ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 336E

Analysis of Algorithms II
HOMEWORK REPORT

HOMEWORK NO	:	1
HOMEWORK DATE	:	28.03.2023

150200915 : ADIL MAHMUDLU

Sprint 2023

Contents

1 EXPLANATION OF THE CODE AND THE SOLUTION

The functions used in the code are graphFile, bfs, nextChild, and dfs. There are some calculations in the main function as well that are worth mentioning, such as creation of graph (the adjecency matrix). The graphFile function is simply creating and filling the graph.txt file in O(n^2) time complexity.

1.1 Creation of graph in main Function

```
graph <- integer array(n, n) of 0s
x <- integer array(n)
y <-integer array(n)
p <- integer array(n)
for kid1 from 0 to n:
    for kid2 from 0 to kid1:
        distance <- (x[kid1] - x[kid2]) ** 2 + (y[kid1] - y[kid2]) ** 2
        if distance <= p[kid1] AND distance <= p[kid2]:
            graph[kid1][kid2] <- 1;
            graph[kid2][kid1] <- 1;
```

The pseudocode above creates adjecency matrix in $O(n^2)$ complexity. Each kid pair is only considered once, as one kid's sending possibility is dependent on the receiver. Therefore, it actually does $T(n^2/2)$ calculations. Space complexity is $O(n^2)$ for the graph.

1.2 bfs Function

```
function bfs(graph, n, source, target)
    q <- integer queue
    visited <- integer array(n) of 0s
    distance = integer array(n) of -1s
    previous = integer array(n) of -1s
    q.push(source)
    visited[source] <- 1
    distance[source] <- 0
    while q is not empty:
        current <- q.pop()
        for i from 0 to n:
            if graph[current][i] and not visited[i]:
```

```
            q.push(i)
            visited[i] <- 1
            distance[i] <- distance[current] + 1
            previous[i] <- current
            if i == target:
                bfsOutFile <- create filestream object "bfs.txt" for writing
                bfsOutFile.write("BFS:\n" + distance[i] + " " + source)
                passSeq <- integer stack
                current <- target
                while current != source:
                    passSeq.push(current)
                    current <- previous[current]
                while not passSeq.empty():
                    bfsOutFile.write("->" + passSeq.pop())
                bfsOutFile.close()
                return
    return
```

The function bfs calculates the shortest path to the target kid and creates and writes to the file bfs.txt. It does breadth first search, and therefore requires queue. The time complexity of the function depends on the graph G(V, E) where V is number of kids and E is edges, or possible passes, between them. The time complexity then is $O(V + E)$ and space complexity is $O(V)$.

1.3 dfs Function

```
function dfs(graph, n, source)
    visited <- integer array(n) of 0s
    previous <- integer array(n) of -1s
    passSeq <- integer array
    dfsOutFile <- create filestream object "dfs.txt" for writing
    dfsOutFile.write("DFS:\n")
    if not nextChild(source, graph, n, source, visited, previous, passSeq):
        dfsOutFile.write("-1 ")
        dfsOutFile.close()
        return
    dfsOutFile.write(passSeq.size() + " ")
    for i from passSeq.size() - 1 down to 0:
        dfsOutFile.write(passSeq[i] + "->")
    dfsOutFile.write(source)
    dfsOutFile.close()
    return
```

The dfs function calls the recursive nextChild function for finding the cycle. If the nextChild function returns false, no cycle is found and -1 is written in the file created by the dfs function. Otherwise, file is filled with the sequence of kids. The time Complexity of the function itself is $O(n)$ and space complexity is $O(n)$ as well, but it contains a recursive function which has higher time and space complexity.

1.4 nextChild Function

```
nextChild(current, graph, n, source, visited, previous, passSeq) {
    visited[current] <- 1
    for i from 0 to n:
        if graph[current][source] == 1 AND previous[current] != source:
            while current != source:
                passSeq.push(current)
                current <- previous[current]
            passSeq.push(source)
            return true
        if graph[current][i] AND !visited[i]:
            previous[i] <- current
            if nextChild(i, graph, n, source, visited, previous, passSeq):
                return true
    return false
```

The recursive function nextChild calls itself for a depth first search of source kid, trying to find a cycle. The function returns back to dfs function with true if found or false if cycle is not found. The time complexity is depends on the graph G(V, E), just like bfs function. For V kids and E edges, or passes between them, the time complexity is $O(V + E)$ while the space complexity is $O(V)$, or $O(n)$.

2 Why should you maintain a list of discovered nodes? How does this affects the outcome of the algorithms?

For both the breadth first and depth first search algorithms, the newly found nodes must be kept for later search. These algorithms depend on the graph connections to traverse and search for the target and the nodes they traverse are key components of these graphs that allow these graphs to branch through the graph. Breadth First Search algorithm searches in lowest depth levels before going deep into the graph. Therefore, it requires queue of nodes to search. The queue makes sure the low depth leveled nodes are searched before the high depth leveled nodes and newly discovered nodes in higher depth levels are added to the end of the queue. This way, the nodes are searched in the order they are found. Depth First Search algorithm searcher one branch to the maximum depth before moving on to the next branch of the graph. Therefore, it needs stack of nodes to search. Stack makes sure the high depth leveled nodes are searched before low depth leveled nodes. The nodes are not searched in the order they are found, but rather by their depth. Stack makes it easy to go back in the branch and try a different branch for the search.

3 How does increasing the number of the kids affects the memory complexity of the algorithms?

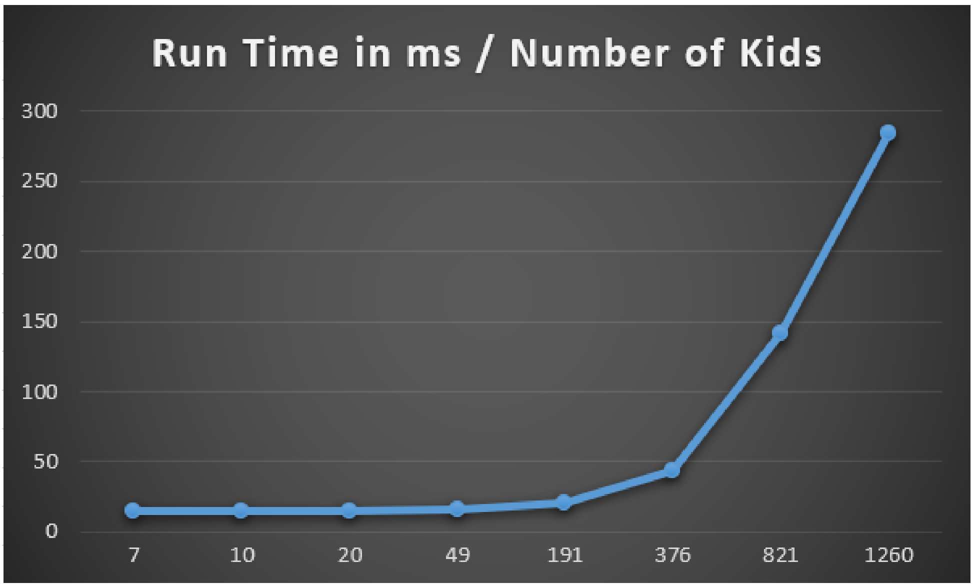

Figure 1: Line Graph of Runtime in ms/Number of Kids

The space complexity of the program only depends on the number of kids. The program keeps the adjecency matrix of the kids and therefore has space complexity of $O(n^2)$.