# Contents

# List of Tables

# List of Figures

# 1 Introduction

When a man attempts to realize his ideas, he takes a think and invents some solution. It is great if it goes right. But at this moment he may ask himself: ”Can I improve quality of my solution? And if I could, how?” Well, he can simply modify current solution and choose the better one. However, this single step can be repeated. Then every solution modification induces a new solution. It can be better or not again. What modifications and how many should a man does to get an optimal solution? How does the optimal solution look like? Fundamental questions but it is hard to answer in general. For the further exploration of immediate solutions modifications should be combined and the number of them can be vast. Improving the solution by this way we can denote as a combinatorial optimization.

This work is trying to show just another approach how to approximate to an ideal - an optimal solution - preferably faster and more closely.

## 1.1 Problem statement

As we mentioned above we have to deal with combinatorial optimization problems. Some definitions are presented in following sections.

### 1.1.1 Informal definition

One of many definitions says that combinatorial optimization problem is an optimization problem in which the space of possible solutions is discrete instead of continuous and the goal is to find the best possible solution. Instances of such problems that are believed to be hard in general by exploring the usually-large solution space of these instances.

### 1.1.2 Formal definition

An instance of a combinatorial optimization problem can be described in a formal way as a tuple (X,P,Y,f,extr) where

- $X$ is the solution space (on which $f$ and $P$ are defined).
- $P$ is the feasibility predicate.
- $Y$ is the set of feasible solutions.
- $f$ is the objective function.
- extr is the extreme (usually min or max).

The goal is to find solution from $Y$ which minimize or maximize the $f$ function.

### 1.1.3 Combinatorial explosion

The naive way of solving combinatorial problems can be paraphrased as ”generate and test”: In a first step one enumerates all combinations from which one selects all solutions in the second step. In most cases however, ”generate and test” is simply not feasible. This is obvious if the set of combinations is infinite. But even if it is finite then it is usually very large, i.e. exponentially large in size of the problem description. In this case, the generation step runs into a combinatorial explosion (from which it usually returns only several billions of years later). Due to this facts the most of combinatorial problems are NP-hard.

## 1.2 Thesis motivation

A few years ago there was a request for solving the problem of the optimal operation assigning to machines during the process of realizing a technological sequence. The problem classification took some time and several unsuccessful attempts to find polynomial algorithm were made. Finally, the problem was classified as Job Shop Scheduling Problem (section 2.1.3). In its general form it is a NP-complete problem although there are polynomial solutions known only for a few restricted cases. It is the reason we could find no polynomial algorithm. Some approaches for solving our problem were found. The first implementation of the problem was built upon the Genetic Algorithms (Yamada-Nakano approach [21]) but the results were satisfiable only for particular (small) instances. Other approaches were looking for and the approach of Tabu Search were discovered (Nowicki-Smutnicki approach [16]).

After some time spending with the algorithm settings, observing the algorithm behavior and reading dedicated articles (chapter 3) we formulated following facts:

- The performance of the the algorithms is strongly dependent on nature of entry data and parameter settings as well - for another entry data another parameter settings it is required.
- Algorithm convergence is strongly dependent on parameter settings
- Very fast solution improvement at the search process beginning.
- Very slow or no solution improvement after certain time.
- In most of cases premature convergence has occurred.
- Genetic Algorithm is hardly capable to reach the optimum in the tight surroundings.
- Tabu Search can miss promising solution space areas.
- Area of state space exploration differs in compliance with starting solution.

We have had two algorithm implementations and the initial idea was to run one algorithm upon the results of the another one. It was the moment the corner-stone of our framework was laid and because the main task of such framework should have been a coordination of algorithm alternation, we have named the framework **SearchCoordinator**.

When the first version of SearchCoordinator was done and we were able to switch algorithm during the runtime the idea of setting new parameters before next run was born. We started thinking about a strategy of alternation and parameter setting to influence a search convergence.

The motivation of the work presented in this thesis has been to create a robust framework which allows easy implementation of various combinatorial problems and provides flexible environment for experiments.

## 1.3 The goals of the thesis

The major goal of this thesis is the design and practical investigation of framework for providing, using, tracing and switching meta-heuristic algorithms. Consequently, the following topics were investigated.

- Algorithm runtime information extraction and its evaluation
- Prediction of algorithm parameters for next run
- Creating the strategy of alternation

## 1.4 Organization of the thesis

We can divide this thesis into two parts.

The first part, chapters 1-5, is theoretical and it brings information about the problems we are interested in and the approaches commonly used to solve them. Our concept and its similarity (difference) with others is introduced. At the end of this part analysis, design and implementation chapters are presented.

The second part, chapters 6-8, is more practical. All the experiments we have performed are described here. Some observation description and suggestions are presented. At the end the experimental results are summarized and an alternation strategy is formulated.

In the rest the conclusions remain.

# 2 Theory

## 2.1 Challenging problems

Satisfiability Problem (SAT), Traveling Salesman Problem (TSP), Job Shop Scheduling Problem (JSSP) - these are well-known combinatorial problems that are often used for performance demonstration all sorts of approaches - as well as in this thesis. We have implemented all the three problems in order to compare results among the problems and as a demonstration of the implementation facility with our framework.

For each problem, data for a performance comparison exist - benchmark data. The data differ in a size and in complexity as well. We have used benchmarks for all problems that are commonly available - see appendix B for more details.

### 2.1.1 Satisfiability Problem

The formal definition of SAT requires the function to be expressed in the so-called conjunctive normal form (CNF), i.e., as an AND of ORs. In this form, each OR term is called a clause and acts as a constraint on the possible values of its variables. For example the clause (A OR  B OR C) is satisfied by all 0-1 value assignments to A, B, and C except A = C = 0 and B=1. A formula in CNF can, therefore, be viewed as a system of simultaneous constraints in the space of binary assignments to its variables.

In complexity theory, the Boolean satisfiability problem (SAT) is a decision problem. An instance of the problem is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true?

The restriction of SAT to instances where all clauses have length k is denoted k-SAT. Of special interest are 2-SAT and 3-SAT: 3 is the smallest value of k for which k-SAT is NP-complete, while 2-SAT is solvable in linear time.

### 2.1.2 Traveling Salesman Problem

The traveling salesman problem (TSP) asks for the shortest route to visit a collection of cities and return to the starting point.

The undirected problem can be stated in graph theory terms as follows. Let $G = (V, A)$ be a graph, where $V = \{v_1, ..., v_n\}$ is a vertex (or node) set and $A = \{(v_i, v_j)|v_i, v_j \in V, i \neq j\}$ is an edge set, with a non-negative cost (or distance) matrix $C = (c_{ij})$ associated with A. The TSP consists in determining the minimum cost Hamiltonian cycle on the problem graph, where the symmetry implied by the use of undirected edges rather than directed arcs can also be expressed by stipulating that $c_{ij} = c_{ji}$

### 2.1.3 Job Shop Scheduling Problem

The job-shop problem is defined formally as follows. There are a set of jobs $J = \{1, ..., r\}$ , a set of machines $M = \{1, ..., m\}$ and a set of operations $O = \{1, ..., o\}$. Set O is decomposed into subsets corresponding to the jobs. Job $j$ consists of a sequence of $o_j$ operations indexed consecutively by $(l_{j-i} + 1, ..., l_{j-i} + o_j)$ which should be processed in that order, where $l_j = \sum_{i=1}^{j} o_i$ is the total number of operations of the first $j$ jobs, $j = 1, ..., r$ $(l_0 = 0)$, and $l_j = \sum_{j=1}^{r} o_j = o$ Operation $i$ must be processed on machine $\mu_i \in M$ during an uninterrupted processing time $p_i > 0, i \in O$. We assume, without loss of generality, that any successive operations of the same job are going to be processed on different machines. Each machine can process at most one operation at a time. A feasible schedule is defined by start times

| job | Operations routing (processing time) | | |
|---|---|---|---|
| 1 | 1(3) | 2(3) | 3(3) |
| 2 | 1(2) | 3(3) | 2(4) |
| 3 | 2(3) | 1(2) | 3(1) |

Table 2.1: A 3 x 3 Problem

$S_i > 0, i \in O$, such that the above constraints are satisfied. The problem is to find a feasible schedule that minimizes the makespan $\max_{i \in O}(S_i + p_i)$ (see [16]).

An example of a 3 x 3 JSSP is given in Table 2.1 (see [21]). The data includes the routing of each job through each machine and the processing time for each operation (in parentheses). Figure 2.1 shows a solution for the problem represented by "Gantt-Chart". The JSSP can

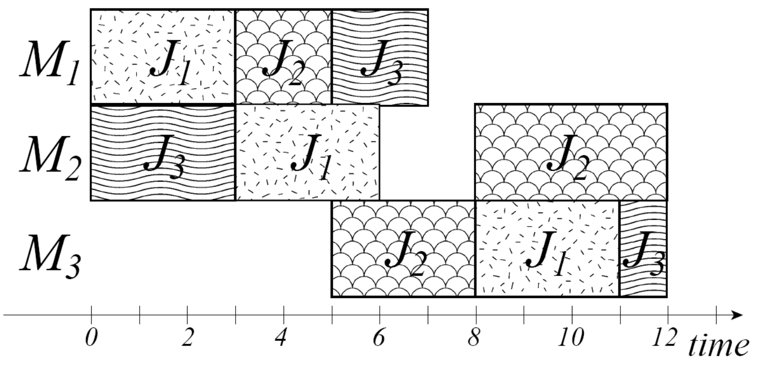


Figure 2.1: A Gantt-Chart representation of a solution for a 3 x 3 problem

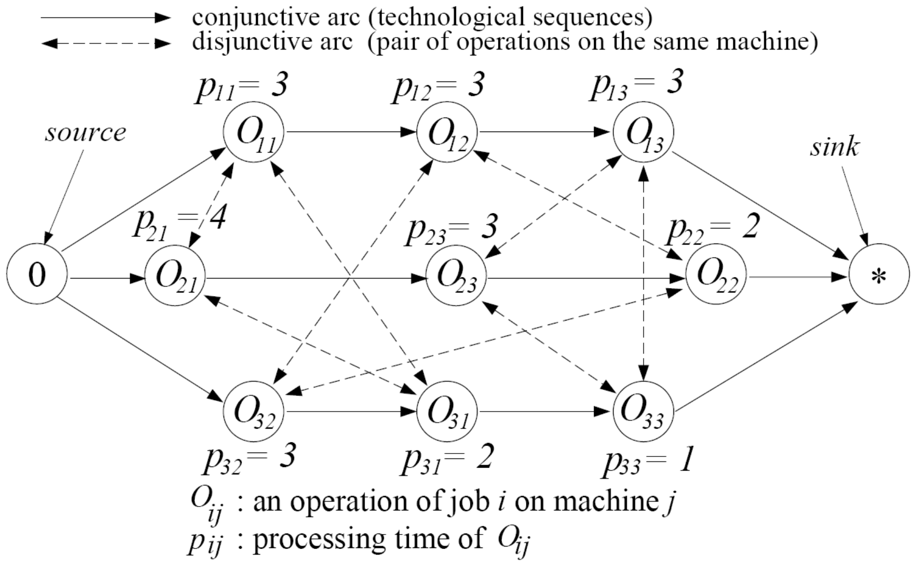

conjunctive arc (technological sequences)
disjunctive arc (pair of operations on the same machine)
source
sink
0
*
p11 = 3
p12 = 3
p13 = 3
p21 = 4
p23 = 3
p22 = 2
p32 = 3
p31 = 2
p33 = 1
O11
O12
O13
O21
O23
O22
O32
O31
O33
Oij : an operation of job i on machine j
pij : processing time of Oij

Figure 2.2: Disjunctive graph of a 3x3 problem

be described by a disjunctive graph $G = (V, C \cup D)$, where $V$ is a set of nodes representing operations of the jobs together with two special nodes, a *source*(0) and a *sink*$\star$, representing the beginning and end of the schedule, respectively. $C$ is a set of conjunctive arcs representing technological sequences of the operations. Figure 2.2 shows this in a graph representation for the problem given in Table 2.1.

The makespan is given by the length of the longest weighted path from *source* to *sink* in the disjunctive graph. This path is called a critical path. Any operation on the critical path is called a critical operation.

It is possible to decompose the critical path into a number of blocks where a block is a maximal sequence of adjacent critical operations that require the same machine.

## 2.2 Searching methods

In this section we will talk about searching so we must know what we are searching for and where. For this purpose we define *state space* as a set of all possible values of the state vector of a system.
Formally, it can be defined as a tuple [N, A, S, G] where:

- N is a set of states
- A is a set of arcs connecting the states
- S is a nonempty subset of N that contains start states
- G is a nonempty subset of N that contains the goal states.

In a matter of fact the state space is what is being investigated by the searching methods.

### 2.2.1 Planning vs. scheduling

Basically, there are two classes of problems connected with search space term. The planning and the scheduling.

**Planning** Open any recent AI textbook [11, 8, 13] and you find a chapter on planning. Fundamentally, planning is a synthesis task. It involves formulating **a course of action** to achieve some desired objective or objectives. Simply, we are familiar with an initial and final state but we don't know the optimal sequence of defined operation to applied to the initial state to get the final one.

**Scheduling** Scheduling is defined as the problem of assigning limited resources to tasks over time to optimize one or more objectives. Simply, we are familiar with an initial state (solution) but we don't know the final state (we don't know optimal configuration of final state vector).

When we have a look on the formal definition of state space described above we can assign to particular symbols following meanings. N (a set of states) is space of all possible resource assignments. A (arcs connecting the states) is a neighbor function which one solution transform to an another. S (nonempty subset of N that contains start states) is a set of initial resource configurations. G (nonempty subset of N that contains the goal states) is a set of final resource configuration. In this case we rather speak about solution space instead of state space.

In chapter 2.1 we mentioned the problems we are interested in. They are all the scheduling problems. Both classes have their specific approaches and so only scheduling related ones will be described in this thesis.

### 2.2.2 Brute force

Brute-force search, also known as exhaustive search, is the simplest and crudest of all possible heuristics. It is a primitive approach which relies on the computer's processing power. This means to exhaustively search through all possible combinations (checking every single point in the function space) until a solution is found. In context of combinatorial problems we can easily presume that complexity of such approach is an exponential or even factorial function. Due to that facts brute force method is not attractive for us.

### 2.2.3 Heuristics

Heuristic is the art and science of discovery. The word comes from the same Greek root as "Eureka!". It means "to find" or "pertaining to finding". When we use a heuristic function, in fact, we exploit some problem-specific information, knowledge or experience to reduce state space cardinality.

#### 2.2.3.1 Random solutions

The first heuristic is also the simplest: choose any random solution. This may look like a ridiculous solution. What good is a random solution? By itself, it is not much good. But consider repeatedly finding a random solution and taking the best solution as our choice. Suddenly the repeated solution doesn't look too silly. First, because it is so simple to find random solutions, we might be able to find hundreds of solutions in the time it takes another heuristic to find one.

#### 2.2.3.2 Hill climbing, Greedy algorithm

Hill-climbing is a heuristic in which one searches for an optimum combination of a set of variables by varying each variable one at a time as long as the result increases and then stopping when the result decreases.

The basic idea is to always head towards a state which is better than the current one. An algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for many instances of other problems. The main drawback is disability overcome the local optima.

### 2.2.4 Meta-heuristics

Meta-heuristic algorithms are algorithms which, in order to escape from local optima, drive some basic heuristic: either a constructive heuristic starting from a null solution and adding elements to build a good complete one, or a local search heuristic starting from a complete solution and iteratively modifying some of its elements in order to achieve a better one. The meta-heuristic part permits the low-level heuristic to obtain solutions better than those it could have achieved alone, even if iterated. Usually, the controlling mechanism is achieved either by constraining or by randomizing the set of local neighbor solutions to consider in local search (as is the case of Simulated Annealing [7] or Tabu Search [5]), or by combining elements taken by different solutions.

#### 2.2.4.1 Genetic algorithms

Genetic algorithms are a general methodology for searching a discrete solution space in a way that is similar to process of natural selection procedure in biological systems. The algorithm is a remarkably general one, and it can be applied to different problems if following conditions

are met:
a) solutions to the problem can be expressed in form of a string of characters
b) a "fitness" criterion, which in some way quantities the quality of a solutions, can be computed for any valid string
c) strings in which "part" of a good solution is present are rewarded by allocation of a higher fitness than "average" strings

An algorithm is implemented as a computer simulation in which a population of abstract representations (called chromosomes or the genotype or the genome) of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached. [18, 20, 12]

#### 2.2.4.2 Tabu Search

Tabu Search is optimization method, belonging to the class of local search techniques. It enhances the performance of a local search method by using memory structures. Tabu Search is generally attributed to Fred Glover[5].

