IMPLEMENTATION OF BEST FIRST SEARCH AND A* SEARCH ALGORITHM

AI LAB 5

RAJAT KUMAR

RA1911003010652

CODE FOR A* SEARCH ALGORITHM:

```
from collections import deque

class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list


    def get_neighbors(self, v):
        return self.adjacency_list[v]


    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
```

```
        return H[n]


    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
```

```
for v in open_list:
    if n == None or g[v] + self.h(v) < g[n] + self.h(n):
        n = v;

if n == None:
    print('Path does not exist!')
    return None

# if the current node is the stop_node
# then we begin reconstructin the path from it to the start_node
if n == stop_node:
    reconst_path = []

    while parents[n] != n:
        reconst_path.append(n)
        n = parents[n]

    reconst_path.append(start_node)

    reconst_path.reverse()

    print('Path found: {}'.format(reconst_path))
    return reconst_path
```

```
# for all neighbors of the current node do
for (m, weight) in self.get_neighbors(n):
    # if the current node isn't in both open_list and closed_list
    # add it to open_list and note n as it's parent
    if m not in open_list and m not in closed_list:
        open_list.add(m)
        parents[m] = n
        g[m] = g[n] + weight

    # otherwise, check if it's quicker to first visit n, then m
    # and if it is, update parent data and g data
    # and if the node was in the closed_list, move it to open_list
    else:
        if g[m] > g[n] + weight:
            g[m] = g[n] + weight
            parents[m] = n

            if m in closed_list:
                closed_list.remove(m)
                open_list.add(m)

# remove n from the open_list, and add it to closed_list
# because all of his neighbors were inspected
open_list.remove(n)
```

```
            closed_list.add(n)


        print('Path does not exist!')
        return None
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

OUTPUT:

RESULT: Hence, implementation of A* search algorithm is successfully done.

CODE FOR BEST FIRST SEARCH ALGORITHM:

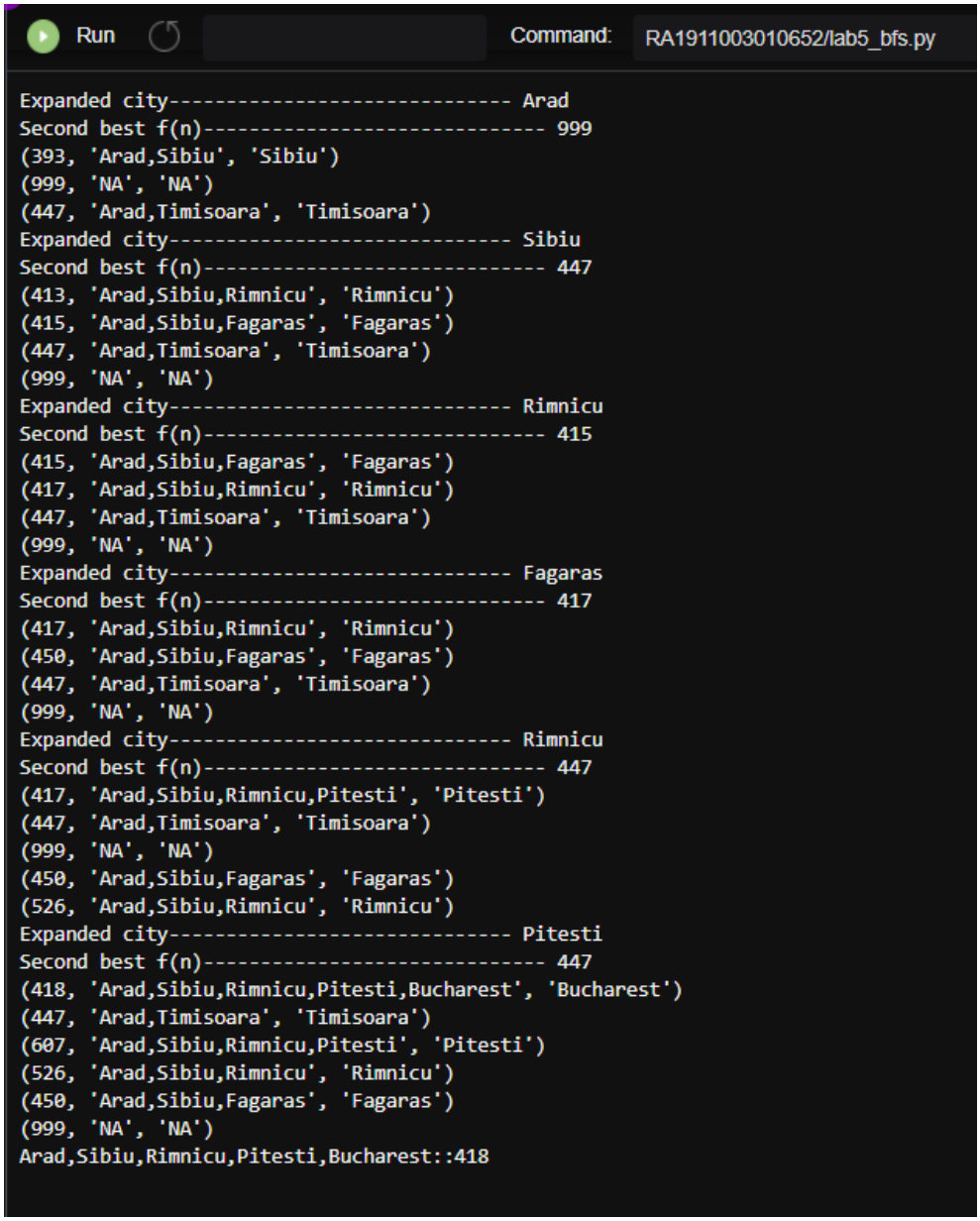
```
dict_hn={'Arad':336,'Bucharest':0,'Craiova':160,'Drobeta':242,'Eforie':161,
        'Fagaras':176,'Giurgiu':77,'Hirsova':151,'Iasi':226,'Lugoj':244,
        'Mehadia':241,'Neamt':234,'Oradea':380,'Pitesti':100,'Rimnicu':193,
        'Sibiu':253,'Timisoara':329,'Urziceni':80,'Vaslui':199,'Zerind':374}


