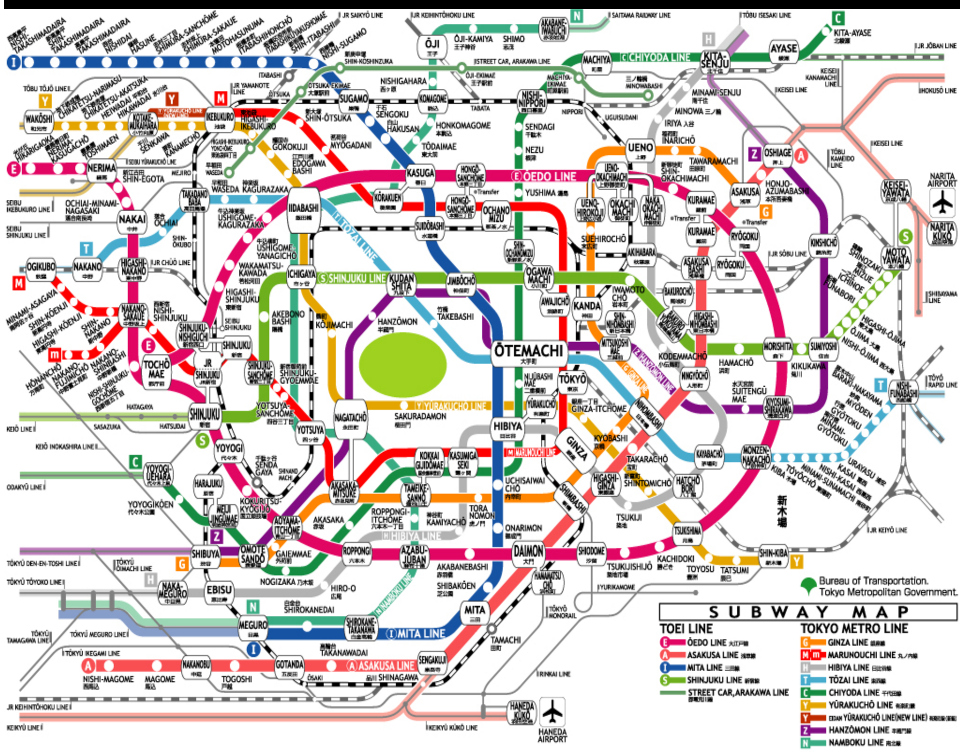
CS/ENGRD 2110
Object-Oriented Programming and Data Structures

Fall 2012

Doug James

Lecture 19: DFS, BFS & Shortest Paths



Today

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

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)
}
```

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
}
```

Shortest Paths in Graphs

- Finding the shortest (min-cost) path in a graph is a problem that occurs often
  - Best flight from Ithaca, NY to Duesseldorf, Germany?
  - How closely are two people connected on Facebook?
  - Driving directions from Ithaca, NY to Queens, NY?
  - AI path planning in robotics
  - Result depends on our notion of cost
    - Number of hops
    - Least mileage
    - Least time
    - Cheapest
    - Least boring
  - All of these "costs" can be represented as edge weights
- How do we find a shortest path?

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

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.

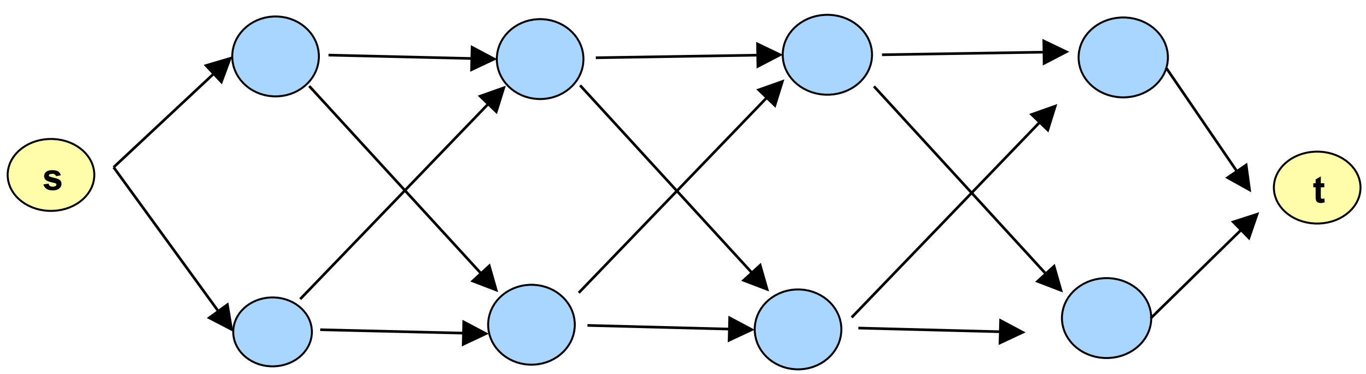
For each target vertex t, collect all simple paths with target t.

