Graph algorithms

CS/ENGRD 2110
Object-Oriented Programming
and Data Structures

Scott McElfresh

Today

- Reachability
  - Depth-First Search
  - Breadth-First Search
- Shortest Path
  - Unweighted graphs
  - Weighted graphs
  - Dijkstra’s algorithm

Graphs

- **Graph** G = <V,E>
  - Set V of **vertices** (**nodes**)
  - Set E of **edges**
    - Elements of E are pairs (v,w) with v and w in V
    - Directed graph: ordered pairs
    - Undirected graph: unordered pairs
- **Weighted** Graph
  - Elements of E are (v,w,x) where x is a weight.

- |V| = size of V, often denoted n or N
- |E| = size of E, often denoted m or M

Paths and cycles

- A **path** is a sequence of nodes
  - $v_1, v_2, \ldots v_n$ such that (v_i, v_{i+1}) in E for $0<i<N$
  - The **length** of the path is N-1
  - **Simple path**: all v_i are distinct for $0<i<=N$
- A **cycle** is a path such that $v_1 = v_n$
  - An **acyclic** graph has no cycles.
- A graph is **connected** if
  - Given any two vertices v_i and v_j, there exists a path from v_i to v_j

Representations

- How do we represent a graph internally?
- Option 1: **Adjacency Matrix**
  - An NxN matrix (where N = |V|)
  - M[i,j] = 1 if there is an edge from v_i to v_j
  - M[i,j] = 0 if there is not
  - $O(N^2)$ for space
    - okay for **dense** graphs ($M = O(N^2)$ (ie, quadratic number of edges)
    - expensive for large **sparse** graphs (number of edges not quadratic)
    - Most real-world graphs are sparse (linear number or slightly larger)

Representations

- How do we represent a graph internally?
- Option 2: **Adjacency List**
  - An N element array, i'th element is a linked list to represent adjacent edges on v_i
  - Each edge is a list node. The number of list nodes equals the number of edges.
  - O(M) space (linear in the size of the graph)
    - O(M) space for the list nodes
    - O(N) for the vertex list
    - Typically, M > N

Reachability Algorithms

- **Depth First Search (DFS)**
  - Explore nodes by going deeper and deeper into the graph. Use back tracking to try different paths (uses a stack).
- **Breadth First Search (BFS)**
  - Explore the nodes in an orderly manner. Look at the nodes that are closest to the source. Then look at their neighbors, etc. (uses a queue)

DFS algorithm

- Let R be the set of vertices reachable from a starting node x.   Let S be a stack.

```
DFS(vertex x){
      S.push(x)
      while (S is not empty){
         u = s.pop()
         if (u is not in R) {
            put u into R
            for all (u,v) in E {
               S.push(v)
            }
         }
      } // end while
}
```

Note: a node can end up in the stack more than once.

Recursive DFS

```
DFS (vertex x){
    put x into R
    for all  (x,y) in E
        if (y is not in R)
            DFS (y)
}
```

Finding Cycles in a graph

- When does a graph have a cycle?
- If the graph is connected and every node has out-degree at least 1, then the graph has a cycle.
- Informally,
  - Start from any node and walk through the graph.
  - Since you can go out from any node, you can touch all the nodes and you will eventually run into a node that you have already visited.

Finding a cycle – using DFS

- Modify DFS. Use colors to keep track of
  - Nodes that are not visited
  - Nodes we are visiting now (are not finished exploring all of it's out edges)
  - Node that are already visited (finished exploring all out edges)
- If DFS runs into a node that we are still visiting, then we have a cycle.

```
Color all vertices White

DFS (vertex x){
   color x to be gray // in process
   put x into R
   for all  (x,y) in E
        if (y is not in R)
           DFS (y)
   color x to be black // finished processing
}
```

BFS algorithm

- Let R be the set of vertices reachable from a starting node x.   Let Q be a queue.

```
BFS(vertex x){
      Q.enqueue(x)
      while (Q is not empty){
         u = Q.dequeue()
         if (u is not in R) {
            put u into R
            for all (u,v) in E {
                Q.enqueue(v)
            }
         }
      } // end while
}
```

Thought Problem

- How can DFS and/or BFS help us with Topological Sorting?

Single Source, Shortest Paths

Problem: Given a graph G=(V,E) compute the distances of each vertex x from a source vertex **s**, where distance is the length of the shortest path.

Unweighted Graph

dist[s] = 0;
…
dist[y] = dist[x] + 1,
    where (x,y) in E

Weighted Graph

dist[s] = 0;
….
dist[y]= dist[x]+w(x,y)
    where (x,y) in E

Many applications

- Shortest paths model many useful real-world problems.
  - Minimization of latency in the Internet.
  - Minimization of cost in power delivery.
  - Job and resource scheduling.
  - Route planning.

Claim: The shortest path is a **simple** path. (ie, no vertex is repeated in the list)

Claim: There are only a finite number of simple paths in a given graph.

