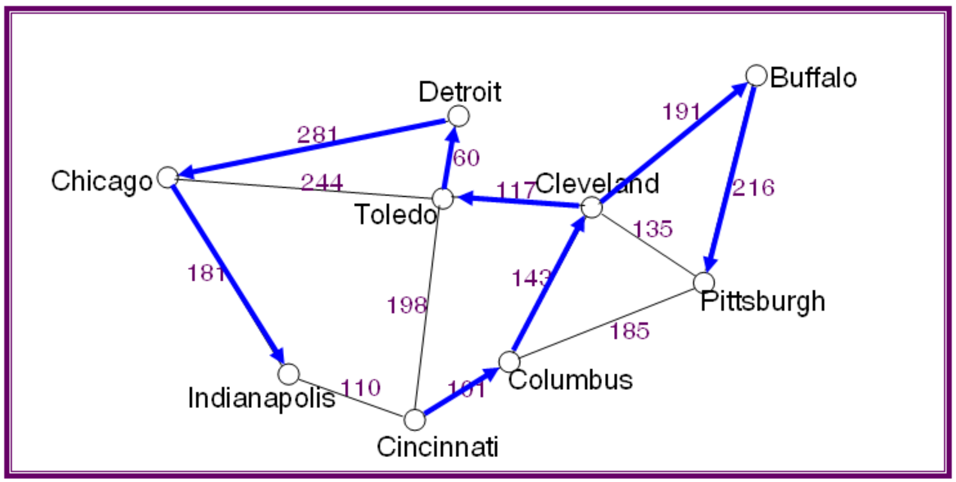
Shortest-Path Problem

EECS 233

Depth-first Traversal

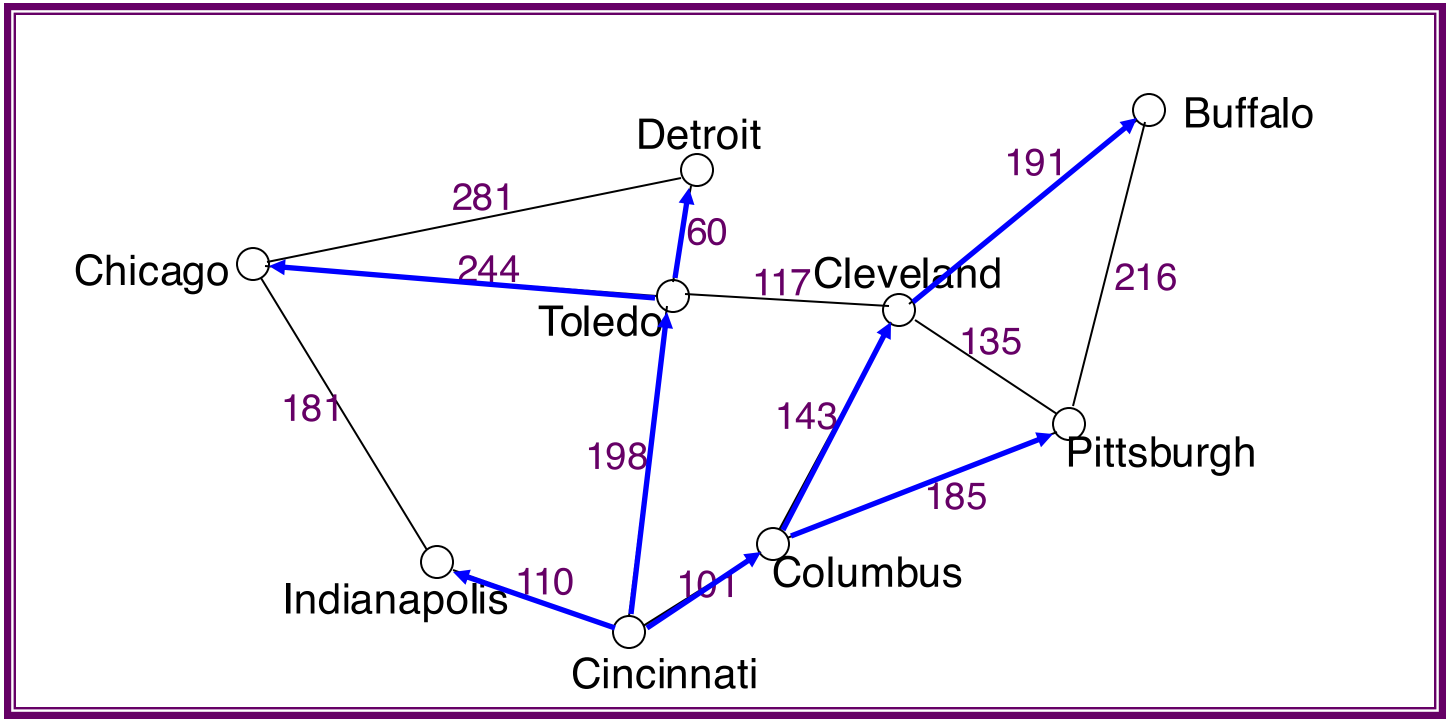
- Visit a vertex, then make recursive calls on all of its yet-to-be-visited neighbors:

```
depthFirstTrav(v)
    myDepthFirstTrav(v, NULL)

myDepthFirstTrav(node, parent)
    visit node and mark it as visited
    node.parent  = parent
    for each vertex w in node's neighbors
        if (w has not been visited)
            myDepthFirstTrav(w, node)
```




```
public class Graph {
        class Vertex {
                private String id;
                private linkedList <Edge> edges;
                    // adjacency list
                private boolean encountered;
                private boolean done;
                private Vertex parent;
                private double cost;
                …
        }
        class Edge {
                private int endNode;
                private double cost;
                …
        }
        private Vertex[] vertices;
        private int numVertices;
        private int maxNum;
        …
}
```

