Data Structures and Algorithms

Welcome!

Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- Searching graphs (DFS / BFS)

What is a graph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one. (In our course)
- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Each edge is unique.

What is a graph?

Graph G = <V, E>

- V is a set of nodes
  - At least one: $|V| > 0$.
- E is a set of edges:
  - $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
  - $e = (v,w)$, for $v \neq w$
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

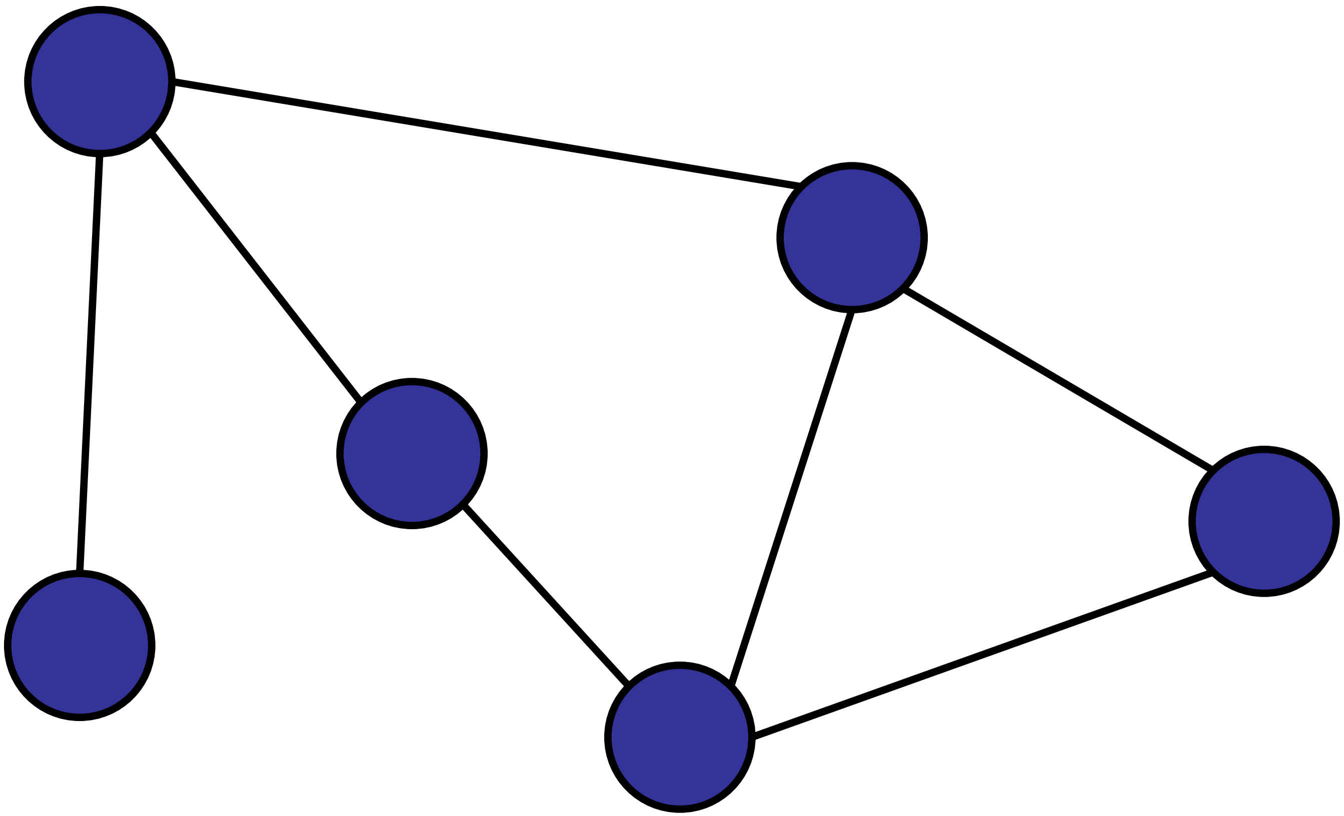
Do not allow self-loops

Only one edge for each pair of nodes

Graph Terminology

Connected:

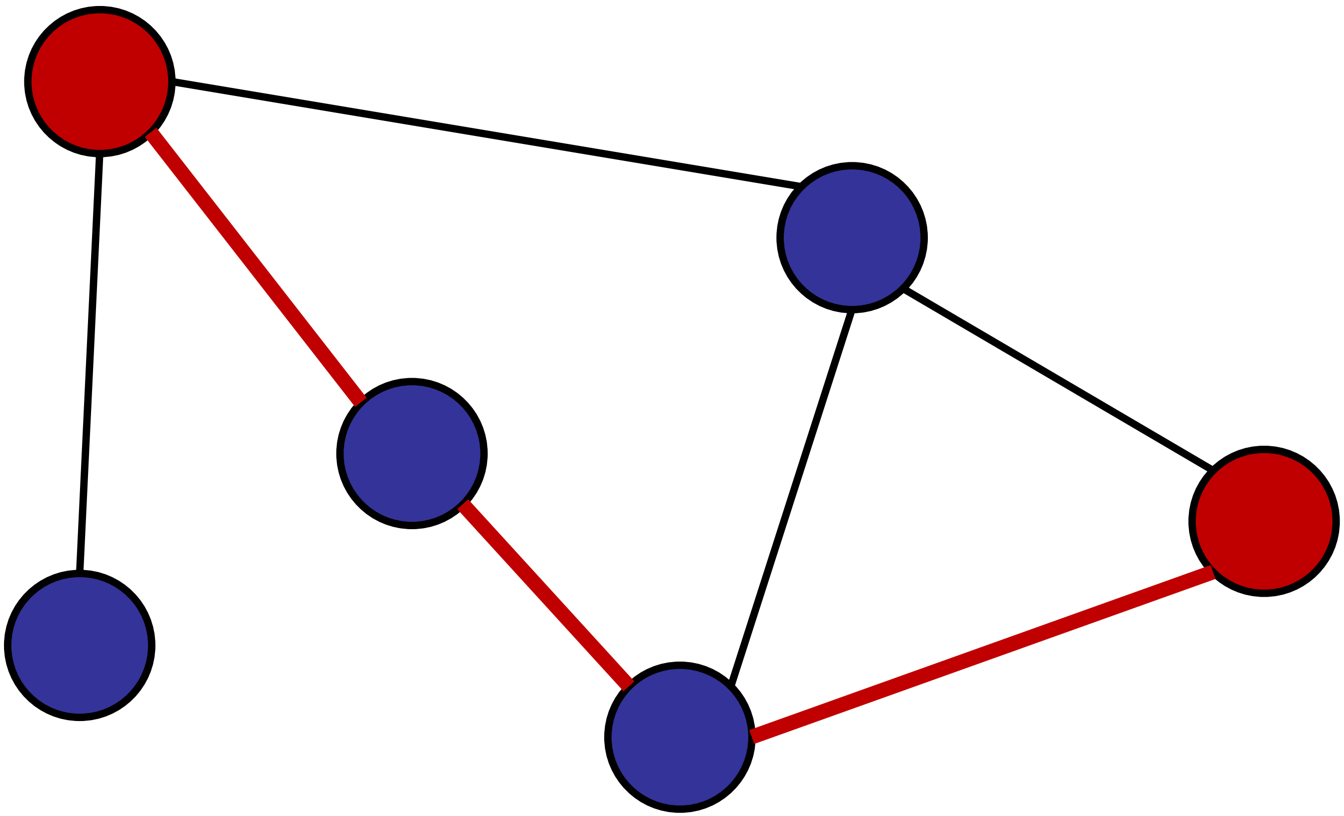
– Every pair of nodes is connected by a path.



Graph Terminology

Connected:

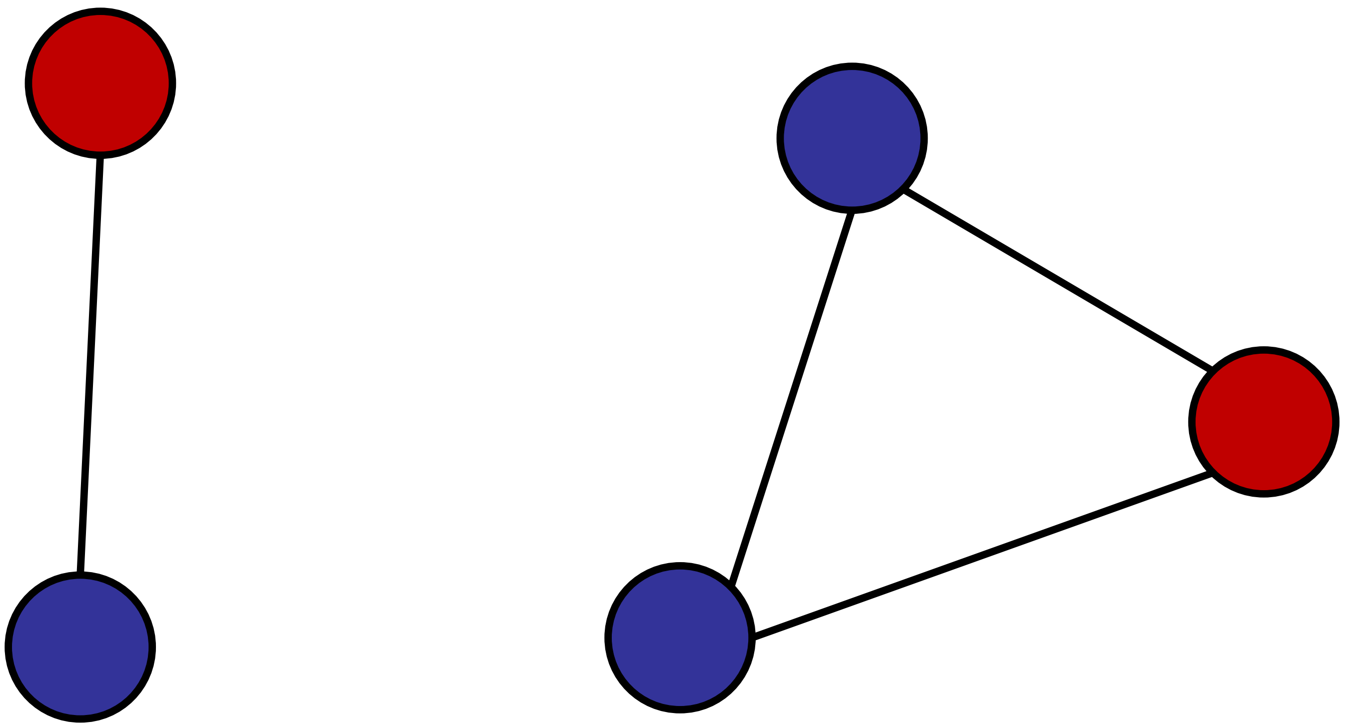
– Every pair of nodes is connected by a **path**.



Graph Terminology

Disconnected:

- Some pair of nodes is **not** connected by a path.

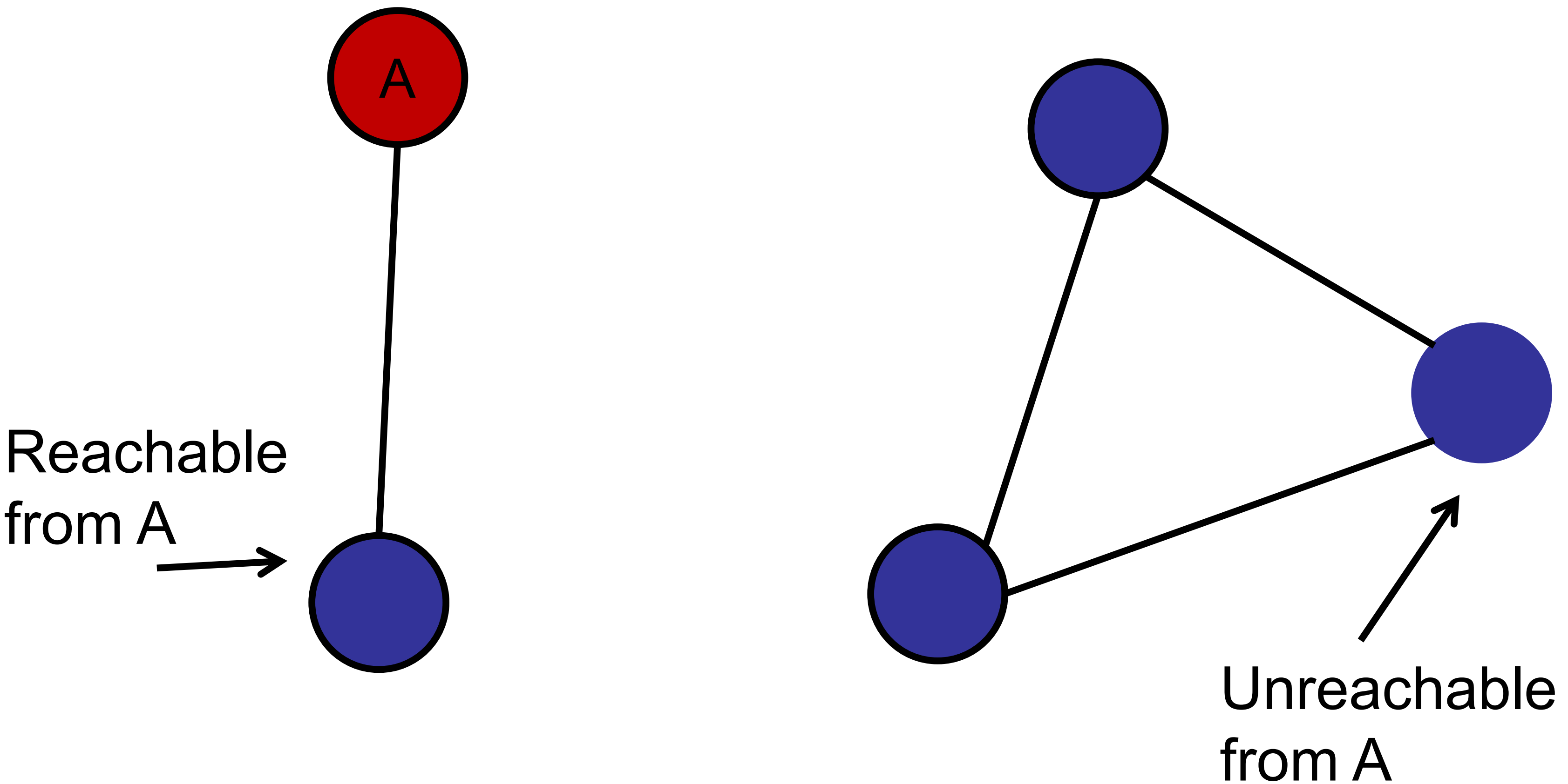


- Two **connected components**.

Graph Terminology

Disconnected:

– Some pair of nodes is not connected by a path.

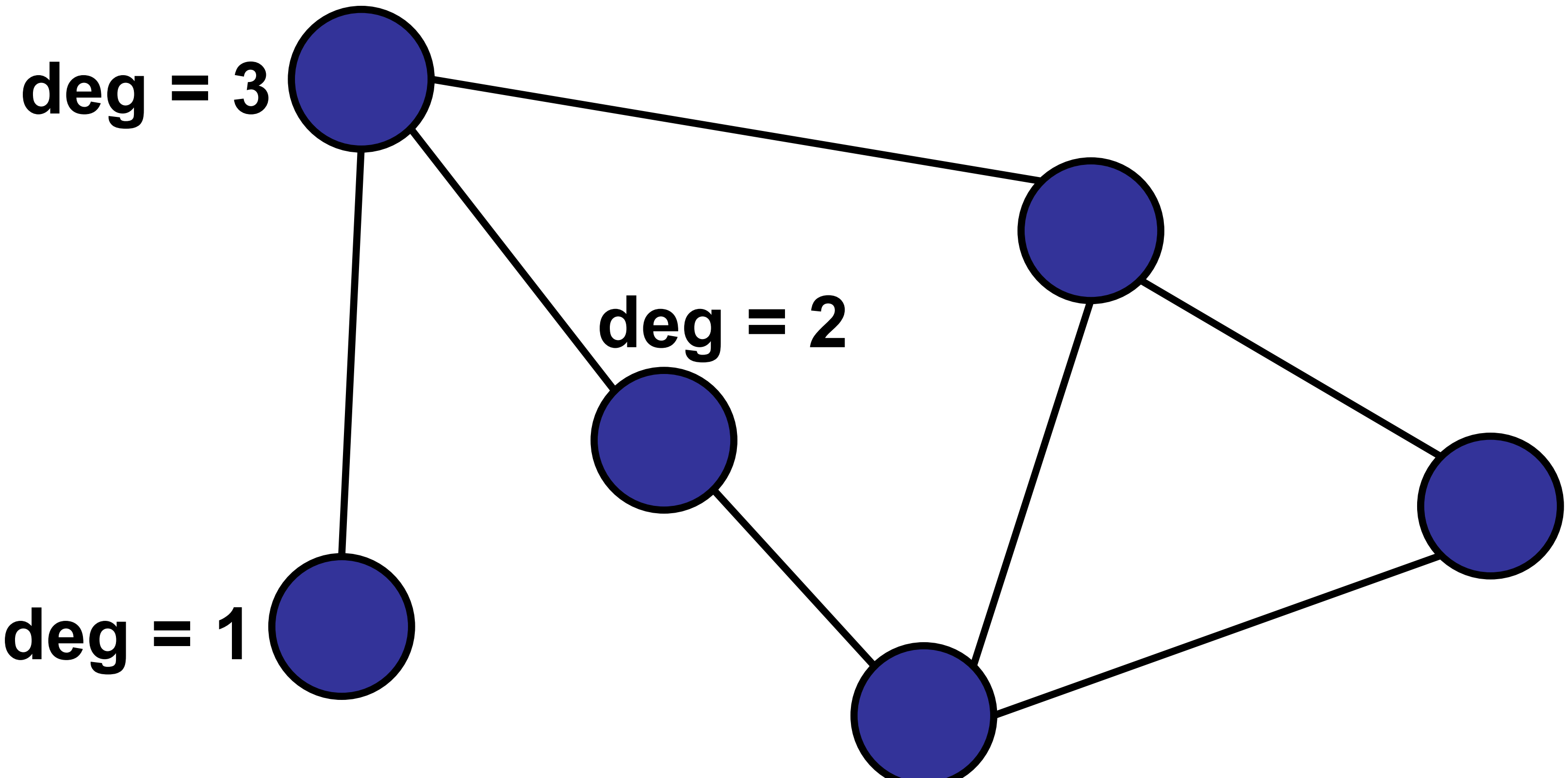


– Two **connected components.**

Graph Terminology

Degree of a node:

– Number of **adjacent** edges.

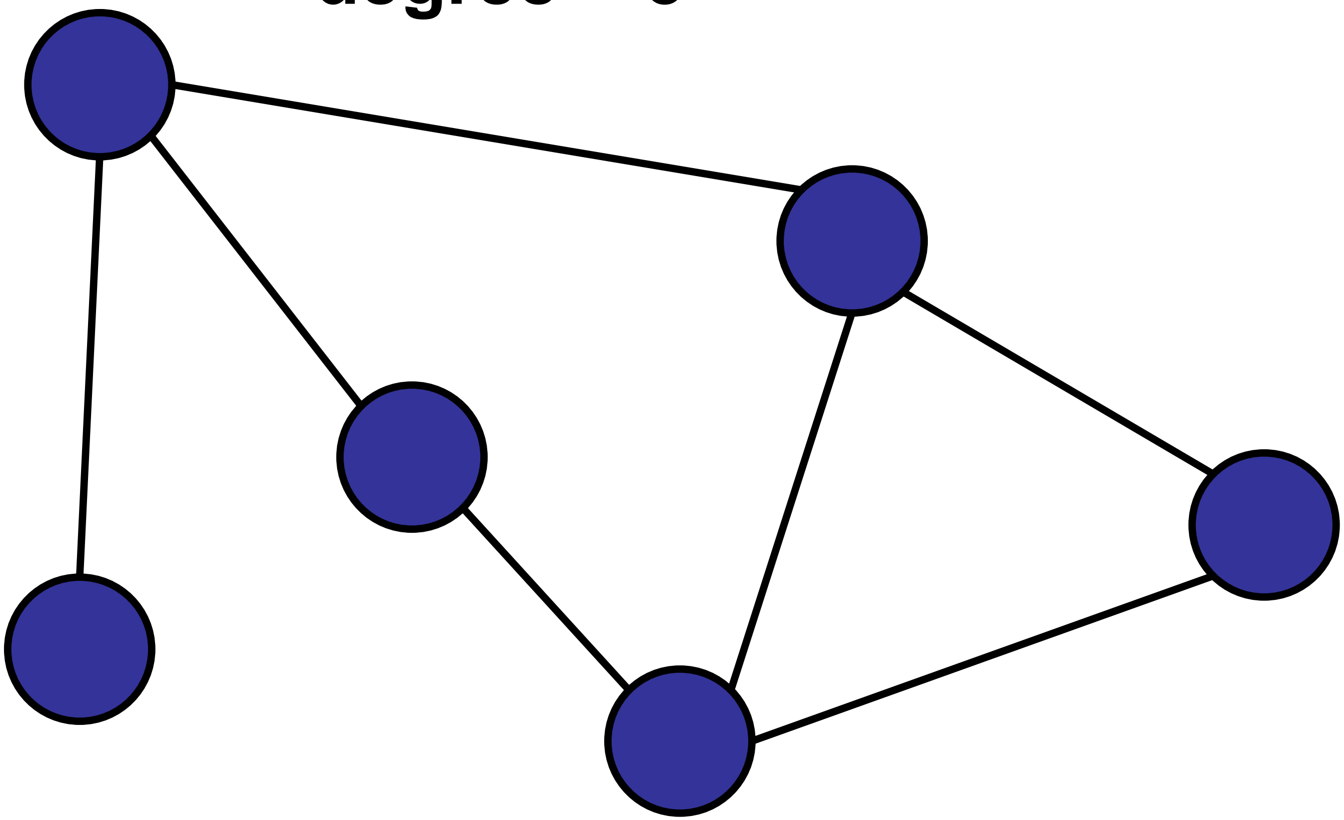


Graph Terminology

Degree of a graph:

- Maximum number of **adjacent** edges.

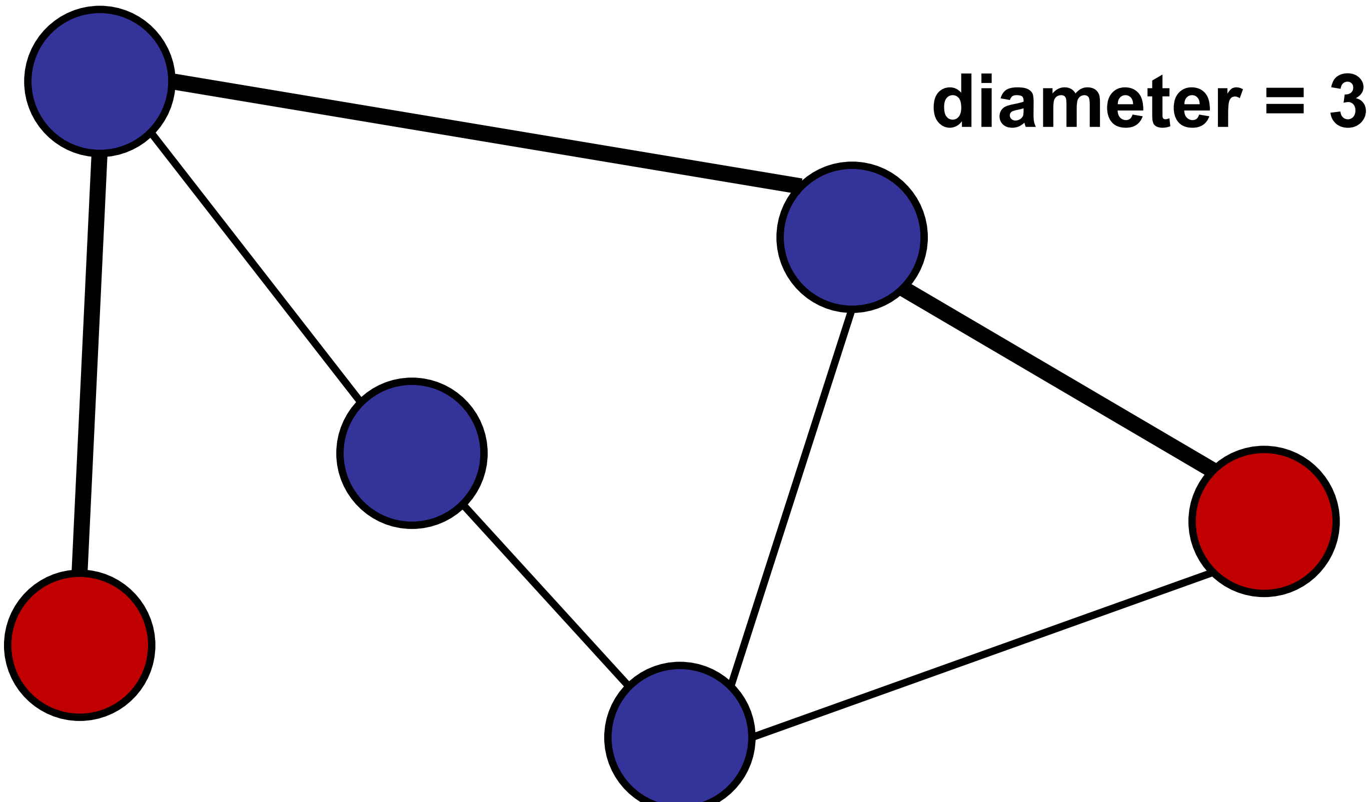
degree = 3



Graph Terminology

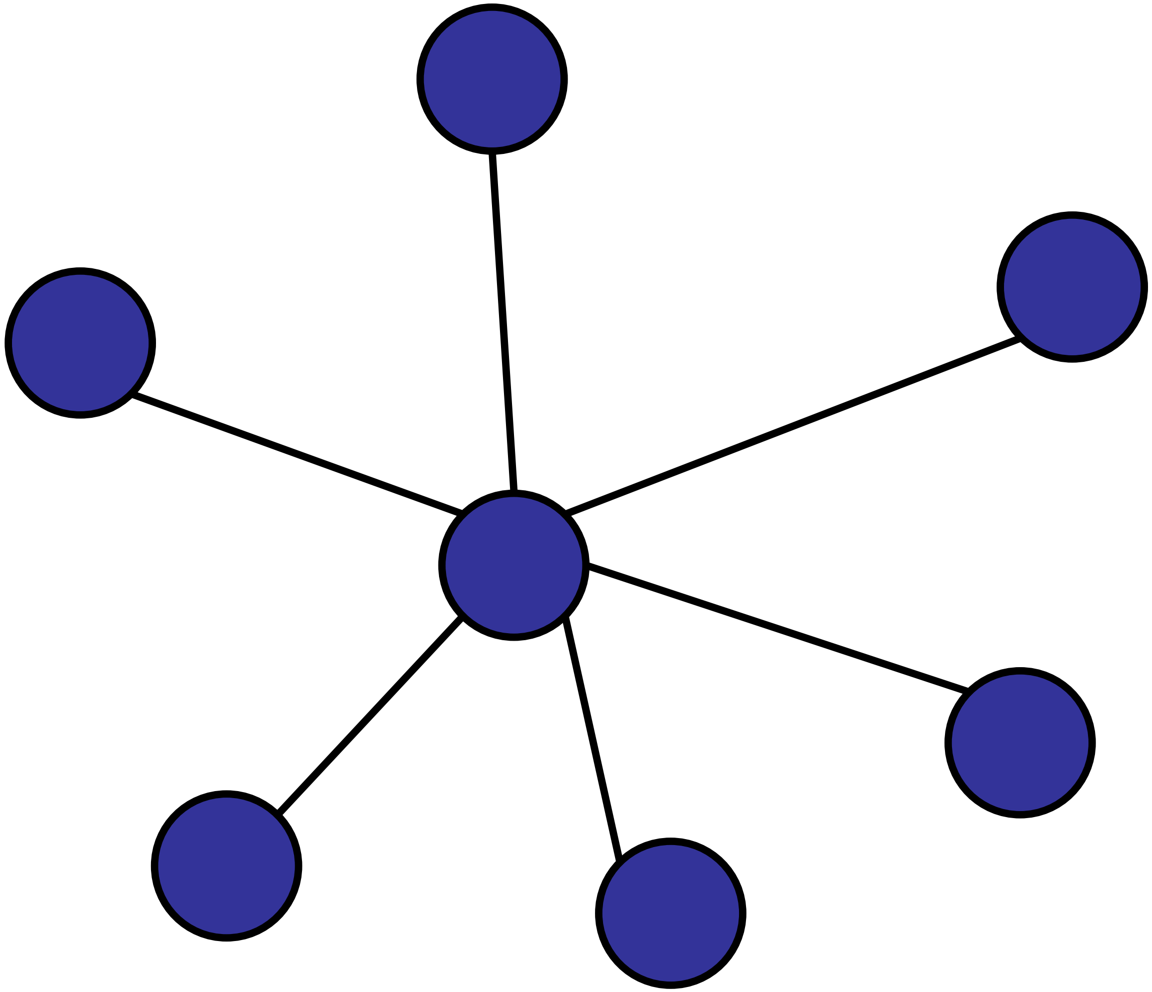
Diameter:

– Maximum distance between two nodes, following the **shortest** path.



