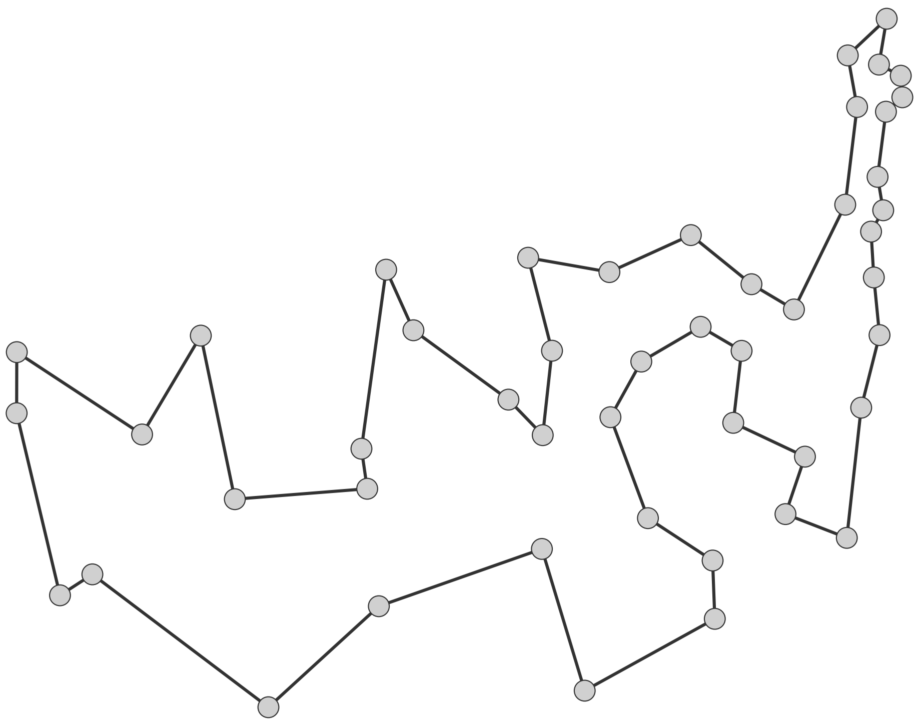
Performance Analysis of TSP Solving Techniques

Basaglia Alberto and Stocco Andrea



Abstract

The Traveling Salesman Problem (TSP) is definitely one of the most intensively studied problems in optimization. It asks the following question: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?* TSP is computationally difficult (NP-Hard), but there exist many *heuristics* and *exact methods* to solve it. In this report we are going to show and compare the performance of these techniques.

Contents

1 Introduction **3**

2 Heuristics **3**
2.1 Greedy . . . 3
2.2 2-opt . . . 4

3 Metaheuristics **6**
3.1 Tabu Search . . . 6
3.2 VNS (Variable Neighborhood Search) . . . 9

4 Exact Methods **11**
4.1 Benders' loops . . . 11
4.1.1 Patching . . . 12
4.2 Opening CPLEX . . . 13
4.2.1 Warm start . . . 15
4.2.2 Solution posting . . . 15
4.2.3 Fractional cuts . . . 15

5 Matheuristics **16**
5.1 Diving . . . 16
5.2 Local Branching . . . 17

6 Results **18**
6.1 Experimental Setup . . . 18
6.2 Performance profile . . . 18
6.3 Hyperparameters tuning . . . 19
6.3.1 Greedy . . . 19
6.3.2 Tabu search . . . 21
6.3.3 VNS . . . 23
6.3.4 Fast-heuristic . . . 24
6.3.5 Diving . . . 25
6.3.6 Local branching . . . 26
6.4 Best heuristic . . . 27
6.5 Best exact method . . . 28
6.6 Best matheuristic . . . 29

7 Conclusions **30**

1 Introduction

The Traveling Salesman Problem is for sure one of the most intensively studied problems in optimization. The TSP asks the question of what is the shortest path that visits exactly once all the nodes in a graph. We can also define it as finding the shortest Hamiltonian path of a graph.

The problem was formulated mathematically by the Irish mathematician William Rowan Hamilton and by the British mathematician Thomas Kirkman in the 19th century [4].

The state-of-the-art for solving the TSP problem nowadays is definitely the Concorde software. Concorde was created by D. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. It employs advanced combinatorial optimization techniques, linear programming, and cutting-plane methods [3][14].

The objective of this report is not to invent some new techniques that can compete with Concorde, but instead to cover basic approaches, that can be implemented without much effort, and still lead to good results.

2 Heuristics

Heuristics are algorithms to determine a near-optimal solution to an optimization problem. In certain scenarios, exact methods aren't able to provide the optimal solution in a reasonable amount of time or they can't find it at all. In contrast, a heuristic is generally capable of offering a solution that is more or less close to the optimal one. While there is no assurance regarding the optimality of the provided solution, it may still be considered acceptable in some instances.

2.1 Greedy

A greedy algorithm is a heuristic that attempts to find an optimal solution by selecting the locally best possible choice at each iteration. For instance, in our case, when determining the next node to visit, the greedy algorithm will always choose the nearest unvisited one. Algorithm 1 shows the pseudocode of this procedure

Algorithm 1 Greedy

```
Output: $solution$
procedure GREEDY($startingnode, nnodes$)
    $solution \leftarrow 0, 1, \ldots, nnodes - 1$
    $solution \leftarrow$ SWAP($solution, 0, startingnode$)
    for $i \leftarrow 0$ to $nnodes - 1$ do
        $mindist \leftarrow \infty$
        $minindex \leftarrow -1$
        for $j \leftarrow i + 1$ to $nnodes$ do
            if $dist(i, j) < mindist$ then
                $mindist \leftarrow dist(i, j)$
                $minindex \leftarrow j$
            end if
        end for
        $solution \leftarrow$ SWAP($solution, i + 1, minindex$)
    end for
end procedure
```

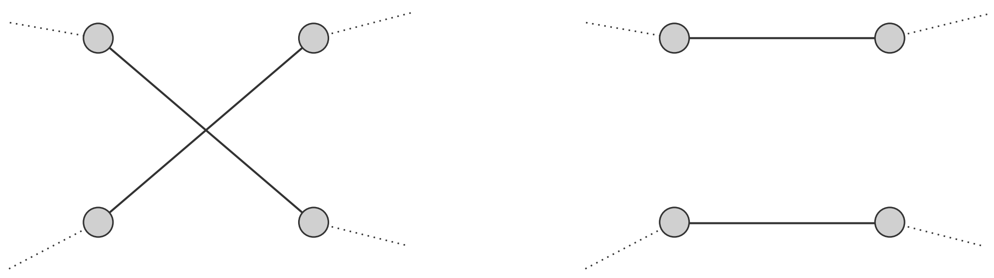
If we run this approach on an instance having the cities on a 2d plane and using the Euclidean distance as the cost of the edges, we will see, with a very high probability, an high number of edges crossing. It is possible to prove that an optimal solution, in this context, should not contain any crossing between edges. This fact can give us the intuition that a very simple yet clever technique could be used to drastically improve our solution. This technique, known as *2-opt*, will be covered in the next section.

Another important point to discuss is the node the algorithm is started on. One of the strategies we could think of would be to generate one randomly. Although feasible, in our experiment we decided to run the greedy procedure on all the possible starting nodes and then keep the best.

2.2 2-opt

The 2-opt algorithm is an optimization technique that can be used to improve a sub-optimal solution. It evaluates the possibility of removing two edges in the tour and reconnecting the now disconnected nodes in the other way. This process aims to improve the cost of the tour by eliminating inefficient segments while preserving the overall tour structure. By iteratively exploring these adjustments and accepting them if they result in a shorter tour cost, the algorithm gradually refines the solution. This iterative refinement continues until no further improvements can be made, resulting in a solution closer to the optimal one for the given problem instance. Although it is not the only possible situation where a 2-opt move is beneficial, figure 1 illustrates an example of such a move.

Figure 1: Example of a 2-opt move



(a) Before the move (b) After the move

It's easy to see that the removal of the crossing edges is beneficial to the cost of the solution.

