**IT3160E INTRODUCTION TO ARTIFICIAL INTELLIGENCE**

**CAPSTONE PROJECT**

**Class: 131117**

**Lecturer: Than Quang Khoat**

1. **GROUP INFO**

|  |  |  |
| --- | --- | --- |
| **Name** | **Student ID** | **Task** |
| Nguyen Khanh Trung | 20205133 | * Team Management * Implementing GBFS * Data Analysis * Writing Report |
| Nguyen Phuong Quang | 20205191 | * Researching and importing Data * Implementing UCS |
| Hoang Van Phuong | 20200478 | * Generating Test Case and Exporting Data * Visualization * Making SlideShow |
| Bui Van Thanh | 20200585 | * Implementing A\* * Data Analysis |

1. **PROBLEM DESCRIPTION**

Problem: Route Planning

*Overview*

We’re a writing a program to find the shortest route between two Vietnamese cities (e.g. Hanoi and Hai Phong). The intelligent vehicle can only travel between 2 adjacent cities, and the objective is to minimize the number of kms between two cities. In this project, we are only considering **the northern side of Vietnam** as the dataset for simplicity (As we are extracting the data from the <https://en.wikipedia.org/wiki/Main_Page> site manually).

*Approach*

We’re writing the program to solve the problem through implement three different appropriate search algorithms: Uniform-cost Search (UCS), Greedy Best First Search (GBFS) and A\* search with the heurist function Estimated straight-line distance from n to the end\_city.

The program will have several outputs for each of the search algorithms:

* Time complexity (number of nodes expanded in order to solve the route planning problem)
* Space complexity (number of nodes kept in memory)
* The path used to solve the route planning problem (solution) if there was a solution
* The cumulated number of km of the solution (if any)

There’ll also be deep analysis and comparison between the three algorithms and visualization for the solution.

1. **DETAILS**

*Input*

The city map and heuristic distance will then be read from two json files: neighbor.json and sld.json, having the following format:

File neighbor.json

{

City\_1: {

Neighbor\_City\_1: Distance\_1,

Neighbor\_City\_2: Distance\_2,

…

},

City\_2: {

Neighbor\_City\_1: Distance\_1,

Neighbor\_City\_2: Distance\_2,

…

},

…

}

File sld.json

{

City\_1: {

Neighbor\_City\_1: Heuristic\_Distance\_1,

Neighbor\_City\_2: Heuristic\_Distance\_2,

…

},

City\_2: {

Neighbor\_City\_1: Heuristic\_Distance\_1,

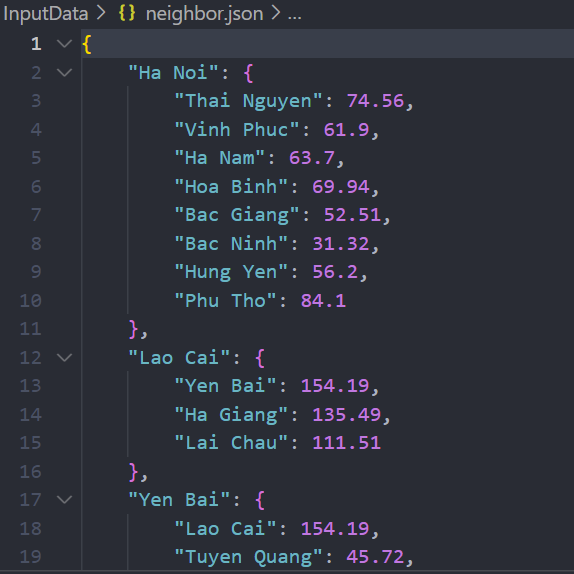
Neighbor\_City\_2: Heuristic\_Distance\_2,

