CSCI 136
Data Structures & Advanced Programming

Lecture 32

Fall 2017

Instructors: Bills

Last Time

- Adjacency List Implementation Details
  - Featuring many Iterators!
- More Fundamental Graph Properties
- An Important Algorithm: Minimum-cost spanning subgraph

Today's Outline

- Finish up Prim's Algorithm
- More Core Algorithms: Directed Graphs
  - Dijkstra's Algorithm
  - Time permitting
    - Cycle Detection
    - Topological Sorting

Recall: 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)
- How close might this get us to the MCST?

Recall: 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 coloring

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 ← cheapest edge in A between $V_1$ and $V_2$
        add e to MCST
        let u ← 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)

- 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* $A \neq \emptyset$ *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 $V_1 \leftarrow \{v\}$ and $V_2 \leftarrow V_1 - \{v\}$
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
     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 $V_1$ and $V_2$ (subtle code!!)
            v = e.there();
            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| ≤ 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|

Single Source Shortest Paths

The Problem: Given a graph G and a starting vertex v, find, for *each* vertex u≠v reachable from v, a shortest path from v to u.

- The Single Source Shortest Paths Problem
- Arises in many contexts, including network communications
- Uses edge weights (but we'll call them "lengths"): assume they are non-negative numbers
- Could be a directed or undirected graph

Single Source Shortest Paths

- We'll look at directed graphs
  - So the paths must be directed paths
- Let's think....
- Suppose we have a set shortest paths $\{P_u : u \neq v\}$, where P_u is a shortest path from v to u
- Let H be the subgraph of G consisting of each vertex of G along with all of the edges in each P_u
- What can we say about H?

Single Source Shortest Paths

Observations

- If some vertex u has in-degree greater than 1, we can drop one of the incoming edges: Why?
  - Only the last edge of the shortest path from v-u is needed as an in-edge to u [Why?]
  - So we assume H has in-deg(u)=1 for all u≠v
    - We need *no* in-edges for v [Why?]
- H can't have any directed cycles
  - Well, v can't be on any cycles (in-deg(v) = 0)
  - If there were a cycle, some vertex on it would have in-degree > 1 [Why?]

Single Source Shortest Paths

Observations

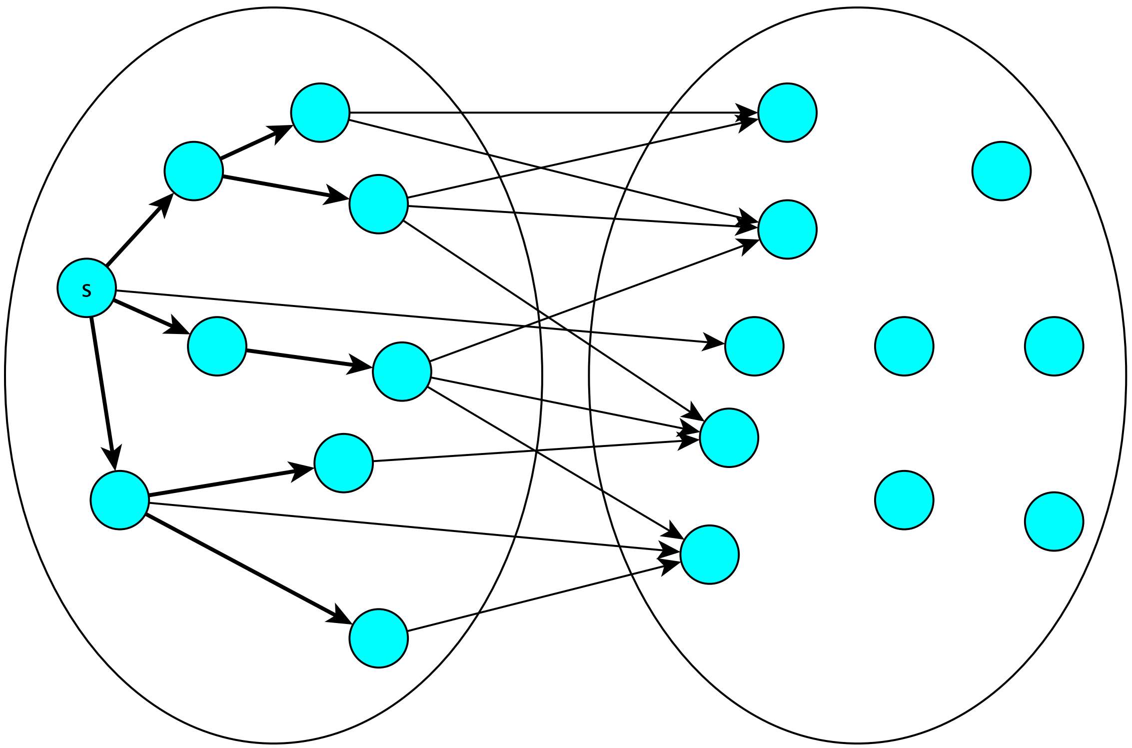
- In fact, even disregarding edge directions, there would be no cycles
  - Some vertex would have in-degree at least 2
    - Or else there's a directed cycle (Why?)
- So, we can assume that there is some set of shortest paths that forms a (directed) tree
- This suggests that we try again to

  Greedily grow a tree
- The question is: How?

The Right Kind of Greed

- Build a MCST?
  - No: It won't always give shortest paths
- 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 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?

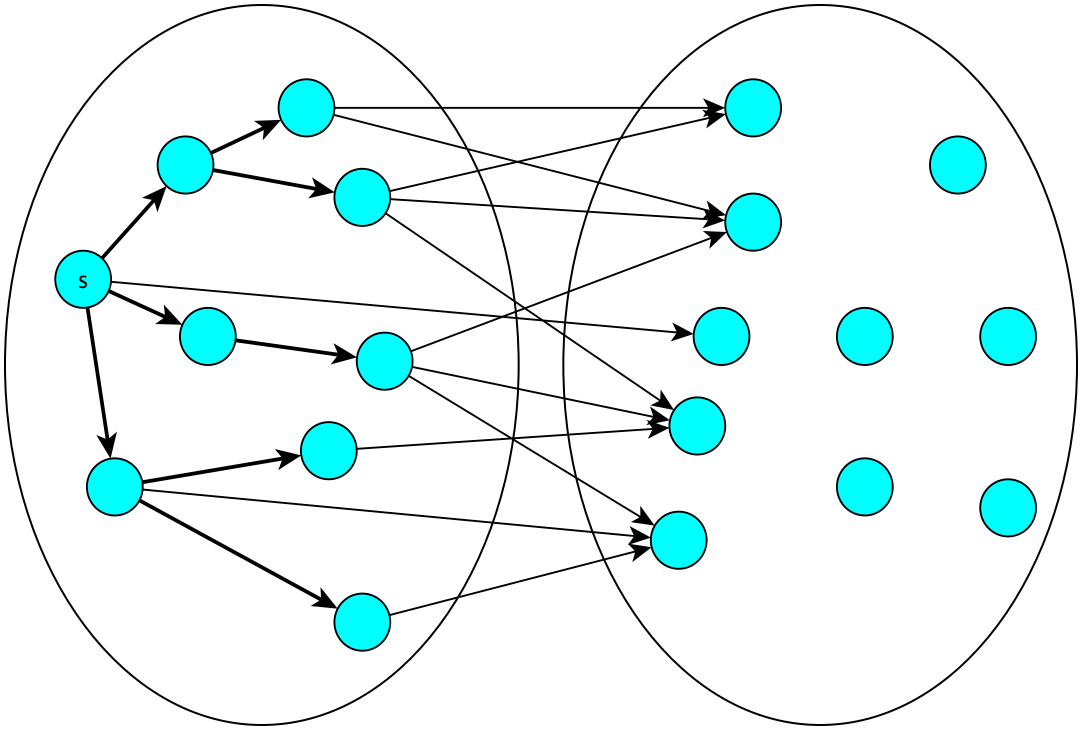
Finding the Best Vertex to Add to T_k



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! [Proof?] Repeat until k = |V|

Some Notation Reminders

- l(e) : length (weight) of edge e
- d(u,v) : *distance* from u to v
  - Length of shortest path from u to v

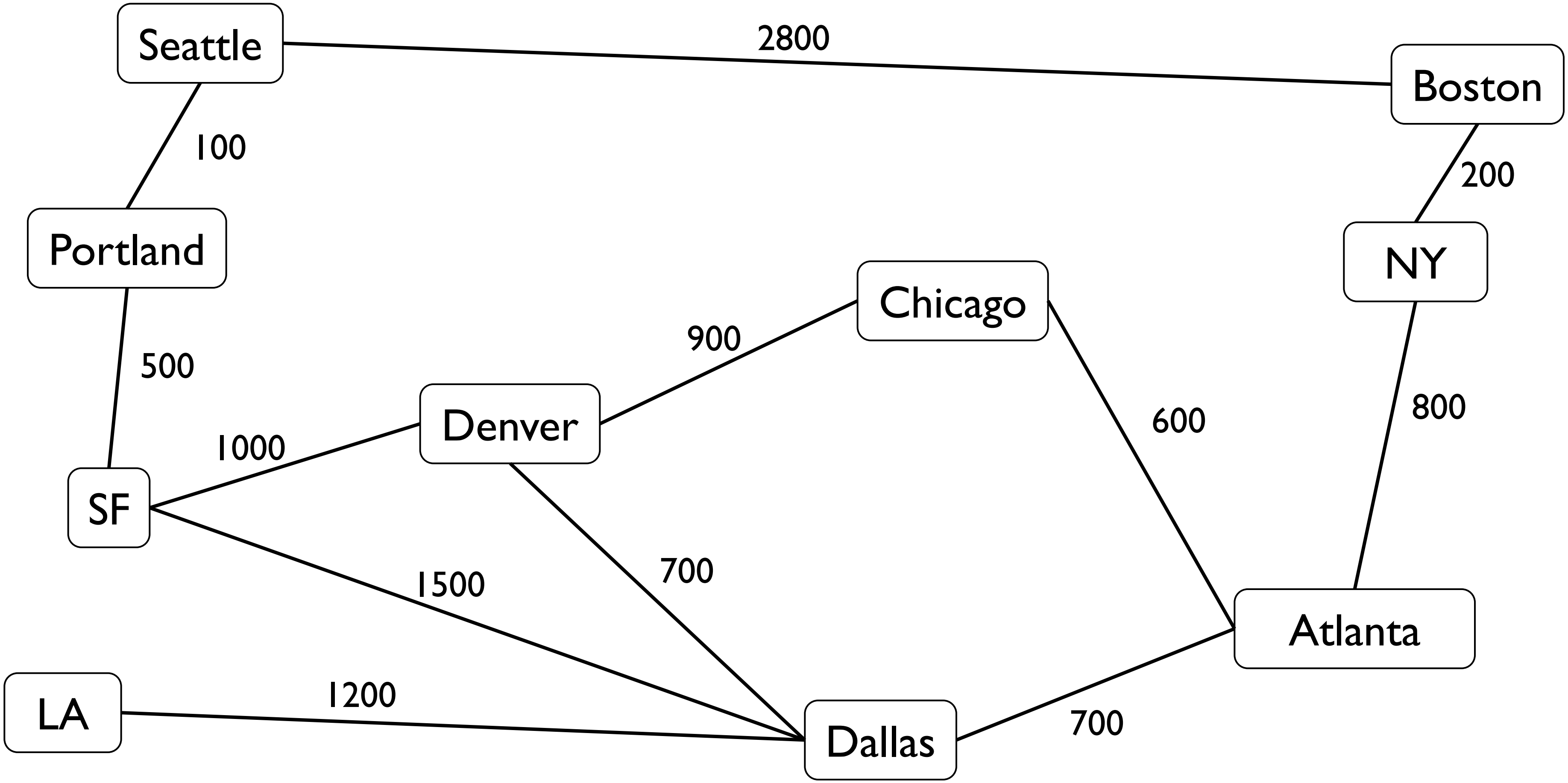
- The priority queue stores an *estimate* of the distance from s to w by storing, for some edge (v,w), d(s,v) + l(v,w)
  - The estimate is always an *upper bound* on d(s,w)

Dijkstra: What Do We Return?

