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 (TSP), is one of the most famous and studied optimization problems in the Operations Research field.

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.

Let's 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.

This problem can be formulated through an Integer Linear Programming (ILP) model.

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 each node has a degree of 2 ($|\delta(v)| = 2, \forall v \in V$). 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

All the algorithms written in this thesis (to solve the TSP) where written using the C language, both due to its speed and CPLEX support for the language.

Plotting and testing are done with Python3, due to it's extensive plotting libraries and the ease in writing it.

2.1 Generating the instances

The instances used to test the algorithms have been randomly generated.

The problem space is a 2D grid: $[0, 10\_000] \times [0, 10\_000] \subset \mathbb{R}^2$.

Each point is generated from an i.i.d. uniform distribution of the grid defined earlier.

2.2 Installing CPLEX

To view the exact solution of our problem (optimal solution), we opted for (IBM ILOG) CPLEX as a MIP (Mixed Integer Programming) solver.

To get cplex visit IBM's web page and look for the CPLEX Optimization Studio.

TODO: maybe add something else?

2.3 Reading the solution plots

Each solution found is accompanied by a plot showing (graphically) the solution with some info.

Here's a list of the explanation of the info showed in the plots:

1. The name of the algorithm used:

greedy	:	Nearest Neighborg greedy algorithm
g2opt	:	applying 2opt to the greedy solution
tabu	:	Tabu algorithm

vns	: VNS algorithm
cplex	: CPLEX exact algorithm

2. In the same line of the algorithm there might be written some parameters used:

g2opt	: (f) → g2opt with first swap policy
g2opt	: (b) → g2opt with best swap policy
tabu	: int-int-double → tabu with fixed_tenure - variable_tenure - variability_frequency parameters
cplex	: benders loop → using benders loop
cplex	: mipst → giving cplex a "warm" start
cplex	: ccb → using the candidate callback
cplex	: rcb → using the relaxation callback
cplex	: n/g-patch → using patching if the solution provided is disconnected
cplex	: ccb/rcb-patch → using patching inside the (respective) callbacks

   (each parameter will be explained in the respective section).
   If the algorithm exceeded the time limit or has been terminated early by the user, an asterisk (*) will be shown at the end of the first line

3. cost: the cost of the Hamiltonian Cycle found

4. double/double: time needed to find that solution / total computing time

2.4 Performance profiler

To compare different algorithms we used the performance profiles, a tool provied by our professor that plots how often an algorithm is within a percentage of the cost from the best solution found

TODO: Explain better how to read the plot

Chapter 3

Heuristics

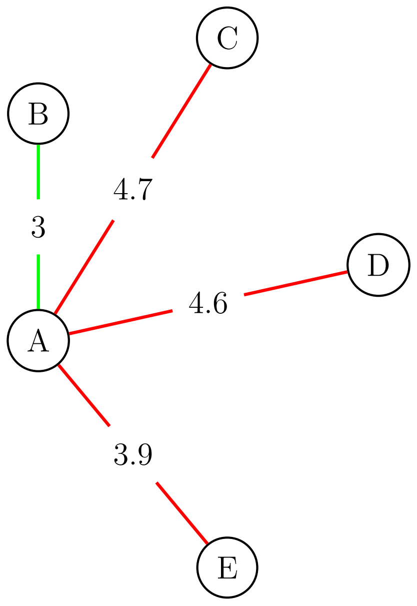
An heuristic is any approach to problem solving that employs a practical method that's not fully optimized but it's sufficient to reach an immediate short-term goal or approximation.

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)

