Operations Reserach 2 Final Thesis

Youssef Ben Khalifa

March 18, 2025

1 Introduction

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 The work done is subdivided into two main categories:

1. **Heuristics**: Implementation of common heuristic algorithms including:
   - Greedy Randomized Adaptive Search (GRASP)
   - Extra Mileage
   - Tabu Search
   - Variable Neighborhood Search (VNS)
2. **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.

The thesis is organized as follows:

- First, we present the mathematical formulation of the TSP problem and the data structures used
- Next, we discuss each heuristic algorithm implementation in detail
- Then, we explore the integration with CPLEX and mathheuristic approaches
- Finally, we analyze the performance and effectiveness of each method

1.1 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.

Mathematical Formulation The TSP can be formulated as an optimization problem as follows:

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

subject to:

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

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1, \dots, n\} \tag{3}$$

$$u_i - u_j + n x_{ij} \leq n - 1 \quad \forall i \in \{2, \dots, n\}, j \in \{2, \dots, n\} \tag{4}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in \{1, \dots, n\} \tag{5}$$

$$u_i \in \mathit{text} \quad \forall i \in \{1, \dots, n\} \tag{6}$$

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.

2 TSP data structure

Throught the implementation of the Metaheuristics and Mathheuristics, we will be using the following data structures:

- instance : A structure that holds the TSP instance data. This is implemented as a C data structure in which we hold all the metadata relevant to the loaded TSP instance.
- solution : A structure that holds the TSP solution data. Normally this is represented as a simple integer array in which we map for every nodex index its successor in the tour. This type of solution representation suits both the indirected and directed graph cases.

The code for the data structures implementation can be found in the appendinx section.

The cost functio used is the euclidean distance between two nodes in the graph. The code for the implementation can be found in the appendix section **??**.

3 TSP Heurisitcs

3.1 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).

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.

```
void tsp_grasp(instance* inst, int starting_node);
```

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.

```
    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.

```c
    for (int i = 0; i < inst->nnodes; i++) {
      nearest_node = euclidean_nearest_node(inst, current_node_index,
  remaining_nodes, &remaining_nodes_count);
      if (nearest_node.node == -1) {
        inst->solution[current_node_index] = starting_node;
        inst->best_cost_value += euclidean_distance(inst->xcoord[
  current_node_index],
                                inst->ycoord[current_node_index],
                                inst->xcoord[starting_node],
                                inst->ycoord[starting_node], false);
        continue;
      }
      inst->solution[current_node_index] = nearest_node.node;
      inst->best_cost_value += nearest_node.cost;
      current_node_index = nearest_node.node;
      }

```

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.

```c
    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);

```

The complete code of the imeplementation can be found in the appendix sectionA.1.

3.2 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 imeplementation takes as input arguments:

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

```c
void tsp_extra_mileage(instance* inst, pair starting_pair);
```

where the *pair* data structure is defined as follows:

```c
typedef struct {
  int node1;
  int node2;
} pair;
```

An heuristic state data stcuture is also used to keep track of the covered nodes and the final solution.

```c
typedef struct
{
  int covered_nodes_count;
  int* covered_nodes;
  int uncovered_nodes_count;
  int* uncovered_nodes;
} heuristic_state;
```

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

```c
    clock_t start_time = clock();
    heuristic_state state;
    pair current_pair = starting_pair;
    initialize_instance(inst, &state);
    inst->solution[current_pair.node1] = current_pair.node2;
    inst->solution[current_pair.node2] = current_pair.node1;
    state.covered_nodes[state.covered_nodes_count++] = current_pair.node1;
    state.covered_nodes[state.covered_nodes_count++] = current_pair.node2;
    state.uncovered_nodes[current_pair.node1] = state.uncovered_nodes[--state.
    uncovered_nodes_count];
    state.uncovered_nodes[current_pair.node2] = state.uncovered_nodes[--state.
    uncovered_nodes_count];

```

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.

