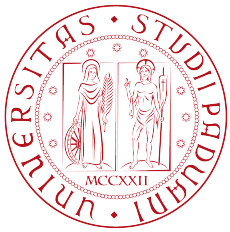


UNIVERSITY OF PADOVA

MASTER OF SCIENCE IN COMPUTER ENGINEERING

FINAL REPORT
OPERATIONS RESEARCH 2

LUCA BORIN 1134473

IACOPO MANDATELLI 1151791

Prof. MATTEO FISCHETTI

07 November 2017
ACADEMIC YEAR 2016-2017

Abstract

The purpose of this report is to illustrate various techniques used to approach the *Traveling Salesman Problem* (TSP) and to compare the results obtained by testing the aforementioned methods with respect to reference instances.
The topics treated in this document are strictly connected to the *Operations Research 2* course program. We desire to provide a complete description of different approaches, but we leave out the implementation details, considering that all the code is well documented and easily available in a dedicated repository.
In the first chapter we provide a formal problem formulation and the description of the software used, as well as the structure and the link of the repository containing the code. The chapters 2-3-4 are dedicated respectively to exact, heuristic and matheuristic algorithms. The chapter 5 contains all the details about the methods used to conduct the tests, in addition to the discussion of the results obtained. The chapter 6 presents a synthesis of the test conclusions. Lastly, in the chapter 7 all the tables with the raw data obtained from the test runs are reported.

Contents

Chapter 1

Introduction

1.1 Problem formulation

The Traveling Salesman Problem (TSP) consists of finding the minimum cost Hamiltonian circuit in a given oriented graph G = (V,A), where V is a finite set of vertexes and A is a family of sorted pairs of vertexes, called arcs. From a computational point of view, it is an NP-hard problem, so the exhaustive research of the best route combination can't be considered a valid approach to aim for the solution.
There are many possible metrics that can be used to assign a weight to each edge, but the most common used is the 2D euclidean distance. Without loss of generality, the graph is considered complete: if a connection should not be considered, it is possible to assign an arbitrary high weight to it, in order to discourage its choice during the optimization process.

A possible PLI model [5] uses the following decision variables:

$$x_{ij} = \begin{cases} 1 & \text{if the arc (i,j)} \in \text{A is chosen in the optimal circuit} \\ 0 & \text{otherwise} \end{cases}$$

The resulting model is the following one:

$$min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1.1.1}$$

$$\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1, \quad j \in V \tag{1.1.2}$$

$$\sum_{(i,j)\in\delta^+(i)} x_{ij} = 1, \quad i \in V \tag{1.1.3}$$

$$\sum_{(i,j)\in\delta^+(S)} x_{ij} \geq 1, \quad S \subset V : 1 \in S \tag{1.1.4}$$

$$x_{ij} \geq 0 \text{ integer}, (i,j) \in A \tag{1.1.5}$$

The bounds provided by (1.1.4) are necessary to forbid the creation of solutions that consider multiple disjoint circuits: every vertex $t \neq 1$ has to be reachable from the vertex 1.

Applications

The TSP has many practical applications beside the initial formulation. We present a brief list, suggesting the W.J. Cook book [2] for a more complete presentation:

- Road trips

- Mapping genomes
- Aiming telescopes, X-rays, and lasers
- Guiding industrial machines
- Soldering printed circuit board
- Organizing data
- Scheduling jobs

1.2 Software Environment

A detailed version of the software used during the project development is presented in Table 1.1.

software	version
Operating System	Ubuntu 14.04.5 LTS
Programming language	C11
MIP solver	IBM ILOG CPLEX 12.7.1 - Student
IDE	CLion 2017.1
TSP support library	Concorde 03.12.19 release
VCS	Git 2.13
Build manager	CMake 3.9.3

Table 1.1: Software used, the version column also works as a link to the reference site.

We have chosen CLion among the many available Integrated Development Environments (IDEs) because it has on-the-fly code analysis, a powerful debugger and a good integration (as shown in the image below) with Git, a Version Control System (VCS) used for tracking changes in files among multiple devices, and CMake, a build manager that simplifies the compilation process of large projects.

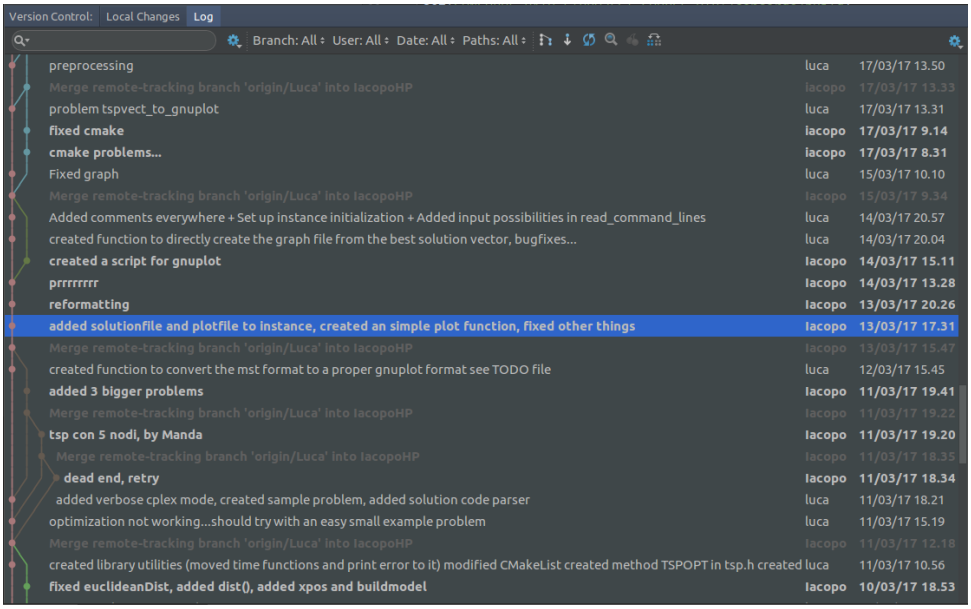


Figure 1.1: Screenshot from the VCS tab of CLion with the timeline of our project.

The code of the project is public and entirely available within the Bitbucket repository at the URL: [SEND AN EMAIL AT borin.luca.93@gmail.com OR mandamondo@gmail.com]. The project repository is structured as follows: in the root folder there are the CMakeList file, which manages the build and linking steps, the and the README.md file, which explains how the repository is structured and how to execute the code, and the file Report.pdf which is a copy of this document. The *src* folder contains the source code, the *include* folder groups the header files and the *data* folder maintains the textual problem instances used to test the various algorithms, ordered by type and dimension. Apart from the straightforward installation of all the above-mentioned software components, the only noteworthy step was the copy of the Concorde library folder in the root directory of our project, and the configuration of the CMakeLists file:

```
cmake_minimum_required(VERSION 2.8)

project(P_NP)

find_package(CPLEX REQUIRED)

set(CMAKE_C_STANDARD 99)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fopenmp")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ../bin)
set(CONCORDE_PATH ${PROJECT_SOURCE_DIR}/concorde/build)

INCLUDE_DIRECTORIES(include ${CPLEX_INCLUDE_DIRS} ${CONCORDE_PATH})

add_library(UTILITIES src/utilities.c)
add_library(LINKED_LIST src/linked_list.c)
add_library(PREPROCESSING src/preprocessing.c)
add_library(EXACT_TSP src/exact_tsp.c)
add_library(HEURISTIC_TSP src/heuristic_tsp.c)
add_library(MATHEURISTIC_TSP src/matheuristic_tsp.c)
add_library(SOLVER src/solver.c)

set(SOURCE_FILES src/main.c)

add_executable(TSP_SOLVER ${SOURCE_FILES})

target_link_libraries(TSP_SOLVER SOLVER)

target_link_libraries(SOLVER EXACT_TSP HEURISTIC_TSP MATHEURISTIC_TSP
UTILITIES PREPROCESSING LINKED_LIST m ${CONCORDE_PATH}/concorde.a
${CPLEX_LIBRARIES})
```

Using a static linking approach (*add_ library* command), we can create a standalone executable which can be run easily on the department cluster without a previous installation of all the required libraries.

Chapter 2

Exact methods

In order to find an optimal solution of a TSP instance, it is possible to formulate it as an integer linear programming problem (see section 1.1) and then solve it using a proper MIP solver. Observing the model, it is clear that it is not reasonable to add immediately all the connection constraints (1.1.4): generally there are $O(2^{|V|})$ bounds, which would make impossible for the solver to reach a solution because of time and memory limitations. Consequently, it is better to start with a simplified model, without the subtour elimination bounds, thus searching for a relaxed solution and progressively verifying if (1.1.4) is respected or not. This idea is used in all the exact algorithms that we implemented:

- loop method;
- heuristic loop method;
- lazy constraint callback;
- usercut callback.

2.1 Loop method

The loop method (see Listing 2.1) tries to reach the optimal solution progressively adding to the initial relaxed model the eventually violated subtour elimination bounds. When an optimal solution is reached and it contains just one connected component, it represents a global optimum and the algorithm terminates, otherwise the cycle continues.

```
LoopMethod(problem):

  do:
    MIPSolver(problem)

    *Retrieve the current_solution*

    *Find the connected components cc in the current_solution*

    If(|cc| > 1)
        *Add to the problem the proper subtour elimination constraints*

  while(|cc| > 1)

  return current_solution
```

Listing 2.1: Loop method.

Considering that the number of existing subtours is finite, the mathematical convergence of the loop method is assured, because in the worst case all possible subtours are added to the model. Given the number of distinct subtour constraints, this upper bound is not very reassuring, however, in practice, this approach is not as bad as one might think and the registered performances are comparable with

the other exact methods. The weak conceptual element resides in the fact that the same problem is iteratively solved from the beginning every time, just adding at each iteration some new constraints: the information obtained by the previous problem space explorations is lost and a new decision tree is created repeatedly.

