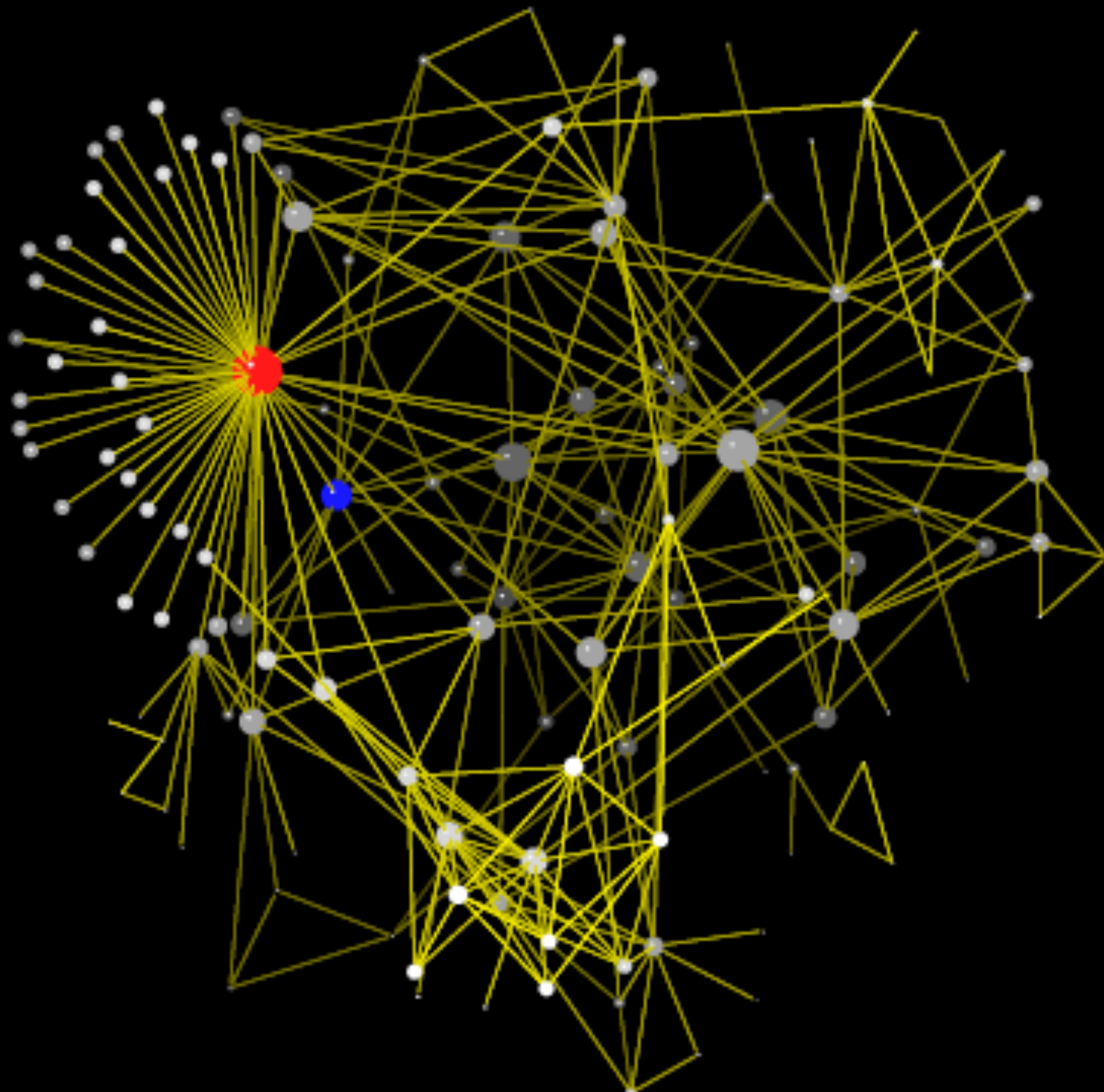
CSCI 335 Software Design and Analysis III

Graph Algorithms

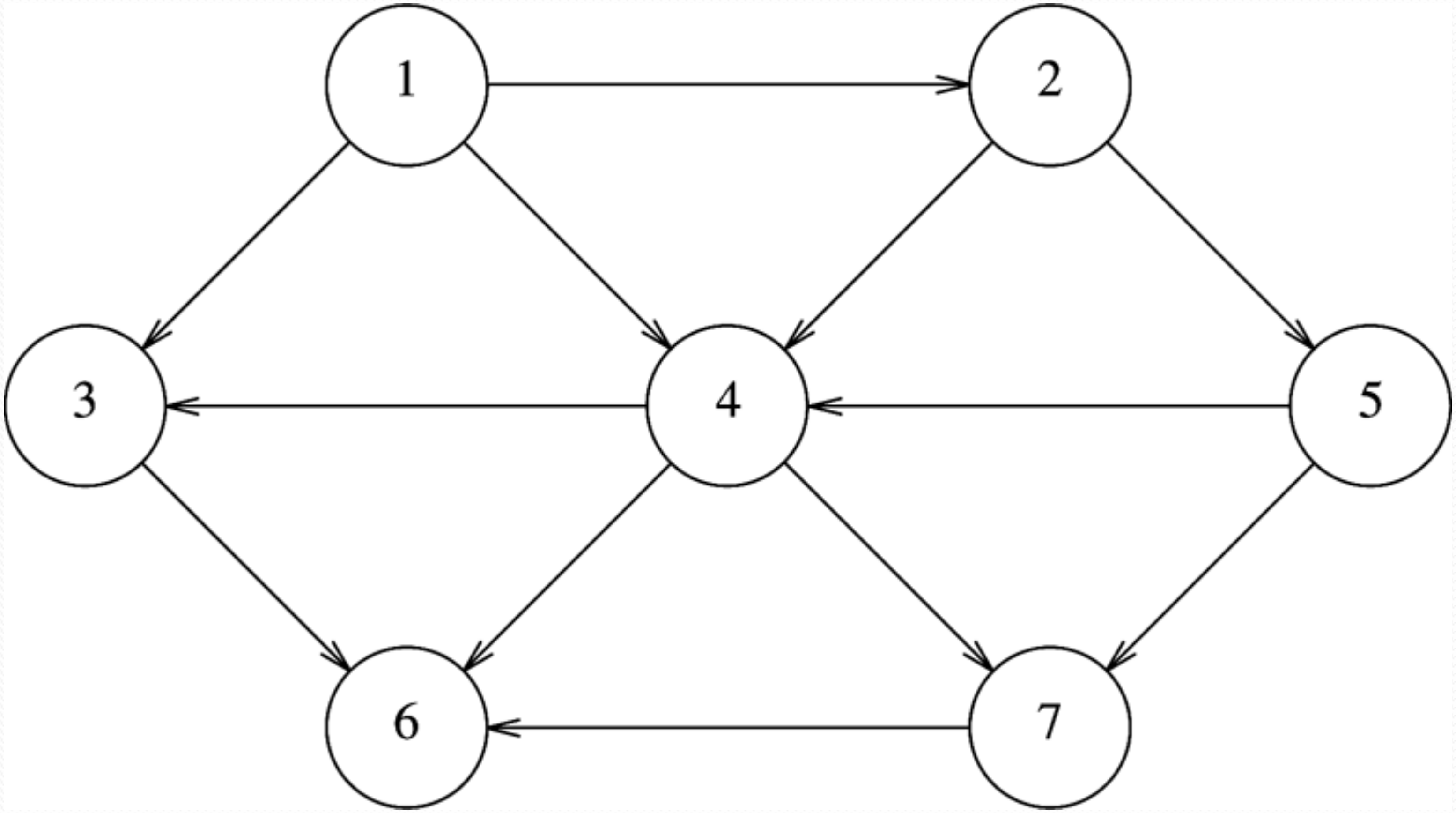
(Topological Sorting, Shortest Paths, Dijkstra)



Graphs

- G=(V,E)
- Set V={v1,v2,…}: vertices
- Set E={(vi,vj) | vi, vj in V}: edges
- Weighted graph: weight on edge (E={(vi,vj,wij): vi, vj in V, wij is the weight})
- Directed vs. undirected graphs
- Path {v1,v2,…,vn}
- Length of a path
- Loop
- Simple path
- Cycle
- Directed Acyclic Graph (DAG)
- Strongly connected directed graph
- Weakly connected directed graph
- Complete graph
- Examples…

Graphs (example)



Graphs: representation

- Adjacency matrix A

  A[u][v] is True iff (u,v) is an edge

  of

  A[u][v] is the cost c of the edge (u,v) or ∞ if no-edge

- Space: $\Theta(|V|^2)$
  - OK for dense graphs
  - BAD for sparse graphs (e.g. $|E| = \Theta(|V|)$)

Graphs: representation

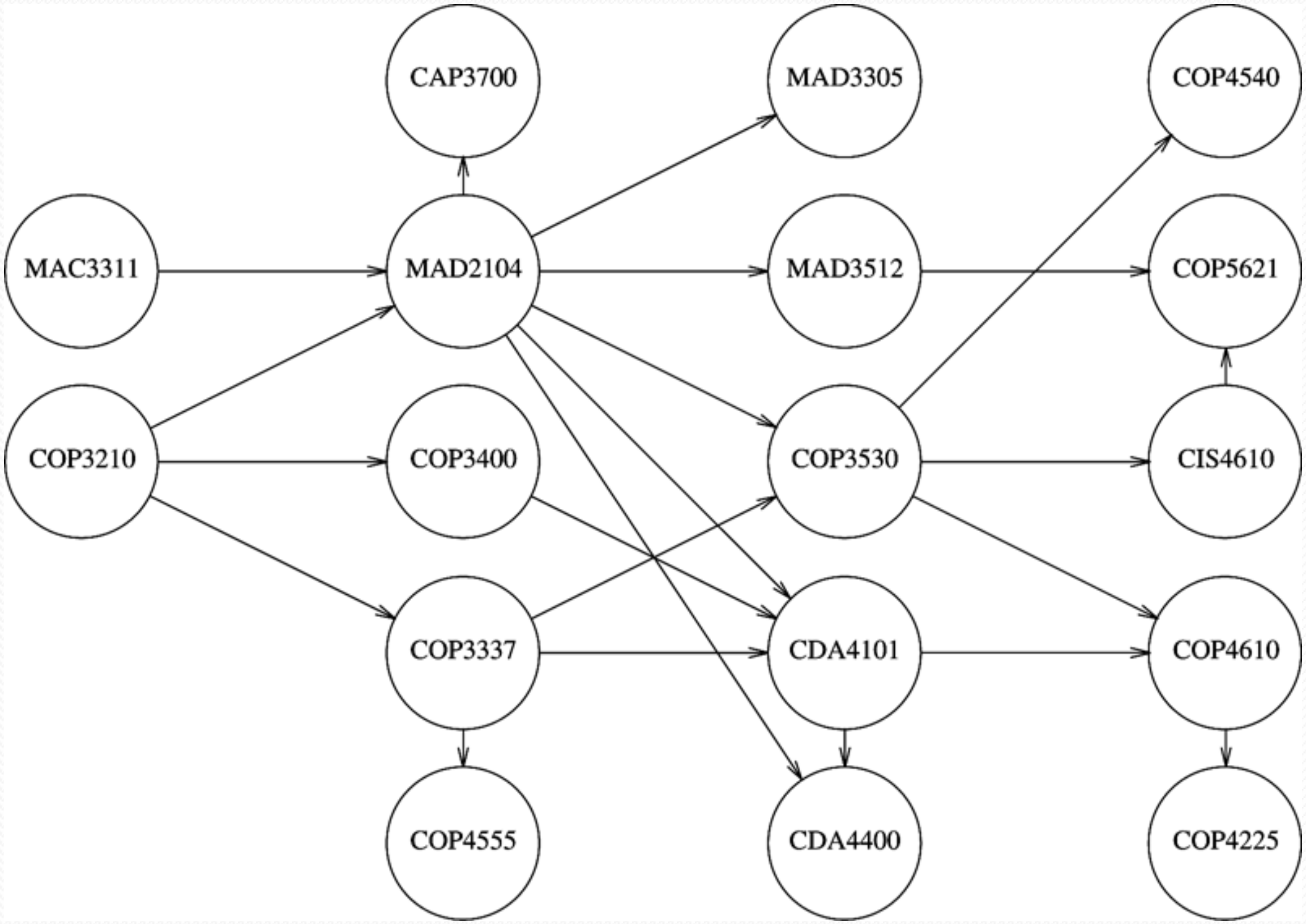
- Adjacency list:

  For each vertex -> list of adjacent vertices
- Space: $\Theta(|V|+|E|)$: linear
- GOOD for sparse graphs

Graphs: adjacency list

1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

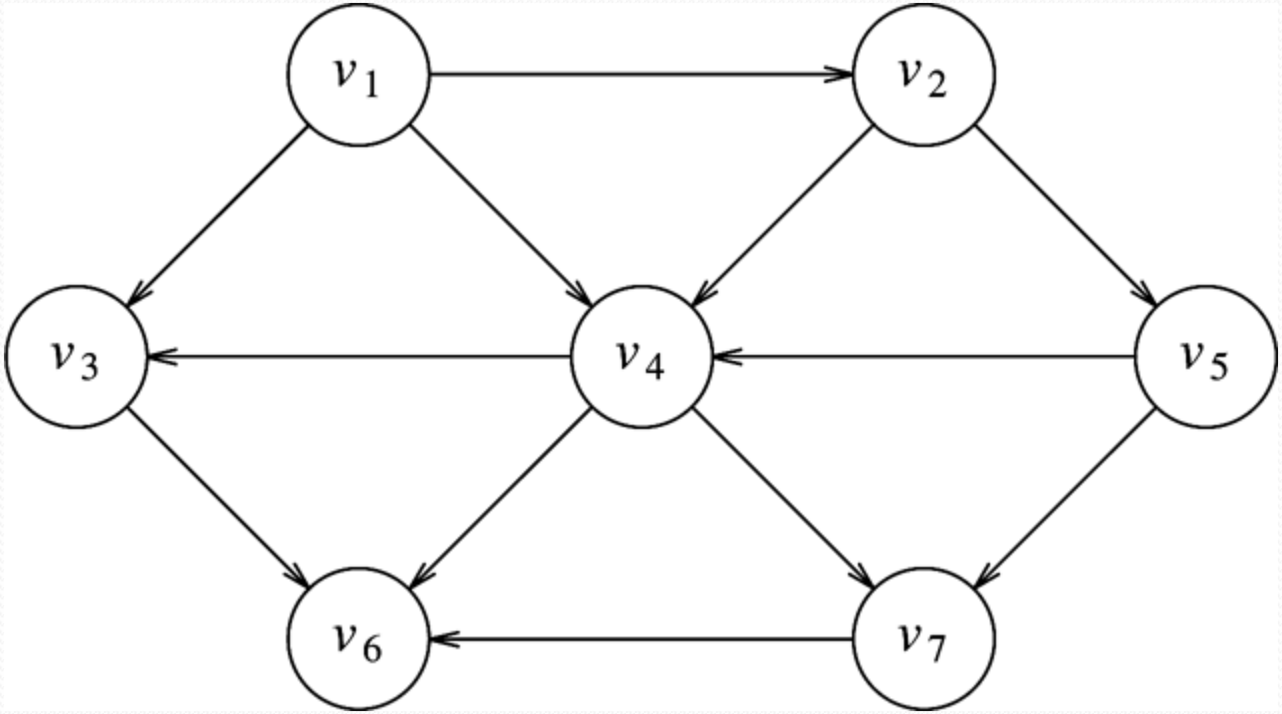
Topological sort



Topological sort

- Defined for **directed graphs**
- Not unique
- Does not exist if graph has at least one cycle.

