Graph Algorithms #2 - Toposort, Shortest path

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

- Announcements
- Thing of the day
- Toposort
- Shortest path algorithms

Announcements

- Next MA is due out and will be on Topo Sort

Attendance Day!
(It’s a quick one)

goo.gl/P8qz1v

Network graphs of the Internet!

https://personalpages.manchester.ac.uk/staff/m.dodge/cybergeography/atlas/historical.html

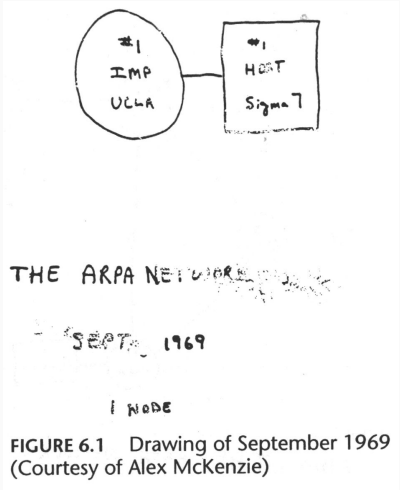
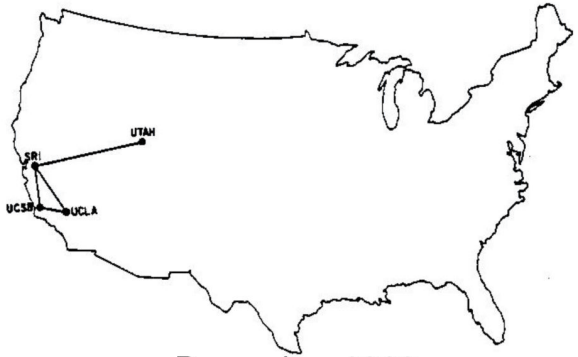
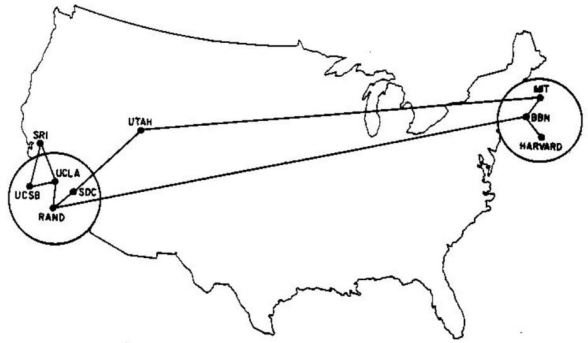


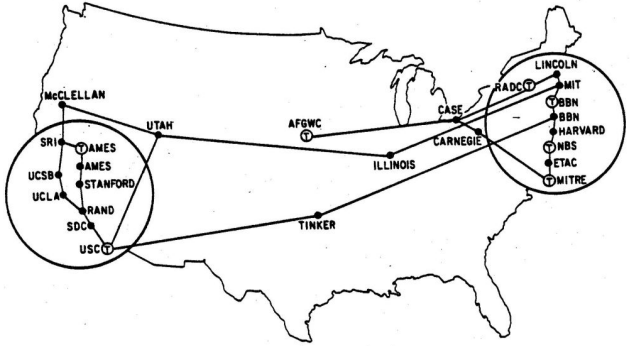
FIGURE 6.1 Drawing of September 1969 (Courtesy of Alex McKenzie)



