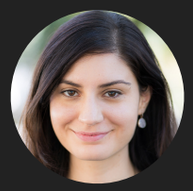
GRAPHS

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

22: Graphs



Alexandra Papoutsaki
she/her/hers

Our lecture today is opening a new (and final) chapter in this class: graphs. Graphs are widely used data structures with diverse applications.

Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226

We'll start by separating graphs into undirected and directed ones. We will see plenty of examples of how graphs can model real-world problems and then we'll look at three fundamental algorithms for processing undirected graphs and consider some of the challenges of developing algorithms for graphs.

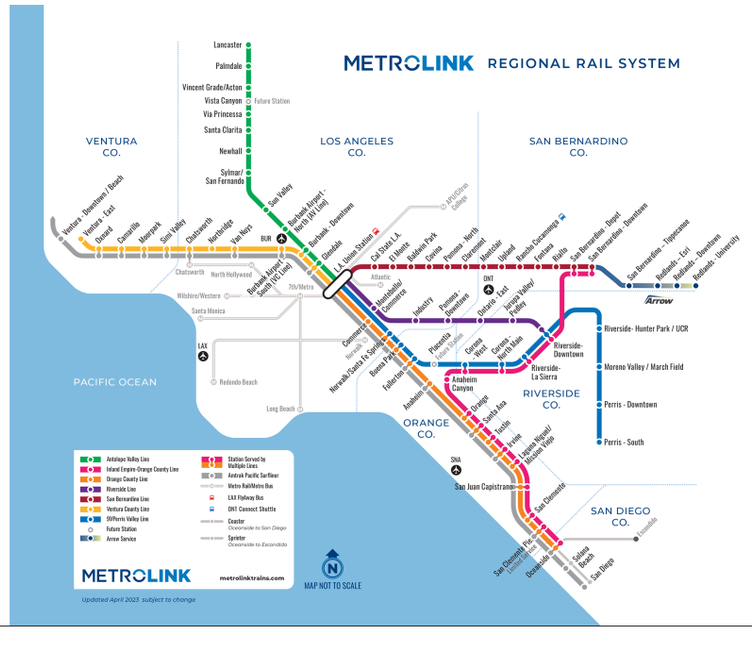
Why study graphs?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of theoretical computer science.

Why do we even study graphs? Well, there are literally thousands of practical applications where graphs are an appropriate model and we'll take a look at a few others in just a minute. Because there are so many applications, there's literally hundreds of graph algorithms known and these are sometimes quite sophisticated, as we'll see. Even without the applications, a graph is really an interesting and broadly useful abstraction to try and understand: it's so simple to describe, but it leads to quite complex and difficult to understand structures. In general, graph theory and graph algorithms are a very challenging branch of theoretical computer science.

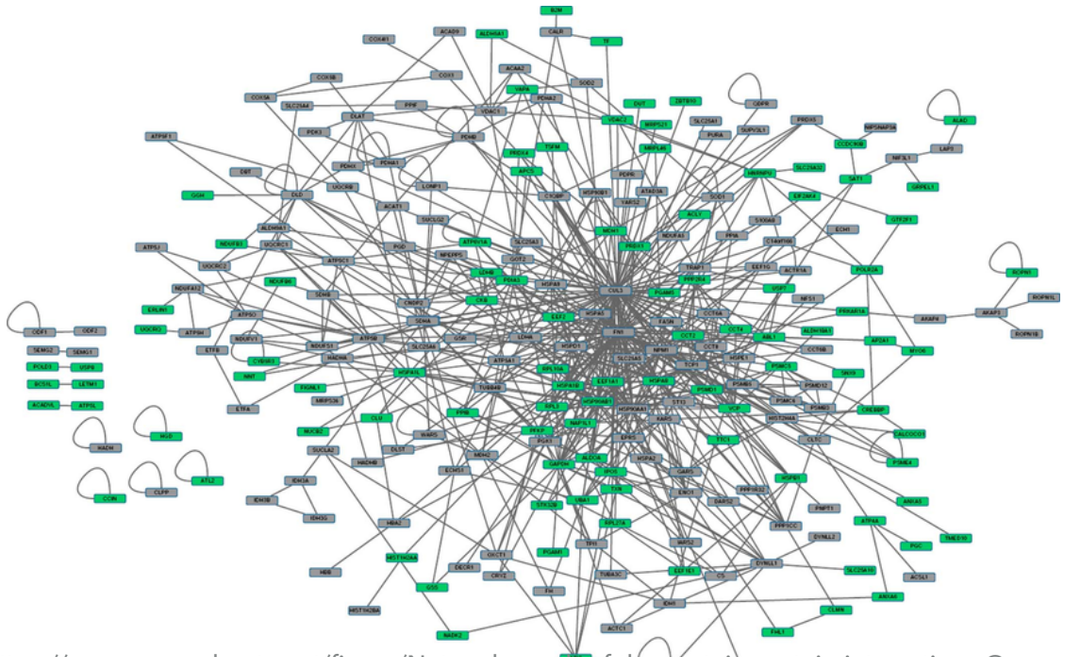
Undirected Graphs

- Graph: A set of *vertices* connected pairwise by *edges*.



What is a graph? A graph connects data in a non-hierarchical way. Data are stored in vertices which are connected pairwise by edges. We are used to seeing graphs, even if without realizing it. Here's a real-world example of a graph: The metro link system of Southern California. The vertices are the stations and if there's a rail between two stations there is an edge.