…

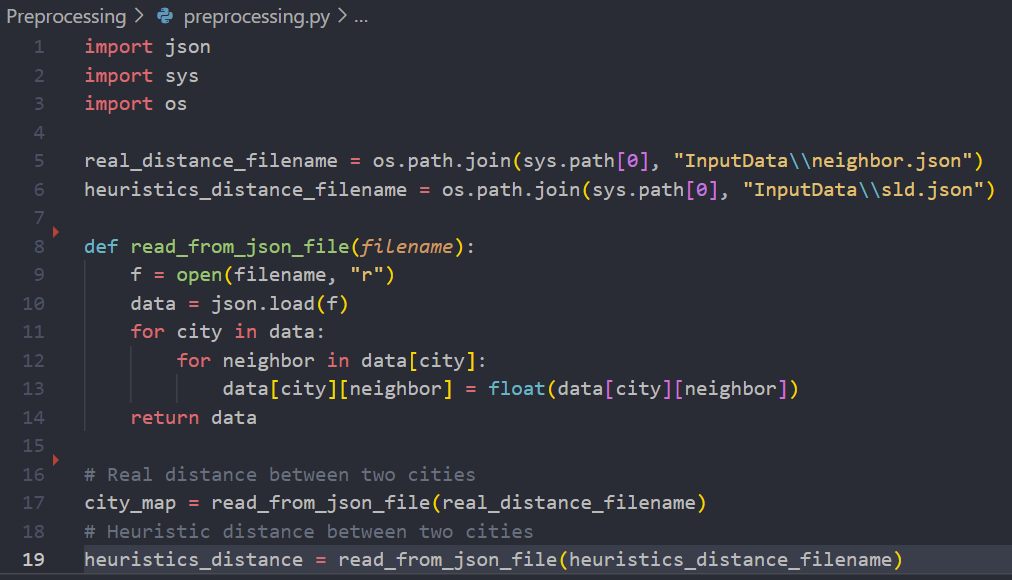
},

…

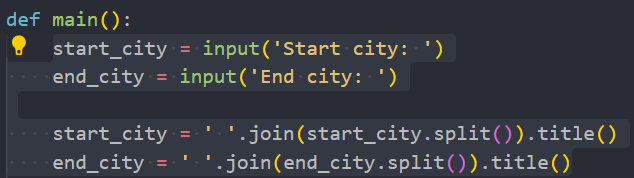
}



Data from the two files will be read by the preprocessing.py file containing the ***json*** library into 2 variables: *city\_map* and *heuristics\_distance* (which will now be nested dictionary)



The starting city and ending city will be input from the keyboard in main.py file and stored in 2 variable *start\_city* and *end\_city*.



*Output*

The program will have several outputs:

* Time complexity (number of nodes expanded in order to solve the route planning problem)
* Space complexity (number of nodes kept in memory)
* The path used to solve the route planning problem (solution) if there was a solution
* The cumulated number of km of the solution (if any)

*Algorithms*

**Uniform-Cost Search (UCS)**

Note: Using Priority Queue





<https://www.educative.io/answers/what-is-the-python-priority-queue>

Main component:

The UCS(start\_city, end\_city, city\_map) function:

Initialize:

time\_complextity = 0

space\_complextity = 0

Queue = Q.PriorityQueue(); (Create new priority queue)

Queue.put((0, [start\_city])) (insert first element in queue)

For each "while" loop the path expanded by the UCS algorithm will be retrieved and considered:

Each element in queue is a tuple of the form (cost, path) while path is an array containing the path.

Example: (390, [ ‘Ha Noi’, ‘Phu Tho’, ‘Yen Bai’, ‘Lai Chau’]

* “ node = queue.get() “ will get the element that have the minimal cost.

Because when you using method get() with priority queue, the value returned is a element with the minimal cost;

Example:

Graphical user interface

Description automatically generated

* “ current = node[1][len(node[1]) – 1]” gets the last city in the array containing the path with the minimum cost under consideration and “ cost = node[0] “ gets that cost (Note: In this step we will be using a variable named *time\_complexity*  to keep track of the number of nodes generated and stored for data analysis later on).
* Check if the end\_city is in the array containing the path with the minimum cost under consideration, then that is the path to find and program ends, outputs the solution.
* In the “for” loop we will generate all the neighbor of the current city end store all of them in queue. (Note: In this step we will be using a variable named *space\_complexity* to keep track of maximum number of elements stored in a node and stored for data analysis later).

**Greedy Best First Search (GBFS)**

Note: External Libraby: HeapDict (Need to be installed)

Installation and Documentation:

<https://pypi.org/project/HeapDict/>

<https://www.geeksforgeeks.org/priority-queue-using-queue-and-heapdict-module-in-python/?ref=rp>

Brief Explanation:

Heapdict implements the MutableMapping ABC, meaning it works pretty much like a regular Python dictionary. It’s designed to be used as a priority queue (The value in the key-value pair will be treated as the priority of the pair in the heapdict).

Main Component:

The GBFS(*start\_city, end\_city, city\_map, heuristics\_distance*) function:

Initialize:

*cur\_city = start\_city*

*time\_space = 1*

*visited = {start\_city : None}* (A dictionary with the key-value pair is a city and its parent city for traceback the actual path later on)

