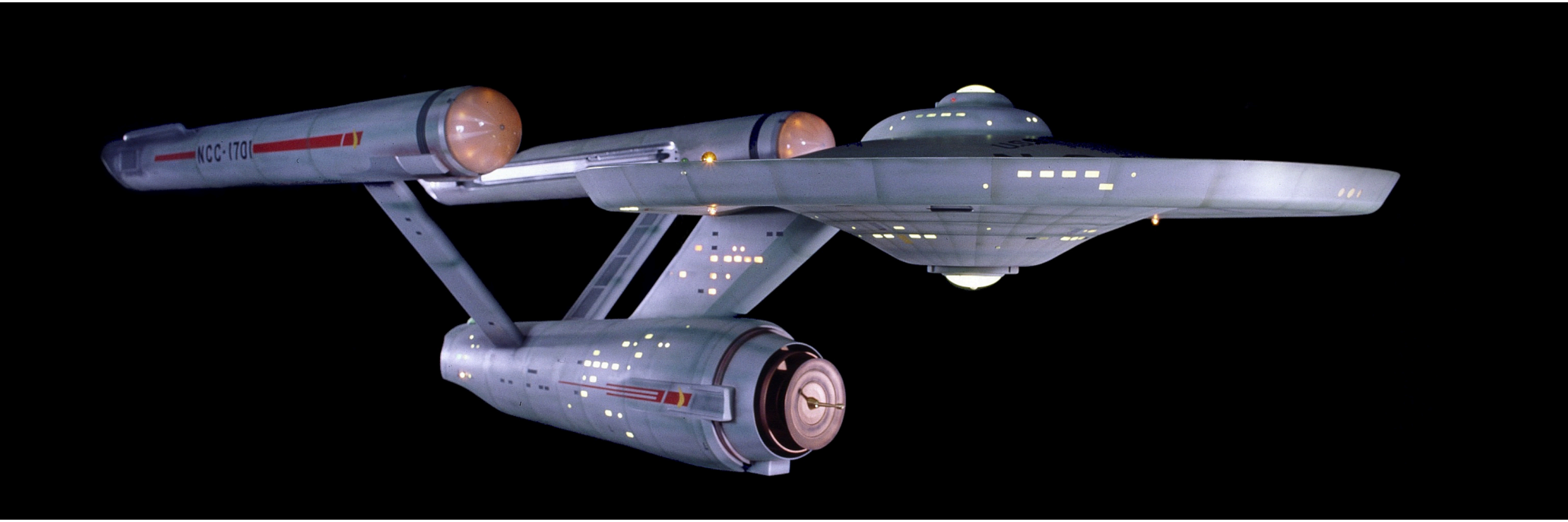


CSCI 241

Lecture 21
Dijkstra Proof of Correctness
More Graph Stuff, MSTs

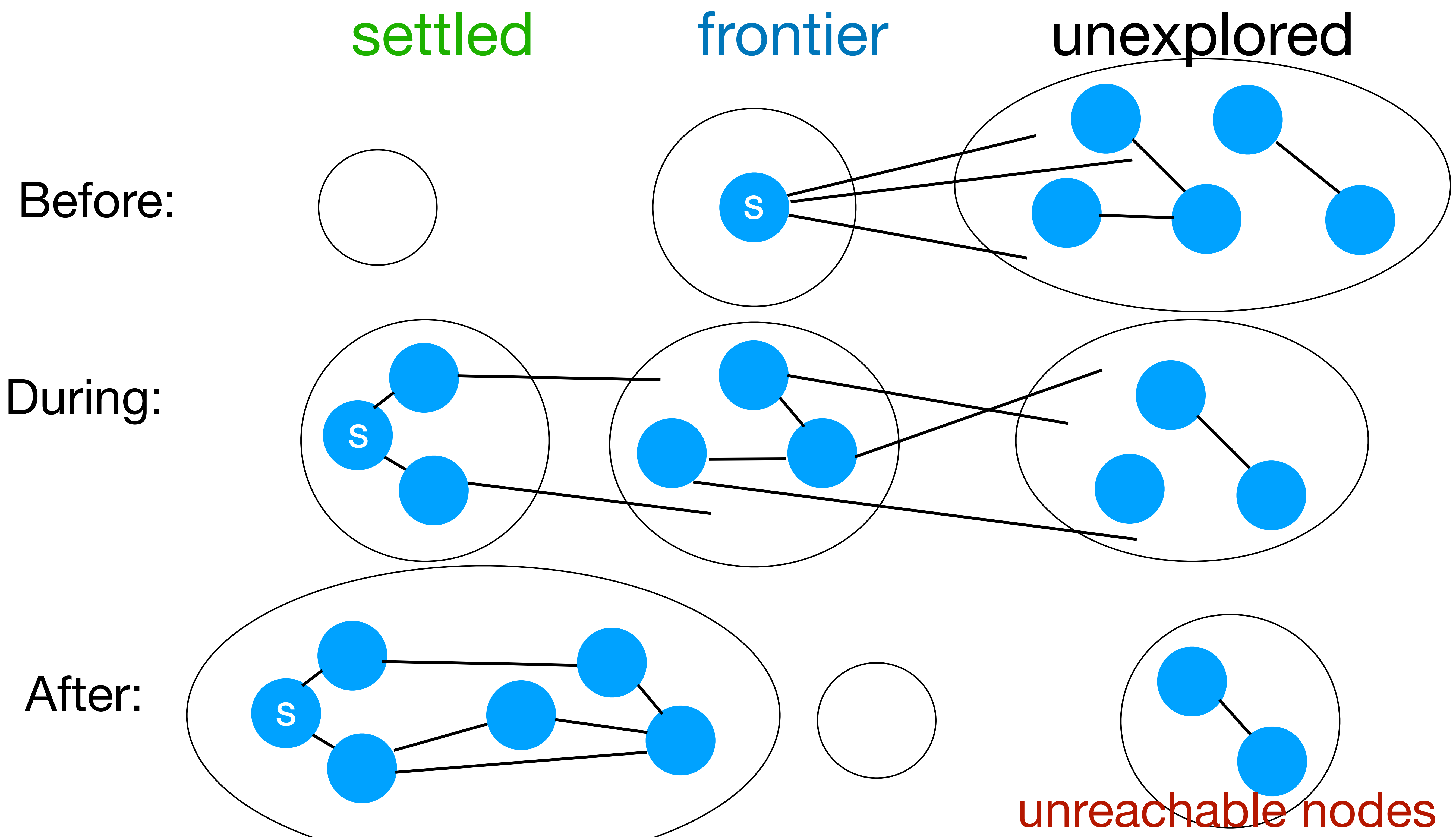
Announcements

- Extra office hours are a possibility tomorrow if there's demand.
- Final exam study guide coming soon.
  - Same as midterm: it's just the Goals from each lecture.
- Final exam:
  - 3/18 10:30am-12:30pm
  - You'll be allowed **two** 2-sided 8.5x11 sheets of hand-written notes.

Goals

- See a proof of correctness of Dijkstra's algorithm.
- Know what it means for a graph to be planar
- Know the definition of a Directed Acyclic Graph (DAG) and how to check whether a graph is a DAG using Topological Sort.

Dijkstra’s Shortest Paths: Cartoon




The next slide is so important,
I’m going to show it to you
again.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>
4. No need to explicitly store Settled or Unexplored sets: a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
  for each neighbor w of f {
     if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
     }
  }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. **Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>**
4. No need to explicitly store Settled or Unexplored sets:
   a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>
