CS/COE 1501
Algorithm Implementation

5-Graph Algorithms

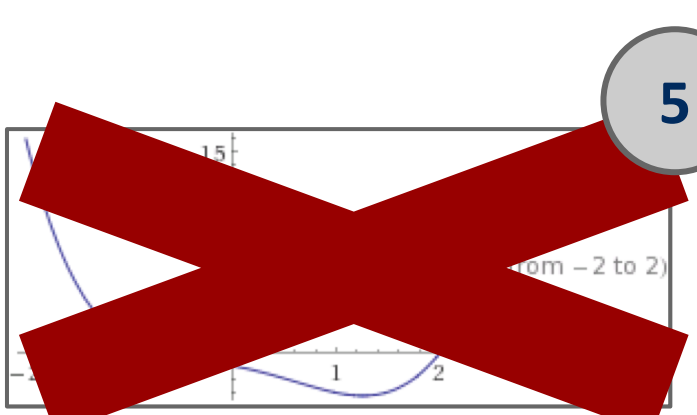
Fall 2018

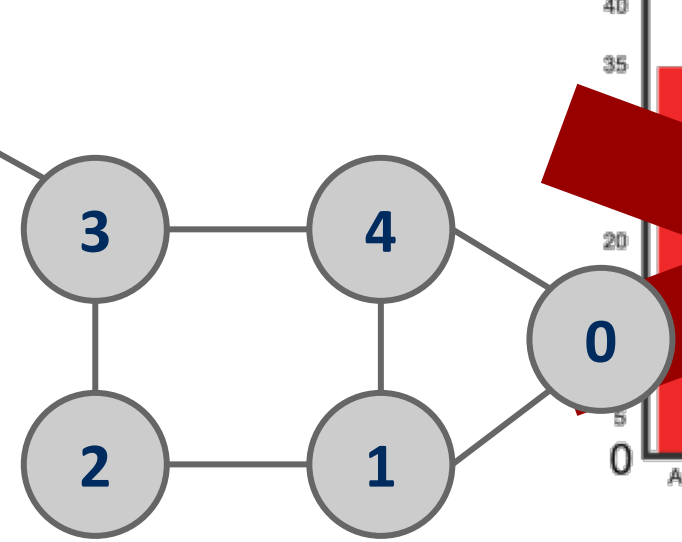
Sherif Khattab

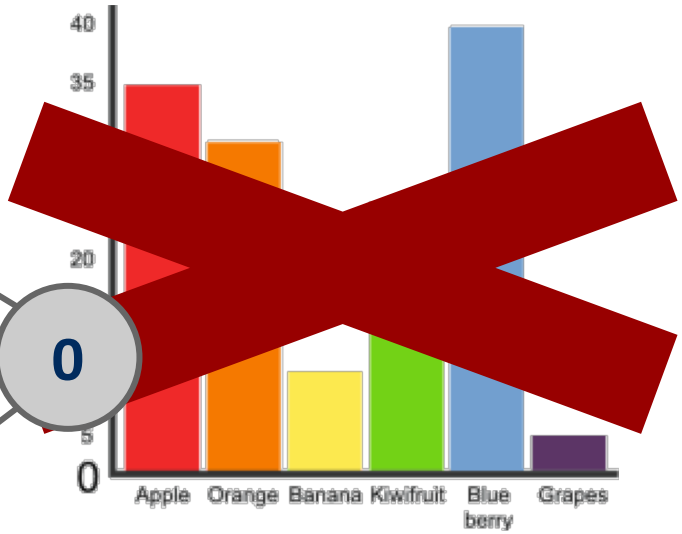
ksm73@pitt.edu

6307 Sennott Square

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

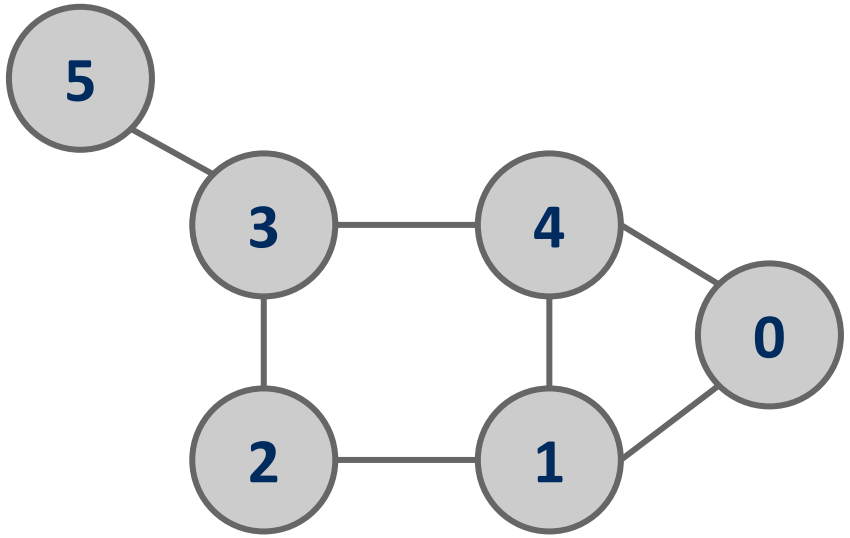

15
1
2


5
3
4
0
2
1


40
35
20
5
0
Apple
Orange
Banana
Kiwifruit
Blue berry
Grapes

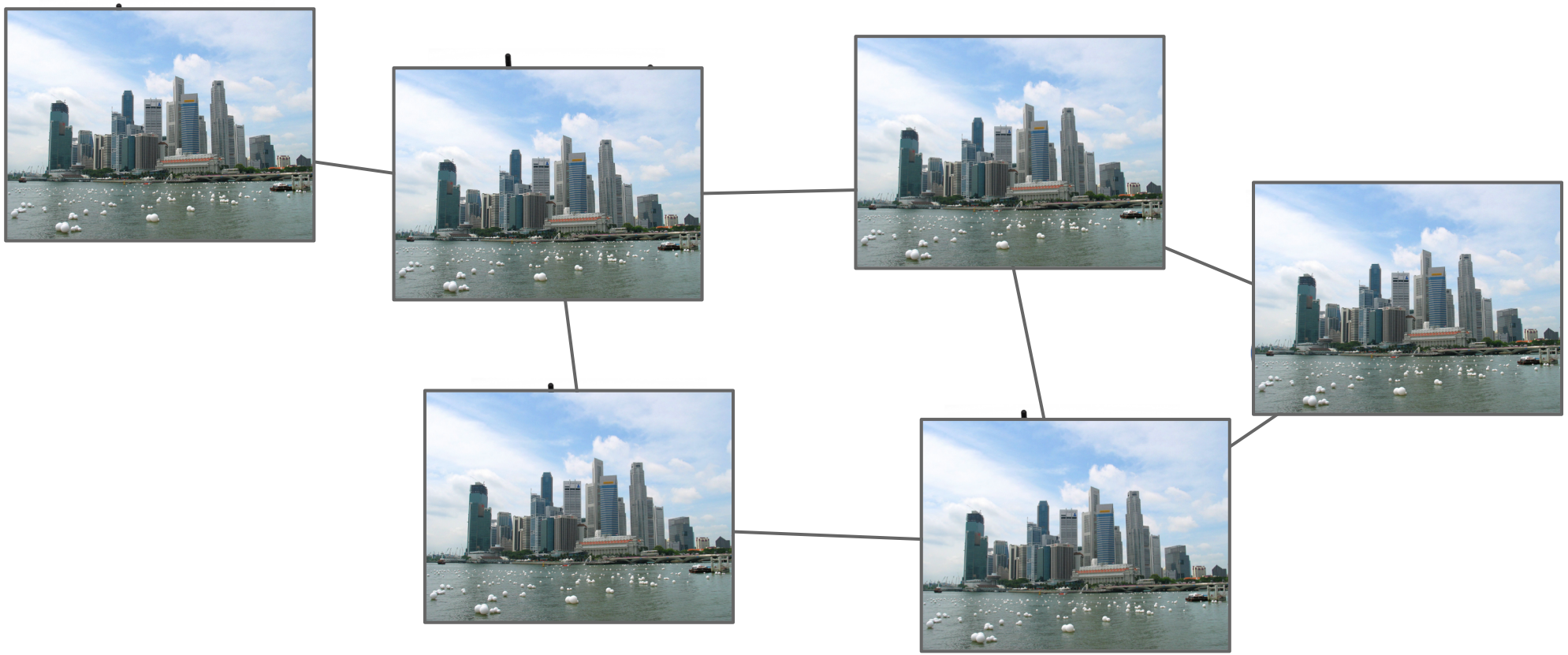
Graphs

- A graph G = (V, E)
  - Where V is a set of vertices
  - E is a set of edges connecting vertex pairs
- Example:
  - V = {0, 1, 2, 3, 4, 5}
  - E = {(0, 1), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4), (3, 5)}



Why?

- Can be used to model many different scenarios



Some definitions

- Undirected graph
  - Edges are unordered pairs: (A, B) == (B, A)
- Directed graph
  - Edges are ordered pairs: (A, B) != (B, A)
- Adjacent vertices, or neighbors
  - Vertices connected by an edge

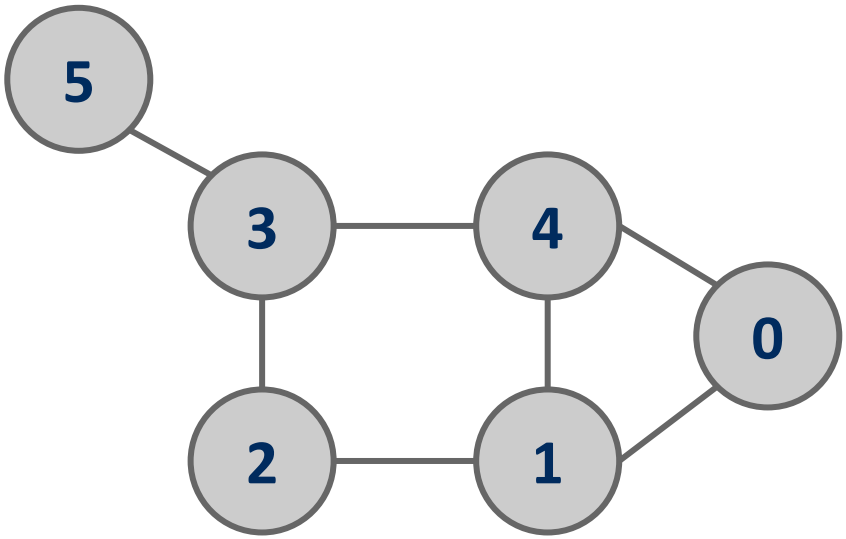
Graph sizes

- Let v = |V|, and e = |E|
- Given v, what are the minimum/maximum sizes of e?
  - Minimum value of e?
    - Definition doesn’t necessitate that there are any edges…
    - So, 0
  - Maximum of e?
    - Depends…
      - Are self edges allowed?
      - Directed graph or undirected graph?
    - In this class, we'll assume directed graphs have self edges while undirected graphs do not

More definitions

- A graph is considered *sparse* if:
  - e <= v lg v
- A graph is considered *dense* as it approaches the maximum number of edges
  - I.e., e == MAX - ε
- A *complete* graph has the maximum number of edges

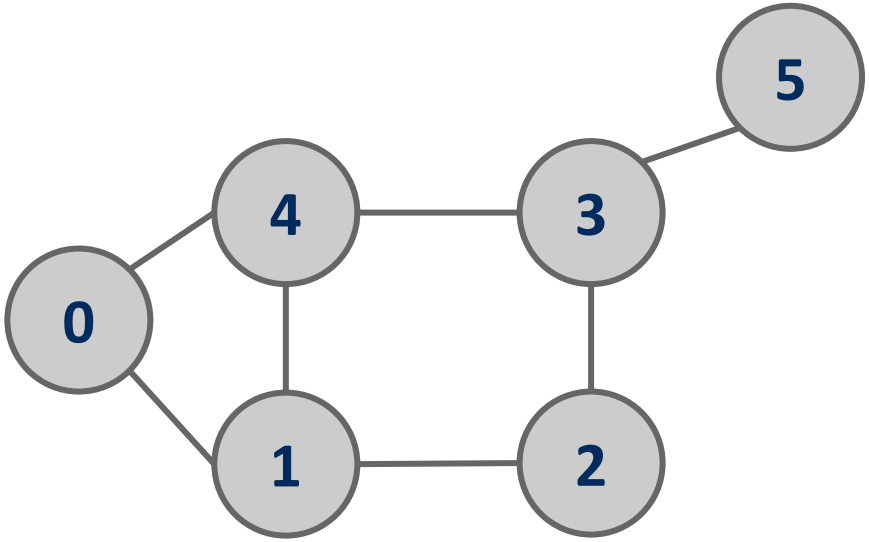
Question:



==

or

!=



- ?

Representing graphs

- Trivially, graphs can be represented as:
  - List of vertices
  - List of edges
- Performance?
  - Assume we're going to be analyzing static graphs
    - I.e., no insert and remove
  - So what operations should we consider?

Using an adjacency matrix

- Rows/columns are vertex labels
  - M[i][j] = 1 if (i, j) ∈ E
  - M[i][j] = 0 if (i, j) ∉ E

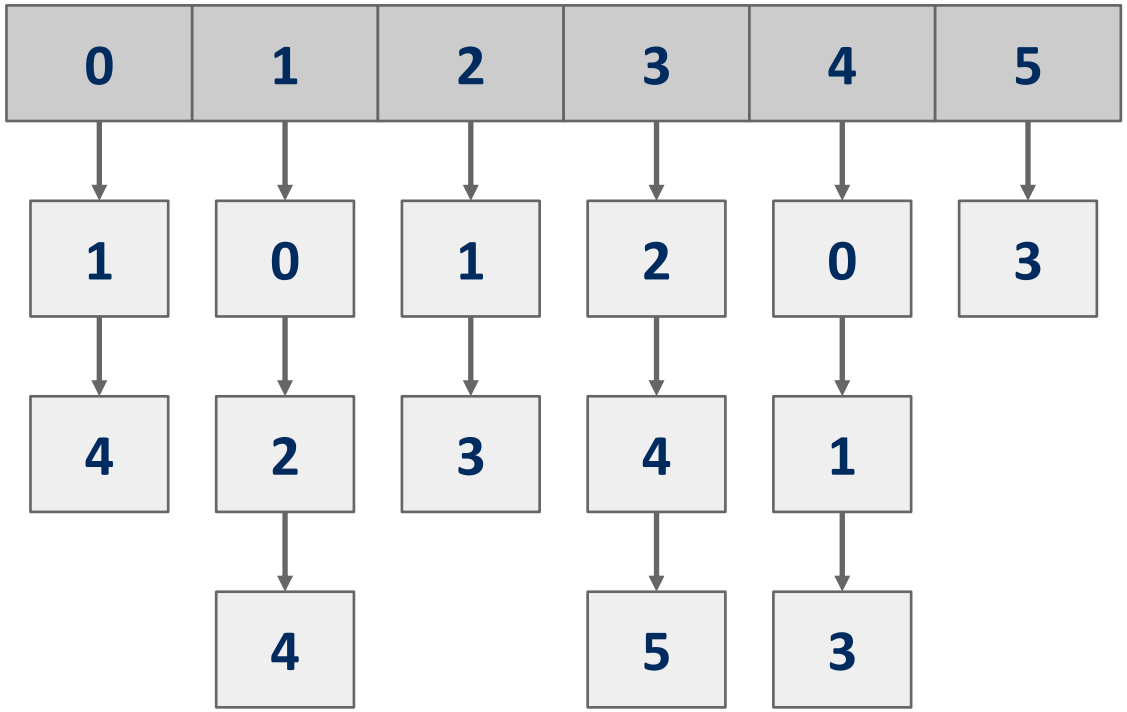
	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0

Adjacency matrix analysis

- Runtime?
- Space?

Adjacency lists

- Array of neighbor lists
  - A[i] contains a list of the neighbors of vertex i



Adjacency list analysis

- Runtime?
- Space?

Comparison

- Where would we want to use adjacency lists vs adjacency matrices?
  - What about the list of vertices/list of edges approach?

