University of Turin

Department of Computer Science

Combinatorial Optimization thesis

Brsnch snd Bound for TSP

Leonardo Magliolo
844910

Gianmario Cuccuru
984100

2023/2024 ACADEMIC YEAR

Abstract

The following paper was developed as an implementation paper for the Combinatorial Optimization course and is concerned with the analysis of the Travelling Salesman Problem, focusing in particular on the 1-Tree-based relaxation proposed by Held & Karp in the 1970s. After a theoretical introduction to the problem, the relaxation proposed by Held & Karp will be examined in detail, which allows for a simpler problem to be solved. Next, a Branch scheme of the problem will be proposed for solving by the Branch and Bound method. Finally, an analysis of the implemented implementation and the results obtained will be provided.

Index

1 Introduction **5**
1.1 The problem of the traveling salesman 5
1.2 NP-completeness of TSP 5
1.3 Mathematical formulation 6

2 The work of Held and Karp **7**
2.1 1-tree 7
2.2 Lagrangian Relaxation 8
2.2.1 An alternative approach 10

3 Branch and Bound **11**
3.1 A general outline 11
3.2 Lower bound calculation 12
3.3 Branching scheme 12
3.4 Closure of knots 13
3.5 Visiting strategies 13

4 Implementation **15**
4.1 Tools used 15
4.2 Data structures 15
4.3 MST calculation 16
4.4 Visiting strategies 16
4.5 Implementation structure 16
4.6 Testing 17

Cspitle 1

Introduction

Given a set of cities and the relative cost of transit between each pair of cities, Traveling Salesman , or TSP for short, is to find the cheapest route to visit all the cities once and only once and finally return to the starting point. The route simply represents the order in which the cities are visited and is called a tour or circuit. The traveling salesman problem, while simple, has been the subject of much research and has wide practical applications beyond the classical routing problem, such as machine scheduling, clustering, and computer wiring. However, the traveling salesman problem is an NP-complete problem and, at the state of art, there are no known algorithms capable of solving it in polynomial time. The basis of solving techniques for TSP is Branch and Bound, for which good bounding techniques are essential. In this case, the goodness of bounding techniques is determined by two often criteria:

the limit must be tight and quick to calculate.

1.1 The problem of the traveling salesman

The Traveling Salesman Problem can be represented by an undirected graph $G = (V, E)$ with costs $c_{(ij)}$ associated with the arcs. The goal is to find a set of arcs $C^* \subseteq E$ with the lowest possible cost. This set C^* must form a Hamiltonian circuit, i.e., a loop through each node of the graph once and only once. In this paper, the symmetric version of the TSP is considered, in which for each pair of nodes $i, j \in V$ is worth $c_{(ij)} = c_{ji}$. In this version, each arc can be traveled in both directions with the same cost. In the context of the traveling salesman problem, vertices are sometimes called cities, and the Hamiltonian circuit is commonly known as a traveling salesman tour.

1.2 NP-completeness of TSP

As mentioned earlier, the traveling salesman problem is known to be an NP-complete pro- blem. To prove this, we must first prove that the TSP belongs to NP. To verify the validity of a tour, we check that the tour visits each vertex once and only once. Then we add up the total cost of the arcs and check whether the total cost is minimal. This can all be completed in time

polynomial, thus proving that the TSP belongs to NP. Second, we prove that the TSP is NP-hard. One way to do this is to show that the Hamiltonian cycle problem, which is known to be NP-complete, can be reduced to the TSP. Suppose we have a graph $G = (V, E)$, which represents an instance of the Hamiltonian cycle problem. We then construct a complete graph $G'(') = (V, E'(('))$, where each pair of vertices is connected by an arc. The cost function is defined as:

$$t()(i, (j)()) \; 0= \begin{cases} & \text{If } (i, j)\in E \\ 1 & \text{If } (i, j)\notin E \end{cases}$$

If G possesses a Hamiltonian cycle, then there exists a cycle in G' with total cost 0, since all the arcs of the original Hamiltonian cycle are present in E and thus have cost 0 in G'. Conversely, if G' has a cycle of total cost 0, then all the arcs of this cycle must be present in E, which means that G possesses a Hamiltonian cycle.

