Graphs

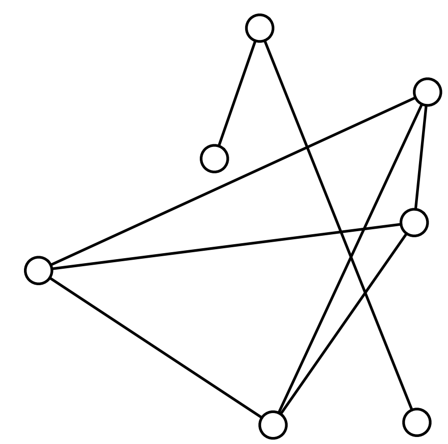
Thomas Schwarz, SJ

Graph Definition

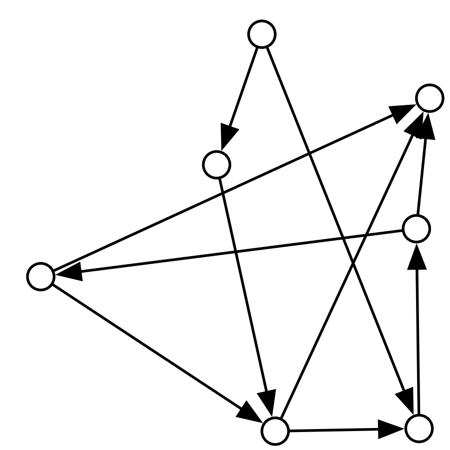
- A graph has a set of vertices V and a set of edges.
  - Directed edges are pairs (u, v) with $u, v \in V$
  - Undirected edges are two-sets $\{u, v\}$ with $u, v \in V$
- A graph with directed edges is called a directed graph
- A graph with undirected edges is just called a graph

Graph Definition

- Graphs are represented by:
  - drawing the vertices as small circles
  - drawing the edges as edges
- Directed edges are drawn as arrows



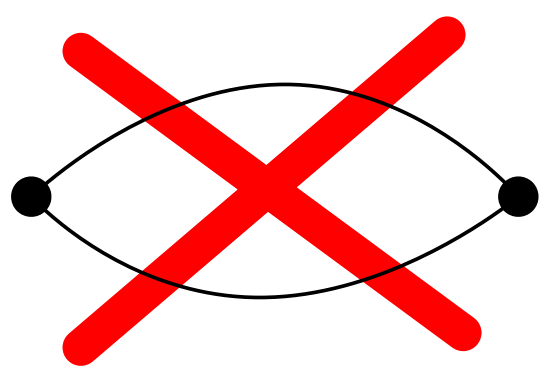
An undirected graph with 7 vertices and 7 edges

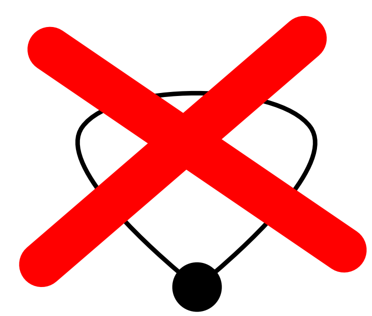


A directed graph

Graph Definition

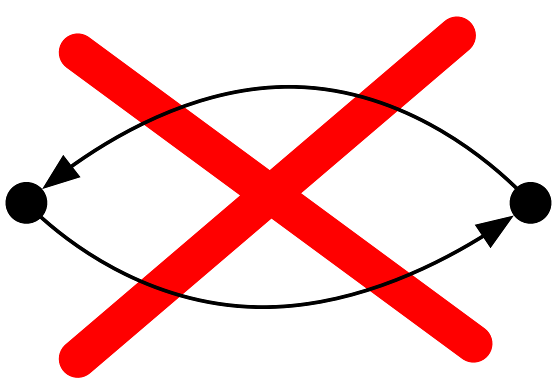
- Computer scientist sometimes differ from mathematicians in what is called a graph
  - In Mathematics, a(n undirected) graph can
    - Have only one edge at most between two vertices
    - Cannot have an edge to the same vertex

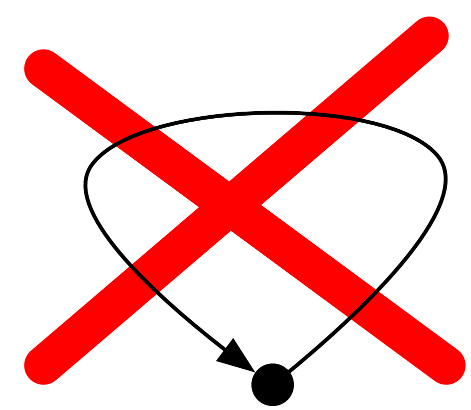




Graph Definition

- Computer scientist sometimes differ from mathematicians in what is called a graph
  - In Mathematics, a directed graph can
    - Have only one edge at most between two vertices
    - Cannot have an edge to the same vertex





Graph Definition

- Mathematicians call a graph that allows multiple edges between the same pair of vertices
    - a multigraph

Graph Representations

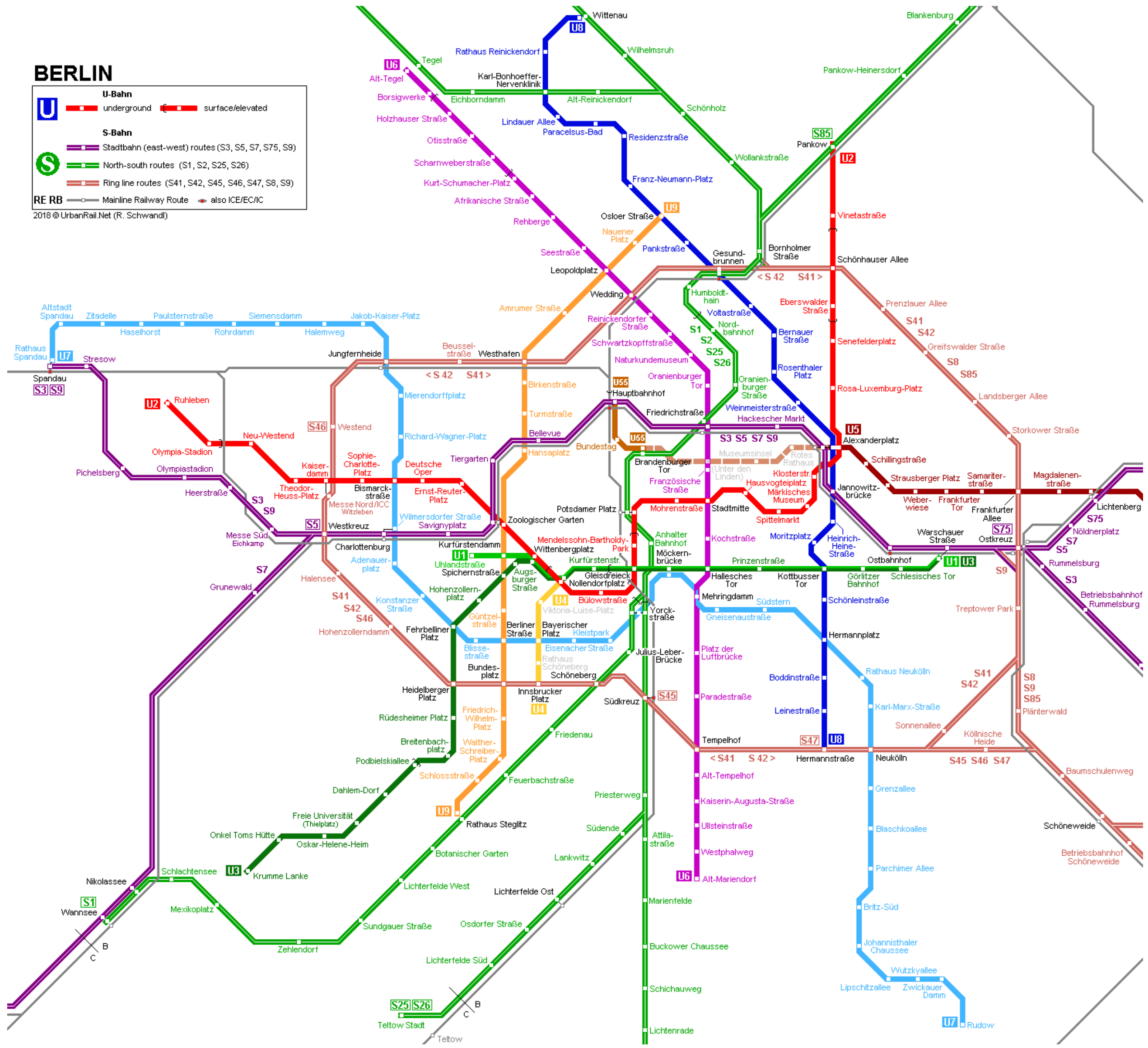
- To understand graphs, we can use:
  - The visual representation
    - E.g. The neighbor graph
    - Take a political map

Graph Representations

- Examples:
  - Place a vertex in every entity (state, not DDFF)
  - Connect vertices if the entities have a common border

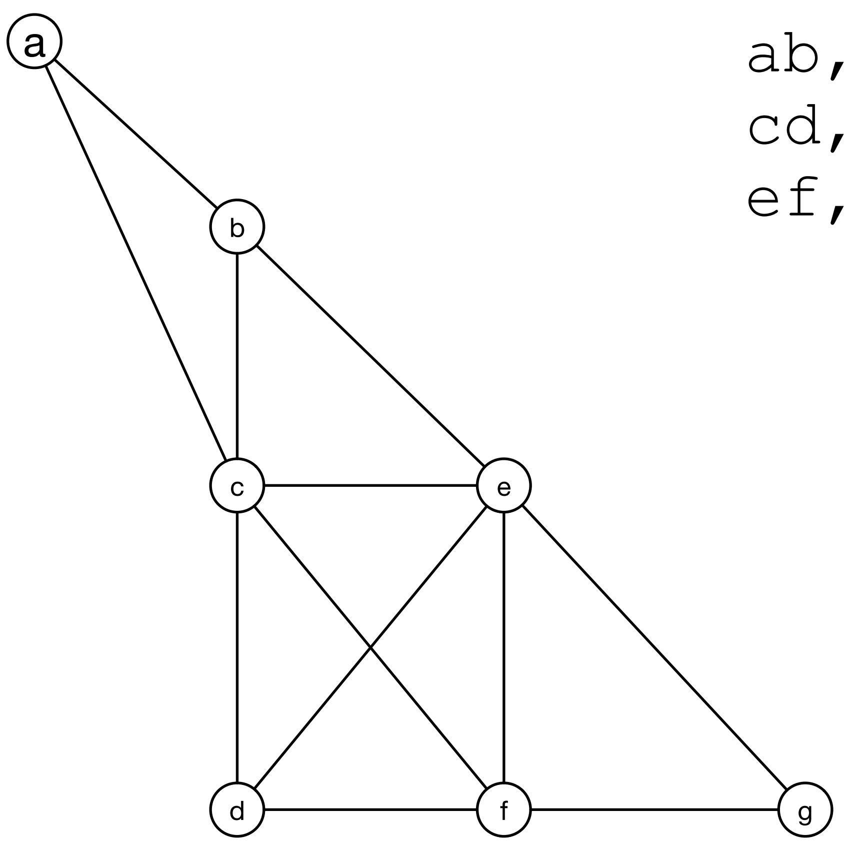
Graph Representations

- Vertices are stations
- Edges represent a connection via underground or light rail
- This is <u>multi-graph</u> because several edges can connect a station

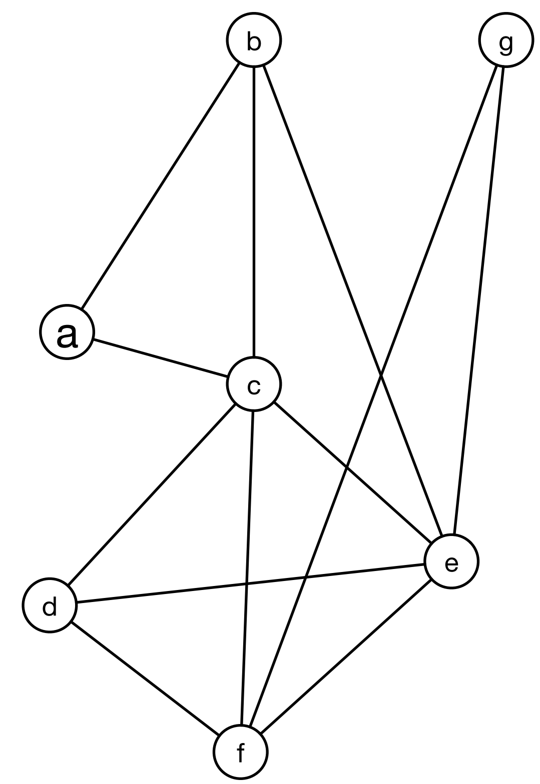


Graph Definition

- Different visualizations can still give you the same graph, as you can see from the examples below

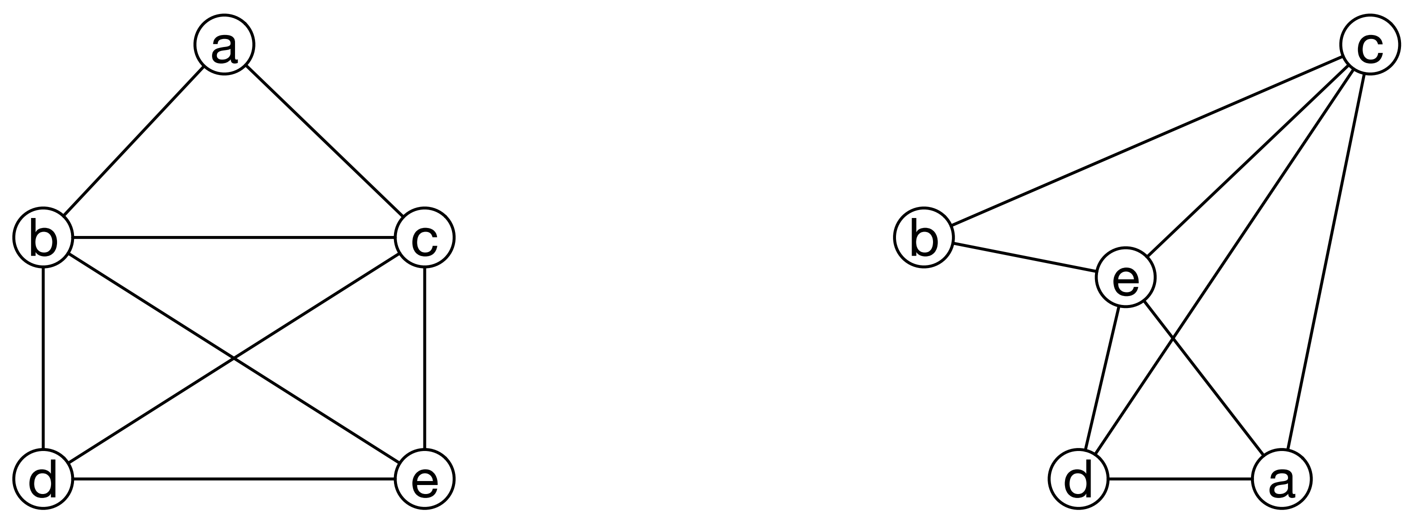



```
ab, ac, bc, ce
cd, cf, de, df
ef, eg, fg
```



Graph Definition

- Two graphs are isomorphic, if there is a renaming of the vertices that converts one into the other and vice versa
  - Mathematically, a renaming is a bijection



  - These two do not look the same, but they are isomorphic: $a \to b,\ b \to c,\ c \to e,\ d \to d,\ e \to a$

Graph Definition

- Two graphs are isomorphic, if there is a renaming of the vertices that converts one into the other and vice versa

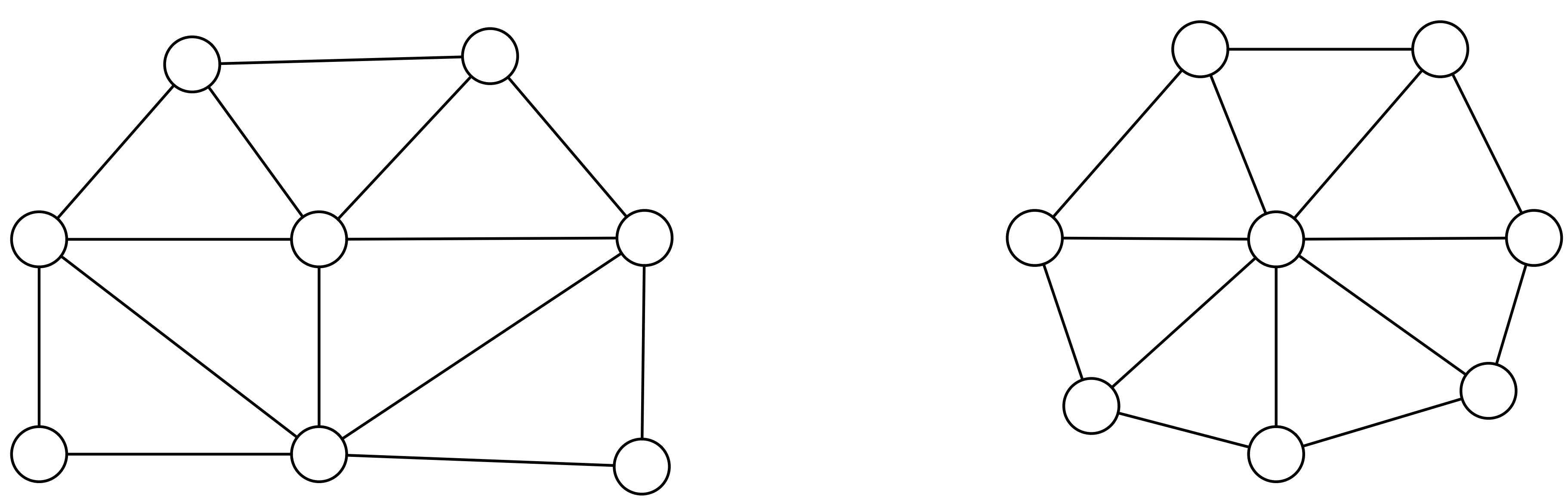
$G = (V, E)$ is isomorphic to $G' = (V', E')$

$$\Leftrightarrow$$

$\exists f : V \to V'$ bijection $: \forall v_1, v_2 \in V : (f(v_1), f(v_2)) \in E' \Leftrightarrow (v_1, v_2) \in E$

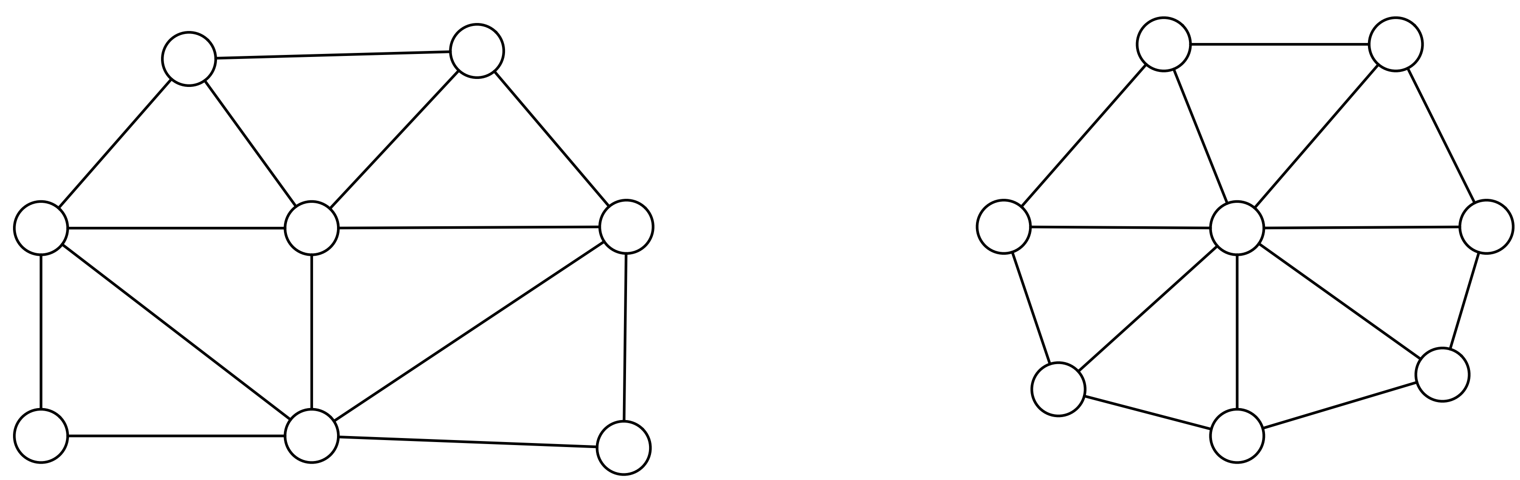
Graph Definition

- Determining whether two graphs are isomorphic is a known, difficult question
  - Some results are easy, e.g. vertices of the same *rank* (the number of edges adjacent to a vertex) need to be mapped to vertices of the same rank
  - So, these two graphs ***cannot*** be isomorphic



Graph Definition

- These two graphs ***cannot*** be isomorphic



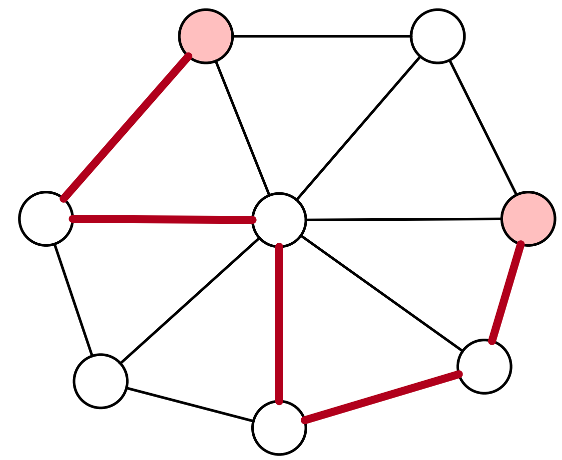
- The left graph has two vertices of degree 2
- The right graph has no vertices of degree 2
- But the number of vertices and edges is equal

Graph Definitions

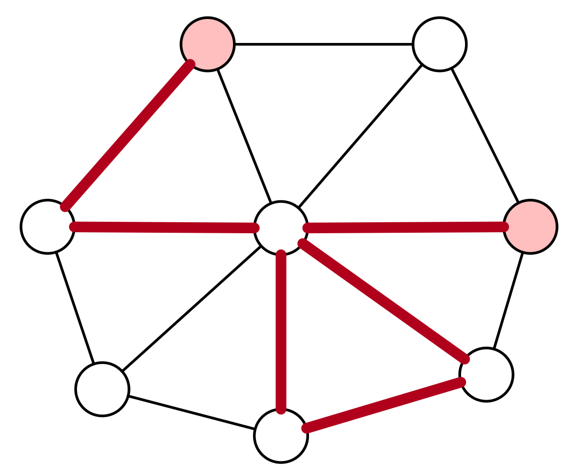
- There are a number of important properties of graphs
  - No need to learn them by heart, the ones used in CS will get repeated over and over again
    - A path between two vertices $u, w \in V$ of a graph $G = (V, E)$ is a list of vertices $u = v_0, v_1, \ldots, v_{n-1}, v_n = w$ such that there is an edge between all v_i and v_{i+1}
    - Furthermore, no vertices can be repeated

Graph Definitions

- Example for a path:
  - Has length 5 (number of edges)



- Example for a walk that is not a path
  - We visit the center vertex twice

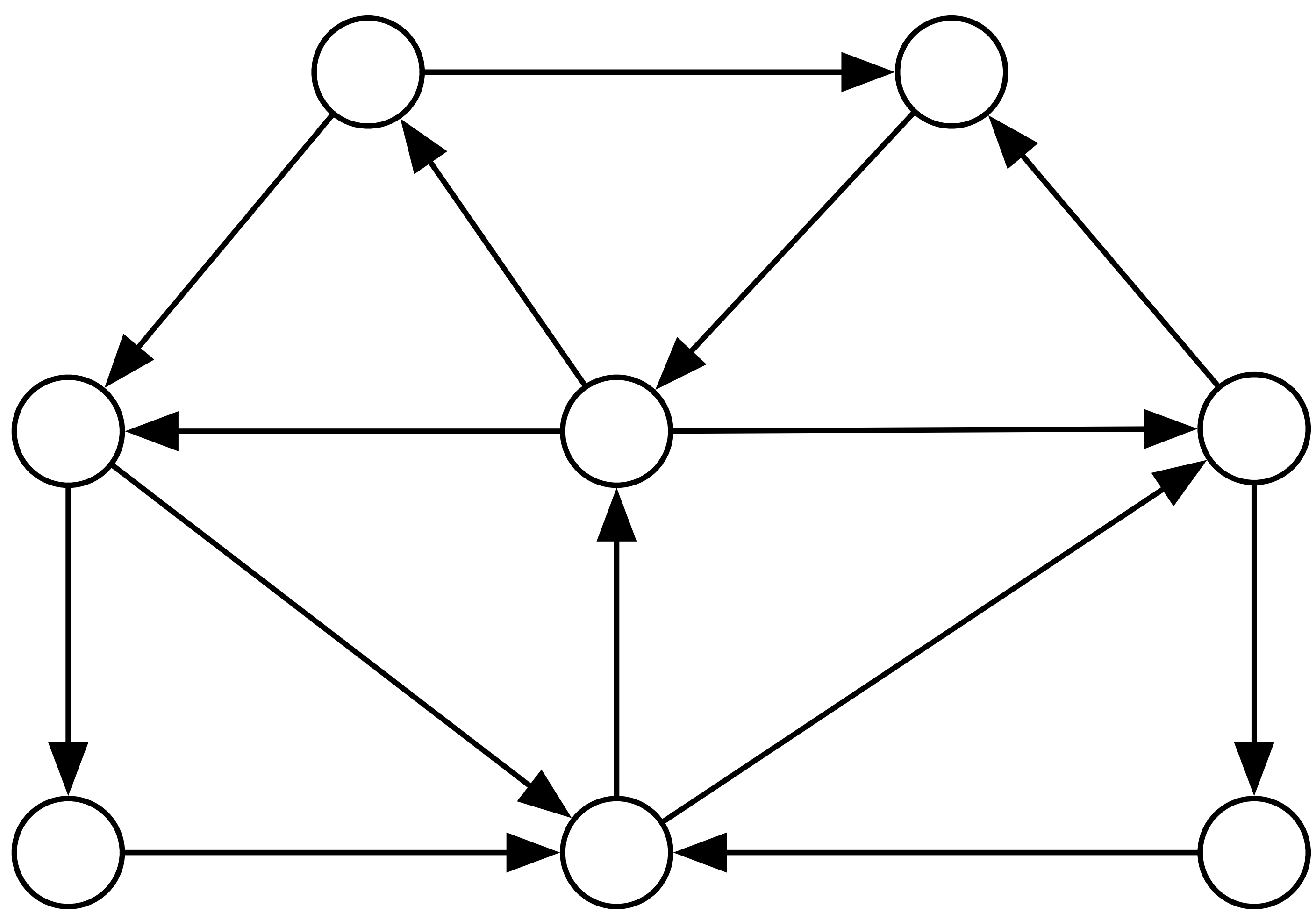


Graph Definitions

- For directed graphs, the paths need to follow the arrow

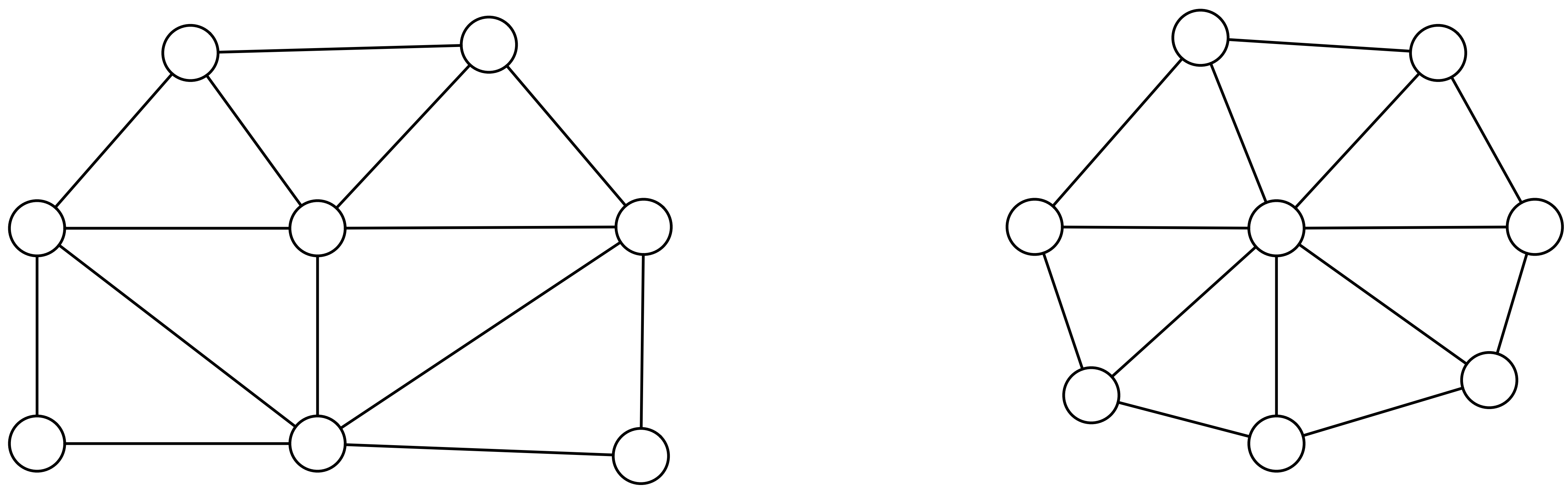
Graph Definitions

- A directed graph (digraph) is strongly connected if there is a path from every vertex to every other vertex