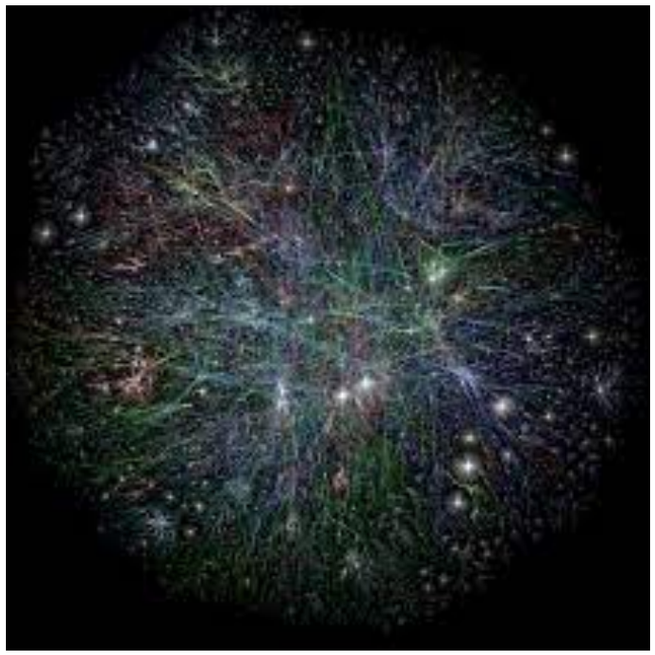
Protein-protein interaction graph



https://www.researchgate.net/figure/Network-graph-of-the-protein-protein-interactions-Green-color-represents-proteins_fig4_272297002

In genomics, we can represent proteins as vertices and the edges are interactions among the proteins. Biologists are trying to understand how these biological processes work by looking at interactions and connections.

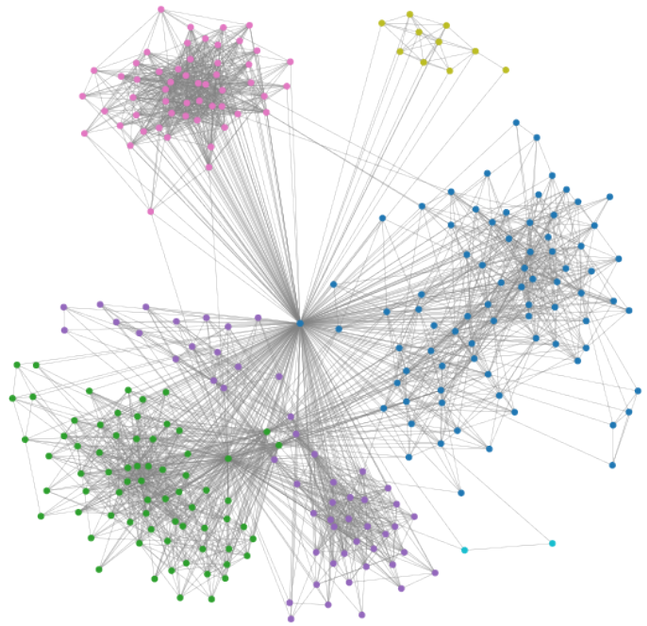
The Internet



https://www.opte.org/the-internet

A graph that we see a lot in Computer Science is the Internet, the infrastructure on which the entire Web is built on to connect computers and communication devices.

Social media

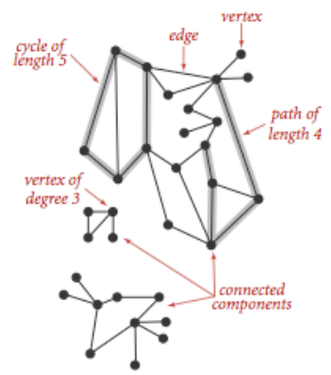


https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Social media can be modeled as graphs: all social media users are vertices and edges between them represent concepts like “friendship”, “following”, etc

Graph terminology

- Path: Sequence of vertices connected by edges
- Cycle: Path whose first and last vertices are the same
- Two vertices are connected if there is a path between them



Anatomy of a graph

Let's establish some graph terminology. A path is a sequence of vertices connected by edges. A cycle is a path that starts and ends in the same vertex. Two vertices are considered to be connected if there is a path between them.

Examples of graph-processing problems

- Is there a path between vertex s and t?
- What is the shortest path between s and t?
- Is there a cycle in the graph?
- Euler Tour: Is there a cycle that uses each edge exactly once?
- Hamilton Tour: Is there a cycle that uses each vertex exactly once?
- Is there a way to connect all vertices?
- What is the shortest way to connect all vertices?
- Is there a vertex whose removal disconnects the graph?

Here are some classic examples of graph-processing problems:

- Is there a path between vertex s and t?
- What is the shortest path between s and t?
- Is there a cycle in the graph?
- Euler Tour: Is there a cycle that uses each edge exactly once?
- Hamilton Tour: Is there a cycle that uses each vertex exactly once?
- Is there a way to connect all vertices?
- What is the shortest way to connect all vertices?
- Is there a vertex whose removal disconnects the graph?

Lecture 22: Graphs

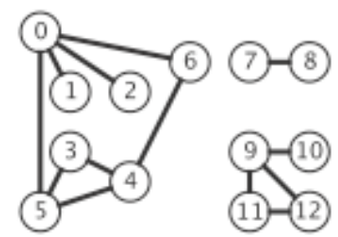
- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

Let's look into the API for undirected graphs.

Graph representation

- Vertex representation: Here, integers between 0 and V-1.
  - We will use a dictionary to map between names of vertices and integers (indices).

```
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
```



We will make the assumption that the vertex names have been mapped (through dictionary) to indices running from 0 to V-1, with |V| being the cardinality of the set, i.e. the number of vertices. Beyond vertices, we need to also store information about the edges by indicating all the pairs that are connected. E.g., here you can see a graph on the right and on the left the pairs of vertices that have an edge between them.

Basic Graph API

- `public class Graph`
  - `Graph(int V)`: create an empty graph with V vertices.
  - `void addEdge(int v, int w)`: add an edge v-w.
  - `Iterable<Integer> adj(int v)`: return vertices adjacent to v.
  - `int V()`: number of vertices.
  - `int E()`: number of edges.

In terms of creating an API, we can imagine a class Graph that has a constructor that creates an empty graph with 0...V-1 vertices. The API would support an addEdge message that takes two vertices, v and w, that are connected with an edge, v-w. The class can also have an adj method that given an index of a vertex, would return all the vertices that are adjacent (i.e. directly connected via an edge) to v. And two getters V and E that would return the number of vertices and edges.