Special Graphs

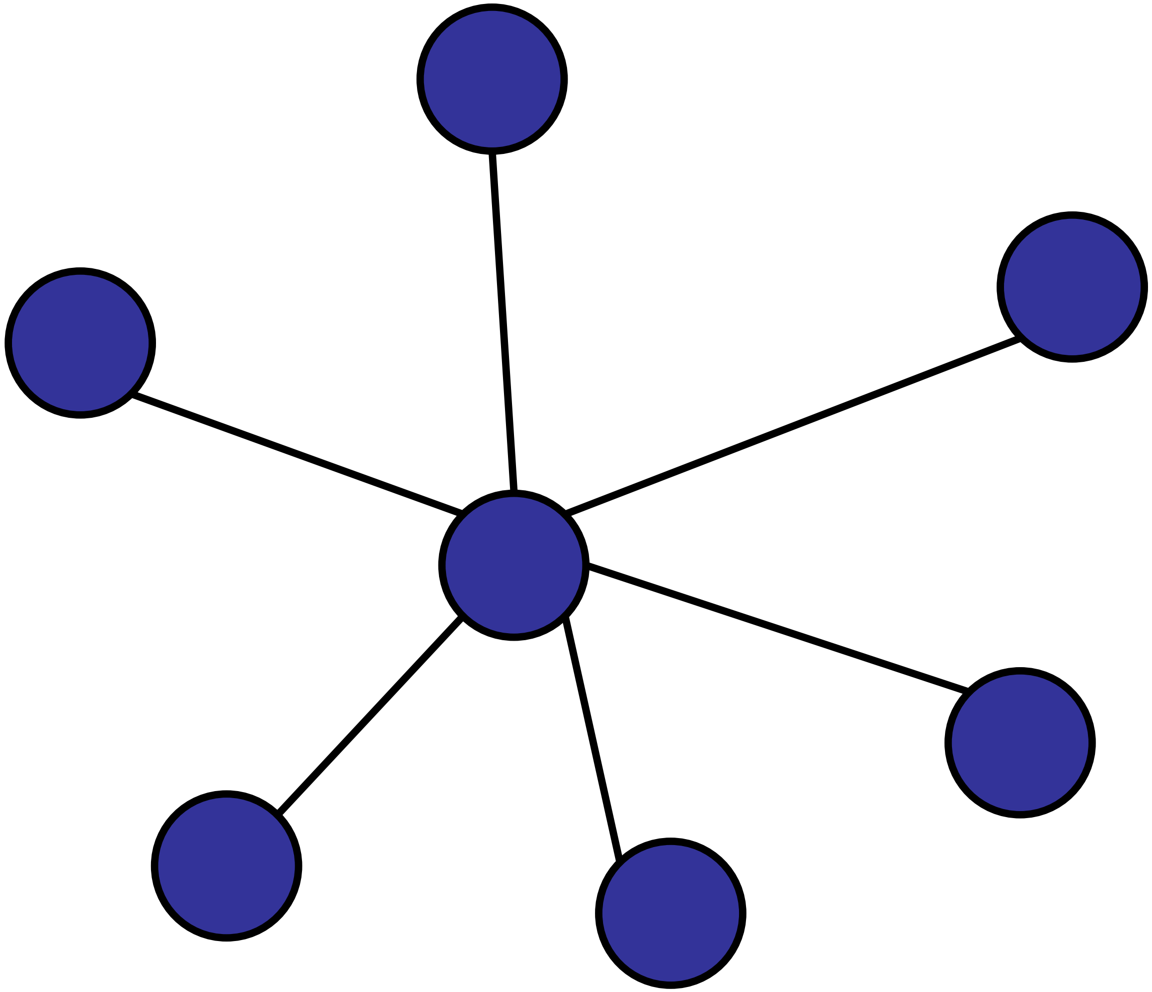
Star



One central node, all edges connect center to edges.

Special Graphs

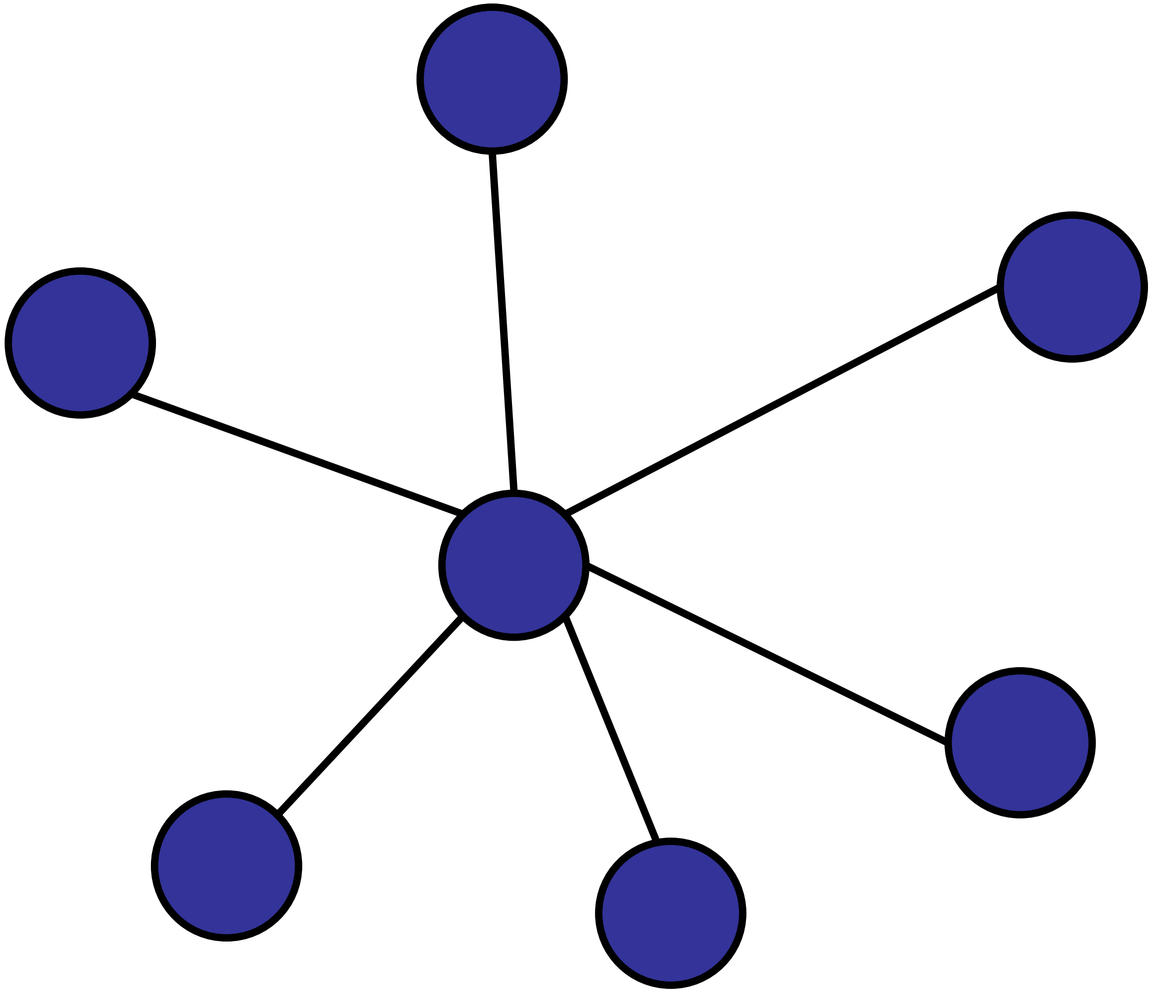
Star



One central node, all edges connect center to edges.

Special Graphs

Star



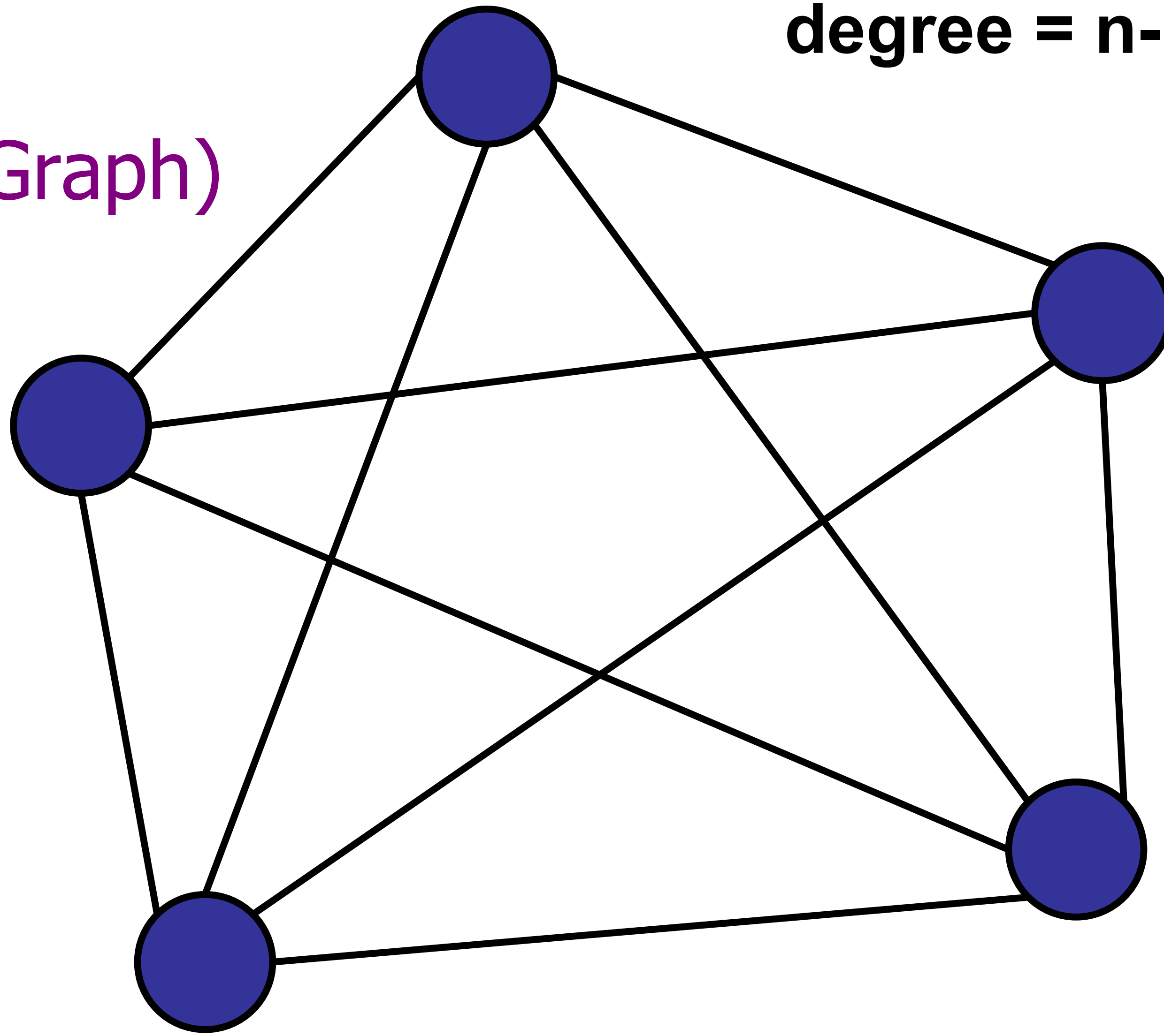
One central node, all edges connect center to edges.

Special Graphs

Clique
(Complete Graph)

diameter = 1

degree = n-1



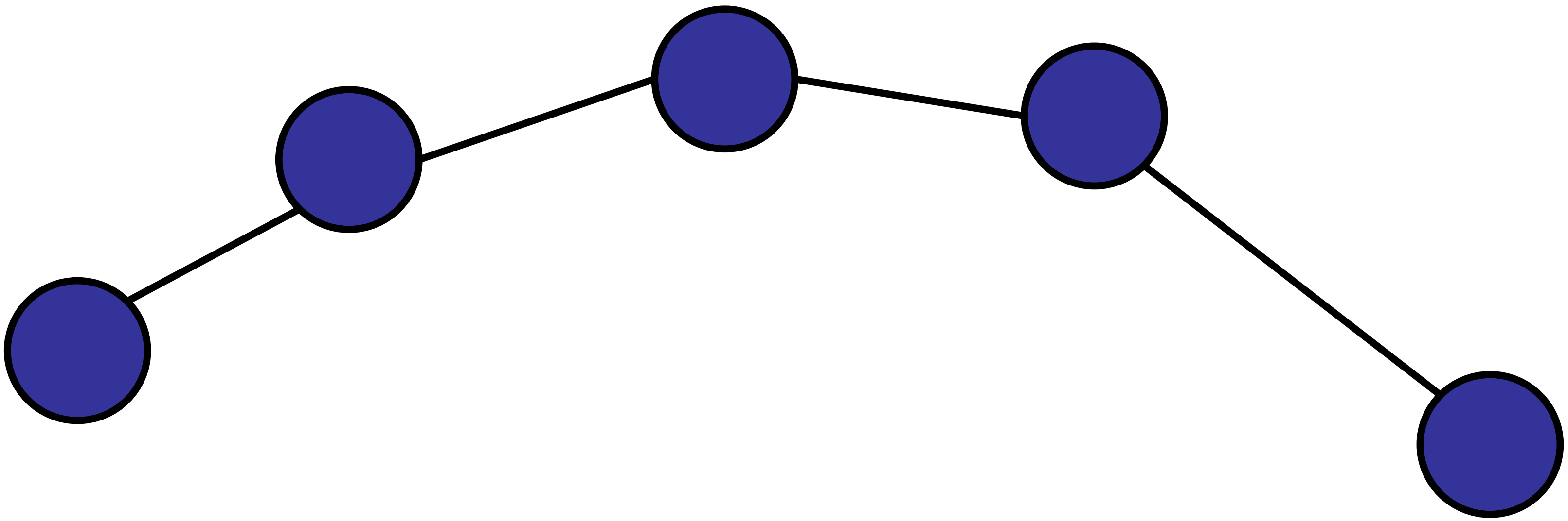
All pairs connected by edges.

Special Graphs

diameter = n-1

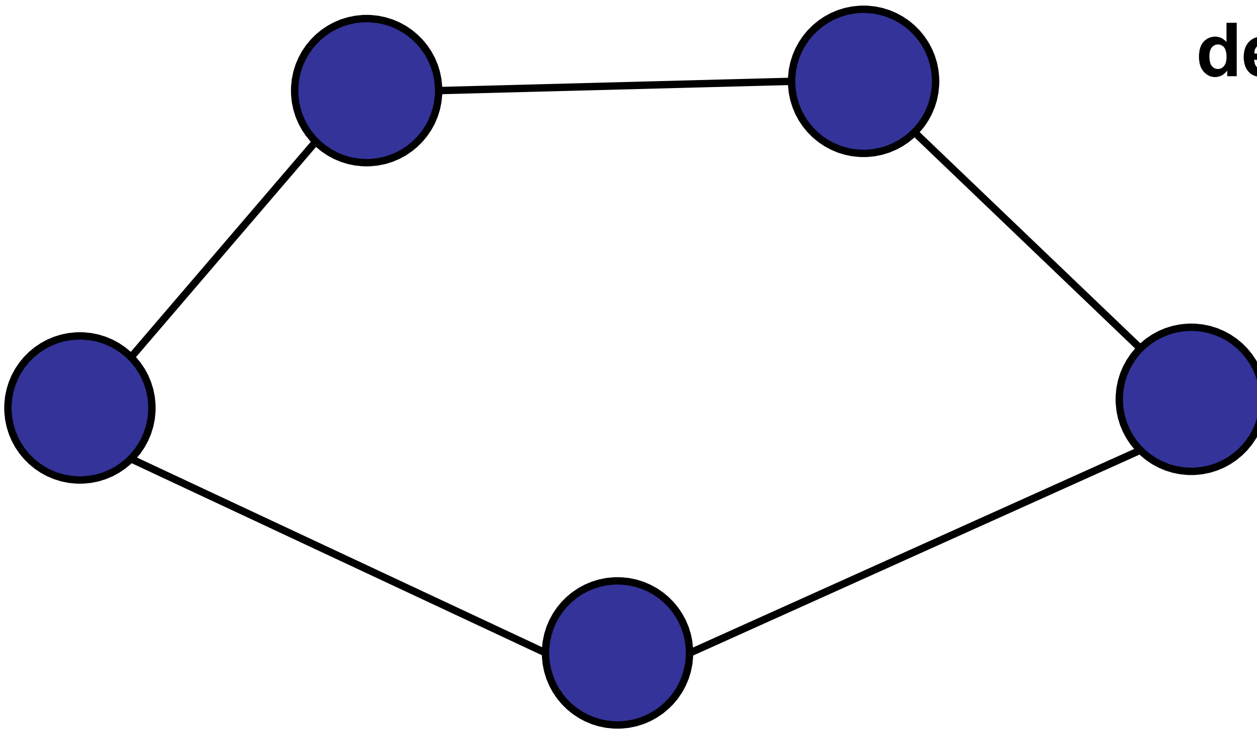
degree = 2

Line (or path)



Special Graphs

Cycle



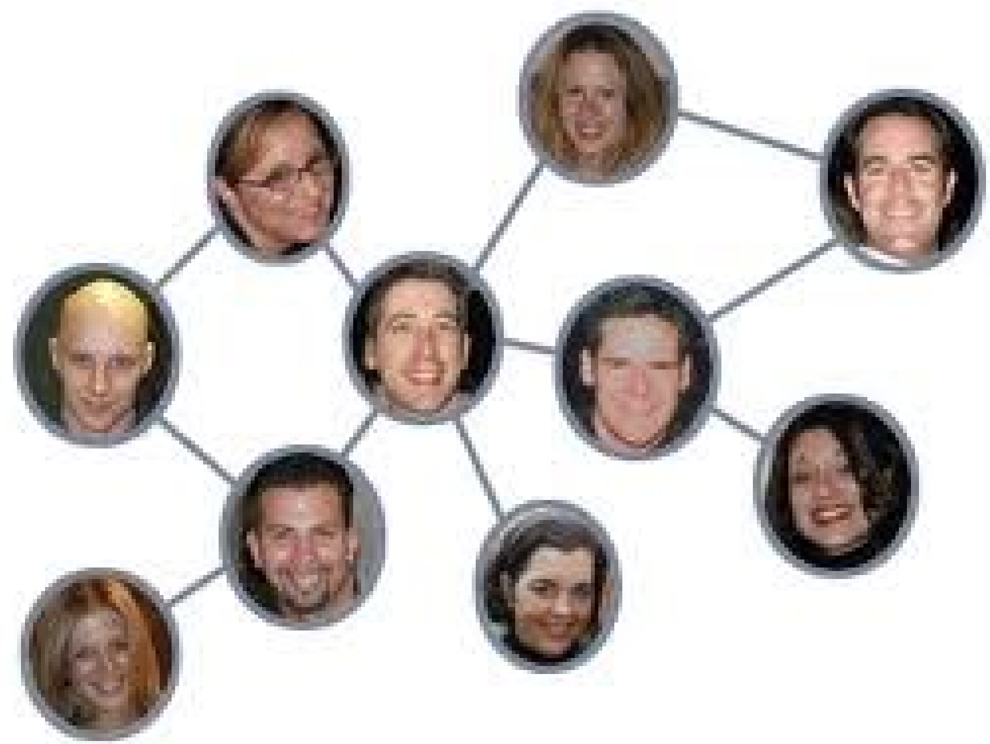
diameter = n/2
or
diameter = n/2-1

degree = 2

Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship



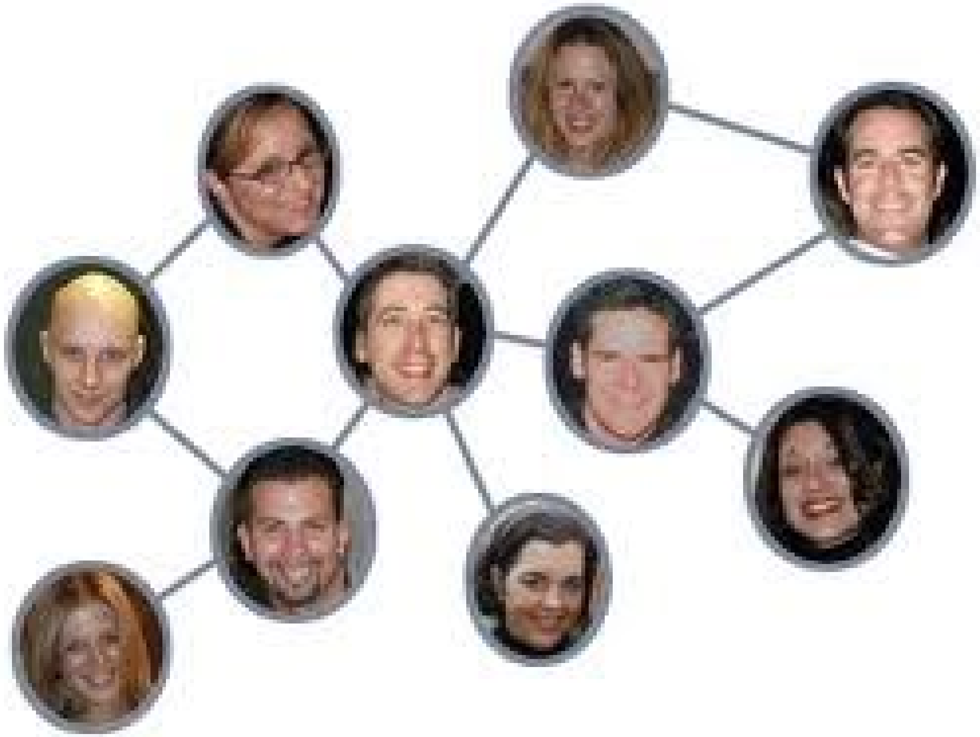
Where do we find graphs?

Social network:

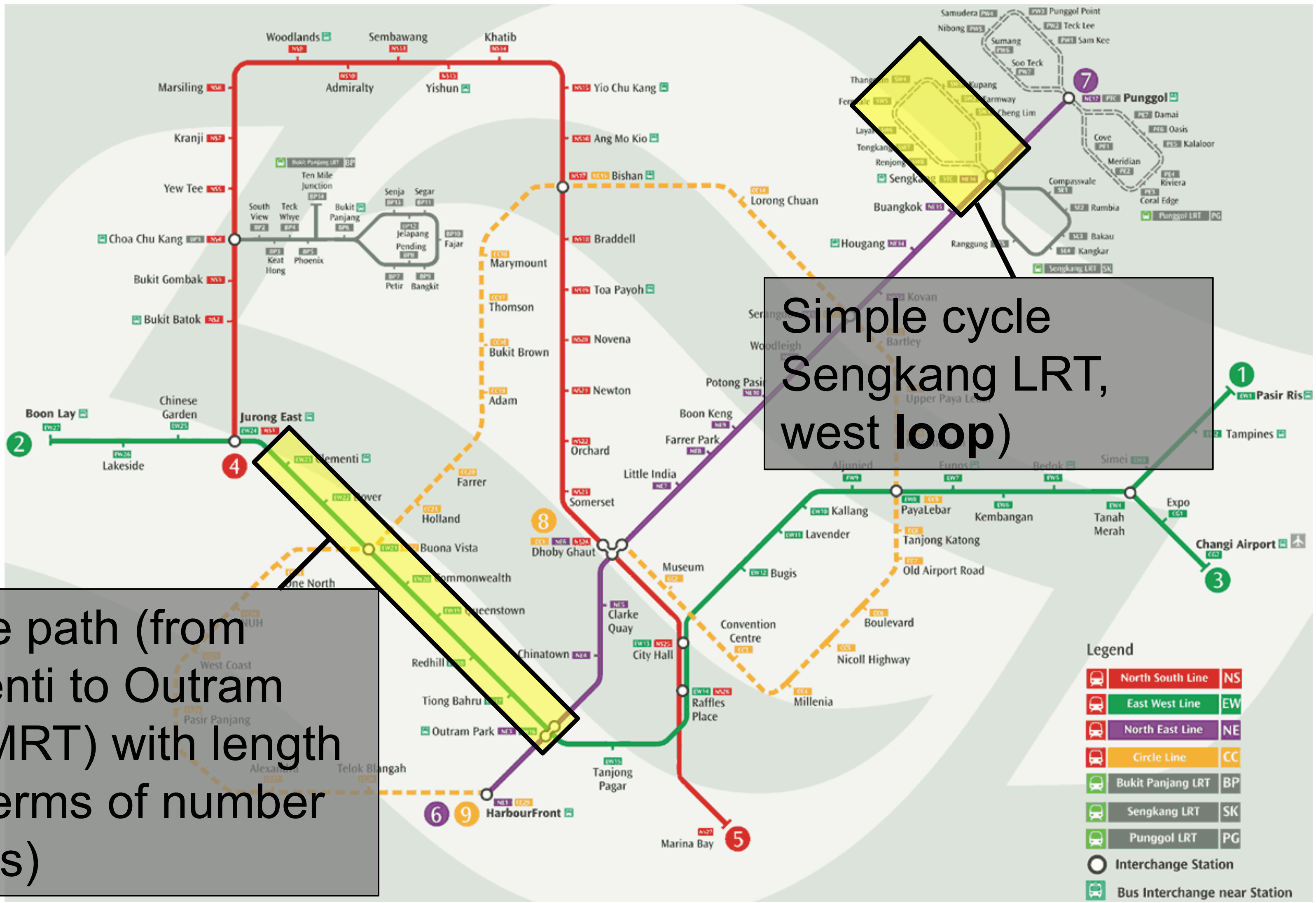
- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?



