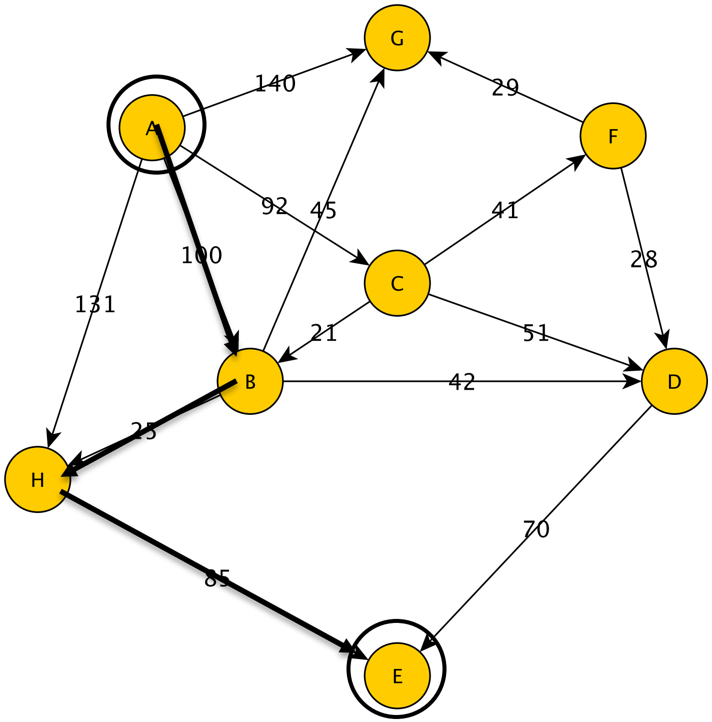
CSCI 136
Data Structures &
Advanced Programming

Shortest Paths in Weighted Graphs
(Dijkstra's Algorithm)

Shortest Paths With Edge Weights



The Problem

Input:

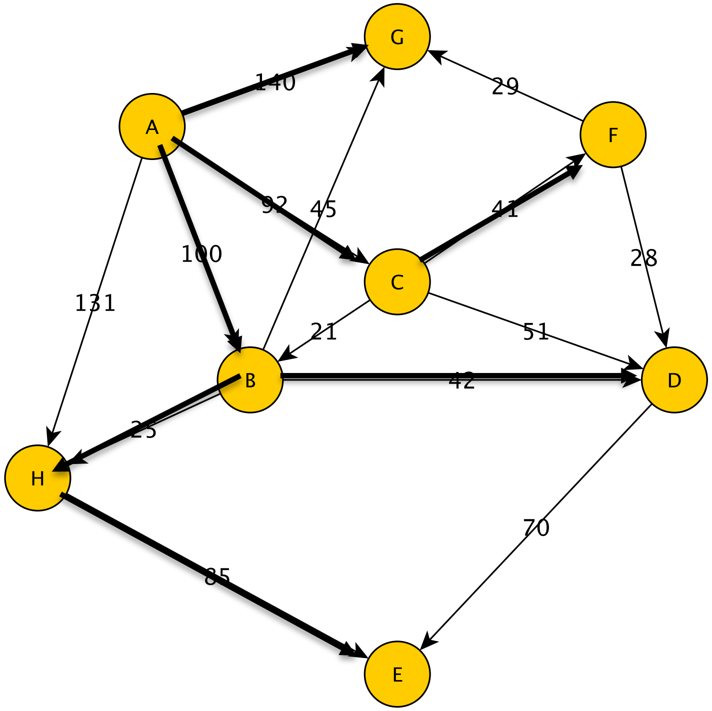
- A directed graph G=(V,E)
- A non-negative *length* for each edge
- Vertices s, v in V

Output:

- A *shortest* path from s to v
    - Path length: sum of lengths of edges on path

Single Source Shortest Paths

Appears to not be any simpler than finding shortest paths from s to *every vertex reachable from s!*
So….



The Problem

Input:

- A directed graph G=(V,E)
- A non-negative length for each edge
- A vertex s in V

Output:

- Shortest paths from s to every vertex reachable from s

All Pairs Shortest Paths

The Setup: Graph G=(V,E) for which each edge e in E has an *edge weight* w(e).

- It's tradition: We say edge *weights,* not edge *lengths*

The Problem: Compute shortest paths between each pair of vertices.

- It's tradition: We say *shortest* paths, not *lightest-weight* paths

Idea: For each vertex s, find shortest paths from s to *every other vertex reachable from s*

- Used for transportation, communication, and other networks
- The graph can be directed or undirected
- For specificity, we'll work with directed graphs

Single Source Shortest Paths

What does such a set of directed paths look like?

- Suppose we have a set shortest paths $\{P_u : u \neq s\}$, where P_u is a shortest path from s to u
  - There's a path P_u for each vertex u reachable from s
- Let H be the subgraph of G consisting of each vertex of G along with the edges in each P_u
- What can we say about H?
  - In example, it looked like a directed tree
  - Is that always the case?

Aside : An Optimality Property

Let P_u be a shortest path from s to u

- Write P_u as , given by $s = v_0, v_1, \ldots, v_k = u$
  - We can ignore edges in our notation: each (v_i,v_{i+1}) is an edge
- Consider any portion $v_i, v_{i+1}, \ldots, v_j$ of the path.
- Claim: $v_i, v_{i+1}, \ldots, v_j$ must be a shortest path from v_i to v_j
  - If there were a shorter path P' from v_i to v_j, we could replace $v_i, v_{i+1}, \ldots, v_j$ in P with P'
  - But this is a shorter path from v to u
  - Contradiction!

So: *Sub-paths of shortest paths must be shortest paths*

Single Source Shortest Paths

Claim: There always exists a family of shortest paths that forms a tree (ignoring edge directions)

Proof:

- Suppose, for each vertex u reachable from s, we have a shortest path P_u from s to u
- Let H be the subgraph of G consisting of the vertices and edges in each P_u
  - H is the set of vertices reachable (in G) from s
- If some vertex u has in-degree greater than 1, we can drop one of the incoming edges

Single Source Shortest Paths

If some vertex u has in-degree greater than 1, we can drop one of the incoming edges

- If there are two edges entering u, then one of them must be from P_u and the other from P_v, for some v
- So the initial portions of those paths from s to u must both have the same weight!
  - Recall: Subpaths of shortest paths are shortest paths
- So, replacing the portion of, say P_v from s to u with P_u gives a new shortest path from s to v.
  - So: The edge of P_v entering u can be dropped from H
  - But no other edge of P_v can be dropped!

Single Source Shortest Paths

Claim: H can’t have any directed cycles

- Well, s can’t be on any cycles (in-deg(v) = 0)
  - Otherwise, s appeared as a vertex somewhere along one of the paths P_u
  - But then P_u can't be a *shortest* path from s to u
- If there were a cycle, some vertex on it would have in-degree > 1
  - Since s is not on the cycle, There must be a path *from* s to some vertex u on the cycle.
  - But then u has indegree > 1

Single Source Shortest Paths

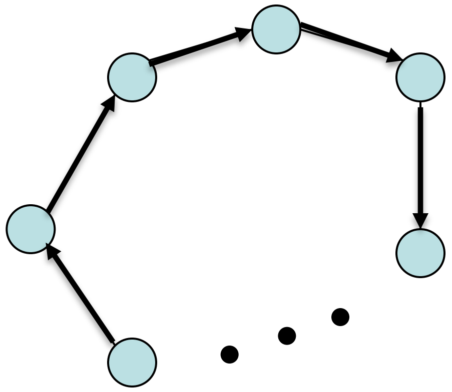
In fact, even disregarding edge directions, there would be no cycles

- 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
- Dijkstra's Algorithm: Greedily grow such a tree
- The question is: How?

Single Source Shortest Paths

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



So, the paths form a directed tree with root v!

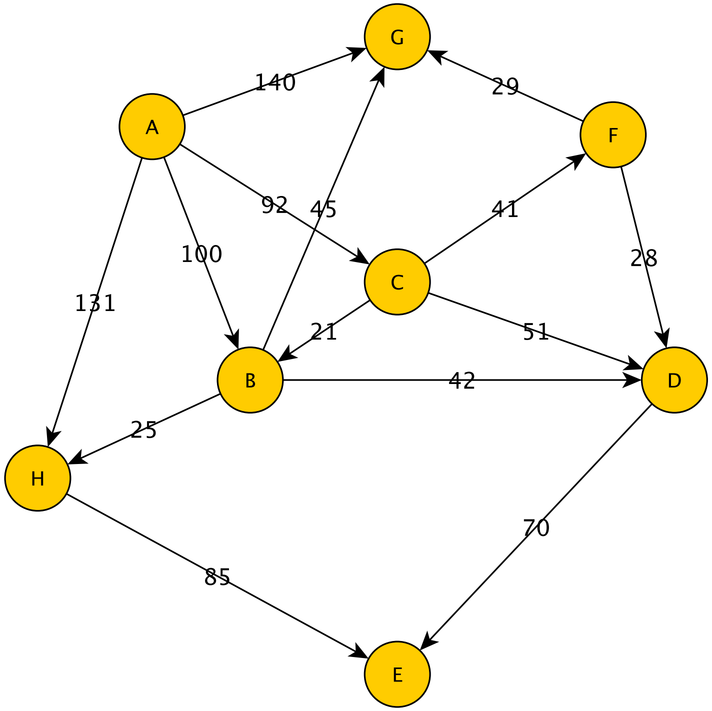
Single Source Shortest Paths

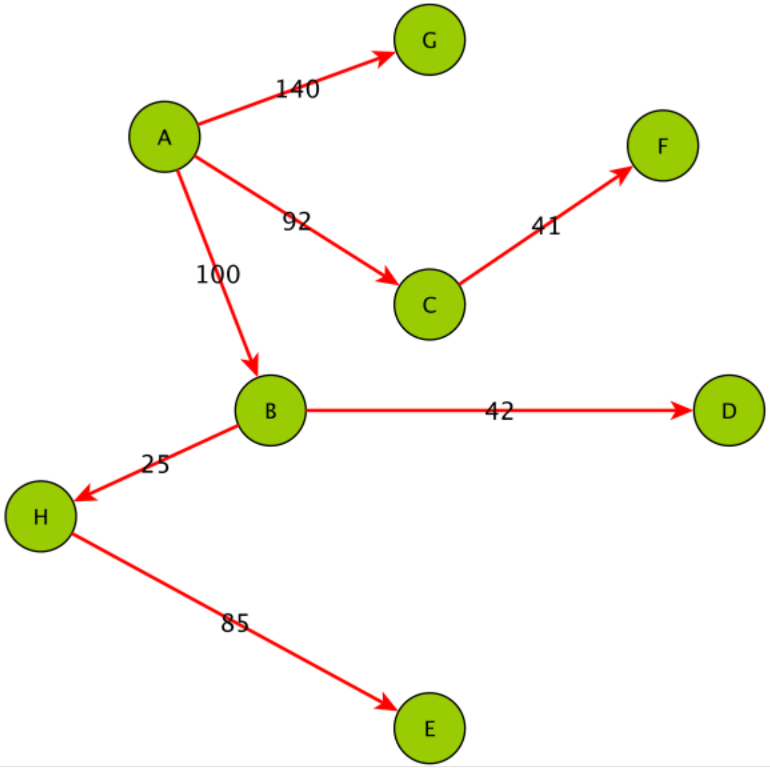
Thus: There always exists a family of shortest paths that forms a tree (ignoring edge directions)

Dijkstra's algorithm *grows* a tree T of shortest paths from s to every vertex reachable from s

- Begins with T just containing s
- Repeatedly adds a new vertex and edge to T
    - At all times, T consists of shortest paths (in G) from s to every other vertex of T
- Next vertex/edge is selected *greedily*

Dijkstra Shortest Paths Tree



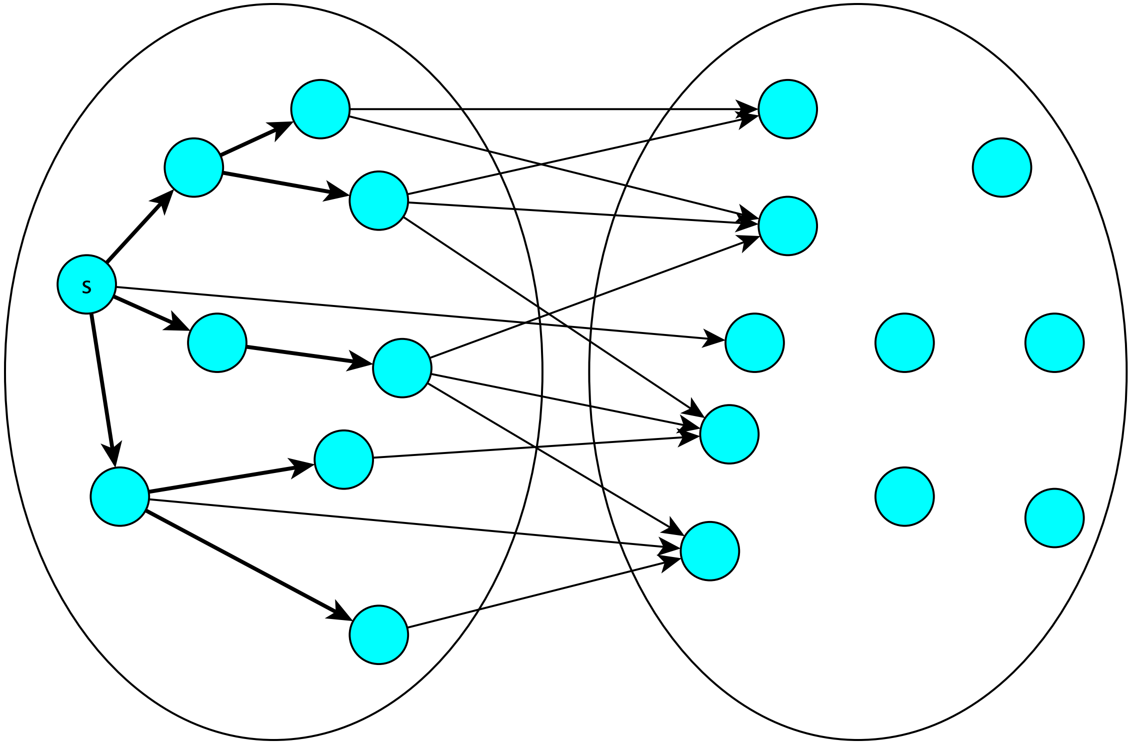


The Tree of Shortest Paths Found by Dijkstra's Algorithm

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 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?