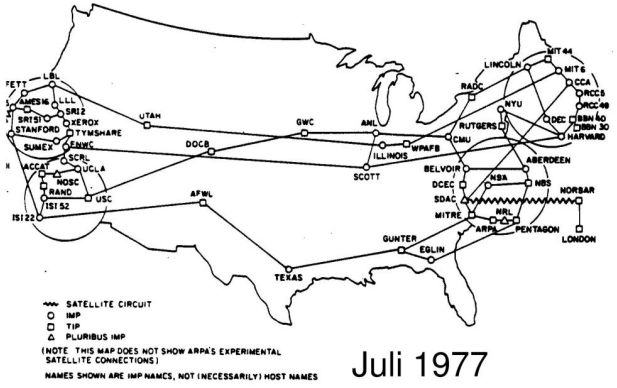
Dezember 1969



Juni 1970



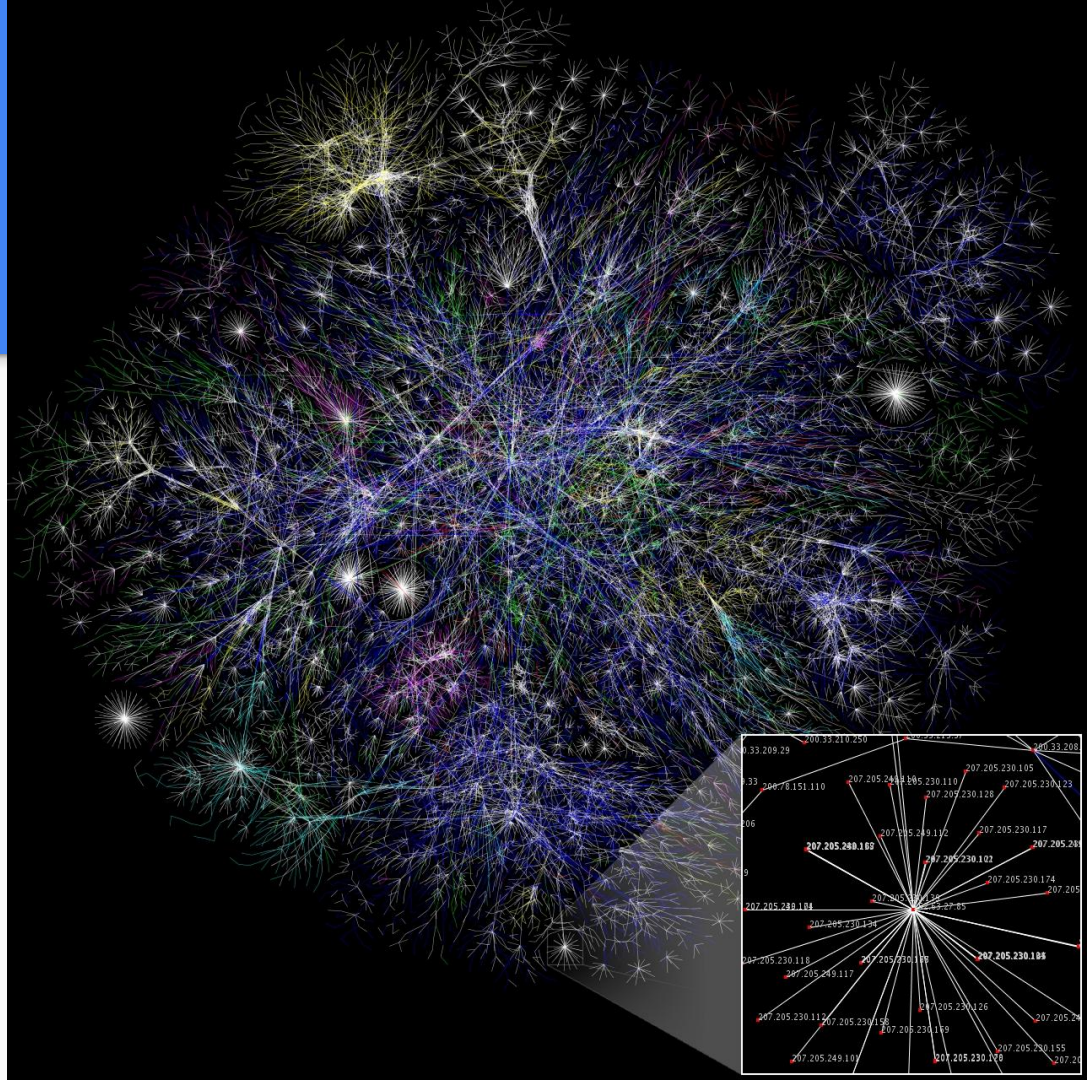
März 1972



Juli 1977

Today it’s a bit different

List of hosts by country.



Going over Exam #2

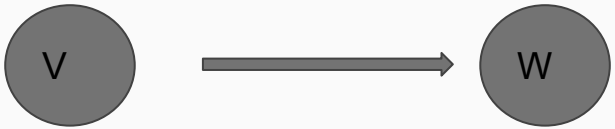
Graphs and Graph Algorithms

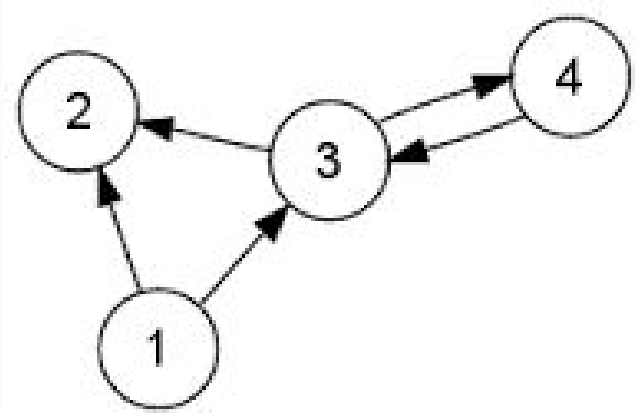
- What is a graph?

  A graph G = (V, E) consists of a set of vertices, V, and a set of edges, E. Each edge is a pair (v, w), where v, w $\in$ V.

Directed graphs - digraphs

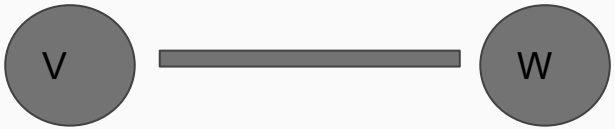
- Directed graphs are when the edges are directed.
- This means they have a front and a back, normally shown as an arrow
- Where have we seen these already?

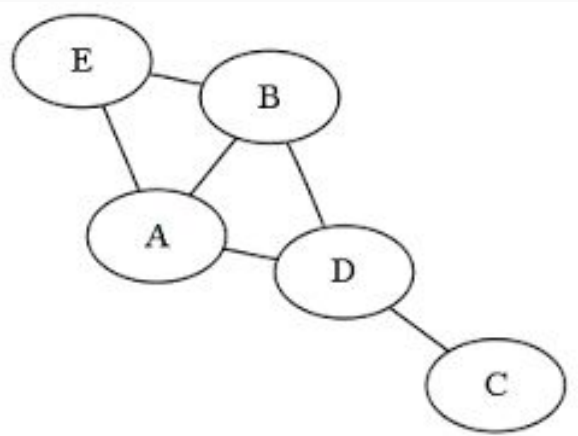




Undirected Graphs

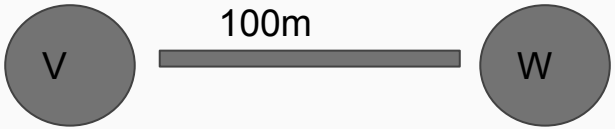
- Edges do not have a front and back, normally shown with a line
- This below is (v, w)

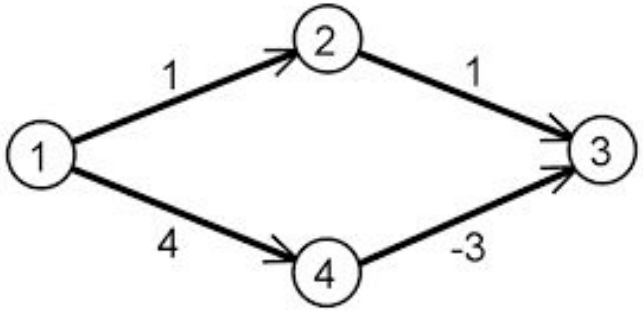




Weight or Cost of an edge

- Edges can carry a cost to traverse them
  - For example, two intersections are connected and the cost is how many meters long the connecting road is