Even more definitions

- Path
  - A sequence of adjacent vertices
- Simple Path
  - A path in which no vertices are repeated
- Simple Cycle
  - A simple path with the same first and last vertex
- Connected Graph
  - A graph in which a path exists between all vertex pairs
- Connected Component
  - Connected subgraph of a graph
- Acyclic Graph
  - A graph with no cycles
- Tree
  - ?
  - A connected, acyclic graph
    - Has exactly v-1 edges

Graph traversal

- What is the best order to traverse a graph?
- Two primary approaches:
  - Depth-first search (DFS)
    - "Dive" as deep as possible into the graph first
    - Branch when necessary
  - Breadth-first search (BFS)
    - Search all directions evenly
      - I.e., from i, visit all of i's neighbors, then all of their neighbors, etc.

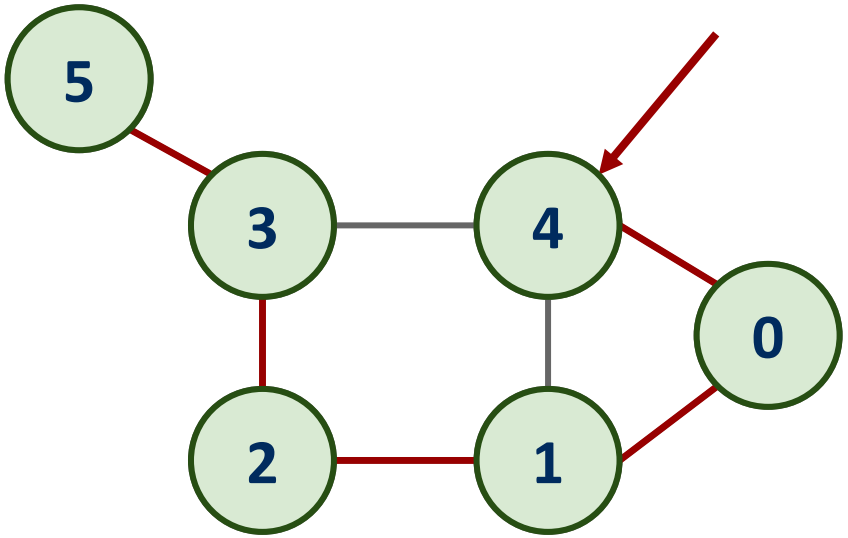
DFS

- Already seen and used this throughout the term
  - For tries…
  - For Huffman encoding…
- Can be easily implemented recursively
  - For each vertex, visit first unseen neighbor
  - Backtrack at deadends (i.e., vertices with no unseen neighbors)
    - Try next unseen neighbor after backtracking

Runtime: theta(v+e)

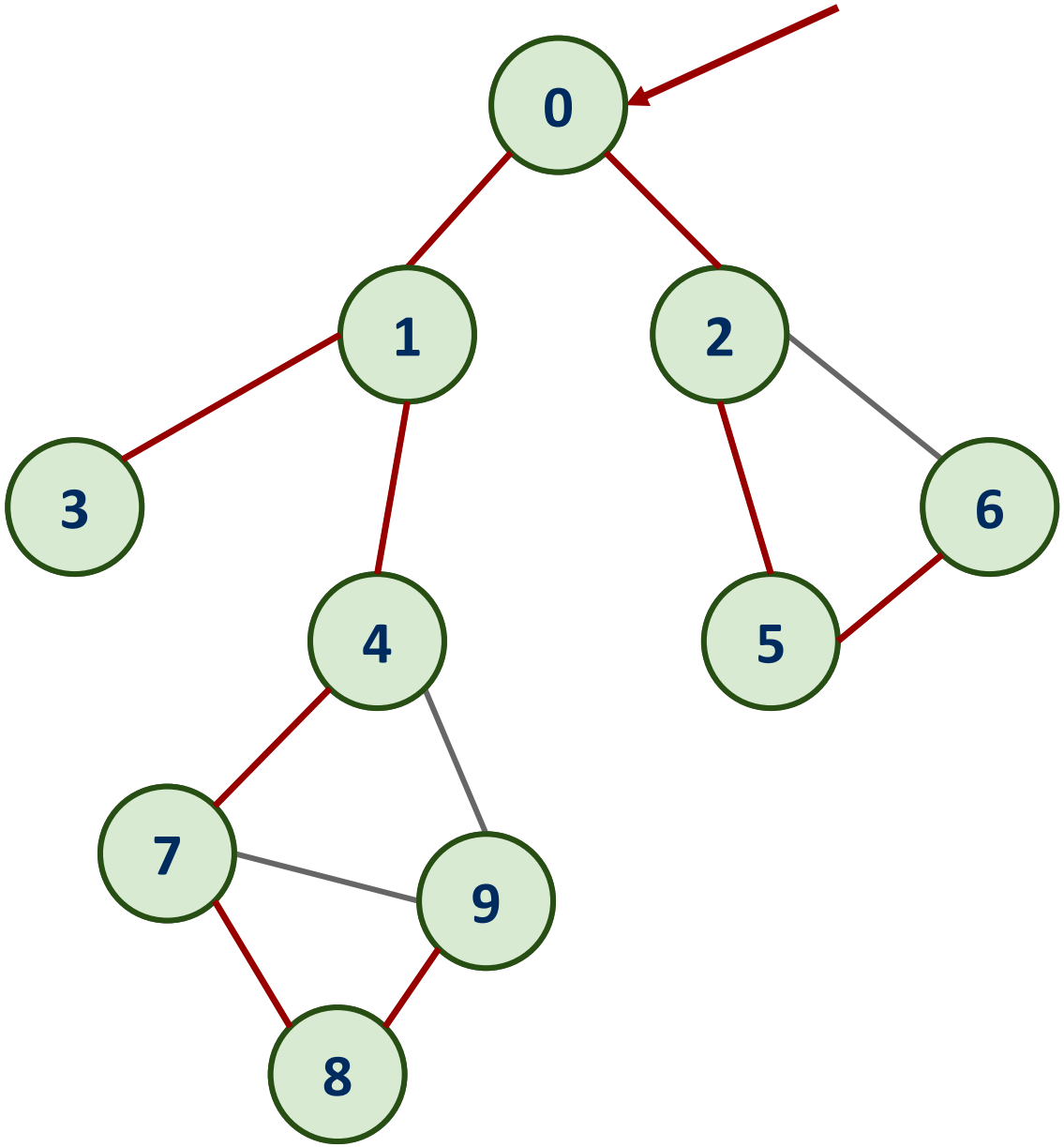
v: vertices
e: edges

DFS example



DFS example 2

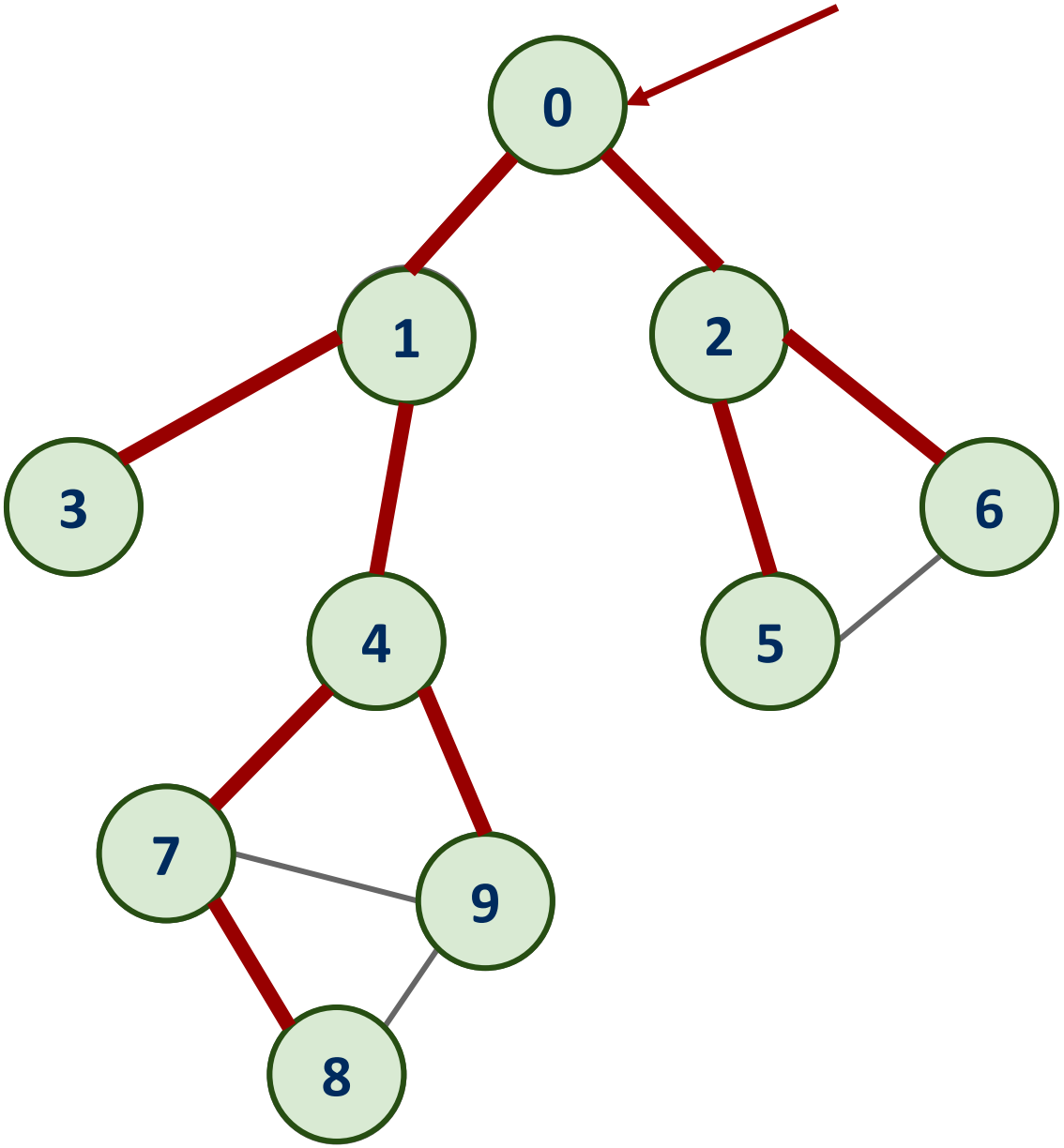
Spanning Tree:



BFS

- Can be easily implemented using a queue
  - For each vertex visited, add all of its neighbors to the queue
    - Vertices that have been seen but not yet visited are said to be the *fringe*
  - Pop head of the queue to be the next visited vertex
- See example

BFS example



Q

1 7 2 9 3 8 4 5 6

Shortest paths

- BFS traversals can further be used to determine the *shortest path* between two vertices

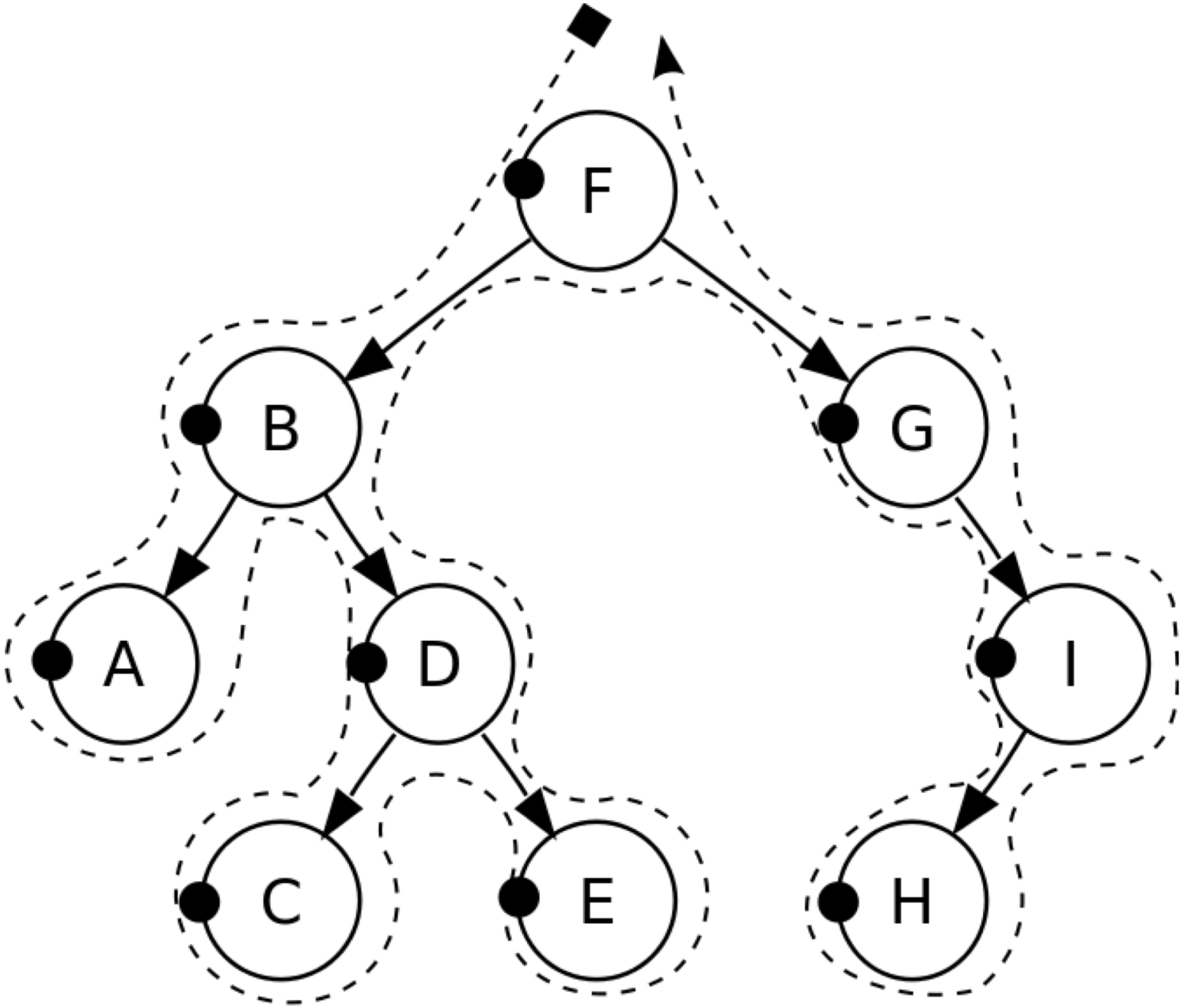
Analysis of graph traversals

- At a high level, DFS and BFS have the same runtime
  - Each vertex must be seen and then visited, but the order will differ between these two approaches
- How will the representation of the graph affect the runtimes of of these traversal algorithms?