2.2 Heuristic loop method

Considering the desire of speedup, an heuristic alternative to the loop method has been implemented (see Listing 2.2). The main goal is to try to reduce the initial space exploration, adding to the original model new constraints relative to the admissible edges.

```
HeuristicLoopMethod(problem):

  *Add heuristic bounds to problem*

  current_solution = LoopMethod(problem)

  *Remove heuristic bounds from problem*

  current_solution = LoopMethod(problem)

  return current_solution
```

Listing 2.2: Heuristic loop method.

An empirical idea is that the optimal tour might be formed by edges delimited by vertexes close to each other. The heuristic that we implemented selects a certain number n of shortest edges for each vertex of the graph: the number of variables that can be used to build a solution dramatically decreases from $O(2^{|V|})$ to $\approx n*|V|$, leading to a much faster exploration of the problem space by the first loop method. However, the optimality of the solution is not assured directly, so a further loop investigation is required: if the optimal solution was already reached, the pure loop method rapidly certifies it, otherwise the classical optimization occurs, starting from the heuristic solution if the MIP solver consider it beneficial. From an analytical point of view the performances in the worst-case scenario are the same as the loop method.

2.3 Lazy constraint callback

The main limitation of the discussed loop approaches is that new decision trees are created at each iteration by the MIP solver. A valid alternative is to check for the possibility to insert the proper subtour elimination constraints as soon as the solver has a new admissible solution, which consists of an integer solution of the relaxed problem: before the incumbent update, a proper function called *lazy callback* is invoked and eventually the bounds are inserted into the model, without interrupting the branch and bound algorithm.
These constraints are called *lazy* because they are not inserted in the initial model, but they are added only when necessary. The new bounds can be *local* or *global*, respectively if they are valid only for the subtree of the branch and cut where they are inserted or for the entire decision tree.
Considering that as long as the solver continues the exploration of the problem space new constraints can be added, it may happen that the new constraints are

more restrictive than the previous ones: setting a proper option, the MIP solver is allowed to delete the bounds that it considers slack, speeding up the exploration. However, in view of such purging, the user must not assume that any previously added constraint is still in the current relaxation and possibly old bounds could be added repeatedly.

2.4 Usercut callback

Lazy constraints can be added to the model only when an integer solution is reached. However, the insertion of proper bounds also in case of fractional solutions could be beneficial for the exploration. This lead to the concept of usercut callback, which is called during MIP branch and cut once a lower cost solution is reached.
Notice that the number of calls is generally much higher with respect to the lazy scenario, consequently the inserted callback should be really efficient in order to avoid bottlenecks in the solution process. Considering these performance requirements, we used the highly optimized Concorde callable library [1], which includes over 700 functions related to TSP-like problems. In our implementation we used the *CCcut_connect_components* and *CCcut_violated_cuts* functions, respectively to find the connected components and create a global minimum cut. To reduce the amount of cuts inserted in our model we randomized the addition, introducing a new constraint only in the 10% of the calls to the usercut callback function.

Chapter 3

Heuristic methods

The exact methods, presented in chapter 2, after the termination provide an optimal feasible solution with respect to the given problem. However, also if the solver is highly optimized and the adopted techniques try to minimize the complexity of the mathematical space exploration, if the problem size is large enough, it becomes computationally infeasible.
The so-called *heuristic methods* try to approximate the original problem in order to search for a feasible solution, renouncing to the guarantees about the optimality of the final solution found, certified only if the entire problem space is maintained. They don't use directly neither the mathematical model nor the MIP solver.
In this chapter, we present the following heuristic methods:

- Nearest Neighbor (NN);
- 2-opt algorithm;
- multistart;
- multi-threaded multistart;
- Variable Neighborhood Search;
- Simulated Annealing.

3.1 Nearest Neighbor

The Nearest Neighbor approach follows a greedy policy, building a tour using a best local move which can be chosen with different policies. The simplest version of the algorithm choses each time the closest node to the last visited one (see Listing 3.1).

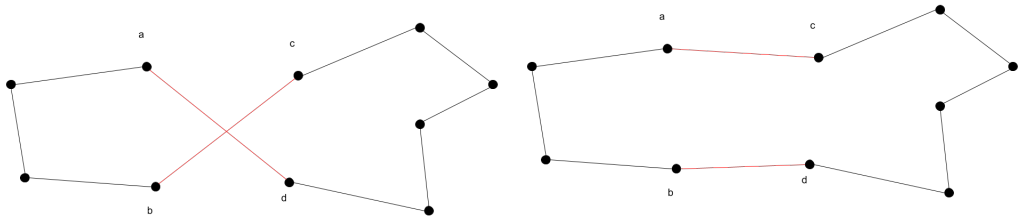
```
NearestNeighbor(problem):
    *Initialize all the problem nodes as UNVISITED*
    tour = empty tour
    start = ID of a random selected node from problem.nodes
    current = start
    problem.nodes[current] = VISITED

    while (not all problems.nodes are VISITED):
        closest = *Find the closest UNVISITED node from current*
        problem.nodes[closest] = VISITED
        Add the edge [current-closest] to tour
        current = closest

    Add the edge [current-start] to tour

    return tour
```

Listing 3.1: Simplest Nearest Neighbor algorithm.

This approach leads in general to poor tours because of the short-sighted idea that the closest node is always the best one. However, it can constitute a good starting point for other algorithms due to its simplicity and relatively low computational complexity.

3.2 2-opt algorithm

The 2-opt algorithm starts from an initial tour (provided for example through NN, section 3.1), which is modified using a series of 2-opt moves. A 2-opt move consists of an edge swap: it involves four vertexes and two edges and the idea is to invert the connection. The following figure helps to illustrate the concept:



(a) Tour before the 2-opt move.

(b) Tour after the 2-opt move on the red edges.

Figure 3.1: Example of a 2-opt move.

The complete series of 2-opt moves include also pairs of edges that does not crosses over themselves: if there is a reduction in the tour cost after the rearrangement of the edges, the move is still performed. The pseudocode for the 2-opt algorithm is the following one:

```
2-opt-optimization(tour):
    num_iterations = 0

    while (2-opt-swap(tour) is successful AND iter < max_iterations):
        num_iterations = num_iterations + 1

    return tour

2-opt-swap(tour):

    while(stop policy OR all pairs are explored):
      *Select two edges from tour*

      *Compute the cost of the alternative configuration (swap)*

      if(new cost < old cost):
        perform the swap on tour

    return tour
```

Listing 3.2: 2-opt algorithm.

We implemented two swap policies, which correspond to different stop policies in the 2-opt-swap algorithm:

- *first found swap*: the first swap found that provides a reduction on the tour cost is applied, independently of the amount of this update;
- *best swap*: explore all possible swaps and apply the best one in terms of cost reduction.

There are not clear proofs that one policy is better than another, because a local better update does not imply a successive faster exploration of the mathematical space. Clearly, the immediate computational cost of the first one is lower because as soon as a good move is found the pair exploration is interrupted. Following this motivation, we decided to apply the first found swap policy where the 2-opt algorithm is used.

3.3 Multistart

The 2-opt algorithm tries to repeatedly improve through 2-opt moves the current solution up to the provided time limit or the absence of new good moves, as described in the previous section. However, depending on the given problem and on the particular starting point, the 2-optimality neighborhood could be explored rapidly considering the computational power of the current processors. Under these circumstances, it is useful to consider new techniques which could exploit completely the given computational time.
The Greedy Randomized Adaptive Search Procedure (GRASP)[4] is an iterative randomized sampling technique in which each iteration provides a solution to the given problem. The traditional procedure alternates a fast construction of an initial solution (typically using a low cost heuristic algorithm) to a local search procedure, which tries to improve it. The exploration is repeated up to the termination condition (e.g. time limit) and the best solution over all GRASP iterations is kept as the final result. The Multistart algorithm (see Listing 3.3) is a particular realization of this GRASP approach.

```
Multistart(problem):

    best_tour = empty_tour()
    best_tour.cost = infinity
    local_tour = empty_tour()

    while (time limit not expired):
        local_tour = *Heuristic_tour_creation(problem)*
        *local_optimization(local_tour)*

        if(local_tour.cost < best_tour.cost)
            best_tour = local_tour

    return best_tour
```

Listing 3.3: Multistart algorithm structure.

The implementation of the algorithm can be specialized choosing a specific heuristic and local optimization algorithm. An easy method that could be used to build a valid tour is the Nearest Neighbor algorithm (Listing 3.1), while the 2-opt algorithm (Listing 3.2) could optimize the starting solution.
A distinctive characteristic of the heuristic algorithms is their intrinsic randomness, which can be exploited to iteratively explore in different manners the complex mathematical space of the TSP problem. However, the proposed Nearest Neighbor algorithm introduces a limited variability in its behavior, because the only random contribution comes from the choice of the initial starting point, but the subsequent steps are deterministic. Consequently, the same tour is generated over multiple executions which start from the same node. Furthermore, there is the not negligible probability that also the choice of close different initial points would lead to an equal

solution.
Considering this desire of randomization, the algorithm can be modified (see Listing 3.4) in order to make it choose a different point than the closest one with a certain probability p $>$ 0 (for example the second or third closest node could be chosen).

```
RandomNearestNeighbor(problem, p):
    *Initialize all the problem nodes as UNVISITED*
    tour = empty tour
    start = ID of a random selected node from problem.nodes
    current = start
    problem.nodes[current] = VISITED.

    while (not all problems.nodes are VISITED):
        next =  *The ID of
                    - the closest UNVISITED node with probability p
                    - the second/third closest UNVISITED node with probability 1-p*
        problem.nodes[closest] = VISITED
        Add the edge [current-closest] to tour
        current = closest

    Add the edge [current-start] to tour

    return tour
```

Listing 3.4: Random Nearest Neighbor algorithm

This version is more convenient with respect to the GRASP scenario, because it reduces the probability that the same starting tour is uselessly optimized multiple times. In the Listing 3.5 is presented the pseudocode of the algorithm that we implemented in our project.

```
Multistart(problem,p):

    best_tour = empty_tour()
    best_tour.cost = infinity
    local_tour = empty_tour()

    while (time limit not expired):
        local_tour = RandomNN(problem,p)
        2-opt-optimization(local_tour)

        if(local_tour.cost < best_tour.cost)
            best_tour = local_tour

    return best_tour
```

Listing 3.5: Multistart algorithm.