Compute their cost, determine the min.

Bad Idea

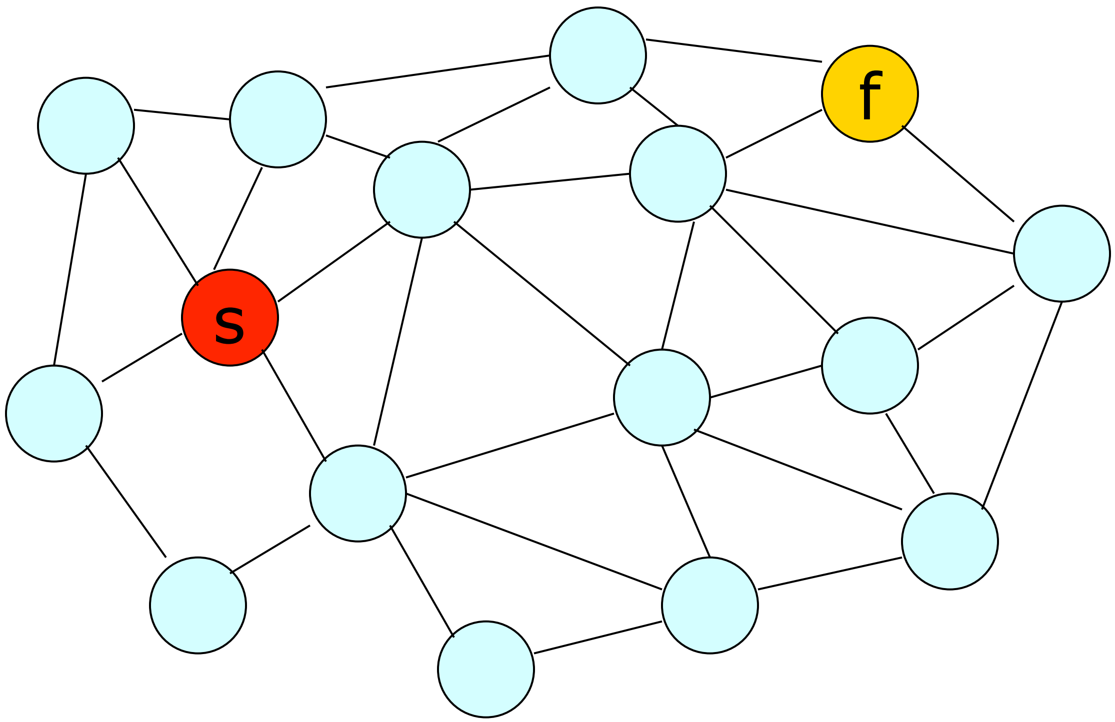
Even in an acyclic graph, the number of simple paths may be exponential in n.

Exercise: determine the number of paths s to t.



