Applied Combinatorial Algorithms First Assignment

Daniel Scheurer Franco
CS&E Masters
Eindhoven University of Technology
Eindhoven, Netherlands
d.scheurer.franco@student.tue.nl

I. Introduction

This report details the first assignment of Applied Combinatorial Algorithms (5LIG0).

II. Problem

This assignment revolves around the Volvo CE automatic quarry, which uses several automatic haulers in an electric quarry site where different types of gravel are produced.

There are three separate parts for this project, each with different conditions:

1) Single hauler, unlimited battery
2) Single hauler, limited battery
3) Multiple haulers, unlimited battery

The quarry site may have static objects (such as walls) and loading and unloading points. These are abbreviated as **SO**, **LP**, and **ULP**, respectively.

In each instance, the haulers have to complete a mission, which is comprised of visits to each LP and ULP as efficiently as possible.

Also, in the second part, the hauler must manage its battery, visiting charging stations (CS) whenever needed.

III. Modeling and Data Structures

To use the graph algorithms learned in class, we need first to model the problem as a graph. Since the input instances are in grid format (as shown in Figure 1), I chose to model the problem by picking every cell (x, y) to be a node in the graph.

Also, as the hauler cannot cross through (or be on top of) any static object, I left the static objects out of the graph altogether. This way, if there is only one hauler, every move is a valid move.

A. Graph

The graph is stored as an adjacency list using a dictionary, meaning the entry for node (x, y) contains a list with all the nodes the cell (x, y) is connected to. Furthermore, as the hauler cannot move diagonally, these are restricted to at most four options.

An adjacency list provides the neighbors of a node in linear time or $\mathcal{O}(n)$ for a node with n neighbors. This is optimal for a graph search algorithm such as Dijkstra's algorithm, used in the solution for the first part.

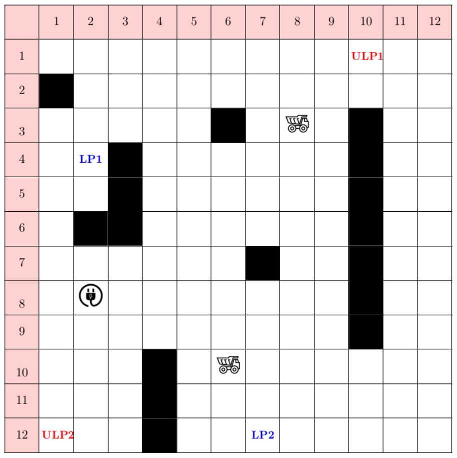

Fig. 1. Problem grid example from assignment

The dictionary lookup to find the correct adjacency list takes $O(1)$ in the average case and $O(n)$ in the worst case [1].

In the script, this list is stored in a *defaultdict* from the *collections* library, which is included by default with *Python*.

The difference between a regular *Python dict* and a *defaultdict* is that it is possible to configure the *defaultdict* to return an empty list if the searched key is not present, instead of throwing a *KeyError* as a regular dictionary would. This makes it simpler to construct the dictionary node by node when converting the grid to the adjacency list.

B. Hauler class

To separately handle each hauler, I defined a Hauler class containing some attributes and functions that are useful for the decision-making process.

The attributes are the move queue (stores the moves the hauler still has to make for a step of the mission), the hauler's history (all the nodes the hauler has visited, including the initial node), its current position, its mission, and a copy of the graph.

It also has auxiliary attributes such as a *boolean* indicator of whether the hauler has finished its mission or not, an integer to indicate when the mission was finished, and an integer to store the index of the hauler (concerning its order in the input).

The move queue and the mission are stored in queues (using the *deque* structure, also from *collections*), enabling FIFO ordering and $\mathcal{O}(1)$ time complexity for insertion and removal of elements.

Lastly, there is also a tuple named *is_charging*, used in part two of the assignment. This tuple stores whether a hauler is charging and, if so, for how long it has been charging.