4. **No need to explicitly store Settled or Unexplored sets: a node is in S or F iff it is in the map.**

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
     }
   }
}
```

4. No need to explicitly store Settled or Unexplored sets:
w is in S or F <=> it is in the map.

The only time we need to check membership in S is **here**.

If w is not in S or F,
it must be in Unexplored.

therefore,
we haven't found a path to it.

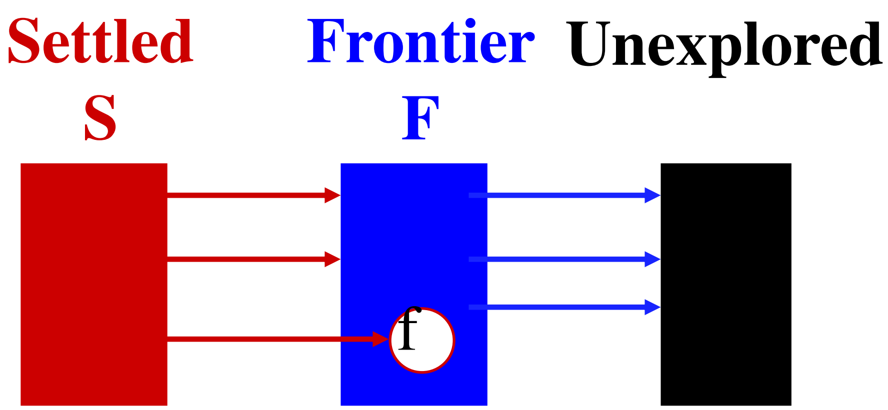
therefore,
it has no d or bp yet.

therefore,
it isn't in the map!

Proof of Correctness

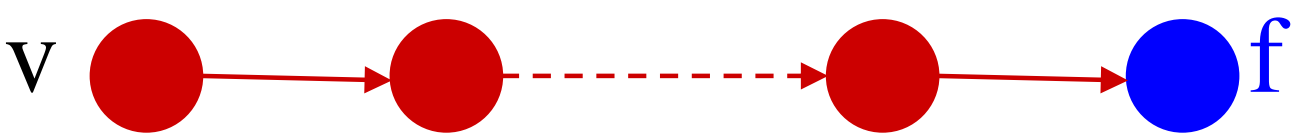
- Dijkstra’s algorithm is **greedy**: it makes a sequence of *locally* optimal moves, which results in the *globally* optimal solution.
    - Most algorithms don’t work like this - need to prove that it results in the global optimum.
- Specifically: It is not obvious that there cannot still be a shorter path to the Frontier node with smallest d-value.

Proof of Correctness: Invariant



The while loop in Dijkstra’s algorithm maintains a 3-part invariant:

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.



2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path

3. All edges leaving S go to F (or: no edges from S to Unexplored)

Proof of Correctness: Theorem

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
     }
   }
}
```

Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

Proof: Show that any other path from v to if has length >= f.d

Case 1: if v is in F, then S is empty and v.d = 0, which is trivially the shortest distance from v to v.

Proof of Correctness: Theorem

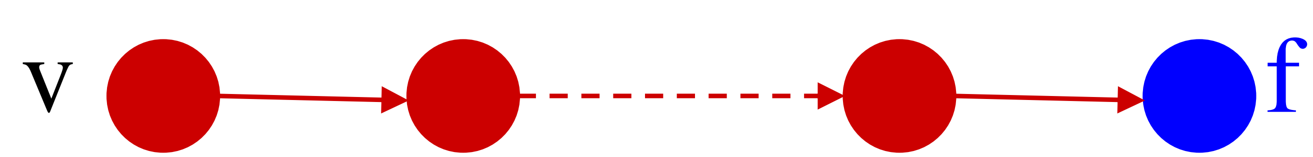
```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
     }
   }
}
```

Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

Proof: Show that any other path from v to if has length >= f.d

Case 2: v is in S. Part 2 of the invariant says:

- f.d is the length of the shortest path from v to f containing all settled nodes except f, and f.d is the length of such a path.



Proof of Correctness: Theorem

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

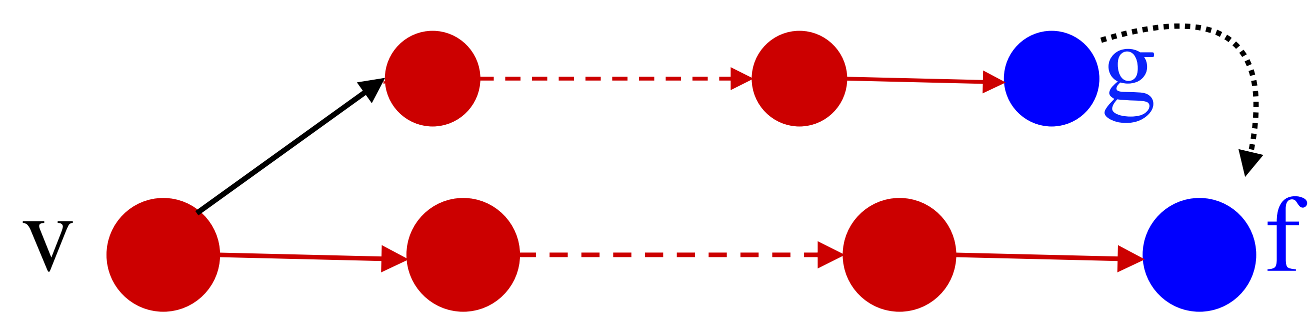
Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

Proof: Show that any other path from v to if has length >= f.d

Case 2: v is in S. Part 2 of the invariant says:

- f.d is the length of the shortest path from v to f containing all settled nodes except f, and f.d is the length of such a path.

Any other v-f path must either be longer or go through another frontier node g then arrive at f:



Proof of Correctness: Theorem

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

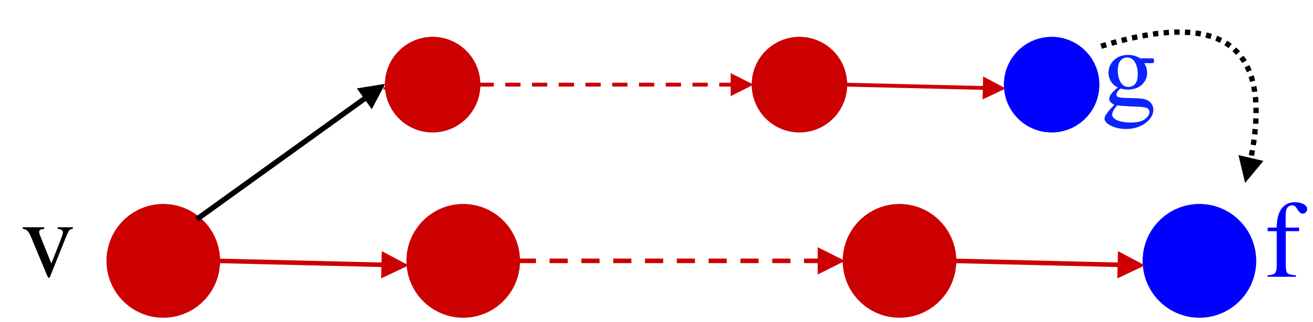
Proof: Show that any other path from v to if has length >= f.d

Case 2: v is in S. Part 2 of the invariant says:

- f.d is the length of the shortest path from v to f containing all settled nodes except f, and f.d is the length of such a path.

