Report about Capstone Project

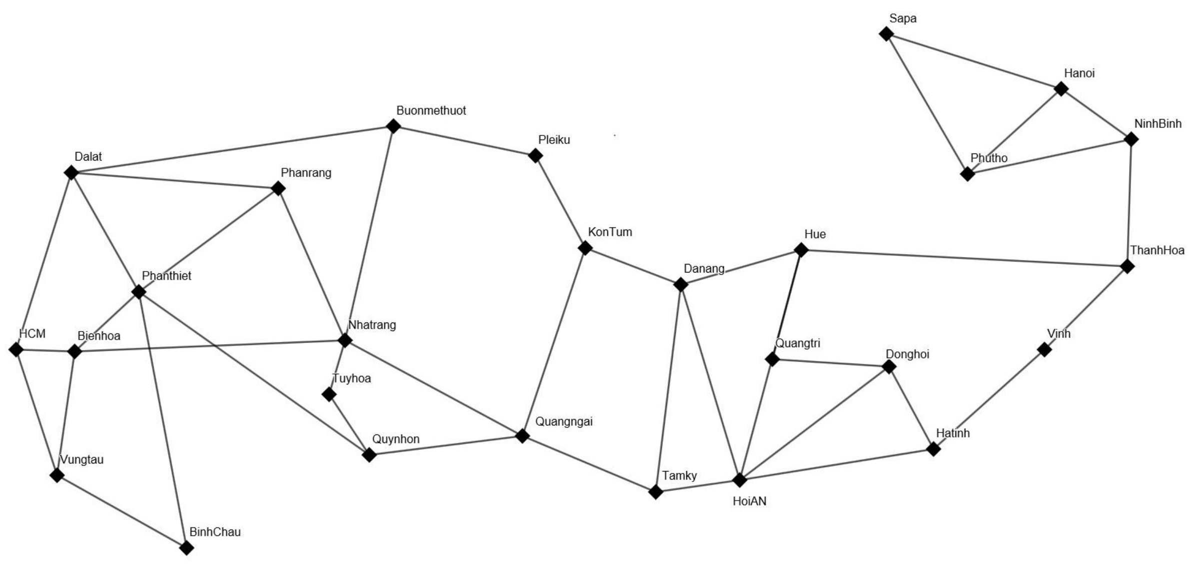
1. Presentation of the subject

If you are a business or organization that has delivery drivers or delivers goods or services to many clients, then this applies to you. Couriers, food delivery services, field service business, florists, and many more have the same **route planning** problem: how to most efficiently plan, create a map and optimize routes to save time for you and your clients.

2. Description of the problem

- Similar the route planning, the intelligent vehicle can only travel between 2 adjacent cities, but each city will open at a limited time, and the objective is to minimize the distance between two cities.
- Formulation:

  + Initial state: In HCM city
  + Action/Transition model: Action (In: HCM) = {Go: DaLat, Go: BienHoa, Go: VungTau}
  + Goal test: {In: Sapa}
  + Path cost: Sum of distance

