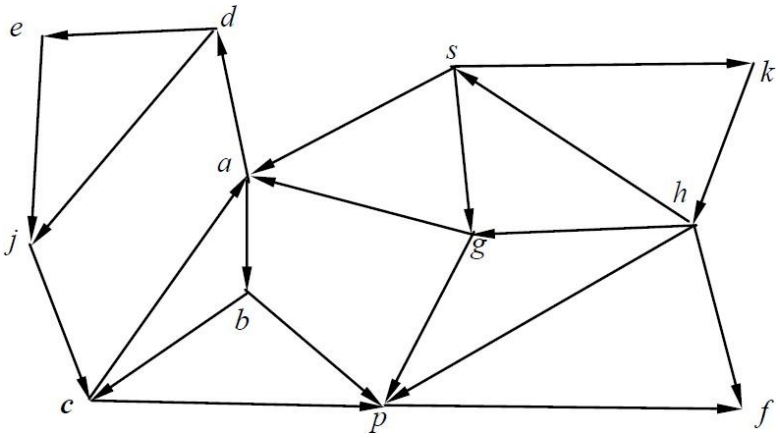
Due Date: 26th Nov 2023
20% penalty for 1 day late
40% penalty for 2 days late
Submission not allowed afterwards

CS2009: Design and Analysis of Algorithms (Fall 2023)
Assignment 4
Total Marks: 100

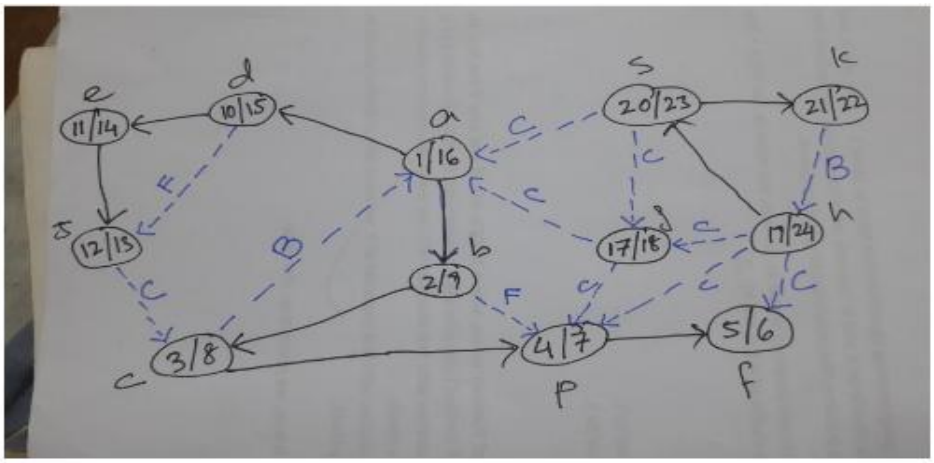
1. a) Conduct a DFS for the following graph. (Label each vertex u with the start time and the finish time) or (Show all steps showing stack and Visit Order). You should start the traversal from vertex a, and follow the alphabetic order whenever you need to make choices. [10 Points]



b) List all edges (back edges,forward edges, cross edges) that belong to each of the following sets: [5 Points]

c) Identify the strongly connected components and draw the component graph. [10 Points]

Solution



b) List all edges (back edges,forward edges, cross edges) that belong to each of the following sets: [5 Points]

Solution:

The set of back edges:

$(c, a), (k, h)$.

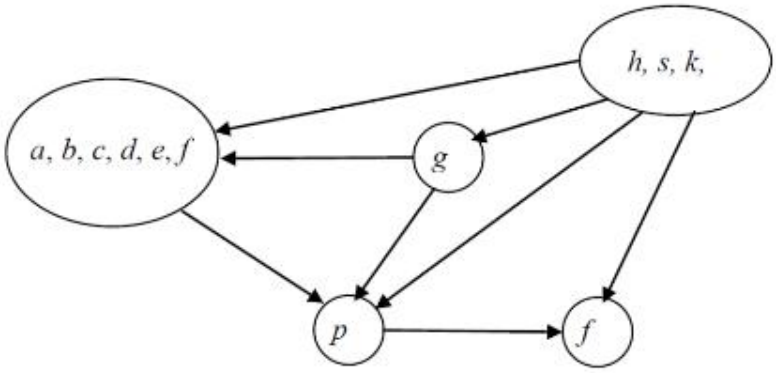
The set of forward edges:

$(d, j), (b, p)$.