Algorithm 3 shows a possible implementation of this procedure to improve the solution found with the greedy approach. In this case, as discussed in the previous section, the greedy procedure is executed for every possible node. Then, for each of the solutions, the *2-opt* procedure is applied. Another approach, that would use *2-opt* only once instead of n times, would be to compute the greedy procedure on all the starting nodes first, and then *2-opt* only on the best one. This approach won't be covered in this report.

Algorithm 2 Best 2-opt Swap

```
Output: $bestsolution$
procedure BEST2OPTSWAP($solution, nnodes$)
    $bestsolution \leftarrow solution$
    for $i \leftarrow 0$ to $nnodes - 2$ do
        for $j \leftarrow i + 2$ to $nnodes$ do
            $newsolution \leftarrow$ REVERSESUBSEQUENCE($solution, i + 1, j$)
            if COST($newsolution$) $<$ COST($bestsolution$) then
                $bestsolution \leftarrow newsolution$
            end if
        end for
    end for
end procedure
```

Algorithm 3 Greedy + 2-opt

```
Output: $solution$
procedure GREEDY2OPT($nnodes$)
    $solution \leftarrow \emptyset$
    $solutioncost \leftarrow \infty$
    for all $node \in tsp$ do
        $currentsolution \leftarrow$ GREEDY($node, nnodes$)
        while $true$ do
            $2optsolution \leftarrow$ BEST2OPTSWAP($currentsolution, nnodes$)
            if COST($2optsolution$) $<$ COST($currentsolution$) then
                $currentsolution \leftarrow 2optsolution$
            else
                $break$
            end if
        end while
        if COST($currentsolution$) $< solutioncost$ then
            $solution \leftarrow currentsolution$
            $solutioncost \leftarrow$ COST($currentsolution$)
        end if
    end for
end procedure
```

3 Metaheuristics

A metaheuristic is a versatile problem-solving approach characterized by its iterative nature and adaptability across various optimization problems. Unlike specific algorithms tailored to particular problems, metaheuristics serve as overarching strategies that guide subordinate heuristics to efficiently explore and exploit solution spaces. They intelligently combine different concepts to navigate through search spaces, aiming to find near-optimal solutions effectively. In our experiments, we will adopt metaheuristic using *2-opt* as the underlying heuristic. We will prove that simple but very clever ideas (like Tabu Search and VNS) combined with the previously seen heuristics allow us to get solutions very close to the optimal ones.

3.1 Tabu Search

Tabu Search is a metaheuristic that efficiently explores the solution space by intelligently navigating through a neighborhood of solutions while maintaining a short-term memory to avoid revisiting previously visited or less promising solutions. The algorithm is particularly effective for combinatorial optimization problems like the Traveling Salesman Problem (TSP).

This technique was introduced by Fred W. Glover in 1986 and then formalized in 1989 [9].

By design, this procedure will initially head directly towards a local minimum, thanks to the underlying heuristic. The only purpose of this metaheuristic is then to "escape" it and, hopefully, converge to a better minimum.

The search procedure will be alternating between 2 different behaviors by its nature. First of all we will identify a phase where the method, starting from a "bad" solution, uses the underlying heuristic to improve its cost. We will call this "intensification phase". Once a local minimum is reached, we will apply some bad moves to escape from this solution. We name this the "diversification phase". We hope that the alternation between improving the solution and moving away from it allows us to explore different local minimum, allowing us to get an overall better solution.

Algorithm 4 shows a very abstract description of the procedure we use in practice. The first solution is found using a greedy method (the closest neighbor approach) and then *2-opt* is used for the intensification phase. It is important to notice that *delta* is how much the solution cost would improve (decrease by).

Algorithm 4 Tabu Search

```
procedure TABUSEARCH($solution$)
    GREEDY(solution)
    $tabulist \leftarrow \emptyset$
    while !$stop$ do
        $move \leftarrow$ FINDBESTSWAPNOTABU($solution, tabulist$)
        $delta \leftarrow$ DELTA($move$)
        $solution \leftarrow$ APPLY($solution, move$)
        if $delta \leq 0$ then
            $tabulist \leftarrow tabulist \cup \{move\}$
        end if
        REMOVEOLD($tabulist$)
    end while
end procedure
```

The procedure begins by solving the problem with a greedy approach and with an empty tabu list.

At every iteration of the loop, we find the best move that is not in the tabu list. If this move has a positive delta (meaning that its application would lead to an improvement in the solution) we apply it and jump to the next iteration. If the best move has a negative delta we apply it anyway, adding it to the list of the tabu moves.

In our experiments we leave the loop after a predefined time-limit. Other possibilities can involve a maximum number of iterations, a lack of any improvement in the last iterations or any other kind of stopping criteria.

It is then important to define a way to remove moves from the list. In the pseudo-code we define it with a generic "removeold" function, in practice there are many ways the remove tabu entries.

We define "tenure" of the tabu method the window of iterations in which tabu moves are considered. For example a tenure of 100 iterations would mean that we remove from the tabu list all moves that weren't added in the last 100 iterations.

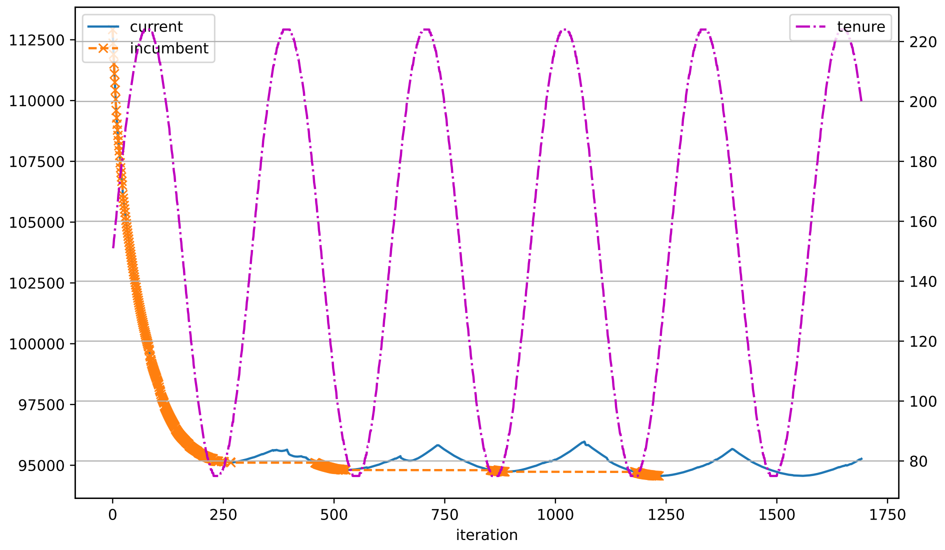
This *hyperparameter* is crucial for the effectiveness of the method. In our implementation we experimented with a fixed tenure (dependent on the number of nodes of the instance) and with some functions of the number of the current iteration. Some good results were obtained with a sinusoidal function.

This approach using variable values for the tenure can be quite effective compared to a fixed one as we will see in the chapter dedicated to experimental results. An intuition behind this can be given by the fact that varying the tenure can be beneficial to the search of a new minimum because it tries to solve both the problems of having a tenure that is too big and one that is too small. A small tenure can be problematic because we cannot escape the local minimum enough to not get back to it. A big tenure has the disadvantage that we might find ourselves in a situation where the move needed to reach a new local minimum is blocked by the tabu list. Having a variable tenure can, sometimes, on the long run, avoid the drawbacks of a fixed approach.

Figure 2 shows an example of the solving procedure using a sinusoidal tenure. The blue line shows the cost of the current solution. The orange line represents the incumbent and a cross is drawn when a new incumbent is found. The purple line is the value of the tenure.

At the beginning of the solving procedure it is possible to see that the solution cost improves very quickly. That is due to the 2-opt procedure being applied until a local minimum is found.