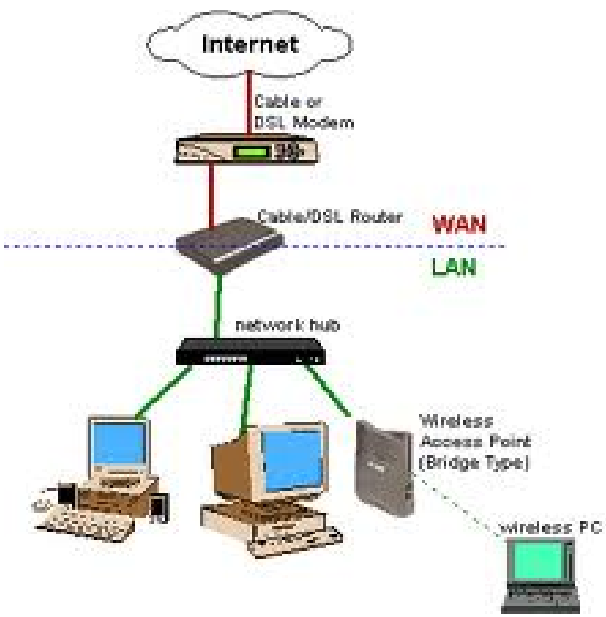
Transportation Network

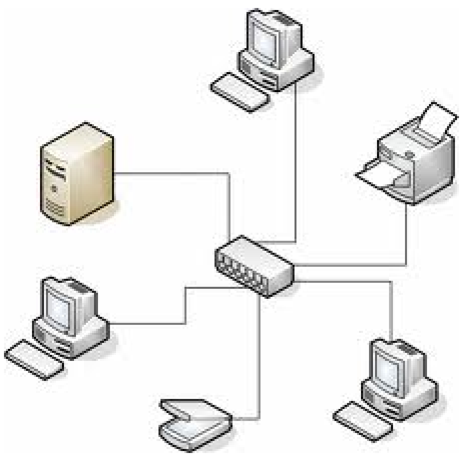


Simple cycle Sengkang LRT, west **loop**)

Simple path (from Clementi to Outram Park MRT) with length 7 (in terms of number of hops)

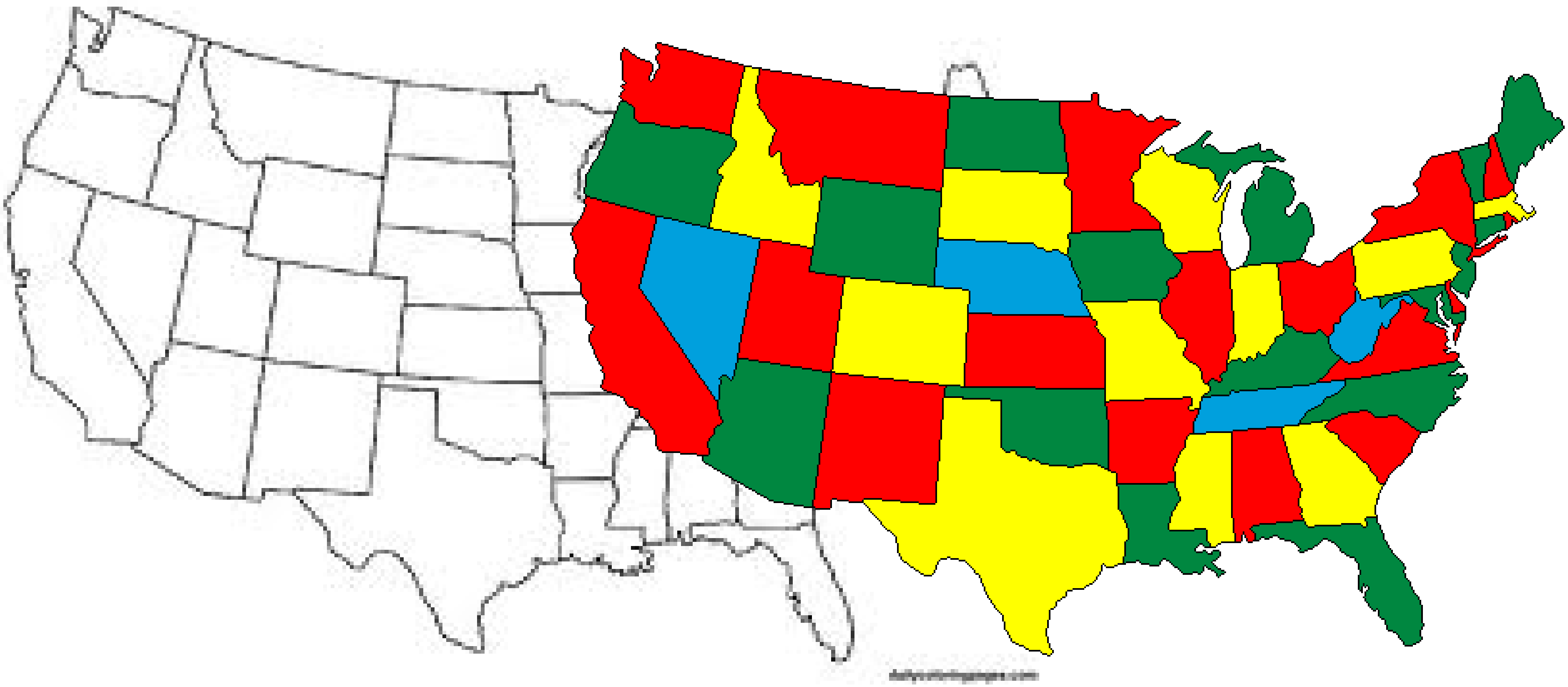
Internet / Computer Networks



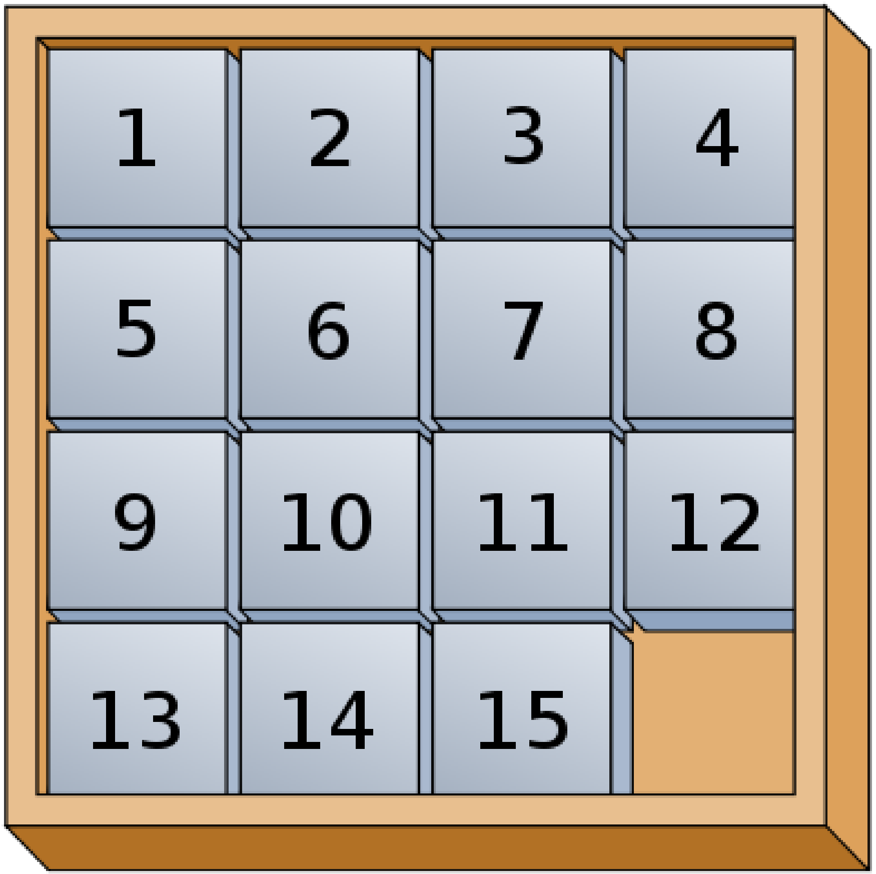


Communication Network

Optimization



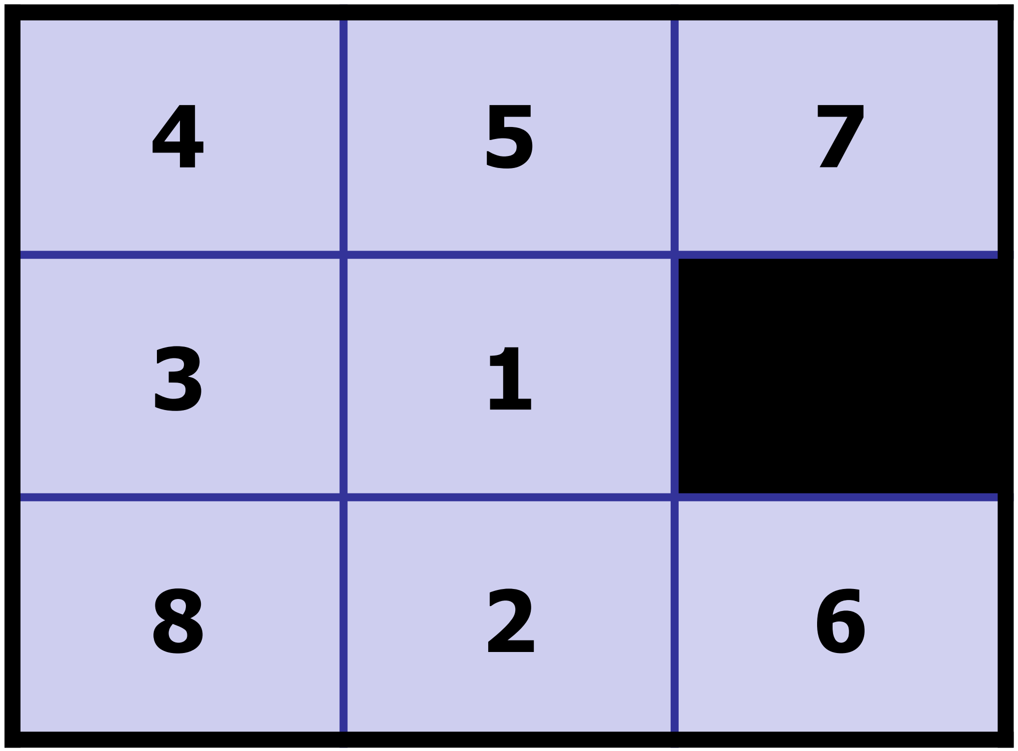
Sliding Puzzle



Sliding Puzzle

4	5	7
3	1	6
8	2	

Sliding Puzzle



Sliding Puzzle

4	5	
3	1	7
8	2	6

Sliding Puzzle

4		5
3	1	7
8	2	6

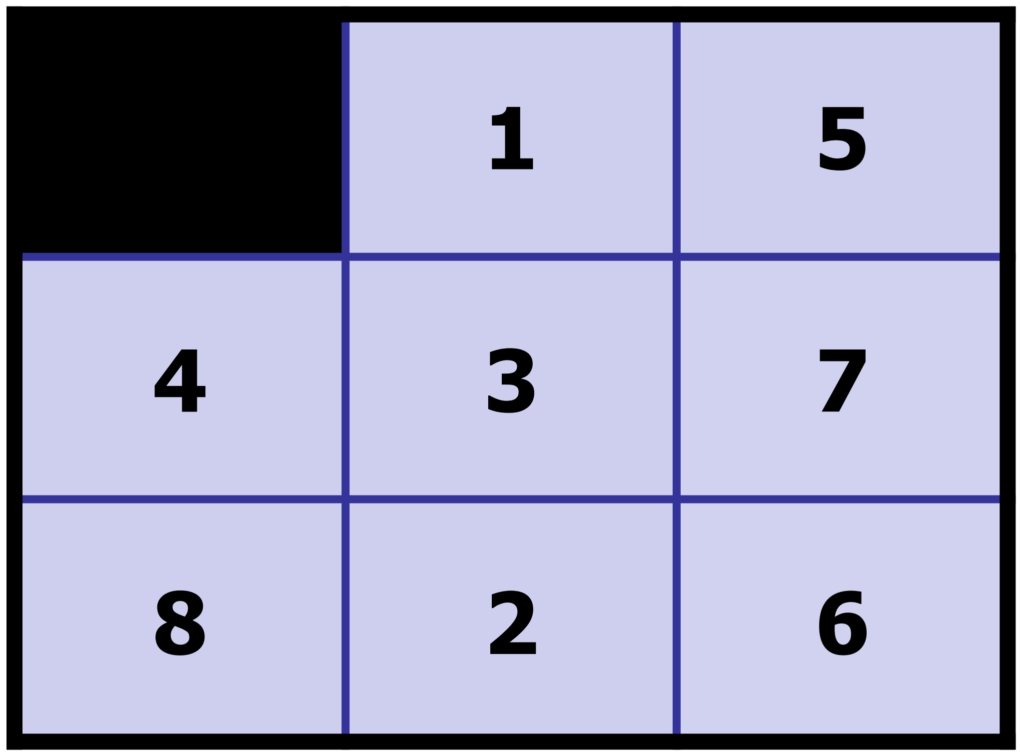
Sliding Puzzle

4	1	5
3		7
8	2	6

Sliding Puzzle

4	1	5
	3	7
8	2	6

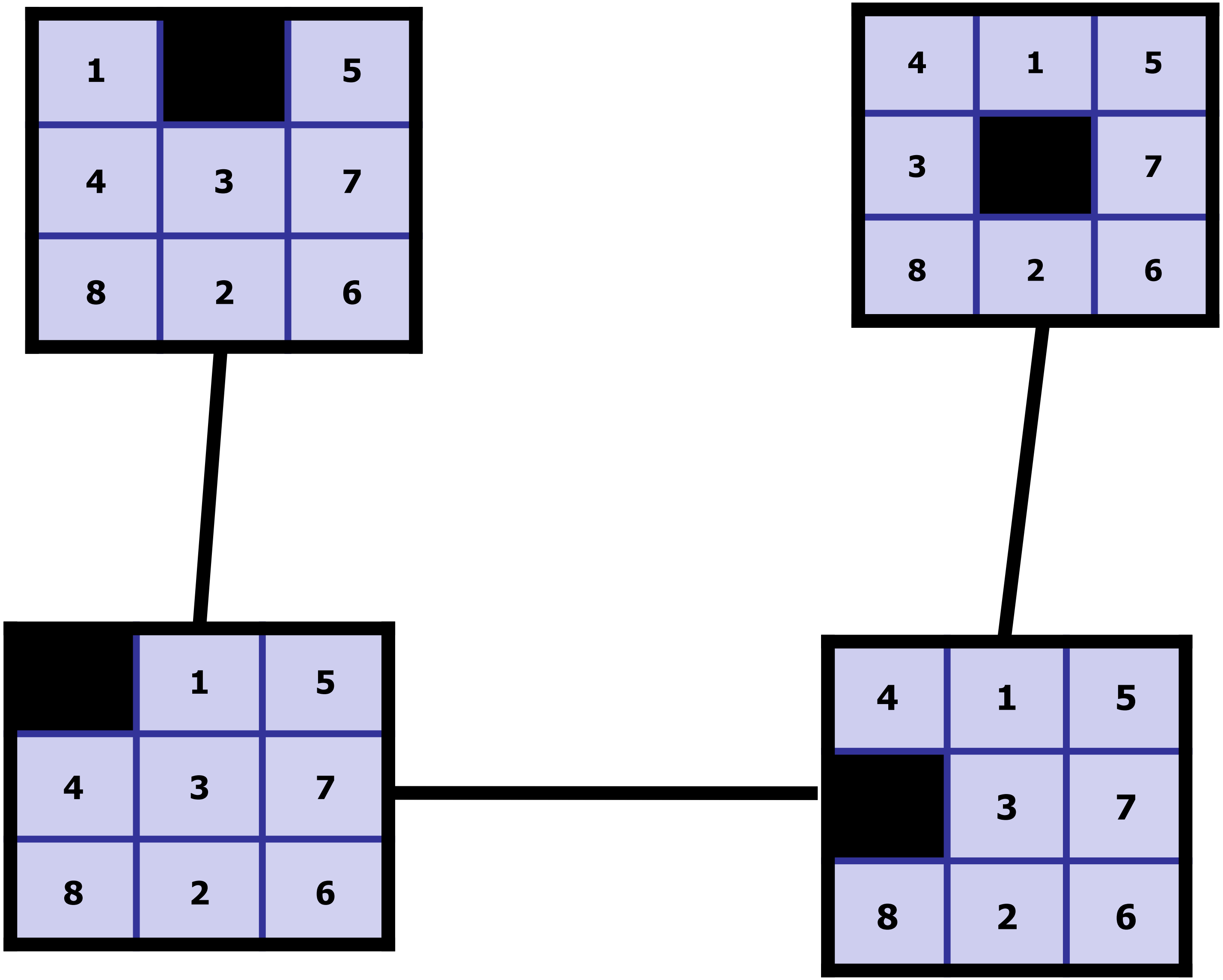
Sliding Puzzle



Sliding Puzzle

1		5
4	3	7
8	2	6

Sliding Puzzle is a Graph



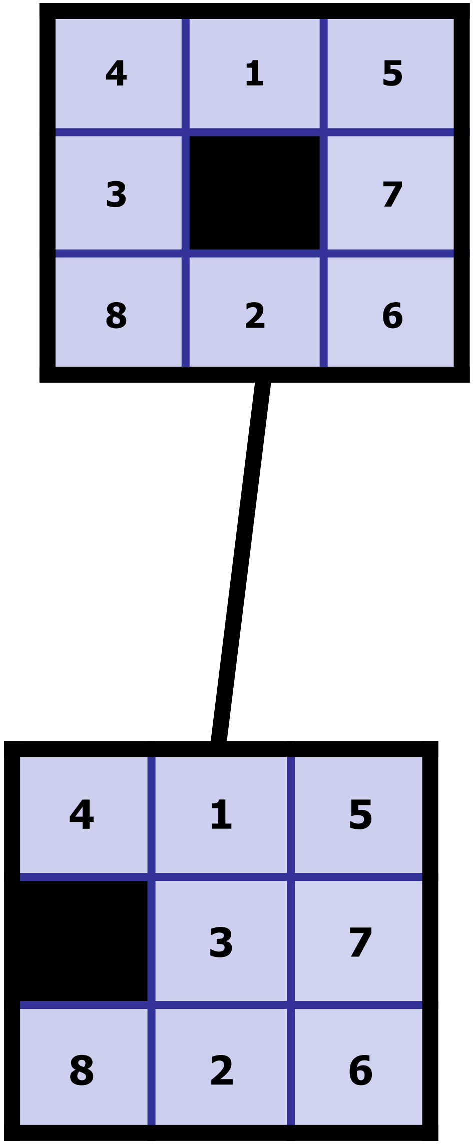
Sliding Puzzle

Nodes:

- State of the puzzle
- Permutation of nine tiles

Edges:

- Two states are edges if they differ by only one move.



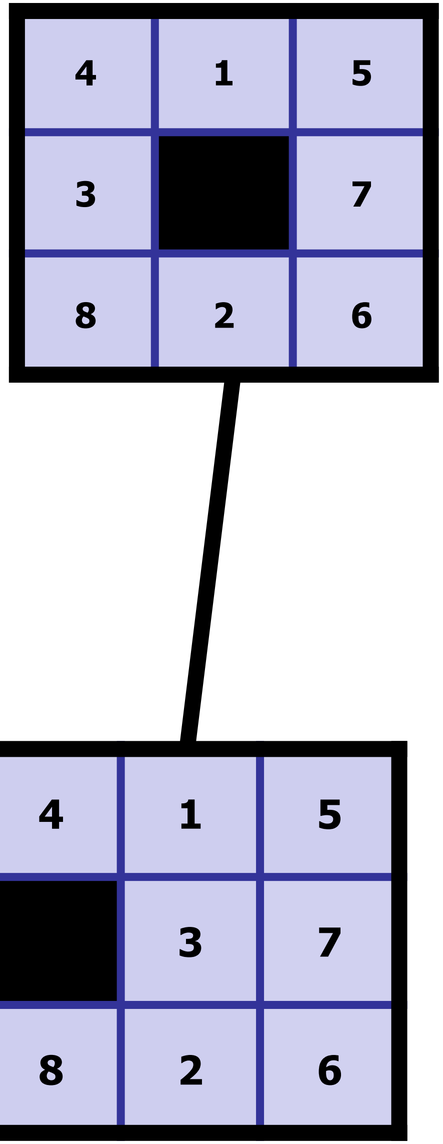
Sliding Puzzle

Nodes:

- State of the puzzle
- Permutation of nine tiles

Edges:

- Two states are edges if they differ by only one move.



Nodes = 9! = 362,880

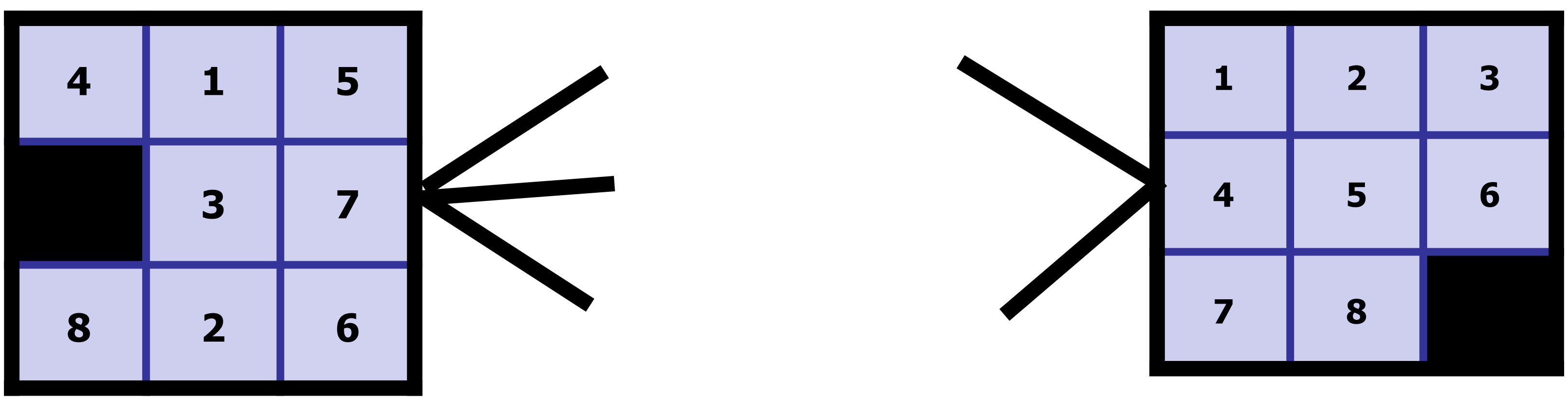
Edges < 4*9! < 1,451,520

Sliding Puzzle

Number of moves to solve the puzzle?

Initial, scrambled state:

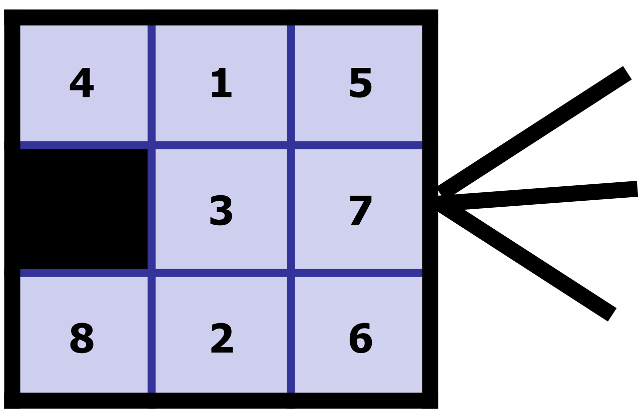
Final, unscrambled state:



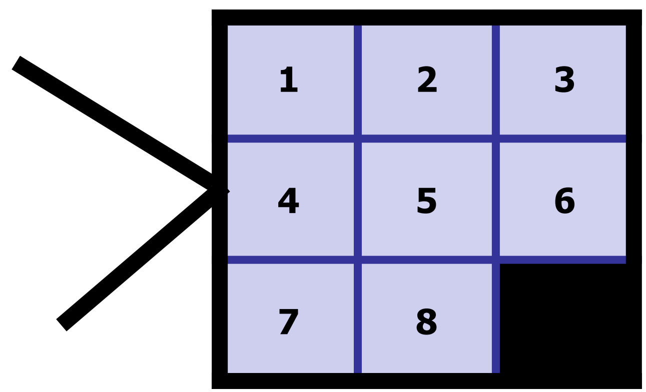
Sliding Puzzle

Number of moves <= Diameter

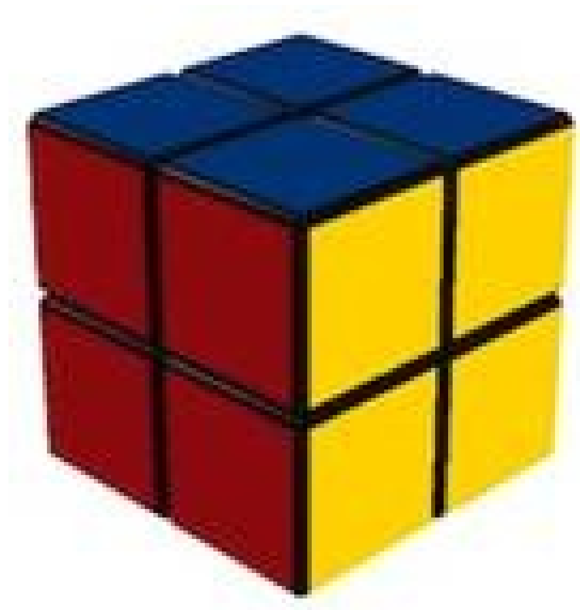
Initial, scrambled state:



Final, unscrambled state:

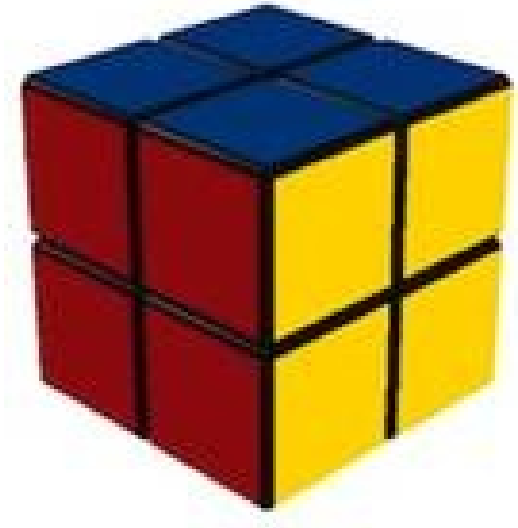


2 x 2 x 2 Rubik's Cube



Record solve time: 0.69 seconds

2 x 2 x 2 Rubik's Cube



Configuration Graph

- Vertex for each possible state
- Edge for each basic move
  - 90 degree turn
  - 180 degree turn

Puzzle: given initial state, find a path to the solved state.