The set of cross edges:

$(j, c), (s, a), (g, a), (s, g), (g, p), (h, p), (h, f), (h, g).$

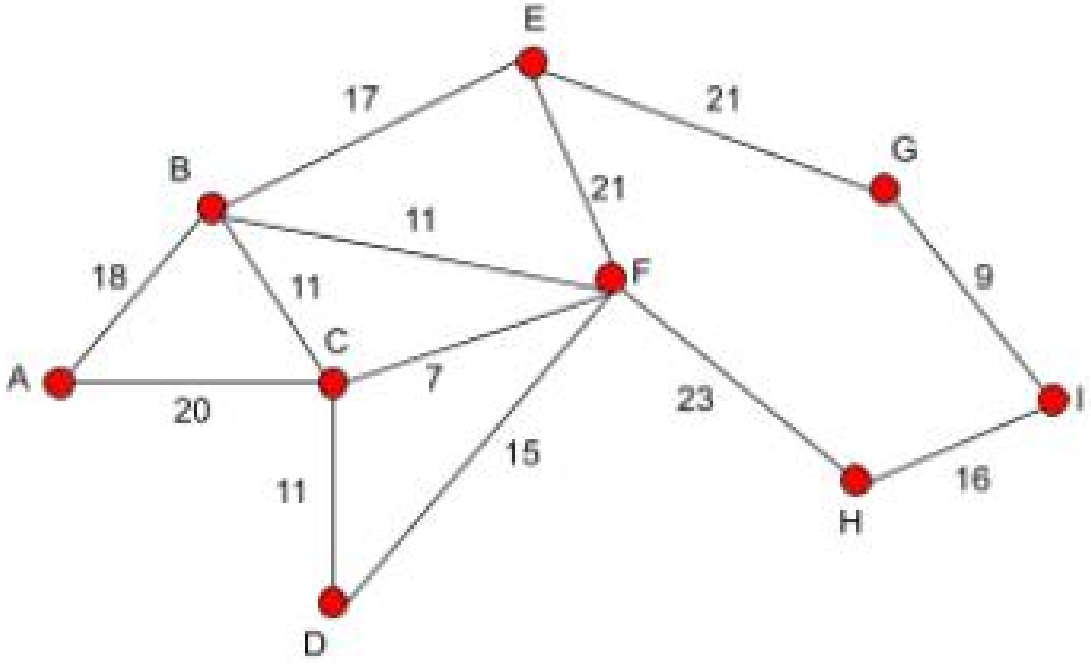
c) Identify the strongly connected components and draw the component graph. [10 Points]



2. a) Does Kruskal's algorithm begin by selecting an edge or node? [5 Points]

(b) The diagram below represents a network of paths in a park. The number on each edge represents the length of the path in metres. Using Kruskal's algorithm, find a minimum spanning tree for the network in the diagram and state its total length. [10 Points]