3.4 Multi-threaded multistart

The Multistart algorithm (see Listing 3.5) works at each iteration on a new random starting solution, optimizing it independently with respect to the previous ones. This led to the idea of parallelizing the algorithm, in order to exploit the multi-threaded architecture of modern processors.
OpenMP (Open Multi-Processing) [9] is an Application Programming Interface (API) which allows to easily introduce a certain level of parallelism in C, C++

and Fortran programs. However, the developer is still responsible for correctly using this tool, avoiding pitfalls like uncontrolled concurrent accesses, deadlocks or race conditions. In this context, the required OpenMP knowledge to parallelize the Multistart algorithm is really limited to the creation of a parallel code segment, but to better understand the approach that we used, it is useful to insert the following piece of code:

```
1  int solve_heuristic_tsp_multi(tour *best_tour)
2  {
3      //Starting time
4      struct timespec wall_start = wallStart();
5
6      //Problem nodes
7      int nnodes = best_tour->instance_problem->nnodes;
8
9      //Number of available threads
10     int n_threads = omp_get_num_procs();
11
12     //Create and initialize a local and a best tour structure for each
           executing flow
13     tour **thread_local = (tour **) malloc(n_threads * sizeof(tour *));
14     tour **thread_best = (tour **) malloc(n_threads * sizeof(tour *));
15     for (int i = 0; i < n_threads; ++i)
16     {
17         thread_local[i] = (tour *) malloc(sizeof(tour));
18         thread_best[i] = (tour *) malloc(sizeof(tour));
19         copy_instance(best_tour, thread_local[i]);
20         copy_instance(best_tour, thread_best[i]);
21     }
22
23     //Start the parallel OpenMP section
24     #pragma omp parallel num_threads(n_threads)
25     {
26         //Choose random seeds using the thread id combined with prime
               numbers
27         int thread_id = omp_get_thread_num();
28         unsigned int seed = 42589 + 17 * thread_id;
29         unsigned int start_seed = 51637 + 1303 * thread_id;
30
31         //Classic Multistart solution approach
32         while((wallEnd(wall_start) < best_tour->timelimit)
33         {
34             clear_tour(thread_local[thread_id]);
35
36             thread_local[thread_id]->start_node = rand_r(&start_seed) %
                   nnodes;
37
38             nearest_neighbor(thread_local[thread_id], thread_local[
                   thread_id]->start_node, &seed);
39
40             opt_2(thread_local[thread_id], 0);
41
42             if (thread_local[thread_id]->cost < thread_best[thread_id]->
                   cost)
43             {
44                 copy_tour(thread_local[thread_id], thread_best[thread_id]);
45             }
46         }
47     }
48
49     //Among the explored solutions, find the best candidate
```

```
50      int best = -1;
51      double best_cost = HUGE_VAL;
52      for (int i = 0; i < n_threads; ++i)
53      {
54          if (thread_best[i]->cost < best_cost)
55          {
56              best_cost = thread_best[i]->cost;
57              best = i;
58          }
59      }
60      copy_tour(thread_best[best], best_tour);
61
62      //Free the occupied memory
63      for (int i = 0; i < n_threads; ++i)
64      {
65          free_tour(thread_local[i]);
66          free_tour(thread_best[i]);
67      }
68
69      return 0;
70  }
```

Initially an appropriate number of support structures are created: each thread owns two tour instances, which are used in the local exploration. The parallel section is represented by the block of lines 24–47: OpenMP creates a team of *n_ threads* threads and when everyone exits from the cycle, they join back into one. Each thread independently stores the current and best TSP tour and, once the time limit is reached, the costs of each local best tour are compared and the lowest one is chosen as the global best tour. It is convenient to underline that each thread has a different initial random seed, which allows to differentiate the random nearest neighbor solutions, hopefully increasing the explored area. In this context the classic usage of the *rand* function is not possible, because it is based on a single random seed, which would lead to uncontrolled concurrent accesses and an unpredictable system behavior. The performance improvement introduced using this approach naturally is not exactly proportional to the number of threads, because of the necessary synchronization overhead and the usage of the hyperthreading technology.

3.5 Variable Neighborhood Search

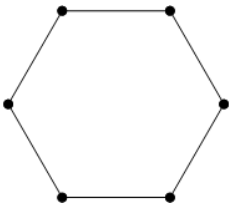
Variable Neighborhood Search (VNS) [7] is an heuristic method that can be used to locally optimize a global optimization problem. Differently from the Multistart approach, which tries to reach better solutions starting at each iteration from a new random initial one, it introduces variability performing an alternative improving swap move to exit from the 2-optimality neighborhood.

```
VNS(problem):

    best_tour = empty_tour()
    best_tour.cost = infinity
    local_tour = NearestNeighbor(problem)

    while (time limit not expired):

        2-opt-optimization(local_tour)

        if(local_tour.cost < best_tour.cost)
            best_tour = local_tour
```

```
        3-opt-move(local_tour)

    return best_tour
```

Listing 3.7: VNS algorithm.

Starting from an initial solution, built for example using the Nearest Neighbor algorithm, each iteration provides a complete exploration of the 2-optimality neighborhood of the current solution. However, once the first optimization ends, the execution of a random 3-opt move allows to create a new starting solution which is outside from the already explored 2-opt neighborhood. A 3-opt neighborhood completely contains a 2-opt neighborhood centered at the same solution, but not vice versa: the subsequent 2-optimization will occur on a new region which is (partially) unexplored. The usage of this kind of perturbation through an higher order swap move introduces enough variability to use effectively the 2-optimization as local search, without remaining confined to a small solution region.
Notice that an even higher k-opt neighborhood move could be applied ($k \geqslant 3$), but particular attention should be directed to the choice of the random k-opt move, because some of them could be simply the combination of more 2-opt moves, leading to a false new starting solution strictly contained in the 2-opt neighborhood already explored. For example in Figure 3.2 all the eight possible 3-opt moves are represented, but just four of them (e,f,g,h) are actually different from a combination of 2-opt swaps. In our implementation, without loss of generality, we use the configuration h to randomly selected edges which are compatible with this move (adjacent edges are not admissible).



Original configuration.

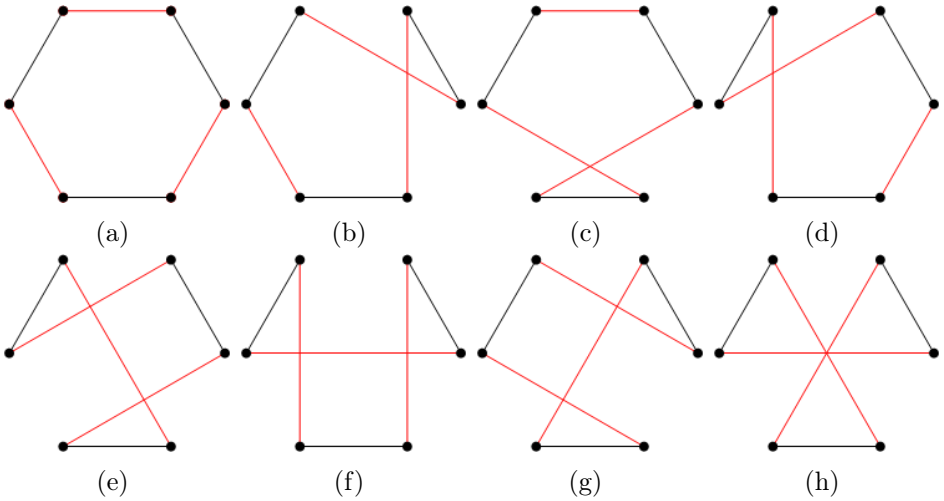


Figure 3.2: Example of 3-opt swap moves.

3.6 Simulated Annealing

The name and the idea come from annealing in metallurgy, a technique involving the controlled cooling of a material (typically metal alloy) to improve the size of the microscopic crystals and therefore augment the strength and the resistance of the material. The controlled annealing of the material minimize the energy of the system, concept that can be translated in our context to the minimization of the cost function. The physical temperature indicates the reactivity of the system to phase transitions, concept that in our scenario is represented through a parameter T, which expresses the change of the cost function.
Typically, the function that gives the temperature at each step of the algorithm is a convex monotonically decreasing one, with more steps spent at lower temperatures (to simulate the physical behavior of temperature decrease).
A common accepting function for a candidate tour at the iteration i appears to be:

$$P\big(c(T_i), c(T_{i-1}), t_i\big) = e^{-\frac{c(T_i)-c(T_{i-1})}{t_i}} \tag{3.6.1}$$

where $c(T_i)$ is the cost of the candidate tour, $c(T_{i-1})$ is the cost of the old tour and t_i is the temperature at the i-th iteration.
Applied to the TSP, the simulated annealing process works as follows:

1. Start with a random tour through the nodes, which is likely to be a very bad tour.
2. Pick a new candidate tour T' at random from the neighbors of the existing tour T and calculate its cost.
3. There may be be two cases about the choice of the new tour, depending on the cost of the candidate tour:
   - Lower than the old one: we accept it right away.
   - Greater than the old one: we may still accept it, according to some probability. The probability of accepting a worse tour is a function of the difference between the cost of the candidate and the current tour, and the temperature of the annealing process (higher temperature makes the algorithm more likely to accept an inferior tour).
4. Diminish the temperature and go back to step 2, until the temperature has reached its minimum value. When the cycle ends return the sequence of nodes forming the tour and it's cost.