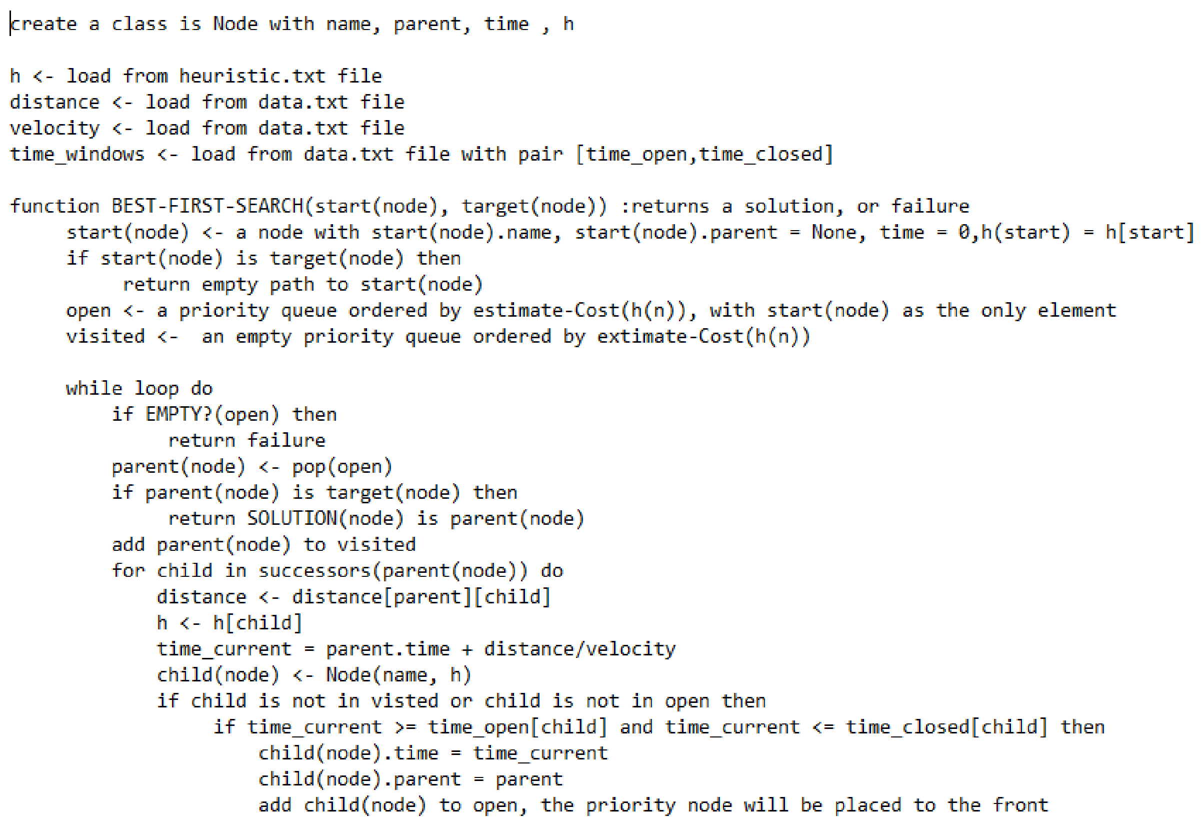


A graph about problem

3. Algorithms for the problem

In this case, we use 3 algorithms: A* search, Uniform-Cost-Search and Best-First-Search. Here are descriptions of 3 algorithms present by Pseudo Code.

- Best first search



```
create a class is Node with name, parent, time , h

h <- load from heuristic.txt file
distance <- load from data.txt file
velocity <- load from data.txt file
time_windows <- load from data.txt file with pair [time_open,time_closed]

function BEST-FIRST-SEARCH(start(node), target(node)) :returns a solution, or failure
     start(node) <- a node with start(node).name, start(node).parent = None, time = 0,h(start) = h[start]
     if start(node) is target(node) then
          return empty path to start(node)
     open <- a priority queue ordered by estimate-Cost(h(n)), with start(node) as the only element
     visited <-  an empty priority queue ordered by extimate-Cost(h(n))

     while loop do
         if EMPTY?(open) then
              return failure
         parent(node) <- pop(open)
         if parent(node) is target(node) then
              return SOLUTION(node) is parent(node)
         add parent(node) to visited
         for child in successors(parent(node)) do
             distance <- distance[parent][child]
             h <- h[child]
             time_current = parent.time + distance/velocity
             child(node) <- Node(name, h)
             if child is not in visted or child is not in open then
                  if time_current >= time_open[child] and time_current <= time_closed[child] then
                      child(node).time = time_current
                      child(node).parent = parent
                      add child(node) to open, the priority node will be placed to the front
```

- A* Search

```
create a class is Node with name, parent, time , h, g, f
f <- g + h

h <- load from heuristic.txt file
distance <- load from data.txt file
velocity <- load from data.txt file
time_windows <- load from data.txt file with pair [time_open,time_closed]

function BEST-FIRST-SEARCH(start(node), target(node)) :returns a solution, or failure
     start(node) <- a node with start(node).name, start(node).parent = None, time = 0,h(start) = h[start],g(start) = 0
     if start(node) is target(node) then
          return empty path to start(node)
     open <- a priority queue ordered by total-estimate-Cost(f(n)), with start(node) as the only element
     visited <-  an empty priority queue ordered by total-extimate-Cost(f(n))

     while loop do
         if EMPTY?(open) then
              return failure
         parent(node) <- pop(open)
         if parent(node) is target(node) then
              return SOLUTION(node) is parent(node)
         add parent(node) to visited
         for child in successors(parent(node)) do
             distance <- distance[parent][child]
             h <- h[child]
             g <- parent.g + distance
             f <- h + g
             time_current = parent.time + distance/velocity
             child(node) <- Node(name, g, h, f)
             if child is not in visted or child is not in open then
                  if time_current >= time_open[child] and time_current <= time_closed[child] then
                      child(node).time = time_current
                      child(node).parent = parent
                      add child(node) to open, the priority node will be placed to the front
```

- Uniform Cost Search

```
create a class is Node with name, parent, time , g

distance <- load from data.txt file
velocity <- load from data.txt file
time_windows <- load from data.txt file with pair [time_open,time_closed]

function UNIFORM-COST-SEARCH(start(node), target(node)) :returns a solution, or failure
     start(node) <- a node with start(node).name, start(node).parent = None, time = 0, g(start) = 0
     if start(node) is target(node) then
          return empty path to start(node)
     open <- a priority queue ordered by path-Cost(g(n)), with start(node) as the only element
     visited <-  an empty priority queue ordered by path-Cost(g(n))

     while loop do
         if EMPTY?(open) then
              return failure
         parent(node) <- pop(open)
         if parent(node) is target(node) then
              return SOLUTION(node) is parent(node)
         add parent(node) to visited
         for child in successors(parent(node)) do
             distance <- distance[parent][child]
             g <- parent.g + distance
             time_current = parent.time + distance/velocity
             child(node) <- Node(name,g)
             if child is not in visted or child is not in open then
                  if time_current >= time_open[child] and time_current <= time_closed[child] then
                      child(node).time = time_current
                      child(node).parent = parent
                      add child(node) to open, the priority node will be placed to the front
```

