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:
Luca Fantin (2119287)
Matteo Zanella (2122187)

Academic 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

Let us 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 We 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 us 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: $\text{cost(cycle)} \coloneqq \sum_{e \in \text{cycle}} c(e)$.

This problem can be formulated through an *Integer Linear Programming (ILP)* model. First, let us 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 several isolated cycles. This issue is solved by constraints 1.3, called *Subtour Elimination Constraints (SEC)*, which 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, they are defined for every subset of nodes including v_1, which exist in exponential number in N. Including all of them at once is therefore computationally infeasible.

All pseudocodes presented in this thesis will use the aforementioned undirected complete graph $G = (V, E)$ and cost function $c : E \to \mathbb{R}$.

Chapter 2

Project Setup

Our project is composed of a C program implementing a number of different TSP algorithms. We chose the C language both because of its speed and of the support offered for the software used to compute exact solutions for the TSP, IBM ILOG *CPLEX*.

While the main program has been written in C, some functionalities were written in Python, in particular plotting the TSP solutions and automating tests. The language was chosen due to its extensive plotting libraries and its simplicity; such features proved to be very useful to write an interface to execute the C main program.

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

CPLEX is the *MIP (Mixed Integer Programming)* solver we chose for the project. It can be downloaded and installed from the IBM SkillsBuild downloads page. CPLEX comes with an app with its own UI, but for this project we will only use its C library.

2.3 Performance profiles

To compare different algorithms we used the *performance profiler* [1], a tool for benchmarking and comparing different optimization software and algorithms.

They were generated by a Python script developed by Domenico Salvagnin. In this thesis we used this tool to compare a certain performance metric, either the cost of the best solution found or the time taken to find such solution, for different algorithms taken from a set $\mathcal{A}$ executed on the same set of instances $\mathcal{I}$ of the same size but generated with different random seeds.

The Python script takes as input a table of the values of the performance metric $m(a, i)$ for every combination of algorithm $a \in \mathcal{A}$ and test instance $i \in \mathcal{I}$. $\forall i \in \mathcal{I}$ the script takes the best value of the metric across all algorithms $m^*(i) \coloneqq \max_{a \in \mathcal{A}}\{m(a, i)\}$ and computes $m_{\%}(a, i) = m(a, i)/m^*(i)\ \forall a$. With these measures, $\forall i \in \mathcal{I}$ we can measure the gap between the value of $m(a, i)$ returned by the considered algorithms $a \in \mathcal{A}$ and the best value $m^*(i)$ among those as a percentage.

These values are computed for all algorithms and instances, and with these the performance profile is drawn. It appears as a graph with a line $\forall a \in \mathcal{A}$ formed by points where the abscissa is a value x of $m_{\%}(a, i)$ and the ordinate is the percentage of instances $i \in \mathcal{I}$ for which $m_{\%}(a, i) \leq x$. Thus, this graph represents how often the considered algorithms perform within a certain percentage from the best one.

Chapter 3

Heuristics

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

Given the NP-Hard nature of the Travelling Salesman Problem, finding the optimal solution might require lots of time, hence the need to have fast heuristic methods to find solutions that are close to the optimal.

3.1 Nearest Neighbor (Greedy)

A first approach to the TSP is to iteratively build a solution by starting from a certain node and considering 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 seeks the locally optimal choice at each stage.

The solution found using the greedy algorithm is dependent on 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.

3.1.1 Pseudocode

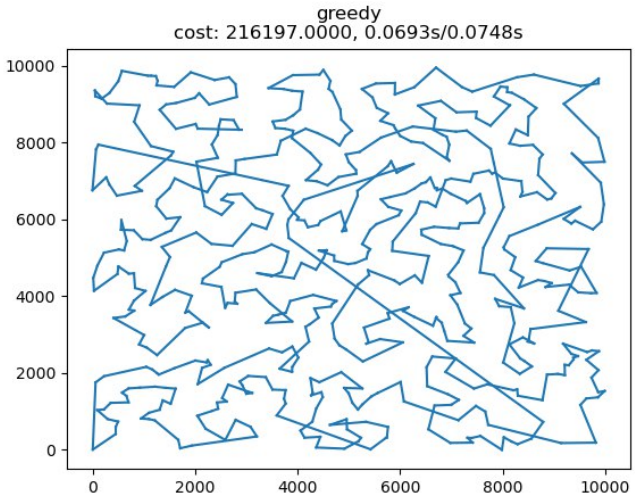
Algorithm 1 TSP greedy algorithm

Input starting node $s \in V$
Output Hamiltonian cycle of V, cost of cycle

cycle $\leftarrow [s]$
cost $\leftarrow 0$
for $i = 0$ to $|V| - 2$ **do**
    next $\leftarrow \text{argmin}_v\{c(\text{cycle}[i], v) : v \in V \setminus \text{cycle}\}$
    cost $\leftarrow$ cost $+ c(\text{cycle}[i], \text{next})$
    cycle$[i+1] \leftarrow$ next
end for
cost $\leftarrow$ cost $+ c(\text{cycle}[|V|-1], s)$

return cycle, cost

3.1.2 Results analysis



The solutions found with the greedy algorithm are a good starting point, but sometimes the algorithm picks edges that cross a long distance, increasing the cost of the solution considerably.

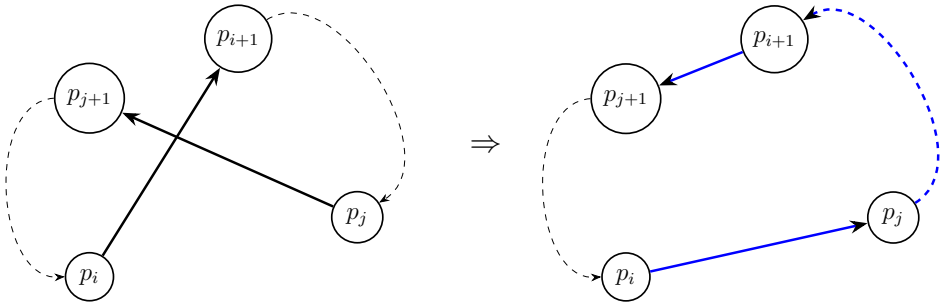
This is caused by the fact that the greedy algorithm optimizes locally, without knowing whether that local choice is good or not in the long term: it might happen that after adding some edges to the cycle, the closest nodes are all already in the cycle, so the edge that will be considered may have a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

In the next section a tecnique to fix this problem will be explored.