Implementation
In our implementation, we represent the tour as an ordered sequence of nodes, thus simplifying the generation phase of a new candidate.
To generate a new candidate tour T', we take the old tour T and randomly choose two indexes i and j such that $0 \leq i < j \leq |nodes|$; we then reverse the portion of the tour that lies between i and j, therefore performing a 2-opt move (see section 3.2).
We fixed the number of iterations to $5 \cdot 10^6$ and consequently the function that gives the temperature at each iteration i is $t(i) = e^{-3.68 \cdot 10^{-6} \times i} - 10^{-8}$ (see Figure 3.3).
These choices (number of iterations and temperature function) have been made because they seem to work reasonably well in our case, but no rigorous demonstration can be produced.

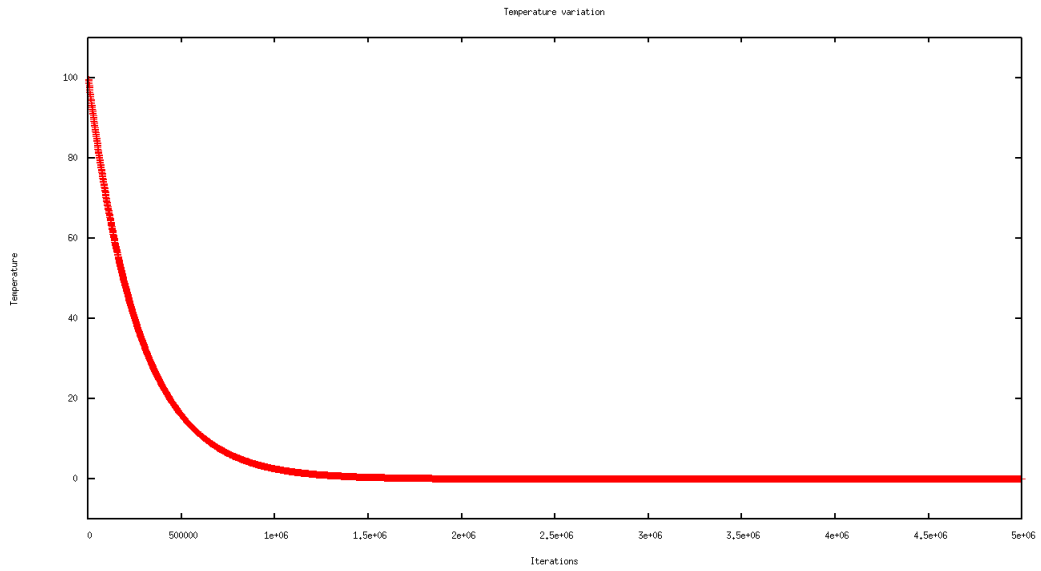
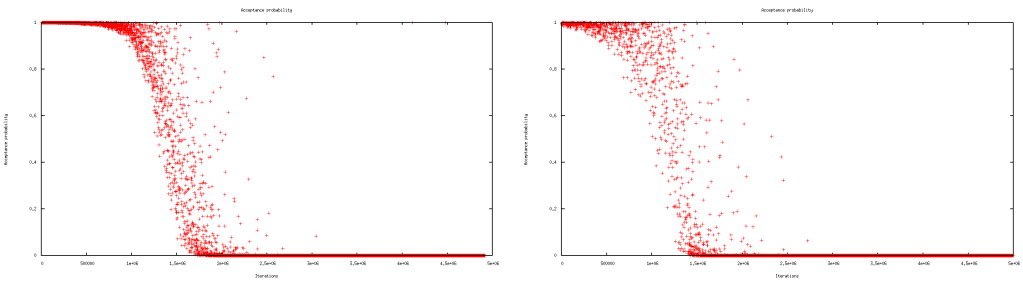


Figure 3.3: Temperature decrease to the increasing iterations.

Thanks to an idea of our professor Matteo Fischetti, we slightly changed the formula 3.6.1, dividing the costs at the exponent by their means, thus normalizing the expression:

$$P\big(c(T_i), c(T_{i-1}), t_i\big) = e^{-\frac{2 \cdot c(T_i) - c(T_{i-1})}{t_i \cdot c(T_i) + c(T_{i-1})}} \tag{3.6.2}$$

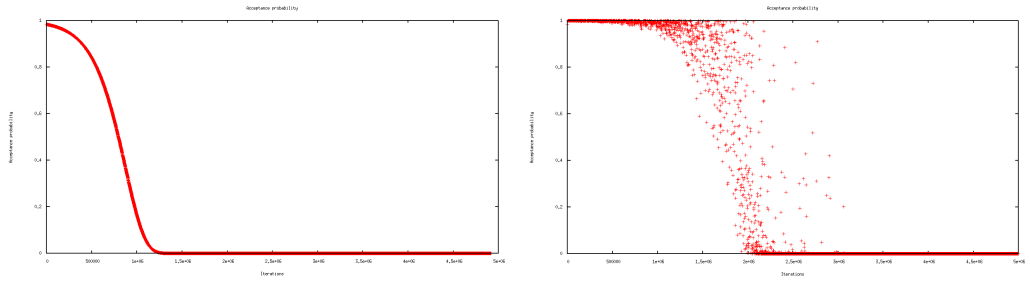
This change did not have a big impact on small instances, as you can see in Figure 3.4, where the probability distribution remained substantially unchanged.



(a) Standard formula 3.6.1. (b) Normalized formula 3.6.2.

Figure 3.4: Comparison of probability distributions with the standard formula and the normalized one. Instance dimension of 130 nodes.

However, in the instances with many nodes (400+) the different results of the two formulas are clear from the Figure 3.5: the standard equation 3.6.1 needs to be adjusted (either changing the temperature function or increasing the numerator of the exponential) as the number of nodes increases, while the tweaked equation 3.6.2 always delivers consistent results as the number of nodes changes.
As we can seen from Figure 3.5a the probability to accept a worse tour drops to zero too quickly, not giving enough time to the algorithm to move from the neighborhood of the temporary solution. This event therefore has the effect of freezing the future tours to a certain neighborhood, very often a not promising one.

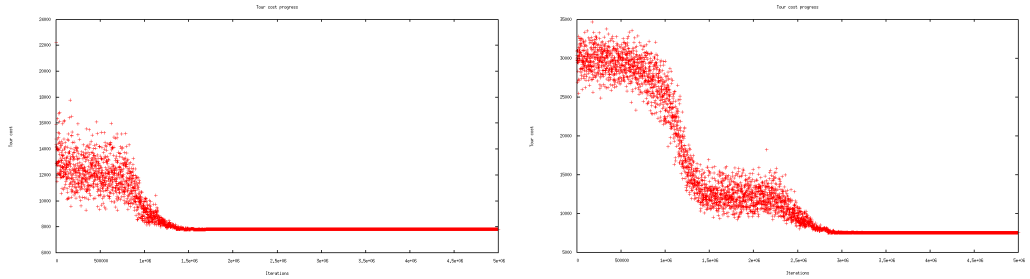


(a) Standard formula 3.6.1.

(b) Normalized formula 3.6.2.

Figure 3.5: Comparison of probability distributions with the standard formula and the normalized one. Instance dimension of 1291 nodes.

We can see the occurrence of this effect also with smaller instances, looking at the graphs of the tours cost. Examining the Figure 3.6, the costs quickly settle down in the case 3.6a while they are more free to vary in the case 3.6b, eventually reaching a better final result (for this instance the final costs are: 7620 for 3.6a and 7342 for 3.6b).



(a) Tour costs with 3.6.1.
Final cost: 7620.

(b) Tour costs with 3.6.2.
Final cost: 7342.

Figure 3.6: Comparison of the costs behavior with the standard formula and the normalized one. Instance dimension of 575 nodes.

We can conclude that, for our scopes, the improved version of the probability function 3.6.2 yields on average better results, so in the test section we used this one.

Chapter 4

Matheuristics methods

Matheuristics are optimization algorithms made by the inter-operation of metaheuristics and mathematical programming (MP) techniques. They allow to mix the useful aspects of both worlds, using the heuristic approach to find an initial solution and a mathematical programming algorithm as a black box tool to improve it.
Considering that the TSP problem is NP-hard, the research of an exact optimal solution over problems of medium-big size is generally unfeasible. On the other side, an heuristic approach can frequently offer a good initial approximation but, due to the intrinsic randomness and the wide solution space, it is pretty hard to reach directly a local optimal solution. The idea is to use the MIP solver to progressively explore some neighborhoods of the initial heuristic solution, taking advantage of an initial heuristic hint and subsequently of the power of the mathematical formulation. However, due to the natural difficulty of the target problem, no guarantees about the quality of the solution in terms of closeness to the global optimum can be provided, because the solver only finds the optimum solution with respect to the solution subspace generated by the problem formulation, which is a downsized version of the canonical one (see section 1.1).
The scientific community proposed many matheuristic solutions. The most famous are:

- Hard Fixing;
- Local Branching;
- R.I.N.S.;
- Polishing Heuristic;
- Proximity Search.

We chose to implement the hard fixing and local branching approaches.