Topological sort



Topological sort (algorithm)

- Use the **indegree** of a vertex v (i.e. number of vertices pointing to it): the number of edges (u,v)
- <u>Algorithm</u>
  1. Compute **indegree** of all vertices
  2. Choose a vertex with **indegree** = 0
  3. Remove vertex and edges pointing to it
  4. Update **indegree** of vertices
  5. Go to 2

  Stop when all vertices are considered

  Throw exception if cycle (how is it detected?)

Topological sort algorithm

```
void Graph::TopologicalSort() {
    for (int counter = 0; counter < num_vertices_; ++counter) {
        Vertex v = FindNewVertexOfIndegreeZero();
        if (v == NOT_A_VERTEX)
            throw CycleFoundException();
        v.top_num_ = counter;
        for each Vertex w adjacent to v
            w.indegree_--;
    }
}
```

Topological sort (algorithm)

- Analysis:

  $O(|V|^2)$ <-findNewVertexOfDegreeZero() is $O(|V|)$
- Improvement?
  - Keep vertices of **indegree** zero in Queue
  - Dequeue vertex v
  - Update **indegree**s of vertices connected to v

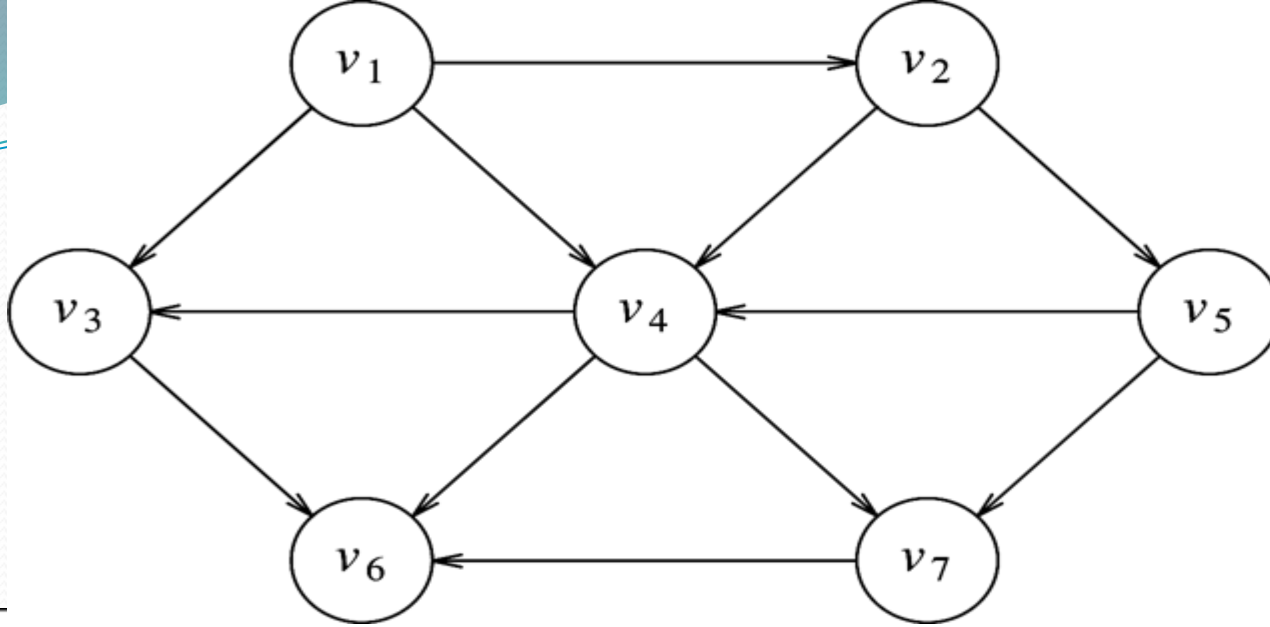
Complexity?

```
void Graph::TopologicalSort() {
      Queue<Vertex> q;
      int counter = 0;
      for each Vertex v
         if (v.indegree_ == 0) q.enqueue(v);

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

           for each Vertex w adjacent to v
               if (--w.indegree_ == 0) q.enqueue(w);
       }
       if (counter != num_vertices_)
         throw CycleFoundException()
}
```

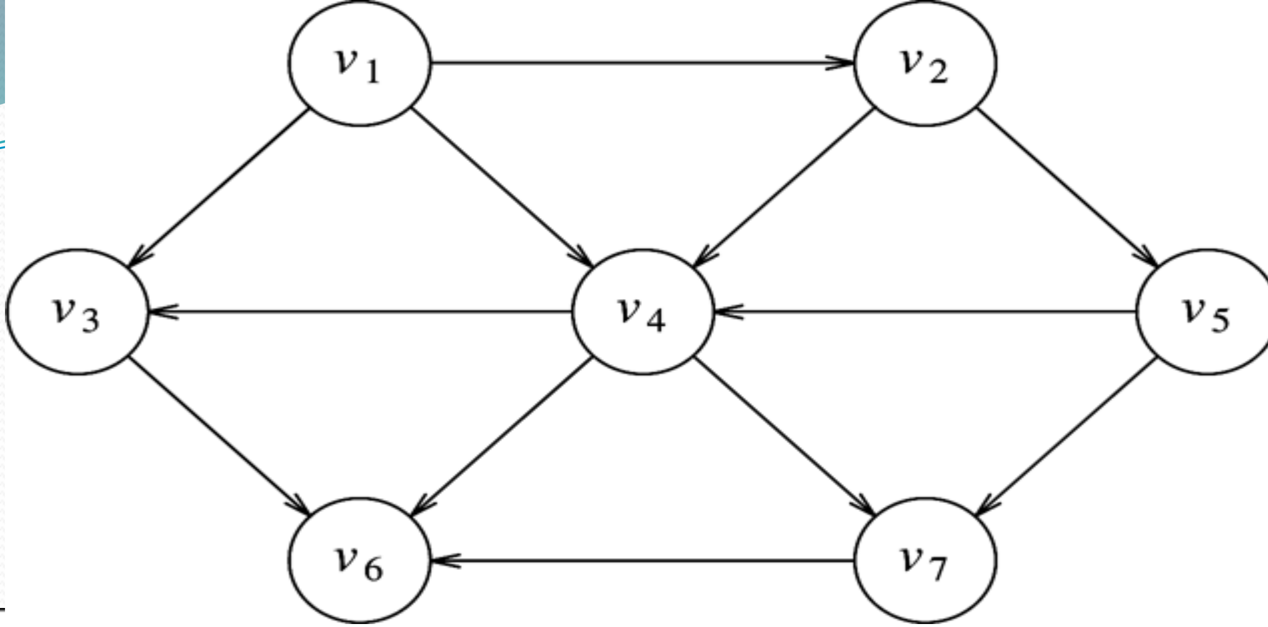
```
void Graph::TopologicalSort() {
      Queue<Vertex> q;
      int counter = 0;
      for each Vertex v
         if (v.indegree_ == 0) q.enqueue(v);

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

           for each Vertex w adjacent to v
               if (--w.indegree_ == 0) q.enqueue(w);
       }
       if (counter != num_vertices_)
         throw CycleFoundException()
}
```

Complexity?

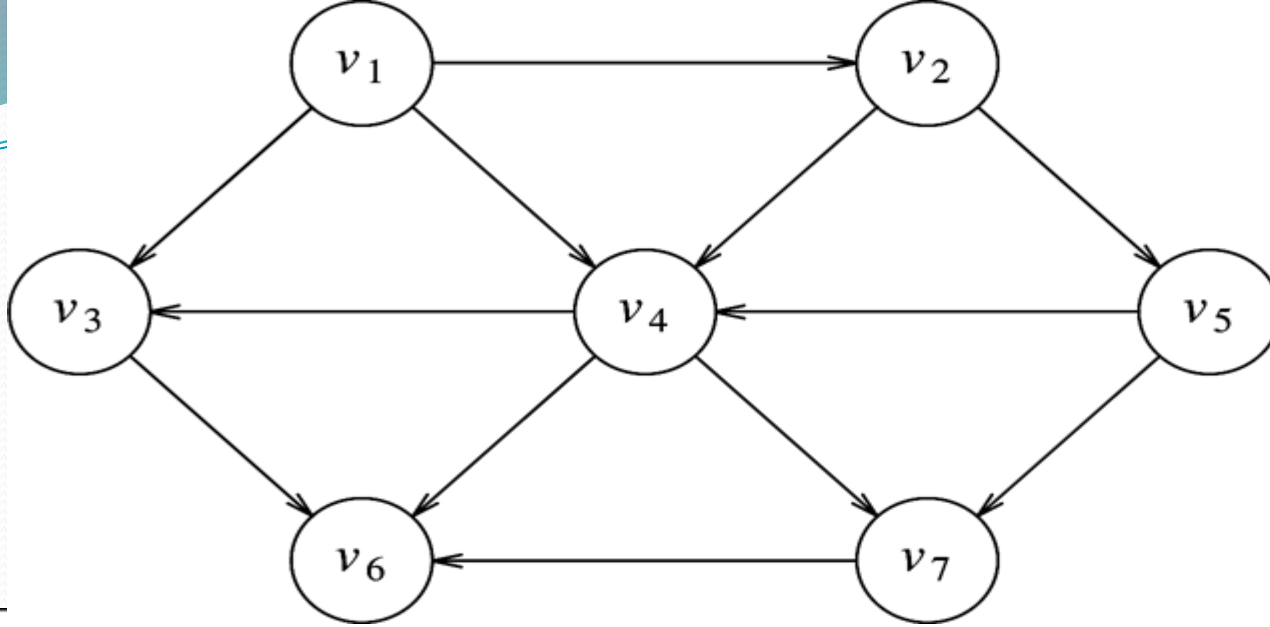
$O(\ |V| + |E|\ )$



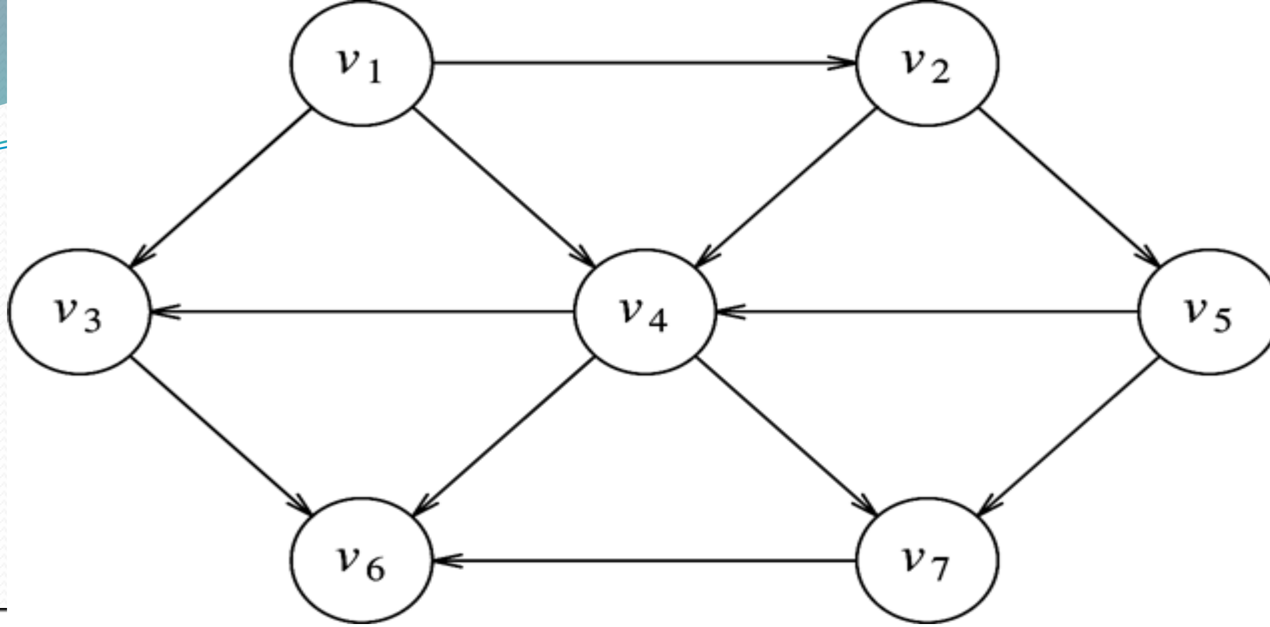
	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0						
v_2	1						
v_3	2						
v_4	3						
v_5	1						
v_6	3						
v_7	2						
Enqueue	v_1						
Dequeue	v_1						