```c
      while (state.covered_nodes_count < inst->nnodes)
      {
        log_message(LOG_LEVEL_INFO, "Covered nodes count: %d\n", state.
    covered_nodes_count);
        solution best_node;
        for (int i = 0; i < state.covered_nodes_count; i++)
        {
          log_message(LOG_LEVEL_INFO, "Current node index: %d\n", i);
          int current_node = state.covered_nodes[i];
          int current_node_opposite = inst->solution[current_node];
          double min_distance_delta = INFINITY;
          best_node.node = -1;
          best_node.node_index = -1;
          for (int j = 0; j < state.uncovered_nodes_count; j++)
          {
            double distance1 = euclidean_distance(inst->xcoord[current_node],
                                 inst->ycoord[current_node],
                                 inst->xcoord[state.uncovered_nodes[j]],
                                 inst->ycoord[state.uncovered_nodes[j]], false);
            double distance2 = euclidean_distance(inst->xcoord[state.
    uncovered_nodes[j]],
                                 inst->ycoord[state.uncovered_nodes[j]],
                                 inst->xcoord[current_node_opposite],
                                 inst->ycoord[current_node_opposite], false);
            double existing_pair_distance = euclidean_distance(inst->xcoord[
    current_node],
                                    inst->ycoord[current_node],
                                    inst->xcoord[current_node_opposite],
                                    inst->ycoord[current_node_opposite], false);

            double distance_delta = distance1 + distance2 - existing_pair_distance
    ;

            if (distance_delta < min_distance_delta)
            {
              min_distance_delta = distance_delta;
              best_node.node = state.uncovered_nodes[j];
              best_node.node_index = j;
              best_node.cost = distance_delta;
            }
          }
          if (best_node.node > -1)
          {
            log_message(LOG_LEVEL_INFO, "Best node found: %d\n", best_node.node);
            inst->solution[current_node] = best_node.node;
            inst->solution[best_node.node] = current_node_opposite;
            state.covered_nodes[state.covered_nodes_count++] = best_node.node;
            state.uncovered_nodes[best_node.node_index] = state.uncovered_nodes[--
    state.uncovered_nodes_count];
          }
        }
        }

```

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.

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

```

the code for the implementation can be found in the appendix section A.2.

3.3 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].

3.3.1 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).
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 $(i, j), (h, k) \in T$ such that either:

   - $c_{ij} + c_{hk} > c_{ih} + c_{jk}$

- $\Delta((i,j),(h,k)) = C_T()$

2. Replacing the two edges with the edges (i,h) and (j,k).

4 Neighbourhood Search Heauristics

4.1 Tabu Search

The Tabu search is a type of neighborhood search algorithm. These type of algorithms are base on a search strategy that explore the solution search space by applying the two-opt move to the current solution. 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.

```
    int** tabu_list = (int**)malloc(TABU_TENURE * sizeof(int*));
    for (int i = 0; i < TABU_TENURE; i++)
    {
      tabu_list[i] = (int*)malloc(size * sizeof(int));
    }
    int tabu_index = 0;

```

2. **Setup:** Initialize the current solution and the current iteration count.

```
    Solution best_solution;
    best_solution.solution = (int*)malloc(size * sizeof(int));
    memcpy(best_solution.solution, initial_solution, size * sizeof(int));
    best_solution.cost = compute_solution_cost(inst, best_solution.solution);
    Solution current_solution = best_solution;

```

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.

```
        int num_neighbors = size;
        int** neighbours = (int**)malloc(num_neighbors * sizeof(int*));
        for (int i = 0; i < num_neighbors; i++)
        {
          neighbours[i] = (int*)malloc(size * sizeof(int));
        }
        generate_neighbors(current_solution.solution, size, neighbours,
    num_neighbors);

```

   - Select the best candidate solution that is not in the tabu list.

```
        Solution best_neighbor;
        best_neighbor.solution = NULL;
        best_neighbor.cost = INT_MAX;

        for (int i = 0; i < num_neighbors; i++)
        {
          int cost = evaluate_solution(neighbours[i], size);
          if (cost < best_neighbor.cost && !is_tabu(neighbours[i], tabu_list,
     TABU_TENURE, size))
          {
            best_neighbor.solution = neighbours[i];
            best_neighbor.cost = cost;
          }
        }

