Operations Reserach 2 Final Thesis

Youssef Ben Khalifa

February 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.

1.1 The Traveling Salesman Problem

The Traveling Salesman Problem is among the most disucssed and researched problems in the field of Operations Research.

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:

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

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

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

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

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

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.

```
1    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.

```
1        clock_t start_time = clock();
2        int current_node_index = starting_node;
3        int remaining_nodes_count = inst->nnodes;
4        solution nearest_node;
5        int* remaining_nodes = (int*)malloc(inst->nnodes * sizeof(int));
6        inst->solution = (int*)malloc(inst->nnodes * sizeof(int));
7        inst->best_cost_value = 0;
8        for (int i = 0; i < inst->nnodes; i++) {
9          inst->solution[i] = -1;
10         remaining_nodes[i] = i;
11       }
12       remaining_nodes[current_node_index] = remaining_nodes[--
     remaining_nodes_count];
13
```

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.

```
1     for (int i = 0; i < inst->nnodes; i++) {
2         nearest_node = euclidean_nearest_node(inst, current_node_index,
      remaining_nodes, &remaining_nodes_count);
3         if (nearest_node.node == -1) {
4             inst->solution[current_node_index] = starting_node;
5             inst->best_cost_value += euclidean_distance(inst->xcoord[
      current_node_index],
6                                   inst->ycoord[current_node_index],
7                                   inst->xcoord[starting_node],
8                                   inst->ycoord[starting_node], false);
9             continue;
10        }
11        inst->solution[current_node_index] = nearest_node.node;
12        inst->best_cost_value += nearest_node.cost;
13        current_node_index = nearest_node.node;
14        }
15
```

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.

```
1     clock_t end_time = clock();
2     double elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
3     inst->elapsed_time = elapsed_time;
4     inst->best_cost_value = compute_solution_cost(inst, inst->solution);
5
```

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.

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

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

```
1    typedef struct {
2       int node1;
3       int node2;
4    } pair;
```

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

```
1    typedef struct
2    {
3       int covered_nodes_count;
4       int* covered_nodes;
5       int uncovered_nodes_count;
6       int* uncovered_nodes;
7    } 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

   ```
   1    clock_t start_time = clock();
   2    heuristic_state state;
   3    pair current_pair = starting_pair;
   4    initialize_instance(inst, &state);
   5    inst->solution[current_pair.node1] = current_pair.node2;
   6    inst->solution[current_pair.node2] = current_pair.node1;
   7    state.covered_nodes[state.covered_nodes_count++] = current_pair.node1;
   8    state.covered_nodes[state.covered_nodes_count++] = current_pair.node2;
   9    state.uncovered_nodes[current_pair.node1] = state.uncovered_nodes[--state.
        uncovered_nodes_count];
   10   state.uncovered_nodes[current_pair.node2] = state.uncovered_nodes[--state.
        uncovered_nodes_count];
   11
   ```

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.