Figure 2: Example of variable tenure



3.2 VNS (Variable Neighborhood Search)

In *Tabu Search* we have seen two different phases:

- the *Intensification Phase* to move towards a better solution
- the *Diversification Phase* in which we try to escape from a local minimum using only non-tabu moves

With this approach, we may spend a considerable amount of time without achieving any improvement in solution quality (Diversification Phase). One potential solution could involve restarting from a random point each time we encounter a local minimum (Multistart). However, this approach would significantly prolong the Intensification Phase. VNS is a metaheuristic that employs minimal permutations of the solution to evade local minima, as opposed to restarting from a completely different starting point. This technique was proposed by Mladenović and Hansen in 1997 [12]. With this approach, we transition from one solution to a better one using the 2-opt method until reaching a local minimum. Subsequently, we perform a random number of permutations (called "kicks" in the following sections) involving three nodes (3-opt) to navigate away from the minimum. This is a simplified version of the algorithm; state-of-the-art techniques also allow for larger permutations of nodes (e.g., 4-opt, 5-opt, etc.) instead of multiple 3-opt operations. Algorithm 5 shows a very abstract description of the procedure

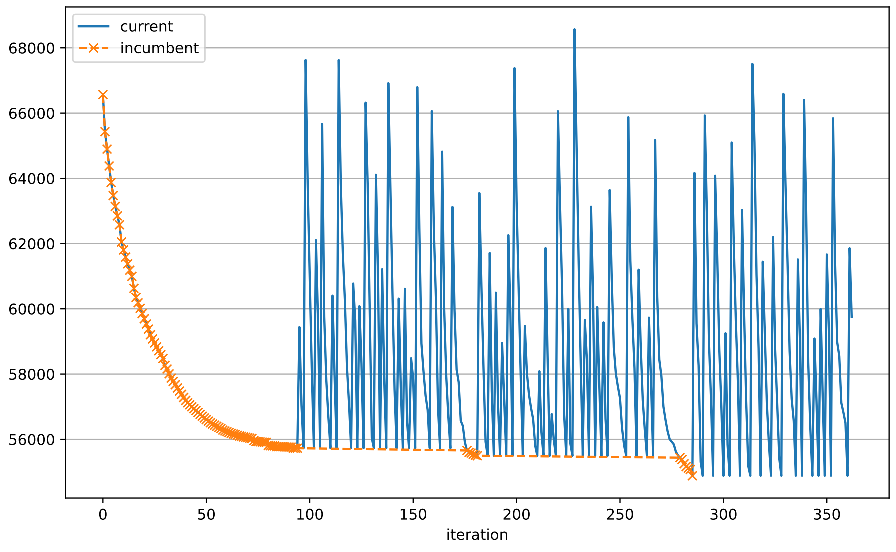
Algorithm 5 VNS

```
procedure VNS(solution)
    GREEDY(solution)
    while !stop do
        move ← FINDBEST2OPTSWAP(solution)
        delta ← DELTA(move)
        if delta ≤ 0 then
            for RANDOM(range) do
                move ← FIND3OPTSWAP(solution)
                solution ← APPLY(solution, move)
            end for
        else
            solution ← APPLY(solution, move)
        end if
    end while
end procedure
```

The number of kicks is chosen randomly within a range, making the minimum and the maximum number of kicks the only hyperparameters of this algorithm. Tuning these hyperparameters can lead to better performance of the algorithm. Performing too many kicks might result in restarting the intensification phase with a completely different solution, prolonging the time spent during this phase, while performing too few might not sufficiently change the solution, preventing us from visiting several different local minima. The tuning of these hyperparameters will be discussed in Section 6.

Figure 3 shows an example of the solving procedure using VNS. In this plot it is possible to see that, after the solver gets stuck in a local minimum, it can escape it after a series of kicks.

Figure 3: Example of VNS



4 Exact Methods

Although, as we will see, a good heuristic can achieve very good results, it is also important to consider approaches that lead to an optimal solution of the TSP.

All of the techniques that we will cover are based on integer linear programming. It is hence useful to provide an integer linear formulation for the problem. Many formulations exist, but the one we will focus on is the one proposed by Dantzig, Fulkerson and Johnson [6].

Initially we define an undirected complete graph $G = (V, E)$. We will the use $c_e : E \to \mathbb{R}^+$ to indicate the cost of an edge. One of the ways to compute this cost function is to place all the points on a 2d surface and compute their Euclidean distance.

The variables of this model are

$$x_e = \begin{cases} 1 \text{ the path uses edge } e \\ 0 \text{ otherwise} \end{cases}$$

The model, that makes use of the subtour elimination constraints, is the following

$$\begin{aligned} \min \sum_{e \in E} & c_e x_e \\ \text{s.t.} \sum_{e \in \delta(h)} & x_e = 2 \quad \forall h \in V \\ \sum_{e \in E(S)} & x_e \leq |S| - 1 \quad \forall S \subset V : |S| \geq 3 \\ & x_e \in \{0, 1\} \quad \forall e \in E. \end{aligned} \tag{1}$$

$\delta(h)$, where h is a node, is defined as the set of edges that are incident to h. $E(S)$, where S is a set of nodes, is the set of all the edges with both extremities in S.

It is important to know that the number of SECs is exponential in the number of the nodes. Hence, it is infeasible to generate all of them from the beginning with the size of the instances we are interested in.

The techniques that will be shown in this section are different ways to generate the subtour elimination constraints. We will use CPLEX to solve the MIP problems.

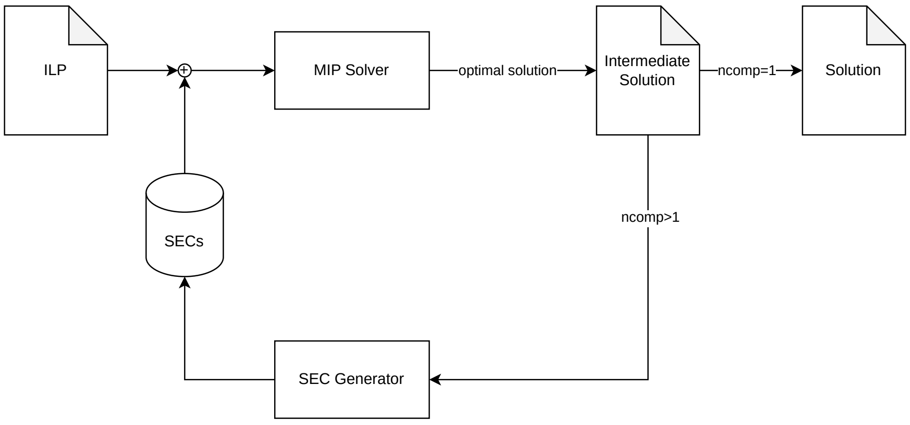
CPLEX is a high-performance mathematical programming solver for linear programming (LP), mixed-integer programming (MIP), and quadratic programming (QP). When solving the TSP using CPLEX, we leverage its robust MIP solving capabilities to handle the solution of the problem [10].

4.1 Benders' loops

The first technique we will explore is the so-called "Benders' loops" technique. The basic idea behind this approach is the iterated use of a MIP solver (like CPLEX) as a black-box. More precisely, we will start by providing the solver with a model lacking all the separation constraints. The solution of this model, with all probability, is going to contain more than one loop. We will then generate some constraints "cutting" the solution we have found. The model, updated with the new constraints, is then passed to the solver and solved again. This loop will be repeated until a solution composed by only one component is found. The process is described in Figure 4.

It is very important to note that the "optimal solution" output of the MIP solver is not the optimal solution of the TSP problem (exception made for the last iteration of the loop). That solution is optimal only for the current set of subtour elimination constraints.

Figure 4: Solving using Benders' loops

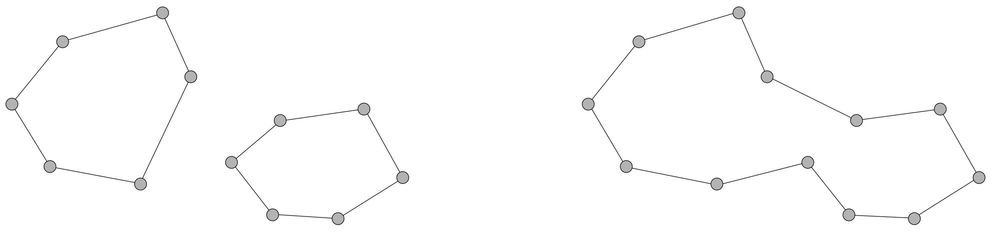


As we will see in Section 6, although this method allows us to solve some instances it has a major flaw. By using the MIP solver as a black box, we have to restart all the process every time a new solution (that contains subtours) is found. This creates one big problem: the branch-and-bound tree will be discarded and the search for the optimal solution will start again. Even though throwing away the search tree can be sometimes beneficial [2][1], it needs to be done following some precise criteria and at the correct time.