This shows that if we can solve the TSP, we can also solve the Hamiltonian cycle problem, thus proving that the TSP is NP-hard.

Combining these two points, we can conclude that the TSP is NP-complete.

1.3 Mathematical formulation

The symmetric TSP is defined on an undirected graph G=(V,E) with costs c_e associated with each arc $and\in E$. A possible mathematical formulation for the TSP uses variables for each arc:

$$\forall e\in E,\; x_e = \begin{cases} 1 & \text{if and is in the cycle} \\ 0 & \text{otherwise} \end{cases}$$

$$\min \sum_{and\in E} c_e x_e$$

$$(1) \quad s.t. \sum_{and\in \delta(v)} x_e = 2 \qquad \forall v\in V$$

$$(2) \quad \sum_{and\, E\in} x_e \le |\, S\,| - 1 \qquad \forall\, S\subset V : 3 \le |\, S\,| \le |\, V\,| - 1$$

$$(3) \quad x_e\in \{0, 1\} \quad e\forall\in E$$

The goal is to minimize the sum of the costs of the selected arcs. Constraint (1) imposes that every node must be connected by exactly two arcs and ensures that every node in the graph is included in the Hamiltonian circuit. Constraint (2), on the other hand, amounts to the elimination of all sub-cycles: $E(S\subseteq E$ is the set of arcs that have both ends in $S\subset V$. Sub-cycles of order 2 are already excluded from constraint (1). Finally, constraint (3) defines the domain of the decision variables, which take value 1 if the arc is included in the Hamiltonian circuit, 0 otherwise.

Cspitle 2

Held snd Ksrp's work.

The traveling salesman problem (TSP), while simple, has been the subject of much research and has wide practical applications. Being NP-complete, several approaches have been proposed to solve it efficiently, such as exact and heuristic methods. Exact solvers must not only identify the optimal solution, but also prove its goodness, often through a dual limit.

Held and Karp (1970) proposed a relaxation that provides strong dual bounds. For example, in practice, these bounds are used in Concorde, the present state-of-the-art solver (Applegate et al. 2006) or in global constraint design in constraint programming (Benchimol et al. 2010, 2012). Subsequently, a branch-and-bound algorithm using this relaxation was proposed again by Held and Karp (1971) and was shown to be optimal for several benchmark instances available at the time of its publication.

This algorithm exploits a structure called minimum 1-tree that can serve as a valid relaxation for TSP and thus obtain dual bounds.

2.1 1-tree

Finding optimal solutions for large TSP instances requires sophisticated approaches because of combinatorial explosion of the solution space. In the branch-and-bound method, bounds are used to prune the search tree and speed up the search itself accordingly, thus allowing solvers to prove optimality without exploring the entire tree. To achieve this, the relaxation of Held and Karp (1970) provides a robust dual bound based on a variant of the minimum spanning tree.

Let G(V,E) be a complete graph with costs c_e associated with each arc $and \in E$. A minimum 1-tree is a minimum spanning tree of $G \setminus \{1\}$ to which node 1 is added along with the pair of minimum cost arcs connecting it to the tree. It can be seen that the choice of node 1 is arbitrary and depends on the labeling of V . The mathematical formulation of the minimum 1-tree is as follows:

$$\min \sum_{e \in E} c_e x_e$$

$$(1) \quad s.t. \sum_{e \in \delta(1)} x_e = 2$$

$$(2) \quad \sum_{e \in \delta(i)} x_e = 2 \qquad \forall i \in V \setminus \{1\}$$

