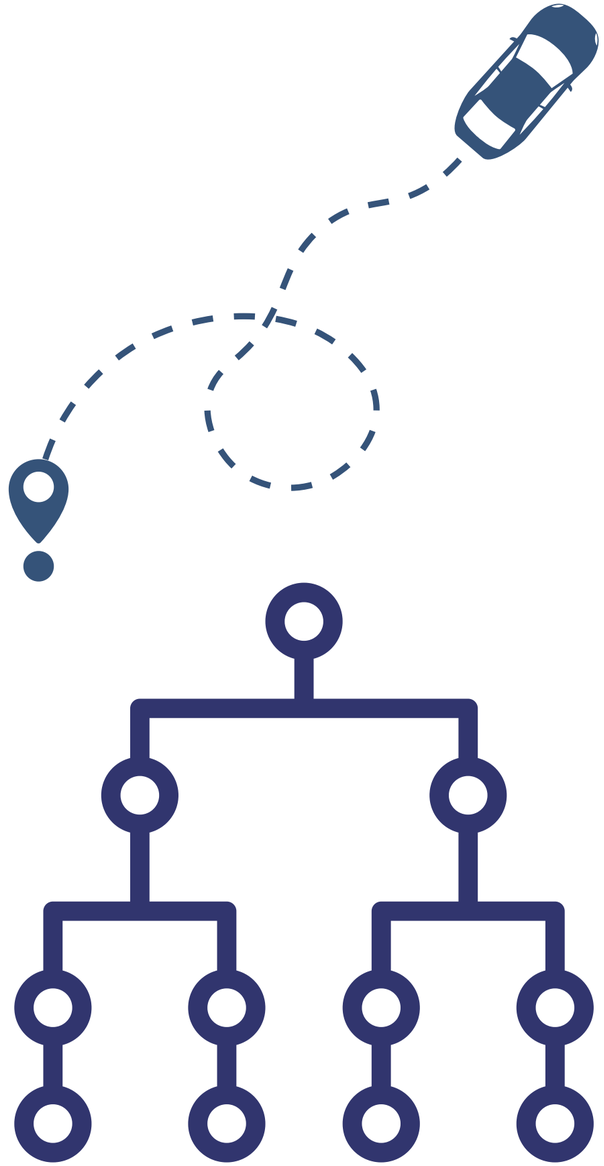
JAN. 09, 2024 • HUST

Design an optimal route with fuel constraints

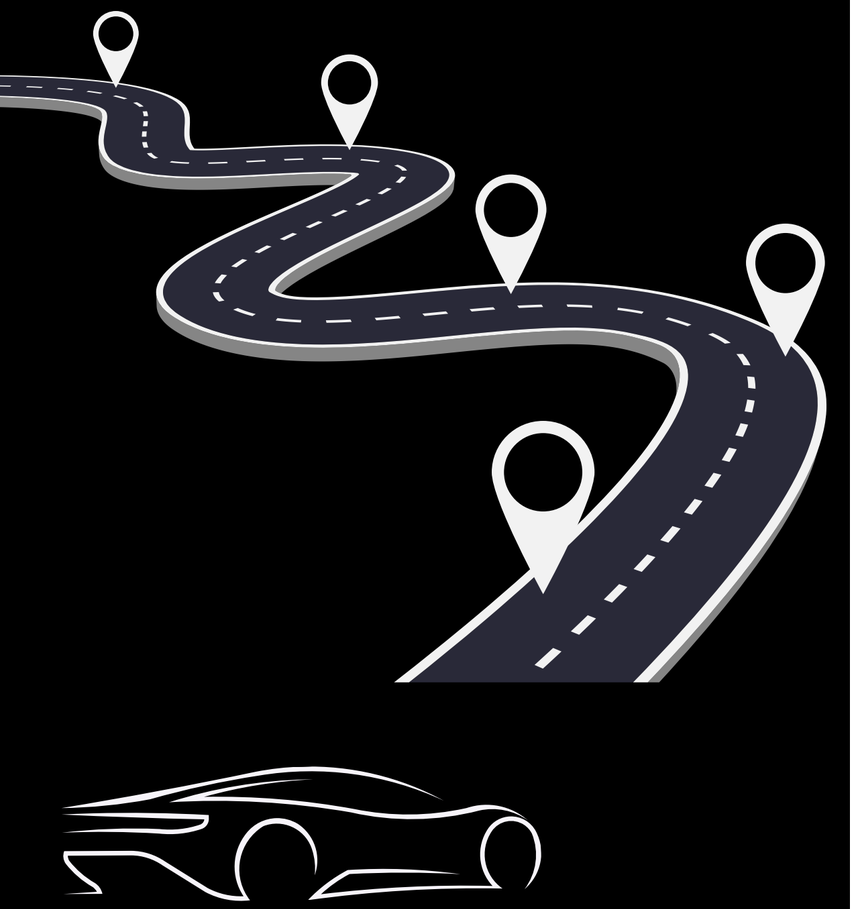


HUST • Jan. 9, 2024

Group 21

HUST • Dec. 25, 2023

Table of Contents



INTRODUCTION

Design an optimal route
with fuel constraints

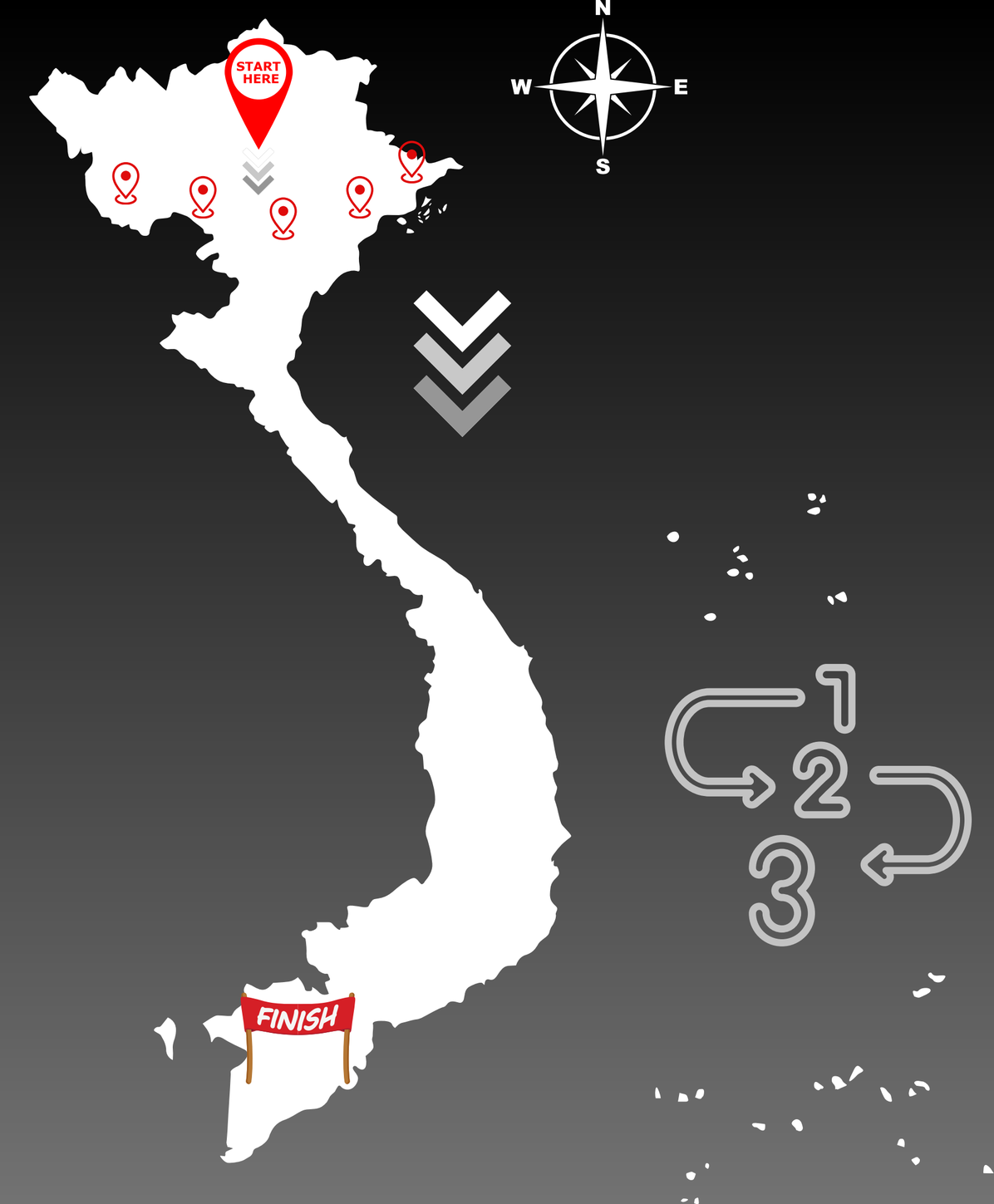
GROUP 21

Objective

Developing an AI-based solution to find the shortest path in a race from the Start to Finish positions while considering fuel constraints and mandatory sequential level-based stops

Approach

Utilize graph representation for efficient pathfinding, enforce fuel constraints, and ensure sequential level traversal for optimal path selection.

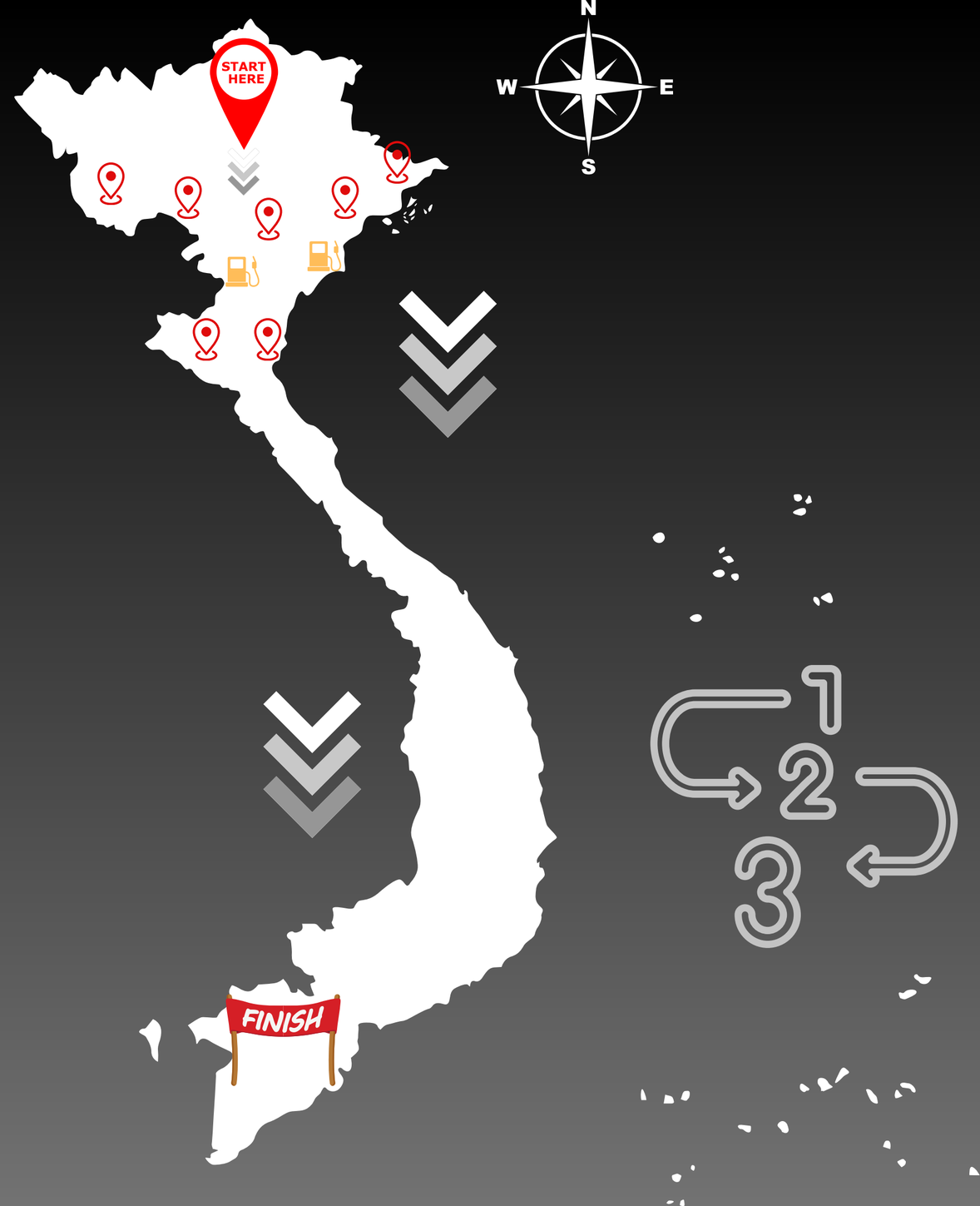


Objective

Developing an AI-based solution to find the shortest path in a race from the Start to Finish positions while considering fuel constraints and mandatory sequential level-based stops

Approach

Utilize graph representation for efficient pathfinding, enforce fuel constraints, and ensure sequential level traversal for optimal path selection.



Objective

Developing an AI-based solution to find the shortest path in a race from the Start to Finish positions while considering fuel constraints and mandatory sequential level-based stops

Approach

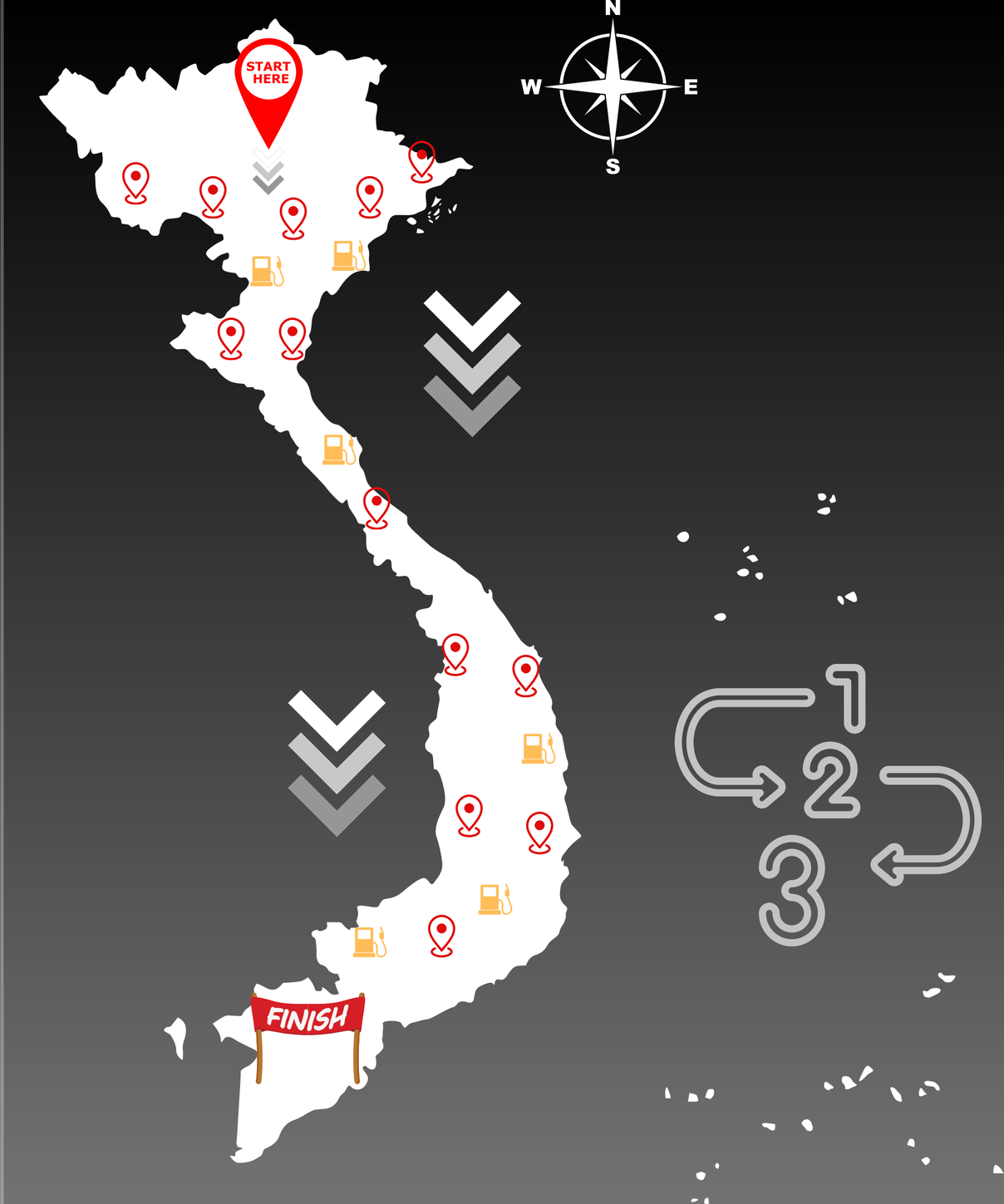
Utilize graph representation for efficient pathfinding, enforce fuel constraints, and ensure sequential level traversal for optimal path selection.

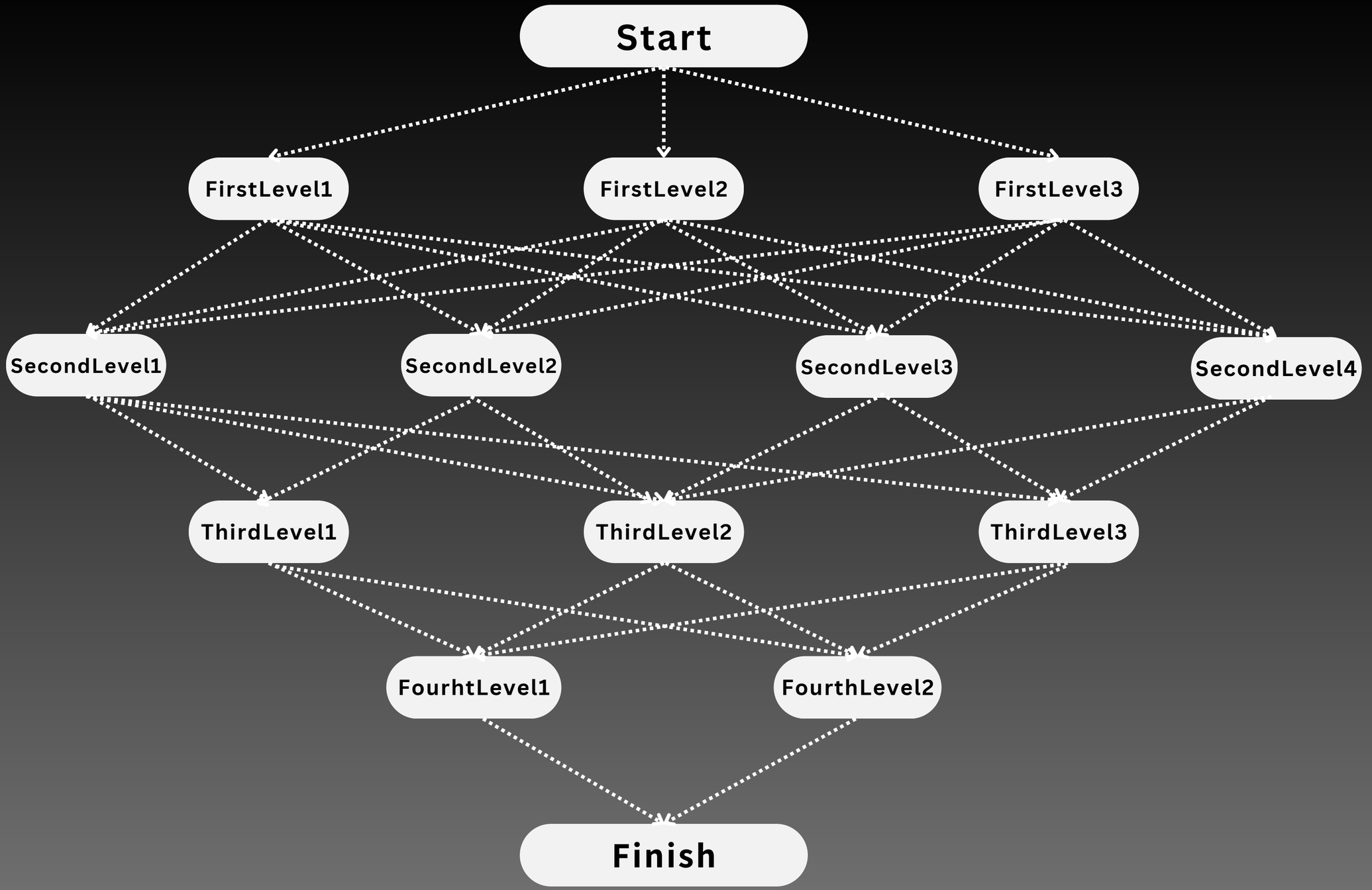
Objective

Developing an AI-based solution to find the shortest path in a race from the Start to Finish positions while considering fuel constraints and mandatory sequential level-based stops

Approach

Utilize graph representation for efficient pathfinding, enforce fuel constraints, and ensure sequential level traversal for optimal path selection.



