**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 * Making SlideShow |
| Nguyen Phuong Quang | 20205191 | * Researching and Collecting Data * Implementing UCS |
| Hoang Van Phuong | 20200478 | * Generating Test Case and Exporting Data * Visualization |
| Bui Van Thanh | 20200585 | * Importing Data * 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,

…

},

…

}

{

    "Ha Noi": {

        "Thai Nguyen": 74.56,

        "Vinh Phuc": 61.9,

        "Ha Nam": 63.7,

        "Hoa Binh": 69.94,

        "Bac Giang": 52.51,

        "Bac Ninh": 31.32,

        "Hung Yen": 56.2,

        "Phu Tho": 84.1

    },

    "Lao Cai": {

        "Yen Bai": 154.19,

        "Ha Giang": 135.49,

        "Lai Chau": 111.51

    },

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)

import json

import sys

import os

real\_distance\_filename = os.path.join(sys.path[0], "InputData\\neighbor.json")

heuristics\_distance\_filename = os.path.join(sys.path[0], "InputData\\sld.json")

def read\_from\_json\_file(*filename*):

    f = open(filename, "r")

    data = json.load(f)

    for city in data:

        for neighbor in data[city]:

            data[city][neighbor] = float(data[city][neighbor])

    return data

# Real distance between two cities

city\_map = read\_from\_json\_file(real\_distance\_filename)

# Heuristic distance between two cities

heuristics\_distance = read\_from\_json\_file(heuristics\_distance\_filename)

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*.

def main():

    start\_city = input('Start city: ')

    end\_city = input('End city: ')

    start\_city = ' '.join(start\_city.split()).title()

    end\_city = ' '.join(end\_city.split()).title()

*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)

*Algorithm Implementation*

**Uniform-Cost Search (UCS)**

Note: Using Priority Queue (Do not need to be installed)





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

Main component:

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

Initialize:

*time = 1*

*space = 1*

*cur\_city = start\_city*

*cities\_list = [start\_city]*

*cost = 0*

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

Each element in the queue will be a tuple of the form (*cost, path*) with *path* is a list containing the cities of the path.

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

*Queue.put((0, cities\_list))* (Insert the first element into queue)

For each iteration till the queue is empty, the path expanded by the UCS algorithm will be retrieved and considered:

* Starting from the current city (*cur\_city*) generate all of its unvisited neighbor cities, append them into their corresponding *cities\_list* path and recalculate the path cost (*cost*). Put the (*cost, cities\_list)* tuple into the PriorityQueue (Note: In this step we will be using two variables named *time* and *space* to keep track of the number of nodes generated and stored for data analysis later on).
* Pop the path having the minimum cost out of the PriorityQueue, store its corresponding cities list, cost and last city into variables named *cities\_list, cost, last\_city* respectively.
* Check if *end\_city* == *last\_city*, if condition satisfied, break out of the loop and output the solution. Else, update the *visited* list, *cur\_city* and start the next iteration.

def UCS(*start\_city*, *end\_city*, *city\_map*):

    #Input handling

    if start\_city not in city\_map:

        raise TypeError(str(start\_city) + ' not found in graph !')

        return

    if end\_city not in city\_map:

        raise TypeError(str(end\_city) + ' not found in graph !')

        return

    if(start\_city == end\_city):

        print('Total distance: 0')

        print('Best route: ')

        return

    #Init

    time = 1

    space = 1

    cur\_city = start\_city

    cities\_list = [start\_city]

    cost = 0

    visited = [start\_city]

    # Initial queue

    queue = Q.PriorityQueue()

    queue.put((0, cities\_list))

    #UCS

    while not queue.empty():

        for neighbor in city\_map[cur\_city]:

            if neighbor not in visited:

                temp = cities\_list[:]

                temp.append(neighbor)

                time += 1

                space += 1

                queue.put((cost + city\_map[cur\_city][neighbor], temp))

        # Pop the top priority item out of the PriorityQueue

        node = queue.get()

        space -= 1

        # Get the cost, cities\_list and last\_city

        cost = node[0]

        cities\_list = node[1]

        last\_city = cities\_list[len(cities\_list) - 1]

        if end\_city == last\_city:

            break

        #Update cur\_city

        visited.append(last\_city)

        space += 1

        cur\_city = last\_city

    print(f"Time complexity: {time}")

    print(f"Space complexity: {space}")

    print(f'Total distance: {cost}')

    print(f'Shortest path: {cities\_list}')

    printMap(cities\_list)

