.PeriodCount));
```

where `ind` is the perturbed timetable (individual), `ind.Table` is the timetable hashtable (described in 4.2.1). As we can see, the unlocked subset contains only unlocked student-subjects whose start times do not precede the current day. The `day_offset` is defined as:

```
int day_offset = ind.ProblemContext.CurrentDay *
ind.ProblemContext.Data.PeriodCount;
```

After selecting a random element of the unlocked subset, its time is changed to a randomly selected new position, which must also satisfy certain conditions (it must not be in the past, for instance).

```
int rnd_ss = (int)Math.Floor(ind.ProblemContext.Random.NextDouble() *
(unlockedSubset.Count()));
var ss = unlockedSubset.ElementAt(rnd_ss);

day_offset = Math.Max(day_offset, ss.Student.arrival *
ind.ProblemContext.Data.PeriodCount);

int rnd_subject_pos = day_offset + (int)Math.Floor(
ind.ProblemContext.Random.NextDouble() * ((
ind.ProblemContext.Data.PeriodCount *
ind.ProblemContext.Data.DayCount) - ss.Subject.units - day_offset));

ind.Table[ss].Time = rnd_subject_pos;
```

8.2.2 Non Overnight Position Perturbation Operator

This operator guarantees, that its execution will not cause any new violation of constraint 3.2.3.C7 by making sure, that the following equation holds:

$$\left\lfloor \frac{P_n}{N_p} \right\rfloor = \left\lfloor \frac{P_n + N_s}{N_p} \right\rfloor,$$

where P_n is the randomly generated period for the subject to start on, N_p is the number of periods per day and N_s is the number of periods taken by the subject.

8.2.3 Informed Position Perturbation Operator

This operator introduces the so called *inviable set*, which is a subset of an unlocked set and is defined by the following lambda selector:

```
var inviableSubset = unlockedSubset.Where(x => x.Bad);
```

In other words, it selects all the elements from the unlocked set, which have `Bad` predicate on them. After defining the inviable set, it continues to operate the same way as shown in 8.2.1, but it uses the inviable set instead of the unlocked one until there are no more hard constraint violations. After that, it falls back to using the unlocked set.

8.2.4 Non Overnight Informed Position Perturbation Operator

This operator combines features of operators 8.2.2 and 8.2.3.

8.2.5 Stochastic Non Overnight Informed Position Perturbation Operator

This operator is very similar to the operator 8.2.4, but unlike it, this operator can be set to ignore `Bad` predicates with certain probability, causing the search process to be less likely to get stuck in a local extreme.

8.2.6 Blind Teacher Perturbation Operator

This operator changes the teacher of a student-subject randomly selected from the unlocked set. The new teacher is selected from a set of teachers which have appropriate profficiency.

8.2.7 Informed Teacher Perturbation Operator

This operator acts similarly to the operator 8.2.3, but it changes teachers instead of postions.

8.2.8 Stochastic Informed Teacher Perturbation Operator

This operator is very similar to the operator 8.2.7, but unlike it, this operator can be set to ignore `Bad` predicates in the same way as shown in 8.2.5.

8.3 First-improving Local Search

As it was stated in 6.8, first-improving local search is one of the perturbation-based state space search methods, which is suitable for the problem of our concern, because it can cope with large neighbourhoods.

In the object model, this algorithm is represented by the `RandomisedLocalSearch` class (randomised because of the stochastic neighbour selector mentioned in the inquiry (6.3)).

8.3.1 Using Blind Operators on “SLS” Scenario

In the figures 8.1 and 8.2, we can see the dependency of hard and soft constraints (respectively) on the number of objective function evaluations. We can see that the convergence of the number of hard constraint violations is rather fast and it does converge to zero. Soft constraints converge slower, which was expected, because their share in the objective function is lower.

In the figures 8.3, 8.4, 8.5, 8.6, 8.7 and 8.8, we can see, that the algorithm becomes increasingly stable (the experiment was repeated 10 times).

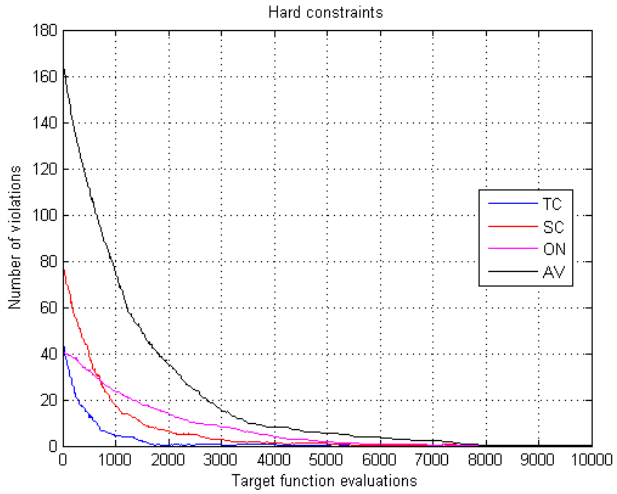


Figure 8.1. Hard constraint progress for first-improving local search with blind p.o.

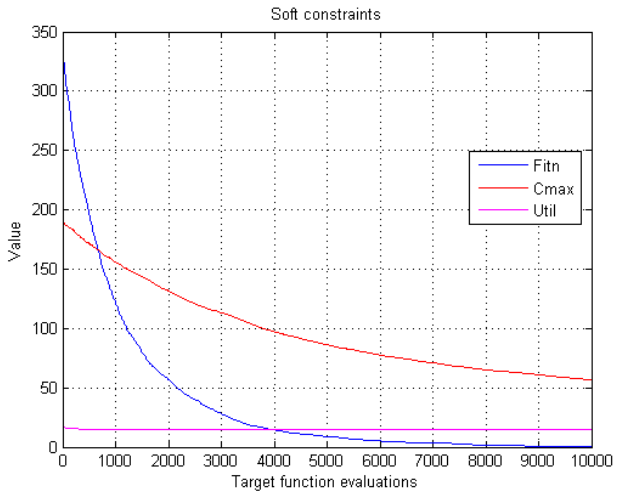


Figure 8.2. Soft constraint progress for first-improving local search with blind p.o.

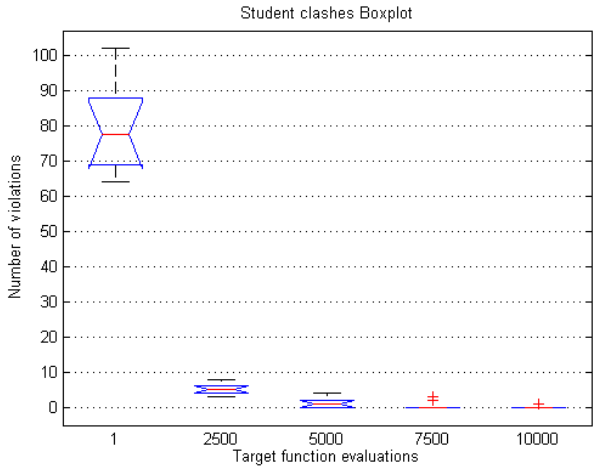


Figure 8.3. Student clashes boxplots for first-improving local search with blind p.o.

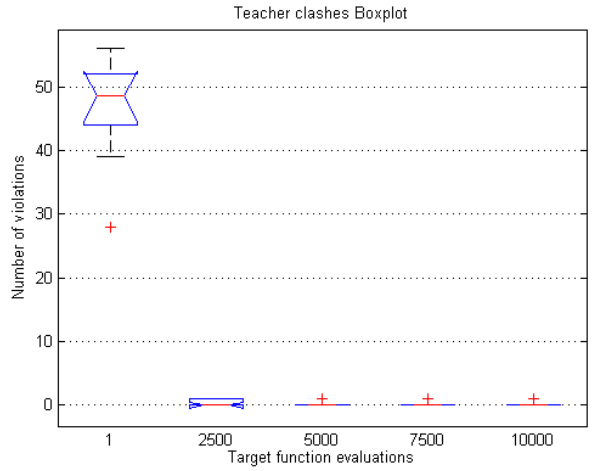


Figure 8.4. Teacher clashes boxplots for first-improving local search with blind p.o.

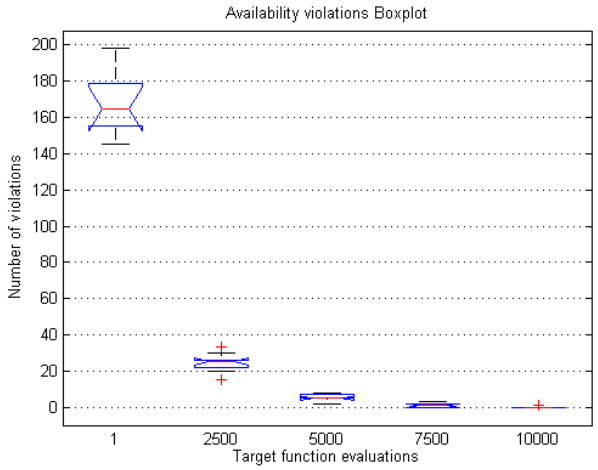


Figure 8.5. Availability violation boxplots for first-improving local search with blind p.o.

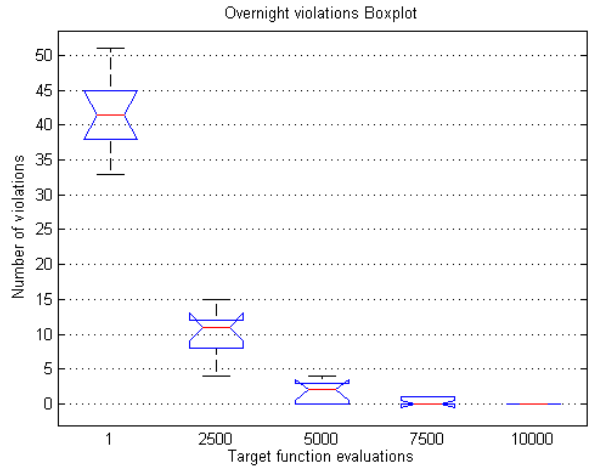


Figure 8.6. Overnight violation boxplots for first-improving local search with blind p.o.

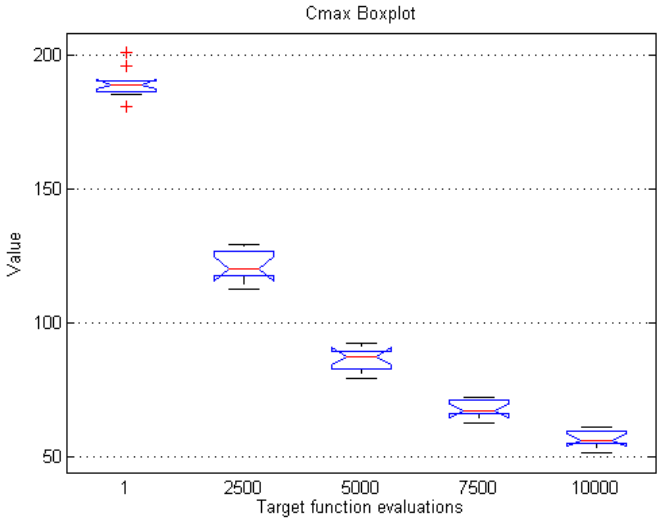


Figure 8.7. Cmax boxplots for first-improving local search with blind p.o.

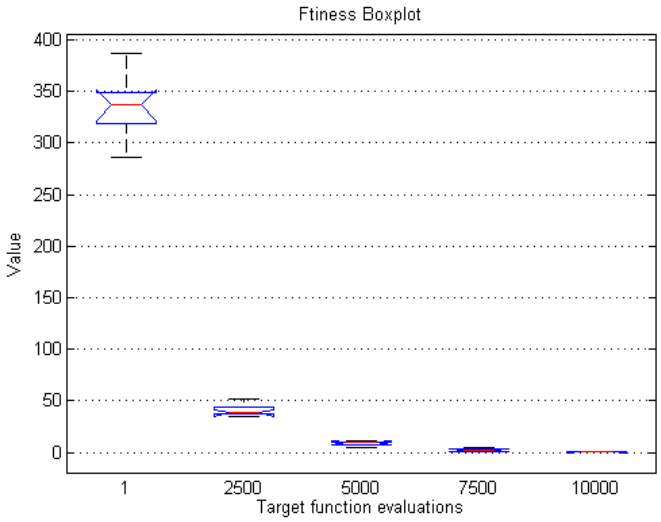


Figure 8.8. Fitness boxplots for first-improving local search with blind p.o.

8.3.2 Comparison of Blind and Non-overnight Operators on "SLS" Scenario

As we can see in the figure 8.9, non-overnight perturbation operators have achieved very similar results in terms of convergence and stability, so a statistical test was performed to see if there is any significant difference.

Results of the statistical test ($\alpha = .05$) showed, that fitness variances at 10^4th objective function evaluation are equal, but means are not (non-overnight operator produced lower mean fitness), and thus the non-overnight operators perform better on the SLS scenario.

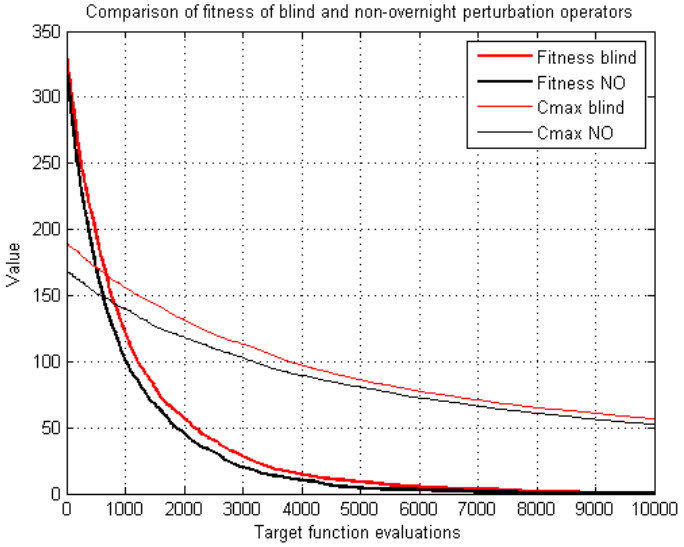


Figure 8.9. Comparison of blind and non-overnight p.o. objective function and Cmax.

8.3.3 Comparing Informed, Uninformed and Stochastic Informed Operators on "STS" Scenario