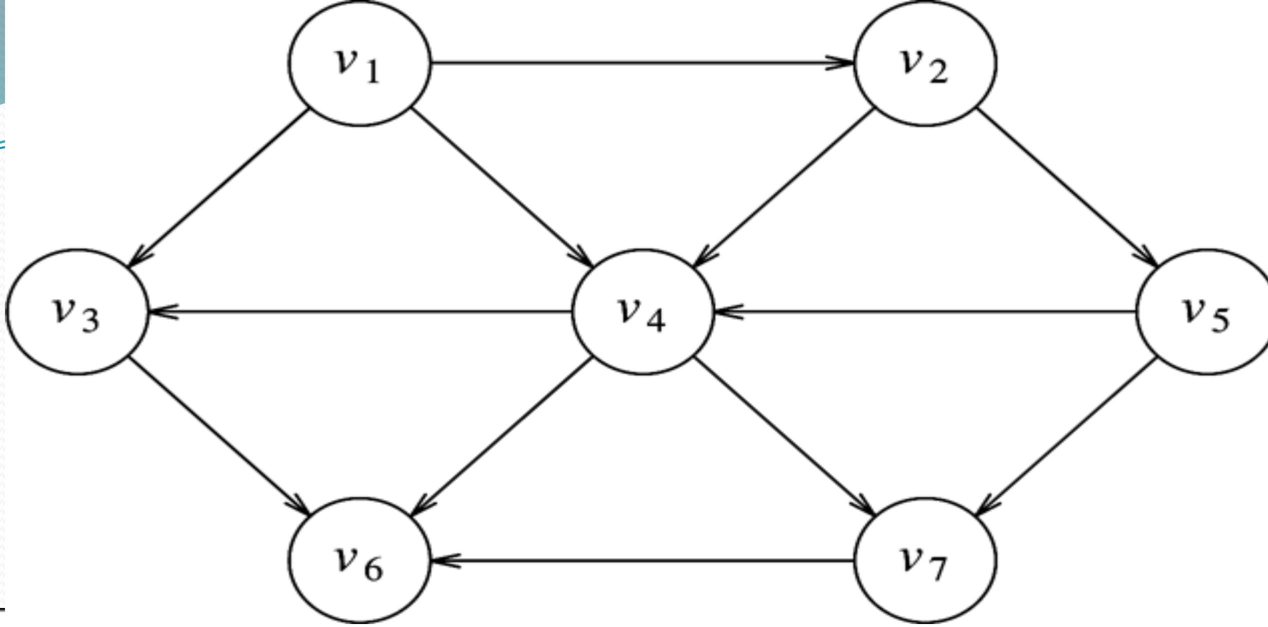
	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0*					
v_2	1	0					
v_3	2	1					
v_4	3	2					
v_5	1	1					
v_6	3	3					
v_7	2	2					
Enqueue	v_1	v_2					
Dequeue	v_1	v_2					



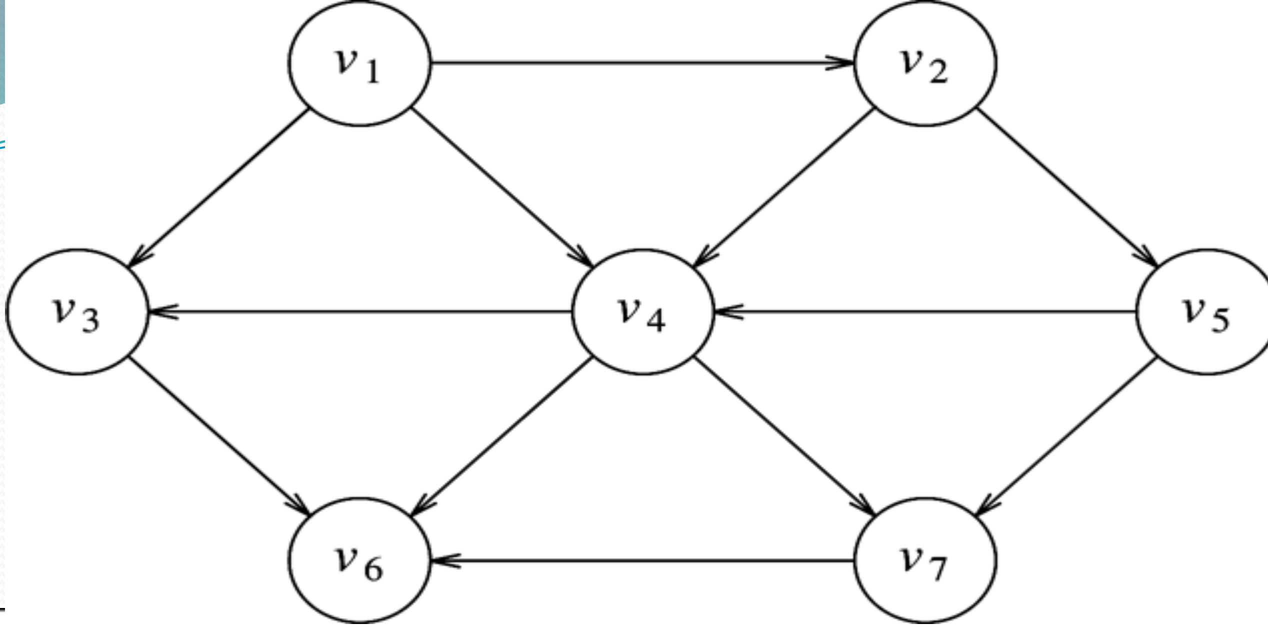
	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0*	0*				
v_2	1	0	0*				
v_3	2	1	1				
v_4	3	2	1				
v_5	1	1	0				
v_6	3	3	3				
v_7	2	2	2				
Enqueue	v_1	v_2	v_5				
Dequeue	v_1	v_2	v_5				



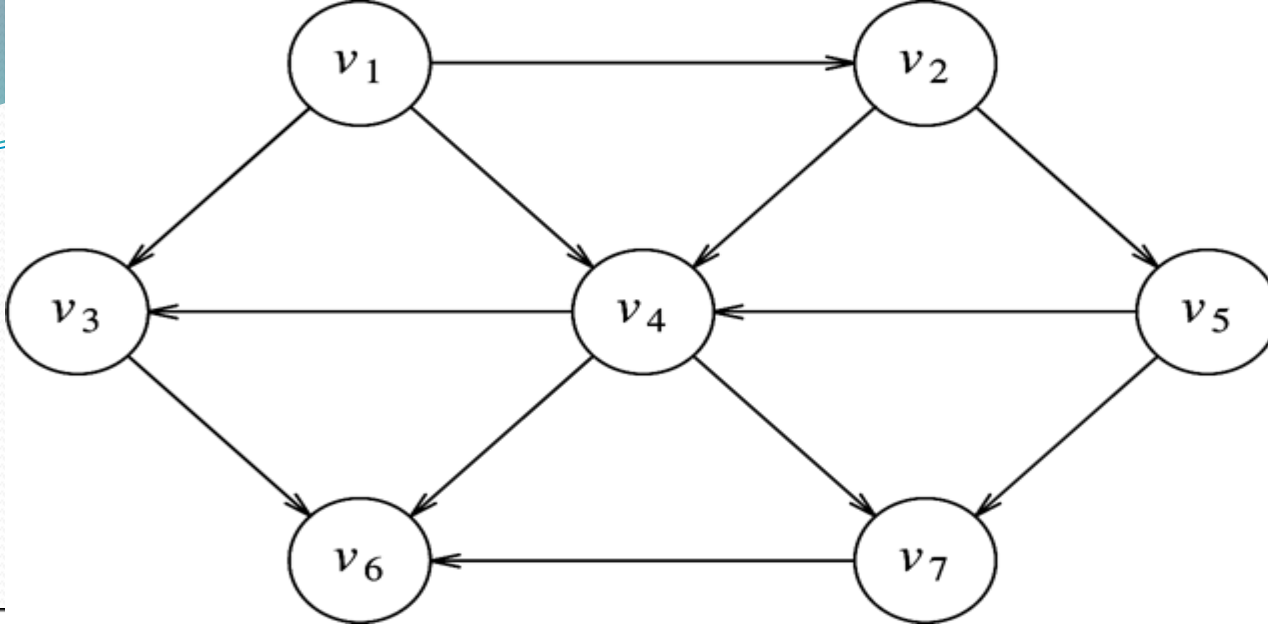
	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0*	0*	0*			
v_2	1	0	0*	0*			
v_3	2	1	1	1			
v_4	3	2	1	0			
v_5	1	1	0	0*			
v_6	3	3	3	3			
v_7	2	2	2	1			
Enqueue	v_1	v_2	v_5	v_4			
Dequeue	v_1	v_2	v_5	v_4			



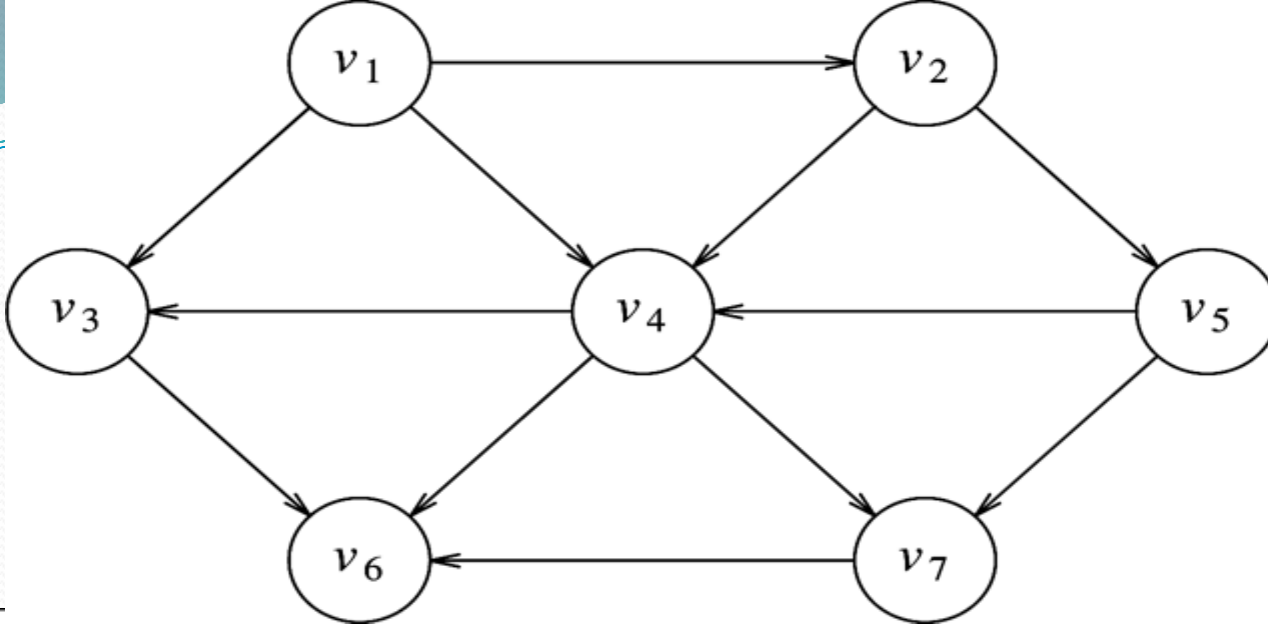
	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0*	0*	0*	0*		
v_2	1	0	0*	0*	0*		
v_3	2	1	1	1	0		
v_4	3	2	1	0	0*		
v_5	1	1	0	0*	0*		
v_6	3	3	3	3	2		
v_7	2	2	2	1	0		
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		
Dequeue	v_1	v_2	v_5	v_4	v_3		



	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0*	0*	0*	0*	0*	
v_2	1	0	0*	0*	0*	0*	
v_3	2	1	1	1	0	0*	
v_4	3	2	1	0	0*	0*	
v_5	1	1	0	0	0*	0*	
v_6	3	3	3	3	2	1	
v_7	2	2	2	1	0	0	
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	



	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0*	0*	0*	0*	0*	0*
v_2	1	0	0*	0*	0*	0*	0*
v_3	2	1	1	1	0	0*	0*
v_4	3	2	1	0	0*	0*	0*
v_5	1	1	0	0	0*	0*	0*
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0*
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6



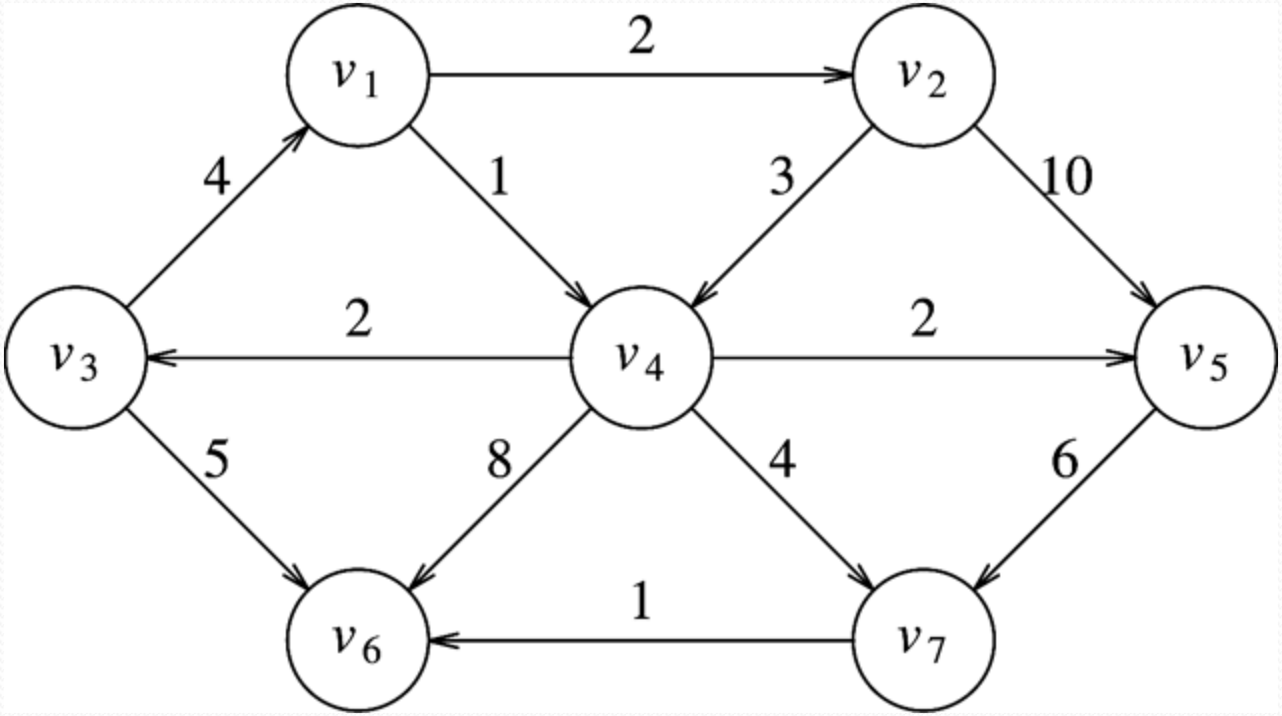
	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0*	0*	0*	0*	0*	0*
v_2	1	0	0*	0*	0*	0*	0*
v_3	2	1	1	1	0	0*	0*
v_4	3	2	1	0	0*	0*	0*
v_5	1	1	0	0	0*	0*	0*
v_6	3	3	3	3	2	1	0*
v_7	2	2	2	1	0	0	0*
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

