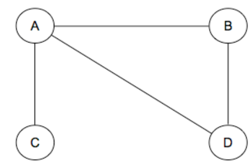
Lecture 39: Dijkstra's Algorithm

CSCI 62
Fall, 2016

Kim Bruce & Peter M

Graph Representations

- Adjacency Matrix
- Adjacency List



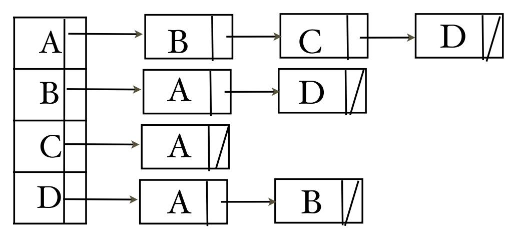
Adjacency Matrix

	A	*B*	*C*	*D*
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Can put in edge weights as entries

- Good for dense graphs
- Constant time lookup for edges.
- Symmetric if undirected.

Adjacency Lists



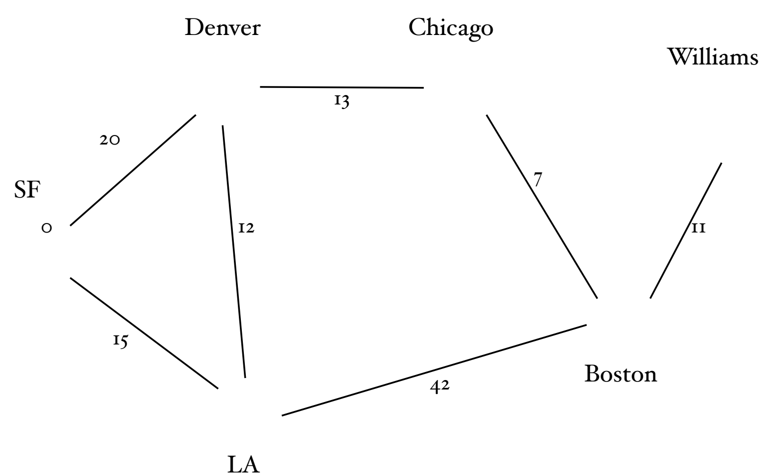
- Good for sparse graphs, saves space
  - lists on right can be vectors, array on left could be map
- Linear time lookup for edges.

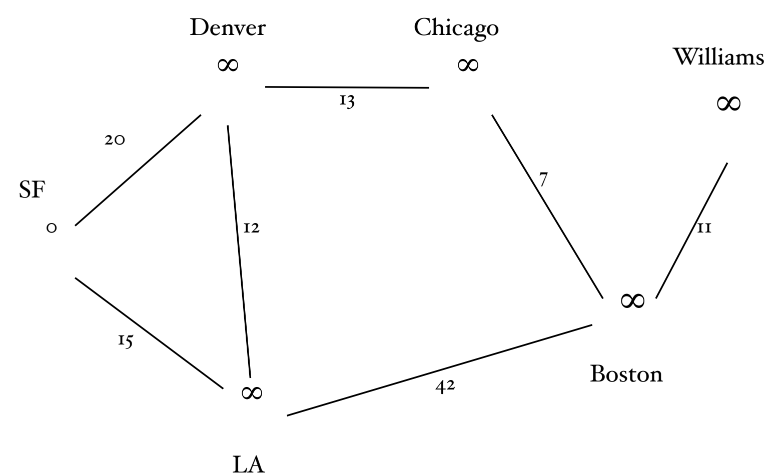
Complexity of BFS

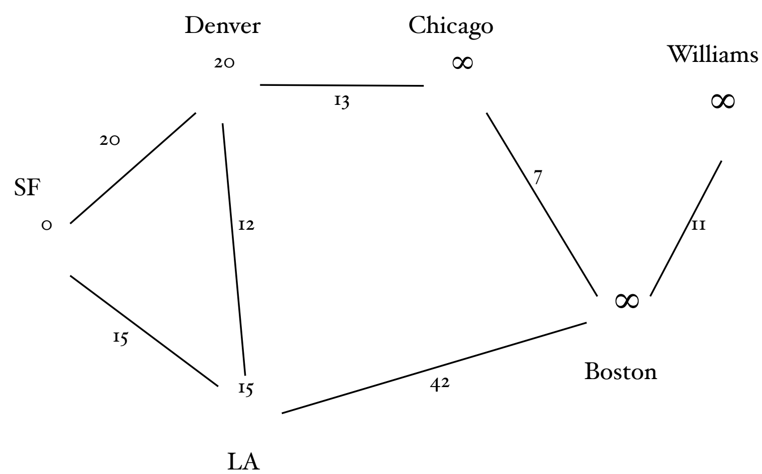
- Enqueue the start node
- while the queue is not empty
  - Dequeue a node
  - if the the node has not been visited previously,
    - visit it
    - enqueue all of the node's children
- Adjacency List:
  - Each edge contributes both end points to queue
  - O(max(v,e)) if each visit takes constant time
- More expensive with adjacency matrix

