Data Structures and Algorithms

- Graph 1 -

School of Electrical Engineering
Korea University

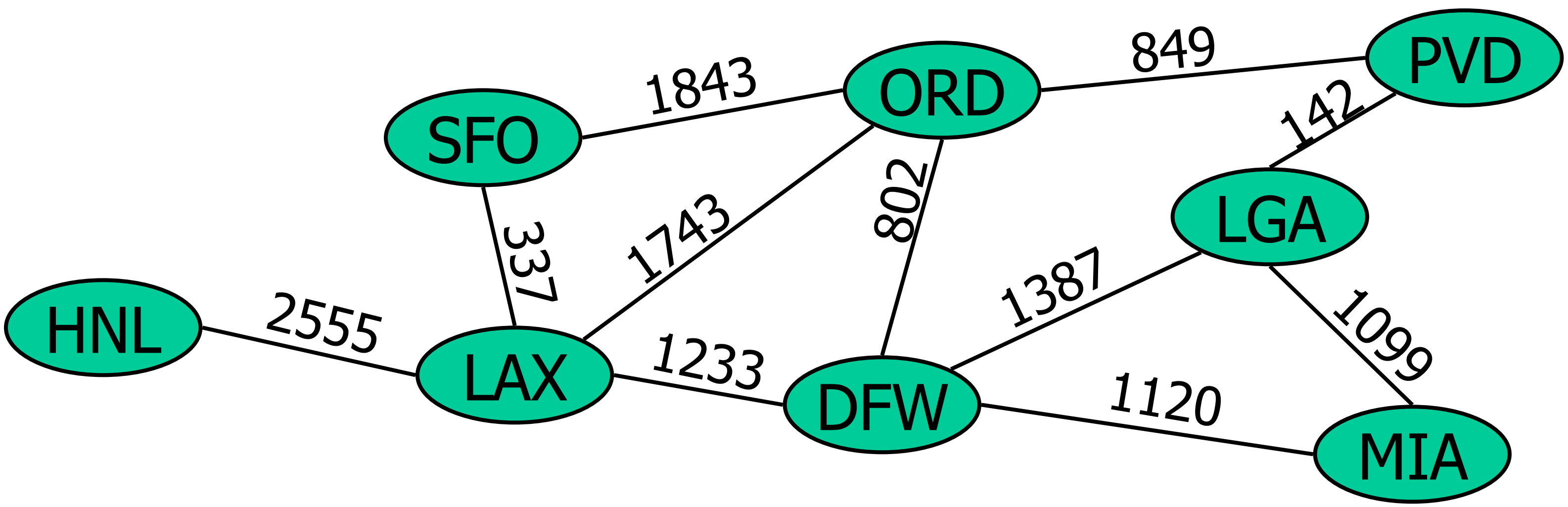
Graph

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges

- Each edge is a pair ($\mathbf{v}$, $\mathbf{w}$), where $\mathbf{v}$, $\mathbf{w} \in V$

Graph - Example

- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route



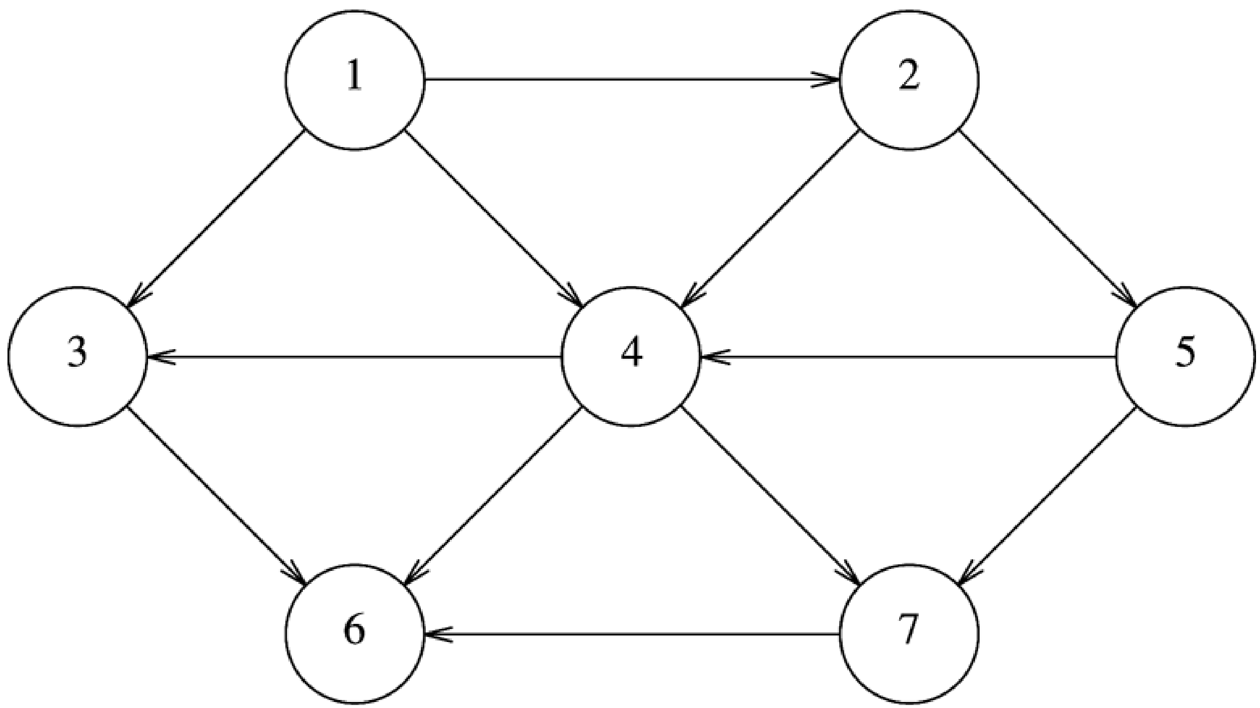
Graphs-Terminology

- The vertex pair is ***ordered*** (***unordered***), then the graph is ***directed*** (***undirected***)
- Vertex w is ***adjacent*** to v if and only if $(v, w) \in E$
- An edge could have a ***weight*** or a ***cost***
- A ***path*** (simple path) is a sequence of vertices $w_1, w_2, \cdots, w_N$ s.t. $(w_i, w_{i+1}) \in E$ for all $1 \leq i < N$
- ***Path length*** is the number of edges
- ***Cycle*** in a directed graph

Edge Types

- Directed edge
  - ordered pair of vertices **(*v*,*w*)**
  - first vertex ***v*** is the *origin*
  - second vertex ***w*** is the *destination*
  - e.g., a flight
- Directed graph (=digraph)
  - all the edges are directed as in route network

