Shortest Paths:

Dijkstra's Algorithm.
A* Search.

Shortest Paths

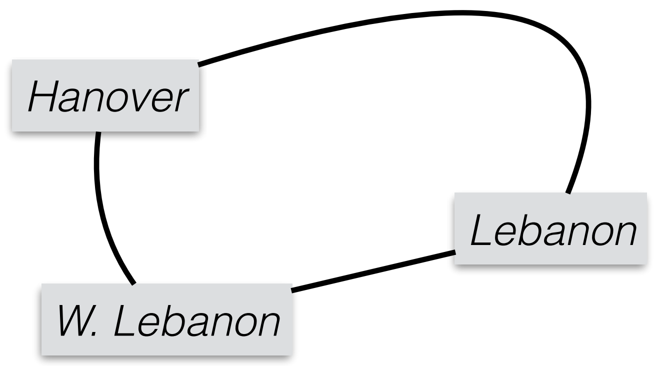
- **BFS** is good for finding a path between two vertices with as few edges as possible!
- Well, when "all edges are created equal" at least…
- Consider a road map (or Google Maps :) — want the shortest path between two points
  - ex. Hanover - Boston
  - Not all paths are *reasonable* (i.e., up into Maine/Canada)
  - Not all paths are "equal"
- < Enter ***Weighted Graphs*** >

WWW.FREEWORLDMAPS.NET
CANADA
Québec
Ottawa
Montréal
MAINE
Augusta
VERMONT
Montpelier
Toronto
NEW HAMPSHIRE
Concord
Rochester
Syracuse
Buffalo
Albany
NEW YORK
Boston
MASSACHUSETTS
Hartford
Providence
CONNECTICUT
RHODE ISLAND
New Haven
Stamford
PENNSYLVANIA
Newark
New York
Pittsburgh
Allentown
Harrisburg
Trenton
Philadelphia
NEW
MARYLAND
Atlantic

http://www.freeworldmaps.net/

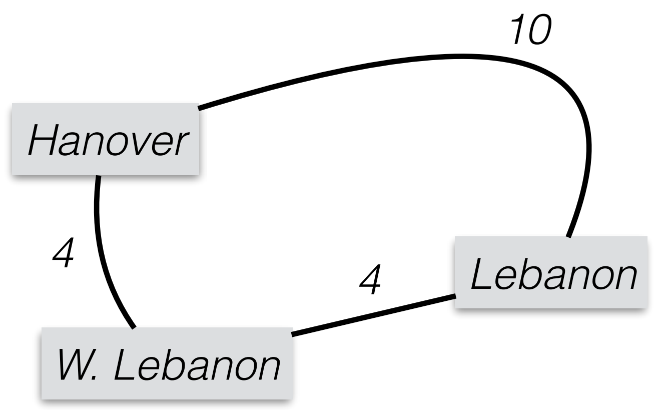
Weighted Graphs

- Rather than each edge being treated equally…



Weighted Graphs

- Rather than each edge being treated equally, we apply **weights** to the edges. (i.e., *e = (u,v) and w(u,v) = w(e))*

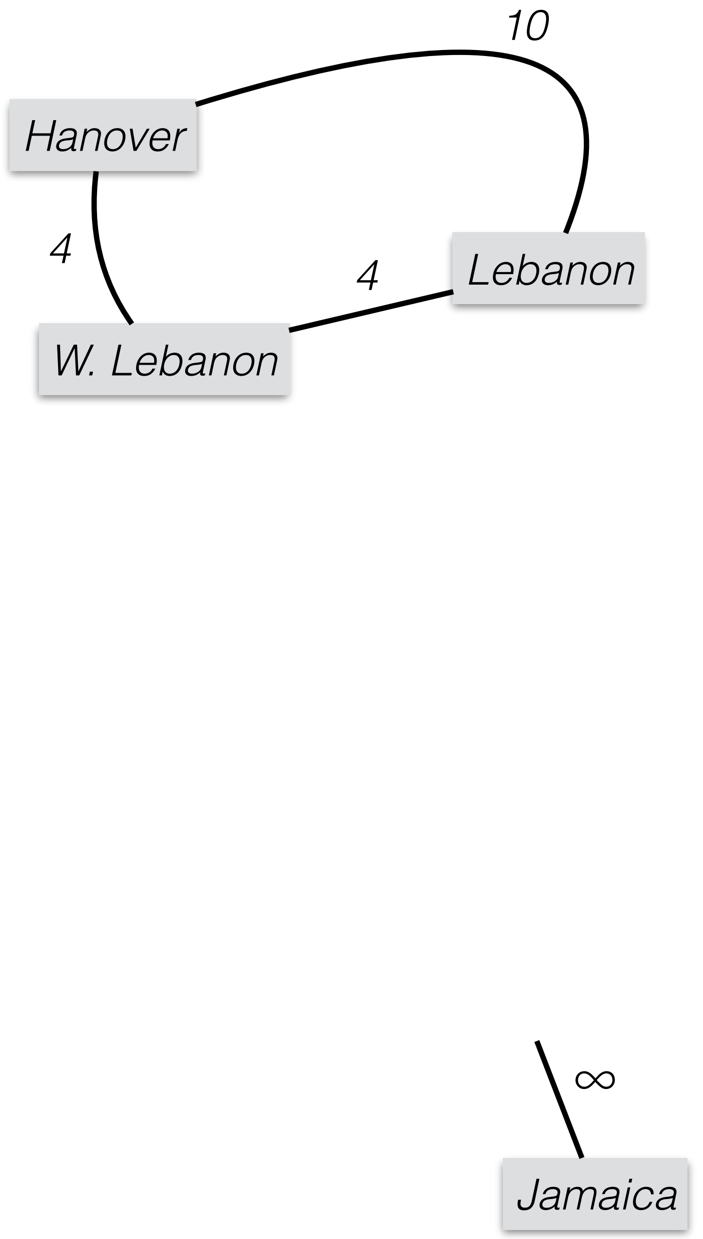


Weighted Graphs

- Rather than each edge being treated equally, we apply **weights** to the edges. (i.e., *e = (u,v) and w(u,v) = w(e)).*
- The ***distance*** of a path is now defined not by the *number of edges* in a particular path, but rather, the *sum of the weights of the edges* in a path.
- ***Shortest path*** = path between *u* and *v* with the smallest weighted sum.

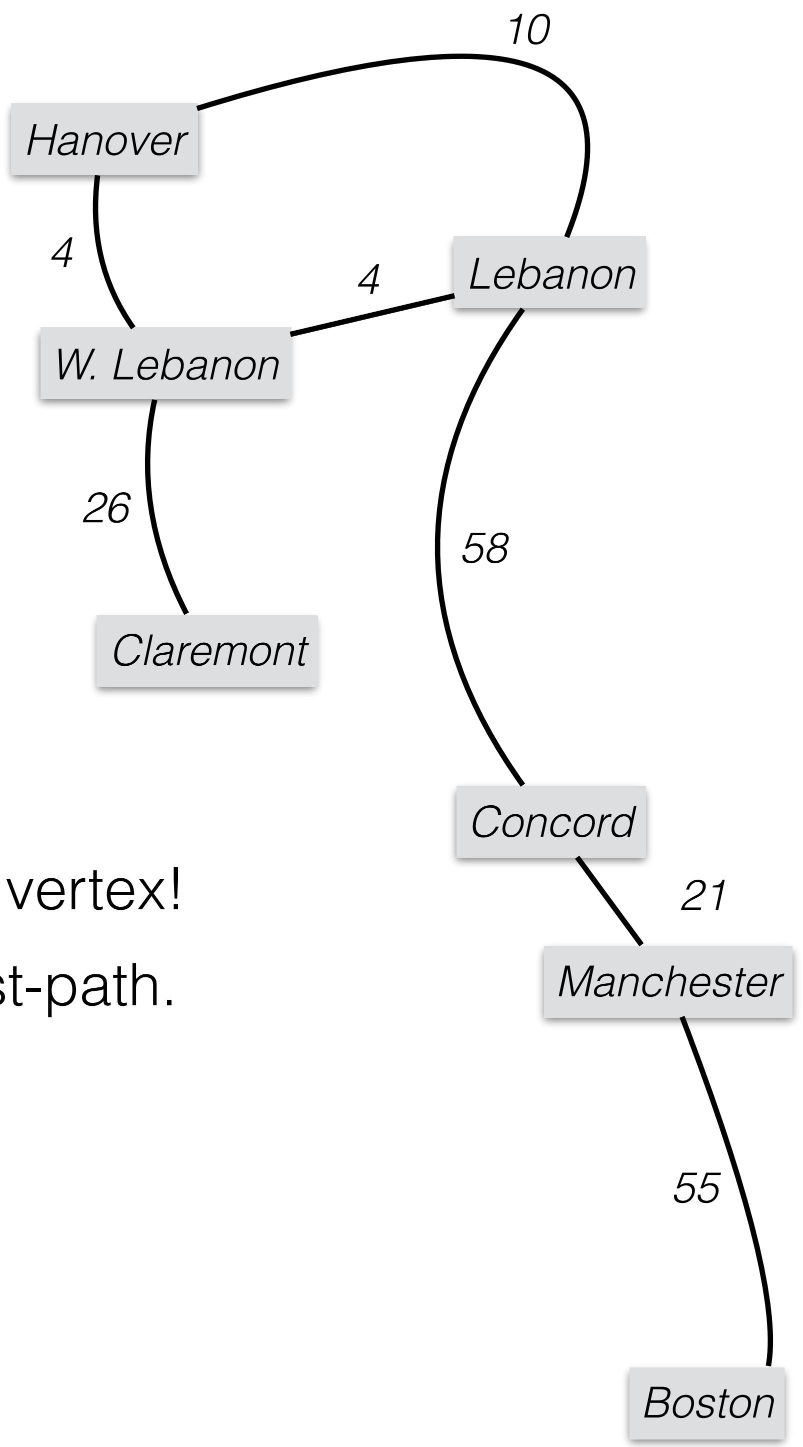
Notes

- Commonly use d(u,v) = ∞ if no path exists between u & v.
- Negative weights complicate things… so we don't talk about that here…



Dijkstra’s Algorithm

- Dijkstra’s Alg. == “greedy,” weighted BFS starting at vertex *S*.
  - single-source shortest path (i.e., start at 1 point only).
  - Non-negative edge weights.
- Greedy algorithm
  - I.e., at each step, take “best” choice (no look-ahead).
  - Sometimes greedy approach is good (optimal)…Dijkstra’s!
- How does it work?!
  - Like BFS, grow a “cloud”, where
    - distance from start vertex is known
    - at each step, add one vertex to the cloud
    - when all vertices are added, we know shortest distance to any vertex!
      - Note: it is easy (and often useful) to construct/display shortest-path.



