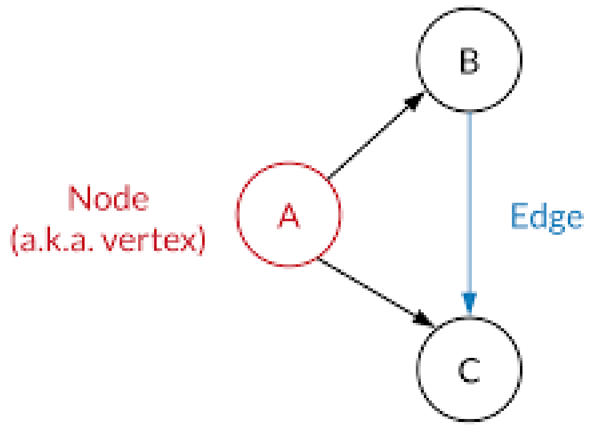
Final Review

Sara Prettyman

May 2, 2023



Contents

1 Unit 8: Introduction to Graphs

1.1 Introduction to Graphs

1.1.1 Introduction to Graphs

- A graph is a finite set of nodes with edges between nodes
- A graph G is a structure (V,E) consisting of:
  - a finite set V called the set of nodes (vertices)
  - a set that is a subset of VE

Terminology

- What are some characteristics of a binary tree? Rigid structure where each node has a single node that points to it.
- GRAPH - set of vertices/nodes and set of edges/arcs
  - More formally, as graph is a ordered pair of finite set V and E
  - V is the set of vertices
  - E is the set of edges
  - If edges have weight, termed weighted graph
  - If edges have direction, (can traverse from A to B but not B to A)
    -called directed graph otherwise undirected
- SUCCESSOR - the follows relationship of a directed graph
- PREDECESSOR - the precedes relationship in a directed graph
- DEGREE - the number of edges incident to or touching a vertex
  - IN-DEGREE - the number of edges coming into a vertex (digraph only)
  - OUT-DEGREE - the number of edges going out of a vertex. (digraph only)
- Diagraph - another name for a directed graph
- CONNECTED - a graph is connected if you can get from a node to every other node following edges.
- When (x,y) is an edge we say: x is adjacent to y, y is adjacent from x.

Intuition

- Nodes represent entities like people and cities
- Edges represent relationships between entities x and y, for instance x likes y.

1.1.2 Graph Representations

- For graphs to be computationally useful, they have to be conveniently represented in programs.
- Two common representations are adjacency matrix representation, adjacency list representation.

Adjacency Matrix Representation

- Each graph of n nodes is represented by an nxn matrix A, that is, a two dimensional array A.
- The nodes are (re)labeled 0,1,2,3,.., n-1. They are given integer labels.
- $$A[i][j] = \begin{cases} \text{true, if (i,j) is an edge} \\ \text{false, if (i,j) is not an edge.} \end{cases}$$

  Like A[0][1], if 0 is connected to 1.
- PRO: Simple to implement and easy and fast to tell if a pair is an edge.
- CON: No matter how few edges the graph has, it take up O(n^2) memory.
- The adjacency matrix is a reasonable way to store the graph for the following operations:
  - Get successors for node x. (you can just look at row x and see if there is a 1 for each column)
  - Determine if the graph is connected. (you can use a disjoint set to union all nodes traversed, and in the end if there is only one root node, then it is connected.)
  - Get predeccessors fo node x. (you can just look at column x and see if there is a 1 for each row)
  - Determine if there is an edge from node x to node y. (you can just look at A[x][y] and see if it is a 1)s
- Time and space complexity:
  - Space: O(n^2) where n is the number of nodes and m is the number of edges
  - Check if edge (u,v) exist: O(1)
  - Remove a vertex: O(n)
  - Adjacency matrices are more efficient for dense graphs with a large number of edges

Adjacency List Representation

- A graph of n nodes is represented by a one-dimensional array L of linked list, where
  - L[i] is a linked list containing all the nodes adjacent from node i
  - The nodes in list L[i] may or may not be in any particular order.
- Easy operation is know which nodes follow a value.
- Hard to see which edges go from a value. It isn't that hard, just trivial .
- Pros: saves space on memory
- Cons: it takes time to O(n to look up if a pair of nodes share an edge. Have to search the linked list L[i]
- The adjacency list is a reasonable way to store the graph for the following operations:
  - Determine if there is an edge from x to y (you can just look at the linked list L[x] and see if y is in the list)
  - List all edges in the graph (you can just look at each linked list L[i] and print out all the edges)
  - Get successors of node x (you can just look at linked list L[x] and print out all the nodes)
- Time and space complexity:
  - Space: O(n+m) where n is the number of nodes and m is the number of edges
  - Check if edge (u,v) exist: O(degree(u))
  - Remove a vertex: O(n+m)
  - Adjacency lists are more efficient for sparse graphs with a small number of edges

1.1.3 Directed vs Undirected Graphs

- An undirected graph is said to be connected if there is a path between every pair of nodes. Otherwise the graph is disconnected, meaning there are nodes riding solo.
- If an undirected graph is not connected, then each piece is called a connected component.
- If the graph is connected, then the whole graph is one single connected component.