3.2 2-opt

The 2-opt algorithm takes an existing cycle and tries to improve its cost by changing some of the edges that compose the cycle without breaking it.

In general, we can define a *k-opt* solution for the TSP problem as a solution obtained by changing k edges in an existing tour. In this context, we implement a 2-opt algorithm that identifies and swaps two edges to improve the tour's overall cost: in most cases this means untangling a crossover between two edges.



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

If such pair of edges exists, then it 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 $i > j$ will not 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.

This process is then repeated until no more edges that can lead to an improvement to the cost of the cycle can be found.

3.2.1 Pseudocode

Algorithm 2 TSP 2-opt algorithm

Input Hamiltonian cycle of V, cost of cycle
Output Hamiltonian cycle of V, cost of cycle

while *a swap improving the cost exists* **do**
  $(i, j) \leftarrow$ *find viable swap in the 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 section of cycle between indices $i + 1$ and j
end while

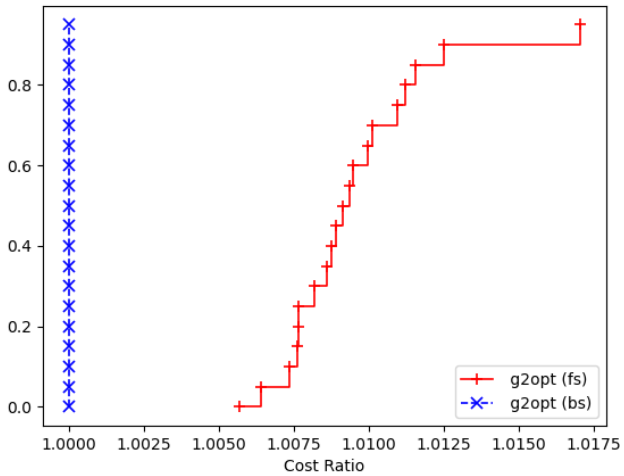
return cycle, cost

3.2.2 Swap policy

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

- returning the *first swap* found that improves the cost (referred as g2opt fs)
- looking among all possible pair of edges and returning the swap that yields the largest cost improvement (the *best swap*, referred as g2opt bs)

Using the performance profiler it is easy to see that the best swap policy finds solutions with an improvement of a 1% factor with respect to the first swap policy:



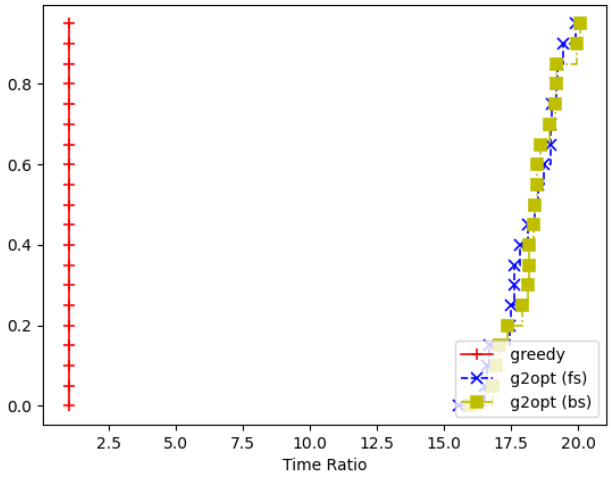
2
-opt policies]20 instances, size 600, Time limit: 120s

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.

3.2.3 A faster implementation

The 2-opt algorithm is a good asset since it improves the solutions of a factor of 15% with respect to the greedy solution (we'll look into that in the next section), but it requires much more time, even using multithreading:



G
reedy / greedy+2-opt comparison]20 instances, size 600, Time limit: 120s

As we can see, the 2-opt algorithm takes 15-20 times as much to reach these improvements: this slowdown gets worse with bigger problems due to the exponential time required to solve it.

The time needed by the 2-opt algorithm to fix the greedy solutions depends on the number of nodes, as stated before, but also on the quality of the solution that is being improved, so if we can somehow improve the solution before using the 2-opt algorithm, the time required to complete it will be much lower.

The idea is to use a Divide and Conquer approach, applying the 2-opt algorithm to 2 sub-problems and then applying it to the union of the solutions found.

Algorithm 3 TSP f2opt algorithm

Input Hamiltonian cycle of V, cost of cycle
Output Hamiltonian cycle of V, cost of cycle

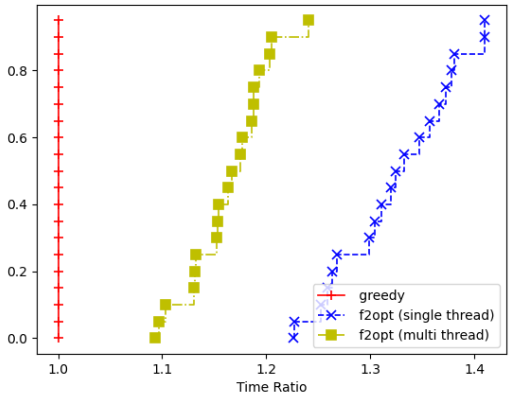
if *reached max recursion depth* **then**
    (cycle, cost) ← *apply 2-opt to the cycle*
    return cycle, cost
end if

while *not reached max recursion depth* **do**
    (left, right) ← *split the cycle in two parts*
    (left, left_cost) ← *apply the f2opt algorithm to the "left" part of the cycle*
    (right, right_cost) ← *apply the f2opt algorithm to the "right" part of the cycle*
    (cycle, cost) ← *join (left, right) back into one cycle*
    (cycle, cost) ← *apply 2-opt to the cycle*
end while