4.1 Hard Fixing

The Hard Fixing approach does not add new constraints to the original mathematical model, but it tries to simplify the research of a solution fixing a certain amount of variables. Starting from a feasible solution, obtained for example through a pure heuristic algorithm, the idea is to explore its neighborhood, but, considering the properties of the problem, not through enumeration. Specializing this idea to the TSP problem, fixing a variable to 1.0 means to force the inclusion of certain edges in the solution provided by the MIP solver.
There are many possible approaches that could be used to choose which edges should be fixed and which should be let free. For example, one could think to discriminate the edges applying a certain distance threshold or searching for patterns, but there are not clear winners with respect to the optimal solution. The most simple idea is to randomly and independently fix a certain amount of edges, starting with an high percentage (e.g. 80%) and progressively reducing it if after a certain amount

of iterations if the best current solution can't be improved. Clearly, the complexity of the task assigned to the MIP solver increases with the degrees of freedom, so it is convenient to start with an high amount of fixed edges in order to rapidly improve the solution and, at least, reach rapidly a feasible one, which can be provided to the user if the time limit is particularly restrictive.
In Listing 4.1 is presented the pseudocode of the hard-fixing approach.

```
HardFixing(start_tour):

    current_tour = start_tour
    best_tour = start_tour
    best_cost = current_tour.cost

    *Provide to the MIP solver the start_tour solution as Warm Start*

    prob = 0.8
    prob_trials = 0

    while(time limit not expired and prob >= 0):
        for each edge e in current_tour:
            fix e to 1.0 with probability prob in the MIP model

        *Execute the polish_algorithm*

        current_tour = *Perform the optimization using the MIP solver*

        if(best_cost <= current_tour.cost):
            prob_trials = prob_trials + 1

            if(prob_trials > max_prob_trials):
                prob = prob - 0.2
                prob_trials = 0
            else:
                best_cost = current_tour.cost
                best_sol = current_tour

         *Reset the variable boundaries in the MIP model*

    return best_sol
```

Listing 4.1: Hard fixing algorithm.

In our specific case, the initial feasible heuristic solution provided to Cplex is called a *Warm-Start*: the MIP solver is able to apply its preprocessing and analyze this solution, which could be used as a starting point for the optimization process. By default, Cplex finds a valid solution starting just from the input model, but in complex scenarios this could be a difficult task: an heuristic algorithm designed specifically to solve a certain problem takes advantage of a domain knowledge which is not accessible to a generic MIP solver. Hopefully the heuristic solution is closer to the optimal one or, at least, should provide a better start, although there are no formal guarantees. However, notice that providing a Warm-Start we are not forcing Cplex to consider it as root node: if its proprietary preprocessing techniques are able to find a better starting point, the initial heuristic solution is simply ignored.
Finally it is useful to analyze better what we called the *polish_ algorithm* in Listing 4.1. In general, starting from a solution with randomly fixed edges leads to the creation by the MIP solver of a first solution with different connected components and, consequently, the subtour elimination conditions are progressively added to the model (see section 2.3). However, we can immediately see that certain edges

will lead to the creation of these separated groups, so with an efficient algorithm we can force other boundaries in the model to avoid worthless MIP efforts. The *polish_ algorithm* has the goal of finding these edges and forcing them to zero.

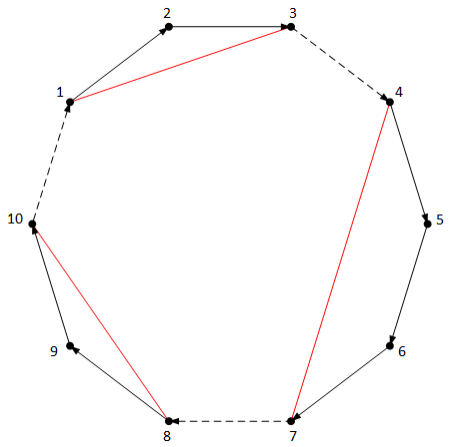


Figure 4.1: Example of the *polish_ algorithm* behavior.

In Figure 4.1 is depicted an example of the execution of the *polish_ algorithm*: the dotted lines represent the edges which are kept free, the red lines represent the edges which would create separate connected components, while the black lines represent the edges forced to 1.0. Starting from a random node and following the input tour, the algorithm detects the forbidden red edges and fixes them to 0.0 in the model, in order to force the MIP solver to find an alternative valid solution, without the addition of the related connection constraints.

4.2 Local Branching

The Local Branching is a matheuristic technique invented in 2003 by M. Fischetti and A. Lodi [6]. The main idea behind this technique is to add linear inequalities to the mathematical model, the so called *soft fixing constraints*, defining a suitable neighborhood of the current solution, which can be explored by the solver. The algorithm takes advantage of an initial solution x^H, called *Warm Start*, which can be found with any of the heuristic methods seen in chapter 3.
The PLI model for the TSP remains the same as the one in section 1.1, only a further constraint is added, which can be expressed in different ways; for example if we want to fix to 1.0 at least 80% of the edges of x^H, the constraint would take the following form:

$$\sum_{(i,j)\,:\; x^H_{ij}=1} x_{ij} \geq \lceil 0.8 * |nodes| \rceil, \quad i,j \in V \tag{4.2.1}$$

In our algorithm, however, we chose to express the right hand side of the inequality as a function of the parameter r, which represent the degree of freedom of the solution compared to x^H: this means that at most r edges of x^H can be changed.

$$\sum_{(i,j)\,:\; x^H_{ij}=1} x_{ij} \geq |nodes| - r, \quad i,j \in V,\ r \in \mathbb{N} \tag{4.2.2}$$

Unlike the Hard Fixing then, the variables to keep in solution are chosen by the mathematical model, avoiding a too rigid fixing of the variables in favor of a more

flexible condition.
The MIP solver is then started: if it finds a better solution this becomes the new incumbent (guaranteed to be the best of the neighborhood), the row of the matrix relative to Equation 4.2.2 is deleted and a new one is added to the model keeping the same value for r. The exploration of the neighborhood of the just found solution is then restarted.
If the solver does not find a better solution (or the time-limit reserved to the solver has expired), the tour is retrieved and the Equation 4.2.2 is updated with a bigger value of r, thus increasing the neighborhood dimension and therefore giving more freedom to the model to change the original solution (at the price of increasing the computational time).
Some implementation choices, regarding whether to stop at a certain value of r and the set of values that r can assume, fall into the tuning phase, as they are dependent on the problem and on the instance. For our purposes we have chosen the values $\{3, 5, 10\}$.
The pseudocode relative to the algorithm that we implemented in our project is presented in the following Listing:

```
LocalBranching(problem):

    tour x_h = *Call an heuristic method on the problem*
    *Build the MIP problem and add x_h as a Warm Start*

    optimum_reached = false

    while (time limit not expired && !optimum_reached)
        *Add the local branching constraint*

        *Start the MIP solver*

        *Retrieve the solution*

        if (sol.best_val < curr.best_val)
            curr.best_val = sol.best_val
        else
            if (current_r == 3)
                current_r = 5
            else
                if (current_r == 5)
                    current_r = 10
                else
                    optimum_reached = true

        *Delete the old local branching constraint from the model*

    return curr.best_tour
```

Chapter 5

Tests

For the development of our algorithms as well as the final tests, we used the TSPLIB library [8] provided by the Heidelberg University, a *standard de facto* in the field. In particular we used the *EUC-2D problems*, each one defined by a set of nodes in $\mathbb{R}^2$, and the distance metric between them is the integer rounding of the euclidean distance. Each problem is provided through a textual file divided in two sections: the *specification* part presents various information on the file format and its content, while the *data* part contains the node coordinates.
For our purposes we considered two different types of "time": the *wall time* is defined as the amount of real time elapsed from the start of the optimization up to the end, whereas the *cpu time* is defined as the amount of time for which a CPU is actively used for processing instructions of a particular process (in a multi core system, the CPU time is the sum of the times registered over all cores). In a single core CPU the wall time is always greater or equal than the CPU time, due to the OS management.
The tests were executed on the department cluster and for consistency in the final evaluation, we ran all of them on the same blade machine of the cluster, equipped with an 8 core processor (2 quad core Intel Xeon E5450 @3.00 GHz with 12MB Cache) and 16 Gb of RAM, running Linux x64 as Operating System.

5.1 Test description

5.1.1 Exact algorithms

From the TSPLIB we selected all the 37 instances having less than 500 nodes as our testbed for the exact methods; for every method (loop_method, lazy_callback and usercut_callback) 15 runs have been executed (corresponding to 15 different seeds), all with the wall time-limit set to one hour. We then calculated the average of the 15 runs for each (exact) method and instance, calculating separately wall and cpu time, obtaining all the entries of Table 7.1.

5.1.2 Heuristic algorithms

From the TSPLIB we selected 19 instances with a number of nodes ranging from 500 to 2000 as our testbed for the heuristic methods. In graphs and tables we compared four heuristic methods: multistart, multi-threaded multistart, VNS and simulated annealing, corresponding to the labels seq-multi, par-multi, VNS and sim-ann.
For every method 10 runs have been executed (corresponding to 10 different seeds), all with the wall time-limit set to 30 minutes. We lowered the number of runs compared to the exact tests for logistical reasons (the exact tests took us too much time), then we calculated the averages of the various runs, forming the entries of the Table 7.2.

5.1.3 Matheuristic algorithms

From the TSPLIB we selected 20 instances with a number of nodes ranging from 500 to 2000 as our testbed for the matheuristic methods; for every configuration

15 runs have been executed (corresponding to 5 different seeds and 3 different time ratios), all with the wall time-limit set to 30 minutes. A configuration is the result of the combination of an heuristic algorithm (seq-multi, par-multi and VNS) followed by either hard fixing or local branching, for a total of 6 different alternatives. A variable ratio of the time-limit has been given to the heuristic part of the algorithms to compute the warm-start, specifically 0.1, 0.2 or 0.3.
Finally, we calculated the average for each corresponding run, dividing the results in Table 7.3 for hard fixing and Table 7.4 for local branching.

