SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

COURSE: IT3160E

Design an optimal route with fuel constraints

Author

NGUYEN XUAN THANH - 20225460

NGUYEN TRI THANH - 20225457

LE NHAT QUANG - 20225522

TRAN QUANG HUY -20225500

Mentor

THAN QUANG KHOAT

December, 2023

Table of Contents

December 2023

Abstract

Pathfinding stands as a vital problem-solving endeavor within the realm of computer science, with the objective of identifying the most efficient route between two points within a specified setting. Its utility extends across diverse domains, including robotics, video games, network routing, and logistics. The core objective of pathfinding algorithms is to adeptly traverse intricate graphs or grids, taking into account variables such as distance, obstacles, and cost. This experiment introduces the algorithms to solve a problem of constrained shortest pathfinding, underscoring its applicability and adaptability in confronting challenges within a wide array of technological contexts.

1. Introduction

In this project, we came up with the idea and developed it based on optimization problems related to pathfinding in the pass. Assume that, in a race, your task is to go from the Start position to the Finish position. Initially, you start from the Start position, and then you must sequentially pass through positions categorized into many levels (before reaching a position with level n, you must pass through a position with level n-1, there are many positions with the same level) before reaching the Finish position. There are gas stations scattered along the way, and your car can only travel within a limited distance before needing to refuel. The goal is to find an optimal solution to reach the Finish position with the shortest distance. The requirement is to satisfy constraints about fuel.

Notable features:

- Graph Representation: The project uses a graph to model the racetrack, allowing for efficient pathfinding and distance calculations.

- Fuel Constraints: The AI considers the limited fuel capacity of the car (x km) and plans refueling stops accordingly.

- Sequential Levels: The AI ensures that positions are passed through in a specific order, moving from level n-1 to level n.

- Optimal Path: The AI calculates and presents the shortest distance path while adhering to the fuel constraints.

The intuitive idea underlying potential solutions: The key idea here is to construct a racetrack graph with position nodes. The AI aims to minimize travel distance within fuel constraints. It accounts for the sequential levels of positions and the necessity to pass through all four levels. Dynamic programming efficiently calculates the optimal path while considering fuel consumption, and backtracking yields the final solution. In essence, the AI must balance covering necessary positions and refueling strategically for the shortest overall distance, avoiding unnecessary detours and ensuring the car doesn't run out of fuel on its way to the Finish position.

Here are steps that our team implemented the project:

- Phase 1 (Weeks 1-4): Data Generation and Visualization and Initial Algorithm Development.
- Phase 2 (Weeks 5-6): Algorithm Testing and Refinement.
- Phase 3 (Weeks 6-7): Performance Evaluation and Optimization.
- Phase 4 (Week 8): Final Testing and Project Wrap-up.

Finally, we generated robust AI models that capable of efficiently determining the shortest race route considering fuel constraints and mandatory level-based stops. From that, we made comprehensive comparisons to choose the most optimal solutions for the project.

2. Literature Review

Shortest pathfinding is a fundamental problem in computer science and graph theory, focusing on determining the most efficient route between two points in a network or graph. The objective is to find the path with the minimum cost, where the cost can represent various factors such as distance, time, or other relevant metrics. This problem is pervasive in numerous real-world applications, ranging from navigation systems and logistics to network routing and game development. There exists plenty of scientists and researchers who found interest in this topic, and also a multitude of algorithms has been developed to address the shortest pathfinding challenge, each catering to specific scenarios and constraints. Notable algorithms include Dijkstra's algorithm, Greedy algorithm, A* algorithm, and variations like Bidirectional A* and Bellman-Ford algorithm. Shortest pathfinding and constrained pathfinding plays a critical role in optimizing resource utilization, enhancing decision-making processes, and contributing to the efficiency of diverse systems across different domains. Based on previous research, in this experiment, we conduct the performance evaluation of the algorithms: A* algorithm, Greedy algorithm and Bidirectional A* algorithm.

A* algorithm, as an improved version of Dijkstra’s algorithm, proved to be more effective due to its better estimation. There are also some research comparing the effectiveness of A* algorithm and Dijkstra’s algorithm, for instance, Indonesia’s Foead, D., Ghifari, A., Kusuma, M. B., Hanafiah, N., & Gunawan, E. has conducted a research about its application, the comparison with other algorithms and potential future development [1]. While beginning to show its age, improved algorithms based on the classic A* algorithm are more than capable of keeping up with modern pathfinding demands.

[1] Daniel Foead, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, Eric Gunawan (2021). A Systematic Literature review of A* pathfinding. *Procedia Computer Science 179,* 507–514

These derivative search algorithms such as Bidirectional A* is used to overcome the limitations of A*.

Bidirectional heuristic search stands as a widely recognized approach for tackling pathfinding challenges, wherein the objective is to identify paths, often of minimum cost, connecting nodes in a graph. Various real-world scenarios, such as determining the fastest route on a map or assessing the similarity of DNA sequences, can be formulated as pathfinding problems.

In the context of bidirectional brute-force search, simultaneous searches are conducted both forward from the initial state and backward from the goal states, with solutions identified at the point of intersection. While the concept of incorporating heuristics to guide the search process is not new, it has not gained widespread adoption and is generally perceived as ineffective. However, Bidirectional A* enhances the capabilities of A* by concurrently executing two A* searches, originating from both the initial state and the goal state, proven to be more suitable for more complicated pathfinding problems[2].

All the previously mentioned research has their own purpose but there does not exist an overview of comparing these algorithms in a specific constrained pathfinding problem. Therefore, in this experiment, we will try to compare the performance of these algorithms in solving the problem mentioned above.

3. Methodology

3.1. Data Preprocessing

Data cleaning

The original **Food premises hygiene data set** (CSV) is collected and generated by organizations in the public sector in Belfast updated on the **Belfast City Council** website in the **Open and linked data** section. It contains information about food premises, such as restaurants and takeaways, in Belfast[3].

Why we choose this dataset? Because:

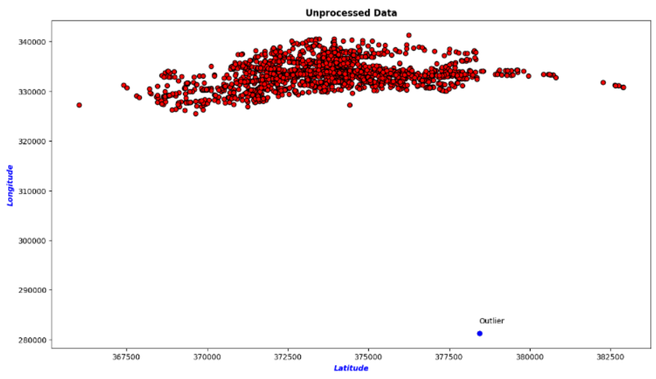
- It is free to use for any legal purpose (learning purpose) under the Open Government License.
- It contains no personal data and meets Data Protection Act legislation.
- It is published in a form that makes it easy to manipulate in software like mobile apps[3].

Representation of the food premises as data points based on their coordinates.

[2] Barker, J. K. (2015). *Front-To-End bidirectional heuristic search*

[3] Belfast City Council, Open and linked data.

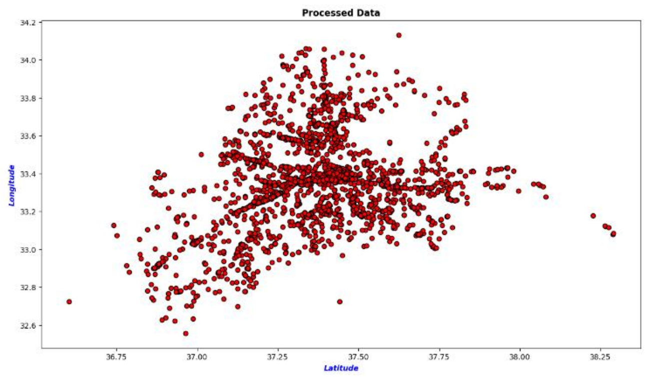
URL: https://www.belfastcity.gov.uk/open-and-linked-data



Distribution of unprocessed data points

- The coordinates of the points must be normalized to be in [-90,90]×[-180,180].
- The **outlier** (blue point) is removed for the implementation of the algorithms.
- Points of the same coordinate are excluded.