Single-Source Shortest Paths

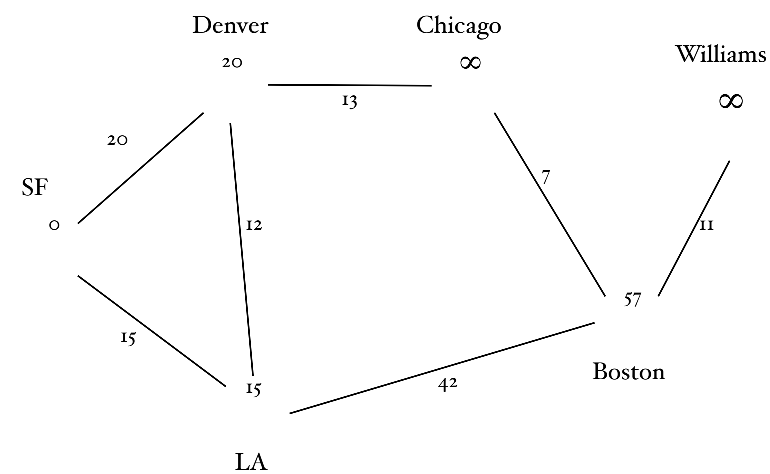
- Like Breadth-first search
  - If all paths have length 1
  - Otherwise use priority queue and mark with predecessor so can find shortest path.
  - Assume all edges have non-negative weights.
  - Due to Dijkstra