Example of how to use the Graph API to process the graph

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

Using these basic blocks, we can start processing the graph. E.g., we could come up with a method degree that given a graph and a vertex, would compute the degree of v: the number of vertices that are connected to v.

Graph density

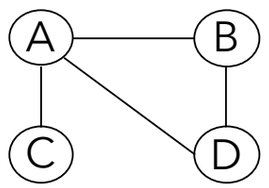
- In a simple graph (no parallel edges or loops), if $|V| = n$, then:
  - minimum number of edges is 0 and
  - maximum number of edges is $n(n-1)/2$.
- Dense graph -> edges closer to maximum.
- Sparse graph -> edges closer to minimum.

When it comes to graphs, you can imagine that a graph with a fixed number of vertices, |V| = n, can have no, few, or many edges. The minimum number of edges would be 0: all vertices would be disjointed. The maximum number would be n(n-1)/2~O(n^2) where all vertices are connected with all vertices. In general, we consider a graph to be edge when the number of edges is close to the maximum and sparse only when a few edges exist.

Graph representation: adjacency matrix

- Maintain a $|V|$-by-$|V|$ boolean array; for each edge `v-w`:
  - `adj[v][w] = adj[w][v] = true;`
- Good for dense graphs (edges close to $|V|^2$).
- Constant time for lookup of an edge.
- Constant time for adding an edge.
- $|V|$ time for iterating over vertices adjacent to v.
- Symmetric, therefore wastes space in undirected graphs ($|V|^2$).
- Not widely used in practice.

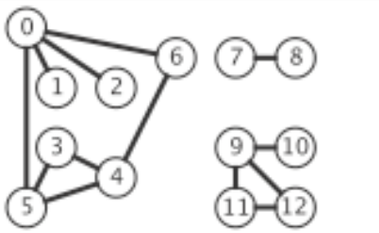
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

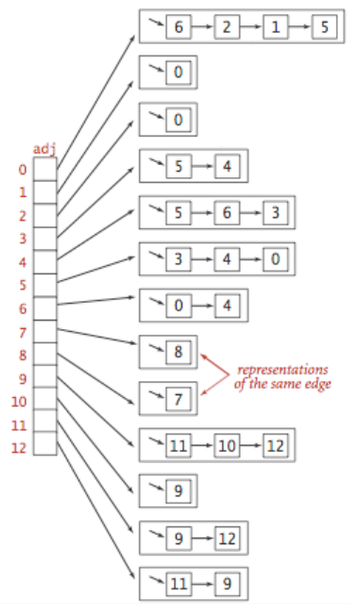


There are two main ways that we can represent a graph. The first one is through an adjacency matrix. An adjacency matrix is a |V|x|V| boolean array where an edge v-w would correspond to the adj[v][w] adj[w][v] be marked as true. You can see a graph on the bottom right, and its corresponding adjacency matrix above it.
Adjacency matrices are good for dense graphs, that is graphs whose number of edges is close to |V|^2.
In terms of performance, it would take constant time to lookup and add an edge. If it comes to returning all vertices adjacent to a specific vertex, it would take |V| time since we have to go through the entire row.
One note: the matrix is symmetric (A-B is the same with B-A) in undirected graphs, therefore it is not widely used in practice.

Graph representation: adjacency list

- Maintain vertex-indexed array of lists.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Algorithms based on iterating over vertices adjacent to v.
- Space efficient ($|E| + |V|$).
- Constant time for adding an edge.
- Lookup of an edge or iterating over vertices adjacent to v is $degree(v)$.





An alternative, and more popular, representation of graphs is the adjacency list. An adjacency list maintains a vertex-index array of lists which makes it great for sparse graphs (edges proportional to |V| instead of |V|^2) which is more likely to be encountered in the real world. To search for edges, we would need to go to the specified index and iterate over its indices. Therefore, the lookup of an edge or iterating over adjacent vertices is determined by the degree of the provided vertex. Adding an edge can be done in constant tie since we can just add the adjacent vertex to the head of the corresponding list. And finally, the space efficiency depends on the number of vertices PLUS the number of edges.

Adjacency-list graph representation in Java

```
public class Graph {

    private final int V;
    private int E;
    private ArrayList<ArrayList<Integer>> adj;

    //Initializes an empty graph with V vertices and 0 edges.
    public Graph(int V) {
        this.V = V;
        this.E = 0;
        adj = new ArrayList<ArrayList<Integer>>(V);
        for (int v = 0; v < V; v++) {
            adj.add(new ArrayList<Integer>());
        }
    }

    //Adds the undirected edge v-w to this graph. Parallel edges and self-loops allowed
    public void addEdge(int v, int w) {
        E++;
        adj.get(v).add(w);
        adj.get(w).add(v);
    }


    //Returns the vertices adjacent to vertex v.
    public Iterable<Integer> adj(int v) {
       return adj.get(v);
    }
```

Here's how we could code a Graph class given the adjacency list representation. You will notice that we maintain a list of lists. When adding an edge, we just increase the number of edges, and add the vertex on the corresponding links for both vertices.

Lecture 22: Graphs

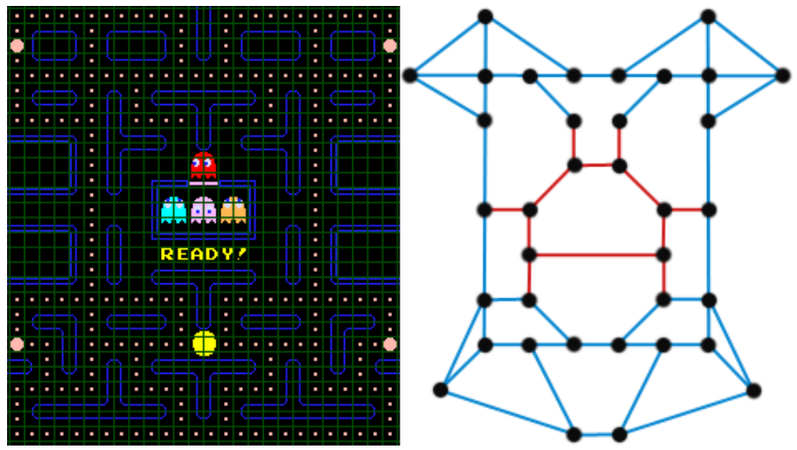
- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226

Let's now start looking into some interesting things we can do with undirected graphs.

