# Review questions for basic graph algorithms

## Summary

- Vertex representation: using integers
- Three common representation of graph:
  - set of edges
  - adjacency matrix
  - adjacency list: most commonly used because 1) graphs in practice are often sparse; 2) graph algorithms often need to iterate over vertices adjacent to a vertex.
    - Each edge (u,w) is added to adjacency list twice: once to the adjacency list of u and the other to the adjacency list of w. Thus, a self-loop on u or a parallel edge from/to u will appear twice in the adjacency list of u.
- API of undirected graph
- design pattern for graph processing: decouple graph data type from graph processing
  - create a graph object
  - pass the graph to a graph-processing routine
  - query the graph-processing routine for info
- depth-first search: mimic maze exploration
- breadth-first search
- connected component
- complexity of common graph algorithms:
  - is a graph bipartite?
  - Euler cycle: trace each edge once
  - Halmilton cycle: trace each vertex once (NP complete)
  - planarity: linear time DFS algorithm discovered by Tarjan in 70s but probably too complex for most practioners
  - graph iso-morphism (open problem)

## Typical graph-processing code

*compute the degree of* v

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

*compute maximum degree*

```
public static int maxDegree(Graph G)
{
   int max = 0;
   for (int v = 0; v < G.V(); v++)
      if (degree(G, v) > max)
         max = degree(G, v);
   return max;
}
```

*compute average degree*

```
public static double averageDegree(Graph G)
{  return 2.0 * G.E() / G.V();  }
```

*count self-loops*

```
public static int numberOfSelfLoops(Graph G)
{
   int count = 0;
   for (int v = 0; v < G.V(); v++)
      for (int w : G.adj(v))
         if (v == w) count++;
   return count/2;   // each edge counted twice
}
```

## Graph API

| | public class Graph | |
|---|---|---|
| | Graph(int V) | *create an empty graph with V vertices* |
| | Graph(In in) | *create a graph from input stream* |
| void | addEdge(int v, int w) | *add an edge v-w* |
| Iterable<Integer> | adj(int v) | *vertices adjacent to v* |
| int | V() | *number of vertices* |
| int | E() | *number of edges* |
| String | toString() | *string representation* |

```
In in = new In(args[0]);
Graph G = new Graph(in);

for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "-" + w);
```

read graph from input stream

print out each edge (twice)

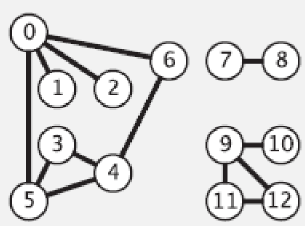
Graph input format.

tinyG.txt

*V* → 13

13 ← *E*

```
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```




```
% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
…
12-11
12-9
```

*count self-loops*

```
public static int numberOfSelfLoops(Graph G)
{
   int count = 0;
   for (int v = 0; v < G.V(); v++)
      for (int w : G.adj(v))
         if (v == w) count++;
   return count/2;   // each edge counted twice
}
```

## Adjacency-list graph representation: Java implementation

```
public class Graph
{
   private final int V;
   private Bag<Integer>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists ( using Bag data type )

create empty graph with V vertices

add edge v–w (parallel edges and self-loops allowed)

iterator for vertices adjacent to v

| representation | space | add edge | edge between v and w? | iterate over vertices adjacent to v? |
|---|---|---|---|---|
| list of edges | E | 1 | E | E |
| adjacency matrix | $V^2$ | 1 * | 1 | V |
| adjacency lists | E + V | 1 | degree(v) | degree(v) |

* disallows parallel edges

## Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

- Create a Graph object.
- Pass the Graph to a graph-processing routine.
- Query the graph-processing routine for information.

| public class Paths | | |
|---|---|---|
| | `Paths(Graph G, int s)` | *find paths in G from source s* |
| `boolean` | `hasPathTo(int v)` | *is there a path from s to v?* |
| `Iterable<Integer>` | `pathTo(int v)` | *path from s to v; null if no such path* |

```
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
   if (paths.hasPathTo(v))
      StdOut.println(v);
```

print all vertices connected to s

## Depth-first search

```
public class DepthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int s;

   public DepthFirstSearch(Graph G, int s)
   {
      ...
      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);
            edgeTo[w] = v;
         }
   }
}
```

marked[v] = true if v connected to s

edgeTo[v] = previous vertex on path from s to v

initialize data structures

find vertices connected to s

recursive DFS does the work

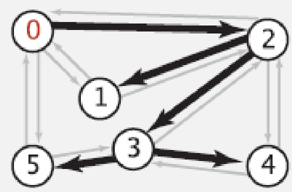
## Depth-first search properties

Proposition. After DFS, can find vertices connected to $s$ in constant time and can find a path to $s$ (if one exists) in time proportional to its length.

