CSCI 136
Data Structures & Advanced Programming

Lecture 32

Fall 2019

Instructors: B&S

Last Time

- Fundamental Graph Algorithms
  - Find Connected Components
  - Find minimum length paths (edge count)
  - Find minimum length paths (edge weights)
    - Dijkstra's Algorithm: What to compute

Today’s Outline

- Dijkstra’s Algorithm
  - How to compute it
  - Correctness and Complexity
- Minimum-cost spanning subgraph: Prim

Single Source Shortest Paths

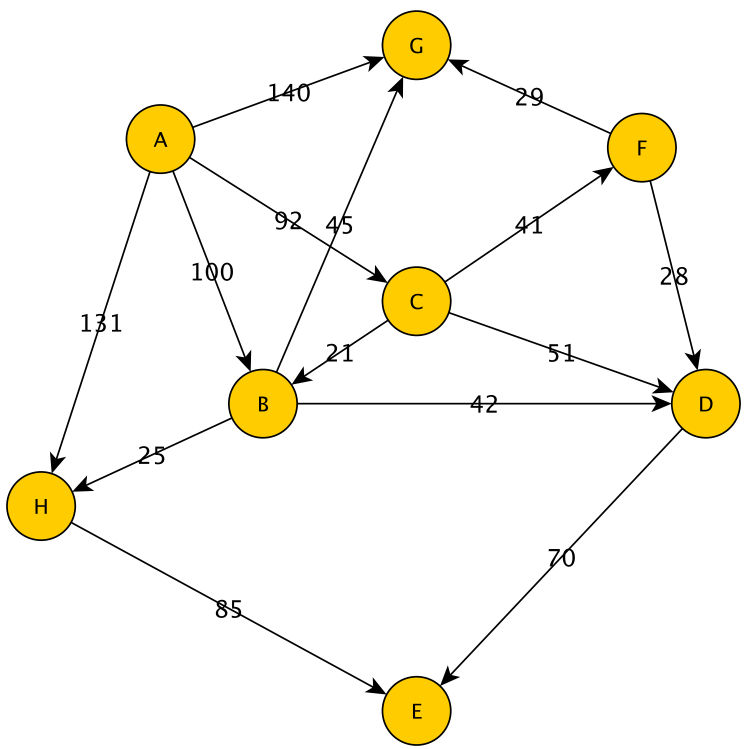
Theorem (from previous lecture)

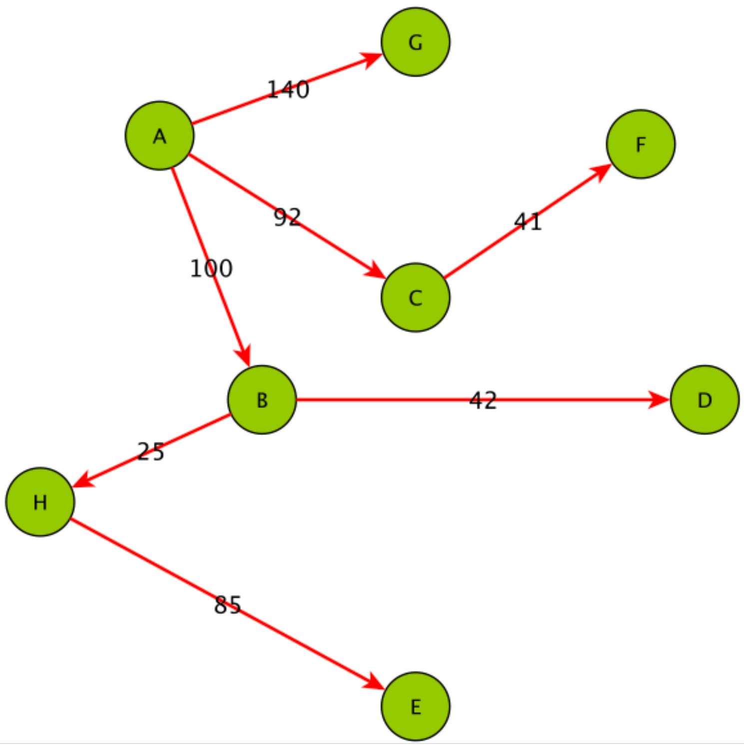
Let G=(V,E) be a directed graphs with non-negative edge weight function $w: E \rightarrow \mathbb{R}^+ \cup 0$.

Then for any vertex v, G contains a subgraph T_v of G such that T_v is a tree consisting minimum-weight paths from v to every other vertex of G.

Dijkstra's Algorithm: Efficiently construct such a tree T_v for each vertex v in G

Dijkstra Shortest Paths Tree



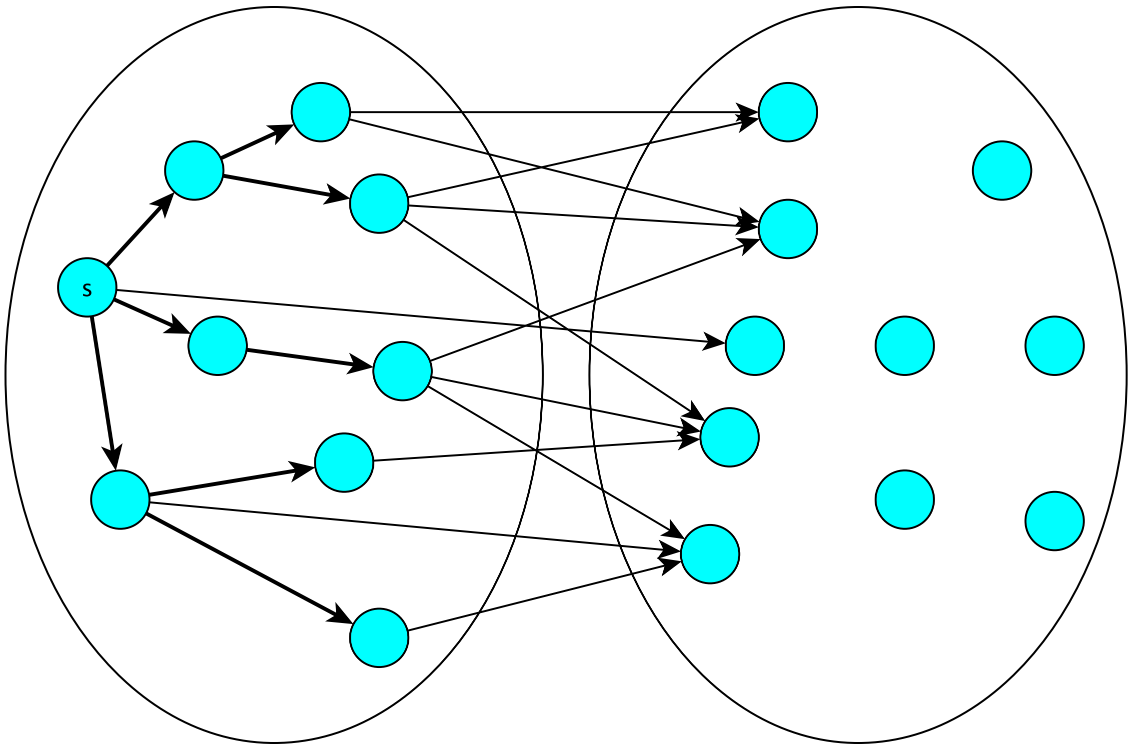


The Tree of Shortest Paths Found by Dijkstra's Algorithm

The Right Kind of Greed

- A start: take shortest edge from start vertex s
  - That must be a shortest path!
  - And now we have a small tree of shortest paths
- What next?
  - Design an algorithm by thinking inductively
  - Suppose we have found a tree T_k that has shortest paths from s to the k-1 vertices "closest" to s
  - What vertex would we want to add next?

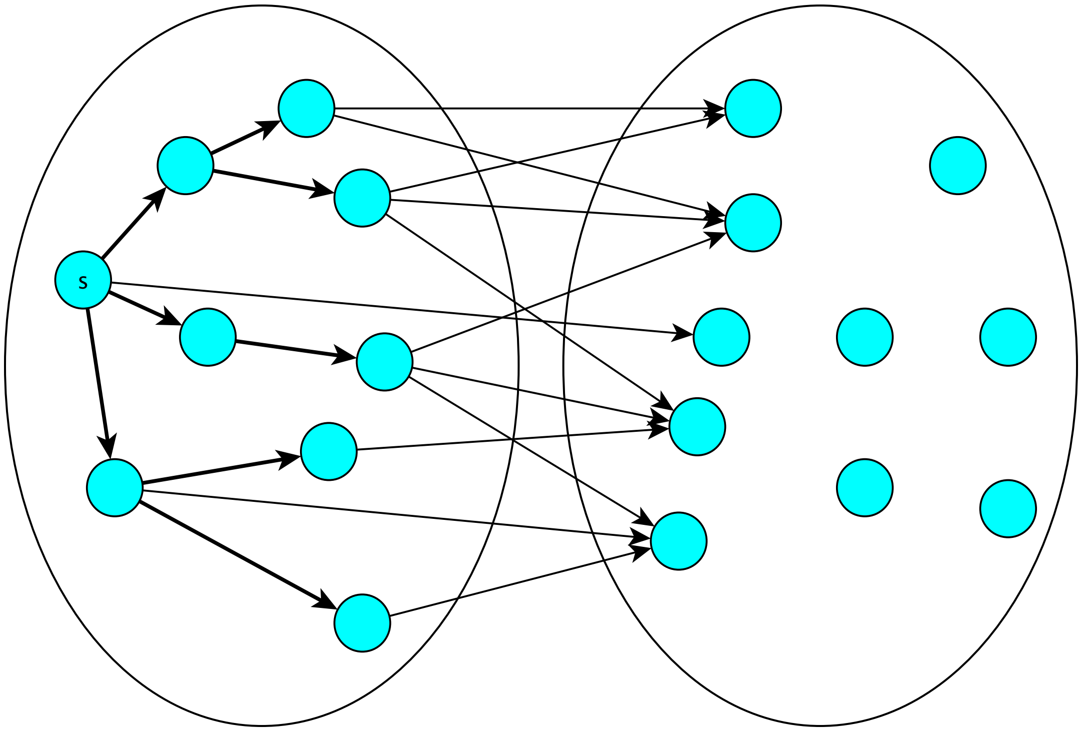
Finding the Best Vertex to Add to T_k



Not all edges are displayed

Question: Can we find the next closest vertex to s?

What's a Good Greedy Choice?



Idea: Pick edge e from u in T_k to v in G-T_k that minimizes the length of the tree path from s up to–and through–e

Now add v and e to T_k to get tree T_{k+1}

Now T_{k+1} is a tree consisting of shortest paths from s to the k vertices closest to s! Repeat until k = |V|

Some Notation

- l(e) : length (weight) of edge e
- d(u,v) : *distance* from u to v
    - Weight of minimum-weight path from u to v
    - That is, length of minimum-length path….

- Note: d(,) defines a valid distance measure. That is
    - d(u,u) = 0 for every vertex u
    - d(u,v) = d(v,u) for every pair of vertices
    - d(u,v) ≤ d(u,w) + d(w,v) for every triple of vertices

- So we'll now use phrases like minimum-length and closest in our discussion

Dijkstra's Insight

Theorem

Let G=(V,E) be a directed graphs with non-negative edge length function $l: E \rightarrow \mathbb{R}^{+} \cup 0$

Let s be a vertex of G and let T_k be a tree of shortest paths from so to the k *closest* vertices to s (including s).

Let u be a vertex u in G - T_k that minimizes d(s,v) + l(v,u) over all edges (u,v) for which v is in T_k and u is in G - T_k

Then, the tree $T_{k+1} = T_k \cup (v,u)$ consists of shortest paths from s to the k+1 closest vertices to s

Let's prove the induction step….

Dijkstra's Algorithm

Dijkstra(G, s) // l(e) is the length of edge e

let T ← *({s}, ∅) and PQ be an empty priority queue*

for each neighbor v of s, add edge (s,v) to PQ with priority l(e)

