Operations Reserach 2 Final Thesis

Youssef Ben Khalifa

April 9, 2025

1 Introduction

The Traveling Salesman Problem (TSP) stands as one of the most iconic and extensively studied problems in combinatorial optimization and computer science. Despite its deceptively simple formulation—finding the shortest possible route that visits each city exactly once and returns to the origin city—the TSP has profound implications for both theoretical computer science and practical applications.

In this thesis we will go over the implementation of the main heuristics involving the famous TSP problem. The entire implementation is done using the C programming language and the CPLEX optimization LP optimization framework. The goal is to research and implement the most common Metaheuristics and Mathheuristics for solving the TSP problem using MIP problems.

Throughout this thesis, we will explore the historical development of each optimization method discussed. We will trace the evolution of metaheuristics from early construction heuristics to sophisticated approaches like Tabu Search and Variable Neighborhood Search, examining how these methods emerged in response to computational challenges. Similarly, we will investigate the historical context behind mathheuristics, which represent a more recent fusion of mathematical programming with heuristic frameworks, developed to address the limitations of pure exact methods when tackling large-scale combinatorial problems. This historical perspective will provide valuable insights into why particular techniques were developed and how they have evolved to their current form.

1.1 The TSP Problem and Its Significance

The TSP belongs to the class of NP-hard problems, meaning that no known algorithm can solve it optimally in polynomial time as the problem size increases. This fundamental hardness has made the TSP a benchmark problem for evaluating algorithmic approaches and computational methods. The significance of the TSP extends far beyond its theoretical interest, with applications spanning numerous domains:

- Logistics and transportation planning
- Circuit board drilling in manufacturing
- DNA sequencing in bioinformatics
- Vehicle routing and delivery optimization
- Telecommunications network design
- Warehouse order picking

The universality of the TSP's structure—finding an optimal arrangement or sequence—makes it relevant to any field where minimizing traversal costs is essential.

1.2 Heuristics and Metaheuristics for TSP

Given the computational intractability of finding exact solutions for large TSP instances, researchers have developed various metaheuristics—high-level problem-independent algorithmic frameworks that

provide strategies for developing heuristic optimization algorithms. Unlike exact methods, metaheuristics sacrifice the guarantee of optimality for computational efficiency, providing near-optimal solutions in reasonable time frames.

Metaheuristics for the TSP include:

- Construction heuristics like Nearest Neighbor and Greedy algorithms that build tours incrementally
- Local search methods such as 2-opt and 3-opt that iteratively improve solutions
- Population-based approaches like Genetic Algorithms that evolve multiple solutions simultaneously
- Memory-based techniques such as Tabu Search that use historical information to guide the search
- Adaptive methods like GRASP (Greedy Randomized Adaptive Search Procedure) that combine greedy elements with randomization

These approaches have proven remarkably effective in practice, often producing solutions within a few percentage points of optimality for instances with thousands of cities.

1.3 Mathheuristics for TSP

Mathheuristics represent a more recent development in the field, combining mathematical programming techniques with metaheuristic frameworks. These hybrid methods leverage the strengths of exact optimization methods (like linear and integer programming) while maintaining the scalability advantages of heuristics.

For the TSP, mathheuristics typically involve:

- Decomposition techniques that break the problem into manageable subproblems
- Variable fixing approaches that reduce the problem size by fixing certain decision variables
- Local branching methods that explore neighborhoods defined by mathematical constraints
- Cut generation procedures that iteratively strengthen mathematical formulations

By integrating mathematical rigor with heuristic flexibility, mathheuristics often achieve superior performance compared to either approach used independently, especially for complex or large-scale instances.

1.4 Scope of This Thesis

This thesis investigates both traditional metaheuristics and modern mathheuristics for solving the TSP, implementing and analyzing several approaches using the C programming language and the CPLEX optimization framework. We focus on understanding the mathematical foundations, implementation details, and computational performance of these methods, with particular attention to their practical effectiveness across different problem instances.

Through systematic experimentation and analysis, we aim to provide insights into the strengths and limitations of different solution approaches, contributing to the ongoing effort to develop more efficient algorithms for this foundational optimization problem.

Throught the next chapters we will be going over following:

1. **Introduction to the TSP Problem**: quick overview of the Mathematical formulation of the TSP problem.
2. **Heuristics**: Implementation of common heuristic algorithms including:
    - Greedy Randomized Adaptive Search (GRASP)
    - Extra Mileage
    - Refining Heuristics (2-opt move)

3. **Neighbourhood Search**: Implementation of neighbourhood search algorithms including:
    - Tabu Search
    - Variable Neighborhood Search (VNS)
4. **Mathheuristics**: Implementation of mathematical optimization techniques combined with heuristics:
    - Hard Fixing
    - Local Branching

For each algorithm, we will examine the implementation details, mathematical foundations, and effectiveness in solving the TSP problem. The implementations for the mathheuristic algorithms are done utilizing the CPLEX optimization framework.

2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is one of the most extensively studied problems in combinatorial optimization. The problem asks to find the shortest possible route that visits each city exactly once and returns to the origin city, given a set of cities and the distances between them. Formally, given a set of n cities and a cost matrix $[cij]$ that specifies the cost (or distance) of traveling from city i to city j, the goal is to find a permutation of the cities that minimizes the total tour length. Despite its simple description, the TSP is NP-hard, meaning there is no known polynomial-time algorithm that can solve it optimally for large instances.

2.1 History of the TSP

The Travelling salesman problem was first formulated in the 1930's, whose first mathematical formul was given my Meril M. Flood. Since its introduction, the TSP problem has been one of the most researched subject in the mathematical optimization filed, and since then it has served as a benchmark for many optimization algorithms. Throughout the years, several exact approaches have been developed that are able to compute an exact solution even for very large instances of the problem (Applegate, Bixby, Chvatal, Cook, & Problem, 2006), and also many heuristic approaches have been proposed that approximate with incredible accuracy even instances with milions of nodes.