Mazes as graphs

- Vertex = intersection; edge = passage

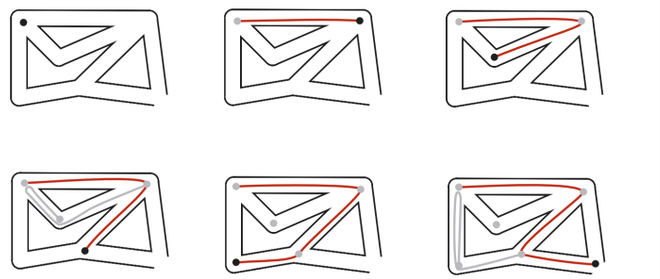


http://oatzy.blogspot.com/2011/09/playing-with-pac-man.html

We will start with depth-first search, which is a classical graph processing algorithm, actually one of the oldest algorithms that we study. As a warm-up, one way to think about depth first search, is in terms of mazes. Let’s assume you have a maze like the one drawn on the left; then, you can model it with a graph as in the right. We create a vertex for every intersection in the maze and an edge for every passage connecting two intersections. If you're at the entrance to this maze and you want to find a pot of gold somewhere what you're gonna need to do is either explore every intersection or explore every edge. We're gonna talk about the explore every intersection option. In case you are wondering about the colors, the blue edges are the ones with pellets. The red edges are the ones without pellets, which you don't necessarily have to visit.

How to survive a maze: a lesson from a Greek myth

- Theseus escaped from the labyrinth after killing the Minotaur with the following strategy instructed by Ariadne:
  - Unroll a ball of string behind you.
  - Mark each newly discovered intersection and passage.
  - Retrace steps when no unmarked options.
- Also known as the Trémaux algorithm.



How we survive a maze has fascinated humans for thousands of years. In Greek mythology, the hero Theseus escaped from the labyrinth of King Minoas after killing the Minotaur with the following strategy instructed by Ariadne:
Unroll a ball of string behind you.
Mark each newly discovered intersection and passage.
Retrace steps when no unmarked options.
This algorithm was officially coined at the end of the 19th century by Tremaux.

Depth-first search

- Goal: Systematically traverse a graph.
- DFS (to visit a vertex v)
  - Mark vertex v.
  - Recursively visit all unmarked vertices w adjacent to v.
- Typical applications:
  - Find all vertices connected to a given vertex.
  - Find a path between two vertices.

Which brings us to depth-first search or DFS. Our goal is to systematically traverse a graph. DFS starts at a vertex, marks that vertex as visited, and recursively visits all unmarked vertices adjacent to it. This simple algorithm has many applications which its most typical ones being finding all vertices connected to a given vertex and finding a path between two vertices.

Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

4.1 DEPTH-FIRST SEARCH DEMO

Let's look into a visualization of DFS. We will need a vertex index array to keep track of which vertices are visited. So that will just be an array of booleans and we'll initialize that with all false. We're also gonna keep another data structure, a vertex indexed array of ints. That for every vertex gives us the vertex that took us there. S

Depth-first search

- Goal: Find all vertices connected to s (and a corresponding path).
- Idea: Mimic maze exploration.
- Algorithm:
  - Use recursion (ball of string).
  - Mark each visited vertex (and keep track of edge taken to visit it).
  - Return (retrace steps) when no unvisited options.

- When started at vertex s, DFS marks all vertices connected to s (and no other).

If we restate our goal as finding all vertices connected to s (and a corresponding path), we can mimic the maze exploration by using the following algorithm: Use recursion (ball of string).
Mark each visited vertex (and keep track of edge taken to visit it).
Return (retrace steps) when no unvisited options.

When started at vertex s, DFS marks all vertices connected to s (and no other).

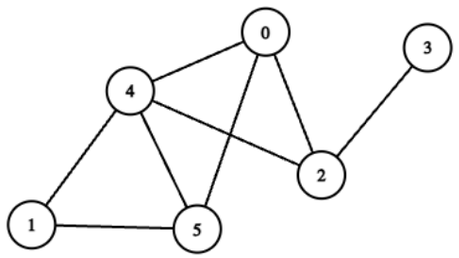
Implementation of depth-first search in Java

```
public class DepthFirstSearch {
     private boolean[] marked;     // marked[v] = is there an s-v path?
     private int[] edgeTo;         // edgeTo[v] = previous vertex on path from s to v

    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        dfs(G, s);
    }

    // depth first search from v
    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }
```

This is how we can implement DFS. As we already saw, we will need a vertex index array to keep track of which vertices are visited. So that will just be an array of booleans and we'll initialize that with all false. We're also gonna keep another data structure, a vertex indexed array of ints. That for every vertex gives us the vertex that took us there.

PRACTICE TIME

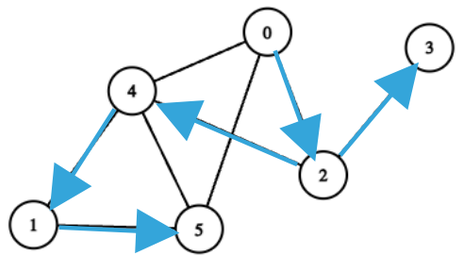
- Run DFS on the following graph starting at vertex 0 and return the vertices in the order of being marked. Assume that the adj method returns back the adjacent vertices in increasing numerical order.



Let's practice by running the recursive dfs we just saw on the following graph starting at vertex 0 and assuming that we get back adjacent vertices in increasing numerical order.

ANSWER

- Vertices marked as visited: 0, 2, 3, 4, 1, 5



V	marked	edgeTo
0	T	-
1	T	4
2	T	0
3	T	2
4	T	2
5	T	1

Depth-first search analysis

- DFS marks all vertices connected to s in time proportional to $|V| + |E|$ in the worst case.
  - Initializing arrays `marked` and `edgeTo` takes time proportional to $|V|$.
  - Each adjacency-list entry is examined exactly once and there are $2|E|$ such entries (two for each edge).
- Once we run DFS, we can check if vertex v is connected to s in constant time. We can also find the v-s path (if it exists) in time proportional to its length.

DFS marks all vertices connected to s in time proportional to |V|+|E| in the worst case. Initializing arrays marked and edgeTo takes time proportional to .
Each adjacency-list entry is examined exactly once and there are such entries (two for each edge).
Once we run DFS, we can check if vertex v is connected to s in constant time. We can also find the v-s path (if it exists) in time proportional to its length.

Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

DFS is not the only search algorithm to explore a graph. Breadth-first search is also a popular option

Breadth-first search

- BFS (from source vertex s)
  - Put s on a queue and mark it as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex v.
    - Enqueue each of v's unmarked neighbors and mark them.
- Basic idea: BFS traverses vertices in order of distance from s.

Depth-first search finds some path from a source vertex s to a target vertex v. We are often interested in finding the shortest such path (one with a minimal number of edges). Breadth-first search is a classic method based on this goal. To find a shortest path from s to v, we start at s and check for v among all the vertices that we can reach by following one edge, then we check for v among all the vertices that we can reach from s by following two edges, and so forth. To implement this strategy, we maintain a queue of all vertices that have been marked but whose adjacency lists have not been checked. We put the source vertex on the queue, then perform the following steps until the queue is empty:
Remove the next vertex v from the queue.
Put onto the queue all unmarked vertices that are adjacent to v and mark them.
Basic idea: BFS traverses vertices in order of distance from s.

Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

4.1 BREADTH-FIRST SEARCH DEMO

Here's a visualization of BFS.

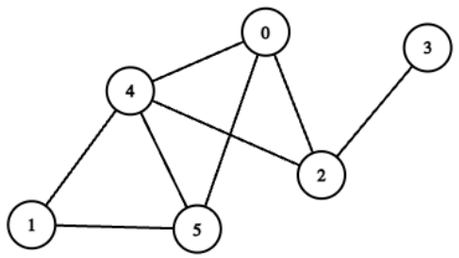
Breadth-first search in Java

```
public class BreadthFirstSearch {
   private boolean[] marked;  // marked[v] = is there an s-v path
    private int[] edgeTo;      // edgeTo[v] = previous edge on shortest s-v path
    private int[] distTo;      // distTo[v] = number of edges shortest s-v path

    public BreadthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        bfs(G, s);
   }

   private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        distTo[s] = 0;
        marked[s] = true;
        q.enqueue(s);

        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
        }
    }
```

And here's its implementation (very similar to iterative DFS!)

PRACTICE TIME

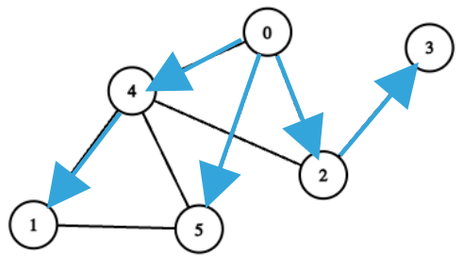
- Run the BFS on the following graph starting at vertex 0 and return the vertices in the order of being marked. Assume that the adj method returns back the adjacent vertices in increasing numerical order.



Let's practice with BFS on our usual graph.

ANSWER

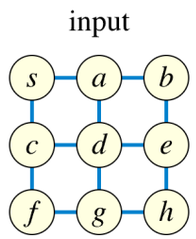
- Vertices marked as visited: 0, 2, 4, 5, 3, 1



V	marked	edgeTo	distTo
0	T	-	0
1	T	4	2
2	T	0	1
3	T	2	2
4	T	0	1
5	T	0	1

PRACTICE TIME

- Run DFS and BFS on the following graph starting at vertex s. Assume that the adj method returns back the adjacent vertices in lexicographic order.

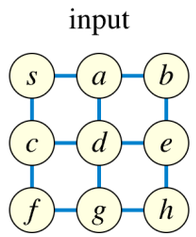


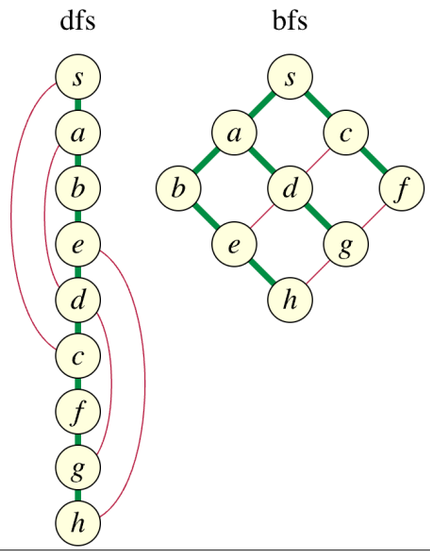
https://11011110.github.io/blog/2013/12/17/stack-based-graph-traversal.html

Let's run DFS and BFS on the graph above.

ANSWER

- Run DFS and BFS on the following graph starting at vertex s. Assume that the adj method returns back the adjacent vertices in lexicographic order.
- DFS: s->a->b->e->d->c->f->g->h
- BFS: s->a->c->b->d->f->e->g->h





Let's run DFS and BFS on the graph above.

Summary

- DFS: Uses recursion.
- BFS: Put unvisited vertices on a queue.
- Shortest path problem: Find path from s to t that uses the fewest number of edges.
  - E.g., calculate the fewest numbers of hops in a communication network.
  - E.g., calculate the Kevin Bacon number or Erdös number.
- BFS computes shortest paths from s to all vertices in a graph in time proportional to $|E| + |V|$
  - The queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of k+1.