Adjustment for the problem' s size



Distribution of processed data points

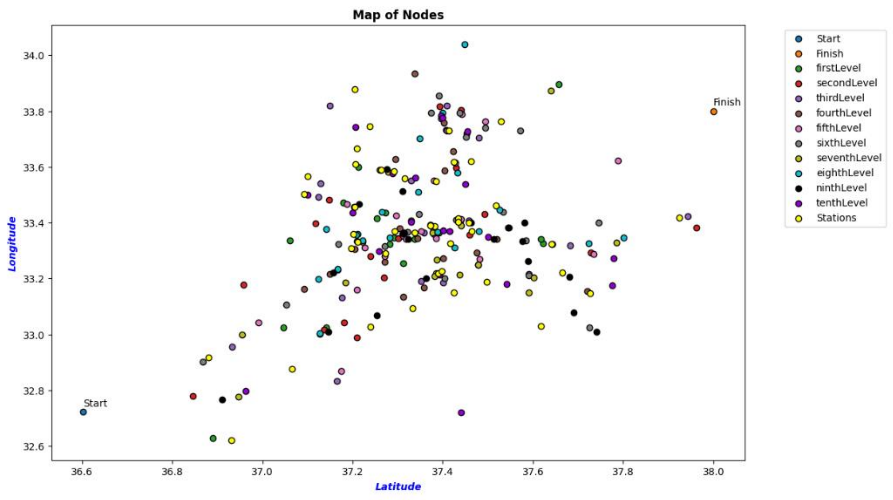
Arbitrary inputs include:

- The number of levels: n
- The number of points for each level: $l_1,\ l_2,\ \dots,\ l_n$
- The number of stations: s
- $s + \sum_{i=1}^{n} l_i \leq 2344$ (the number of data points in the processed data set)

The point at the **left bottom corner** of Figure 2 is chosen as the **"Start".** We will create the **"Finish"** with the coordinate of **(38, 33.8)**, which means that it will be at the **top right corner** of Figure 2.

A random generator is used to choose $l_i\ \forall i = \overline{1,\ n}$ points for each level and s stations from the processed data set. The generator will pick the same points for the same input even when it is activated several times.

Their corresponding coordinates and the Haversine distance from each of them to the **"Finish"** will be put into a data frame, which will be exported as a CSV file named "updatedData.csv". For $n = 10, l_i = 20\ \forall i = \overline{1,\ 10}\ and\ s = 45,$ the distribution of the chosen data points will be as follow:



Distributions of the problem's data points.

Lower bounds for the fuel constraint

The fuel constraint is extremely important in this problem. It can determine the existence of a solution path, as well as its form.

- Setting it too low, the solution might not exist.
- Setting it too high, the solution path might only consist of level points.
- The lower bound is defined as the **maximum distance** of:

- The **minimum distance** from "Start" to all its assigned neighbours
- The **minimum distance** from each station their assigned neighbours
- The **minimum distance** from "Finish" to all its assigned neighbours

To ensure that a vehicle must **visit at least one station** to get to "Finish" while not violating the fuel constraint, we should choose the constraint **as close as possible** to the lower bound.

It is also useful to remember that with these calculations, we know:

$$\boldsymbol{\forall x_1 \neq x_2 \geq lowerBound, \qquad \exists x_1\ such\ that\ x = fuel\ constraint\ and}$$

$$\boldsymbol{\exists x_2\ such\ that\ x_2 \neq fuel\ constraint.}$$

3.2. Greedy Best First Search

3.2.1 Introduction

The greedy algorithm is a simple yet powerful approach to problem-solving that follows a heuristic of making locally optimal choices at each step with the hope of finding a global optimum. It belongs to the class of optimization algorithms and is often used for solving optimization problems.

Best-first search is an algorithm that traverses a graph by continuously selecting the most promising node based on a specific criterion. A greedy algorithm consistently picks the most appealing choice at every stage. Therefore, the greedy best-first search algorithm integrates aspects of both strategies, continuously progressing towards the goal without retracing steps, as it adheres to a greedy approach that doesn't revisit paths.

In Greedy Best-First Search, heuristics determine the "best" node to select next. A heuristic is essentially an estimate of the proximity to the target. This function, typically represented as "h," can be any function, but it must satisfy the condition that h(n) is zero when n is the goal. The desired characteristics of a heuristic function are twofold: it should be relatively inexpensive to compute and serve as a reasonably precise predictor of the remaining cost to reach the goal. These attributes make the heuristic function a practical tool in navigating towards the objective efficiently.

Breadth-First Search (BFS) prioritizes expansion based on the proximity to the source node, ensuring a comprehensive exploration of the graph. This method may require backtracking if it encounters a dead end. On the other hand, Greedy Best-First Search employs a heuristic to prioritize nodes that seem closer to the target, aiming for a quicker resolution. While this approach tends to be faster, it does not guarantee the most optimal solution and does not involve retracing steps or backtracking in the path.

3.2.2 Heuristic: Nearest distance to the next position

Begin at the start position with a full tank of fuel. For the current position, determine the next target position within the next level that the car can reach with the remaining fuel. If the target position is not directly reachable due to fuel constraints, identify the nearest optimal fuel station to refuel, ensuring the detour is minimal. Record each move, whether a direct move to the next level or a detour to a fuel station, along with the distance traveled. Continue this process, moving from level to level and refueling as necessary, until reaching the finish line.

Heuristic: Nearest distance to the next position

At each step, the algorithm might choose the path that leads to the next level (or ultimately the finish line) that has the shortest distance from the current position. This is based on the assumption that shorter paths will contribute to a shorter overall route.

Another crucial aspect is the fuel constraint. The heuristic could involve choosing a path or a refueling stop based on maximizing the distance traveled per unit of fuel or ensuring that the vehicle will reach the next refueling station or level before the fuel runs out. When the fuel is not sufficient to reach the next level, the algorithm might opt for the nearest fuel station. When deciding to refuel, the algorithm might prioritize fuel stations that closer to the current path or the next target to minimize detours and after refueling, allow reaching the next level with the least additional travel.

Given the level-based structure of the route, the algorithm always aims to move to a higher level (closer to the finish line) with each decision, ensuring progress towards the goal.

Pseudocode

Algorithm 1 Greedy Algorithm for Route Optimization

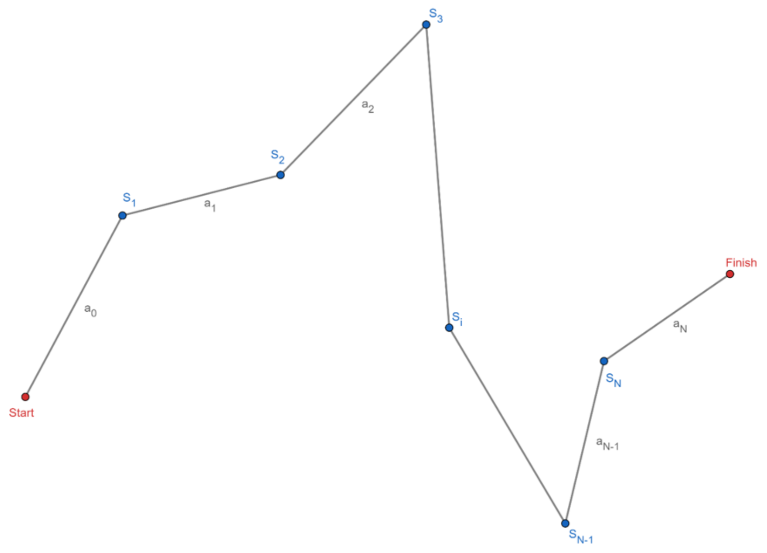
```
function GREEDY
    $current \leftarrow$ "Start"
    $distance \leftarrow 0$
    $path \leftarrow$ [current]
    $fuel \leftarrow$ MAX_FUEL_CAPACITY
    while $current \neq$ "Finish" do
        $(next\_position, required\_fuel) \leftarrow$ DETERMINENEXTPOSITION$(current, fuel)$
        if $required\_fuel \leq fuel$ then
            $current \leftarrow next\_position$
            $fuel \leftarrow fuel - required\_fuel$
            $distance \leftarrow distance + required\_fuel$
            $path.append(current)$
        else
            $(nearest\_station, distance\_to\_station) \leftarrow$
FINDNEARESTSTATION$(current, fuel)$
            $current \leftarrow nearest\_station$
            $fuel \leftarrow$ MAX_FUEL_CAPACITY
            $distance \leftarrow distance + distance\_to\_station$
            $path.append($"Refuel at " $+ current)$
        end if
    end while
    return $path, distance$
end function
```

Pseudo code

3.2.3 Heuristic: Minimal Number of Unvisited Levels

Properties of an ideal solution

Let's take a closer look at the ideal solution in this particular case.



