Undirected Graphs

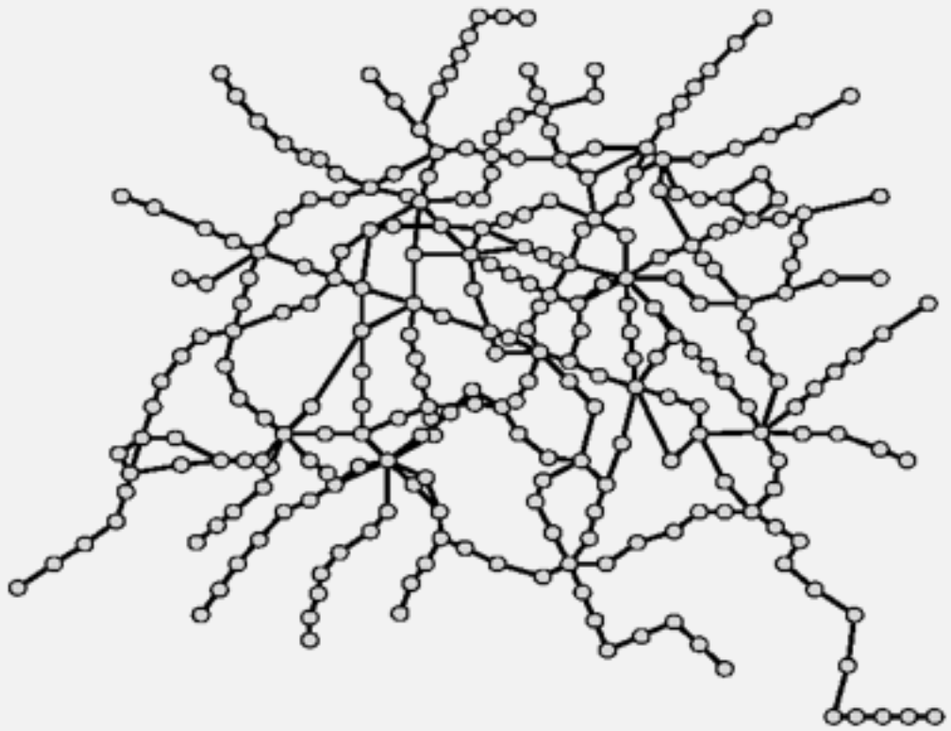
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

Undirected graphs

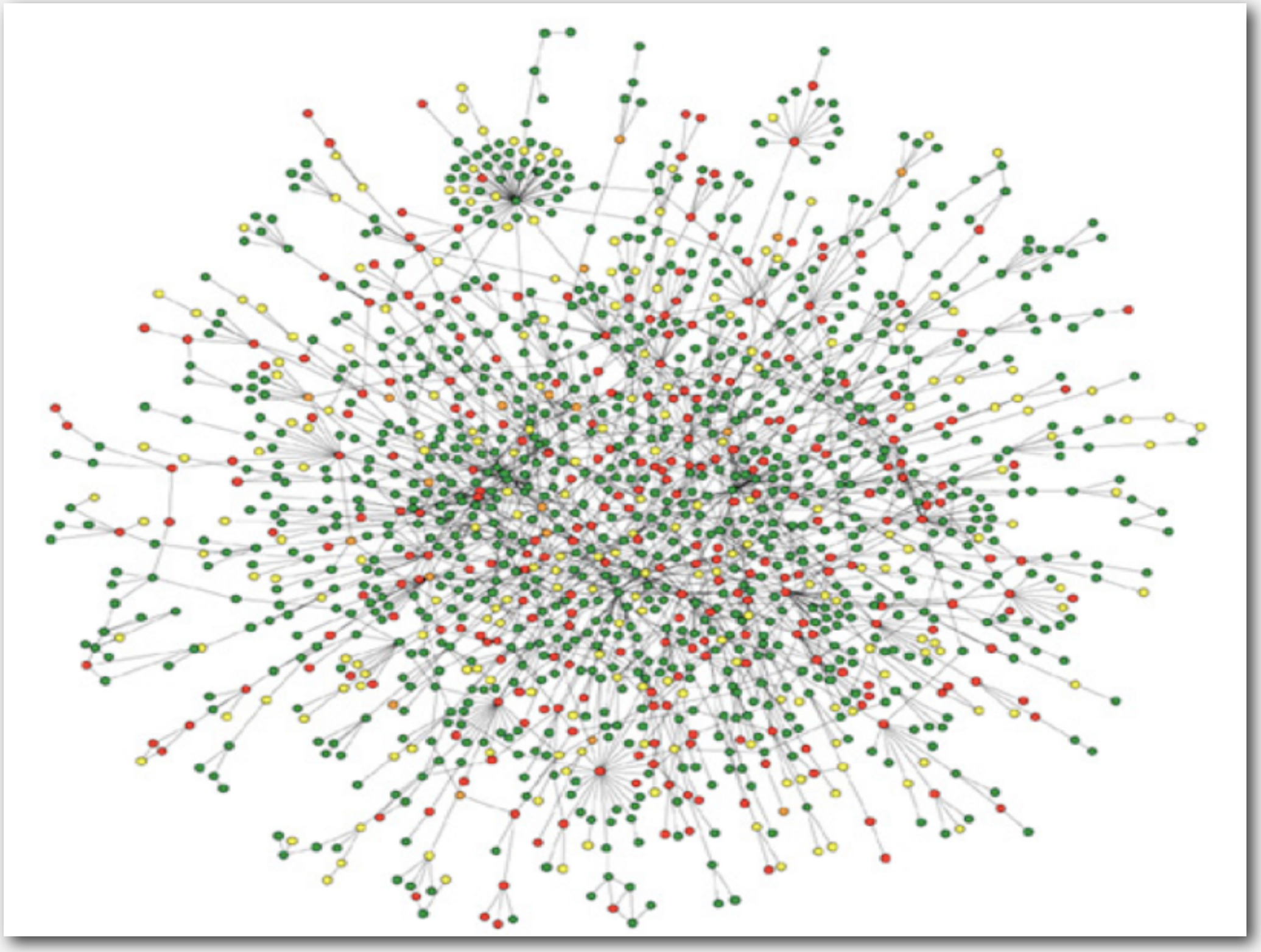
Graph. Set of vertices connected pairwise by edges.

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.

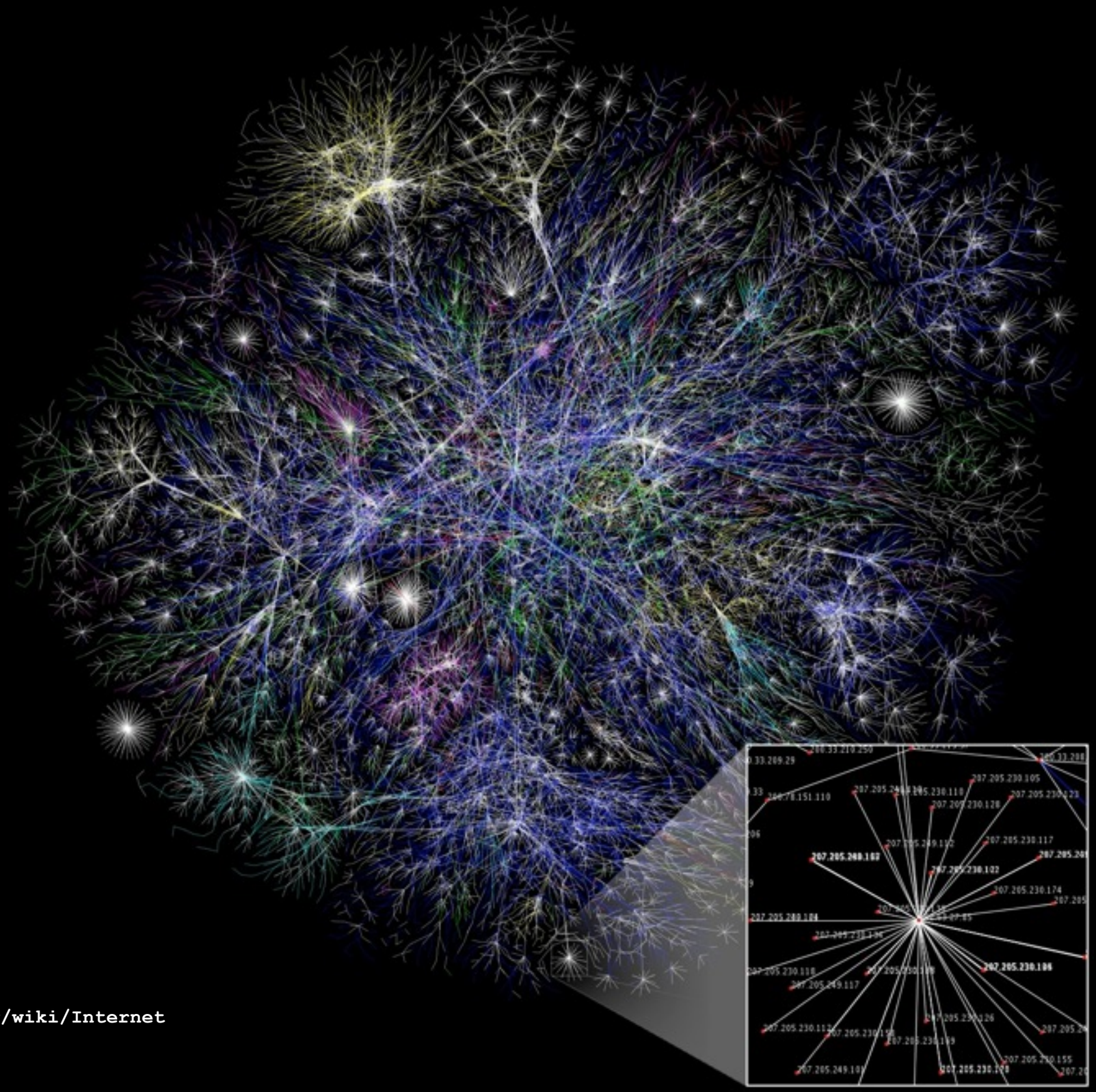


Protein-protein interaction network



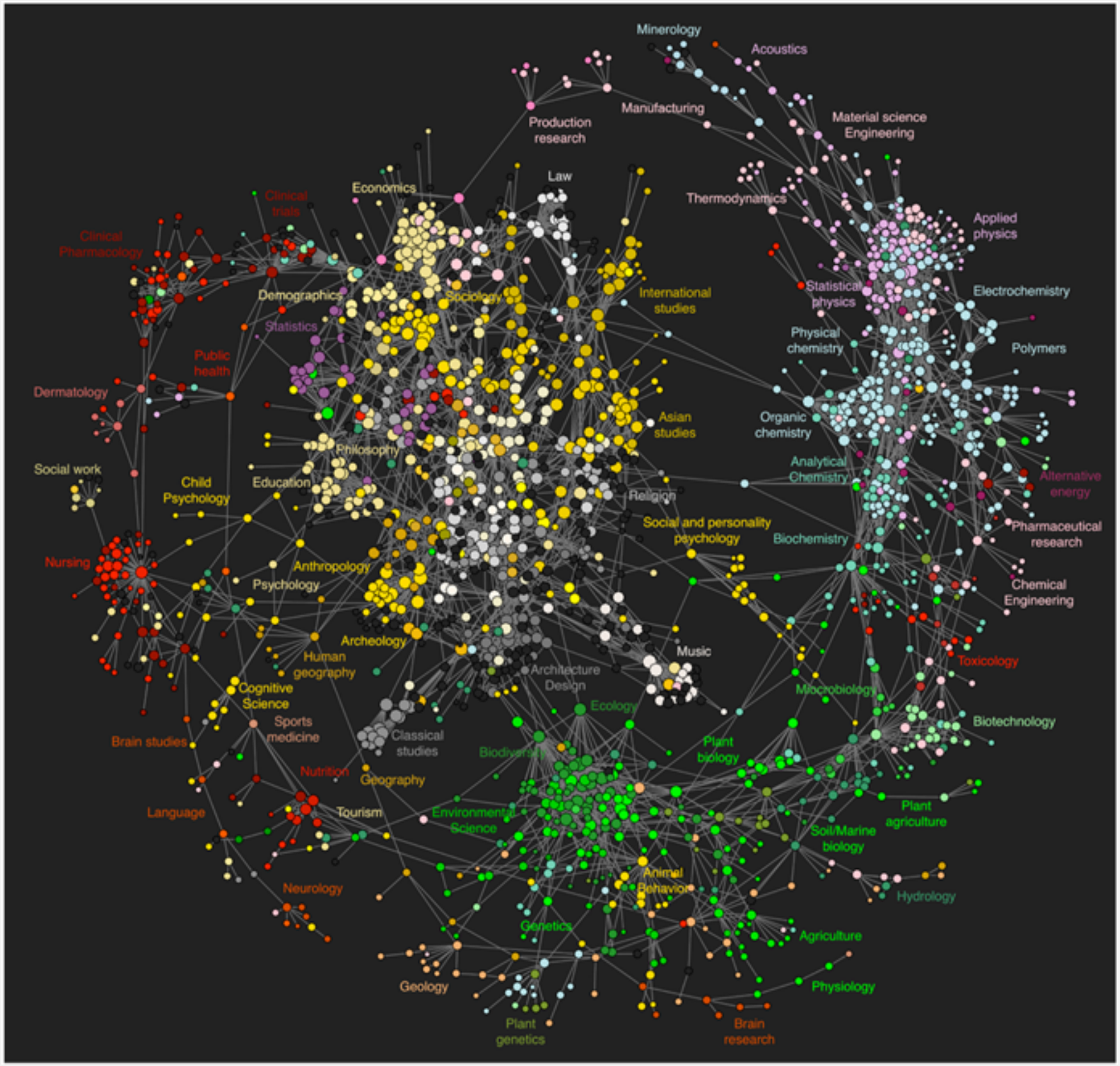
Reference: Jeong et al, Nature Review | Genetics

The Internet as mapped by the Opte Project



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

Map of science clickstreams



http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

Graph applications

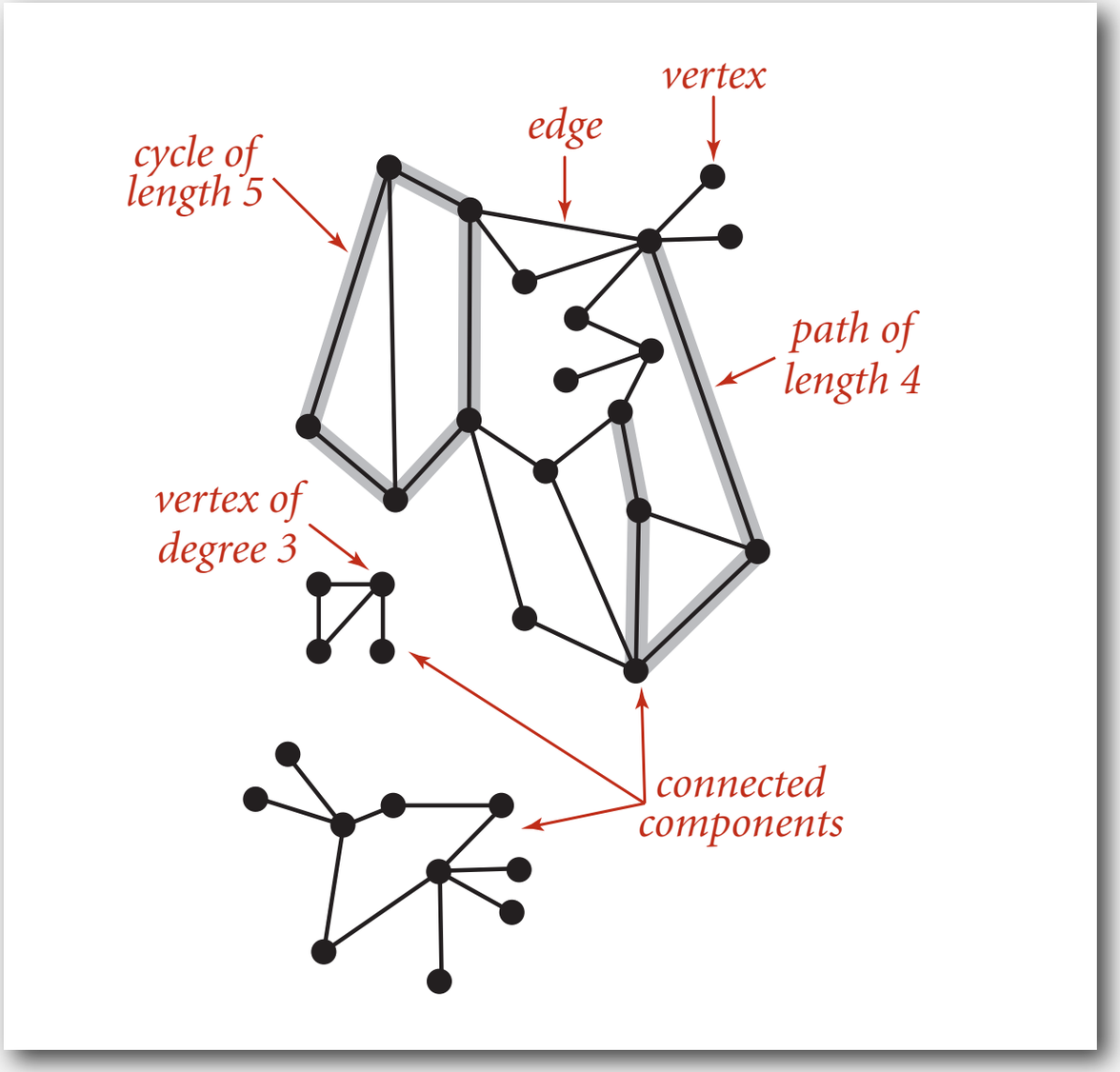
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position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

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.



Some graph-processing problems

Path. Is there a path between s and t ?

Shortest path. What is the shortest path between s and t ?

Cycle. 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?

Connectivity. Is there a way to connect all of the vertices?

MST. What is the best way to connect all of the vertices?

Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges?

Graph isomorphism. Do two adjacency lists represent the same graph?

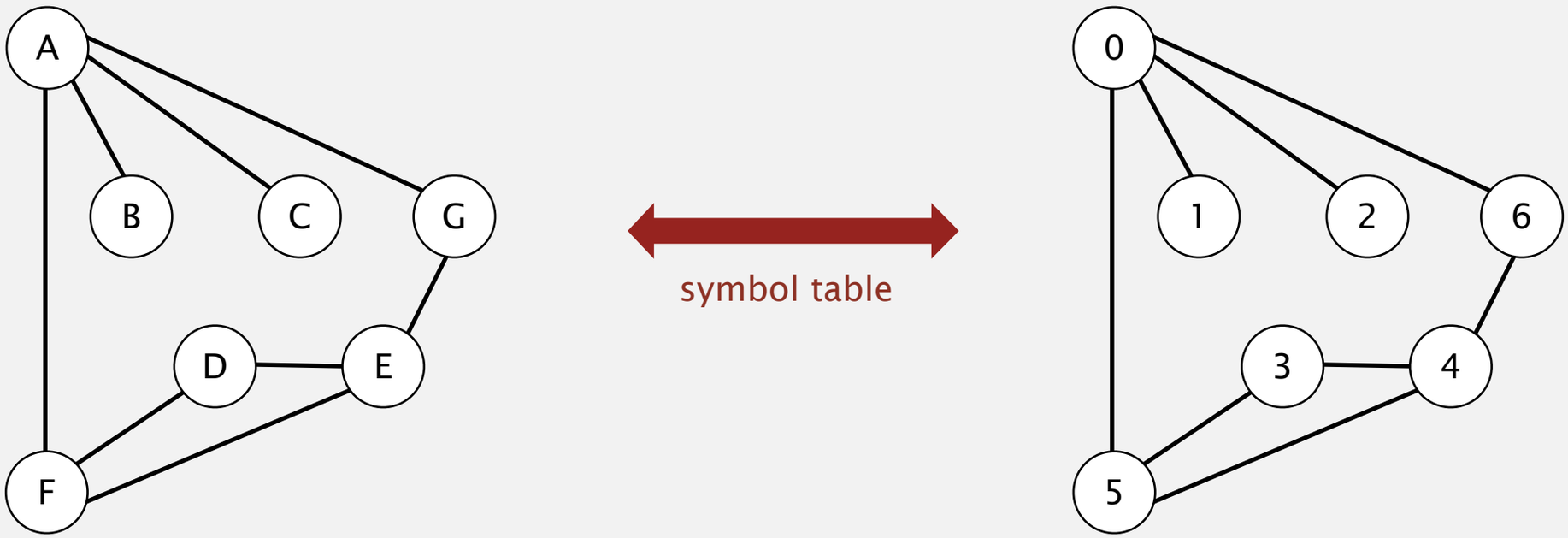
Challenge. Which of these problems are easy? difficult? intractable?

- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

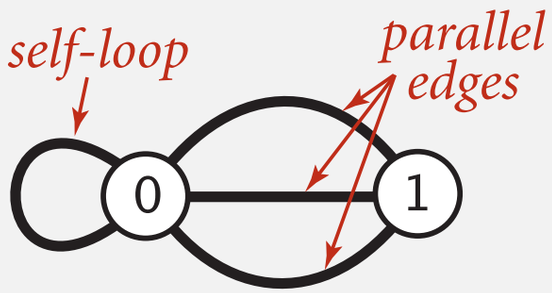
Graph representation

Vertex representation.

- Use integers between 0 and $V - 1$.
- Applications: convert between names and integers with symbol table.



Anomalies.



Graph API

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

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

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

read graph from input stream

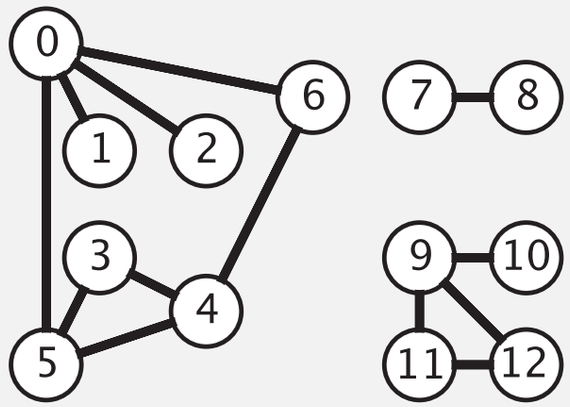
print out each edge (twice)

Graph API: sample client

Graph input format.

tinyG.txt

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




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

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

read graph from input stream

print out each edge (twice)

Typical graph-processing code

compute the degree of v

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

compute maximum degree

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

compute average degree

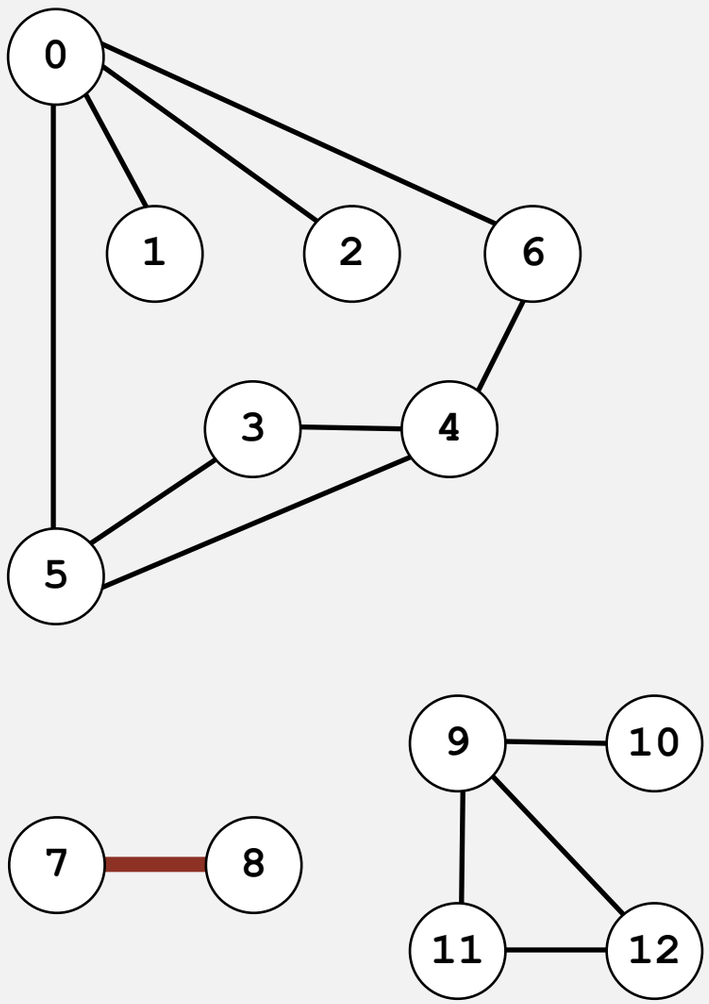
```
public static double averageDegree(Graph G)
{  return 2.0 * G.E() / G.V();  }
```

count self-loops

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

Set-of-edges graph representation

Maintain a list of the edges (linked list or array).

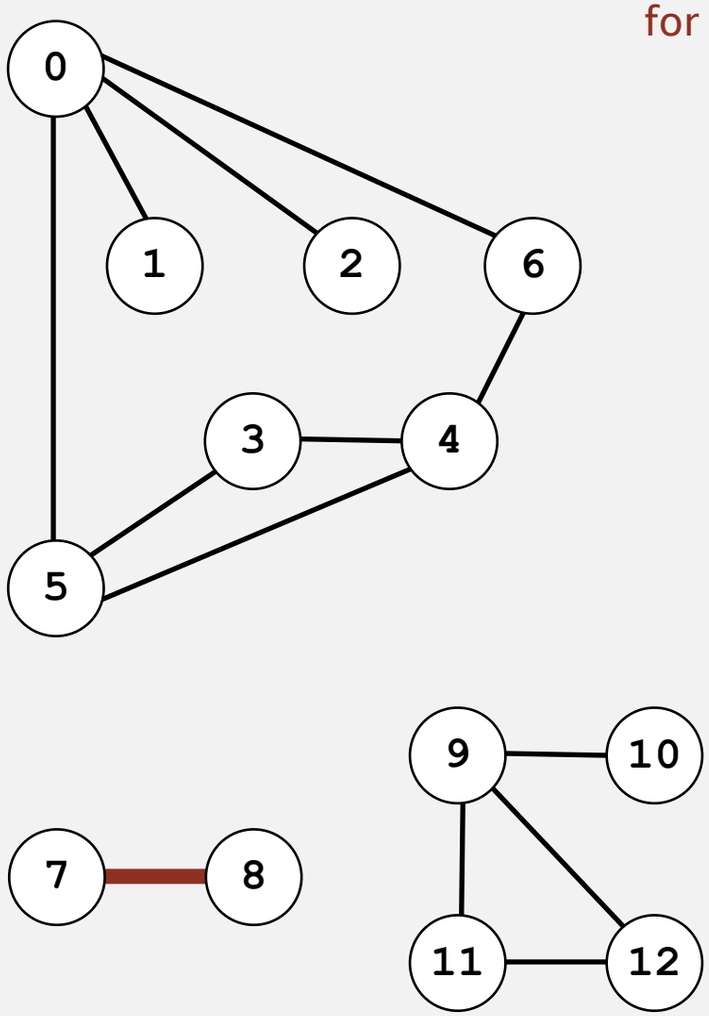



```
 0  1
 0  2
 0  5
 0  6
 3  4
 3  5
 4  5
 4  6
 7  8
 9 10
 9 11
 9 12
11 12
```

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;

for each edge v–w in graph: `adj[v][w] = adj[w][v] = true`.

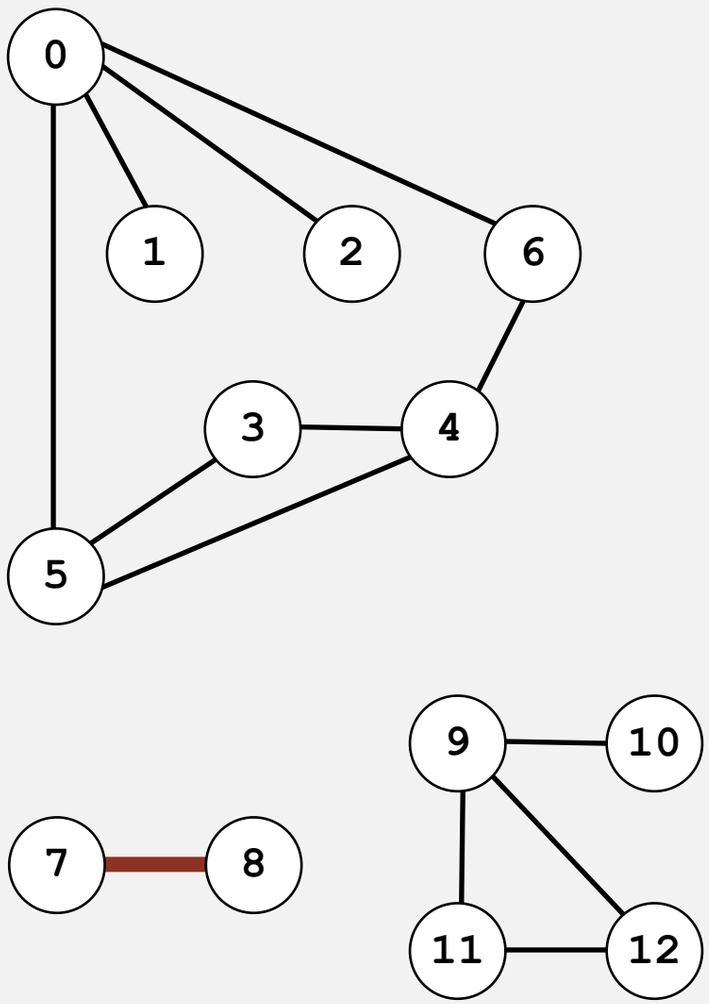


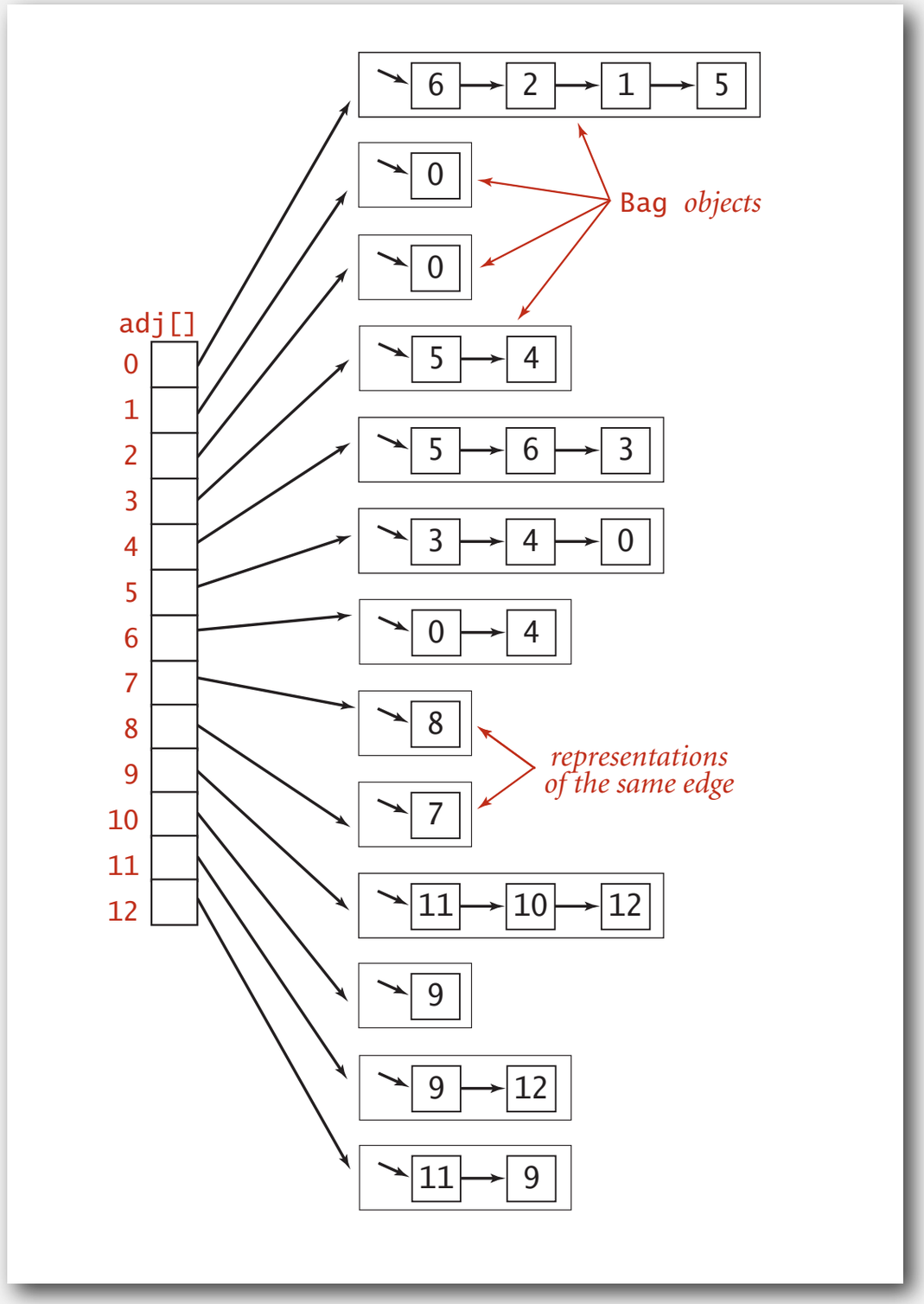
two entries for each edge

	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

Adjacency-list graph representation

Maintain vertex-indexed array of lists.





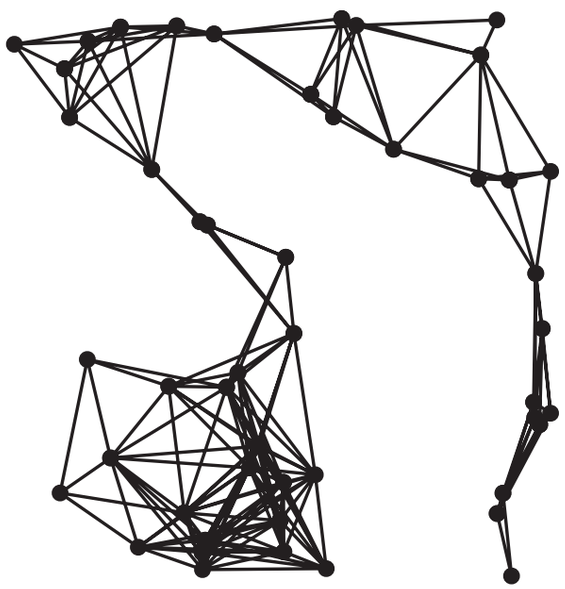
Graph representations

In practice. Use adjacency-lists representation.

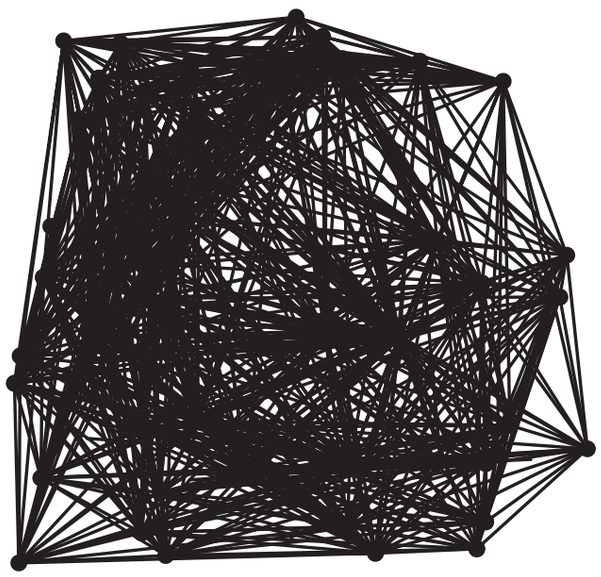
- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be "sparse."

huge number of vertices,
small average vertex degree

sparse (E = 200)



dense (E = 1000)



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."

huge number of vertices,
small average vertex degree

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	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

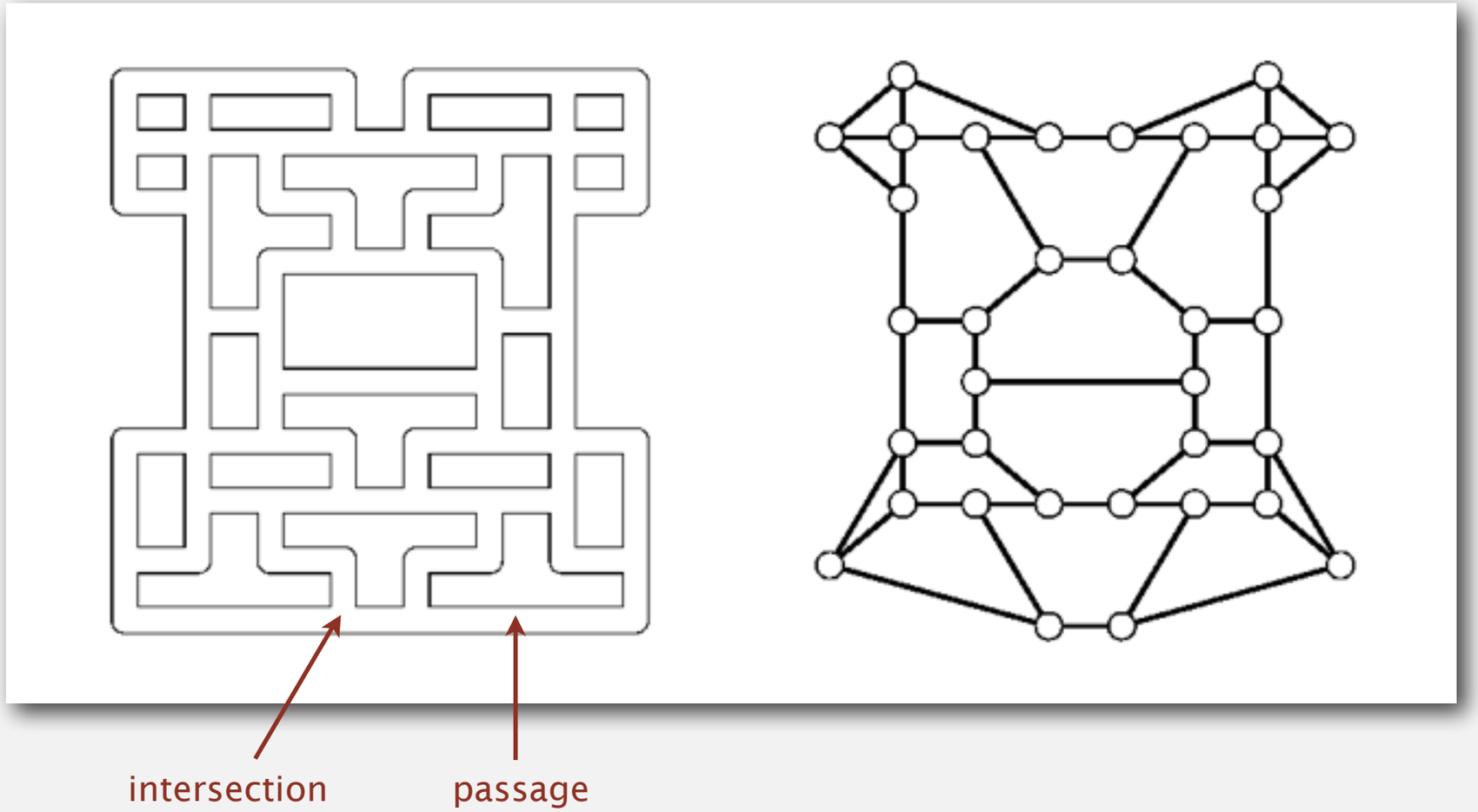
* disallows parallel edges

- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

Maze exploration

Maze graphs.

- Vertex = intersection.
- Edge = passage.



Goal. Explore every intersection in the maze.

Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked vertices w adjacent to v.

Typical applications. [ahead]

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

Depth-first search (warmup)

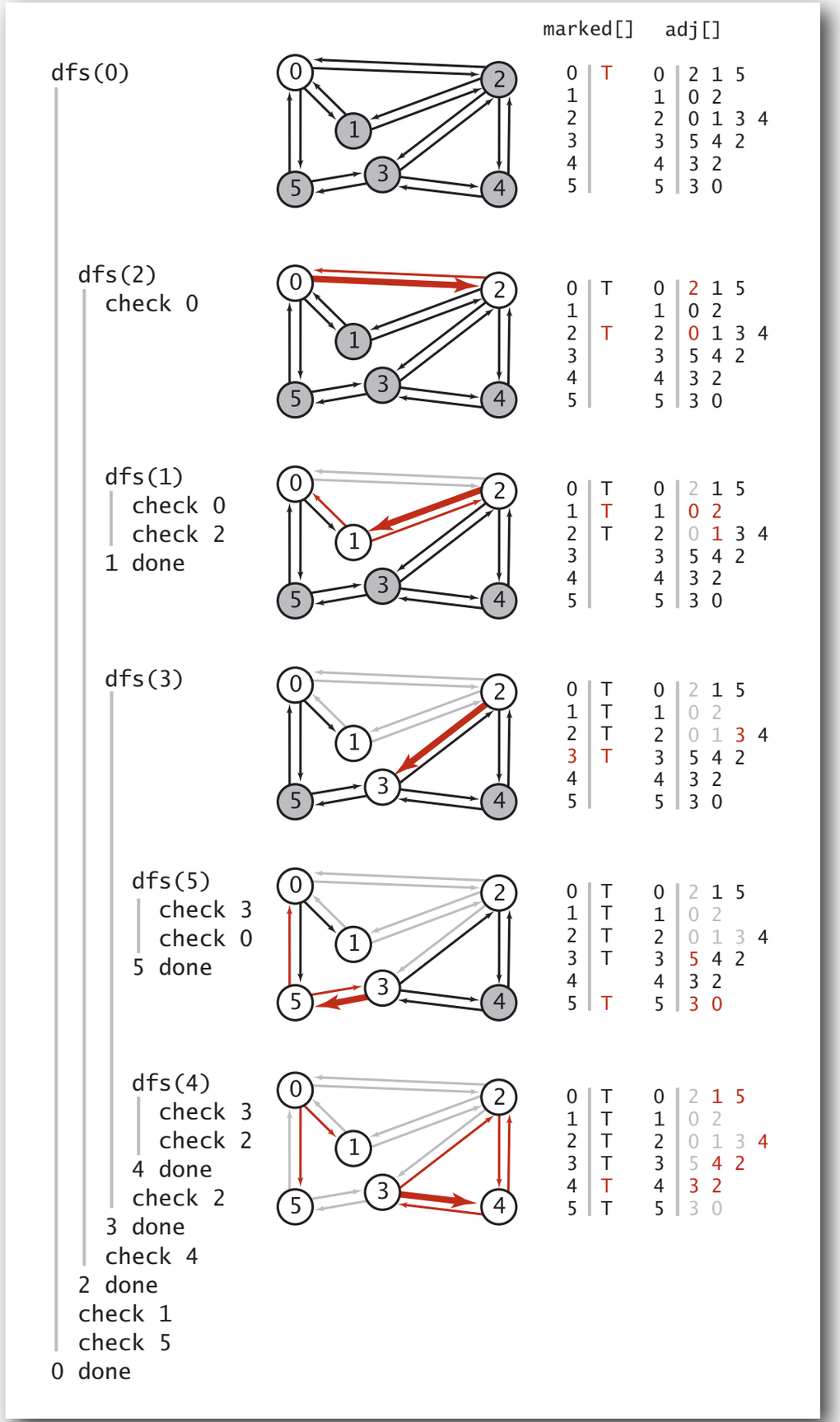
Goal. Find all vertices connected to s.

Algorithm.

- Use recursion stack
- Mark each visited vertex.
- Return (retrace steps) when no unvisited options.

Data structure.

- `boolean[] marked` to mark visited vertices.



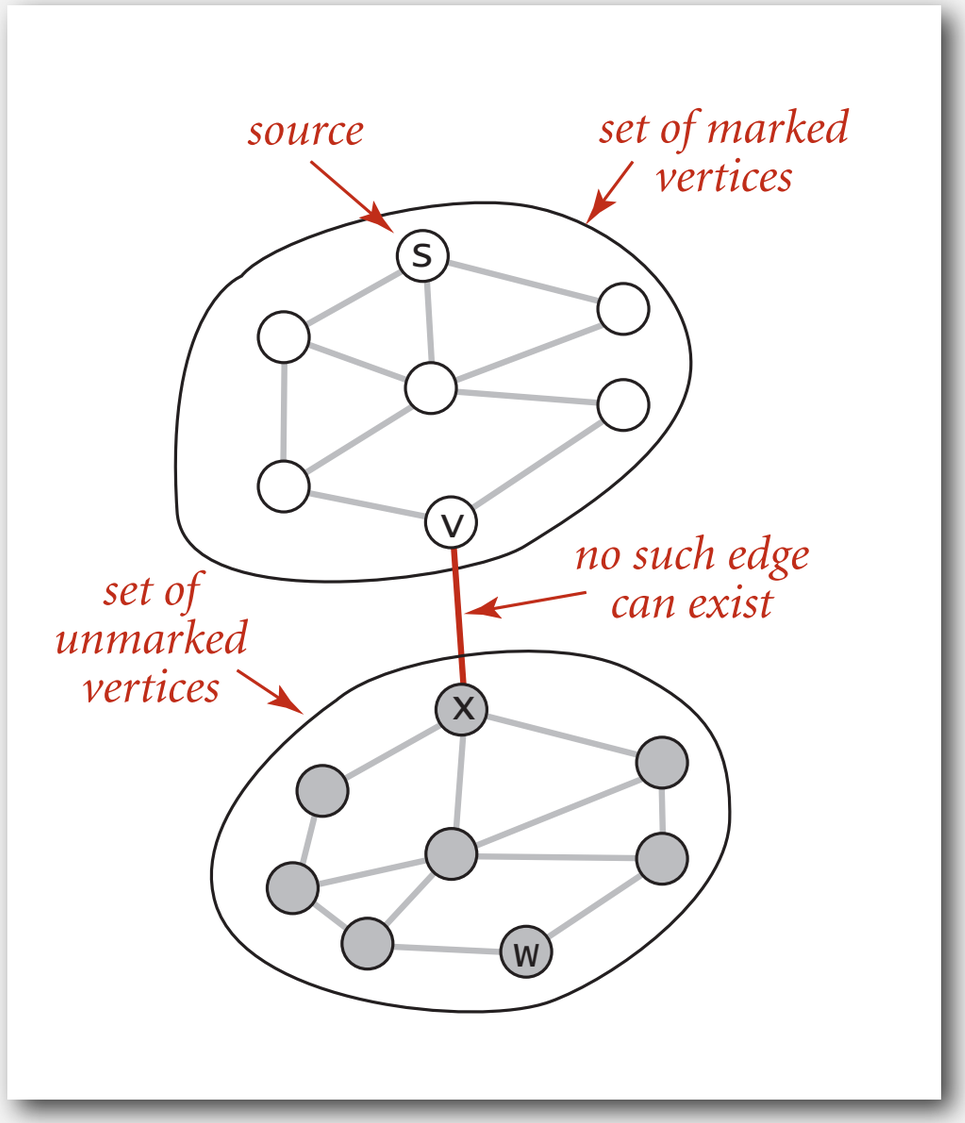
Depth-first search properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees.

Pf.

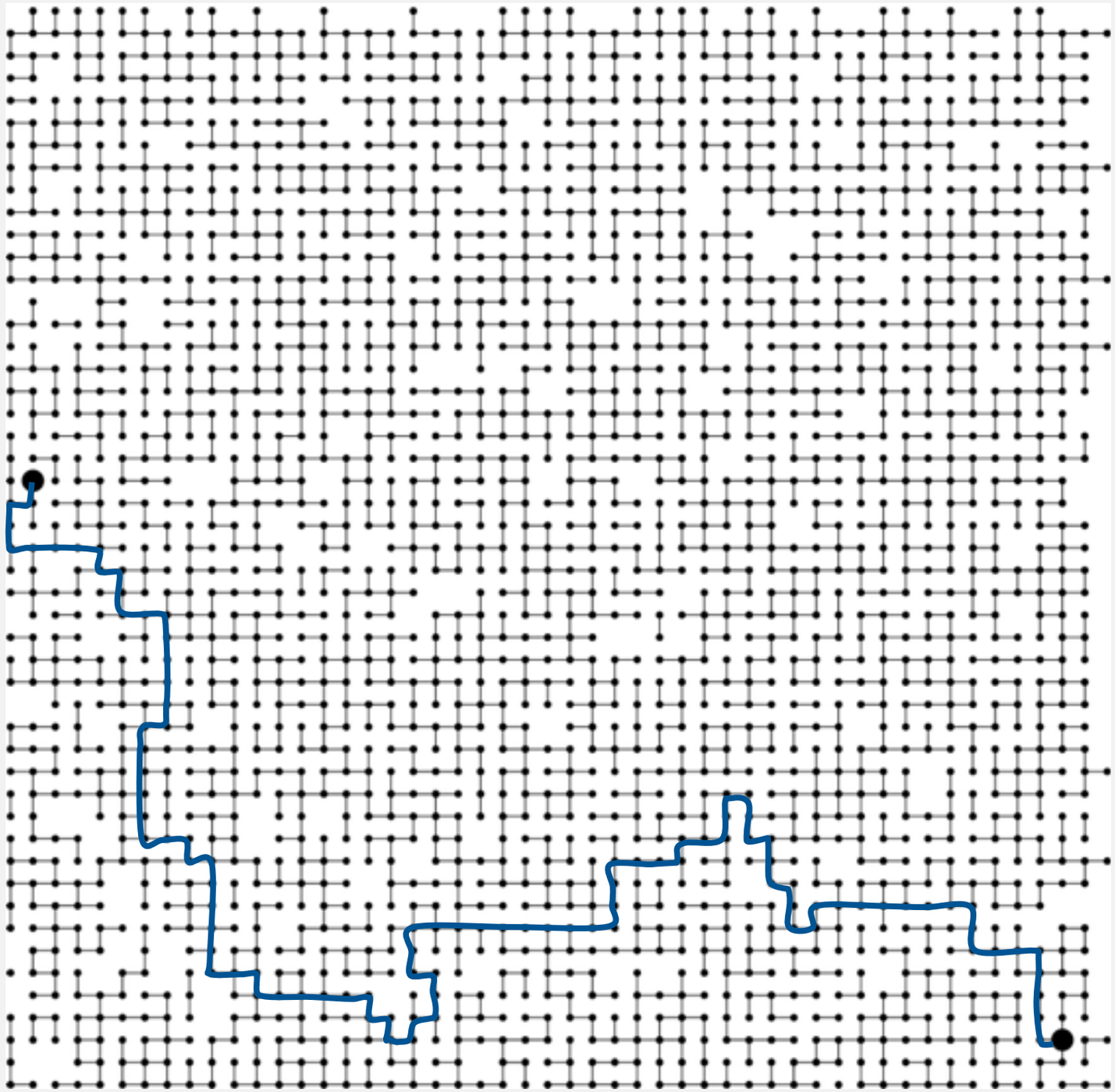
- Correctness:
  - if w marked, then w connected to s (why?)
  - if w connected to s, then w marked
    (if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one)

- Running time: each vertex connected to s is visited once.



Paths in graphs

Goal. Does there exist a path from s to t ?



Paths in graphs: union-find vs. DFS

Goal. Does there exist a path from s to t ?

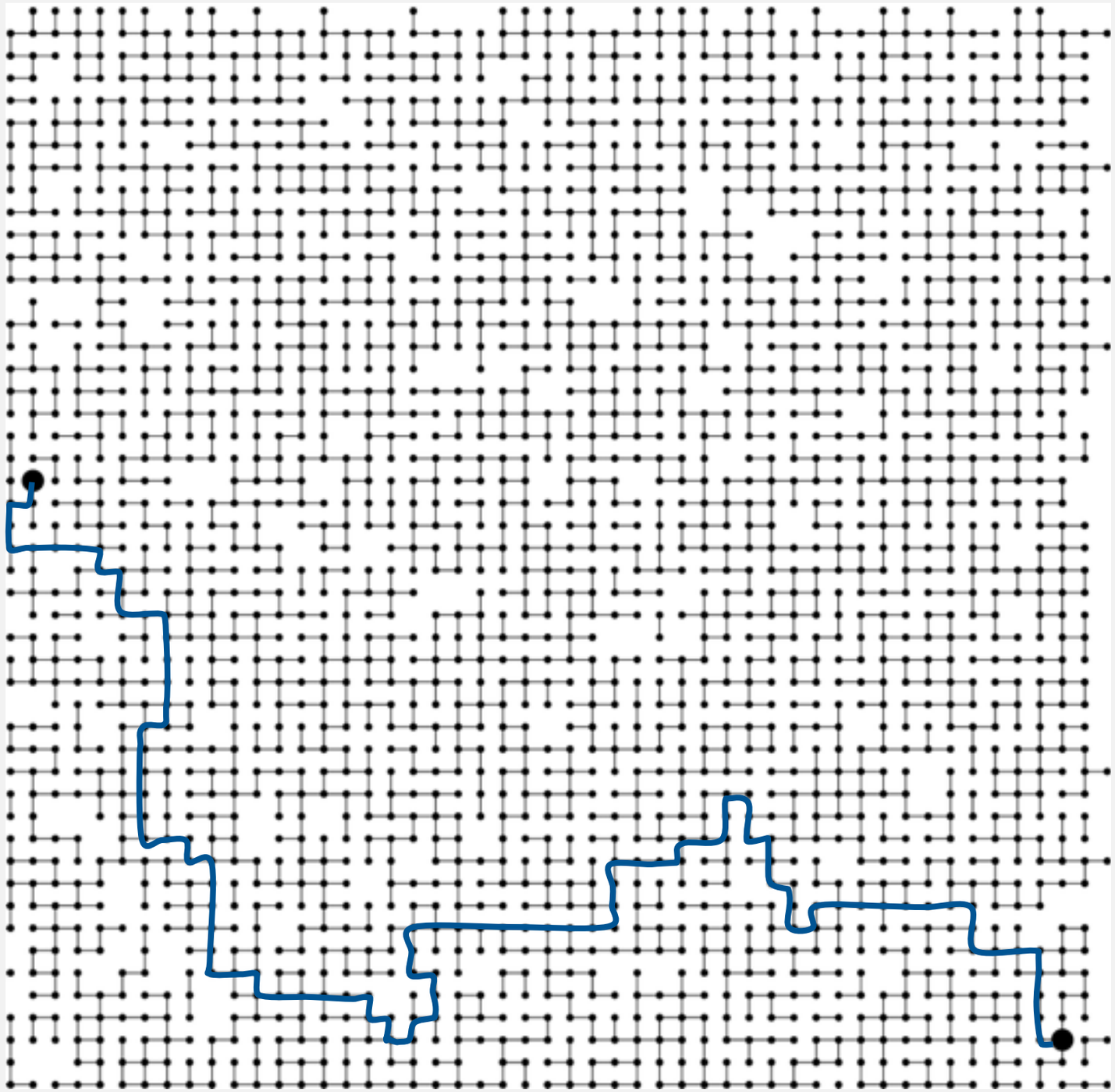
method	preprocessing time	query time	space
union-find	V + E log* V	log* V	V
DFS	E + V	1	E + V

Union-find. Can intermix connected queries and edge insertions.

