Data Structures and Algorithms
(ECEG 4171)

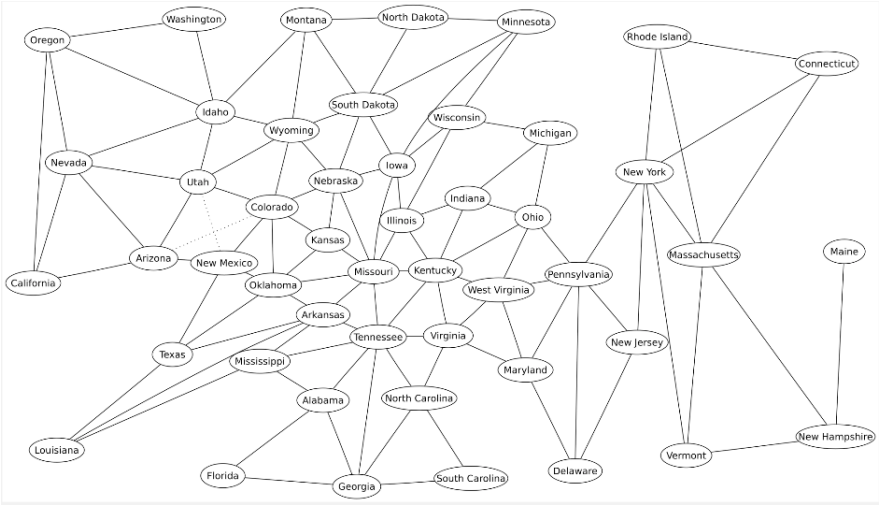
Chapter Seven
Graphs

Ephrem A. (M.Sc.)
School of Electrical and Computer Engineeing
Chair of Computer Engineering
30th December, 2019

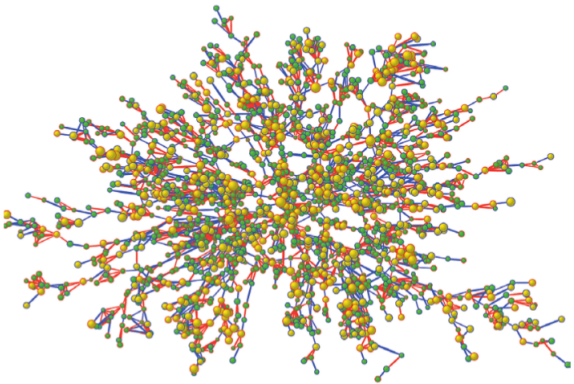
Inroduction

- A graph is a way of representing relationships that exist between pairs of objects.
- A graph G = (V, E) consists of a set of vertices V and a set of edges E, such that each edge in E is a connection between a pair of vertices in V.
  - V = set of vertices
  - E = set of edges = subset of V x V
  - Thus $|E| = O(|V|^2)$
- Why study graph algorithms?
  - Thousands of practical applications.
  - Hundreds of graph algorithms known.
  - Interesting and broadly useful abstraction.
  - Challenging branch of computer science, engineering and discrete math.

Border graph of 48 contiguous States



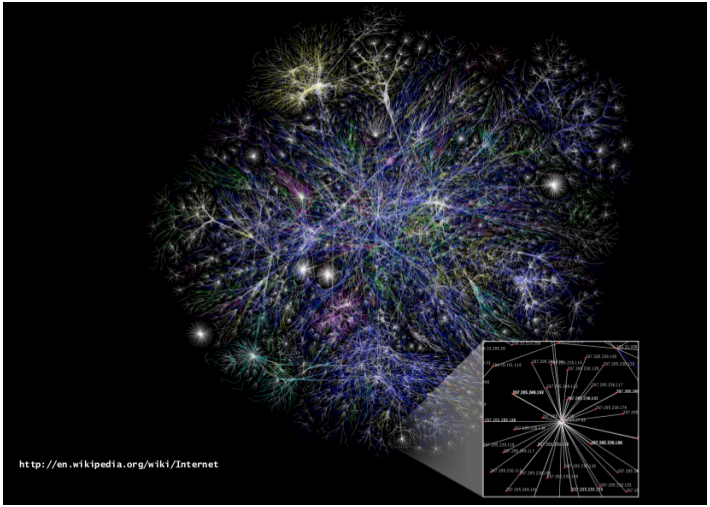
Framingham heart study



Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ¿=30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

10 million Facebook friends

The Internet as mapped by the Opte Project



Applications of Graphs

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	intersection	street
internet	class C network	connection
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

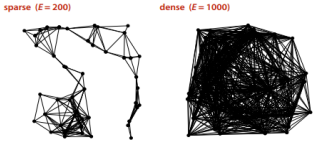
Graph Variations

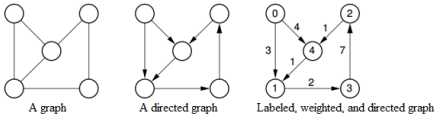
- In a directed graph (digraph):
  - Edges are directed from one vertex to another.
  - Edge (u, v) goes from vertex u to vertex v, notated $u \rightarrow v$.
- An undirected graph is a graph whose edges are not directed: Edge(u, v) = Edge(v, u).
- Two vertices are adjacent if they are joined by an edge. Such vertices are also called neighbors.
- An edge connecting vertices u and v, written as (u, v), is said to be incident on vertices u and v.
- A sequence of vertices $v_1, v_2, \ldots, v_n$ forms a path of length $n-1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$.
- The length of a path is the number of edges it contains.
- Two vertices are connected if there is a path between them.
- A connected graph is graph which has a path from every vertex to every other.

Graph Variations

- A cycle is a path of length three or more that connects some vertex v_1 to itself. A graph is cyclic if it contains one or more cycles.
- A graph without cycles is called acyclic. Thus, a directed graph without cycles is called a directed acyclic graph or DAG.
- A weighted graph associates weights with either the edges or the vertices.
  - E.g. a road map: edges might be weighted with distance.
- A labeled graph associates labels with it.

- If $|E| \approx |V|^2$, the graph is dense
- If $|E| \approx |V|$, the graph is sparse. E.g see fig in right for V=50



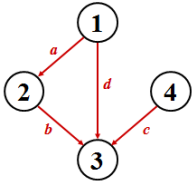


Graph Respresentations

- There are two commonly used methods for representing graphs:
  - Adjacency matrix
  - Adjacency list
- In most cases, we assume V = {0, 1, . . . , n-1}
- An adjacency matrix reprsents the graph as a VxV matrix A:

$$A[i,j] = 1 \text{ if edge(i, j)} \in \text{E (or weight of edge)}$$
$$= 0 \text{ otherwise}$$

- Example:



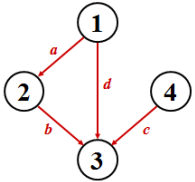
A	1	2	3	4
1				
2				
3			??	
4				

Graph Respresentations

- There are two commonly used methods for representing graphs:
  - Adjacency matrix
  - Adjacency list
- Assume V = {0, 1, . . . , n-1}
- An adjacency matrix reprsents the graph as a VxV matrix A:

$$A[i,j] = 1 \text{ if edge(i, j)} \in \text{E (or weight of edge)}$$
$$= 0 \text{ otherwise}$$

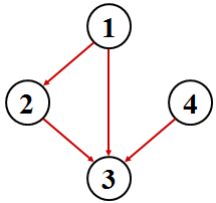
- Example:



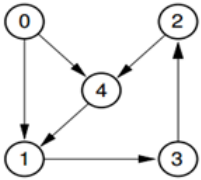
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

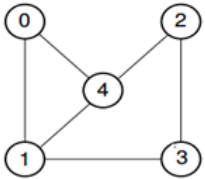
Adjacency List

- For each vertex $v \in V$, store a *list* of vertices adjacent to v.
- Example:
  - Adj[1] = {2, 3}
  - Adj[2] = {3}
  - Adj[3] = { }
  - Adj[4] = {3}

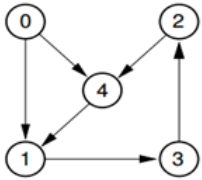


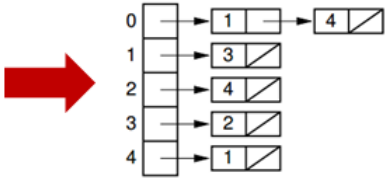
Adjacency List

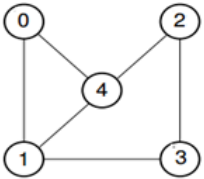


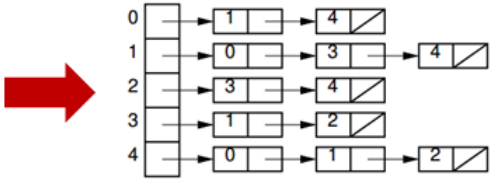


Adjacency List









Adjacency List

- The degree of a vertex v is the number of incident edges.
  - Directed graphs have in-degree (the number of edges pointing to a vertex), and out-degree (the number of edges pointing from a vertex).
- For directed graphs, the number of items in adjacency list is

$$\sum_{\forall v \in V} outdegree(v) = |E|$$

  Hence, takes $\Theta(V + E)$ storage.
- For undirected graphs, the number of items in adjacency list is

$$\sum_{\forall v \in V} degree(v) = 2|E|$$

- Also takes $\Theta(V + E)$ storage. Therefore, adjacency list takes $\Theta(V + E)$ storage.

Adjacency Matrix or Adjacency List?

$Q1$ How much storage does the adjacency matrix require?

Adjacency Matrix or Adjacency List?

Q1 How much storage does the adjacency matrix require?

A $O(V^2)$

Adjacency Matrix or Adjacency List?

Q1 How much storage does the adjacency matrix require?

A $O(V^2)$

Q2 What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 5 vertices?

Adjacency Matrix or Adjacency List?

Q1 How much storage does the adjacency matrix require?

A $O(V^2)$

Q2 What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 5 vertices?

A 10 bits

Adjacency Matrix or Adjacency List?

Q1 How much storage does the adjacency matrix require?

A $O(V^2)$

Q2 What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 5 vertices?

A 10 bits

- Undirected graph $\rightarrow$ matrix is symmetric ($A = A^T$).
- No self-loops $\rightarrow$ don't need diagonal
- Store only the entries on and above the diagonal of the adjacency matrix.

- The adjacency matrix is a dense represenation:
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse. For this reason the adjacency list is often a more appropriate representation.

Representation	Space	Add Edge	edge between v and w?	iterate over vertices adjacent to v?
Adjacency matrix	V^2	1	1	V
Adjacency list	$E + V$	1	degree(v)	degree(v)

Undirected Graph Implementation

- Graph API

public class	Graph	
	Graph(int V)	*create a V-vertex graph with no edges*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge v-w to this graph*
Iterable<Integer>	adj(int v)	*vertices adjacent to v*
String	toString()	*string representation*

```
// degree of vertex v in graph G
public static int degree(Graph G, int v){
    int degree = 0;
    for (int w : G.adj(v))
       degree++;
    return degree;
}
```

Graph Implementation

```
public class Graph {
    private final int V; // Number of vertices
    private int E; // Number of edges
    private List<Integer>[] adj; // Adjacency lists

    public Graph(int V){
        this.V = V;
        this.E = 0;
        adj = (LinkedList<Integer>[]) new LinkedList[V]; // Empty graph with V vertices.
        for(int i = 0; i < V; i++)
            adj[i] = new LinkedList();
    }
    public void addEdge(int v, int w){
        adj[v].add(w);
        adj[w].add(v);
        E++;
    }
    public int V(){ return V; }
    public int E(){ return E; }
    public Iterable<Integer> adj(int v){
        return adj[v];
    }
}
```

Graph Traversals

- Given: a graph G = (V, E), directed or undirected:
- Goal: methodically explore every vertex and every edge
- Ultimately:
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Might also build a forest if graph is not connected.
- Two techniques:
  - Depth-first search (DFS)
  - Breadth-first search (BFS)

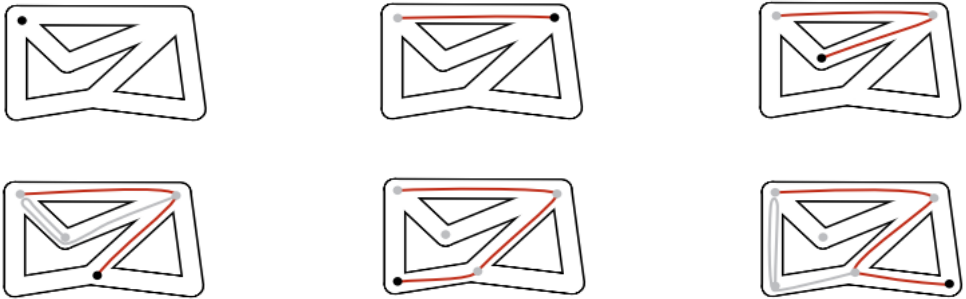
Maze Exploration

- Maze graph: vertex = intersection, edge = passage.

- **Goal:** Explore every intersection in the maze.

Tremaux exploration

- Algorithm:
  - Unroll a ball of string behind you.
  - Mark each visited intersection and each visited passage.
  - Retrace steps when no unvisited options.