Due to the results of the previous experiments, which show, that the non-overnight operators are more effective than the blinds ones, only non-overnight operators will be used henceforth.

In this experiment, we compared three of the non-overnight operator variants (blind, informed and stochastic informed) and compared progress, stability and final values of hard and soft constraints.

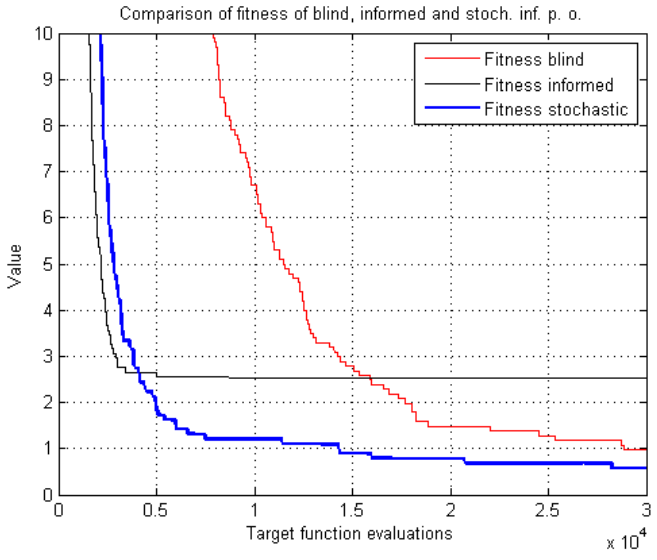


Figure 8.10. Fitness value progress comparison for blind, informed and stochastic inf. op.

As we can see in the figure 8.10, the informed operator has got stuck in a local extreme and it was unable to recover from it, because the inviable set contained too few elements. The blind operator has a significantly slower convergence rate, but it is able to overcome the extremes. The stochastic informed operator, which chooses to ignore the inviable set with probability 0.5, converges nearly as fast as the blind

operator, but has no problems with getting stuck. It was shown by statistical test (with $\alpha = .05$), that the blind and stochastic informed operators have the same means and variances at $3 \cdot 10^4$ evaluations, but yet the stochastic one is clearly better because of the convergence ratio it can achieve.

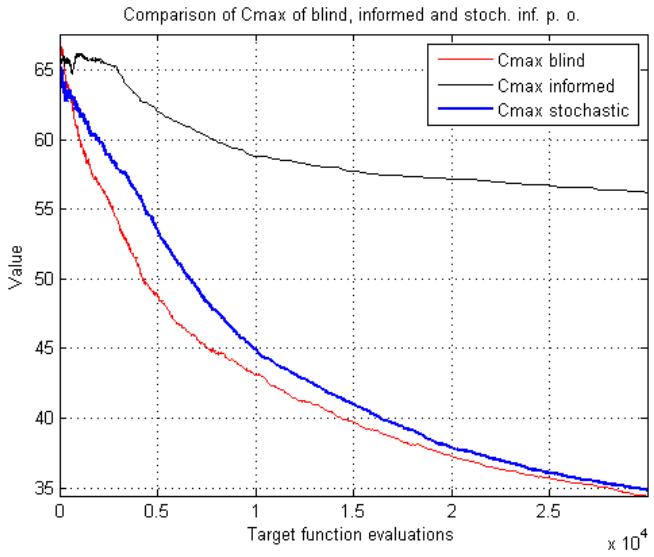


Figure 8.11. Cmax value progress comparison for blind, informed and stochastic inf. op.

In the figure 8.11 we can see, that in case of the informed operator, the Cmax converges slowly, because most of the student-subjects lie outside the inviable set. Statistical test has shown (with $\alpha = .05$), that the blind and stochastic operators have the same means and variances at $3 \cdot 10^4$ evaluations.

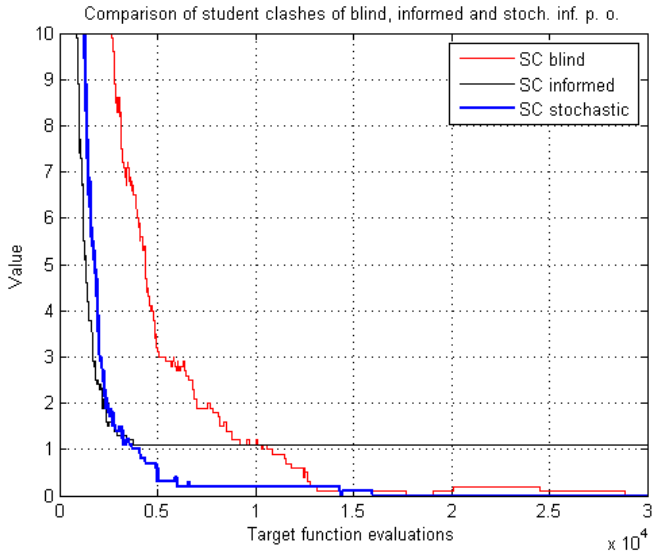


Figure 8.12. SC progress comparison for blind, informed and stochastic inf. op.

In the figures 8.12, 8.13 and 8.14 we can see the progression of the number of hard constraint violations. In case of availability constraints, the stochastic informed operator was shown to be significantly better at $3 \cdot 10^4$ evaluations than the remaining two operators.

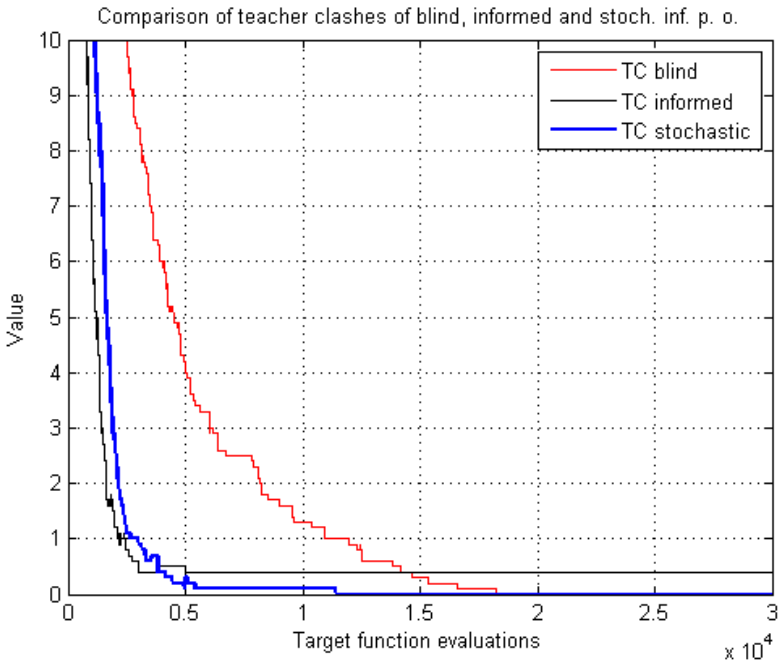


Figure 8.13. TC progress comparison for blind, informed and stochastic inf. op.

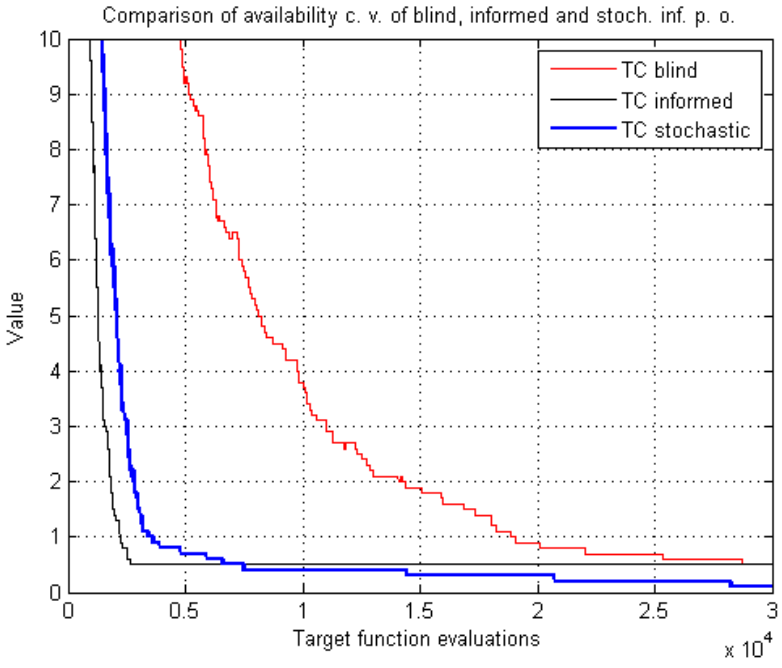


Figure 8.14. AV progress comparison for blind, informed and stochastic inf. op.

8.3.4 Comparing Informed, Uninformed and Stochastic Informed Operators on "STU" (unsatisfiable) Scenario

As we can see in the figure 8.15, all the operators perform equally bad on the unsatisfiable scenario.

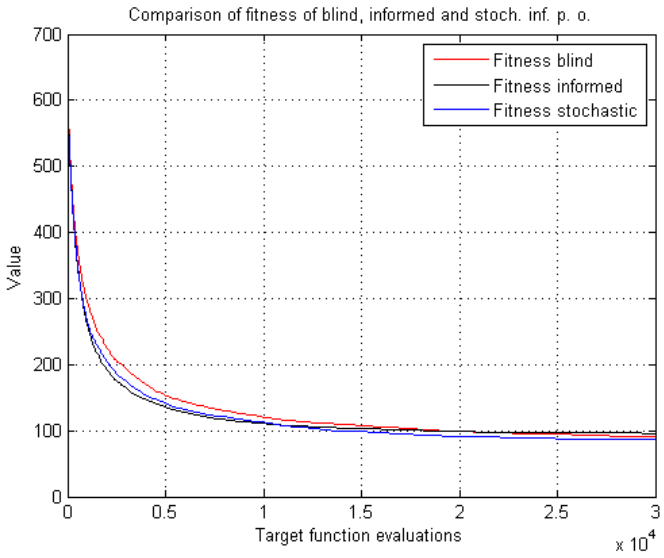


Figure 8.15. Fitness progress comparison for blind, informed and stochastic inf. op.

8.3.5 Using Blind and Stochastic Informed Operators on Realistic Scenarios

According to the previous experiments, stochastic informed operators outperform the remaining ones on small scenarios. In this experiment, we shall compare their performance on realistic scenarios ("RS1" and "RS2").

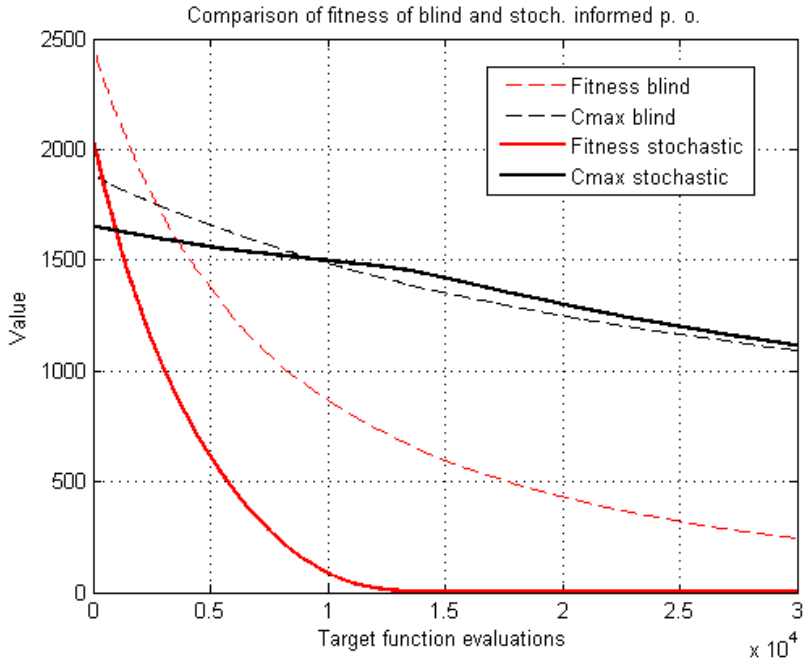


Figure 8.16. Fitness progress comparison for blind and stochastic inf. op. on RS1

We can see clearly from the figure 8.16, that the stochastic perturbation operators outperform the blind ones on the first realistic scenario as well. The algorithm was unable to achieve feasible and consistent timetable with blind operators, whereas with the stochastic informed ones, it was. In the figure 8.17 there is a similar result obtained

from the same experiment, but on the "RS2" scenario (which was generated by using exactly the same parameters as "RS1").

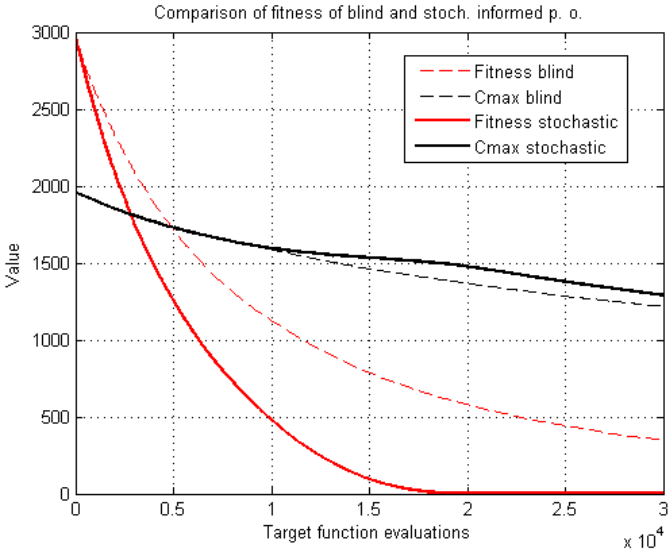


Figure 8.17. Fitness progress comparison for blind and stochastic inf. op. on RS2

8.3.6 Conclusion on First-improving Local Search

The first-improving local search was thoroughly analysed and found suitable for the problem. Several types of perturbation operators were tested with different benchmarks and the best one (informed stochastic) was found.

8.4 Stochastic Hillclimbing

Stochastic hillclimbing is the second algorithm suggested by the theoretical analysis conclusions in 6.8. It differs from the first-improving local search in one aspect—it can randomly select to accept worse solution. This behaviour should improve the ability of overcoming local extrema in the fitness function.

8.4.1 "STU" Scenario Benchmark

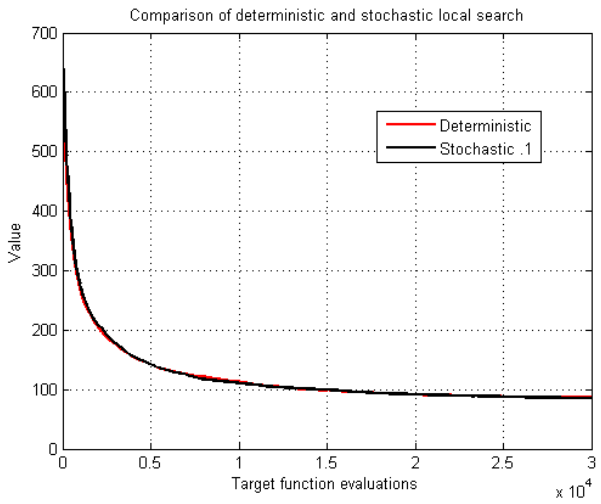


