UNIVERSITÀ DEGLI STUDI DI PADOVA

SCHOOL OF ENGINEERING DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN COMPUTER ENGINEERING

Exact and heuristic approaches to the Travelling Salesman Problem (TSP)

Operations Research 2

Supervisor:
Matteo Fischetti

Candidates:
Fantin Luca (2119287)
Zanella Matteo (2122187)

Accademic Year 2023/2024

Contents

Chapter 1

Introduction

The *Travelling Salesman Problem*, also known as *TSP*, is one of the most famous and studied optimization problems in the Computer Science and Operations Research fields. Although its first mathematical formulation was proposed in the 19th century by the mathematicians *William Rowan Hamilton* and *Thomas Kirkman*, it received scientific attention from the 1950s onwards.
In 1972, *Richard M. Karp* proved the *NP-hard nature* of the TSP; this meant that the computation time for any solving algorithm can grow exponentially with the input size. Despite this, many different approaches have been developed over the years, yielding both exact and approximate solutions.

In this paper, several algorithms are explained, developed and tested against each other, both exact and approximate.

1.1 Problem formulation

Consider an *undirected graph* $G = (V, E)$, where V is a set of $|V| = N$ *nodes* (or *vertices*) and E is a set of $|E| = M$ *edges.*
Define a *Hamiltonian cycle* of G, $G^* = (V, E^*)$, as a graph whose edges form a cycle going through each node $v \in V$ exactly once.
We also define a *cost function* for the edges $c : E \to \mathbb{R}^+$, $c_e \coloneqq c(e)\ \forall\ e \in E$.

The target of the TSP is finding an Hamiltonian cycle of G of minimum total cost, obtained by summing the costs of all edges in the cycle.
We can formulate this problem through *Integer Linear Programming (ILP).*
First, let's define the following decision variables to represent whether or not a certain edge is included in the Hamiltonian cycle:

$$x_e = \begin{cases} 1 & \text{if } e \in E^* \\ 0 & \text{otherwise} \end{cases} \qquad \forall\ e \in E$$

The ILP model is the following:

$$\begin{cases} \min \sum_{e \in E} c_e x_e & (1.1) \\ \sum_{e \in \delta(h)} x_e = 2 \quad \forall\, h \in V & (1.2) \\ \sum_{e \in \delta(S)} x_e \leq |S| - 1 \quad \forall\, S \subset V : v_1 \in S & (1.3) \\ 0 \leq x_e \leq 1 \quad \text{integer} \quad \forall\, e \in E & (1.4) \end{cases}$$

Constraints 1.2 impose that every node of the graph must be touched by exactly two edges of the cycle. This group of contraints alone isn't enough to guarantee to find a valid Hamiltonian Cycle: the solution found could be composed by lots of isolated cycles.
Constraints 1.3, called *Subtour Elimination Constraints (SEC)*, guarantee that any solution found through this model is made up of only one connected component: every vertex $v \neq v_1$ must be reachable from v_1.
Despite their importance, their number is exponential in N, thus, considering all of them at once is computationally expensive.

Chapter 2

Project Setup

...

2.1 Generating the instances

...

2.2 Reading the solutions

...

2.3 Performance profiler

...

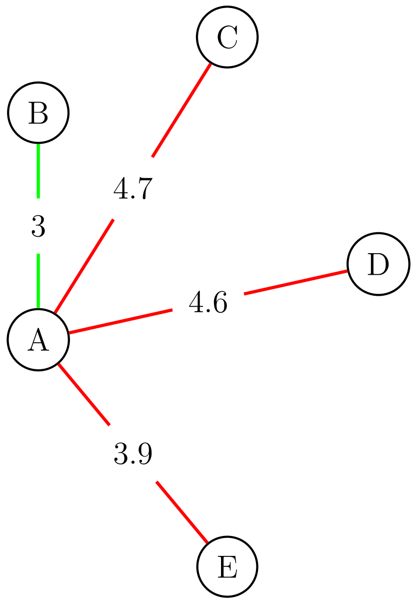
Chapter 3

Heuristics

An heuristic is any approach to problem solving that employs a practical method that's not fully optimized but it's sufficient to reach an immediate short-term goal or approximation.
Given the NP-Hard nature of the Travelling Salesman Problem, finding the optimal solution may require a long time, hence the need to have heuristics method to find solutions that are close to the optimal.