Brute Force

Enumerate all simple paths starting at s.

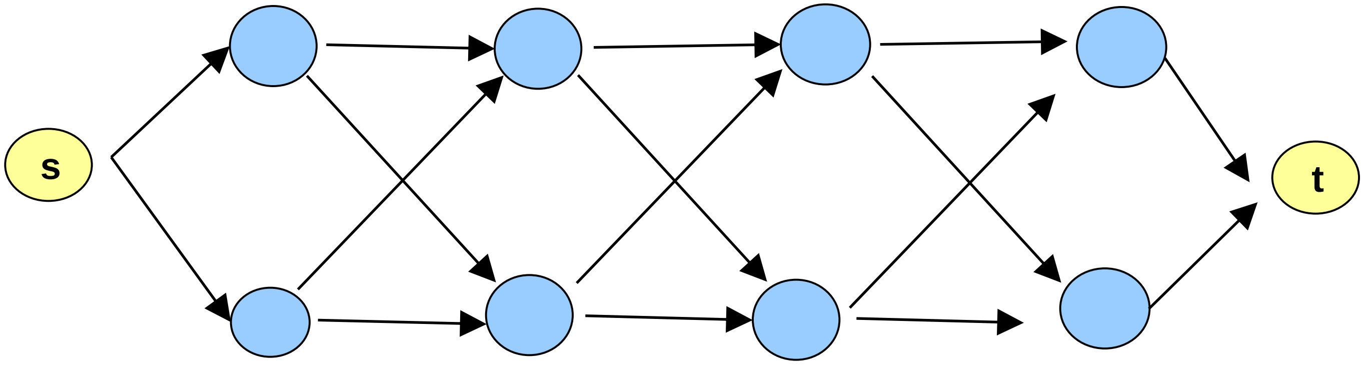
For each target vertex t, collect all simple paths with target t.

Compute their cost, determine the min.

Bad Idea

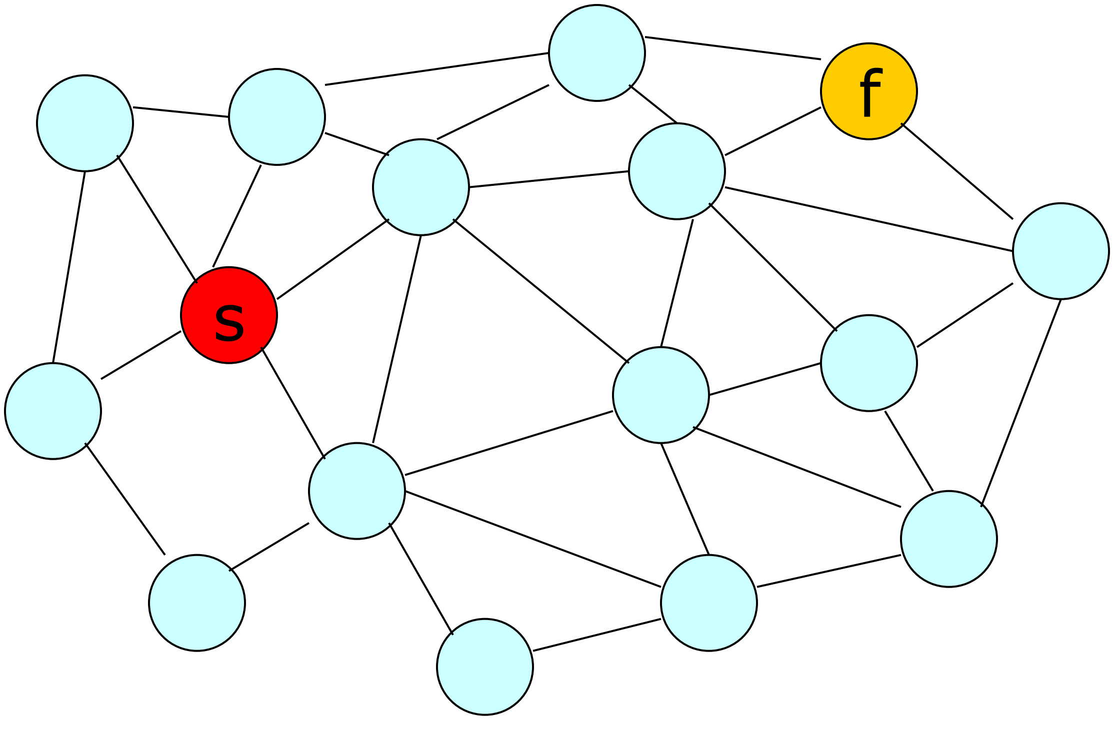
Even in an acyclic graph, the number of simple paths may be exponential in n.

Exercise: determine the number of paths s to t.