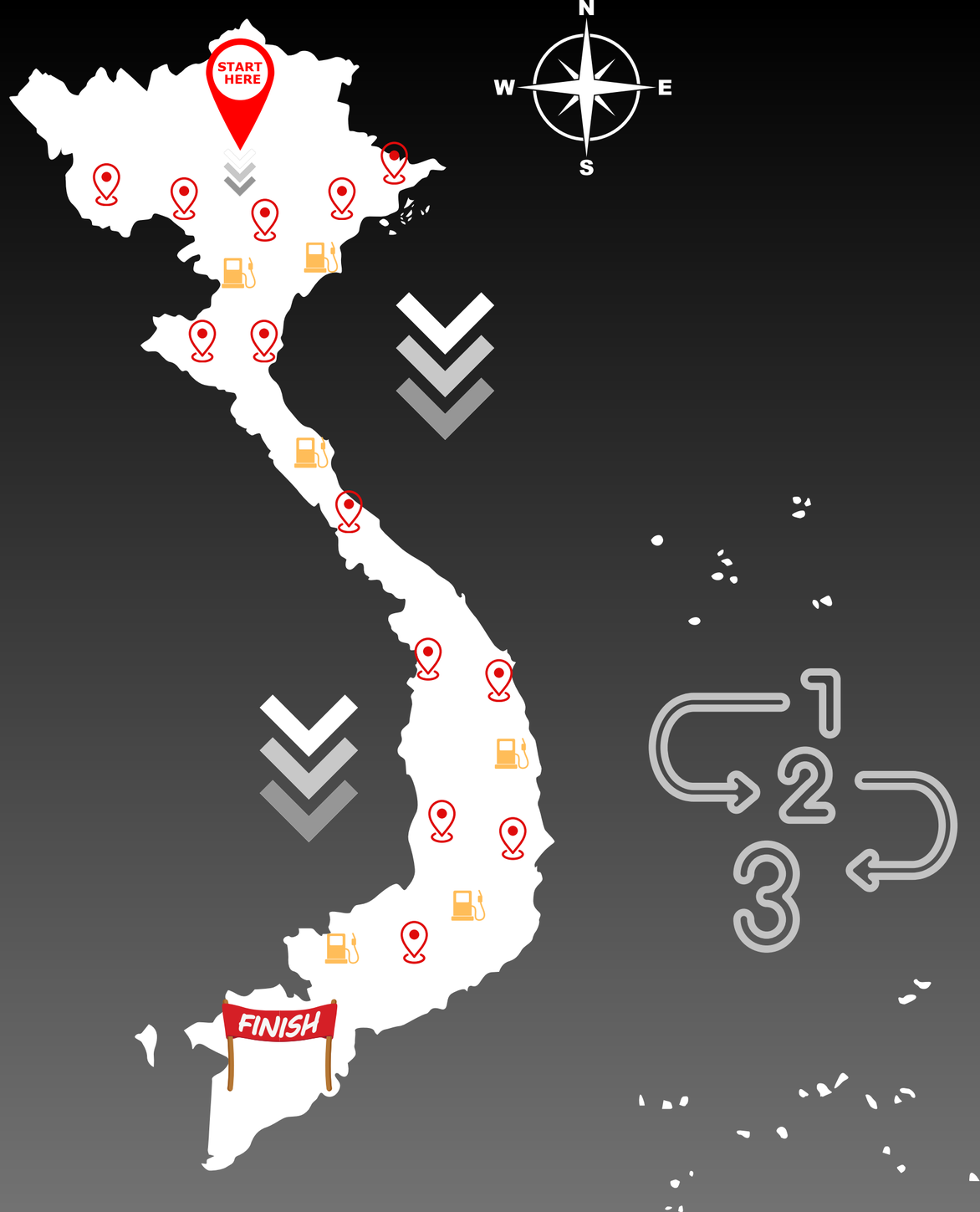

Start
FirstLevel1
FirstLevel2
FirstLevel3
SecondLevel1
SecondLevel2
SecondLevel3
SecondLevel4
ThirdLevel1
ThirdLevel2
ThirdLevel3
FourhtLevel1
FourthLevel2
Finish

Objective

Developing an AI-based solution to find the shortest path in a race from the Start to Finish positions while considering fuel constraints and mandatory sequential level-based stops

Approach

Utilize graph representation for efficient pathfinding, enforce fuel constraints, and ensure sequential level traversal for optimal path selection.



Key Features of the AI Model

GRAPH REPRESENTATION

Modeling the track as a graph for efficient pathfinding and distance calculation.

FUEL CONSTRAINTS

Considering the limited fuel capacity for strategic refueling planning.

SEQUENTIAL LEVELS

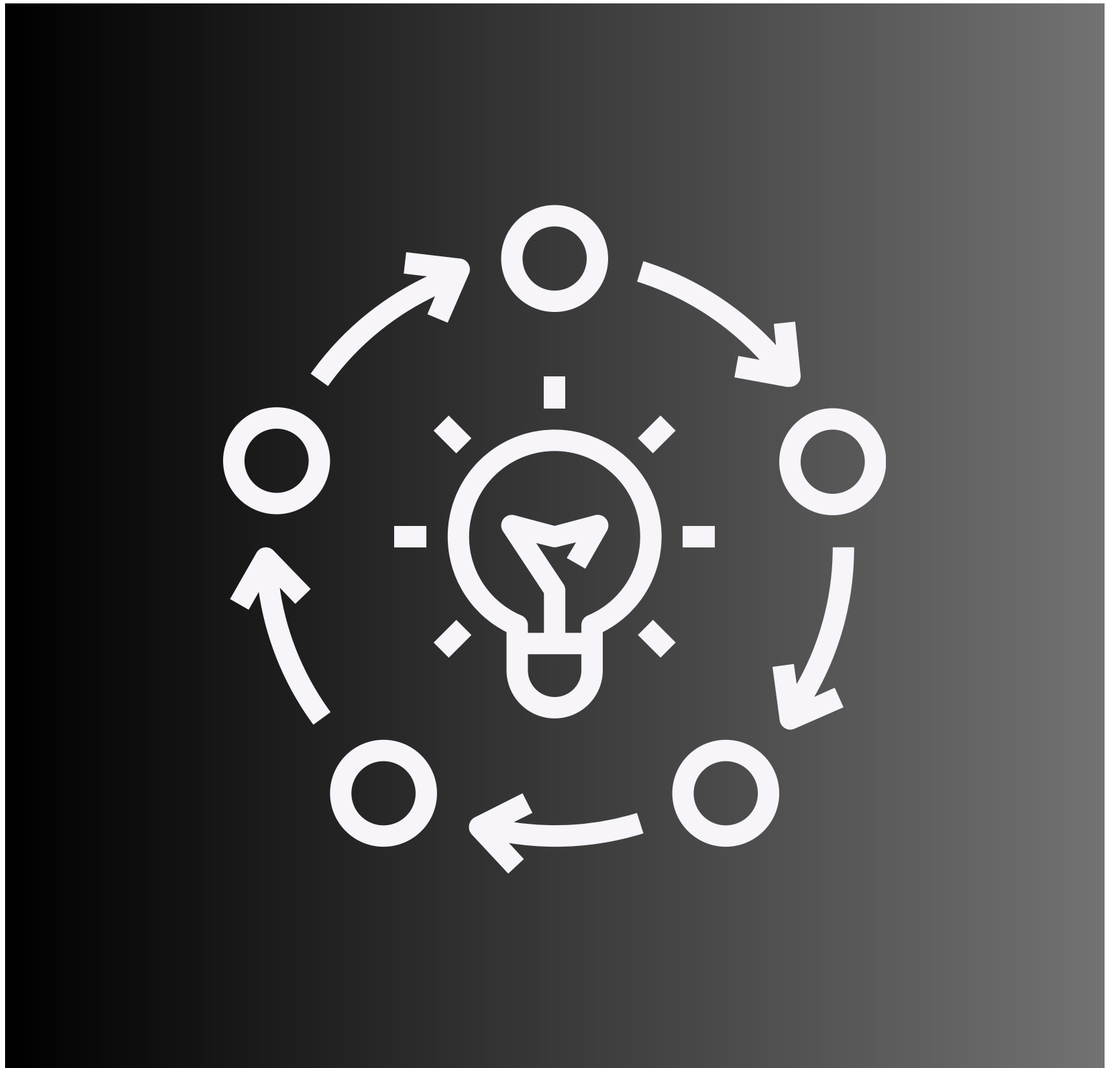
Ensuring that positions are passed through in a specific order, from level **n-1** to level **n**.

OPTIMAL PATH CALCULATION

Using dynamic programming and backtracking for constrained shortest path.

HUST • Jan. 9, 2024

DESIGN AN OPTIMAL ROUTE WITH FUEL CONSTRAINTS



Intuition and Methodology

- **Intuitive Idea:** Constructing a racetrack graph to minimize travel distance within **fuel constraints** while ensuring **sequential level progression**.

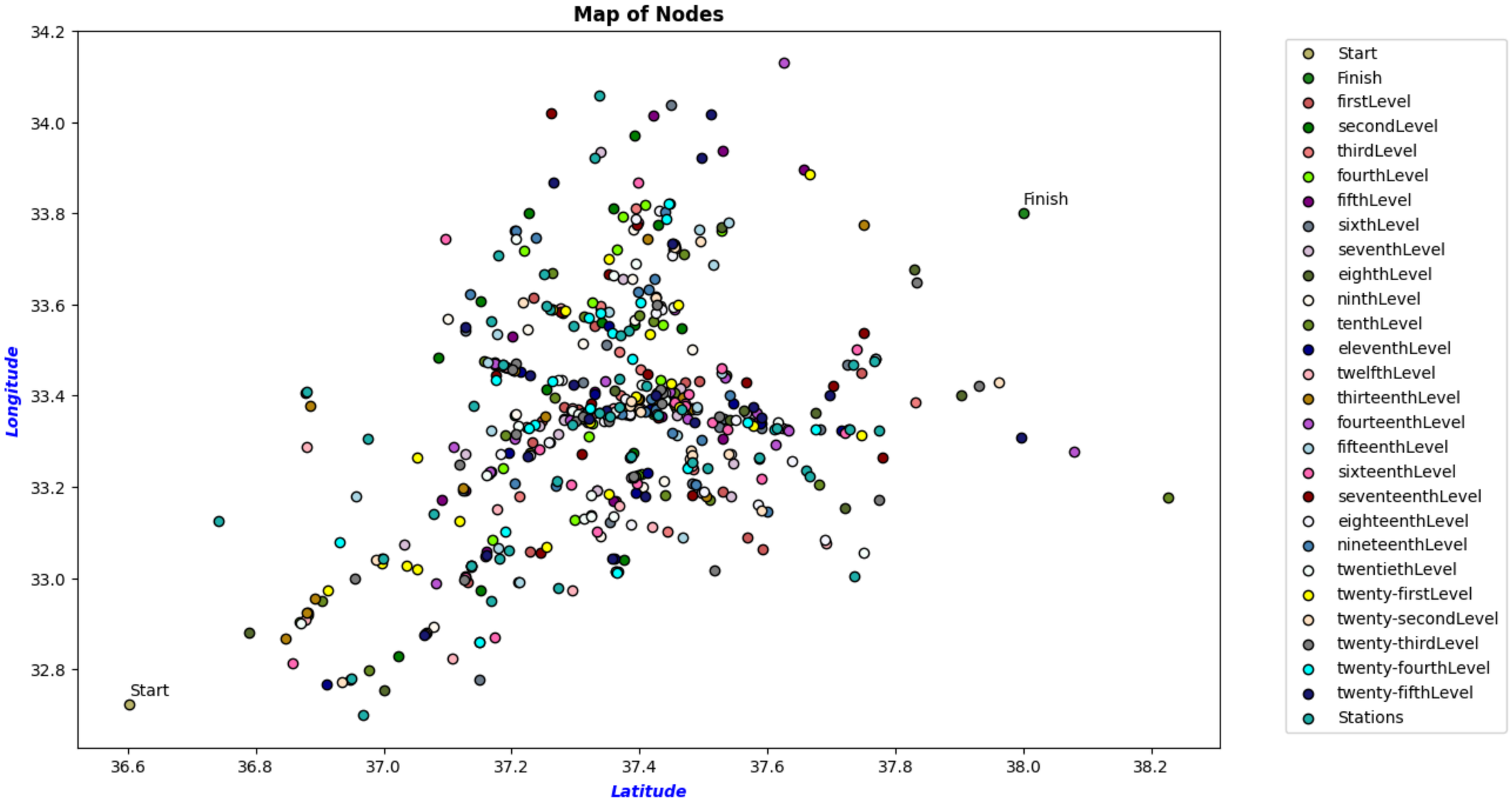
- **Methodology:** Employing dynamic programming for optimal path calculation and backtracking for solution retrieval, focusing on the balance between covering necessary positions and refueling strategically.

