Dijkstra‘s Algorithm and Priority Queue Implementations

CSE 101: Design and Analysis of Algorithms

Lecture 5

CSE 101: Design and analysis of algorithms

- Dijkstra's algorithm and priority queue implementations
  - Reading: Sections 4.4 and 4.5
- Homework 2 is due Oct 16, 11:59 PM
- Quiz 1 is Oct 18

Depth-first search is not good for

- Finding shortest distances between vertices

Distance

- Definition: In a graph G with at least two vertices u and v, the **distance** from u to v is the length of the shortest path from u to v
- If there is no path from u to v then we say that the distance is infinite
- The distance from a vertex to itself is 0

Graph reachability

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
 Pick v in F. stack, queue, or array of booleans
 For each neighbor u of v:
  If u is not in X or F:
   move u from U to F.
 Move v from F to X.

Return X.

X: explored
F: frontier (reached but have not yet explored)
U: unreached

Differences

- F is stack, last-found is first explored
  - Depth-first search
- F is queue, first found is first explored
  - Breadth-first search
- F is priority queue based on total length
  - Dijkstra's algorithm

Stack versus queue

- The queue gives us some extra information
  - The queue starts with just the node s, the only one that has distance 0
  - Then, you put each neighbor of s into the queue and now the queue has all the nodes at distance 1 from s
  - Then, you put each neighbor of those into the queue one after the other until you have all the nodes at distance 2 from s
  - In fact, for each distance d from s, the queue contains all the nodes at distance d from s and nothing else. As these are ejected, their undiscovered neighbors are the next nodes added to the queue.

Breadth-first search

- Given a graph G and a starting vertex s, breadth-first search computes distances from s to every other node
  - It keeps this information in an array dist
- To do this, breadth-first search computes distances "layer by layer"
- It sets dist(s)=0 and the next layer is all the vertices adjacent to s. If v is adjacent to s then it sets dist(v)=1.
- For each vertex v, breadth-first search will set dist(v) = dist(prev(v))+1

Breadth-first search

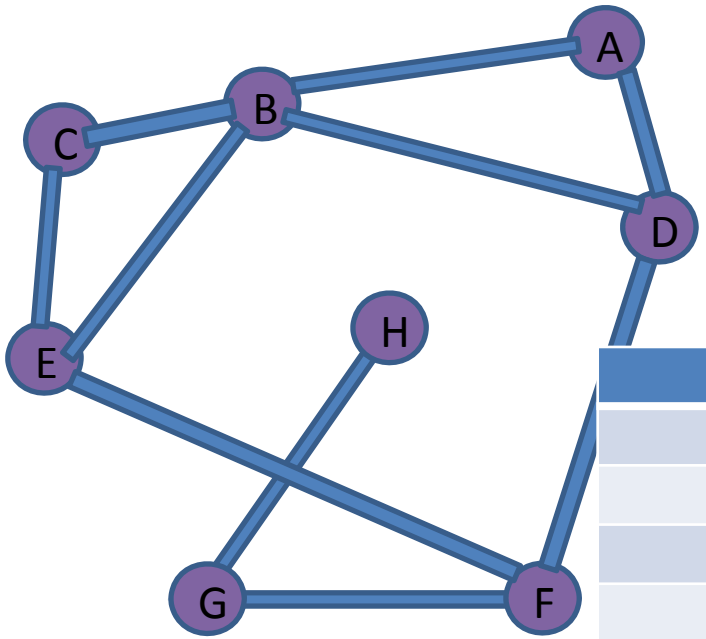
procedure BFS(G, s)

Input: (Directed or undirected) graph G = (V,E), and a vertex s in V.