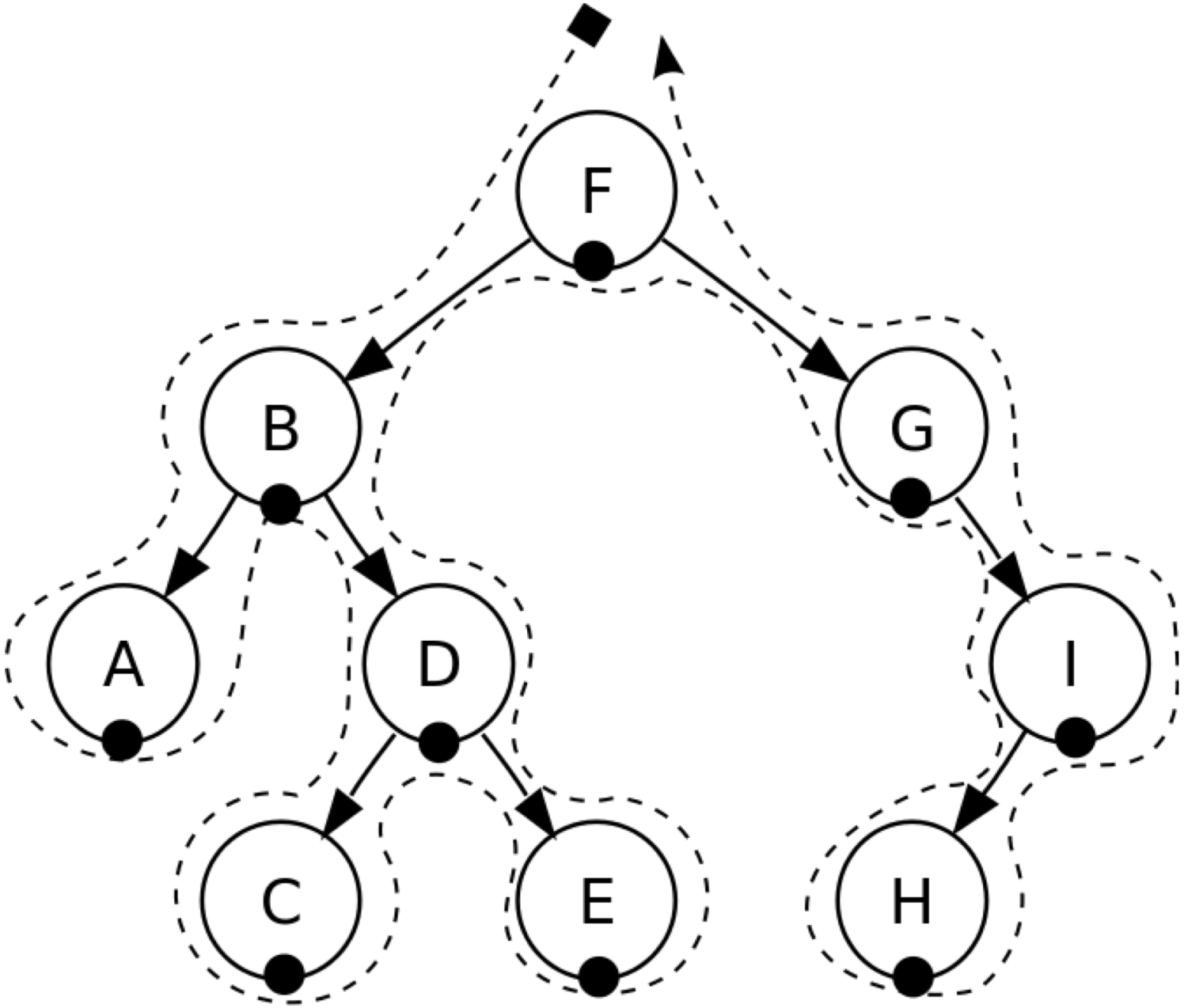
DFS and BFS would be called from a wrapper function

- If the graph is connected:
  - dfs()/bfs() is called only once and returns a *spanning tree*
- Else:
  - A loop in the wrapper function will have to continually call dfs()/bfs() while there are still unseen vertices
  - Each call will yield a spanning tree for a connected component of the graph

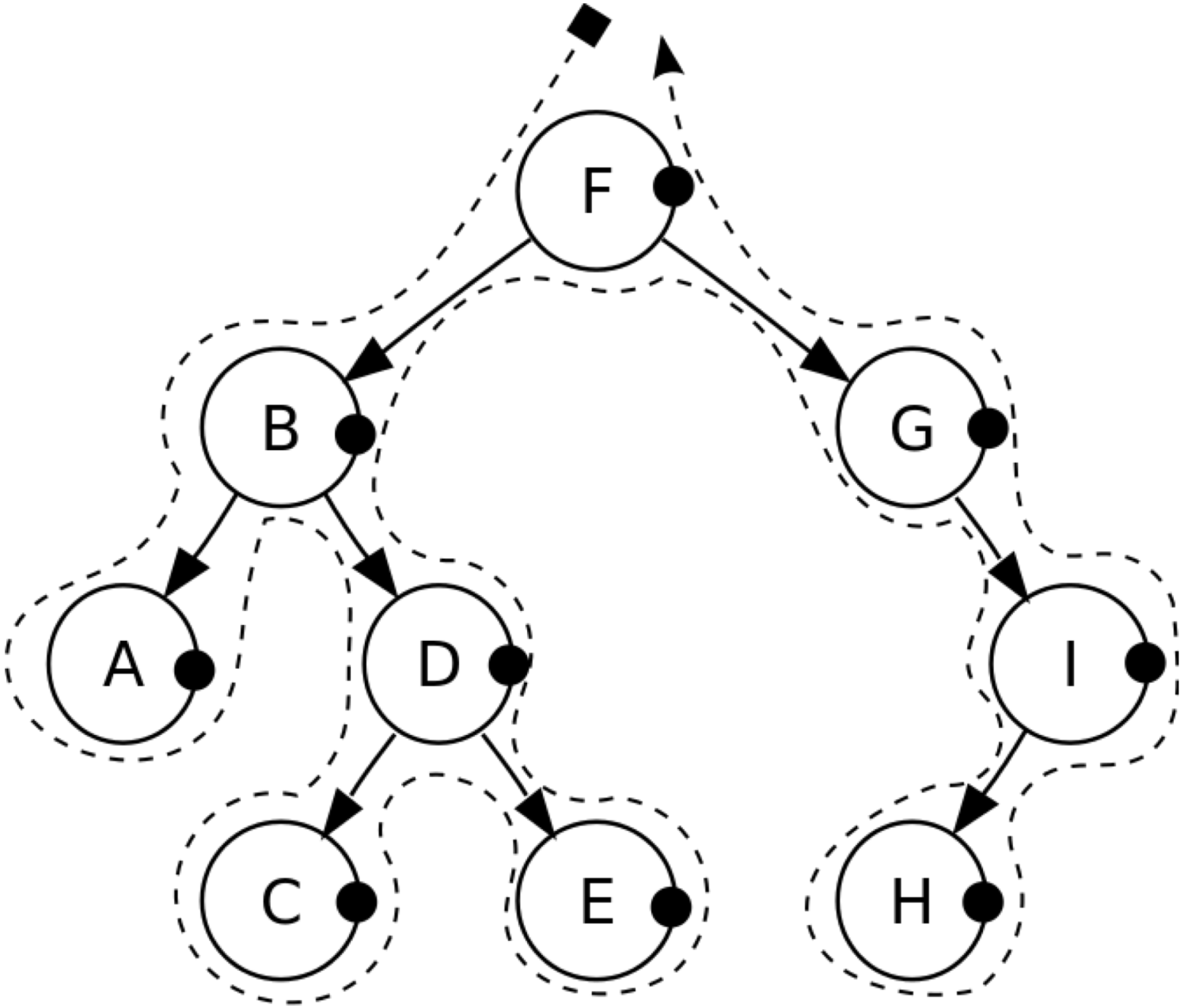
DFS pre-order traversal



DFS in-order traversal



DFS post-order traversal



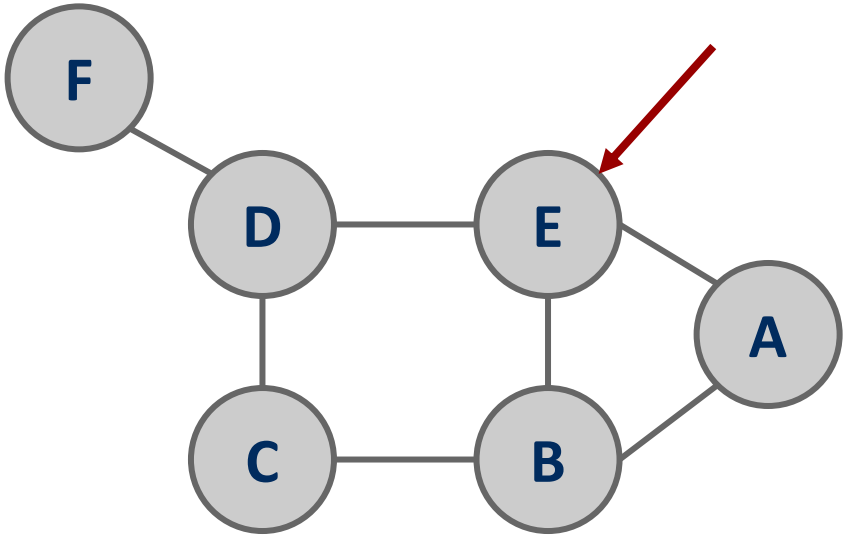
Biconnected graphs

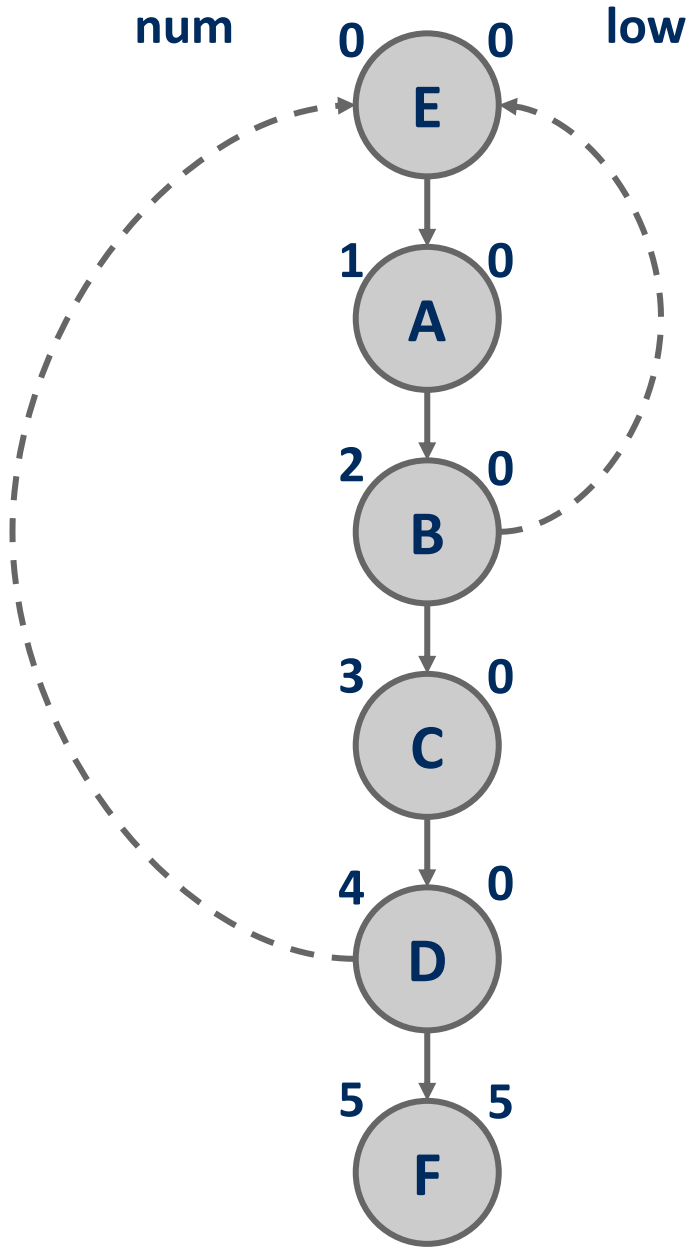
- A *biconnected graph* has at least 2 distinct paths (no common edges or vertices) between all vertex pairs
- Any graph that is not biconnected has one or more *articulation points*
  - Vertices, that, if removed, will separate the graph
- Any graph that has no articulation points is biconnected
  - Thus we can determine that a graph is biconnected if we look for, but do not find any articulation points

Finding articulation points

- Variation on DFS
- Consider building up the spanning tree
  - Have it be directed
  - Create “back edges” when considering a vertex that has already been visited in constructing the spanning tree
  - Label each vertex v with with two numbers:
    - num(v) = pre-order traversal order
    - low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then at most one back edge
      - Min of:
        - num(v)
        - Lowest num(w) of all back edges (v, w)
        - Lowest low(w) of all spanning tree edges (v, w)

Finding articulation points example





So where are the articulation points?

- If any (non-root) vertex v has some child w such that low(w) $\geq$ (only >) num(v), v is an articulation point
- What about if we start at an articulation point?
  - If the root of the spanning tree has more than one child, it is an articulation point

We said spatial layouts of graphs were irrelevant

- We define graphs as sets of vertices and edges
- However, we'll certainly want to be able to reason about bandwidth, distance, capacity, etc. of the real world things our graph represents
  - Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network
  - Having a road between two cities that is a 1 lane country road is very different from having a 4 lane highway
  - If two airports are 2000 miles apart, the number of flights going in and out between them will be drastically different from airports 200 miles apart

We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix?
  - Adjacency list?
  - Do we need a whole new graph representation?
- How do weights affect finding spanning trees/shortest paths?
  - The weighted variants of these problems are called finding the *minimum spanning tree* and the *weighted shortest path*

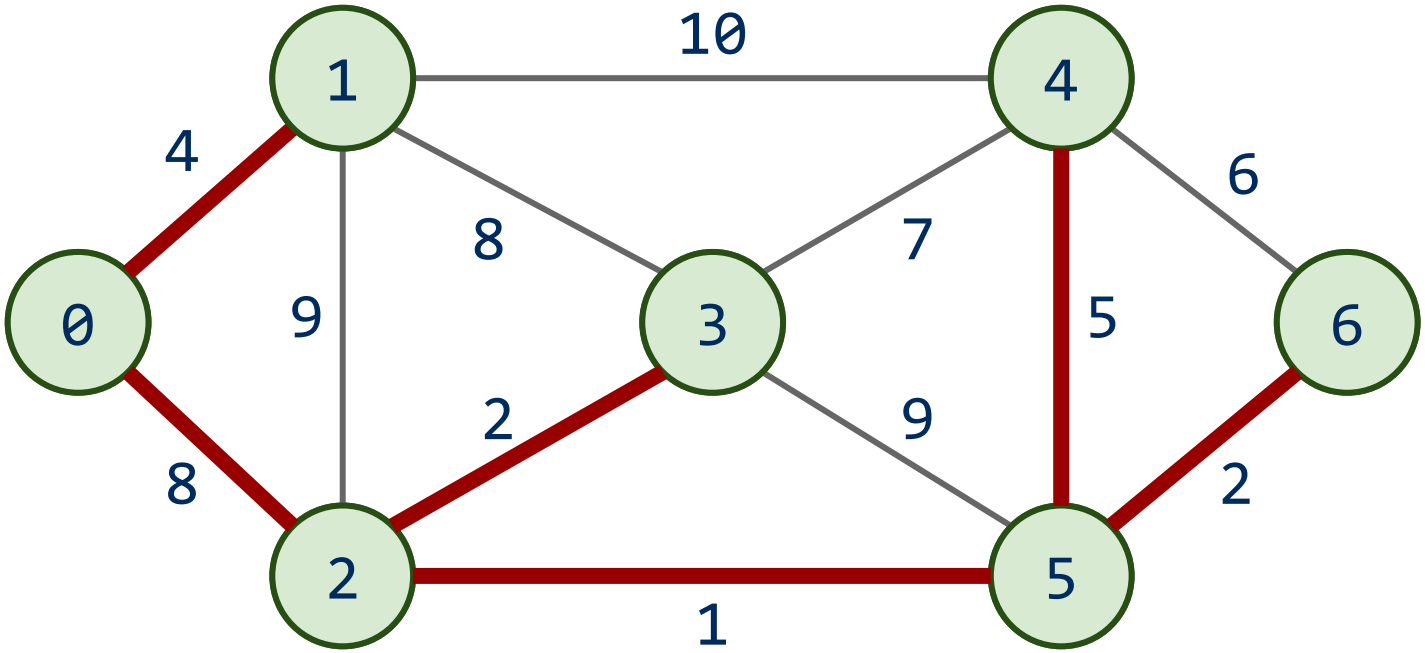
Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees
- MST is the spanning tree that has the minimum sum of the weights of its edges

Prim's algorithm

- Initialize T to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find minimum edge weight edge that connects a vertex in T to a vertex not yet in T
  - Add the edge with its vertex to T