Any other v-f path must either be longer or go through another frontier node g then arrive at f:

d.f <= d.g,
so that path cannot be shorter



Proof of Correctness: Invariant Maintenance

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
          w.d =  f.d + weight(f, w);
          add w to F;
      } else if (f.d+weight(f,w) < w.d) {
          w.d = f.d+weight(f,w);
      }
   }
}
```

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.

2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path

3. All edges leaving S go to F (or: no edges from S to Unexplored)

Proof of Correctness: Invariant Maintenance

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.
2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path
3. All edges leaving S go to F (or: no edges from S to Unexplored)

At initialization:

1. S is empty; trivially true.
2. v.d = 0, which is the shortest path.
3. S is empty, so no edges leave it.

Proof of Correctness: Invariant Maintenance

```
S = { }; F = {v}; v.d = 0;
while (F ≠ {}) {
  f = node in F with min d value;
  Remove f from F, add it to S;
  for each neighbor w of f {
    if (w not in S or F) {
      w.d = f.d + weight(f, w);
      add w to F;
    } else if (f.d+weight(f,w) < w.d) {
      w.d = f.d+weight(f,w);
    }
  }
}
```

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.
2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path
3. All edges leaving S go to F (or: no edges from S to Unexplored)

At each iteration:

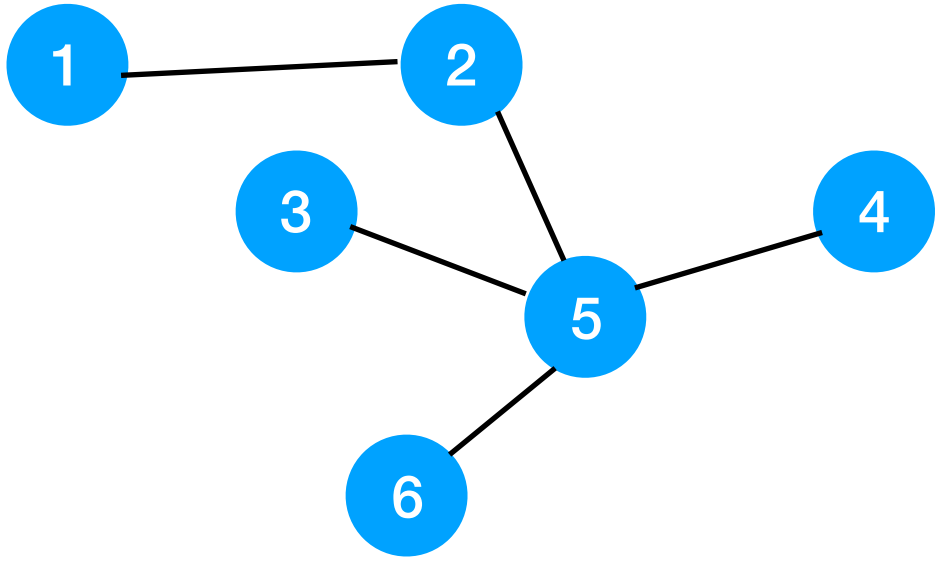
1. Theorem says f.d is the shortest path, so it can safely move to S
2. Updating w.d maintains Part 2 of the invariant.
3. Each neighbor is either already in F or gets moved there.

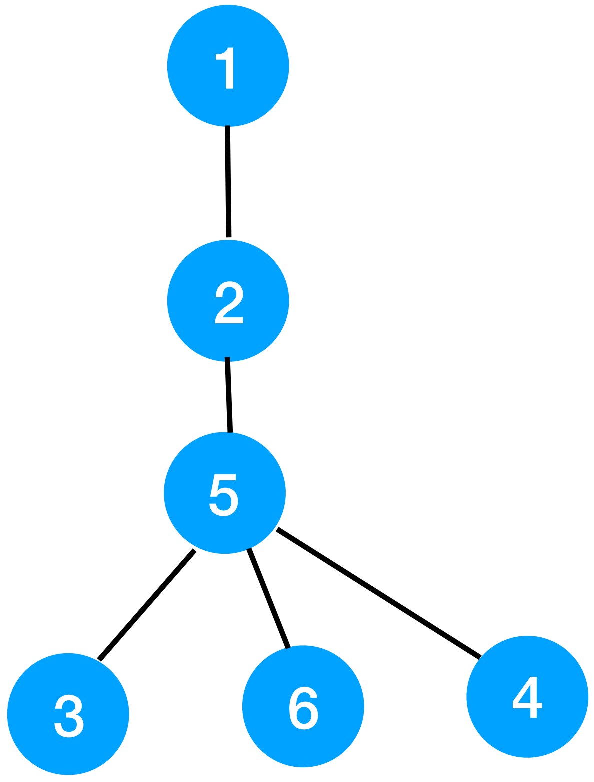
Questions?

Drawing Graphs

- The same graph can be drawn (infinitely!) many different ways.

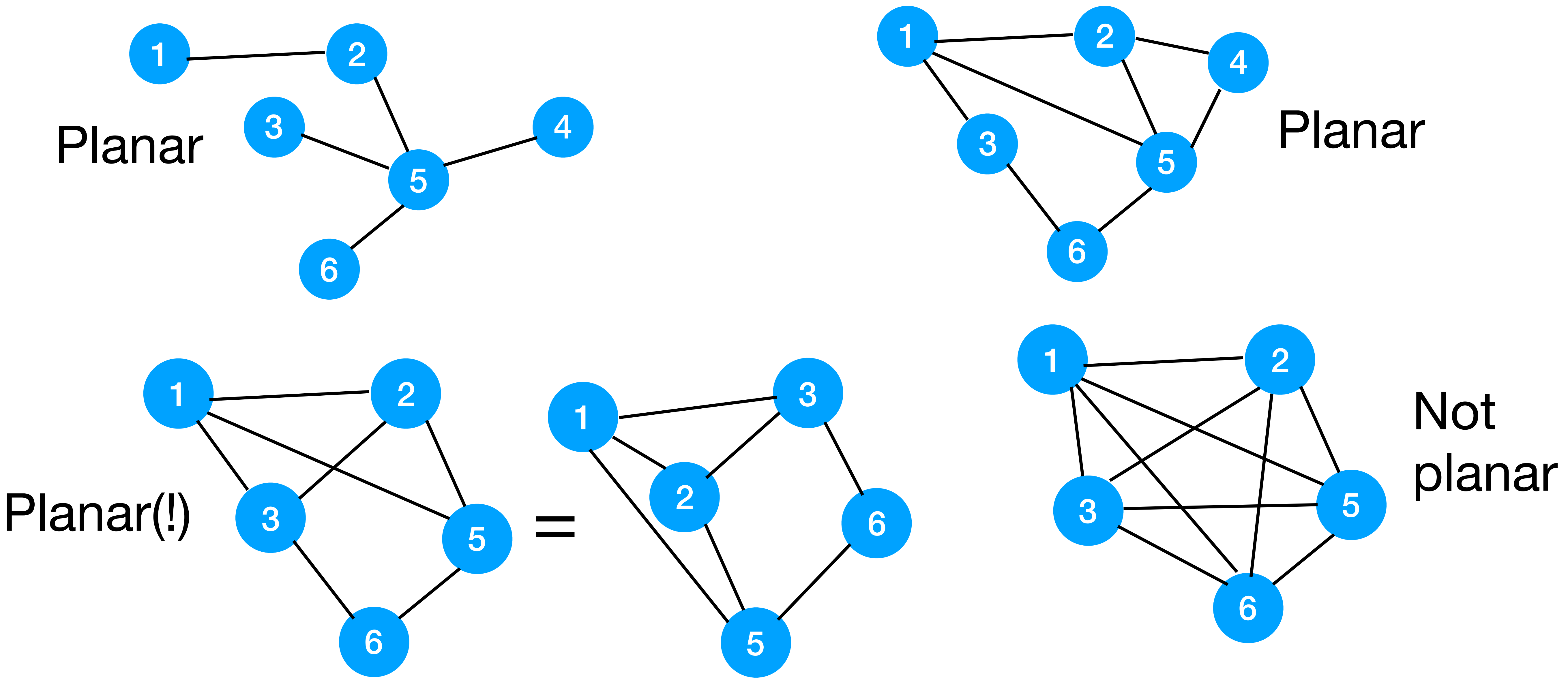
V = {1,2,3,4,5,6}
E = {(1,2), (2,5), (3,5)
(4,5), (5,6)}





Planarity

- If a graph can be drawn without crossing edges, it is planar.

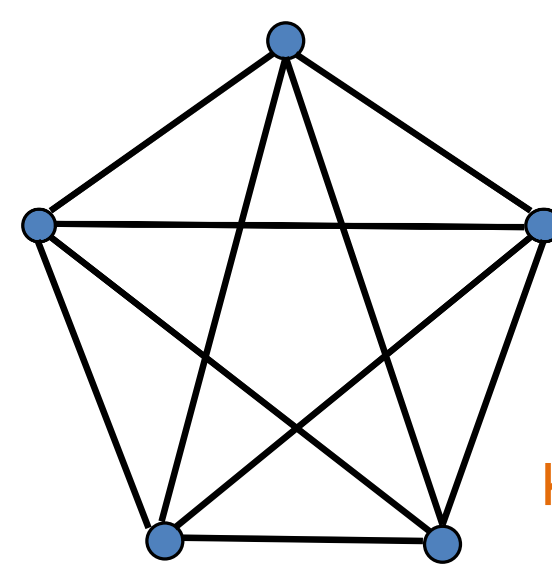


Detecting Planarity

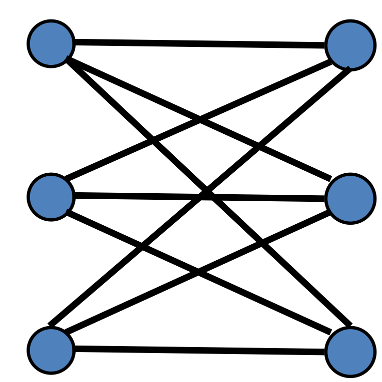
A subgraph of a graph is a graph whose vertex and edge sets are subsets of the larger graph's.

- Elements of the edge subset can only contain nodes in the vertex subset.

- There's a (non-obvious) theorem that says a graph is planar if and only if it does not contain* one of these as a subgraph:



K_5



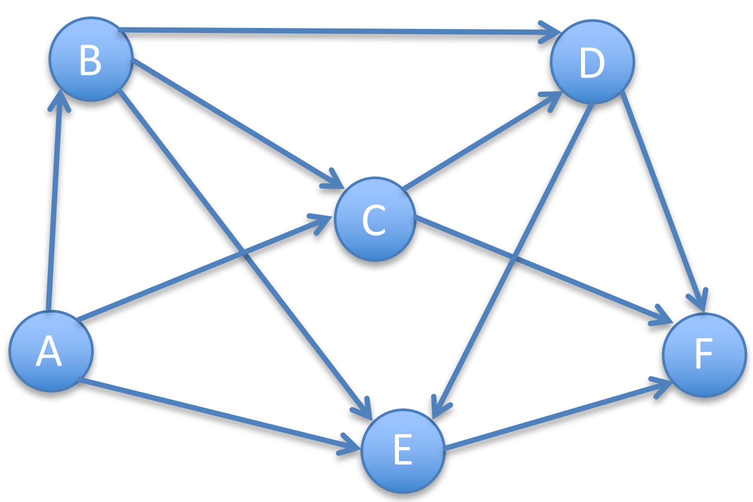
$K_{3,3}$

*The definition of "contain" is slightly more general than having one of these directly as a subgraph.

DAGs

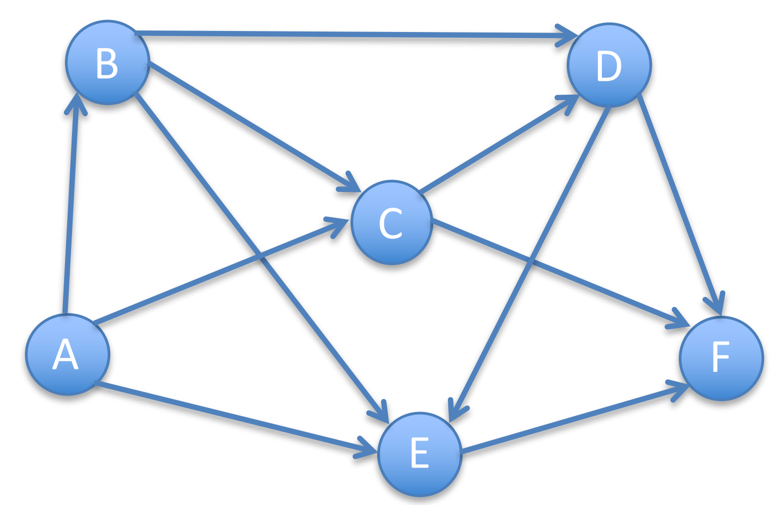
- A DAG, or Directed Acyclic Graph is a…

  graph that is directed and acyclic.



Is this a DAG?

- How do we tell if a directed graph is acyclic?
  - If a node has indegree 0, it can't be part of a cycle.
  - Edges coming from that node also can't be part of a cycle.



Algorithm:

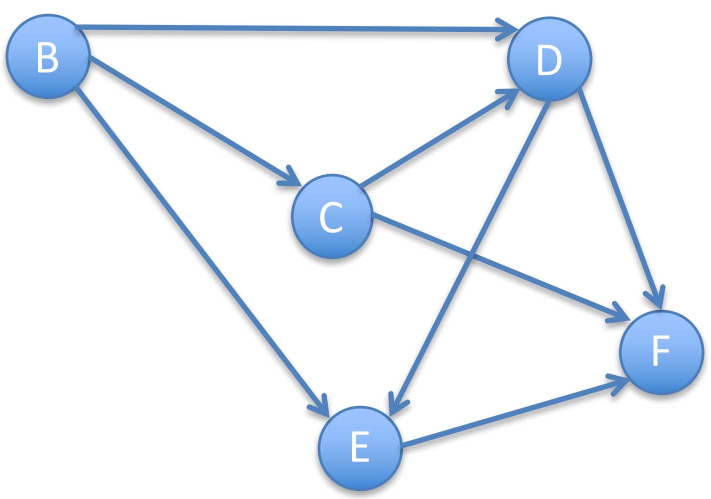
while there is a node with indegree 0:

delete the node and all edges coming from it

if the graph is empty, the original graph was a DAG

Is this a DAG?

- How do we tell if a directed graph is acyclic?
  - If a node has indegree 0, it can't be part of a cycle.
  - Edges coming from that node also can't be part of a cycle.



Algorithm:

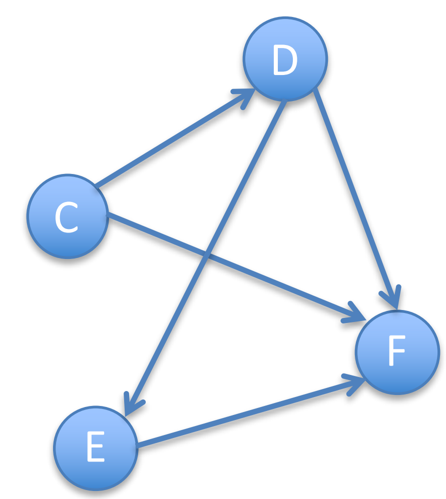
while there is a node with indegree 0:

delete the node and all edges coming from it

if the graph is empty, the original graph was a DAG

Is this a DAG?

- How do we tell if a directed graph is acyclic?
  - If a node has indegree 0, it can’t be part of a cycle.
  - Edges coming from that node also can’t be part of a cycle.



Algorithm:

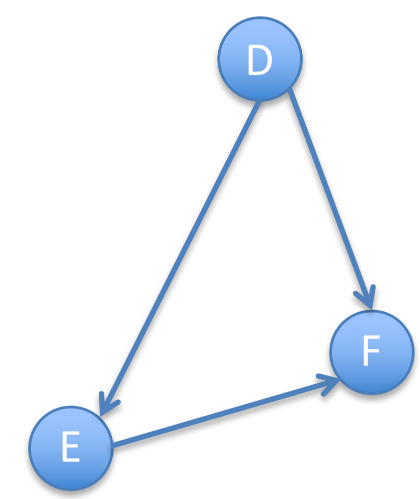
while there is a node with indegree 0:

delete the node and all edges coming from it

if the graph is empty, the original graph was a DAG

Is this a DAG?

- How do we tell if a directed graph is acyclic?