Breadth-First Traversal



Order: Cincinnati, Columbus, Toledo, Indianapolis, Pittsburgh, Cleveland, Detroit, Chicago, Buffalo

- Starting from Cincinnati, what would be the order of visits?

- **breadth-first**:
  - visit a vertex
  - visit all of its neighbors
  - visit all unvisited vertices 2 edges away
  - visit all unvisited vertices 3 edges away, etc.

The Shortest-Path Problem

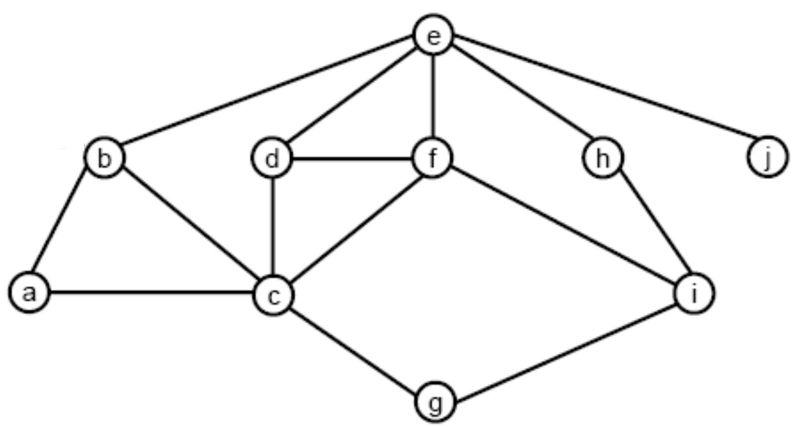
- What is the “shortest” path from one vertex to another – i.e., the one with the minimal total cost
  - Travel: lowest mileage or fastest trip
  - Internet: forwarding traffic along the “best” router paths.
- Given a source, find the shortest path to a destination
- Given a source, find the shortest path to many (all) destinations

First, A Special Case

- A solution for an *unweighted* graph:

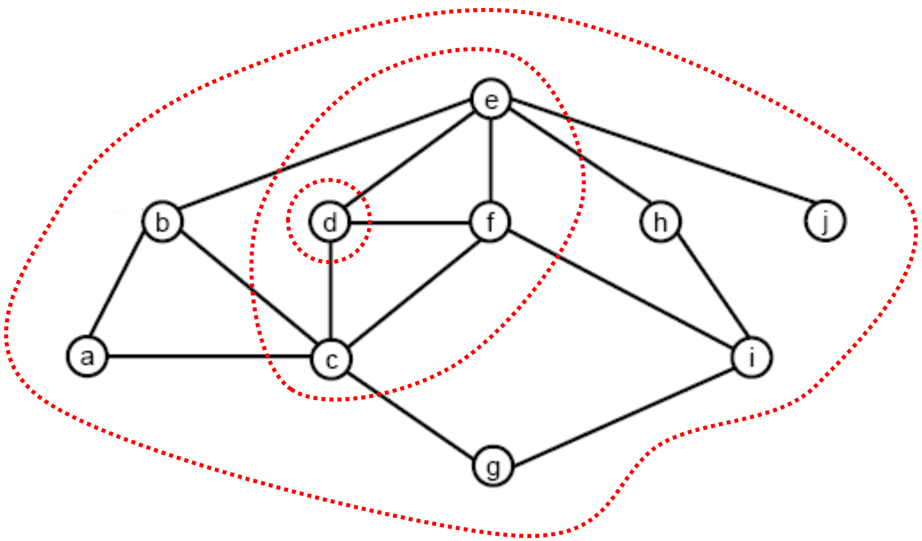
First, A Special Case

- A solution for an *unweighted* graph:



First, A Special Case

- A solution for an *unweighted* graph:
  - Start a breadth-first traversal from the origin, x
  - Stop the traversal when you reach the target, y
  - The path from x to y in the resulting (possibly partial) spanning tree is *a* shortest path
- A breadth-first traversal works for an unweighted graph because
  - the shortest path is simply one with the fewest edges
  - a breadth-first traversal visits nodes in order according to the number of edges they are from the origin.



Method for the Special Case
(Pseudo-code)

```
shortestPath_bfTrav(source, desti)
    mark source as encountered
    source.parent = null
    source.cost = 0;
    create a new queue q
    q.insert(source);

    while (!q.isEmpty())
        v = q.remove()
        if (v==desti) break;
        for each vertex w in v's neighbors
            if w is not encountered
                mark w as encountered
                w.parent = v;
                w.cost = v.cost + 1;
                q.insert(w);
```

Shortest paths for Weighted Graphs: Dijkstra's Algorithm

- Finds the shortest paths from vertex v to *all other vertices* that can be reached from v.
  - The single-source-all-destinations problem

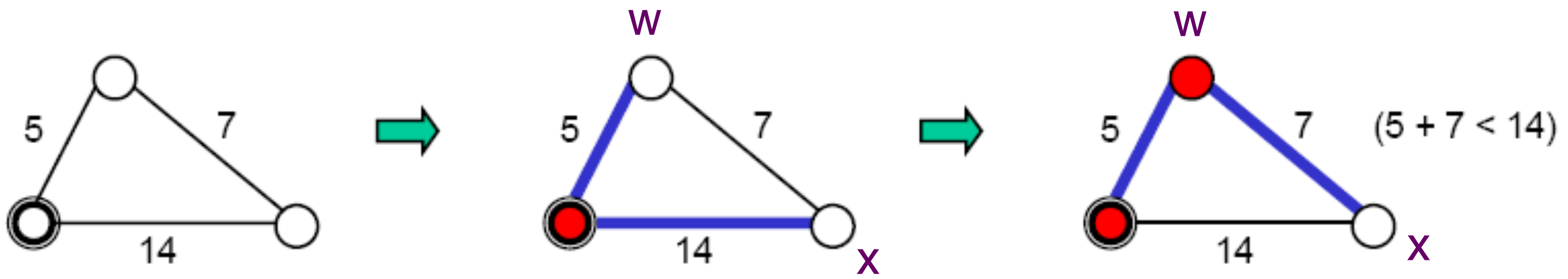
- Assumptions:
  - Graph is connected.
  - Edge cost is non-negative
    - More hops = higher cost

- Basic idea:
  - maintain estimates of the shortest paths from v to every vertex (along with their associated costs)
  - gradually refine these estimates as we traverse the graph