Numerous research efforts—such as those by Chisman (1975), Garg & Ravi (2000), and Henry-Labordere (1969)—highlight the strong academic interest in extensions and variants of the Traveling Salesman Problem (TSP). Over the years, several important extensions have been explored, including:

- the Vehicle Routing Problem (Laporte, 1992),
- he TSP with Time Windows (da Silva & Urrutia, 2010),
- the Steiner TSP (Rodríguez-Pereira, Fernández, Laporte, Benavent, & Sykora, 2019),
- the Selective TSP (Laporte & Martello, 1990),
- the Multi-objective TSP (Psychas, Delimpasi, & Marinakis, 2015),
- the Multiple TSP (Cheikhrouhou & Khoufi, 2021),
- the Bottleneck TSP (Garfinkel & Gilbert, 1978),
- the Prize Collecting TSP (Balas, 1989),
- the Clustered TSP (Chisman, 1975), and
- the Generalized TSP (Henry-Labordere, 1969; Saskena, 1970; Srivastava, Kumar, Garg, & Sen, 1969).

These extensions reflect the versatility and practical significance of the TSP framework across a wide range of optimization scenarios.

2.2 Mathematical Formulation

First let us define the components that characterize the TSP problem: we begin from a graph $G = (V, E)$ where V is the set of vertices (cities) and E is the set of edges connecting the vertices, and a cost matrix C containing the weights/costs of each edge $(i, j) \in E$. Spoiler: in the implementation instead of adopting a big cost matrix, we will be simply using the euclidean distance between the nodes for computing the cost $c_{i,j}$ of a given edge (i, j) , therefore the cost of a tour will simply be computed as the sum of the euclidean distances between the nodes. This will not affect the final formulation.

We then can formalize Traveling Salesman Problem as an optimization problem as follows:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{1}$$

subject to:

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in \{1, \ldots, n\} \tag{2}$$

$$u_i - u_j + n x_{ij} \leq n - 1 \quad \forall i \in \{2, \ldots, n\}, j \in \{2, \ldots, n\} \tag{3}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in \{1, \ldots, n\} \tag{4}$$

where:

- c_{ij} is the cost of traveling from node i to node j.
- x_{ij} is a binary variable that indicates whether the salesman travels from node i to node j.
- u_i is a continuous variable that represents the position of node i in the tour.

3 Development and environment setup

All the code implementations mentioned in this thesis have been developed and tested on a Linux environment using the C programming language. The main components of the development setup are:

System Environment

- Operating System: Ubuntu 22.04 LTS
- Architecture: x86_64
- Compiler: GCC 11.4.0
- Build System: CMake 3.22.1

Dependencies

- IBM ILOG CPLEX Optimization Studio 22.1.1
- Standard C Libraries (stdio.h, stdlib.h, math.h)
- Python 3.8 or higher (for profiling and testing)

Project Structure The project is organized as follows:

- `/src`: Contains all source files
- `/include`: Header files
- `/test`: Test instances and test cases
- `/build`: Build directory (generated by CMake)
- `/profiler`: Profiling scripts and results

Build Configuration The project uses CMake as its build system. The main CMakeLists.txt file configures the build environment and links necessary dependencies, particularly the CPLEX libraries required for the mathematical optimization components.

4 Data Structures

The implementation uses several key data structures to efficiently manage and process TSP-related data:

4.1 TSP Instance Structure

The core data structure is the `instance` struct, which holds all problem-related data:

```
typedef struct {
  int nnodes;          // number of nodes
  double* xcoord;      // x coordinates of nodes
  double* ycoord;      // y coordinates of nodes
  int* solution;       // current solution tour
  double best_cost;    // cost of best solution found
  double elapsed_time; // computational time
} instance;
```

4.2 Solution Representation

Solutions are represented in two main ways:

- **Tour Array**: A simple integer array where `solution[i]` represents the node visited at position i
- **Edge Matrix**: A binary matrix where `x[i][j] = 1` indicates edge (i,j) is in the tour

4.3 Auxiliary Data Structures

Several helper structures are used throughout the implementations:

- **Distance Matrix**: Pre-computed matrix of Euclidean distances between nodes
- **Component Map**: Array used in subtour detection algorithms
- **Tabu List**: Circular queue for storing recently visited solutions
- **Candidate List**: Dynamic array for storing potential moves

These data structures are designed to balance memory efficiency with computational speed, particularly important for larger TSP instances.

5 Heuristics for TSP

Metaheuristics represent a sophisticated class of computational methods that have evolved to address complex optimization problems where exact methods become impractical due to computational constraints. As optimization problems grow in scale and complexity, traditional exact methods often struggle with the combinatorial explosion of possible solutions. Metaheuristics emerged as an effective approach to navigate these vast solution spaces by intelligently balancing exploration of new regions with exploitation of promising areas. In this section we will go over a brief history of the development of the metaheuristics algorithms and the implementation of the following metaheuristics:

- Greedy Randomized Adaptive Search (GRASP)
- Extra Mileage
- Refining Heuristics (2-opt move)

for each algorihtm you will see the pseudocode for both the algorithm and the implementation of the algorithm in C.

5.1 Greedy Randomized Adaptive Search (GRASP)

The GRASP method is basically a greedy randomized adaptive search procedure. It consists of two main phases: construction and local search. In the construction phase, a feasible solution is built, one element at a time, in a greedy randomized fashion. In the local search phase, the neighborhood of the constructed solution is explored until a local minimum is found.

Construction Phase In the construction phase, the solution is built iteratively. At each iteration, a candidate list is created, containing the best elements to be added to the solution. One of these elements is chosen randomly, according to a probability distribution, and added to the solution.

Local Search Phase In the local search phase, the algorithm explores the neighborhood of the constructed solution to find a local minimum. This is done by iteratively replacing the current solution with a better solution from its neighborhood, until no better solution can be found.

Algorithm The GRASP algorithm can be summarized as follows:

1. **Initialization:** Set the best solution found to null.
2. **Construction:** Build a feasible solution using the greedy randomized approach.
3. **Local Search:** Improve the constructed solution using local search.
4. **Update:** If the improved solution is better than the best solution found, update the best solution.
5. **Termination:** Repeat steps 2-4 until a stopping criterion is met (e.g., a maximum number of iterations).