while T doesn't have all vertices of G and PQ is non-empty

    repeat

        e ← *PQ.removeMin()   // skip edges with both*

    until PQ is empty or e=(u,v) for u∈*T, v* ∉ *T // ends in T*

    if e=(u,v) for u∈*T, v* ∉ *T*

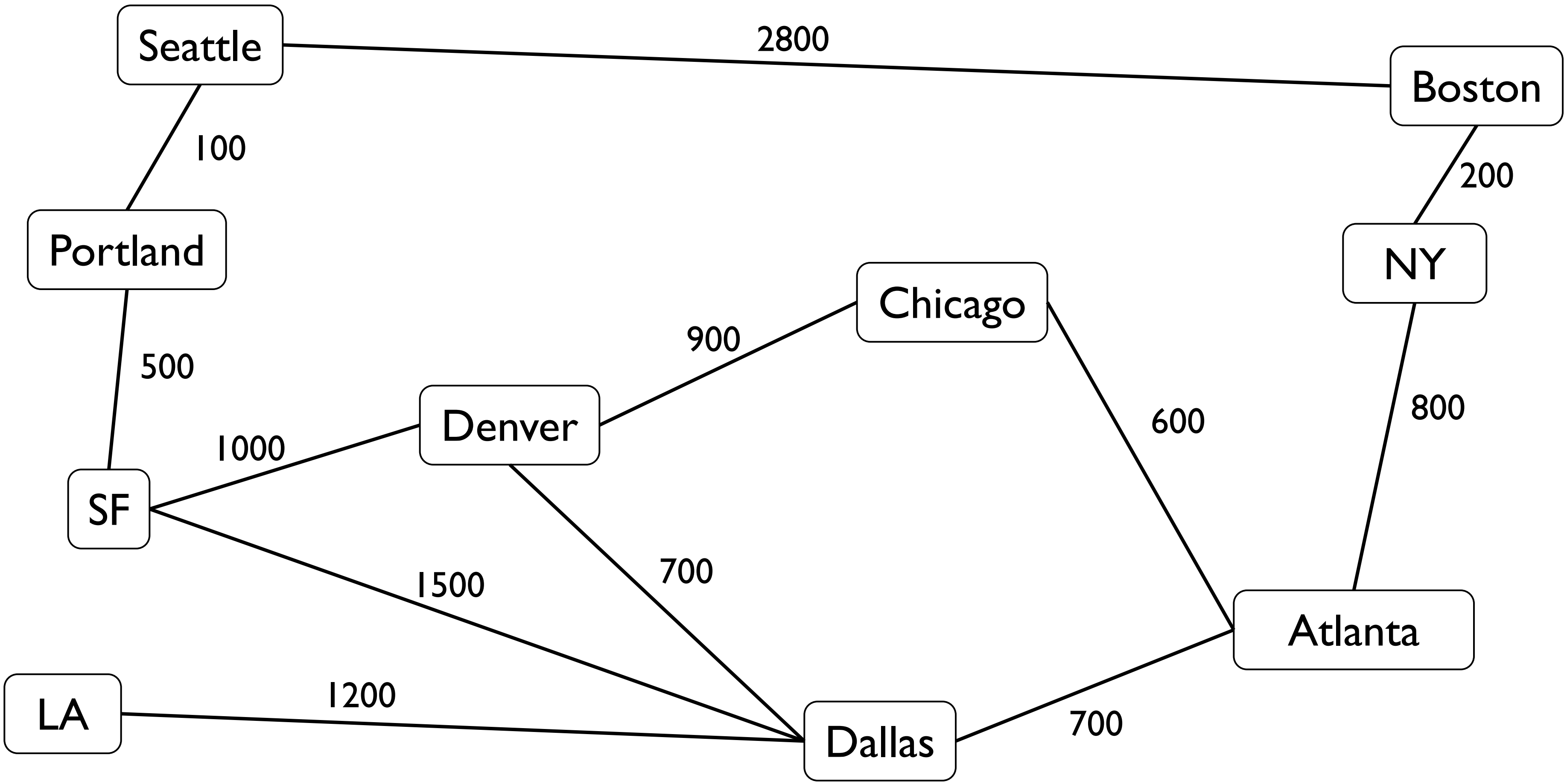
        add e (and v) to T

        for each neighbor w of v

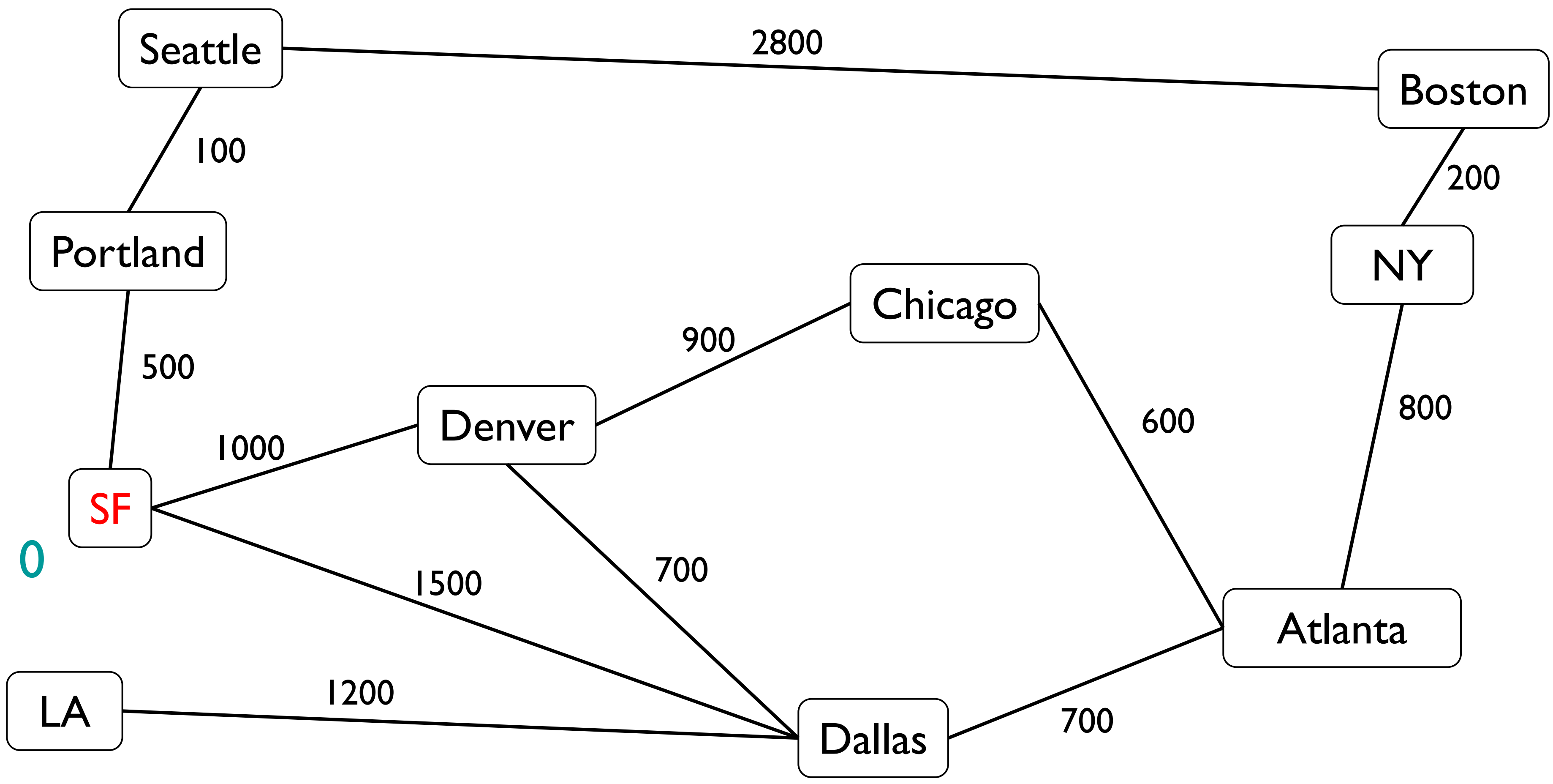
            add edge (v,w) to PQ with weight/key d(s,v) + l(v,w)

Dijkstra: What Do We Return?

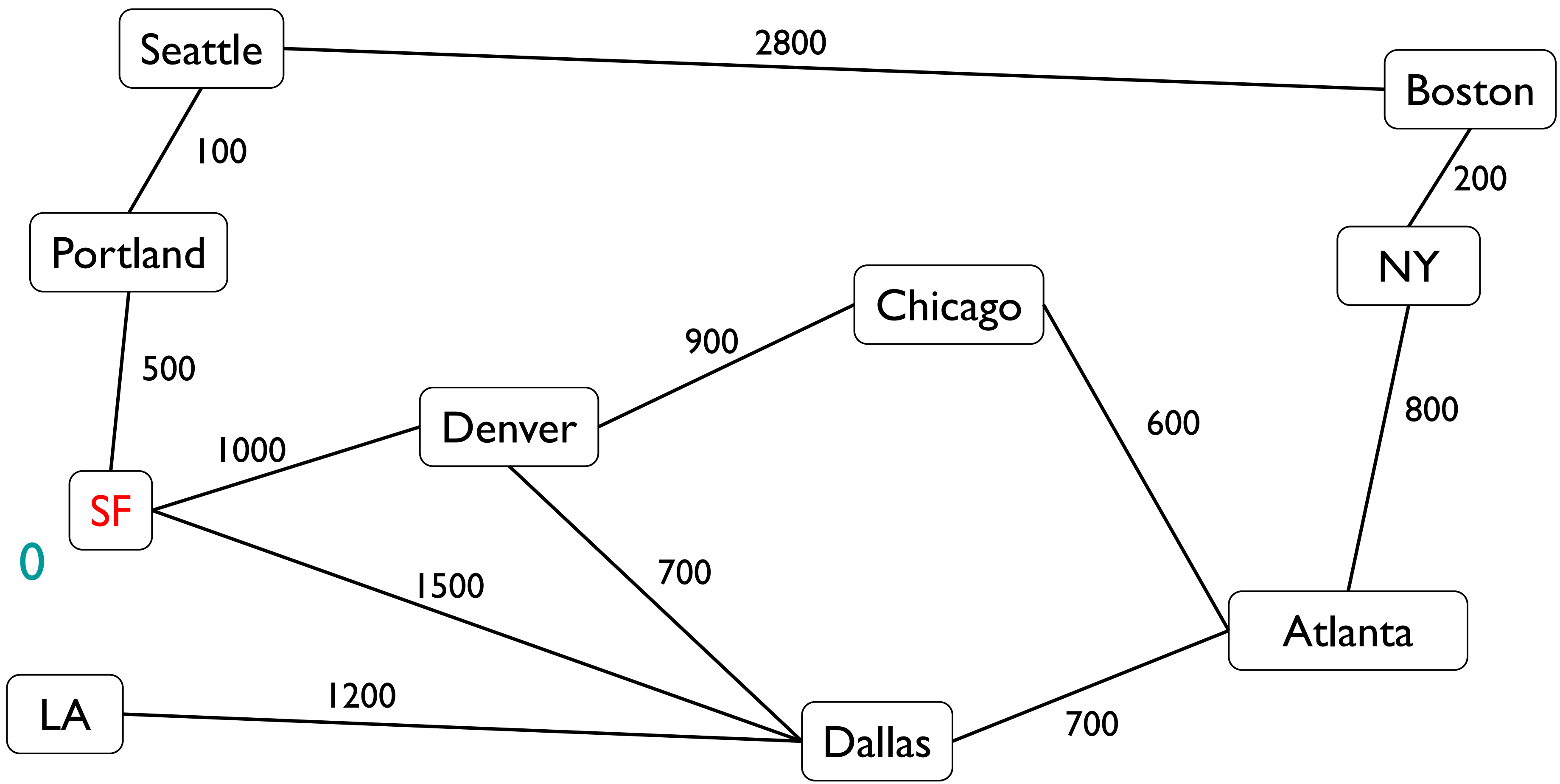
- As we find a new edge e = (v,w) to add to the tree of shortest paths, add it to a map.
- Precisely:
  - Use the PQ association(X,Y) edgeInfo where
    - X is d(s,v) + l(v,w)
    - Y is the edge e=(v,w)
  - Add the key/value pair (w, edgeInfo) to the map
- So the map entry with key w tells us the edge the shortest path used to get to w



Dijkstra's Algorithm

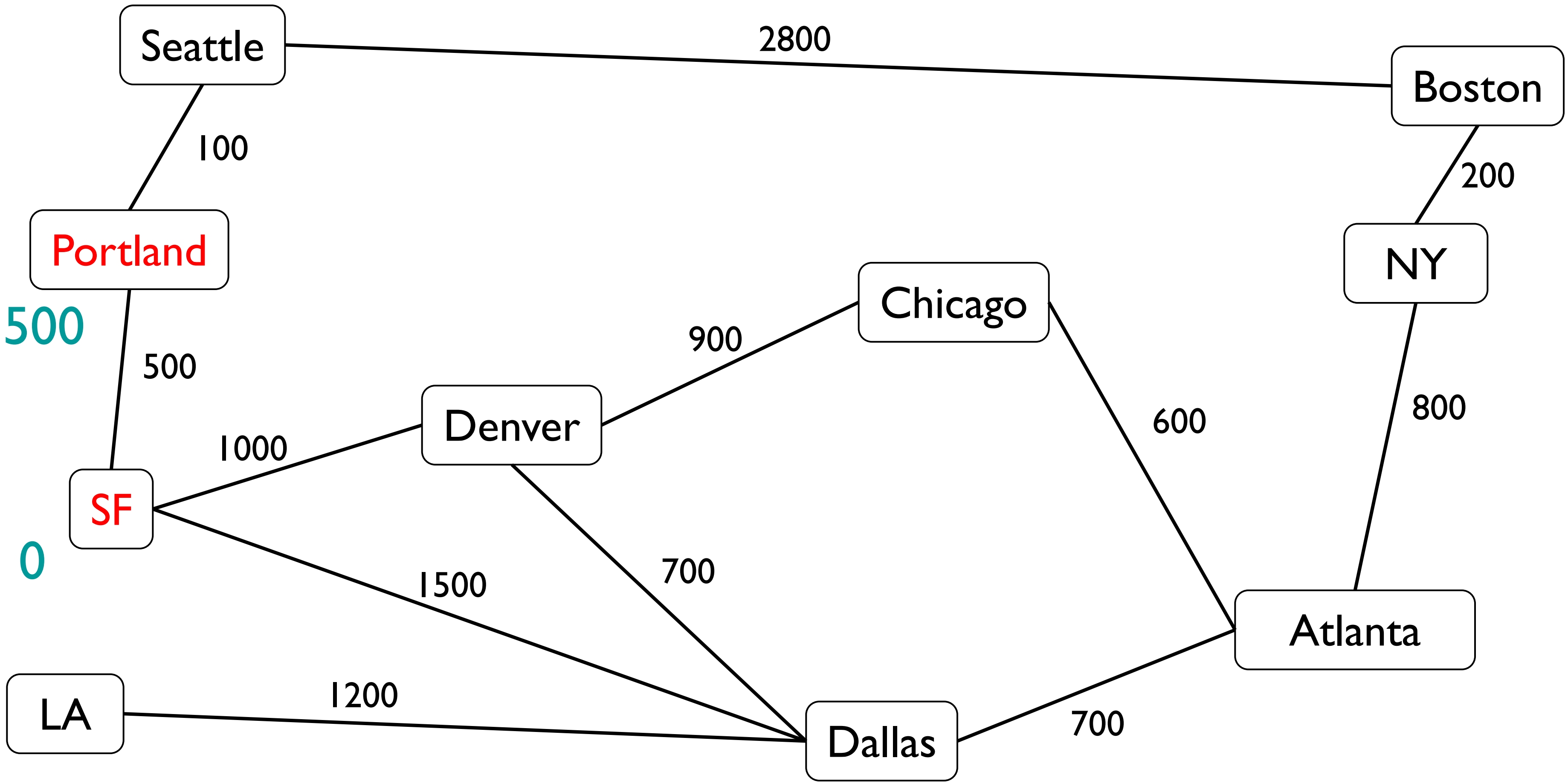


Priority Queue



Priority Queue

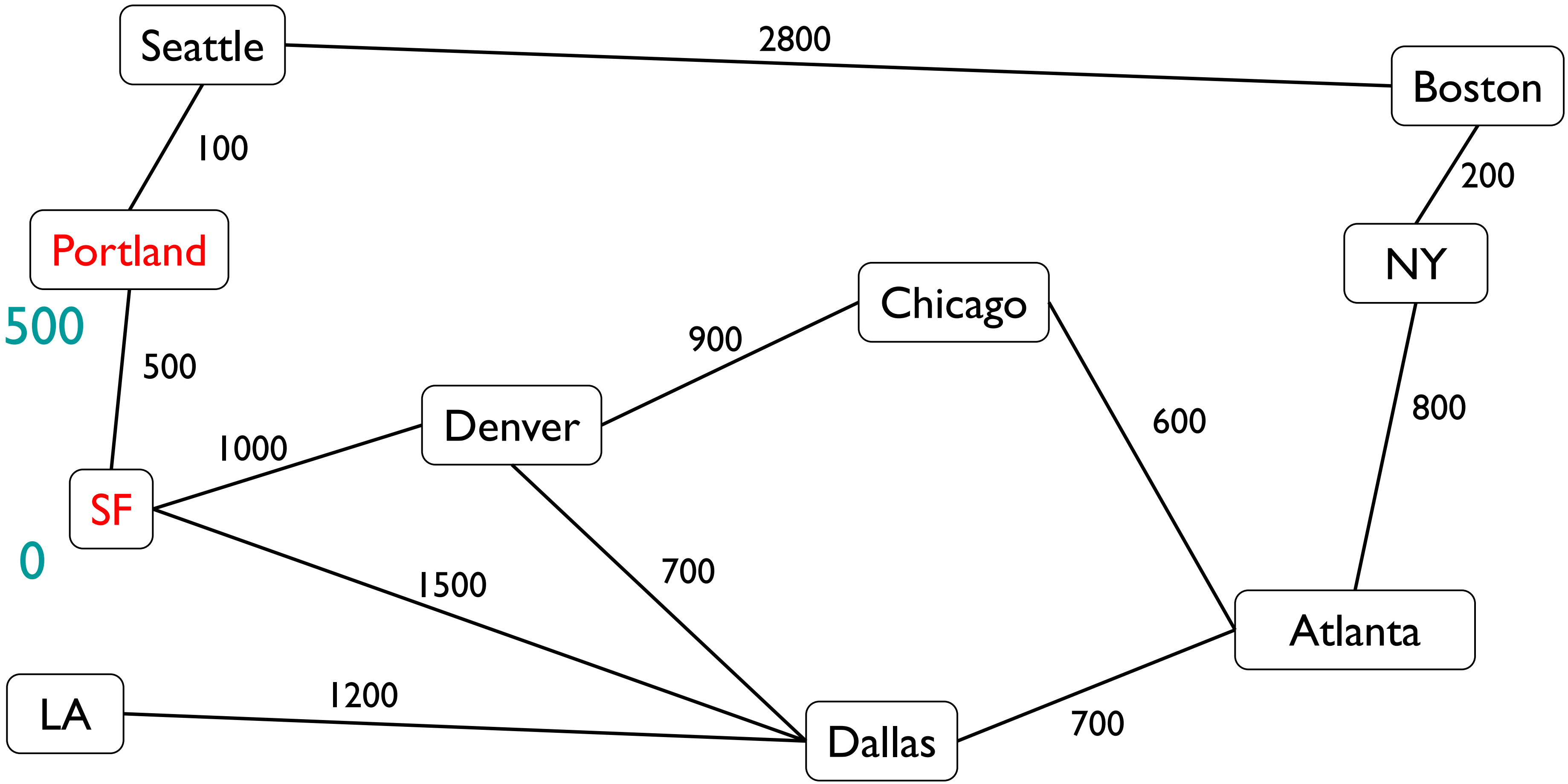
SF->Port; 500 | SF->Den; 1000 | SF->Dal 1500



