



CS3345 Data Structures & Algorithm Analysis

Lecture Notes 24

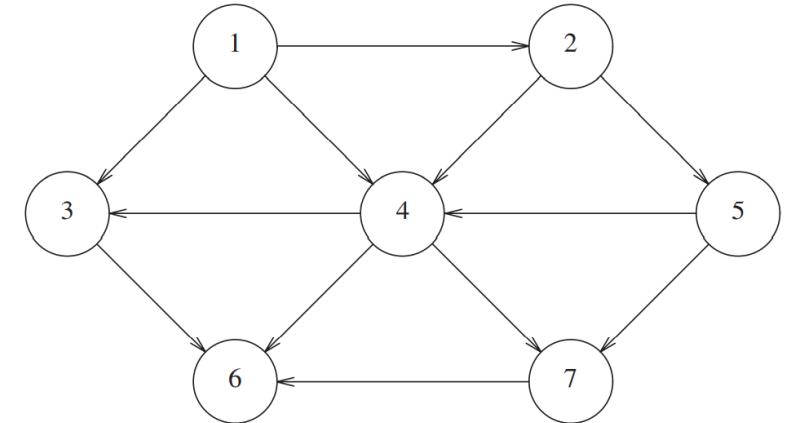
Omar Hamdy

Associate Professor

Department of Computer Science

Graph Representation

- Consider the following directed graph:
- One simple way to represent the graph is by using a 2D adjacency matrix.
- For each edge (u,v) , $A[u][v] = \text{true}$.
- For weighted edges, the weight is stored instead
 - For non-existent edges, $A[u][v]$ can be set to very large or very small depending on the application
- This representation is very simple, however needs $\theta(|V|^2)$ of space, which is only justifiable if the graph is **dense**: $|E| = \theta(|V|^2)$.
- If the graph is **sparse**, adjacency list is a better alternative of size $\theta(|V| + |E|)$.

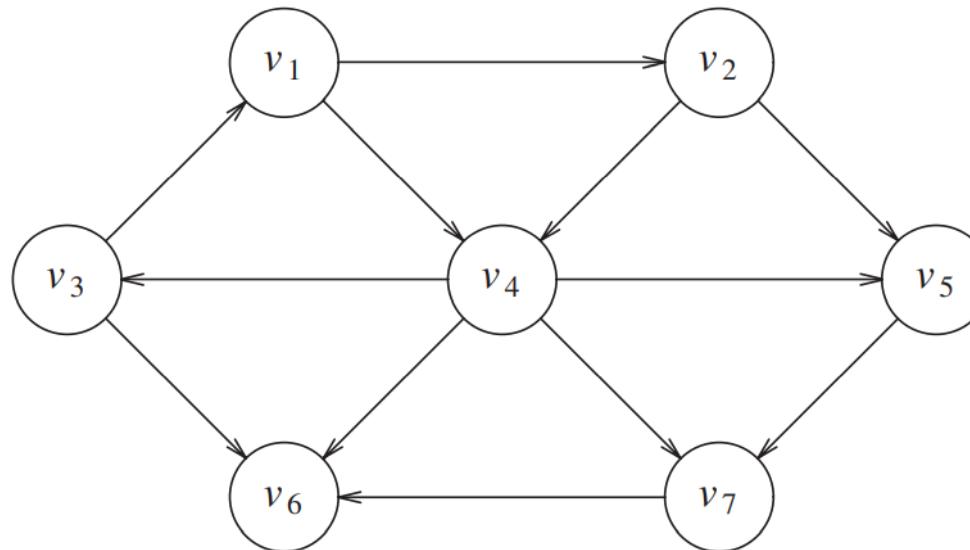


Shortest Path Algorithms

- Shortest path is one key optimization problem that applies to many real-life situations.
- For example, we can model airplane or other mass transit routes by graphs and use a shortest path algorithm to compute the best route between two points
- Typically, these problems require finding the shortest path from a specific vertex s , to another vertex t .
- To do so, the current algorithms find the shortest path from this vertex s to ALL other vertices in the graph.
- This approach makes the algorithm simpler, and without notable performance difference.