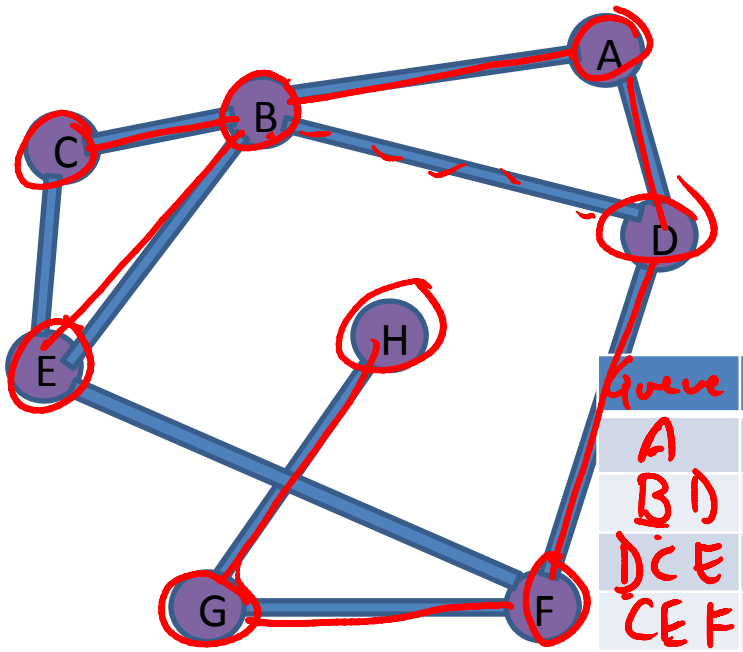
Output: For all vertices u reachable from s, dist(u) is the distance from s to u and for all vertices u not reachable from s, dist(u) = ∞

```
for each vertex u in V:
  dist(u)=∞
dist(s) = 0
Q = [s] (queue just containing s)
while Q is not empty
  u = eject(Q) dequeue
  for all edges (u,v) in E
    if dist(v)=∞ then
      inject(Q,v) enqueue
      dist(v)=dist(u) + 1
```

Example



Example



Queue	A	B	C	D	E	F	G	H
A	0	∞	∞	∞	∞	∞	∞	∞
B D		1	∞	1	∞	∞	∞	∞
D C E			2	1	2	∞	∞	∞
C E F			2		2	2	∞	∞
E F			2		2	2	∞	∞
F			2		2	2	∞	∞
G H							3	3 4

Correctness of breadth-first search

- We want to show that BFS assigns dist() correctly to all vertices reachable from s
- Proof by induction on distance
  - Base case: dist(s) = 0
- Inductive hypothesis
  - For any vertex v that is distance k from s, dist(v) = k
- Inductive step
  - Suppose u is distance k + 1
  - (Want to show: breadth-first search assigns dist(u) = k + 1)

Correctness of breadth-first search

- Inductive step: suppose u is distance k + 1 from s
  - dist(u) starts at infinity
  - There exists a vertex w such that w is distance k from s and (w,u) is in E
  - In the last for loop when w is ejected, we encounter the edge (w,u) and dist(u) is infinity so we inject u into Q and set dist(u) = dist(w) + 1 = k + 1

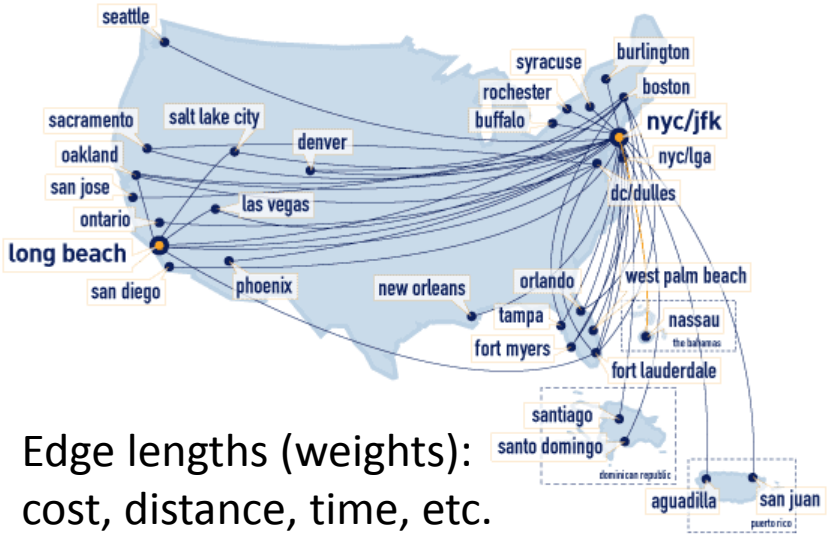
Depth-first search vs breadth-first search

- Depth-first search gives information about the whole graph
- Breadth-first search gives information related to a given vertex within the graph