Directed graph

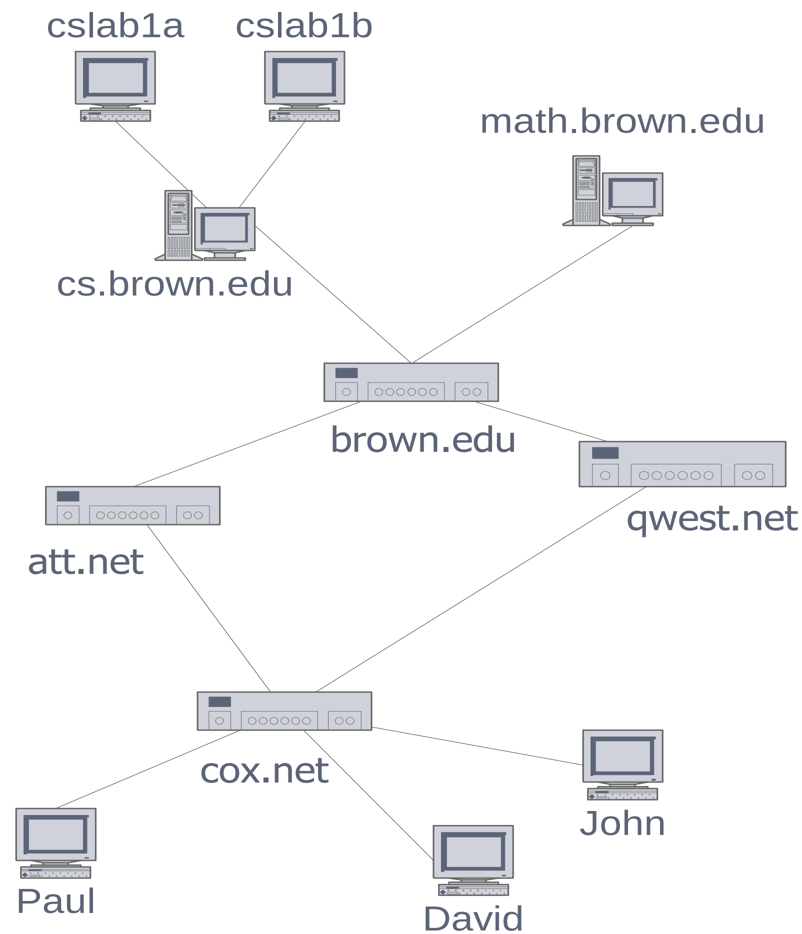


Edge Types

- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram

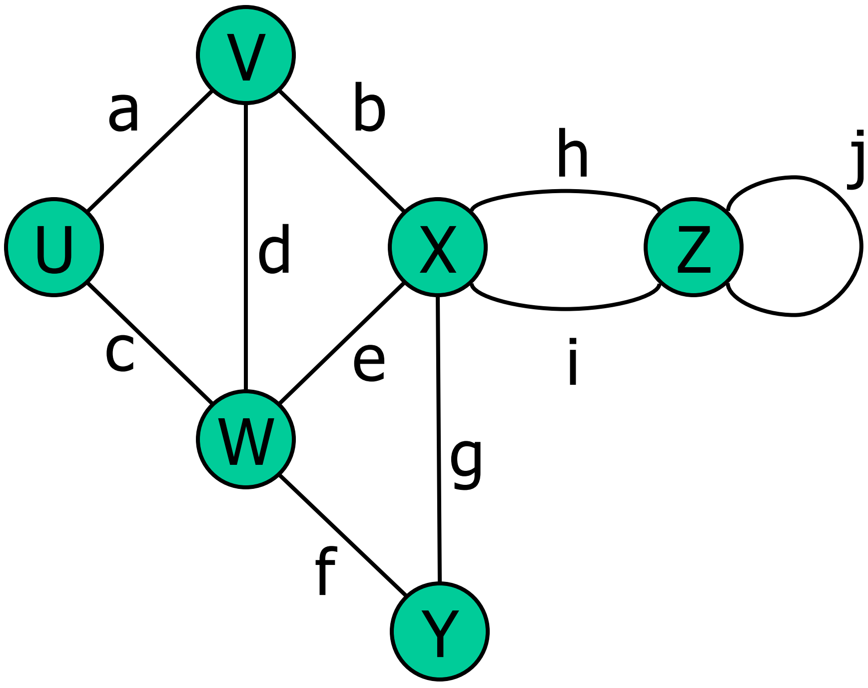


Terminology(cont.)

- An undirected graph is ***connected*** if there is a path from every vertex to every other vertex
- A directed graph with this property is ***strongly connected***
- If not strongly connected, but connected, then the graph is ***weakly connected***
- A ***complete*** graph is a graph in which there is an edge between every pair of vertices

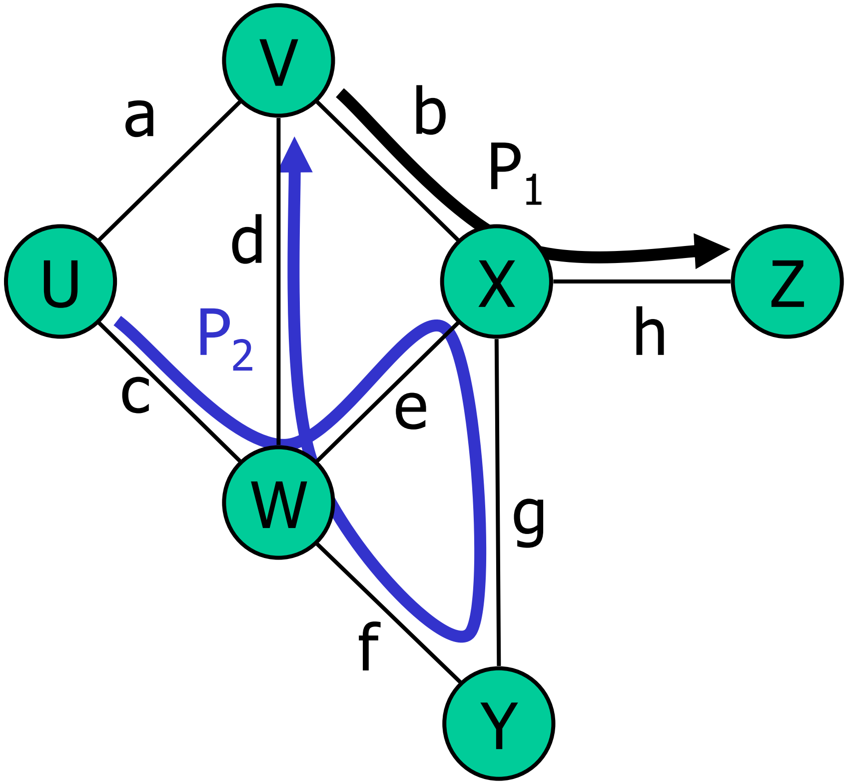
Terminology(cont.)

- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - Total no.of edges
  - X has degree 5



Example

- P_1=(V,b,X,h,Z) is a simple path
- P_2=(U,W,X,Y,W,V) is not simple



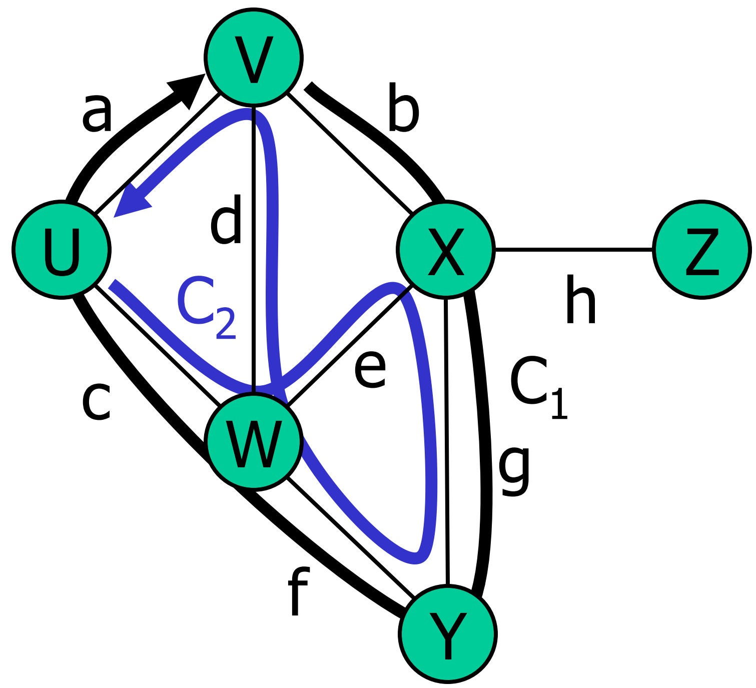
Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoint

- Simple cycle
  - cycle such that all its vertices and edges are distinct

Example

- C_1=(V,X,Y,W,U,V) is a simple cycle
- C_2=(U,W,X,Y,W,V,U) is a cycle that is not simple



Properties

Property 1

$\sum_n \deg(n) = 2e$

Proof: each endpoint is counted twice

Property 2

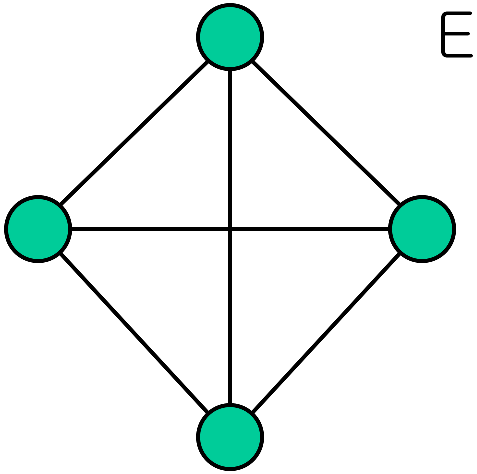
In an undirected graph with no self-loops and no multiple edges

$e \le v(v-1)/2$

Proof: each vertex has degree at most $(n-1)$

Notation

v	number of vertices
e	number of edges
$\deg(n)$	degree of vertex n



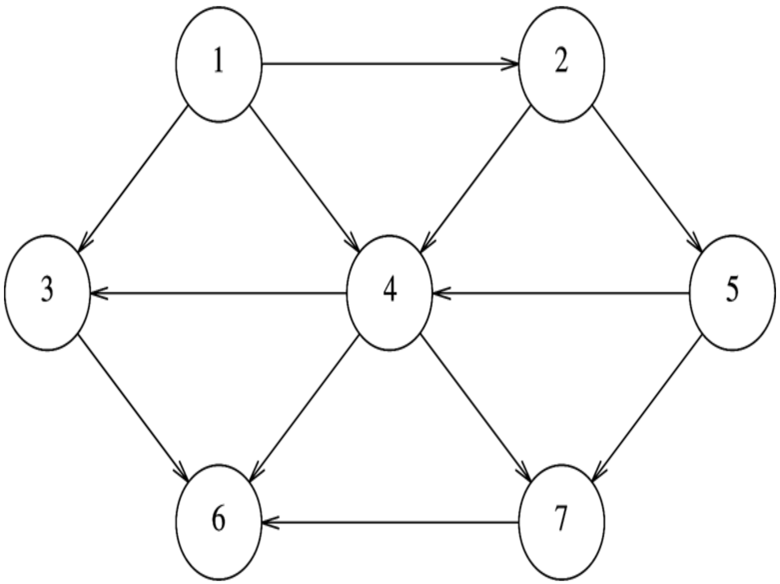
Example

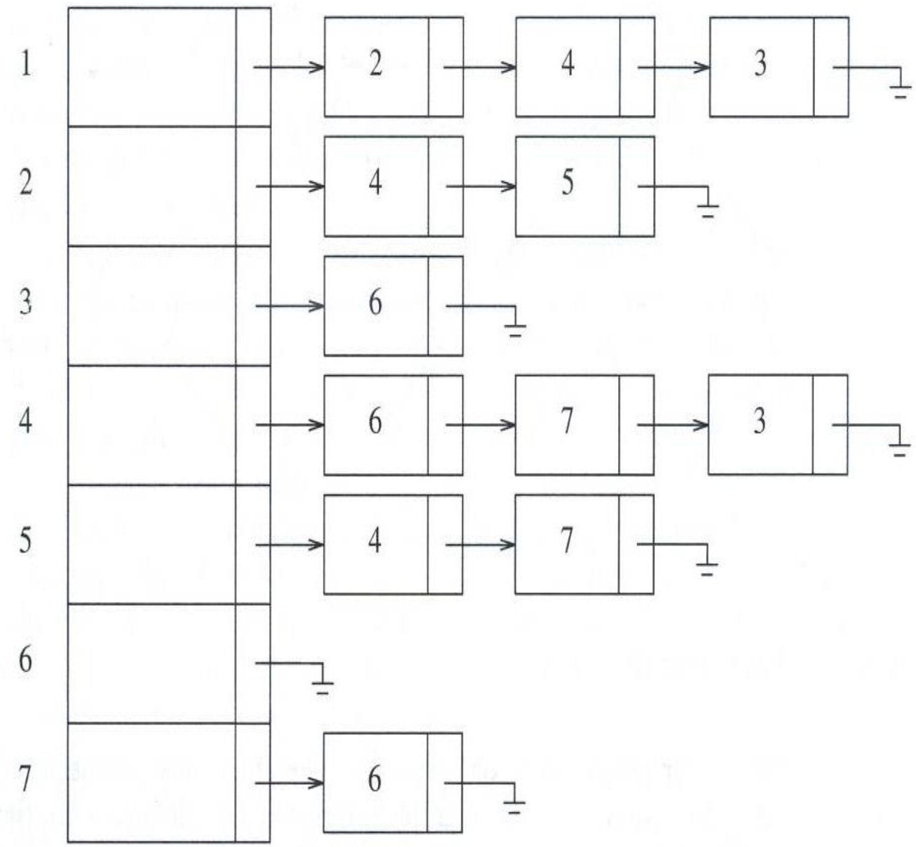
- $v = 4$
- $e = 6$
- $\deg(n) = 3$

Graph representation

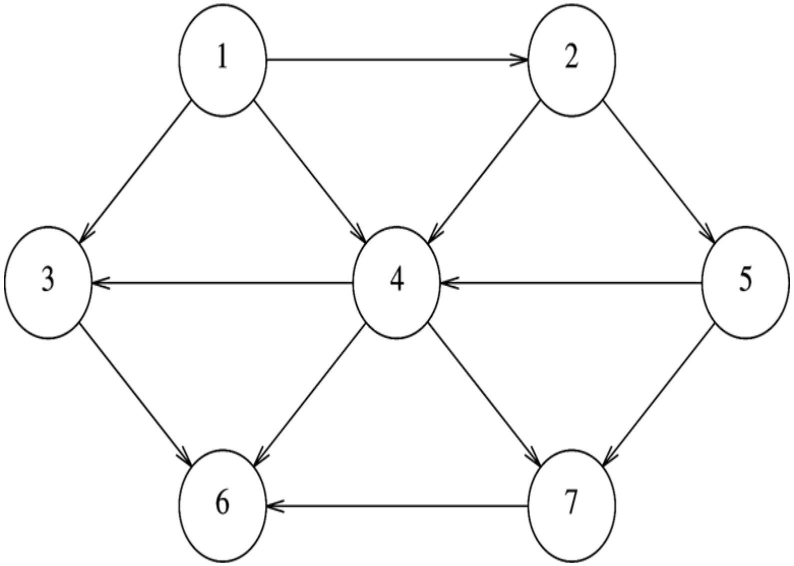
- Adjacency matrix representation
  - ✓ Simple
  - ✓ Appropriate when the graph is dense
  - ✓ $|E| = \Theta(|V|^2)$
- Adjacency list representation
  - ✓ When the graph is sparse
  - ✓ Space requirement is $O(|E| + |V|)$