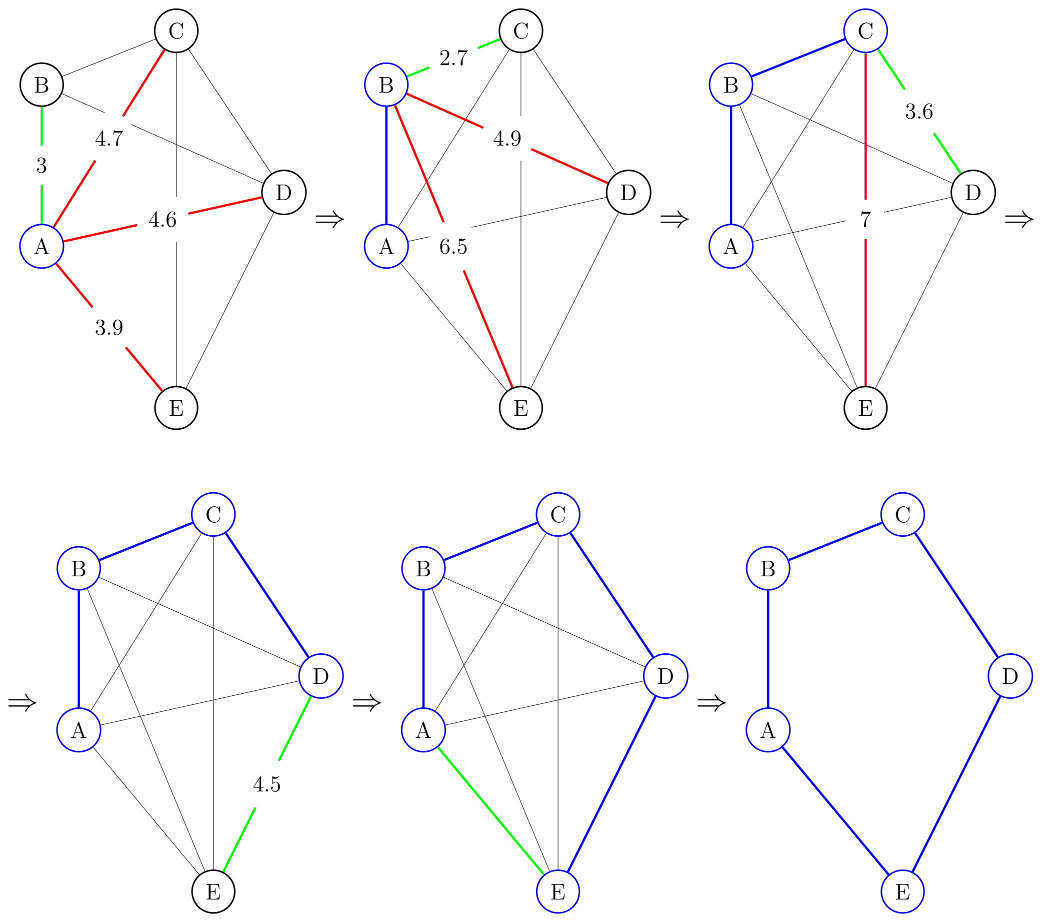
3.1 Nearest Neighbor (Greedy)

A first approach to the TSP, is to consider the edges with the *smallest weights first* when choosing the next node in the cycle.
This type of logic is called *greedy*: a greedy algorithm looks for the locally optimal choice at each stage.



In this example, among the edges connected to the node A, the edge (A, B) is the one with the smallest weight, so node B should be A's successor.