4.1.1 Patching

At the end of the Benders' loop we are guaranteed to obtain a solution to the TSP problem. However, if we finish the procedure earlier because of the time limit, we'll return a solution containing more than one loop. To solve this problem we implemented the so-called "Patching Heuristic". The idea behind this algorithm is to find a way to merge together all the components returned by the MIP solver so that, at each iteration of the Benders' loop, we have a solution composed of only one component. In this way, if we exit the Benders' loop because of the time limit, we'll always be able to return a correct solution to the problem. Figure 5 shows an example of patching.

Figure 5: Example of patching



(a) MIP solver solution (b) Patched solution

The Patching Heuristic aims to find the optimal way to merge two different components. As shown in Figure 5, the patched solution is the one that minimizes the cost of the resulting single component.

The solution to the TSP is an *undirected graph* because we are considering instances of the *symmetric TSP* problem. However, in our implementation, such solution is stored as a permutation of the nodes and so, implicitly, we have a *directed graph* as solution to the problem. If we have two loops running in opposite directions, it could be that the patched solution is not the optimal one (because of an intersection of the newly added edges). To address this issue, after each patching step we apply a 2-opt optimization to refine the solution.

An abstract description of the technique is presented in Algorithm 6.

Algorithm 6 Bender's loop + Patching Heuristic

```
Output: $solution$
procedure BENDERS+PATCHING($model$)
    while !$stop$ do
        $solution \leftarrow$ MIPSOLVER($model$)
        if COMPONENTS($solution$) == 1 then
            $break$
        end if
        $model \leftarrow model \cup SEC$
        $solution \leftarrow$ PATCHCOMPONENTS($solution$)
    end while
end procedure
```

4.2 Opening CPLEX

The title of this section refers to the fact that, to improve what we did in the previous section, we use an approach where we need to interact with CPLEX and control its execution.

The technique that will be presented in this section solves the problem described in Section 4.1 by generating the constraints when the MIP solver finds a new candidate solution. In order to achieve this, CPLEX provides us a C interface that we can use to control the execution of the branch-and-cut.

The process will go as follows: first, we will provide the solver with the ILP model, without the cuts. Whenever the solver finds a candidate solution, the execution will be passed to a function that we have defined.

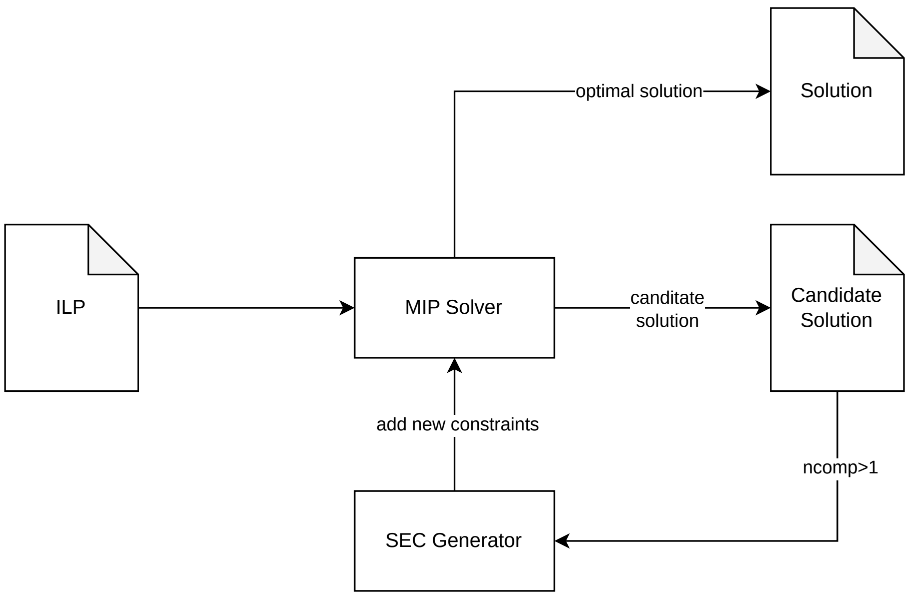
To do this we “install” a callback. CPLEX allows us to pass a function to it, that will be called when a specific condition is met. In our case we will use one[1] that is called when CPLEX has found a new candidate for an integer-feasible solution.

Inside this function we will count the number of components. If the number of components is greater than one, we will generate the corresponding subtour elimination constraints and reject the solution.

It is possible to see a diagram of such process in Figure 6.

[1]CPX_CALLBACKCONTEXT_CANDIDATE

Figure 6: Solving using candidate callback



Even though, as we will see in Section 6, the method mentioned above allows us to obtain satisfying results, there exist ways to improve it even more. Three of these methods will be covered in this report.

4.2.1 Warm start

The first method involves providing the MIP solver with a starting solution, known as a "warm start". CPLEX will save this solution in its "solution pool" and will be able to use it if it can be useful. The MIP solver will process the provided solution before starting the branch-and-cut algorithm, giving it an incumbent solution, that can help eliminating portions of the search space, potentially resulting in a smaller branch-and-cut tree.

4.2.2 Solution posting

The second method is the "heuristic solution posting", where heuristic solutions are provided to the solver during the optimization process.

The procedure we developed works as follows: every time the callback function is executed, as we have previously seen, CPLEX will have found a solution formed by many subtours. After generating the SECs, we can leverage the method we have described in Section 4.1.1 to "glue" the subcycles together and obtain a feasible solution. We will see in Section 6 that *posting* this solution to CPLEX can be beneficial to the process.

4.2.3 Fractional cuts

The third and last method we tried was the generation of violated cuts from the relaxed solutions CPLEX runs into. In order to do so, we use a callback[2] that is invoked whenever CPLEX has found a relaxed solution. This solution is usually not integer and, most of the times, comes from the solution of a node LP-relaxation.

It is possible to show that *the separation problem for finding a violated subtour elimination constraint can be reduced to a maximum flow problem.* To accomplish this we used a routine from the Concorde [3] software to solve such maximum flow problem, corresponding to the LP relaxation of our TSP problem.

This procedure involves the function `CCconnect_components`, which identifies connected components in the graph, and `CCviolated_cuts`, which generates violated cuts.

By using these functions we can add SECs derived from the fractional solution of an integer relaxation of the problem, rather than from the integer solution.

Generating cuts from these solutions can help the solver to narrow down the size of the search space, eliminating useless branches early. Hence the algorithm can converge to the optimal solution more efficiently.

Since this procedure is time consuming, we cannot afford to execute it on every node of the branch-and-cut tree. Our approach was to apply it only on the solutions generated by a subset of the nodes. We arbitrarily decided to perform it on a tenth of the nodes, using the modulo operator to decide which ones. In our implementation, the relaxed solutions coming from a node with index divisible by 10 will go through this procedure. By using the modulo operator, we make sure that such operation is always performed when the solutions are coming from the root node of the branch-and-cut tree, as its index is 0 and therefore a multiple of 10.

[2]CPX_CALLBACKCONTEXT_RELAXATION

5 Matheuristics

In this section we will explore another useful set of techniques that we will call *matheuristics*. The core idea behind these techniques is to use heuristics in conjunction with the mathematical models that one would normally use to solve the problem optimally. One of the advantages of such approaches is that they can be applied to different problems, with little to none needed adaptations.

What is important to note here is that matheuristics isn't a new class of techniques for solving optimization problems but it is instead a nomenclature that was given to the concept of using mathematical tools to design heuristics [5].

The term was coined to give a name to the first edition of a workshop in Bertinoro, Italy in 2006.

In this report we will focus on two matheuristic techniques. The first one that will be explored is Diving, while the second one is Local Branching.