- Depth-first search uses a stack
- Breadth-first search uses a queue

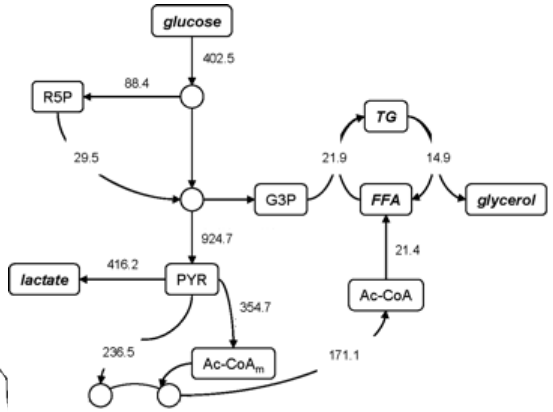
- Breadth-first search does not restart at other connected components since all vertices not connected to your starting vertex are distance infinity away

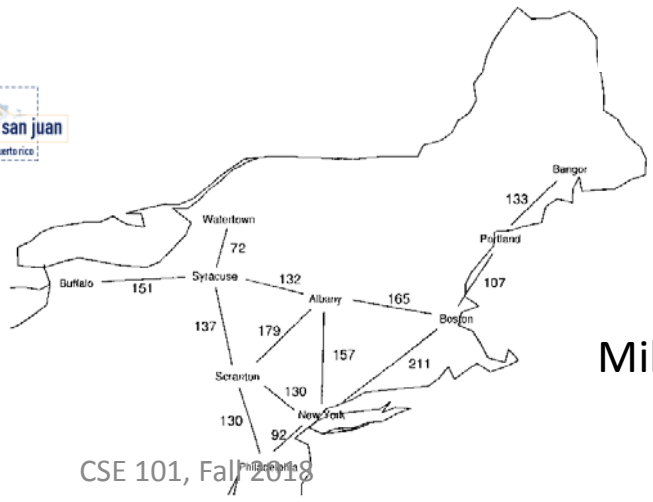
Graphs with edge lengths



Edge lengths (weights):
cost, distance, time, etc.

Energy





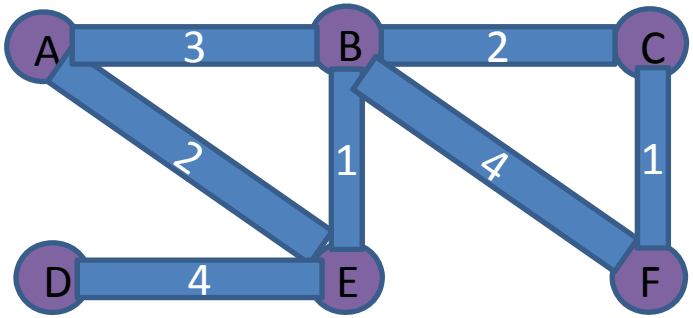
Miles

Edge lengths (weights)

- Edges can be given values such as
  - distance
  - cost
  - time
- We will denote the length (weight) of edge $e = (u, v)$ as $\ell_e, \ell_{(u,v)}, \ell(e), \ell(u, v)$

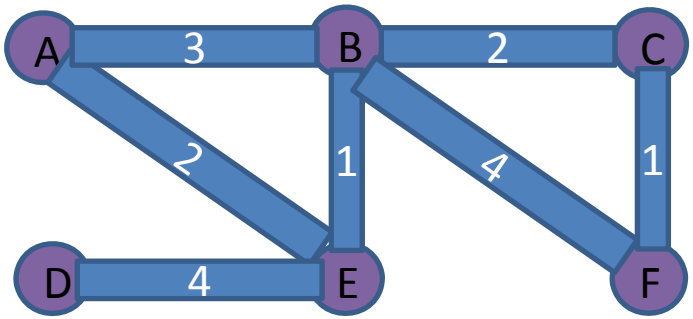
Breadth-first search on weighted graphs

- Breadth-first search only works to find shortest distances on graphs in which each edge has equal weight

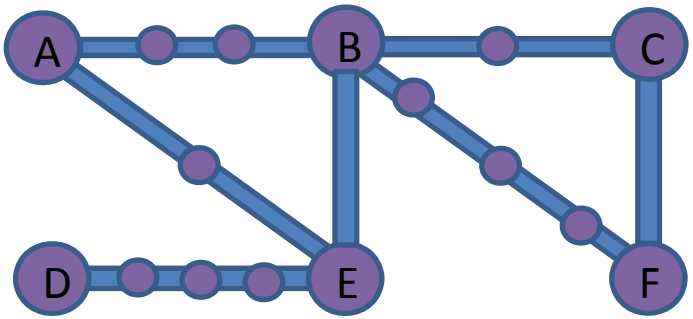


Breadth-first search on weighted graphs

- On a graph G with integer edge lengths, form G' by adding $\ell_e - 1$ many new vertices between u and v for every edge $e = (u, v)$. Then, run breadth-first search on G'.