Graph representation





Graph representation



	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3							
4							
5							
6							
7							

Adjacency matrix

Topological Sort

- An ordering of vertices in a directed acyclic graph such that
  - if there exists a path from v_i to v_j, v_i appears before v_j in the ordering
  - For a graph with cycle → no topological ordering

Prerequisite graph

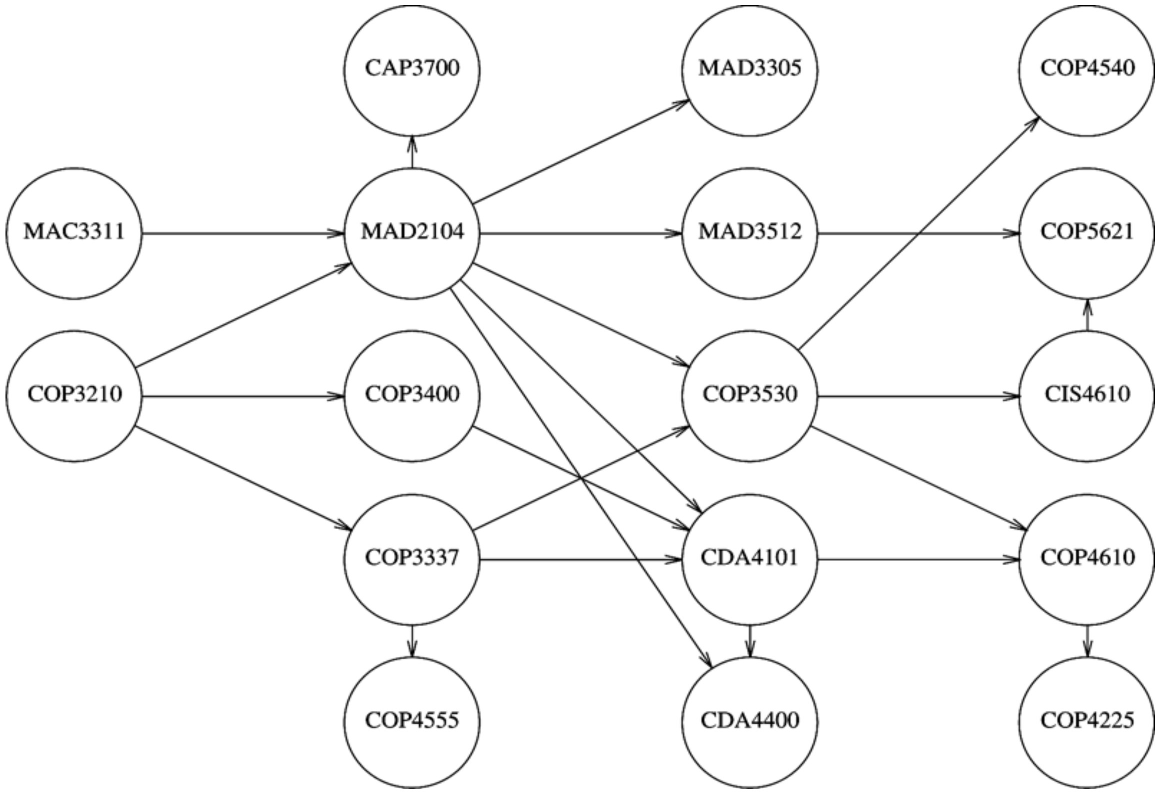
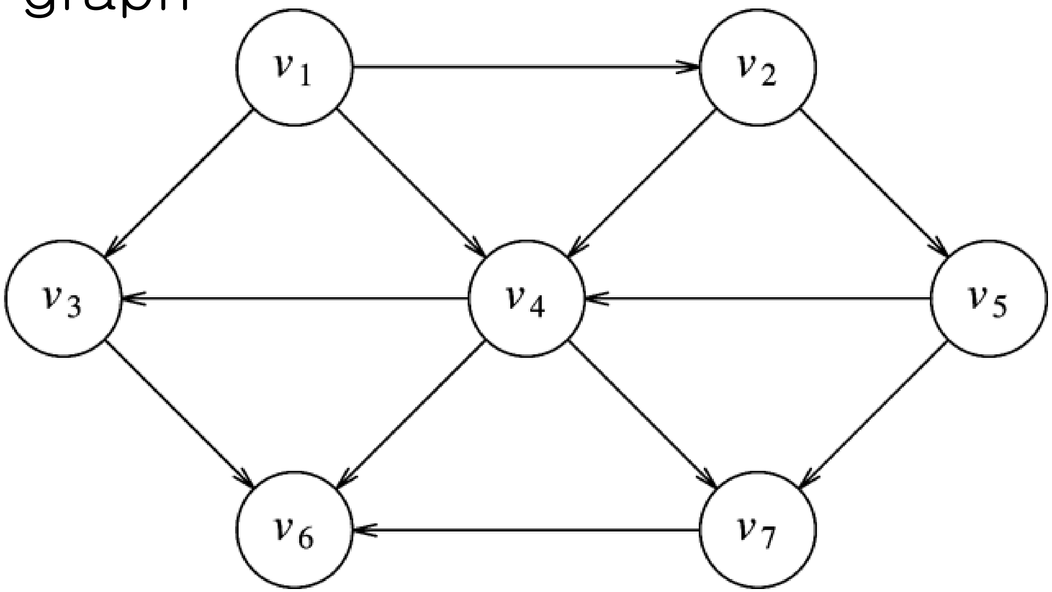


Figure 9.3 An acyclic graph representing course prerequisite structure

Topological ordering

– An acyclic graph



- Possible topological ordering:

$v_1, v_2, v_5, v_4, v_3, v_7, v_6$

$v_1, v_2, v_5, v_4, v_7, v_3, v_6$

Simple topological sort algorithm

```
Topsort( Graph G )
{
    int Counter;
    Vertex V, W;

    for( Counter = 0; Counter < NumVertex; Counter++ )
    {
        V = FindNewVertexOfDegreeZero( );  // sequential scan
        if ( V == NotAVertex )
        {
            Error( "Graph has a cycle" );
            break;
        }
        TopNum[ V ] = Counter;
        for each W adjacent to V
            Indegree[ W ]--;
    }
}
```

Better topological sort algorithm

```
Topsort( Graph G );
{
        Queue Q;
        int Counter = 0;
        Vertex V, W;

        Q = CreateQueue( NumVertex ); MakeEmpty( Q );
        for each vertex V
            if( Indegree[ V ] == 0 )
                Enqueue( V, Q );

        while( !IsEmpty( Q ) )
        {
            V = Dequeue( Q );
            TopNum[ V ] = ++Counter;  /*Assign next number */

            for each W adjacent to V
                if( --Indegree[ W ] == 0 )
                    Enqueue( W, Q );
        }

        if ( Counter != NumVertex )
            Error( "Graph has a cycle" );

        DisposeQueue( Q );  /* Free the memory */
}
```

