Department of
Computer Science and Engineering

Title: Iterative Deepening Depth-First Search(IDDFS)

Artificial Intelligence Lab

CSE 404

Green University of Bangladesh

1 Objective(s)

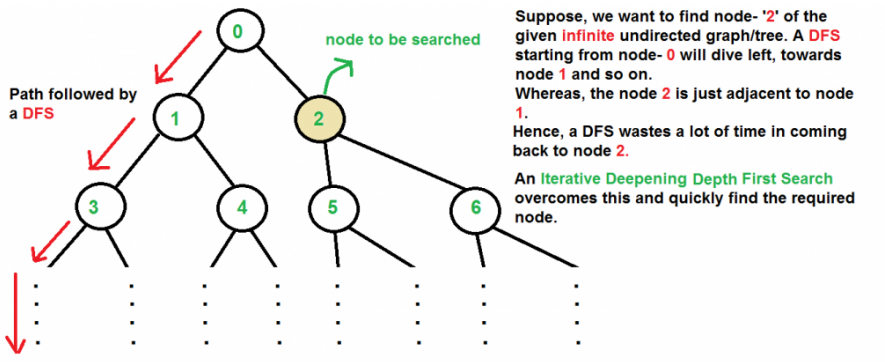
- To understand how to represent a graph using an adjacency list.
- To understand how Iterative Deepening Depth-First Search(DIDFS) works.

2 Problem analysis

Iterative deepening search or more specifically iterative deepening depth-first search (IDS or IDDFS) is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found. IDDFS is optimal like breadth-first search, but uses much less memory; at each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

There are two common ways to traverse a graph, BFS and DFS. Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to following reasons.

- 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).



- 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).

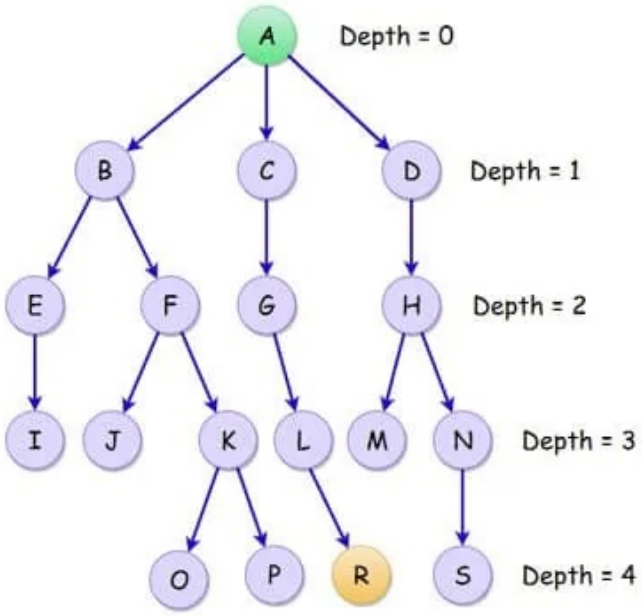
As a general rule of thumb, we use iterative deepening when we do not know the depth of our solution and have to search a very large state space. Iterative deepening may also be used as a slightly slower substitute for BFS if we are constrained by memory or space.

3 Algorithm

```
1  // Returns true if target is reachable from
2  // src within max_depth
3  bool IDDFS(src, target, max_depth)
4      for limit from 0 to max_depth
5         if DLS(src, target, limit) == true
6             return true
7      return false
8
9  bool DLS(src, target, limit)
10     if (src == target)
11         return true;
```

```
12
13      // If reached the maximum depth,
14      // stop recursing.
15      if (limit <= 0)
16          return false;
17
18      foreach adjacent i of src
19          if DLS(i, target, limit?1)
20              return true
21
22      return false
```

4 Example



Here in the given tree, the starting node is A and the depth initialized to 0. The goal node is R where we have to find the depth and the path to reach it. The depth from the figure is 4. In this example, we consider the tree as a finite tree, while we can consider the same procedure for the infinite tree as well. We knew that in the algorithm of IDDFS we first do DFS till a specified depth and then increase the depth at each loop. This special step forms the part of DLS or Depth Limited Search. Thus the following traversal shows the IDDFS search.

