University of Padua

Travelling Salesman Problem

Course Report of
Operations Research 2

Professor
Matteo Fischetti

Author, ID number
Daniel Carlesso, 2088626
Luigi Frigione, 2060685

Academic Year 2023/2024

Contents

List of Figures

1 Introduction

1.1 Problem History

The Travelling Salesman Problem, or TSP for short, is one of the most intensively studied problems in computational mathematics. Its origin gets back to 1832 when a handbook for travelling salesmen, called "Der Handlungsreisende von einem alten Commis-Voyageur", mentions the problem and includes example tours through Germany and Switzerland, but contains no mathematical treatment. Then, still in the 1800s, the first mathematical problems related to TSP were introduced and studied by the Irish mathematician Sir William Rowan Hamilton and by the British mathematician Thomas Penyngton Kirkman.
In the 1950s and 1960s, the problem became increasingly popular in scientific circles in Europe and the United States after the RAND Corporation in Santa Monica offered prizes for steps in solving the problem. Notable contributions were made by George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson from the RAND Corporation, who expressed the problem as an integer linear program and developed the cutting plane method for its solution.
Richard M. Karp showed in 1972 that the Hamiltonian cycle problem was NP-complete, which implies the NP-hardness of TSP. This supplied a mathematical explanation for the apparent computational difficulty of finding optimal tours.
In 1976, Christofides and Serdyukov independently of each other made a big advance in this direction: the Christofides-Serdyukov algorithm yields a solution that, in the worst case, is at most 1.5 times longer than the optimal solution.

1.2 Problem Definition

Given a weighted graph G=(V,E) where V is the set of vertices and E the set of edges, the goal is to find the Hamiltonian cycle[1] between the vertices of the graph G that minimizes the sum of the weights of the edges in the cycle. In this paper we consider a specific formulation of the TSP, which is the Symmetric TSP, which differs for the generic one because the distance

[1] A Hamiltonian cycle is a cycle that visits each vertex exactly once.

between two vertices is the same in each opposite direction, forming an undirected graph. Another specification that we add within this paper is that the graph G is also complete, which means that every pair of distinct vertices is connected by a unique edge.
The mathematical formulation is the following:

$$\min \sum_{e \in E} c_e x_e \tag{1}$$

$$\sum_{e \in \delta(h)} x_e = 2, \quad \forall h \in V \tag{2}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \quad \forall S \subsetneq V : |S| \geq 3 \tag{3}$$

$$x_e \in \{0, 1\}, \quad \forall e \in E \tag{4}$$

where (2) constrains the degree of each vertex to be equal to 2 to create a cycle, and (3), called Subtour Elimination Constraint (SEC), constrains the solution to form a unique connected component.

1.3 The program

The project comprises a suite of algorithm implementations and a performance analysis tool.
These algorithms are designed to address the Symmetric Traveling Salesman Problem (TSP) in various contexts, as we'll clarify later. The performance analyzer serves a dual purpose: fine-tuning hyperparameters and providing statistical evidence regarding algorithm behavior relative to others.

Both programs utilize the [6] GetOpt library to parse command line arguments and a Finite-State Automaton to validate them.

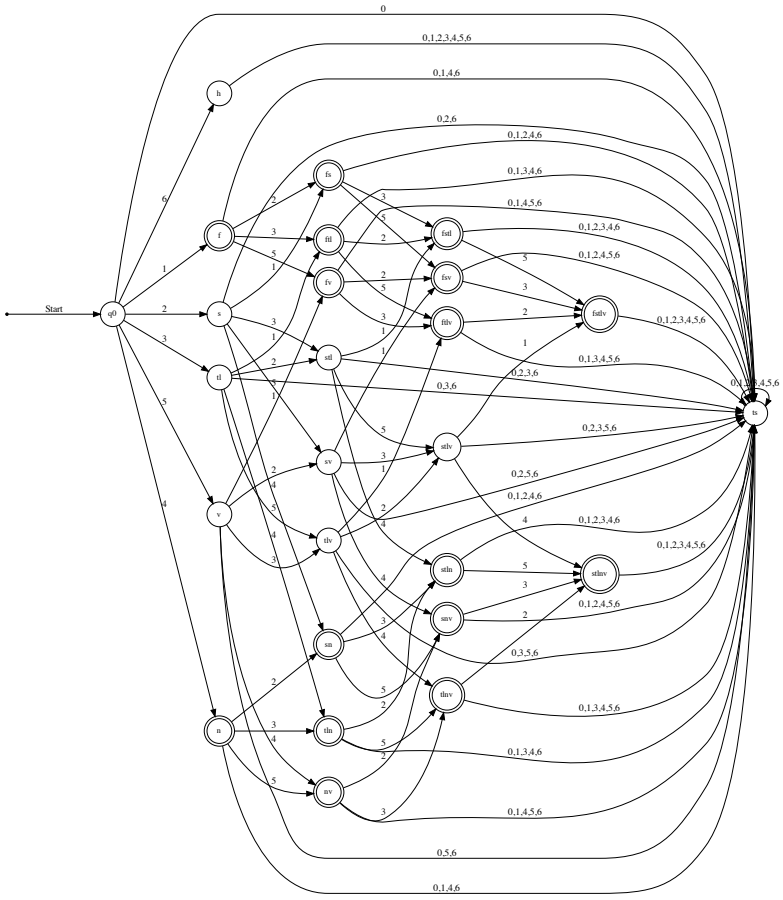

Figure 1: Input validator automaton.

1.3.1 Implementations

To run the program it is sufficient to execute the `./main --help` (or just `./main`) program.
It will display the help section with specific information.

An interactive terminal will then guide you through the choice of the preferred algorithm to run within specified parameters.