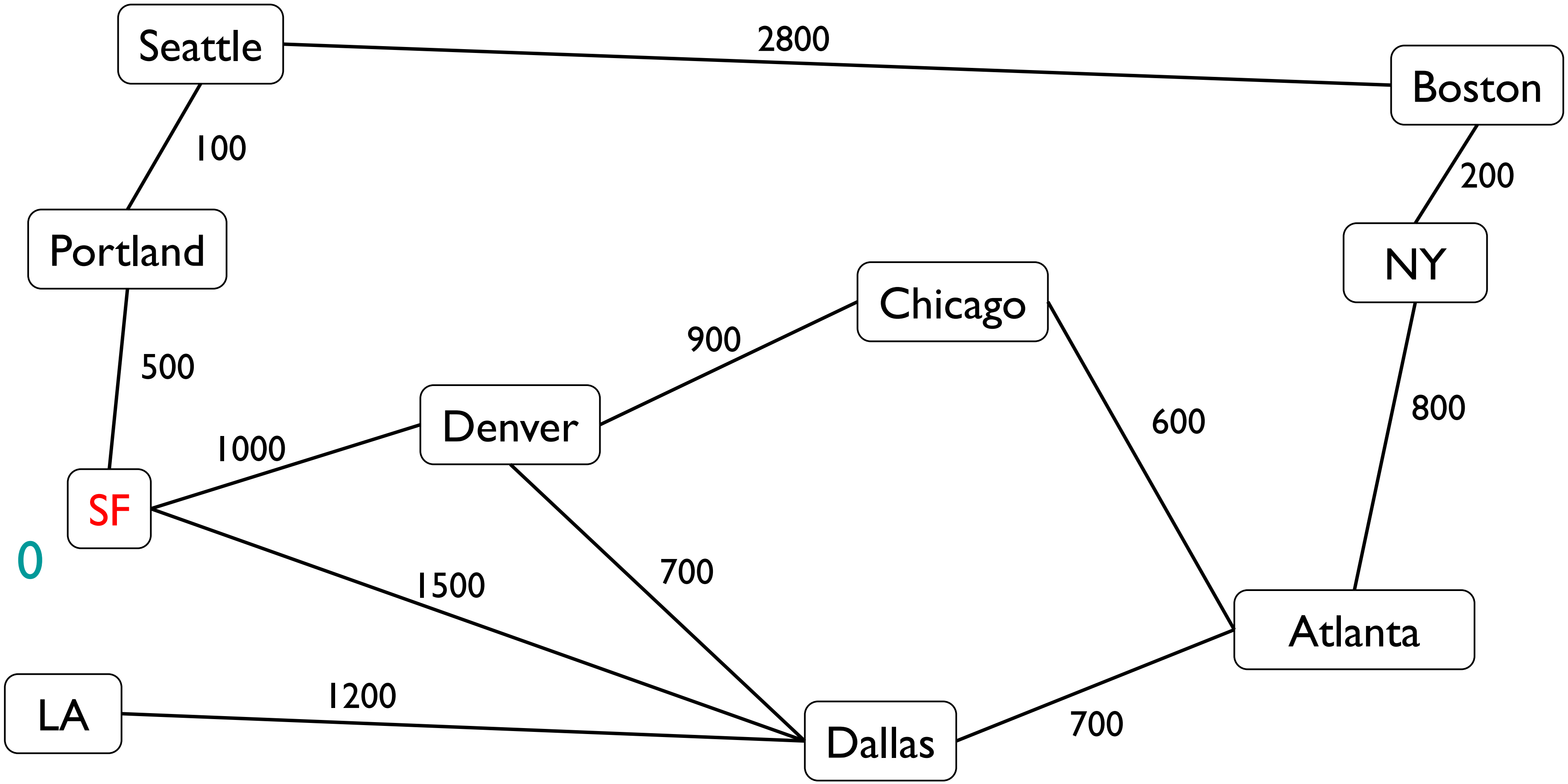
- As we find a new vertex v to add to the tree of shortest paths, add edges e=(v,w) to a map.
- Precisely:
    - Use the PQ association(X,Y) edgeInfo where
        - X is d(s,v) + l(v,w)
        - Y is the edge e=(v,w)
    - Add the key/value pair (w, edgeInfo) to the map
- So the map entry with key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