Figure 8.18. Comparison of deterministic and stochastic local search

The objective function we are using in this thesis seems not to have major extrema, so the stochastic hillclimbing was tested on the unsatisfiable scenario to see if it can ouperform the deterministic local search in the lowest number of hard constraint violations. As it can be seen in the figure 8.18 and as supported by statistical tests, it cannot.

8.4.2 Conclusion on Stochastic Hillclimbing

The stochastic hillclimbing algorithm was found unsuitable for the problem of our concern, because its main advantage lies in overcoming an objective function property, which the one of ours seems not to have.

8.5 Simulated Annealing

Simulated annealing is the last algorithm suggested by the theoretical analysis conclusions in 6.8. It differs from the first-improving local search in introducing a temperature function, according to which it allows transitions to worse states. In our case, the temperature function is:

$$T = 1 - tanh\left(\frac{i}{0.1N_i}\right)$$

where i is the current iteration and N_i is the total number or iterations.

8.5.1 "STU" Scenario Benchmark

In the figure 8.19 we can see the progress of the objective function in comparison with deterministic local search. The "hot" stage of the annealing has no effect at all, because the objective function seems not to have major local extrema for the algorithm to overcome.

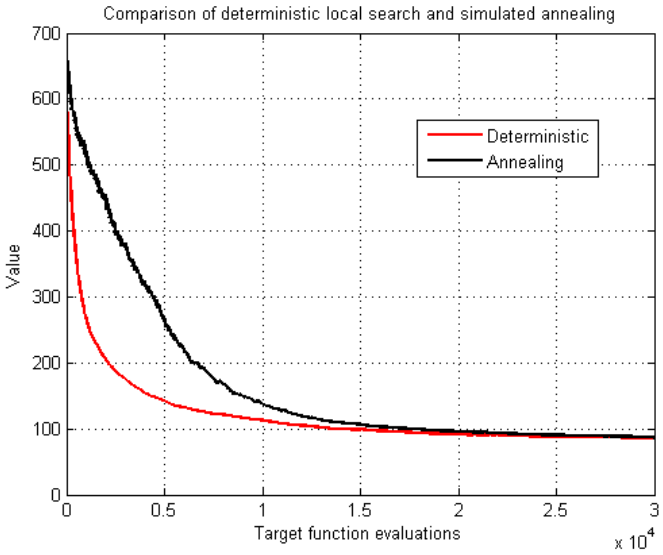


Figure 8.19. Comparison of deterministic local search as simulated annealing

8.5.2 Conclusion on Simulated Annealing

The simulated annealing algorithm was found unsuitable for the problem of our concern, because of the same reasons as stated in 8.4.2.

8.6 Comparison Tables

	Scenario
Perturbation Operator Type	**SLS**
Overnight	0.7914
Non-overnight	0.4142

Table 8.1. Comparison of overnight and non-overnight operators on the SLS scenario (local search, measured at 10^4 evaluations)

	Scenario	
Perturbation Operator Type	**STU**	**STS**
Blind	90.6766	0.9753
Informed	96.1931	2.5349
Stochastic Informed	86.2798	0.5763

Table 8.2. Comparison of blind, informed and stochastic informed non-overnight operators on STU ans STS scenarios (local search, measured at 30^4 evaluations)

	Scenario
Algorithm	**STU**
First-improving Local Search	86.2798
Stochastic Hillclimbing	84.9843
Simulated Annealing	86.7820

Table 8.3. Comparison of local search, stochastic hillclimbing and simulated annealing on the STU scenario (measured at 30^4 evaluations)

Chapter 9
Experiments with Genetic Algorithms

In this chapter, the algorithms mentioned in chapter 7 will be subjected to several benchmarks in order to find out whether or not they are suitable for the problem of our concern. In addition, several recombination operators will be proposed and compared.

9.1 Settings

9.1.1 Mutation Operator

According to the results of chapter 8, stochastic informed non-overnight perturbation operator was selected as a mutation operator.

9.1.2 Recombination Operators

There are three recombination operators:

- **Uniform recombination operator**: Child takes parent student-subjects completely at random.
- **One point recombination operator**: A random student-subject index is selected. Child takes student-subjects from the first parent up to this index. After this index, the child takes student-subjects from the seconds parent.
- **Mild recombination operator**: Child has nearly exactly the same genotype as the first parent, only a single randomly selected student-subject is taken from the second parent.

9.1.3 Selection Operator

According to the results of preliminary benchmarking, the fitness proportional (roulette) selection operator was selected.

9.1.4 Population Renewal

Each population is nearly completely replaced by its offspring; only two best individuals from the old population are transfered to the new one (elitism) without having to undergo selection process.

9.1.5 Inner Local Search

If there is a need to use an inner local search (memetic algorithms), the first-improving variant with stochastic informed non-overnight perturbation is used (according to the results of chapter 8).

9.2 Standard Genetic Algorithm

A standard darwinian evolution theory based genetic algorithm with random population initialisation and no inner local search was implemented and subjected to benchmarks. All the following benchmarks were made with 0.8 recombination probability, 0.2 mutation probability and with population of 50.