IV. First Part: Single hauler, unlimited battery

In this part, the hauler calculates the shortest path in every step of the mission, meaning that at every LP or ULP, the hauler stops and calculates the shortest path to the next LP or ULP in the mission list until the mission is finished.

To try to improve a little on this, I introduced *dynamic programming* techniques by using a dictionary to store the *prev* array calculated by the *compute_shortest_path* function. This allows the hauler to skip repeated calculations and is also be useful for the case where multiple haulers are operating simultaneously.

A. Approach

As the most interesting function of my implementation is the *move* function from the *Hauler* class, I wrote a pseudocode adaptation in Algorithm 1

The flow of execution is as follows:

1) The hauler tests if it still has moves to be done in the move queue;
2) If the move queue is empty, whether it still has steps to complete the mission and, if so, it computes and stores the *prev* array for the current node;
3) Lastly, if the mission is finished, the hauler updates its status and is moved from the *working_haulers* list to the *finished_haulers* list.

The implementation of Dijkstra's algorithm was taken directly from the pseudocode shown in class. There is a priority queue Q that sorts the nodes such that the node popped from the queue is always the one with lowest distance, a visited nodes set named $visited$, a distance vector $dist$, and a previous node map $prev$.

The only tweak is that this function returns the full $prev$ vector, so the hauler can store it in the dictionary storing the computed paths.

B. Optimality

As shown in [2], Dijkstra's algorithm is actually optimal and always computes the shortest path for a graph problem with positive weights.

Since my program runs a correct implementation of this algorithm, it can also find the shortest path between each of the LPs and ULPs.

As, for this configuration, there is no better overall tour between points than the sum of the optimal paths, the algorithm implemented for this part correctly calculates an optimal solution for all instances.

Algorithm 1: Hauler.move() method

```
Input: step: Current simulation step,
       lp_ulp_shortest_paths: Dictionary of
       precomputed shortest paths
Data: move_q: Queue of pending moves,
      mission: List of mission destinations,
      history: List of visited nodes
Output: Updated state variables for the robot
1  if len(move_q) $> 0$ then
2      cur_move $\leftarrow$ move_q.popleft();
3      Append cur_move to history;
4      cur_pos $\leftarrow$ cur_move;
5  end
6  else if len(mission) $> 0$ then
7      next_lp_ulp $\leftarrow$ mission.popleft();
8      prev $\leftarrow$ [];
9      if cur_pos $\in$ lp_ulp_shortest_paths then
10         prev $\leftarrow$ lp_ulp_shortest_paths[cur_pos];
11     end
12     else
13         prev $\leftarrow$ compute_shortest_path(cur_pos);
14         lp_ulp_shortest_paths[cur_pos] $\leftarrow$ prev;
15     end
16     construct_step_queue(prev, next_lp_ulp);
17     Recursively call
         move(step, lp_ulp_shortest_paths);
18 end
19 else
20     finished $\leftarrow$ True;
21     finished_at $\leftarrow$ step;
22 end
```

C. Complexity

Dijkstra's algorithm runs in $O(V^2)$ time [2]. Considering that a problem input has a grid of size $m \times n$, h haulers, and mission length l, we can rewrite this as $O(m^2n^2hl)$, since V is at most mn (when there are no static objects) and the algorithm is run at most l times per hauler, thus the h and l terms. Therefore, the complete program has running time $O(m^2n^2hl)$.

Considering memory complexity, the program uses an adjacency list to store the graph, thus using $O(V + E)$ memory. Converting to the input units, this becomes $O(mn + mn)$, since $E < 4mn$. The other data structures used, such as priority queues and regular queues also use $O(mn)$ memory. Therefore, the memory complexity of the algorithm depends solely on the grid size and uses $O(mn)$ memory in the worst case.

D. Discussion

In this part, the results from my implementation compared well to the baseline results provided.

