COMP3888 - phase 1.1

October 2020

1 Introduction

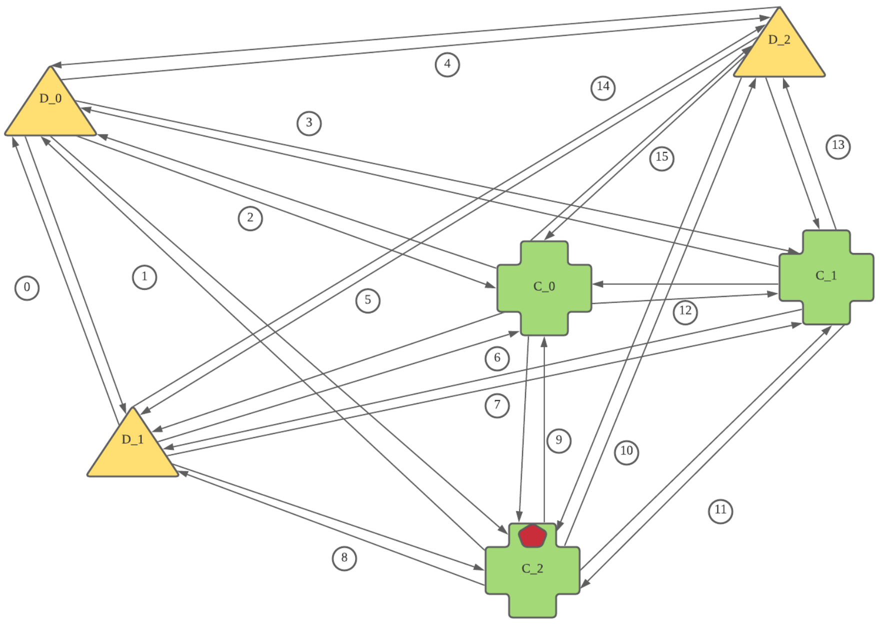
This document describes a new feature of enabling finding optimal path even when a drone is required to fly to multiple destinations as specified by client. It might not be a solution with best time complexity. The algorithm still base its fundamental approach on Dijkstra's. Adding a few more steps including exploring routines.

2 Detailed steps

2.1 Change to graph construction

The original steps of constructing the graph requires extra steps since now we have more than one destinations. The nodes can be categorized as two groups, charging stations and destinations. Now the starting point, reasonably, should be considered as a charging station.
The steps of building graph will be as simple as **connect all nodes with one another bidirectionally**. Calculation of path length is unchanged.

```
for node_1 in nodes:
    for node_2 in rest:
        path = connect(node_1, node_2)
        path.cost = x...y...z...
```



2.2 Change to battery check

At this stage, we cannot perform the start battery check and destination battery reserve anymore. Now, the steps that can be done with graph construction is only **avoiding paths that are longer than maximum range**. For more complex battery reserve, it must be done in Dijkstra's since we no longer know the incoming battery when flying to a destination, the drone can come from any where.

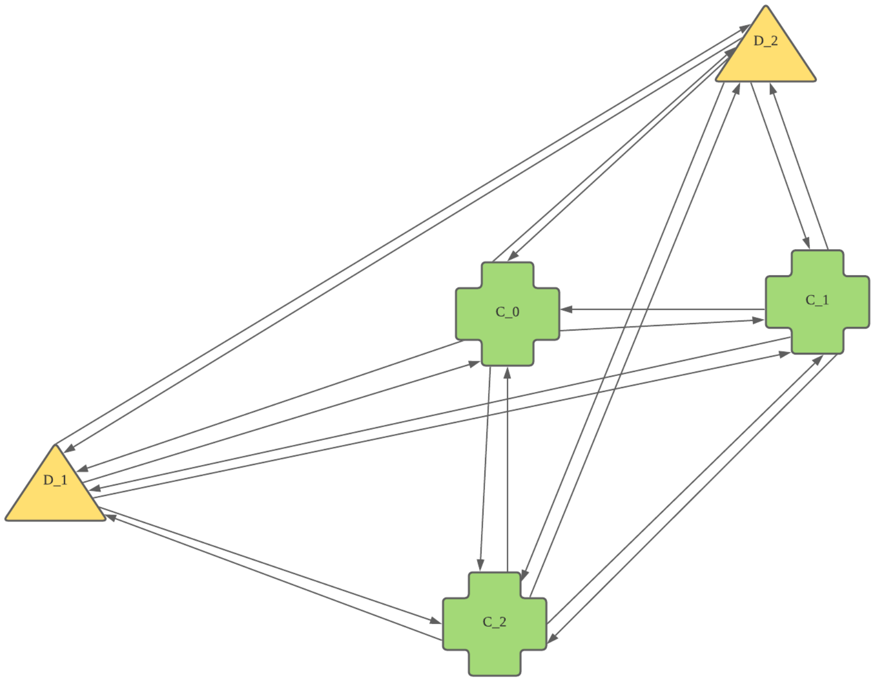
```
for path in paths:
    if path.cost > maximum:
        paths.remove(path)
```