Dijkstra's Algorithm

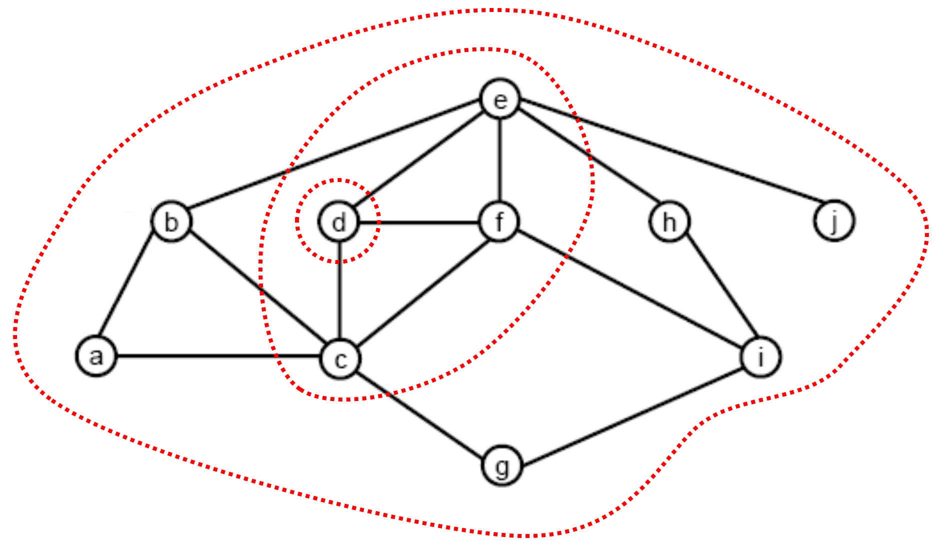
- We say that a vertex w is finalized if we are certain we have found the shortest path from v to w.

- We repeatedly do the following:
  - find the not-yet-finalized vertex w with the lowest cost estimate
  - mark w as finalized (shown as a filled circle below)
  - examine each not-yet-finalized neighbor x of w to see if there is a shorter path to x that passes through w; if there is, update the shortest-path estimate for x

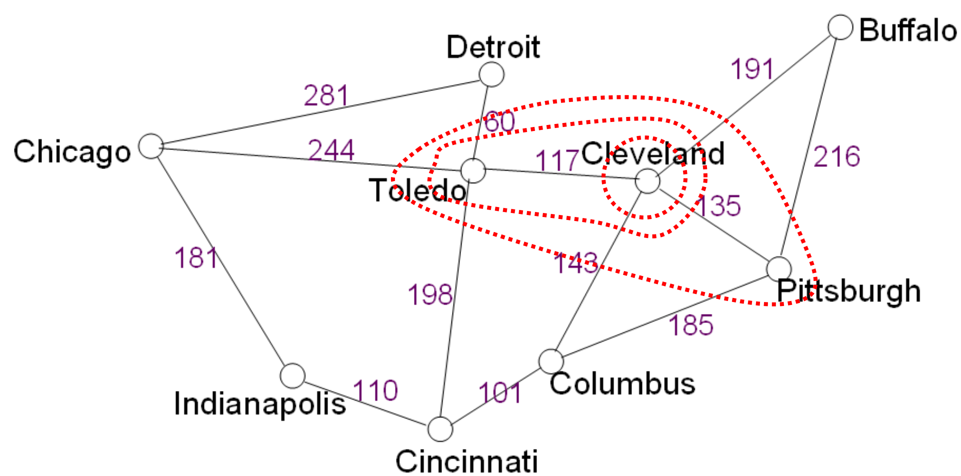


Intuition Behind Dijkstra's

- For an unweighted graph, the breadth-first traversal method expands the search circle by one hop every round