Current: 500 SF->Port (need to add Port's neighbors to PQ)

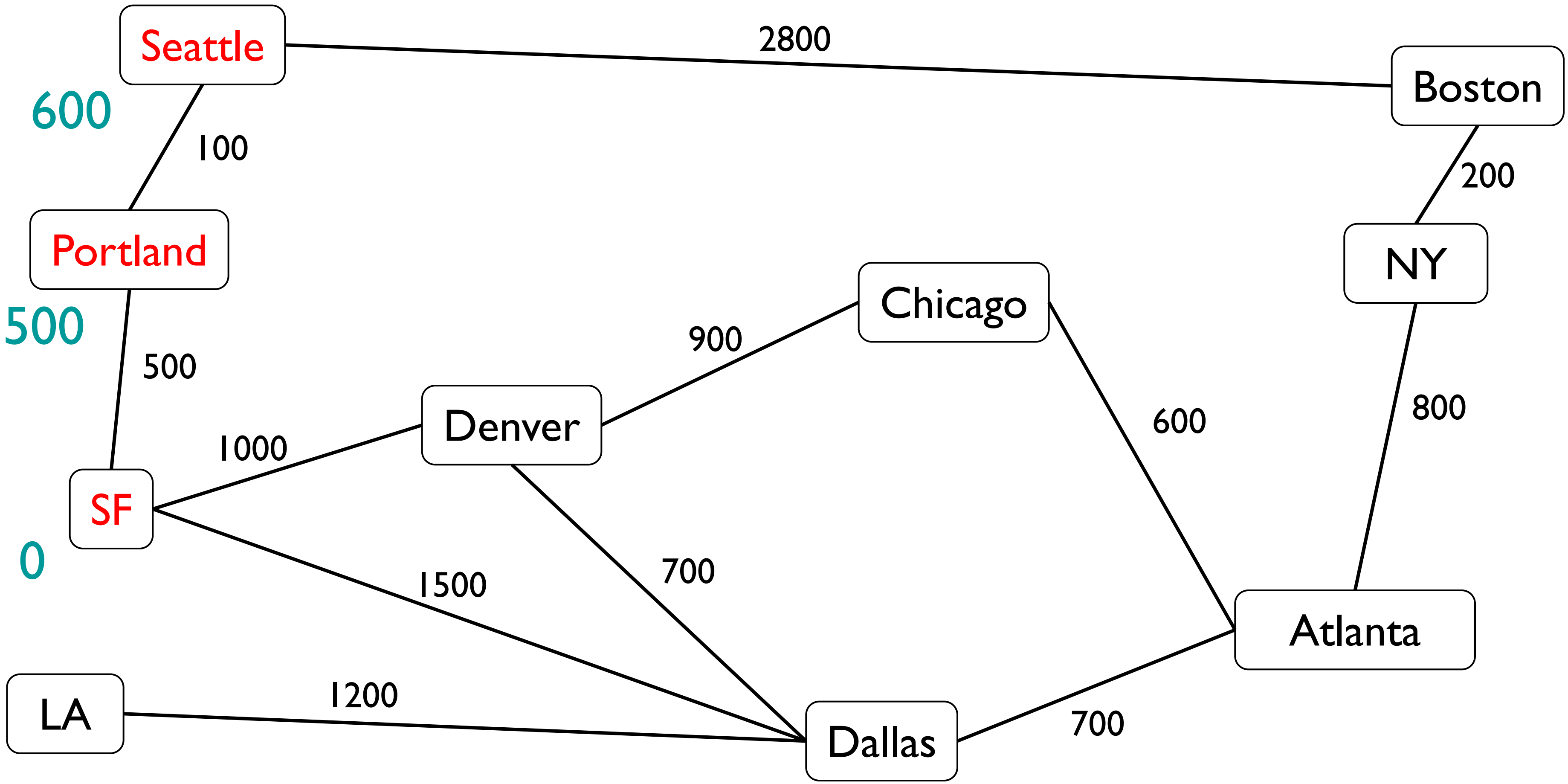
SF->Den; 1000

SF->Dal 1500



Current: 500 SF->Port

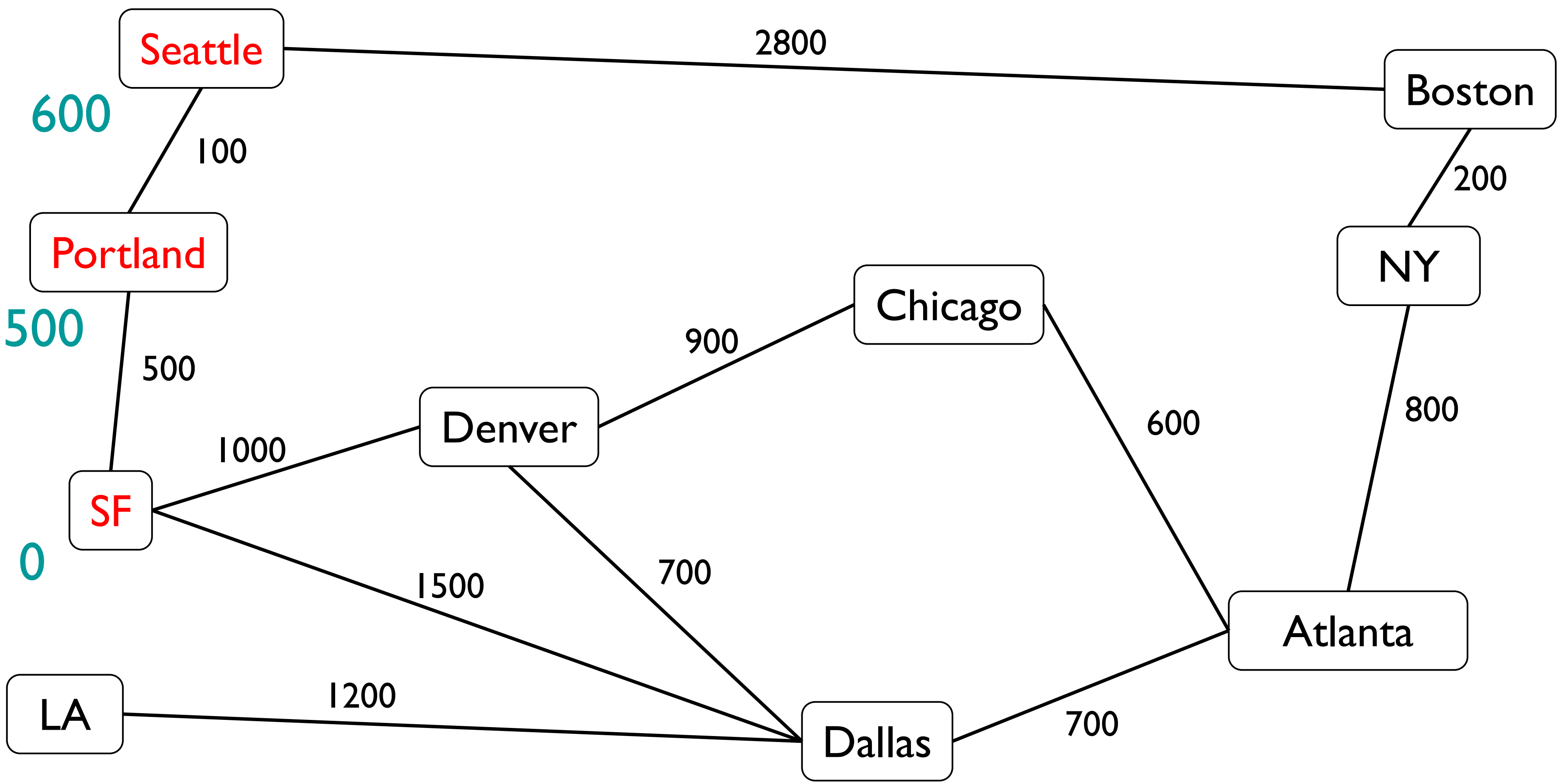
SF->Port->Sea; 600 | SF->Den; 1000 | SF->Dal 1500