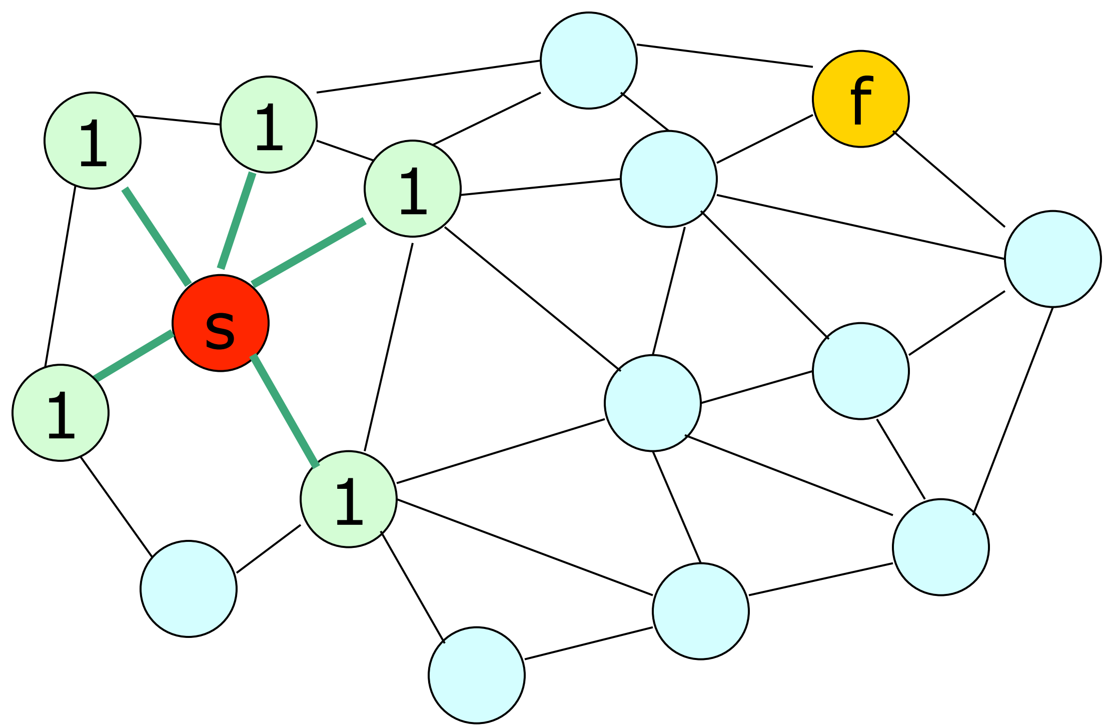
Single Source, Shortest Paths

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



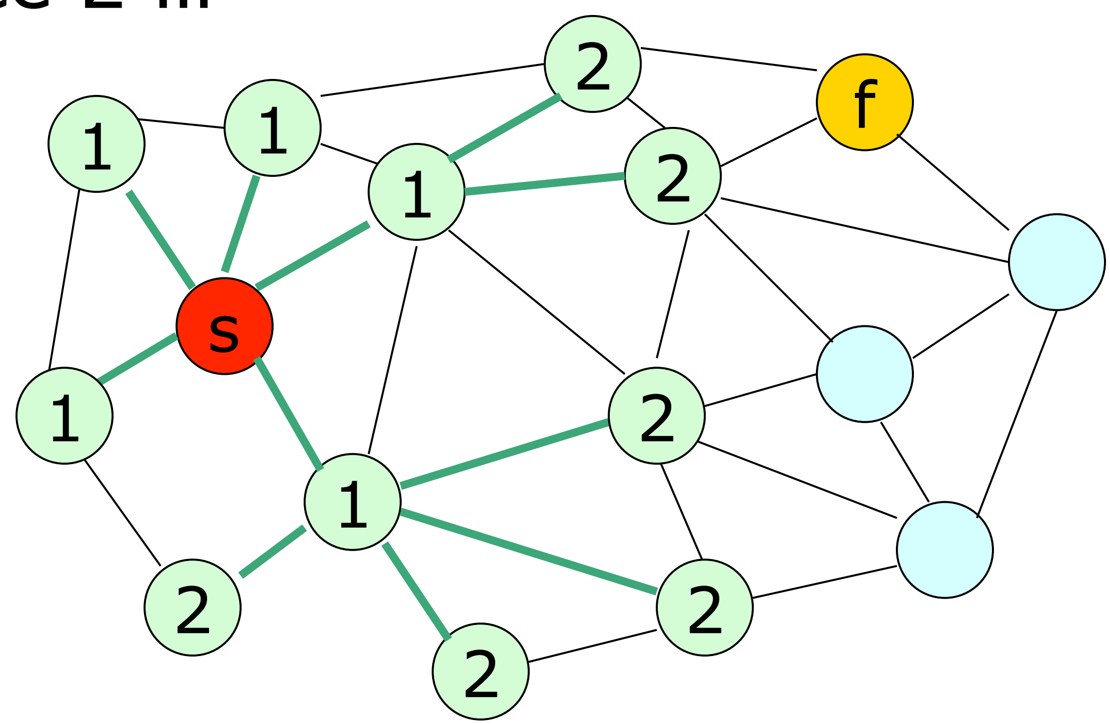
Single Source, Shortest Paths

- BFS
  - First, visit all nodes at distance 1



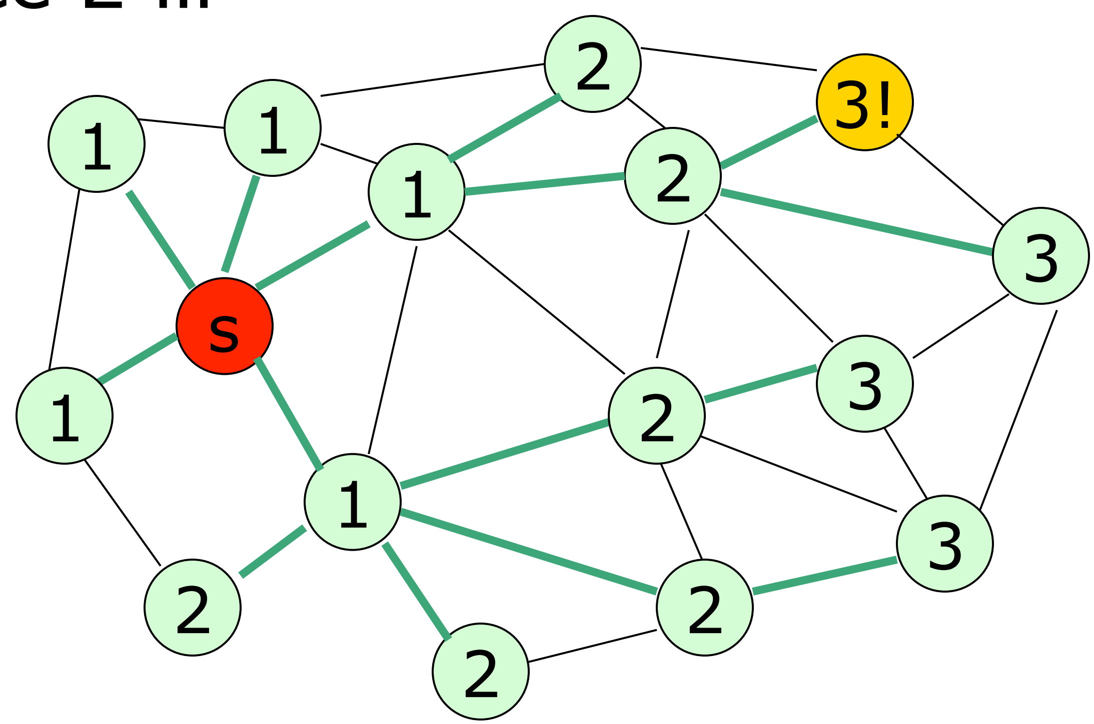
Single Source, Shortest Paths

