un-informedSearches

June 18, 2021

[1]:
```
#!/usr/bin/env python
# coding: utf-8

# In[86]:



#Making graphs
#These algorithms can be applied to traverse graphs or trees. To represent such␣
 ↪data structures in Python,
#all we need to use is a dictionary where the vertices (or nodes) will be␣
 ↪stored as keys and the adjacent vertices as values.

from queue import PriorityQueue
#im using priority queue to get the minimum value from queue



small_graph = {
    'A': ['B', 'C'],
    'B': ['D', 'A'],
    'C': ['A'],
    'D': ['B']
}

def dfs(graph, start, goal):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)

            if node == goal:
                #print("Success.. Goal found!")
                return True
            for neighbor in graph[node]:
                if neighbor not in visited:
```

```
                    stack.append(neighbor)
    return False

from collections import deque
def bfs(graph, start, goal):
    visited = set()
    queue = [start]

    while queue:
        node = queue.pop()
        if node not in visited:
            visited.add(node)

            if node == goal:
                return True
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
    return False

class Graph:
    def __init__(self):
        self.edges = {
        'S': ['A', 'G'],
        'A': ['B', 'C'],
        'B': ['D'],
        'C': ['D', 'G'],
        'D': ['G']
        }
        self.weights = {'S':[1,12],
                        'A': [3,1],
                        'B': [3],
                        'C': [1,2],
                        'D': [3]
                       }

    def neighbors(self, node):
        return self.edges[node]

    def get_cost(self, from_node, to_node):

        #check if path exists or not
        if to_node in self.edges[from_node]:
            #get the index of to_node
            index = self.edges[from_node].index(to_node)
        else:
            print("To_Node doesn't exists.")
```

```
            return
        cost = self.weights[from_node][index]
        return cost

def ucs(graph, start, goal):
    visited = set()
    queue = PriorityQueue()
    queue.put((0, start))

    while queue:
        cost, node = queue.get()
        print("Node to be expanded is: ",node+", Cost = ",cost)
        if node not in visited:
            visited.add(node)
            print("Visited Nodes are : ", visited)


            if node == goal:
                print("Min Path = ",cost)
                return
            #Visiting every node connected to the current node and finding␣
↪total cost (root-to-uptill now)
            for i in graph.neighbors(node):
                if i not in visited:
                    total_cost = cost + graph.get_cost(node, i)
                    queue.put((total_cost, i))
                    #print("Node with total cost to be added is : ", i,␣
↪total_cost)
    return False


# In[87]:


g = Graph()
#print(g.neighbors('A'))
#print(g.edges)
#print(g.get_cost('C','G'))
ucs(g, 'S', 'G')


# In[ ]:
```

```
# In[ ]:
```

```
Node to be expanded is:  S, Cost =  0
Visited Nodes are :  {'S'}
Node to be expanded is:  A, Cost =  1
Visited Nodes are :  {'S', 'A'}
Node to be expanded is:  C, Cost =  2
Visited Nodes are :  {'C', 'S', 'A'}
Node to be expanded is:  D, Cost =  3
Visited Nodes are :  {'C', 'D', 'S', 'A'}
Node to be expanded is:  B, Cost =  4
Visited Nodes are :  {'A', 'S', 'B', 'C', 'D'}
Node to be expanded is:  G, Cost =  4
Visited Nodes are :  {'A', 'S', 'B', 'C', 'D', 'G'}
Min Path =  4
```

[ ]:

```
```