Current: 600 SF->Port->Sea

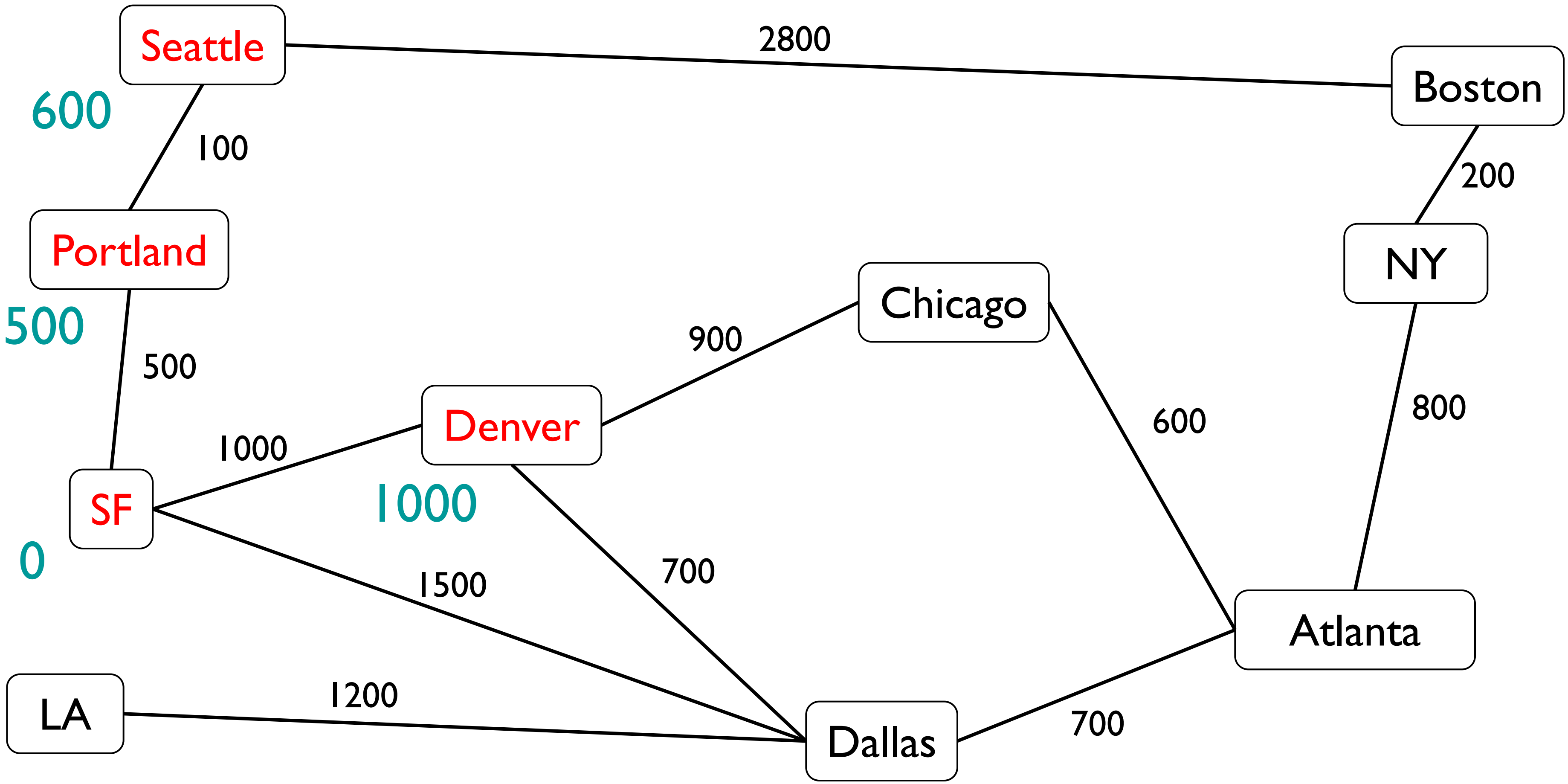
SF->Den; 1000

SF->Dal 1500



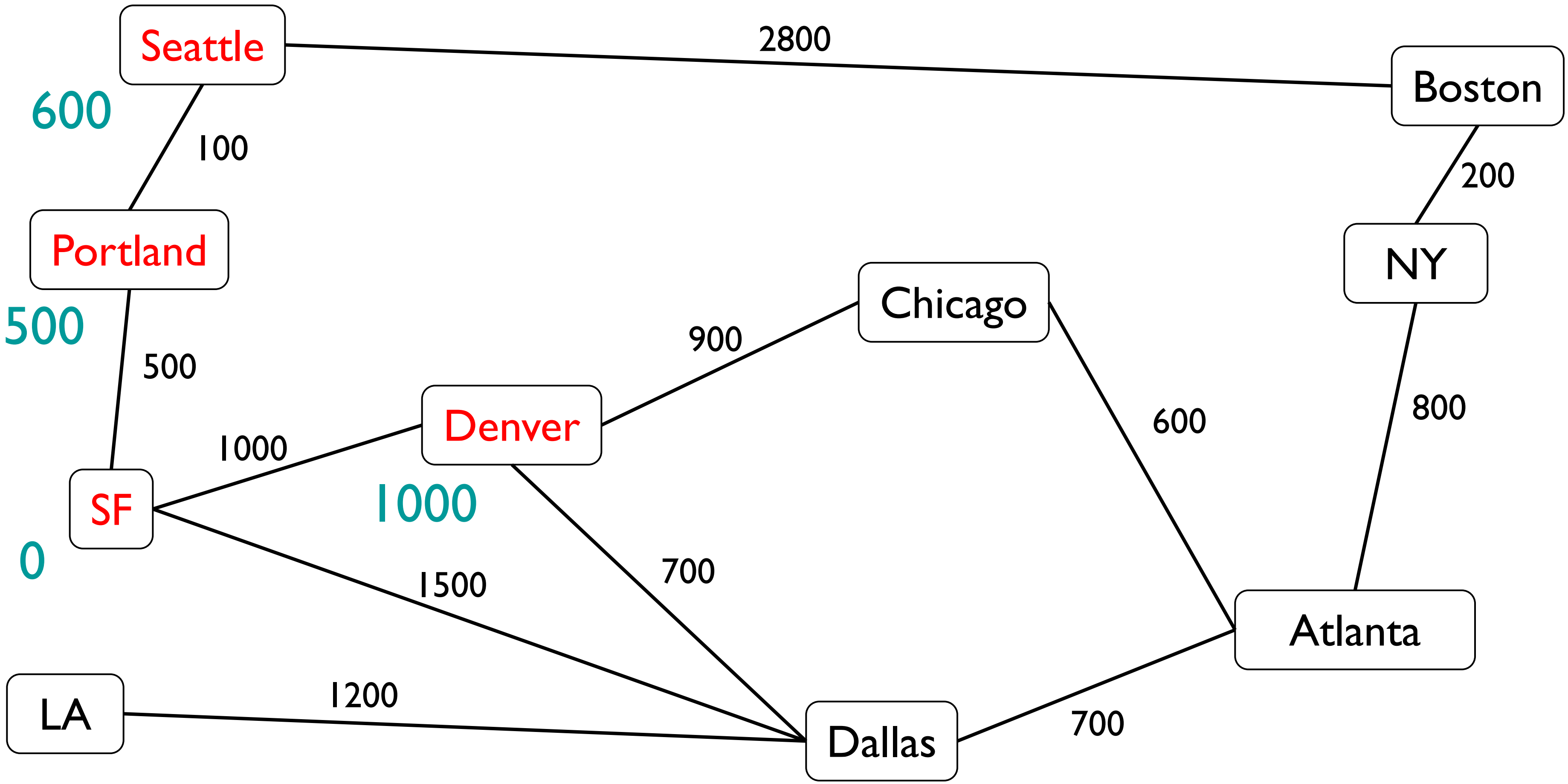
Current: 600 SF->Port->Sea

SF->Den; 1000	SF->Dal; 1500	SF->Port->Sea->Bos 3400



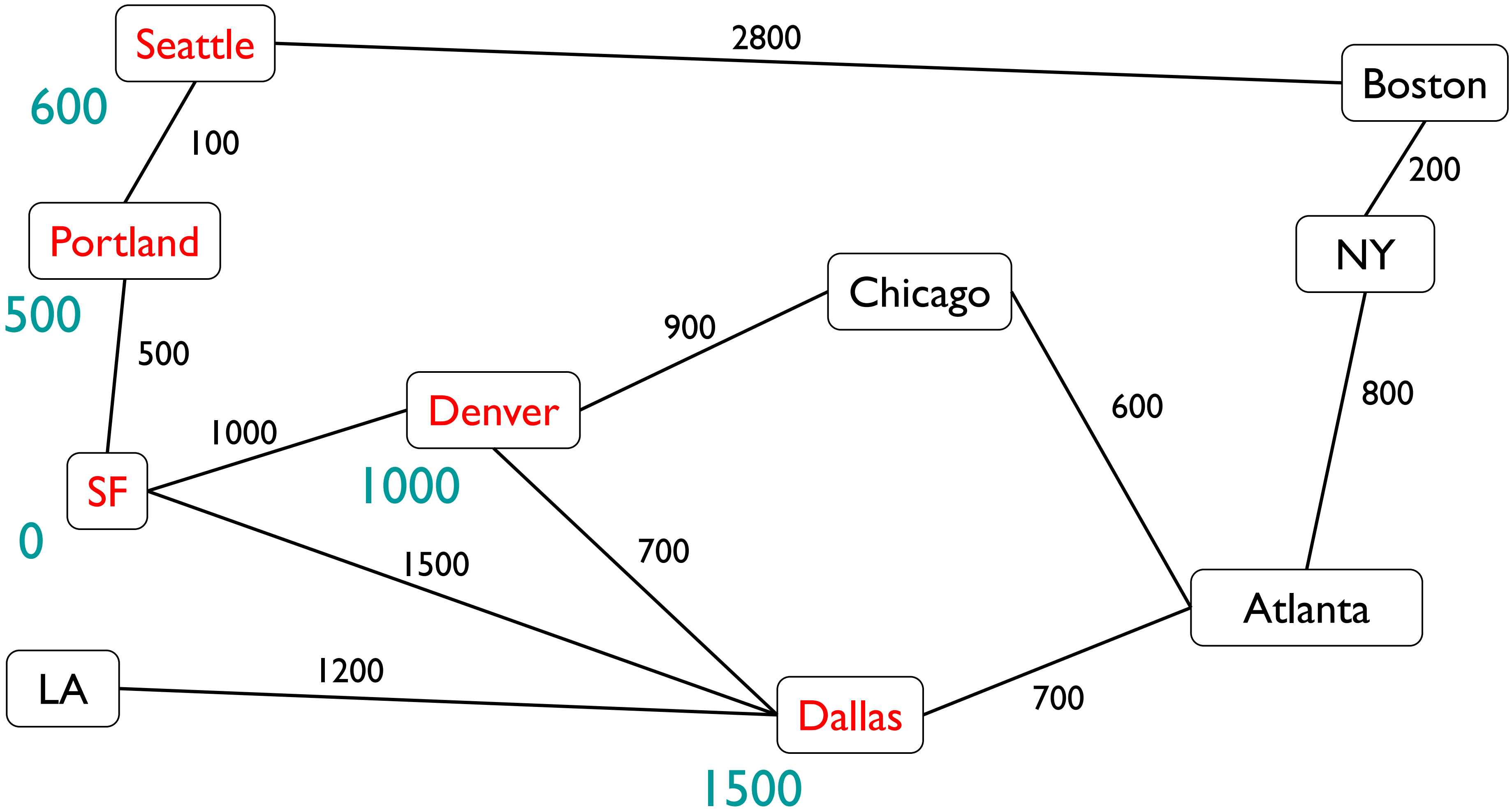
Current: 1000 SF->Den

SF->Dal; 1500	SF->Port->Sea->Bos 3400



Current: 1000 SF->Den

SF->Dal; 1500	SF->Den->Dal; 1700	SF->Den->Chi; 1900	SF->Port->Sea->Bos 3400

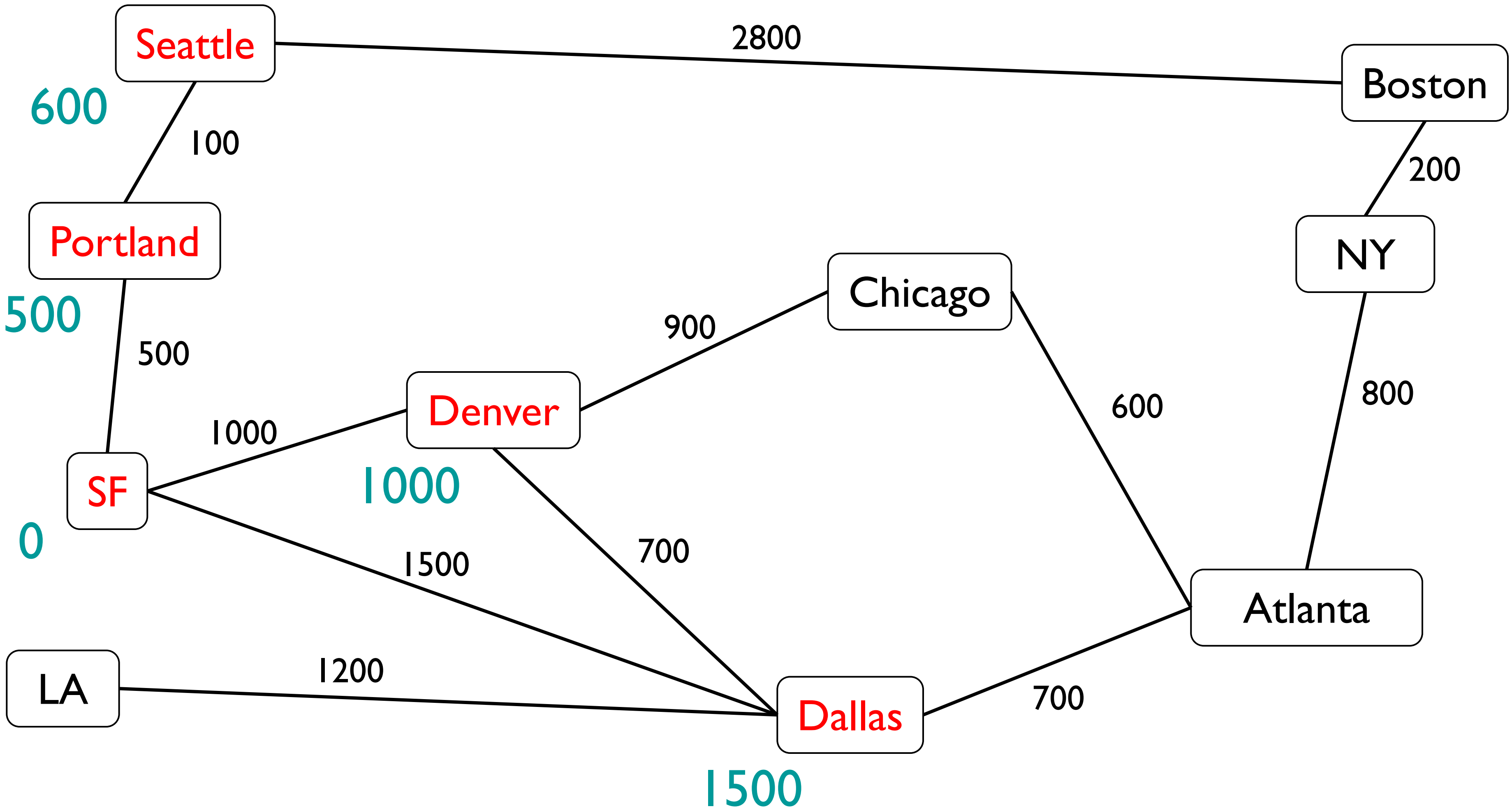


Current: 1500 SF->Dal

SF->Den->Dal; 1700

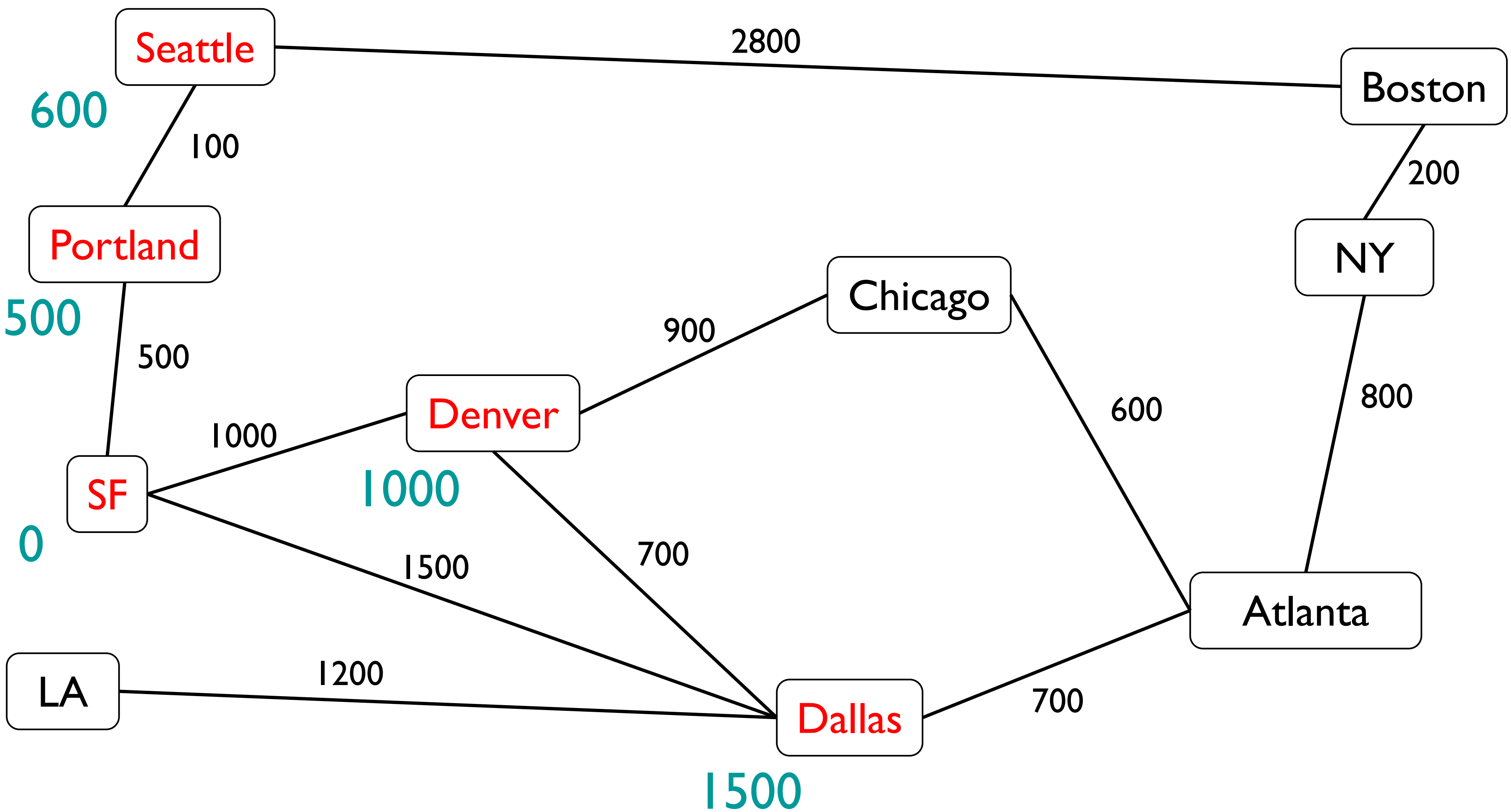
SF->Den->Chi; 1900

SF->Port->Sea->Bos 3400



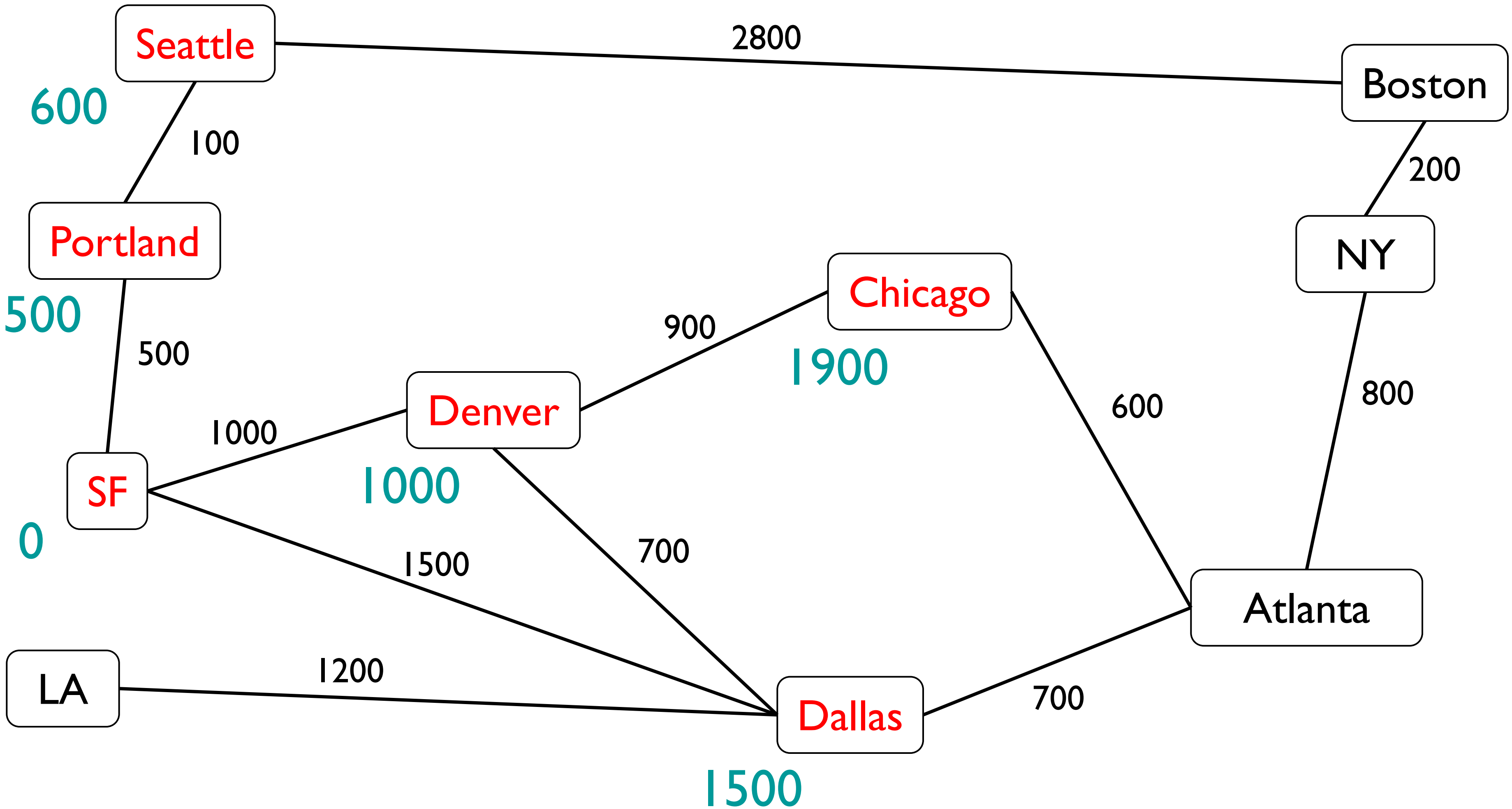
Current: 1500 SF->Dal

SF->Den->Dal; 1700	SF->Den->Chi; 1900	SF->Dal->Atl; 2200	SF->Dal->LA; 2700	SF->Port->Sea->Bos 3400