Algorithm 1 TSP GRASP Metaheuristic

```
1: procedure TSP_GRASP(instance, max_iterations)
2:     best_solution ← null
3:     best_cost ← ∞
4:     for iteration ← 1 to max_iterations do
5:         // Construction Phase
6:         solution ← ∅
7:         current ← RandomStartNode()
8:         solution.Add(current)
9:         while —solution— ¡ n do
10:            candidates ← BuildCandidateList(current)
11:            RCL ← BuildRestrictedCandidateList(candidates)
12:            next ← RandomSelect(RCL)
13:            solution.Add(next)
14:            current ← next
15:        end while
16:        // Local Search Phase
17:        improved_solution ← LocalSearch(solution)
18:        cost ← ComputeCost(improved_solution)
19:        if cost ¡ best_cost then
20:            best_solution ← improved_solution
21:            best_cost ← cost
22:        end if
23:    end for
24:    return best_solution
25: end procedure
```

Implementation The implementation of the GRASP algorithm takes as input arguments:

- `instance`: The TSP instance data structure.
- `start_node`: The index of the starting node for the tour.

The function internally allocates and sets the solution as an array integers representing the tour.... The GRASP algorithm is implemented as follows:

1. **Setup**:
   - The function initializes the current node index to the starting node (which is passed as an argument) and sets the remaining nodes count to the total number of nodes.
   - It allocates memory for the solution array and remaining nodes arrays. These two arrays will be used respectively to store the tour and the remaining nodes for the main loop to visit.
   - The solution array is initialized with `-1` to indicate that no nodes have been visited yet.
   - The remaining nodes array is initialized with all node indices.
2. **Main Loop**:
   - The function iterates over all nodes to construct the solution.
   - For each node, it finds the nearest unvisited node using the `euclidean_nearest_node` function.
   - If no nearest node is found (i.e., all nodes are visited), it connects the current node back to the starting node to complete the tour.
   - Otherwise, it updates the solution with the nearest node and logs the current node index and nearest node found.
   - The current node index is updated to the nearest node for the next iteration.

3. **Finalization**:

   - After constructing the solution, the function records the end time and calculates the elapsed time.
   - It logs the total time taken and the cost of the solution.
   - Finally, it frees the allocated memory for the remaining nodes array.

Here's the pseudocode of the implementation:

Algorithm 2 TSP GRASP

```
1: procedure TSP_GRASP(instance, solution, alpha)
2:     solution ← ∅
3:     candidates ← {1, ..., n}                          ▷ Set of unvisited nodes
4:     current ← RandomNode(candidates)
5:     solution.Add(current)
6:     Remove(candidates, current)
7:     while candidates not empty do
8:         RCL ← ∅                                       ▷ Restricted Candidate List
9:         c_min ← MinDistance(current, candidates)
10:        c_max ← MaxDistance(current, candidates)
11:        threshold ← c_min + α(c_max − c_min)
12:        for all node ∈ candidates do
13:            if Distance(current, node) ≤ threshold then
14:                RCL.Add(node)
15:            end if
16:        end for
17:        next ← RandomSelect(RCL)
18:        solution.Add(next)
19:        Remove(candidates, next)
20:        current ← next
21:    end while
22:    solution.Add(solution[0])                         ▷ Return to starting node
23:    return solution
24: end procedure
```

5.2 Extra Mileage

In the extra mileage heuristic, we try to build a valid tsp solution starting an edge of the graph and we try and build a tour by selecting the adjacent edge with the minimum cost.

Algorithm The extra mileage algorithm can be summarized as follows:

1. **Initialization:** Initialize the starting edge.
2. **Construction:** Build a feasible solution by iteratively selecting the edge with the minimum cost.
3. **Update:** If the improved solution is better than the best solution found, update the best solution.
4. **Termination:** Repeat steps 2-3 until a stopping criterion is met (e.g., a maximum number of iterations).

Implementation The extra mileage algoithm implementation takes as input arguments:

- `instance`: The TSP instance data structure.
- `starting_pair`: The pair of nodes that will be used to start the tour.

Ideally the starting pair should be the most distant pair of nodes in the graph in order to optimize the construction of the solution, by design this is not a restriction that has been instrisically implemented into the function, so any pair of nodes can be used as a starting pair. The extra mileage algorithm is implemented as follows:

1. **Setup**:
   - The function initializes the current_pair to the starting pair (which is passed as an argument).
   - An heuristic state data stcuture is initialized to keep the track of the covered nodes and the final solution
2. **Main Loop**:
   - While there are uncovered nodes (condition based on the comparison of the covered nodes count and the total number of nodes), the function iterates over the uncovered nodes to construct the solution:
     - For each covered node, a second loop is exectued to find the nearest uncovered node such that the the resulting edge between the two nodes is the argument that minimizes the total cost of the tour up until that point.
     - if such node is found, that node is selected and added to the tour, the covered nodes count is incremented and the node is removed from the uncovered nodes list.
3. **Finalization**:
   - Once all nodes are covered, the function calculates the total cost of the solution.
   - It records the end time and calculates the elapsed time.

Algorithm 3 TSP Extra Mileage

```
1: procedure TSP_ExtraMileage(instance, solution, start_pair)
2:     solution ← ∅
3:     candidates ← {1, ..., n}                          ▷ Set of unvisited nodes
4:     tour ← [start_pair.first, start_pair.second]      ▷ Initial tour with farthest pair
5:     Remove(candidates, start_pair.first)
6:     Remove(candidates, start_pair.second)
7:     while candidates not empty do
8:         best_node ← null
9:         best_pos ← null
10:        min_cost ← ∞
11:        for all node ∈ candidates do
12:            for all pos ∈ {1, ..., —tour—} do
13:                extra_dist ← Distance(tour[pos-1], node)
14:                extra_dist ← extra_dist + Distance(node, tour[pos])
15:                extra_dist ← extra_dist - Distance(tour[pos-1], tour[pos])
16:                if extra_dist ¡ min_cost then
17:                    min_cost ← extra_dist
18:                    best_node ← node
19:                    best_pos ← pos
20:                end if
21:            end for
22:        end for
23:        Insert(tour, best_node, best_pos)
24:        Remove(candidates, best_node)
25:    end while
26:    solution ← tour
27:    return solution
28: end procedure
```

5.3 Refining Heuristics

Refining heuristics are a type of heuristic algorithm that iteratively improve an initial solution by applying a series of local search moves. These can be applied on top of the existing heuristics we have introduced so far in order to improve their performance. In the next section we will go over a one particular refining heuristic called the *2-opt move*[Nil03].

5.3.1 2-opt Move

Assume we have a graph $G = (V, E)$ and a directed tour $T \subseteq E$, valid as a tsp solution, the 2-opt move takes two edges in that tour and swaps the nodes between them. In particular, let $(i, j), (h, k) \in T$, the 2-opt move replaces the edges (i, j) and (h, k) with the edges (i, h) and (j, k).

