Hudson Cho, Ryan Wilson, Jesse Washburn, Colin Shuster, Samhith Patibandla
COSC 336
05/6/2025

Assignment 8

Instructions.

1. Due date announced on Blackboard.

2. This is a team assignment. Work in teams as in the previous assignments. Submit one assignment per team, with the names of all students making the team.

3. Your programs must be written in Java.

4. Write your programs neatly - imagine yourself grading your program and see if it is easy to read and understand. At the very beginning present your algorithm in plain English or in pseudo-code (or both). Comment your programs reasonably: there is no need to comment lines like "i++" but do include brief comments describing the main purpose of a specific block of lines.

5. You will submit on Blackboard 2 files. The first file should be a .pdf file with the solutions of the Exercise and a short description in English or in pseudocode of the algorithm for the programming task you are required to do and the results that you are required to report. Also include in this file, the Java code so that the grader can make comments.

   Files 2 will contain the java code for the programming task.

   For editing the above document with Latex, see the template posted on the course website.

   assignment-template.tex and

   assignment-template.pdf

   To append in the latex file a pdf file, place it in the same folder and then include them in the latex file with

   ```
   \includepdf[pages=-,pagecommand={},width=\textwidth]{file.pdf}
   ```

   To append in the latex file a .jpg file (for a photo), use

   ```
   \includegraphics[width=\linewidth]{file.jpg}
   ```

Exercise 1.

We have seen that Dijkstra's algorithm can be implemented in two ways: Variant (a) uses an array to store the $dist[]$ values of the unknown nodes, and Variant (b) uses a MIN-HEAP to store these values.

(a) Suppose in your application $m \leq 3n$. Which variant gives a faster runtime? Justify your answer.

(b) Suppose in your application $m \geq n^2/3$. Which variant gives a faster runtime? Justify your answer.

(c) Suppose that your application $m = n^{3/2}$. Which variant gives a faster runtime? Justify your answer.

Solution:

(a) When $m \leq 3n$, use the array-based variant of Dijkstra. Its runtime is $O(n^2)$ which is efficient enough for sparse graphs.

(b) When $m \geq \frac{n^2}{3}$, the graph is dense and the array-based version ($O(n^2)$) is still better than the heap version ($O((n+m)\log n)$).

(c) When $m = n^{3/2}$, the heap version is better: $O((n+n^{3/2})\log n) = O(n^{3/2}\log n)$ which grows slower than $O(n^2)$.

Exercise 2. Recall that when we do DFS with timing every node u gets 2 numbers that were denoted $u.d$ and $u.f$. $u.d$ is the discovery time and $u.f$ is the finish time.

Show that in a DAG (directed acyclic graph), for any two nodes u and v such that there exists a path from u to v, it holds that $u.f > v.f$.

Hint: There are two cases to analyze. Case 1 is that $u.d < v.d$ (in words, u is discovered before v), and Case 2 is that $v.d < u.d$ (so, v is discovered before u). In both cases, you need to argue that $u.f > v.f$. □

Solution: Assume a path exists from u to v in a DAG. We analyze two cases:

- **Case 1:** $u.d < v.d$. Since v is reachable from u, DFS will finish exploring v and its descendants before returning to finish u. So $u.f > v.f$.
- **Case 2:** $u.d > v.d$. Since u is not discovered from v (it would imply a back edge, which contradicts acyclicity), DFS finishes v before discovering and finishing u, so $u.f > v.f$.

Therefore, in all cases, if $u \to v$, then $u.f > v.f$. □

Programming Task

Write the program that modifies Breadth First Search (see for example the basic version of BFS in Notes 10) in such a way that given an undirected connected graph G, and a starting node s, it will print for every node v the length of the shortest path from s to v and also the number of shortest paths from s to v. Thus, you'll have two arrays $dist$ and $npath$, and for each vertex v, at the end of the program, $dist[v]$ will be equal to the length of a shortest path from s to v (like in the basic version of BFS), and $npath[v]$ will be equal to the number of shortest paths from s to v.

For example, if G_1 is the first graph below, the length of a shortest path from 1 to 7 is 3, and there are three shortest paths from 1 to 7, namely

1 - 2 - 5 - 7,

1 - 3 - 5 - 7 and

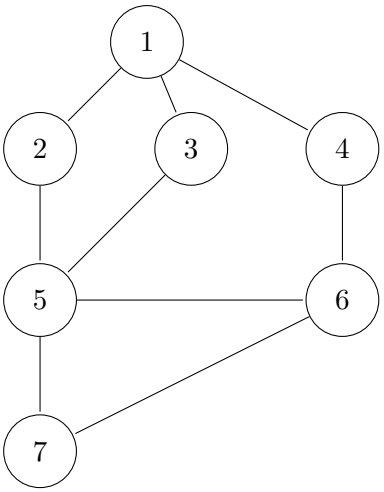
1 - 4 - 6 -7.

So, $dist[7] = 3$, and $npath[7] = 3$.