The makespan was equal in all of the test cases in the test data set and the running time of the program was between 0

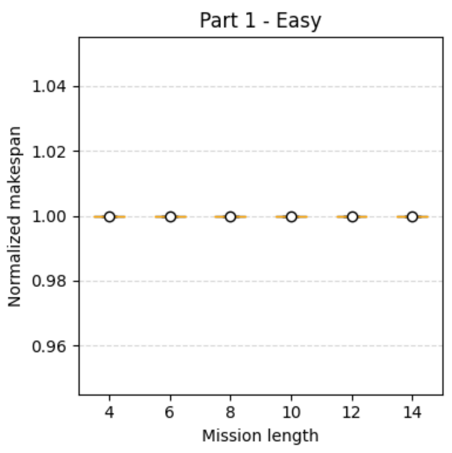

Fig. 2. Normalized Makespan for Part 1 - Easy Cases

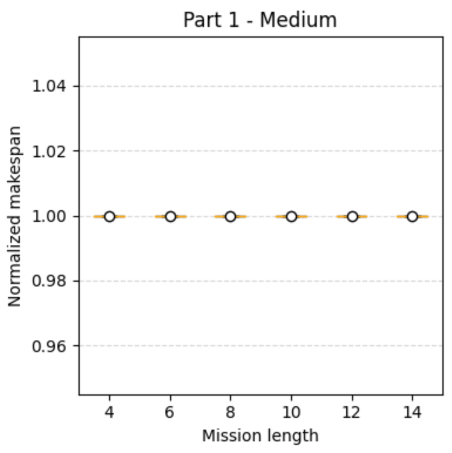

Fig. 3. Normalized Makespan for Part 1 - Medium Cases

and 2 milliseconds (floored), as shown in Figures 5, 6, and 7. The algorithm is somewhat consistent, with some larger instances having faster running times than smaller ones, most likely due to the flooring of these values.

Future work on this part could be improving on the algorithmic choice or preparation, which is the decisive variable for instances with a single hauler and unlimited battery. For instance, [3] shows that time complexity in pathfinding in 2D grids can be further improved by reducing the total graph considered for pathfinding (PBGG).

This means the algorithms have fewer nodes to consider when searching for the shortest path, and can improve the average running time of a Dijkstra algorithm by 67% [3].