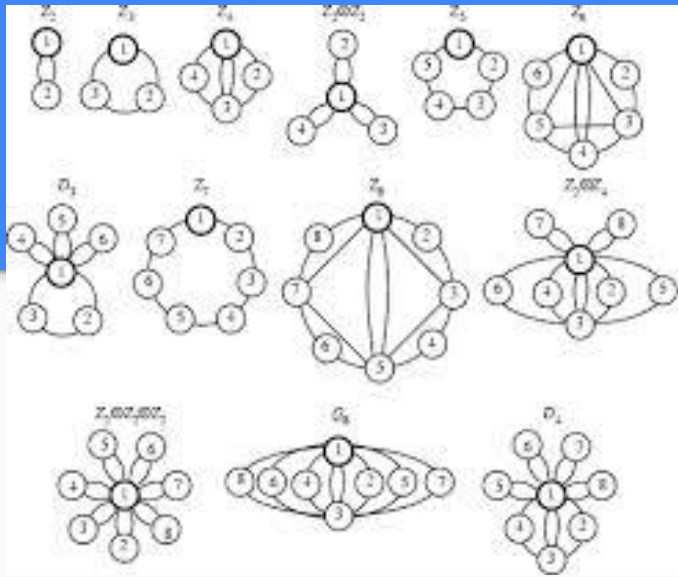


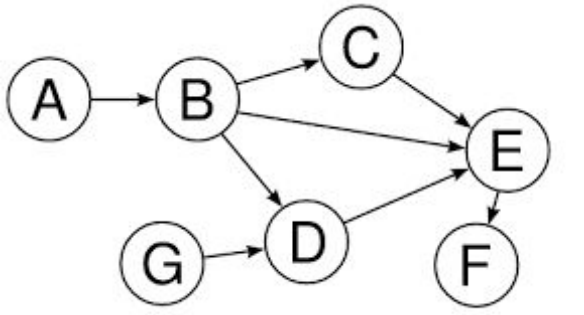
Paths

- A path is a sequence of vertices
  - $w_1, w_2, w_3, \ldots, w_N$ such that $(w_i, w_{\{i+1\}}) \in E$ for $1 \leq i < N$
- The length of the path is the number of edges on the path (not vertices!)
  - So the length is equal to N-1

Cycles

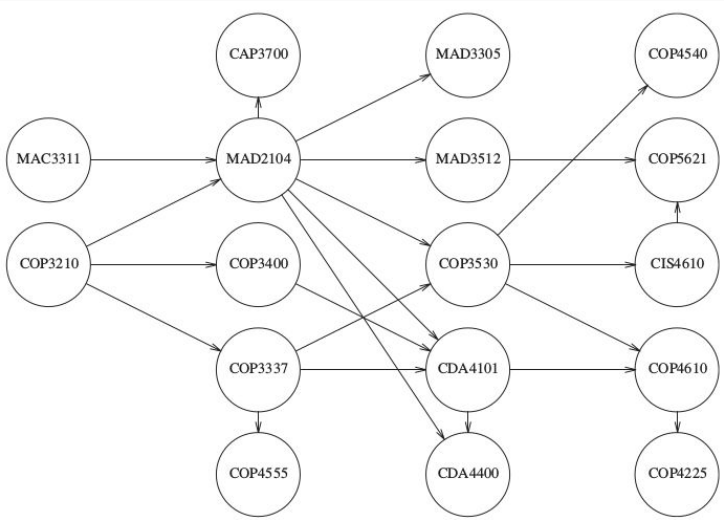
- Directed graph of at least length 1
  - such that $w_1 == w_N$
- Need to be avoided in finding paths





Topographical Sort

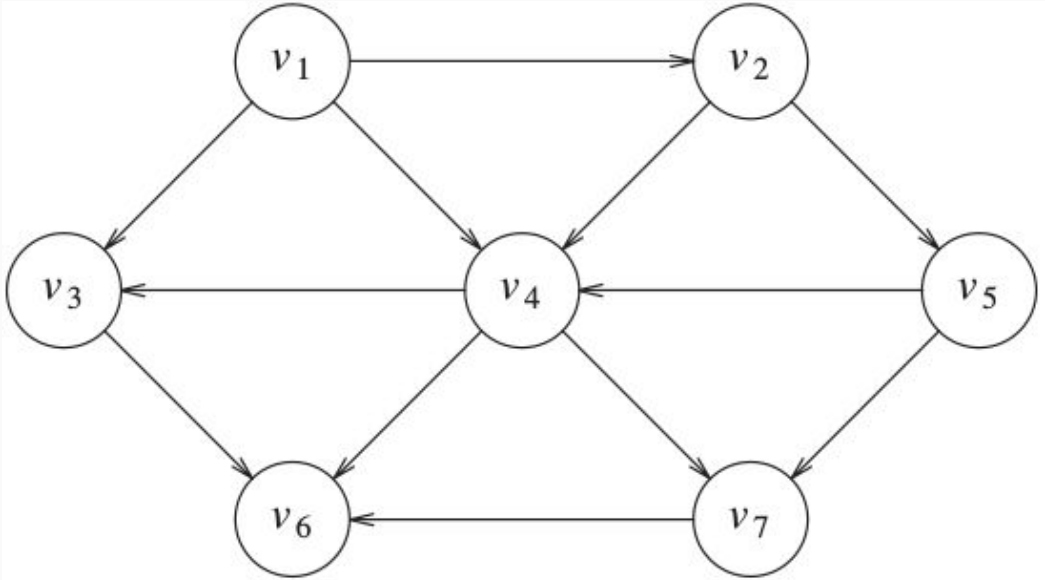
- Topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j , then v_j appears after v_i in the ordering
- Can’t work if there’s a cycle in the graph
- Does not guarantee a unique ordering
- Basically discovers a dependency list