$$(3) \quad \sum_{\substack{i,j \in S \\ i<j}} x_{i,j} \leq |S| - 1 \qquad \forall S \subset V \setminus \{1\} \wedge |S| \geq 3$$

$$(4) \quad x_e \in \{0, 1\}$$

Constraints (1) and (2) define the structure of the 1-tree, and constraint (3) enforces the elimi- nation of subtours. Thus, the problem is to find an algorithm that can efficiently solve the problem finding a minimum spanning tree on the graph G. In the above formulation, $\delta(v)$ denotes the arcs containing the vertex $\in V$, while $c_e \in \mathrm{R}$ represents the cost of an arc $e \in E$. Finally, $x_e \in \{0, 1\}$ is the decision variable indicating whether or not arc e is included in the 1-tree.

It can be seen that every tour in G is a 1-tree, and if a minimum 1-tree is a tour, then the latter is an optimal relaxation for TSP. Therefore, every minimum 1- tree is a valid relaxation for TSP. However, a solution of this whole program is not necessarily a tour. To do so, a new set of constraints must be applied:

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall v \in V \setminus \{1\}$$

These constraints force each node to have only two arcs, one in and one out, thus transforming the problem into finding a minimum-cost Hamiltonian cycle, which we know is an NP-hard problem.

2.2 Lagrangian Relaxation

To obtain a valid 1-tree relaxation efficiently, you can spo- stand the above constraints, one for each node, in the objective function and penalize their violations by associating with them appropriate Lagrangian multipliers $\theta_v \in \mathrm{R}$ for each $v \in V \setminus \{1\}$. The mathematical formulation of the minimum 1-tree with the introdution of Lagrangian multipliers, obtained by Lagrangian relaxation of constraint (1) on minimum cost Hamiltonian cycle problem, is as follows:

$$\min \left(\sum_{e \in E} c_e x_e - \sum_{v \in V \setminus \{1\}} \theta_v \left(2 - \sum_{e \in \delta(v)} x_e \right) \right)$$

$$(1) \quad \sum_{e \in \delta(1)} x_e = 2$$

$$(2) \quad \sum_{e \in E} x_e = |V|$$

$$(3) \quad \sum_{\substack{i,j \in S \\ i<j}} x_{i,j} \leq |S| - 1 \quad \forall S \subset V \setminus \{1\} \wedge |S| \geq 3$$

$$(4) \quad x_e \in \{0, 1\}$$

Thus, intuitively, any node with a degree other than two will be penalized. An optimal relaxation for the 1-tree can be identified by going to optimize on the variables θ_v. In this regard, Held and Karp (1970, 1971), have- proposed an iterative approach. The idea is to adjust the Lagrangian multipliers θ, step by step, to construct a 1-tree sequence that provides increasingly better bounds. An overview of the process is proposed in the figure below.

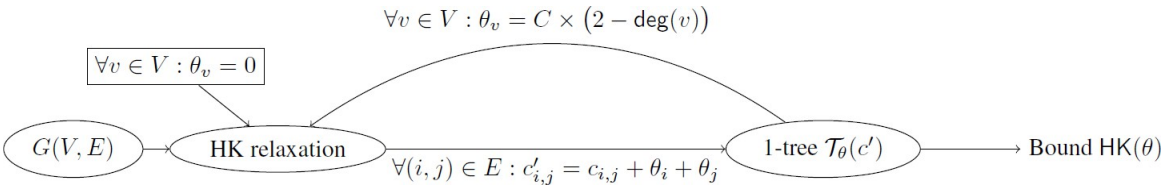

Figure 2.1: Held and Karp's approach.

First, an initial minimum 1-tree is calculated by searching for a minimum spanning tree on $G \setminus \{1\}$ and adding the two cheapest arcs incident on the node 1. If the minimum 1-tree obtained is a tour, then it corresponds to the optimal solution of the TSP. Otherwise, some nodes are penalized because at least one node has a degree greater than 2.

