VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

PROJECT

ARTIFICIAL INTELLIGENCE

MIDTERM

Supervisor: **PhD BÙI THANH HÙNG**

Authors: **LÊ MINH ĐĂNG– 520K0108**

HO CHI MINH CITY, 2022

VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

PROJECT

ARTIFICIAL INTELLIGENCE

MIDTERM

Supervisor: **PhD BÙI THANH HÙNG**

Authors: **LÊ MINH ĐĂNG– 520K0108**

HO CHI MINH CITY, 2022

ACKNOWLEDGEMENTS

With enthusiasm, sharing the teacher's useful teaching experiences. Although the time is not long, it has helped me confidently go into the major with the lessons, experiences, as well as encouraging reminders of the lecturers.

I sincerely thank you.

PROJECT IS COMPLETED

AT TON DUC THANG UNIVERSITY

I commit that this project is my / our own project and is supervised by PhD. Bùi Thanh Hùng. The contents, results in this topic are honest and unpublished in any form before. The data in the tables for analyzing, commenting and evaluating are collected from various sources and by the authors and the citations are specified in the References section.

In addition, a number of comments and assessments as well as data from other authors and organizations that are also used in the project are referenced and annotated clearly.

If this project has any cheating or plagiarism, I take full responsibility for the content of my project. Ton Duc Thang University is not related to infringement of copyright caused by me during the process of this project implementation.

Ho Chi Minh City, April 4th , 2022

Authors

(signature and full name)

Lê Minh Đăng

ABSTRACT

Through many days of studying, I was able to develop more thinking ability, be able to apply algorithms and gain more understanding in the field of artificial intelligence.

CONTENTS

PROBLEMS

Problem 1:

Find number of islands by two algorithms:

- BFS (Breadth First Search) (2.5 marks)
- UCS (Uniform Cost Search) (2.5 marks)

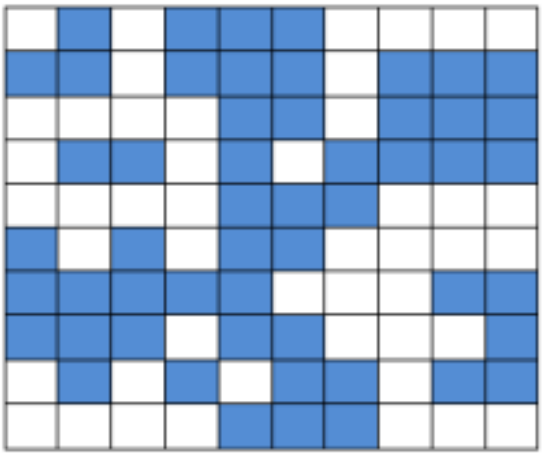
BFS:

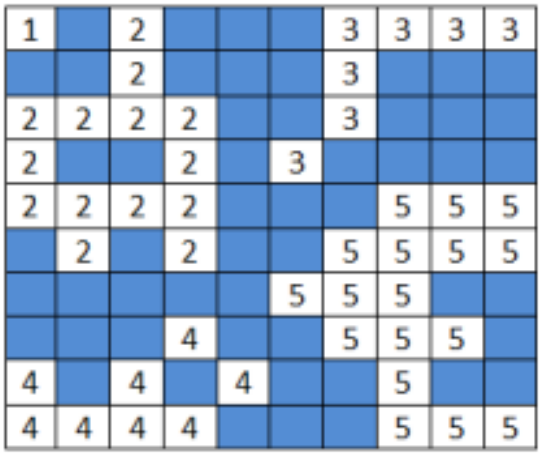
There are generally two algorithms that are used for the traversal of a graph. They are BFS (Breadth-First Search) and DFS (Depth First Search) algorithms. Traversal of the Graph is visiting exactly once each vertex or node and edge, in a well-defined order. It is also essential to keep track of the vertices that have been visited so that the same vertex is not traversed twice. In the Breadth First Search algorithm, the traversing starts from a selected node or source node, and the traversal continues through the nodes directly connected to the source node. In simpler terms, in the BFS algorithm, one should first horizontally move and traverse the current layer after moving to the next layer.