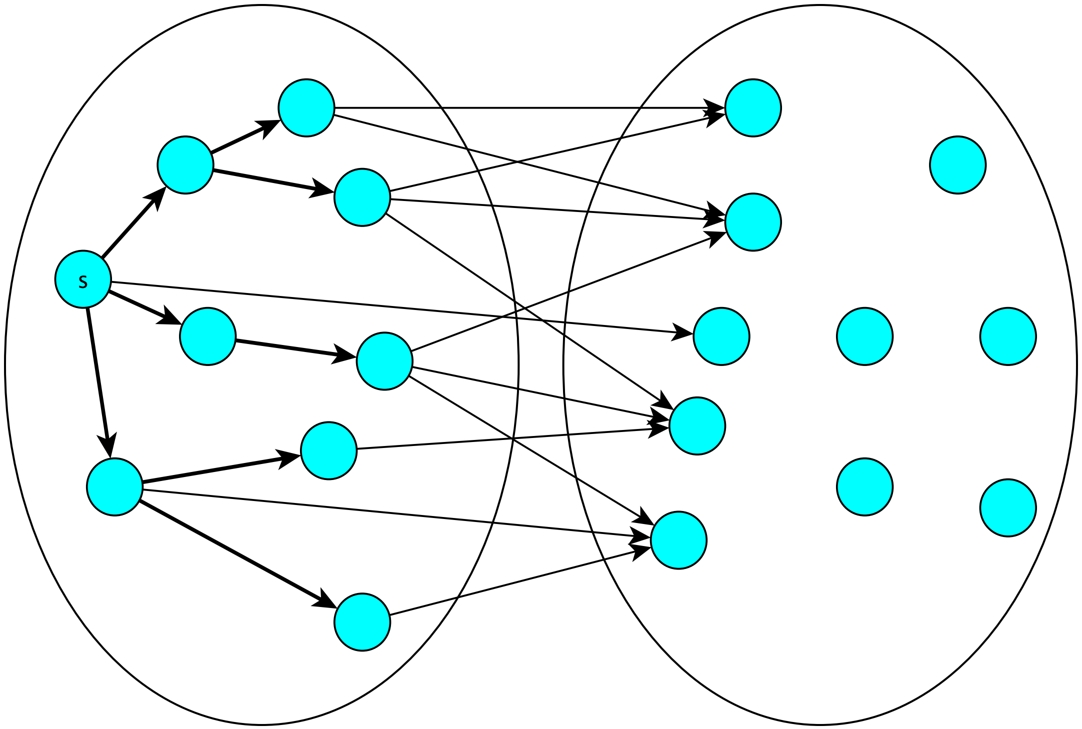
Finding the Best Vertex to Add to T_k



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

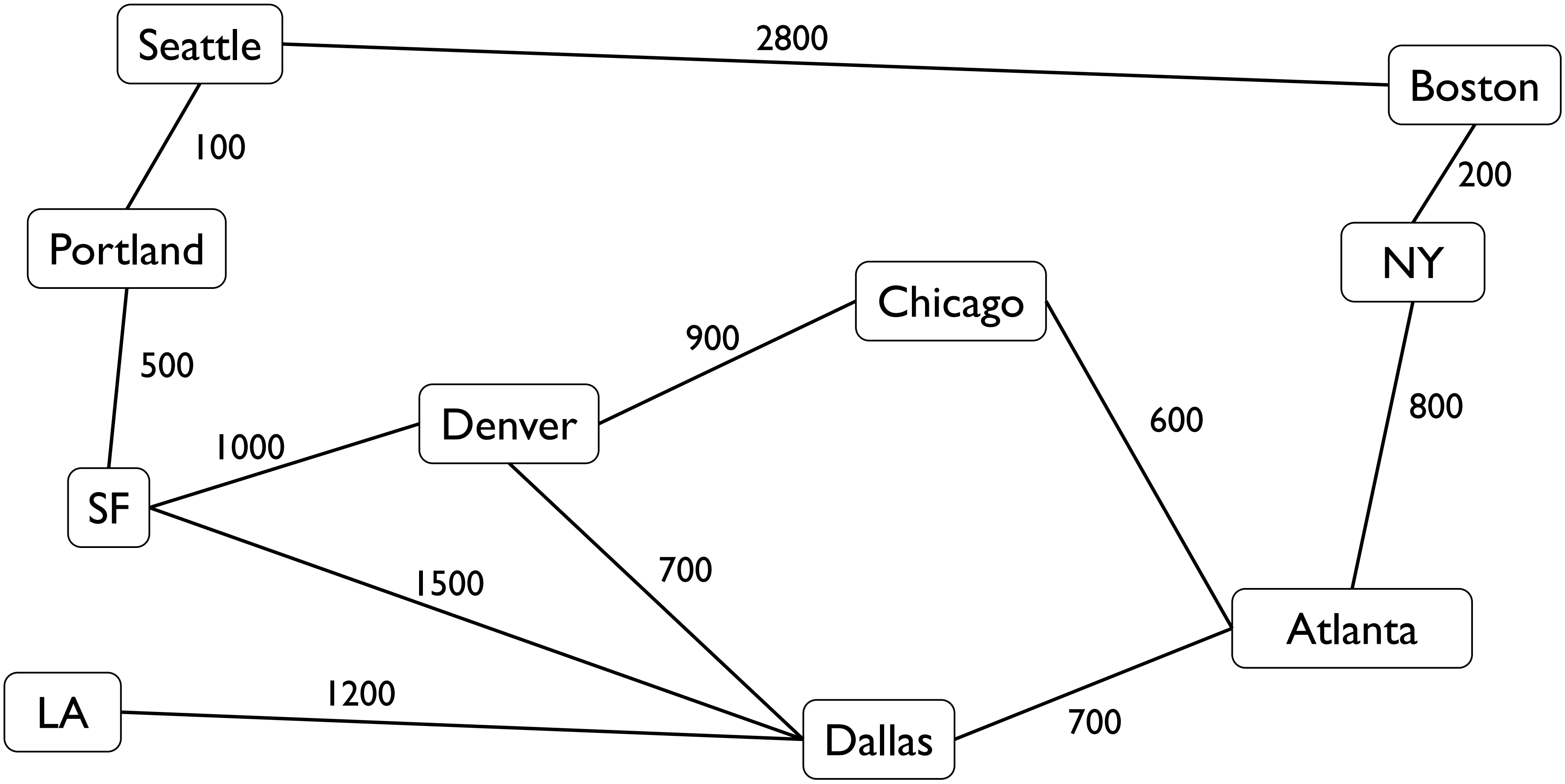
- The priority queue stores an *estimate* of the distance from s to w by storing, for edge (v,w), d(s,v) + l(v,w)
  - The estimate is always an *upper bound* on d(s,w)

Dijkstra: Data Structures

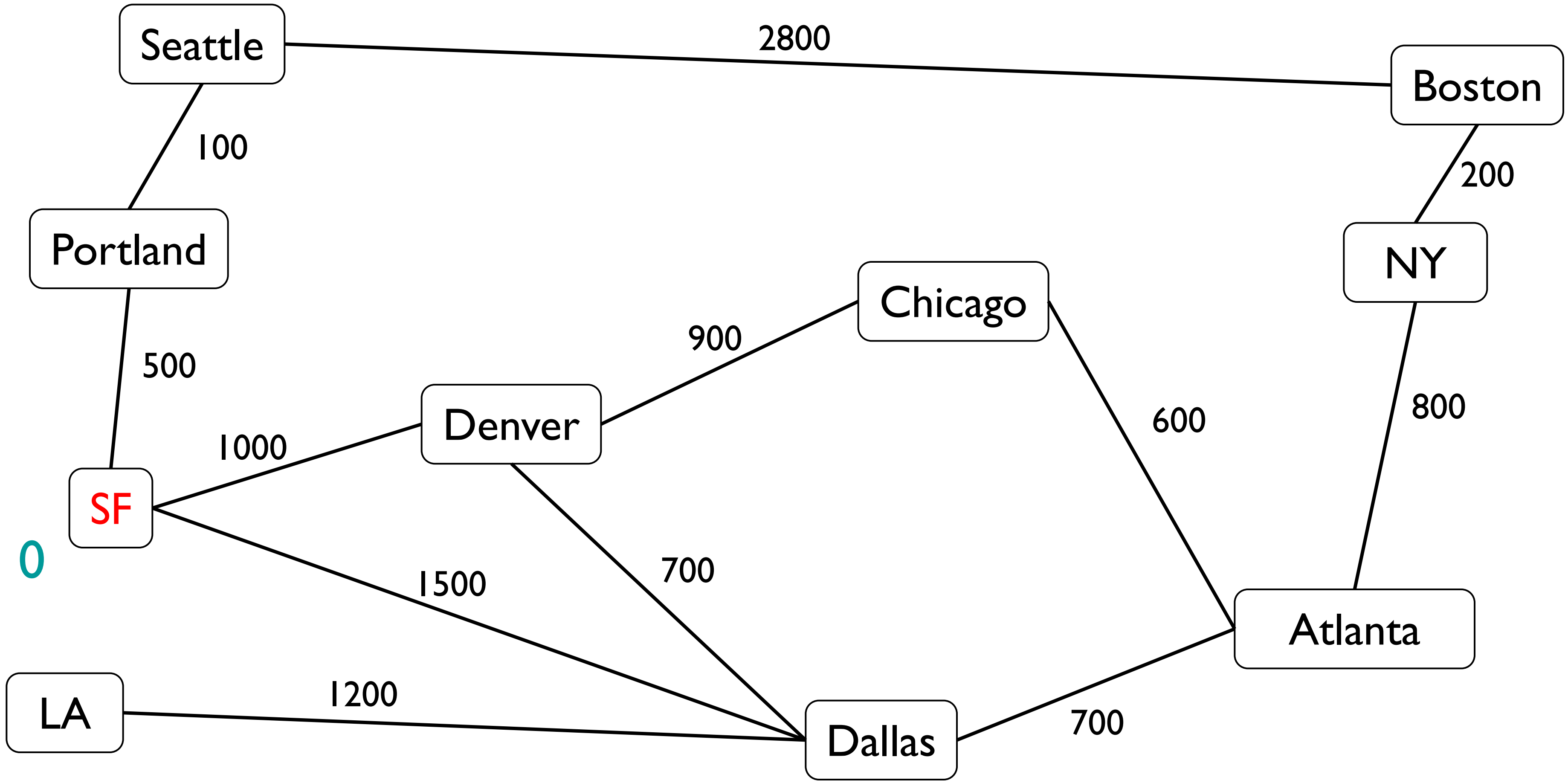
- Map: Store the tree T of shortest paths
  - Key is a vertex label v
  - Value is edge of T having v as *destination* vertex
    - From this we can find path in T from s to v
- Priority Queue: Store edges (v,w) with current approximate distance
  - As Comparable Association(Key,Value) where
    - Key is d(s,v) + l(v,w) : The estimated distance from s to w
    - Value is the edge e=(v,w)
  - The PQ will always contain all edges from vertices of T to vertices *not* in T
    - As well as some *vestigal* edges with both ends in T