As we will see in Section 6, these techniques will be used to solve heuristically instances bigger than what we will be able to solve optimally using approaches based on the branch-and-cut.

5.1 Diving

The first technique that we are going to explore is *Diving*.

Diving is a refinement algorithm, meaning that it is a way to improve a feasible sub-optimal solution.

The idea goes as follows: we start from a solution (that can be obtained with an heuristic or in the worst case generated randomly) and we fix part of the selected edges. The fixing of the edges will be done by adding constraints to the model, allowing us to use the MIP solver as a black box.

We can fix all the edges

$$x_e = 1 \quad \forall e \in \tilde{E}$$

where

$$\tilde{E} \subset E^H := \left\{ e \in E : x_e^H = 1 \right\}$$

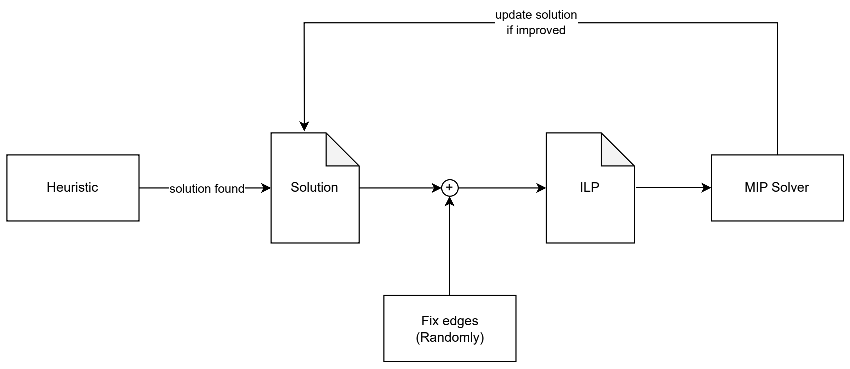
meaning that the edges we fix are a subset of the edges that are part of the heuristic solution.

It is important to note that only the edges with a value of 1 are fixed. We decide to focus on "positive" decisions to avoid overly constraining the subproblem, thereby maintaining a more flexible and effective search space that enhances the chances of finding optimal or near-optimal solutions [11].

What is left now is deciding how many and which edges to fix. The approach that will be explored in this report is choosing a number of edges (which will become an hyperparameter of our system) and select the edges to fix randomly.

Figure 7 shows a description of the process.

Figure 7: Diving



5.2 Local Branching

The second approach this report will focus on is *Local Branching.*

Local Branching was initially introduced in a paper by Fischetti and Lodi [8] and was then applied by many other researchers over the following years.

The idea behind this technique is to decide the number of edges to fix, like we did for Diving, but then leaving the choice of *what edges* to the MIP solver.

This is possible thanks to the Local Branching constraints, which, in their simplified asymmetric version are constructed like this

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

This forces at least $n - k$ edges of the heuristic solution we apply this method on to be set to 1 in the solution of the model this constraint is part of.

We say that this is an *asymmetric* version because the “original” Local Branching constraint would consider also the variables set to 0 becoming 1. In the TSP problem it would make no sense to add such constraints since every feasible solution has the same number of variables set to 1.

The *simplified* comes instead from the fact that we know a priori the number of variables set to 1 in a solution. This allows us to put n directly into the right-hand-side of the constraint and avoid building a more complicated cut where we consider the difference of variables set to 1 between the input heuristic solution and the solution of the model.

In its original version the Local Branching constraint introduced by Fischetti, that could be written as

$$\sum_{i\in I:\hat{x}_i=0} x_i + \sum_{i\in I:\hat{x}_i=1} (1 - x_i) \leq k,$$

limits the Hamming distance between the heuristic and the solution we will find to a value k.

This can be seen as a technique to use the MIP solver as a black box to find the best k-opt moves in a feasible solution.

This leaves us with one hyperparameter to tune: the value of k for each iteration. Our approach in this report will be to decide an initial value of k that we will call k_0. The first iteration of the local branching will use $k_1 = k_0$. We will keep running the solver with the same value of k until no improved solution can be found. At that point, say iteration t, we increment k by a delta: $k_{t+1} = k_t + \Delta k$. This process will continue until we reach the timelimit.

6 Results

This section is structured as follows: first we will try to optimize the hyperparameters of the approaches that have some. Then, we will compare the methods in their best configuration, searching for the best one.

Naturally, the main difference we have among the approaches that we present is if we are computing an optimal solution or if we are applying an heuristic. In the former case, the metric we are going to compare our systems on is the time it took them to arrive to the optimal solution. For the latter we instead fix a time limit (the maximum time we allow our system to run for) and we seek the system that found the best solution (in our case the smallest one, since the TSP is a minimization problem).

It is important to know that we set a timelimit also for the exact method. This needs to be done for practical reasons while running the experiment.

Before going through the experiments that were performed, it can be useful to discuss some parameters that will be used in this section. The instances will be generated randomly and we will refer to them using the seed we initialize our number generator with. The size of the instances will be

- 300 for the optimal methods
- 1000 for matheuristics
- 1500 for heuristics

For what concerns the value of the time limit, we will use 120 seconds for all the experiments.

To make comparisons a little bit more statistically valid, all the systems will be run on 20 different instances of the same size. We will then take the results for each instance and plot the Performance Profile [7]. All our considerations will be based on these plots.

It is also important to discuss how the distance between the points in the 2d grid was computed: in our experiments we use the ATT formula from the TSPLIB [13].

6.1 Experimental Setup

All the experiments were run on the same machine equipped with an Intel Core i7 6700k and 16GB of RAM. The MIP solver used was IBM ILOG CPLEX 22.1.0 [10].

6.2 Performance profile

Performance profiles are graphical representations used to compare the performance of different algorithms across multiple problem instances [7]. A *Performance Profile* is a cumulative distribution function used to compare the performance of a set of solvers on a given set of problems. Given n_s solvers and n_p problems, let $t_{p,s}$ be the computing time required by solver s to solve problem p. The performance ratio $r_{p,s}$ is defined as:

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in S\}}$$

where S is the set of all solvers. To handle cases where a solver fails to solve a problem[3], a parameter $r_M \geq r_{p,s}$ is chosen such that $r_{p,s} = r_M$ if solver s does not solve problem p.

[3]This doesn't happen in the experiments that will be presented

The performance profile $\rho_s(\tau)$ for solver s is defined as the fraction of problems for which the performance ratio $r_{p,s}$ is within a factor τ of the best possible ratio:

$$\rho_s(\tau) = \frac{1}{n_p} \left| \{ p \in P : r_{p,s} \leq \tau \} \right|$$

where P is the set of all problems. The function ρ_s is a nondecreasing, piecewise constant function, continuous from the right at each breakpoint.

In summary, the performance profile $\rho_s(\tau)$ represents the probability that the solver s performs within a factor τ of the best possible performance on the given set of problems. A higher value of $\rho_s(\tau)$ indicates better overall performance of the solver.

It is important to note that, in the case of heuristics, we will use $c_{p,s}$, the cost obtained by solver s for problem p, instead of $t_{p,s}$, allowing us to compare the approaches based on the value of the solution.

Using these plots allows for identifying which algorithm is consistently better across a range of problem instances. We will use them widely in the next sections for hyperparameter tuning and to determine the best method among different algorithms.

