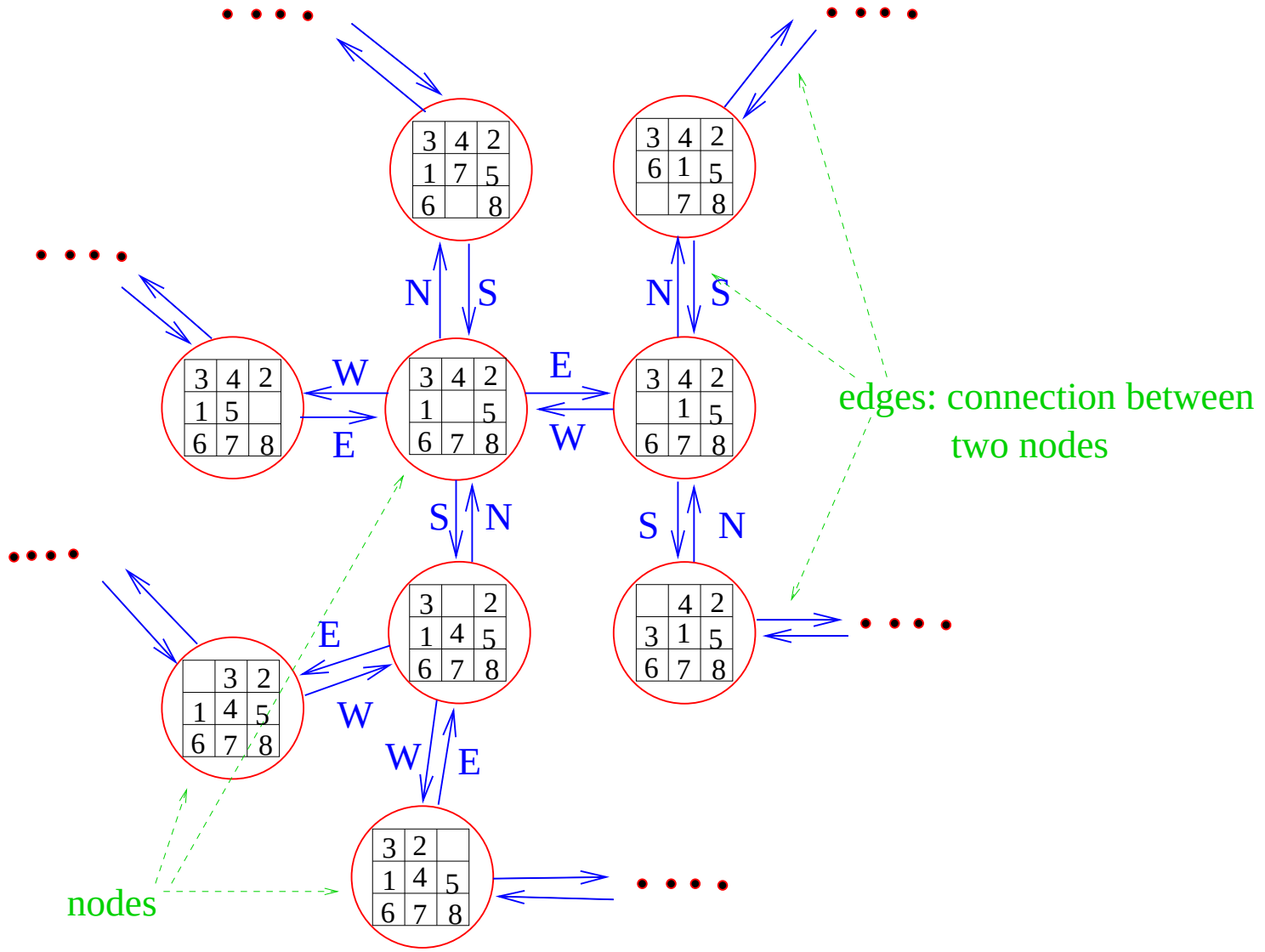
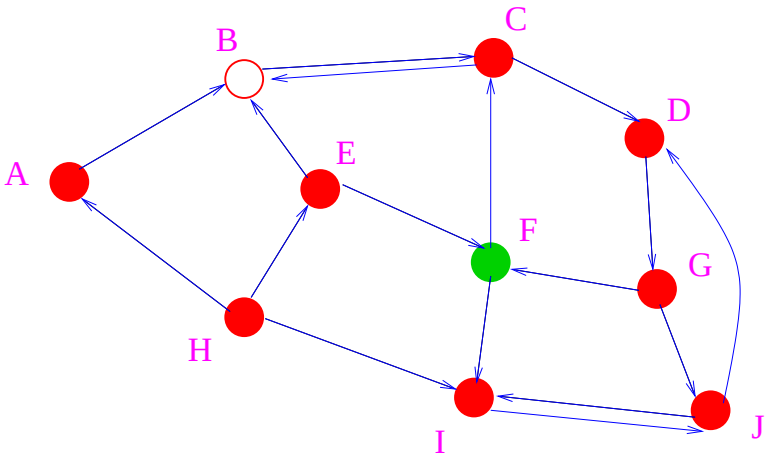
Graphs