return cycle, cost

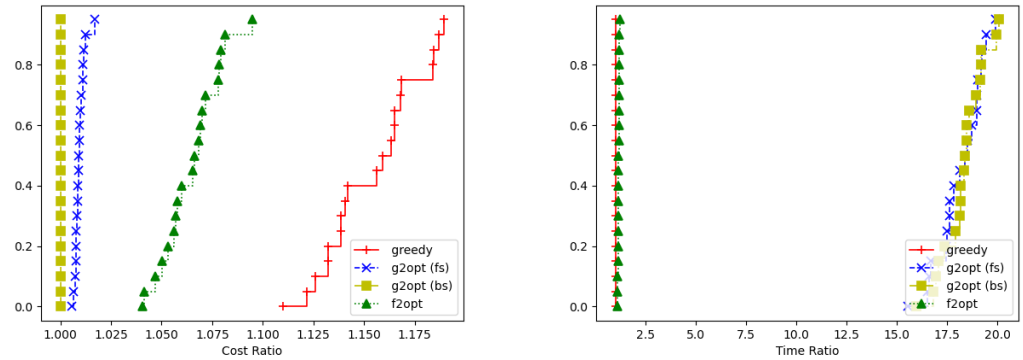
The splitting procedure used consists in dividing the set of nodes in two parts based on their coordinates, so that in each of the two subsets obtained the nodes are close to each other; the merging procedure consists just in gluing at random the two cycles together: trying some fancy ways of merging the two cycles would lead to a waste of computational resources since the 2-opt algorithm will be applied to the resulting cycle, probably re-arranging the edges made to glue the two cycles together.

This implementation, called *f2opt* from now on, is faster since it applies the 2-opt algorithm only to the base cases and to cycles that have lots of edges already in the right place, so the number of swaps needed to end the 2-opt algorithm is significantly lower: the slowdown goes from 20 down to 1.4, with respect to the greedy algorithm (down to 1.2 if using multithreading).



G
reedy / f2opt comparison]20 instances, size 600, Time limit: 120s

3.3 Comparison Greedy / 2-opt

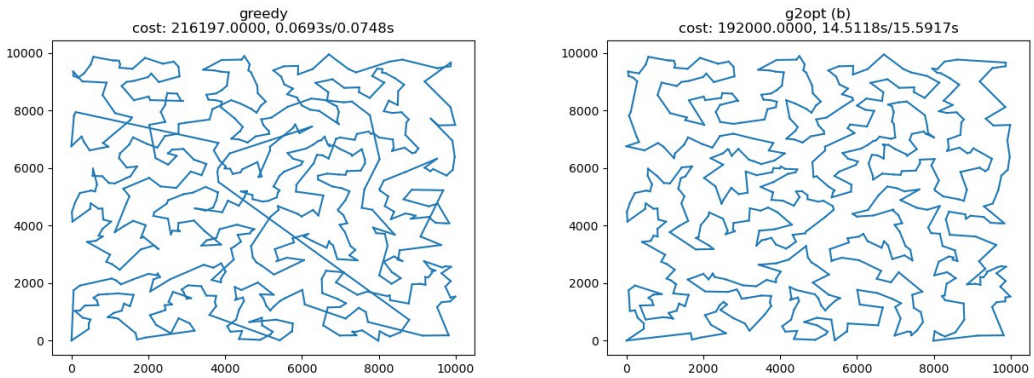


20 instances, 600 nodes, time limit: 120s

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, while the f2opt algorithm reaches an improvement of 6-8% whith a fraction of time needed by the basic 2-opt algorithm.

From these results it's clear that the normal implementation of the 2-opt algorithm has better results: the f2opt algorithm has been created for instances which are too large for the 2-opt algorithm to handle.

The f2opt algorithm will be further used in this paper for matheuristics approach, where we try to solve larger instances, letting us save time in the initial phases of the algorithm so that our CPLEX solver has the most time to operate.



20 instances, 600 nodes, time limit: 120s

As demonstrated, the 2-opt algorithm eliminates crossing paths, resulting in a more optimal and visually coherent solution.

Chapter 4

Metaheuristic

A *metaheuristic* is an abstract approach that can be applied to a range of problems, which can then be specialized: in our case we used the greedy solutions polished with the 2-opt algorithm as a starting point, and then further process those solutions with one of two metaheuristic methods, the tabu search and the Variable Neighborhood Search. Both unlock a greater search space compared to the aforementioned heuristic methods by allowing "bad moves" to escape a locally optimal solution.

4.1 Tabu search

The *tabu search* algorithm [2] is based on the idea of allowing the 2-opt algorithm to perform swaps that still are the best ones, but not necessarily swaps that improve the cost of the solution. Once we find a local optimum, the tabu search algorithm will keep searching, moving away from that locally optimal solution hoping to find a new one with a lower cost, whereas the 2-opt algorithm would stop.

If we allowed a bad move, at the next iteration the best move found by the 2-opt procedure would 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 swap that has been done in a previous round, and it basically consists on the 2 edges, or equivalently the 4 nodes, that were considered in the swap.

To store the tabu move we have more options on what to mark as a tabu move:

1. one of the nodes (fix the two edges connected to that node)
2. both nodes (fix all four edges in the swap)
3. one 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 the 2-opt algorithm will need to consult to see whether a swap is admitted or not. Once the tabu list is filled up, the oldest tabu move will be removed to let the one to be saved.

An important parameter of the tabu list is its size: a small size means that the algorithm is not very free to explore the search space, while a big size means that the algorithm will worsen the solution too much, possibly preventing it to ever find a better solution. This can also be intepreted as setting 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 hereby be referred as the its *tenure*. 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 see if a move is still tabu or not: if the move is present in the tabu list, by comparing the current counter with the one stored along the move, we can check how many iterations has passed since it has been marked as tabu, and if more iterations than the tenure has occurred, that move is no longer tabu.

We tried out two ways of checking the tenure:

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

$$\text{counter} - \text{move.counter} < \text{tenure}$$

   where move.counter is the counter stored along the move in the tabu list.

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

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

   where $f(\text{tenure}, \text{counter}) = A \cdot \sin(\text{counter} \cdot B) + \text{tenure}$, and A, B are parameters specified by the user (A will be referred as tenure_variability and B as tenure_frequency).

