https://algs4.cs.princeton.edu

4.1 UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *applications of DFS and BFS*

Last updated on 3/27/19 9:15 PM

https://algs4.cs.princeton.edu

4.1 UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *applications of DFS and BFS*

Undirected graphs

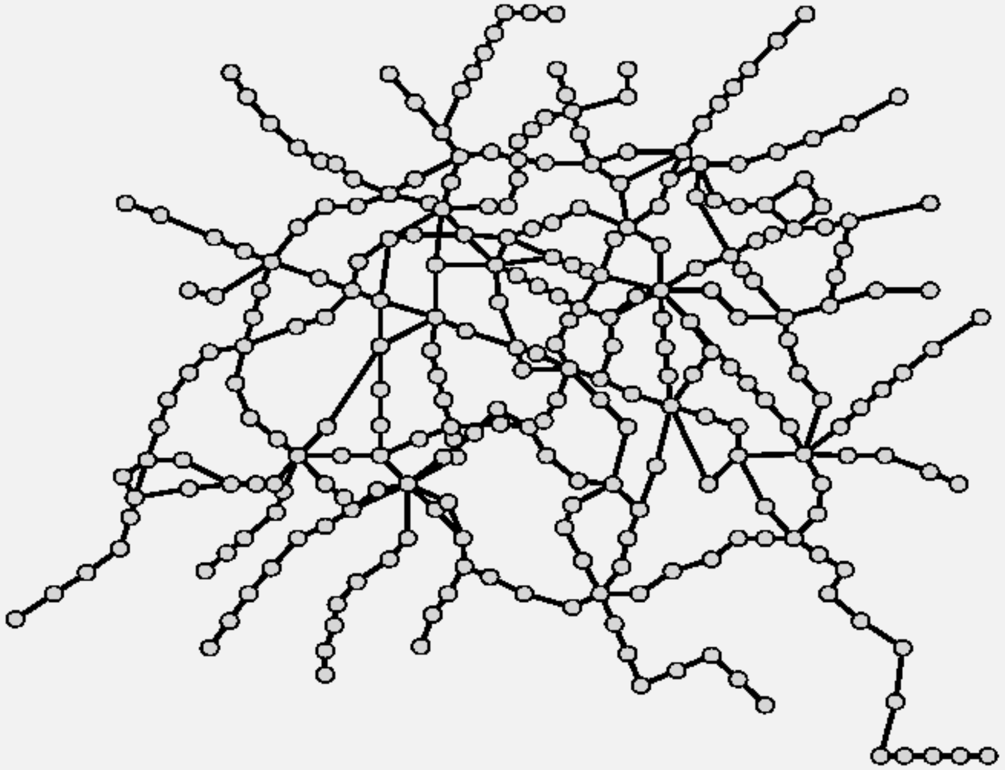
Graph. Set of vertices connected pairwise by edges.

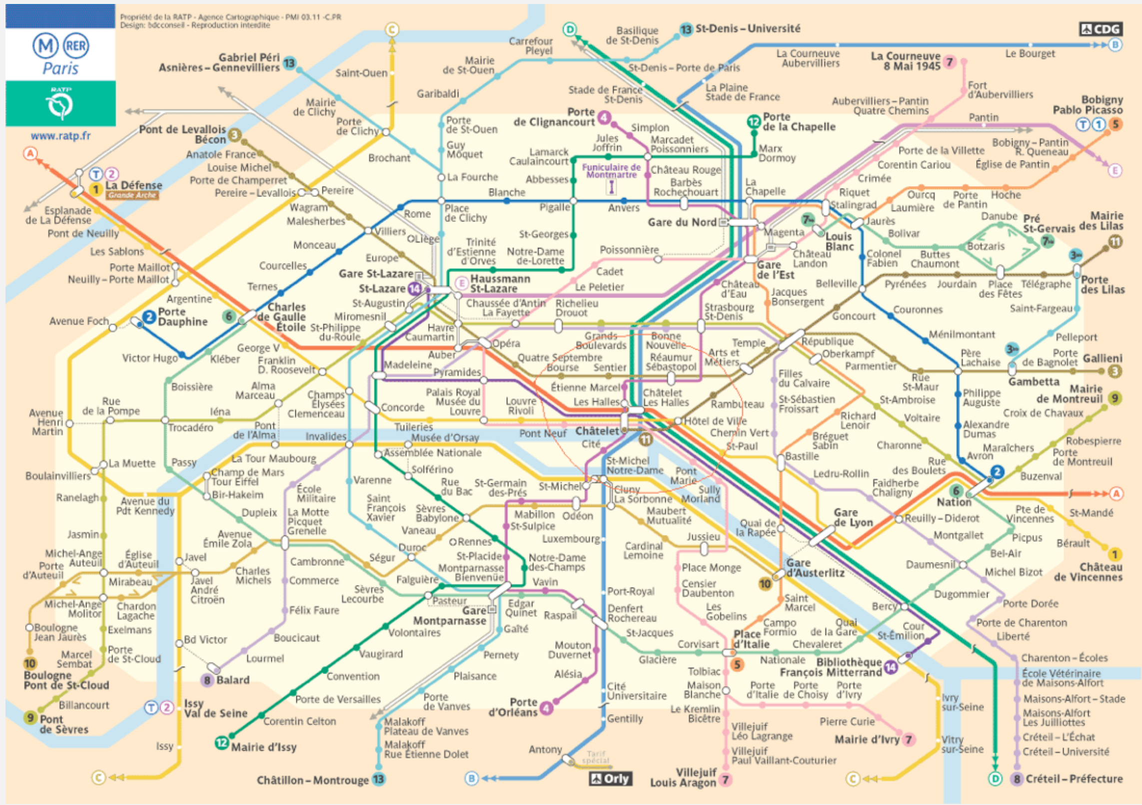
Why study graph algorithms?

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

Rail network

Vertex = station; edge = route





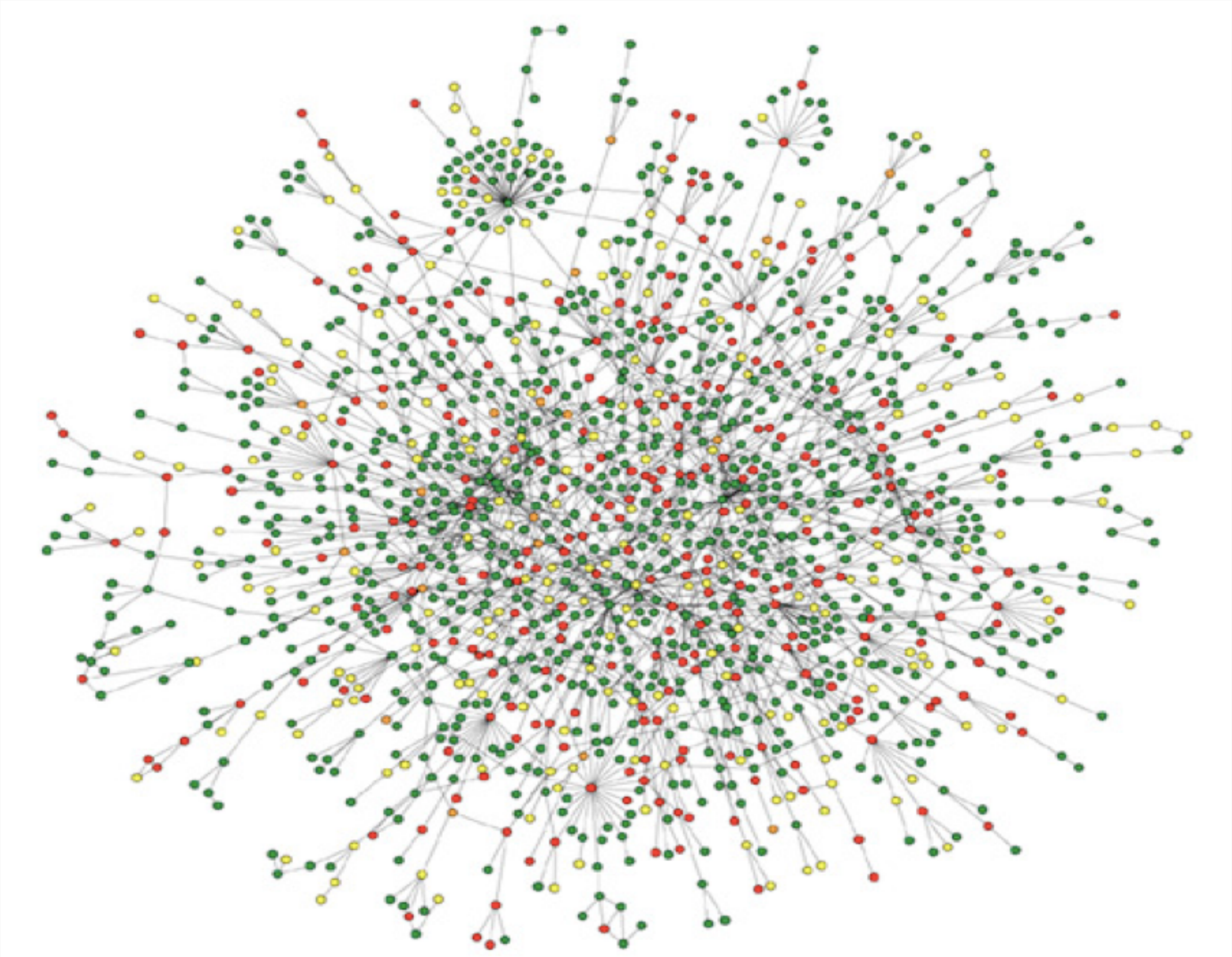
Social networks

Vertex = person; edge = social relationship.

"Visualizing Friendships" by Paul Butler

Protein-protein interaction network

Vertex = protein; edge = interaction.

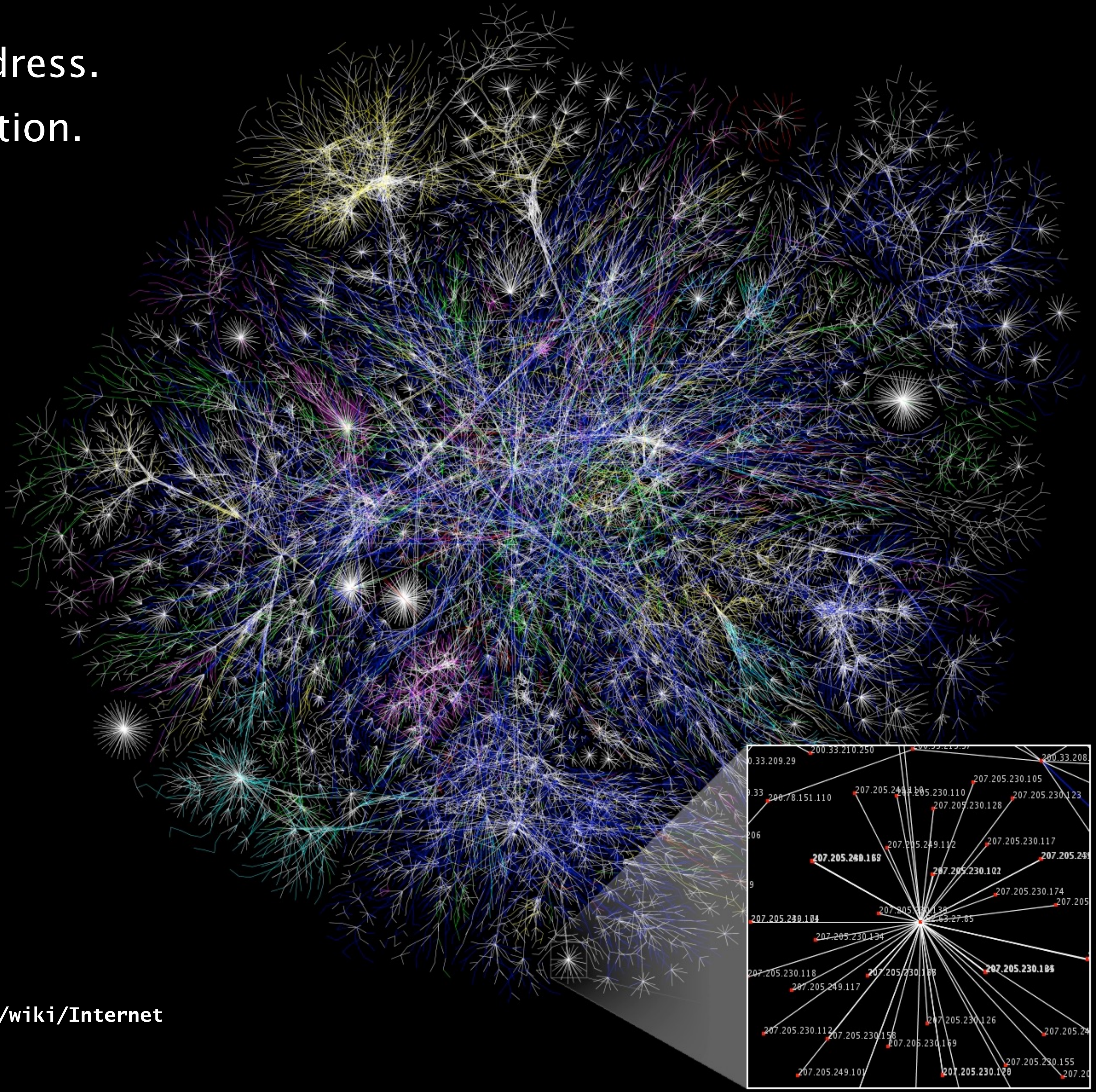


Reference: Jeong et al, Nature Review | Genetics

The Internet as mapped by the Opte Project

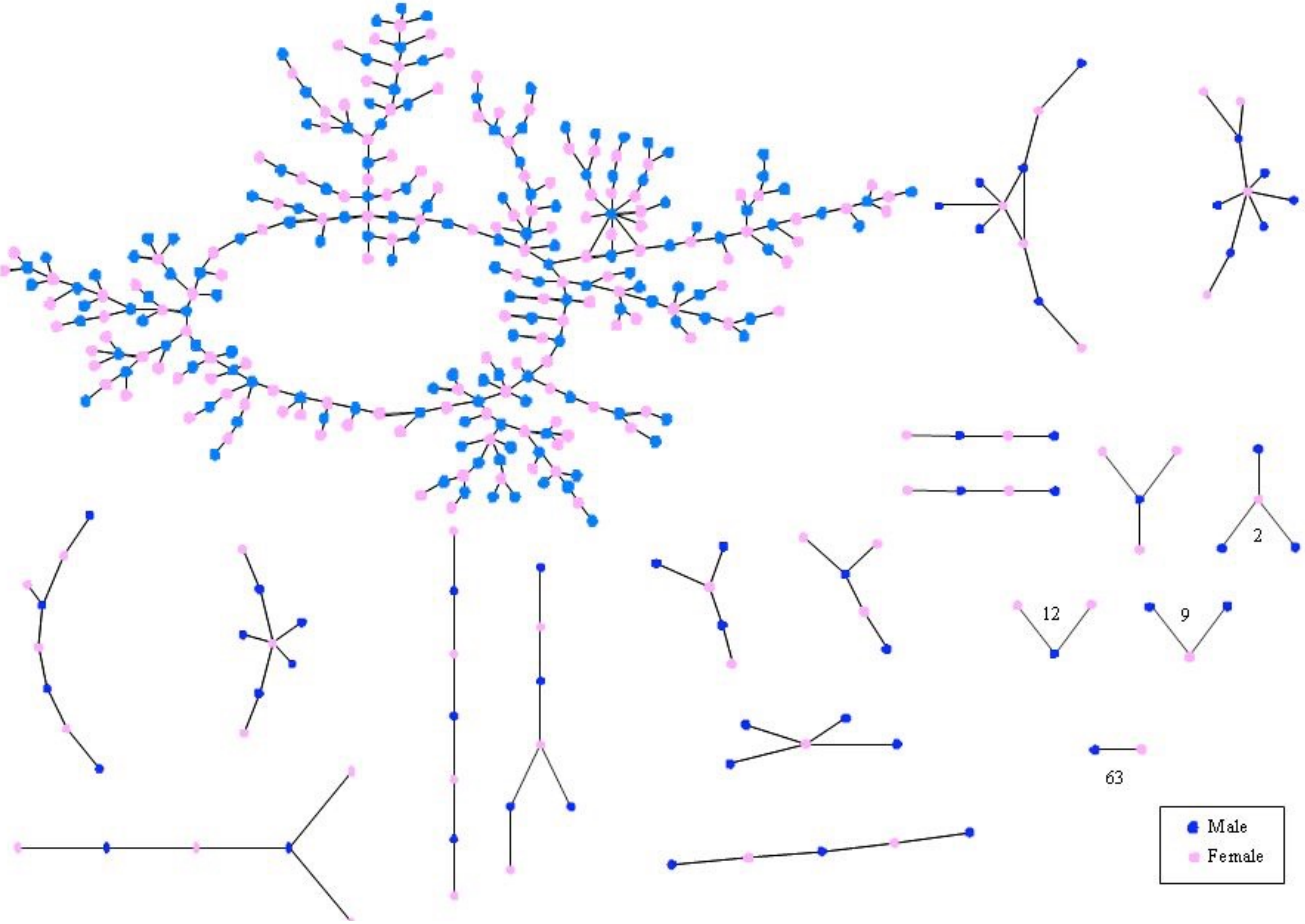
Vertex = IP address.

Edge = connection.



`http://en.wikipedia.org/wiki/Internet`

Romantic and sexual relationships in a high school



Relationship graph at "Jefferson High"

Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. American Journal of Sociology, 110(1): 44–99, 2004.

Graph applications

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

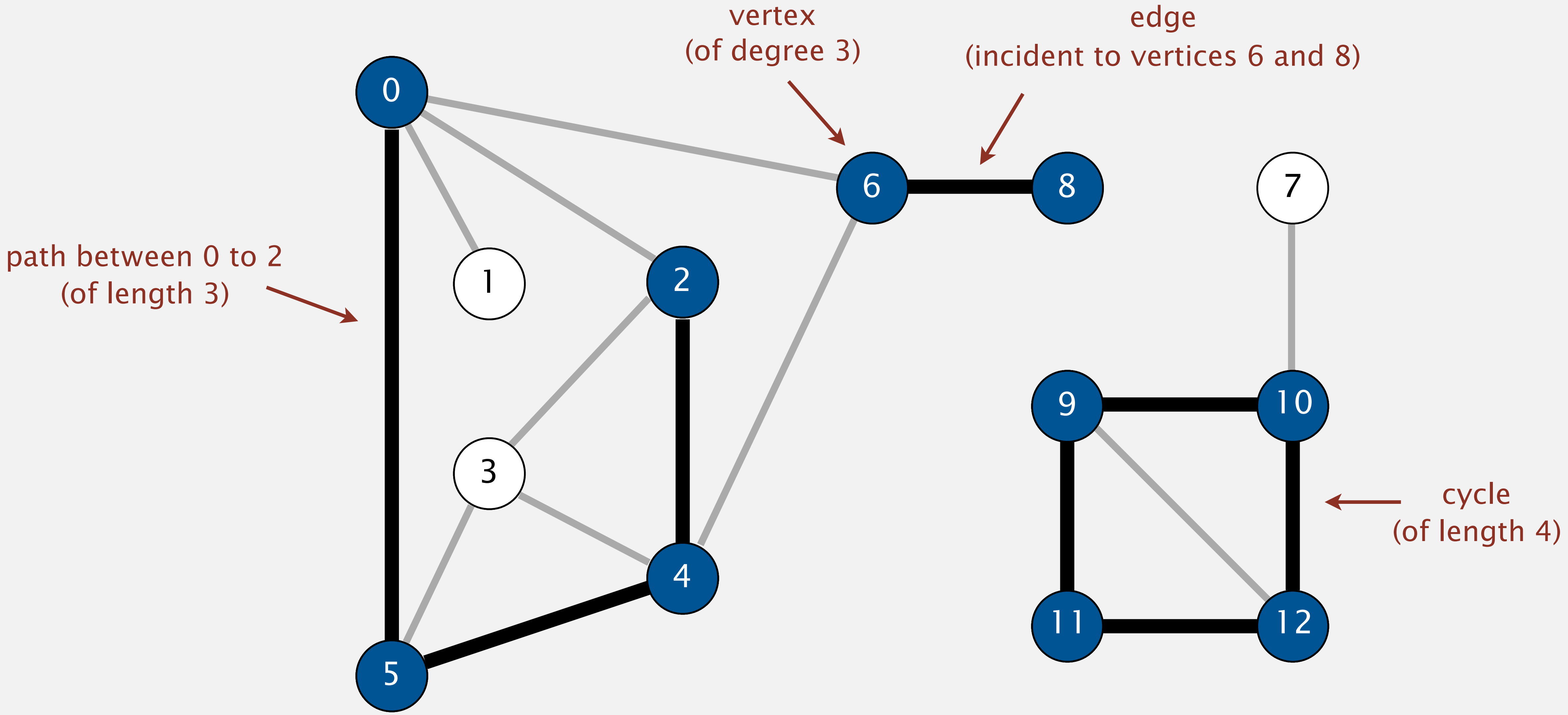
Graph terminology

Graph: set of vertices connected pairwise by edges.

Path: sequence of vertices connected by edges, with no repeated edges.

Two vertices are connected if there is a path between them.

Cycle: Path (with at least 1 edge) whose first and last vertices are the same.



Some graph-processing problems

problem	description
s-t path	*Is there a path between s and t ?*
shortest s-t path	*What is the shortest path between s and t ?*
cycle	*Is there a cycle in the graph ?*
Euler cycle	*Is there a cycle that uses each edge exactly once ?*
Hamilton cycle	*Is there a cycle that uses each vertex exactly once ?*
connectivity	*Is there a path between every pair of vertices ?*
biconnectivity	*Is there a vertex whose removal disconnects the graph ?*
planarity	*Can the graph be drawn in the plane with no crossing edges ?*
graph isomorphism	*Are two graphs isomorphic?*

Challenge. Which graph problems are easy? Difficult? Intractable?

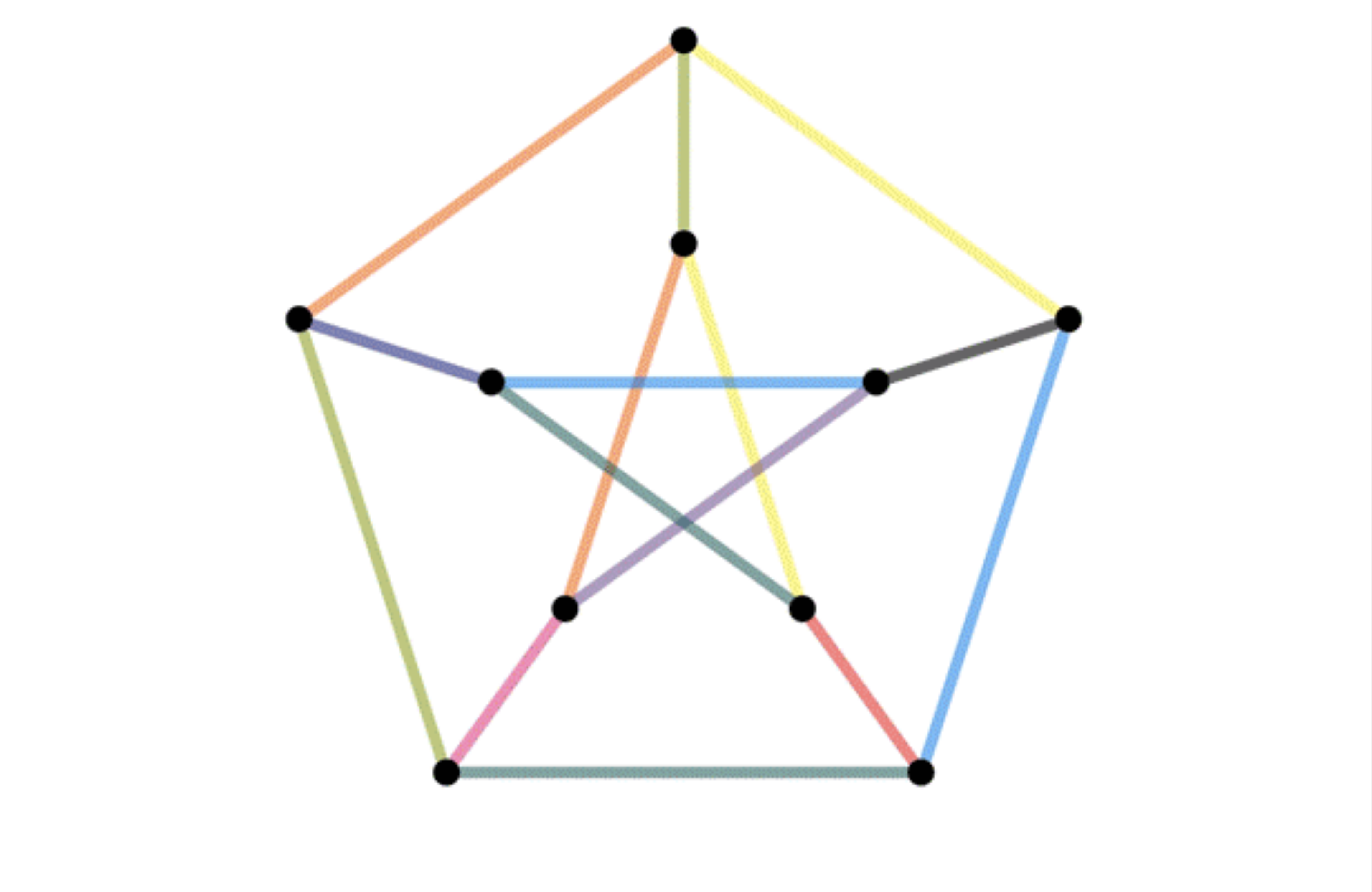
https://algs4.cs.princeton.edu

4.1 UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *applications of DFS and BFS*

Graph representation

Graph drawing. Provides intuition about the structure of the graph.



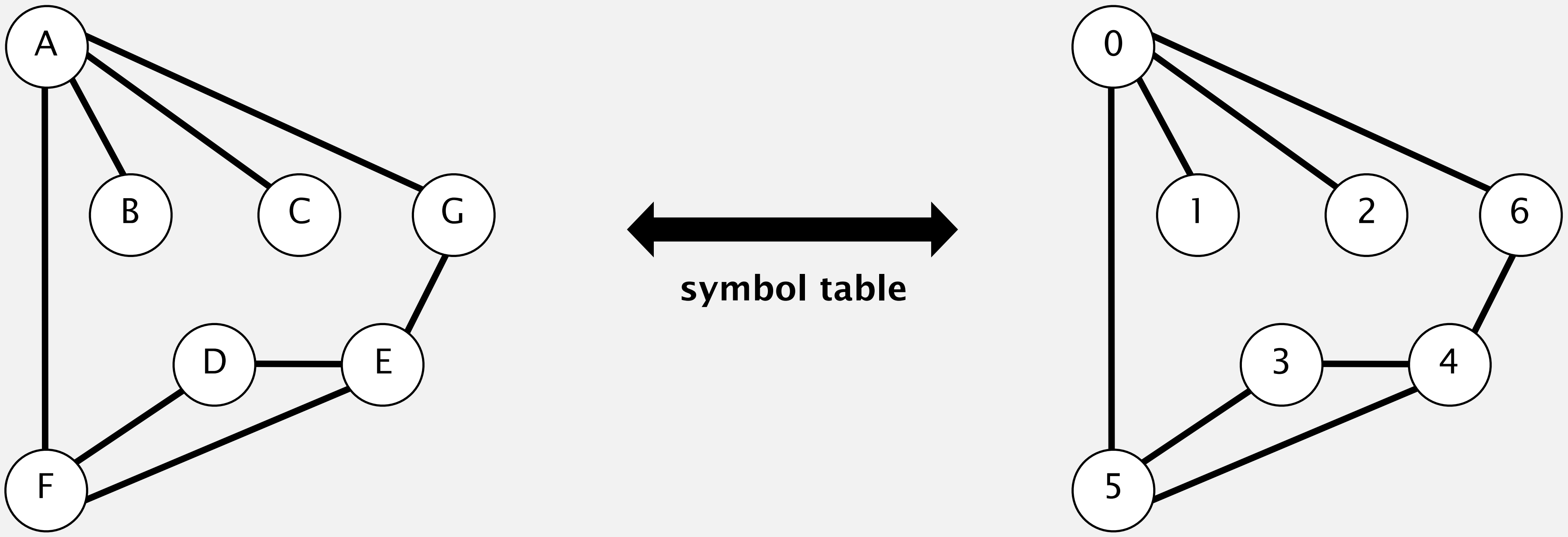
different drawings of the same graph

Caveat. Intuition can be misleading.