Algorithm The goal of this type of operation is to swap two "longer" edges for two "shorter" ones (by "longer" and "shorter" we refer to the edge cost): therefore in our refining heuristic we aim at:

1. Selecting two edges $(i, j), (h, k) \in T$: for selecting the two edges we can adopt two different strategies:
   - Select the first pair of edges that satisfy the following:
   $$c_G(i, j) + c_G(h, k) > c_G(i, h) + c_G(j, k) \tag{5}$$
   - Or select the pair of edges in G which swap results in the maximum cost reduction:
   $$argmax_{((i,j),(h,k))\in E\times E}(\Delta c_G = (c_G(i, h) + c_G(j, k)) - (c_G(i, j) + c_G(h, k))) \tag{6}$$

   where $c_G(i, j)$ is the cost associated with edge (i, j).
2. Replacing the two edges with the edges (i, h) and (j, k).
3. If the graph is directed, invert the direction of the tour of the edges between (j, k) and (i, h).

Implementation The 2-opt move implementation takes as input arguments:

- `instance`: an instance of the TSP problem so solve.
- `solution`: an instance of a valid TSP solution.

The function is implemented as follows:

1. **Main loop:**
   - The function iterates over all edges in the solution to find a valid 2-opt move. In this implementations, this is identified by the pair of edges that satisfy (6)
   - If a valid 2-opt move is found, the pair is saved as the designated edges to be swapped.
2. **Swap:** We apply the 2-opt move on the solution by swapping the designated edges.
3. **Finalization:** We invert the direction of the edges between (j, k) and (i, h) if the graph is directed.

Algorithm 4 TSP 2-opt Local Search

```
1: procedure TSP_TwoOpt(instance, solution)
2:     improved ← true
3:     while improved do
4:         improved ← false
5:         best_gain ← 0
6:         best_i ← -1
7:         best_j ← -1
8:         for i ← 0 to n-2 do
9:             for j ← i+1 to n-1 do
10:                gain ← CalculateTwoOptGain(i, j)
11:                if gain ¡ best_gain then
12:                    best_gain ← gain
13:                    best_i ← i
14:                    best_j ← j
15:                    improved ← true
16:                end if
17:            end for
18:        end for
19:        if improved then
20:            ReversePath(solution, best_i+1, best_j)
21:        end if
22:    end while
23:    return solution
24: end procedure
25: function CalculateTwoOptGain(i, j)
26:    old_dist ← Distance(tour[i], tour[i+1])
27:    old_dist ← old_dist + Distance(tour[j], tour[j+1])
28:    new_dist ← Distance(tour[i], tour[j])
29:    new_dist ← new_dist + Distance(tour[i+1], tour[j+1])
30:    return new_dist - old_dist
31: end function
```

6 Metaheuristics for the TSP

Metaheuristics represent a sophisticated evolution from classical heuristic approaches in solving the Traveling Salesman Problem (TSP). While traditional heuristics typically follow a fixed set of rules to construct or improve solutions, metaheuristics employ more dynamic and adaptive strategies to explore the solution space effectively.

The key distinctions between classical heuristics and metaheuristics are:

- **Search Strategy**: Classical heuristics often follow a deterministic path, while metaheuristics incorporate randomization and adaptive mechanisms to escape local optima.
- **Solution Quality**: Traditional heuristics may get stuck in local optima, whereas metaheuristics employ various techniques (like perturbation or memory structures) to find better solutions.
- **Computational Complexity**: Classical heuristics typically have lower computational requirements but may produce inferior solutions compared to metaheuristics.
- **Flexibility**: Metaheuristics can be adapted to various problem types and instances, while classical heuristics are often problem-specific.

Common metaheuristic approaches for the TSP include:

- **Tabu Search**: Uses memory structures to avoid revisiting recent solutions
- **Variable Neighborhood Search**: Systematically changes neighborhood structures to escape local optima
- **Simulated Annealing**: Employs probabilistic acceptance of worse solutions
- **Genetic Algorithms**: Evolves solutions through selection and recombination operations

These methods have proven particularly effective for large-scale TSP instances where exact methods become computationally infeasible.

6.1 Tabu Search

The Tabu search is a type of neighborhood search algorithm. Differently with what happened with greedy algorithms like the GRASP, the Tabu search algorithm is able to escape local optima by allowing the search to move to worse solutions, which can help the algorithm explore new regions of the search space and potentially find better solutions. The main idea behind the Tabu search heuristic is to exclude the already explored solutions from the search space. This is done by maintaining a tabu list that stores the solutions that have been visited recently. The tabu list is used to prevent the search from revisiting the same solutions, which can help the algorithm escape local optima and explore new regions of the search space[Nil03].

Algorithm The Tabu search algorithm can be summarized as follows:

1. **Initialization:** Initialize the tabu list and the best solution found.
2. **Setup:** Initialize the current solution and the current iteration count.
3. **Main Loop:** Repeat the following steps until a stopping criterion is met:
    - Generate a set of candidate solutions by applying a set of moves to the current solution.
    - Select the best candidate solution that is not in the tabu list.
    - Update the tabu list with the selected solution.
    - Update the current solution with the selected solution.
    - If the selected solution is better than the best solution found, update the best solution.
    - Increment the iteration count.
4. **Finalization:** Return the best solution found.