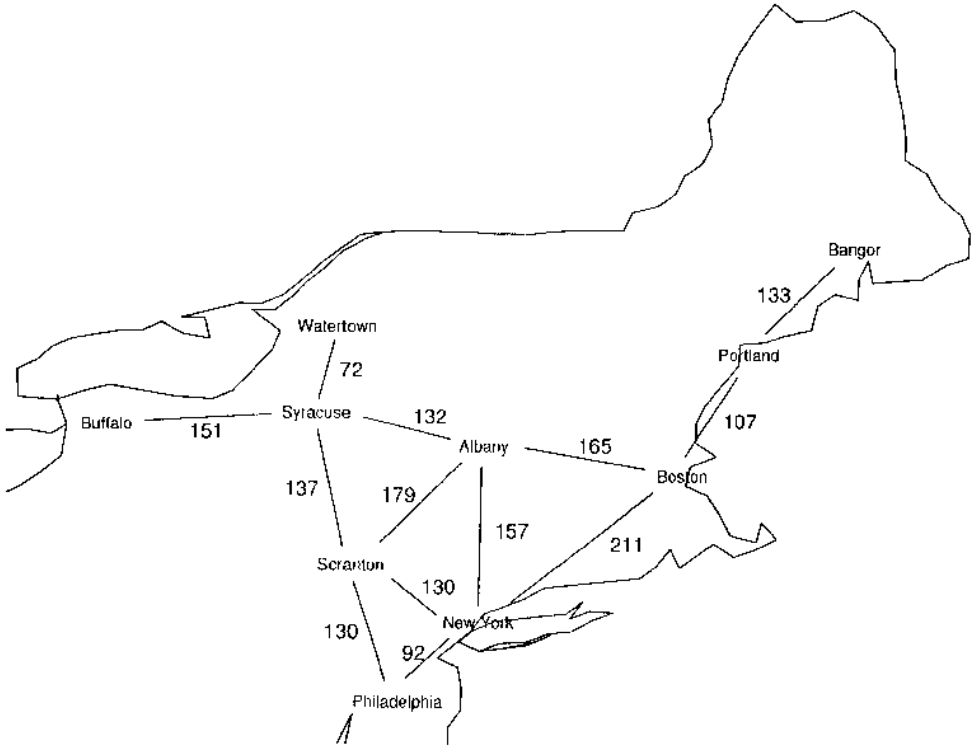
G



G'

Problems with this method

- If the edge lengths (weights) are large integers then it is impractical
- In this example with 10 vertices, we must add 1,783 more vertices!



High-level to low-level algorithm design

- While our end goal is to develop fully specified algorithms, with all of the details including data structures, and a complete time analysis, it is useful to first come up with a **high-level** version of the algorithm
  - A high-level version specifies **what** the algorithm wants to do at every step, in terms of abstractions such as sets, relations, orderings, graphs, and so on, but not all of the details of **how** it will do these steps
- We then need to fill in those details. Usually we make a **low-level or implementation-level algorithm** from a high-level one by matching abstract mathematical structures to **data structures**.

Advantages of high-level design

- **Clarity**. Presenting a high-level algorithm first gives the reader (and you) the main idea of what is going on, before getting caught up in details
- **Correctness proofs** are usually much easier to do for high-level algorithms. Then, we just need to make sure that the low-level version does what the high-level version specifies.
- **Flexibility**. By showing that the high-level algorithm works, we show that **any** low-level implementation will solve the problem (given that the data structures and implementation details do what they claim). Then, we can change the details of the implementation to fit a given situation (e.g., dense vs. sparse graphs, memory efficiency vs. time efficiency, parallel vs. sequential) without worries.

Case study: Dijkstra's algorithm

- We will use this as a test case for high-level algorithm design. We will present an abstract version of Dijkstra's algorithm, prove correctness at the abstract level, and then discuss a few ways of implementing it for different situations.