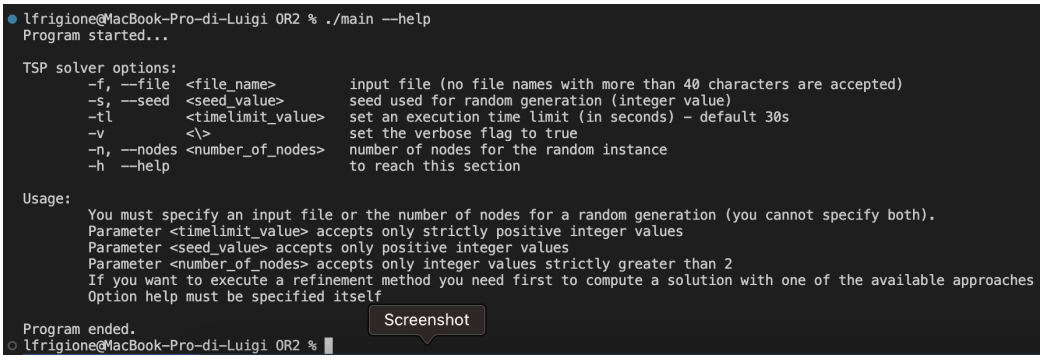

Figure 2: Main helper.

1.3.2 Performance analyzer

To run the analyzer instead, execute the `./pprof --help` (or just `./pprof`) program.
It will recall the main helper and specify the allowed arguments for the performance profiler.

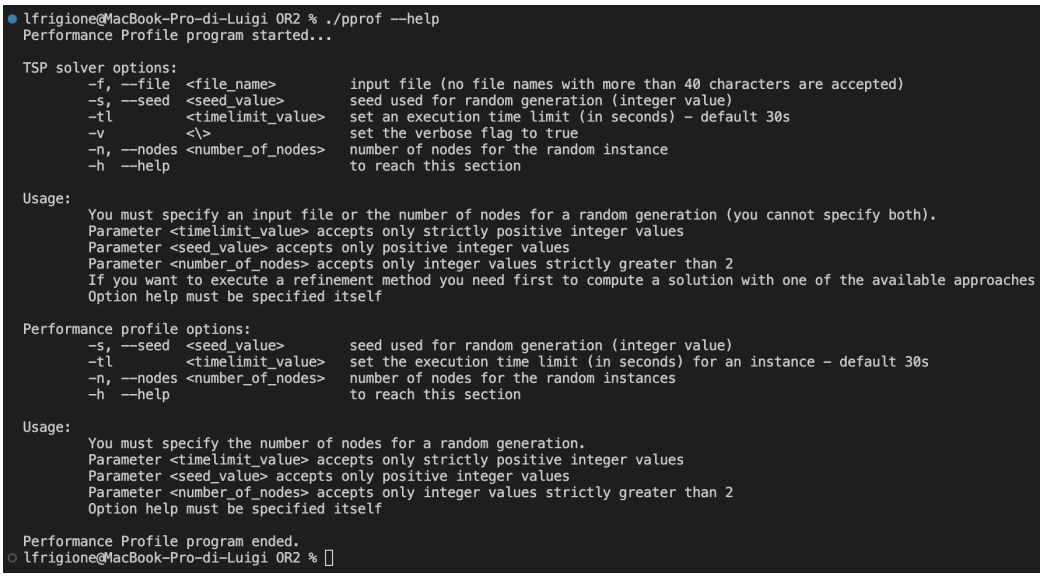

Figure 3: Performance profiler helper.

2 Heuristics

Exact methods are certainly preferable when we want to find the optimal solution without being bothered by execution time. However, this is not always the case, as for some instances, the running time can be very high. In these situations, we must resort to heuristic methods to find a solution that is close to optimal.
There are two types of heuristics:

- **Constructive heuristics**: they generate a solution from scratch starting from the problem instance.
- **Refinement heuristics**: they improve a previously generated solution.

2.1 Nearest Neighbor

The Nearest Neighbor heuristic is a greedy 2-approximation algorithm, which means that the returned solution's cost is at most twice the cost of the optimal one.
The idea is the following: starting from an arbitrary node, at each iteration, the algorithm selects the next vertex as the nearest vertex to the current one. It then visits that vertex, making it the current visiting node, and repeats these steps until all nodes have been visited. This algorithm has computational complexity of $\mathcal{O}(n^2)$, since for each vertex it checks the distance to every other unvisited vertex to find the minimum.
The algorithm is presented in Algorithm 1.
In our implementation we adopted 3 policies for selecting the starting node:

- *First Node*: Starts from the first node.
- *Random Node*: Starts from a node selected uniformly at random.
- *Best Start*: Applies the algorithm starting from each of the present nodes and returns the best solution found.

2.2 2-OPT

This is a refinement method, introduced in [3], which is based on removing crossings and other bad connections in the provided cycle.

Algorithm 1: Nearest Neighbor

Input: TSP instance, starting node v
Output: Hamiltonian cycle
n $\leftarrow |V|$; //number of vertices in the instance
visited $\leftarrow$ {v};
current = v;
while $|visited| \neq n$ **do**
    next $\leftarrow$ unvisited nearest neighbor of current;
    visited $\leftarrow$ visited $\cup$ {next};
    add edge (current, next) in the solution's cycle;
    add weight of (current, next) to solution's cost;
    current = next;
end
add edge (current, v) to close the solution's cycle;
add weight of (current, v) to solution's cost;
return solution;

The algorithm scans all pairs of edges and for each pair of edges it checks if their non-trivial swap improves the solution's cost or not.
More specifically, given a pair of edges defined by 4 distinct nodes, say $e_1 = (a, b)$ and $e_2 = (c, d)$, their non-trivial swap is defined as (a,c) and (b,d). The swap is performed once per iteration, and the choice of the swap is made by looking at the pair of edges that minimizes the delta metric, which is defined as follows:

$$\Delta(e_1, e_2) = dist(a, c) + dist(b, d) - dist(a, b) - dist(c, d)$$

where dist(a,b) is the function that computes the distance between a and b. Since the algorithm is scanning all pairs of edges to search for the one that minimizes the metric, it has computational complexity of $\mathcal{O}(n^2)$ for a single move.
The process is repeated until no improvement is possible.
The algorithm is presented in Algorithm 2.