We expect the dynamic approach to perform better for a few reasons:

1. it lowers the risk of remaining stuck for many iterations: when we reach a local optimum, with the dynamic approach the algorithm will start to forget some moves in a few iterations and will escape from that situation;

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

4.1.3 Pseudocode

Algorithm 4 TSP tabu search algorithm

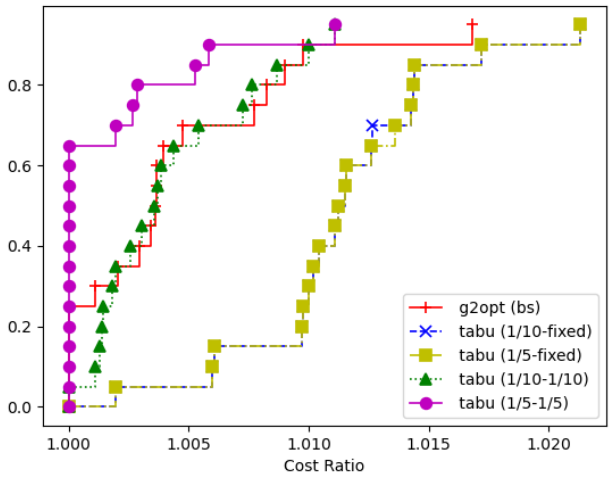
Input starting node $s \in V$
Output Hamiltonian cycle of V, cost of cycle

(cycle, cost) ← *result of greedy algorithm on s and V*

while *time limit not exceeded* **do**
  (i, j) ← *swap to be applied on cycle (checking the tabu list)*
  if *swap (i,j) leads to a worse position* **then**
    add (i, j) to tabu list
  end if
  update cost
  reverse section of cycle between indices $i + 1$ and j
end while

return cycle, cost

4.1.4 Results analysis



20 instances, 600 nodes, time limit: 120s

This plot reports various executions of the tabu algorithm with different parameters: the first two use a tenure of 1/10 and 1/5 of the number of nodes while using the static tenure approach; the last two use the dynamic approach insthead, whith a tenure_variability of 1/10 and 1/5 of the number of nodes.

As we expected, the dynamic approach yields slightly better results than the static approach: it is important to point out that the latter generates worse solutions than the 2-opt algorithm; this is due to the fact that the 2-opt algorithm looks among all possible greedy solutions based on the starting node, while the tabu algorithm only uses

one starting point to explore the search space (or a fixed amount if using multithreading), making it possible that it doesn't start from the best 2-opt solution.

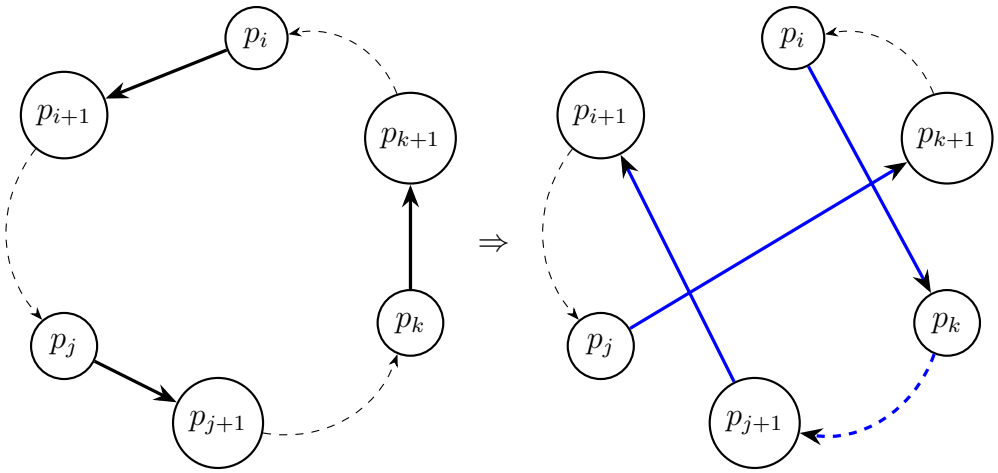
We can see how the dynamic approach manages to surpass this limitation and improve the solutions with respect to the 2-opt algorithm.

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? Which one works better: a static or a dynamic tenure? And in the latter, how much should the tenure vary? This set of hyperparameters should be set with procedures that are unaffordable for the scope of this thesis. We may also run into overfitting.

A metaheuristic method that does not require hyperparameters is the *Variable Neighborhood Search (VNS)* [3], which approaches the same base idea of tabu search in a different way. Once we are in a local minimum, if we make a 2-opt swap (as we do with the tabu search 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). Once a k-opt swap is performed, it is impossible for a 2-opt swap to reverse that change.

The kicks we applied are 3-opt swaps using the following schema:



In some scenarios one 3-opt swap is enough to escape the local minimum, but in other it is not and more swaps are needed, thus creating a hyperparameter. To prevent this, we used multithreading to perform different 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 Pseudocode

Algorithm 5 TSP VNS algorithm

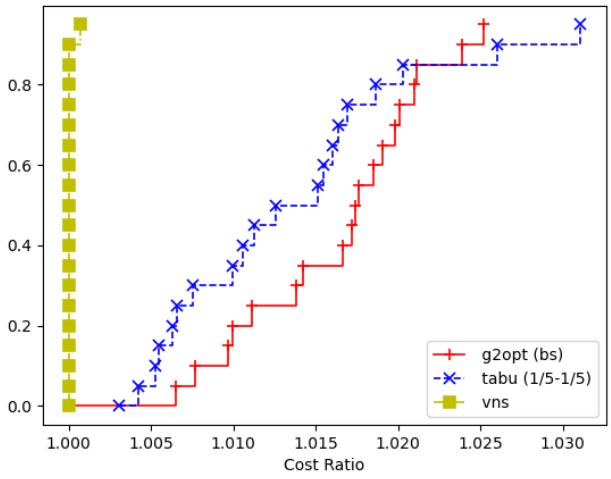
Input starting node $s \in V$
Output Hamiltonian cycle of V, cost of cycle

(cycle, cost) ← *result of greedy algorithm on s and V*

while *time limit not exceeded* **do**
    (cycle, cost) ← *apply random kicks to the cycle*
    (cycle, cost) ← *apply 2-opt to the cycle*
end while

return cycle, cost

4.3 Comparison tabu Search / VNS



20 instances, 600 nodes, time limit: 120s

As we can see, the VNS algorithm has consistently better performances than the tabu search algorith, up to a 3% improvement: this might be due to the fact that the tabu search algorithm requires a finer parameter tuning, while the VNS algorithm does not need any.

Chapter 5

Exact methods

After analyzing some heuristic and metaheuristic approaches, we started working on some exact algorithms, so that we can find exact solutions to our TSP instances.