Algorithm 5 TSP Tabu Search

```
1: procedure TSP_TabuSearch(instance, initial_solution, tabu_size)
2:     best_solution ← initial_solution
3:     best_cost ← ComputeCost(best_solution)
4:     current_solution ← initial_solution
5:     tabu_list ← EmptyQueue(tabu_size)
6:     iterations ← 0
7:     while iterations ¡ MAX_ITERATIONS do
8:         best_move ← null
9:         best_gain ← ∞
10:        for all i, j ∈ PossibleMoves(current_solution) do
11:            if (i, j) ∉ tabu_list or AspirationCriteria() then
12:                gain ← CalculateSwapGain(i, j)
13:                if gain ¡ best_gain then
14:                    best_gain ← gain
15:                    best_move ← (i, j)
16:                end if
17:            end if
18:        end for
19:        ApplyMove(current_solution, best_move)
20:        current_cost ← ComputeCost(current_solution)
21:        AddToTabuList(tabu_list, best_move)
22:        if current_cost ¡ best_cost then
23:            best_solution ← current_solution
24:            best_cost ← current_cost
25:        end if
26:        iterations ← iterations + 1
27:    end while
28:    return best_solution
29: end procedure
```

6.2 Variable Neighbourhood Search

The Variable Neighbourhood Search (VNS) heuristic is a metaheuristic that combines local search with a systematic change of neighbourhood structures. The idea is to explore different neighbourhoods of the current solution to escape local optima and find better solutions. Many formulations and version for the VNS algorithm have been proposed in the literature, such as the *Basic VNS*, *Reduced VNS* and *General VNS.*[HMBP10]

All of these forumlations are based on a neighbourhood search, which is defined within a solution space that can be explored using the *two-opt move*, *three-opt move* or the *k-opt move*, depending on the general scheme of the algorithm. These operations that can be made in the solution space for the TSP problem allow to generate a neighborhood starting from a given solution, each of which neighbour differ from one single *2-opt move*[Nil03].

Algorithm In our imeplementation we will go over the General VNS scheme, in which the neighbours are simply generated from a single *two-opt move*, and the *shaking function* The VNS algorithm can be summarized as follows:

1. **Initialization:** Initialize the current solution and the best solution found.
2. **Setup:** Initialize the neighbourhood structure and the current iteration count.
3. **Main Loop:** Repeat the following steps until a stopping criterion is met:
    - Apply a local search algorithm to the current solution.
    - Generate a new solution by applying a perturbation to the current solution.

- If the new solution is better than the current solution, update the current solution.
- If the current solution is better than the best solution found, update the best solution.
- Change the neighbourhood structure.
- Increment the iteration count.

4. **Finalization:** Return the best solution found.

Algorithm 6 TSP Variable Neighborhood Search

```
1: procedure TSP_VNS(instance, initial_solution, k_max)
2:     best_solution ← initial_solution
3:     best_cost ← ComputeCost(best_solution)
4:     k ← 1
5:     while k ≤ k_max do
6:         current_solution ← best_solution
7:         // Shaking phase
8:         current_solution ← Shake(current_solution, k)
9:         // Local search phase
10:        improved ← true
11:        while improved do
12:            improved ← false
13:            neighbor ← LocalSearch(current_solution, k)
14:            neighbor_cost ← ComputeCost(neighbor)
15:            if neighbor_cost ¡ ComputeCost(current_solution) then
16:                current_solution ← neighbor
17:                improved ← true
18:            end if
19:        end while
20:        if ComputeCost(current_solution) ¡ best_cost then
21:            best_solution ← current_solution
22:            best_cost ← ComputeCost(current_solution)
23:            k ← 1
24:        else
25:            k ← k + 1
26:        end if
27:    end while
28:    return best_solution
29: end procedure
30: function SHAKE(solution, k)
31:    for i ← 1 to k do
32:        pos1, pos2 ← RandomPositions(solution.length)
33:        SwapNodes(solution, pos1, pos2)
34:    end for
35:    return solution
36: end function
```

6.3 Simulated Annealing

Simulated Annealing (SA) is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. The algorithm is designed to find an approximate solution to optimization problems by exploring the solution space in a way that allows for both exploration and exploitation. The key idea behind SA is to allow the algorithm to accept worse solutions with a certain probability, which helps it escape local optima and explore the solution space more effectively.

Algorithm The Simulated Annealing algorithm can be summarized as follows:

1. **Initialization:** Initialize the current solution, temperature, and cooling schedule.
2. **Main Loop:** Repeat the following steps until a stopping criterion is met:
   - Generate a new solution by applying a perturbation to the current solution.
   - Calculate the cost of the new solution.
   - If the new solution is better than the current solution, accept it.
   - If the new solution is worse than the current solution, accept it with a certain probability based on the temperature.
   - Update the temperature according to the cooling schedule.
3. **Finalization:** Return the best solution found.

Algorithm 7 Simulated Annealing for TSP

```
procedure SimulatedAnnealing(instance, initial_solution)
    current_solution ← initial_solution
    current_cost ← ComputeCost(current_solution)
    best_solution ← current_solution
    best_cost ← current_cost
    temperature ← initial_temperature
    cooling_rate ← 0.95
    max_iterations ← 1000
    for i = 1 to max_iterations do
        Generate a random valid TSP solution next_solution
        new_cost ← ComputeCost(next_solution)
        if new_cost < current_cost then
            current_solution ← next_solution
            current_cost ← new_cost
            if new_cost < best_cost then
                best_solution ← next_solution
                best_cost ← new_cost
            end if
        else
            delta ← new_cost − current_cost
            probability ← e^(−delta/temperature)
            random ← Random number between 0 and 1
            if random < probability then
                current_solution ← next_solution
                current_cost ← new_cost
            end if
        end if
        temperature ← temperature × cooling_rate
    end for
    return best_solution
end procedure
```

7 TSP with CPLEX

Given the popularity of the TSP problem in the mathematical optimization field and the growing necessity to solve larger and larger instances of the problem, optimzation softwares and libraries have started to establish themselves as a reliable approach to solve an instance of the tsp problem efficiently by exploting leveraging available computation power. One of the most popular and commonly used is CPLEX, a commercial optimization solver developed by IBM. CPLEX is a high-performance optimization solver developed by IBM that can be used to solve linear programming (LP), mixed-integer programming (MIP), and quadratic programming (QP) problems. It provides a powerful API that allows users to model and solve optimization problems in a variety of programming languages, including C, C++, Java, and Python.

7.1 History of CPLEX

CPLEX, named after the C programming language and the simplex method, was initially developed by Robert Bixby in 1988. It began as an implementation of the simplex algorithm for linear programming written in C. The software was first commercialized through CPLEX Optimization Inc., founded by Bixby.

In 1997, ILOG acquired CPLEX Optimization Inc., leading to significant expansions in the solver's capabilities. Under ILOG, CPLEX evolved to handle more complex problem types including mixed-integer programming, quadratic programming, and constraint programming.

IBM acquired ILOG in 2009, rebranding the software as IBM ILOG CPLEX Optimization Studio. Under IBM's stewardship, CPLEX has continued to evolve with enhanced performance, parallel processing capabilities, and integration with modern programming environments.