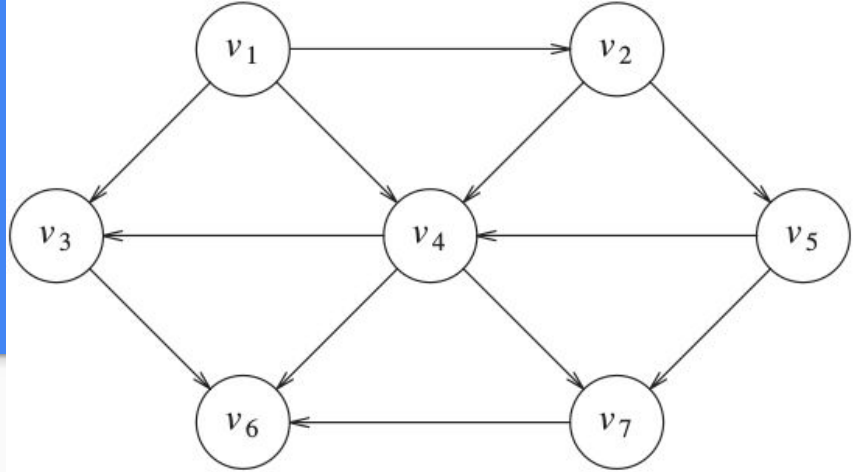
Topographical Sorting Example

{v_1 , v_2 , v_5 , v_4 , v_3 , v_7 , v_6} and {v_1 , v_2 , v_5 , v_4 , v_7 , v_3 , v_6} are both valid topological orderings

1) Find node with no in edges
   a) Indegree of zero
   b) Called a “source node”
2) Print out node && remove from graph
3) Repeat



Our toposort example



- Put nodes in a heap by “in” edge degree

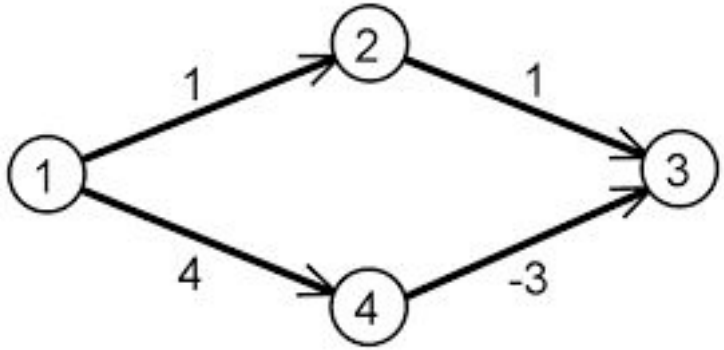
First column is Toposort output

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	
V5 - 0	V3 - 1	V6 - 3	V4 - 1	V7 - 2		
V4 - 0	V3 - 1	V6 - 3	V7 - 1			
V7 - 0	V3 - 0	V6 - 2				
V3 - 0	V6 - 1					
V6 - 0						

Shortest Path Algorithms
<These are highly valuable to know>

- How to go across graph in shortest number of steps from A to B?
  - “Short” can be defined in lots of ways - entirely application dependent
  - This is where the cost of an edge truly starts to matter big time

What’s shortest path from 1 to 3?



Formally, this is:

Single-Source Shortest-Path Problem:

Given as input a weighted graph, G = (V, E), and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

Cost of a path is:

Associated with each edge (v_i , v_j) is a cost $c_{\{i,j\}}$ to traverse the edge. The cost of a path:

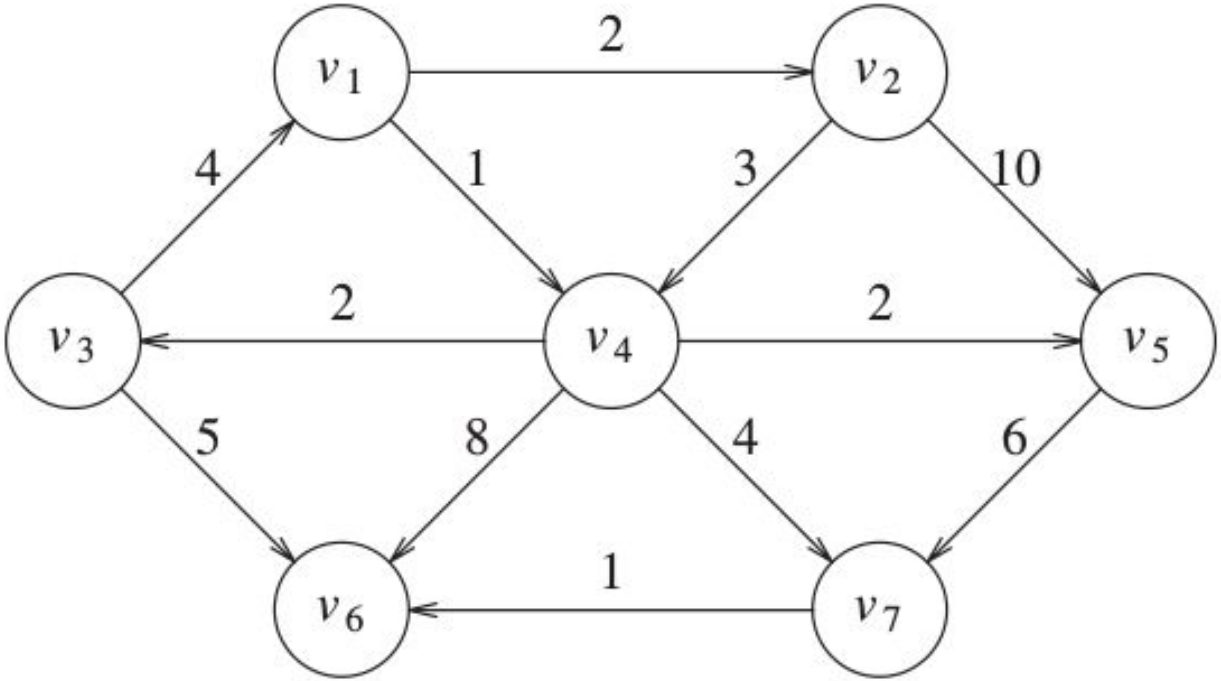
$$v_1 v_2 \ldots v_N \text{ is } \sum_{i=1}^{N-1} c_{i,i+1}$$