The main idea of Held and Karp (1970, 1971) is to penalize such nodes by changing the cost $c_{(ij)}$ of the arcs $(i, j) \in E$, according to the values of θ_i and θ_j. Let there be $c'_{ij} \in \mathbb{R}$ the modified costs. They are calculated as follows.

$$c'_{ij} = c_{(ij} + \theta_i + \theta_j \qquad \forall (i, j \in E$$

A theoretical property demonstrated by Held and Karp (1970, 1971) is that an optimal tour for the TSP is invariant under this perturbation, whereas an optimal 1-tree is not. This allows the solution to be improved by finding better multipliers. The equation below proposes a standard choice for calculating multipliers, where $C \in \mathbb{R}$ is an arbitrary constant and $deg(v)$ denotes the degree of the node $\in V$ in the 1-tree current.

$$\theta_v = C \times 2 - deg(v) \qquad \forall v \in V$$

At this point, a new minimum 1-tree can be computed from the graph with the updated costs c'_{ij}. This 1-tree is denoted by $T_\theta(c')$, where $c' = \{c_1, ..., c_{|E|}\}$ is the set of all modified costs and $\theta = \theta_1, ..., \theta_{|V|}$ is the set of all multipliers. The notation $cost(T_\theta(c'))$ is also used to refer to the total cost of the 1-tree.

This process is repeated and a new 1-tree $T_\theta(c')$ is obtained until no more improvement is obtained, that is, when a local minimum is reached. The cost of the optimal 1-tree provides a lower bound on the objective function:

$$HK(\theta) = cost\ T_{(\theta)}(c') \sum_{i=1} 2 - \theta_i \tag{2.1}$$

This limit, $HK(\theta)$, is commonly known in the literature as the Held- Karp limit. This approach is typically incorporated into the branch-and- bound algorithm, which uses this limit to prune the search.

2.2.1 An alternative approach

In the work of Parjadis et al. in 2023 [3] an alternative approach for updating Lagrangian multipliers is proposed. The Held-Karp $HK(\theta)$ limit has two interesting properties: (1) it can be parameterized by Lagrangian multipliers θ and (2) it is always valid, i.e., it never exceeds the optimal TSP cost. Both properties provide an opportunity to use a learning-based approach to compute the limit. To this end, it is proposed to construct a
model $\Phi_w : G(V, E) \rightarrow \mathrm{R}^{|V|}$ able to directly predict all multipliers θ for an instance of TSP given as input (i.e., a graph). The model is parameterized with p parameters $w = \{w_1, - - - , w_p\}$. The advantages are twofold. First, it eliminates some parts of the iterative process of Held and Karp (1970) and saves execution time. Second, it allows for potentially tighter bounds. The process is illustrated in Figure 2.2.

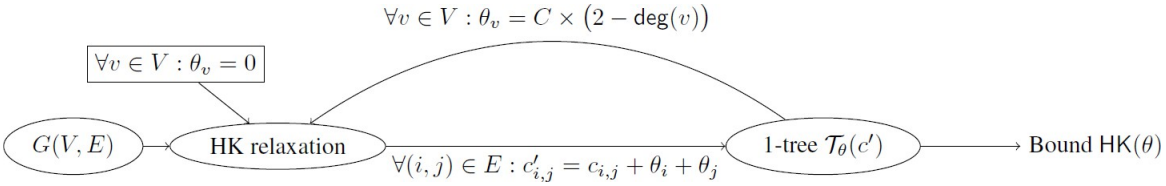

Figure 2.2: Approach of Parjadis et al.