Rather than being a single algorithm, the Tabu search would be better described as a set of concepts that the algorithms falling under this heading share.

Tabu Search uses a local or neighborhood search procedure to iteratively move from a solution $x$ to a solution $x'$ in the neighborhood of $x$, until some stopping criterion has been satisfied. To explore regions of the search space that would be left unexplored by the local search procedure each solution as the search progresses. The solutions admitted to $N^*(x)$, the new neighborhood, are determined through the use of special memory structures. The search now progresses by iteratively moving from a solution $x$ to a solution $x'$ in $N^*(x)$.

The neighborhood function is the most important part of the tabu search algorithm, as it significantly affects both the running time and the quality of solutions.

Perhaps the most important type of short-term memory to determine the solutions in $N^*(x)$, also the one that gives its name to tabu search, is the use of a tabu list. In its simplest form, a tabu list contains the solutions that have been visited in the recent past (less than n moves ago, where n is the tabu tenure). Solutions in the tabu list are excluded from $N^*(x)$. Other tabu list structures prohibit solutions that have certain attributes (e.g. traveling salesman problem (TSP) solutions that include certain arcs) or prevent certain moves (e.g. an arc that was added to a TSP tour cannot be removed in the next $n$ moves). Selected attributes in solutions recently visited are labeled tabu-active. Solutions that contain tabu-active elements are tabu. This type of short-term memory is also called recency-based memory.

Tabu lists containing attributes are much more effective, although they raise a new problem. When a single attribute is forbidden as tabu, typically more than one solution ends up being tabu. Some of these solutions that must now be avoided might be of excellent quality and might not have been visited. To overcome this problem, aspiration criteria are introduced which allow overriding the tabu state of a solution to include it in the allowed set. A commonly used aspiration criterion is to allow solutions which are better than the currently best known solution.

We have employed principles described above in our work. See chapter 5.1.8 for implementation details.

#### 2.2.4.3 Simulated annealing

Simulated annealing (SA) [7] is a generic probabilistic meta-algorithm for the global optimization problem, namely locating a good approximation to the global optimum of a given function in a large search space.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the temperature), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local minima, which are the bane of greedier methods.

#### 2.2.4.4 Ant colony optimization

Ant Colony Optimization (ACO) [3] is a paradigm for designing meta-heuristic algorithms for combinatorial optimization problems. The essential trait of ACO algorithms is the combination of a priori information about the structure of a promising solution with a posteriori information about the structure of previously obtained good solutions.

This algorithm is inspired by observation on real ants. Individually each ant is blind, frail and almost insignificant yet by being able to co-operate with each other the colony of ants demonstrates complex behaviors. One of these is the ability to find the closest route to a food source or some other interesting land mark. This is done by laying down special chemicals called pheromone. As more ants use a particular trail, the pheromone concentration on it increases hence attracting more ants.

## 2.3 Paired student *t-test*

For the comparison of two sets of results we will use paired student *t-test*.

The paired student *t-test* is generally used when measurements are taken from the same subject before and after some manipulation. The manipulation in our case is any change of the way the algorithm performs the searching. Consider results $X$ and results $Y$. The first set of results $X$ are results before the algorithm modifications. After the modifications we get results $Y$ and that results we test against results $X$.

We will briefly describe the test procedure. We assume the result populations have approximately normal distribution.

Two paired data selections

| $x_i$ | $y_i$ | $d_i$ |
|---|---|---|
| $x_1$ | $y_1$ | $x_1 - y_1$ |
| $x_2$ | $y_2$ | $x_2 - y_2$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $x_n$ | $y_n$ | $x_n - y_n$ |

$$\overline{d} = \frac{1}{n} \sum_{i=1}^{n} d_i$$

$$s_d = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (d_i - \overline{d})^2}$$

Null hypothesis $H_0 : \mu_1 = \mu_2$

Formula for the *t-test*: $\boxed{t = \frac{\overline{d}}{s_d}\sqrt{n}}$

Decision rules
if $|t| > t_{\alpha(n-1)} \Rightarrow$ we deny $H_0 : \mu_1 = \mu_2$ and we admit $A : \mu_1 \neq \mu_2$
if $t > t_{2\alpha(n-1)} \Rightarrow$ we deny $H_0 : \mu_1 = \mu_2$ and we admit $A : \mu_1 > \mu_2$
if $t < -t_{2\alpha(n-1)} \Rightarrow$ we deny $H_0 : \mu_1 = \mu_2$ and we admit $A : \mu_1 < \mu_2$

For our experiments we will assume significance level $\alpha = 0.05$.

For a better illustration, instead of presenting exact values of *t-test* formula we will use symbols. The symbol meanings are explained in Table 2.2.

| Symbol | Meanings |
|---|---|
| $-$ | measuring has not been performed |
| $\nearrow$ | the quality of results $Y$ has improved significantly |
| $\searrow$ | the quality of results $Y$ has worsened significantly |
| $\approx$ | the quality of results $Y$ is approximately the same as $X$ |

Table 2.2: *t-test* symbol meanings

# 3 The theory related to SearchCoordinator

This chapter compares some significant approaches with our one. The key features of our framework can be described as follows:

**Global solution inventory** We maintain a global solution population the algorithms are working with. Each algorithm can modify this population by its own search process. It is the way how the algorithms share solution information.

**Algorithm abstraction** We will assume that each algorithm works like a black box. Black box term is commonly used for something (a device, system or object) what it is viewed primarily in terms of its input and output characteristics. An algorithm takes a solution on its input, its parameters are set, and after given iteration number puts an another (better) solution on its output.

**Meta-runtime environment** It is the environment where the algorithms are run. The main search process is composed from particular algorithm search processes.

**Algorithm behavior observing** Each algorithm sends current information about its search process during its performing. The information are collected and various statistics are calculated.

**Search strategy** The information collected during the algorithm computation are evaluated and then next steps of computation are chosen. This process is performed within the frame of particular search strategy.

All these topics will be discussed later in next chapters. Extractions from selected special articles are presented in next sections and correspondences with our key features are included.

> The correspondences are described below the extraction paragraph with double-line on the left.

## 3.1 A Parallel Tabu Search and Its Hybridization with Genetic Algorithms

See [14] for details.

This paper proposes two parallel meta-heuristics. One is a cooperative parallel tabu search which incorporates the historical information exchange among processors in addition to its own local search on each processor. The other is a cooperative parallel search between genetic algorithm and tabu search processes

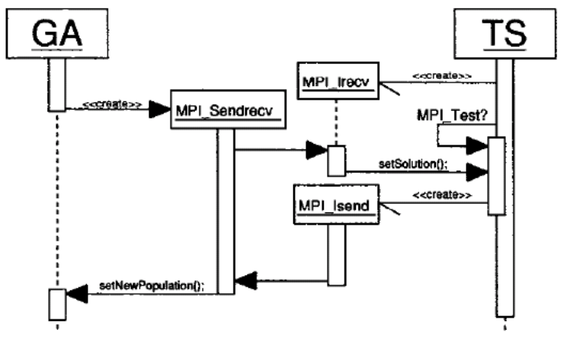
**Cooperative Parallel Tabu Search (cPTS)** The aim of parallelization for TS is to carry out its searching as fast as possible using multiprocessors. Each processor stores its own historical information that were explored during the search process, e.g., the obtained minimum solution so far and also the contents in the tabu memory. cPTS performs to share and use these historical information among processors. The obtained best solution at each processor is treated like a chromosome and a crossover operation is performed to generate new starting solutions. Two new solutions generated by EX (the edge recombination crossover) are assigned to the processors as new current solutions.

The cooperative partner is selected from the neighbor processors. In this work two types of partner selection are presented. One is the best-solution monopolized selection in which each processor cooperates with the processor with the best-solution found so far. The other is the tournament selection in which two processors selected randomly and the processor with better solution is selected for the partner.

> Basic difference is in the fact that our concept is single-threaded. The idea of more search processes is similar.

**Hybrid Parallel Meta-Heuristic (HPMH)** Two processors, each processor carries out GA-based or TS-based searching process independently. By message passing among two meta-heuristic searching processes, their exploration of searching space is cooperated. The initiator of a message passing is GA-process. When its population is occupied by an elite chromosome which has the highest fitness value in the population, GA-process sends the elite to TS-process. Since all chromosome is the same in this case, GA-process cannot evolve by itself, that is, an execution of a genetic operation is useless. Therefore the GA-process initiates a message passing and waits for the TS-process to send new population under the blocking communication. On the other hand, since TS-process is not necessary to interrupt its own searching for a sending message of GA-process, we employ the non-blocking communication for TS-process. When an elite chromosome is sent from GA-process, TS-process replaces the current searching solution by the migrant elite chromosome if the migrant is better than the current solution. Furthermore TS-process sends solution candidates found by its own searching so far. See Figure 3.1.

> There are two independent algorithms: (1) Tabu search (2) Genetic algorithm. They share founded solutions with each other. The correspondences there is in using principles of genetic algorithms and tabu search simultaneously. There are two phases performing repeatedly: (1) Tabu search phase (2) Phase modifying the best solutions by genetic operators.



**A Message passing between GA and TS**

Figure 3.1: Genetic algorithm and Tabu search cooperation

## 3.2 Evolutionary Potential Timetables Optimization by Means of Genetic and Greedy Algorithms

See [1] for details.

The article describes solution for Timetabling problem using genetic and greedy algorithms. Authors have proposed the Genetic engine. The core of this engine is based on alternation of genetic and greedy algorithms.

- **Variation of alternations number.** On a population is applied GA until a convergence level (G generations). Then, the greedy algorithm is applied on the best individual.
- **Variation of population size.** GA iterations number has been fixed at 200 and alternations number of genetic and greedy algorithms at 50.
- **Variation of individual number handled by the greedy algorithm.**

- Alternation of genetic and greedy algorithms are applied on **all** individuals of the population generated by the **last** GA generation.
- Alternation of genetic and greedy algorithms are applied on **the best** individual (which has the minimum fitness value) generated by the **last** GA generation

The proposed technique described in this work demonstrates the improved performance by the alternation of genetic and greedy algorithms when applied to the combinatorial timetable problem to obtain a global optimal timetable. See Figure 3.2

Alternation of search methods. One method works with solution founded by the second one. Certain parameters are traced during the running time. Several strategies of alternation are used.

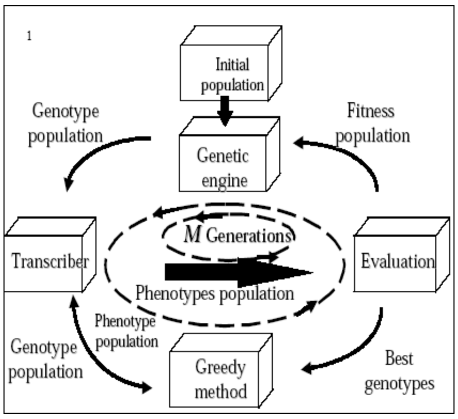


Figure 3.2: Alternation of Genetic and Greedy algoritm

## 3.3 Genetic Algorithm Restricted by Tabu Lists in Data Mining

See [10] for details.

In the approach, described in article, GA is discussed aside the following paradigms: simulated annealing, tabu search, artificial neural networks and expert system. Ideas of Tabu search was used for the extension of GA.

Genetic algorithm - proposed by this approach - uses two lists of restrictions: Long Lists with size $m$ and Short List with size $n$. The values of $m$ and $n$ are adjusted according to the problem. The lists: (a) store the best individuals of the previous generations; (b) maintain the elitism and the diversity of the population; and (c) avoid the convergence to a local minimum.

The idea behind the algorithm is that, at the end of each generation, the best individual will be stored in both lists. When the next generation begins and new individuals are selected for the reproduction, the lists of restrictions do not allow that two similar individuals are selected at the same time. The similarity among individuals is given by a function to measure that distance, either in the phenotypic or genotypic space.

Genetic Algorithm is used for location an area of state space, Tabu Search then for the solution intensification.

## 3.4 A Hybrid Search Based on Genetic Algorithms and Tabu Search for Vehicle Routing

See [17] for details.

The article introduces two-phased hybrid search for the vehicle routing problem with time windows (VRPTW). A global customer clustering phase based on genetic algorithm and a **post-optimization** local search technique based on tabu search. They observed that the GA is effective in setting the number of vehicles to be used while the tabu search is more effective in reducing the total distance traveled by the vehicles. Thus the hybrid search technique is more more suitable for the multiobjective optimization for the VRPTW than applying either the GA or TS independently.

|| Genetic Algorithm and Tabu Search advantages are combined in order to get more feasible solution.

## 3.5 Combination Genetic/Tabu Search Algorithm For Hybrid FlowShops Optimization

See [19] for details.

In the article Genetic algorithm and Tabu search are discussed for solving FlowShop problem. With respect to disadvantages both algorithms have the algorithm they have developed is a combination of both approaches, and tries to combine the positives of the two methods.

The algorithm combines the principles of the two approaches. At initiation it creates a set of random valid solutions, and for several iterations it optimizes them using tabu search-based method. Then the algorithm applies genetic principles to the set of solutions, and this creates a new generation of solutions. The solutions retained from the previous generation keep the associated tabu lists; new solutions begin with clear tabu lists. The process of several tabu-principles iterations followed by a genetic-principles iteration continues until computation termination criteria are met.

|| In fact, all the ideas presented in the article are familiar with our concept.

## 3.6 Genetic Local Search for Job Shop Scheduling Problem

See [2] for details.

This paper describes an hybrid algorithm (GTS) combining Genetic Algorithms and Tabu Search for the JSSP.

**GLS Template**

1. Generate initial population
2. Execute for every individual an initial optimization by applying local search
3. Assign fitness to every individual
4. Select individuals for recombination
5. Apply the recombination operator producing a new generation of offspring
6. Optimize every new offspring by applying local search
7. Insert the offspring in the population and reduce it to the original size
8. Repeat the steps from 3 to 7 until a stop criterion is met

Computational experiments have shown that on large size instances the GA counterpart makes indeed the difference. The best mix of TS and GA for those instances is half and half.

The concept described in the article is a special case of our approach. We can set up Genetic Algorithm followed by Tabu Search repeatedly.

## 3.7 Genetic Algorithms for Job-Shop Scheduling Problems

See [21] for details.

The authors of this work - Takeshi Yamada and Ryohei Nakano - are distinguished researchers in branch of Job Shop Scheduling Problem and Genetic algorithms. The first serious application of GAs to solve the JSSP was proposed by Nakano and Yamada using a bit string representation and conventional genetic operators. Although this approach is simple and straightforward, it is not very powerful.

In their work they have proposed some improvements and concluded that: *The approaches by GAs are suitable for middle-size problems; however, it seems necessary to combine each with other heuristics such as the shifting bottleneck or local search to solve larger-size problems.*

This article extraction is included here for supporting the idea of hybrid algorithms or algorithm alternation.

# 4 Analysis and Design

One of our framework requirements is an ability to handle various algorithms in an uniform way. The first thing we have to struggle out is just the variance of the algorithms. Despite all differences we may have observed some common characteristics. Lets have a look at them in a more detailed way.

## 4.1 Common algorithm characteristics

From the meta-heuristic algorithms we expect some common behavior. Firstly, we put an initial solution in, set some parameters and run it. After performing the given number of iterations we get an optimized solution. We will use such abstraction for the algorithm manipulation. Then the main terms we will work with are: Solutions, Algorithms, Parameters.

### 4.1.1 Solutions

Each algorithm takes one or more initial solutions on its input. Problem-specific input data are encoded into problem-specific solution instances.

The problem encoding is a crucial difference the algorithms have. Whether we want to run one algorithm on results of the other we must be able convert one problem encoding to another. A trivial approach is just to perform the conversion between each two algorithms (this concept is used in [1] - Transcriber) but this is not feasible enough for more than two algorithms. If we had two algorithms and we would like to add the third then we would have to figure out two conversions to and from the remaining algorithms. Four algorithm conversions in sum. More

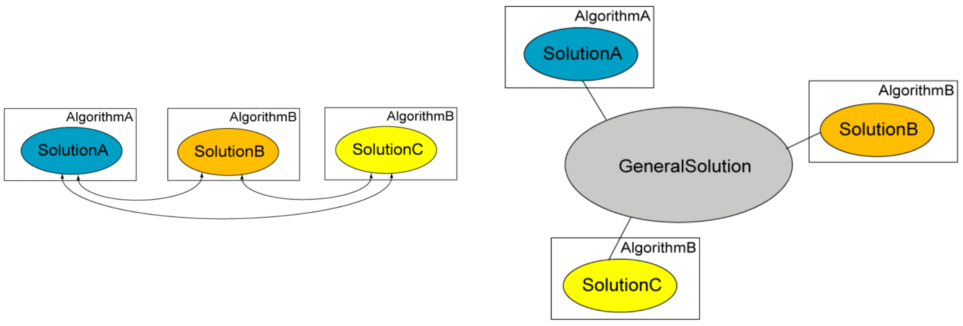


Figure 4.1: Algorithm conversions

feasible approach is to figure out a general description of the problem solution. Then each algorithm is responsible for the conversion from and into this general description "encoding". See Figure 4.1 for both cases. We have used the concept with general solution.