Simple, all positive edge example

v_1 to v_6?

v_1 -> v_4 -> v_7 -> v_6

Sum cost of 6



The problem of negative edges

Go: v_5 -> v_4

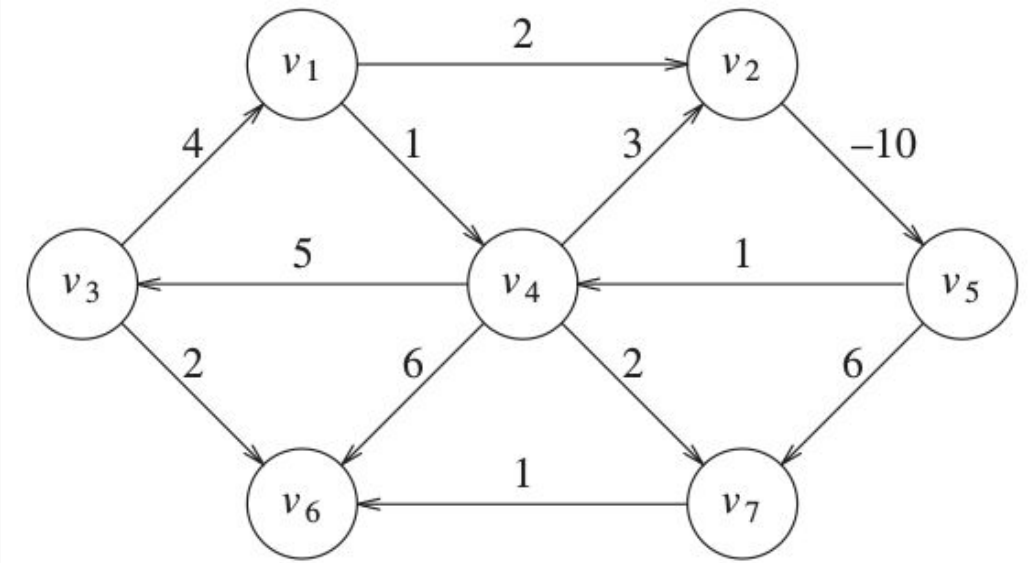
Takes 1 right?

What about:

v_5 -> v_4 -> v_2 -> v_5 -> v_4?

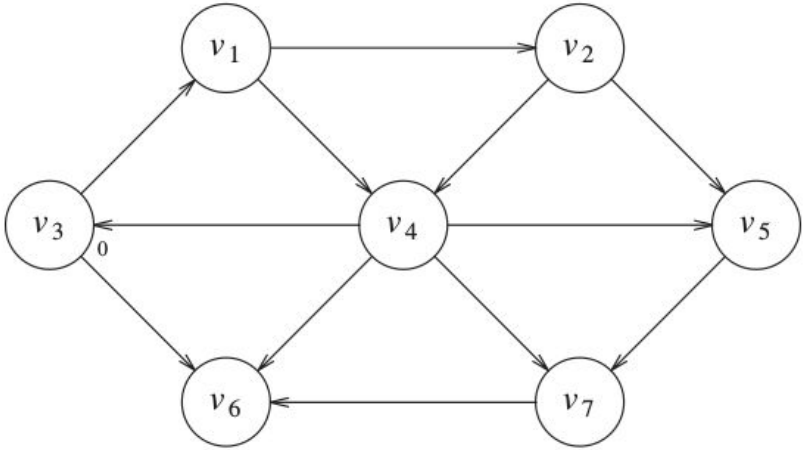
Perhaps you go around again?

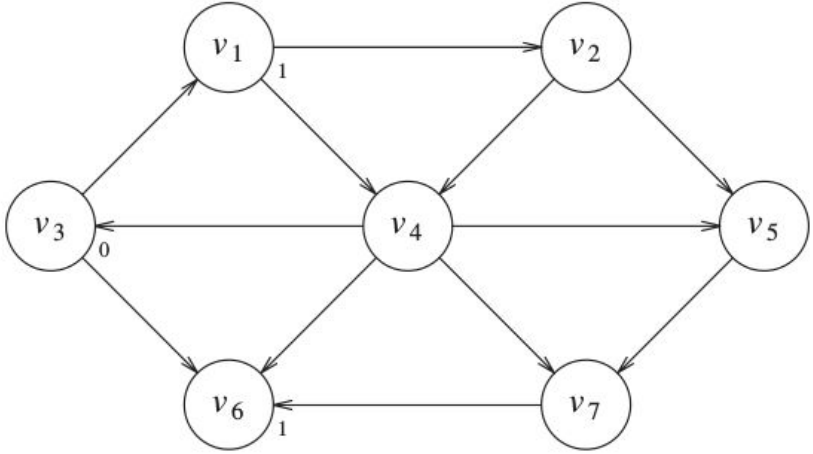
When a negative cycle exists, shortest paths are not defined!