Depth-first search. Constant time per query.

Pathfinding in graphs

Goal. Does there exist a path from s to t? If yes, find any such path.



Pathfinding in graphs

Goal. Does there exist a path from s to t? If yes, find any such path.

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

Union-find. Not much help.

Depth-first search. After linear-time preprocessing, can recover path itself in time proportional to its length.

easy modification

Q: When is union-find better than DFS?

Depth-first search (pathfinding)

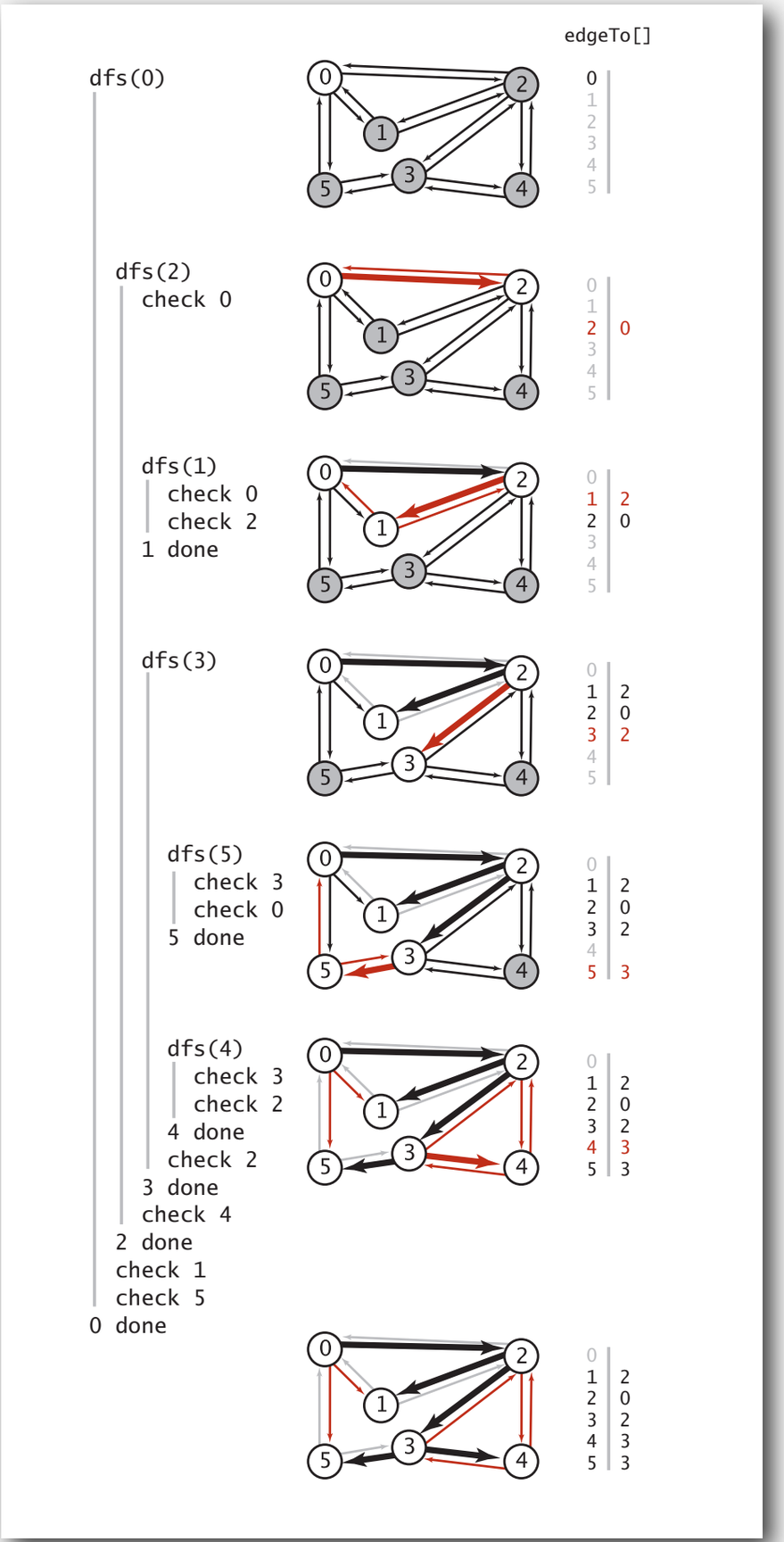
Goal. Find paths to all vertices connected to a given source s.

Algorithm.

- Use recursion
- Mark each visited vertex:
  - by keeping track of edge taken to visit it.
- Return (retrace steps) when no unvisited options.

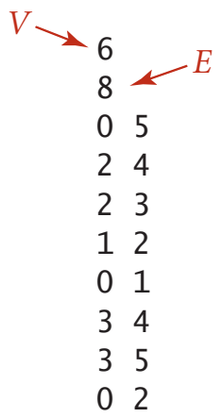
Data structures.

- `boolean[] marked` to mark visited vertices.
- `int[] edgeTo` to keep tree of paths.
- `(edgeTo[w] == v)` means that edge `v-w` was taken to visit `w` the first time

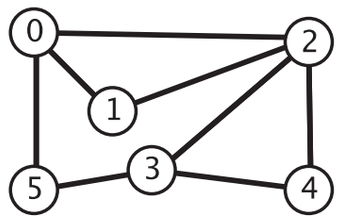


Depth-first search (pathfinding trace)

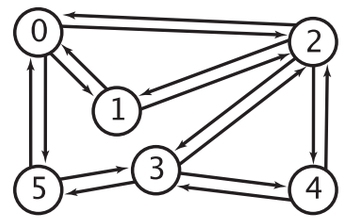
tinyCG.txt



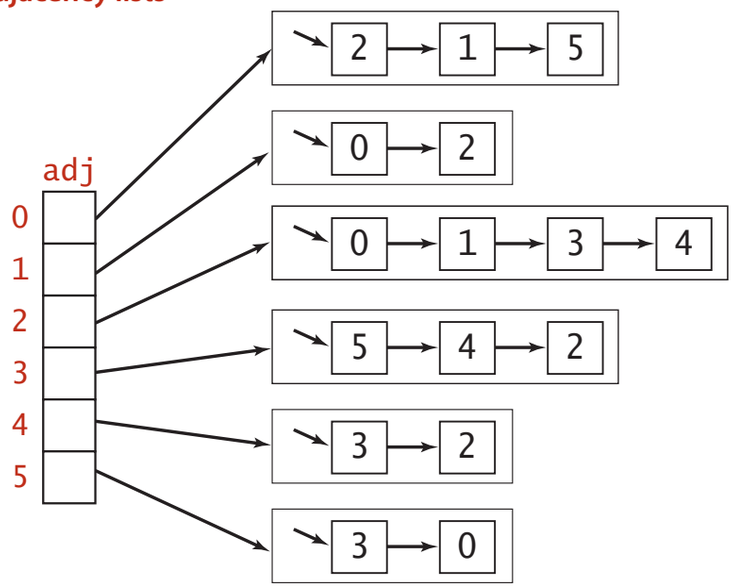
standard drawing

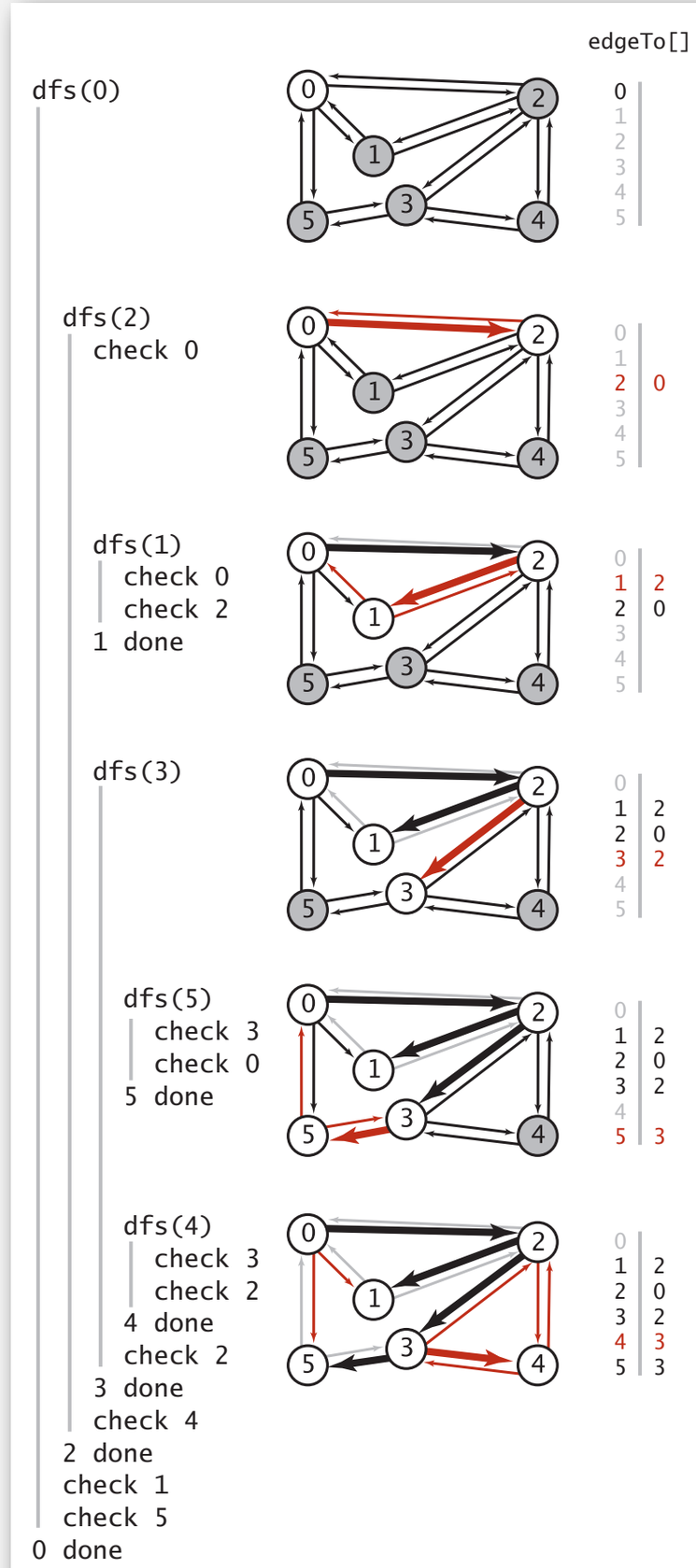


drawing with both edges



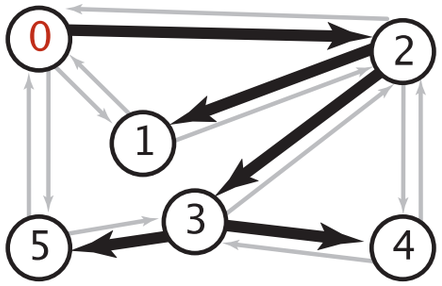
adjacency lists





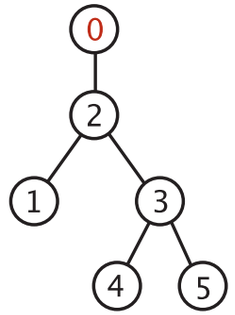
Depth-first search (pathfinding iterator)

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



edgeTo[]

0	
1	2
2	0
3	2
4	3
5	3



x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

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

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

Depth-first search summary

Enables direct solution of simple graph problems.

- ✓ Does there exists a path between s and t ?
- ✓ Find path between s and t.
- Connected components (to follow).
- Euler tour
- Cycle detection
- Bipartiteness checking

Basis for solving more difficult graph problems.

- Biconnected components (beyond scope).
- Planarity testing (beyond scope).

- graph API
- depth-first search
- **breadth-first search**
- connected components
- challenges

Breadth-first search

Depth-first search. Put unvisited vertices on a stack.

Breadth-first search. Put unvisited vertices on a queue.

Shortest path. Find path from s to t that uses fewest number of edges.

BFS (from source vertex s)

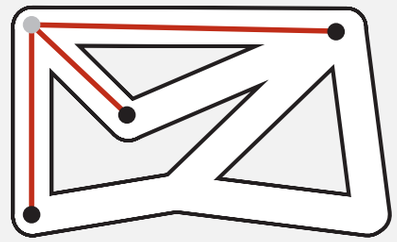
Put s onto a FIFO queue, and mark s as visited.

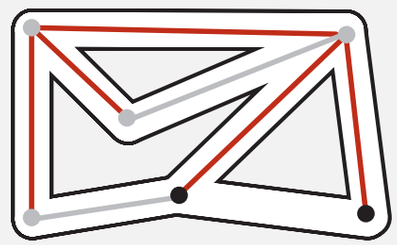
Repeat until the queue is empty:

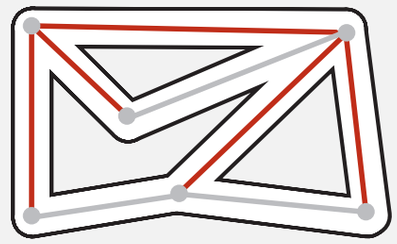
- remove the least recently added vertex v

- add each of v's unvisited neighbors to the queue,

and mark them as visited.



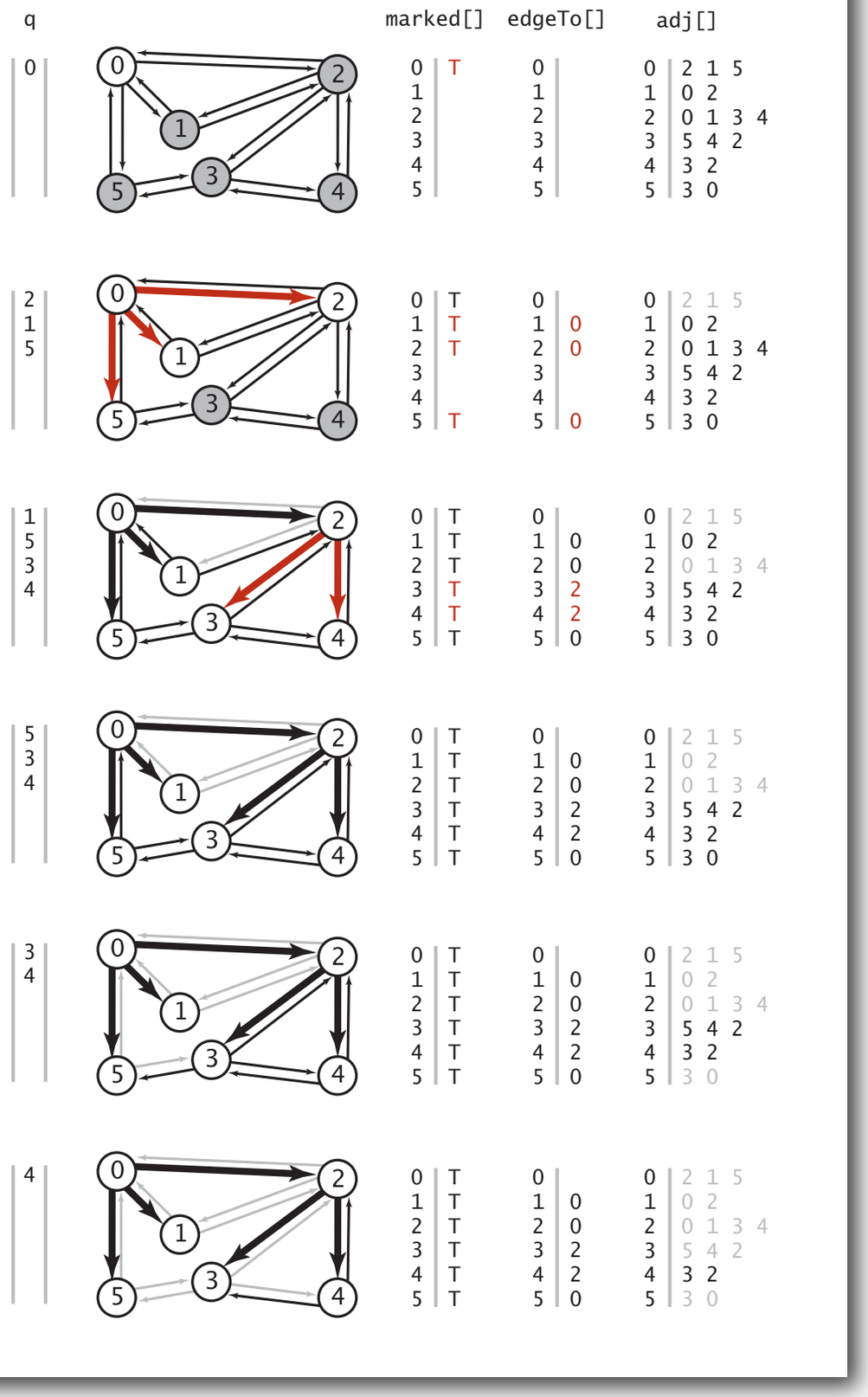




Intuition. BFS examines vertices in increasing distance from s.

Breadth-first search (pathfinding)

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

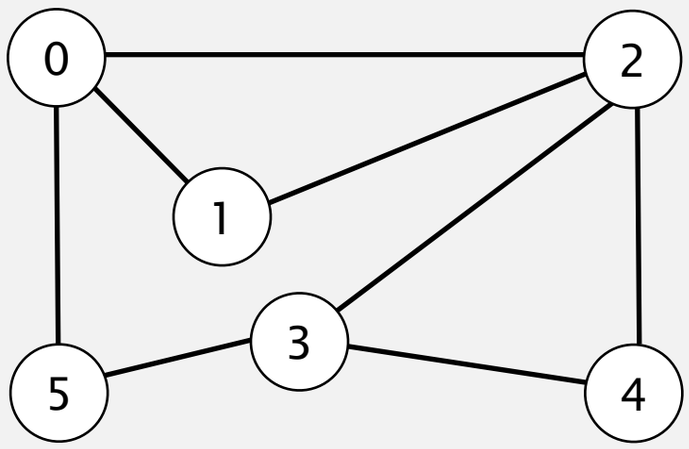


Breadth-first search properties

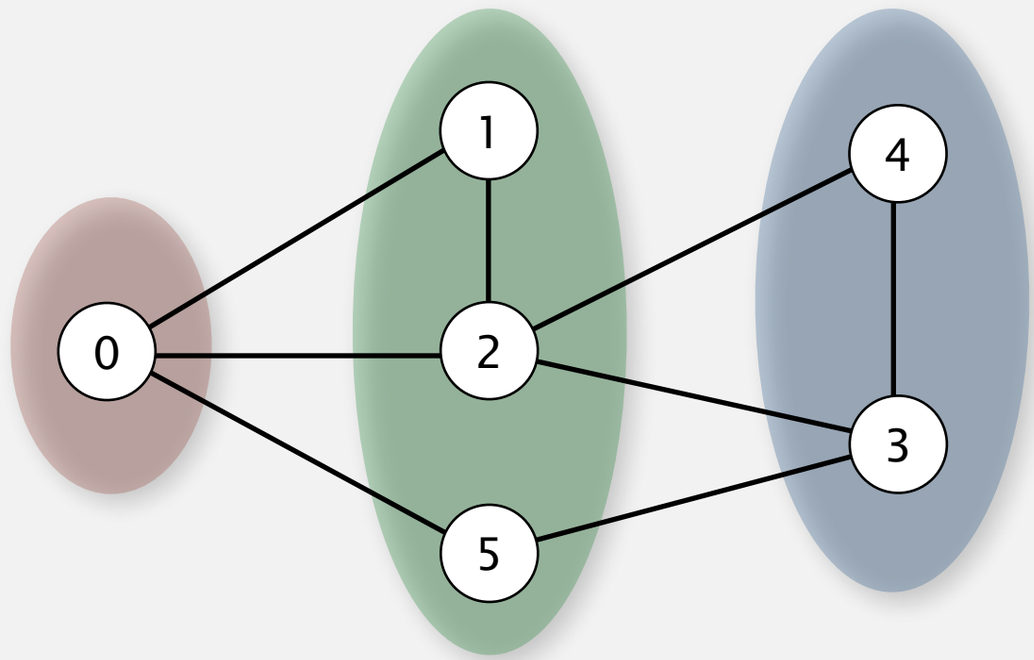
Proposition. BFS computes shortest path (number of edges) from s in a connected graph in time proportional to $E + V$.

Pf.

- Correctness: queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance $k + 1$.
- Running time: each vertex connected to s is visited once.



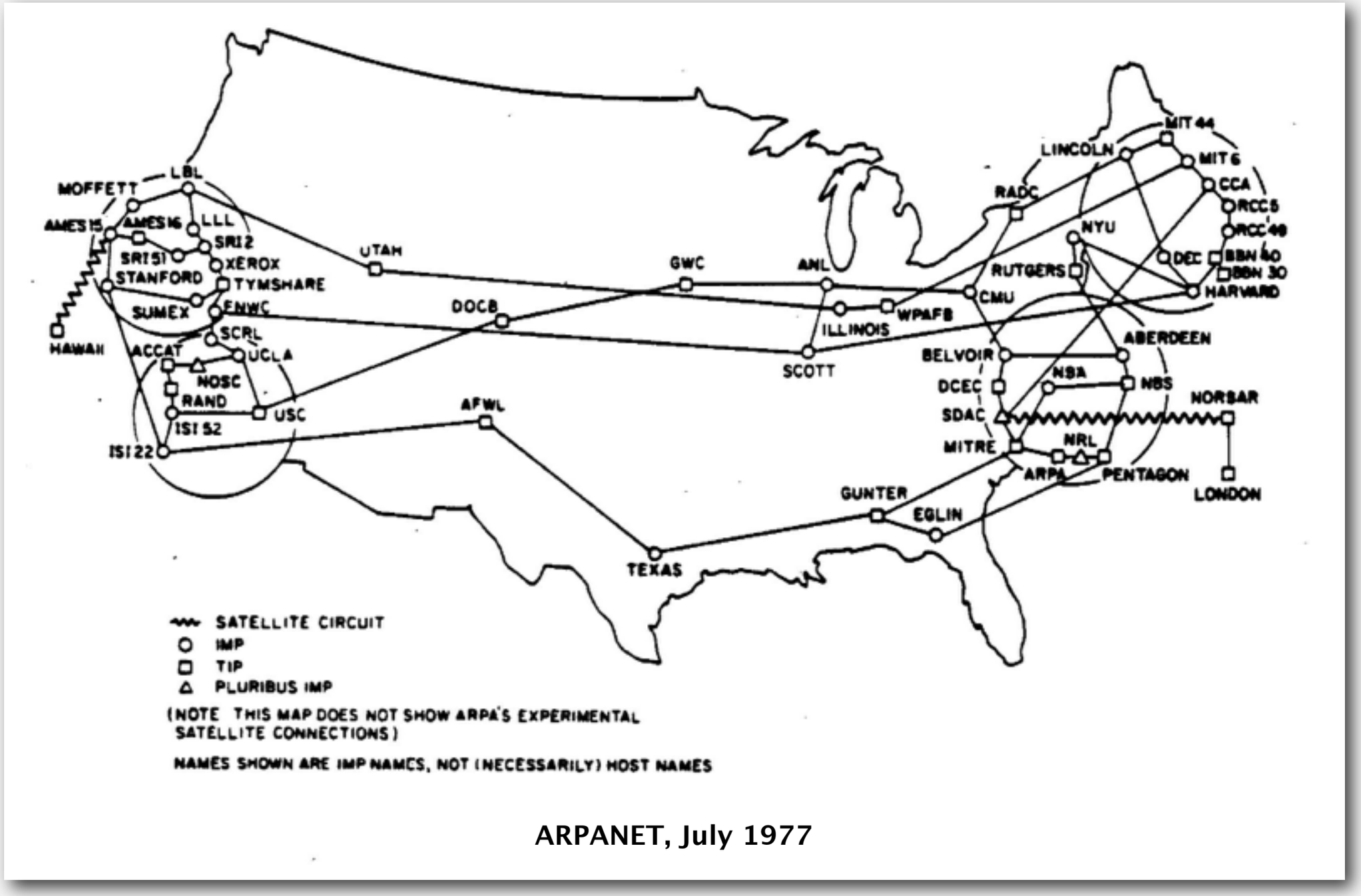
standard drawing



dist = 0 dist = 1 dist = 2

Breadth-first search application: routing

Fewest number of hops in a communication network.



ARPANET, July 1977

- graph API
- depth-first search
- breadth-first search
- **connected components**
- challenges

Connectivity queries

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

Goal. Preprocess graph to answer queries: is v connected to w ?
in constant time.

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

Union-Find? Not quite.

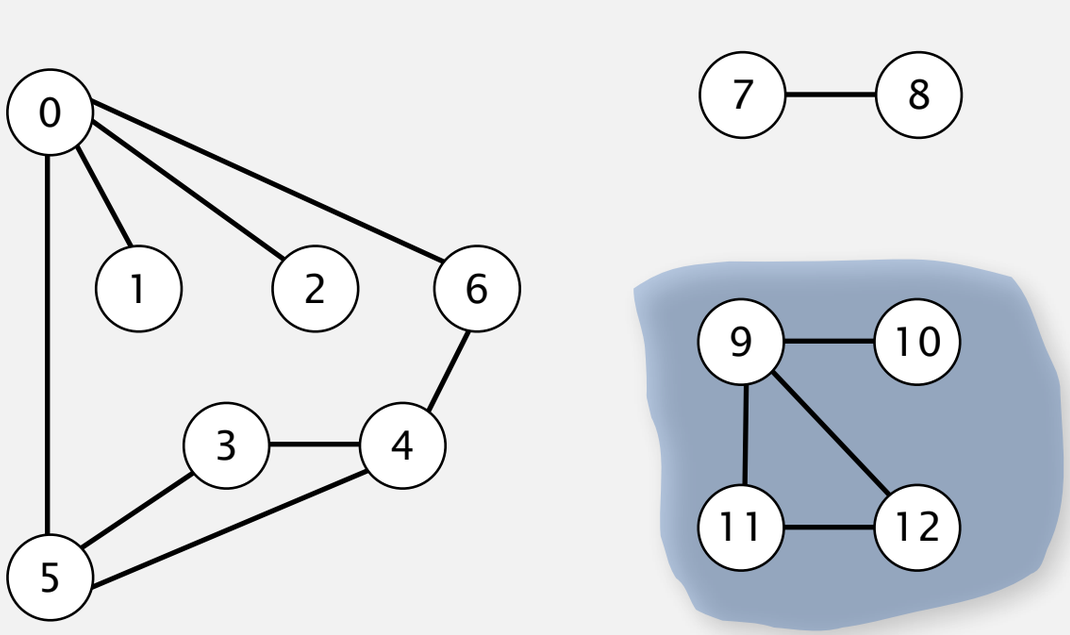
Depth-first search. Yes. [next few slides]

Connected components

The relation "is connected to" is an equivalence relation:

- Reflexive: v is connected to v.
- Symmetric: if v is connected to w, then w is connected to v.
- Transitive: if v connected to w and w connected to x, then v connected to x.

Def. A connected component is a maximal set of connected vertices.



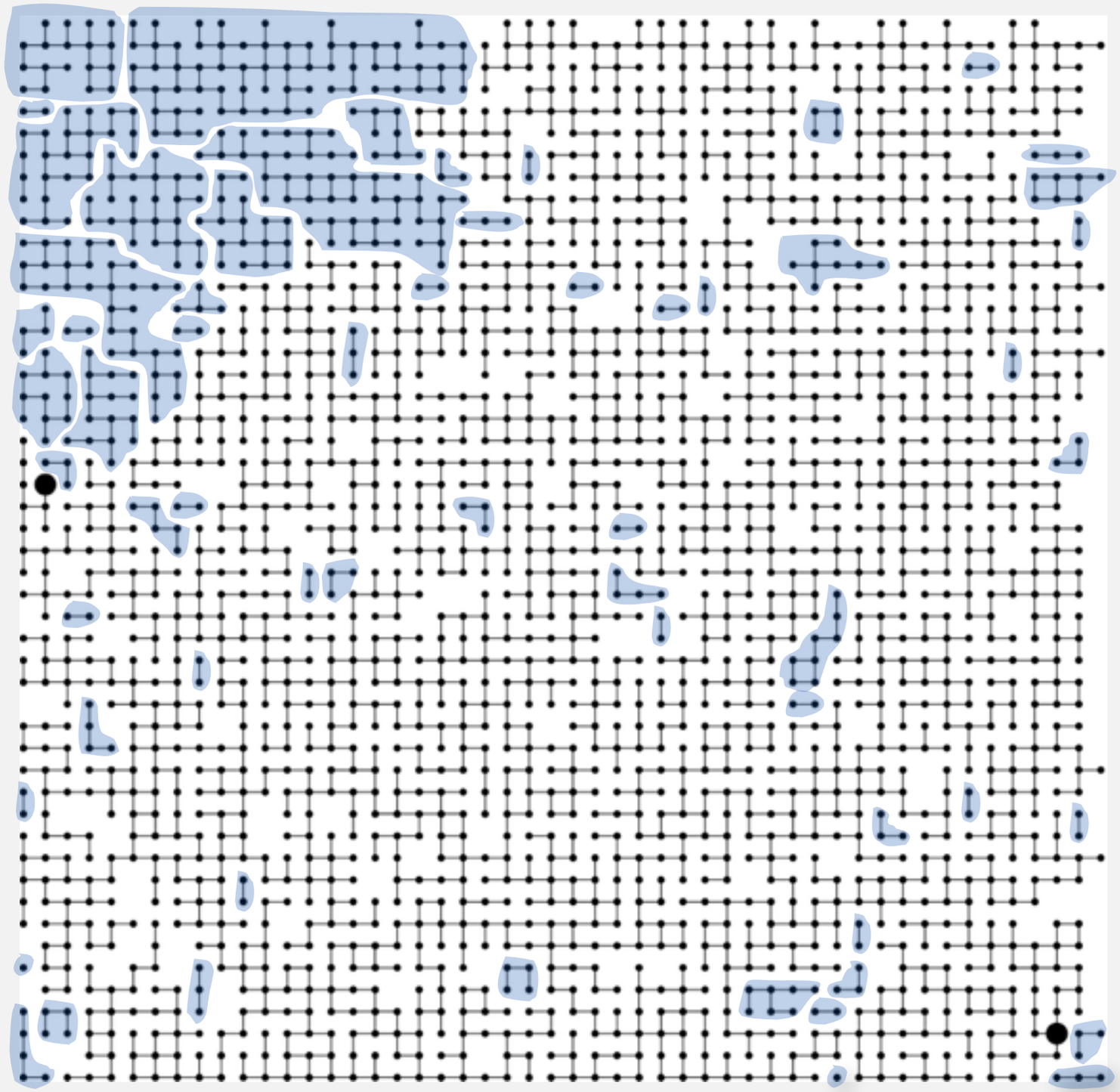
3 connected components

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

Remark. Given connected components, can answer queries in constant time.

Connected components

Def. A connected component is a maximal set of connected vertices.



63 connected components

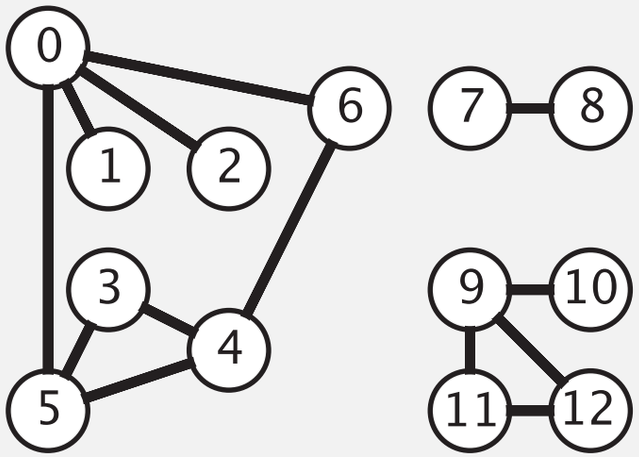
Connected components

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.



tinyG.txt

```
V → 13
    13 ← E
0  5
4  3
0  1
9  12
6  4
5  4
0  2
11  12
9  10
0  6
7  8
9  11
5  3
```

Finding connected components with DFS

```
public class CC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public CC(Graph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

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

}
```

id[v] = id of component containing v

number of components

run DFS from one vertex in each component

see next slide

Finding connected components with DFS (continued)

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

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

private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
}
```

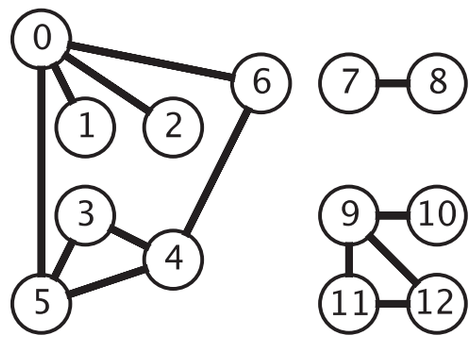
number of components

id of component containing v

all vertices discovered in same call of dfs have same id

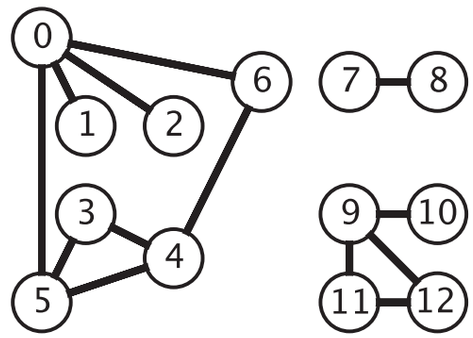
Finding connected components with DFS (trace)

```
                        count   marked[]                  id[]
                                0 1 2 3 4 5 6 7 8 9 10 11 12   0 1 2 3 4 5 6 7 8 9 10 11 12
dfs(0)                    0     T                              0
  dfs(6)                  0     T             T                0               0
    check 0
    dfs(4)                0     T       T     T              0         0     0
      dfs(5)              0     T       T T T                0         0 0 0
        dfs(3)            0     T     T T T T                0       0 0 0 0
          check 5
          check 4
        3 done
        check 4
        check 0
      5 done
      check 6
      check 3
    4 done
  6 done
  dfs(2)                  0     T     T T T T T                0     0 0 0 0 0
    check 0
  2 done
  dfs(1)                  0     T T T T T T T                0 0   0 0 0 0 0
    check 0
  1 done
  check 5
0 done
```



Finding connected components with DFS (trace)

	count	marked[]													id[]												
		0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12
0 done																											
dfs(7)	1	T	T	T	T	T	T	T	T						0	0	0	0	0	0	0	1					
| dfs(8)	1	T	T	T	T	T	T	T	T	T					0	0	0	0	0	0	0	1	1				
| | check 7																											
| 8 done																											
7 done																											
dfs(9)	2	T	T	T	T	T	T	T	T	T	T				0	0	0	0	0	0	0	1	1	2			
| dfs(11)	2	T	T	T	T	T	T	T	T	T	T		T		0	0	0	0	0	0	0	1	1	2		2	
| | check 9																											
| | dfs(12)	2	T	T	T	T	T	T	T	T	T	T		T	T	0	0	0	0	0	0	0	1	1	2		2	2
| | | check 11																											
| | | check 9																											
| | 12 done																											
| 11 done																											
| dfs(10)	2	T	T	T	T	T	T	T	T	T	T	T	T	T	0	0	0	0	0	0	0	1	1	2	2	2	2
| | check 9																											
| 10 done																											
| check 12																											
9 done																											



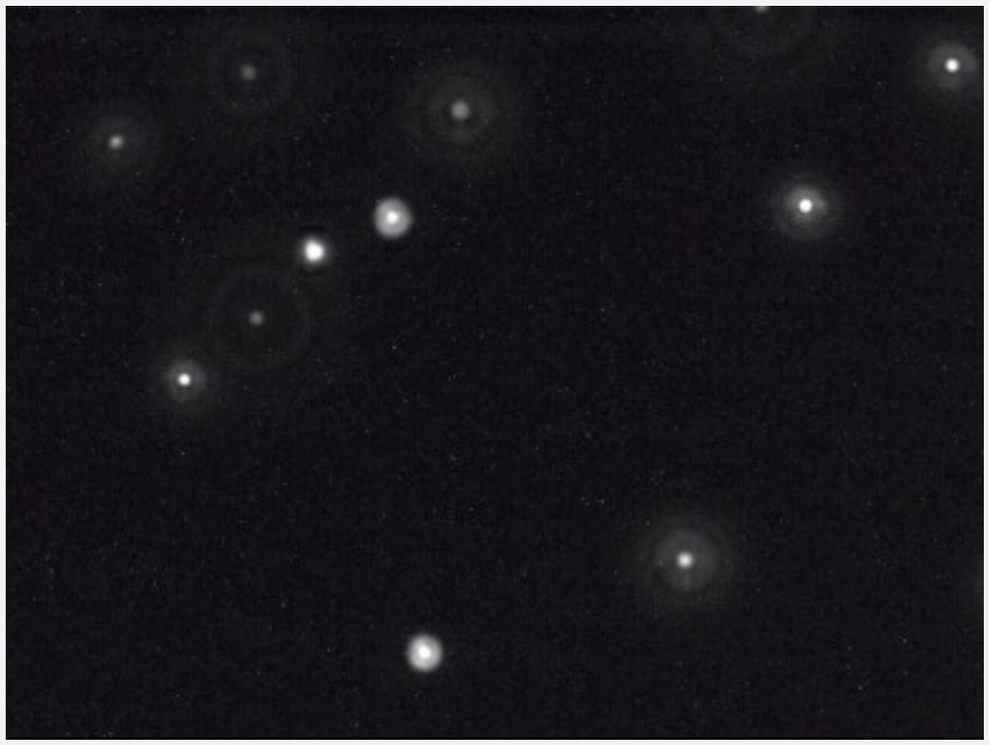
Connected components application: particle detection

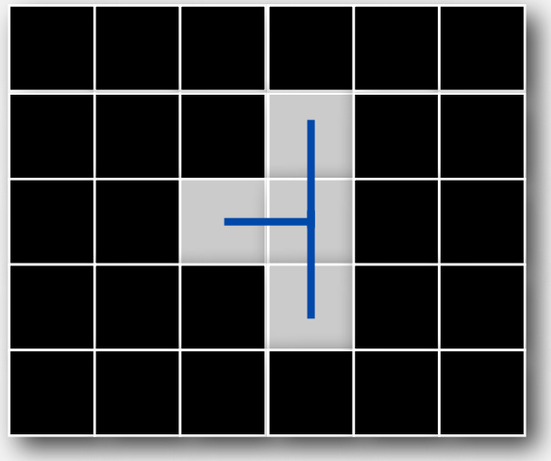
Particle detection. Given grayscale image of particles, identify "blobs."

- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value ≥ 70.
- Blob: connected component of 20-30 pixels.

black = 0

white = 255

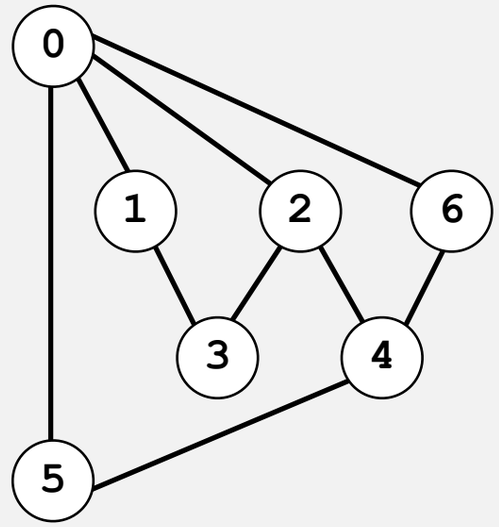




Particle tracking. Track moving particles over time.

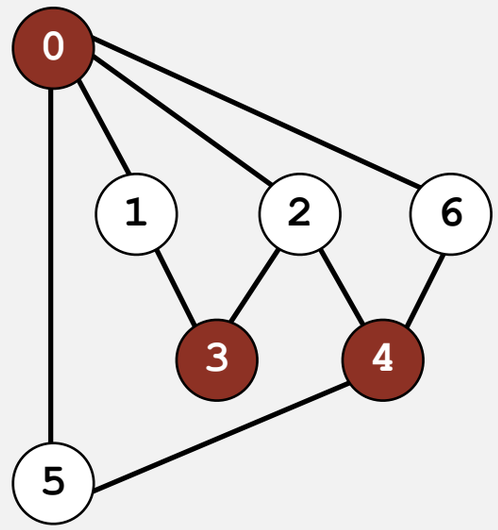
Graph-processing challenge 1

Problem. Is a graph bipartite? (from wikipedia: the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V (that is, U and V are each independent sets) such that every edge connects a vertex in U to one in V. Vertex set U and V are often denoted as partite sets)



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

How?

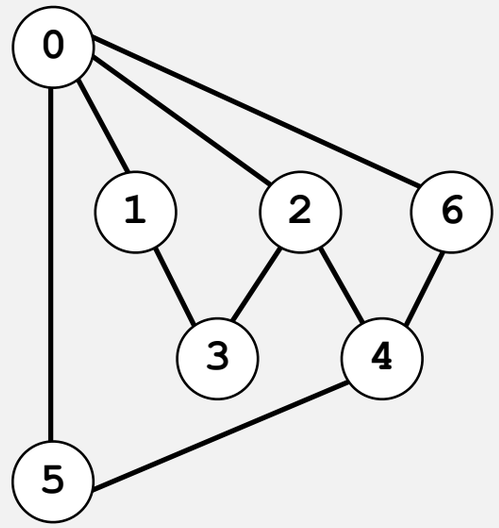


0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

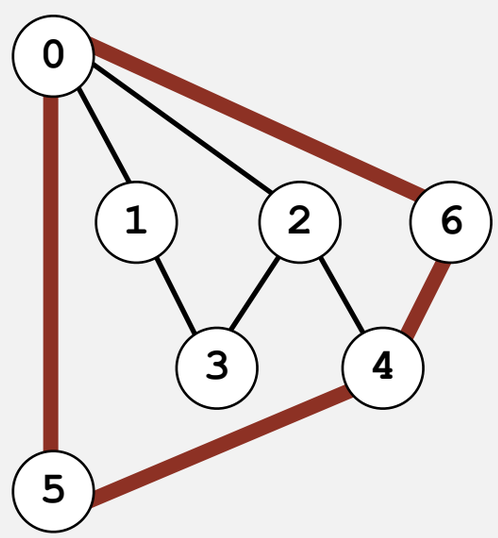
Graph-processing challenge 2

Problem. Find a cycle.

How?



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

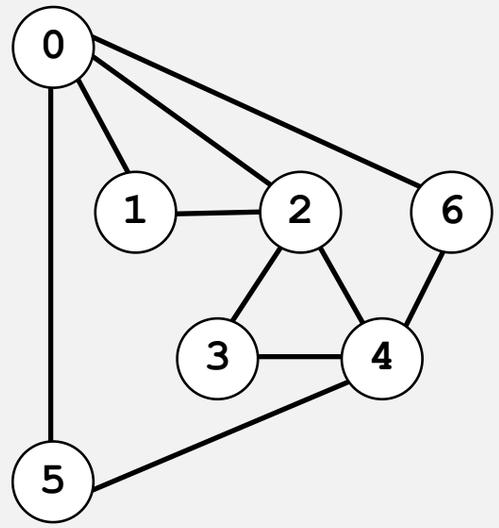


Graph-processing challenge 3

Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.

How difficult?



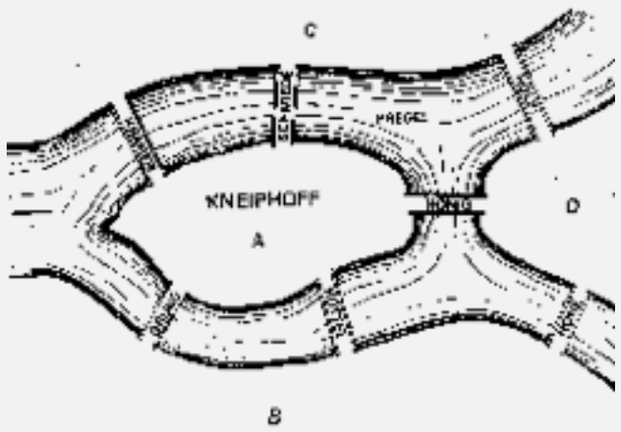
0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6

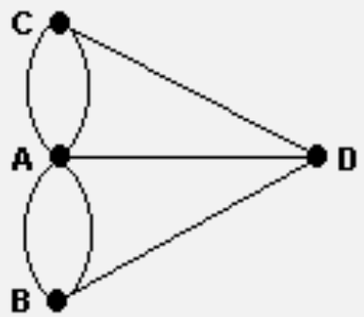
0-1-2-3-4-2-0-6-4-5-0

Bridges of Königsberg

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

> *" … in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches ... and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once. "*





Euler tour. Is there a (general) cycle that uses each edge exactly once?

Answer. Yes iff connected and all vertices have even degree.

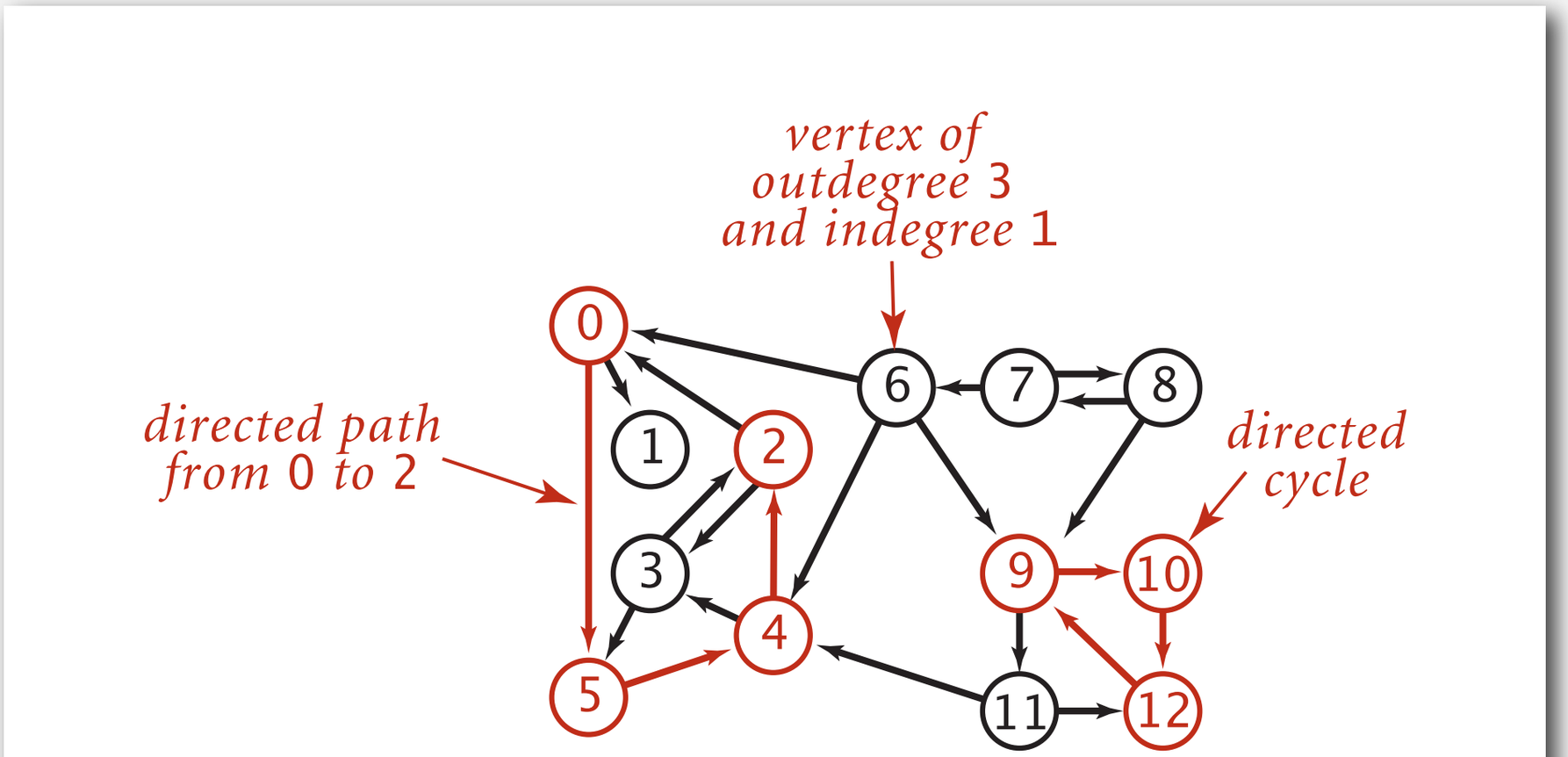
To find path. DFS-based algorithm (see textbook).

Directed Graphs

- digraph API
- digraph search
- topological sort
- strong components

Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



Digraph applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

Path. Is there a directed path from s to t ?

Shortest path. What is the shortest directed path from s to t ?

Topological sort. Can you draw the digraph so that all edges point upwards?

Strong connectivity. Is there a directed path between all pairs of vertices?

Transitive closure. For which vertices v and w is there a path from v to w ?

PageRank. What is the importance of a web page?

- digraph API
- digraph search
- topological sort
- strong components

Digraph API

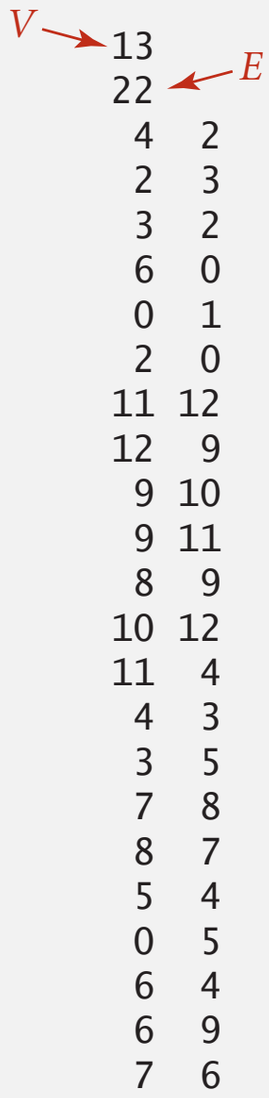
public class Digraph	
	`Digraph(int V)` — *create an empty digraph with V vertices*
	`Digraph(In in)` — *create a digraph from input stream*
`void`	`addEdge(int v, int w)` — *add a directed edge v→w*
`Iterable<Integer>`	`adj(int v)` — *vertices pointing from v*
`int`	`V()` — *number of vertices*
`int`	`E()` — *number of edges*
`Digraph`	`reverse()` — *reverse of this digraph*
`String`	`toString()` — *string representation*

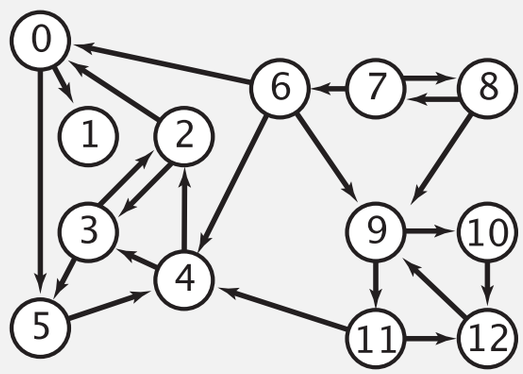
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

read digraph from input stream

print out each edge (once)

Digraph API