Prim's algorithm



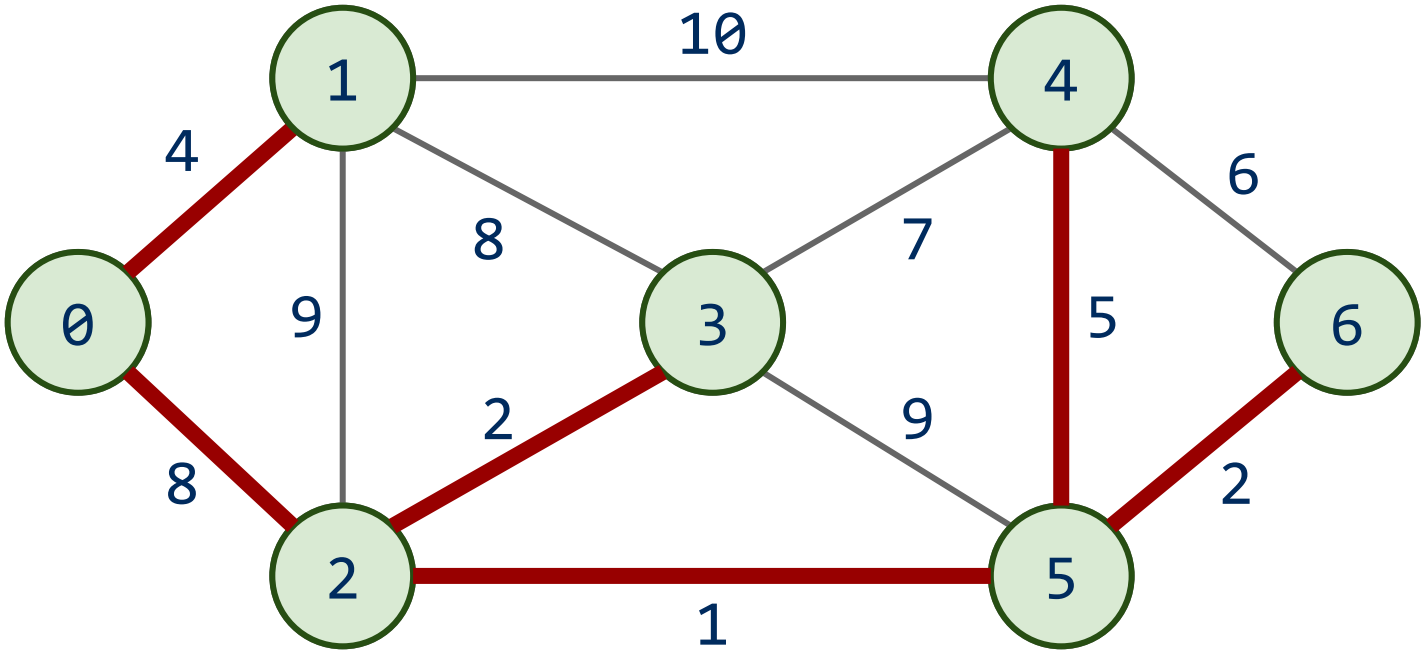
Runtime of Prim's

- At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - v - 1 possible edges
  - Next iteration:
    - 2(v - 2) possibilities
      - Each vertex in T shared v-1 edges with other vertices, but the edges they shared with each other already in T
  - Next:
    - 3(v - 3) possibilities
  - …
- Runtime:
  - $\Sigma_{i = 1 \text{ to } v} (i * (v - i))$
    - Evaluates to $\Theta(v^3)$

Do we need to look through all remaining edges?

- No! We only need to consider the *best* edge possible for each vertex!

Prim's algorithm



	0	1	2	3	4	5	6
Parent:	--	0	0	2	5	2	5
Best Edge:	0	4	8	2	5	1	2

OK, so what's our runtime?

- For every vertex we add to T, we'll need to check all of its neighbors to check for edges to add to T next
  - Let's assume we use an adjacency matrix:
    - Takes Θ(v) to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
    - Time to pick next vertex?
  - What about with an adjacency list?

We mentioned priority queues in building Huffman tries

- Primary operations they needed:
  - Insert
  - Find item with highest priority
    - E.g., findMin() or findMax()
  - Remove an item with highest priority
    - E.g., removeMin() or removeMax()
- How do we implement these operations?
  - Simplest approach: arrays

Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - Θ(1)
- Find:
  - Search for the highest priority item (e.g., min or max)
  - Θ(n)
- Remove:
  - Search for the highest priority item and delete
  - Θ(n)

- Runtime for use in Huffman tree generation?

Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - Θ(n)
- Find:
  - Return the item at the end of the array
  - Θ(1)
- Remove:
  - Return and delete the item at the end of the array
  - Θ(1)

- Runtime for use in Huffman tree generation?

So what other options do we have?

- What about a binary search tree?
  - Insert
    - Average case of Θ(lg n), but worst case of Θ(n)
  - Find
    - Average case of Θ(lg n), but worst case of Θ(n)
  - Remove
    - Average case of Θ(lg n), but worst case of Θ(n)
- OK, so in the average case, all operations are Θ(lg n)
  - No constant time operations
  - Worst case is Θ(n) for all operations

Is a BST overkill?

- Our find and remove operations only need the highest priority item, not to find/remove *any* item
  - Can we take advantage of this to improve our runtime?
    - Yes!

The heap

- A heap is complete binary tree such that for each node T in the tree:
  - T.item is of a higher priority than T.right_child.item
  - T.item is of a higher priority than T.left_child.item
- It does not matter how T.left_child.item relates to T.right_child.item
  - This is a relaxation of the approach needed by a BST

The *heap property*

Heap PQ runtimes

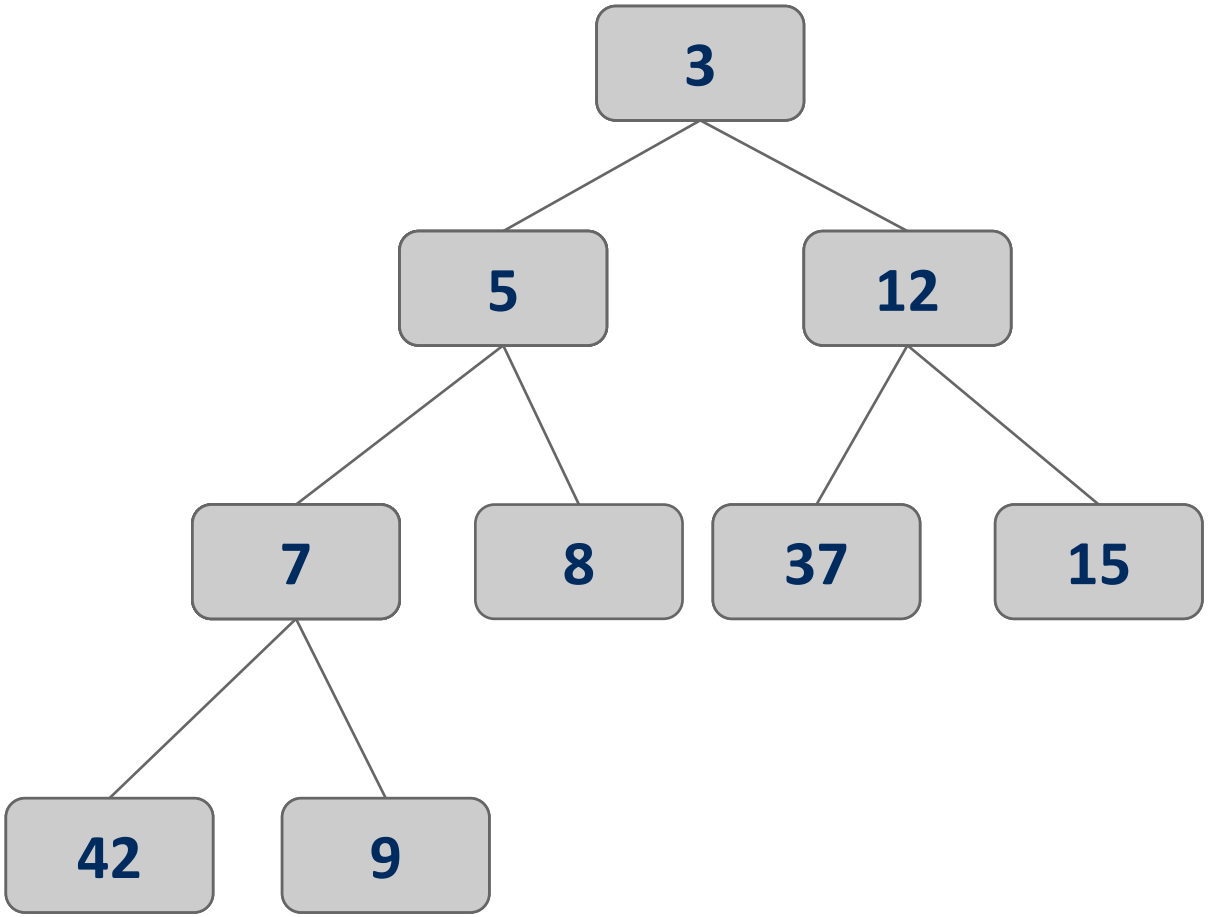
- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

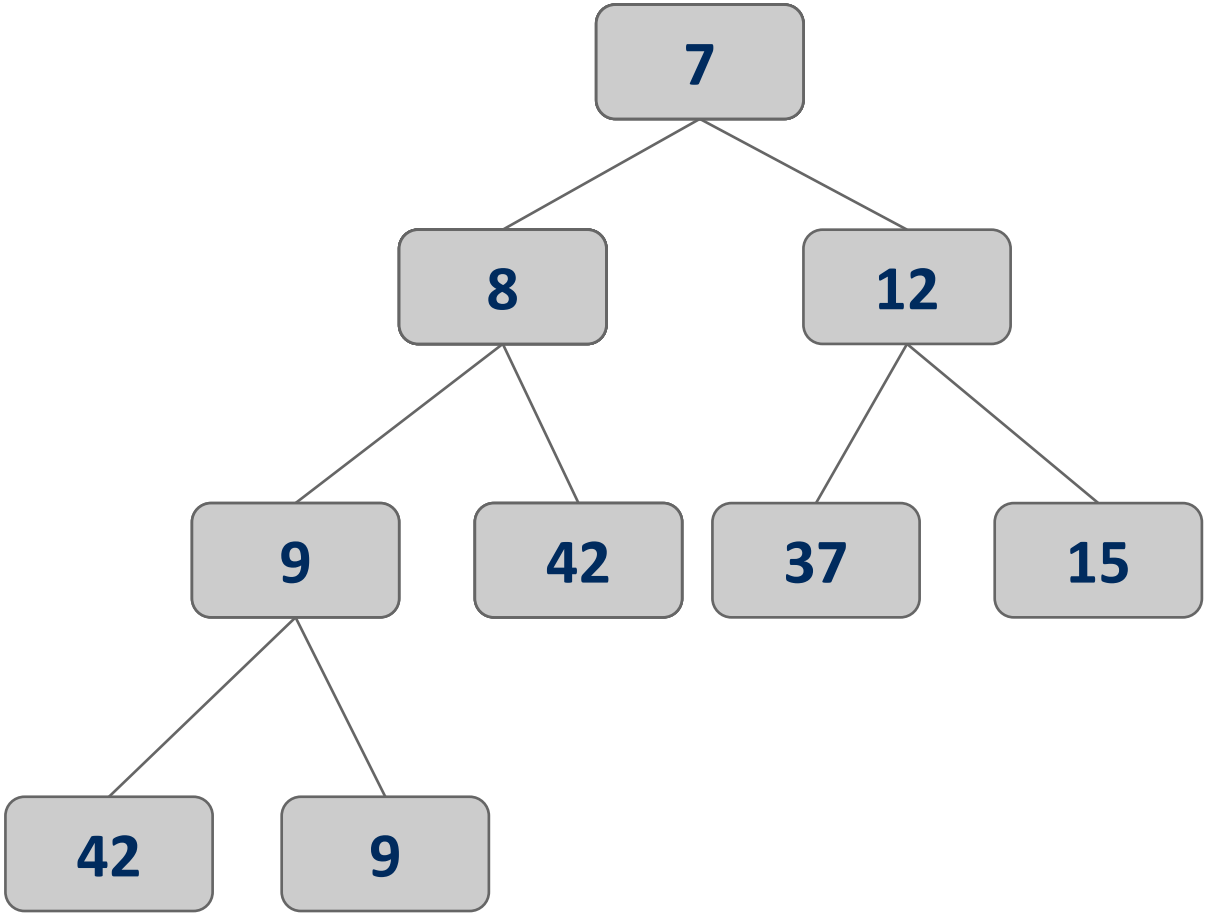


Heap remove

- Tricky to delete root…
  - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
    - But then the root is violating the heap property…
      - So we push the root down the tree until it is supporting the heap property

Min heap removal

NO!



Heap runtimes

- Find
  - $\Theta(1)$
- Insert and remove
  - Height of a complete binary tree is lg n
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are $\Theta(\lg n)$

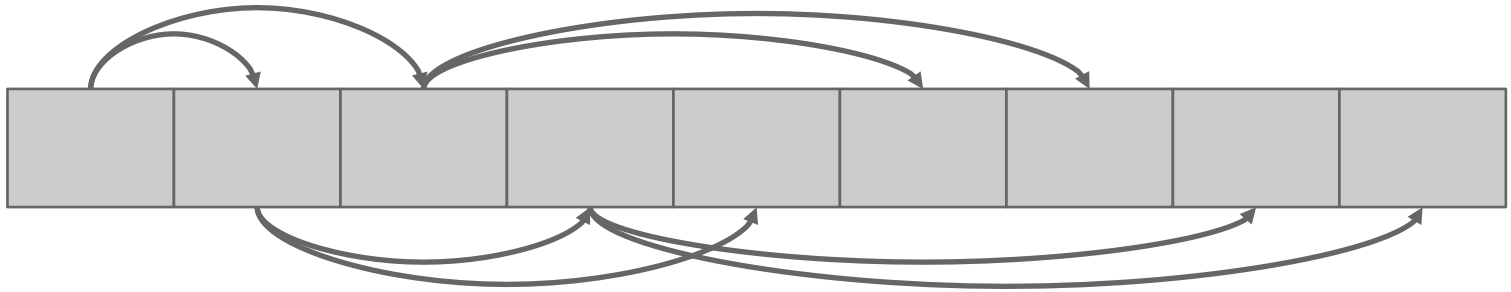
Heap implementation

- Simply implement tree nodes like for BST
  - This requires overhead for dynamic node allocation
  - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree…
  - We can easily represent a complete binary tree using an array

Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i
  - parent(i) = ⌊(i - 1) / 2⌋
  - left_child(i) = 2i + 1
  - right_child(i) = 2i + 2

For arrays indexed from 0



Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat

Heap sort analysis

- Runtime:
  - Worst case:
    - n log n
- In-place?
  - Yes
- Stable?
  - No

Storing Objects in PQ

- What if we want to update an Object?
  - What is the runtime to find an arbitrary item in a heap?
    - Θ(n)
    - Hence, updating an item in the heap is Θ(n)
  - Can we improve of this?
    - Back the PQ with something other than a heap?
    - Develop a clever workaround?

Indirection

- Maintain a second data structure that maps item IDs to each item's current position in the heap
- This creates an *indexable* PQ

Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

```
class CardPrice implements Comparable<CardPrice>{
      public String store;
      public double price;
      public CardPrice(String s, double p) { … }
      public int compareTo(CardPrice o) {
            if (price < o.price) { return -1; }
            else if (price > o.price) { return 1; }
            else { return 0; }
      }
}
```

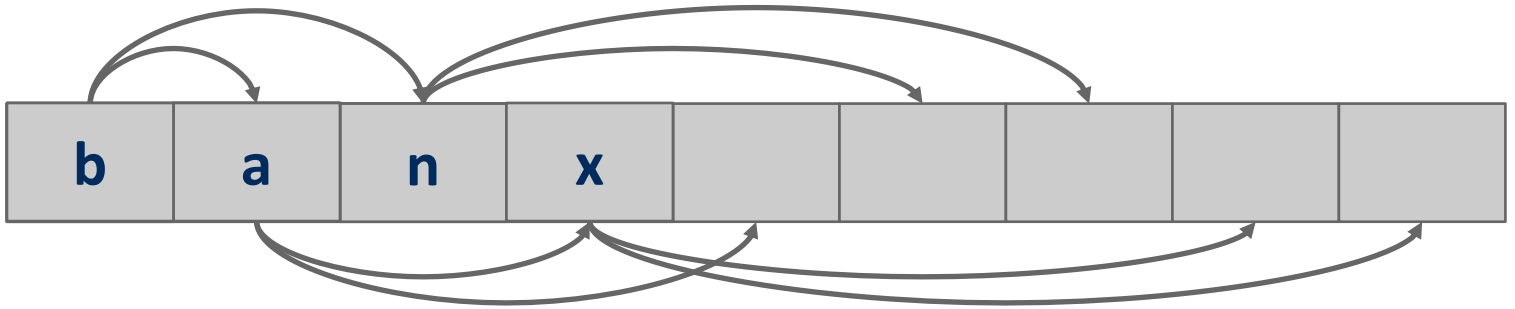
Indirection example

- n = new CardPrice("NE", 333.98);
- a = new CardPrice("AMZN", 339.99);
- x = new CardPrice("NCIX", 338.00);
- b = new CardPrice("BB", 349.99);

- Update price for NE: 340.00
- Update price for NCIX: 345.00
- Update price for BB: 200.00

Indirection

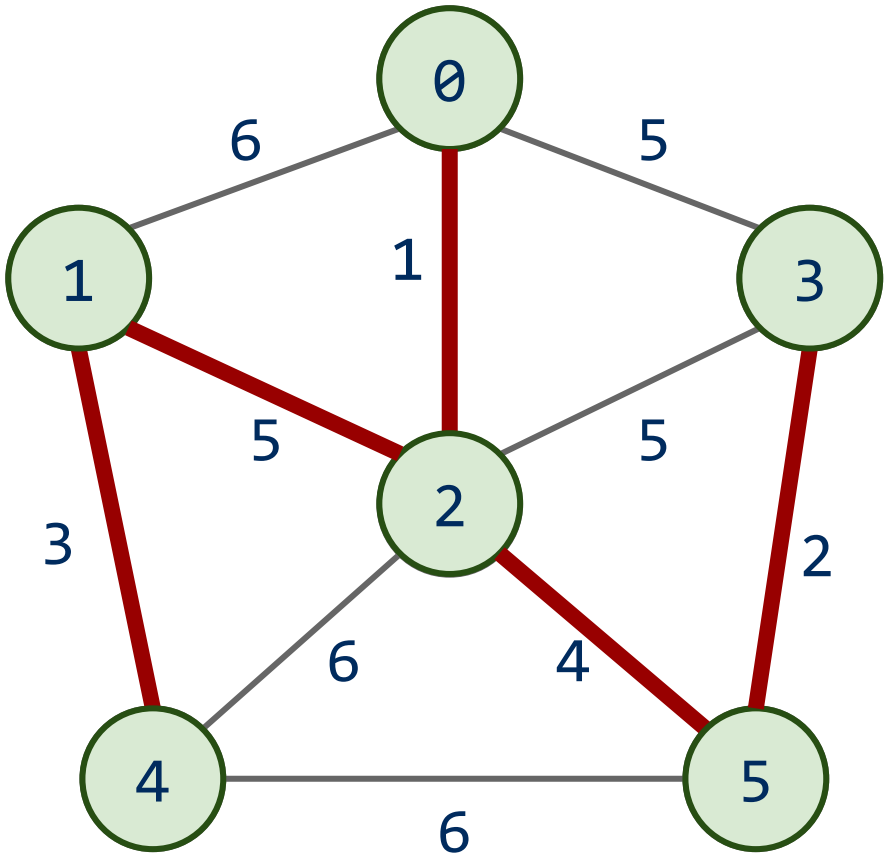
"NE":2
"AMZN":1
"NCIX":3
"BB":0



Prim's MST: What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in Θ(lg n)
    - Also Θ(lg n) to add to the priority queue
- What does our algorithm look like now?
  - Visit a vertex
  - Add edges coming out of it to a PQ
  - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

1: (0, 2)

2: (5, 3)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (2, 1)

6: (0, 1)

6: (2, 4)

6: (5, 4)

Runtime using a priority queue

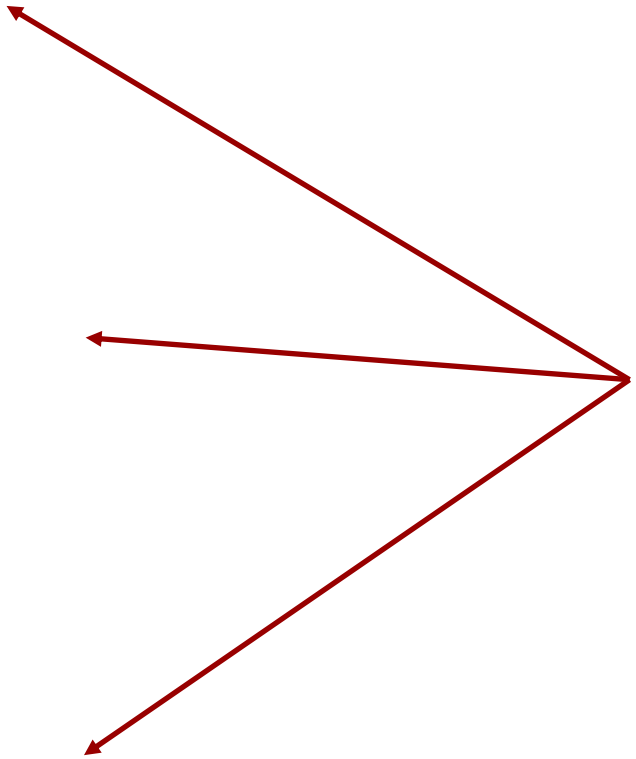
- Have to insert all e edges into the priority queue
  - In the worst case, we'll also have to remove all e edges
- So we have:
  - $e * \Theta(\lg e) + e * \Theta(\lg e)$
  - $= \Theta(2 * e \lg e)$
  - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Do we really need to maintain e items in the PQ?

- I suppose we could not be so lazy
- Just like with the adjacency matrix implementation, we only need the best edge for each vertex
  - PQ will need to be indexable
- This is the idea of *eager Prim's*
  - Runtime is Θ(e lg v)

Comparison of Prim's implementations

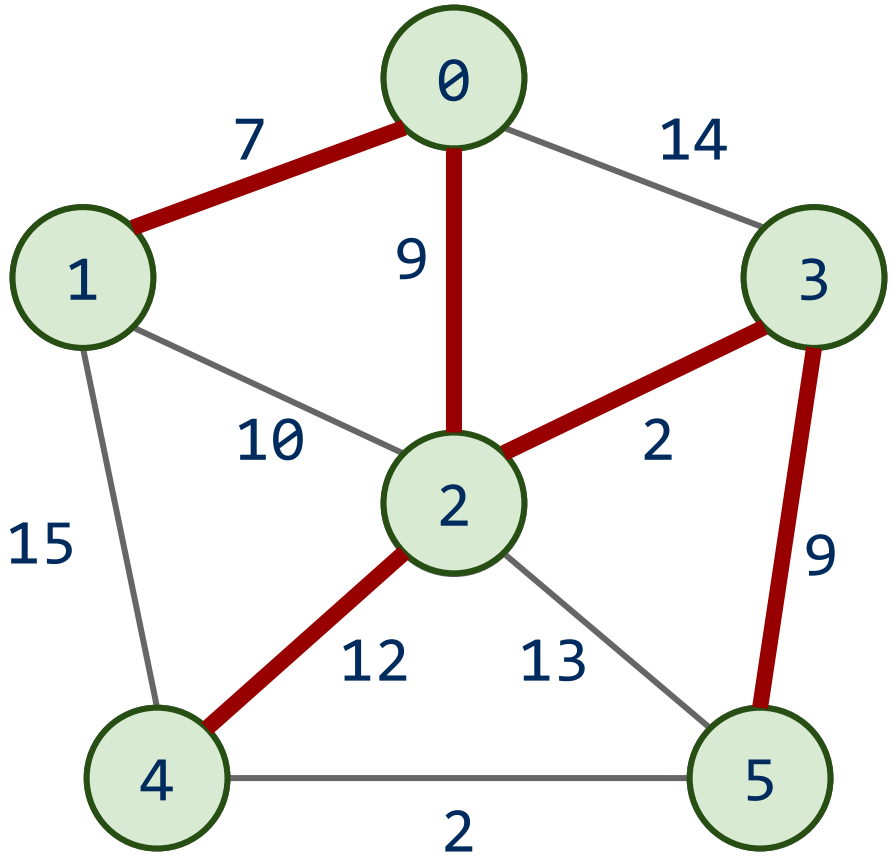
- Parent/Best Edge array Prim's
  - Runtime: $\Theta(v^2)$
  - Space: $\Theta(v)$
- Lazy Prim's
  - Runtime: $\Theta(e \lg e)$
  - Space: $\Theta(e)$
  - Requires a PQ
- Eager Prim's
  - Runtime: $\Theta(e \lg v)$
  - Space: $\Theta(v)$
  - Requires an indexable PQ



Weighted shortest path

- Dijkstra's algorithm:
  - Set a distance value of MAX_INT for all vertices but start
  - Set cur = start
  - While destination is not visited:
    - For each unvisited neighbor of cur:
      - Compute tentative distance from start to the unvisited neighbor through cur
      - Update any vertices for which a lesser distance is computed
    - Mark cur as visited
    - Let cur be the unvisited vertex with the smallest tentative distance from start

Dijkstra's example



	Distance	Via
0	0	--
1	7	0
2	9	0
3	11	2
4	21	2
5	20	3

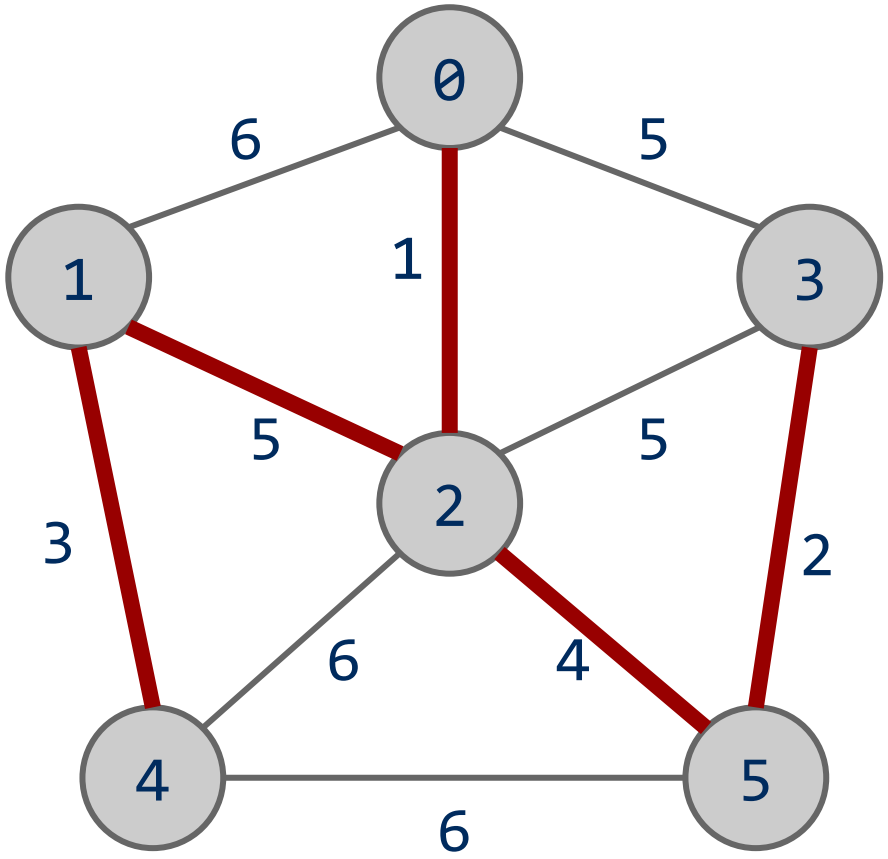
Analysis of Dijkstra's algorithm

- How to implement?
  - Best path/parent array?
    - Runtime?
  - PQ?
    - Turns out to be very similar to Eager Prims
      - Storing paths instead of edges
    - Runtime?

Back to MSTs: Another MST algorithm

- Kruskal's MST:
  - Insert all edges into a PQ
  - Grab the min edge from the PQ that does not create a cycle in the MST
  - Remove it from the PQ and add it to the MST

Kruskal's example



PQ:

1: (0, 2)

2: (3, 5)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (1, 2)

6: (0, 1)

6: (2, 4)

6: (4, 5)

Kruskal's runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?

Dynamic connectivity problem

- For a given graph G, can we determine whether or not two vertices are connected in G?
- Can also be viewed as checking subset membership
- Important for many practical applications
- We will solve this problem using a *union/find* data structure

A simple approach

- Have an *id* array simply store the component id for each item in the union/find structure
  - How do we determine if two vertices are connected?
  - How do we establish the connected components?
    - Add graph edges one at a time to UF data structure using *union* operations

Example

U(2, 0)

U(4, 7)

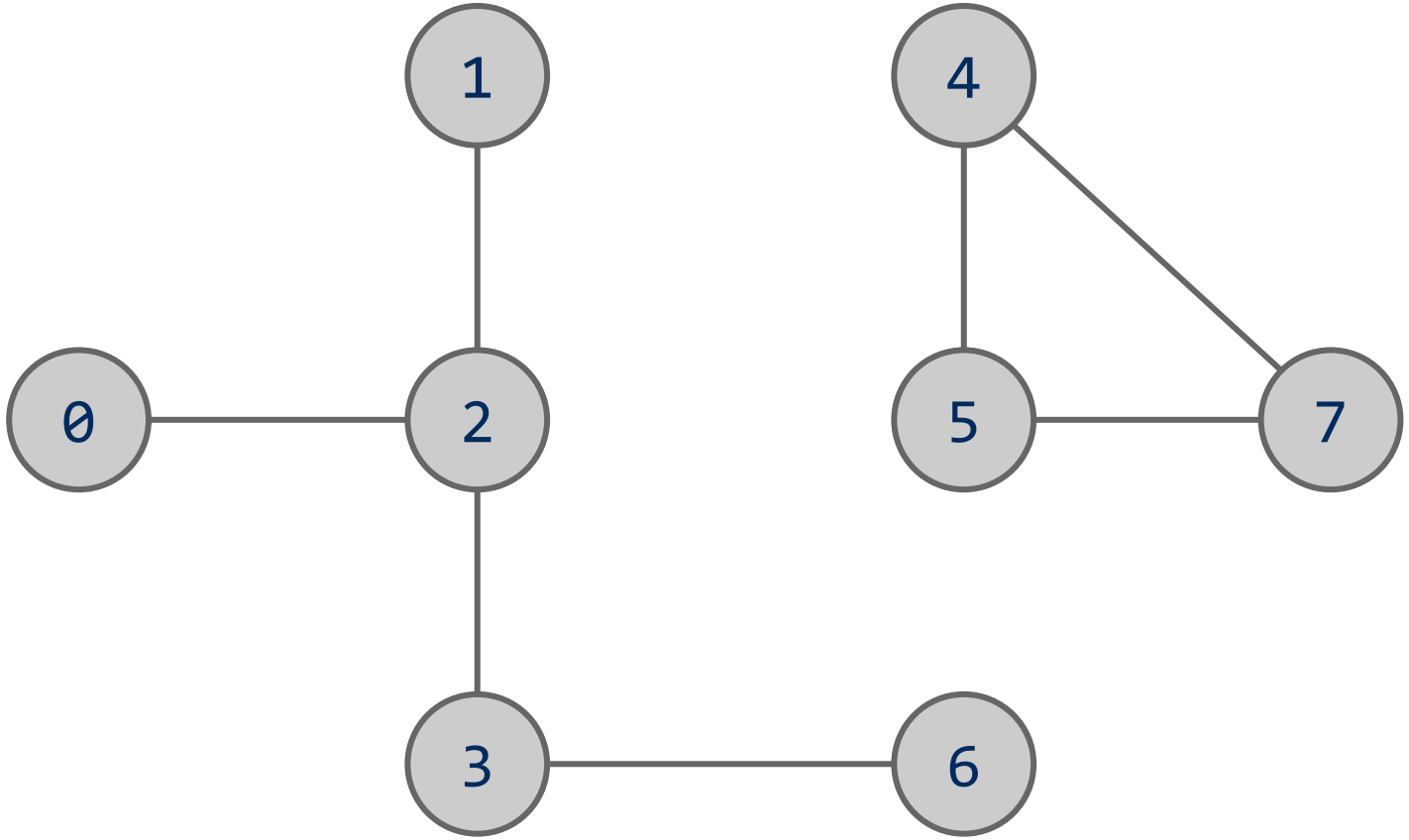
U(1, 2)

U(3, 2)

U(4, 5)

U(5, 7)

U(6, 3)

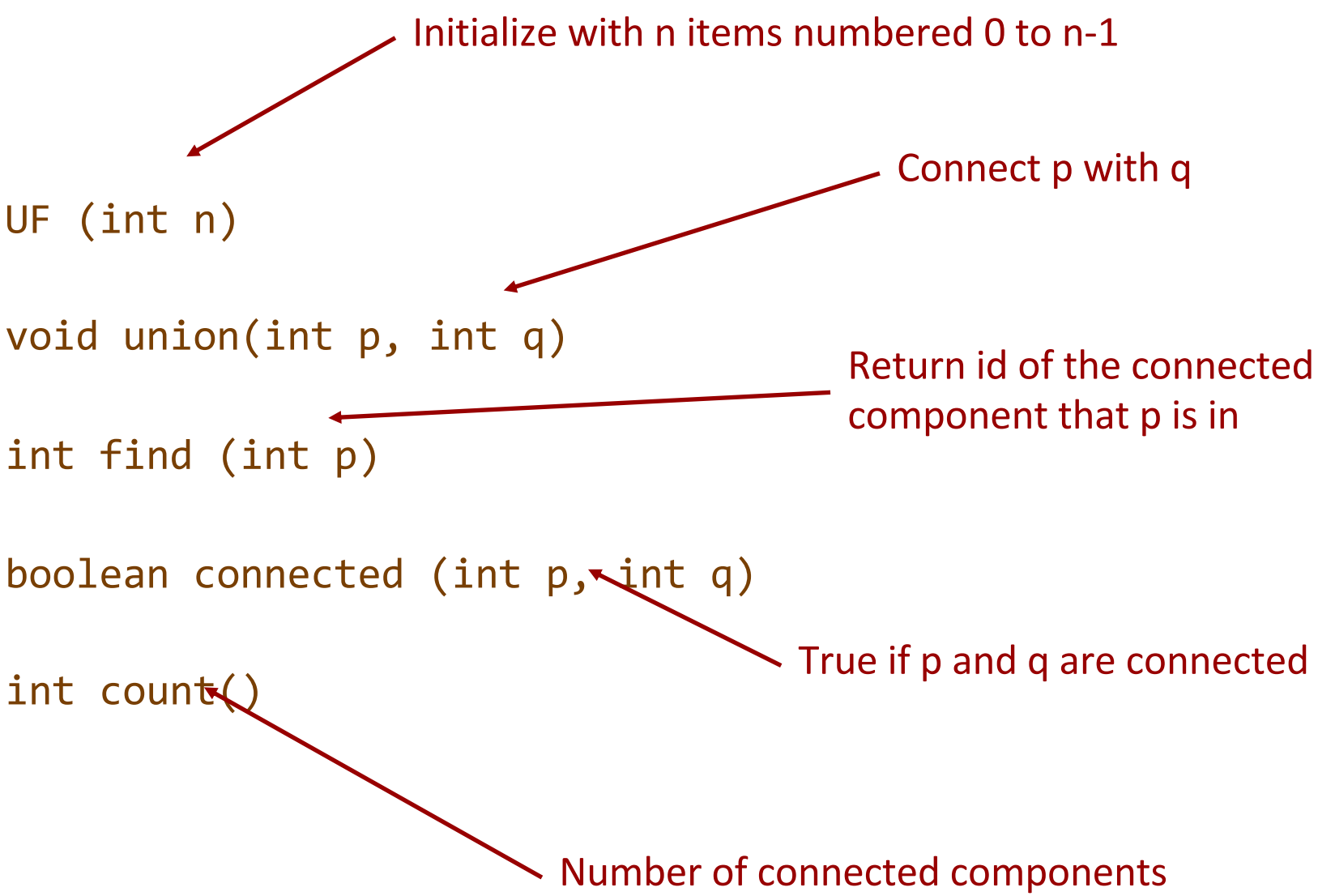


	0	1	2	3	4	5	6	7
ID:	6	6	6	6	4	4	6	4

Analysis of our simple approach

- Runtime?
  - To find if two vertices are connected?
  - For a union operation?

Union Find API



Covering the basics

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

public boolean connected(int p, int q) {
    return find(p) == find(q);
}
```

Implementing the Fast-Find approach

```
public UF(int n) {
    count = n;
    id = new int[n];
    for (int i = 0; i < n; i++) { id[i] = i; }
}

public int find(int p) { return id[p]; }

public void union(int p, int q) {
       int pID = find(p), qID = find(q);
       if (pID == qID) return;
       for(int i = 0; i < id.length; i++)
              if (id[i] == pID) id[i] = qID;
       count--;
}
```

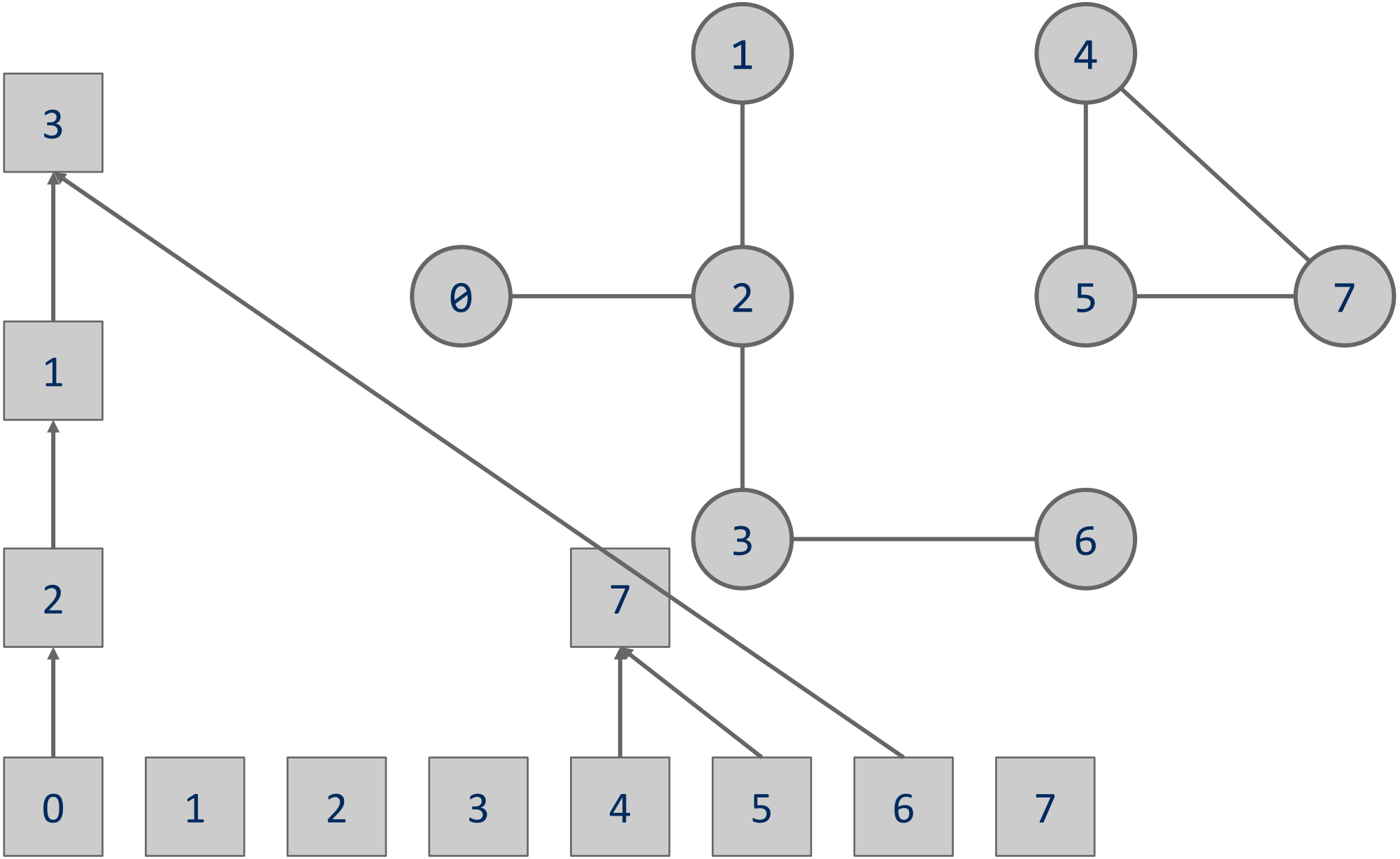
Kruskal's algorithm

- With this knowledge of union/find, how, exactly can it be used as a part of Kruskal's algorithm?
  - What is the runtime of Kruskal's algorithm?

Can we improve on union()'s runtime?

- What if we store our connected components as a forest of trees?
  - Each tree representing a different connected component
  - Every time a new connection is made, we simply make one tree the child of another

Tree example



Implementation using the same `id` array

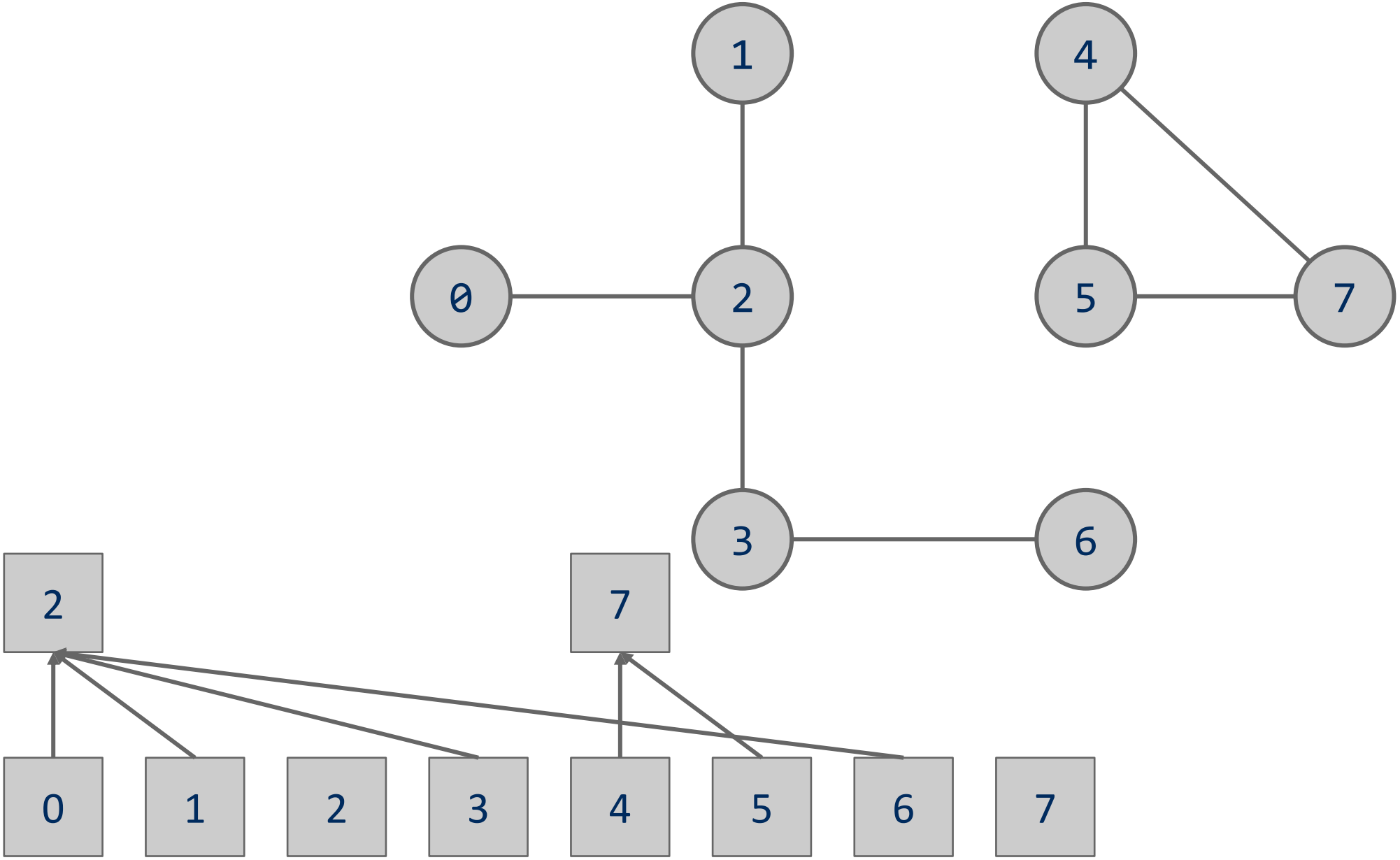
```
public int find(int p) {
    while (p != id[p]) p = id[p];
    return p;
}

public void union(int p, int q) {
       int i = find(p);
    int j = find(q);
       if (i == j) return;
       id[i] = j;
       count--;
}
```

Forest of trees implementation analysis

- Runtime?
  - find():
    - Bound by the height of the tree
  - union():
    - Bound by the height of the tree
- What is the max height of the tree?
  - Can we modify our approach to cap its max height?

Weighted tree example



Weighted trees

```
public UF(int n) {
    count = n;
    id = new int[n];
    sz = new int[n];
    for (int i = 0; i < n; i++) { id[i] = i; sz[i] = 1; }
}