Dijkstra's Algorithm

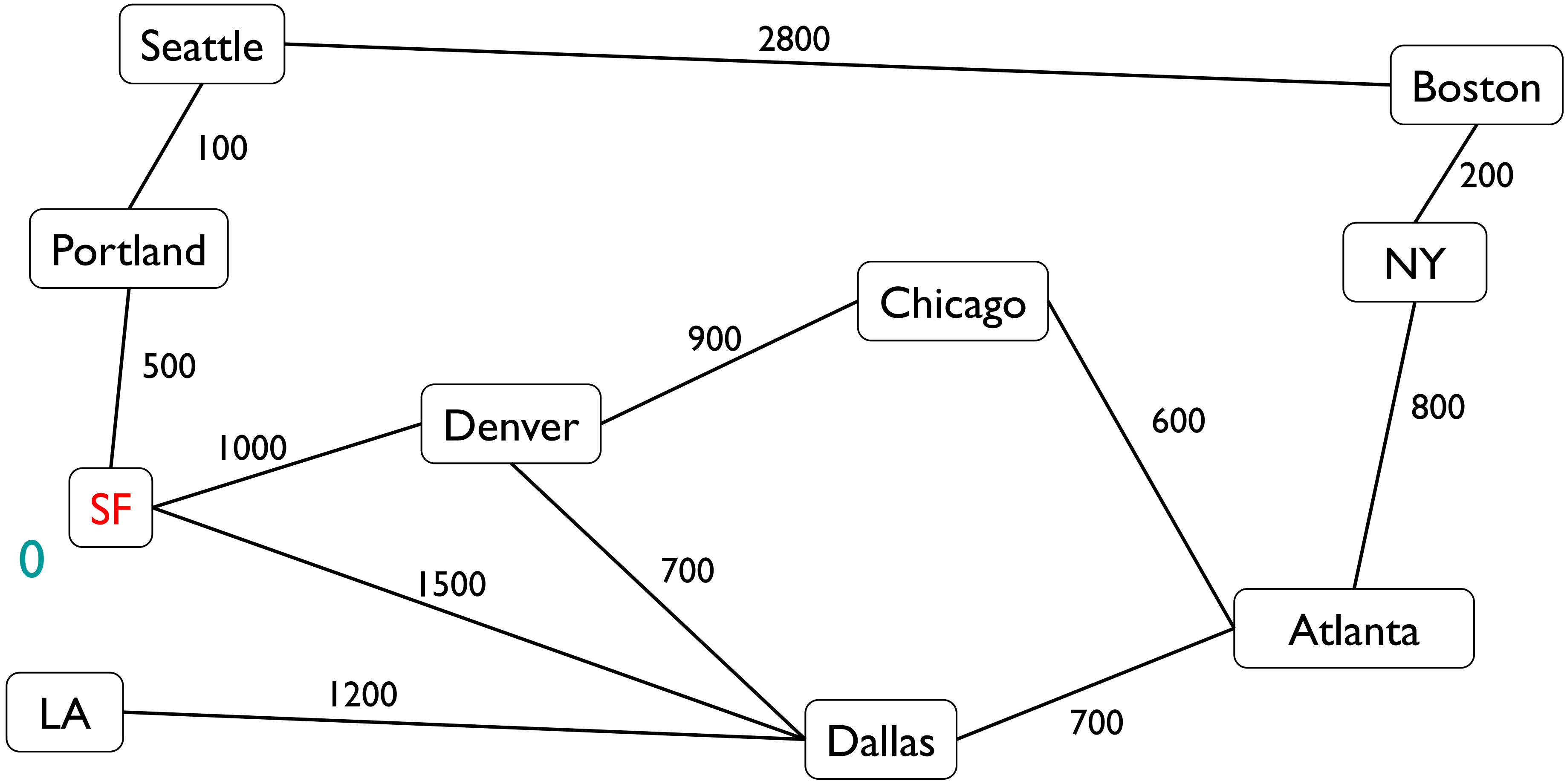
```
Dijkstra(G, s)  // l(e) is the length of edge e
let T ← ({s}, ∅)  and PQ be an empty priority queue
for each neighbor v of s, add edge (s,v) to PQ with priority l(e)
while T doesn't have all vertices of G and PQ is non-empty
    repeat
        e ← PQ.removeMin()   // skip edges with both
    until 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's Algorithm

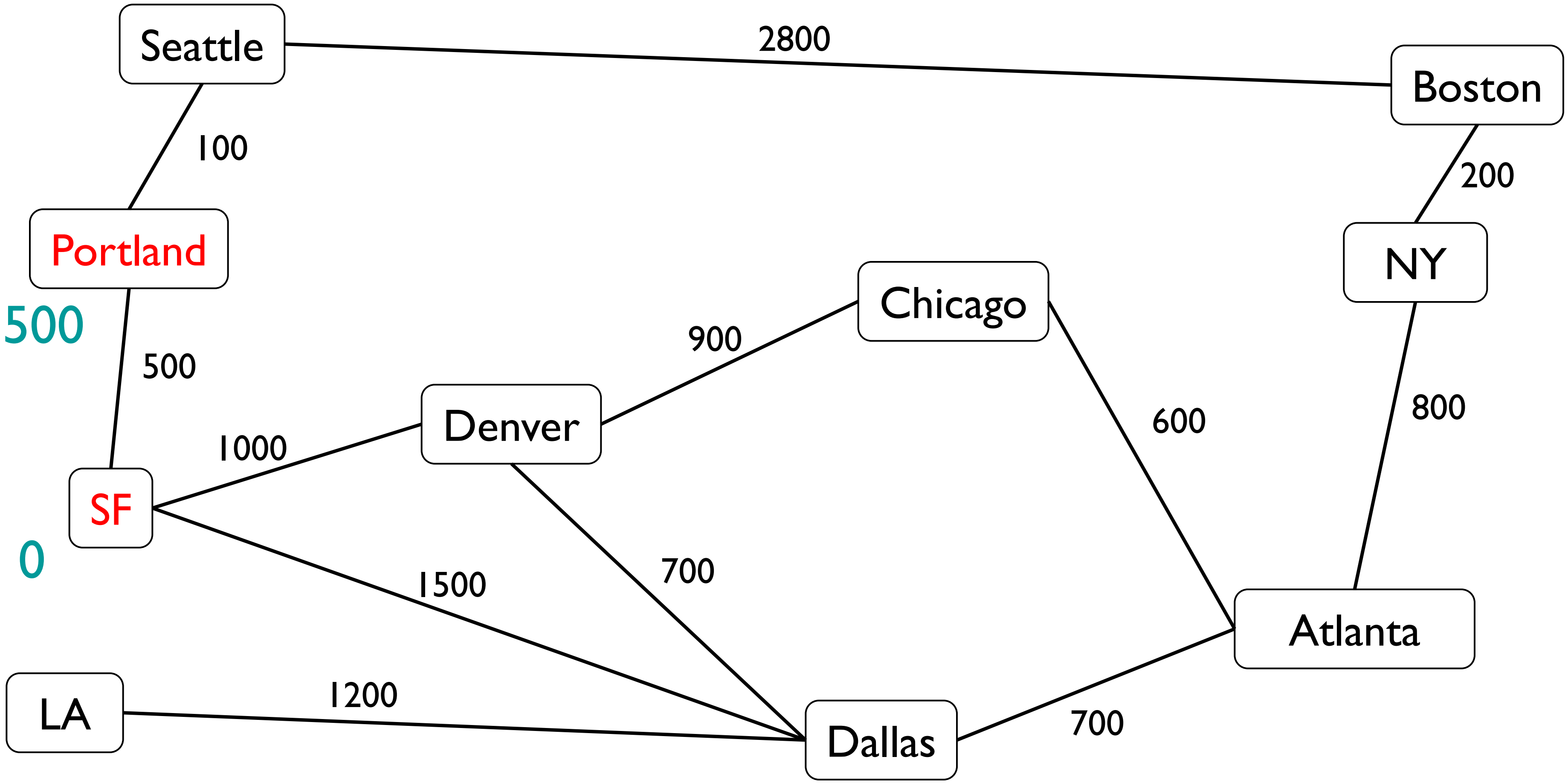


Priority Queue



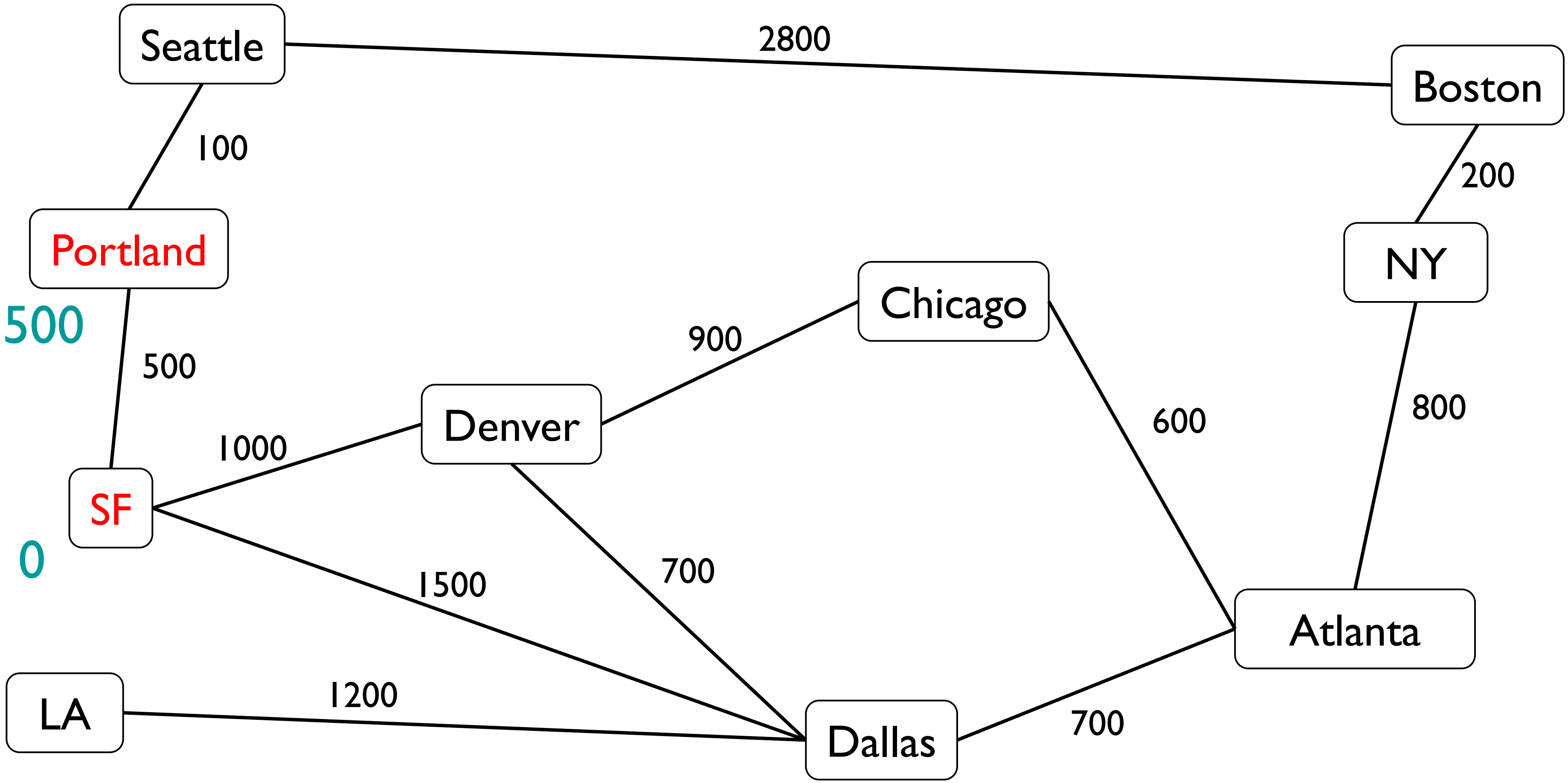