Rough visualization of the optimal path

Assume that our shortest path consists of N stations $S_i \ \forall i = \overline{1,\ N}$ with the fuel constraint F.

Let $a_i \ \forall i = \overline{1,\ N-1}$ be the $d(S_i,\ S_{i+1})$ with $a_0 = d(Start, S_0)$ and $a_N =$

$d(S_n, Finish)$ where $d(A, B)$ is defined as the Haversine distance between A and B.

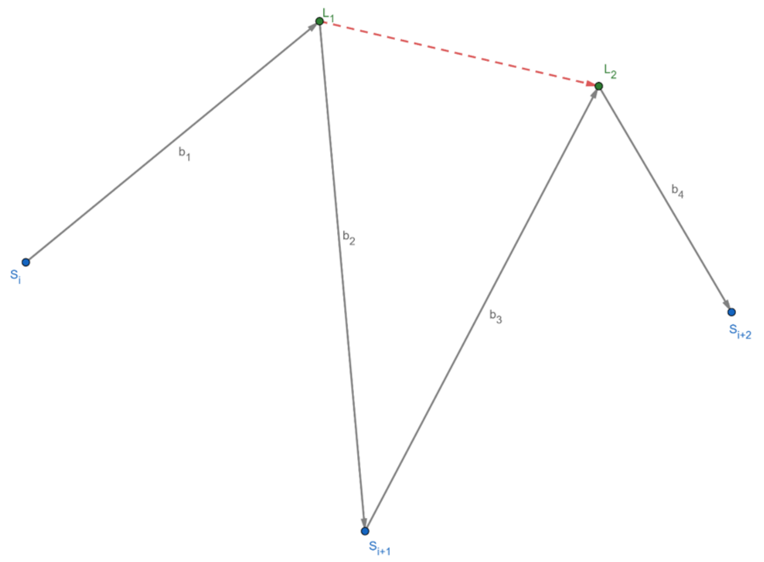
To save the travelling time, the vehicle shouldn't stop by a station if it still can reach a level point and have enough fuel to go to another station. The optimal path would then have this property:

$$\boldsymbol{Property}: \forall \boldsymbol{i} = \overline{\mathbf{0}, \boldsymbol{N}-\mathbf{1}}, \forall \boldsymbol{j} = \overline{\mathbf{0}, \boldsymbol{N}}, \quad \mathbf{0} < \boldsymbol{a_j} \leq \boldsymbol{L} < \boldsymbol{a_i} + \boldsymbol{a_{i+1}}$$

Proof by contradiction:

We start with the assumption that $\exists i \ in \ [0, N-1] \ such \ that \ L \geq a_i + a_{i+1}$. Consider the three stations $S_i, S_{i+1} \ and \ S_{i+2}$, we denote the green points as the closest level points to S_{i+1}. Then, we encounter three situations where further cases with several level points are proved in the same manner:

- *To traverse through the two sub-paths $S_i \rightarrow S_{i+1}$ and $S_{i+1} \rightarrow S_{i+2}$, the vehicle visits at least one level point in each path.*

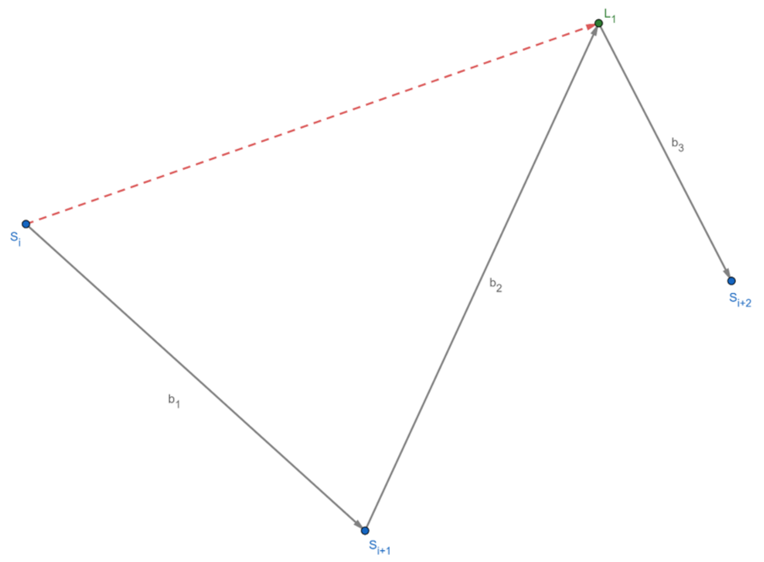


Level(s) between two sub-paths

Here, $a_i = b_1 + b_2$ and $a_{i+1} = b_3 + b_4$. Applying the triangular inequality, we have that:

$$\boldsymbol{L} \geq \boldsymbol{b_1} + (\boldsymbol{b_2} + \boldsymbol{b_3}) + \boldsymbol{b_4} \geq \boldsymbol{b_1} + \boldsymbol{d}(\boldsymbol{L_1}, \boldsymbol{L_2}) + \boldsymbol{b_4}$$

- *To traverse through two sub-paths $S_i \rightarrow S_{i+1}$ and $S_{i+1} \rightarrow S_{i+2}$, the vehicle visits at least one level point in either of the two.*

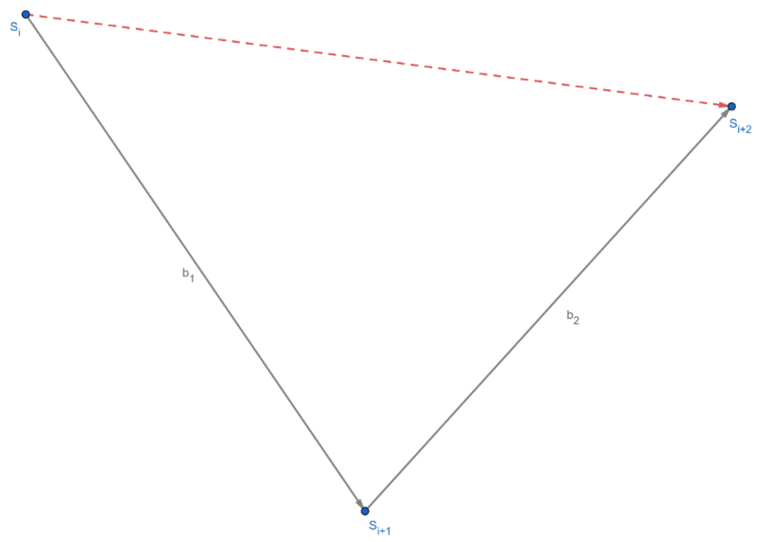


Level(s) between one sub-path

Here, $a_1 = b_1, a_2 = b_2 + b_3$. Applying the triangular inequality, we obtain:

$$\boldsymbol{L \geq (b_1 + b_2) + b_3 \geq d(S_i, L_1) + b_3}$$

- *To traverse through two sub-paths $S_i \rightarrow S_{i+1}$ and $S_{i+1} \rightarrow S_{i+2}$, the vehicle doesn't visit any stations in any paths.*



No level between any paths

Here, $a_1 = b_1$ and $a_2 = b_2$. Applying the triangular inequality, it's clear that:

$$\boldsymbol{L \geq b_1 + b_2 \geq d(S_i, S_{i+2})}$$

Equality happens if and only if all level points in between and S_{i+1} are in the same line S_iS_{i+2}. Still, it passes through S_{i+1}, which is completely unnecessary.

Otherwise, all three results show that we don't need to visit S_{i+1} to travel from $S_i \rightarrow S_{i+2}$ while still visiting all level points in between and respecting the fuel constraint. Hence, the original path wouldn't be optimal.