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



Single Source, Shortest Paths

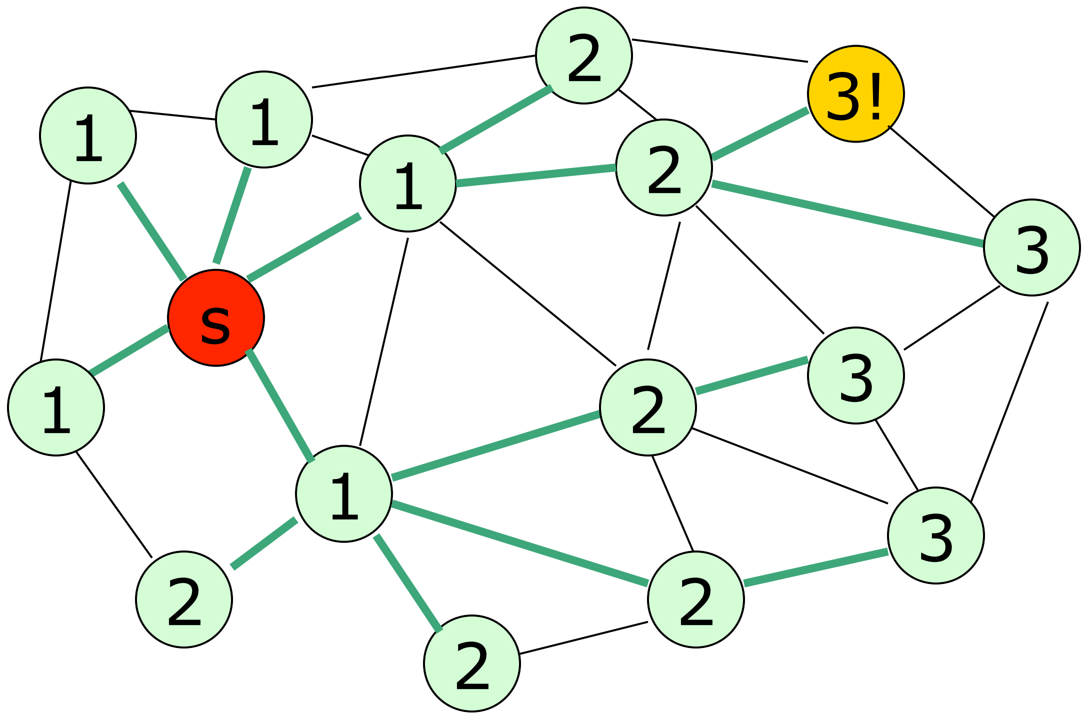
- BFS
  - First, visit all nodes at distance 1
  - Then, distance 2 …
  - Then, 3 …



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?

Breadth-First Search for Shortest Paths Unweighted Graphs