Shortest Path Algorithms

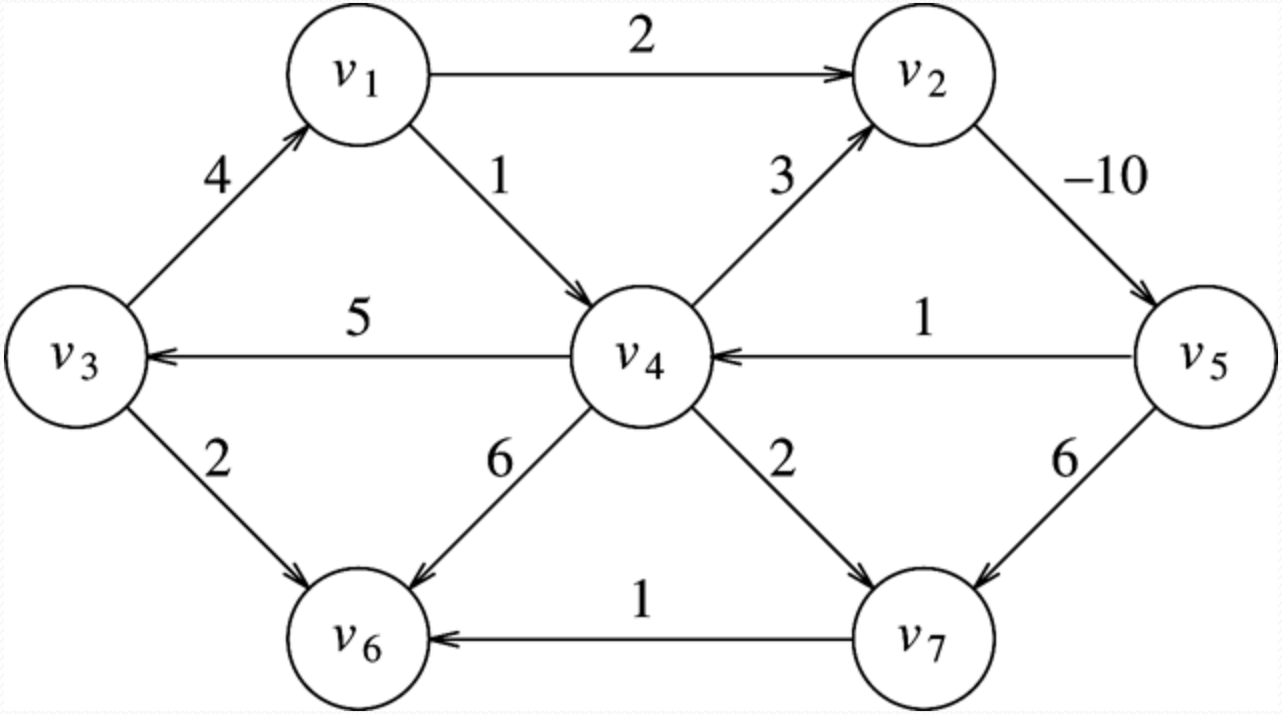
- **Weighted** (sum of edge weights) vs. **unweighted** (number of edges) path length.
- **Single source shortest path**:

  Given a graph G and a vertex s, find the shortest path from s to any other vertex of G.

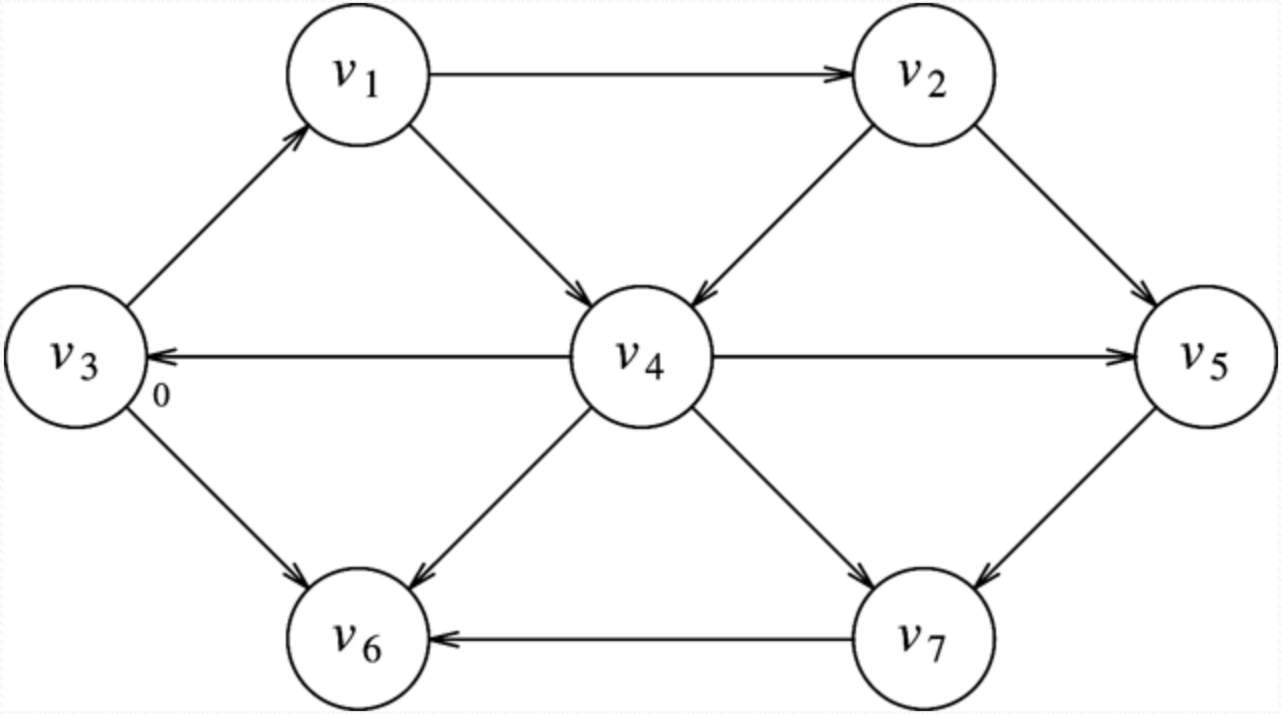
Shortest Path, positive weights



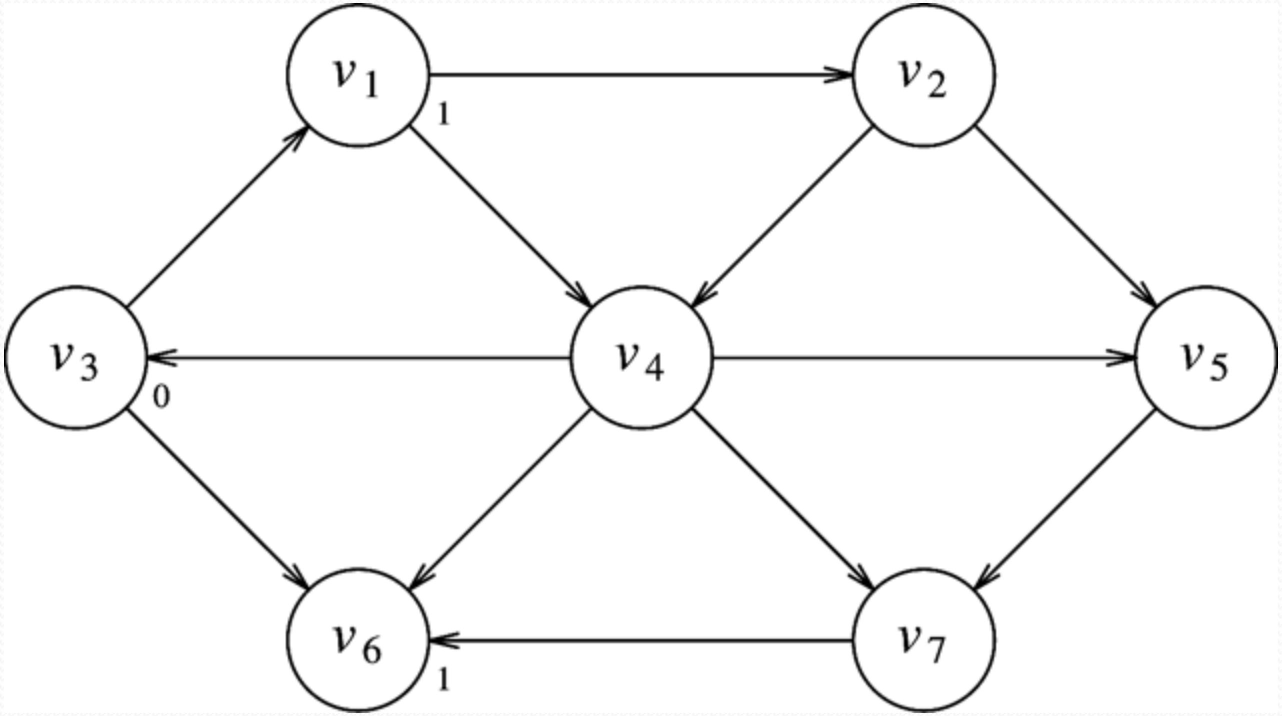
Negative weights?



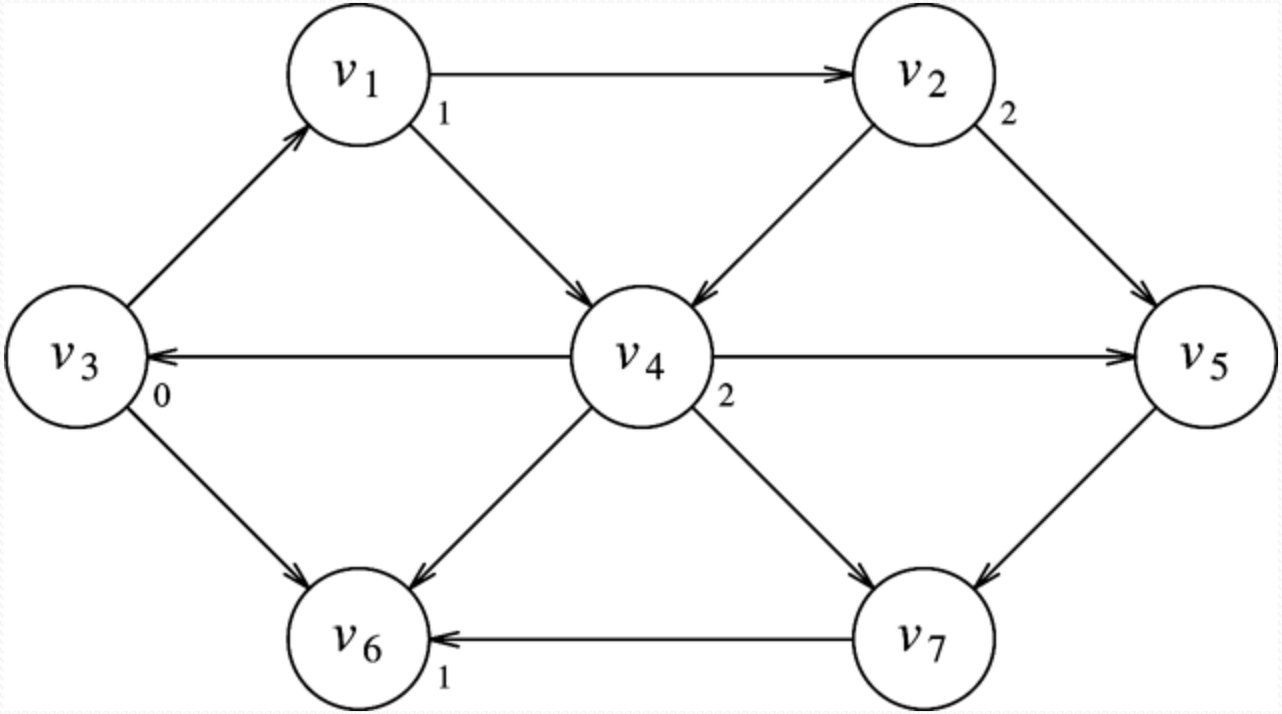
Unweighted Shortest Path



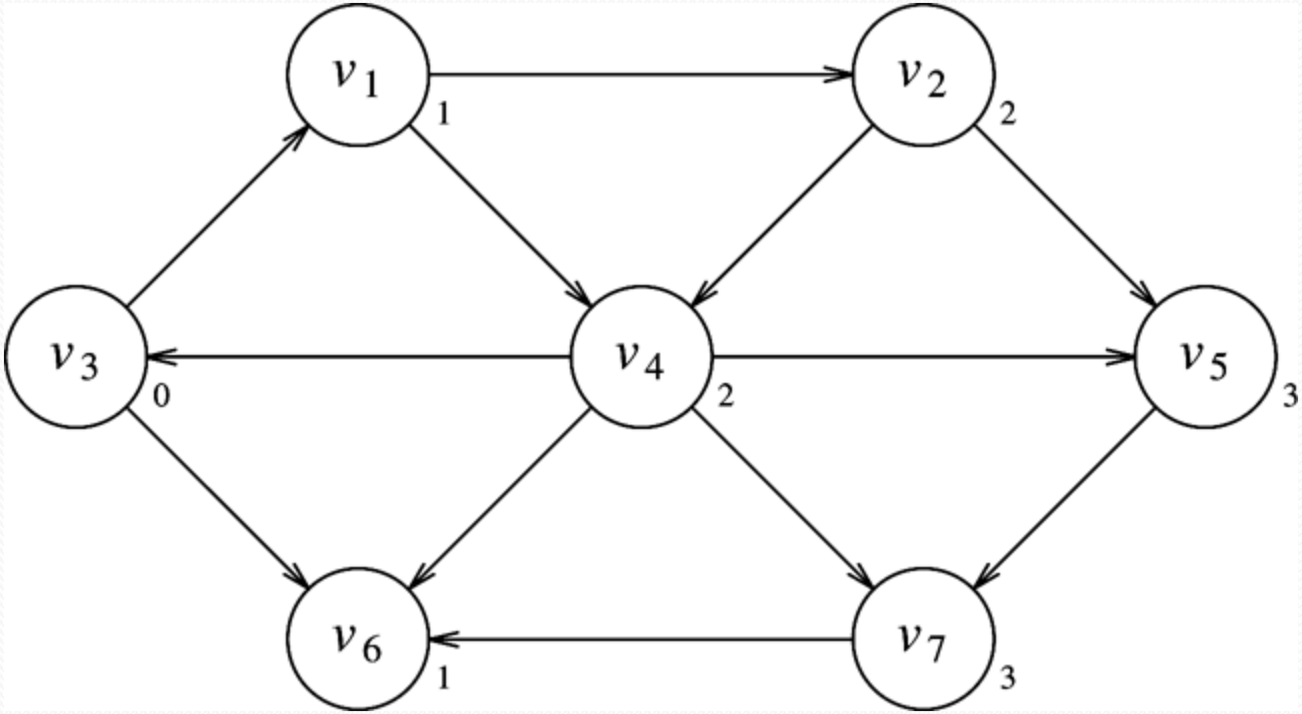
Unweighted Shortest Path



Unweighted Shortest Path



Unweighted Shortest Path



BREADTH FIRST SEARCH

Implementation

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

Initial configuration

Complexity?