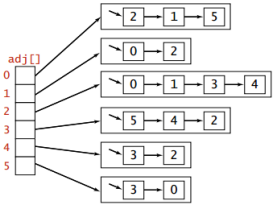
Depth-first search (DFS)

- Explore deeper in the graph whenever possible.
- Edges are explored out the most recently discovered vertex v that has still unexplored edges
- When all of v's edges have been explored, backtrack to the vertex from which v was discovered.
- DFS code:

```
public class DepthFirstSearch{
    private boolean[] marked;
    public DepthFirstSearch(Graph G, int s){
        marked = new boolean[G.V()];
        dfs(G, s);
    }
    private void dfs(Graph G, int v){
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }
    public boolean marked(int w){ return marked[w]; }
}
```

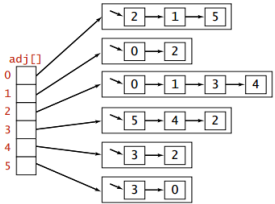
DFS Example

Adjacency list

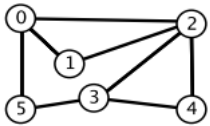


DFS Example

Adjacency list

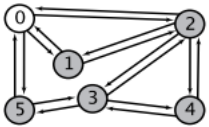


Standard graph



- The search begins when the constructor calls the recursive dfs() to mark and visit vertex 0.

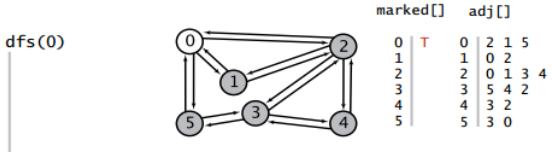
dfs(0)



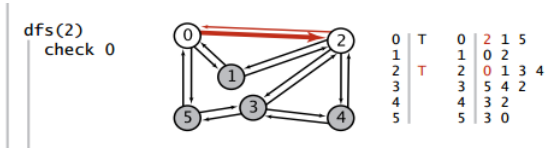
	marked[]	adj[]
0	T	2 1 5
1		0 2
2		0 1 3 4
3		5 4 2
4		3 2
5		3 0

DFS Example

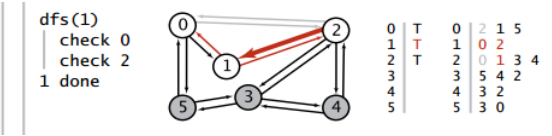
Mark vertex 0.



Since 2 is first on 0's adjacency list and is unmarked, dfs() recursively calls itself to mark and visit 2.



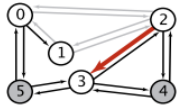
Now, 0 is first on 2's adjacency list and is marked, so dfs() skips it. Then, since 1 is next on 2's adjacency list and is unmarked, dfs() recursively calls itself to mark and visit 1.



DFS Example

Since both vertices on its list (0 and 2) are already marked, no recursive calls are needed, and dfs() returns from the recursive call dfs(1). The next edge examined is 2-3

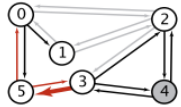
```
dfs(3)
```



0	T	2 1 5
1	T	0 2
2	T	0 1 3 4
3	T	5 4 2
4		3 2
5		3 0

Vertex 5 is first on 3's adjacency list and is unmarked, so dfs() recursively calls itself to mark and visit 5. Both vertices on 5's list (3 and 0) are already marked, so no recursive calls are needed,

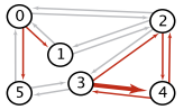
```
dfs(5)
  check 3
  check 0
5 done
```



0	T	2 1 5
1	T	0 2
2	T	0 1 3 4
3	T	5 4 2
4		3 2
5	T	3 0

vertex 4 is next on 3's adjacency list and is unmarked, so dfs() recursively calls itself to mark and visit 4, the last vertex to be marked. After 4 is marked, dfs() needs to check the vertices on its list, then the remaining vertices on 3's list, then 2's list, then 0's list, but no more recursive calls happen because all vertices are marked.

```
    dfs(4)
      check 3
      check 2
    4 done
    check 2
  3 done
  check 4
2 done
check 1
check 5
0 done
```



0	T	2 1 5
1	T	0 2
2	T	0 1 3 4
3	T	5 4 2
4	T	3 2
5	T	3 0

Breadth-first search

- Not recursive. Uses queues as a secondary auxilary data structure.
- Expands frontier of explored vertices across the breadth of the frontier.
- Builds a tree over the graph:
  - Pick a source vertex to be the root.
  - Find or discover its children, then their children, etc.
- Procedure:
  - Repeat until queue is empty:
    - Remove vertex v from queue
    - Add to queue all unmarked vertices adjacent to v and mark them.

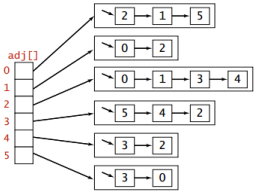
BFS Code

```
public class BreadthFirstSearch{
    private boolean[] marked; // Is a shortest path to this vertex known?
    private int[] edgeTo; // last vertex on known path to this vertex
    private final int s; // source
    public BreadthFirstSearch(Graph G, int s){
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        bfs(G, s);
    }
    private void bfs(Graph G, int s){
        Queue<Integer> queue = new LinkedList<Integer>();
        marked[s] = true; // Mark the source
        queue.add(s); // and put it on the queue.
        while (!queue.isEmpty()){
            int v = queue.remove(); // Remove next vertex from the queue.
            for (int w : G.adj(v))
                if (!marked[w]){ // For every unmarked adjacent vertex,
                    edgeTo[w] = v; // save last edge on a shortest path,
                    marked[w] = true; // mark it because path is known,
                    queue.add(w); // and add it to the queue.
                }
        }
    }
}
```

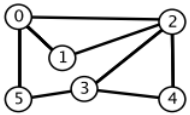
- `edgeTo[]` is a vertex-indexed array such that `edgeTo[w] = v` means that `v-w` was the edge used to access `w` for the first time.

BFS Example

Adjacency list

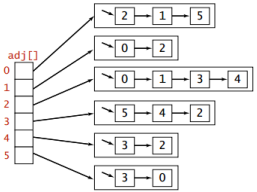


Standard graph

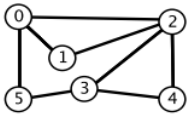


BFS Example

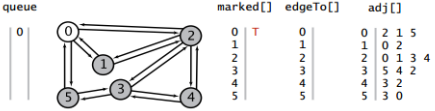
Adjacency list



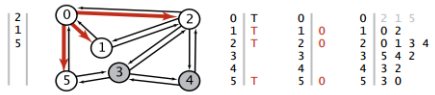
Standard graph



Vertex 0 is is marked and put on the queue, then the loop starts. edgeTo[] stores the last vertex on the path of the current vertex.

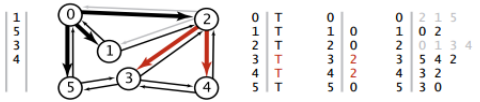


Removes 0 from the queue and puts its adjacent vertices 2, 1, and 5 on the queue, marking each and setting the edgeTo[] entry for each to 0.

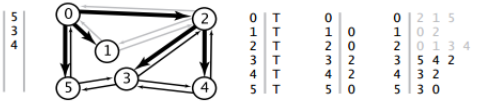


BFS Example ...

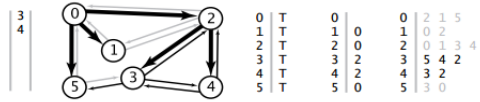
Removes 2 from the queue, checks its adjacent vertices 0, 1, 3 and 4 and puts only vertices 3 and 4 on the queue (the unmarked), marking each and setting the edgeTo[] entry for each to 2.



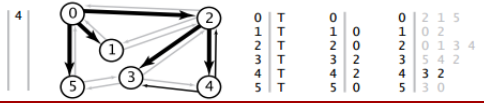
Removes 1 from the queue and checks its adjacent vertices 0 and 2, which are marked.



Removes 5 from the queue and checks its adjacent vertices 3 and 0, which are marked.

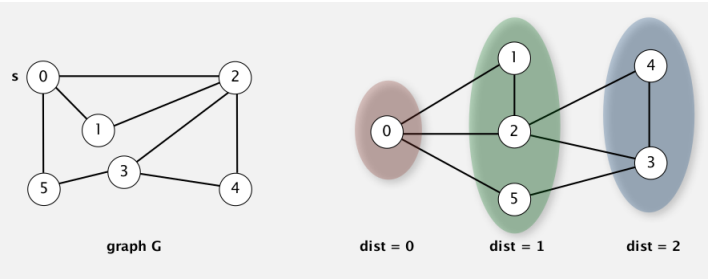


Removes 3 from the queue and checks its adjacent vertices 5, 4, and 2, which are marked.
Removes 4 from the queue and checks its adjacent vertices 3 and 2, which are marked



Single-source Shortest Paths

- Given a graph and a source vertex s, is there a path from s to a given target vertex v? If so, find a shortest such path (one with a minimal number of edges).
- In any connected graph G, BFS computes shortest paths from s to all other vertices in time proportional to E + V.
- This is because the queue always consists of ≥ 0 vertices of distance `k` from s, followed by ≥ 0 vertices of distance `k+1`.



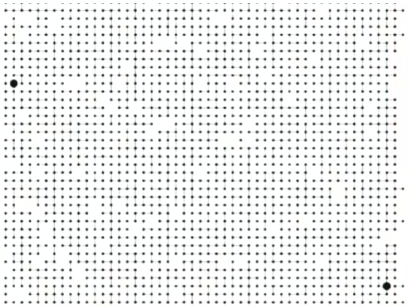
Implementation

```
public class BreadthFirstSearch{
    /*
      Same as before
      */
    public boolean hasPathTo(int v){ return marked[v]; }
    public Iterable <Integer> pathTo(int v){
        if (!hasPathTo(v)) return null;
        Stack<Integer> path = new Stack<>();
        for (int x = v; x != s; x = edgeTo[x])
            path.push(x);
        path.push(s);
        return path;
    }
}
```

- The `hasPathTo()` determines whether a given vertex v is connected to s.
- The `pathTo()` gets a path from s to v with the property that no other such path from s to v has fewer edges.

Connected Components in a Graph

- We say vertices v and w are connected if there is a path between them.
- The depth-first search can be used to find the connected components in a graph.
- For example: in the grid graph given below, how do you know if the two big dots are connected?
- The graph below has 63 connected components.



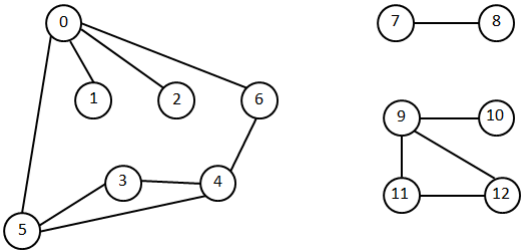
Connected Components in a Graph: Code

```
public class CC{
    private boolean[] marked;
    private int[] id;
    private int count;
    public CC(Graph G){
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int s = 0; s < G.V(); s++)
            if (!marked[s]){
                dfs(G, s);
                count++;
            }
    }
    private void dfs(Graph G, int v){
       marked[v] = true;
       id[v] = count;
       for (int w : G.adj(v))
           if (!marked[w])
              dfs(G, w);
    }
    public boolean connected(int v, int w){ return id[v] == id[w]; }
    public int id(int v){ return id[v]; }
    public int count(){ return count; }
}
```

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



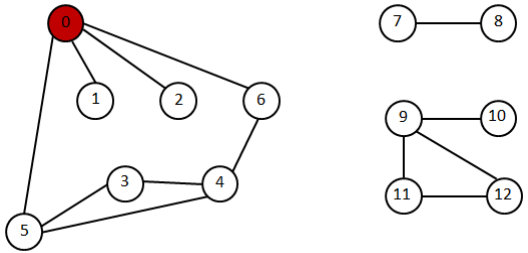
Standard graph

v	marked[]	id[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



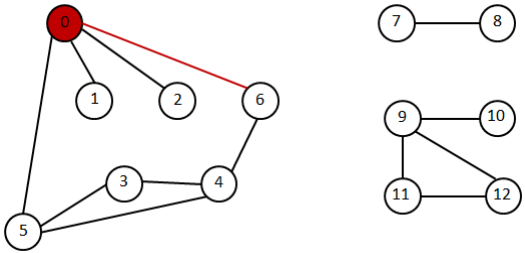
visit 0: check 6, check 2, check 1, and check 5

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



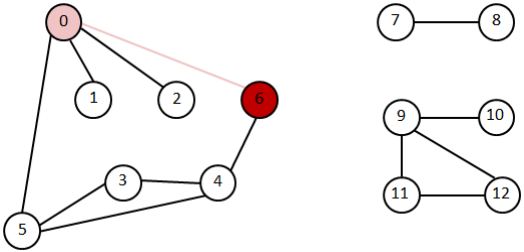
visit 0: check 6, check 2, check 1, and check 5

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



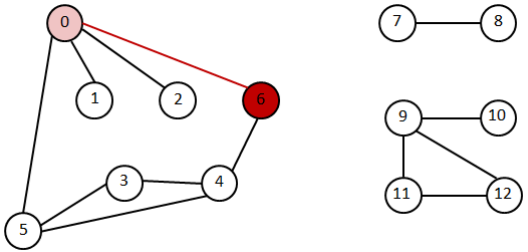
visit 6: check 0, and check 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