2 x 2 x 2 Rubik's Cube

How many vertices?

$$8! \cdot 3^8 = 264{,}539{,}520$$

cubelets

Each cubelet is
in one of 8 positions.

Each of the 8 cubelets
can be in one of three
orientations

2 x 2 x 2 Rubik's Cube

How many vertices?

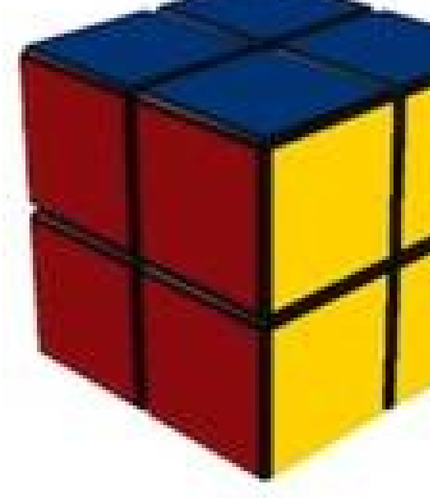
$$7! \cdot 3^7 = 11{,}022{,}480$$

Symmetry:

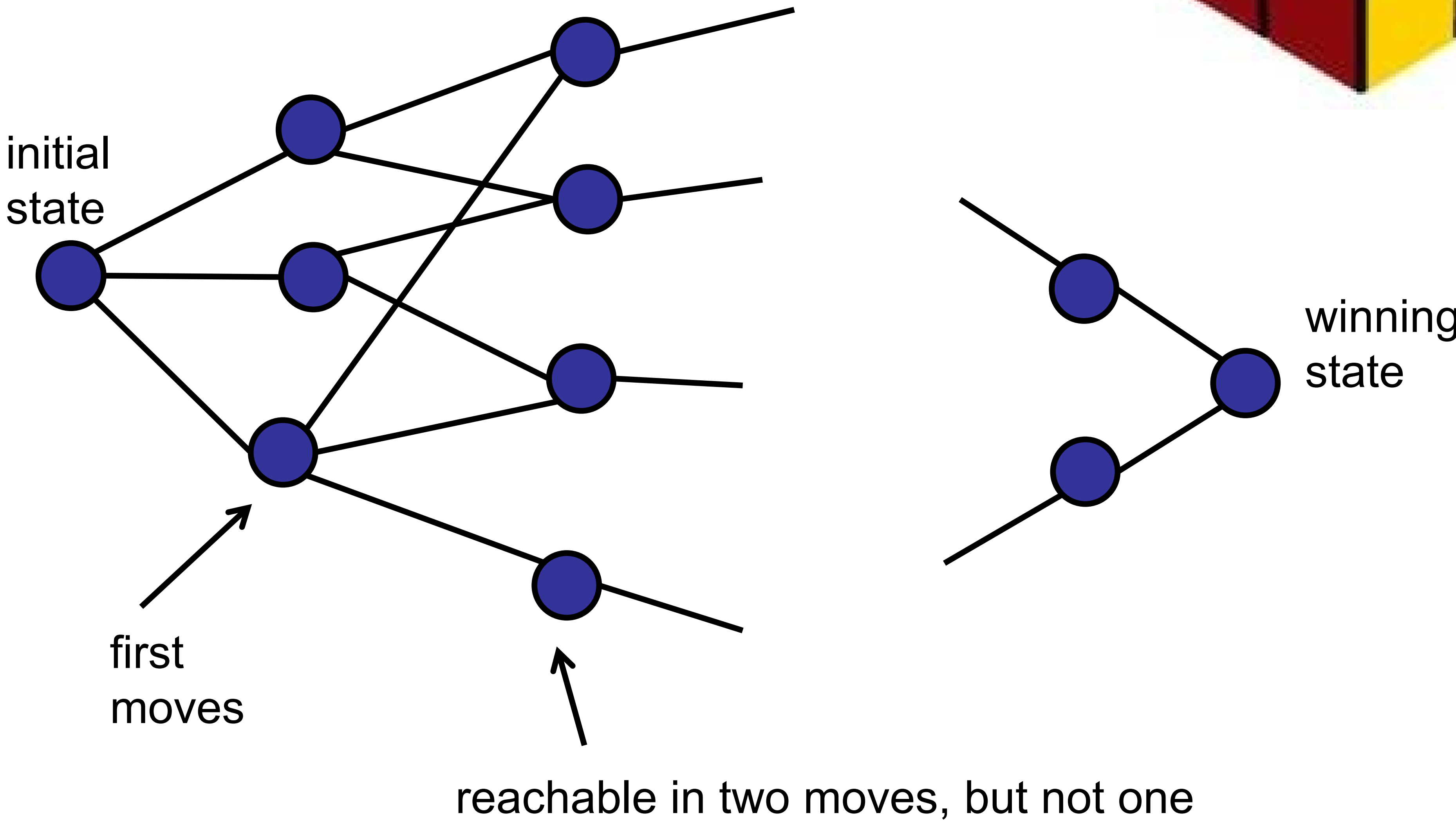
Fix one cubelet.

Each of the 8 cubelets can be in one of three orientations

2 x 2 x 2 Rubik's Cube

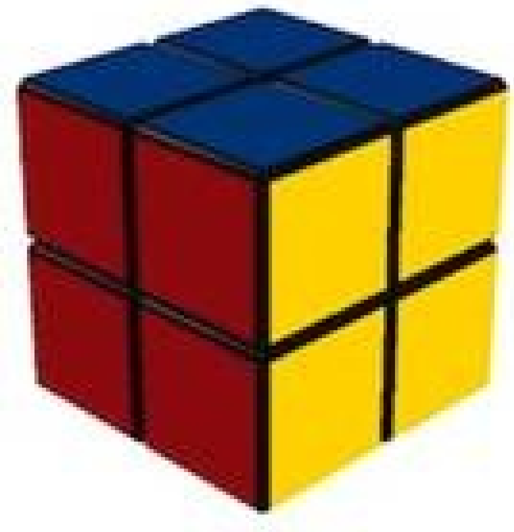


Geography of Rubik's configurations:



#configurations requires n turns

n	90 deg. Turns only	90/180 deg. turns
0	1	1
1	6	9
2	27	54
3	120	321
4	534	1,847
5	2,256	9,992
6	8,969	50,136
7	33,058	227,536
8	114,149	870,072
9	360,508	1,887,748
0	930,588	623,800
11	1,350,852	2,644
12	782,536	
13	90,280	
14	276	

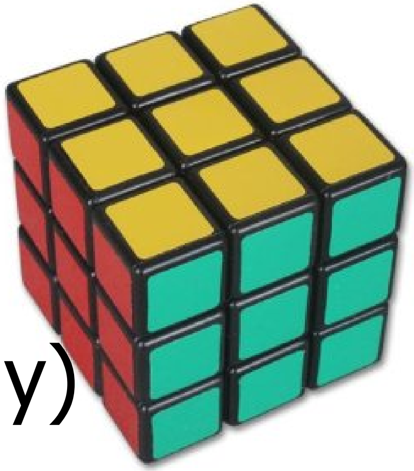


#configurations requires n turns

n	90 deg. turns	90/180 deg. turns
0	1	1
1	6	9
2	27	54
3		
4		
5		
6		
7		
8		
9	360,508	1,887,748
0	930,588	623,800
11	1,350,852	2,644
12	782,536	
13	90,280	
14	276	

Challenge:
How do you generate this table?

3 x 3 x 3 Rubik's Cube



Configuration Graph

- 43 quintillion vertices (approximately)
- Diameter: 20
  - 1995: require at least 20 moves.
  - 2010: 20 moves is enough from every position.
  - Using Google server farm.
  - 35 CPU-years of computation.
  - 20 seconds / set of 19.5 billion positions.
  - Lots of mathematical and programming tricks.

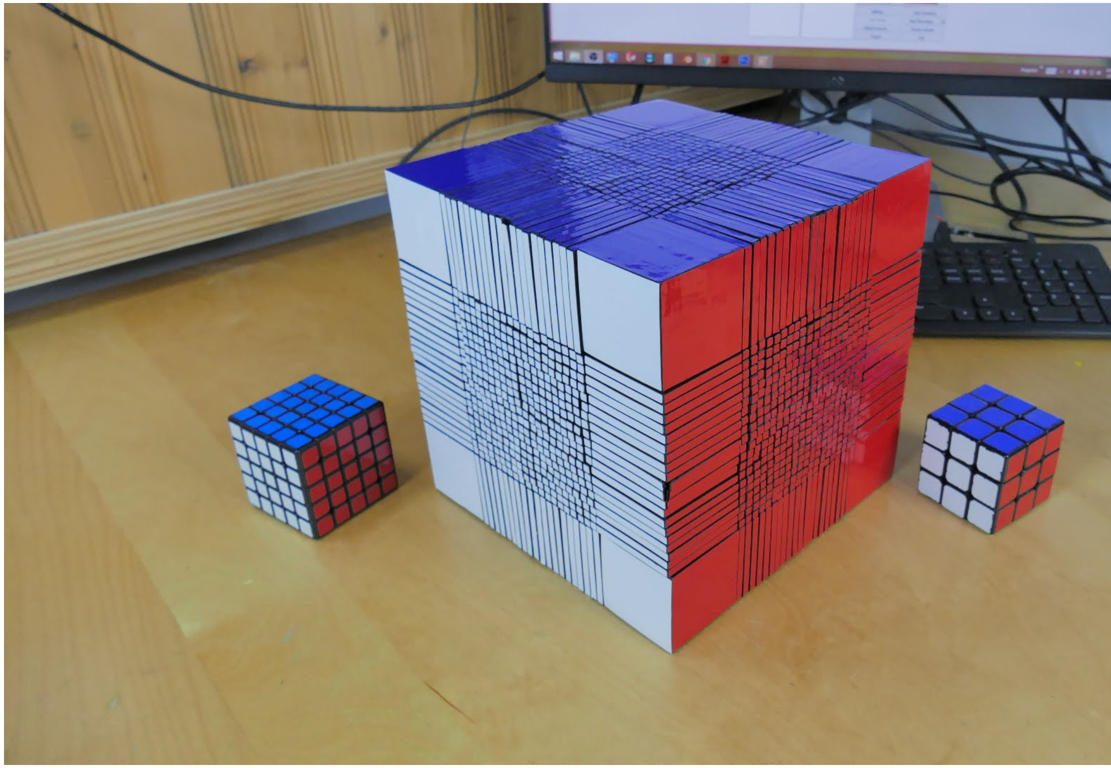
Proof of the max no. of moves needed

Date	Lower bound	Upper bound	Gap	Notes and Links
July, 1981	18	52	34	Morwen Thistlethwaite proves 52 moves suffice.
December, 1990	18	42	24	Hans Kloosterman improves this to 42 moves.
May, 1992	18	39	21	Michael Reid shows 39 moves is always sufficient.
May, 1992	18	37	19	Dik Winter lowers this to 37 moves just one day later!
January, 1995	18	29	11	Michael Reid cuts the upper bound to 29 moves by analyzing Kociemba's two-phase algorithm.
January, 1995	20	29	9	Michael Reid proves that the "superflip" position (corners correct, edges placed but flipped) requires 20 moves.
December, 2005	20	28	8	Silviu Radu shows that 28 moves is always enough.
April, 2006	20	27	7	Silviu Radu improves his bound to 27 moves.
May, 2007	20	26	6	Dan Kunkle and Gene Cooperman prove 26 moves suffice.
March, 2008	20	25	5	Tomas Rokicki cuts the upper bound to 25 moves.
April, 2008	20	23	3	Tomas Rokicki and John Welborn reduce it to only 23 moves.
August, 2008	20	22	2	Tomas Rokicki and John Welborn continue down to 22 moves.
July, 2010	20	20	0	Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge prove that God's Number for the Cube is exactly 20.

3 x 3 x 3 Rubik's Cube

What is the diameter of an (n x n x n) cube?

- a 22 x 22 x 22 Rubik's Cube
- Link



In general:

$\theta(n^2 / \log n)$

Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- Searching graphs (DFS / BFS)

Representing a Graph

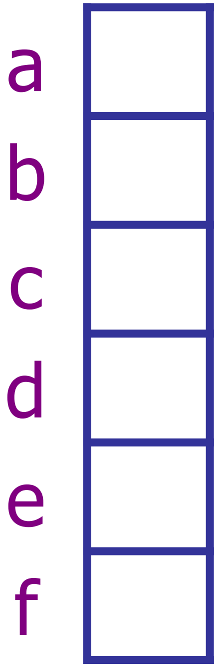
Graph consists of:

- Nodes
- Edges

Representing a Graph

Graph consists of:

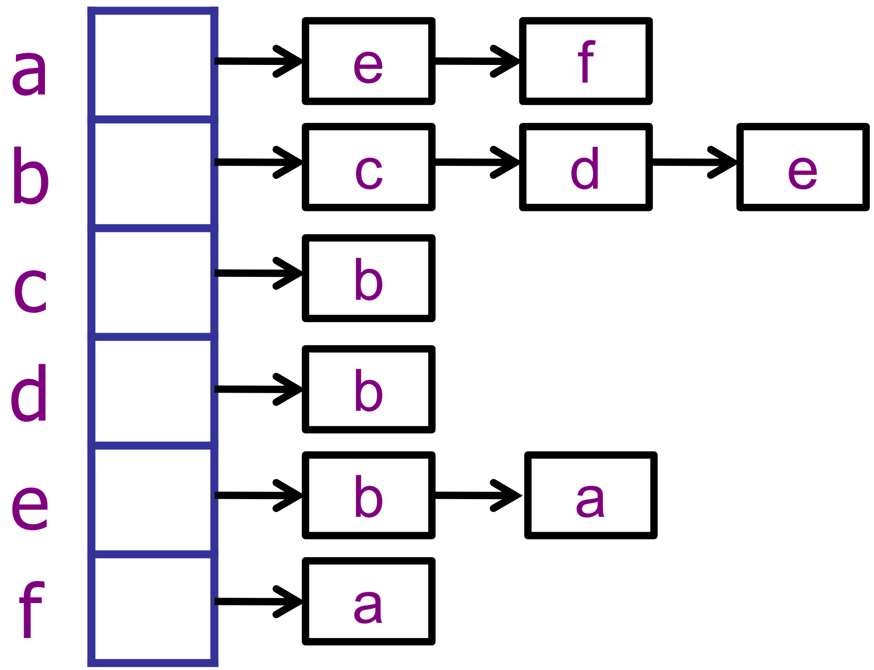
- Nodes: stored in an array
- Edges

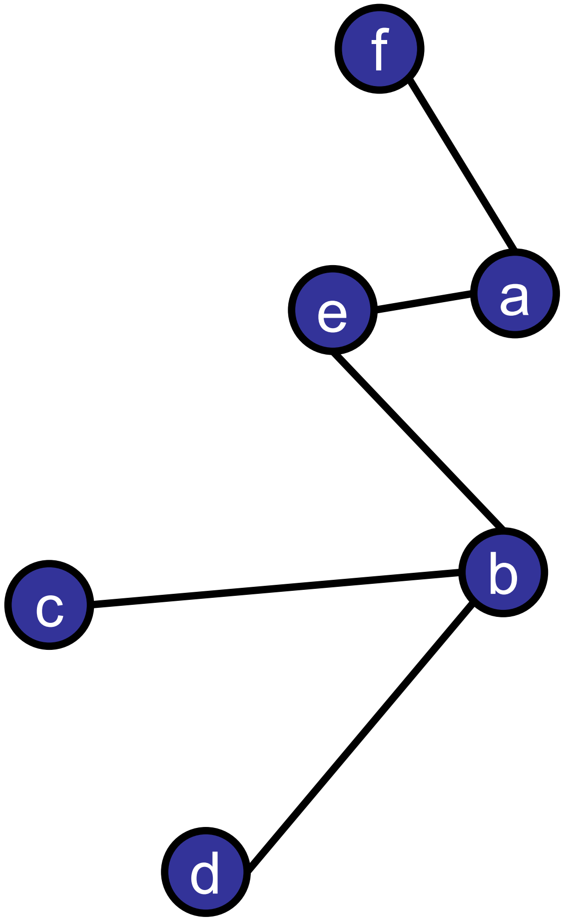


Adjacency List

Graph consists of:

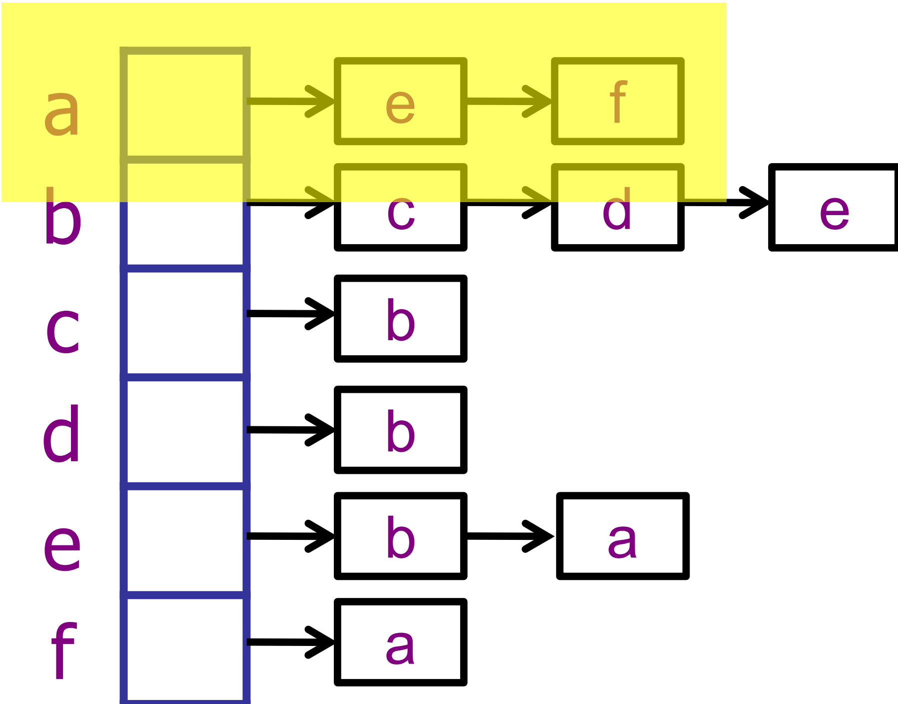
- Nodes: stored in an array
- Edges: linked list per node





Adjacency List in C++

```
class Node {
  int key;
   LinkedList<int>;
}

class Graph {
  Node nodeList[MAXNODE];


}
```

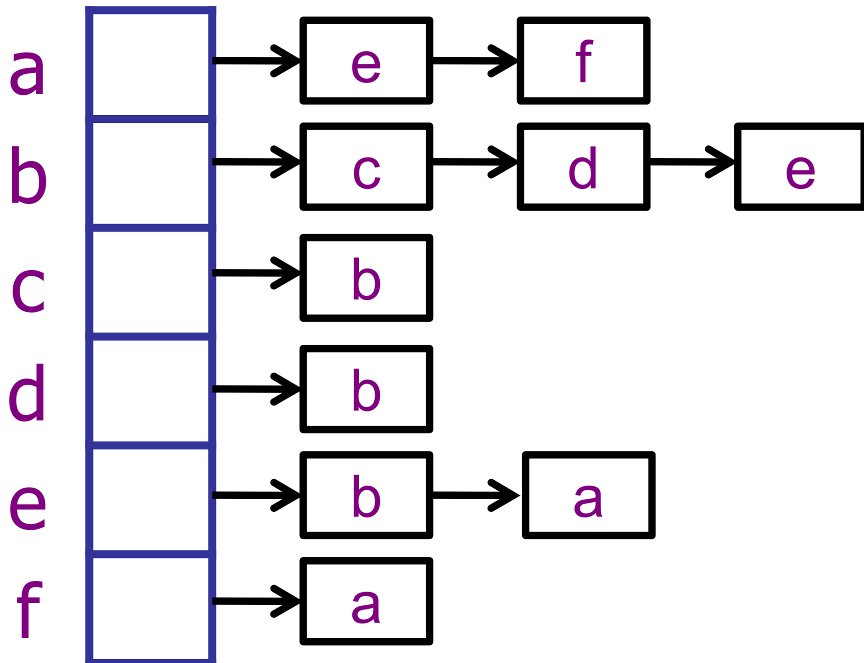


Adjacency List in C++

```
class Graph{
    LinkedList<LinkedList<int>> m_nodes;


}
```

More concise code is not *always* better…

- Harder to read
- Harder to debug
- Harder to extend



Representing a Graph

Graph consists of:

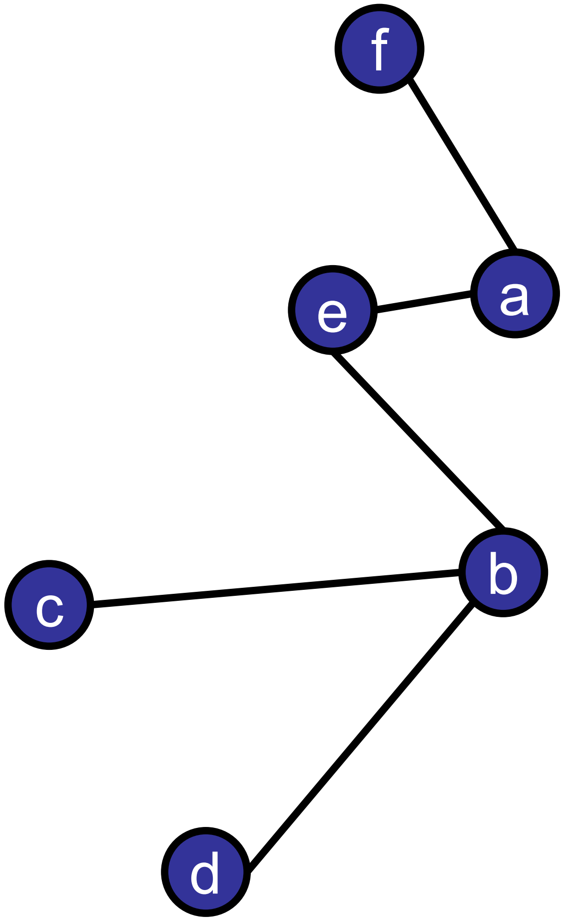
- Nodes
- Edges = pairs of nodes

Adjacency **Matrix**

Graph consists of:

- Nodes
- Edges = pairs of nodes

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**1**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**1**	**1**	0	0	0	0
f	**1**	0	0	0	0	0



Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

Neat property:

- A^2 = length 2 paths

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

Adjacency Matrix

To find out if c and d are 2-hop neighbors:

- Let B = A^2.
- B[c, d] = A[c, .]•A[., d]

- B[c, d] = 1 iff

  A[c, x] == A[x, d]

  for some x.

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

Adjacency Matrix

To find out if c and d are 2-hop neighbors:

- Let $B = A^2$.
- $B[c, d] = A[c, .] \bullet A[., d] > 0 \; ? \; 1 : 0$

- $B[c, d] = 1$ iff

  $A[c, x] == A[x, d]$

  for some x.

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**1**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**1**	**1**	0	0	0	0
f	**1**	0	0	0	0	0

Adjacency Matrix

Graph represented as:

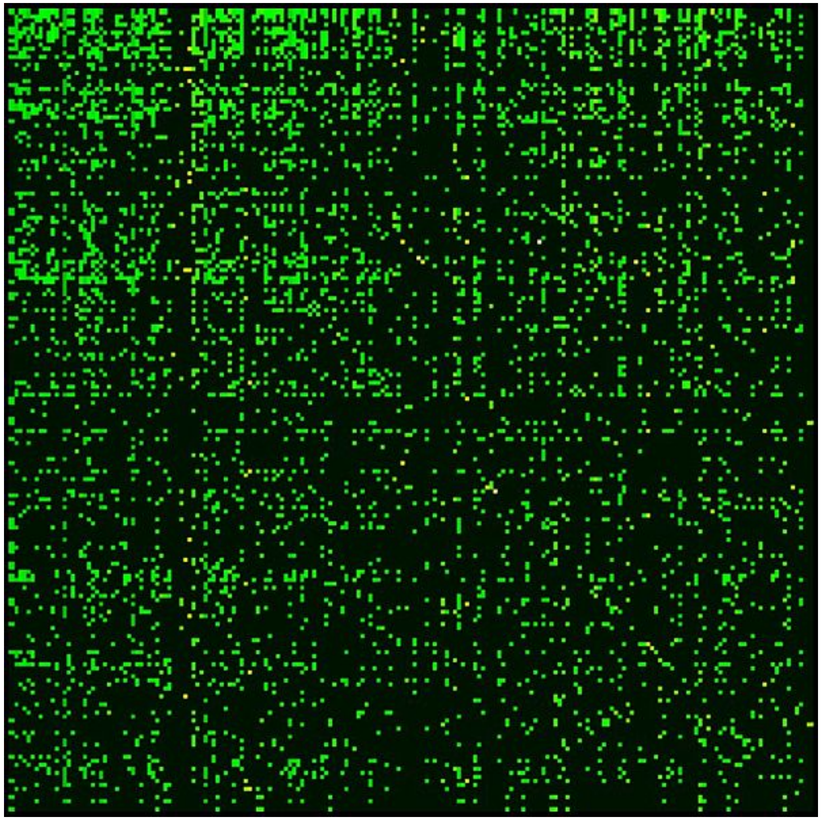
$A[v][w] = 1$ iff $(v,w) \in E$

Neat properties:

- A^2 = length 2 paths
- A^∞ = Google pagerank

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

A Google matrix is a particular stochastic matrix that is used by Google's PageRank algorithm. The matrix represents a graph with edges representing links between pages. The rank of each page can be generated iteratively from the Google matrix using the power method. However, in order for the power method to converge, the matrix must be stochastic, irreducible and aperiodic.



Explanation:
https://www.youtube.com/watch?v=bTI1aC-PYD8

Adjacency Matrix in C++

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

```
class Graph {

  boolean[][] m_adjMatrix;


}
```

	a	b	c	d
a	0	0	0	0
b	0	0	**1**	**1**
c	0	**1**	0	0
d	0	**1**	0	0
e	**1**	**1**	0	0
f	**1**	0	0	0

Adjacency Matrix in C++

Graph represented as:

A[v][w] = 1 iff $(v,w) \in E$

```
class Graph {

  Node[][] m_adjMatrix;


}
```

	a	b	c	d
a	0	0	0	0
b	0	0	1	1
c	0	1	0	0
d	0	1	0	0
e	1	1	0	0
f	1	0	0	0

Adjacency Matrix in C++

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

	a	b	c	d
a	0	0	0	0
b	0	0	**1**	**1**
c	0	**1**	0	0
d	0	**1**	0	0
e	**1**	**1**	0	0
f	**1**	0	0	0

```
class Graph {

  List<List<Boolean>> m_adjMatrix;


}
```

Resizable, but harder to use.