Dijkstra's Algorithm

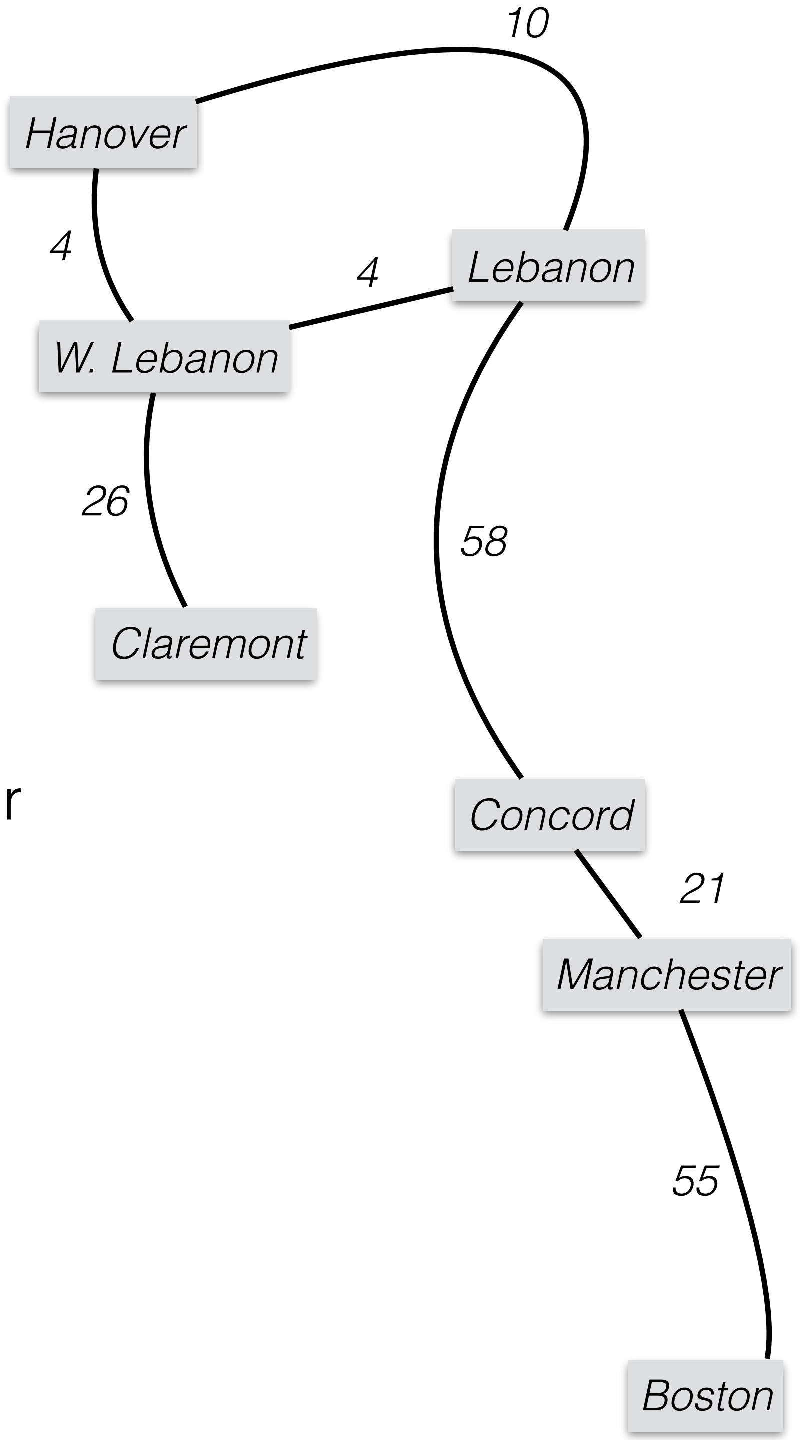
(cont.)

- Use a PQ —
  - keep track of distance from start to all vertices
  - key = distance to start vertex (so far)
  - begin with *start* having distance 0; all others have distance ∞
- Initialize "cloud" (initially empty)
- At each step —
  - remove min. vertex (v) + add to "cloud" w/ min. distance as key
  - check all vertices (v') adjacent to v to see if we've found a shorter path — ***relaxation***.

Edge Relaxation

```
if D[u] + w(u,v) < D[v] then
    D[v] = D[u] + w(u,v)
```

galleryhip.com

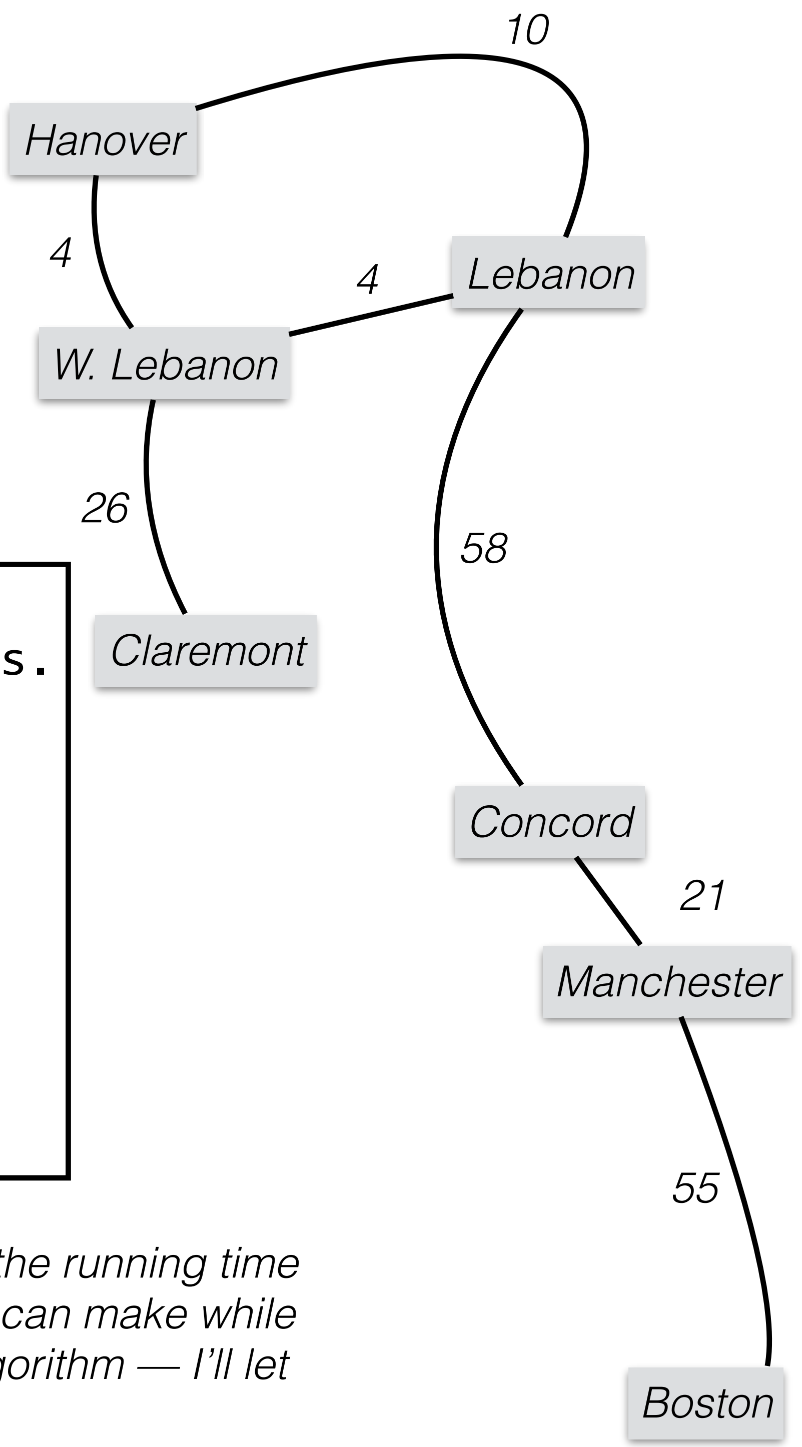


Dijkstra's Algorithm

Input: a graph (G) w/ non-negative edge weights and a start vertex, *s*.

Output: The length of a shortest path from s to v for each vertex of G

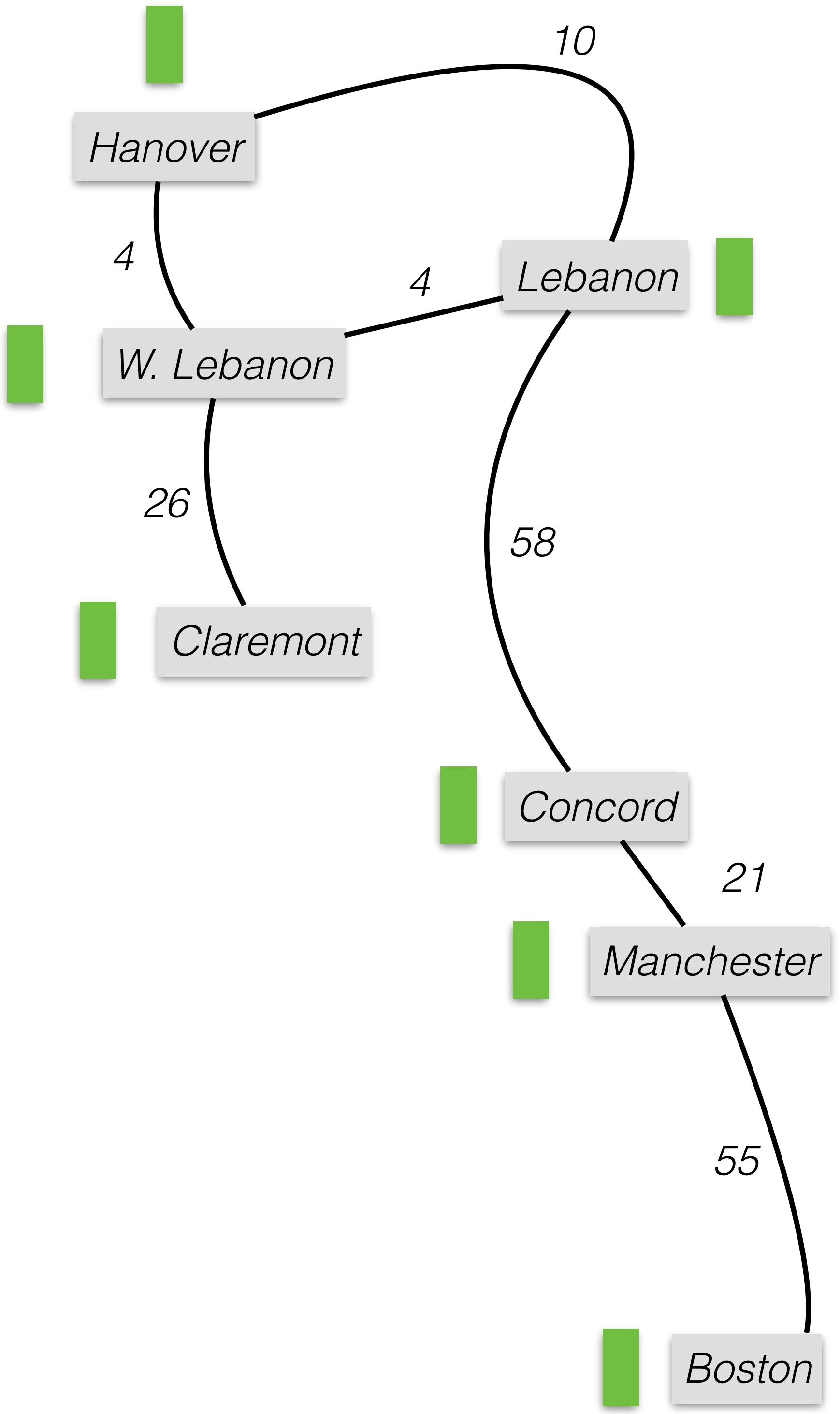
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



The book and the online notes have an interesting discussion about the running time of Dijkstra's algorithm. There are lots of interesting tradeoffs that one can make while implementing the various data structures that are needed for this algorithm — I'll let you explore those more!

