AI Capstone Project – Group 5

Name	Student ID	Mail
Tran Vuong Quoc Dat	20200145	dat.tvp200145@sis.hust.edu.vn
Nguyen Truong Truong An	20204866	an.ntt204866@sis.hust.edu.vn
Le Duc Anh Tuan	20204929	tuan.lda204929@sis.hust.edu.vn
Nguyen Nho Trung	20204894	trung.nn204894@sis.hust.edu.vn
Nguyen Thanh Dat	20204903	dat.nt204903@sis.hust.edu.vn

1. Introduction

Write a program to find the optimal path between 2 warehouses so that both time and cost are optimized(e.g., warehouse in Hanoi and Vinh city). The warehouse system is divided into two types: main warehouse and secondary warehouse

Goods between any 2 main warehouses can be transported directly by plane, the default speed is v_plane km/h. Goods between 2 secondary warehouses or between 1 secondary warehouse to 1 main warehouse and vice versa will go by road, the default speed is v_truck km/h. Every time it passes through a main warehouse, the goods must be stored there for k hours.

Input:

- A map of cities in Vietnam with cities and distance between them, and by default, each city will have one warehouse. There are 3 main warehouses, they will be randomly generated and distributed co that Each domain will have 1 main warehouse(E.g: [Ha Noi in the north, Da Nang in the central and Ho Chi Minh in the south]
- file 'data_excel.xlsx':Contains data on the distance between the warehouses according to the crow's flight path

- start warehouse and end warehouse. start and ending warehouse can be either main warehouse or secondary warehouse

```
start warehouse:Vinh Phuc

end warehouse: Son La
```

Output:

- The path from start warehouse to end warehouse
- The cost parameter represents the optimization function of the path, the smaller this parameter is, the more optimal the path is both in terms of time and transportation cost.

```
main warehouse:  Son La , Nghe An , Ho Chi Minh
Path: Son La <- Phu Tho <- Vinh Phuc <-
Cost:  99.22415194416666
```

2. Implementation plan

2.1. Pre-processing

- Collect coordinate data and create a dataset of 64 provinces in Vietnam, then by default, there will be 1 secondary warehouse in each province
- Processing from coordinate data to distance data
- Collect v_truck: truck speed, v_plane: plane speed, cost_truck: truck cost, cost_plane: plane cost
- Convert distance data into evaluation parameter for each algorithm

  A,B,C,D: node represents the warehouses

  - AStar:

    A dictionary contains nodes, g(x) and h(x)

    The data structure:

    ```
    data = {‘A’ : [‘B’,g(B),’C’,g(C),’D’,g(D)’,h(A)] }
    ```

    evaluation function:

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

    Nodes with small f(x) will be given priority
  - UCS:

    a dictionary contains nodes and g(x)

    The data structure:

    ```
    data = {‘A’ : [‘B’,g(B),’C’,g(C),’D’,g(D)’] }
    ```

    evaluation function:

    $$f(x) = g(x)$$
  - DFS

    a dictionary contains nodes and g(x)

    The data structure:

    ```
    data = {‘A’ : [‘B’,’C’,’D’] }
    ```

    This algorithm doesn’t have an evaluation function

Approved according to FILO rules

- IDA:

  Data get directly from excel file

  evaluation function:

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

Parameter formula:

- g(x): cost parameter from node x to node parent
  Fomulation: g(x) = (distance/v + storage_time) * cost
  E.g: g(B) = cost parameter from B to A(A is B’s parent node)

- h(x): cost parameter from node x to node goal
  Fomulation: h(x) = (distance/v) * 2 * cost
  E.g: h(B) = cost parameter from B to E(E is the end warehouse)

By default, between 2 main warehouse: v = v_plane = 800km/h, cost = cost_plane = 160.000 vnđ/kg, storage_time = 2h
In other cases: v = v_truck = 60km/h, cost = cost_truck = 35.000 vnđ/kg, storage_time = 0h

2.2.Deployment

2.2.1. Data constructor programming:

Create separate data for each algorithm and package it into modules, others just need to import and use

2.2.2. Programming algorithms

We program 4 algorithms:

- A*
- Uniform cost search
- IDA*
- Depth First Search

2.2.3. Apply algorithm, use constraints to find the optimal path:

- If the start warehouse and end warehouse aren’t in the same domain:
  Good must be transported to the main warehouse that placed in the same area to start warehouse before transported to end warehouse.
- If the start warehouse and end warehouse aren’t in the same domain:

If the path from the start warehouse to the main warehouse is the longest path of the triangle start warehouse - main warehouse - end warehouse that goods must be transported to the main warehouse before being transported to the end warehouse.
In other cases, goods can be transported directly from the start warehouse to the end warehouse.