Topological sort

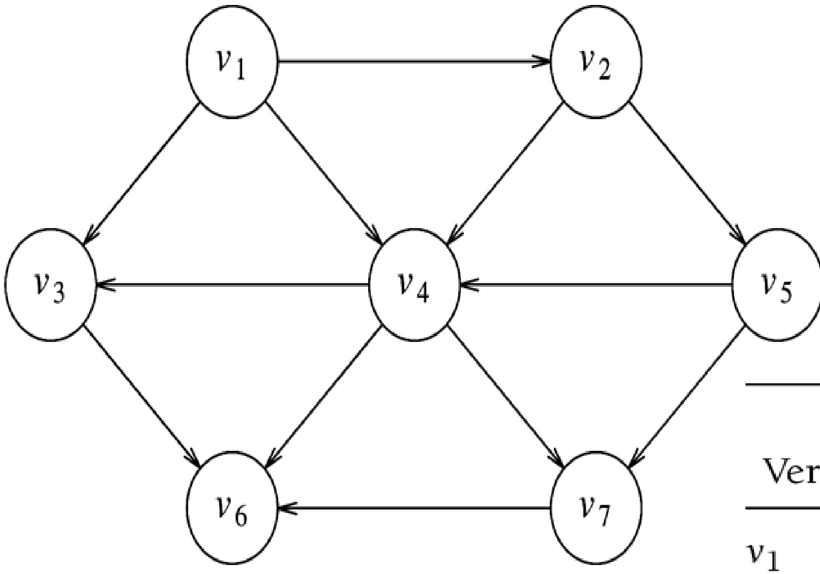
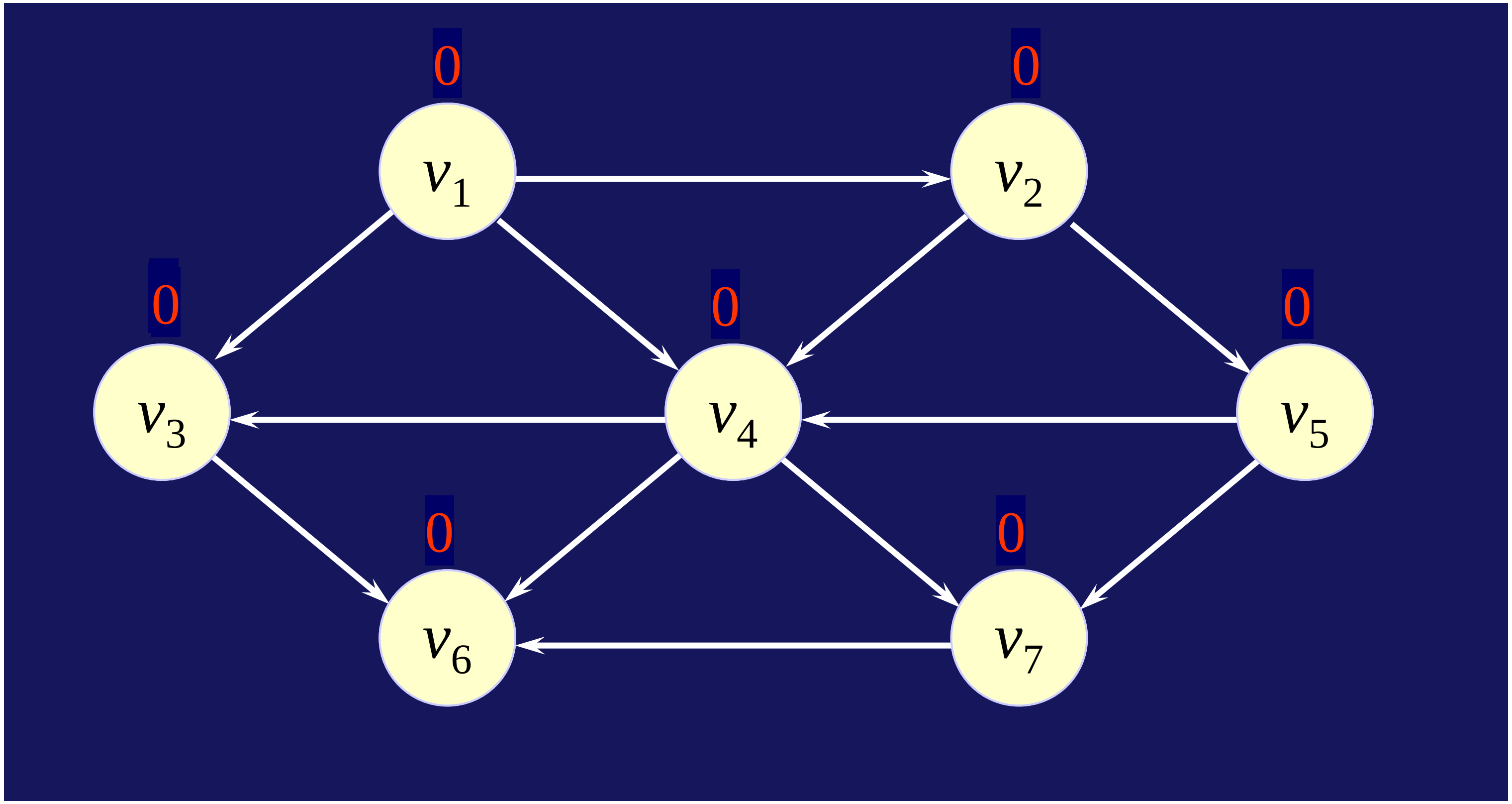


Figure 9.6 Result of applying topological sort to the graph in Figure 9.4

	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

Example again.



Shortest Path Algorithm

- Let's view a graph as the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges are assigned weights which might be the distance between the two cities connected by the edge or the average time to drive the edge

1. Is there a path from A to B?
2. If there is more than one path from A to B, which is the shortest path?

Shortest-Path Algorithms

- The input is a weighted graph: associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the edge.
- The cost of a path $v_1 v_2 \cdots v_N$ is $\sum^{N-1} c_{i,i+1}$

Single-Source Shortest Path Problem :

Given as input (i) a weighted graph G, and (ii) a distinguished vertex s, find the shortest weighted path from s to every other vertex in G

Example

- Which is the shortest path from v_1 to v_6?