### 4.1.2 Algorithm

According to principles of object-oriented design [4] the algorithm should solve the problem of given class in general and provide interfaces to implement and thus specify the problem-dependent details. This is outlined in Figure 4.2. User of such algorithm just have to implement appropriate interfaces and the internal algorithm structure is hidden to him. It is a very useful view for us because we can see the algorithm as a black box with respect to its interfaces.

We have employed the *Adapter* design pattern [4] to wrap up the algorithm into our abstract view - AlgorithmAdapter (see Figure 4.3). Functionality of the embedded algorithm is accessible through the AlgorithmAdapter interface. The interface is briefly predicated in the opening of this section - it will be discussed later. Now, when we have the algorithms unified by the

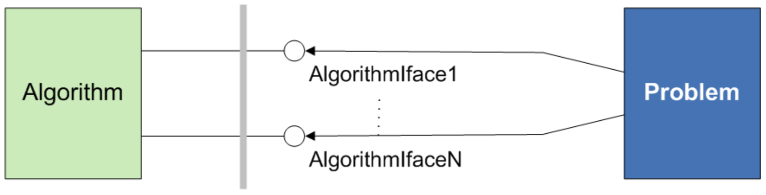


Figure 4.2: Basic algorithm

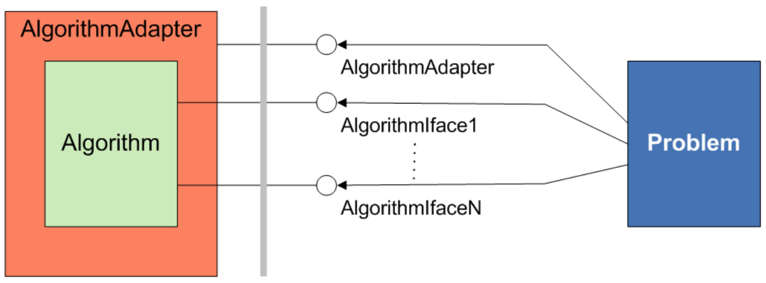


Figure 4.3: Basic algorithm with adapter

adapters we can create the collection of the adapters. We have named the collection manager the SearchCoordinator. See Figure 4.4.

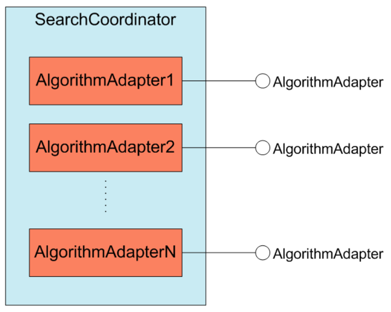


Figure 4.4: SearchCoordinator holds adapters

## 4.2 SearchCoordinator framework

The SearchCoordinator manages the collection of AlgorithmAdapters, controls the run of computation and coordinates the collaboration of all framework's components. A schema can be seen in Figure 4.5 - it is the synthesis of previous three figures with some additions. We can see the SearchCoordinator (problem independent) side on the left and the Problem (problem dependent) side on the right. We haven't spoken about DataStore and AILogic yet.

### 4.2.1 DataStore

DataStore is - as name says - a central store of solutions in general form as we mentioned in section 4.1.1. It is used for a storing the solutions during an algorithm alternation phase. Although DataStore is a part of SearchCoordinator only the Problem side can manipulate with data it keeps.

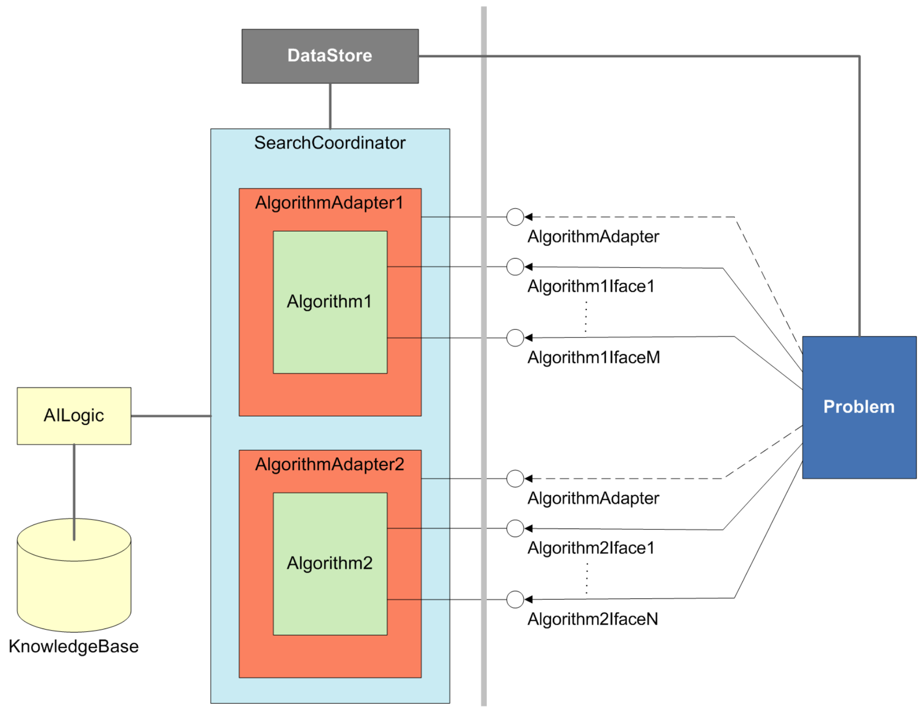


Figure 4.5: SearchCoordinator schema

### 4.2.2 AILogic

AILogic serves for reasoning the search process. It holds all information about problem which is solving. The SearchCoordinator puts algorithm runtime information into the AILogic and gets reasons for next steps of searching process. The search strategy is located here.

# 5 Implementation

## 5.1 Class overview

The SearchCoordinator framework will get concrete contours in this section. We will describe classes it is comprised of and explain the purpose of each class. In fact we will describe a class diagram shown in Figure 5.1 - classes and their relationships are outlined by UML class diagram. For a better illustration, colors of the class headers are pursuant to Figure 4.2 colors.

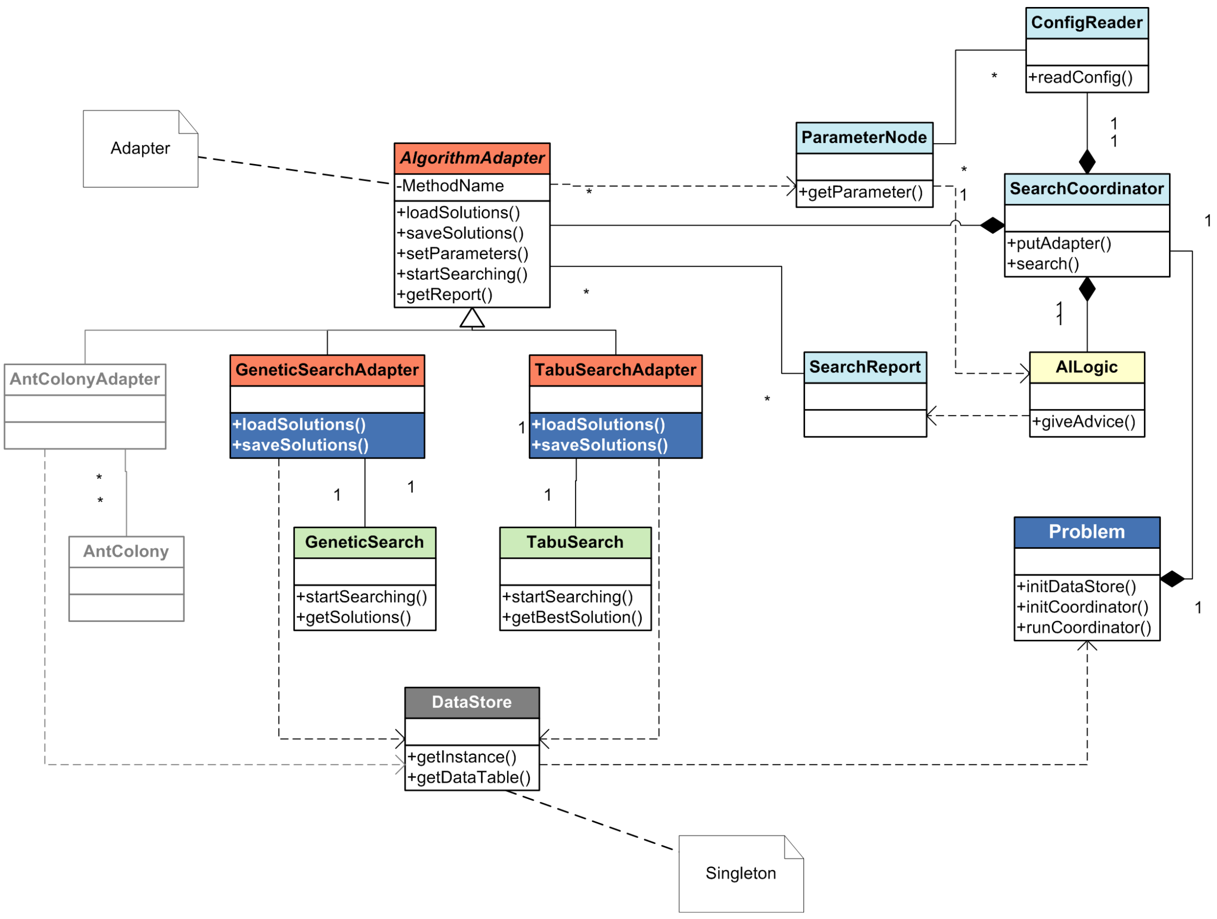


Figure 5.1: SearchCoordinator class diagram

### 5.1.1 SearchCoordinator

SearchCoordinator is a basic class of our framework and so it has the same name. We have already mentioned it in section 4.2. This class executes several duties. At first it manages the collection of AlgorithmAdapters. There is `putAlgorithmAdapter()` method for this purpose - it takes the algorithm adapter on its input and stores it in the internal collection. The second method `search()` initiates and run the main computation. The search cycle is shown in Figure 5.2.

Before running a particular algorithm AILogic is requested for a reason (will be described later). The reason contains the name of next algorithm to be run and its parameters. Then the parameters are pass into the algorithm and the algorithm is run. After finishing an algorithm

```
while (EndCondition == false)
{
    Parameters pn = AILogic.getReason();
    AlgorithmAdapter adapter = algorithmAdapters.get(pn.getAlgorithmName());
    adapter.setParameters(pn);
    adapter.startSearching();
    AILogic.putReport( adapter.getReport());
}
```

Figure 5.2: Pseudocode of the SearchCoordinator cycle

report is put back into AILogic. This process can be seen in Figure 5.3 - UML sequence diagram.

We can see that SearchCoordinator class knows nothing about the Problem and about the algorithms it manages as well. This can be done by virtue of the abstraction we have done in previous chapter.

### 5.1.2 ConfigReader

The initial parameters are read from xml file by `ConfigReader` class and stored to a structure called `ParameterNode` - it is a tree. `ParameterNode` structure is used for handling the parameters in general - not only during the initial configuration but during a reconfiguration (chapter 7) as well.

### 5.1.3 DataStore

`DataStore` class serves for storing the data during the computation. It manages structures - `DataTable`s. `DataTable`s are accessible through their names. They contains `DataRow`s and they contain `DataCell`s. The only one object of `DataStore` can be instanced in the application and that's why `DataStore` has been designed according to design pattern *Singleton* (see [4]).

### 5.1.4 AlgorithmAdapter

There was something spoken about `AlgorithmAdapter` in previous sections. The main purpose of this interface is to abstract various meta-heuristic algorithms and to convert their interfaces to unified one. `AlgorithmAdapter` exposes five methods to implement.

- `loadSolutions()` - method for reading solutions from `DataStore`
- `saveSolutions()` - method for saving solutions into `DataStore`
- `setParametes()` - this method passes parameters into a algorithm
- `startSearching()` - search process is run by this method
- `getReport()` - method for retrieving algorithm report

`loadSolutions()` and `saveSolutions()` are only methods user has to implement on the Problem side (see Figure 4.5) In Figure 5.1 `AlgorithmAdapter` is drawn as a superclass for `GeneticSearchAdapter`, `TabuSearchAdapter` and `AntColonyAdapter`. The Ant colony algorithm is not used in our framework (but in future may be) and so the `AntColonyAdapter` is just outlined there.

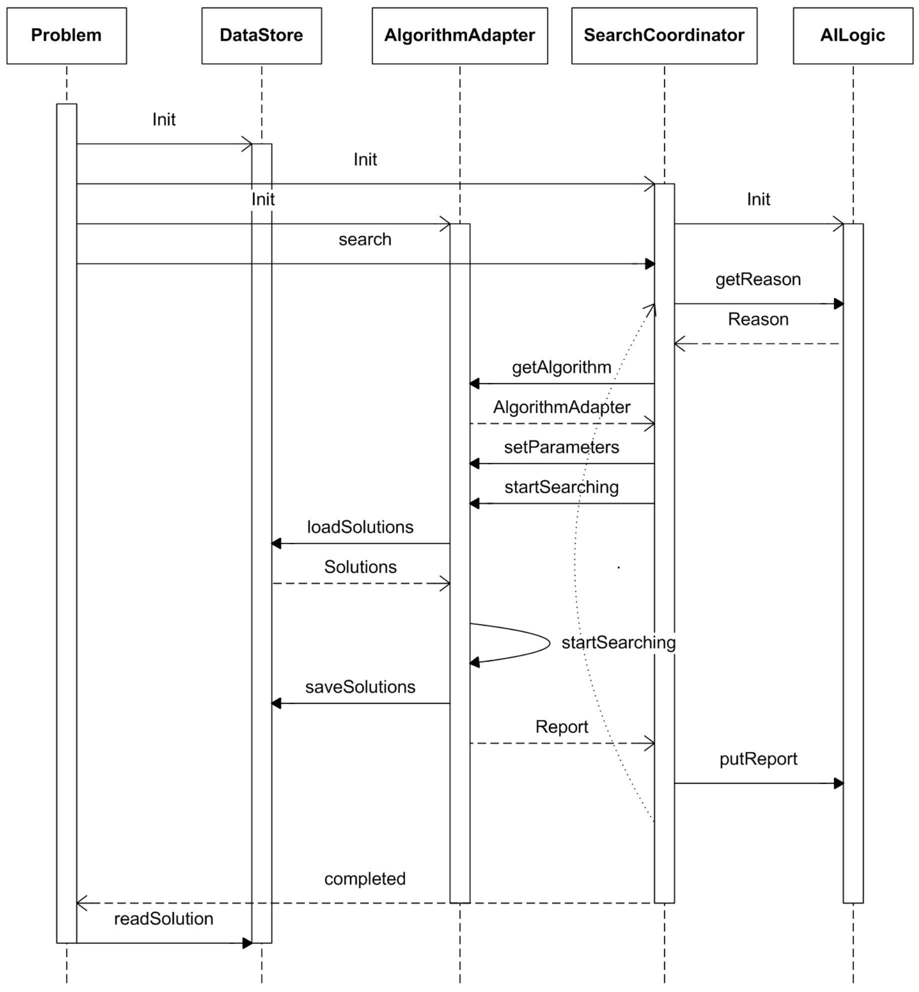


Figure 5.3: SearchCoordinator sequence diagram

### 5.1.5 GeneticSearchAdapter

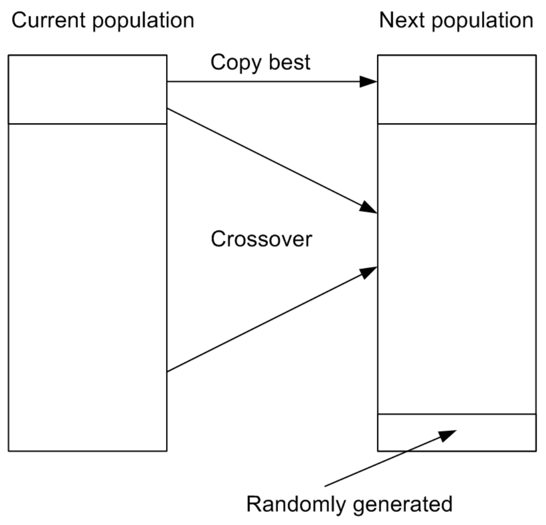
`GeneticSearchAdapter` is an abstract class with partial implementation. The methods `startSearching()`, `getReport()` and `setParametes()` are implemented and it is not recommended to override them. The main tasks of this class are the initialization of Genetic Algorithm classes, parameter settings and running the algorithm. Constructor of this class accepts all implemented interfaces required for initializing Genetic Algorithm. The `startSearching()` method just recall a method that starts the genetic search process.