Graph Definitions

- An undirected graph is connected if there is a path from every vertex to every other vertex
  - This is not a connected graph



  - But it consists of two connected components

Graph Definitions

- Interesting question
  - Is the friends graph on facebook connected
    - The "friend" relation is mutual, so all users are vertices and there is an edge if two users are in a friends-relation
  - Probably not, because we signed up my mom on facebook and she did not like it, so she is no longer friends with anyone
  - But how about "active users"
    - Could there be a republican and a democratic facebook
      - No, but maybe there are isolated groups

Euler Tours

- An Euler tour is a closed tour that traverses each edge of the graph only once.
  - Graphs with an Euler tour are called Eulerian
- Theorem: An undirected, connected graph is Eulerian if each vertex has even degree.
  - Recall: Degree is the number of edges of the vertex

Euler Tours

- Königsberg bridge problem
  - Königsberg had seven bridges over the river Pregel
  - Is it possible to have an afternoon walk crossing all bridges exactly once

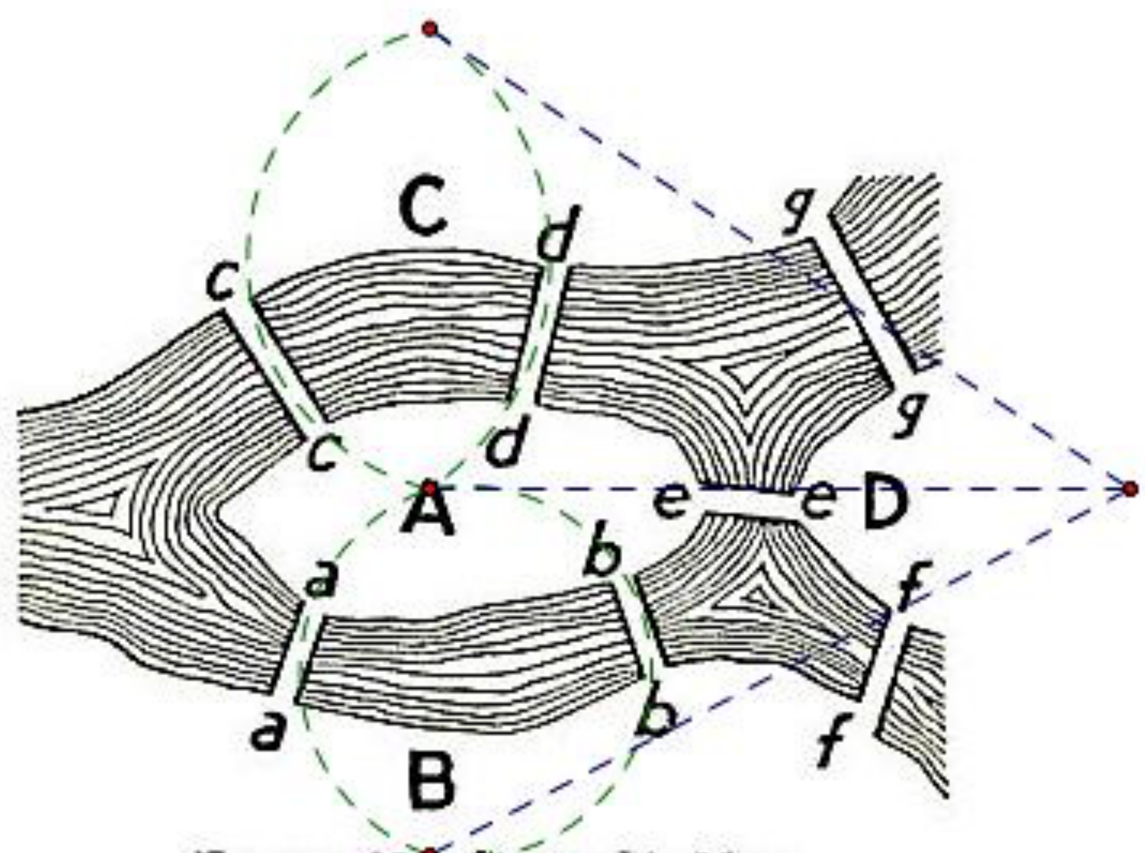


FIGURE 98. *Geographic Map: The Königsberg Bridges.*

Euler Tours

- Solved by Euler
    - Translate into a multi-graph (multiple edges allowed)

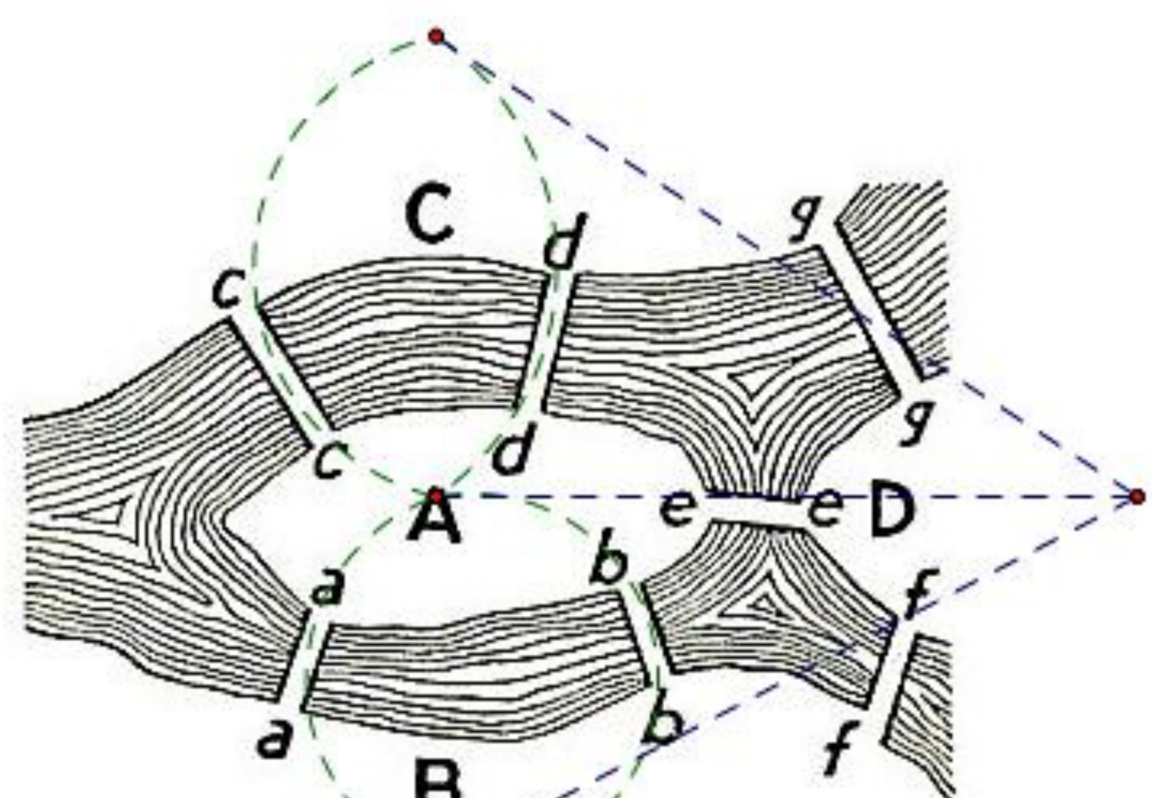
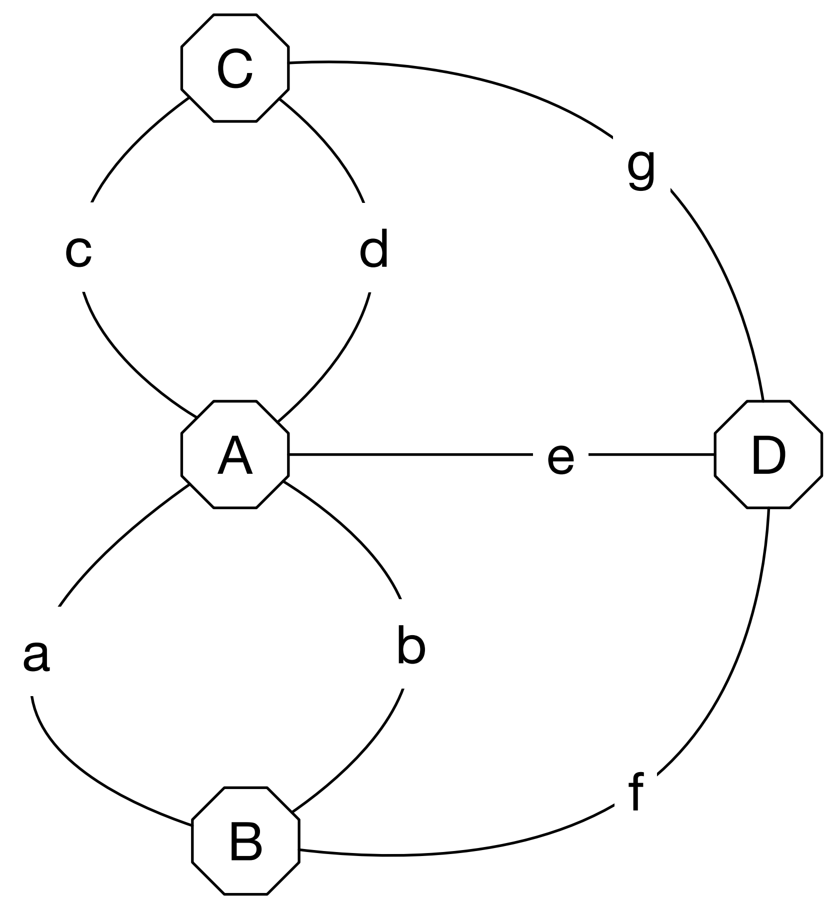
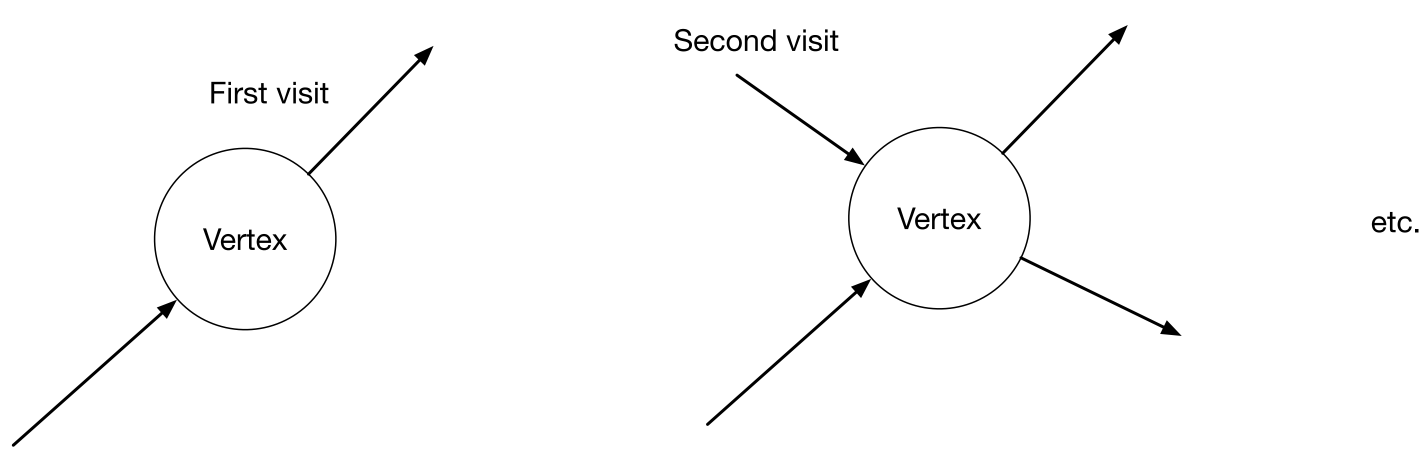


FIGURE 98. *Geographic Map: The Königsberg Bridges.*



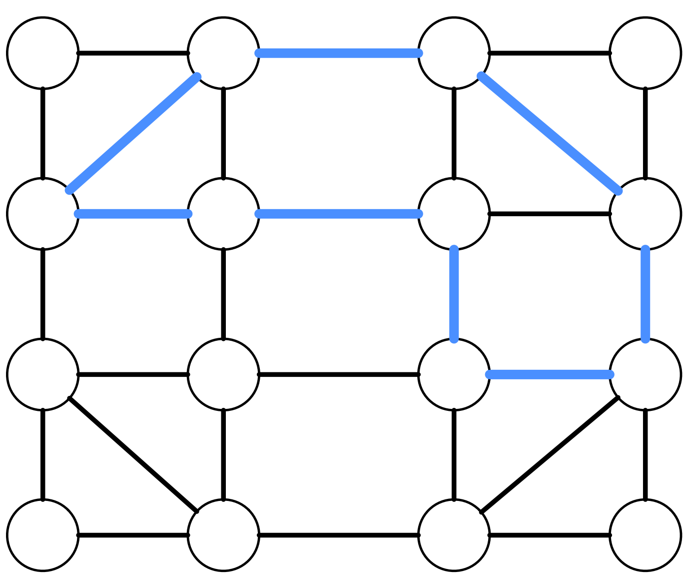
Euler Tours

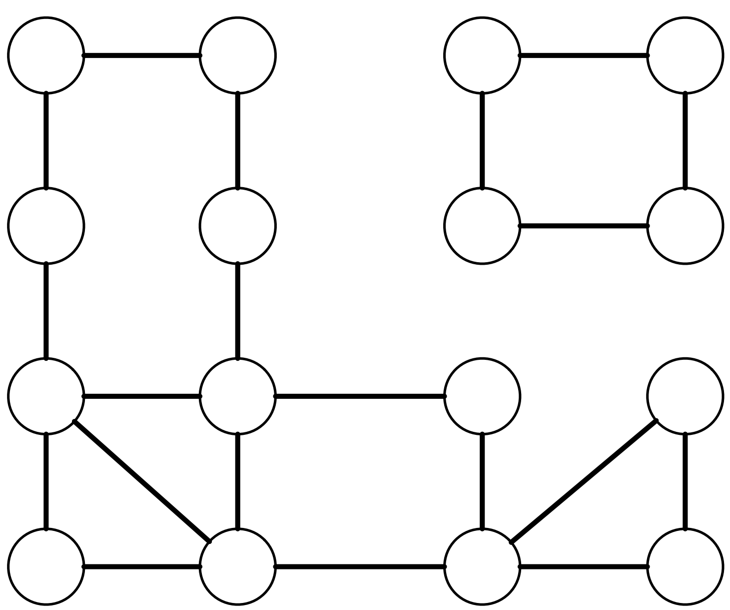
- Actually, <u>all</u> edges have odd degree, so such a tour is not possible
- To show that the theorem is correct:
    - Euler tour exists implies all vertex degrees are even
        - Because an Euler tour visits all edges and every time it visits an edge, it needs to come and to go.



Euler Tours

- Other direction can be shown using *Fleury's algorithm*
  - Key observation:
    - If we remove the edges from a closed tour
      - (starts and ends at the same vertex)
    - then in the remaining graph all vertices have still even degree



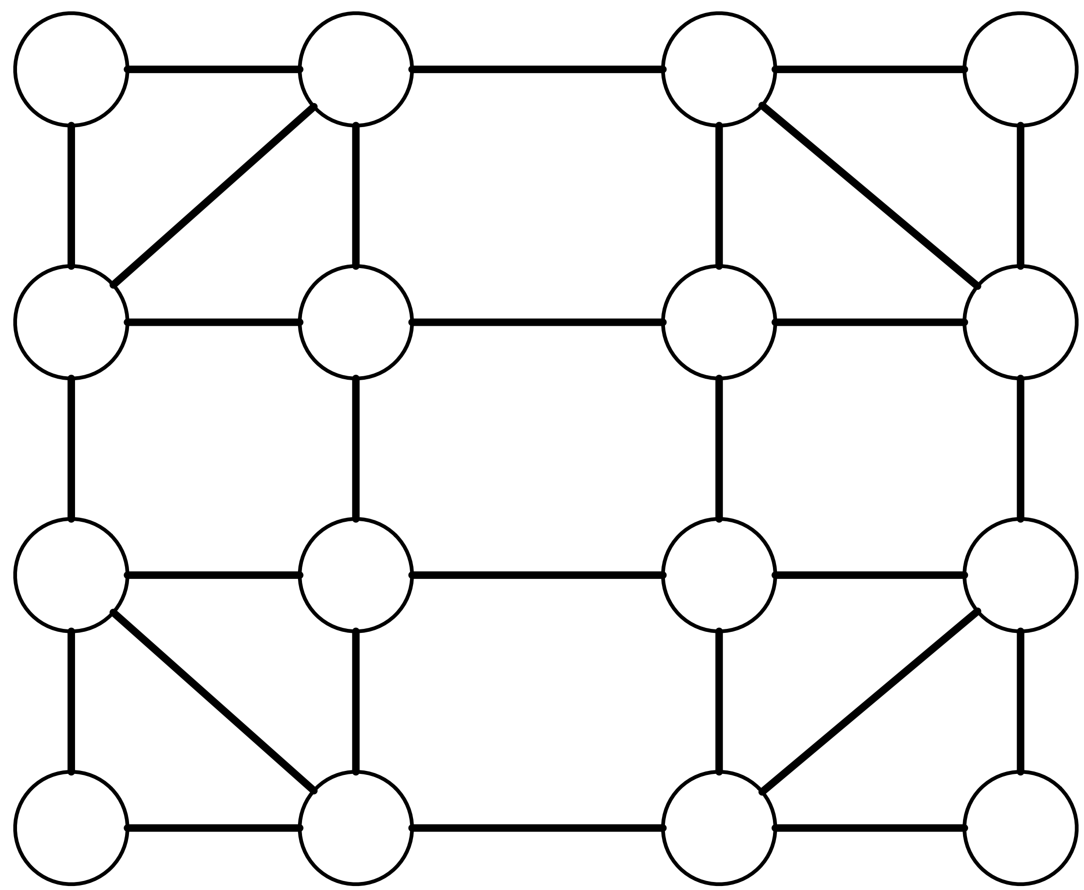


Euler Tours

- Fleury's algorithm:
  - Start at a node and walk anywhere, marking the edge
  - Leave the node that you arrived at
  - Continue until you can no longer find an unused edge
    - At this point, you are back in the starting vertex
  - If any of the vertices visited has a unused edges, start with that edge until you are back at that edge.
  - Splice the new circuit into the old one