Today, CPLEX remains one of the leading commercial optimization solvers, widely used in both industry and academia for solving large-scale mathematical programming problems. In our case we will be using the C API to model and solve the TSP problem using CPLEX. The CPLEX C API provides a set of functions that allow users to create and manipulate optimization models, set parameters, and solve the models.

7.2 Modeling the TSP with CPLEX

The main goal here is to exploit the power of the CPLEX library to solve instances of the TSP problem as efficiently as possibile. To model the TSP with CPLEX we first start from its MIP formulation:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{7}$$

$$\text{subject to: } \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{8}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in \{1, \ldots, n\} \tag{9}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset \tag{10}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in \{1, \ldots, n\} \tag{11}$$

where:

- x_{ij} is a binary variable equal to 1 if edge (i, j) is in the tour, 0 otherwise
- c_{ij} is the cost of edge (i, j)
- S represents any proper subset of nodes
- The third constraint eliminates subtours

Therefore, in order to solve a TSP instance using CPLEX we first need to model the problem using the CPLEX API. The CPLEX API provides a set of functions that allow us to create and manipulate optimization models, set parameters, and solve the models. The following sections will go over the main steps to model and solve a TSP instance using CPLEX, in which for each paragraph we will also reference the CPLEX API documentation, along with a code snippet directly from the implementation.

CPLEX Environment The first step in using CPLEX is to create an environment object that will be used to manage the optimization process. The environment object is created using the `CPXopenCPLEX` function, which returns a pointer to the CPLEX environment. [IBM22]

```
CPXENVptr env = CPXopenCPLEX(&status);
```

Decision Variables Decision variables are the unknowns in your optimization problem that CPLEX needs to determine. In the TSP context:

- Each x_ij variable represents whether edge (i, j) is included in the tour
- Binary variables (0 or 1) where:
  - $x_ij = 1$ means edge (i, j) is in the tour
  - $x_ij = 0$ means edge (i, j) is not in the tour

To define decision variables in the CPLEX environment we use the `CPXnewcols` function, which creates a set of new columns (variables) in the model. Each variable represents an edge in the graph and is binary (0 or 1) to indicate whether the edge is included in the tour.

```
int num_edges = inst->nnodes * inst->nnodes;
double* lb = (double*)malloc(num_edges * sizeof(double));
double* ub = (double*)malloc(num_edges * sizeof(double));
char* xctype = (char*)malloc(num_edges * sizeof(char));
char** colnames = (char**)malloc(num_edges * sizeof(char*));
for (int i = 0; i < num_edges; i++) {
  lb[i] = 0.0;
  ub[i] = 1.0;
  xctype[i] = 'B';
  colnames[i] = (char*)malloc(100 * sizeof(char));
  sprintf(colnames[i], "x_%d_%d", i / inst->nnodes, i % inst->nnodes);
}
status = CPXnewcols(env, lp, num_edges, NULL, lb, ub, xctype, colnames);
```

Constraints Constraints are the conditions that the solution must satisfy. In the TSP context:

- Each node must be visited exactly once
- The subtour elimination constraints ensure that the solution is a valid tour

To add constraints to the CPLEX model, we use the `CPXaddrows` function, which adds a set of rows (constraints) to the model.

```
int error = CPXaddrows(env, lp, 0, 1, non_zero_variables_count, &
  right_hand_side_value, &constraint_sense,&izero,index,coefficients, NULL, &rname);
```

To the `CPLXaddrows` function we pass the usual arguments spcifying the CPLEX environment and pointer, along with the following arguments:

- `non_zero_variables_count`: The number of non-zero variables in the constraints
- `right_hand_side_value`: The array representing right-hand side value of the constraints
- `constraint_sense`: The sense of the constraints (e.g., less than or equal to, greater than or equal to, equal to)
- `izero`: The index of the first variable in the constraint
- `index`: The array of variable indices in the constraint

- `coefficients`: The array of coefficients for each variable in the constraint
- `rname`: The name of the constraint

With the steps listed above we have successfully modeled a generic MIP problem into CPLEX, but this does not correctly represent the TSP problem. For that we still need to model the subtour elimination constraints into the model.

7.3 Bender's subtour elimination method

Up until this point we have just modelled a generic integer linear problem into cplex, but this does not solve the TSP problem. In order to have CPLEX return feasible solutions to the TSP problem, we need to add constraints that eliminate subtours in the solution. This can be done using the Bender's subtour elimination method, which is a cutting-plane algorithm that adds constraints to the model to eliminate subtours.

Subtour Elimination Constraints The subtour elimination constraints are defined as follows:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subset V, 2 \leq |S| \leq |V| - 1 \tag{12}$$

where V is the set of nodes in the graph and S is a subset of nodes that forms a subtour.

Implementation The implementation of the Bender's subtour elimination method involves adding the subtour elimination constraints to the model and solving the model iteratively until no subtours are found. This is can be done in two ways:

1. **Iterative Approach:** Solve the model, check for subtours in the solution, and add constraints for each subtour found. This process continues until a solution with no subtours is obtained. The steps are:
   - Solve the initial TSP model
   - Find connected components in the solution graph
   - If multiple components exist, add subtour elimination constraints
   - Repeat until only one component remains
2. **CPLEX Callback Approach:** Utilize CPLEX's lazy constraint callback mechanism to add subtour elimination constraints during the optimization process. This approach:
   - Registers a callback function with CPLEX
   - CPLEX calls this function whenever it finds a new integer feasible solution
   - The callback checks for subtours and adds necessary constraints immediately
   - More efficient as it integrates with CPLEX's branch-and-cut framework

Iterative Approach Here is the implementation of the iterative approach to the Bender's subtour elimination method:

1. **Setup**:
   - The function initializes the current node index to the starting node (which is passed as an argument) and sets the remaining nodes count to the total number of nodes.
   - It allocates memory for the solution array and remaining nodes arrays. These two arrays will be used respectively to store the tour and the remaining nodes for the main loop to visit.
   - The solution array is initialized with `-1` to indicate that no nodes have been visited yet.
   - The remaining nodes array is initialized with all node indices.

```
    clock_t start_time = clock();
    int current_node_index = starting_node;
    int remaining_nodes_count = inst->nnodes;
    solution nearest_node;
    int* remaining_nodes = (int*)malloc(inst->nnodes * sizeof(int));
    inst->solution = (int*)malloc(inst->nnodes * sizeof(int));
    inst->best_cost_value = 0;
    for (int i = 0; i < inst->nnodes; i++) {
      inst->solution[i] = -1;
      remaining_nodes[i] = i;
    }
    remaining_nodes[current_node_index] = remaining_nodes[--
  remaining_nodes_count];