### 5.1.6 TabuSearchAdapter

The implementation of `TabuSearchAdapter` is similar to the `GeneticSearchAdapter` implementation. The abstract class initiates TabuSearch algorithm and convert its interface to `AlgorithmAdapter` interface.

### 5.1.7 GeneticSearch

This class is an entry point to the subsystem which is the implementation of simple Genetic Algorithm. Genetic Algorithm implemented by class `GeneticSearch` works with the population of subjects loaded from `DataStore`. In Figure 5.4 the modification of current population is outlined and main algorithm loop as well. A simplified class diagram is shown in Figure 5.5.



```
Genetic Algorithm
{
    Generate initial population Pt
    Evaluate population Pt
    While stopping criteria not satisfied
    Repeat
    {
        Select elements from Pt to copy into Pt+1
        Crossover elements of Pt and put into Pt+1
        Mutation elements of Pt and put into Pt+1
        Evaluate new population Pt+1
        Pt = Pt+1
    }
}
```

Figure 5.4: Genetic Algorithm

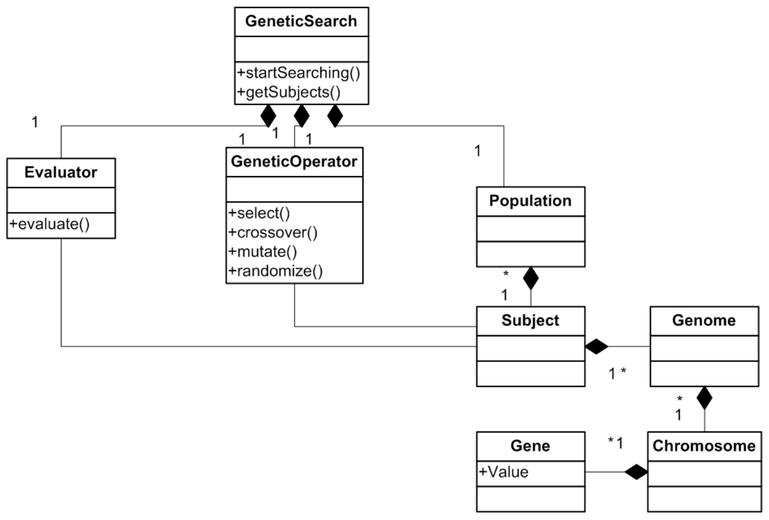


Figure 5.5: GeneticSearch class diagram

`Evaluator` class is an objective function. It evaluates quality of subjects. `GeneticOperator` is responsible for all genetic operations. Although it contains a basic implementation, it can be easily extended and its methods can be overridden. For a higher algorithm commonness we have provided the `Subject` class with `Genome`. `Genome` is a chromosome container. In default implementation we use only one chromosome. The `Chromosome` class manages a collection of `Gene`s that each holds an integer value.

The algorithm parameters are following:

- Population count - the number of subjects in a population
- The number of iterations - iterations to be done
- Crossover length - a maximal length of chromosome fragment to be crossover
- Elitism rate - it gives the maximal number of elite subjects
- Mutation rate - it gives how many subjects will be mutated at most

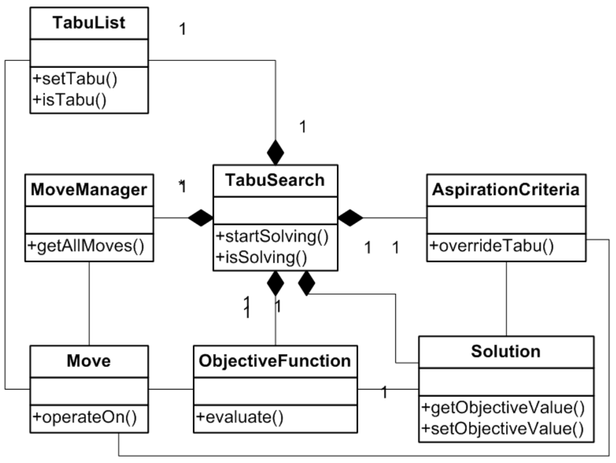
- Mutation length - it gives the maximal number of mutated genes
- Random subject rate - the maximal number of new random subjects

The maximal number serves as upper bound for a given parameter and the value is set randomly every iteration.

### 5.1.8 TabuSearch

```
TabuSearch()
{
    sol = InitialSolution()
    aspiration = AspirationCriteria()
    bestCost = Cost(sol)
    bestSolution = sol
    tabuList = Empty()
    move = null
    while keepSearching()
    {
        N(sol) = GetNeighborMoves()
        if N <> null then
            move = GetBestMove(N, tabuList, aspiration)
        UpdateTabuList(move)
        newSol = ApplyMove(sol, move)
        if Cost(newSol) > bestCost then
            bestSolution = newSol
        sol = newSol
    }
    return bestSolution
}
```

(a)



(b)

Figure 5.6: TabuSearch pseudocode (a) and class diagram (b)

We started from Coin-Or implementation OpenTS [6]. OpenTS asks to define basic elements common to all tabu searches and then performs iterations based on these elements. The key elements must be defined are: Solution structure, Objective function, Tabu list, Move, Move manager. See Figure 5.6.

OpenTS uses these elements to search the solution space. Given a starting, or current, solution, the move manager is asked to generate a list of moves for the iteration. OpenTS uses the objective function to determine the value of the solution that would result from each of these moves. With the help of the tabu list, OpenTS determines which move is the best, and that move operates on the starting, or current, solution which results in a new current solution.

### 5.1.9 SearchReport

This class holds the information about last search process. These information are collected and evaluated inside `AILogic` class.

### 5.1.10 AILogic

This class maintains the strategy the search process is guided by. Both `AILogic` and `SearchParameter` will be described later.

### 5.1.11 Problem

The Problem box in Figure 5.1 must'nt be the only class but set of classes that implement a given problem. In general, several tasks are required from the Problem application. We have already spoken about the implementations of specific problem interfaces. Another three task are to be done:

**DataStore initialization** It consists in creating all data tables and filling them with appropriate data - generally, with initial solutions and another information needed for evaluation of them.

**SearchCoordinator initialization** This task consists in creating SearchCoordinator and in instancing all algorithm adapter classes with their classes that implement the particular algorithm interfaces.

**Running the SearchCoordinator** The purpose of this task is to run the SearchCoordinator computation.

That's all we can say about the Problem implementation in general. Another facts depend on the implementation of specific problems will be described in next section.

## 5.2 Problem implementations

To solve some problem using the SearchCoordinator a user have to implement interfaces of particular algorithms and interfaces of SearchCoordinator as well. According to our experience the most reasonable way how to proceed is as follows:

- Implementation of all algorithm interfaces
- Functionality testing of the algorithm
- Implementation the AlgorithmAdapter interface (the only interface required by SearchCoordinator)
- Putting the algorithm into SearchCoordinator

The user have to proceed this simple sequence for all algorithms he wants to use with SearchCoordinator. Following sections describe the implementations of the algorithm interfaces for all problems. The implementation of the AlgorithmAdapter interfaces is purely technical issue and it is not so interesting.

### 5.2.1 SAT

#### 5.2.1.1 Genetic Algorithm

According to SAT problem description in section 2.1.1 our deal is to find such truth values of each logic variable to satisfy a given equation.

**Encoding** This problem encoding is simple. The chromosome of length $n$ looks as follows

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | ... | $x_{n-3}$ | $x_{n-2}$ | $x_{n-1}$ | $x_n$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | ... | 1 | 0 | 0 | 1 |

It is a sequence of genes.

The gene values are 0(false) and 1(true). The order of gene in the chromosome is an index of the logic variable and gene value is its value.

**Solution evaluation** While thinking of problem encoding was easy, solution evaluation is little bit complicated. Table of clauses is needed. During the evaluation process this table is walking through and each clause is evaluated according to variable values retrieved from the proper genes. Two values are significant for a objective function. The first is a number of unsatisfied clauses and the second is a number of variables in the unsatisfied clauses. Therefore, there are two values on output of objective function (at the end of solution evaluation). The first value has a higher priority (we are attempting to minimalize a number of unsatisfied clauses).

*Example:*

$F(X_1) = \{5, 3\} < \{4, 5\} = F(X_2)$
$F(X_3) = \{5, 3\} > \{5, 4\} = F(X_4)$

**Genetic operators**

**Selection** Roulette rank selection is used.

**Crossover** Two point crossover is used.

**Mutation** Inversion of several gene values is provided.

#### 5.2.1.2 Tabu Search

**Encoding** The encoding is similar to the Genetic Algorithm encoding. Instead of using chromosome an array of bits is used.

**Moves** A move is inversion of certain bit. Random moves of the given number are generate each iteration.

**Solution evaluation** Here we can meet two occurrences. The move is set or not at the beginning of evaluation. If not we have to calculate objective value of a whole solution. Otherwise, we simulate the move and update objective value.

### 5.2.2 TSP

In this case we work with 2D coordinates of cities and the goal is to find such city number sequence as we get the shortest path.

#### 5.2.2.1 Genetic Algorithm

**Encoding** We assign an unique number to each city. Then the tour is defined as sequence of that city numbers. We have to store city numbers and their 2D coordinates in DataStore table. Gene values are indexes into the table. Length of chromosome is same as number of cities.

Chromosome of length *10* looks as follows

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 6 | 4 | 5 | 2 | 1 | 8 | 0 | 9 |

**Solution evaluation** The evaluation process works with two adjacent gene values - indexes. City coordinates on current indexes are read from Datastore's table and Euclidian distance is calculated. The distance is added to the path distance. The path distance is then returned as objective value.

**Genetic operators**

**Selection** Roulette rank selection is used.

**Crossover** Two point crossover is used. This action is problematic with a standard crossover. Invalid path may occur after the crossover. Thus, we used Partially Matched Crossover (PMX) [9].

**Mutation** Swap of several gene pairs is provided.

#### 5.2.2.2 Tabu Search

**Encoding** Similarity with Genetic Algorithm representation can be seen again. Instead of a chromosome we used an array of city numbers. An another data structure but the same meaning.

**Moves** A move is a swap of any two values. A given number of such moves are generated for each iteration.

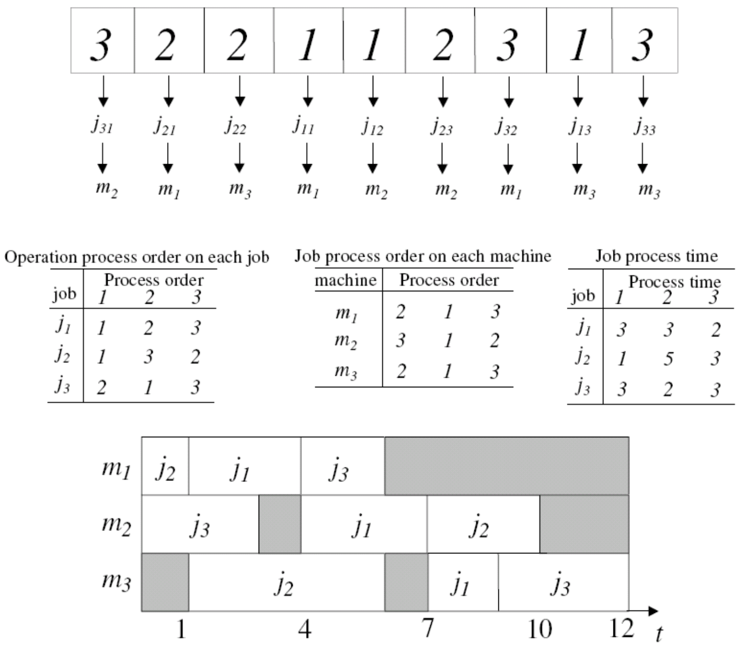
**Solution evaluation** Here we can meet two occurrences again. The move is set or not at the beginning of evaluation. If not we have to calculate objective value of a whole solution - the length of the whole path. Otherwise, we simulate the move and update objective value - path distance.

### 5.2.3 JSSP

#### 5.2.3.1 Genetic Algorithm

**Encoding** We have used the operation-based representation method proposed in [21]. This representation encodes a schedule as a sequence of operations. Each gene denotes each job, and a chromosome can be decoded in to a schedule indirectly (unfortunately it is not one-to-one mapping). In case all jobs have the same number of operation a chromosome consists of $n \times m$ genes where $n$ is the number of jobs and $m$ is the number of operations per job. The chromosome length the same as the number of all job's operations.

Each gene is scheduled in appearance order according to job-machine processing order table. For example, in case of the chromosome, [3,2,2,1,1,2,3,1,3] of Figure 5.7, first gene "3" denotes operation 1 of job 3. Second gene "2" denotes operation 1 of job2. Third gene "2" denotes operation 2 of job 2. We schedule these operations using according to job-machine processing order table.



3 2 2 1 1 2 3 1 3
$j_{31}$ $j_{21}$ $j_{22}$ $j_{11}$ $j_{12}$ $j_{23}$ $j_{32}$ $j_{13}$ $j_{33}$
$m_2$ $m_1$ $m_3$ $m_1$ $m_2$ $m_2$ $m_1$ $m_3$ $m_3$
$m_1$ $j_2$ $j_1$ $j_3$
$m_2$ $j_3$ $j_1$ $j_2$
$m_3$ $j_2$ $j_1$ $j_3$
1 4 7 10 12 $t$

Operation process order on each job

| job | Process order 1 | 2 | 3 |
|---|---|---|---|
| $j_1$ | 1 | 2 | 3 |
| $j_2$ | 1 | 3 | 2 |
| $j_3$ | 2 | 1 | 3 |

Job process order on each machine

| machine | Process order | | |
|---|---|---|---|
| $m_1$ | 2 | 1 | 3 |
| $m_2$ | 3 | 1 | 2 |
| $m_3$ | 2 | 1 | 3 |

Job process time

| job | Process time 1 | 2 | 3 |
|---|---|---|---|
| $j_1$ | 3 | 3 | 2 |
| $j_2$ | 1 | 5 | 3 |
| $j_3$ | 3 | 2 | 3 |

Figure 5.7: JSSP: An example of chromosome decoding

**Solution evaluation** The schedule is reconstructed from the chromosome and the makespan is calculated. The schedule makespan is returned as a objective function value.

**Genetic operators**

**Selection** Roulette rank selection is used.

**Crossover** Two point crossover is used. This action is problematic with a standard crossover. Invalid schedule (a schedule with disrupted operation precedences) may occur after the crossover. Thus, we used Partially Matched Crossover (PMX) [9].

**Mutation** Swap of several gene pairs is provided.

#### 5.2.3.2 Tabu Search

**Encoding** The encoding is the same as used in Genetic Algorithm.

**Moves** In this work, we use the approach of Nowicki and Smutnicki (1996)[15]. The approach is based on a critical path (see section 2.1.3) operation swaps. If a job predecessor and a machine predecessor of a critical operation are also critical, then choose the predecessor (from among these two alternatives) which appears first in the operation sequence. The critical path thus gives rise to the following neighborhood of moves. Given $b$ blocks, if $1 < l < b$, then swap only the last two and first two block operations. Otherwise, if $l = 1(b)$ swap only the last (first) two block operations (see Figure 5.8). In the case where the first and/or last block contains only two operations, these operations are swapped. If a block contains only one operation, then no swap is made.