Trade-offs

Adjacency Matrix vs. List?

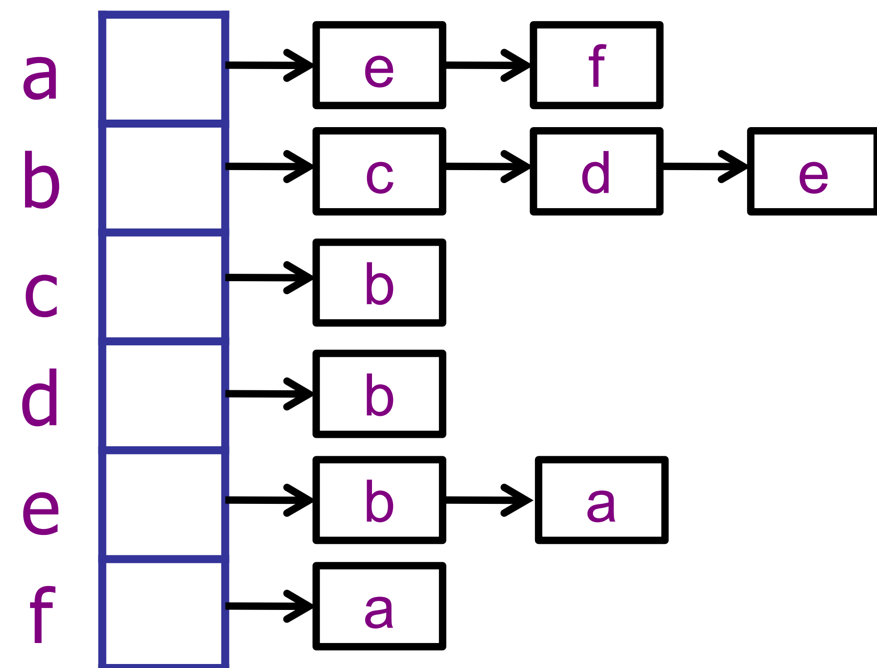
Adjacency List

Memory usage for graph $G = (V, E)$:

- array of size $|V|$
- linked lists of size $|E|$

Total: $O(V + E)$

For a cycle: $E = O(V)$



Adjacency Matrix

Memory usage for graph G = (V, E):

- array of size |V|*|V|

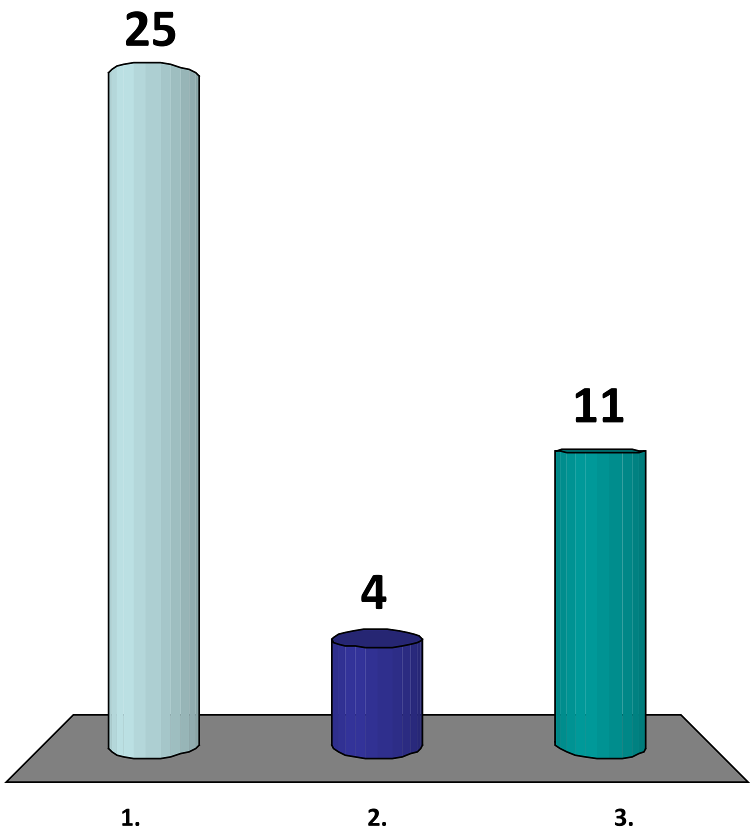
Total: $O(V^2)$

For a cycle: $O(V^2)$

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**1**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**1**	**1**	0	0	0	0
f	**1**	0	0	0	0	0

For a clique, which representation is better?

1. Adjacency matrix
2. Adjacency list
3. Equivalent



Adjacency List vs. Matrix

Memory usage for graph $G = (V, E)$:

- Adjacency List: $O(V + E)$
- Adjacency Matrix: $O(V^2)$

For a cycle: $O(V)$ vs. $O(V^2)$

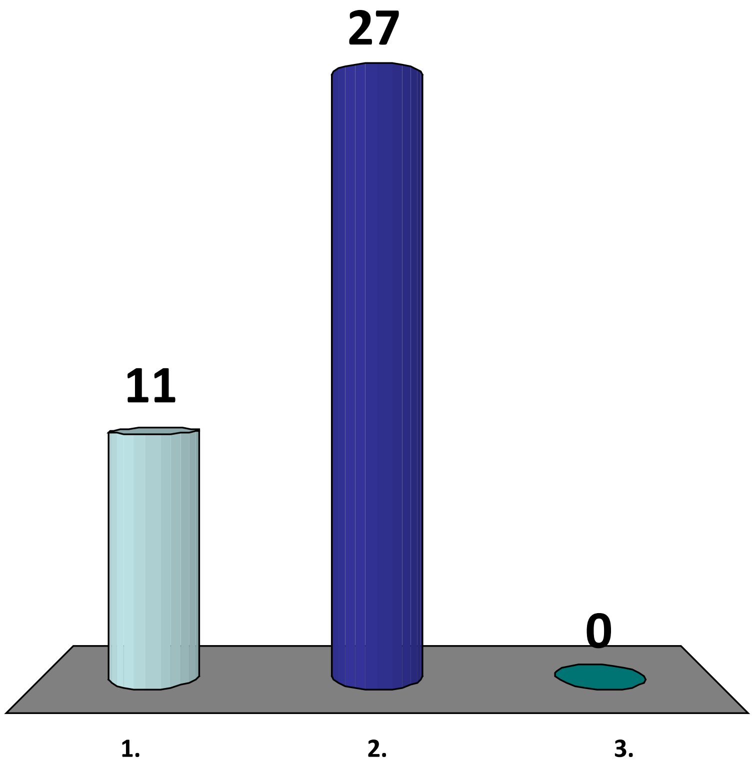
For a clique: $O(V + E) = O(V^2)$ vs. $O(V^2)$

Base rule: if graph is dense then use an adjacency matrix; else use an adjacency list.

dense: $|E| = \theta(V^2)$

Which representation for Facebook Graph? Query: Are Bob and Joe friends?

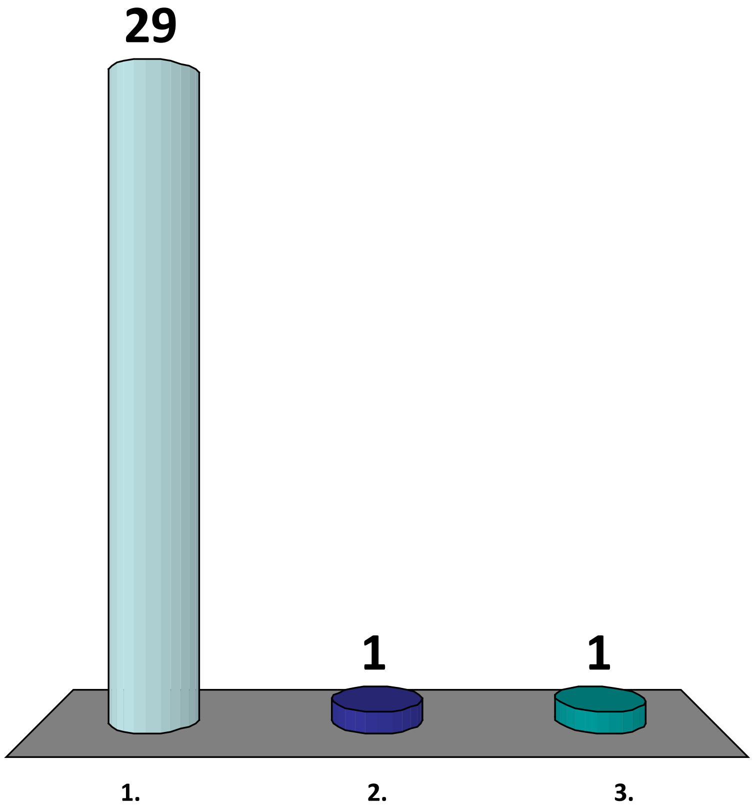
1. Adjacency List
2. Adjacency Matrix
3. Equivalent



List: (much) better space.
Matrix: somewhat faster

Which representation for Facebook Graph? Query: List all my friends?

1. Adjacency List
2. Adjacency Matrix
3. Equivalent



Trade-offs

Adjacency Matrix:

- Fast query: are v and w neighbors?
- Slow query: find me any neighbor of v.
- Slow query: enumerate all neighbors.

Adjacency List:

- Fast query: find me any neighbor.
- Fast query: enumerate all neighbors.
- Slower query: are v and w neighbors?

Graph Representations

Key questions to ask:

- Space usage: is graph dense or sparse?
- Queries: what type of queries do I need?
  - Enumerate neighbors?
  - Query relationship?

Roadmap

Today: Graph Basics

- **What is a graph?**
- **Modeling problems as graphs.**
- **Graph representations (list vs. matrix)**
- Searching graphs (DFS / BFS)

Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- Searching graphs (DFS / BFS)

Searching a Graph

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

  Or: visit **all** the nodes in the graph;

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

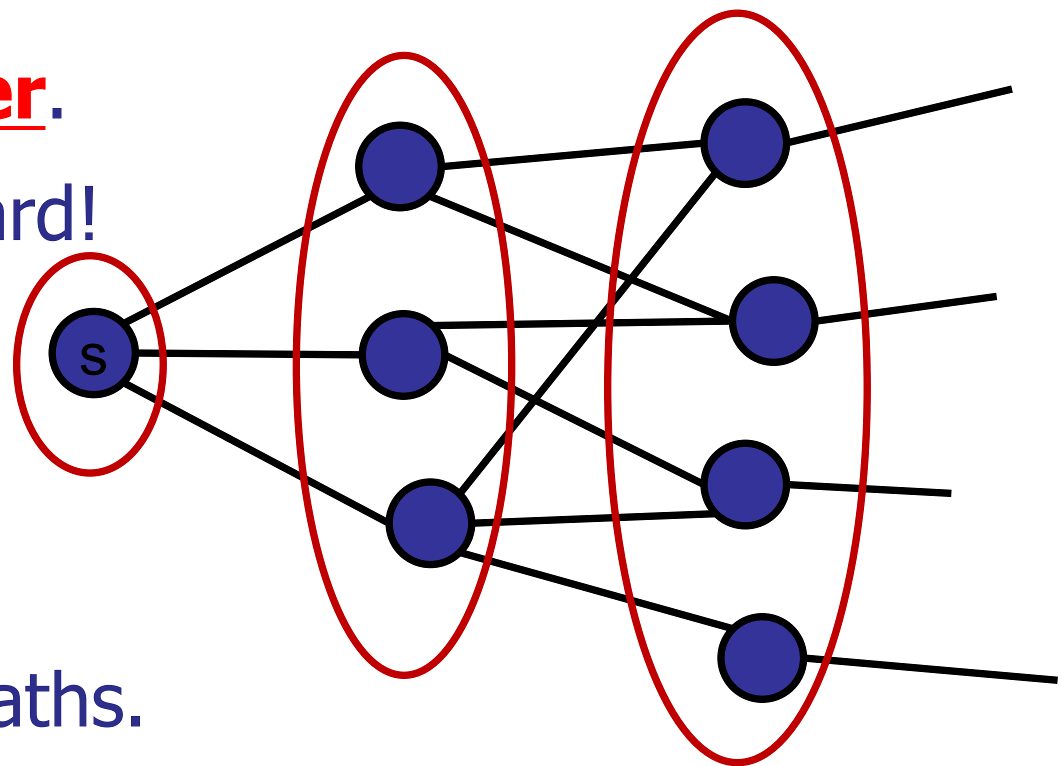
Graph representation:

- Adjacency list

Searching a graph

Breadth-First Search:

- Explore level by level
- Frontier: current level
- Initially: {s}
- Advance **frontier**.
- Don't go backward!

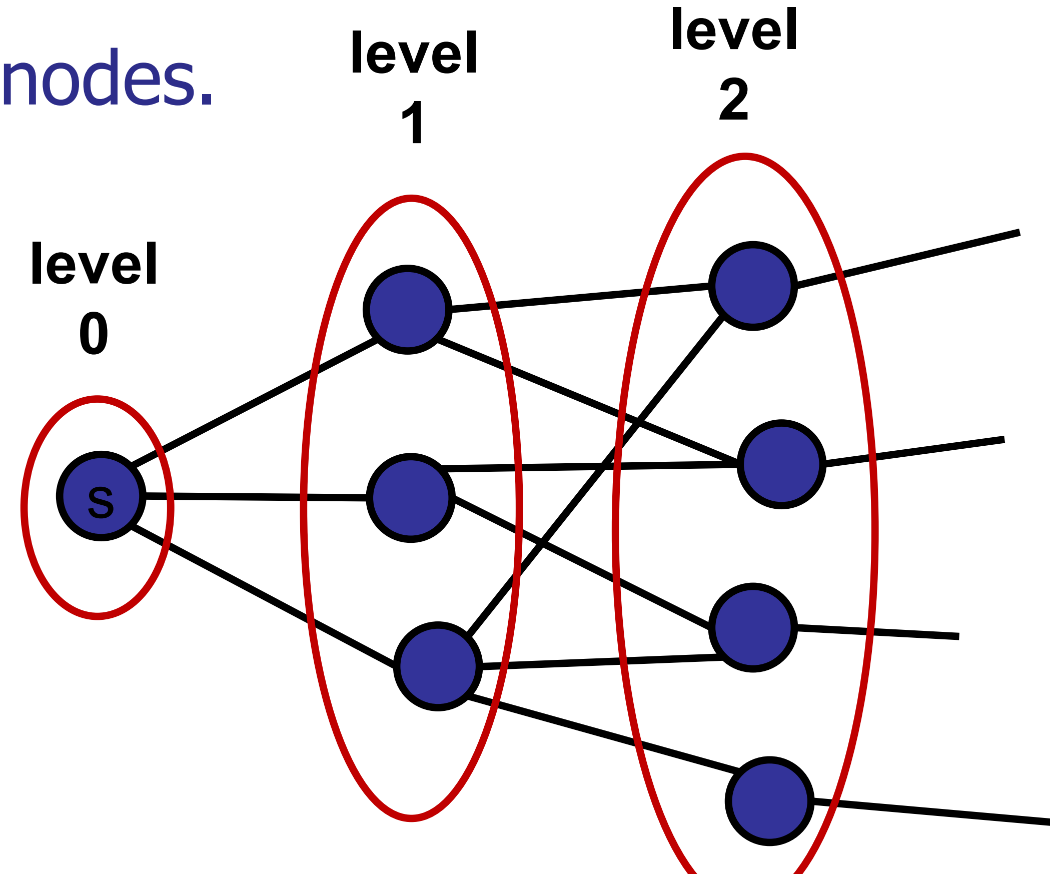


- Finds shortest paths.

Searching a graph

Breadth-First Search:

- Build levels.
- Calculate level[i] from level[i-1]
- Skip already visited nodes.



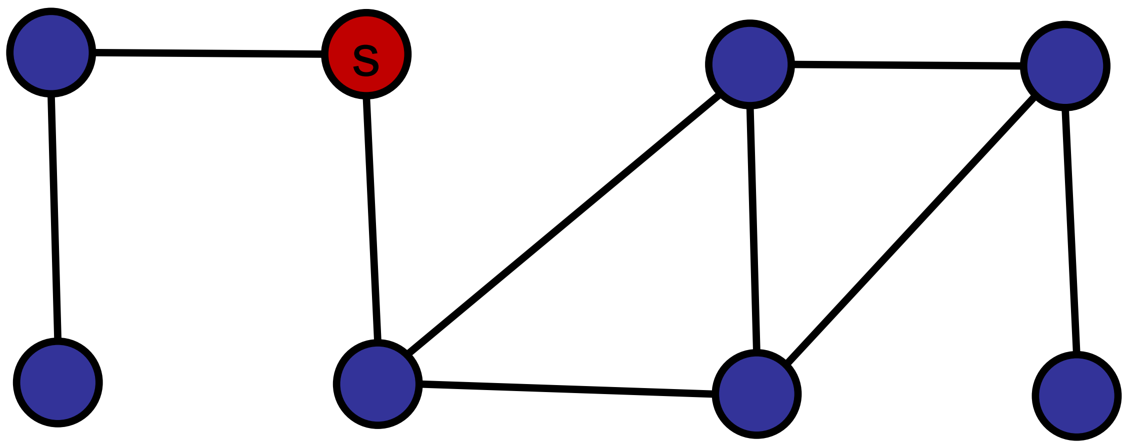
Breadth-First Search

```
BFS(Node[] nodeList, int startId) {
  boolean visited[numNode] = {0};

  int parent[numNode];

  for (int i=0;i<numNode;i++)
     parent[i] = -1; // no parent yet

  Set<int> frontier;
  frontier.insert(startId);
  visited[startId] = true;
  // Main code goes here!

}
```

Breadth-First Search

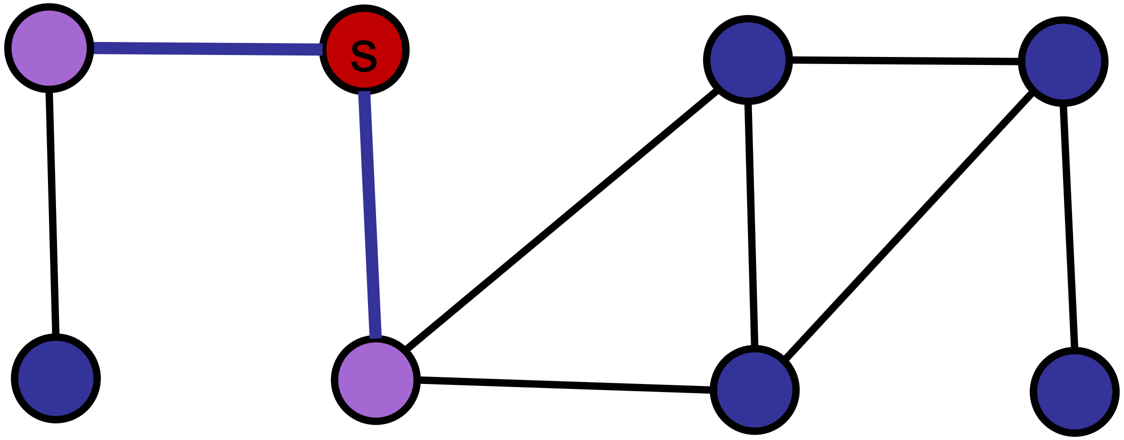
```
while (!frontier.isEmpty()){
    Set<int> nextFrontier = new Set<Integer>;
     while (!frontier.isEmpty()){
          extract a vertex v from frontier
          for (w = every neighbor of v) {
                if (!visited[w]) {
                      visited[w] = true;
                      parent[w] = v;
                      nextFrontier.add(w);
                }
          }
    }
    frontier = nextFrontier;
}
```

Breadth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Breadth-First Search Example



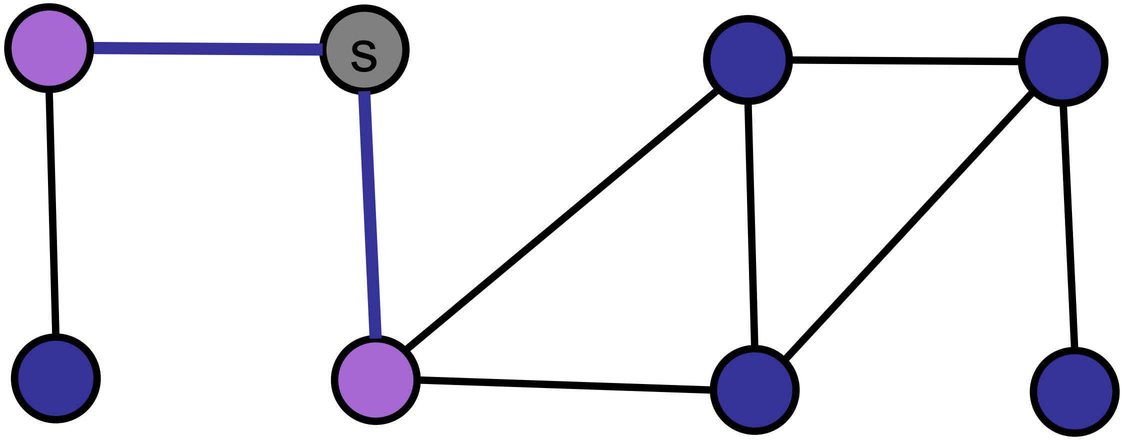
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

Breadth-First Search Example



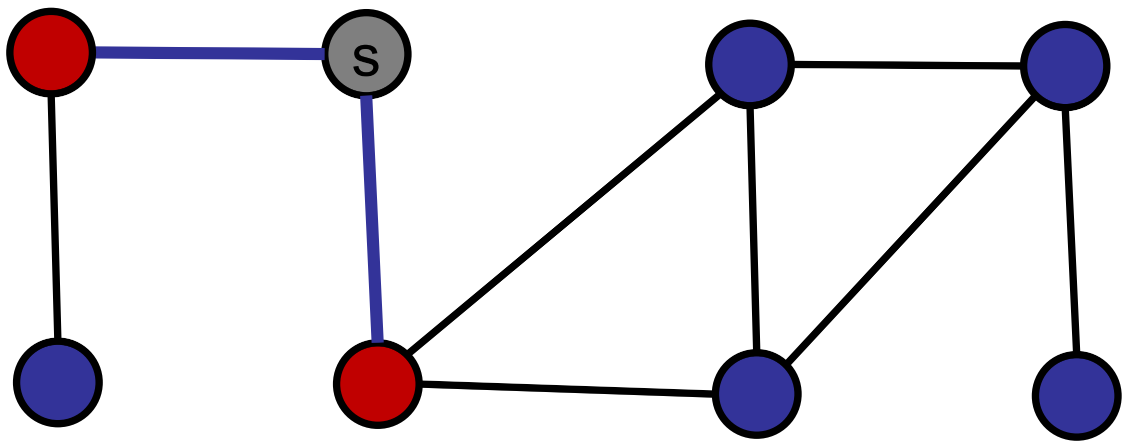
Red = active frontier

Purple = next

Gray = visited

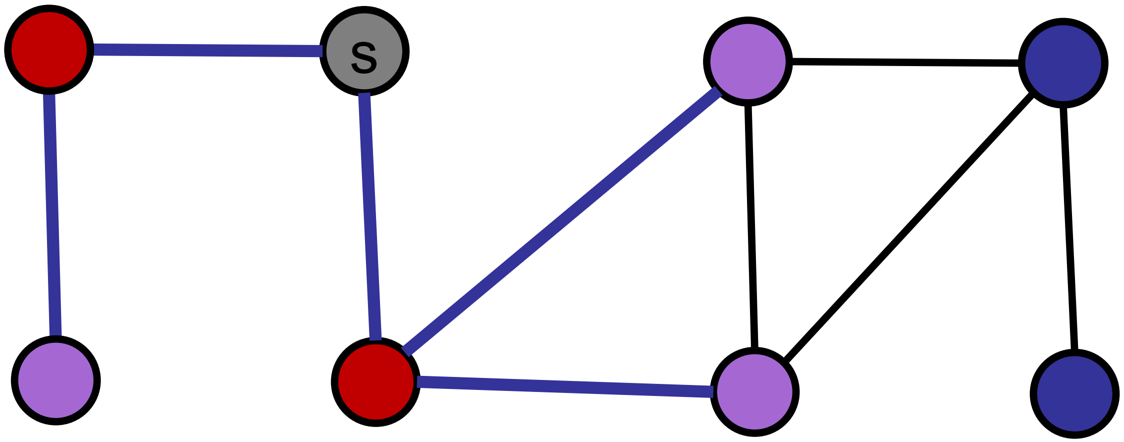
Blue = unvisited

Breadth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Breadth-First Search Example



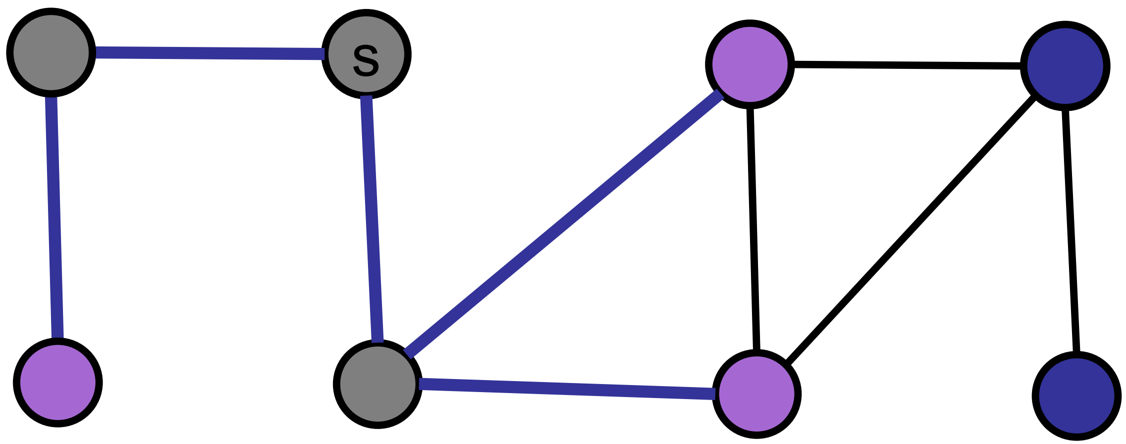
Red = active frontier

Purple = next

Gray = visited

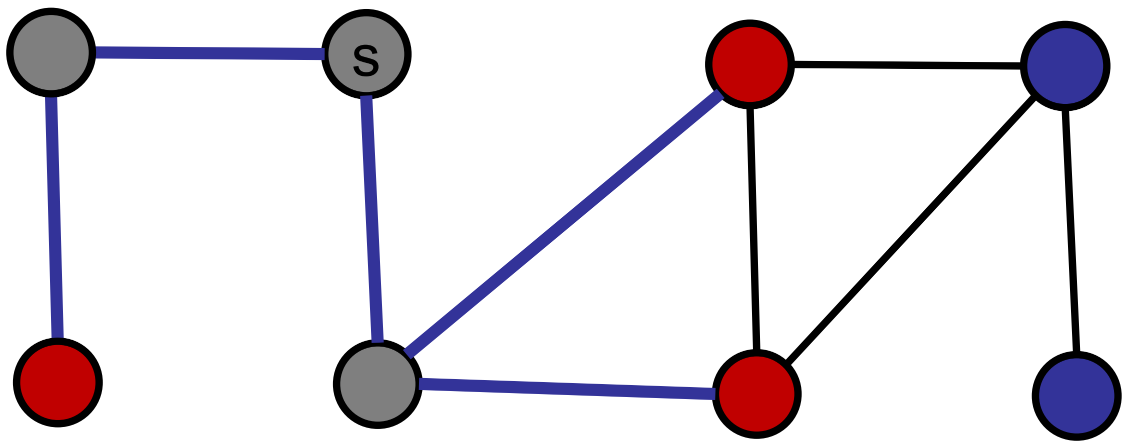
Blue = unvisited

Breadth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Breadth-First Search Example