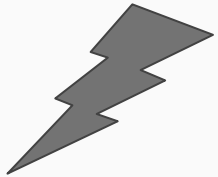


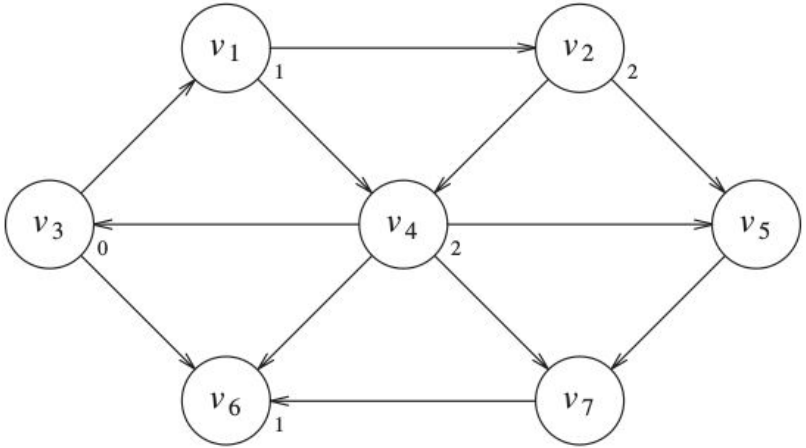
Unweighted Shortest Paths

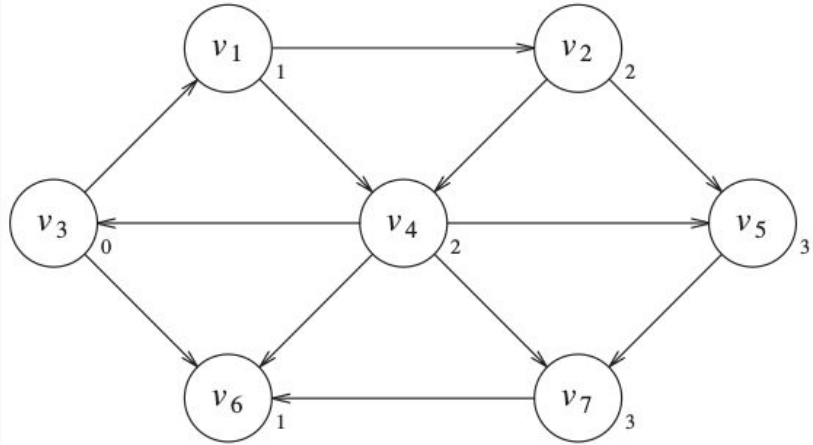
- Only care about number of edges in path, not their costs (cost == 1)
- Mark starting node (s) with length 0
- Look at all adjacent vertices with distance 1 from s
- Repeat for all vertices at distance 2, then 3, etc
- Once all nodes are marked, you're finished
- This is a breadth first search: the network is examined in layers, starting from a root node. Basically, level order traversal for trees
- Final result is all vertices are marked with distance from initial s
- Done in O(|E| + |V|) time