6.3 Hyperparameters tuning

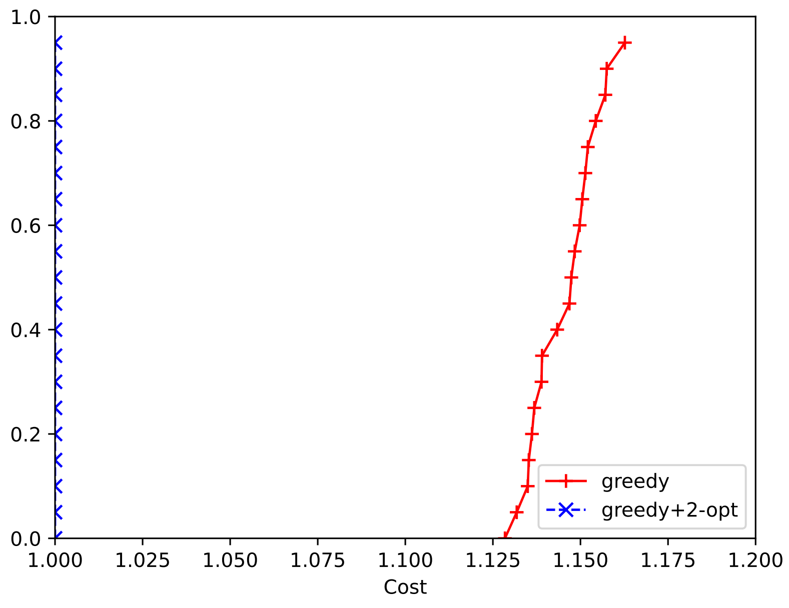
Before comparing the methods with each other, it is useful to discuss some of the hyperparameters these approaches require.

6.3.1 Greedy

With regard to the greedy approach, the hyperparameter we should tune is deciding whether or not to use the 2-opt procedure. We can see the plot in Figure 8. From this performance profile we can conclude that using the 2-opt procedure is beneficial to the cost of the best solution we can find.

Specifically, integrating the 2-opt algorithm resulted in a notable improvement of 12% in solution cost.

Figure 8: Performance profile of the greedy approaches



6.3.2 Tabu search

On the subject of Tabu Search, there are a few hyperparameters we need to tune: we need to discuss what the size of the tenure will be. In our experiments we will explore two different approaches: the first one will use a fixed value of the tenure for the whole experiment. The second approach we will cover is to use a sinusoidal value for the tenure.

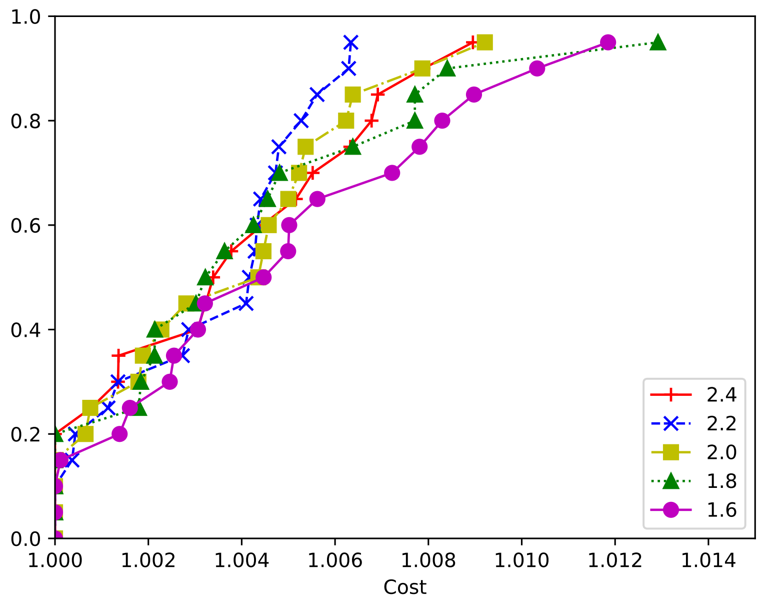
In the first case there will be only the value of the fixed tenure to tune. The size of the tenure T is computed as follows.

$$T = \max\left\{\frac{n}{\alpha_0}, T_{min}\right\}$$

where n is the number of nodes, T_{min} is the minimum value for the tenure and α_0 is the parameter we want to tune. The value of T_{min} is arbitrarily set to 10.

The performance profile for this parameter is presented in Figure 9.

Figure 9: Performance profile of the fixed Tabu Search on the parameter α_0



Although all parameters yield very similar performance, the Tabu Search with $\alpha_0 = 2.2$ appears slightly superior to the others. Therefore, we will adopt this parameter for the subsequent experiments.

With regard to the second approach, we have more than one parameter. First of all we will need to decide the parameters for the sinusoidal function. In our case the value of the tenure T will be computed on the base of the current iteration i. We will also include the number of nodes n and two parameters that we will use to scale the sinusoidal function, α_1 and α_2. Another parameter that will be used is a minimum value for the tenure, indicated with T_{min}. In our experiment we set this value to 10.

First of all we compute the value of the scale s:

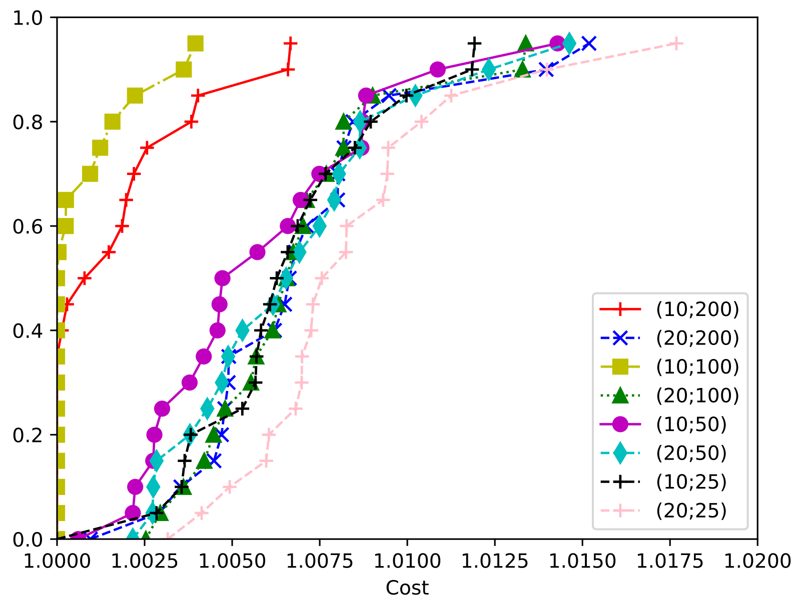
$$s = \frac{n}{\alpha_1}$$

Then we can compute the value of the tenure:

$$T = \max\left\{\sin\left(\frac{i}{\alpha_2}\right) \cdot s + 2s, T_{min}\right\}$$

The performance profile for this parameter is presented in Figure 9.

Figure 10: Performance profile of the sinusoidal Tabu Search on the parameters $(\alpha_1; \alpha_2)$

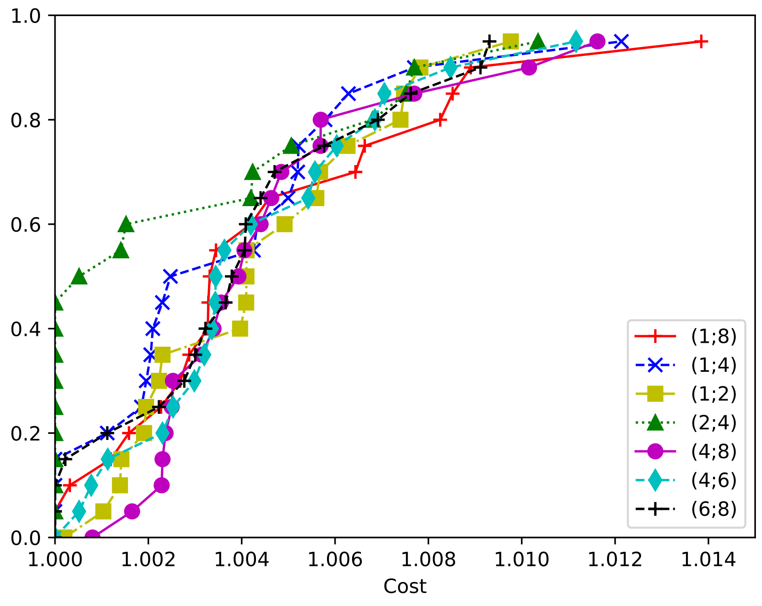


Tabu Search with sinusoidal tenure parameters (10;100) and (10;200) exhibit significantly better performance compared to the others. The set of parameters (10;100) appears slightly superior to the other one. Therefore, we'll use these parameter for the subsequent results.

6.3.3 VNS

The parameter we have to tune for the VNS is the number of "kicks" we apply after reaching a local minimum. Since in our implementation we generate a random number of kicks to be applied, we have two parameters to tune: the minimum the maximum number of kicks. The performance profile for this experiment is shown in Figure 11.