9.2.1 Comparing Recombination Operators

As we can see in the figure 9.1, uniform recombination operator outperforms the other two in terms of convergence.

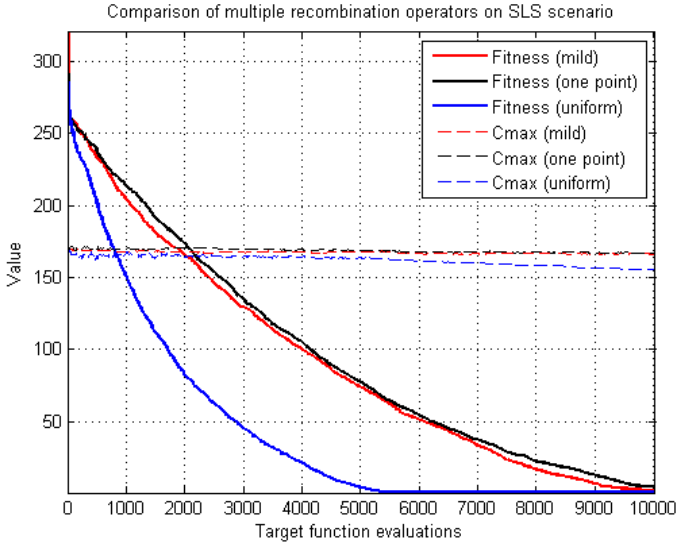


Figure 9.1. Comparison of SGA recombination operators

9.2.2 Comparing SGA with Stochastic Informed First-improving Local Search

In the figure 9.2 we can see, that the stochastic informed non-overnight local search performs far better (in terms of convergence) on the "SLS" scenario than the standard genetic algorithm with uniform recombination.

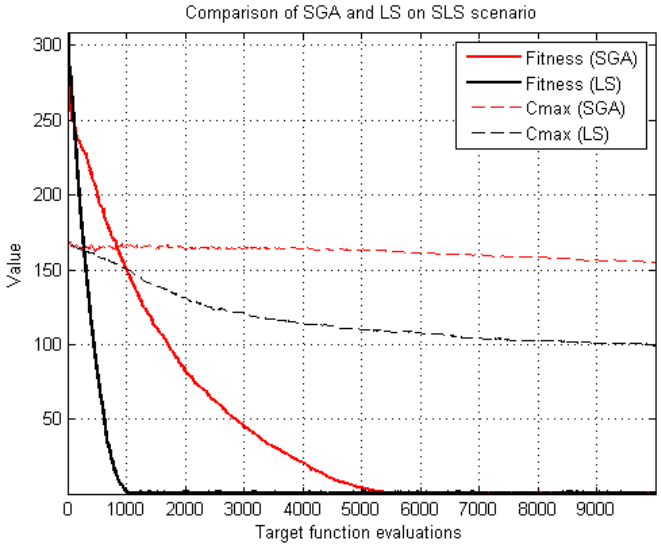


Figure 9.2. Comparison of SGA and stochastic informed first-improving LS on "SLS"

Genetic algorithms are known for their ability to overcome local extrema, so this behaviour was expected on this scenario. The following experiment tries to find out whether this ability will or will not help finding better solution on an unsatisfiable scenario.

9.2.3 SGA and LS on Unsatisfiable Scenario

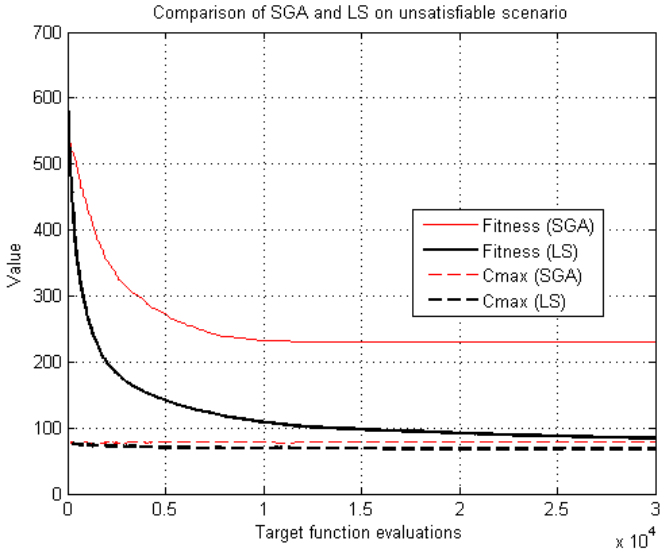


Figure 9.3. Comparison of SGA and stochastic informed first-improving LS on "STU"

Even on the unsatisfiable "STU" scenario, standard genetic algorithm seems to perform worse than the best of the tested local search algorithms.

9.3 Memetic Algorithms

Two memetic algorithms were implemented and subjected to benchmarks—an LTH (low-level teamwork hybrid), which embeds a local search into a standard genetic algorithm, and an HRH (high-level relay hybrid), which is a standard genetic algorithm with initial population built by a local search.

The local search procedure used in these hybrids is stochastic informed non-overnight first-improving local search, which achieved the highest performance in experiments concerning peturbation-based search methods.

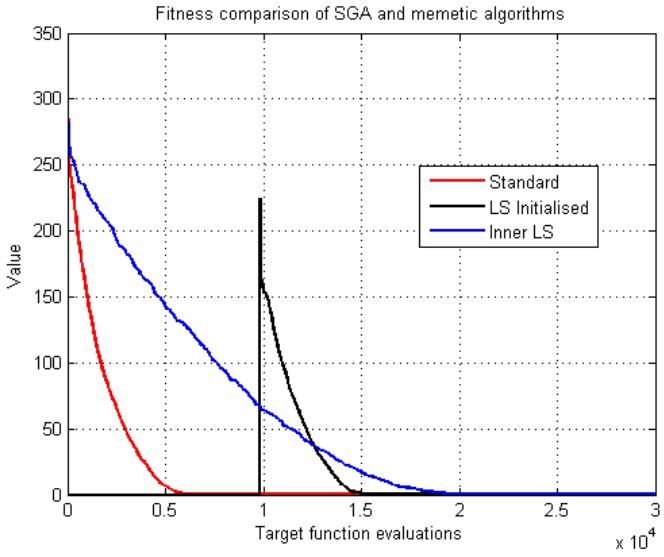


Figure 9.4. Fitness comparison of SGA and memetic LS on "SLS" scenario

9.3.1 Comparing Memetic Algorithms with SGA

As we can see in the figure 9.4, on a small, loose and satisfiable scenario, memetic algorithm with inner local search tends to have slower convergence rate, while ls-initialised hybrid converges very fast. But fitness function evaluation-wise, the standard genetic algorithm still performs better.

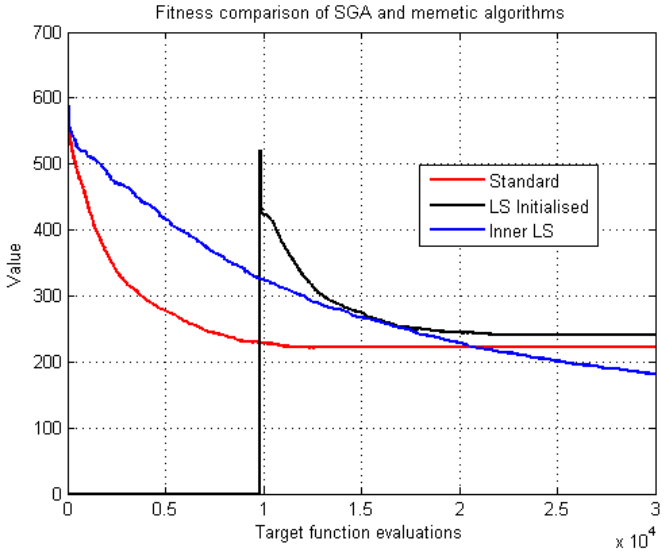


Figure 9.5. Fitness comparison of SGA and memetic LS on "STU" scenario

However, on an unsatisfiable scenario, as we can see in the figure 9.5, the LTH (inner LS) hybrid overcomes the extrema which the remaining two algorithms are unable to overcome.

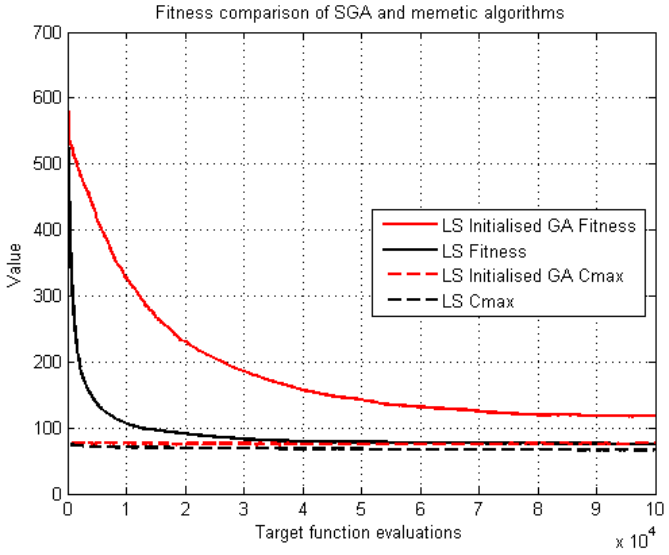


Figure 9.6. Fitness comparison of SGA and memetic LS on "STU" scenario

If we repeat the last benchmark with 10^5 fitness function evaluations and if we compare the results with our best-so-far local search (figure 9.6), we will see that the hybrid which seemed promising is still being outperformed.

9.4 Comparison Tables

	Scenario
Recombination Operator	**STU**
Mild	90.6766
One-point	96.1931
Uniform	86.2798

Table 9.1. Comparison of mild, one-point and uniform recombination operators on SLS scenario (standard genetic algorithm, measured at 30^4 evaluations)

	Scenario	
Algorithm	**STU**	**SLS**
First-improving Local Search	0.4142	82.7829
Standard Genetic Algorithm	0.4453	221.3097
Memetic Algorithm (Ls-initialised)	0.4415	241.4105
Memetic Algorithm (Inner Ls)	0.4403	181.1022

Table 9.2. Comparison of first improving local search (stochastic informed perturbation operator) with standard genetic algorithm, memetic algorithm with ls-initialised population and with memetic algorithm with embedded inner local search (measured at 30^4 evaluations)

Chapter 10
Conclusion on experiments

According to experiments in chapters 8 and 9, the most suitable algorithm for the problem of our concern is **first-improving local search with stochastic neighbourhood and stochastic informed position and teacher perturbation operator**. This algorithm will be subjected to the final benchmark, where the whole timetabling process is simulated.

There are six strategies, which differ in their *plans* (for information about plans, see section 5.4.1):

- **Lock strategy with rerandomisation:** Consists of the following plan chunks:

```
DTAction.Improve,
DTAction.LockRemaining,
DTAction.AdvanceDay,
DTAction.Randomize,
```

- **Lock strategy:** Consists of the following plan chunks:

```
DTAction.Improve,
DTAction.LockRemaining,
DTAction.AdvanceDay,
```

- **Delete strategy with rerandomisation:** Consists of the following plan chunks:

```
DTAction.Improve,
DTAction.DeleteRemaining,
DTAction.AdvanceDay,
DTAction.Randomize,
```

- **Delete strategy:** Consists of the following plan chunks:

```
DTAction.Improve,
DTAction.DeleteRemaining,
DTAction.AdvanceDay,
```

- **Alternative strategy with rerandomisation:** Consists of the following plan chunks:

```
DTAction.Improve,
DTAction.LockRemaining,
DTAction.AdvanceDay,
DTAction.Randomize,

DTAction.Improve,
DTAction.DeleteRemaining,
DTAction.AdvanceDay,
DTAction.Randomize,
```

- **Alternative strategy:** Consists of the following plan chunks:

```
DTAction.Improve,
DTAction.LockRemaining,
DTAction.AdvanceDay,

DTAction.Improve,
DTAction.DeleteRemaining,
DTAction.AdvanceDay,
```

The reason for using so many different strategies is to find out, whether there is any difference between rerandomisation ano no rerandomisation in between particular days.

As we can see in the figure 10.1, fitness function steadily declines in both cases as more students are having their schedules locked. Red dots on the x-axis denote day advancement.

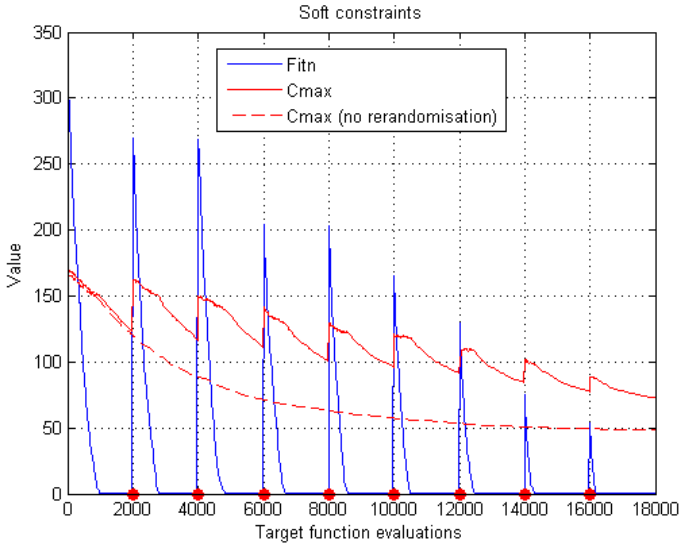


Figure 10.1. Final algorithm with day advancement and rerandomisation (lock strategy)

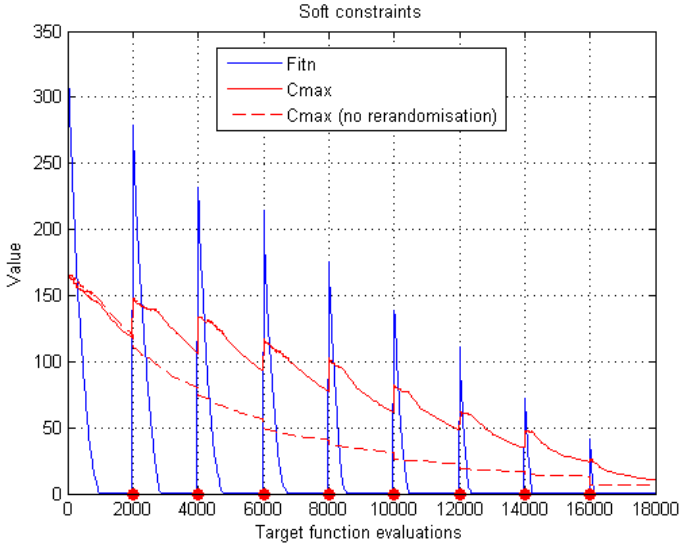


Figure 10.2. Final algorithm with day advancement and rerandomisation (delete strategy)

Figures 10.2 and 10.3 show the same decline on both rerandomisation and non-rerandomisation delete and alternative strategies.

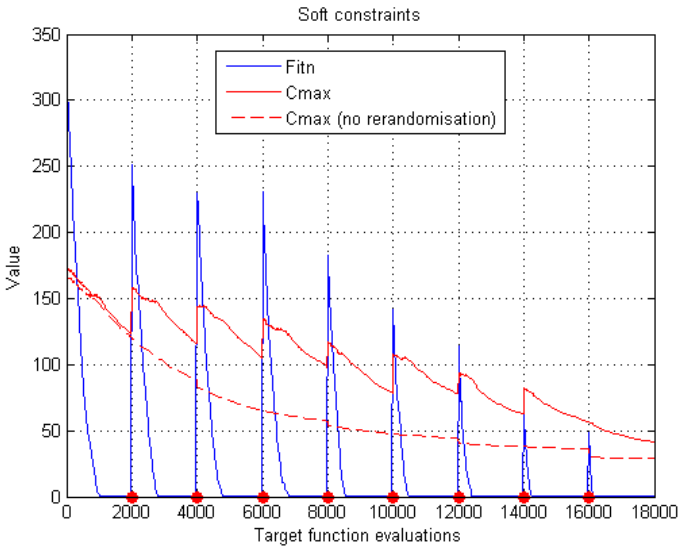


Figure 10.3. Final algorithm with day advancement and rerandomisation (alternative lock/delete strategy)

We conclude that rerandomisation does not improve the final solution.

10.1 Comparison Table

	Strategy		
Rerandomisation	**Lock**	**Delete**	**Alternative**
Enabled	0.3404	0.4748	0.3178
Disabled	0.3090	0.3544	0.2933

Table 10.1. Comparison of rerandomisation and no rerandomisation when applied on different lock strategies (measured at $18 \cdot 10^4$ evaluations)

Chapter 11
Graphical User Interface

In order to visualise algorithm progress and to browse and examine particular timetables of students and teachers, a graphical user interface was designed.

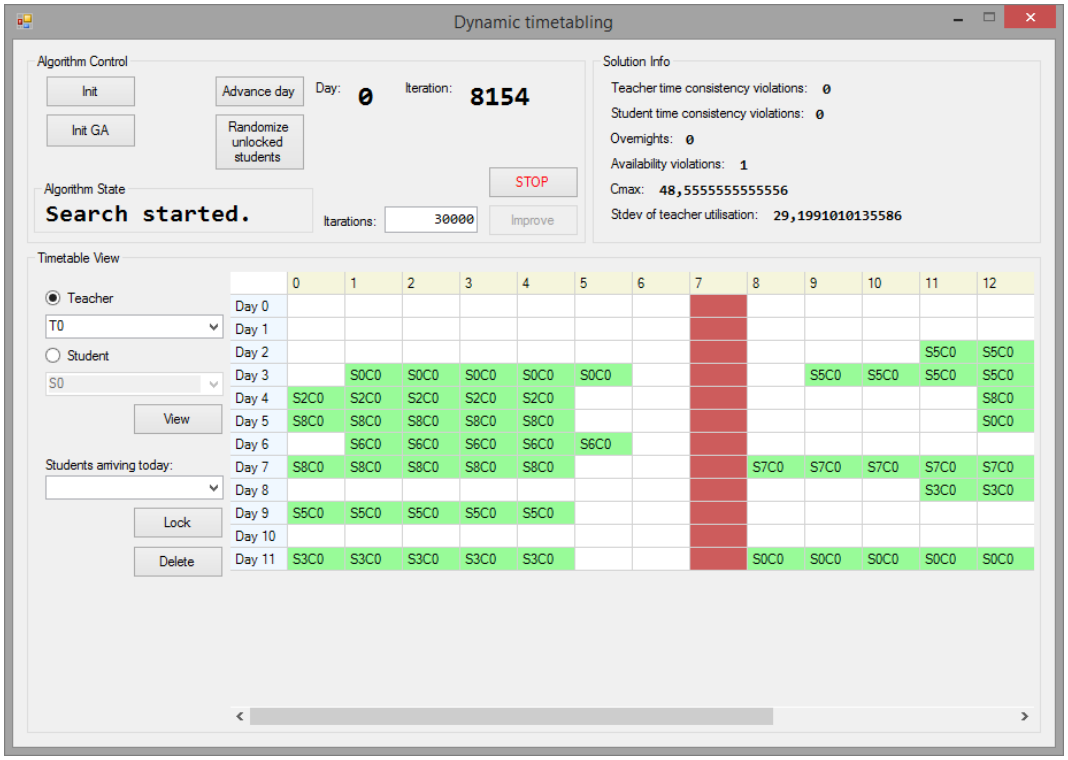


Figure 11.1. Dynamic timetabling graphical user interface

This interface allows users to load scenarios and execute search algorithms upon these scenarios. It provides full functionality of the meta-algorithm, so the user can stop search at any point, lock or delete arriving students, advance day and/or randomize remaining students.

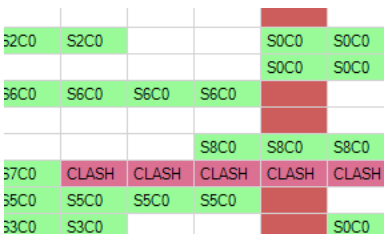


Figure 11.2. Visualisation of clashes

Users can also examine timetables and see the concrete clashes, if there are any.

Chapter 12
Testing the Dynamic Timetabling Library

While functionality of search algorithms themselves is tested by various benchmarks with various input data (in this thesis, this is covered in chapters 8 and 9), the rest of a code base must be covered by unit tests and itegration tests, to make sure, that separate pieces of code work properly on their own, and that code layers interoperate correctly, respectively.

12.1 Unit Testing

Unit testing is a software testing method, in which elementary pieces of code are tested **in isolation**. A unit test of a code piece (a class, a method, etc.) should test whether or not this piece serves its purpose and ignore any other puproses and consequences (this is very closely related to the **Single responsibility principle**, as stated in 4.2). This isolation should also be maintained on code pieces, which depend of lower levels of the software model, for instance: A database accessor is dependent on a database connection provider, but its unit test should **only** concern the accessor's purpose, which is to read data. It should not concern connecting a database, in other words, it should be isolated from the conncetion process. In unit testing, this is done by:

- Strictly obeying **Dependency inversion principle** (4.2)
- Using fakes (see 12.1.1)
- Using mocks (see 12.1.2)

Level of Dependency inversion principle obedience directly affects ability to isolate unit tests.

12.1.1 Fakes (fake objects)

If a piece of code obeys Dependency inversion principle to the greatest extent, it depends solely on abstractions (interfaces and abstract classes in case of C#).

Fakes are types (classes in C#) that extend (implement or extend in C#) these abstract types, but instead of accessing lower level functionality (and thus creating dependencies), they provide "fake" functionality, which is mostly very trivial, but it does suffice for the purpose of unit testing.

Fakes aren't by any means dynamic. They are concrete implementations of abstract types written by a programmer (contrary to mocks).

Let us demonstrate the use of fakes on the simple database accessor example given in 12.1. Suppose we have a class `DatabaseAccessor`, which depends on the inteface `IDatabaseConnector`. A concrete implementation of this interface is injected to the accessor through the constructor. Normally, a database dependent implementation of the interface would be injected, but for the sake of isolation of our test, we create another concrete implementation called `FakeDatabaseConnector`, which does not actually connect to a database, but instead to an ordinary in-memory list of items specified directly in the fake. By injecting this fake to the accessor, we isolate it from an external dependency.

12.1.2 Mocks (mock objects)

Mocks are very similar to fakes, but unlike fakes, mocks are not created by manually implementing interfaces of our concern. They are created dynamically by using very high-level language features, such as reflection or generics.

For our platform of choice (.NET framework 4.5), there exists a widely used framework called "Moq", which provides an extensive mocking library. This library is used by dynamic timetabling library unit test project.

Let us, again, demonstrate the use of mock on a trivial exmple, this time by using actual code, which uses the Moq framework. In the first phase of a test, a mock object must be created:

```
var mockConnector = new Mock<IDatabaseConnector>();
```

Then the mock object must be set up. By setting up, we can specify behaviour of almost any method, any property and any field. We can choose what values methods should return (and even based on their parameter properties) or we can specify a callback function, which will be called instead of the member method.

In our case, we want our method to, for instance, return a filled list of objects, so we set it up that way:

```
mockConnector
.Setup(x => x.ReturnSomething())
.Returns(new List<object> { 1, "something" });
```

And at last we inject the mock to the accessor through the `Object` property of the mock:

```
IDatabaseAccessor = new DatabaseAccessor(mockConnector.Object);
```

Because the Moq framework is used for unit testing of the dynamic timetabling library, we will show a real example of mocking, which uses some more advanced features of the framework.

The situation is, that we need to test that the `RandomisedLocalSearch` raises an appropriate event when it encounters a new individual with better fitness. Using fakes would be very time consuming, because depends on a large stack of abstractions (abstractions which depent on another abstractions), so a full fake stack would have to be created. By using mocks, we can create a mock stack directly in our test method and set up only those features we know that the class under test uses.

So first we need to mock `IData` and `IAuxiliaryVars` (for information about the object model, see 4.2). We dont need to set them up, because they will never be accessed.

```
var mockData = new Mock<IData>();
var mockAux = new Mock<IAuxiliaryVars>();
```

Then we need to set up mock random number generator, which will act deterministicaly and return 0.1 all the time, so we will have control of the algorithm flow.

```
var mockRand = new Mock<IPseudoRandomNumberGenerator>();
mockRand.Setup(x => x.NextDouble()).Returns(0.1);
```

Then we have to create a fitness functor, which would always return the best possible fitness (because we want to raise an event, which should be raised upon encountering a better solution). Because the fitness functor sets the vital `LastFit` parameter on its argument, we need to register a callback function on the mock, which would be called

instead of method `Calculate` and which would always return an inidividual with fitness 0.

```
var mockFitnessFunctor = new Mock<IFitnessFunctor>();
mockFitnessFunctor.Setup(x => x.Calculate(It.IsAny<IIndividual>()))
.Callback<IIndividual>(p => { p.LastFit = 0.0; });
```

Then we need to specify mocks for the rest of the functors. These functors would normally alter provided objects, but because we do not set them up, they will not do so (they will leave provided objects unaltered).

```
var mockMutationFunctor = new Mock<IMutationFunctor>();
var mockRecombinationFunctor = new Mock<IRecombinationFunctor>();
var mockRandomizationFunctor = new Mock<IRandomizationFunctor>();
```

Then we need to create a mock problem context (another layer), which would access the mock we have defined instead of real implementations.

```
var mockContext = new Mock<IProblemContext>();
mockContext.Setup(x => x.RandomizationFunctor)
.Returns(mockRandomizationFunctor.Object);
mockContext.Setup(x => x.Data).Returns(mockData.Object);
mockContext.Setup(x => x.AuxiliaryVars).Returns(mockAux.Object);
mockContext.Setup(x => x.FitnessFunctor)
.Returns(mockFitnessFunctor.Object);
mockContext.Setup(x => x.PositionMutationFunctor)
.Returns(mockMutationFunctor.Object);
mockContext.Setup(x => x.TeacherMutationFunctor)
.Returns(mockMutationFunctor.Object);
mockContext.Setup(x => x.RecombinationFunctor)
.Returns(mockRecombinationFunctor.Object);
mockContext.Setup(x => x.RandomizationFunctor)
.Returns(mockRandomizationFunctor.Object);
mockContext.Setup(x => x.CurrentDay).Returns(0);
mockContext.Setup(x => x.Random).Returns(mockRand.Object);
```

Then we need to specify an input `IIndividual` and an individual which would be returned as a result of perturbation process (the process itself will not be called). The first individual will have its fitness set to worst possible, whereas the second individual (which will be obtained as a result of perturbation) will have its fintess set to the best possible. This setting makes sure, that the event should be raised if the implementation is correct.