public void union(int p, int q) {
       int i = find(p), j = find(q);
       if (i == j) return;
       if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
       else               { id[j] = i; sz[i] += sz[j]; }
       count--;
}
```

Weighted tree approach analysis

- Runtime?
  - find()?
  - union()?

- Can we do any better?

Kruskal's algorithm, once again

- What is the runtime of Kruskal's algorithm??

Defining network flow

- Consider a directed, weighted graph G(V, E)
  - Weights are applied to edges to state their *capacity*
    - c(u, w) is the capacity of edge (u, w)
    - if there is no edge from u to w, c(u, w) = 0
- Consider two vertices, a *source s* and a *sink t*
  - Let's determine the maximum flow that can run from s to t in the graph G

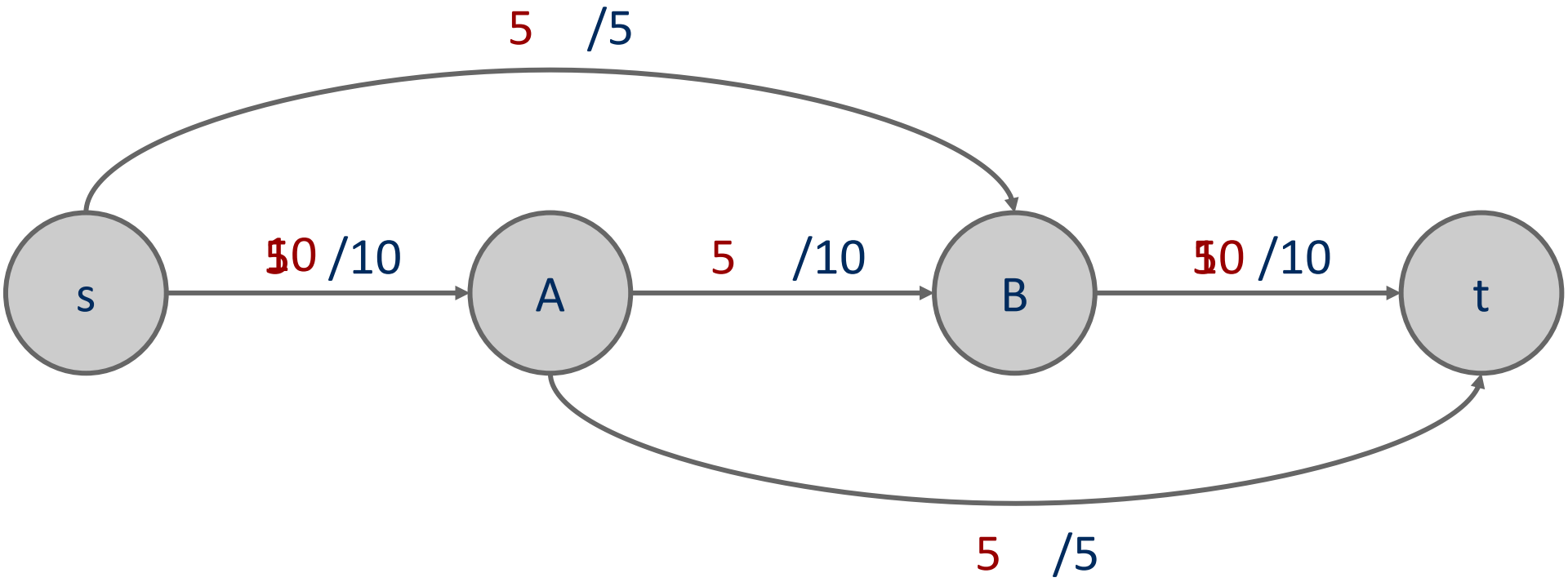
Flow

- Let the f(u, w) be the amount of flow being carried along the edge (u, w)
- Some rules on the flow running through an edge:
  - $\forall (u, w) \in E \; f(u, w) <= c(u, w)$
  - $\forall u \in (V - \{s,t\}) \; (\Sigma_{w \in V} f(w, u) - \Sigma_{w \in V} f(u, w)) = 0$

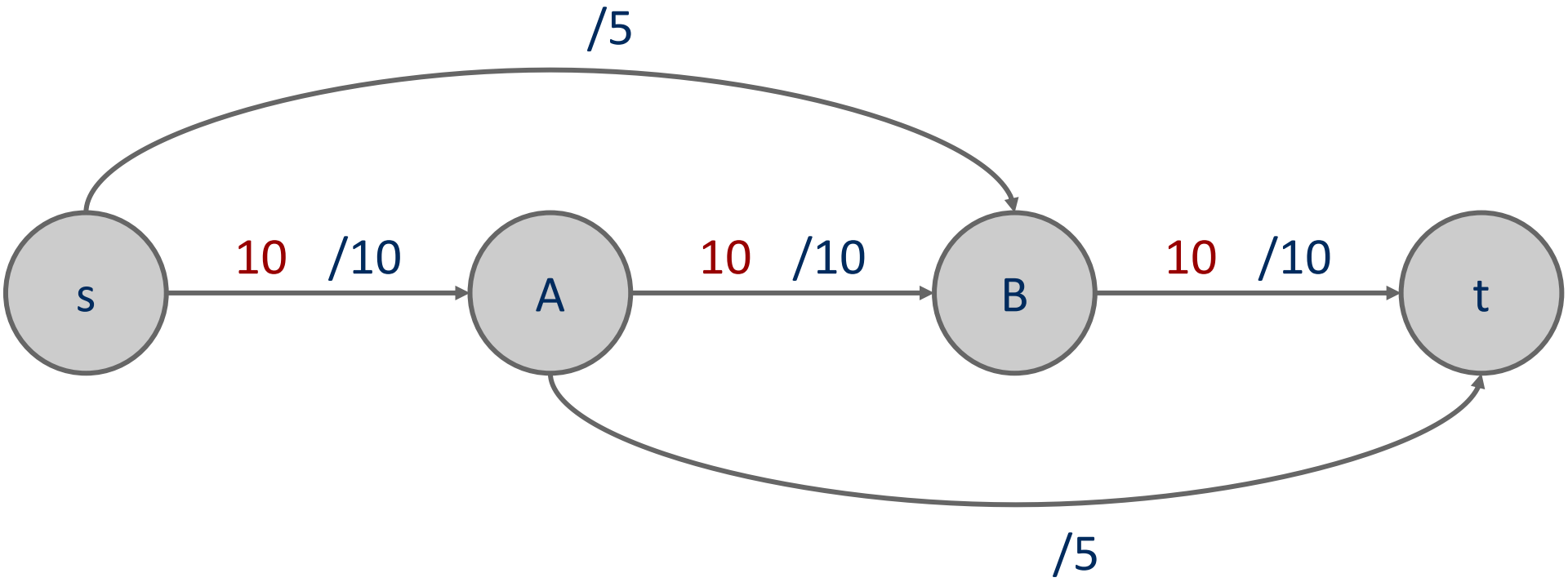
Ford Fulkerson

- Let all edges in G have an allocated flow of 0
- While there is path *p* from s to t in G s.t. all edges in p have some *residual capacity* (i.e., $\forall(u, w) \in p\ f(u, w) < c(u, w)$):
  - (Such a path is called an *augmenting path*)
  - Compute the residual capacity of each edge in p
    - Residual capacity of edge (u, w) is c(u, w) - f(u, w)
  - Find the edge with the minimum residual capacity in p
    - We'll call this residual capacity *new_flow*
  - Increment the flow on all edges in p by new_flow

Ford Fulkerson example



Another Ford Fulkerson example



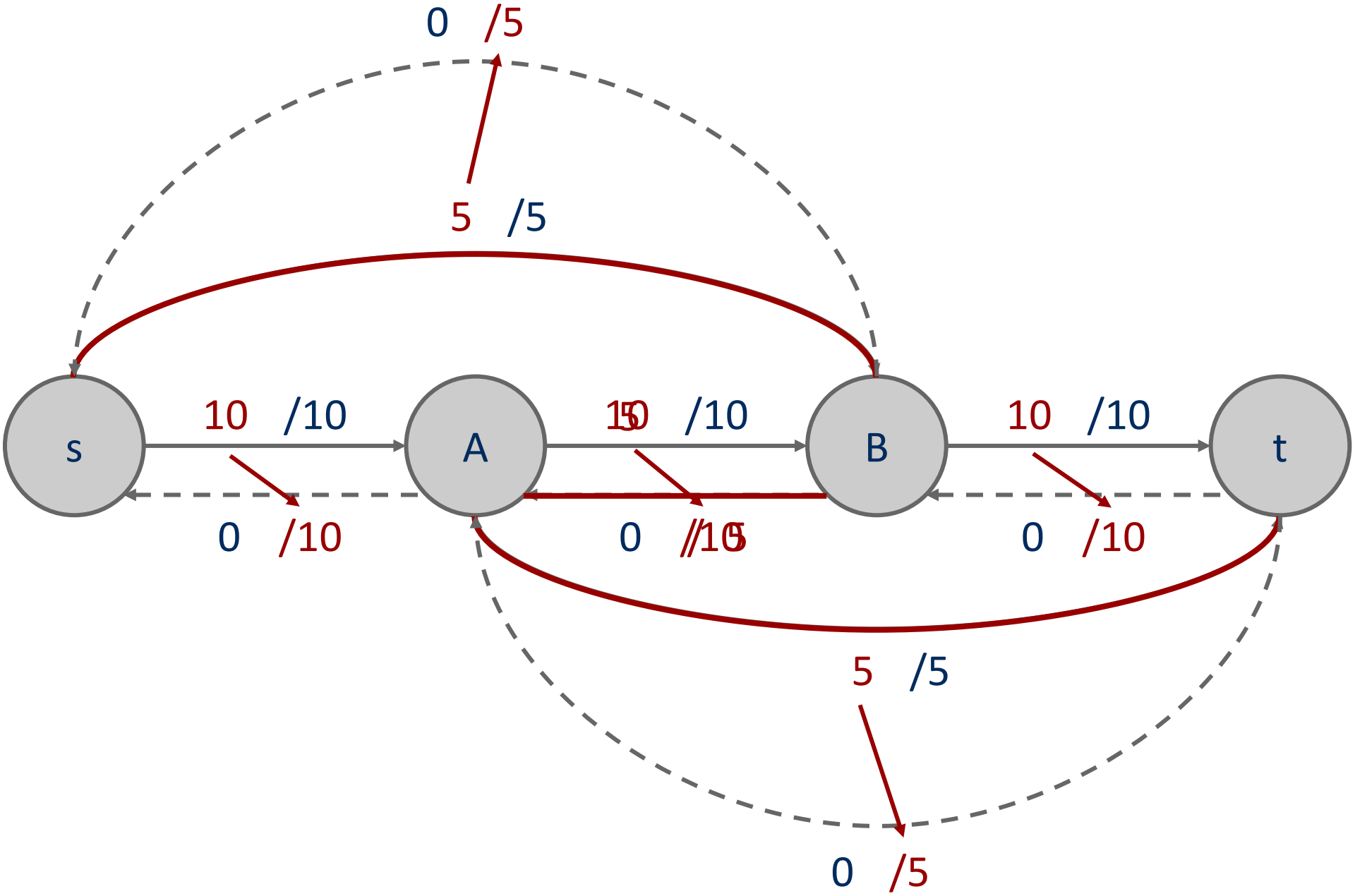
Expanding on residual capacity

- To find the max flow we will have need to consider re-routing flow we had previously allocated
  - This means, when finding an augmenting path, we will need to look not only at the edges of G, but also at *backwards edges* that allow such re-routing
    - For each edge (u, w) ∈ E, a backwards edge (w, u) must be considered during pathfinding if f(u, w) > 0
      - The capacity of a backwards edge (w, u) is equal to f(u, w)

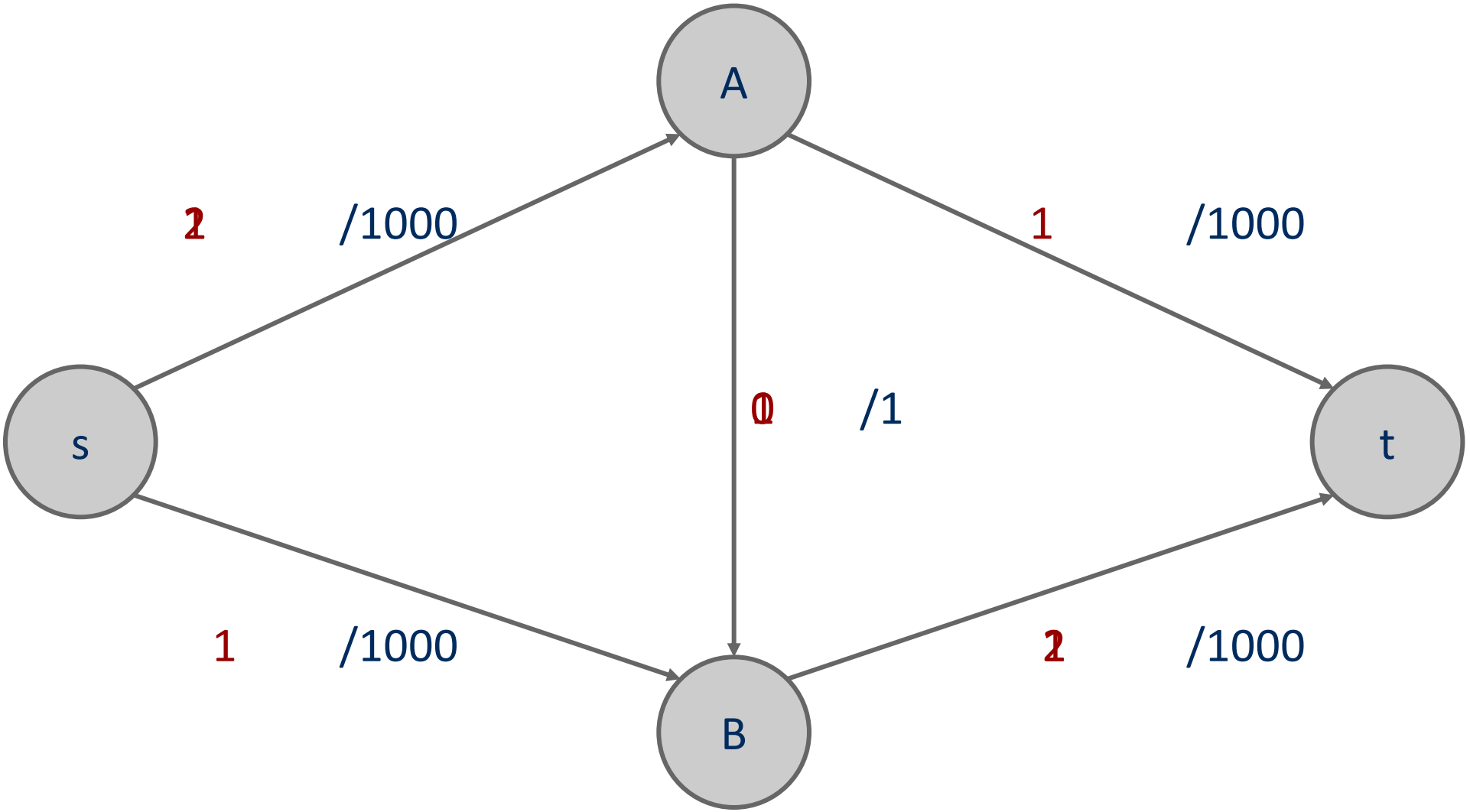
The residual graph

- We will perform searches for an augmenting path not on G, but on a residual graph built using the current state of flow allocation on G
- The residual graph is made up of:
  - V
  - An edge for each $(u, w) \in E$ where $f(u, w) < c(u, w)$
    - (u, w)'s mirror in the residual graph will have 0 flow and a capacity of $c(u, w) - f(u, w)$
  - A backwards edge for each $(u, w) \in E$ where $f(u, w) > 0$
    - (u, w)'s backwards edge has a capacity of $f(u, w)$
    - All backwards edges have 0 flow

Residual graph example



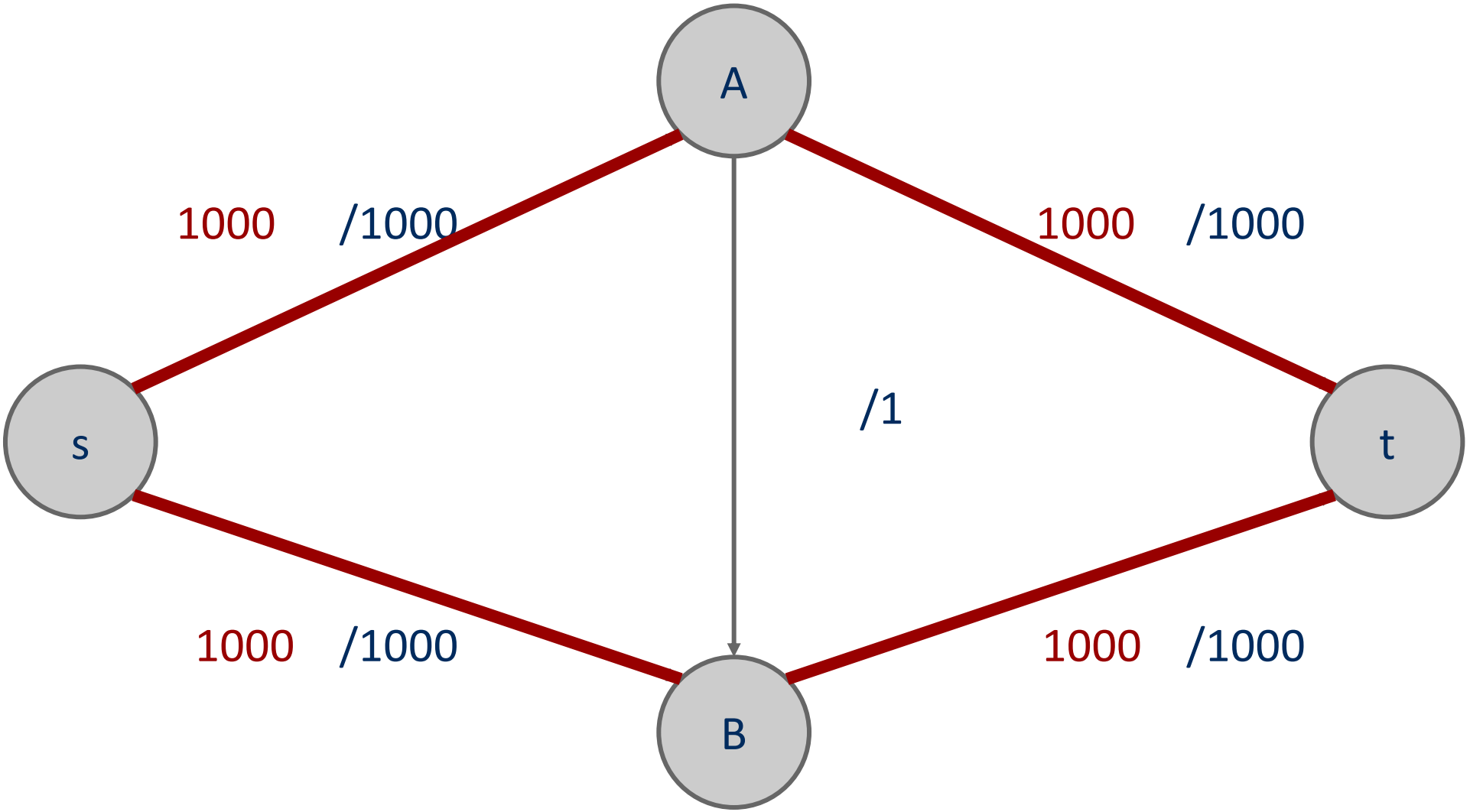
Another example



Edmonds Karp

- How the augmenting path is chosen affects the performance of the search for max flow
- Edmonds and Karp proposed a shortest path heuristic for Ford Fulkerson
  - Use BFS to find augmenting paths

Another example



But our flow graph is weighted...

- Edmonds-Karp only uses BFS