Dijkstra's Algorithm

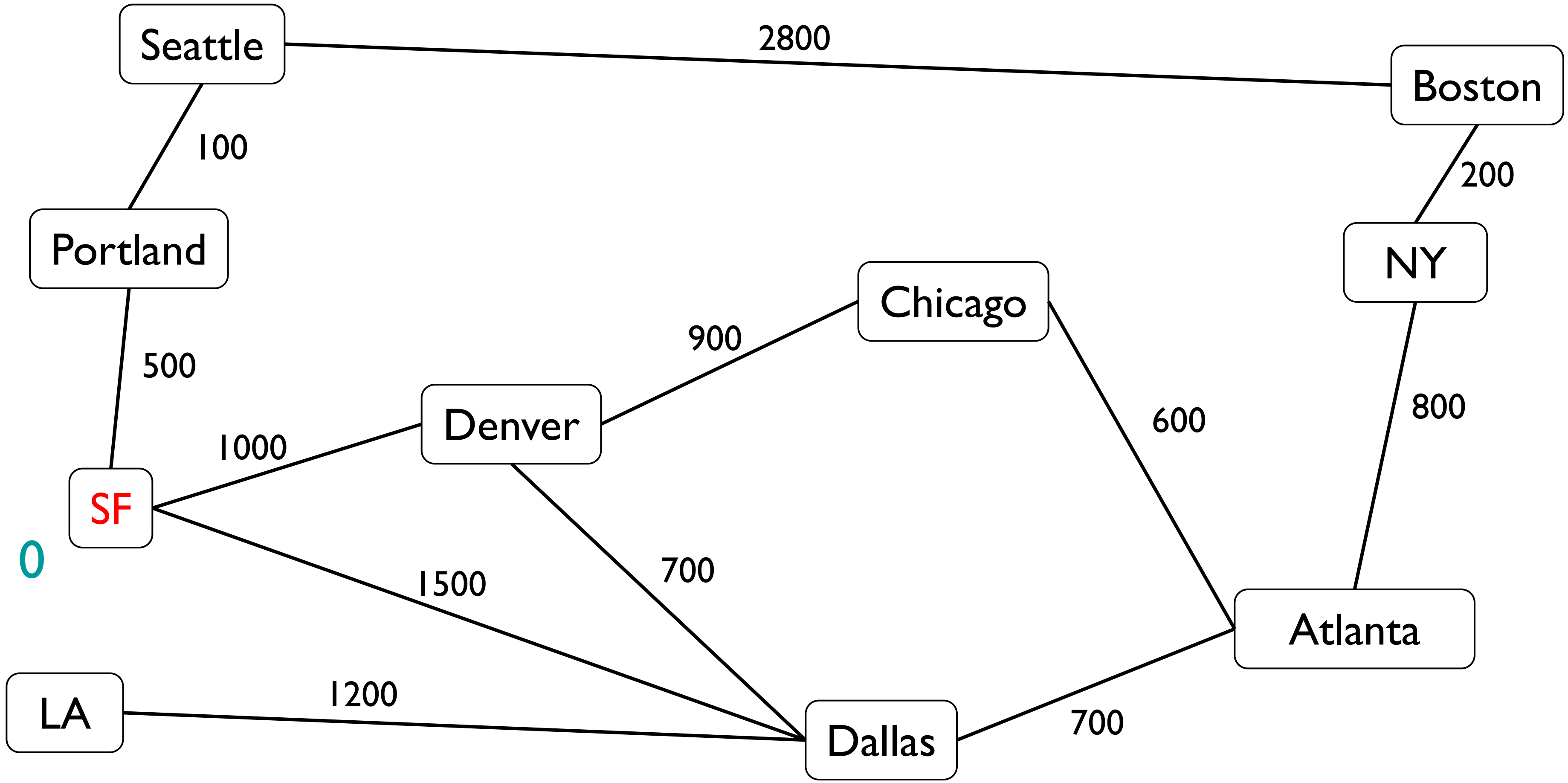
```
Dijkstra(G, s)  // l(e) is the length of edge e
let T ← ({s}, ∅)  and PQ be an empty priority queue
for each neighbor v of s, add edge (s,v) to PQ with priority l(e)
while T doesn't have all vertices of G and PQ is non-empty
    repeat
        e ← PQ.removeMin()   // skip edges with both ends in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

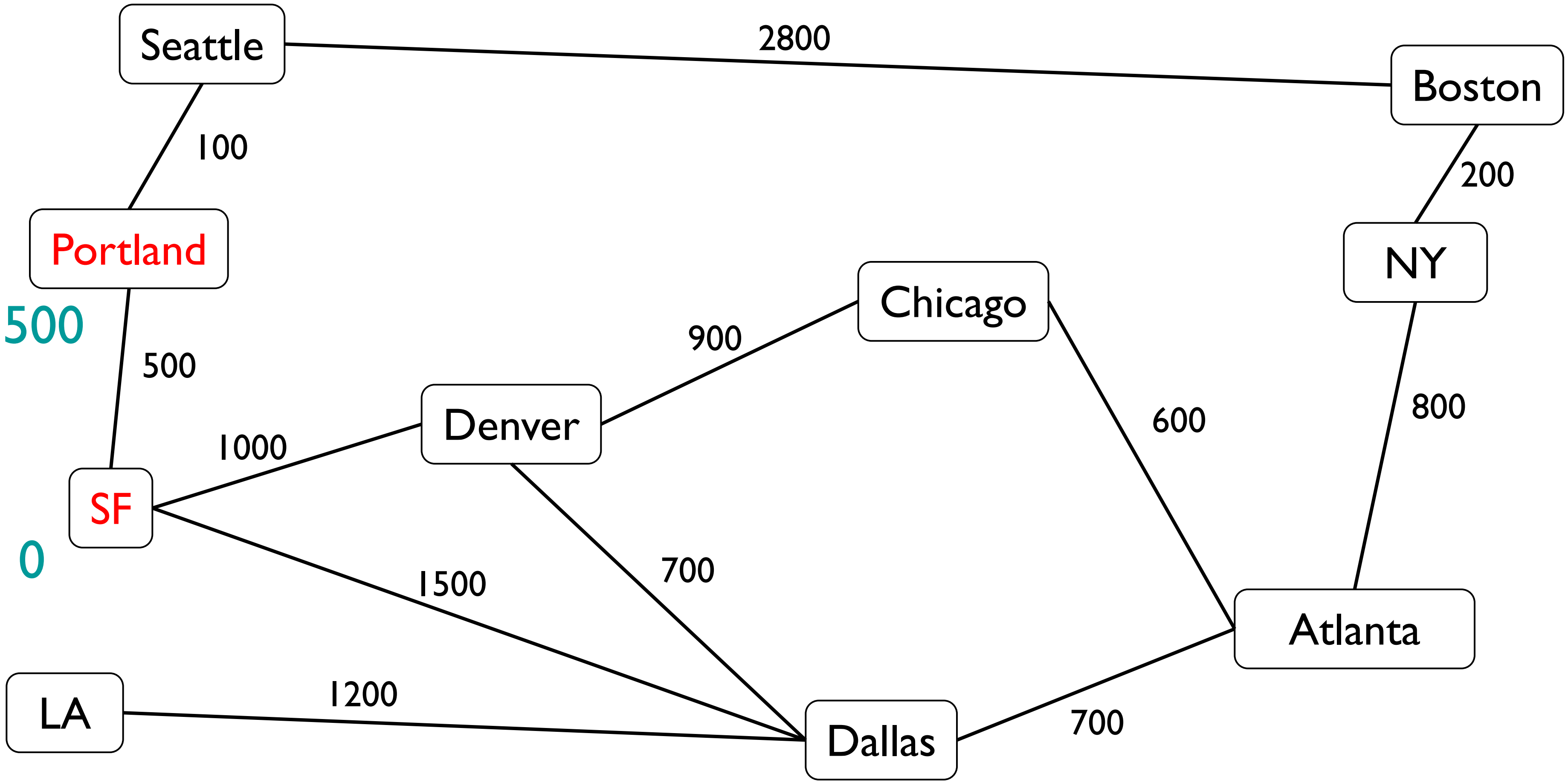


Priority Queue

SF->Port; 500

SF->Den; 1000

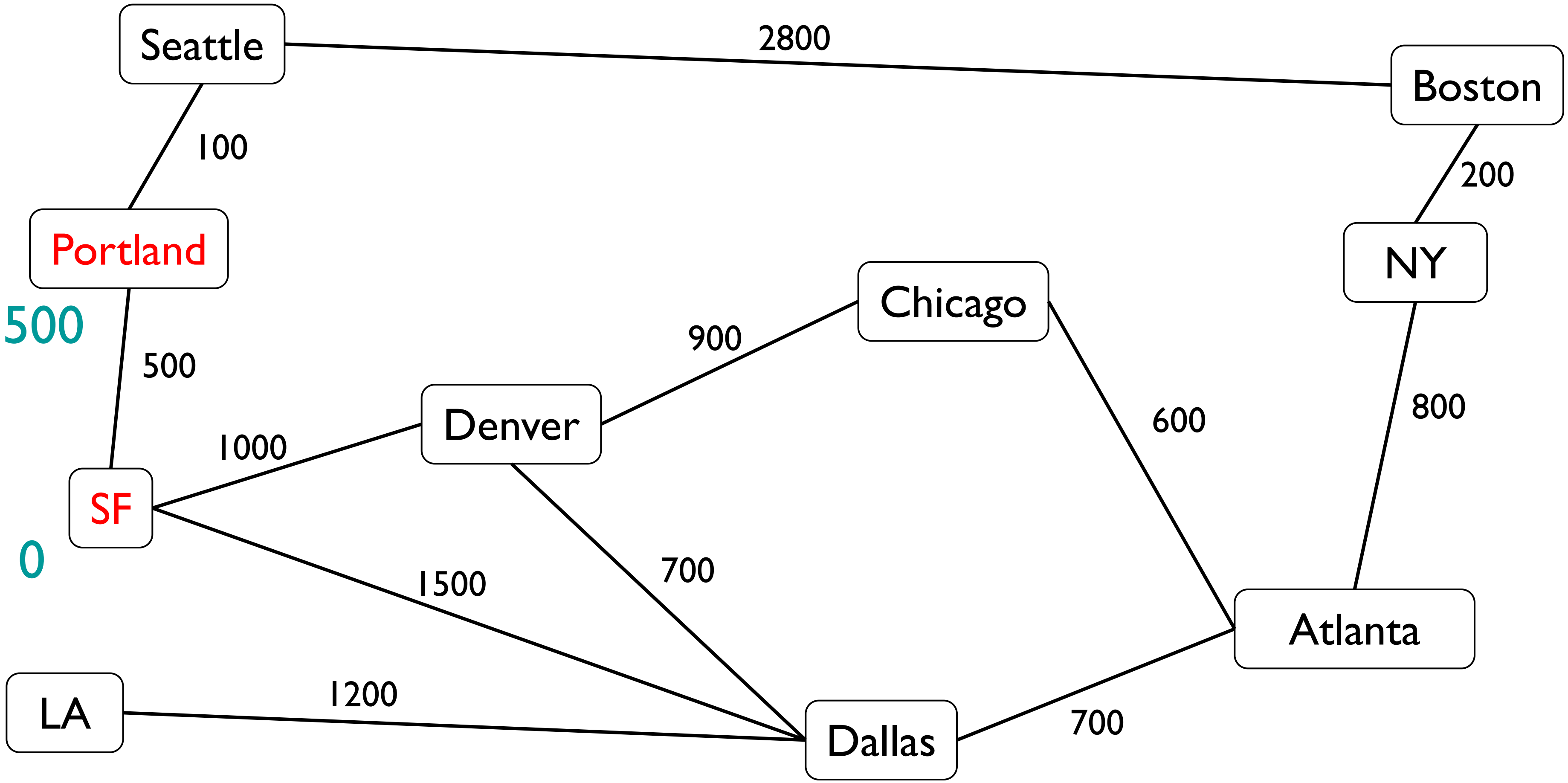
SF->Dal 1500



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

SF->Den; 1000

SF->Dal 1500