The exact algorithms for the TSP algorithm described in this section are all based on CPLEX, which uses a proprietary implementation of the *branch & cut (B&C)* 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 6 Benders' loop

Input none
Output Hamiltonian cycle of V, cost of cycle

build CPLEX model from G with objective function and degree constraints

while *time limit not exceeded* **do**
  x^* ← *solution of CPLEX model*
  *determine connected components of x^{**}

  if *number of connected components == 1* **then**
    cycle, cost ← *output values computed from x^{**}
    return cycle, cost
  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

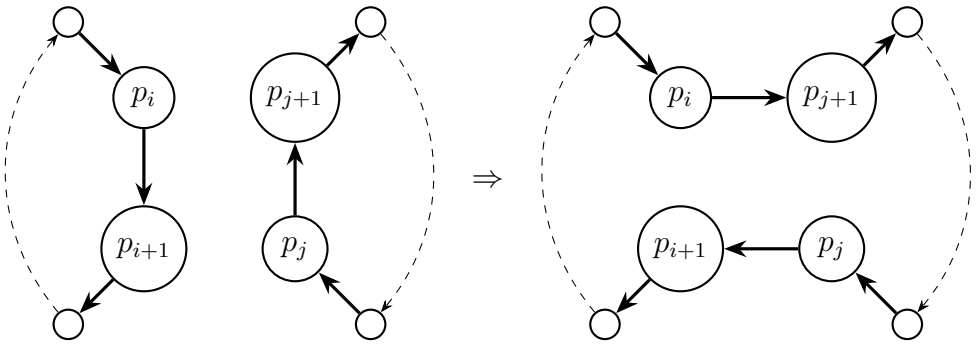
return $x^*, \text{cost}(x^*)$

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 $(p_i, p_{j+1}), (p_j, p_{i+1})$ and $(p_i, p_j), (p_{j+1}, p_{i+1})$. 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.

The scheme on the following page shows an example of the first type of swap we can perform.



5.2.1 Pseudocode

Algorithm 7 Patching heuristic

Input solution x^* returned by CPLEX with ncomp connected components
Output x^* with 1 connected component

```
while ncomp ≠ 1 do
    best_swap ← null, best_delta ← +∞
    for k1 ← 0 to ncomp − 1, k2 ← k1 + 1 to ncomp − 1 do
        for each (p_i, p_{i+1}) ∈ x* with p_i, p_{i+1} ∈ k1, (p_j, p_{j+1}) ∈ x* with p_j, p_{j+1} ∈ k2 do
            swap ← *swap patching components k1 and k2 involving (p_i, p_{i+1}), (p_j, p_{j+1})*
            delta ← *increase in cost of solution produced by performing swap*
            if delta < best_delta then
                best_swap ← swap
                best_delta ← delta
            end if
        end for
    end for
    *perform best_swap*
    *increase cost of x* by best_delta*
    ncomp ← ncomp − 1
end while

*perform 2-opt algorithm on x***
return x*
```

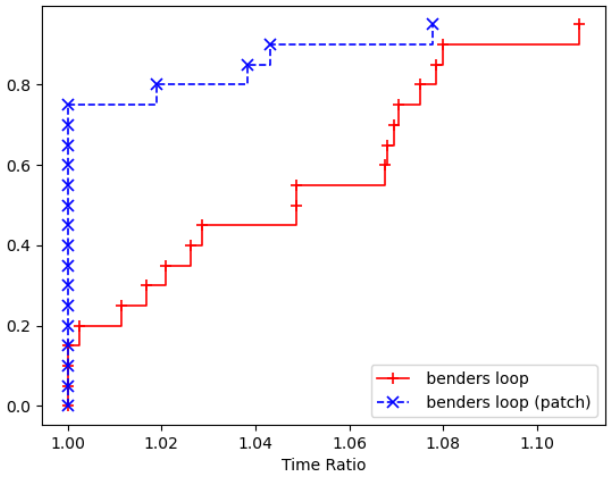
When choosing the swap in the innermost cycle, we choose the best option between $(p_i, p_{j+1}), (p_j, p_{i+1})$ and $(p_i, p_j), (p_{j+1}, p_{i+1})$.

5.2.2 Results analysis

We compared the results obtained by Benders' loop both with and without the patching heuristic. Within this timeframe the algorithms produced solutions with the same costs but took different amounts of time.

The patching heuristic allowed the algorithm to reach a feasible solution in less time. This is in line with the nature of Benders' loop: the base version of the algorithm produces

a feasible solution only at the very last iteration, an issue that can be avoided by patching the components after every iteration.



20 instances, 300 nodes, time limit: 360s

5.3 Inside CPLEX using callbacks

CPLEX lets us create functions to be used as *callbacks*. In some points of its execution it calls some undefined functions; we can set our own customized functions in those point to execute our code inside CPLEX in order to help it with our knowledge of the problem.

We will help CPLEX mainly by addding SEC to the model and *posting* some heuristic solutions to help him improve the upper bound. Posting heuristic solutions helps CPLEX by reducing the gap between the incumbent and the upper bound, making the B&C inside CPLEX faster.

5.3.1 Candidate callback

The *candidate callback* is called each time an integer solution is found. The solution might contain more than one connected component, since we never specified SEC in the model; we have to find those connected components and add a SEC for each one.

Once we add the SEC obtained by the candidate solution, we patch that solution using the patching algorithm shown in section 5.2 and post it as an heuristic solution.

Finally, we reject the candidate solution, making sure the incumbent is only updated with connected integer solutions.

5.3.2 Relaxation callback

While the candidate relaxation intercepts integer solutions, the *relaxation callback* intercepts solutions of the relaxation problem. Since these are solutions too, we might try

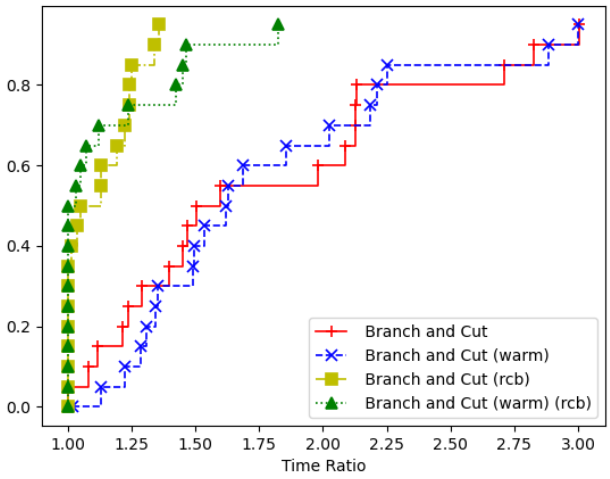
to create SEC out of those too.

