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 constant time or $\mathcal{O}(1)$, which is optimal for a graph search algorithm such as Dijkstra's algorithm, used in the solution for the first part.

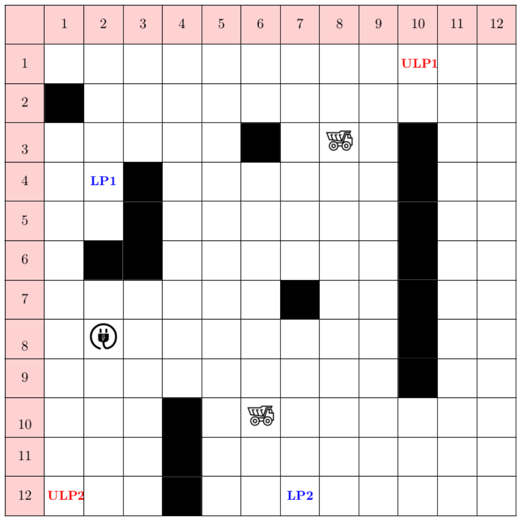

Fig. 1. Problem grid example from assignment

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 added a dictionary to store the *prev* array calculated by the *compute_shortest_path* function. This allows the hauler to skip repeated calculations and can also be useful for the case where multiple haulers are operating simultaneously.

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.

A. Approach

For this part, I chose to use Dijkstra's algorithm to find the shortest path between nodes. A pseudocode adaptation from the *Python* code can be found in Algorithm 1.

This algorithm was implemented 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.

In comparison to the baseline results provided, the algorithm is visibly functionally correct as the makespan is exactly the same as the baseline. In short, this program allows the haulers to easily complete their missions by taking the shortest route between interest points.

B. Optimality

As shown in [1], Dijkstra's algorithm is actually optimal and always computes the shortest path for a graph problem with positive weights.

Since my program runs a simple and functionally correct implementation of this algorithm, it is also proven to find the shortest path between each of the LPs and ULPs. Also, this implementation finds the shortest path overall, since there is no better path that could be taken in the conditions given for this part.

Algorithm 1: Dijkstra's Algorithm for Shortest Path - Pseudocode from Python

```
Input: Graph $G = (V, E)$ with weights $w$, start node $s$
Output: Vector of previous nodes for shortest path
        from $s$ to any node
Initialize distance $dist[v] \leftarrow \infty$ for all $v \in V$;
$dist[s] \leftarrow 0$;
Initialize previous node map $prev[v] \leftarrow$ None for all
  $v \in V$;
Initialize priority queue $Q \leftarrow [(0, s)]$;
Initialize visited nodes set $visited \leftarrow \{\}$;
while $Q$ is not empty do
    pop $(cur\_dist, v)$ from $Q$;
    if $v \in visited$ then
        continue;
    end
    Mark $v$ as visited;
    foreach neighbor $u$ of $v$ with edge weight $w$ do
        $test\_dist \leftarrow cur\_dist + w$;
        if $test\_dist < dist[u]$ then
            $dist[u] \leftarrow test\_dist$;
            $prev[u] \leftarrow v$;
            push $(test\_dist, u)$ into $Q$;
        end
    end
end
return $prev$;
```

C. Complexity

Dijkstra's algorithm runs in $O(V^2)$ time [1]. 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.

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 and 3 milliseconds.

As shown earlier, Dijkstra's algorithm is optimal and, therefore, the makespan values, both from the baseline and the implementation here described, are optimal.

On running time, the algorithm is somewhat consistent, with some larger instances having faster running times than smaller ones. This could be because, at small enough grid sizes, the time taken by the hardware dominates over the time of the algorithm itself.

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, [2] 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% [2].

References

[1] 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

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