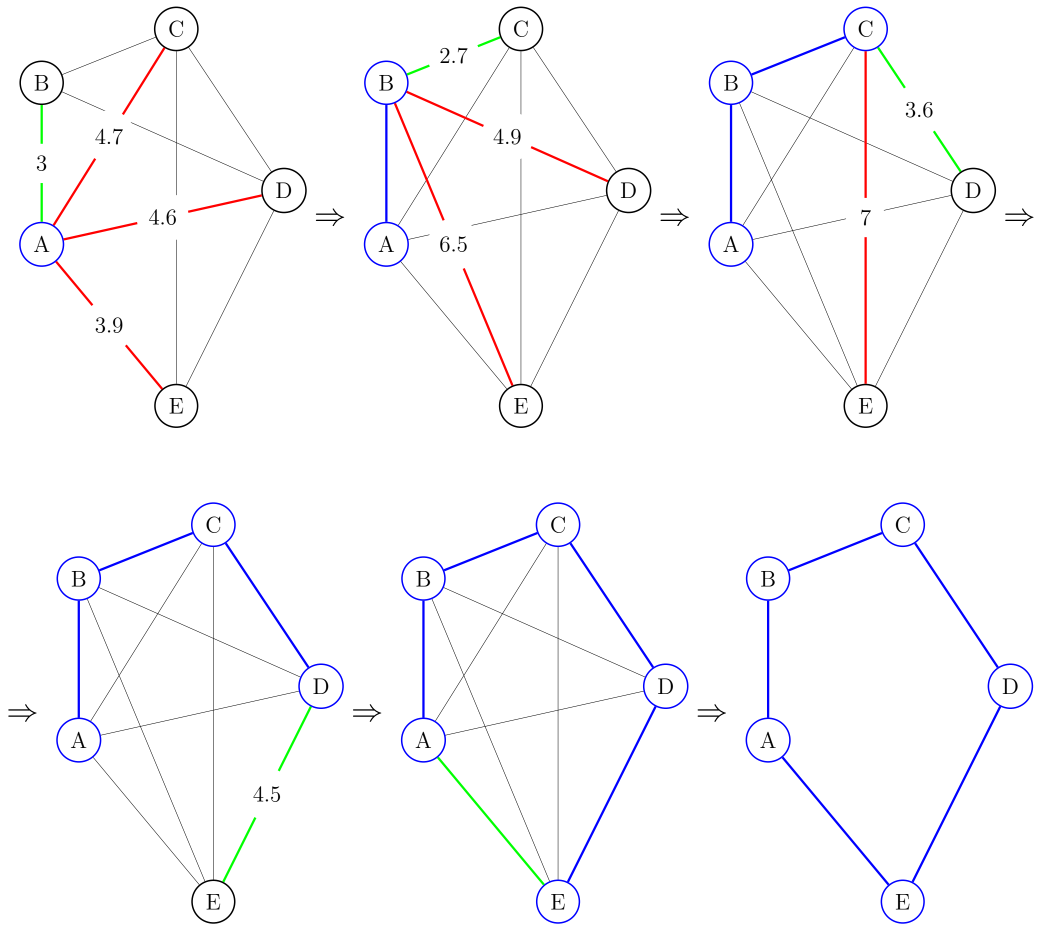
A first approach to the TSP, is to consider the edges with the smallest weights first when choosing the next node in the cycle.