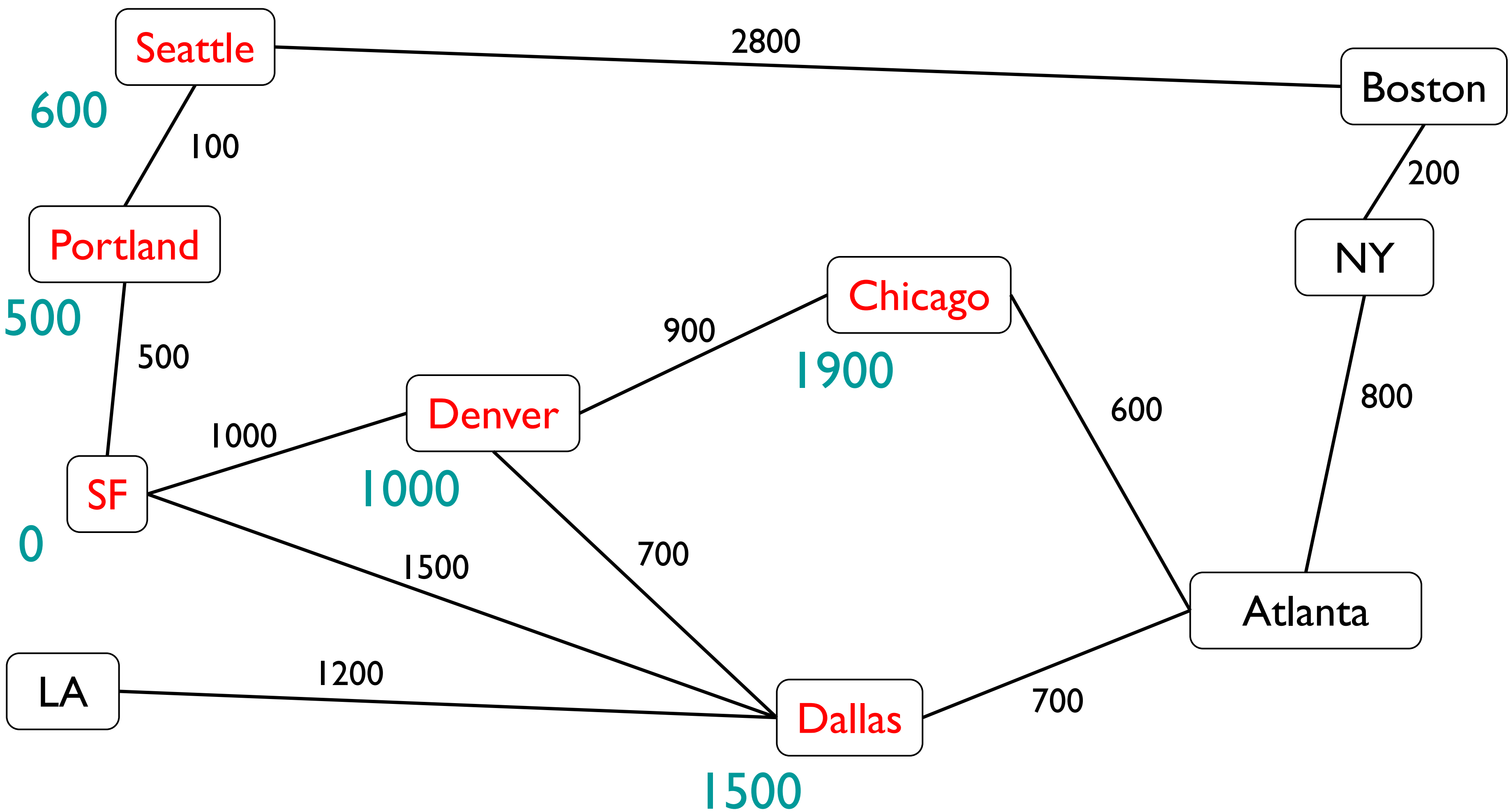
Current: 1700 SF->Den->Dal (we already have Dallas!)

SF->Den->Chi; 1900	SF->Dal->Atl; 2200	SF->Dal->LA; 2700	SF->Port->Sea->Bos 3400



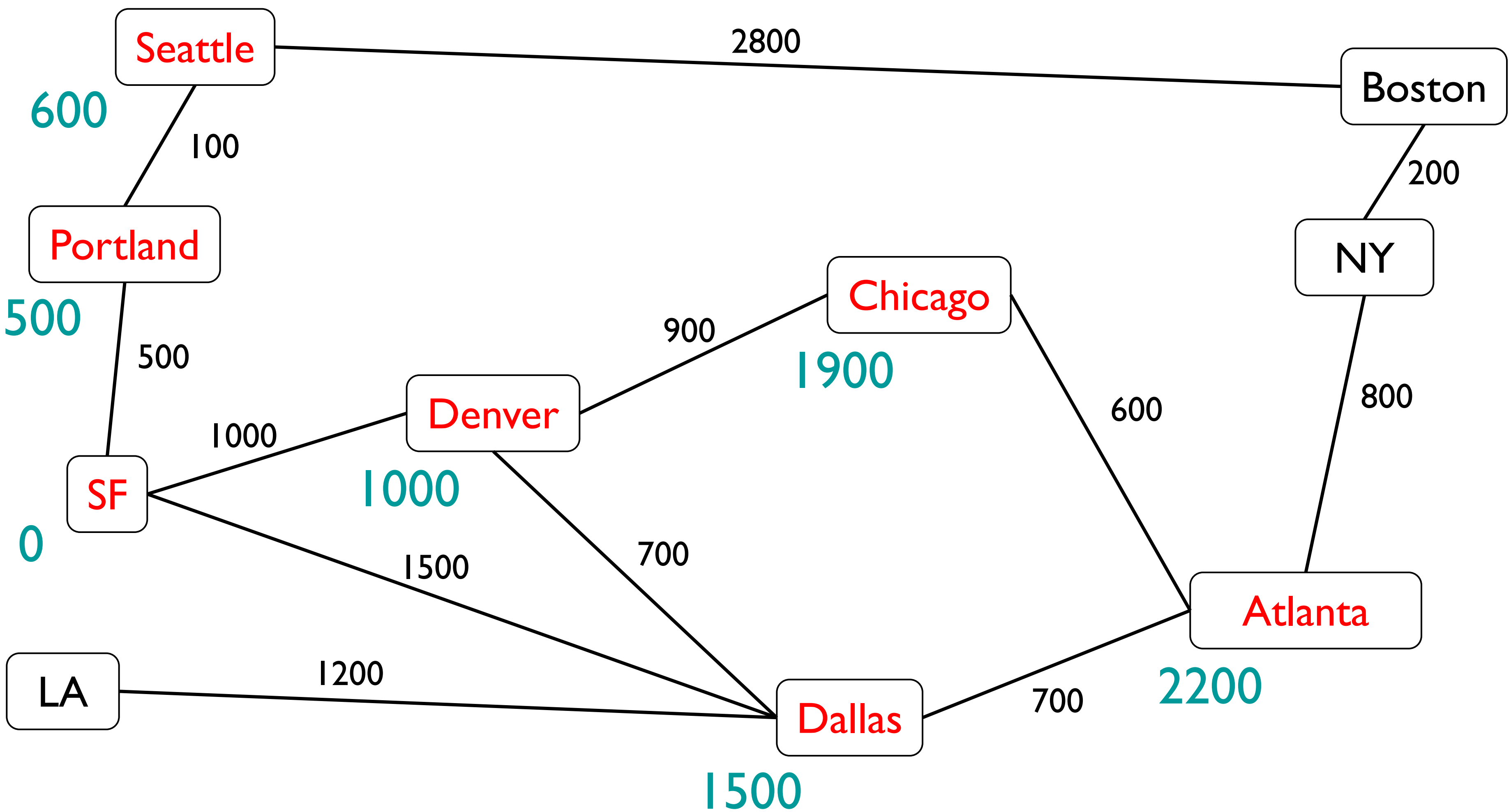
Current: 1900 SF->Den->Chi

SF->Dal->Atl; 2200 SF->Dal->LA; 2700 3400 SF->Port->Sea->Bos



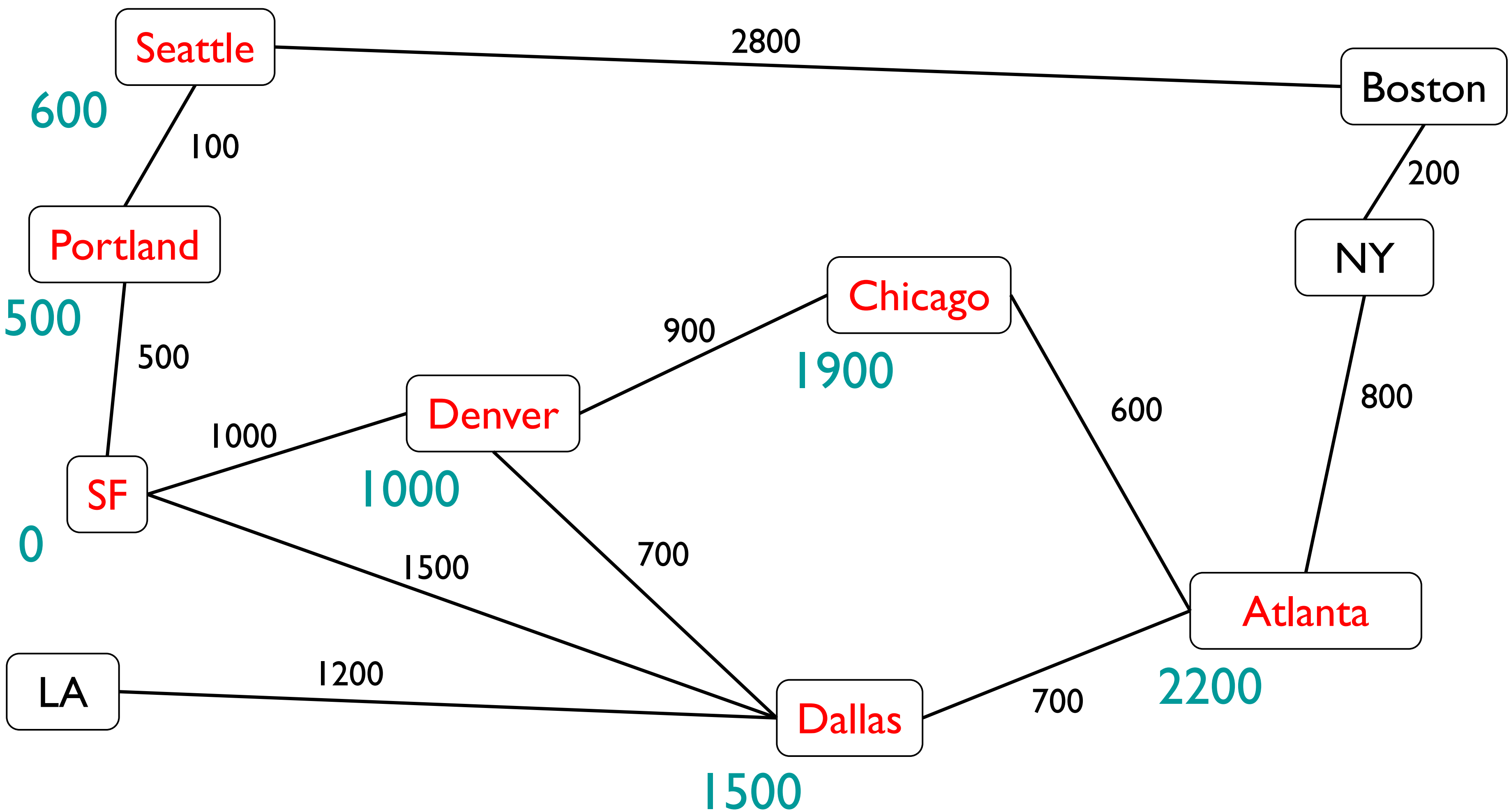
Current: 1900 SF->Den->Chi

SF->Dal->Atl; 2200	SF->Den->Chi->Atl; 2500	SF->Dal->LA; 2700	SF->Port->Sea->Bos 3400