Dijkstra's Algorithm

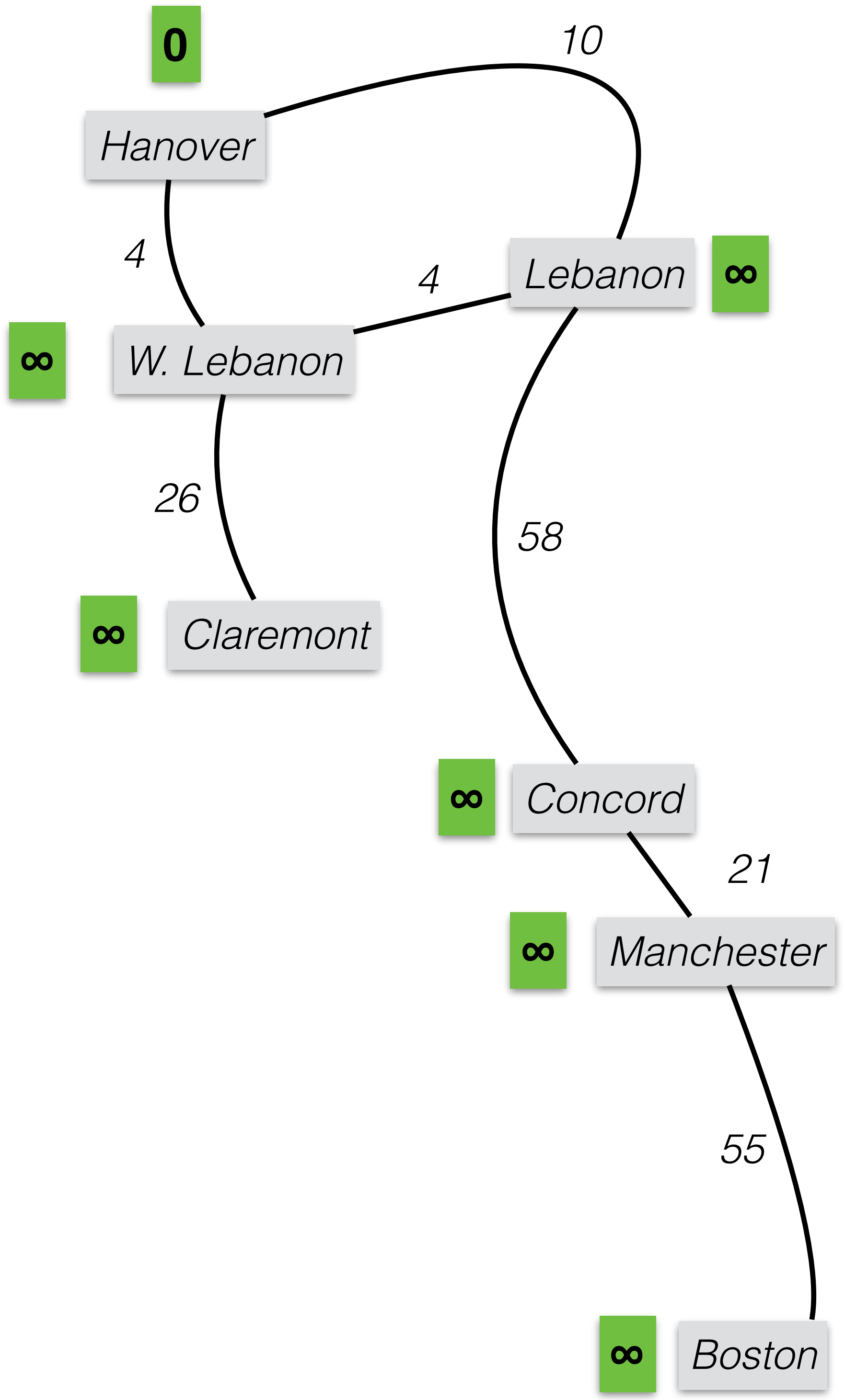
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Dijkstra’s Algorithm

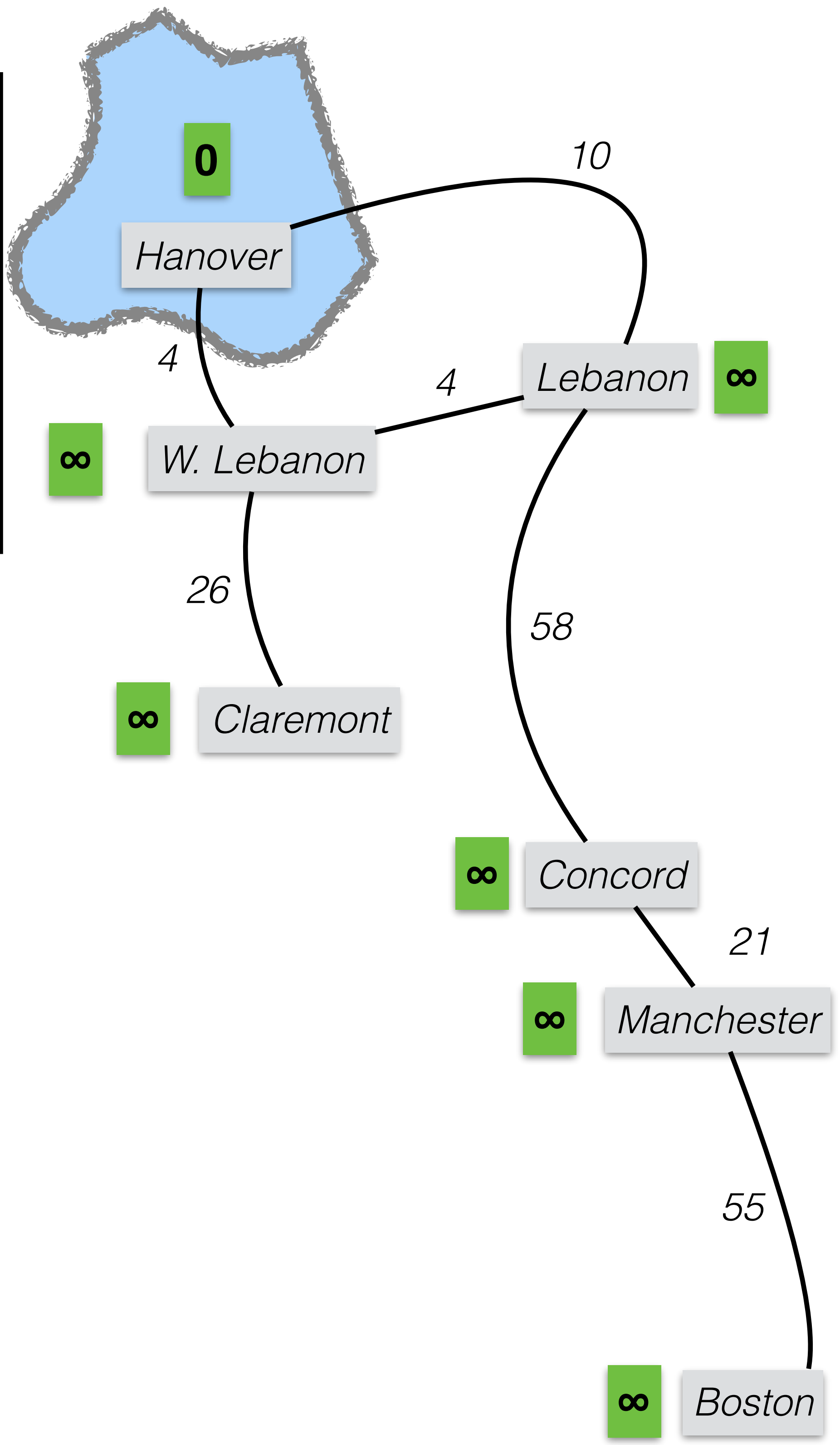
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```

Initialization



Dijkstra’s Algorithm

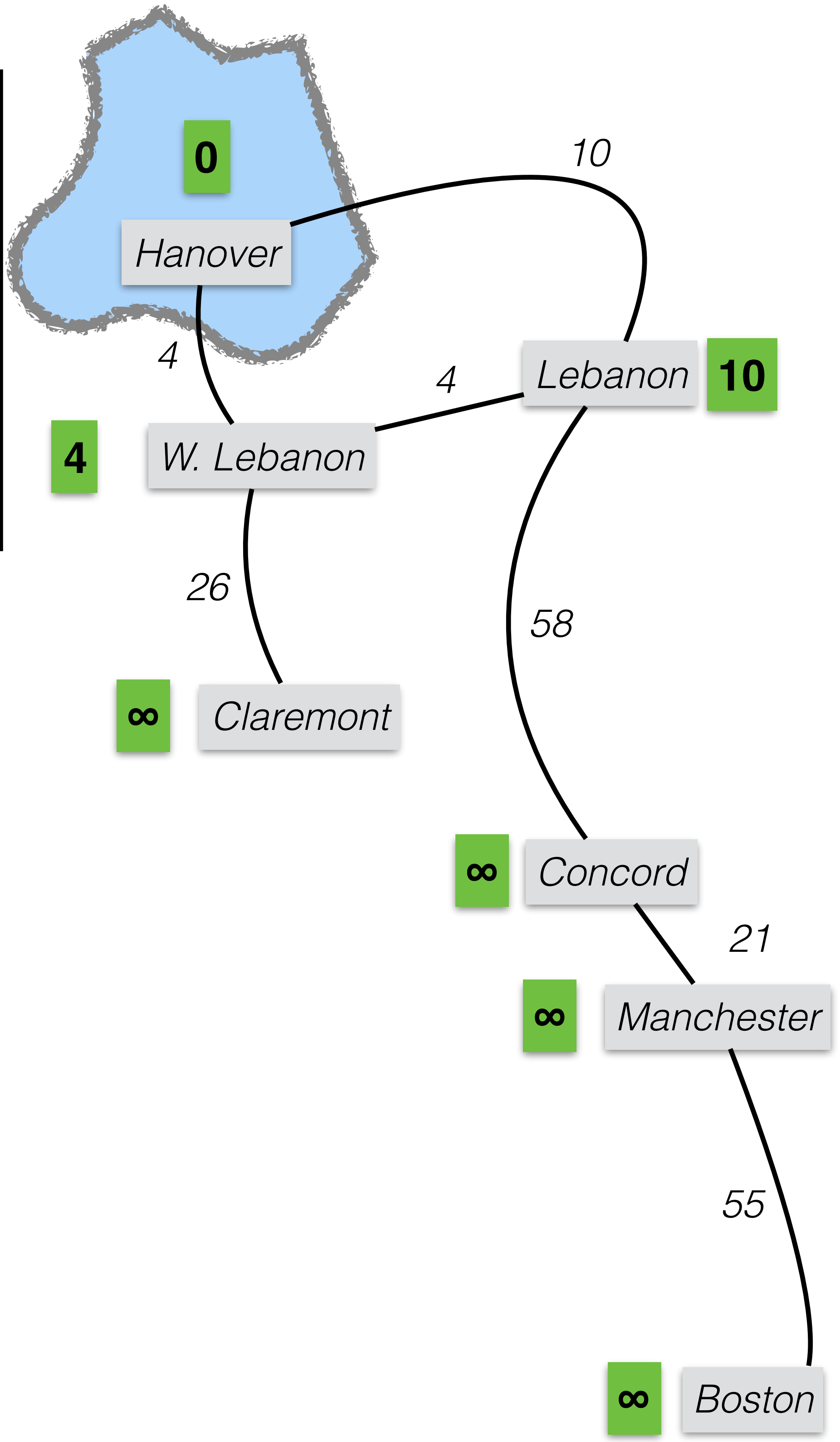
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Q.removeMin()