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

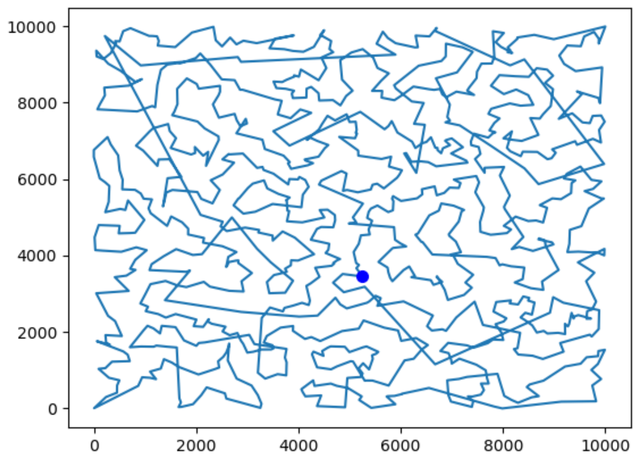
```
    Input Starting node ($s \in V$), Set of nodes ($V$)
    Output List of $n := |V|$ nodes forming an Hamiltonian cycle, Cost of the cycle
cycle ← [$s$]
cost ← 0
for $i = 0$ to $n - 2$ do
    next ← $\text{argmin}_v \{c_{cycle[i],v} \mid v \in V \wedge v \notin \text{cycle}\}$
    cost ← cost +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

TODO: new plot.



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.

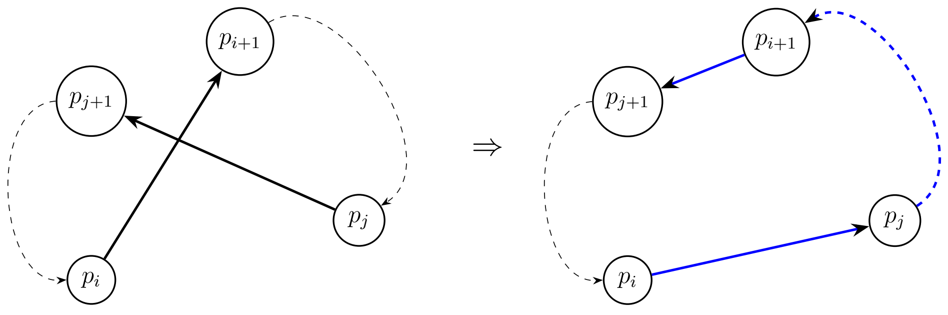
This is caused by the fact that the greedy algorithm optimizes locally, without knowing wether that local choice is good or not in the long term: it might happen that after adding some edges to the cycle, the closest nodes are all already in the cycle, so the edge that will be considered may have a very 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:



$p_i \coloneqq \text{cycle}[i]$

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:

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

3.2.1 Pseudocode

Algorithm 2 2-opt 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 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
```

Where the find_swap method finds a swap that improves the cost.

3.2.2 Swap policy

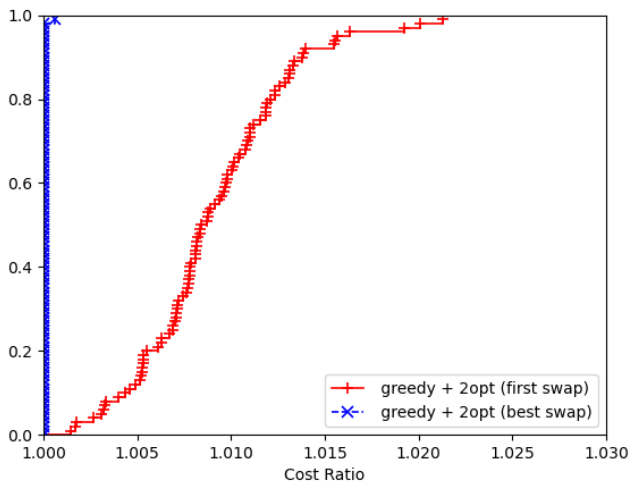
In this paper two different ways of finding a swap have been confronted:

- 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:

TODO: new plot



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.

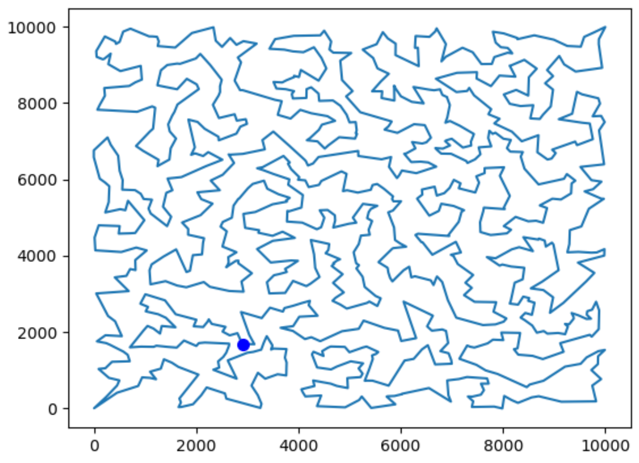
It's also worth noticing that the the first swap policy requires less time to improve the solution

3.2.3 Results analysis

TODO: new plot

The 2-opt 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 2-opt algorithm, after applying the greedy algorithm on each possible starting node, and keeping the best one.