```

- Update the tabu list with the selected solution.

```
        if (best_neighbor.solution != NULL)
        {
          current_solution = best_neighbor;
          if (current_solution.cost < best_solution.cost)
          {
            best_solution = current_solution;
          }
          add_to_tabu_list(current_solution.solution, tabu_list, &tabu_index, size);
        }

```

- 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.

```
inst->best_cost_value = compute_solution_cost(inst, best_solution.solution);
log_message(LOG_LEVEL_INFO, "Best solution cost: %d\n", best_solution.cost);
memcpy(inst->solution, best_solution.solution, size * sizeof(int));

clock_t end_time = clock();
double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
inst->elapsed_time = elapsed_time;
log_message(LOG_LEVEL_INFO, "Tabu Search solution time: %f seconds\n", elapsed_time);
log_message(LOG_LEVEL_INFO, "Tabu Search solution cost: %f\n", best_solution.cost);

```

4.2 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.

```
int* current_solution = (int*)malloc(inst->nnodes * sizeof(int));
memcpy(current_solution, initial_solution, inst->nnodes * sizeof(int));
bool stop_criterion = true;
int iterations_without_improvement = 0;
const int max_iterations = 100;
const int time_limit_milliseconds = 3600;
const int max_stuck_iterations = 10;
double best_solution_cost = compute_solution_cost(inst, current_solution);

```

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.

5 TSP with CPLEX

5.1 Brief introduction to CPLEX

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.

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.

5.2 Modeling the TSP with CPLEX

To model the TSP with CPLEX, we need to define the decision variables, constraints, and objective function of the problem. The decision variables represent the edges of the graph, and the constraints ensure that each node is visited exactly once in the tour. The objective function is to minimize the total cost of the tour.

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.

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

Decision Variables 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);
```

5.3 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:

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1 \quad \forall S \subset V, 2 \leq |S| \leq |V| - 1 \tag{7}$$

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.

5.4 Patching Heuristic

6 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]
In this section we will go over the implementation of the following Mathheuristics:

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

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]

6.1 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;

   $$\bar{\mathcal{S}} = \{(i, j) \in \mathcal{S} : (i, j) = 0, |\bar{\mathcal{S}}| = \rho|\mathcal{B}|\} \tag{8}$$

   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

```
    int error = CPXcallbackgetcandidatepoint(context, xstar, 0, ncols - 1, NULL);

```

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.

```
int cplex_hard_fixing(instance* instance, CPXCALLBACKCONTEXTptr context, double p_fix)
{
    int ncols = instance->ncols;
    double* xstar = (double*)calloc(ncols, sizeof(double));
    int* indices = (int*)calloc(ncols, sizeof(int));
    double* bd = (double*)calloc(ncols, sizeof(double));
    char* lu = (char*)calloc(ncols, sizeof(char));

    // Initialize random seed
    srand(time(NULL));

    // Get the current solution
    int error = CPXcallbackgetcandidatepoint(context, xstar, 0, ncols - 1, NULL);
    if (error)
    {
        free(xstar);
        free(indices);
        free(bd);
        free(lu);
        return error;
    }

    // Reset the bounds of all variables to their original values (0.0 to 1.0)
    for (int i = 0; i < ncols; i++)
    {
        indices[i] = i;
        lu[i] = 'B'; // Both lower and upper bounds
        bd[i] = 0.0; // Lower bound
    }
    error = CPXcallbackpostheursoln(context, ncols, indices, bd, 0.0,
    CPXCALLBACKSOLUTION_PROPAGATE);
    if (error)
    {
        free(xstar);
        free(indices);
        free(bd);
        free(lu);
        return error;
    }

    for (int i = 0; i < ncols; i++)
    {
        bd[i] = 1.0; // Upper bound
    }

    error = CPXcallbackpostheursoln(context, ncols, indices, bd, 0.0,
    CPXCALLBACKSOLUTION_PROPAGATE);
    if (error)
    {
        free(xstar);
        free(indices);
        free(bd);
        free(lu);
        return error;
    }