2.3 Patching

The Patching, or Glueing, algorithm is an important heuristic that comes in help of exact methods. When running an exact method that adds SECs

Algorithm 2: 2-OPT

```
Input: TSP instance, TSP solution
Output: possibly improved solution
do
    minDelta ← 0;
    foreach pair of edges in the cycle do
        delta ← compute delta metric;
        if delta < minDelta then
            save the move;
            minDelta = delta;
        end
    end
    if minDelta < 0 then
        perform the move associated to minDelta;
        add minDelta to solution's cost;
    end
while minDelta < 0;
```

iteratively, it can happen that the method exceeds the time limit and the solution found is not valid, in particular the solution is not a single connected component, i.e. a single cycle. Here the patching algorithm comes to help. More specifically, the algorithm finds a pair of edges belonging to two different connected components s.t. they are the ones that minimize the delta metric. Defining E as the set of edges of the considered solution and $comp[e]$ as the connected component which edge e belongs to, the pair of edges is then described as follows:

$$(e_1^*, e_2^*) = argmin\{\Delta(e_1, e_2) : \forall e_1, e_2 \in E, e_1 \neq e_2, comp[e_1] \neq comp[e_2]\} \quad (5)$$

At this point the algorithm removes these edges and connects their endpoints s.t. the two initial components are now merged into one connected component, as showed in Figure 4. This procedure is repeated until the solution becomes one single cycle.
The algorithm is presented in Algorithm 3.

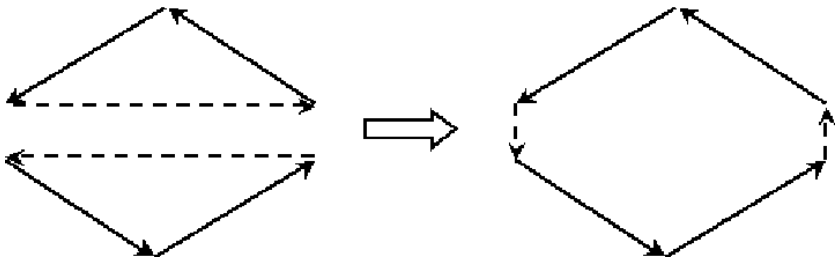

Figure 4: Patching two connected components.

Algorithm 3: Patching

Input: TSP instance, TSP invalid solution
Output: valid solution
fill E with edges in the input solution;
fill *comp* array with connected components each edge belongs to;
repeat
  $(e_1^*, e_2^*) = argmin\{\Delta(e_1, e_2) : \forall e_1, e_2 \in E, e_1 \neq e_2, comp[e_1] \neq comp[e_2]\}$;
  remove e_1^* and e_2^* from the solution;
  connect the two components to form a single one;
  update E and *comp* accordingly;
until *solution is a single cycle*;
return solution;

3 Metaheuristics

Metaheuristics, in their original definition, are solution methods that orchestrate an interaction between local improvement procedures and higher level strategies to create a process capable of escaping from local optima and performing a robust search on a solution space [8].
Over time, these methods improved their strategies for overcoming the trap of local optimality even in complex solution spaces e.g. neighborhood structures.
We present two algorithms of this family:

- **Tabu Search**: which may perform moves that get the solution worse by keeping an history of not allowed operations for some iterations.

- **Variable Neighborhood Search**: which, at each iteration, runs the 2-OPT algorithm to improve the solution and when it reaches a minimum it performs a variable number of worsening moves.

These methods do not generate a solution from scratch, they need a previously computed solution to perform their search to reduce the cost.

3.1 Tabu Search

Tabu Search was introduced by Fred Glover in 1986 [7], and its basic idea to escape local optima is to consider the moves history as a tabu (they will not be applied again). In this way, at a certain point, the method will be forced to diversify, even if this results in an increase in cost. This process allows us to avoid getting stuck in a loop or a local optimum, as represented in the figure 5.

More specifically, it applies the least worsening move that is not tabu. Tabu moves are freed after a certain amount of time, otherwise the algorithm would stop because no move is possible. For this purpose, over the years, a lot of different implementation on how to store tabu list have been introduced, e.g. tabu list with fixed length and FIFO policy, tabu list with variable length or even tabu list with time constraint.

In our implementation we use a tabu list with variable time constraint. More-

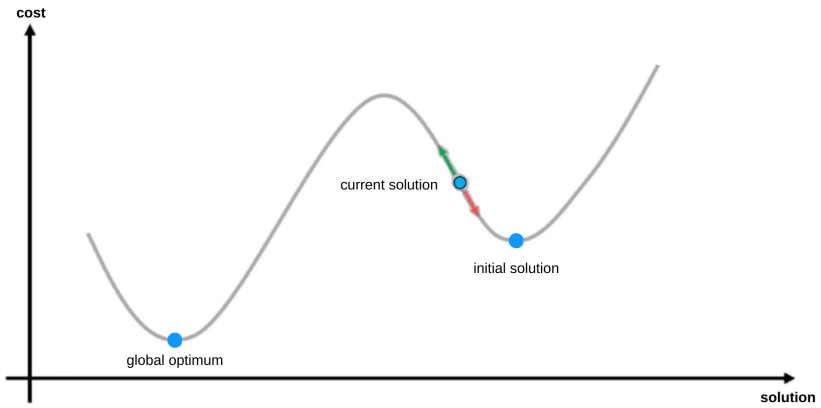

Figure 5: Tabu Search. If the red move is not set as tabu we could have an infinite loop between the initial solution and the current one without exploring other areas of the solution space.

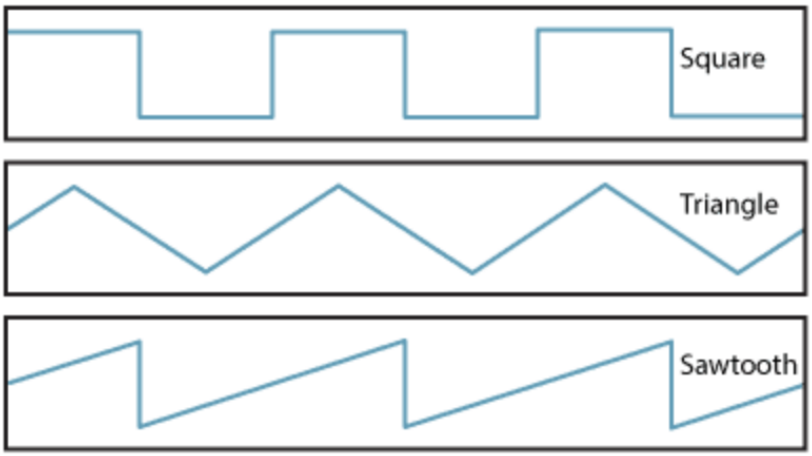

Figure 6: Different tenure functions implemented.