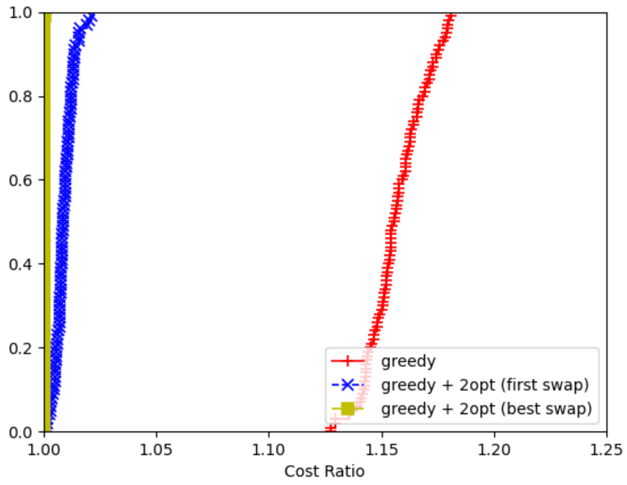
Instance: **A**, Algorithm: **greedy + 2-opt (best swap)**, Cost: **244793.5477**

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 2-opt 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 / 2-opt

TODO: new plot



100 instances, Time Limit: 120s

The 2-opt algorithm manages to consistently improve the cost of the solutions of 15-

20% with respect to the solutions found by the greedy algorithm.

TODO: Add cost comparison with exact algorithms.

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 2-opt 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 2-opt 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 with a lower cost.

Allowing a bad move, means that at the next iteration the new best move will revert it, 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

A tabu move is intended as the worsening move that has been done in a previous round, and it basically consists on the 4 edges, or the 2 nodes, that were considered in the swap.

To store the tabu move we have more options:

1. Mark as tabu one of the nodes (fix the two edges connected to that node)

2. Mark as tabu both nodes (fix all four edges in the swap)

3. Mark as tabu one edge or more edges

Marking one or both nodes as tabu moves would restrict our area of search, since after a few tabu moves, lots of edges cannot be changed, so we opted to mark as a tabu move the two edges (p_i, p_{i+1}) and (p_j, p_{j+1}).

4.1.2 The tabu list

The tabu list is intended as the list of tabu moves, that 2-opt will need to consult to see wether a move is admitted or not.

Once the tabu list is filled up, the oldest tabu move will be removed to let the new tabu move be saved.

An important factor regarding the tabu list, is its size: a small size means that the algorithm is not free to explore much the search space, while a big size means that the algorithm will worsen too much the solution, risking in not be able to ever find a better solution.

Another way to look at the size of the tabu list is to consider it as its memory: a small tabu list will forget earlier tabu moves, while a big tabu list will have a longer memory.

The size (or memory) of the tabu list will be referred as the tenure of the tabu list.

We built the tabu list as a fixed length array, where we stored each tabu move together with a counter which increases at each new tabu move.

We used the counter to view if a move is still a tabu move or not: by comparing the current counter with the one stored along the move, we can check how many iterations has passed and if more iterations than the tenure has occurred, that moves is no longer a tabu move.

We tried out two ways of checking the tenure:

1. Static approach: a move in the list is no longer a tabu move if

$$\text{counter} - \text{counter(move)} < \text{tenure}$$

   Where counter(move) is the counter stored along the move in the tabu list.

2. Dinamic approach: a move in the list is no longer a tabu move if

$$\text{counter} - \text{counter(move)} < f(\text{tenure}, \text{counter})$$

   Where $f(\text{tenure}, \text{counter}) := \text{A} \cdot \sin(\text{counter} \cdot \text{B}) + \text{tenure}$, and A, B are parameters given by the user (referred as variable_tenure and variability_frequency).

Here are shown two graphs, plotting the iteration counter and the cost of the solution the algorithm finds itself at:

TODO: Plot cost of tabu.

The dinamic approach performs better for a few reasons:

1. Lowers the risk of getting stuck: with the dinamic approach if the algorithm get stuck, in a few iteration it will start to forget some moves and escapes from that situation

2. Allows for a more dinamic exploration of the search space: the dinamic approach allows the algorithm to "forget" something to look for a better solution in the search space, but the remember it later if that lead to nothing.