**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 = 1*

*space = 1*

*visited = {start\_city : None}* (A dictionary with the key-value pair is a city and its parent city for tracing back 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 two variables named *time* and *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

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

    #Input handling

    if start\_city not in heuristics\_distance.keys():

        print('Can not find the start city. Please select a start city again.')

        return

    elif end\_city not in heuristics\_distance.keys():

        print('Can not find the end city. Please select an end city again.')

        return

    if start\_city == end\_city:

        print('Total distance: 0')

        print('Best route: ')

        return

    #Init

    cur\_city = start\_city

    time = 1

    space = 1

    visited = {start\_city: None}

    hd = heapdict.heapdict()

    #GBFS

    while cur\_city != end\_city:

        #Generate unvisited neighbor cities

        for neighbor\_city in city\_map[cur\_city].keys():

            if neighbor\_city not in visited.keys():

                hd[f"{neighbor\_city}@{cur\_city}"] = heuristics\_distance[neighbor\_city][end\_city]

                time += 1

                space += 1

        # Pop the top priority item out of the Heapdict

        try:

            next\_city, visited[next\_city] = hd.popitem()[0].split("@")

        except KeyError and IndexError:

            print("The algorithm can not return a solution!")

            return

        #Update cur\_city

        cur\_city = next\_city

    shortest\_path, total\_distance = trace\_back(visited, end\_city, city\_map)

    print(f"Time complexity: {time}")

    print(f"Space complexity: {space}")

    print(f'Total distance: {total\_distance}')

    print(f'Shortest path: {shortest\_path}')

    printMap(shortest\_path)

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

def trace\_back(*visited*: dict, *end\_city*, *city\_map*):

    path = []

    total\_distance = 0

    cur\_city = end\_city

    cur\_city\_parent = visited[end\_city]

    while 1:

        if cur\_city\_parent is not None:

            path.append(cur\_city)

            total\_distance += city\_map[cur\_city\_parent][cur\_city]

            cur\_city = visited[cur\_city]

            cur\_city\_parent = visited[cur\_city]

        else:

            path.append(cur\_city)

            break

    path.reverse()

    return path, total\_distance

**A\* Search**

Main Component:

The A\_star\_algorithm(*start\_city, end\_city, city\_map, heuristics\_distance*) function:

Initialize:

*min\_cost\_value = 1e9 (a value for checking for optimal node)*

*visited = [start\_city] (a list of generated nodes)*

*optimal\_node = [start\_city] (a list of optimal nodes, no longer need to be generated)*

*cur\_city = start\_city*

*cur\_cost = {cur\_city : 0} (a dictionary storing cities and the current cost to reach those cities)*

*route = {cur\_city : None} (a dictionary storing cities and their respective parent cities)*

*f ={end\_city:1e9+1} (a dictionary storing cities and their evaluation)*

*f[cur\_city] = heuristics\_distance[cur\_city][end\_city] + cur\_cost[cur\_city]*

*time\_space = 1*

For each iteration till the solution is found:

* Starting from the current city (*cur\_city*) generate all of it neighbors which are not optimal, update the evaluation and route of all the city of the *visited* list. (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).
* Add the *cur\_city* to *optimal* list, remove it from *visited* list, and find out a new *cur\_city*: the city has the smallest evaluation.

Traceback the optimal path with the TraceBack function and Output the solution.

# initialize values

    min\_cost\_value = 1e9

    visited = [start\_city]

    optimal\_node = [start\_city]

    cur\_city = start\_city

    cur\_cost = {cur\_city : 0}

    route = {cur\_city : None}

    f ={end\_city:1e9+1}

    f[cur\_city] = heuristics\_distance[cur\_city][end\_city] + cur\_cost[cur\_city]

    time\_space = 1

    # find the best route and the min\_distance

    while f[end\_city] != min\_cost\_value:

        for neighbor\_city in city\_map[cur\_city].keys():

            # Check unvisited cities for optimal\_node

            if neighbor\_city not in optimal\_node:

                temp\_cur\_cost = cur\_cost[cur\_city] + city\_map[cur\_city][neighbor\_city]

                if neighbor\_city not in f.keys() or f[neighbor\_city] > temp\_cur\_cost + heuristics\_distance[neighbor\_city][end\_city]:

                    cur\_cost[neighbor\_city] = temp\_cur\_cost

                    f[neighbor\_city] = cur\_cost[neighbor\_city] + heuristics\_distance[neighbor\_city][end\_city]

                    route[neighbor\_city] = cur\_city

                if neighbor\_city not in visited:

                    visited.append(neighbor\_city)

                time\_space = time\_space + 1

        #Insert the optimal node

        optimal\_node.append(cur\_city)

        visited.remove(cur\_city)