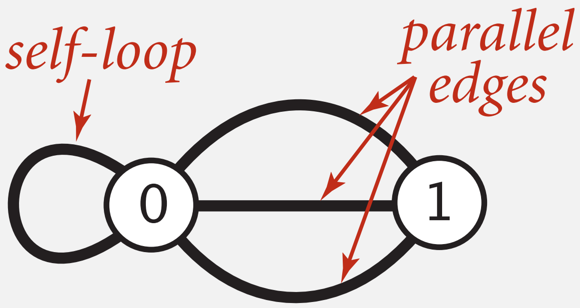
Graph representation

Vertex representation.

- This lecture: integers between 0 and $V - 1$.
- Applications: use symbol table to convert between names and integers.



Anomalies.



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)`	*vertices adjacent to v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*

All graph processing can be done using above API. Example:

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

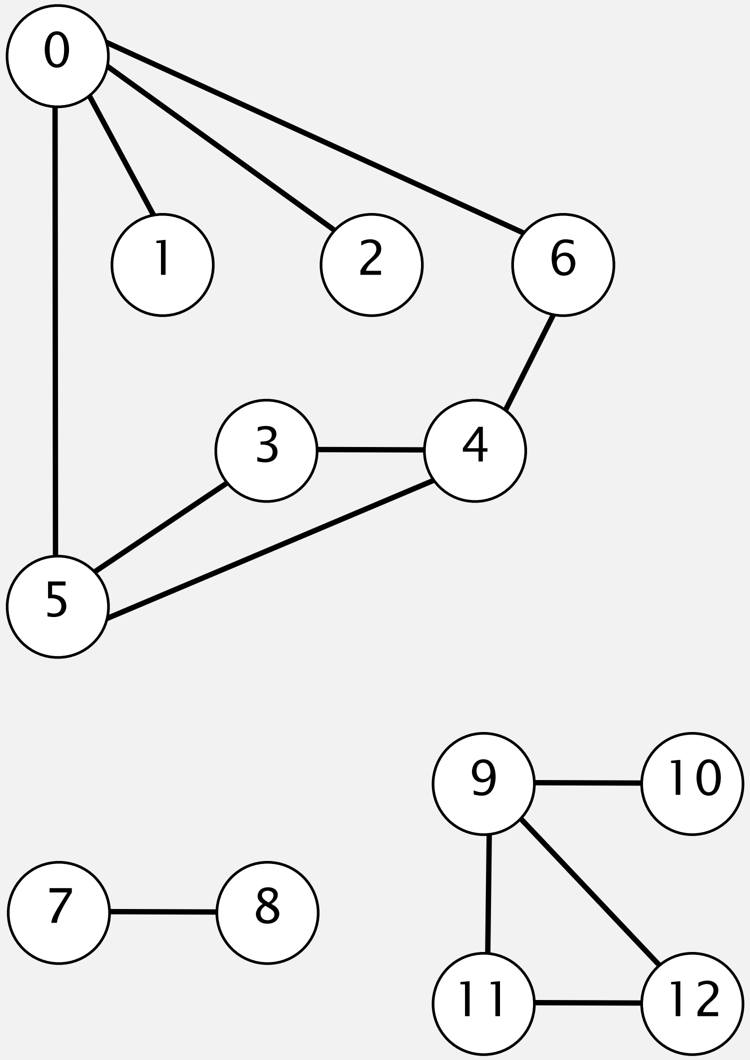
Arvind's view:

This API is oversimplified. Any competent graph API must provide `degree()` and other methods.

Graph representation: adjacency matrix

Maintain a *V*-by-*V* boolean array; for each edge *v*–*w* in graph:

`adj[v][w] = adj[w][v] = true.`

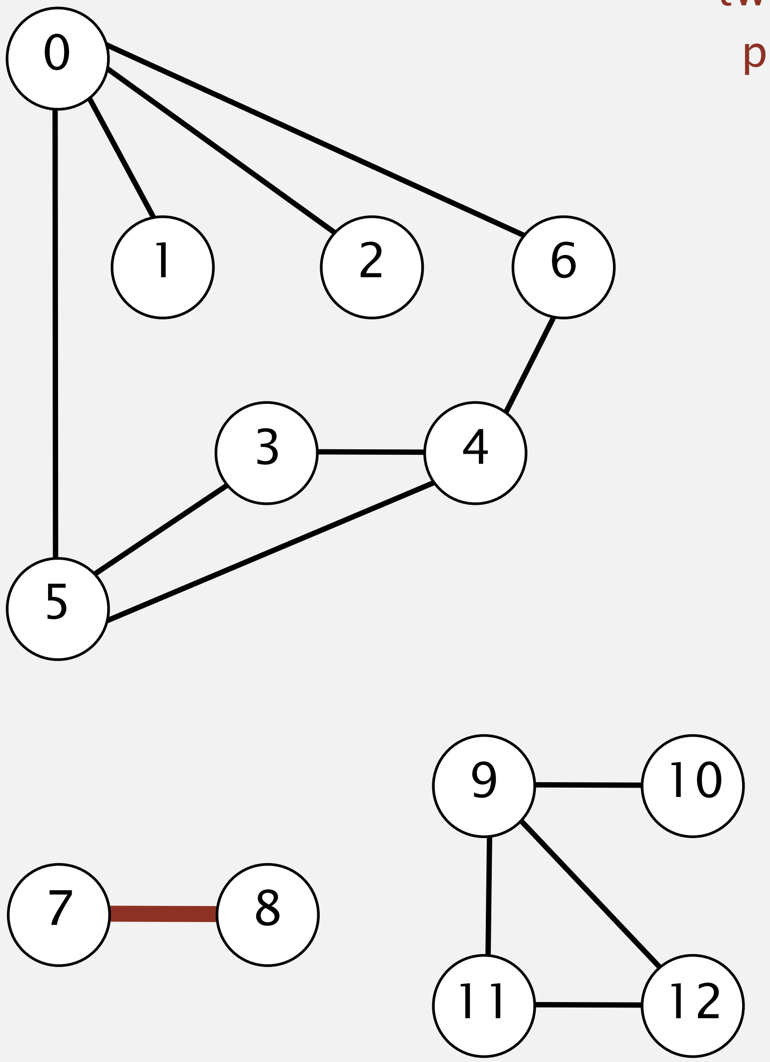


	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

Graph representation: adjacency matrix

Maintain a V-by-V boolean array; for each edge $v–w$ in graph:

`adj[v][w] = adj[w][v] = true.`



two entries per edge

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

Undirected graphs: quiz 1

Which is the order of growth of running time of the following code fragment if the graph uses the adjacency-matrix representation, where V is the number of vertices and E is the number of edges?

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

print each edge twice

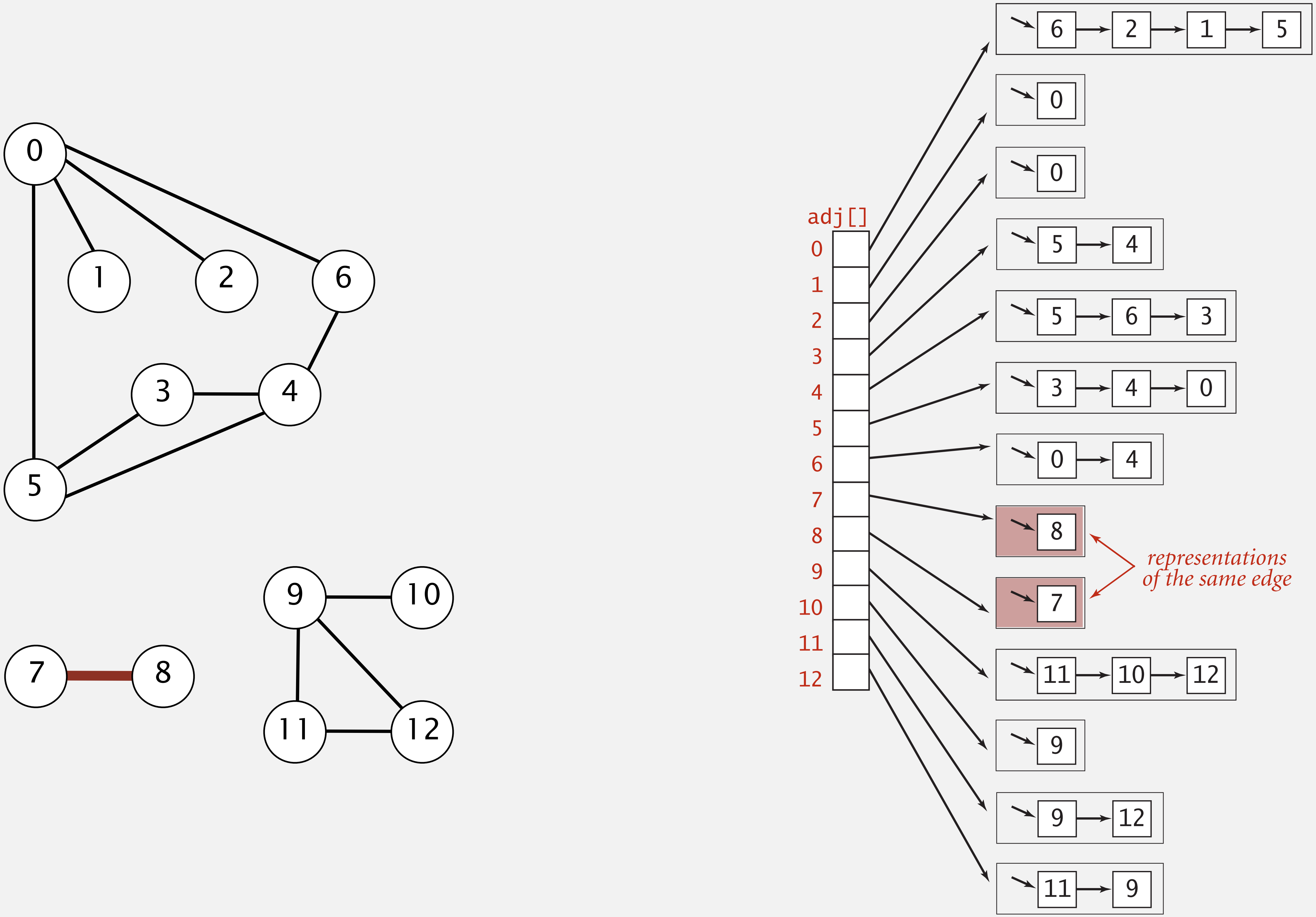
- **A.** V
- **B.** $E + V$
- **C.** V^2
- **D.** $V E$

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

adjacency-matrix representation

Graph representation: adjacency lists

Maintain vertex-indexed array of lists.



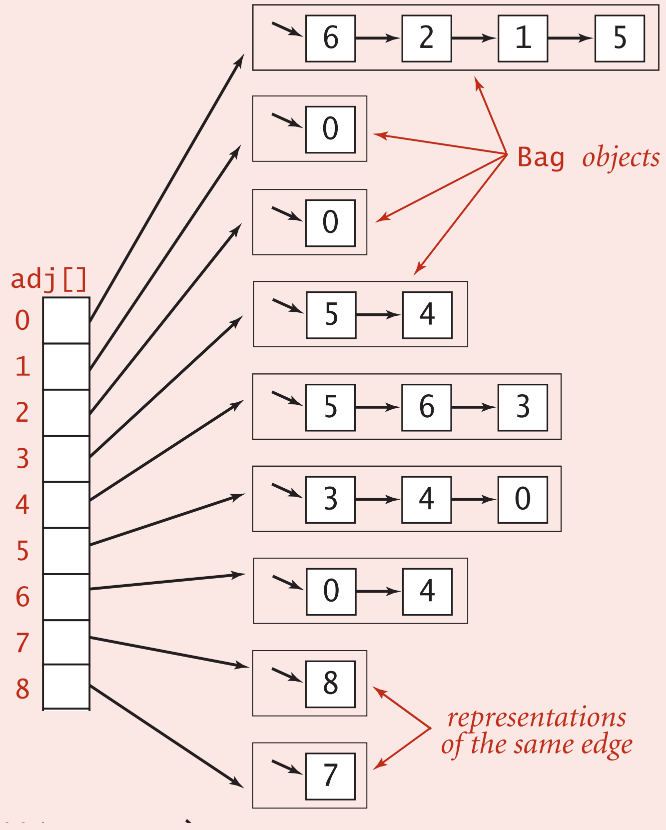
Undirected graphs: quiz 2

Which is the order of growth of running time of the following code fragment if the graph uses the adjacency-lists representation, where V is the number of vertices and E is the number of edges?

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

print each edge twice

- **A.** V
- **B.** $E + V$
- **C.** V^2
- **D.** $V E$



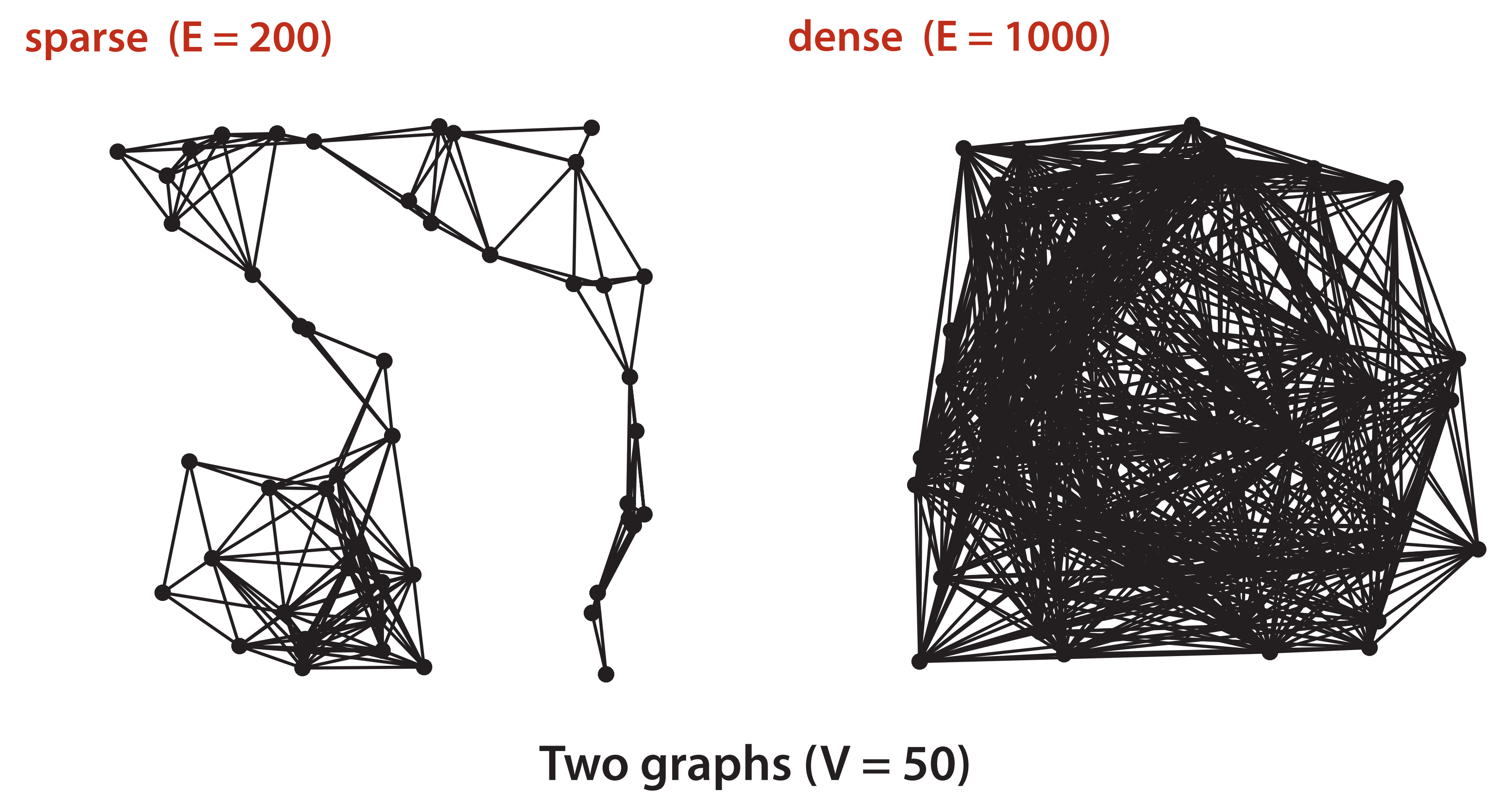
Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to *v*.
- Real-world graphs tend to be sparse (not dense).

proportional to V edges

proportional to V^2 edges



Two graphs (V = 50)

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be sparse (not dense).

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	$\boxed{E + V}$	1	$degree(v)$	$\boxed{degree(v)}$

† disallows parallel edges

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];  }
}
```

https://algs4.cs.princeton.edu/41undirected/Graph.java.html

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

https://algs4.cs.princeton.edu

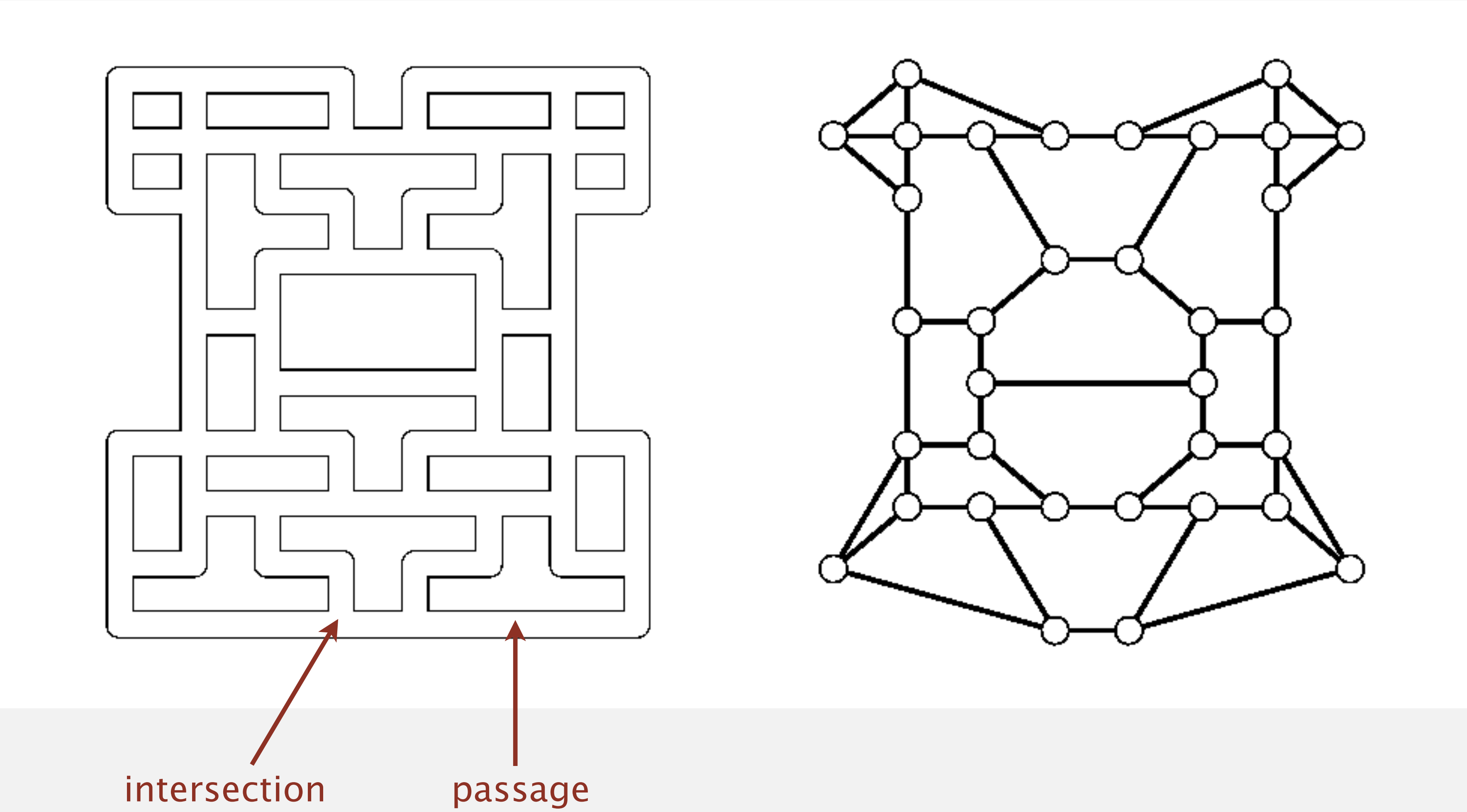
4.1 UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *applications of DFS and BFS*

Warmup: maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.

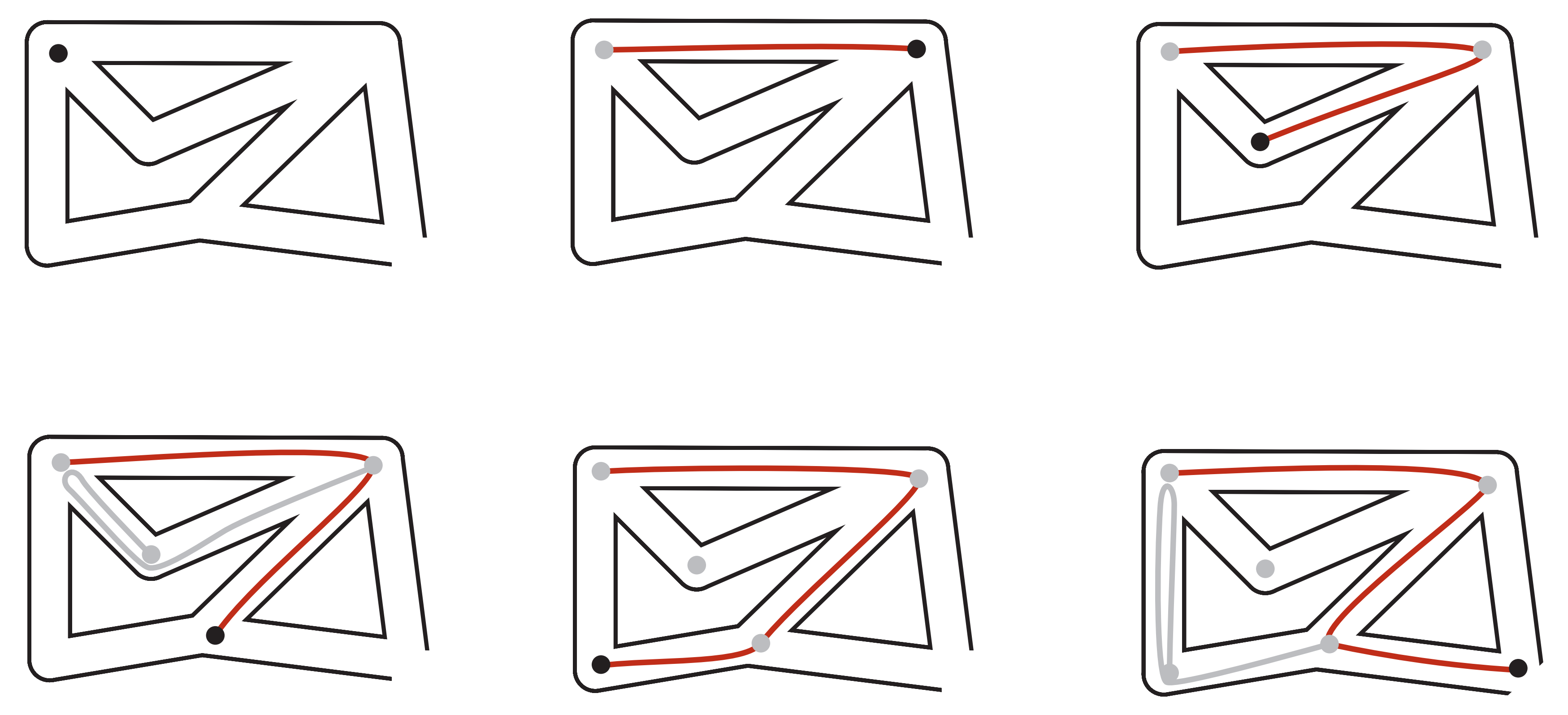


Goal. Explore every intersection in the maze.

Maze exploration algorithm in Greek myth

How Theseus escaped from the labyrinth after killing the Minotaur:

- Unroll a ball of string behind you.
- Mark each newly discovered intersection.
- Retrace steps when no unmarked options.



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.
```

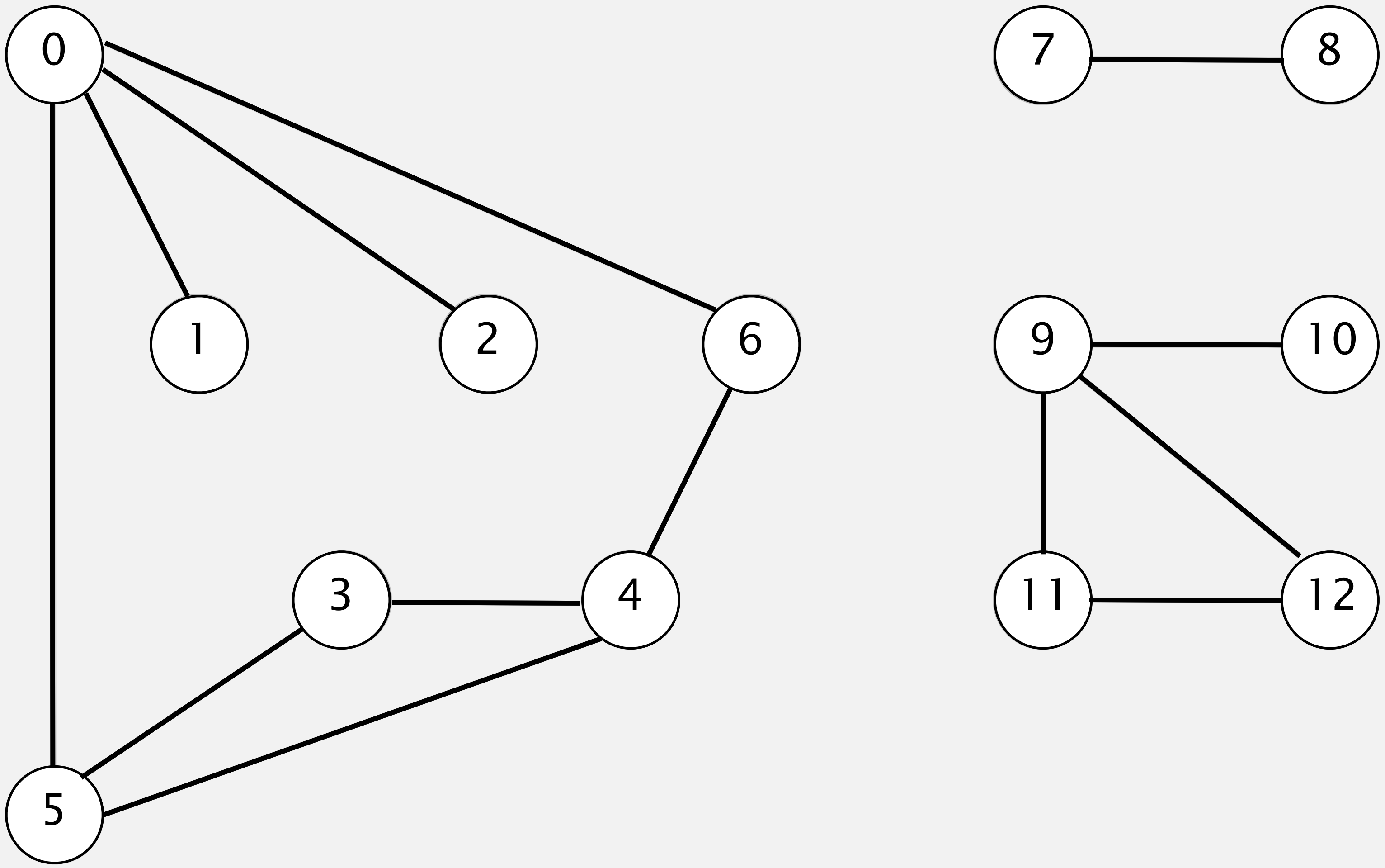
Typical applications.

- Find all vertices connected to a given vertex.
- Find a path between two vertices.