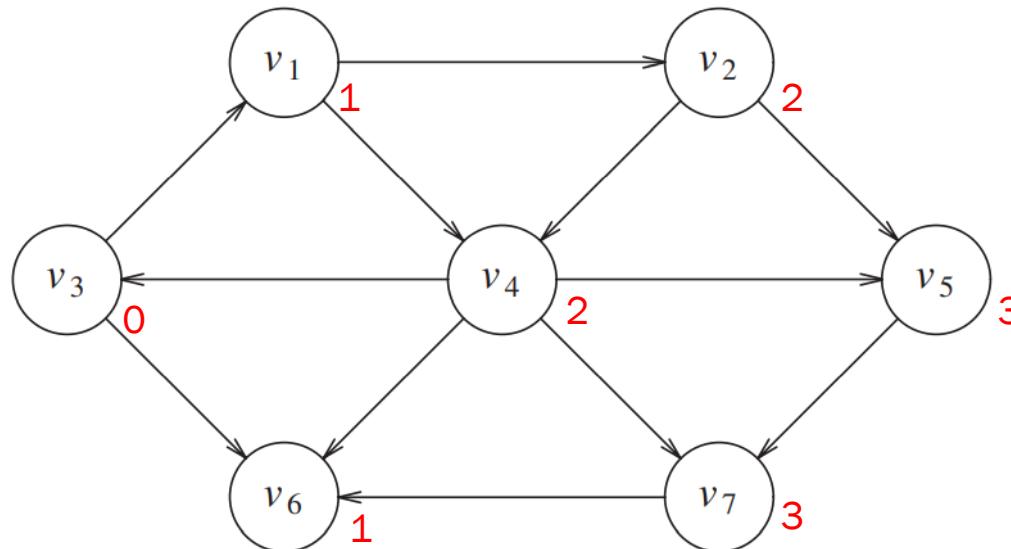
Breadth First Search Algorithm

- We shall study this algorithm using unweighted graphs. This is a special case of weighted graphs where each edge has an equal weight of 1.
- The algorithm assigns a number which represents the path length (number of edges)
- Let us study the algorithm by example. Assume s is chosen to be v₃ in the below graph



Breadth First Search Algorithm

- First, v_3 distance is 0, which we indicate on the graph.
- Then we look for all vertices that are at distance 1 away from s
- Then we look for all vertices that are at distance 2 away from s by looking at all the adjacent vertices to v_1 and v_6
- Following the same approach, we get the following distances



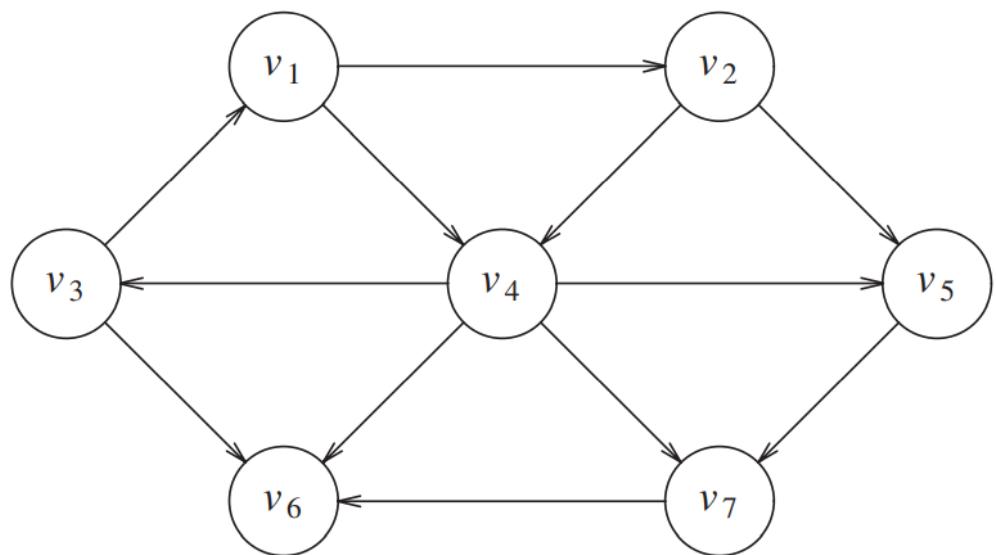
Breadth First Search Algorithm

- The algorithm is named Breadth First, because it starts by processing all the ones closest to the starting point, before proceeding to the next layer away.
- To build the code for it, we first start by defining a distance configuration table and initialize it as follows:

- d_v represents the distance between s and each vertex, initially all set to infinity
- Known column is a true/false attribute to mark all the vertices that got already processed.
- When a vertex is marked known, it is guaranteed that no cheaper path will ever be found for that vertex.
- P_v lists the path through which this vertex is reached.

v	<i>known</i>	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

Breadth First Search Simple Algorithm



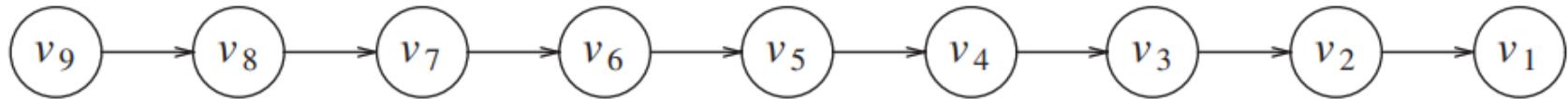
```
void unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Breadth First Search Algorithm Analysis

- The running time of this algorithm is $O(|V|^2)$.
- The inefficiency is partially because the outer loop continues until `NUM_VERTICES - 1`, even if all vertices become known earlier.
- Fixing this will change average case, but not worst case because we could have a graph like this:

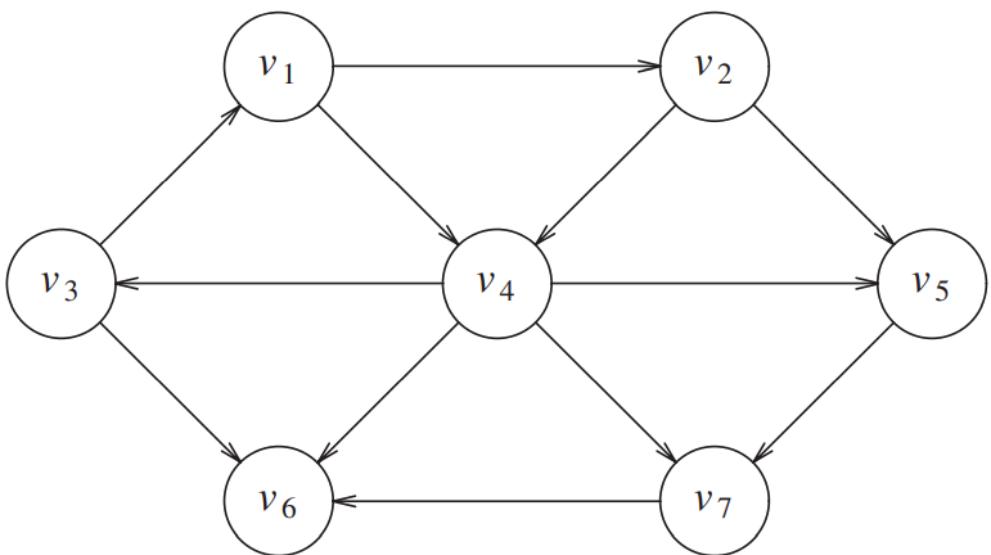


- A better approach is to limit each iteration to process only the adjacent vertices to the current one. Hence, achieving $O(|V| + |E|)$
- The following queue algorithm provides an implementation to this idea

Breadth First Search Algorithm