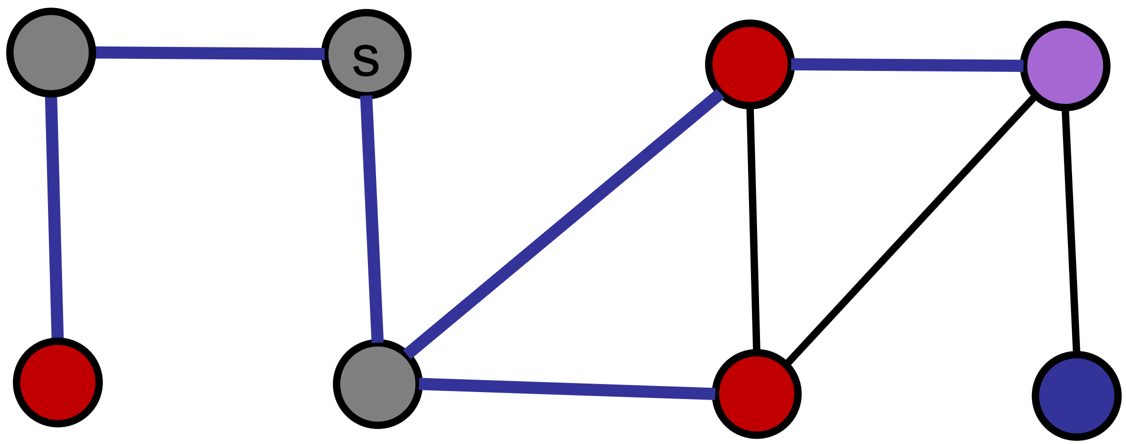
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

Breadth-First Search Example



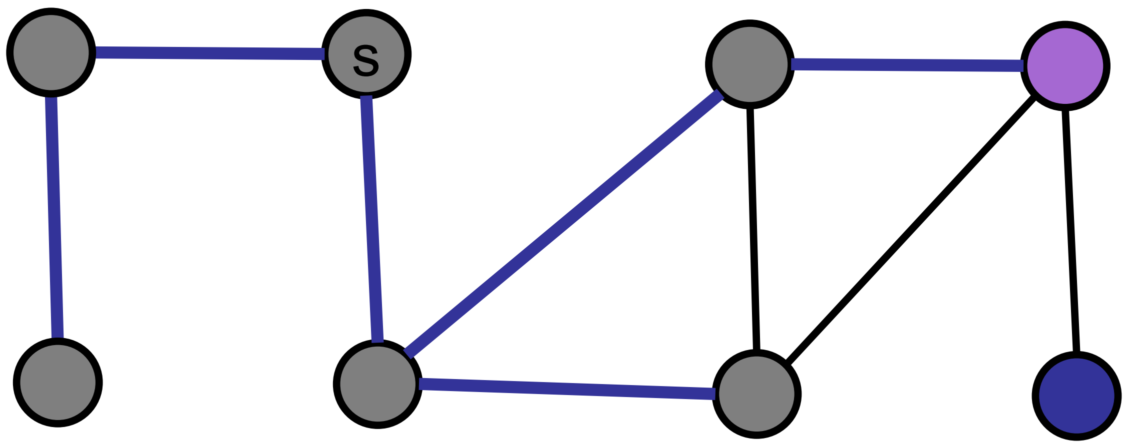
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

Breadth-First Search Example



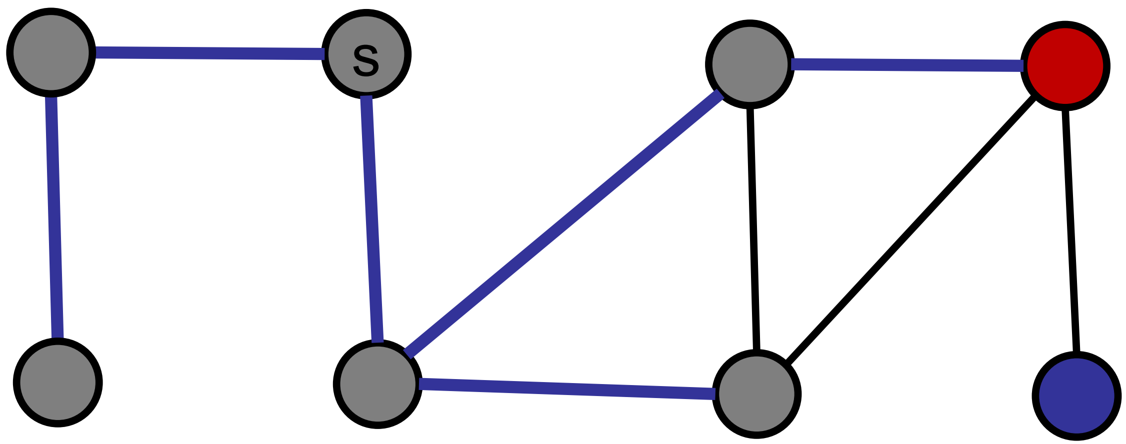
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

Breadth-First Search Example



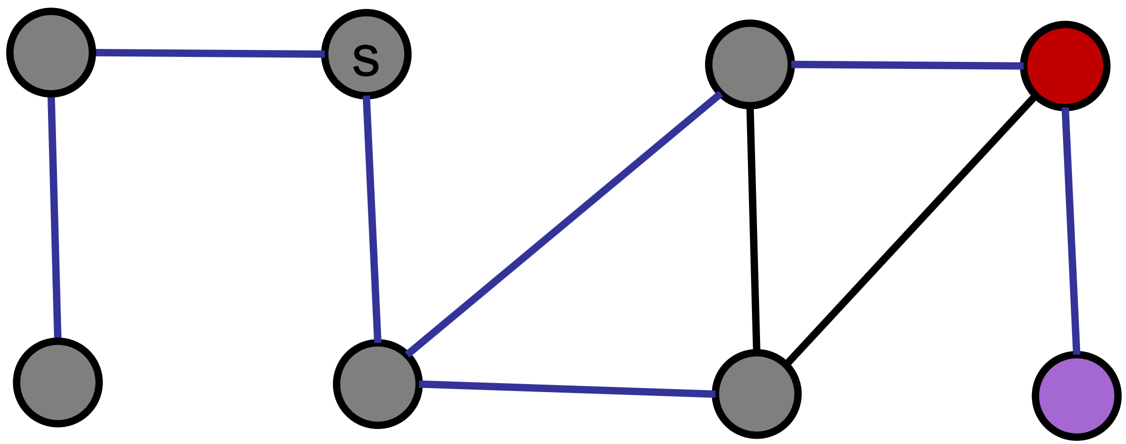
Red = active frontier

Purple = next

Gray = visited

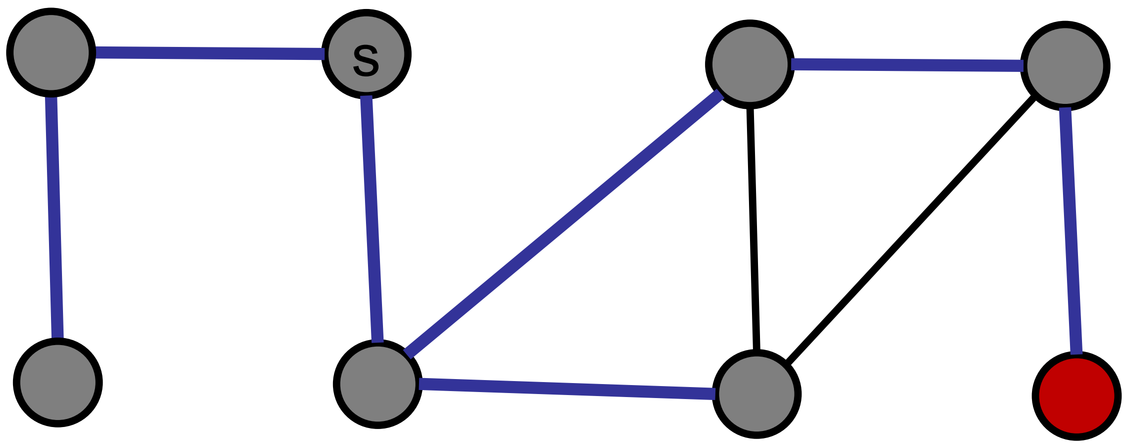
Blue = unvisited

Breadth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Breadth-First Search Example



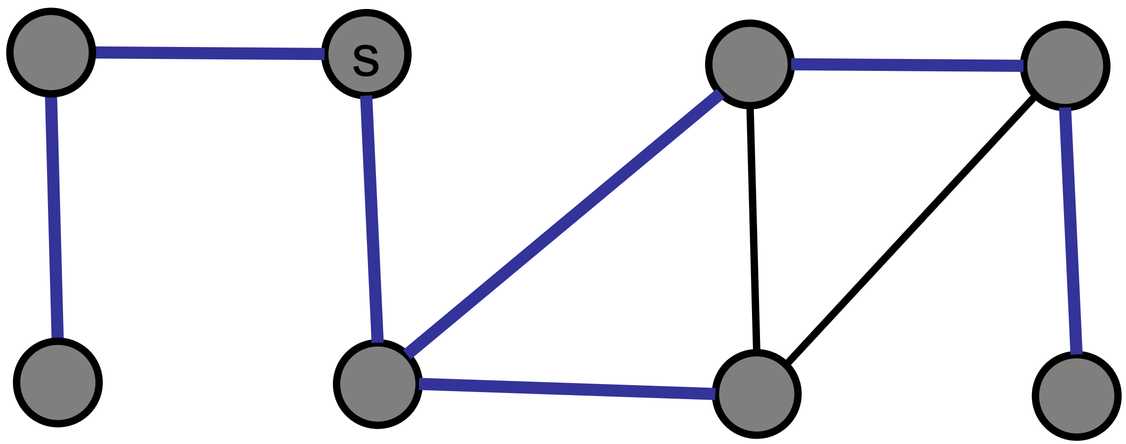
Red = active frontier

Purple = next

Gray = visited

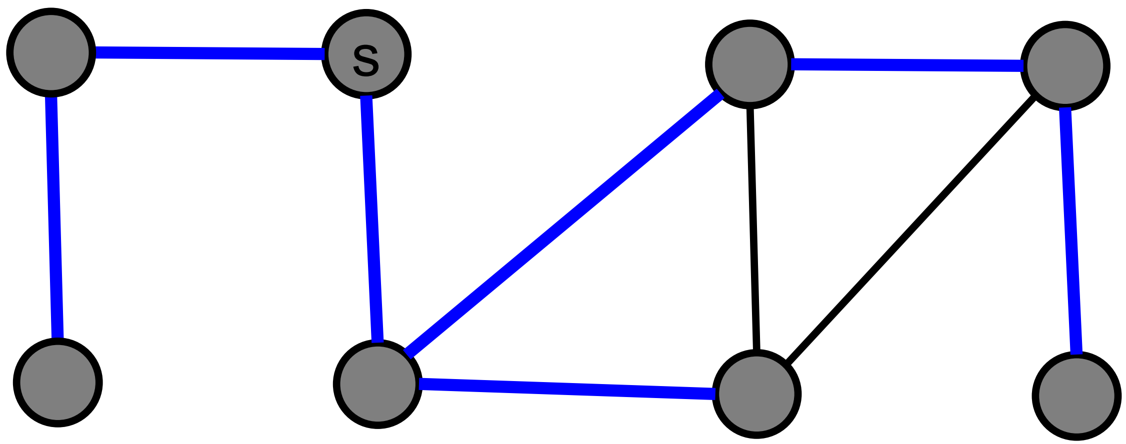
Blue = unvisited

Breadth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Breadth-First Search Example



Red = active frontier

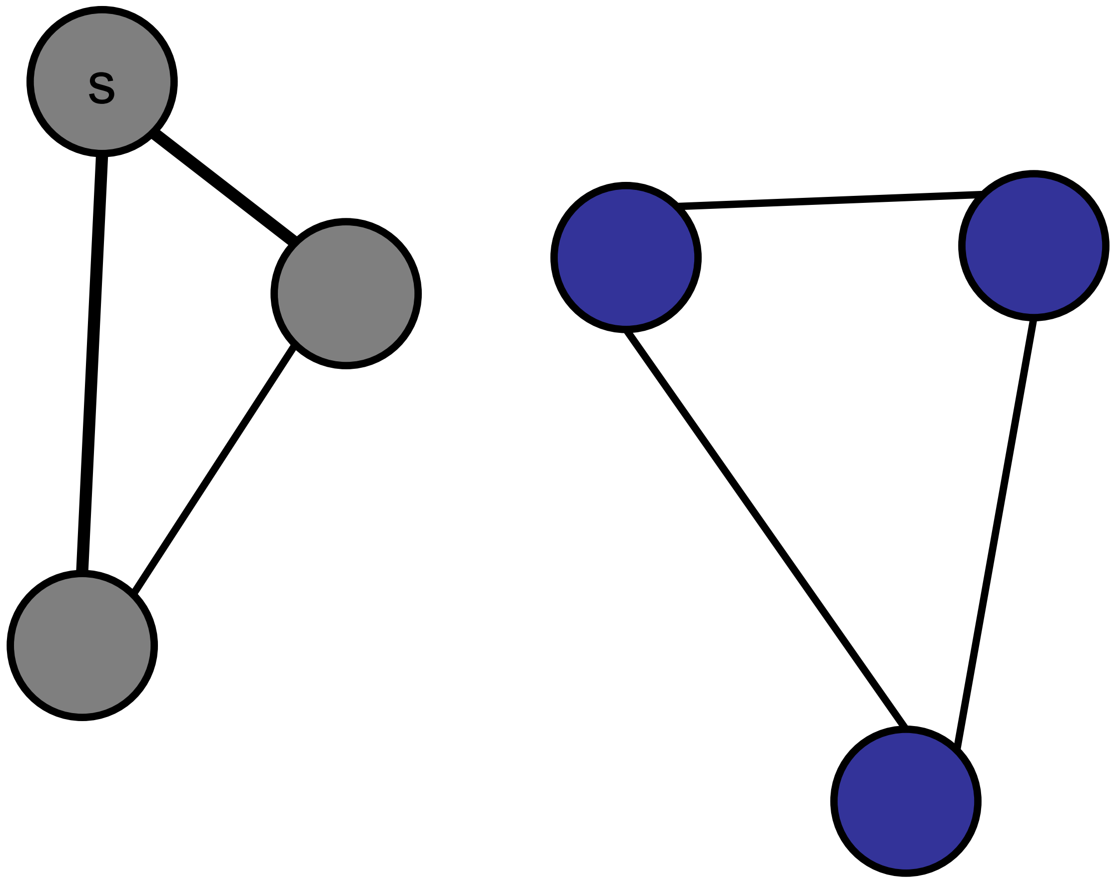
Purple = next

Gray = visited

Blue = unvisited

BFS on Disconnected Graph

Example:



Breadth-First Search

```
BFS(Node[] nodeList) {
  boolean visited[numNode] = {0};
  int parent[numNode];

  for (int i=0;i<numNode;i++)
     parent[i] = -1; // no parent yet

  for (int start = 0; start < numNode; start++) {
     if (!visited[start]){
           Bag<Integer> frontier = new Bag<Integer>;
           frontier.add(start);
           visited[start] = true;
           // Main code goes here!
     }
  }
```

Breadth-First Search

Analysis:

- Vertex v = "start" once. ← O(V)
- Vertex v added to nextFrontier (and frontier) once.
  - After visited, never re-added.
- Each v.nbrlist is enumerated once.
  - When v is removed from frontier.

O(E)

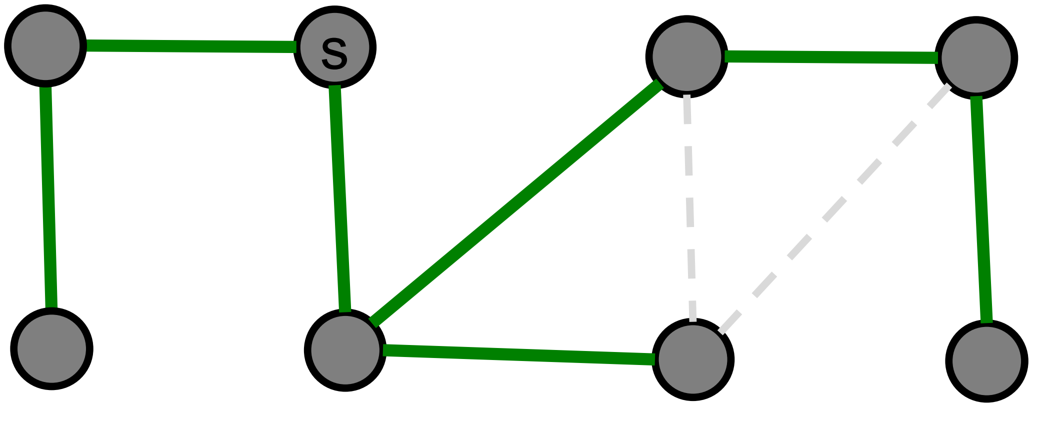
Breadth-First Search

```
while (!frontier.isEmpty()){
    Set<int> nextFrontier = new Set<Integer>;
     while (!frontier.isEmpty()){
          extract a vertex v from frontier
          for (w = every neighbor of v) {
                if (!visited[w]) {
                      visited[w] = true;
                      parent[w] = v;
                      nextFrontier.add(w);
                }
          }
    }
    frontier = nextFrontier;
}
```

Breadth-First Search

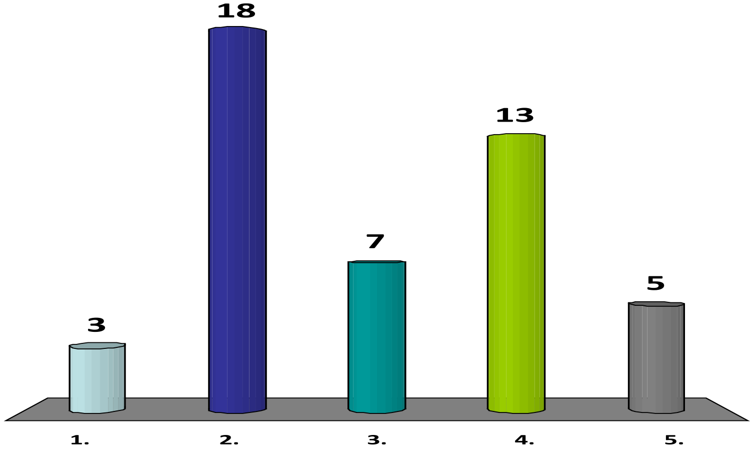
Shortest paths from the start node:

– Parent pointers store shortest path.



Which is true? (More than one may apply.)

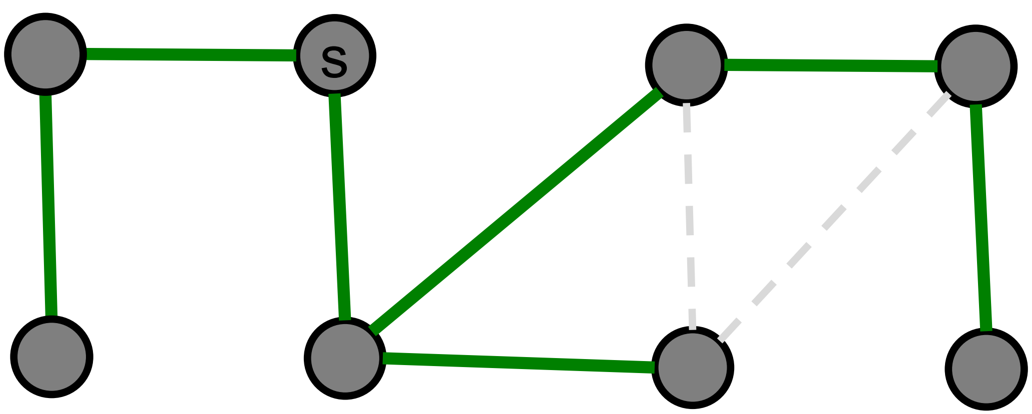
1. Shortest path graph is a cycle.
2. Shortest path graph is a tree.
3. Shortest path graph has low-degree.
4. Shortest path graph has low diameter.
5. None of the above.



Breadth-First Search

Shortest paths:

- Parent pointers store shortest path.
- Shortest path is a tree.
- (Possibly high degree; possibly high diameter.)



What if there are two components?

Searching a Graph

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

  Or: visit **all** the nodes in the graph;

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

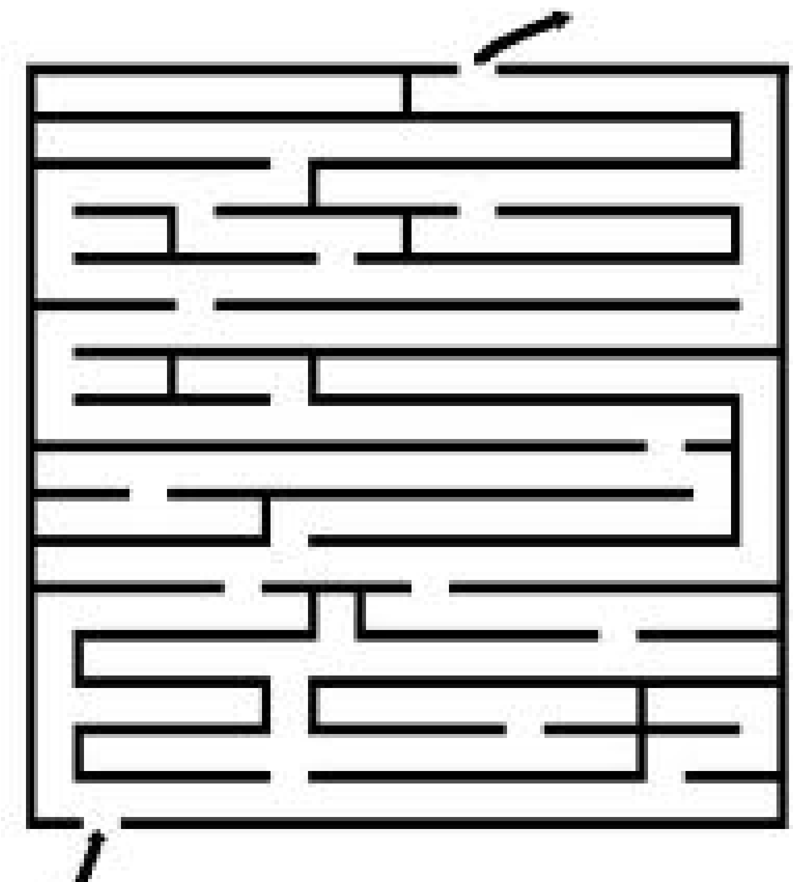
Graph representation:

- Adjacency list

Depth-First Search

Exploring a maze:

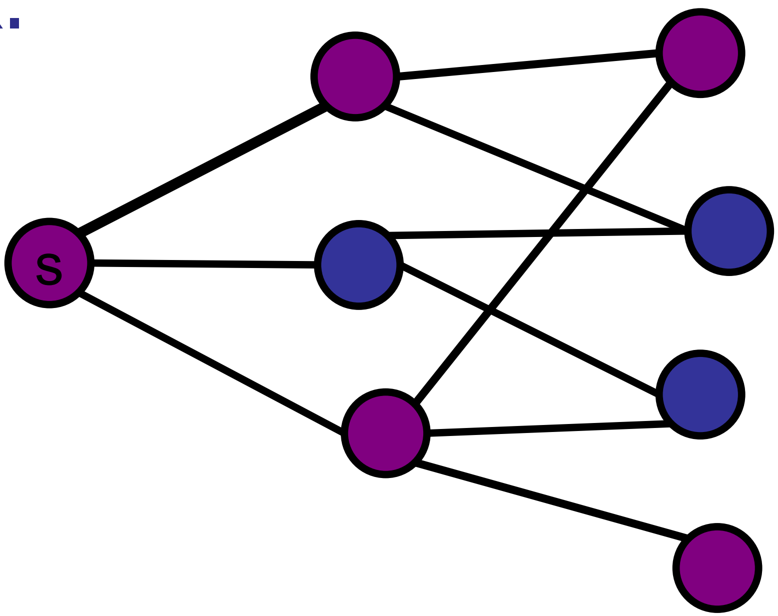
- Follow path until stuck.
- Backtrack along breadcrumbs until reach unexplored neighbor.
- Recursively explore.



Searching a graph

Depth-First Search:

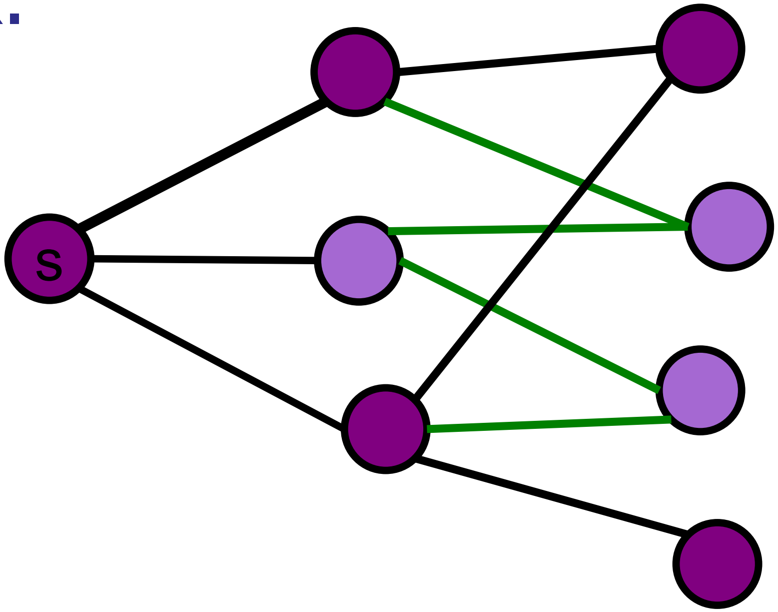
- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don't repeat a vertex.



Searching a graph

Depth-First Search:

- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don’t repeat a vertex.