Pf. edgeTo[] is parent-link representation of a tree rooted at s.

```
public boolean hasPathTo(int v)
{  return marked[v];  }

public Iterable<Integer> pathTo(int v)
{
   if (!hasPathTo(v)) return null;
   Stack<Integer> path = new Stack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
      path.push(x);
   path.push(s);
   return path;
}
```




edgeTo[]

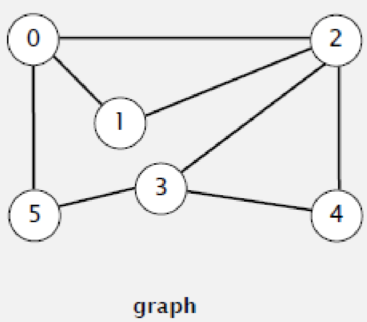
| | |
|---|---|
| 0 | |
| 1 | 2 |
| 2 | 0 |
| 3 | 2 |
| 4 | 3 |
| 5 | 3 |

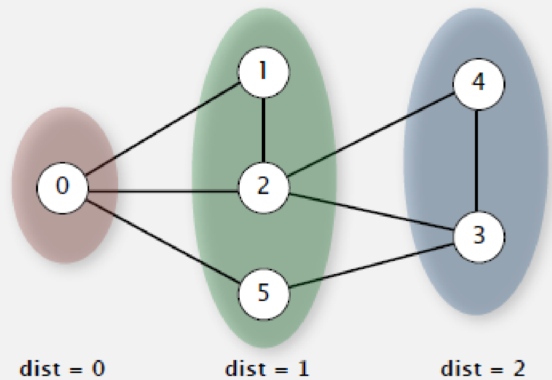
## Breadth-first search properties

Proposition. BFS computes shortest paths (fewest number of edges) from $s$ to all other vertices in a graph in time proportional to $E + V$.

Pf. [correctness] Queue always consists of zero or more vertices of distance $k$ from $s$, followed by zero or more vertices of distance $k + 1$.

Pf. [running time] Each vertex connected to $s$ is visited once.



## Breadth-first search

```
public class BreadthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   …

   private void bfs(Graph G, int s)
   {
      Queue<Integer> q = new Queue<Integer>();
      q.enqueue(s);
      marked[s] = true;
      while (!q.isEmpty())
      {
         int v = q.dequeue();
         for (int w : G.adj(v))
         {
            if (!marked[w])
            {
               q.enqueue(w);
               marked[w] = true;
               edgeTo[w] = v;
            }
         }
      }
   }
}
```

## Connectivity queries

Def. Vertices $v$ and $w$ are connected if there is a path between them.

Goal. Preprocess graph to answer queries of the form *is $v$ connected to $w$?* in constant time.

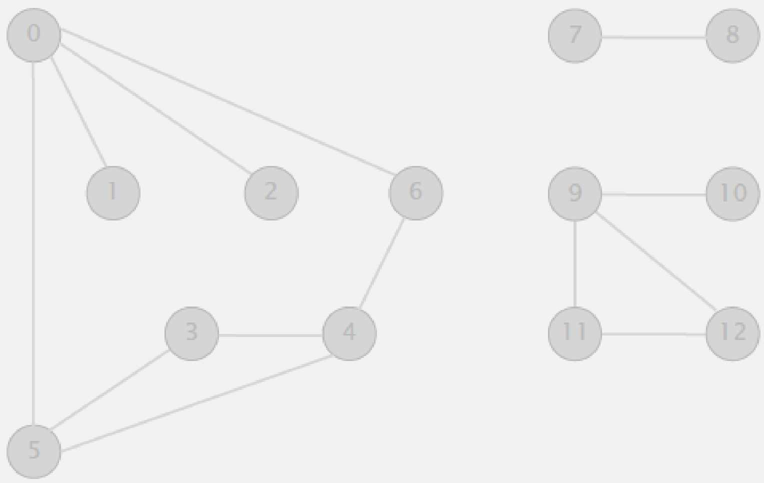
| public class CC | | |
|---|---|---|
| | `CC(Graph G)` | *find connected components in G* |
| `boolean` | `connected(int v, int w)` | *are v and w connected?* |
| `int` | `count()` | *number of connected components* |
| `int` | `id(int v)` | *component identifier for v* |

Union-Find? Not quite.
Depth-first search. Yes. [next few slides]

## Connected components demo

To visit a vertex $v$ :

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




| 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 |

**done**

## Finding connected components with DFS

```
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 v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

   public int count()
   public int id(int v)
   private void dfs(Graph G, int v)

}
```

id[v] = id of component containing v

number of components

run DFS from one vertex in each component

see next slide

## Finding connected components with DFS (continued)

```
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

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);
}
```

number of components

id of component containing v

all vertices discovered in same call of dfs have same id

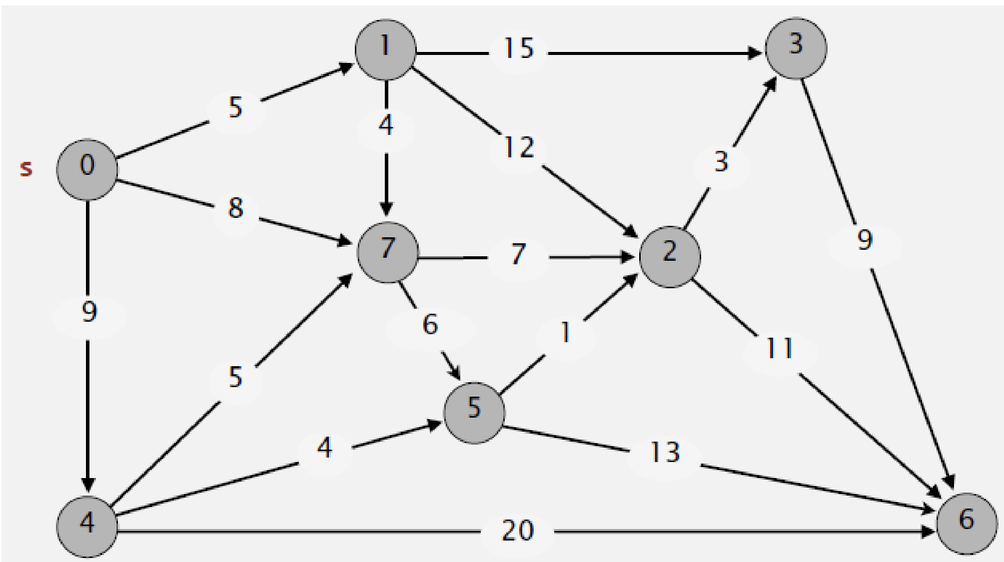
## Review problems

1.

For an interactive tutorial introducing basic concepts in graph theory, see http://www.utm.edu/departments/math/graph/ (it's highly recommended that you go through all the quiz questions in this link)

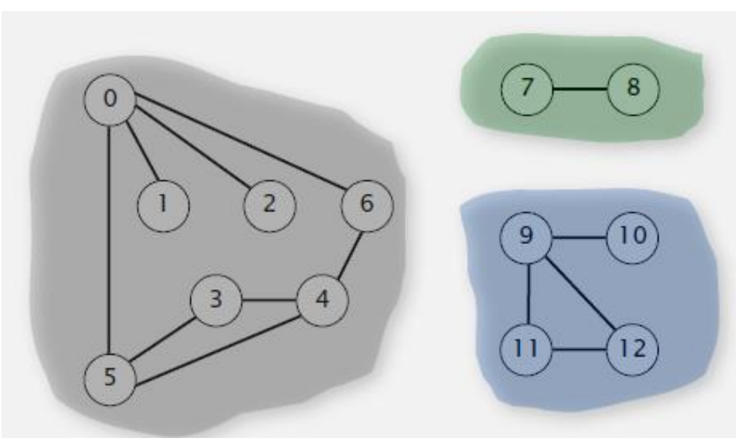
2.

Apply DFS, BFS, and dikstra algorithm manually on the following graph:




3.

Apply 'connected component' algorithm manually on the following graph:




4.

For more advanced exercises in graph algorithm, see http://algs4.cs.princeton.edu/41undirected/

## Review questions for digraph algorithms

### Summary

Its useful to compare similarity between graph algorithms and digraph algorithms

| | |
|---|---|
| Common themes | • Vertex representation: using integers<br>• Three common representation of graph:<br>• API<br>• design pattern for graph processing: decouple graph data type from graph processing<br>• depth-first search: mimic maze exploration<br>• breadth-first search<br>• connected component |
| Difference | • DAG and topologoical sort<br>• Algorithm to find strong components for digraph is more complex than its graph counterparts |