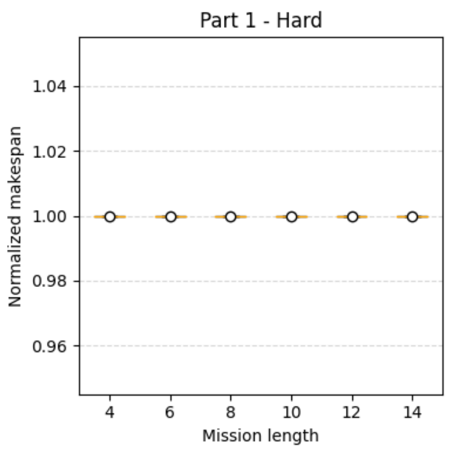

Fig. 4. Normalized Makespan for Part 1 - Hard Cases

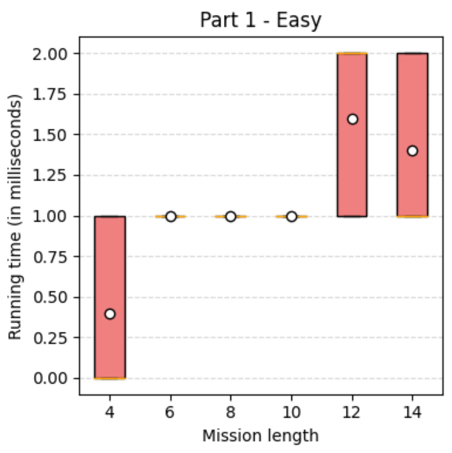

Fig. 5. Running time for Part 1 - Easy Cases

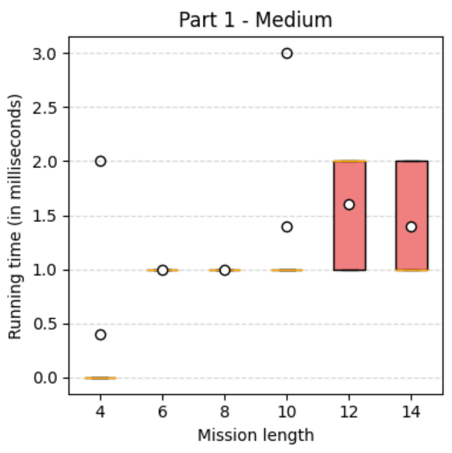

Fig. 6. Running time for Part 1 - Medium Cases

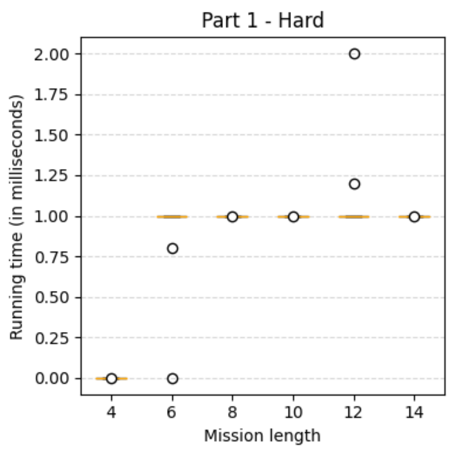

Fig. 7. Running time for Part 1 - Hard Cases

V. Second Part: Single Hauler, Limited Battery

For this part, I chose to keep the Dijkstra's algorithm I implemented for the first part.

A. Approach

This algorithm is simple, and consists in taking the path with the least detour that visits a charging station, stopping to charge, and continue the mission.

The shortest paths are calculated by Dijkstra's algorithm, with two calls being made to the function: one to calculate the shortest path from the current position to the charging station, and another from the charging station to the LP or ULP in question.

Considering the pseudocode in Algorithm 2, which refers to the *move* function inside the *Hauler* class, the following modifications can be seen in comparison to Algorithm 1:

- There is now the test to see if the hauler is at a CS, and the appropriate way to record it is charging.
- Instead of simply calculating the shortest path, we now calculate the least-detour path passing between the current position, a CS, and the target LP/ULP.
- The step queue is constructed in two steps, as the hauler will first move to the CS and then to the target LP/ULP.

The rest of the code is the same as for part 1, with the exception of auxiliary functions used to compute the least-detour path.

B. Optimality

I chose to implement a sub-optimal solution. I first implemented the solution I provide in my submission, as it seemed it was the simplest solution possible, then tried to both tweak and refactor it closer to optimality. Despite my efforts, I was unsuccessful in doing so and, thus, the final implementation is only the sub-optimal one.

This implementation is far from optimal, as shown by Figures 8, 9, and 10. However, I did find one simple tweak that enables a small improvement to the makespan of the solution: setting a threshold to decide whether or not to charge when reaching the charging station.

This has actually to be tested before running the program, as during testing I saw many instances where the hauler fails to charge when passing by the charging station and this results in it running out of charge later. For the test cases provided, I found the threshold of 75% of the battery capacity to work without problems.

C. Complexity

Both running time and memory complexity are kept the same as in part 1, and this can be shown through the similarity between the plots included for each part.

Algorithm 2: Hauler.move() method

```
Input: step: Current simulation step,
       insp: Dictionary of precomputed shortest paths,
       ind: Dictionary of precomputed distances
Data: move_q: Queue of pending moves,
      mission: List of mission destinations,
      history: List of visited nodes,
      cs_list: List of charging stations,
      csq: construct_step_queue function,
      fldcs: find_least_detour_charging_station
      function
Output: Updates the hauler state inplace
if $len(move\_q) > 0$ then
    if $cur\_pos \in cs\_list$ and
     $energy \leq 0.75 \cdot battery\_capacity$ then
        if $charging\_flag = True$ then
            Charge battery;
            Append cur_pos to history;
            return;
        end
    end
    cur_move ← move_q.popleft();
    energy ← energy − 50;
    Append cur_move to history;
    cur_pos ← cur_move;
    charging_flag ← True;
end
else if $len(mission) > 0$ then
    nlp ← mission.popleft();
    cs ← fldscs(nlp, insp, ind);
    csq(important_nodes_shortest_paths[cs], nlp);
    csq(insp[cur_pos], cs);
    Recursively call move(step, insp, ind);
end
else
    finished ← True;
    finished_at ← step;
end
```