MODELLING

Problem Formulation

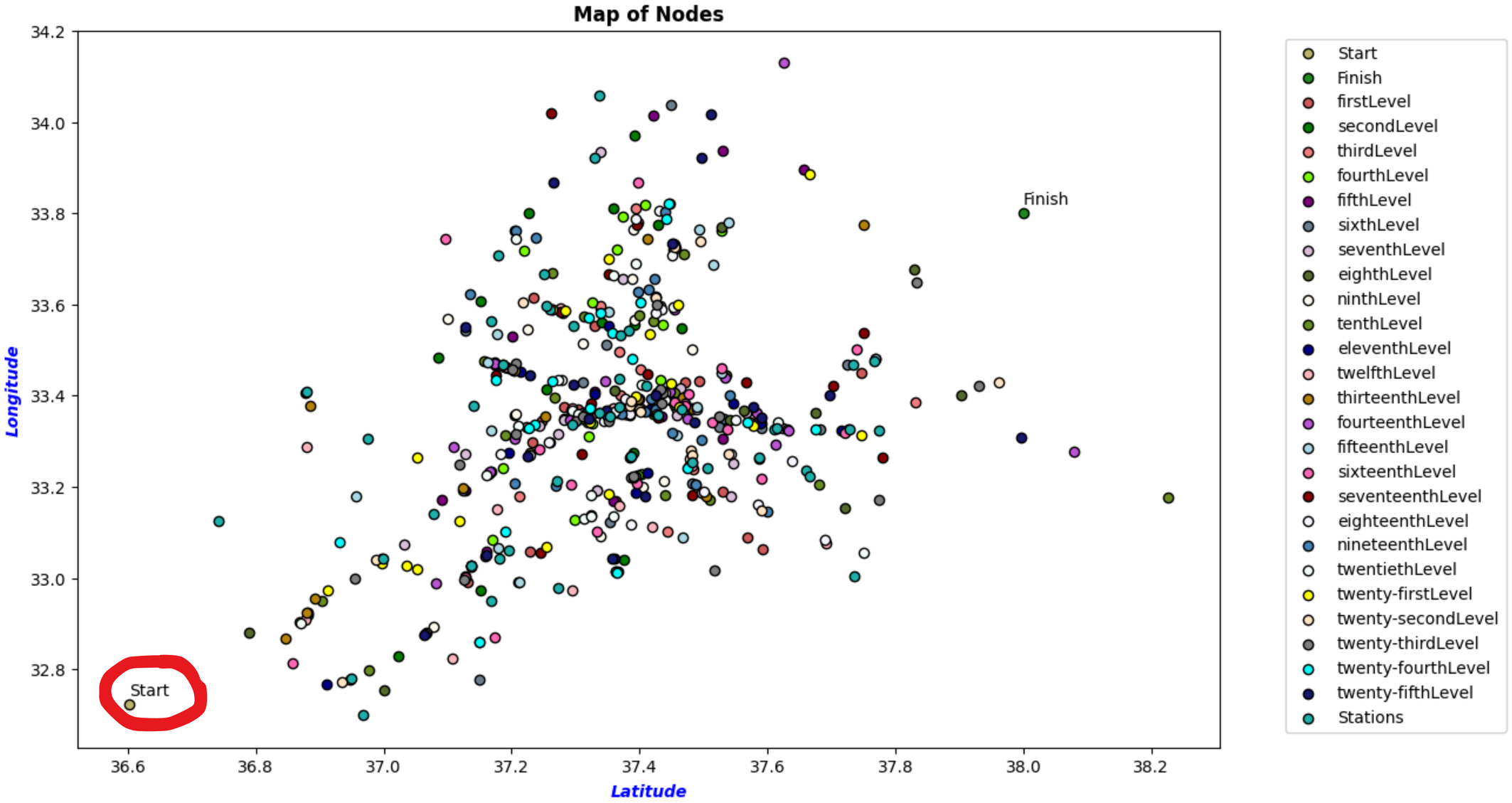
Problem Formulation

PATH



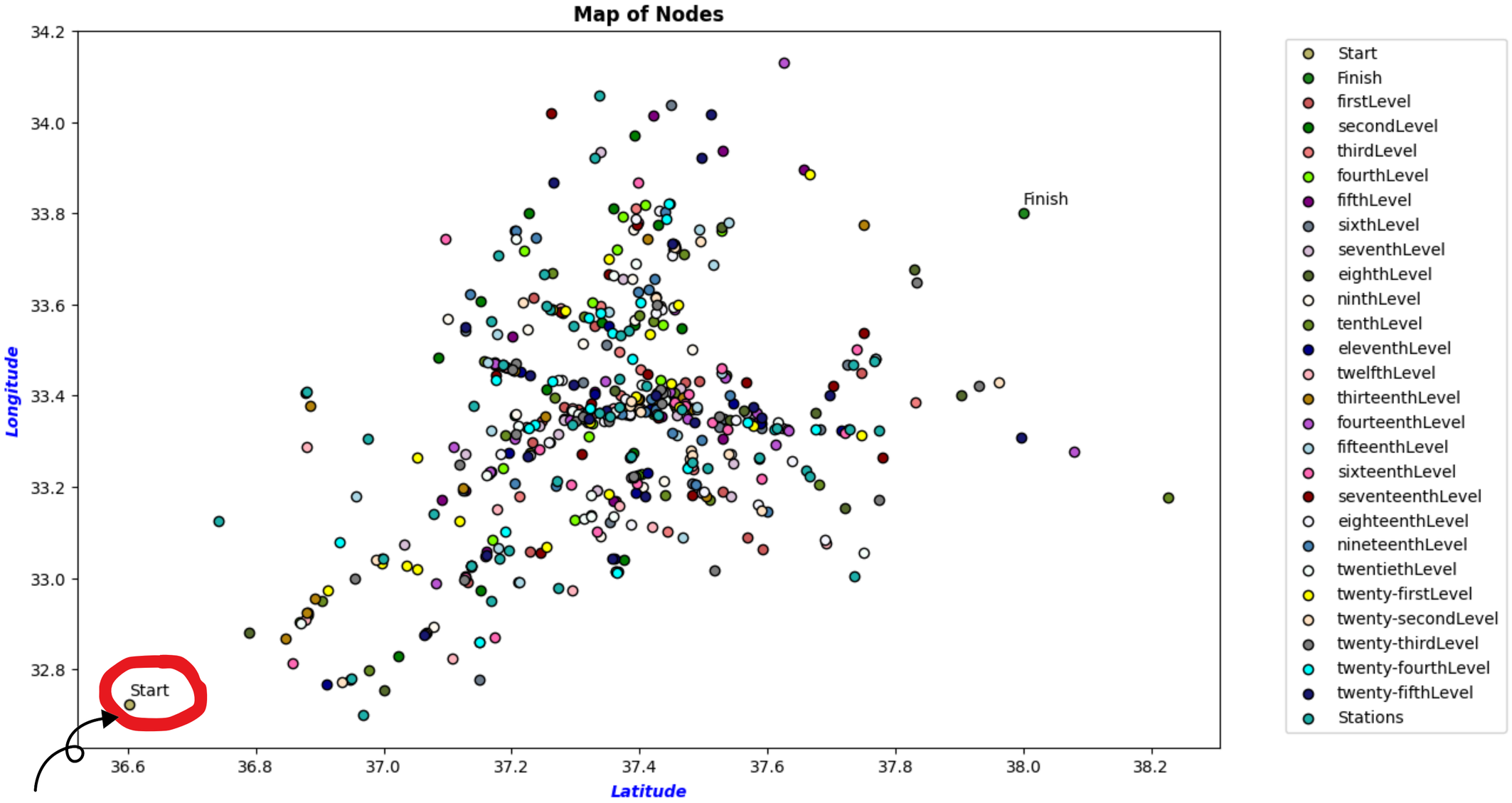
Problem Formulation

PATH



Problem Formulation

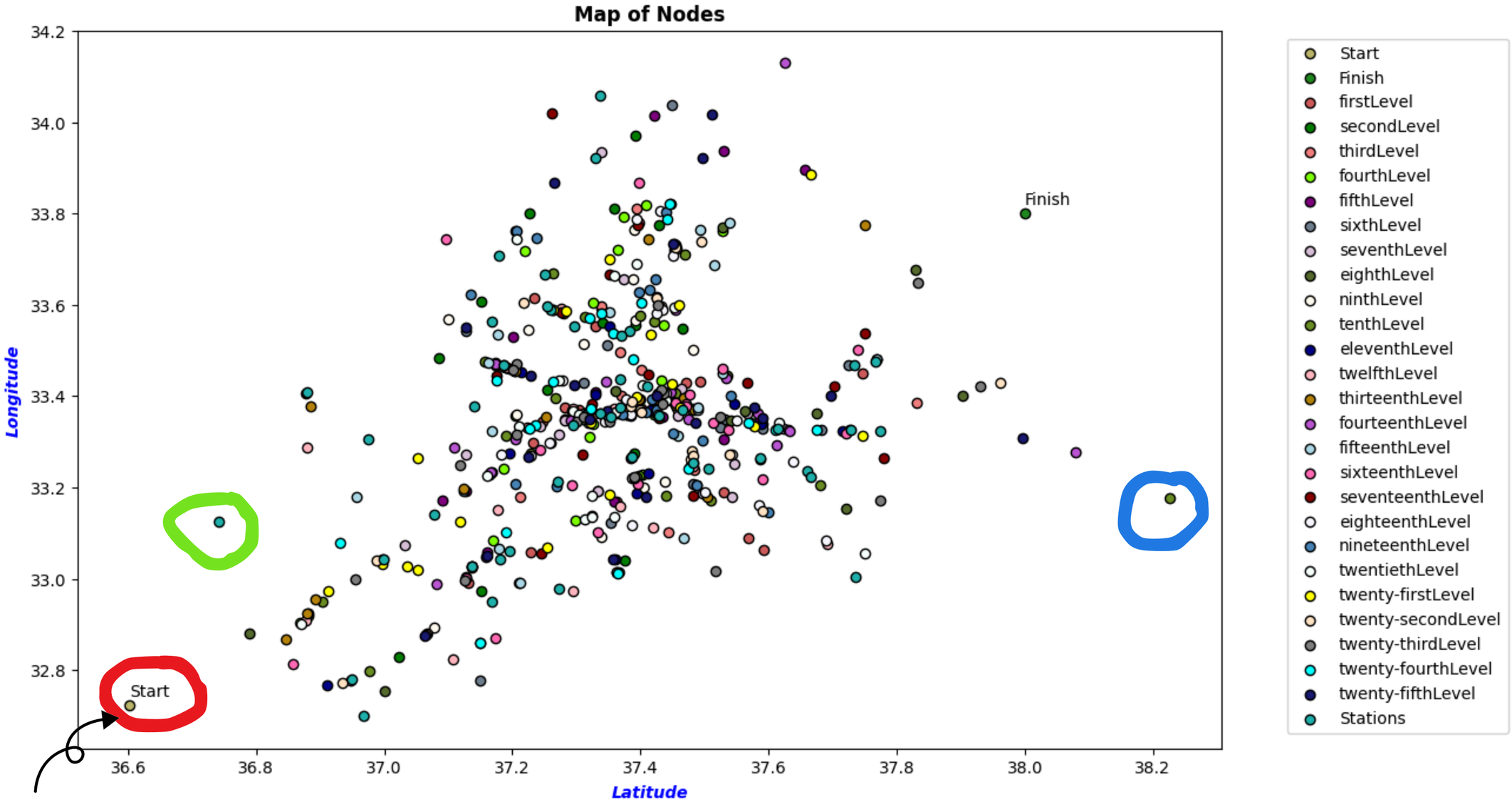
PATH



Initial State

Problem Formulation

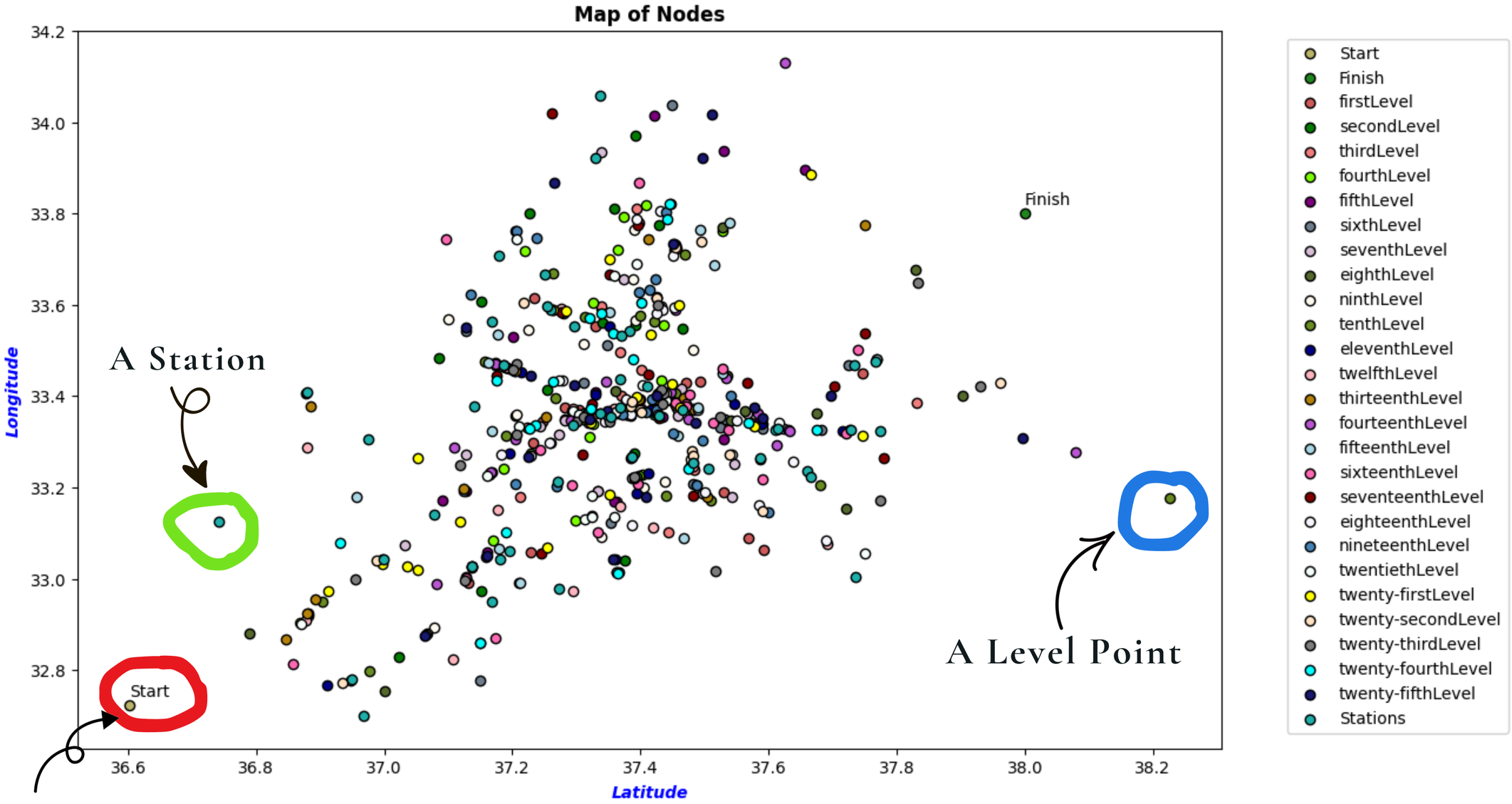
PATH



Initial State

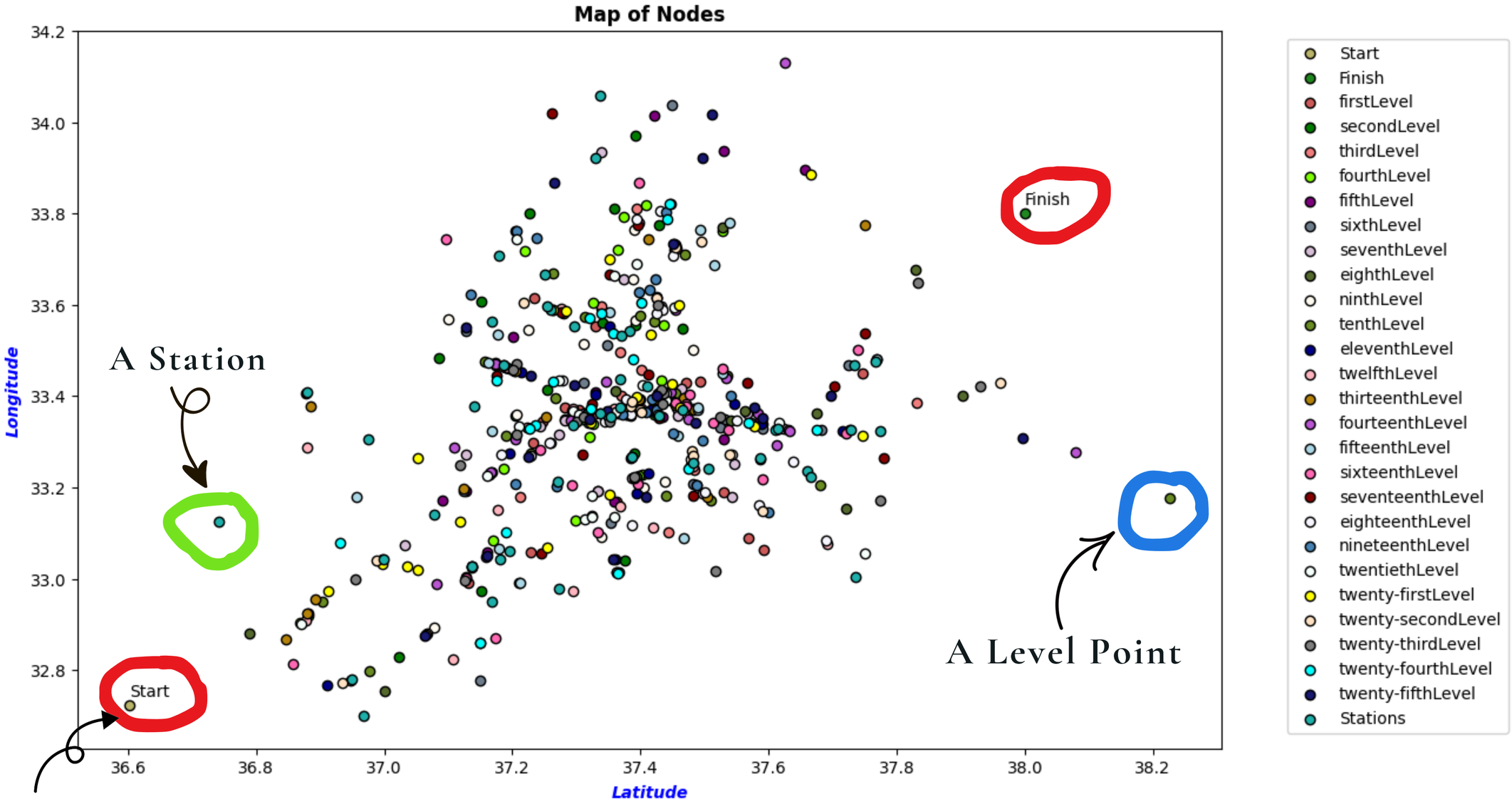
Problem Formulation

PATH



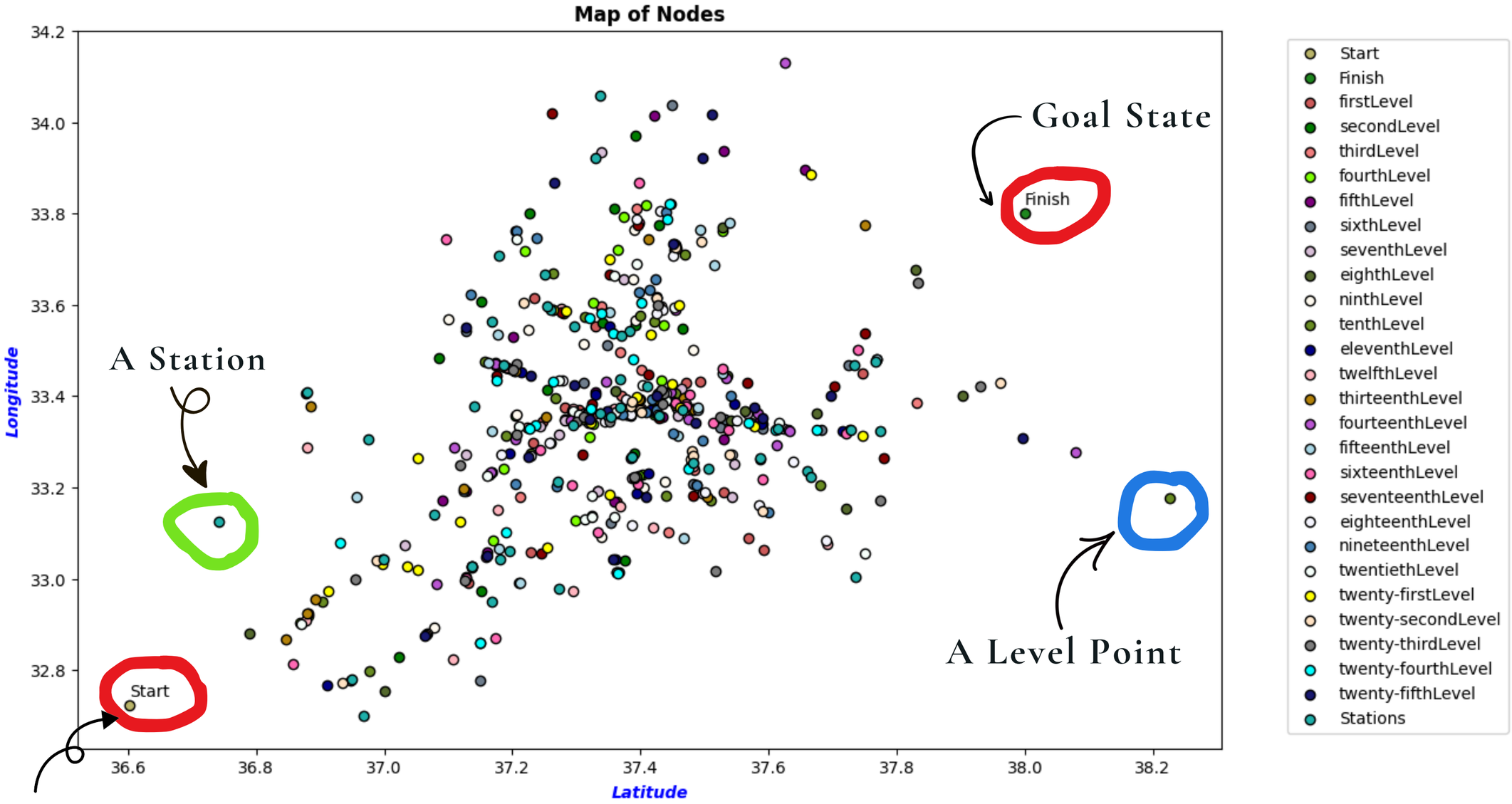
Problem Formulation

PATH



Problem Formulation

PATH



CONSTRAINTS

CONSTRAINTS

CONSTRAINTS

Levels passed: Our agent must pass through **all** levels **sequentially** before it reaches the **finish point.**

CONSTRAINTS

CONSTRAINTS

Levels passed: Our agent must pass through **all** levels **sequentially** before it reaches the **finish point.**

Total distance traveled: If the total distance travel is **larger/no less** than the **fuel constraint**, the path is deemed **unsatisfactory.**

CONSTRAINTS

CONSTRAINTS

Levels passed: Our agent must pass through **all** levels **sequentially** before it reaches the **finish point.**

Total distance traveled: If the total distance traveled is **larger/no less** than the **fuel constraint**, the path is deemed **unsatisfactory.**

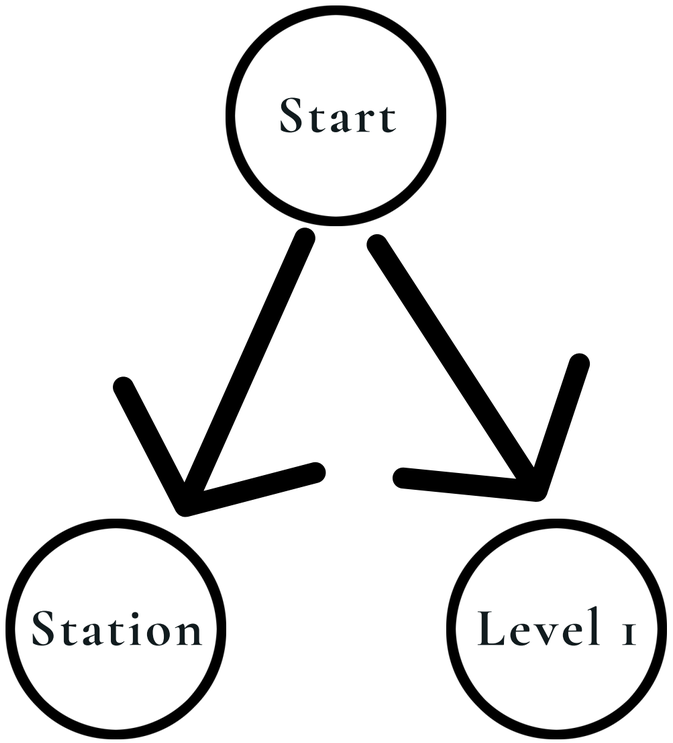
Cost function = Performance measure = total distance traveled: An algorithm is judged by its result's cost function.