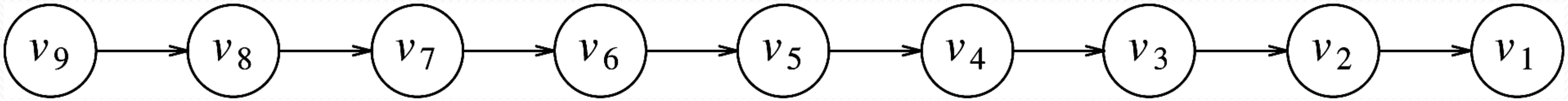
```
void Graph::UnweightedShortestPath(Vertex s) {
    for each Vertex v {
            v.distance_ = kInfinity;
            v.known_ = false;
    }
    s.distance_ = 0
    for (int curr_distance = 0; curr_distance < num_vertices_;
         ++curr_distance_) {
        for each Vertex v {
            if (!v.known_ && v.distance_ == curr_distance) {
                v.known_ = true;
                for each Vertex w adjacent to v {
                    if (w.distance_ == kInfinity) {
                        w.distance_ = curr_distance + 1;
                        w.path_ = v;
                    }
                }  // end for each Vertex w…
            }  // end if (!v.known…
        }  // end for each Vertex v
}
```

Complexity?

$O(|V|^2)$

```
void Graph::UnweightedShortestPath(Vertex s) {
    for (each Vertex v) {
            v.distance_ = kInfinity;
            v.known_ = false;
    }
    s.distance_ = 0
    for (int curr_distance = 0; curr_distance < num_vertices_;
         ++curr_distance_) {
        for (each Vertex v) {
            if (!v.known_ && v.distance_ == curr_distance) {
                v.known_ = true;
                for (each Vertex w adjacent to v) {
                    if (w.distance_ == kInfinity) {
                        w.distance_ = curr_distance + 1;
                        w.path_ = v;
                    }
                }  // end for each Vertex w…
            }  // end if (!v.known…
        }  // end for each Vertex v
}
```

BAD example



```
void Graph::UnweightedShortestPath(Vertex s) {
    Queue<Vertex> q;
    for each Vertex v {
        v.distance_ = kInfinity;
    }
    s.distance_ = 0;
    q.enqueue(s);
    while (!q.isEmpty()) {
        Vertex v = q.dequeue();
        for each Vertex w adjacent to v {
            if (w.distance_ == kInfinity) {
                w.distance_ = v.distance_ + 1;
                w.path_ = v;
                q.enqueue(w);
            }
        }  // end for
    }  // end while
}
```

Complexity?

BREADTH FIRST SEARCH

```
void Graph::UnweightedShortestPath(Vertex s) {
    Queue<Vertex> q;
    for each Vertex v {
        v.distance_ = kInfinity;
    }
    s.distance_ = 0;
    q.enqueue(s);
    while (!q.isEmpty()) {
        Vertex v = q.dequeue();
        for each Vertex w adjacent to v {
            if (w.distance_ == kInfinity) {
                w.distance_ = v.distance_ + 1;
                w.path_ = v;
                q.enqueue(w);
            }
        }  // end for
    }  // end while
}
```

Complexity?

$O(\ |V| + |E|\ )$

BREADTH FIRST SEARCH

v	Initial State *known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0
Q:		v_3	

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

	Initial State			v_3 Dequeued			v_1 Dequeued		
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
v_2	F	∞	0	F	∞	0	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4		

	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		

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

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

v	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
	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	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued		
	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
v_2	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0
v_4	F	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
v_6	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7		

v	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
	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	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
	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		

Weighted Shortest Path: **Dijkstra**

- GREEDY algorithm
  - GREEDY algorithms do not always work
  - **Dijkstra's** algorithm works fine

http://www.dgp.toronto.edu/~jstewart/270/9798s/Laffra/DijkstraApplet.html

https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

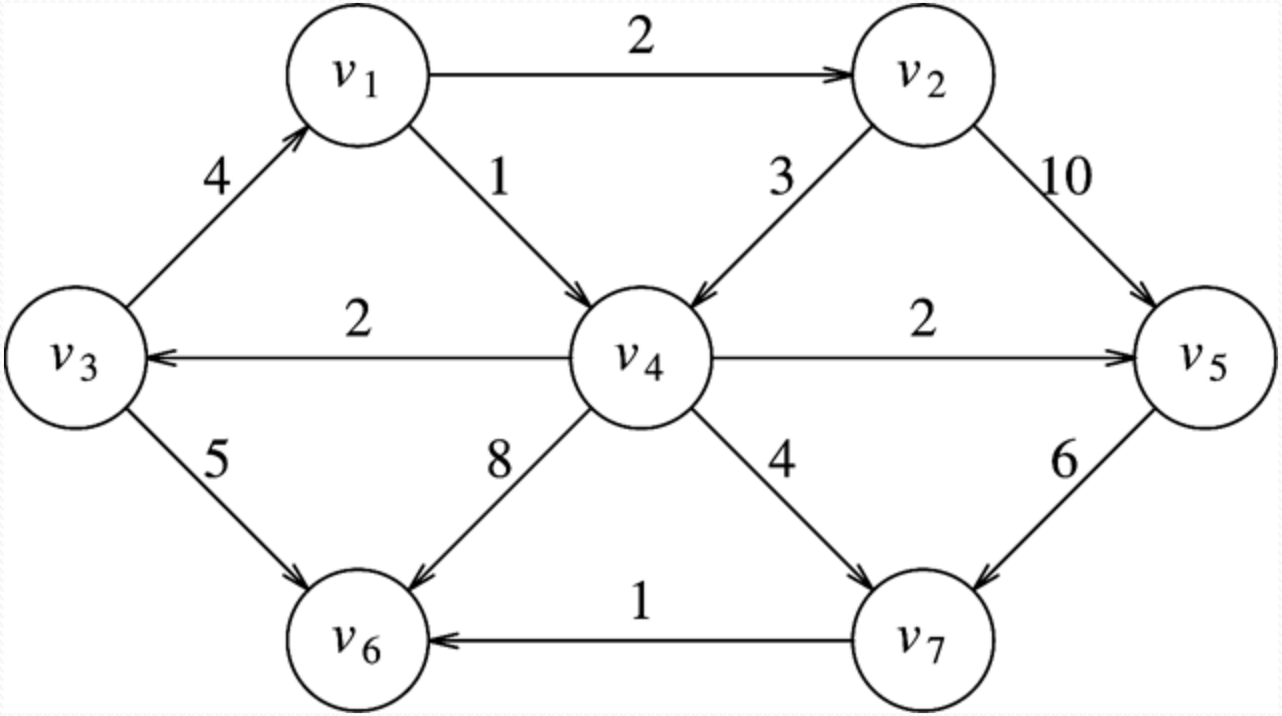
Weighted Shortest Path: **Dijkstra**

- Greedy algorithm
  - Greedy algorithms are not always optimal
  - **Dijkstra's** algorithm is optimal
  - Work on next vertex with smallest distance to *s*
- Dynamic programming
  - Reuse solutions to smaller subproblems
  - Shortest path from *s* to v_n goes through the shortest path from *s* to v_{n-1}.

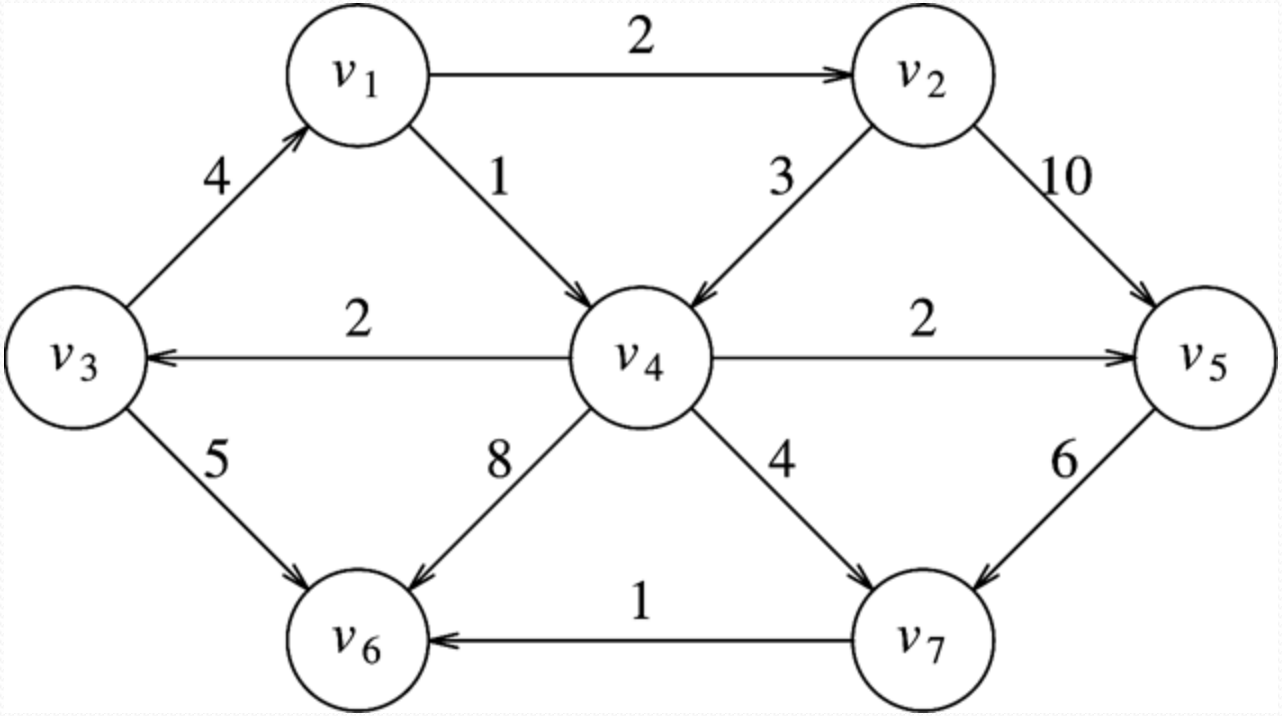
http://www.dgp.toronto.edu/~jstewart/270/9798s/Laffra/DijkstraApplet.html

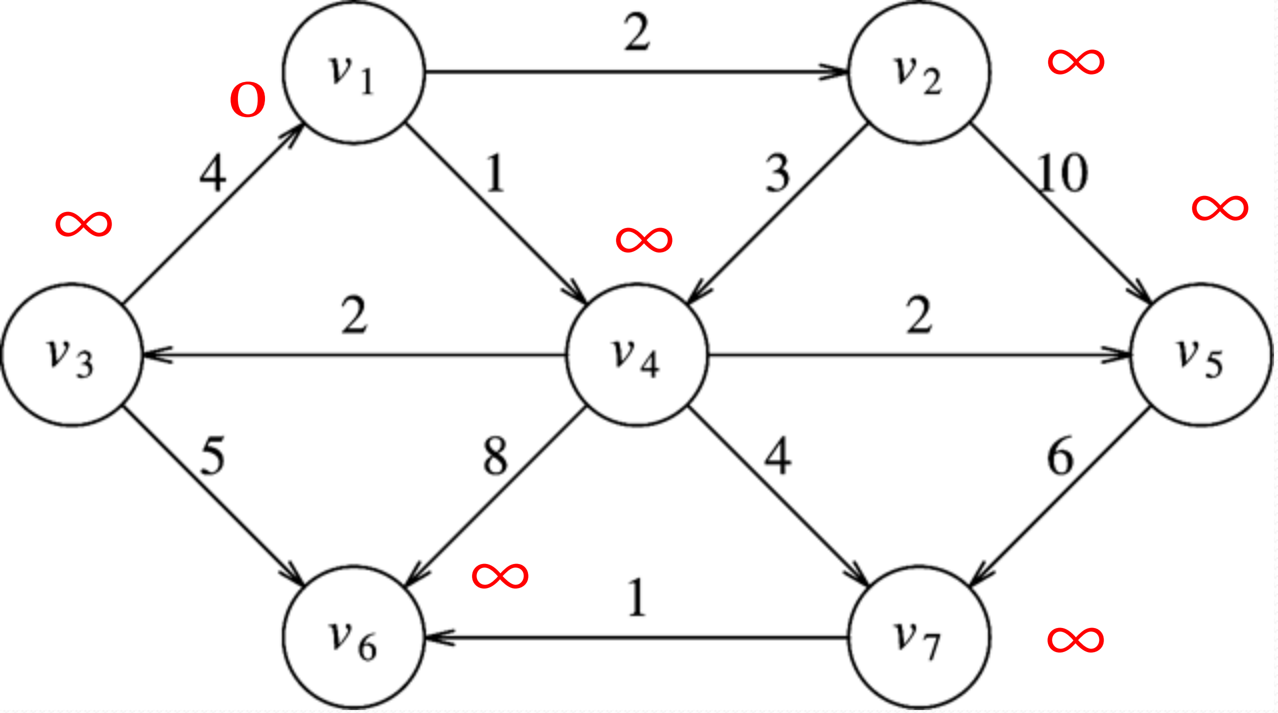
Dijkstra

Can we use breadth first search as in unweighted graph?