        # Update cur\_city

        cur\_city = list(f.items())[0][0]

        min\_cost\_value = list(f.items())[0][1]

        for neighbor\_city in visited:

            if f[neighbor\_city] < min\_cost\_value:

                cur\_city = neighbor\_city

                min\_cost\_value = f[neighbor\_city]

    shortest\_path, total\_distance = trace\_back(route, end\_city, city\_map)

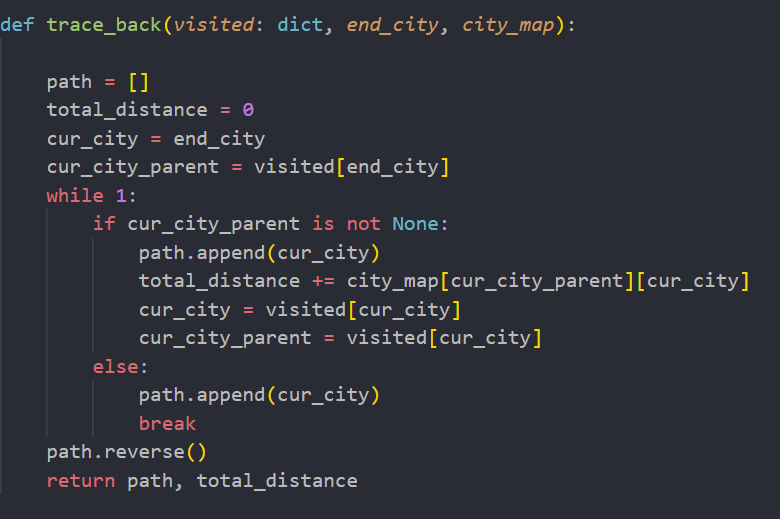
    print(f"Time complexity: {time\_space}")

    print(f"Space complexity: {time\_space}")

    print(f'Total distance: {total\_distance}')

    print(f'Path found: {shortest\_path}')

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



*Visualization*

Note: Internal Library: csv (No need to be installed)

External Libraby: matplotlib, networkx (Need to be installed)

Installation and Documentation:

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

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

<https://topdev.vn/blog/ve-do-thi-trong-python-voi-thu-vien-matplotlib/>

<https://www.youtube.com/watch?v=Ak7GamuoIr4&t=3657s>

<https://helpex.vn/question/networkx-thay-doi-mau-sac-chieu-rong-theo-cac-thuoc-tinh-canh-ket-qua-khong-nhat-quan-60944065f45eca37f4bf6dc0>

Brief explanation:

The csv library helps with reading the coordinate of the cities and their respective neighbors from .csv files while the matplotlib and networkx libraries help with drawing the graph.

Main Component: The printMap(*path\_format*) function:

* Read the coordinate of the cities and their respective neighbors from city\_coordinate.csv and neighbor.csv files.
* Generate data for city\_coordinates and neighbors.
* Use the networkx library to draw the nodes the edges and colors (this includes the path returned by the chosen algorithm highlighted in red).
* Use the matplotlib library to show the graph.

def printMap(*path\_format*):

    path = []

    # reformat the returned path for reading

    for item in path\_format:

        item\_delete\_space = item.replace(" ","")

        path.append(item\_delete\_space)

    # read the coordinate

    with open(city\_coordinate\_file\_path, "r") as csv\_file:

        coordinate\_reader = csv.reader(csv\_file, *delimiter*=',')

        coordinate\_reader = list(coordinate\_reader)

    # generate data for city nodes

    pos = {}

    for i in range(1, len(coordinate\_reader)):

        location = coordinate\_reader[i][1]

        pos[location] = (float(coordinate\_reader[i][3]), float(coordinate\_reader[i][2]))

    with open(neighbor\_file\_path, "r") as csv\_file:

        neighbor\_reader = csv.reader(csv\_file, *delimiter*=',')

        neighbor\_reader = list(neighbor\_reader)

    for i in range(1, len(neighbor\_reader)):

        if neighbor\_reader[i] == ['', '.', '']:

            continue

    # generate data edges

    distance = []

    for i in range(1, len(neighbor\_reader)):

        if neighbor\_reader[i] == ['', '.', '']:

            continue

        distance.append(

            list((neighbor\_reader[i][0], neighbor\_reader[i][1], neighbor\_reader[i][2])))