Priority Queue

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



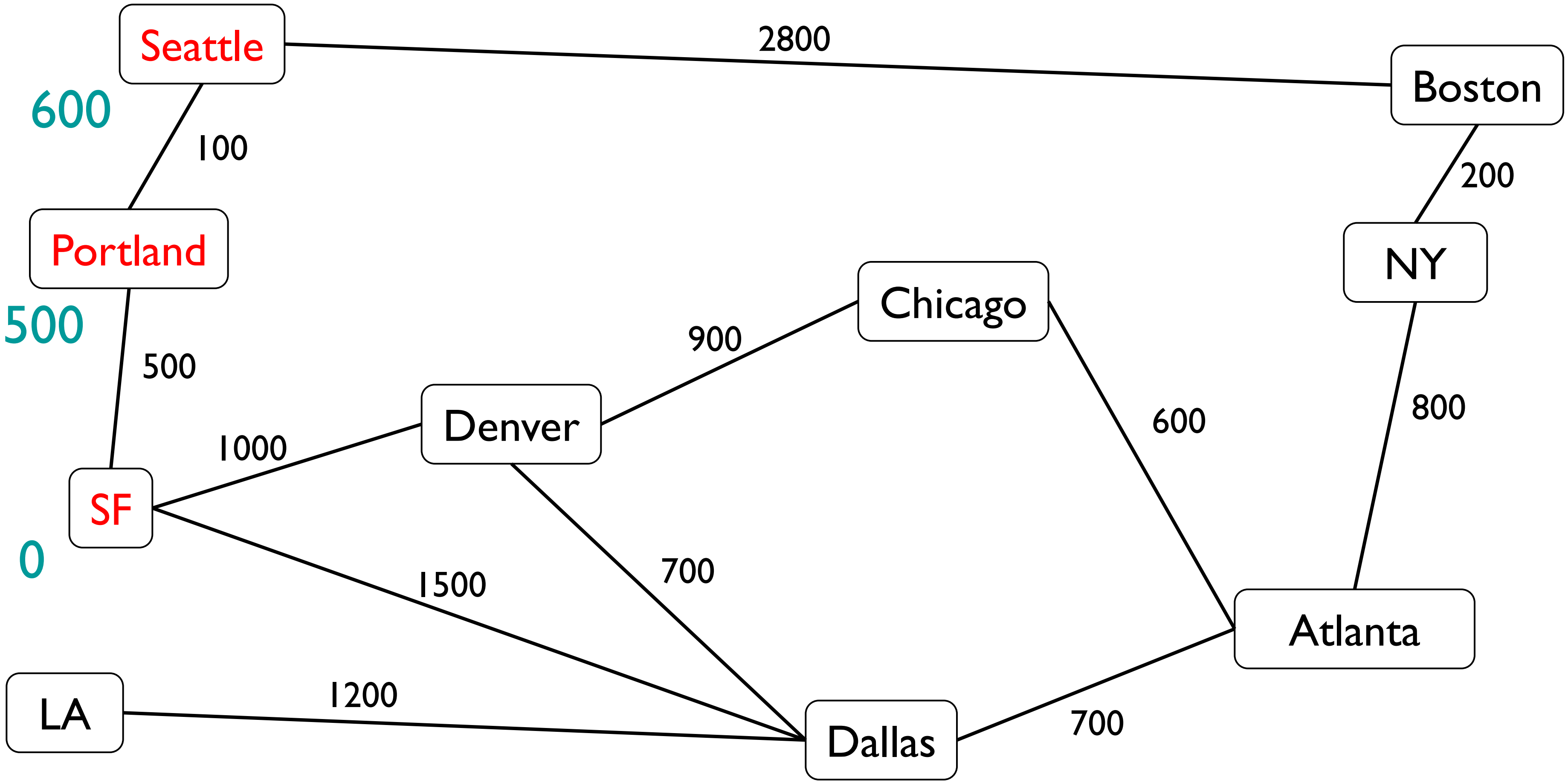
Current: 500 SF->Port (need to add Port's neighbors to PQ)

SF->Den; 1000   SF->Dal 1500



Current: 500 SF->Port

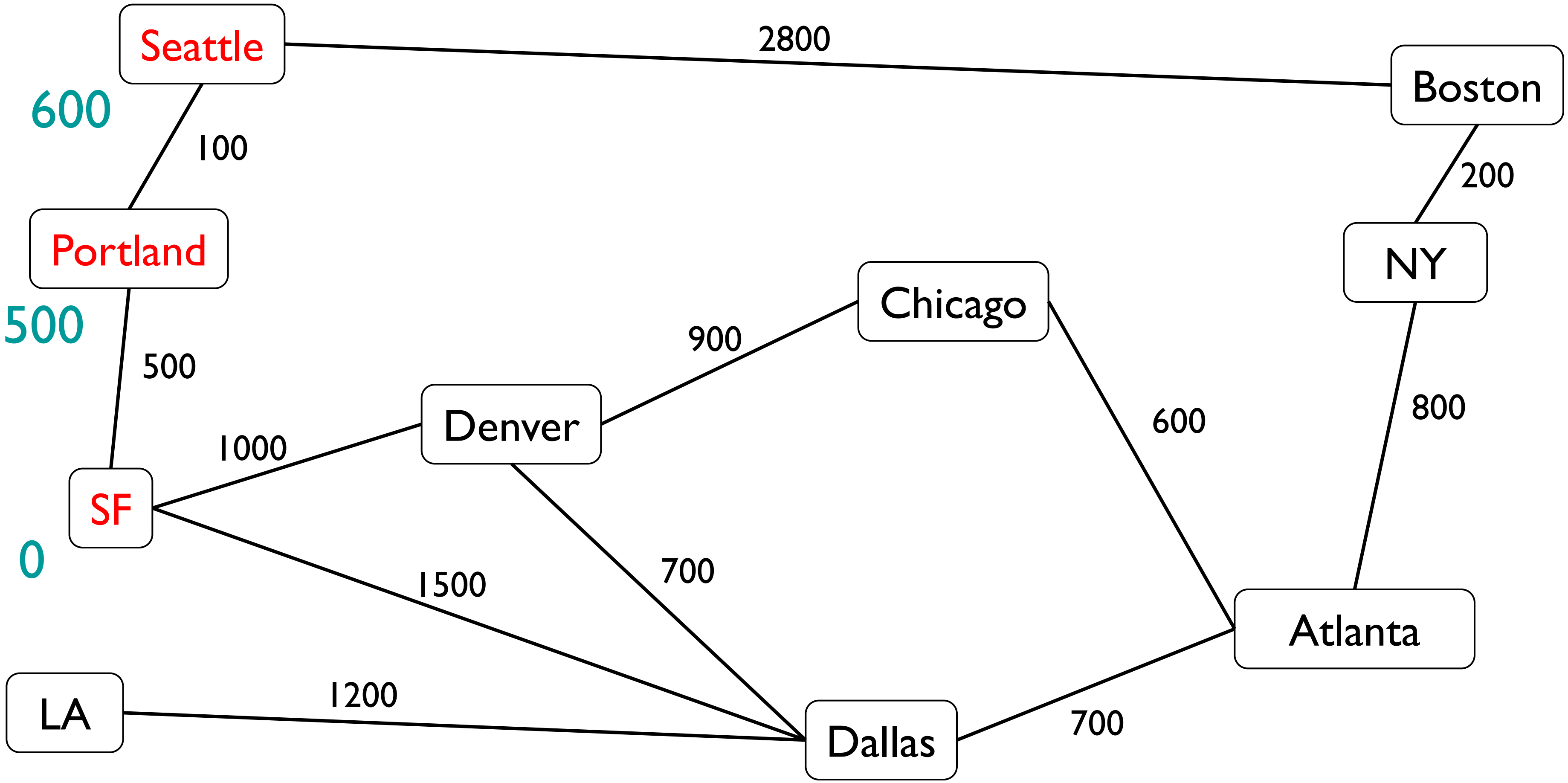
SF->Port->Sea; 600	SF->Den; 1000	SF->Dal 1500



Current: 600 SF->Port->Sea

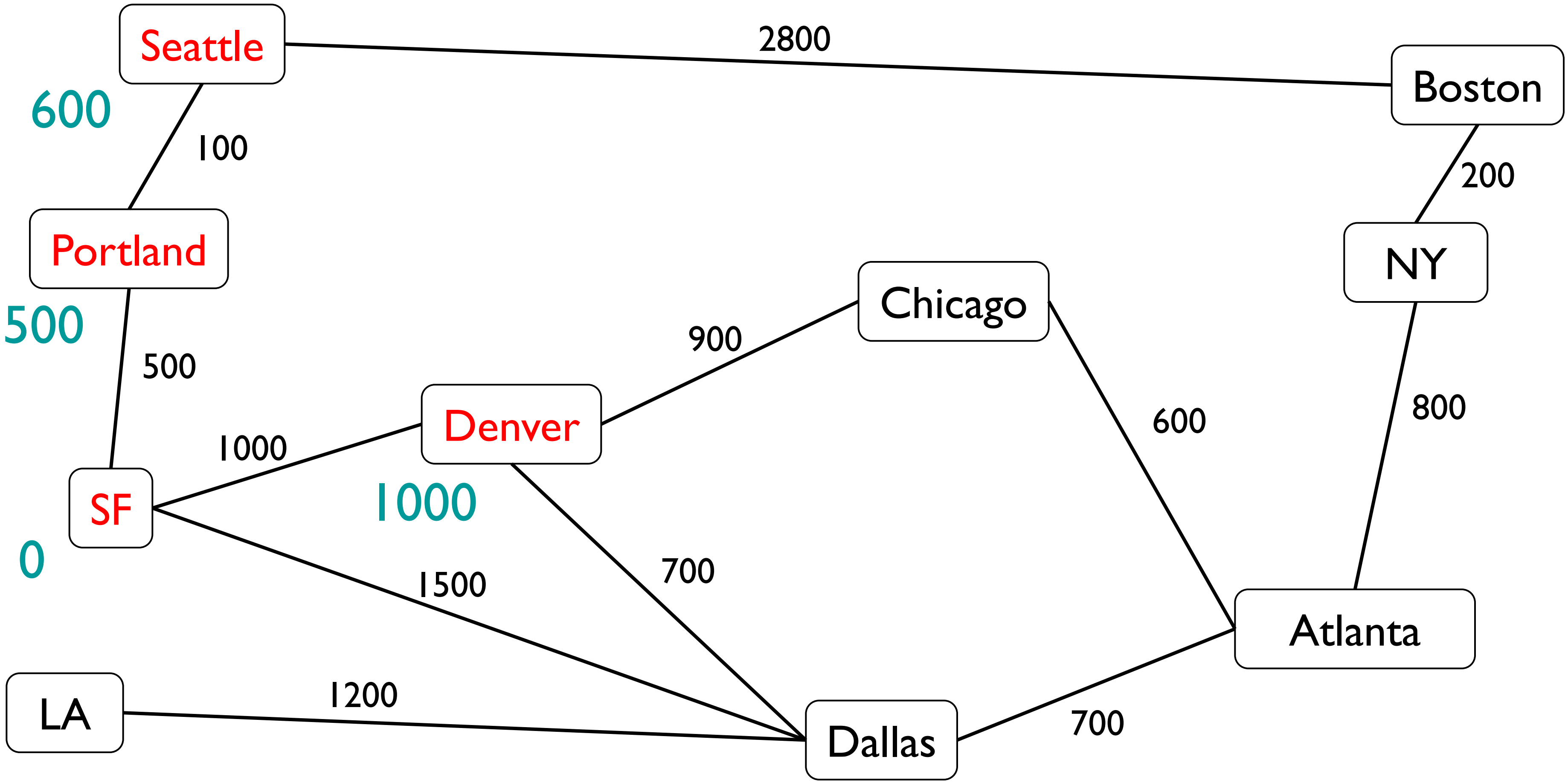
SF->Den; 1000

SF->Dal 1500



Current: 600 SF->Port->Sea

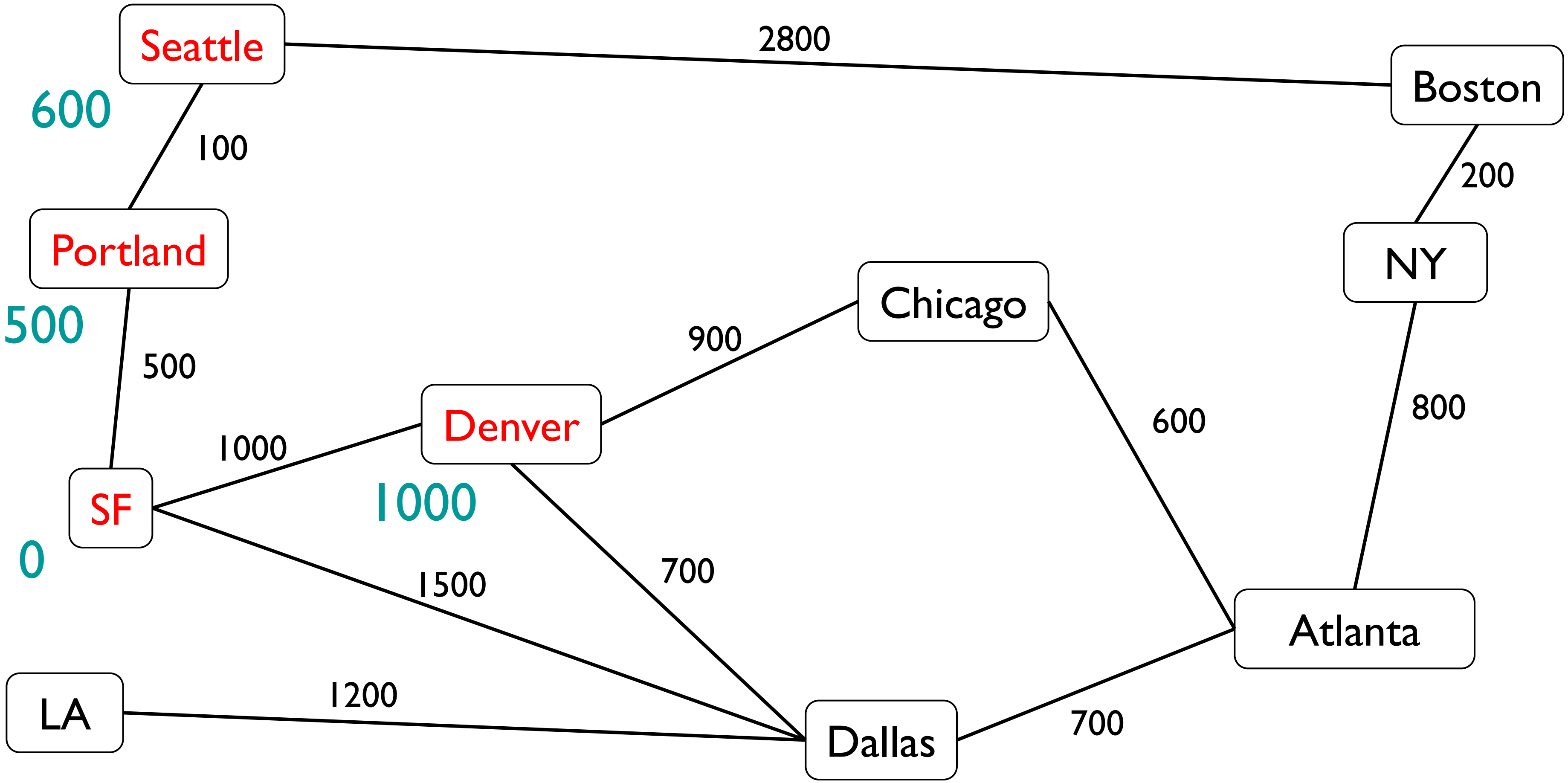