Algorithm of BFS

- In a given graph, let us start from a vertex, i.e. in the above diagram, it is node 0. The level in which this vertex is present can be denoted as Layer 0.
- The next step is to find all the other vertices that are adjacent to the starting vertex, i.e. node 0 or immediately accessible from it. Then we can mark these adjacent vertices to be present at Layer 1.
- It is possible to come to the same vertex due to a loop in the graph. So, we should travel only to those vertices which should be present in the same layer.
- Next, the parent vertex of the current vertex that we are at is marked. The same is to be performed for all the vertices at Layer 1.

- The next step is to find all those vertices that are a single edge away from all the vertices at Layer 1. This new set of vertices will be at Layer 2.
- The above process is repeated until all the nodes are traversed.





Using BFS to count the number of the Island above:

```
def BFS(self, vis, si, sj):


    row = [-1, -1, -1, 0, 0, 1, 1, 1]
    col = [-1, 0, 1, -1, 1, -1, 0, 1]

    q = deque()
    q.append([si, sj])
    vis[si][sj] = True


    while (len(q) > 0):
        temp = q.popleft()

        i = temp[0]
        j = temp[1]

        for k in range(8):
            if (self.isSafe(i + row[k], j + col[k], vis)):
                vis[i + row[k]][j + col[k]] = True
                q.append([i + row[k], j + col[k]])
```

In this function I call another function to check if the Vertex is visited or not

```
def isSafe(self, i, j, vis):

    return ((i >= 0) and (i < self.R) and
            (j >= 0) and (j < self.C) and
        (self.G[i][j] and (not vis[i][j])))
```

After the search is done, we will call the function to count the number of found islands.

```
def countIslands(self):

    vis = [[False for i in range(self.C)]
           for i in range(self.R)]

    res = 0

    for i in range(self.R):
        for j in range(self.C):
            if (self.G[i][j] and not vis[i][j]):
                self.BFS(vis, i, j)
                res += 1

    return res
```

This is my example

```
if __name__ == '__main__':

    mat = [ [ 1, 1, 0, 0, 0, 1 ],
            [ 0, 1, 0, 0, 0, 0 ],
            [ 1, 0, 0, 1, 1, 0 ],
            [ 0, 0, 0, 0, 0, 0 ],
            [ 1, 0, 1, 0, 1, 0 ],
            [ 1, 0, 1, 0, 1, 0 ],
            [ 1, 0, 1, 0, 1, 0 ]]
    g = Graph(len(mat),len(mat[0]),mat)
    print (g.countIslands())
```

And the result is: **6**

UCS:

Uniform Cost Search is an algorithm used to move around a directed weighted search space to go from a start node to one of the ending nodes with a minimum cumulative cost. This search is an uninformed search algorithm since it operates in a brute-force manner, i.e. it does not take the state of the node or search space into consideration. It is used to find the path with the lowest cumulative cost in a weighted graph where nodes are expanded according to their cost of traversal from the root node. This is implemented using a priority queue where lower the cost higher is its priority.

Algorithm of UCS :

- Insert RootNode into the queue.
- Repeat till queue is not empty:
- Remove the next element with the highest priority from the queue.
- If the node is a destination node, then print the cost and the path and exit

Using UCS to count the number of island

```
def UCS(self, vis, si, sj, count):

    row = [-1, -1, -1, 0, 0, 1, 1, 1]
    col = [-1, 0, 1, -1, 1, -1, 0, 1]

    pq = PriorityQueue()
    pq.put((self.G[si][sj], si, sj))
    vis[si][sj] = True

    while not pq.empty():
        temp = pq.get();

        cost = temp[0];
        i = temp[1];
        j = temp[2];

        for k in range(8):
            if (self.isIsland(i + row[k], j + col[k], vis)):
                vis[i + row[k]][j + col[k]] = True
                count-=1
                pq.put((self.G[i + row[k]][j + col[k]] + cost, i + row[k], j + col[k]))

    return count
```

I use another function to check if it visited or not

```
def isIsland(self, i, j, vis):

    return ((i >= 0) and (i < self.R) and
            (j >= 0) and (j < self.C) and
        (self.G[i][j] and (not vis[i][j])))
```

I create a function t count island

```
def countIslands(self):
    count = 0
    for i in range(self.R):
        for j in range(self.C):
            if self.G[i][j] != 0:
                count+=1

    vis = [[False for i in range(self.C)]
            for i in range(self.R)]

    for i in range(self.R):
        for j in range(self.C):
            if (self.G[i][j] and not vis[i][j]):
                count = self.UCS(vis, i, j, count)

    return count
```