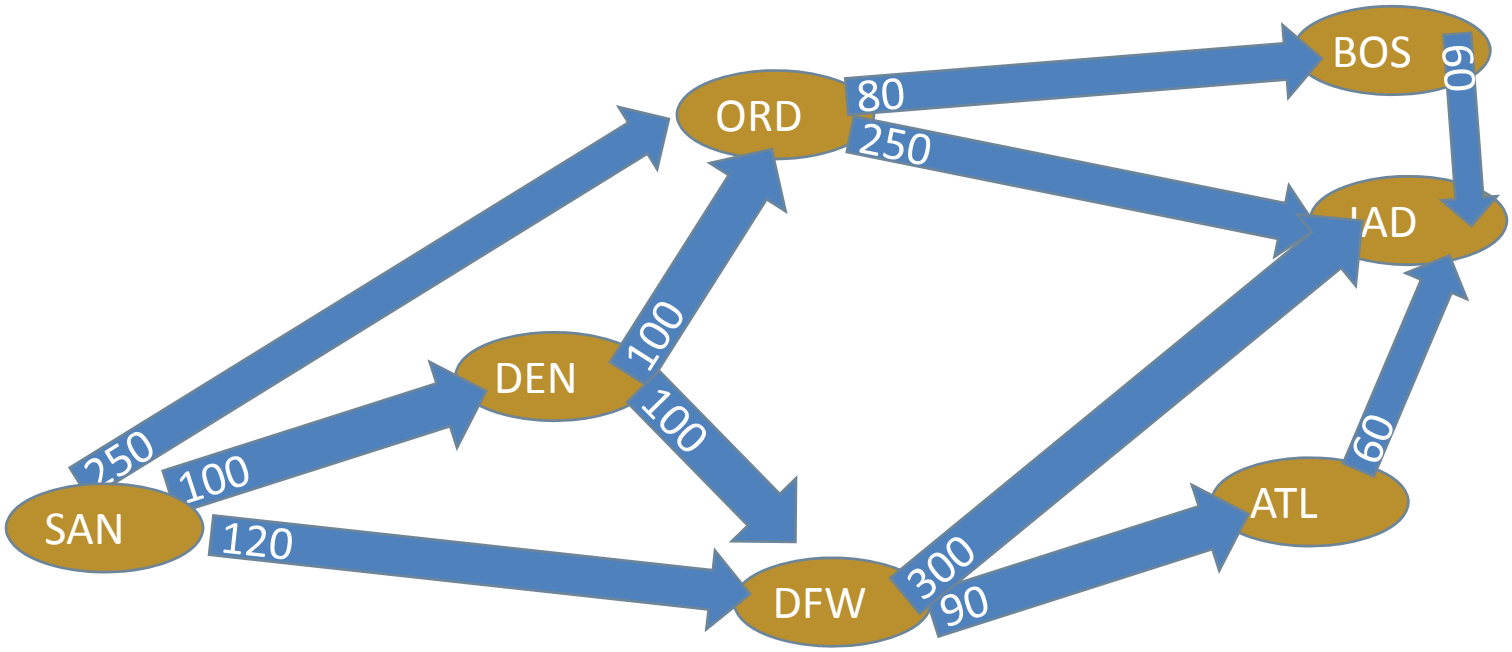
Single source shortest paths problem

- Instance: a directed graph G = (V,E), a set of edge lengths ℓ = {ℓ(e):e is in E}, and the starting vertex s
- Output: for all vertices u reachable from s, dist(u) is set to the distance from s to u (i.e., the minimum, over all paths from s to u, of the sum of the edge lengths). For all vertices u unreachable from s, dist(u) is set to infinity.