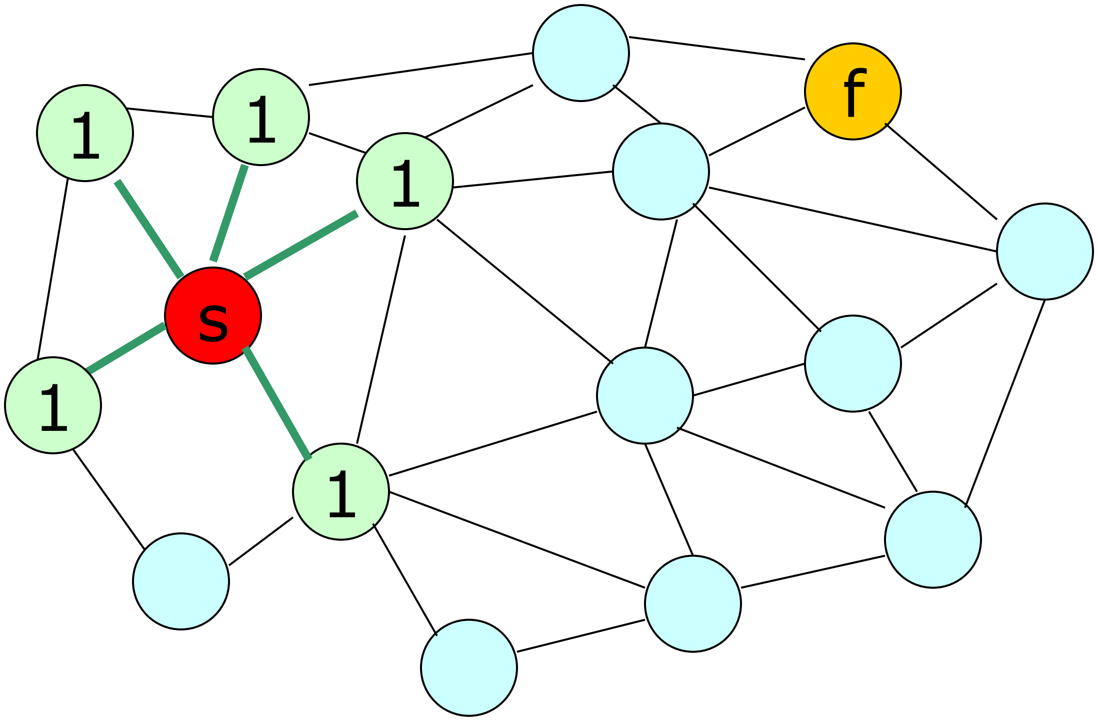
Single Source, Shortest Paths

- Unweighted graphs: BFS
  - Modified to keep track of current distance from **s**