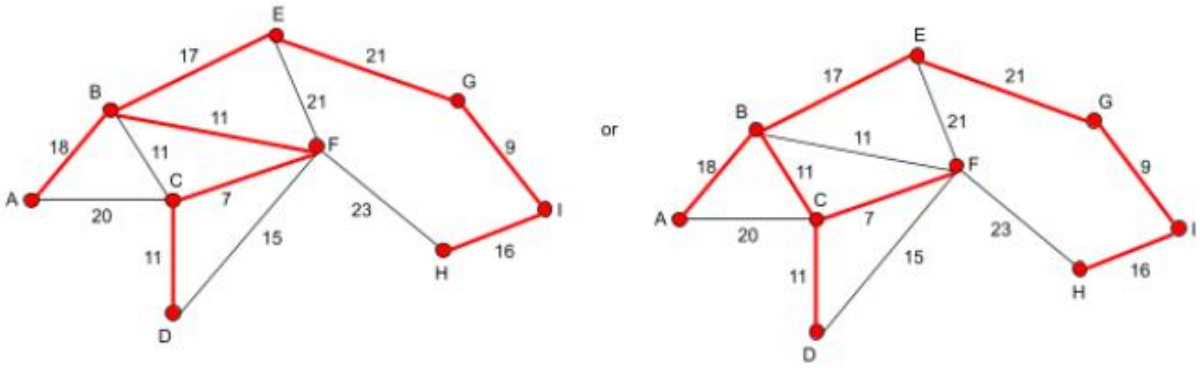
(c) Find the time complexity of algorithm. [5 Points]



Solution

(a) Kruskal's algorithm begins by selecting the edge of least weight.

(b) Total length = 110 m



$|V| <= |E| <= |V|^2$

```
KRUSKAL (G=(V,E),w)
A = 0;
for each vertex v in V
      MAKE-SET(v)
sort the edges E of G  into nondecreasing order by
  weight w
for each (u,v); // taken from the sorted list
      if FIND-SET(u)<>FIND-SET(v)
            A  = A + {(u,v)}
            UNION (u,v)
return A
```

A number of |V| calls of MAKE-SET — O(1)

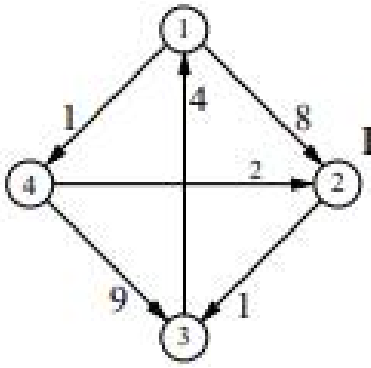
Sorting can be done in |E| * log (|E|)

O(|E|) calls of FIND-SET and O(|V|) calls of UNION — O(1)

O(V * log (V))

Union-Find implemented with linked lists and weighted union

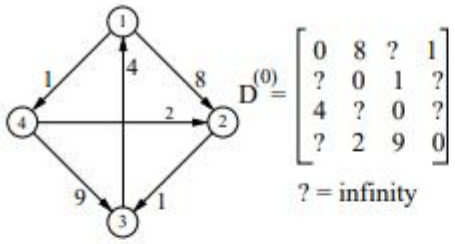
3. Using Floyd-Warshall, find all pairs shortest path following Figure (D^0 weight matrix is also provided). Discuss the its complexity as well [10 Points]



$$D^{(0)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & ? & 0 & ? \\ ? & 2 & 9 & 0 \end{bmatrix}$$

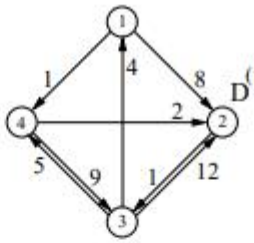
? = infinity

Solution :

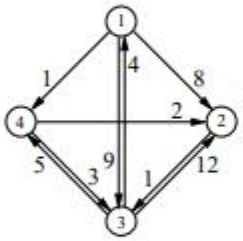


$$D^{(0)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & ? & 0 & ? \\ ? & 2 & 9 & 0 \end{bmatrix}$$

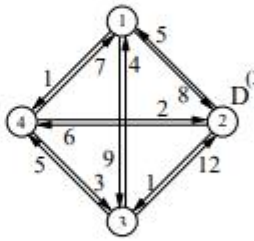
? = infinity



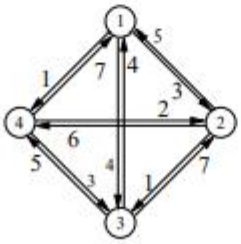
$$D^{(1)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 9 & 0 \end{bmatrix}$$



$$D^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 3 & 0 \end{bmatrix}$$



$$D^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



$$D^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Clearly the algorithm's running time is Θ(n3). The space used by the algorithm is Θ(n2).