Dijkstra's Algorithm

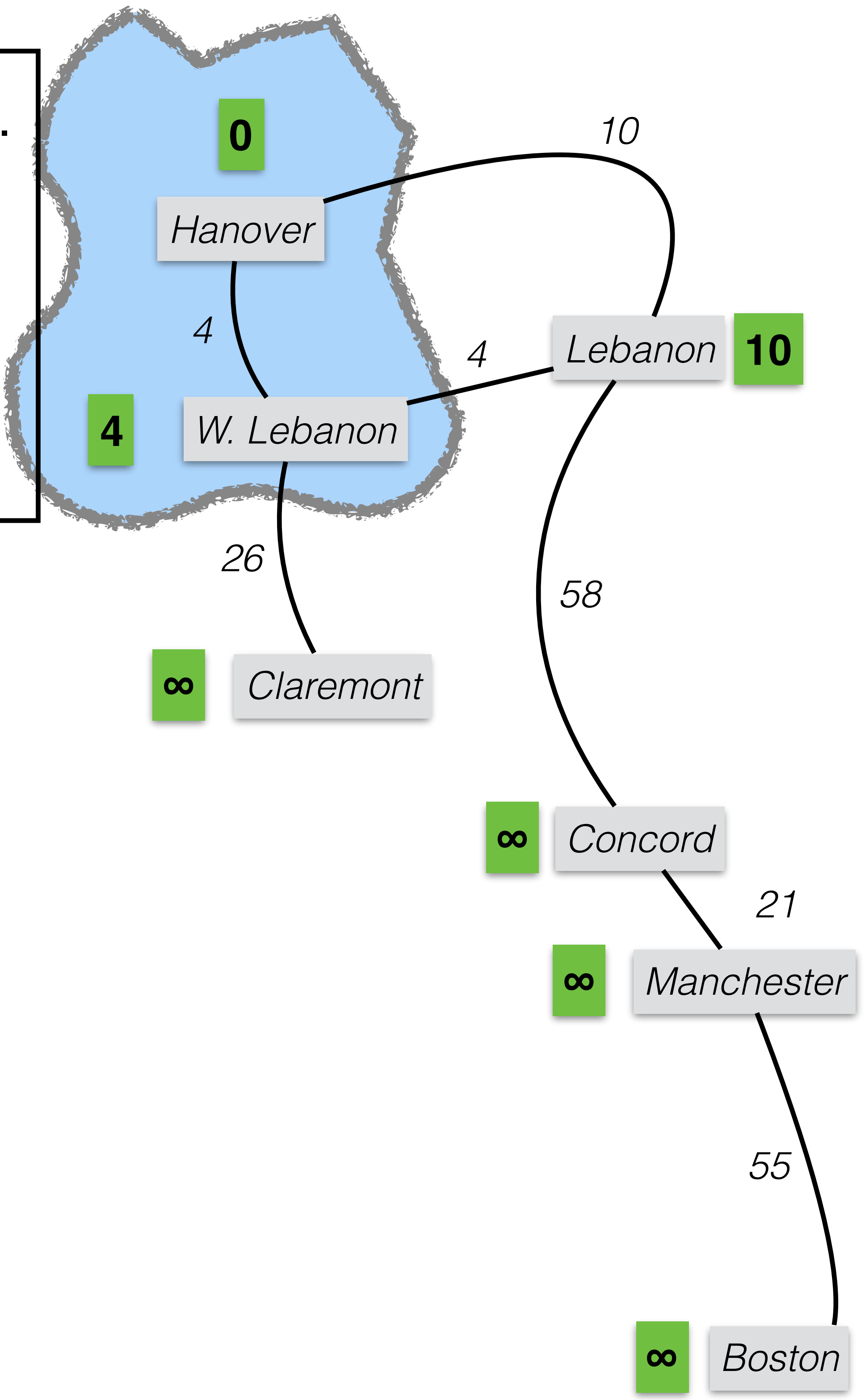
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Relaxation

Dijkstra’s Algorithm

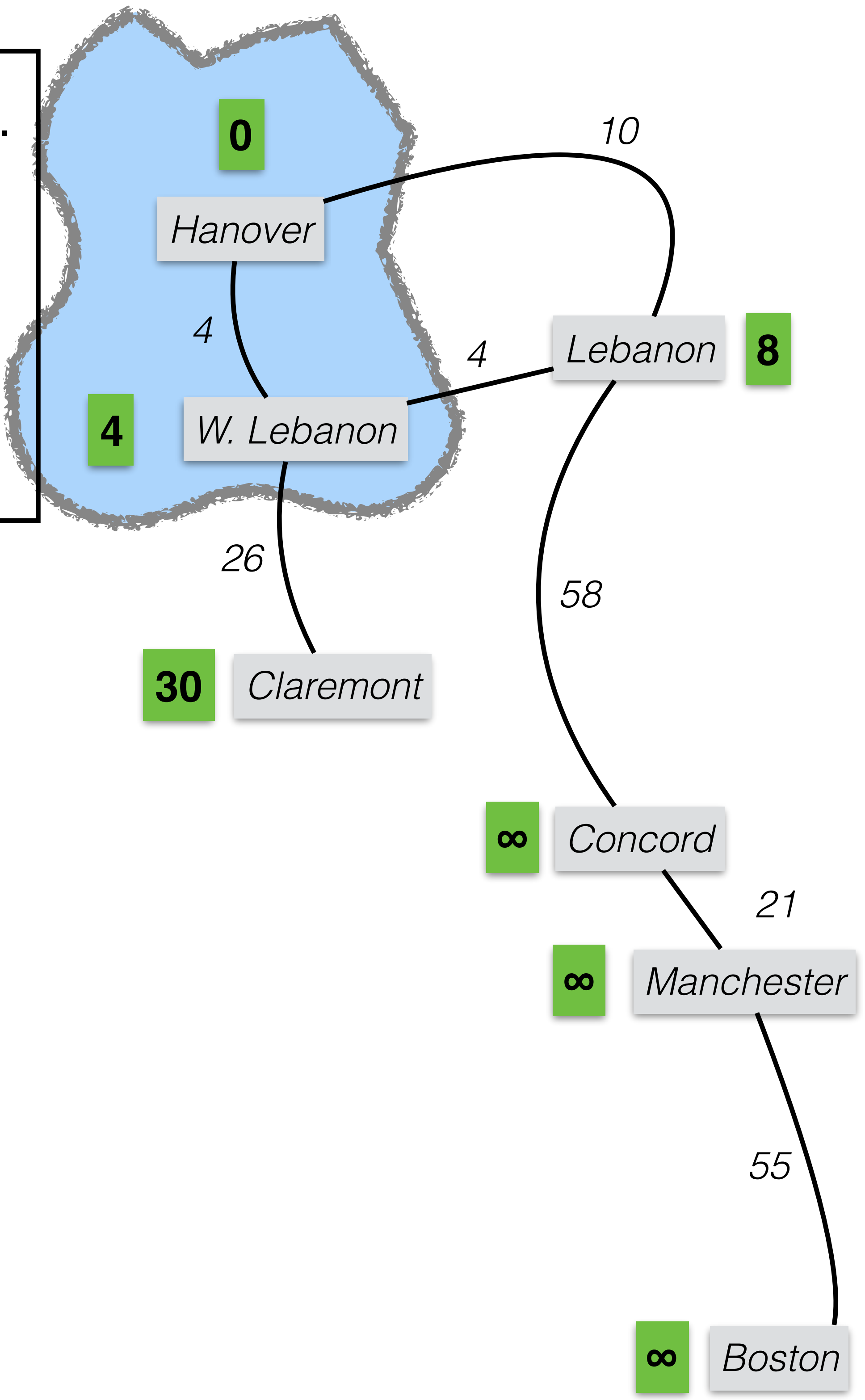
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Q.removeMin()

Dijkstra’s Algorithm

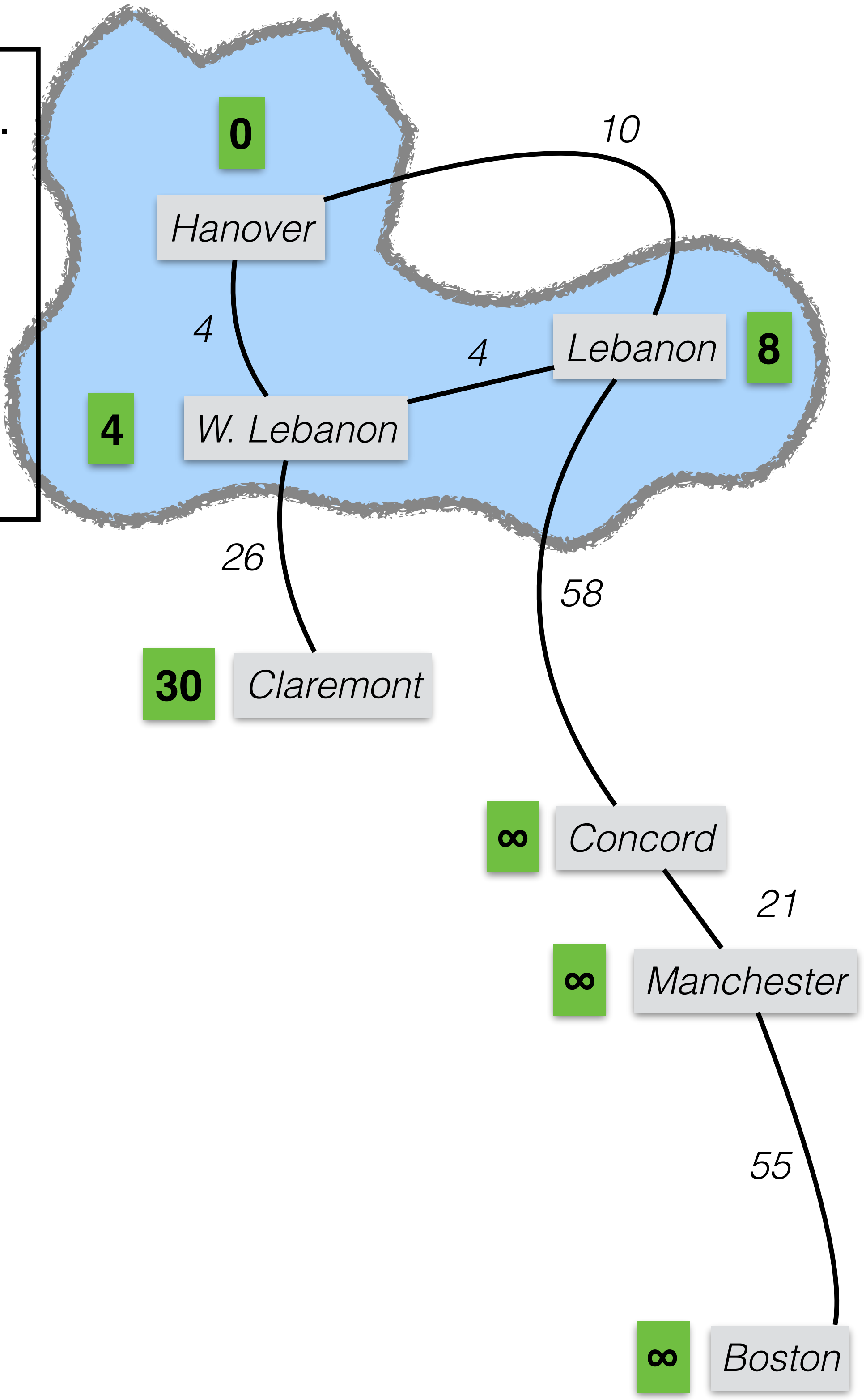
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Relaxation

Dijkstra’s Algorithm

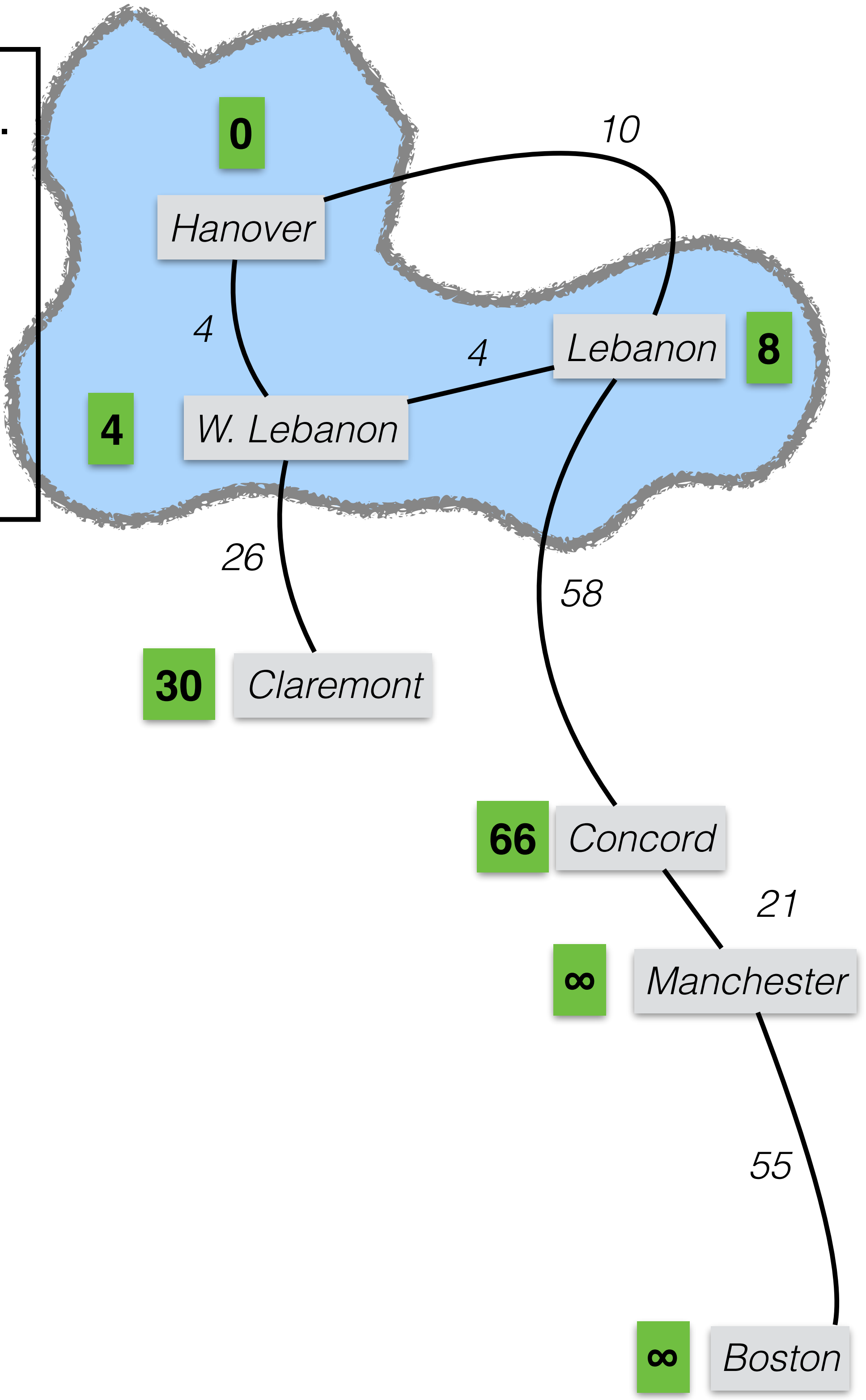
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Q.removeMin()

