ROUTES PROBLEM 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 sales, florists, and many more have the same **route planning** problem: make routes more profitable, create a map and optimize routes to save time and also resources.

Our aims of this project are to try to approach the problem with our algorithms and applying on the 63 cities of Vietnam.

2. Description of the problem

- Similar the route planning, the intelligent vehicle can only travel from the current city to the city near it within a certain time, and also additional time if it goes at peak hour. The objective is to try to minimize the time cost to travel between the two cities.
- Formulation:

+ Initial state: In starting city

+ Action/Transition model: Action {In: starting city} = {Go: adjacent cities}

+ Goal test: {In: Goal city}

+ Path cost: Sum of distance traveling

3. Algorithms for the problem

The UCS function here is a modified version of the normal UCS. Instead of using a normal queue, we use a priority queue. This queue will

automatically compare the distance cost between variables when a new city class variable is added, and the smallest value will be placed first, no matter the current city it is, using the heap sort. This approach is for more efficiency.

```
create a class which name is city containing id, distance_cost, path, name

function UCS_search(start,goal):
        q = empty priority queue
        p = city() with p.name = start, p.distance_cost = 0
        create empty list named visited
        while q is not empty do:
                take the first class variable in q and dequeue it
                if p.name is the goal then:
                        return p.distance_cost, p.path
                if p.name not in visited then:
                        for city that is adjacent to p.name do:
                                p1 = city() with p1.name = adjacent_city
                                p1.distance_cost = p.distance_cost + the cost to go to adjacent city
                                create a list p1.path equal the list p.path plus the adjacent_city
                                add p1 to the q
                                add adjacent_city to the visited
        return 0 if not find the way to the goal
```

We also apply UCS to solve the minimum time cost (with both traffic and not). We only present it in Pre Code

```
function UCS_search(start,goal,current_time):
        q <- empty priority queue
        p <- city() with p.name <- start, p.time_cost <- 0,p.current_time <- current_time
        create empty list named visited
        while q is not empty do:
                take the first class variable in q and dequeue it
                if p.name is the goal then:
                        return p.time_cost, p.path
                if p.name not in visited then:
                        for city that is adjacent to p.name do:
                                p1 <- city() with p1.name <- adjacent_city
                                if current_time in a traffic time of a the next city do:
                                        time_cost <- time_cost + traffic_time
                                p1.time_cost <- p.time_cost + time_cost
                                create a list p1.path equal the list p.path plus the adjacent_city
                                add p1 to the end of q
                                add adjacent_city to the visited
        return 0 if not find the way to the goal
```

The A* star function is similar to the UCS but we will compare air distance cost plus the normal distance cost, as the heuristic variable.

```
create a class which name is city containing id, distance_cost, path, name, heuristic

function A*_search(start,goal):
        q = empty priority queue
        p = city() with p.name = start, p.distance_cost = 0
        create empty list named visited
        while q is not empty do:
                take the first class variable in q and dequeue it
                if p.name is the goal then:
                        return p.distance_cost, p.path
                if p.name not in visited then:
                        for city that is adjacent to p.name do:
                                p1 = city() with p1.name = adjacent_city
                                p1.distance_cost = p.distance_cost + the cost to go to adjacent city
                                p1.heuristic = p.distance_cost + 2 * the cost to go to adjacent city with air distance
                                create a list p1.path equal the list p.path plus the adjacent_city
                                add p1 to the q
                                add adjacent_city to the visited
        return 0 if not find the way to the goal
```

We multiply the air distance cost by 2 here because after some tries we find that this heuristic will dominate run faster but it sometimes won't give the best answer.

NOTE: In the topic, we mention the complete search but we don't use it, after discussing and reading some materials about the problem, we just want to consider the efficient difference between the A* and the UCS algorithms. Also, the complete search can present by using backtracking with the maximum number of cities to visit. But we all agree that it's much worse than UCS.

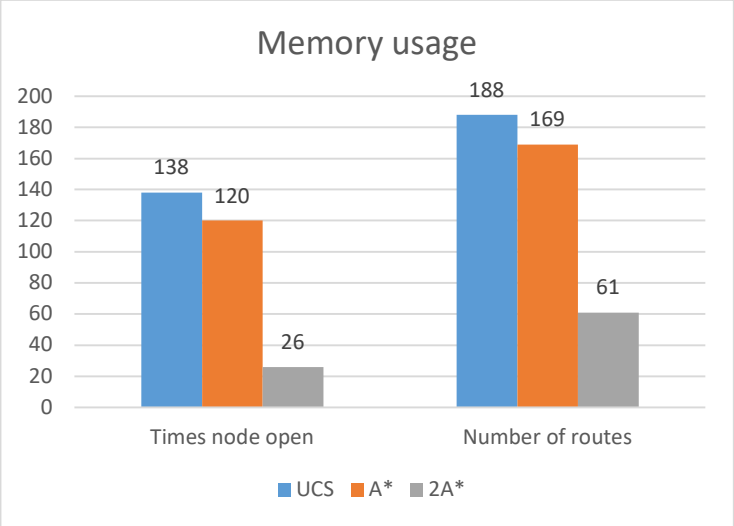
4. Performance comparison between algorithms