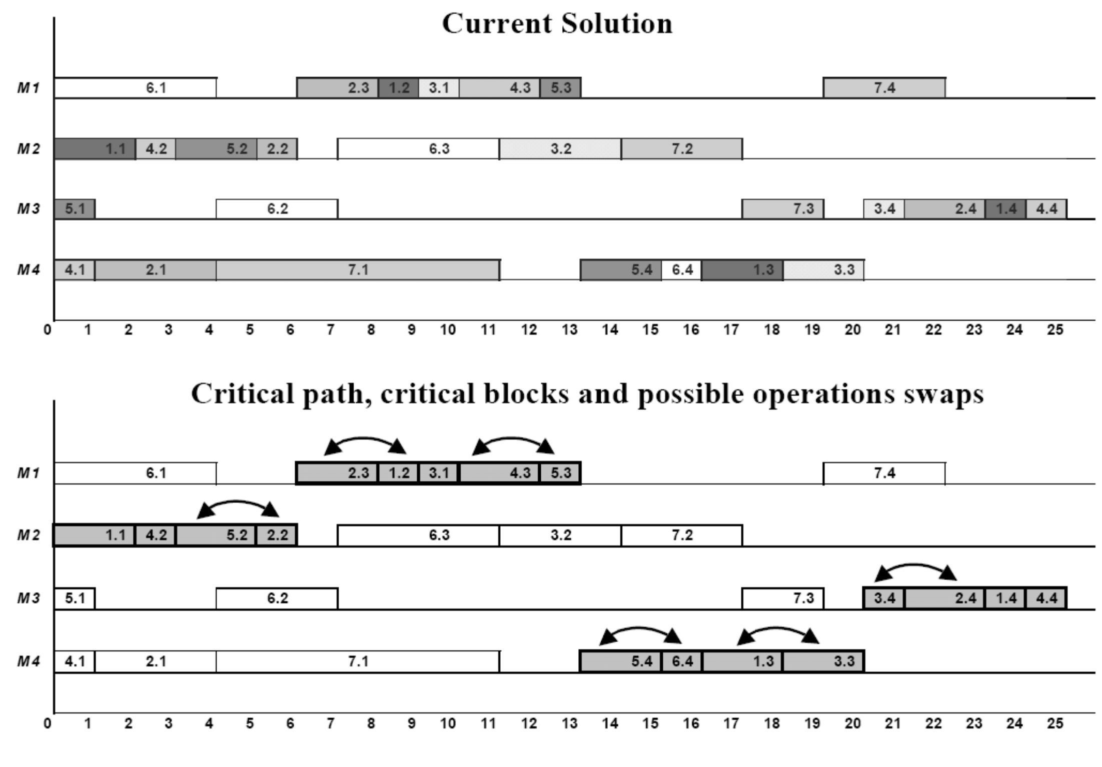


Figure 5.8: JSSP: Neighborhood of Nowicki and Smutnicki (1996).

**Solution evaluation** Firstly, the critical path changes are applied on encoded solution (see section 5.2.3.1). Then the schedule is reconstructed from the chromosome and the makespan is calculated. The schedule makespan is returned as a objective function value.

# 6 Functionality evaluation

## 6.1 Parameter settings

In this section we will demonstrate functionality of basic building blocks - standalone Genetic Algorithm and Tabu Search. Firstly, we will set up parameters experimentally and then we will show an *iteration strength* of the algorithms. At the end of chapter we will use some statistics to compare both algorithms.

The optimal parameter settings of particular algorithms are not crucial for our experiments. However, we are attempting to design a treatment with maximal performance - it can be assumed we achieve it with the best performance of particular algorithms.

### 6.1.1 Parameter settings for Genetic Algorithm

In section 5.1.7 we have introduced our Genetic Algorithm implementation. Our implementation needs to set up seven parameters before it runs. We proposed four sets of that parameter values - configurations (see Table 6.1) and we were interested in results we get with each parameter set.

| | C1 | C2 | C2 | C2 |
|---|---|---|---|---|
| Population size | 100 | 50 | 50 | 100 |
| Iteration count | 500 | 500 | 500 | 500 |
| Elite subjects | 0.1 | 0.1 | 0.1 | 0.15 |
| Mutation rate | 0.05 | 0.1 | 0.1 | 0.15 |
| Max. chrom. mutation | 0.1 | 0.1 | 0.3 | 0.15 |
| Max. chrom. crossover | 0.6 | 0.6 | 0.4 | 0.6 |
| Random subject rate | 0.1 | 0.1 | 0.05 | 0.1 |

Table 6.1: Parameter configurations for Genetic Algorithm

We ran the algorithm on three instances of each problem. The results are shown in Figure 6.1. Graphs of convergence for selected problem are included for illustration. We have achieved the best performance with the first configuration (C1) in most of cases. Thus, we will use this configuration in next experiments. Graph in Figure 6.1 shows an influence of initial parameter settings on quality of results. With improper parameter settings a premature convergence may occurs.

### 6.1.2 Parameter settings for Tabu Search

Our Tabu Search implementation needs to set two parameters - iteration number and tabu list length. As well as we considered four parameter configurations for Genetic Algorithm we proposed four configurations for Tabu Search - see Table 6.2. It is clear that the number

Table 6.2: Parameter configurations for Tabu Search

| | C1 | C2 | C2 | C2 |
|---|---|---|---|---|
| Iteration count | 500 | 500 | 500 | 500 |
| Tabu list length | 7 | 20 | 30 | 50 |

of iteration has a direct influence on solution quality and so we were interested only in tabu

| SAT | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| uf100-01 | **3** | 6 | 8 | 7 |
| uf250-01 | 32 | 34 | 45 | **24** |
| f600 | **136** | 139 | 175 | 145 |

| TSP | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| berlin52 | **12974** | 13961 | 15007 | 14504 |
| eil76 | **1225** | 1307 | 1579 | 1325 |
| kroA100 | **86294** | 91870 | 103499 | 91930 |

| JSSP | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| ft10 | **1046** | 1090 | 1094 | 1048 |
| la22 | **1111** | 1140 | 1141 | 1118 |
| swv20-01 | **3148** | 3233 | 3350 | 3282 |

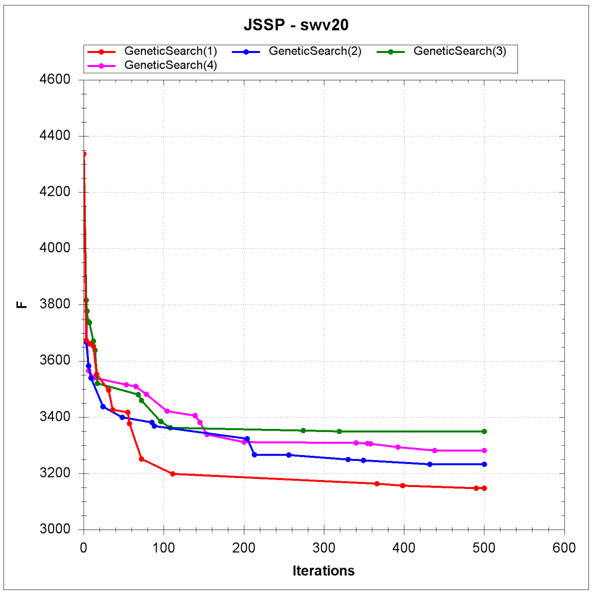


Figure 6.1: Parameter settings: results for Genetic Algorithm

| SAT | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| uf75-01 | 1 | 1 | **0** | 5 |
| uf100-01 | 5 | 4 | **3** | 4 |
| uf250-01 | 9 | **7** | 9 | 10 |
| f600 | 30 | **27** | 29 | **27** |

| TSP | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| berlin52 | 12805 | 12726 | **12364** | 12805 |
| eil76 | 971 | 923 | **866** | 867 |
| kroA100 | 53680 | 53284 | **43868** | 44674 |

| JSSP | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| ft10 | 1035 | 1047 | **1020** | 1034 |
| la22 | 1035 | 1043 | **1026** | 1037 |
| yn1 | 1068 | 1072 | 1056 | **1038** |

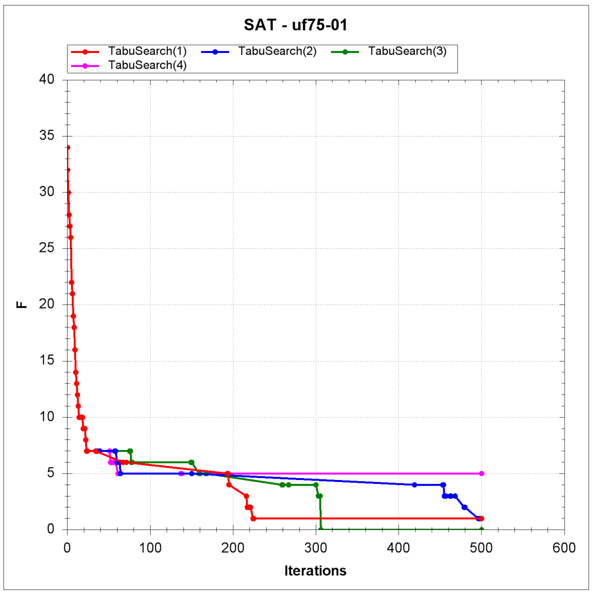


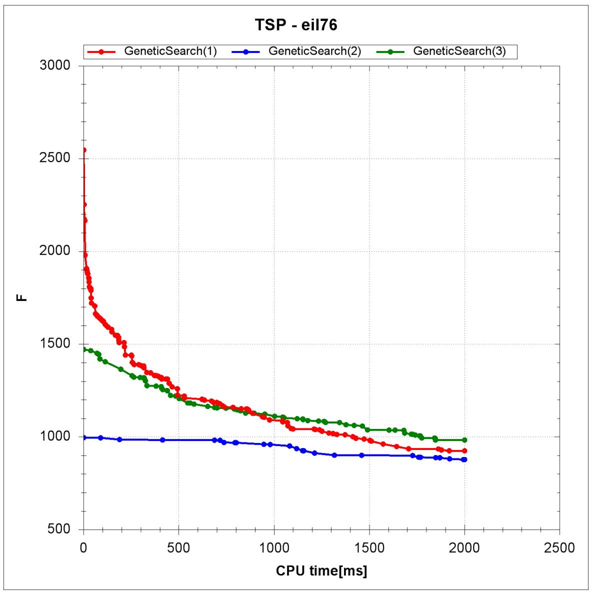
Figure 6.2: Parameter settings: results for Tabu Search

list length. The best performance was achieved using the third configuration (C3). Graph in Figure 6.2 shows different convergence with various settings of tabu search length.

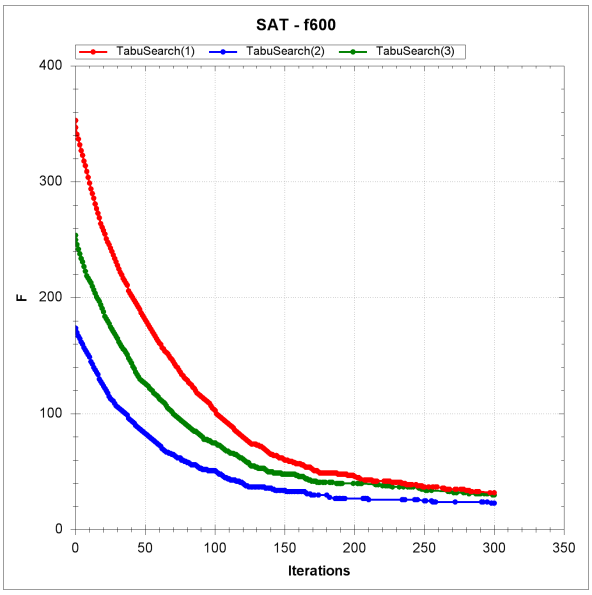
## 6.2 Iteration strength

Now we have set all algorithm parameters and we may ask how the algorithms behave with different initial solutions. In this context we speak about an *Iteration strength*. It is the capability of algorithm to achieve near-optimal solution from initial solutions of different quality. We have performed measures on both our algorithms - Genetic Algorithm and Tabu search - for all three problems. It is obvious from Figure 6.3 that both algorithms are capable to avoid the traps.

See A for another results.



Genetic Algorithm



Tabu Search

Figure 6.3: Iteration strength: SAT

## 6.3 Statistical evaluation

Results of the statistical evaluation provide complex information about quality of optimized solutions. This chapter introduces a test which will be used later to determine whether an algorithm alternation gives statistically better results. We ran Genetic Search algorithm on 30 random initial solutions with C1 parameter configuration (see Table 6.1). See Table 6.3 for statistics - acronyms in table headers stand for: lb - lower bound, best - the best result achieved by the algorithm and C1 parameters, avg - average value of the best solution, var - variance, worst - the worst result achieved, avg. time - an average time of algorithm's run.

From the tables we can see that for more complex input data the result solutions are farther from optimal solution and average time of one iteration increases. For Tabu Search results see

| problem | | lb | best | avg | var | worst | avg. time |
|---|---|---|---|---|---|---|---|
| uf100-01 | sat | 0 | 2 | 3.77 | 1.58 | 6 | 29.8 |
| uf150-01 | sat | 0 | 2 | 6.70 | 3.48 | 9 | 44.6 |
| uf250-01 | sat | 0 | 9 | 17.57 | 14.25 | 26 | 93.0 |
| f600 | sat | 0 | 97 | 108 | 31.94 | 119 | 181.3 |
| berlin52 | tsp | 7542 | 12178 | 13093 | 245894 | 13951 | 6.4 |
| eil76 | tsp | 538 | 1182 | 1280 | 2085 | 1374 | 9.1 |
| kroA100 | tsp | 21282 | 77974 | 84136 | 9677015 | 90690 | 12.4 |
| ft10 | jssp | 930 | 961 | 1038 | 729 | 1114 | 16.6 |
| la22 | jssp | 927 | 1040 | 1096 | 848 | 1154 | 22.8 |
| yn01 | jssp | 888 | 1177 | 1235 | 813 | 1297 | 56.8 |

Table 6.3: Statistical evaluation of Genetic Algorithm

Table 6.4. Similar summary about data complexity and average time can be done also in this case.

| problem | | lb | best | avg | var | worst | avg. time |
|---|---|---|---|---|---|---|---|
| uf100-01 | sat | 0 | 1 | 4.26 | 3.86 | 9 | 10.13 |
| uf150-01 | sat | 0 | 1 | 4.43 | 2.91 | 8 | 15.5 |
| uf250-01 | sat | 0 | 4 | 7.93 | 5.19 | 13 | 22.9 |
| f600 | sat | 0 | 18 | 26.13 | 16.12 | 35 | 52.7 |
| berlin52 | tsp | 7542 | 10767 | 12122 | 400066 | 13372 | 2.6 |
| eil76 | tsp | 538 | 805 | 882 | 1915 | 1018 | 5.63 |
| kroA100 | tsp | 21282 | 44957 | 50587 | 7944655 | 55390 | 10.3 |
| ft10 | jssp | 930 | 936 | 967 | 123 | 991 | 3.7 |
| la22 | jssp | 927 | 935 | 950 | 251 | 1024 | 5.4 |
| yn01 | jssp | 888 | 912 | 923 | 37 | 937 | 22.8 |

Table 6.4: Statistical evaluation of Tabu Search

### 6.3.1 Comparison of GA and TS

For the comparison of both algorithms we chose population of 30 initial solution and we ran the algorithms with 30 second timeout. We performed *paired student t-test* on results to get a decision of significant difference of both population means.

Table 6.5 shows comparison for selected problems.

| problem | GA mean | TS mean | time[s] | *t-test* result | |
|---|---|---|---|---|---|
| uf50-01 | 1.43 | 0.86 | 30 | $\nearrow$ | TS mean is lower |
| uf75-01 | 1.23 | 2.03 | 30 | $\searrow$ | GA mean is lower |
| uf100-01 | 4.43 | 4.33 | 30 | $\approx$ | GA and TS means are the same |
| uf250-01 | 30.03 | 6.57 | 30 | $\nearrow$ | TS mean is lower |
| berlin52 | 11106 | 12001 | 30 | $\searrow$ | GA mean is lower |
| eil76 | 1120 | 873 | 30 | $\nearrow$ | TS mean is lower |
| kroA100 | 78103 | 873 | 30 | $\nearrow$ | TS mean is lower |

Table 6.5: Statistical comparison of both algorithm

There are statistical significant difference between the means in almost all cases. We can say that on small problem instances Genetic Algorithm has a chance compete with Tabu Search in performance - the results are approximately the same. For more complex instances Tabu Search gives significantly better results.

# 7 Algorithm alternation

We are coming at the *algorithm alternation* which is the main theme of this work and the key feature of our framework. The alternation is performed by manipulating with the algorithms via their adapters. The algorithm adapters are easily selected, set and run. Thus, there is a vast space for many experiments with this feature. We have performed some experiments and compared results with standalone algorithm results.

## 7.1 Algorithm reconfiguration

By the term *reconfiguration* we mean a process when the algorithm is run for a certain number of iterations, then the parameters are changed and it is run again with previous solution population. Simply, the *reconfiguration* lies in an alternation the algorithm with itself.

The purpose of this experiment was to take if runtime algorithm reconfiguration has an influence on output solution quality. We have tested the results with solution obtained by the algorithm without reconfiguration whether they differ significantly.

### 7.1.1 Genetic Algorithm reconfiguration

We started from setting configuration C1 defined in section 6.1.1 and we changed the parameters randomly but at most by the given number of percent. The algorithm was run 30 times on selected problem instances with 30 second timeout.

