SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

IT3160E - CAPSTONE PROJECT

Group 10 - Modified Route Planning

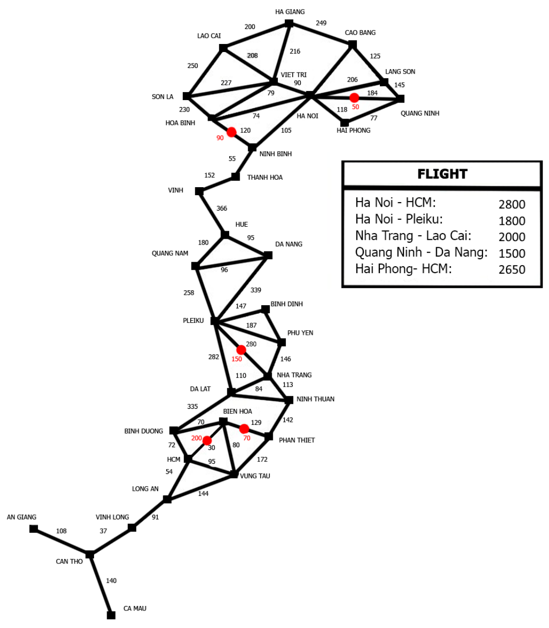
Authors:	Student ID:
Hoang Long Vu	20204897
Hoang Gia Nguyen	20204889
Nguyen Duc Quyet	20200520
Nguyen Duy Khanh	20204914
Nguyen Huu Tuan Duy	20204907

Dec 2021

Contents

1 Presentation of the subject

In this project, we need to write a program to find a path with minimum cost between two Vietnamese cities (e.g. Lao Cai and Ha Noi). The vehicle can only travel on road between two adjacent cities, or travel by plane if there exists an airway between two cities.



FLIGHT	
Ha Noi - HCM:	2800
Ha Noi - Pleiku:	1800
Nha Trang - Lao Cai:	2000
Quang Ninh - Da Nang:	1500
Hai Phong- HCM:	2650

Figure 1: Simplified map for the problem

The problem contains:

- 32 cities in total.
- 5 fixed air routes with pre-defined price.
- 10 fixed toll stations between two cities with pre-defined cost (marked red on the map above.

The data for all the programs are read from the following files:

- `route-info.csv` contains the distance and toll fee between adjacent city pairs. Concretely, the rows from 2 to 33 includes: <starting city><destination><distance><toll fee> (if there is no such toll on the trajectory, then the cost is 0).
- `air-info.csv` contains five rows with the flight cost between two cities: <starting city> <destination><flight cost>.
- The straight-line distance[1] between each city pair is stored in the file `heuristics.csv` as a 32 × 32 matrix.

- The label for each city in the `heuristics.csv` is stored in the `city-label.csv` file (indexing from 1 to 32).

The cost of travelling by road is calculated as: $roadCost = roadDistance \times 1,500$ (VND). The vehicle is charged after passing any toll stations on the roads.
The toll fee and flight cost is adjusted so that there should be no obvious bias for any choice (e.g. the flight cost is too expensive compared to taking the road, or the toll fee is not significantly expensive enough that can easily be ignored). When considering two arbitrary cities, the rule of thumb is as follows: $straightLineCost < flightCost < roadCost$ so that the algorithms are able to choose to take the flight if the cost is better, an illustration will be presented in the **Section 5, Pair 3**. Our objective is to *minimise* the amount of money needed to travel from the starting city to the destination.

Outputs:

- The path used to solve.
- Time complexity.
- Space complexity.
- The amount of money spent to reach the destination.

2 Problem Description

- Kind of problem: Search problem (branching factors: number of adjacent cities) where the environment is known, fully observable, and deterministic.
- States: The locations of the cities.
- Initial state: Any randomised starting city location.
- Action: Go from one city to another.
- Transition model: Successor location after the latest action.
- Goal test: Determine whether the current state is the destination.
- Path cost: The total amount of money spent to travel between two cities.

3 Algorithms for problem solving

3.1 Uniformed Search

Uniform-cost Search (*UCS*). In this problem, the step-cost is not fixed between every adjacent cities. Therefore, Uniform-cost Search should be a promising candidate to solve this problem.