4.1.3 Pseudocode

Algorithm 3 tabu 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, cost $\leftarrow$ greedy($s, V$)

while *Within the time limit* do

    $(i, j) \leftarrow$ find_tabu_swap(cycle)
    add_to_tabu(i, j)
    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
```

Where the find_tabu_swap() method returns the best swap (allowing for bad moves) after checking the tabu list.

4.1.4 Results analysis

TODO: Comparison with 2-opt

TODO: Comparison between dinamic / static approach

4.2 Variable Neighborhood Search (VNS)

For the tabu algorithm to work, a list of moves must be stored as tabu moves: how many moves to store? Is it better a static or dinamic tenure? In the dinamic approach, how much should the tenure vary? Those are all hyperparameters that should be set through lots of try and error generating the risk of overfitting.

Another approach to metaheuristics that doesn't require hyperparameters, is the Variable Neighborhood Search (VNS), which has the same base idea of the tabu search, but approaches to it differently.

Once we are in a local minimum, if we make a 2-opt swap (as we do with the tabu algorithm), we must save that move as a tabu move, since the next 2-opt swap will revert it.

The VNS approach is to make a swap that requires more than two edges to be swapped (entering the family of k-opt), in our case a 3-opt swap.

Once a 3-opt swap is performed, it's impossible for a 2-opt swap to reverse that change, since it should change 3 edges and it's allowed to change only 2 of them.

In some scenarios one 3-opt swap is enough to escape the local minimum, but in other it's not and more swaps are needed (an hyperparameter).

To prevent having to set an hyperparameter, we used multithreading to perform dirrerent numbers of 3-opt swaps on a local minimum, then use the 2-opt algorithm to lower the cost, and choose the best among the solutions found.

Another approach that can be explored in the future is using multithreading to keep the k best choices among the solutions found and keep exploring them in parallel (with special attention to keep the list of parallel runs under control, avoiding exponential growth).

4.2.1 Results analysis

TODO: Comparison with 2-opt

4.3 Comparison Tabu / VNS

TODO: Comparison with tabu

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 4 Benders' loop

```
  Input undirected complete graph $G = (V, E)$, cost function $c : V \to \mathbb{R}$
  Output List of $n \coloneqq |V|$ 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: $\sum_{e \in \delta(S)} x_e \le |S| - 1$
    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 5 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

Matheuristics

Up to this point we have explored two different approaches to solving the TSP problem that can be considered the extremes of a scale. On one hand we have several heuristic algorithms that can quickly solve instances of several tens of thousands of nodes in a few minutes, but with little guarantee of finding an optimal solution; on the other hand we have exact algorithms based on cplex, which can find optimal solutions for way smaller instances of a few hundreds of nodes.

To solve instances with around 1,000 nodes we use a series of hybrid methods called *matheuristics*. These methods take a closed-box MIP solver, that is guaranteed to find an optimal solution to the model they receive in input, and we use it as a heuristic; this is achieved by giving as input a restricted version of the original model, from which an arbitrary set of feasible solutions has been excluded. This way, we still exploit the power of the MIP solver, while restricting the space of possible solutions, thus reducing the required time to solve the model.

6.1 Diving

This matheuristic is based on iteratively solving the TSP model and restricting it by taking the best solution found up to a certain iteration and *hard fixing* some of its edges, using a heuristic solution as the starting incumbent. These edges are called *'yes' edges* and they are fixed by setting the values of their respective variables in the model to 1. This method is called *diving* because fixing a series of variables is analogous to reaching a certain depth of the branch & bound tree in a single iteration.

There are several possible approaches to decide how many and which edges to fix. In this thesis we choose them in a completely random way, generating a random number in the range $[0, 1]$ and comparing it with a constant fixing probability. While it is the simplest possible approach, it has the advantage of having a very small probability of getting stuck on a certain neighbourhood of solutions.