The Table 7.1 shows the test results we have performed. The values in the header determine the ratio of parameter changes.

| problem | | 30% | 40% | 50% | 60% | 70% |
|---|---|---|---|---|---|---|
| uf250-01 | sat | ↗ | ↗ | ↗ | ↗ | ↗ |
| f600 | sat | – | ↗ | ↗ | ↗ | ↗ |
| eil76 | tsp | ↗ | ↗ | ↗ | ↗ | – |
| kroA100 | tsp | ↗ | ↗ | ↗ | ↗ | ↗ |
| ft10 | jssp | – | ↗ | ↗ | ↗ | – |
| la22 | jssp | ↗ | ↗ | ↗ | ↗ | ↗ |
| yn1 | jssp | – | ↗ | ↗ | ↗ | ↗ |

Table 7.1: Genetic Algorithm reconfiguration *t-test*

We can observe that reconfiguration improves the solution for any ratio (see Table 7.1). It is an interesting appearance. It could be explained by the fact that the parameters were randomly set to more feasible values during the search process.

The Table 7.1.1 shows the best reconfiguration result for given problems. The best results has been achieved for ratio approximately about 50%.

The Figure 7.1 shows reconfiguration results for two problem instances. The reconfiguration contribution is obvious from that graphs. It is interesting that the series have just one minimum.

### 7.1.2 Tabu Search reconfiguration

We have performed the same measurements as above for Tabu Search. The *t-test* results are shown in Table 7.3.

Genetic Algorithm reconfiguration contribution is indisputable. The same can't be said about Tabu Search reconfiguration. For some cases we have reached the improvements, for other not (see Table 7.3 and Figure 7.3). In some cases we have even got worse results than with the

| problem | | $F_0$ | $F_{min}$ | $min[\%]$ | $t$ | *t-test* result |
|---|---|---|---|---|---|---|
| uf250-01 | sat | 29.17 | 10.23 | 60 | 23.52 | ↗ |
| f600 | sat | 168.9 | 98.6 | 60 | 40.57 | ↗ |
| eil76 | tsp | 1032 | 944 | 30 | 6.72 | ↗ |
| kroA100 | tsp | 70682 | 58087 | 50 | 13.52 | ↗ |
| ft10 | jssp | 1022 | 1004 | 50 | 3.07 | ↗ |
| la22 | jssp | 1079 | 1026 | 40 | 6.25 | ↗ |
| yn1 | jssp | 1316 | 1205 | 60 | 13.46 | ↗ |

Table 7.2: Genetic Algorithm reconfiguration - $F_0$: Objective function mean without reconfiguration, $F_{min}$: Objective function mean with reconfiguration by $min$ percent, $min[\%]$: percent value the algorithm gives best results with, $t$ the value of t-test

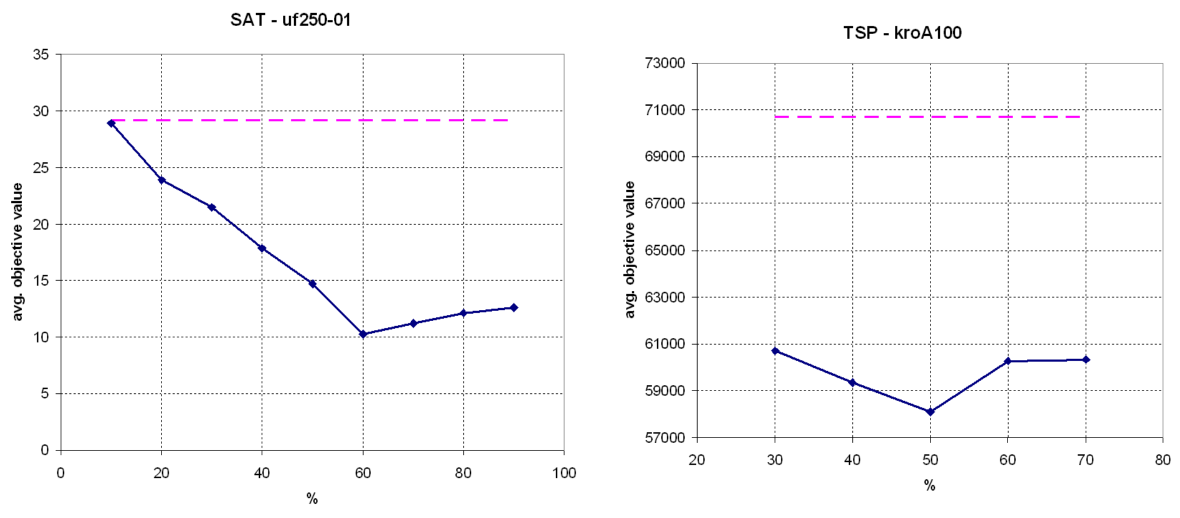
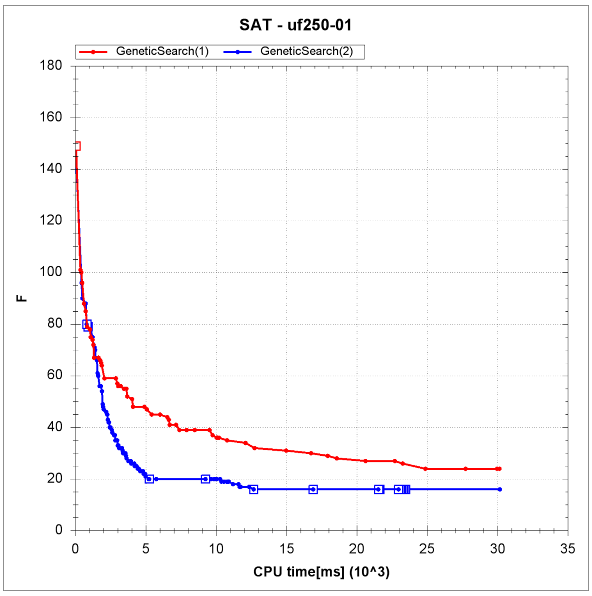


Figure 7.1: Genetic Algorithm reconfiguration improvement - The ratio of parameter changes on the x-axis, F - the objective value mean on the y-axis. A dash line is presented for comparison with no-reconfiguration approach.

simple algorithm. We explain this behavior by that for Tabu Search is more suitable find all the parameter settings and don't change it during the search process.

A results downgrade is well visible in Figure 7.3. A Tabu Search reconfiguration for certain instances with worse convergence are shown in Figure 7.4.



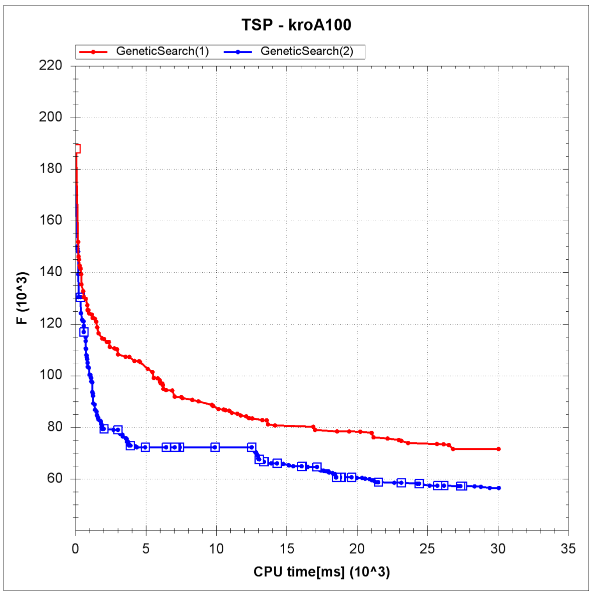


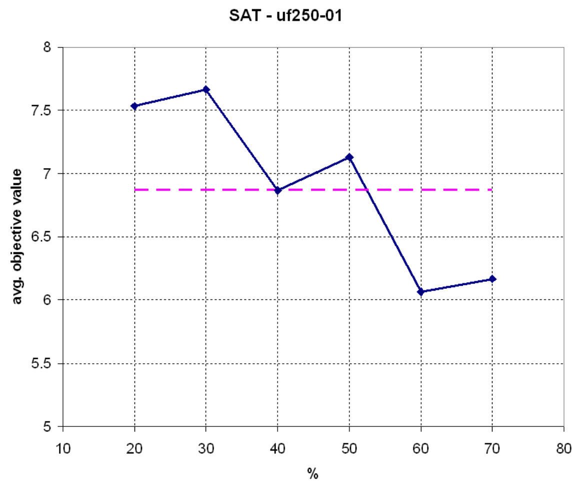
Figure 7.2: Genetic Algorithm reconfiguration - A comparison of Genetic Algorithm without a reconfiguration (Genetic algorithm(1) series) and Genetic Algorithm with a reconfiguration (Genetic Algorithm(2) series). The small squares in the charts are the alternation phases - the points where the parameters are changing.

| problem | | 30% | 40% | 50% | 60% | 70% |
|---|---|---|---|---|---|---|
| uf250-01 | sat | ↘ | ≈ | ≈ | ↗ | ≈ |
| f600 | sat | ≈ | ↘ | ≈ | ↘ | ≈ |
| eil76 | tsp | ≈ | ≈ | ≈ | ≈ | ≈ |
| kroA100 | tsp | ↗ | ↗ | ↗ | ↗ | ↗ |
| ft10 | jssp | ≈ | ≈ | ≈ | ≈ | ≈ |
| la22 | jssp | ≈ | ≈ | ↗ | ↗ | - |
| yn1 | jssp | ↘ | ↘ | ↘ | ↘ | ↘ |

Table 7.3: Tabu Search reconfiguration *t-test*

| problem | | $F_0$ | $F_{min}$ | min [%] | t | *t-test* result |
|---|---|---|---|---|---|---|
| uf250-01 | sat | 6.86 | 6.07 | 60 | 3.38 | ↗ |
| f600 | sat | 29.47 | 29.01 | 50 | 1.06 | ≈ |
| eil76 | tsp | 637 | 625 | 20 | 2.34 | ↗ |
| kroA100 | tsp | 38063 | 34184 | 60 | 8.96 | ↗ |
| ft10 | jssp | 953 | 949 | 40 | 1.92 | ≈ |
| la22 | jssp | 948 | 940 | 50 | 2.51 | ↗ |
| yn1 | jssp | 920 | 936 | 60 | -6.11 | ↘ |

Table 7.4: Tabu Search reconfiguration statistics - $F_0$: Objective function mean without reconfiguration, $F_{min}$: Objective function mean with reconfiguration by $min$ percent, $min$[%]: percent value the algorithm gives best results with, $t$ the value of t-test



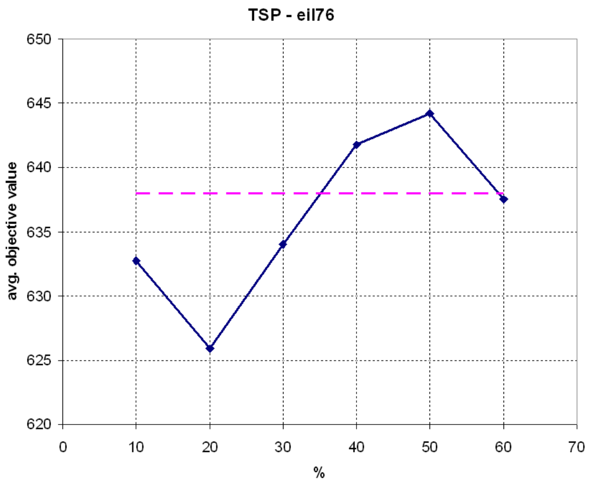
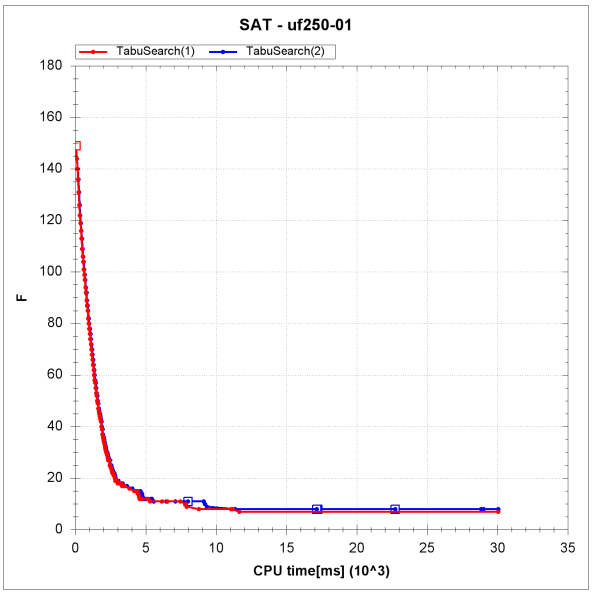


Figure 7.3: Tabu Search reconfiguration - The ratio of parameter changes on the x-axis, F - the objective value mean on the y-axis. A dash line is presented for comparison with no-reconfiguration approach.



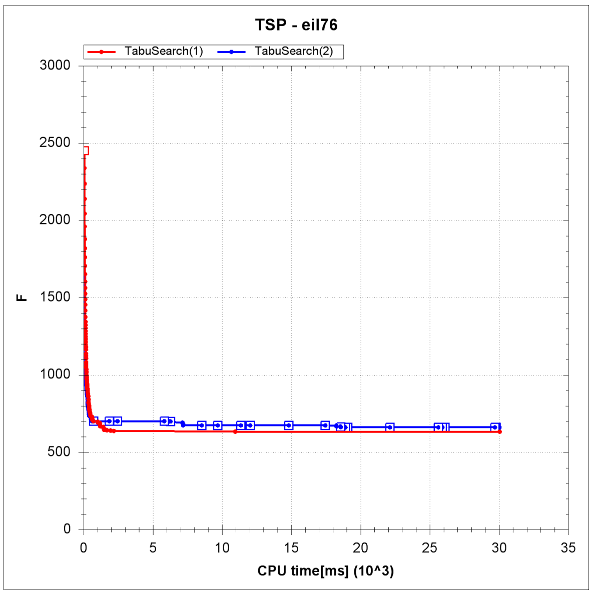


Figure 7.4: Tabu Search reconfiguration convergence - a comparison of Tabu Search without a reconfiguration (Tabu search(1) series) and Tabu Search with a reconfiguration (Tabu Search(2) series). The small squares in the charts are the alternation phases - the points where the parameters are changing.

## 7.2 Regular alternation

By the term a *regular alternation* we mean a process when the algorithms are regularly changed for each other after a number of iterations. The first algorithm starts, stops - then the second algorithm starts, stops - the first one starts ... - and so on until a timeout is reached.

We have performed some experiments with the regular alternation.

### 7.2.1 One alternation phase

The search process is composed of exactly two dissimilar search sub-processes. The order of the sub-processes is significant.