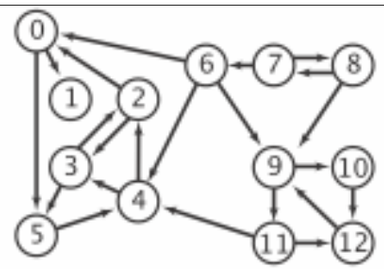
To summarize, DFS uses recursion to go as deep as possibly while BFS put them on a queue.
The shortest path problem states that we need to find a path from s to t that uses the fewest number of edges.
E.g., calculate the fewest numbers of hops in a communication network.
E.g., calculate the Kevin Bacon number or Erdös number.
BFS computes shortest paths from s to all vertices in a graph in time proportional to
The queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of k+1.

Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

And that's all for undirected graphs. Let's look into directed graphs next.

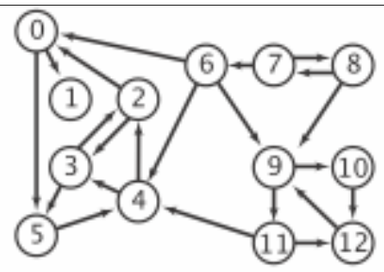
Directed Graph Terminology



- Directed Graph (digraph) : a set of vertices V connected pairwise by a set of directed edges E.
  - E.g., V = {0,1,2,3,4,5,6,7,8,9,10,11,12},
    E = {{0,1}, {0,5}, {2,0}, {2,3},{3,2},{3,5},{4,2},{4,3},{5,4},{6,0},{6,4},{6,9},{7,6}{7,8},{8,7},{8,9},
    {9,10},{9,11},{10,12},{11,4},{11,12},{12,9}}.
- Directed path: a sequence of vertices in which there is a directed edge pointing from each vertex in the sequence to its successor in the sequence, with no repeated edges.
  - A simple directed path is a directed path with no repeated vertices.
- Directed cycle: Directed path with at least one edge whose first and last vertices are the same.
  - A simple directed cycle is a directed cycle with no repeated vertices (other than the first and last).
- The length of a cycle or a path is its number of edges.

Directed graphs or digraphs are a set of vertices V connected pairwise by a set of directed edges E, like in the picture above. Let's establish some terminology about directed graphs.
A directed path 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, with no repeated edges.
A simple directed path is a directed path with no repeated vertices.
A directed cycle is a directed path with at least one edge whose first and last vertices are the same.
A simple directed cycle is a directed cycle with no repeated vertices (other than the first and last).
The length of a cycle or a path is its number of edges.

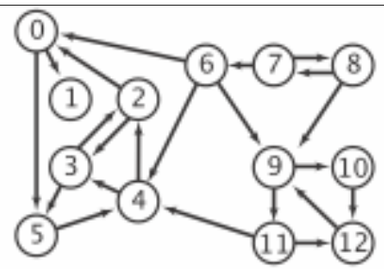
Directed Graph Terminology



- Self-loop: an edge that connects a vertex to itself.
- Two edges are parallel if they connect the same pair of vertices.
- The outdegree of a vertex is the number of edges pointing from it.
- The indegree of a vertex is the number of edges pointing to it.
- A vertex w is reachable from a vertex v if there is a directed path from v to w.
- Two vertices v and w are strongly connected if they are mutually reachable.

A self-loop is an edge that connects a vertex to itself. We say that two edges are parallel if they connect the same pair of vertices.
The outdegree of a vertex is the number of edges pointing from it.
The indegree of a vertex is the number of edges pointing to it.
A vertex w is reachable from a vertex v if there is a directed path from v to w.
Two vertices v and w are strongly connected if they are mutually reachable.

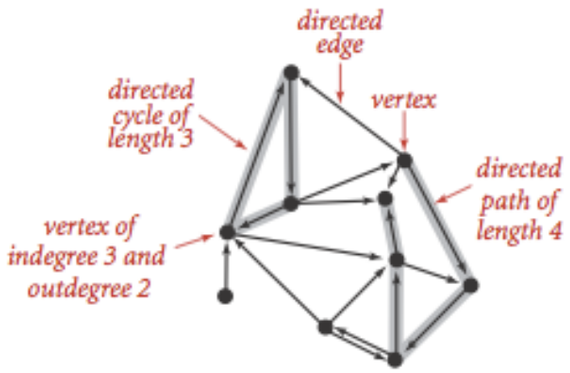
Directed Graph Terminology



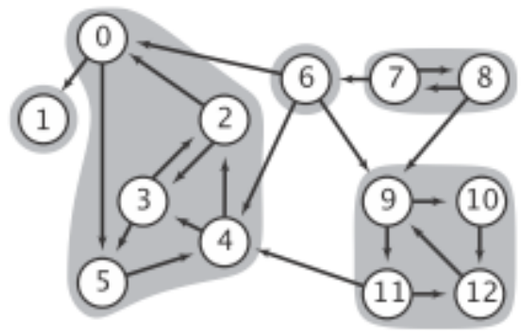
- A digraph is strongly connected if there is a directed path from every vertex to every other vertex.
- A digraph that is not strongly connected consists of a set of strongly connected components, which are maximal strongly connected subgraphs.
- A directed acyclic graph (DAG) is a digraph with no directed cycles.

A digraph is strongly connected if there is a directed path from every vertex to every other vertex.
A digraph that is not strongly connected consists of a set of strongly connected components, which are maximal strongly connected subgraphs.
A directed acyclic graph (DAG) is a digraph with no directed cycles.

Anatomy of a digraph



Anatomy of a digraph



A digraph and its strong components

Here are two figures that show all those definitions we just saw.

Digraph Applications

Digraph	Vertex	Edge
Web	Web page	Link
Cell phone	Person	Placed call
Financial	Bank	Transaction
Transportation	Intersection	One-way street
Game	Board	Legal move
Citation	Article	Citation
Infectious Diseases	Person	Infection
Food web	Species	Predator-prey relationship

Digraphs have a ton of applications, with a great way of modeling networks such as the Web, phone calls, financial transactions, etc.

Popular digraph problems

Problem	Description
s->t path	Is there a path from s to t?
Shortest s->t path	What is the shortest path from s to t?
Directed cycle	Is there a directed cycle in the digraph?
Topological sort	Can vertices be sorted so all edges point from earlier to later vertices?
Strong connectivity	Is there a directed path between every pair of vertices?