4. Go through the lecture https://www.youtube.com/watch?v=2E7MmKv0Y24 and write summary with focus on proof of Dijkstra Algorithm [10 Points]

Solution:

The proof of correctness for Dijkstra's algorithm generally involves demonstrating two main properties:

Initialization: The algorithm initializes the distance of the source node to itself as 0 and all other nodes' distances as infinity. This initialization step ensures that the algorithm starts with the correct base case.

Optimality Property: At each step, the algorithm selects the node with the minimum distance from the source among the unvisited nodes and updates the distances to its neighboring nodes if a shorter path is found. The proof of correctness involves showing that once a node's distance is finalized (i.e., marked as visited), it is the shortest path distance from the source.

The proof often involves induction or the principle of mathematical induction to demonstrate that at each step of the algorithm, the selected node's distance remains the shortest among the explored nodes. By assuming the optimality of the shortest path to all visited nodes and extending it to the next node, the proof establishes that the algorithm correctly computes the shortest paths.

Key elements in the proof include:

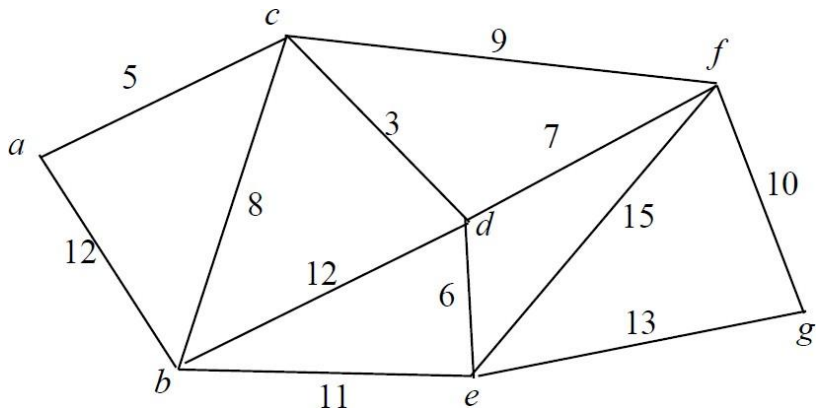
Invariant Property: The property that at every step, the algorithm maintains the correct shortest distance to all visited nodes.

Greedy Choice Property: Dijkstra's algorithm makes a greedy choice by selecting the node with the minimum distance at each step. The proof shows that this locally optimal choice leads to a globally optimal solution.

The proof of Dijkstra's algorithm emphasizes that at each step, the algorithm correctly updates the shortest distances until all nodes have been visited, ensuring the shortest path from the source to every other node.

Remember, while Dijkstra's algorithm is effective for finding shortest paths in graphs without negative edge weights, it may not produce correct results in graphs with negative weights without further modifications (like the Bellman-Ford algorithm).

5. (Longest path problem) Given a weighted (undirected) graph G(V, E), the weight of an edge is called the width of the edge. The longest path is defined to be thelargest weight among all edges on the path. (An edge with the largest weight is called the heavy edge.)



A path P(u, v) is called the longest if the cost of the path is the largest among all paths from u to v.

a) Modify the Dijkstra's algorithm to compute the longest path from a given vertex $s \in V$ to every other vertex. The pseudo code is required. [10 Points]

b) For the above graph, use the algorithm in part (a) to compute the longest path from source node a to each and every other node. You need to show each step, including the initialization step. Also show the final widest path tree. [10 Points].

Solution :

```
class Node:
  def __init__(self, val):
    self.val = val
    self.children = []

def longest_path(root):
  if not root:
    return 0

  # Recursively find the height of each subtree
  heights = [0]
  for child in root.children:
    heights.append(longest_path(child))
```

```
    # Sort the heights in descending order
    heights.sort(reverse=True)

    # Calculate the longest path passing through root
    max_path = max(heights[0], heights[0] + heights[1] + 1)

    return max_path

# Usage:
# Create the tree structure
root = Node(1)
root.children = [Node(2), Node(3), Node(4)]
root.children[0].children = [Node(5), Node(6)]
root.children[1].children = [Node(7)]

result = longest_path(root)
print("Longest path length:", result)
```

The Dijkstra algorithm is primarily used for finding the shortest path in a graph or a tree. If you want to find the longest path in a tree or graph, you would need a different approach, as Dijkstra's algorithm won't directly apply.