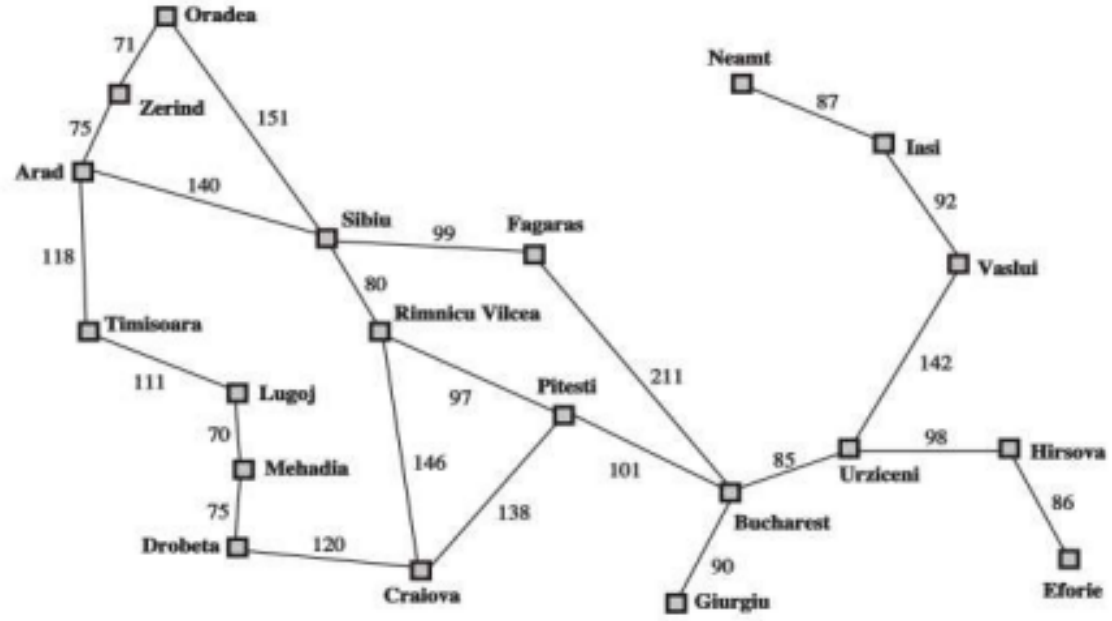
My example of this

```
if __name__ == '__main__':

    mat = [ [ 1, 1, 0, 0, 0, 1 ],
            [ 0, 1, 0, 0, 0, 0 ],
            [ 1, 0, 0, 1, 1, 0 ],
            [ 0, 0, 0, 0, 0, 0 ],
            [ 1, 0, 1, 0, 1, 0 ],
            [ 1, 0, 1, 0, 1, 0 ],
            [ 1, 0, 1, 0, 1, 0 ]]


    g = Graph(len(mat),len(mat[0]),mat)
    print (g.countIslands())
```

The result is: **6**

Problem 2

Romania Travelling



Greedy Best First Search

In this algorithm, we expand the closest node to the goal node. The closeness factor is roughly calculated by heuristic function h(x). The node is expanded or explored when f (n) = h (n). This algorithm is implemented through the priority queue. It is not an optimal algorithm. It can get stuck in loops.

A* Search

A* search is a combination of greedy search and uniform cost search. In this algorithm, the total cost (heuristic) which is denoted by f(x) is a sum of the cost in uniform cost search denoted by g(x) and cost of greedy search denoted by h(x). f (x) = g (x) + h (x) In this g(x) is the backward cost which is the cumulative cost from the root node to the

current node and h(x) is the forward cost which is approximate of the distance of goal node and the current node.

Here are 2 functions of these algorithm in Romania Travelling problem

G-BFS

```
# Greedy Best First Search Algorithm
def GBFS(startNode, heuristics, graph, goalNode="Bucharest"):
    priorityQueue = queue.PriorityQueue()
    priorityQueue.put((heuristics[startNode], startNode))

    path = []

    while priorityQueue.empty() == False:
        current = priorityQueue.get()[1]
        path.append(current)

        if current == goalNode:
            break

        priorityQueue = queue.PriorityQueue()

        for i in graph[current]:
            if i[0] not in path:
                priorityQueue.put((heuristics[i[0]], i[0]))

    return path
```