#### 7.2.1.1 Genetic Algorithm, Tabu Search order

The first algorithm we run was Genetic Algorithm. Then Tabu Search was run upon the GA results. Because of the fact Tabu Search achieves the better performance than Genetic Algorithm (in our case) we compare the result gathered by alternation only with Tabu search results. Table 7.5 shows Tabu Search results for certain problem instances with 30s timeout. We will compare alternation results with these ones.

| problem | TS mean |
|---|---|
| uf100-01 | 3.36 |
| uf250-01 | 6.50 |
| berlin52 | 8839 |
| eil101 | 764 |

Table 7.5: Tabu Search results with 30s timeout

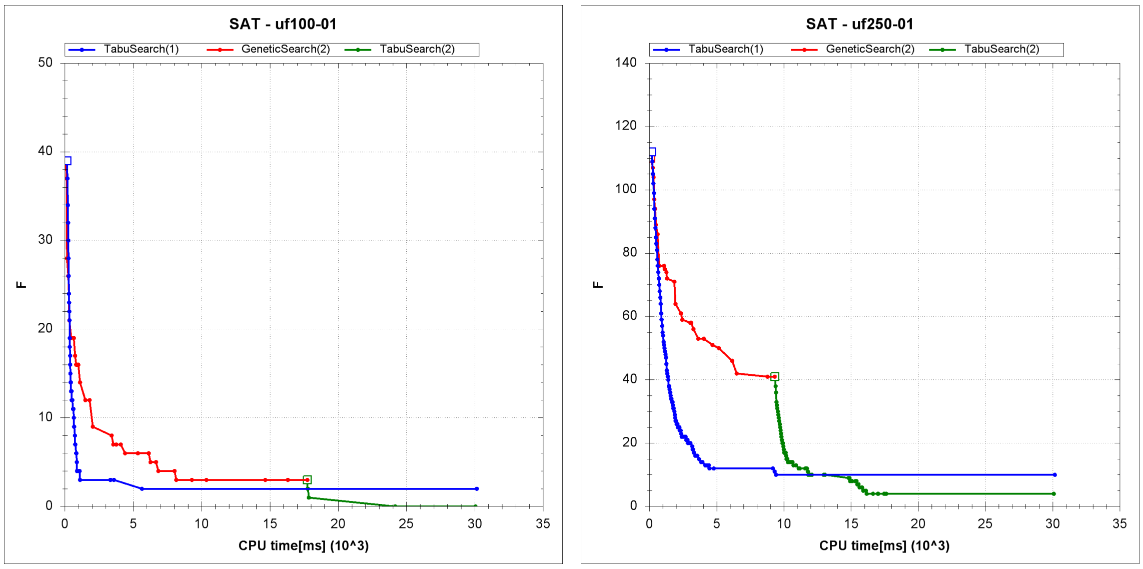


Figure 7.5: GA/TS Regular alternation convergence - in both cases Tabu Search convergence is faster than convergence of Genetic Algorithm but after short time TS does not find a better solutions anymore (premature convergence occurred). On the other hand Tabu Search in second phase of alternation has a better starting position and achieves better results

We have set 30 second timeout. This time is divided between the algorithm processes. When we have a look on Table 7.6 we can see several improvements and downgrades as well. It seems

| problem | GA/TS | TS n | GA/TS mean | t result |
|---|---|---|---|---|
| uf100-01 | 1/1 | 1 | 2.70 | ≈ |
| | 1/1 | 5 | 2.66 | ≈ |
| | 1/1 | 10 | 2.56 | ↗ |
| | 1/1 | 15 | 2.23 | ↗ |
| | 1/1 | 25 | 2.46 | ↗ |
| | 2/1 | 15 | 2.73 | ≈ |
| | 1/2 | 15 | 2.33 | ≈ |
| uf250-01 | 1/1 | 1 | 6.0 | ≈ |
| | 1/1 | 5 | 8.33 | ↘ |
| | 1/1 | 10 | 8.67 | ↘ |
| | 1/2 | 1 | 5.33 | ↗ |
| | 1/2 | 5 | 5.66 | ↗ |
| | 1/2 | 10 | 7.33 | ↘ |
| | 2/1 | 1 | 6.83 | ≈ |
| | 2/1 | 5 | 8.67 | ↘ |
| | 2/1 | 10 | 8.67 | ↘ |
| berlin52 | 1/1 | 1 | 8557 | ↗ |
| | 1/1 | 3 | 8785 | ≈ |
| eil101 | 1/1 | 1 | 784 | ≈ |
| | 1/1 | 3 | 796 | ↘ |
| | 1/1 | 5 | 797 | ↘ |

Table 7.6: GA/TS regular alternation results - GA/TS in the header means the ratio between the Genetic search running time and the Tabu Search running time; TS n is the number of best solutions from the population of 100 subjects Tabu Search is searching through

that a better chance to find a better solution is when Tabu Search algorithm searches through more solutions. However, the more solutions Tabu Search algorithm searches through the less iterations it performs on one solution because of timeout. Since the number of iterations also has a significant influence on solution quality a balance between the number of solutions and the number of iterations has to be found.

#### 7.2.1.2 Tabu Search, Genetic Algorithm

The first algorithm we run was Tabu Search. Then Genetic Algorithm was run upon the TS results. The results are shown in Table 7.7.

### 7.2.2 Multiple alternation

The search process is composed of many different search sub-processes.

#### 7.2.2.1 Constant parameters

The algorithm parameters are set at the startup and they are not changed during the alternation phases. Table 7.8 shows that no improvements were achieved. Tabu search, which is the main factor in search process, starts with initial solutions and it has a lot of work with them. Genetic Algorithm hardly achieves better results because it is under its quality threshold.

In Figure 7.7(a),the alternation can be seen from two sights: (1) we improved Genetic Algorithm significantly, (2) we worsened the Tabu Search. It is obvious that the importance of

| problem | TS/GA | TS n | TS/GA mean | t result |
|---|---|---|---|---|
| uf100-01 | 1/1 | 1 | 3.17 | $\approx$ |
| | 1/1 | 5 | 1.50 | $\approx$ |
| | 1/1 | 10 | 1.23 | $\nearrow$ |
| | 1/1 | 15 | 1.33 | $\nearrow$ |
| | 1/1 | 25 | 2.50 | $\nearrow$ |
| | 1/1 | 50 | 4.60 | $\searrow$ |
| | 2/1 | 10 | 1.83 | $\nearrow$ |
| | 1/2 | 10 | 1.90 | $\nearrow$ |
| | 1/2 | 20 | 2.33 | $\nearrow$ |
| | 1/2 | 30 | 4.67 | $\searrow$ |
| uf250-01 | 1/1 | 1 | 8.3 | $\searrow$ |
| | 1/1 | 5 | 11.7 | $\searrow$ |
| | 1/1 | 10 | 25.8 | $\searrow$ |
| | 1/2 | 1 | 9.10 | $\searrow$ |
| | 2/1 | 1 | 8.90 | $\searrow$ |
| | 2/1 | 5 | 20.30 | $\searrow$ |

Table 7.7: TS/GA regular alternation results - TS/GA in the header means the ratio between the Tabu Search running time and the Genetic search running time; TS n is the number of best solutions from the population of 100 subjects Tabu Search is searching through

| problem | GA rate [%] | TS rate [%] | TS n | GS/TS mean | t result |
|---|---|---|---|---|---|
| berlin52 | 5 | 5 | 1 | 9063 | $\approx$ |
| | 5 | 10 | 1 | 8599 | $\approx$ |
| | 10 | 10 | 1 | 8906 | $\approx$ |
| | 10 | 10 | 3 | 8875 | $\approx$ |
| | 10 | 20 | 3 | 8724 | $\approx$ |
| | 10 | 10 | 5 | 9186 | $\searrow$ |

Table 7.8: Multiple alternation

the Genetic Algorithm improvement is lower than the worsening of Tabu Search. It is more reasonable to use Tabu Search directly.

#### 7.2.2.2 Randomized parameters

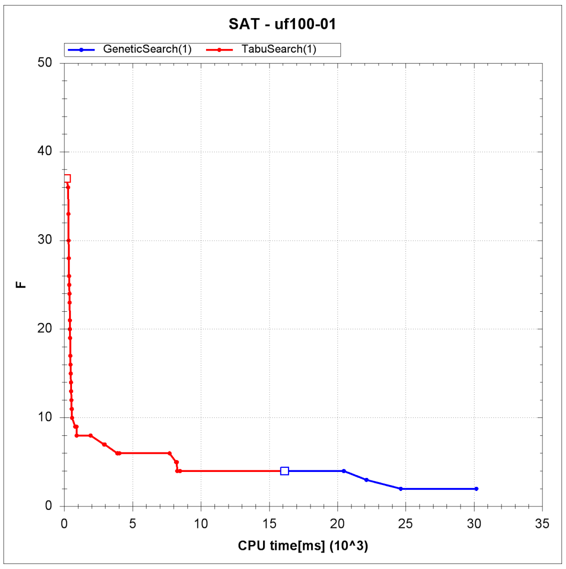
The algorithm parameters are randomly changed during the alternation phase. Table 7.9 shows results for one instance.

| problem | GS/TS mean | t result |
|---|---|---|
| berlin52 | 8968 | $\approx$ |
| problem | TS/GS mean | t result |
| berlin52 | 8980 | $\approx$ |

Table 7.9: Alternation with randomized parameters

## 7.3 Random alternation

The idea of this alternation is fully random algorithm selection and parameter setting. This experiment was done just for the completeness and the results does not differ from multiple



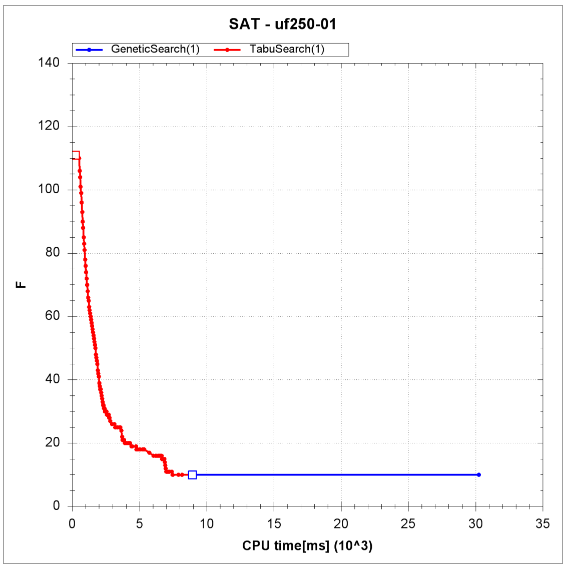
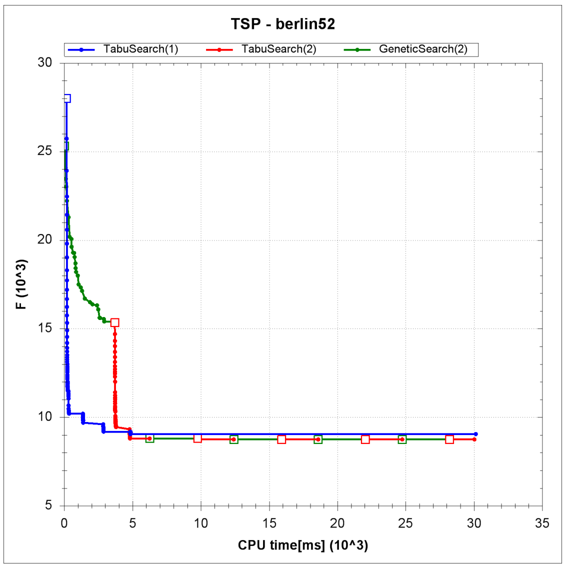
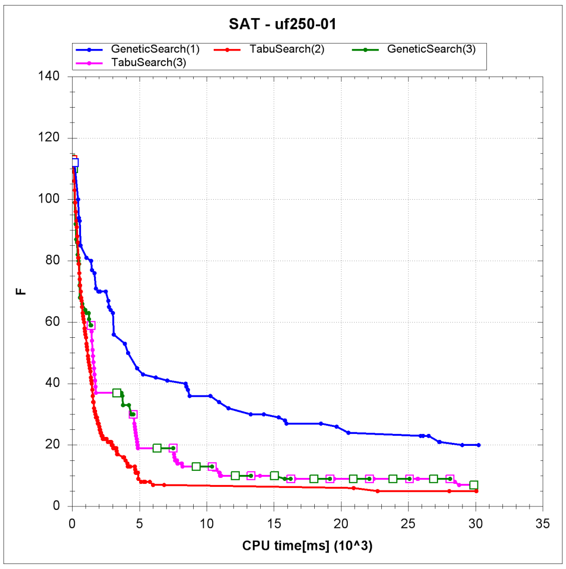


Figure 7.6: TS/GA Regular alternation convergence - for smaller instances, Genetic Algorithm can find new solution after Tabu Search phase.



(a)



(b)

Figure 7.7: GS/TS multiple alternation with constant parameters

alternation results markedly. See A.2.

## 7.4 Alternation observations

- Initial Tabu Search convergence is unbeatable.
- Genetic Algorithm should not hinder the Tabu Search.
- It is better to search through less solutions and more iterations with Tabu Search.

## 7.5 Alternation with a strategy

Previous experiments either were regular (the order of the algorithm's runs was given) or random (the algorithms were selected randomly). We have observed that some searching phases were very useful some others not. To employ the best the algorithms provide we decided to design a strategy the search process (performing the alternation) is guided by.

The strategy is a set of decision rules, according them the particular algorithm is chosen and set. The information about previous runs are collected from algorithm reports and evaluated with decision rules.

In the topic of this thesis stands: runtime analysis and parameter adaptation. We analyze the search reports and the parameters are changed (adapted) according the decision rules.

### 7.5.1 Algorithm reports

In section 5.1.9 we have mentioned a SearchReport class. It serves for the algorithm reporting. During the algorithm runtime each algorithm sends particular information to its adapter (a wrapper class). The adapter is listening to the algorithm and stores the information it gets. When the algorithm is finished the adapter class creates the report and publishes it for next processing.

The report may differ with a kind of the algorithm. It generally contains following items:

- ID - decision rule id
- The number of iterations
- Number of new (better) solutions
- The number of iterations since the last change of the objective value
- Minimal (the best) objective value
- Objective value mean of whole population

### 7.5.2 Our strategy

1. We started with a global population of random solutions.
2. From previous observations we have found out that the initial convergence of Tabu Search algorithm is unbeatable. A solution quality is improving very fast during a first number of iterations. From this reason the first step is choosing one solution from the global population and running the Tabu Search as long as the solution is improving.
3. Now the Genetic Algorithm should have its turn. It is naive to think that a Genetic Algorithm with a population of random solution and with one best quality solution has a chance to improve the best solution.
4. From this reason we apply thin (a few iterations) Tabu Search on the whole population until the objective value mean decreases under some threshold. Then we can apply the Genetic Algorithm. This phase should be in accordance with a keeping of the population diversity.
5. A deeper Tabu Search phase follows - it means that we run the Tabu Search algorithm to search through a few best solutions from the global population. In time this phase is not improving Genetic Algorithm set on the divergence is run to discover other regions of solution space.

6. All the previous steps can be repeated now.

In order for the decision rules to be easier, we have prepared sets of algorithm parameters with certain purpose. For instance "Tabu Search intensification parameter set", "Genetic Algorithm diversification set", "tiny Tabu Search parameter set" and so on (see Table 7.10).

| paramameter set name | purpose |
|---|---|
| GS-intensification | intensification |
| GS-diversification | diversification - it should discover new regions of solution space |
| TS-init-intensification | intensification of one initial solution |
| TS-tiny | population mean improvement |
| TS-intensification | intensification of several best solutions (deep search) |

Table 7.10: Parameter sets the strategy works with

Pseudo-code of our strategy is shown in Figure C.1.

### 7.5.3 Strategy results

We run the SearchCoordinator with the alternation strategy on complex problems (30 times again) to determine the effectivity of our strategy.

| problem | avg | time[s] | *t-test* result | improvement[%] |
|---|---|---|---|---|
| berlin52 | 8279 | 300 | $\nearrow$ | 4.81 |
| eil101 | 723 | 300 | $\nearrow$ | 2.77 |
| yn1 | 947 | 300 | $\approx$ | – |
| f600 | 22.6 | 300 | $\approx$ | – |

Table 7.11: Strategy results

We can observe that the first strategy phase guarantees that we don't get worse results. The other phases can be just to the benefit.

Although, we have got a better results for particular instances, globally, we can not say that our strategy brings the break through in results quality. The improvement has been achieved on smaller instances. The complex instances still resist. Here is space for another experiments and future work. We assume that better result would achieve in case of the better performance of Genetic Algorithm.

## 7.6 Experiment summary

In all these experiments we were trying to know the algorithm behavior on given problems at first. We have gathered that Tabu Search algorithm gives better results than Genetic algorithm in most cases. The solutions founded by Tabu Search are of higher quality and in a shorter time. This fact can be simply explained by the simplicity of our Genetic algorithm implementation.

Before we were interested in a mutual algorithm alternation we have performed the experiment we called *reconfiguration* (a process of running the algorithm with changed parameters repeatedly). The *reconfiguration* experiments have shown that parameter changes (approximately about 50 per cent) increase the Genetic Algorithm performance significantly. On the other hand the *reconfiguration* of Tabu Search gave embarrassed results.

The mutual algorithm alternation experiments we have performed have only shown almost combinatorial experiment potentialities. We selected a few of them with interesting results. The best results we got with one alternation phase when Genetic algorithm is followed by Tabu Search. However, in these cases we had to set a balance between the number of Tabu Search iterations and the number of solutions Tabu Search were searching through. Nevertheless, we got significantly better result than we could get by the algorithms alone.

After the experiments with the algorithm alternation we evaluated all the experiment output data and pursuant of all observation we have designed an alternation strategy. This strategy is described in section 7.5.2 and it tries to take all advantages the algorithms have.

Frankly, we were expected to get better results with the algorithm alternation. The results we have got does not provide sufficient improvement. We guess it is due to algorithm strength differences - Tabu Search is much more stronger than Genetic algorithm. We may suppose the results will be better if we use equivalent Tabu Search partner(s) in strength.

Naturally, we aware the fact that complexity of problem instances is various and one universal strategy could not be suitable. This can by solved by using a knowledge base which would hold various strategies. But it is rather a suggestion for future work.

The strategy we have proposed is not in final state indeed. It is not the best one we could have invented and it should mainly demonstrate the way the future work may proceed.

# 8 Conclusions

In this work we have proposed a new approach for solving combinatorial optimization problems based on the alternation of meta-heuristic algorithms. In accordance with our proposal we have designed and implemented a framework realizing the alternation.

Upon the framework we have implemented two meta-heuristics algorithm (Genetic Algorithm and Tabu Search) and three problems (Satisfiability Problem (SAT), Traveling Salesman Problem (TSP) and Job Shop Scheduling Problem (JSSP)).

We have confirmed the functionality of the framework and the validity of implemented problems as well.

We have shown framework capabilities of providing a lot of kind of experiments and we have performed a lot of them as well. We were systematically trying to prove or disprove the feasibility of the alternation during the optimalization process.