```
% java TestDigraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
...
11->4
11->12
12-9
```

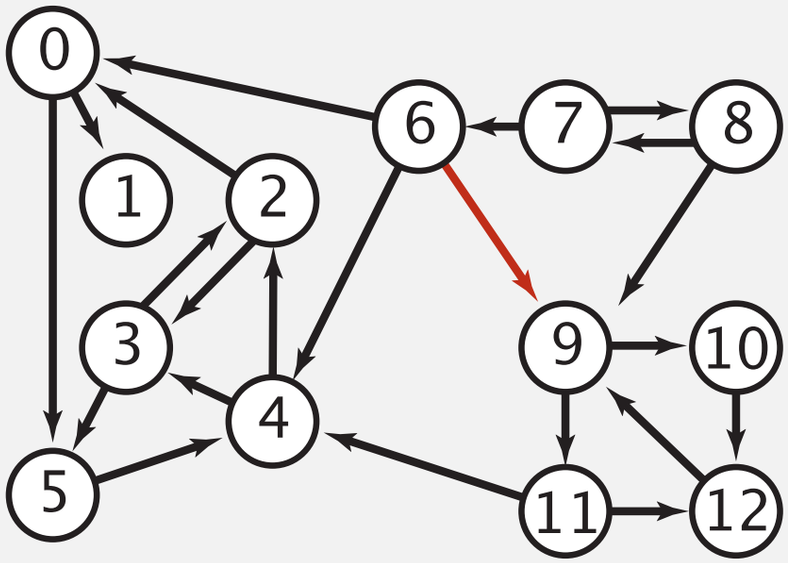
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

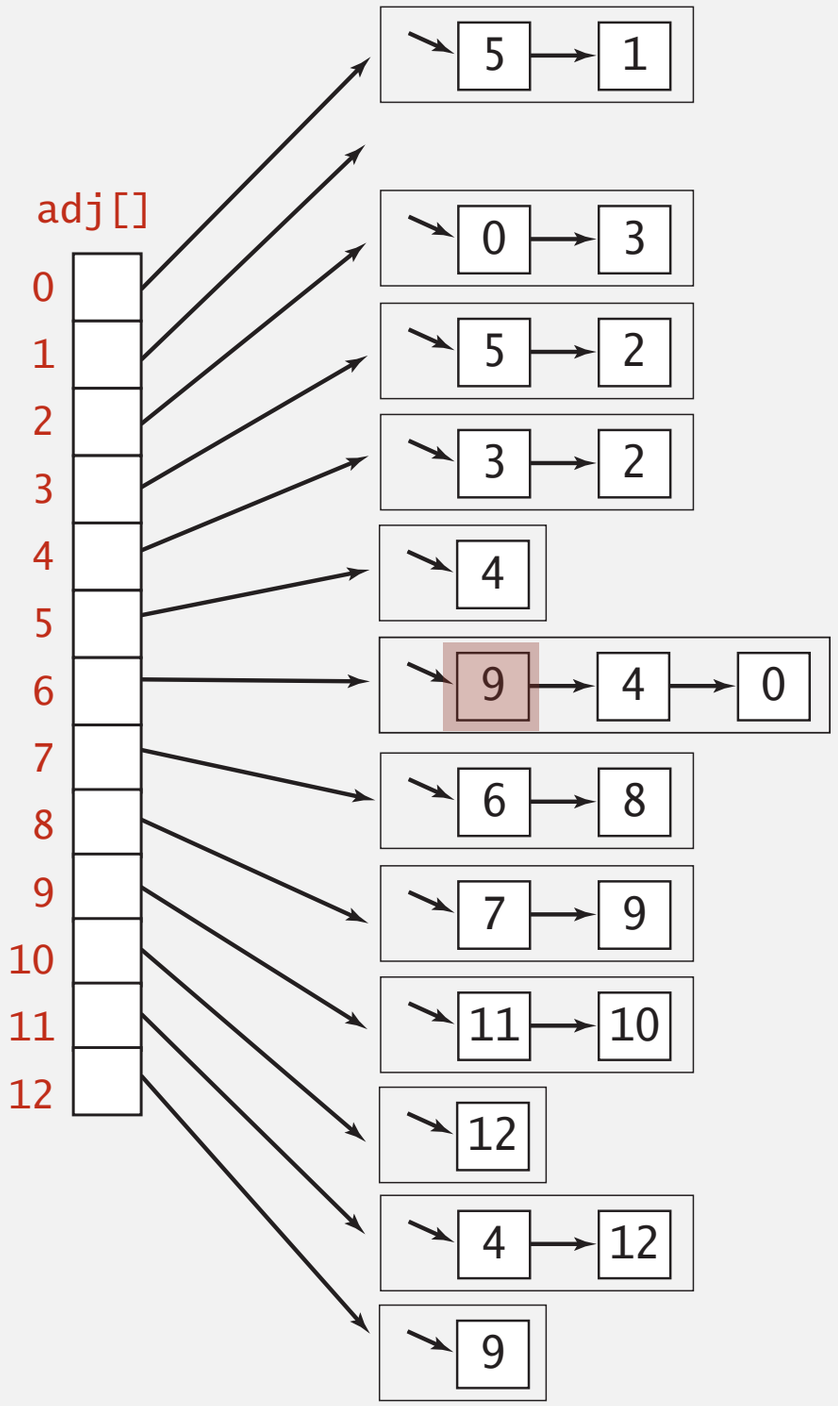
read digraph from input stream

print out each edge (once)

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists





Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v.
- Real-world digraphs tend to be sparse.

huge number of vertices,
small average vertex degree

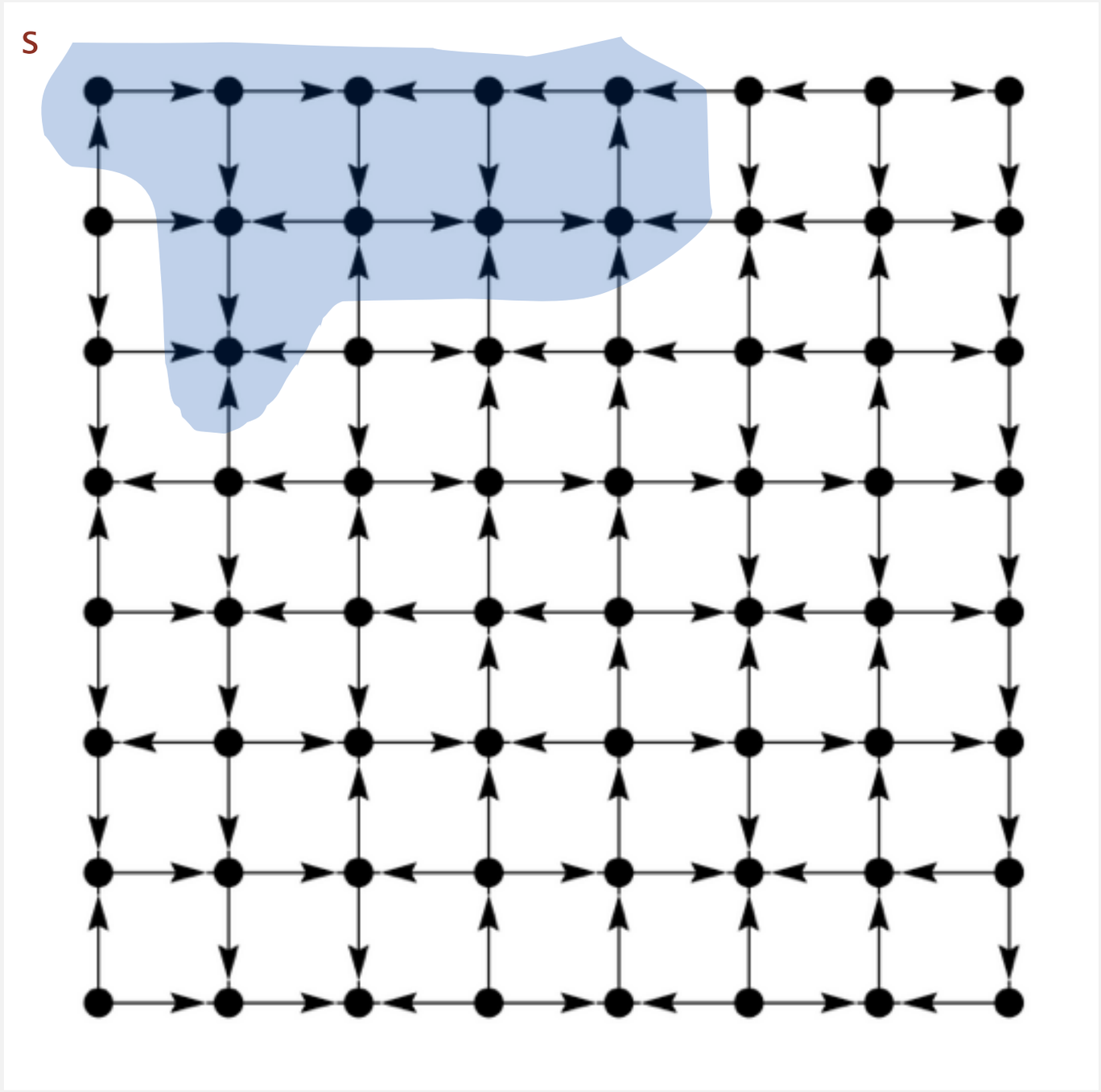
representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing from v?
list of edges	E	1	E	E
adjacency matrix	V	1	1	V
adjacency lists	E + V	1	outdegree(v)	outdegree(v)

† disallows parallel edges

- digraph API
- **digraph search**
- topological sort
- strong components

Reachability

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



Depth-first search in digraphs

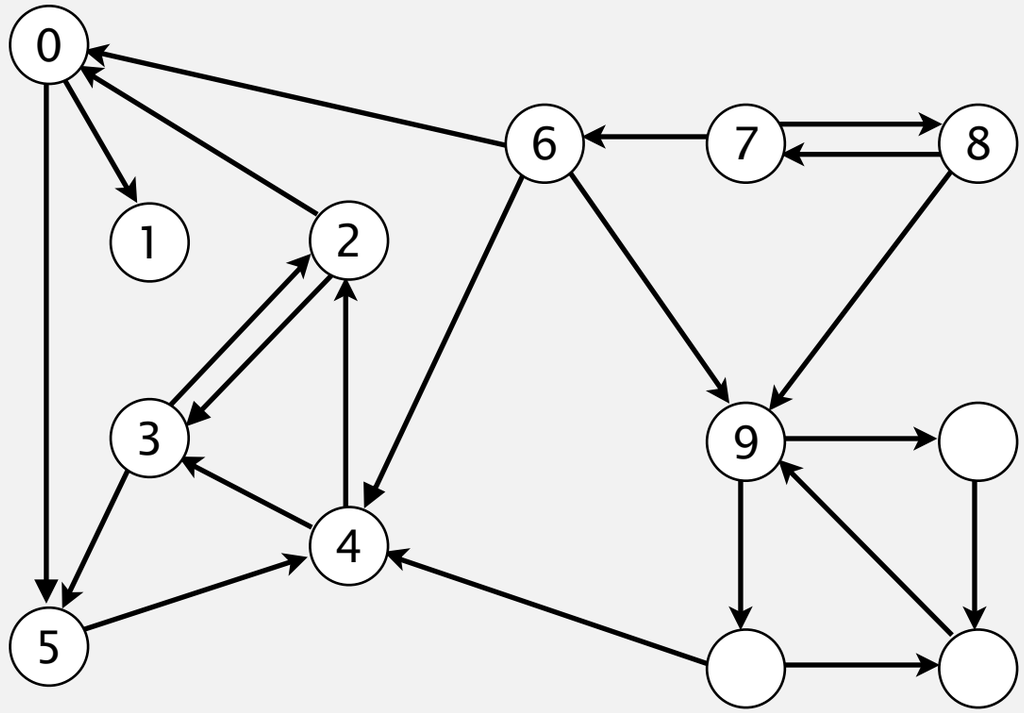
Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked vertices w pointing from v.



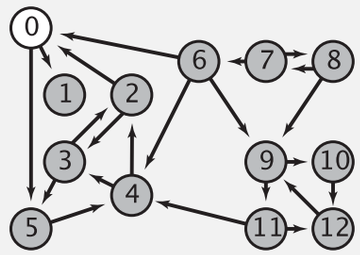
Depth-first search demo

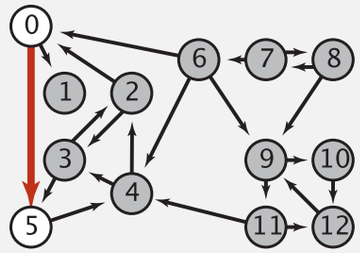
DFS pathfinding trace in a digraph

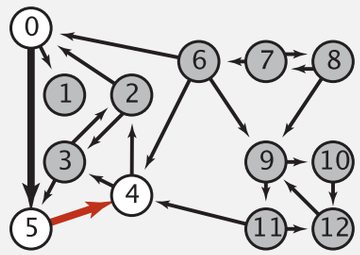
order depends on digraph representation

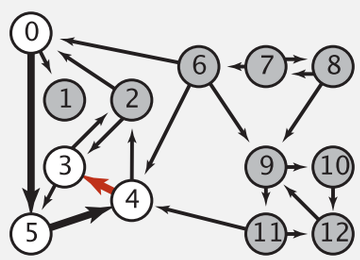
```
dfs(0)
  dfs(5)
    dfs(4)
      dfs(3)
        check 5
        dfs(2)
          check 0
          check 3
        2 done
      3 done
      check 2
    4 done
  5 done
  dfs(1)
  1 done
0 done
```

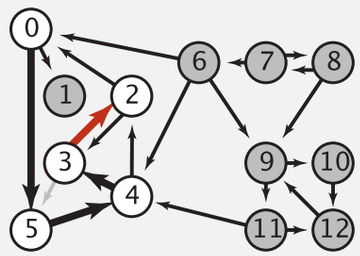
marked[]							edgeTo[]						
0	1	2	3	4	5	...	0	1	2	3	4	5	...
1	0	0	0	0	0		-	-	-	-	-	0	
1	0	0	0	0	1		-	-	-	-	5	0	
1	0	0	0	1	1		-	-	-	4	5	0	
1	0	0	1	1	1		-	-	-	4	5	0	
1	0	0	1	1	1		-	-	3	4	5	0	
1	0	1	1	1	1		-	-	3	4	5	0	
1	0	1	1	1	1		-	-	3	4	5	0	
1	0	1	1	1	1		-	-	3	4	5	0	
1	0	1	1	1	1		-	-	3	4	5	0	
1	0	1	1	1	1		-	-	3	4	5	0	
1	0	1	1	1	1		-	0	3	4	5	0	
1	1	1	1	1	1		-	0	3	4	5	0	
1	1	1	1	1	1		-	0	3	4	5	0	

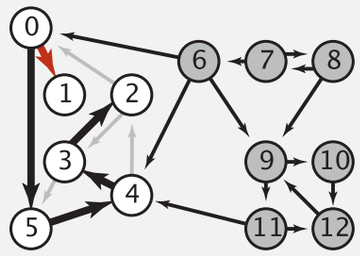












Reachability application: program control-flow analysis

Every program is a digraph.

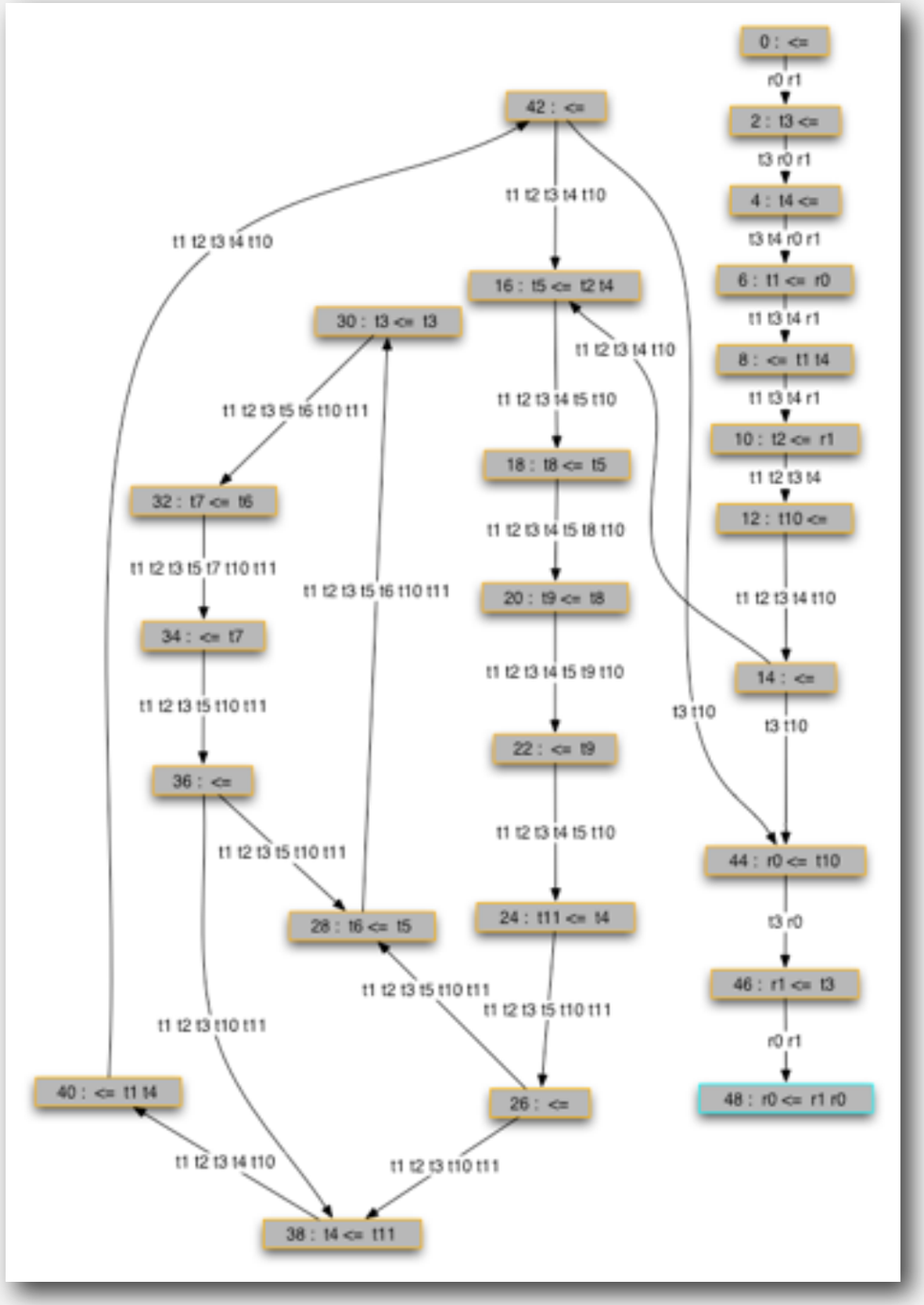
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



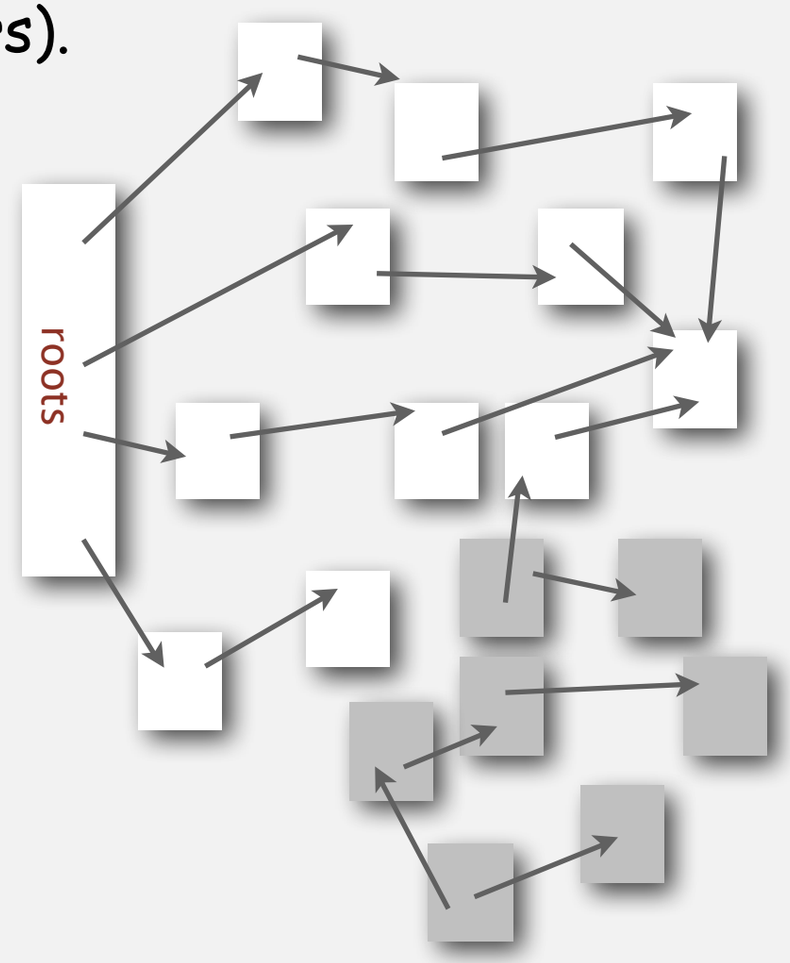
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).

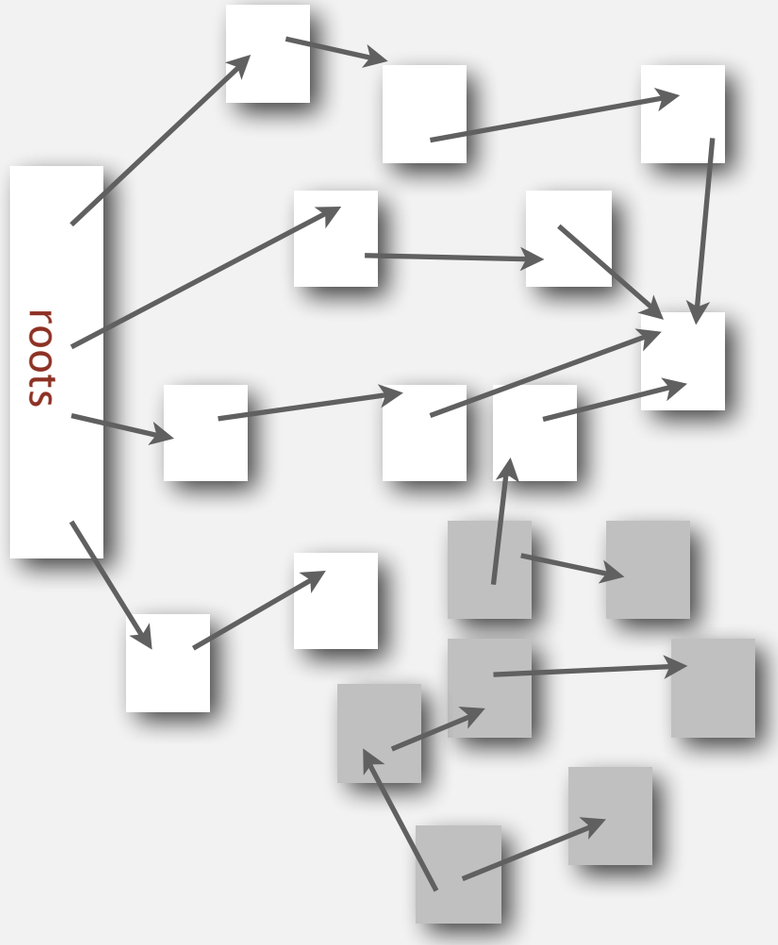


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



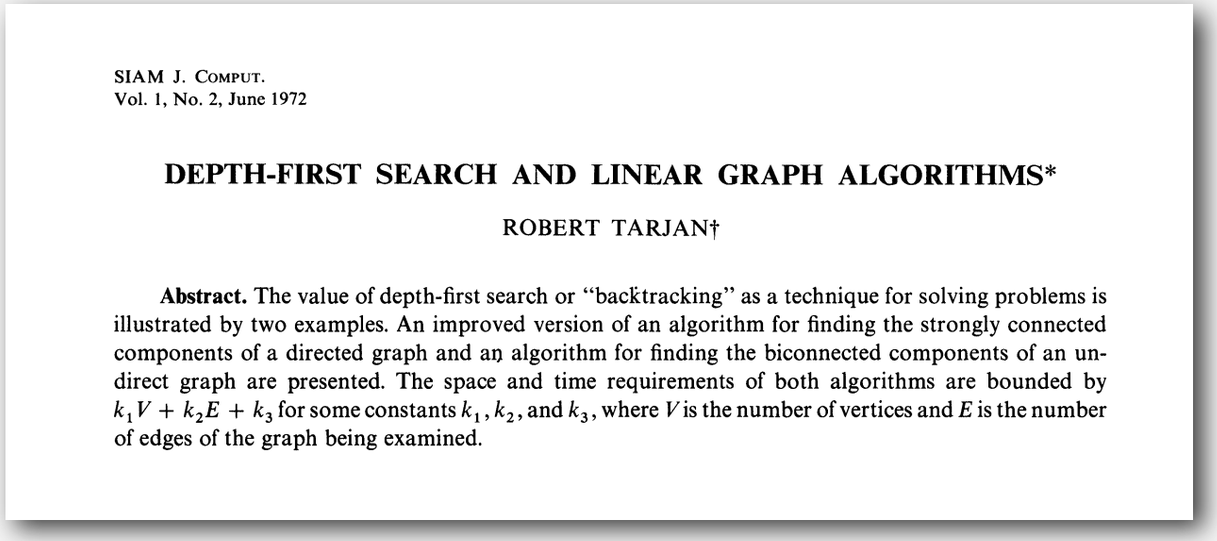
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.

✓ • Reachability.
- Path finding.
- Topological sort.
- Transitive closure.
- Directed cycle detection.

Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.



SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Breadth-first search in digraphs

Same method as for undirected graphs.

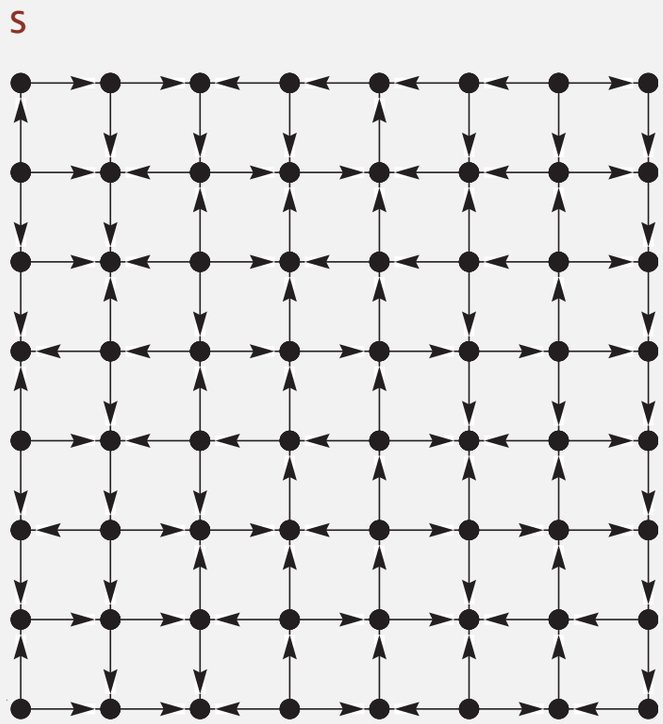
- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:**
  add to queue and mark as visited.

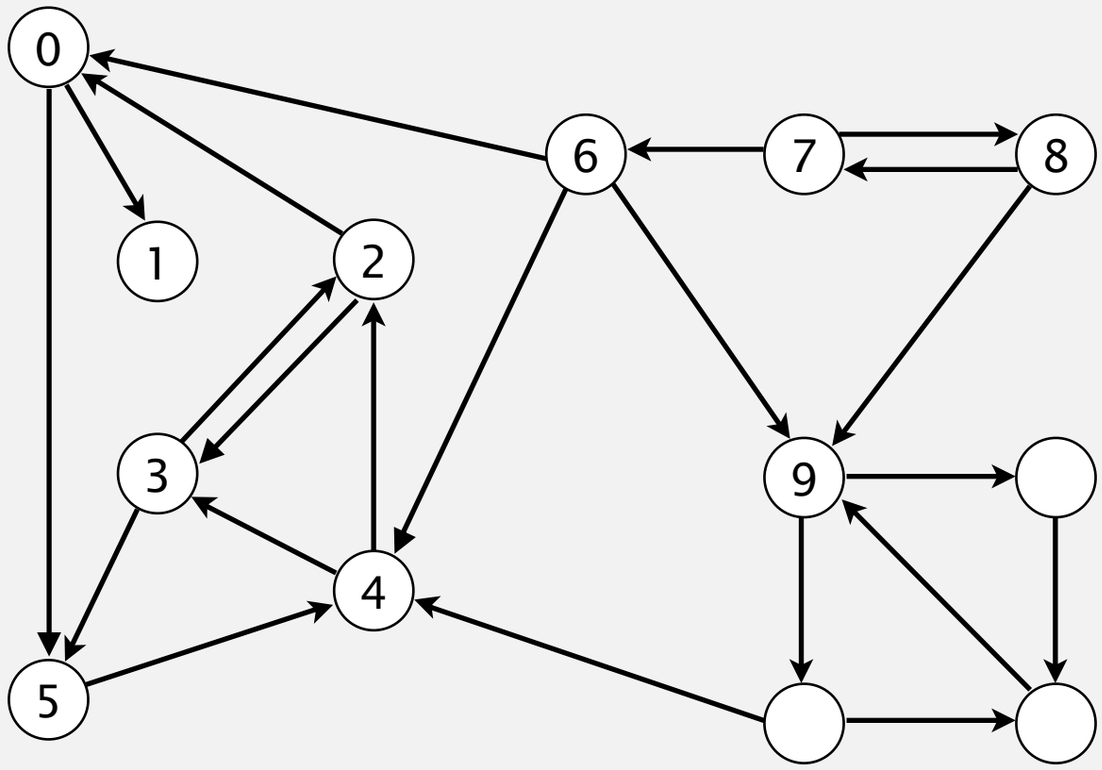


Proposition. BFS computes shortest paths (fewest number of edges).

Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex.

Ex. Shortest path from $\{ 1, 7, 10 \}$ to 5 is $7 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 5$.



Q. How to implement multi-source constructor?

A. Use BFS, but initialize by enqueuing all source vertices.

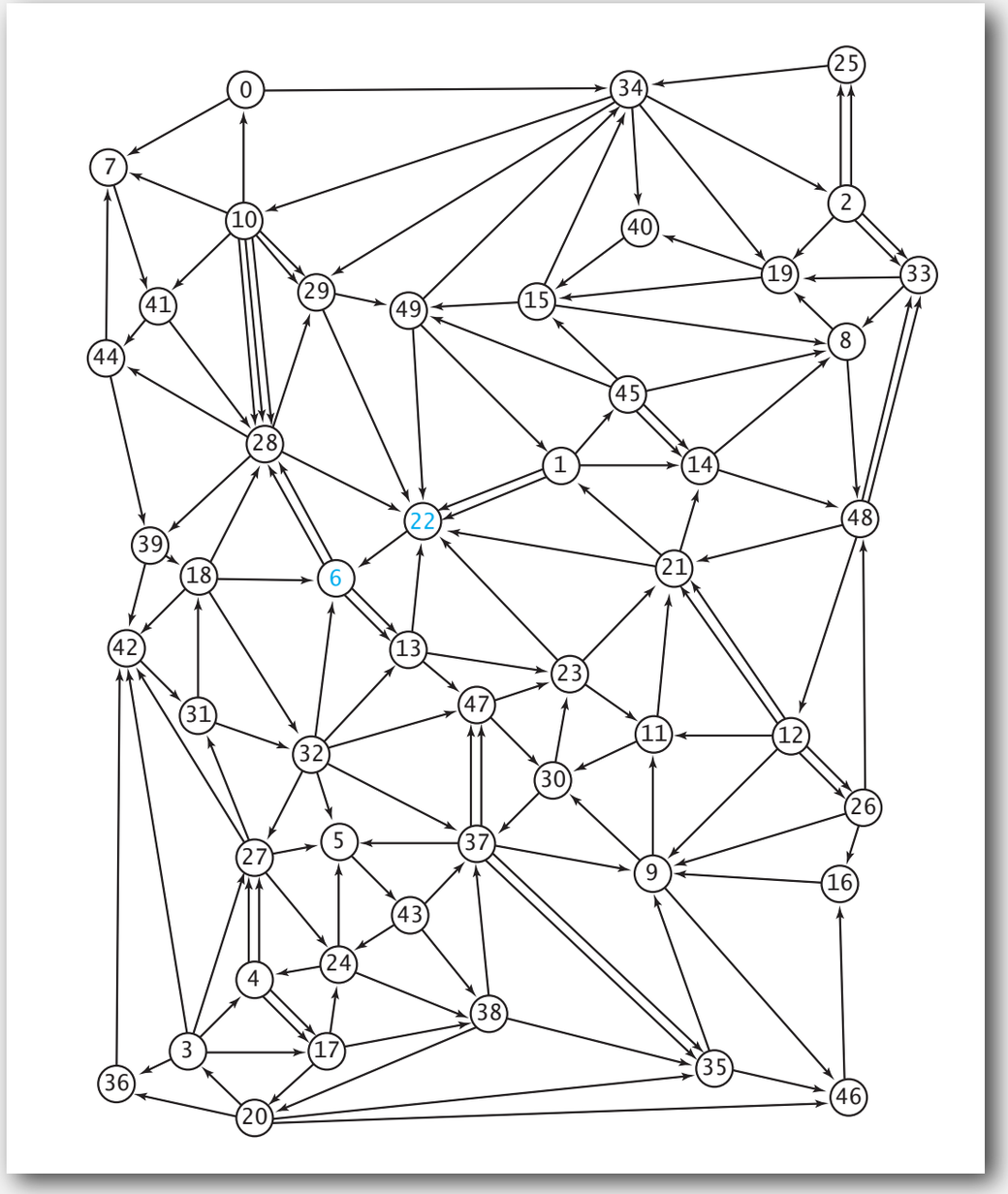
Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say `www.princeton.edu`.

Solution. BFS with implicit graph.

BFS.

- Choose root web page as source s.
- Maintain a `Queue` of websites to explore.
- Maintain a `SET` of discovered websites.
- Dequeue the next website and enqueue websites to which it links (provided you haven't done so before).



Q. Why not use DFS?

- digraph API
- digraph search
- **topological sort**
- strong components

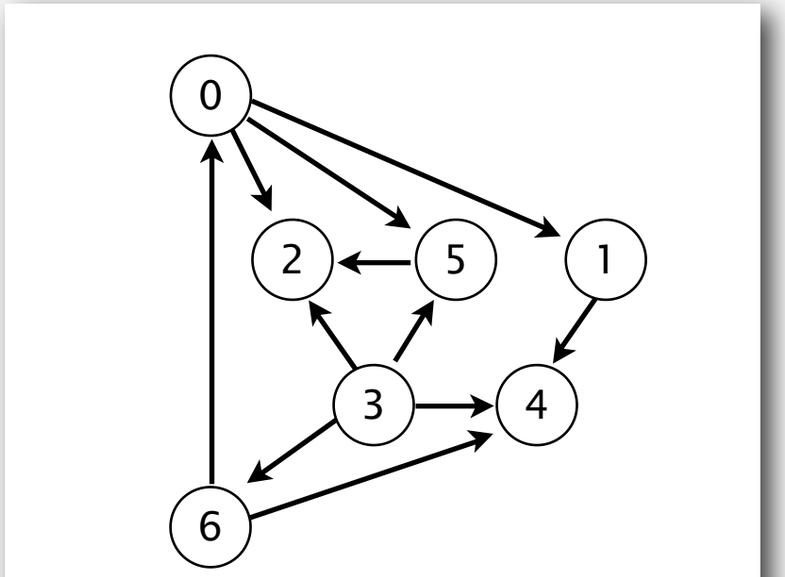
Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

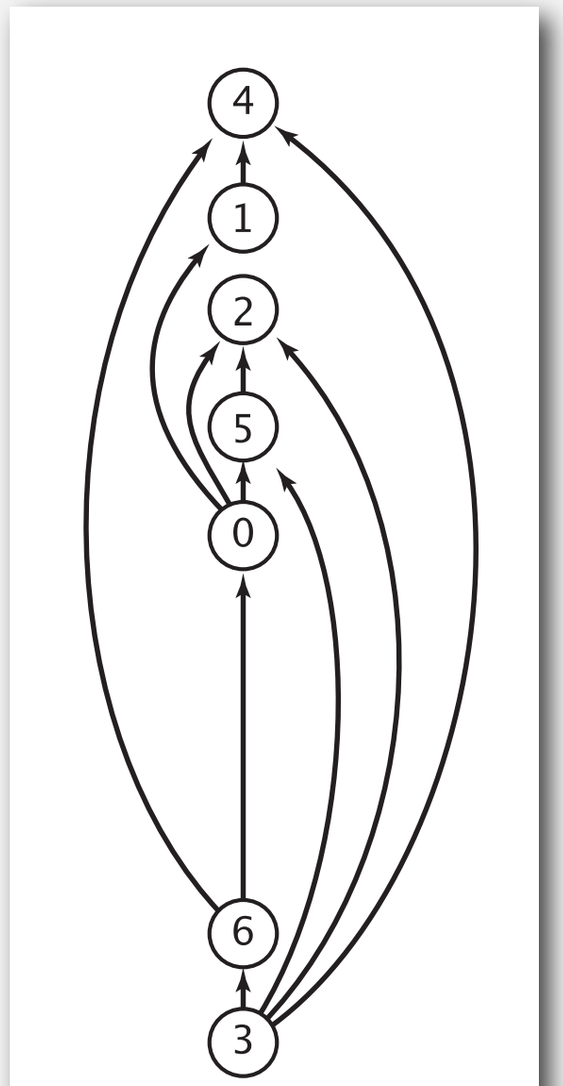
Digraph model. vertex = task; edge = precedence constraint.

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

tasks



precedence constraint graph

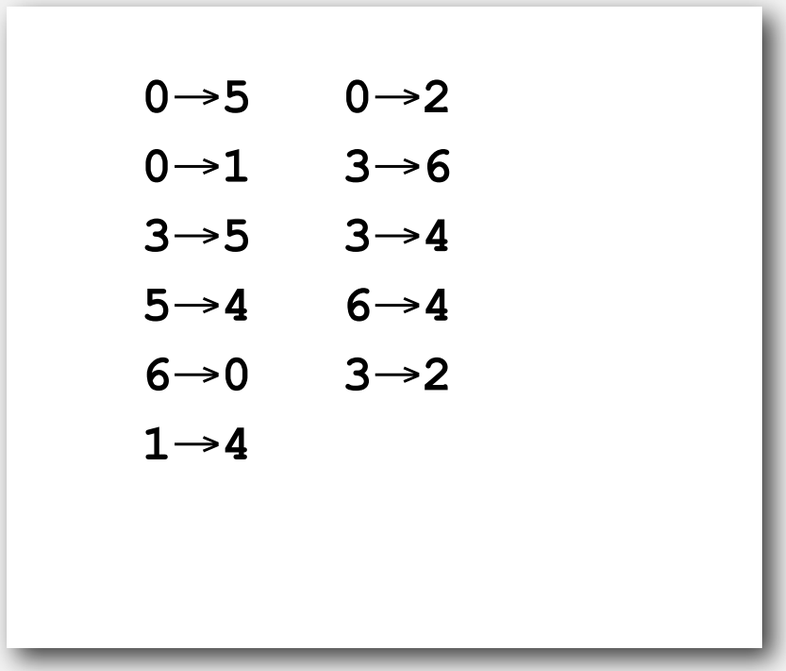


feasible schedule

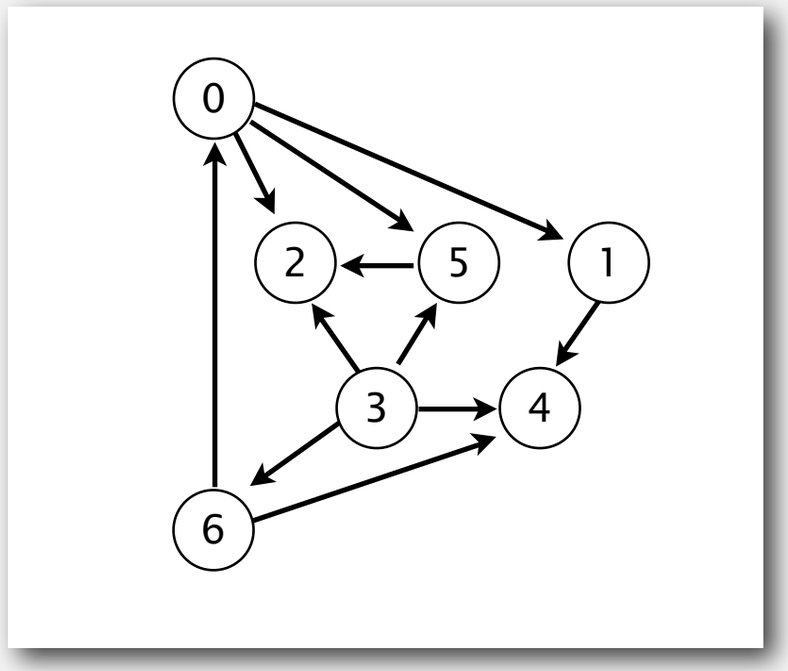
Topological sort

DAG. Directed acyclic graph.

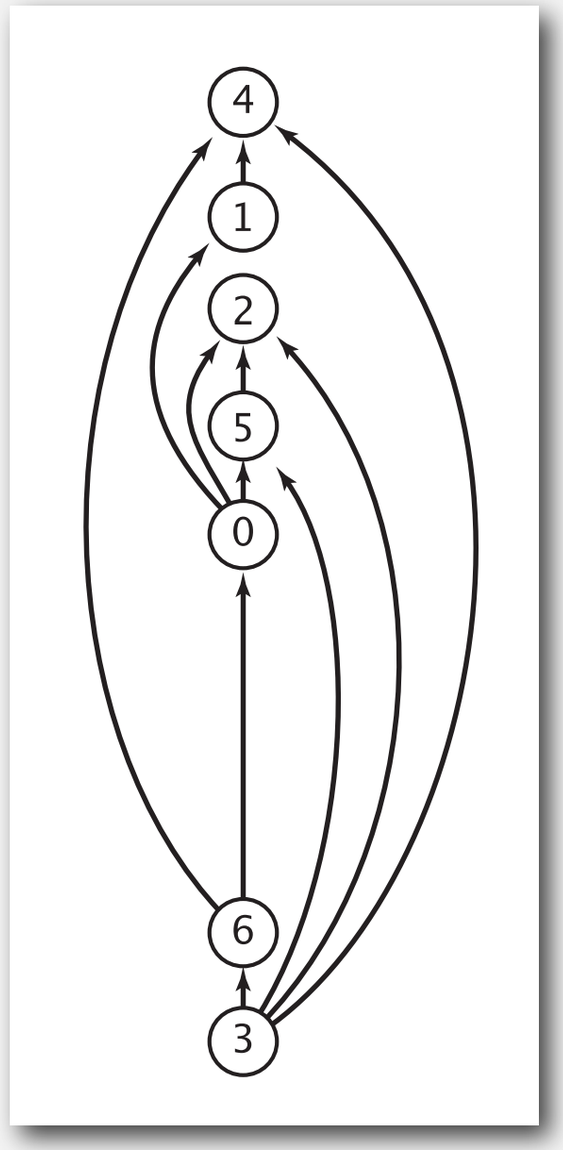
Topological sort. Redraw DAG so all edges point upwards.



directed edges



DAG



topological order

Solution. DFS. What else?

Topological sort demo

Topological sort in a DAG: correctness proof

Proposition. Reverse DFS postorder of a DAG is a topological order.

postorder == put vertex on a stack after recursive call

reverse postorder = not the same as preorder!

Pf. Consider any edge $v \rightarrow w$. When `dfs(v)` is called:

- Case 1: `dfs(w)` has already been called and returned.
  Thus, w was done before v.

- Case 2: `dfs(w)` has not yet been called.
  `dfs(w)` will get called directly or indirectly
  by `dfs(v)` and will finish before `dfs(v)`.
  Thus, w will be done before v.

- Case 3: `dfs(w)` has already been called,
  but has not yet returned.
  Can't happen in a DAG: function call stack contains all vertices pointing from 3 are done before 3 is done, so they appear after 3 in topological order

```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
0 done
check 1
check 2
dfs(3)
    check 2
    check 4
    check 5
    dfs(6)
    6 done
3 done
check 4
check 5
check 6
done
```

Ex: → dfs(3)

case 1 → check 2, check 4, check 5

case 2 → dfs(6), 6 done

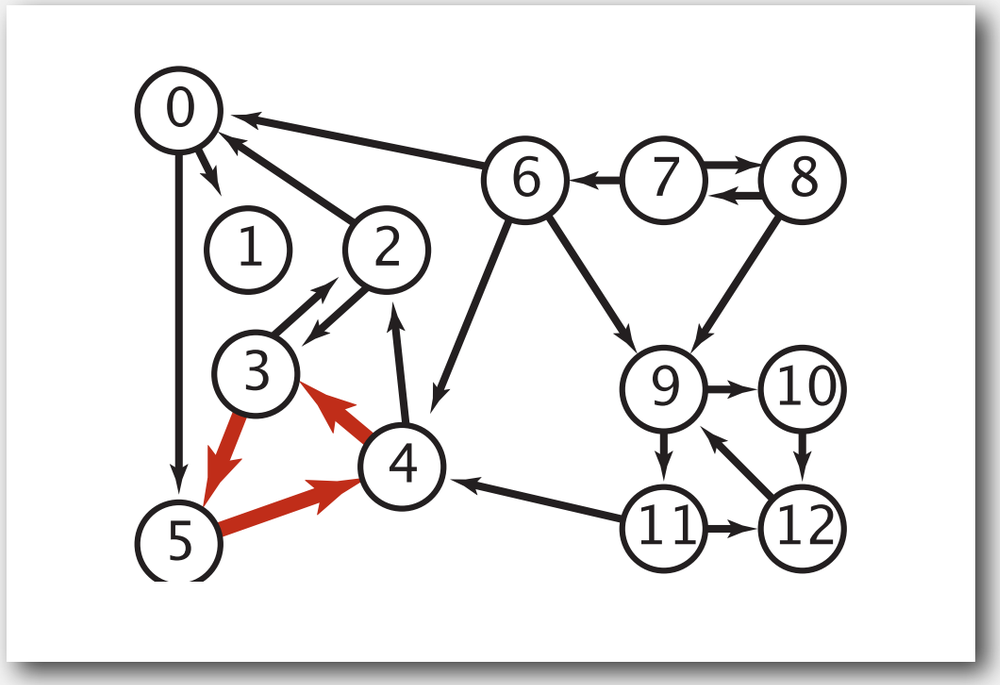
Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.

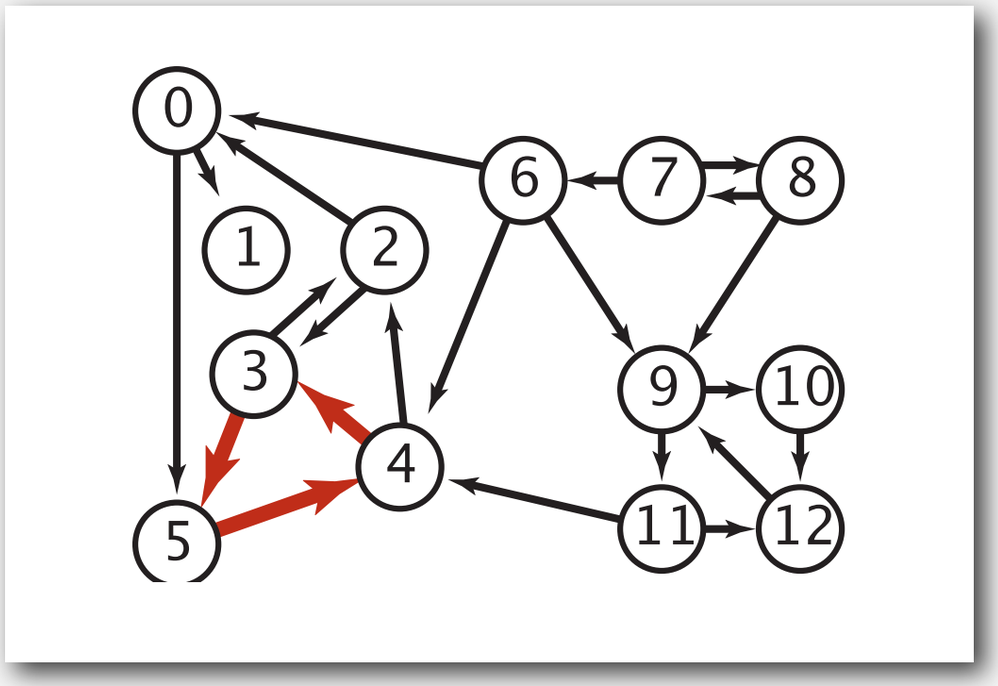
Goal. Given a digraph, find a directed cycle.



Solution. DFS. What else?

Directed cycle detection application: precedence scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?



precedence constraint graph is not a DAG

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection applications

- Causalities.
- Email loops.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Precedence scheduling.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.

- digraph API
- digraph search
- topological sort
- **strong components**

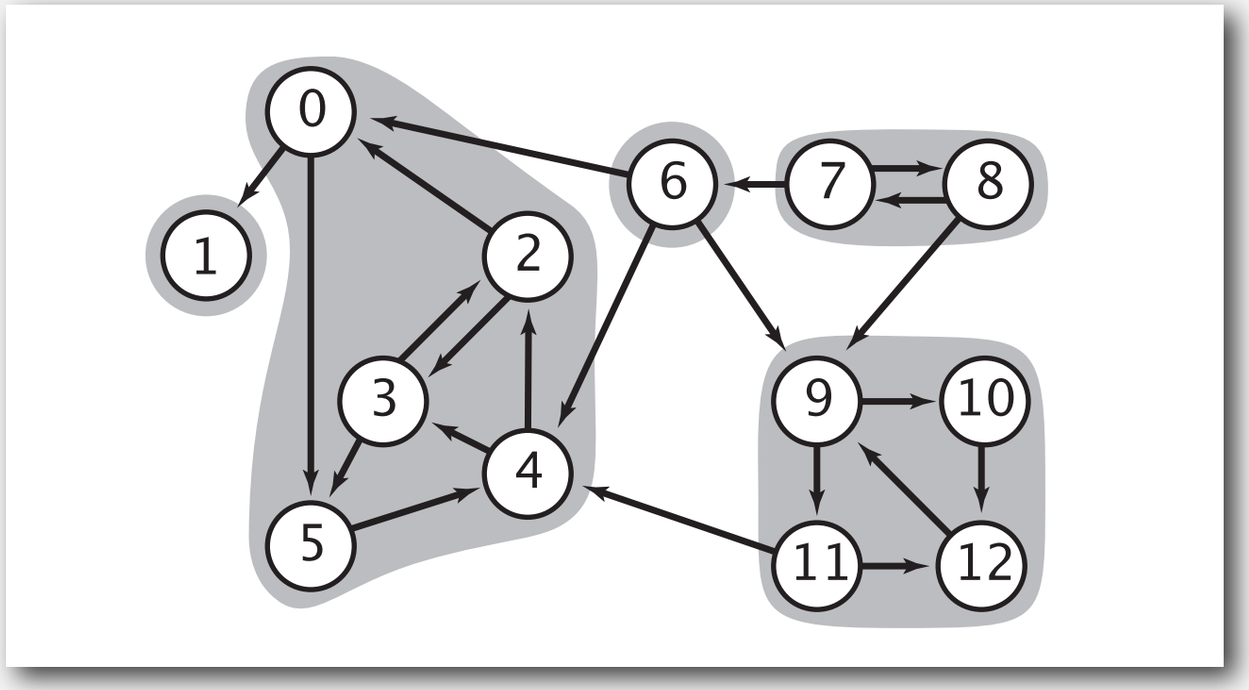
Strongly-connected components

Def. Vertices v and w are strongly connected if there is a directed path from v to w and a directed path from w to v.

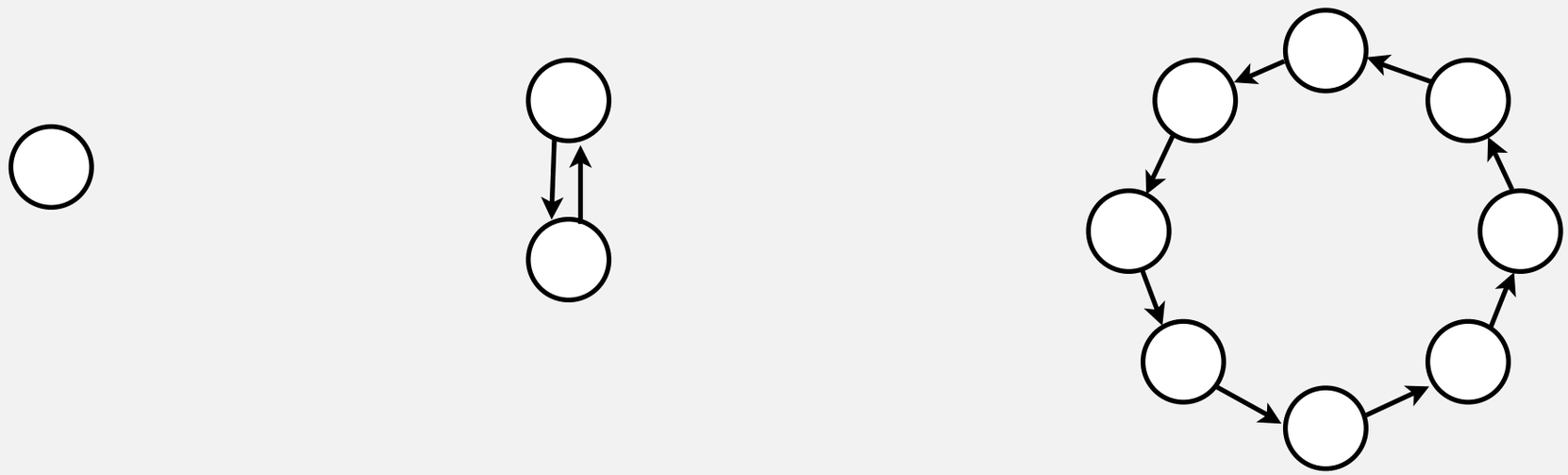
Key property. Strong connectivity is an equivalence relation:

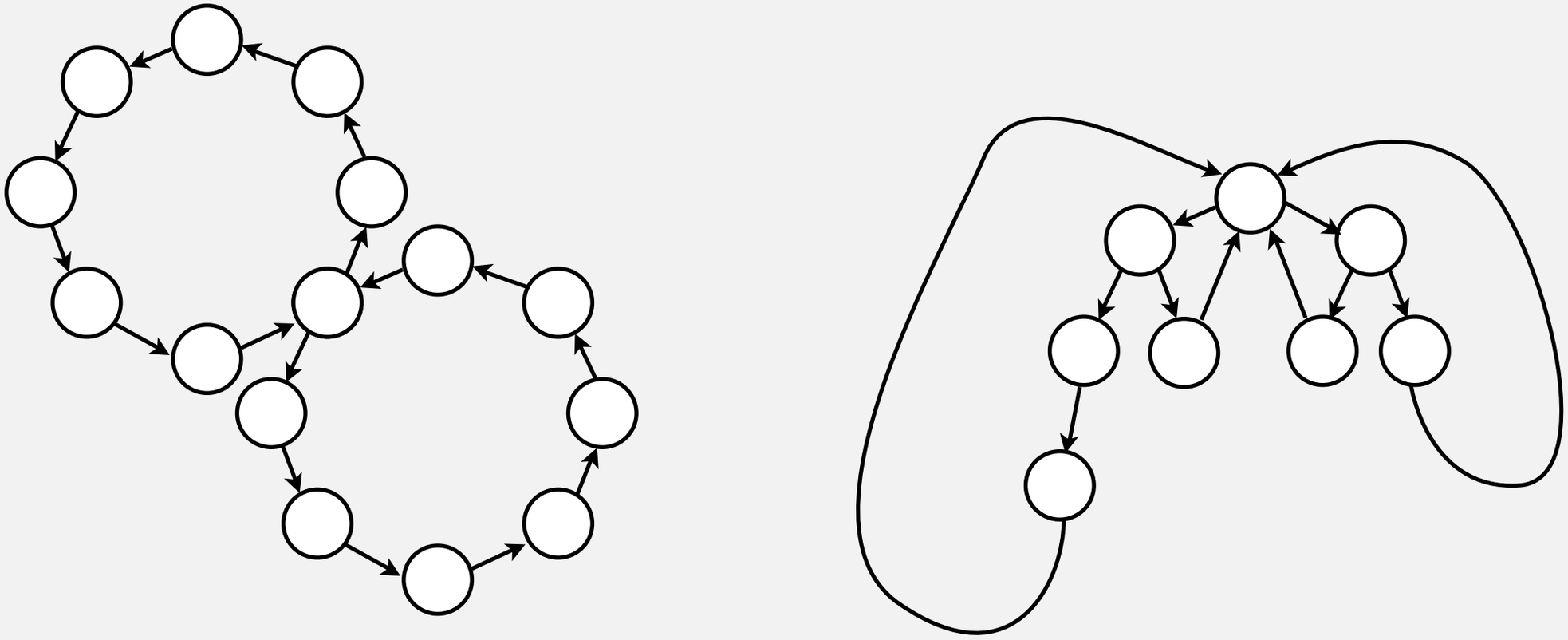
- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is strongly connected to x.

Def. A strong component is a maximal subset of strongly-connected vertices.



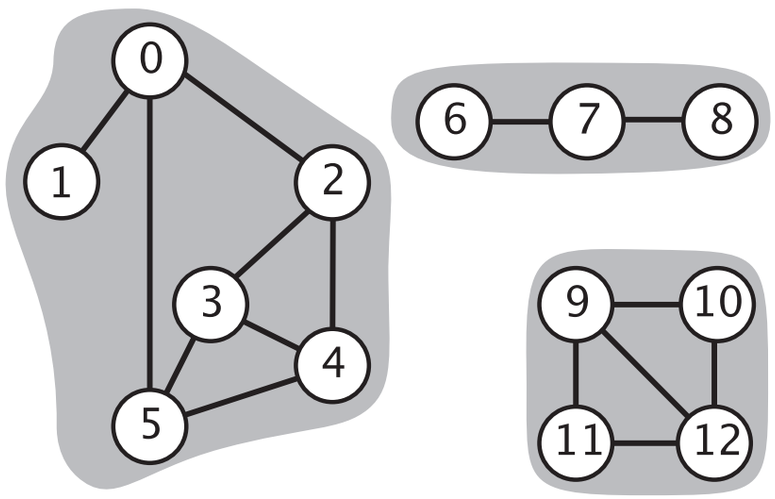
Examples of strongly-connected digraphs





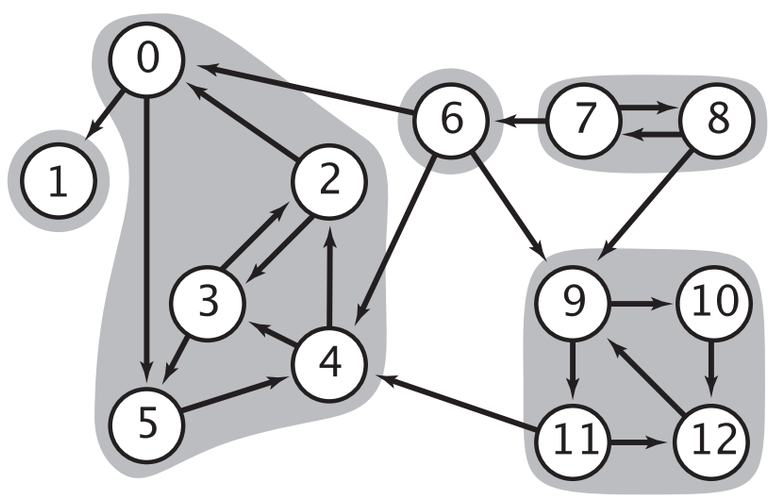
Connected components vs. strongly-connected components

v and w are connected if there is a path between v and w



3 connected components

v and w are strongly connected if there is a directed path from v to w and a directed path from w to v



5 strongly-connected components

connected component id (easy to compute with DFS)

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