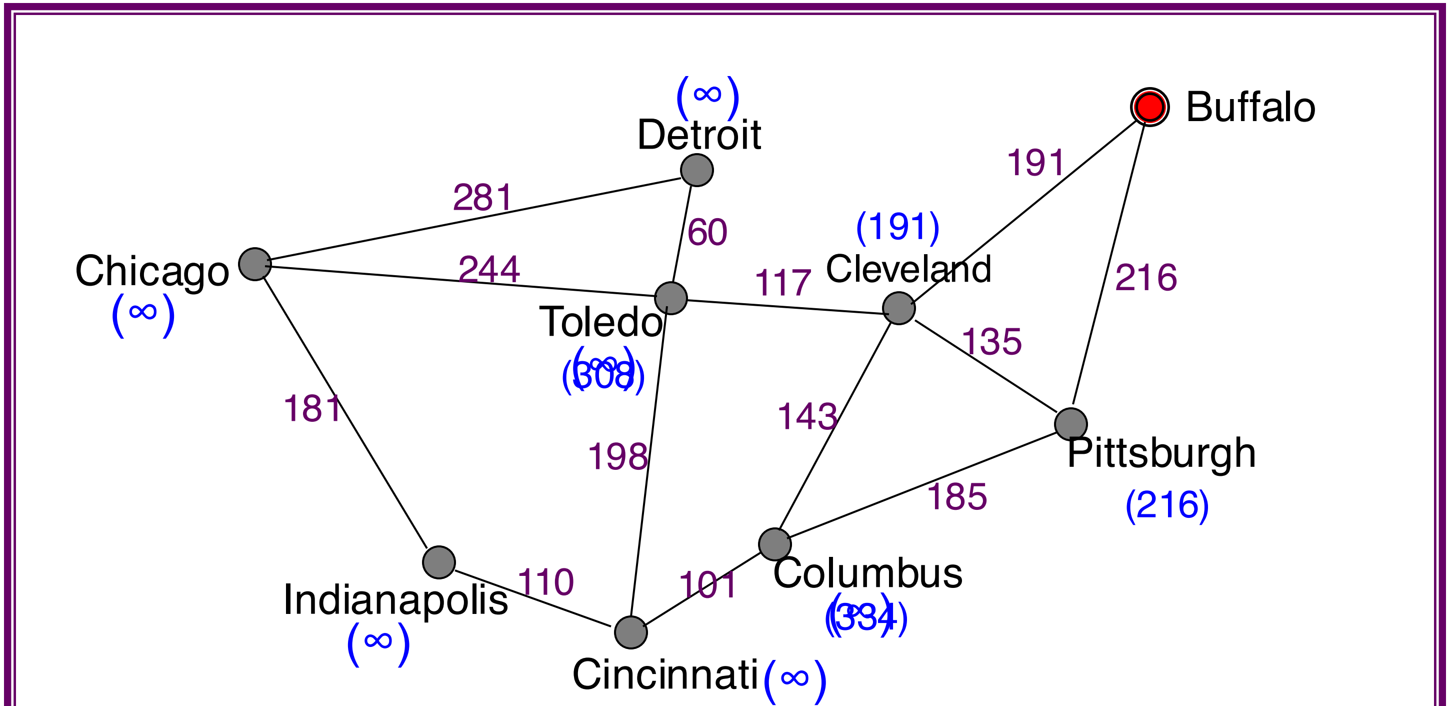


- For a weighted graph, Dijkstra's method expands the search circle **by one lowest-cost node** every round



Example: Shortest Path from Buffalo To Toledo

- Distance to Buffalo is 0, so it is finalized
- Initially set distance estimates:
  - To all neighbors - edge weights
  - To other cities - unknown, using *infinity*
- Which will be finalized?



Dijsktra’s Algorithm

Notation:

c(x,y): link cost from node x to y;
∞ if not direct neighbors

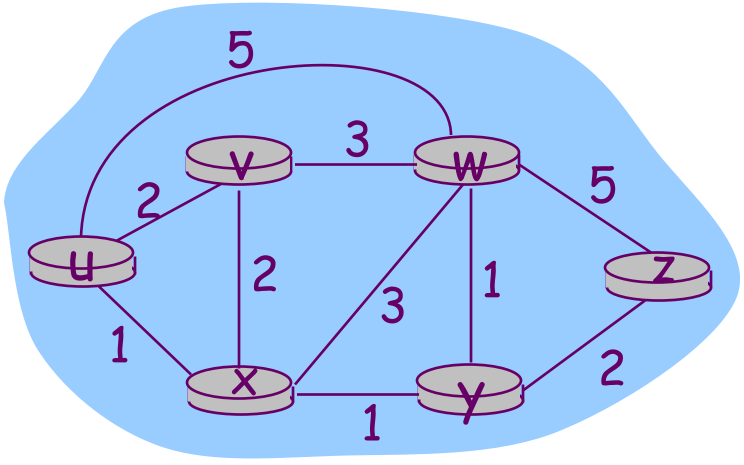
D(v): current path cost estimate from source to dest. v

p(v): predecessor node along path from source to v

N: set of nodes whose least cost path definitively known

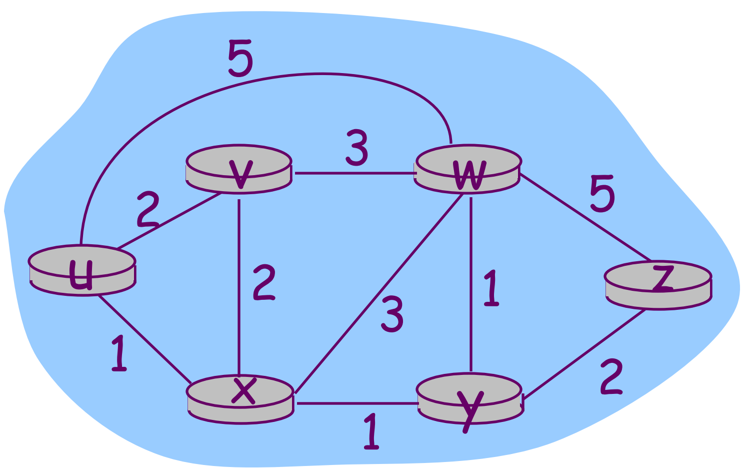
u: starting node

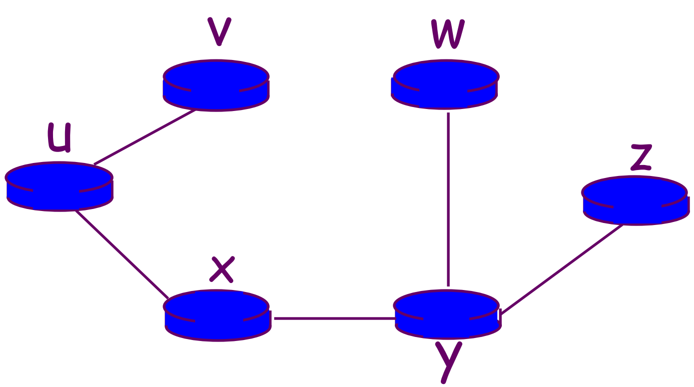
```
Initialization:
  N = {u}
  for all nodes v
      if v adjacent to u then
          D(v) = c(u,v)
          p(v) = u
      else D(v) = ∞

Loop
  find w not in N with smallest D(w)
  add w to N
  update D(v) for all v adjacent to w and not in N' :
        D(v) = min( D(v), D(w) + c(w,v) )
        Update p(v)
  /* new cost to v is either old cost to v or known
   shortest path cost to w plus cost from w to v */
until all nodes are in N
```