   ```
   1    while (state.covered_nodes_count < inst->nnodes)
   2    {
   3       log_message(LOG_LEVEL_INFO, "Covered nodes count: %d\n", state.
        covered_nodes_count);
   4       solution best_node;
   ```

```c
 5            for (int i = 0; i < state.covered_nodes_count; i++)
 6            {
 7              log_message(LOG_LEVEL_INFO, "Current node index: %d\n", i);
 8              int current_node = state.covered_nodes[i];
 9              int current_node_opposite = inst->solution[current_node];
10              double min_distance_delta = INFINITY;
11              best_node.node = -1;
12              best_node.node_index = -1;
13              for (int j = 0; j < state.uncovered_nodes_count; j++)
14              {
15                double distance1 = euclidean_distance(inst->xcoord[current_node],
16                                inst->ycoord[current_node],
17                                inst->xcoord[state.uncovered_nodes[j]],
18                                inst->ycoord[state.uncovered_nodes[j]], false);
19                double distance2 = euclidean_distance(inst->xcoord[state.
    uncovered_nodes[j]],
20                                inst->ycoord[state.uncovered_nodes[j]],
21                                inst->xcoord[current_node_opposite],
22                                inst->ycoord[current_node_opposite], false);
23                double existing_pair_distance = euclidean_distance(inst->xcoord[
    current_node],
24                                   inst->ycoord[current_node],
25                                   inst->xcoord[current_node_opposite],
26                                   inst->ycoord[current_node_opposite], false);

28                double distance_delta = distance1 + distance2 - existing_pair_distance
    ;

30                if (distance_delta < min_distance_delta)
31                {
32                  min_distance_delta = distance_delta;
33                  best_node.node = state.uncovered_nodes[j];
34                  best_node.node_index = j;
35                  best_node.cost = distance_delta;
36                }
37              }
38              if (best_node.node > -1)
39              {
40                log_message(LOG_LEVEL_INFO, "Best node found: %d\n", best_node.node);
41                inst->solution[current_node] = best_node.node;
42                inst->solution[best_node.node] = current_node_opposite;
43                state.covered_nodes[state.covered_nodes_count++] = best_node.node;
44                state.uncovered_nodes[best_node.node_index] = state.uncovered_nodes[--
    state.uncovered_nodes_count];
45              }
46            }
47            }
48
```

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.

```c
1        inst->best_cost_value = compute_solution_cost(inst, inst->solution);
2        clock_t end_time = clock();
3        double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
4        inst->elapsed_time = elapsed_time;
5
```

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

3.3   Tabu Search

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.

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.

3.4   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 *k-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.

4   TSP with CPLEX

4.1   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.

4.2 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.

```
1    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.

```
1    int num_edges = inst->nnodes * inst->nnodes;
2    double* lb = (double*)malloc(num_edges * sizeof(double));
3    double* ub = (double*)malloc(num_edges * sizeof(double));
4    char* xctype = (char*)malloc(num_edges * sizeof(char));
5    char** colnames = (char**)malloc(num_edges * sizeof(char*));
6    for (int i = 0; i < num_edges; i++) {
7      lb[i] = 0.0;
8      ub[i] = 1.0;
9      xctype[i] = 'B';
10     colnames[i] = (char*)malloc(100 * sizeof(char));
11     sprintf(colnames[i], "x_%d_%d", i / inst->nnodes, i % inst->nnodes);
12   }
13   status = CPXnewcols(env, lp, num_edges, NULL, lb, ub, xctype, colnames);
```

4.3 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:

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

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.

```
1    clock_t start_time = clock();
2    int current_node_index = starting_node;
3    int remaining_nodes_count = inst->nnodes;
4    solution nearest_node;
5    int* remaining_nodes = (int*)malloc(inst->nnodes * sizeof(int));
6    inst->solution = (int*)malloc(inst->nnodes * sizeof(int));
7    inst->best_cost_value = 0;
8    for (int i = 0; i < inst->nnodes; i++) {
9      inst->solution[i] = -1;
10     remaining_nodes[i] = i;
11   }
12   remaining_nodes[current_node_index] = remaining_nodes[--
     remaining_nodes_count];
13
```

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.

```
1    do
2    {
3      xstar = TSPopt(instance, env, lp);
4      init_data_struct(instance, &component_map, &succ, &ncomp);
5      build_solution(xstar, instance, solution, component_map, ncomp);
6      error = add_bender_constraint(env, lp, NULL, component_map, instance, *
     ncomp);
7    }while (*ncomp > 1);
8
```

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.

```
1    clock_t end_time = clock();
2    double elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
3    inst->elapsed_time = elapsed_time;
4    inst->best_cost_value = compute_solution_cost(inst, inst->solution);
5
```

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.