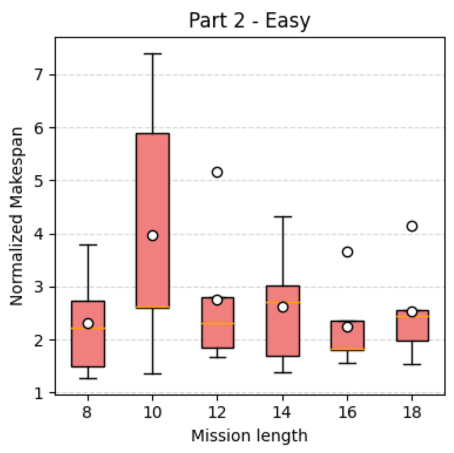

Fig. 8. Normalized Makespan for Part 2 - Easy Cases

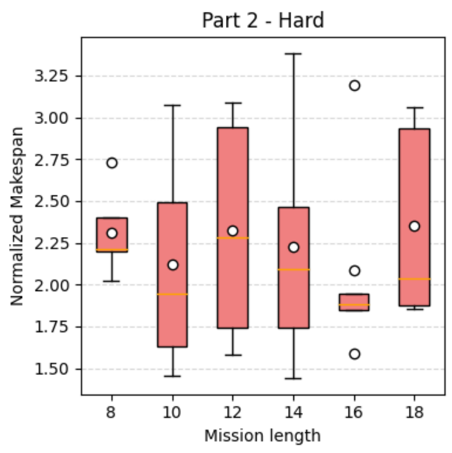

Fig. 10. Normalized Makespan for Part 2 - Hard Cases

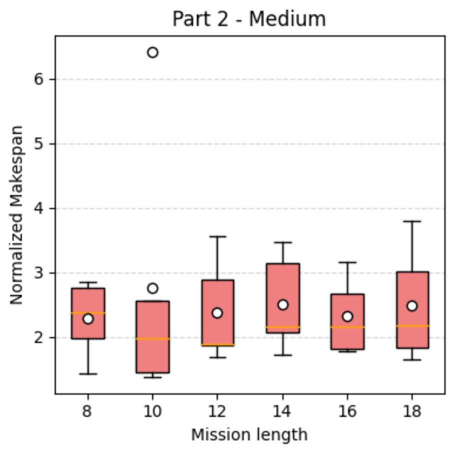

Fig. 9. Normalized Makespan for Part 2 - Medium Cases

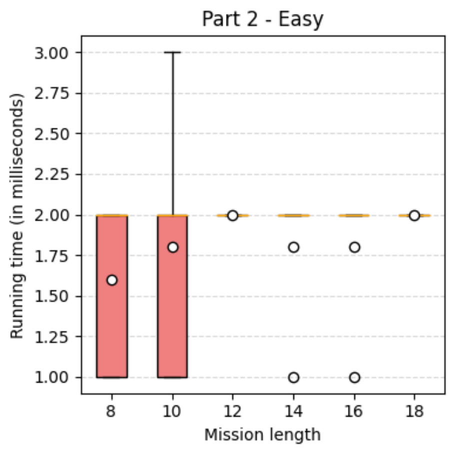

Fig. 11. Running time for Part 2 - Easy Cases

D. Discussion

Considering future work on this part, for instance, I could aim to get closer to optimality by designing a heuristic to be used with an algorithm such as $A*$. This is usually some combination of the contributing factors for the solution, which, in this problem, could be the current energy, the distance from the target LP/ULP, and the distance from the nearest charging station.