It has been turned out that proposed approach improve the convergence and the quality of founded solutions in many cases. The experiments has shown that Tabu Search algorithm gives much more better results than Genetic Algorithm. It caused that it is difficult to achieve significantly better results by the algorithm alternation.

The main advantage of our framework is good scalability for other algorithms and problems as well.

We place great hopes in another algorithms - e.g. Ant Colony Optimization - that we are planning to add to improve performance of our approach. We guess it is the way for future work.

## 8.1 Suggestions for future work

Although, we have performed a lot of experiments and implemented many features, a lot of questions still remain unanswered. The questions may look like:

What if we

- invented more sophisticated alternation strategy.
- improved Genetic Algorithm (e.g. niching, deterministic crowding).
- added other (more feasible) algorithms into SearchCoordinator.
- did something else.

These are the topics for the future work. All indicia and the number of special articles about hybrid algorithms show that our way of solution space exploration need not be wrong.

# 9 Bibliography

[1] Evolutionary potential timetables optimization by means of genetic and greedy algorithms. page 24, 1999.

[2] R. Tadei A. Moraglio, H.M.M. Ten Eikelder. Genetic local search for job shop scheduling problem. 2004.

[3] M. Dorigo, G. Di Caro, and L. Gambardella. Ant algorithms for discrete optimization, 1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Navrh programu pomoci vzoru.* GRADA Publishing, Praha, CZ, 2003.

[5] Fred Glover and Manuel Laguna. Tabu search. pages 70–150, 1993.

[6] Robert Harder. Opents. Website, 2007. `http://www.coin-or.org/Ots`.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[8] Amit Konar. *Artificial intelligence and soft computing: behavioral and cognitive modeling of the human brain.* CRC Press, Inc., Boca Raton, FL, USA, 2000.

[9] P. Larranaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artif. Intell. Rev.*, 13(2):129–170, 1999.

[10] Fábio M. Lopes and Aurora T. R. Pozo. Genetic algorithm restricted by tabu lists in data mining. *sccc*, 0:8, 2001.

[11] Ilias Maglogiannis, Kostas Karpouzis, and Max Bramer. *Artificial Intelligence Applications and Innovations: 3rd IFIP Conference on Artificial Intelligence Applications and Innovations (AIAI), 2006, June 7-9, ... Federation for Information Processing).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[12] Samir W. Mahfoud. *Niching methods for genetic algorithms.* PhD thesis, Urbana, IL, USA, 1995.

[13] Vladimir Marik, Jiri Lazansky, and Olga Stepankova. *Umela inteligence 1.* Academia, Prague, CZ, Boca Raton, FL, USA, 1993.

[14] Takashi Matsumura, Morikazu Nakamura, and Shiro Tamaki. A parallel tabu search and its hybridization with genetic algorithms. page 18, 2000.

[15] Eugeniusz Nowicki and Czeslaw Smutnicki. A fast taboo search algorithm for the job shop problem. *Manage. Sci.*, 42(6):797–813, 1996.

[16] Eugeniusz Nowicki and Czeslaw Smutnicki. Some new ideas in ts for job shop scheduling. *Metaheuristic Optimization via Memory and Evolution*, 30(2):165–190, 2005.

[17] B. Ombuki, M. Nakamura, and O. Maeda. A hybrid search based on genetic algorithms and tabu search for vehicle routing. 2002.

[18] Pavel Osmera. Geneticke algoritmy a jejich aplikace. 0:108, 2001.

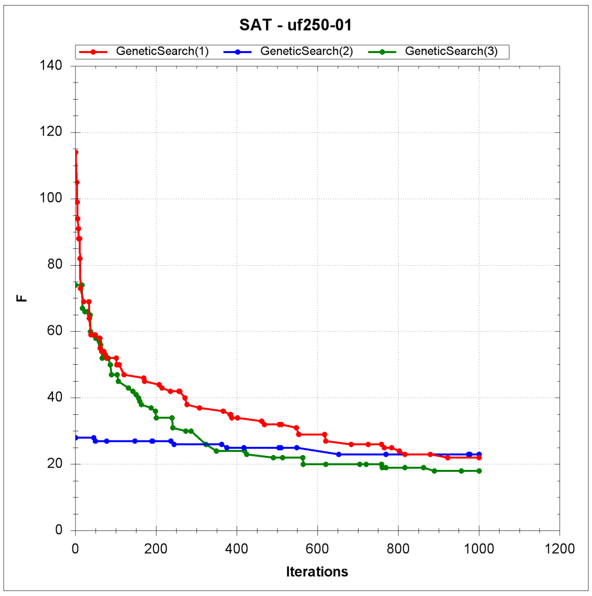
[19] Jaroslav Pozivil and Martin Zdansky. Combination genetic/tabu search algorithm for hybrid flowshops optimization. 2002.

[20] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

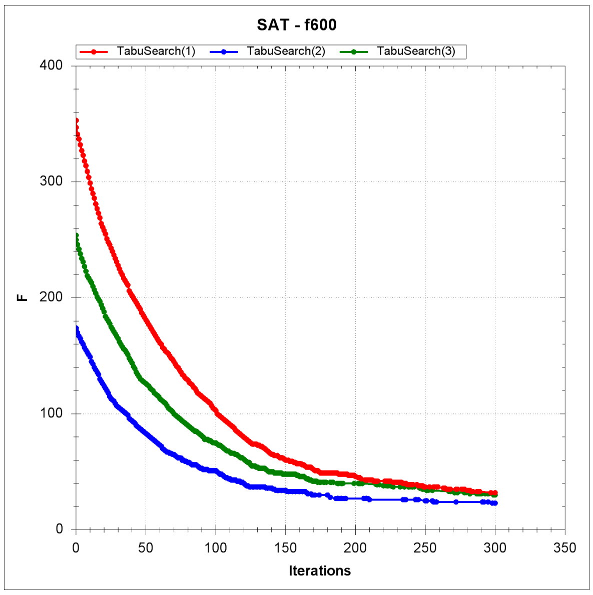
[21] T. Yamada and R. Nakano. Genetic algorithms for job-shop scheduling problems. 1997.

# A Another experiment output

## A.1 Iteration strength

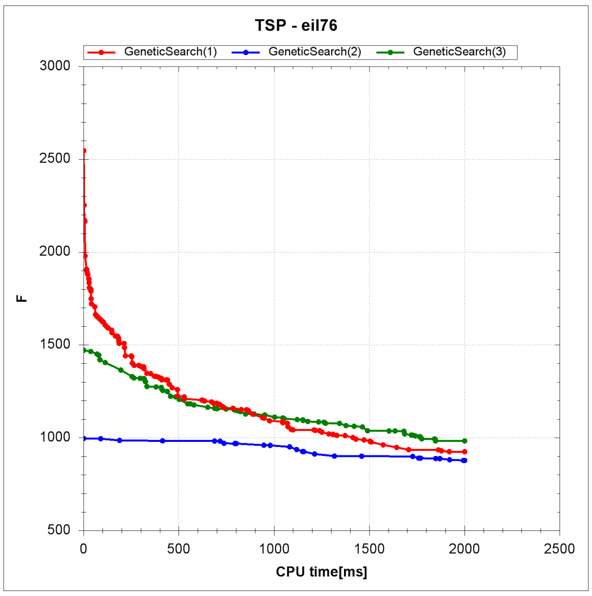


Genetic Algorithm

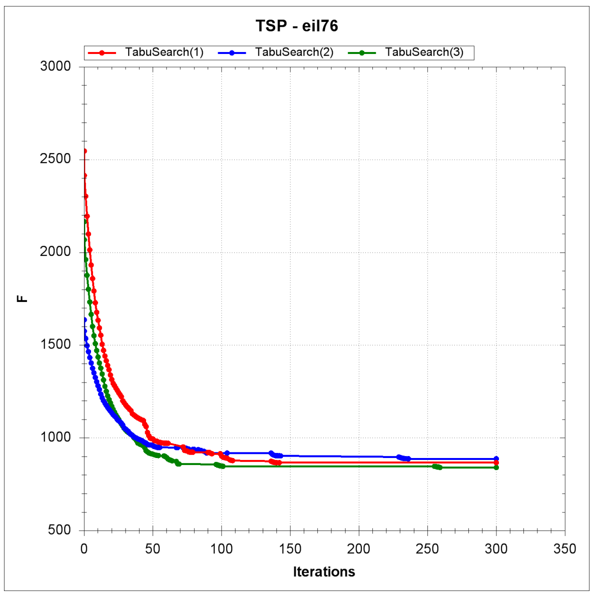


Tabu Search

Figure A.1: Iteration strength: SAT



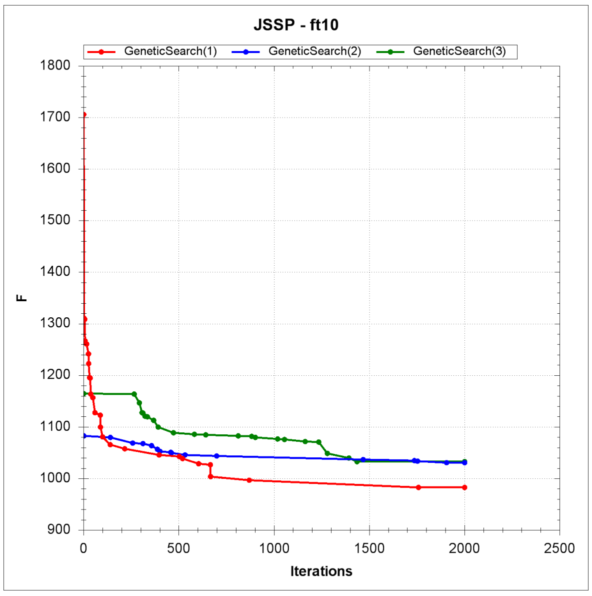
Genetic Algorithm



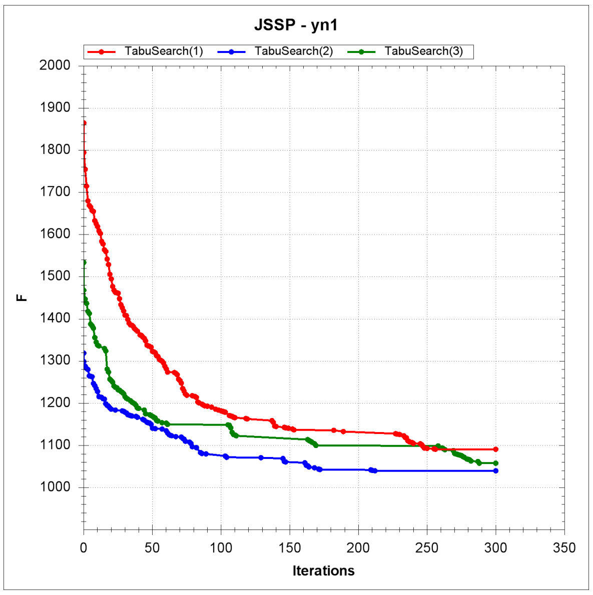
Tabu Search

Figure A.2: Iteration strength: TSP

## A.2 Multiple alternation with random parameters

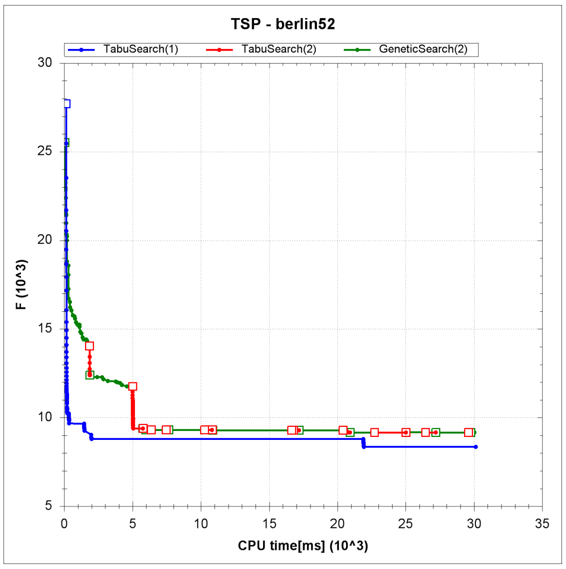


Genetic Algorithm



Tabu Search

Figure A.3: Iteration strength: JSSP



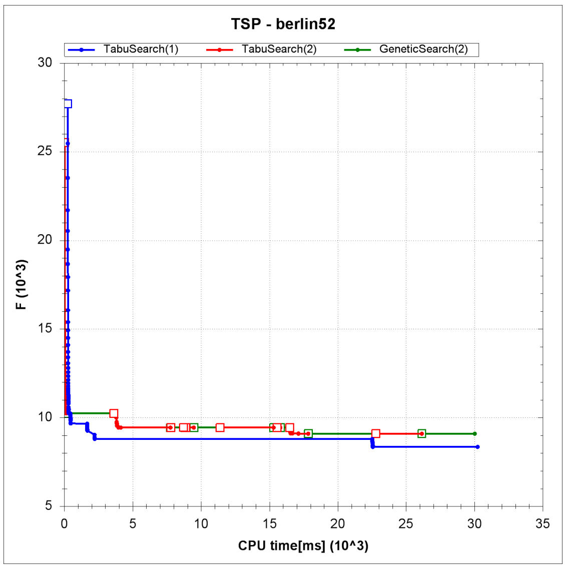


Figure A.4: GS/TS multiple alternation with random parameters

## B Benchmark data

Following table contain information about benchmark data we have used. The instances can be easily found on Internet according the names.

| class | name | variables | clauses | satisfiable |
|---|---|---|---|---|
| Uniform Random-3-SAT | uf20 | 20 | 91 | yes |
| | uf50 | 50 | 218 | yes |
| | uf75 | 75 | 325 | yes |
| | uf100 | 100 | 430 | yes |
| | uf150 | 150 | 645 | yes |
| | uf250 | 250 | 1065 | yes |
| DIMACS LRAN | f600 | 600 | 2550 | yes |
| | f1000 | 1000 | 4250 | yes |
| MAX-3SAT | max-100-500 | 100 | 500 | yes |
| | max-100-1000 | 100 | 1000 | yes |
| | max-100-1500 | 100 | 1500 | yes |

Table B.1: SAT benchmarks

| name | dimension | best |
|---|---|---|
| berlin52 | 52 | 7542 |
| eil51 | 51 | 426 |
| eil76 | 52 | 538 |
| eil101 | 52 | 629 |
| kroA100 | 100 | 21282 |

Table B.2: TSP benchmarks

| name | jobs | operations | operations | best |
|---|---|---|---|---|
| FT10 | 10 | 10 | 10 | 930 |
| FT20 | 20 | 5 | 5 | 1165 |
| LA20 | 10 | 10 | 10 | 902 |
| LA21 | 15 | 10 | 10 | 1046 |
| LA22 | 15 | 10 | 10 | 927 |
| LA36 | 15 | 15 | 15 | 1268 |
| LA37 | 15 | 15 | 15 | 1397 |
| LA38 | 15 | 15 | 15 | 1196 |
| LA40 | 15 | 15 | 15 | 1222 |
| YN01 | 20 | 20 | 20 | 888 |
| YN02 | 20 | 20 | 20 | 909 |

Table B.3: JSSP benchmarks

## C Alternation strategy pseudocode

Note that `lastReport` is a report from the previous algorithm's run and `parameterSets` is a collection of predefined parameter sets.

```
if (lastReport not exists)
    return parameterSets.get("TS-init-intensification"); // Initial parameter set

if (lastReport.MethodName == "TabuSearch")
{
    if (lastReport.ParameterSetName == "TS-init-intensification")
    {
        if (lastReport.NumberOfNewSolutions > 3)
            return parameterSets.get("TS-init-intensification");
        else
            return parameterSets.get("TS-tiny");
    }

    if (lastReport.ParameterSetName == "TS-tiny")
    {
        if (lastReport.AvgObjVal > 1.5 * _minObjVal)    // Threshold: 1.5 * _minObjVal
            return parameterSets.get("TS-tiny");
        else
            return parameterSets.get("GS-intensification");
    }

    if (lastReport.ParameterSetName == "TS-intensification")
    {
        if (lastReport."NumberOfNewSolutions" != 0)
            return parameterSets.get("TS-intensification");
        else
            return parameterSets.get("GS-diversification");
    }
}

if (lastReport.MethodName == "GeneticSearch")
{
    if (lastReport.ParameterSetName == "GS-intensification" and
        lastReport.NumberOfNewSolutions != 0)
        return parameterSets.get("GS-intensification");

    if (lastReport.ParameterSetName == "GS-diversification")
        return parameterSets.get("TS-tiny");

    if (lastReport.NumberOfNewSolutions")
        return parameterSets.get("TS-intensification");
}
```

Figure C.1: Our alternation strategy pseudo-code