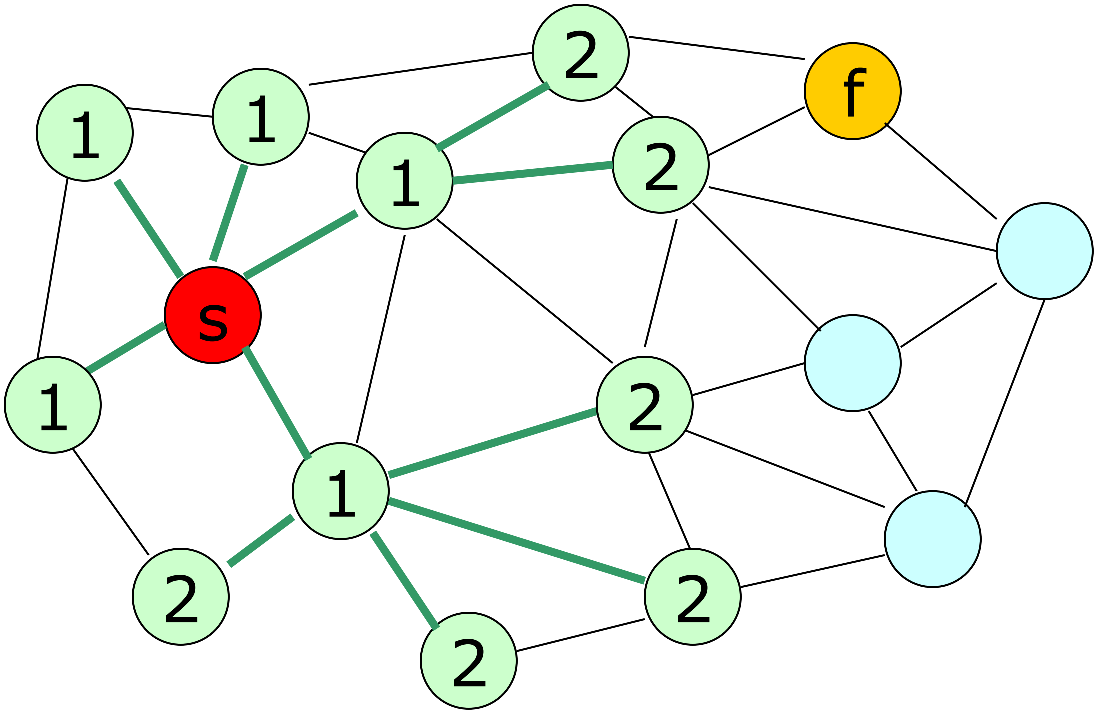
Single Source, Shortest Paths

- BFS
  - First, visit all nodes at distance 1



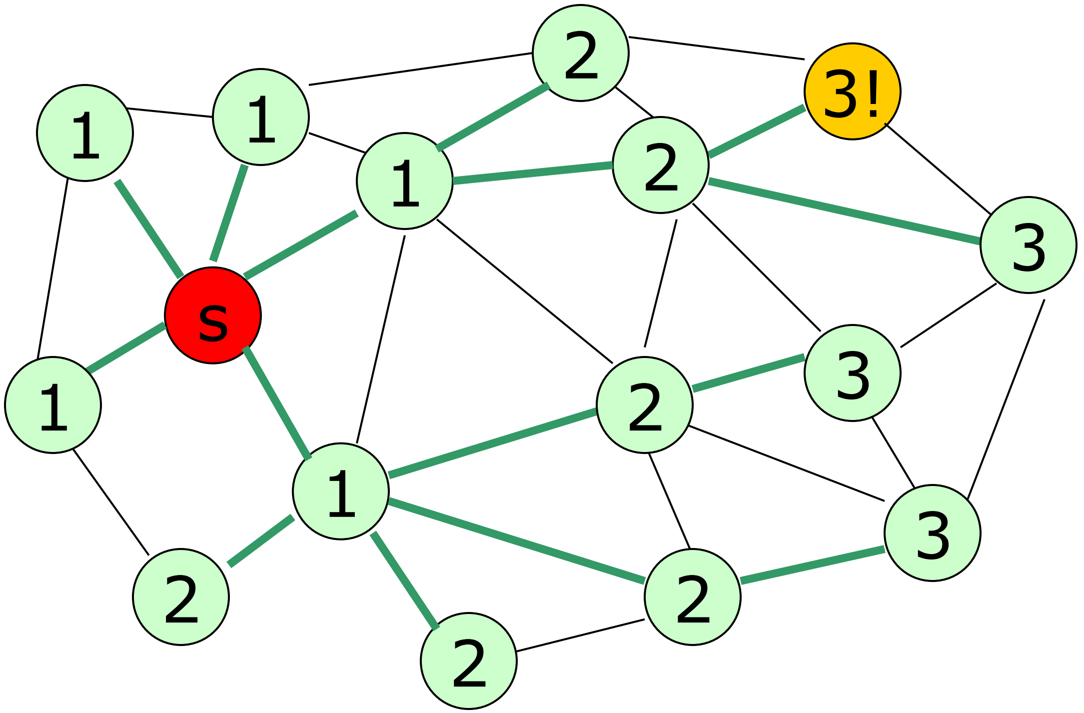
Single Source, Shortest Paths

- BFS
  - First, visit all nodes at distance 1
  - Then, distance 2 …



Single Source, Shortest Paths

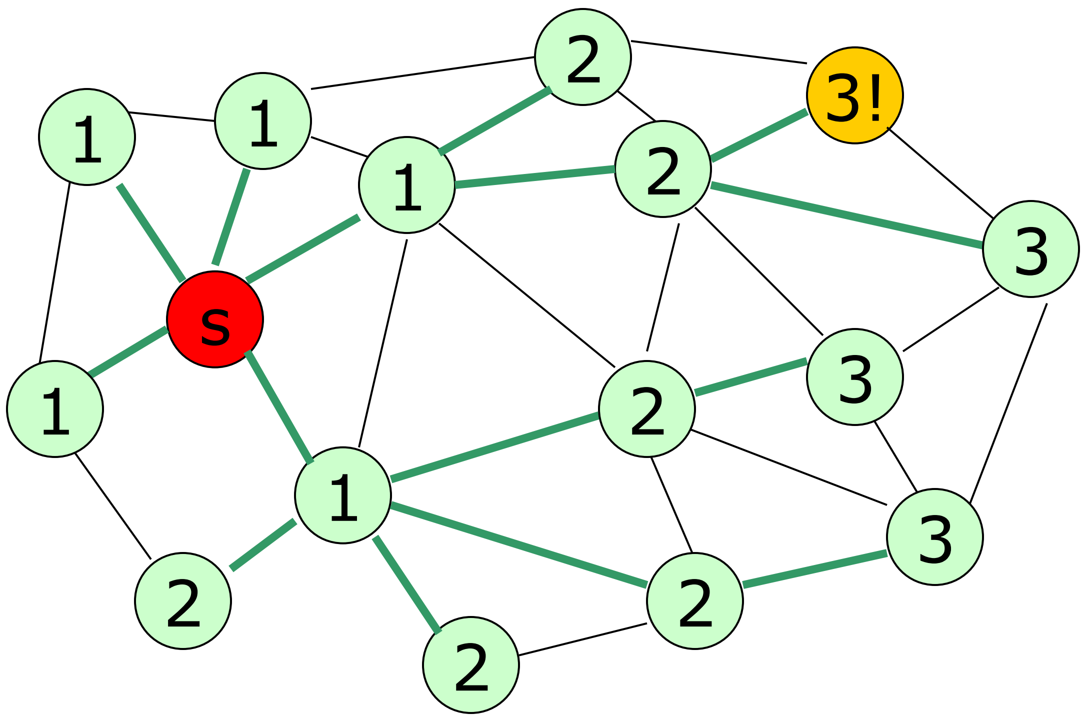
- BFS
  - First, visit all nodes at distance 1
  - Then, distance 2 …
  - Then, 3 …