- 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

Graph: set of Nodes and Edges between nodes





V = {A,B,C,D,E,F,G,H,I,J}

E = {(A,B), (B,C),(C,B),.....}

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

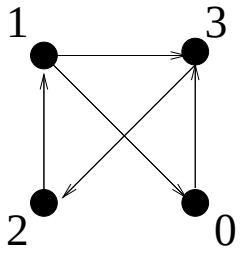
Representations of graphs

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.

Adjancency Matrix



Graph

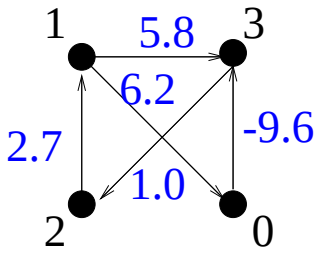
M:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency Matrix

M(i,j) is 0 if edge (i->j) is absent
1 if edge (i->j) is present

If edges have information, this can be stored in adjancency matrix.



Graph

M:

$$\begin{bmatrix} 0 & 0 & 0 & -9.6 \\ 6.2 & 0 & 0 & 5.8 \\ 0 & 2.7 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \end{bmatrix}$$

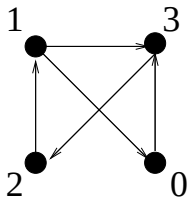
Adjacency Matrix

Adjacency matrix: lots of 0's if graph is "sparse" in the sense that nodes typically have relatively few neighbors.

Disadvantages: wasted space

determining which nodes are adjacent to a given node requires wading through lots of 0's

Adjancency List



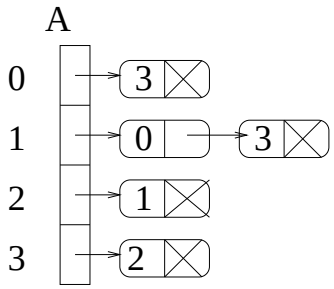
Graph

M:

0	0	0	1
1	0	0	1
0	1	0	0
0	0	1	0

Adjacency Matrix

M(i,j) is 0 if edge (i->j) is absent
1 if edge (i->j) is present



Adjacency List

Array of lists
A(i) stores list of nodes adjacent to node i

Determining if edge (i->j) exists in a graph:

O(1) operation for adjacency matrix, O(|E|) operation for adjacency list

Enumerating all nodes adjacent to node i in graph:

O(|V|) operation for adjacency matrix,
O(1) per adjacent node for adjacency list

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

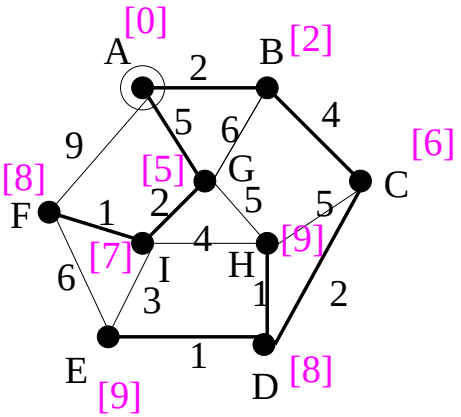
Suppose you have a USAir route map with inter-city distances marked on the map.

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

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.

Intuition behind Dijkstra's algorithm

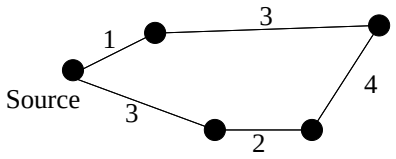
Imagine that each node is a small weight, and that each edge is a string of the appropriate length connecting these weights.

Initially, entire contraption is on a table.

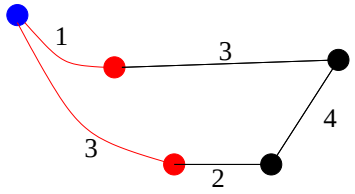
Pick up the source node from the table and keep lifting it until all nodes are off the table.

Height of source when a node comes off the table = shortest distance from source to that node!