    G = nx.Graph()

    # generate city nodes

    for key in list(pos.keys()):

        G.add\_node(key, *pos*=pos[key])

    # generate edges

    for edge in distance:

        for i in pos.keys():

            # generate colors

            if i in path and path.index(i) < len(path) - 1:

                G.add\_edge(path[path.index(i)], path[path.index(i) + 1], *color*='r',*weight* = 5)

            else:

                G.add\_edge(edge[0], edge[1], *color*='black', *weight* = 1)

    # apply colors

    colors = nx.get\_edge\_attributes(G, 'color').values()

    weights = nx.get\_edge\_attributes(G, 'weight').values()

    # draw

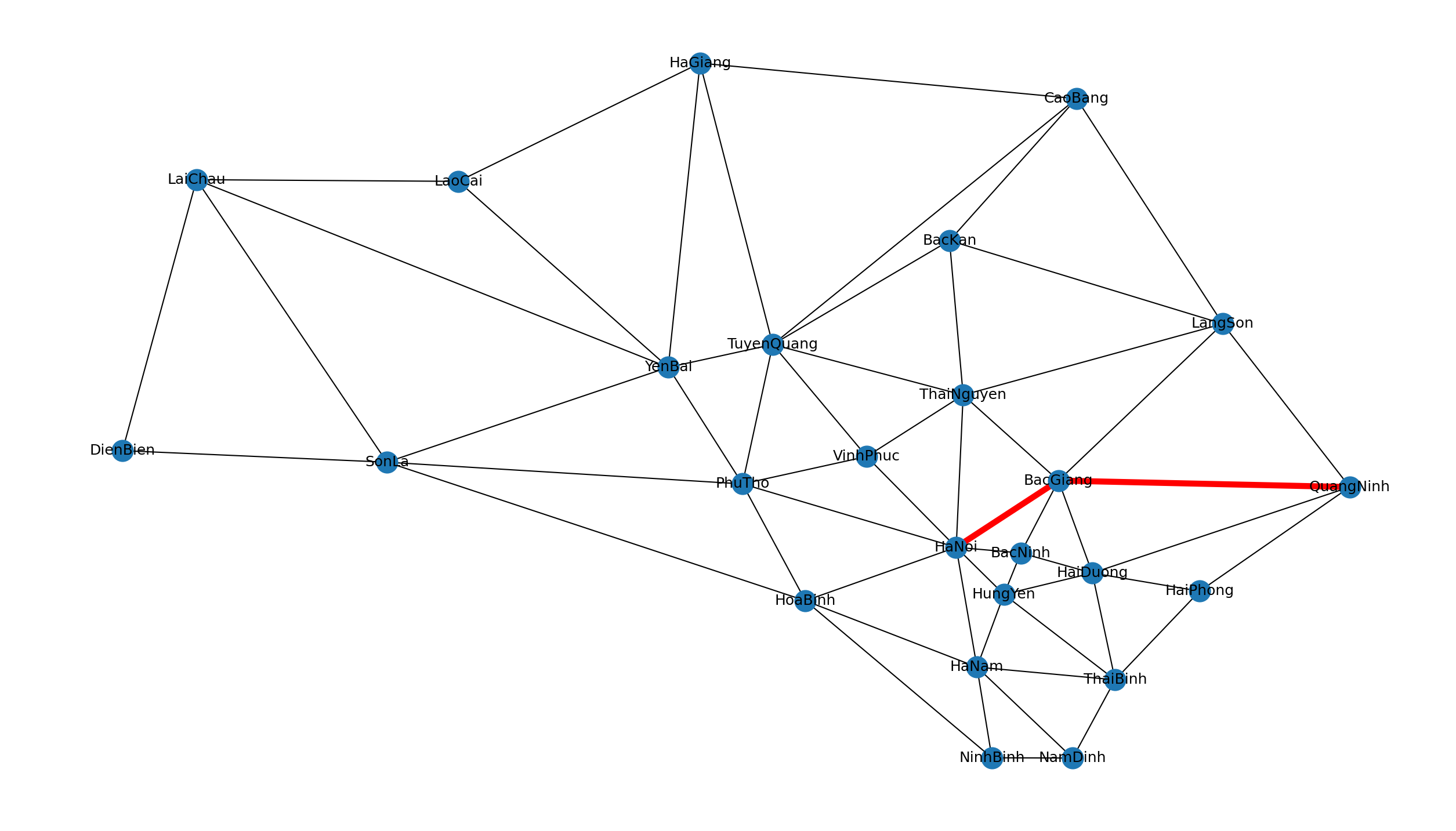
    plt.figure(3, *figsize*=(10, 10))

    nx.draw(G, *pos*=pos, *edge\_color*=colors, *width* = list(weights), *with\_labels*=True)

    # output the graph

    plt.show()

Output: The graph with the returned path from the chosen algorithm (The path highlighted in red).