Heuristic generation

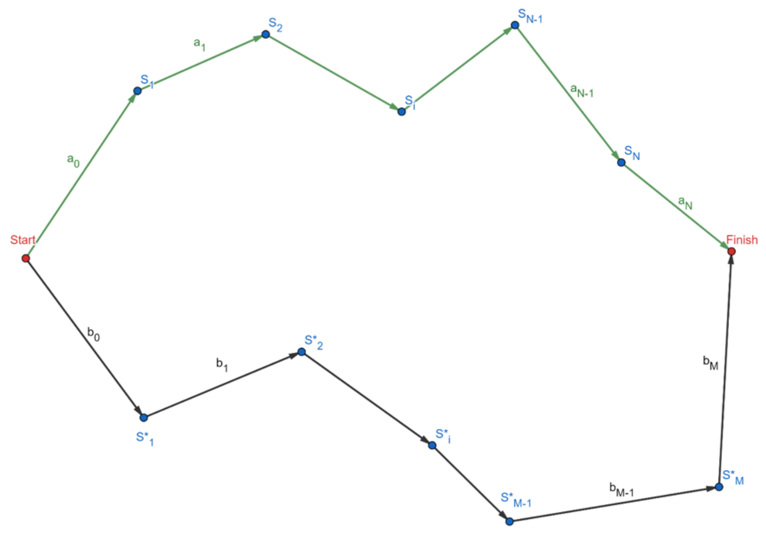
$$\boldsymbol{Heurtistic\ function = the\ number\ of\ unvisited\ levels\ of\ a\ path}$$

```
1: procedure Heuristic(path):
2:     $N \rightarrow NumberOfLevels$
3:     $check = [0\ for\ i\ in\ range(N)]$
4:     $for\ node\ in\ path$:
5:         $if\ node\ in\ Level(i)$:
6:             $count[i] += 1$
7:     $bool\_check = bool(check)$
8:     $return\ bool\_check.count(False)$
```

With this heuristic function, the algorithm will prioritize expanding paths having the largerst number of level points visited. It strives to get to the next level and ONLY visits a station if necessary (to respect the fuel constraint).

As such, the solution path produced by the algorithm will have the same property mentioned above.

Upper bound for the number of stations the ideal solution path



Optimal and Greedy BFS solutions

Let the number of stations in the ideal solution path and the Greedy BFS solution be N and M respectively, L be the fuel constraint and S is the distance travelled along the optimal solution.

Let $S_i\ \forall i = \overline{1, N}$ and $S_j^*\ \forall j = \overline{1, M}$ be stations in the optimal and Greedy BFS solution paths respectively.

Let $a_i\ \forall i = \overline{1,\ N-1}$ be the $d(S_i,\ S_{i+1})$ with $a_0 = d(Start, S_0)$ and $a_N =$

$d(S_n, Finish)$ where $d(A, B)$ is defined as the Haversine distance between A and B.

Let $b_i\ \forall i = \overline{1,\ M-1}$ be the $d({S_i}^*,\ {S_{i+1}}^*)$ with $a_0 = d(Start, {S_0}^*)$ and $a_N = d({S_n}^*, Finish)$ where $d(A, B)$ is defined as the Haversine distance between A and B.

($a_i, b_j, L, S \in R$ while $N, M \geq 0$ *and* $N, M \in Z$)

It is naturally intuitive to ask whether it is possible for N > M and if it is, what is the upper bound for M?

We have that:

$$\sum_{i=0}^{N} a_i \leq \sum_{j=0}^{M} b_j \leq (M+1).L$$

This is followed by:

$$\sum_{i=M+1}^{N} a_i \leq \sum_{j=0}^{M} (L - a_j) < \sum_{j=1}^{M+1} a_j \ \ (1)$$

Similarly, we would also obtain that:

$$a_N + \sum_{i=0}^{N-M-2} a_i \leq \sum_{j=N-M-1}^{N-1} (L - a_j) < \sum_{j=N-M}^{N} a_j \ \ (2)$$

Adding both sides of (1) and (2) we get:

$$\sum_{i=0}^{N-M-2} a_i + \sum_{i=M+2}^{N} a_i < \sum_{j=1}^{M} a_j + \sum_{j=N-M}^{N-1} a_j \ \ (3)$$

If $N \geq 2M + 4$, then $N - M - 2 > M + 2$ and $N - M > M$, this shows that:

$$\sum_{i=0}^{N-M-2} a_i + \sum_{i=M+2}^{N} a_i > S > \sum_{j=1}^{M} a_j + \sum_{j=N-M}^{N-1} a_j$$

As such, $N \leq 2M + 4$.

If $N = 2M + 3$, then (3) becomes:

$$\sum_{i=0}^{M+1} a_i + \sum_{i=M+2}^{2M+3} a_i < \sum_{j=1}^{M} a_j + \sum_{j=M+3}^{2M+2} a_j \ \rightarrow \textit{this is completely wrong}.$$

If $N = 2M + 2$, then (3) becomes:

$$\sum_{i=0}^{M} a_i + \sum_{i=M+2}^{2M+2} a_i < \sum_{j=1}^{M} a_j + \sum_{j=M+2}^{2M+1} a_j \rightarrow This\ doesn't\ make\ sense.$$

Hence, $M < N \leq 2M + 1$.

In conclusion, $\mathbf{2M + 1}$ **is an upper bound of** $\boldsymbol{N}$. This result can be used to prune the search tree given a small M, as well as a condition preventing the "dead end" where other algorithms may keep on traversing between two stations.

a) Neighbours

Greedy Best-First Search algorithms prioritize expanding the "first" neighbour of the current node in the priority queue sorted by the heuristic function. The assignment of neighbours for each node, especially for this problem where order of travel is respected, can affect the solution.

To minimize the number of stations visited, the assignments are as follows:

- If the node = Start, neighbours = [Level-1-points, stations]
- If the node in Level(i),
  + If $i \leq N - 1$, neighbours = [Level-(i+1)-points, stations]
  + Else, neighbours = [stations, Finish]
- If the node is a station:
  + If all levels have been visited, neighbours = [Finish]
  + Else, neighbours = [NextUnvisitedLevel-points]
- Each node is visited only once.

Forcing the next neighbours of a station to be level points is risky as the fuel constraint must still be respected. Still, this fast-forwards the exploration process.

b) Algorithm

```
1: GreedyBFS(start, goal):
2:     queue = [[start]]
3:     while queue:
4:         path = queue.pop(0)
5:         node = path[−1]
6:         if node == goal:
7:             return path
8:         if node in finalLevel:
9:             path.append(Finish)
10:            if FuelConstraint(path): return path
```

```
        $\text{else: path.remove(Finish)}$
    $\text{for neighbour in neighbours(node):}$
        $\text{newPath} = \text{path.copy()}$
        $\text{newPath.append(neighbour)}$
        $\text{if FuelConstraint(newPath):}$
            $\text{queue.append(newPath)}$
    $\text{queue.sort(key} = \text{Heuristic)}$
```

In essence, this Greedy BFS algorithm can find a path satisfying the fuel constraint with minimal number of stops to stations in real time. This path has the same property described above as the ideal solution path. Further implementation of advanced optimization techniques is needed for minimizing the distance travelled.

3.2.4 Conclusion

The Greedy Algorithm is typically fast due to its straightforward decision-making process. However, the algorithm does not guarantee an optimal solution; it is prone to shortsightedness. Certain route configurations or distributions of positions and fuel stations might lead to suboptimal paths, especially if early greedy choices preclude better options later on.

3.3. A* Algorithm

In this part, we introduce the A* 1-step look ahead algorithm (A* 1-step in short), which is the well-known A* algorithm. One of the key features of A* is the use of a heuristic function (h) that estimates the cost from the current node to the goal.[4] This heuristic helps guide the algorithm towards the goal efficiently. The heuristic function should be admissible, meaning it never overestimates the cost to reach the goal. In other words, it should always be optimistic. A* uses an evaluation function, often denoted as $\mathbf{f(n) = g(n) + h(n)}$. This function combines the cost of the path from the start node to the current node (***g(n)***) with the heuristic estimate of the cost from the current node to the goal (***h(n)***). The most commonly used heuristic estimator is Euclidean distance, as it consistently provides the minimum distance between any two points in space. [5]Therefore, Euclidean distance is an admissible heuristic for A* algorithm. Moreover, a heuristic is deemed consistent (or monotonic) if it meets the condition $\mathbf{h(n) \leq c(n, s) + h(s)}$ and $\mathbf{h(t) = 0}$, where ***s*** represents a successor of any node n in the graph, denotes that $\mathbf{c(n, s)}$ is the actual cost to travel from node ***n*** to node ***s***. In the context of the A* algorithm, using a consistent heuristic guarantees that A* will find the optimal solution (i.e., the shortest path) to a given problem.