    // Fix a subset of variables based on the given probability
    int fix_count = 0;
    for (int i = 0; i < ncols; i++)
    {
        if (((double)rand() / RAND_MAX) < p_fix)
        {
            indices[fix_count] = i;
            lu[fix_count] = 'L'; // Only lower bound
            bd[fix_count] = 1.0; // Fix to 1
            fix_count++;
        }
    }

    // Apply the fixing
    if (fix_count > 0)
    {
        error = CPXcallbackpostheursoln(context, fix_count, indices, bd, 0.0,
    CPXCALLBACKSOLUTION_PROPAGATE);
        if (error)
        {
            free(xstar);
            free(indices);
            free(bd);
            free(lu);
            return error;
        }
    }

    free(xstar);
    free(indices);
    free(bd);
    free(lu);
    return error;
}
```

6.2 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:

$$\Delta(x, \hat{x}) := \sum_{j \in \mathcal{S}} (1 - x_j) + \sum_{j \in \mathcal{B} \setminus \mathcal{S}} x_j \leq k \tag{9}$$

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 9, 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:

```
 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.

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

    int non_zero_variables_count = 0;
    for (int i = 0; i < ncols; i++)
    {
      if (xstar[i] > 0.8)
      {
        index[non_zero_variables_count] = i;
        coefficients[non_zero_variables_count] = 1.0;
        non_zero_variables_count++;
      }
    }

    right_hand_side_value = inst->nnodes - k;

```

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

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

```

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

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

```

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 9 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:

$$\sum_{j\in\mathcal{S}}(1-x_j) \leq k \tag{10}$$

$$\Rightarrow \sum_{j\in\mathcal{S}} 1 - \sum_{j\in\mathcal{S}} x_j \leq k' \tag{11}$$

$$\Rightarrow -\sum_{j\in\hat{\mathcal{S}}} x_j \leq k' - \sum_{j\in\mathcal{S}} 1 \tag{12}$$

$$\Rightarrow \sum_{j\in\mathcal{S}} x_j \geq |\mathcal{S}| - k' \tag{13}$$

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.

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

A Metaheuristics

A.1 GRASP

```c
void tsp_grasp(instance* inst, int starting_node) {
    clock_t start_time = clock();
    log_message(LOG_LEVEL_INFO, "Solving TSP with GRASP\n");
    log_message(LOG_LEVEL_INFO, "Instance details: nnodes = %d\n", inst->nnodes);
    for (int i = 0; i < inst->nnodes; i++) {
        log_message(LOG_LEVEL_INFO, "Node %d: (%f, %f)\n", i, inst->xcoord[i],
inst->ycoord[i]);
    }
    log_message(LOG_LEVEL_INFO, "Starting GRASP with starting node %d\n",
starting_node);

    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;

    // Initialize solution
    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
];

    for (int i = 0; i < inst->nnodes; i++) {
        log_message(LOG_LEVEL_INFO, "Current node index: %d\n", current_node_index
);
        nearest_node = euclidean_nearest_node(inst, current_node_index,
remaining_nodes, &remaining_nodes_count);

        if (nearest_node.node == -1) {
            inst->solution[current_node_index] = starting_node;
            inst->best_cost_value += euclidean_distance(inst->xcoord[
current_node_index],
                                                        inst->ycoord[
current_node_index],
                                                        inst->xcoord[starting_node
],
                                                        inst->ycoord[starting_node
], false);
            continue;
        }
        inst->solution[current_node_index] = nearest_node.node;
        inst->best_cost_value += nearest_node.cost;
        current_node_index = nearest_node.node;
    }