dict_gn=dict(
Arad=dict(Zerind=75,Timisoara=118,Sibiu=140),
Bucharest=dict(Urziceni=85,Giurgiu=90,Pitesti=101,Fagaras=211),
Craiova=dict(Drobeta=120,Pitesti=138,Rimnicu=146),
Drobeta=dict(Mehadia=75,Craiova=120),
Eforie=dict(Hirsova=86),
Fagaras=dict(Sibiu=99,Bucharest=211),
Giurgiu=dict(Bucharest=90),
Hirsova=dict(Eforie=86,Urziceni=98),
Iasi=dict(Neamt=87,Vaslui=92),
Lugoj=dict(Mehadia=70,Timisoara=111),
Mehadia=dict(Lugoj=70,Drobeta=75),
Neamt=dict(Iasi=87),
Oradea=dict(Zerind=71,Sibiu=151),
Pitesti=dict(Rimnicu=97,Bucharest=101,Craiova=138),
Rimnicu=dict(Sibiu=80,Pitesti=97,Craiova=146),
Sibiu=dict(Rimnicu=80,Fagaras=99,Arad=140,Oradea=151),
Timisoara=dict(Lugoj=111,Arad=118),
```

```
Urziceni=dict(Bucharest=85,Hirsova=98,Vaslui=142),
Vaslui=dict(Iasi=92,Urziceni=142),
Zerind=dict(Oradea=71,Arad=75)
)
import queue as Q


start='Arad'
goal='Bucharest'
result=''

def get_fn(citystr):
    cities=citystr.split(',')
    hn=gn=0
    for ctr in range(0,len(cities)-1):
        gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
    hn=dict_hn[cities[len(cities)-1]]
    return(hn+gn)

def printout(cityq):
    for i in range(0,cityq.qsize()):
        print(cityq.queue[i])