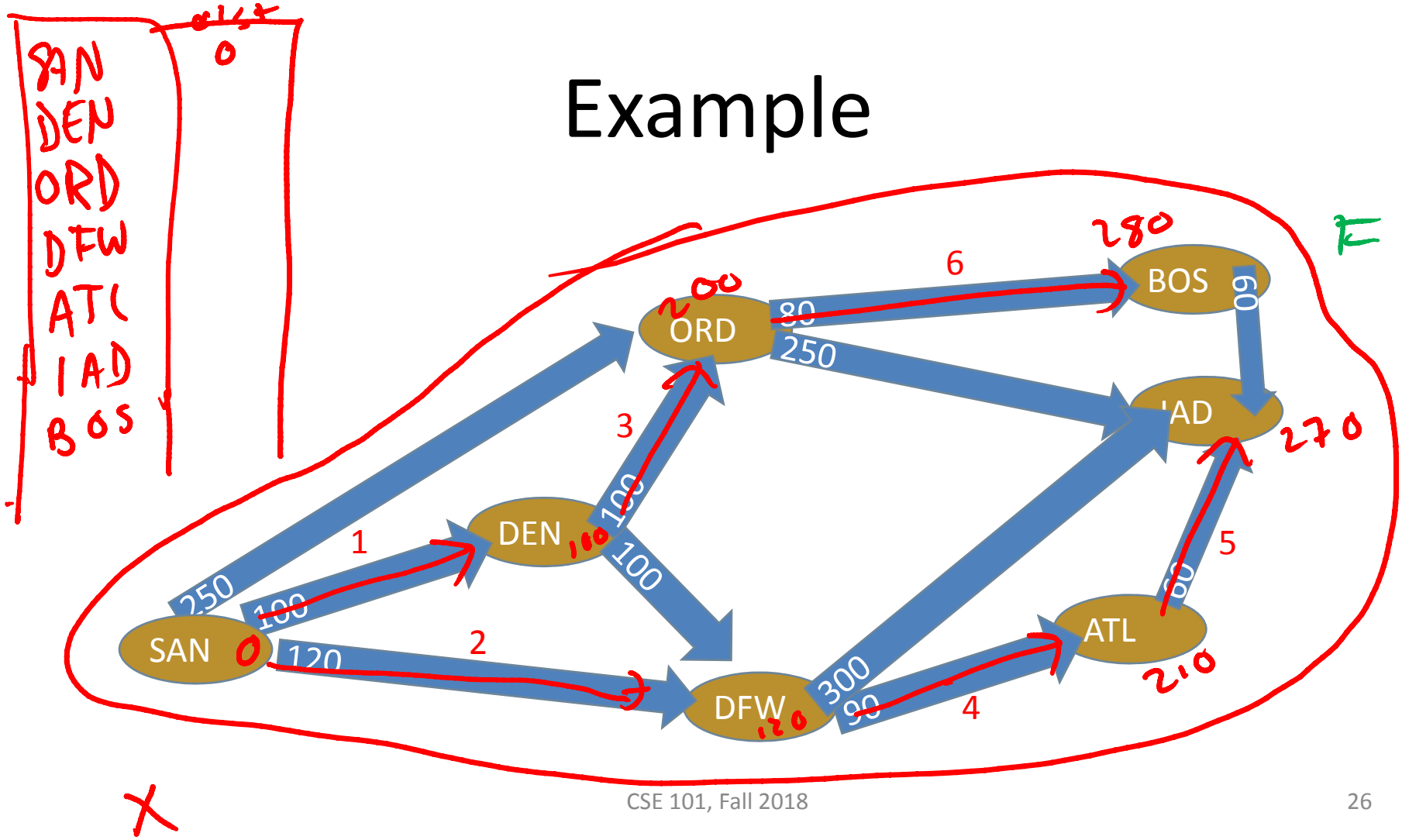
Dijkstra's high-level algorithm

- Let X and F be sets
- All vertices start out in F with dist set to ∞
- Set dist(s)=0 and move s from F to X
- Repeat until F is empty or there are no edges from X to F:
  - Let w be the vertex in F with the minimum value:
    $$dist(v) + \ell(v, w) \text{ for all } v \text{ in } X$$
  - Set $dist(w) = dist(v) + \ell(v, w)$
  - Move w from F to X

Example


BOS
ORD
80
60
250
IAD
DEN
100
100
250
100
SAN
120
DFW
300
90
ATL
60

Example


SAN
DEN
ORD
DFW
ATL
IAD
BOS
SAN
DEN
ORD
DFW
ATL
IAD
BOS
250
100
120
1
2
3
100
100
250
6
80
60
90
5
4
300

Correctness of Dijkstra's algorithm

- Let **truedist**(v) be the actual shortest distance from s to v
- We want to show that at the end of the algorithm dist(v) = **truedist**(v) for all vertices v
- In the middle of the algorithm, this is not necessarily true for all vertices. But, once a vertex moves to X, its dist is locked in.
- We will prove that dist(v) = **truedist**(v) for all v in X

- We will prove this by induction on k, the number of elements in X
- (Note: only one element is put into X at a time)