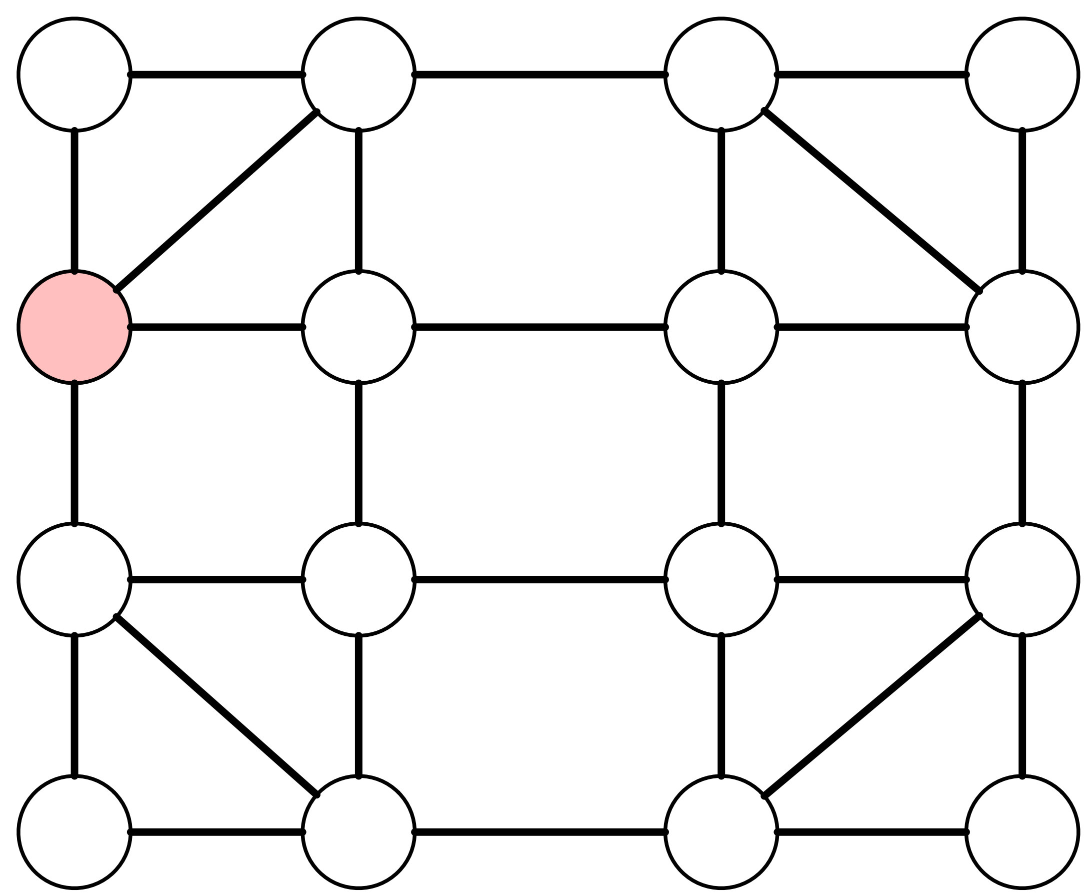
Euler Tours

- Example



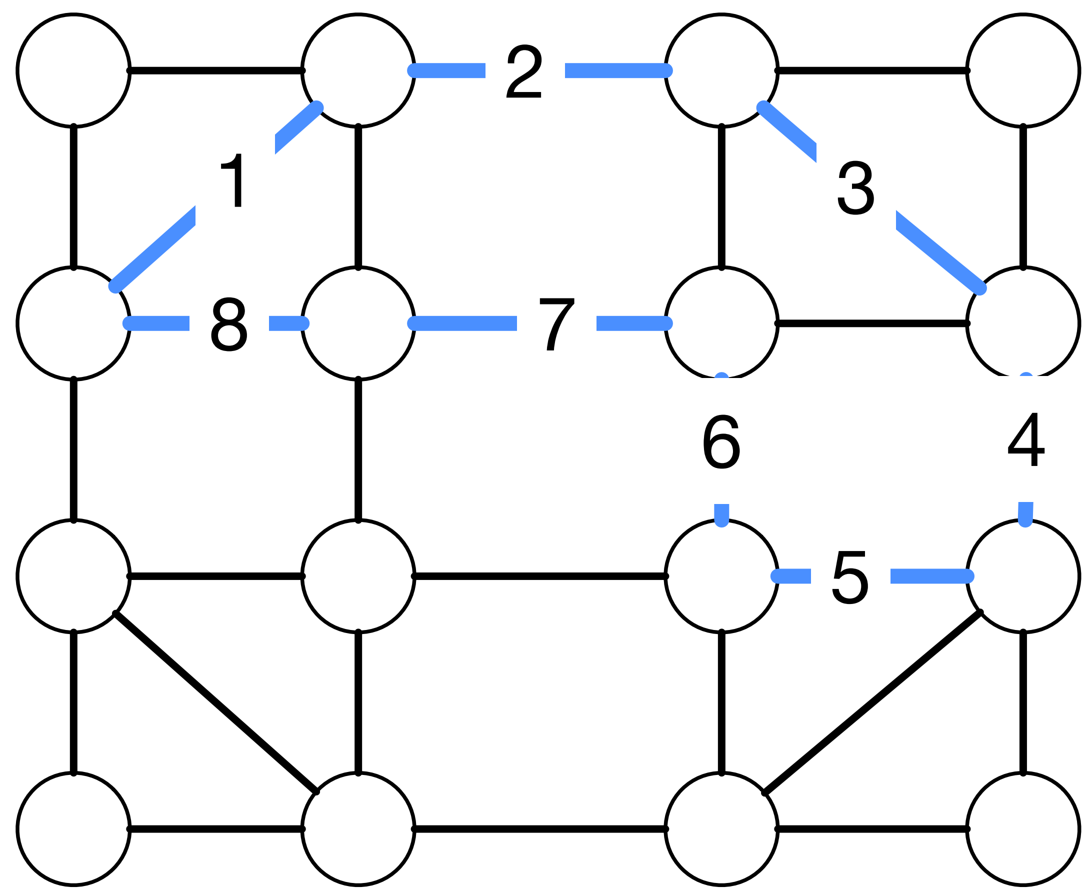
Euler Tours

- Start at a random vertex



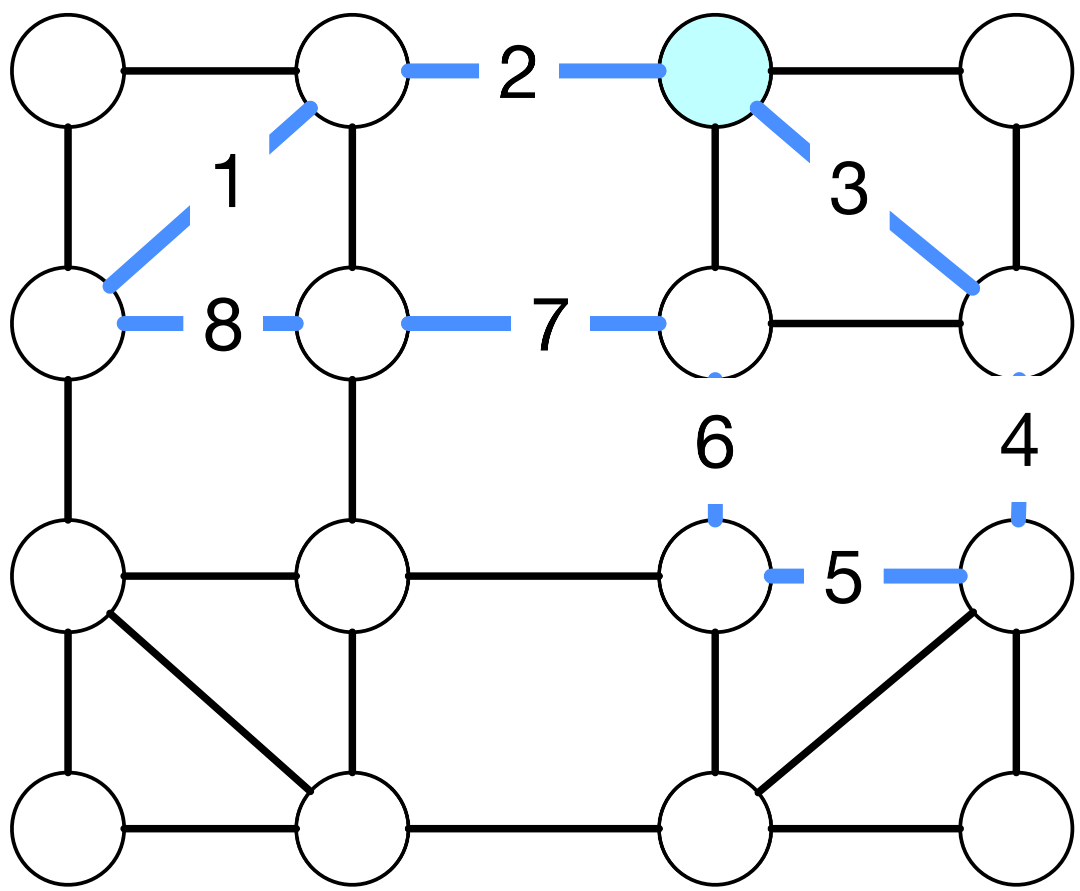
Euler Tours

- Make a tour



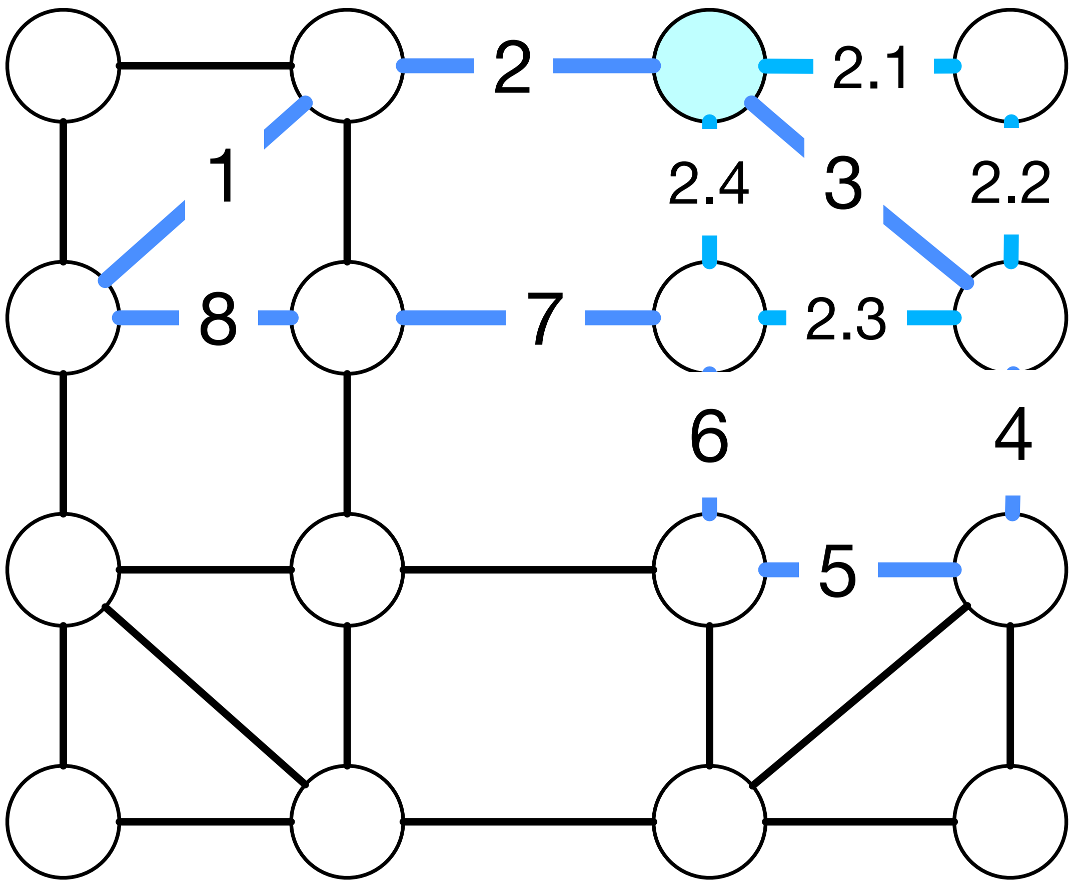
Euler Tours

- Check for vertices with unused edges and pick a random one



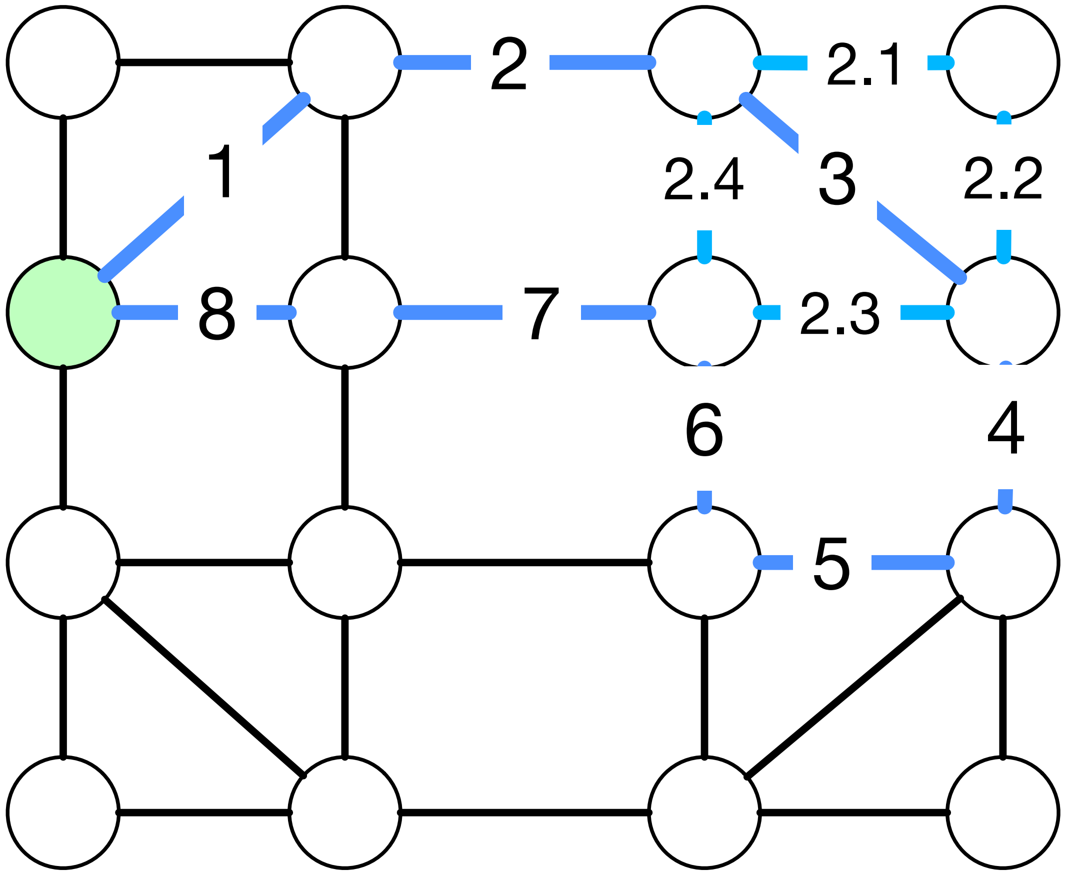
Euler Tours

- Start out creating a random circuit of unused edges



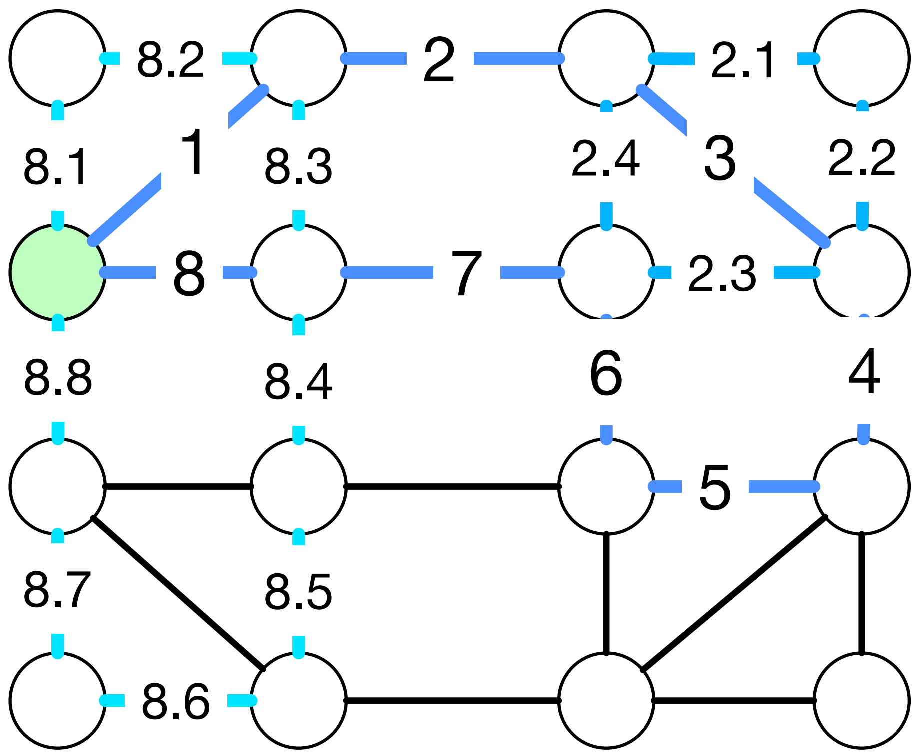
Euler Tours

- Pick another vertex with unused edges



Euler Tours

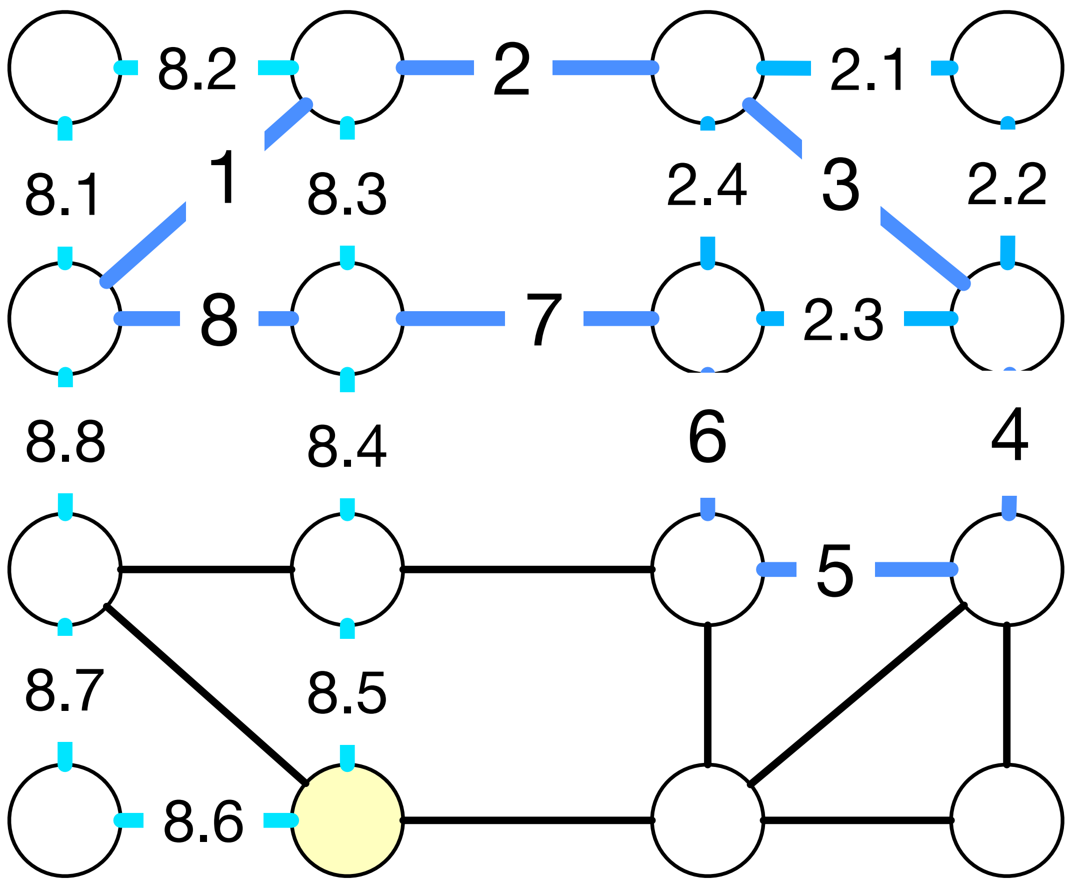
- Start a new part of the circuit



- Circuit so far: 1, 2, 2.1, 2.2, 2.3, 2.4, 3, 4, 5, 6, 7, 8, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8

Euler Tours

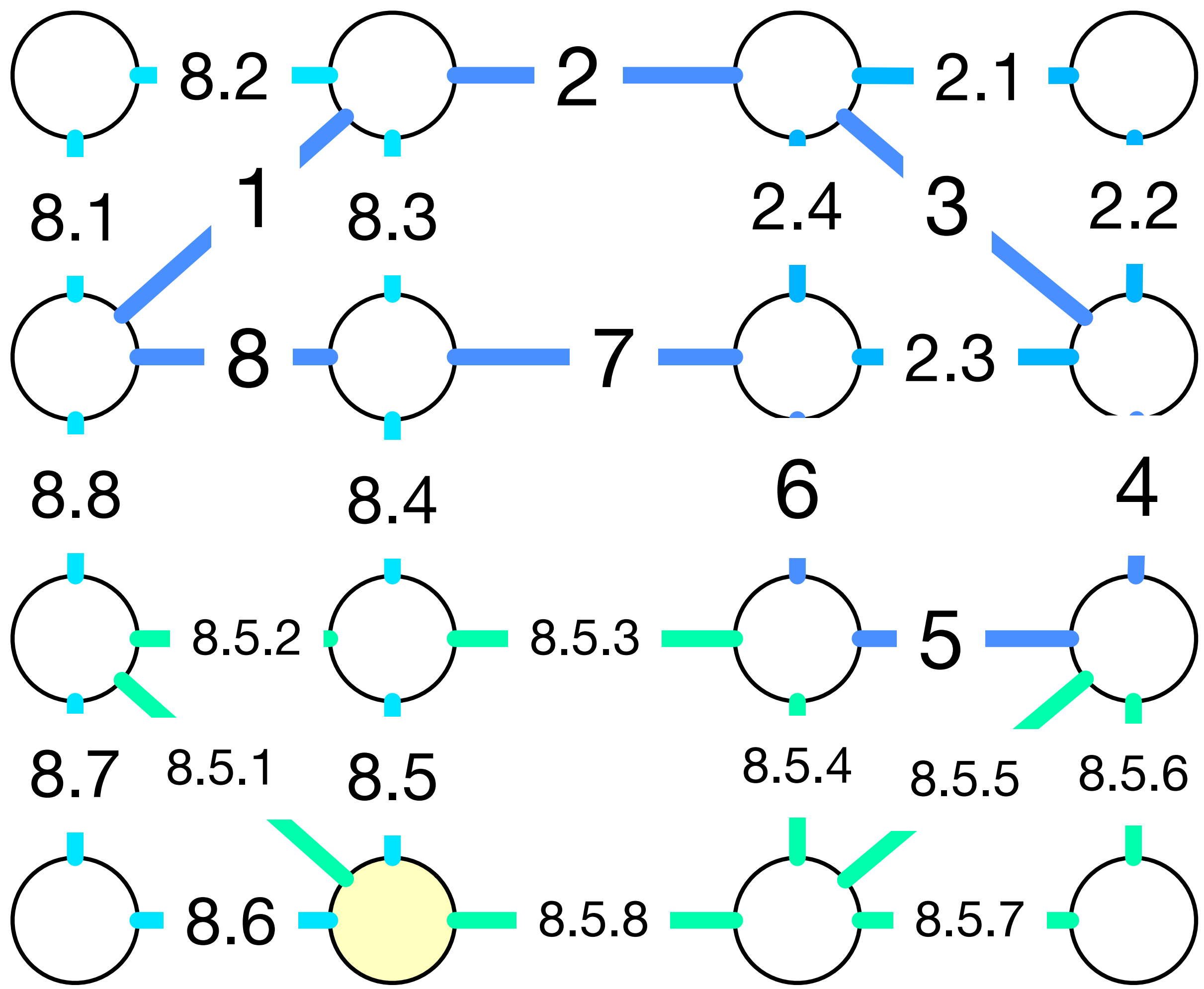
- In the new circuit, there are still vertices without all edges used.
- Pick one



- Circuit so far: 1, 2, 2.1, 2.2, 2.3, 2.4, 3, 4, 5, 6, 7, 8, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8

Euler Tours

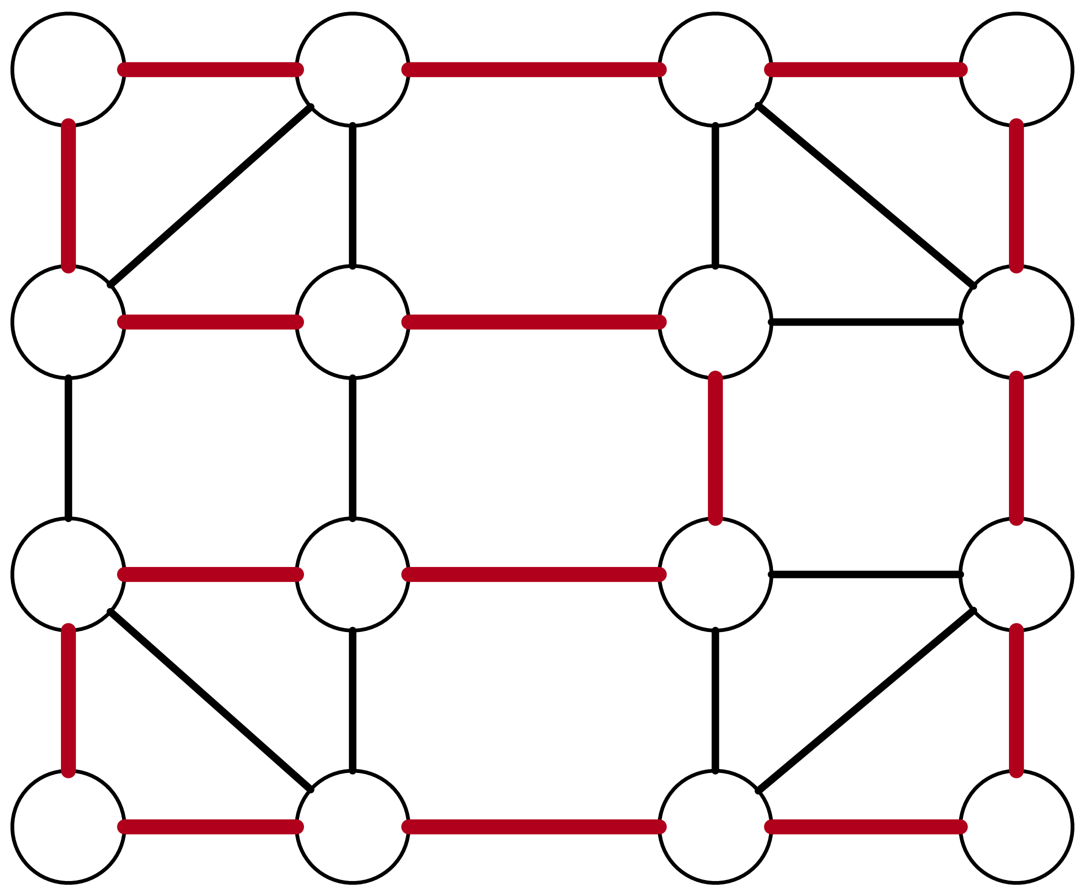
- And after this, we are done



- Circuit is: 1, 2, 2.1, 2.2, 2.3, 2.4, 3, 4, 5, 6, 7, 8, 8.1, 8.2, 8.3, 8.4, 8.5, 8.5.1, 8.5.2, 8.5.3, 8.5.4, 8.5.5, 8.5.6, 8.5.7, 8.5.8 8.6, 8.7, 8.8

Hamiltonian Circuit

- Similar question: Is there a circuit that goes through all vertices



Hamiltonian Circuit

- Turns out to be very difficult
    - Can be shown to not be decidable with a polynomial time algorithm

Graph Definitions

- Distance in a graph:
  - Length of the shortest path between two vertices

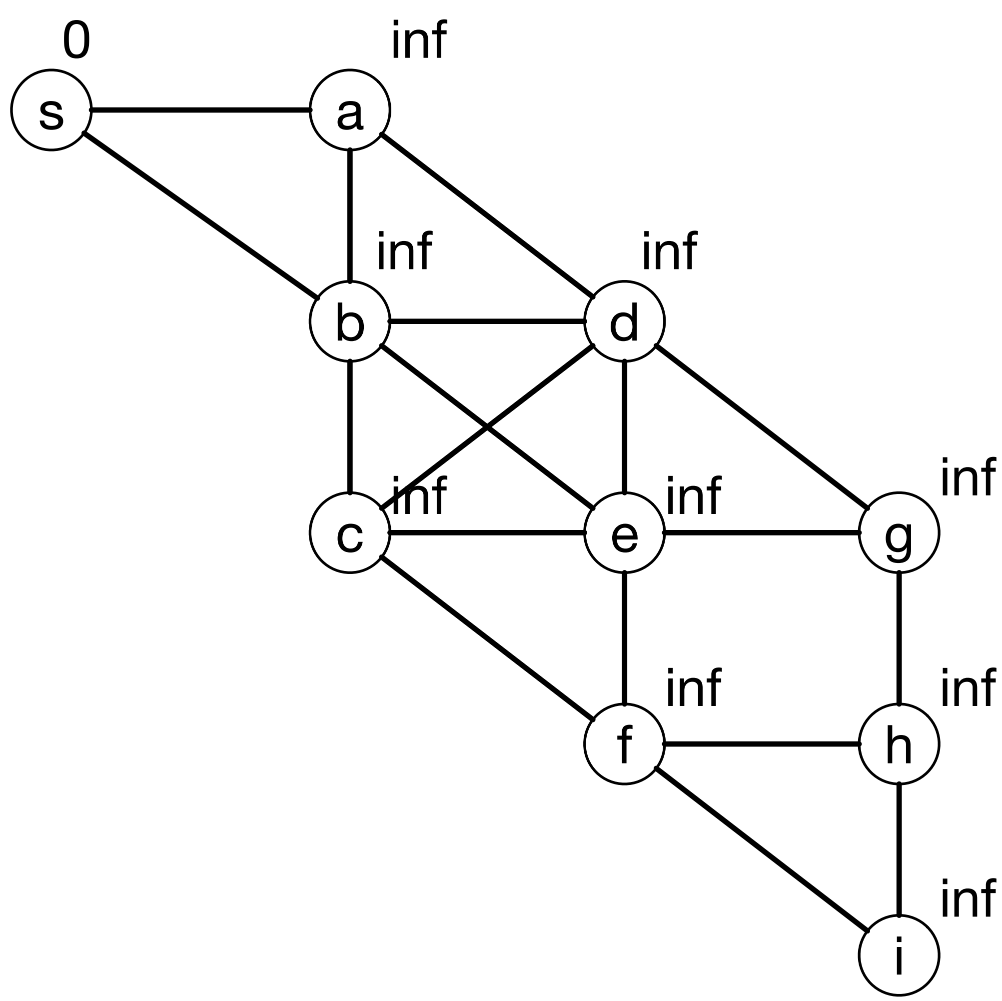
$\delta(u, w) = \min\{n \mid \exists v_0 = u, v_1, \ldots v_n = w$ such that $(v_i, v_{i+1} \in E \, \forall i \in \{0,\ldots,n-1\}\}$

Dijkstra's Algorithm

- Want to determine the distance between a vertex s and all other vertices in an undirected graph
  - Dynamic programming algorithm
    - Add intermediate vertices one by one
    - Start: Every vertex not s gets distance infinity
      - s gets distance 0
    - Put all vertices into a priority heap ordered by distance
      - We can quickly extract a vertex with minimum distance

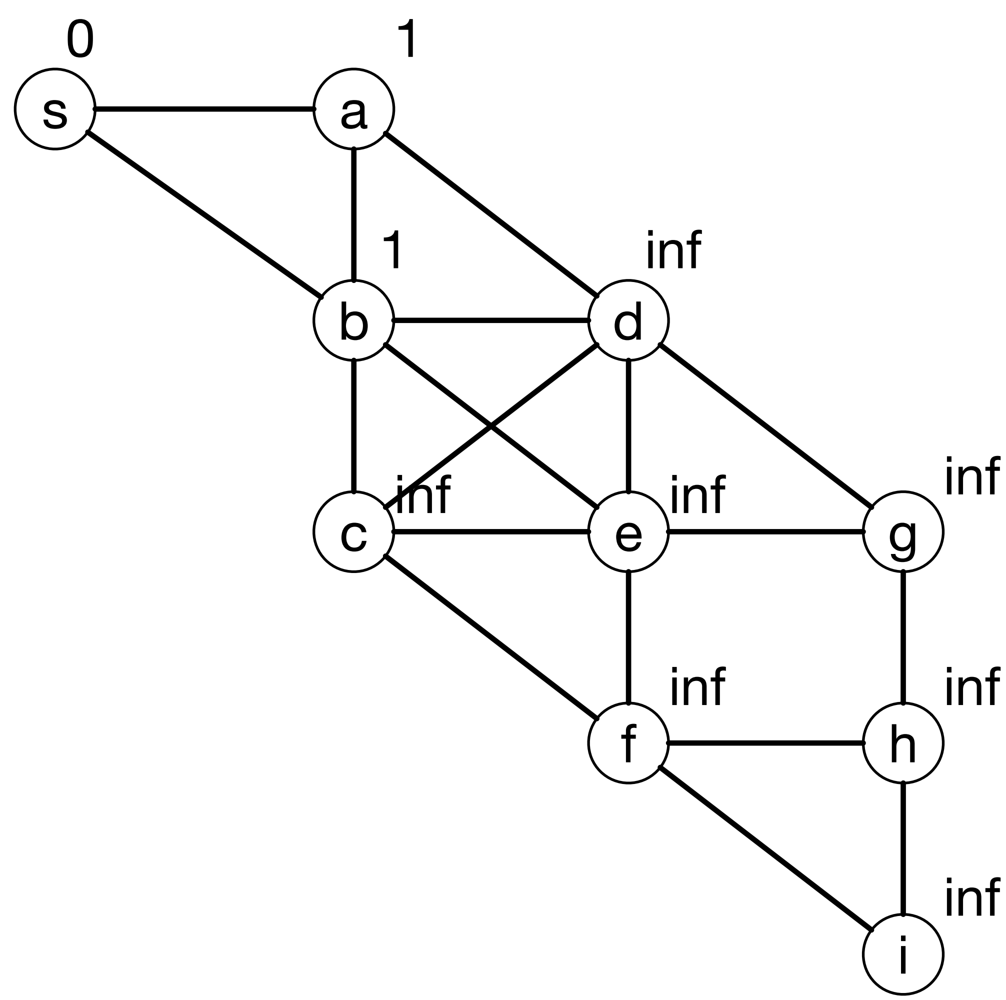
Dijkstra's Algorithm

- Example:



Dijkstra's Algorithm

- Update s:
  - Give all neighbors of s distance 1



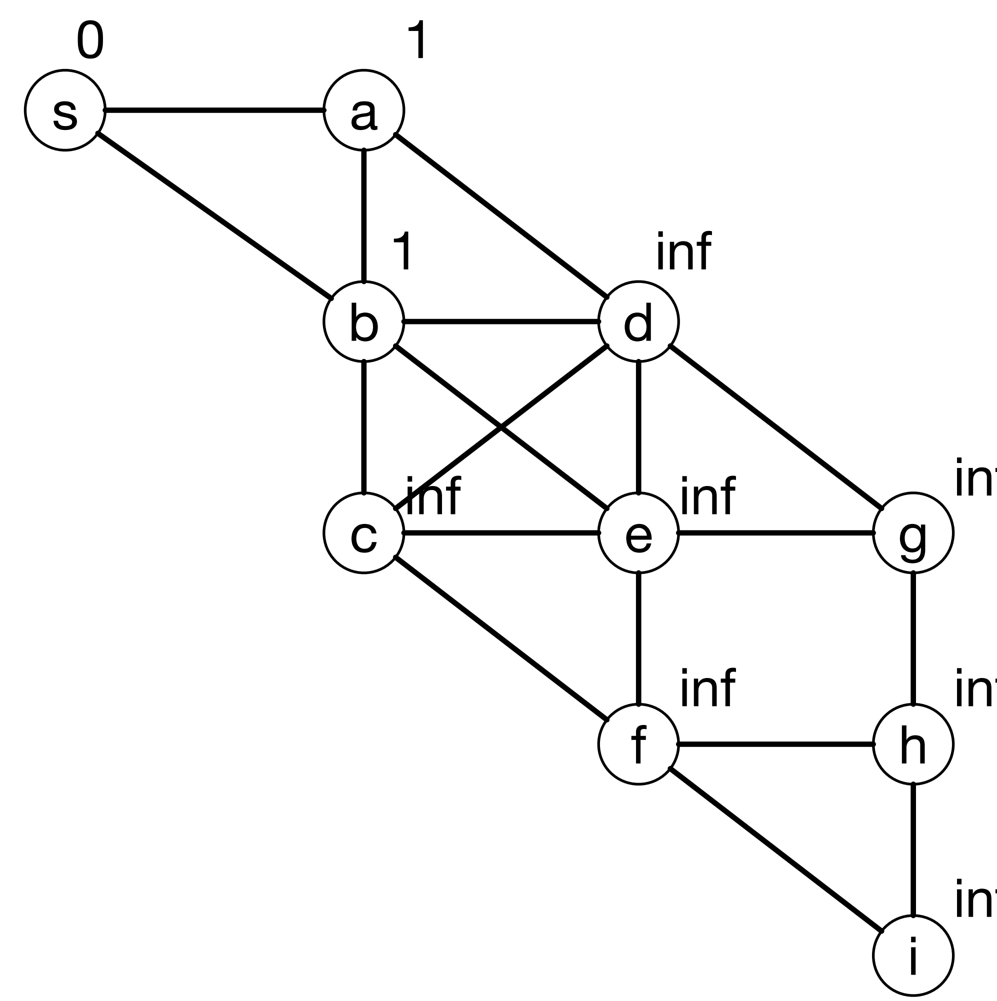
-

Dijkstra's Algorithm

- The heap gives us one of $\{a, b\}$ as a minimum distance node.
  - Pick a.
  - Update all its neighbors by giving them an updated distance
    - Minimum of current value
    - Value of a plus 1
  - a is connected to b, c, and s

Dijkstra's Algorithm

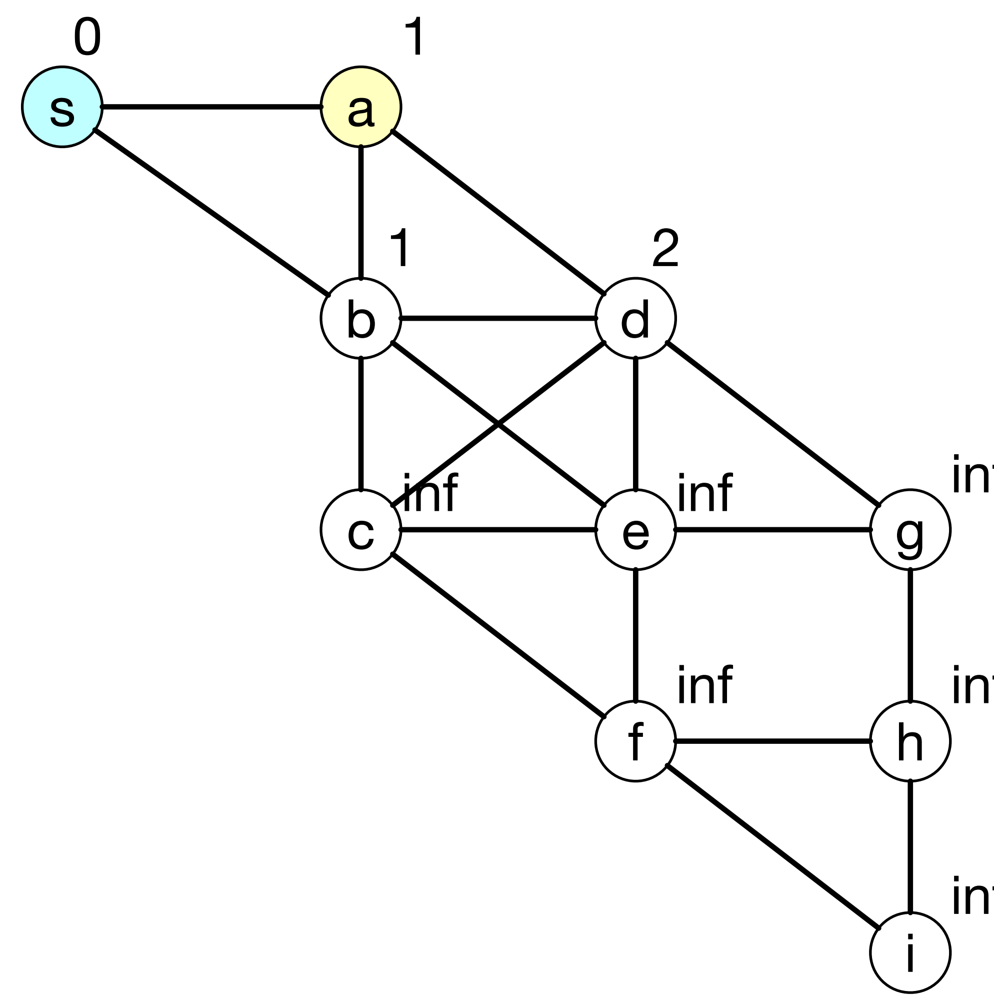
- b gets min(1, 1+1)
- s gets min(0, 1+1)
- d gets min(inf, 1+1)



Dijkstra's Algorithm

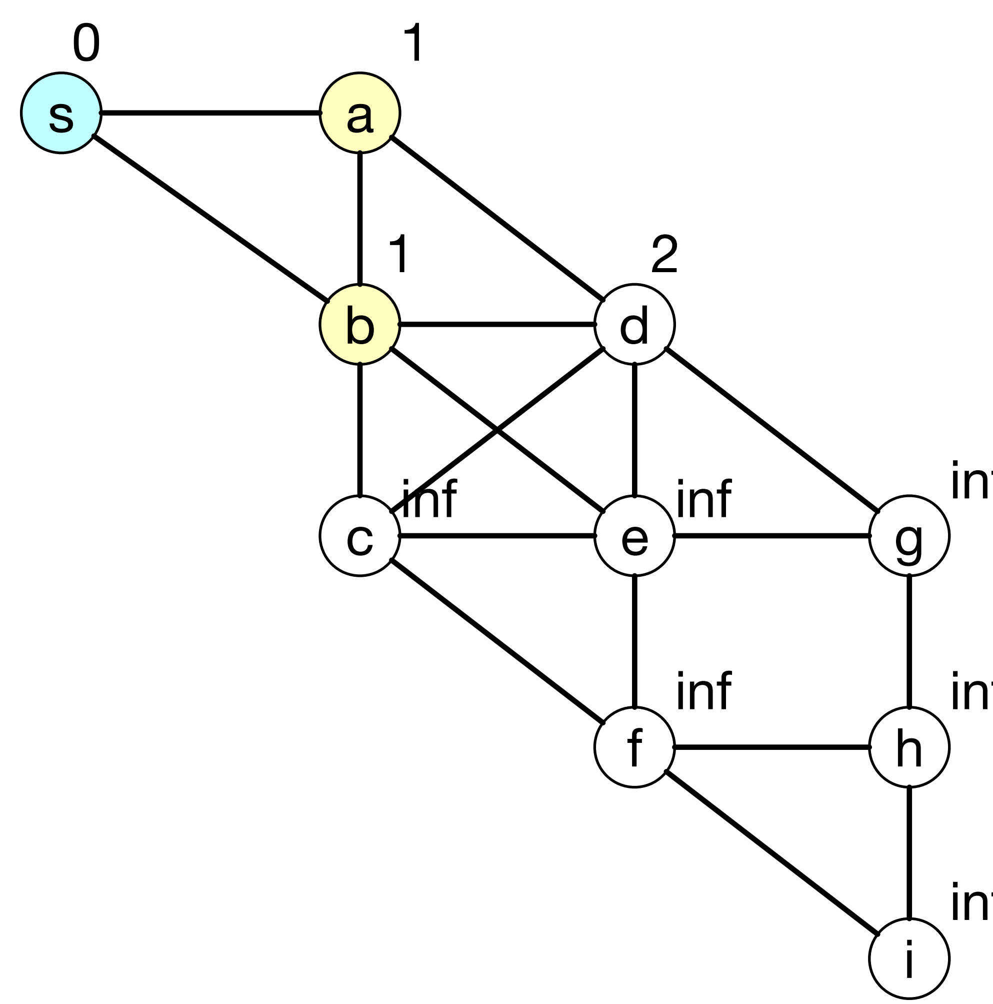
- b gets min(1, 1+1)
- s gets min(0, 1+1)
- d gets min(inf, 1+1)

- After update, mark a as used by removing it from the priority queue



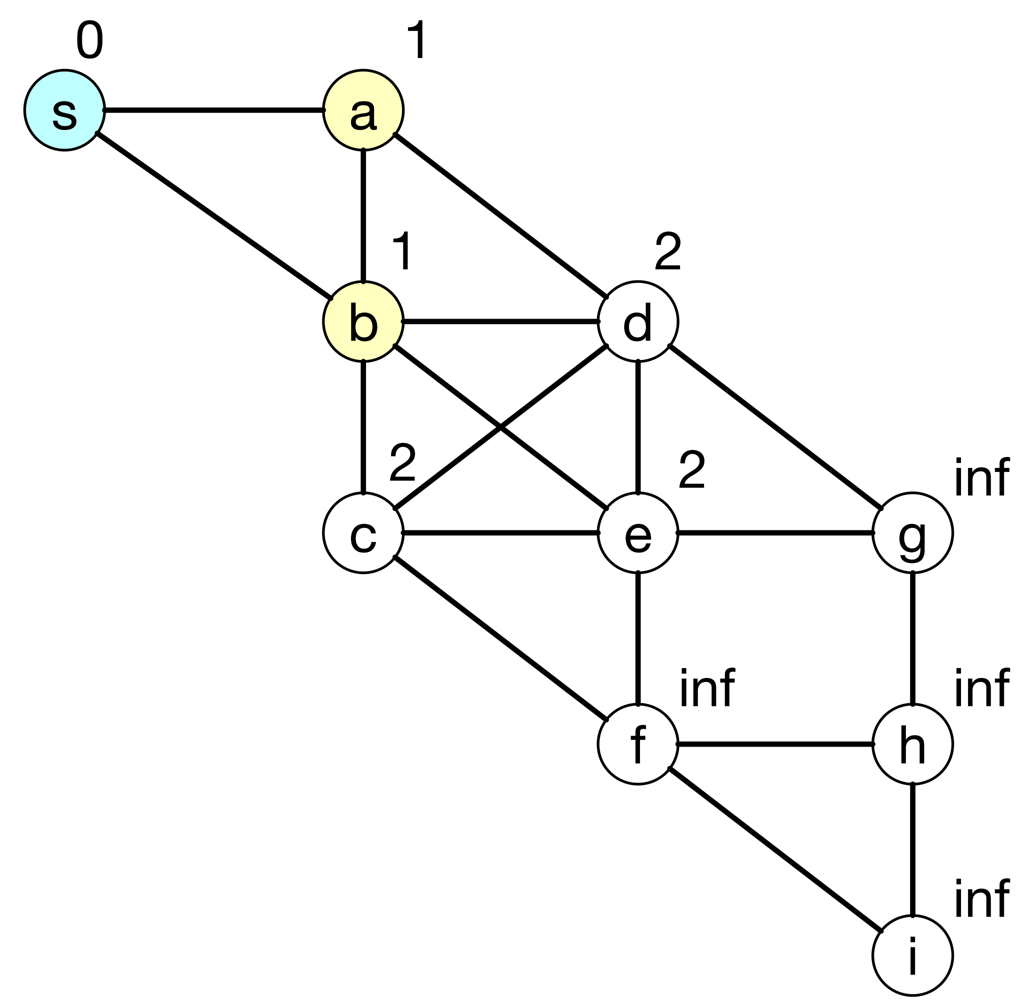
Dijkstra's Algorithm

- Pick the node with minimum distance that is not marked
- Which would be b
- Update its neighbors



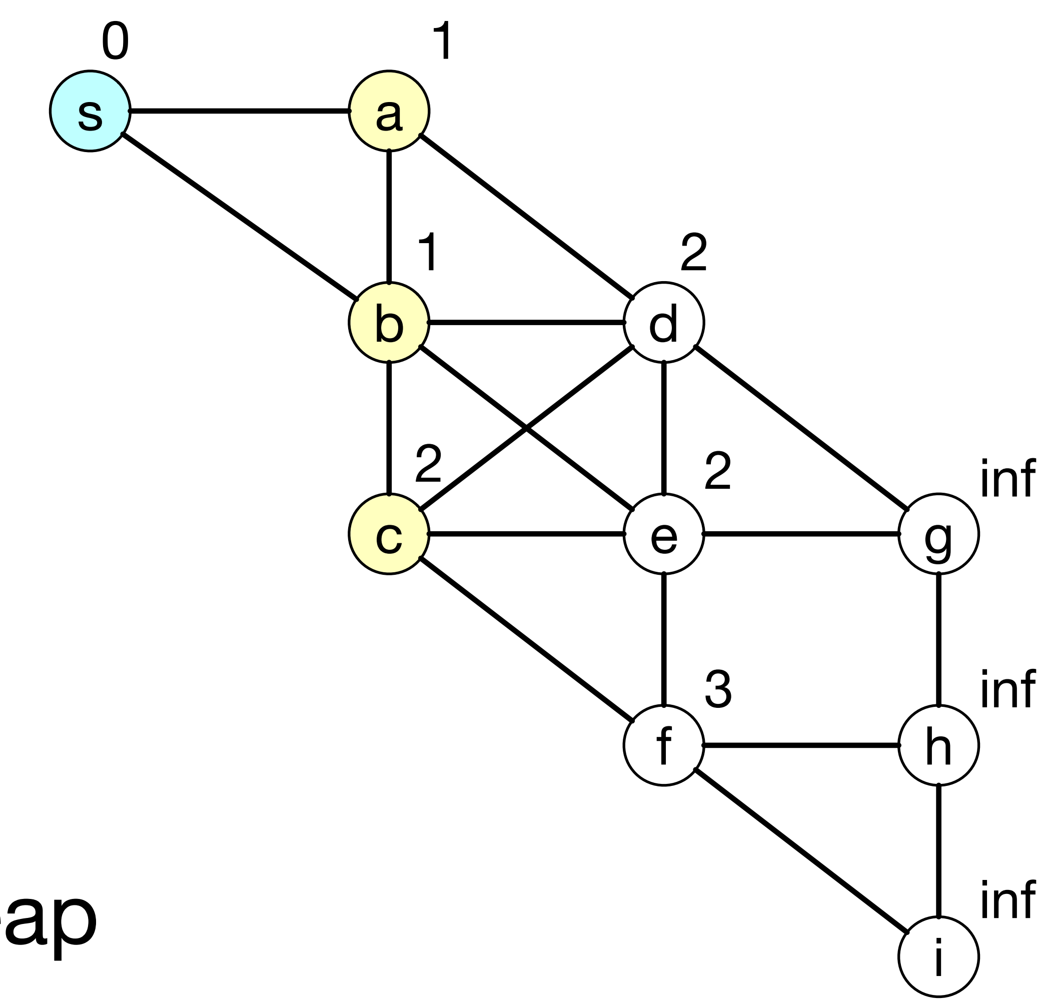
Dijkstra's Algorithm

- d gets min(2,1+1)
- c gets min(inf, 1+1)
- e gets min(inf, 1+1)
- s gets min(0, 1+1)



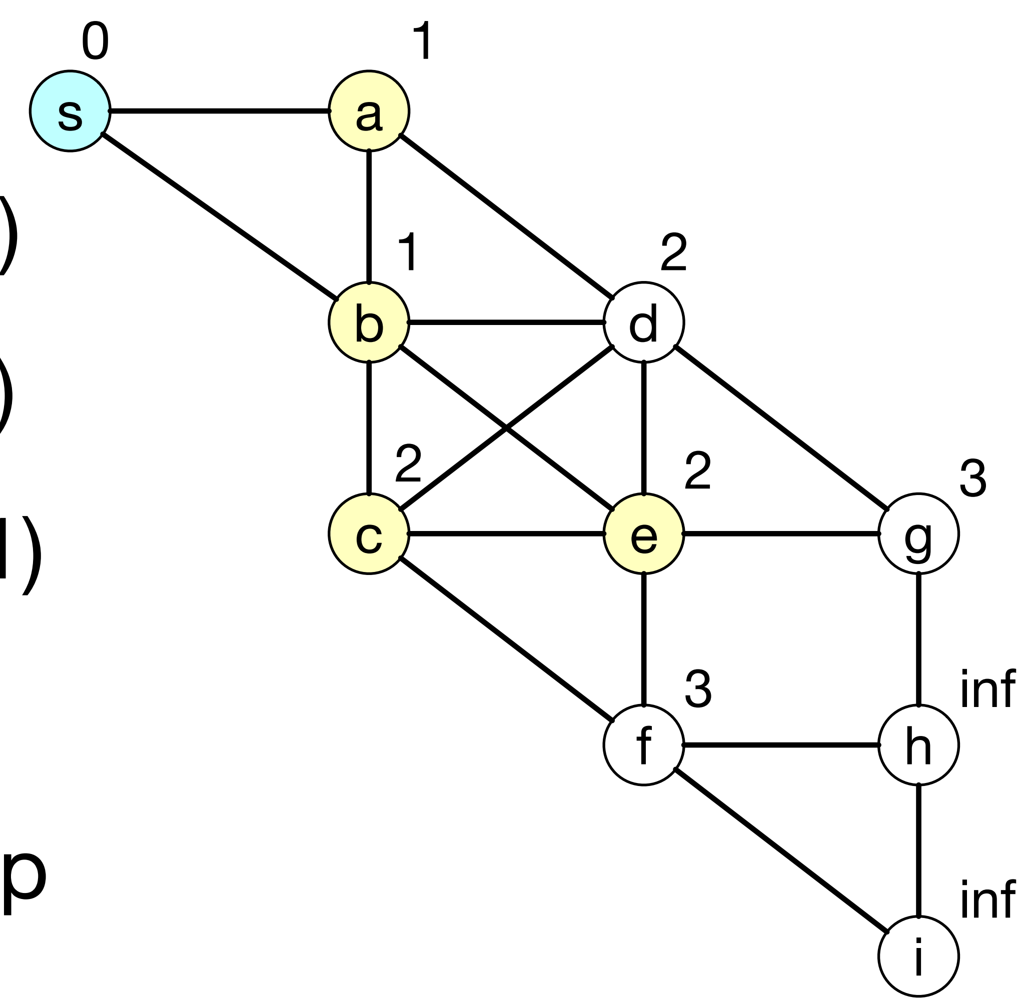
Dijkstra's Algorithm

- Select one of the vertices with minimum distance:
  - Either c, d, or e
  - Pick c
    - b gets min(1,2+1)
    - d gets min(2, 2+1)
    - e gets min(2, 2+1)
    - f gets min(inf, 2+1)
  - Remove c from the priority heap



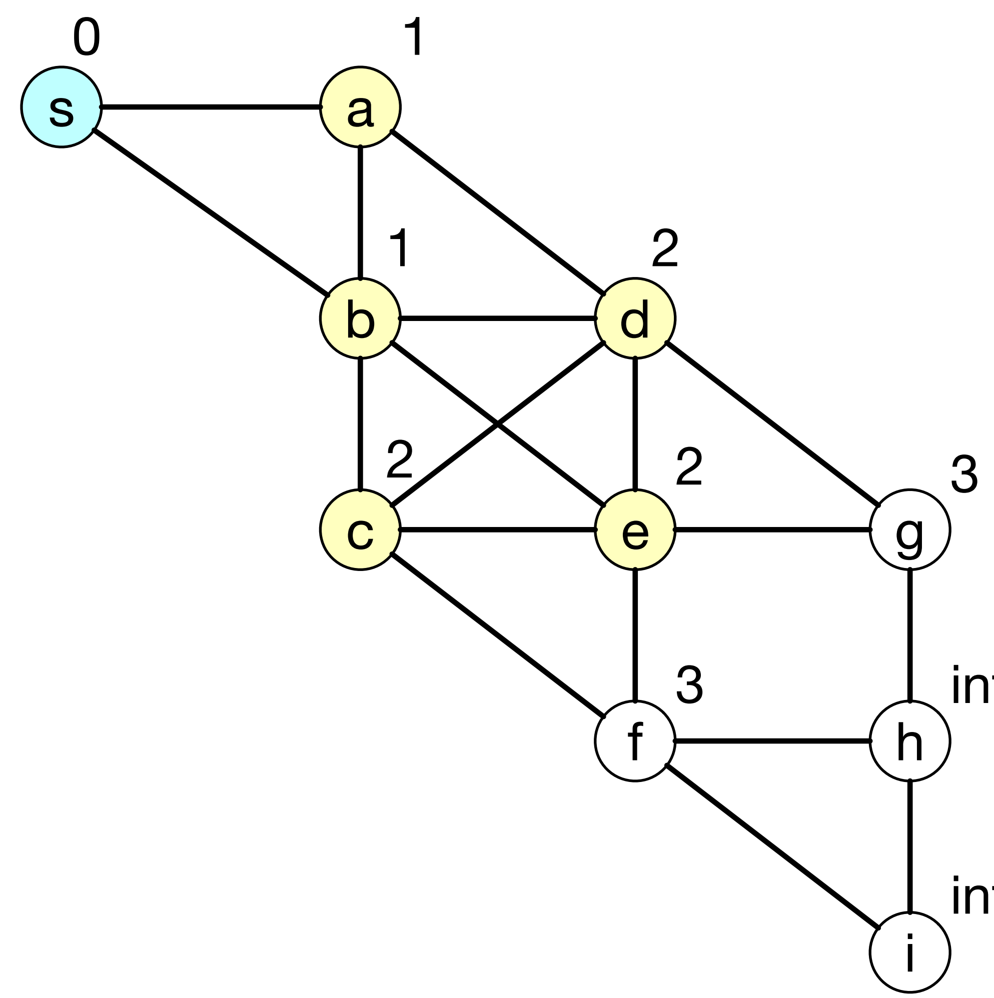
Dijkstra's Algorithm

- Select e
  - Update b with min(1,2+1)
  - Update c with min(2,2+1)
  - Update d with min(2, 2+1)
  - Update f with min(3,2+1
- Remove e from priority heap