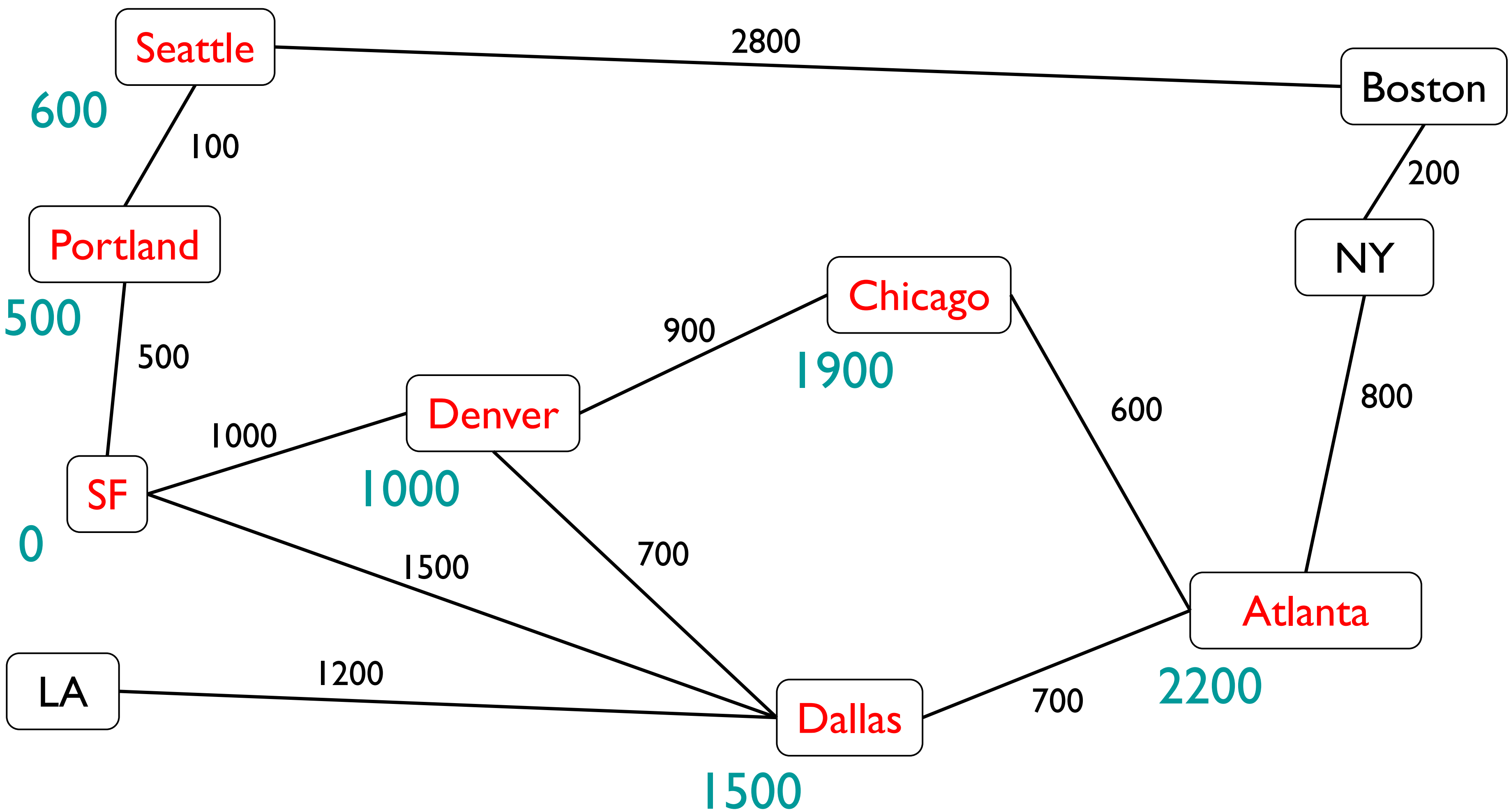
Current: 2200 SF->Dal->Atl

SF->Den->Chi->Atl; 2500 SF->Dal->LA; 2700 SF->Port->Sea->Bos 3400



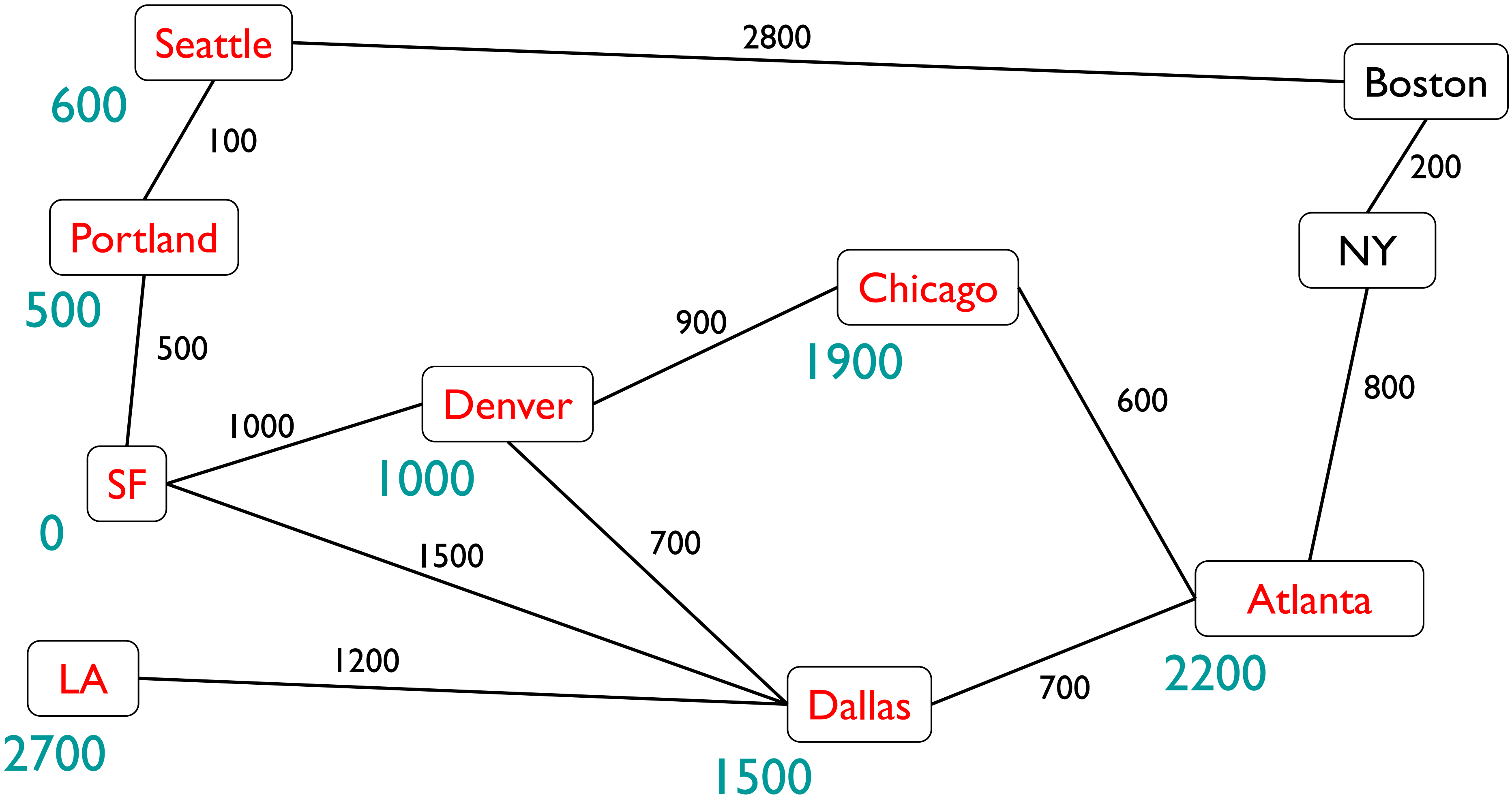
Current: 2200 SF->Dal->Atl

SF->Den->Chi->Atl; 2500 | SF->Dal->LA; 2700 | SF->Dal->Atl->NY; 3000 | SF->Port->Sea->Bos 3400



Current: 2500 SF->Den->Chi->Atl

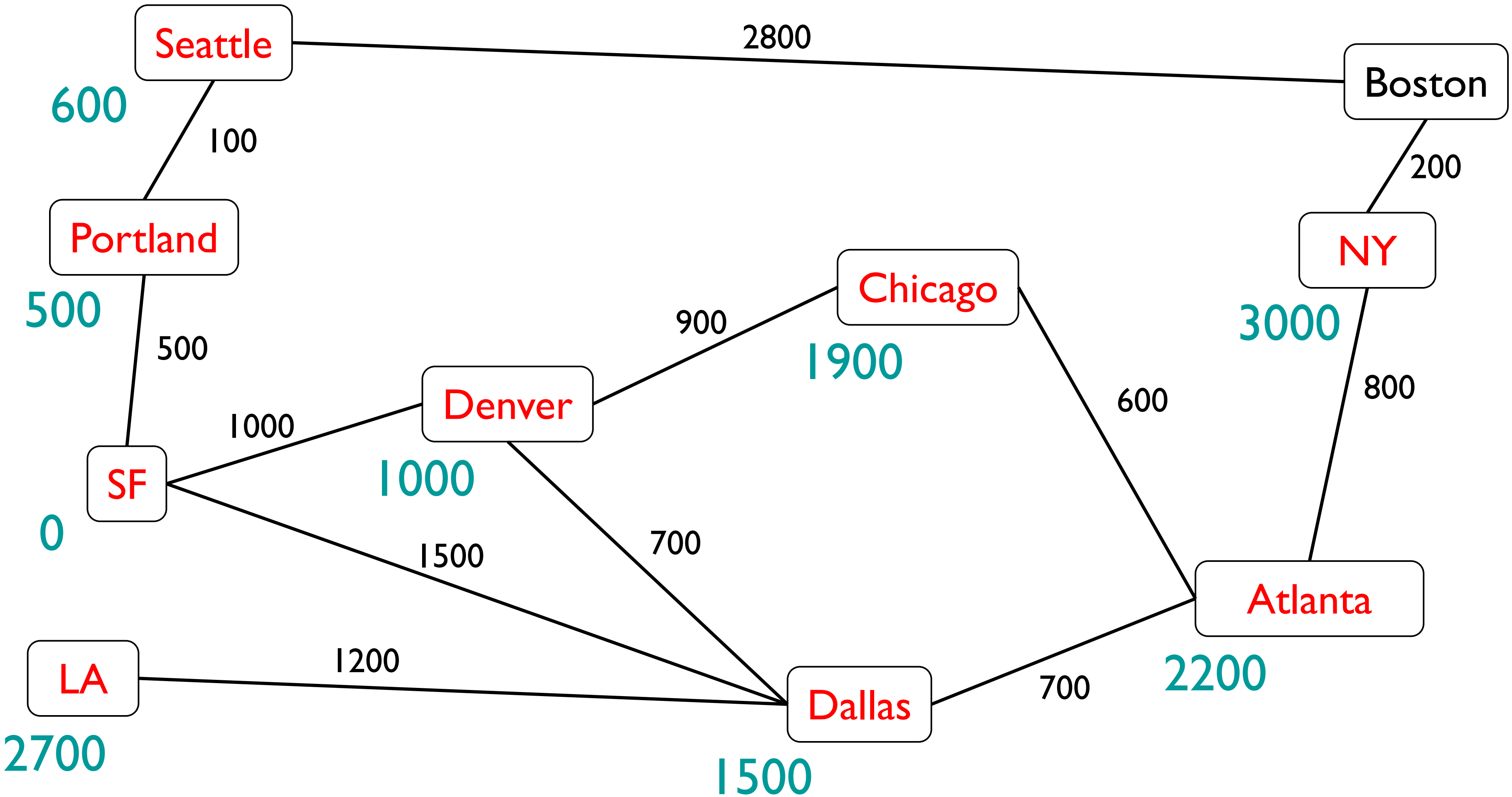
SF->Dal->LA; 2700	SF->Dal->Atl->NY; 3000	SF->Port->Sea->Bos 3400