```
void unweighted( Vertex s )
{
    Queue<Vertex> q = new Queue<Vertex>( );
    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 );
            }
    }
}
```

Breadth First Search Algorithm



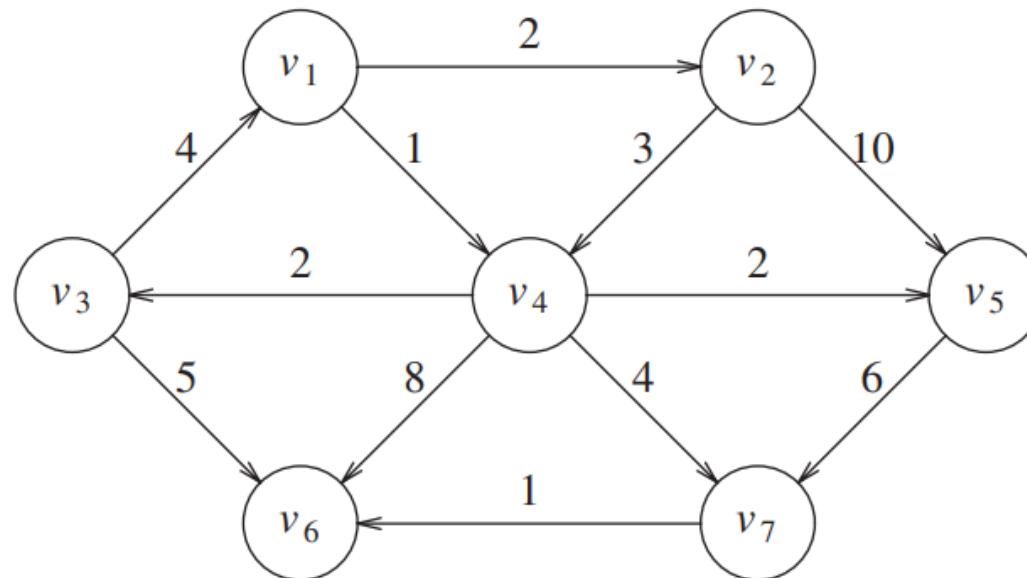
Initial State				v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	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	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		

Dijkstra's algorithm

- We can use the main idea from breadth first search algorithm to handle the more complex weighted graph problem.
- This is done using Dijkstra's algorithm, which is classified under what is known as greedy algorithms.
- Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage.
- Example, assume a cashier is returning 68 cents of change to a customer. How do you expect the order will look like?
 - 2 quarters, 1 dime, 1 nickel and 3 cents
 - However, this approach does not work all the time. Imagine there is a 12-cent coin, and you want to return 15-cent change.