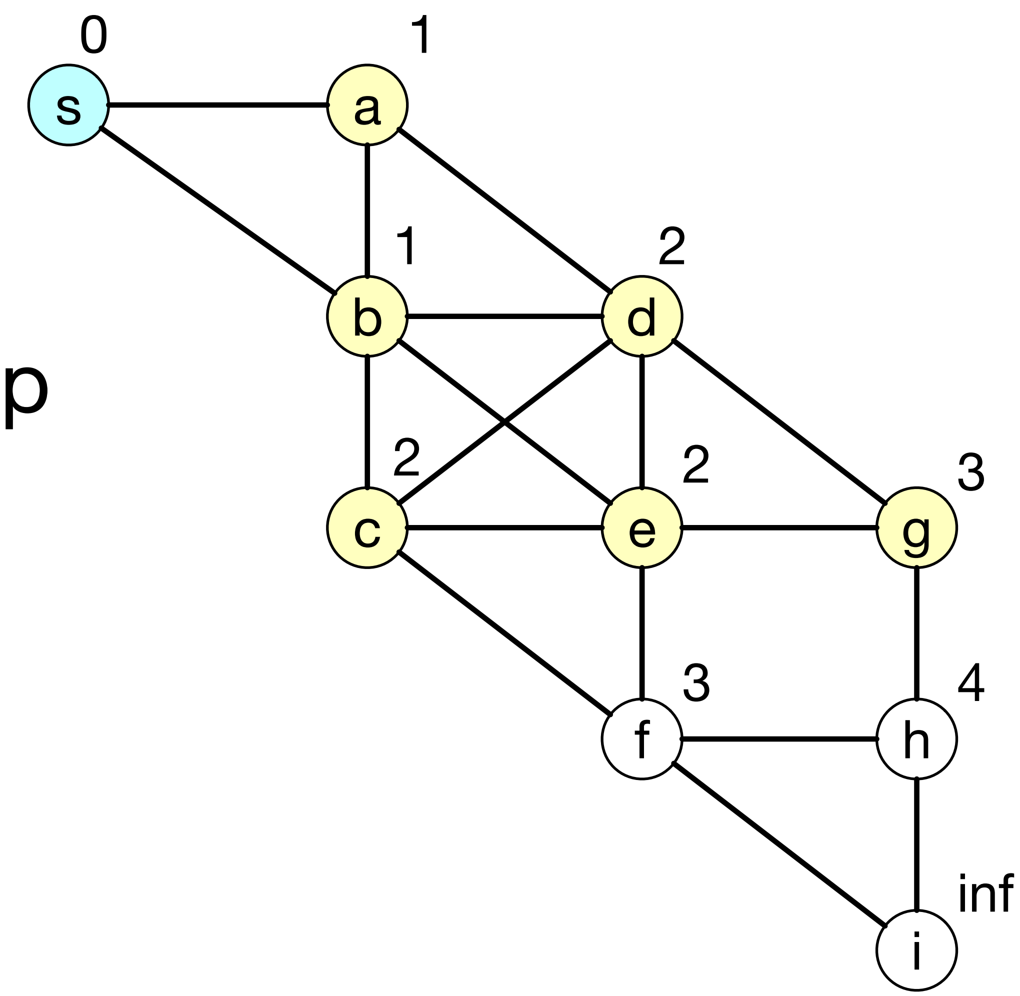
Dijkstra's Algorithm

- Select d
  - Updates have no effect
- Remove d from heap



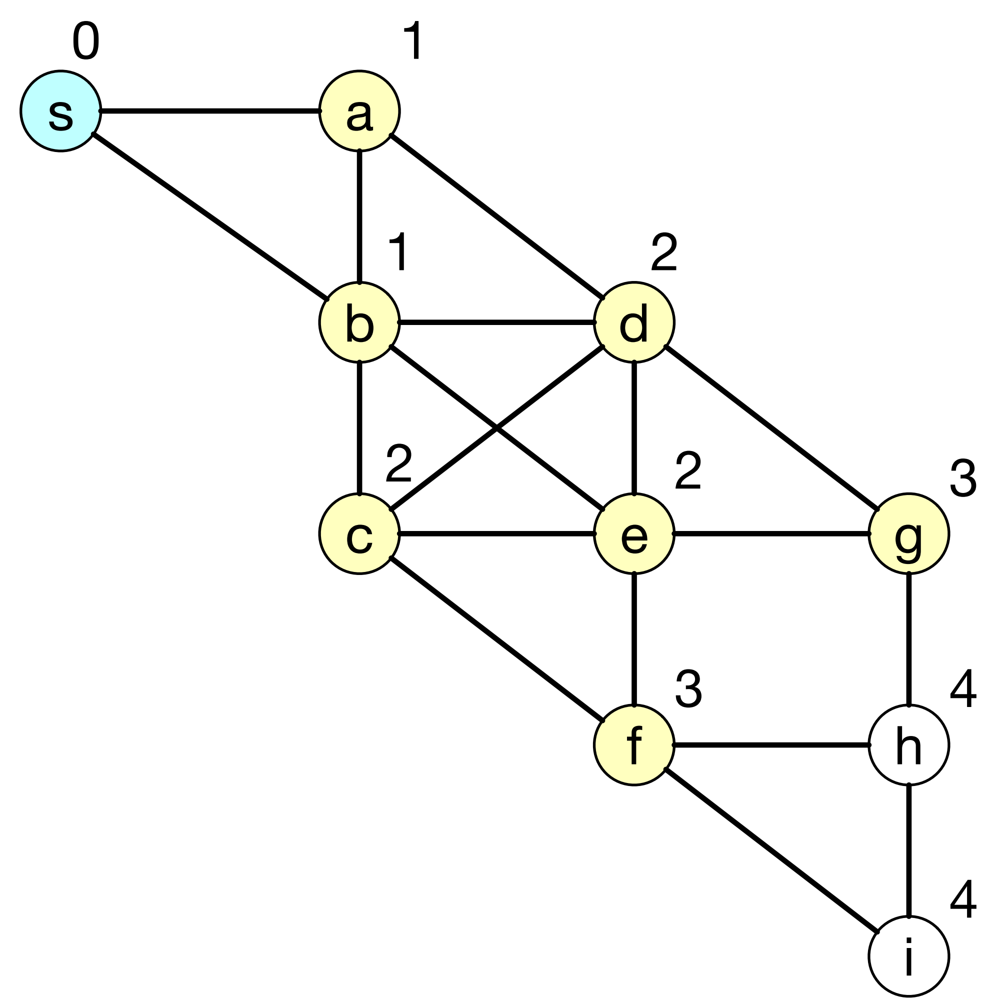
Dijkstra's Algorithm

- Select g
  - Only change is h gets 4
- Remove g from priority heap



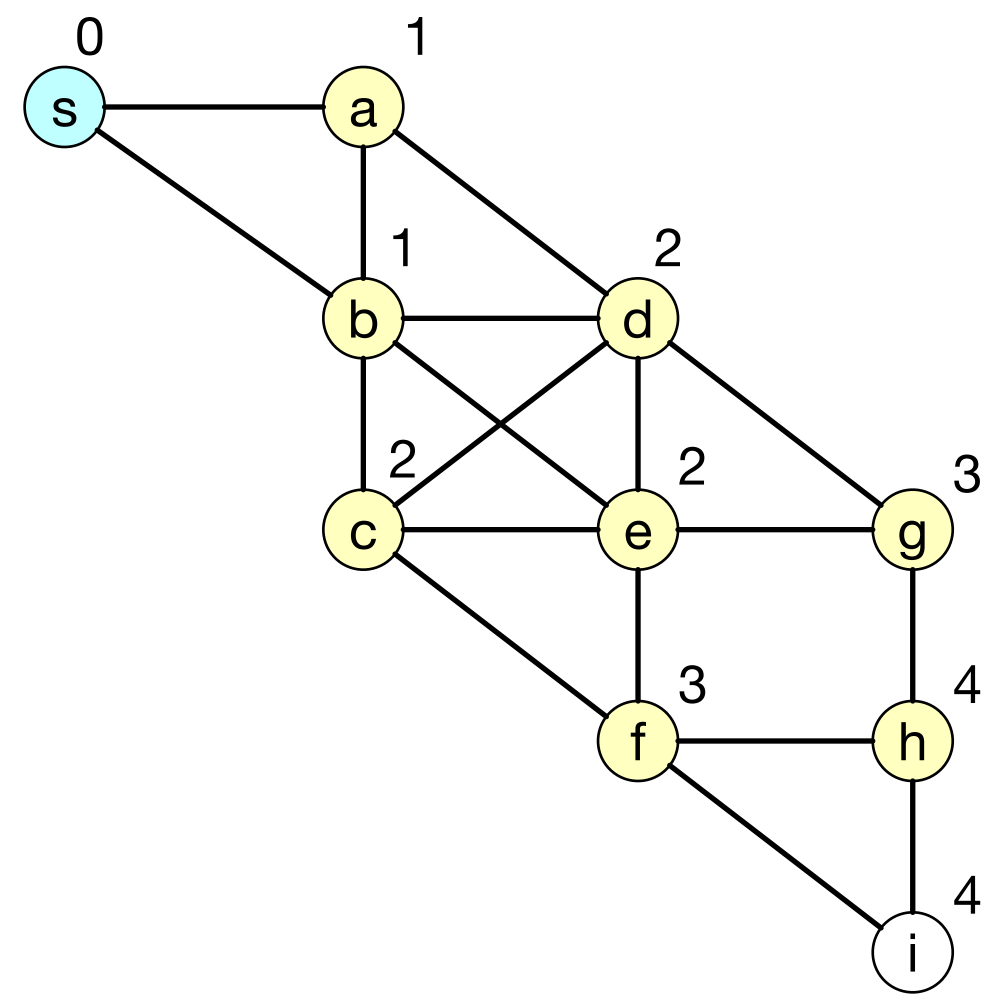
Dijkstra's Algorithm

- Need to select f
  - Update only changes h



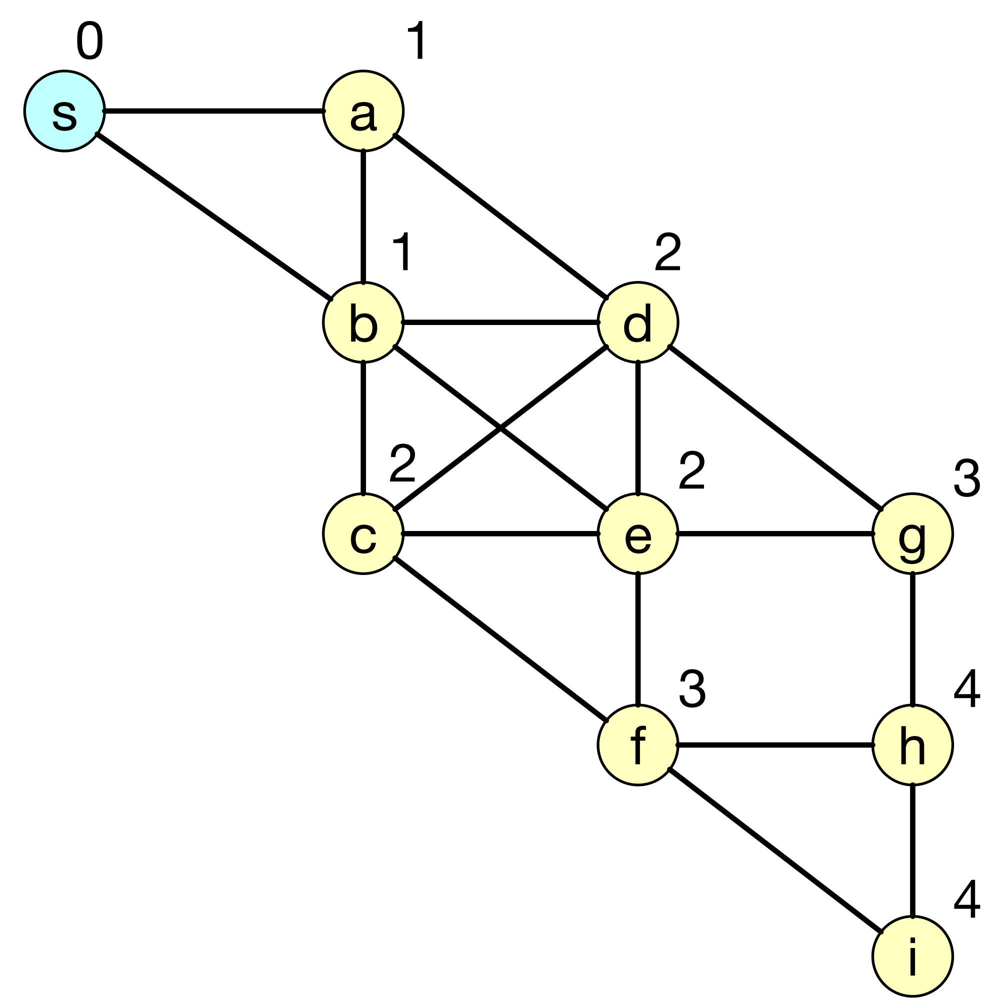
Dijkstra's Algorithm

- Need to select h
  - Does not change any value



Dijkstra's Algorithm

- Need to select i as the only node left
  - But that does not change any values

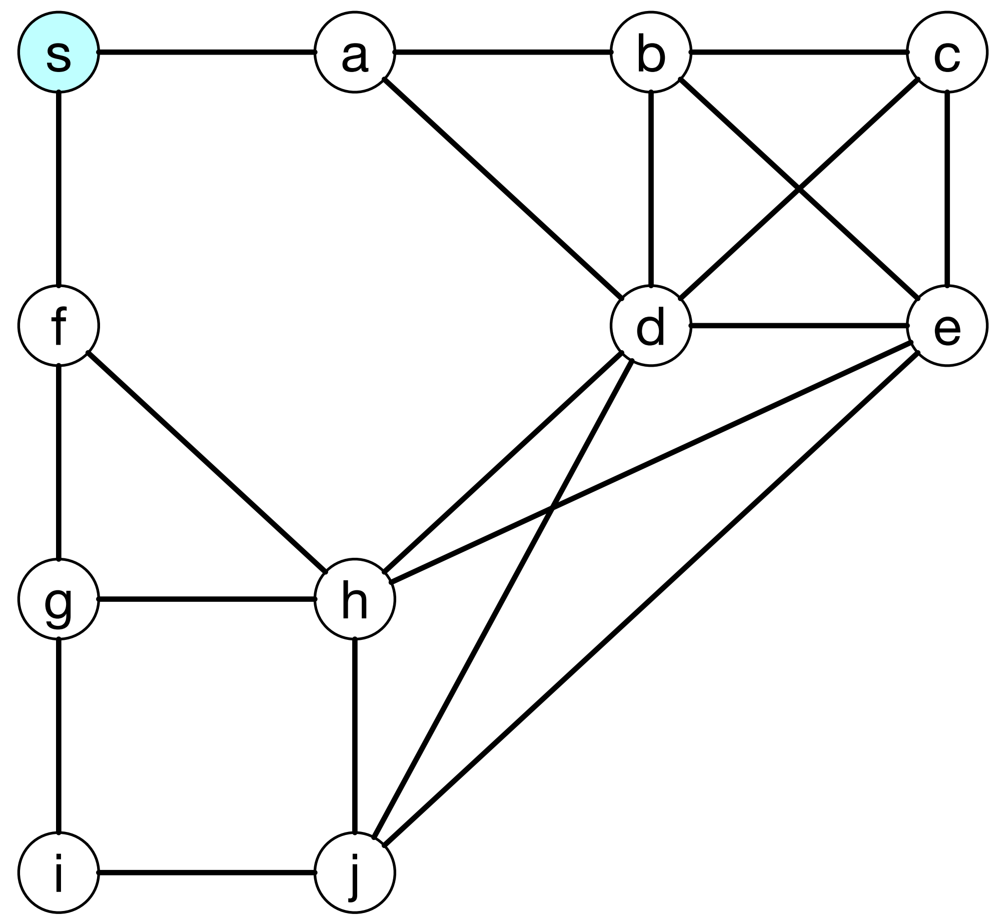


Dijkstra's Algorithm

- Dijkstra's algorithm can be generalized to weighted graphs

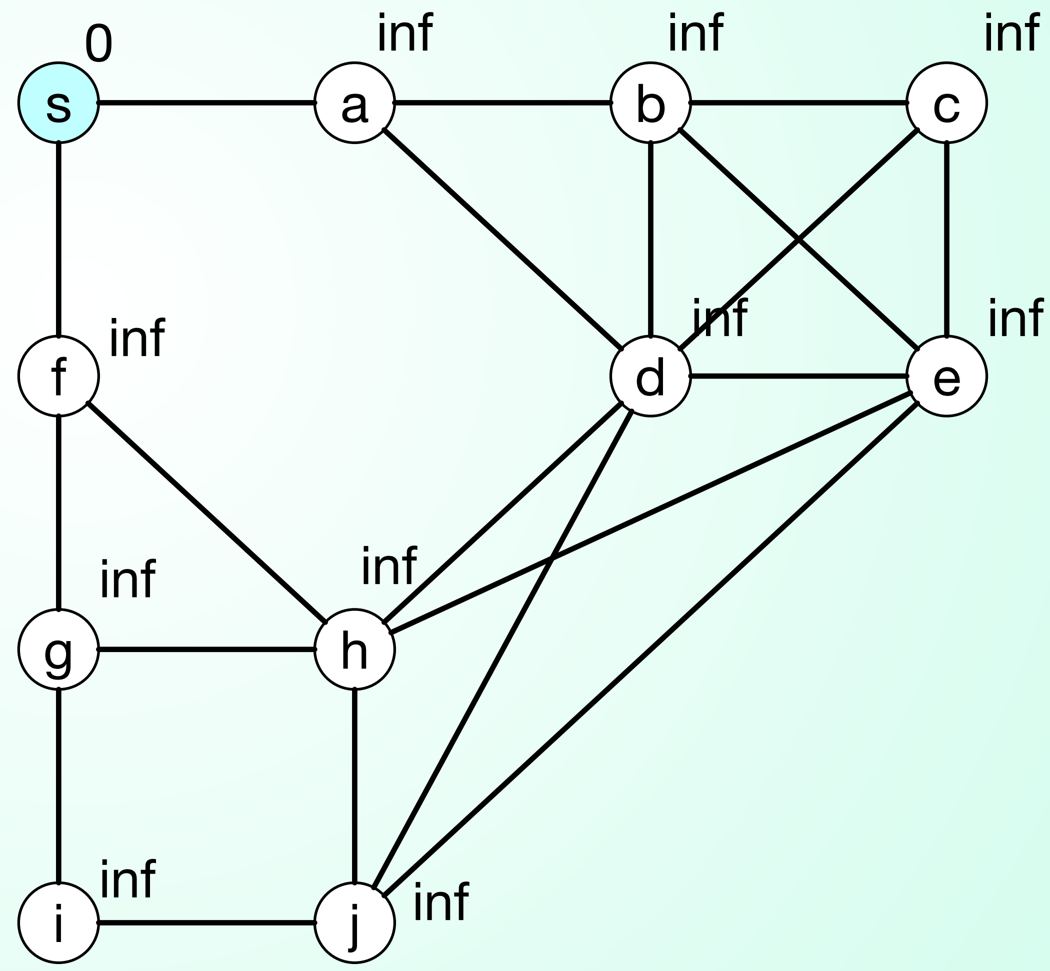
Dijkstra's algorithm

- Your turn
- Rule:
  - Of course you choose smallest distance first, but you break ties in order of the alphabet, e.g. select a over f



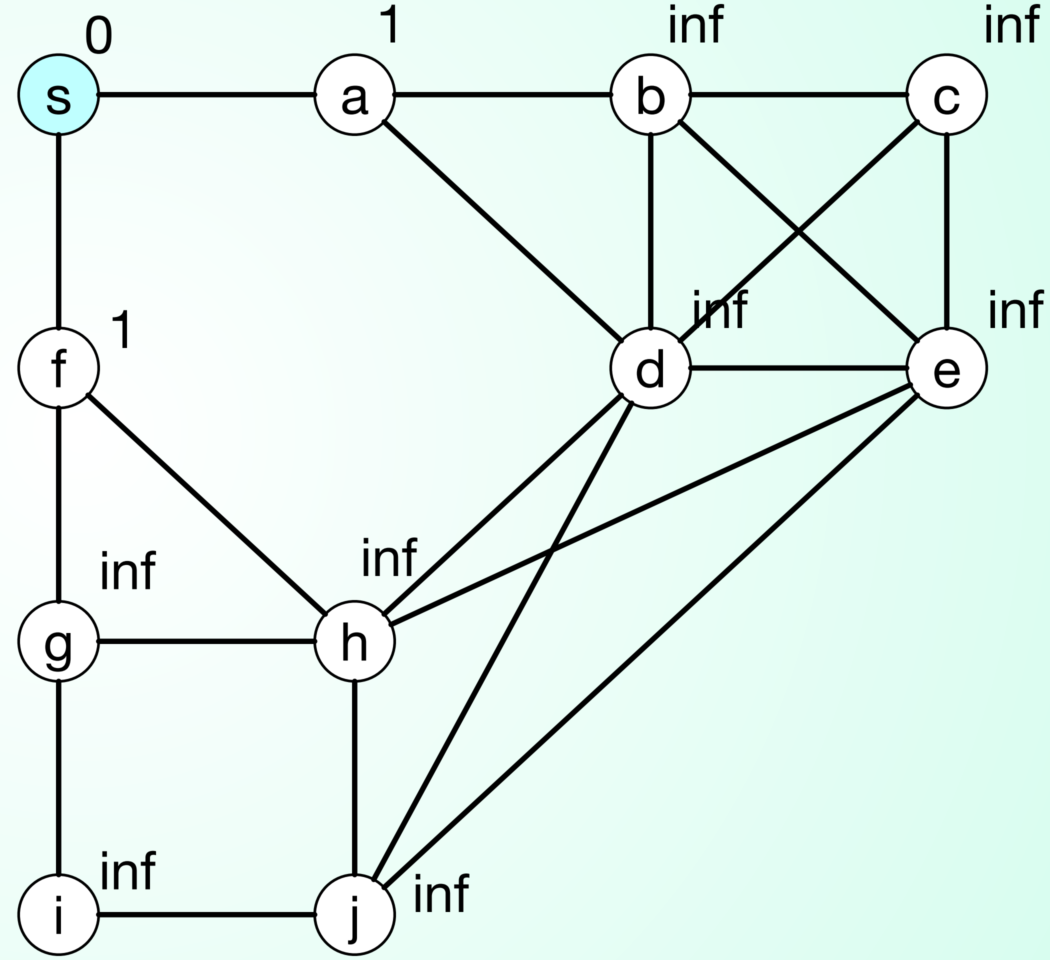
Dijkstra's algorithm

- Select s



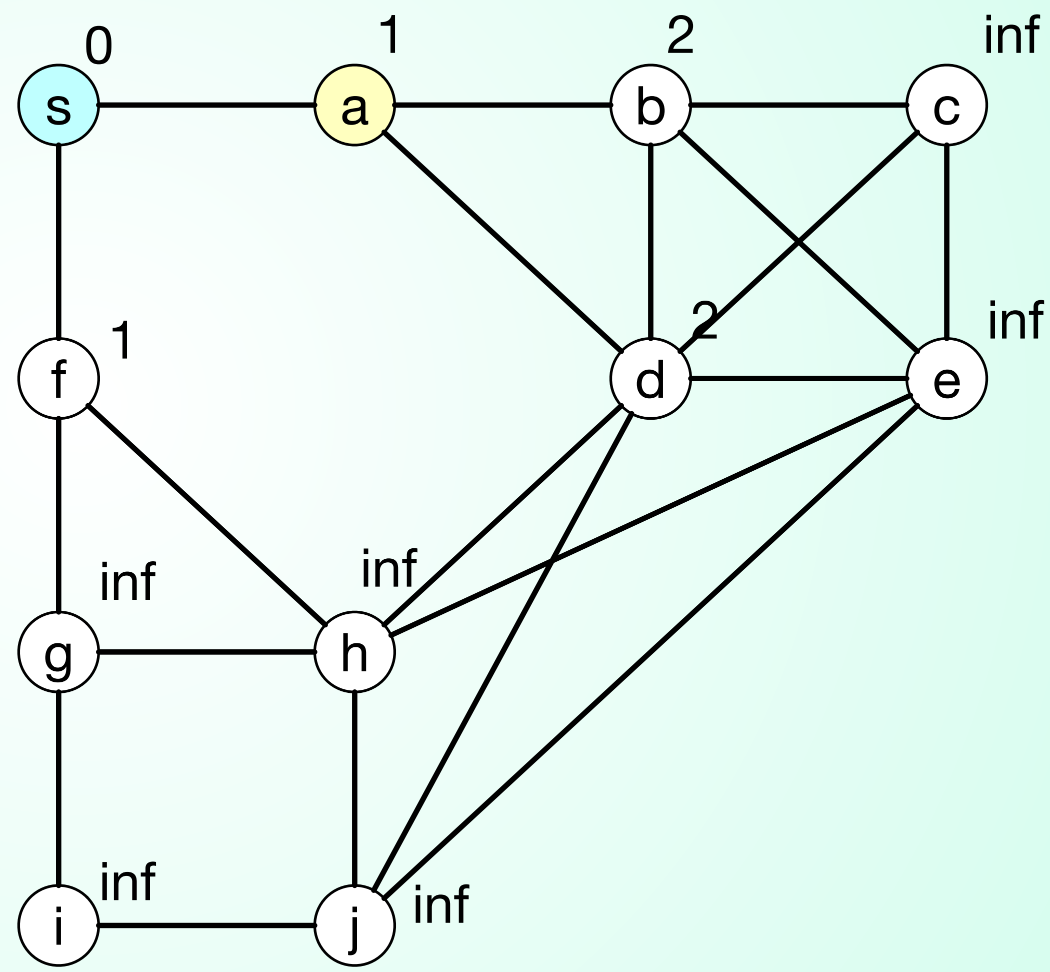
Dijkstra's algorithm

- Update a and f



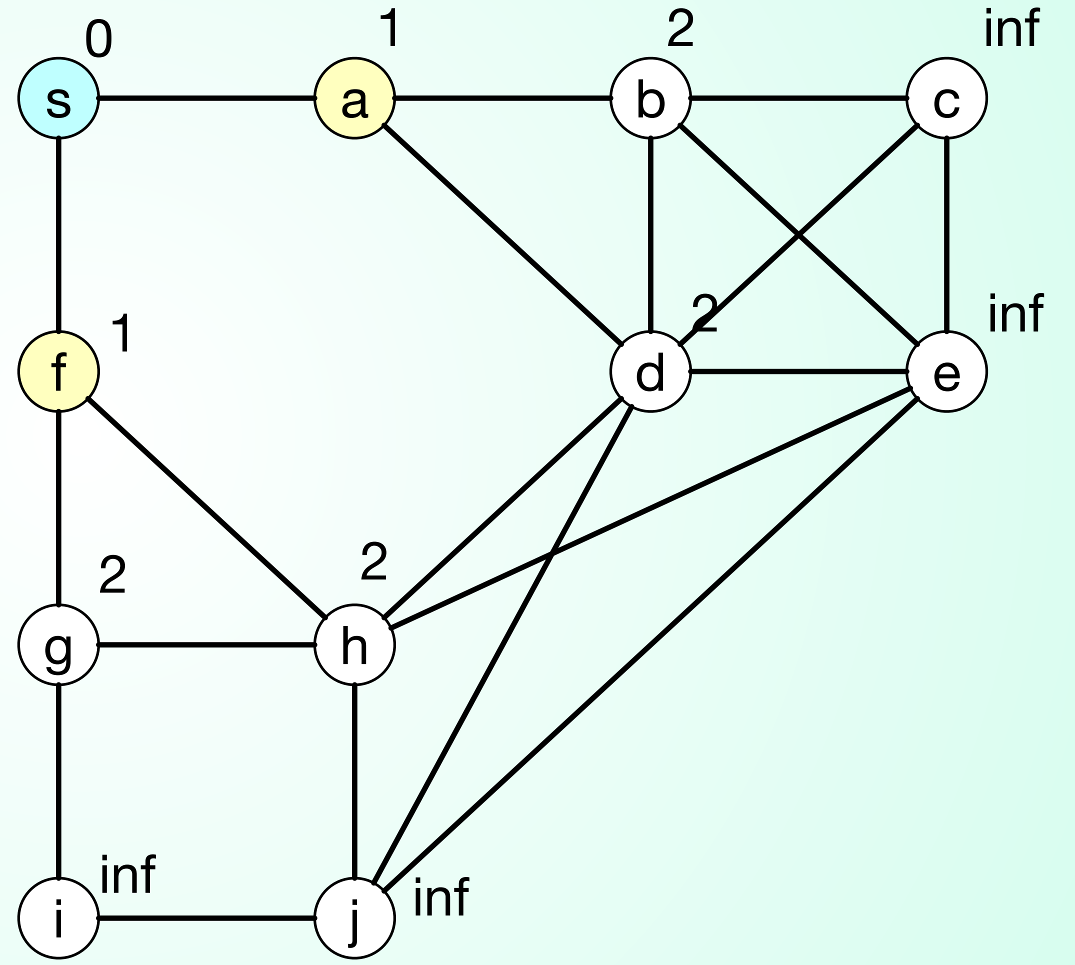
Dijkstra's algorithm

- Select a
    - Update b and d
    - s stays the same



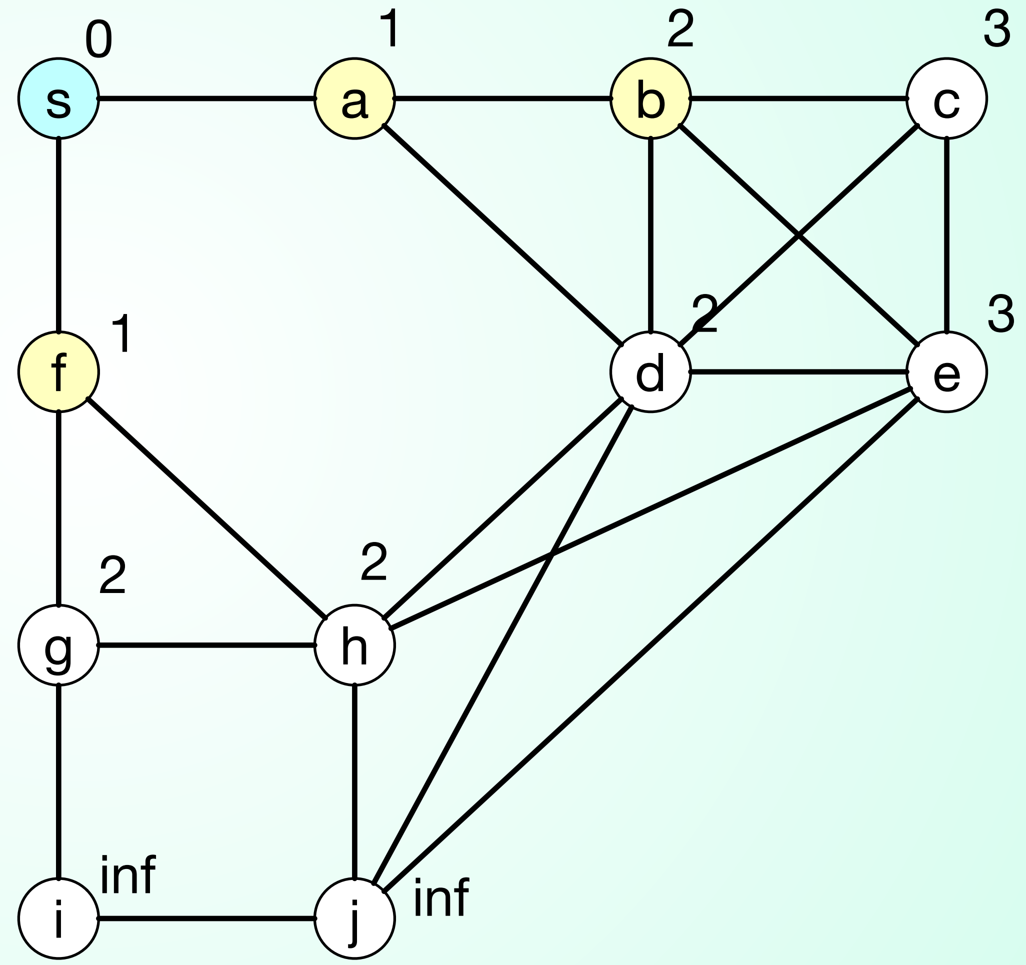
Dijkstra's algorithm

- Select f (no choice here)