Figure 11: Performance profile of the VNS on the parameters $(min; max)$



In this scenario, the performance of the algorithms with the different parameters does not vary significantly. For the following results, we decided to use VNS with parameter (2;4) as it demonstrates slightly better performance across most cases.

6.3.4 Fast-heuristic

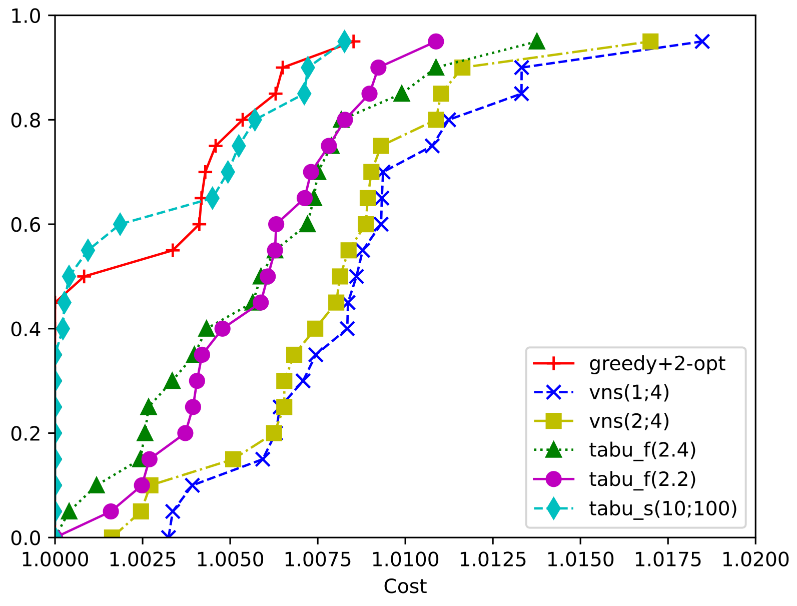
Since for the next experiments we will need an heuristic to run for a tenth of the total available timelimit, we are going to discuss what is the best procedure we can use to find the first heuristic solution. The total available runtime is 120 seconds, so we will compare the heuristics we discussed so far on a 12 seconds timelimit.

Since this heuristics will be used as an initial phase for matheuristic, the experiments were run with 1000 nodes.

To avoid comparing all the heuristics again, we picked one or two (in the case a clear winner is not present) from every approach.

The label in the legend should be interpreted as the name of the method (as covered in the previous sections) followed by the parameters, if present. *tabu_f* refers to the Tabu Search using a fixed tenure while *tabu_s* refers to the sinusoidal tenure version.

Figure 12: Performance profile of the heuristics with 12 seconds

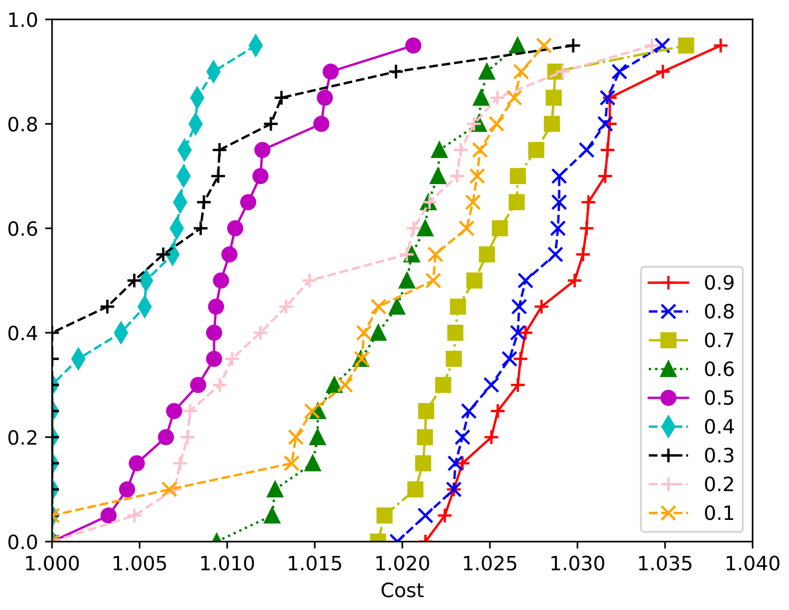


From the plot we can see that the two best performing approaches are the greedy with 2-opt and Tabu Search with a sinusoidal tenure with parameters $(\alpha_1; \alpha_2) = (10; 100)$. By interpreting this plot, we decided to use greedy in conjunction with 2-opt as the initial heuristic for the next experiments.

6.3.5 Diving

With regard to the diving algorithm, we need to tune the percentage of edges to fix in the starting solution provided by the greedy + 2-opt algorithm to enable the MIP solver to handle the otherwise too large problem. The results are shown in the performance profile in Figure 13

Figure 13: Performance profile of Diving

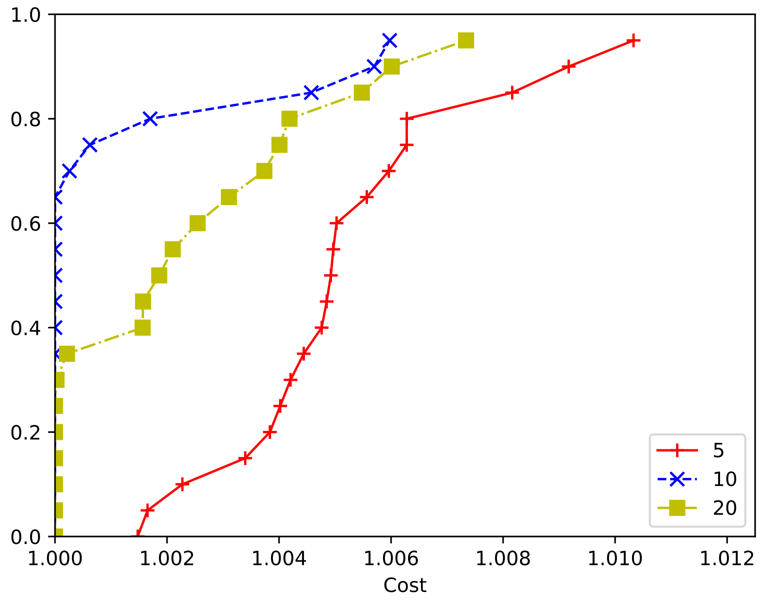


Fixing too many edges of the provided solution (80%, 90%) leads to the worst performance. This may be because the MIP solver struggles to find a good solution due to the limited degrees of freedom we allow it. Instead, fixing the 40% of the edges belonging to the solution, and hence giving CPLEX more degrees of freedom, leads to the best performance.

6.3.6 Local branching

As we mentioned in Section 14, the local branching algorithm depends on two hyperparameters: the initial number k_0 of edges to be fixed and the Δk used to increment the generic k_t when an improved solution is found. Figure 14 shows the performance profile for tuning k_0. For simplicity, Δk is arbitrarily set to 5.

Figure 14: Performance profile of Local Branching

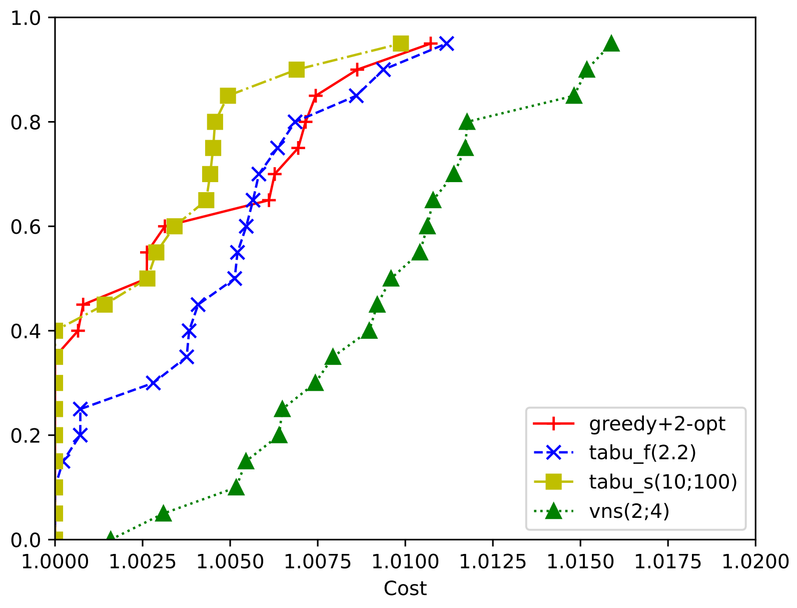


The best performance is obtained by setting $k_0 = 10$, so we will use this parameter to compare it with the other matheuristics.