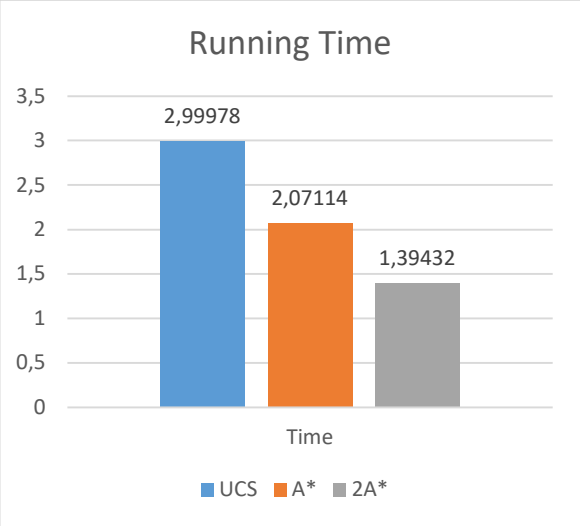
We would want to observe the efficiency of two algorithms also the base A* with no multiplying. And there are two kinds of comparing we think of:

- The distance needed to travel in terms of the time cost to run the function: This is not consistent because the distance path connecting city to city depends on the complexity of the path, something like it is curved or straight, so it is hard to determine if the algorithm is effective.

- The number of times it visits cities and which we define as times node open, and the number of times it travels to the next city which we define as number of routes. And also, the running time between the three algorithms: This comparison we think will present the efficient performance more obviously.

We perform a test case from Hanoi to Ho chi Minh City





As we can see, the 2 A* dominates in both the memory usage and running time test and give a far better result than the two rests. The A* performs better than the UCS but just a little bit. The UCS has the worst performance of all three in terms of using memory and time.

5. Problems occurring during working on the assignment and how we handled them

- Input data takes too much time to make. Because of the approach in which we attempt to solve the problem, we need to motivate the team and try hard to input data by hand including creating a distance matrix between cities, and an additional time cost matrix with google support so it can be as accurate as possible.

- Additional time is hard to implement. Our based knowledge is not too much, so we struggle to learn or find a function that can predict traffic time. At last, we came to a simple code, trying to add time costs when traveling at a certain time. Not the best but the only solution we find.
- Call API of Google is not available in Vietnam. At first, we don't know what to do because we only know and are familiar with google maps. However, then we know and learn to use the Goong map API but it is a bit annoying because you need the city coordinate to call it precisely.

6. Conclusion and possible extensions

Conclusions:

- The n*n matrix data input, each of the cities does not loop through all the rest city, only to the adjacent city, so the time complexity of the two algorithms is O(b^m).
- The heuristic is not as far as good. It can give the optimal time and use less memory if the path connecting cities is straight but the more complex it is, the less effective the A* function is, and in some cases, the better UCS performance is in time cost.
- The A* and UCS can give optimal results in most of starting and goal cities because the Vietnam city map is not too complex and also central Vietnam is at most two adjacent cities (not including the previous city) for each city in here which are much more simple for the algorithms.

Possible extension:

- Try to predict the traffic time by google support and machine learning instead of giving it a certain time. Because time and resources are limited, we can't give the best way to implement this but only the simple one. In the future, a better prediction algorithm will give a more accurate time cost.
- Using a tree as input data rather than a matrix for less complicated and reduces the unnecessary loop. With the linked list data structure, the adjacent city with the leftmost city and right city (not necessarily to be in order), also includes the previous city that we travel from. The cycle will be prevented by adding the visited list. The space cost for data input will be less than the matrix and more efficient.
- Add the borderline between cities and some specific places in each city as the starting or destination point to achieve higher accuracy for possible future applications instead of just traveling from city to city.

7. List of bibliographic references

https://docs.goong.io/rest/ using: API key, distance matrix

https://support.route4me.com/faq/route-planning-glossary/what-is-route-planning/ using: introduction