Correctness of Dijkstra's algorithm

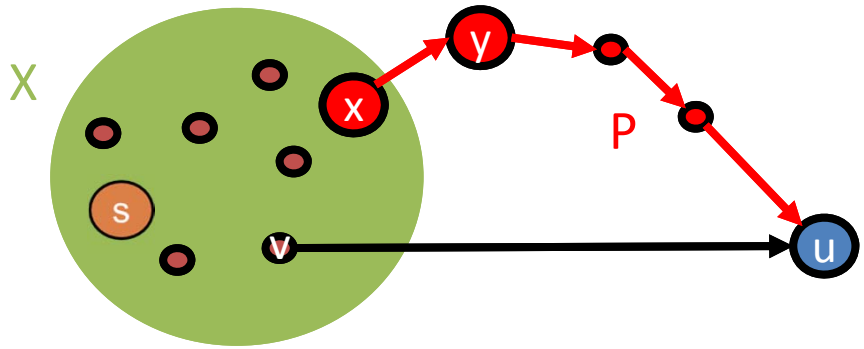
- Base case: the starting vertex is put into X. X={s}, dist(s) = 0. Since **truedist**(s) = 0, this is correct.
- Inductive hypothesis: for some k ≥ 1, assume that when |X|=k, for all vertices v in X, dist(v) = **truedist**(v)
- (What to show: that after the $(k+1)^{st}$ vertex is added to X, that vertex is assigned its correct distance value)

Correctness of Dijkstra's algorithm

- Inductive step: after k ejections, let u be the next vertex to be ejected with v its neighbor in X: dist(v)+ℓ(v,u) is the minimum of all edges leaving X
  - Then, we set the dist(u) = dist(v)+ℓ(v,u). By the induction hypothesis, dist(v)=truedist(v) so there is a path from s to v of distance dist(v) and hence a path from s to u of distance dist(u), therefore truedist(u)$\leq$ dist(u)
  - Let's assume by contradiction that dist(u) $\neq$ truedist(u), i.e., truedist(u)<dist(u)

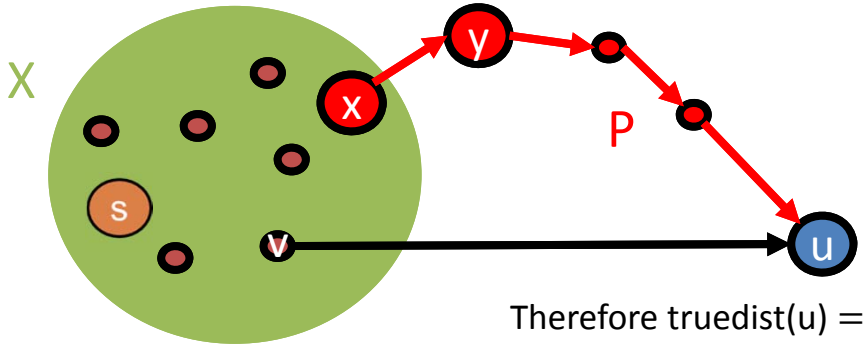
Correctness of Dijkstra's algorithm

- Inductive step: after k ejections, let u be the next vertex to be ejected with v its neighbor in X: dist(v)+ℓ(v,u) is the minimum of all edges leaving X
- Let's call the true shortest path P



Correctness of Dijkstra's algorithm

- Inductive step: after k ejections, let u be the next vertex to be ejected with v its neighbor in X: dist(v)+ℓ(v,u) is the minimum of all edges leaving X
- Let's call the true shortest path P



There must be some edge (x,y) in P with x$\in X$ and $y \notin X$

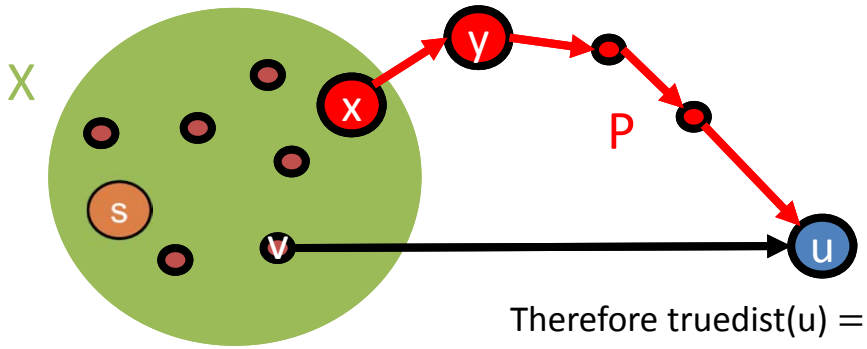
dist(x) + ℓ(x,y) $\geq$ dist(v) + ℓ(v,u) = dist(u) (by choice of u)

dist(x) = truedist(x) (by IH)

Therefore truedist(u) = len(P) $\geq$ truedist(x) + $\ell(x, y) \geq$ dist(u) > truedist(u)

Contradiction.

Where did we use non-negativity?