To create SEC out of fractionary solutions we have to solve a *Network Flow problem*, for which we used the *Concorde* software, which is currently the best TSP solver commercially available.

Similarly to the candidate callback, we might want to post a patched version of the solution CPLEX gave us: to do so we need to create a new type of patching, which can handle fractionary solutions. Instead of working with the components at hand, we compute a whole new solution using a variant the greedy algorithm introduced in section 3.1. Instead of choosing the edges looking only at the cost, we use the product $c_e \cdot (1 - x_e^*)$. We consider x_e^* as a confidence value of how much that edge is good or bad for the solution: if $x_e^* == 1$ then CPLEX is "confident" about that edge being good, so the weighted cost will go to 0 and we will consider it in our patched solution; otherwise, if $x_e^* == 0$ then CPLEX choose not to include that edge, and so we will not give it a "discount" to the cost.

5.3.3 Results analysis

We tested these callbacks in an execution of the CPLEX solver without Benders' loop.



20 instances, 300 nodes, time limit: 360s

Note that:

- Branch and Cut: CPLEX + Candidate Callback

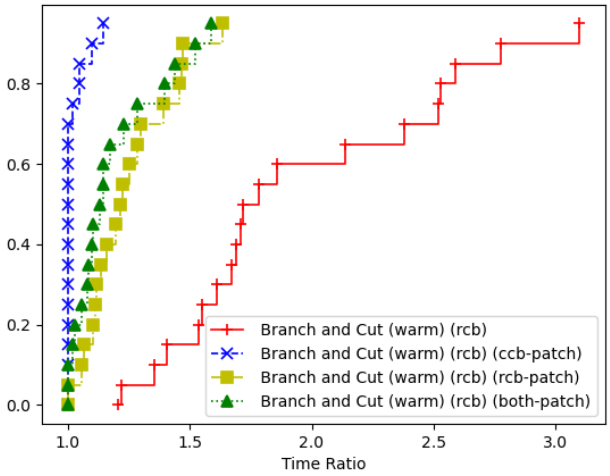
- (warm): giving CPLEX a warm start (posting an heuristic at start)

- (rcb): using the relaxation callback

The candidate callback is necessary to find a feasible solution without using Benders' Loop, so we might consider it as a baseline.

As we can see, adding the relaxation callback will give us a speedup up to 200-300% times with respect to the baseline.

Patching and posting the solutions found inside the callbacks will give us another speedup of 200-300%:



20 instances, 300 nodes, time limit: 360s

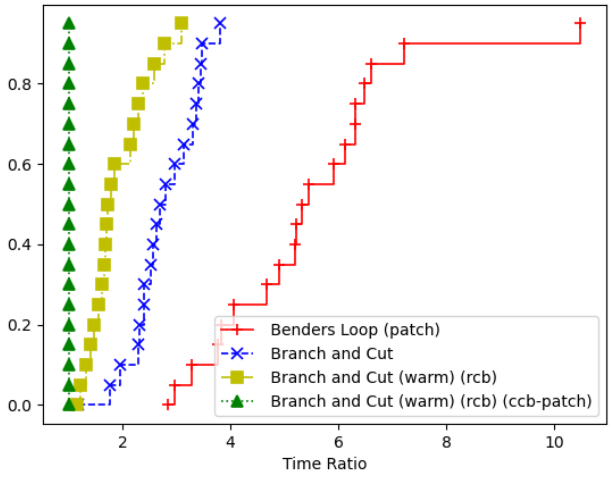
Note that:

- (ccb-patch): patching and posting in the candidate callback
- (rcb-patch): patching and posting in the relaxation callback
- (both-patch): patching and posting in both callbacks

As we can see, patching and posting inside the candidate callback alone gives us the best performances: patching and posting the solutions just in the relaxation callback, still gives us a great improvement, but apparently patching and posting in the candidate callback is enough to lower the upper bound enough to speed up CPLEX, and the time needed to patch inside the relaxation callback is just an overhead.

5.4 Comparison Benders / Callbacks

All those improvements lead us to an high performance sofftware to find the optimal solution of our TSP. Here is shown the overall progress with exact methods:



20 instances, 300 nodes, time limit: 360s

Our final software is 8-10 times faster than Benders' Loop: this is due to the fact that by operating inside CPLEX with the callback, we are making the changes "locally", while CPLEX has to recreate its decision tree at each iteration of Benders' Loop.

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 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 small instances.

To solve mid-sized instances (around 1,000 nodes) we use a series of hybrid methods called *matheuristics* [4]. 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. In 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. While it is the simplest possible approach, it has the advantage of having a very small probability of getting stuck on a certain neighborhood of solutions.

6.1.1 Pseudocode

Algorithm 8 Diving matheuristic algorithm

Input $pfix$ edge fixing probability
Output Hamiltonian cycle of V, cost of cycle

$x^H \leftarrow$ *heuristic solution of TSP on G*
build CPLEX model from G with objective function and degree constraints

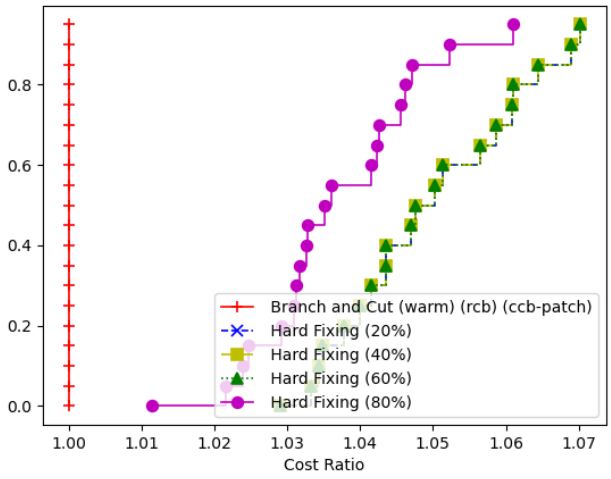
while *time limit not exceeded* **do**
  $\tilde{E} \leftarrow$ *subset of E with $|V| * pfix$ edges of x^H chosen at random with probability $pfix$*
  fix every $x_e^H \in \tilde{E}$ to 1 in model

  $x^* \leftarrow$ solution returned by CPLEX for input model with fixed edges
  if $\text{cost}(x^*) \leq \text{cost}(x^H)$ **then**
    $x^H \leftarrow x^*$
  end if

  unfix every $x_e^H \in \tilde{E}$ in model
end while

6.1.2 Hyperparameter tuning

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



20 instances, 1000 nodes, time limit: 120s

The diving algorithm yields worse solutions than our best CPLEX solver. As we saw, the major performance boost we had when testing the different CPLEX settings came from patching and posting solutions inside the callbacks. Since by patching we might generate solutions which would be rejected by the model with the fixed variables, much of this work is performed in vain; this might be the cause of this performance drop.