Is Connected - Algorithm

- Mark all nodes as unvisited
- Select an unvisited node. Make the node as visited.
- Recursively visit all nodes reachable from this node - mark each node as visited
- Are all nodes visited: yes, connected. no, disconnected.
- Adjacency List Representation - Is Connected Code:

```java
            // Must add a visited ArrayList to the Graph class
            // Alternatively, add a visited property to the nodes

            public boolean isConnected() {
                clearVisitedFlags();
                traverse(0);
                for (int i = 0; i < vertexCount; i++) {
                    if (!visited[i]) return false;
                }
                return true;
            }

            public void traverse(int n) {
                if (visited[n]) return;
                visited[n] = true;
                for (EdgeInfo e : adjacencyList.get(n)) {
                    traverse(e.to);
                }
            }
```

Find Connected Components - Algorithm

- Mark all nodes as unvisited.
- Initialize component (piece) name to 0
- Repeat
  - Select an unvisited code: Make the node as visited, make the nodes component
  - Recursively visit all nodes reachable from this node: make the node as visited, make the nodes component
  - Increment component name by 1.

Strongly and Weakly Connected Components

- Every pair of vertices are reachable from each other
- Graph G is strongly connected if, for every (u, v) in V, there is some path from u to v and some path from v to u
- A graph is weakly connected if the underlying undirected graph is connected
- Can collapse a Strongly Connected Component (SCC) into a single supernode

Subgraph

- Suppose the graph on the left represents one way streets in a town and the nodes represent hotels
- Maybe I only care about hotels 0, 1, 3, 6. I can take those nodes and edges between them and talk about the subgraph
- Subgraph - a graph that consists of a subset of vertices and all the edges between those vertices

1.1.4 More Terminology

Additional Info

- By definition a **simple graph** does not contain
  - Multiple copies of the same edge
  - **Self-edges -** edges of the form (a, a); this is also called a loop
    - Also known as a self-loop; an edge that connects a vertex to itself.
- If we allow multiple edges between the same pair of nodes, it is termed a **multi-graph**.
  - e.g., multiple edges between hotels could represent two different roads between the hotels
- Paths
  - **Path** - a sequence of edges
  - **Simple Path -** there are no repeated vertices or edges
  - **Length of a path -** the sum of the lengths/weights of the edges on the path
- Cycle
  - A path that begins and ends at the same place
  - **Simple Cycle -** a cycle with no repeated vertices; except for the beginning and ending vertex

1.1.5 Homework 8.1

1. **An undirected graph contains a path from every node to every other node. How do we describe this graph?**
   Answer: A connected graph.

2. **In a directed graph, what feature describes a strongly connected component?**
   Answer: Every pair of vertices are reachable from each other. For every pair of vertices, (u, v) and (v, u), in the component, there exists a directed path that connects them.

3. **Consider the following constructor for a graph**

```
                Graph(int size) {
                        G = new GraphNode[size];
                        nodeCount = size;
                        for (int i = 0 ; i < size ; i ++) {
                           G[i].name = "A" + i;
                           G[i].nodeID= i;
                           G[i].isVisted = false;
                           G[i].dist = INFINITY;
                        }
                     }
```

   What is true about the nodes?
   Answer: The nodes are indexed by integers.

4. **Consider the following code**

```
                public class EdgeInfo {
                        public EdgeInfo(int from, int to){
                            this.from = from;
                            this.to = to;
                    }

                        private int from;        // source of edge
                        private int to;          // destination of edge
                    }

                public class Graph {
                    private int numVertex;
                    private LinkedList<EdgeInfo>[] list;

                    public Graph() {
                        this.numVertex = 0;
                    }

                    public Graph(int numVertex) {
                        this.numVertex = numVertex;
                        list = new LinkedList[numVertex];
                        for (int i = 0; i < numVertex; i++) {
                            list[i] = new LinkedList<EdgeInfo>();
                        }
                    }
                    }
```

   What does the following code do?

```
                    for (int j = 0; j < list[i].size(); j++) {
                        System.out.print(list[i].get(j) + "␣");
                    }
```

   Answer: Prints out all successors of node i.

5. **Consider the following code**

```
        public class EdgeInfo {
                public EdgeInfo(int from, int to){
                    this.from = from;
                    this.to = to;
            }

                private int from;          // source of edge
                private int to;            // destination of edge
            }

        public class Graph {
            private int numVertex;
            private LinkedList<EdgeInfo>[] list;

            public Graph() {
                this.numVertex = 0;
            }

            public Graph(int numVertex) {
                this.numVertex = numVertex;
                list = new LinkedList[numVertex];
                for (int i = 0; i < numVertex; i++) {
                    list[i] = new LinkedList<EdgeInfo>();
                }
            }
            }
```

What does the following code do?

```
            public boolean doIt(int i) {
                for (int j = 0; j < list[i].size(); j++) {
                    if (i == list[i].get(j).to) return true;
                }
                return false;
            }
```

Answer: Checks if there is a self-loop from i to i.

