***Practical 2***

**Aim:** *Implement Iterative deep depth first search for Romanian map problem.*

**Theory:**

There are two common ways to traverse a graph, [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) and [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/). Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to following reasons.

1. **DFS** first traverses nodes going through one adjacent of root, then next adjacent. The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges).
2. **BFS** goes level by level, but requires more space. The space required by DFS is O(d) where d is depth of tree, but space required by BFS is O(n) where n is number of nodes in tree (Why? Note that the last level of tree can have around n/2 nodes and second last level n/4 nodes and in BFS we need to have every level one by one in queue).

**IDDFS** combines depth-first search’s space-efficiency and breadth-first search’s fast search (for nodes closer to root).

**How does IDDFS work?**   
IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

**Algorithm:**

// Returns true if target is reachable from

// src within max\_depth

bool IDDFS(src, target, max\_depth)

for limit from 0 to max\_depth

if DLS(src, target, limit) == true

return true

return false

bool DLS(src, target, limit)

if (src == target)

return true;

// If reached the maximum depth,

// stop recursing.

if (limit <= 0)

return false;

foreach adjacent i of src

if DLS(i, target, limit?1)

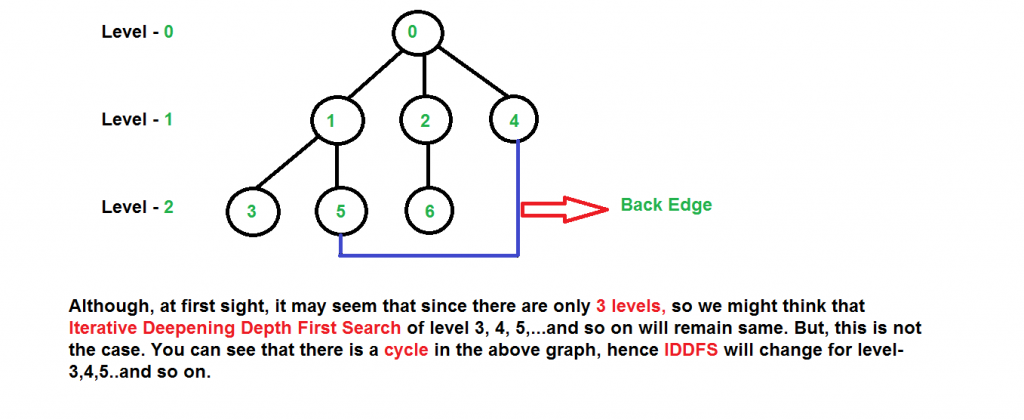
return true

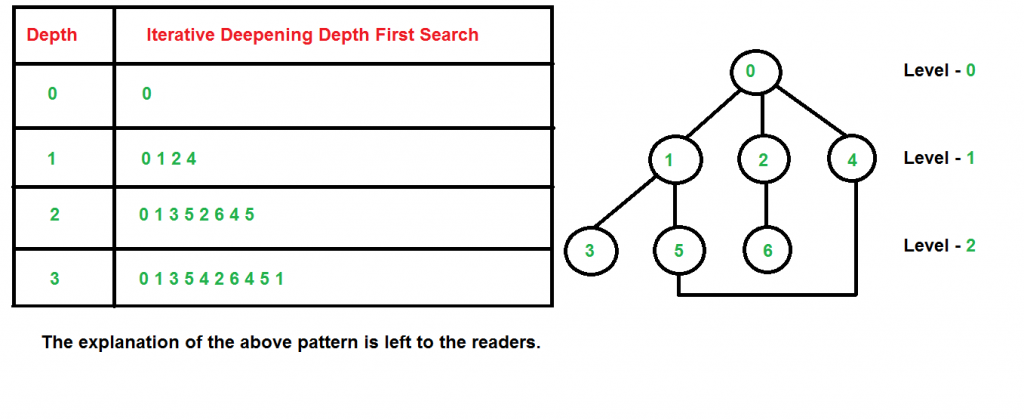
return false

An important thing to note is, we visit top level nodes multiple times. The last (or max depth) level is visited once, second last level is visited twice, and so on. It may seem expensive, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level. So it does not matter much if the upper levels are visited multiple times.

