Lab Report-04

Course Title: Artificial Intelligence

Course Code: CSE 366

Semester: Fall 2021

Section No: 01

Submitted By

Name: Syeda Tamanna Sheme

ID:2018-2-60-010

Submitted To

Md Al-Imran

Lecturer

Department of Computer Science & Engineering

Date of Submission: 27 November, 2021

Lab-04: BFS, DFS, UCS Algorithm

Theory

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). Python is named after a TV Show called ëMonty Pythonís Flying Circusí and not after Python-the snake.

Python 3.0 was released in 2008. Although this version is supposed to be backward incompatibles, later on many of its important features have been backported to be compatible with version 2.7.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable.

Characteristics of Python

Following are important characteristics of python −

- It supports functional and structured programming methods as well as OOP.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python is one of the most widely used language over the web.

Breadth-First Search

Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

BFS Algorithm

As breadth-first search is the process of traversing each node of the graph, a standard BFS algorithm traverses each vertex of the graph into two parts: 1) Visited 2) Not Visited. BFS starts from a node, then it checks all the nodes at distance one from the beginning node, then it checks all the nodes at distance two, and so on. So as to recollect the nodes to be visited, BFS uses a queue.

The steps of the algorithm work as follow:

- Putting any one of the graph's vertices at the back of the queue.
- Now take the front item of the queue and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
- Keep continuing steps two and three till the queue is empty.

BFS pseudocode

The pseudocode for BFS in python goes as below:

```
create a queue Q

mark v as visited and put v into Q

while Q is non-empty

   remove the head u of Q

   mark and enqueue all (unvisited) neighbors of u
```

BFS implementation in Python (Source Code)

```
# sample graph implemented as a dictionary
graph = {'A': ['B', 'C', 'E'],
         'B': ['A','D', 'E'],
         'C': ['A', 'F', 'G'],
         'D': ['B'],
         'E': ['A', 'B','D'],
         'F': ['C'],
         'G': ['C']}


# finds shortest path between 2 nodes of a graph using BFS
def bfs_shortest_path(graph, start, goal):
    # keep track of explored nodes
    explored = []
    # keep track of all the paths to be checked
    queue = [[start]]

    # return path if start is goal
    if start == goal:
      return "That was easy! Start = goal"
```

```
 # keeps looping until all possible paths have been checked
    while queue:
        # pop the first path from the queue
        path = queue.pop(0)
        # get the last node from the path
        node = path[-1]
        if node not in explored:
            neighbours = graph[node]

# go through all neighbour nodes, construct a new path and
            # push it into the queue

            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                # return path if neighbour is goal
                if neighbour == goal:
                    return new_path

            # mark node as explored
            explored.append(node)

    # in case there's no path between the 2 nodes
    return "So sorry, but a connecting path doesn't exist :("

bfs_shortest_path(graph, 'G', 'D') # returns ['G', 'C', 'A', 'B', 'D']

print(type(graph))
print('G', 'C', 'A', 'B', 'D')
```

Output:

```
<class 'dict'>

G C A B D
```

Time Complexity

The time complexity of the Breadth first Search algorithm is in the form of O(V+E), where V is the representation of the number of nodes and E is the number of edges. Also, the space complexity of the BFS algorithm is O(V).

Applications

Breadth-first Search Algorithm has a wide range of applications in the real-world. Some of them are as discussed below:

- In GPS navigation, it helps in finding the shortest path available from one point to another.
- In pathfinding algorithms
- Cycle detection in an undirected graph
- To build index by search index
- In Ford-Fulkerson algorithm to find maximum flow in a network.

Depth First Search

The Depth-First Search is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

DFS Algorithm

The recursive method of the Depth-First Search algorithm is implemented using stack. A standard Depth-First Search implementation puts every vertex of the graph into one in all 2 categories: 1) Visited 2) Not Visited.

The DSF algorithm follows as:

- Putting any one of the graph's vertex on top of the stack.
- After that take the top item of the stack and add it to the visited list of the vertex.
- Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
- Lastly, keep repeating steps 2 and 3 until the stack is empty.

DFS pseudocode

The pseudocode for Depth-First Search in python goes as below: In the init() function, notice that we run the DFS function on every node because many times, a graph may contain two different disconnected part and therefore to make sure that we have visited every vertex, we can also run the DFS algorithm at every node.

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
```

```
        if v.visited == false

            DFS(G,v)

init() {

    For each u ∈ G

        u.visited = false

     For each u ∈ G

       DFS(G, u)

}
```

DFS Implementation in Python (Source Code)

```python
# Using a Python dictionary to act as an adjacency list
graphs = {
 'A' : ['B','C'],
 'B' : ['D', 'E'],
 'C' : ['F'],
 'D' : [],
 'E' : ['F'],
 'F' : []
}
def dfs(graph, node):
    visited = [node]
    stack = [node]
    while stack:
        node = stack[-1]
        if node not in visited:
            visited.extend(node)
        remove_from_stack = True
        for next in graph[node]:
            if next not in visited:
                stack.extend(next)
                remove_from_stack = False
                break
        if remove_from_stack:
            stack.pop()
    return visited