However, the problem we propose requires traversing nodes labeled with levels as a

[4] S. Russell and P. Norvig (2010), Artificial Intelligence: A Modern Approach. Pearson Education, 3rd ed.

[5] R. Dechter and J. Pearl (1985), “Generalized best-first search strategies and the optimality of A,” Journal of the ACM (JACM), vol. 32, no. 3, pp. 505–536. Publisher: ACM New York, NY, USA.

mandatory constraint. Therefore, here we apply the concepts of backtracking, dynamic programming, and recursion. The problem can be divided into subproblems of finding a path from the previous level to the current level, satisfying constraints on fuel limitations. For each level, the problem is also divided into two stages: starting from a node labeled with the level and starting from a node labeled as a station, named *PointPath* and *StationPath,* respectively.

In the *PointPath* stage, we iterate through nodes at the current level, then choose a node with the smallest ***f(n)*** value. We perform a check on the condition: is there enough fuel to reach that node? If yes, we add that node to the current path. Otherwise, we proceed to search for a node labeled as a station, applying a similar ***f(n)*** calculation manner to select the station with the smallest ***f(n)*** value. We continue checking the fuel condition, and if it is not sufficient, we apply backtracking, choosing the station with the second smallest ***f(n).*** This process repeats until we find a suitable station. If no suitable station can be chosen, we apply backtracking again, searching for a node at the previous level different from the last node in the path based on the above idea, simultaneously removing the last node from the path. This is similar to performing PointPath with the level reduced by 1, but we do not consider the recently removed node.

Once a suitable station is found, we can proceed to the next stage, StationPath. The idea is similar to PointPath, with the difference that if the fuel condition is not satisfied, our path may contain multiple stations. However, the algorithm cannot loop infinitely because, in designing the data, there will always exist at least one pair of nodes labeled as a station and a node labeled with a level at each level, satisfying the fuel condition. In other words, there will always be a path from this level to a station as well as from a station to another level.

Algorithm 1 PointPath Function

Input:

1. path, state(distance move, fuel level), level
2. G1(V1,E1): graph between 2 levels' nodes
3. Gs(Vs,Es): graph between stations and all nodes

```
function PointPath
    $curr\_level \leftarrow state.level$
    $priorityQueue \leftarrow$ priority queue of unvisited nodes sorted by 2 keys: level and $f$ value
    while priorityQueue.isEmpty() is not True do
        $current \leftarrow$ priorityQueue.get()
        $traverse\_level \leftarrow$ current node's level
        if $traverse\_level \geq$ curr_level then
            pass
        else
            Remove the last node from the path
            $state \leftarrow$ Recover previous level's state
            $curr\_level \leftarrow traverse\_level$
        end if
        if Fuel constraint is violated then
            $priorityQueue\_Station \leftarrow$ priority queue of valid stations sorted by $f$ value
            if priorityQueue_Station.empty() then
                continue
            else
                $station \leftarrow$ priorityQueue_Station.get()
                Add $station$ to the path
                $state \leftarrow$Update new state
                Empty $priorityQueue\_Station$
                Find next level's node via STATIONPATH function
            end if
        else
            Add $current$ to the path
            $state \leftarrow$Update new state
        end if
        if $curr\_level \geq level$ then
            return $path, state$
        else
            for $i$ in range(level - curr_level $- 1, -1, -1$) do
                $path, state \leftarrow$ PointPath
            end for
            return $path, state$
        end if
    end while
end function
```

1

Algorithm 2 StationPath Algorithm

Input:

1. path, state(distance move, fuel level), level
2. Gs(Vs,Es): graph between stations and all nodes

```
function STATIONPATH
    $pQueue$ ← priority queue of unvisited nodes at current level sorted by $f$ value
    while $pQueue.empty()$ is not True do
        $current$ ← $pQueue.get()$
        if Fuel constraint is violated then
            $priorityQueue\_Station$ ← priority queue of valid stations sorted by $h$ value
            if $priorityQueue\_Station.empty()$ then
                continue
            else
                $station$ ← $priorityQueue\_Station.get()$
                Add $station$ to the path
                $state$ ←Update new state
                Empty $priorityQueue\_Station$
                Find next level's node via STATIONPATH function
            end if
        else
            Add $current$ to the path
            $state$ ←Update new state
        end if

```

3.4. A* 2-step look-ahead heuristic

Here we apply the idea of A* algorithm with 2-step look ahead heuristic (A* 2-step in short). [6]**T**heir k-step look-ahead heuristic performs shorter runtime compare to general Euclidean heuristic. They also studied that k=2 is appropriate to small number of nodes. In this study, we apply k=2, try to find the lowest distance travel route for our own data map.

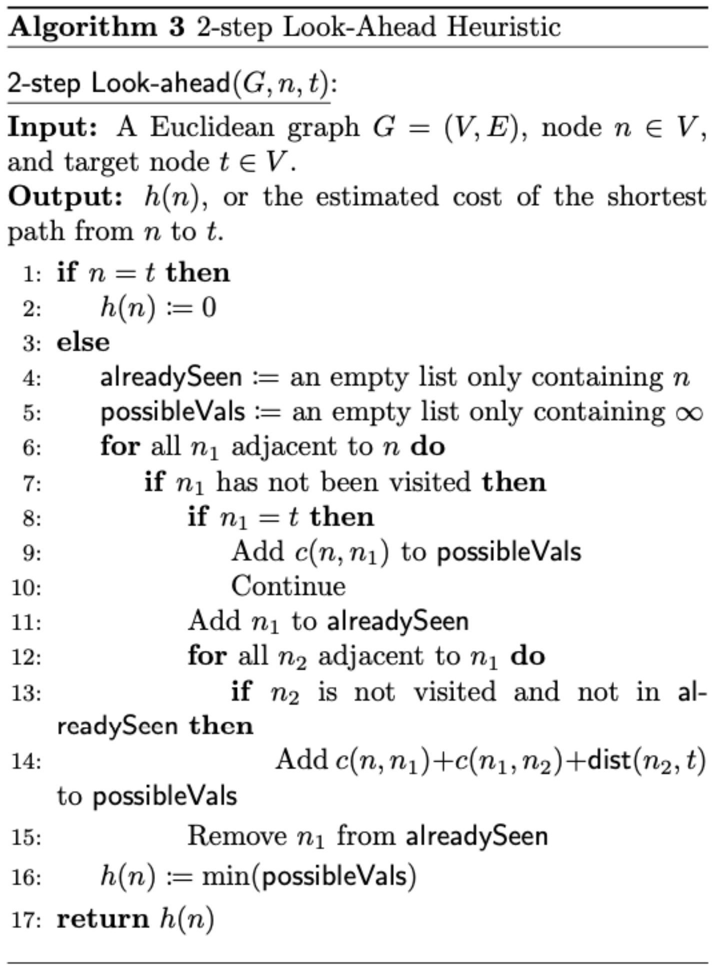
A* 2-step look-ahead heuristic means that we estimate the cost to the destination from the unvisited adjacent node of the current node. For example, suppose we need to find a path from node s to node e. Denotes that N1 is a list that contains all adjacent nodes to node s, N2 contains all unvisited adjacent nodes of all nodes appear in N1. So for popular heuristic, Euclidean distance, $\mathbf{f(n1) = c(s, n1) + dist(n1, e)}$. Here we define $\mathbf{c(s, n1)}$ as actual cost and $\mathbf{dist(n1, e)}$ is the Euclidean distance to travel from n1 to e. For the new heuristic, 2-step look-ahead heuristic, $\mathbf{f(n1) = min(dist(n1, n2) + dist(n2, e))}$ for all n2 in N2.

This 2-step look-ahead heuristic is admissible and consistent, as h(n) will never be greater than the cost of the true shortest path from n to t, so it is suitable for shortest path finding. Moreover, this heuristic seems to be more efficient because **i**t estimates the cost to the destination more accurately and closely to the actual cost compare to euclidean

[6]Kevin Y. Chen, An Improved A* Search Algorithm for Road Networks Using New Heuristic Estimation.

distance heuristic.