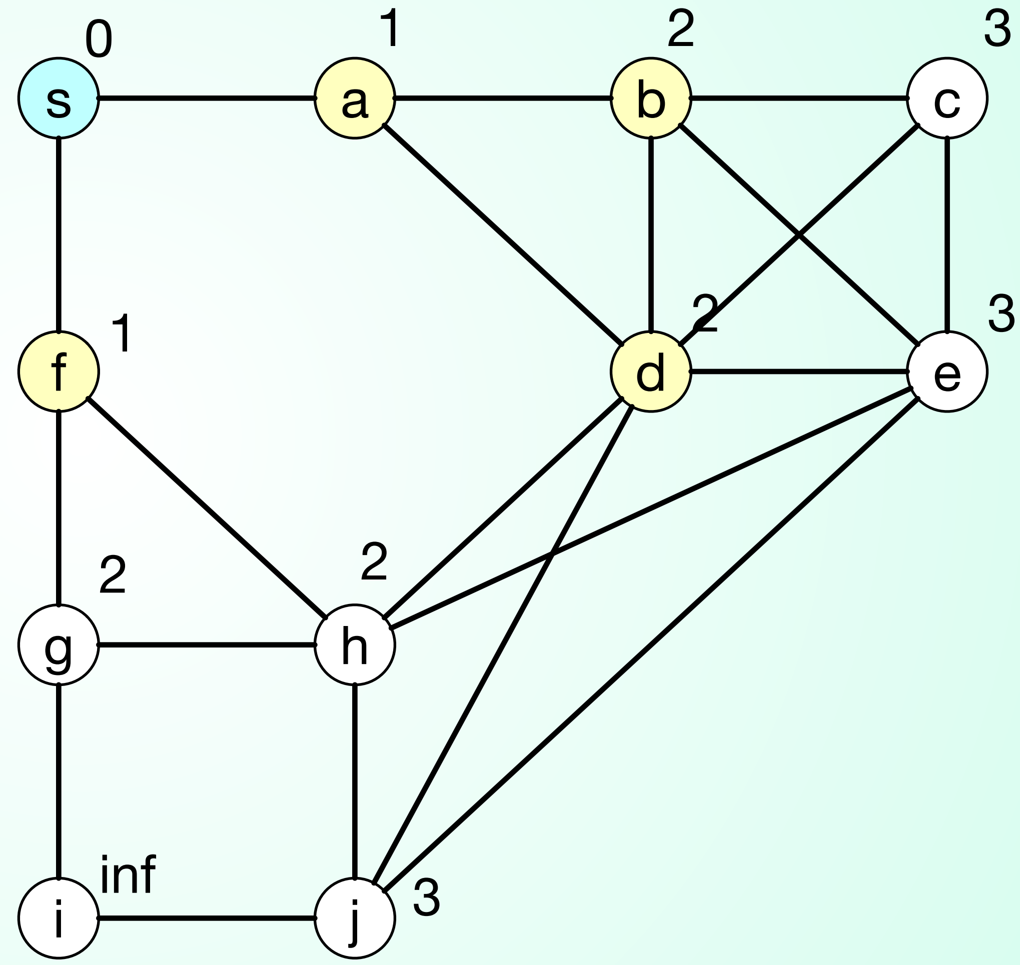
Dijkstra's algorithm

- Select b



Dijkstra's algorithm

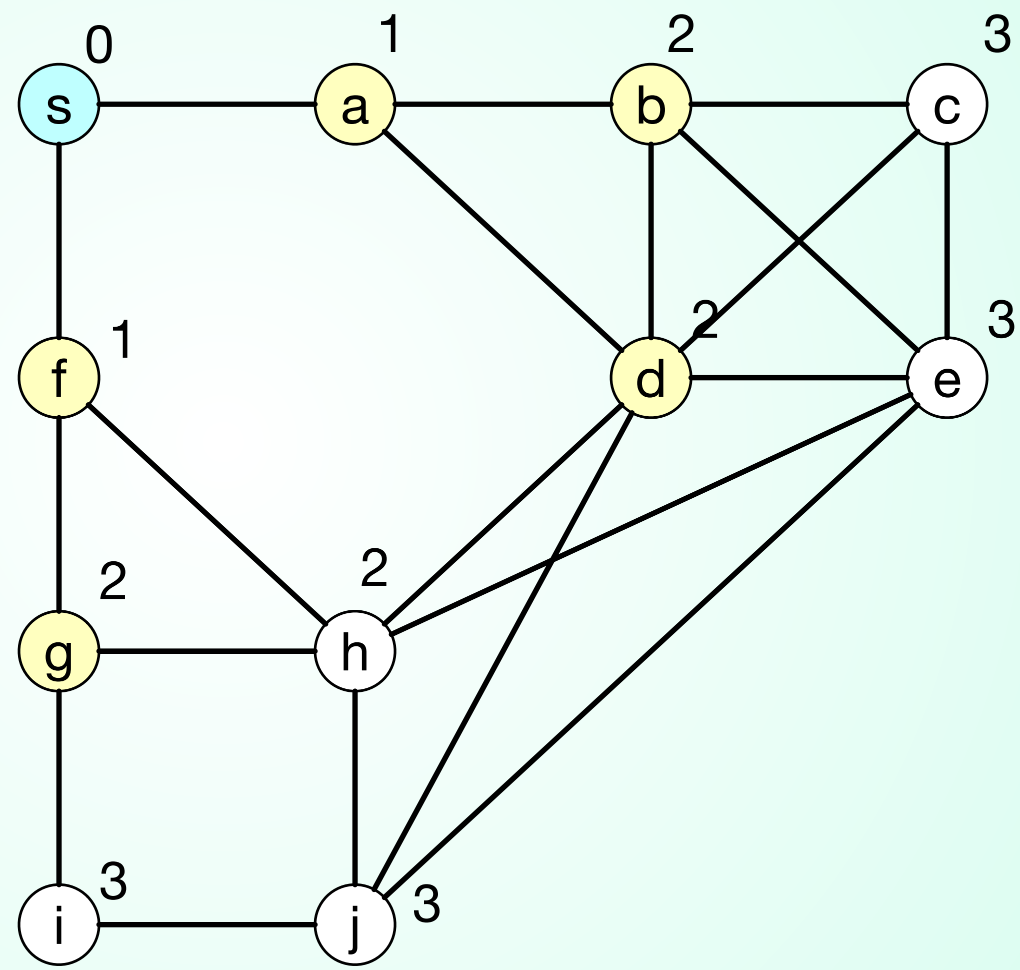
- Select d



Dijkstra's algorithm

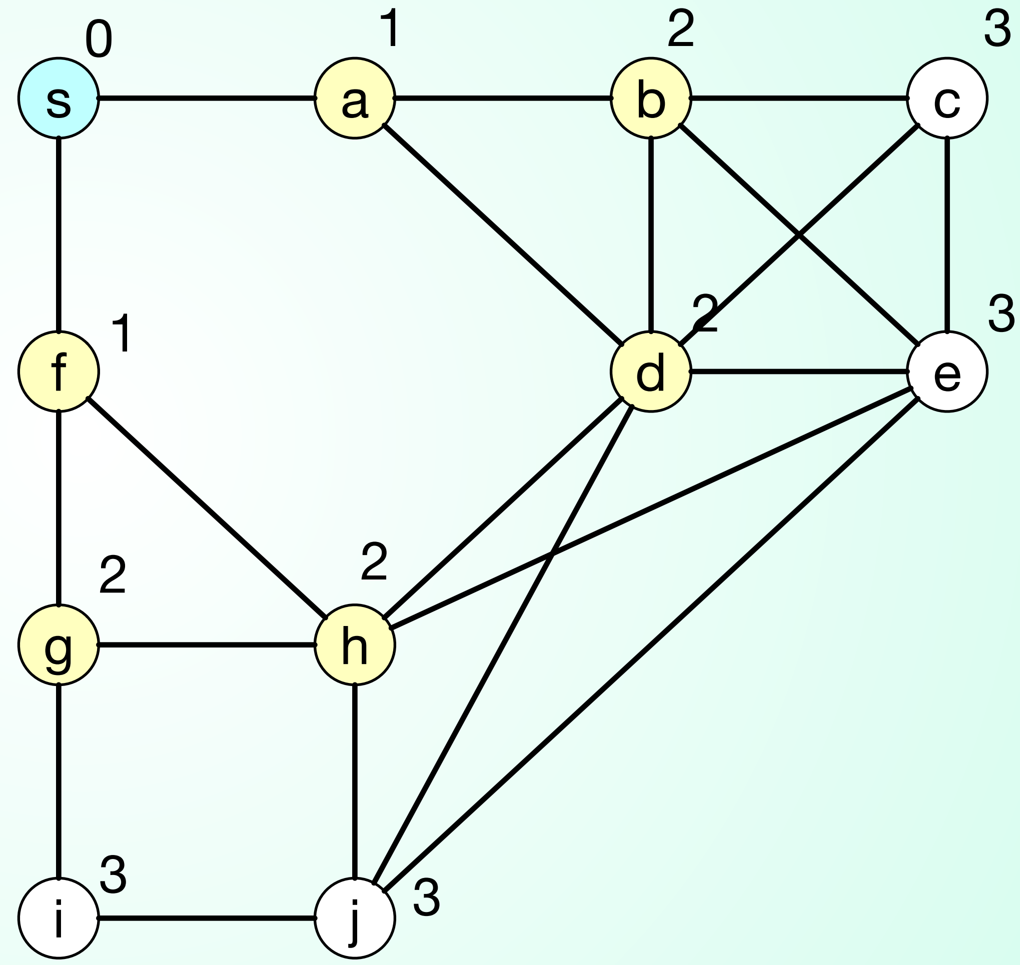
Dijkstra's algorithm

- Select g



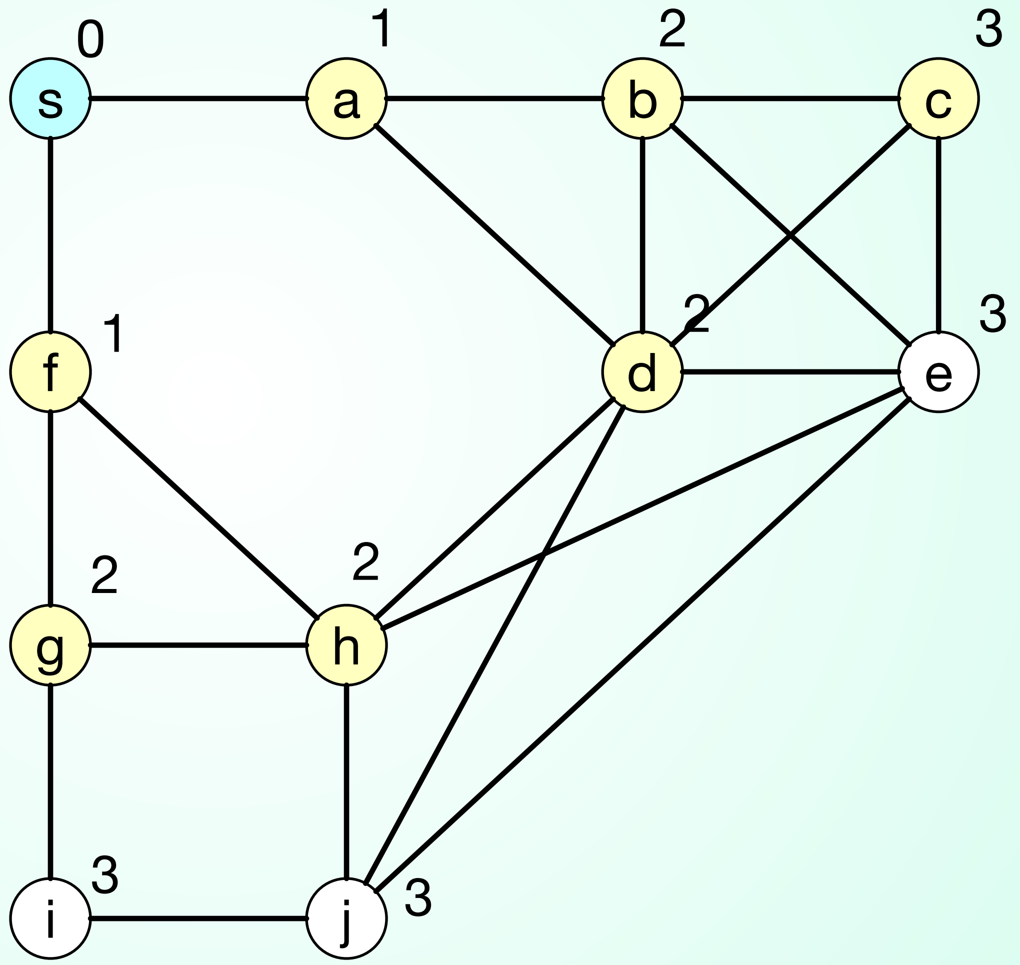
Dijkstra's algorithm

- Select h



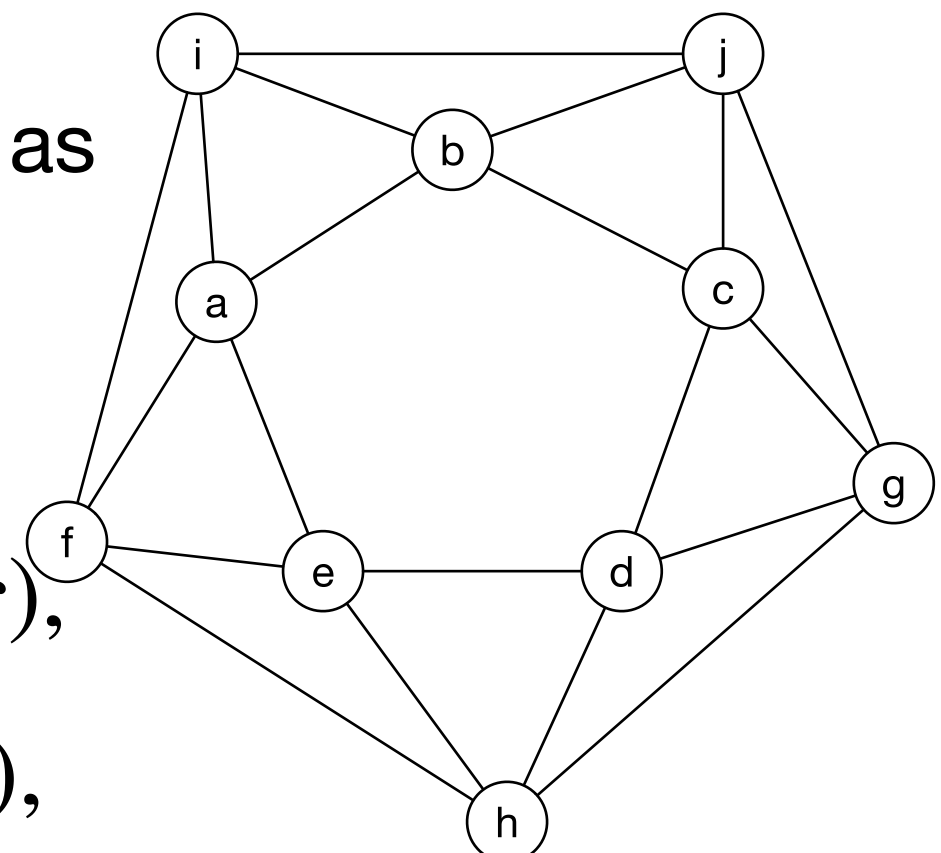
Dijkstra's algorithm

- Select c
  - We might as well stop here
  - All updated values will be 4 or more, and every node has already a 3



Graph Representations

- For computational purposes, we can use:
  - List of vertices and list of edges as pairs



$$V = \{a, b, c, d, e, f, g, h, i, j\}$$

$$E = \{(a, b), (a, e), (a, f), (a, i), (b, c),$$

$$(b, i), (b, j), (c, d), (c, g), (c, j), (d, e),$$
$$(d, h), (d, g), (e, f), (e, h), (f, h), (f, i),$$

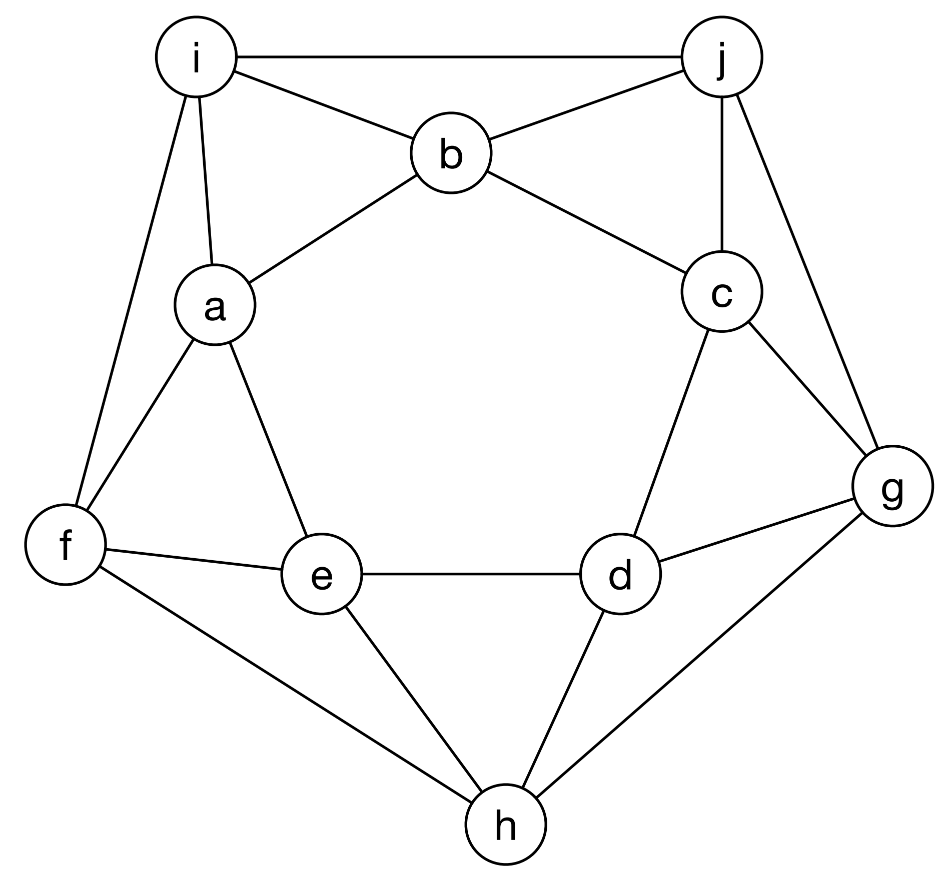
$$(g, h), (g, j), (i, j)\}$$

Dijkstra's Algorithm

- Need to maintain a priority heap
  - Otherwise
    - Look at every node
    - And every edge twice

Graph Representations

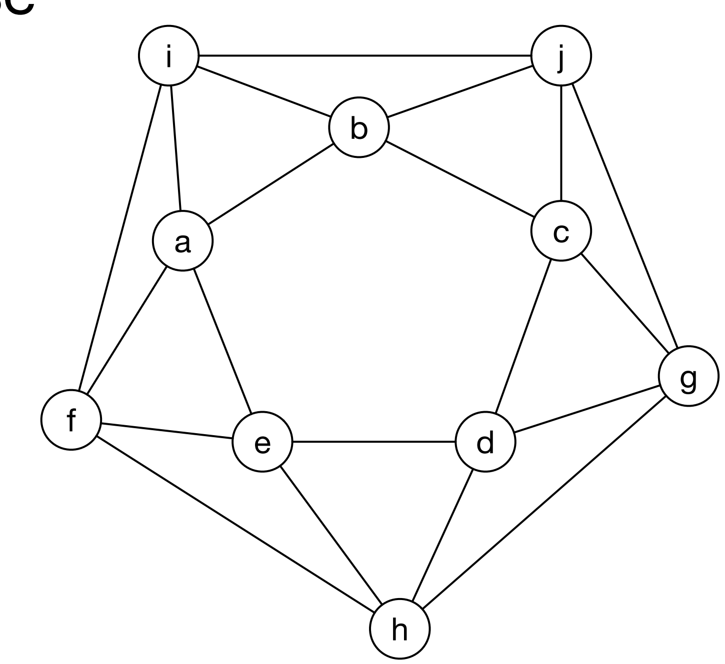
- An adjacency list
  - For every vertex the list of vertices to which there is an edge
  - ```
    a: b,e,f,i
    b: a,c,i,j
    c: b,d,g,j
    d: c,e,g,h
    e: a,f,d,h
    f: a,e,h,i
    g: c,d,h,j
    h: d,e,f,g
    i: a,b,f,j
    j: b,c,g,i
    ```


    ```

Graph Representations

- An adjacency matrix
  - square matrix conceptually labeled with vertices
  - coefficient
    $$a_{i,j} = \begin{cases} 1 & \text{edge between } v_i \text{ and } v_j \\ 0 & \text{otherwise} \end{cases}$$

	a	b	c	d	e	f	g	h	i	j
a	0	1	0	0	1	1	0	0	1	0
b	1	0	1	0	0	0	0	0	1	1
c	0	1	0	1	0	0	1	0	0	1
d	0	0	1	0	1	0	1	1	0	0
e	1	0	0	1	0	1	0	1	0	0
f	1	0	0	0	1	0	0	1	1	0
g	0	0	1	1	0	0	0	1	0	1
h	0	0	0	1	1	1	1	0	0	0
i	1	1	0	0	0	1	0	0	0	1
j	0	1	1	0	0	0	1	0	1	0



Number of Vertices and Edges

- Graph $G = (V, E)$ with vertices V and edges E
  - Whether directed or undirected, graph can have as many edges as there are pairs of vertices
  - The latter is $\binom{|V|}{2} = \frac{|V|(|V|-1)}{2}$
  - Number of edges is at most $O(|V|^2)$

Number of Vertices and Edges

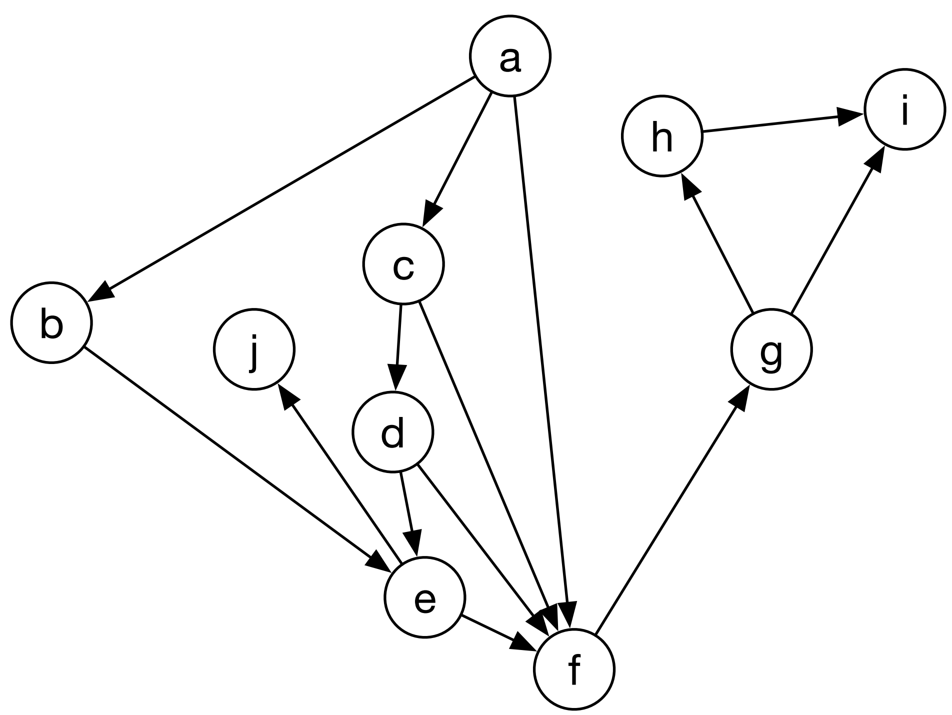
- Graph $G = (V, E)$ with vertices V and edges E
  - Graph algorithms usually need to look at each edge at least once
    - there are some idiosyncratic exceptions
    - They usually run in time at least $\Theta(|V|^2)$
  - However, many important graphs are *sparse*:
    - No edge between most pairs of vertices

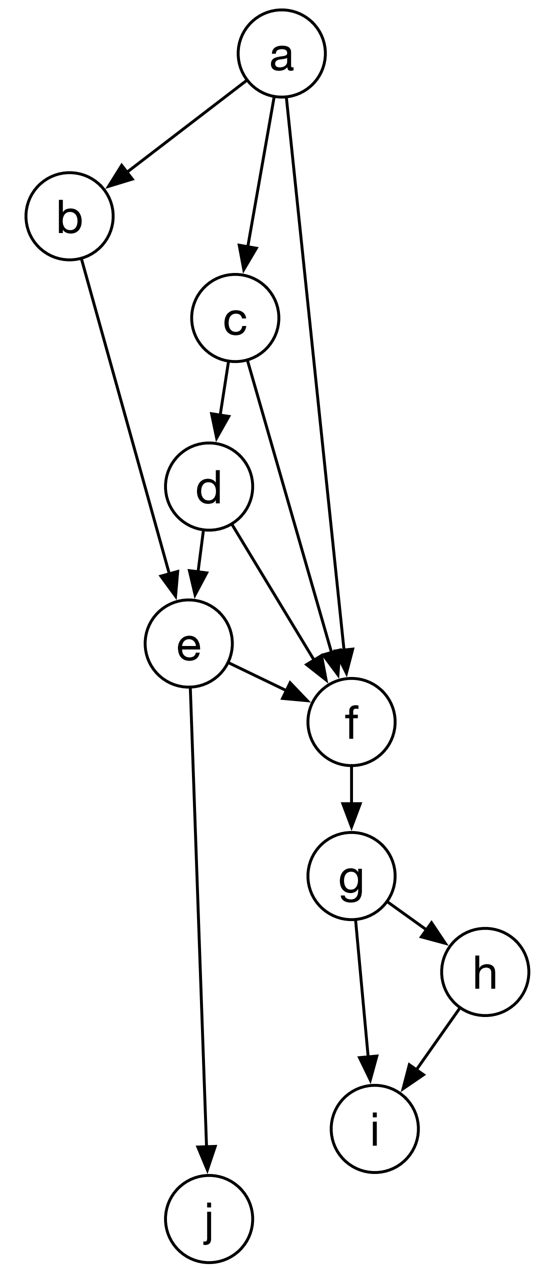
Topological Sort

- We can use a directed graph in order to represent a precedence relation
    - Topological sort:
        - Given a directed graph:
            - Order all vertices in an order such that an edge always goes from a preceding to a succeeding vertex
            - Or show that this is impossible because there is a cycle

Topological Sort

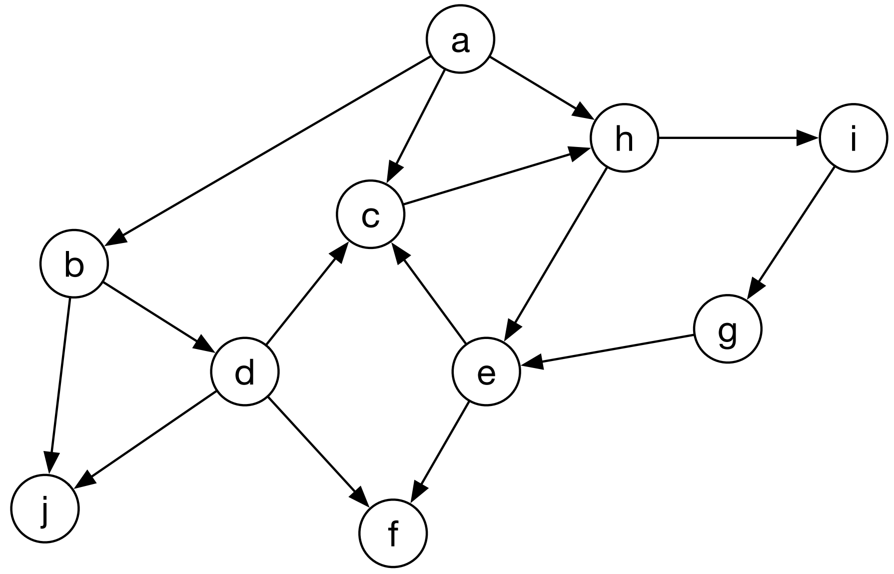
- Example 1:
  - Can arrange all vertices such that arrows only go down
  - Sort is a,b,c,d,e,f,g,h,i,j

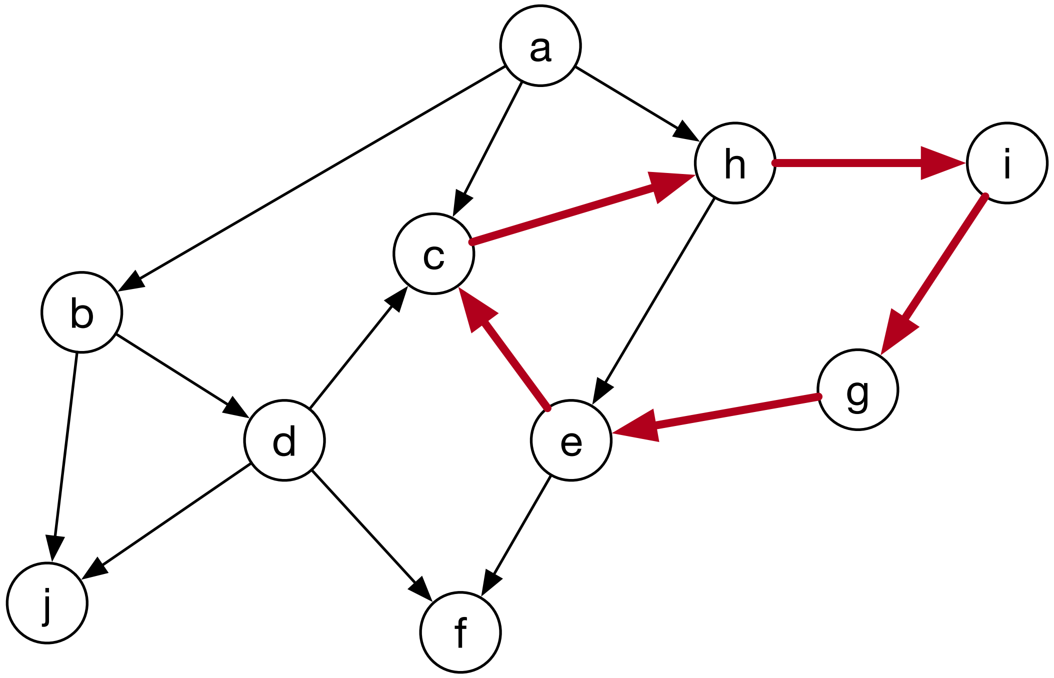




Topological Sort

- Example:
  - There is a cycle, a topological sort is not possible

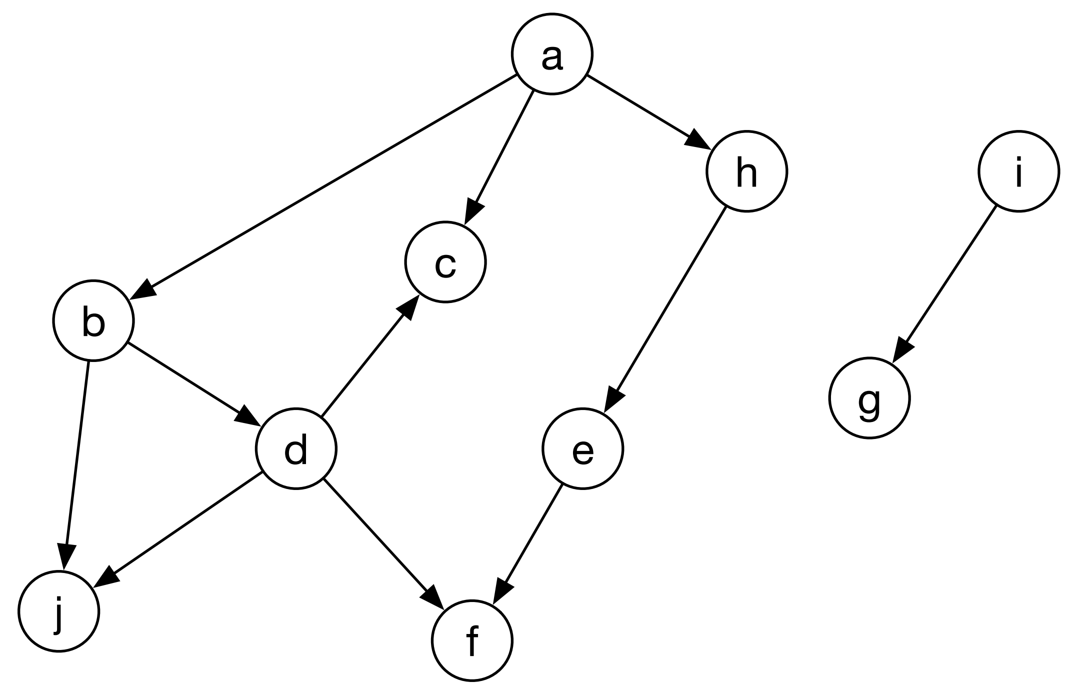




Topological Sort

- A simple algorithm:
  - Go to the adjacency list

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

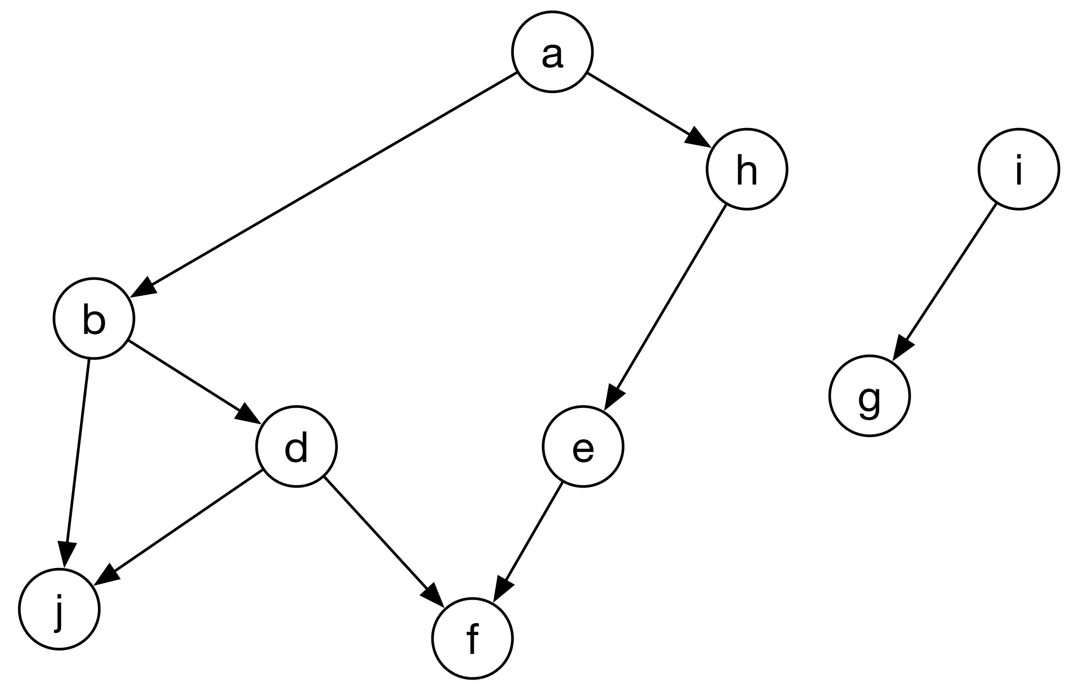


  - Find a vertex with empty list, add it to a list, and remove it from the graph

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

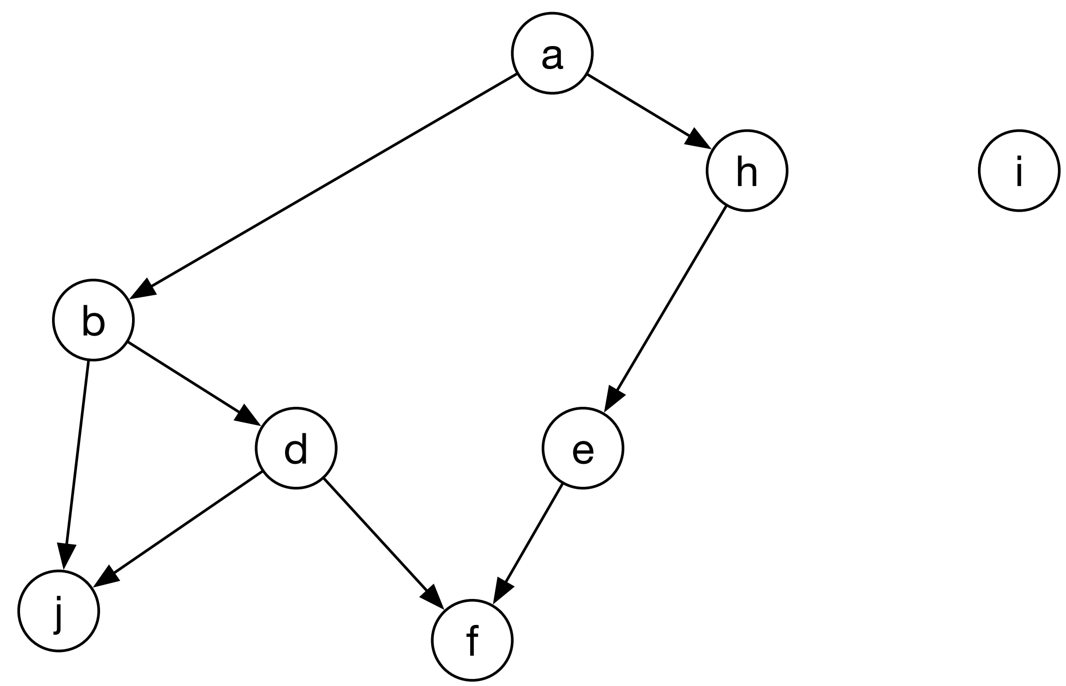


- List contains $\{c\}$

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

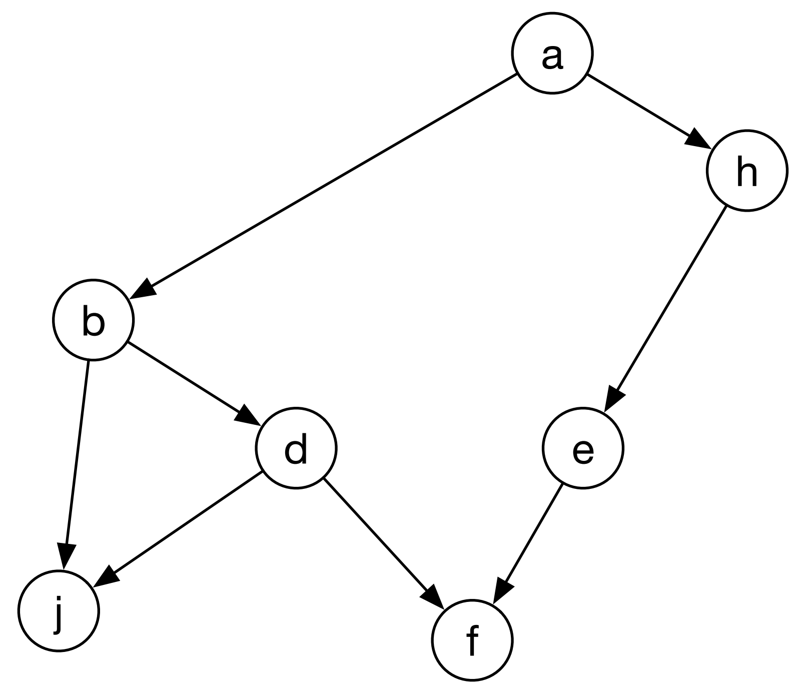


- Remove g and add it to the list $\{c, g\}$

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

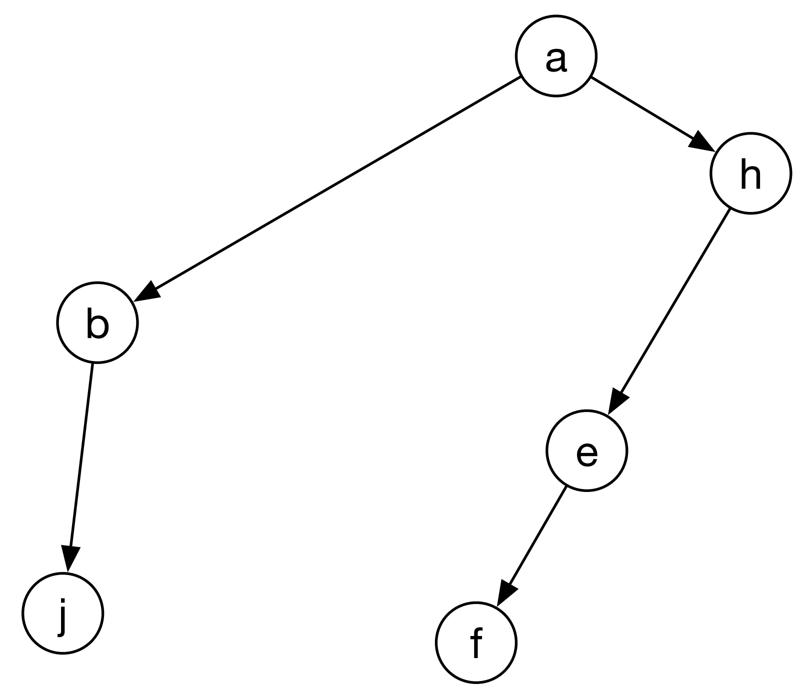


- Remove i and add it to the list $\{c, g, i\}$

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

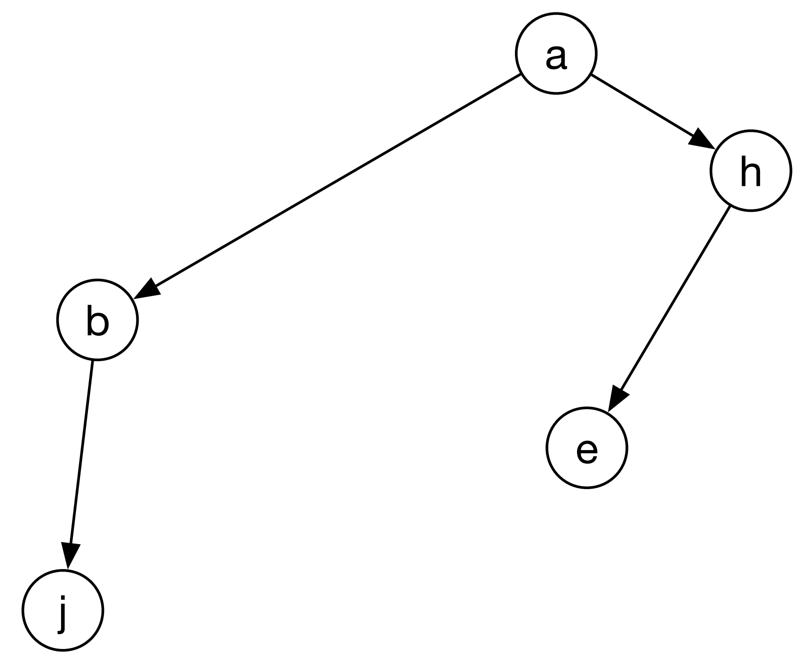


- Remove d and add it to the list $\{c, g, i, d\}$

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

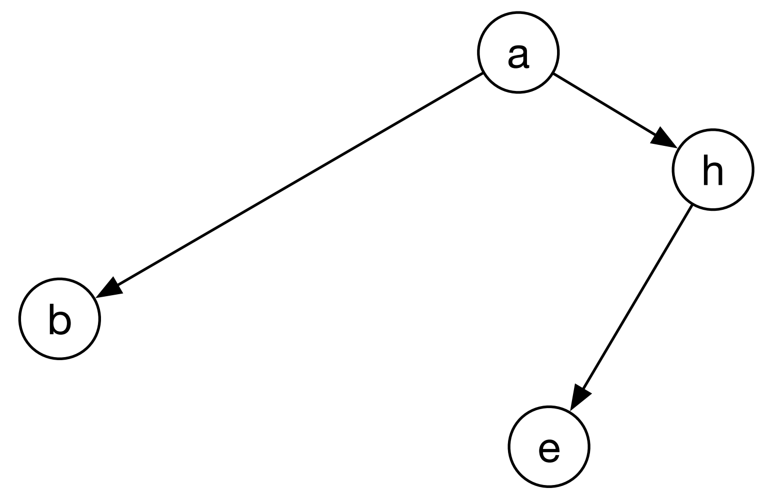


- Remove f and add it to the list $\{c, g, i, d, f\}$

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

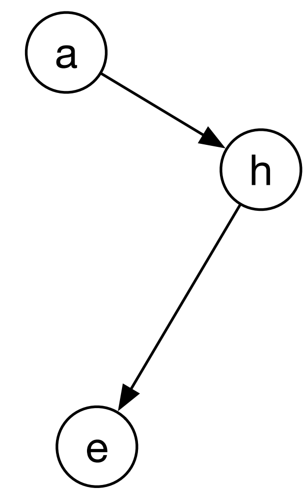


- Remove j and add it to the list $\{c, g, i, d, f, j\}$

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

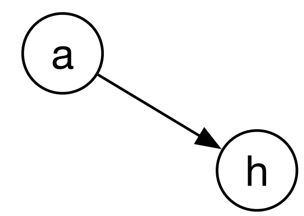


- Remove b and add it to the list $\{c, g, i, d, f, j, b\}$

Topological Sort

- A simple algorithm