CONSTRAINTS

ACTIONS

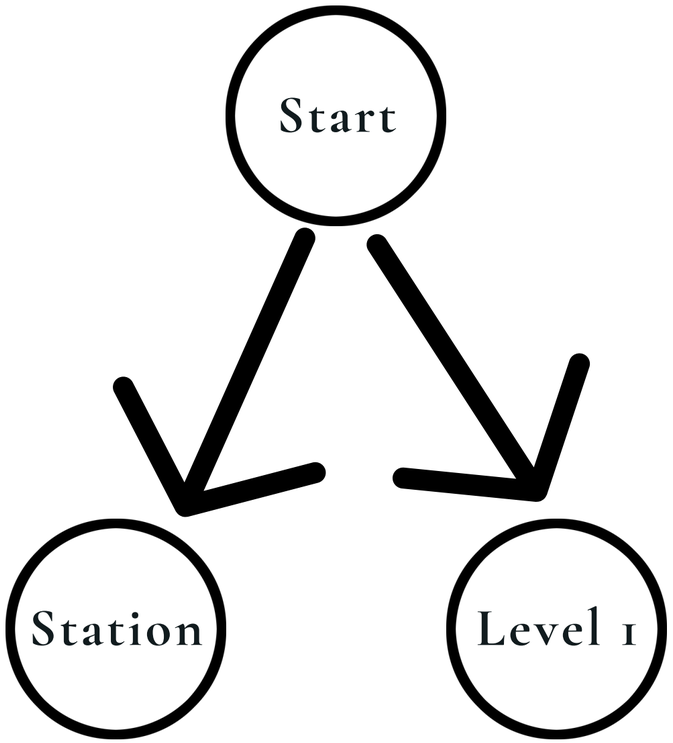
ACTIONS

At the Start

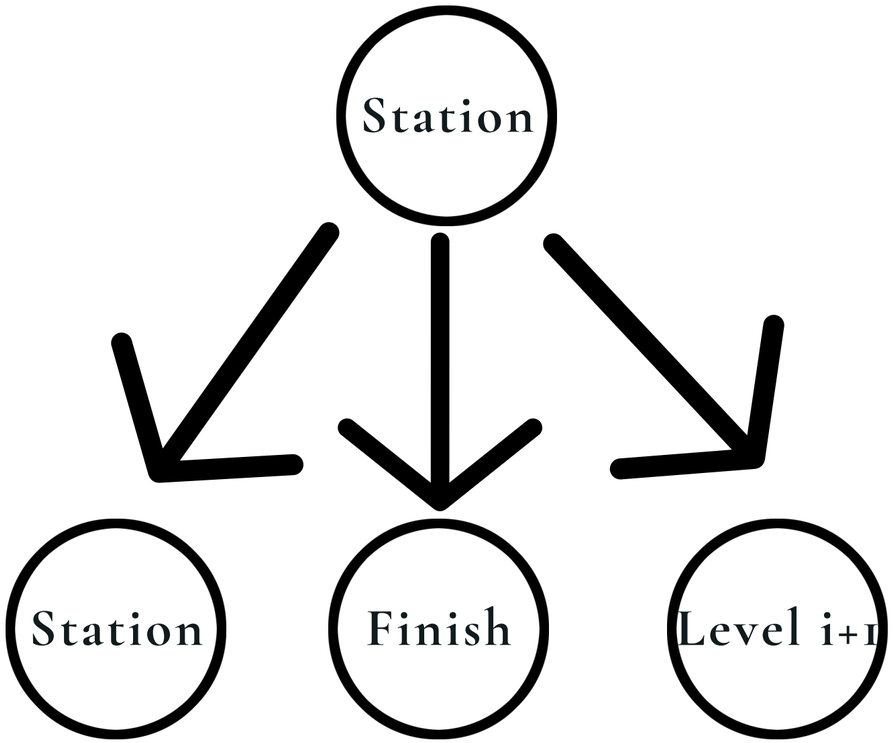


ACTIONS

At the Start

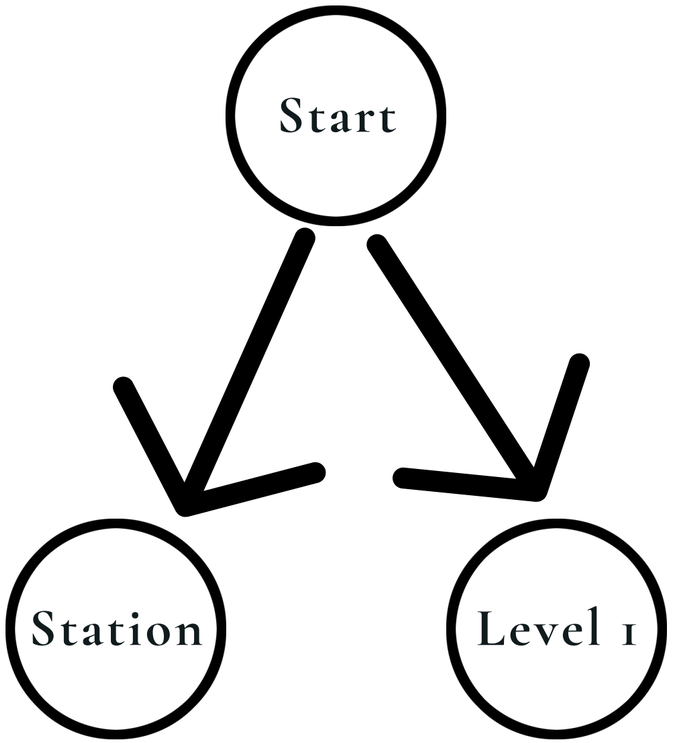


At a Station

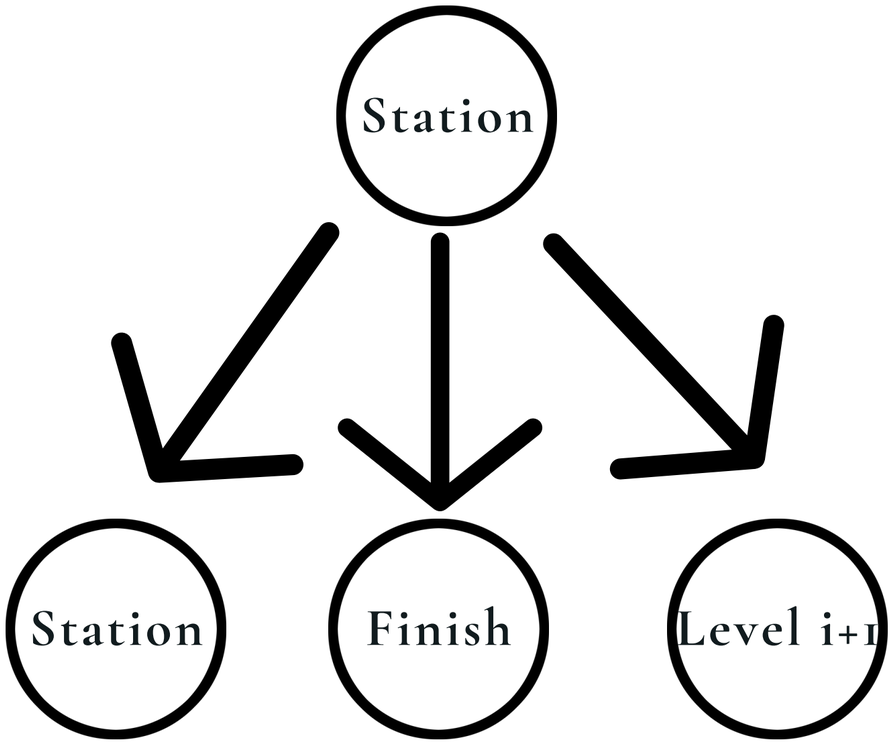


ACTIONS

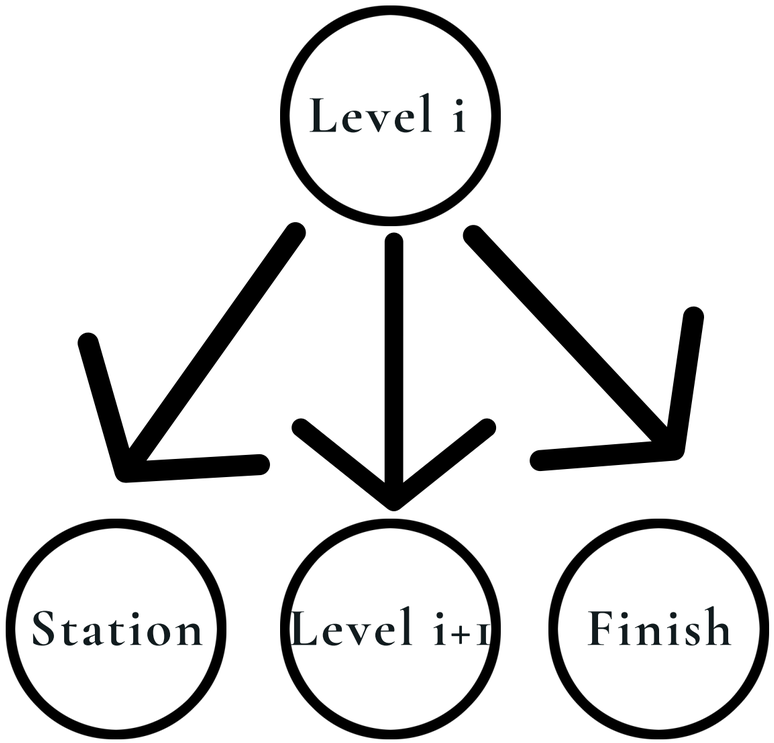
At the Start

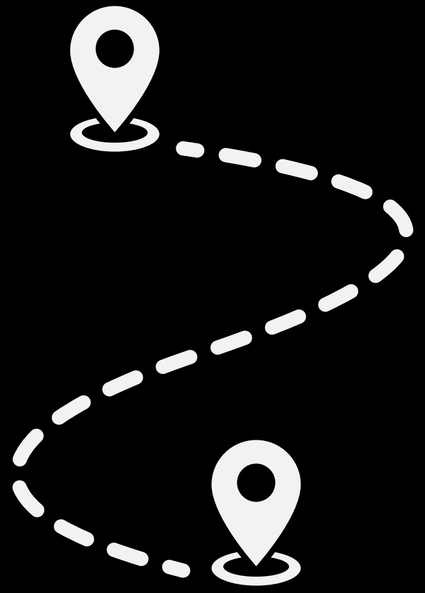


At a Station



At a Level Point





Implement

Design an optimal route with fuel constraints

GROUP 21

ALGORITHMS

Greedy BFS

Greedy Algorithm

- **Definition:** A strategy that makes the locally optimal choice at each step with the hope of finding a global optimum.
- **Application:** Utilized in various optimization problems across different fields.

Best-First Search (BFS) Approach

- **Core Idea:** Traverses a graph by selecting the most promising node based on a specific criterion.
- **Greedy BFS Integration:** Combines the continuous, non-retracing progression of greedy algorithms with the node prioritization of BFS.

GREEDY BEST-FIRST SEARCH

Heuristic

Nearest distance to the next position

Path Finding

At each step, prefer paths leading to the next level or finish line with the **shortest distance.**

h(n) is defined as the heuristic function representing the **shortest distance** from the **current** position (node n) to the nearest node of the **next level** (node n+1) or the **nearest station.**

Greedy BFS

NEAREST DISTANCE TO THE NEXT POSITION

Pseudocode

Algorithm 1 Greedy Algorithm for Route Optimization

```
function GREEDY
    current ← "Start"
    distance ← 0
    path ← [current]
    fuel ← MAX_FUEL_CAPACITY
    while current ≠ "Finish" do
        (next_position, required_fuel) ← DETERMINENEXTPOSITION(current, fuel)
        if required_fuel ≤ fuel then
            current ← next_position
            fuel ← fuel − required_fuel
            distance ← distance + required_fuel
            path.append(current)
        else
            (nearest_station, distance_to_station)                    ←
FINDNEARESTSTATION(current, fuel)
            current ← nearest_station
            fuel ← MAX_FUEL_CAPACITY
            distance ← distance + distance_to_station
            path.append("Refuel at " + current)
        end if
    end while
    return path, distance
end function
```

NEAREST DISTANCE TO THE NEXT POSITION

GrEEDY BEST-FIRST SEARCH

PROS

EFFICIENCY

- Typically faster than methods like BFS due to focused search when combine with Greedy Algorithm.

SIMPLICITY

- Straightforward implementation with clear heuristic guidelines.

GrEEDY BEST-FIRST SEARCH

PROS

EFFICIENCY

- Typically faster than methods like BFS due to focused search when combine with Greedy Algorithm.

SIMPLICITY

- Straightforward implementation with clear heuristic guidelines.

CONS

OPTIMALITY

- Does not guarantee the most efficient path due to local decision-making.

COMPLEXITY IN VARIED TERRAIN

- Performance might vary based on the complexity and layout of the graph.

GREEDY BEST-FIRST SEARCH

Heuristic

Minimal Unvisited Levels

Heuristic function

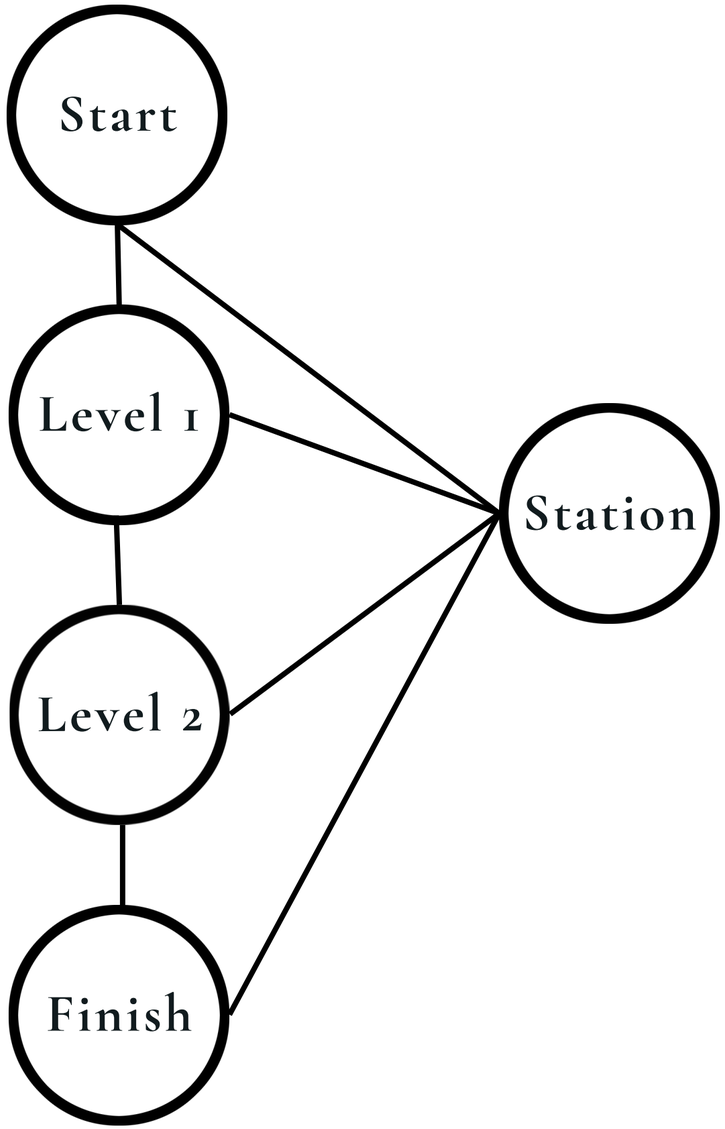
Heuristic function

f(n) = h(n) = the number of **unvisited levels** of a path

Intuition: The algorithm will prioritize expanding on paths having a lower number of unvisited levels.

Example: f([Start, Station]) > f([Start, FirstLevel])

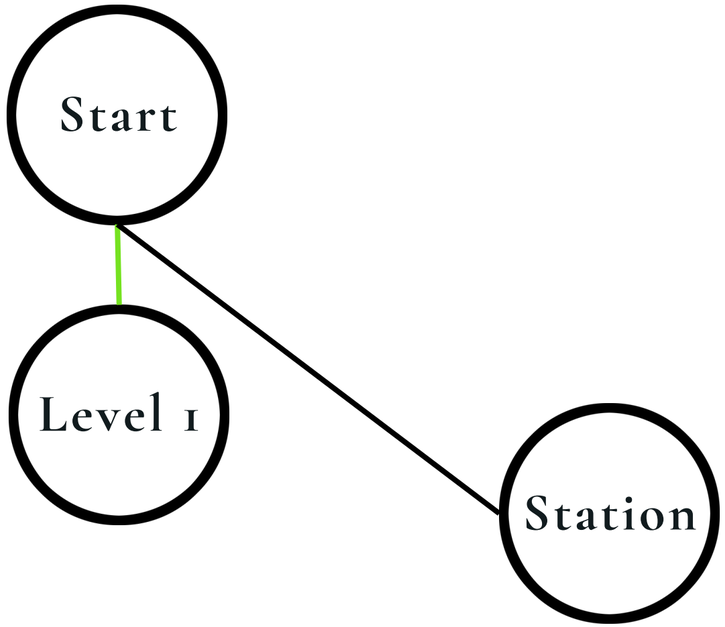
The algorithm



Pseudo Code

```
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
11:            else: path.remove(Finish)
12:        for neighbour in neighbours(node):
13:            newPath = path.copy( )
14:            newPath.append(neighbour)
15:            if FuelConstraint(newPath):
16:                queue.append(newPath)
17:        queue.sort(key = Heuristic)
```

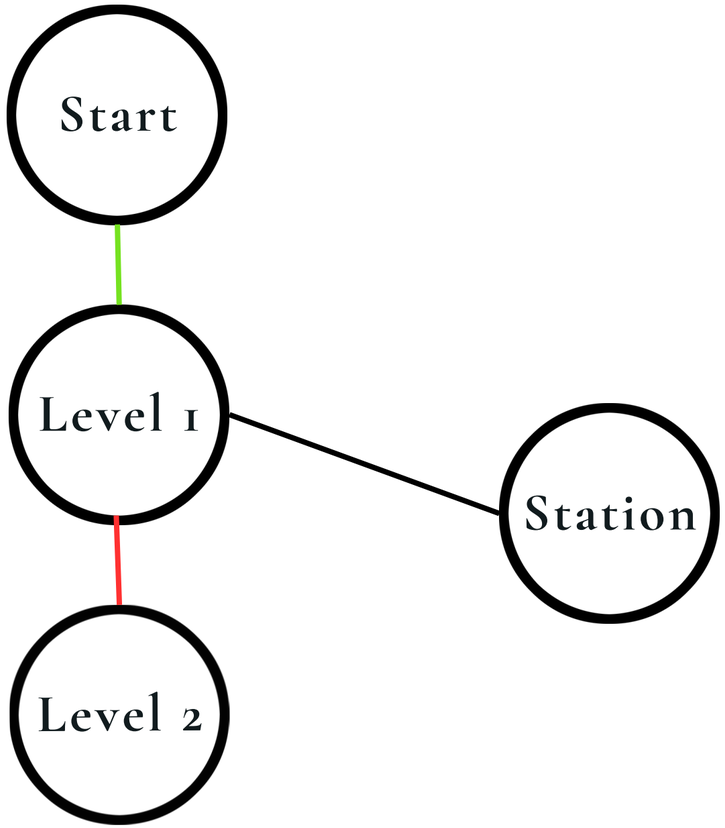
The algorithm



Pseudo Code

```
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
11:            else: path.remove(Finish)
12:        for neighbour in neighbours(node):
13:            newPath = path.copy( )
14:            newPath.append(neighbour)
15:            if FuelConstraint(newPath):
16:                queue.append(newPath)
17:        queue.sort(key = Heuristic)
```

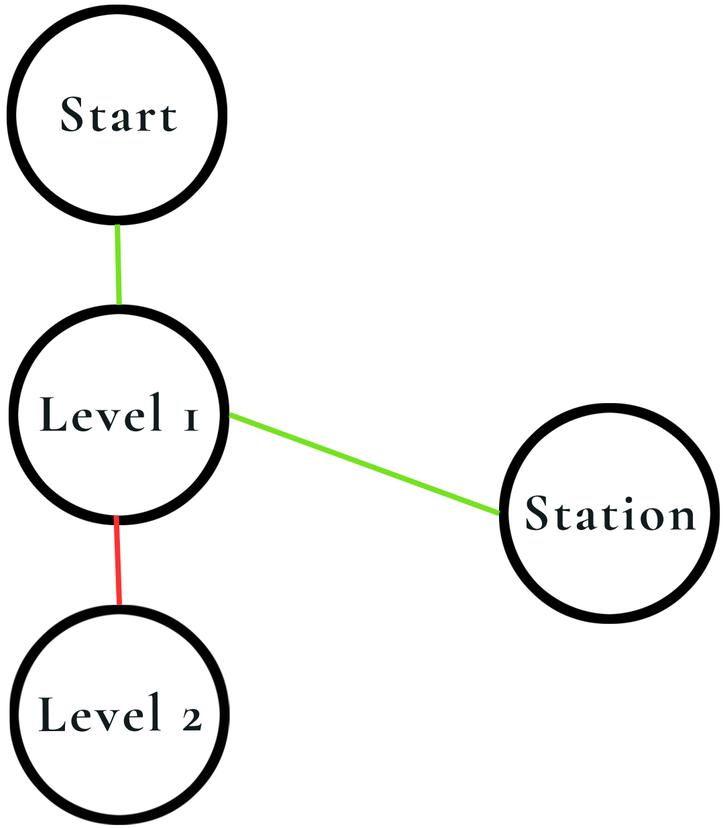
The algorithm



Pseudo Code

```
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
11:            else: path.remove(Finish)
12:        for neighbour in neighbours(node):
13:            newPath = path.copy( )
14:            newPath.append(neighbour)
15:            if FuelConstraint(newPath):
16:                queue.append(newPath)
17:        queue.sort(key = Heuristic)
```

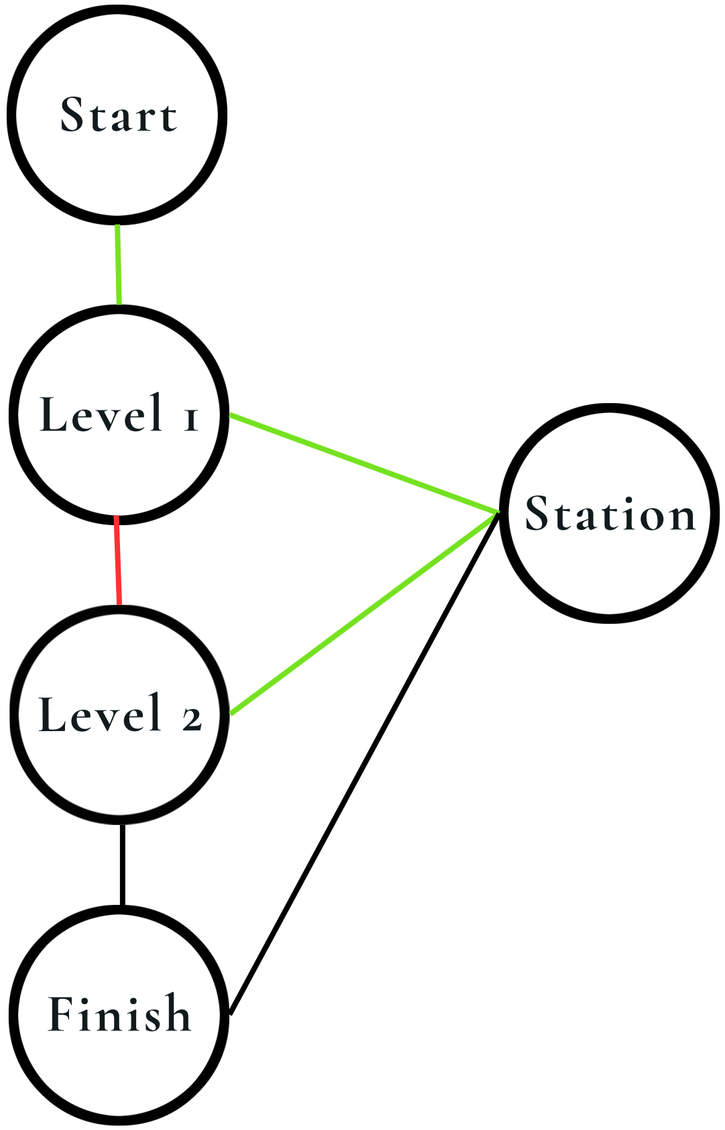
The algorithm



Pseudo Code

```
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
11:            else: path.remove(Finish)
12:        for neighbour in neighbours(node):
13:            newPath = path.copy( )
14:            newPath.append(neighbour)
15:            if FuelConstraint(newPath):
16:                queue.append(newPath)
17:        queue.sort(key = Heuristic)
```

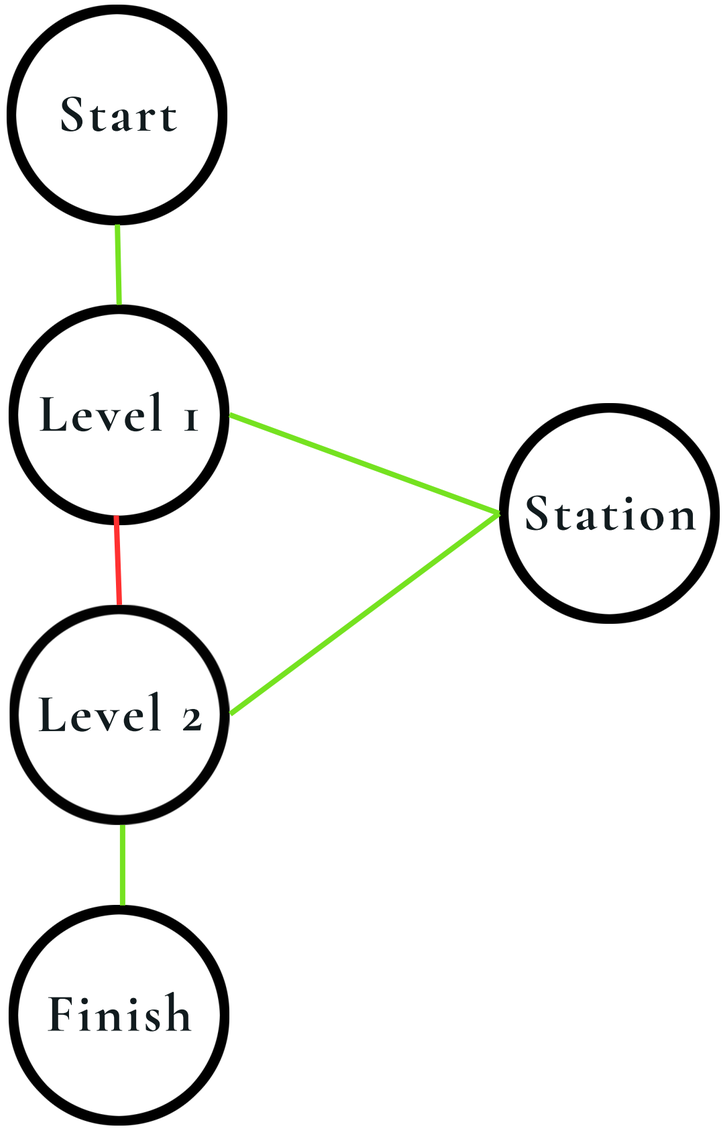
The algorithm



Pseudo Code

```
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
11:            else: path.remove(Finish)
12:        for neighbour in neighbours(node):
13:            newPath = path.copy( )
14:            newPath.append(neighbour)
15:            if FuelConstraint(newPath):
16:                queue.append(newPath)
17:        queue.sort(key = Heuristic)
```

The algorithm



Pseudo Code

```
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
11:            else: path.remove(Finish)
12:        for neighbour in neighbours(node):
13:            newPath = path.copy( )
14:            newPath.append(neighbour)
15:            if FuelConstraint(newPath):
16:                queue.append(newPath)
17:        queue.sort(key = Heuristic)
```

Greedy Breadth-First Search

CONS

TIME COMPLEXITY

Greedy Breadth-First Search

CONS

TIME COMPLEXITY

- How fast the algorithm depends on the **order in which each neighbors** of the nodes expanded are considered.