6.4 Best heuristic

In this section we will show a comparison between the best hyperparameter configuration for the presented heuristic techniques. Figure 15 depicts the performance profile among all the heuristic algorithms discussed previously.

Figure 15: Performance profile among the different approaches



Greedy + 2-opt and Tabu Search with sinusoidal tenure exhibit very similar performance across most instances, while VNS and Tabu Search with fixed tenure are less effective. In the end, the Tabu Search with sinusoidal tenure stands out as the best heuristic because it consistently performs above the other curves across nearly all instances.

6.5 Best exact method

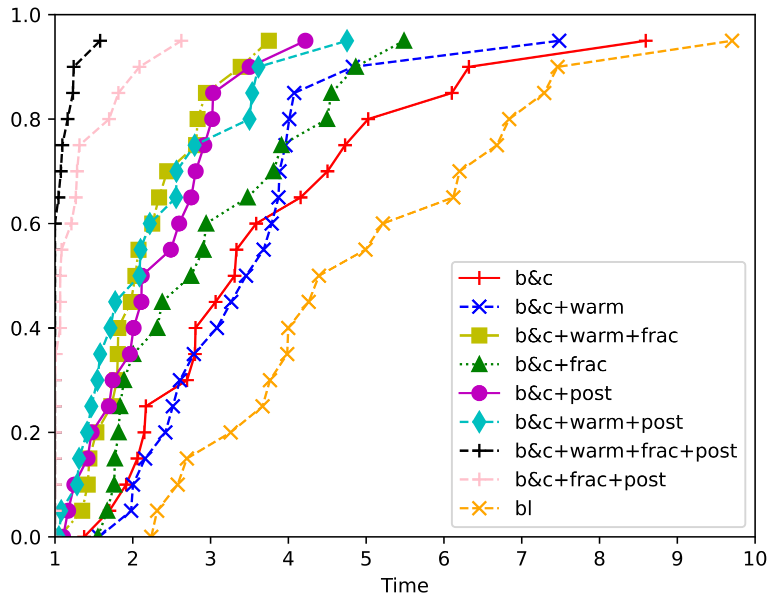
In this section we will show a comparison between all the exact methods we discussed previously. The following clarifies the meaning of the labels of Figure 16:

- **b&c**: we interact with CPLEX adding the SECs like discussed in Section 4.2
- **bl**: Benders' loop, covered in Section 4.1

For the first approach, we also test the following additional techniques:

- **warm**: "warm start" is used
- **frac**: cuts generated from fractional solutions are added
- **post**: heuristic solutions generated from infeasible solutions are posted

Figure 16: Performance profile of the various branch-and-cut techniques

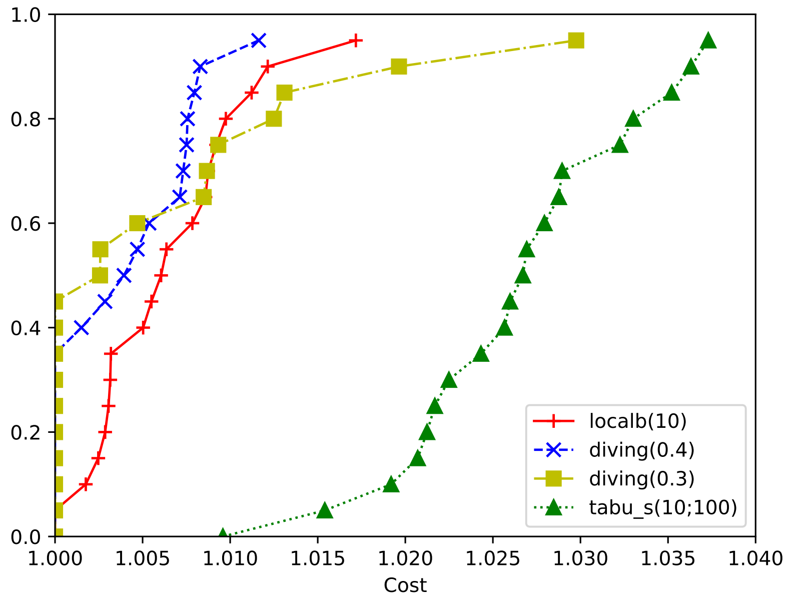


Every branch-and-cut variation have better performance compared to the Benders' loop technique. While warm start does not significantly improve the performance of the branch-and-cut, fractional cuts and heuristic solution posting variations show better results. The combination of all these techniques proves to be the most effective.

6.6 Best matheuristic

In this section we'll show a comparison between all the matheuristic algorithms we presented. Figure 17 shows the performance profile of these techniques. For reference, it shows also what the best heuristic (Tabu Search with sinusoidal tenure) would have performed compared with the matheuristic algorithms.

Figure 17: Performance profile of the various matheuristic techniques, along with the best heuristic



The matheuristic algorithms outperform the Tabu Search. Among them, the diving methods demonstrate superior performance compared to the local branching method.

7 Conclusions

In this study, we explored various methods for solving the Traveling Salesman Problem. Our analysis included the use of heuristics and matheruristics to obtain near-optimal solutions for large instances, and advanced techniques such as branch-and-cut and fractional cuts to find optimal solutions for smaller instances. Among the heuristic algorithms, we found that Greedy + 2-opt and Tabu Search with sinusoidal tenure achieved very similar performance, both outperforming VNS and Tabu Search with fixed tenure. However, Tabu Search with sinusoidal tenure emerged as slightly better overall, as shown in the performance profiles. In our comparison of exact methods, we observed that providing the solver with heuristic solutions during the optimization process and using fractional cuts were more effective than using a warm start. The best results were achieved by combining all these techniques, as evinced from performance profiles. We also observed that matheuristics are more effective than a pure Tabu Search approach, demonstrating that the combination of heuristic and exact methods can be a winning strategy when the instance sizes are too large to be solved by exact methods alone. The performance profiles have also shown that the Diving algorithm outperformed Local Branching. Future research could focus on finding improved methods to tune hyperparamenters, possibly leveraging Deep Learning models. Additionally, the implementation of other heuristic techniques such as Simulated Annealing and Genetic Algorithms could be explored.

References

[1] Tobias Achterberg. *Constraint Integer Programming.* Ph.d. thesis, Technische Universität Berlin, Berlin, Germany, 2007.

[2] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

[3] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, pages 645–656, 1998.

[4] Norman Biggs, E Keith Lloyd, and Robin J Wilson. *Graph Theory, 1736-1936.* Oxford University Press, 1986.

[5] Marco Antonio Boschetti and Vittorio Maniezzo. Matheuristics: using mathematics for heuristic design. *4OR*, 20(2):173–208, 2022.

[6] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[7] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91:201–213, 2002.

[8] Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical programming*, 98:23–47, 2003.

[9] Fred Glover and Manuel Laguna. *Tabu Search.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[10] IBM Corporation. *IBM ILOG CPLEX Optimization Studio Documentation.* IBM Corporation, Armonk, NY, 22.1.0 edition, 2023.

[11] Vittorio Maniezzo, Marco Antonio Boschetti, and Thomas Stützle. Diving heuristics. *Matheuristics: Algorithms and Implementations*, pages 133–141, 2021.

[12] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

[13] Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[14] Boldizsár Tüű-Szabó, Péter Földesi, and László T Kóczy. Analyzing the performance of tsp solver methods. In *Computational Intelligence and Mathematics for Tackling Complex Problems 2*, pages 65–71. Springer, 2022.