5.2 Test results

5.2.1 Exact algorithms

Considering the exact algorithms, we used a benchmarking software to compare the proposed algorithms, which takes advantage of the performance profiles functions presented in [3]. The comparison depicted in Figure 5.1 is relative only to the wall time, because it is the metric used to fix the execution time limit. Notice that if a particular instance exceeded the time limit of one hour, its contribution is not considered in the algorithm profiling.

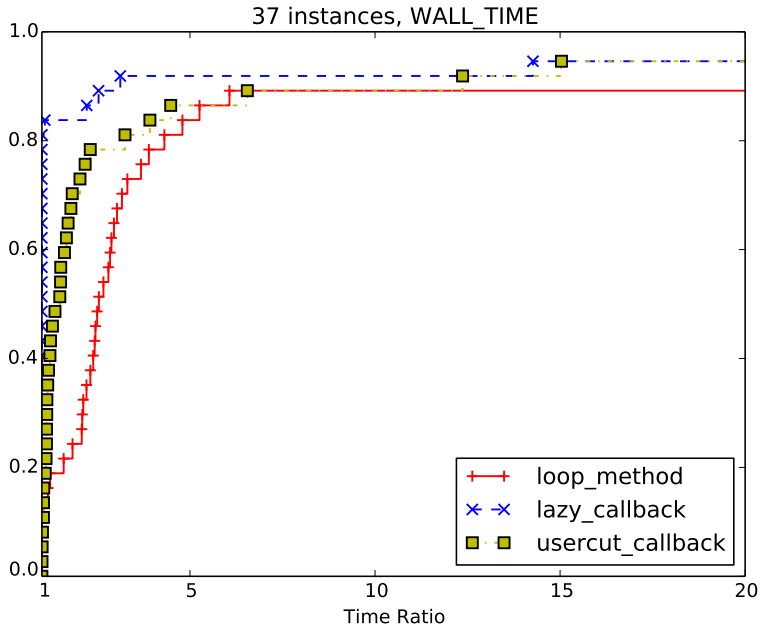


Figure 5.1: Performance profile for the exact methods, wall time.

In order to interpret the previous graph is useful to briefly recap the profiling concept, leaving the reference paper [3] as a more complete and accurate source for the interested reader. The goal of the analysis is to compare the tested algorithms transversally to the set of instances: by how much the best algorithm, with respect to each instance, is faster then the other ones?
Let I and A be respectively the set of instances and algorithms examined. Call $t_{i,j}$ the average time (over various runs with different random seeds) obtained for instance $i \in I$ and method $j \in A$, and let $B_i = \underset{j \in A}{argmin}\{t_{i,j}\}$ be the fastest algorithm with respect to each instance i. Define the Time Ratio (TR) as a real number >= 1,

represented in the chart as the horizontal axis. As the value of TR increases, $\forall j \in A$, the software counts how many times the j-th algorithm performed better than the best algorithm times TR, and represents the result in the vertical axis.
Fixed TR, compute:

$$p_{TR,j} = \frac{|\{i : \quad t_{i,j} \leq t_{i,B_i} * TR, \quad \forall i \in I\}|}{|I|} \quad \forall j \in A$$

which represents the performance of the j-th algorithm with respect to time ratio TR, compared to the best one. Connecting all the points obtained for different TR values, we obtain a curve for each algorithm. Consistently with the previous definition, when $TR = 1$, the y coordinate of a point is the number of times the corresponding algorithm is faster than the others, across all instances $i \in I$.
An higher curve increasing TR represents a wider gap with respect to the observed algorithms and indicates better performances transversally to all the test instances. Observing the Figure 5.1, the lazy_callback method provides the best performances, followed by the usercut_callback and loop_method. This could be due to the fact that the lazy callback is called much less than the usercut callback, because the former is invoked when the MIP solver finds an integer feasible solution for the LP model, while the latter is also called when a fractional solution is reached. Furthermore, in order to reduce the amount of constraints added to the model, in 90% of the calls to the usercut function nothing is done, but a certain invocation overhead is still present. However, a different implementation of these two methods or the application to a different test set could bring to different results.

5.2.2 Heuristic and Matheuristic algorithms

In the following analysis we grouped together the heuristic and matheuristic results for two reasons:

- both approaches don't provide guarantees about the solutions found.
- we set a common time limit of 30 minutes in order to consistently compare the obtained results.

As we can see from the Figure 5.2, simulated annealing (**sim-ann**) does not provide results in line with other heuristic methods, and the motivation is to be sought in the formulation of the algorithm itself: the primary cycle runs for a fixed number of iterations instead of running until the time limit is expired, due to the fact that the temperature is a function of the number of iterations. At the same time, a premature termination of the algorithm due to the time limit expiration would lead to a poor final refinement of the incumbent solution. Consequently we chose the number of iterations in order to remain within the time limit, giving up some computation but assuring the correct execution. Considering this setup, a fair comparison with the other heuristic methods could not be carried out since simulated annealing has systematically less execution time compared to the other methods.

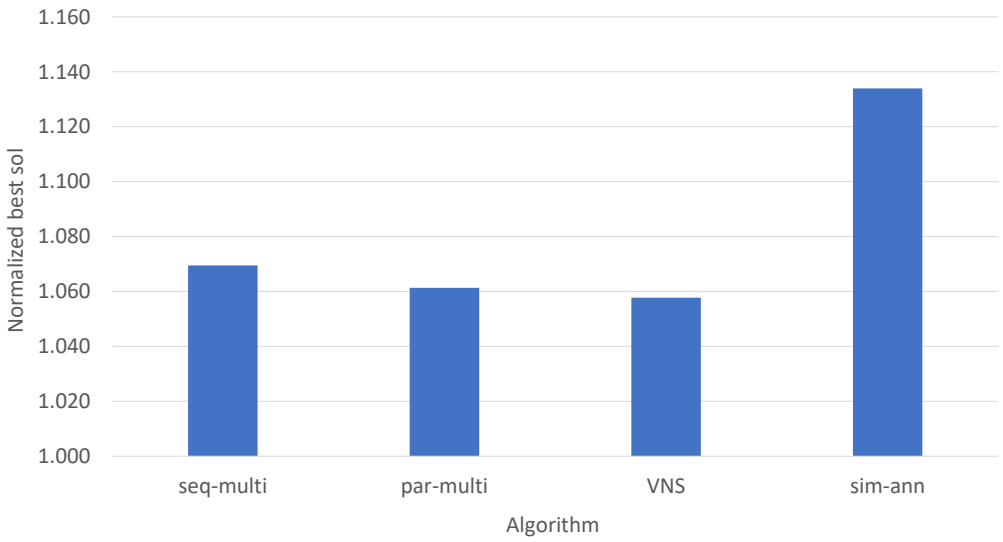


Figure 5.2: Normalized solution values for heuristic methods (the value 1 is the optimum).

An alternative way to execute our tests would be to fix the time limit of other heuristic methods equal to that necessary for simulating annealing: this approach though, besides than being complicated to implement due to the fact that the execution time of simulated annealing is not constant during repeated runs, results hardly reproducible in the future or on different machines.
Regarding the other methods, as we expected the multithreaded version (**par**allel-**multi**start) of the multistart algorithm (**seq**uential-**multi**start) on average produces better results: the problem space explored by the parallel algorithm under the same time limit is possibly wider, so the potential to reach different solutions should be higher. The quality of the best solution strictly depends on the likelihood that a good region of the problem space is randomly reached by the initial tour creation, because the capability of optimization provided within each iteration is limited by the 2-opt routine to the 2-optimality neighborhood.
The average performance of par-multi and VNS algorithms are comparable within our test set. Again, the intrinsic randomness in these algorithms does not allow an absolute declaration of a winner, because there are different aspects which could play a positive and negative role at the same time. VNS starts from a unique random solution and progressively tries to improve using a 2-optimization and to explore the space using random 3-opt moves. If the starting point is unfortunate, the time required to move away from that region could be high, leading to best solutions far from a global optimum. However, if the starting point is favorable, the algorithm insists in that region because generally it is not able to move too far away. On the other hand, par-multi limits its exploration to the 2-optimality neighborhood and then tries to explore a new random region, not related to the previously explored ones, which leads to the symmetric considerations with respect to VNS.

In the following sections we considered par-multi and VNS as the reference heuristic methods for their superior results (although in the table chapter at the end of this report are reported also the other combinations of heuristic and matheuristic methods).

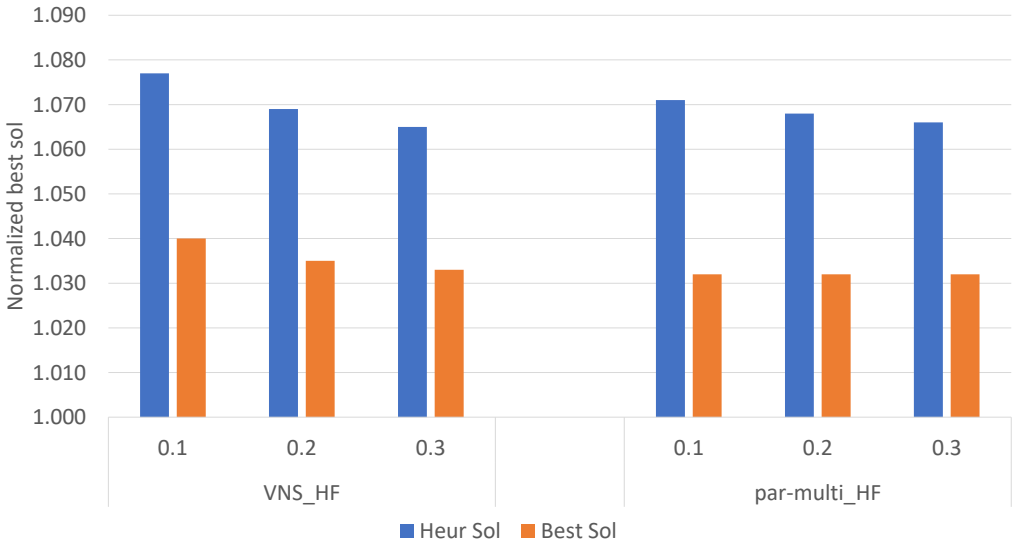


Figure 5.3: Hard Fixing comparison with varying time ratio.