A*

```python
# Astar Algorithm
def Astar(startNode, heuristics, graph, goalNode="Bucharest"):
    priorityQueue = queue.PriorityQueue()
    distance = 0
    path = []

    priorityQueue.put((heuristics[startNode] + distance, [startNode, 0]))

    while priorityQueue.empty() == False:
        current = priorityQueue.get()[1]
        path.append(current[0])
        distance += int(current[1])

        if current[0] == goalNode:
            break

        priorityQueue = queue.PriorityQueue()

        for i in graph[current[0]]:
            if i[0] not in path:
                priorityQueue.put((heuristics[i[0]] + int(i[1]) + distance, i))

    return path
```

The result of these algorithms are visualized in a graph

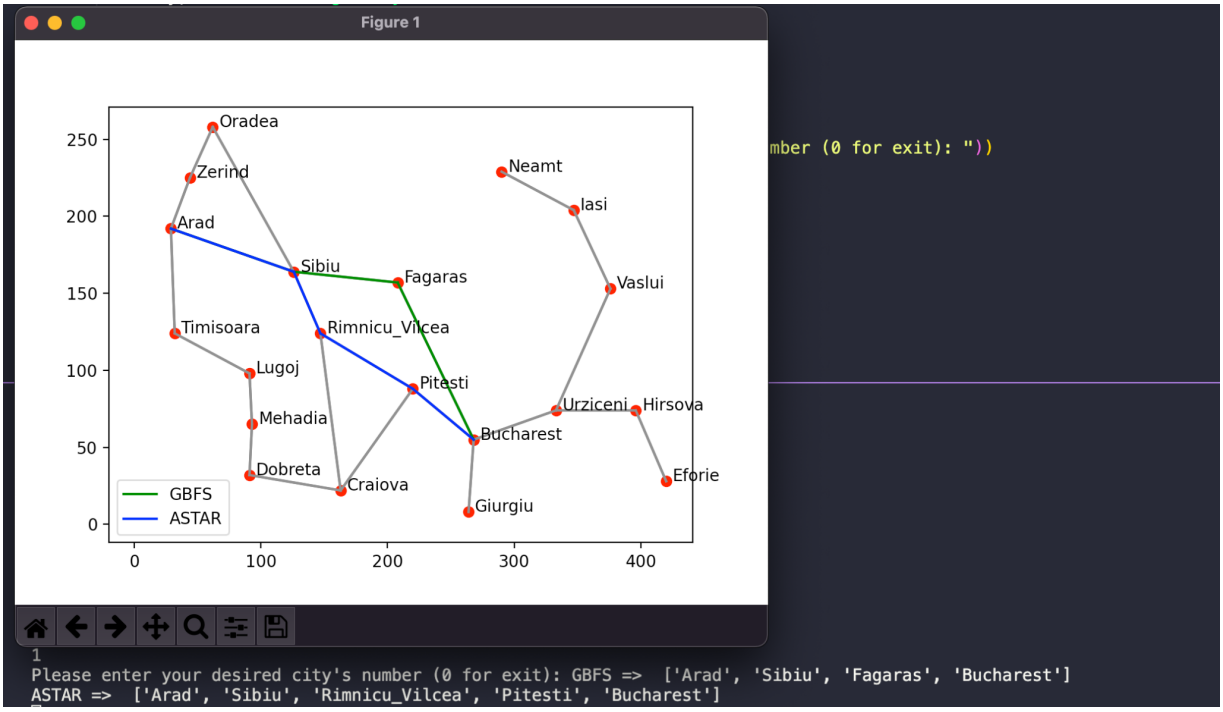


Figure 1
Oradea
Zerind
Arad
Sibiu
Fagaras
Neamt
Iasi
Vaslui
Timisoara
Rimnicu_Vilcea
Lugoj
Pitesti
Urziceni
Hirsova
Mehadia
Bucharest
Dobreta
Craiova
Eforie
Giurgiu
GBFS
ASTAR
0
50
100
150
200
250
0
100
200
300
400
mber (0 for exit): "))
1
Please enter your desired city's number (0 for exit): GBFS => ['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
ASTAR => ['Arad', 'Sibiu', 'Rimnicu_Vilcea', 'Pitesti', 'Bucharest']

REFERENCES

https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/
https://www.educba.com/search-algorithms-in-ai/
https://www.educba.com/bfs-algorithm/
https://www.edureka.co/blog/breadth-first-search-algorithm/
https://www.educba.com/uniform-cost-search/
https://github.com/aimacode/aima-python/blob/master/search.py
https://notebook.community/jo-tez/aima-python/search