```
1    int error = 0;
2    CPXENVptr env = CPXopenCPLEX(&error);
3    if (error) print_error("CPXopenCPLEX() error");
4    CPXLPptr lp = CPXcreateprob(env, &error, "TSP model version 1");
5    if (error) print_error("CPXcreateprob() error");
6    double lower_bound = -CPX_INFBOUND;
7    double upper_bound = CPX_INFBOUND;
8
9    CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);
10   if (_verbose >= 60) CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_ON);
11   CPXsetintparam(env, CPX_PARAM_RANDOMSEED, 1);
12   CPXsetdblparam(env, CPX_PARAM_TILIM, 36000);
13   CPXsetintparam(env, CPX_PARAM_CUTUP, upper_bound);
14
15   build_model(instance, env, lp);
16   if (contextid == NULL) contextid = CPX_CALLBACKCONTEXT_CANDIDATE;
17   if (CPXcallbacksetfunc(env, lp, contextid, callback_driver, instance))
     print_error("CPXcallbacksetfunc() error");
18
```

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.

```
1
2
```

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.

4.4 Patching Heuristic

5 Mathheuristics

The Mathheuristics are a class of optimization algorithms that combine mathematical programming techniques with heuristics to solve combinatorial optimization problems. They are typically used to find high-quality solutions to difficult problems that cannot be solved optimally using traditional methods. In our case we will in particular focues on the Local Branching and Fixed Branching Mathheuristics.

5.1 Local Branching

The idea behind Local Branching is to optimize the solution search by adding a set of constraints to the model that restrict the search space to a local region around the current solution. This is done by adding a set of constraints that limit the number of variables that can change from their current values in the solution. The implementation involves selecting a set of variables, representing the edges of the tour,

5.2 Fixed Branching

References

[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

```
1    void tsp_grasp(instance* inst, int starting_node) {
2        clock_t start_time = clock();
3        log_message(LOG_LEVEL_INFO, "Solving TSP with GRASP\n");
4        log_message(LOG_LEVEL_INFO, "Instance details: nnodes = %d\n", inst->nnodes);
5        for (int i = 0; i < inst->nnodes; i++) {
6            log_message(LOG_LEVEL_INFO, "Node %d: (%f, %f)\n", i, inst->xcoord[i],
     inst->ycoord[i]);
7        }
8        log_message(LOG_LEVEL_INFO, "Starting GRASP with starting node %d\n",
     starting_node);
9
10       int current_node_index = starting_node;
11       int remaining_nodes_count = inst->nnodes;
12       solution nearest_node;
13       int* remaining_nodes = (int*)malloc(inst->nnodes * sizeof(int));
14
15       inst->solution = (int*)malloc(inst->nnodes * sizeof(int));
16       inst->best_cost_value = 0;
17
18       // Initialize solution
19       for (int i = 0; i < inst->nnodes; i++) {
20           inst->solution[i] = -1;
21           remaining_nodes[i] = i;
22       }
23
24       remaining_nodes[current_node_index] = remaining_nodes[--remaining_nodes_count
     ];
25
26       for (int i = 0; i < inst->nnodes; i++) {
27           log_message(LOG_LEVEL_INFO, "Current node index: %d\n", current_node_index
     );
28           nearest_node = euclidean_nearest_node(inst, current_node_index,
     remaining_nodes, &remaining_nodes_count);
29
30           if (nearest_node.node == -1) {
31               inst->solution[current_node_index] = starting_node;
32               inst->best_cost_value += euclidean_distance(inst->xcoord[
     current_node_index],
33                                                            inst->ycoord[
     current_node_index],
34                                                            inst->xcoord[starting_node
     ],
35                                                            inst->ycoord[starting_node
     ], false);
36               continue;
37           }
38           inst->solution[current_node_index] = nearest_node.node;
39           inst->best_cost_value += nearest_node.cost;
40           current_node_index = nearest_node.node;
41       }
42
43       clock_t end_time = clock();
44       double elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
45       inst->elapsed_time = elapsed_time;
46       log_message(LOG_LEVEL_INFO, "GRASP solution time: %f seconds\n", elapsed_time)
     ;
47       log_message(LOG_LEVEL_INFO, "GRASP solution cost: %f\n", inst->best_cost_value
     );
48
49       free(remaining_nodes);
50   }
```

A.2 Extra Mileage