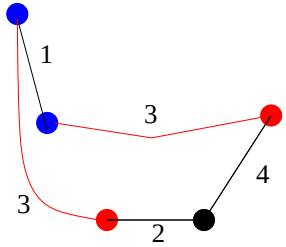
Analog computer for Dikstra's algorithm



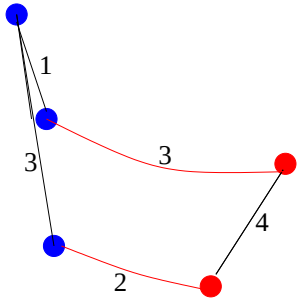
(a) Initial Configuration



(b) Source node is lifted



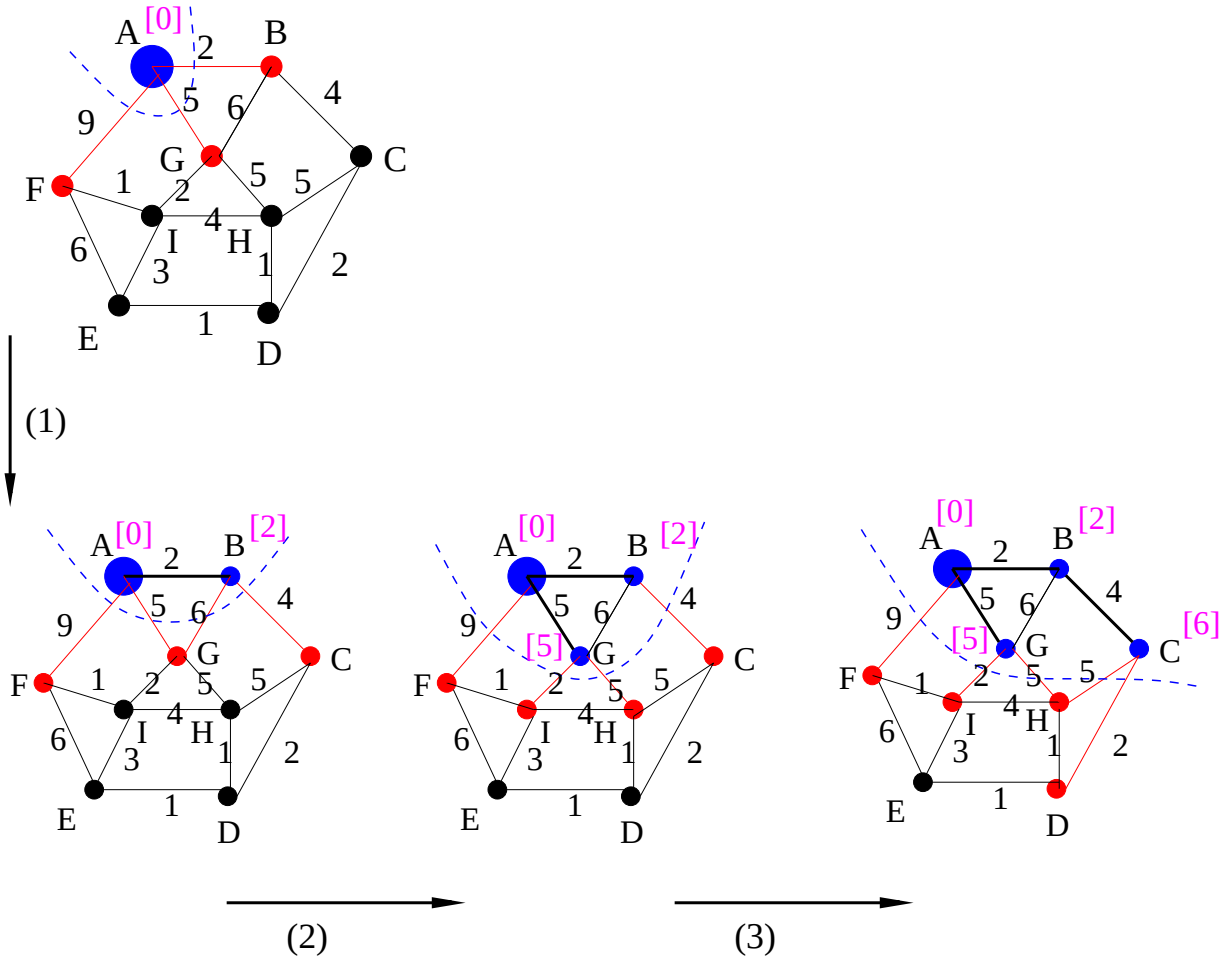
(c) Two nodes are lifted



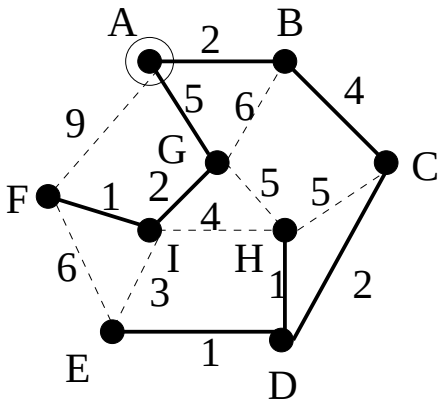
(d) Three nodes lifted

Blue node: lifted, Red node: not lifted, but connected to a blue node by some edge
Black: all other nodes Bridge: edge between blue and red node

A few steps of Dijkstra's algorithm for running example



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?