**Illustration:**There can be two cases:

1. ***When the graph has no cycle:*** This case is simple. We can DFS multiple times with different height limits.
2. ***When the graph has cycles.*** This is interesting as there is no visited flag in IDDFS.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/iddfs11.png)

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/iddfs2.png)

**Time Complexity:**Suppose we have a tree having branching factor ‘b’ (number of children of each node), and its depth ‘d’, i.e., there are **bd** nodes. In an iterative deepening search, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree, which is expanded d+1 times. So the total number of expansions in an iterative deepening search is-

(d)b + (d-1)b2 + .... + 3bd-2 + 2bd-1 + bd

That is,

Summation[(d + 1 - i) bi], from i = 0 to i = d

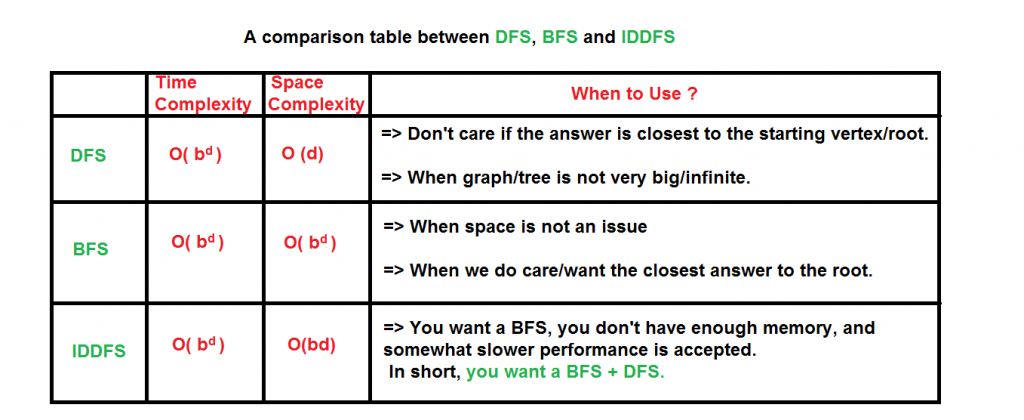
Which is same as O(bd)

After evaluating the above expression, we find that asymptotically IDDFS takes the same time as that of DFS and BFS, but it is indeed slower than both of them as it has a higher constant factor in its time complexity expression. IDDFS is best suited for a complete infinite tree

**Complexity:**

**Time:**O(bd)

**Space:**O(d)

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/iddfs4.png)

***Code:***

dict\_graph = {}

# Read the data.txt file

with open('data.txt', 'r') as f:

    for l in f:

        city\_a, city\_b, p\_cost = l.split()

        if city\_a not in dict\_graph:

            dict\_graph[city\_a] = {}

        dict\_graph[city\_a][city\_b] = int(p\_cost)

        if city\_b not in dict\_graph:

            dict\_graph[city\_b] = {}

        dict\_graph[city\_b][city\_a] = int(p\_cost)

# Iterative Deepening Search Method

def IterativeDeepening(graph, src, dst):

    level = 0

    count = 0

    stack = [(src, [src], 0)]

    visited = {src}

    while True:

        level += 1

        while stack:

            if count <= level:

                count = 0

                (node, path, cost) = stack.pop()

                for temp in graph[node].keys():

                    if temp == dst:

                        return path + [temp], cost + graph[node][temp]

                    else:

                        if temp not in visited:

                            visited.add(temp)

                            count += 1

                            stack.append((temp, path + [temp], cost + graph[node][temp]))

            else:

                q = stack

                visited\_bfs = {src}

                while q:

                    (node, path, cost) = q.pop(0)

                    for temp in graph[node].keys():

                        if temp == dst:

                            return path + [temp], cost + graph[node][temp]

                        else:

                            if temp not in visited\_bfs:

                                visited\_bfs.add(temp)

                                q.append((temp, path + [temp], cost + graph[node][temp]))

                break

print(dict\_graph)

print("------------------------------------------------")

#src = raw\_input("Enter the source:")

#dst = raw\_input("Enter the Destination: ")

src = "Oradea"

dst = "Iasi"

print("for ID")

print (IterativeDeepening(dict\_graph, src, dst))

data.txt

Oradea Zerind 71

Oradea Sibiu 151

Zerind Arad 75

Arad Sibiu 140

Arad Timisoara 118

Timisoara Lugoj 111

Lugoj Mehadia 70

Mehadia Drobeta 75

Drobeta Craiova 120

Sibiu Rimnicu\_Vilcea 80

Sibiu Fagaras 99

Rimnicu\_Vilcea Piteshi 97

Rimnicu\_Vilcea Craiova 146

Craiova Piteshi 138

Piteshi Bucharest 101

Fagaras Bucharest 211

Bucharest Giurgiu 90

Bucharest Urziceni 85

Urziceni Hirsova 98

Urziceni Vaslui 142

Hirsova Eforie 86

Vaslui Iasi 92

Neamt Iasi 87

**Output:**

Oradea to Iasi



***Conclusion:***

*Implemented Iterative deep depth first search for Romanian map problem.*