⇨ SF->Den; 1000 SF->Dal; 1500 SF->Port->Sea->Bos 3400



Current: 1000 SF->Den

➡ SF->Dal; 1500

SF->Port->Sea->Bos 3400



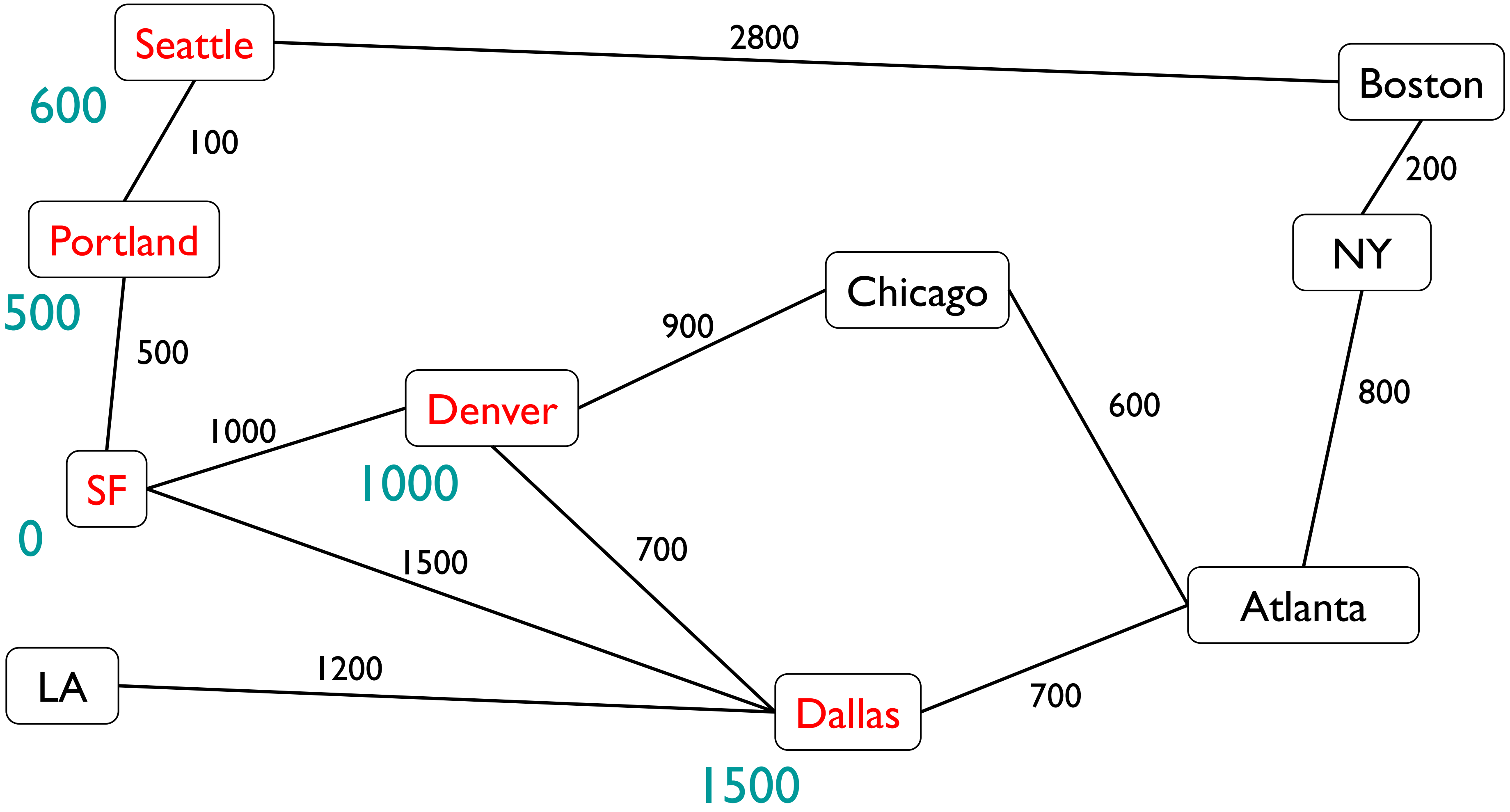
Current: 1000 SF->Den

⇨ SF->Dal; 1500

SF->Den->Dal; 1700

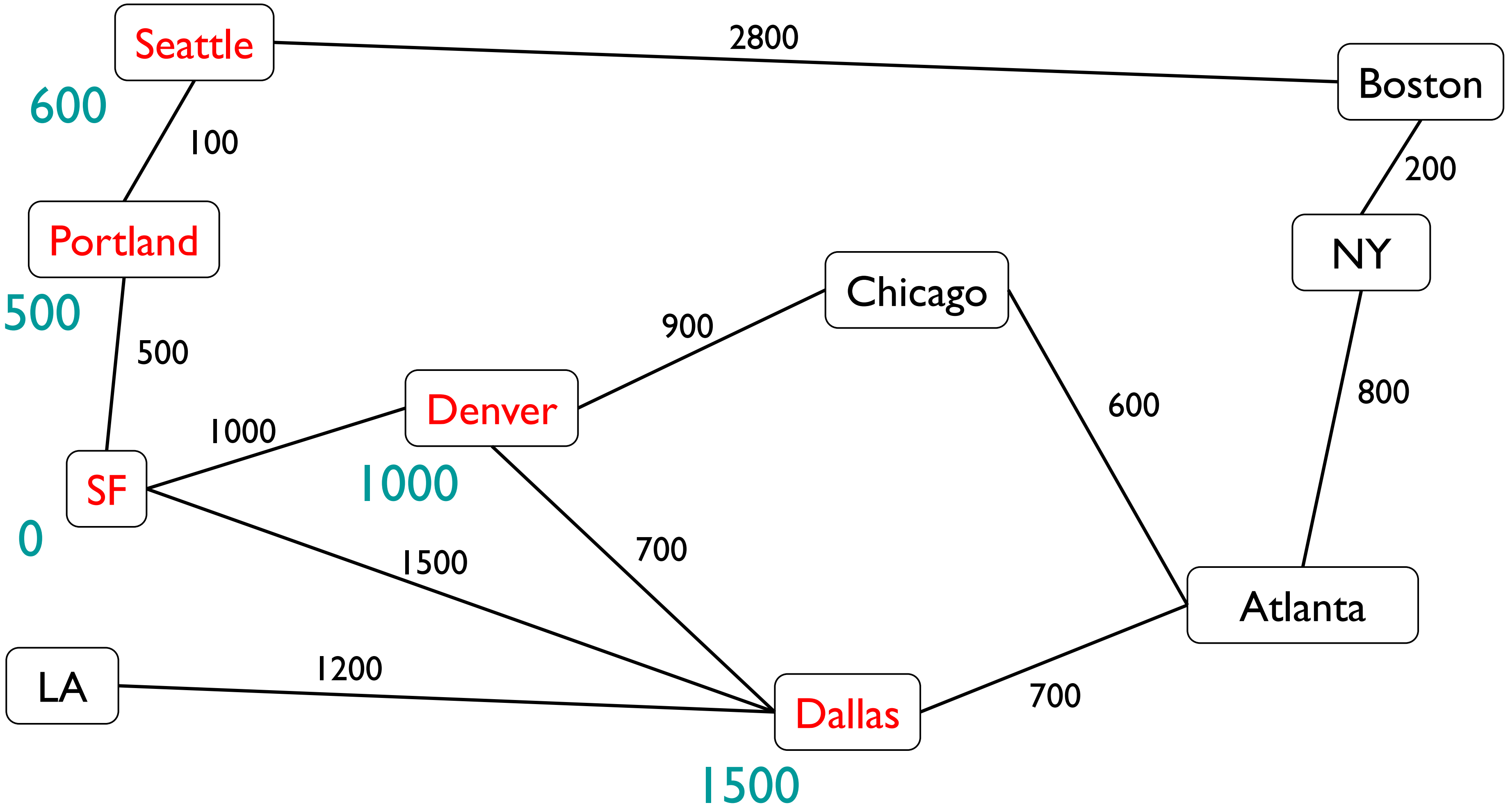
SF->Den->Chi; 1900

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

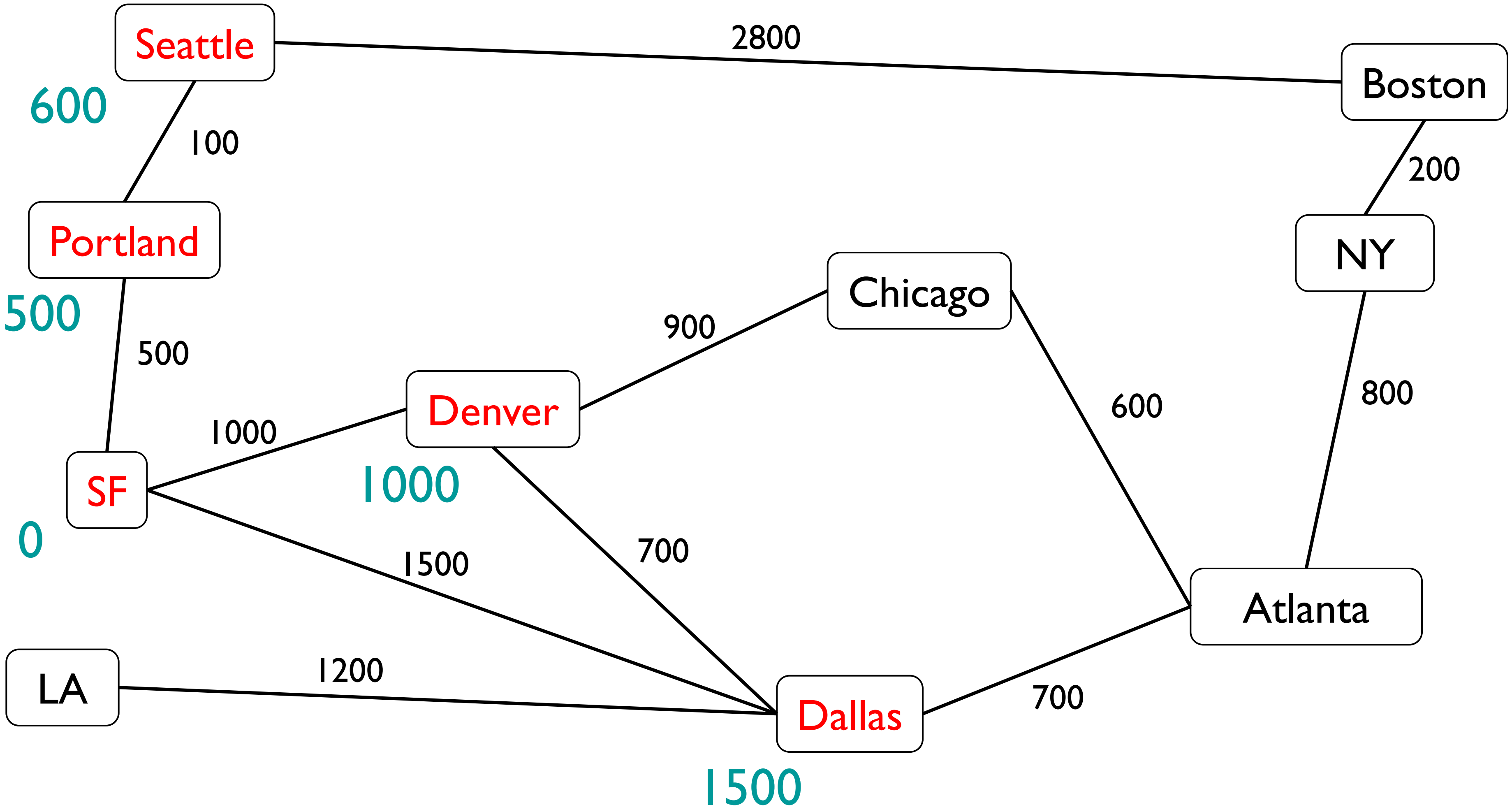
⇨ SF->Den->Dal; 1700

SF->Den->Chi; 1900

SF->Dal->Atl; 2200

SF->Dal->LA; 2700

SF->Port->Sea->Bos 3400



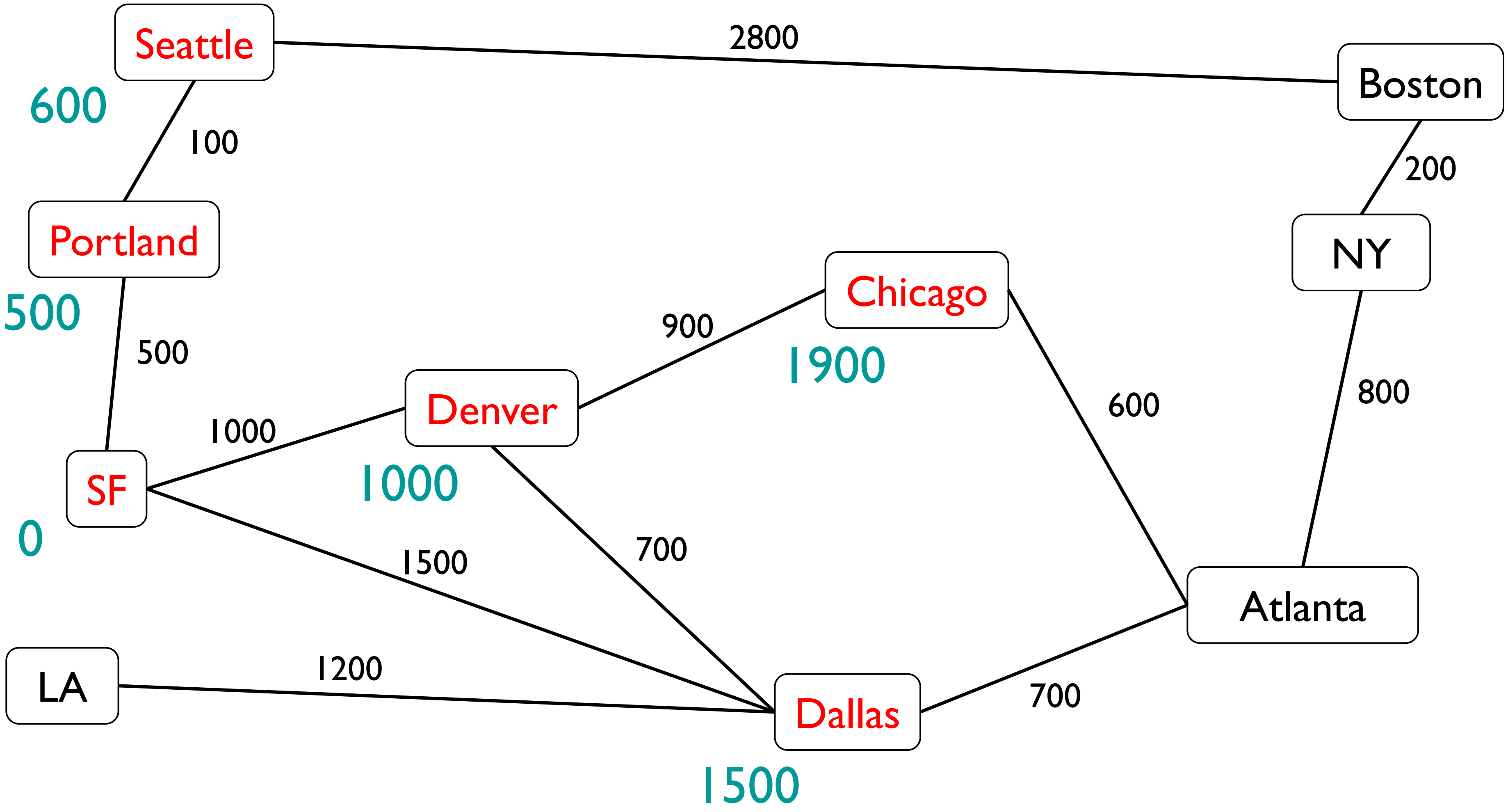