Single Source, Shortest Paths

- Note: we have actually calculated shortest path from **s** to **every** node in graph!
  - Not just from **s** to **f**

*In general, computing shortest paths from **s** to every other node is just as expensive as computing the shortest path between any given pair of nodes*



BFS for shortest path

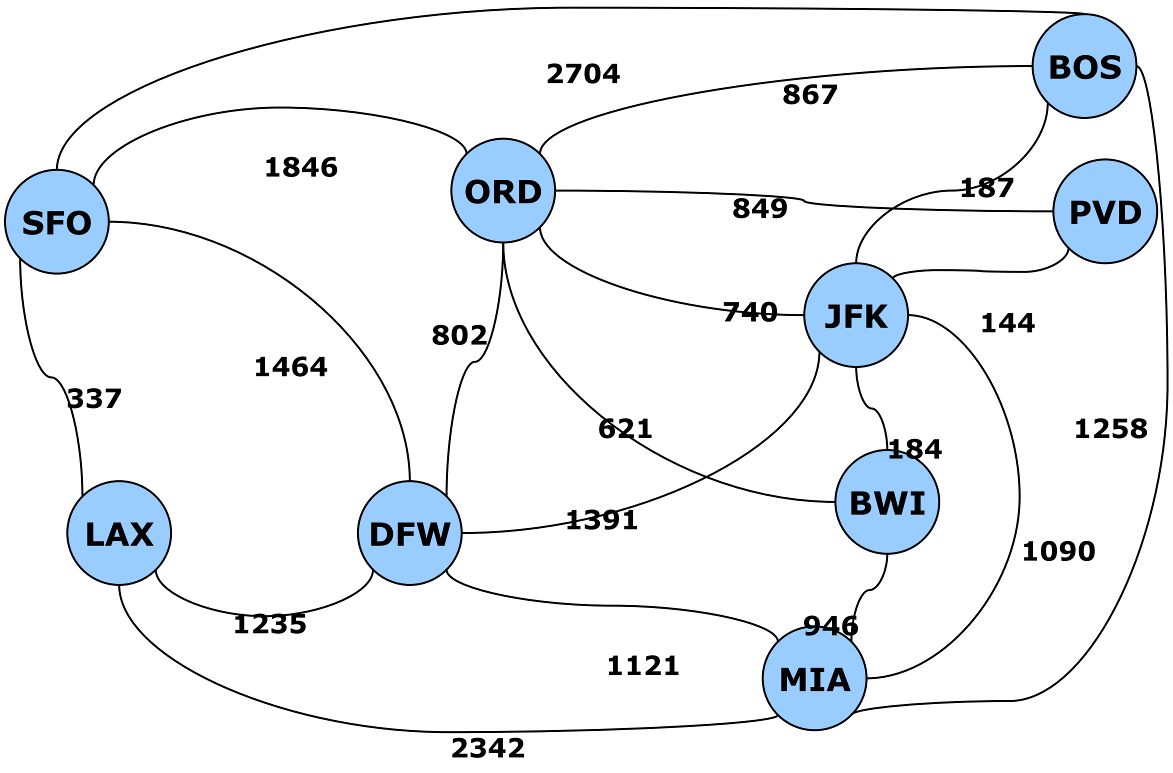
```
for each vertex x
    dist[x] = infinity;   // will represent distance from s to x
Q.enqueue(s);
dist[s] = 0;

while (! Q.empty())
    x = Q.dequeue();
    for all (x,y) in E
        if (dist[y] = infinity)
            dist[y] = dist[x] + 1;
            Q.enqueue(y);
```

Claim: $O(n + m)$ runtime

- Will DFS work in this context?
- Consider a weighted graph where all edge weights are equal.
  - Use the same BFS algorithm.
- What about a graph with different weights on edges?

Weighted Edges, Shortest Paths

- BFS algorithm is only relevant for **unweighted** graphs
- What about weighted graphs?



General Rules

We maintain an array dist[x]:

- initially dist[s] = 0, dist[x] = ∞ for all other vertices

- at any time during the algorithm, we store the cost of a real path from s to x in dist[x] (but not necessarily the cost of the shortest path, we may have an overestimate).

