Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

BLG 336E
The LaTeX
Report

Analysis of Algorithms II, Project 1

Leminur Çelik - 150190085

March 28$^{\text{th}}$, 2023

Contents

1 Description of Code

1.1 Pseudo-code

1.1.1 Graph Implementation

I implement the code which creates (n x n) adjacency matrix as a graph. If the squared distance between i-th vertex and j-th vertex is less then or equal to the strength of i-th and j-th, then there is an edge between two vertices.

Algorithm 1 Set Edges

```
function SetEdges(*nodes, numberVertices)
    i ← 0
    while i < numberVertices do
        while j < numberVertices do
            if i ← j then
                adjMatrix[i][j] ← 0
            end if
            x − distance ← nodes[i].GET-X-COORDINATE − nodes[j].GET-X-COORDINATE()
            y − distance ← nodes[i].GET-Y-COORDINATE − nodes[j].GET-Y-COORDINATE()
            squared − distance ← POW(x − distance, 2) + POW(y − distance, 2)
            if squared−distance ≤ nodes[i].GET-STRENGTH-LEVEL and squared−distance ≤
nodes[j].GET-STRENGTH-LEVEL then
                adjMatrix[i][j] ← 1
                adjMatrix[j][i] ← 1
            end if
        end while
    end while
end function
```

1.1.2 Breadth First Search Implementation

Graph traversed using Breadth First Search(BFS) algorithm to find the minimum amount of pass from the starting point to the target point.

Algorithm 2 Breadth First Search

function BREADTHFIRSTSEARCH(numberVertices, sourceVertex, distance, predecessor)
    $visited \leftarrow$ vector of bool with number of vertices elements, initialized to false
    $q \leftarrow$ queue of int
    q.PUSH(sourceVertex)
    $visited[sourceVertex] \leftarrow true$
    $distance[sourceVertex] \leftarrow 0$
    while $!q.\text{EMPTY}()$ **do**
        $visitedVertex \leftarrow q.\text{FRONT}()$
        $q.\text{POP}()$
        $i \leftarrow 0$
        while $i < numberVertices$ **do**
            if $adjMatrix[visitedVertex][i] \leftarrow 1$ *and* $(!visited[i])$ **then**
                $q.\text{PUSH}(i)$
                $distance[i] \leftarrow distance[visitedVertex] + 1$
                $predecessor[i] = visitedVertex$
                $visited[i] = true$
            end if
        end while
    end while
end function

1.1.3 Depth First Search Implementation

It was tested whether the graph contains a cycle starting from the source vertex and returning to this vertex using Depth First Search(DFS) algorithm. If it contains this cycle, the path traversed is saved.

Algorithm 3 Depth First Search

function CONTAINSCYCLE(vertex, visited, predecessor, savedPath, sourceVertex, numberVertices)
    $visited[vertex] \leftarrow true$
    $i \leftarrow 0$
    while $i < numberVertices$ **do**
        if $adjMatrix[sourceVertex][vertex] \leftarrow 1$ *and* $(predecessor[vertex]! = sourceVertex)$ **then**
            return true
        end if
        if $adjMatrix[i][vertex] \leftarrow 1$ *and* $(!visited[i])$ **then**
            $predecessor[i] \leftarrow vertex$
            if CONTAINSCYCLE(i, visited, predecessor, savedPath, sourceVertex, numberVertices) **then**
                return true
            end if
        end if
    end while
    return false
end function

Algorithm 4 Depth First Search

function DEPTHFIRSTSEARCH(numberVertices, sourceVertex, start, distance, predecessor)
  $visited \leftarrow$ vector of bool with number of vertices elements, initialized to false
  $predecessor \leftarrow$ vector of int with number of vertices, initialize to -1
  $savedPath \leftarrow$ vector of int
  $visited[sourceVertex] \leftarrow true$
  $i \leftarrow 0$
  while $i < numberVertices$ **do**
    if $adjMatrix[sourceVertex][i] \leftarrow 1$ *and* $(!visited[i])$ **then**
      $predecessor[i] = sourceVertex$
      if CONTAINSCYCLE(i, visited, predecessor, savedPath, sourceVertex, numberVertices) **then**
        print path
      end if
      if (!CONTAINSCYCLE(i, visited, predecessor, savedPath, sourceVertex, numberVertices)) **then**
        print -1
      end if
    end if
  end while
end function

1.2 Time Complexity

The graph has n vertices, the time complexity to build an adjacency matrix is $O(n^2)$. Finding the adjacent vertices of selected vertex requires checking all elements in the row. This takes linear time O(n). Summing over all the n iterations, the total running time is $O(n^2)$. In the Breadth First Search function, the for loop loops n times, and the while loop loops n times depending on this number, resulting in n^2. In the DFS algorithm, it returns n times in the for loop in the Depth First Search function and calls the Contains Cycle function, and inside that function it returns n times in the for loop, so it becomes n^2. Therefore, time complexity of BFS and DFS are $O(n^2)$.