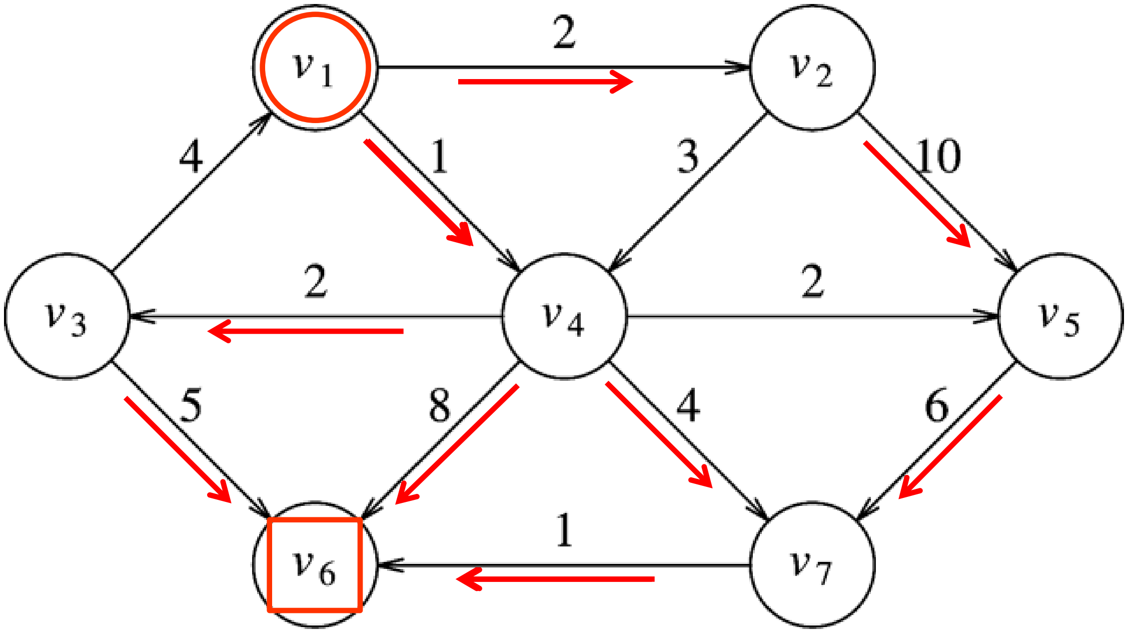


Figure 9.8 A directed graph G

Graph with negative cost cycle

- Same question on the nodes v_5 to v_4?

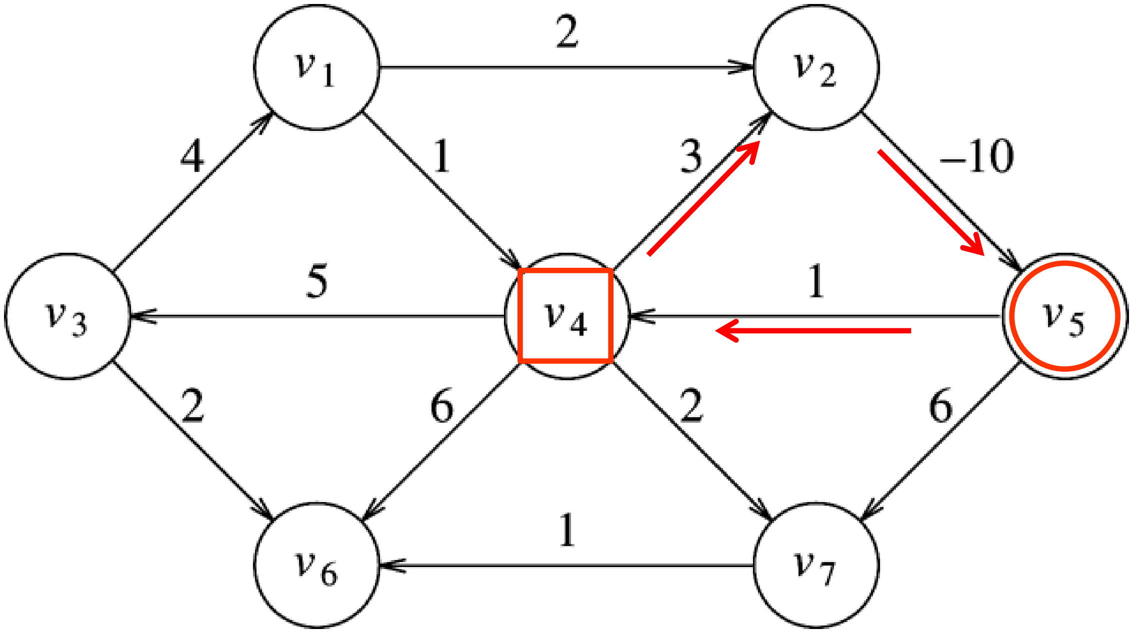


Figure 9.9 A graph with a negative-cost cycle

Four different versions

1. Unweighted shortest path problem
2. Weighted shortest path problem for graphs with no negative edges
3. Weighted shortest path problem for graphs with negative edges
4. Weighted problem for the special case of acyclic graphs