```
public int connected(int v, int w)
{  return cc[v] == cc[w];  }
```

constant-time client connectivity query

strongly-connected component id (how to compute?)

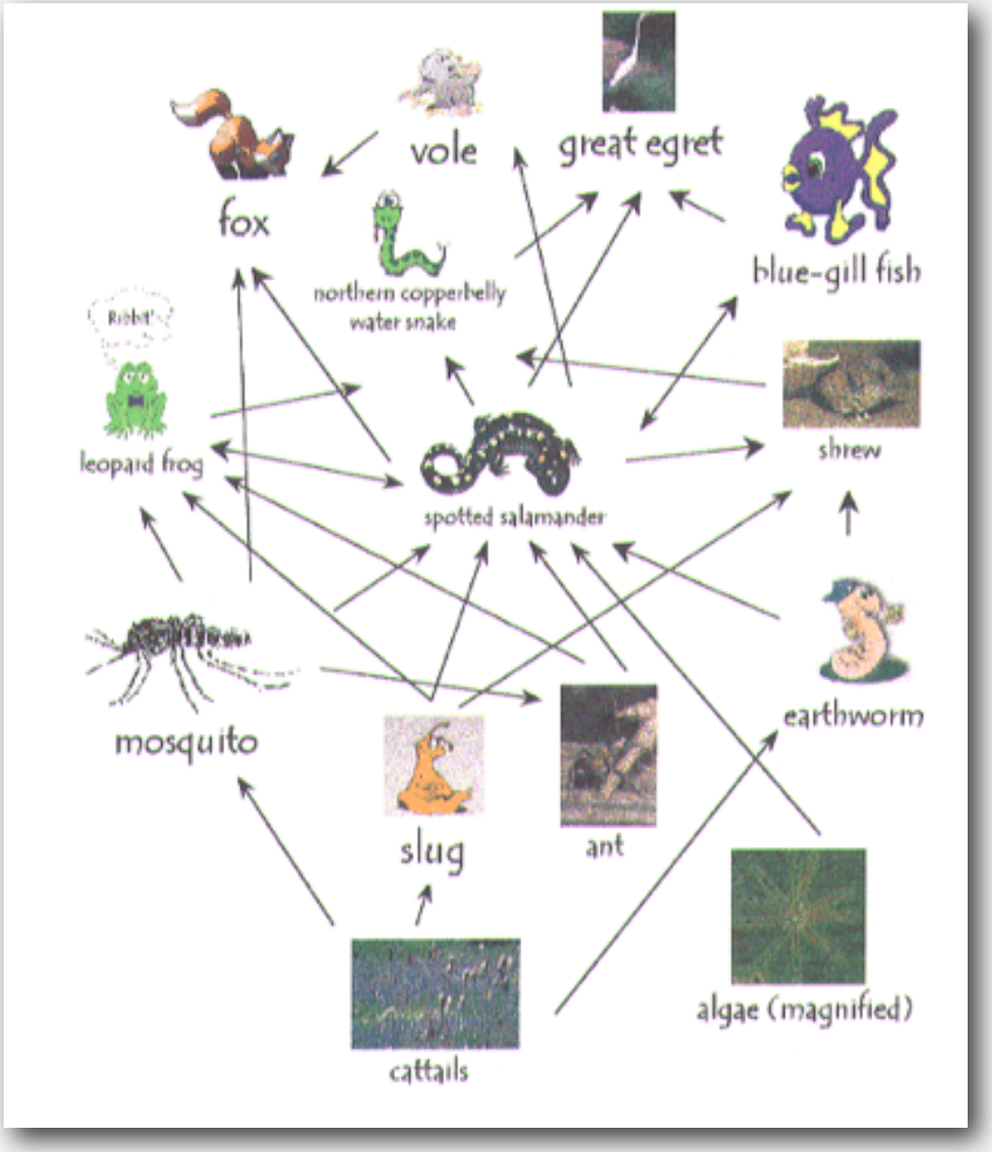
	0	1	2	3	4	5	6	7	8	9	10	11	12
scc[]	1	0	1	1	1	1	3	4	4	2	2	2	2

```
public int stronglyConnected(int v, int w)
{  return scc[v] == scc[w];  }
```

constant-time client strong-connectivity query

Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.



http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

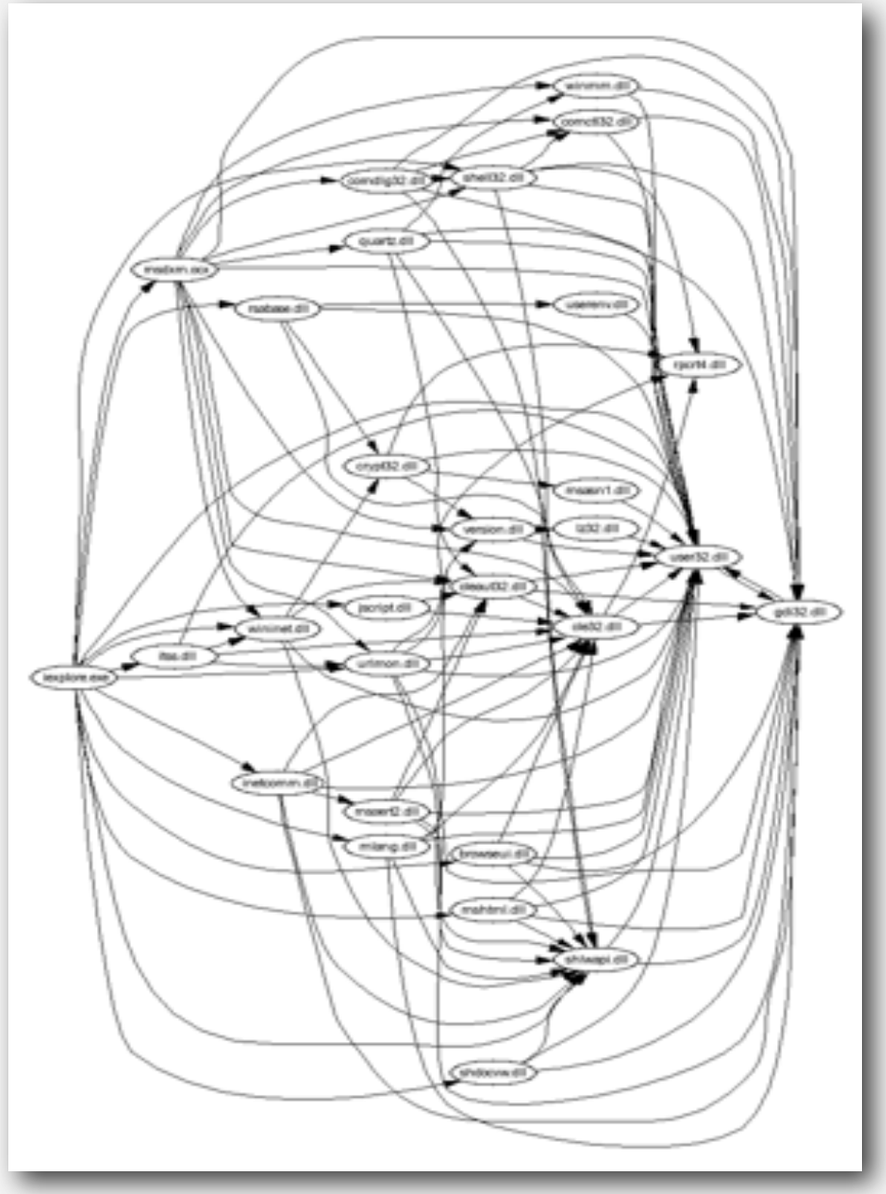
Strong component. Subset of species with common energy flow.

Strong component application: software modules

Software module dependency graph.

- Vertex = software module.
- Edge: from module to dependency.

Firefox



Internet Explorer

Strong component. Subset of mutually interacting modules.

Approach 1. Package strong components together.

Approach 2. Use to improve design!

Strong components algorithms: brief history

1960s: Core OR problem.

- Widely studied; some practical algorithms.
- Complexity not understood.

1972: linear-time DFS algorithm (Tarjan).

- Classic algorithm.
- Level of difficulty: Algs4++.
- Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).

- Forgot notes for lecture; developed algorithm in order to teach it!
- Later found in Russian scientific literature (1972).

1990s: more easy linear-time algorithms.

- Gabow: fixed old OR algorithm.
- Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.

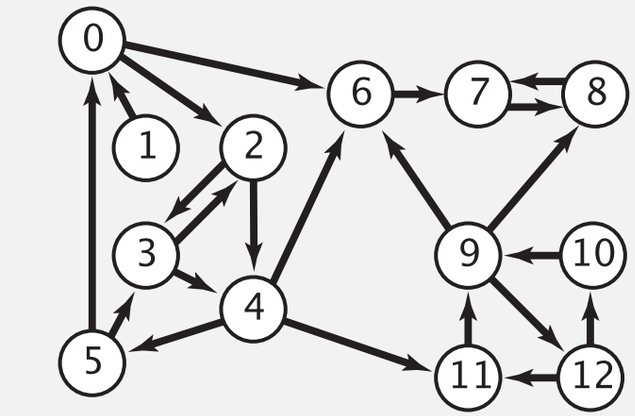
Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

- Run DFS on G^R to compute reverse postorder.
- Run DFS on G, considering vertices in order given by first DFS.

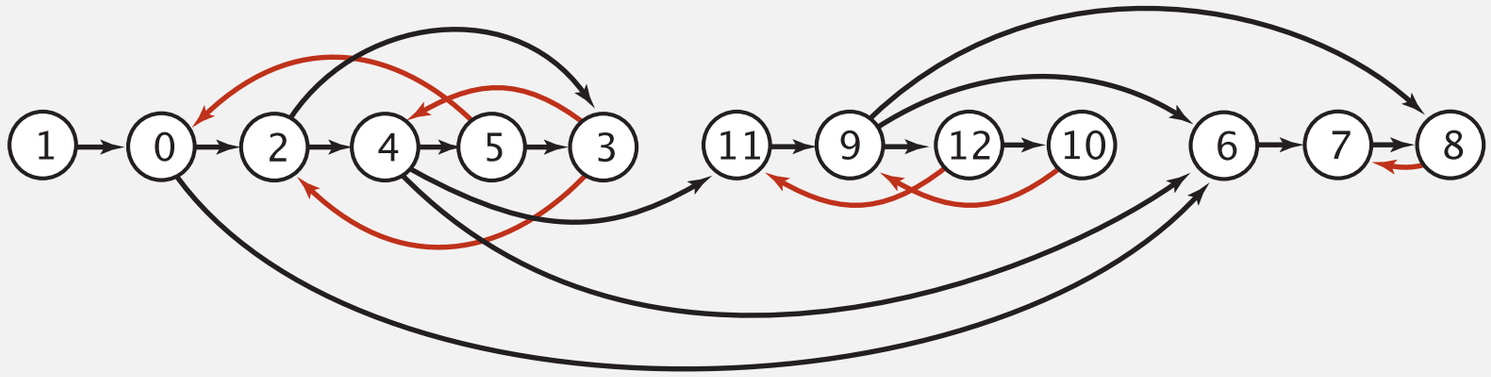
DFS in reverse digraph (`ReversePost`)

G^R



check unmarked vertices in the order

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



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

reverse postorder

```
dfs(0)
  dfs(6)
    dfs(7)
      dfs(8)
        check 7
      8 done
    7 done
  6 done
  dfs(2)
    dfs(4)
      dfs(11)
        dfs(9)
          dfs(12)
            check 11
      ...
```

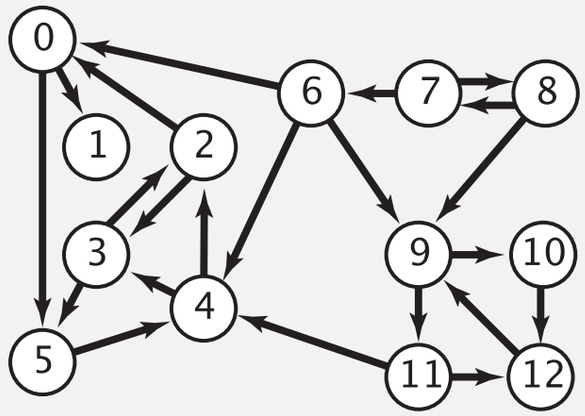
Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

- Run DFS on G^R to compute reverse postorder.
- Run DFS on G, considering vertices in order given by first DFS.

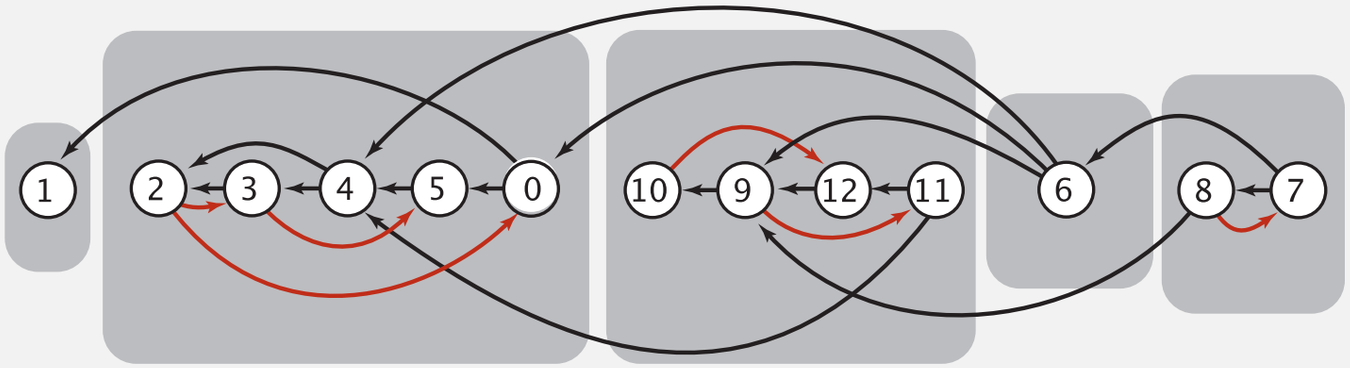
DFS in original digraph

G



check unmarked vertices in the order

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




```
dfs(1)
1 done
```

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

```
dfs(11)
  check 4
  dfs(12)
    dfs(9)
      check 11
      dfs(10)
        check 12
      10 done
    9 done
  12 done
11 done
check 9
check 12
check 10
```

```
dfs(6)
  check 9
  check 4
  check 0
6 done
```

```
dfs(7)
  check 6
  dfs(8)
    check 7
    check 9
  8 done
7 done
check 8
```

Proposition. Second DFS gives strong components. (!!)

Kosaraju's algorithm: intuition?

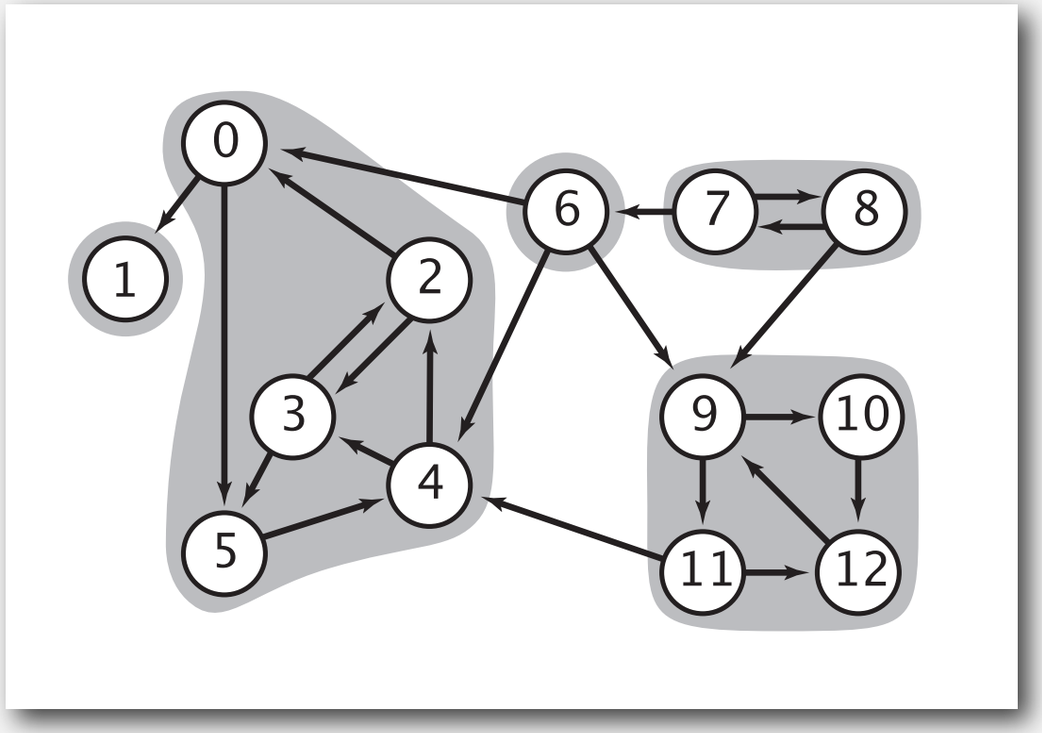
Reverse graph. Strong components in G are same as in G^R.

Kernel DAG. Contract each strong component into a single vertex.

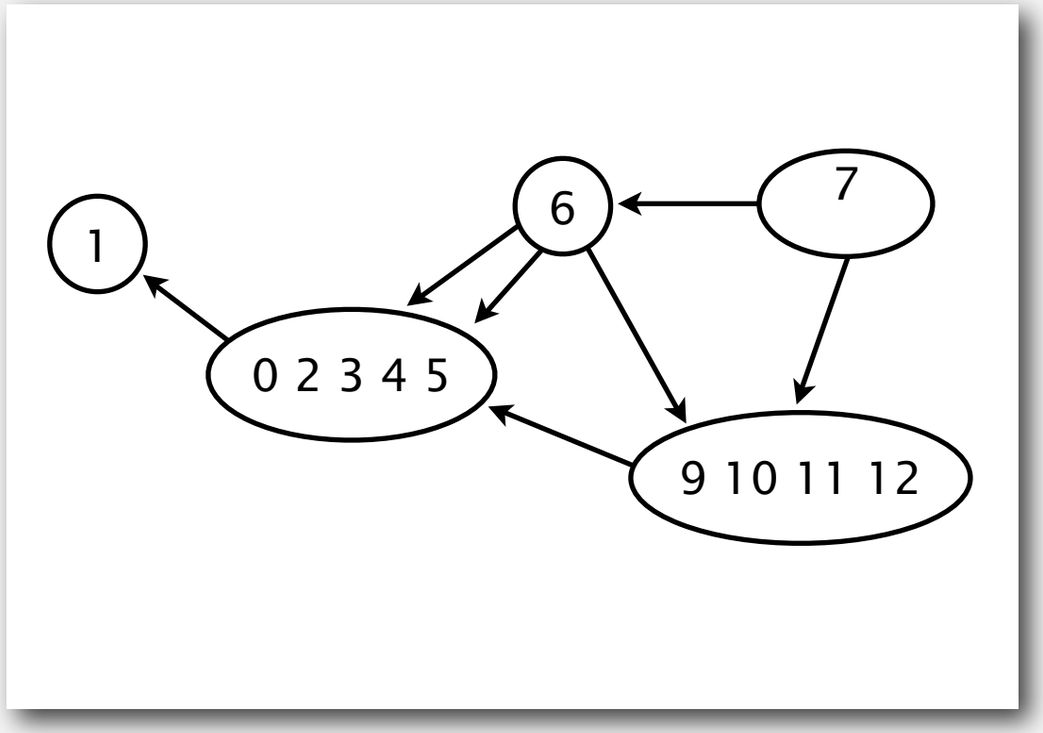
Idea.

how to compute?

- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.

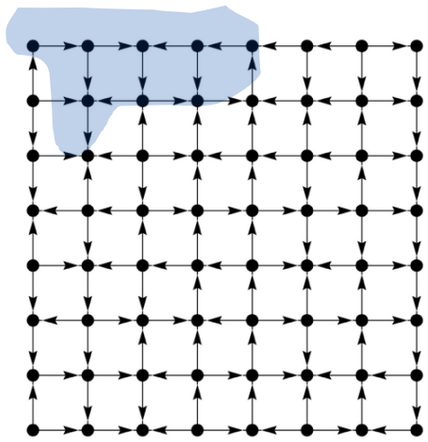
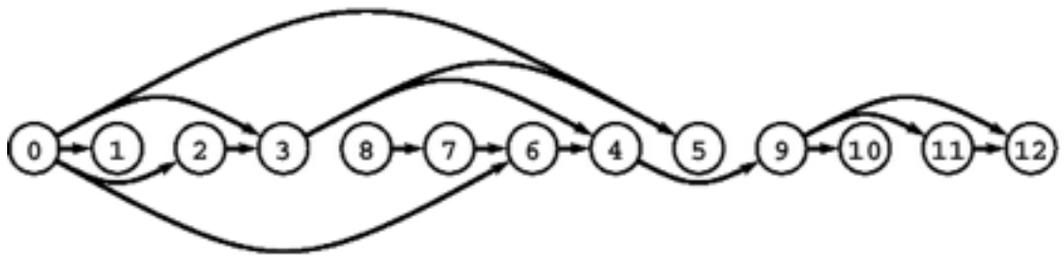
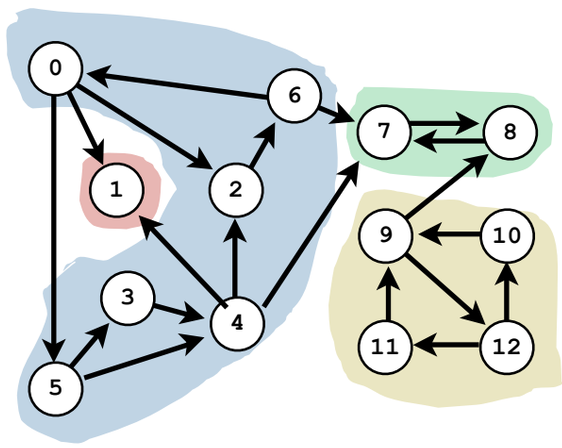


digraph G and its strong components

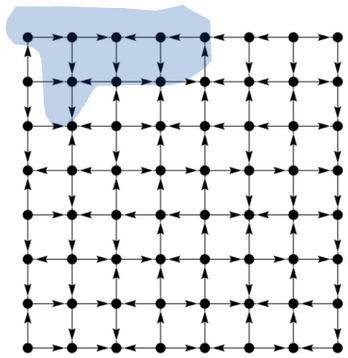
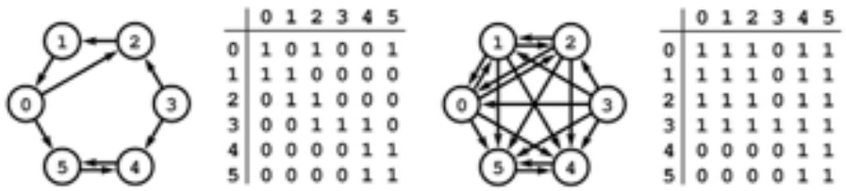
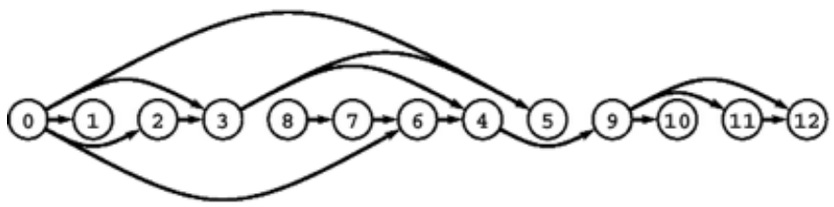
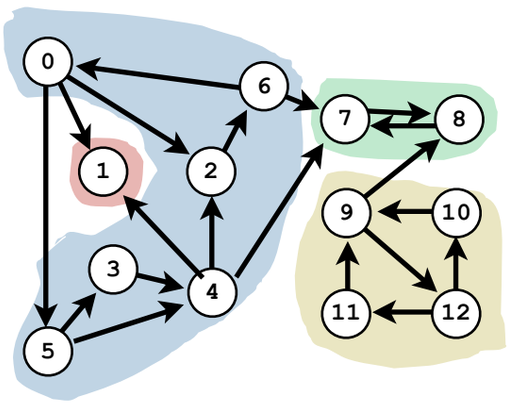


kernel DAG of G

Digraph-processing summary: some algorithms covered

single-source reachability		DFS
topological sort (DAG)		DFS
strong components		Kosaraju DFS (twice)

Digraph-processing summary:continued

single-source reachability		DFS
transitive closure		DFS (from each vertex)
topological sort (DAG)		DFS
strong components		Kosaraju DFS (twice)

Minimum Spanning Trees

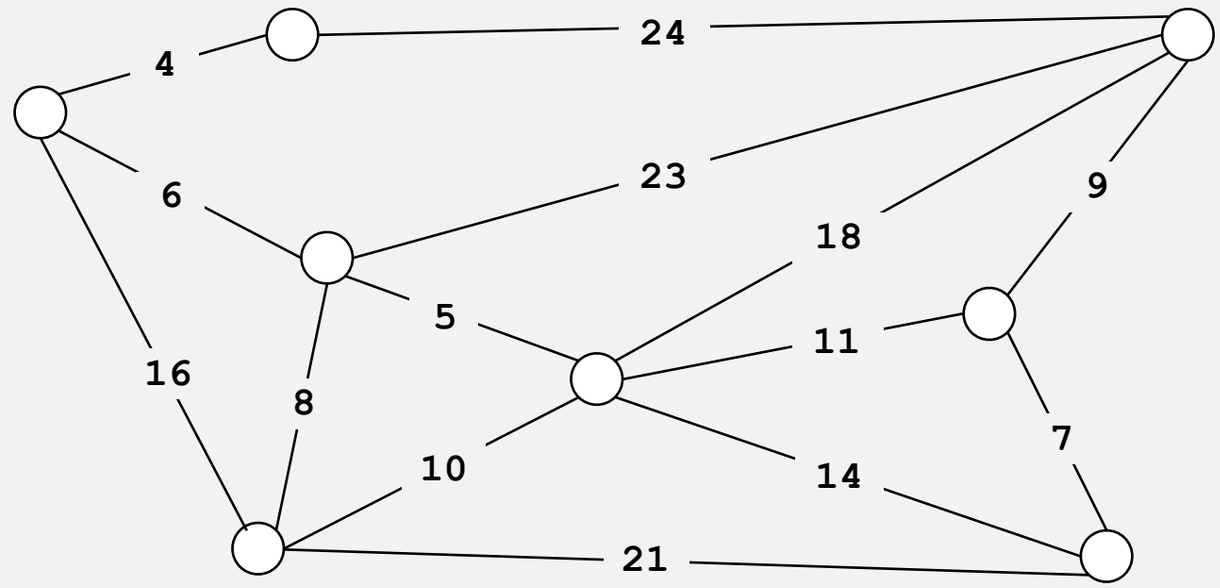
- edge-weighted graph API
- greedy algorithm
- Kruskal's algorithm
- Prim's algorithm
- advanced topics

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



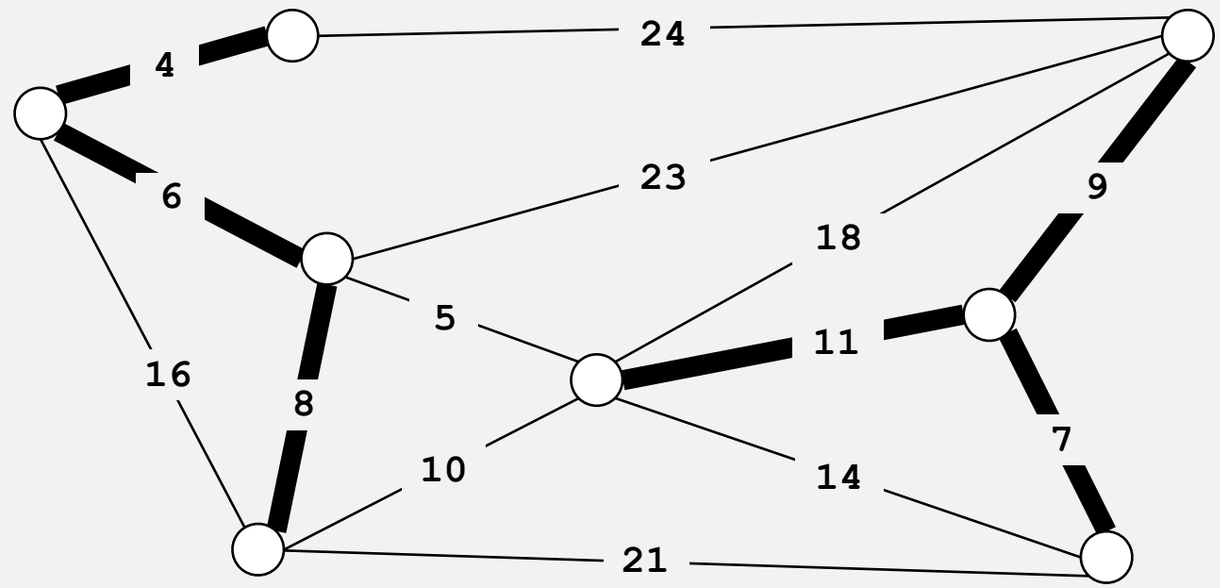
graph G

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



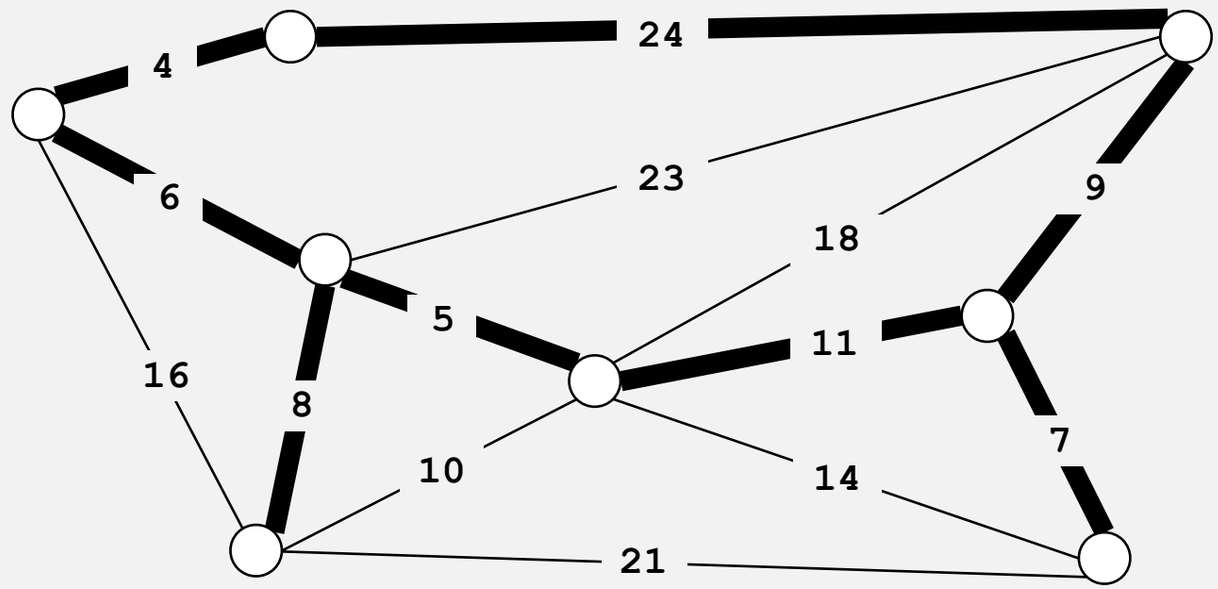
not connected

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



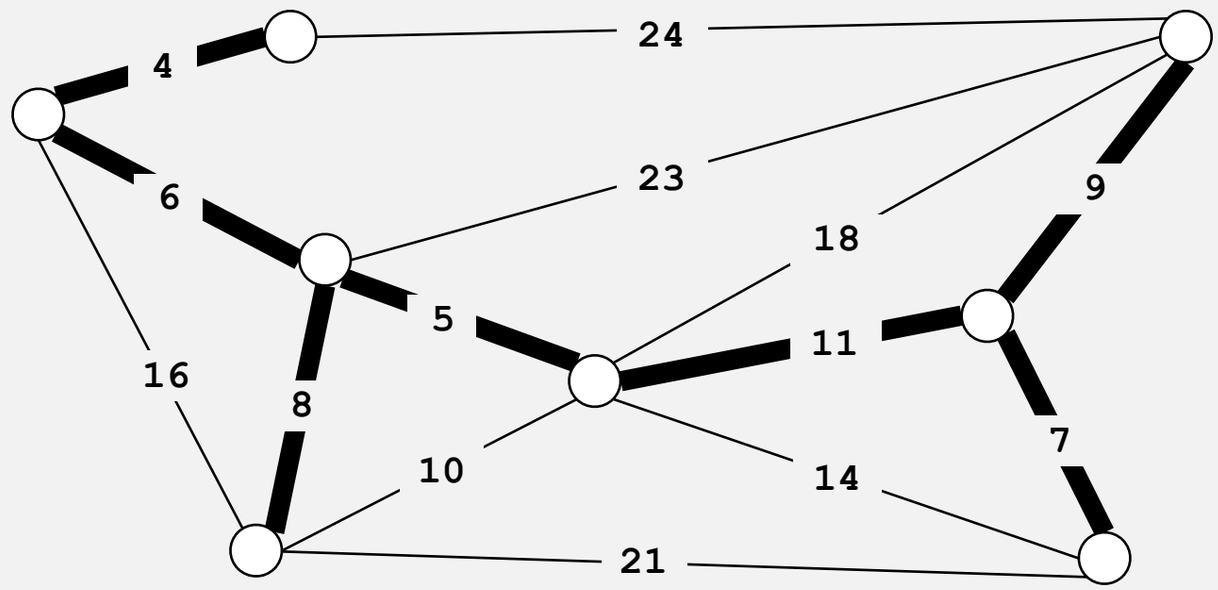
not acyclic

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.

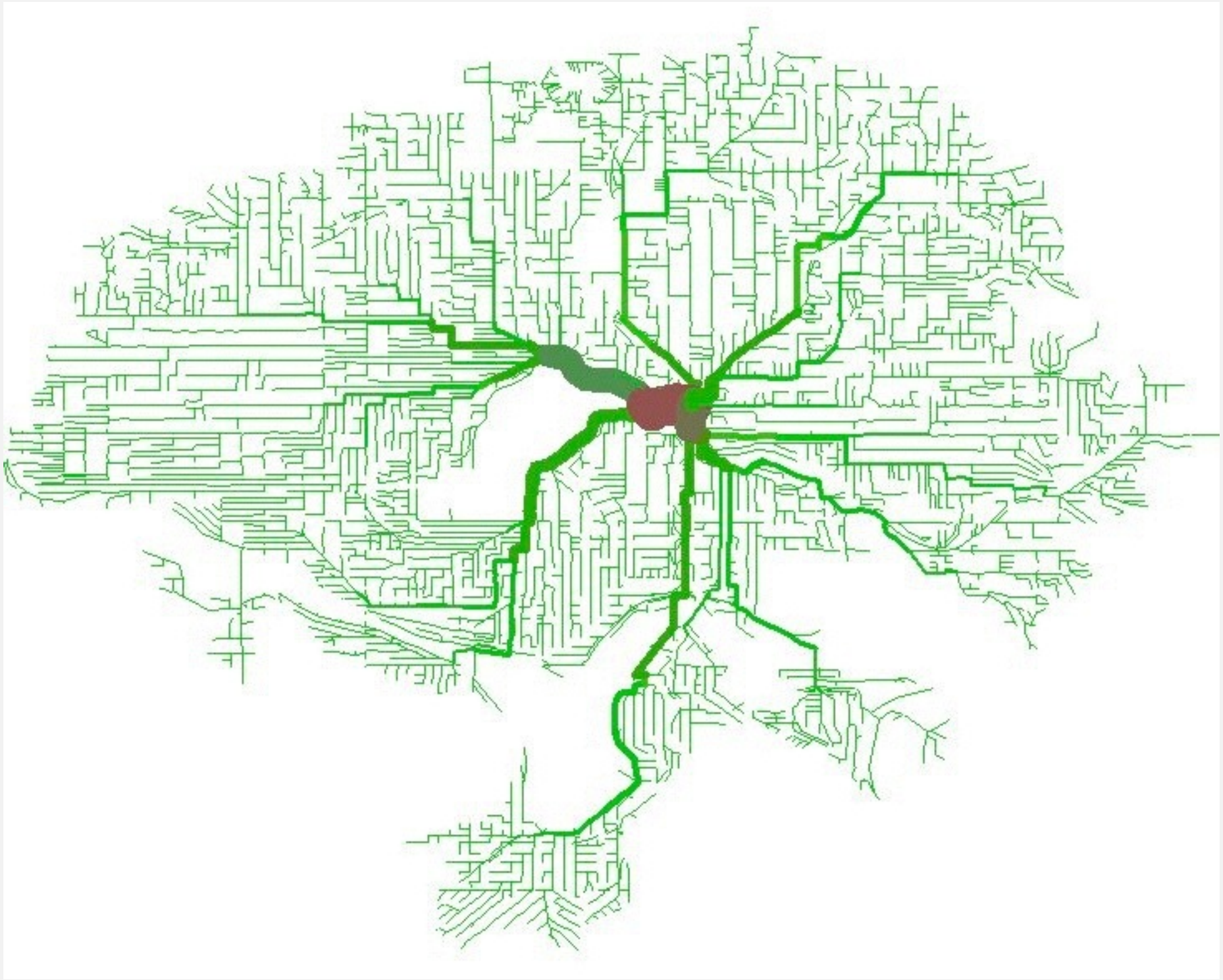


spanning tree T: cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Brute force. Try all spanning trees?

Network design

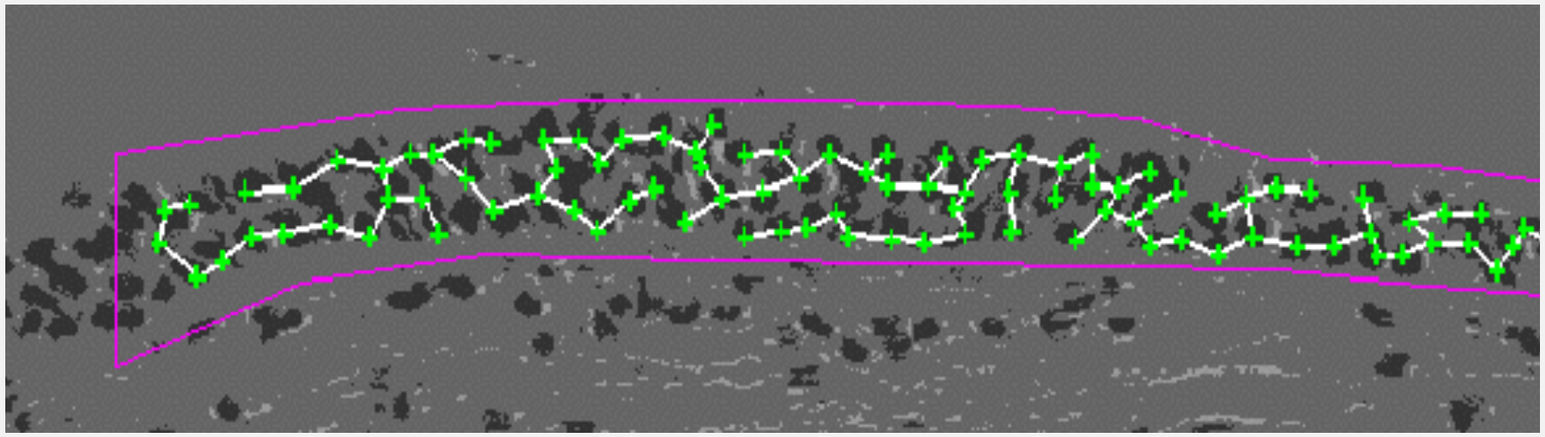
MST of bicycle routes in North Seattle

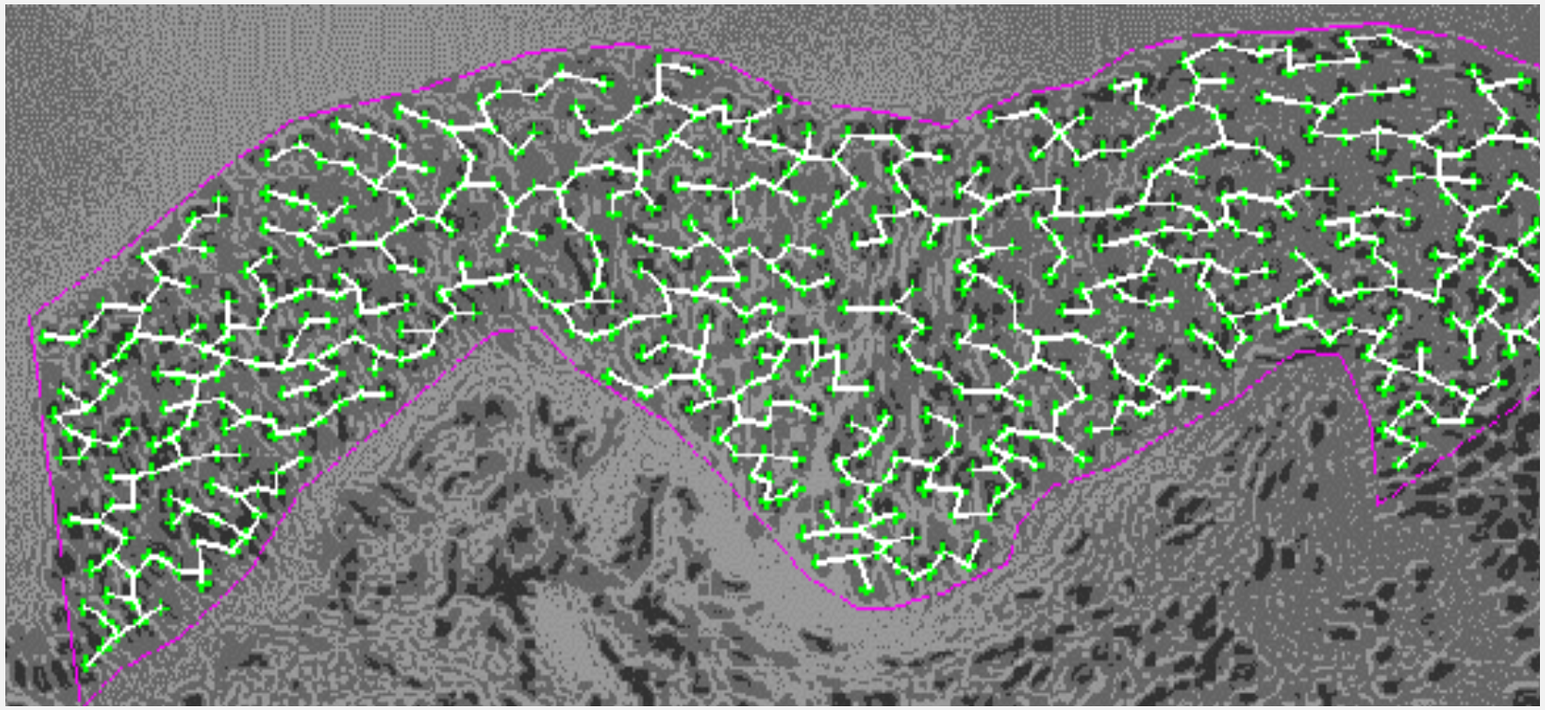


http://www.flickr.com/photos/ewedistrict/21980840

Medical image processing

MST describes arrangement of nuclei in the epithelium for cancer research





http://www.bccrc.ca/ci/ta01_archlevel.html

Applications

MST is fundamental problem with diverse applications.

- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, cable, computer, road).

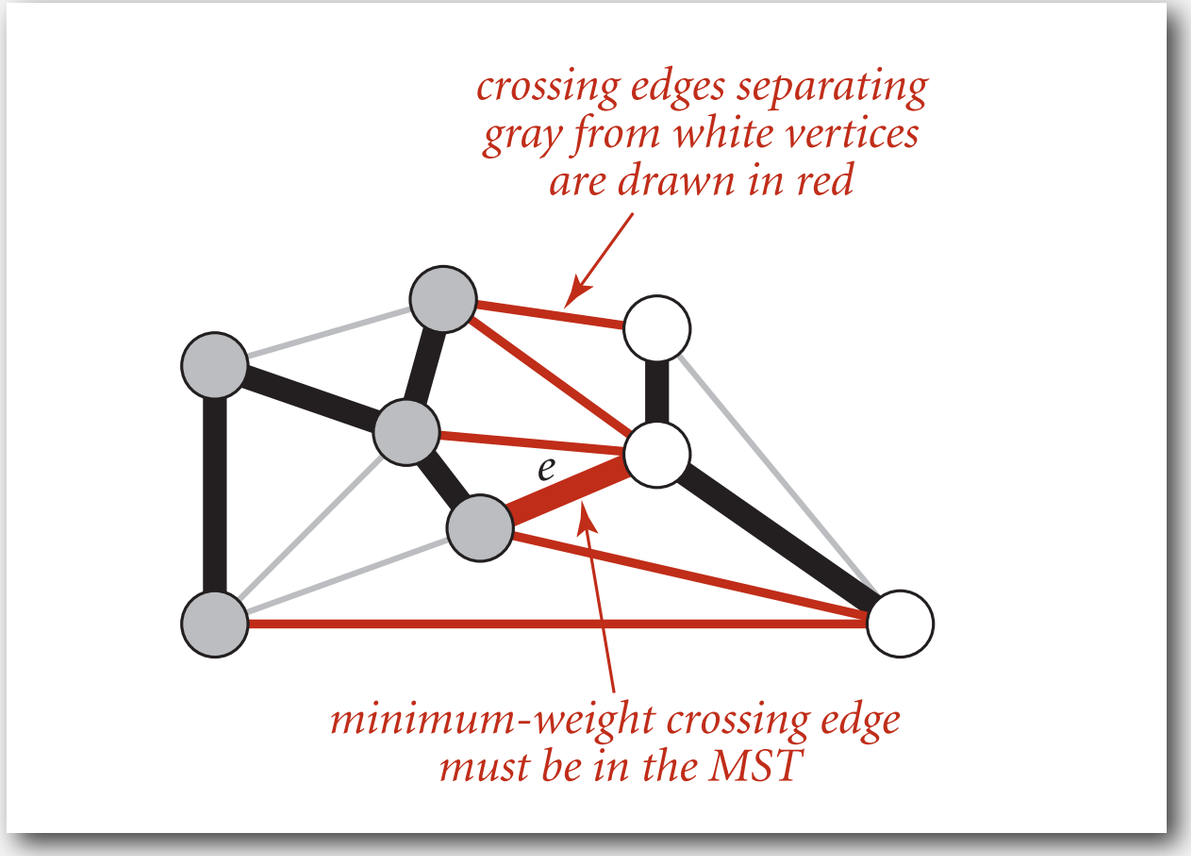
`http://www.ics.uci.edu/~eppstein/gina/mst.html`

Cut property

Simplifying assumptions. Edge weights are distinct; graph is connected.

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.
A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.



Cut property: correctness proof

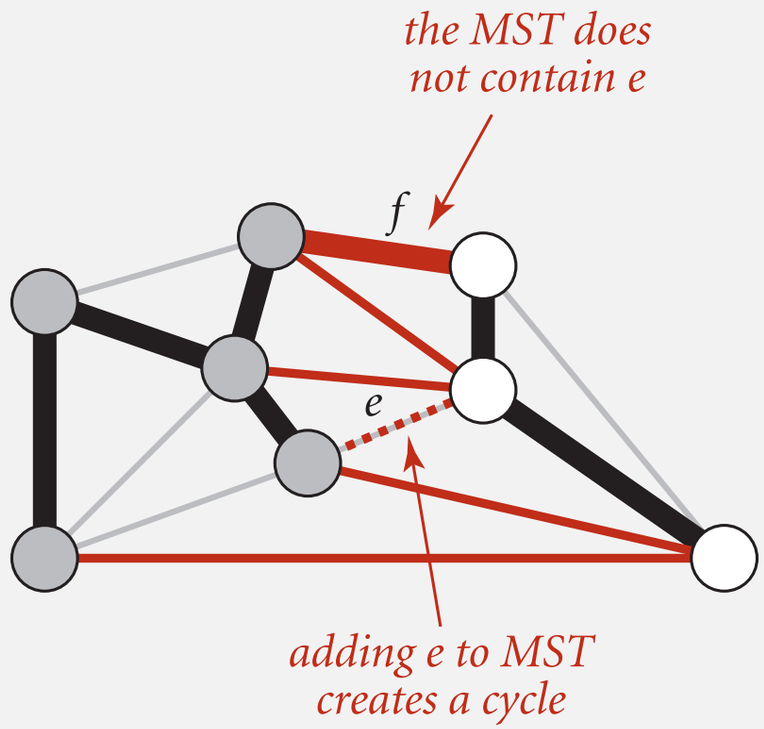
Simplifying assumptions. Edge weights are distinct; graph is connected.

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.
A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.

Pf. Let e be the min-weight crossing edge in cut.

- Suppose e is not in the MST.
- Adding e to the MST creates a cycle.
- Some other edge f in cycle must be a crossing edge.
- Removing f and adding e is also a spanning tree.
- Since weight of e is less than the weight of f, that spanning tree is lower weight.
- Contradiction. ▪



Greedy MST algorithm demo

Proposition. The following algorithm computes the MST:

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Continue until $V - 1$ edges are colored black.

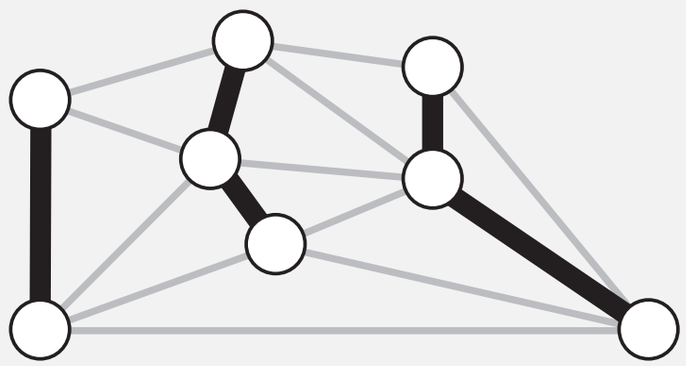
Greedy MST algorithm: correctness proof

Proposition. The following algorithm computes the MST:

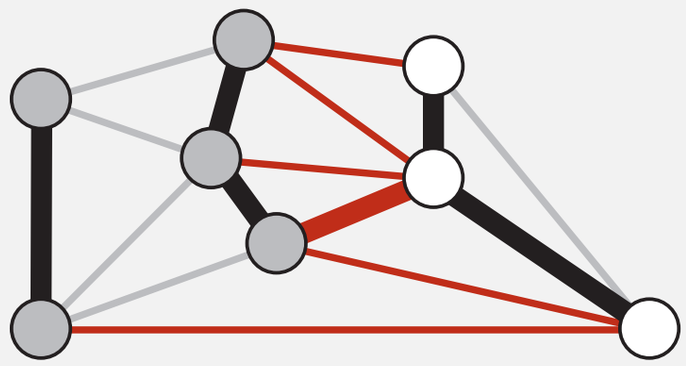
- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Continue until $V - 1$ edges are colored black.

Pf.

- Any edge colored black is in the MST (via cut property).
- If fewer than $V - 1$ black edges, there exists a cut with no black crossing edges. (consider cut whose vertices are one connected component)



fewer than V-1 edges colored black



a cut with no black crossing edges

Greedy MST algorithm: efficient implementations

Proposition. The following algorithm computes the MST:

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Continue until $V - 1$ edges are colored black.

Efficient implementations. How to choose cut? How to find min-weight edge?

Ex 1. Kruskal's algorithm.

Ex 2. Prim's algorithm.

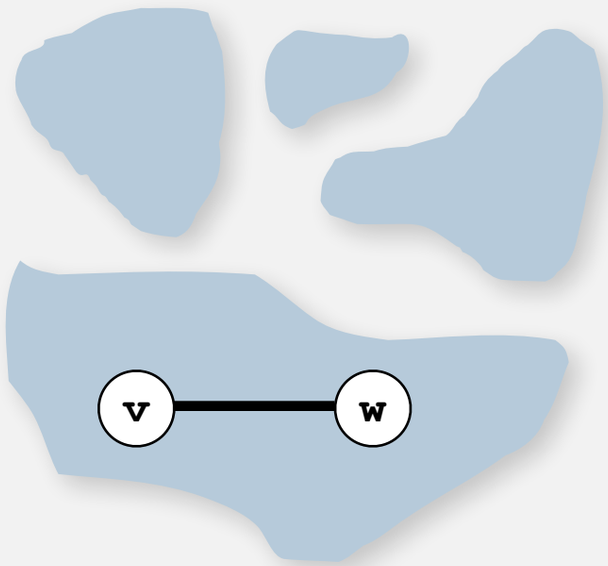
- edge-weighted graph API
- greedy algorithm
- **Kruskal's algorithm**
- Prim's algorithm
- advanced topics

Kruskal's algorithm: implementation challenge

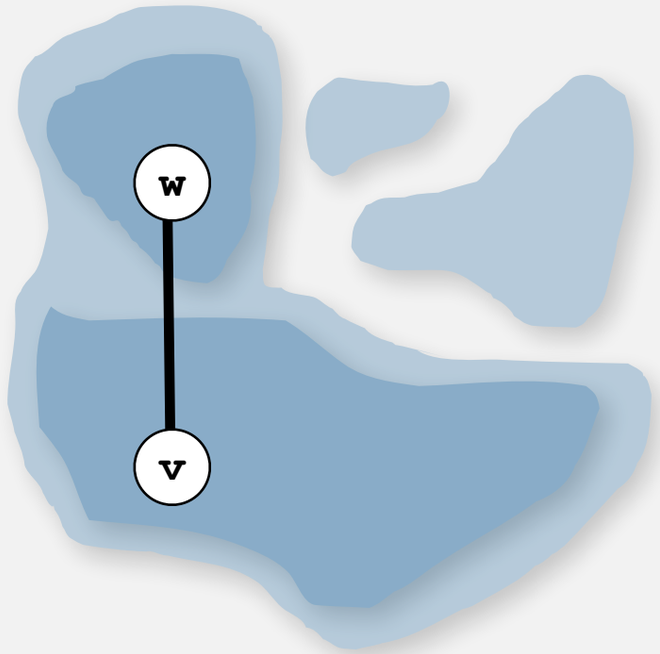
Challenge. Would adding edge v–w to tree T create a cycle? If not, add it.

Efficient solution. Use the union-find data structure.

- Maintain a set for each connected component in T.
- If v and w are in same set, then adding v–w would create a cycle.
- To add v–w to T, merge sets containing v and w.

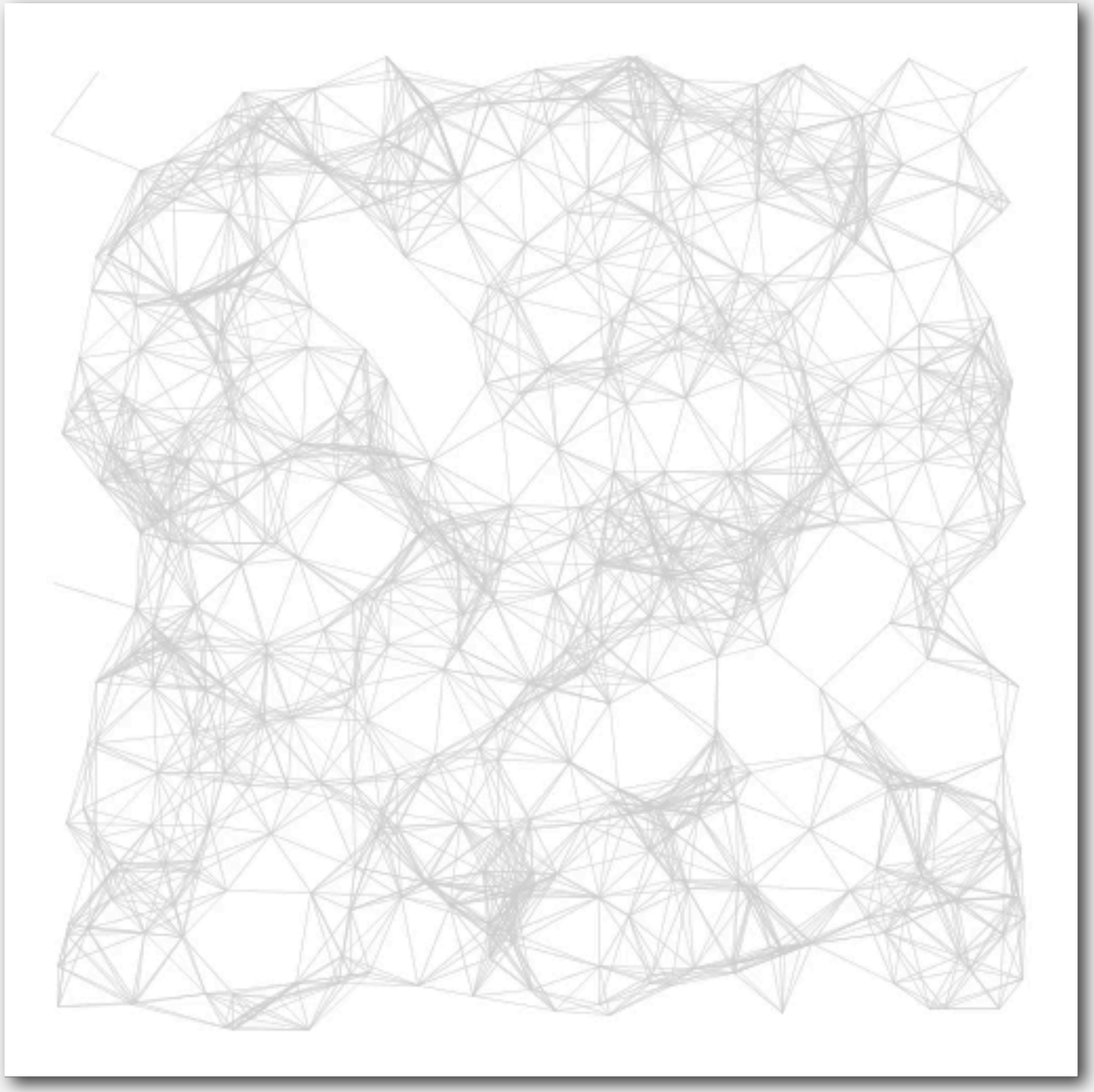


Case 1: adding v–w creates a cycle



Case 2: add v–w to T and merge sets containing v and w

Kruskal's algorithm: visualization



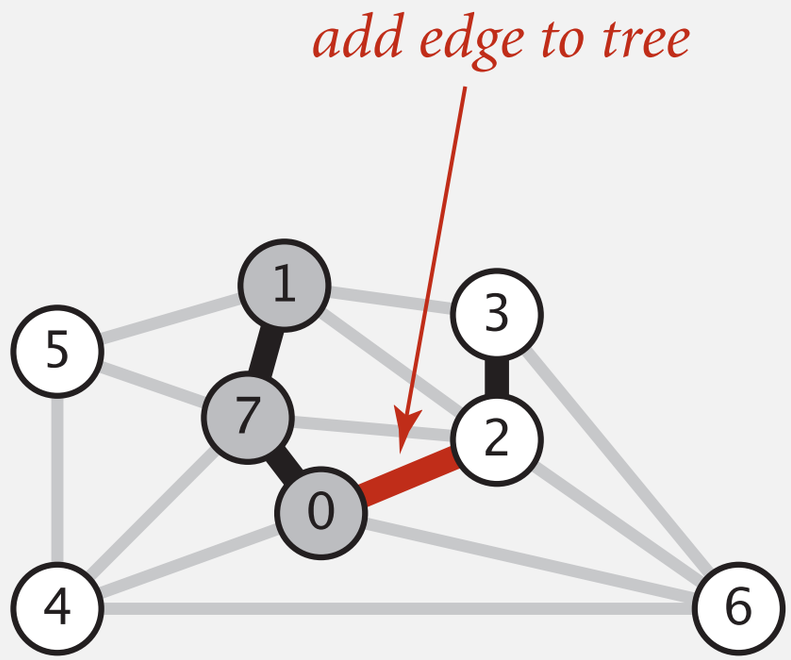
Kruskal's algorithm: correctness proof

Proposition. Kruskal's algorithm computes the MST.

Pf. Kruskal's algorithm is a special case of the greedy MST algorithm.

- Suppose Kruskal's algorithm colors the edge $e = v$–w black.
- Cut = set of vertices connected to v in tree T.
- No crossing edge is black.
- No crossing edge has lower weight. Why?

  - Because we are choosing edges in ascending order of weight!

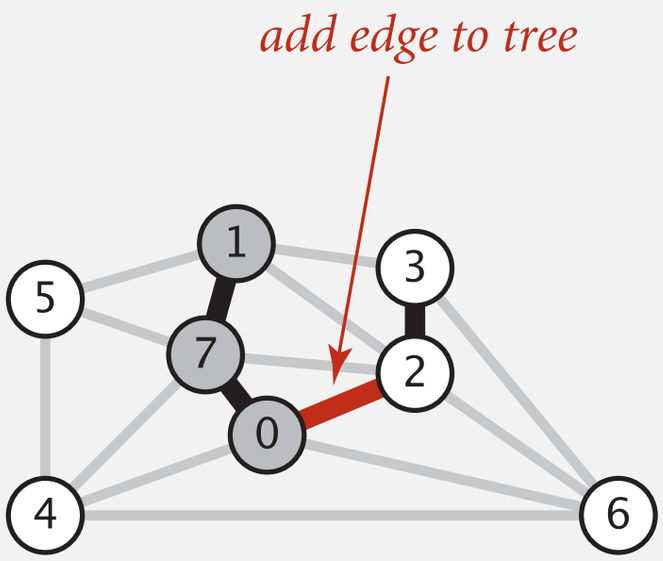


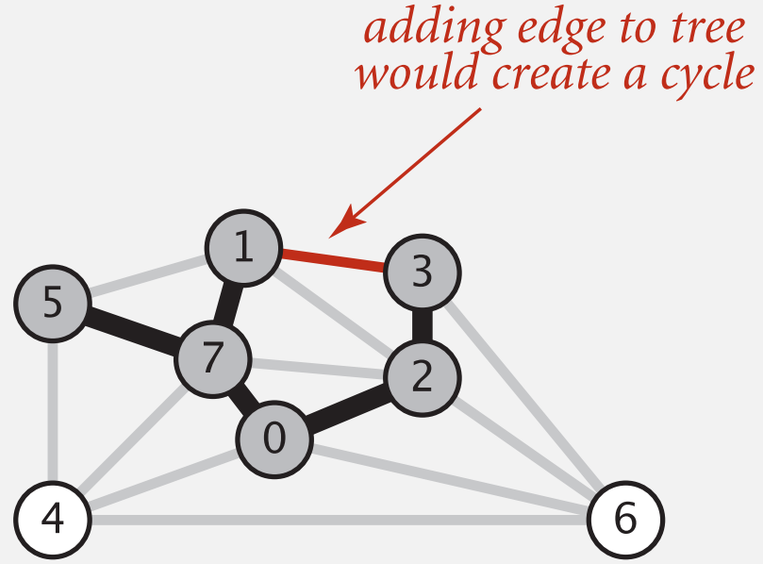
Kruskal's algorithm: implementation challenge

Challenge. Would adding edge v–w to tree T create a cycle? If not, add it.

How difficult?

- $E + V$
- V ← run DFS from v, check if w is reachable (T has at most V – 1 edges)
- $\log V$
- $\log^* V$ ← use the union-find data structure !
- 1





Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Pf.

operation	frequency	time per op
build pq	1	E
delete-min	E	log E
union	V	log* V
connected	E	log* V

† amortized bound using weighted quick union with path compression

recall: log* V ≤ 5 in this universe

Remark. If edges are already sorted, order of growth is $E \log^* V$.

- edge-weighted graph API
- greedy algorithm
- Kruskal's algorithm
- **Prim's algorithm**
- advanced topics

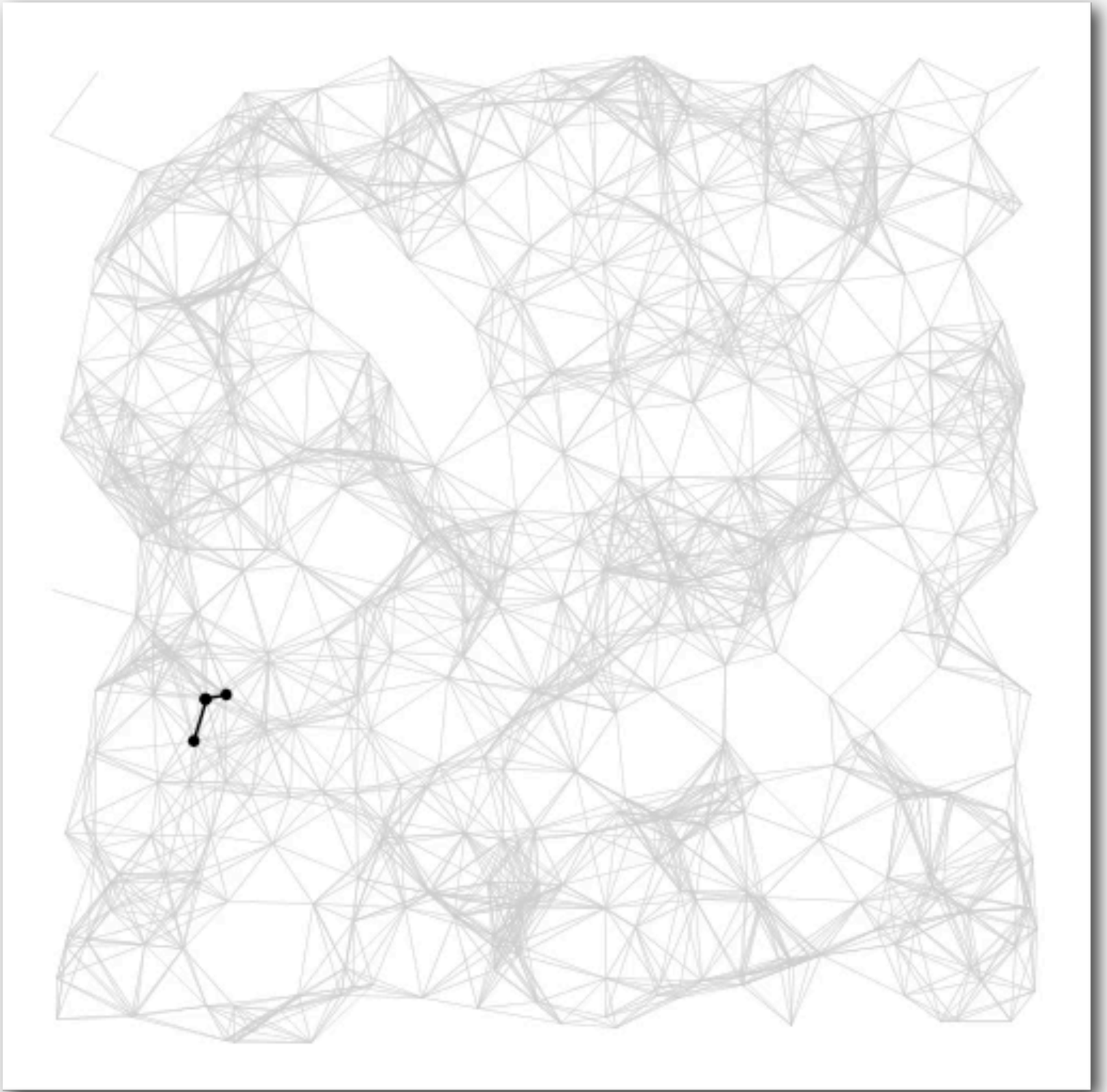
Prim's algorithm demo

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

Start with vertex 0 and greedily grow tree T. At each step,

add to T the min weight edge with exactly one endpoint in T.

Prim's algorithm: visualization



Prim's algorithm: implementation challenge

Challenge. Find the min weight edge with exactly one endpoint in T.

How difficult?

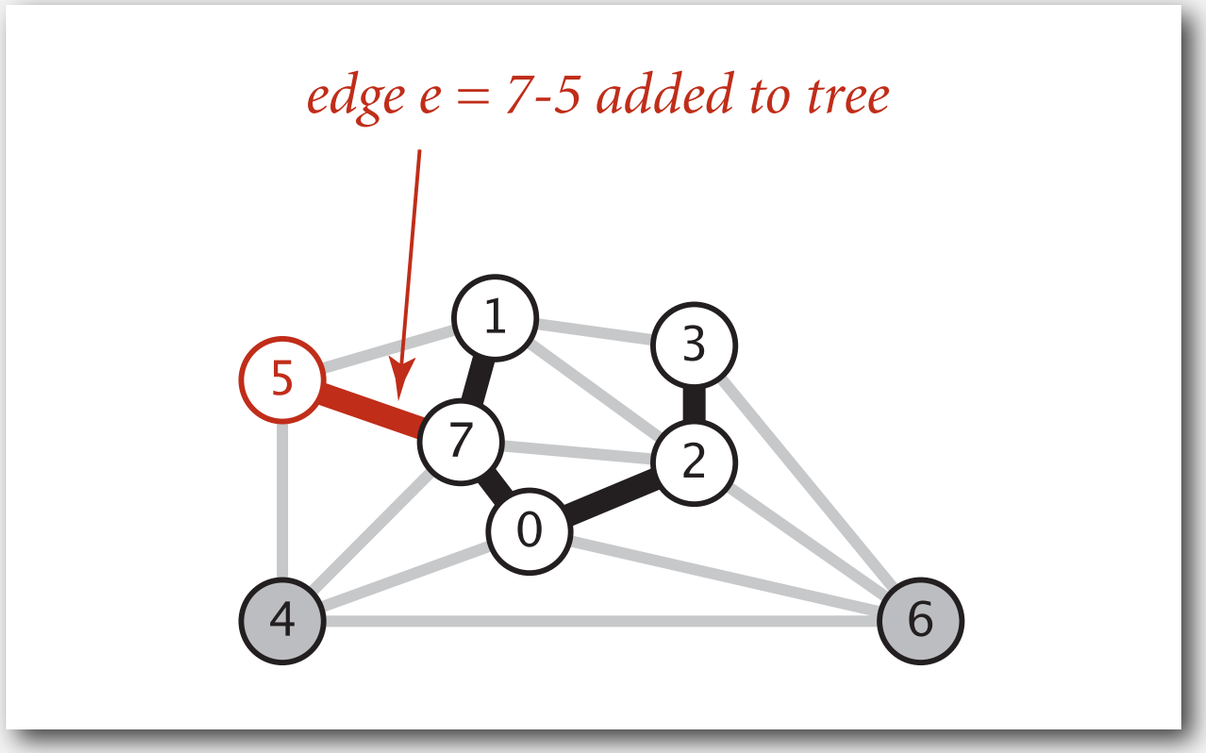
- E ← try all edges
- V
- $\log E$ ← use a priority queue !
- $\log^* E$
- 1

Prim's algorithm: proof of correctness

Proposition. Prim's algorithm computes the MST.

Pf. Prim's algorithm is a special case of the greedy MST algorithm.

- Suppose edge e = min weight edge connecting a vertex on the tree to a vertex not on the tree.
- Cut = set of vertices connected on tree.
- No crossing edge is black.
- No crossing edge has lower weight.



Prim's algorithm demo: lazy implementation

Prim's algorithm: lazy implementation

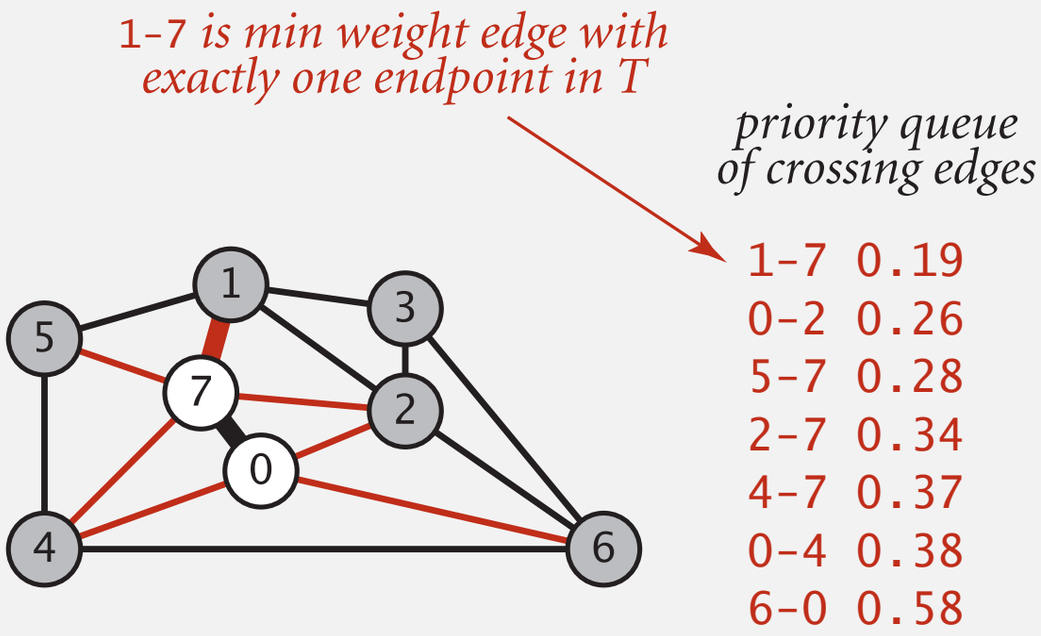
Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge $e = v$–w to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise, let v be vertex not in T:

  - add to PQ any edge incident to v (assuming other endpoint not in T)

  - add v to T



Lazy Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

Pf.

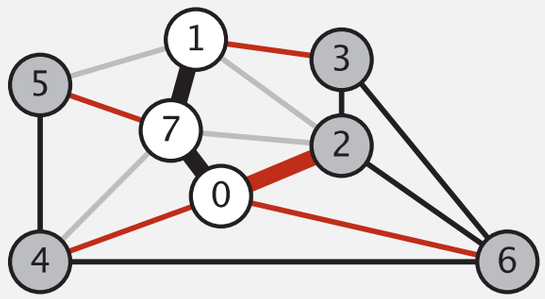
operation	frequency	binary heap
delete min	E	log E
insert	E	log E

Prim's algorithm: eager implementation

Challenge. Find min weight edge with exactly one endpoint in T.

Eager solution. Maintain a PQ of vertices connected by an edge to T, where priority of vertex v = weight of shortest edge connecting v to T.

- Delete min vertex v and add its associated edge $e = v$–w to T.
- Update PQ by considering all edges $e = v$–x incident to v
  - ignore if x is already in T
  - add x to PQ if not already on it
  - decrease priority of x if v–x becomes shortest edge connecting x to T