Current: 1700 SF->Den->Dal (we already have Dallas!)

SF->Den->Chi; 1900

SF->Dal->Atl; 2200

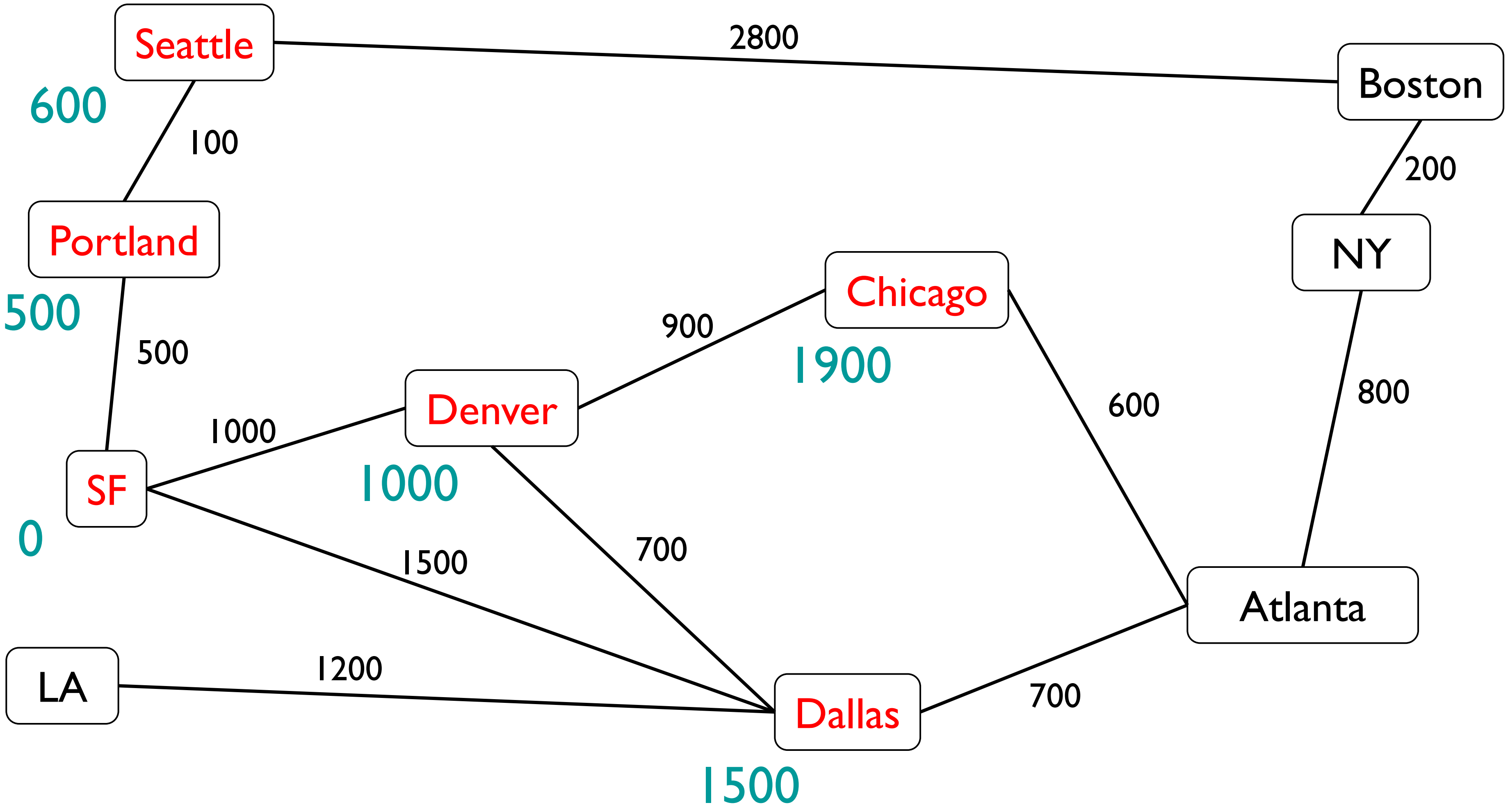
SF->Dal->LA; 2700

SF->Port->Sea->Bos 3400



Current: 1900 SF->Den->Chi

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



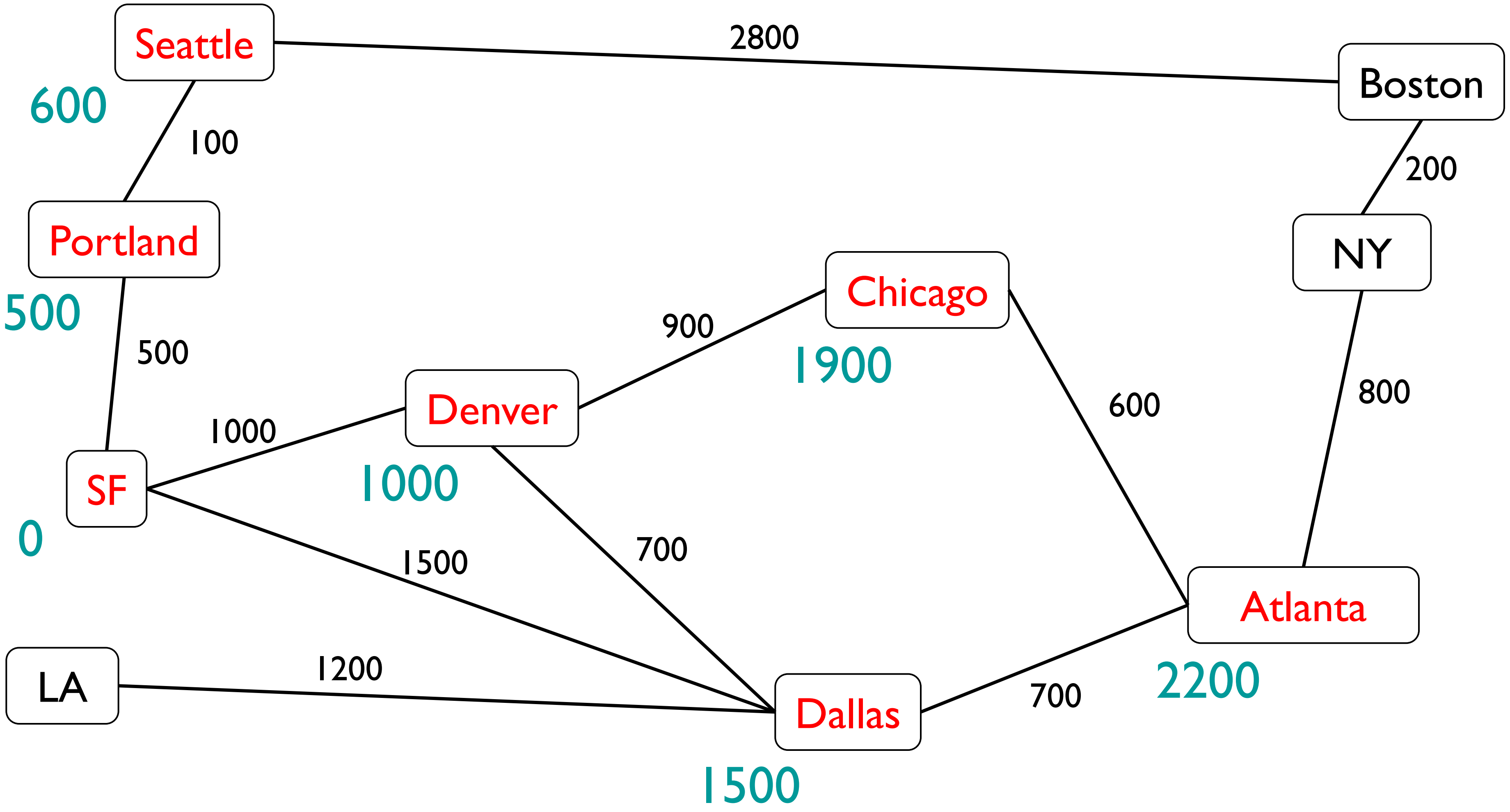
Current: 1900 SF->Den->Chi

⇨ SF->Dal->Atl; 2200

SF->Den->Chi->Atl; 2500

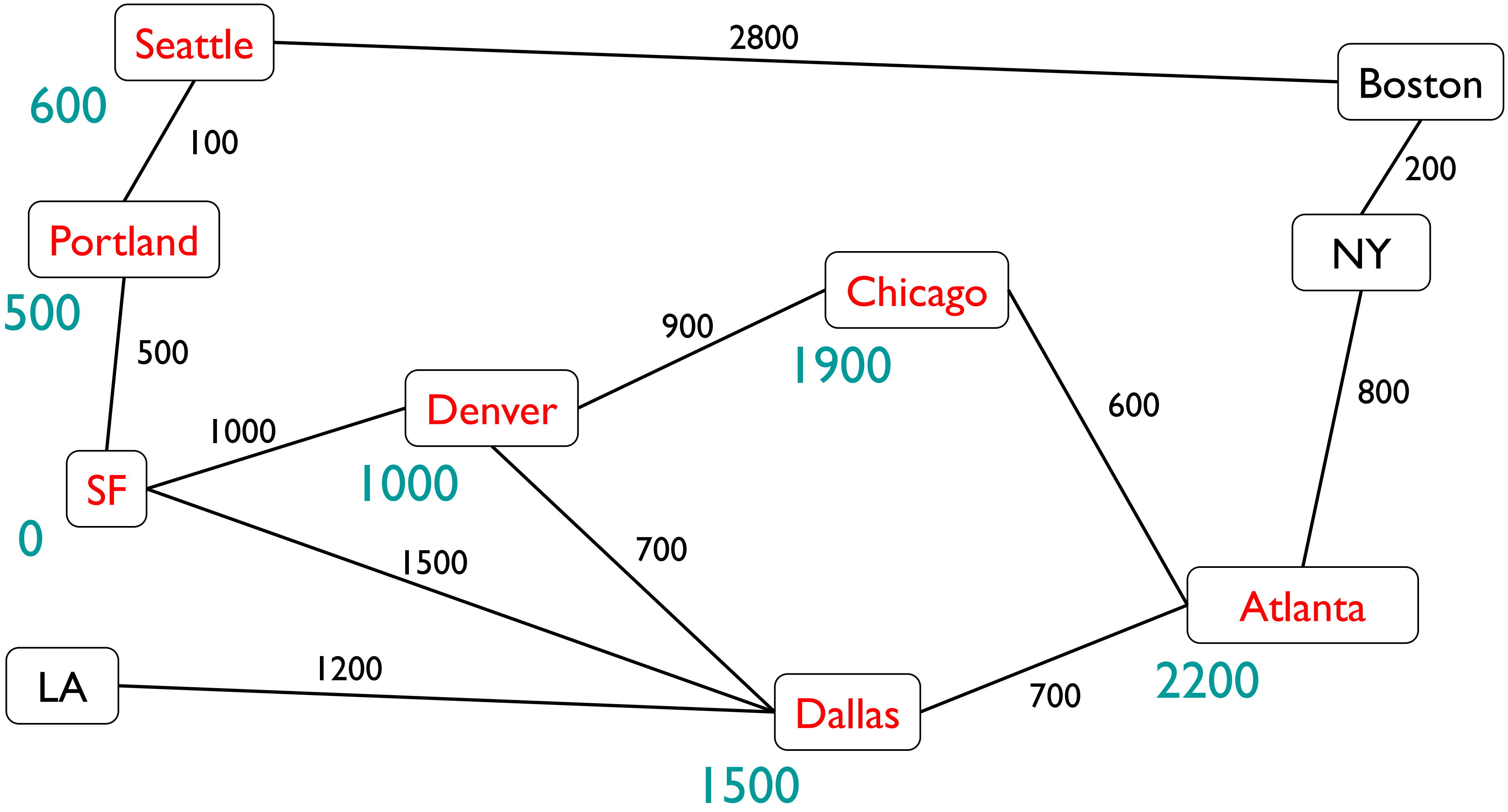
SF->Dal->LA; 2700

SF->Port->Sea->Bos 3400



Current: 2200 SF->Dal->Atl

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