over, since for blocking a move it is sufficient to block one of the four nodes that characterizes it, we created a list, called *tabuList*, of dimension equal to the number of nodes of the problem instance s.t. at each index we store the iteration at which that node was set as tabu.
We have also a parameter called *tenure*[2] that controls the time for which a node is set as tabu. At iteration *currentIteration* a vertex v is tabu if it satisfies the following inequality:

$$currentIteration - tabuList[v-1] \leq tenure$$

In our case the *tabuList* is represented by an array of fixed length, what is varying is the *tenure* and we make it vary with three different functions: *Sawtooth*, *Triangle* and *Square*, represented in Figure 6. The *tenure*'s value starts from 10 and it is guaranteed to be always greater or equal to the starting value. In our implementation the search is performed only between 2-opt moves.
The Tabu search algorithm is presented in Algorithm 4.

[2]The tenure is a parameter that represents the tabu list size

Algorithm 4: Tabu Search

```
Input: TSP instance, TSP solution
Output: possibly improved solution
bestSol ← copy initial solution;
tempSol ← copy initial solution;
minCost ← initial solution's cost;
currentCost ← minCost;
repeat
    perform least worsening 2-opt not tabu move to tempSol;
    update currentCost;
    if currentCost < minCost then
        bestSol ← copy tempSol;
        minCost = currentCost;
    end
until time limit reached;
return bestSol;
```

3.2 Variable Neighborhood Search (VNS)

The idea behind this algorithm, introduced in [9], is to explore the solution space letting the neighborhood vary, as the name suggests. Moreover, in the original concept, when a local optimum is reached, the algorithm should search for a new optimum in the various neighborhoods of different sizes and update the incumbent solution if a better one is found. Over the years there have been different variations of this algorithm, some with deterministic neighborhood changes (e.g. Variable Neighborhood Descent), some with stochastic neighborhood change (e.g. Reduced VNS), others as a combination of both (e.g. Basic VNS).

In our implementation, at each iteration we perform the 2-OPT algorithm until it stops, i.e. we reached a local optimum, then, to explore other areas of the solution space, we apply a number k[3] of 3-opt *kick*s[4]. Therefore, 3-opt kicks perform a variable neighborhood search, and a different number of them is applied at each iteration.
The effectiveness of the above procedure can be investigated in Figure 7 and

[3]The value k is sampled uniformly at random between 1 and the constant L.

[4]The vertices that characterize the kick are selected uniformly at random. A k-opt kick allows to move in a k-neighborhood.

the algorithm is described in Algorithm 5.

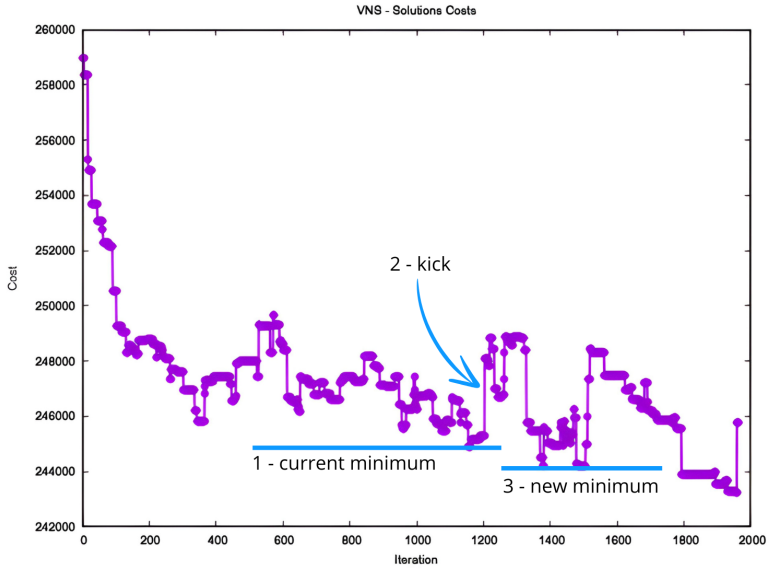

Figure 7: Costs of the solutions found by the VNS algorithm for each iteration.

Algorithm 5: Variable Neighborhood Search (VNS)

```
Input: TSP instance, TSP solution
Output: possibly improved solution
temp ← copy input solution;
repeat
    perform 2-OPT algorithm on temp;
    if temp better than solution then
        solution ← temp;
    end
    kick temp;
until time limit reached;
return solution;
```

4 Heuristic methods results

In this section we will compare the performances of all the combinations of heuristic and metaheuristic methods w.r.t. the solution cost obtained by

each method. The testbed is made of 20 instances of 2000 nodes each. The time limit is set to 120s and we use the default seed, which is 4149311, to generate the instances.
We first compare Nearest Neighbor starting policies in Figure 8, then we compare each policy when combined with other methods such as 2-OPT, Tabu Search and VNS.

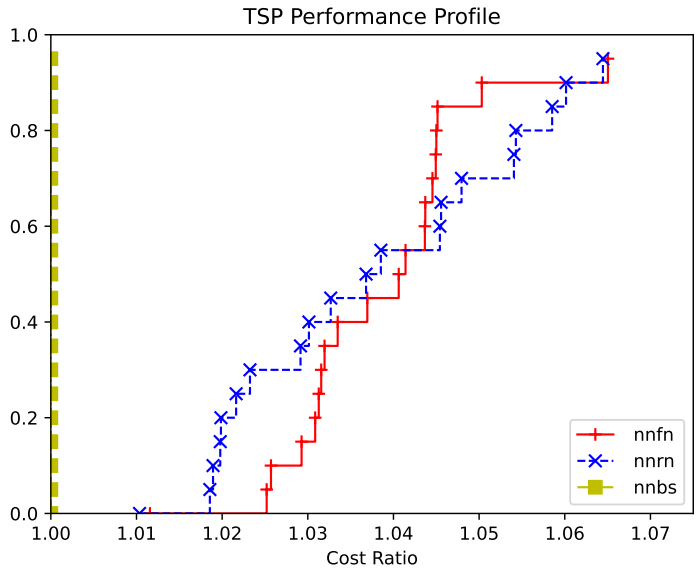

Figure 8: Performance profile of NN with various starting node policies: First Node (nnfn), Random Node (nnrn), Best Start (nnbs).