def expand(cityq):
```

```
    global result
    tot,citystr,thiscity=cityq.get()
    nexttot=999
    if not cityq.empty():
        nexttot,nextcitystr,nextthiscity=cityq.queue[0]
    if thiscity==goal and tot<nexttot:
        result=citystr+'::'+str(tot)
        return
    print("Expanded city------------------------------",thiscity)
    print("Second best f(n)------------------------------",nexttot)
    tempq=Q.PriorityQueue()
    for cty in dict_gn[thiscity]:
            tempq.put((get_fn(citystr+','+cty),citystr+','+cty,cty))
    for ctr in range(1,3):
        ctrtot,ctrcitystr,ctrthiscity=tempq.get()
        if ctrtot<nexttot:
            cityq.put((ctrtot,ctrcitystr,ctrthiscity))
        else:
            cityq.put((ctrtot,citystr,thiscity))
            break
    printout(cityq)
    expand(cityq)
def main():
    cityq=Q.PriorityQueue()
```

```
    thiscity=start
    cityq.put((999,"NA","NA"))
    cityq.put((get_fn(start),start,thiscity))
    expand(cityq)
    print(result)
main()
```

OUTPUT:



RESULT: Hence, the implementation of Best First Search is done successfully done.

RAJAT KUMAR
RA1911003010652

AI - LABS
A* SEARCH ALGORITHM

ALGORITHMS:-

step 1: Start
step 2: Make an Open list containing starting node
* If it reaches the dest^n node:
→ make a closed empty list
* If it does not ~~match~~ reach the dest^n node, then consider a node with the lowest f-score in the open list.
Finished

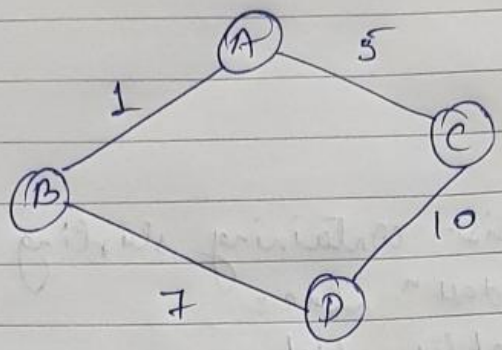
Step 3: ~~Pop~~ Else
* Put the current node in the list and check
→ For each neighbor of the current node:
If the neighbor has lower g value than the current node and is in closed list
Replace neighbor with this new node as the neighbor's parent.

step 4:
Else if (current g is lower ~~than~~ and neighbor is in the open list):
Replace neighbor with the lower g value and change the neighbor's parent to current node.

step 5: Else if the neighbor is not in both lists:
Add to the open list and set its g.

step 6: stop.

Consider an example



To calculate best path:

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

$f(n)$ = final cost
$g(n)$ = cost of travelling from one node another.
$h(n)$ = heuristic approx of the node's value

Starting Node → A

$f(A) = g(A) + h(A)$
$= 0 + 6 = 6$

$f(A-B) = 1+4$
$f(A-C) = 5+2$

$f(A-B-D) = (1+7)+0$
$f(A-C-D) = (5+10)+0$

It is clear that node B gives the best path, so that is the node you need to reach the destination.

RAJAT KUMAR
RA1911003010652

classmate
Date
Page

AI LAB-5
BEST FIRST SEARCH

ALGORITHM:-

step 1:- Create 2 empty lists: OPEN and CLOSED
step 2:- Start from the initial node and put it in the ordered list (OPEN).
step 3:- Repeat the next steps until GOAL node is reached.

(i) If OPEN is empty, then EXIT the loop returning FALSE.

(ii) select the first/top node in the OPEN list and moved it to CLOSED list.
Also captured the info of the parent node.

(iii) If N is not a goal node, then move the node to the closed list and exit the loop returning 'True'. Backtracking used to find soln.

(iv) If N is not a GOAL state, expand node N to generate the 'immediate' next node linked to Node N and add all OPEN List.

(v) Reorder the nodes in the OPEN List in ascending order according to f(n).

Time complexity : $O(n * \log n)$.