```
1  void tsp_extra_mileage(instance* inst, pair starting_pair)
2  {
3      clock_t start_time = clock();
4      log_message(LOG_LEVEL_INFO, "Solving TSP with Extra Mileage\n");
5      log_message(LOG_LEVEL_INFO, "Instance details: nnodes = %d\n", inst->nnodes);
6      for (int i = 0; i < inst->nnodes; i++) {
7          log_message(LOG_LEVEL_INFO, "Node %d: (%f, %f)\n", i, inst->xcoord[i], inst->
   ycoord[i]);
8      }
9
10     heuristic_state state;
11     pair current_pair = starting_pair;
12
13     initialize_instance(inst, &state);
14
15     inst->solution[current_pair.node1] = current_pair.node2;
16     inst->solution[current_pair.node2] = current_pair.node1;
17     state.covered_nodes[state.covered_nodes_count++] = current_pair.node1;
18     state.covered_nodes[state.covered_nodes_count++] = current_pair.node2;
19     state.uncovered_nodes[current_pair.node1] = state.uncovered_nodes[--state.
   uncovered_nodes_count];
20     state.uncovered_nodes[current_pair.node2] = state.uncovered_nodes[--state.
   uncovered_nodes_count];
21
22     while (state.covered_nodes_count < inst->nnodes)
23     {
24         log_message(LOG_LEVEL_INFO, "Covered nodes count: %d\n", state.
   covered_nodes_count);
25         solution best_node;
26         for (int i = 0; i < state.covered_nodes_count; i++)
27         {
28             log_message(LOG_LEVEL_INFO, "Current node index: %d\n", i);
29             int current_node = state.covered_nodes[i];
30             int current_node_opposite = inst->solution[current_node];
31             double min_distance_delta = INFINITY;
32             best_node.node = -1;
33             best_node.node_index = -1;
34             for (int j = 0; j < state.uncovered_nodes_count; j++)
35             {
36                 double distance1 = euclidean_distance(inst->xcoord[current_node],
37                                                       inst->ycoord[current_node],
38                                                       inst->xcoord[state.
   uncovered_nodes[j]],
39                                                       inst->ycoord[state.
   uncovered_nodes[j]], false);
40                 double distance2 = euclidean_distance(inst->xcoord[state.
   uncovered_nodes[j]],
41                                                       inst->ycoord[state.
   uncovered_nodes[j]],
42                                                       inst->xcoord[
   current_node_opposite],
43                                                       inst->ycoord[
   current_node_opposite], false);
44                 double existing_pair_distance = euclidean_distance(inst->xcoord[
   current_node],
45                                                       inst->ycoord[
   current_node],
46                                                       inst->xcoord[
   current_node_opposite],
47                                                       inst->ycoord[
   current_node_opposite], false);
48
49                 double distance_delta = distance1 + distance2 - existing_pair_distance
   ;
50
51                 if (distance_delta < min_distance_delta)
52                 {
53                     min_distance_delta = distance_delta;
54                     best_node.node = state.uncovered_nodes[j];
```

```
55                        best_node.node_index = j;
56                        best_node.cost = distance_delta;
57                    }
58                }
59                if (best_node.node > -1)
60                {
61                    log_message(LOG_LEVEL_INFO, "Best node found: %d\n", best_node.node);
62                    inst->solution[current_node] = best_node.node;
63                    inst->solution[best_node.node] = current_node_opposite;
64                    state.covered_nodes[state.covered_nodes_count++] = best_node.node;
65                    state.uncovered_nodes[best_node.node_index] = state.uncovered_nodes[--
       state.uncovered_nodes_count];
66                }
67            }
68        }
69
70        inst->best_cost_value = compute_solution_cost(inst, inst->solution);
71        clock_t end_time = clock();
72        double elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
73        inst->elapsed_time = elapsed_time;
74        log_message(LOG_LEVEL_INFO, "Extra Mileage solution time: %f seconds\n",
       elapsed_time);
75        log_message(LOG_LEVEL_INFO, "Extra Mileage solution cost: %f\n", inst->
       best_cost_value);
76 }
```