CONS

TIME COMPLEXITY

- How fast the algorithm depends on the **order in which each neighbors** of the nodes expanded are considered.

OPTIMIZATION

Greedy Breadth-First Search

CONS

TIME COMPLEXITY

- How fast the algorithm depends on the **order in which each neighbors** of the nodes expanded are considered.

OPTIMIZATION

- Since the heuristic function completely ignores the distance traveled. The result isn’t always optimal.

PROS

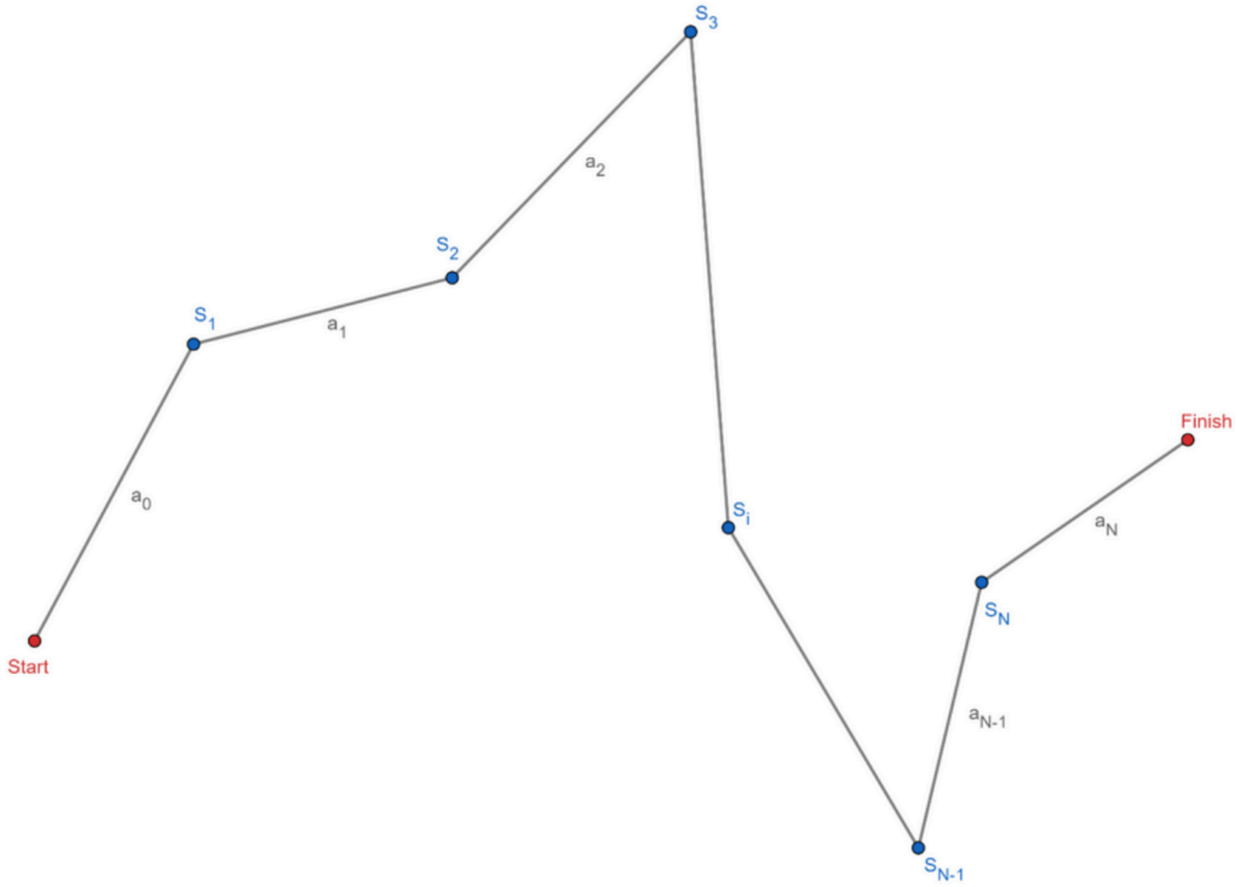
INPUT FOR LOCAL SEARCH ALGORITHMS

PROS

INPUT FOR LOCAL SEARCH ALGORITHMS

- In reality, the **ideal solution path** would allow the agent to travel in the **shortest distance** while **saving time of refueling**. As it shouldn't stop by a station when it can still reach another level and have enough fuel to get to another station.
- The result produced by this algorithm and the ideal solution shares **a property**.
- We may use the **local search algorithms** on this result to find the optimal path.

PROPERTY OF THE IDEAL SOLUTION



Formulation

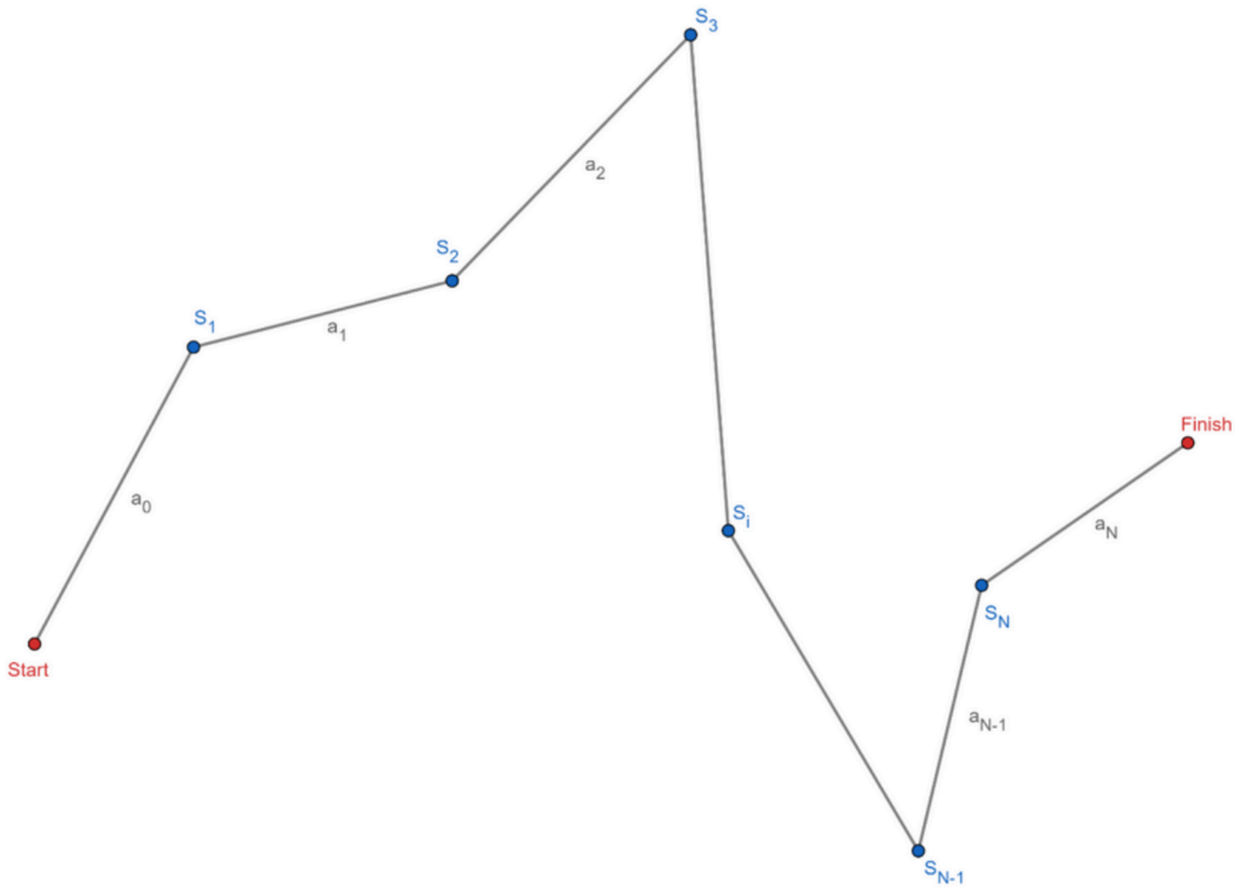
- Let **L** be the fuel constraint
- **i** is numbered from 1 to N, **S(i)** are stations, **a(i)** are the Haversine distances shown in the figure.

Then:

$$0 < a(j) <= L < a(i) + a(i+1)$$

where **i = 0, ..., N-1** and **j = 0, ..., N**

PROPERTY OF THE IDEAL SOLUTION



$$0 < a(j) <= L < a(i) + a(i+1)$$

where **i = 0, …, N-1** and **j = 0, …, N**

- Let **N, M** be the **number of stations** in the **ideal** and **Greedy BFS** solutions.
- Using this property, we can show that:

$$0 <= N <= 2M+1$$

- If we can find a **smaller upper bound** for **N** and **a solution with a smaller number of stations visited**, then the local search algorithms could then be applied.

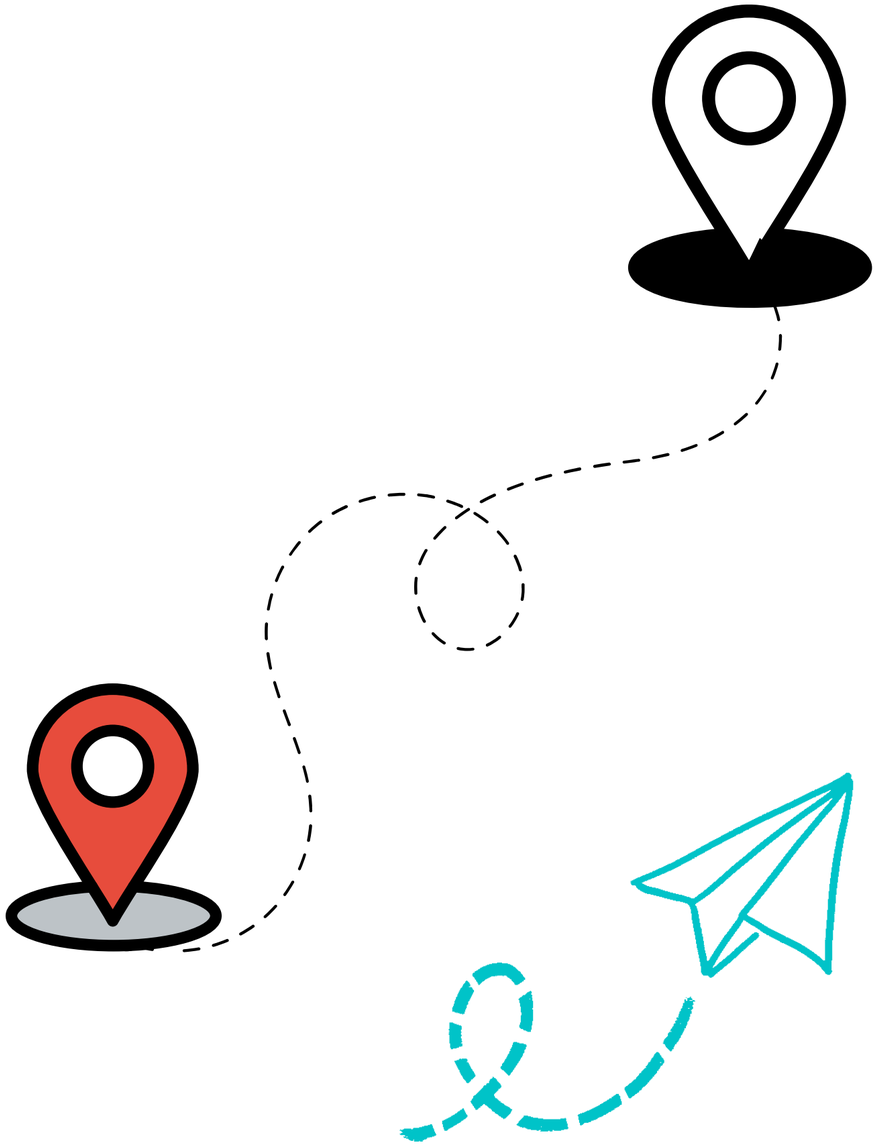
A* algorithm: 1-step look-ahead heuristic

1. INTRODUCTION

- The A* algorithm is a popular and widely used pathfinding algorithm in computer science, particularly in the field of artificial intelligence and robotics.

- A* is designed to find the shortest path from a starting point to a goal point in a weighted graph or grid.

- The algorithm is informed and efficient, combining elements of both Dijkstra's algorithm and Greedy Best-First Search.



2. Evaluation function

2. EVALUATION FUNCTION

- The evaluation function:

$$f(n) = g(n) + h(n)$$

g(n) = The cost from the root node to (the current one) n
h(n) = The estimated cost from n to goal
f(n) = The estimated total cost of path through n to goal

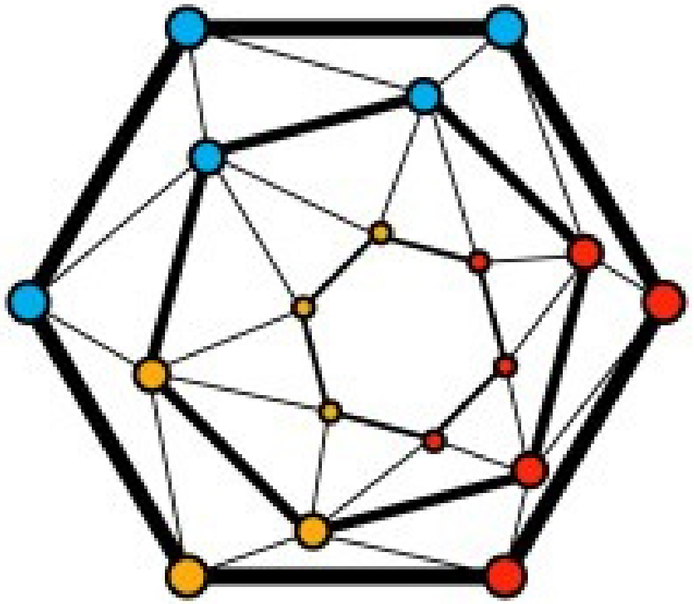
2. EVALUATION FUNCTION

- Heuristic function h(n):
  - Admissible heuristic:
    Never overestimates the cost to reach the goal
  - Consistent heuristic:

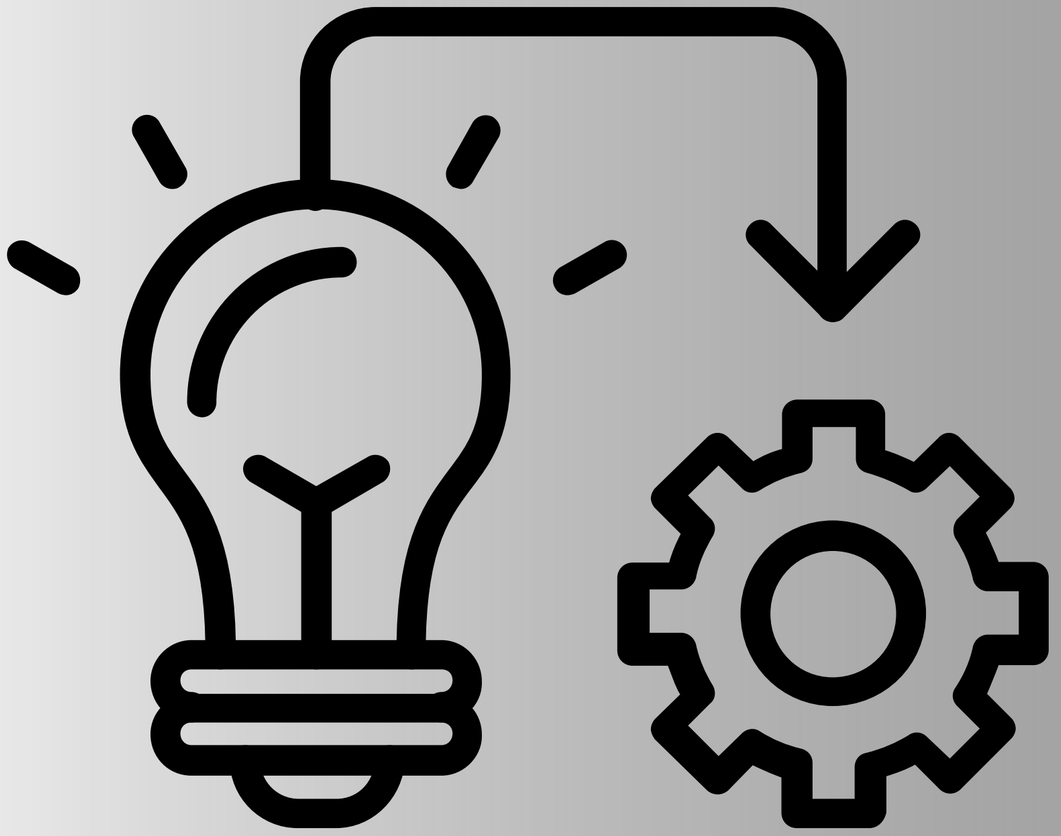
$$h(n) \leq c(n,s) + h(s)$$

for ***s*** is a successor of node ***n,***

c(n,s) is the actual cost



→ **Euclidean Distance** is an admissible heuristic estimator



3. Implementation

3. IMPLEMENTATION

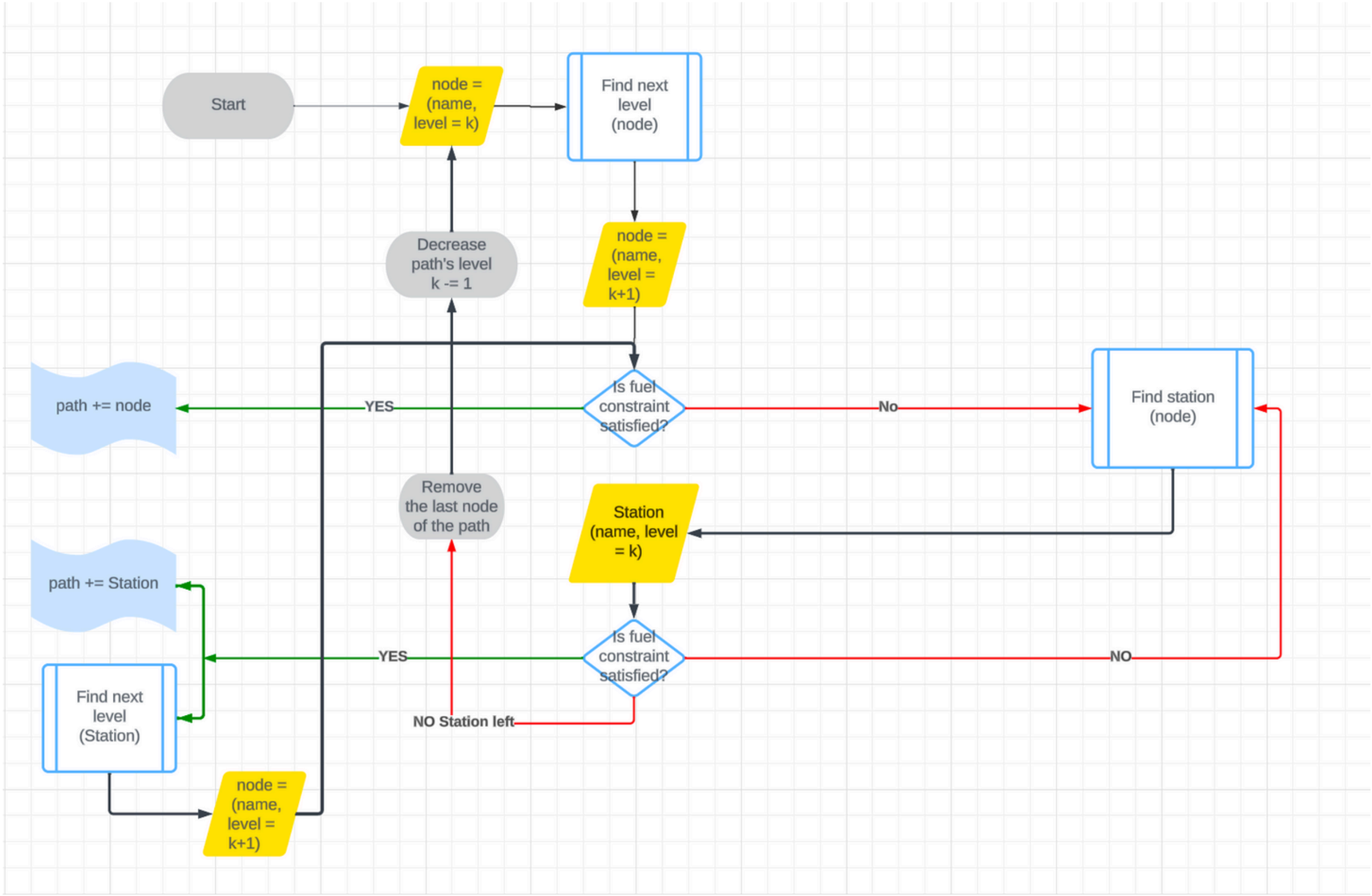
- Apply the concepts of backtracking, dynamic programming, and recursion.