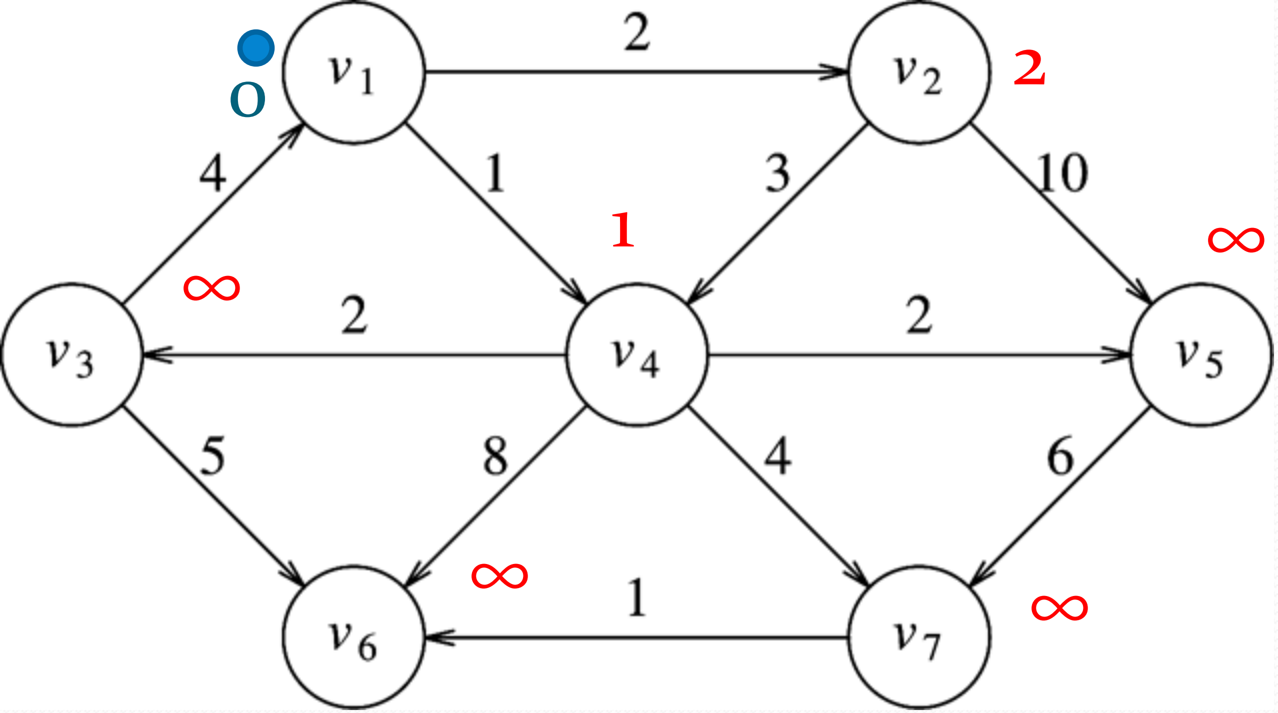


Dijkstra



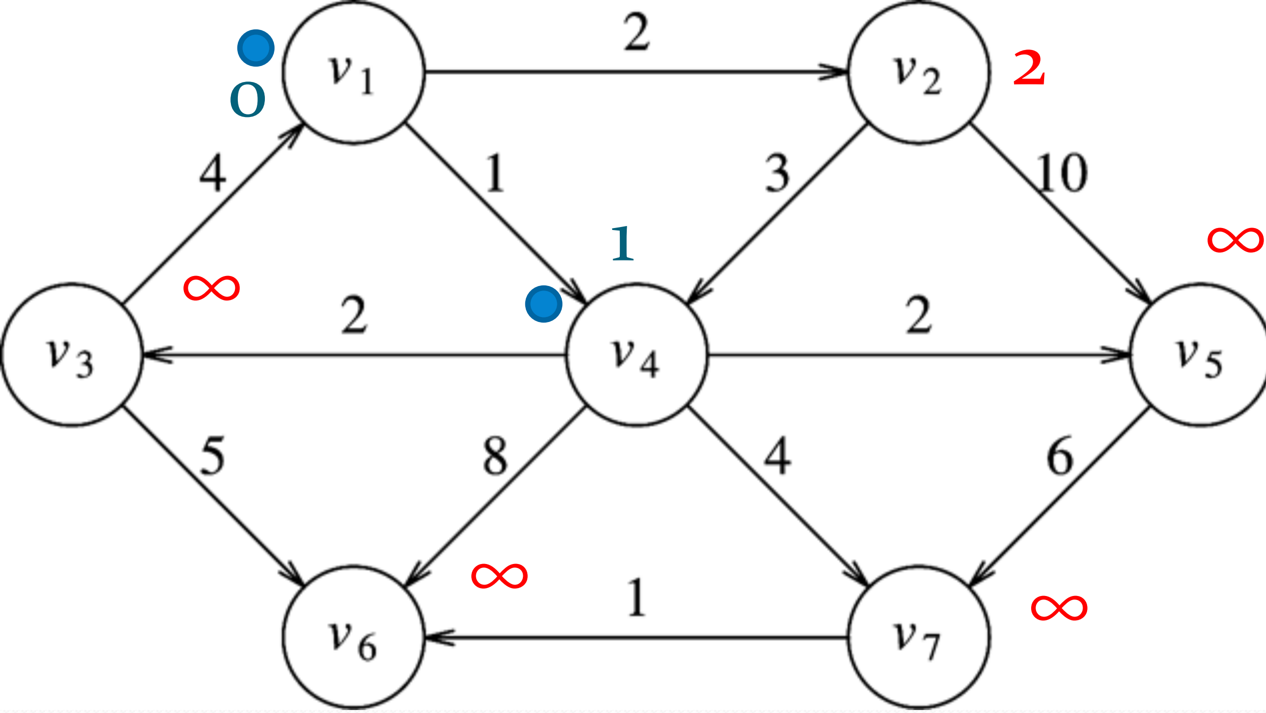


v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

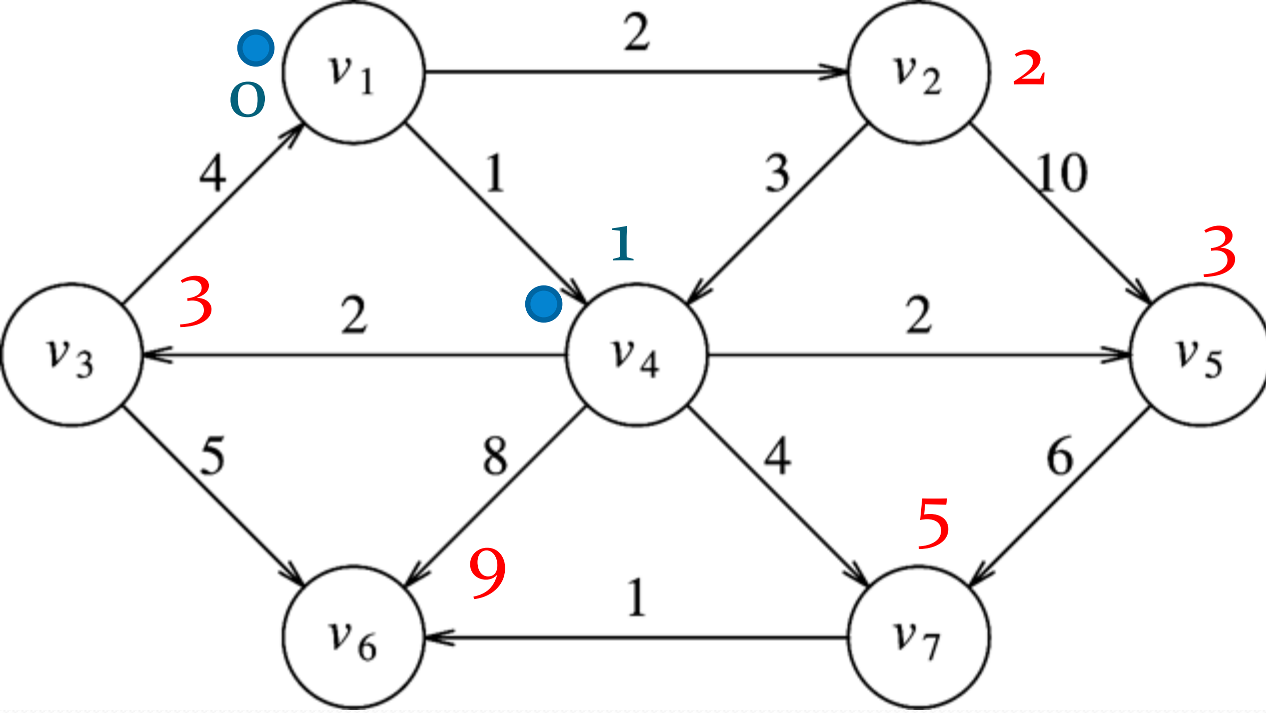


v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

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

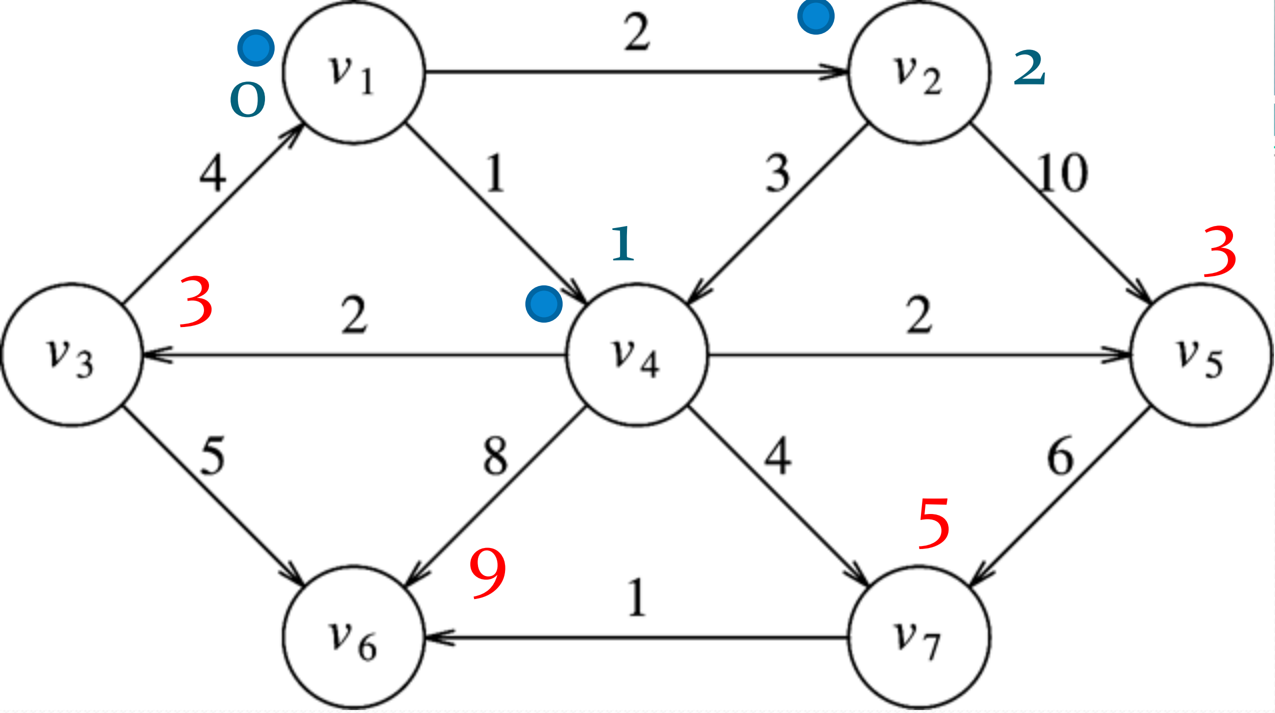


v		known	d_v	p_v
v_1	•	T	0	0
v_2		F	2	v_1
v_3		F	∞	0
v_4		F	1	v_1
v_5		F	∞	0
v_6		F	∞	0
v_7		F	∞	0