Algorithm 1: Uniform-cost search

```
path_cost ← $distance * 1500 + toll$
function UNIFORM-COST-SEARCH(start(node), goal(node))
returns a solution, or failure
start(node) ←
 a node with start(node).name = name, start(node).parent = None, path_cost = 0
frontier ← a priority queue ordered by path_cost, with Node as the only element
visited ← an empty set
while loop do
    if EMPTY?(frontier) then
        node ← POP(frontier)
    if node is goal then
        returns SOLUTION(node)
    add node to visited
    for each child in successor(node) do
        path_cost += path_cost(node) if
         child is not in visited or child is not in frontier then
            frontier ← $INSERT(child, frontier)$else
            child is in frontier with higher path_cost replace that frontier node
            with node
```

3.2 Informed Search

Heuristics choice: We decided to choose the straight-line distance (*SLD*) to be our heuristics for the Informed Search algorithms. This heuristic is admissible as there exists no path that is shorter than the straight one. Moreover, the general triangle inequality is satisfied when each side is measured by the straight-line distance, thus our SLD heuristic is also consistent.[2]

1. **Greedy Best-first search (*GBFS*)**. This algorithm expands the node that is estimated to be closest go goal, with good time complexity if guided by a good heuristic.

Algorithm 2: Greedy Best-first search

```
function GREEDY-BEST-FIRST-SEARCH(start(node), goal(node))
returns a solution, or failure
start(node) ←
  a node with start(node).name = name, start(node).parent = None, path_cost = 0
if start(node) is stop(node) then
    return empty path to stop(node)
open ←
  a priority queue ordered by Distance(h(n)), with start(node) as the only element
closed ← an empty priority queue ordered by Distance(h(n))
while loop do
    if EMPTY?(open) then
        return failure
    parent(node) ← POP(open)
    if parent(node) is stop(node then
        return SOLUTION(node) is parent(node)
    add parent(node) to closed
    for (child, cost) in successors(parent(node)) do
        g ← g[parent][child] + cost
        h ← h[child]
        child(node) ← Node(name, h, g)
        if child is not in closed or child is not in open then
            child(node).parent = parent
            add child(node) to open, the priority node will be placed to the front
```

2. **A* Search**. Even though GBFS is complete (in our finite search space), it is not optimal. Therefore, we wanted to see more improvements by trying implementing A* Search. A* is one of the most popular choices for path finding problems. If combined with the given SLD heuristics, A* would assure the optimality and completeness. Furthermore, the branching factor and the sample space of this problem is not considerably large, thus we believe A* would be the most promising algorithm to solve the problem.

Algorithm 3: A* search

```
distance, tollCost ← loaded from route-info.csv
function A-STAR-SEARCH(start(node), goal(node))
returns a solution, or failure
start(node) ← a node with start(node).name=name, start(node).parent=None
evaluationCode = heuristic(start, goal)
frontier
 ← a priority queue ordered by evaluationCost, with Node as the only element
visited ← an empty set
flyCost ← flyCost[start][goal]
evaluationCost
 ← distance[node][child] + heuristic[node][child] + tollPrice[node][child]
while loop do
    if EMPTY?(frontier) then
        return failure
    node ← POP(frontier
    if node is goal then
        if evaluationCode ≥ flyCost then
            return flyCost
        else
        SOLUTION(node)
    add node to visited
    for each child in successor(node) do
        evaluationCost = distance[node][child] + toll[node][child] +
         heuristic[node][child]
        if child is not in vistied or child is not in frontier then
            frontier ← INSERT(child, frontier)else
            if child is in frontier with higher evaluationCost then
                replace that frontier node with child
```

3. **Iterative Deepening A* (*IDA**) Search**. Similar to A* search, IDA* is complete and optimal under the SLD heuristic. The main advantage of IDA* is that this algorithm has significantly smaller space complexity compared to the former A* algorithm. However, IDA* suffers from regenerating the same nodes multiple times, thus it is expected to have poorer time complexity compared to A* Search.

Algorithm 4: IDA* search

```
distance ← loaded from route-info.csv
function IDA(start(node), goal(node), heuristic(function), threshold)
frontier ← stack with start node as the only element
while length(frontier) > 0 do
    node, cost ← POP(frontier)
    if node is goal then
        return node
    for each node in successor(node) do
        pathCost = cost + distance(node, nextNode) + heuristic(nextNode)
        if pathCost > threshold then
            frontier.PUSH(nextNode, pathCost - heuristic(nextNode))
        threshold ← min value in frontier
```

4 Algorithms implementation

Firstly, we chose to randomise the input and output of the problem. Therefore, we need to fetch the straight-line distance between every city pair which seems to be an endless process of search-and-fill. We also need to run around 50 instances to test if there are any mistakes in our heuristics data, which is another time-consuming task. However, we are quite satisfied with our heuristics as it is consistent and admissible, thus our A* and IDA* will always be optimal and complete.
We tried implementing IDA* algorithm using recursion at first. However, it became really troublesome when we wanted to print the final path, thus we decided to switch to stack implementation to overcome this problem.