Their popularity has led to a few standardized problems, such as is there a path from s to t, what is the shortest such path, is there a directed cycle, can vertices be sorted so all edges point from earlier to later vertices, and is there a directed path between every pair of vertices?

Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

How do we go about implementing a digraph in Java?

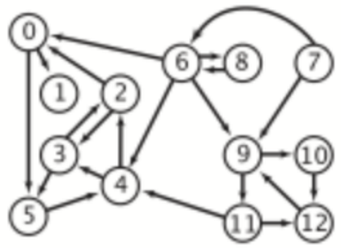
Basic Graph API

- `public class Digraph`
  - `Digraph(int V)`: create an empty digraph with V vertices.
  - `void addEdge(int v, int w)`: add an edge v->w.
  - `Iterable<Integer> adj(int v)`: return vertices adjacent from v.
  - `int V()`: number of vertices.
  - `int E()`: number of edges.
  - `Digraph reverse()`: reverse edges of digraph.

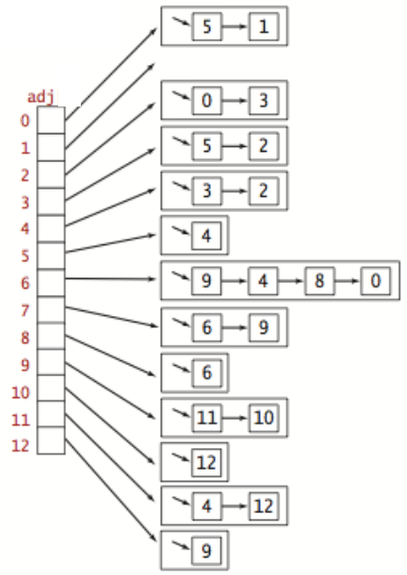
Here's a basic API for it.

DIRECTED GRAPHS

Digraph representation: adjacency list



- Maintain vertex-indexed array of lists.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Algorithms based on iterating over vertices adjacent from v.
- Space efficient ($|E| + |V|$).
- Constant time for adding a directed edge.
- Lookup of a directed edge or iterating over vertices adjacent from v is $outdegree(v)$.



And here's how the adjacency list could look like. The difference here is that Lookup of a directed edge or iterating over vertices adjacent from a vertex depends on its outdegree.

Adjacency-list digraph representation in Java

```
public class Digraph {

    private final int V;
    private int E;
    private ArrayList<ArrayList<Integer>> adj;

    //Initializes an empty digraph with V vertices and 0 edges.
    public Digraph(int V) {
        this.V = V;
        this.E = 0;
        adj = new ArrayList<ArrayList<Integer>>(V);
        for (int v = 0; v < V; v++) {
            adj.add(new ArrayList<Integer>());
        }
    }

    //Adds the directed edge v->w to this digraph.
    public void addEdge(int v, int w) {
        E++;
        adj.get(v).add(w);
   }


    //Returns the vertices adjacent from vertex v.
    public Iterable<Integer> adj(int v) {
       return adj.get(v);
    }
```

And here's the implementation with the only difference from undirected graphs that we add only once the edge from v to w, and not from w to v.

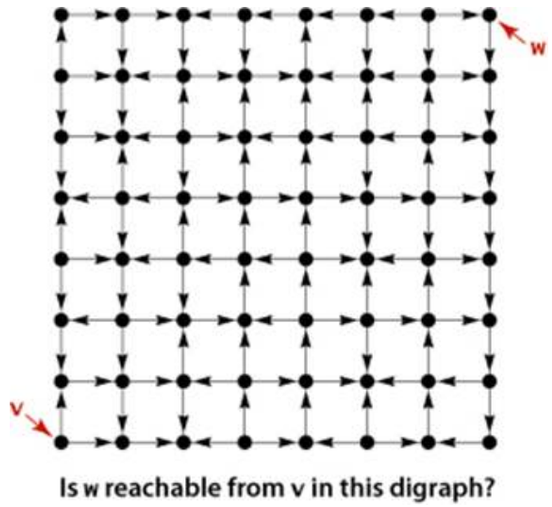
Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

Let's see how DFS would play out in digraphs.

Reachability

- Find all vertices reachable from s along a directed path.



Is w reachable from v in this digraph?

https://apprize.info/science/algorithms_2/2.html

It will help us answer the reachability problem which asks us to find all vertices reachable from s along a directed path. Or whether a specific vertex is reachable from another vertex in a digraph.

Depth-first search in digraphs

- Same method as for undirected graphs.
  - Every undirected graph is a digraph with edges in both directions.
  - Maximum number of edges in a simple digraph is $n(n-1)$.
- DFS (to visit a vertex V)
  - Mark vertex V.
  - Recursively visit all unmarked vertices W adjacent from V.
- Typical applications:
  - Find a directed path from source vertex S to a given target vertex V.
  - Topological sort.
  - Directed cycle detection.

DFS is essentially the same for digraphs as with undirected graphs. Note that the maximum number of edges in a simple digraph has twice the edges of an undirected graph. Typical applications of DFS in digraphs are finding a directed path from a source vertex to a target vertex, topological sort, and detecting cycles.

Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

4.2 DIRECTED DFS DEMO

Here's a demo of DFS.

Directed depth-first search in Java

```
public class DirectedDFS {
   private boolean[] marked;    // marked[v] = is there an s->v path?

   public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    // directed depth first search from v
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }
```

And as you can see the code doesn't change much, neither for the recursive...

Depth-first search analysis

- DFS marks all vertices reachable from s in time proportional to $|V| + |E|$ in the worst case.
  - Initializing arrays `marked` takes time proportional to $|V|$.
  - Each adjacency-list entry is examined exactly once and there are E such edges.
- Once we run DFS, we can check if vertex v is reachable from s in constant time. We can also find the s->v path (if it exists) in time proportional to its length.

In general, again DFS's performance depends on |V|+|E|. Once we run DFS, we can check if vertex v is reachable from s in constant time. We can also find the s->v path (if it exists) in time proportional to its length.

Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

What about BFS?

Breadth-first search

- Same method as for undirected graphs.
  - Every undirected graph is a digraph with edges in both directions.
- BFS (from source vertex s)
  - Put s on queue and mark s as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex v.
    - Enqueue all unmarked vertices adjacent from v, and mark them.
- Typical applications:
  - Find the shortest (in terms of number of edges) directed path between two vertices in time proportional to $|E| + |V|$.

Not much is different here either! Typical applications is finding the shortest directed path between two vertices.

Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

4.2 DIRECTED BFS DEMO

Here's a BFS demo.

Summary

- Single-source reachability in a digraph: DFS/BFS.
- Shortest path in a digraph: BFS.

In general, if you want to answer the single-source reachability problem in digraphs you can use either DFS or BFS. For the shortest path, you can use BFS.

Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

And as a final topic, let's see how we can find the strongly connected components of a digraph.

Is a digraph strongly connected?

- A strongly connected digraph is a directed graph in which it is possible to reach any vertex starting from any other vertex by traversing edges.
- Pick a random starting vertex s.
- Run DFS/BFS starting at s.
  - If have not reached all vertices, return false.
- Reverse edges.
- Run DFS/BFS again on reversed graph.
  - If have not reached all vertices, return false.
  - Else return true.

A strongly connected digraph is a directed graph in which it is possible to reach any vertex starting from any other vertex by traversing edges. To find whether a digraph is strongly connected, we can pick a random starting vertex and run either BFS or DFA on it. If at the end, we have not reached all vertices we return false. Next, we reverse all edges of the digraph and run DFS or BFS starting on the same vertex. If we gave not reached all vertices, we return false. Otherwise, we return true.

Lecture 22: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components

Readings:

- Recommended Textbook: Chapter 4.1 (Pages 522-556), Chapter 4.2 (Pages 566-594)
- Website:
  - https://algs4.cs.princeton.edu/41graph/
  - https://algs4.cs.princeton.edu/42digraph/

Visualization

- https://visualgo.net/en/dfsbfs

Problem 1

- What is the maximum number of edges in an undirected graph with V vertices and no parallel edges?
- What is the minimum number of edges in an undirected graph with V vertices, none of which are isolated (have degree 0)?
- What is the maximum number of edges in a digraph with V vertices and no parallel edges?
- What is the minimum number of edges in a digraph with V vertices, none of which are isolated?

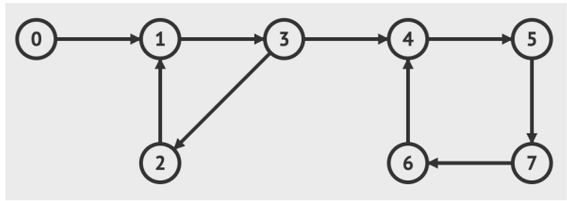
Problem 2

- Assume you are given the following 16 edges of an undirected graph with 12 vertices, inserted in an adjacency list in this order:

- 8-4
- 2-3
- 1-11
- 0-6
- 3-6
- 10-3
- 7-11
- 7-8
- ...
- 11-8
- 2-0
- 6-2
- 5-2
- 5-10
- 5-0
- 8-1
- 4-1

Problem 3

- Run DFS and BFS on the following digraph starting at vertex 0.



Answer 1

- What is the maximum number of edges in an undirected graph with V vertices and no parallel edges?
  - $n(n-1)/2$, where $n = |V|$.
- What is the minimum number of edges in an undirected graph with V vertices, none of which are isolated (have degree 0)?
  - $n-1$.
- What is the maximum number of edges in a digraph with V vertices and no parallel edges?
  - $n(n-1)$, where $n = |V|$.
- What is the minimum number of edges in a digraph with V vertices, none of which are isolated?
  - $n-1$.

Answer 2

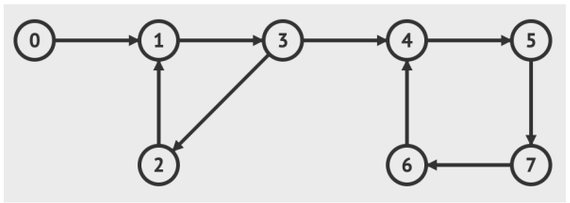
- Assume you are given the following 16 edges of an undirected graph with 12 vertices, inserted in an adjacency list in this order:

- 8-4
- 2-3
- 1-11
- 0-6
- 3-6
- 10-3
- 7-11
- 7-8
- ...
- 11-8
- 2-0
- 6-2
- 5-2
- 5-10
- 5-0
- 8-1
- 4-1

- 0 -> 5 -> 2 -> 6
- 1 -> 4 -> 8 -> 11
- 2 -> 5 -> 6 -> 0 -> 3
- 3 -> 10 -> 6 -> 2
- 4 -> 1 -> 8
- 5 -> 0 -> 10 -> 2
- 6 -> 2 -> 3 -> 0
- 7 -> 8 -> 11
- 8 -> 1 -> 11 -> 7 -> 4
- 9 ->
- 10 -> 5 -> 3
- 11 -> 8 -> 7 -> 1

Answer 3

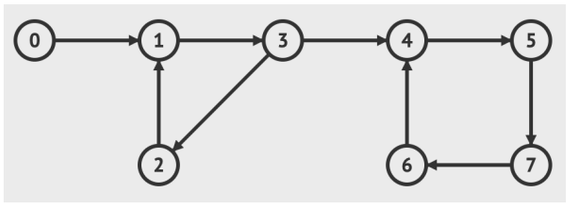
- DFS - Order of visit: 0, 1, 3, 2, 4, 5, 7, 6



V	marked	edgeTo
0	T	-
1	T	0
2	T	3
3	T	1
4	T	3
5	T	4
6	T	7
7	T	5

Answer 3

- BFS - Order of visit: 0, 1, 3, 2 4, 5, 7, 6



V	marked	edgeTo	distTo
0	T	-	0
1	T	1	1
2	T	3	2
3	T	1	2
4	T	3	3
5	T	4	4
6	T	7	6
7	T	5	5