The goal is to find the model parameters w that produce the highest possible limit. This corresponds to a maximization problem that can be solved gradient-based optimization. The formulation for the bound maximization problem and its gradient is given below.

$$\max_w HK\ \Phi_w(G) \rightarrow \nabla_w HK\ \Phi_w(G) \tag{2.2}$$

Cspitle 3

Brsnch snd Bound

There are several approaches for the exact determination of an NP-complete problem such as TSP; among them, implicit enumeration approaches such as branch and bound are the most popular. These algorithms systematically explore the space of solutions in search of an optimal solution. Lower and upper bounds on the optimal value of the objective function are exploited to obtain information about the problem, thus allowing areas of the solution space in which the optimal solution is not present to be excluded from the search; these areas are said to be visited implicitly by the algorithm

However, even using the best available technologies, one can never rule out the possibility of having to examine a substantial fraction of the solution space.

3.1 A general outline

Implicit enumeration algorithms can be seen as a special case of the well-known "divide and conquer" algorithmic scheme, which addresses the solution of a problem by the following steps:

- Break the problem down into a number of "smaller" subproblems;
- solve individual subproblems separately, typically by recursively applying the same procedure until the solution can be obtained by some alternative procedure (base case);
- Combine the solutions of the individual subproblems into a solution for the original problem.

In other words, subdividing a problem into a number of subproblems, such that the union of the admissible regions contains the admissible region of the original problem, and obtaining an upper and lower evaluation on the optimal value of the objective function of each of the identified subproblems.

3.2 Lower bound calculation

As mentioned earlier, to determine a minimum 1-tree and thus obtain a lower bound for the TSP problem, a first minimum 1-tree can be computed by searching for a minimum spanning tree on $G\setminus\{n\}$ and adding the two cheapest arcs incident on node n.

The cost of the above procedure is dominated by the computation of the minimum spanning tree, which can be easily solved using an algorithm such as Kruskal's, which has computational cost equal to $O(m \log n)$. On the other hand, the selection of the minimum cost arcs can be done by a simple scanning of the arcs in $O(m)$.

The computation and updating of the lower bound is based on the identification of admissible so- luations during the execution of the algorithm. If the computed 1-tree turns out to be a Hamiltonian circuit, then the lower bound can be updated if its cost is less than the currently known cost. The lower bound will be initialized to$+\infty$ at the beginning of the procedure.

3.3 Branching scheme

Let us introduce at this point the most widely used way to implement the separation operation, namely, the division of X into $X_1, X_2, \ldots, X_k$. To simplify, let us initially assume that it is $X \subseteq \{0, 1\}^n$, i.e., that the solutions of the problem can be described by n binary decisions, each represented by a variable x_i, $i = 1, \ldots, n$: one possible way to partition X is to make decisions about some of the variables. For example, fixed at any index i, we can partition X as $X_0 \cup X_1$, where

$$X_0 = \{x \in X : x_i = 0\} \text{ and } X_1 = \{x \in X : x_i = 1\}$$

Moreover, the sets thus obtained can themselves be partitioned according to the same pattern. Indeed, it is possible to represent the admissible set X by means of a decision tree (or branch tree), which associates each admissible solution with a sequence of decisions that generates it.

In the particular case of TSP, if the solution of the relaxation obtained turns out to be a Hamiltonian circuit, then the corresponding node is closed by am- missibility. If, on the other hand, the node is not closed by admissibility, then what we want to achieve is to prevent the same circuit present in the parent node from being formed in the child nodes, and we do this by introducing the subdivision rule that will now be illustrated and that allows us to construct a decision tree exactly as illu- strated above.

We denote by $\{(i_1, j_1), (i_{(2)}, j_2), \ldots, (i_r, j_r)\}$ the arcs that make up that cir- cuit. The subdivision rule operates as follows: the first child will be generated by imposing that the arc $(i_{(1)}, j_1)$ is not part of the solution, i.e., by imposing $x_{(i)(1)(j)_1} = 0$; the second child will be generated by imposing instead that the arc $(i_1, j_{1)}$ is present but the arc $(i_{(2)}, j_{2)}$ is absent, i.e., by setting $x_{i_{(1)(j)(1)}} = 1$ and $x_{i_{(2)(j)_2}} = 0$. The process is iterated in this way until we arrive at the r-th child, which will have all the first r-1 arcs present in the circuit but will not contain the last one. Thus, each node in the branch tree thus constructed will be associated with both the set E_0, which

contains all the arcs that are NOT part of the solution, either the set E_1 that contains all the arcs that are instead part of the solution. It is essential that E_0 and E_1 are disjointed, that is, $E_0 \cap E_1 = \emptyset$.