```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```

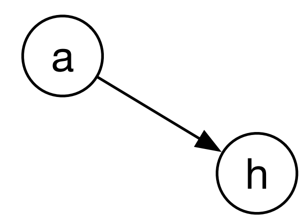


- Remove e and add it to the list $\{c, g, i, d, f, j, b, e\}$

Topological Sort

- A simple algorithm

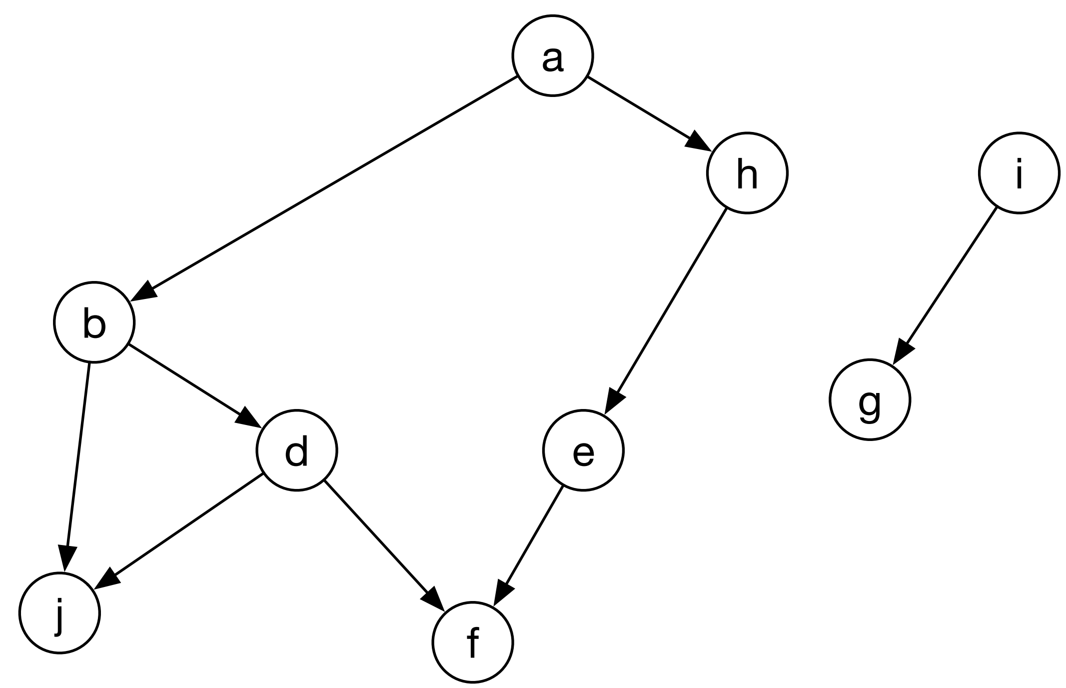
```
a: b,c,h
b: d,j
c:
d: c
e: f
f:
g:
h: e
i: g
j:
```



- Remove a and add it to the list $\{c, g, i, d, f, j, b, e, h, a\}$

Topological Sort

- The reverse list is the topological sort:
  - $\{a, h, e, b, j, f, d, i, g, c\}$



Topological Sort

- In this version, we have
  - To determine the length of the adjacency list
  - After selecting a vertex, delete that vertex from all the adjacency lists
- The latter means scanning all adjacency lists repeatedly
- This is inefficient

Topological Sort

- Question: How can we do this better?

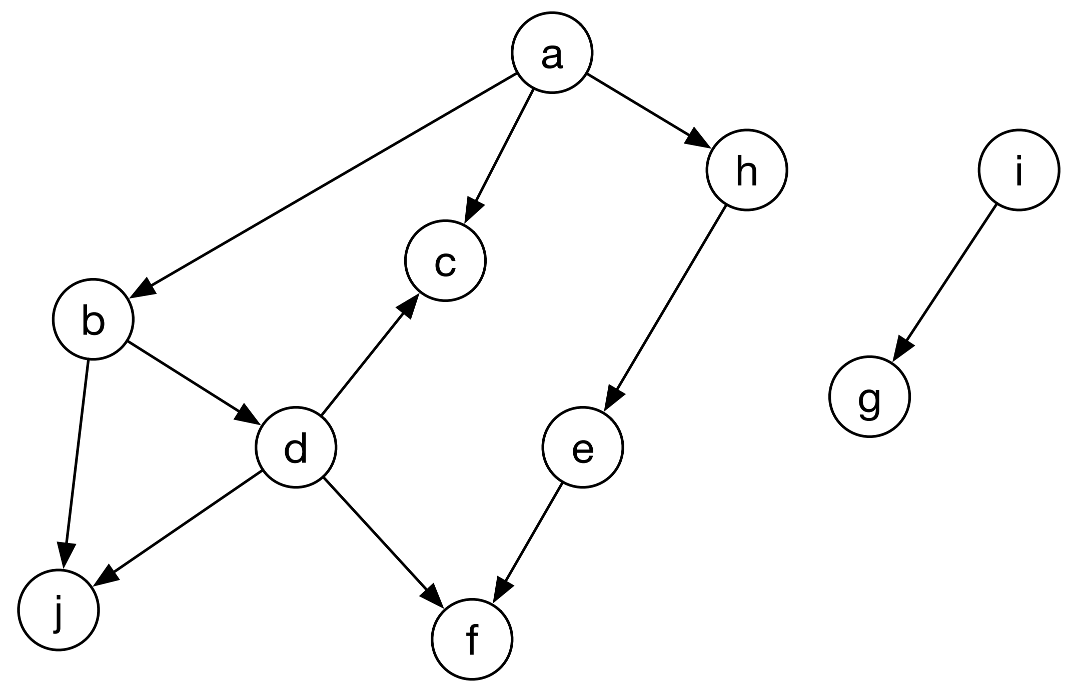
Topological Sort

- Instead of optimizing the search for vertices, we can optimize the selection of the vertex for removal
- Better algorithm:
  - Find the in-degree for all vertices
    - That is the number of edges going into a vertex
    - While there are vertices with in-degree 0
      - Remove the vertex
      - Update the in-degrees