Depth-first search demo

To visit a vertex v :

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

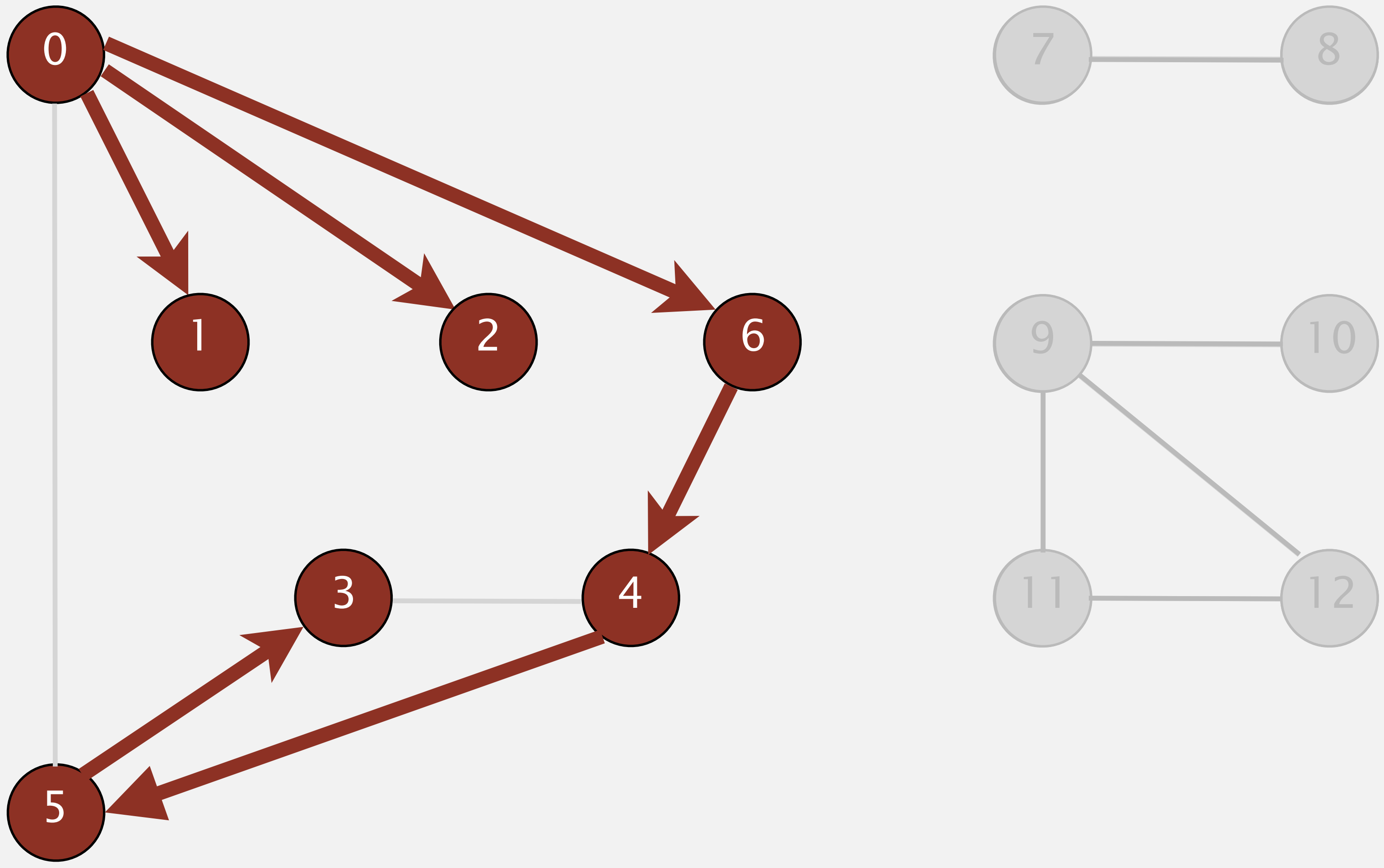


graph G

Depth-first search demo

To visit a vertex v :

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



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

vertices connected to 0

(and associated paths)

Undirected graphs: quiz 3

Run DFS using the following adjacency-lists representation of graph G, starting at vertex 0. In which order is dfs(G, v) called?

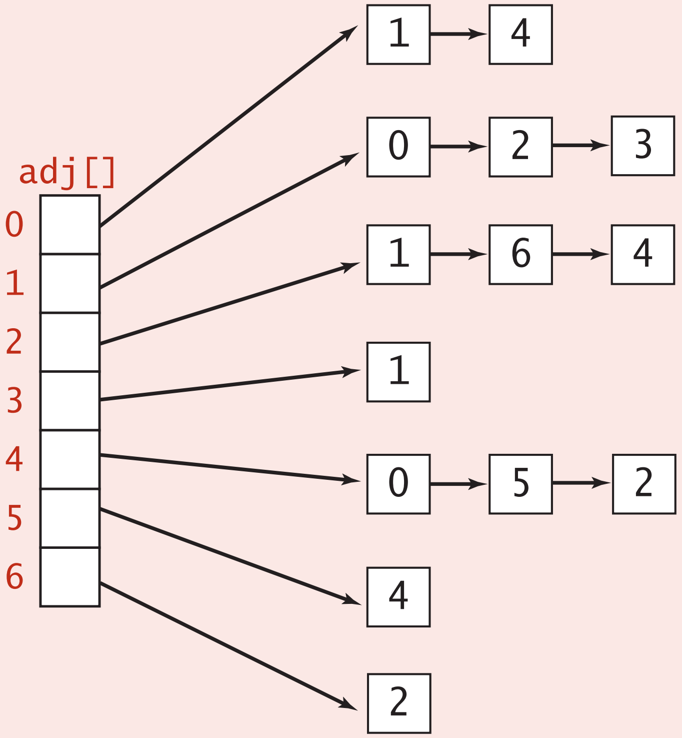
DFS preorder

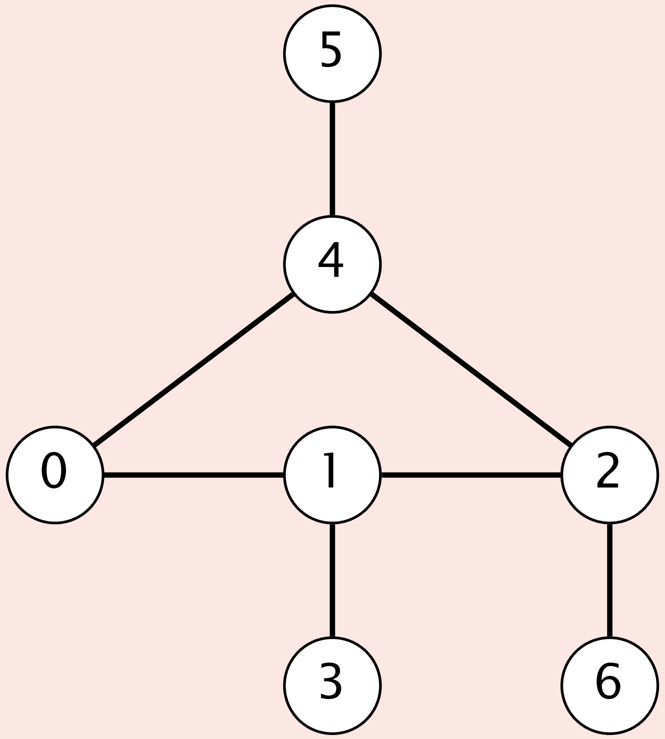
A. 0 1 2 4 5 3 6

B. 0 1 2 4 5 6 3

C. 0 1 4 2 5 3 6

D. 0 1 2 6 4 5 3





Depth-first search: Java implementation

```java
public class DepthFirstPaths
{

   private boolean[] marked;
   private int[] edgeTo;
   private int s;

   public DepthFirstPaths(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])
         {
            edgeTo[w] = v;
            dfs(G, w);
         }
   }

}
```

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

https://algs4.cs.princeton.edu/41undirected/DepthFirstPaths.java.html

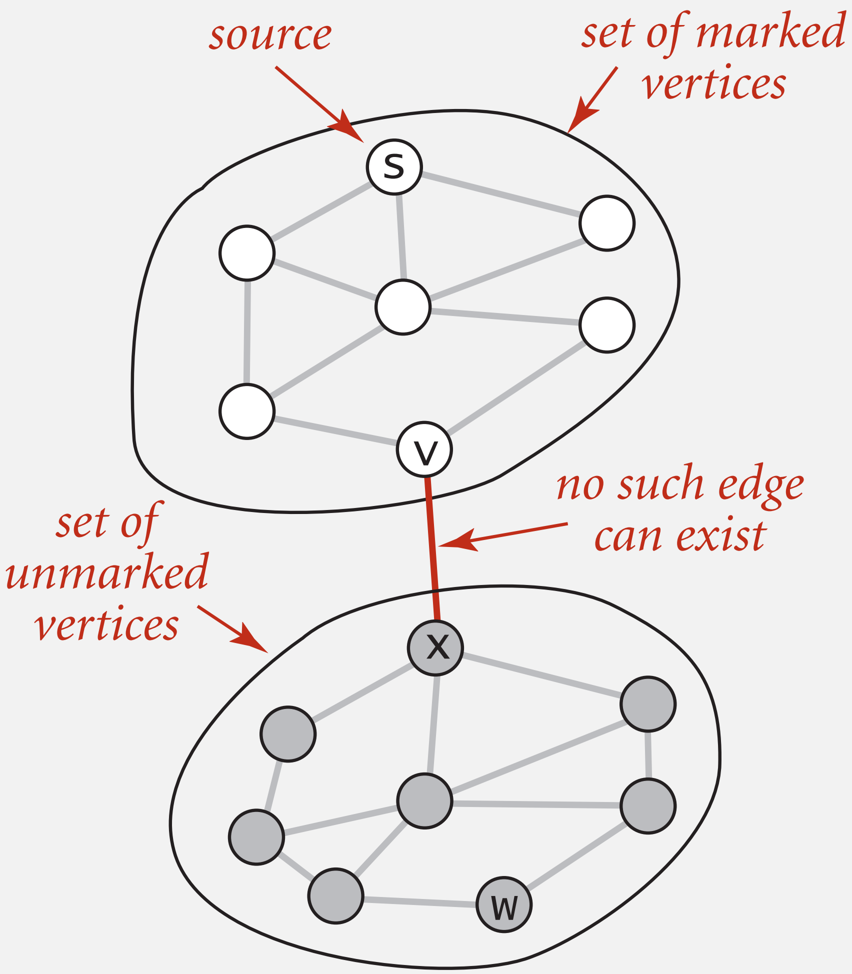
Depth-first search: properties

Proposition. DFS marks all vertices connected to s (and no others).

Proof.

- If w marked, then w connected to s (why?)
- If w connected to s, then w marked.
  (if w unmarked, then consider the last edge on a path from s to w that goes from a marked vertex to an unmarked one).

Skipped in class



Depth-first search: properties

Proposition. DFS marks all vertices connected to s in time proportional to $V + E$ in the worst case.

Proof.

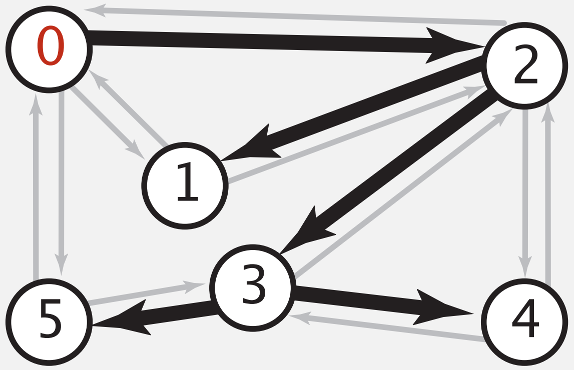
- Initialize two arrays of length V.
- Each vertex is visited at most once.
  (visiting a vertex takes time proportional to its degree)

$$degree(v_0) + degree(v_1) + degree(v_2) + \ldots = 2E$$

Depth-first search: properties

Proposition. After DFS, can check if vertex v is connected to s in constant time; can find v–s path (if one exists) in time proportional to its length.

Proof. `edgeTo[]` is parent-link representation of a tree rooted at vertex `s`.



`edgeTo[]`

0	
1	2
2	0
3	2
4	3
5	3

Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

4.1 UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *applications of DFS and BFS*

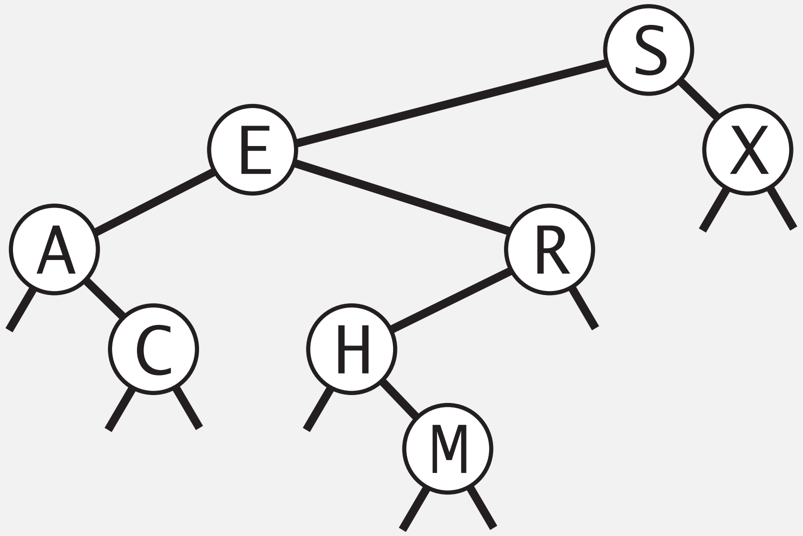
Graph search

Tree traversal. Many ways to explore a binary tree.

- Inorder: `A C E H M R S X`
- Preorder: `S E A C R H M X`
- Postorder: `C A M H R E X S`
- Level-order: `S E X A R C H M`

stack/recursion

queue



Graph search. Many ways to explore a graph.

- Preorder: vertices in order of calls to `dfs(G, v)`.
- Postorder: vertices in order of returns from `dfs(G, v)`.
- Level-order: vertices in increasing order of distance from `s`.

stack/recursion

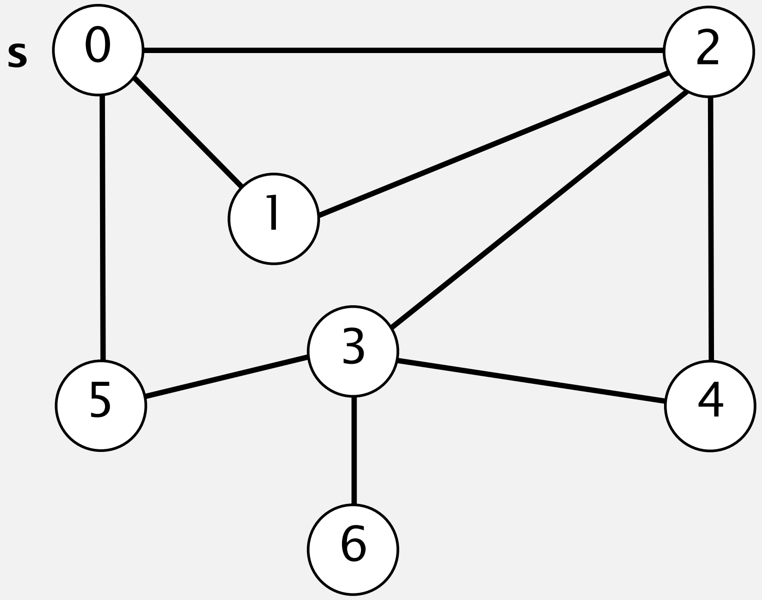
queue

Breadth-First Search (BFS)

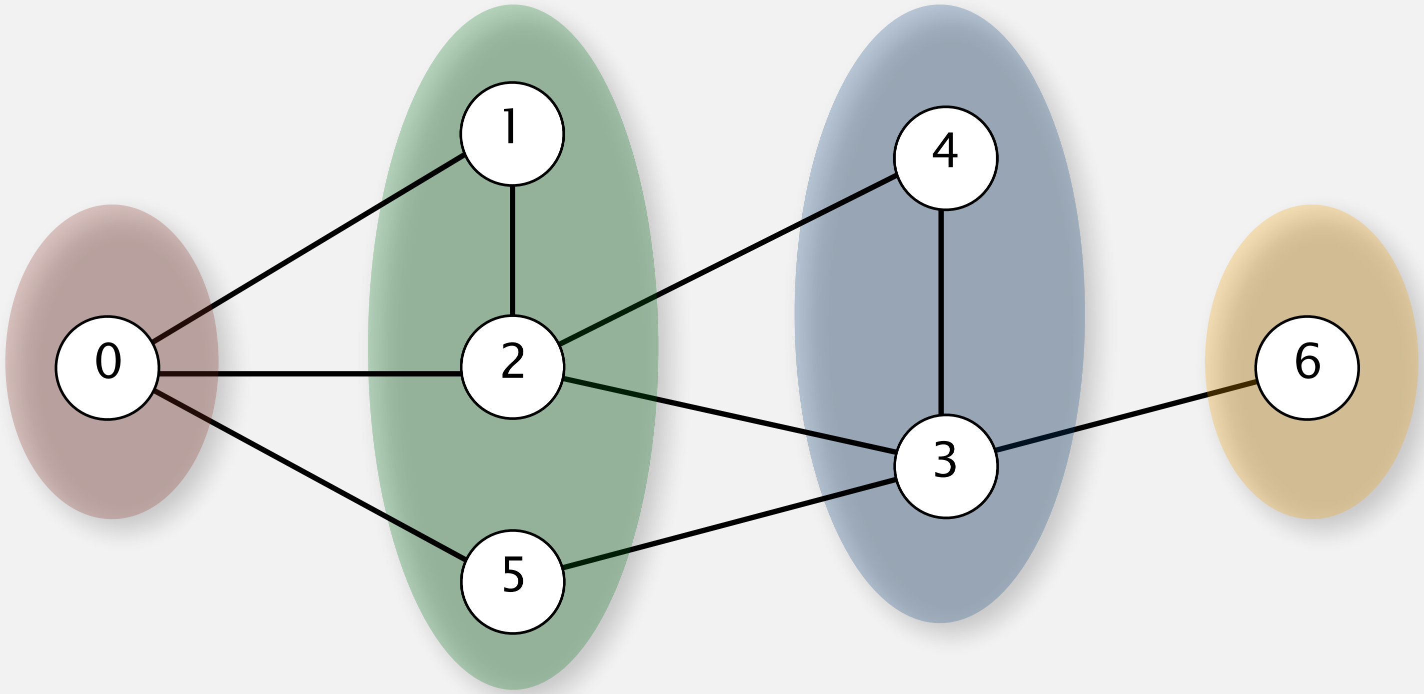
BFS (from source vertex s)

```
Put s on a queue, and mark s as visited.
Repeat until the queue is empty:
  - dequeue vertex v
  - enqueue each of v's unmarked neighbors, and mark them.
```

Intuition. BFS traverses vertices in order of distance from s.



graph G



dist = 0

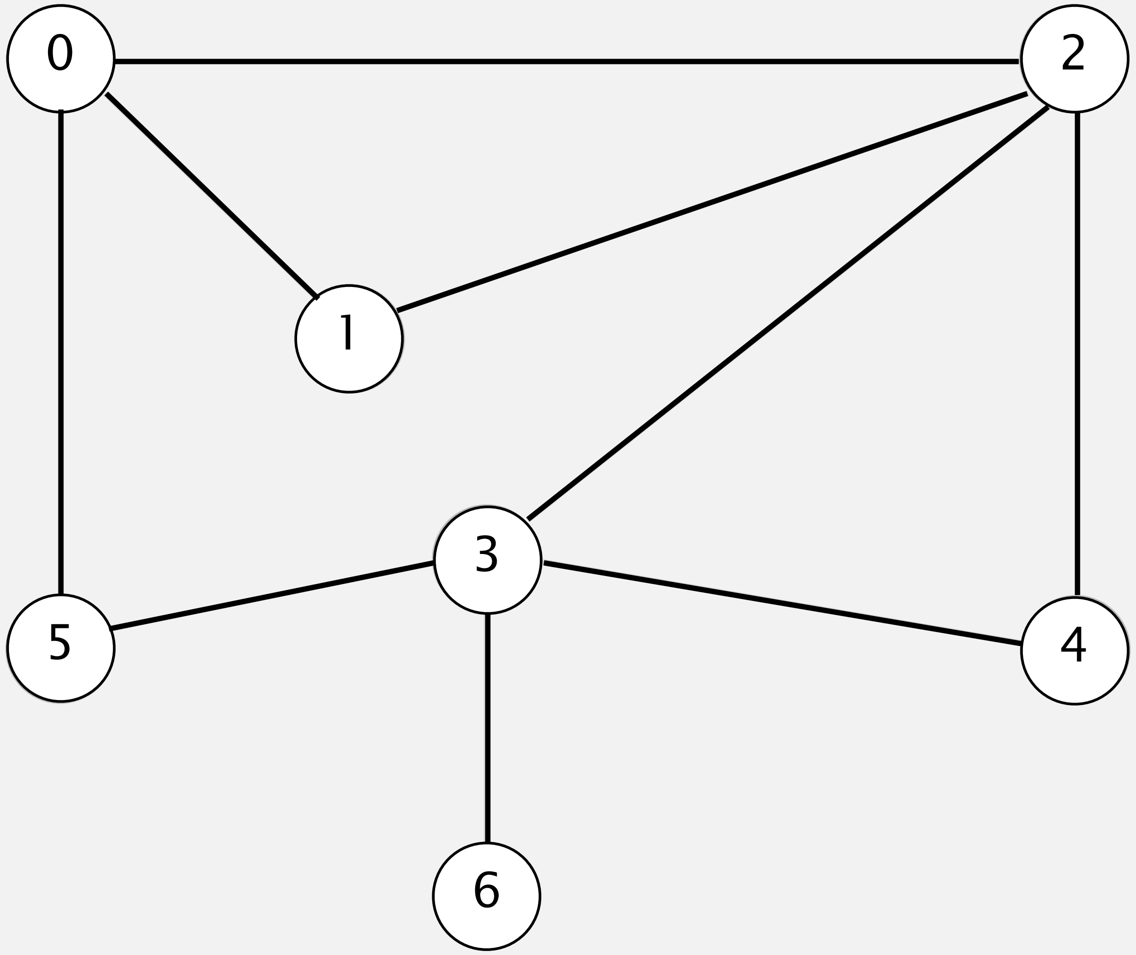
dist = 1

dist = 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.

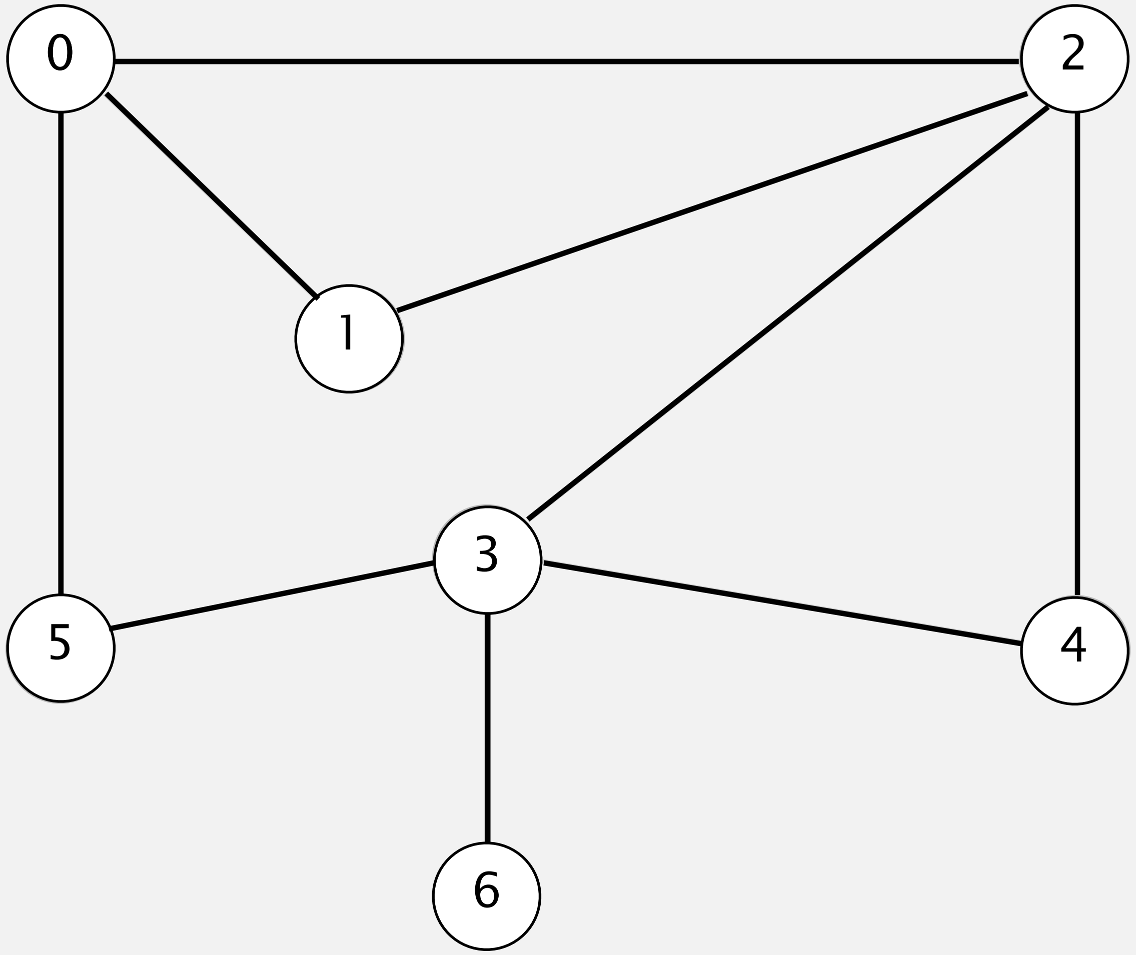


graph G

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

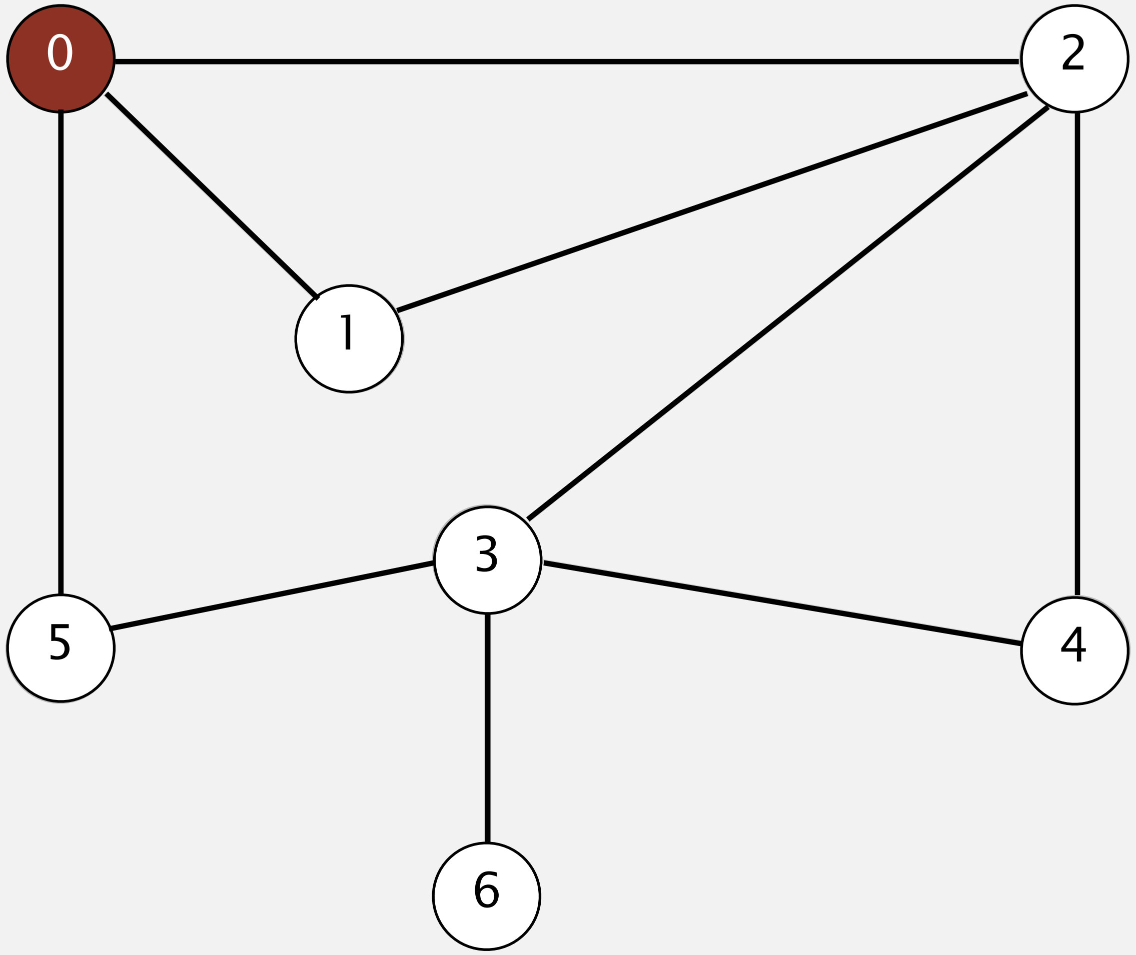
v	edgeTo[]	marked[]
0	–	T
1	–	F
2	–	F
3	–	F
4	–	F
5	–	F
6	–	F