Unweighted shortest path

- Edge has no weight
- Special case of weighted shortest path

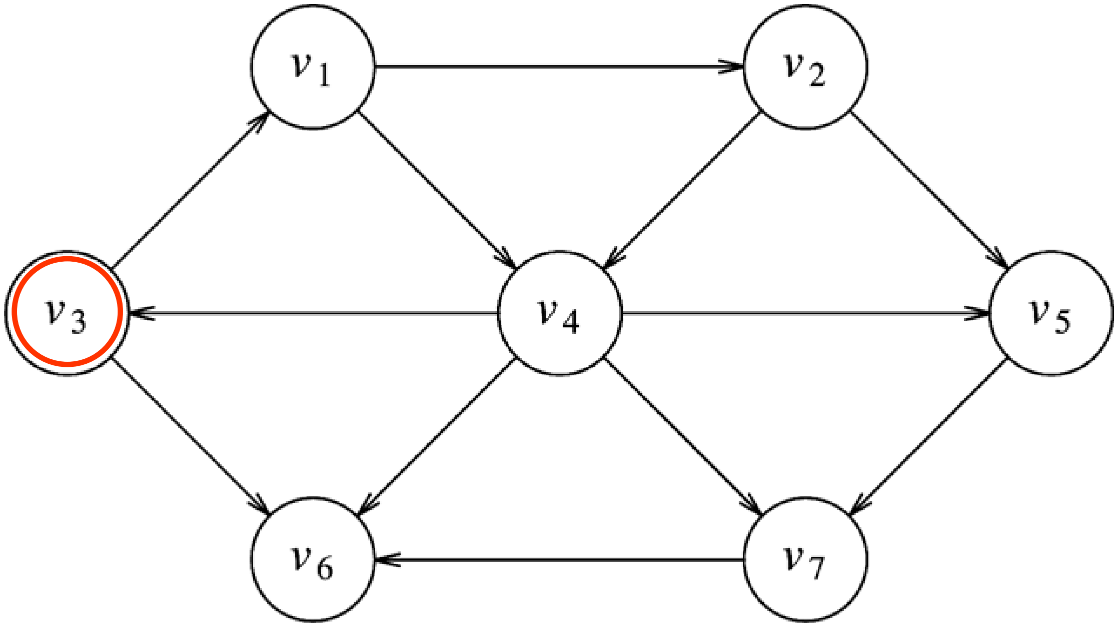


Figure 9.10 An unweighted directed graph G

Finding a shortest path

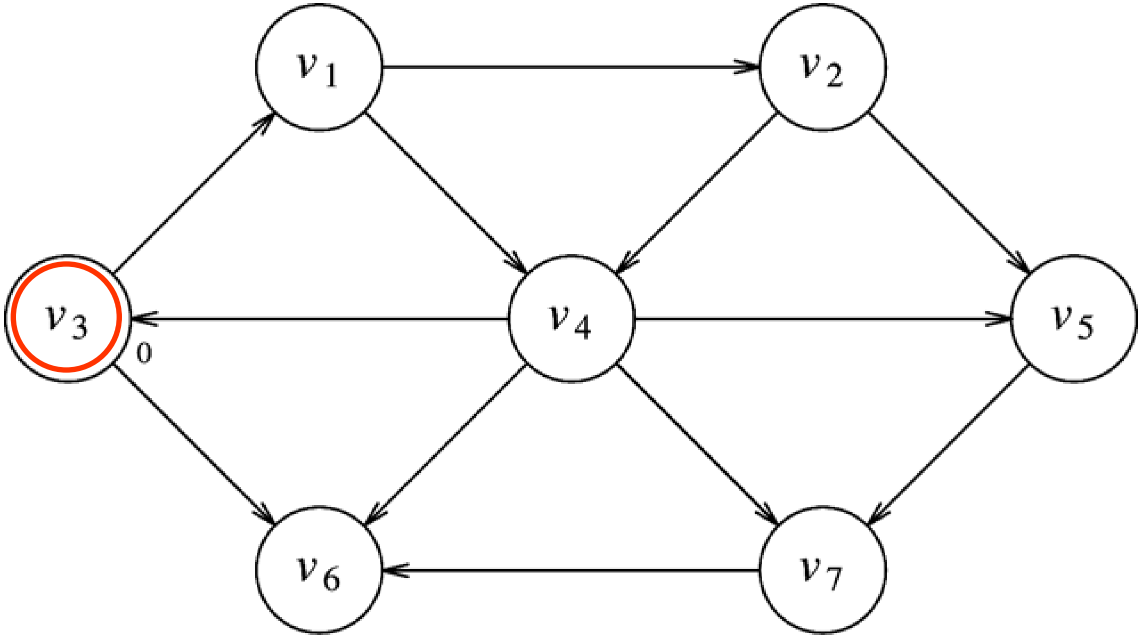


Figure 9.11 Marking the start node as reachable in 0 edges

Finding a shortest path

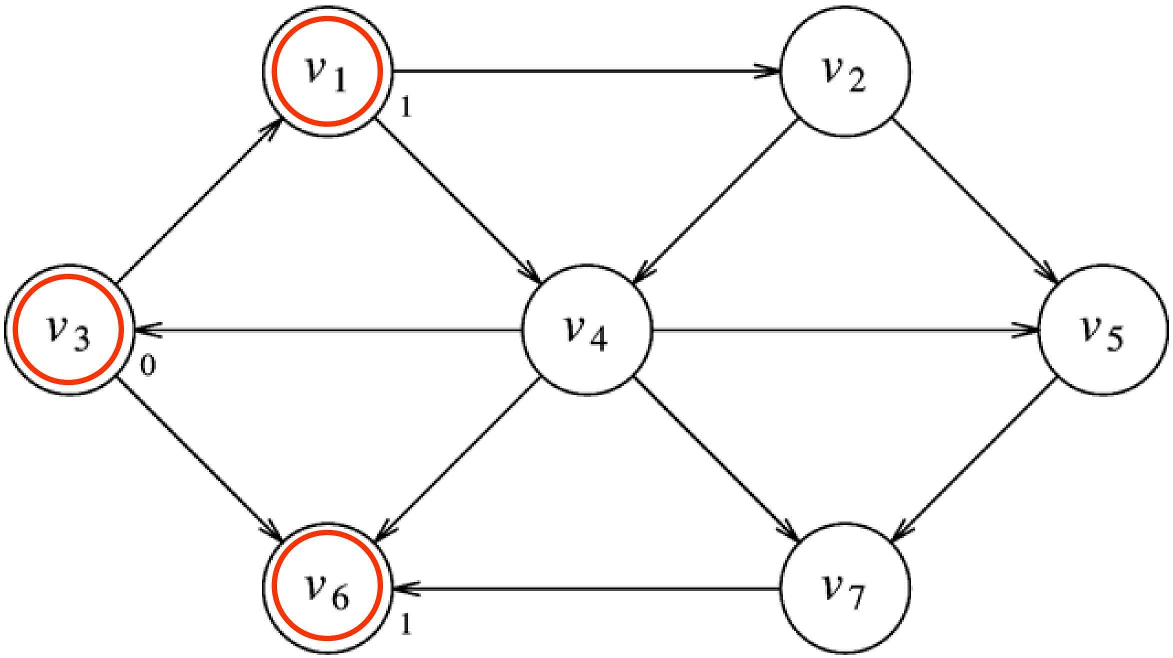


Figure 9.12 Finding all vertices whose path length from *s* is 1

Finding a shortest path

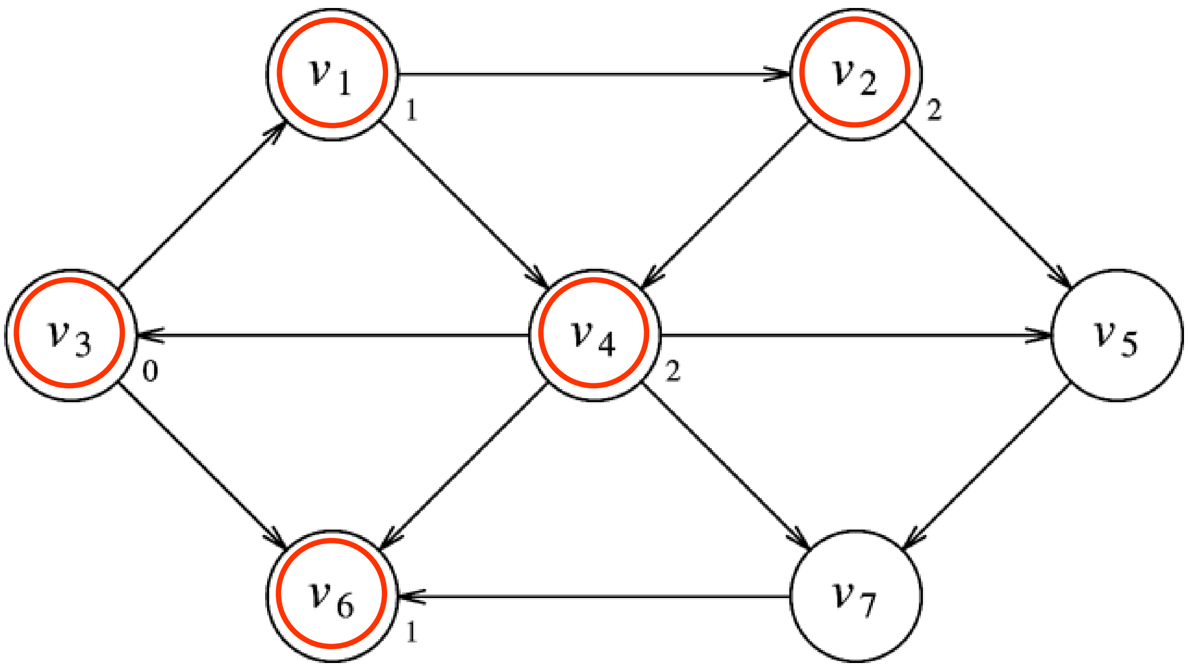


Figure 9.13 Finding all vertices whose shortest path is 2

Finding a shortest path

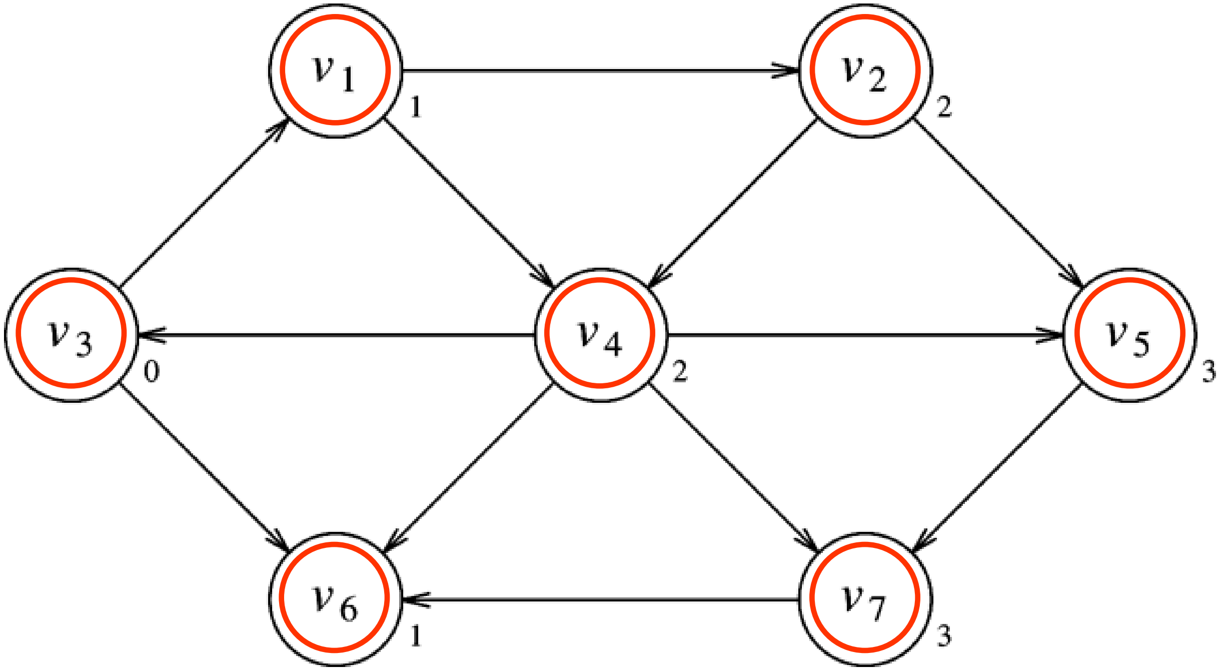


Figure 9.14 Final shortest paths

Graph

- d_v: distance from s
- p_v: actual paths

v	Known	d_v	p_v
v_1	0	∞	0
v_2	0	∞	0
v_3	0	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figure 9.15 Initial configuration of table used in unweighted shortest-path computation

Draft Algorithm

```
        Unweighted( Table T )  /* Assume T is initialized */
        {
            int CurrDist;
            Vertex V, W;

/* 1*/      for( CurrDist = 0; CurrDist < NumVertex; CurrDist++ )
/* 2*/          for each vertex V
/* 3*/              if ( !T[ V ].Known && T[ V ].Dist == CurrDist )
                    {
/* 4*/                  T[ V ].Known = True;
/* 5*/                  for each W adjacent to V
/* 6*/                      if( T[ W ].Dist == Infinity )
                            {
/* 7*/                          T[ W ].Dist = CurrDist + 1;
/* 8*/                          T[ W ].Path = V;
                            }
                    }
        }
```

Analysis on Draft Algorithm

- The running time is $O(|V|^2)$ due to the doubly nested ***for*** loops in the algorithm