v_1
v_2
v_3
0
v_4
v_5
v_6
v_7


v_1
1
v_2
v_3
0
v_4
v_5
v_6
1
v_7




v_1
1
v_2
2
v_3
0
v_4
2
v_5
v_6
1
v_7


v_1
1
v_2
2
v_3
0
v_4
2
v_5
3
v_6
1
v_7
3

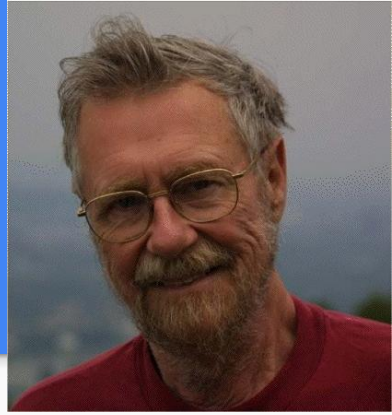
Unweighted path algorithm

```
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

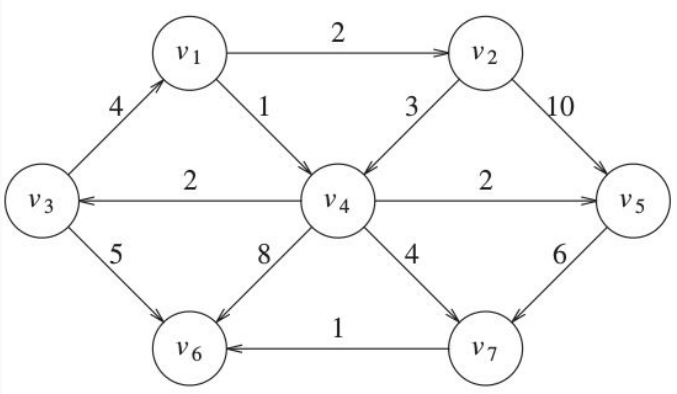
What if there's weights to consider? Never fear! Dijkstra is here!



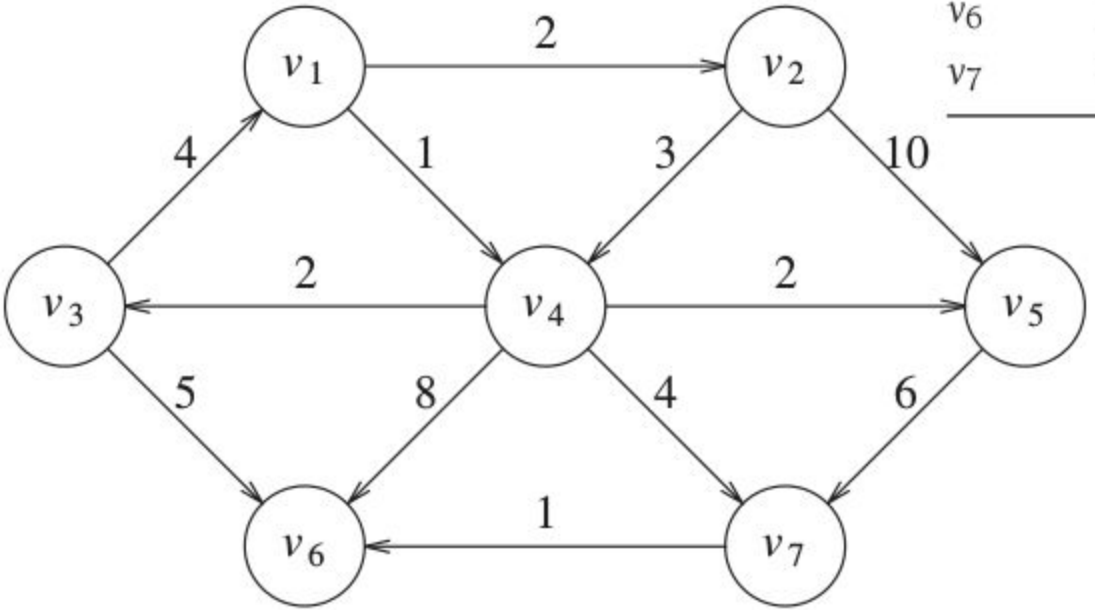
- Dijkstra's Algorithm takes into account edge weights for finding paths
- Don't just keep the raw distance, but tally up the cost to get there
- Greedy algorithm - follow lowest cost path first every time.
  - Queue is sorted by shortest path so far (priority queue time!)
- Again, keep a table of the vertices and their costs. Start them at INF
  - If a node popped off of the queue shortens another node's path then benefits cascade down the chain automagically
- Heavily used in network routing and shortest path network choices

Algorithm definition

- Select vertex v which has the smallest d_v among unknown vertices
  - Declares shortest path from s to v is known
  - If unweighted:
    - set $d_w = d_v + 1$ (if d_w = INF), thus lowering value of d_w if v was shorter path
  - If weighted
