- Representations of graphs
- Graph algorithms:
  - Dijkstra’s single-source, shortest paths algorithm
  - Prim’s minimal spanning tree algorithm
- Graph traversals: breadth-first, depth-first, best-first

We will use $|V|$ to denote size of vertex set and $|E|$ to denote size of edge set of a graph $G = (V, E)$. 
Representations of graphs

Specification of nodes, edges and labels on nodes

- **Explicit representations:**
  - Adjacency matrix
  - Adjacency lists

- **Implicit representations:** (eg) 8-puzzle graph

Nodes, edges etc. are specified *implicitly* by giving rules for generating graph as needed.

To get a feel for graphs, let us study some graph algorithms.

Suppose you have a USAir route map with inter-city distances marked on the map.

You want to know the shortest distance from Ithaca to every city served by USAir.

This is called a *single-source* (Ithaca), shortest-distance problem with positive weights (inter-city distances).

Algorithm: Dijkstra’s algorithm
A few steps of Dijkstra’s algorithm

What is shortest distance from A to other cities?
Intuition behind algorithm:
Imagine each node is a small weight
and each edge is a thread of appropriate length
Pick up A and lift it gradually, looking at sequence in which other nodes get lifted up.

Useful to divide nodes at any point during lifting into three categories:

• Lifted node: is already lifted up at this point
• Frontier node: not lifted up yet, but is adjacent to a lifted node
• Unseen node: not lifted and is not adjacent to lifted node

Bridge: edge, one end of which is lifted, and other is frontier

Running Example

Shortest distances from A to:
A: empty route, length 0
B: (A->B), length 2
C: (A->B,B->C) length 6
D: (A->B,B->C,C->D) length 8
....

At end of algorithm, each node has a minDistance value = length of shortest path from start node to it (shown in purple in figure).

Note: edges on shortest paths to nodes form a tree.

End of Dijkstra’s algorithm

Edges on shortest-path routes are thick.
These edges form a tree.

How do we express this intuitive description as an algorithm?
What is the asymptotic complexity of this algorithm?
Foreach node, maintain an integer called its minDistance which is the length of the shortest path from the start node. Initialize to $\infty$.

Maintain a heap that contains an entry for each bridge (t $\rightarrow$ f) with priority $=$ minDistance(t) + length(t $\rightarrow$ f) (intuitively, this tells you how high the root node has to be raised before this bridge becomes taut).

Pseudo-code:

Heap h = new Heap();
h.put(new PQElement(data = (dummyroot->startNode), priority = 0));
while (h is not empty) {
    . get minimum priority bridge (t $\rightarrow$ f)
    . make f a lifted node and set minDistance(f) = minDistance(t) + length(t $\rightarrow$ f)
    . remove all bridges ending at f from heap
    . for each edge (f $\rightarrow$ n)
        . if (n is not lifted)
            . make n a frontier node if it is not already one
            . stick edge (f $\rightarrow$ n) into PQ with priority = minDistance(f) + length(f $\rightarrow$ n)
}

Difficulty: how do we find all bridges ending at f in heap??

Correctness of algorithm

- Induction on iterations of while loop
  Intuitively, each iteration moves one new node into the lifted set. Therefore, we do an induction on the set of nodes ordered in the sequence in which they get put into the lifted set.

  - Argument:
    - Base case: start node has a trivial path of length 0 to itself
    - Inductive case: assume that the shortest paths to all nodes currently in the lifted set have been computed correctly, and argue that the next node that gets lifted is the right one.

More relaxed algorithm: permit heap to contain some edges between lifted node. When we get an edge out, process edge only if destination edge is not already lifted.

Heap h = new Heap();
h.put(new PQElement(data = (dummyroot->startNode), priority = 0));
while (h is not empty) {
    . get minimum priority edge (l $\rightarrow$ f)
    . if (f is not already lifted){ //we have a bridge
        . make f a lifted node and set minDistance(f) = minDistance(l) + length
        . for each edge (f $\rightarrow$ n)
            . if (n is not lifted)
                . make n a frontier node
                . stick edge (f $\rightarrow$ n) into PQ with priority = minDistance(f) + length
    }
}
Invariant: at the top of the while loop,

1. each lifted node has its minimal path length computed correctly
2. PQ contains all bridges, and their priority is computed correctly

We can argue that (i) the invariant holds before the first iteration, and (ii) if it holds before iteration $i$ begins, it holds before iteration $i + 1$ begins.

Sketch of proof: (see picture)

Suppose minimal priority bridge is $(L \rightarrow N)$.

Let minimal length path from $A$ to $L$ be path $p$.

We argue that path $p + (L \rightarrow N)$ is minimal path to $N$.

If not, there is another path $q$ from $A$ to $N$ that has strictly smaller length. Since path $q$ starts at a blue node $(A)$ and ends at a red node $(N)$, there must be at least one bridge edge on this path. Let $(Z \rightarrow R)$ be the first bridge on this path.

By inductive assumption,

\[ \text{length}(A \rightarrow Z) + \text{length}(Z \rightarrow R) \geq \text{length}(A \rightarrow L) + \text{length}(L \rightarrow N) \]

Since all edge lengths are non-negative, this means that

\[ \text{length}(A \rightarrow Z) + \text{length}(Z \rightarrow R) + \text{length}(R \rightarrow N) \geq \text{length}(A \rightarrow L) + \text{length}(L \rightarrow N) \]

contradicting the assumption that path $q$ has strictly smaller length than path $p$.

Therefore, when we extract the min from the priority queue in iteration $i$ and make a new node lifted, we have computed its length correctly.

We now add all bridges whose end-point is $L$ to maintain the second part of the invariant.

Complexity of algorithm:

- Every edge is examined once and inserted into PQ when one of its two end points is first lifted
- Every edge is examined again when its other end point is lifted
- Number of insertions and deletions into PQ = $|E| + 1$

So algorithm complexity = $O(|E| \log(|E|))$
Concluding remarks:

There are faster but much more complicated algorithms for single-source, shortest-path problem that run in time $O(|V|\log(|V|) + |E|)$ but use things called Fibonacci heaps.

In practice, our algorithm will probably run better ....

For these and fancier data structures and algorithms, take CS 410 and CS 482.