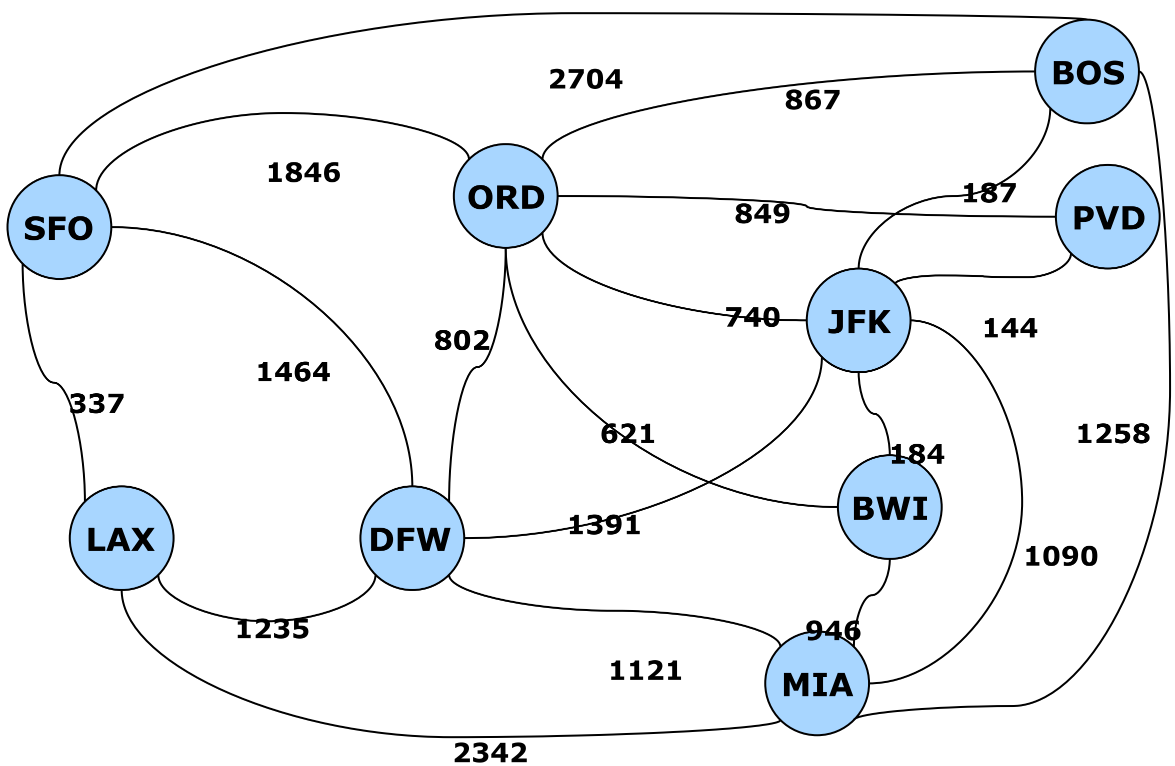
- Input: start node s, destination node t
- Put start s node into queue and mark s as visited.
- While queue not empty
  - Poll n off queue.
  - FOR all (unmarked) successors n' of n
    - IF n' equals t THEN return path
    - Put n' into queue
    - Mark n' as visited.
- Time complexity:
  - O(m) time