    - Set $d_w = d_v + c_{\{v,w\}}$ (if this is an improvement for d_w)

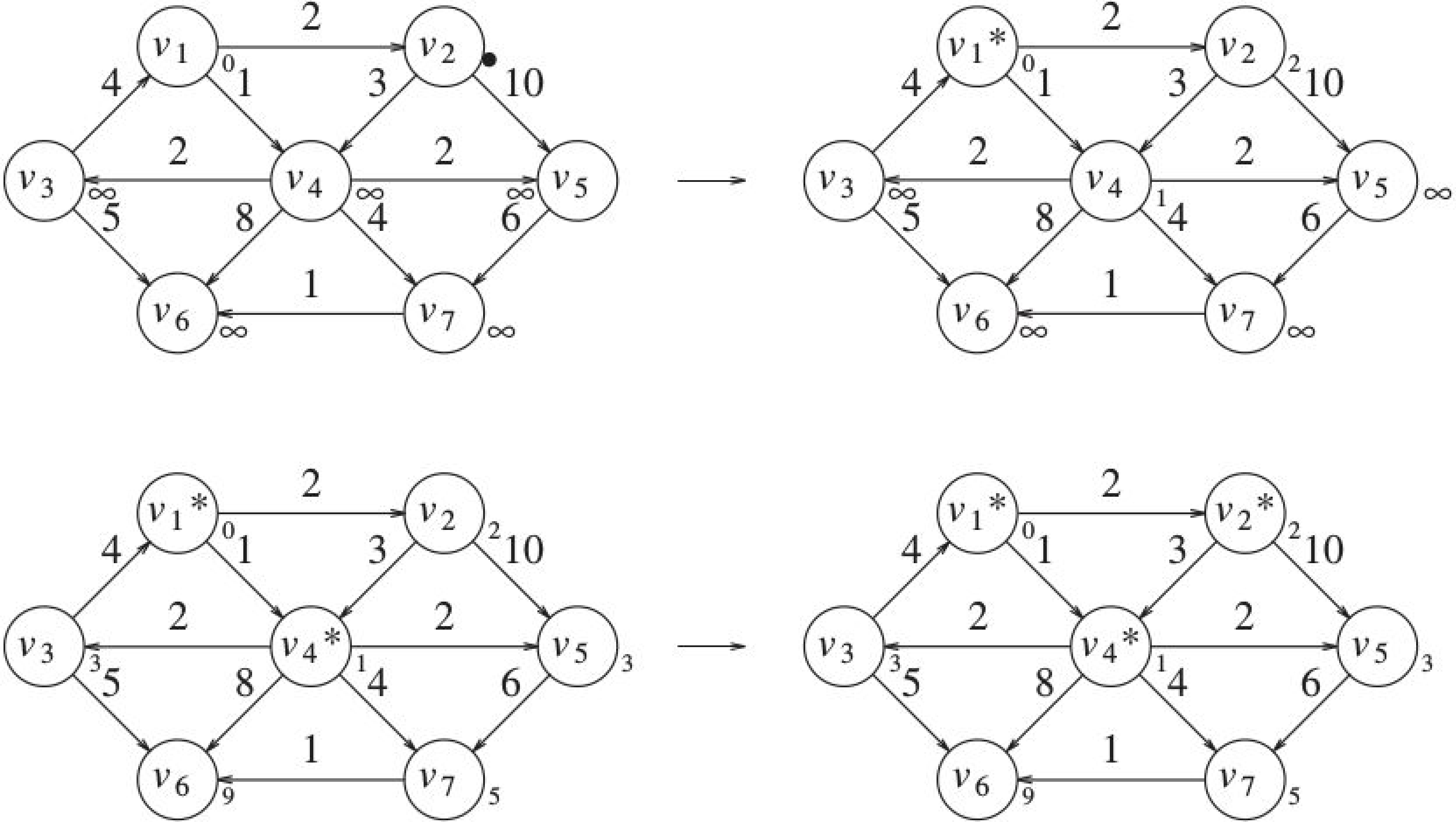


Starting example at v_1

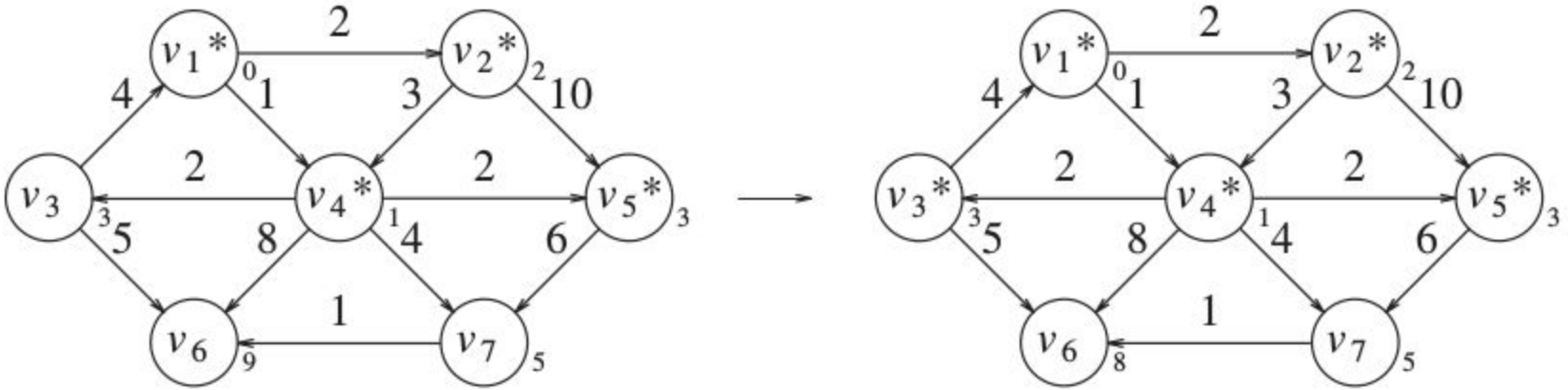


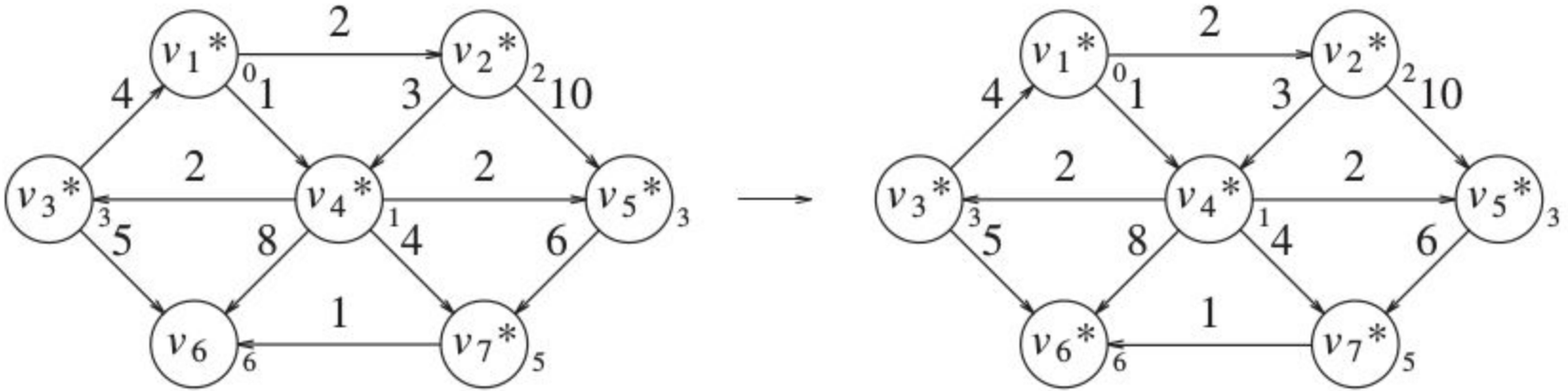
v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0



v1
v2
v3
v4
v5
v6
v7
v1*
v2
v3
v4
v5
v6
v7
v1*
v2
v3
v4*
v5
v6
v7
v1*
v2*
v3
v4*
v5
v6
v7


v_1*
v_2*
v_3
v_4*
v_5*
v_6
v_7
2
4
1
3
10
2
2
5
8
4
6
1
0
2
3
1
2
9
5
v_1*
v_2*
v_3*
v_4*
v_5*
v_6
v_7
2
4
1
3
10
2
2
5
8
4
6
1
0
2
3
1
2
8
5


v_1*
v_2*
v_3*
v_4*
v_5*
v_6*
v_7*
2
4
1
3
10
2
2
5
8
4
6
1
0
2
3
1
2
6
5
v_1*
v_2*
v_3*
v_4*
v_5*
v_6*
v_7*
2
4
1
3
10
2
2
5
8
4
6
1
0
2
3
1
2
6
5

Dijkstra's Code

```
struct Vertex
{
    List      adj;     // Adjacency list
    bool      known;
    DistType  dist;    // DistType is probably int
    Vertex    path;    // Probably Vertex *, as mentioned above
        // Other data and member functions as needed
};
void Graph::printPath( Vertex v )
{
    if( v.path != NOT_A_VERTEX )
    {
        printPath( v.path );
        cout << " to ";
    }
    cout << v;
}
```

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

For Friday: NO CLASS!!
For Monday: More graphs

Also, starting to talk about categories of algorithmic complexity.