This makes it possible to solve for each child a subproblem of the type $S = (E_0, E_1)$, containing all Hamiltonian circuits formed by the arcs present in E_1 but not the arcs present in E_0. For each subproblem, the calculation of the lower bound is done according to the procedure described in the previous section, but taking into account in the calculation of the MST the arcs that must be present (present in E_1) and those that must be excluded (present in E_0). In our implementation, the calculation of the MST is done using the Kruscal algorithm, which chooses the least weighted arcs in ascending order, avoiding cycles. Kruscal's implementation, instead of initializing the set E_t with the empty set, it will be initialized with the arcs present in E_1 that are not incident on node n; obviously, the arcs present in E_0 should not be considered. Once the covering tree T , the two best arcs incident on n, i.e., those with the lowest cost, will be added to it.

3.4 Closure of knots

The closure criteria for a generic branch tree node P_i are now defined.

If $\hat{z} < LB(P_i)$, that is, when the lower bound obtained is greater than the best Hamiltonian circuit found so far, then further exploration of the part of the solution space represented by the node can be avoided. In this case, the node is said to have been pruned by the higher evaluation, or in other words it is closed by bound. In fact, the node is discarded and we move on to visit another of the active nodes (if there are any) A special case is when we cannot compute a 1-tree for the node; in this case there is no admissible solution for the relaxation under consideration and the node is closed (or pruned) by ineligibility.

It should be noted that "pruned" nodes are those where the visitation stops, i.e., the leaves of the subtree that is actually generated are not necessarily, and hopefully are not, leaves of the entire decision tree. Eliminating a node allows the corresponding entire subtree of the decision tree not to be explicitly visited: this is why it is called pruning.

If the 1-tree generated by the relaxation turns out to be a Hamiltonian circuit, then there is no need to explore further. The node is closed because it is a possi- ble optimal candidate (potato for admissibility). If the node also has a better cost, i.e., less than the current solution found, then the latter would be updated.

3.5 Visiting strategies

The visiting strategy of the decision tree is basically dictated by the strate- gia of selecting the next node to be visited from the set Q. It is usually distinguished

between topological and information-based visits. Topological visits choose the next node to visit solely on the basis of the topological structure of decision tree; this corresponds to node selection strategies that depend solely on its position in *Q*. The two best-known topological visitation strategies are fan-first, or breadth-first, corresponding to implementing Q as a queue, and sounding-first, or depth-first, corresponding to implementing *Q* as a stack.

Of these, in the context of implicit enumeration algorithms the depth-first strategy can be par- ticularly useful. Since pruning nodes through higher evaluation requires lower evaluation, the use of a depth-first strategy may be indicated since it brings the visit quickly ver- so the leaves of the decision tree, and thus may allow for fast generation of admissible solutions. Topological visits offer some advantages an implementation perspective. First, they are simple to implement and the cost of running *Q* is low. Second, the depth-first strategy lends itself to being implemented in such a way as to keep the number of active nodes in the decision tree very low.

From the point of view of spatial complexity, the breadth-first strategy requires exponential amounts of memory relative to the depth of the tree, since all nodes at a certain level of the tree must be stored concurrently. In contrast, the depth-first strategy is much more efficient since it requires storing only a sequence of nodes along a path from origin to leaf.