  - If a node has indegree 0, it can't be part of a cycle.
  - Edges coming from that node also can't be part of a cycle.



Algorithm:

while there is a node with indegree 0:

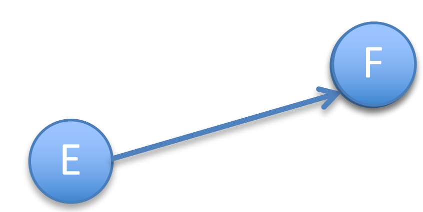
delete the node and all edges coming from it

if the graph is empty, the original graph was a DAG

Is this a DAG?

- How do we tell if a directed graph is acyclic?
  - If a node has indegree 0, it can't be part of a cycle.
  - Edges coming from that node also can't be part of a cycle.

Algorithm:



while there is a node with indegree 0:

  delete the node and all edges coming from it

if the graph is empty, the original graph was a DAG

Is this a DAG?

- How do we tell if a directed graph is acyclic?
  - If a node has indegree 0, it can’t be part of a cycle.
  - Edges coming from that node also can’t be part of a cycle.

Algorithm:

while there is a node with indegree 0:

  delete the node and all edges coming from it

if the graph is empty, the original graph was a DAG

Topological Sort

Topological sort (or toposort):

i = 0

while there is a node with indegree 0:

delete* the node and all edges coming from it

label* the deleted node i

increment i

if the graph is empty, the original graph was a DAG

Topological Sort

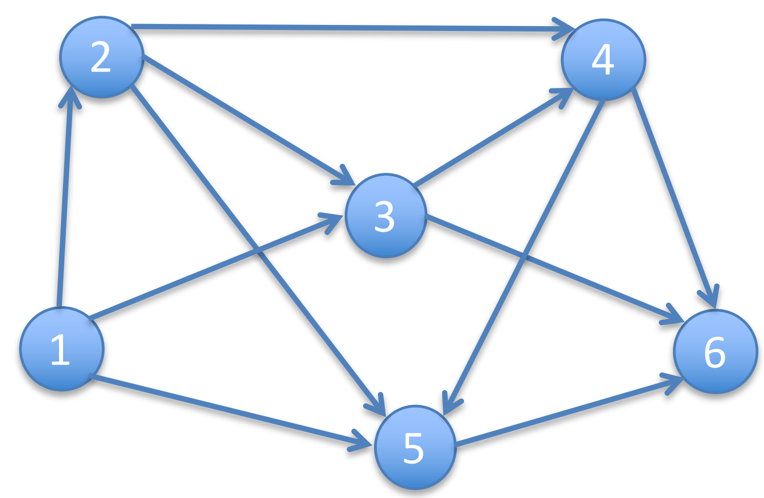
Topological sort (or toposort):