There must be some edge (x,y) in P with x∈ X and $y \notin X$

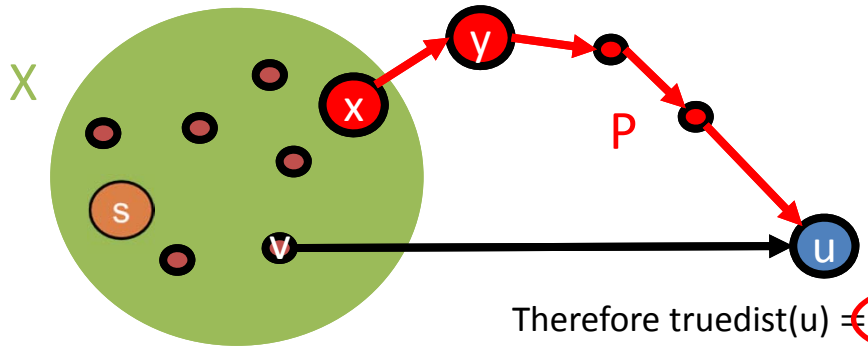
dist(x) + ℓ(x,y) ≥ dist(v) + ℓ(v,u) = dist(u) (by choice of u)

dist(x) = truedist(x) (by IH)

Therefore truedist(u) = len(P) ≥ truedist(x) + $\ell(x, y)$ ≥ dist(u) > truedist(u)

Contradiction.

Where did we use non-negativity?



There must be some edge (x,y) in P with x∈ X and $y \notin X$

dist(x) + ℓ(x,y) ≥ dist(v) + ℓ(v,u) = dist(u) (by choice of u)

dist(x) = truedist(x) (by IH)

Therefore truedist(u) = len(P) ≥ truedist(x) + $\ell(x, y)$ ≥ dist(u) > truedist(u)

Contradiction.

Dijkstra's algorithm, towards low-level

procedure dijkstra(G,ℓ,s)
X: set of vertices we know distances to
F: set of vertices we aren't sure of the distance to

- Let X and F be sets
- All vertices start out in F with dist set to ∞
- Set dist(s)=0 and move s from F to X
- Repeat until F is empty or there are no edges from X to F:
  - Let w be the vertex in F with the minimum value:
    $$dist(v) + \ell(v, w) \text{ for all } v \text{ in } X$$
  - Set $dist(w) = dist(v) + \ell(v, w)$
    (Note: only need to keep track of best current edge to each vertex in F, so only one number per node in F. Can use same array, dist to keep track of this number)
  - Move w from F to X

Dijkstra's algorithm, towards low-level

procedure dijkstra(G,ℓ,s)
X: set of vertices we know distances to
F: set of vertices we aren't sure of the distance to

- Let X and F be sets
- All vertices start out in F with dist set to ∞
- Set dist(s)=0 and move s from F to X

  We will maintain the invariant that for all u ∈ F, dist(u) is the minimum over all e = (v,u), v ∈ X of dist(v)+l(e)
- Repeat until F is empty or there are no edges from X to F:
  - Let w be the vertex in F with the minimum value:

    $dist(v) + \ell(v, w)$ for all v in X
  - Set $dist(w) = dist(v) + \ell(v, w)$
  - Move w from F to X

Structures in Dijkstra's algorithm

- Graph G: How does it change? Where do we access it?
- Set X: How does it change? Where do we access it?
- Set F: How does it change? Where do we access it?

Structures in Dijkstra's algorithm

- Graph G: no changes, need to list members
  - Adjacency list
- Set X: insert, check membership
  - Array of booleans
- Set F: Find and delete the element with minimum key, dist(u). Decrease the keys of some elements u'.

Priority queue

- A priority queue is a data structure of a set of objects (vertices) along with key values for each object that can be changed (alarm settings). Additionally, it can support the following operations.
  - insert
  - deletemin
  - decreasekey
  - makequeue

Array as a priority queue

- Array: indexed by vertex, giving key value
  [key(A),key(B),key(C),key(D),key(E),key(F)]

- makequeue
- deletemin
- decreasekey

Array as a priority queue

- Dijkstra's algorithm takes time

  makeque + deletemin * |V| + decreasekey * |E|

  – If we use an array, then it will take

    $|V| + O(|V|)\ |V| + O(1)\ |E| = O(|V|^2 + |E|) = O(|V|^2)$

Next lecture

- Priority queue implementations and minimum spanning trees
  - Reading: Sections 4.5 and 5.1