Even if it is quite obvious that nnbs will be the winner, it is still interesting to observe how a random policy behaves almost the same with respect to a very trivial one, nnfn.

In Figure 12 we adopted a random solution as starting point and, as can be seen, the Tabu Search algorithm is the clear winner of the comparison.

In Figure 13 there is not a landslide victory of a specific algorithm, but we can say that Nearest Neighbor + Tabu Search with Square tenure function (nnbs_square_tabu) is the best since it finds a better solution more times than the others. It is important also noticing, by looking back at Figure 12, that no matter how good an initial solution is, the Tabu Search algorithm still beats VNS and 2-OPT overall.

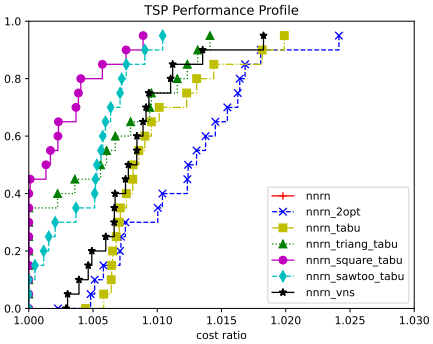

Figure 9: Performance profile of the Nearest Neighbor algorithm starting from a Random Node (nnrn).

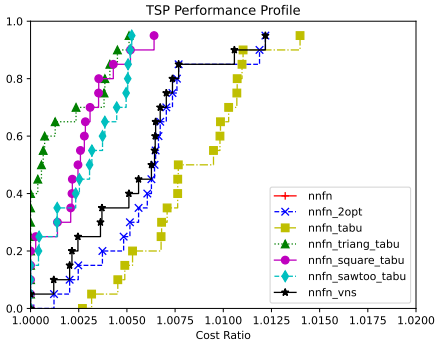

Figure 10: Performance profile of the Nearest Neighbor algorithm starting from the First Node (nnfn).

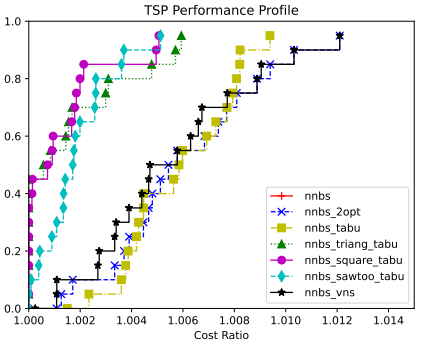

Figure 11: Performance profile of the Nearest Neighbor algorithm with the Best Start policy (nnbs).

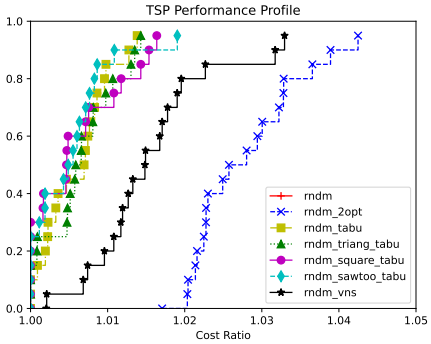

Figure 12: Performance profile of 2-OPT, Tabu Search and VNS methods starting from a random solution (rndm).

5 Exact Methods

5.1 Benders' Loop

Benders decomposition was originally proposed in 1962 to solve Mixed Integer Programs (MIP)[5].

[5]Programs to solve problems where some decision variables are discrete and others are not.

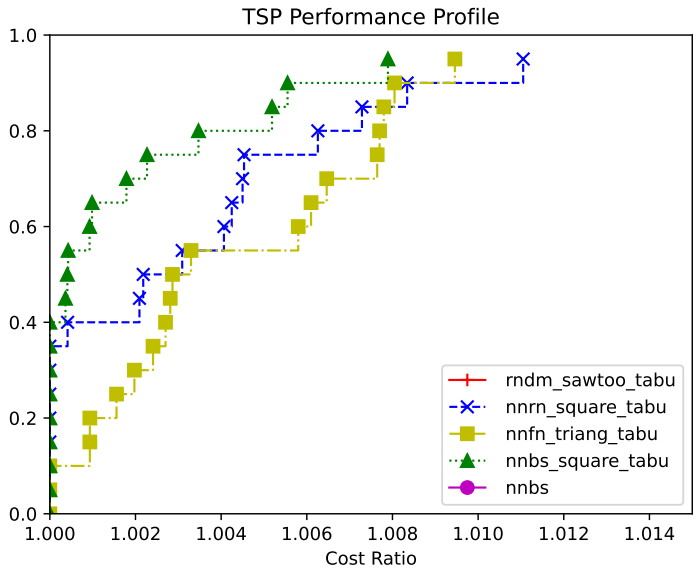

Figure 13: Performance profile of our top five heuristic methods.

As originally stated, Benders method is iterative. It is based on the idea that the majority of the possible subtours are made by bad edge combinations, corresponding to subtrees of the branching tree that the MIP solver would discard. At each iteration the CPLEX MIP solver solves the instance ignoring SECs. Then, if there are subtours in the solution, constraints to avoid those specific subtours are added. The method ends whether a valid solution is found or the time limit is reached. The pseudocode is represented in Algorithm 6.

Algorithm 6: Benders' Loop

```
Input: TSP instance
Output: Hamiltonian cycle
while true do
    solution ← CPLEX MIP solver;
    if solution has no subtours or time limit reached then
        break;
    end
    add SECs related to this solution;
end
return solution;
```

This approach strikes a balance between avoiding the initial inclusion of an exponential number of constraints and the need to recompute the solution at each iteration

5.2 Callbacks

Another way to add SECs is to exploit the CPLEX branch-and-cut technique. More specifically, CPLEX allows to dynamically interact with the optimization process through *callback procedures*, i.e. user-defined procedures that are executed in specific moments during the process.
In our case we implemented two types of callbacks to improve the solver performance:

- **Candidate Callback**: called whenever a new candidate solution, i.e. a new integer-feasible solution, is found.
- **User-Cut Callback**: called each time a new fractional solution is found.