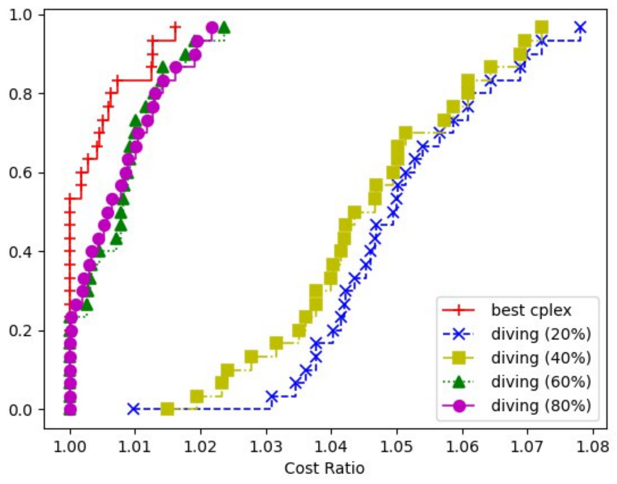
6.1.1 Pseudocode

Algorithm 6 Diving matheuristic algorithm

```
**Input** undirected complete graph $G = (V, E)$, cost function $c : V \to \mathbb{R}$, $pfix$ edge fixing probability
**Output** List of $n := |V|$ nodes forming an Hamiltonian cycle, cost of the cycle
  $x^H \leftarrow$ heuristic solution of TSP on G
  build CPLEX model from $G$ with objective function and degree constraints
  **while** timelimit not exceeded **do**
      $\tilde{E} \leftarrow$ subset of $E$ with $V| * pfix$ edges of $x^H$ chosen at random with probability $pfix$
      **for each** $x_e^H \in \tilde{E}$ **do**
          $x_e^H \leftarrow 1$ in the CPLEX model
      **end for**
      $x^* \leftarrow$ solution returned by CPLEX for input model with fixed edges
      **if** cost$(x^*) \leq$ cost$(x^H)$ **then**
          $x^H \leftarrow x^*$
      **end if**
      **for each** $x_e^H \in \tilde{E}$ **do**
          $x_e^H \leftarrow 0$ in the CPLEX model
      **end for**
  **end while**
```

6.1.2 Hyperparameter tuning

$pfix$ is a hyperparameter of the algorithm, thus we tested different values for it.



30 instances, 1000 nodes, time limit: 120s

The diving algorithm returns results with costs similar to our best version of the CPLEX solver. However, there is a significant improvement for high value of $pfix$.

6.2 Local Branching

In the diving algorithm, we decided both how many and which variables to fix in the mathematical problem before using the cplex solver. In the *local branching* algorithm, instead, we choose only how many variables we want to fix, leaving to the TSP solver the responsibility of choosing which ones to fix. This is expressed in the model by adding an additional constraint.

Given x^H the current incumbent, we define the *local branching constraint* as:

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

The left-hand sum is the number of variables we want the TSP solver to keep from x^H, while k is the number of variables we want the TSP solver to fix. By setting the direction of the constraint as $\geq$, the solver explores a neighbourhood of different solutions that can be reached by changing $n-k$ variables; as such, the right-hand side quantity $n-k$ represents the number of degrees of freedom given to the solver.

This constraint is not guaranteed to be valid for the set of feasible solution, because it might cut out the optimal solution; however, it allows us to greatly reduce the integrality gap.

6.2.1 Pseudocode

Algorithm 7 Local branching matheuristic algorithm

```
  **Input** undirected complete graph $G = (V, E)$, cost function $c : V \to \mathbb{R}$, integer $k_{\text{init}}$
  **Output** List of $n := |V|$ nodes forming an Hamiltonian cycle, cost of the cycle
$x^H \leftarrow$ heuristic solution of TSP on G
build CPLEX model from $G$ with objective function and degree constraints
$k \leftarrow k_{\text{init}}$
**while** timelimit not exceeded **do**
    add constraint local branching constraint $\sum_{e:x_e^H=1} x_e \geq n - k$
    $x^* \leftarrow$ solution returned by CPLEX for input model with local branching constraint
    **if** cost($x^*$) $\leq$ cost($x^H$) **then**
        $x^H \leftarrow x^*$
    **end if**
    **if** $x^H$ has not been improved for 5 iterations **then**
        $k \leftarrow k + 10$
    **end if**
**end while**
```

6.2.2 Hyperparameter tuning

In this algorithm, k_{init} is the hyperparameter. We tried both setting an arbitrary value and estimating it from x^H.

TODO: plot

Chapter 7

Bibliography