 - According to the greedy algorithm, you will return 12-cent coin and 3 cents, which is worse than returning a dime and a nickel.

Dijkstra's algorithm

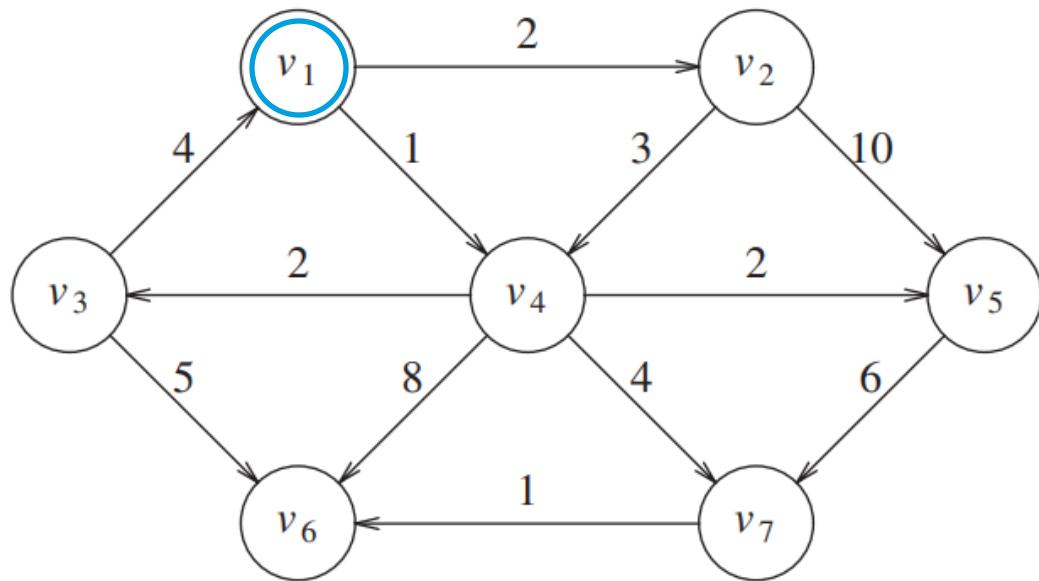
- At each stage, Dijkstra's algorithm selects a vertex v , which has the smallest d_v among all the unknown vertices, then declares that the shortest path from s to v is known.
- The remainder of any stage consists of updating the values of the adjacent vertices d_w
 - $d_w = d_v + c_{v,w}$ if this new value for d_w would be an improvement (known as relaxation).
- We will study the algorithm using an example, with starting point 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

Dijkstra's algorithm

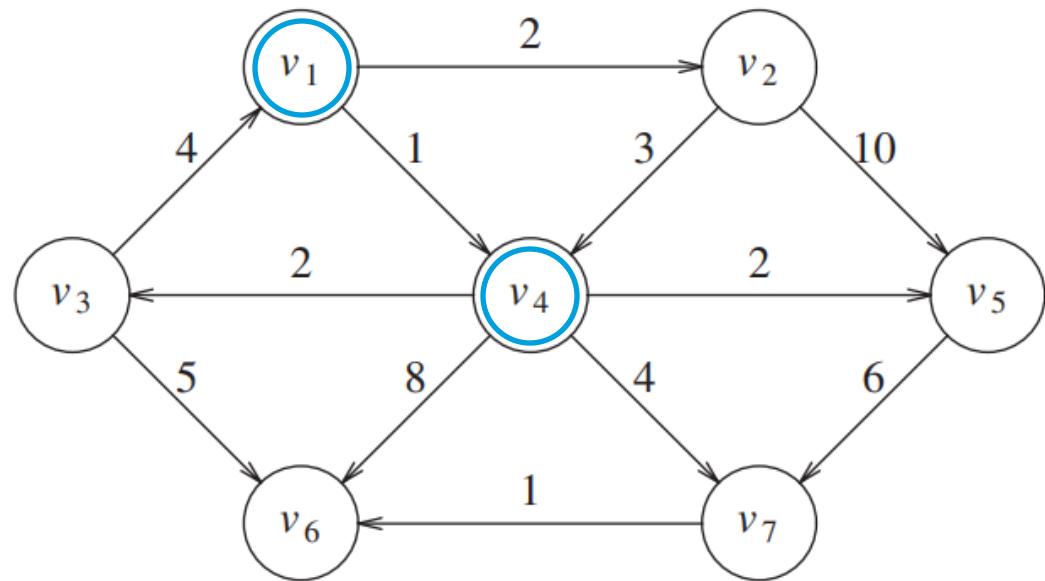
- The graph after v_1 is declared as known, and its adjacent vertices v_2 and v_4 are updated
- What is the next vertex from there?



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

Dijkstra's algorithm

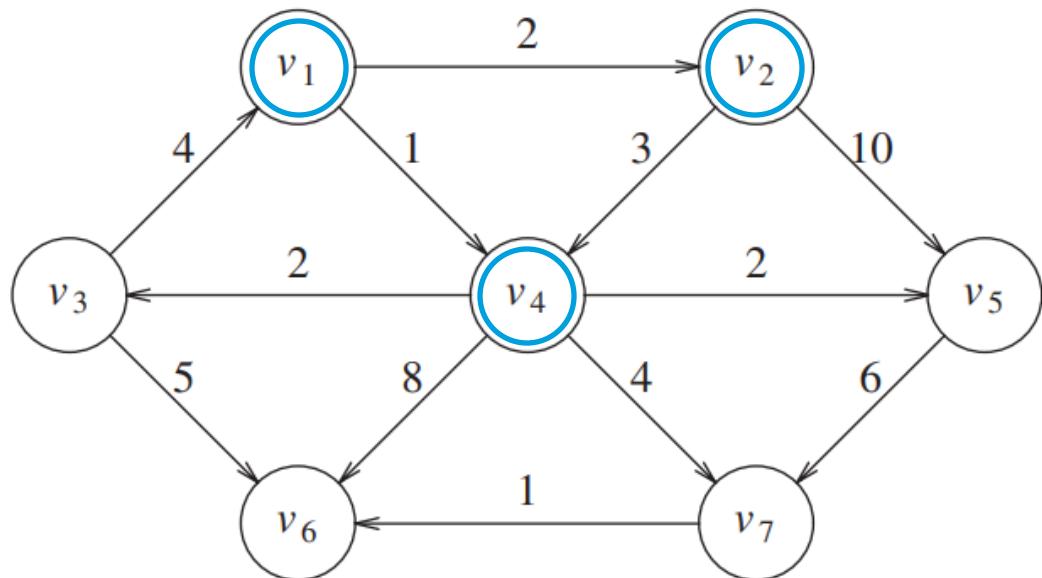
- Next, vertex 4 is selected because it has the smallest distance among all the unknown vertexes.