- edge (x,y) requires attention if

$$\text{dist}[x] + \text{cost}(x,y) < \text{dist}[y]$$

Prototype Algorithm

When an edge requires attention we relax it:

dist[y] = dist[x] + cost(x,y)

Thus we now have a better estimate for the shortest path from s to x. This produces a prototype algorithm:

```
initialize dist[];

while( some edge (x,y) requires attention )
    relax  (x,y);
```

Correctness

Claim: Upon completion of the algorithm dist[x] is the correct distance from s to x, for all x.

Proof:
Suppose otherwise, pick x such that the path from s to x has minimal length (number of edges, not weights). Then there is some vertex y such that (y,x) is an edge, dist[y] is correct and dist[y] + cost(y,x) < dist[x].

But then (y,x) requires attention, contradiction.

Termination

Claim: The algorithm always terminates.

Proof:
Suppose otherwise. Then there is one edge (x,y) that is relaxed infinitely often.

But then there must be infinitely many simple paths from s to y, contradiction.

Make sure you understand why the paths must be simple.

Dijkstra's Algorithm

The problem is to choose the right edge to be relaxed.

Dijkstra's algorithms always picks the edges (x,y) such that dist[x] is minimal – but works on each x only once.

This sounds like a recipe for disaster, how do you know that there are no shortcuts that will be discovered later?

Dijkstra's Algorithm

```
initialize dist[];
insert all v in V into PQ;
     // priorities:  dist

while( PQ not empty )
     x = PQ.deleteMin( );
     forall  (x,y) in E  do
          if( (x,y) requires attention )
               relax edge
```

Dijkstra's Algorithm

```
initialize dist[];
insert all v in V into PQ;
          // priorities:  dist

while( PQ not empty )
   x = PQ.deleteMin(  );
   forall  (x,y) in E  do
      // if (x, y) requires attention
      if( dist[x] + cost[x,y] < dist[y] );
            // relax edge - update our current estimate
            // of distance from s to y

         dist[y] = dist[x] + cost[x,y];
         PQ.promote( y );
```

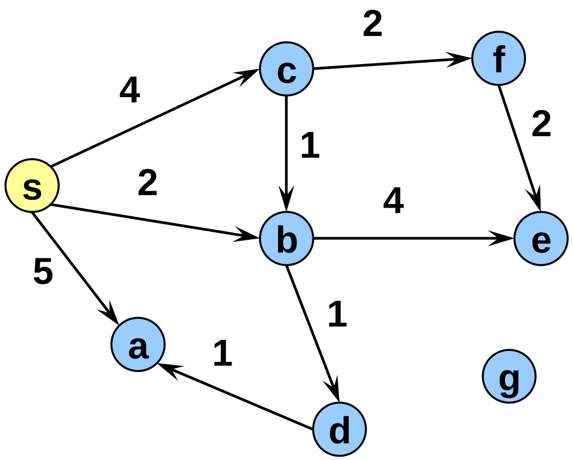
Dijkstra's algorithm

Initialization

a. Set ***dist*(*s*) = *0***

b. For all vertices ***v*** ∈ ***V***, ***v*** ≠ ***s***, set ***dist*(*v*) = ∞**

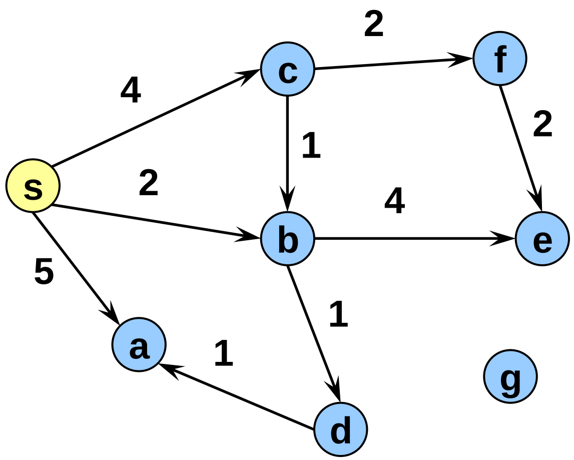
c. Insert all vertices into priority queue ***P***, using distances as the keys