```
0
1 1-7 0.19
2 0-2 0.26   <-- red: on PQ
3 1-3 0.29
4 0-4 0.38
5 5-7 0.28
6 6-0 0.58
7 0-7 0.16
```

black: on MST

Prim's algorithm: running time

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
array	1	V	1	V
binary heap	log V	log V	log V	E log V
d-way heap (Johnson 1975)	d log	d log	log	E log
Fibonacci heap (Fredman-Tarjan 1984)	1	log V	1	E + V log V

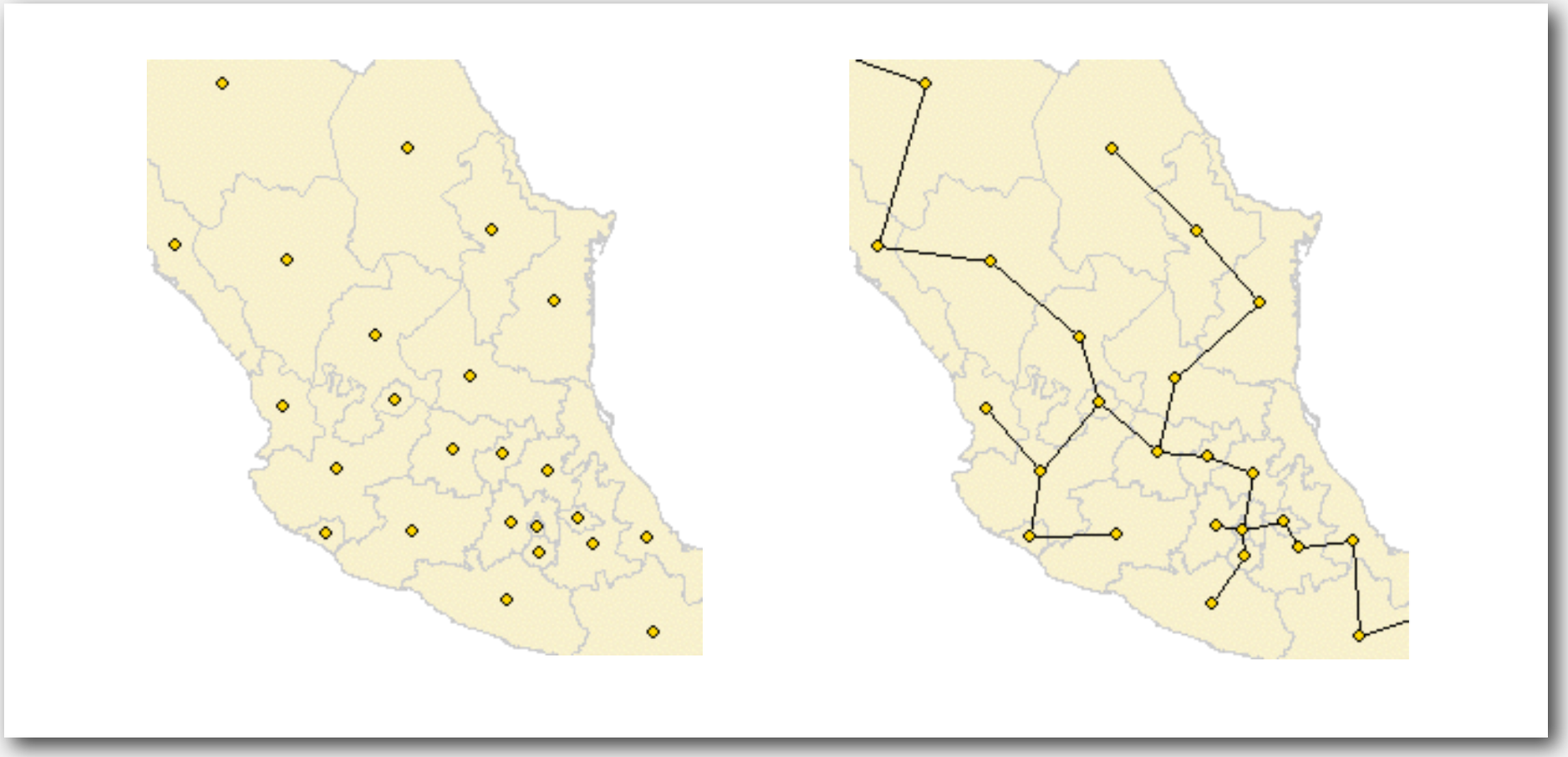
† amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Euclidean MST

Given N points in the plane, find MST connecting them, where the distances between point pairs are their Euclidean distances.



Brute force. Compute ~ $N^2 / 2$ distances and run Prim's algorithm.

Ingenuity. Exploit geometry and do it in ~ $c\, N \log N$.

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	E log log V	Yao
1976	E log log V	Cheriton-Tarjan
1984	E log* V, E + V log V	Fredman-Tarjan
1986	E log (log* V)	Gabow-Galil-Spencer-Tarjan
1997	E α(V) log α(V)	Chazelle
2000	E α(V)	Chazelle
2002	optimal	Pettie-Ramachandran
20xx	E	???

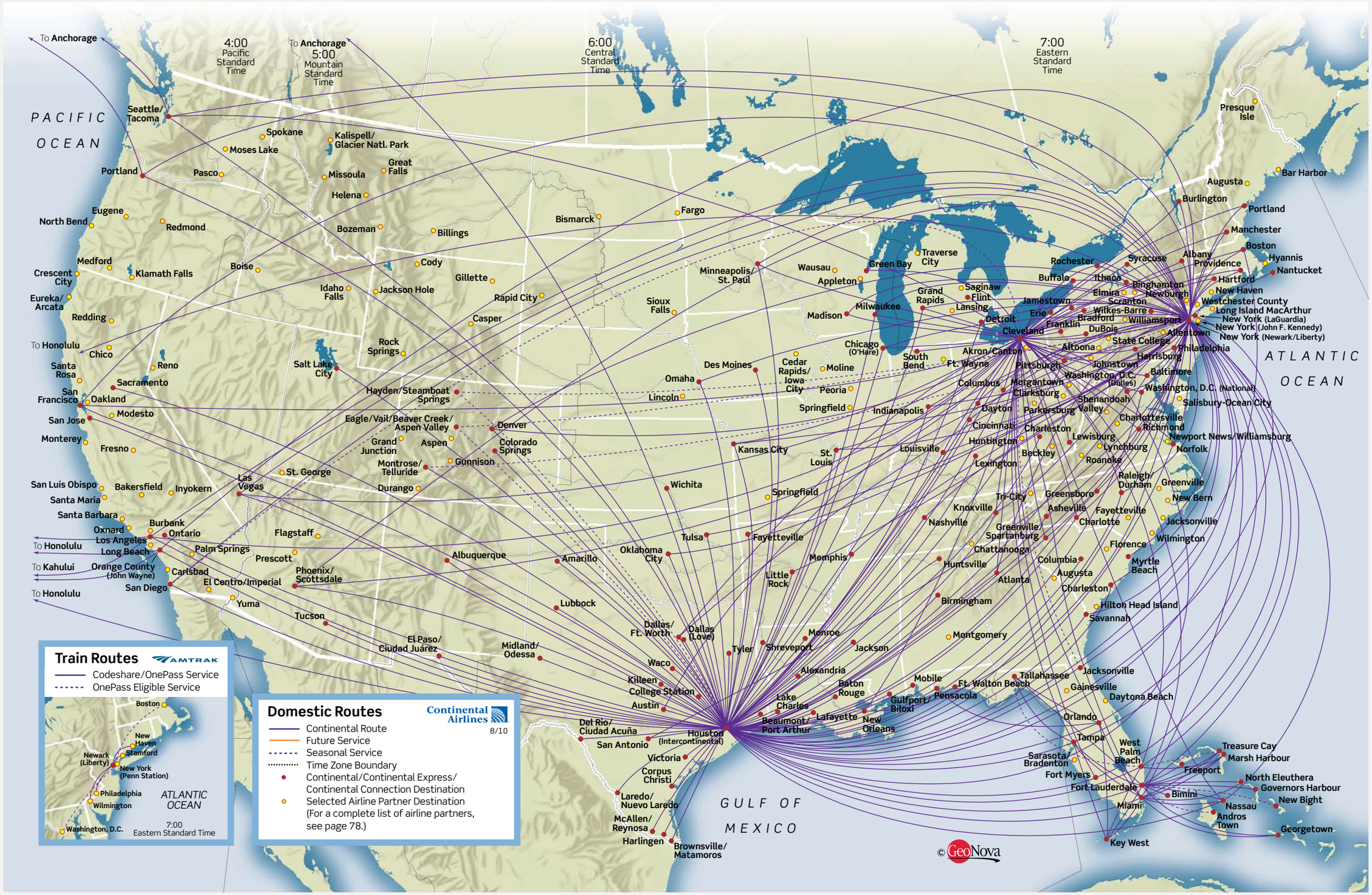
Remark. Linear-time randomized MST algorithm (Karger-Klein-Tarjan 1995).

- edge-weighted graph API
- greedy algorithm
- Kruskal's algorithm
- Prim's algorithm
- **application: clustering**

Shortest Paths

- edge-weighted digraph API
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights

Continental U.S. routes (August 2010)



http://www.continental.com/web/en-US/content/travel/routes

Shortest path applications

- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in advanced algorithms.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

http://en.wikipedia.org/wiki/Seam_carving

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest path variants

Which vertices?

- Source-sink: from one vertex to another.
- Single source: from one vertex to every other.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.
- Arbitrary weights.
- Euclidean weights.

Cycles?

- No directed cycles.
- No "negative cycles."

Simplifying assumption. There exists a shortest path from s to each vertex v.

- edge-weighted digraph API
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights

Data structures for single-source shortest paths

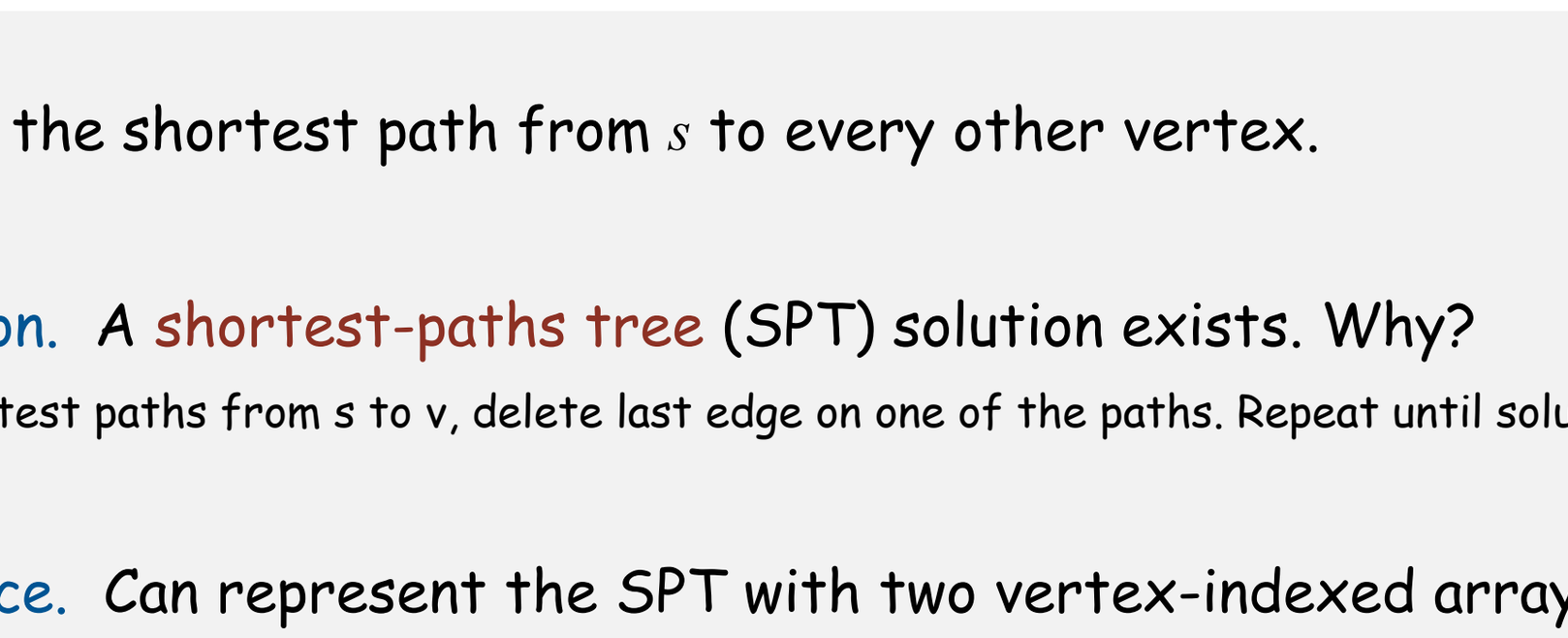
Goal. Find the shortest path from s to every other vertex.

Observation. A shortest-paths tree (SPT) solution exists. Why?

- If two shortest paths from s to v, delete last edge on one of the paths. Repeat until solution is a tree.

Consequence. Can represent the SPT with two vertex-indexed arrays:

- `distTo[v]` is length of shortest path from s to v.
- `edgeTo[v]` is last edge on shortest path from s to v.



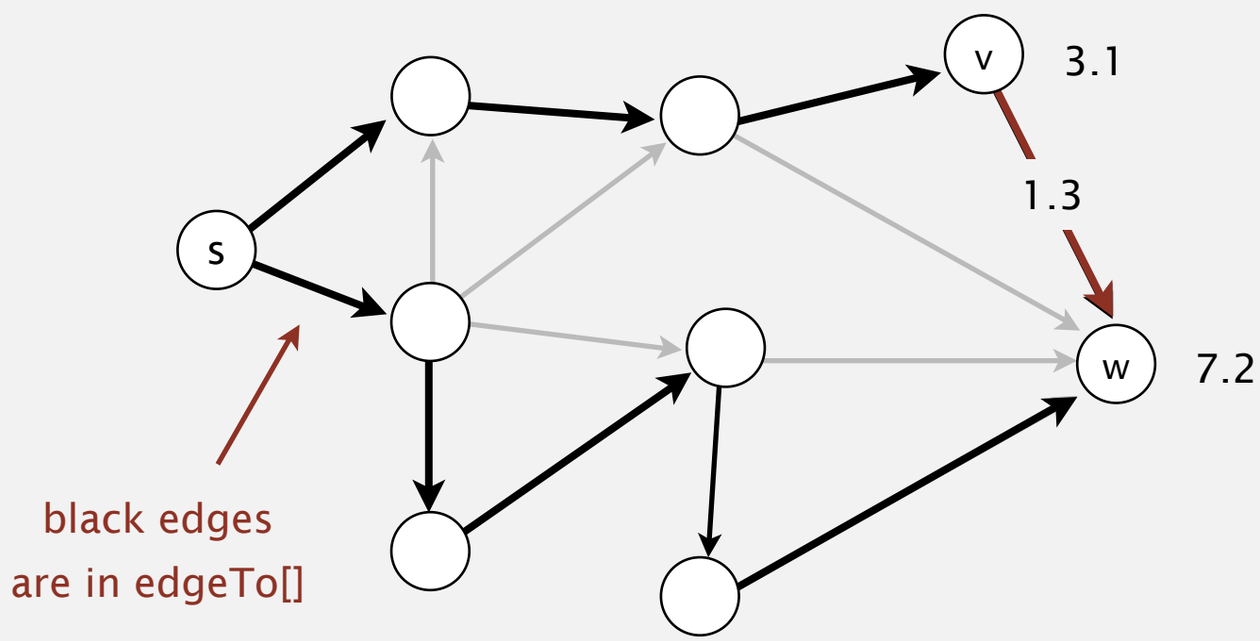
shortest-paths tree from 0

Edge relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest known path from s to v.
- `distTo[w]` is length of shortest known path from s to w.
- `edgeTo[w]` is last edge on shortest known path from s to w.
- If $e = v \rightarrow w$ gives shorter path to w through v, update `distTo[w]` and `edgeTo[w]`.

v→w successfully relaxes

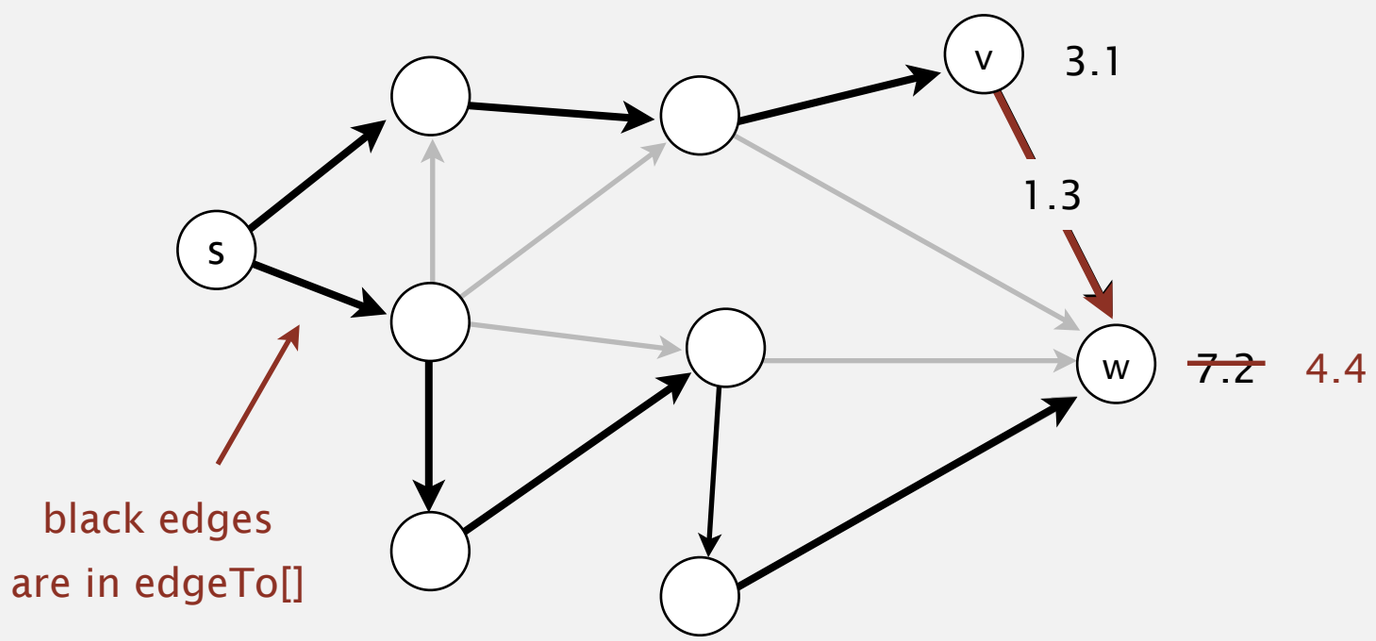


Edge relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest known path from s to v.
- `distTo[w]` is length of shortest known path from s to w.
- `edgeTo[w]` is last edge on shortest known path from s to w.
- If $e = v \rightarrow w$ gives shorter path to w through v, update `distTo[w]` and `edgeTo[w]`.

v→w successfully relaxes



Edge and vertex relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest known path from s to v.
- `distTo[w]` is length of shortest known path from s to w.
- `edgeTo[w]` is last edge on shortest known path from s to w.
- If $e = v \rightarrow w$ gives shorter path to w through v, update `distTo[w]` and `edgeTo[w]`.

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

vertex relaxation:

apply above to all edges associated with a vertex

Shortest-paths optimality conditions

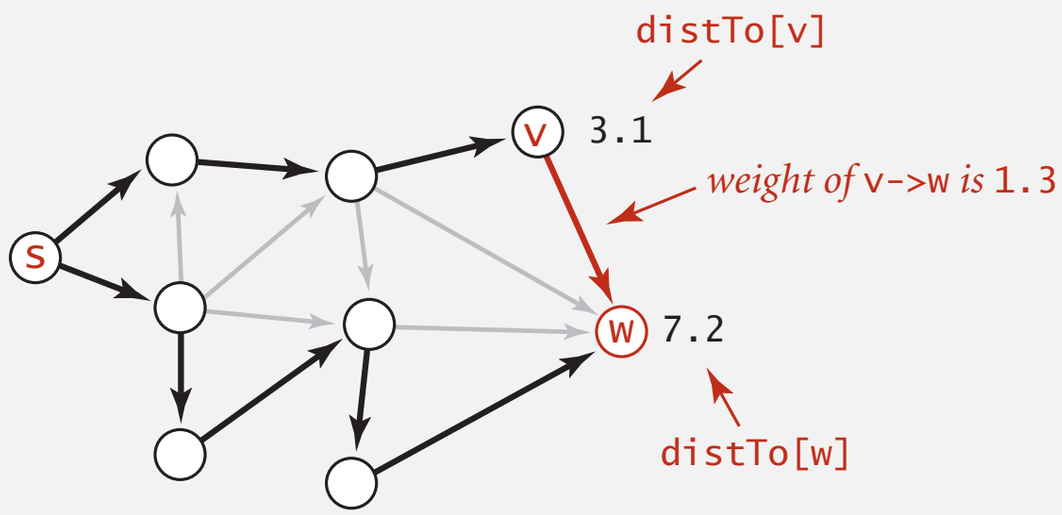
Proposition. Let G be an edge-weighted digraph.

Then `distTo[]` are the shortest path distances from s iff:

- For each vertex v, `distTo[v]` is the length of some path from s to v.
- For each edge $e = v \rightarrow w$, `distTo[w] ≤ distTo[v] + e.weight()`.

Pf. $\Leftarrow$ [necessary]

- Suppose that `distTo[w] > distTo[v] + e.weight()` for some edge $e = v \rightarrow w$.
- Then, e gives a path from s to w (through v) of length less than `distTo[w]`.



Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.

Then `distTo[]` are the shortest path distances from s iff:

- For each vertex v, `distTo[v]` is the length of some path from s to v.
- For each edge $e = v \rightarrow w$, `distTo[w] ≤ distTo[v] + e.weight()`.

Pf. $\Rightarrow$ [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = w$ is a shortest path from s to w.
- Then,

```
distTo[v      ≤  distTo[v      + e
distTo[v      ≤  distTo[v      + e
...
distTo[v      ≤  distTo[v      + e
```

e_i = i^{th} edge on shortest path from s to w

- Add inequalities; simplify; and substitute `distTo[v0] = distTo[s] = 0`:

$$\texttt{distTo[w]} = \texttt{distTo[}v_k\texttt{]} \le \texttt{e}_k\texttt{.weight()} + \texttt{e}_{k-1}\texttt{.weight()} + \ldots + \texttt{e}_1\texttt{.weight()}$$

weight of some path from s to w

weight of shortest path from s to w

- It cannot be smaller than the length of shortest path, Thus, `distTo[w]` is the weight of shortest path to w.

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Proposition. Generic algorithm computes SPT from s. ⟵ assuming SPT exists

Pf sketch.

- Throughout algorithm, `distTo[v]` is the length of a simple path from s to v and `edgeTo[v]` is last edge on path.
- Each successful relaxation decreases `distTo[v]` for some v.
- The entry `distTo[v]` can decrease at most a finite number of times. ▪

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

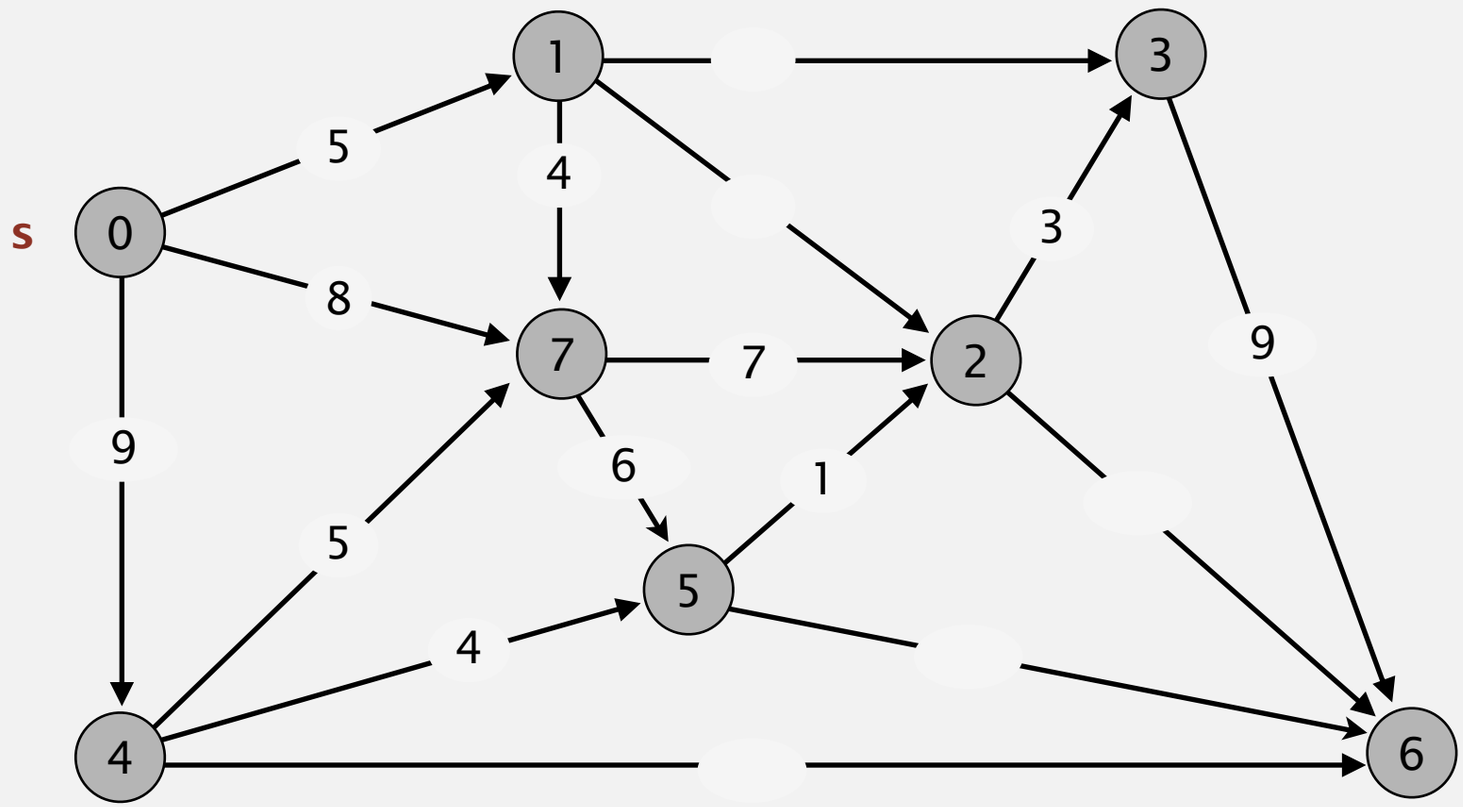
Ex 2. Topological sort algorithm (no directed cycles).

Ex 3. Bellman-Ford algorithm (no negative cycles).

- edge-weighted digraph API
- shortest-paths properties
- **Dijkstra's algorithm**
- edge-weighted DAGs
- negative weights

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.

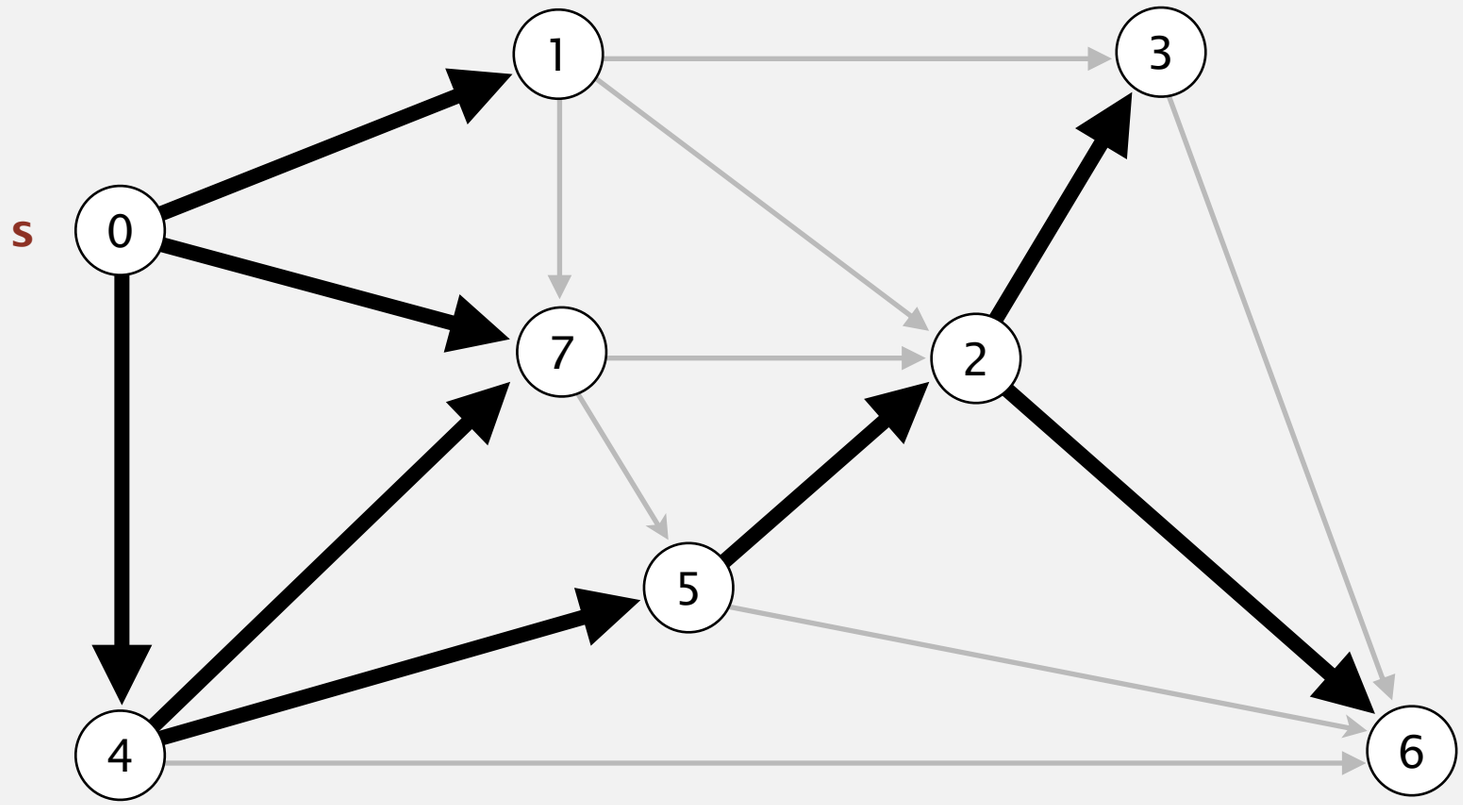


an edge-weighted digraph

```
0→1    5.0
0→4    9.0
0→7    8.0
1→2   12.0
1→3   15.0
1→7    4.0
2→3    3.0
2→6   11.0
3→6    9.0
4→5    4.0
4→6   20.0
4→7    5.0
5→2    1.0
5→6   13.0
7→5    6.0
7→2    7.0
```

Dijkstra's algorithm demo

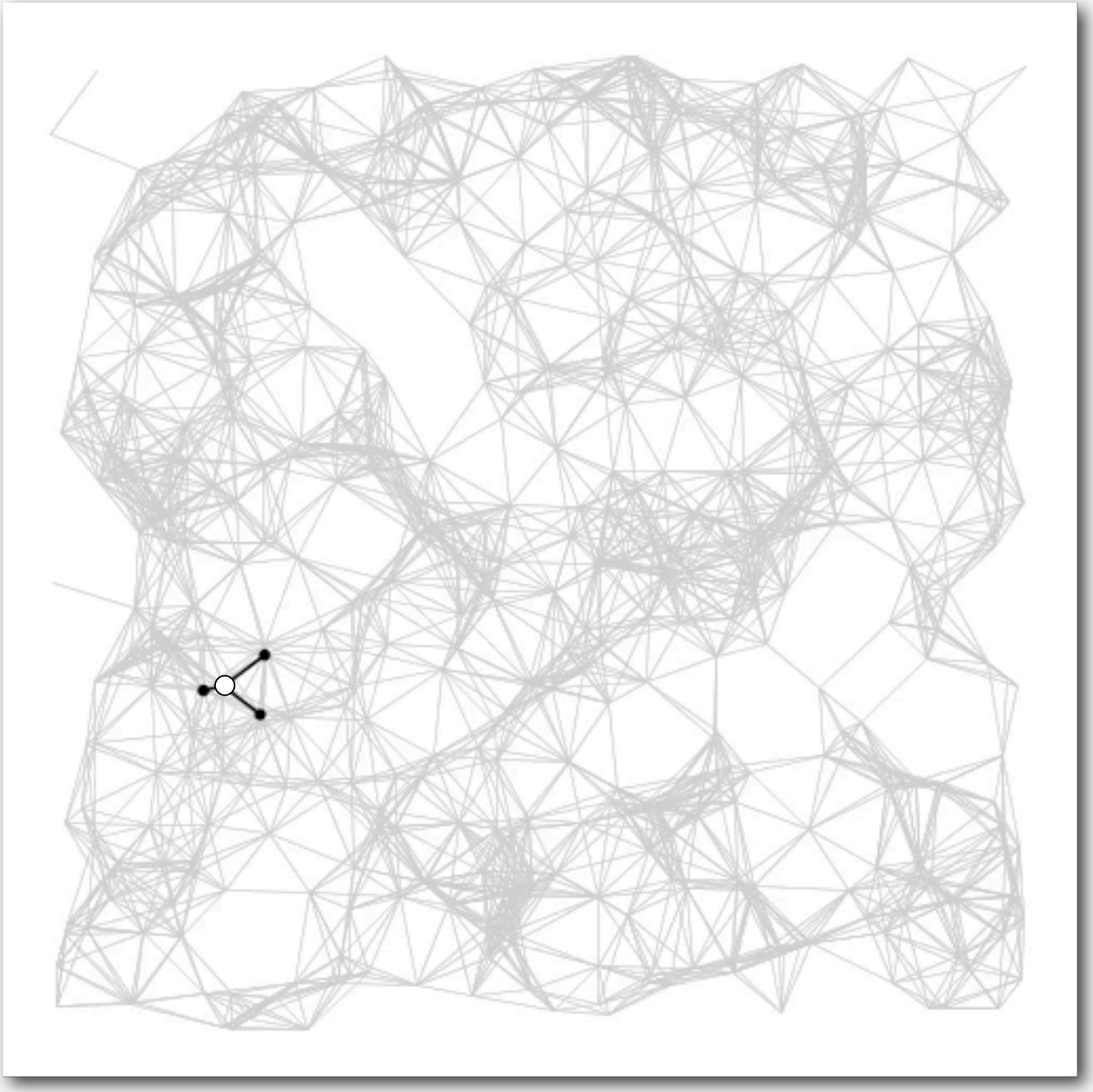
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.



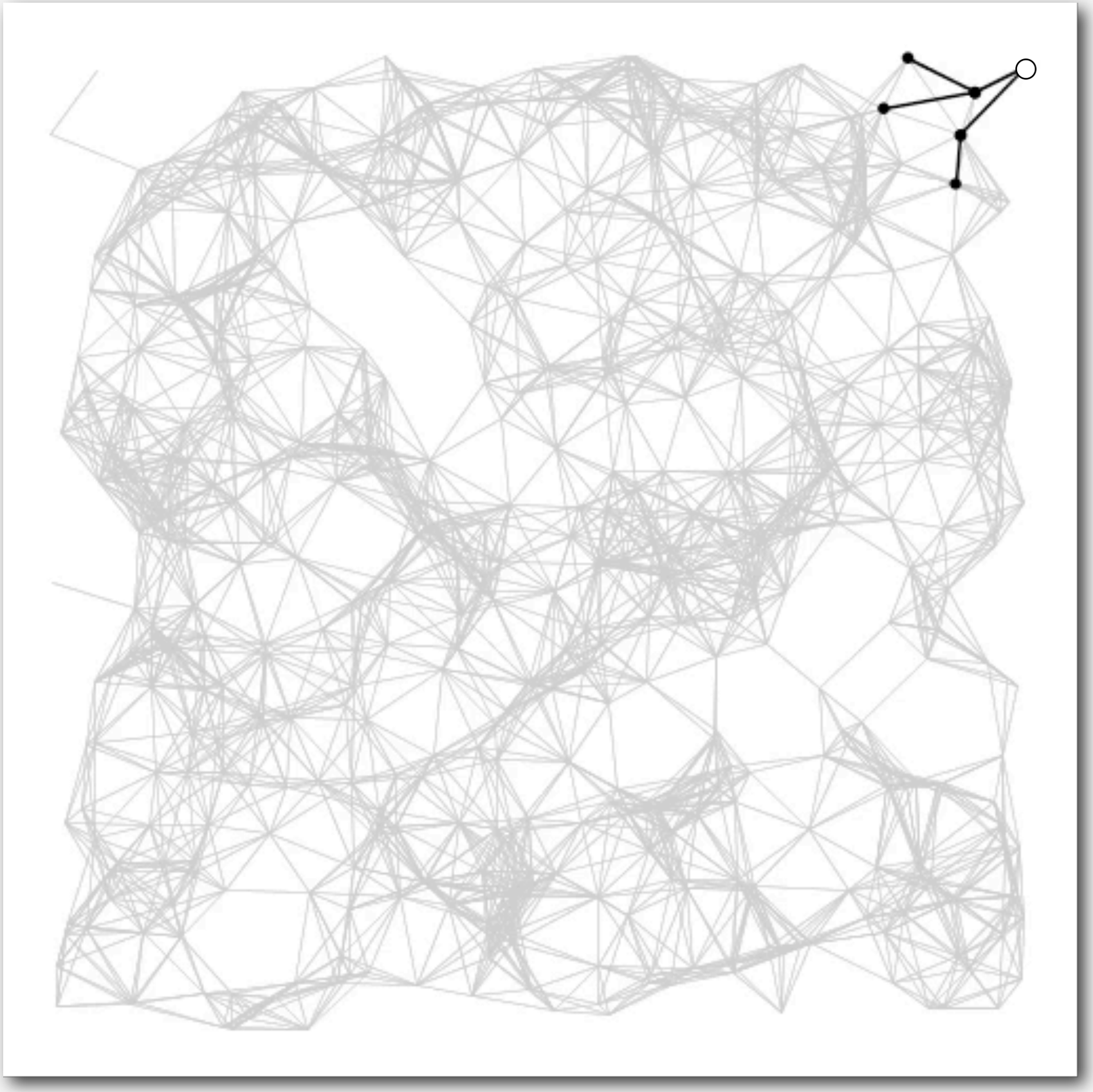
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Dijkstra's algorithm visualization



Dijkstra's algorithm visualization



Dijkstra's algorithm: correctness proof

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when v is relaxed), leaving `distTo[w] ≤ distTo[v] + e.weight()`.
- Inequality holds until algorithm terminates because:
  - `distTo[w]` cannot increase ← `distTo[]` values are monotone decreasing
  - `distTo[v]` will not change ← edge weights are nonnegative and we choose lowest `distTo[]` value at each step
- Thus, upon termination, shortest-paths optimality conditions hold. ▪

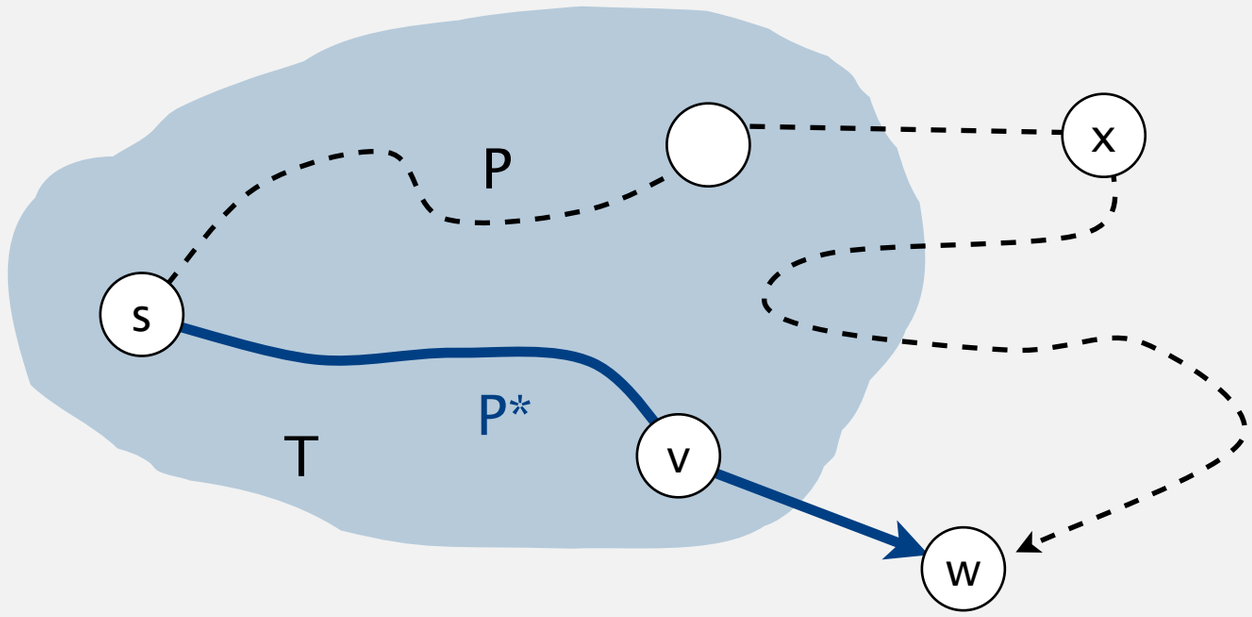
Dijkstra's algorithm: alternate correctness proof

Invariant. For v in T, `distTo[v]` is the length of the shortest path from s to v.

Pf. [by induction on $|T|$]

- Let w be next vertex added to T.
- Let P^* be the $s \rightsquigarrow w$ path through v.
- Consider any other $s \rightsquigarrow w$ path P, and let x be first node on path outside T.
- P is already as long as P^* as soon as it reaches x by greedy choice.
- Thus, `distTo[w]` is the length of the shortest path from s to w.

assuming that edge weights are nonnegative



Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
array	1	V	1	V
binary heap	log V	log V	log V	E log V
d-way heap (Johnson 1975)	d log	d log	log	E log
Fibonacci heap (Fredman-Tarjan 1984)	1	log V	1	E + V log V

† amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- d-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Priority-first search

Insight. Four of our graph-search methods are the same algorithm!

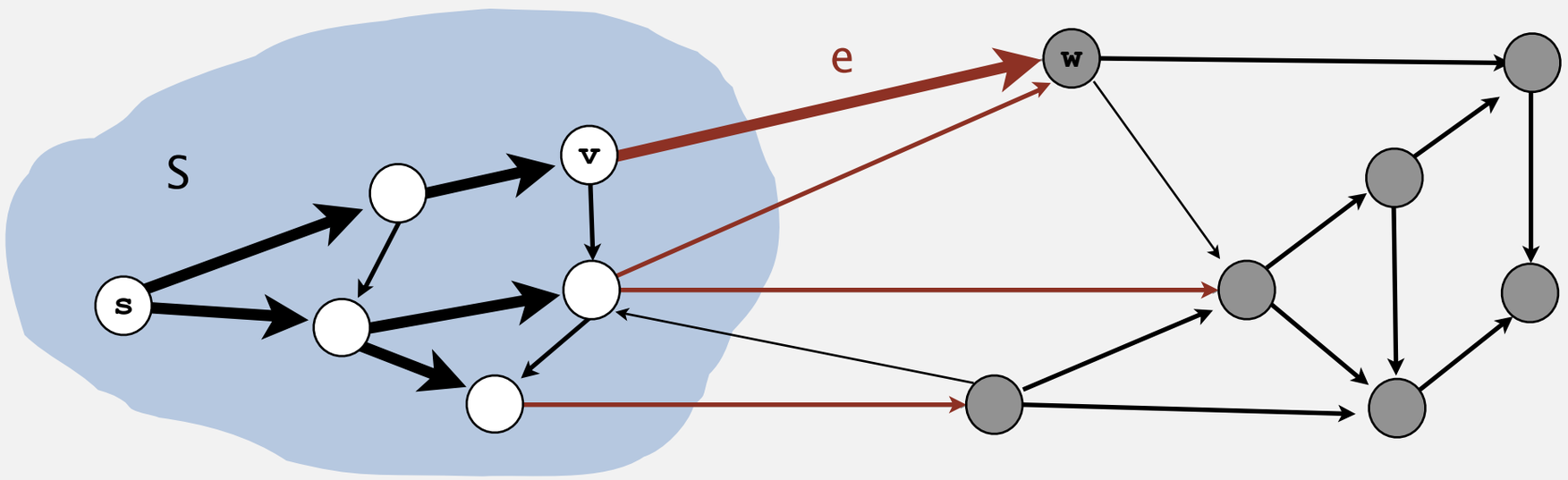
- Maintain a set of explored vertices S.
- Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.

BFS. Take edge from vertex which was discovered least recently.

Prim. Take edge of minimum weight [closest to the tree].

Dijkstra. Take edge to vertex that is closest to S.



Other problems using Dijkstra's algorithm

- Source-sink shortest paths: Given an edge-weighted digraph, a source s, a target t, find the shortest path from s to t
  - terminate as soon as t comes of the priority queue
- Single source shortest paths in undirected graphs
  - obvious: convert digraph into undirected graph
- All-Pairs shortest path:
  - one for each vertex as source
- Shortest paths in euclidean space

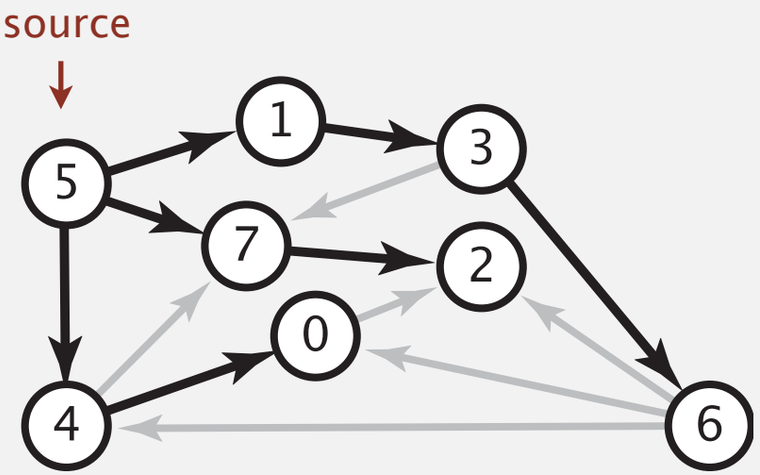
- edge-weighted digraph API
- shortest-paths properties
- Dijkstra's algorithm
- **edge-weighted DAGs**
- negative weights

Acyclic edge-weighted digraphs

Q. Suppose that an edge-weighted digraph has no directed cycles.
Is it easier to find shortest paths than in a general digraph?

A. Yes!

```
5->4  0.35
4->7  0.37
5->7  0.28
5->1  0.32
4->0  0.38
0->2  0.26
3->7  0.39
1->3  0.29
7->2  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



Shortest paths in edge-weighted DAGs

Topological sort algorithm.

- Consider vertices in topologically order.
- Relax all edges pointing from vertex.

Shortest paths in edge-weighted DAGs

Topological sort algorithm.

- Consider vertices in topologically order.
- Relax all edges pointing from vertex.

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$.

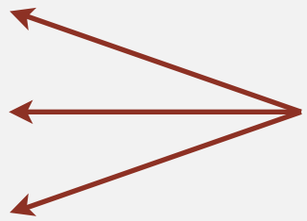
Pf.

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when v is relaxed), leaving `distTo[w] ≤ distTo[v] + e.weight()`.
- Inequality holds until algorithm terminates because:
  - `distTo[w]` cannot increase ← `distTo[]` values are monotone decreasing
  - `distTo[v]` will not change ← because of topological order, no edge pointing to v will be relaxed after v is relaxed
- Thus, upon termination, shortest-paths optimality conditions hold. ▪

Longest paths in edge-weighted DAGs

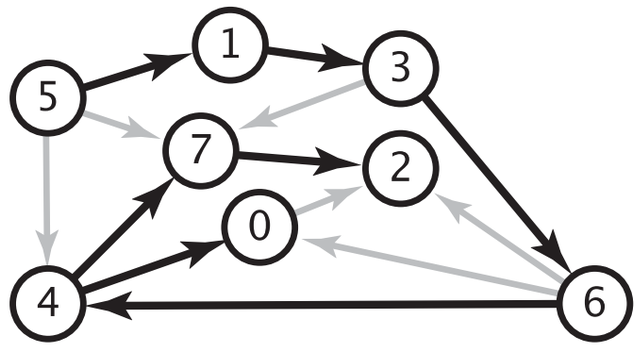
Formulate as a shortest paths problem in edge-weighted DAGs.

- Negate all weights.
- Find shortest paths.
- Negate weights in result.



equivalent: reverse sense of equality in `relax()`

longest paths input		shortest paths input	
5->4	0.35	5->4	-0.35
4->7	0.37	4->7	-0.37
5->7	0.28	5->7	-0.28
5->1	0.32	5->1	-0.32
4->0	0.38	4->0	-0.38
0->2	0.26	0->2	-0.26
3->7	0.39	3->7	-0.39
1->3	0.29	1->3	-0.29
7->2	0.34	7->2	-0.34
6->2	0.40	6->2	-0.40
3->6	0.52	3->6	-0.52
6->0	0.58	6->0	-0.58
6->4	0.93	6->4	-0.93

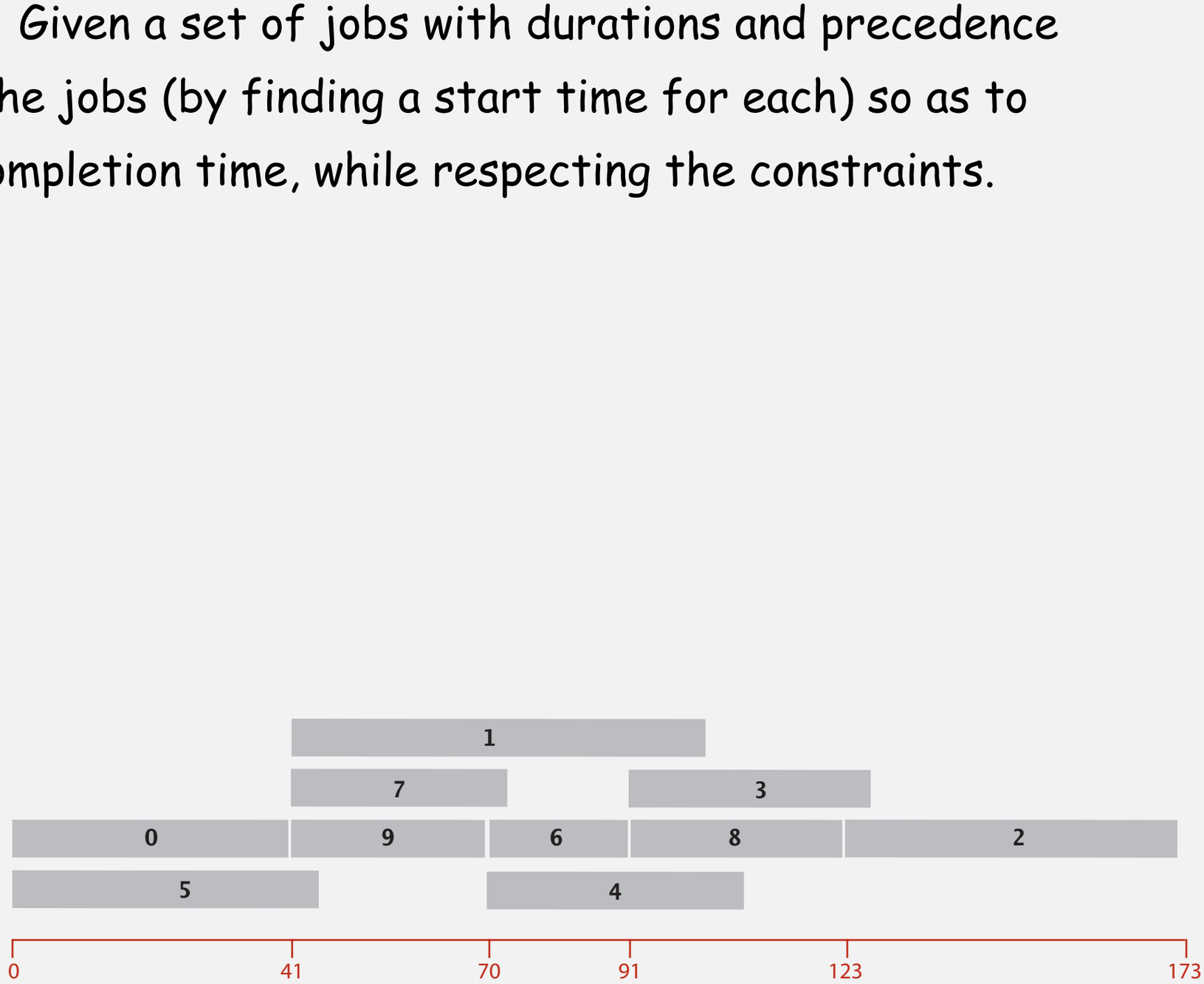


Key point. Topological sort algorithm works even with negative edge weights.

Longest paths in edge-weighted DAGs: application

Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

job	*duration*	*must complete before*	
0	41.0	1	7 9
1	51.0	2	
2	50.0		
3	36.0		
4	38.0		
5	45.0		
6	21.0	3	8
7	32.0	3	8
8	32.0	2	
9	29.0	4	6



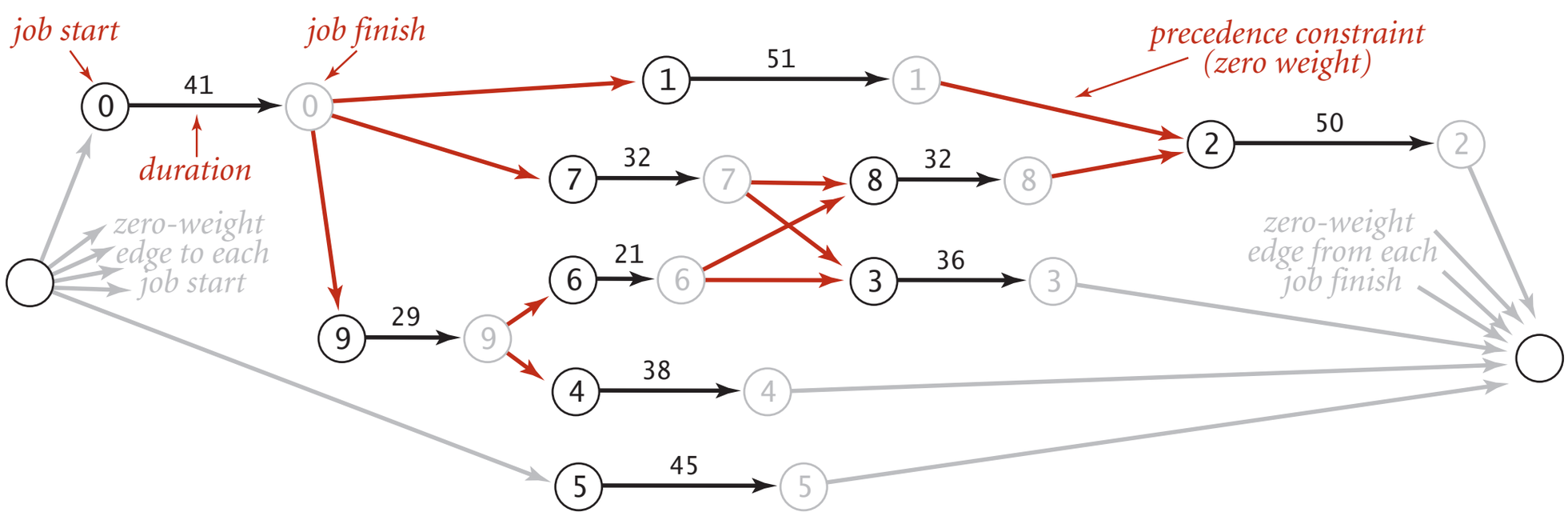
Parallel job scheduling solution

Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:

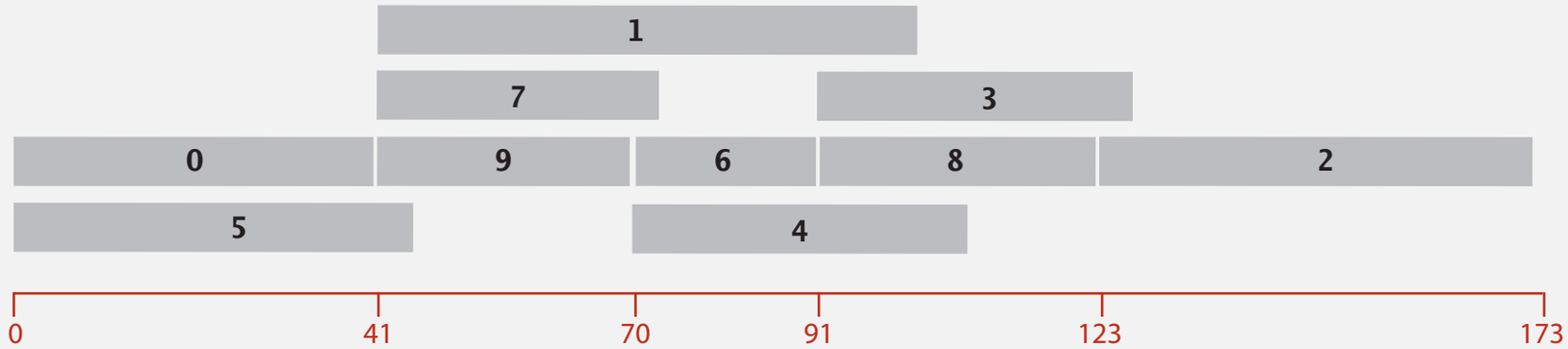
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
  - begin to end (weighted by duration)
  - source to begin (0 weight)
  - end to sink (0 weight)

job	*duration*	*must complete before*
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6

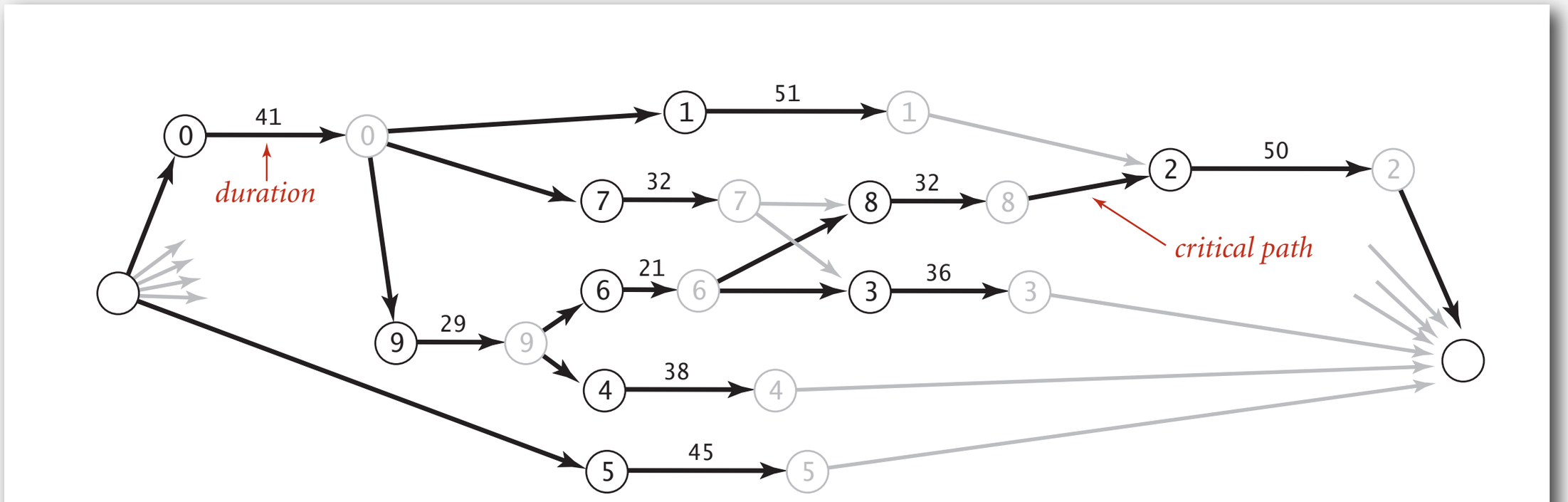


Critical path method

CPM. Use longest path from the source to schedule each job.



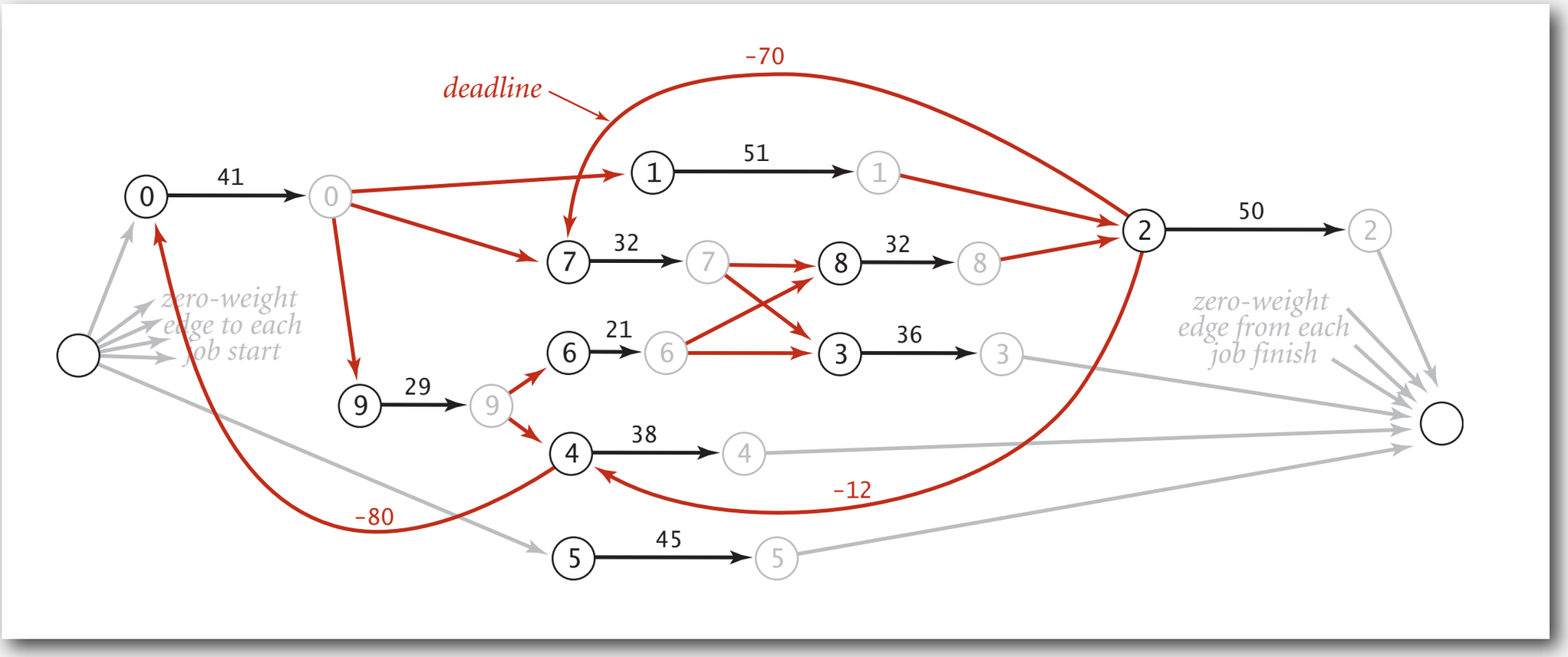
Parallel job scheduling solution



Deep water

Deadlines. Add extra constraints to the parallel job-scheduling problem.

Ex. "Job 2 must start no later than 12 time units after job 4 starts."



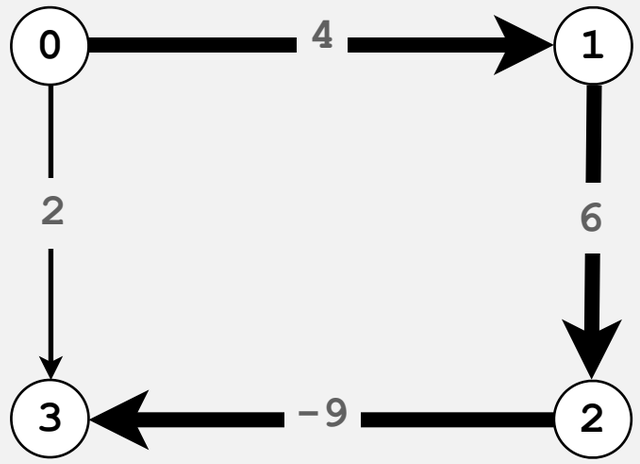
Consequences.

- Corresponding shortest-paths problem has cycles (and negative weights).
- Possibility of infeasible problem (negative cycles).

- edge-weighted digraph API
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- **negative weights**

Shortest paths with negative weights: failed attempts

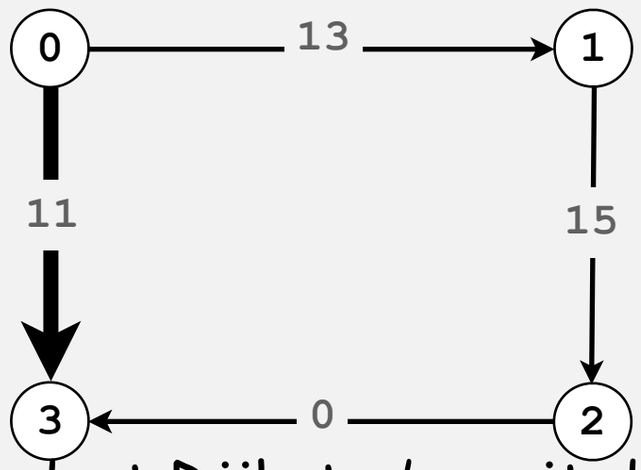
Dijkstra. Doesn't work with negative edge weights. Intuition of having least number of edges breaks, i.e., low-weight paths have more edges.



Dijkstra selects vertex 3 immediately after 0.

But shortest path from 0 to 3 is 0→1→2→3.

Re-weighting. Add a constant to every edge weight doesn't work. Bears little relationship to the shortest path in the old graph



Adding 9 to each edge weight changes the shortest path from 0→1→2→3 to 0→3.

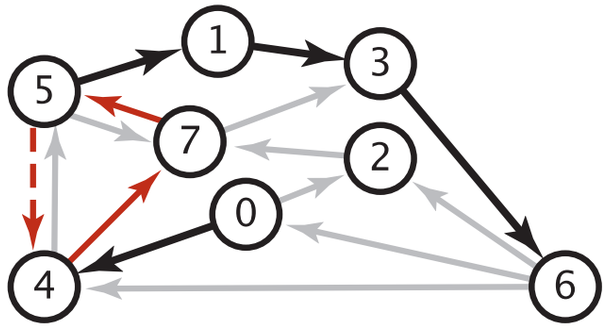
Bad news. Can't adapt Djikstra's as it depends on examining paths in increasing order of distance from source. Need a different algorithm.

Negative cycles

Def. A negative cycle is a directed cycle whose sum of edge weights is negative.

digraph

```
4->5  0.35
5->4 -0.66
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

shortest path from 0 to 6

0->4->7->5->4->7->5...->1->3->6

Proposition. A SPT exists iff no negative cycles.

assuming all vertices reachable from s

Shortest paths with negative weights: dynamic programming algorithm

Dynamic programming algorithm

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat V times:

- **Relax each edge.**

```
for (int i = 0; i < G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
         relax(e);
```

← pass i (relax each edge)

Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

Pf idea. After pass i, found shortest path containing at most i edges.

Bellman-Ford algorithm

Observation. If `distTo[v]` does not change during pass i, no need to relax any edge pointing from v in pass $i+1$.

FIFO implementation. Maintain queue of vertices whose `distTo[]` changed.

be careful to keep at most one copy of each vertex on queue (why?)

Overall effect.

- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Bellman-Ford algorithm demo

Single source shortest-paths implementation: cost summary

algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	E + V	E + V	V
Dijkstra (binary heap)	no negative weights	E log V	E log V	V
dynamic programming	no negative cycles	E V	E V	V
Bellman-Ford		E + V	E V	V

Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

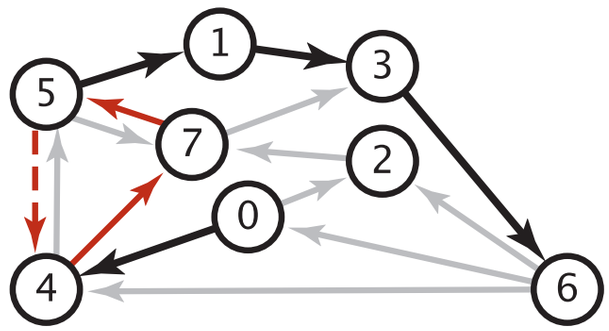
Finding a negative cycle

Negative cycle. Add two method to the API for `SP`.

`boolean hasNegativeCycle()`	*is there a negative cycle?*
`Iterable <DirectedEdge> negativeCycle()`	*negative cycle reachable from s*

digraph

```
4->5  0.35
5->4 -0.66
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```

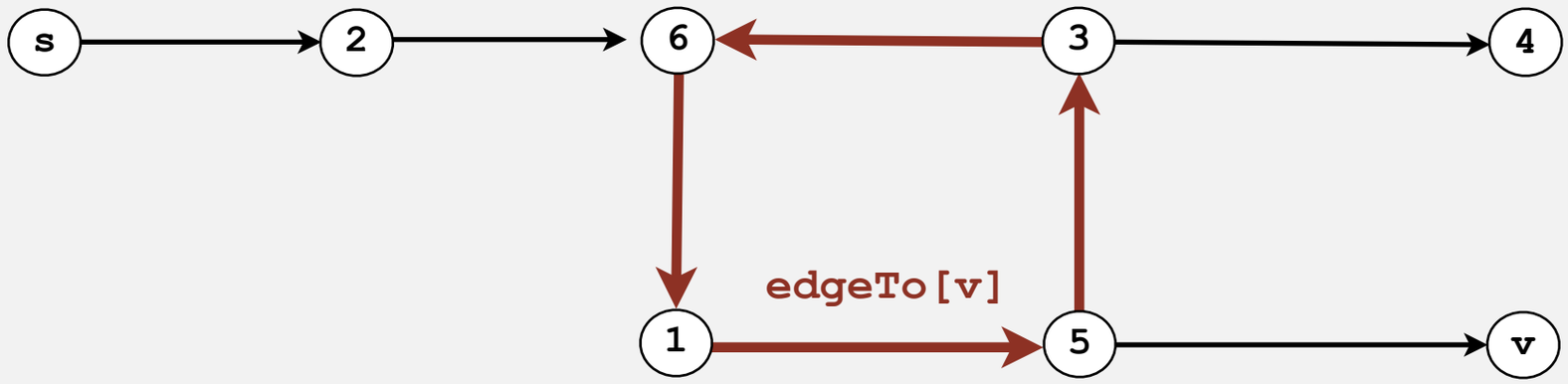


negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.



Proposition. If any vertex v is updated in phase V, there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

In practice. Check for negative cycles more frequently.

Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.35	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.62	1	0.953
CAD	0.995	0.732	0.65	1.049	1

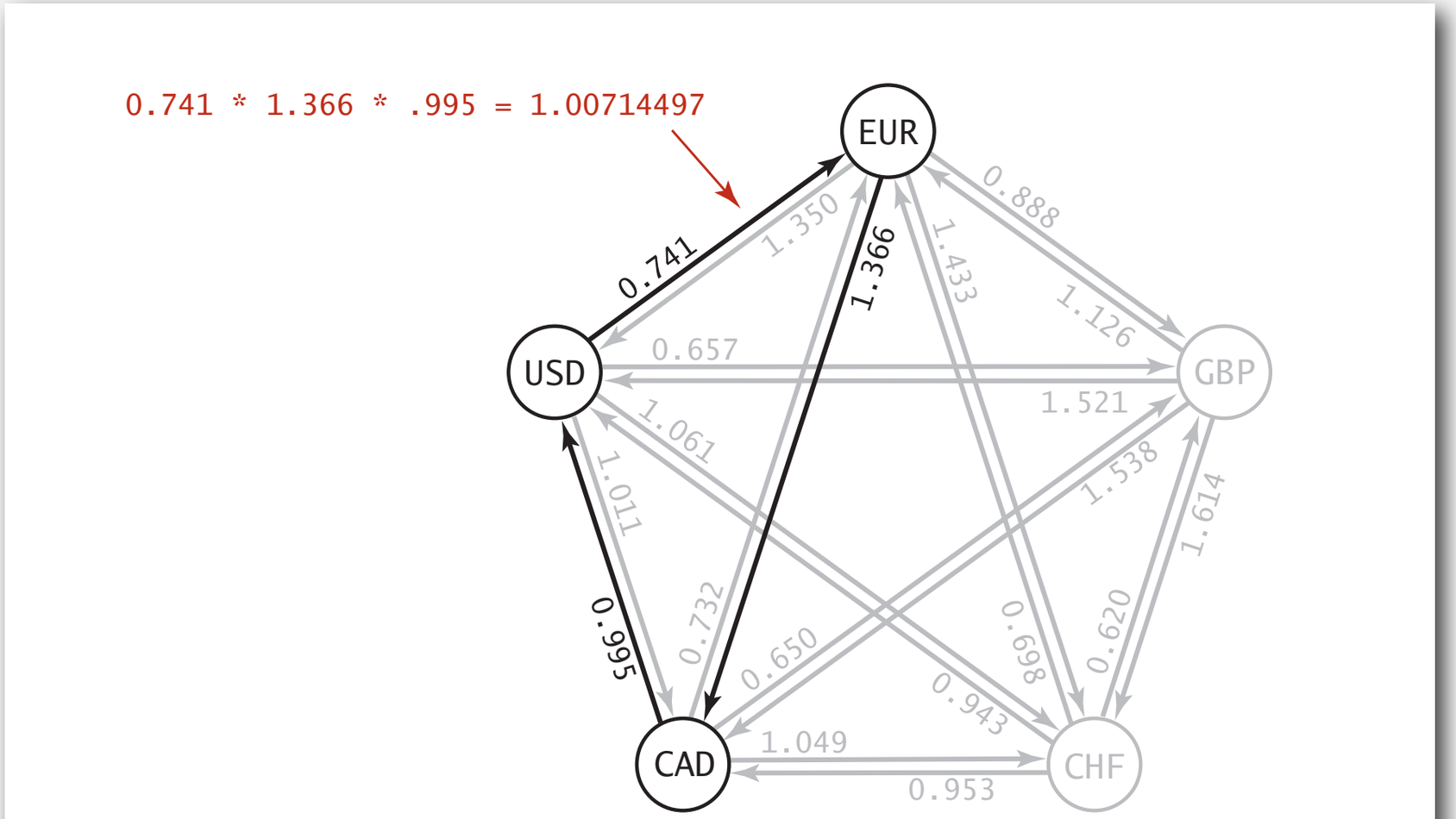
Ex. \$1,000 ⇒ 741 Euros ⇒ 1,012.206 Canadian dollars ⇒ \$1,007.14497.

1000 × 0.741 × 1.366 × 0.995 = 1007.14497

Negative cycle application: arbitrage detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1.

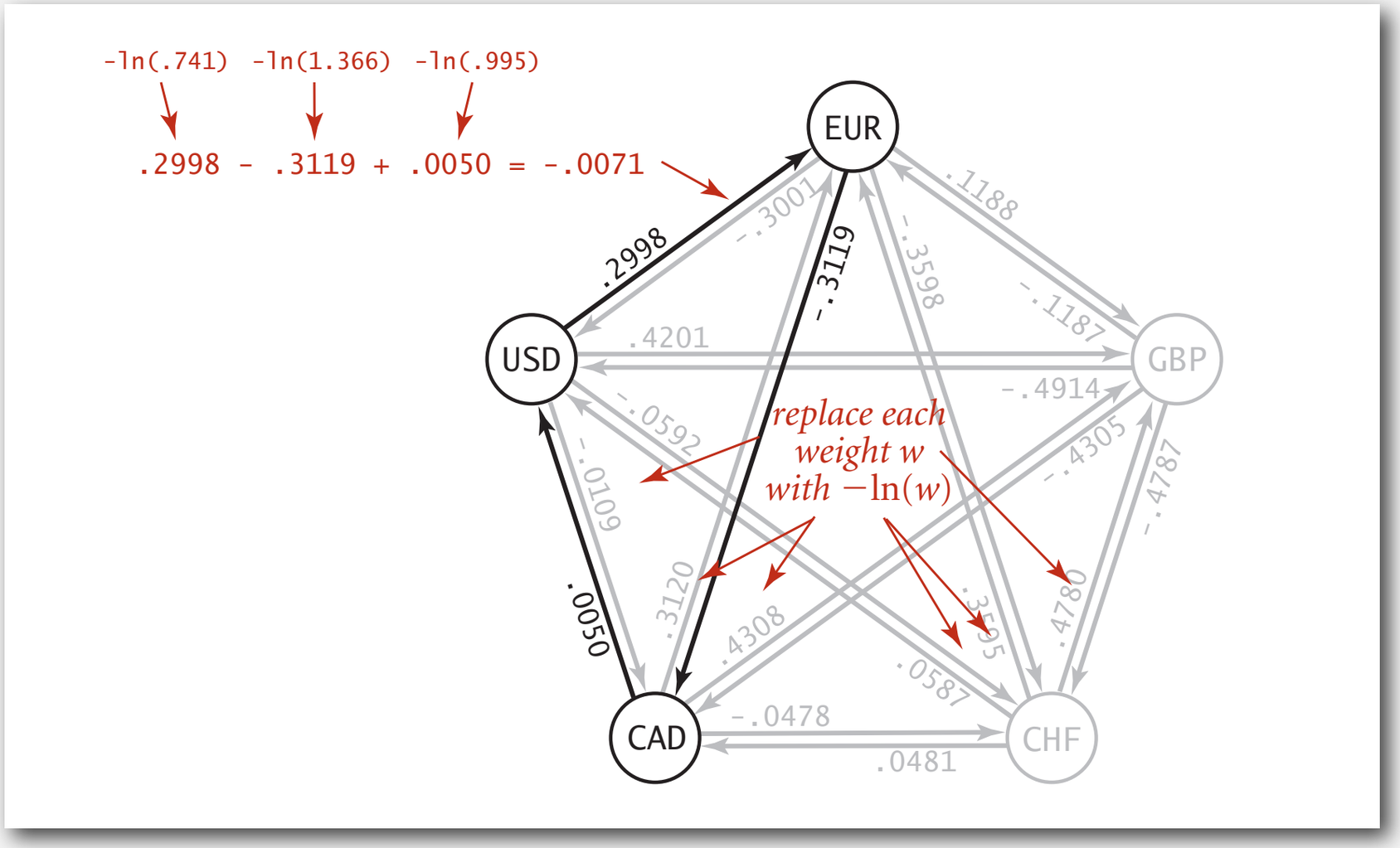


Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ be $-\ln$ (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0.
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).



Remark. Fastest algorithm is extraordinarily valuable!

Shortest paths summary

Dijkstra's algorithm.

- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.

- Arise in applications.
- Faster than Dijkstra's algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.