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 [5], 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 neighborhood of different solutions that can be reached by changing $n - k$ variables (looking for the best k-opt swap to apply).

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 9 Local branching matheuristic algorithm (v1)

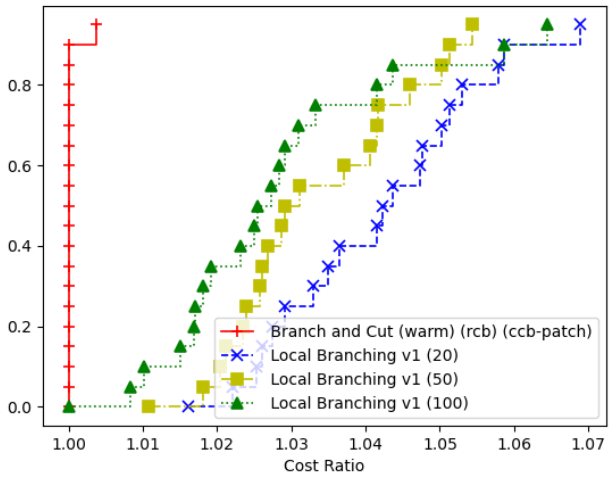
Input integer k_{init}
Output Hamiltonian cycle of V, cost of cycle

$x^H \leftarrow$ *heuristic solution of TSP on G*
build CPLEX model from G with objective function and degree constraints
$k \leftarrow k_{\text{init}}$
while *time limit 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
    save improvements
    remove local branching constraint
end while

return x^H, cost(x^H)

6.2.2 Hyperparameter tuning

In this algorithm, k_{init} is the hyperparameter.



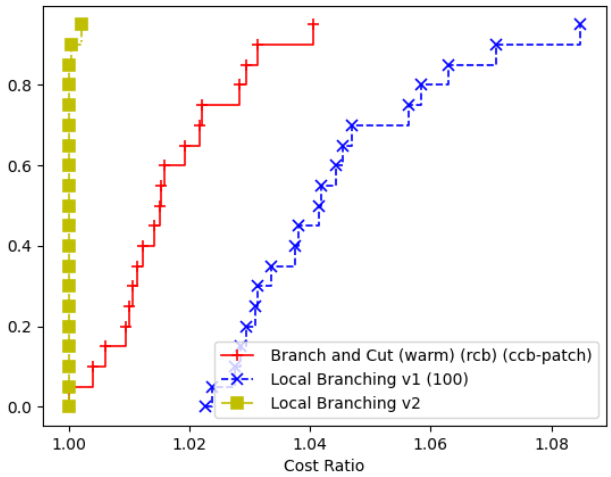
20 instances, 1000 nodes, time limit: 120s

Similarly to the diving algorithm, the gap between the costs found by our best CPLEX solver and the local branching algorithm is smaller than 10%. We can also see an improvement for greater starting values of k.

6.2.3 A more dynamic approach

The choice of k helps us reducing the integrality gap, so we might want to change this value dynamically based on the results of each iteration of the local branching algorithm.

As we can see, with this approach the algorithm manages to move through the search space more smoothly, improving its performances of a 6-8% factor, beating our best CPLEX setting.



20 instances, 1000 nodes, time limit: 120s

Algorithm 10 Local branching matheuristic algorithm (v2)

Input integer $k_{\text{init}} = 100$
Output Hamiltonian cycle of V, cost of 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 *time limit not exceeded* **do**
  if *this is a new iteration* **then**
    add constraint local branching constraint $\sum_{e:x_e^H=1} x_e \geq n - k$
    add the best solution found so far as a warm start
  end if
  $x^* \leftarrow$ *solution returned by CPLEX for input model with local branching constraint*
  if *CPLEX exited for time limit and no visible improvement* **then**
    if *more than 3 repetitions* **then**
      $k \leftarrow k + 10$
      finish this iteration
    else
      repeat this iteration without changing CPLEX model
    end if
  else
    if *CPLEX found optimal solution for this model* **then**
      $k \leftarrow k - 10$
    end if
  end if
  save improvements
  remove local branching constraint
end while

return x^H, cost(x^H)

6.2.4 Considering multiple solutions

The idea behind local branching is to limit the search space using a solution that we know is feasible, but not optimal. The constraint we add to the model ensures that we only fix how many edges we want to take from the solution found in the past iteration. If we want to consider more than one past solution, we can use this variation of the constraint:

$$\sum_{e \in E} \text{count}(e) \cdot x_e \geq n - k$$

where count(e) is the number of times edge e has been considered in past solutions. Note that if we consider just the solution from the last iteration of local branching, count(e)$\in \{0, 1\}$ $\forall e \in E$, which leads to the first version of the constraint.

With this constraint we give more importance to edges that have been considered "optimal" in more than one solution. We follow the hypotesis that edges that are found more times are more likely to be in the optimal solution.

The mathematical meaning of the constraint is changed: in the normal local branching, k is the exact number of edges that CPLEX can change, while in this constraint this claim does not hold. The left-hand side of the constraint now gives more importance to edges whose count is greater than 1; if CPLEX chooses to fix an edge with an high count, it has a bigger search space, since count(e) will bring the lower bound $n - k$ closer, hence it now has more freedom.