s	a	b	c	d	e	f	g
0	∞	∞	∞	∞	∞	∞	∞

P

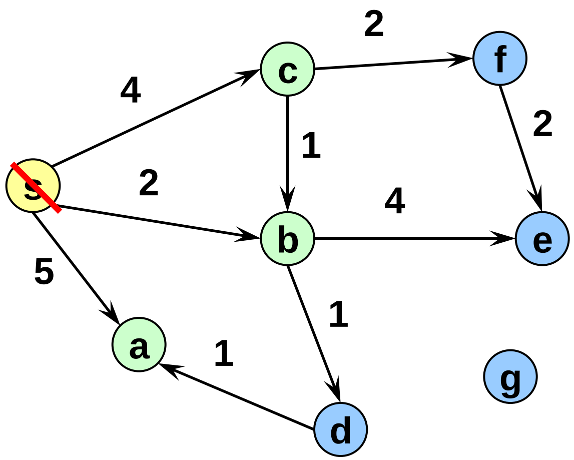
Dijkstra’s algorithm



Processed

s	a	b	c	d	e	f	g
0	∞	∞	∞	∞	∞	∞	∞

Dijkstra's algorithm

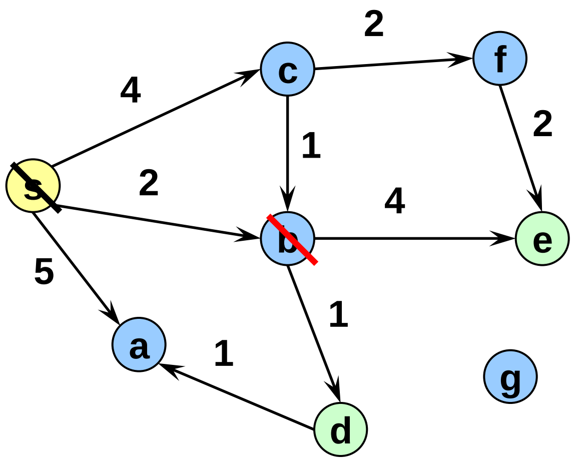


Processed

s (D = 0)

b	c	a	d	e	f	g
2	**4**	**5**	∞	∞	∞	∞

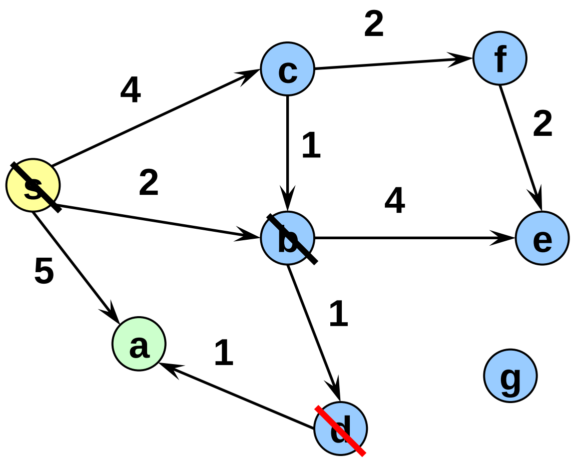
Dijkstra's algorithm



Processed
s (D = 0)
b (D = 2)

d	c	a	e	f	g
3	4	5	6	∞	∞

Dijkstra’s algorithm



Processed

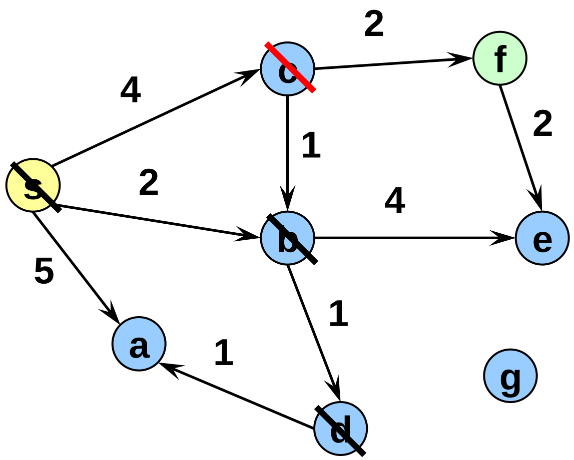
s (D = 0)

b (D = 2)

d (D = 3)

c	a	e	f	g
4	**4**	**6**	∞	∞

Dijkstra’s algorithm

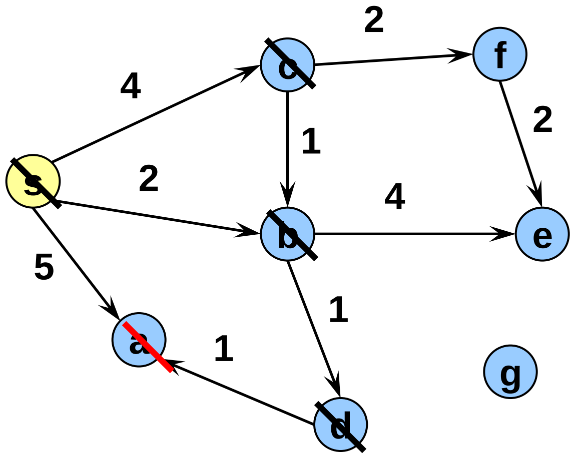


Processed

s (D = 0)
b (D = 2)
d (D = 3)
c (D = 4)