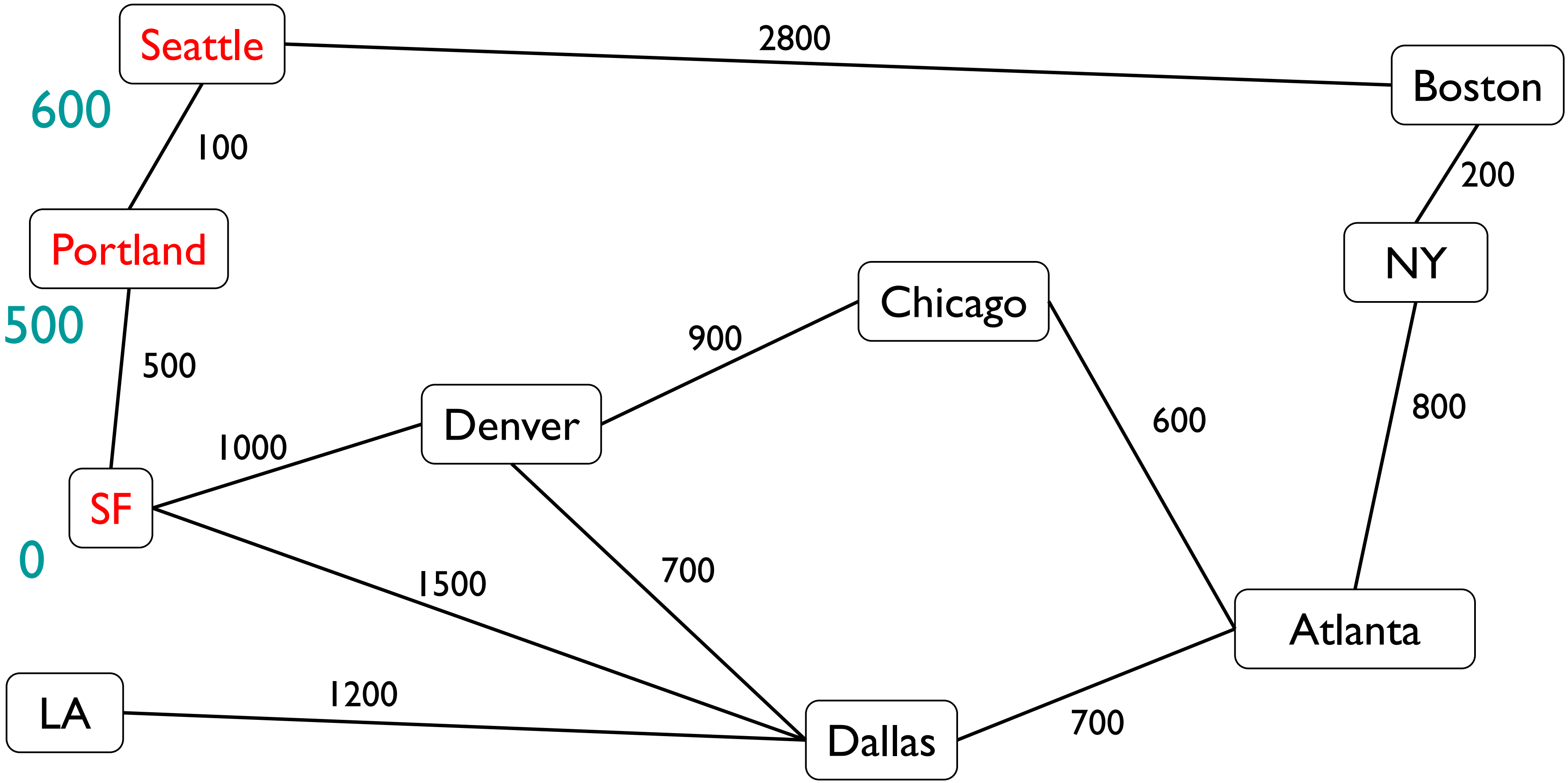


Current: 500 SF->Port

⇨ SF->Port->Sea; 600

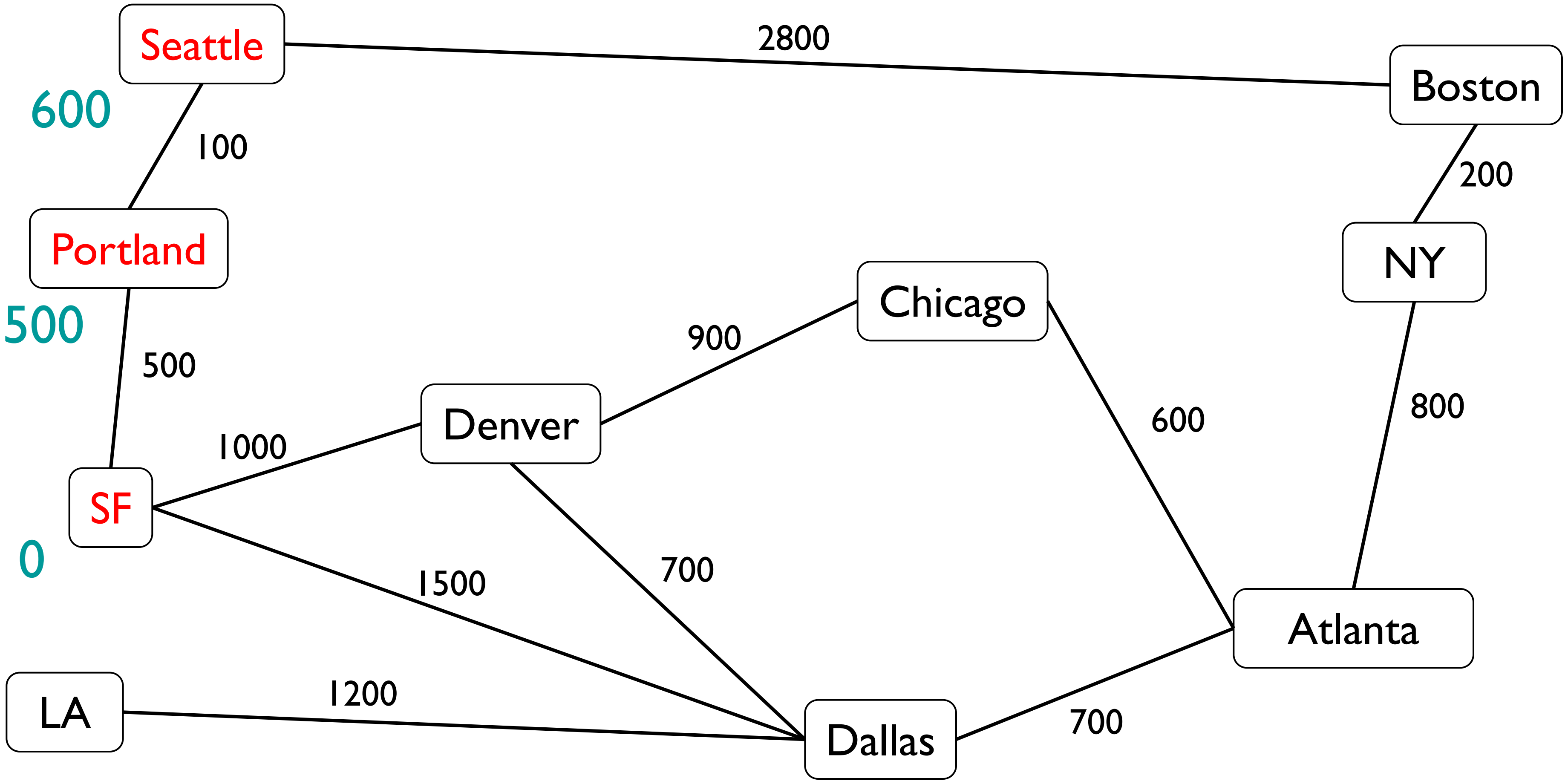
SF->Den; 1000

SF->Dal 1500



Current: 600 SF->Port->Sea

⇨ SF->Den; 1000 SF->Dal 1500

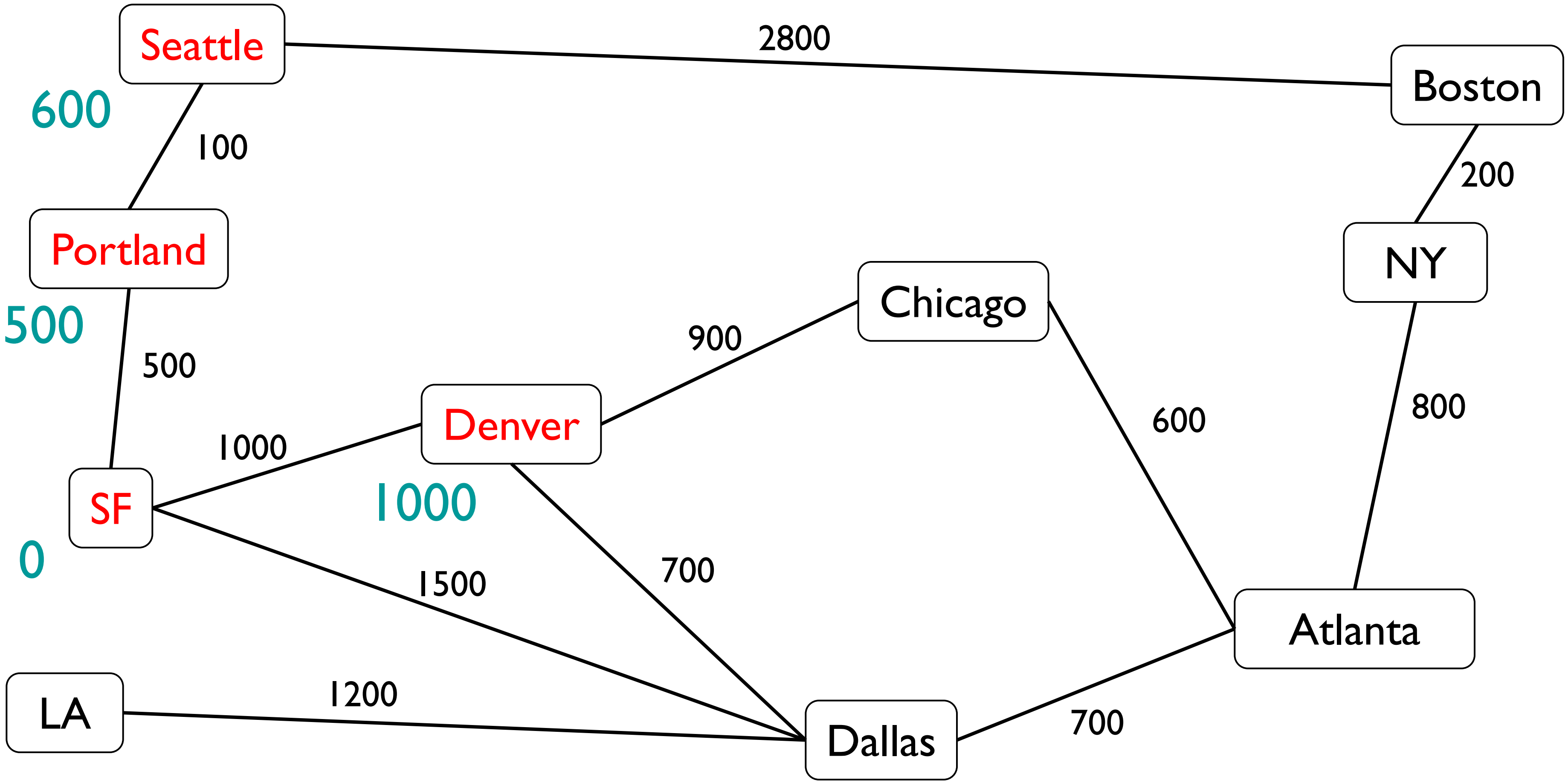


Current: 600 SF->Port->Sea

SF->Den; 1000

SF->Dal; 1500

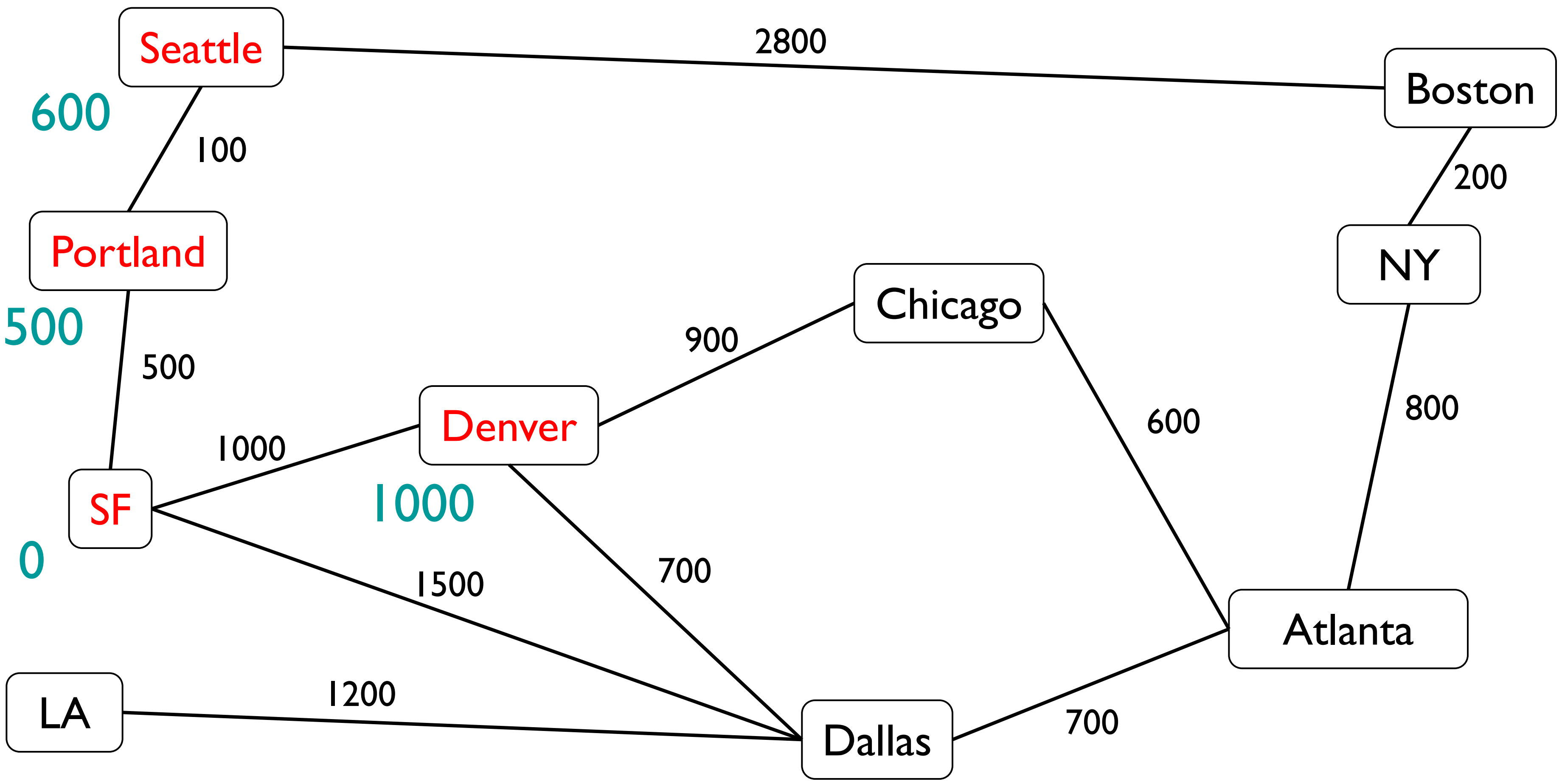
SF->Port->Sea->Bos 3400



Current: 1000 SF->Den

⇨ SF->Dal; 1500

SF->Port->Sea->Bos 3400



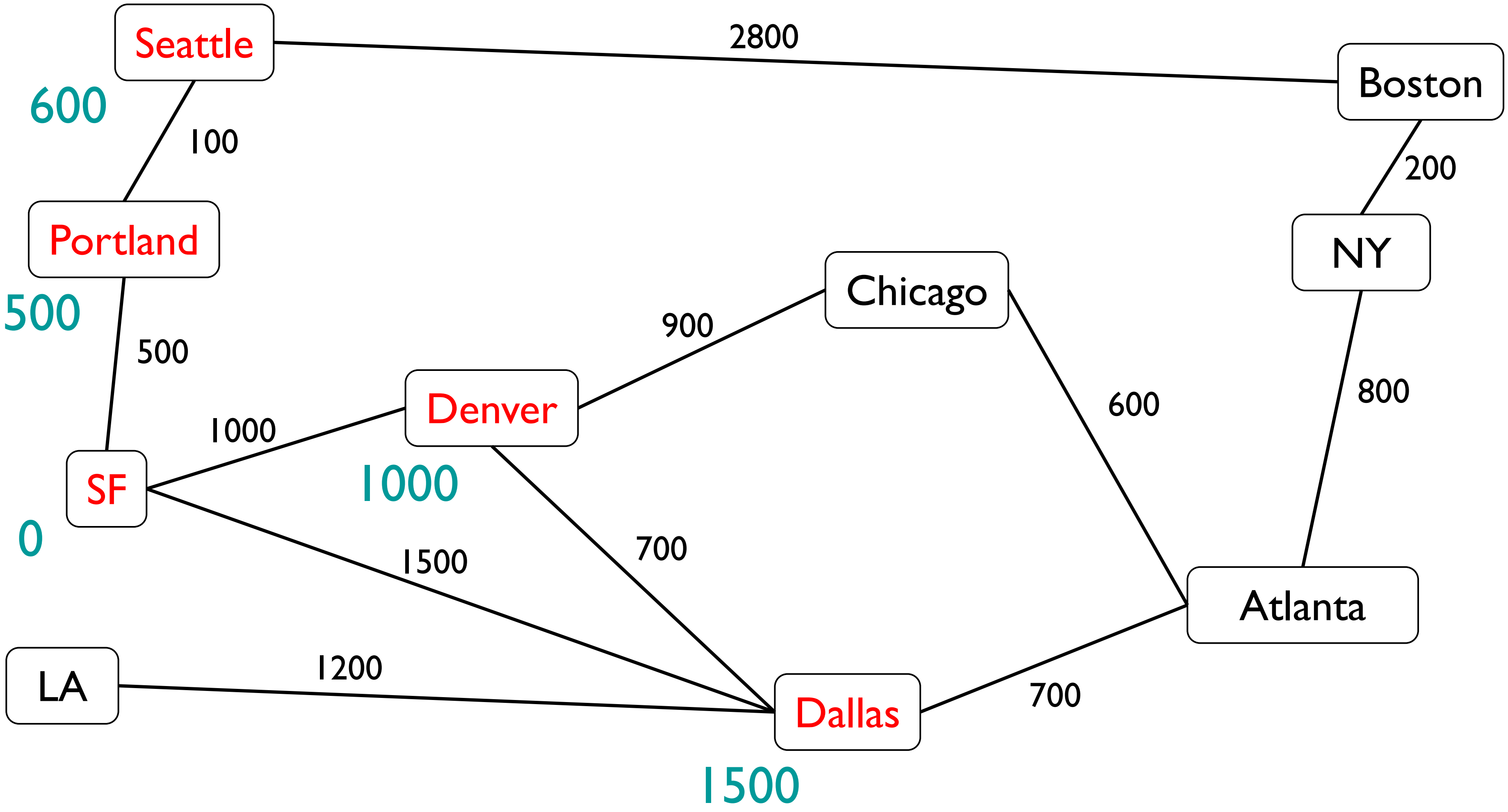
Current: 1000 SF->Den

SF->Dal; 1500

SF->Den->Dal; 1700

SF->Den->Chi; 1900

SF->Port->Sea->Bos 3400

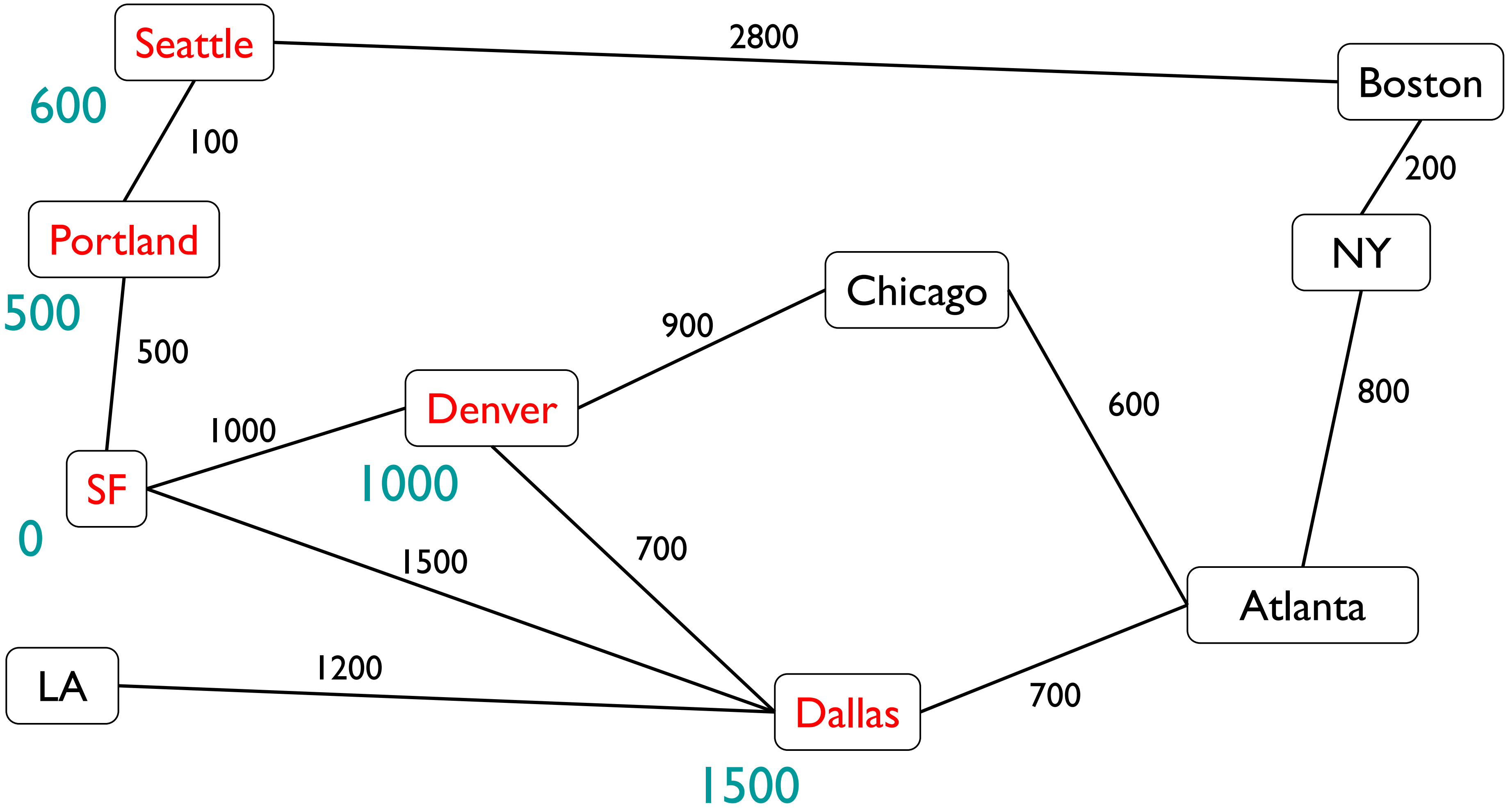