One way to find the longest path in a tree or graph is through a dynamic programming approach on trees, or through algorithms specifically designed for finding the longest path in a graph, like the Floyd-Warshall algorithm or a modified depth-first search (DFS) algorithm.

6 -Word search puzzle problem: Given the following 2d, 4x5 grid of letters

O F O O T
V O Q U O
E O I H O
R T G H F

a) Find the word "foot" in the grid.
b) The word may be formed in any direction - up, down, left or right (not diagonals!) but all of the letters in a word must occur consecutively. Assuming the grid starts in the upper left corner with position (0,0) , and that the row is the first coordinate, “foot” would be found at 3 places in the grid: [(0,1) to (0,4)] ,[(4,4) to (0,4)] , and [(0,1) to (4,2)].
c) Assume you are provided with the above word search puzzle grid. Write a algorithm to efficiently search all occurrences of the word. Display all the coordinates (row and column) of the starting and ending positions of word occurrence. [15 Points]

Solution

```
#include <iostream>
#include <string>

using namespace std;

const int ROWS = 4;
const int COLS = 5;

// Function to check if the given cell is within the grid
bool isValid(int row, int col) {
   return (row >= 0 && row < ROWS && col >= 0 && col < COLS);
}

// Helper function for recursive word search
void searchWordRecursive(char grid[][COLS], const string& word,
   int row, int col, int directionRow, int directionCol, int index,
   int& startRow, int& startCol, int& endRow, int& endCol) {

   if (index == word.length()) {
      cout << "Word found: " << word << " ("
         << startRow << "," << startCol << ") to ("
         << endRow << "," << endCol << ")" << endl;
      return;
   }

   if (!isValid(row, col) || grid[row][col] != word[index]) {
      return;
```

```
  }

  if (index == 0) {
    // Store the starting position
    startRow = row;
    startCol = col;
  }

  // Store the current position
  endRow = row;
  endCol = col;

  char original = grid[row][col];
  grid[row][col] = '*'; // Mark as visited

  // Recursively search in the specified direction
  searchWordRecursive(grid, word, row + directionRow, col + directionCol, directionRow,
directionCol, index + 1, startRow, startCol, endRow, endCol);

  grid[row][col] = original; // Restore the original character
}

// Function to search for a word using backtracking and recursion
void searchWord(char grid[][COLS], const string& word) {
  for (int i = 0; i < ROWS; ++i) {
    for (int j = 0; j < COLS; ++j) {
      if (grid[i][j] == word[0]) {
        for (int dr = -1; dr <= 1; ++dr) {
          for (int dc = -1; dc <= 1; ++dc) {
            if ((dr == 0 || dc == 0) && (dr != 0 || dc != 0)) {
              int startRow, startCol, endRow, endCol;
              searchWordRecursive(grid, word, i, j, dr, dc, 0, startRow, startCol, endRow,
endCol);
            }
          }
        }
      }
    }
  }
}

int main() {
  char grid[ROWS][COLS] = {
    {'H', 'E', 'L', 'L', 'O'},
    {'E', 'O', 'R', 'L', 'D'},
    {'L', 'B', 'C', 'D', 'E'},
```

```
    {'L', 'G', 'H', 'I', 'J'}
  };

  string word;

  cout << "Enter the word to search for: ";
  cin >> word;

  cout << "Word Search Puzzle:" << endl;
  for (int i = 0; i < ROWS; ++i) {
    for (int j = 0; j < COLS; ++j) {
      cout << grid[i][j] << ' ';
    }
    cout << endl;
  }

  cout << "Searching for '" << word << "':" << endl;
  searchWord(grid, word);

  return 0;
}
```

Output