5.2.1 Candidate Callback

The candidate callback is a *lazy-constraint callback*, which means that it is used to add constraints that are not used unless they are violated. Since SECs can be too many to be managed efficiently by the solver, i.e. the initial insertion of all the SECs would worse the performance, we add them step by step. In particular, each time that a new candidate solution is found by the solver, we analyze it and if it is made by more than one component we add the associated SECs, so that the invalid solution can be ignored by the next

iterations of the solver.
From this callback, it is possible to obtain a valid solution using the Patching algorithm (Section 2.3). This patched solution can be posted to CPLEX via the *CPXcallbackpostheursoln* method to improve the performances. CPLEX will put the solution in a pool and, when it will run its internal heuristics, if the posted solution is complete[6] and feasible and it has the best cost, it will use it as new incumbent.
In our implementation, every time that a candidate solution is invalid, we apply the Patching algorithm and then we post it.

5.2.2 User-Cut Callback

Another type of callback is the user-cut one, which is invoked each time CPLEX finds a candidate solution, performing what is specified in the Section 5.2.1, but also each time CPLEX finds a relaxed solution[7]. The relaxed solution is usually not integer feasible. It can, for example, be the solution to a node LP or can also come from another place.
In this phase CPLEX permits to apply a user-defined cut strategy for pruning the tree. In this scenario, unfortunately, the previously described methods cannot be used, since the solution we are working on is fractional. We used some functions of the Concorde [2] library: *CCcut_connect_components* for counting the connected components and *CCcut_violated_cuts* which computes the min-cut of a flow problem. Thanks to these methods, we were able to add cuts to the CPLEX model via a user-defined callback. In particular, *CCcut_violated_cuts* finds the cuts that violates the following constraint[8]:

$$\sum_{(i,j)\in\delta(S)} x_{ij} \geq 2, \quad \forall S \subsetneq V, S \neq \emptyset \tag{6}$$

Since the Concorde's functions are costly, we cannot use them on each node of the branching tree. We decided to use them only 10% of the time.

[6]The user can provide a solution that is missing some variables values and specify a strategy for completing the solution when posting it to CPLEX.

[7]The relaxed solution does not take into consideration the integer constraint on the variables values.

[8]$(i,j) \in \delta(S)$ indicates $i \in S$, $j \in V \setminus S$

5.3 Comparison between exact methods

Here we want to compare the performances of the exact methods w.r.t. the time spent by each method to find the optimal solution. The testbed is made of 20 instances of 500 nodes each, the time limit is set to 120s and we use the default seed, which is 4149311, to generate the instances.

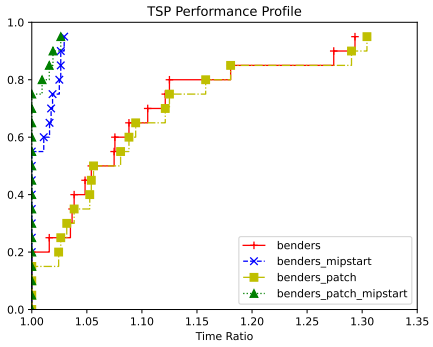

Figure 14: Performance profile of Benders' Loop algorithm variants.

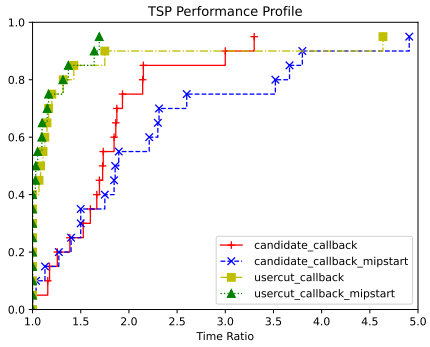

Figure 15: Performance profile of the Candidate and User-Cut callbacks variants.

In Figure 14, we can see that the best variant is benders_patch_mipstart, which has an heuristic solution as starting point and performs patching whenever the solution is not made of a unique connected component, this is to still provide a valid solution to the user if the time limit is exceeded.
In Figure 15, we compare the callbacks variants, i.e. with or without warm start (expressed as "mipstart" in the names).
In Figure 16, instead, we can see our best variant of Benders' Loop compared with the best variants of Candidate and User-Cut callbacks. Here the clear winner is the User-Cut callback (with a warm start), as it exploits the internal speed of CPLEX instead of restarting the solver each time as Benders' Loop does.

6 Matheuristics

Matheuristics is a combination of the heuristic approach and the mathematical programming approach. This term was firstly introduced in 2006 at the

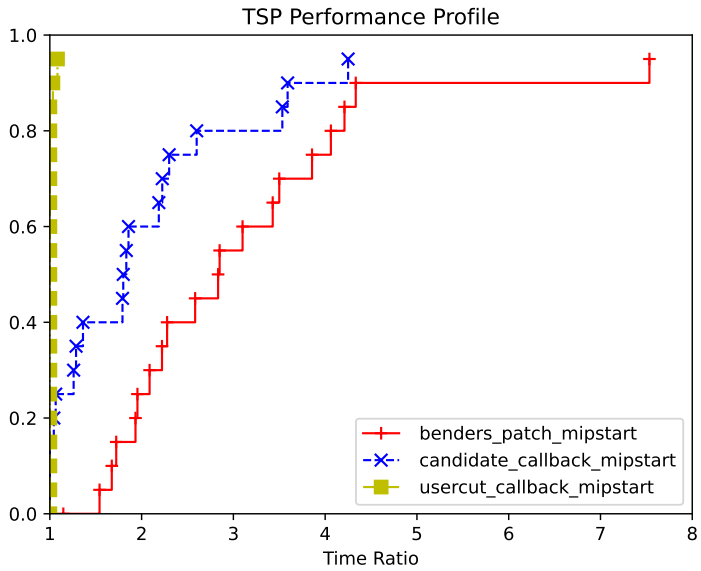

Figure 16: Performance profile of exact methods.

international workshop in Bertinoro, Italy, to discuss the use of mathematical tools for the design of heuristics. The entire branch of Matheuristics has been started by a work of Matteo Fischetti and Andrea Lodi, called Local Branching [5]. In the following sections we present two matheuristics:

- **Hard Fixing (or Diving)**
- **Local Branching (or Softfixing)**

6.1 Hard Fixing

The main idea of Hard (Variable) Fixing, as suggested by the name, is to fix some variables[9] to reduce the size of the problem, yielding to a much faster execution. More specifically, when given an exact MIP solver as a black box and an integer-feasible solution, an arbitrary percentage μ of the variables (i.e., edges) is fixed at the values of the solution before invoking the solver. The solver is given a portion σ of the total time limit. When it reaches

[9]Fixing a variable means that the solver cannot change that variable's value during the execution.