1.1.6 Graph Node, Adjacency List Example

```java
import java.util.*;

class GraphNode {
    public int id;
    public int parent;
    public LinkedList<EdgeInfo> successor;
    public boolean visited;

    public GraphNode(int id) {
        this.id = id;
        this.successor = new LinkedList<>();
        this.parent = -1;
        this.visited = false;
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(id + ": ");

        for (var edge : successor) {
            sb.append(edge.toString());
        }

        sb.append("\n");
        return sb.toString();
    }

    public void addEdge(int v1, int v2, int capacity) {
        successor.addFirst(new EdgeInfo(v1, v2, capacity));
    }

    public class EdgeInfo {
        int from;         // source of edge
        int to;           // destination of edge
        int capacity;     // capacity of edge

        public EdgeInfo(int from, int to, int capacity) {
            this.from = from;
            this.to = to;
            this.capacity = capacity;

        }

        public String toString() {
            return "Edge " + from + "->" + to + " (" + capacity + ") ";
        }
    }
}
```

1.2 Graph Traversals

1.2.1 Introduction

Two standard techniques

- Depth First Search (DFS), Breadth First Search (BFS). The nodes of an undirected graph are visited in a systemic manner so that every node is visited exactly once; both traversals can be performed on a directed graph
- Both BFS and DFS consider an embedded tree
  -When a node x is visited, it is labeled as visited, and it is added to the tree
  - If the traversal got to node x from node y, y is viewed as the parent of x, and x a child of y

1.2.2 Depth First Search

1. Select an unvisited node x, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node
3. If the current node has no unvisited neighbors, backtrack to its parent, then make that parent the new current node
4. Repeat steps 3 and 4 until no more nodes can be visited
5. If there are still unvisited nodes, repeat from step 1
   -This will (potentially) create a forest of trees
   Note: For a binary tree (a directed graph), this is what an in-order traversal does

The queue in a breadth first traversal could be replaced with...: Nothing, it must be a queue.
Time Complexity: O(V+E), where V is the number of vectrices and E is the number of edges in the graph. It is important to set the visited flag to true during traversal to prevent infinite loops.
Advantages: Memory requirement is linear with respect to the search graph, making it more memory-efficient compared to BFS, DFS can find a solution without examining much of the search space if the solution is found early in the traversal.
Disadvantages: DFS does not guarantee the shortest path to the solution, DFS can get stuck going down the wrong path and never reach the solution, DFS can get stuck in an infinite loop if the search space is infinite and the solution is not found.

1.2.3 Breadth First Search

1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level
2. Try all one-step extensions of current paths before trying larger extensions
3. Repeat step 2 until no more nodes can be visited
4. If there are still unvisited nodes, repeat from step 1. This will (potentially) create a forest of trees

Time Complexity: O(V+E), where V is the number of vectrices and E is the number of edges in the graph. BFS always finds the shortest path in an unweighted graph, as it explores all possible paths in increasing order of the number of edges.
Advantages: BFS guarantees finding a minimal solution if multiple solutions exist, BFS can be used to find the shortest path between two vertices in an unweighted graph, BFS is less prone to getting stuck in infinite loops compared to DFS, as it explores all possible paths in increasing order of the number of edges.
Disadvantages: BFS can be memory-intensive, as it stores all of the vertices in the queue, BFS can be slow, as it explores all possible paths in increasing order of the number of edges.

1.2.4 Topological Order

Outline

- Let G be a directed graph and V(G) = { v1, v2, ... vn }, where n 0
- We assume the graph has no cycles (DAG); because the graph has no cycles
  -There exists a vertex u in G such that u has no predecessor (start)
  - There exists a vertex v in G such that v has no successor (end)
- A topological ordering of V(G) is a linear ordering vj, vk, ... vn of the vertices such that if vj is a predecessor of vk, j $\neq$ k, then vj precedes vk, that is, j < k in this linear ordering

Algorithm

- Find a vertex that has no predecessor and make it first in the topological ordering
- Next, find the vertex v, all of whose predecessors have been placed in the topological ordering and place v next in the topological ordering
- Implementation details
  - To keep track of the number of vertices of a vertex, use an array (e.g., predecessorCount)
  - Initially predecessorCount[j] is the number of predecessors of the vertex vj
  - The container (a FIFO queue) used to guide the breadth first traversal is initialized to those vertices vk, where predecessorCount[k] is 0

- Pseudo code

```
Count the number of predecessors of each node
For each node with a count of 0, enqueue the node
While queue is not empty
    current node = dequeue node
    report the node // this prints the nodes in topological order
    For each successor of current node
        subtract one from its predecessor count
        if the predecessor count is 0, enqueue the node
```

- Questions:
  A topological ordering exists if... The graph is a DAG (directed acyclic graph) meaning direct and has no cycles.
  Consider the algorithm for finding a topological ordering. What best describes the type of queue required?

```
topologicalOrdering() {
    Queue q = new Queue();
    int counter = 0;

    for each vertex v
       if (v.indegree == 0)
          q.enqueue(v);
    while (!q.isEmpty()) {
       v = q.dequeue();
       v.number == ++counter;
       for each w adjacent to v
          if (--w.indegree == 0)
             q.enqueue(w);
    }
 }
```

  This is either a priority or FIFO queue that can be used.
  In the code for topological ordering shown in the other problem, what is true of the variable, indegree, associated with each node? (Select all correct answers)
  The algorithm assumes indegree is set before entering
  If a topological order exists, the indegree of all nodes will be zero after executing
  indegree is the number of predecessors
  Time complexity same as other.