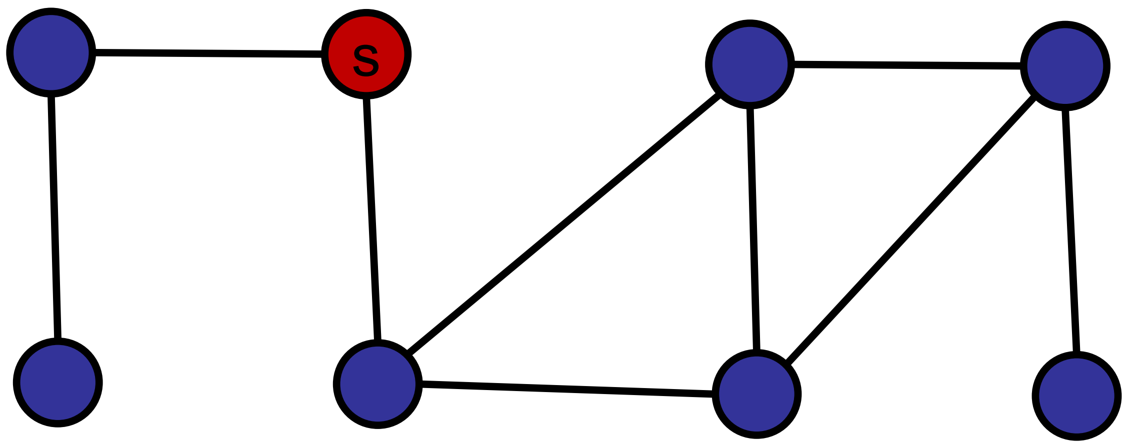
Depth-First Search

```
DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for every neighbor v of startId {
     if (!visited[v]){
           visited[v] = true;
           DFS-visit(nodeList, visited, v);
     }
  }
}
```

Depth-First Search

```
DFS(Node[] nodeList){
  boolean visited [numNode] = {0};

  for (start = 0; start<nodeList.length; start++) {
     if (!visited[start]){
           visited[start] = true;
           DFS-visit(nodeList, visited, start);
     }
  }
}
```

Depth-First Search Example



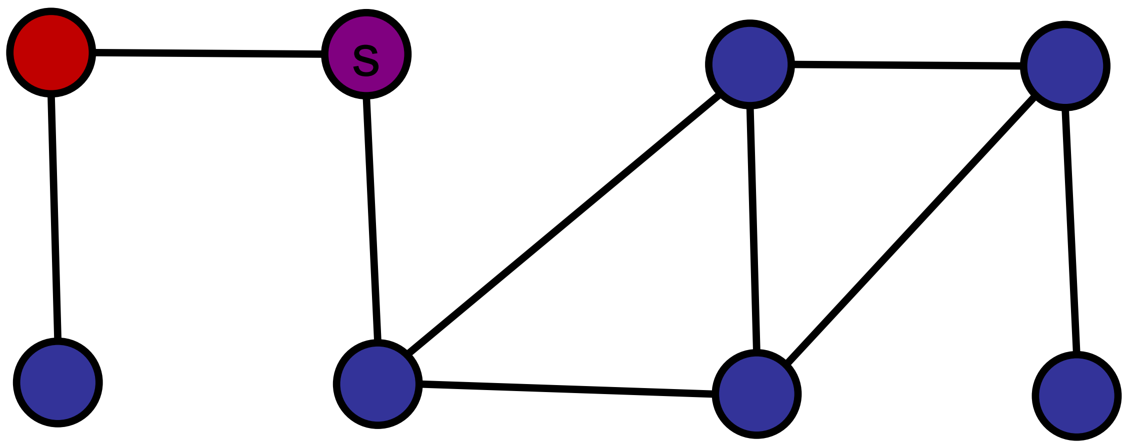
Red = active frontier

Purple = next

Gray = visited

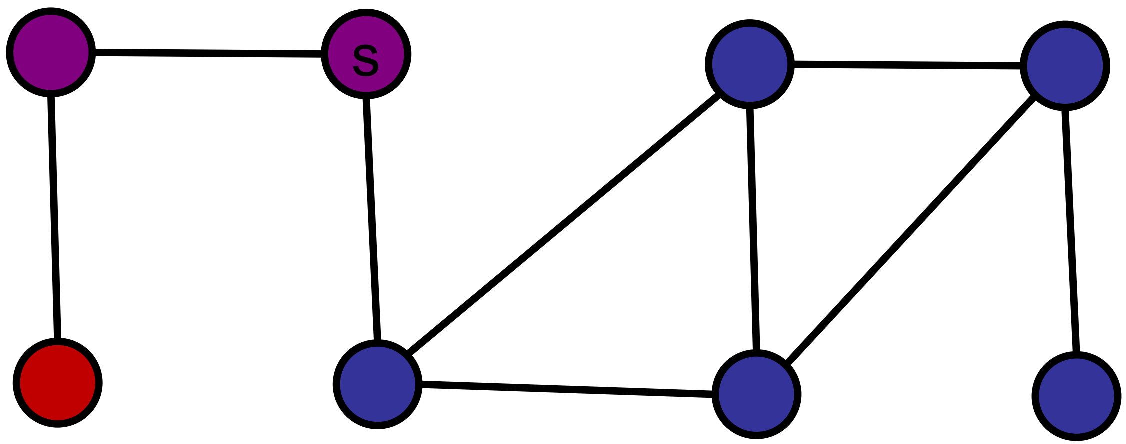
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



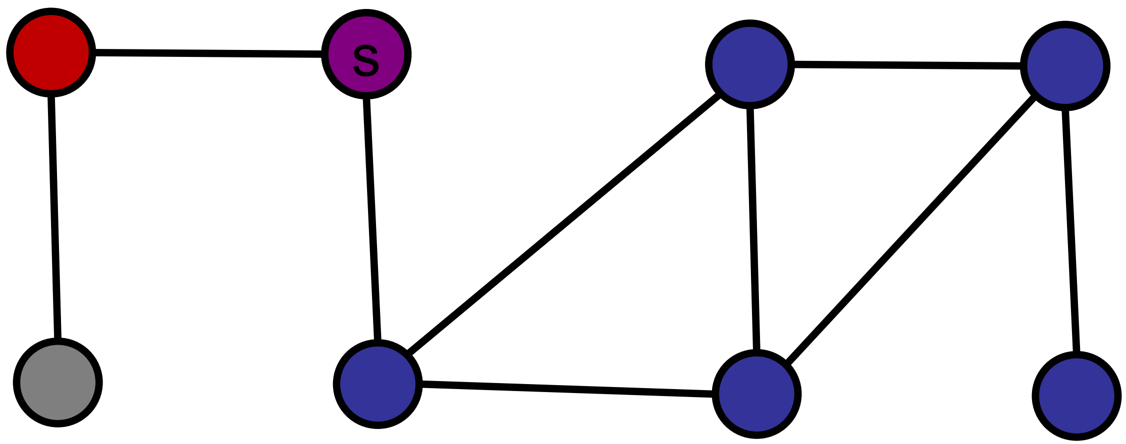
Red = active frontier

Purple = next

Gray = visited

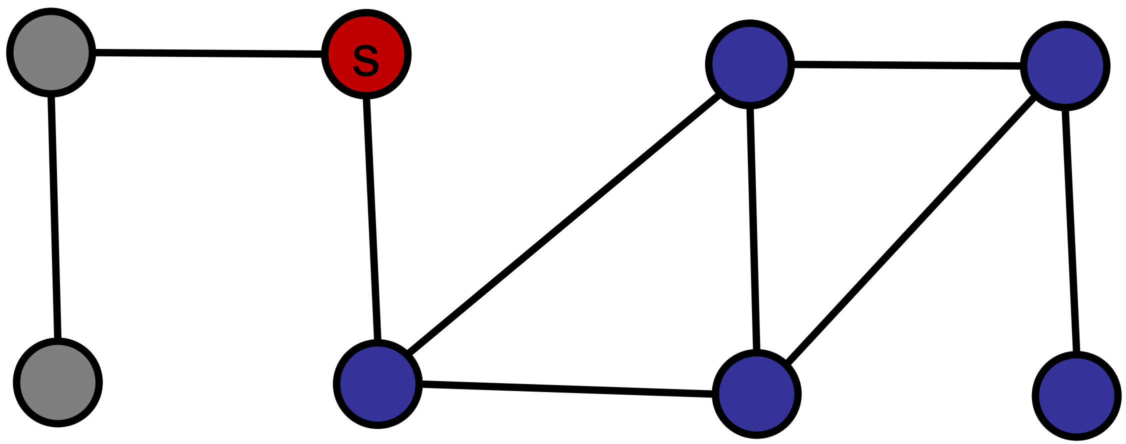
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



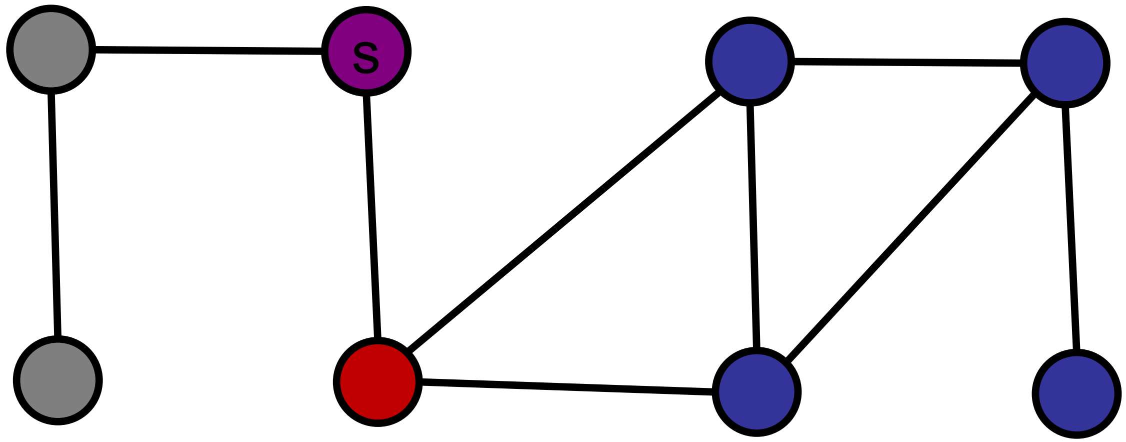
Red = active frontier

Purple = next

Gray = visited

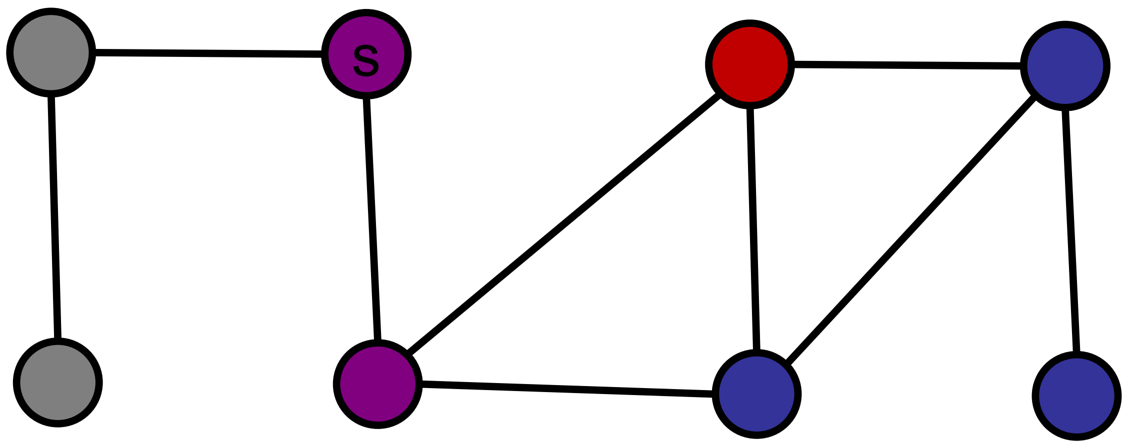
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



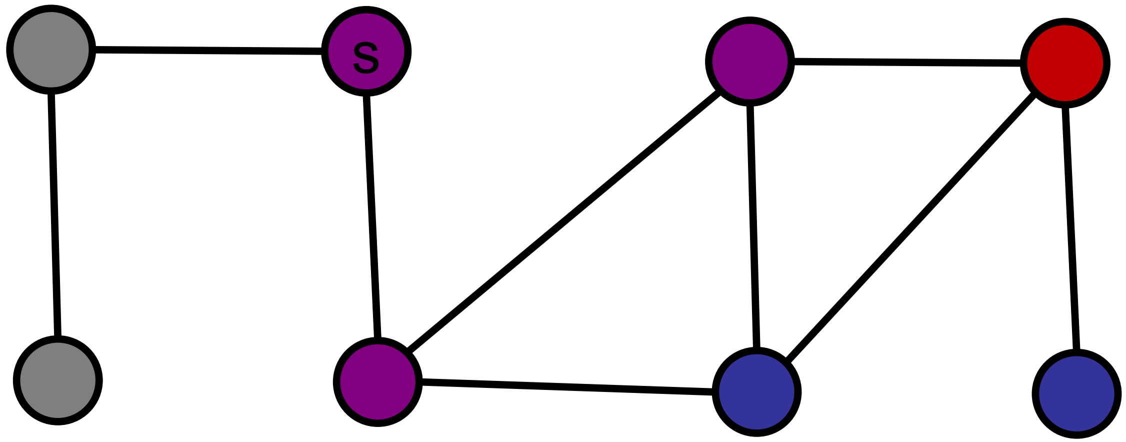
Red = active frontier

Purple = next

Gray = visited

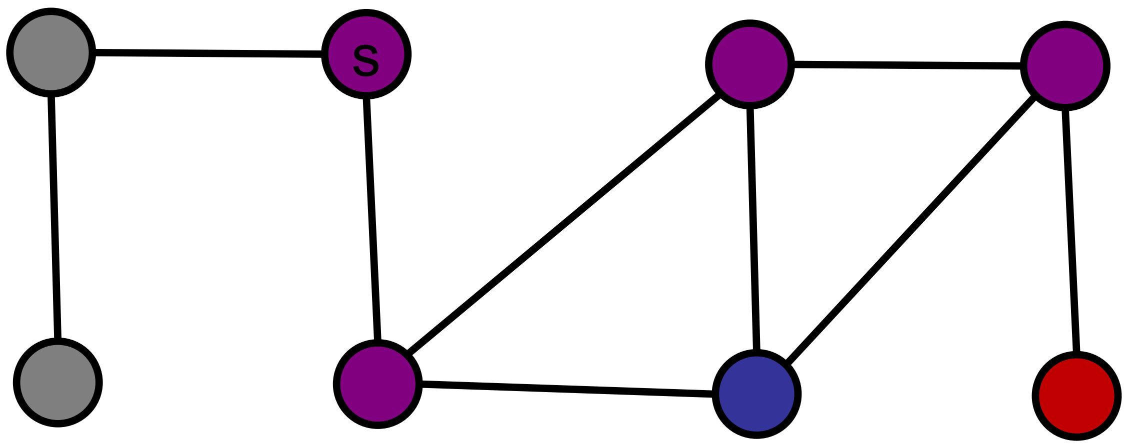
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



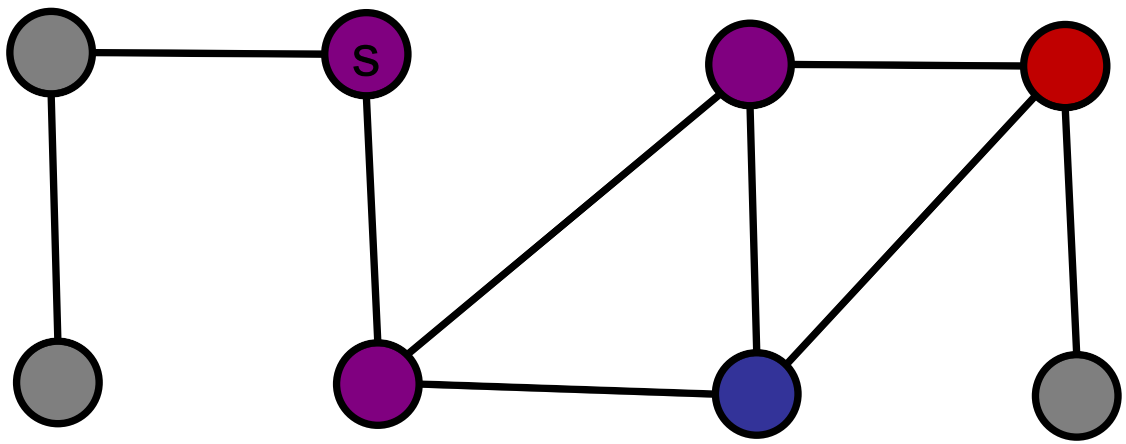
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

Depth-First Search Example



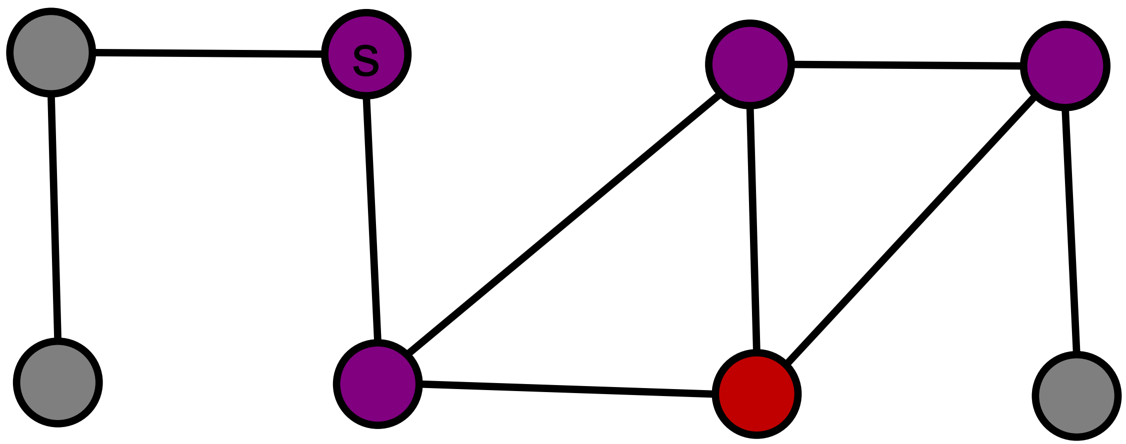
Red = active frontier

Purple = next

Gray = visited

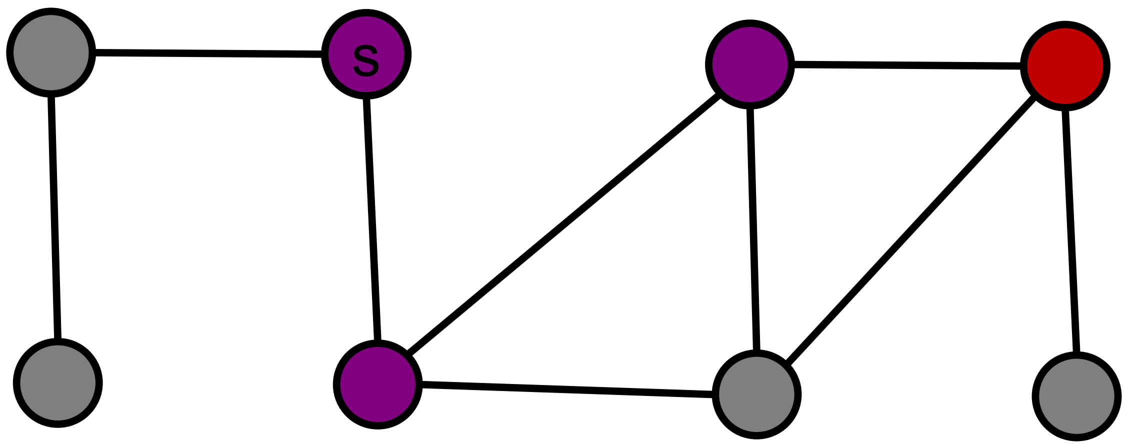
Blue = unvisited

Depth-First Search Example



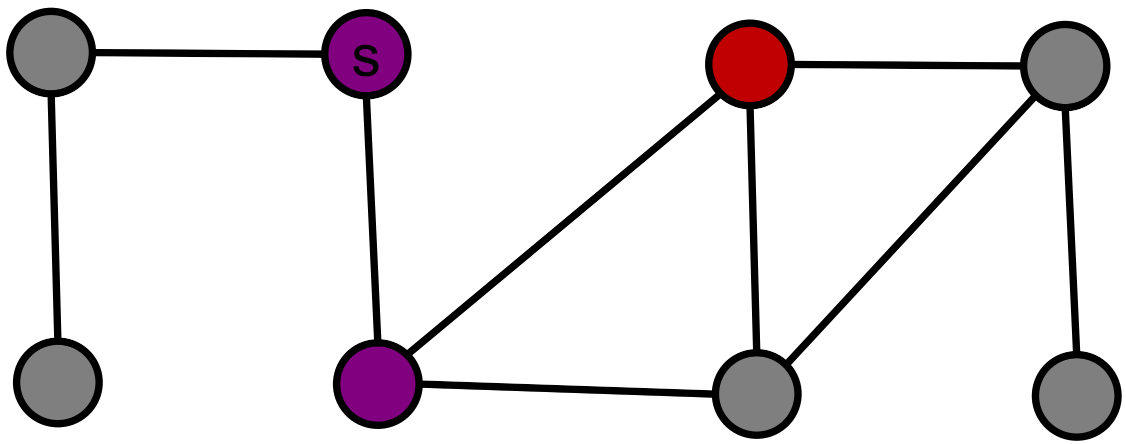
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



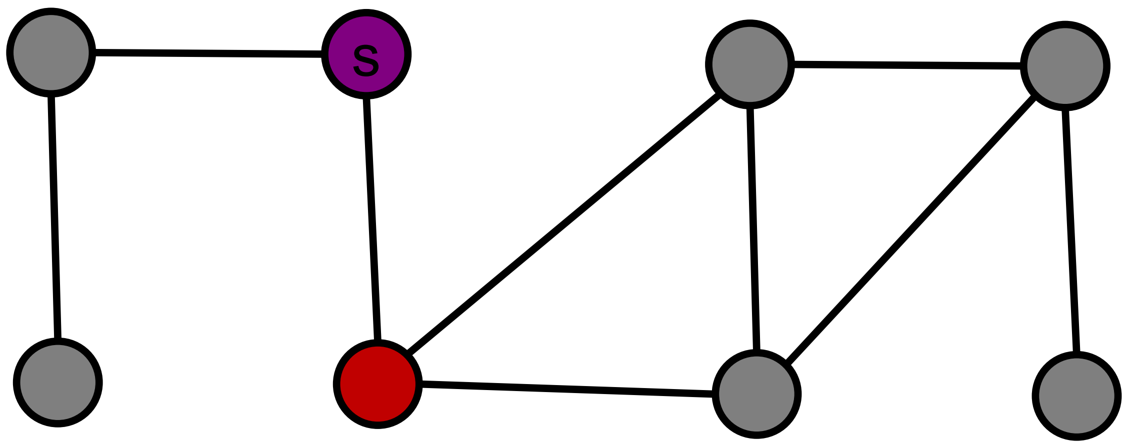
Red = active frontier

Purple = next

Gray = visited

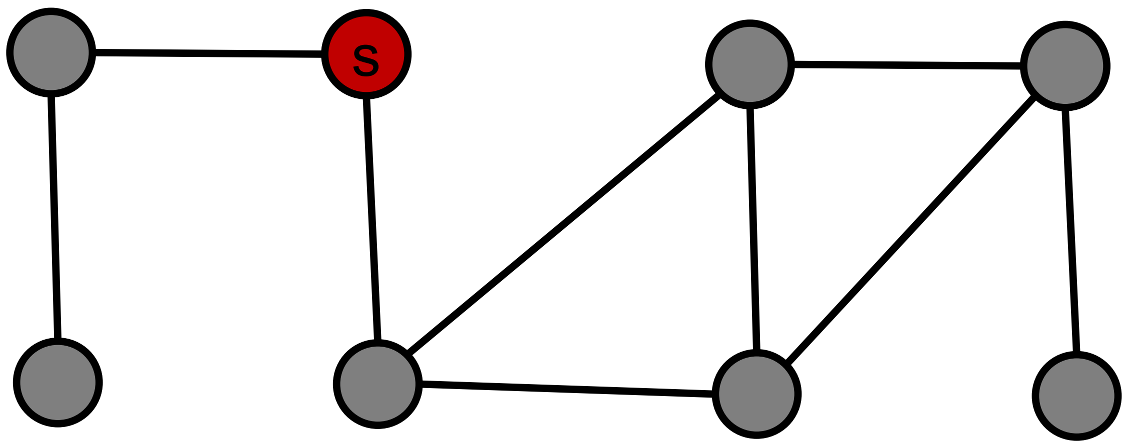
Blue = unvisited

Depth-First Search Example



Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Depth-First Search Example



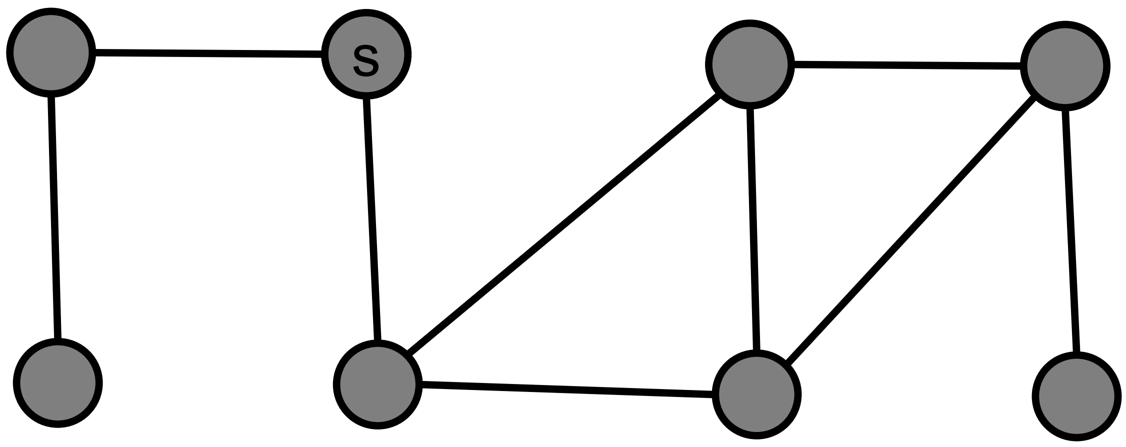
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