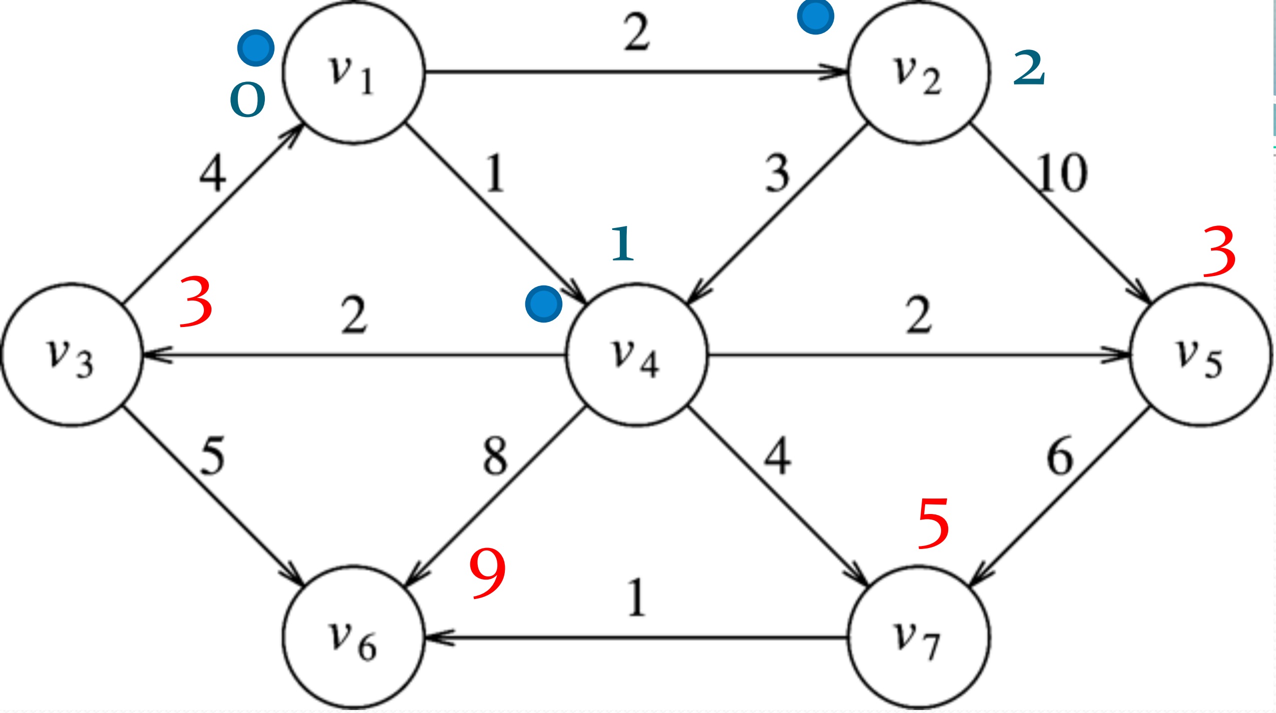


v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

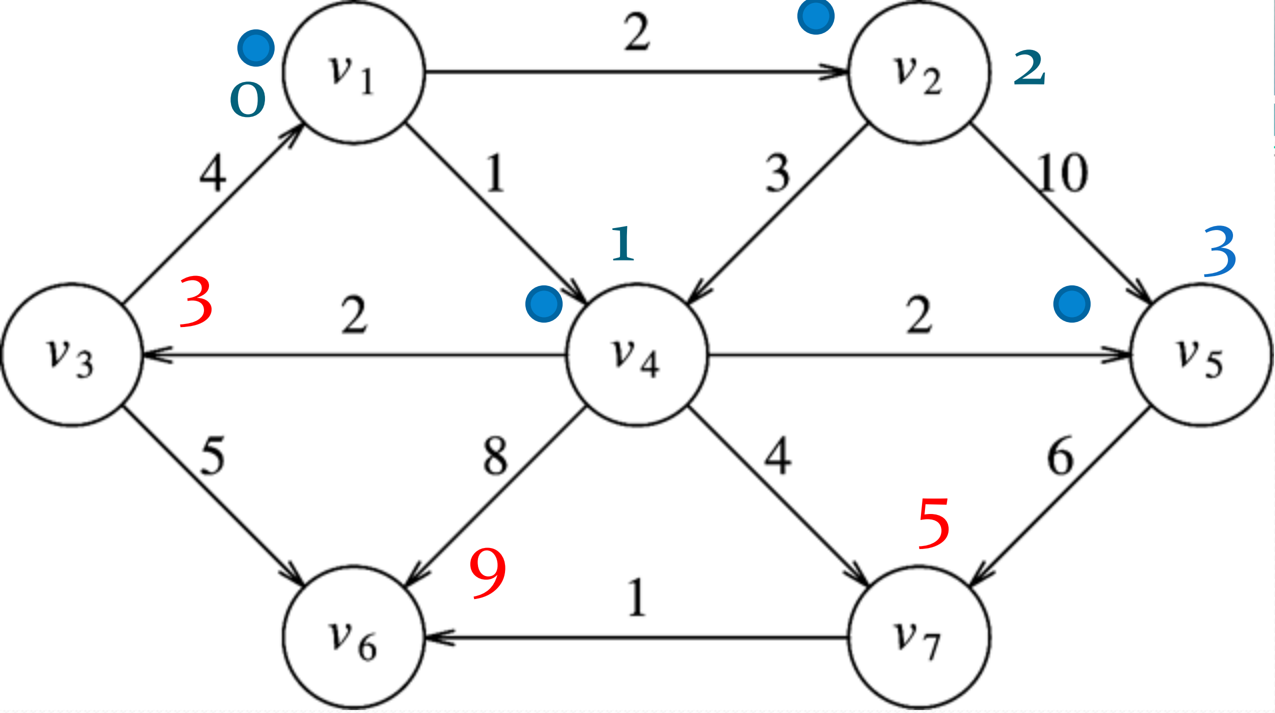


v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

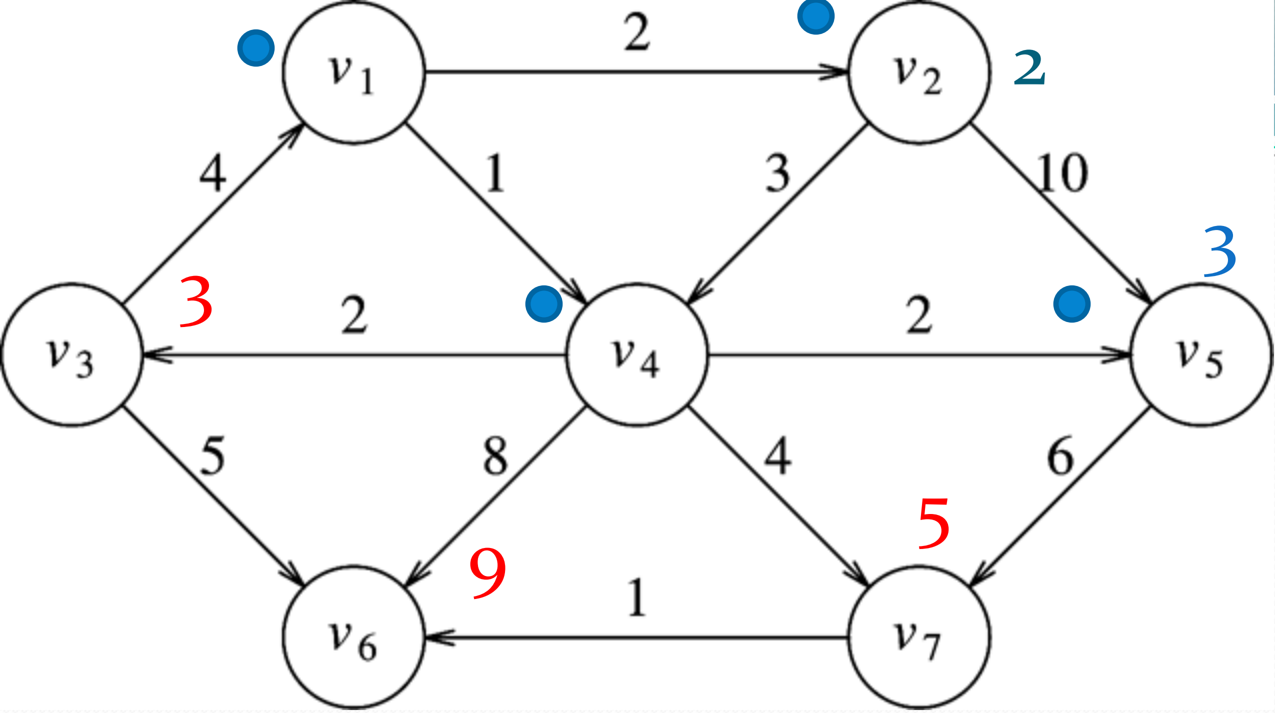


v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

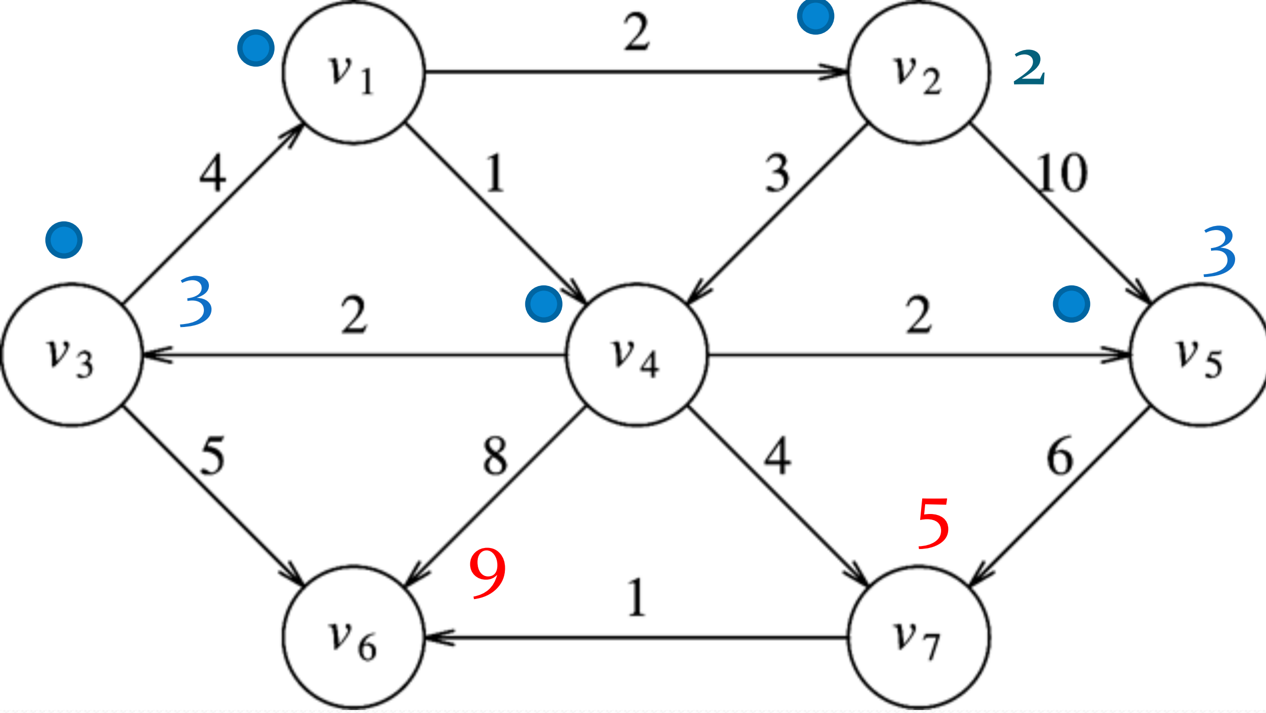


v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

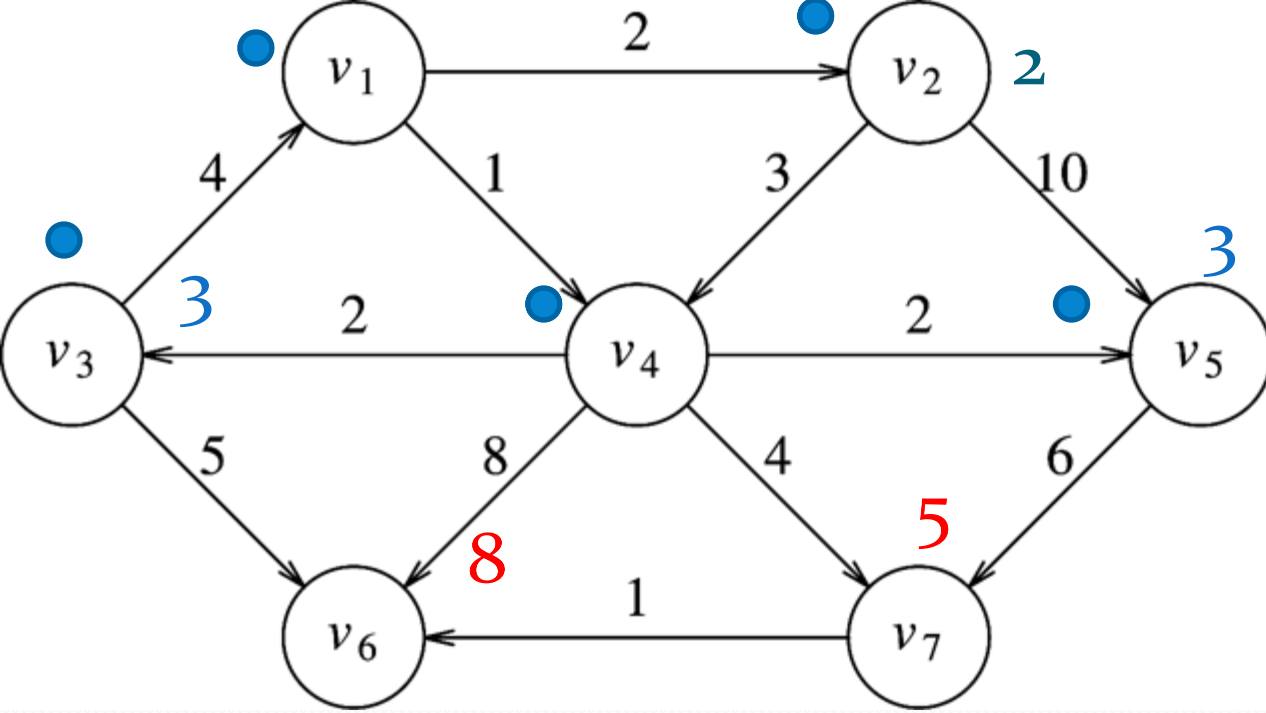


v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

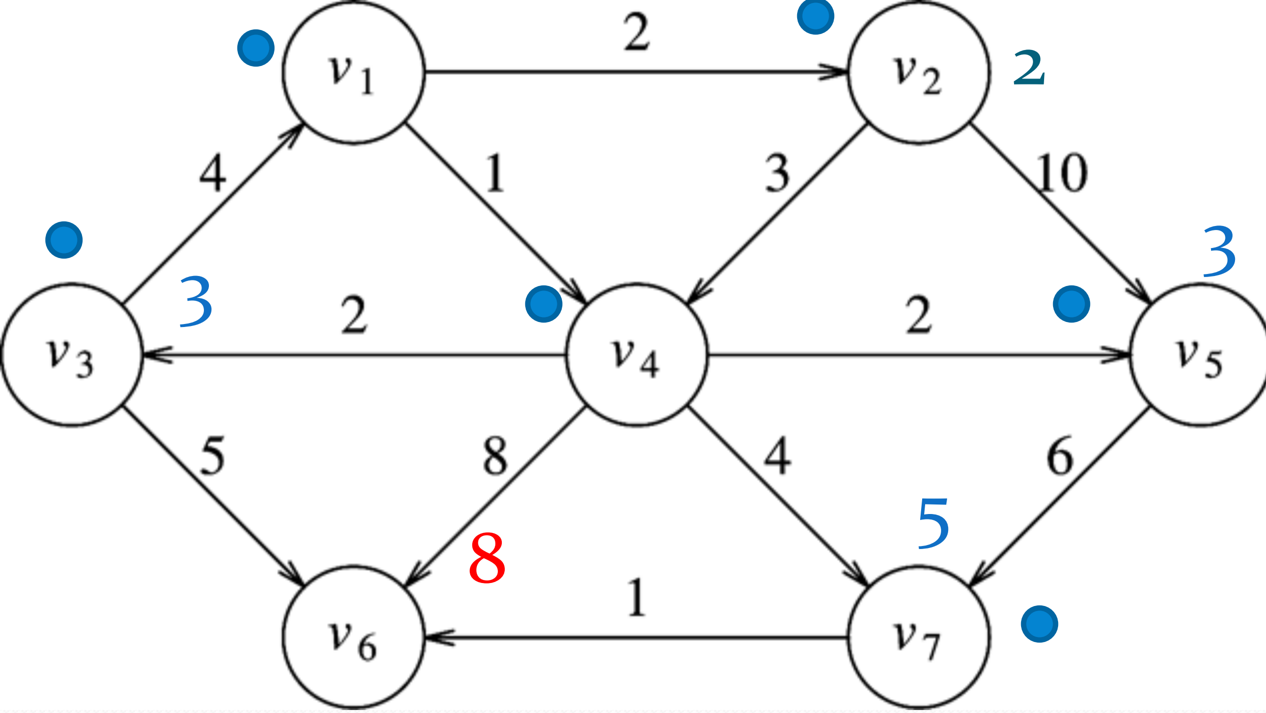


v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

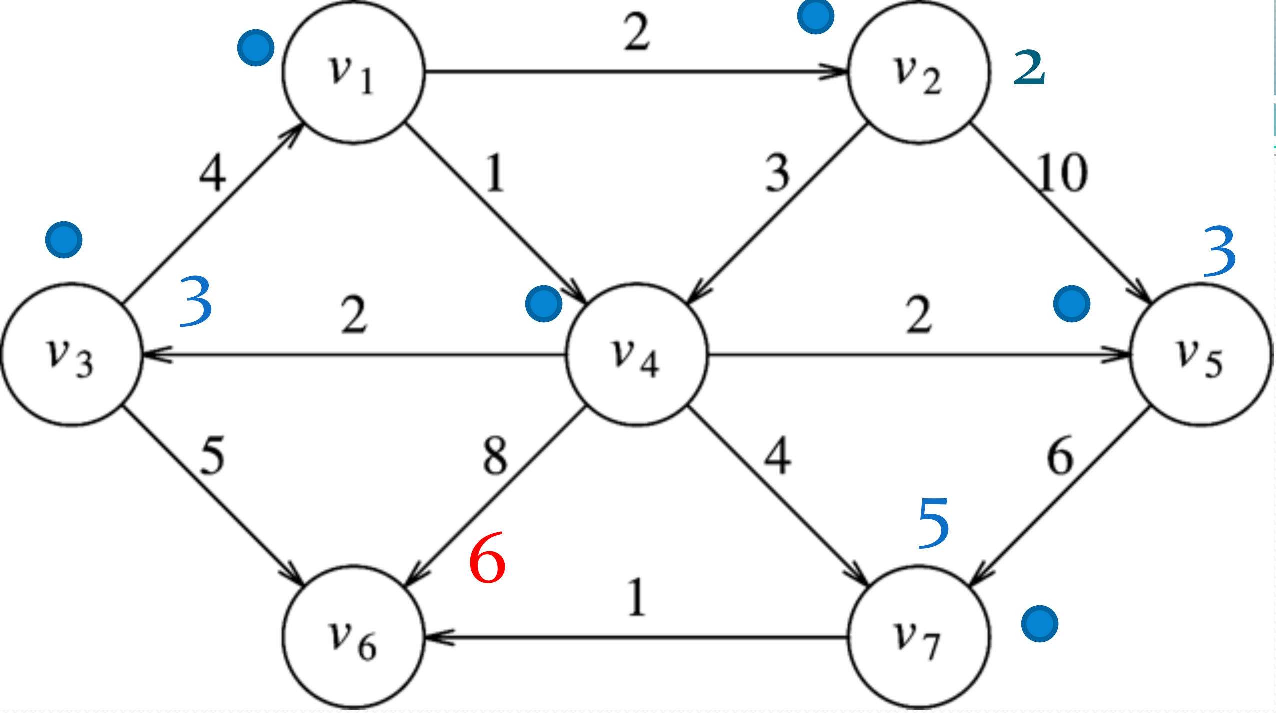


v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

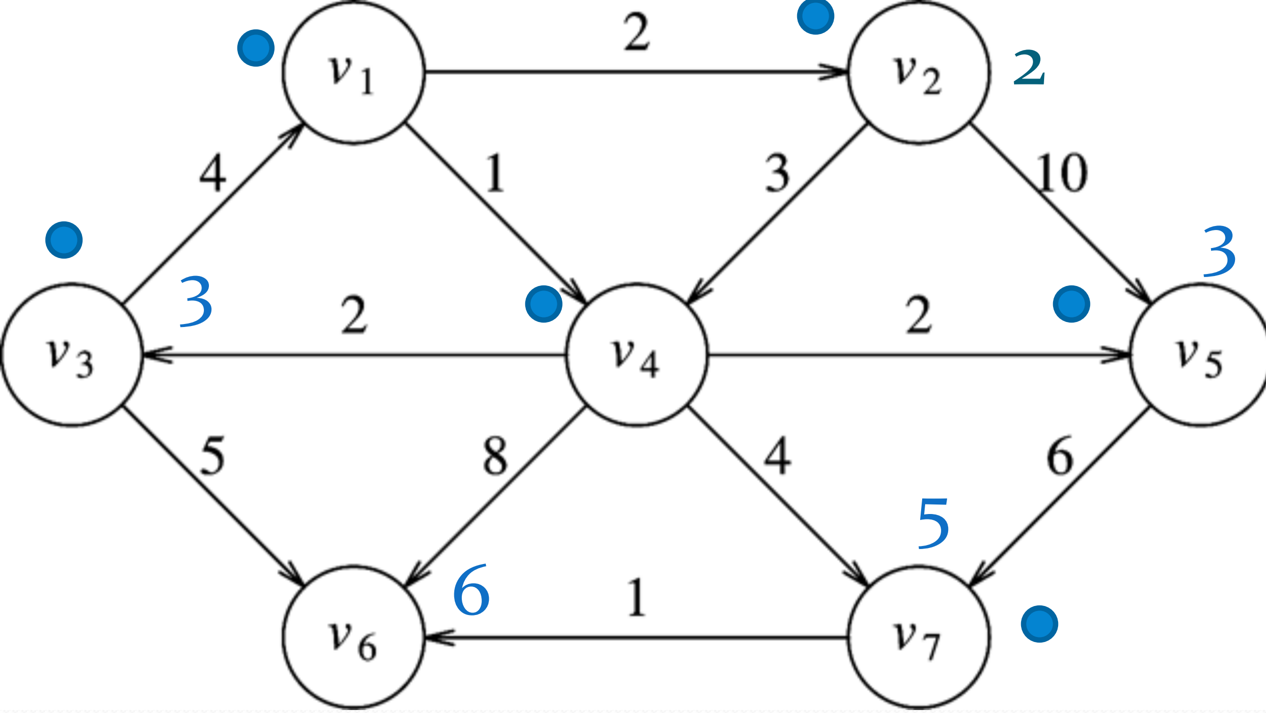


v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4



v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4



v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Implementation

```
/**
 * PSEUDOCODE sketch of the Vertex structure.
 * In real C++, path would be of type Vertex *,
 * and many of the code fragments that we describe
 * require either a dereference * or use the
 * -> operator instead of the . operator.
 * Needless to say, this obscure the basic algorithmic ideas
 */
struct Vertex {
    List<Vertex> adjacent_;    // Adjacency list
    bool         known_;
    DistType     distance_;   // DistType can be float, double, etc.
    Vertex       previous_in_path_; // Other data and member
                                    // functions as needed
};
```

```
void Graph::DijkstraShortestPath(Vertex s) {
    for each Vertex v {
        v.distance_ = kInfinity;
        v.known_ = false;
    }
    s.distance_ = 0;
    while (true) {
        Vertex v = smallest unknown distance vertex; // How?
        if (v == NOT_A_VERTEX) break;
        v.known_ = true;
        for each Vertex w adjacent to v {
            if (!w.known_) {
              const auto new_distance_through_v = v.distance_ + c(v,w);
              if (new_distance_through_v < w.distance_) {
                w.distance_ = new_distance_through_v; // Decrease w.distance_
                w.previous_in_path_ = v;
              }
            }
        }  // end of for each Vertex w…
    }  // end of while (true)
}
```

Analysis

- Cost in loop: $O(|V| * TimeToFindMinimum)$
- +Cost to update distance_ at each vertex = $O(|E|)$
- Total:

$O(|V| * TimeToFindMinimum + |E|)$

Find v of min distance + update w.distance_ for each (u,w) in E

- Simple approach:

TimeToFindMinimum=$O(|V|)$

$\Rightarrow$Total cost is

$\mathbf{O(|V|^2+|E|)=O((|V|^2)}$

If graph is dense: $|\mathbf{E}|=\mathbf{\Theta}(|\mathbf{V}|^2)$ => Cost is optimal

