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 : 2023/2024

Supervisor:
Matteo Fischetti

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

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

...

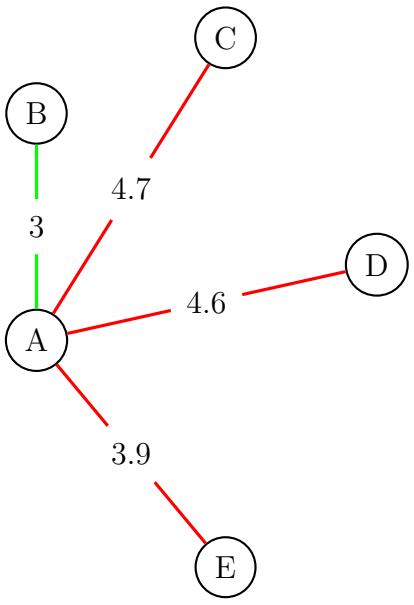
Chapter 3

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.

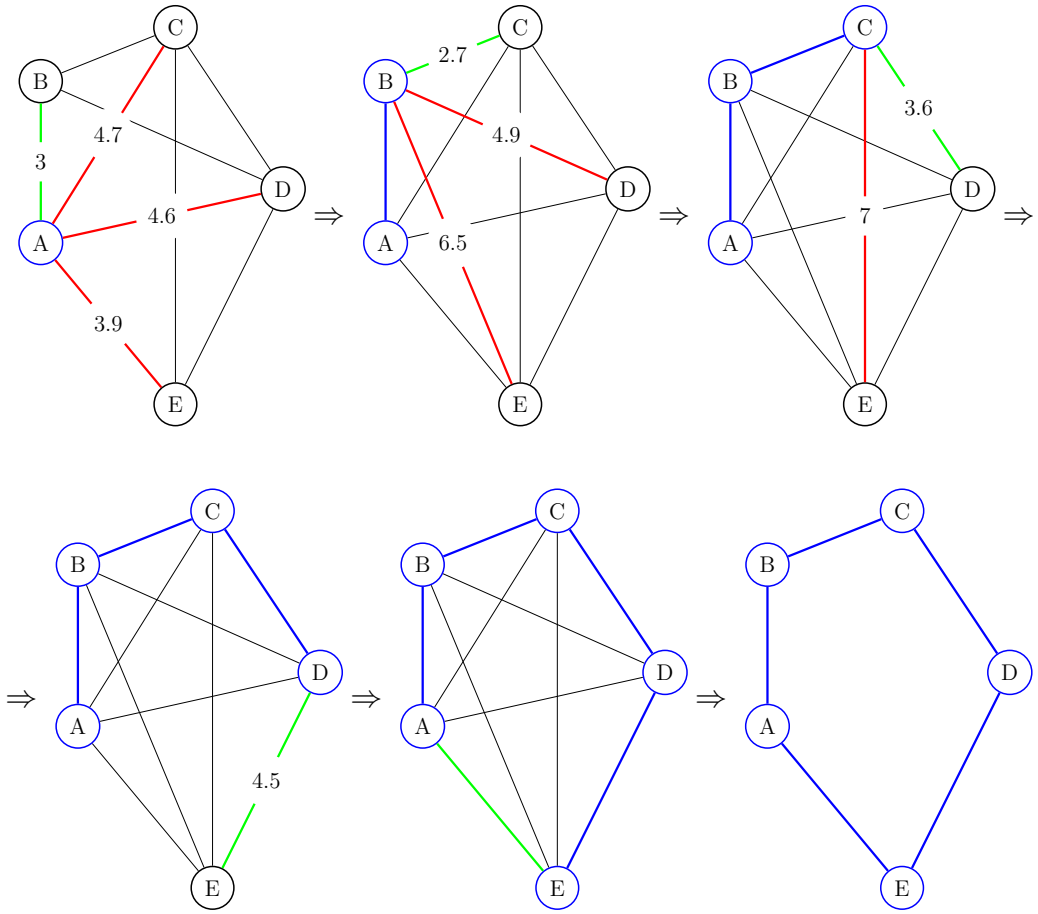
3.1 Nearest Neighbor

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

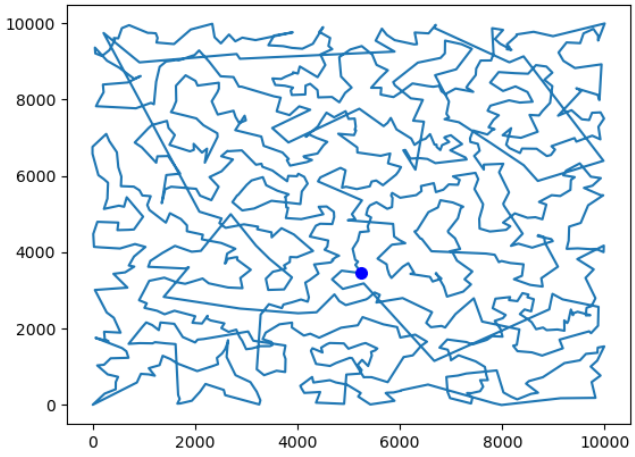
Input Starting node ($s \in V$), Set of nodes (V)
Output List of $n := |V|$ nodes forming an Hamiltonian cycle, Cost of the cycle