```
var mockIndividual1 = new Mock<IIndividual>();
mockIndividual1.Setup(x => x.LastFit).Returns(int.MaxValue);

var mockIndividual2 = new Mock<IIndividual>();
mockIndividual2.Setup(x => x.LastFit).Returns(0);

mockIndividual1.Setup(x => x.MutatePositionOnClone())
.Returns(mockIndividual2.Object);
mockIndividual1.Setup(x => x.MutateTeacherOnClone())
.Returns(mockIndividual2.Object);
mockIndividual2.Setup(x => x.MutatePositionOnClone())
.Returns(mockIndividual2.Object);
mockIndividual2.Setup(x => x.MutateTeacherOnClone())
.Returns(mockIndividual2.Object);
```

Then we need to define a crucial mock—a mock, which will represent an event listener.

```
var betterSolutionListenerMock = new Mock<BetterSolutionEventListener>()
```

We need to register this mock on the algorithm under test:

```
ISearchAlgorithm search = new RandomisedLocalSearch(0);
search.BetterSolutionFound +=
betterSolutionListenerMock.Object.SearchAlgorithm_BetterSolutionFound;
```

Then we eventually run the algorithm under test (with all the underlying layers being mocked and thus making the algorithm under test completely isolated). We can run it for several iterations, because there is loop set up on the mock individuals (first one returns second one on perturbation and the second one always returns itself, so the event should be guaranteed to be raised on the first iteration).

```
search.Improve(mockContext.Object, mockIndividual1.Object, 10);
```

Finally, we shall verify, that the event was raised, by checking, that the event callback method on the mock event handler was called:

```
betterSolutionListenerMock.Verify
(x => x.SearchAlgorithm_BetterSolutionFound(
    It.IsAny<object>(),
    It.IsAny<EventArgs>()
));
```

We verify, that the method `SearchAlgorithm_BetterSolutionFound` was called with first parameter being any subtype of `object` and the second parameter being any subtype of `EventArgs`.

12.2 Integration Testing

Contrary to unit testing, integration testing can extend the level of isolation to any subset of the code base, because its purpose is to test interoperability of the whole stack of software layers.

An integration test should consist of code pieces, which have already been unit tested and the testing should have several stages. It should start at the lowest level (in our case, it is a scenario file parser) and continually add higher layers (in our case, problem context, dynamic timetabling facade and graphical user interface).

The dynamic timetabling library is covered by integration tests.

Chapter 13
Conclusion

The main aim of this thesis was to review algorithms suitable for the problem defined in 1.2, to implement them, to select the best of them according to thorough benchmarks, and to design, implement and test a human resource management tool which would make use of this algorithm.

In this chapter, the goals declared in 1.1.1 are mapped to respective achievements.

13.1 Goals to Achievements Mapping

- SotA algorithms were reviewed (chapter 2) and suitable algorithms were selected (chapter 10)
- These algorithms were analysed (chapters 3, 6 and 7) and subjected to benchmarks (chapters 8 and 9)
- A tool was designed (chapter 4) along with benchmarking framework (chapter 5.4)
- Benchmarking (chapter 5) and testing methodology (chapter 12) was described
- Tool was provided with unit tests, abstraction tests and benchmarks

13.2 Work not Declared in Goals

- Graphical user interface for the tool was designed and implemented
- Statistical matlab-exporting support library was implemented
- Custom ILP problem definition LP-exporting framework was implemented

13.3 Results

After reviewing the state of the art, no suitable algorithms were found, which would suit the problem of our concern. The reviewed algorithms were almost always too specific or only theoretical and the corresponding papers did not provide any implementation, so a novel solution, tailored for the unusual specifications of our problem, was proposed.

The proposed algorithm (called dynamic timetabling meta-algorithm), requires predictions of student arrival dates and curriculums on its input, and makes use of an inner search algorithm, which is supposed to have the ability to improve an existing solution.

Based on statements about state space size, several algorithms were selected to participate in final algorithm selection process:

- Randomised first-improving local search
- Stochastic hillclimbing
- Simulated annealing
- Standard genetic algorithm
- LTH hybrid memetic algorithm

- HRH hybrid memetic algorithm

These algorithms were subjected to thorough benchmarks, and throughout the benchmarking process, several petrurbation, mutation and recombination operators were tested:

- Blind perturbation/mutation operators
- Informed perturbation/mutation operators
- Stochastic informed perturbation/mutation operators
- Uniform recombination operator
- Mild recombination operator
- One-point recombination operator

The benchmarks have shown, that given the objective (fitness) function we were using, there is no need for algorithms to have quality which makes them able to easily overcome local extrema, because the empirical probability of getting stuck in such an exteme was shown (by repeating the experiments) to be very low.

The benchmarks have also shown, that selection and recombination processes in genetic and memetic algorithms do not lead to any improvement in quality of the final solution, nor to better fitness convergence. On the other side, the type of perturbation operator was proved to be crucial.

According to benchmarks, the best algorithm was selected (**first-improving local search with stochastic neighbourhood and stochastic informed position and teacher perturbation operator**) and implemented as an inner search procedure of the dynamic timetabling meta-algorithm.

Then the meta-algorithm itself was subjected to several benchmarks on various execution plans. According to these benchmarks, it was concluded on the settings of the algorithm (no-rerandomisation variant was shown to have better results on all the execution plans).

13.4 Future Work

The dynamic timetabling tool will be provided with the ability to dynamicaly alter the scenario during its execution. It will also be provided with database connectivity and with web-based user interface.

The tool will also be provided with the ability to prioritise user-selected clashes and manually rearrange student and teacher schedules.

References

[1] Marie Demlová. Přednášky z předmětu teorie algoritmů.

[2] Stephen Cook. The complexity of theorem proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971.

[3] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, 1971.

[4] T. B. Cooper and J. H. Kingston. *The complexity of timetable construction problems.* 1995.

[5] W. Herroelen and R. Leus. Robust and reactive project scheduling: A review and classification of procedures. *International Journal of Production Research 42 (8)*, 2004.

[6] W. Herroelen and R. Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research 165*, 2003.

[7] H. Alashwal and S. Deris. Dynamic timetabling using reactive constraint agents, 2007.

[8] A. Elkhyari, Ch. Guéret, and N. Jussien. Solving dynamic timetabling problems as dynamic resource constrained project scheduling problems using new constraint programming tools.

[9] J. Correa and M. Wagner. Lp-based online scheduling: from single to parallel machines, 2007.

[10] S. Daskalaki, T. Birbas, and E. Housos. An integer programming formulation for a case study in university timetabling. *European Journal of Operational Research 153, 117-135*, 2004.

[11] Ch. Guéret, J. Narendra, P. Boizumault, and Ch. Prins. Building university timetables using constraint logic programming.

[12] R. Achá and R. Nieuwenhuis. Curriculum-based course timetabling with sat and maxsat, 2012.

[13] M. Cangalovic and J. Schreuder. Exact colouring algorithm for weighted graphs applied to timetabling problems with lectures of different lengths. *European Journal of Operational Research 51, 248-258*, 1991.

[14] Nelishia Pillay. A survey of school timetabling research. *Annals of operations research DOI 10.1007/s10479-013-1321-8*, 2013.

[15] Yu Zheng, Jing-fa Liu, Wue-hua Geng, and Jing-yu Yang. Quantum-inspired genetic evolutionary algorithm for course timetabling. *Third international conference on genetic and evolutionary computing*, 2009.

[16] J. Nelson, I. Craddock, and K. Imamura. Academic course scheduling by simulated annealing.

[17] J. Frausto-Solís, F. Alonso-Pecina, and J. Mora-Vargas. An efficient simulated annealing algorithm for feasible solutions of course timetabling.

[18] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall, 2002.

[19] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[20] Barbara Liskov. Keynote address – data abstraction and hierarchy, 1988. ACM SIGPLAN Notices 23 (5): 17–34.

[21] Mirko Navara. *Pravděpodobnost a matematická statistika.* Nakladatelství ČVUT, 2007.

[22] Petr Pošík. Přednášky z předmětu evoluční optimalizační algoritmy.

Appendix A
Hardware and Software Specification

A.1 Hardware specifiaction

All benchmarks in this thesis were run of the following hadrware specification:

- Intel Core i5-4690 at 3.50 GHz
- 8 GB memory

A.2 Software specification

All development and benchmarking software was run on:

- Microsoft Windows 8.1 Pro 64bit

Software development platform for the dynamic timetabling algorithm and for statistical testing was:

- Microsoft .NET Framework 4.5 (with C# being the language of choice)
- Microsoft Visual Studio Ultimate 2013 (under MSDNAA licence)

Third party .NET libraries

- More Linq
- NUnit Framework
- Moq Framework

Software used for visualisation purposes:

- Matlab R2009b

Typesetting software:

- TEX Typesetting System

Truly exceptional TEX support software by RNDr. Petr Olšák:

- $\mathcal{CS}$plain
- OPMac plainTEX macros
- CTUstyle plainTEX template

Appendix B
DVD Content

Folder **C_SHARP** — C# source files.
Folder **MATLAB** — MATLAB source files including raw benchmark data.
Folder **TeX** — TEX sources and resources.
Folder **THESIS** — Colour and black-and-white versions of this document.
File **thesis.pdf** — Thesis in with inserted assignment.