- The graph after v4 is declared as known, and its adjacent vertices v3, v5, v6, v7 are updated



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

Dijkstra's algorithm

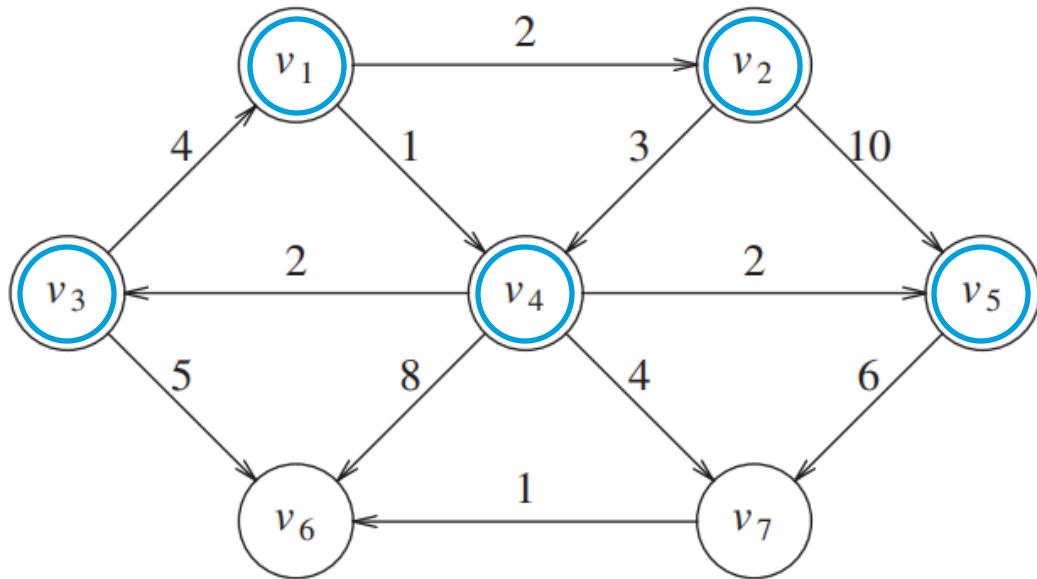
- Next, vertex 2 is selected because it has the smallest distance among all the unknown vertexes.
- Note vertex v5 is not updated because it has a lesser value than what v2 is offering.



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

Dijkstra's algorithm

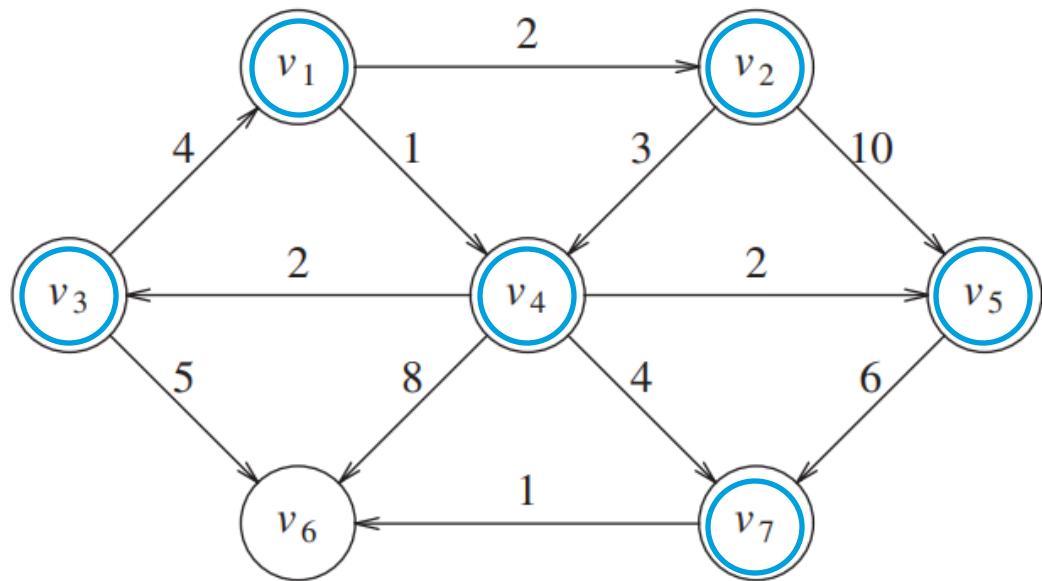
- Next, vertices 3 and 5 are selected, and only v6 is updated (why?)
- What is next?



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

Dijkstra's algorithm

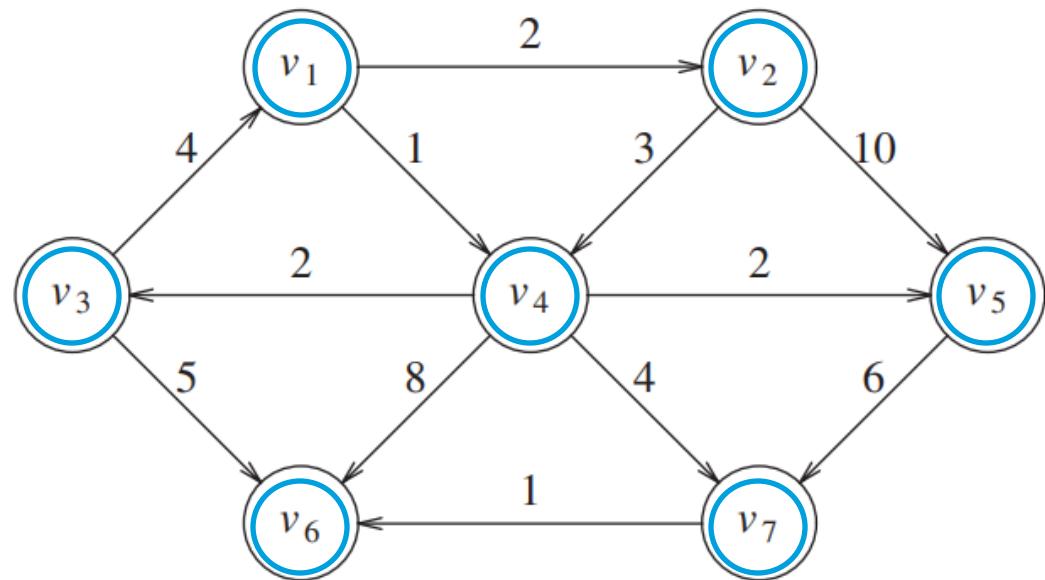
- Next, vertex 7 is selected, and v6 is updated (why?)



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

Dijkstra's algorithm

- Next, vertex 6 is declared known and the program terminates



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

Dijkstra's algorithm

```
class Vertex
{
    public List      adj;      // Adjacency list
    public boolean   known;
    public DistType dist;     // DistType is probably int
    public Vertex   path;
                    // Other fields and methods as needed
}
```

Dijkstra's algorithm

```
void 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;
                }
            }
    }
}
```

Dijkstra's algorithm

```
void printPath( Vertex v )
{
    if( v.path != null )
    {
        printPath( v.path );
        System.out.print( " to " );
    }
    System.out.print( v );
}
```

Dijkstra's algorithm Complexity

- At each phase, the algorithm scans for all the vertices to find the minimum d_v . Hence, each phase is taking $O(|V|)$. Therefore, all stages are $O(|V^2|)$.
- The updates are all constant and hence, the overall update will take $O(|E|)$.
- Therefore, the algorithm performance is $O(|V^2| + |E|) = O(|V^2|)$
- There are advanced data structures that can lower the performance to $O(|E| + |V|\log|V|)$

To Do List

- Review lecture notes
- Study sections 9.1 through 9.3