In this part, as shown in Figures 8, 9, and 10, my implementation is far from optimal. It is correct, however, as the hauler never runs out of battery or collides with a static object.

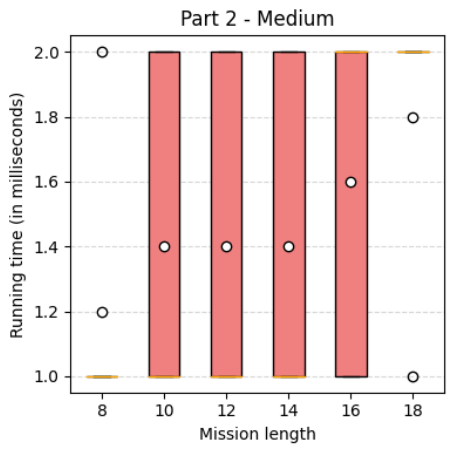

Fig. 12. Running time for Part 2 - Medium Cases

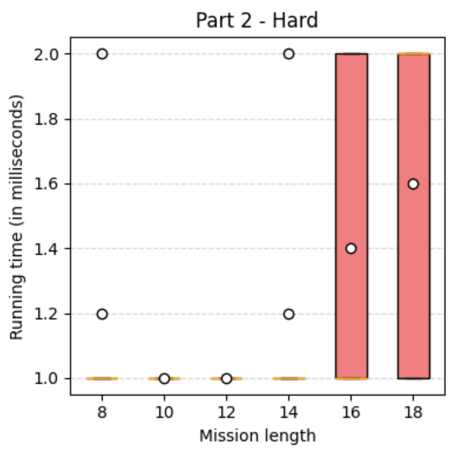

Fig. 13. Running time for Part 2 - Hard Cases

VI. Third Part: Multiple Haulers, Unlimited Battery

For this part, I reused the same Dijkstra's algorithm implementation as in parts 1 and 2.

A. Approach

My implementation of this problem is also simple, with the only big difference being the function to solve the conflicts between haulers, which is shown in Algorithm 3.

This algorithm uses a priority queue to keep track of which nodes should decide (and execute) their moves first.

In this way, each hauler is assigned a priority based on their travelled distance (the further, the higher the priority) and, in cases of tie, the index of the hauler (its order in the input) allows the algorithm to clear any ambiguity.

There is also the case where a hauler with lower priority has to divert from their planned route to give way to a higher-priority hauler. For this case, the function in Algorithm 4 provides the hauler with a new move.

Algorithm 3: Solve Move Conflicts for Haulers

```
Input: h_list: List of haulers,
       lusp: Dictionary of precomputed shortest paths,
       q: Priority queue for moves,
       so_list: List of stationary obstacles or
       constraints
Data: history: List of visited nodes for each hauler,
      cur_pos: Current position of each hauler,
      next_move: Function to determine the next
      move for a hauler
Output: Resolved queue of hauler moves without
        conflicts
1  Initialize minheap ← {};
2  foreach $h \in h\_list$ do
3      p ← len($h$.history) · −1;
4      minheap.push(($p$, $h$.ind, $h$, $h$.next_move(lusp)));
5  end
6  Initialize pos_set ← $\emptyset$;
7  while minheap is not empty do
8      ($p$, _, $h$, pos) ← aux_q.pop();
9      if pos ∈ pos_set then
10         if $h$.cur_pos ∉ pos_set then
11             $h$.add_move($h$.cur_pos);
12             minheap.push(($p$, $h$.ind, $h$, $h$.cur_pos));
13         end
14         else
15             find_move($h$, $h$.cur_pos, pos_set, so_list);
16             minheap.push(($p$, $h$.ind, $h$, $h$.next_move(lusp)));
17         end
18     end
19     else
20         Add pos to pos_set;
21         q.push(($p$, $h$.ind, $h$, pos));
22     end
23 end
```

B. Optimality

This implementation is not optimal, as seen by the comparisons to the baseline, but it manages to find an optimal solution for many cases. From my inspection, these are cases where no hauler has to change their course, and the conflict can be solved by one hauler staying in place while the other one moves.