By repeating this process for each new node added to the path and connecting the last node (E) to the starting node (A):



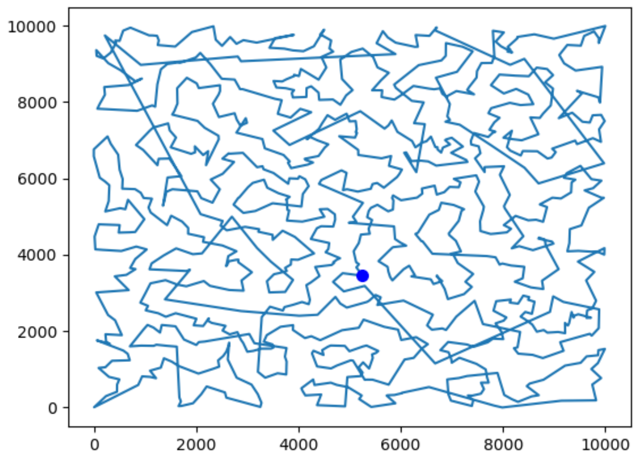
3.1.1 Pseudocode

Algorithm 1 Greedy algorithm for the TSP

```
    Input Starting node ($s \in V$), Set of nodes ($V$)
    Output List of $n := |V|$ nodes forming an Hamiltonian cycle, Cost of the cycle
cycle $\leftarrow [s]$
cost $\leftarrow 0$
for $i = 0$ to $n - 2$ do
    next $\leftarrow \text{argmin}_v \{c_{cycle[i],v} \mid v \in V \wedge v \notin \text{cycle}\}$
    cost $\leftarrow$ cost $+ c_{cycle[i],next}$
    cycle$[i+1] \leftarrow$ next
end for
cost $\leftarrow$ cost $+ c_{cycle[n-1],s}$

return cycle, cost
```

The solution found using the greedy algorithm is dependent to the starting node: a possible solution to this is to iterate through all possible starting nodes and keeping track of the best solution found so far.

3.1.2 Results analysis

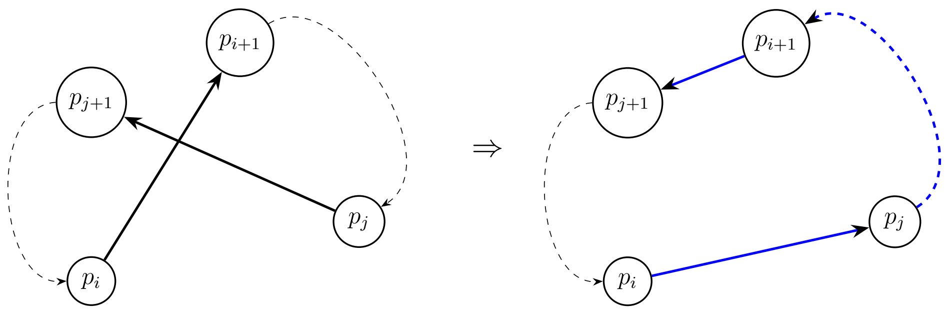


Instance: **A**, Algorithm: **greedy**, Cost: **282030.8675**

The solutions found with the greedy algorithm are a good starting point, but sometimes the algorithm uses edges that cross a long distance, increasing a lot the cost of the solution.
This is caused by the fact that the greedy algorithm optimizes locally, without knowing wether that local choice is good or not in the long term: it might happen that after adding some edges to the cycle, the closest nodes are all already in the cycle, so the edge that will be considered may have a very large weight.
In the next section a tecnique is explored that will fix this problem.

3.2 2opt

The 2opt algorithm takes an existing cycle and tries to improve it's cost by changing some of the edges that compose the cycle without braking it.
The main idea behind the 2opt algorithm is to find two edges that cross each other and fix them by removing the intersection:



$p_i :=$ cycle$[i]$

This process is then repeated until no more edges that can lead to an improvement to the cost of the cycle can be found.
To find a pair of edges that can be changed to improve the cost it's sufficient to find a pair of nodes p_i, p_j, that satisfies the following inequality:

$$c_{p_i,p_{i+1}} + c_{p_j,p_{j+1}} > c_{p_i,p_j} + c_{p_{i+1},p_{j+1}}$$

Note that the cycle list is to be considered as a circular array: situations with indexes out of bounds or $i > j$ won't be treated in this paper since the fix is trivial.