**

*Menu*

Brief Explanation: The menu will be integrated in the main.py file with appropriate features.

Main Components:

* The *start\_city* and *end\_city* variables for storing the input start city and end city.
* Display the available algorithms and let the user choose one among them for execution and visualization.
* Execute the chosen algorithm with the given *start\_city* and *end\_city* and display the solution.

def main():

    start\_city = input('Start city: ')

    end\_city = input('End city: ')

    start\_city = ' '.join(start\_city.split()).title()

    end\_city = ' '.join(end\_city.split()).title()

    print("Enter the search algorithm: ")

    print("1. Uniform-cost Search")

    print("2. Greedy Best First Search")

    print("3. A\* Search")

    choice = int(input("1-3: "))

    if choice == 1:

        UCS(start\_city, end\_city, city\_map)

    elif choice == 2:

        GBFS(start\_city, end\_city, city\_map, heuristics\_distance)

    elif choice == 3:

        A\_star\_algorithm(start\_city, end\_city, city\_map, heuristics\_distance)

    else:

        print("Invalid Choice")

if \_\_name\_\_ == '\_\_main\_\_':

    main()

*Data Analysis*

Note: In this part, we will generate 50 random cases for exporting data for data analysis.

Each of the search algorithms (UCS, GBFS, A\*) defines an *"evaluation function"*, for each node *n* in the graph (or search space), denoted by *f(n).* This evaluation function is used to determine which node, while searching, is "expanded" first, that is, which node is first removed from the [*"fringe"*](https://ai.stackexchange.com/q/5949/2444), so as to *"visit"* its children.

🡪 The main difference seperates these three algorithms is how they calculate their evaluation function*.*

* UCS: *f(n) = g(n)*
* GBFS: *f(n) = h(n)*
* A\*: *f(n) = g(n) + h(n)*

With:

*g(n)* is the is the path cost of moving to a node n.

*h(n)* or the heuristic function is the estimated cost (i.e the straight line distance) that it will take to get to the final goal state (end city) from if we were to go to n.

In the case of the Uniform Cost Search algorithm: In this problem, UCS is **complete** (all the cost between 2 adjacent cities > 0) and **optimal** (nodes are expanded in increasing order of *g(n)* as we are storing generated nodes in a PriorityQueue of *g(n)*). But having no heuristic function: *f(n) = g(n)* (i.e cannot deal with heuristic function), UCS turns out to be much worse in computation time and memory required for storage compare to GBFS and A\* 🡪 An admissible heuristic has great effect on the time and space complexity of a search algorithm.

In the case of [the Greedy Best First Search algorithm](https://en.wikipedia.org/wiki/Best-first_search#Greedy_BFS): the evaluation function is *f(n) = h(n)*, that is, the GBFS algorithm first expands the node whose estimated distance to the goal is the smallest. So, greedy BFS does not use the "past knowledge", i.e. *g(n)*. Hence its connotation "greedy". In this problem and in general, the GBFS algorithm is **not complete**, that is, there is always the risk to take a path that does not bring to the goal (infinite loop or dead end). It is also **not optimal**, that is, the path found may not be the optimal one.

In the case of [the A\* algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm), the evaluation function is  *f(n) = g(n) + h(n)*, where h is an “[*admissible heuristic function*](https://en.wikipedia.org/wiki/Admissible_heuristic)*”*. The *"star"*, often denoted by an asterisk, \*, refers to the fact that A\* uses an admissible heuristic function, which essentially means that A\* is **optimal**, that is, it always finds the optimal path between the starting node and the goal node. A\* is also **complete** (unless there are infinitely many nodes to explore in the search space). However, A\* needs to keep all nodes in memory while searching, not just the ones in the fringe, because A\*, essentially, performs an "exhaustive search" (which is "informed", in the sense that it uses a heuristic function).

In most test case, A\* uses more memory than GBFS 🡪 A\* becomes impractical when the search space is huge. However, A\* also guarantees that the found path between the starting node and the goal node is the optimal one and that the algorithm eventually terminates. GBFS, on the other hand, uses less memory, but does not provide the optimality and completeness guarantees of A\*.

*Recommended Further Research*