enqueue 0

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

0

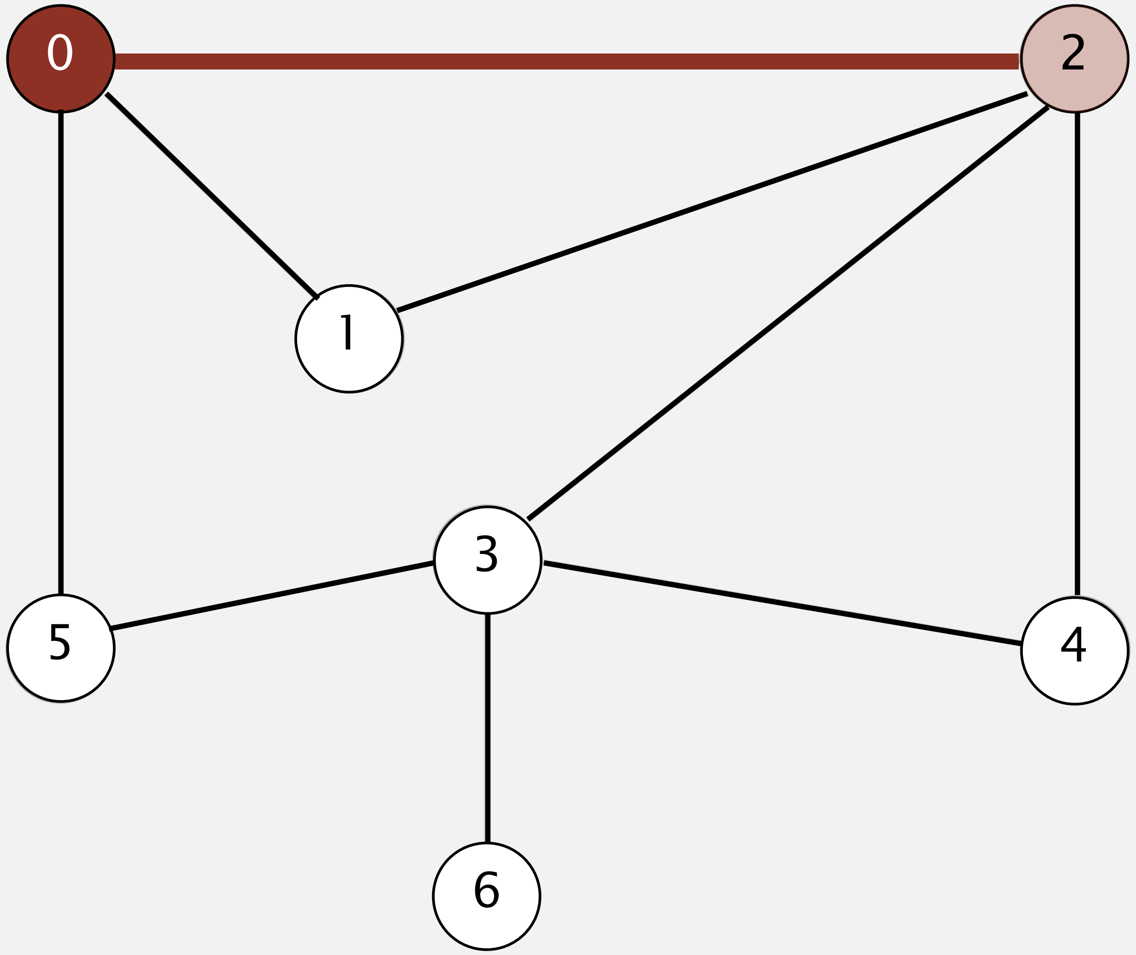
v	edgeTo[]	marked[]
0	–	T
1	–	F
2	–	F
3	–	F
4	–	F
5	–	F
6	–	F

dequeue 0

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

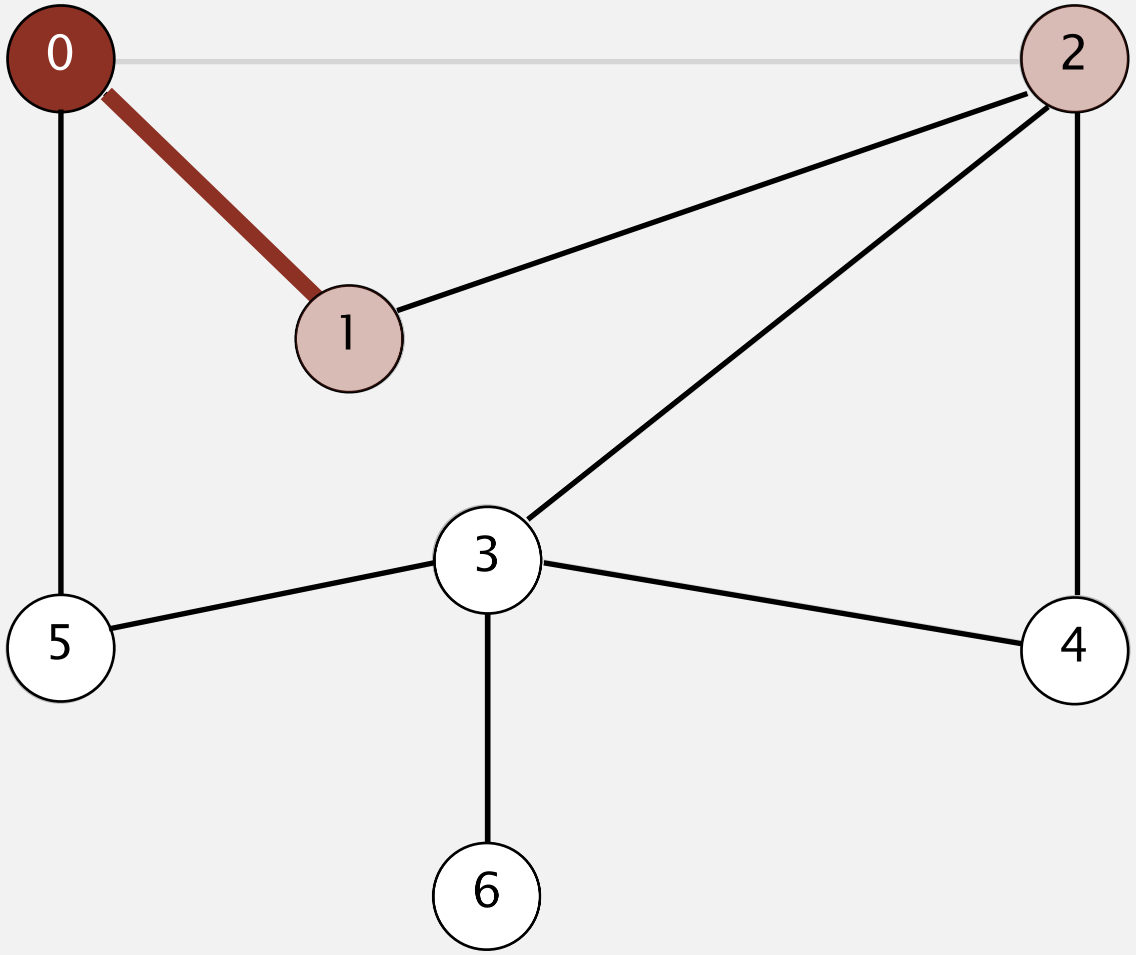
v	edgeTo[]	marked[]
0	–	T
1	–	F
2	0	T
3	–	F
4	–	F
5	–	F
6	–	F

dequeue 0: check 2, check 1, check 5

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

2

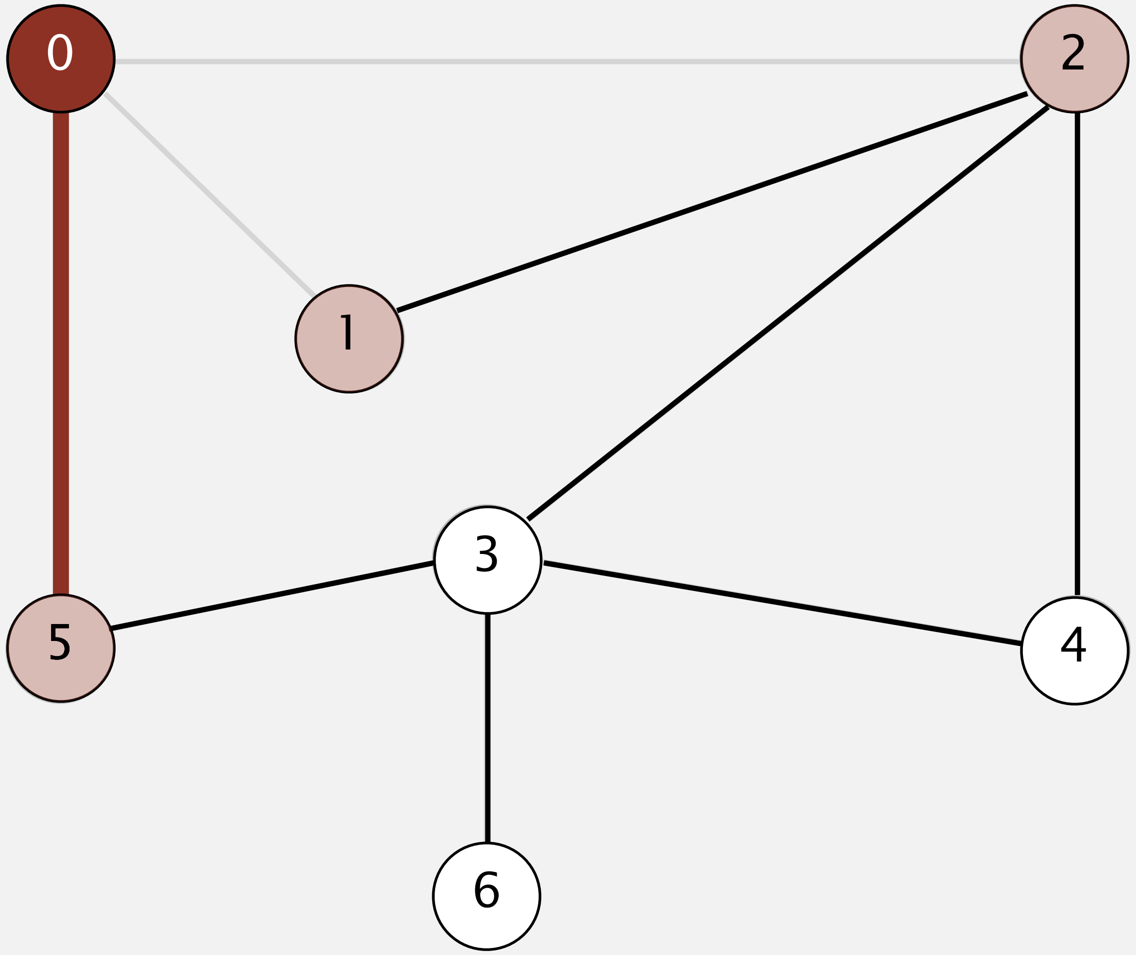
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	–	F
4	–	F
5	–	F
6	–	F

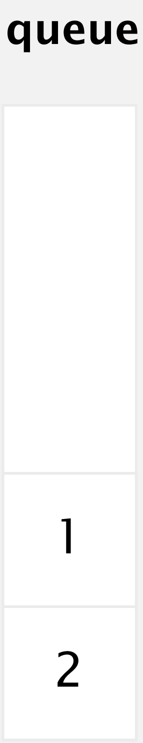
dequeue 0: check 2, **check 1,** check 5

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.





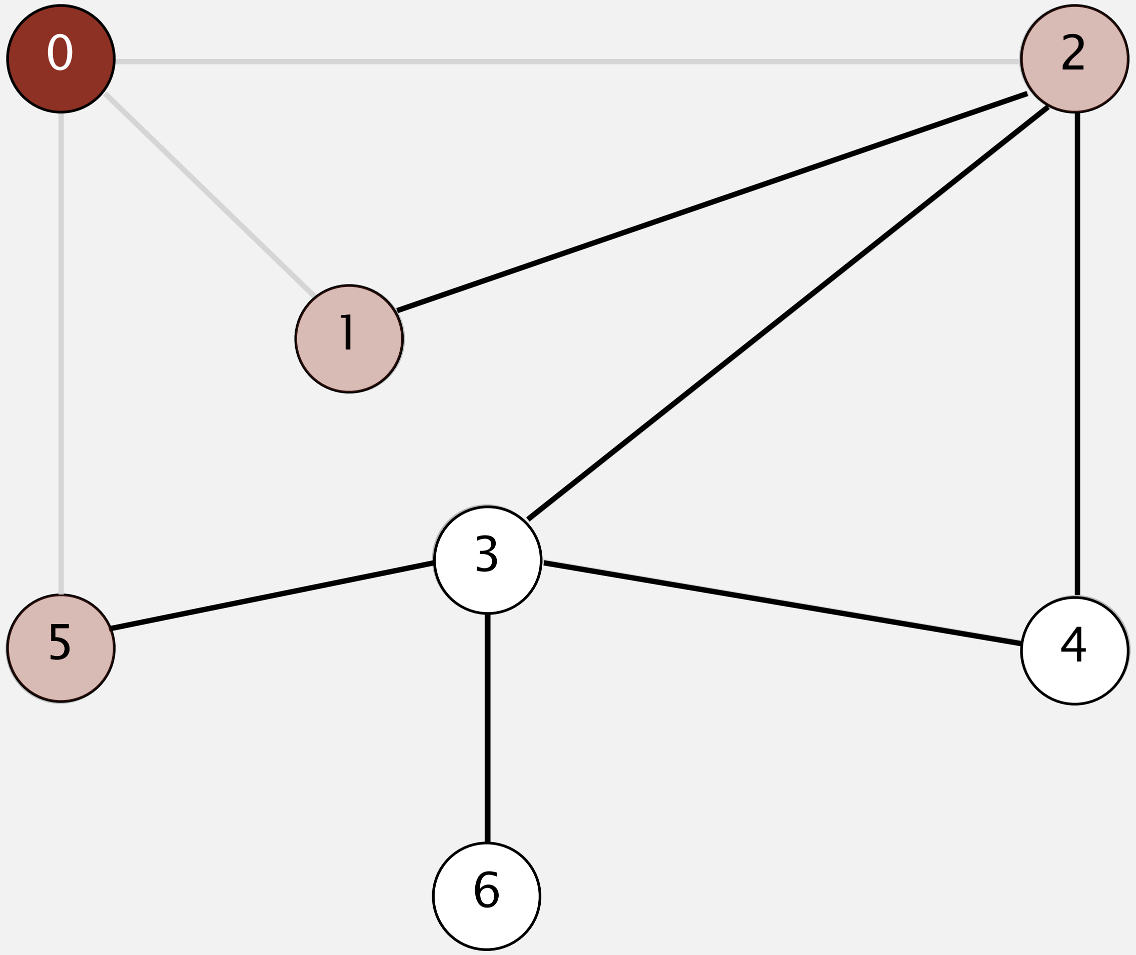
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	–	F
4	–	F
5	0	T
6	–	F

dequeue 0: check 2, check 1, **check 5**

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



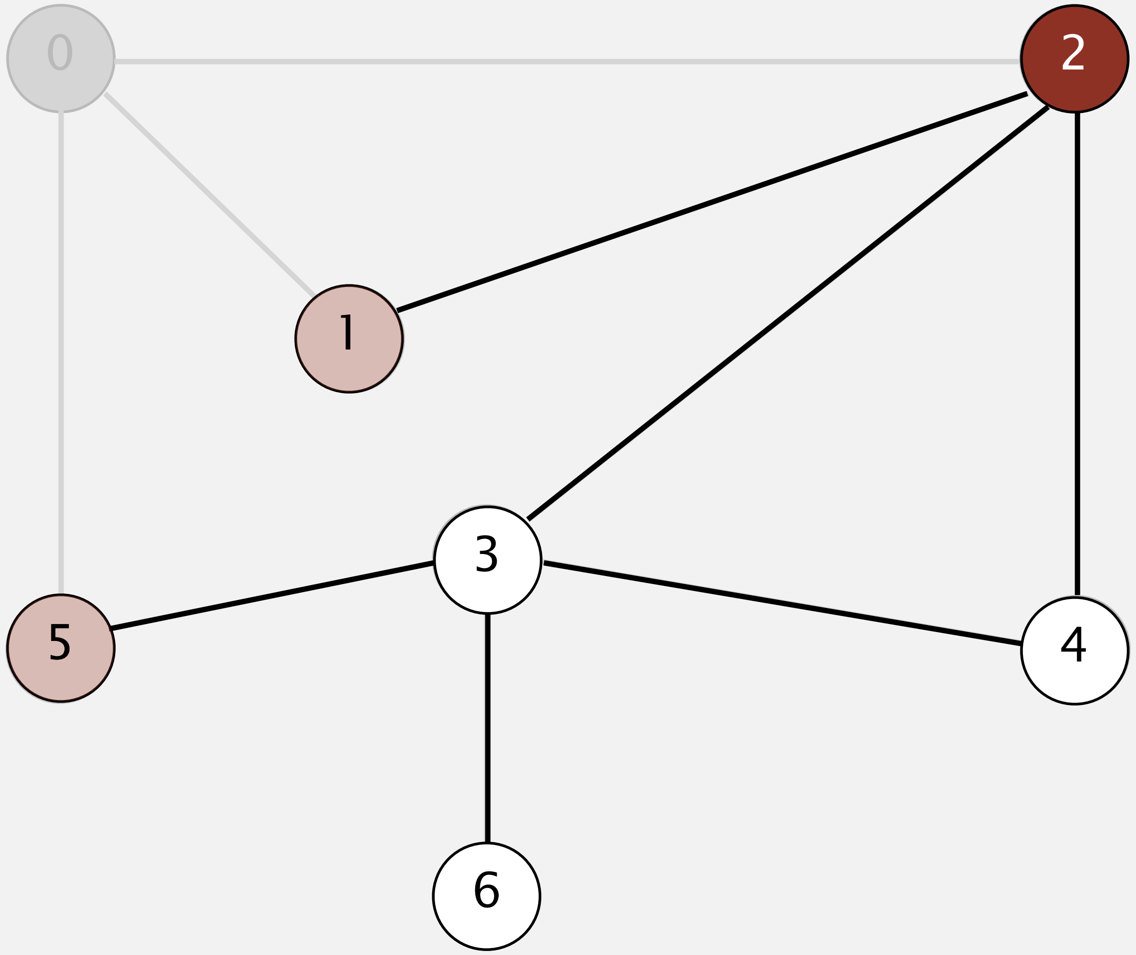
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	–	F
4	–	F
5	0	T
6	–	F

0 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

5
1
2

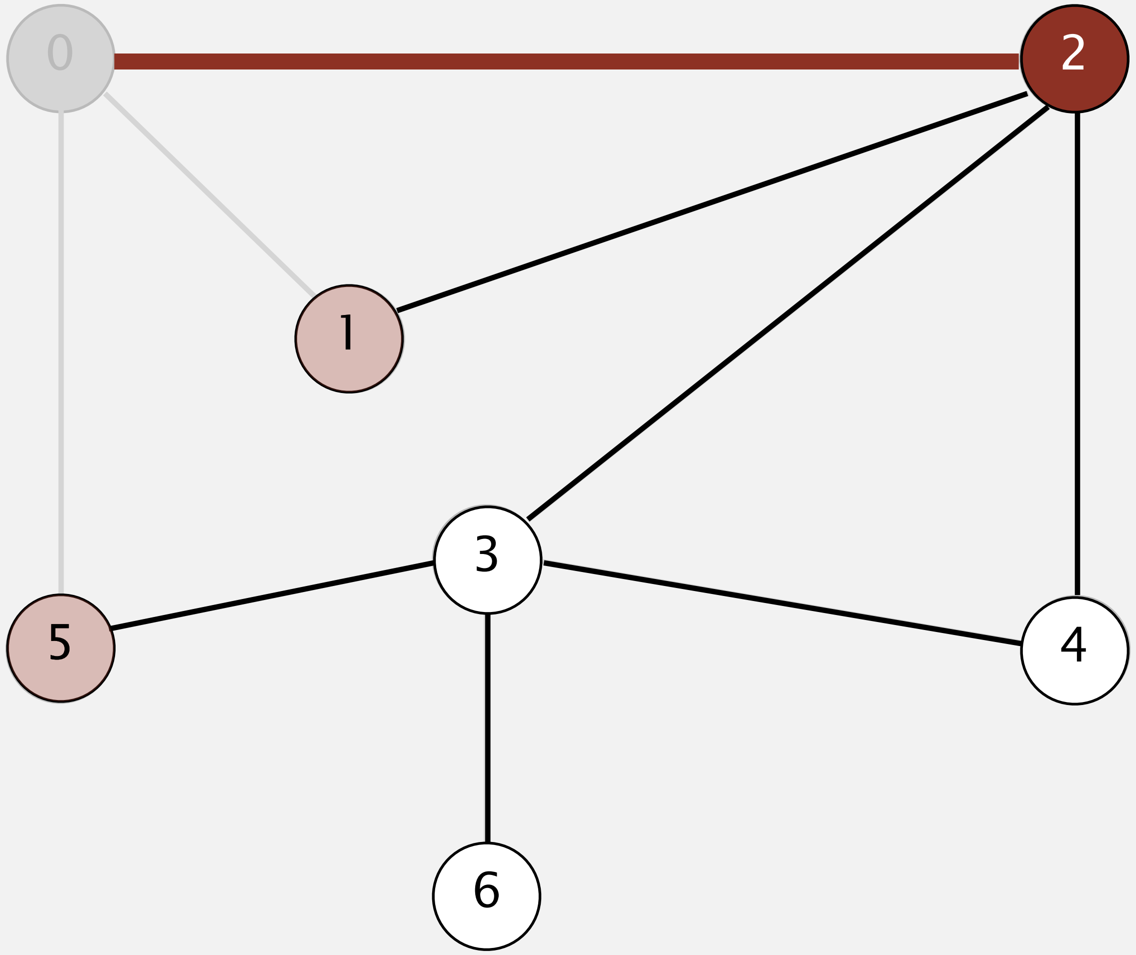
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	–	F
4	–	F
5	0	T
6	–	F

dequeue 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

5
1

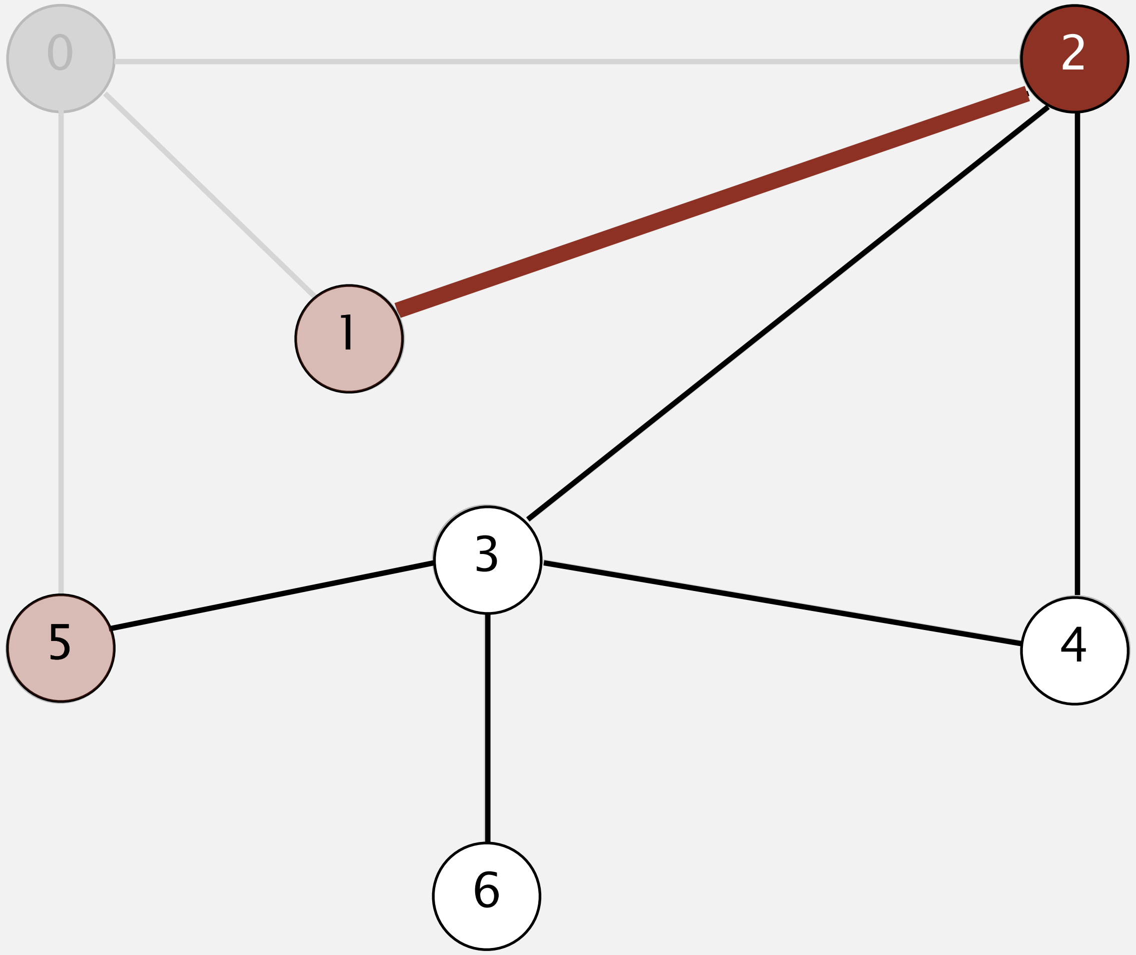
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	–	F
4	–	F
5	0	T
6	–	F

dequeue 2: check 0, check 1, check 3, check 4

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

5

1

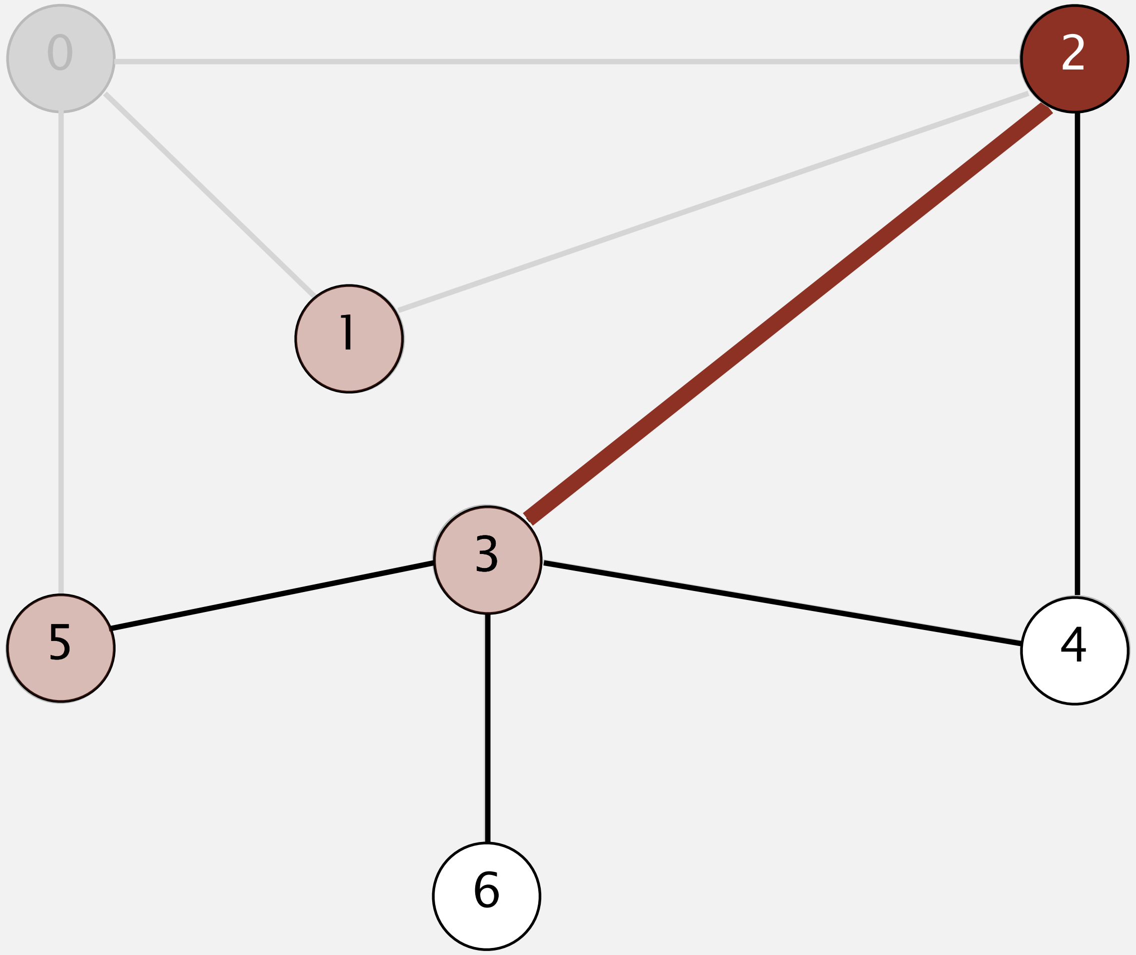
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	–	F
4	–	F
5	0	T
6	–	F