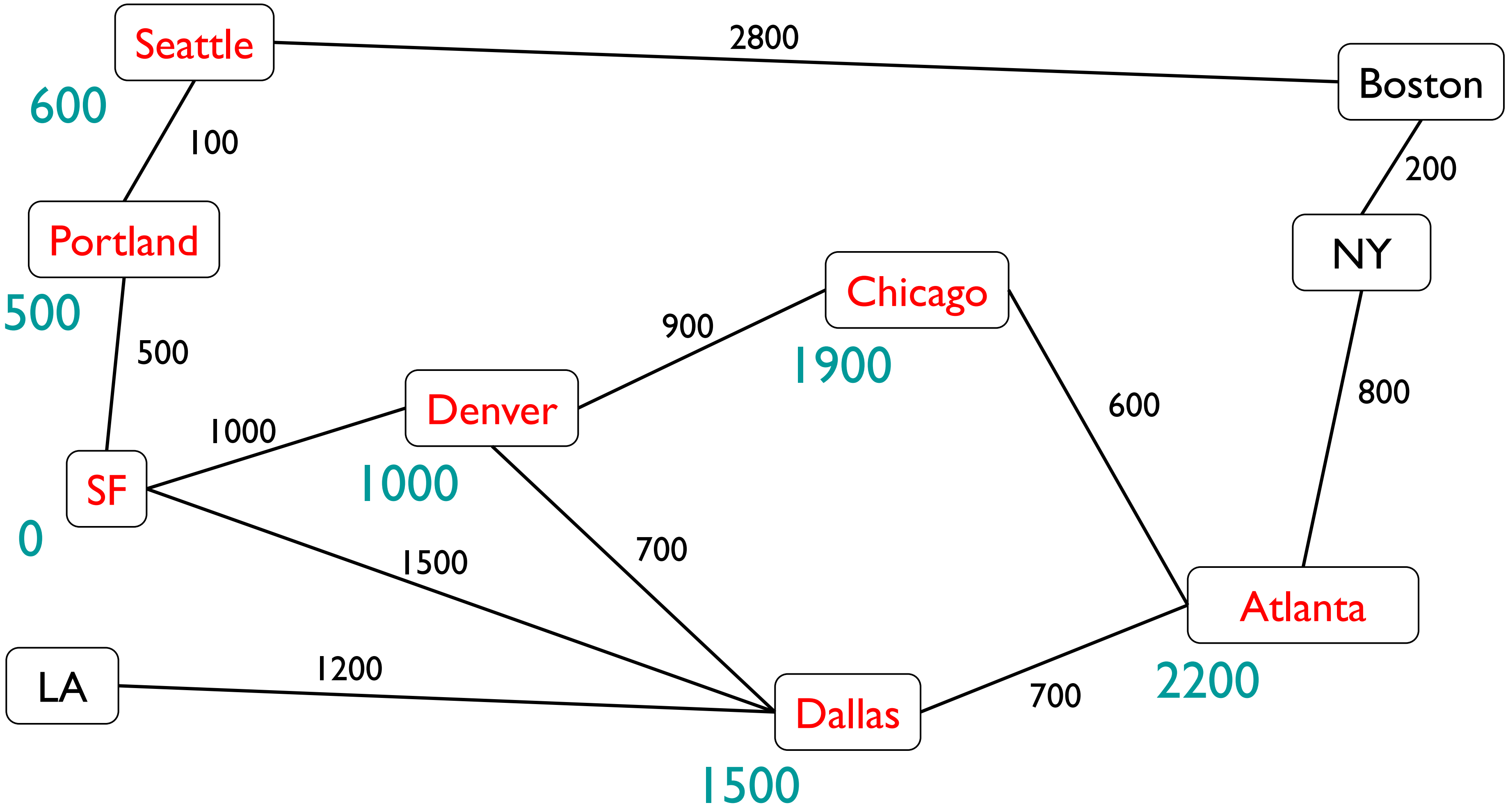
Current: 2200 SF->Dal->Atl

⇨ SF->Den->Chi->Atl; 2500

SF->Dal->LA; 2700

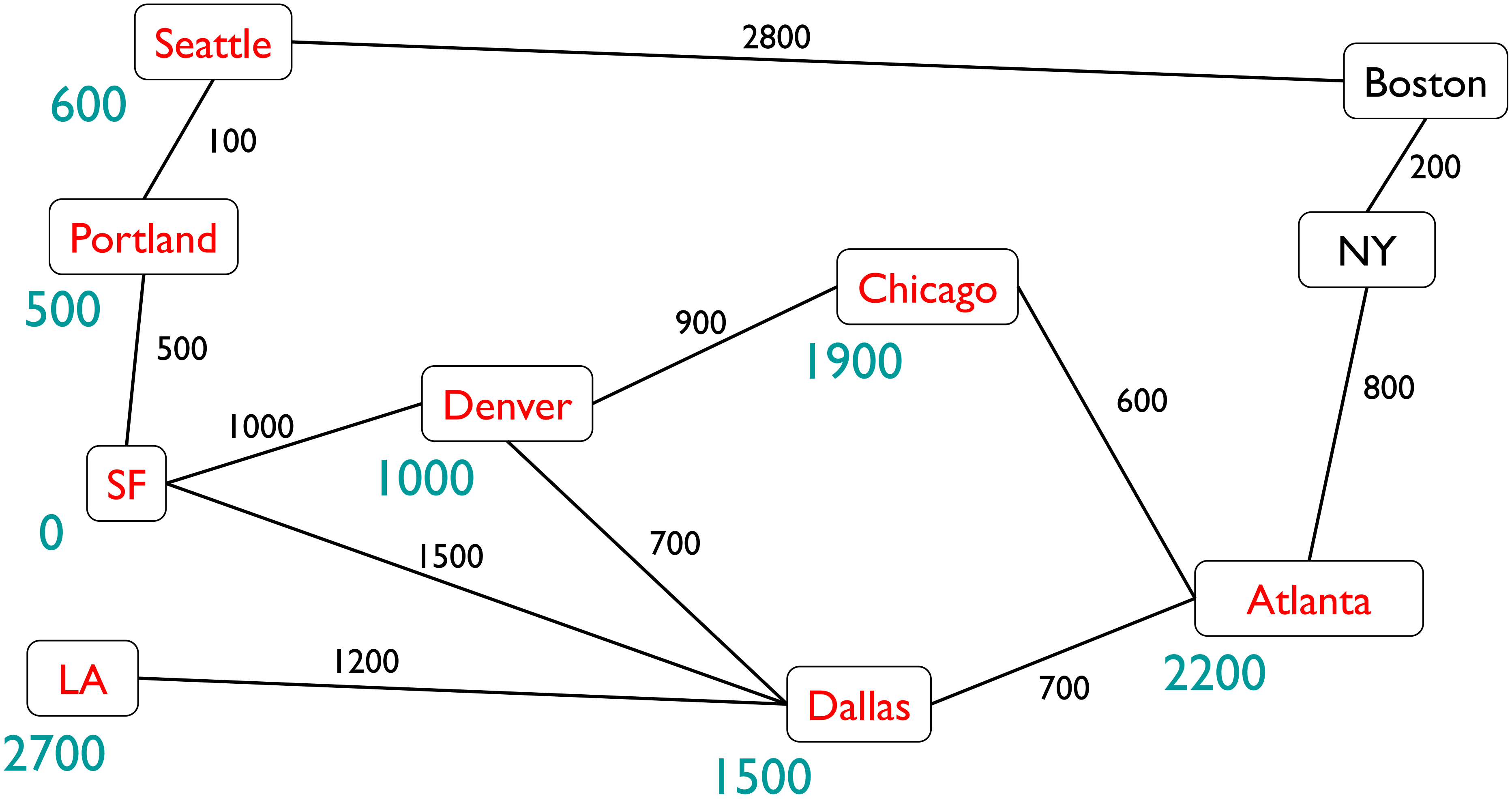
SF->Dal->Atl->NY; 3000

SF->Port->Sea->Bos 3400



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

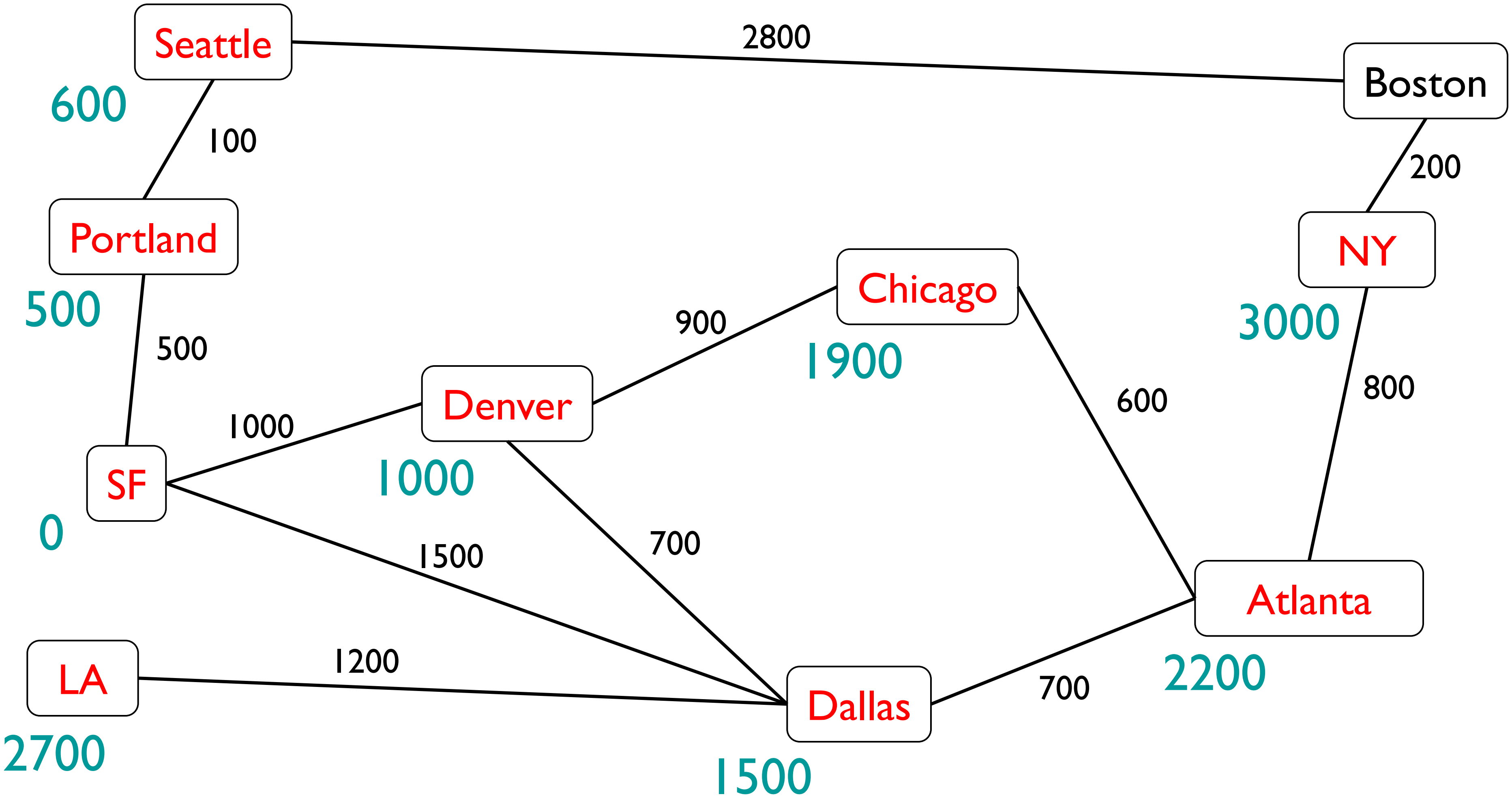
➡ SF->Dal->LA; 2700 SF->Dal->Atl->NY; 3000 SF->Port->Sea->Bos 3400



Current: 2700 SF->Dal->LA

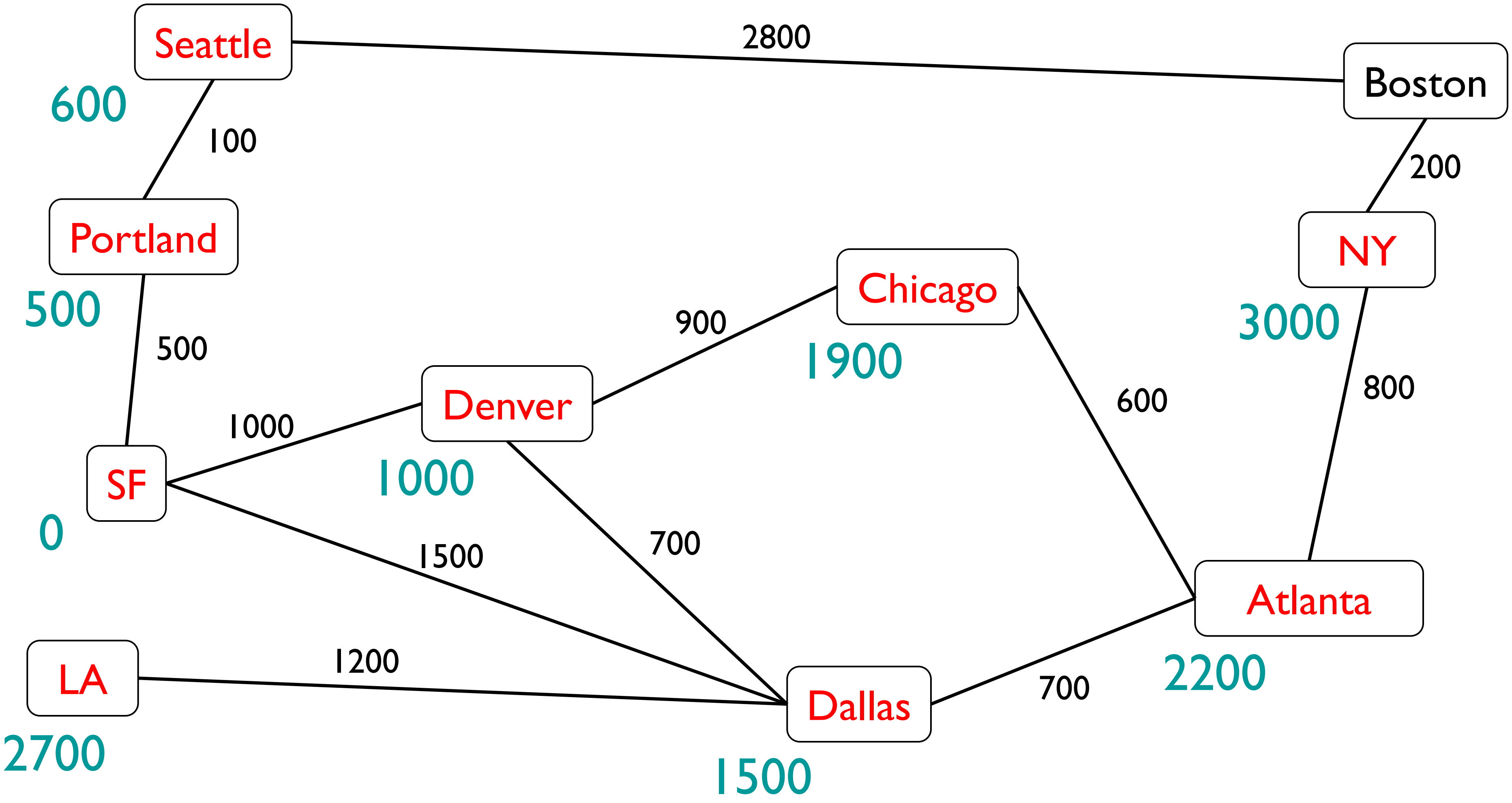
⇨ SF->Dal->Atl->NY; 3000

SF->Port->Sea->Bos 3400