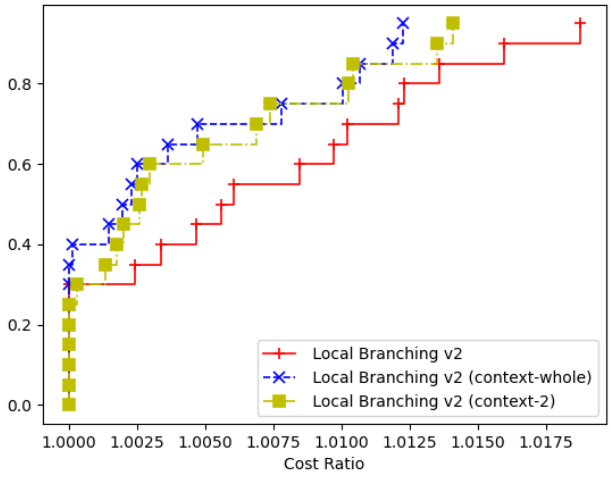
To sum up, with this constraint CPLEX will still search the best k-opt swap around the suggested solution, but it will also try solutions with more swaps around solutions whose edges have been already considered in other iterations of the local branching.

6.2.5 Results analysis

This new constraint is not optimal: if the number of solutions considered start to increase indefinitely the left-hand side of the constraint will reach large values, so large that the meaning of the degrees of freedom that we give to CPLEX, k, will start to vanish.

In our implementation this is not a problem since we give each iteration of local branching 1/10 of the total time limit, so at most count(e) ≤ 10. Moreover, with the v2 version of the algorithm illustrated in section 6.2.3 we almost never reach such a high count.

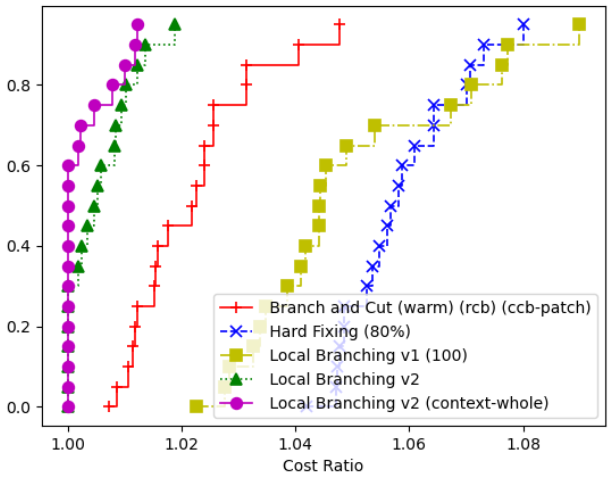
To avoid this problem, we made two different versions of this local branching: one that considers each of the past solutions found, and one which considers only a fixed number of recent past solutions.



20 instances, 1000 nodes, time limit: 120s

Here we can see that using the new constraint gives slightly better solutions than the original one and that limiting the number of past solutions considered in the constraint does not make a difference, as previously explained.

6.3 Comparison diving / local branching



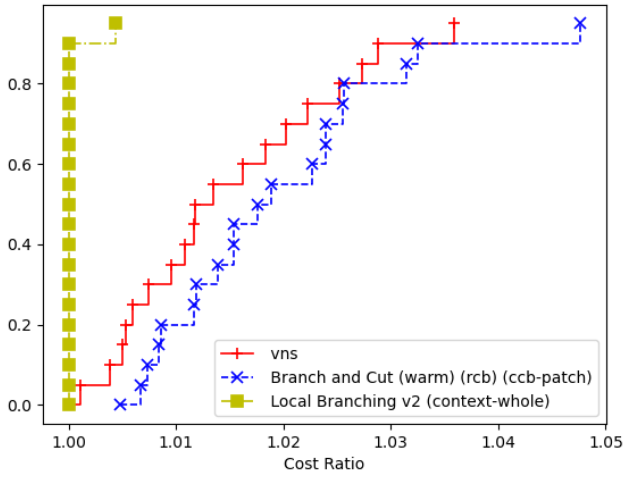
20 instances, 1000 nodes, time limit: 120s

Beating our best CPLEX solver was hard given the range of optimization techniques we used, but in the end we managed to find a matheuristic method which can help us with problems which are too big for CPLEX to handle by itself.

Chapter 7

Conclusions

To sum up the performances of the algorithms proposed, this performance profile compares the best algorithms among metaheuristic, exact and matheuristic approaches:



20 instances, 1000 nodes, time limit: 120s

As we can see, our implementation of the vns algorithm manages to compete with our best CPLEX solver: this is thanks to the speed of the f2opt algorithm which is used to find the first local minimum, since the 2-opt algorithm requires too much time with instances of such size.

The best algorithm for large instances still remains the local branching, which manages to navigate the search space efficiently thanks to our multiple solutions approach to the construction of the constraint.

7.1 Future works

The algorithms proposed are far from optimal and could use some improvements.

Currently the f2opt algorithm merges the sub sub-problems randomly, but an approach similar to the patching seen in section 5.2 could be applied.

The tabu algorithm uses multithreading by performing multiple searches from different starting point at the same time, while the vns algorithm uses it to choose the best number of kicks to do: finding a better way to use multithreading in our tabu search algorithm might close the gap between those two metaheuristic methods.

We could invest some time looking for better patching algorithms for our fractionary solutions, so that in the relaxation callback we can post solutions with an higher quality, lowering the upper bound inside CPLEX.

One last improvement we could make is to further explore the idea of using multiple solutions inside the local branching algorithm: we might use different statistics rather than the number of times an edge has been selected and we might find a better right-hand side to our constraint, so that the degrees of freedom given by that constraint is more controlled.

Bibliography

[1] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical Programming* 91.2 (2002), pp. 201–213.

[2] Fred Glover. "Tabu Search: A Tutorial". In: *Interfaces* 20.4 (1990), pp. 74–94. DOI: 10.1287/inte.20.4.74.

[3] Pierre Hansen et al. "Variable Neighborhood Search". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Springer International Publishing, 2019, pp. 57–97. DOI: 10.1007/978-3-319-91086-4_3.

[4] Martina Fischetti and Matteo Fischetti. "Matheuristics". In: Nov. 2016, pp. 1–33. DOI: 10.1007/978-3-319-07153-4_14-1.

[5] Matteo Fischetti and Andrea Lodi. "Local branching". In: *Mathematical Programming* 98.1 (Sept. 2003), pp. 23–47. ISSN: 1436-4646. DOI: 10.1007/s10107-003-0395-5.