Ex No.5: *Implementation of Best First Search and A* Search for an Application*

Best First Search (Informed Search)

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So, both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So, the implementation is a variation of BFS, we just need to change Queue to Priority Queue.

Algorithm :

1) Create an empty PriorityQueue
   PriorityQueue **pq**;
2) Insert "start" in pq.
   pq.insert(start)
3) Until PriorityQueue is empty
   u = PriorityQueue.DeleteMin
   If u is the goal
   Exit
   Else
   Foreach neighbor v of u
   If v "Unvisited"
   Mark v "Visited"
   pq.insert(v)
   Mark u "Examined"
   End procedure

A Algorithm*

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all path's transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighbouring nodes, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost

Algorithm :

1)Make an open list containing starting node .

2)If it reaches the destination node :

- Make a closed empty list
- If it does not reach the destination node, then consider a node with the lowest f-score in the open list
- We are finished

3) Else : Put the current node in the list and check its neighbors

4) · For each neighbor of the current node :

- If the neighbor has a lower g value than the current node and is in the closed list: Replace neighbor with this new node as the neighbor's parent

5) · Else If (current g is lower and neighbor is in the open list):

6) Replace neighbor with the lower g value and change the neighbor's parent to the current node. · Else If the neighbor is not in both lists:

7) Add it to the open list and set its g

Code :

Best First Search

```
from queue import PriorityQueue
v = 5
graph = [[] for i in range(v)]
def best_first_search(source, target, n):
  visited = [0] * n
  visited[0] = True
  pq = PriorityQueue()
  pq.put((0, source))
  while pq.empty() == False:
    u = pq.get()[1]
    print(u, end=" ")
    if u == target:
      break
    for v, c in graph[u]:
      if visited[v] == False:
        visited[v] = True
        pq.put((c, v))
  print()
def addedge(x, y, cost):
  graph[x].append((y, cost))
  graph[y].append((x, cost))
addedge(0, 1, 5)
addedge(0, 2, 1)
addedge(2, 3, 2)
addedge(1, 4, 1)
addedge(3, 4, 2)
source = 0
target = 4
best_first_search(source, target, v)
```

Output:

In [19]:

```
from queue import PriorityQueue
v = 5
graph = [[] for i in range(v)]
def best_first_search(source, target, n):
  visited = [0] * n
  visited[0] = True
  pq = PriorityQueue()
  pq.put((0, source))
  while pq.empty() == False:
    u = pq.get()[1]
    print(u, end=" ")
    if u == target:
      break
    for v, c in graph[u]:
      if visited[v] == False:
        visited[v] = True
        pq.put((c, v))
  print()
def addedge(x, y, cost):
  graph[x].append((y, cost))
  graph[y].append((x, cost))
addedge(0, 1, 5)
addedge(0, 2, 1)
addedge(2, 3, 2)
addedge(1, 4, 1)
addedge(3, 4, 2)
source = 0
target = 4
best_first_search(source, target, v)
```

```
0 2 3 4
```

In [ ]:

A* Search

```
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]

    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 = set([start_node])
    closed_list = set([])
    g = {}

    g[start_node] = 0
    parents = {}
    parents[start_node] = start_node

    while len(open_list) > 0:
        n = None
        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 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 (m, weight) in self.get_neighbors(n):
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            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)
            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:

```
            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 (m, weight) in self.get_neighbors(n):
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            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)
        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')
```

```
Path found: ['A', 'B', 'D']
```

Out[18]:
```
['A', 'B', 'D']
```

In [ ]:

Result :

A* and best first search algorithms were implemented successfully.