```
i = 0
while there is a node with indegree 0:
    delete* the node and all edges coming from it
    label* the deleted node i
    increment i
if the graph is empty, the original graph was a DAG
```

*This is pseudocode: we probably don't want to actually modify the graph. We'll need to store extra data with nodes and edges, and possibly overlay additional data structures to make it efficient.

Topological Sort

- Here are the labels we applied to the example graph:

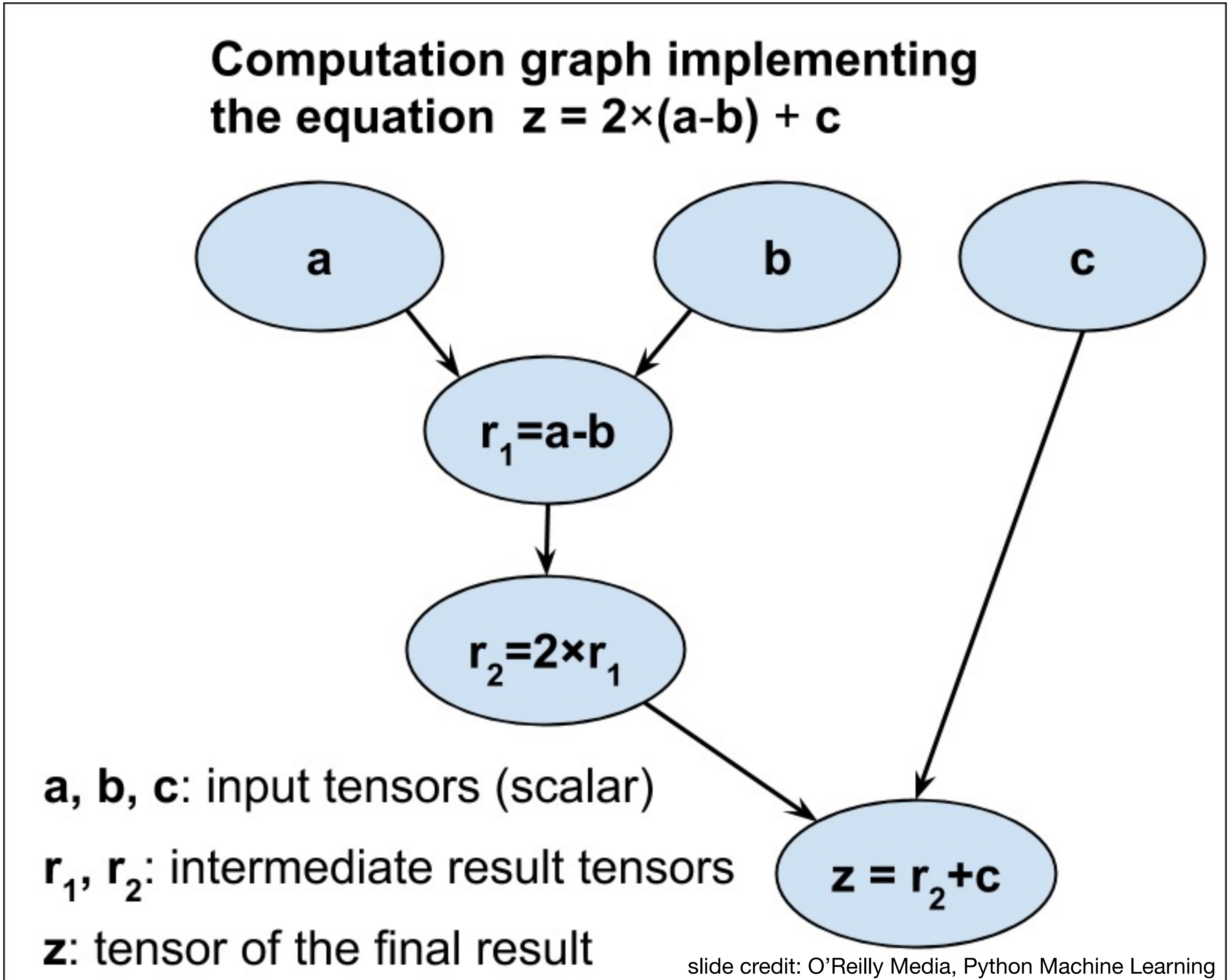


- Property: all edges go from a lower-numbered node to a higher-numbered node.

- Useful for dependency resolution, job scheduling,

- Ordering is not necessarily unique: could have chosen from among multiple nodes with indegree 0.

Tensorflow Computation Graphs



slide credit: O'Reilly Media, Python Machine Learning