5 Results

Some predictions and expectations (where b is the branching factor, d is the depth of the shallowest solution and m is the maximum depth of the search tree):

- Uniform-cost Search is optimal and complete under our finite state space in this problem. If we let $C*$ be the cost of the optimal solution, and assume that every action costs at least ϵ, then the space and time complexity of uniform-cost search is both $O(b^{1+\lfloor \frac{C*}{\epsilon} \rfloor})$.
- Being guided by a consistent and admissible *SLD* heuristics, IDA* and A* Search algorithms are also complete and optimal. The space and time complexity of A* are both exponential with path length; whereas it is $O(bd)$ and $O(b^d)$, respectively, for IDA* algorithm.
- Greedy Best-first search is not optimal, but still complete in our finite search space. The space and time complexity are both $O(b^m)$, however it is guided by the *SLD* heuristic, thus we believe that the time complexity is much better.

In order to test the performance and to characterise the properties of our algorithms, we decided to pick the four most "interesting" city pairs (the sample space is relatively small, thus we believe that other pairs of cities will just fall into the same category of the four chosen pairs). Concretely:

- **Pair 1: LaoCai - CaMau**
These two cities lie on two "endpoints" in Vietnam map and has the largest straight-line distance, thus we believe this would be an interesting pair to try out.

Algorithm	Path	Cost	Time	Space
UCS	LaoCai, NhaTrang, DaLat, BinhDuong, HoChiMinh, LongAn, VinhLong, CanTho, CaMau	3258500	0.000808000564	9
A*	LaoCai, NhaTrang, NinhThuan, PhanThiet, VungTau, LongAn, VinhLong, CanTho, CaMau	3258500	0.001030921936	104
IDA*	LaoCai, NhaTrang, DaLat, BinhDuong, HoChiMinh, LongAn, VinhLong, CanTho, CaMau	3258500	19.30786705	1546609
GBFS	LaoCai, NhaTrang, DaLat, BinhDuong, HoChiMinh, LongAn, VinhLong, CanTho, CaMau	3258500	0.0012228488922	9

Comments

  - All four algorithms succeeded in finding the optimal solution. The time and space complexity of UCS is quite good. GBFS expands the same number of nodes as UCS, but the running time is significantly larger.
  - Despite managing to find the optimal cost, IDA* expanded a huge number of nodes and suffer from a long running time. There are many intermediary nodes between LaoCai and CaMau, thus IDA* needs to regenerate a multitude of nodes and leads to a poor performance.

- **Pair 2: Pleiku - HoChiMinh**
We put a lot of toll stations on the paths between these two cities. Therefore, we want to see if our algorithms could succeed in "avoiding" these stations to minimise the total travelling cost.

Algorithm	Path	Cost	Time	Space
UCS	Pleiku, DaLat, BinhDuong, HoChiMinh	1033500	0.00048494338989	4
A*	Pleiku, DaLat, BinhDuong, HoChiMinh	1033500	0.00089311599731	30
IDA*	Pleiku, DaLat, BinhDuong, HoChiMinh	1033500	0.00867319107055	366
GBFS	Pleiku, DaLat, BinhDuong, HoChiMinh	1033500	0.0006217956542	4

Comments

  - All four algorithms succeeded in "avoiding" the tolls stations located on a short path ($Pleiku \to NhaTrang \to NinhThuan \to PhanThiet \to BienHoa \to HoChiMinh$) and found the optimal solution.
  - UCS, GBFS and A* still has an exceptional performance, and the IDA* algorithm has seen some dramatic improvements in space and time complexity as the state space is much smaller now.

- **Pair 3: HaiPhong - VungTau**
In the problem formulation, we specified a flight path between *HaiPhong* and *HoChiMinh* (which locates between *HaiPhong* and *VungTau*). The flight cost was adjusted

so that a “good” algorithm should take a flight from *HaiPhong* to *HoChiMinh* first, then travel to *Vung Tau* on road rather than going from *HaiPhong* to *VungTau* with just the road path.

Algorithm	Path	Cost	Time	Space
UCS	HaiPhong, HoChiMinh, VungTau	2792500	0.00076317787170	3
A*	HaiPhong, HoChiMinh, VungTau	2792500	0.00070405006408	64
IDA*	HaiPhong, HoChiMinh, VungTau	2792500	7.2701029777526855	973439
GBFS	HaiPhong, HoChiMinh, VungTau	2792500	0.00079584121704	3

Comments

- All four algorithms were "smart" enough to minimise the cost by taking an air route from *HaiPhong* to *HoChiMinh*. Despite generating a little bit more nodes than UCS, A* algorithm has gain its advantage against UCS and GBFS in terms of time complexity.
- On the other hand, IDA* suffered from poor time and space complexity even though this algorithm managed to find the optimal cost.

- **Pair 4: SonLa - Quang Ninh**
  There is a lot of paths connecting these two cities, and there are two toll stations lie on those paths. We want to see which algorithms will manage to find the best path.

Algorithm	Path	Cost	Time	Space
UCS	SonLa, HoaBinh, HaNoi, HaiPhong, QuangNinh	748500	0.0004999637603759766	5
A*	SonLa, HoaBinh, HaNoi, HaiPhong, QuangNinh	748500	0.00067901611328125	22
IDA*	SonLa, HoaBinh, HaNoi, HaiPhong, QuangNinh	748500	0.0012371540069580078	251
GBFS	SonLa, VietTri, Hanoi, QuangNinh	786500	0.00100517272949	4

Comments

- UCS, A*, and IDA* algorithms succeeded in finding the optimal solution in quite good space and time complexities.
- UCS is still the most effective one, with an exceptionally good space complexity of 5, and a slightly better running time compared to A* Search.
- On the other hand, this is the case where GBFS has lost its accuracy. Due to the property of being "greedy", this algorithm took the shortest path in the first place (going to *VietTri*) but failed to optimise the final cost, thus did not find the optimal path to the problem.

- **Pair 5: QuangNinh - NhaTrang**
  After travelling on road from *QuangNinh* to *DaNang* (which is intended to be the optimal path to travel between these two cities, rather than the air route like in the **Pair 3**), our algorithms should avoid the toll station between *Pleiku* and *NhaTrang* to optimise the final cost.

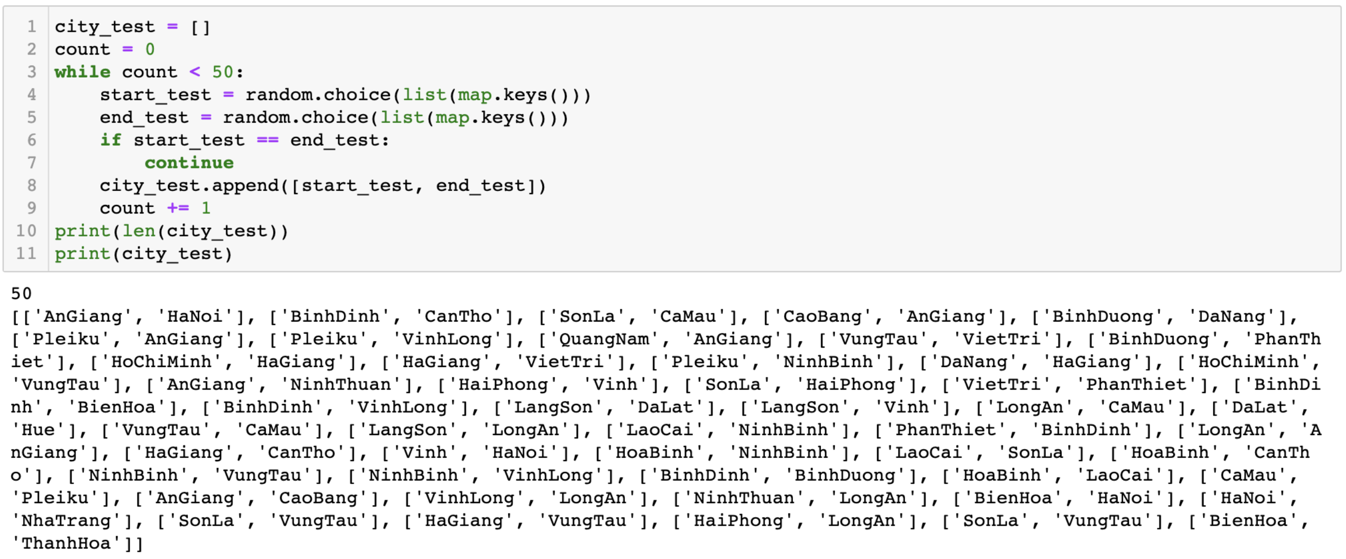
Algorithm	Path	Cost	Time	Space
UCS	QuangNinh, HaiPhong, HaNoi, NinhBinh, ThanhHoa, Vinh, Hue, DaNang, Pleiku, PhuYen, NhaTrang	2458500	0.0016965866088867188	11
A*	QuangNinh, HaiPhong, HaNoi, NinhBinh, ThanhHoa, Vinh, Hue, DaNang, Pleiku, PhuYen, NhaTrang	2458500	0.0013461112976074219	62
IDA*	QuangNinh, HaiPhong, HaNoi, NinhBinh, ThanhHoa, Vinh, Hue, DaNang, Pleiku, PhuYen, NhaTrang	2458500	0.098621129989	45026
GBFS	QuangNinh, DaNang, Pleiku, NhaTrang	2678500	0.00104117393493	4