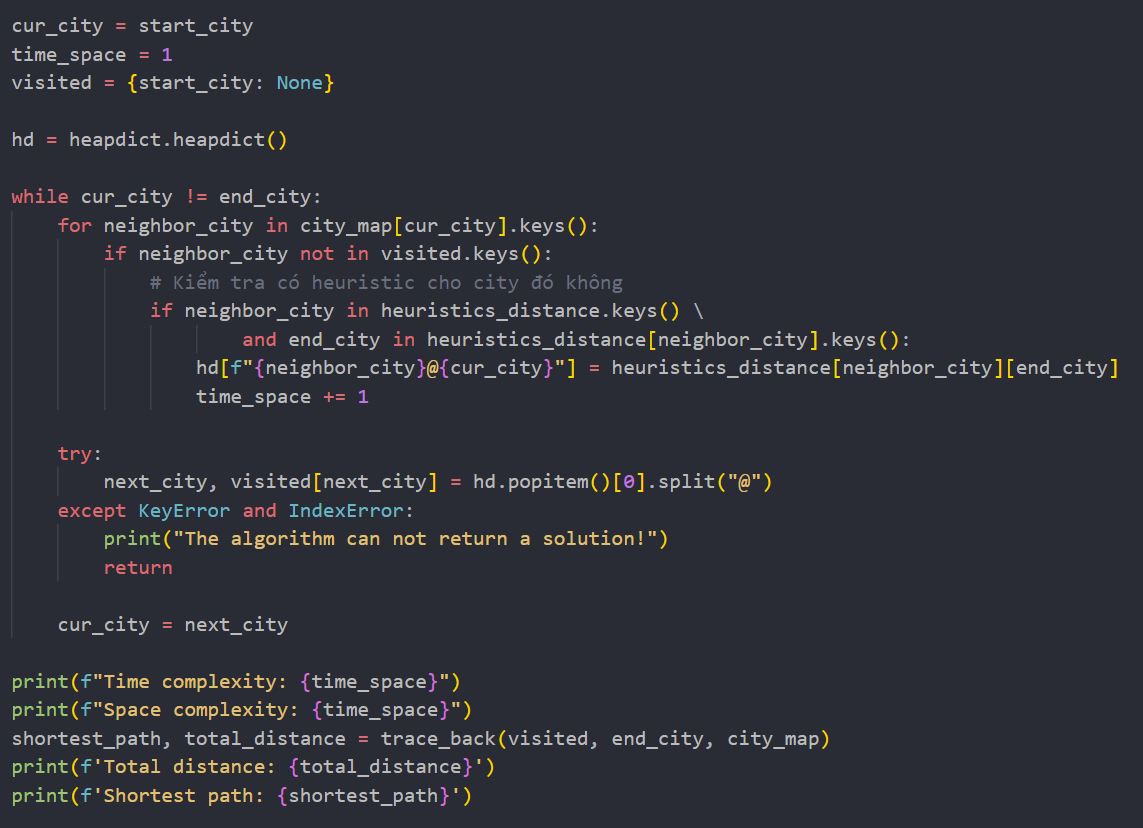
*hd = heapdict.heapdict()* (Create a heapdict object)

For each iteration till the solution is found or the algorithm get stuck in a loop:

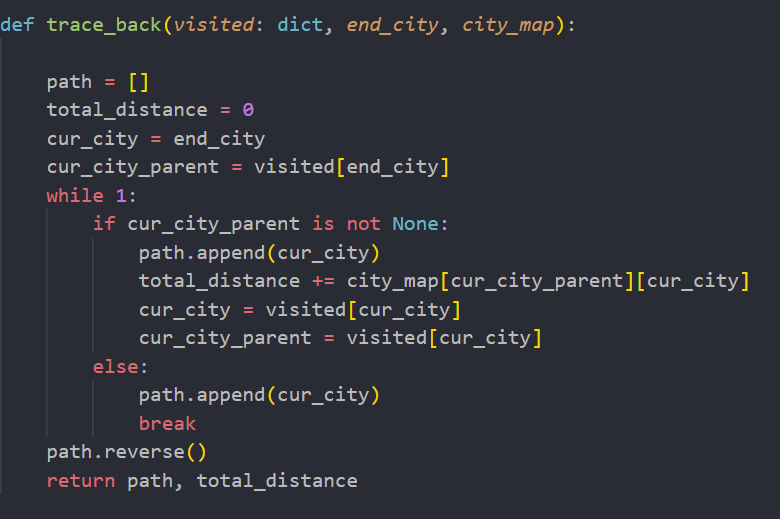
* Starting from the current city (*cur\_city*) generate all of its neighbor cities and store them, their parent city and their respective heuristics distance into the heapdict. (Note: In this step we will be using a variable named *time\_space* to keep track of the number of nodes generated and stored for data analysis later on).
* Check if the heapdict is not empty: pop the top priority item out of the heapdict and store it in a variable named *next\_city* (This will be the *cur\_city* for the next iteration) and update the *visited* dictionary with *next\_city* and its parent city. If the heapdict is empty, this means the algorithm get stuck in a loop 🡪 Return and Output
* Update *cur\_city = next\_city* and start next iteration.

Trace back the actual path and path cost with the TraceBack() function

Output the solution



The TraceBack(*visited, end\_city, city\_map*) function:



**A\* Search**

*Visualization*

*Data Analysis*

*Applications*

Optimized Travelling, Delivery and many more.