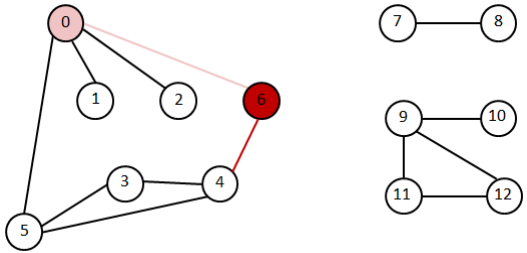
visit 6: check 0, and check 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



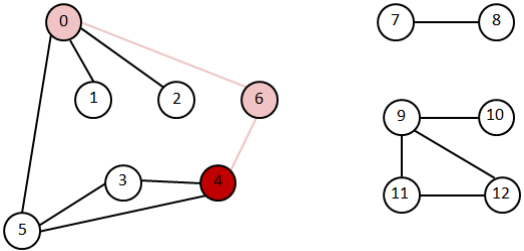
visit 6:, check 0, and check 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



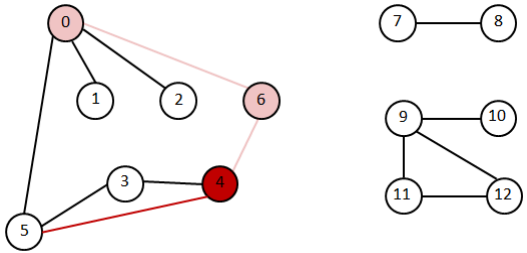
visit 4: check 5, check 6, and check 3

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	T	0
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



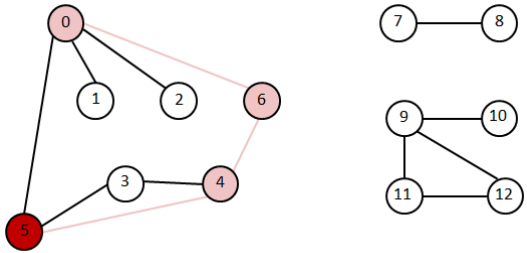
visit 4: check 5, check 6, and check 3

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	T	0
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



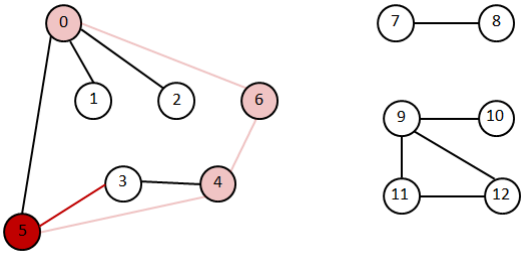
visit 5: check 3, check 4, and check 0

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



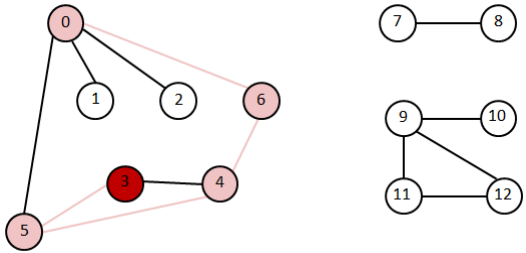
visit 5: check 3, check 4, and check 0

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	F	–
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



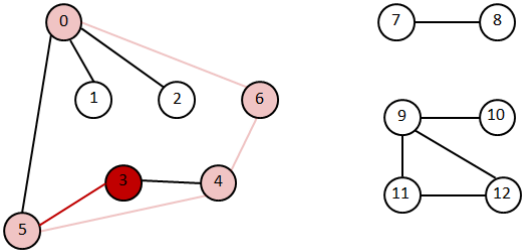
visit 3: check 5, and check 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



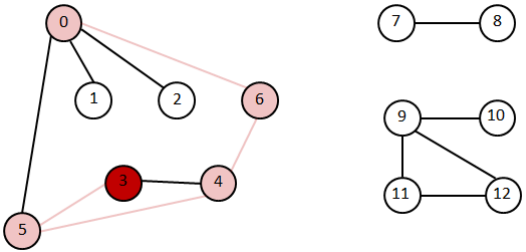
visit 3: check 5, and check 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



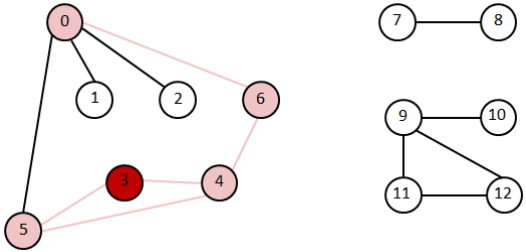
visit 3: check 5, and check 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



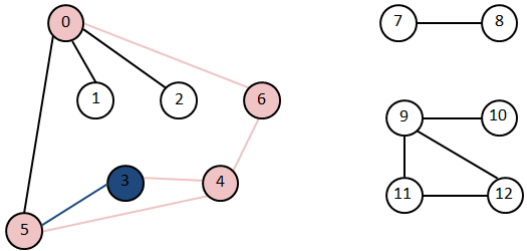
visit 3: check 5, and check 4

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



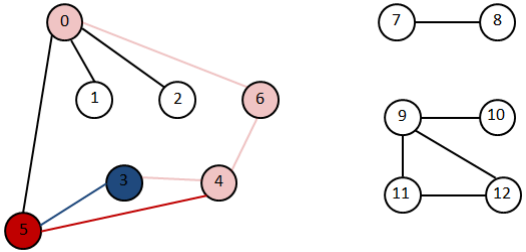
3 done

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



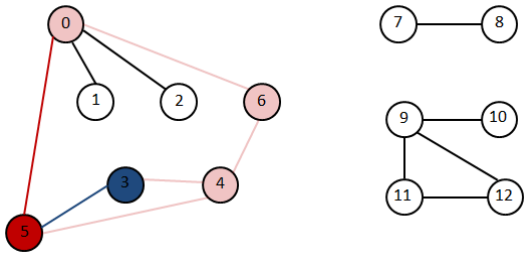
visit 5: check 3, check 4 and check 0

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



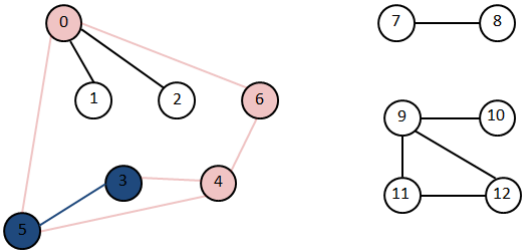
visit 5: check 3, check 4 and check 0

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



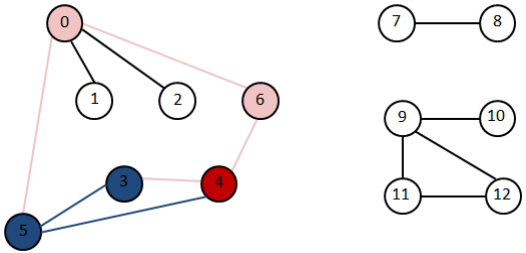
5 done.

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



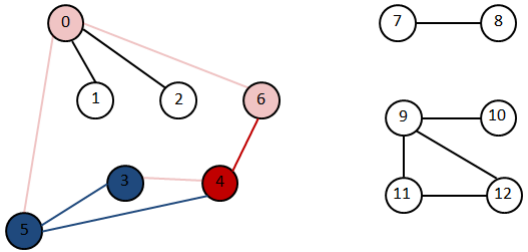
visit 4: check 5, check 6, and check 3

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



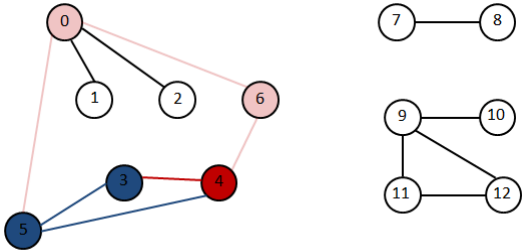
visit 4: check 5, check 6, and check 3

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



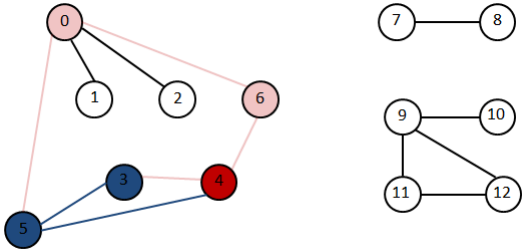
visit 4: check 5, check 6, and check 3

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



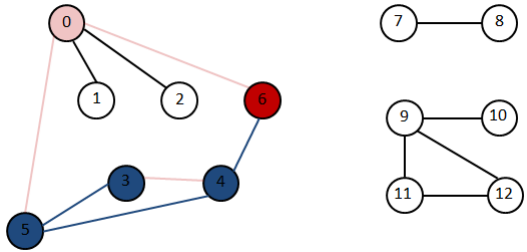
visit 4: check 5, check, and check 3

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



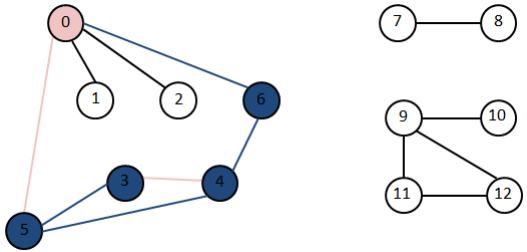
4 done

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



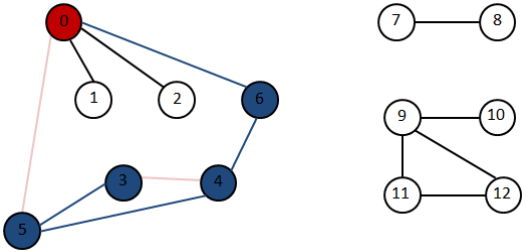
6 done

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



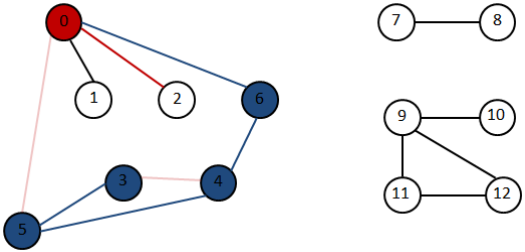
visit 0: check 6 check 2, check 1, and check 5

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



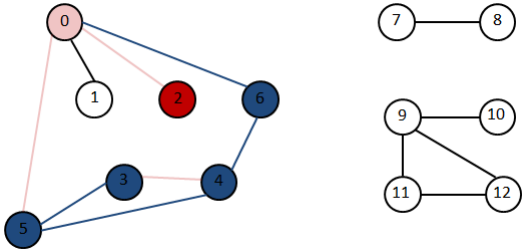
visit 0: check 6 check 2, check 1, and check 5

v	marked[]	id[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



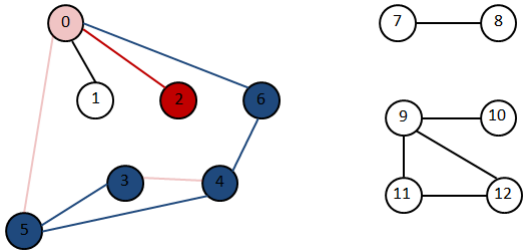
visit 2: check 0

v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



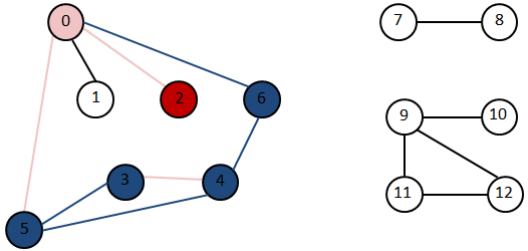
visit 2: check 0

v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



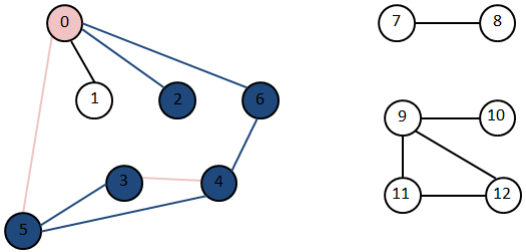
visit 2: check 0

v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



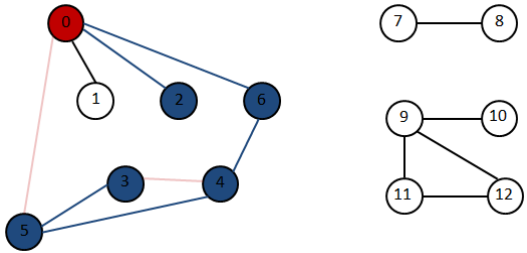
2 done

v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



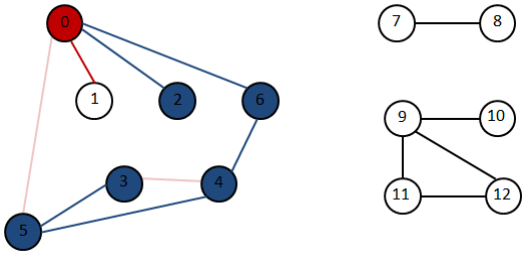
textcolorredvisit 0: check 6 check 2, check 1, and check 5

v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



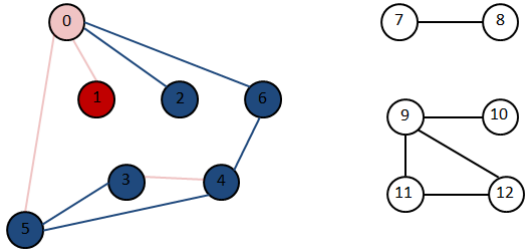
textcolorredvisit 0: check 6 check 2, check 1, and check 5

v	marked[]	id[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



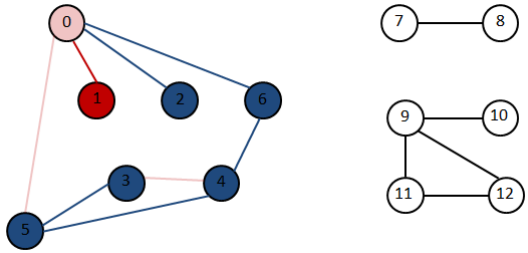
visit 1: check 0

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.



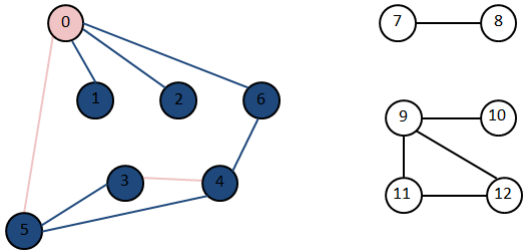
visit 1: check 0

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



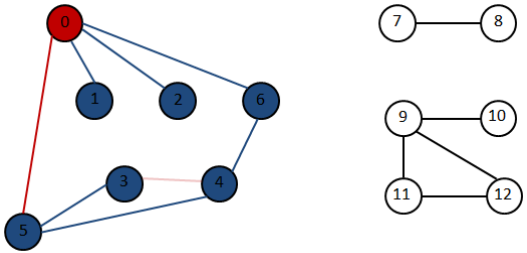
1 done

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex v

- Mark vertex v as visited
- Recursively visit all unmarked vertices adjacent to v.



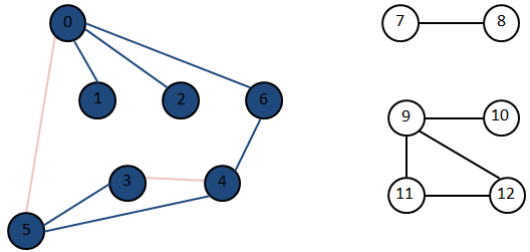
visit 0: check 6, check 2, check 1 , and check 5

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

To visit vertex *v*

- Mark vertex *v* as visited
- Recursively visit all unmarked vertices adjacent to *v*.

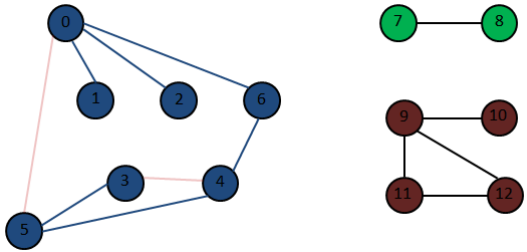


0 done

v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Example

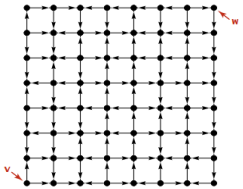
- After visiting all 13 vertices, we find three connected components, with ids 0, 1, and 2.



v	marked[]	id[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Directed Graphs

- In directed graphs, edges are one-way.
- A directed graph (or **digraph**) is a set of vertices and a collection of directed edges.
- A directed path in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence.
- We say that a vertex w is reachable from a vertex v if there is a directed path from v to w.
- For example, is w reachable from v in this digraph?



- Identifying with reachability can be challenging without the help of computer (graph traversals).

Digraph API

public class	Digraph	
	Digraph(int V)	*create a* V*-vertex digraph with no edges*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v->w *to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to* v *by edges pointing from* v
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

Digraph Implementation

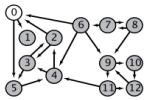
```
public class Digraph {
    private final int V;
    private int E;
    private List<Integer>[] adj;
    public Digraph(int V) {
        this.V = V;
        this.E = 0;
        adj = (LinkedList<Integer>[]) new
              LinkedList[V];
        for (int v = 0; v < V; v++)
            adj[v] = new LinkedList<Integer>();
    }
    public int V() { return V; }
    public int E() { return E; }
    public void addEdge(int v, int w) {
        adj[v].add(w); // v -> w
        E++;
    }
    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
    public Digraph reverse() {
        Digraph R = new Digraph(V);
        for (int v = 0; v < V; v++)
            for (int w : adj(v)) R.addEdge(w, v);
        return R;
    }
    public String toString(){
        String s = V + " vertices, " + E + "
              edges.\n";
        for(int i = 0; i < V; i++){
            s += (i + ": ");
            for(int w : this.adj(i))
                s += (w + " ");
            s += "\n";
        }
        return s;
    }
}
```

DFS and BFS in Digraphs

- DFS and BFS in digraph use the same method as in undirected graph. This is because every undirected graph is actually a digraph with edges in both directions.
- To visit a vertex v in DFS:

```
Mark v as visted
Recursively mark all unvisited
    vertices w pointing from v
```

- For example, trace of to find vertices reachable from vertex 0 in a digraph below:
  First,



marked[]	
0	T
1	
2	
3	
4	
5	
.	
.	
.	

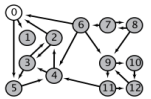
adj[]	
0	5 1
1	
2	0 3
3	5 2
4	3 2
5	4
.	
.	
.	

DFS and BFS in Digraphs

- DFS and BFS in digraph use the same method as in undirected graph. This is because every undirected graph is actually a digraph with edges in both directions.
- To visit a vertex v in DFS:

```
Mark v as visted
Recursively mark all unvisited
    vertices w pointing from v
```

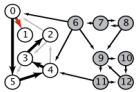
- For example, trace of to find vertices reachable from vertex 0 in a digraph below:
  First,



marked[]	
0	T
1	
2	
3	
4	
5	

adj[]	
0	5 1
1	
2	0 3
3	5 2
4	3 2
5	4

  Finally,



0	T
1	T
2	T
3	T
4	T
5	T

0	5 1
1	
2	0 3
3	5 2
4	3
5	4

Topological Sort

- A widely applicable problem-solving model is arranging for the completion of a set of jobs, under a set of constraints, by specifying when and how the jobs are to be performed.
- The most important type of constraints is precedence constraints:
  - certain tasks must be performed before certain others.
- For example, a college student planning a course schedule, under the constraint that certain courses are prerequisite for certain other courses.
- How can we schedule the courses such that they are all completed while still respecting the constraints (prerequisites) as shown below?

Topological Sort

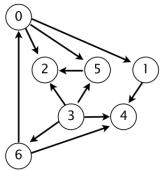
- **Problem:** Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?
- Digraph model. vertex = task; edge = precedence constraint.
- **Solution:** Use topological sort.
  - A topological sort redraws a directed acyclic graph (DAG) such that all its directed edges point from a vertex earlier in the order to a vertex later in the order.
  - Use DFS

tasks

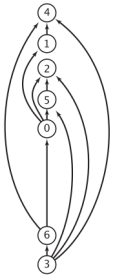
0. Algorithms
1. Complexity Theory
2. Artificial Intelligence
3. Intro to CS
4. Cryptography
5. Scientific Computing
6. Advanced Programming

directed edges

0→5	0→2
0→1	3→6
3→5	3→4
5→2	6→4
6→0	3→2
1→4	

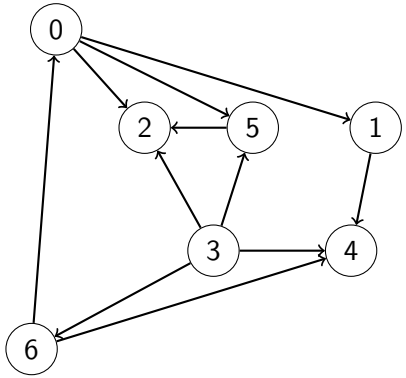


precedence constraint graph



Topological Sort

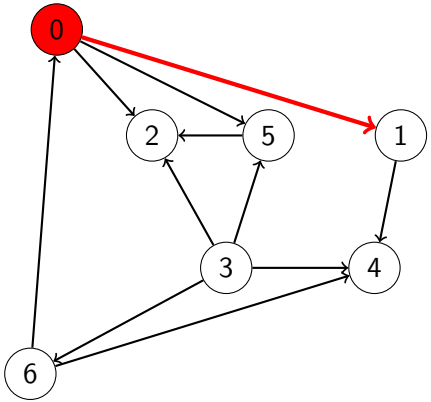
- Run depth-first search
- Return vertices in reverse postorder
- Postorder: Put the vertex on a queue after the recursive calls.
- Reverse postorder: Put the vertex on a stack after the recursive calls.



postorder

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

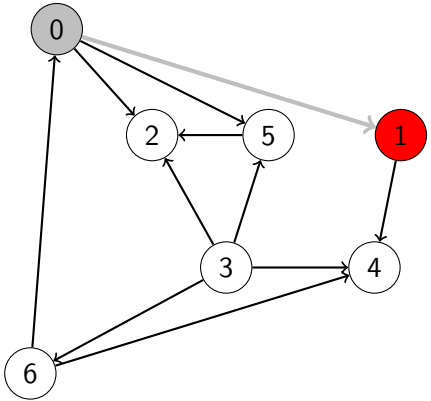


postorder

vist 0: check 1, check 2, check 5.

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

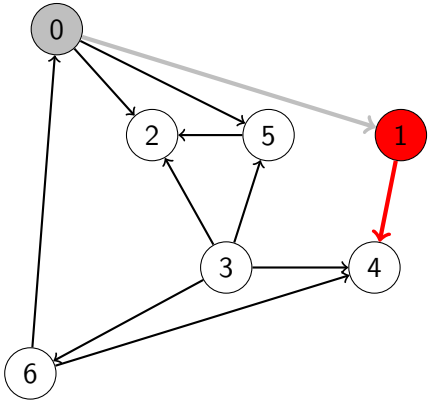


postorder

vist 1: check 4

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

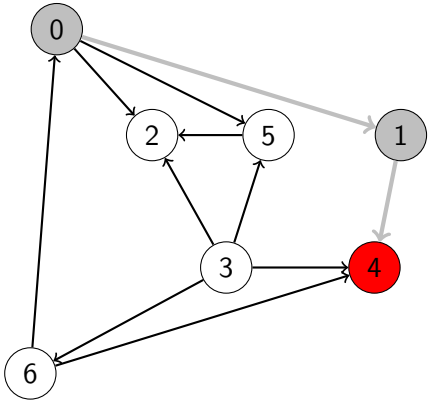


vist 1: check 4

postorder

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

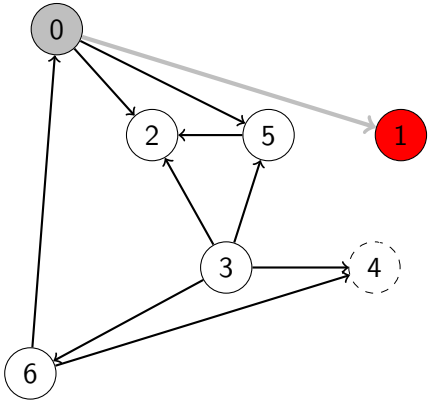


postorder
4

vist 4: 4 done

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

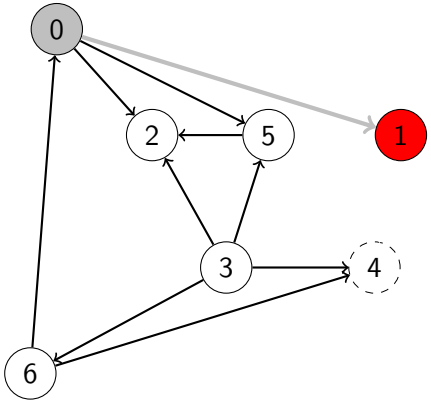


postorder
4

vist 4: 4 done

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

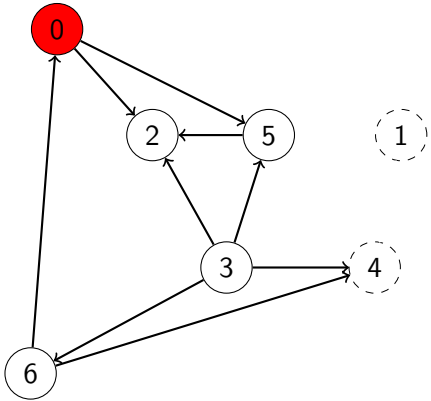


1 done

postorder
4 1

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

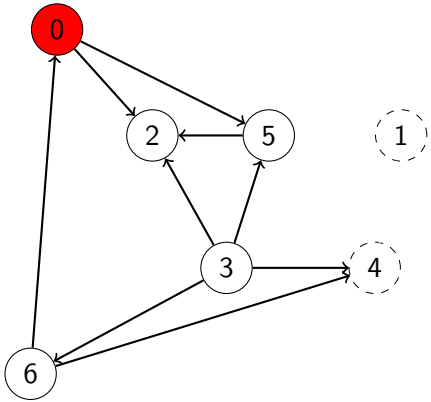


1 done

postorder
4 1

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

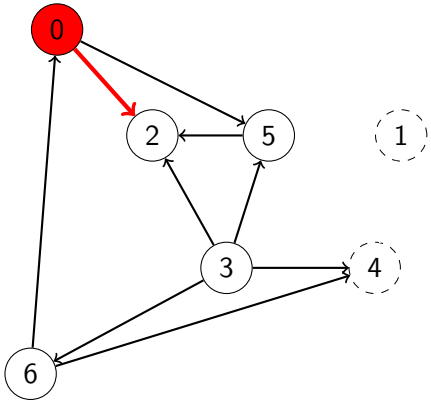


visit 0: check 1, check 2, and check 5

postorder
4 1

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

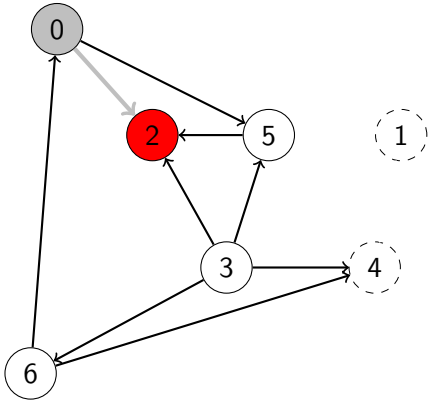


visit 0: check 1, check 2, and check 5

postorder
4 1

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

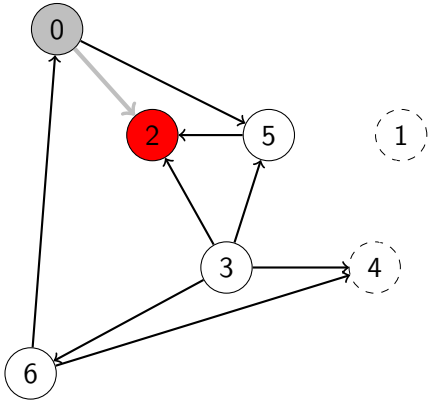


visit 2:

postorder
4 1

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

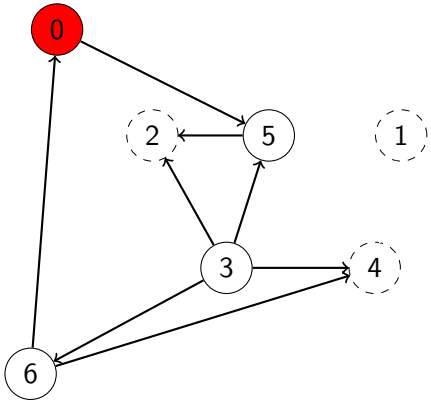


postorder
4 1 2

2 done:

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

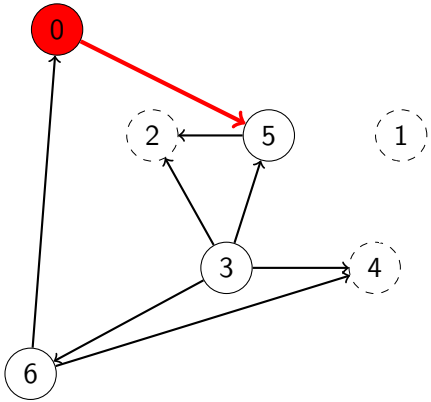


visit 0: check 1, check 2, and check 5

postorder
4 1 2

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

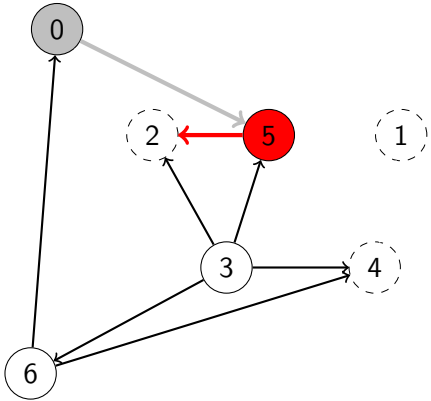


visit 0: check 1, check 2, and check 5

postorder
4 1 2

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

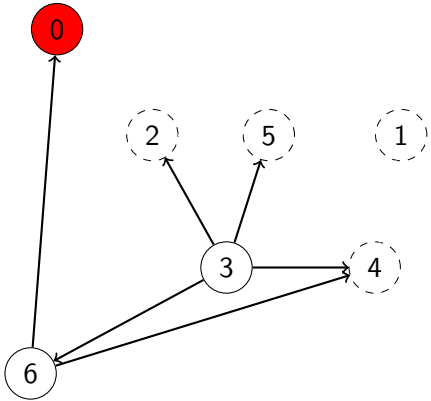


visit 5: check 2

postorder
4 1 2

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

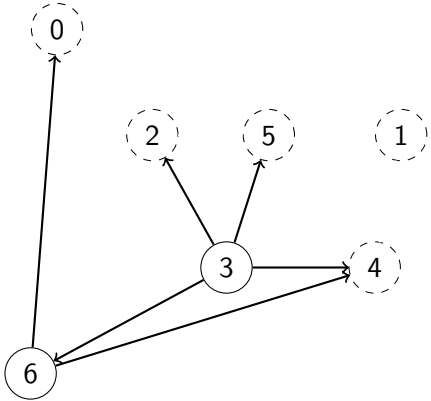


5 done

postorder
4 1 2 5

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

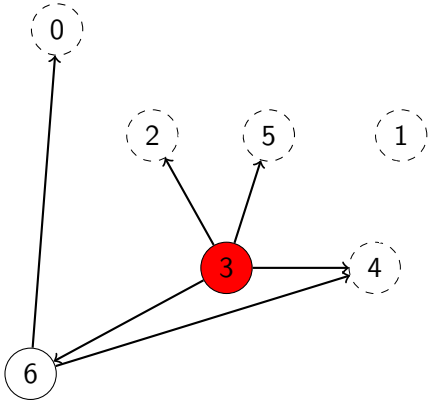


0 done

postorder
4 1 2 5 0

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

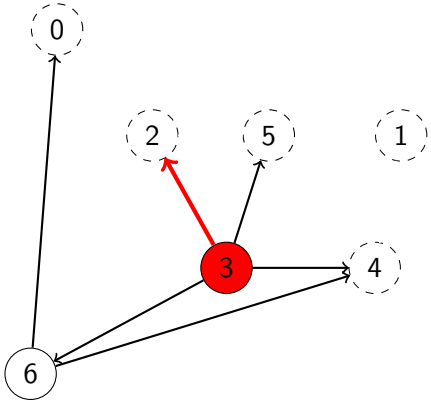


postorder
4 1 2 5 0

visit 3: check 2, check 4, check 5, and check 6

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

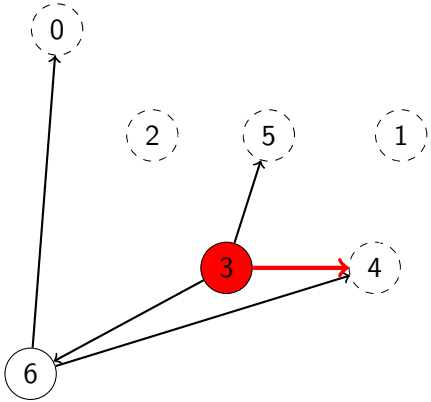


postorder
4 1 2 5 0

visit 3: check 2, check 4, check 5, and check 6

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

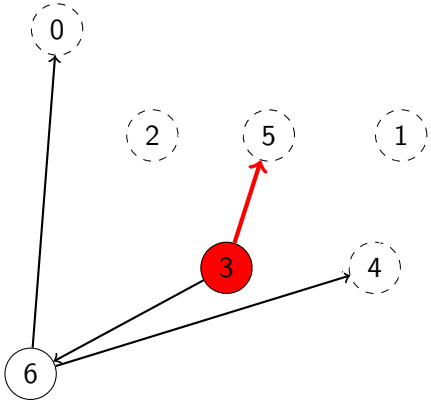


postorder
4 1 2 5 0

visit 3: check 2, check 4, check 5, and check 6

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

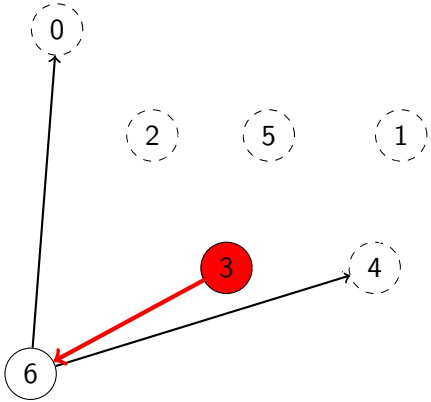


postorder
4 1 2 5 0

visit 3: check 2, check 4, check 5, and check 6

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

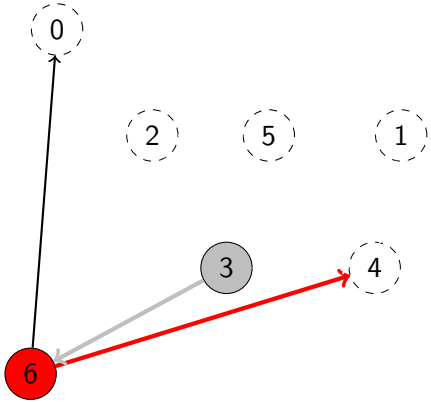


postorder
4 1 2 5 0

visit 3: check 2, check 4, check 5, and check 6

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

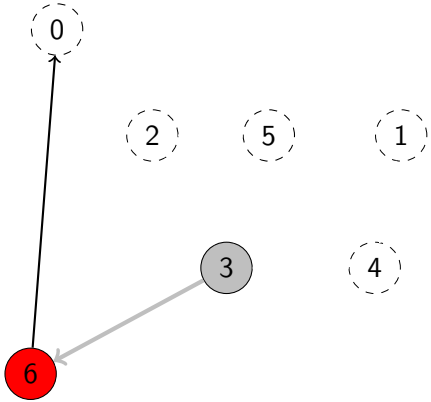


visit 6: check 4, and check 0

postorder
4 1 2 5 0

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

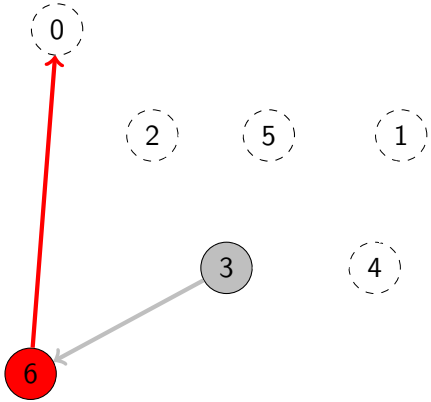


postorder
4 1 2 5 0

visit 6: check 4, and check 0

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

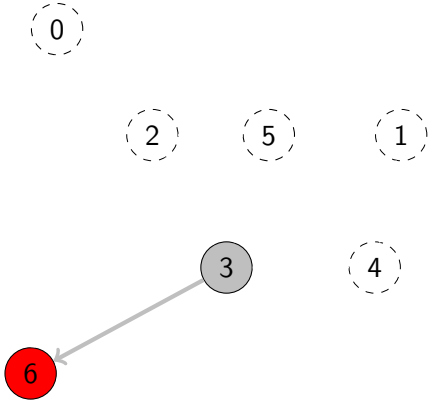


postorder
4 1 2 5 0

visit 6: check 4, and check 0

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

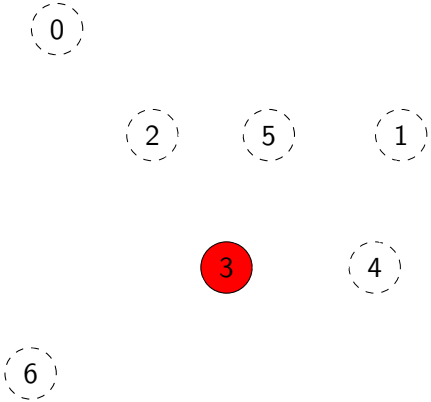


postorder
4 1 2 5 0 6

6 done:

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.



postorder
4 1 2 5 0 6

3 done:

Topological Sort

- Run depth-first search
- Return vertices in reverse postorder.

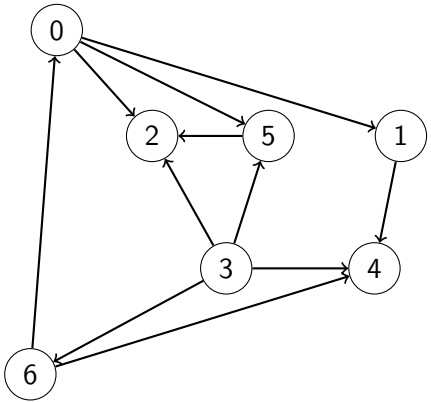


Abbildung 2: a directed acyclic graph

postorder
4 1 2 5 0 6 3

topological order (reverse postorder
3 6 0 5 2 1 4

Implementation

```
public class DepthFirstOrder{
    private boolean[] marked;
    private Stack<Integer> reversePost; // vertices in reverse postorder
    public DepthFirstOrder(Digraph G){
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }
    private void dfs(Digraph G, int v){
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
        reversePost.push(v);
    }
    public Iterable<Integer> reversePost(){ return reversePost; }
}
```

Minimum Spanning Trees

- A spanning tree of G is a subgraph T that is both a tree (connected and ascyclic) and spanning (includes all the vertices).
- Given: undirected graph with positive edge weights (connected).
- Goal: find a minimum weight spanning tree.
- Applications:
  - Soldering the shortest test of wires needed to connect a set of terminals on a circuit board.
  - Connecting a set of cities by telephone lines in such a way as to require the least amount of cable
  - Image registration and segmentation, etc.

Edge-Weighted Graph

Weighted-Edge

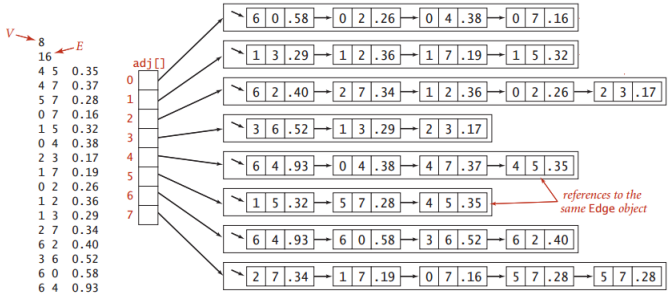
```
public class Edge implements Comparable<Edge>{
    private int v;
    private int w;
    private float weight;
    public Edge(int v, int w, float weight){
        this.v = v;
        this.w = w;
        this.weight = weight;
    }
    public float weight(){ return weight; }
    public int either(){ return v; }
    public int other(int vertex){
        if(vertex == v) return w;
        else if(vertex == w) return v;
        else throw new RuntimeException("Inconsistent edge.");
    }
    public int compareTo(Edge that){
        if(this.weight < that.weight) return -1;
        else if(this.weight > that.weight) return 1;
        else return 0;
    }
    public String toString(){
        return String.format("%d-%d %.2f", v, w, weight);
    }
}
```

For edge e: `int v = e.either(); w = e.other(v)`

Edge-Weighted Graph

```
public class EdgeWeightedGraph {
    private final int V;
    private int E;
    private List<Edge>[] adj;
    public EdgeWeightedGraph(int V){
        this.V = V;
        this.E = 0;
        adj = (LinkedList<Edge>[]) new LinkedList[V];
        for(int i = 0; i < V; i++) adj[i] = new LinkedList();
    }
    public int V(){ return V; }
    public int E(){ return E; }
    public void addEdge(Edge e){
        int v = e.either();
        int w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
        E++;
    }
    public Iterable<Edge> adj(int v){ return adj[v]; }
    public Iterable<Edge> edges(){
        List<Edge> edges = new LinkedList<>();
        for(int v = 0; v < V; v++)
            for(Edge e : adj[v])
                if(e.other(v) > v) edges.add(e);
        return edges;
    }
}
```

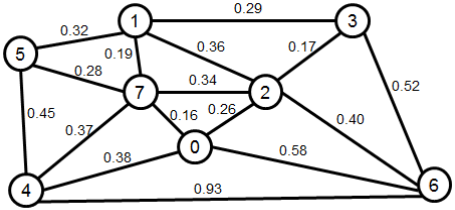
Edge-weighted graph representation



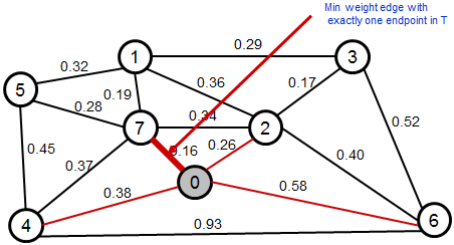
Prim's Algorithm

- A greedy algorithm to find MST.
- Start with 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one end point in T.
- Repeat until $|V| - 1$ edges.

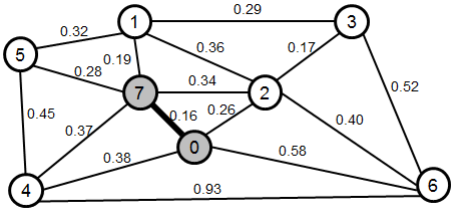
Example


1
3
5
7
2
0
4
6
0.29
0.32
0.36
0.17
0.19
0.28
0.34
0.52
0.45
0.16
0.26
0.40
0.37
0.38
0.58
0.93

Example


Min weight edge with
exactly one endpoint in T
0
1
2
3
4
5
6
7
0.29
0.32
0.36
0.17
0.19
0.28
0.34
0.52
0.45
0.16
0.26
0.37
0.40
0.38
0.58
0.93

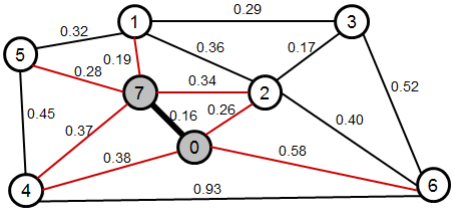
Example



MST edges:

0-7

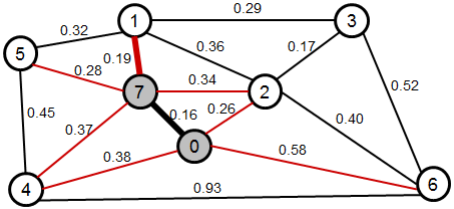
Example



MST edges:

0-7

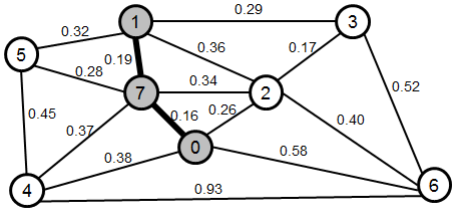
Example



MST edges:

0-7

Example

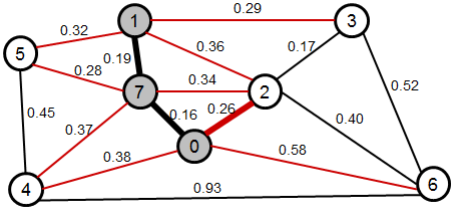


MST edges:

0-7

1-7

Example

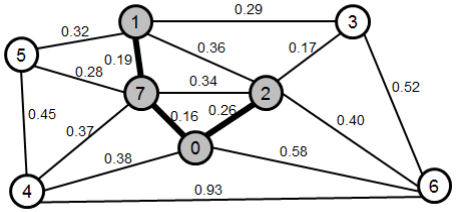


MST edges:

0-7

1-7

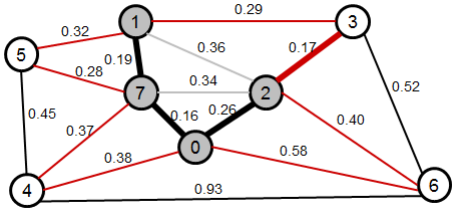
Example



MST edges:

0-7
1-7
0-2

Example



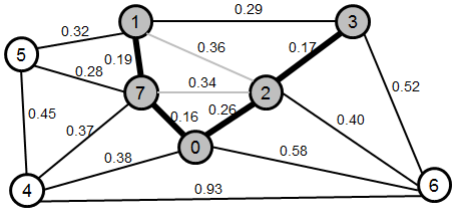
MST edges:

0-7

1-7

0-2

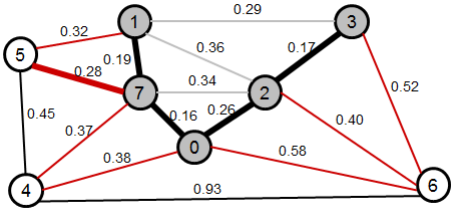
Example



MST edges:

0-7
1-7
0-2
2-3

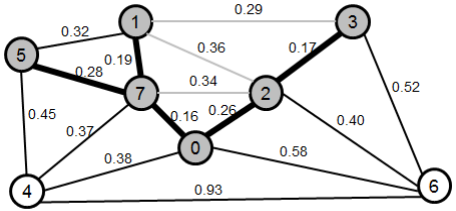
Example



MST edges:

0-7
1-7
0-2
2-3

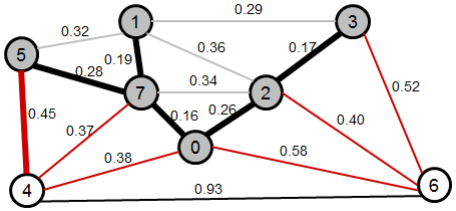
Example



MST edges:

0-7
1-7
0-2
2-3
5-7

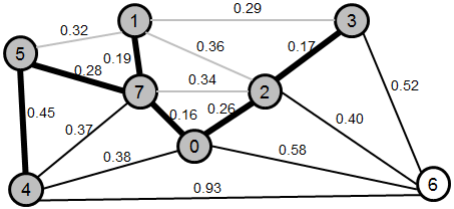
Example



MST edges:

0-7
1-7
0-2
2-3
5-7

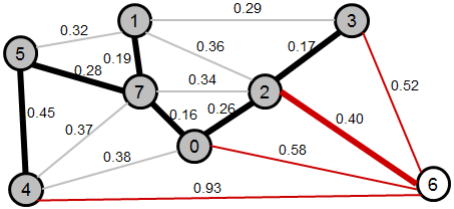
Example



MST edges:

0-7
1-7
0-2
2-3
5-7
5-4

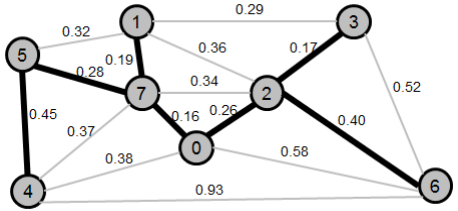
Example



MST edges:

0-7
1-7
0-2
2-3
5-7
5-4

Example



MST edges:

0-7
1-7
0-2
2-3
5-7
5-4
2-6

Prim MST Implementation

```
public class PrimMST {
    private boolean[] marked; // MST vertices
    private Queue<Edge> mst; // MST edges
    private PriorityQueue<Edge> pq; // crossing
        (and ineligible ) edges
    public PrimMST(EdgeWeightedGraph G){
        pq = new PriorityQueue<>();
        marked = new boolean[G.V()];
        mst = new LinkedList<>();
        visit (G, 0); // assumes G is connected
        while (!pq.isEmpty()){
            Edge e = pq.remove(); // Get
                lowest-weight
            int v = e.either(), w = e.other(v);
                // edge from pq.
            if (marked[v] && marked[w]) continue;
                // Skip if ineligible .
            mst.add(e); // Add edge to tree.
            if (!marked[v]) visit (G, v); // Add
                vertex to tree
            if (!marked[w]) visit (G, w); //
                (either v or w).
        }
    }
    private void visit (EdgeWeightedGraph G, int
        v){
        // Mark v and add to pq all edges from v
            to unmarked vertices.
        marked[v] = true;
        for (Edge e : G.adj(v))
            if (!marked[e.other(v)]) pq.add(e);
    }
    public Iterable <Edge> edges(){ return mst; }
}
```

- Running Time: $O(E \lg E)$

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?

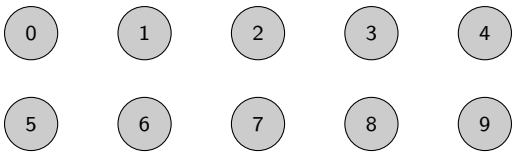
Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?

	0	1	2	3	4	5	6	7	8	9
`id[]`	0	1	2	3	4	5	6	7	8	9

(initially)

For example,

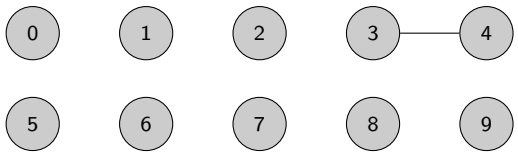


initially, `id`'s entry=index→all objects are independent (disconnected).

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?

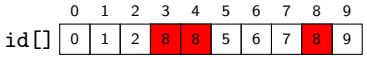
`union(4, 3)`



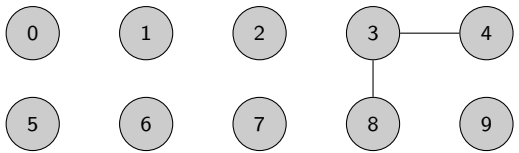
Change entries whose ids is equal to the first id to the second one.

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?



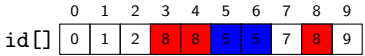

```
union(4, 3)
union(3, 8)
```



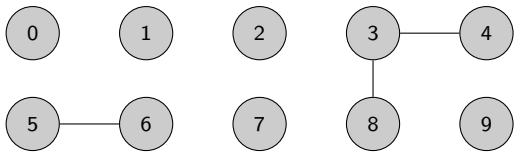
3 and 4 have to connect to 8, so both entries have to change to 8.

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?



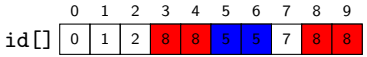

```
union(4, 3)
union(3, 8)
union (6, 5)
```



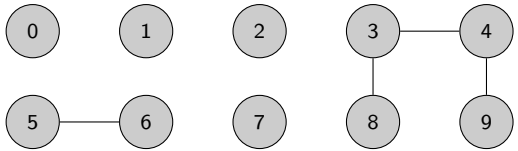
change 6 to 5.