1.3 Shortest Path Algorithm

1.3.1 Introduction to Shortest Paths

- The input is a weighted graph
  - associated with each edge (v_i, v_j) is a cost $c_{i,j}$
- The cost of a path is the sum of the weights of the edges, the weighted path length
- The **unweighted path length** is only the number of edges on the path
- **Single-Source Shortest-Path Problem:** Given a weighted graph G = (V, E), and a distinguished starting vertex s, find the shortest weighted path from s to every other vertex in G.
  - "The shortest path from where I am to everywhere else.
- An edge has a weight (cost) associated with it
  - Could have negative weights; really? yes, really!
- Positive cost: $ to traverse (gas or electricity)
- Negative cost: $ of profit made from selling something along that path

1.3.2 Unweighted Shortest Paths

- Using some vertex s, which is an input parameter, find the shortest path from s to all other vertices in a unweighted graph
  - Assume s = v3
  - For example, multiple ways to get to v7; how do we do this?

- Using some vertex s, which is an input parameter, find the shortest path from s to all other vertices in a unweighted graph
  - Use a breadth-first search; process nodes by distance, 1, 2, ...

- Pseudo-code

```
void unweighted(Vertex s) {
    Queue<Vertex> q = new Queue<Vertex>();
    for each Vertex v {
        v.dist = INFINITY;
    }
    s.dist = 0;
    q.enqueue(s);
    while (!q.isEmpty()) {
        Vertex v = q.dequeue();
        for each Vertex w adjacent to v {
            if (w.dist == INFINITY) { // Hasn t been visited.
                w.dist = v.dist + 1;
                w.predecessor = v; // track the shortest path back
                q.enqueue(w);
            }
        }
    }
}
```

1.3.3 Weighted Shortest Paths - Dijkstra's Algorithm

- Add the edge weight from the node to its successor to the current distance
- Pull nodes from a priority queue, based on shortest distance
- known vertices are those for which the shortest path has been determined
- The initial distance dv is tentative. It is the shortest path length from s to v using only known vertices
- It is a greedy algorithm: proceeds in stages doing the best at each stage
- Select a vertex v with smallest dv among all unknown vertices and declare it as known
  - Remainder of the stage consists of updating the values dw for all edges (v, w)
- Pseudo-code

```
class GraphNode {
    public int id;
    public List adj; // Adjacency list
    public boolean known;
    public int dist;
    public int path;
}
```

```
void dijkstra(GraphNode s) {
        for each GraphNode v {
            v.dist = INFINITY;
            v.known = false;
        }
        s.dist = 0;
        while (there is an unknown distance vertex) {
```

```
                    GraphNode v = unknown vertex with smallest distance
                    v.known = true; // We know we ll never find a better path
                    for each GraphNode w adjacent to v {
                        if (!w.known) {
                            cost = cost of edge from v to w;
                        if (v.dist + cost < w.dist) {
                                w.dist = v.dist + cost;
                                w.path = v.id;
                            }
                        }
                    }
                }
            }
```

Time Complexity: O(E + Vlog(V))

1.3.4 Weighted Shortest Paths - Negative Edge Weights

- Dijkstra's algorithm doesn't work with negative edge costs
- Possible solution is adding a constant to make everything positive. So cost+c ¿ 0. However, this doesn't work. It is bias against paths with more edges. cost =# of edges in path times c. 2 edges has 2c, where 1 edge is just c. That is an incorrect handling
- Alternative: Modify Dijkstra's algorithm.
  - Forget about the concept of a known vertex.
  - Use a regular queue (FIFO), don't need a priority queue.
  - Place starting vertex on queue (i.e V_1)
  - Examine length of all successors
    - For any successor whose distance has decreased, change their distance and place that node on the queue again.
    - If a node has been checked —V— times, don't add it back in the queue...prevents infinite loops due to negative edge costs
  - Repeat until no more nodes in the queue.
- In Modify Dijkstra's, you can revisit vertex. Works for a cyclic and non-cyclic graph. If there was a cycle, you stop visiting a vertex after a certain number of loops.

1.3.5 Weighted Shortest Paths - Acyclic Graphs

- Method for cyclic graphs with negative edge costs.
- Consider the nodes in topological order; as all predecessors are final
- Running time = $O(|V| + |E|)$
- This works because when a vertex is selected, its distance can no longer be lowered, because by topological ordering rule, it has no incoming edges emanating from unknown nodes