- 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 “Level” node ***(PointPath)***
  - starting from a “Station” node ***(StationPath)***

3. IMPLEMENTATION

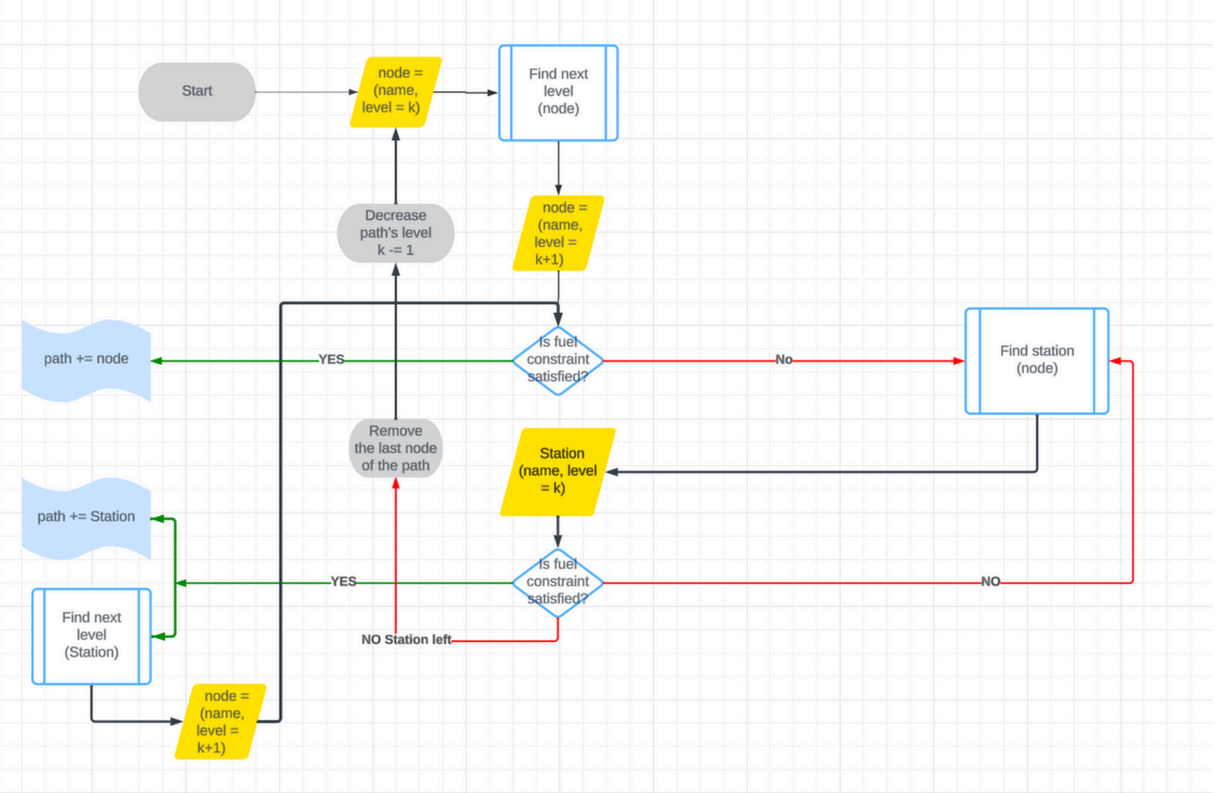
A* r-step Algorithm



PointPath

- Iterate through nodes at the current level, selecting the node with the smallest f(n) value.

- Check if there's enough fuel to reach that node;
  - if yes, add it to the path.
  - If not, search for the station with the smallest f(n) value and sufficient fuel.

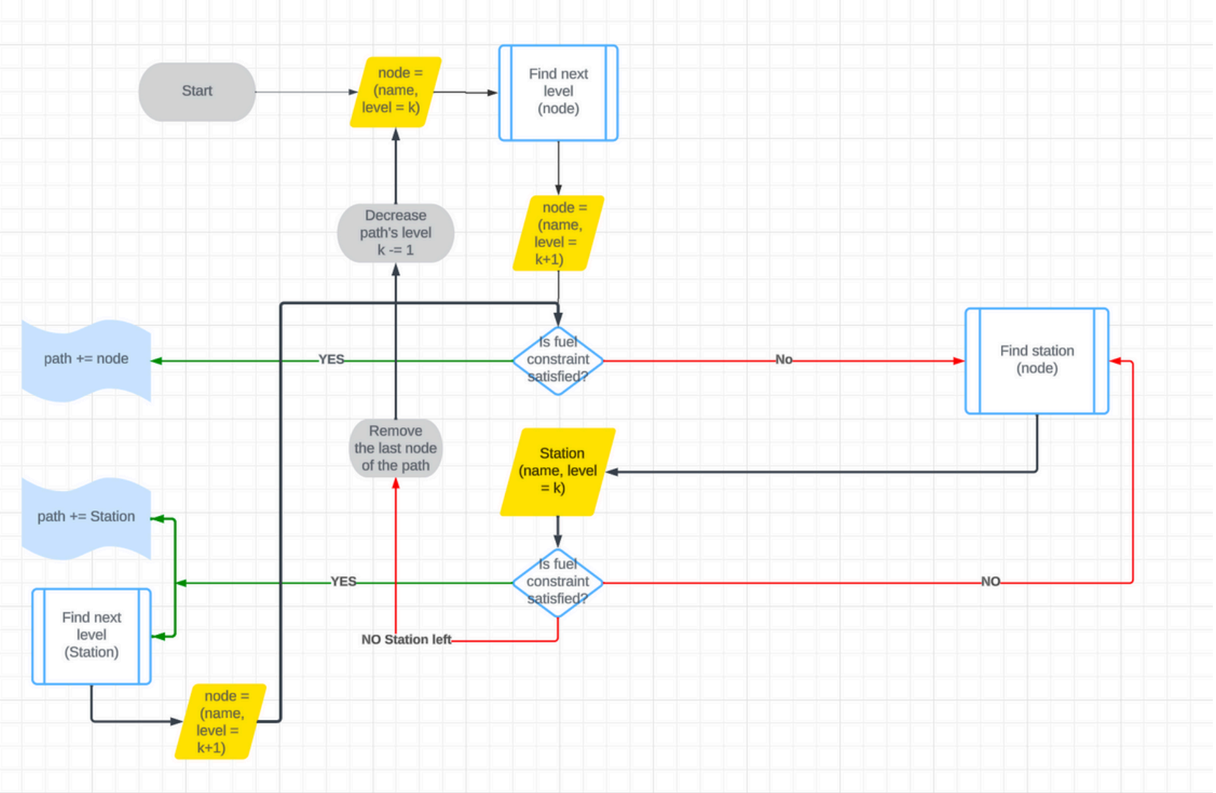


A* 1-step Algorithm

PointPath

- If there is no suitable station, we backtrack, searching for a node at the previous level different from the last node in the path, removing the last node.

- This is akin to PointPath with the level reduced by 1, excluding the recently removed node.



PointPath

Pseudo Code

Algorithm 1 PointPath Function

Input:

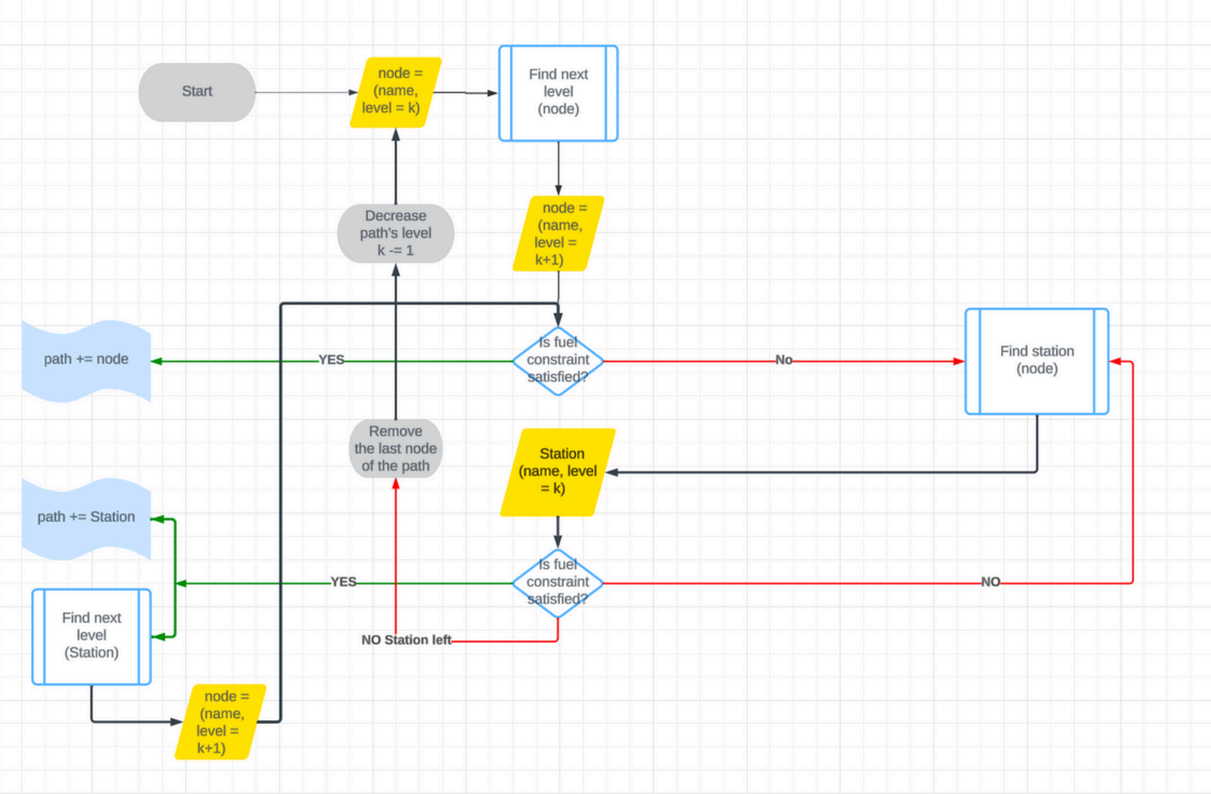
1. path, state(distance move, fuel level), level
2. G1(V1,E1): graph between 2 levels' nodes
3. Gs(Vs,Es): graph between stations and all nodes

```
1: function PointPath
2:     curr_level ← state.level
3:     priorityQueue ← priority queue of unvisited nodes sorted by
   2 keys: level and f value
4:     while priorityQueue.isEmpty() is not True do
5:         current ← priorityQueue.get()
6:         traverse_level ← current node's level
7:         if traverse_level ≥ curr_level then
8:             pass
9:         else
10:            Remove the last node from the path
11:            state ← Recover previous level's state
12:            curr_level ← traverse_level
13:        end if
14:        if Fuel constraint is violated then
15:            priorityQueue_Station ← priority queue of valid
   stations sorted by f value
16:            if priorityQueue_Station.empty() then
17:                continue
18:            else
19:                station ← priorityQueue_Station.get()
20:                Add station to the path
21:                state ←Update new state
22:                Empty priorityQueue_Station
23:                Find next level's node via STATIONPATH function
24:            end if
25:        else
26:            Add current to the path
27:            state ←Update new state
28:        end if
29:        if curr_level ≥ level then
30:            return path, state
31:        else
32:            for i in range(level - curr_level − 1, −1, −1) do
33:                path, state ← PointPath
34:            end for
35:            return path, state
36:        end if
37:    end while
38: end function
```

A* 1-step Algorithm

StationPath

- Similar to PointPath but allowing for multiple stations in the path.
- Despite the posssibility of multiple stations, the algorithm won't loop infinitely.
- At each level, there exists at least one pair of nodes labeled as a station and a node labeled with a level, meets the fuel condition.
- Consequently, there will always be a path from this level to a station and from a station to another level.



A* 1-step Algorithm

StationPath

Pseudo Code

Algorithm 2 StationPath Algorithm

Input:

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

```
1: function StationPath
2:     pQueue ← priority queue of unvisited nodes at current level sorted by f value
3:     while pQueue.empty() is not True do
4:         current ← pQueue.get()
5:         if Fuel constraint is violated then
6:             priorityQueue_Station ← priority queue of valid stations sorted by h value
7:             if priorityQueue_Station.empty() then
8:                 continue
9:             else
10:                station ← priorityQueue_Station.get()
11:                Add station to the path
12:                state ←Update new state
13:                Empty priorityQueue_Station
14:                Find next level's node via STATIONPATH function
15:            end if
16:        else
17:            Add current to the path
18:            state ←Update new state
19:        end if
20:
```

A* 1-step Algorithm

A* algorithm: 2-step look-ahead heuristic

INTUITIVE IDEA

- Estimate the cost to the destination from the unvisited adjacent node of the current node

- Popular heuristic, Euclidean distance or 1-step look-ahead heuristic:
  - $\mathbf{h(n1) = c(s, n1) + dist(n1, e).}$

  $\mathbf{c(s, n1)}$ is actual cost
  $\mathbf{dist(n1, e)}$ is the Euclidean distance to travel from n1 to e.

- 2-step look-ahead heuristic:
  - $\mathbf{h(n1) = c(s, n1) + min(dist(n1, n2) + dist(n2, e)}$

  for all n2 in N2.)

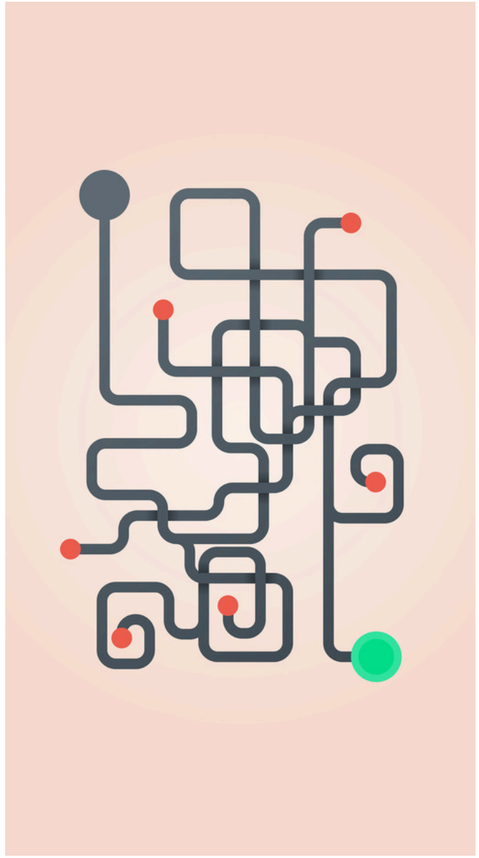
Denotes that N1 is a list that contains all adjacent nodes to node s, N2 contains all unvisited adjacent nodes of all nodes appear in N1

INTUITIVE IDEA

- This 2-step look-ahead heuristic is admissible and consistent, so it is suitable for shortest path finding.

- This is an efficient heuristic because it estimates the cost to the destination more accurately and closely to the actual cost compare to euclidean distance heuristic.

- Our problem divided into levels and cannot pass from current level to previous level, so this heuristic is very appropriate and easy to implement.



PSEUDO CODE

Algorithm 3 2-step Look-Ahead Heuristic

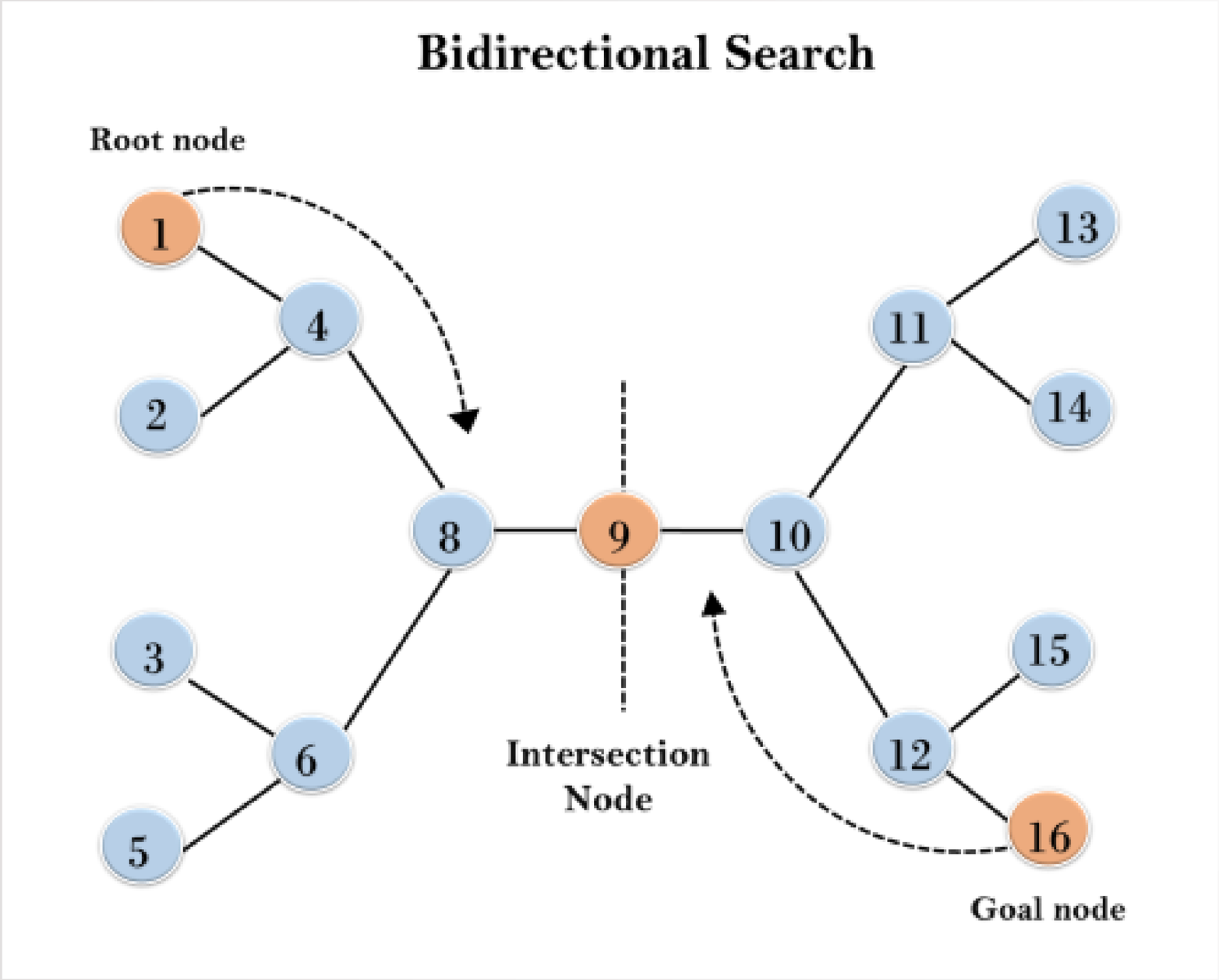
2-step Look-ahead(G, n, t):