a	e	f	g
4	**6**	**6**	∞

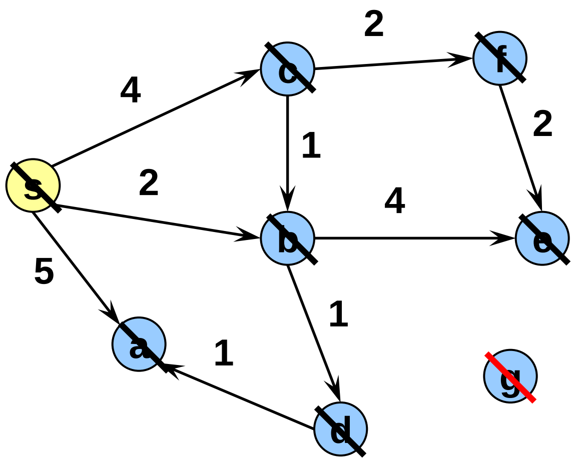
Dijkstra’s algorithm



Processed
s (D = 0)
b (D = 2)
d (D = 3)
c (D = 4)
a (D = 4)
...

e	f	g
6	**6**	∞

Dijkstra's algorithm



Processed

s (D = 0)

b (D = 2)

d (D = 3)

c (D = 4)

a (D = 4)

e (D = 6)

f (D = 6)

g (D = ∞)

Single source, shortest distances

Dijkstra’s Algorithm is greedy

1. Optimization problem
   - Of the many feasible solutions, finds the *minimum* or *maximum* solution.
2. Can only proceed in stages
   - no direct solution available
3. Greedy-choice property:

   A locally optimal (greedy) choice will lead to a globally optimal solution.

   Here, the deleteMin step is the greedy choice
4. Optimal substructure:

   An optimal solution contains within it optimal solutions to subproblems

Features of Dijkstra’s Algorithm

- Each vertex is processed exactly once (when it becomes the top of the priority queue)
- Each edge is processed exactly once
- *Distances* may be revised *multiple times*: current values represent ‘best guess’ based on our observations so far
- Once a vertex is processed we are guaranteed to have found the shortest path to that vertex…. ***why?***

Performance (using a heap)

Initialization: O(n)

Visitation loop: n calls

- deleteMin(): O(log n)
- Each edge is considered only once during entire execution, for a total of m updates of the priority queue, each O(log n)

Overall cost: O((n+m) log n)

Aside

Heap is used unevenly: n delete-mins but m promote operations.

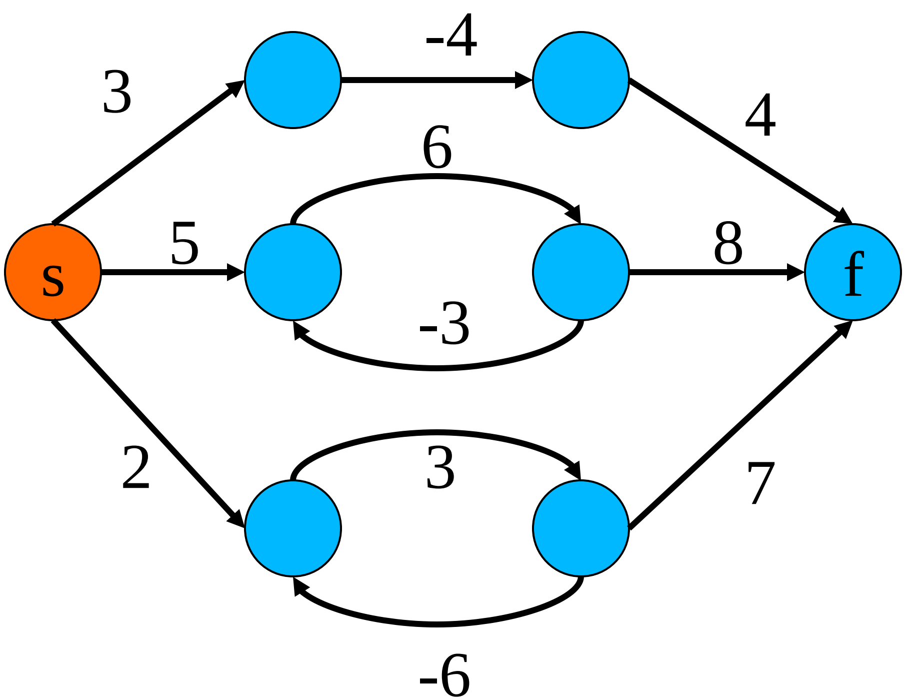
Can be exploited by using a better data structure (Fibonacci heap) to get running time $O(n \log n + m)$.

Representing shortest paths

- We now have an algorithms to compute the length of the shortest path between s and x.
- But what if we actually want to find the vertices on the shortest path?
- Fact: if $s=s_0,s_1,...,s_n=x$ is the shortest path from s to x, then $s=s_0, s_1,...,s_{n-1}$ is the shortest path from s to s_{n-1}.
- Idea: With each $dist(x)$, remember the previous node $prev(x)$ **=** s_{n-1} in the shortest path.

Thought Problem:

- What is the minimum cost distance between s and f?



Thought Problem:

- What do we do when there are negative edge weights?
- Other ideas and algorithms may be needed.