Depth-First Search Example



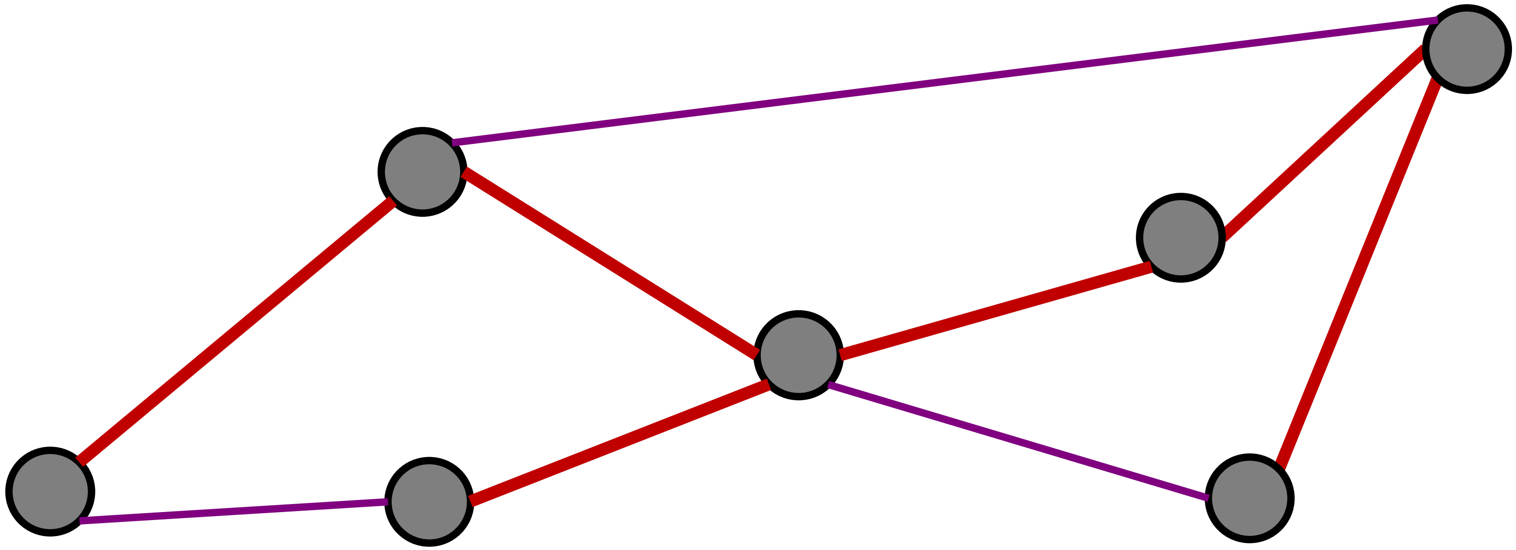
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

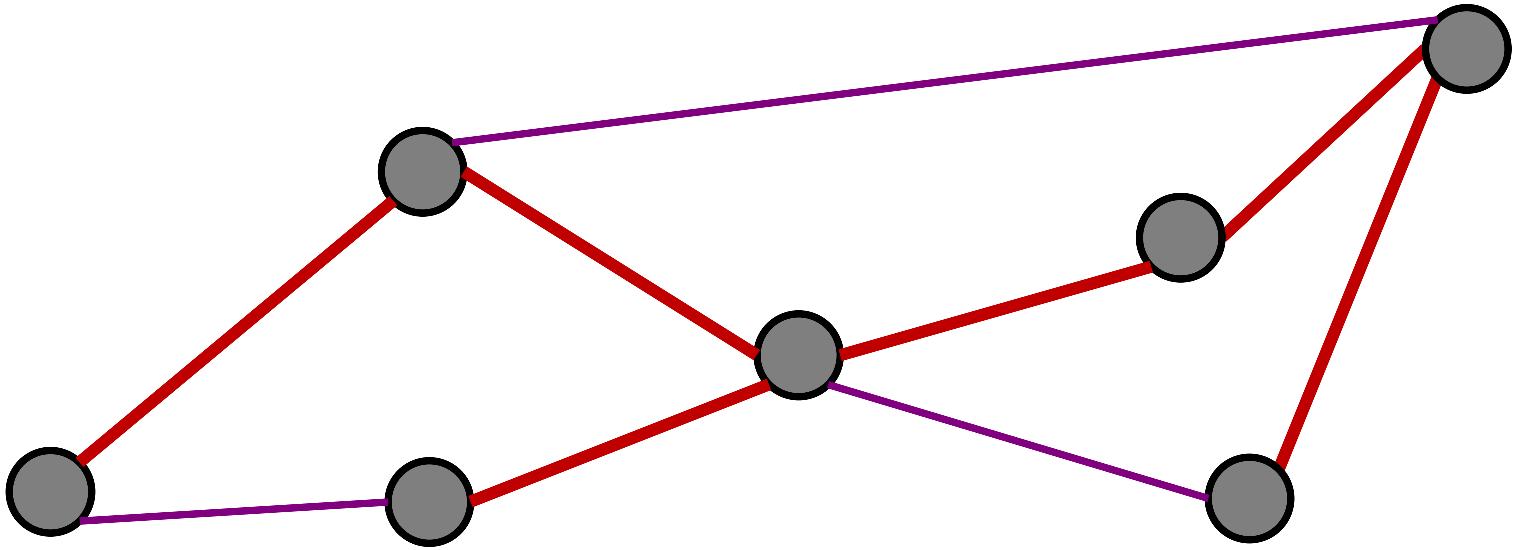
DFS parent edges



Red = Parent Edges

Purple = Non-parent edges

DFS parent edges = tree



Red = Parent Edges

Purple = Non-parent edges

Note: not shortest paths!

Depth-First Search

Analysis:

O(V)

- DFS-visit called only once per node.
  - After visited, never call DFS-visit again.

- In DFS-visit, each neighbor is enumerated.

O(E)

Graph Search

BFS and DFS are the same algorithm:

- BFS: use a queue
  - Every time you visit a node, add all unvisited neighbors to the queue.
- DFS: use a stack
  - Every time you visit a node, add all unvisited neighbors to the stack.

Graph Search

Breadth-first search:

Same algorithm, implemented with a queue:

Add start-node to queue.

Repeat until queue is empty:

- Remove node v from the front of the queue.
- Visit v.
- Explore all outgoing edges of v.
- Add all unvisited neighbors of v to the queue.

Graph Search

Depth-first search:

Same algorithm, implemented with a stack:

Add start-node to stack.

Repeat until stack is empty:

- Pop node v from the top of the stack.
- Visit v.
- Explore all outgoing edges of v.
- Push all unvisited neighbors of v on the top of the stack.

Review: Searching Graphs

BFS and DFS are the same algorithm:

- BFS: use a queue
  - Every time you visit a node, add all unvisited neighbors to the queue.
- DFS: use a stack
  - Every time you visit a node, add all unvisited neighbors to the stack.

What is a directed graph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.
- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Each edge is unique.
  - Each edge is **directed**.

What is a directed graph?

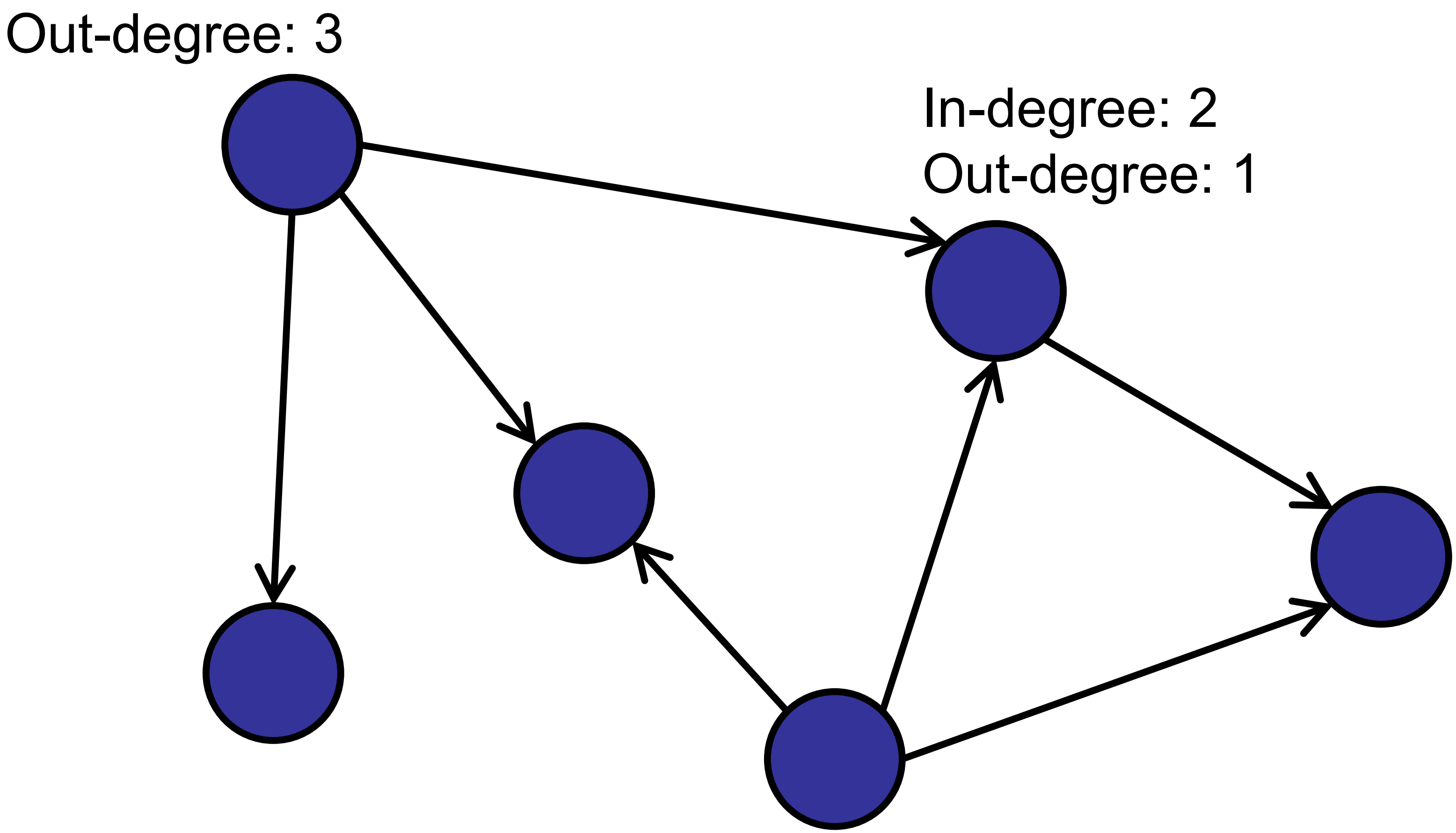
Graph $G = \langle V, E \rangle$

- V is a set of nodes
  - At least one: $|V| > 0$.
- E is a set of edges:
  - $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
  - $e = (v,w)$ Order matters!
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

What is a directed graph?

In-degree: number of incoming edges

Out-degree: number of outgoing edges



Representing a (Directed) Graph

Adjacency List:

- Array of nodes
- Each node maintains a list of neighbors
- Space: $O(V + E)$

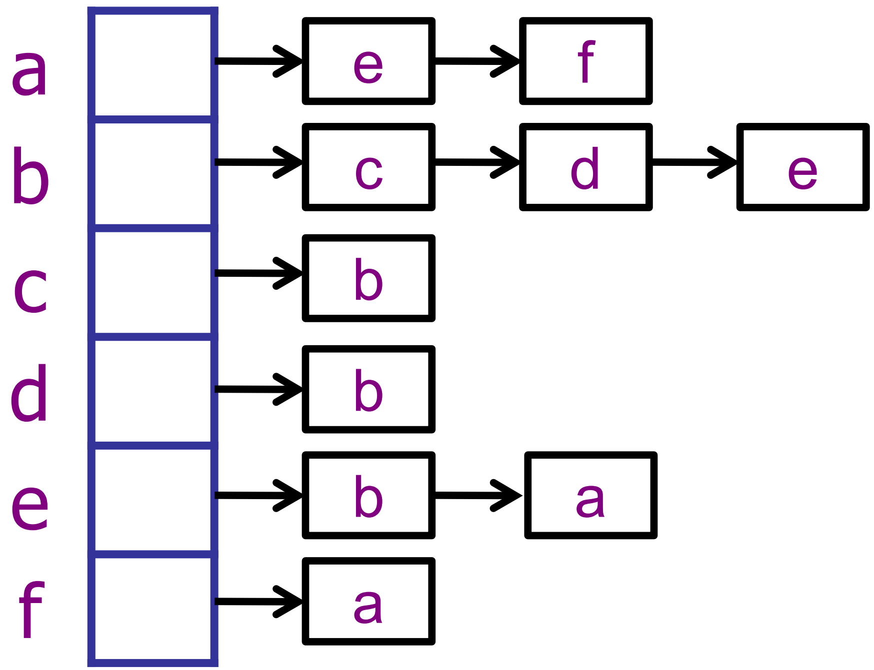
Adjacency Matrix:

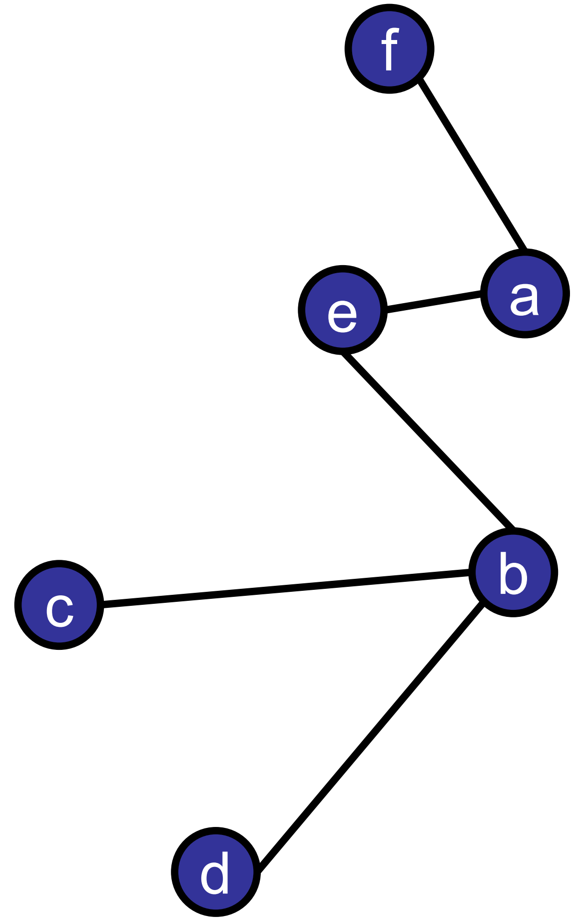
- Matrix $A[v,w]$ represents edge (v,w)
- Space: $O(V^2)$

Adjacency List

Undirected Graph consists of:

- Nodes: stored in an array
- Edges: linked list per node

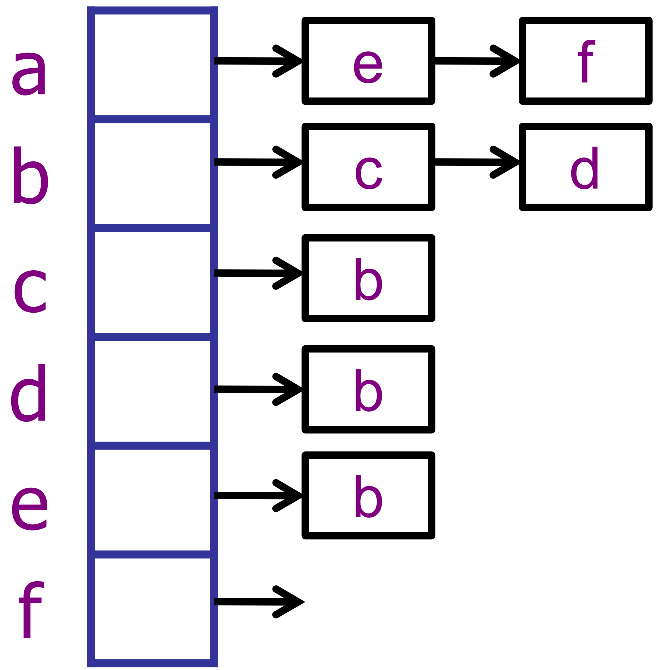


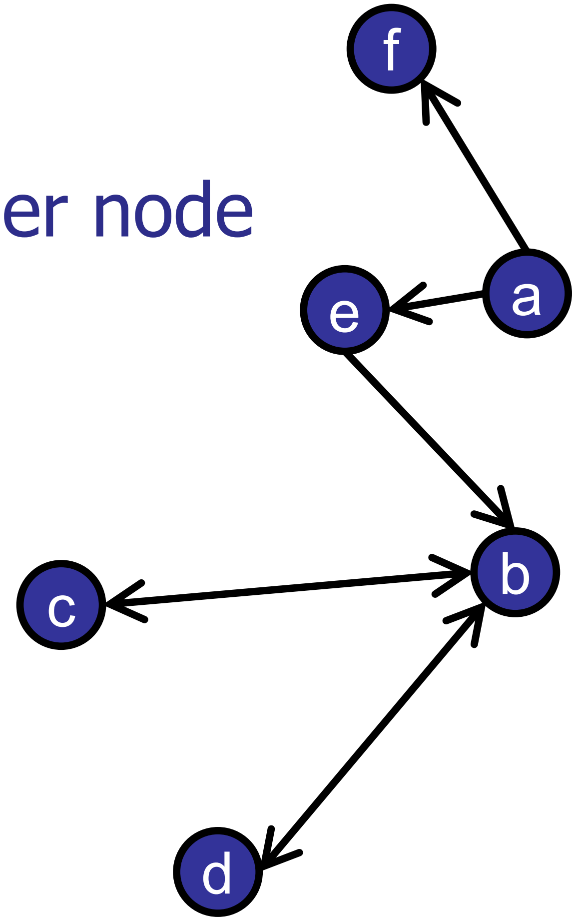


Adjacency List

Directed Graph consists of:

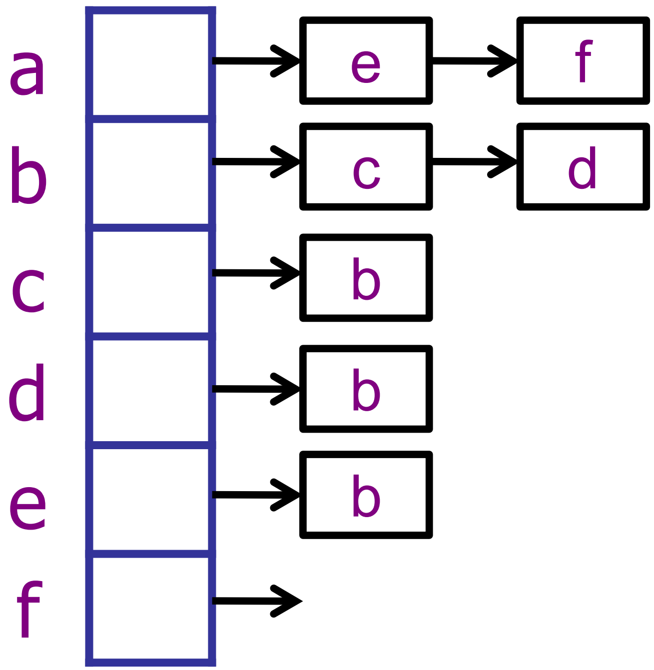
- Nodes: stored in an array
- **Outgoing** Edges: linked list per node





Adjacency List in C++

```
class Node {
  int key;
   LinkedList<int>;
}


class DirectedGraph {
  Node nodeList[MAXNODE];



}
```



Representing a (Directed) Graph

Adjacency List:

- Array of nodes
- Each node maintains a list of neighbors
- Space: $O(V + E)$

Adjacency Matrix:

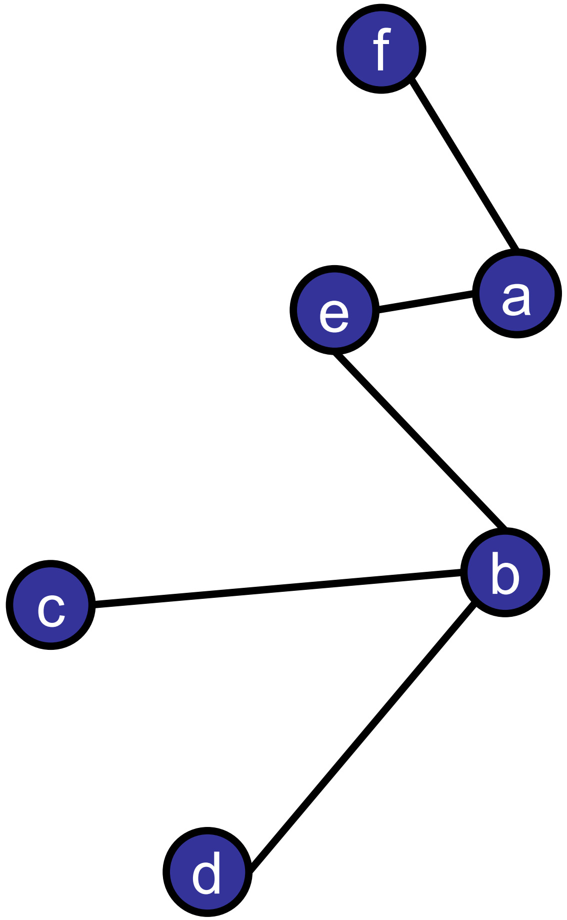
- Matrix $A[v,w]$ represents edge (v,w)
- Space: $O(V^2)$

Adjacency Matrix

Undirected Graph consists of:

- Nodes
- Edges = pairs of nodes

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**1**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**1**	**1**	0	0	0	0
f	**1**	0	0	0	0	0

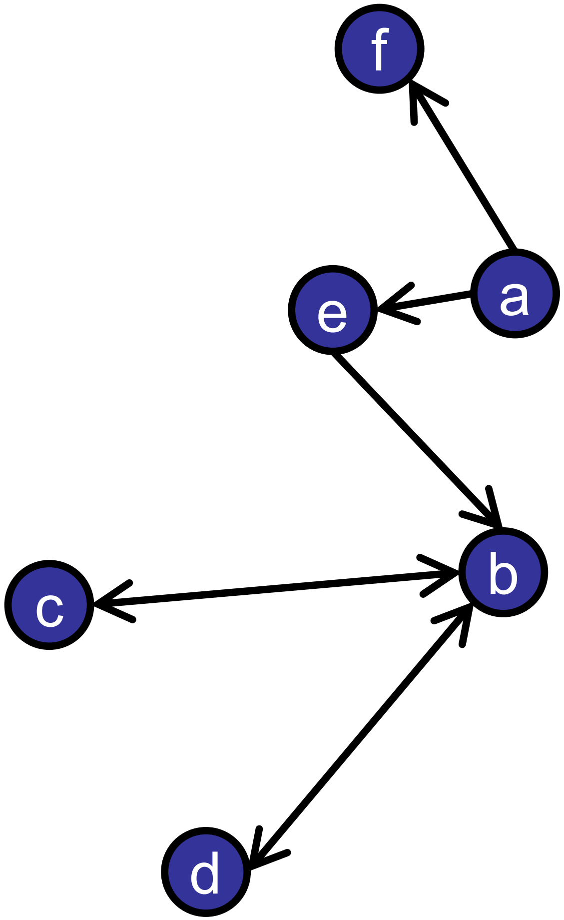


Adjacency Matrix

Directed Graph consists of:

- Nodes
- Edges = pairs of nodes

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**0**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**0**	**1**	0	0	0	0
f	**0**	0	0	0	0	0



Adjacency Matrix

Graph represented as:

- $A[v][w] = 1$ iff $(v,w) \in E$

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**0**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**0**	**1**	0	0	0	0
f	**0**	0	0	0	0	0

Searching a (Directed) Graph

Breadth-First Search:

- Search level-by-level
- Follow outgoing edges
- Ignore incoming edges

Depth-First Search:

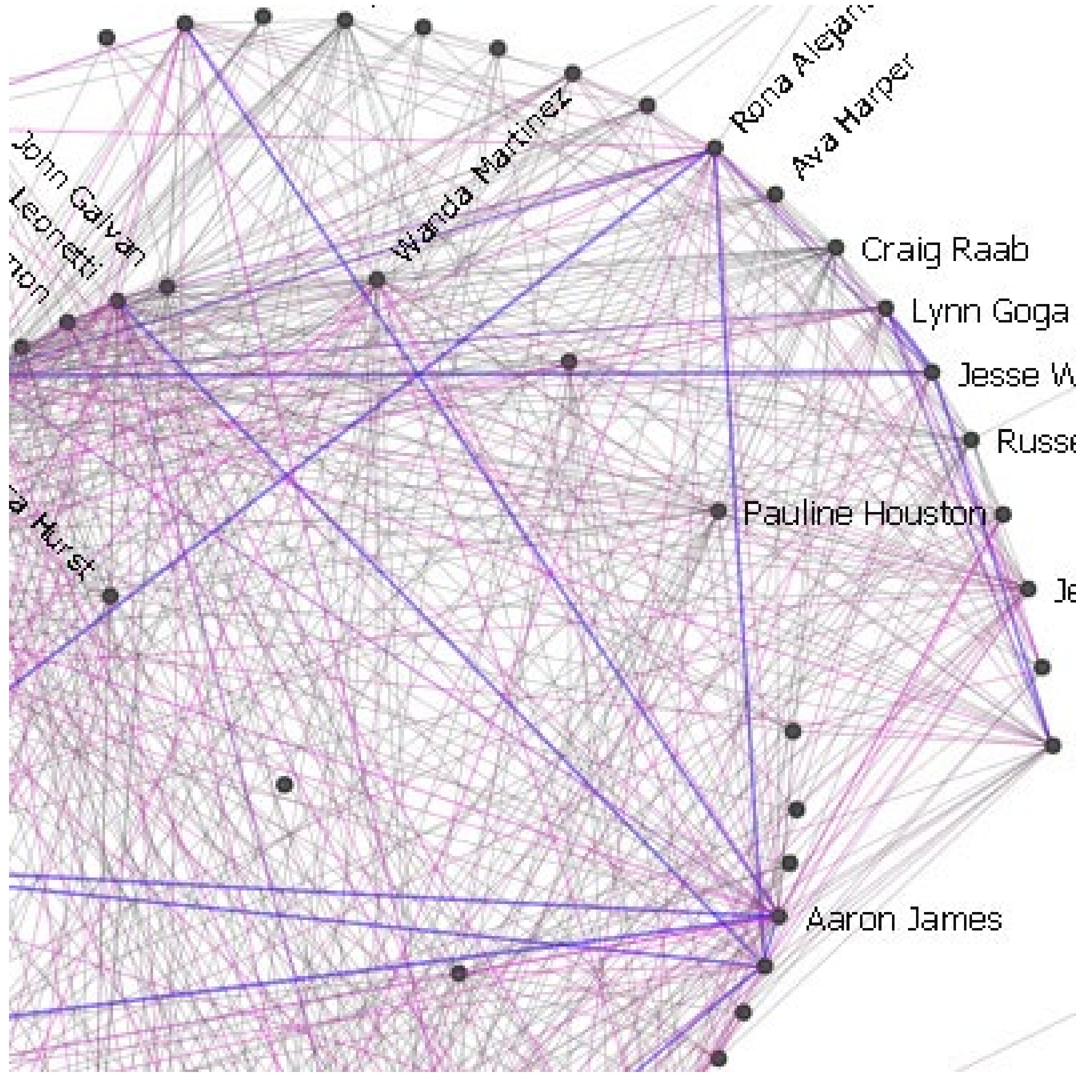
- Search recursively
- Follow outgoing edges
- Backtrack (through incoming edges)

Applications of directed graphs

Directed Graphs

Is friendship always bidirectional?:

- Nodes are people
- Edge = friendship



Facebook: yes

Google+: no

Directed Graphs

Markov text generation:

- Nodes are kgrams
  - A k-gram is a contiguous sequence of k items e.g. syllables, letters, words, etc.
- Edge = one kgram follows another