print (dfs(graphs, 'A'))
```

Output

```
['A', 'B', 'D', 'E', 'F', 'C']
```

Time Complexity

The time complexity of the Depth-First Search algorithm is represented within the sort of

O(V+E), where V is that the number of nodes and E is that the number of edges.

The space complexity of the algorithm is O(V).

Applications

Depth-First Search Algorithm has a wide range of applications for practical purposes. Some of them are as discussed below:

- For finding the strongly connected components of the graph
- For finding the path
- To test if the graph is bipartite
- For detecting cycles in a graph
- Topological Sorting
- Solving the puzzle with only one solution.
- Network Analysis
- Mapping Routes
- Scheduling a problem

Uniform Cost Search

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.

Uniform Cost Search Algorithm

Below is the algorithm to implement Uniform Cost Search:

- Insert Root Node 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

Else insert all the children of removed elements into the queue with their cumulative cost as their priorities. Here root Node is the starting node for the path, and a priority queue is being maintained to maintain the path with the least cost to be chosen for the next traversal. In case 2 paths have the same cost of traversal, nodes are considered alphabetically.

UCS pseudocode

Uniform Cost Search is a type of uninformed search algorithm and an optimal solution to find the path from root node to destination node with the lowest cumulative cost in a weighted search space where each node has a different cost of traversal. It is similar to Heuristic Search, but no Heuristic information is being stored, which means h=0.

```
Insert the root into the queue

While the queue is not empty

    Dequeue the maximum priority element from the queue

    (If priorities are same, alphabetically smaller path is chosen)

    If the path is ending in the goal state, print the path and exit

    Else

        Insert all the children of the dequeued element, with the cumulative costs as priority
```

UCS Implementation in Python (Source Code)

```
#Uniform Cost Search Implementation
import queue as Q
def search(graph, start, end):
    if start not in graph:
       raise TypeError(str(start) + ' not found in graph !')
       return
    if end not in graph:
        raise TypeError(str(end) + ' not found in graph !')
        return

    queue = Q.PriorityQueue()
    queue.put((0, [start]))

    while not queue.empty():
          node = queue.get()
          current = node[1][len(node[1]) - 1]

          if end in node[1]:
               print("Path found: " + str(node[1]) + ", Cost = " + str(node[0]))
               break

          cost = node[0]
          for neighbor in graph[current]:
              temp = node[1][:]
              temp.append(neighbor)
              queue.put((cost + graph[current][neighbor], temp))

def readGraph():
     lines = int( input() )
     graph = {}

     for line in range(lines):
         line = input()

         tokens = line.split()
         node = tokens[0]
         graph[node] = {}
         print

         for i in range(1, len(tokens) - 1, 2):
             print(node, tokens[i], tokens[i + 1])
```

```
            #graph.addEdge(node, tokens[i], int(tokens[i + 1]))
            #graph[node][tokens[i]] = int(tokens[i + 1])
        return graph
print("How many input do you want to give?")
graph = readGraph()
search(graph, 'Arad', 'Zerind')

""" 14 Sample Map Input:

Arad Zerind 75 Timisoara 118 Sibiu 140
Zerind Oradea 71 Arad 75
Timisoara Arad 118 Lugoj 111
Sibiu Arad 140 Oradea 151 Fagaras 99 RimnicuVilcea 80
Oradea Zerind 71 Sibiu 151
Lugoj Timisoara 111 Mehadia 70
RimnicuVilcea Sibiu 80 Pitesti 97 Craiova 146
Mehadia Lugoj 70 Dobreta 75
Craiova Dobreta 120 RimnicuVilcea 146 Pitesti 138
Pitesti RimnicuVilcea 97 Craiova 138 Bucharest 101
Fagaras Sibiu 99 Bucharest 211
Dobreta Mehadia 75 Craiova 120
Bucharest Fagaras 211 Pitesti 101 Giurgiu 90
Giurgiu Bucharest 90"""
```

Output

```
How many input do you want to give?

14

Arad Zerind 75 Timisoara 118 Sibiu 140

Zerind Oradea 71 Arad 75

Timisoara Arad 118 Lugoj 111

Sibiu Arad 140 Oradea 151 Fagaras 99 RimnicuVilcea 80

Oradea Zerind 71 Sibiu 151

Lugoj Timisoara 111 Mehadia 70

RimnicuVilcea Sibiu 80 Pitesti 97 Craiova 146

Mehadia Lugoj 70 Dobreta 75

Craiova Dobreta 120 RimnicuVilcea 146 Pitesti 138

Pitesti RimnicuVilcea 97 Craiova 138 Bucharest 101
```

```
Fagaras Sibiu 99 Bucharest 211
Dobreta Mehadia 75 Craiova 120
Bucharest Fagaras 211 Pitesti 101 Giurgiu 90
Giurgiu Bucharest 90
➔ Arad Zerind 75
```

Time Complexity

Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε. Hence, the worst-case time complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.

Space Complexity: The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.

Optimal: Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Results

After doing the lab task and lab work we learn -BFS, DFS, UCS algorithm. Now we are able to do programs related to these topics

Discussion

Python is a general-purpose, versatile and popular programming language. It’s great as a first language because it is concise and easy to read, and it is also a good language to have in any programmer’s stack as it can be used for everything from web development to software development and data science applications.

This lab task is a great introduction to both fundamental programming concepts and the Python programming language. Python 3 is the most up-to-date version of the language with many improvements made to increase the efficiency and simplicity of the code that we write. Dayby-day, python new version are realising and all the new versions have new features and these new features are better than previous one. So, finally I can say that python will be more user friendly in future.