In contrast, information-based visitation rules use information about the nodes in *Q* to determine the next node to be extracted; in other words, they correspond to implementing *Q* as a priority queue. The most frequently used information-based strategy is best-first, in which each node is associated with the value of the top evaluation produced by the corresponding relaxation and the node with the highest value is selected, which corresponds to the "most promising" subtree, i.e., in which the optimal solution should be most likely to be encountered. The best-first strategy is widely used because it usually succeeds in directing the visit to the "most promising" areas of the decision tree. In contrast, the best-first strategy is more complex to implement, and it typically happens to sequentially examine "distant" nodes in the decision tree, which correspond to "very different" problems. It also presents greater spatial complexity than topological visits, since the use of a priority queue involves the need to me- morize and sort a potentially large number of nodes. This results in significant memory consumption, especially in the case of very large trees, thus making best-first less practical in contexts with limited memory resources.

Cspitle 4 Implementation

We will go on to describe in this section the implementation choices used for solving the TSP using the branch-and-bound method.

4.1 Tools used

During the experimentation phase, we decided to use the following tools:

- **Java** as a programming language, given its known efficiency and simpli- city compared to other languages such as C. In fact, Java automatically manages memory and does not use pointers, thus making the code more lightweight and manu- suable. However, the very implicit memory management using garbage collectors can cause higher memory consumption than in C.

- **TSPLib**, a library used to test and evaluate the implemented algorithm. TSPLib provides a collection of standardized problem instances, with known optimal or best-fit da- ti and solutions, that allow comparison of algorithm performance on a common set of problems.

4.2 Data structures

The main data structure we decided to use is adjacency lists, implemented using HashMap, for both graph and 1-tree representation. This data structure offers several advantages in terms of computational complexity and memory usage.

The representation using HashMap requires memory space proportional to the number of nodes plus the number of arcs ($O(n+ m)$), which is significantly more efficient than an adjacency matrix that would require $O(n^2)$ memory space. The central operations in our implementation include scanning the adjacencies of a node and exploring the graph. With adjacency lists implemented using HashMap, the complexity of scanning the adjacencies of a node is $O(|\ A(u)|\ )$, where $A(u)$ is the set of adjacencies of node u.

4.3 MST calculation

To calculate the MST, we implemented Kruskal's greedy algorithm. The al- gorithm sorts the arcs according to increasing costs and constructs an optimal set T of arcs, each time choosing an arc of minimum weight that does not form cycles with the already chosen arcs. To do this, it manages a partition $W = \{W_1, W_2, \ldots, W_k\}$ of V, set of nodes in the graph, in which each W_i represents a set of nodes so a set of arcs was chosen to connect them. Initially, $T = \emptyset$ and $W = \{\{1\}, \{2\}, \ldots, \{n\}\}$, since no arc was chosen and therefore no node has been connected. At the first iteration, the node (u, v) of minimum weight is chosen; this is placed in T and the sets $\{u\}$ and $\{v\}$ are replaced with the set $\{u, v\}$. At generic iteration i, we examine the arc (x, y) with i-th cost, which is only ag- enced to the solution T if the nodes x and y do not belong to the same set as the partition W (i.e., if the arc does not form cycles with the previously inserted arcs). In this case, after inserting the arc (x, y) into T, the (distinct) sets containing x and y (Find-Set) are replaced in the partition W by their union (Union-Set). Sorting $m = |E|$ arcs requires $O(m \log m)$ steps. The cost of m iterations depends on the cost of the only nonconstant operations, namely Find-Set and Union-Set.

4.4 Visiting strategies

We chose to implement two different versions for branch and bound: a multi-threaded version using the Best-first search strategy and a single-threaded version using the Depth-first search strategy. This choice is due to considerations of efficiency and applicability.

The Best-first search strategy explores the nodes in the search tree that correspond to the "most promising" subtrees, that is, in which one would be most likely to encounter the optimal solution. In this strategy, sibling problems are independent of each other. This means that each subproblem can be solved in parallel without the need for synchronization, taking full advantage of the potential given by parallelization. This results in a significant reduction in computational time.