Algorithm 7: Hard Fixing

Input: TSP solution x^H
Output: Hamiltonian cycle
$x^* \leftarrow x^H$;
build CPLEX model;
while *time limit not reached* **do**
    fix a portion μ of variables in CPLEX based on the x^H values;
    $x^H \leftarrow$ MIP solver with with time limit σ;
    if x^H *better than* x^* **then**
        $x^* \leftarrow x^H$;
    free the variables in CPLEX;
end
return x^*;

its time limit, if the solution found is better than the initial one, the best solution x^* is updated with the one found by the solver.
The procedure is repeated until the overall time limit is reached.
In our implementation we fixed $\sigma = 10\%$ and evaluated multiple values for the hyperparameter μ, as can be seen in Section 6.1.1. We select the variables to fix uniformly at random with probability μ via reservoir sampling. The algorithm is described in Algorithm 7.

6.1.1 Hyperparameter tuning

In this method the hyperparameter is μ, the percentage of variables to fix, whilst $\sigma = 10\%$ is a constant.
We run the algorithm with four values of μ (60%, 70%, 80%, 90%) with a testbed of 20 instances of 1000 nodes each. The time limit is set to 120s and we use the default seed, which is 4149311, to generate the instances.

As we can see in Figure 17 the best value for μ is 80%, therefore we will use this value when comparing Hard Fixing with other Matheuristics in Section 6.3.

6.2 Local Branching

Local Branching is a technique that exploits a MIP solver as a black box to solve the problem. The idea is that, given an integer-feasible solution, the

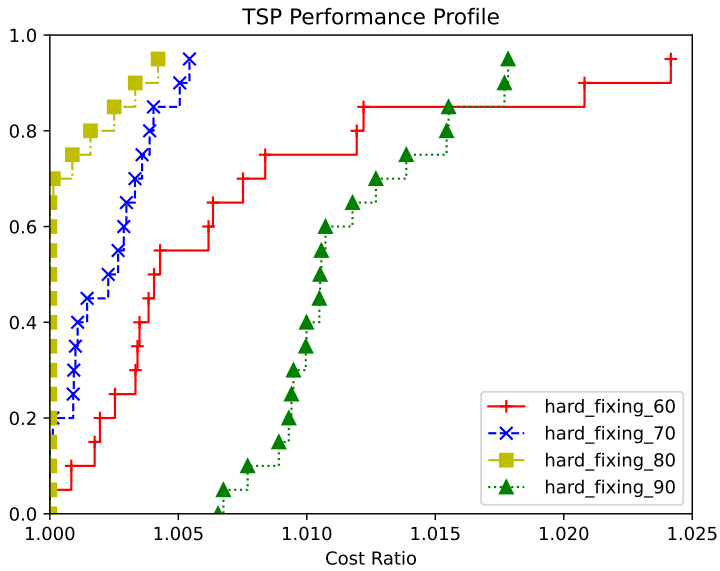

Figure 17: Performance profile of Hard Fixing with four values of μ.

solver will perform a local search on that. In particular, a new constraint to limit the search to a neighborhood is added to the problem before giving it to the MIP solver. This constraint is based on the Hamming distance between the given solution and the new possible solution, i.e. the number of binary variables that change value ($0 \to 1$ or $1 \to 0$) from the given solution to the one found by the solver, is limited by an hyperparameter k. The procedure is to start from an arbitrary value k and, if the search is not successful, then increment it to enlarge the search space.
The constraint can be expressed as follows:

$$\underbrace{\sum_{e:x_e^H=0} x_e}_{0\to1} + \underbrace{\sum_{e:x_e^H=1} (1 - x_e)}_{1\to0} \leq 2k \tag{7}$$

where e is a generic edge, x^H is the given solution to start from and x is the solution found by the solver.
Note that this constraint is dense as every variable has a nonzero coefficient. For TSP, the number of flipping variables from 0 to 1 is equal to the number of variables flipping from 1 to 0, then, the constraint can be adapted as

follows:

$$\sum_{e:x_e^H=1} (1 - x_e) \leq k \tag{8}$$

since the number of variables fixed to 1 in the TSP is known, say n, the final constraint becomes:

$$\sum_{e:x_e^H=1} x_e \geq n - k \tag{9}$$

In our implementation we evaluated different starting values for k, whose performances are shown in Section 6.2.1. The starting value is incremented by $\Delta = 5$ when needed, as anticipated.
The procedure is described in Algorithm 8.

Algorithm 8: Local Branching

```
Input: k, Δ, TSP solution x^H
Output: Hamiltonian cycle
x* ← x^H;
build CPLEX model;
while time limit not reached do
    add (9) to CPLEX based on x^H;
    x^H ← MIP solver;
    if x^H better than x* then
        x* ← x^H;
    else
        k ← k + Δ;
    end
end
return x*;
```

6.2.1 Hyperparameter tuning

In this method the hyperparameter is k, whilst $\Delta = 5$ is a constant.
We ran the algorithm with four values of k (20, 30, 40, 50), with a testbed of 20 instances of 1000 nodes each. The time limit is set to 120s and we use the default seed, which is 4149311, to generate the instances.