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 ∞
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)+dist(n_2, t) to possibleVals
15:            Remove n_1 from alreadySeen
16:    h(n) := min(possibleVals)
17: return h(n)
```

ALGORITHM

Bidirectional A* Algorithm


Bidirectional Search
Root node
1
2
3
4
5
6
8
9
10
11
12
13
14
15
16
Intersection Node
Goal node

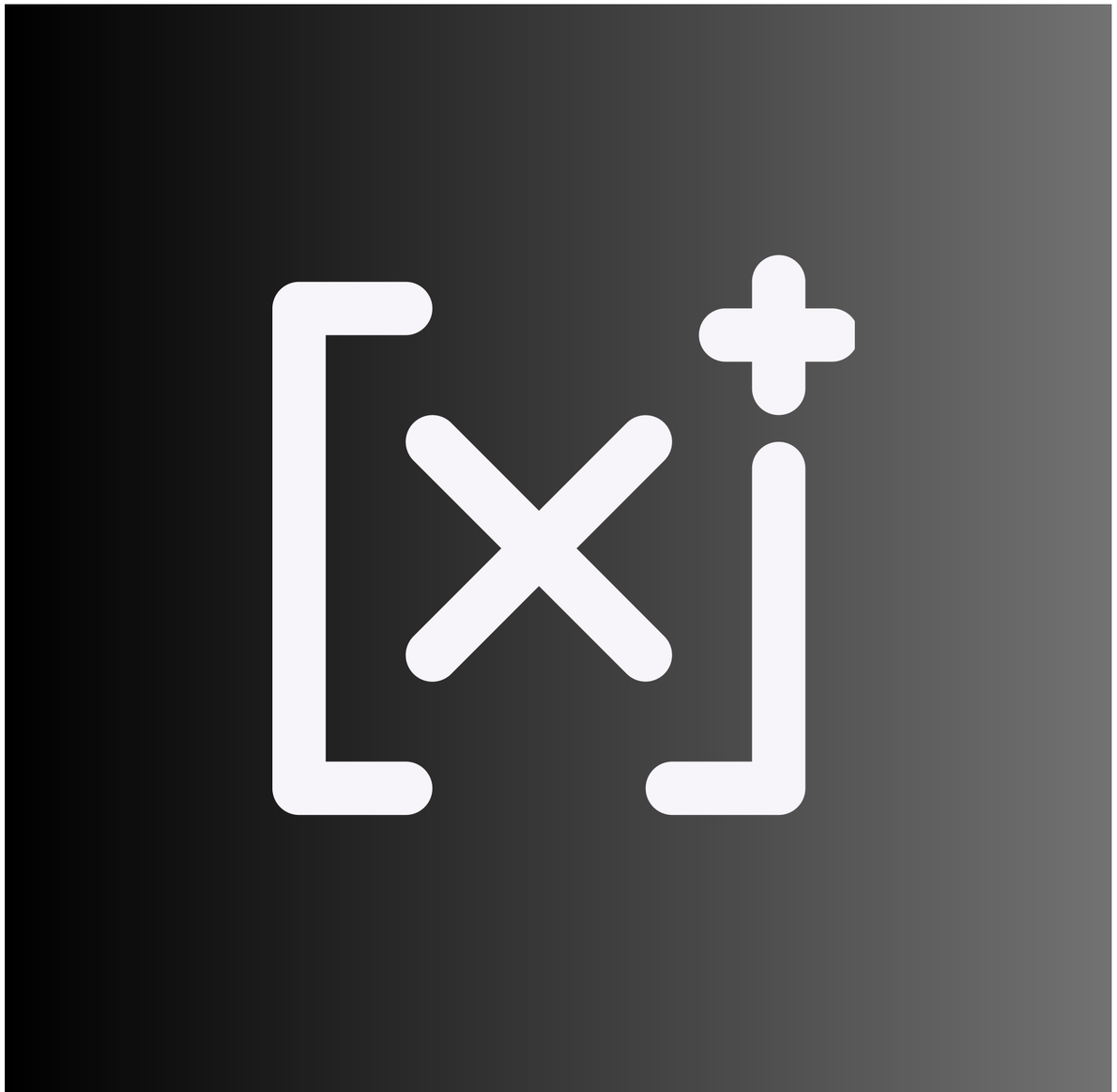
EXAMPLES

DEFINE

Main idea

- Perform two searches: one from the initial state towards the goal state (**forward search**); one from the goal state towards the initial state (**backward search**)
- The searches meet in the middle when the frontiers of the two searches intersect.

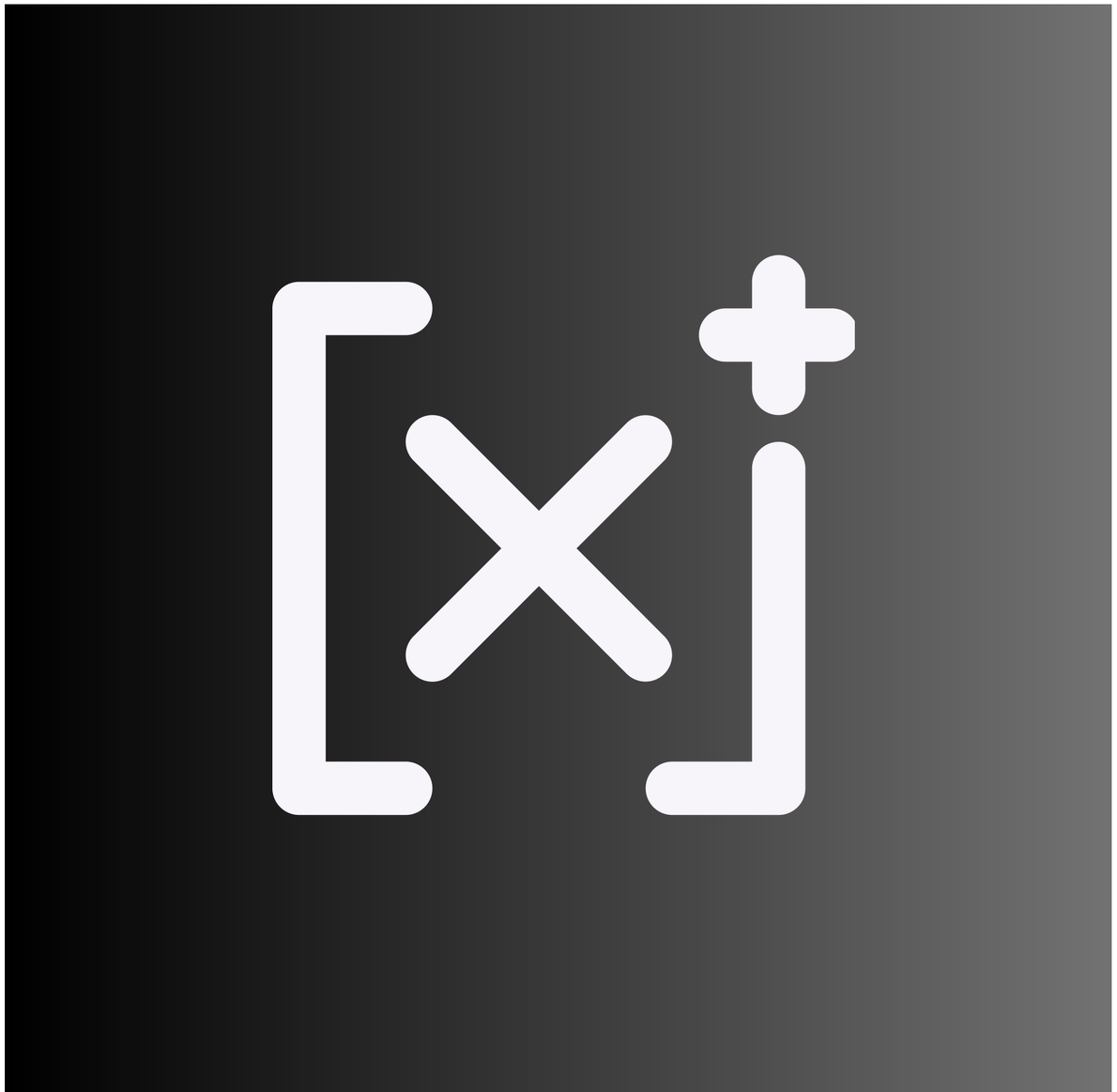
IMPLEMENTATION



PHASE 1

- Alternating between expanding nodes in the forward and backward directions until **a meeting node is found**
- For each direction, popping the node with the **lowest heuristics value** from the queue, then check for meeting points

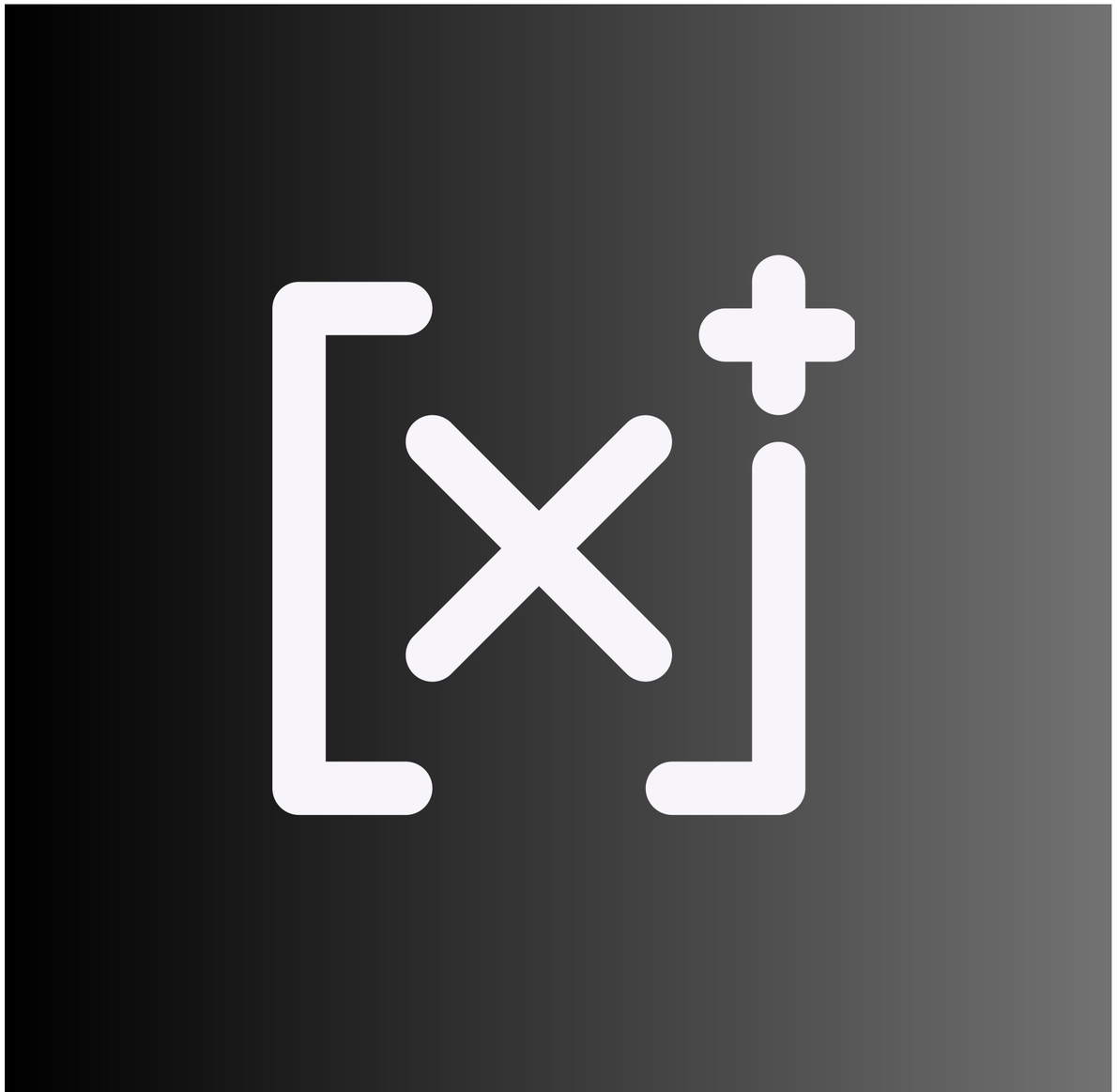
IMPLEMENTATION



PHASE 1

- For the forward direction, the heuristic value equals to the sum of the total distance travelled from the starting node and the straight-line distance from the considered node to the finishing node

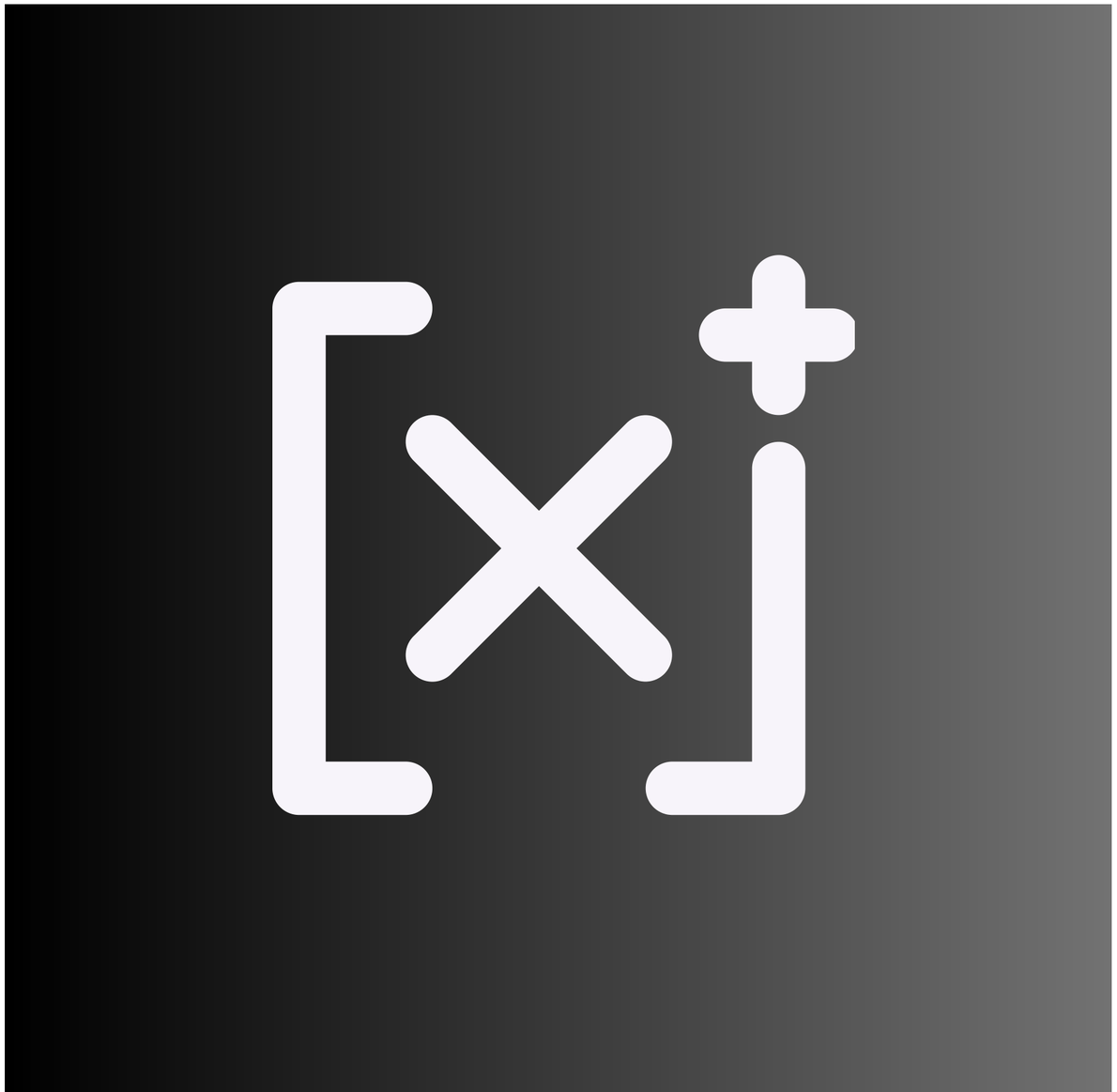
IMPLEMENTATION



PHASE 1

- For the backward direction, the heuristic value equals to the sum of the total distance travelled from the finishing node and the straight-line distance from the considered node to the starting node

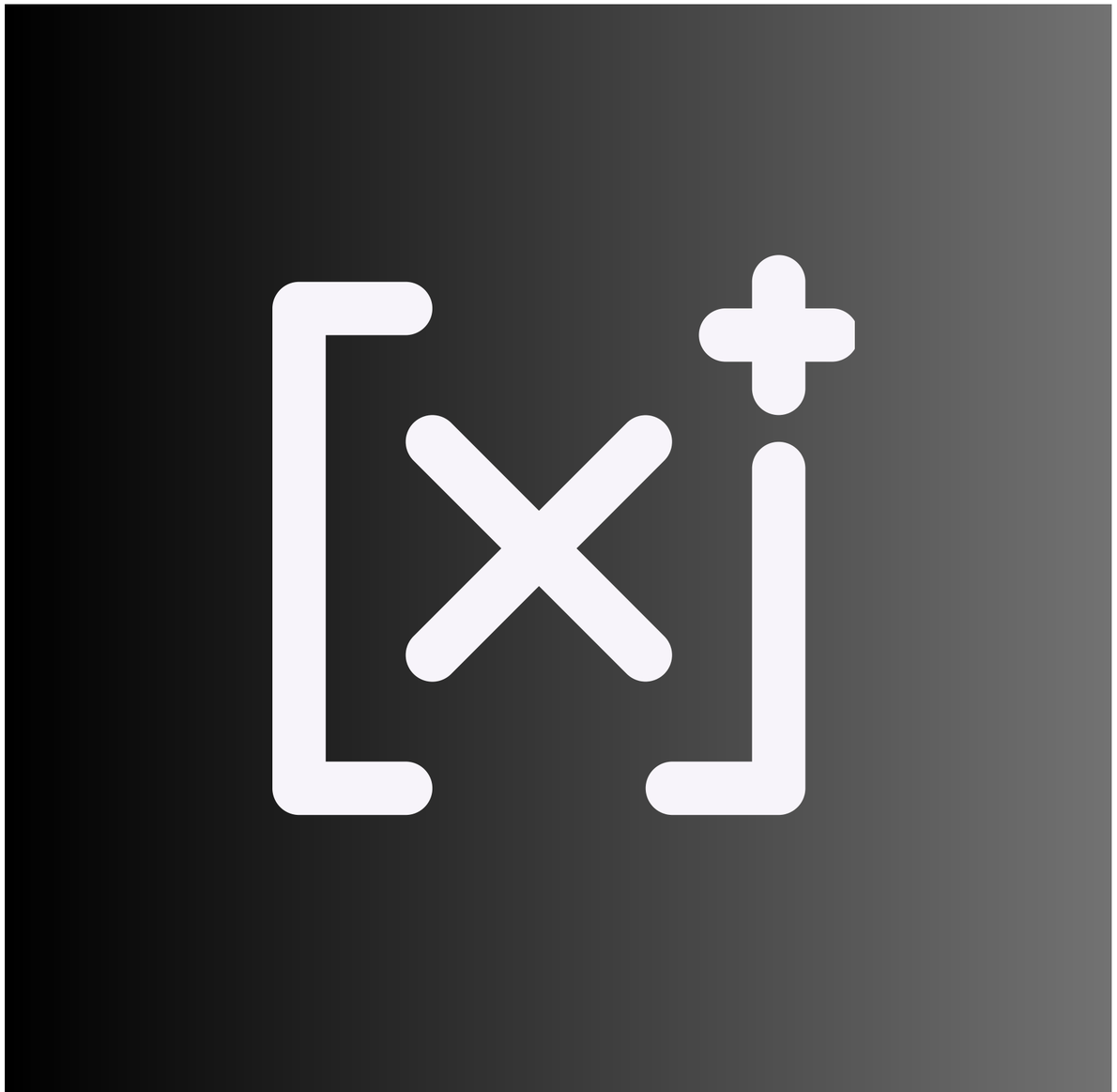
IMPLEMENTATION



PHASE 1

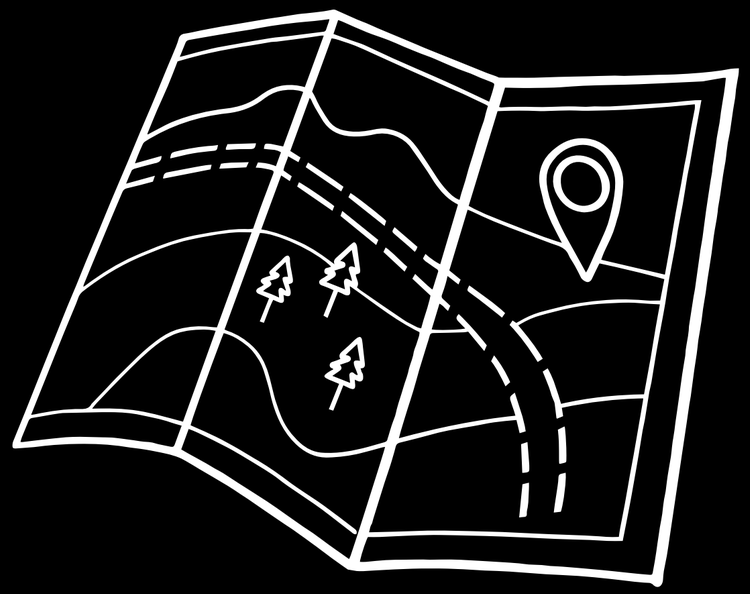
- The function will return the path through all ‘Level’ nodes without the fuel capacity constraint and the total distance travelled.

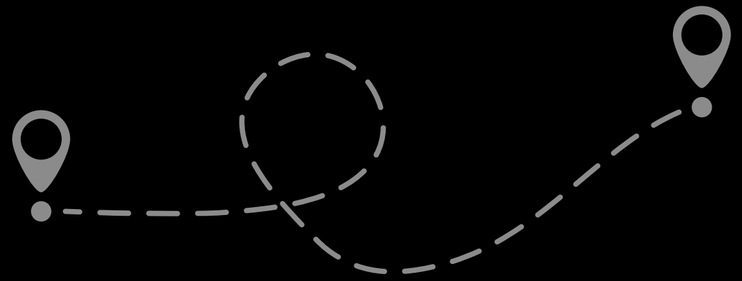
IMPLEMENTATION



PHASE 2

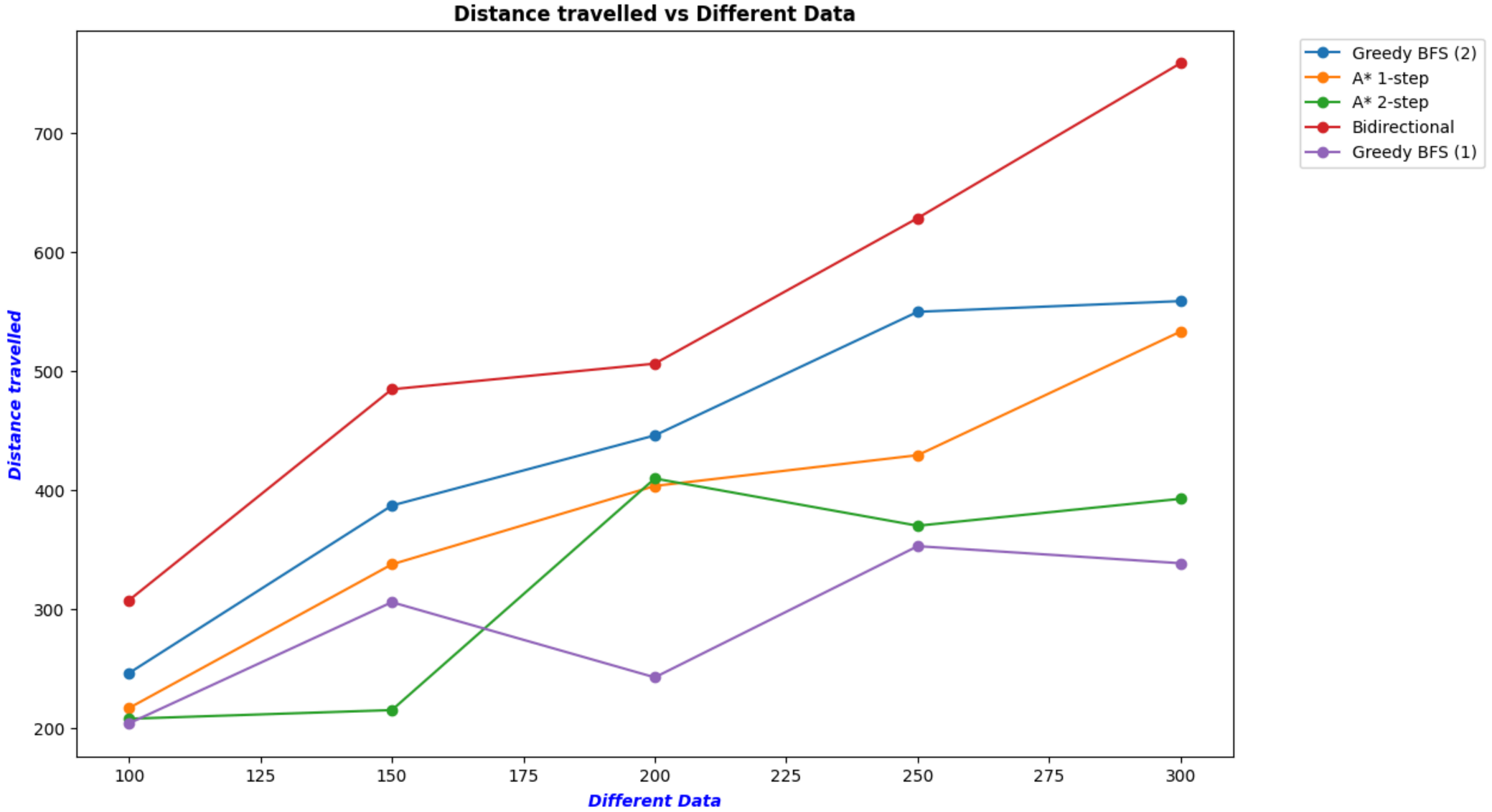
- Given the path in phase 1, for each step check if the total distance travelled violates the fuel capacity constraint, if it does then move to the nearest ‘Station’ node
- Return the final path and update the total distance by adding up the distance from ‘Level’ node to ‘Station’ node





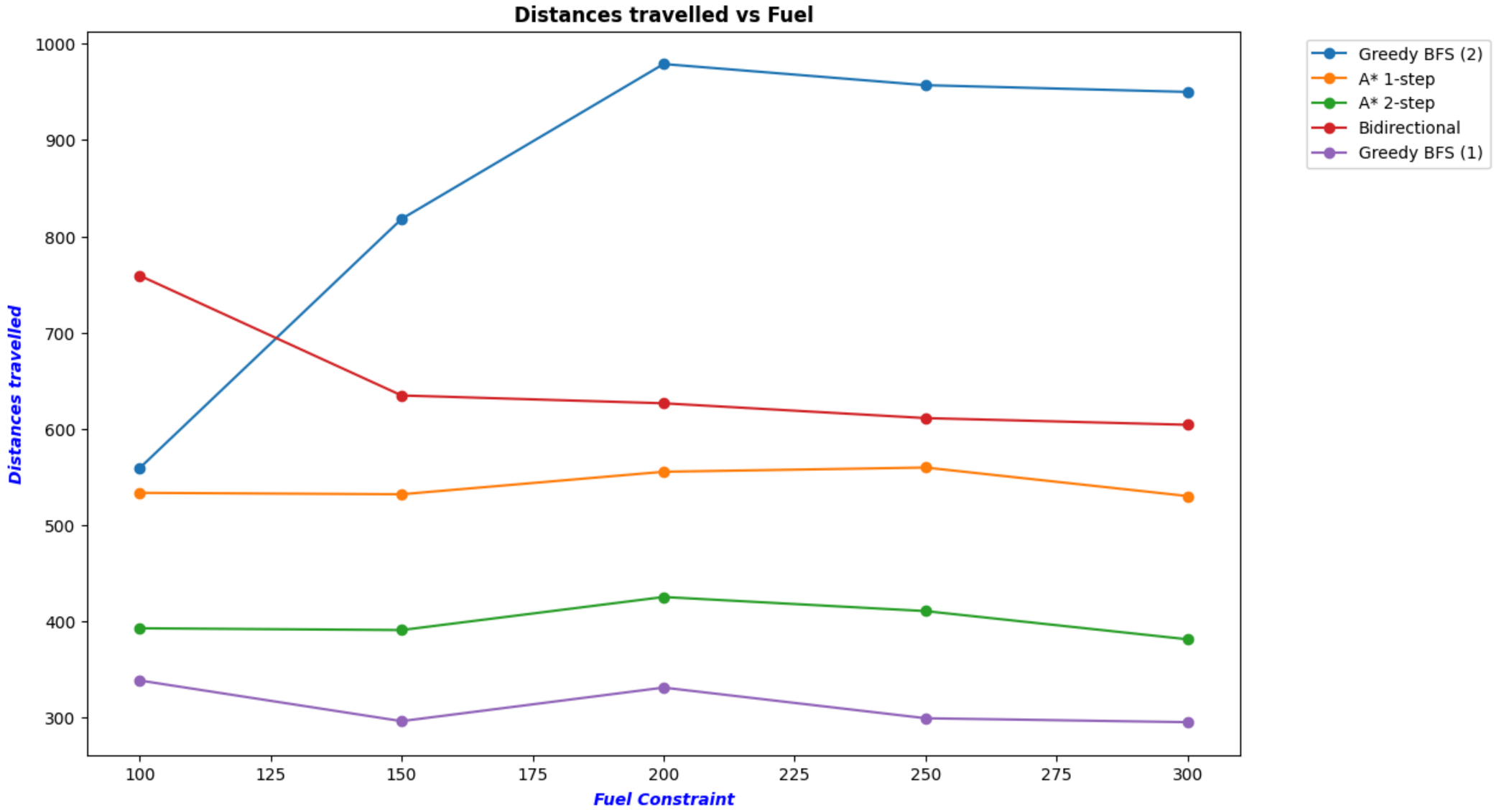
EVALUATION

GROUP 21



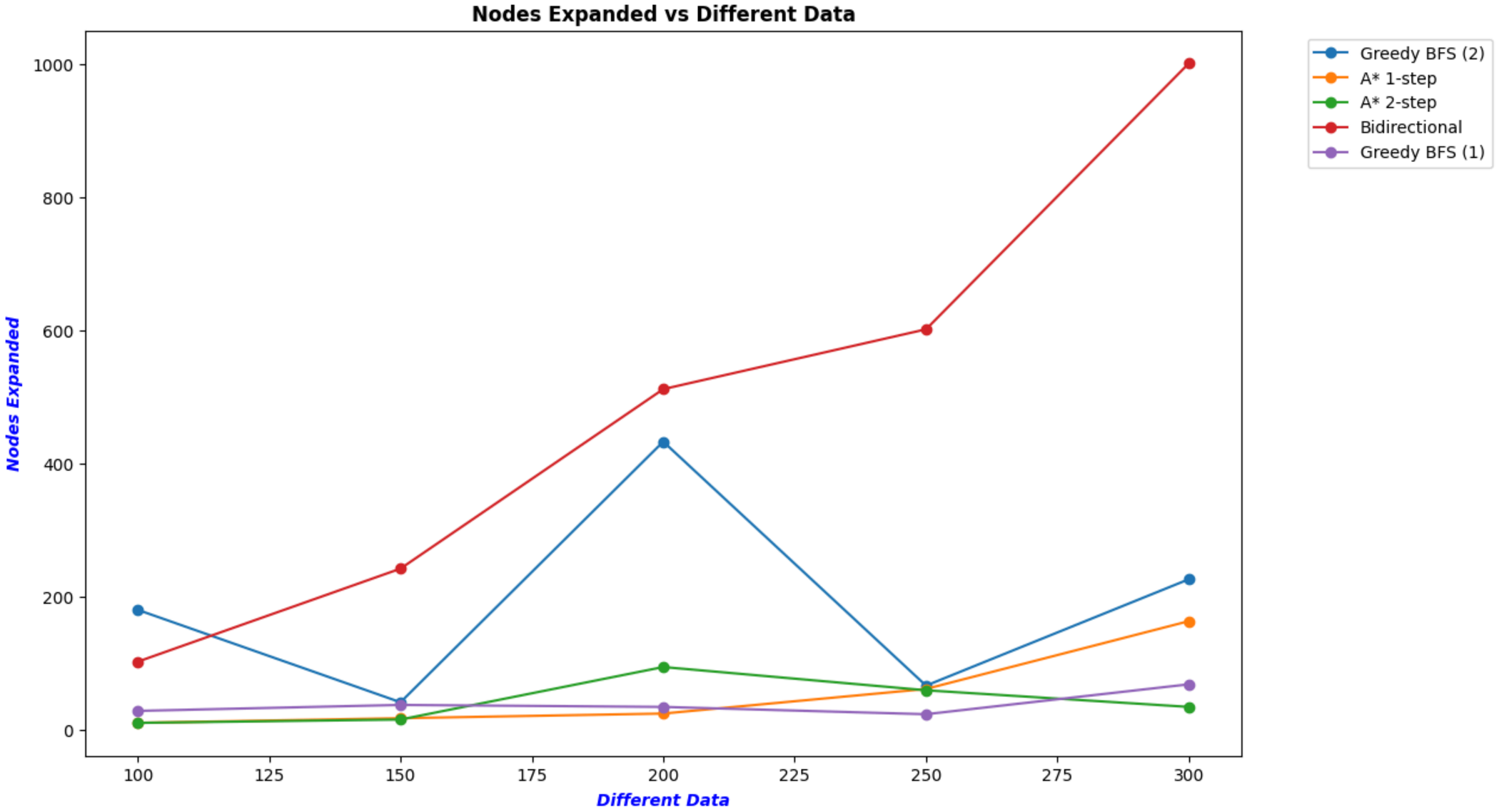
HUST • JAN.2024

TOTAL DISTANCES WITH DIFFERENT INPUT DATA



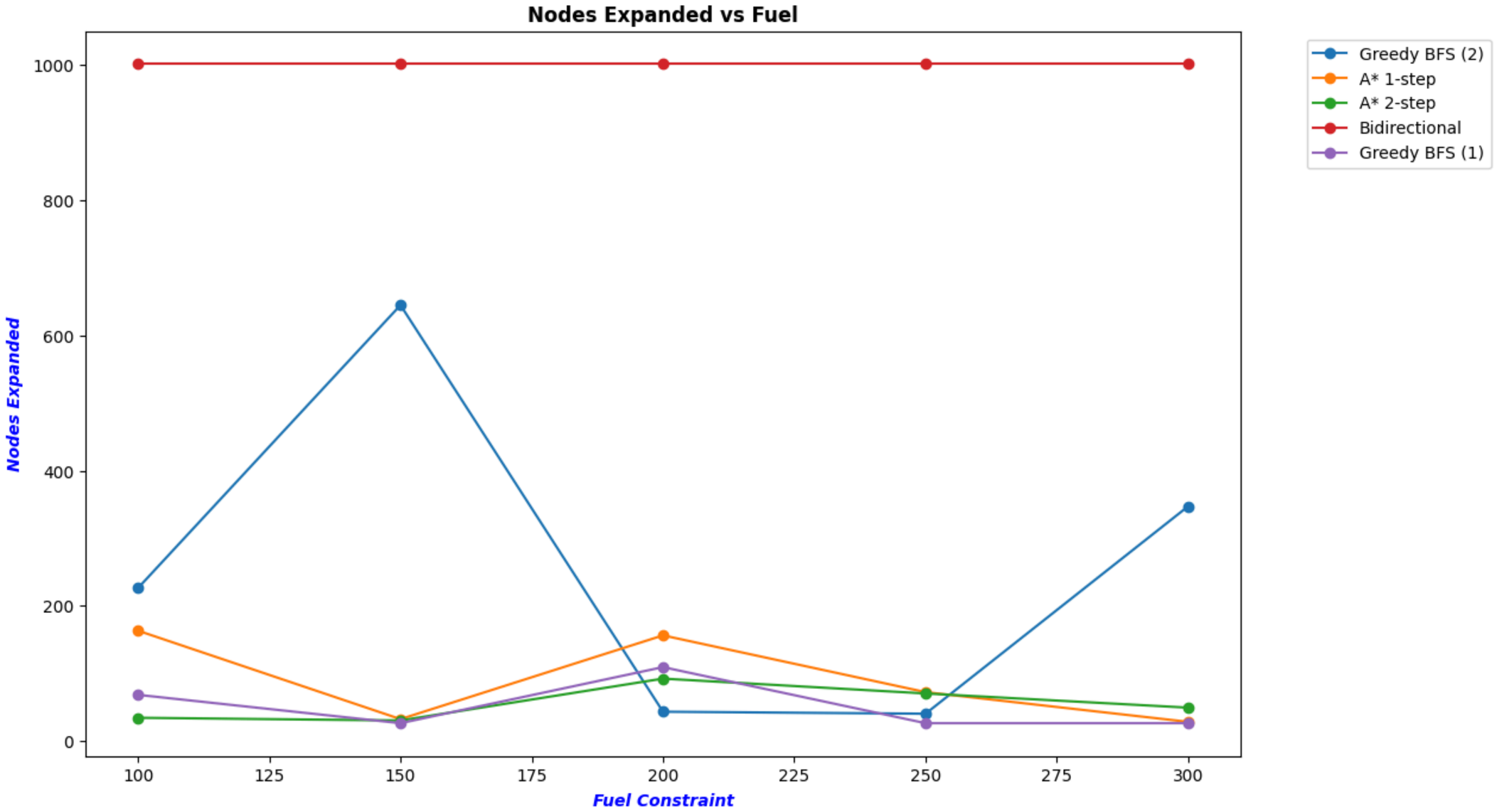
HUST • JAN.2024

TOTAL DISTANCES WITH DIFFERENT FUEL CAPACITY



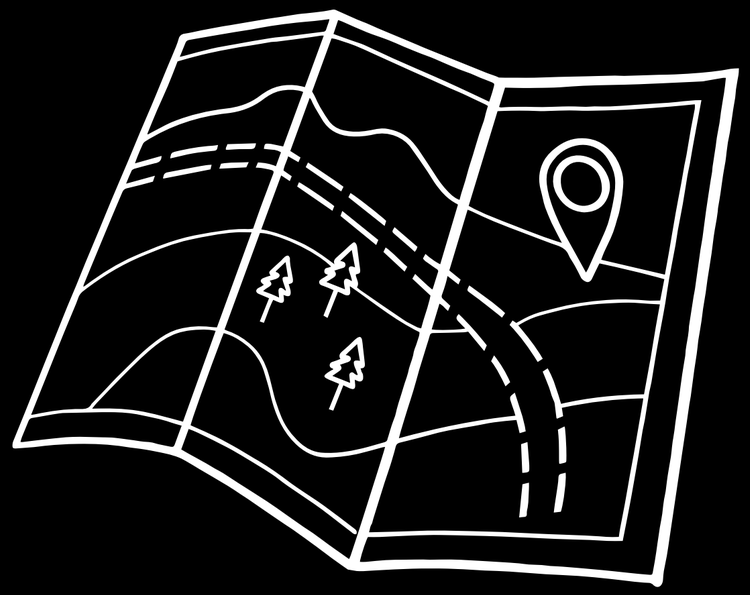
HUST • JAN.2024

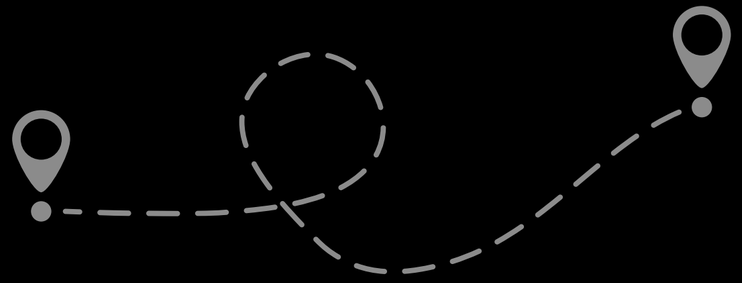
NODES EXPLORED WITH DIFFERENT INPUT DATA



HUST • JAN.2024

NODES EXPLORED WITH DIFFERENT FUEL CAPACITY





CONCLUSION

Design an optimal route with fuel constraints

GROUP 21

CONCLUSION

A* 1-step	Effective in finding the shortest path, even in the presence of fuel constraint.
Greedy (1): Nearest distance to the next position heuristic	Effective in finding the shortest path, even in the presence of fuel constraint.