Dijkstra's algorithm: example

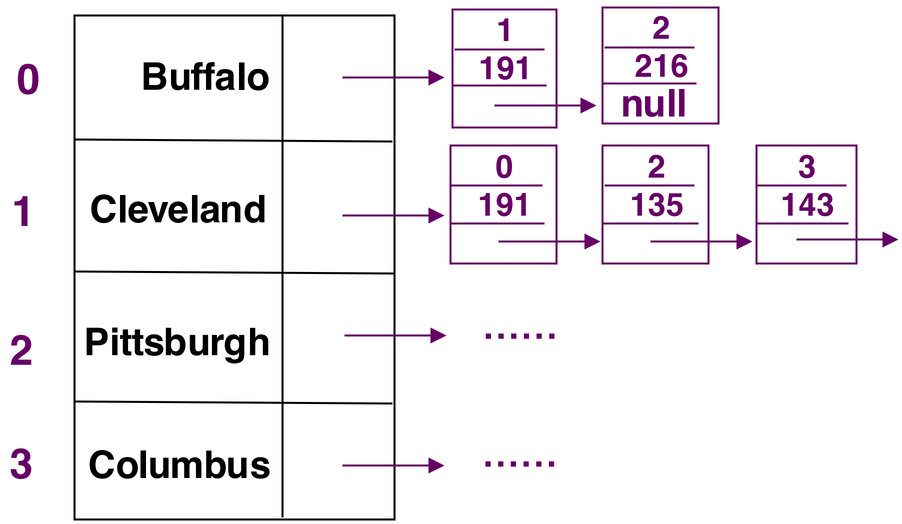
Step	N	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞





Implementation Issues

- Data structure: adjacency lists



- How to find the not-yet-finalized vertex with the minimal cost? What data structure can be used to make it more efficient?

Dijsktra's Algorithm: Complexity

```
Initialization:
  N = {u}
  for all nodes v              // V iterations
    if v adjacent to u         // Check of the total of up to E edges
       then D(v) = c(u,v)
    else D(v) = ∞
                               // O(V) to build the heap
Loop          // V iterations
  find w not in N with smallest D(w)           // logV to reorg
                                               the heap
  add w to N
  update D(v) for all v adjacent to w and not in N :
     D(v) = min( D(v), D(w) + c(w,v) )   // A step per every link (E total); logV
                                           per step to reorganize the heap
  /* new cost to v is either old cost to v or known
   shortest path cost to w plus cost from w to v */
until all nodes in N
```

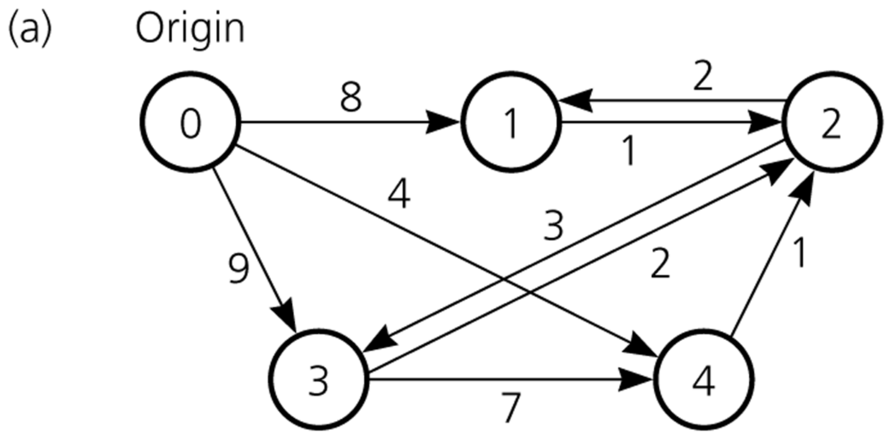
Overall complexity is O(V + E +VlogV + ElogV) = O(ElogV)

All-Pairs Shortest-Path Problem

- Dijkstra's algorithm solves the problem of single-source all-destination shortest paths

- It can be used to solve the all-source all-destination shortest path problem too
  - Run Dijkstra's V times, once for each vertex
  - Running time: O(VE logV)

Shortest Paths



(b)

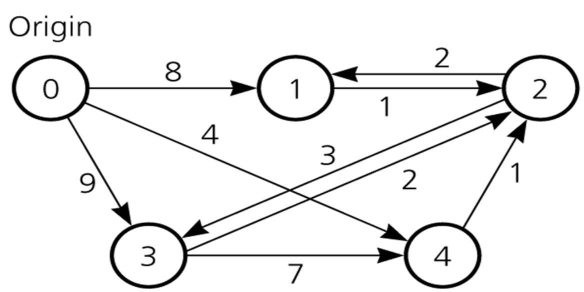
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

A Weighted Directed Graph **Its Adjacency Matrix**

Dijkstra's Shortest-Path Algorithm – Trace

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