Why does BFS find Shortest Path?

- Any node in distance 1 is visited before any node at 2 hops, before any node at distance 3 hops, …
- Whenever a node is at the top of the queue for the first time, we must have gotten there with the minimum number of hops.
- How do we keep track of the path that got BFS there?
  - Store predecessor node on path for each node in graph.

Weighted Edges, Shortest Paths

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



Breadth-First Search for Shortest Paths Weighted Graphs

- Input: start node s, destination node t
- Put start (s,0,null) into min-priority queue.
- Initialize empty dictionary **`path`**.
- While queue not empty
  – Poll minimum element (n,c,prev) off queue.
  – Mark n as “done” in **`path`** by storing prev.
  – IF n equals t THEN return **`path`**
  – IF n is not yet “done”
    - FOR all successors n’ of n that are not “done”
      – Put (n’,c+weight(n,n’),n) into priority queue
- Time complexity:
  – O(m log m) time using heap and adjacency lists
  – Can be improved

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

$$\mathrm{dist}[y] > \mathrm{dist}[x] + \mathrm{cost}(x,y)$$

Prototype Algorithm

When an edge from x to y requires attention we relax it, updating the estimate for dist[y]:

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);
```

Dijkstra's Algorithm

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

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

Dijkstra's Algorithm

```
initialize dist[];
insert all v in V into PQ; // prioritized by 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; // prioritized by dist