```

2. **Main Loop**:
   - The function iterates over all nodes to construct the solution.
   - For each node, it finds the nearest unvisited node using the `euclidean_nearest_node` function.
   - If no nearest node is found (i.e., all nodes are visited), it connects the current node back to the starting node to complete the tour.
   - Otherwise, it updates the solution with the nearest node and logs the current node index and nearest node found.
   - The current node index is updated to the nearest node for the next iteration.

```
    do
    {
      xstar = TSPopt(instance, env, lp);
      init_data_struct(instance, &component_map, &succ, &ncomp);
      build_solution(xstar, instance, solution, component_map, ncomp);
      error = add_bender_constraint(env, lp, NULL, component_map, instance, *
  ncomp);
    }while (*ncomp > 1);

```

3. **Finalization**:
   - After constructing the solution, the function records the end time and calculates the elapsed time.
   - It logs the total time taken and the cost of the solution.
   - Finally, it frees the allocated memory for the remaining nodes array.

```
    clock_t end_time = clock();
    double elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
    inst->elapsed_time = elapsed_time;
    inst->best_cost_value = compute_solution_cost(inst, inst->solution);

```

CPLEX Callback Approach Here is the implementation of the CPLEX callback approach to the Bender's subtour elimination method:

1. **Setup**:
   - Initialize the CPLEX environment and the TSP instance data structure.
   - Create and build the TSP CPLEX model.
   - Register the lazy constraint callback function with CPLEX using the `CPXcallbacksetfunc` function provided by the CPLEX API.

```
    int error = 0;
    CPXENVptr env = CPXopenCPLEX(&error);
    if (error) print_error("CPXopenCPLEX() error");
    CPXLPptr lp = CPXcreateprob(env, &error, "TSP model version 1");
    if (error) print_error("CPXcreateprob() error");
    double lower_bound = -CPX_INFBOUND;
    double upper_bound = CPX_INFBOUND;

    CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);
    if (_verbose >= 60) CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_ON);
    CPXsetintparam(env, CPX_PARAM_RANDOMSEED, 1);
    CPXsetdblparam(env, CPX_PARAM_TILIM, 36000);
    CPXsetintparam(env, CPX_PARAM_CUTUP, upper_bound);

    build_model(instance, env, lp);
    if (contextid == NULL) contextid = CPX_CALLBACKCONTEXT_CANDIDATE;
    if (CPXcallbacksetfunc(env, lp, contextid, callback_driver, instance))
    print_error("CPXcallbacksetfunc() error");

```

2. **Lazy Constraint Callback Function**:
    - The lazy constraint callback function is called by CPLEX whenever a new integer feasible solution is found. (Note: this solution may not be a valid TSP tour);
    - The solution is built and checked for subtours using the `build_solution` function.
    - The function checks the number of connected components in the solution and adds necessary constraints to eliminate them.
    - It uses the `CPXcutcallbackadd` function to add the subtour elimination constraints to the model.
    - The callback function returns control to CPLEX after adding the constraints.

```

```

3. **Main Loop**:
    - Solve the model using CPLEX and the lazy constraint callback function.
    - CPLEX will call the callback function whenever a new integer feasible solution is found.
    - The callback function will add subtour elimination constraints to the model.
    - Repeat until CPLEX returns a valid TSP solution, that is a solution such that the number of subtours is equal to 1.
4. **Finalization**:
    - After solving the model, the function records the end time and calculates the elapsed time.
    - It logs the total time taken and the cost of the solution.
    - Finally, it frees the allocated memory for the solution and the TSP instance data structure.

7.4 Patching Heuristic

8 Mathheuristics

In the field of Mathematical optimization, one of the most tackled issues is the efficiency and the performance with which we are able to solve MIP problems. With time, both solver and the hardware have evolved to the point where we are able to solve even some large scale problems in a reasonable amount of time. However, there are still some problems that are too large to be solved in a reasonable amount of time, and this is where the Mathheuristics come into play. Mathheuristics are a combination of mathematical optimization techniques and heuristics that are used to improve the efficiency and performance of already existing mathematical optimization algorithms.[FL03][FF16]

Over the next sections we will go over a brief introduction to General-Purpose MIP Heuristics and the implementation of the following Mathheuristics algorithms:

- **Hard Fixing**
- **Local Branching**

General-Purpose MIP-Based Heuristics General-Purpose MIP-Based Heuristics form the basis for the Mathheuristics algorithm family, their relevance are jsutified by two main reasons:

- They function as vital components for efficiently solving specialized subproblems that emerge during the application of matheuristic approaches to complex optimization challenges.
- They demonstrate the significant advantages that general-purpose MIP solvers can gain by incorporating metaheuristic concepts such as local search methodologies and evolutionary algorithms.

let us then define the genral form of a MIP:

$$\min c^T x \tag{13}$$
$$Ax \leq b \tag{14}$$
$$x_j \in \{0,1\}, \forall j \in \mathcal{B} \tag{15}$$
$$x_j \text{integer}, \forall j \in \mathcal{G} \tag{16}$$
$$x_j \text{continuous}, \forall j \in \mathcal{C} \tag{17}$$

where A is an $m \times n$ input matrix and b and c are input vectors of dimension m and n, respectively. Here, the variable index set $N = \{1, \ldots, n\}$ is partitioned into (B, G, C), where B is the index set of the binary variables (if any), while sets G and C index the general integer and the continuous variables, respectively.

Mathheuristics algorithms are meant ot be used in concatenation with a "black box" solver, meaning that these algorithms are not directly implemented directly into the solver, but thery are applied onto the input instance that is about to be fed into the solver. The idea is to improve the overall performance of the solver by optimizing the search space for solver to explore.[FF16]

8.1 Hard Fixing Mathheuristic

The idea behind Fixed Branching is to optimize the solution space search by adding a set of constraints to the model that restrict the search space to a local region around the current solution. These constraints are designed to select a random subset of edges in the current solution, so as to create a restricted neighborhood of feasible solutions. This allows the solver to explore a smaller search space and potentially find better solutions faster.