The tree can be visited as: A B E F C G D H

DEPTH = {0, 1, 2, 3, 4}

DEPTH LIMITS	IDDFS
0	A
1	A B C D
2	A B E F C G D H
3	A B E I F J K C G L D H M N
4	A B E I F J K O P C G L R D H M N S

5 Implementation in Java

```
1  package iterativedeepening;
2
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5  import java.util.Stack;
6
7  public class IterativeDeepening
8  {
9      private Stack<Integer> stack;
10     private int numberOfNodes;
11     private int depth;
12     private int maxDepth;
13     private boolean goalFound = false;
14
15     public IterativeDeepening()
16     {
17         stack = new Stack<Integer>();
18     }
19
20     public void iterativeDeeping(int adjacencyMatrix[][], int destination)
21     {
22         numberOfNodes = adjacencyMatrix[1].length - 1;
23         while (!goalFound)
24         {
25             depthLimitedSearch(adjacencyMatrix, 1, destination);
26             maxDepth++;
27         }
28         System.out.println("\nGoal Found at depth " + depth);
29     }
30
31     private void depthLimitedSearch(int adjacencyMatrix[][], int source, int
           goal)
32     {
33         int element, destination = 1;
34         int[] visited = new int[numberOfNodes + 1];
35         stack.push(source);
36         depth = 0;
37         System.out.println("\nAt Depth " + maxDepth);
38         System.out.print(source + "\t");
39
40         while (!stack.isEmpty())
41         {
42             element = stack.peek();
43             while (destination <= numberOfNodes)
44             {
45                 if (depth < maxDepth)
46                 {
47                     if (adjacencyMatrix[element][destination] == 1)
48                     {
49                         stack.push(destination);
50                         visited[destination] = 1;
51                         System.out.print(destination + "\t");
52                         depth++;
53                         if (goal == destination)
54                         {
55                             goalFound = true;
```

```
                            return;
                        }
                        element = destination;
                        destination = 1;
                        continue;
                    }
                } else
                {
                    break;
                }
                destination++;
            }
            destination = stack.pop() + 1;
            depth--;
        }
    }

    public static void main(String... arg)
    {
        int number_of_nodes, destination;
        Scanner scanner = null;
        try
        {
            System.out.println("Enter the number of nodes in the graph");
            scanner = new Scanner(System.in);
            number_of_nodes = scanner.nextInt();

            int adjacency_matrix[][] = new int[number_of_nodes + 1][
                number_of_nodes + 1];
            System.out.println("Enter the adjacency matrix");
            for (int i = 1; i <= number_of_nodes; i++)
                for (int j = 1; j <= number_of_nodes; j++)
                    adjacency_matrix[i][j] = scanner.nextInt();

            System.out.println("Enter the destination for the graph");
            destination = scanner.nextInt();

            IterativeDeepening iterativeDeepening = new IterativeDeepening();
            iterativeDeepening.iterativeDeeping(adjacency_matrix, destination);
        }catch (InputMismatchException inputMismatch)
        {
            System.out.println("Wrong Input format");
        }
        scanner.close();
    }
}
```

6 Implementation in Python

```
class IterativeDeepening:
    def __init__(self):
        self.stack = []
        self.numberOfNodes = 0
        self.depth = 0
        self.maxDepth = 0
        self.goalFound = False

```

```
    def iterativeDeepening(self, adjacencyMatrix, destination):
        self.numberOfNodes = len(adjacencyMatrix) - 1
        while not self.goalFound:
            self.depthLimitedSearch(adjacencyMatrix, 1, destination)
            self.maxDepth += 1
            if self.goalFound:
                print("\nGoal Found at depth", self.depth)
                return
            self.depth = 0
            self.stack = []

    def depthLimitedSearch(self, adjacencyMatrix, source, goal):
        visited = [0] * (self.numberOfNodes + 1)
        self.stack.append(source)
        print("\nAt Depth", self.maxDepth)
        print('\n', source, end='\t')

        while self.stack:
            element = self.stack[-1]
            found = False
            for destination in range(1, self.numberOfNodes + 1):
                if self.depth < self.maxDepth:
                    if adjacencyMatrix[element][destination] == 1 and visited[
                        destination] == 0:
                        visited[destination] = 1
                        self.depth += 1
                        self.stack.append(destination)
                        print(destination, end='\t')
                        if destination == goal:
                            self.goalFound = True
                            return
                        found = True
                        break

            if not found:
                self.stack.pop()
                self.depth -= 1

if __name__ == "__main__":
    try:
        print("Enter the number of nodes in the graph\n")
        number_of_nodes = int(input().strip())

        adjacency_matrix = [[0] * (number_of_nodes + 1) for _ in range(
            number_of_nodes + 1)]
        print("Enter the adjacency matrix\n")
        for i in range(1, number_of_nodes + 1):
            row = list(map(int, input().strip().split()))
            for j in range(1, number_of_nodes + 1):
                adjacency_matrix[i][j] = row[j - 1]

        print("Enter the destination for the graph\n")
        destination = int(input().strip())

        iterativeDeepening = IterativeDeepening()
        iterativeDeepening.iterativeDeepening(adjacency_matrix, destination)
    except ValueError:
        print("Wrong Input format")
```

7 Sample Input/Output (Compilation, Debugging & Testing)

```
Enter the number of nodes in the graph
7
Enter the adjacency matrix
0 1 1 0 0 0 0
0 0 0 1 1 0 0
0 0 0 0 0 1 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
Enter the destination for the graph
7
At Depth 0
1
At Depth 1
1 2 3
At Depth 2
1 2 4 5 3 6 7
Goal Found at depth 2
```

8 Lab Task (Please implement yourself and show the output to the instructor)

1. Write a program to perform IDDFS traversal on a dynamic graph input from the user.
2. Write a program to find the path from source to destination using IDDFS.

9 Lab Exercise (Submit as a report)

- Write a program to perform topological search using IDDFS.

10 Policy

Copying from internet, classmate, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.