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

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

            // Optional: record best path to y was from x
```

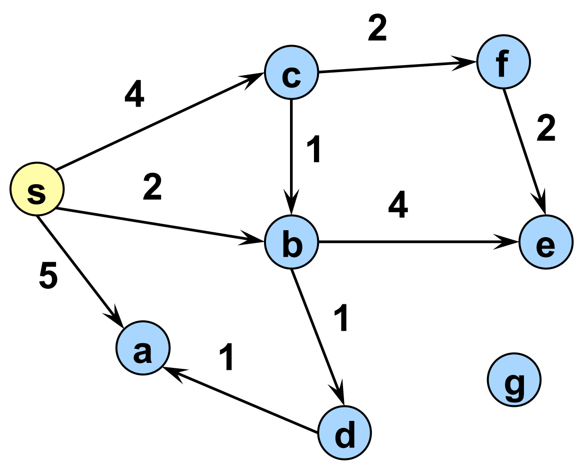
Dijkstra's algorithm

Initialization

a. Set $dist(s) = 0$

b. For all vertices $v \in V$, $v \neq s$, set $dist(v) = \infty$

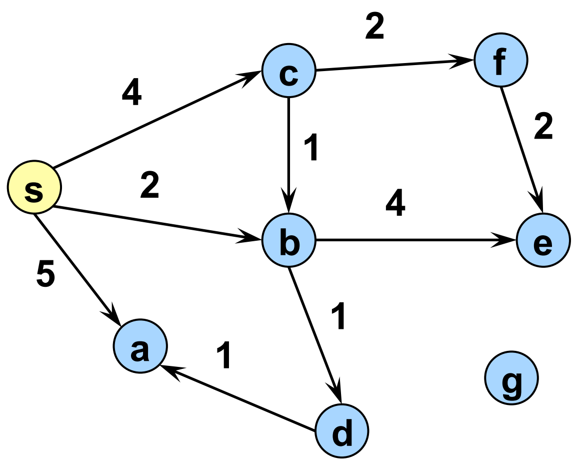
c. Insert all vertices into priority queue P, using distances as the keys



s	a	b	c	d	e	f	g
0	∞	∞	∞	∞	∞	∞	∞

P

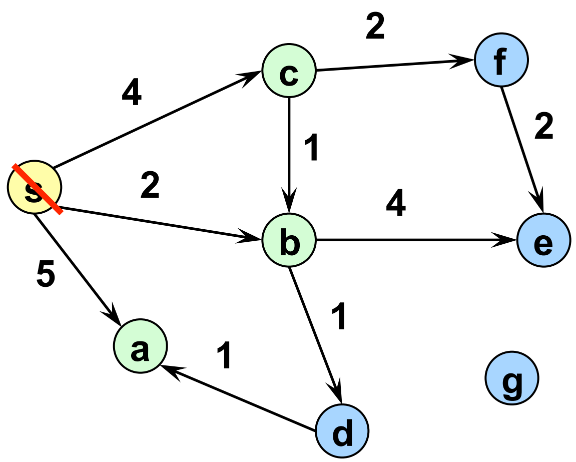
Dijkstra's algorithm



Processed

s	a	b	c	d	e	f	g
0	∞	∞	∞	∞	∞	∞	∞

Dijkstra’s algorithm

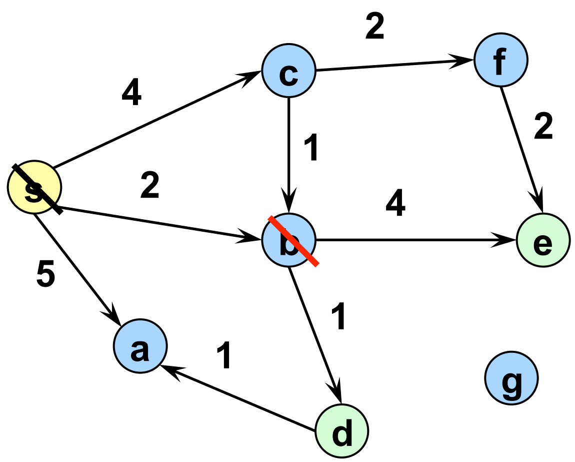


Processed

s (D = 0)

b	c	a	d	e	f	g
2	4	5	∞	∞	∞	∞

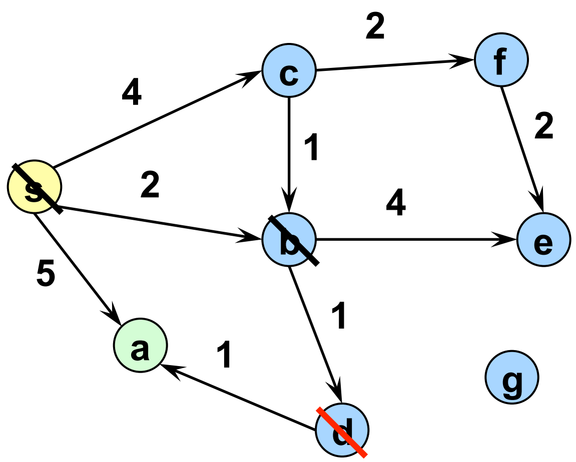
Dijkstra's algorithm



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