2.3.Algorithm optimization and synchronization

We use Github to optimize storage and code updates, Teams, and Whiteboards to exchange ideas and work.

3. Algorithms

A*

```
def AStar(S = node(start), G = node(end)):
    # initial open queue and closed queue
    open = PriorityQueue()
    closed = PriorityQueue()

    # add attribute S (node_start) to open queue to start expand
    S.h = data[S.name][-1]  # h(x):
    S.g = 0                 # g(x)
    open.put(S)             # put to Priority Queue

    # while loop for check all city
    while True:
        if open.empty():
            print("Can't solve!")
            break
        else:
            expand_node = open.get(0)    # take out each element in the open queue by index and expand it; after open  close now
            closed.put(expand_node)      # put expanded nodes to closed, so that these nodes are not expanded again
            expand_node.h = data[expand_node.name][-1]
            print('Scan', expand_node.name, expand_node.h, expand_node.g)
```

```
            # Finding Solution
            if expand_node.name == G.name:   #goal
                print('SOLVE SUCCESSFULLY!\n\nPath to goal city:', end =" ")
                last = data[G.name][-1]
                expand_node.parent(distance = 0,last_h = last)          # print path after solving
                break
            else:
                # scan others node, browse all elements of node 0
                for i in range (0, len(data[expand_node.name])-1, 2):
                    tmp = node(data[expand_node.name][i])
                    tmp.h = data[tmp.name][-1]                          # h(x)
                    tmp.g = expand_node.g + data[expand_node.name][i+1] # g(x)
                    tmp.par = expand_node                               # name of parent
                    tmp.w = data[expand_node.name][i+1]                 # distance current node to parent

                    if tmp not in open.queue and tmp not in closed.queue:
                        open.put(tmp)                                   # put new node to queue

# Run algorithm
AStar()
#print time complexity
print("\nRunning time: {}(s)".format(time.time() - start_time))
```

- *Completeness*: Not complete. It is only complete if the state space is finite, and we avoid repeated states and all costs are >ε.

- *Time complexity*: The number of nodes expanded is exponential in the depth of the solution (the shortest path) d: $O(b^d)$, where b is the branching factor (the average number of successors per state).

- *Space complexity*: $O(b^d)$, It keeps all the generated nodes in memory.

- *Optimality*: Expand node in frontier with best evaluation function score f(n):

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

  + h(n) : heuristic estimate of cost to get from n to goal.

  + g(n) : cost to get from initial state to n.

- Optimal when h(n) is admissible.

Deep First Search:

```
def DFS (S = node(start), G = node(end)):
    print('DFS(',start,end,')')
    open = []
    closed = []
    S.g = 0
    open.append(S)
    i = 0
    while True:
        i+=1
        #print('STEP ',i,': open = ',[x.name for x in open])
        if len(open) == 0:
            print("can't solve")
            break
        else:
            O = open.pop(0)
            closed.append(O)
            lat_check.append(lat[O.name])  #add to expanded node to visualize after
            long_check.append(long[O.name]) #add to expanded node to visualize after
            if len(data[O.name])!=0:
                pass
                #print('Scan',O.name)
                #print()
```

```python
            if O.name == G.name:
                print("\033[1m" + 'solve successfully' + "\033[0m")
                print()
                O.parent()
                break
            else:
                pos = 0
                for i in range(0,len(data[O.name]),2):
                    tmp = node(data[O.name][i])
                    tmp.g = O.g + data[O.name][i+1]
                    tmp.par = O
                    if tmp not in open and tmp not in closed:
                        open.insert(pos,tmp)
                        pos+=1
```

- *Completeness*: Not complete. But,it is complete in finite search spaces

- *Time complexity*: $O(b^m)$, where b is the branching factor (the average number of successors per state) and m is maximum depth of the state space.

