DFS_BFS___UCS_GreedyBFS_

June 18, 2021

[25]:
```python
#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
from collections import deque
#im using priority queue to get the minimum value from queue


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

#Depth-First Search Algorithm
#================================
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
```

```python
#Breadth-First Search Algorithm
#================================
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', 'B', 'C'],
        'A': ['B', 'D'],
        'B': ['D','H'],
        'C': ['L'],
        'D': ['F'],
        'F': ['G'],
        'G': ['E'],
        'H': ['G'],
        'I': ['K'],
        'J': ['K'],
        'K': ['E'],
        'L': ['I', 'J']
        }
        self.weights = {'S':[9,8,7],
                        'A': [8,6],
                        'B': [6,7],
                        'C': [6],
                        'D': [4],
                        'F': [3],
                        'G': [0],
                        'H': [3],
                        'I': [4],
                        'J': [4],
```

```
                        'K': [0],
                        'L': [9,9]
                       }

    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

#Uniform Cost Search Algorithm
#===============================
def ucs(graph, start, goal):

    print("======================\nUniform Cost␣
↪Search\n======================")
    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,"\n")

            if node == goal:
                print("Min Cost = ",cost)
                return queue
            #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) #for UCS, we␣
↪add total path-cost from root-to-current!
                    queue.put((total_cost, i))
```

```python
    return False

#Greedy Best First Search Algorithm
#====================================
def greedyBFS(graph, start, goal):

    print("======================\nGreedy Best First␣
↪Search\n======================")
    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,"\n")

            if node == goal:
                print("Min Cost = ",cost)
                return queue
            #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) #for UCS, we␣
↪add total path-cost from root-to-current!
                    cost = graph.get_cost(node, i)
                    queue.put((cost, i))

                    #print("Node with total cost to be added is : ", i,␣
↪total_cost)
    return False
```

[26]:
```python
g = Graph()
print(g.edges)
print(g.weights)

que = greedyBFS(g, 'S', 'E')
que = ucs(g, 'S', 'E')

#print(que.get())
#print(que.get())
#print(que.get())
```

```python
#print(bfs(small_graph, 'S', 'G'))
#print(dfs(small_graph, 'S', 'G'))
```

```
{'S': ['A', 'B', 'C'], 'A': ['B', 'D'], 'B': ['D', 'H'], 'C': ['L'], 'D': ['F'],
'F': ['G'], 'G': ['E'], 'H': ['G'], 'I': ['K'], 'J': ['K'], 'K': ['E'], 'L':
['I', 'J']}
{'S': [9, 8, 7], 'A': [8, 6], 'B': [6, 7], 'C': [6], 'D': [4], 'F': [3], 'G':
[0], 'H': [3], 'I': [4], 'J': [4], 'K': [0], 'L': [9, 9]}
=======================
Greedy Best First Search
=======================
Node to be expanded is:  S, Cost =  0
Node to be expanded is:  C, Cost =  7
Node to be expanded is:  L, Cost =  6
Node to be expanded is:  B, Cost =  8
Node to be expanded is:  D, Cost =  6
Node to be expanded is:  F, Cost =  4
Node to be expanded is:  G, Cost =  3
Node to be expanded is:  E, Cost =  0
Min Cost =  0
=======================
Uniform Cost Search
=======================
Node to be expanded is:  S, Cost =  0
Node to be expanded is:  C, Cost =  7
Node to be expanded is:  B, Cost =  8
Node to be expanded is:  A, Cost =  9
Node to be expanded is:  L, Cost =  13
Node to be expanded is:  D, Cost =  14
Node to be expanded is:  D, Cost =  15
Node to be expanded is:  H, Cost =  15
Node to be expanded is:  F, Cost =  18
Node to be expanded is:  G, Cost =  18
Node to be expanded is:  E, Cost =  18
Min Cost =  18
```

[ ]:

[ ]:

[ ]: