Operations Reserach 2 Final Thesis

Youssef Ben Khalifa

February 11, 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:

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

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

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

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

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

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.

```
  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.

```
  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.

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

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

```
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.

```
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

```
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.

```
  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 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) is a metaheuristic that combines local search with a systematic change of the neighbourhood structure. The idea is to explore different neighbourhoods of the current solution to escape local optima and find better solutions.

Algorithm 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 Modeling the TSP with CPLEX

4.2 Bender's subtour elimination method

4.3 Patching Heuristic

4.4 CPLEX Callback implementation

References

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

```
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);
}
```