2 Maintain a List of Discovered Nodes

By saving visited nodes, we eliminate the possibility of being visited again. Each node can be visited once. Otherwise, we could traverse the same node multiple times. If we do not mark a node as visited, it will go to another node and come back. This creates a loop. Therefore, the BFS and DFS algorithms will not terminate. We need to keep a visited array for the algorithms to work correctly.

3 Relationship between Number of Kids and Memory Complexity

3.1 Space Complexity

The adjacency matrix of a graph requires $O(V^2)$ memory where V is the number of vertices. The space complexity of BFS and DFS can be expressed as O(V), where V is the number of vertices, because when finding shortest path I used a queue that takes the number of vertices and when finding a cycle I used a vector that takes the number of vertices. Increasing the number of kids means increasing the number of vertices. Creating an adjacency matrix is proportional to the square of the number of vertices. Since the space complexity of BFS and DFS depend on the vertex number, the space complexity of the BFS and DFS increases linearly.

3.2 Run-Time

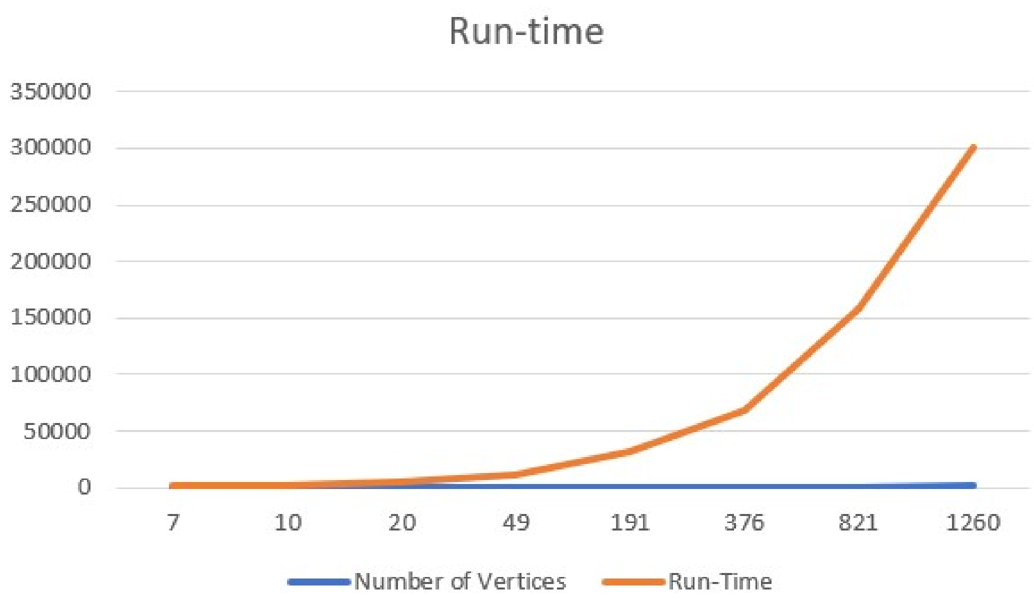

Figure 1: The graph for run-time in microseconds for all cases

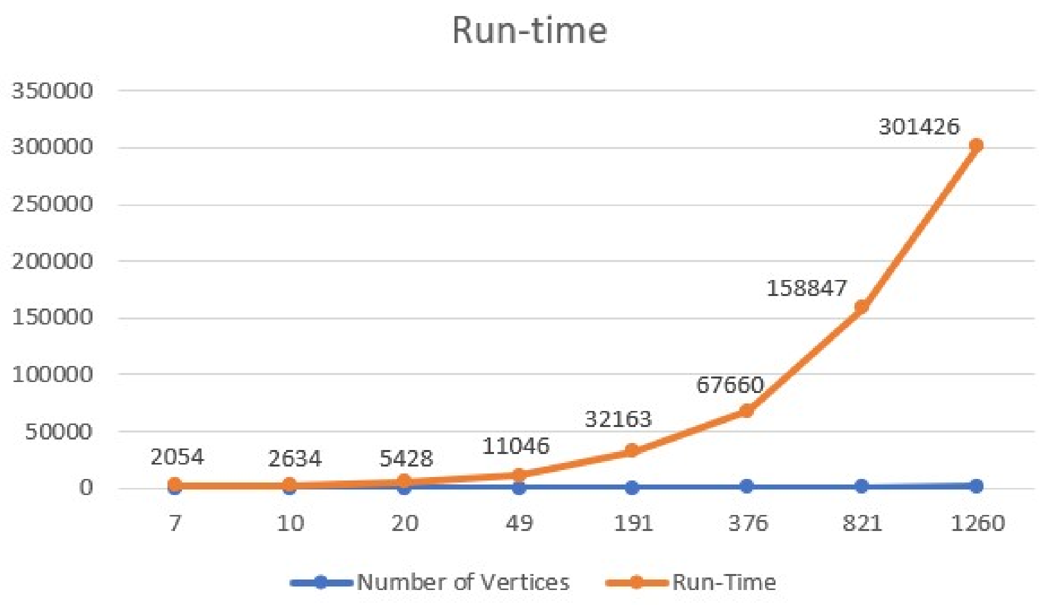

Figure 2: The graph for run-time in microseconds with labels for all cases