2.3 Change to Dijkstra's

Dijkstra's will, as usual, be used between a **destination** and another **destination**. However, when doing Dijkstra's, we can ignore all other destinations. This is because the later routine-finding algorithm will be able to explore them. The following graph demonstrate the visible(nodes and paths that should be included) when performing Dijkstra's. This can improve our performance for a bit. IMPORTANT: Dijkstra's should be able to record not only the optimal path but also all the possible path since we do not know the incoming battery status of the drone.

```
for nodes, paths in (nodes, paths).
```

```
    exclude(other destinations and paths):
        perform Dijkstra's
```



2.4 Extra step: paired Dijkstra's

To perform the rest of the algorithm, we need to perform Dijkstra's on **every pair of destinations**. The route and route cost shall be recorded.

```
for d1 in destinations:
    for d2 in rest:
        path, path_cost = Dijkstra(d1, d2)
        record(path, path_cost)
```

We should be able to imagine the result of this as a matrix

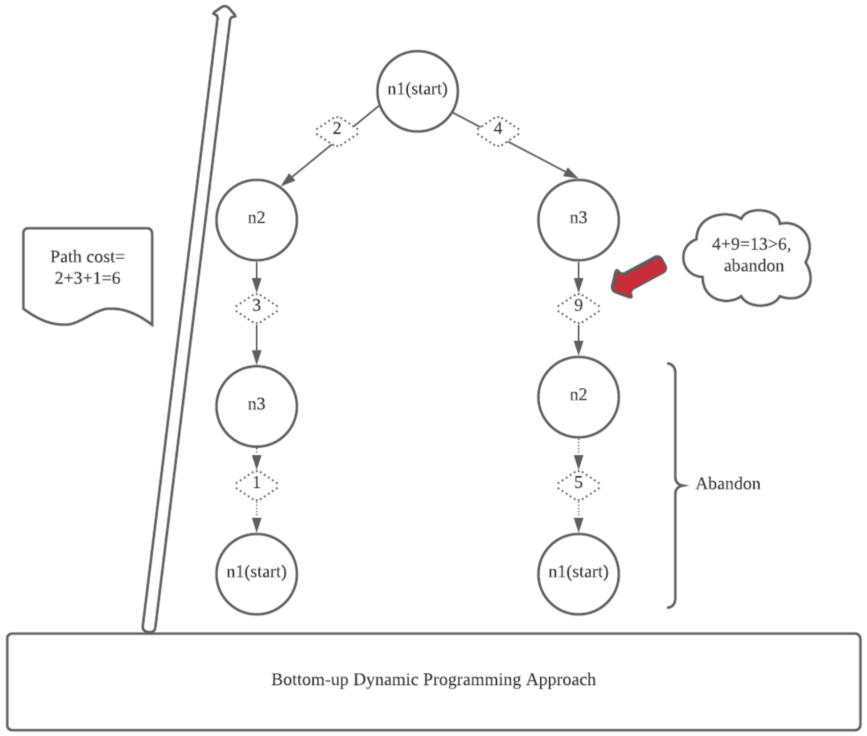
$$\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 0 & 3 \\ 2 & 1 & 0 \end{array}$$

Note: The reason why we want to separate the bidirectional connection is because there is a potential path cost difference. E.g. ascending and descending, difference battery cost. **Note** IMPORTANT: It is not guaranteed that every destination can directly fly to another, we do not know the status of drone battery. This MUST be recorded so that the next step could work.

2.5 Branch and bound algorithm

Inspired by AI, other than brute forcing all possibilities, we can perform a visiting, recording, comparing step to avoid unnecessary tree exploring. Main idea: explore all possible order of visiting, record the current best, avoid paths that cannot do better than this.

```
start = ?
best = ?

branch_and_bound(current, rest):

    if rest is empty:
        distance += return_to_start

        if distance better:
            best = this_route

    for node in rest:

        if < best:

            # should also add up distance
            branch_and_bound(node, rest.pop(node))
```



Moreover, when going down a branch, we need to check if the battery remaining is sufficient to use the next optimal value. **If not**, we might need to reschedule the whole route to accommodate this.

```
attempt_another(node):
    for path in other_paths:
        for child in children:
            attempt(path)+attempt_another(child)
```

Note: Do not use a recursive approach, convert to loops instead

2.6 yield solution

The program should output a similar result as previously designed. Using basic pointer tracking it should be able to get a full route from the graph based on constructed result, which is the path with minimum cost.

3 Complexity

Assumption: n=number of nodes, m=number of charging stations
Constructing graph as well as first step battery check yields

$$O((n+m)^2)$$

For both running time and space complexity.
Paired Dijkstra's, as known, the number of pairs should be $O(n^2)$. Each run requires $O(m^2)$. Thus, paired Dijkstra's requires

$$O(n^2m^2)$$

in total.
The routine finding algorithm, undoubtedly, runs in factorial at worst case.

$$O(n!)$$

. This is primarily the reason why we introduced bounding, this, is practice, should **reduce the running time** for a decent amount of time. It is possible, as mentioned, that we need to reschedule the route. This step is expensive as well, might produce another factorial complexity step.
In the future, we are looking forward to using a approximated(greedy-like) algorithm to find out an lower bound of routine prior to running the optimal step. This retrieved value should serve as the "best" variable mentioned above. It should assist the branch algorithm to locate paths that are worth exploring.