1.3.6 Homework 8.3 - Graphs (Dijkstra)

1. **Consider Dijkstra's algorithm, as show below.**

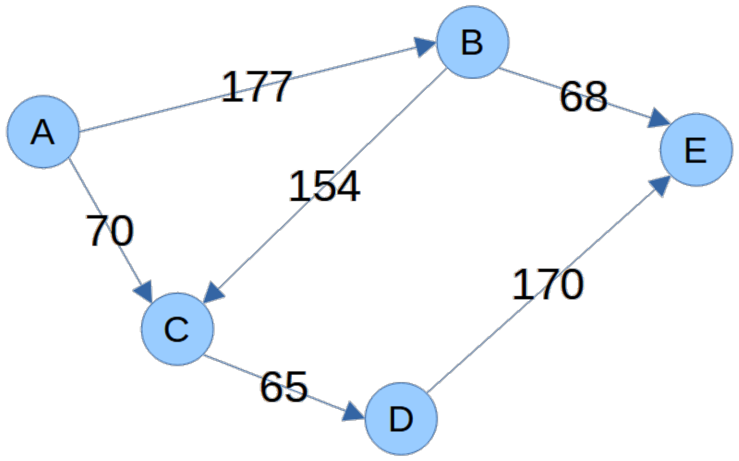
```
        void printShortestPath(int source) {
                PriorityQueue q = new PriorityQueue();

                q.enqueue(Pair(source, 0);
                while (!q.isEmpty()) {
                    Pair p = q.deleteMin();

                    if (!G[p.node].isVisited) {
                        G[p.node].isVisited = true;
```

```
                                G[p.node].dist = p.dist;
                                System.out.println("shortest␣distance␣to␣""␣+␣p.node␣+␣" is
                                     "␣+␣p.dist);

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣(Edge␣e␣:␣G[p.node].adj)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(!G[e.toNode].isVIsited)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣q.enqueue(Pair(e.toNode,␣p.dist+e.cost));
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
```



Starting from node A, when node E is marked as visited, what else is on the priority queue?
E(305)

2. **Consider Dijkstra's algorithm, as show below.**

```
                void printShortestPath(int source) {
                        PriorityQueue q = new PriorityQueue();

                        q.enqueue(Pair(source, 0);
                        while (!q.isEmpty()) {
                            Pair p = q.deleteMin();

                            if (!G[p.node].isVisited) {
                                G[p.node].isVisited = true;
                                G[p.node].dist = p.dist;
                                System.out.println("shortest␣distance␣to␣""␣+␣p.node␣+␣" is
                                     "␣+␣p.dist);

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣(Edge␣e␣:␣G[p.node].adj)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(!G[e.toNode].isVIsited)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣q.enqueue(Pair(e.toNode,␣p.dist+e.cost));
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
```

Starting from node A, when node D is marked as visited, what else is on the priority queue?
B(177)

3. **Consider Dijkstra's algorithm, as show below.**

```
                    void printShortestPath(int source) {
                        PriorityQueue q = new PriorityQueue();

                        q.enqueue(Pair(source, 0);
                        while (!q.isEmpty()) {
                            Pair p = q.deleteMin();

                            if (!G[p.node].isVisited) {
                                G[p.node].isVisited = true;
                                G[p.node].dist = p.dist;
                                System.out.println("shortest␣distance␣to␣""␣+␣p.node␣+␣" is
                                    "␣+␣p.dist);

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣(Edge␣e␣:␣G[p.node].adj)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(!G[e.toNode].isVIsited)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣q.enqueue(Pair(e.toNode,␣p.dist+e.cost));
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
```

In the algorithm, why is it necessary to check if a node has been visited when pulling something off the queue?
There may be two paths to a node, so a node could get put on the queue twice

4. **Why do negative edge weights cause problems with Dijkstra's algorithm?**
When a node is pulled off the queue, a shorter path may exist

5. **Dijkstra's algorithm is like a breadth first traversal, except...**
Dijkstra's algorithm uses a priority queue instead of a first-in-first-out queue

1.4 Network Flow

1.4.1 Introduction to Network Flow & the Ford-Fulkerson algorithm

- Sink has flow in, source has flow out. Conservation Rule – In order for the assignment of flows to be valid, we must have the sum of flow coming into a vertex equal to the flow coming out of a vertex, for each vertex in the graph except the source and the sink
- Capacity Rule – Each flow must be less than or equal to the capacity of the edge
- The flow of the network is defined as the flow from the source to the sink

1.4.2 Residual Graphs

A residual graph is a graph derived from the original graph in the context of network flow problems. It represents the capacity that can still be added to the flow along each edge or the flow that can be removed (sent in the opposite direction). The residual graph is used to find augmenting paths to improve the flow in the network.

Definition: Given a flow network G with capacities c(u, v) and a flow f(u, v), the residual graph G_f is a graph with the same vertices as G and two types of edges:
-Forward edges: These have a capacity equal to the remaining capacity of the corresponding edge in the original graph (c(u, v) - f(u, v)).
-Backward edges: These have a capacity equal to the flow on the corresponding edge in the original graph (f(u, v)).

Augmenting paths: An augmenting path is a path in the residual graph where each edge has a positive residual capacity. It can be used to increase the flow in the network.

Augmenting Path Given a flow network G, an augmenting path is a simple path from the source to the sink in the corresponding residual network G_r. Intuitively, an augmenting path tells us how we can change the flow on certain edges in G so that we increase the overall flow from the source to the sink.

Ford-Fulkerson Algorithm The Ford-Fulkerson algorithm is a method for finding the maximum flow in a flow network by iteratively augmenting the flow along paths from the source to the sink.

Steps:

1. Initialize the flow in all edges to 0.
2. While there exists an augmenting path from the source to the sink in the residual graph (a graph that represents the remaining capacities of the edges):
   - Find an augmenting path using any search method (Depth-First Search).
   - Determine the bottleneck capacity (minimum capacity along the path).
   - Update the flow in the original graph by sending flow along the augmenting path.
   - Update the residual graph to reflect the updated flow.
3. The maximum flow is the sum of flows sent in each iteration.