Sparse graphs $|E|=\Theta(|V|)$

- *Can we improve cost of TimeToFindMinimum?*

Implementation w/ Priority Queue
--using decreaseKey

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

    s.dist = 0;
    PriorityQueue P;   P.Insert(s); // Key is s.dist
    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
}
```

```
if ( P.IsEmpty()) break;
else
v = P.DeleteMin();
```

//UPDATE P.Q. P:

```
const auto new_dist = v.dist + cvw;
if (w.dist == ∞) {
    w.dist = new_dist;
    P.Insert(w);
} else {
    P.SetKey(find(w), new_dist );
}
w.path=v;
```

Sparse graphs $|E|=\Theta(|V|)$

- Use priority queue
  - Select closest node among unknowns => v=deleteMin()
  - Update distances of nodes w adjacent to v:

    p=find(w) [w is the node, p is its position in P.Q.]

    setKey(p, new_smallest_distance)

$O(|E|\log|V|+|V|\log|V|)=O(|E|\log|V|)$

Apply DecreaseKey() for w's

Find v via DeleteMin()

Sparse graphs [another approach]

- Use priority queue
  - Put into queue a node w whenever its distance changes
  - P.Q. contains known/uknown nodes
  - If v=DeleteMin() is known ignore => apply deleteMin() again.
  - Else: Make v known, update dw of w for all (v,w) edges.
- Difference with previous method?
  - Easier to code (no need for find() within the P.Q.)
  - Size of P.Q. can be as large as |E| (nodes can be inserted multiple times…)

Implementation w/ Priority Queue --without decrease key

```
oid Graph::dijkstra( Vertex s )

  for each Vertex v
  {
      v.dist = INFINITY;
      v.known = false;
  }

  s.dist = 0;
PriorityQueue P;   P.Insert(s); // Key is s.dist
  for( ; ; )
  {
      Vertex v = smallest unknown distance vertex;
      if( v == NOT_A_VERTEX )
          break;
      v.known = true;

      for each Vertex w adjacent to v
          if( !w.known )
              if( v.dist + cvw < w.dist )
              {
                  // Update w
                  decrease( w.dist to v.dist + cvw );
                  w.path = v;
              }
  }
```

```
bool success=false;
while (!P.isEmpty()  && !success) {
    v = P.deleteMin();
    if (!v.known) success = true;
}
if (!success) break;
```

//UPDATE P.Q. P:

```
w.dist = v.dist + cvw;
P.Insert(w);
w.path = v
```

Sparse graphs [another approach]

- $O(|E| \log|E| \quad + \quad |V| \log|E|\ ) \ = O(|E|\log|E|)$

  Insert dw's for w's: (v,w) in E *Find v via deleteMin()*

- Since $|E| \leq |V|^2 \Rightarrow \log|E| \leq 2\log\{V| \Rightarrow$ Cost is still

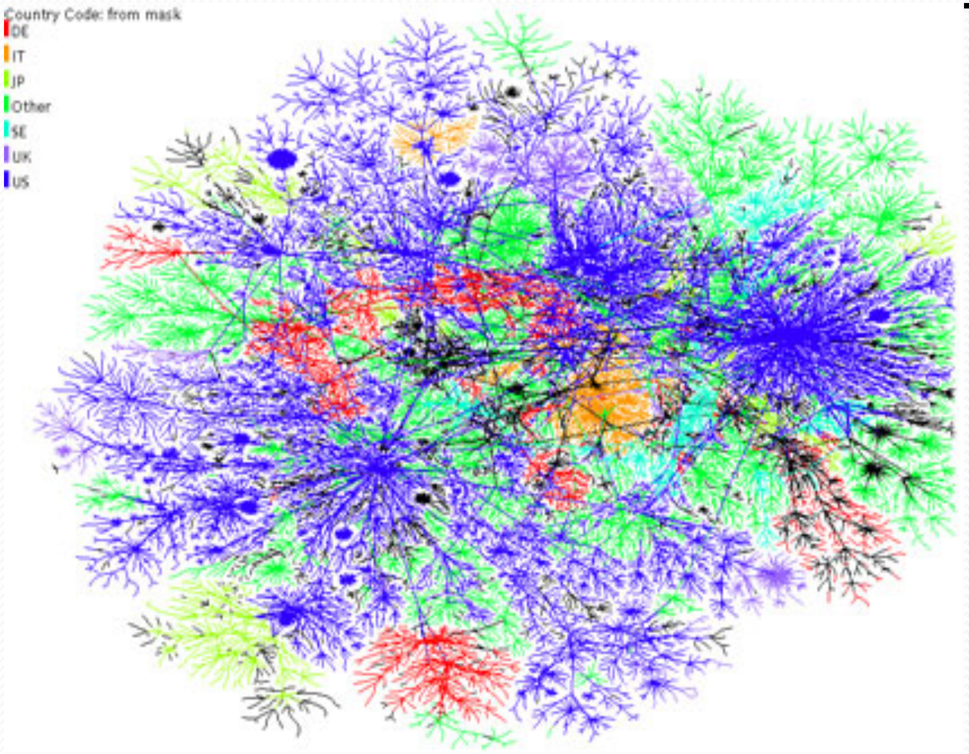
  $$\mathbf{O(|E|\log|V|)}$$

- In practice we would expect it to be slower than previous method though…

Sparse graphs

- Very typical

For instance internet connectivity



Degree of nodes ~ constant