Dijkstra’s Algorithm

```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```

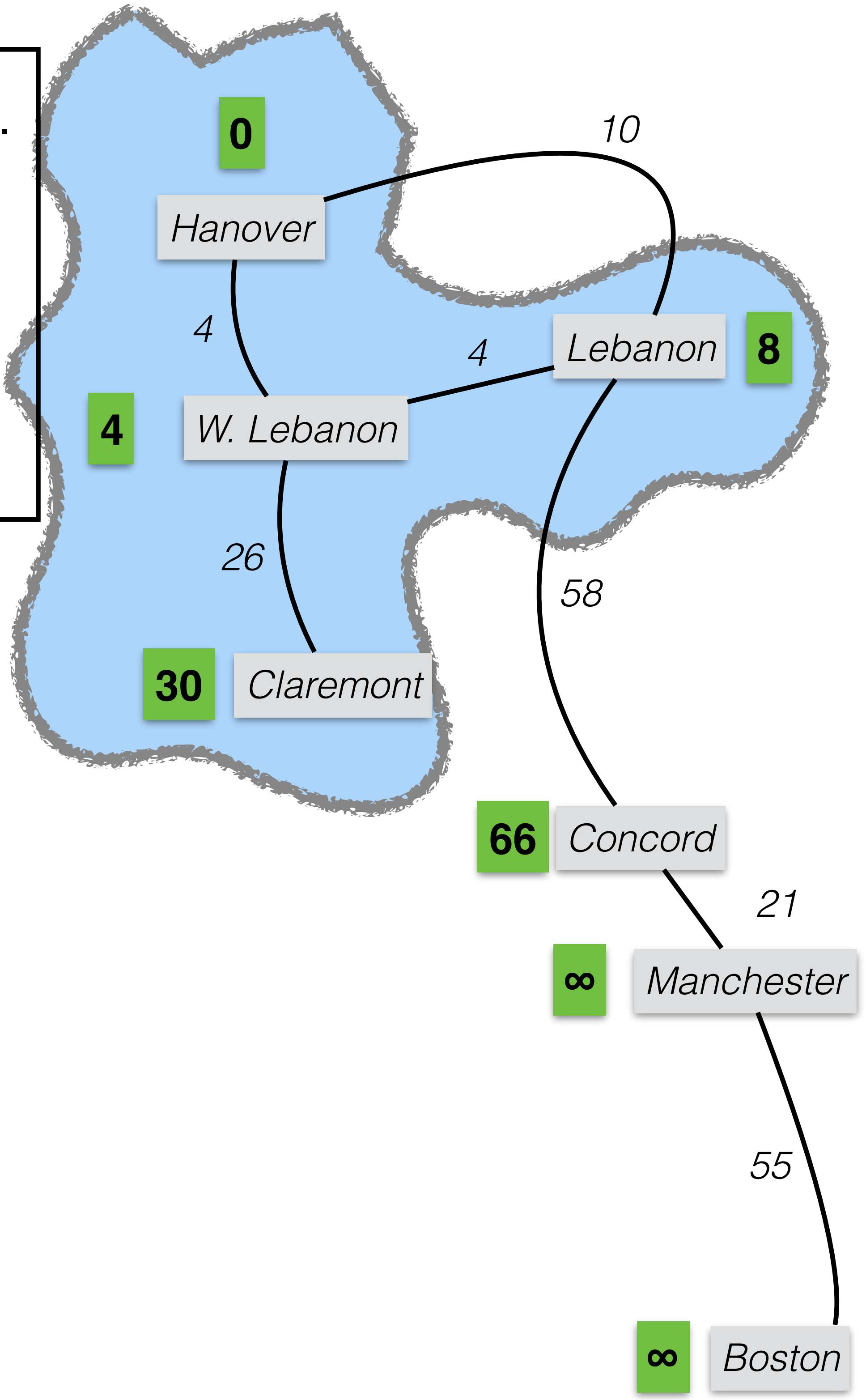


Relaxation

Dijkstra’s Algorithm

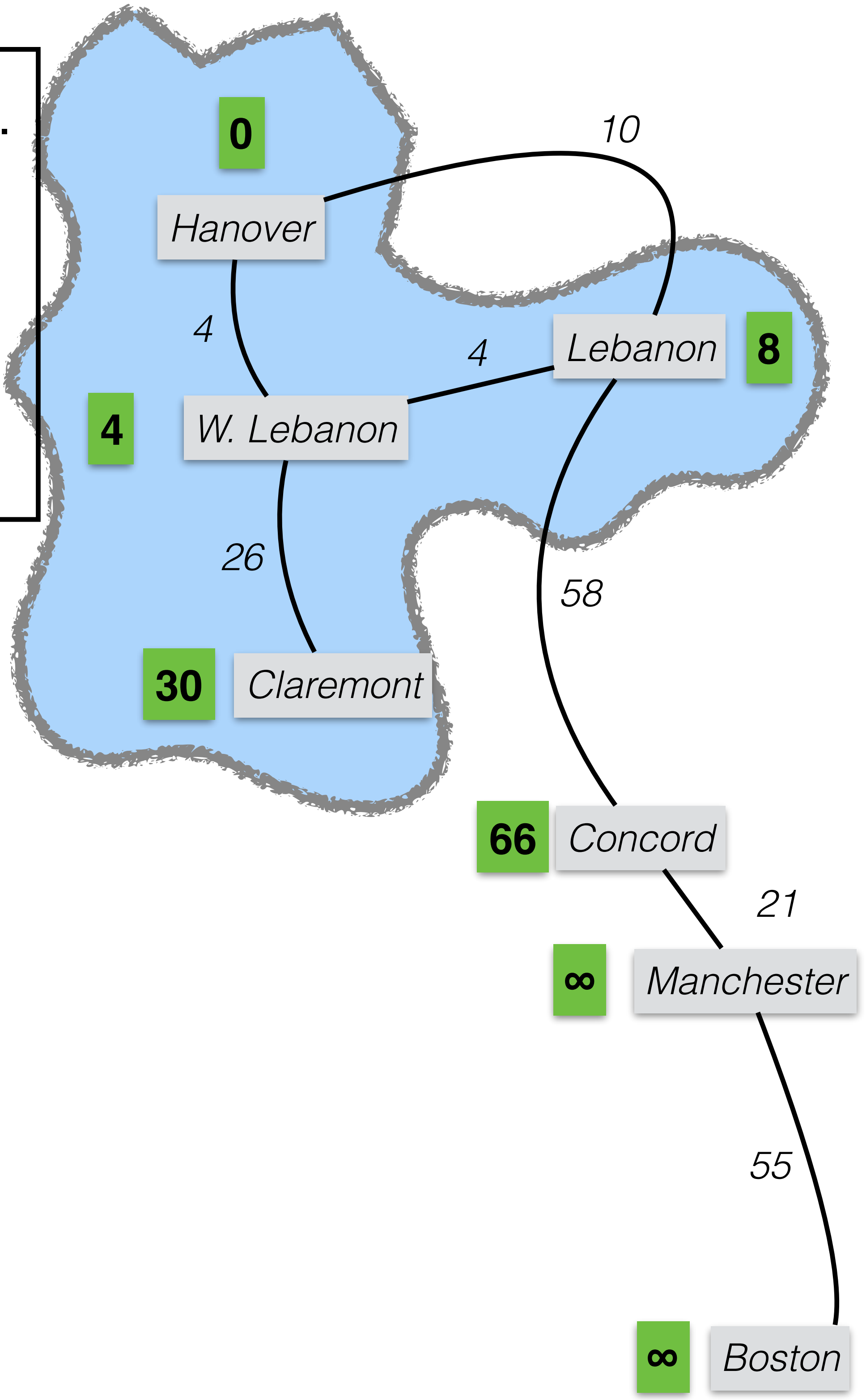
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```

Q.removeMin()



Dijkstra's Algorithm

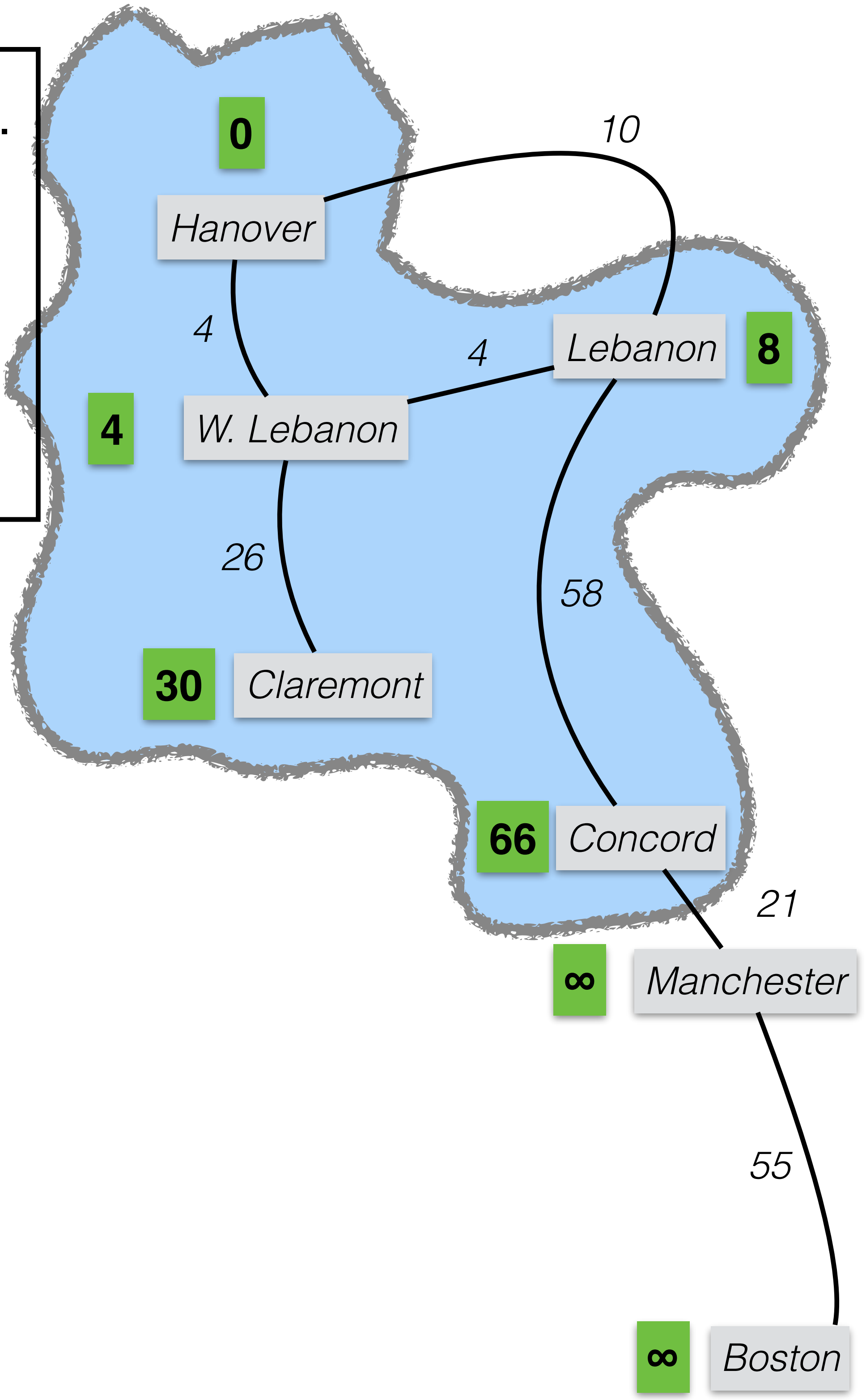
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Relaxation

Dijkstra’s Algorithm

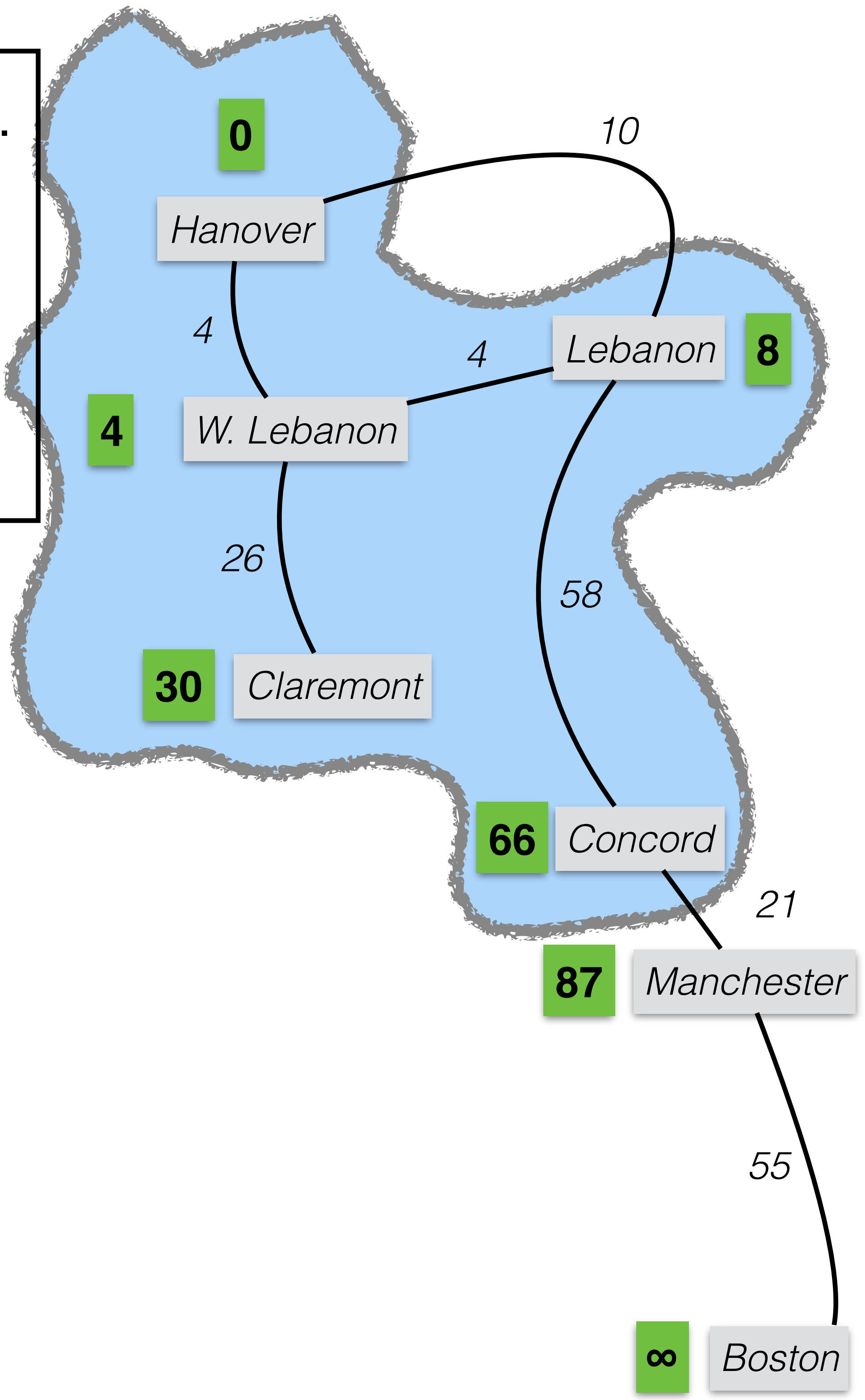