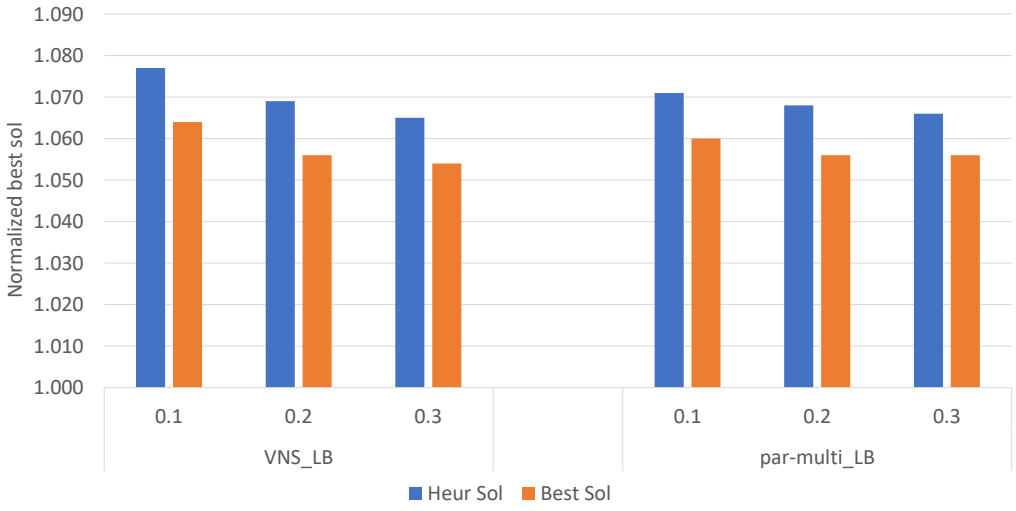


Figure 5.4: Local Branching comparison with varying time ratio.

Looking at the Figure 5.3 and Figure 5.4 we can notice how giving more execution time to the heuristic algorithms to find a warm start (blue bars on the graphs) on average partially improves the final solution found by the matheuristic algorithms (orange bars on the graphs).
In particular there is a major gain in changing the time ratio from 0.1 to 0.2, while the same can not be said for the change from 0.2 to 0.3, as there is only a subtle improvement on the final solution.

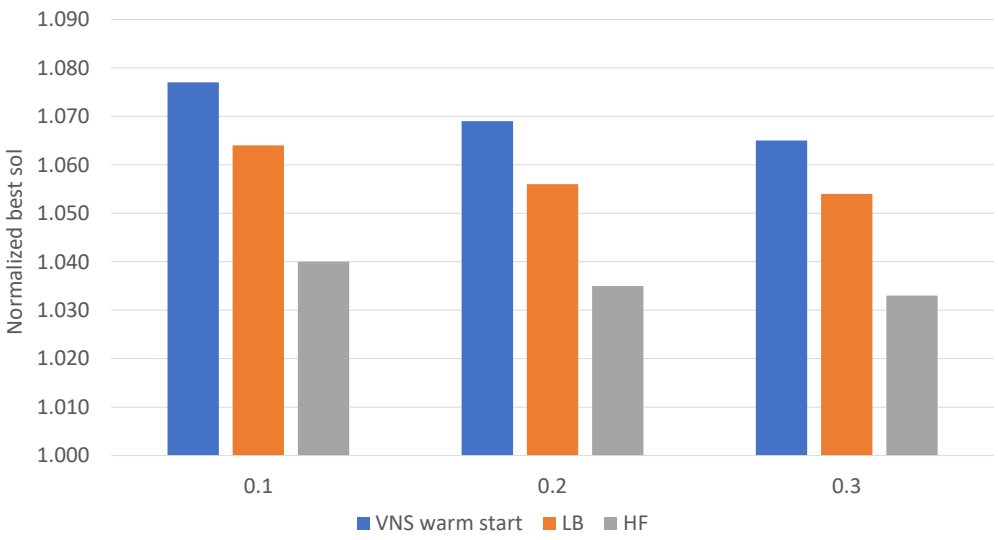


Figure 5.5: Hard Fixing and Local Branching comparison using VNS as heuristic algorithm.

From Figure 5.5 it is pretty clear the different performance between Hard Fixing and Local Branching, with equal warm start (we chose VNS as it is slightly better than par-multi): from our implementation of both matheuristic methods and with the selected testbed, the former yields better solutions than the latter.
This can be seen also from Figure 5.6 (which recalls the Figure 5.2, removing simulated annealing): the average performance of hard fixing is significantly better than the heuristic methods, while the average performance improvement of local branching compared to the heuristic methods is subtle.

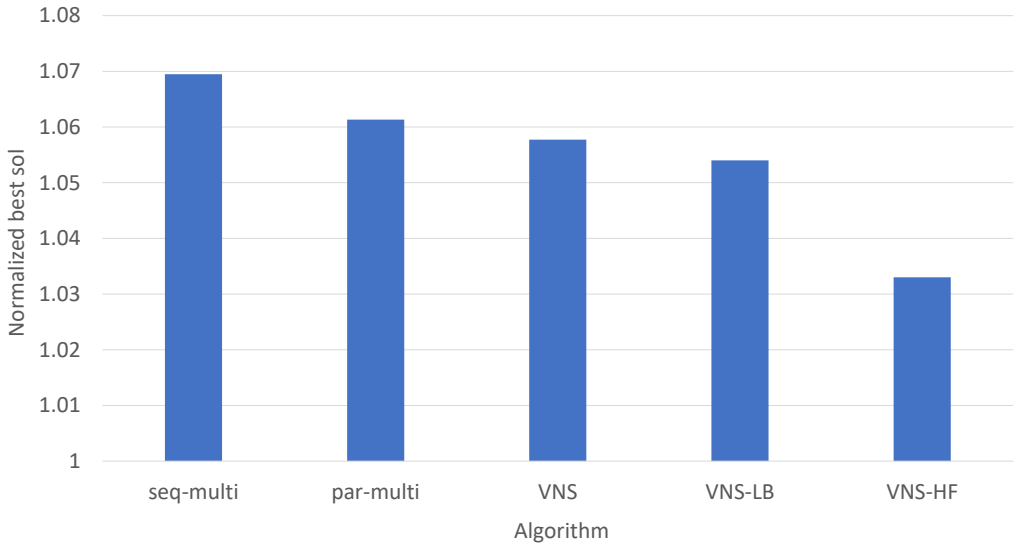


Figure 5.6: Normalized solution values for heuristic and matheuristic (time ration of 0.3) methods (the value 1 is the optimum).

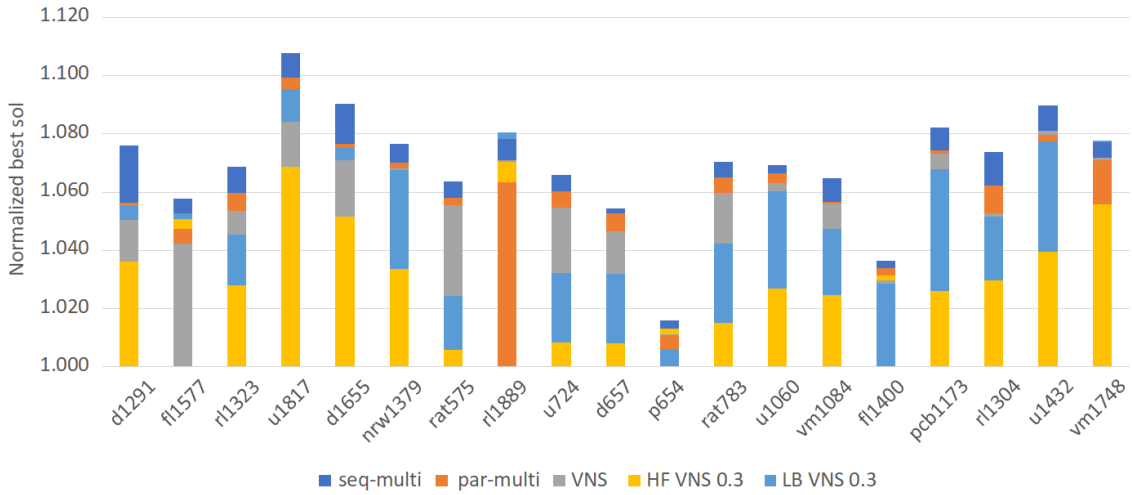


Figure 5.7: Detailed heuristic and matheuristic comparison.

All the previous considerations vary on the basis of the selected testbed of instances, and, while we tried to use the most standard and representative ones, as we can see from Figure 5.7 there are some pathological examples with radically different results. Furthermore, in a real application, a precise analysis of the typology of the target instances should be done.