Current: 1500 SF->Dal

⇨ SF->Den->Dal; 1700

SF->Den->Chi; 1900

SF->Port->Sea->Bos 3400



Current: 1500 SF->Dal

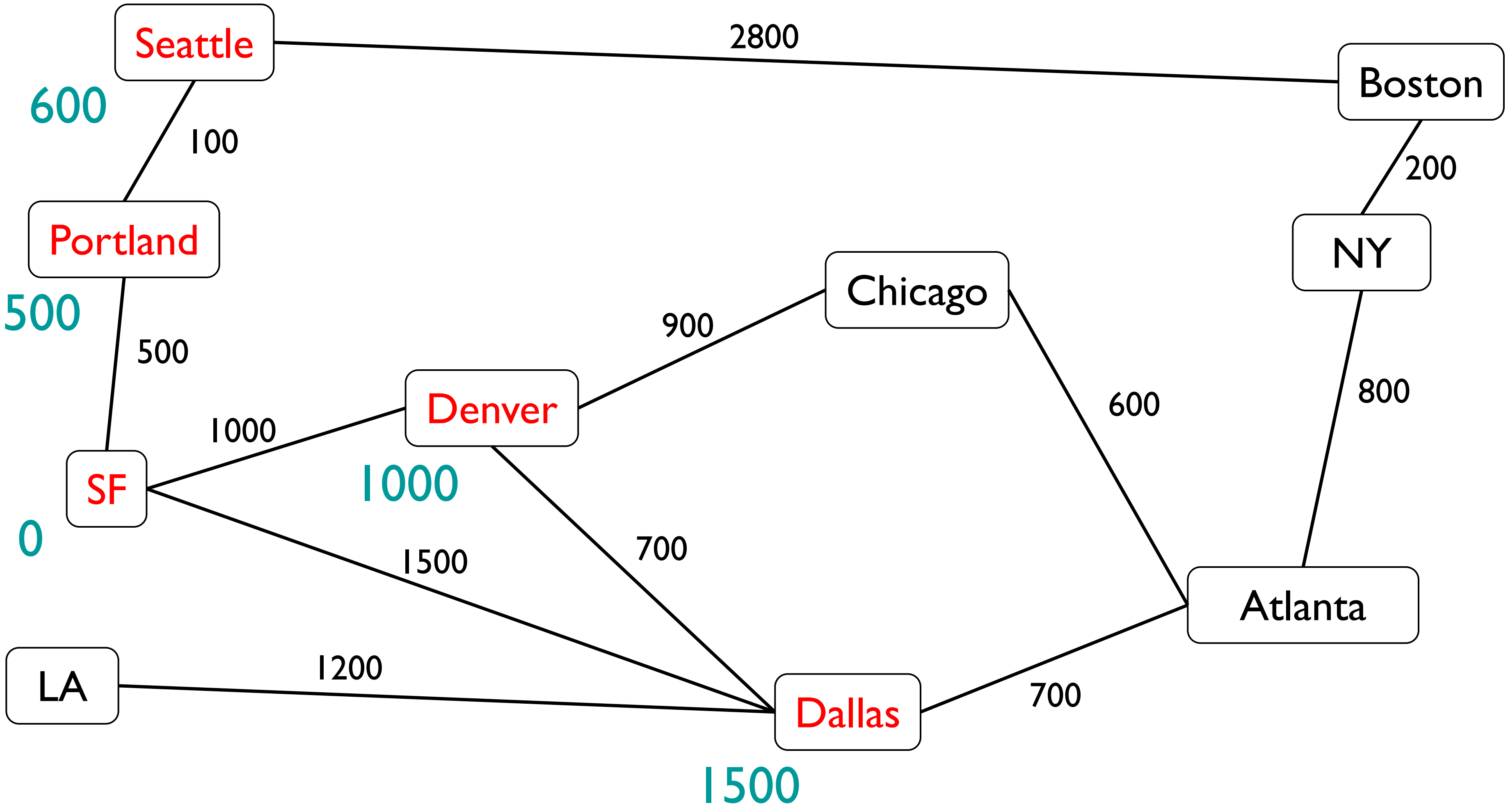
⇨ SF->Den->Dal; 1700

SF->Den->Chi; 1900

SF->Dal->Atl; 2200

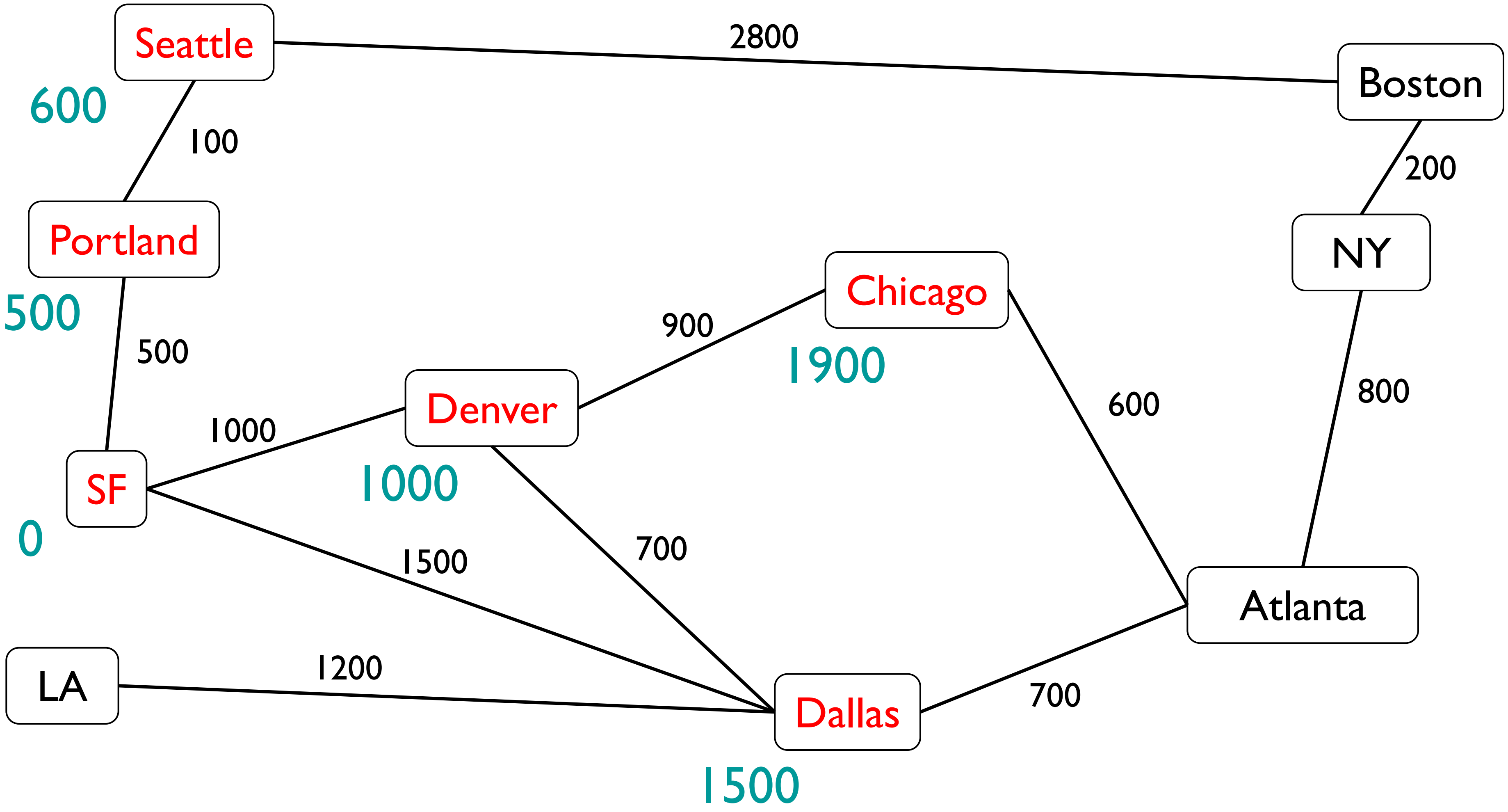
SF->Dal->LA; 2700

SF->Port->Sea->Bos 3400



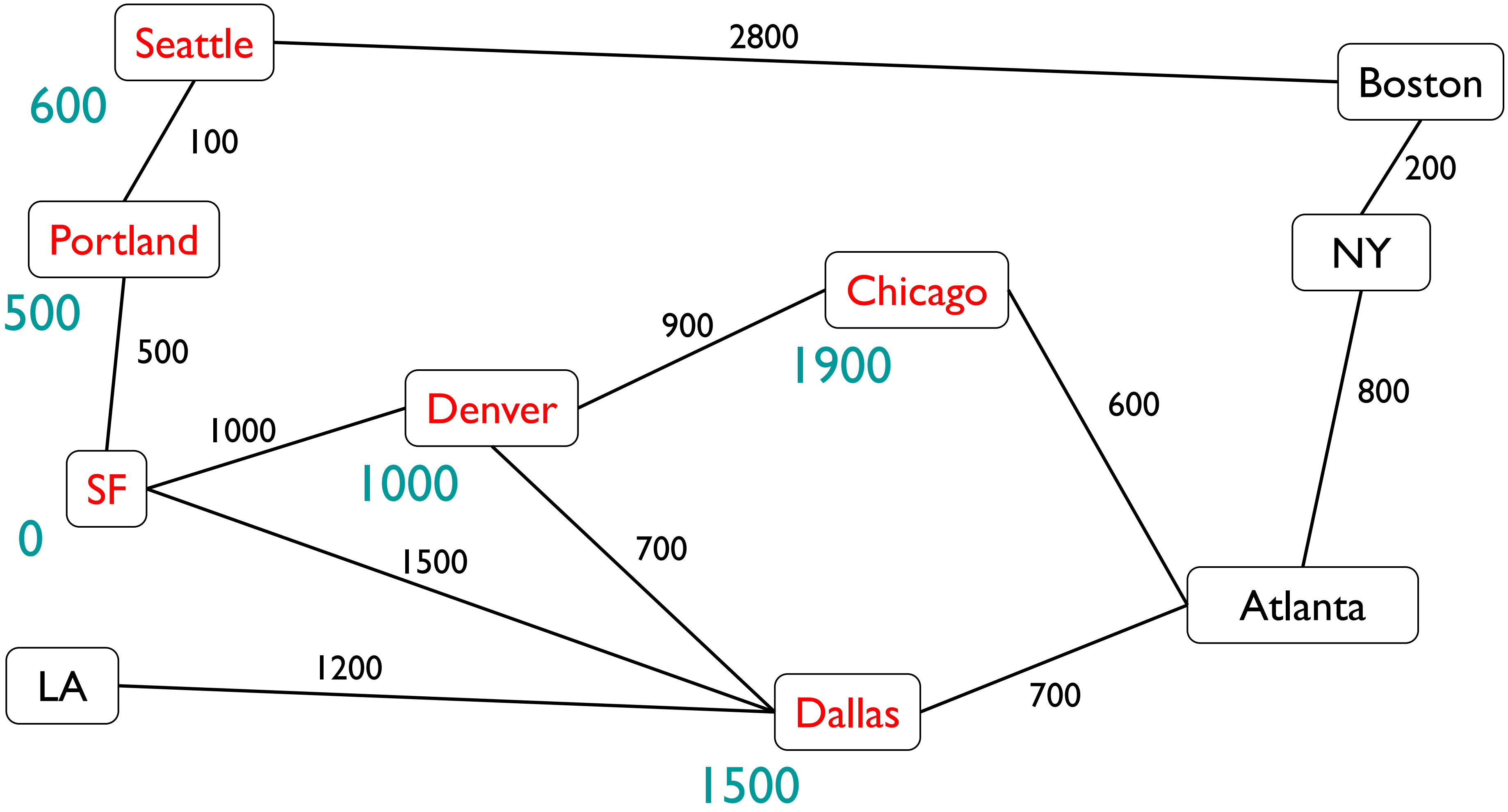
Current: 1700 SF->Den->Dal (we already have Dallas!)

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



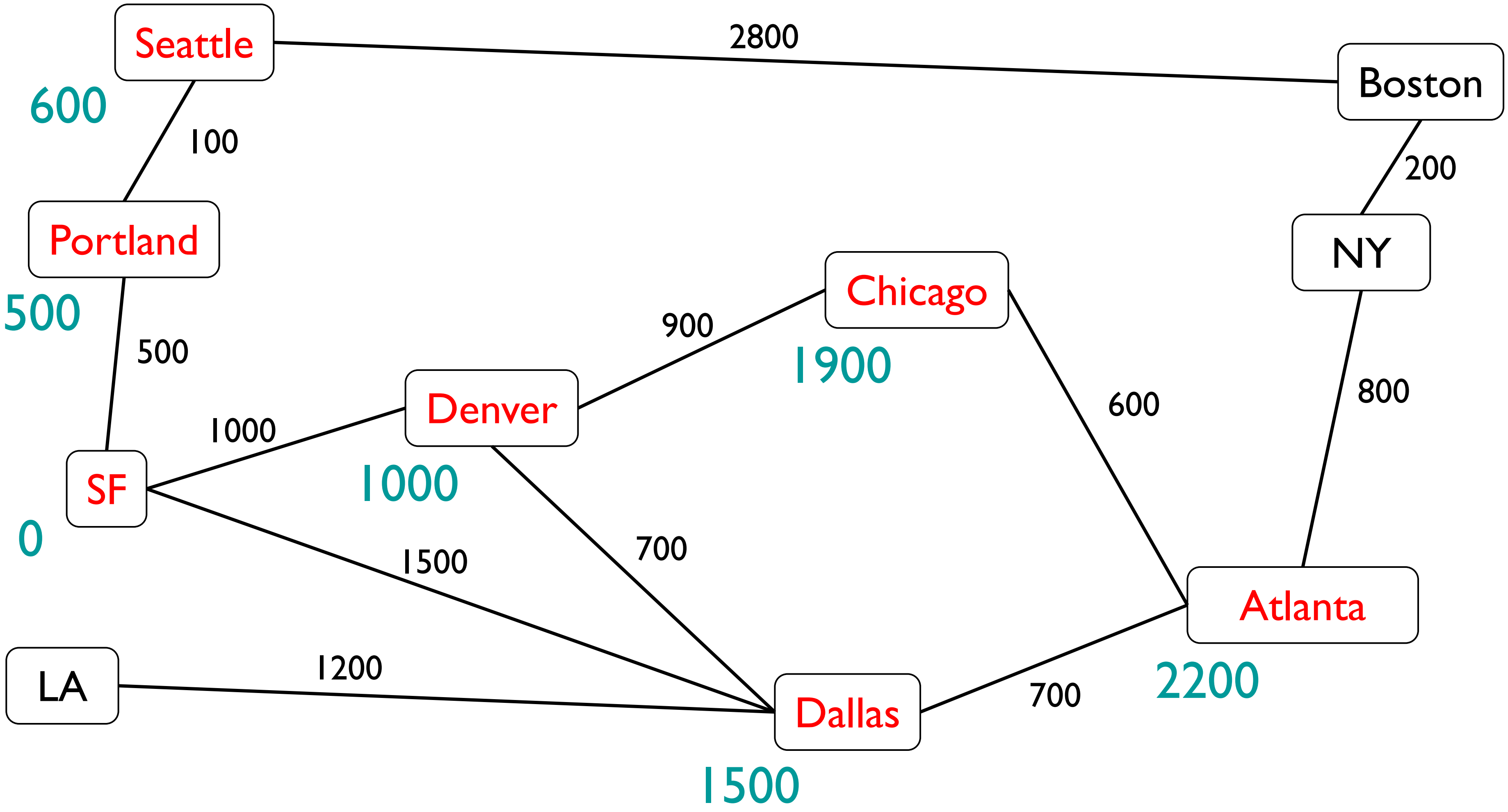
Current: 1900 SF->Den->Chi

SF->Dal->Atl; 2200

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

SF->Dal->LA; 2700

SF->Port->Sea->Bos 3400