dequeue 2: check 0, **check 1,** check 3, check 4

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

5
1

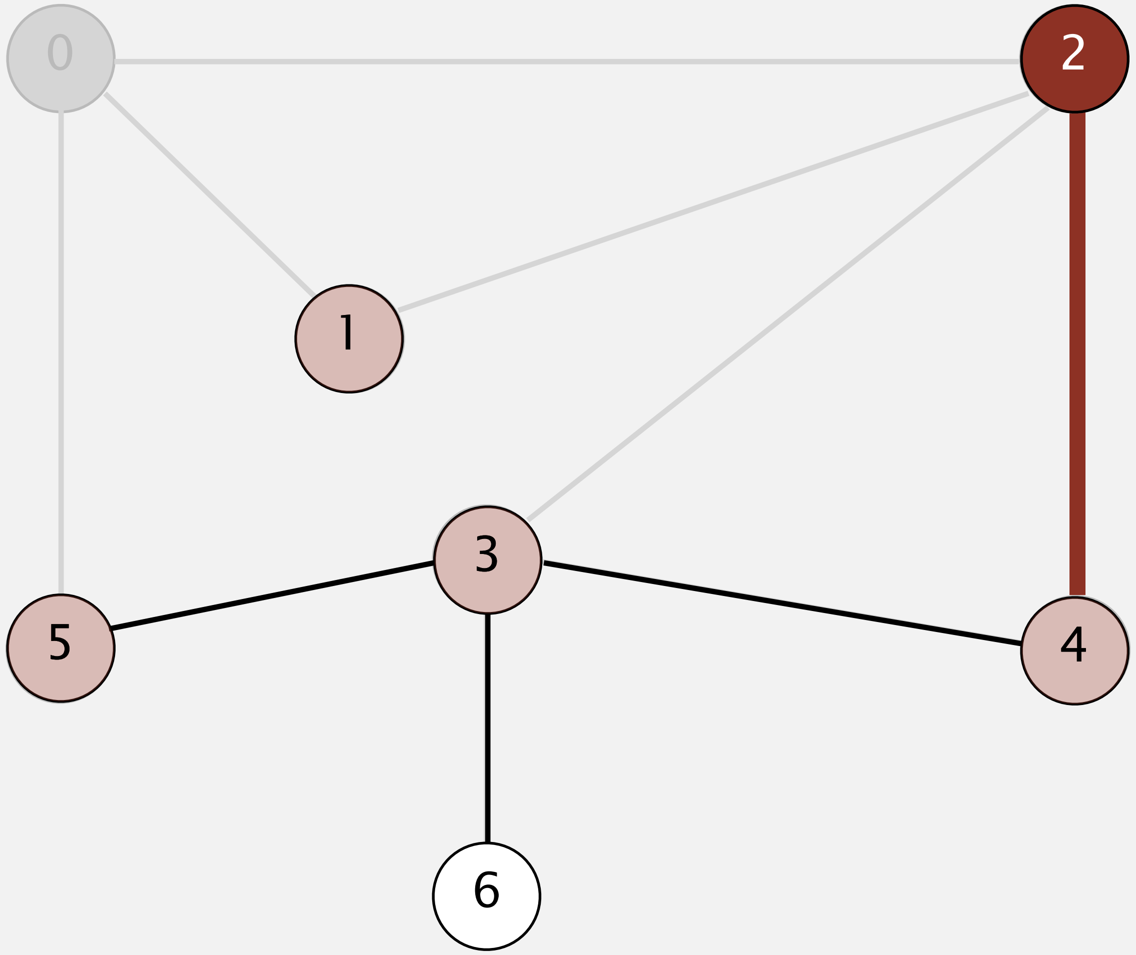
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	–	F
5	0	T
6	–	F

dequeue 2: check 0, check 1, **check 3,** check 4

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

3
5
1

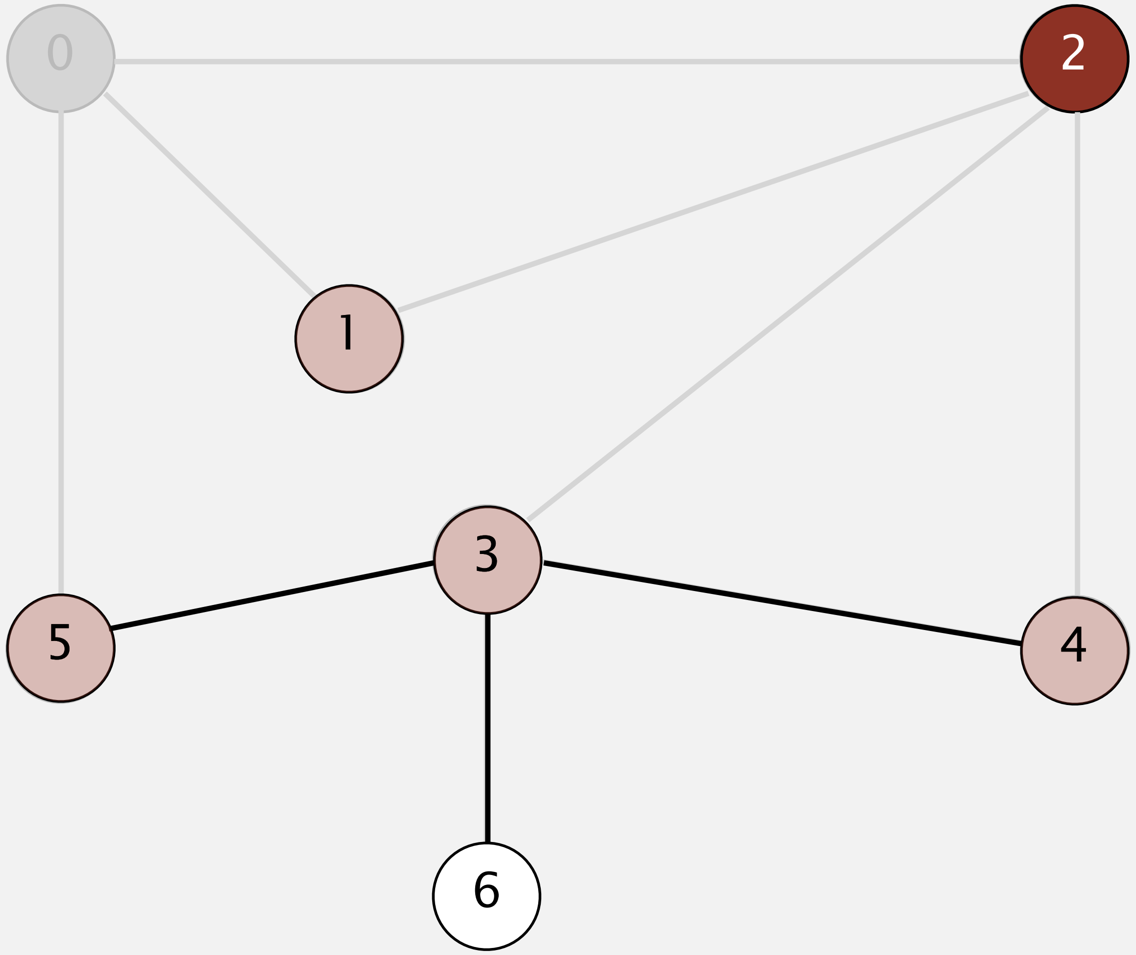
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 2: check 0, check 1, check 3, **check 4**

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3
5
1

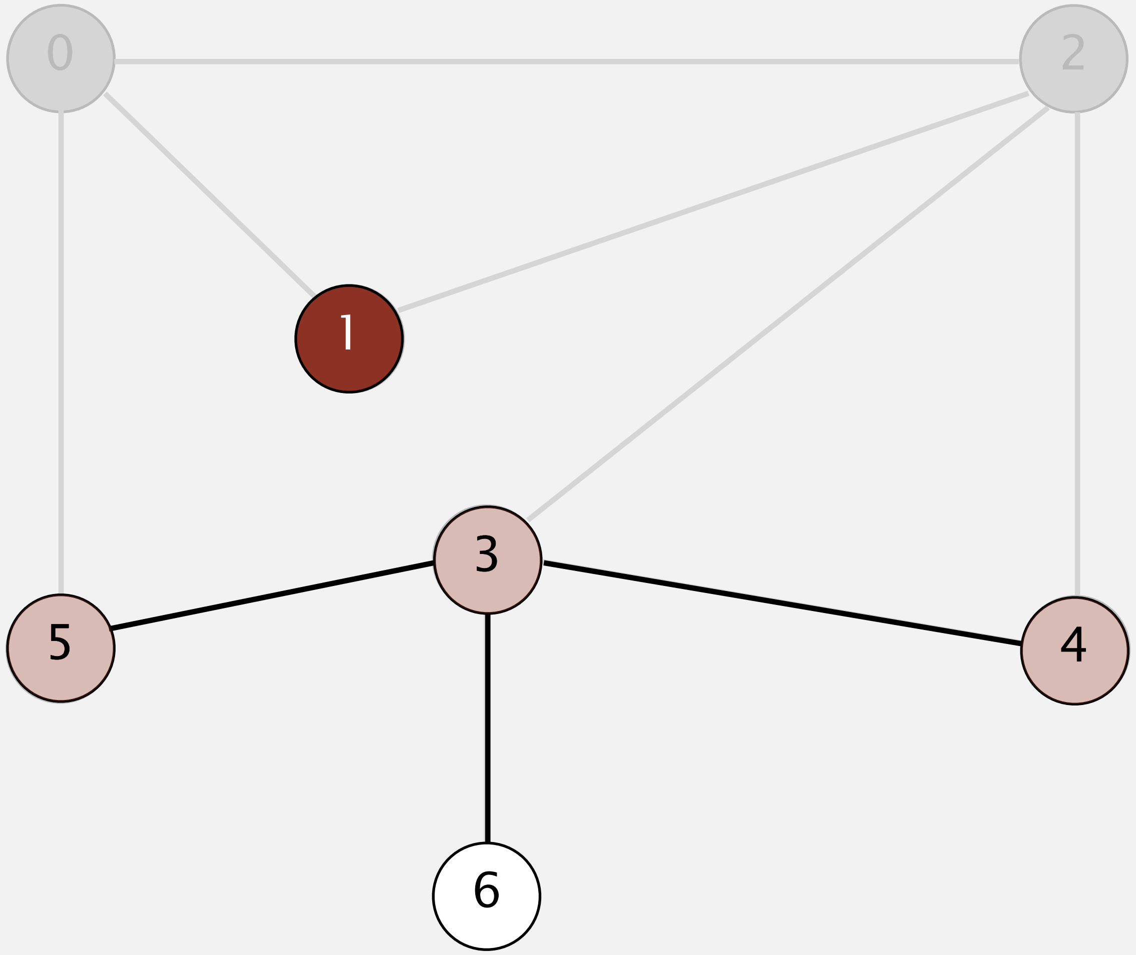
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

2 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3
5
1

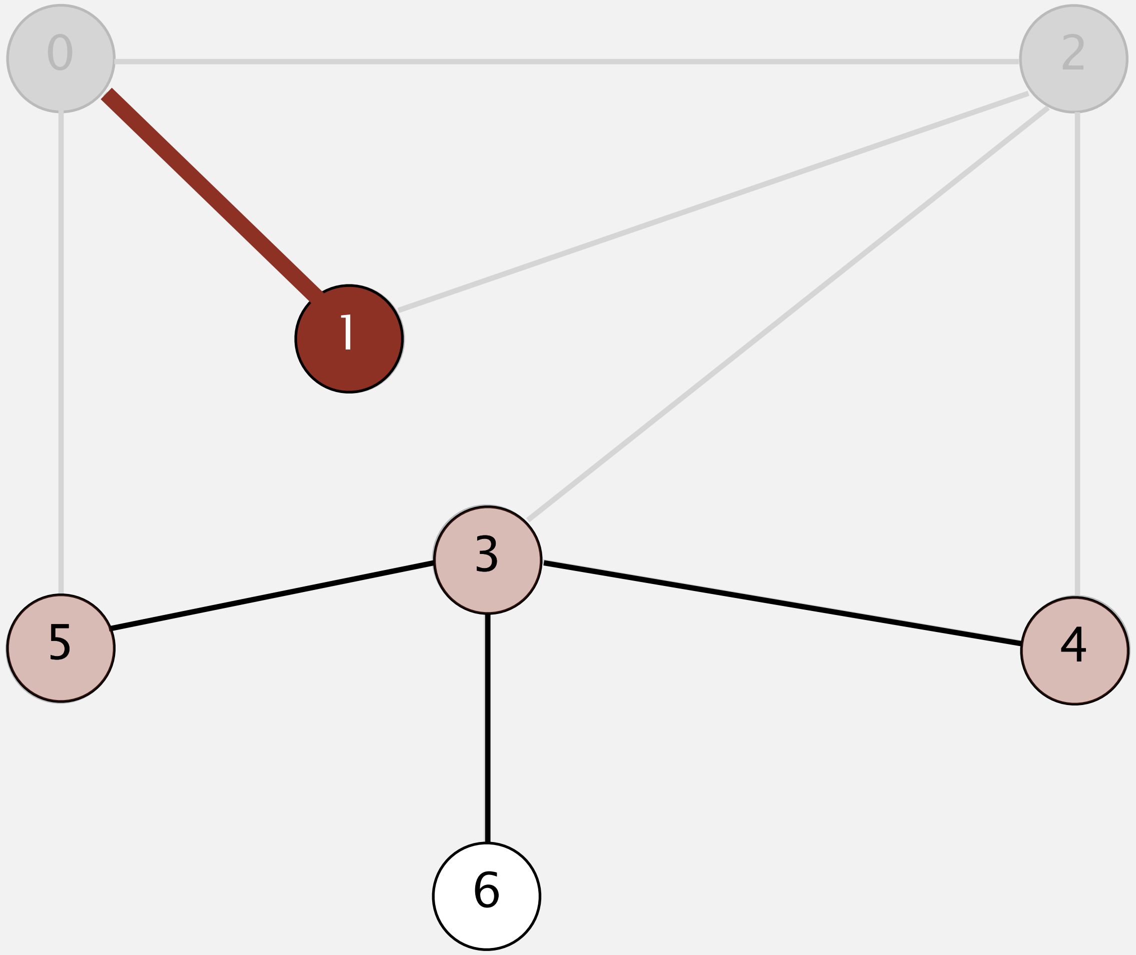
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 1

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3
5

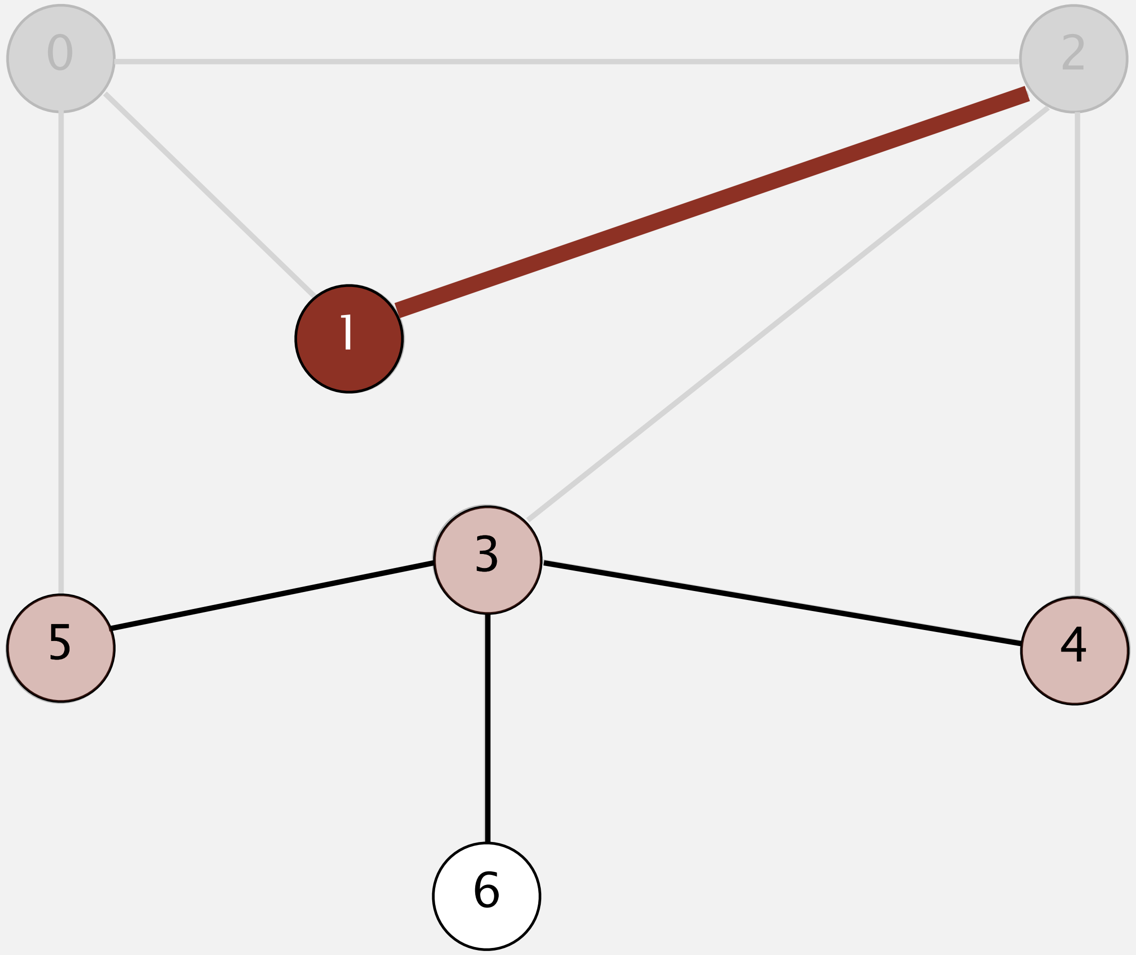
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 1: check 0, check 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3
5

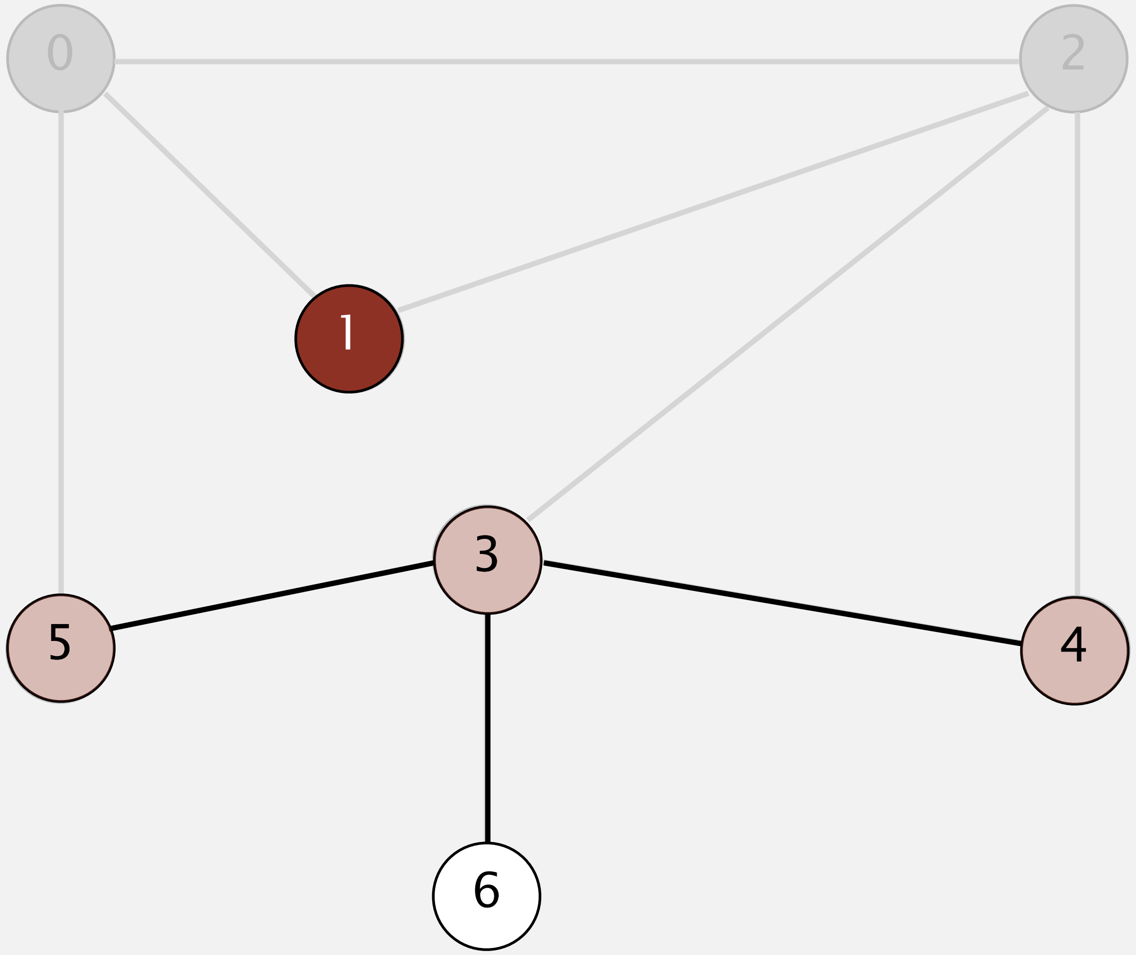
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 1: check 0, check 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue
4
3
5

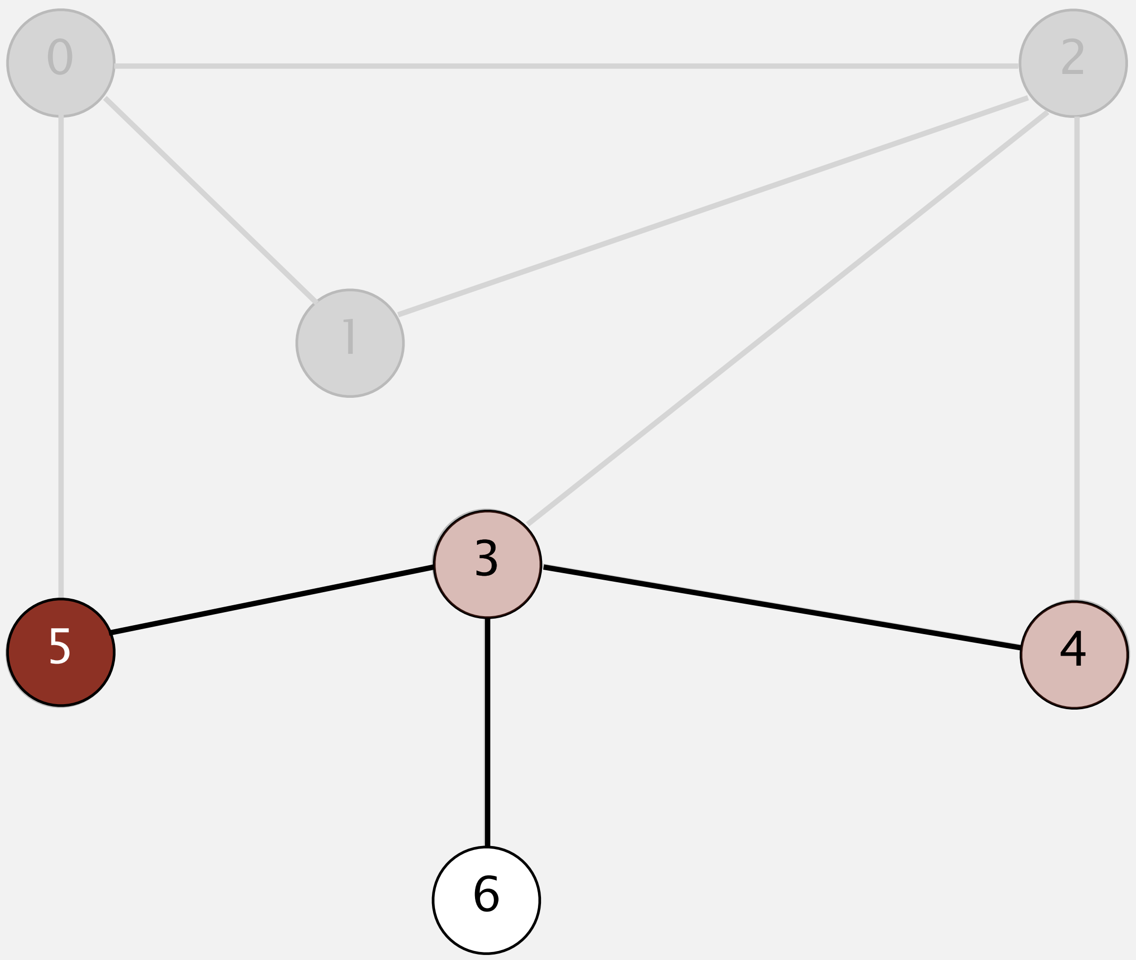
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

1 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3
5

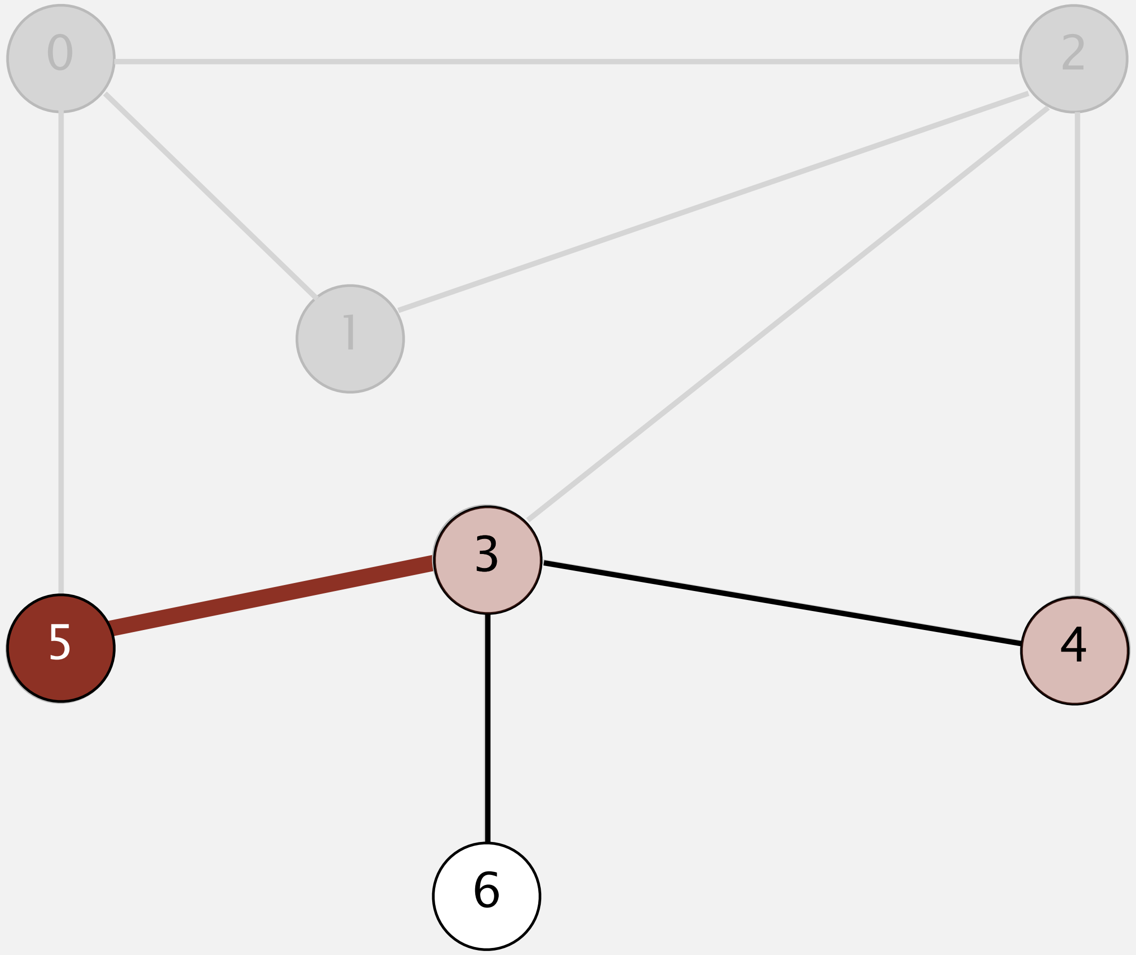
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 5

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3

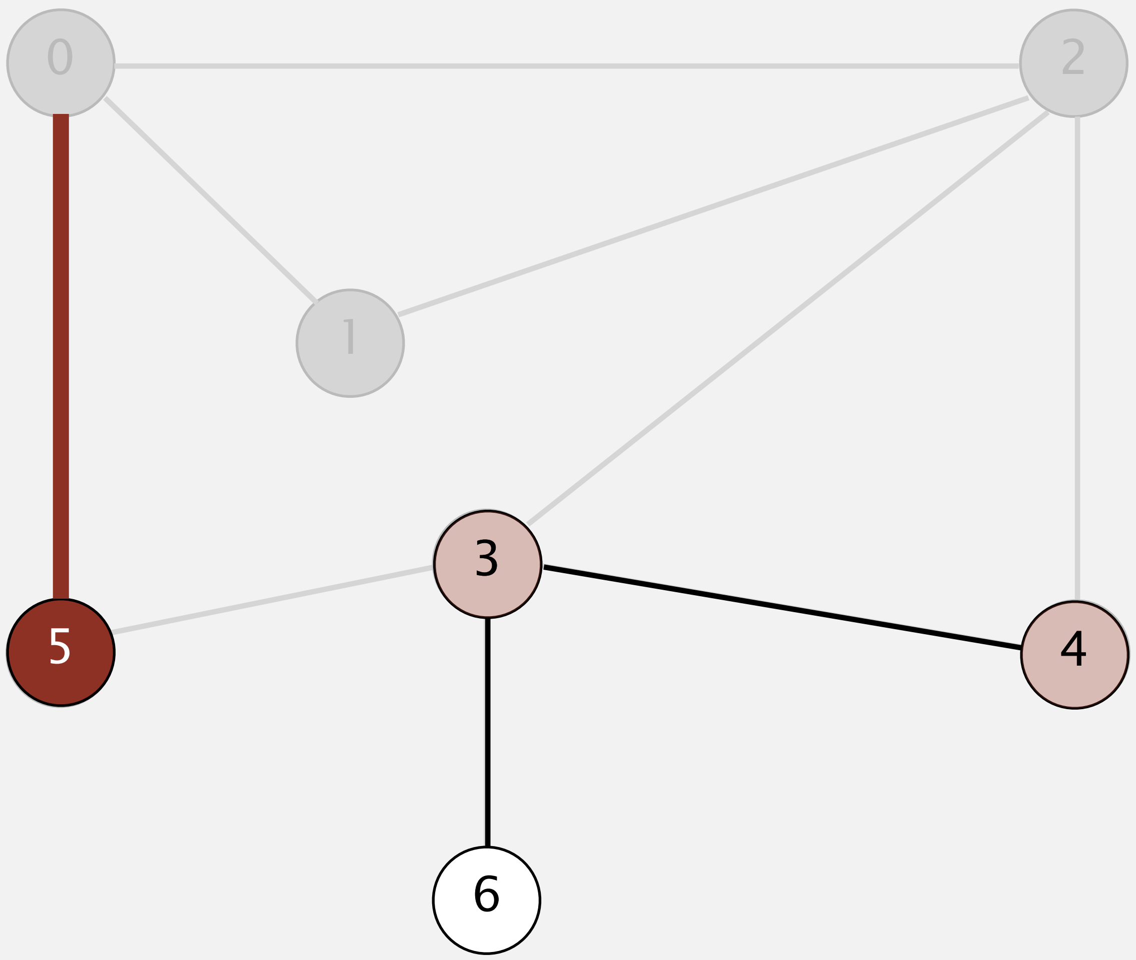
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 5: check 3, check 0

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3

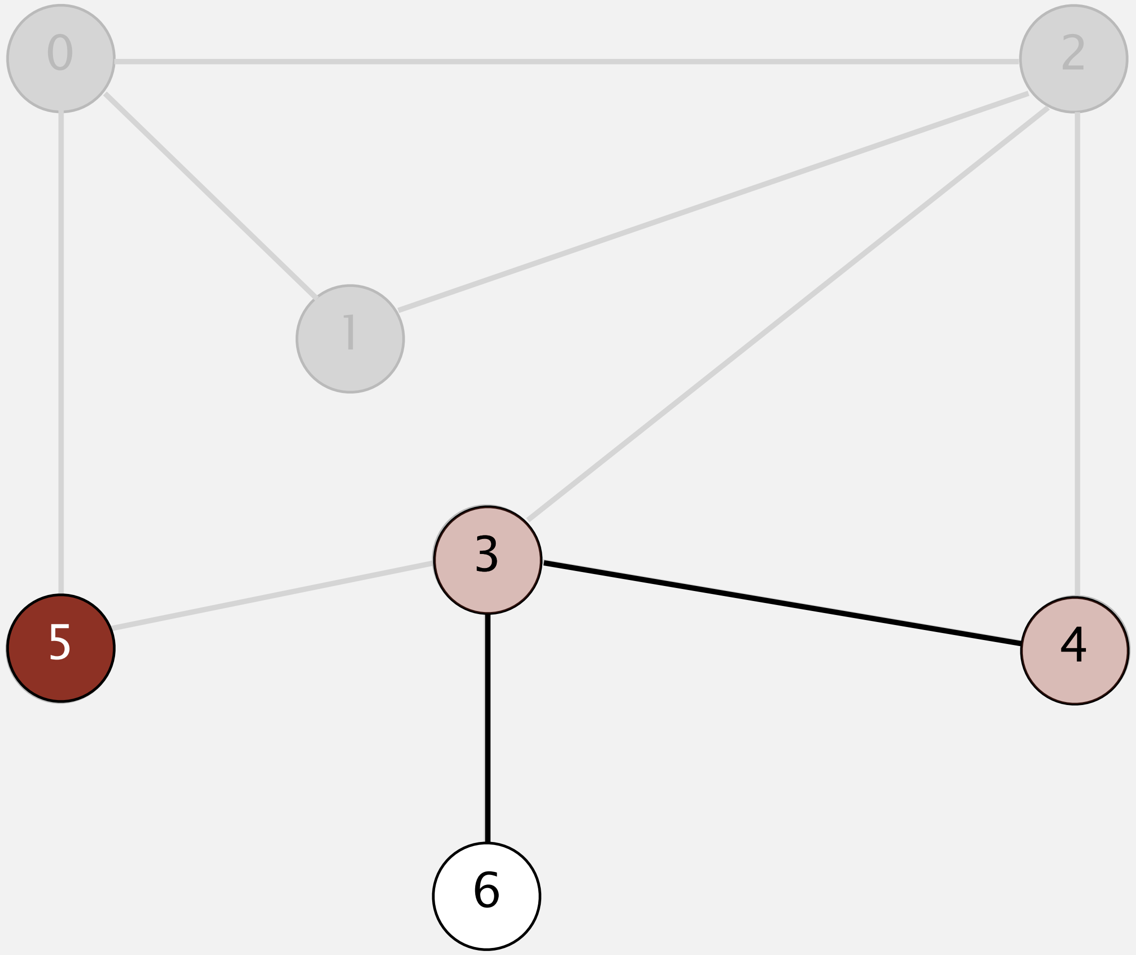
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 5: check 3, **check 0**

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4
3

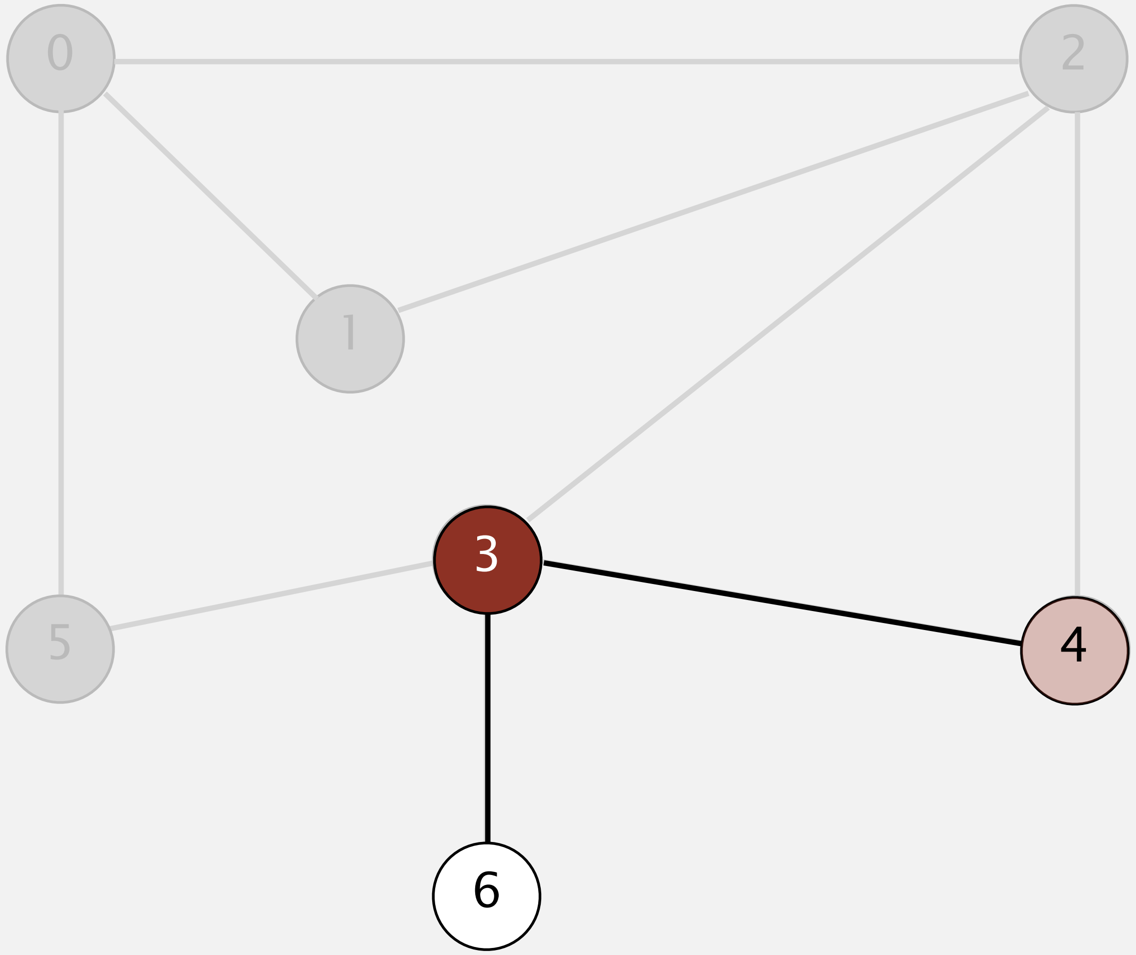
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

5 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4

3

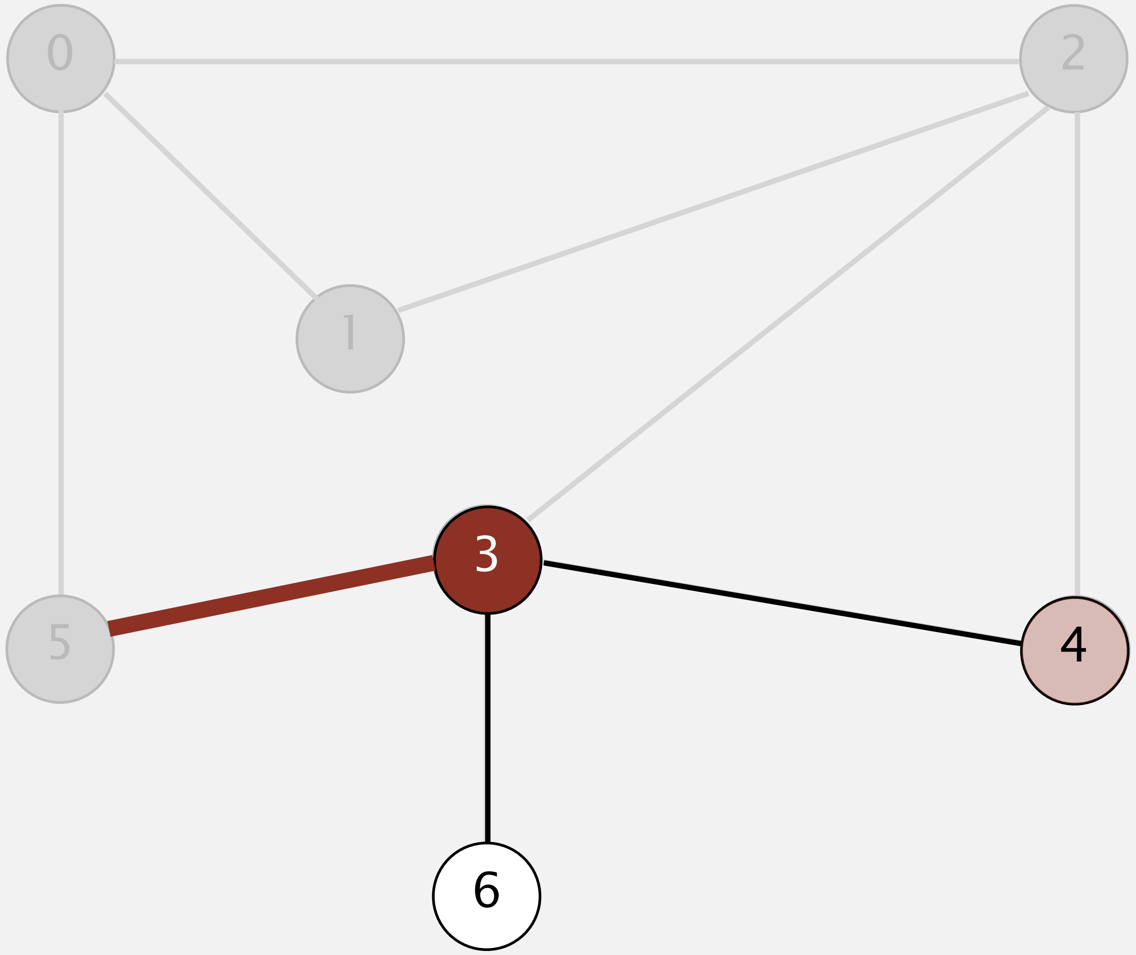
v	**edgeTo[]**	**marked[]**
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 3

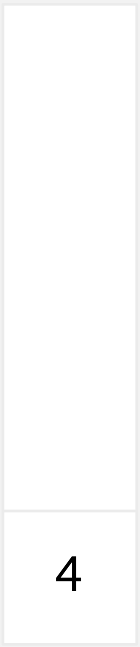
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue



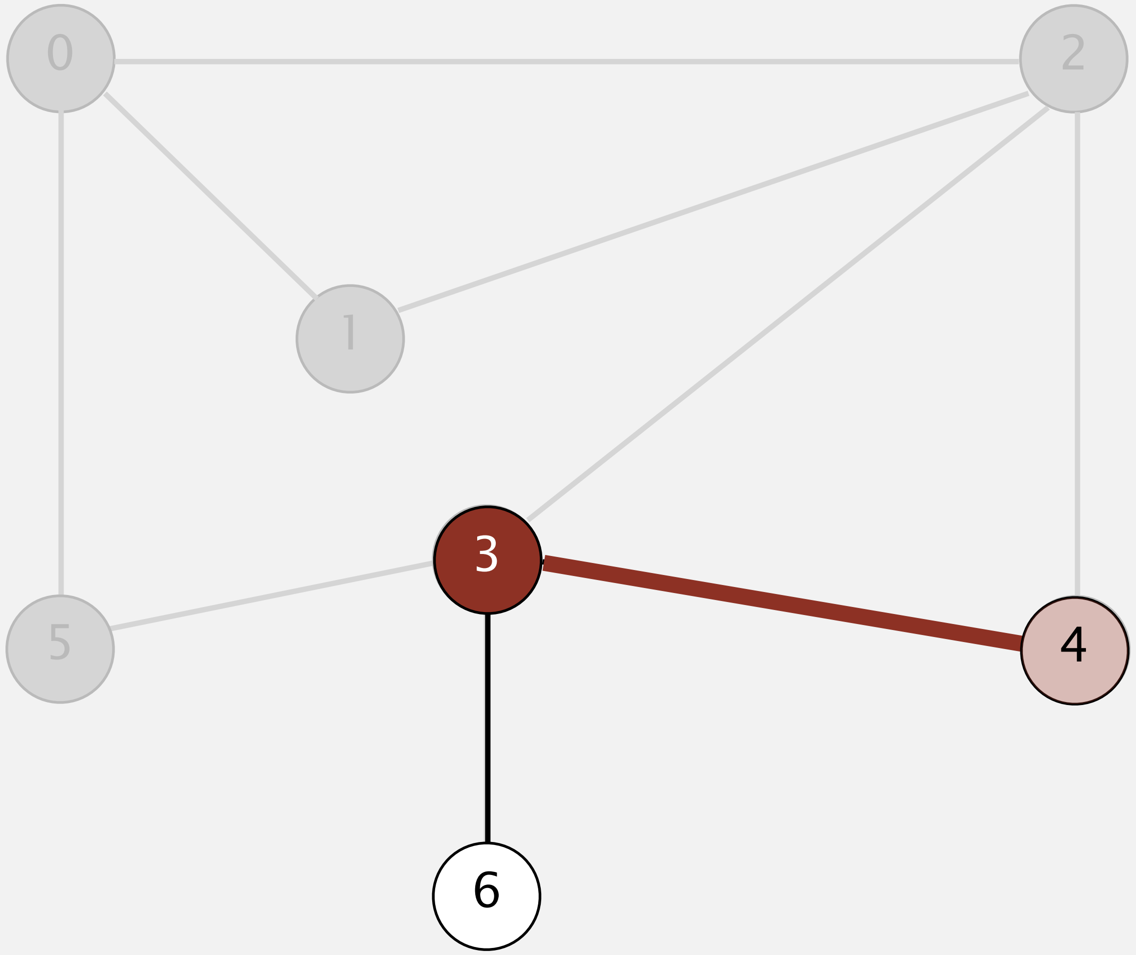
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 3: check 5, check 4, check 2, check 6

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4

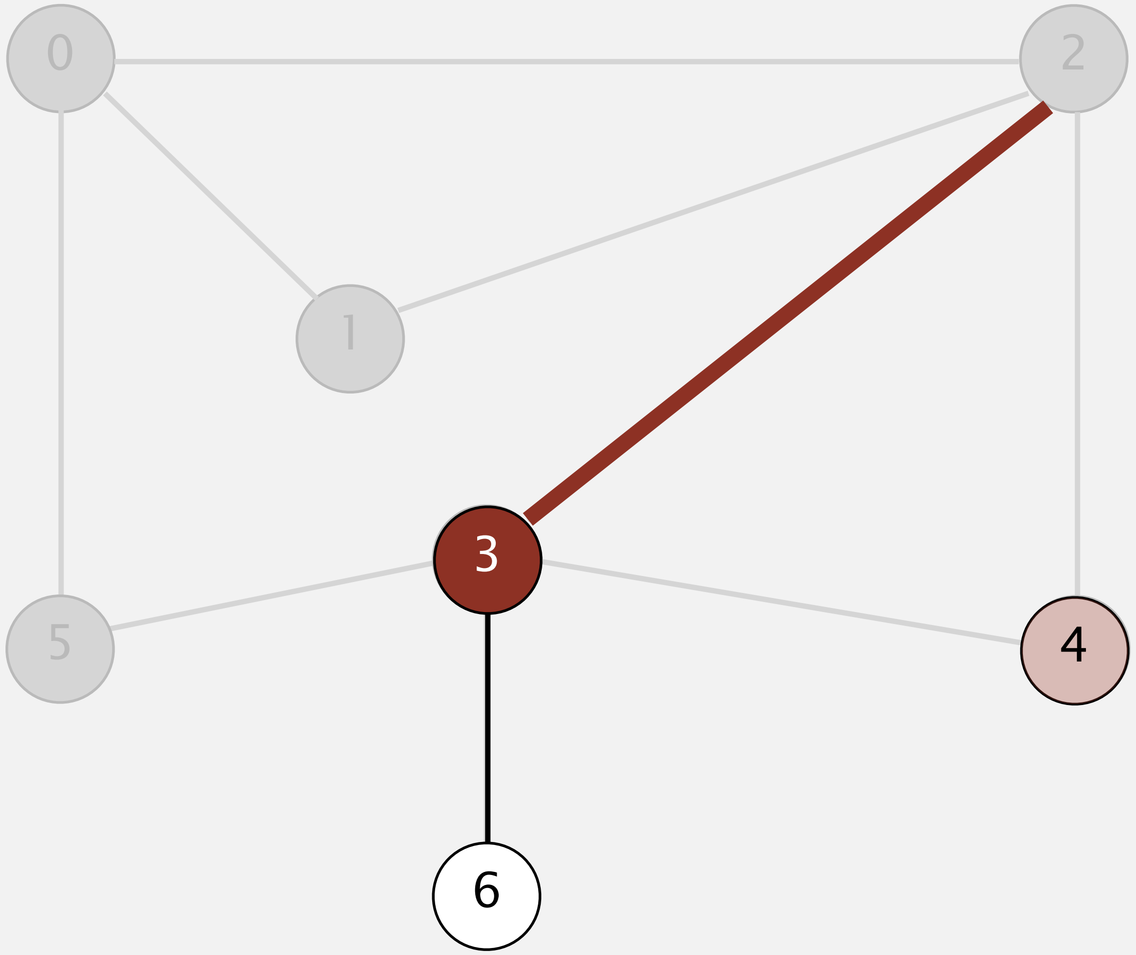
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 3: check 5, **check 4,** check 2, check 6

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4

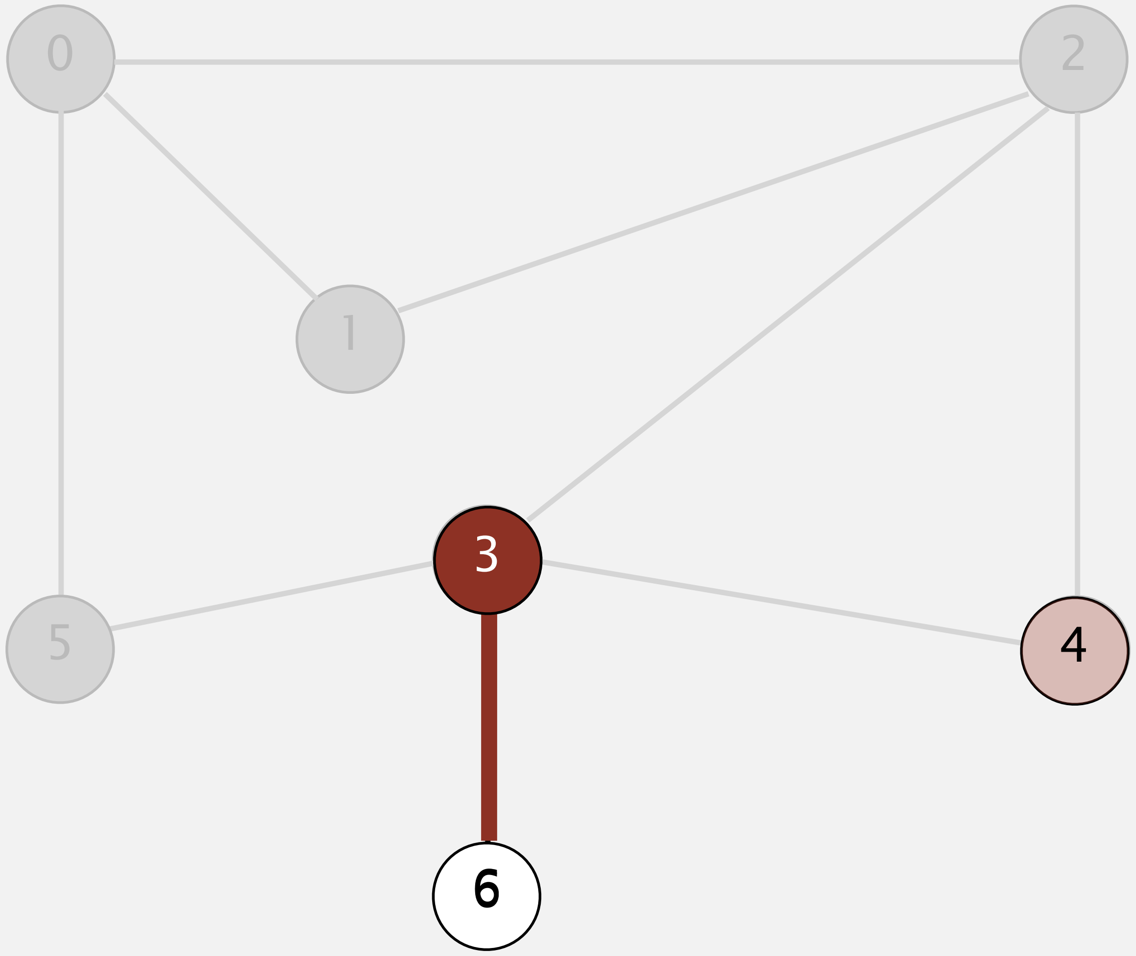
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	–	F

dequeue 3: check 5, check 4, **check 2,** check 6

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

4

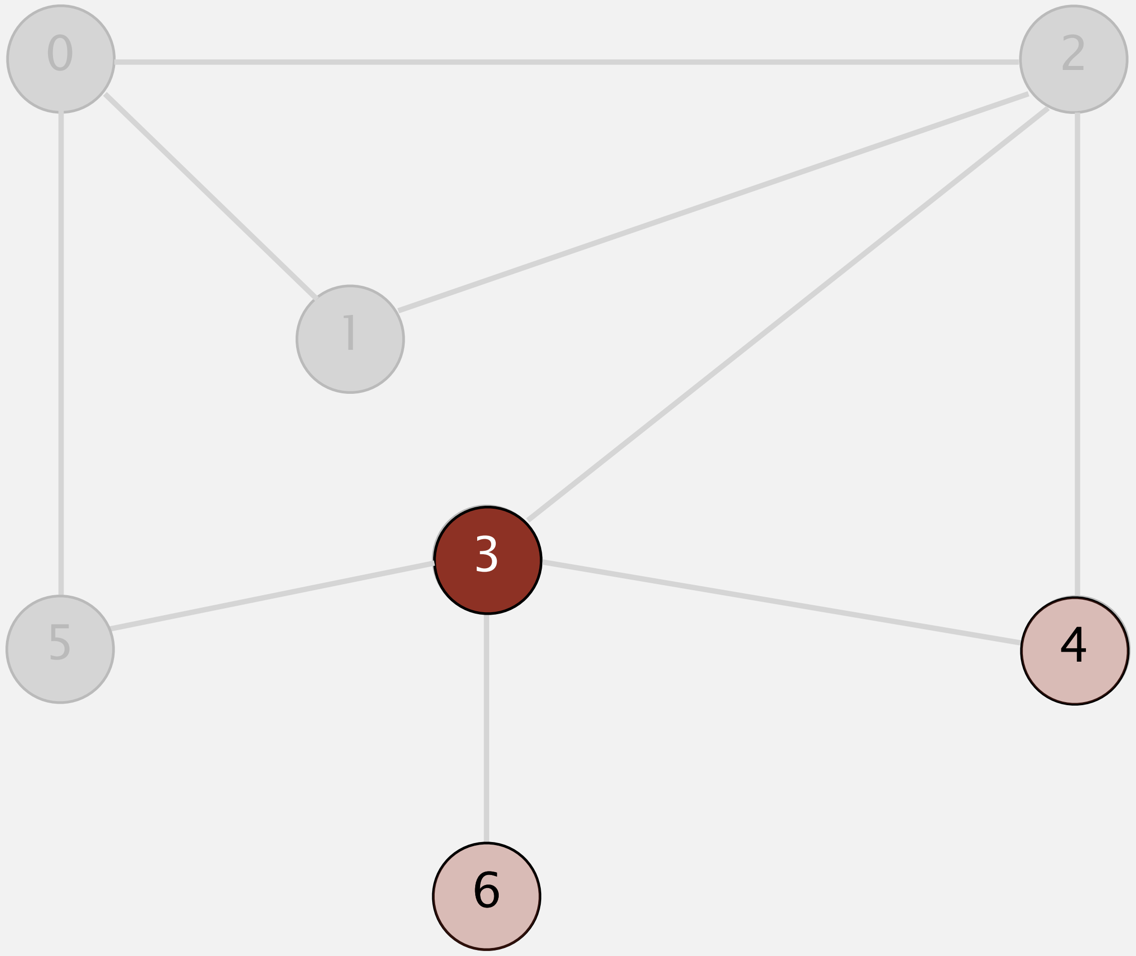
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