    clock_t end_time = clock();
    double elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
    inst->elapsed_time = elapsed_time;
    log_message(LOG_LEVEL_INFO, "GRASP solution time: %f seconds\n", elapsed_time)
;
    log_message(LOG_LEVEL_INFO, "GRASP solution cost: %f\n", inst->best_cost_value
);

    free(remaining_nodes);
}
```

A.2 Extra Mileage

```c
void tsp_extra_mileage(instance* inst, pair starting_pair)
{
    clock_t start_time = clock();
    log_message(LOG_LEVEL_INFO, "Solving TSP with Extra Mileage\n");
    log_message(LOG_LEVEL_INFO, "Instance details: nnodes = %d\n", inst->nnodes);
    for (int i = 0; i < inst->nnodes; i++) {
        log_message(LOG_LEVEL_INFO, "Node %d: (%f, %f)\n", i, inst->xcoord[i], inst->ycoord[i]);
    }

    heuristic_state state;
    pair current_pair = starting_pair;

    initialize_instance(inst, &state);

    inst->solution[current_pair.node1] = current_pair.node2;
    inst->solution[current_pair.node2] = current_pair.node1;
    state.covered_nodes[state.covered_nodes_count++] = current_pair.node1;
    state.covered_nodes[state.covered_nodes_count++] = current_pair.node2;
    state.uncovered_nodes[current_pair.node1] = state.uncovered_nodes[--state.uncovered_nodes_count];
    state.uncovered_nodes[current_pair.node2] = state.uncovered_nodes[--state.uncovered_nodes_count];

    while (state.covered_nodes_count < inst->nnodes)
    {
        log_message(LOG_LEVEL_INFO, "Covered nodes count: %d\n", state.covered_nodes_count);
        solution best_node;
        for (int i = 0; i < state.covered_nodes_count; i++)
        {
            log_message(LOG_LEVEL_INFO, "Current node index: %d\n", i);
            int current_node = state.covered_nodes[i];
            int current_node_opposite = inst->solution[current_node];
            double min_distance_delta = INFINITY;
            best_node.node = -1;
            best_node.node_index = -1;
            for (int j = 0; j < state.uncovered_nodes_count; j++)
            {
                double distance1 = euclidean_distance(inst->xcoord[current_node],
                                                      inst->ycoord[current_node],
                                                      inst->xcoord[state.uncovered_nodes[j]],
                                                      inst->ycoord[state.uncovered_nodes[j]], false);
                double distance2 = euclidean_distance(inst->xcoord[state.uncovered_nodes[j]],
                                                      inst->ycoord[state.uncovered_nodes[j]],
                                                      inst->xcoord[current_node_opposite],
                                                      inst->ycoord[current_node_opposite], false);
                double existing_pair_distance = euclidean_distance(inst->xcoord[current_node],
                                                                   inst->ycoord[current_node],
                                                                   inst->xcoord[current_node_opposite],
                                                                   inst->ycoord[current_node_opposite], false);

                double distance_delta = distance1 + distance2 - existing_pair_distance;

                if (distance_delta < min_distance_delta)
                {
                    min_distance_delta = distance_delta;
                    best_node.node = state.uncovered_nodes[j];
                    best_node.node_index = j;
                    best_node.cost = distance_delta;
                }
            }
            if (best_node.node > -1)
            {
                log_message(LOG_LEVEL_INFO, "Best node found: %d\n", best_node.node);
                inst->solution[current_node] = best_node.node;
                inst->solution[best_node.node] = current_node_opposite;
                state.covered_nodes[state.covered_nodes_count++] = best_node.node;
                state.uncovered_nodes[best_node.node_index] = state.uncovered_nodes[--state.uncovered_nodes_count];
            }
        }
    }

    inst->best_cost_value = compute_solution_cost(inst, inst->solution);
    clock_t end_time = clock();
    double elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
    inst->elapsed_time = elapsed_time;
    log_message(LOG_LEVEL_INFO, "Extra Mileage solution time: %f seconds\n", elapsed_time);
    log_message(LOG_LEVEL_INFO, "Extra Mileage solution cost: %f\n", inst->best_cost_value);
}
```