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Q.removeMin()

Dijkstra's Algorithm

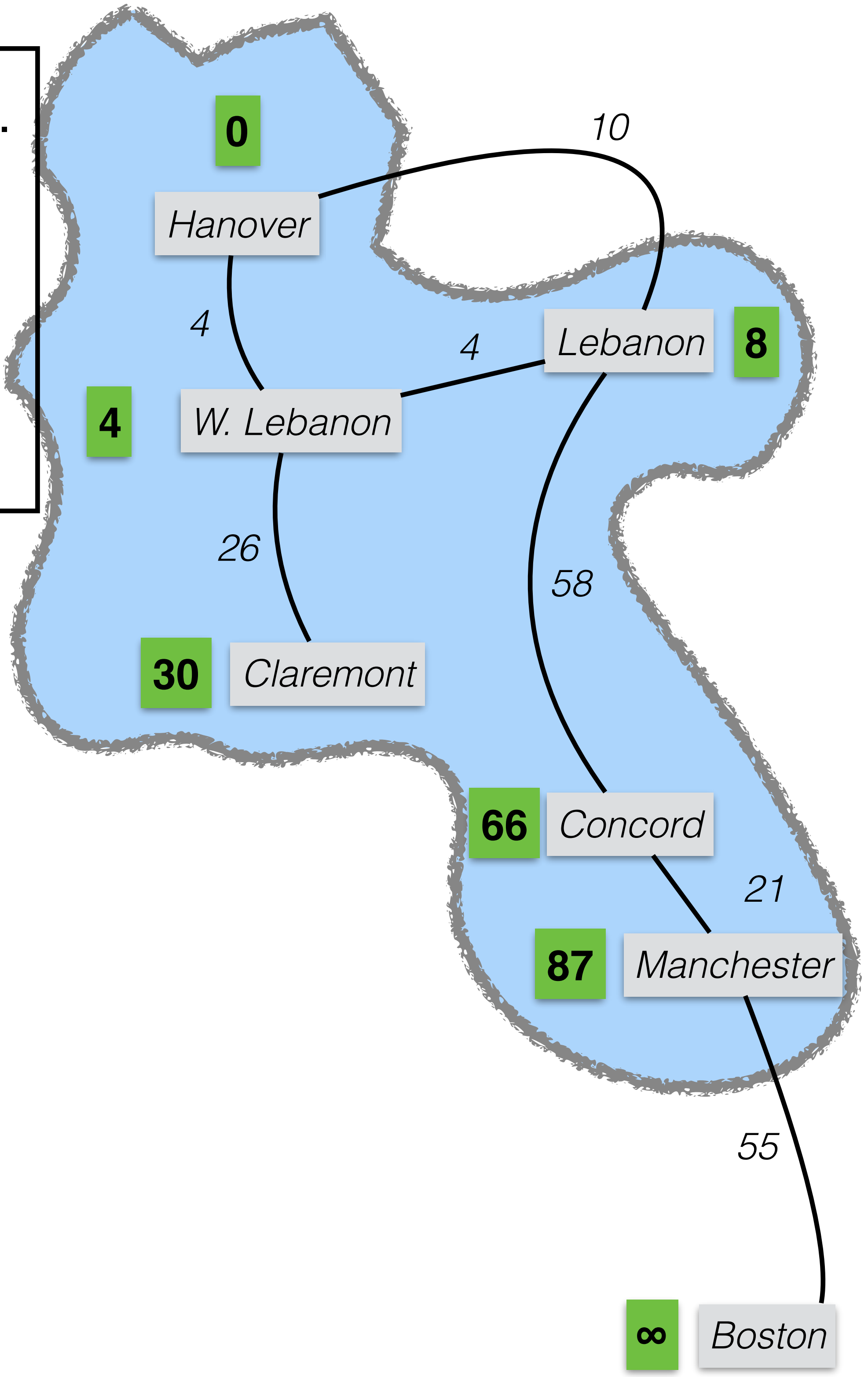
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Relaxation

Dijkstra’s Algorithm

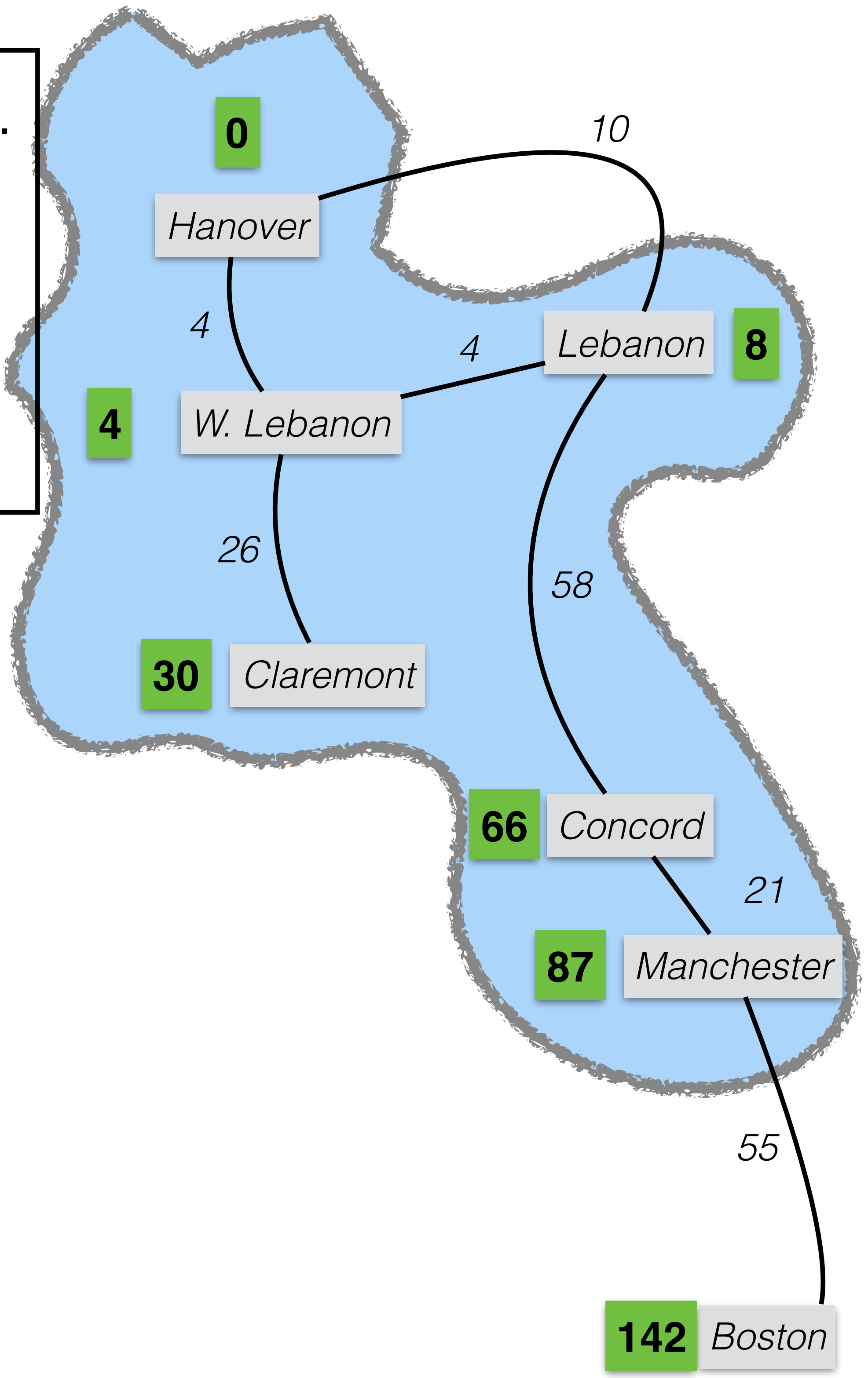
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Q.removeMin()

Dijkstra’s Algorithm

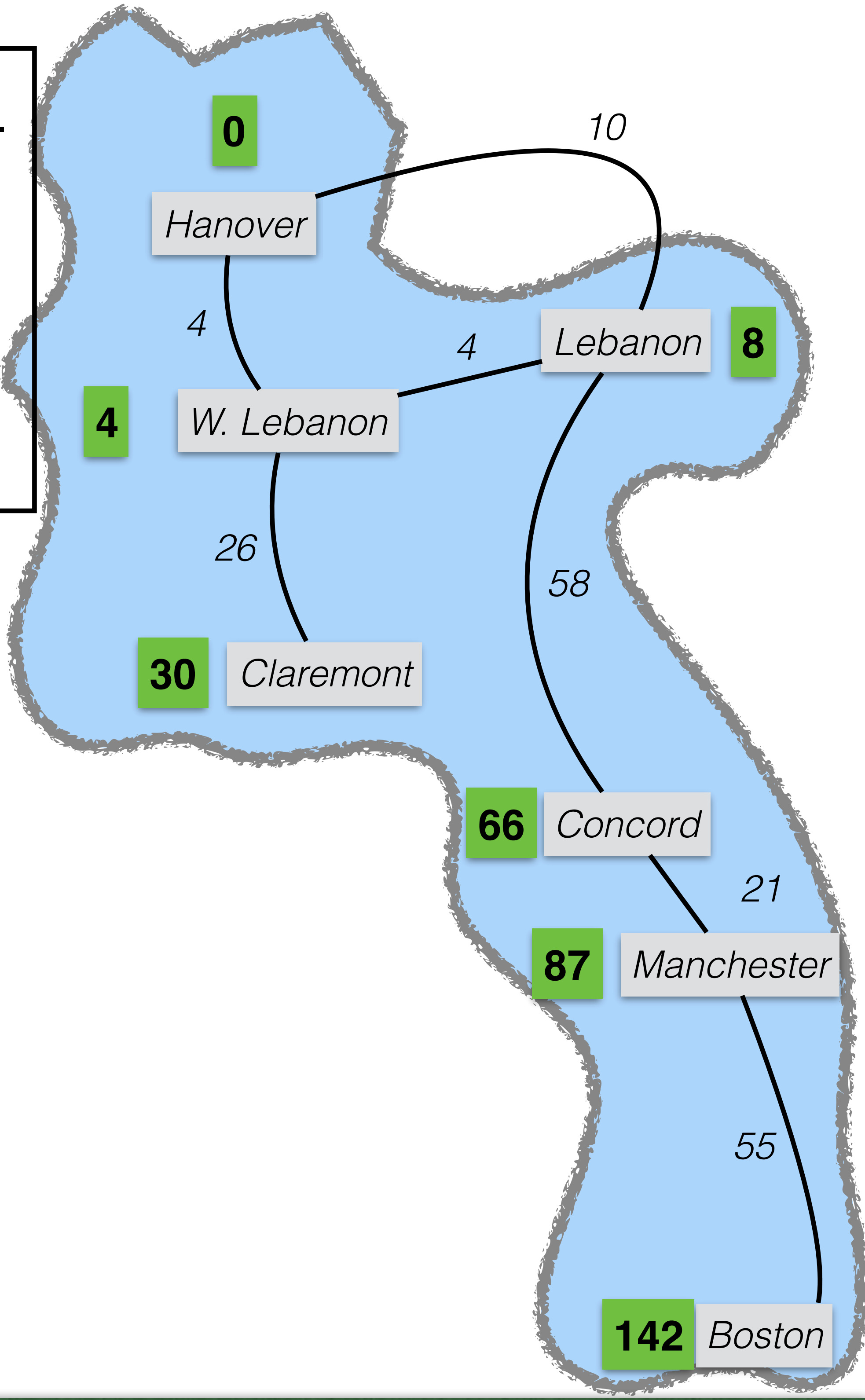
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```



Relaxation

Dijkstra's Algorithm