d	c	a	e	f	g
3	4	5	6	∞	∞

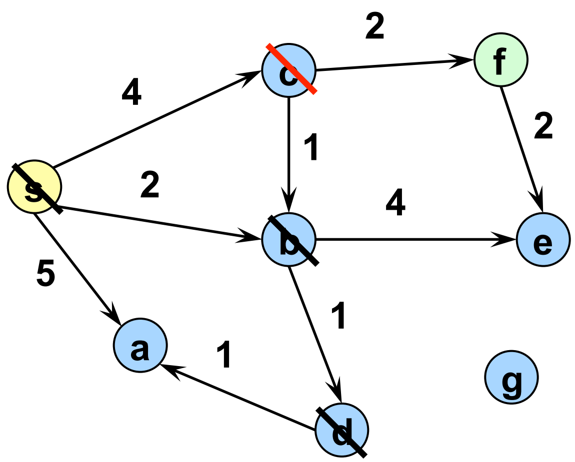
Dijkstra's algorithm



Processed
s (D = 0)
b (D = 2)
d (D = 3)

c	a	e	f	g
4	4	6	∞	∞

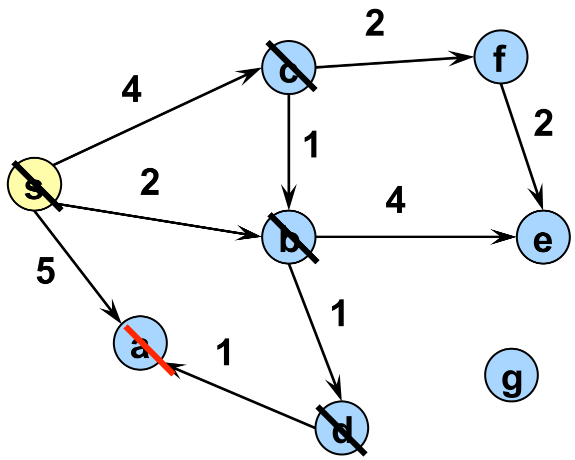
Dijkstra’s algorithm



<u>Processed</u>
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)

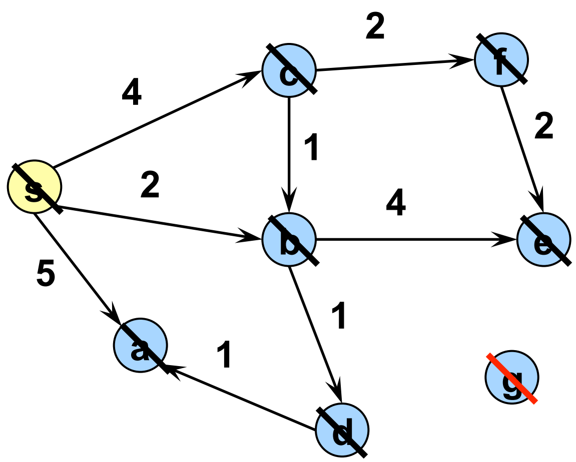
c (D = 4)

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 optimal (*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.

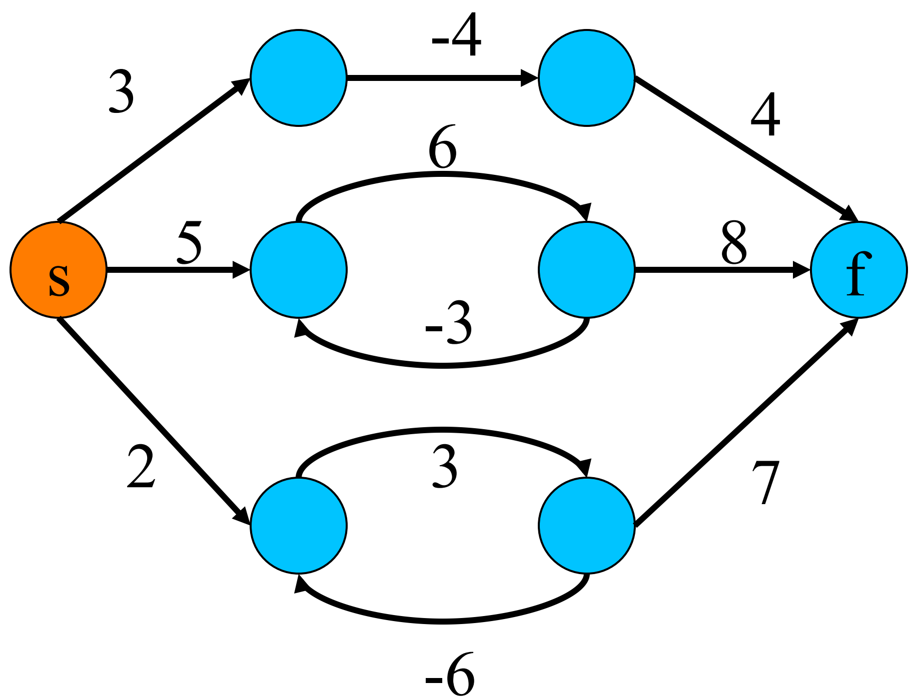
Can be exploited by using a better data structure (Fibonacci heap) to get running time $O(n \log n + m)$.

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: Negative Weights

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



Thought Problem: Negative Weights

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