- Arise in applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

Shortest-paths is a broadly useful problem-solving model.

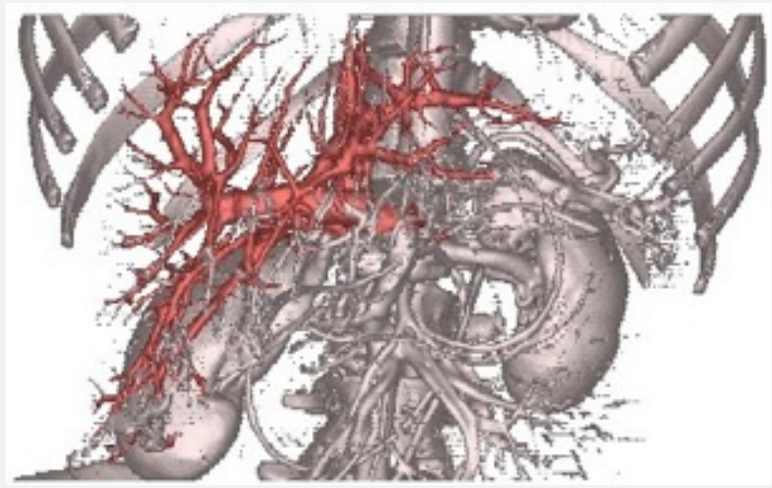
Maximum Flow

- overview
- **Ford-Fulkerson algorithm**
- analysis
- application

Maxflow and mincut applications

Maxflow/mincut is a widely applicable problem-solving model.

- Data mining.
- Open-pit mining.
- Bipartite matching.
- Network reliability.
- Image segmentation.
- Network connectivity.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Multi-camera scene reconstruction.
- Sensor placement for homeland security.
- Baseball elimination. ← see next programming assignment
- Many, many, more.



liver and hepatic vascularization segmentation

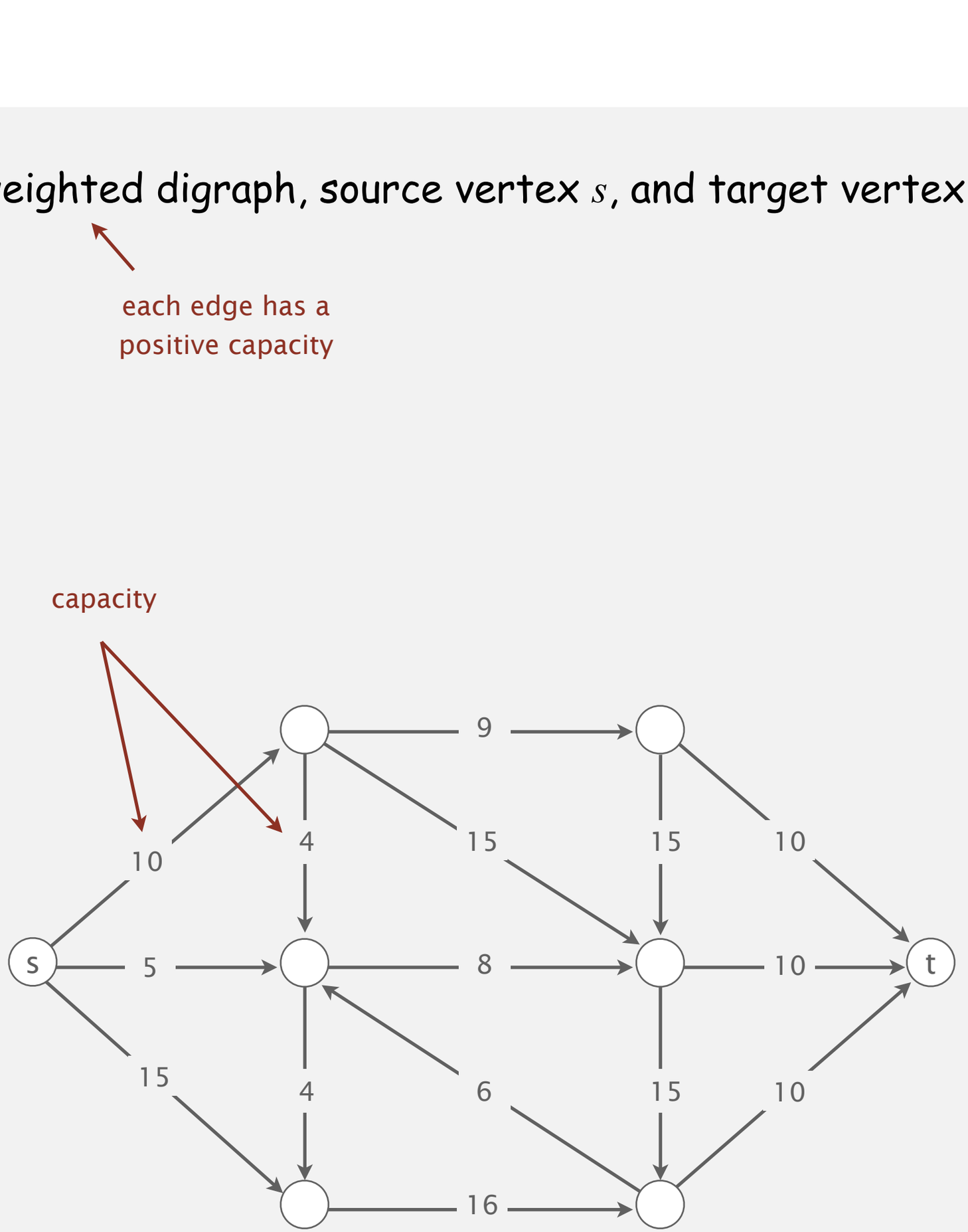
- overview
- Ford-Fulkerson algorithm
- analysis
- applications

Mincut problem

Input. A weighted digraph, source vertex s, and target vertex t.

each edge has a positive capacity

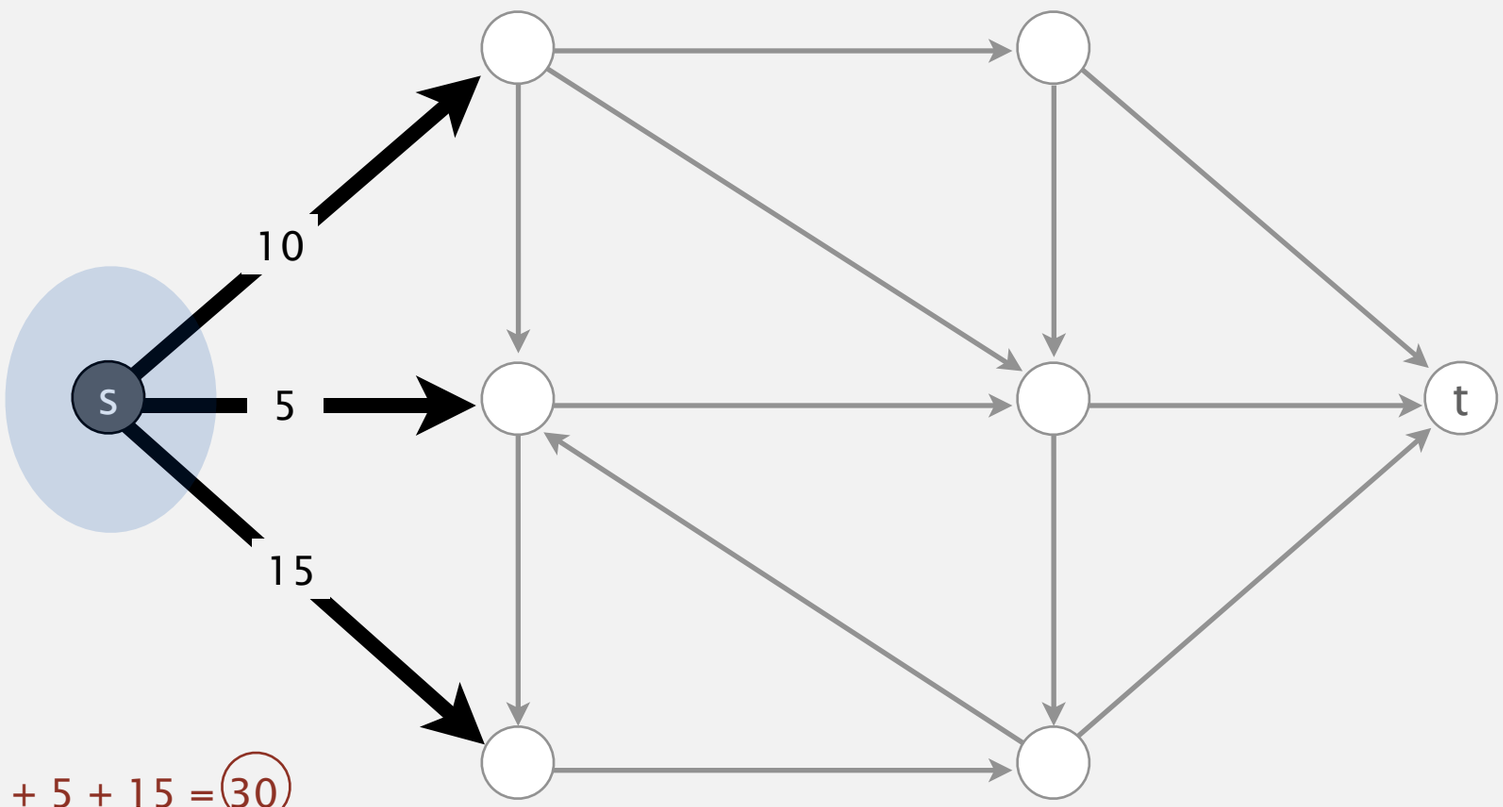
capacity



Mincut problem

Def. A st-cut (cut) is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B.

Def. Its capacity is the sum of the capacities of the edges from A to B.

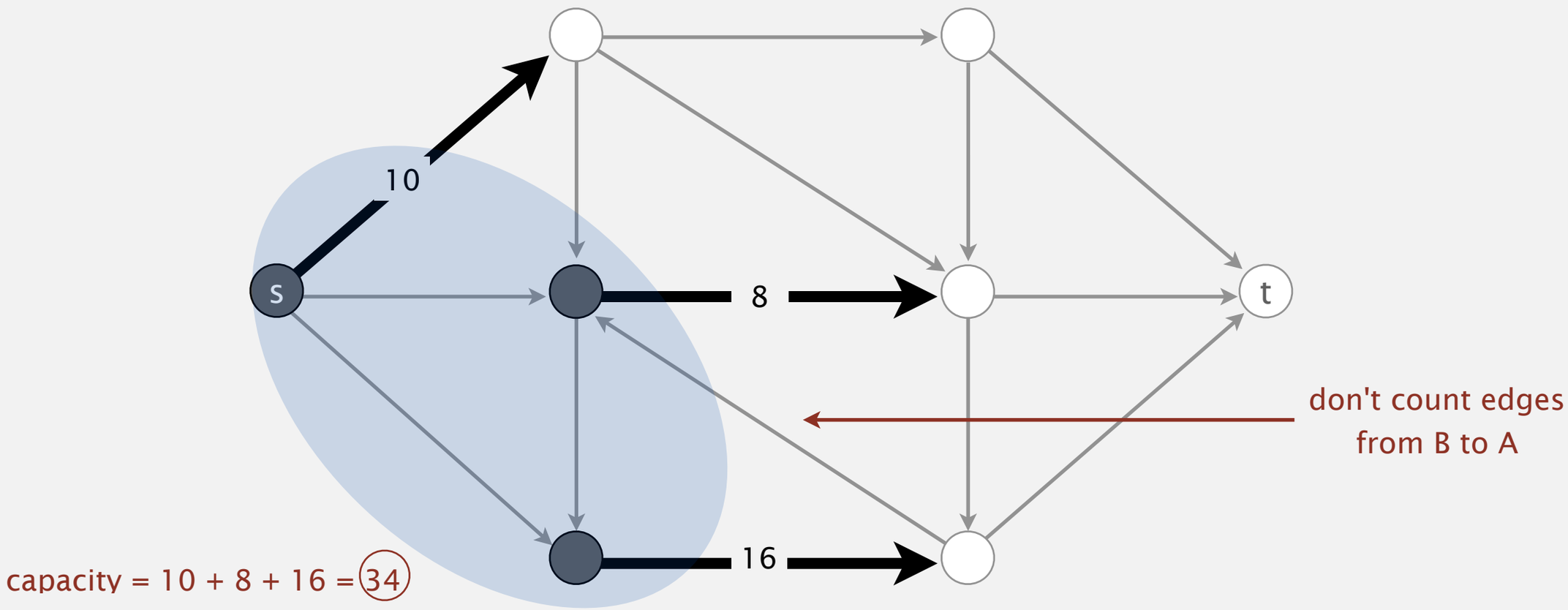


capacity = 10 + 5 + 15 = 30

Mincut problem

Def. A st-cut (cut) is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B.

Def. Its capacity is the sum of the capacities of the edges from A to B.

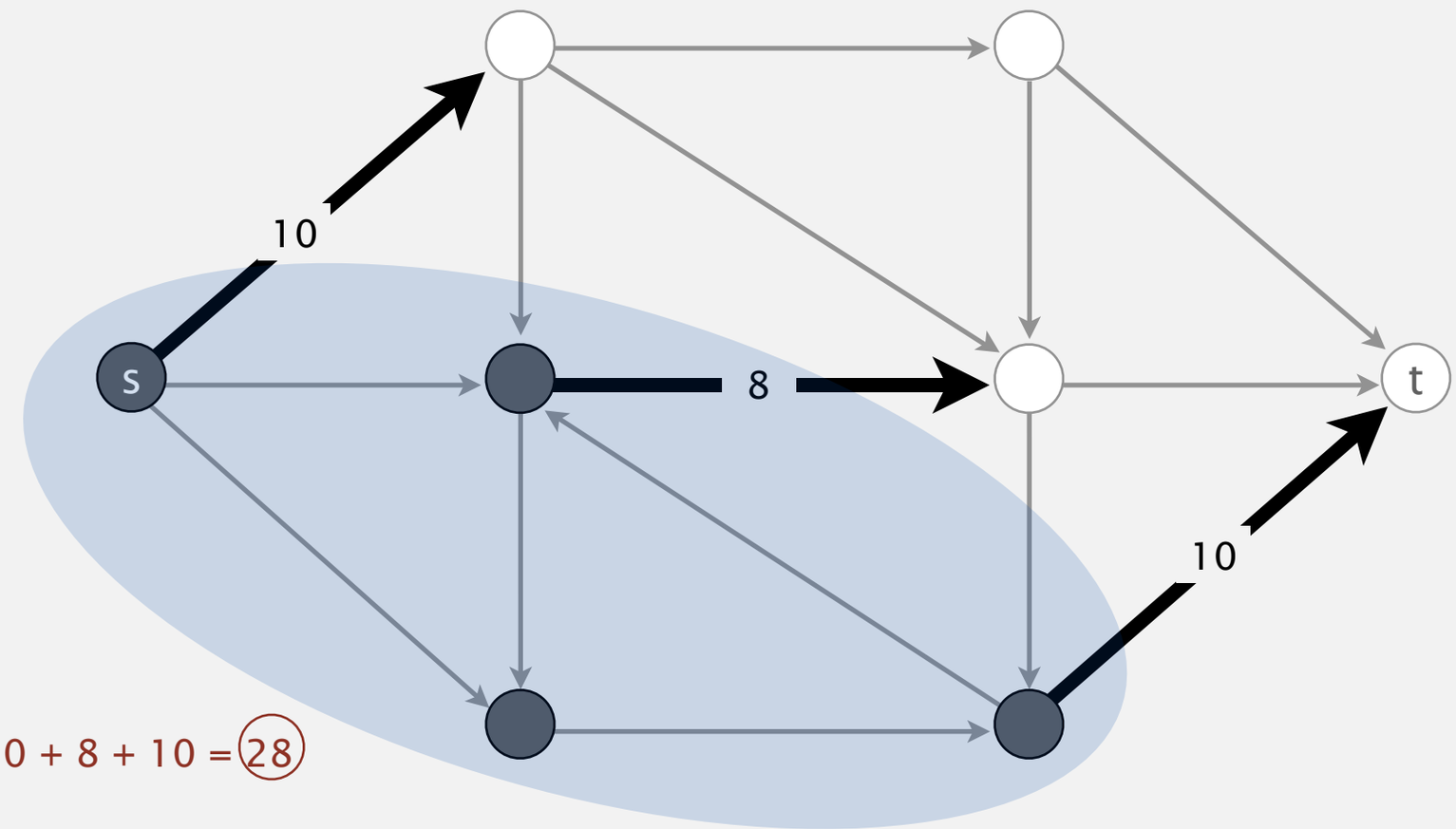


Mincut problem

Def. A st-cut (cut) is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B.

Def. Its capacity is the sum of the capacities of the edges from A to B.

Minimum st-cut (mincut) problem. Find a cut of minimum capacity.

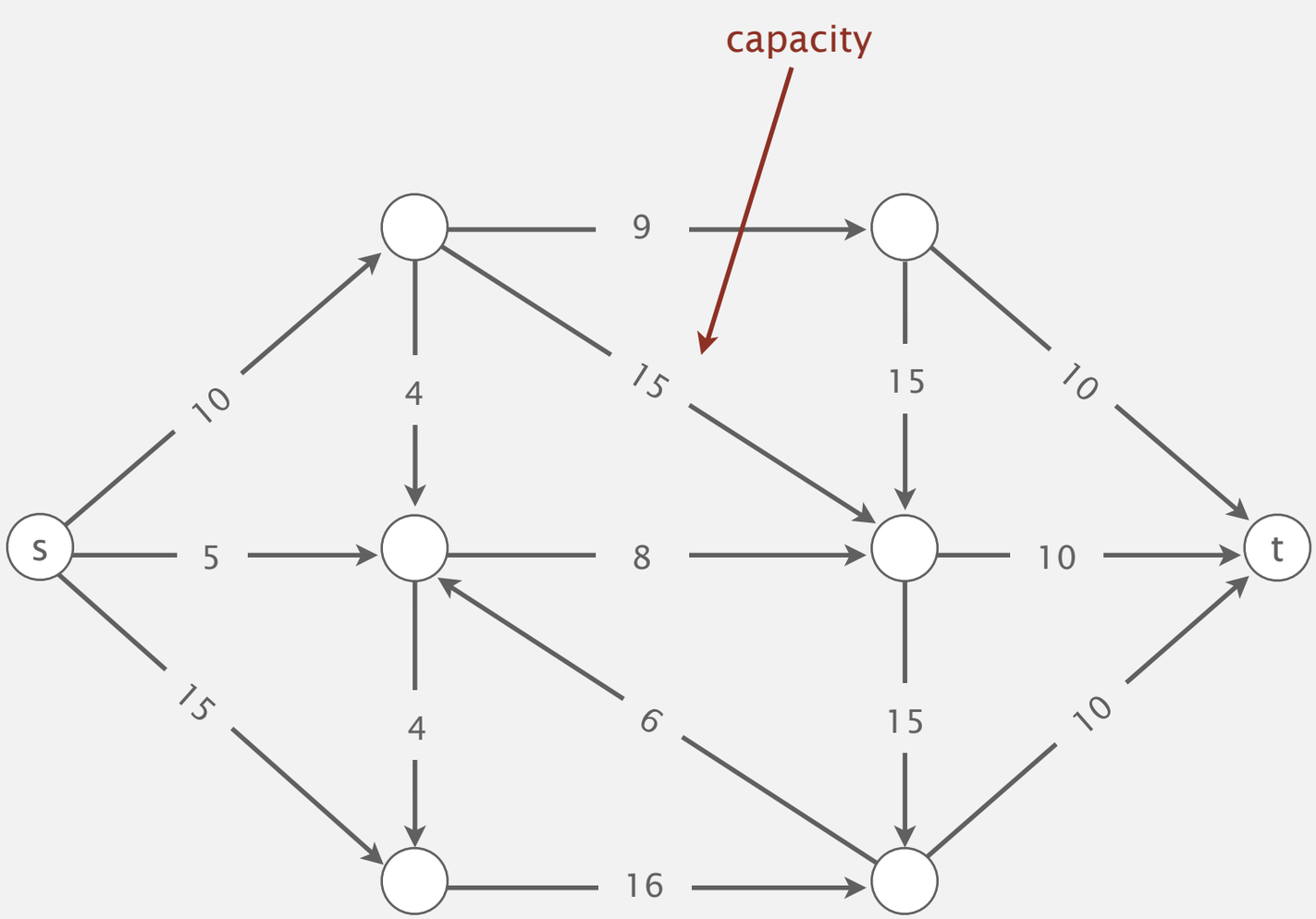


capacity = 10 + 8 + 10 = 28

Maxflow problem

Input. A weighted digraph, source vertex s, and target vertex t.

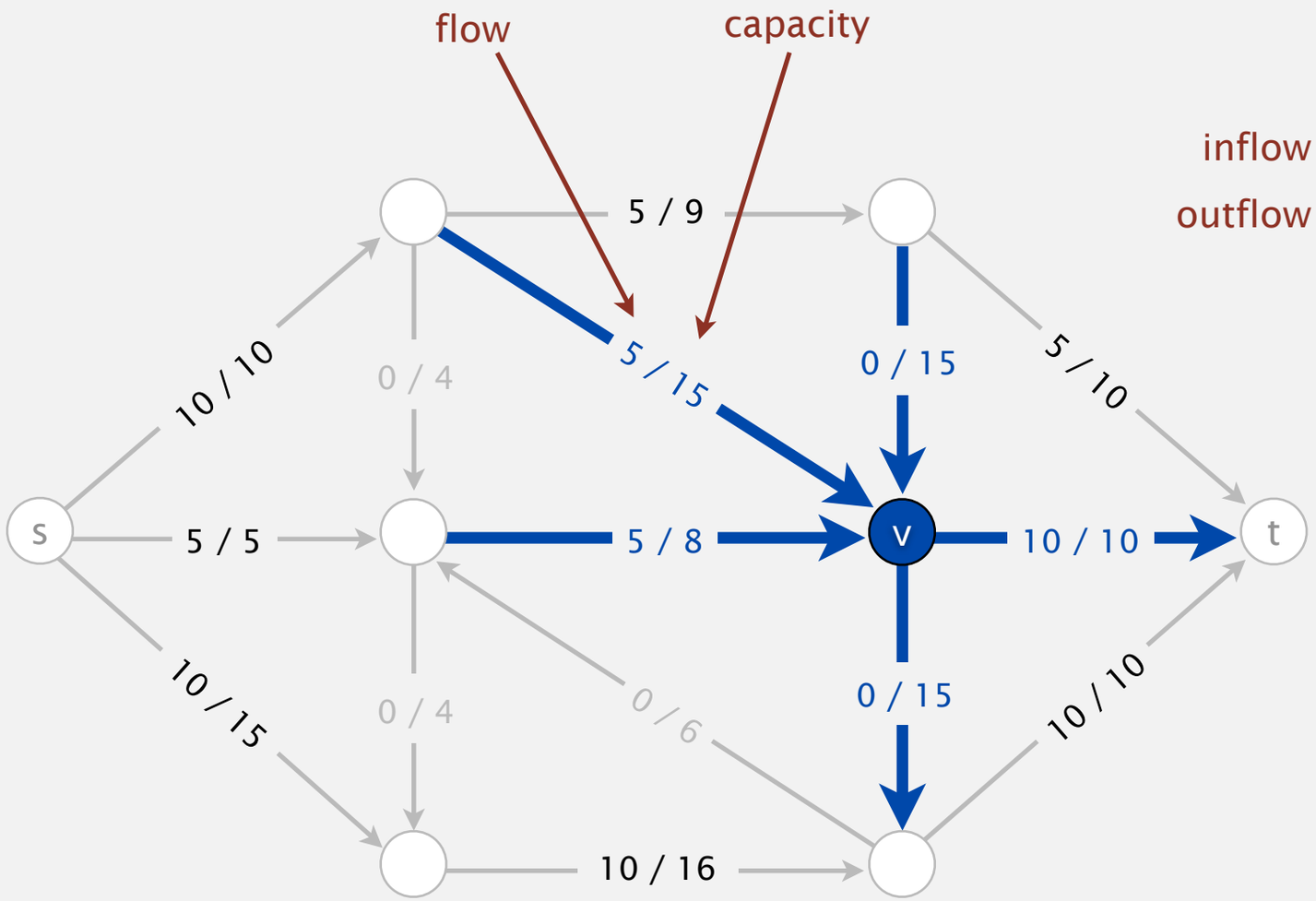
each edge has a positive capacity



Maxflow problem

Def. An st-flow (flow) is an assignment of values to the edges such that:

- Capacity constraint: $0 \le$ edge's flow $\le$ edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).



inflow at v = 5 + 5 + 0 = 10

outflow at v = 10 + 0 = 10

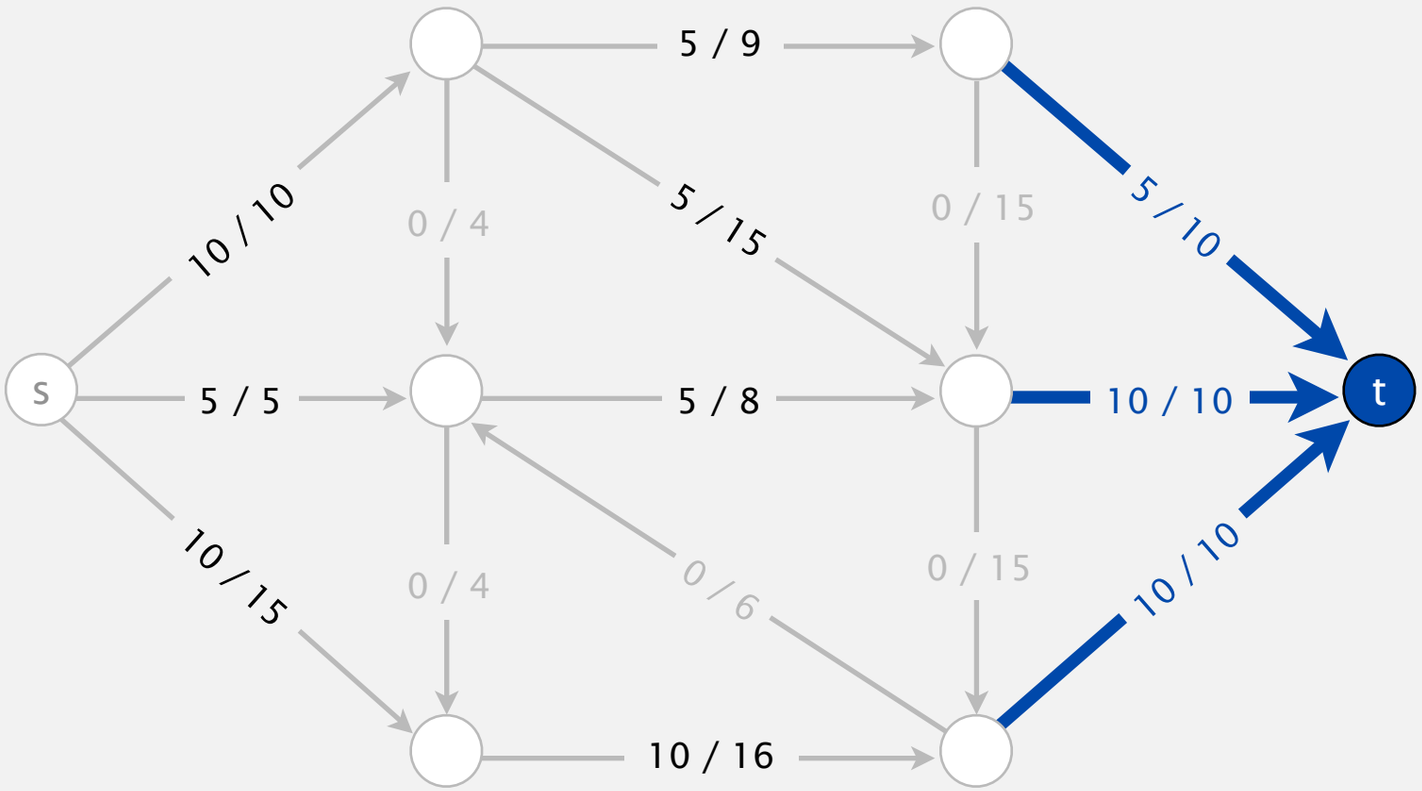
Maxflow problem

Def. An st-flow (flow) is an assignment of values to the edges such that:

- Capacity constraint: $0 \le$ edge's flow $\le$ edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).

Def. The value of a flow is the inflow at t.

we assume no edge points from s or to t



value = 5 + 10 + 10 = 25

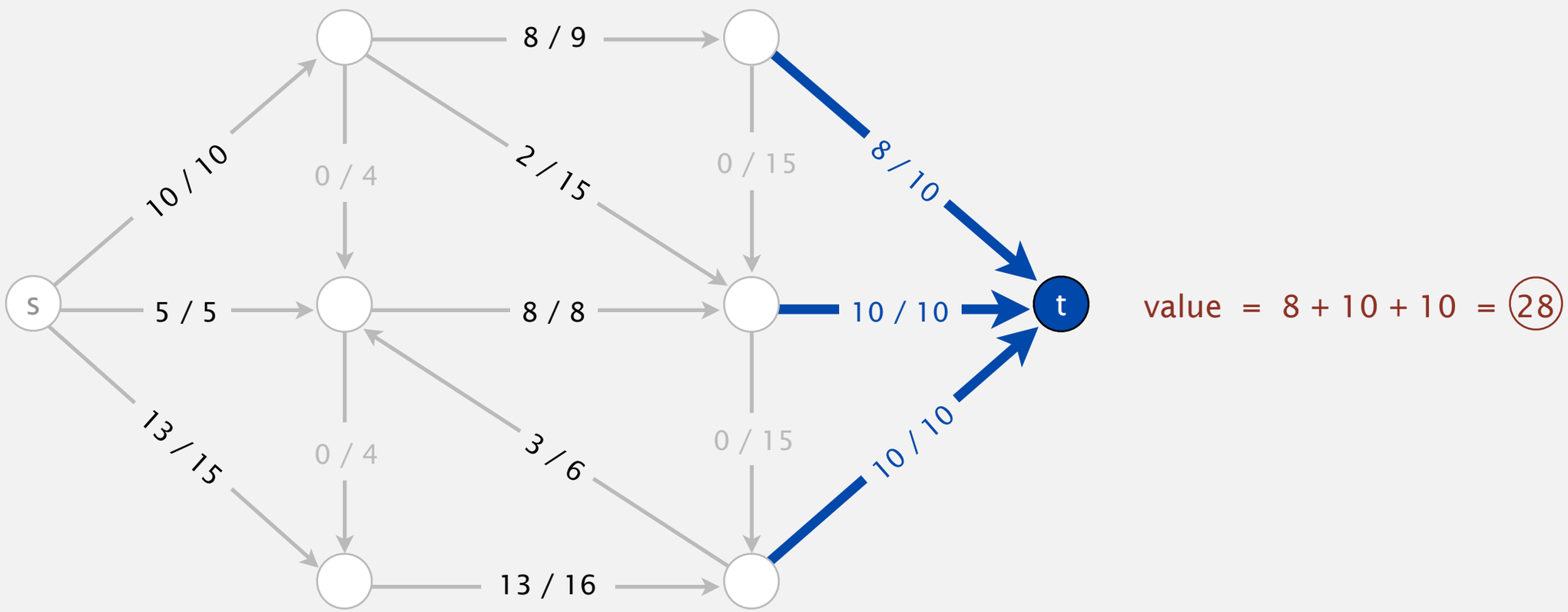
Maxflow problem

Def. An st-flow (flow) is an assignment of values to the edges such that:

- Capacity constraint: $0 \le$ edge's flow $\le$ edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).

Def. The value of a flow is the inflow at t.

Maximum st-flow (maxflow) problem. Find a flow of maximum value.

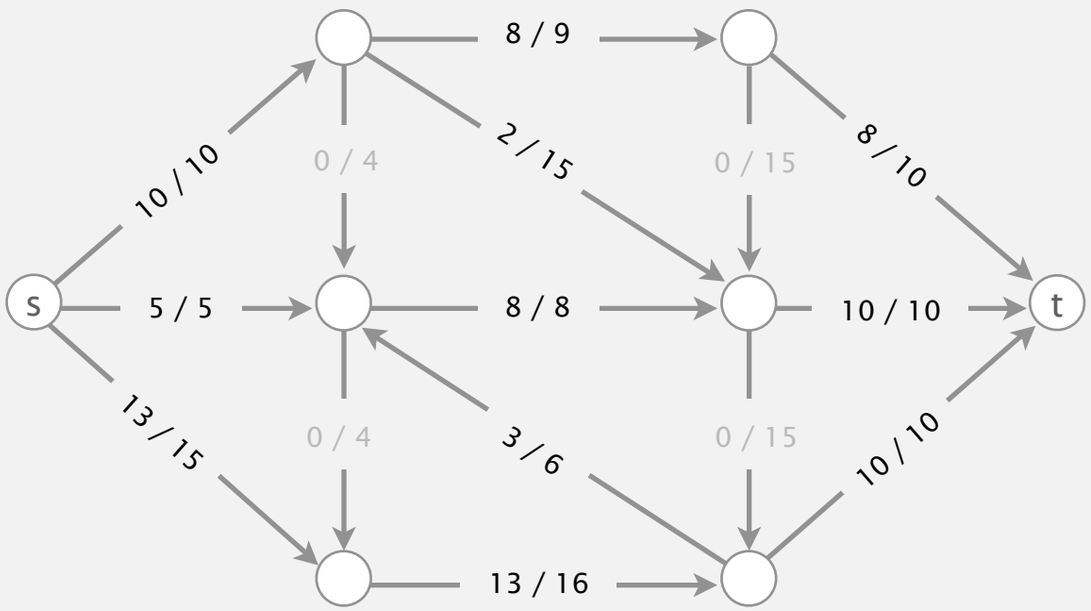


Summary

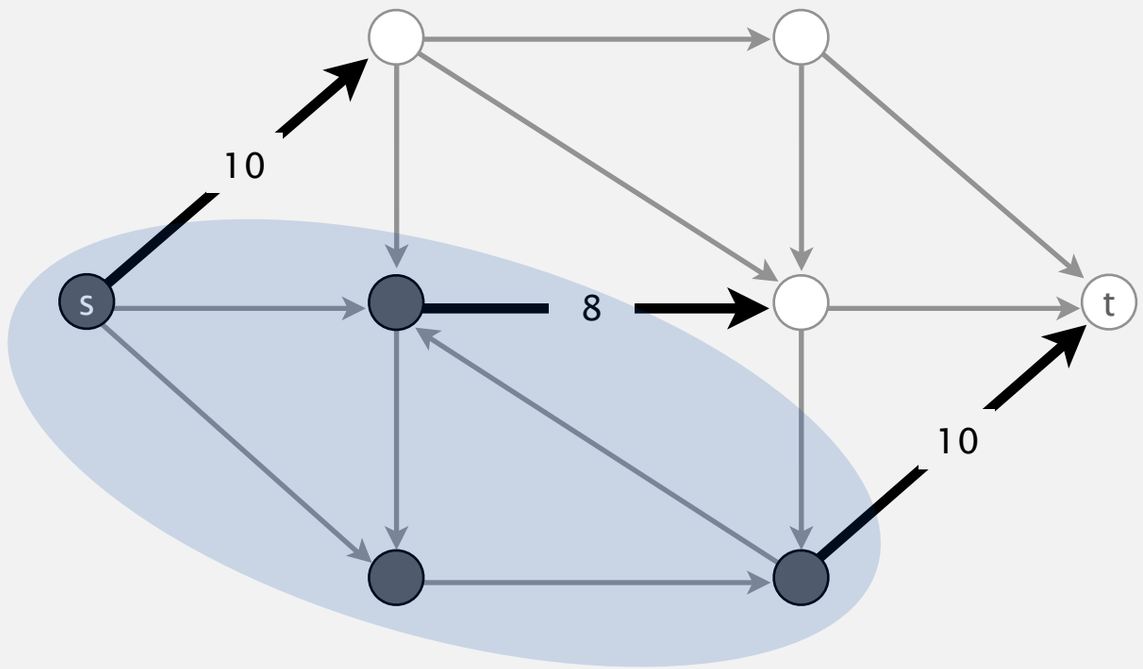
Input. A weighted digraph, source vertex s, and target vertex t.

Mincut problem. Find a cut of minimum capacity.

Maxflow problem. Find a flow of maximum value.



value of flow = 28



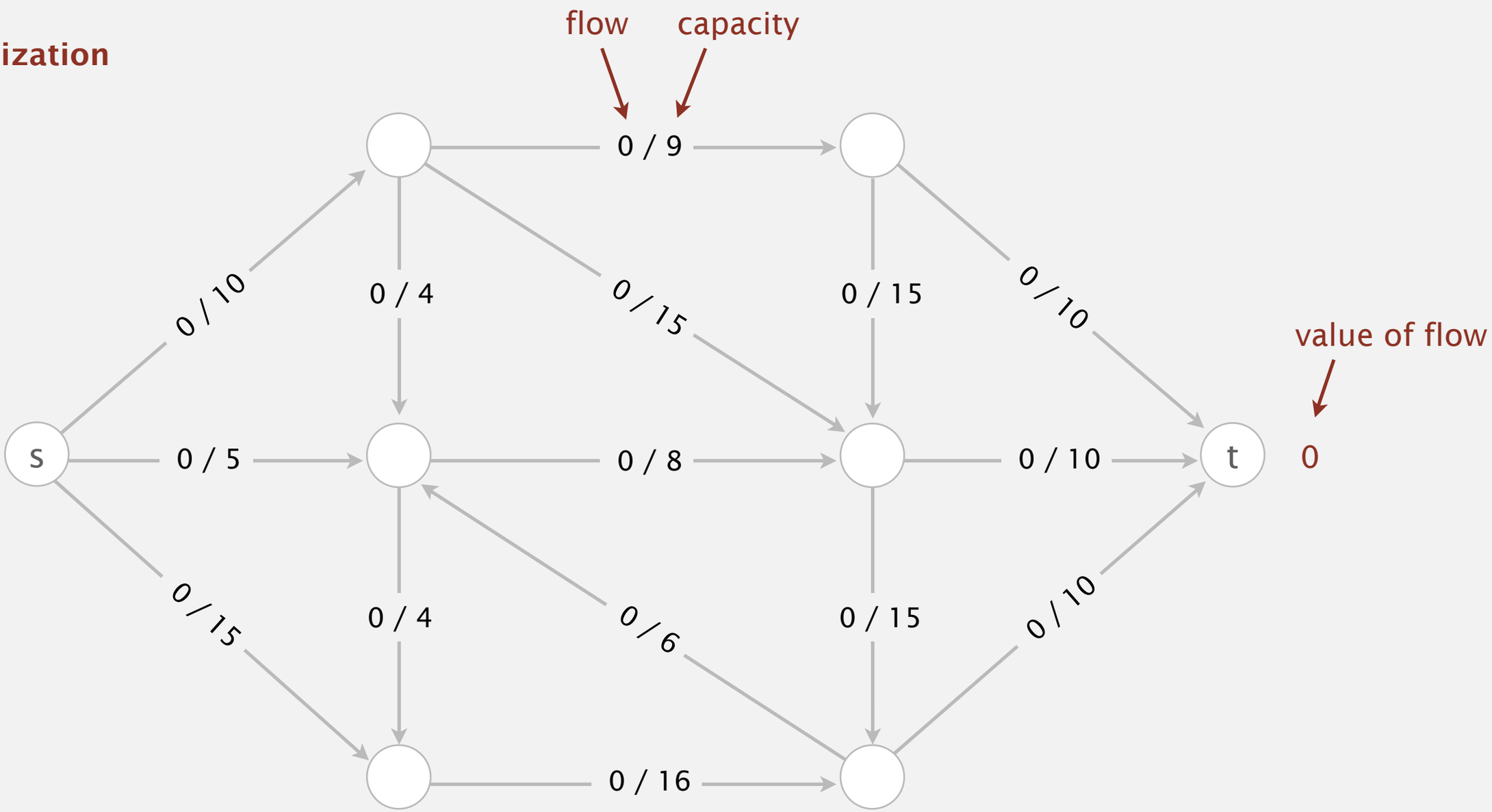
capacity of cut = 28

Remarkable fact. These two problems are dual!

- overview
- **Ford-Fulkerson algorithm**
- analysis
- Java implementation
- applications

Ford-Fulkerson algorithm

Initialization. Start with 0 flow.

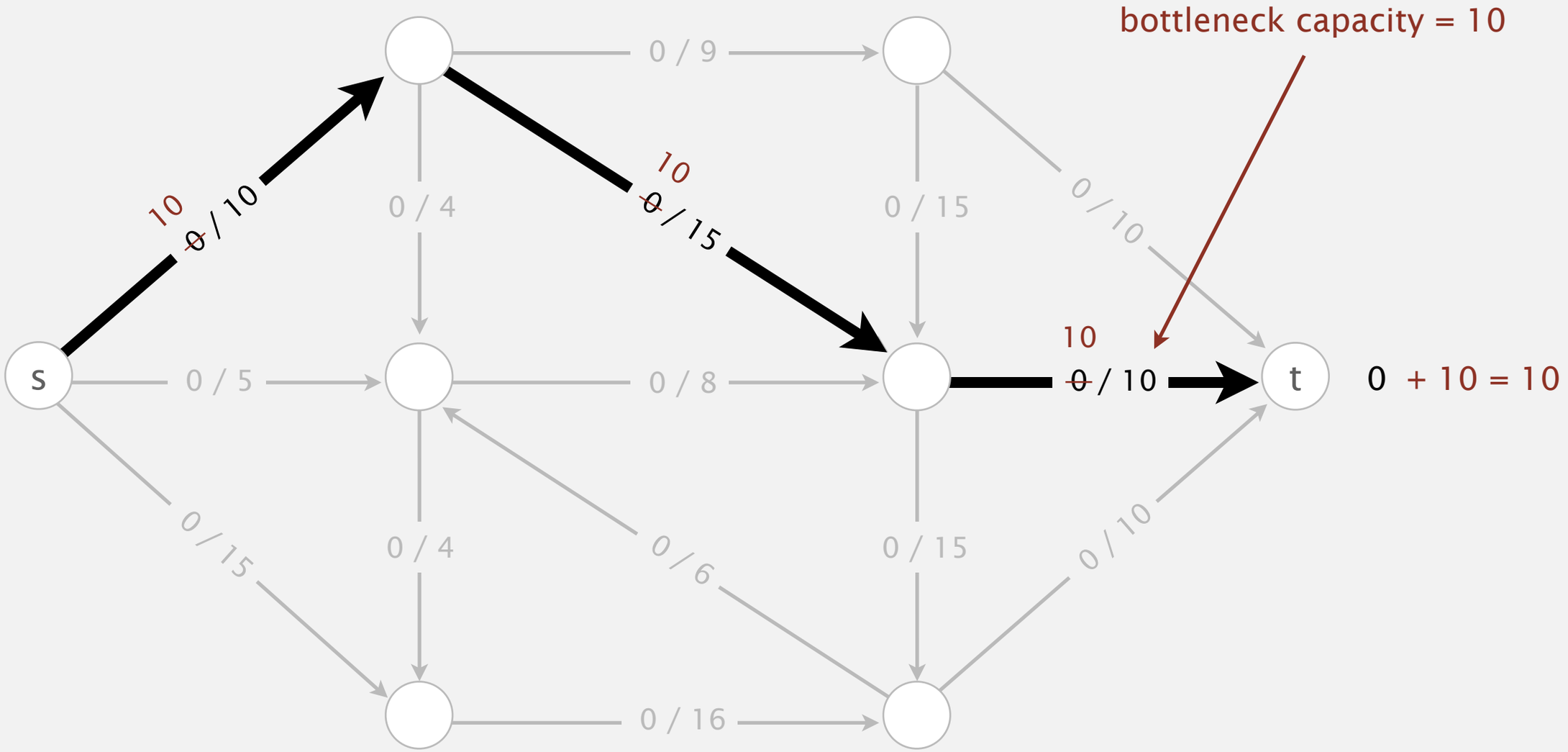


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

1st augmenting path

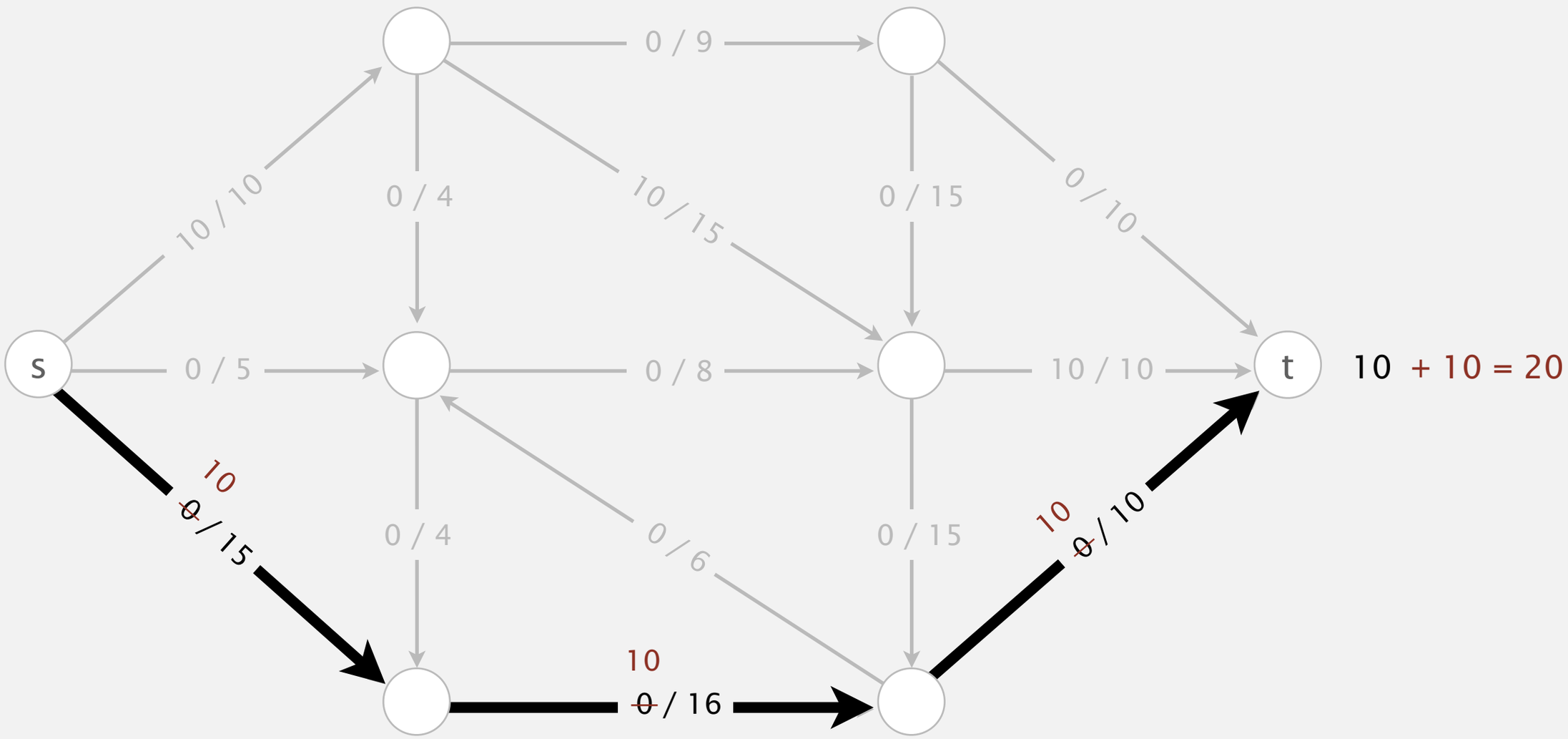


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

2nd augmenting path

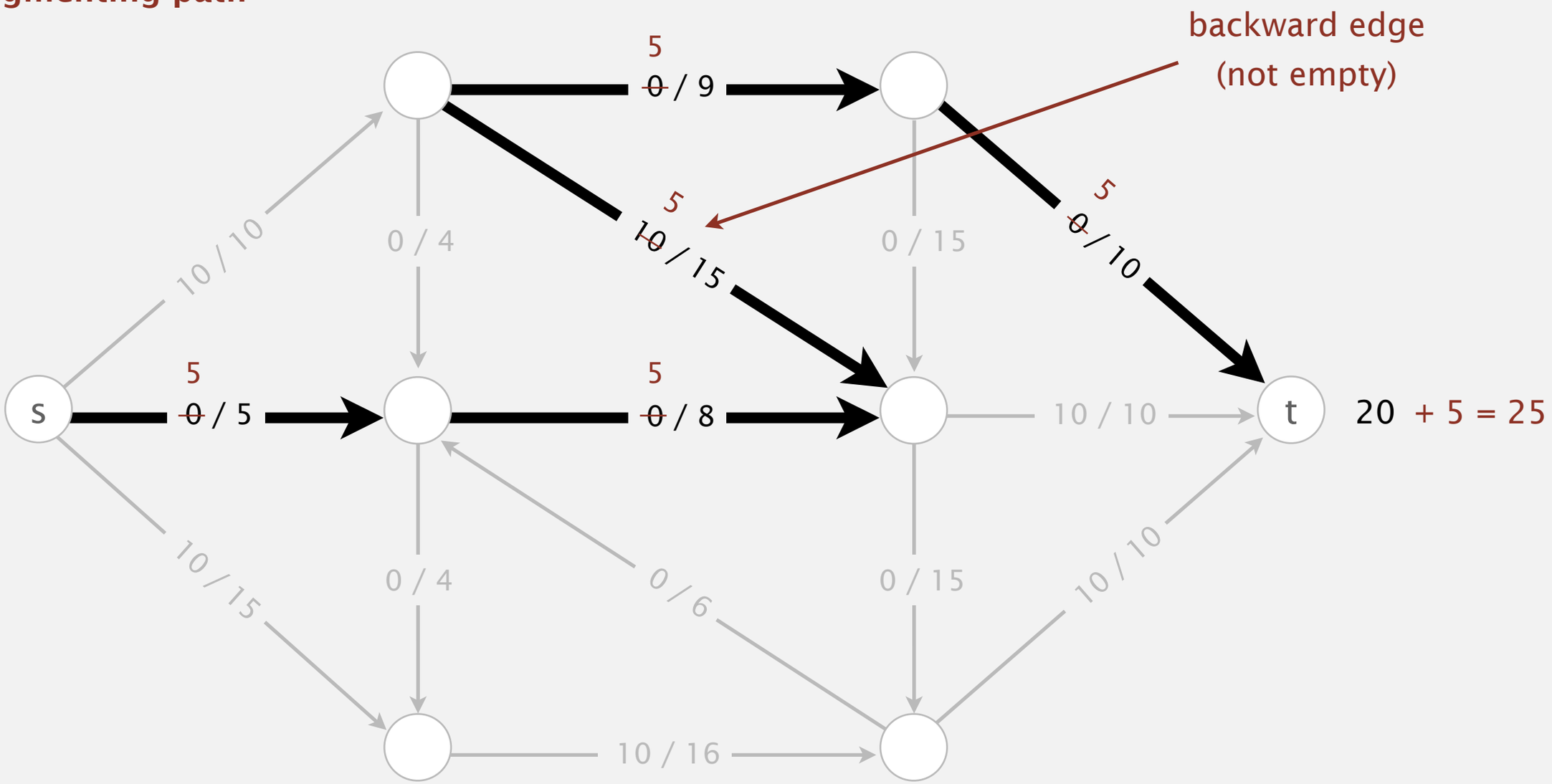


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

3rd augmenting path

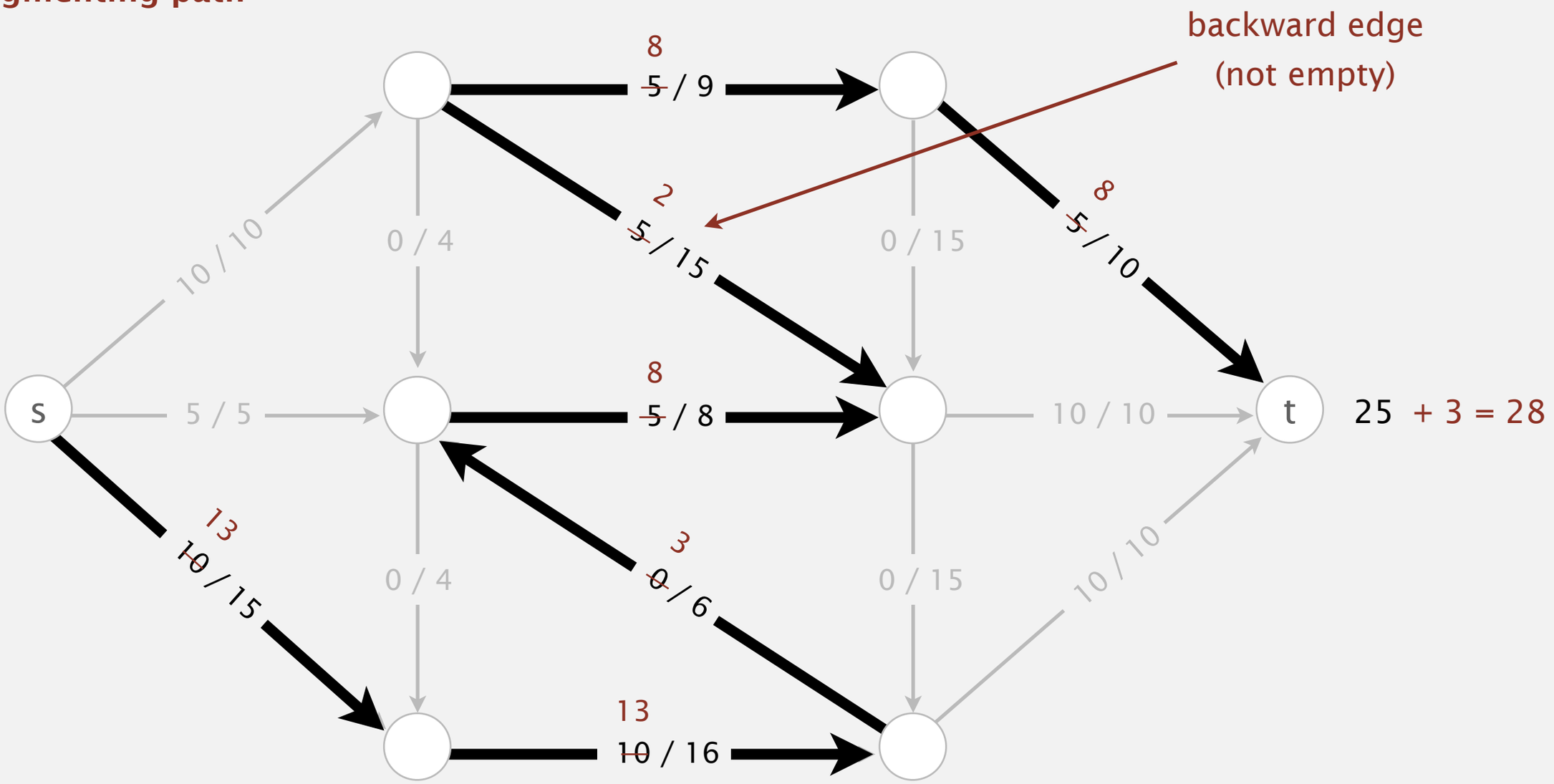


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

4th augmenting path

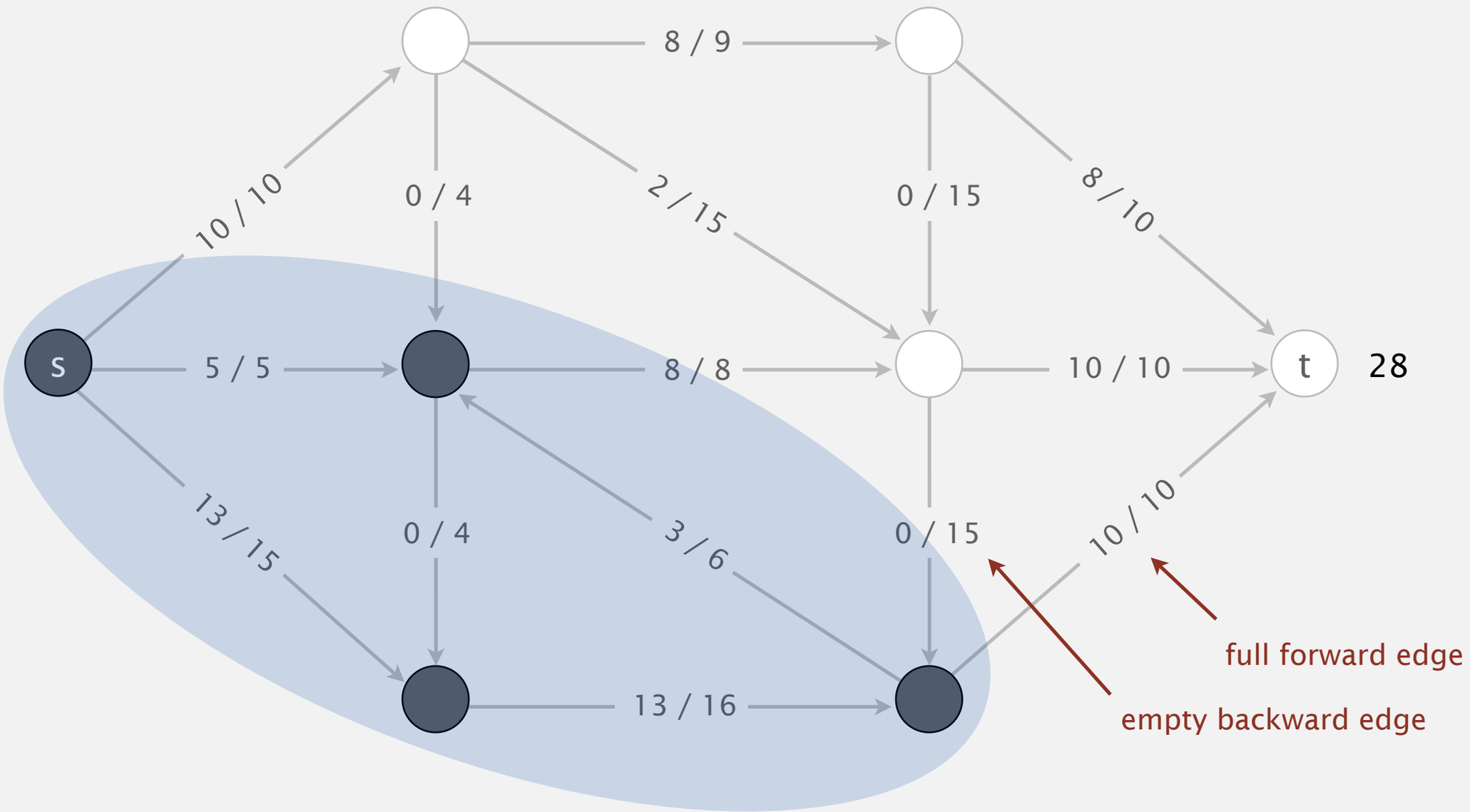


Idea: increase flow along augmenting paths

Termination. All paths from s to t are blocked by either a

- Full forward edge.
- Empty backward edge.

no more augmenting paths



Ford-Fulkerson algorithm

Ford-Fulkerson algorithm

Start with 0 flow.