Chapter 6

Conclusions

In this report we explored various approaches to solve the Traveling Salesman Problem (TSP), discussing both the exact and approximate techniques. We tested our implementation over the TSPLIB library, using the department cluster to realize a large amount of executions. Considering that the TSP is a NP-hard problem and the limited hardware at our disposal, using the latest Cplex version and the discussed exact approaches we were able to solve in a reasonable amount of time problems containing up to 500 nodes.
In general, even though the heuristic and matheuristic algorithms can not certify the quality of the solution found, in practice they can get very close to the optimal one, with a gap between 5 and 10 percent. In many cases, especially in problems with a large number of nodes, such suboptimal solution could be acceptable, since the optimal solution might take too much time to be calculated and the improvement obtained investing further computational effort could be not worth.
The realization of the algorithms described during the course was focused not only on the obvious correctness, but also on their efficiency, choosing proper data structures and explorations techniques. We dedicated particular attention to the creation and testing of the parallel multistart and simulated annealing heuristics.
All the code was progressively well documented and we tried to maintain a tidy and clear coding style to simplify the consultation of the interested readers. Our work could be expanded in the future introducing new techniques, obtaining a wider collection of concrete algorithm implementations to tackle the Traveling Salesman Problem.

Chapter 7

Tables

	loop_method		lazy_callback		usercut_callback	
	WALL	CPU	WALL	CPU	WALL	CPU
a280	34.517	110.028	12.344	87.098	19.862	116.969
ch150	9.065	40.623	4.281	40.756	6.351	46.085
eil51	0.539	1.586	0.163	0.235	0.170	0.216
kroA150	17.978	99.021	6.245	67.416	9.423	82.206
kroB200	18.112	102.339	15.549	207.106	69.661	196.687
lin105	2.594	10.032	0.856	1.294	1.159	1.317
pr124	13.948	77.800	2.299	17.011	2.608	16.318
pr226	1089.490	11672.800	18.177	201.246	27.350	279.166
rat195	32.120	236.437	20.224	285.660	43.844	503.951
st70	0.876	2.556	0.329	0.417	0.587	0.662
berlin52	0.107	0.254	0.025	0.029	0.032	0.036
d198	53.449	366.949	24.241	370.300	55.910	614.651
eil76	0.419	1.107	0.108	0.115	0.127	0.133
kroA200	40.362	324.841	89.291	1613.570	606.952	6038.190
kroC100	4.266	19.158	1.160	4.612	1.928	3.060
lin318	89.026	398.093	86.025	1238.850	337.057	3694.140
pr136	5.152	19.074	1.813	3.526	1.891	3.274
pr264	exceeded	exceeded	22.245	220.338	24.981	227.568
rat99	1.611	4.918	0.634	0.917	0.638	0.891
ts225	exceeded	exceeded	exceeded	exceeded	exceeded	exceeded
bier127	3.458	14.421	1.498	4.051	1.684	4.469
gil262	48.045	183.979	121.596	2097.820	314.699	3370.200
kroB100	6.073	28.605	1.155	8.475	1.406	8.059
kroD100	4.105	18.413	0.856	2.530	1.462	2.025
linhp318	84.297	390.565	91.027	1273.930	273.536	3082.740
pr144	20.772	118.286	7.050	81.804	8.688	95.730
pr299	157.931	1210.550	exceeded	exceeded	3453.030	51281.100
rd100	2.536	13.061	1.017	2.302	1.033	2.363
ch130	3.987	16.711	1.643	2.602	1.726	2.391
eil101	1.492	4.912	0.609	1.427	0.664	1.381
kroA100	3.560	13.923	1.126	3.229	1.251	2.790
kroB150	22.845	143.421	9.573	136.726	19.421	219.819
kroE100	3.437	14.799	1.881	17.667	3.422	8.857
pr107	0.996	2.614	0.038	0.050	0.044	0.057
pr152	10.874	42.799	5.238	48.780	5.921	49.964
pr76	5.233	34.896	7.803	118.813	2.503	23.978
rd400	204.243	1230.590	2913.900	46093.300	2525.670	27764.000

Table 7.1: Table of wall and cpu times for each exact method, measured in seconds. The label "exceeded" means that the corresponding run exceeded the 60 minutes time limit.

	seq-multi	par-multi	VNS	sim-ann
d1291	1.076	1.056	1.051	1.187
fl1577	1.058	1.048	1.042	1.164
rl1323	1.069	1.060	1.054	1.164
u1817	1.108	1.099	1.084	1.202
d1655	1.090	1.076	1.071	1.172
nrw1379	1.077	1.070	1.068	1.120
rat575	1.064	1.058	1.056	1.084
rl1889	1.078	1.063	1.071	1.206
u724	1.066	1.060	1.055	1.098
d657	1.054	1.053	1.047	1.084
p654	1.016	1.011	1.011	1.054
rat783	1.070	1.065	1.060	1.103
u1060	1.069	1.067	1.063	1.113
vm1084	1.065	1.057	1.056	1.123
fl1400	1.037	1.034	1.030	1.083
pcb1173	1.082	1.074	1.074	1.149
rl1304	1.074	1.062	1.053	1.175
u1432	1.090	1.080	1.081	1.108
vm1748	1.077	1.071	1.072	1.157
mean	1.069	1.061	1.058	1.134
std	0.00040	0.00033	0.00031	0.00199

Table 7.2: Table of normalized solution values for each heuristic method with mean and standard deviation.

	seq-multi		par-multi		VNS	
	HEUR	BEST	HEUR	BEST	HEUR	BEST
d1291	1.091	1.054	1.075	1.036	1.064	1.036
fl1577	1.065	1.064	1.054	1.053	1.053	1.051
rl1323	1.078	1.041	1.061	1.032	1.052	1.028
u1817	1.117	1.074	1.099	1.062	1.100	1.069
d1655	1.096	1.084	1.076	1.043	1.081	1.052
nrw1379	1.082	1.030	1.078	1.032	1.074	1.034
rat575	1.068	1.006	1.060	1.013	1.062	1.006
rl1889	1.088	1.076	1.078	1.062	1.083	1.070
u724	1.067	1.011	1.065	1.006	1.056	1.008
d657	1.055	1.011	1.053	1.011	1.053	1.008
p654	1.021	1.020	1.015	1.013	1.014	1.013
rat783	1.078	1.017	1.070	1.018	1.063	1.015
pr1002	1.077	1.030	1.073	1.027	1.070	1.029
u1060	1.074	1.028	1.069	1.029	1.071	1.027
vm1084	1.068	1.025	1.059	1.018	1.060	1.025
fl1400	1.040	1.039	1.038	1.035	1.033	1.031
pcb1173	1.088	1.031	1.079	1.031	1.079	1.026
rl1304	1.083	1.039	1.069	1.042	1.060	1.030
u1432	1.094	1.040	1.080	1.035	1.085	1.040
vm1748	1.086	1.060	1.076	1.052	1.081	1.056
mean	1.076	1.039	1.066	1.033	1.065	1.033
std	0.00043	0.00052	0.00031	0.00027	0.00037	0.00036

Table 7.3: Table of normalized solution values for Hard Fixing with time ratio of 0.3, with mean and standard deviation.

	seq-multi		par-multi		VNS	
	HEUR	BEST	HEUR	BEST	HEUR	BEST
d1291	1.091	1.082	1.075	1.068	1.064	1.055
fl1577	1.065	1.065	1.054	1.054	1.053	1.053
rl1323	1.078	1.068	1.061	1.052	1.052	1.045
u1817	1.117	1.113	1.099	1.095	1.100	1.095
d1655	1.096	1.091	1.076	1.075	1.081	1.075
nrw1379	1.082	1.074	1.078	1.072	1.074	1.068
rat575	1.068	1.032	1.060	1.026	1.062	1.024
rl1889	1.088	1.085	1.078	1.073	1.083	1.081
u724	1.067	1.042	1.065	1.037	1.056	1.032
d657	1.055	1.034	1.053	1.028	1.053	1.032
p654	1.021	1.011	1.015	1.008	1.014	1.006
rat783	1.078	1.057	1.070	1.039	1.063	1.042
pr1002	1.077	1.069	1.073	1.061	1.070	1.060
u1060	1.074	1.065	1.069	1.061	1.071	1.060
vm1084	1.068	1.053	1.059	1.052	1.060	1.047
fl1400	1.040	1.039	1.038	1.034	1.033	1.029
pcb1173	1.088	1.073	1.079	1.069	1.079	1.068
rl1304	1.083	1.073	1.069	1.065	1.060	1.052
u1432	1.094	1.088	1.080	1.073	1.085	1.077
vm1748	1.086	1.081	1.076	1.071	1.081	1.078
mean	1.076	1.065	1.066	1.056	1.065	1.054
std	0.00043	0.00058	0.00031	0.00045	0.00037	0.00050

Table 7.4: Table of normalized solution values for Local Branching with time ratio of 0.3, with mean and standard deviation.

List of Figures

List of Tables

Bibliography

[1] *Concorde TSP solver*. URL: http://www.math.uwaterloo.ca/tsp/concorde/index.html.

[2] W.J. Cook. *In Pursuit of The Traveling Salesman: mathematics at the limits of computation*. Ed. by Princeton University Press. Princeton University Press, 2012.

[3] E. D. Dolan and J.J. Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical Programming* 91.2 (2002), pp. 201–213.

[4] T.A. Feo and M.G.C. Resende. "Greedy Randomized Adaptive Search Procedures". In: *Journal of Global Optimization* 6.2 (1995), pp. 109–133.

[5] M. Fischetti. *Lezioni di Ricerca Operativa*. Ed. by Libreria Progetto. Libreria Progetto, 2014.

[6] M. Fischetti and A. Lodi. "Local branching". In: *Mathematical Programming* 98.1 (2003), pp. 23–47.

[7] P. Hansen, N. Mladenović, and J. A. MorenoPérez. "Variable neighbourhood search: methods and applications". In: *Annals of Operations Research* 175.1 (2010), pp. 367–407.

[8] *Library of instances for the TSP*. URL: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[9] *OpenMP*. URL: http://www.openmp.org.

[10] *Travelling salesman problem*. 2017. URL: https://en.wikipedia.org/wiki/Travelling%5C_salesman%5C_problem.