1.4.3 Example applications of Network Flow

Find MaxFlow algorithm

```java
        public int findMaxFlow(int s, int tt) {
        // TODO:
        int totalFlow = 0;
        StringBuilder allPaths = new StringBuilder();

        while (hasAugmentingPath(s, t)) {
            StringBuilder path = new StringBuilder();
            int availableFlow = Integer.MAX_VALUE;
            int v = t;
            Stack<Integer> stack = new Stack<>();
            while (v != s) {
                int predecessor = vertices[v].parent;
                availableFlow = Math.min(availableFlow, getEdgeCapacity(predecessor, v));
                v = vertices[v].parent;
            }
            path.append("Flow ").append(availableFlow).append(": ");
            v = t;
            while (v != s){
                path.insert(path.indexOf(":") + 2, v + " ");
                int predecessor = vertices[v].parent;
                updateResidualGraph(predecessor, v, -availableFlow);
                updateResidualGraph(v, predecessor, availableFlow);
                v = vertices[v].parent;
            }
            path.insert(path.indexOf(":") + 2, s + " " );
            path.append("\n");
            allPaths.append(path);
            totalFlow += availableFlow;
        }
        return totalFlow;
    }
```

1.4.4 Edmonds-Karp algorithm

The Edmonds-Karp algorithm is an improvement of the Ford-Fulkerson algorithm for solving the max-flow problem in a flow network. It uses BFS (Breadth-First Search) to find the augmenting paths in the residual graph. Ford-folkerson is slow to finding optimal. Does least number of edges. Edmonds-Karp will be sure to choose the shortest augmenting path from the source to the sink

1.4.5 Computing the Minimum Cut

- Task: minimum number of edges to prevent flow.

1.4.6 Computing the Maximum Flow for the Minimum Cost

- Each edge is carrying capacity: cost.v

1.4.7 Homework 8.3

1. **What is the cost per one unit along the minimum cost route?**
   What is the total cost along that route
2. **What is the amount of flow that can be placed on the minimum cost route?**
   What ever the minimum edge value is.

1.5 Minimum Spanning Trees

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight

1.5.1 Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm that starts with an empty set of edges and iteratively adds edges to the MST while ensuring that the resulting graph remains a forest (a collection of trees). This one starts are the edge with the smallest capacity.

Steps:

1. Sort all the edges in the graph in increasing order of their weights.
2. Initialize an empty set for the MST.
3. Iterate through the sorted edges. For each edge (u, v):
   -If adding the edge (u, v) does not create a cycle, add it to the MST set.
4. Repeat step 3 until the MST set has (V - 1) edges, where V is the number of vertices in the graph

Kruskal's Algorithm is generally preferred for sparse graphs (few edges), as it sorts the edges, which takes O(E log E) time, and iteratively adds edges based on their weights. It is also easier to implement for disconnected graphs.

1.5.2 Prim's Algorithm

Prim's Algorithm is also a greedy algorithm, but it starts with an arbitrary vertex and iteratively expands the MST by adding the minimum-weight edge that connects a vertex in the MST to a vertex not in the MST.

Steps:

1. Choose an arbitrary vertex as the starting vertex
2. Add all nodes reachable from this node into an empty priority queue
3. While there are unvisited vertices, perform the following steps:
   - Find the minimum-weight edge (u, v) such that u is in the visited set and v is not.
   - Add edge (u, v) to the MST set and add vertex v to the visited set.

Prim's Algorithm is more efficient for dense graphs (many edges), as it only considers edges connected to the vertices already in the MST, which can be done in O(V^2) time using adjacency matrix representation or O(log V) time using a priority queue and adjacency list representation. It works for undirect, and weighted or unweighted graphs.

1.6 Paths & Circuits

Euler paths and circuits focus on visiting every edge exactly once, while Hamilton paths and circuits focus on visiting every vertex exactly once.

1.6.1 Introduction to Euler Paths & Circuits

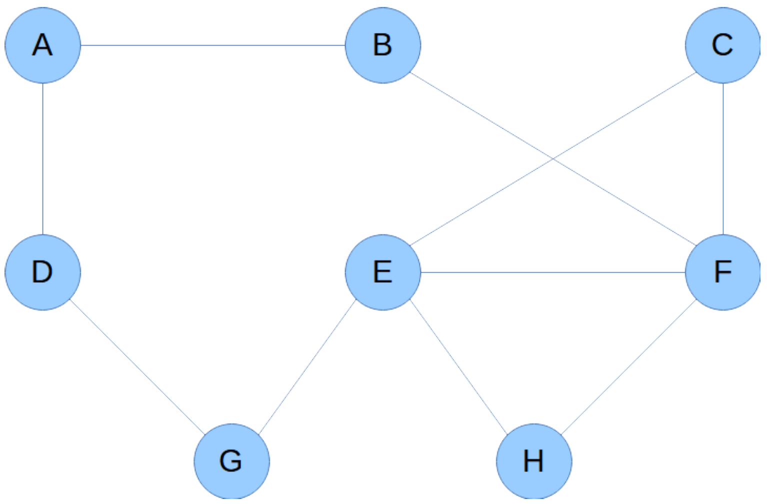
Euler path - visits every edge of a graph exactly once, without repeating any edges. It may repeat vertices
Euler circuit - a closed path that visits every edge of a graph exactly once, without repeating any edges, and starts and ends at the same node