Comments

- Our algorithms did well in finding the optimal solution with acceptable time and space complexity. IDA* has also seen a great improvement in the running time as the state space is significantly smaller in this case.
- Meanwhile, GBFS failed to avoid the flight from *QuangNinh* to *DaNang*, thus could not find the optimal solution in this case.

Time and space complexity comparison

To compare the complexity in terms on time and space, we generated 50 random city pairs for our algorithms. However, IDA* has poor time complexity and takes up to hours to complete. Therefore, we will just compare the performance of three algorithms: Uniform-cost search, A* search, and Greedy Best-first search.

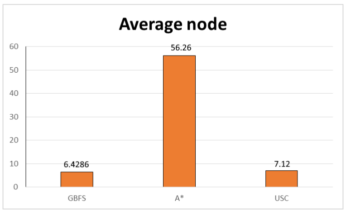


```python
city_test = []
count = 0
while count < 50:
    start_test = random.choice(list(map.keys()))
    end_test = random.choice(list(map.keys()))
    if start_test == end_test:
        continue
    city_test.append([start_test, end_test])
    count += 1
print(len(city_test))
print(city_test)
```

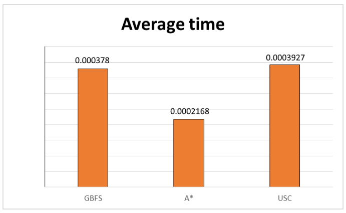
```
50
[['AnGiang', 'HaNoi'], ['BinhDinh', 'CanTho'], ['SonLa', 'CaMau'], ['CaoBang', 'AnGiang'], ['BinhDuong', 'DaNang'],
['Pleiku', 'AnGiang'], ['Pleiku', 'VinhLong'], ['QuangNam', 'AnGiang'], ['VungTau', 'VietTri'], ['BinhDuong', 'PhanTh
iet'], ['HoChiMinh', 'HaGiang'], ['HaGiang', 'VietTri'], ['Pleiku', 'NinhBinh'], ['DaNang', 'HaGiang'], ['HoChiMinh',
'VungTau'], ['AnGiang', 'NinhThuan'], ['HaiPhong', 'Vinh'], ['SonLa', 'HaiPhong'], ['VietTri', 'PhanThiet'], ['BinhDi
nh', 'BienHoa'], ['BinhDinh', 'VinhLong'], ['LangSon', 'DaLat'], ['LangSon', 'Vinh'], ['LongAn', 'CaMau'], ['DaLat',
'Hue'], ['VungTau', 'CaMau'], ['LangSon', 'LongAn'], ['LaoCai', 'NinhBinh'], ['PhanThiet', 'BinhDinh'], ['LongAn', 'A
nGiang'], ['HaGiang', 'CanTho'], ['Vinh', 'HaNoi'], ['HoaBinh', 'NinhBinh'], ['LaoCai', 'SonLa'], ['HoaBinh', 'CanTh
o'], ['NinhBinh', 'VungTau'], ['NinhBinh', 'VinhLong'], ['BinhDinh', 'BinhDuong'], ['HoaBinh', 'LaoCai'], ['CaMau',
'Pleiku'], ['AnGiang', 'CaoBang'], ['VinhLong', 'LongAn'], ['NinhThuan', 'LongAn'], ['BienHoa', 'HaNoi'], ['HaNoi',
'NhaTrang'], ['SonLa', 'VungTau'], ['HaGiang', 'VungTau'], ['HaiPhong', 'LongAn'], ['SonLa', 'VungTau'], ['BienHoa',
'ThanhHoa']]
```