While there exists an augmenting path:

- **find an augmenting path**
- **compute bottleneck capacity**
- **increase flow on that path by bottleneck capacity**

Questions.

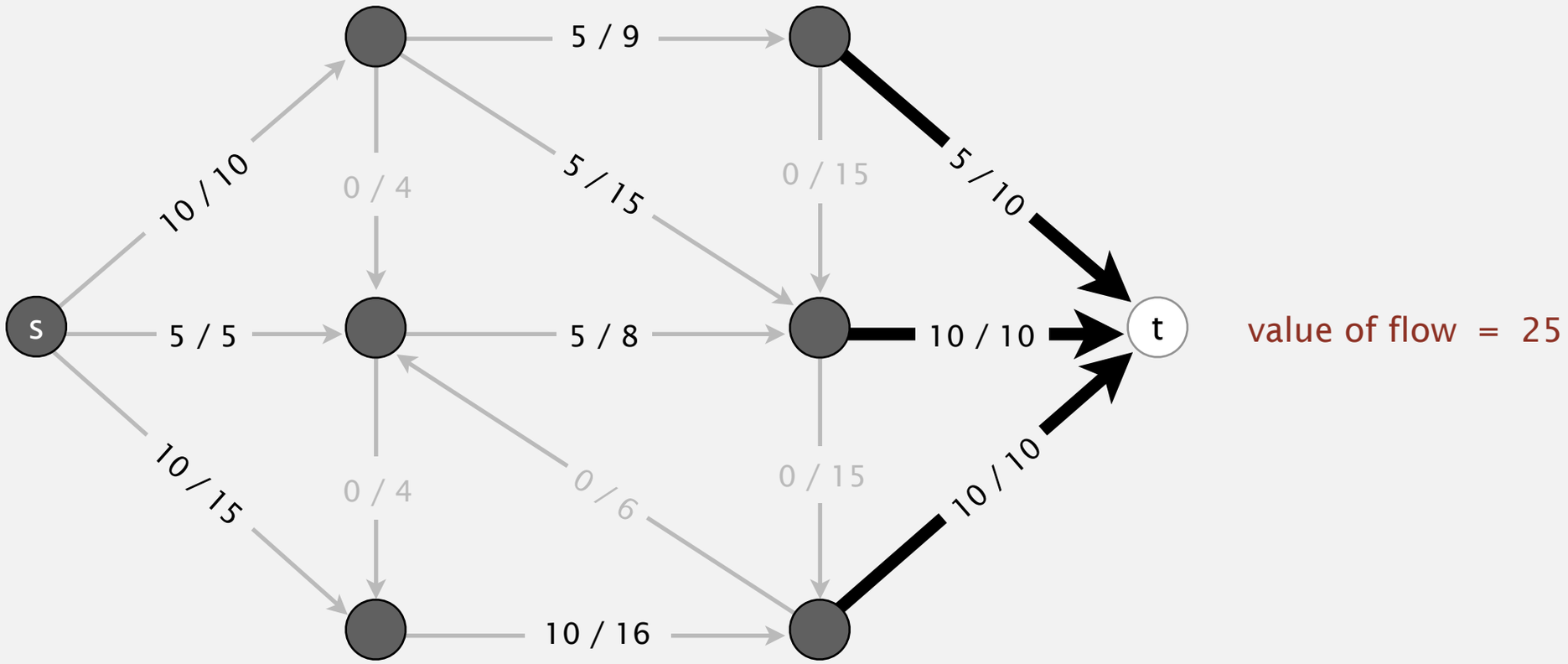
- How to find an augmenting path?
- How to compute a mincut?
- If FF terminates, does it always compute a maxflow?
- Does FF always terminate? If so, after how many augmentations?

Relationship between flows and cuts

Def. The net flow across a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from from B to A.

Proposition. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.

net flow across cut = 5 + 10 + 10 = 25

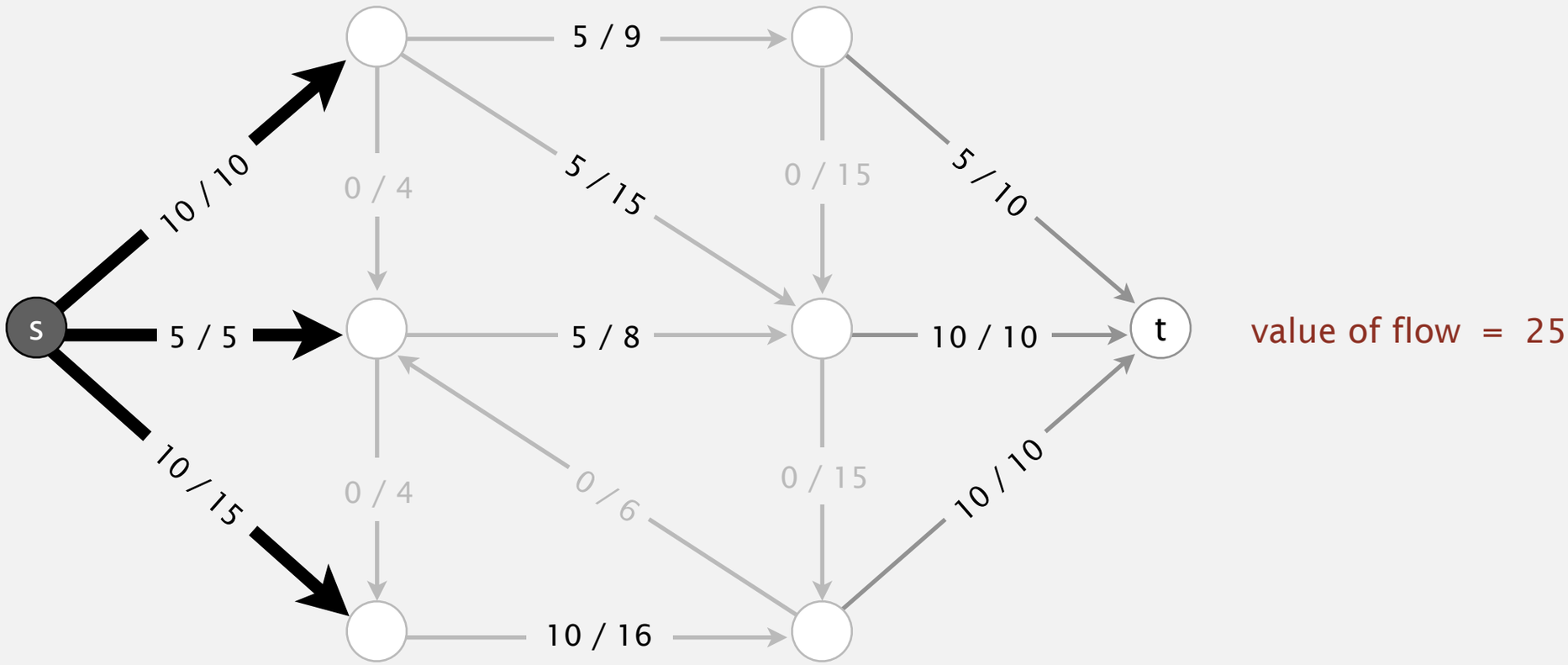


Relationship between flows and cuts

Def. The net flow across a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from from B to A.

Proposition. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.

net flow across cut = 10 + 5 + 10 = 25



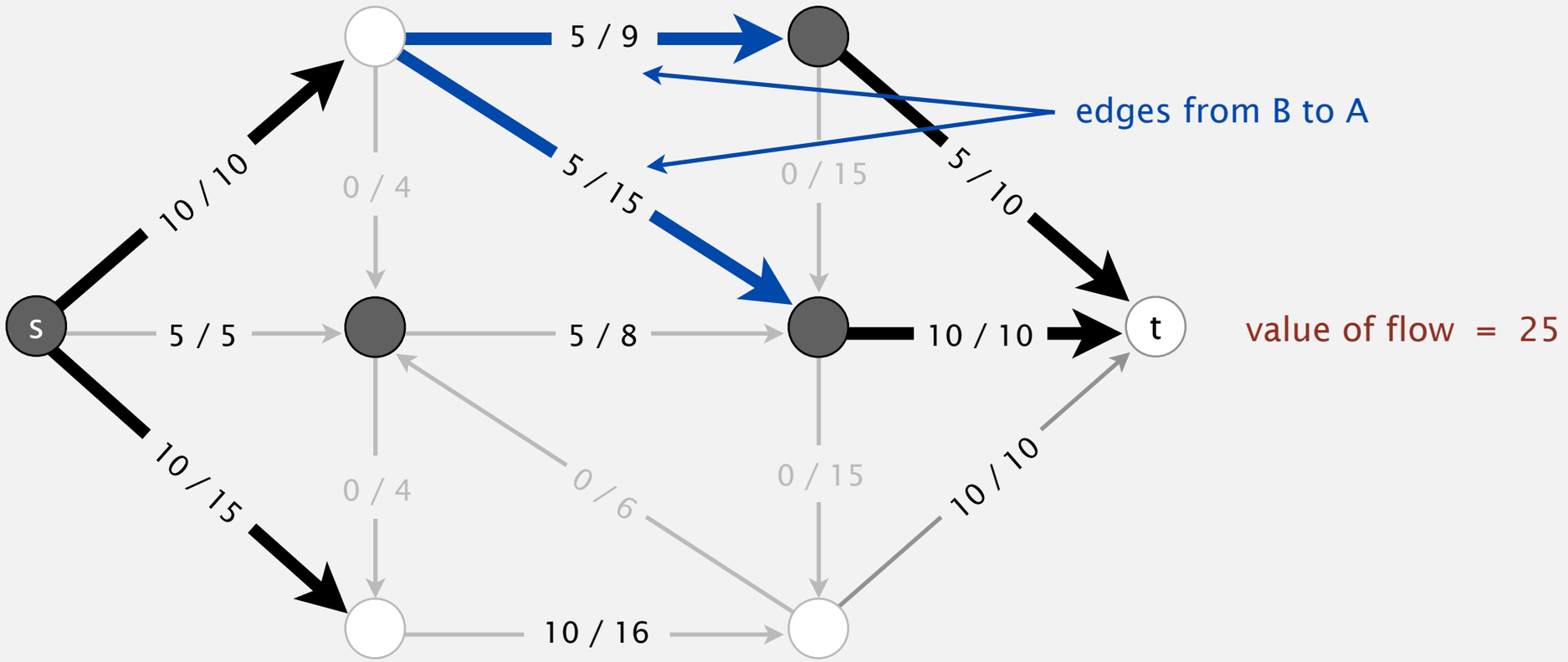
Relationship between flows and cuts

Def. The net flow across a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from from B to A.

Proposition. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.

Corollary. Outflow from s = inflow to t = value of flow.

net flow across cut = (10 + 10 + 5 + 10) – (5 + 5) = 25



Relationship between flows and cuts

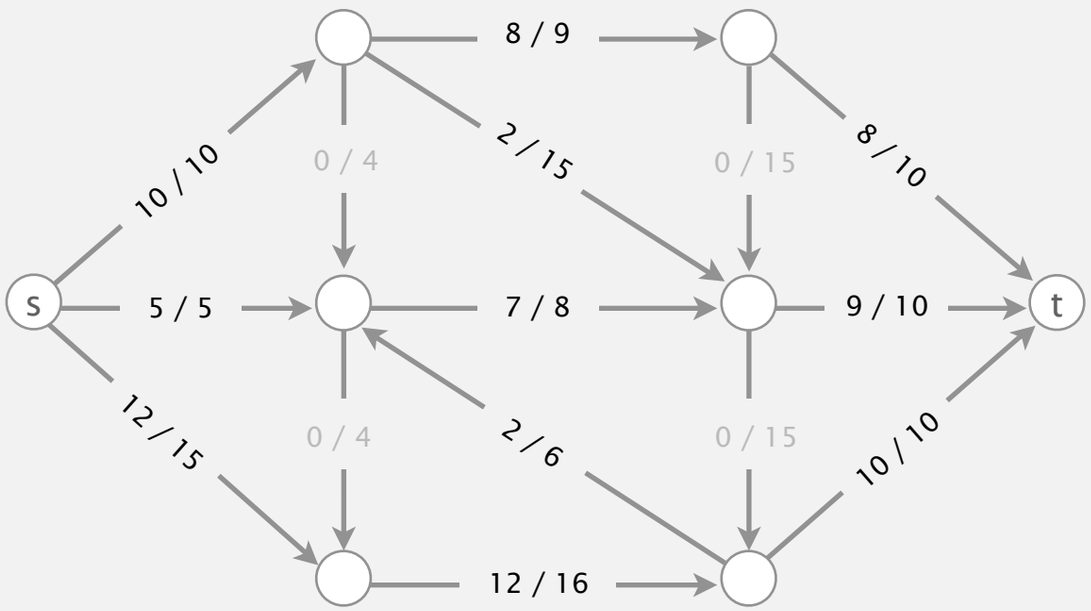
Weak duality. Let f be any flow and let (A, B) be any cut.

Then, the value of the flow ≤ the capacity of the cut.

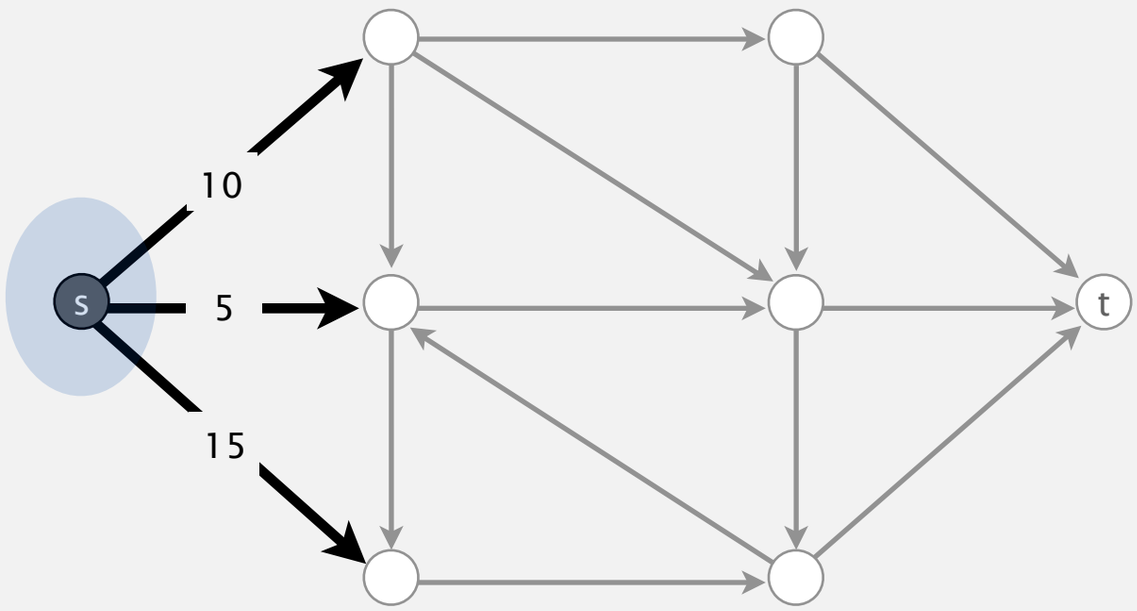
Pf. Value of flow f = net flow across cut (A, B) ≤ capacity of cut (A, B).

↑ flow value lemma

↑ flow bounded by capacity



value of flow = 27



capacity of cut = 30

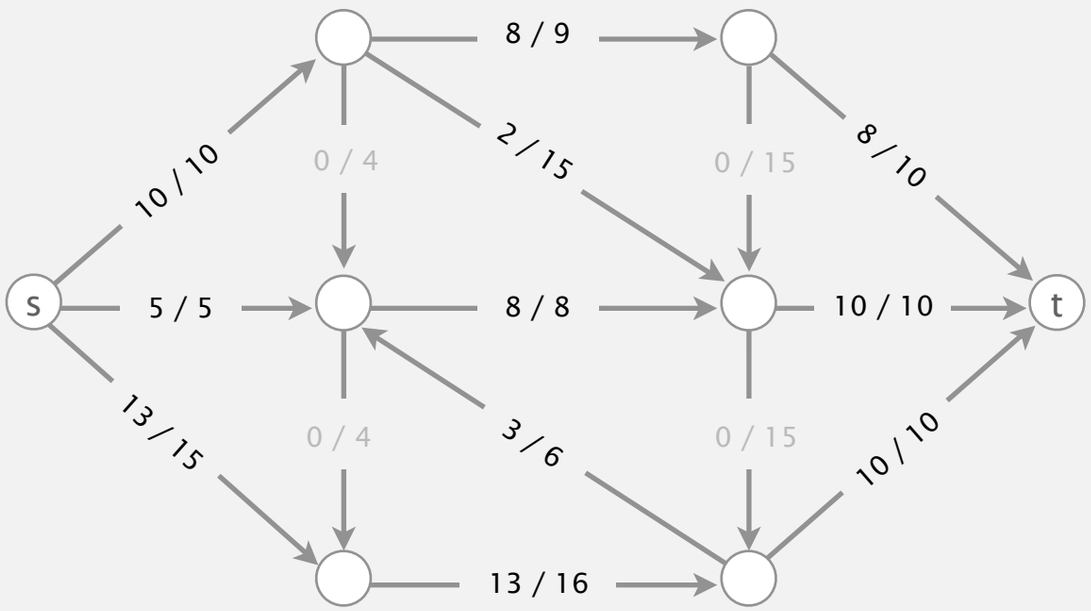
Relationship between flows and cuts

Certificate of optimality. Let f be a flow and let (A, B) be a cut such that the value of the flow f = the capacity of the cut (A, B). Then, f is a maxflow and (A, B) is a mincut.

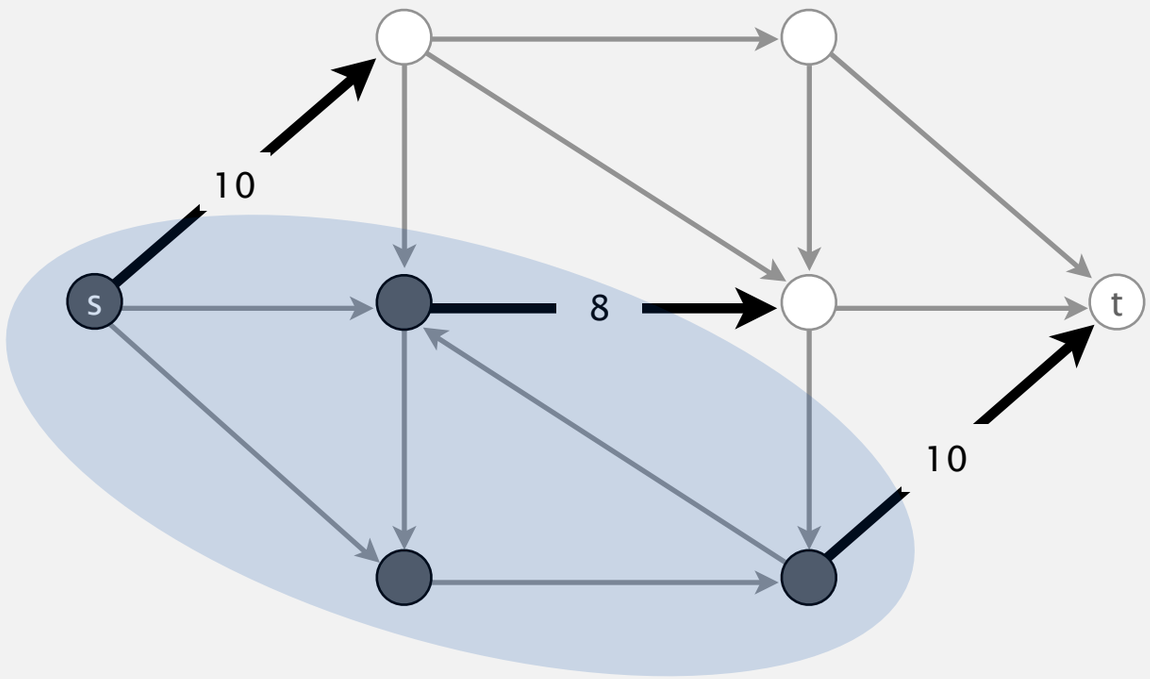
Pf.

weak duality

- For any flow f', value of flow f' $\leq$ capacity of (A, B) = value of f.
- For any cut (A', B'), capacity of (A', B') $\geq$ value of f = capacity of (A, B).



value of flow = 28



capacity of cut = 28

Maxflow-mincut theorem

Augmenting path theorem. A flow f is a maxflow iff no augmenting paths.

Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

Pf. The following three conditions are equivalent for any flow f:

i. There exists a cut whose capacity equals the value of the flow f.

ii. f is a maxflow.

iii. There is no augmenting path with respect to f.

Maxflow-mincut theorem

Augmenting path theorem. A flow f is a maxflow iff no augmenting paths.

Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

Pf. The following three conditions are equivalent for any flow f:

i. There exists a cut whose capacity equals the value of the flow f.

ii. f is a maxflow.

iii. There is no augmenting path with respect to f.

[i $\Rightarrow$ ii]

- Suppose that (A, B) is a cut with capacity equal to the value of f.
- Then, the value of any flow f' $\leq$ capacity of (A, B) = value of f.
- Thus, f is a maxflow.

weak duality

Maxflow-mincut theorem

Augmenting path theorem. A flow f is a maxflow iff no augmenting paths.

Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

Pf. The following three conditions are equivalent for any flow f:

i. There exists a cut whose capacity equals the value of the flow f.

ii. f is a maxflow.

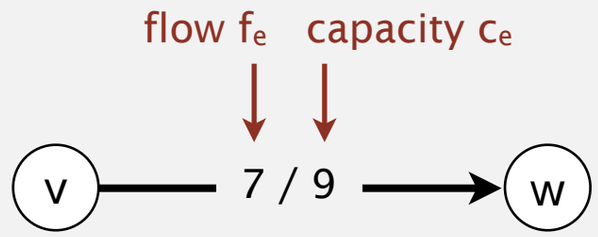
iii. There is no augmenting path with respect to f.

[ii $\Rightarrow$ iii] We prove contrapositive: ~iii $\Rightarrow$ ~ii.

- Suppose that there is an augmenting path with respect to f.
- Can improve flow f by sending flow along this path.
- Thus, f is not a maxflow.

Flow network representation

Flow edge data type. Associate flow f_e and capacity c_e with edge $e = v \rightarrow w$.



Flow network data type. Need to process edge $e = v \rightarrow w$ in either direction:

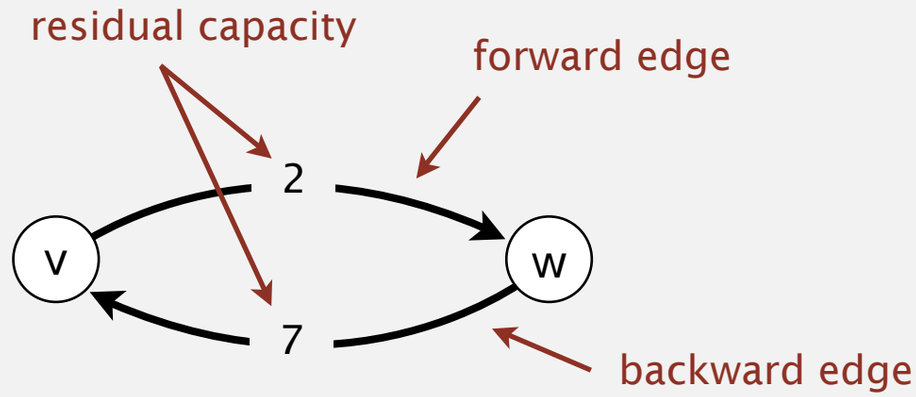
Include e in both v and w's adjacency lists.

Residual capacity.

- Forward edge: residual capacity $= c_e - f_e$.
- Backward edge: residual capacity $= f_e$.

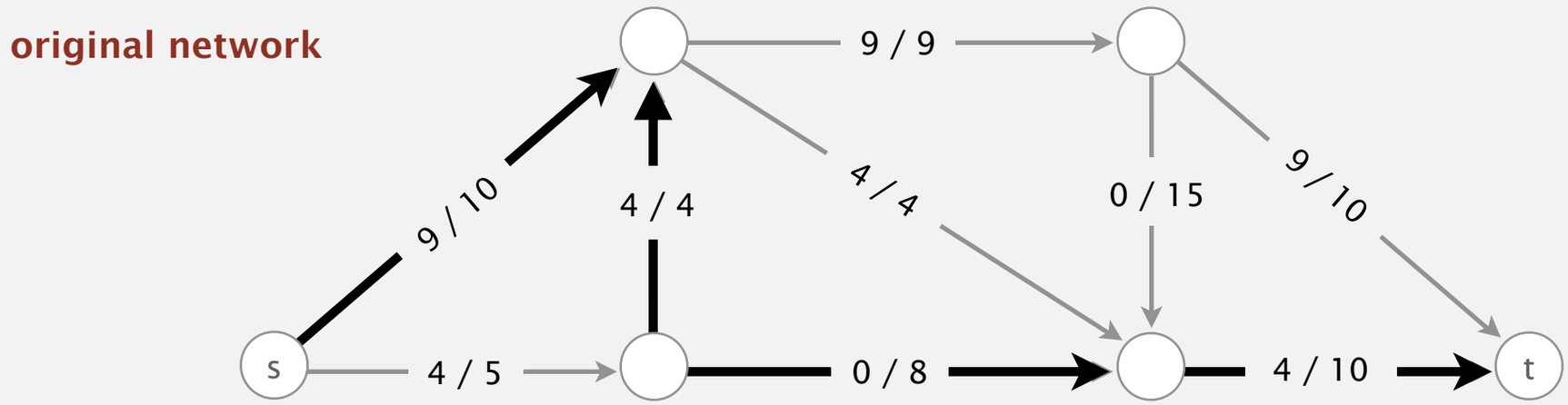
Augment flow.

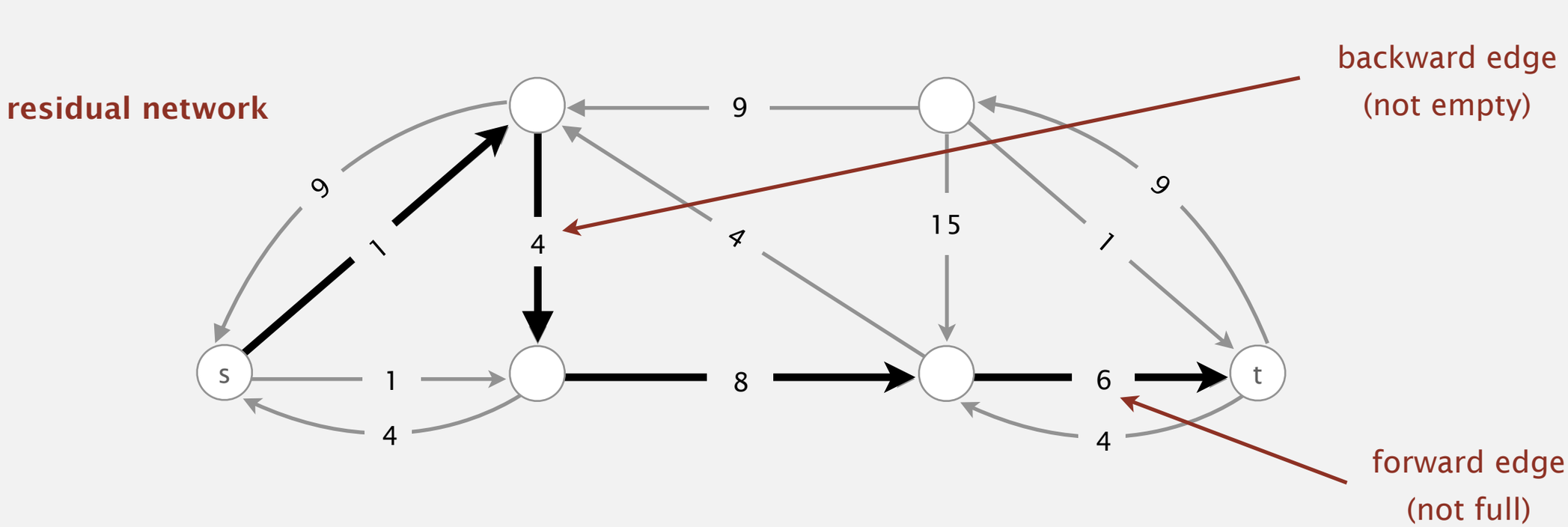
- Forward edge: add Δ.
- Backward edge: subtract Δ.



Flow network representation

Residual network. A useful view of a flow network.





Key point. Augmenting path in original network is equivalent to directed path in residual network.

How to choose augmenting paths?

Shortest path. Use BFS.

DFS path. Use DFS.

Fattest path. Use a PQ, ala Dijkstra.

Random path. Use a randomized queue.

augmenting path with fewest number of edges

shortest path

All easy to implement.

- Define "residual graph."
- Find paths in residual graph.

augmenting path with maximum bottleneck capacity

fattest path

Performance depends on network properties.

- How many augmenting paths?
- How many edges examined to find each augmenting path?
- Caveat: generic FF can converge to wrong value if capacities are irrational.

How to choose augmenting paths?

FF performance depends on choice of augmenting paths.

augmenting path	number of paths	implementation
shortest path	$\leq \frac{1}{2} E V$	queue (BFS)
fattest path	$\leq E \ln(E U)$	priority queue
random path	$\leq E U$	randomized queue
DFS path	$\leq E U$	stack (DFS)

digraph with V vertices, E edges, and integer capacities (max U)

augmenting path with fewest number of edges →

shortest path

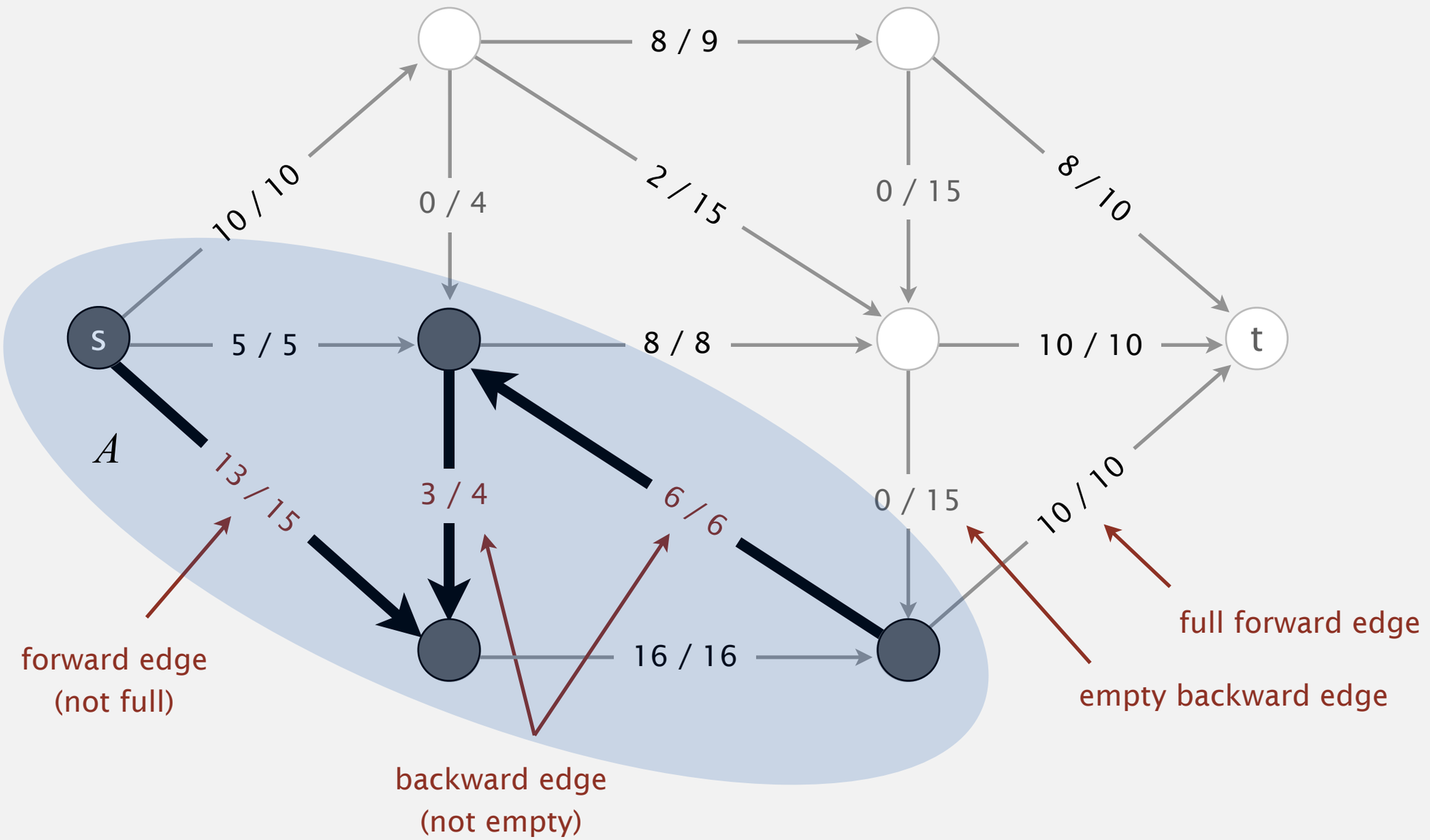
augmenting path with maximum bottleneck capacity →

fattest path

Computing a mincut from a maxflow

To compute mincut (A, B) from maxflow f:

- By augmenting path theorem, no augmenting paths with respect to f.
- Compute A = set of vertices connected to s by an undirected path with no full forward or empty backward edges.



Ford-Fulkerson algorithm

Ford-Fulkerson algorithm

Start with 0 flow.

While there exists an augmenting path:

- **find an augmenting path**
- **compute bottleneck capacity**
- **increase flow on that path by bottleneck capacity**

Questions.

- How to find an augmenting path? BFS works well.
- How to compute a mincut? Easy. ✔
- If FF terminates, does it always compute a maxflow? Yes. ✔
- Does FF always terminate? If so, after how many augmentations?

yes, provided edge capacities are integers
(or augmenting paths are chosen carefully)

not so easy analysis..

Ford-Fulkerson algorithm with integer capacities

Important special case. Edge capacities are integers between 1 and U.

flow on each edge is an integer

Invariant. The flow is integer-valued throughout FF.

Pf. [by induction]

- Bottleneck capacity is an integer.
- Flow on an edge increases/decreases by bottleneck capacity.

Proposition. Number of augmentations ≤ the value of the maxflow.

Pf. Each augmentation increases the value by at least 1.

and FF finds one!

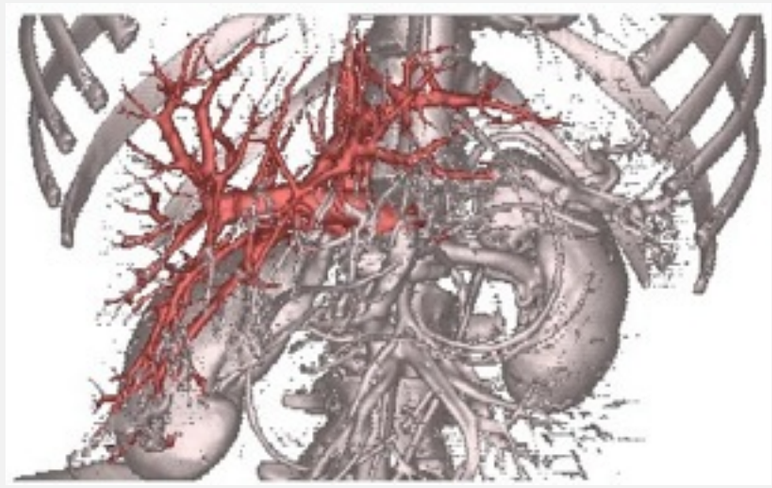
Integrality theorem. There exists an integer-valued maxflow.

Pf. FF terminates; invariant ensures maxflow that FF finds is integer-valued.

Maxflow and mincut applications

Maxflow/mincut is a widely applicable problem-solving model.

- Data mining.
- Open-pit mining.
- Bipartite matching.
- Network reliability.
- Image segmentation.
- Network connectivity.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Multi-camera scene reconstruction.
- Sensor placement for homeland security.
- Baseball elimination. ← see next programming assignment
- Many, many, more.



liver and hepatic vascularization segmentation

Bipartite matching problem

N students apply for N jobs.

Each gets several offers.

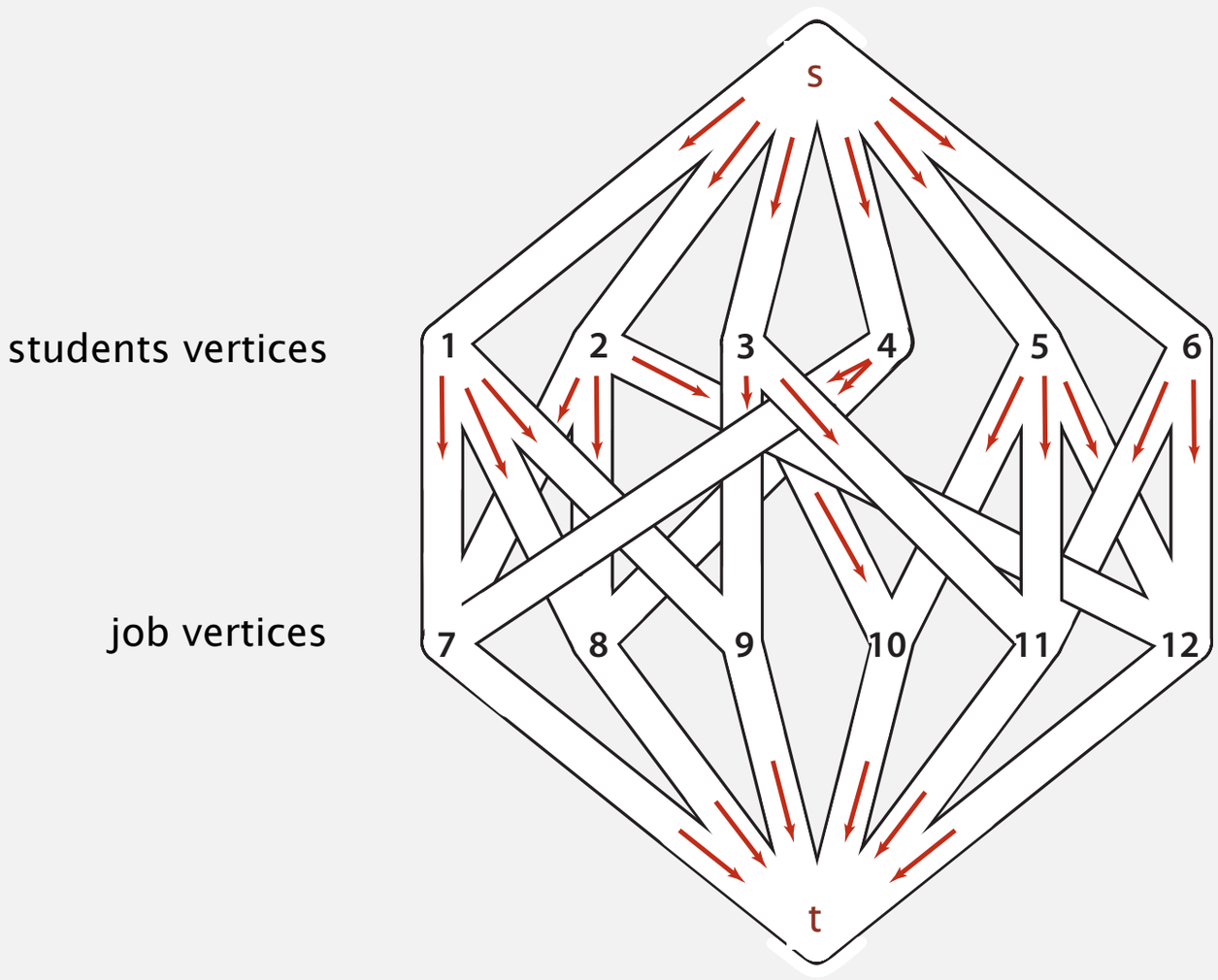
Is there a way to match all students to jobs?

bipartite matching problem

1	Alice	7	Adobe
	Adobe		Alice
	Amazon		Bob
	Facebook		Dave
2	Bob	8	Amazon
	Adobe		Alice
	Amazon		Bob
	Yahoo		Dave
3	Carol	9	Facebook
	Facebook		Alice
	Google		Carol
	IBM	10	Google
4	Dave		Carol
	Adobe		Eliza
	Amazon	11	IBM
5	Eliza		Carol
	Google		Eliza
	IBM		Frank
	Yahoo	12	Yahoo
6	Frank		Bob
	IBM		Eliza
	Yahoo		Frank

Network flow formulation of bipartite matching

- Create s, t, one vertex for each student, and one vertex for each job.
- Add edge from s to each student (capacity 1).
- Add edge from each job to t (capacity 1).
- Add edge from student to each job offered (infinite capacity).



bipartite matching problem

1 Alice
- Adobe
- Amazon
- Facebook

2 Bob
- Adobe
- Amazon
- Yahoo

3 Carol
- Facebook
- Google
- IBM

4 Dave
- Adobe
- Amazon

5 Eliza
- Google
- IBM
- Yahoo

6 Frank
- IBM
- Yahoo

7 Adobe
- Alice
- Bob
- Dave

8 Amazon
- Alice
- Bob
- Dave

9 Facebook
- Alice
- Carol

10 Google
- Carol
- Eliza

11 IBM
- Carol
- Eliza
- Frank

12 Yahoo
- Bob
- Eliza
- Frank

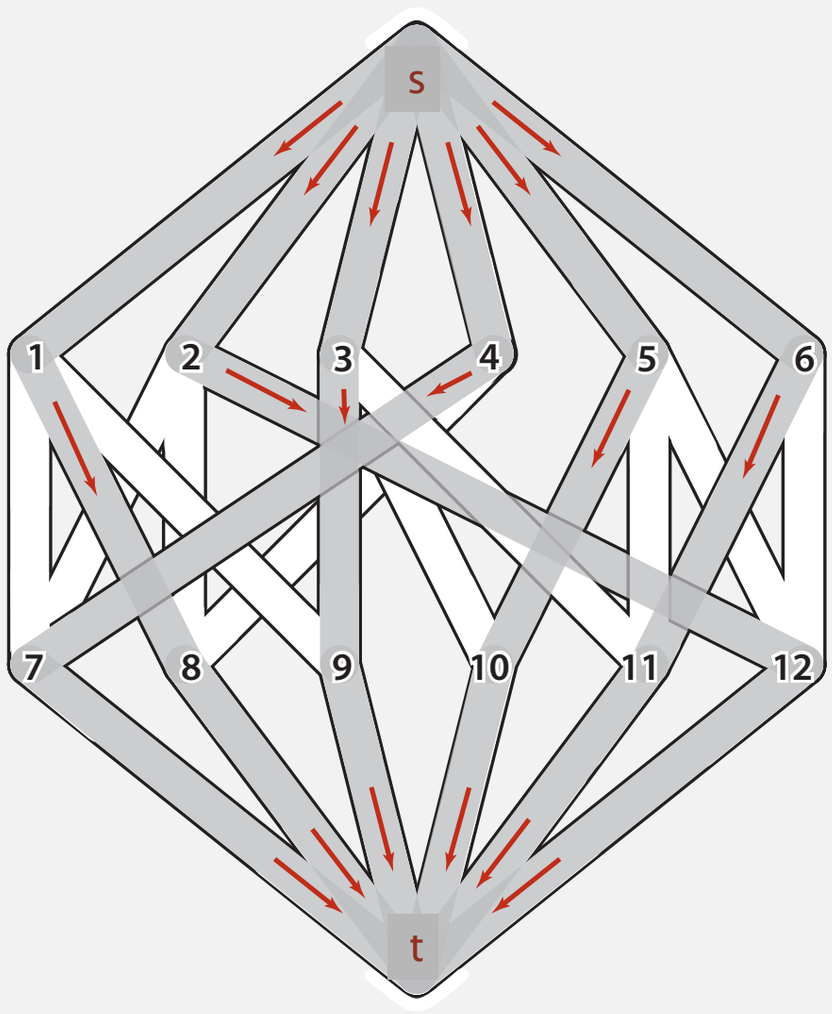
Network flow formulation of bipartite matching

1-1 correspondence between integer-valued maxflow solution of value N and perfect matchings.

perfect matching (solution)

Alice — Amazon
Bob — Yahoo
Carol — Facebook
Dave — Adobe
Eliza — Google
Frank — IBM

maximum flow



bipartite matching problem

1 Alice
Adobe
Amazon
Facebook
2 Bob
Adobe
Amazon
Yahoo
3 Carol
Facebook
Google
IBM
4 Dave
Adobe
Amazon
5 Eliza
Google
IBM
Yahoo
6 Frank
IBM
Yahoo
7 Adobe
Alice
Bob
Dave
8 Amazon
Alice
Bob
Dave
9 Facebook
Alice
Carol
10 Google
Carol
Eliza
11 IBM
Carol
Eliza
Frank
12 Yahoo
Bob
Eliza
Frank

Summary

Mincut problem. Find an st-cut of minimum capacity.

Maxflow problem. Find an st-flow of maximum value.

Duality. Value of the maxflow = capacity of mincut.

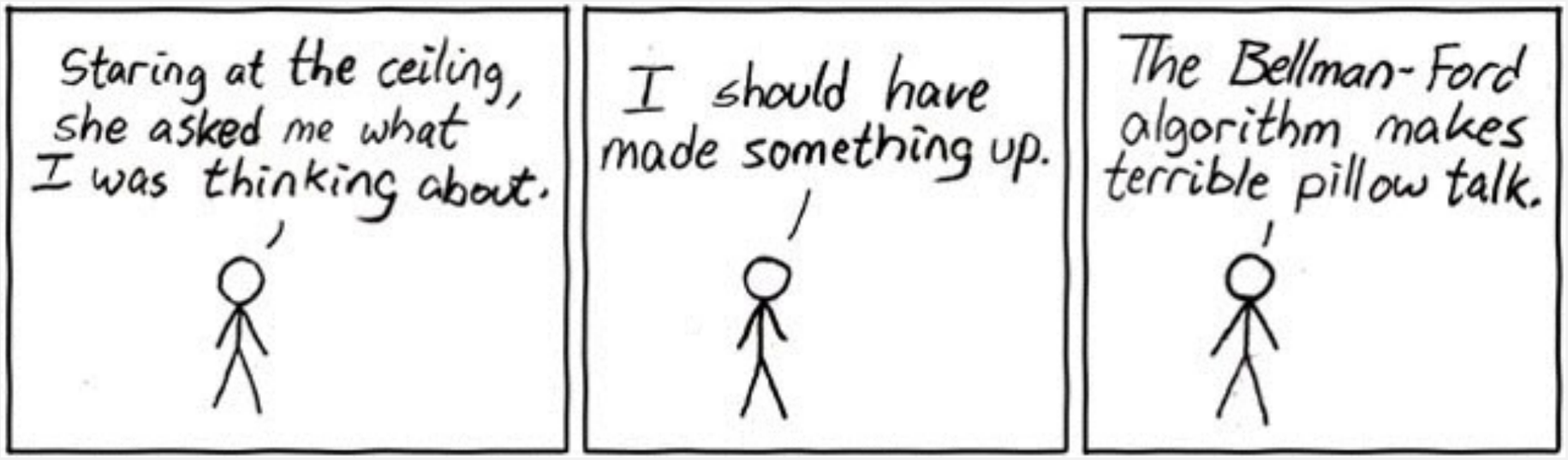
Proven successful approaches.

- Ford-Fulkerson (various augmenting-path strategies).
- Preflow-push (various versions).

Open research challenges.

- Practice: solve real-word maxflow/mincut problems in linear time.
- Theory: prove it for worst-case inputs.
- Still much to be learned!

Its all about which algorithm you choose... :)



Good Luck!