I believe this diverges when a hauler has to move outside of its course. In my implementation, the hauler looks for a temporary location to move to in order to resolve the conflict. Most likely there is a more efficient way of solving this type of conflict, but it did not occur to me at the time of programming and writing this report.

C. Complexity

As in part 1, if we consider a problem input that has a grid of size $m \times n$, h haulers, and mission length l, the complexity of the path calculations is summarized to $O(m^2n^2hl)$, since V is at most mn (when there are no static objects) and the algorithm is run at most l times per hauler.

The conflict solving adds almost no overhead to the question, since we compare each hauler to the others at most once, thus being $O(h^2)$.

Considering memory complexity, it is the same as the part 1. Therefore, the memory complexity of the algorithm depends solely on the grid size and uses $O(mn)$ memory in the worst case.

D. Discussion

In this part, the results from my implementation compared well to the baseline results provided. The makespan was close to equal in all of the test cases in the test data set and the running time of the program was between 0 and 2 milliseconds (floored).

Despite performing well on all test cases, when revisiting my solution, I came to the conclusion that it can be broken

Algorithm 4: Find and Assign a Valid Move for a Hauler

```
Input: $h$: Hauler object,
       cur_pos: Current position of the hauler,
       pos_set: Set of occupied positions
Data: possible_moves: Set of possible moves from the
      current position,
      add_move: Function to add a move to the
      hauler's move queue
Output: Updated move queue for the hauler
1 foreach $move \in possible\_moves$ do
2     if $move \notin pos\_set$ then
3         $h$.add_move(cur_pos);
4         $h$.add_move(move);
5         break;
6     end
7 end
```

by a specific test instance. If the hauler has nowhere to move (haulers in line, stuck, with static objects around), then the solution would be incorrect, as the hauler would stay on course, moving to the same place as another hauler. Despite brainstorming different ideas, I could not find one where a deadlock could be prevented.

Future work on this part would be the implementation of the resolution of conflict in case of no possible moves. With that, I believe the solution will be very close to optimal, and fully correct.

Also, while writing this report, I thought of a different implementation for the conflict solving algorithm, for which the computation, using the idea of merge sort, could be done in $O(h \log h)$ time, for h haulers, instead of the current $O(h^2)$. This would use the idea of solving each conflict recursively and merge the solved conflicts, aiming for the linearithmic time complexity.