```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```

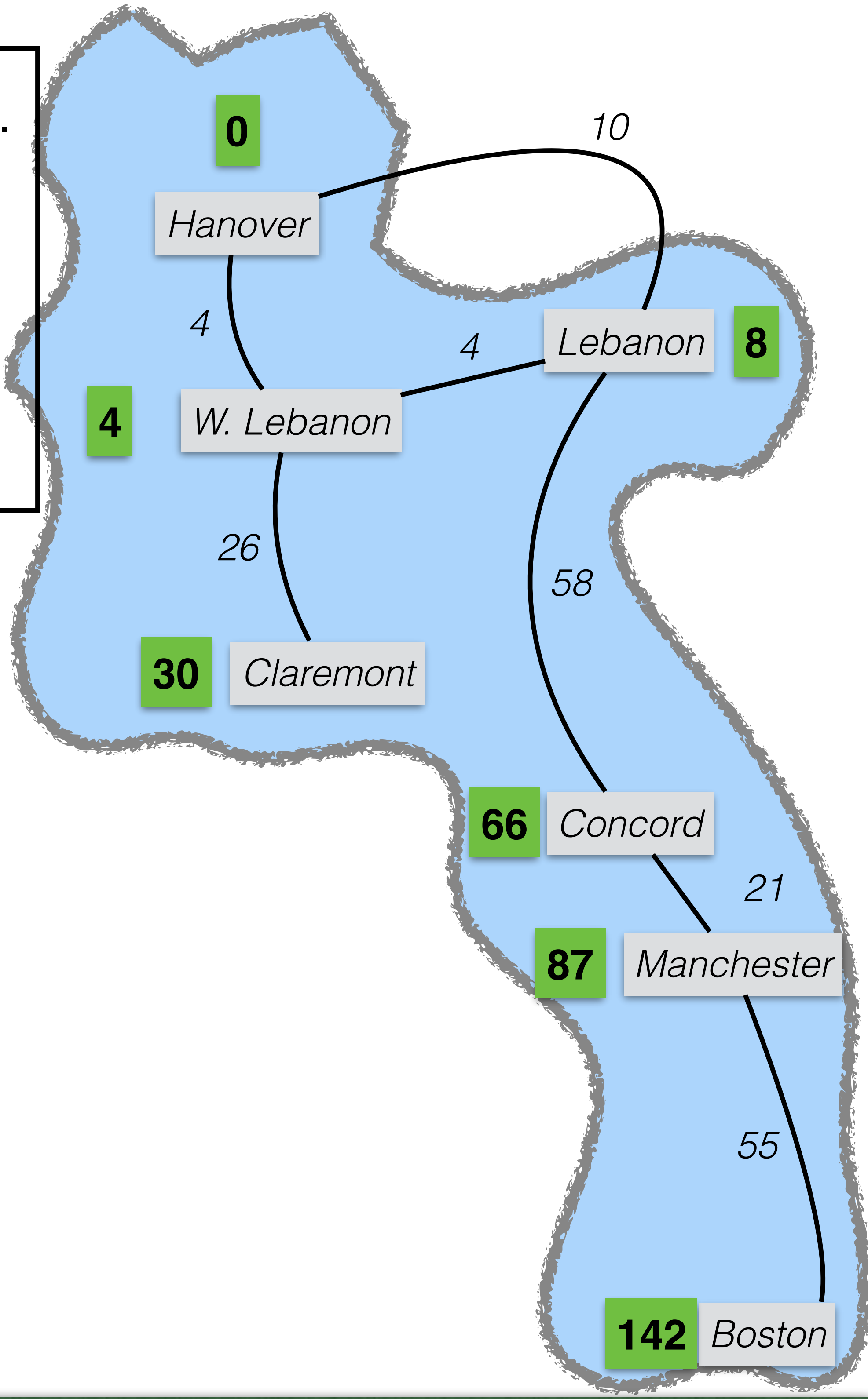


Q.removeMin()