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?



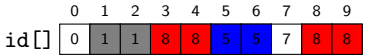

```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
```



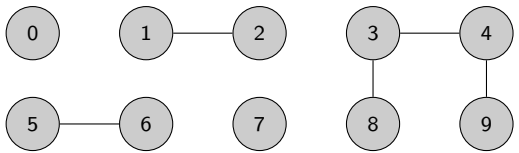
change 9s entry to be the same as 4.

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?




```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
```

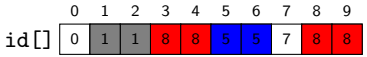


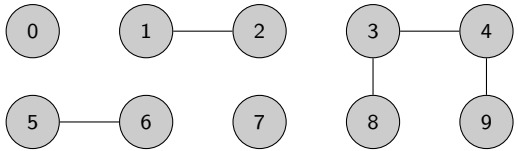
change 2s entry to 1.

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?

```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
connected(8, 9) ✓
```



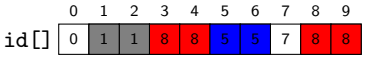


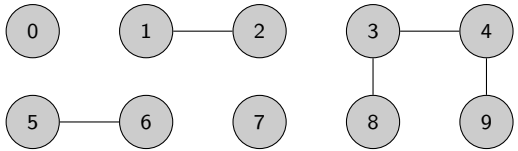
True. 8 and 9 are already connected.

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?

```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
connected(8, 9) ✓
connected(5, 0)x
```



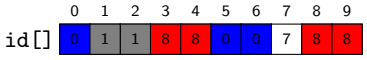


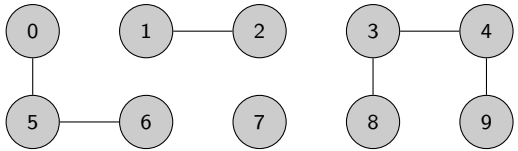
False. 5 and 0 have different entries.

Union Find

- The union find is a simple data structure, that, given a set of N objects, supports two operations:
  - `union(p, q)`: Connect two objects p and q.
  - `connected(p, q)`: Is there a path connecting the two objects p and q?
  - `find(p)`: In which component is object p?

```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
connected(8, 9) ✓
connected(5, 0)x
union(5, 0)
```





entries 5 and 6 change to 0.

Union Find Implementation

```
public class UF{
    private int[] id; // access to component id
    private int count; // number of components
    public UF(int N){
        // Initialize component id array.
        count = N;
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }
    public int count(){ return count; }
    public boolean connected(int p, int q){ return find(p) == find(q); }
    public int find(int p){ return id[p]; }
    public void union(int p, int q){
        // Put p and q into the same component.
        int pID = find(p);
        int qID = find(q);
        if (pID == qID) return;
        // Rename p's component to q's name.
        for (int i = 0; i < id.length; i++)
            if (id[i] == pID) id[i] = qID;
        count--;
    }
}
```

Kruskal's Algorithm

- Also a simple greedy algorithm.
- Consider edges in ascending order of weight.
  - Add next edge to tree T unless doing so would create a cycle.
- Uses priority queue to order edges by weight.
- Uses union find data structure to identify those that cause a cycle.

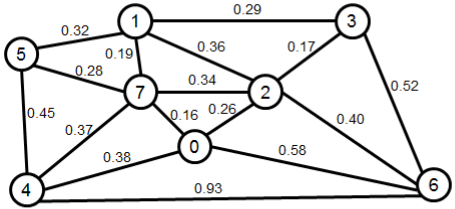
Kruskal's Algorithm

- Also a simple greedy algorithm.
- Consider edges in ascending order of weight.
  - Add next edge to tree T unless doing so would create a cycle.
- Uses priority queue to order edges by weight.
- Uses union find data structure to identify those that cause a cycle.
- **Implementation**

```
import java.util.*;
public class KruskalMST{
    private Queue<Edge> mst;
    public KruskalMST(EdgeWeightedGraph G){
        mst = new LinkedList<>();
        PriorityQueue<Edge> pq = new PriorityQueue<>();
        for(Edge e : G.edges())
            pq.add(e);
        UF uf = new UF(G.V());
        while (!pq.isEmpty() && mst.size() < G.V()-1){
            Edge e = pq.remove(); // Get min weight edge on pq
            int v = e.either(), w = e.other(v); // and its vertices.
            if (uf.connected(v, w)) continue; // Ignore ineligible edges.
            uf.union(v, w); // Merge components.
            mst.add(e); // Add edge to mst.
        }
    }
    public Iterable<Edge> edges(){ return mst; }
}
```

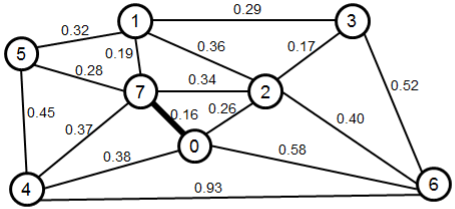
- Running Time: $O(E \lg E)$

Example



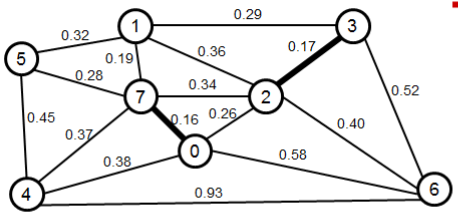
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



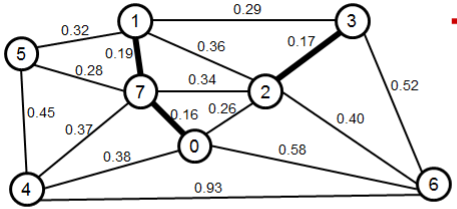
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



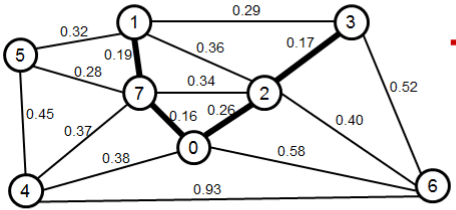
	0 – 7	0.16
→	2 – 3	0.17
	1 – 7	0.19
	0 – 2	0.26
	5 – 7	0.28
	1 – 3	0.29
	1 – 5	0.32
	2 – 7	0.34
	4 – 5	0.35
	1 – 2	0.36
	4 – 7	0.37
	0 – 4	0.38
	6 – 2	0.40
	3 – 6	0.52
	6 – 0	0.58
	6 – 4	0.93

Example



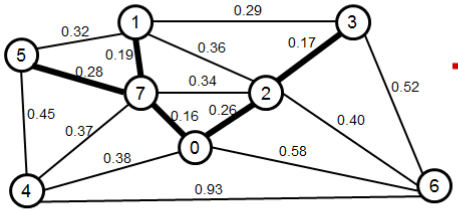
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



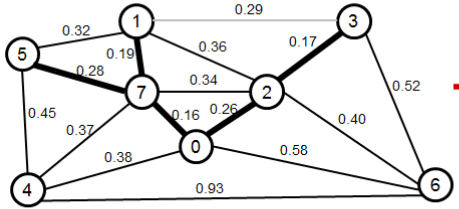
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



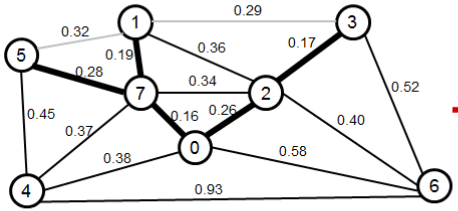
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



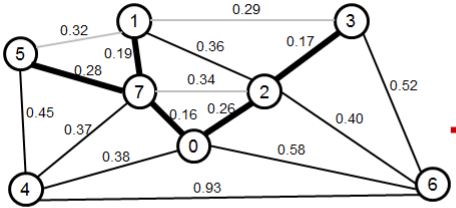
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



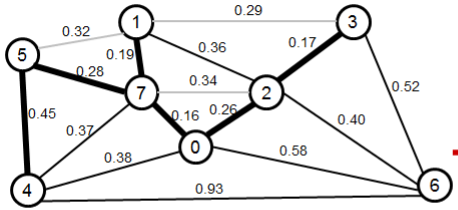
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



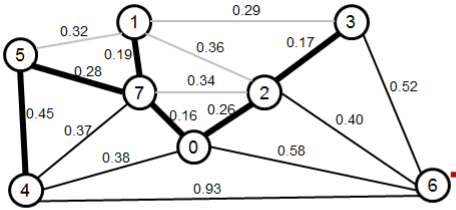
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



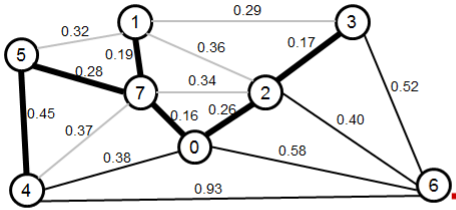
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



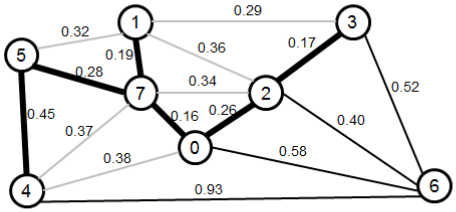
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



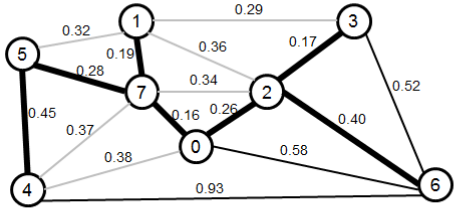
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



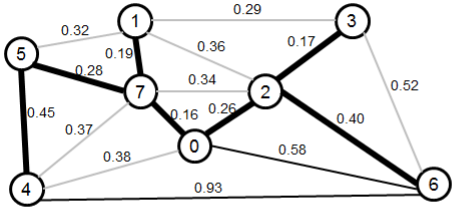
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



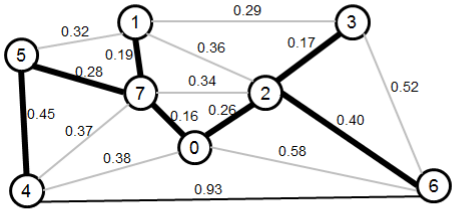
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



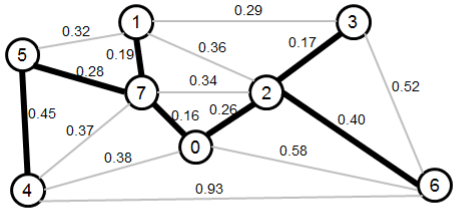
0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

Example



0 – 7	0.16
2 – 3	0.17
1 – 7	0.19
0 – 2	0.26
5 – 7	0.28
1 – 3	0.29
1 – 5	0.32
2 – 7	0.34
4 – 5	0.35
1 – 2	0.36
4 – 7	0.37
0 – 4	0.38
6 – 2	0.40
3 – 6	0.52
6 – 0	0.58
6 – 4	0.93

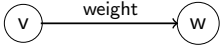
Single-source Shortest Path

- **Problem:** given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - SShortest path- minimum weight
  - Weight of path is sum of edges
  - E.g. a road map: what is the shortest path from Chapel Hill to Charlottesville?
- Shortest path variants:
  - Source-sink: from one vertex to another
  - Single-source: from one vertex to every other (E.g. navigation system in a car).
  - All pairs: between all pairs of vertices (E.g. Maps).