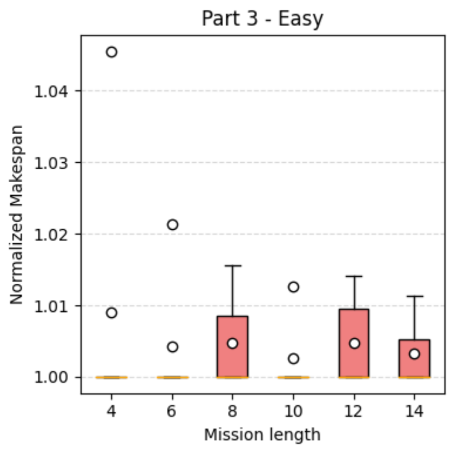

Fig. 14. Normalized Makespan for Part 3 - Easy Cases

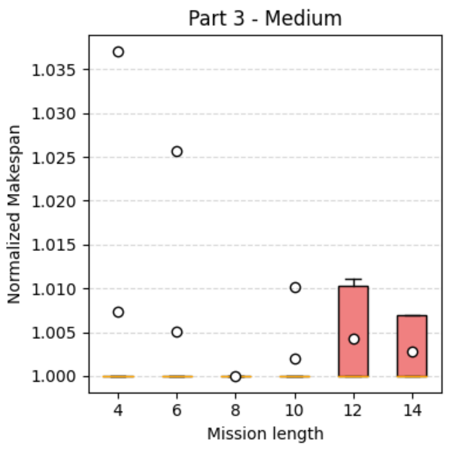

Fig. 15. Normalized Makespan for Part 3 - Medium Cases

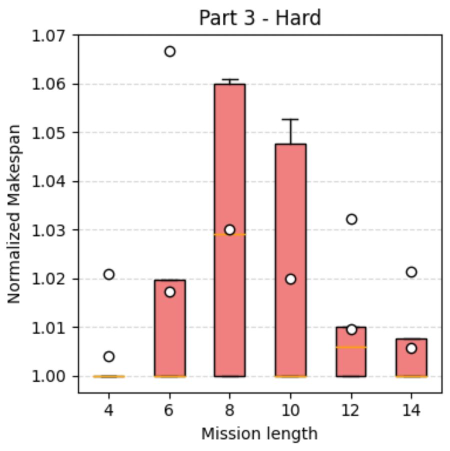

Fig. 16. Normalized Makespan for Part 3 - Hard Cases

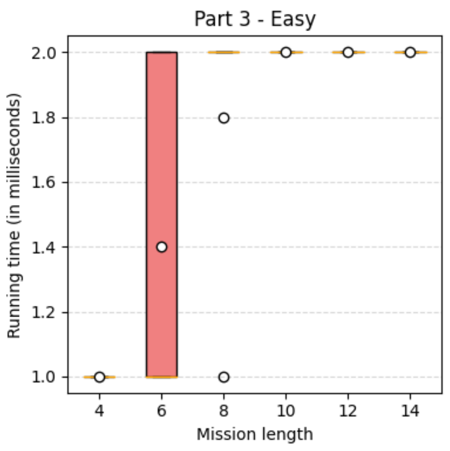

Fig. 17. Running time for Part 3 - Easy Cases

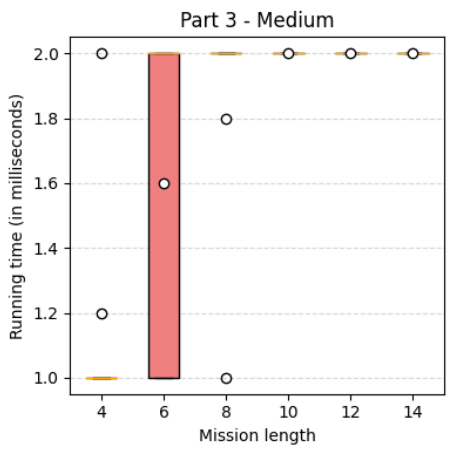

Fig. 18. Running time for Part 3 - Medium Cases

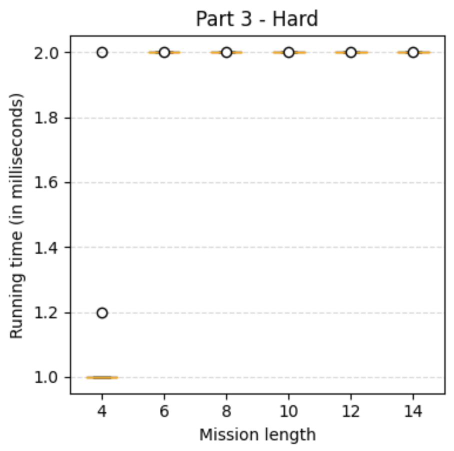

Fig. 19. Running time for Part 3 - Hard Cases

References

[1] "TimeComplexity - Python Wiki — wiki.python.org," https://wiki.python.org/moin/TimeComplexity, [Accessed 08-12-2024].

[2] C. Thomas H., L. Charles E., R. Ronald L., and S. Clifford, *Introduction to Algorithms, Fourth Edition.* The MIT Press, 2022, vol. Fourth edition, ch. VI. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&amp;db=nlebk&amp;AN=2932690&amp;site=ehost-live

[3] C.-Y. Kim and S. Sull, "Grid graph reduction for efficient shortest pathfinding," *IEEE Access*, vol. 11, pp. 74 263–74 276, 2023.