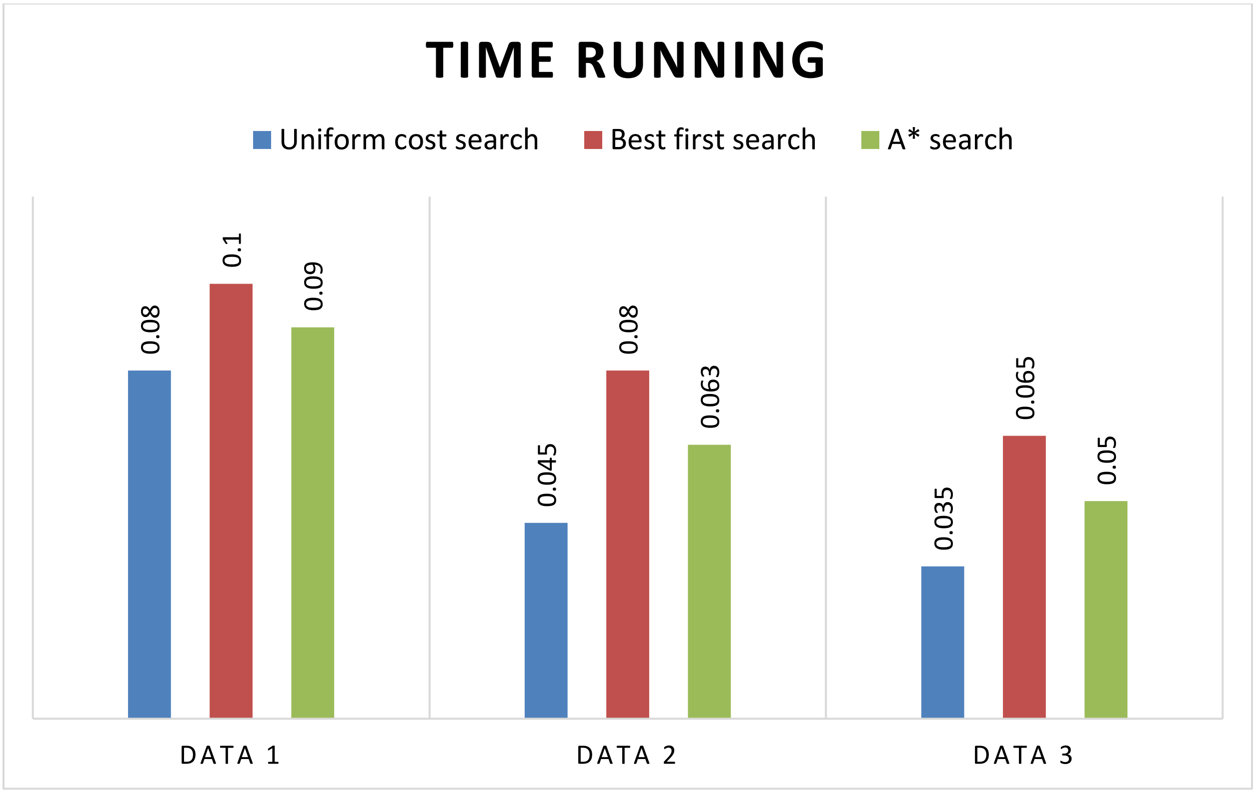
After the experiment, we decide to choose A* search and Uniform-Cost-Search (UCS):

- Firstly, A* and UCS always give the optimal solution, although UCS must have more time to solve the problem.
- On the other hand, Best-First-Search takes less time to solve than UCS but it does not give an optimal solution.

4. **Implementing the algorithms to be used for solving the problem**
    - Firstly, when starting to code, we have many difficulties in selecting methods that are most suitable for our problem.
    - Creating data is also taken much time because we must find the distance of any pair of cities, draw a graph and the most difficult is generating a time window for each city so it can be appropriate as much as possible.
5. **Comparing the result of algorithms**
    Because our project is Route Planning with Time Window, it is a special case that we just generate data by hand so the percent of algorithms successfully solve the problem is 100%

Comparison of Search Strategy

Algorithm	Complete?	Optimal?	Time complexity	Space complexity
UCS	Yes	Yes	Number of nodes with g(n) ≤ C*	
BFS	No	No	Worst case: $O(b^m)$ Best case: $O(bd)$	
A*	Yes	Yes	Number of nodes with g(n) + h(n) ≤ C*	



According to data on 2 tables, we can see that:

- A* search is most appropriate for the problem because it combines the optimal efficiency of BFS (Greedy) and the optimal solution of UCS.
- On the other hand, A* search requires the user input the Euclidean distance. This is the weak point of A* Algorithms.

6. Conclusion and possible extension

- We have been working together, staying up all night together, becoming closer. Besides, we have improved our English as well as teamwork and presentation skills.
- If we have more time, we will do more carefully, expanding the number of roads between cities as well as the number of input cities (similar to the map of Vietnam).
- Perhaps the most difficult thing about scaling up is the time windows because just being wrong in one city cannot find a solution to the problem.