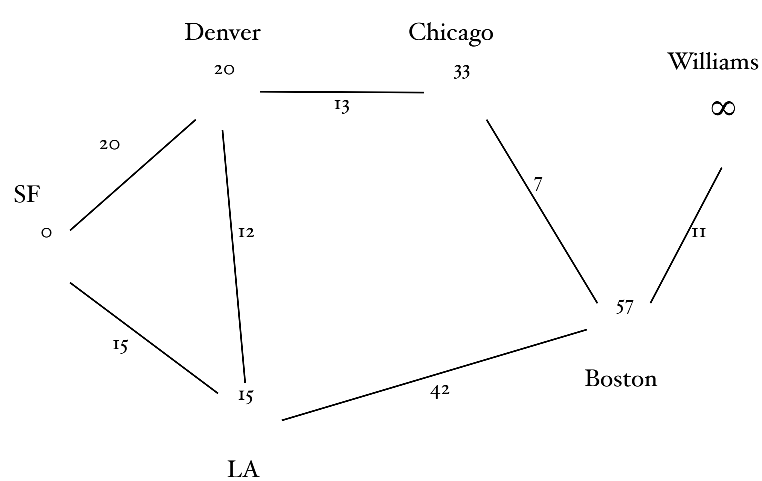




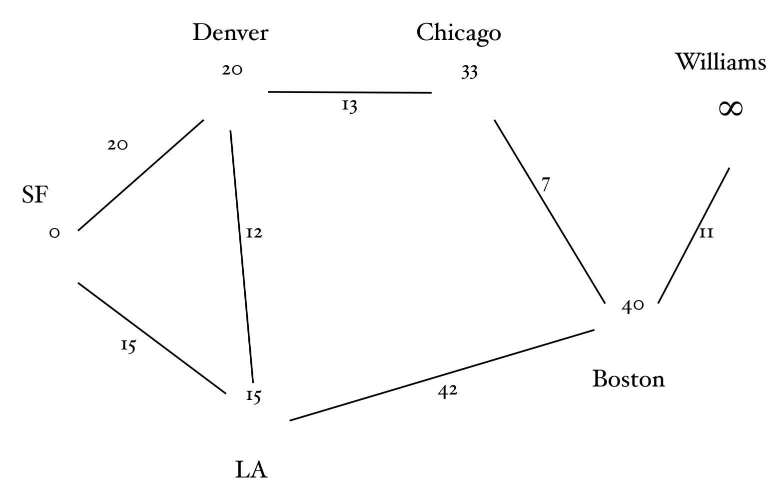
PQ = [SF], adding LA and Denver



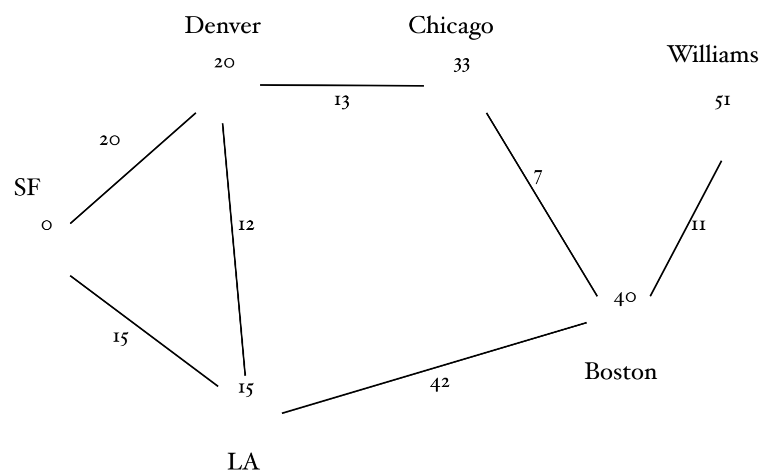
PQ = [LA,D], adding B, D



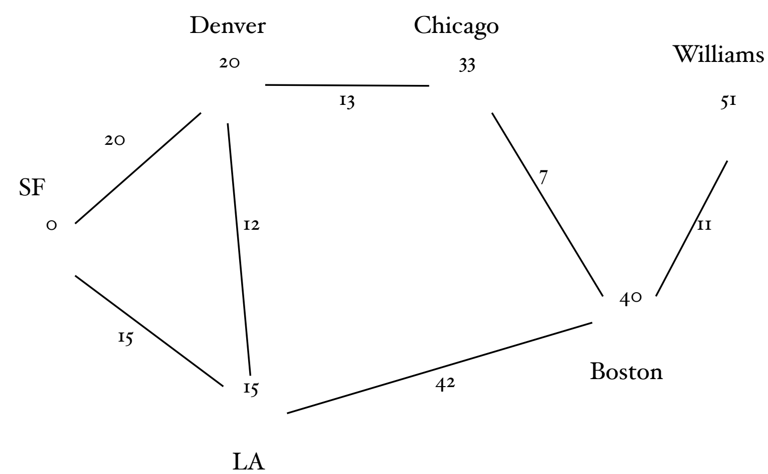
PQ = [D,B], adding C, LA, SF



PQ = [C, B], adding B



PQ = [B], adding W



Single Source Shortest Path Problem

- From a starting node s, find the shortest path (and its length) to all other (reachable) nodes
- The collection of all shortest paths form a tree, called... the shortest path tree!
- If all edges have the same weight, we can use BFS.
- Otherwise ...

Single Source Shortest Path Problem

- If all edges have weights ≥ 0, then we use Dijkstra's algorithm
  - Essentially just BFS with priority queue
  - Priorities are best known distance to a node from the source
  - Keep track of parents as in BFS so can get path
- Example of a "greedy" algorithm!

Dijkstra

- Variables
  - graph G
  - vertex_t s // start node
  - double length[n][n] // Edge lengths (adj. list)
  - double dist[n] // Current best distance
  - vertex_t parent[n] // Current parent
  - pqueue Q // priority queue

Dijkstra Algorithm

- Set dist[v] to ∞ for all v, except dist[s] = 0
- Add s to Q with priority 0.0
- loop while Q is not empty:

```
get node cur with min priority d (distance from s)
if (d ≤ dist[cur])
    for each out going edge cur→v:
        if dist[cur] + length[cur][v] < dist[v]:
            dist[v] = dist[cur] + length[cur][v]
            parent[v] = cur
            Add v to Q with new priority dist[v]
```

Run Dijkstra on Sample Graph

Run-Time of Dijstra

- Let v = #vertices, e = #edges
- Adding and removing from priority queue, O(log v)
  - Each goes on and off once, so O(v log v)
- reduce_priority O(log v)
  - Worst case, once for each edge, so O(e log v)
- Total time O((e+v) log v)