Current: 2700 SF->Dal->LA

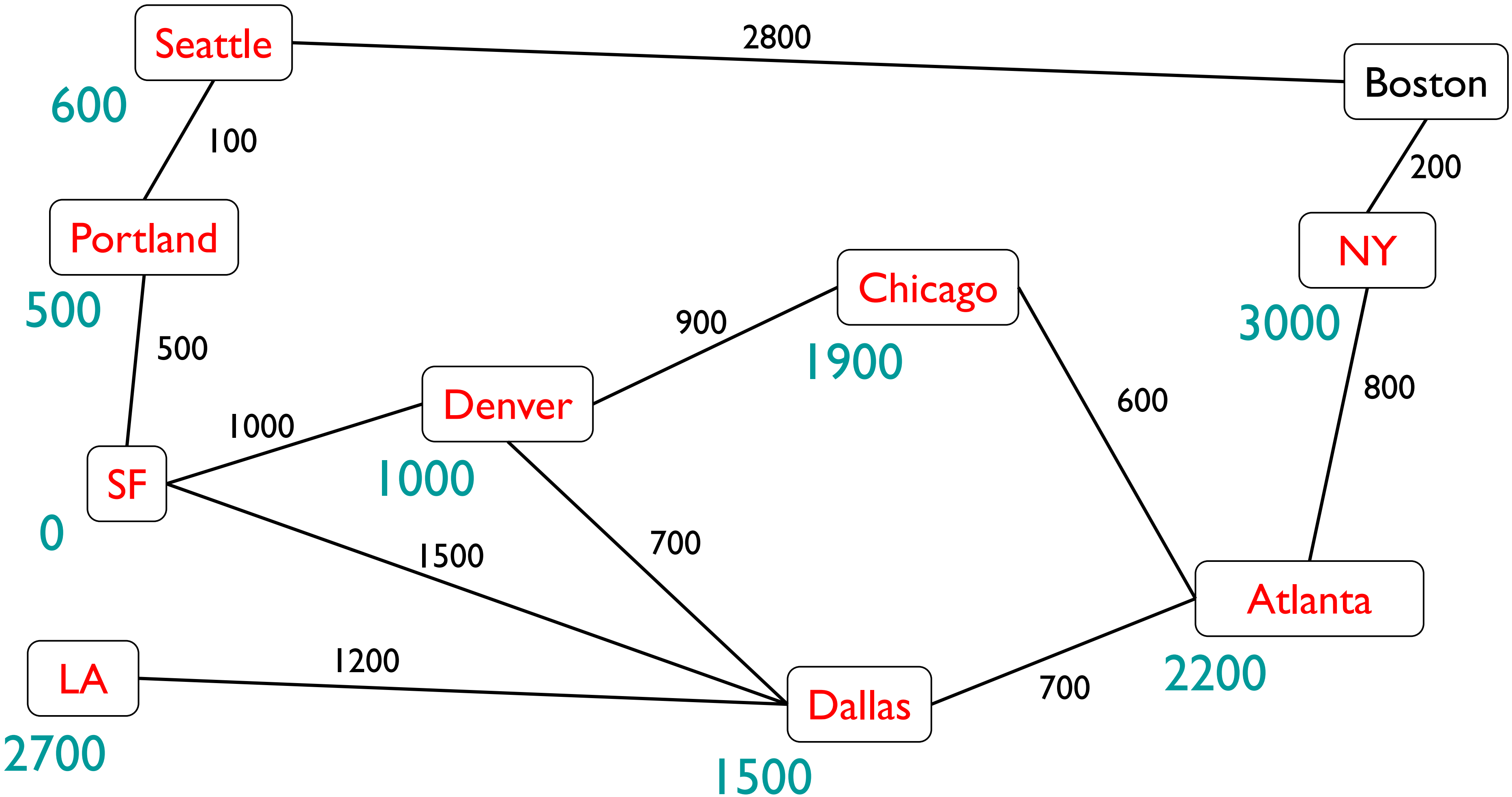
SF->Dal->Atl->NY; 3000

SF->Port->Sea->Bos 3400



Current: 3000 SF->Dal->Atl->NY

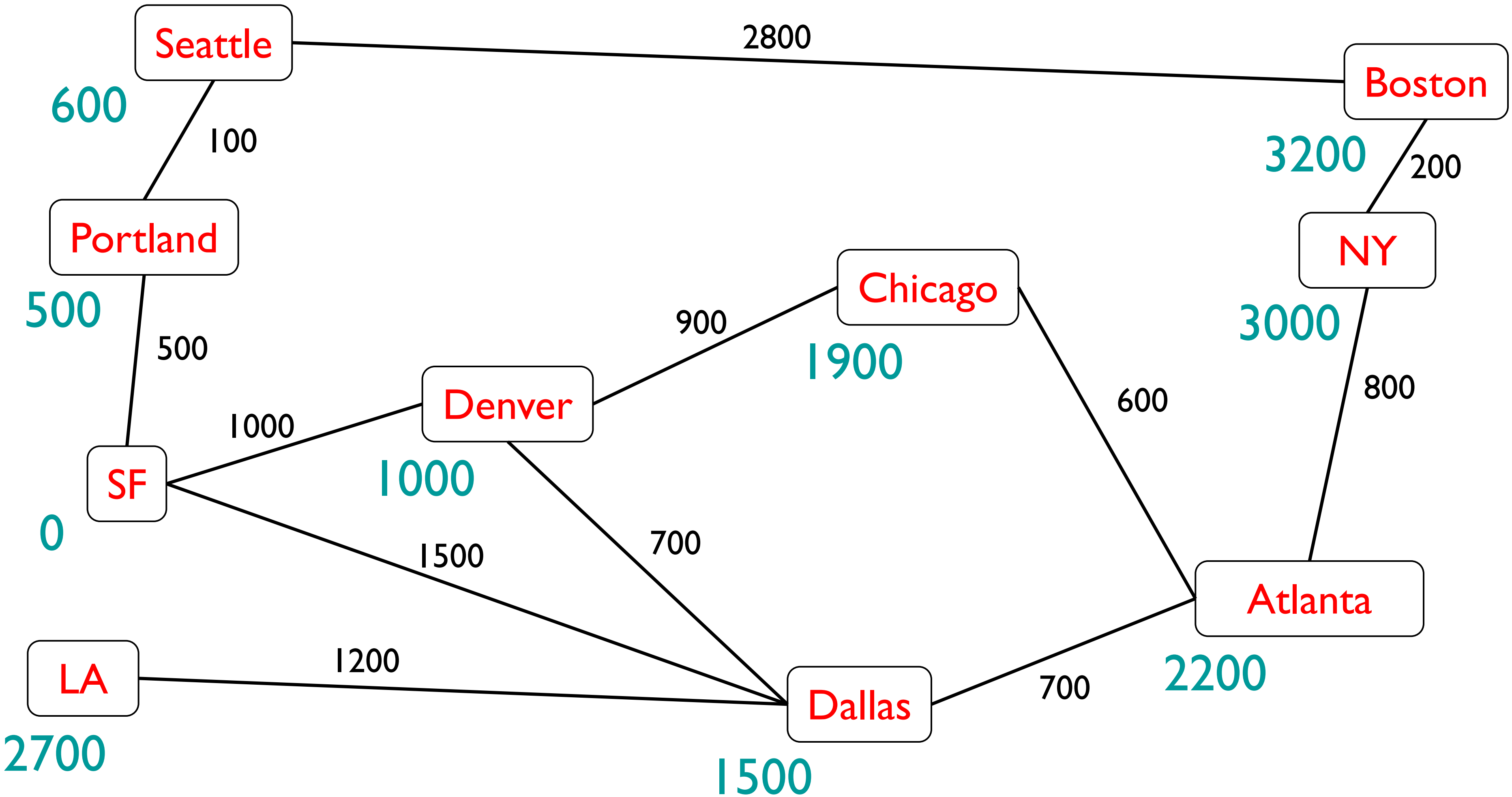
SF->Port->Sea->Bos
3400



Current: 3000 SF->Dal->Atl->NY

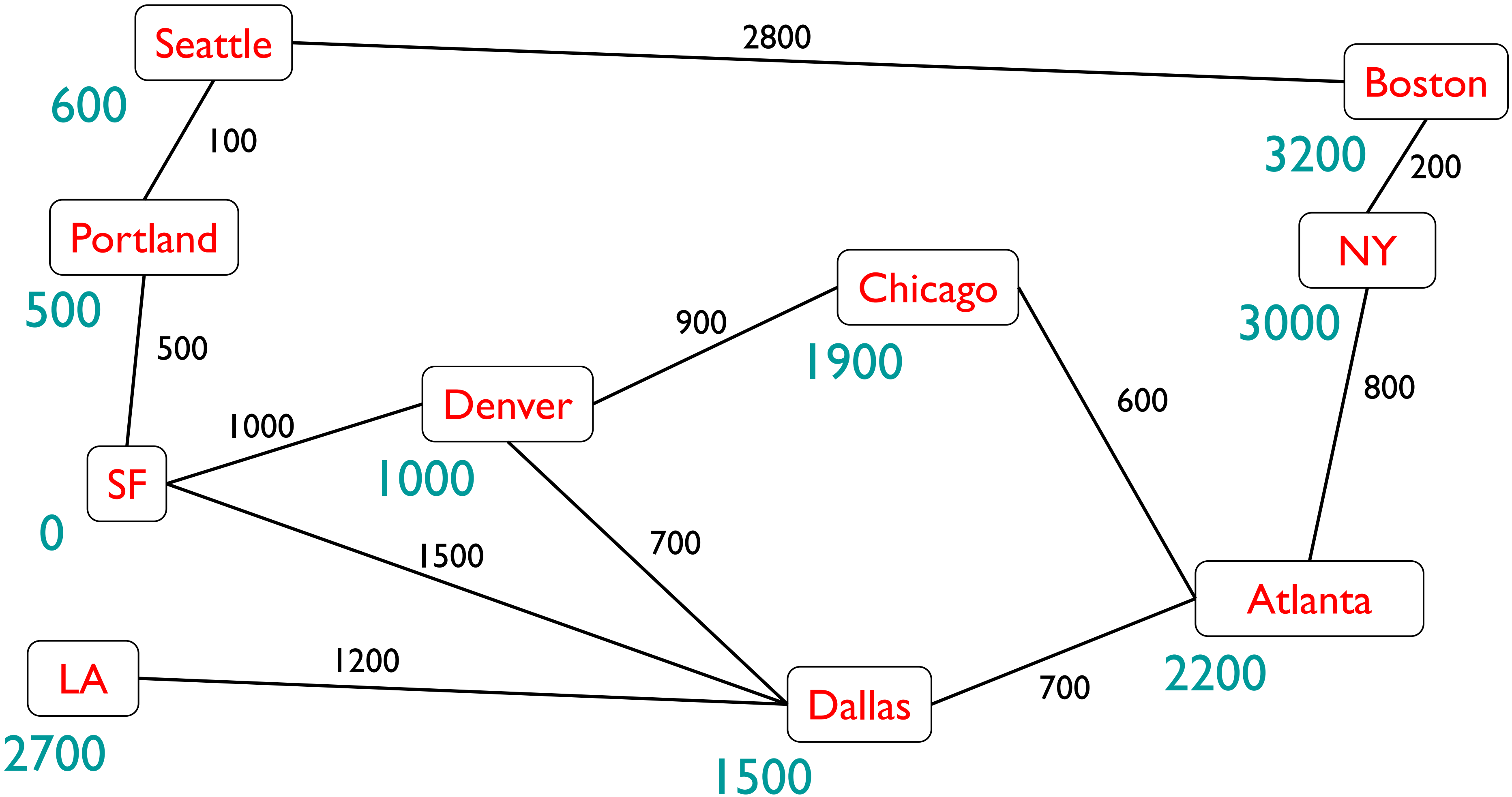
SF->Dal->Atl->NY->Bos; 3200

SF->Port->Sea->Bos 3400

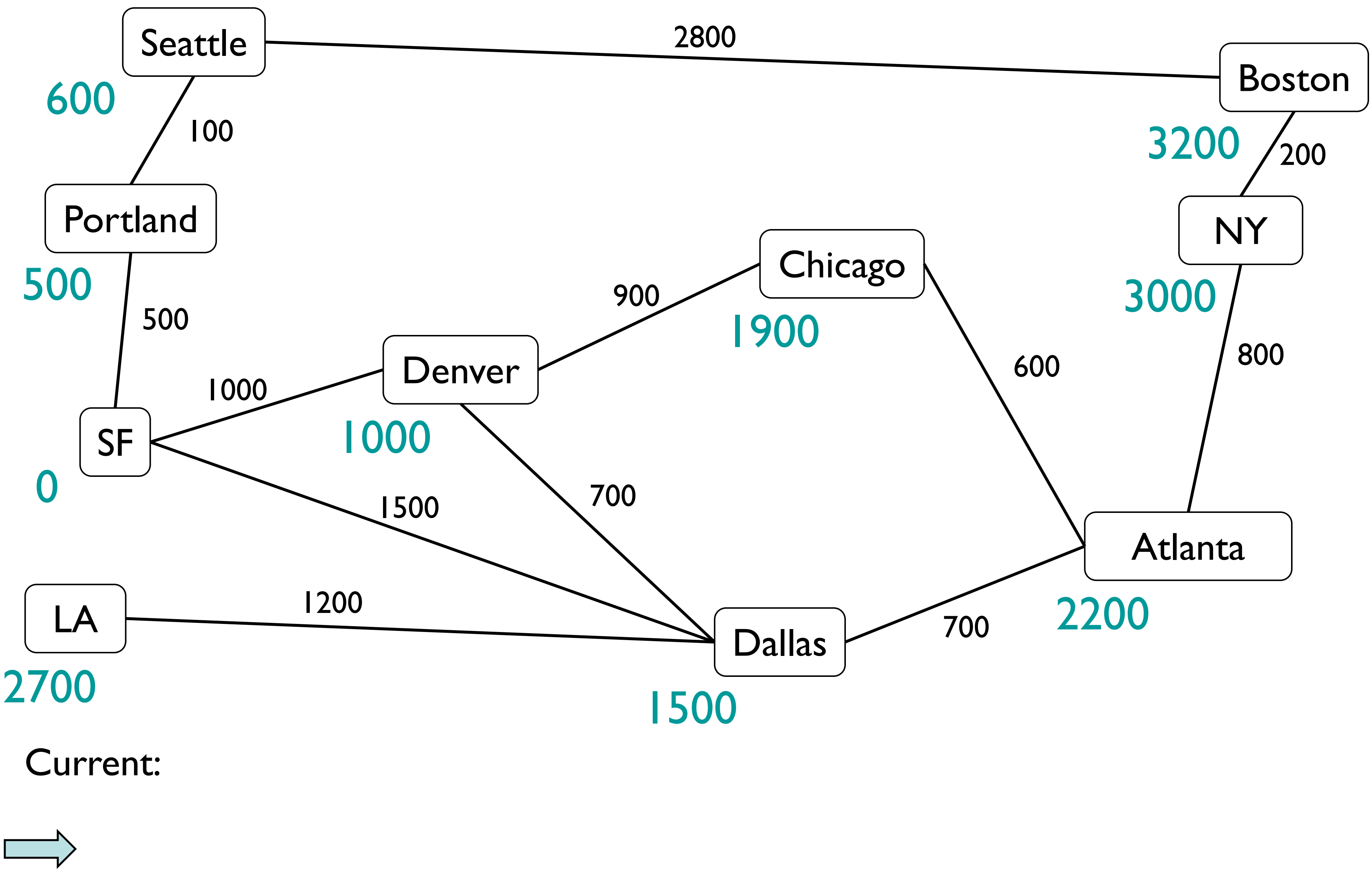


Current: 3200 SF->Dal->Atl->NY->Bos

SF->Port->Sea->Bos
3400



Current: 3400 SF->Port->Sea->Bos



Seattle
600
2800
Boston
3200
100
200
Portland
500
NY
3000
Chicago
1900
900
500
600
800
Denver
1000
1000
SF
0
1500
700
Atlanta
2200
LA
2700
1200
700
Dallas
1500
Current:

Dijkstra: Space Complexity

- Graph: $O(|V| + |E|)$
  - Each vertex and edge uses a constant amount of space
- Priority Queue $O(|E|)$
  - Each edge takes up constant amount of space
- Are there any hidden space costs?
- Result: $O(|V| + |E|)$
  - Optimal in Big-O sense!

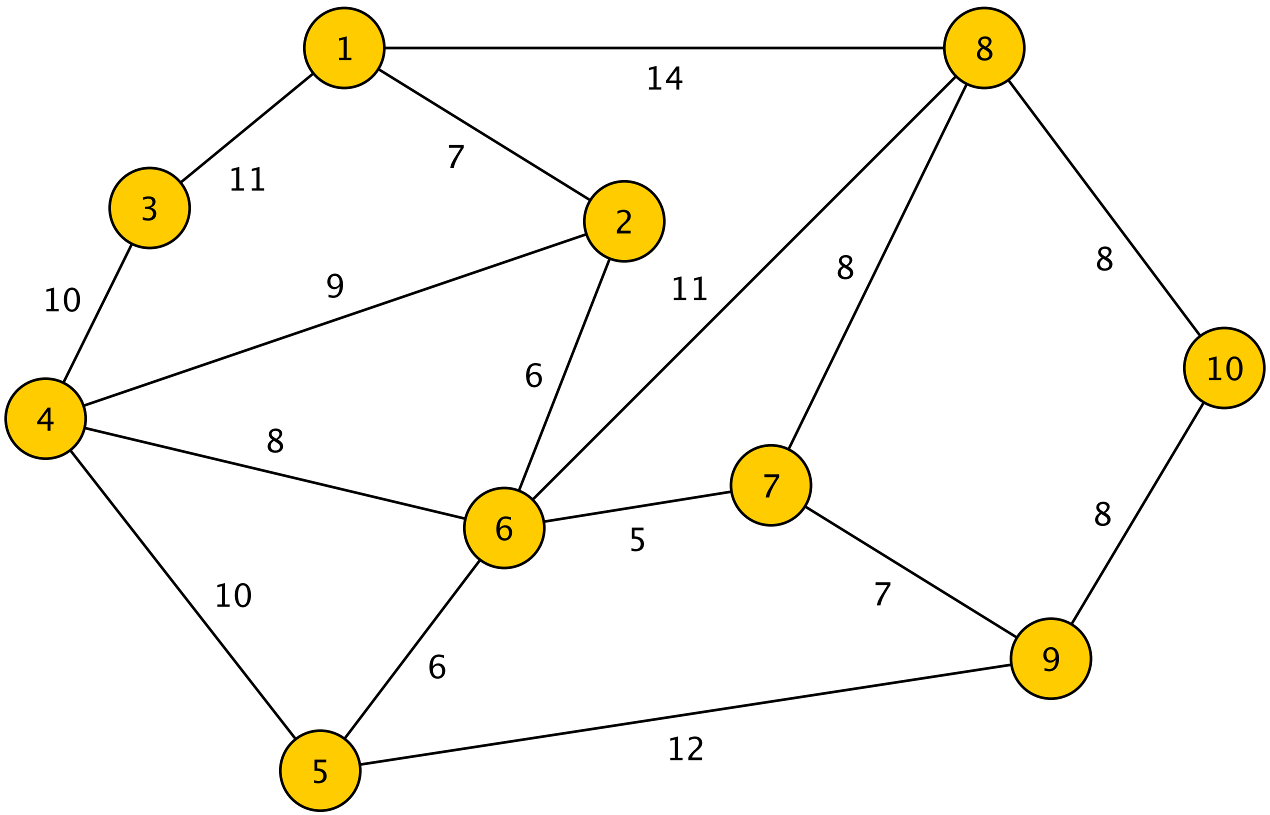
Dijkstra : Time Complexity

Assume Map ops are O(1) time

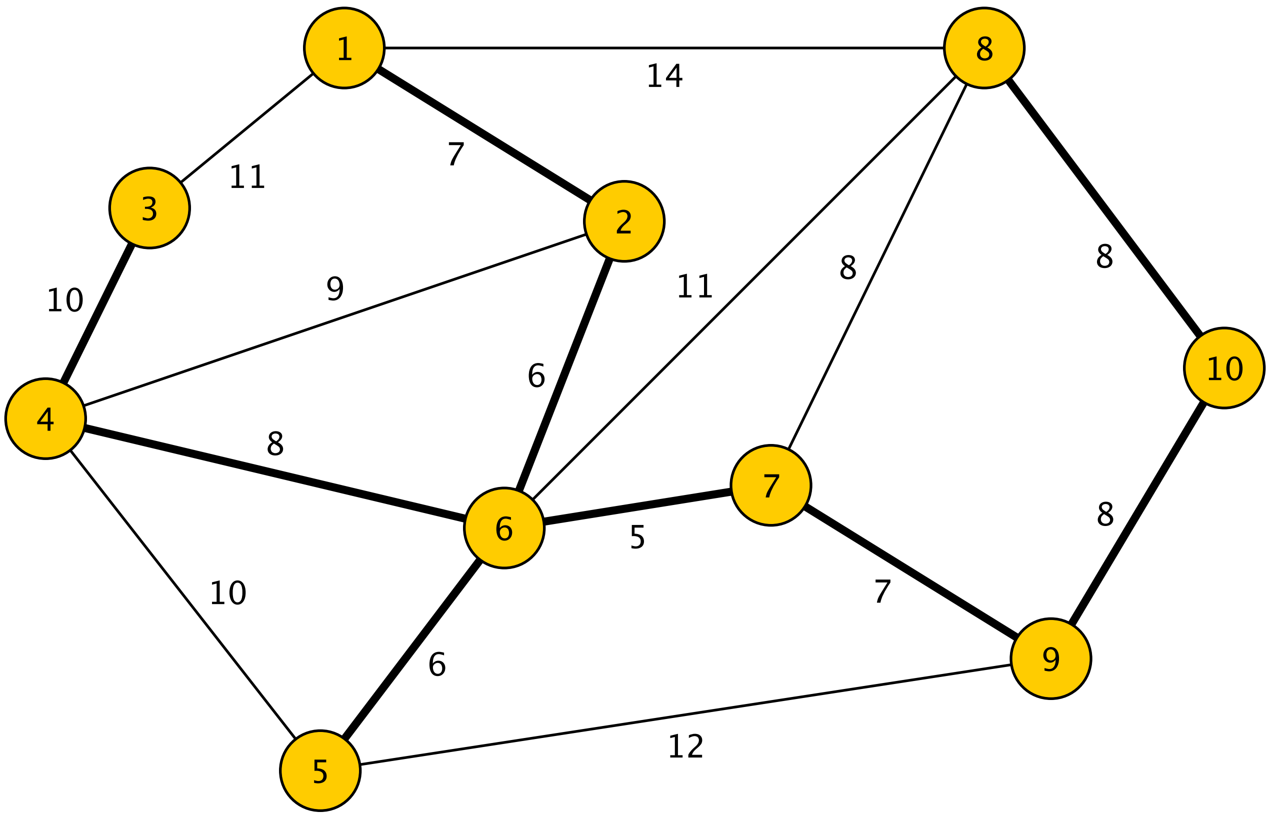
Across *all* iterations of outer while loop

- Edges are added to and removed from the priority queue
  - But any edge is added (and removed) at most once!
  - Total PQ operation cost is O(|E| log |E|) time
    - Which is O(|E| log |V|) time
  - All other operations take constant time
- Thus time complexity is O(|E| log |V|)

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees



Basic Graph Properties

- A *subgraph* of a graph G=(V, E) is a graph G'=(V',E') where
  - $V' \subseteq V$
  - $E' \subseteq E$, and
  - If $e \in E'$ where $e = \{u,v\}$, then $u, v \in V'$
- Special Subgraphs
  - If E' contains *every* edge of E having both ends in V', then G' is called the subgraph of G *induced by* V'
  - If V' = V, then G' is called a *spanning subgraph of G*

Basic Graph Properties

- Recall: An undirected graph G=(V,E) is *connected* if for every pair u,v in V, there is a path from u to v (and so from v to u)
- The maximal sized connected subgraphs of G are called its *connected components*
  - Note: They are induced subgraphs of G
- An undirected graph without cycles is a *forest*
- A connected forest is called a *tree*.
  - Not to be confused with the data structure!