  - Used to find spanning trees and shortest paths for *unweighted* graphs
  - Why do we not use some measure of priority to find augmenting paths?

Implementation concerns

- Representing the graph:
  - Similar to a directed graph
  - Can store an adjacency list of directed edges
    - Actually, more than simply directed edges
      - Flow edges

Flow edge implementation

- For each edge, we need to store:
  - Start point, the from vertex
  - End point, the to vertex
  - Capacity
  - Flow
  - Residual capacities
    - For forwards and backwards edges

FlowEdge.java

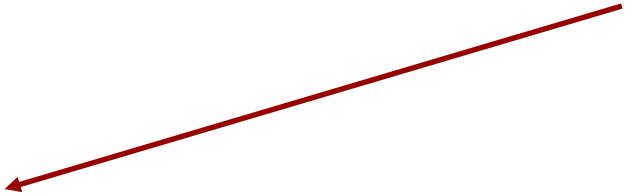
```
public class FlowEdge {
   private final int v;                // from
   private final int w;                // to
   private final double capacity;      // capacity
   private double flow;                // flow
      …
      public double residualCapacityTo(int vertex) {
       if      (vertex == v) return flow;
       else if (vertex == w) return capacity - flow;
       else throw new
        IllegalArgumentException("Illegal endpoint");
   }
   …
}
```

BFS search for an augmenting path (pseudocode)

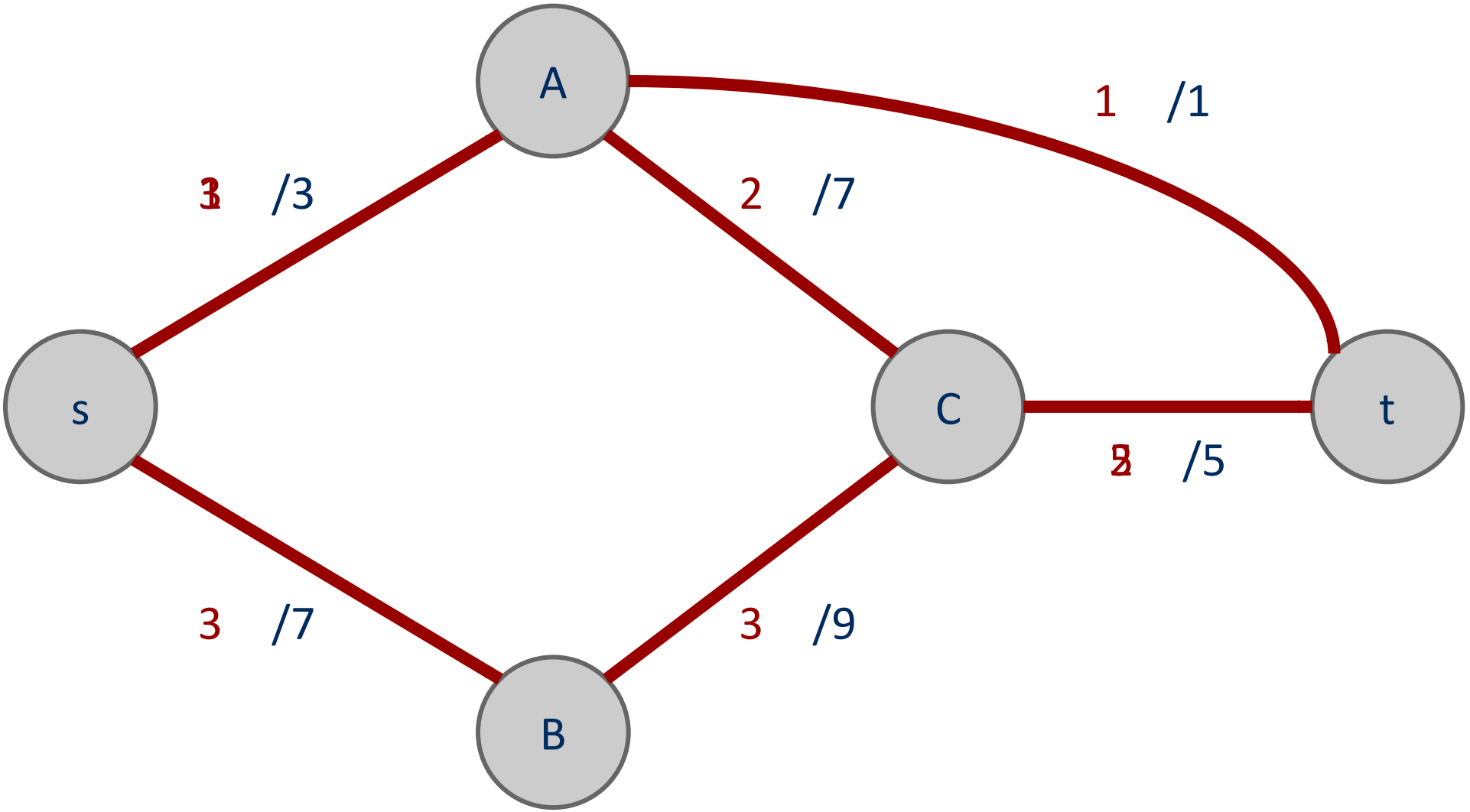
```
edgeTo = [|V|]
marked = [|V|]
Queue q
q.enqueue(s)
marked[s] = true
while !q.isEmpty():
    v = q.dequeue()
    for each (v, w) in AdjList[v]:
        if residualCapacity(v, w) > 0:
            if !marked[w]:
                edgeTo[w] = e;
                marked[w] = true;
                q.enqueue(w);
```

Each FlowEdge object is stored in the adjacency list twice:

Once for its forward edge
Once for its backwards edge



An example to review



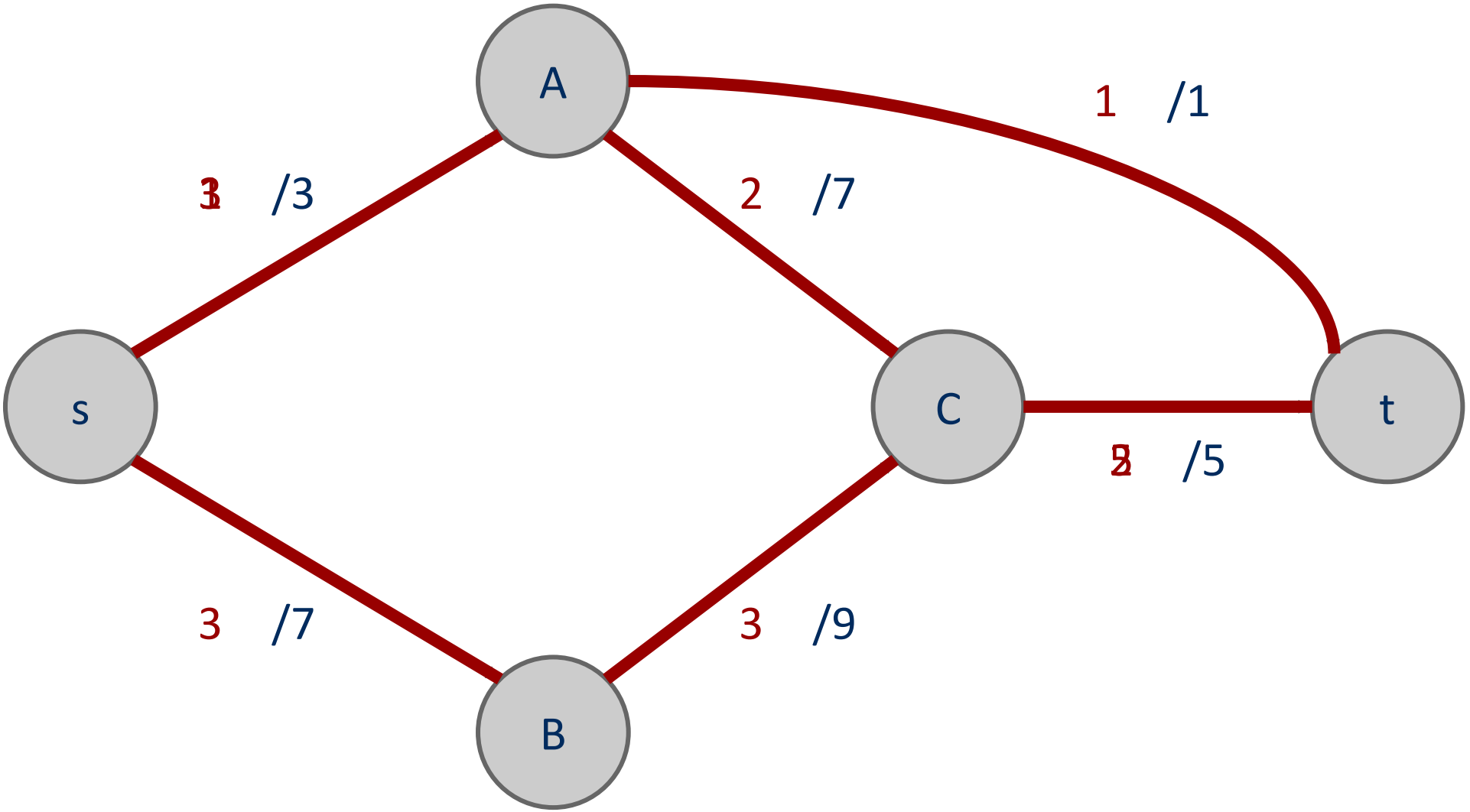
Let's separate the graph

- An st-cut on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t
- May be many st-cuts for a given graph
- Let’s focus on finding the minimum st-cut
  - The st-cut with the smallest capacity
  - May not be unique

How do we find the min st-cut?

- We could examine residual graphs
  - Specifically, try and allocate flow in the graph until we get to a residual graph with no existing augmenting paths
    - A set of saturated edges will make a minimum st-cut

Min cut example



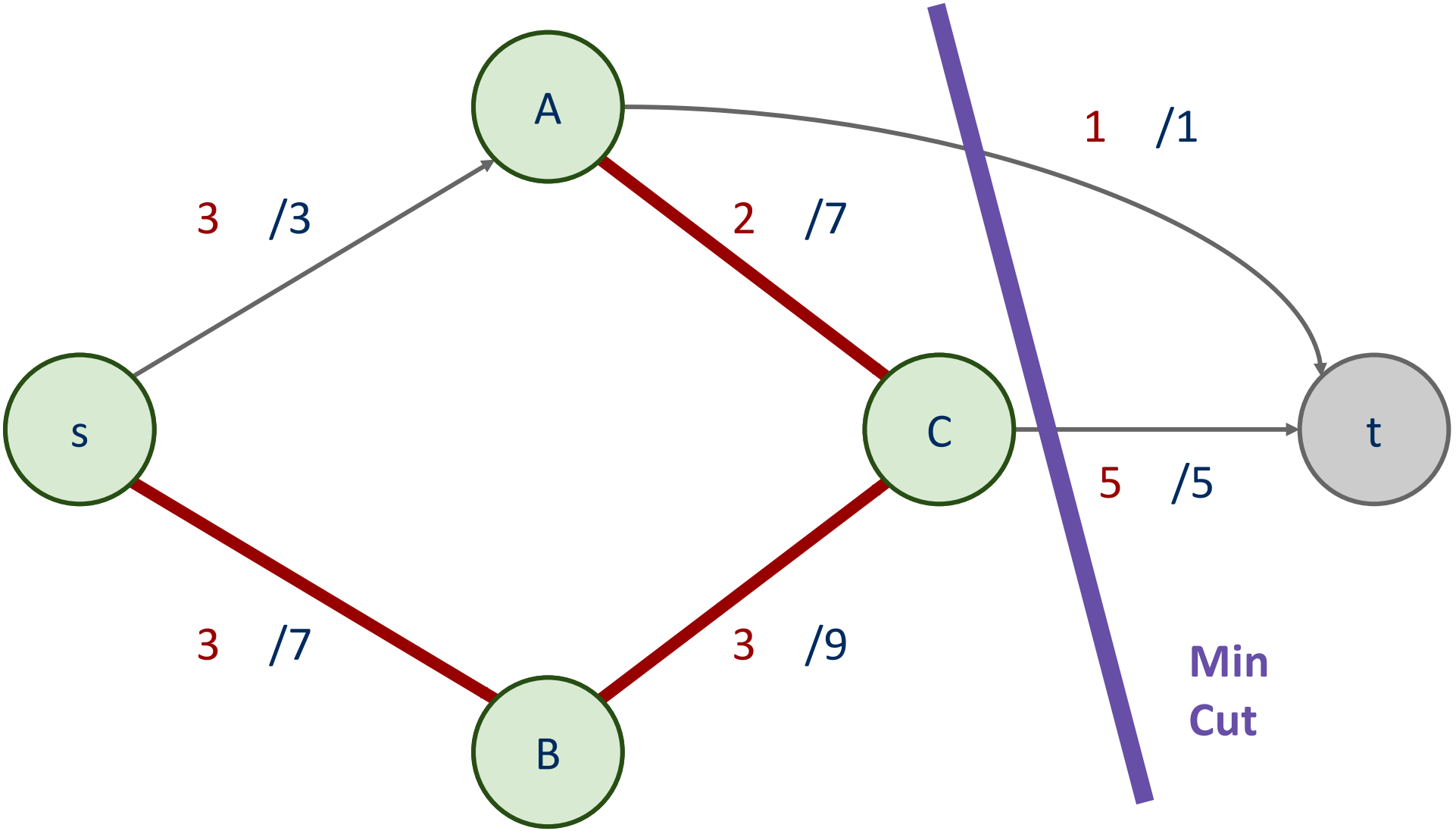
Max flow == min cut

- A special case of duality
  - I.e., you can look at an optimization problem from two angles
    - In this case to find the maximum flow or minimum cut
  - In general, dual problems do not have to have equal solutions
    - The differences in solutions to the two ways of looking at the problem is referred to as the *duality gap*
      - If the duality gap = 0, strong duality holds
        - Max flow/min cut uphold strong duality
      - If the duality gap > 0, weak duality holds

Determining a minimum st-cut

- First, run Ford Fulkerson to produce a residual graph with no further augmenting paths
- The last attempt to find an augmenting path will visit every vertex reachable from s
  - Edges with only one endpoint in this set comprise a minimum st-cut

Determining the min cut



Max flow / min cut on unweighted graphs

- Is it possible?
- How would we measure the Max flow / min cut?
- What would an algorithm to solve this problem look like?

Unweighted network flow