- *Space complexity*: Similar to time complexity, it is also $O(b^m)$, keep all nodes in memory.

- *Optimality*: Not optimal

IDA*

```python
import time
start_time = time.time()
start= input("the start city is : ")
goal= input("the goal city is : ")
list_node_previous={} # solution path
def IDA_star():
    global list_node_previous
    #define a threshold: theta =f(root_node) with f(n)=h(n)+g(n)
    threshold=list_distance[dict_index_province[start]][dict_index_province[goal]]
    while True: # run infinity
        temp=search(start,0,threshold,parent='Null') #function search(node,g score,threshold)
        if temp=='FOUND': #if goal found
            return ('FOUND',threshold)
        threshold=temp

def search(node,g,threshold,parent):  #recursive function
    f=g+list_distance[dict_index_province[node]][dict_index_province[goal]]
    if (f>threshold):#greater f encountered
        return f
    if (f<=threshold):
        list_node_previous[node]=parent
    if node==goal:   #Goal node found
        return 'FOUND'
    minn=10**10 #minn= Minimum integer
```

```
    for tempnode in nextnodes(node):
        #recursive call with next node as current node for depth search
        temp=search(tempnode,g+list_distance[dict_index_province[node]][dict_index_province[tempnode]],threshold,node)
        if temp=='FOUND':# if goal found
            return 'FOUND'
        if (temp<minn):# find the minimum of all f greater than threshold encountered
            minn=temp
    return minn #//return the minimum f encountered greater than threshold

def nextnodes(node):
    return data_neighbour[node] #return list of all possible next nodes from node
print(IDA_star())

end_time=time.time()
print(end_time-start_time)

solution_path_optimal=[]
check=goal
solution_path_optimal.append(check)

while check!=start:
    check=list_node_previous[check]
    solution_path_optimal.append(check)
#print(solution_path_optimal)

print('Solution path of problem is')
print(' -> '.join(solution_path_optimal[::-1]))
```

- *Completeness and optimal*: It is only complete if : h is admissible and , Finite branching factor

- *Time complexity*: The number of nodes expanded is exponential in the depth of the solution (the shortest path) d: O(b^d), where b is the branching factor (the average number of successors per state).

- *Space complexity*: O(bd) in the worst case.

Uniform cost search:

```
def uniform_cost_search(S = node(start), G = node(end)):
    print('Uniform cost search')
    open = []
    closed = []
    open.append(S)
    while True:
        if len(open) == 0:
            print('failed to solve')
            break
        else:
            print([x.name for x in open])
            O = open.pop(0)
            lat_check.append(lat[O.name])
            long_check.append(long[O.name])
            closed.append(O)
            if len(data[O.name])!=0:
                print('Scan',O.name)
            print()
            if O.name == G.name:
                print("\033[1m" + 'solve successfully' + "\033[0m")
                print()
                O.parent()
                break
            else:
                for j in range (0,len(data[O.name]),2):
                    tmp = node(data[O.name][j])
                    tmp.par = O
                    if tmp not in open and tmp not in closed:
                        open.append(tmp)
                        tmp.g = O.g + data[O.name][j+1]
uniform_cost_search()
print('time:', time.time()- start_time,'s')
```

- *Completeness*: Yes, if step cost ≥ ε.

- *Time complexity*: O ($b^{1+(\frac{C*}{\varepsilon})}$) where C* is the cost of the optimal solution.

- *Space complexity*: O ($b^{1+(\frac{C*}{\varepsilon})}$).

- *Optimality*: Yes - nodes expanded in increasing order of g(n).