Algorithm The Hard Fixing algorithm can be summarized as follows:

1. fix a small random probability $\rho \in (0,1)$ (i.e. for the implementation we went for $\rho = 0.3$);
2. extract the subset of edges to be fixed based on the probability ρ: these edges are again selected at random from the subset of edges that are not selected in the current solution;

$$\bar{\mathcal{S}} = \{(i,j) \in \mathcal{S} : (i,j) = 0, |\bar{\mathcal{S}}| = \rho|\mathcal{B}|\} \tag{18}$$

where $\mathcal{S} \subseteq \mathcal{B}$ and $\mathcal{B}$ is the set of binary variables. To put it in words, we want to select a subset of edges that are not selected in the current solution, and we want to fix them to 1 with a probability ρ.

3. add constraints to the model to fix the selected subset of edges to 1.

This process is repeated every time a new integer feasible solution is found, and the constraints are added to the model to restrict the search space. It is implicitly understood that the local branching constraints are reset at every iteration of the algorithm.

Implementation The algorithm is implemented using the CPLEX Callback functionality, which allows us to add constraints to the model during the optimization process. Every time a new integer feasible solution is found, the callback function is called, and we can add constraints to the model to restrict the search space. In the callback function, the cplex_hard_fixing function is called, which fixes a subset of variables in the model based on a given probability p_fix.

The function does the following:

1. **Update incumbent**: Get the current solution from the callback context
2. **Restore to original instance**: Reset the bounds of all variables to their original values (0.0 to 1.0).
3. Fix a subset of variables based on the given probability p_fix.
4. Apply the fixing by adding constraints to the model.

8.2 Local Branching

As for the Hard fixing heuristic, the Local branching algorihtm is meant to be used with a heuristic black TSP solver to optimize its performance. The goal of the Local Branching algorithm is to reduce the solution search space the black solver has to explore by "branching" from a given TSP solution $\bar{x}$ to a restricted neighborhood of $\bar{x}$. The neighborhood is defined analougously to what we did with the VNS or the Tabu Search algorithm, where we adopted the idea of the *k-opt Neighbourhood*: we therefore need to set a hyperparameter k that determines the size of our neighborhood[FL03].

Algorithm Starting from a given solution $\hat{x}$, we add the following constraints to the model:

$$\Delta(x, \hat{x}) := \sum_{j \in \mathcal{S}} (1 - x_j) + \sum_{j \in \mathcal{B} \setminus \mathcal{S}} x_j \leq k \tag{19}$$

where $\mathcal{B}$ is the index set of the binary variables, and $\mathcal{S} := \{j \in \mathcal{B} : \hat{x}_j = 1\}$[FL03]. At each iteration, we then compute the set $\mathcal{S}$ we then compute 19, add it to the model and solve the model. If the solution is feasible, we update the best solution found and repeat the process until a stopping criterion is met.

Implementation Starting from a given solution $\bar{x}$, and given the starting tsp instance, the Cplex programming environment and pointer, as shown in the function declaration below:

```
1  int add_local_branching_constraint(instance* inst, CPXENVptr env, CPXLPptr lp,
     CPXCALLBACKCONTEXTptr cpxcallbackcontex_tptr, const double* xstar, double k)
```

we proceed with the floowing steps:

1. **Constraint Modeling**: Starting from the given solution vector *xstar*, we build the constraint the will be later added to the CPLEX model.

```
1    int ncols = CPXgetnumcols(env, lp);
2    int* index = (int*)calloc(ncols, sizeof(int));
3    double* coefficients = (double*)calloc(ncols, sizeof(double));
4    double right_hand_side_value = k;
5    char constraint_sense = 'G';
6    char* rname = (char*)calloc(100, sizeof(char));
7    sprintf(rname, "local_branching_constraint");
```

```
8
9     int non_zero_variables_count = 0;
10    for (int i = 0; i < ncols; i++)
11    {
12      if (xstar[i] > 0.8)
13      {
14        index[non_zero_variables_count] = i;
15        coefficients[non_zero_variables_count] = 1.0;
16        non_zero_variables_count++;
17      }
18    }
19
20    right_hand_side_value = inst->nnodes - k;
21
```

2. **Constraint Addition**: We then add the constraint to the model: this can be done directly by adding a row directing into the model:

```
1     CPXaddrows(env, lp, 0, 1, non_zero_variables_count, &right_hand_side_value, &
      constraint_sense,&izero,index,coefficients, NULL, &rname);
2
```

   or in the case we are using the CPLEX callback function, we can use the `CPXcallbackaddusercuts` function to add the constraint to the model

```
1     CPXcallbackrejectcandidate(cpxcallbackcontex_tptr, 1, non_zero_variables_count
      ,&right_hand_side_value,&constraint_sense, &izero, index, coefficients);
2
```

3. **Model Resolution**: We then solve the model and check if the solution is feasible. If the solution is feasible, we update the best solution found, remove the latest local Branching constrint (if necessary), and repeat the process until a stopping criterion is met.

Note The 19 in CPLEX needs to be expressed using a particular form: we need to express the constraint to be expressed as an inequality of the form $Ax \leq b$, where A is a matrix of coefficients, x is the vector of variables and b is the right-hand side of the inequality. In our case, we can express the constraint as follows:

$$\sum_{j \in \mathcal{S}} (1 - x_j) \leq k \tag{20}$$

$$\Rightarrow \sum_{j \in \mathcal{S}} 1 - \sum_{j \in \mathcal{S}} x_j \leq k' \tag{21}$$

$$\Rightarrow - \sum_{j \in \hat{\mathcal{S}}} x_j \leq k' - \sum_{j \in \mathcal{S}} 1 \tag{22}$$

$$\Rightarrow \sum_{j \in \mathcal{S}} x_j \geq |\mathcal{S}| - k' \tag{23}$$

References

[FF16] Martina Fischetti and Matteo Fischetti. Matheuristics. 2016.

[FL03] Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical Programming, Series B*, 98:23–47, 2003.

[HMBP10] Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez. Variable neighborhood search. 146:61–86, 2010.

[IBM22] IBM. Initialize the cplex environment. CPLEX Documentation, 2022. Version 22.1.0.

[Nil03] Christian Nilsson. Heuristics for the traveling salesman problem. 2003.