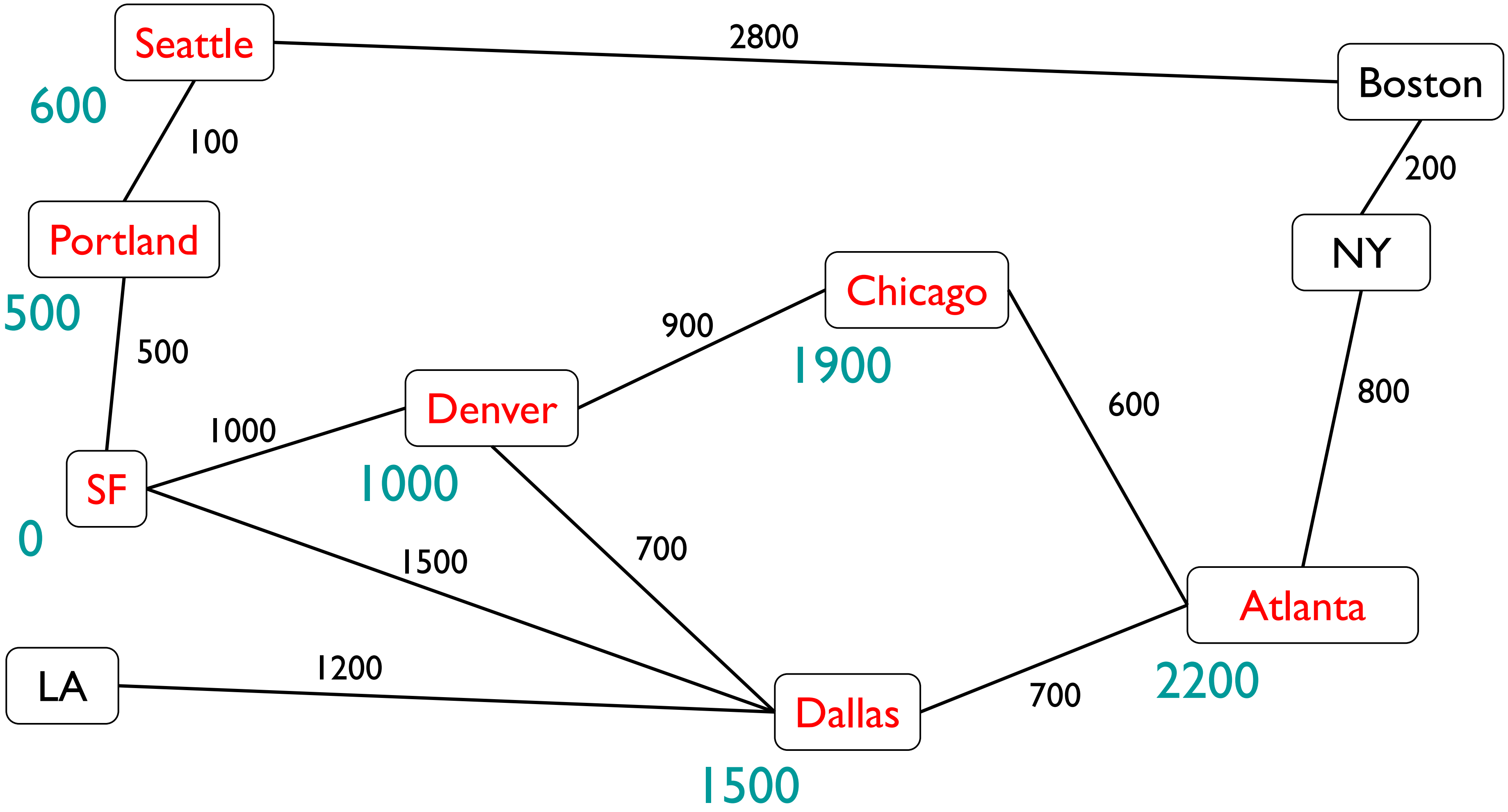


Current: 2200 SF->Dal->Atl

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

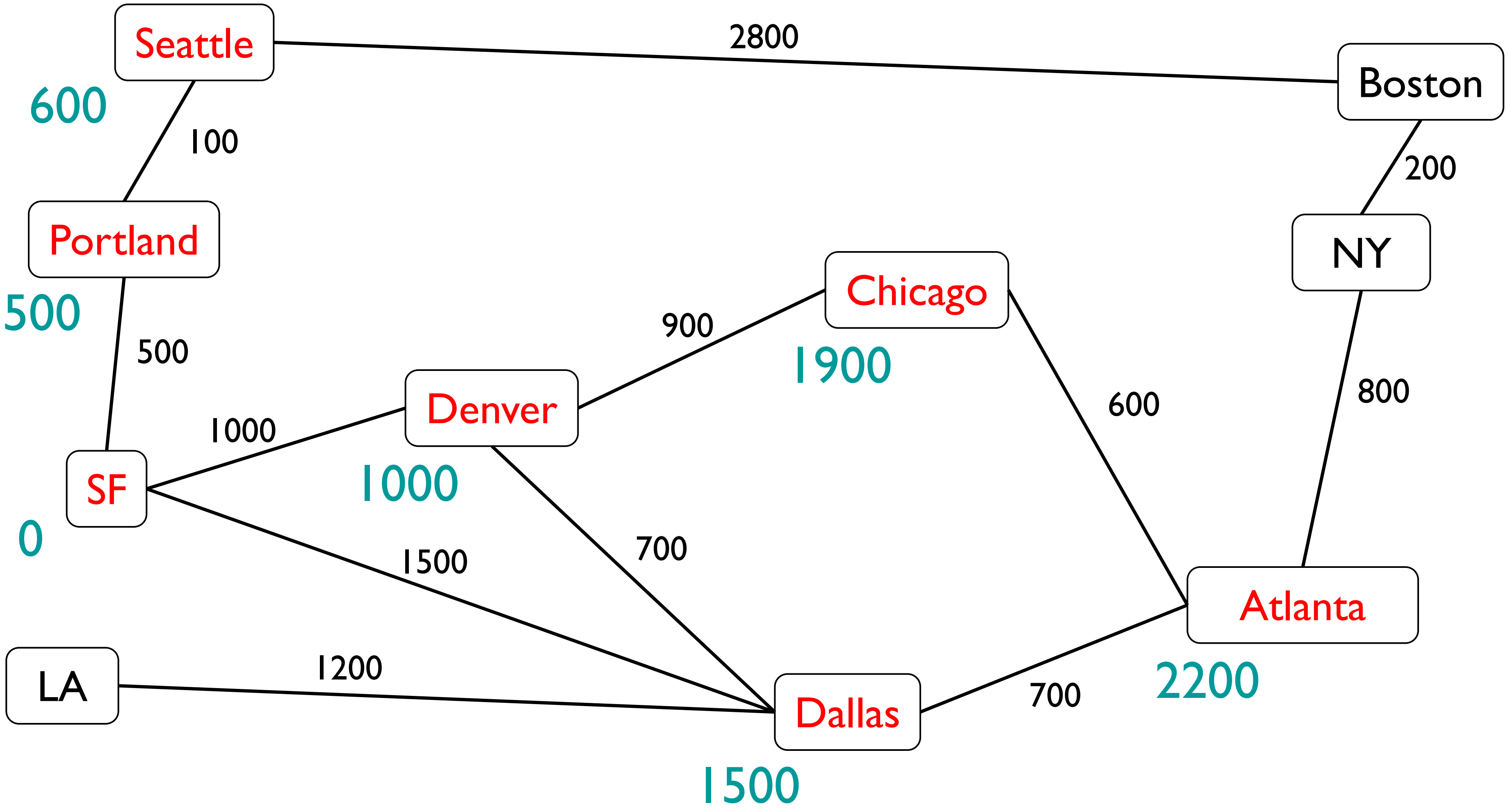
SF->Dal->LA; 2700

SF->Port->Sea->Bos 3400



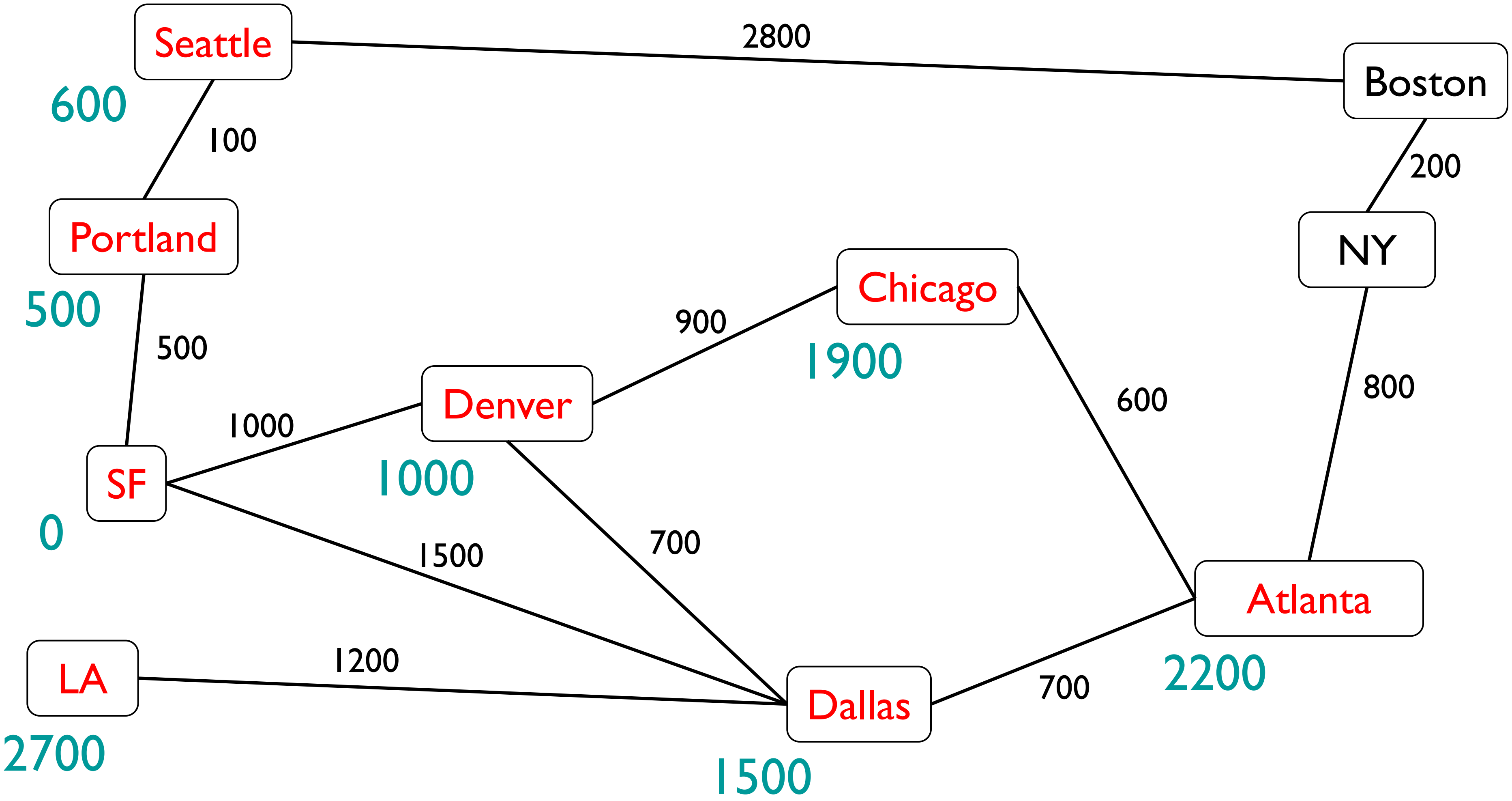
Current: 2200 SF->Dal->Atl

⇨ 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

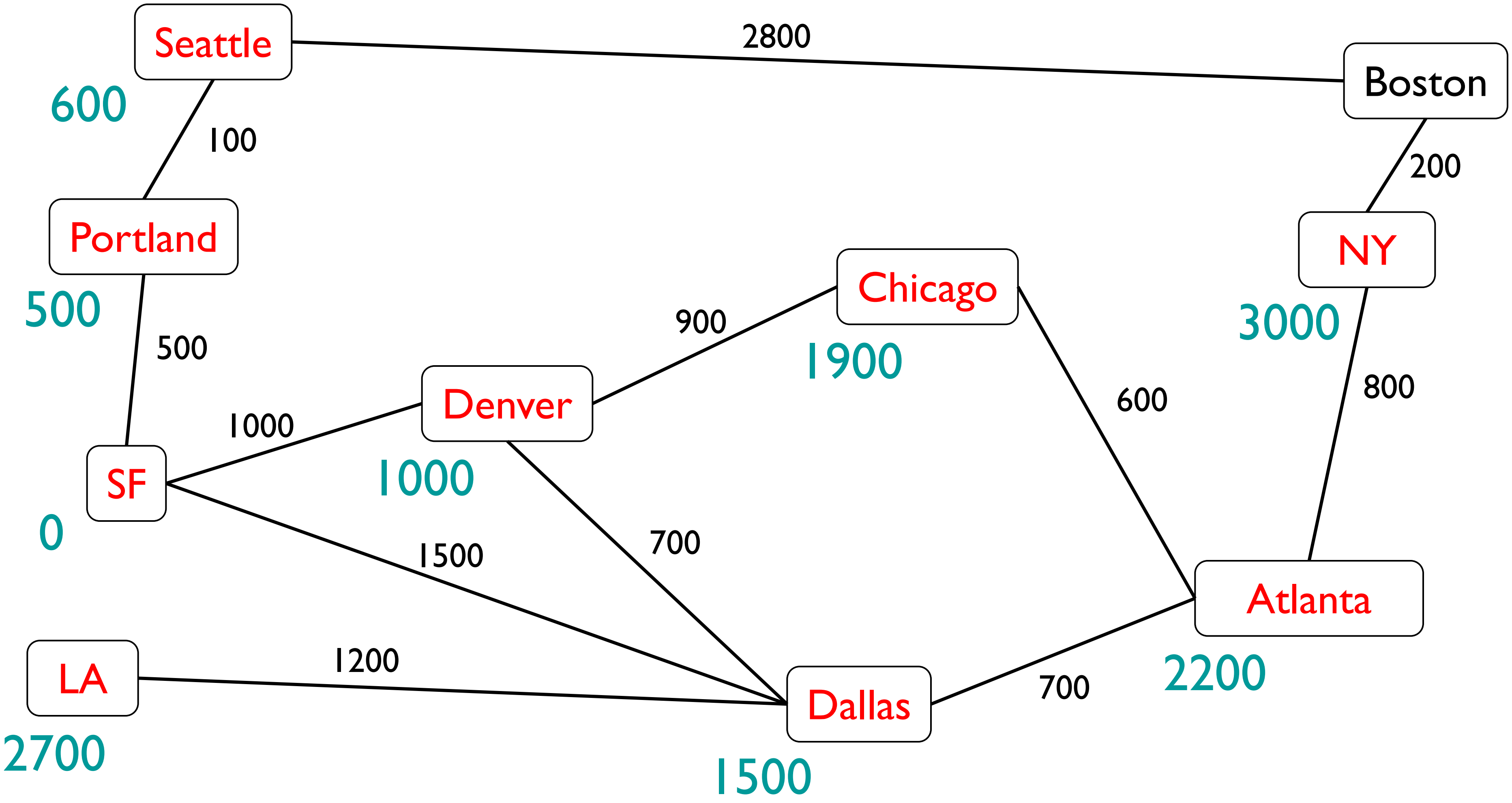
⇨ SF->Dal->LA; 2700 SF->Dal->Atl->NY; 3000 SF->Port->Sea->Bos 3400



Current: 2700 SF->Dal->LA

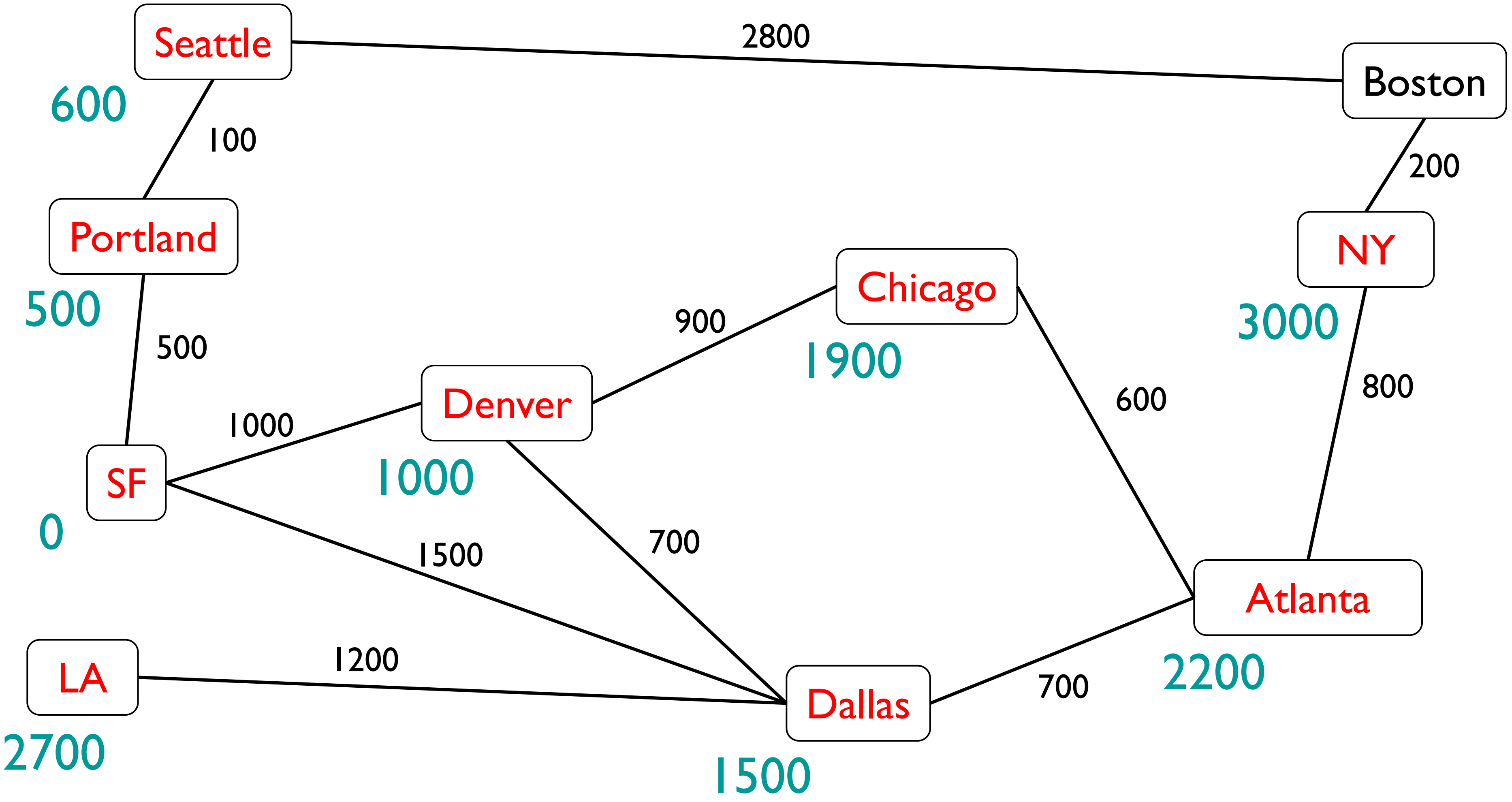
⇨ SF->Dal->Atl->NY; 3000

SF->Port->Sea->Bos 3400



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

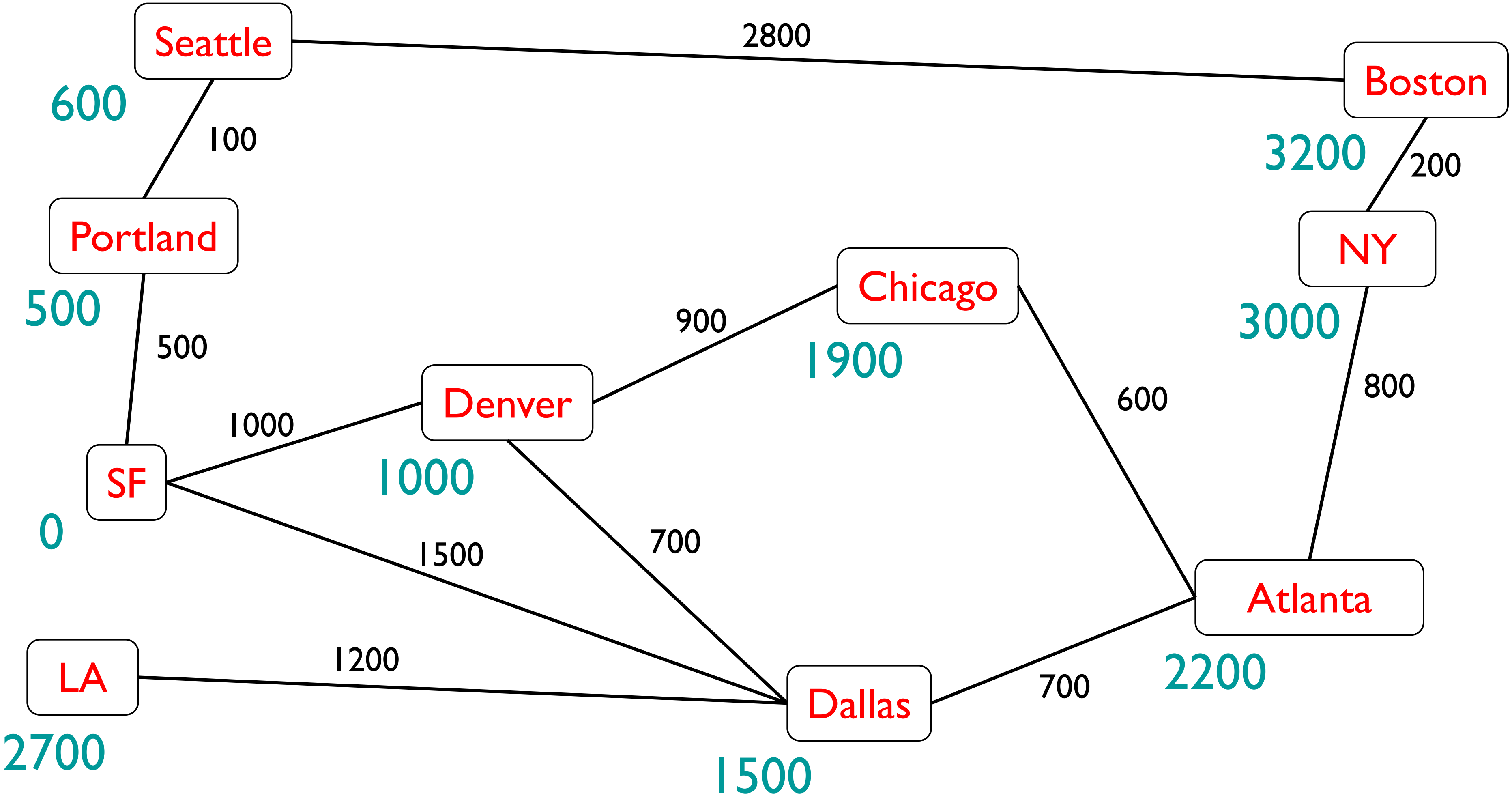