In the Depth-first search strategy, exploration is done by delving into each branch of the search tree before moving on to the next. This creates a sequential dependency between nodes, making difficult to apply a multi-thread strategy, as each node depends on the full exploration of the previous node. However, the DFS strategy can be effective in situations where the multi-threaded version fails, such as when there is insufficient memory available. Thus, the combination of a multi-thread version with Best-first search and a single-thread version with DFS allows for a flexible approach to TSP resolution, allowing DFS to be used in cases where Best-first search is not effective.

4.5 Implementation structure

The developed code is structured as follows:

- **BranchAndBoundTSP**: This package contains all the useful classes in order to solve the TSP problem using precisely the Branch and Bound algorithm detailed earlier. The main classes include the actual implementation of the Branch and Bound algorithm itself, the handling of intermediate problems during the execution of the algorithm, and the exceptions needed to handle cases that cannot be solved.

- **Datastructures**: this package contains all the classes necessary for the gestion of graph structures, the implementation of Kruscal's algorithm for calculating the MST, and the implementation of the LIFO blocking queue.

- **TSPLib**: In this package is the parser used to read and inter- prect the files in the TSPLib library, thus enabling the generation of the corresponding graph. We used a pre-existing parser, adapting it to our specific purposes.

4.6 Test

All tests were conducted using a PC equipped with an AMD Ryzen 7 5800X CPU and 32 GB of RAM. The results obtained are as follows:

Burma 14 - BestFS		
NumThreads	Nodes	Time
1	1732736	64 s
2	1716906	54 s
8	1744534	31 s

Burma 14 - DFS		
NumThreads	Nodes	Time
1	1822381	70 s

Ulysses16 - BestFS		
NumThreads	Nodes	Time
1	3215759	12 m 32 s
2	3376546	10 m 54 s
8	3154971	7 m 8 s

Ulysses16 - DFS		
NumThreads	Nodes	Time
1	3837334	16 m 2 s

Gr17 - BestFS		
NumThreads	Nodes	Time
1	> 4.5 M	> 30 m, Out of memory
2	> 4.5 M	> 23 m, Out of memory
8	> 4.5 M	> 18 m, Out of memory

Gr17 - DFS		
NumThreads	Nodes	Time
1	5089038	33 m 34 s

As can be seen from the tables above, the TSP instances used for testing include Burma14, Ulysses16, and Gr17, all from the TSPLib library. Test results for the BestFS strategy show that for Burma14, as the number of threads increases, the execution time decreases substantially, from 64s with 1 thread to 31s with 8 threads; this highlights how the algo- pace scales better with the use of more threads. Similar behavior is observed for Ulysses16, where the execution time decreases from 12m 32s with 1 thread to 7m 8s with 8 threads, again showing the efficiency multithreaded execution. However, for Gr17 all configurations lead to an execution that terminates in out of memory, thus suggesting that the problem size exceeds the available memory.

As for the DFS strategy, Burma14 completed execution in 70s, Ulysses in 16m 2s and Gr17 in 33m 34s thus showing that the DFS strategy, although slower than BestFs, does not incur memory problems.

In conclusion, the tests clearly show the effectiveness of the BestFS strategy in combination with multi-threaded execution. However, the out-of-memory error for Gr17 suggests that despite the benefits multi-threaded executionmemory management is a crucial issue for large instances. In contrast, the DFS approach, while less time-efficient than BestFS, shows greater stability for memory-intensive instances.

Bibliogrsfis

[1] D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton University Press, 2011.

[2] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138-1162, 1970.

[3] Augustin Parjadis, Quentin Cappart, Bistra Dilkina, Aaron Ferber, and Louis-Martin Rousseau. Learning lagrangian multipliers for the traveling salesman problem, 2023.

[4] L. De Giovanni M. Di Summa. *Methods and Models for Combinatorial Optimization*. University of padova, 2020.