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c, j
e: f
f:
g:
h: e
i: g
j:
```

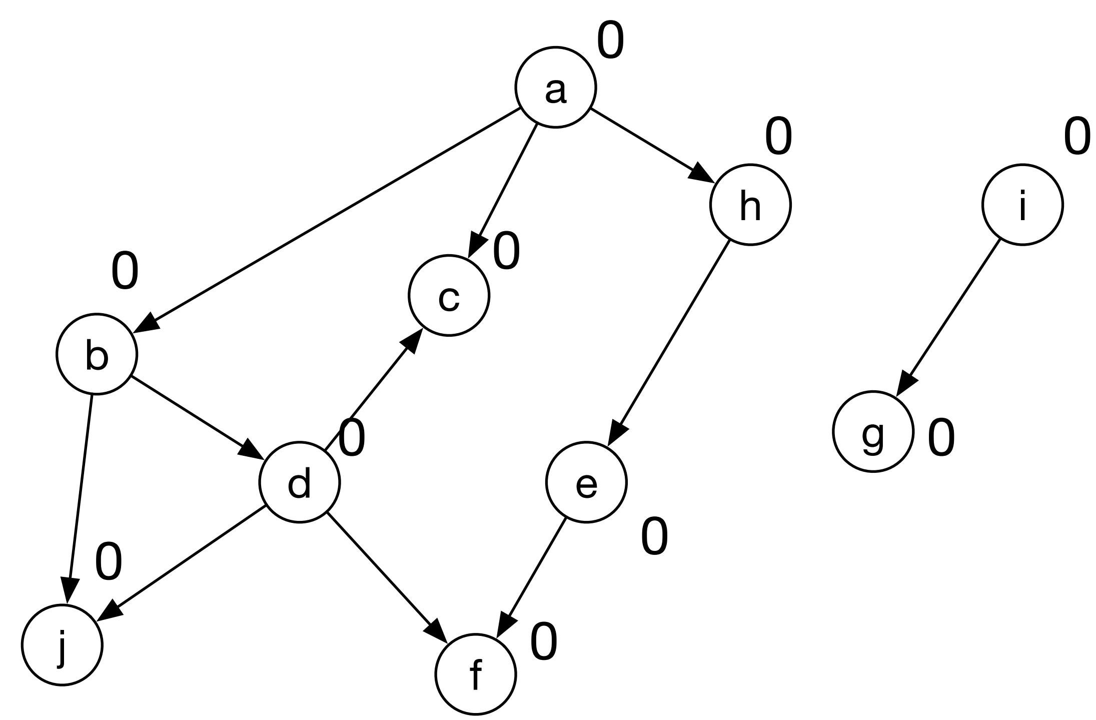


- Initialize in-degree 0 for all vertices

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

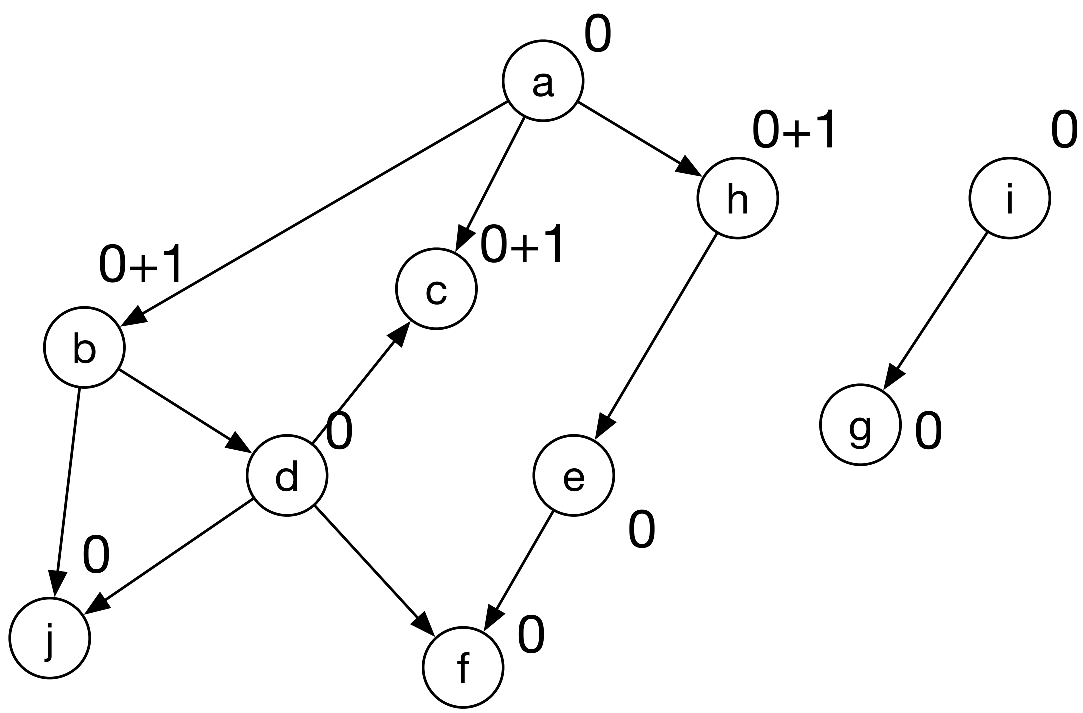


- Initialize in-degree 0 for all vertices

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

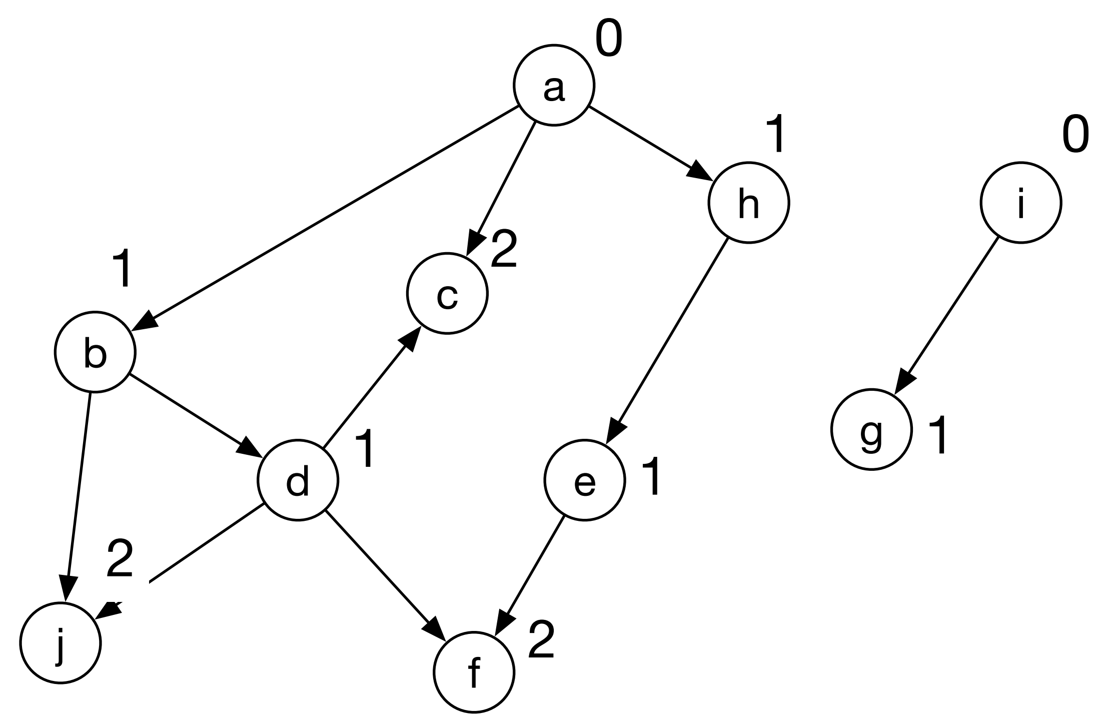


- Go through the adjacency list.
  - For each vertex in an adjacency list, add 1 to the in-degree
  - For a, we change three in-degrees

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

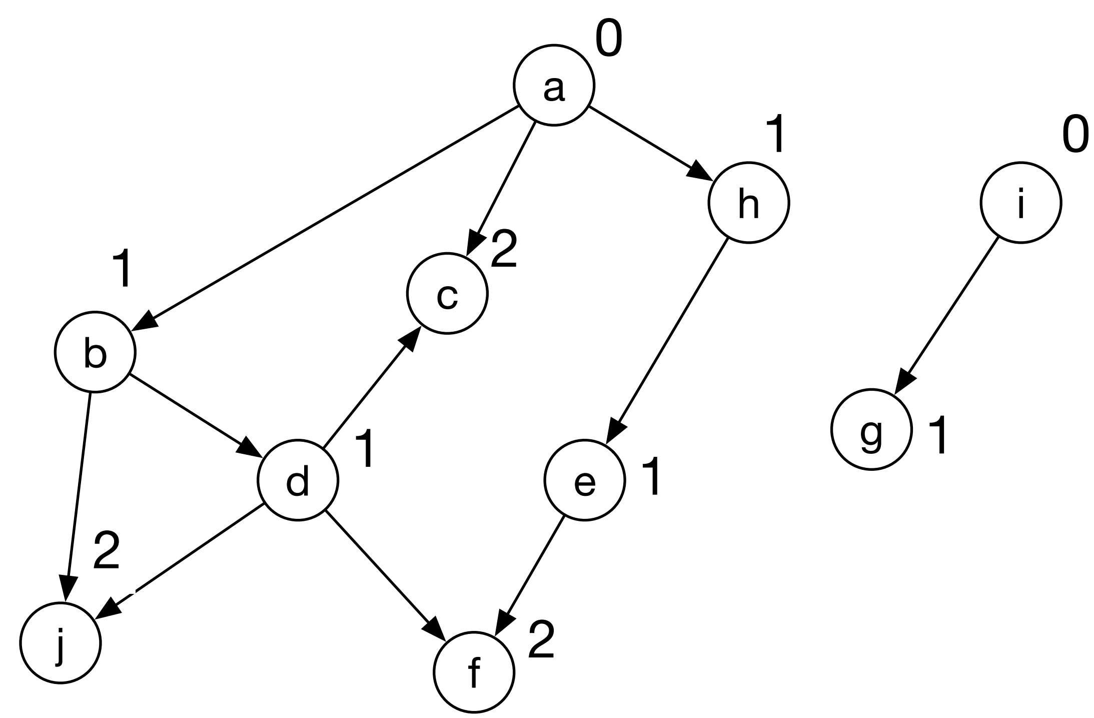


- Go through the adjacency list.
  - After processing all adjacency lists, we have the correct in-degrees

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

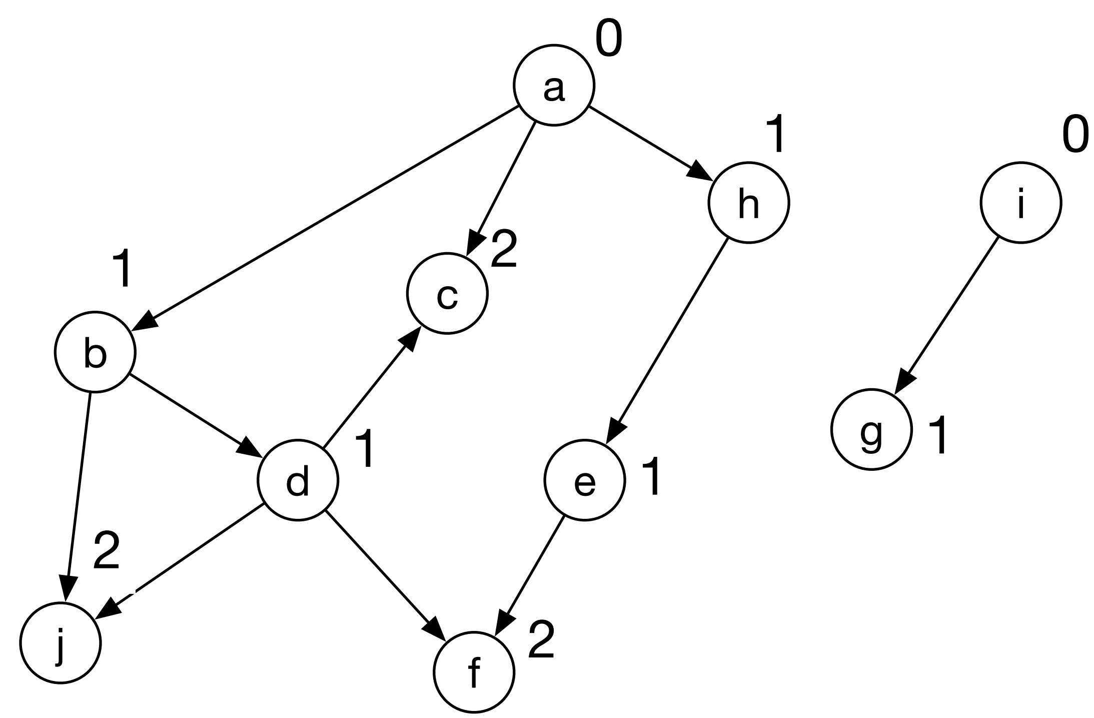


- Now we start the removal phase
  - We need to find a vertex with in-degree 0
  - How can we make this more efficient?

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

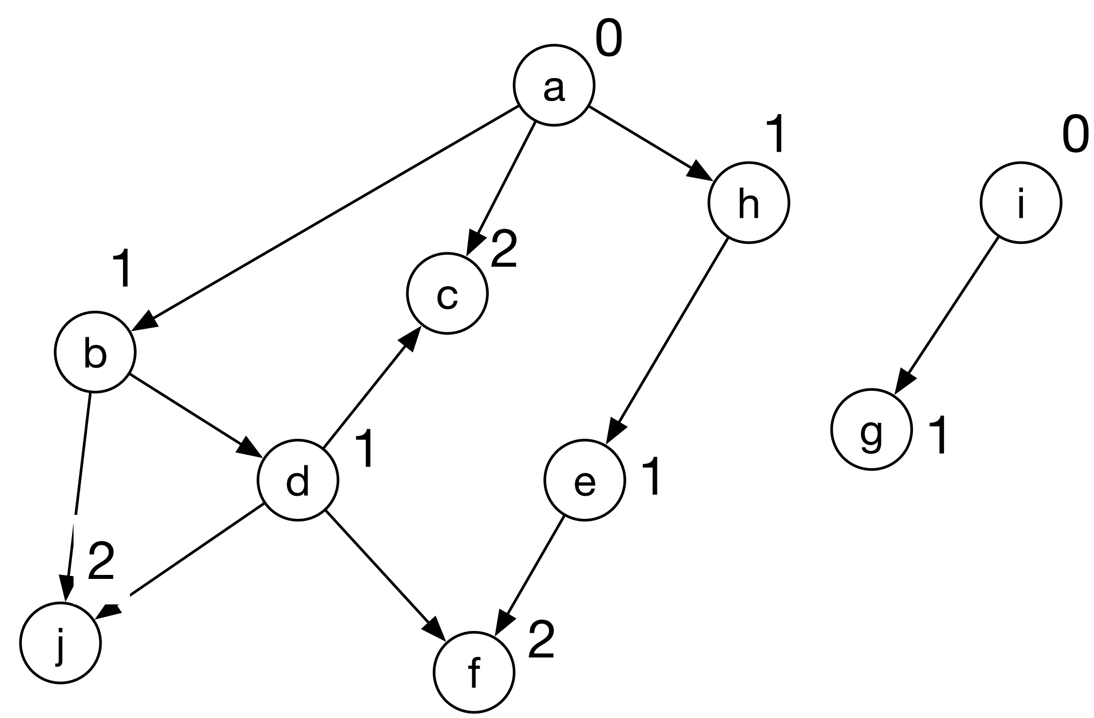


- Now we start the removal phase
  - We need to find a vertex with in-degree 0
  - Could place the vertices in a heap

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

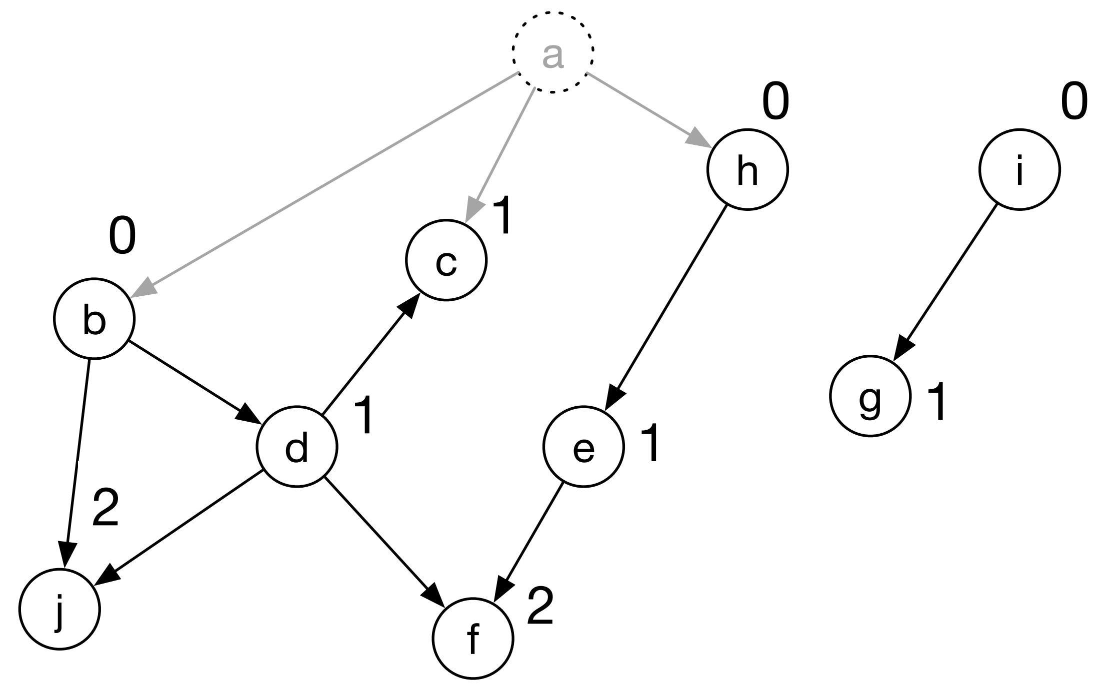


- We select a for the removal
  - We go through its adjacency list and reset the in-degrees of the nodes there

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

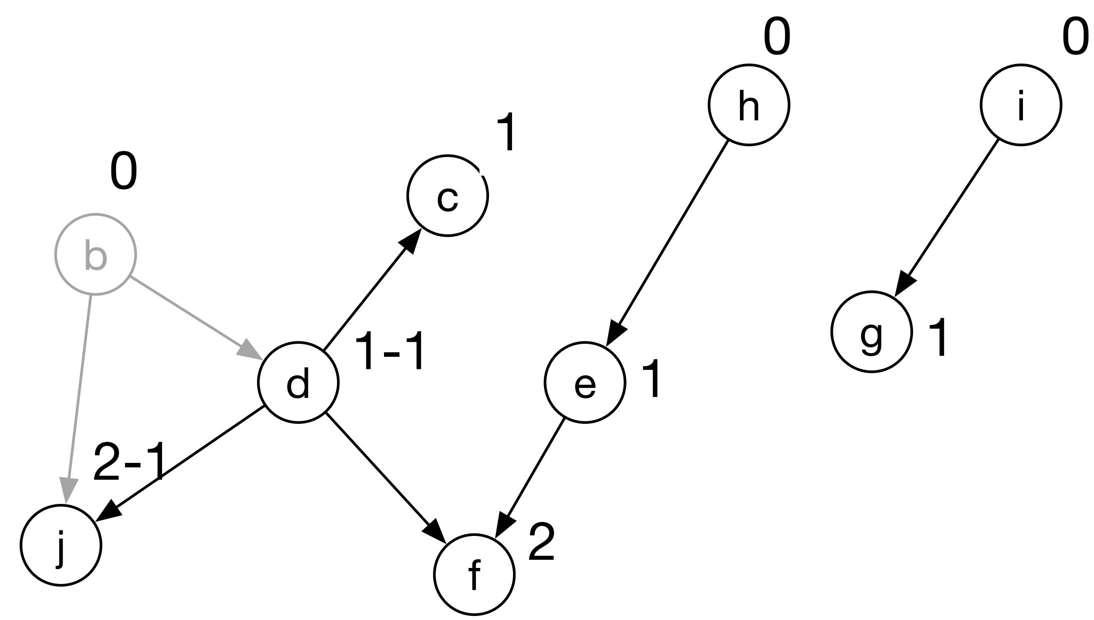


- We select a for the removal: $\{a\}$
  - We go through its adjacency list and reset the in-degrees of the nodes there

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

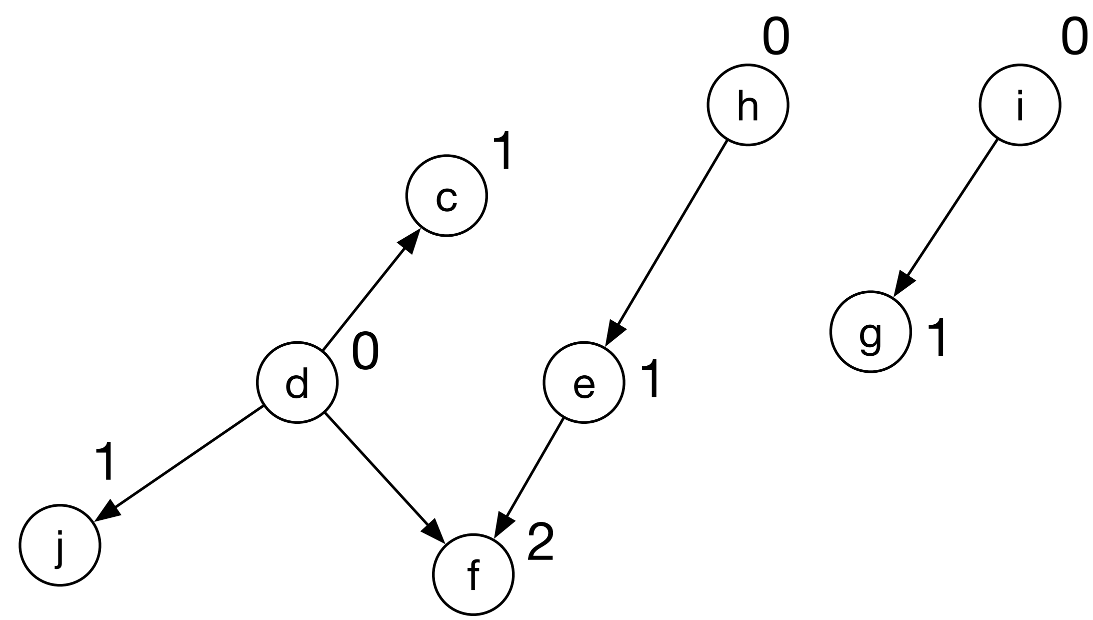


- We update our heap and select one of the 0-in-degree vertices:
  - b: $\{a, b\}$
  - and update the in-degrees of d and j

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

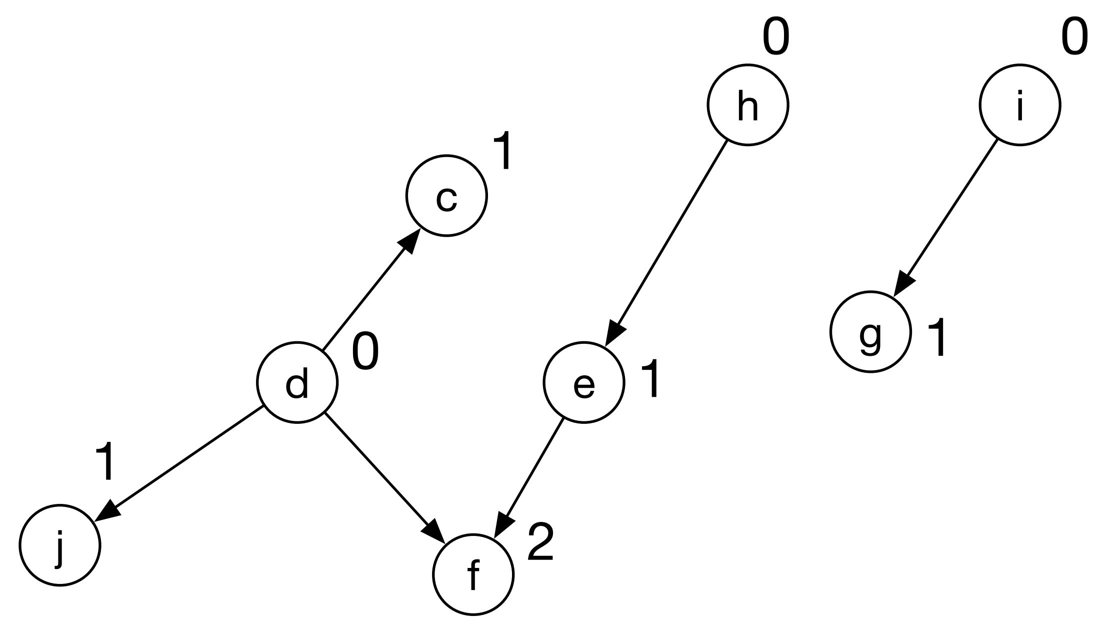


- We update our heap and select one of the 0-in-degree vertices:
  - b: $\{a, b\}$
  - and update the in-degrees of d and j

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```



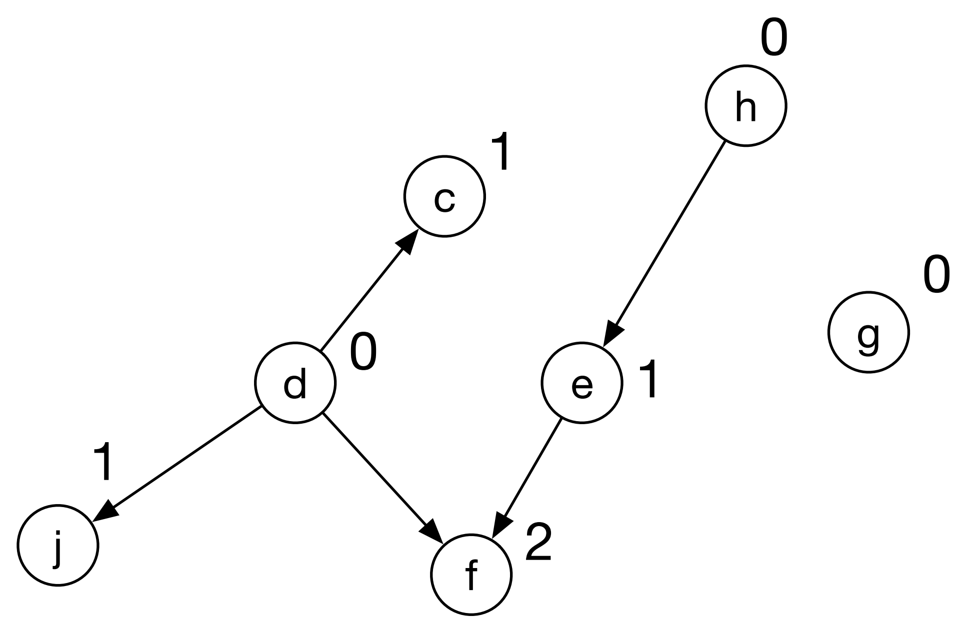
- We now randomly pick on of the vertices with degree 0, let's pick i

- Deleting it means just decrementing the in-degree of g

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

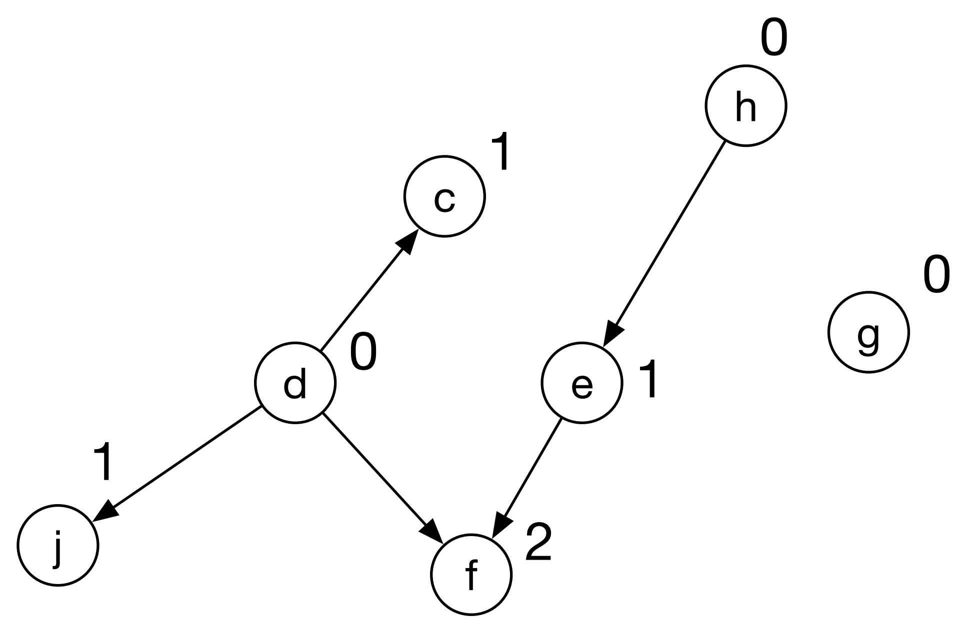


- We add g to our list $\{a, b, i, g\}$

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

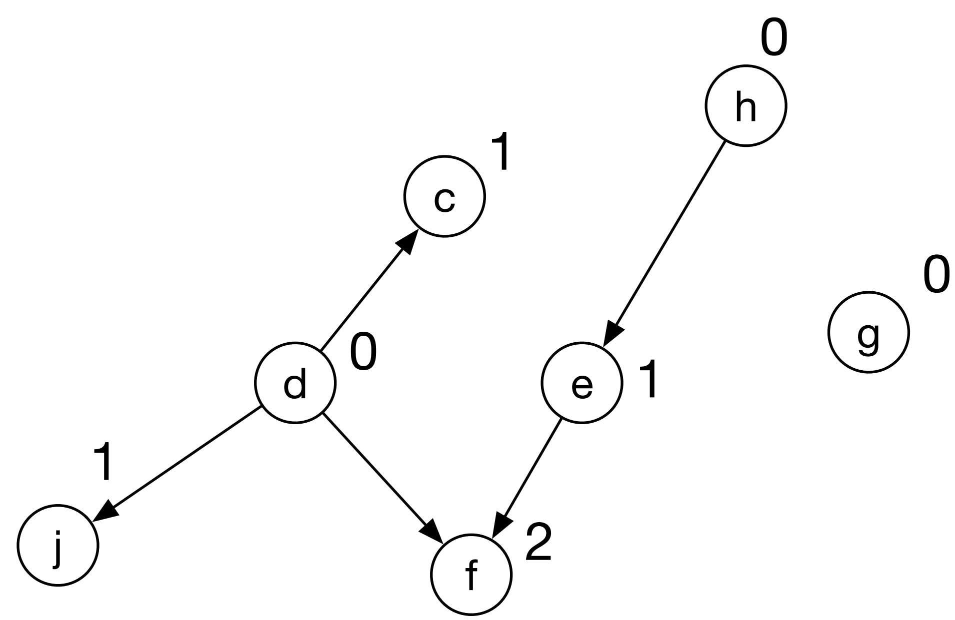


- There are three nodes with in-degree 0, let's pick h

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

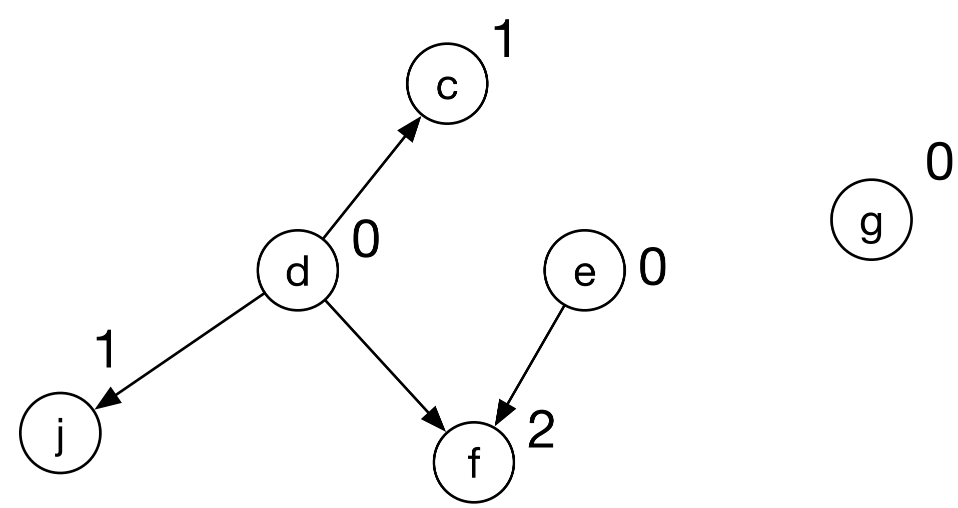


- There are three nodes with in-degree 0, let's pick h

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

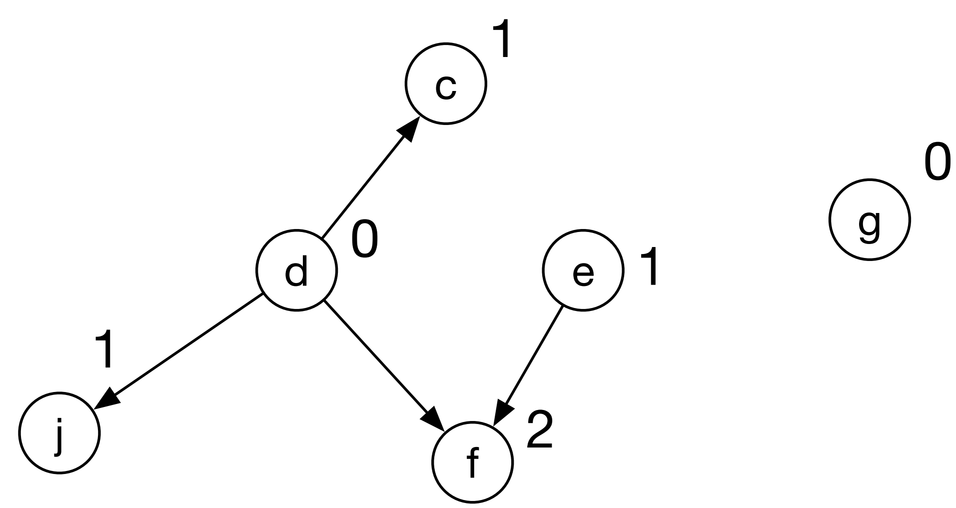


- Need to update in-degree of e
- $\{a, b, i, g, h\}$

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

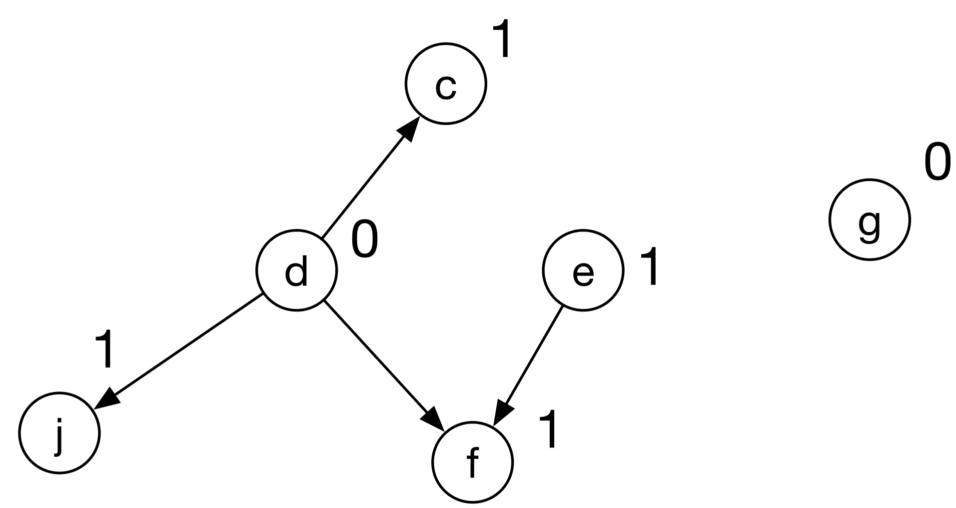


- There are two nodes with in-degree 0, let's pick d

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

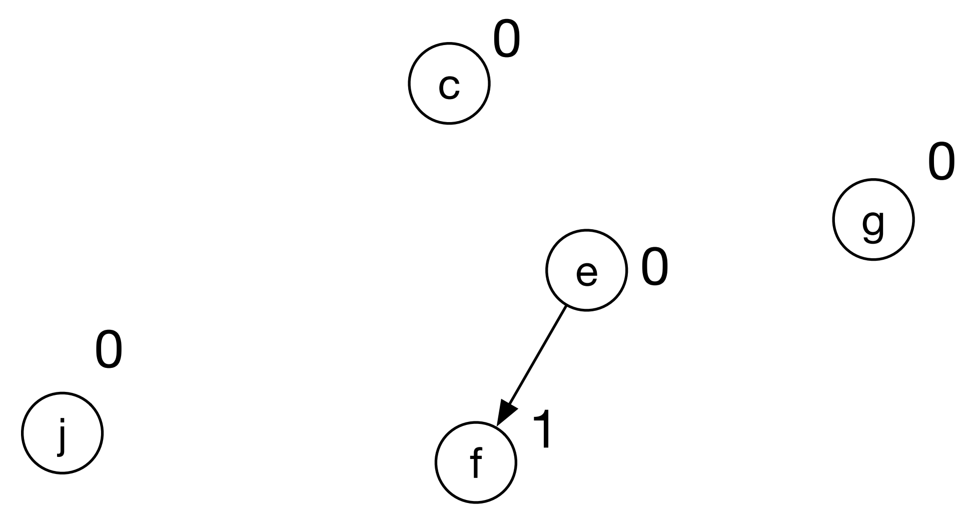


- $\{a, b, i, g, h, d\}$

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

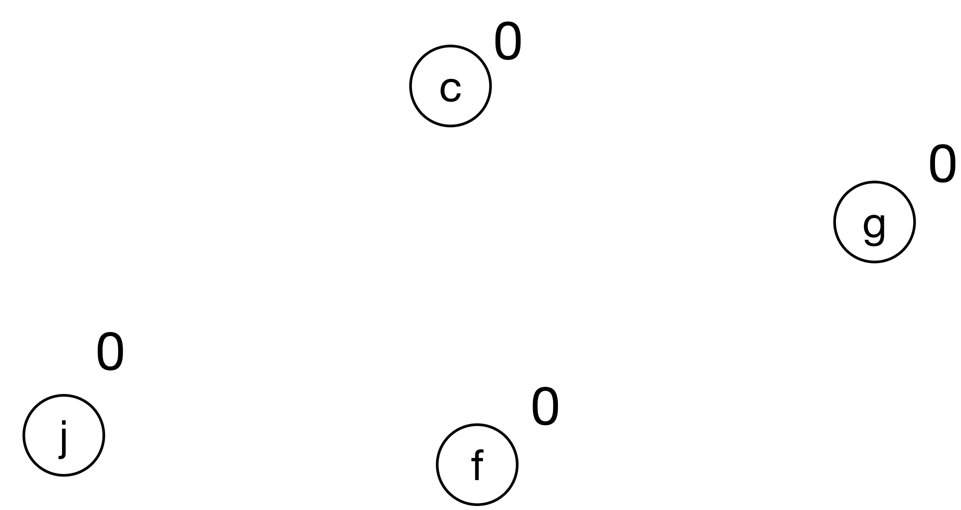


- $\{a, b, i, g, h, d\}$
- Can pick among four nodes: e

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```

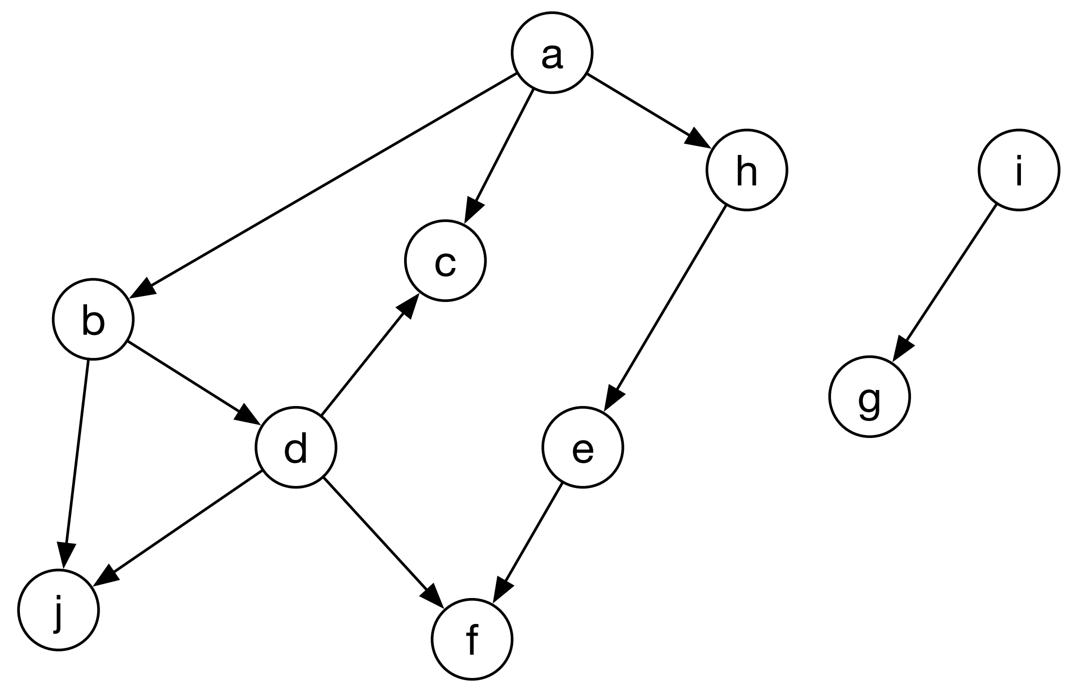


- $\{a, b, i, g, h, d, e\}$
- Can pick among four nodes in any order

Topological Sort

- Example:

```
a: b,c,h
b: d,j
c:
d: c,j
e: f
f:
g:
h: e
i: g
j:
```



- $\{a, b, i, g, h, d, e, h, j, f\}$
- Can pick among four nodes in any order

Topological Sort

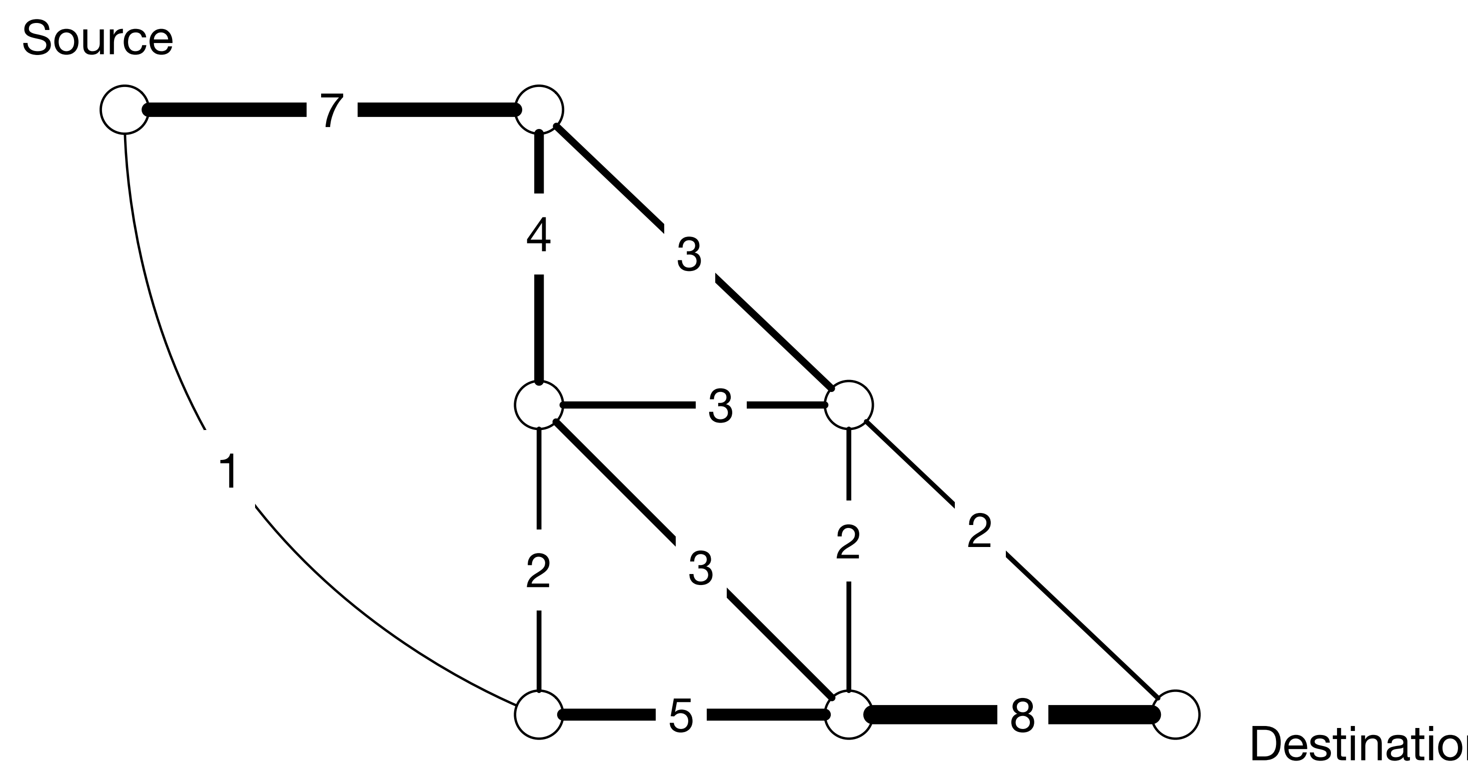
- Analysis for topological sort on $G = (V, E)$
    - Need to establish in-degrees:
        - Process all elements in an adjacency list
            - Correspond to edges
            - work $\sim |E|$
    - For each vertex:
        - find the vertex as a vertex of minimum in-degree
        - update in-degrees by going through the adjacency list
        - Latter work is $\sim |E|$ because we process each adjacency list entry once
        - Delete the adjacency list
        - Work is $\sim V$

Topological Sort

- This algorithm is *almost* $O(|E|)$ but for finding the minimum in-degree
- We will see a better algorithm shortly

Weighted Graphs

- Graphs with edge weights
  - Often, graphs in CS have edge weights
    - Example: edge weight indicates the size of a pipeline
      - such as network connection, capacity of roads, etc.



How much can you pump from source to destination if the pipes have the indicated capacities (Flow Problem)

Weighted Graphs

- Graphs with edge weights
  - Weights can indicate distance
    - What is the shortest distance from source to destination