Greedy(2): Minimal Number of Unvisited Levels	Low effectiveness, not suitable. Only strives to get to the next level while trying not to visit any stations if not needed.
Bidirectional A*	Low effectiveness, not suitable. It consider a large number of paths for both the forward and backward directions.
A* 2-step	One of the effective algorithms. In fact, it is the most efficient algorithm for some datasets and fuel capacities.

CONCLUSION

- The result 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.

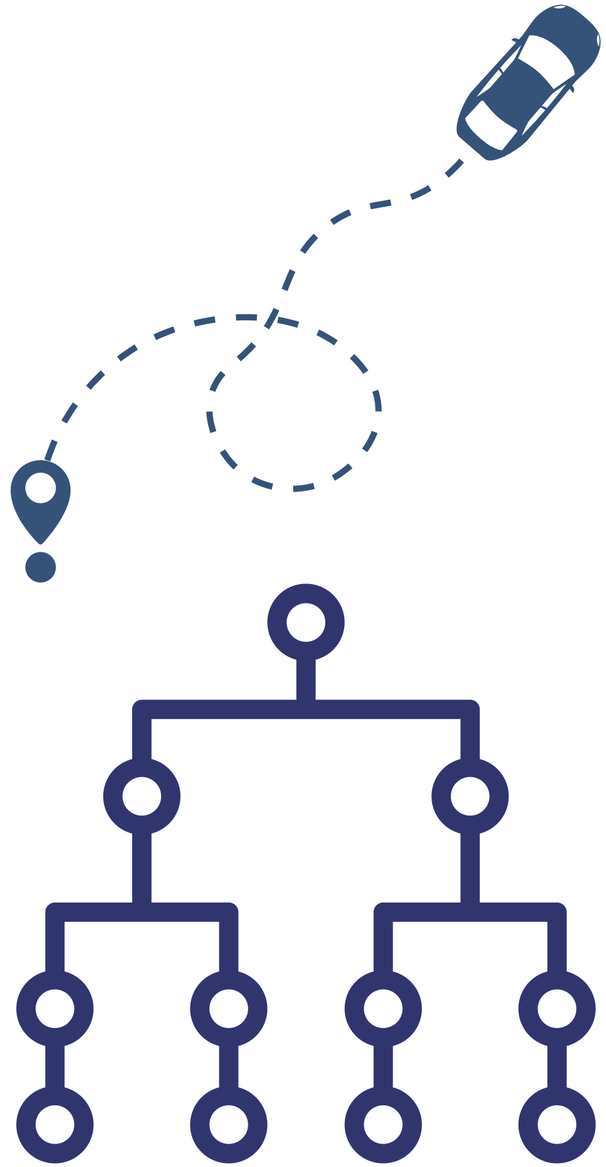
- In most cases, algorithms whose performances are stable through our different data sets should be chosen.

- Here is only the results of data of random points. Need to apply these algorithms in reality network conditions.

- 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.

DESIGN AN OPTIMAL ROUTE
WITH FUEL CONSTRAINTS

Applications in Reality



HUST • Jan.09, 2024

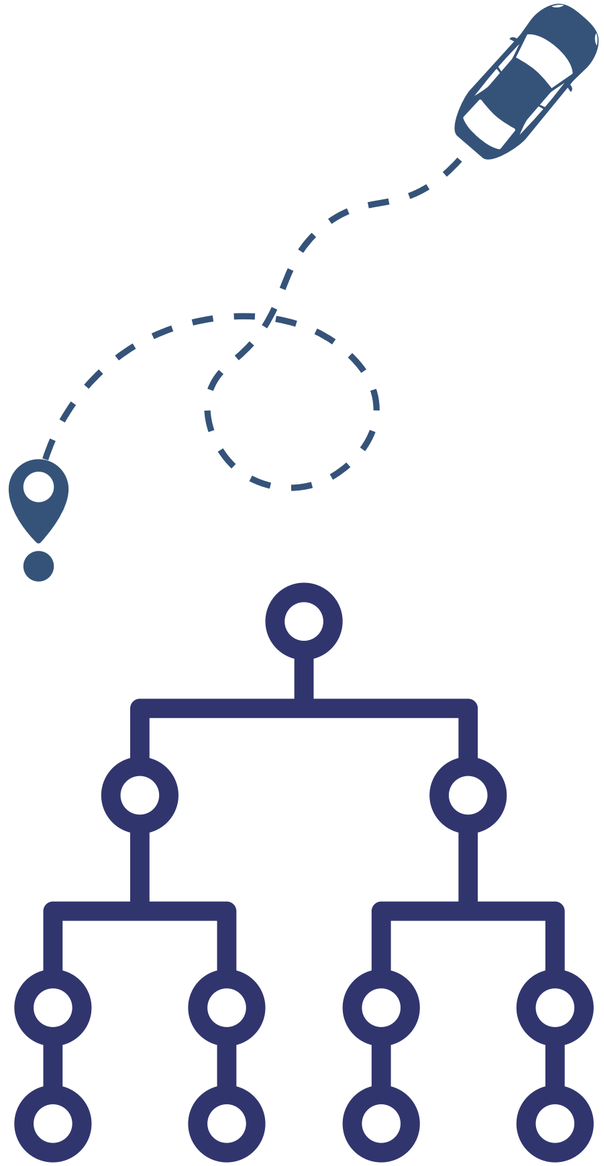
Group 21

Field	Service Type	Description
Logistics and Transportation	Delivery Services	Optimizing routes for delivery trucks, especially for last-mile delivery in urban areas, to ensure the most efficient path is taken while considering the need for refueling or recharging.
Logistics and Transportation	Fleet Management	Managing a fleet of vehicles, whether they're taxis, trucks, or buses, by determining optimal routes that consider traffic, distances, and fuel or charging stations.
Aerospace and Maritime Navigation	Aircraft and Ships Routing	Calculating the most efficient paths for aircraft and ships, considering the need for refueling stops and avoiding hazardous weather conditions or restricted airspace/waters.
Robotics and Drones	Autonomous Vehicles	Implementing algorithms to find the most efficient paths for autonomous cars or drones, considering the need for electric charging stations and navigating through complex environments.
Robotics and Drones	Warehouse Automation	Optimizing the movement of robots in warehouses to pick and place items efficiently, considering battery life constraints.

Field	Service Type	Description
Urban Planning and Infrastructure	Infrastructure Development	Planning the placement of fuel or charging stations in a city or along highways by understanding the typical routes and constraints faced by vehicles.
Urban Planning and Infrastructure	Traffic Management	Improving overall traffic flow by understanding and optimizing the routes taken by the majority of drivers, especially in congested urban areas.
Energy Sector	Supply Chain Optimization	Optimizing the movement of raw materials to production facilities and finished products to markets or ports, considering fuel costs and availability.
Energy Sector	Renewable Energy Integration	Planning routes for electric vehicles considering the location of renewable energy charging stations.
Environmental Conservation	Eco-routing	Developing routes that minimize fuel consumption and emissions, contributing to environmental sustainability and reduced operational costs.

DESIGN AN OPTIMAL ROUTE
WITH FUEL CONSTRAINTS

THANKS FOR LISTENING



HUST • Jan.09, 2024

Group 21