⇨ SF->Port->Sea->Bos
3400



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

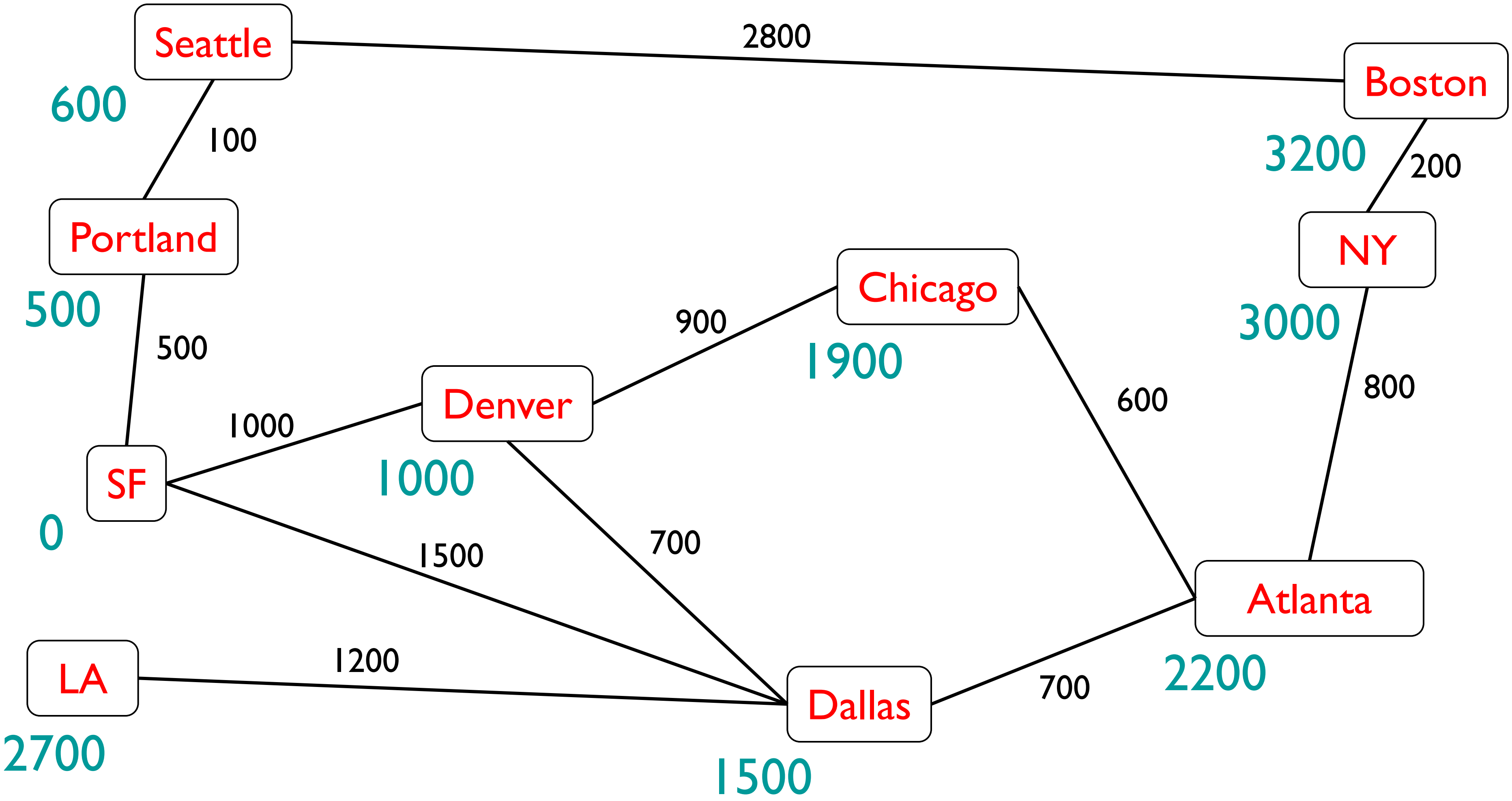
➡ SF->Dal->Atl->NY->Bos; 3200

SF->Port->Sea->Bos 3400

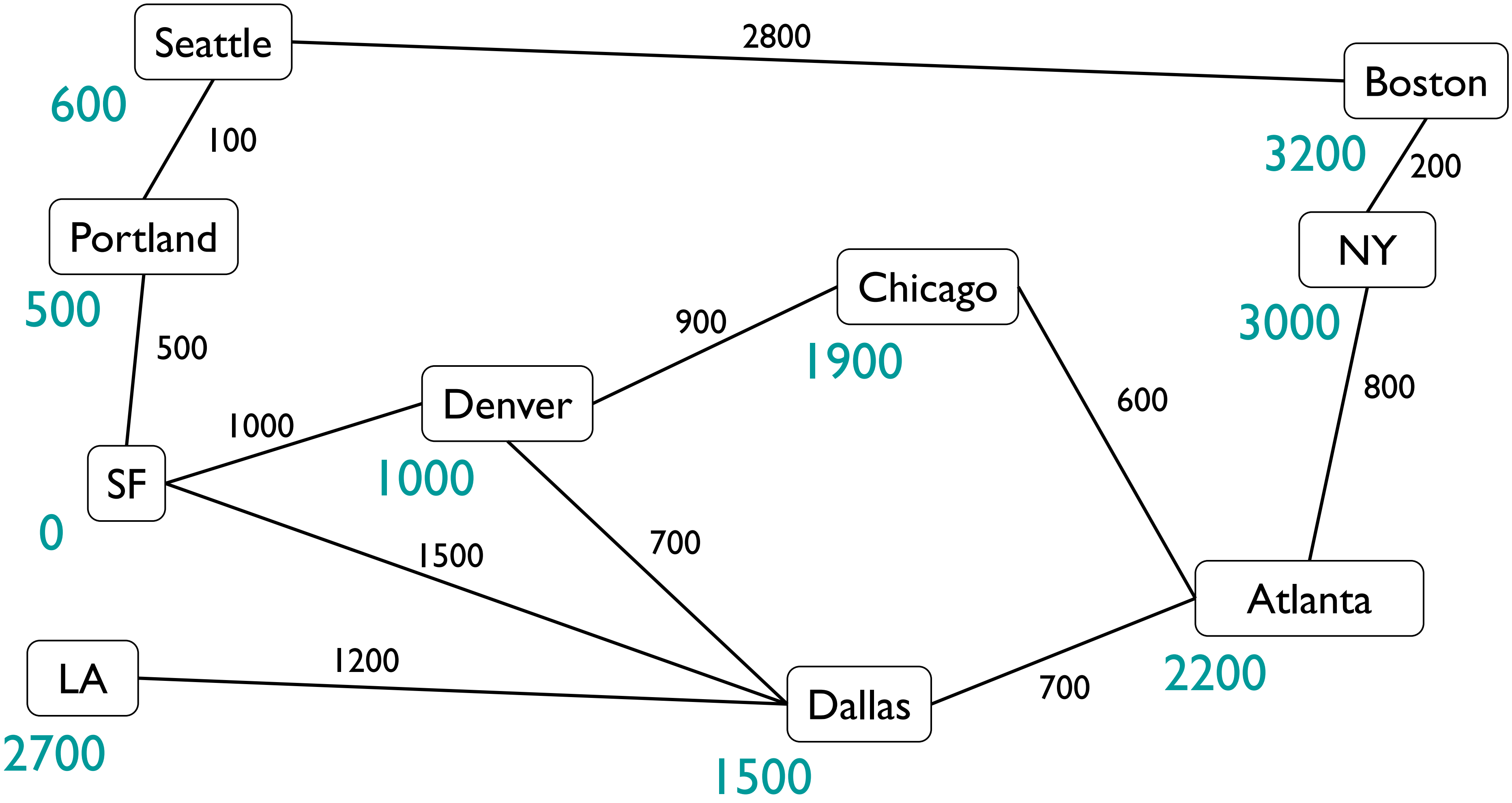


Current: 3200 SF->Dal->Atl->NY->Bos

➡ SF->Port->Sea->Bos
3400



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


Seattle
600
2800
Boston
100
3200
200
Portland
NY
500
Chicago
3000
900
500
1900
Denver
600
800
1000
SF
1000
0
700
1500
Atlanta
LA
1200
700
2200
Dallas
2700
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
- Map: O(|V|)
- 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/removed at most once!
  - Total PQ operation cost is O(|E| log |E|) time
    - Which is O(|E| log |V|) time
      - Since $\log |E| < \log |V|^2 = 2 \log |V|$
  - All other operations take constant time
- Thus time complexity is O(|V| + |E| log |V|)

Summary & Observation

Dijkstra's Algorithm is a highly efficient method for solving shortest path problems

- Employs a relatively simple *greedy algorithm*
- A variant of this algorithm used to be the method used for routing internet traffic
- Faster algorithms are much more complex
- Uses Priority Queue to avoid sorting
- Works on undirected graphs, too

Like this kind of thing? Consider Csci 256!