If this inequality holds then swapping the edges (p_i, p_j), (p_{i+1}, p_{j+1}) with (p_i, p_{i+1}), (p_j, p_{j+1}) will lower the cost of the cycle:

$$\begin{aligned} \text{cost(new cycle)} &= \text{cost(old cycle)} - (c_{p_i,p_{i+1}} + c_{p_j,p_{j+1}}) + (c_{p_i,p_j} + c_{p_{i+1},p_{j+1}}) \\ &\leq \text{cost(old cycle)} \end{aligned}$$

After finding the nodes p_i, p_j the edges (p_i, p_j) and (p_{j+1}, p_{j+1}) will take the place of the edges (p_i, p_{i+1}) and (p_j, p_{j+1}), reversing the route connecting $p_{i+1} \to p_j$.
Suppose the cycle is stored as a list of nodes ordered following the order of the nodes in the cycle, then this step can be done simply by *reversing the list* from the index $i + 1$ to the index j:

$$[..., p_i, \underline{p_{i+1}, ..., p_j}, p_{j+1}, ...] \Rightarrow [..., p_i, \underline{p_j, ...\text{(reversed)}..., p_{i+1}}, p_{j+1}, ...]$$

3.2.1 Pseudocode

Algorithm 2 2opt algorithm for the TSP

Input List (cycle) of $n := |V|$ nodes forming an Hamiltonian cycle, Cost (cost) of the cycle

Output List of n nodes forming an Hamiltonian cycle (with 2opt swaps applied), Cost of the new cycle

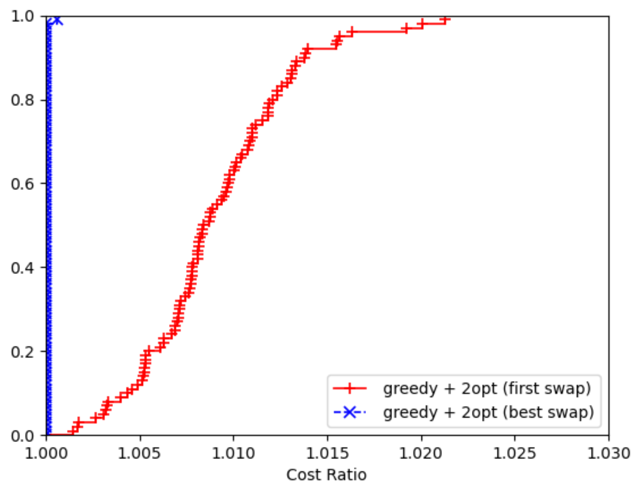
```
while *Exists a swap improving the cost* do
    $(i, j) \leftarrow$ find_swap(cycle)
    cost $\leftarrow$ cost $- (c_{p_i,p_{i+1}} + c_{p_j,p_{j+1}}) + (c_{p_i,p_j} + c_{p_{i+1},p_{j+1}})$
    reverse(cycle, $i + 1, j$)
end while

return cycle, cost
```

3.2.2 Swap policy

In this paper two different ways of finding a swap have been confronted:

- Returning the *first swap* found that improves the cost
- Looking among all possible pair of edges and returning the swap that improves the cost the most (the *best swap*)

Using the performance profiler it's easy to see that the best swap policy finds solutions with an improvement of a 2% factor with respect to the first swap policy:

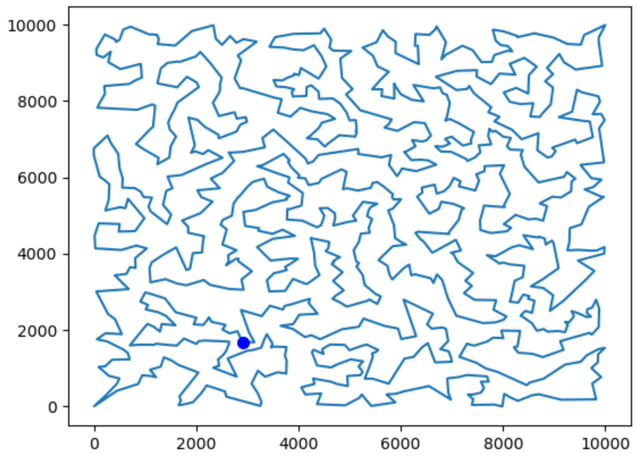


100 instances, Time limit: 120s

This is due to the fact that the first swap policy might get lost in improving the solution by a small amount by finding swaps in a region with an high concentration of nodes while the best swap policy aims to improve the edges with the highest weights.
This leads the best swap policy to fix immediately the worst edges, lowering right away the cost of the solution by a large amount.
It's also worth noticing that the the first swap policy requires less time to improve the solution