1.6.2 Euler Circuit Algorithm

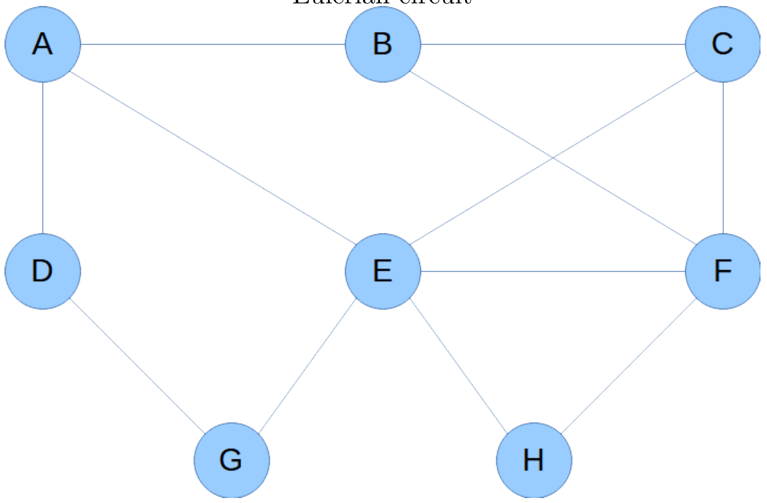
1.6.3 Introduction to Hamilton Paths & Circuits

Hamilton Path - path that visits every vertex of a graph exactly once, without repeating any vertices Hamilton circuit -

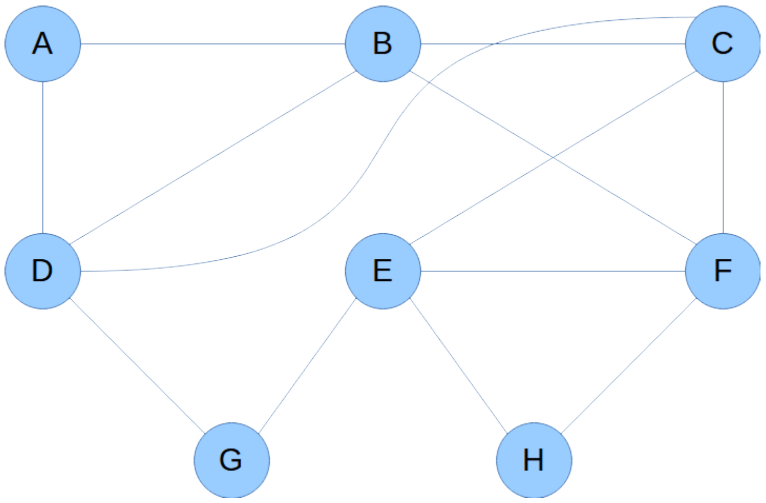
1.6.4 Examples:



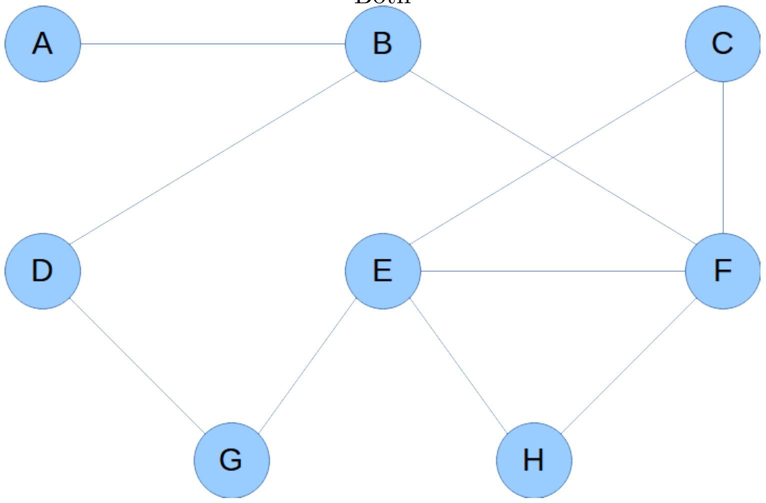
Eulerian circuit



Hamilotnian circuit



Both



Neither

1.7 Strongly Connected Components

1.7.1 Finding Strongly Connected Components

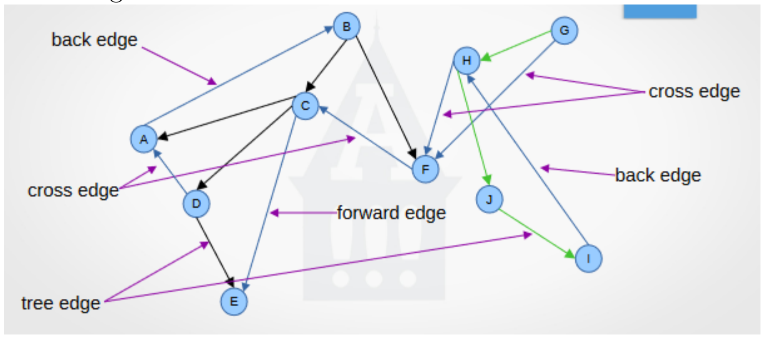
Step:

1. Start at desired node.
2. Follow path by a depth first search until you've hit an already visited node, or at a end node.
3. Once at finality node, number that.
4. Move backwards recursively until you arrive at an unvisited node. Follow that path, repeating steps 2 and 3 until you've visited all possible adjacent nodes from the start node.
5. Go to next start node.
6. Repeat 1-5 until you've traversed the whole graph.
7. Start a new tree with next highest number. Mark all nodes that can be reached. That is one connected component.
8. Go to next highest number. Repeat until all nodes have identfied into a path.

1.8 Edge Types

Tree Edge : Any edge in the tree
Forward Edge : Node to a successor
Back Edge : Node to an ancestor

Cross Edge : Node to an unrelated node



Questions

1. **The object of the Kevin Bacon game is to link a movie actor to Kevin Bacon via shared movie roles. The minimum number of links is an actor's Bacon number. For instance, Tom Hanks has a Bacon number of 1 as he was in Apollo 13 with Kevin Bacon. Sally Field has a Bacon number of 2, because she was in Forrest Gump with Tom Hanks who was in Apollo 13 with Kevin Bacon. Forming this as a graph problem, what are the edges?**
   Been in a movie with

2. **Finding an actor's Bacon number is what kind of problem?**
   Shortest path

3. **A student needs to take a certain number of classes to graduate, and these courses have prerequisites that must be followed. Assume that every class is offered every semester and that a student can take an unlimited number of courses. Given a list a courses that a student wants to take and their prerequisites, compute a schedule that requires the minimum number of semesters. Which algorithm would be most useful?**
   topologicalOrdering

4. **Suppose that a maze exists, but may or may not be solvable. You want to determine if the maze is solvable. Which algorithm will be most helpful?**
   Union Find

5. **We have two sets: People and Candy Bars. A perfect matching is a matching in which every item in the People set is paired with an item in the Candy Bar set, such that all Candy Bars are matched. Which algorithm will be most helpful?**
   Maximum Flow

6. **We need to build a connected network. We have a limited set of possible routes, and we want to select a subset that will make our network (e.g., electrical grid, computer network) connected at the lowest cost. Which algorithm is most helpful?**
   Minimal Spanning Tree

1.9 Biconnected Graphs

An articulation point is a node whose removal does disconnect the graph.

1.10 Hard vs. Easy Problems

Vertex Cover A vertex cover of an undirected graph is a subset of vertices such that it includes at least one endpoint of every edge of the graph. The goal is to find the minimum vertex cover, which is the smallest possible subset of vertices that still satisfies this property.

Steps:

1. Start with an empty set for the vertex cover.

2. While there are uncovered edges, perform the following steps:

- Choose an uncovered edge (u, v).
- Add both vertices u and v to the vertex cover set.
- Remove all edges incident to either u or v from the graph (as they are now covered).

3. The resulting vertex cover set is an approximation of the minimum vertex cover. It does not guarentee an optimal solution.

2 Unit 9: Introduction to Algorithm Design

2.1 Introductinon to Dynamic Programming

Greed Algorithms Greedy algorithms are a class of algorithms that make locally optimal choices at each step, with the hope that these choices will lead to a globally optimal solution. These algorithms are generally simple to implement and have good performance for certain problems. However, they may not always provide optimal solutions.

Memorizing Solving the repeated sub-problem compute problem. An optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.
Example: Fibonacci Sequence - Calculating the nth Fibonacci number using naive recursion results in exponential time complexity. Using memoization, the computed values of Fibonacci numbers are stored in a cache (e.g., an array or dictionary), and when the function is called again with the same input, the cached result is returned instead of re-computing the value, reducing the time complexity to linear.

Dynamic Programming Dynamic programming is a technique used to solve problems by breaking them down into smaller, overlapping subproblems and solving them in a bottom-up manner. The solutions to subproblems are stored in a table to avoid redundant computations, which reduces time complexity. Dynamic programming is particularly useful for optimization problems, and it guarantees an optimal solution when applicable.

- Overlapping subproblems: Problems that can be broken down into smaller subproblems that are solved multiple times.
- Optimal substructure: The optimal solution to the problem can be constructed from optimal solutions of its subproblems.
- Bottom-up approach: Solve smaller subproblems first and use their solutions to construct solutions for larger subproblems.

Example: Longest Common Subsequence (LCS) - The LCS problem involves finding the longest subsequence that is common to two input sequences. Dynamic programming can be applied by creating a table to store the LCS length for each pair of input characters, using a bottom-up approach to fill the table, and finally constructing the LCS from the table.
Example: The Making Change problem involves finding the minimum number of coins required to make a given amount of change. A greedy algorithm can solve the problem for certain coin systems (e.g., US coins) by always choosing the largest denomination coin that is less than or equal to the remaining change. However, for arbitrary coin systems, a dynamic programming approach is needed to find the optimal solution, by constructing a table of minimum coins needed for all amounts from 1 to the target amount.

3 Additional Graphics