Facts About Graphs

Thm: If $G=(V,E)$ is a forest with $|E| > 0$, then G has at least one vertex v of degree 1 (a *leaf*)

- Hint: Consider a longest simple path in G…

Thm: If $G=(V,E)$ is a tree then $|E| = |V| - 1$.

- Hint: Induction on v: delete a leaf

Thm: Every connected graph $G=(V,E)$ contains a spanning subgraph $G'=(V,E')$ that is a tree

- That is, a *spanning tree*

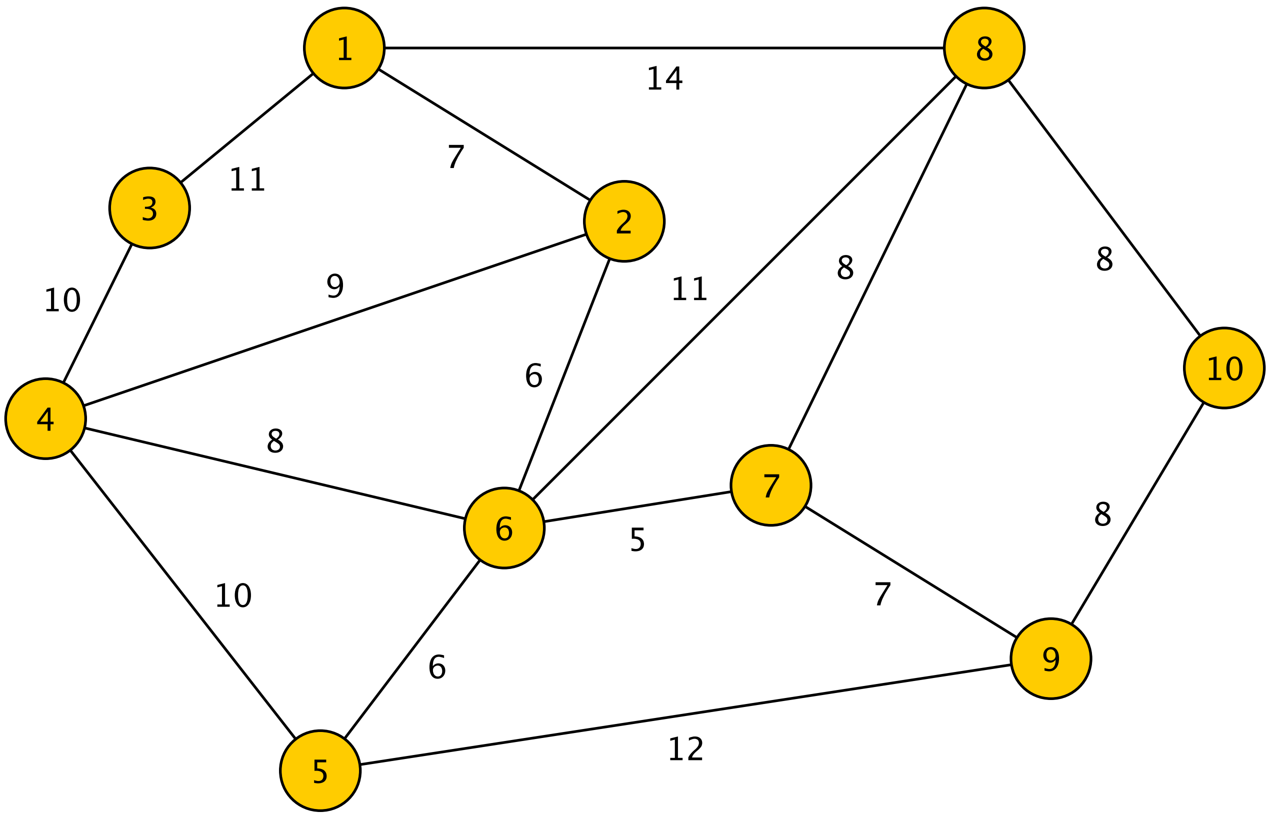
Proof idea:

- If G is not a tree, then it contains a cycle C
- Removing an edge from C leaves G connected (why)
- Repeat until no more cycles remain

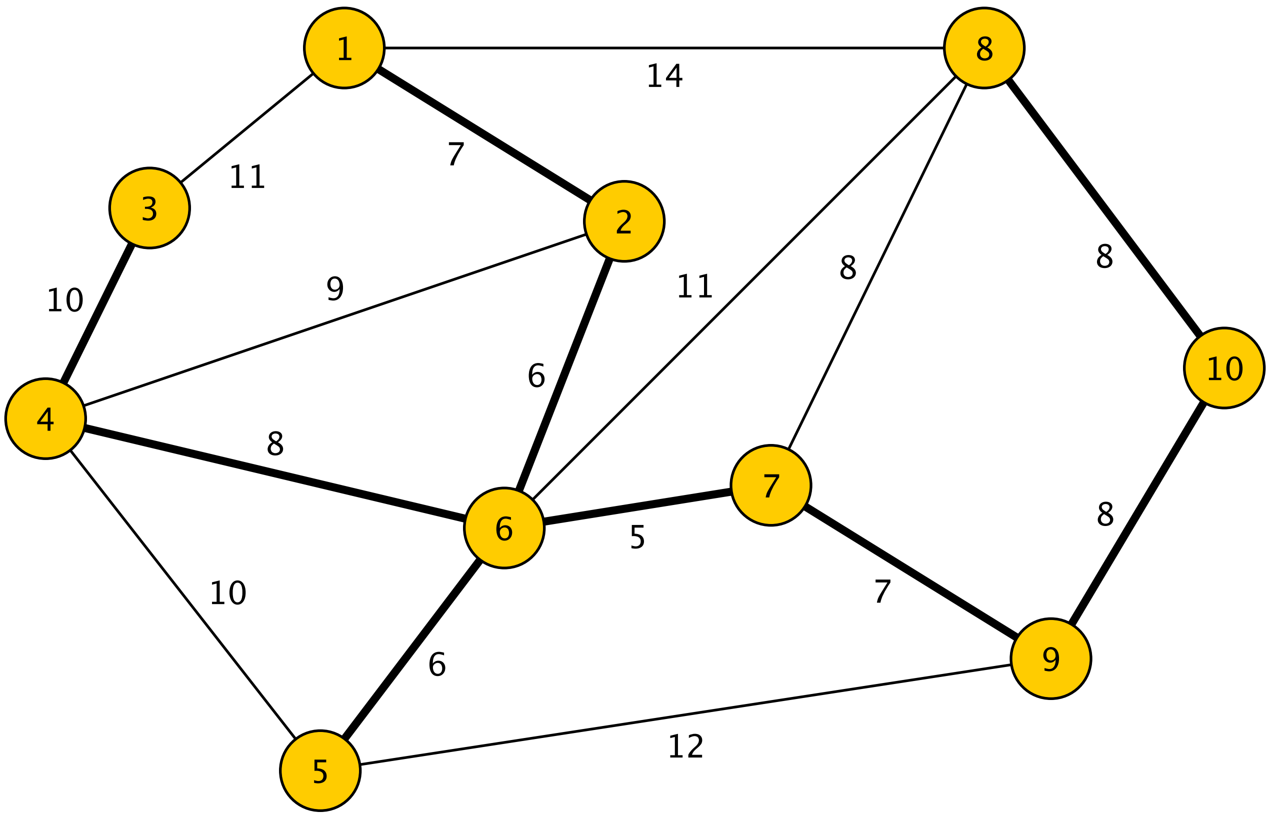
A Famous Problem

- Given a connected, undirected graph G=(V,E) with non-negative edge weights, find a minimum-weight, connected, spanning subgraph of G.
- Note: Such a subgraph must be a spanning tree!
- Frequently, we refer to the edge weights as *costs* and so this problem becomes:
- Given an undirected graph G with edge costs, compute a minimum-cost spanning tree of G.

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees



Finding a MCST

Suppose we just wanted to find a PCST (pretty cheap spanning tree), here's one idea:

Grow It Greedily!

- Pick a vertex and find its cheapest incident edge. Now we have a (small) tree
- Repeatedly add the cheapest edge to the tree that keeps it a tree (connected, no cycles)
- This method is called *Prim's Algorithm*
- How close might this get us to the MCST?

An Amazing Fact

Thm: (Prim 1957) The greedy tree-growing algorithm always finds a minimum-cost spanning tree for any connected graph.

Contrast this with the greedy exam scheduling algorithm, which does *not* always find a minimum schedule (coloring)

Why does this work?

The Key

Def: Sets V_1 and V_2 form a *partition* of a set V if

$$V_1 \cup V_2 = V \text{ and } V_1 \cap V_2 = \emptyset$$

Lemma: Let G=(V,E) be a connected graph and let V_1 and V_2 be a partition of V. *Every* MCST of G contains a cheapest edge between V_1 and V_2

- Let e be a cheapest edge between V_1 and V_2
- Let T be a MCST of G. If $e \notin T$, then $T \cup \{e\}$ contains a cycle C and e is an edge of C
- Some other edge e' of C must also be between V_1 and V_2; e is a cheapest edge, so w(e') = w(e) [Why?]

Using The Key to Prove Prim

We'll assume all edge costs are distinct

Otherwise proof is slightly less elegant

Let T be the tree produced by the greedy algorithm and suppose T* is a MCST for G

Claim: T = T*

Idea of Proof: Show that every edge added to the tree T by the greedy algorithm is in T*

Clearly the first edge added to T is in T*

Why? Use the key!

Using The Key

Now use induction!

- Suppose, for some $k \geq 1$, that the first k edges added to T are in T*. These form a tree T_k
- Let V_1 be the vertices of T_k and let $V_2 = V\text{-}V_1$
- Now, the greedy algorithm will add to T the cheapest edge e between V_1 and V_2
- But any MCST contains the (only!) cheapest edge between V_1 and V_2, so e is in T*
- Thus the first k+1 edges of T are in T*

Prim's Algorithm

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V_1 - \{v\}$

let A be the set of all edges between V_1 *and* V_2

while$(|V_1| < |V|)$

    let e $\leftarrow$ *cheapest edge in A between* V_1 *and* V_2

    add e to MCST

    let u $\leftarrow$ *the vertex of e in* V_2

    move u from V_2 *to* V_1*;*

    add to A all edges incident to u

    // note: A now may have edges with both ends in V_1

Prim's Algorithm (Variant)

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V_1 - \{v\}$

let A $\leftarrow \emptyset$ // *A will contain ALL edges between* V_1 *and* V_2

while $|V_1| < |V|$

    add to A all edges incident to v

    repeat

        remove cheapest edge e from A

    until e is an edge between V_1 *and* V_2

    add e to MCST

    let v $\leftarrow$ *the vertex of e in* V_2

    move v from V_2 *to* V_1;

Prim's Algorithm (Variant)

- Note: If G is not connected, A will eventually be empty even though $|V_1| < |V|$
- We fix this by
  - Replacing *while*$(|V_1| < |V|)$ with *while* $(|V_1| < |V|)$ && $A \neq \emptyset$)
  - Replacing *until* e *is an edge between* V_1 *and* V_2 with
    - *until* $A=\emptyset$ *or* e *is an edge between* V_1 *and* V_2
- Then Prim will find the MCST for the component containing v

Prim's Algorithm (Variant)

```
prim(G) // finds a MCST of connected G=(V,E)
let v be a vertex of G; set V_1 ← {v} and V_2 ← V_1 - {v}
let A ← ∅        // A will contain ALL edges between V_1 and V_2
while |V_1| < |V| && |A| > 0
        add to A all edges incident to v
        repeat
                remove cheapest edge e from A
        until A is empty || e is an edge between V_1 and V_2
        if e is an edge between V_1 and V_2
                let v ← the vertex of e in V_2
                move v from V_2 to V_1;
```

Implementing Prim's Algorithm

- We'll "build" the MCST by marking its edges as "visited" in G
- We'll "build" V_1 by marking its vertices visited
- How should we represent A?
  - What operations are important to A?
    - Add edges
    - Remove cheapest edge
  - A priority queue!
- When we remove an edge from A, check to ensure it has one end in each of V_1 and V_2

ComparableEdge Class

- Values in a PriorityQueue need to implement Comparable
- We wrap edges of the PQ in a class called ComparableEdge
  - It requires the label used by graph edges to be of a Comparable type

Prim's Algorithm (Variant)

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V_1 - \{v\}$

let A $\leftarrow \emptyset$ *// A will contain ALL edges between* V_1 *and* V_2

while $|V_1| < |V|$ && $|A| > 0$

    add to A all edges incident to v

    repeat

        remove cheapest edge e from A

    until A is empty || e is an edge between V_1 *and* V_2

    if e is an edge between V_1 *and* V_2

        let v $\leftarrow$ *the vertex of e in* V_2

        move v from V_2 *to* V_1*;*

MCST: The Code

```
PriorityQueue<ComparableEdge<String,Integer>> q =
  new SkewHeap<ComparableEdge<String,Integer>>();

String v = null;           // current vertex
Edge<String,Integer> e; // current edge
boolean searching;         // still building tree
g.reset();                 // clear visited flags

// select a node from the graph, if any
Iterator<String> vi = g.iterator();
if (!vi.hasNext()) return;
v = vi.next();
```

MCST: The Code

```
do {
     // visit the vertex and add all outgoing edges
     to the priority queue
     g.visit(v);
     Iterator<String> ai = g.neighbors(v);
     while (ai.hasNext()) {
           // turn it into outgoing edge
           e = g.getEdge(v,ai.next());
           // add the edge to the queue
           q.add(new
             ComparableEdge<String,Integer>(e));
     }
     ...
```

MCST: The Code

```
      searching = true;
      while (searching && !q.isEmpty()) {
            // grab next shortest edge
            e = q.remove();
            // Is e between V1 and V2 (subtle code!!)
            v = e.there(); // does e connect V1 to V2?
            if (g.isVisited(v)) v = e.here();
            if (!g.isVisited(v)) {
                  searching = false;
                  g.visitEdge(g.getEdge(e.here(),
                        e.there()));
            }
      }
} while (!searching);
```

Prim : Space Complexity

- Graph: $O(|V| + |E|)$
  - Each vertex and edge uses a constant amount of space
- Priority Queue $O(|E|)$
  - Each edge takes up constant amount of space
- Every other object (including the neighbor iterator) uses a constant amount of space
- Result: $O(|V| + |E|)$
  - Optimal in Big-O sense!

Prim : Time Complexity

Assume Map ops are O(1) time (not quite true!)

For each iteration of do ... while loop

- Add neighbors to queue: O(deg(v) log |E|)
  - Iterator operations are O(1) [Why?]
  - Adding an edge to the queue is O(log |E|)
- Find next edge: O(# edges checked * log |E|)
  - Removing an edge from queue is O(log |E|) time
  - All other operations are O(1) time

Prim : Time Complexity

Over *all* iterations of do ... while loop

Step 1: Add neighbors to queue:

- For each vertex, it's O(deg(v) log |E|) time
- Adding over all vertices gives

$$\sum_{v \in V} \deg(v) \log |E| = \log |E| \sum_{v \in V} \deg(v) = \log |E| * 2 |E|$$

- which is O(|E| log |E|) = O(|E| log |V|)
  - $|E| \leq |V|^2$, so $\log |E| \leq \log |V|^2 = 2 \log |V| = O(\log |V|)$

Prim : Time Complexity

Over *all* iterations of do … while loop

Step 2: Find next edge: O(# edges checked * log |E|)

- Each edge is checked at most once
- Adding over all edges gives O(|E| log |E|) again

Thus, overall time complexity (worst case) of Prim's Algorithm is O(|E| log |V|)

- Typically written as O(m log n)
  - Where m= |E| and n = |V|