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

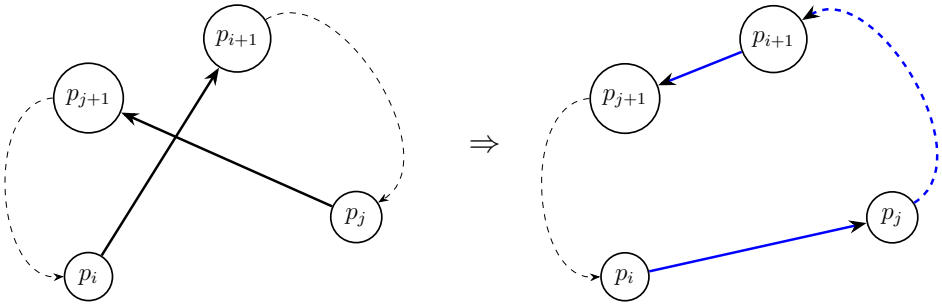


Instance: **A**, Algorithm: **greedy**, Cost: **282030.8675**

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 2-opt

The 2-opt 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 2-opt algorithm is to find two edges that cross each other and fix them by removing the intersection:



$p_i := \text{cycle}[i]$

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 2-opt algorithm for the TSP

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

Output List of n nodes forming an Hamiltonian cycle (with 2-opt 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

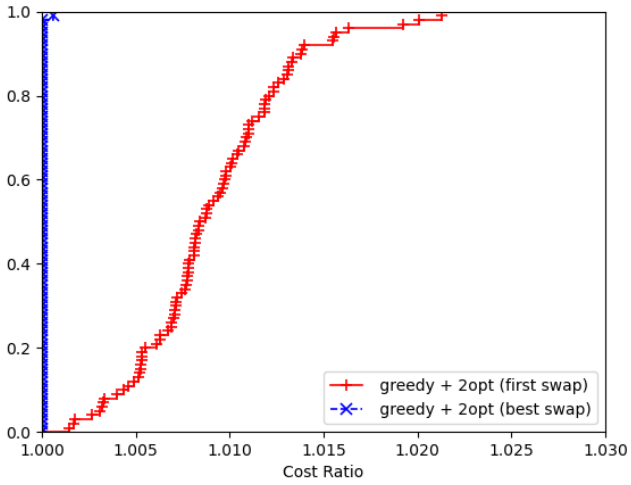
return cycle, cost

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 consistently finds better solutions than the first swap policy:

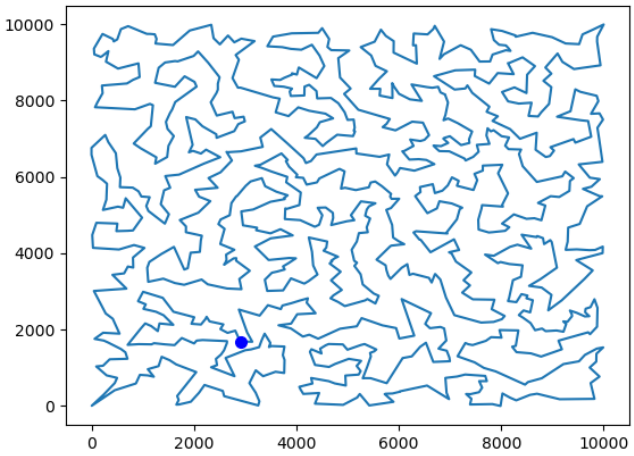


100 instances, Time limit: 120s

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 in application where the time limit is strict, the best swap policy is to be preferred since the big improvements will happen in the first iterations.

3.2.3 Results analysis

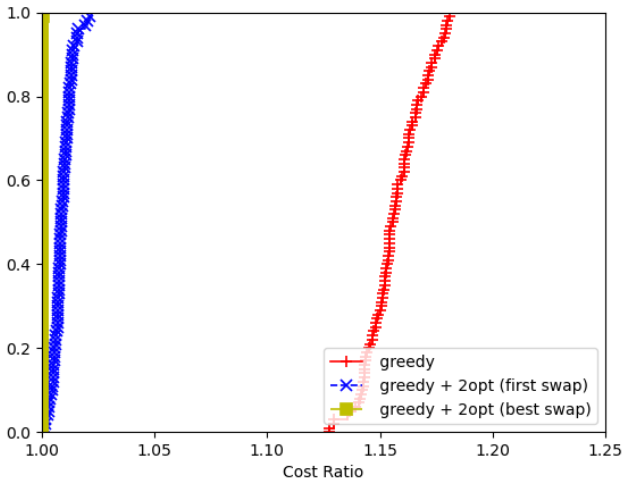
The 2opt algorithm is usually used after finding a cycle with the greedy algorithm to see wether it can be improved.
The solution displayed is obtained by running the 2opt algorithm, after applying the greedy algorithm on each possible starting node, and keeping the best one.



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 algorithms



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

4.1 Tabu search

...

4.2 Variable Neighborhood search (VNS)

...

Chapter 5

Exact methods

Chapter 6

Bibliography