Figure 2: Python code and results for random sample generation

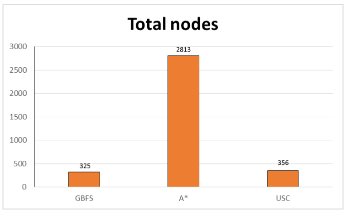
For better correlation, all the three algorithms will be tested on the same 50 instances above. The results are averaged over those 50 instances and illustrated in the following charts:



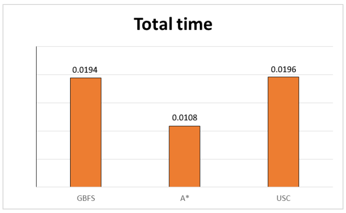
(a) Average nodes expanded



(b) Average running time



(c) Total nodes expanded



(d) Total running time

Figure 3: Time and space complexity comparison

At first glance, it can be seen that Uniform-cost search and Greedy Best-first search bear significant resemblance in terms of space and time complexity. Meanwhile, A* Search has a relatively large space complexity, but also has the best running time in average, and in total, amongst the three algorithms.

6 Conclusion and possible extensions

Conclusions:

- Uniform-cost Search and A* search are the best candidates to solve our problem. Both algorithms are complete and optimal, A* has a slightly better running time in average (as shown in the charts above), but space complexity is a drawback compared to UCS. UCS has seen an outstanding space and time complexity in general.
- IDA* is fairly good, however it tends to expand the same nodes multiple times and suffers from poor time complexity, especially for large state spaces.
- GBFS has a really stable and good space complexity, and the time complexity is even better if guided by the *SLD* heuristic. However, this algorithm could not assure the optimality in this Route Planning problem.

Route planning is a really interesting and practical problem, and we are glad to have chosen this topic for our Capstone Project. We had a chance to practice with the most basic and common algorithms used in AI and path searching; had a chance to interact and corporate with new friends. Moreover, the process of searching and filling the straight-line distance of every city pair took so long that we may even remember all the positions and distances of them.
If we had more time to complete the Capstone Project, we wish to expand our problem to the more practical ones, e.g. include about 77 cities in our problem (which is quite close to the reality), or add more flights between cities (five is actually a modest number). Moreover, we wish to visualise our path with some tools (e.g. *turtle* in Python) for more attractive presentation if we could have more time.

By experiments, we yield an assumption that: *In any case (for our problem) the GBFS algorithm is optimal, the number of nodes it expanded will equal to that number of UCS.* However, we could not find a way and did not have enough time to prove this assumption.

7 List of tasks

We wanted every member to understand the algorithms, and members should be able to discuss, interact with each other and construct the complete program based on other member's ideas. Therefore, we decided that a member will program the complete Python code based on another member's pseudo code. Specifically:
Programming tasks:

- Nguyen Duc Quyet: Proposed and programmed: Uniform-cost Search and
- Nguyen Duy Khanh: Proposed and programmed: A* and IDA* Search

Analytic tasks:

- Writing the pseudo code for algorithms:
  - Uniform-cost Search: Hoang Long Vu
  - Greedy Best-first Search and IDA* Search: Hoang Gia Nguyen
  - A* Search: Nguyen Huu Tuan Duy
- Drawing the problem's map: Nguyen Huu Tuan Duy
- Constructing the data for heuristic: Nguyen Huu Tuan Duy (33%) + Hoang Gia Nguyen (33%) + Hoang Long Vu (33%)
- Analysing and visualising the running time of the algorithms: Hoang Long Vu (80%) + Nguyen Huu Tuan Duy (20%)
- Running the demo video: Nguyen Duy Khanh
- Running random instances of the problem: Nguyen Duc Quyet (50%) + Nguyen Duy Khanh (50%)
- Proposing the most interesting city pairs for testing: Nguyen Duc Quyet (30%) + Hoang Long Vu (70%)
- Writing the report: Hoang Long Vu
- Preparing slides: Hoang Gia Nguyen

References

[1] https://www.distancefromto.net/. (Online; accessed 25-Nov-2021).

[2] Stuart Russell Peter Norvig. *Artificial Intelligence: A Modern Approach - Third Edition*, chapter 3.