### Review problems

Similar to those for undirected graphs

## Review questions for MST algorithms

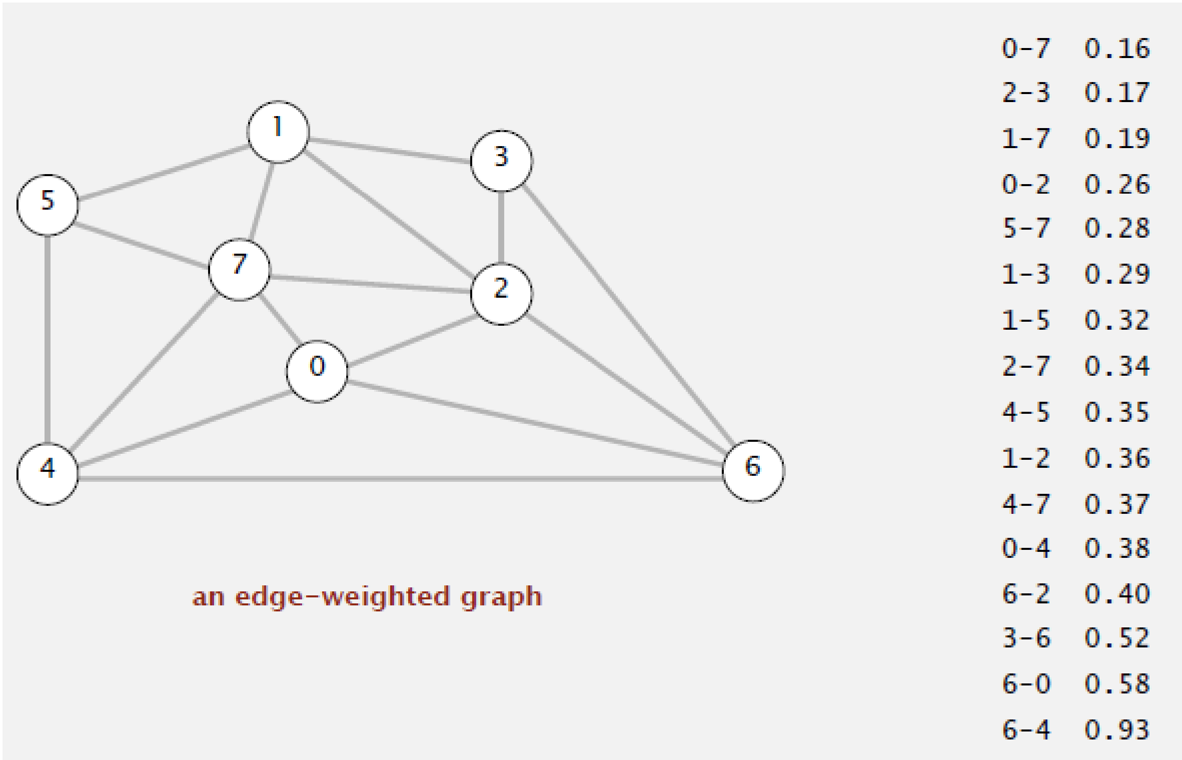
### Summary

- Definition of spanning tree and MST
- Cut property
- Greedy algorithm
- Kruskal algorithm
- Prim algorithm
  - Lazy approch
  - Eager approach

### Review problems

1.

Apply Kruskal and Prim algorithms manually on the following graph:




# Review questions for shortest path algorithms

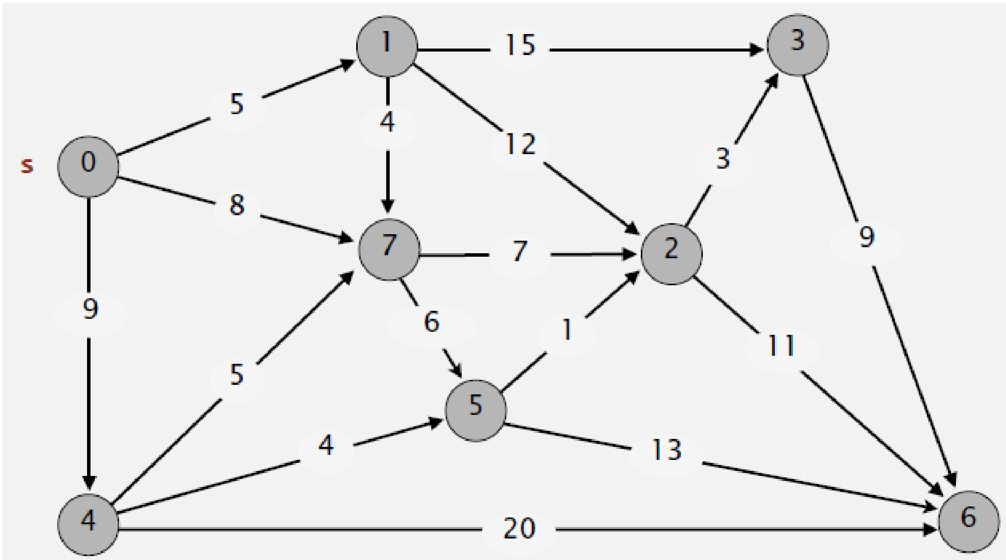
## Summary

- Any segment of a shortest path is also a shortest path
- Single source shortest paths can be stored using an edgeTo array
- Shortest distance for the single source shortest paths can be stored using a distTo array
- Dijkstra algorithm (no negative weight)
- Shortest paths in edge weighted DAGs
- Belman-Ford algorithm (no negative cycles)

## Review problems

1.

Apply Dijkstra algorithm manually on the following graph:


s
0
1
2
3
4
5
6
7
5
8
9
15
4
12
7
3
9
6
1
11
5
4
13
20