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

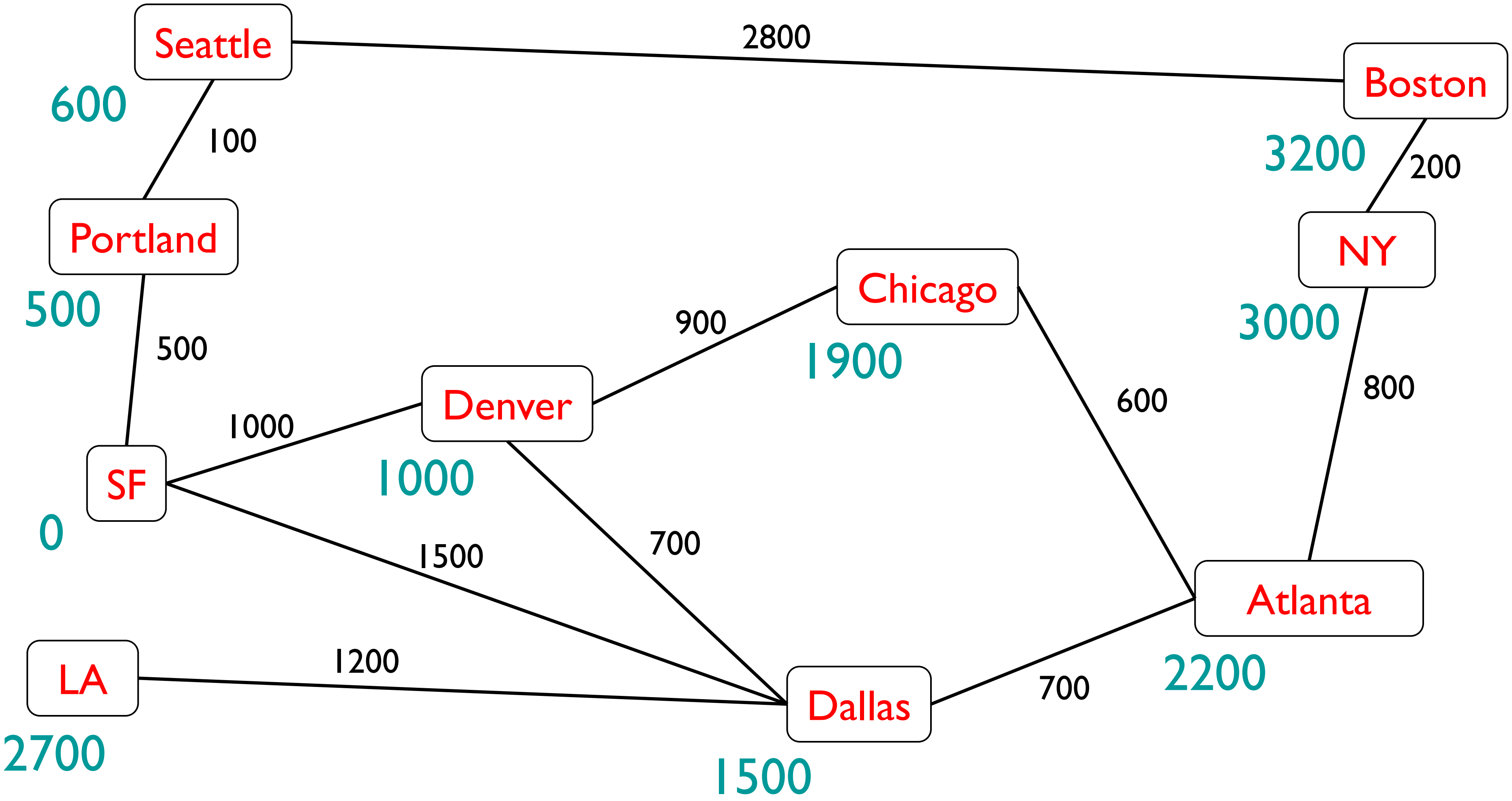
⇨ SF->Port->Sea->Bos
3400



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

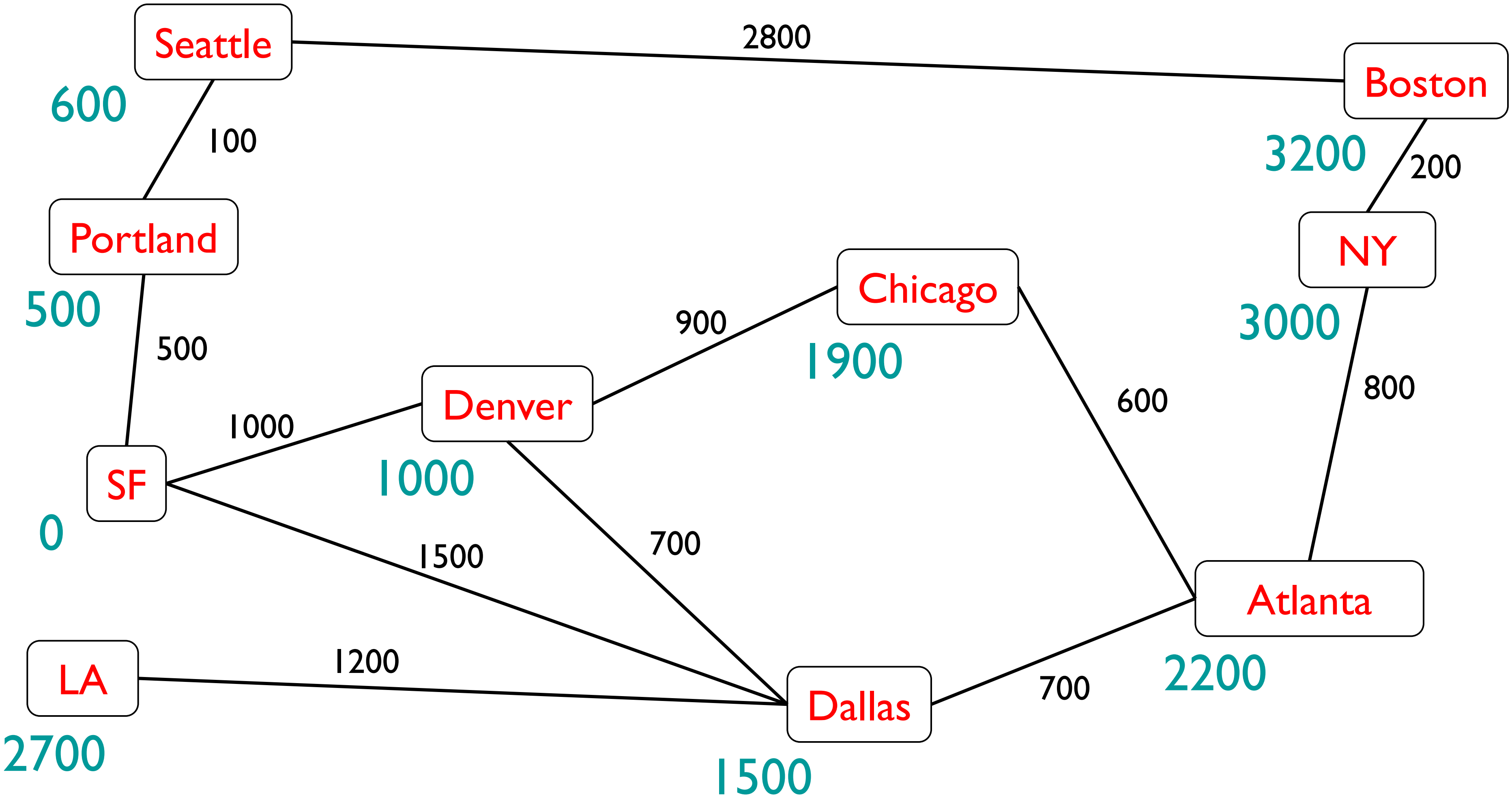
⇨ SF->Dal->Atl->NY->Bos; 3200

SF->Port->Sea->Bos 3400

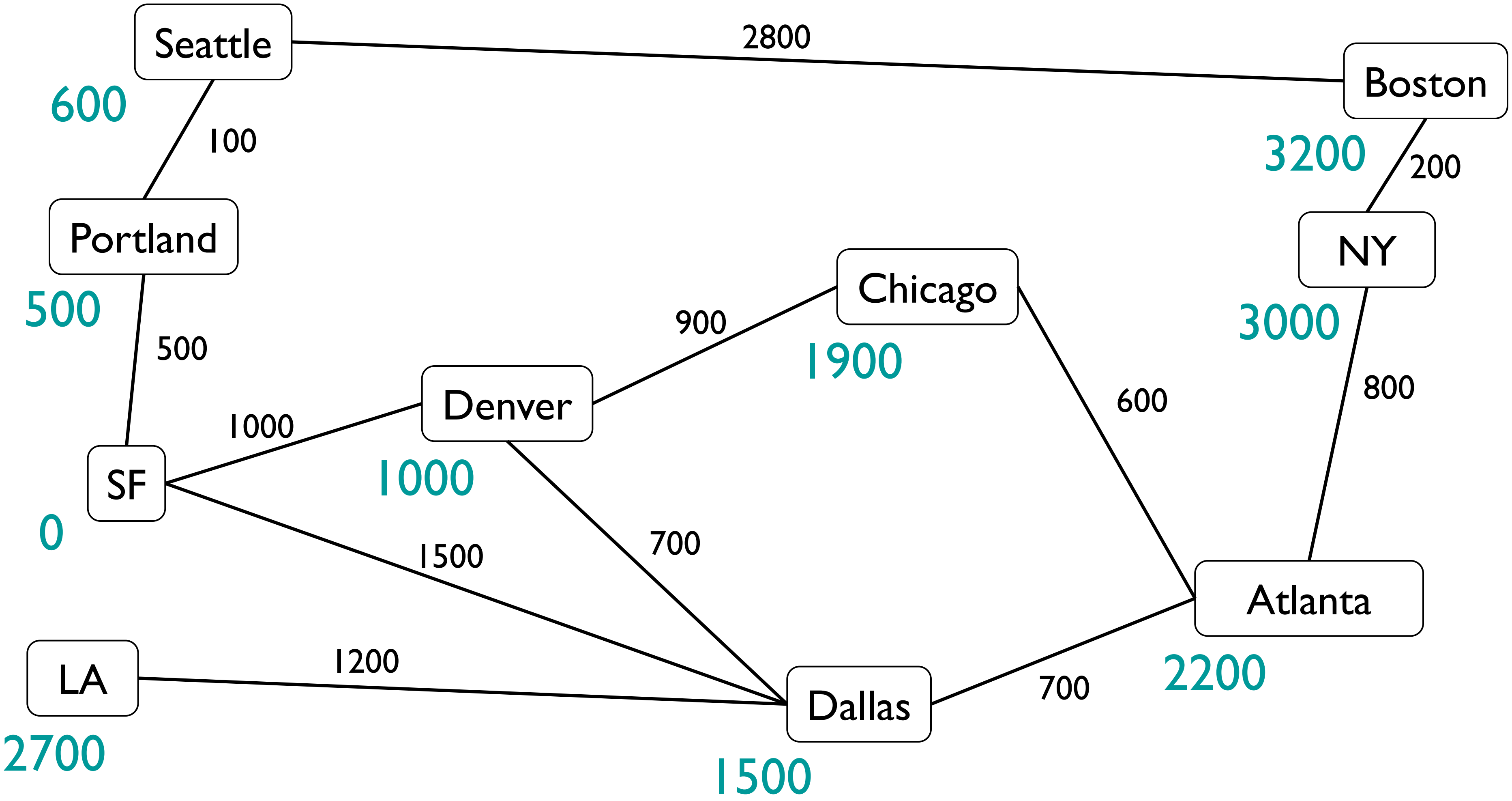


Current: 3200 SF->Dal->Atl->NY->Bos

⇨ 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
Denver
1000
600
800
1000
SF
0
1500
700
Atlanta
2200
LA
2700
1200
Dallas
1500
700

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|)