Weighted Directed Edge Implementation

- We need an API for processing directed edges.

```
public class DirectedEdge{
    private final int v; // edge source
    private final int w; // edge target
    private final double weight; // edge weight
    public DirectedEdge(int v, int w, double weight){
        this.v = v;
        this.w = w;
        this.weight = weight;
    }
    public double weight(){ return weight; }
    public int from(){ return v; }
    public int to(){ return w; }
    public String toString(){ return String.format("%d->%d %.2f", v, w, weight); }
}
```

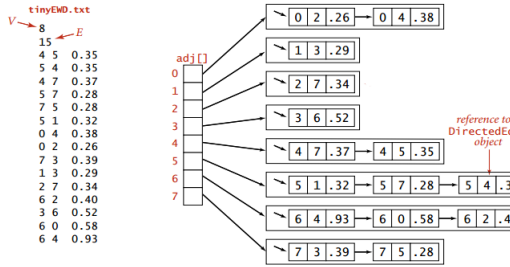


- Expression for processing edge e: `int v = e.from(), int w = e.to();`

Edge-Weighted Directed Graph

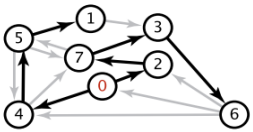
```
public class EdgeWeightedDigraph{
    private final int V; // number of vertices
    private int E; // number of edges
    private List<DirectedEdge>[] adj; // adjacency lists
    public EdgeWeightedDigraph(int V){
        this.V = V;
        this.E = 0;
        adj = (LinkedList<DirectedEdge>[]) new LinkedList[V];
        for (int v = 0; v < V; v++)
            adj[v] = new LinkedList<DirectedEdge>();
    }
    public int V() { return V; }
    public int E() { return E; }
    public void addEdge(DirectedEdge e){
        adj[e.from()].add(e);
        E++;
    }
    public Iterable<DirectedEdge> adj(int v){ return adj[v]; }
    public Iterable<DirectedEdge> edges(){
        List<DirectedEdge> edges = new LinkedList<DirectedEdge>();
        for (int v = 0; v < V; v++)
            for (DirectedEdge e : adj[v])
                edges.add(e);
        return edges;
    }
}
```

Example of Edge-weghted digraph



Data Structures for Single Source Shortest Paths

- **Goal:** Find the shortest path from s to every other vertex (a shortest path trees (SPT)).
- The data structures are:
  - **Edges on the shortest-paths tree**: we use a parent-edge representation in the form of a vertex-indexed array `edgeTo[]`. `edgeTo[v]` is edge that connects v to its parent in the tree (the last edge on a shortest path from s to v).
  - **Distance to the source**: We use a vertex-indexed array `distTo[]` such that `distTo[v]` is the length of the shortest known path from s to v.
- By convention, `edgeTo[s]` is `null` and `distTo[s]` is 0. Also distances to vertices that are not reachable from the source are all `Double.POSITIVE_INFINITY`.
- E.g. Shortest-path trees from 0



	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

Data Structures for Single Source Shortest Paths

- **Goal:** Find the shortest path from s to every other vertex (a shortest path trees (SPT)).
- The data structures are:
  - **Edges on the shortest-paths tree**: we use a parent-edge representation in the form of a vertex-indexed array `edgeTo[]`. `edgeTo[v]` is edge that connects v to its parent in the tree (the last edge on a shortest path from s to v).
  - **Distance to the source**: We use a vertex-indexed array `distTo[]` such that `distTo[v]` is the length of the shortest known path from s to v.
- By convention, `edgeTo[s]` is `null` and `distTo[s]` is 0. Also distances to vertices that are not reachable from the source are all `Double.POSITIVE_INFINITY`.

```
public double distTo(int v){ return distTo[v]; }
public boolean hasPathTo(int v){ return distTo[v] < Double.POSITIVE_INFINITY; }
public Iterable <DirectedEdge> pathTo(int v){
    if (!hasPathTo(v)) return null;
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}
```

Trace of `pathTo(6)`

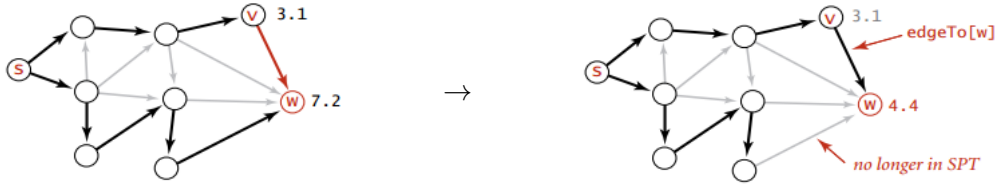
e	path
3->6	
7->3	3->6
2->7	7->3 3->6
0->2	2->7 7->3 3->6
null	0->2 2->7 7->3 3->6

Edge Relaxation

- Shortest-paths implementations are based on a simple operation known as relaxation.
- To relax edge $e = v \rightarrow w$
  - `distTo[v]` is the length of shortest known path from s to v.
  - `distTo[w]` is the length of shortest known path from s to w.
  - `edgeTo[w]` is the last edge on shortest known path from s to w.
  - If $e = v \rightarrow w$ gives shorter path to w through v update both distTo[w] and edgeTo[w].
    - `distTo[w] = distTo[v] + e.weight()` and
    - `edgeTo[w] = e`

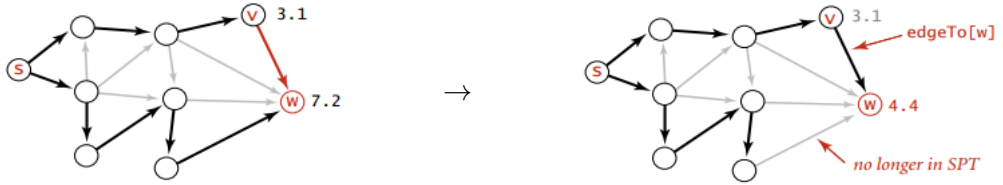
Edge Relaxation

- E.g eligible $e = v \rightarrow w$ (`weight(e) = 1.3`)

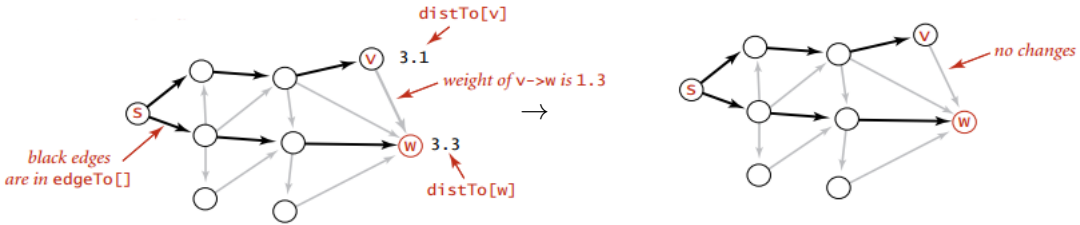


Edge Relaxation

- E.g eligible $e = v \rightarrow w$ (`weight(e) = 1.3`)



- E.g ineligible $e = v \rightarrow w$



Edge Relaxation

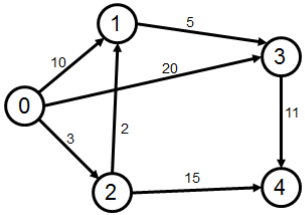
- Shortest-paths implementations are based on a simple operation known as relaxation.
- To relax edge $e = v \rightarrow w$
  - `distTo[v]` is the length of shortest known path from s to v.
  - `distTo[w]` is the length of shortest known path from s to w.
  - `edgeTo[w]` is the last edge on shortest known path from s to w.
  - If $e = v \rightarrow w$ gives shorter path to w through v update both distTo[w] and edgeTo[w].
    - `distTo[w] = distTo[v] + e.weight()` and
    - `edgeTo[w] = e`

```
private void relax(EdgeWeightedDigraph G, int v){
    for (DirectedEdge e : G.adj(v)){
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()){
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```

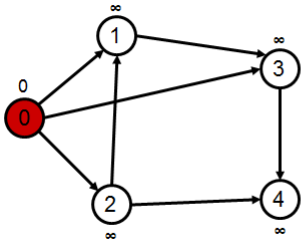
Dijkstra's Algorithm

- Solves the single-source shortest-paths problem in edge-weighted digraphs with nonnegative weights.
- Similar to breadth-first search
  - Grow a tree gradually, advancing from vertices taken from a queue
- Consider vertices in increasing order of distance from the source vertex
- Add vertex to tree and relax all edges pointing from that tree.

Dijkstra's Algorithm Example

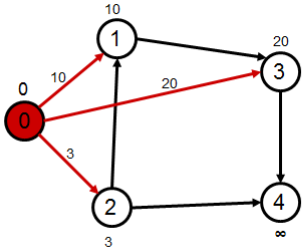


Dijkstra's Algorithm Example



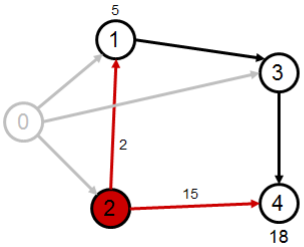
v	D[]	edge
0	0	
1	∞	
2	∞	
3	∞	
4	∞	

Dijkstra's Algorithm Example



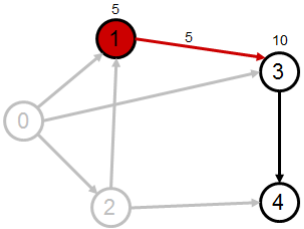
v	D[]	edge
➡ 0	0	-
1	10	0-1
2	3	0-2
3	20	0-3
4	∞	

Dijkstra's Algorithm Example



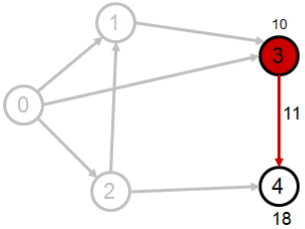
v	D[]	edge
0	0	-
1	5	2-1
➡ 2	3	0-2
3	20	0-3
4	18	2-4

Dijkstra's Algorithm Example



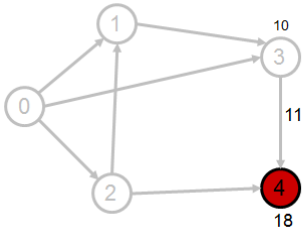
v	D[]	edge
0	0	-
➡ 1	5	2-1
2	3	0-2
3	10	1-3
4	18	2-4

Dijkstra's Algorithm Example



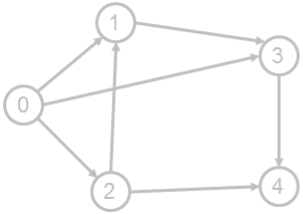
v	D[]	edge
0	0	-
1	5	2-1
2	3	0-2
➡ 3	10	1-3
4	18	2-4

Dijkstra's Algorithm Example



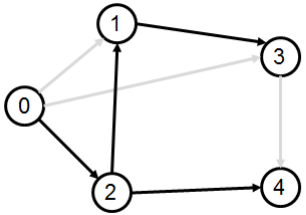
v	D[]	edge
0	0	-
1	5	2-1
2	3	0-2
3	10	1-3
➡ 4	18	2-4

Dijkstra's Algorithm Example



v	D[]	edge
0	0	-
1	5	2-1
2	3	0-2
3	10	1-3
4	18	2-4

Dijkstra's Algorithm Example



v	D[]	edge
0	0	-
1	5	2-1
2	3	0-2
3	10	1-3
4	18	2-4

Index Priority Queue

- Used to refer to keys that are on a priority queue.
- Associates a unique integer *index* with each item.
- Important in implementing Dijkstra's algorithm.
- Index Priority Queue API

```
public class IndexMinPQ<Item extends Comparable<Item>>
```

	`IndexMinPQ(int maxN)`	*create a priority queue of capacity* `maxN` *with possible indices between* `0` *and* `maxN-1`
`void`	`insert(int k, Item item)`	*insert* `item`*; associate it with* `k`
`void`	`change(int k, Item item)`	*change the item associated with* `k` *to* `item`
`boolean`	`contains(int k)`	*is* `k` *associated with some item?*
`void`	`delete(int k)`	*remove* `k` *and its associated item*
`Item`	`min()`	*return a minimal item*
`int`	`minIndex()`	*return a minimal item's index*
`int`	`delMin()`	*remove a minimal item and return its index*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`int`	`size()`	*number of items in the priority queue*

Index Priority Queue Implementation

```
public class IndexMinPQ<K extends Comparable<K>> {
    private int N;       // number of elements on PQ
    private int[] pq;    // binary heap using 1-based indexing
    private int[] qp;    // inverse of pq - qp[pq[i]] = pq[qp[i]] = i
    private K[] keys;    // keys[i] = priority of i
    public IndexMinPQ(int NMAX) {
        keys = (Key[]) new Comparable[maxN + 1];
        pq   = new int[NMAX + 1];
        qp   = new int[NMAX + 1]; // make this of length NMAX??
        for (int i = 0; i <= NMAX; i++) qp[i] = -1;
    }
    public boolean contains(int k) { return qp[k] != -1; }
    public void insert(int k, K key) {
        if (contains(k)) throw new RuntimeException("item already
            in pq");
        N++;
        qp[k] = N;
        pq[N] = k;
        keys[k] = key;
        upheap(N);
    }
    // delete a minimal key and returns its associated index
    public int delMin() {
        if (N == 0) throw new RuntimeException("Priority queue
            underflow");
        int min = pq[1];
        exch(1, N--);
        downheap(1);
        qp[min] = -1;          // delete
        keys[pq[N+1]] = null;  // to help with garbage
            collection
        pq[N+1] = -1;          // not needed
        return min;
    }
    // return a minimal key
    public K minK() {
        if (N == 0) throw new RuntimeException("Priority
            queue underflow");
        return keys[pq[1]];
    }

    // change the key associated with index k
    public void change(int k, K key) {
        if (!contains(k)) throw new RuntimeException("item is
            not in pq");
        keys[k] = key;
        upheap(qp[k]);
        downheap(qp[k]);
    }
    private void upheap(int k) // same as in chapter 4
    private void downheap(int k) // same as in chapter 4
```

Dijkstra's Algorithm Implementation

```
public class DijkstraSP{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;
    public DijkstraSP(EdgeWeightedDigraph G, int s){
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        pq.insert(s, 0.0);
        while (!pq.isEmpty())
            relax(G, pq.delMin());
    }
    private void relax(EdgeWeightedDigraph G, int v){
        for(DirectedEdge e : G.adj(v)){
            int w = e.to();
            if (distTo[w] > distTo[v] + e.weight()){
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) pq.change(w, distTo[w]);
                else pq.insert(w, distTo[w]);
            }
        }
    }
    public double distTo(int v) // discussed previously
    public boolean hasPathTo(int v) // discussed previously
    public Iterable<DirectedEdge> pathTo(int v) // discussed
        previously
}
```

- Running Time: $O(E \lg V)$

Exercise

Show the shortest path generated by running Dijkstra's shortest-paths algorithm on the graph below of beginning at Vertex 0.