Dijkstra’s Algorithm

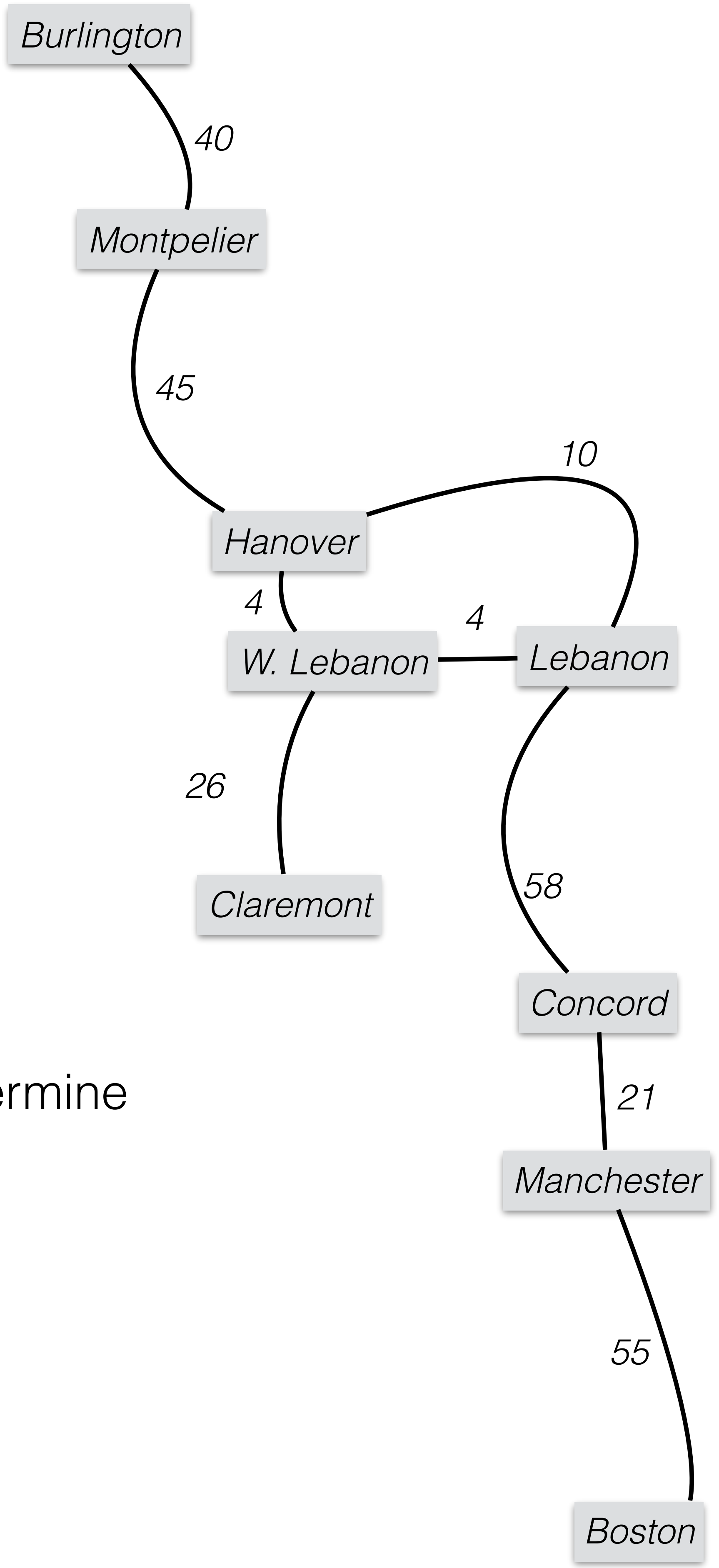
```
Initialize D[s] = 0 and D[v] = ∞ for v != s
Let a PQ (Q) contain all vertices of graph (G) using D labels as keys.
while Q not empty do
  //pull a new vertex u into the cloud
  u = value returned by Q.removeMin()
  for each edge (u,v) s.t. v is in Q do
    //perform the relaxation procedure on edge (u,v)
    if D[u] + w(u,v) < D[v] then
      D[v] = D[u] + w(u,v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```

Done!



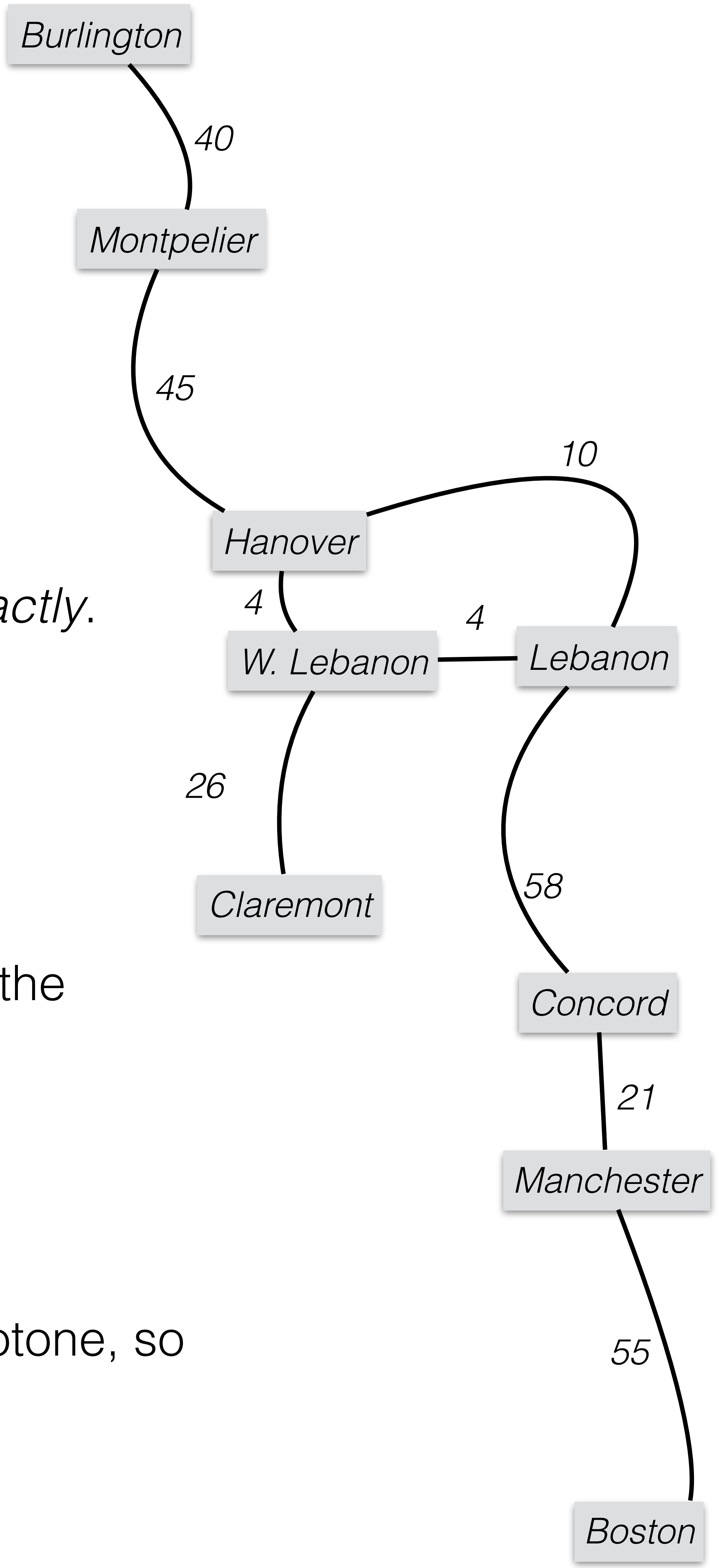
A* Search

- If we are looking for shortest path to a *known goal* we can use a generalization of Dijkstra's algorithm — ***A* Search***
- Use estimated distance remaining
- Add it to distance traveled so far, to get the *key* for an item in the PQ.
  - ex. using approx. euclidean dist.
    - *d(Montpelier,Boston)=130*
    - *d(Hanover,Montpelier)=45 + 130 = 175,*
    - *d(Hanover,Boston) = 120*
    - *Boston will come out of the PQ first!*
- The significant change to Dijkstra's is the update to how we determine the key (distance so far + approx. distance remaining).



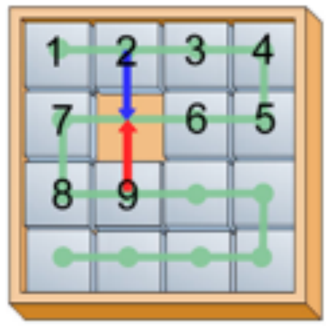
A* Search

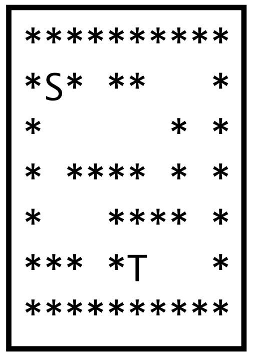
- In order to use A* Search to find S.P., we need 2 things:
  1. Estimate of remaining distance (must be admissible…)
     - i.e., should *underestimate* the remaining distance or *get it exactly*.
     - Triangle inequality —> E.D. is an underestimate of the true travel distance.
     - Ultimate underestimate = 0 —> this is just Dijkstra's!
  2. Cost estimate must be *monotone*
     - If we extend the path (increase the distance travelled so far), the total estimate is never less than the estimate before extending the path.
     - a.k.a. no negative edges.
     - E.D. is good for this because our steps are at least of size d.
- **Note:** Using *just* E.D. (without distance so far) would not be monotone, so it would not be guaranteed to give shortest paths.



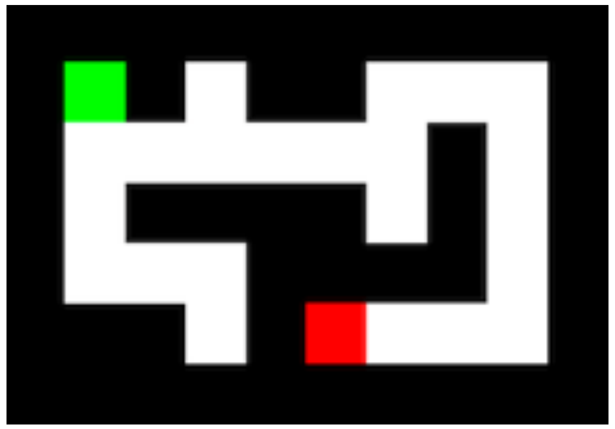
Implicit Graphs

- Mazes are an example of Implicit Graphs…
  - “open” squares are vertices
  - edges connect adjacent squares.
- Other examples of implicit graphs
  - 15-Puzzle
  - Rubix Cubes
  - implicit graph is potentially *huge!*
  - Create vertex objects as we need them + find adjacencies on the fly.
- We will look at: mazeSearch.zip (code) and mazes.zip (maze blueprints)
  - Provides code to convert a text repr. of a maze into a GUI where we can look at various graph search techniques and how they perform on different mazes
  - S = start, T = goal, * = wall, ‘ ‘ = path





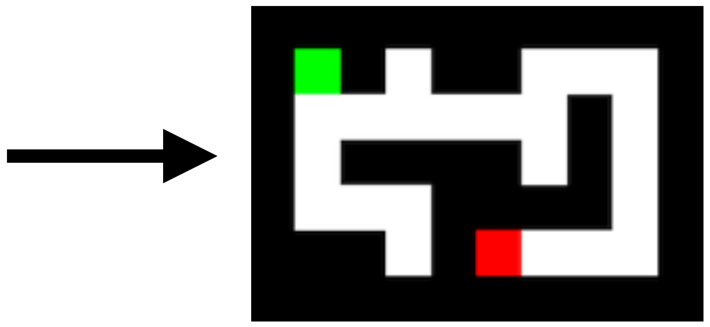
```
**********
*S* **   *
*      * *
* **** * *
*   **** *
*** *T   *
**********
```



Implicit Graphs

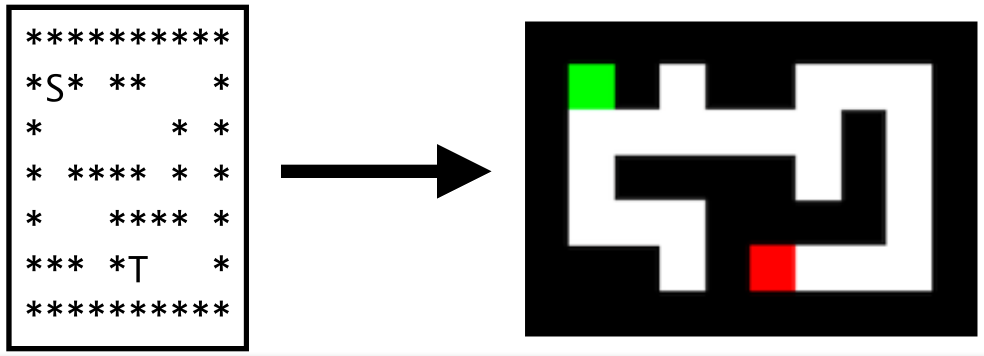
- **DFS** leads to long paths (see: maze5)
- **BFS** guarantees finding the shortest path, but can be slow…
- Can improve DFS by using a **L1 distance** to estimate remaining distance
    - int dist = Math.abs(row-targetRow) + Math.abs(col-targetCol);
- **A*** improves upon this by using combination of L1 distance (remaining) + distance traveled so far

```
**********
*S* **   *
*      * *
* **** * *
*   **** *
*** *T   *
**********
```



Implicit Graphs: Code Overview

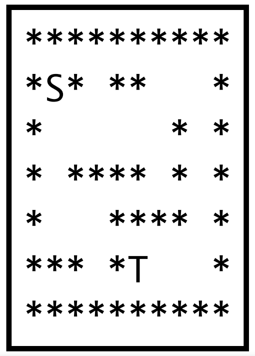
- **[MazeSolver.java]**
  - calls *MazeFrame.java* to setup the basic GUI
  - calls *Maze.java* to create a maze from a text file (uses *MazeIO* helper methods*)* and do some basic format checking/validation + provides functionality for interacting with a maze (stepMaze, isValid, getters, etc.)
- **[Sequencer.java]**
  - *Interface* for interacting with items that are to be "sequenced" (explored in some particular order). Sequencer's should keep track of "previous" sequence item so we can retrace path back to start from goal.
  - StackSequencer.java, QueueSequencer.java, DistSequencer.java, AStarSequencer.java
    - Note: L1 distance in DistSequencer
- **[SequenceItem.java]**
  - Holds an item in the sequencer. Keeps track of *row*, *col*, and *previous* sequence item.
  - PQSequenceItem.java — add notion of "key" (so it can be used in a PQ).
  - AStarSequenceItem.java — add notion of length of "path so far".



```
**********
*S* **   *
*      * *
* **** * *
*   **** *
*** *T   *
**********
```

Implicit Graphs: Code Overview

- **stepMaze()** method (in *Maze.java*) does the bulk of the work.
- A lot of this is more complicated than it needs to be since we want to make it compatible with a nice GUI…
  - It builds a maze from a file
  - Creates a *Sequencer* object based on the selected option
  - Steps through maze, trying to add unvisited/valid neighbors to the *Sequencer*.
  - If the Target is found, we run back up the path and color squares to show the shortest path we found.
    - Note: this is possible because we keep track of the *previous* item when we step.
- Try using the different *Sequencer* objects.
  - StackSequencer/QueueSequencer uses the respective data structure
  - DistSequencer uses a Priority Queue w/ L1 distance (remaining) as the key
  - AStarSequencer uses a PQ w/ L1 distance remaining + distance of path travelled so far.



```
**********
*S* **   *
*      * *
* **** * *
*   **** *
*** *T   *
**********
```