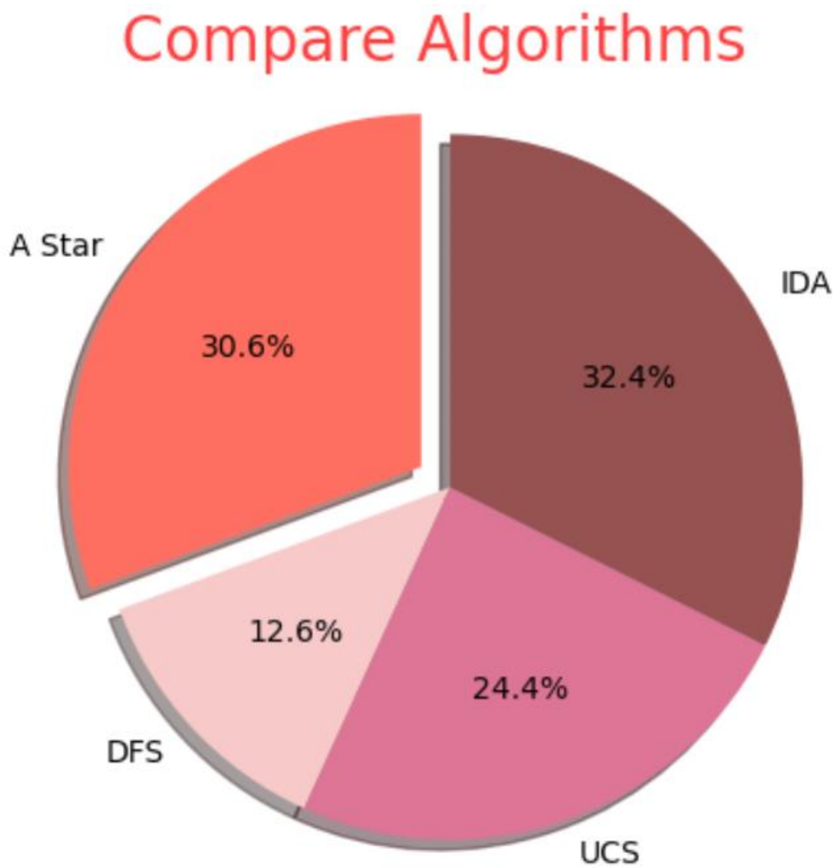
4.Comparing the result of the algorithm used for solving the problem:

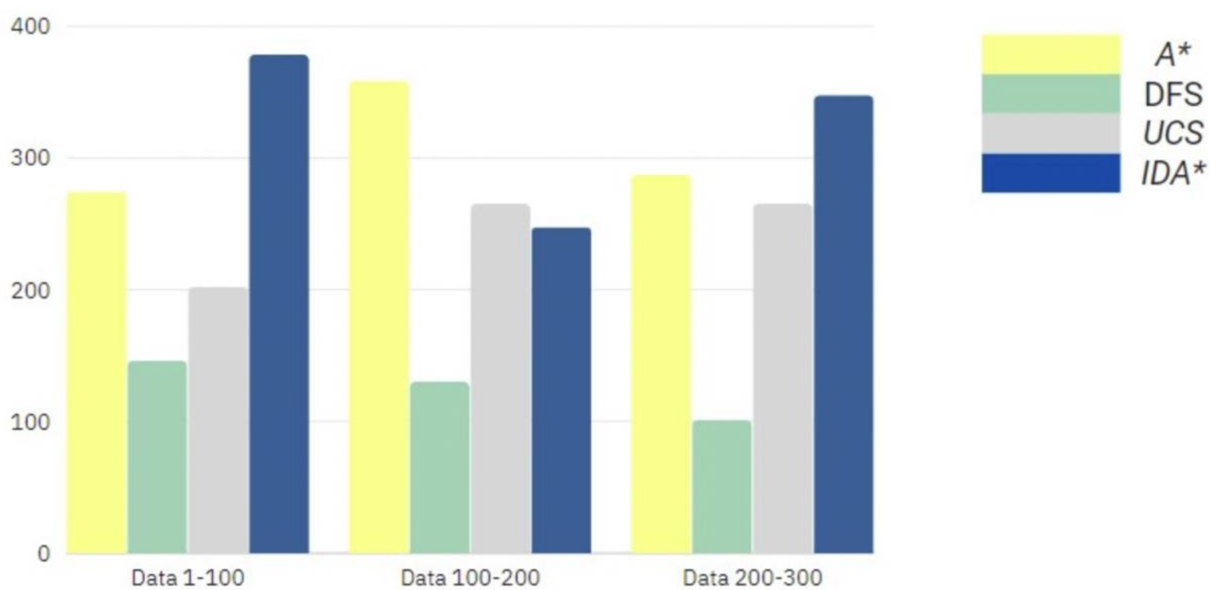
4.1. Random seed data for accuracy

With the data of 63 provinces, we used the function random.seed(42) to randomize the main warehouse as Son La, Nghe An and Ho Chi Minh, with the main warehouse list taking the first 10 elements of each domain (seed(42)). Exporting to a CSV file, we have data of 300 interconnected components, and we compared each data line of 4 algorithms with the order of scoring 1,2,3 and 4 with min cost of 4. , max cost is 1. Below is code we used:

```
(as_point, dfs_point, ucs_point, ida_point) = (0,0,0,0)

# Find order 1,2,3,4 for each data of algorithm
for index in range(300):
    # Store value
    list_store_order = [cost_as[index], cost_dfs[index], cost_ucs[index], cost_ida[index]]
    max_a_line = max(cost_as[index], cost_dfs[index], cost_ucs[index], cost_ida[index])
    min_a_line = min(cost_as[index], cost_dfs[index], cost_ucs[index], cost_ida[index])

    # Delete max & min to find 2nd & 3rd
    list_store_order.remove(max_a_line)
    list_store_order.remove(min_a_line)

    # Find max 1
    if max_a_line in cost_as:
        as_point += 1
    elif max_a_line in cost_dfs:
        dfs_point += 1
    elif max_a_line in cost_ucs:
        ucs_point += 1
    elif max_a_line in cost_ida:
        ida_point += 1

    # Find min 4
    if min_a_line in cost_as:
        as_point += 4
    elif min_a_line in cost_dfs:
        dfs_point += 4
    elif min_a_line in cost_ucs:
        ucs_point += 4
    elif min_a_line in cost_ida:
        ida_point += 4

    # Find 2nd & 3rd
    order_2, order_3 = min(list_store_order), max(list_store_order)
    # Find 2nd
    if order_2 in cost_as:
        as_point += 3
    elif order_2 in cost_dfs:
        dfs_point += 3
    elif order_2 in cost_ucs:
        ucs_point += 3
    elif order_2 in cost_ida:
        ida_point += 3
    # Find 3rd
    if order_3 in cost_as:
        as_point += 2
    elif order_3 in cost_dfs:
        dfs_point += 2
    elif order_3 in cost_ucs:
        ucs_point += 2
    elif order_3 in cost_ida:
        ida_point += 2
```

4.2. Visualize data to compare algorithms





Comparison of optimization among algorithms with different datasets

5. Conclusion and possible extensions:

5.1. Conclusion:

The solution is proved by the optimization and completion of the procedure. Route planning is a problem with a wide variety of practical applications. Using four methods, we may choose the best solution to the issue in the most straightforward way. Furthermore, while evaluating and running our code, the A* search approach is the most optimal when compared to IDA* and A* search.

5.2. Possible extensions:

Find a way to transport n goods through the warehouses for the most optimal cost and time.

6. List of tasks:

Task	Name	Contribution rate
SLIDE		
Idea and slide 2,6,8,9,10	Tran Vuong Quoc Dat	50%
Visualize, draw slide 1,3,4,6,7	Nguyen Truong Truong An	50%
ALGORITHM AND PROGRAMING		
A Star		
	Tran Vuong Quoc Dat	50%
	Le Duc Anh Tuan	50%
IDA		
	Nguyen Nho Trung	60%
	Nguyen Thanh Dat	40%
UCS		
	Tran Vuong Quoc Dat	60%

	Nguyen Truong Truong An	40%
DFS		
	Nguyen Truong Truong An	50%
	Nguyen Thanh Dat	50%
Problem modeling		
	Tran Vuong Quoc Dat	100%
Program Presentation		
Add comment, visualize and arrange programing	Le Duc Anh Tuan	100%
REPORT		
Introduction and modeling(1,2)	Tran Vuong Quoc Dat	30%
Algorithm analysis(3)	Nguyen Thanh Dat	35%
Result visualization and conclusion(4,5)	Le Duc Anh Tuan	35%
VIDEO		
	Nguyen Nho Trung	100%
PROGRAMMING AND TASK SYNCHRONIZATION		
Programing synchronization (on Github)	Le Duc Anh Tuan	100%
Assign task and task synchronization	Tran Vuong Quoc Dat	100%
DATA PROCESSING		

Collect and create data files	Nguyen Nho Trung	40%
Create a formula that transforms data	Tran Vuong Quoc Dat	20%
Convert raw data into usage data	Le Duc Anh Tuan	40%

7. References

GitHub: github.com/tuanlda78202/Logistics