Algorithm

- Bridge edges are the key: one of them will become taut, and raise the weight at the other end.
- Given a set of bridges, which one will become taut first?
  Answer: The one whose red node is nearest to the source.
- Data structure: maintain a data structure containing all bridges (B$\rightarrow$R), ordered by value of (shortest distance from source to B + length of bridge).
- At each step, get the bridge with smallest priority, and add any new bridges that may be formed –¿ we need a heap.

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->f)
  . make f a lifted node and set minDistance(f) = minDistance(t) + lengt
  . remove all bridges ending at f from heap <---??? how
  . for each edge (f->n)
     if (n is not lifted)
       . stick edge (f->n) into PQ with priority = minDistance(f) + leng
}
```

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

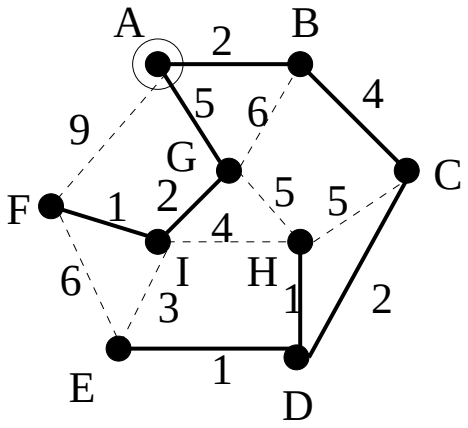
More relaxed algorithm: permit heap to contain some edges between lifted node. When we get an edge out, process edge only if destination edge is not already lifted.

```
Heap h = new Heap();
h.put(new PQElement(data = (dummyroot->startNode), priority = 0));
while (h is not empty) {
  . get minimum priority edge (l->f)
  . if (f is not already lifted){ //we have a bridge
     . make f a lifted node and set minDistance(f) = minDistance(l) + le
     . for each edge (f->n)
        if (n is not lifted)
          . make n a frontier node
          . stick edge (f->n) into PQ with priority = minDistance(f) + l
    }
}
```

Correctness of algorithm

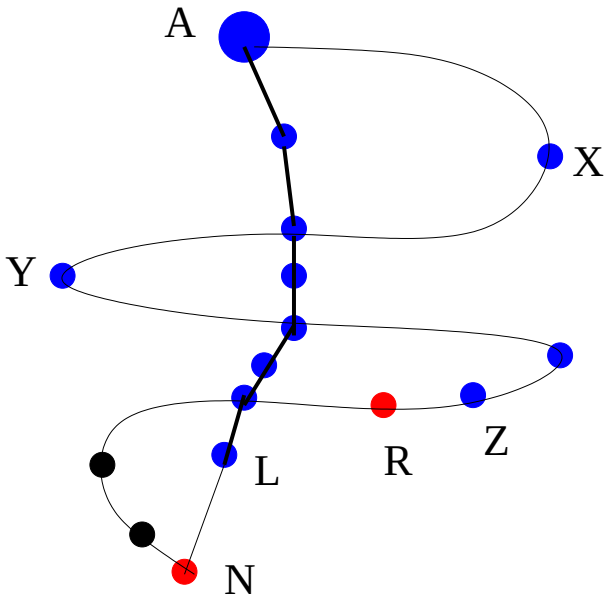
- Induction on iterations of while loop
  Intuitively, each iteration moves one new node into the lifted set. Therefore, we do an induction on the set of nodes ordered in the sequence in which they get put into the lifted set.
- Argument:
  - Base case: start node has a trivial path of length 0 to itself
  - 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.

Picture for proof of correctness



Discovery order:

[A B G C I D F H E]



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,

$length(A \rightarrow\rightarrow Z) + length(Z \rightarrow R) >= length(A \rightarrow\rightarrow L) + length(L \rightarrow N)$

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

$length(A \rightarrow\rightarrow Z) + length(Z \rightarrow R) + length(R \rightarrow\rightarrow N) >= length(A \rightarrow\rightarrow L) + length(L \rightarrow N)$

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 lifte

- 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 ....

Requirement that all edge weights be non-negative is important; otherwise, we need a more complicated algorithm called Warshall's algorithm.

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