To implement this idea to our own problem, we follow the same idea as above, but the only change is the heuristic function. Additionally, if the finish point is in the visiting, the heuristic function still is the Euclidean distance heuristic because there is no extra level after finish point to estimate the cost. Our problem divided into levels and cannot traverse from current level to previous level, so this heuristic is very appropriate and easy to implement.



Algorithm 3 2-step Look-Ahead Heuristic

2-step Look-ahead(G, n, t):

Input: A Euclidean graph $G = (V, E)$, node $n \in V$, and target node $t \in V$.

Output: $h(n)$, or the estimated cost of the shortest path from n to t.

```
1:  if $n = t$ then
2:      $h(n) := 0$
3:  else
4:      alreadySeen := an empty list only containing $n$
5:      possibleVals := an empty list only containing $\infty$
6:      for all $n_1$ adjacent to $n$ do
7:          if $n_1$ has not been visited then
8:              if $n_1 = t$ then
9:                  Add $c(n, n_1)$ to possibleVals
10:                 Continue
11:             Add $n_1$ to alreadySeen
12:             for all $n_2$ adjacent to $n_1$ do
13:                 if $n_2$ is not visited and not in alreadySeen then
14:                     Add $c(n, n_1) + c(n_1, n_2) + \mathrm{dist}(n_2, t)$ to possibleVals
15:             Remove $n_1$ from alreadySeen
16:     $h(n) := \min(\text{possibleVals})$
17: return $h(n)$
```

3.5. Bidirectional A* Search

Bidirectional A* search is an advanced variant of the A* search algorithm, a widely used heuristic search algorithm for finding the shortest path between two points in a graph. The key innovation in Bidirectional A* lies in its simultaneous exploration of the solution space from both the initial and goal states. This bidirectional exploration aims to meet at a central point, then return the path from both the forward and backward directions.

First, the maximum distance constraint will not be implemented directly in the bidirectional A* algorithm, which means for the first searching process, we will only take into consideration the graph between ‘Start’, ‘Finish’ and all the LevelNode nodes. The algorithm employs two priority queues and efficiently explores the graph bidirectionally, searching simultaneously from both the start and goal nodes towards each other. Heuristic values provided for forward and backward directions aid in estimating costs, which has no difference to the heuristic function in the traditional A* searching algorithm. But, for the backward directions, instead of using the distance between one node and the ‘Finish’ node, we will calculate the straight-line distance between one node and the ‘Start’ node. The code maintains distances, parent relationships, and efficiently updates paths as it traverses the graph, ultimately identifying a meeting point where the forward and backward searches converge.

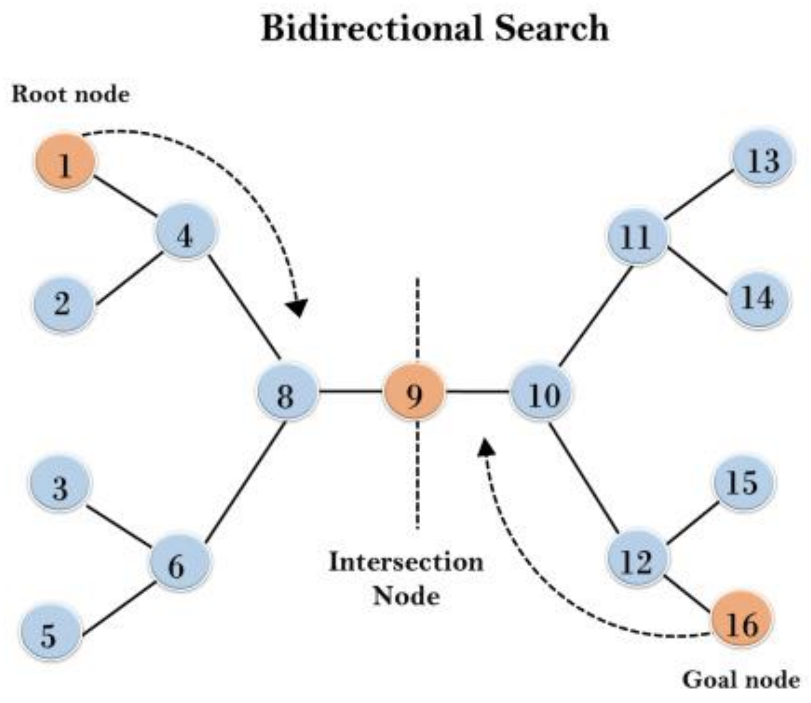
Algorithm 1 Bidirectional A* Algorithm

```
procedure BidirectionalAStar(graph, start, goal, forward_heuristics, backward_heuristics
    forward_queue ← [(0, start)]
    backward_queue ← [(0, goal)]
    forward_visited ← set()
    backward_visited ← set()
    forward_distances ← {start : 0}
    backward_distances ← {goal : 0}
    forward_parents ← {start : None}
    backward_parents ← {goal : None}
    best_path ← None
    best_cost ← $\infty$
    node_explored ← 0
    while forward_queue and backward_queue do
        forward_cost, forward_node ← heappop(forward_queue)
        forward_visited.add(forward_node)
        node_explored ← node_explored + 1
        if forward_node $\in$ backward_visited then    ▷ Found a meeting point
            cost ← forward_distances[forward_node] + backward_distances[forward_node]
            if cost < best_cost then
                best_cost ← cost
                best_path ← GetPath(forward_parents, backward_parents, forward_node)
            end if
        end if
        for neighbor, distance in graph[forward_node] do
            new_cost ← forward_distances[forward_node] + distance
            if neighbor $\notin$ forward_distances then
                forward_distances[neighbor] ← new_cost
                forward_parents[neighbor] ← forward_node
                heappush(forward_queue, (new_cost + forward_heuristics[neighbor], neighbor))
            end if
        end for
        backward_cost, backward_node ← heappop(backward_queue)
        backward_visited.add(backward_node)
        node_explored ← node_explored + 1
        if backward_node $\in$ forward_visited then    ▷ Found a meeting point
            cost ← backward_distances[backward_node] + forward_distances[backward_node]
            if cost < best_cost then
                best_cost ← cost
                best_path ← GetPath(forward_parents, backward_parents, backward_node)
            end if
        end if
        for neighbor, distance in new_graph[backward_node] do
            new_cost ← backward_distances[backward_node] + distance
            if neighbor $\notin$ backward_distances then
                backward_distances[neighbor] ← new_cost
                backward_parents[neighbor] ← backward_node
                heappush(backward_queue, (new_cost + backward_heuristics[neighbor], neighbor))
            end if
        end for
    end while
    return best_path, best_cost, node_explored
end procedure
```

The resulting path is reconstructed by combining the paths from the start to the meeting point and from the goal to the meeting point. In the case of no meeting node is found, then we compare the distance between the forward direction and the backward direction to finalize the temporary path.

Algorithm 1 Get Path Function

```
procedure GetPath($forward\_parents, backward\_parents, meeting\_node$)
    $path \leftarrow []$
    $node \leftarrow meeting\_node$
    while $node \neq None$ do
        $path.insert(0, node)$
        $node \leftarrow forward\_parents[node]$
    end while
    $node \leftarrow backward\_parents[meeting\_node]$
    while $node \neq None$ do
        $path.append(node)$
        $node \leftarrow backward\_parents[node]$
    end while
    return $path$
end procedure
```