dequeue 3: check 5, check 4, check 2, **check 6**

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

6
4

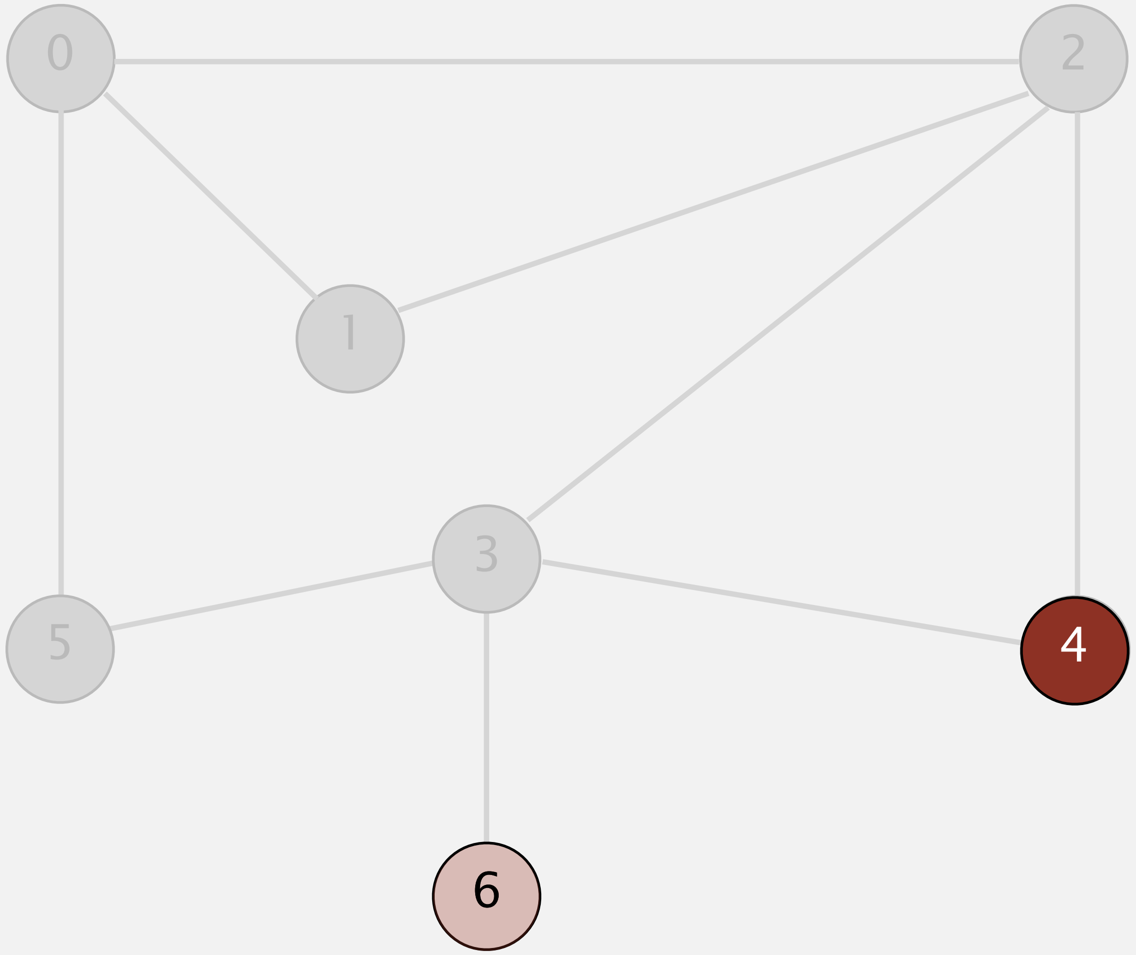
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

3 done

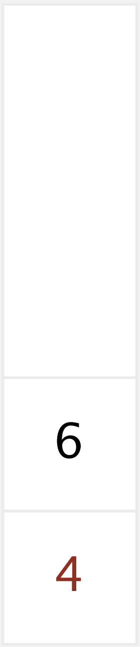
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue



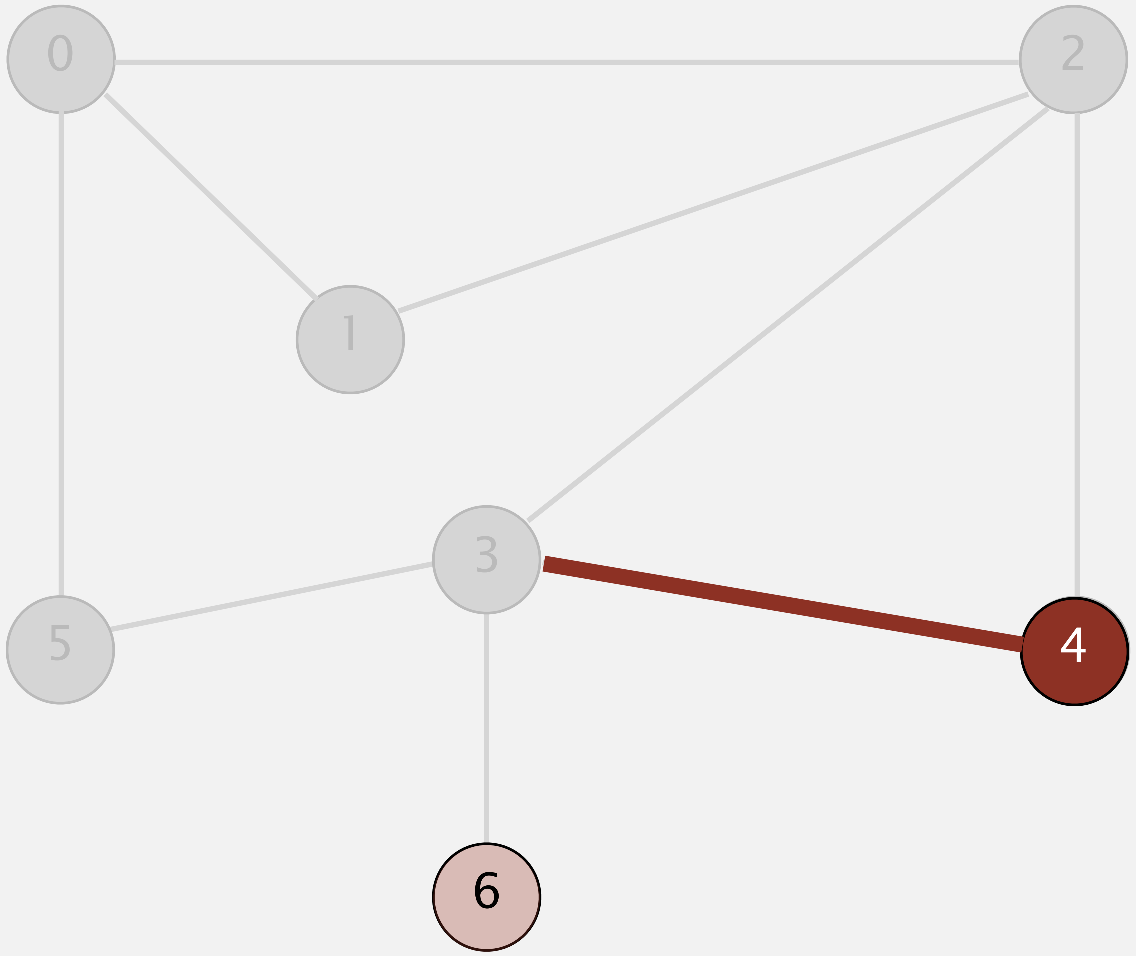
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

dequeue 4

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

6

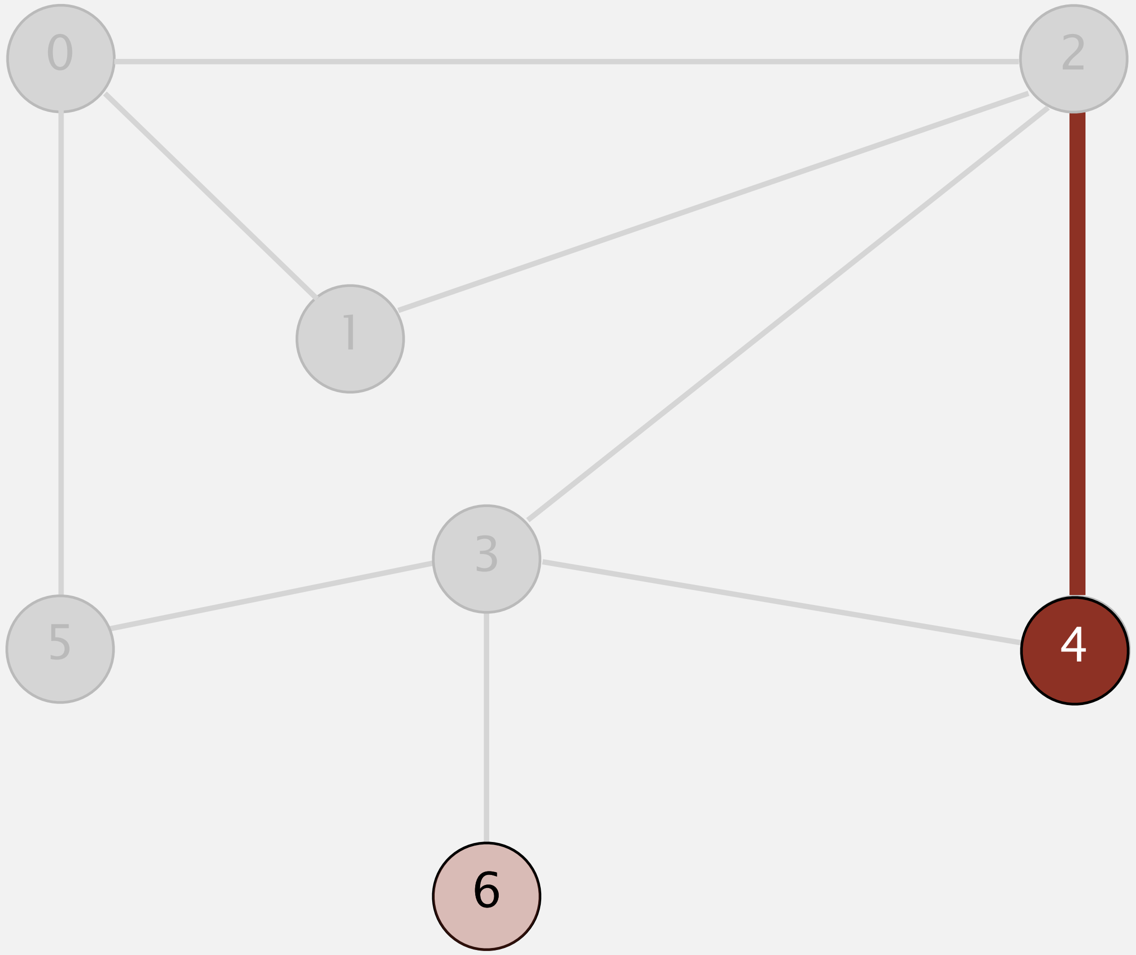
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

dequeue 4: check 3, check 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

6

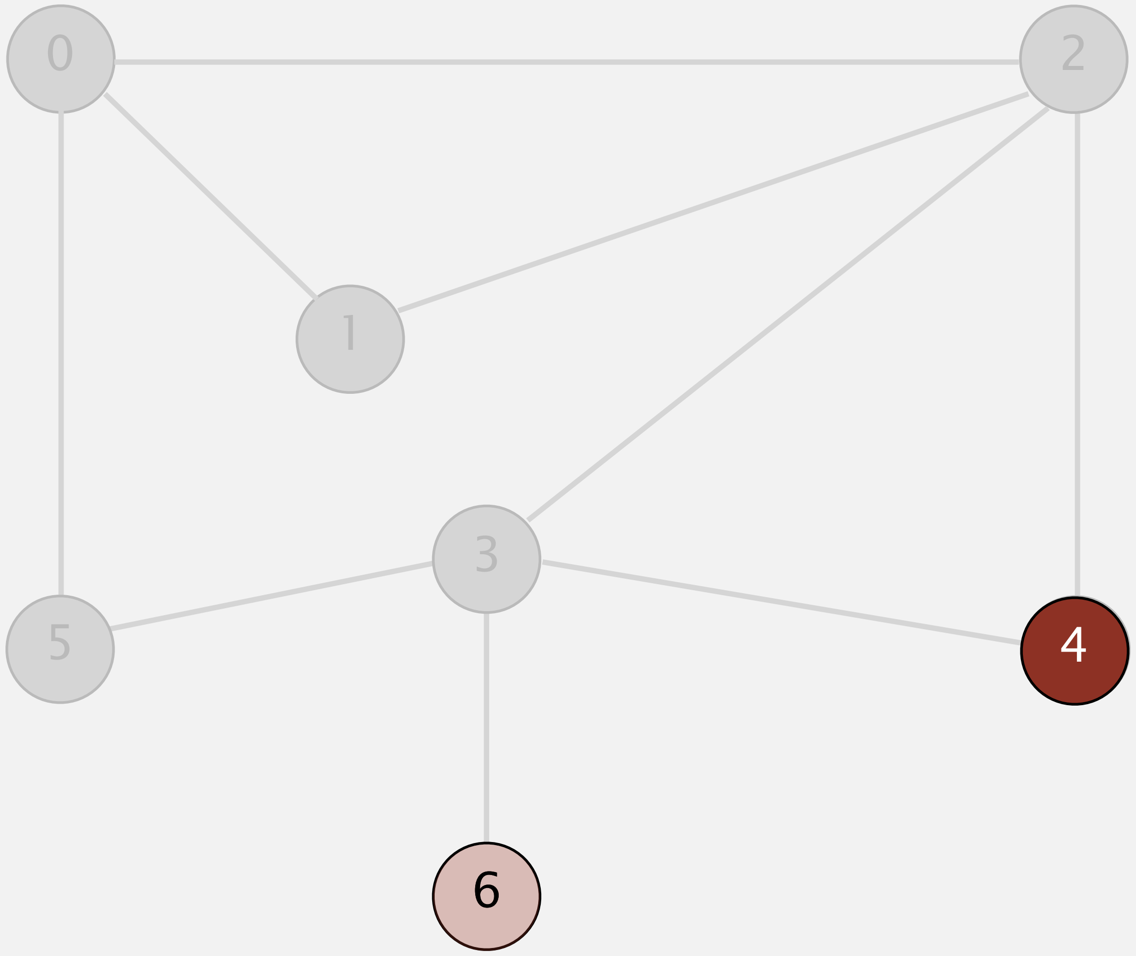
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

dequeue 4: check 3, check 2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

6

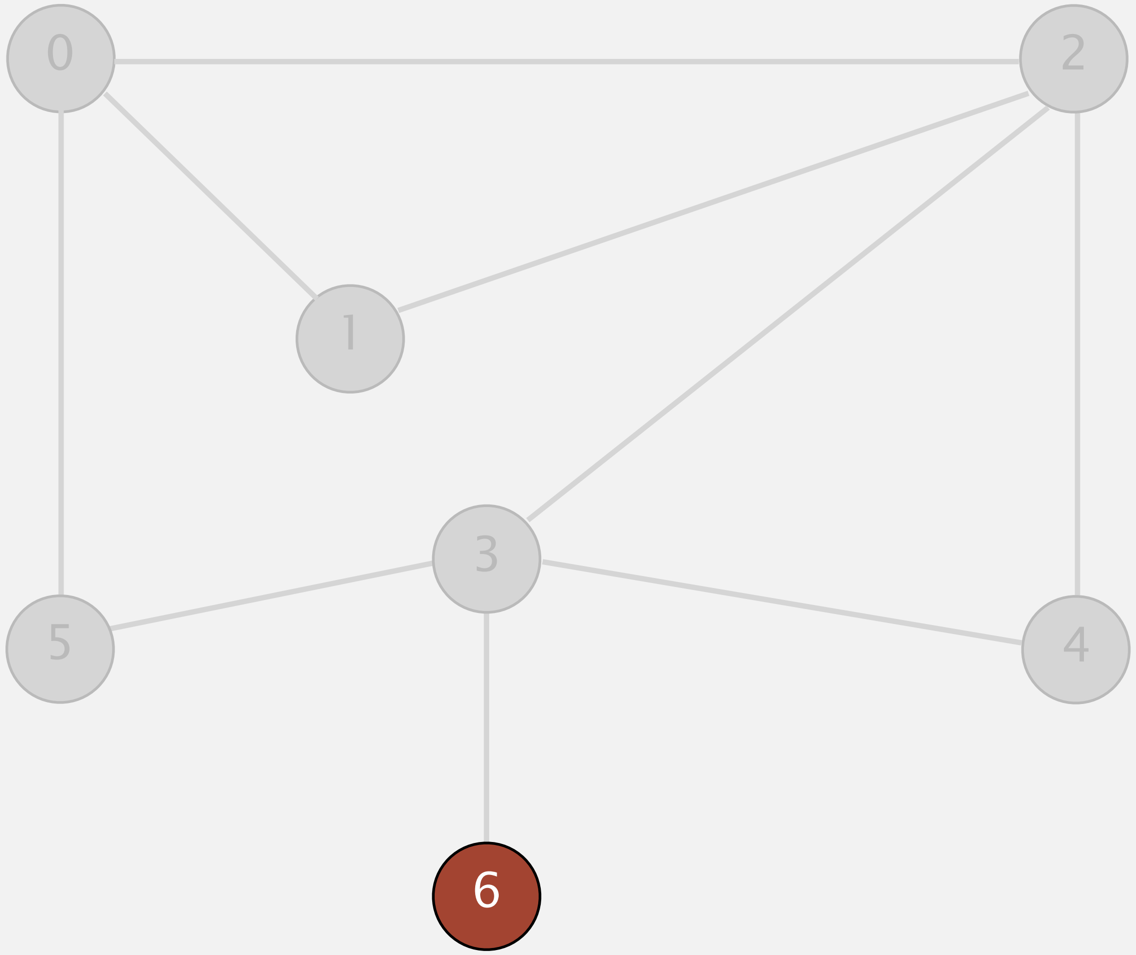
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

4 done

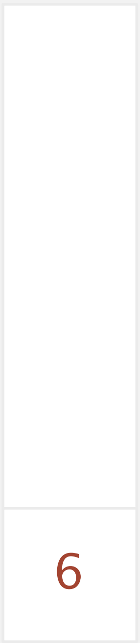
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue



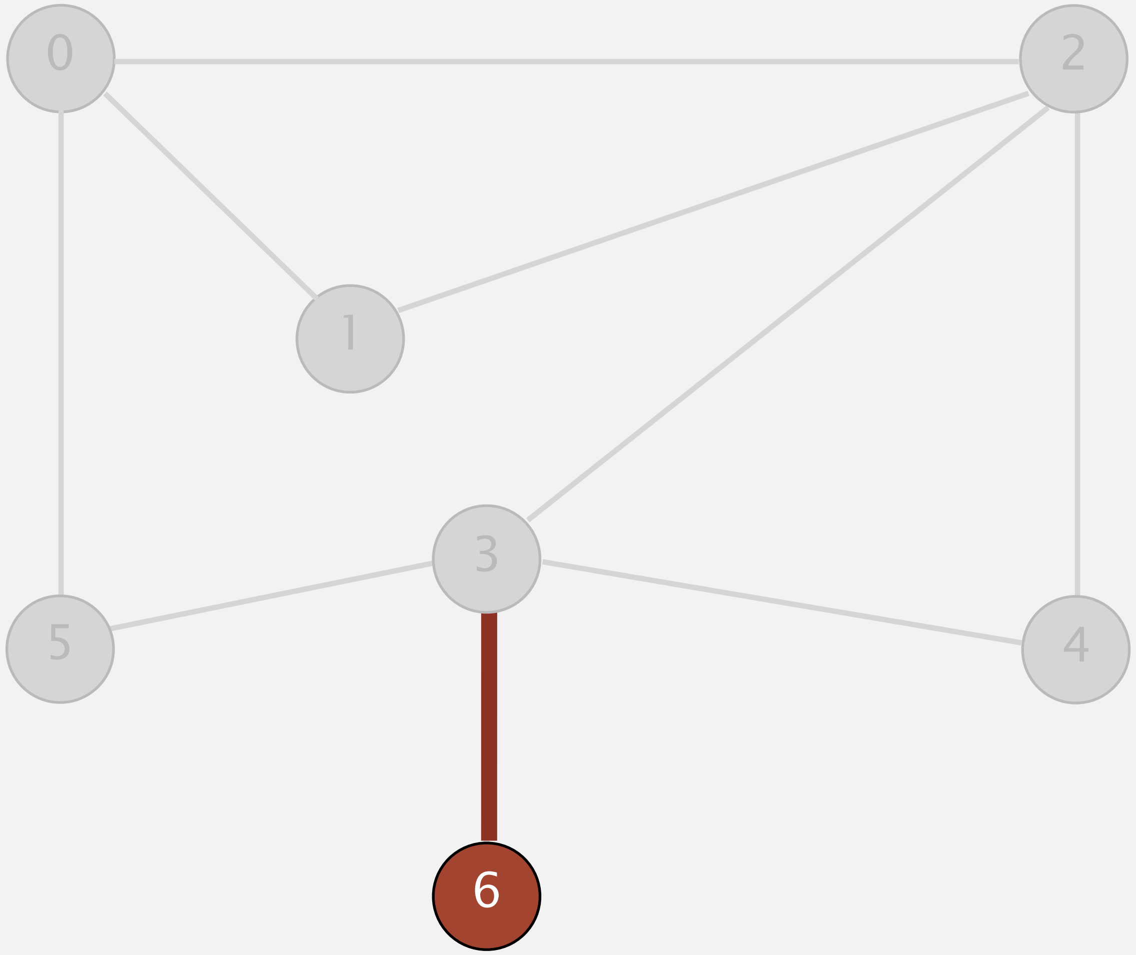
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

dequeue 6

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

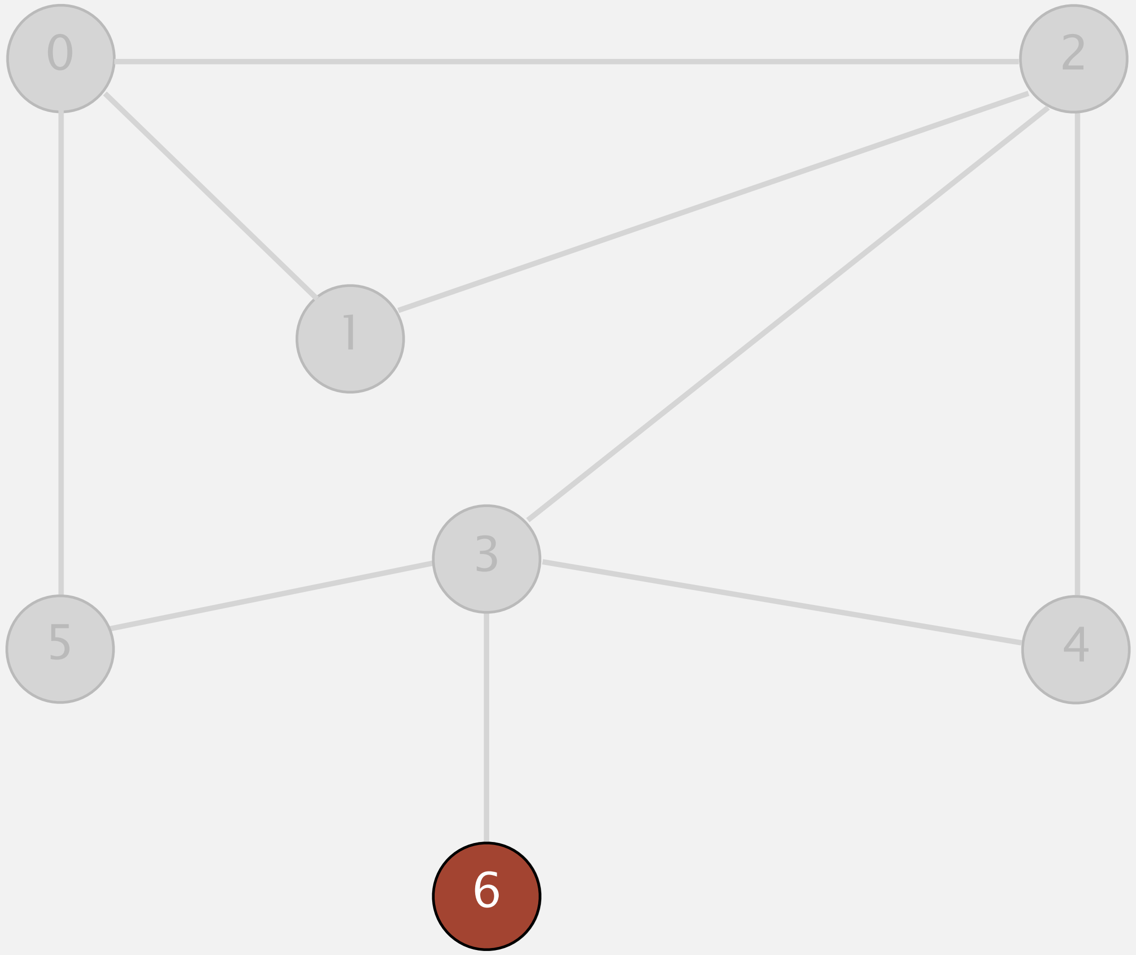
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

dequeue 6 : check 3

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



queue

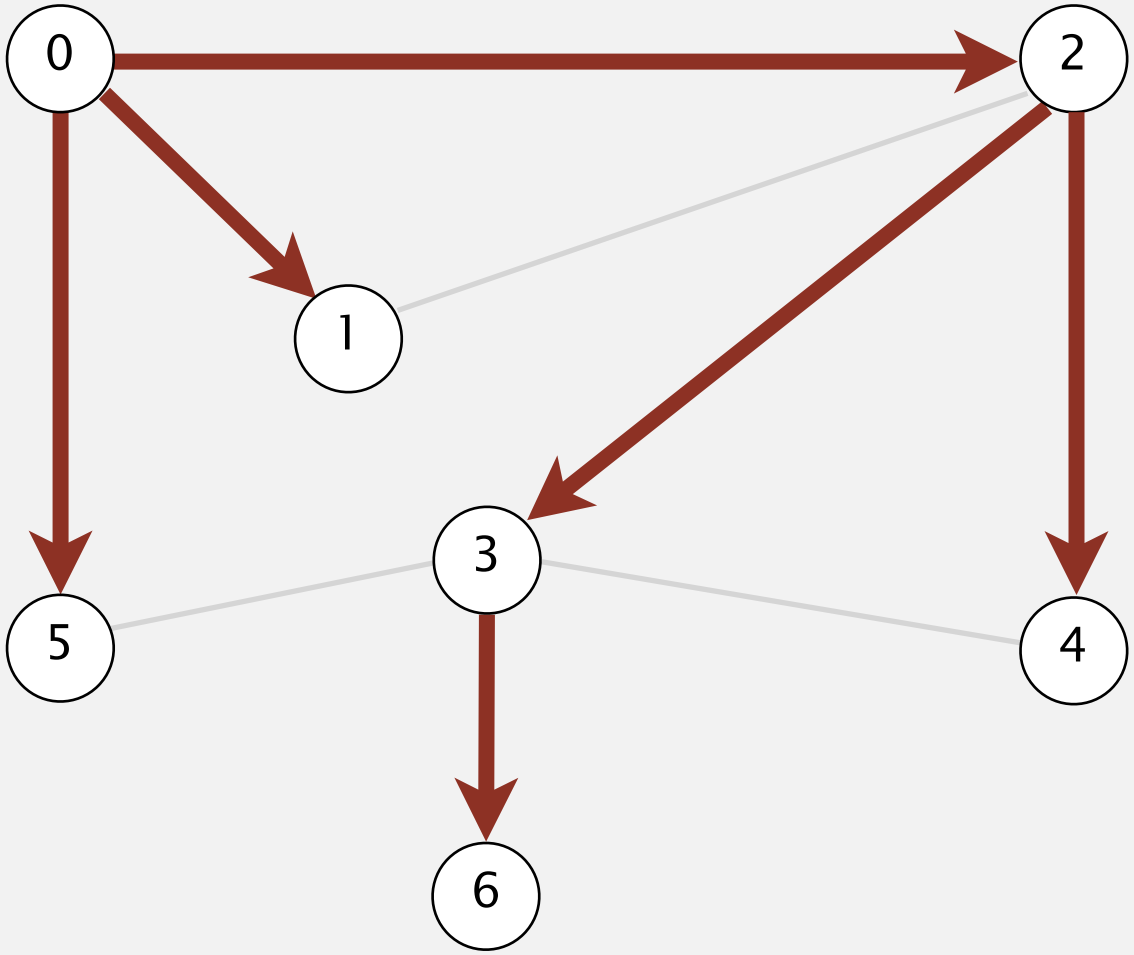
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