3.2.3 Results analysis

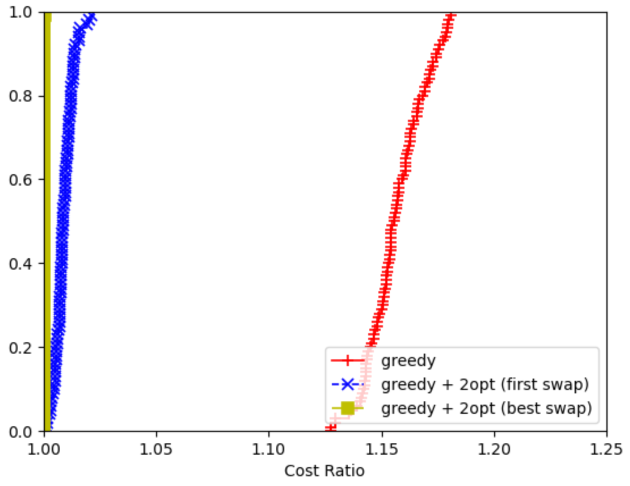
The 2opt algorithm is usually used after finding a cycle with the greedy algorithm to see wether it can be improved.
The solution displayed is obtained by running the 2opt algorithm, after applying the greedy algorithm on each possible starting node, and keeping the best one.



Instance: **A**, Algorithm: **greedy + 2opt (best swap)**, Cost: **244793.5477**

Confronting this solution with the one showed for the greedy algorithm, it's possible to notice that the starting node used for the greedy algorithm has changed: even though the greedy algorithm found that starting node to be the best one (the one that generates the best cycle), after improving the solution using the 2opt algorithm, another starting node has been found to be better.
This is a phenomenon quite common in heuristics: when trying to improve two different solutions, improving the worse solution can lead to better results than improving the better one. This concept will be further explored in the metaheuristics chapter.

3.3 Comparison Greedy / 2opt



100 instances, Time Limit: 120s

The 2opt algorithm manages to consistently improve the cost of the solutions of 15-20% with respect to the solutions found by the greedy algorithm.

Chapter 4

Metaheuristic

A metaheuristic approach is a method in which more than one heuristic is used, with the aim to guide the search processs to efficiently explore the search space.

Metaheuristics will unlock a greater search space with respect to heuristic, by allowing ”bad moves” to escape a locally optimal solution.

4.1 Tabu Search

The *Tabu search* algorithm is based on the idea of allowing the 2opt algorithm to perform swaps that still are the best ones, but not necessarly swaps that improve the cost of the solution.

This means that after finding a local optima, the 2opt algorithm will stop, while the tabu search algorithm will keep searching, moving away from that locally optimal solution, hoping to find a new locally optimal solution which has a lower cost.

Allowing a bad move, means that at the next iteration the new best move will revert the bad move, since that would be the only swap that lowers the cost.

To prevent this, we need to keep track of those bad moves and prevent them from being reverted, marking them as *tabu moves*, hence the name of the algorithm.

...

4.1.1 Storing a Tabu move

...

4.1.2 Results analysis

...

4.2 Variable Neighborhood Search (VNS)

...

4.2.1 Results analysis

...

4.3 Comparison Tabu / VNS

...

Chapter 5

Exact methods

The exact algorithms for the TSP algorithm described in this section are all based on the usage of commercial solver *CPLEX*, which uses a proprietary implementation of the *branch & bound (B&B)* method to return a solution to the input model. Most of the times, we can expect an "optimal" solution with an integrality gap close to zero. However, since it is computationally infeasible to add every possible SEC to the model, CPLEX will return a solution which is feasible for its internal model but infeasible for our original TSP problem. Thus, we use various techniques to find a good solution for the TSP problem using CPLEX.