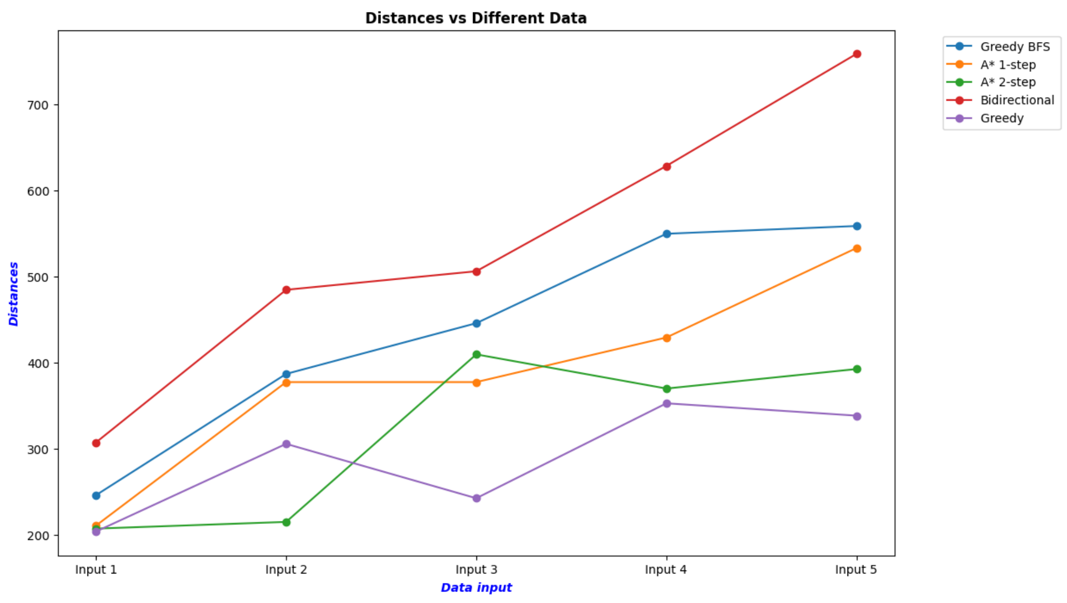
Example of bidirectional search

Secondly, we will implement the maximum constraint in the previously given path using bidirectional A* searching algorithm. In this process, we will simply iteratively go through each node, calculate the total distance from the starting node, if the total distance exceeds the maximum distance given, then at that node, we will travel to the nearest StationNode node within range, and append it to the temporary route in the first process, and simultaneously update the total distance by adding up the distance from a ‘LevelNode’ node coming to a ‘StationNode’ node and from a ‘StationNode’ node to a ‘LevelNode’ node. Keep repeating the process until the ‘Finish’ node is reached, then return the final path and the total distance.

4. Results

The following table shows the number of **nodes expanded** and the **solution path's distances** for five different data set with five different fuel constraints of each algorithm. Here, the Greedy BFS refers to the Greedy algorithm with the Mininal Unvisited Levels.

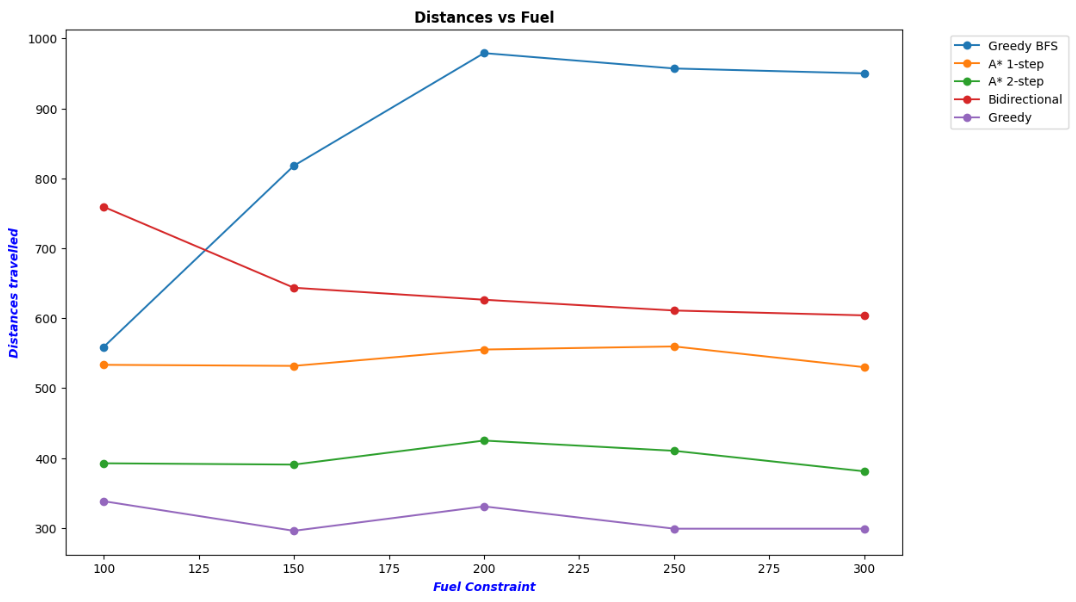
Data Set	Fuel Capacity (km)	Greedy BFS (1)		Greedy BFS (2)		A* 1 Step		A* 2 Step		Bidirectional A*	
		Distance moved	Node Expanded	Distance moved	Node Expanded	Distance moved	Node Expanded	Distance moved	Node Expanded	Distance moved	Node Expanded
Data 1	100	203.70	28	246.00	180	216.60	10	207.61	10	307.31	102
	150	238.42	6	289.00	11	227.76	8	216.98	10	246.52	102
	200	238.42	6	319.00	9	212.36	8	206.85	8	282.55	102
	250	225.30	6	381.00	10	210.93	6	205.42	6	210.93	102
	300	225.30	6	391.00	74	210.93	6	205.42	6	210.93	102
Data 2	100	305.68	37	387.00	41	337.50	17	214.92	15	484.82	242
	150	263.06	24	521.00	33	380.89	15	214.25	13	449.16	242
	200	252.94	11	504.00	129	380.27	13	214.25	13	411.55	242
	250	252.94	11	571.00	72	334.50	39	211.74	11	433.96	242
	300	252.92	11	630.00	205	375.65	13	211.74	11	450.28	242
Data 3	100	242.44	34	446.00	433	403.48	24	409.73	94	506.38	512
	150	290.40	34	592.00	24	411.02	20	396.16	38	416.67	512
	200	249.79	16	696.00	30	500.04	20	375.29	36	408.28	512
	250	247.16	16	617.00	22	372.76	54	366.91	18	442.44	512
	300	247.16	16	684.00	54	409.13	18	373.63	38	408.33	512
Data 4	100	352.83	23	550.00	66	429.37	61	370.00	59	628.86	602
	150	447.38	21	803.00	190	473.38	27	347.89	41	561.40	602
	200	343.85	37	698.00	31	427.53	57	344.71	39	554.32	602
	250	330.38	21	815.00	60	443.38	39	345.95	39	493.88	602
	300	330.38	21	804.00	26	453.71	23	345.40	23	474.12	602
Data 5	100	338.45	68	559.00	226	533.41	163	392.68	34	759.22	1002
	150	296.17	26	818.00	645	531.91	32	390.81	30	634.60	1002
	200	331.04	109	979.00	43	555.30	156	425.18	92	626.46	1002
	250	299.17	26	957.00	40	559.68	72	410.52	70	611.06	1002
	300	295.08	26	950.00	347	530.04	28	381.19	49	604.11	1002



The fuel limit is 100km

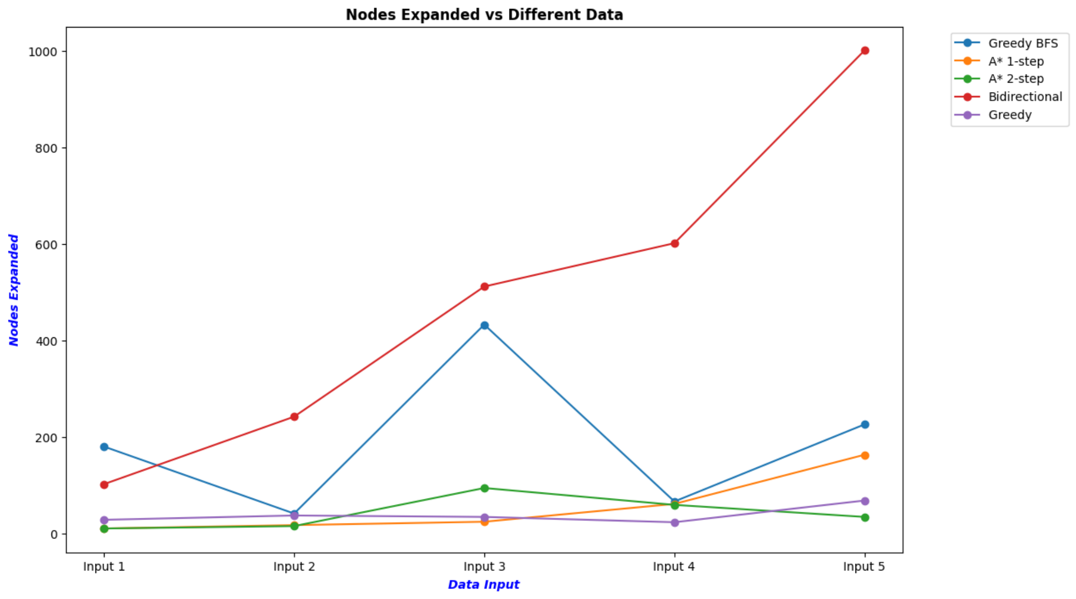
In the case of the fuel capacity is 100, the bidirectional A* algorithm returned the greatest distance among all the algorithms, and the rate of increase in distance also seems to be the highest corresponding to the increasing number of levels. The A* 2-step look-ahead algorithm and the Greedy algorithm tends to perform unstably, presented by the high rate of change in distance comparing to A* 1-step look-ahead algorithm and the

Greedy BFS algorithm. Last but not least, the Greedy BFS algorithm and the A* 1-step look-ahead algorithm’s performance seems to be pretty stable and has the ability to return a fairly optimized solution in terms of distance compared with other algorithms.