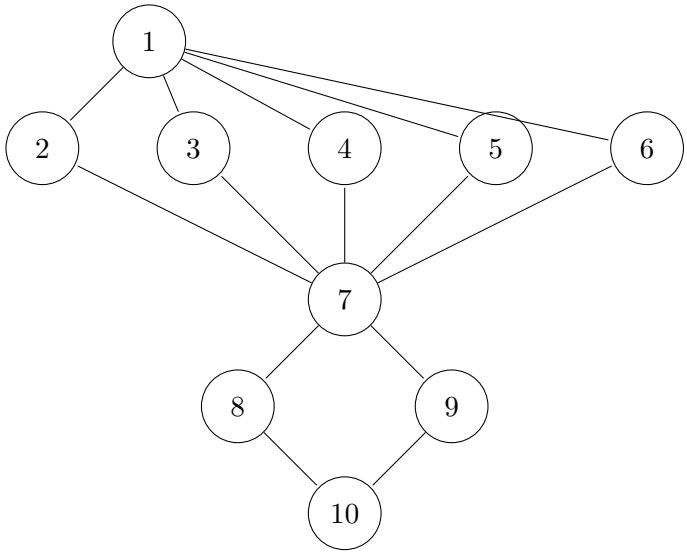
Test your program on the graphs G_1 and G_2 (see the figures) using 1 as the starting node. Your program is required to print the two arrays $dist[]$ and $npath[]$. Report the results you have obtained for these two graphs.

You must use the adjacency list representation of a graph. As in the programming task from assignment 7, for the adjacency list, you **must** use the Java class Adj_List_Graph given in the file Adj_List_Graph.java (see Test_Adj.java for a very simple example of using this class). Note that, unlike Assignment 7, in this assignment you work with undirected graphs. So, in Adj_List_Graph.java, you need to uncomment a line to make it work for undirected graphs.

graph G_1



graph G_2:



Programming Task Solution:

Code Description

This Java program implements a class called `Adj_List_Graph` to represent an undirected graph using an adjacency list. The key method, `shortestPath(int start)`, performs a modified Breadth-First Search (BFS) beginning at a specified start node. It uses two arrays:

`distance[]` to store the shortest path length from the start node to every other node, and `nPaths[]` to count the number of such shortest paths. As the BFS progresses, if a neighbor is reached via an equally short path, its path count is incremented; if a shorter path is found, the distance and path count are updated. The graph is built by reading a flattened adjacency matrix from a file, and edges are added only once since the graph is undirected. After traversal, the program outputs the contents of both the `distance[]` and `nPaths[]` arrays.

Graph G1 Results (Start: Node 1)

Vertex	Distance	# Paths
1	0	1
2	1	1
3	1	1
4	1	1
5	2	2
6	2	1
7	3	3

Graph G2 Results (Start: Node 1)

Vertex	Distance	# Paths
1	0	1
2	1	1
3	1	1
4	1	1
5	1	1
6	1	1
7	2	5
8	3	5
9	3	5
10	4	10

Raw Code for Programming Task

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Scanner;

public class Adj_List_Graph {
    int n;
    ArrayList<ArrayList<Integer>> adj;

    Adj_List_Graph(int num_nodes) {
        n = num_nodes;
        adj = new ArrayList<>(n);
        for (int i = 0; i < n; i++) {
            adj.add(new ArrayList<>());
        }
    }

    public void shortestPath(int start) {
        LinkedList<Integer> queue = new LinkedList<>();
        int[] distance = new int[n];
        int[] nPaths = new int[n];

        for (int i = 0; i < n; i++) {
            distance[i] = Integer.MAX_VALUE;
            nPaths[i] = 0;
        }

        distance[start] = 0;
        nPaths[start] = 1;
        queue.add(start);

        while (!queue.isEmpty()) {
            int curr = queue.poll();

            for (int neighbor : adj.get(curr)) {
                if (distance[neighbor] == distance[curr] + 1) {
                    nPaths[neighbor] += nPaths[curr];
                } else if (distance[neighbor] > distance[curr] + 1) {
                    distance[neighbor] = distance[curr] + 1;
                    nPaths[neighbor] = nPaths[curr];
                    queue.add(neighbor);
                }
            }
        }

        System.out.print("Distances[] = ");
        for (int d : distance) {
            System.out.print((d < Integer.MAX_VALUE ? d : "-") + " ");
        }
        System.out.println();

        System.out.print("Npaths[] = ");
        for (int p : nPaths) {
            System.out.print(p + " ");
        }
```

```
57          System.out.println();
58      }
59
60      public void addEdge(int u, int v) {
61          adj.get(u).add(v);
62          adj.get(v).add(u);
63      }
64
65      public static Adj_List_Graph file_Intake() throws FileNotFoundException
            {
66          Scanner scnr = new Scanner(System.in);
67          System.out.print("Input File Name: ");
68          String userVar = scnr.next();
69
70          while (!new File(userVar).exists()) {
71              System.out.println("File not found. Try again.");
72              System.out.print("Input File Name: ");
73              userVar = scnr.next();
74          }
75
76          Scanner fileScanner = new Scanner(new File(userVar));
77          int num_nodes = fileScanner.nextInt();
78          Adj_List_Graph graph = new Adj_List_Graph(num_nodes);
79
80          int k = 0;
81          while (fileScanner.hasNextInt()) {
82              int from = k / num_nodes;
83              int to = k % num_nodes;
84              int val = fileScanner.nextInt();
85              if (val == 1 && from < to) {
86                  graph.addEdge(from, to);
87              }
88              k++;
89          }
90
91          return graph;
92      }
93  }
```

Raw Code for Driver

```
1
2   public class Test_Adj {
3       public static void main(String[] args) throws FileNotFoundException {
4
5           // Prompt user to enter file name, read the graph from file
6           // Builds the adjacency list representation for Graph G1
7           Adj_List_Graph G1 = Adj_List_Graph.file_Intake();
8           G1.shortestPath(0);
9
10          // Prompt again for the second input file
11          // Builds the adjacency list for Graph G2
12          Adj_List_Graph G2 = Adj_List_Graph.file_Intake();
13          G2.shortestPath(0);
14      }
15  }
```