5.1 Benders' loop

A simple approach to use SECs without computing all of them is the *Benders' loop* technique. We start with a model with no SECs. Given the solution returned by CPLEX, we identify its various connected components and compute the SECs on those components alone. We repeat the procedure with the new model until we get a solution with only one connected component or we exceed the timelimit.

This method is guaranteed to reach a feasible solution for the TSP problem if the timelimit is not exceeded, but this will happen only at the final iteration: if the time runs out before we find such feasible solution, we will have an infeasible one with multiple connected components. Moreover, it does not always improve the lower bound for the final solution, since the number of connected components of the solutions found throughout the algorithm's execution is not always decreasing.

5.1.1 Pseudocode

Algorithm 3 Benders' loop

```
  Input undirected complete graph $G = (V, E)$, cost function $c : V \to \mathbb{R}$
  Output List of $n := |V|$ nodes forming an Hamiltonian cycle, cost of the cycle
build CPLEX model from $G$ with objective function and degree constraints
while timelimit not exceeded do
    $x^* \leftarrow$ solution of CPLEX model
    determine connected components of $x^*$
    if $x^*$ has only connected component then return $x^*$
    end if
    for each connected component $S \subset V$ in $x^*$ do
        add SEC to model: $\sum_{e \in \delta(S)} x_e \leq |S| - 1$
    end for
end while
```

5.2 Patching heuristic

The major flaw of this method is returning a feasible solution only at the very last iteration. A possible solution to this issue is the implementation of a *patching* heuristic. Given the CPLEX solution at any iteration, we patch together the various connected components to provide a feasible solution even if the timelimit is exceeded before getting a solution with a single component.

Two components $k1 \neq k2 \subset V$ are patched by replacing two edges $(p_i, p_{i+1}) \in k1$ and $(p_j, p_{j+1}) \in k2$ with the pair of edges of minimal cost between $(p_i, p_j), (p_{i+1}, p_{j+1})$ and $(p_i, p_{j+1}), (p_j, p_{i+1})$. We iterate through all combinations of $k1, k2$ to find the swap with the lowest increase in cost. Once we find the swap, we perform it and repeat the process until we are left with only one connected component.

This procedure may introduce some crossing edges into x^*. Thus, we apply the 2opt algorithm after this procedure to remove them.

This procedure solves the biggest flaw of Benders' loop with a patching procedure which requires a little amount of time compared to the whole algorithm.

5.2.1 Pseudocode

Algorithm 4 Patching heuristic Benders' loop

```
  Input solution $x^*$ returned by CPLEX with $ncomp$ connected components
  Output $x^*$ with 1 connected component
while $ncomp \neq 1$ do
    $best\_k1 \leftarrow 0, best\_k2 \leftarrow 0, best\_delta \leftarrow -\infty$;
    for $k1 \leftarrow 0$ to $ncomp - 1$ do
        for $k2 \leftarrow k1 + 1$ to $ncomp - 1$ do
            for each $(p_i, p_{i+1}) \in x^*$ with $p_i, p_{i+1} \in k1$ do
                for each $(p_j, p_{j+1}) \in x^*$ with $p_j, p_{j+1} \in k2$ do
                    $delta\_N \leftarrow$ change in cost of solution produced by replacing
                    $(p_i, p_{i+1}), (p_j, p_{j+1})$ with $(p_i, p_{j+1}), (p_j, p_{i+1})$ in $x^*$;
                    $delta\_R \leftarrow$ change in cost of solution produced by replacing
                    $(p_i, p_{i+1}), (p_j, p_{j+1})$ with $(p_i, p_j), (p_{j+1}, p_{i+1})$ in $x^*$;
                    $delta \leftarrow \max\{delta\_N, delta\_R\}$;
                    if $delta > best\_delta$ then
                        $best\_delta \leftarrow delta$;
                    end if
                end for
            end for
        end for
    end for
    patch components $best\_k1, best\_k2$ applying transformation with change in cost
    $delta$;
    cost of $x^* \leftarrow$ cost - $best\_delta$;
    $ncomp \leftarrow ncomp - 1$;
end while
```

Chapter 6

Bibliography