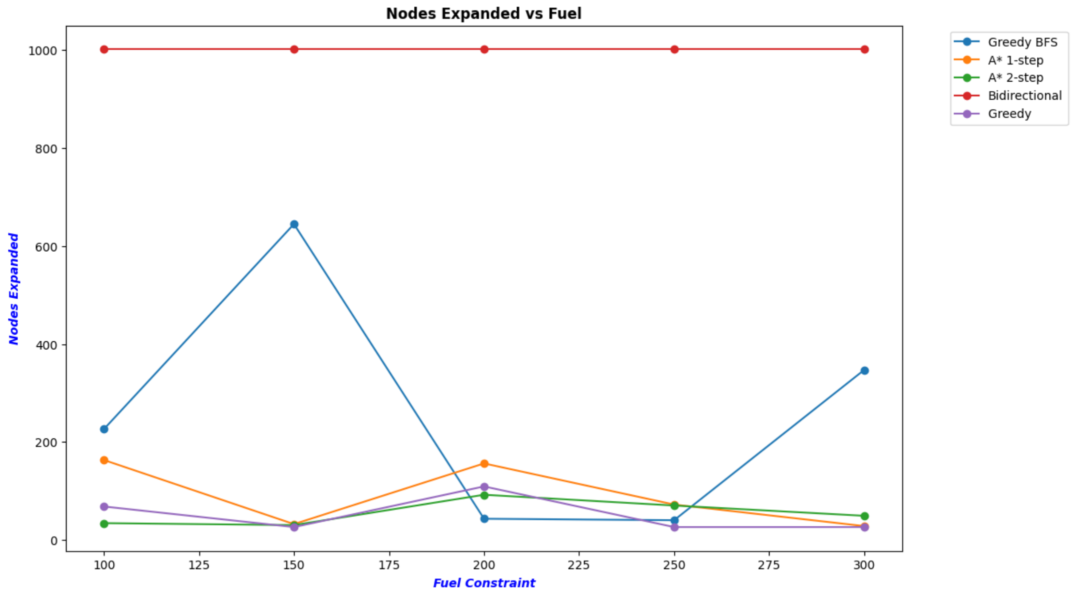
Same data set for different constraints

In the case of maintaining the number of levels to be 5 and changing the fuel capacity, it can be seen that the Greedy BFS has the highest distance for almost every fuel capacity, and its growth rate in terms of distance seems to be pretty high in relation to the increase in fuel capacity. For the bidirectional A* algorithm, the total distance in every case seems to be relatively high compared to algorithms, but it also seems to be inversely proportional to the growth of fuel capacity. Other algorithms have the same trend, but the Greedy algorithm always has the lowest distance in all cases, and the difference in terms of distance is pretty high.



The fuel constraint is 100km.

In the case of fixing the fuel capacity to be 100 and modifying the number of levels, it is observable that the number of expanded node when Bidirectional A* algorithm used is the highest among all algorithms, and it tends to increase rapidly when the fuel capacity is increased. Moreover, the rate of change in the number of nodes explored in the case of Greedy BFS algorithm is also quite high, but not always in the trend of increasing. The other 3 algorithms have almost the same number of nodes in every cases, with almost no rate of change in the case of Greedy and A* 2-step look-ahead algorithm.



In the case of maintaining the number of levels be 25 and modifying the fuel capacity, the Bidirectional A* algorithm has the highest number of expanded nodes, and for every fuel capacity number, it stays the same. This is due to the fact that this number of expanded nodes only takes into account the process of searching from two directions backwards and forwards, not considering the fuel capacity constraint. For other algorithms, it is evident that the rate of change of the Greedy BFS algorithm is the largest, and the number of explored node only comes second right below Bidirectional A* algorithm. For the rest of the algorithms, the situation seems to be the same as the previous case of changing the number of levels and fixing the fuel capacity.

5. Discussion

In this study, our objective is to assess the effectiveness of five pathfinding algorithms in a fuel-constrained routing problem. After conducting experiments with five sets of random data (attached in the accompanying file) and 5 values of fuel capacity, we obtained some promising results.

Firstly, we applied two popular algorithms, A* 1-step and greedy with Nearest distance to the next position heuristic. The experimental results continue to demonstrate that these algorithms are effective in finding the shortest path, even in the presence of fuel constraint. However, as the dataset is random, parameters such as distance moved and node expanded may exhibit occasional anomalies. Nevertheless, overall, these two algorithms perform efficiently as expected.

Subsequently, we implemented three less common algorithms: A* 2-step, greedy with Minimal Number of Unvisited Levels, and bidirectional A*. Experimental results indicated that greedy with Minimal Number of Unvisited Levels and bidirectional A* exhibit low effectiveness and are not suitable for this dataset. This is evident in the parameters of distance moved and nodes expanded, which are consistently the highest for both algorithms. In terms of Bidirectional A* algorithm, this inefficiency can be explained by the fact that this algorithm considers a large number of paths for both the forward and backward directions. In terms of greedy Minimal Number of Unvisited Levels, the algorithm doesn’t take into account the distance of the path it has travelled. It only strives to get to the next level while trying not to visit any stations if not needed. On the other hand, A* 2-step proves to be one of the effective algorithms. In fact, it is the most efficient algorithm for some datasets and fuel capacities.

Examining the parameter of nodes expanded may not clearly indicate the algorithm's effectiveness as the dataset consists of randomly initialized values. However, through analysis, it is evident that a smaller number of nodes expanded corresponds to a more efficient algorithm. Simultaneously, a higher number of nodes expanded leads to longer execution times.

Through experimentation, it can be concluded that A* and greedy algorithms remain the most effective, widely applied in various practical real-life problems. Although the A* 2-step look-ahead heuristic is relatively new, it demonstrates high effectiveness.

6. Conclusion

Overall the A* k-step look-ahead algorithm (k = 2) and the Greedy with Nearest neighbour heuristic perform quite well. It seems that the larger k is, the better the A* informed search algorithm becomes. The result also shows the importance of choosing a suitable heuristic for the Greedy algorithm by displaying that the Nearest Neighbour heuristic outperforms the Minimal Univisited Levels one. In most cases, algorithms whose performances are stable through our different data sets should be chosen.

Still, the problem of identifying exactly a lower bound for the fuel constraint given the positions of stations and level points remains, in a sense that any value no less that that lower bound could be taken as the fuel limit.

7. References

[1] Khan, M. A. (2023). A comprehensive study of Dijkstra's algorithm. *Social Science Research Network.*

[2] Foead, D., Ghifari, A., Kusuma, M. B., Hanafiah, N., & Gunawan, E. (2021). A Systematic Literature review of A* pathfinding. *Procedia Computer Science*, *179*, 507–514

[3] Barker, J. K. (2015). *Front-To-End bidirectional heuristic search. University of California, Los Angeles ProQuest Dissertations Publishing, 2015. 3687357.*

[4] S. Russell and P. Norvig (2010), Artificial Intelligence: A Modern Approach. *Pearson Education, 3rd ed*

[5] R. Dechter and J. Pearl (1985), "Generalized best-first search strategies and the optimality of A," Journal of the ACM (JACM), vol. 32, no. 3, pp. 505–536. Publisher: ACM New York, NY, USA.

[6] Kevin Y. Chen, An Improved A* Search Algorithm for Road Networks Using New Heuristic Estimation