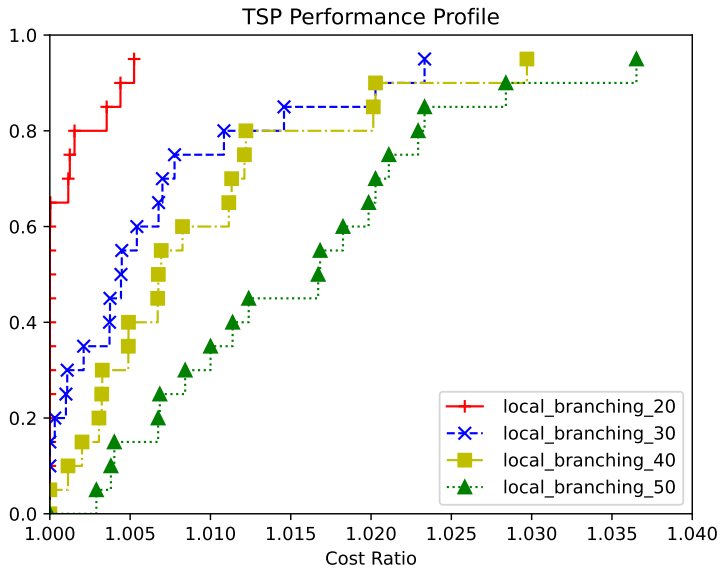

Figure 18: Performance profile of Local Branching with four values of k.

As we can see in Figure 18 the best value for k is 20, therefore we will use this value when comparing Local Branching with Hard Fixing in the next section.

6.3 Comparison between Matheuristics

Here we want to compare the performances of matheuristic methods w.r.t. the solution cost found by each method. The testbed is made of 20 instances of 1000 nodes each, the time limit is set to 120s and we use the default seed, which is 4149311, to generate the instances.

In this comparison we expected Local Branching to win against Hard Fixing since we made CPLEX decide the variables to fix rather than a simple random policy. This was not the case as can be seen in Figure 19: the performances are comparable. A smarter approach may not yield better results, as it could potentially slow down performance.

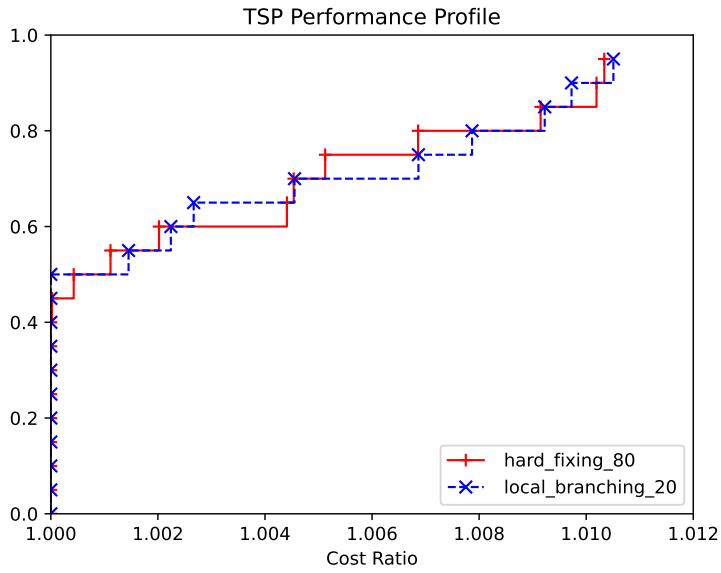

Figure 19: Performance profile of Hard Fixing vs Local Branching.

7 Conclusion

As shown in Section 4, the best heuristic method is nnbs_square_tabu. It is the combination of Nearest Neighbor with Best Start policy and Tabu Search with Square tenure function. This suggests that Tabu Search could be used as a go-to method. Our tests have shown that its application leads to the best results, independently of the quality of the starting solution (among heuristics).
Among exact methods, there is a pretty clear winner which is the User-Cut callback method. The power of this method relies in the CPLEX callback technology which allows to call external (user-defined) functions inside CPLEX processes. This yields to a much faster execution than the Benders' Loop method, which needs to restart the CPLEX MIP solver at each iteration.
With respect to matheuristic methods, instead, there is not a clear winner. The performances of Local Branching with $k = 20$ and Hard Fixing with $\mu = 80\%$ are more or less the same.

In figure 20, we can observe a small difference in solution cost between

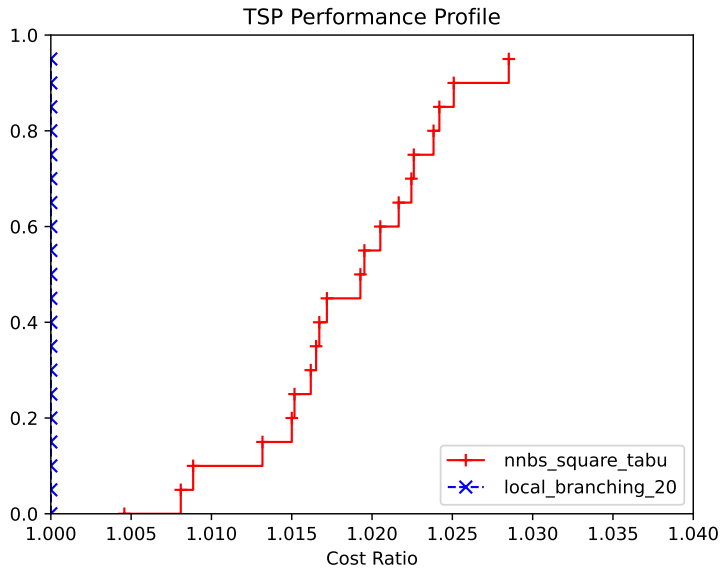

Figure 20: Performance profile of our best matheuristic method vs our best heuristic method.

our best heuristic method (nnbs_square_tabu) and our best Local Branching variant, with a testbed of 20 instances of 1000 nodes each, time limit 120s and seed 4149311.
In conclusion, depending on the main objective, whether it's execution time or accuracy, one method may be preferred over the others. Matheuristic methods can still give a little improvement with respect to heuristic methods for reasonable instance dimensions. In certain situations, where even a 1% savings can be significant, this can make all the difference.

References

[1] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962/63.

[2] Concorde. *https://www.math.uwaterloo.ca/tsp/concorde/index.html*.

[3] G. A. Croes. A Method for Solving Traveling-Salesman Problems. *Operations Research*, 6(6):791–812, December 1958.

[4] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[5] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 09 2003.

[6] GetOpt. *https://www.gnu.org/software/libc/manual/html_node/Getopt.html*.

[7] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986. Applications of Integer Programming.

[8] F. Glover and G. Kochenberger. *Handbook of metaheuristics*. Springer New York, NY, April 11, 2006.

[9] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.