6 done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



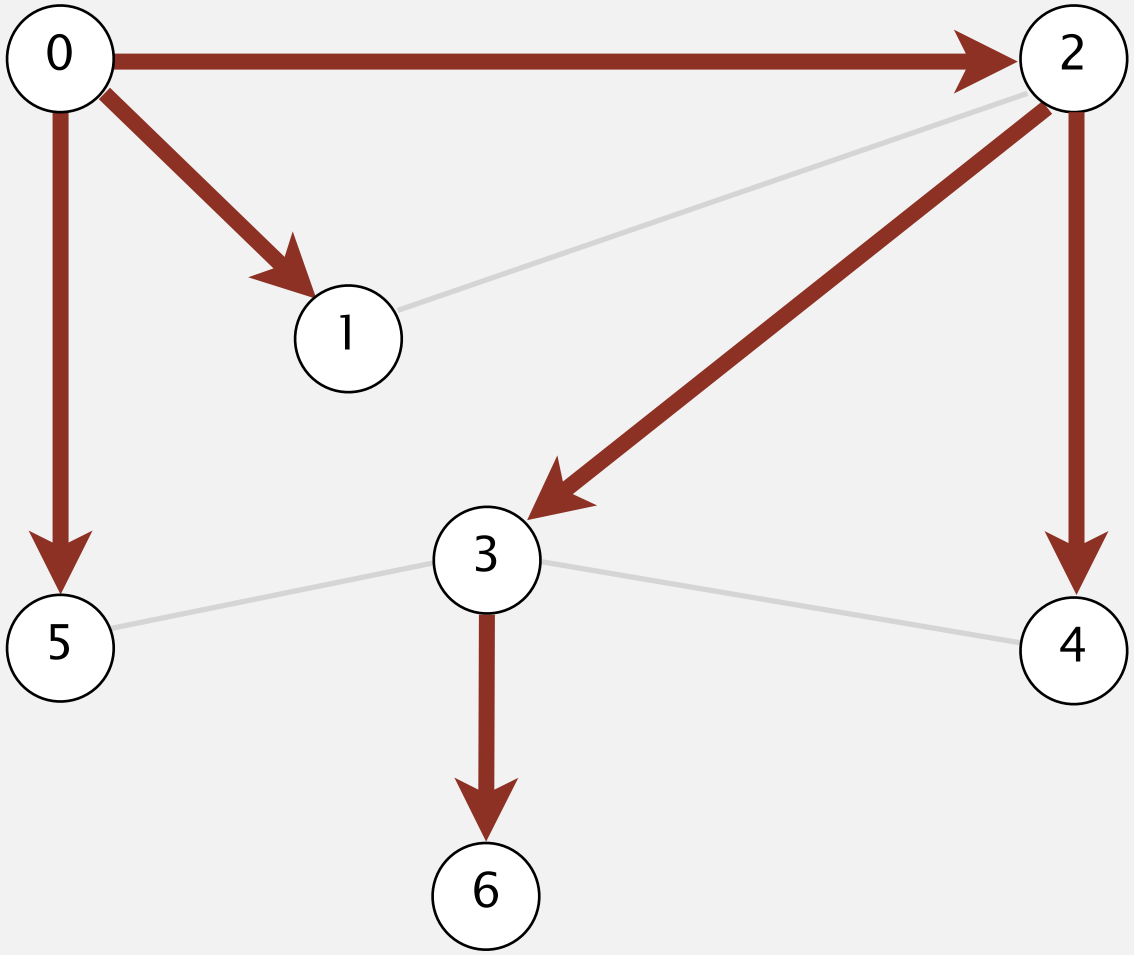
v	edgeTo[]	marked[]
0	–	T
1	0	T
2	0	T
3	2	T
4	2	T
5	0	T
6	3	T

all done

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.



v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1
6	3	3

done

Breadth-first search: Java implementation

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

   …

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

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

https://algs4.cs.princeton.edu/41undirected/BreadthFirstPaths.java.html

initialize FIFO queue of vertices to explore

found new vertex w via edge v–w

Breadth-first search: Java implementation

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

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

      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;
               distTo[w] = distTo[v] + 1;
            }
         }
      }
   }
}
```

initialize FIFO queue of vertices to explore

found new vertex w via edge v–w

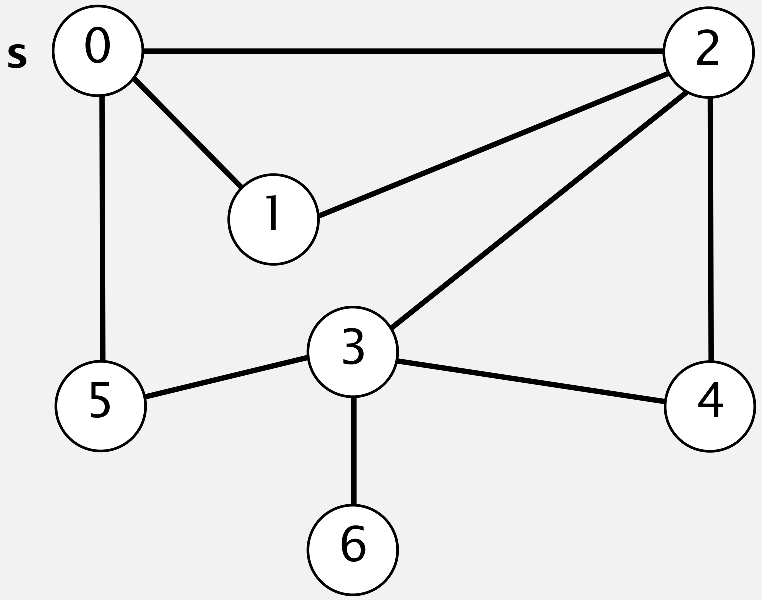
https://algs4.cs.princeton.edu/41undirected/BreadthFirstPaths.java.html

Breadth-first search properties

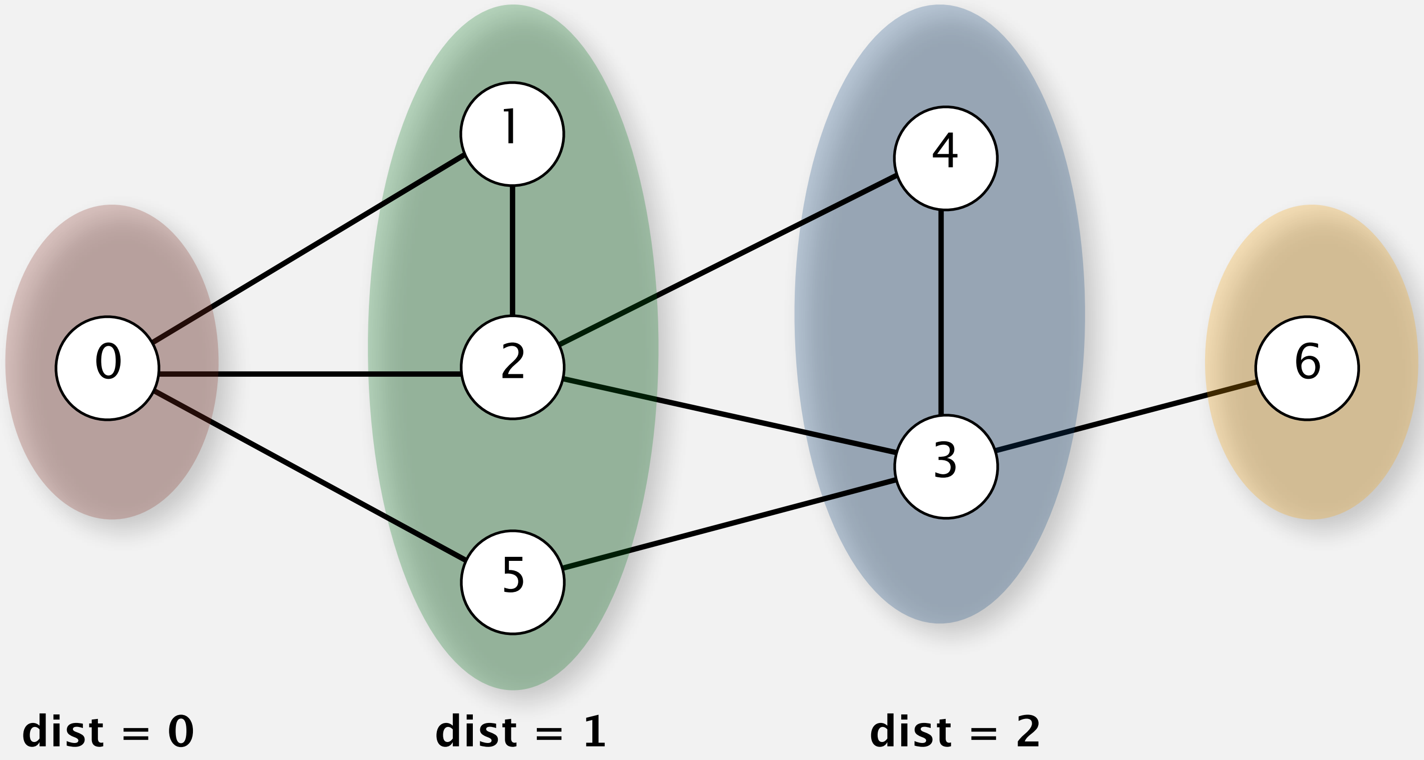
BFS examines vertices in order of increasing distance (# of edges) from s.

queue always consists of ≥ 0 vertices of distance k from s,
followed by ≥ 0 vertices of distance $k+1$

Proposition. In any connected graph G, BFS computes shortest paths from s to all other vertices in time proportional to $E + V$.



graph G



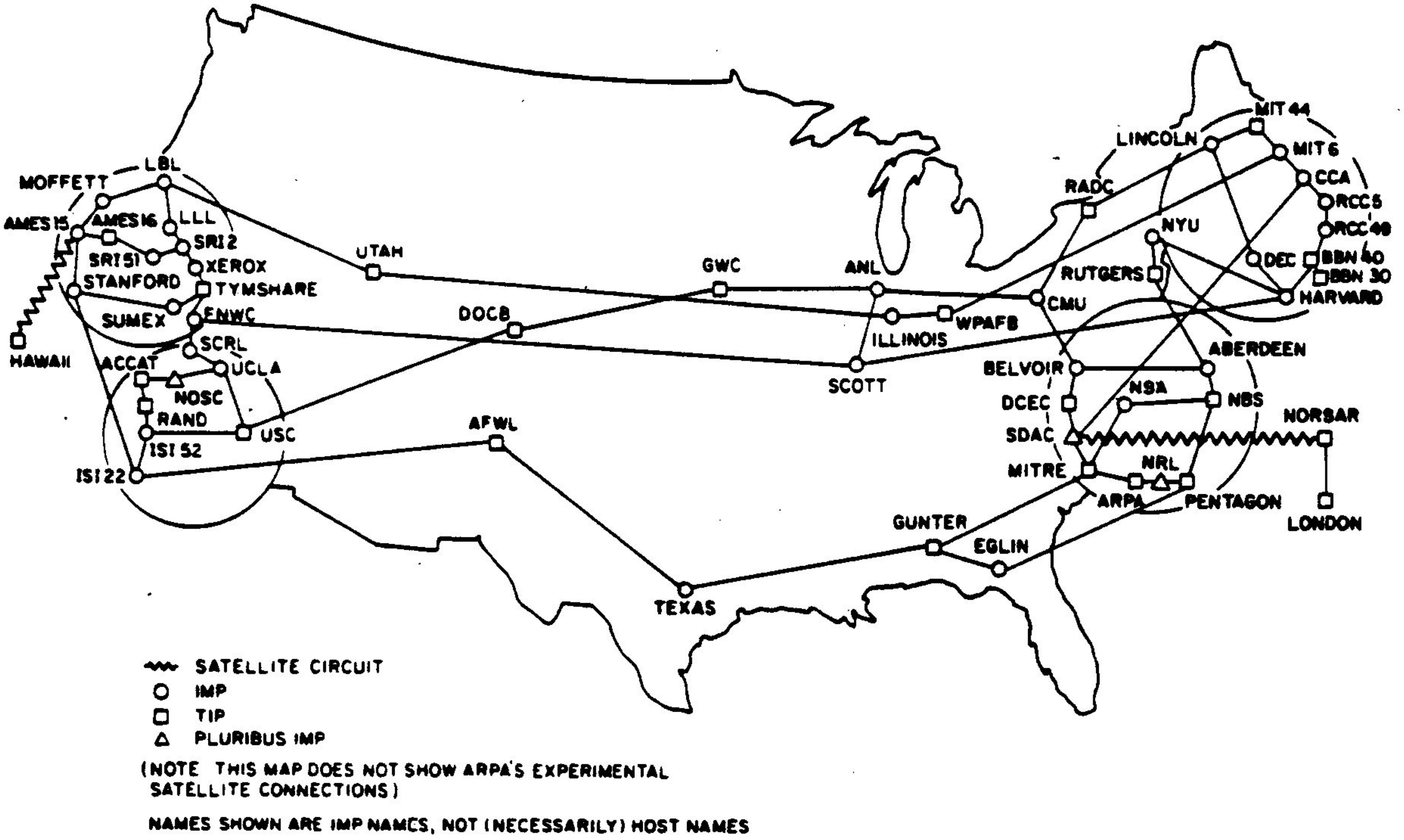
https://algs4.cs.princeton.edu

4.1 UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *applications of DFS and BFS*

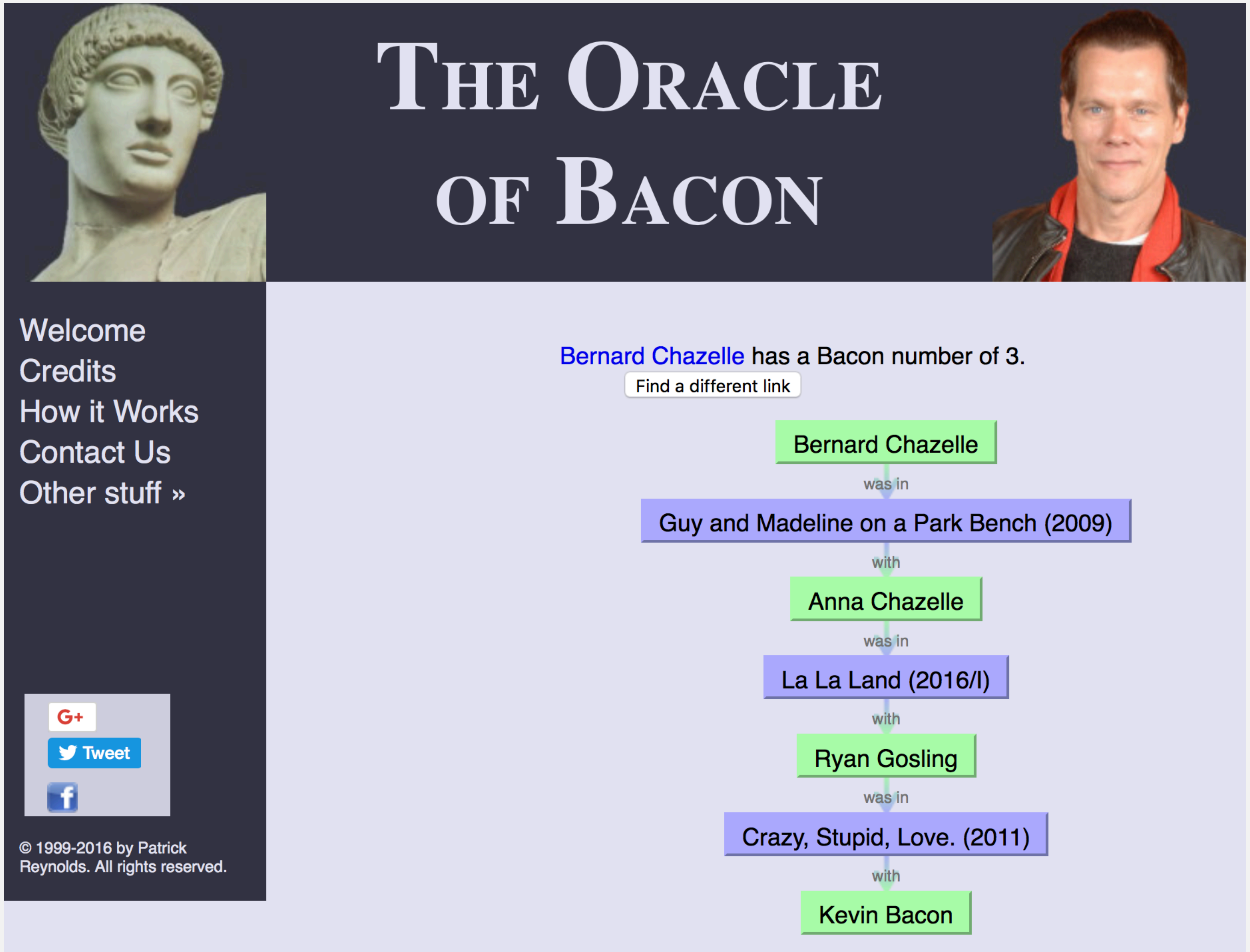
Breadth-first search application: routing

Fewest number of hops in a communication network.



ARPANET, July 1977

Breadth-first search application: Kevin Bacon numbers



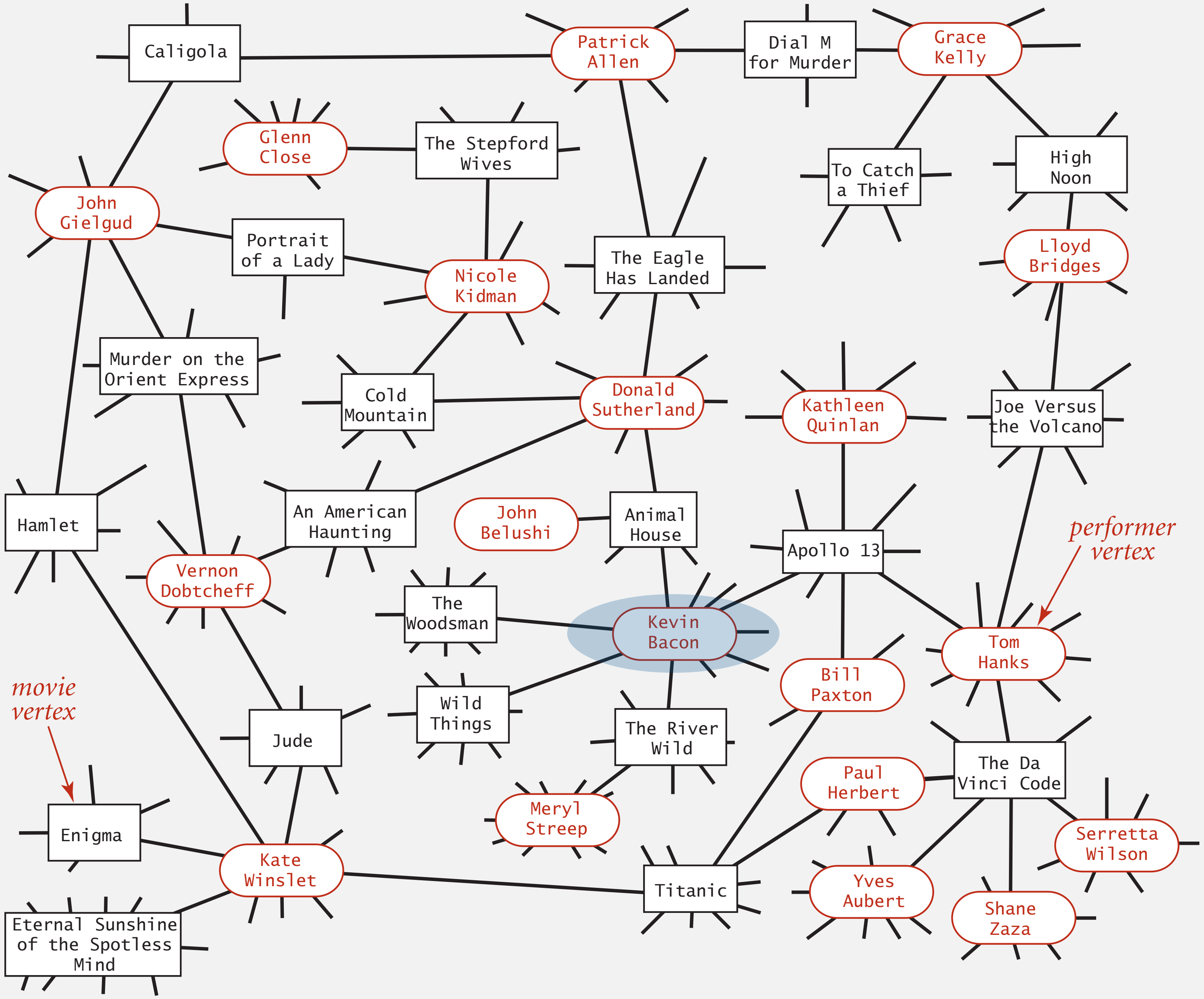
Bernard Chazelle has a Bacon number of 3.

Bernard Chazelle
was in
Guy and Madeline on a Park Bench (2009)
with
Anna Chazelle
was in
La La Land (2016/I)
with
Ryan Gosling
was in
Crazy, Stupid, Love. (2011)
with
Kevin Bacon

http://oracleofbacon.org

Kevin Bacon graph

- Include one vertex for each performer and one for each movie.
- Connect a movie to all performers that appear in that movie.
- Compute shortest path from s = Kevin Bacon.



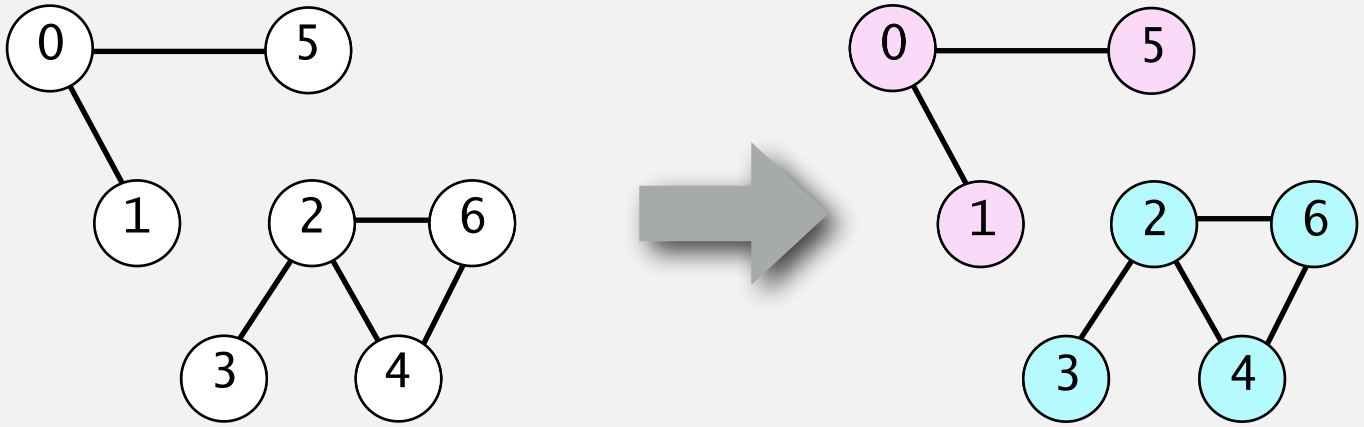
Exercise: applications of DFS and BFS

Recall: a connected component is a maximal set of connected vertices.

Given a graph, partition vertices into connected components using DFS or BFS.

i.e. create an `id[]` array such that `id[u] == id[v]` iff u & v are in same CC.

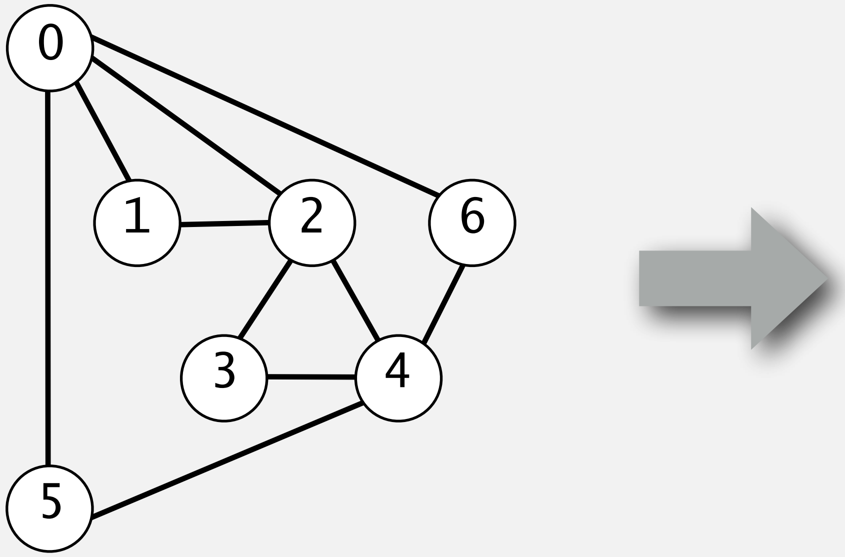
Same property as quick-find



Euler cycle: given a graph, find a general cycle that traverses each edge exactly once, or determine that none exists.

May traverse a node more than once

Challenge



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

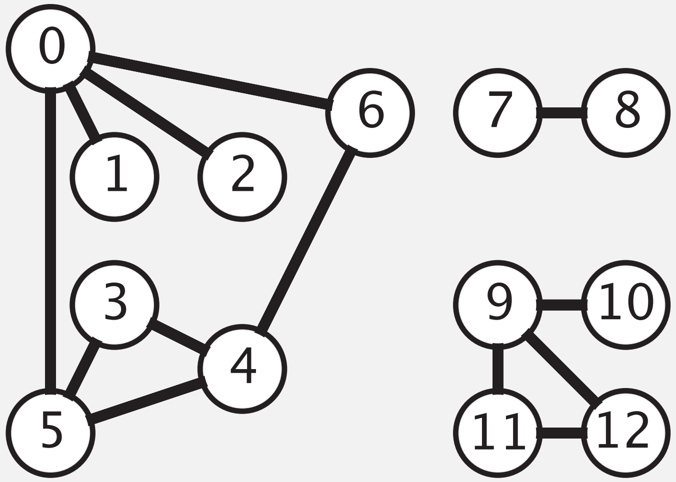
Connected components

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.



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)
   public boolean connected(int v, int w)
   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];  }

   public boolean connected(int v, int w)
   { return id[v] == id[w];  }


   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

v and w connected iff same id

all vertices discovered in
same call of dfs have same id

Graph traversal summary

BFS and DFS enables efficient solution of many (but not all) graph problems.

graph problem	BFS	DFS	time
s-t path	✔	✔	$E + V$
shortest s-t path	✔		$E + V$
cycle	✔	✔	V
Euler cycle		✔	$E + V$
Hamilton cycle			$2^{1.657\, V}$
bipartiteness (odd cycle)	✔	✔	$E + V$
connected components	✔	✔	$E + V$
biconnected components		✔	$E + V$
planarity		✔	$E + V$
graph isomorphism			$2^{c \ln^3 V}$