- The outside loop continues to the end even if all the vertices become known much earlier.
- Extra test to avoid this does not affect the worst-case running time, for instance for the next graph
- Possible solution is using queue (its algorithm on the next slide)

Bad case

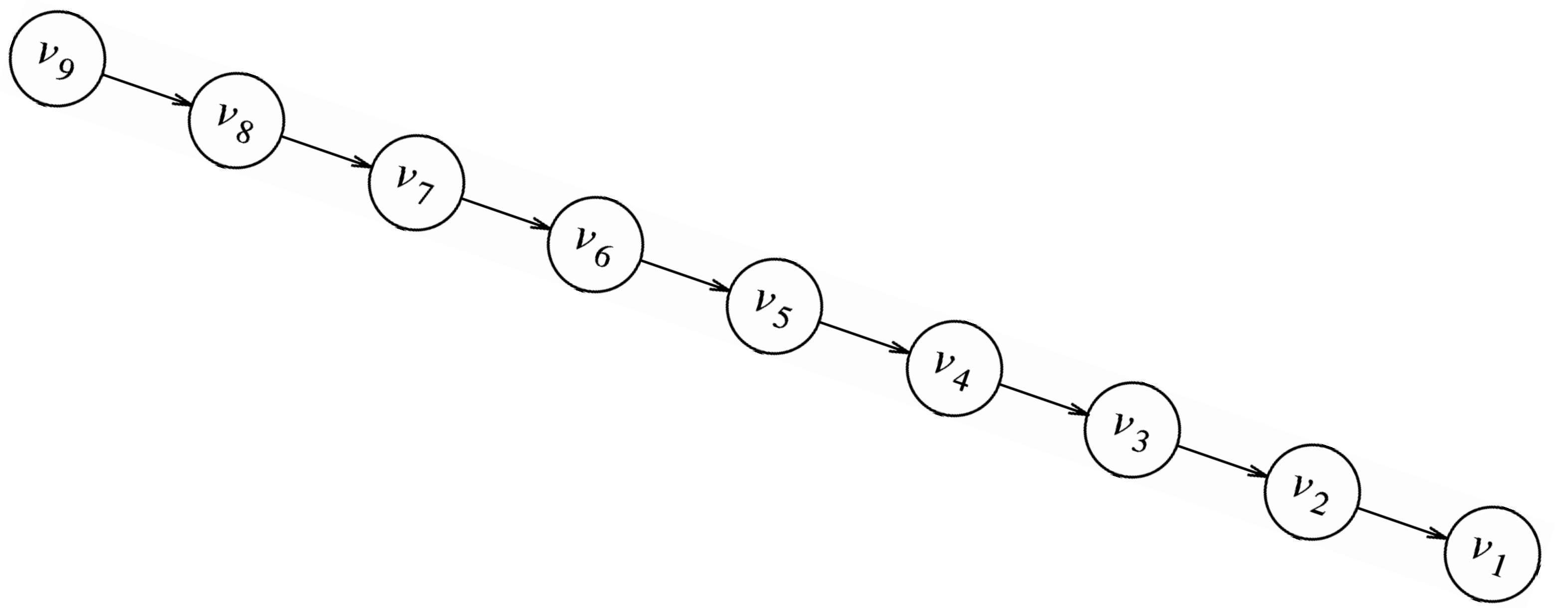


Figure 9.17 A bad case for unweighted shortest-path algorithm using Fig 9.16

```
        Unweighted( Table T ) /* Assume T is initialized (Fig 9.30
        {
            Queue Q;
            Vertex V, W;

/* 1*/      Q = CreateQueue( NumVertex ); MakeEmpty( Q );

            /* Enqueue the start vertex S, determined elsewhere */
/* 2*/      Enqueue( S, Q );

/* 3*/      while( !IsEmpty( Q ) )
            {
/* 4*/          V = Dequeue( Q );
/* 5*/          T[ V ].Known = True; /* Not really needed anymore *

/* 6*/          for each W adjacent to V
/* 7*/              if( T[ W ].Dist == Infinity )
                    {
/* 8*/                  T[ W ].Dist = T[ V ].Dist + 1;
/* 9*/                  T[ W ].Path = V;
/*10*/                  Enqueue( W, Q );
                    }
            }
/*11*/      DisposeQueue( Q ); /* Free the memory */
        }
```

Tracing Execution

	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		

Figure 9.19 How the data changes during the unweighted SPA

Tracing Execution

	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		

Figure 9.19 How the data changes during the unweighted SPA

Weighted Shortest Path

- Using Dijkstra's Algorithm
- For single-source shortest path problem
- Example of Greedy Algorithm: By solving a problem in stages by doing what appears to be the best thing at each stage

- Greedy algorithms do not always work

Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

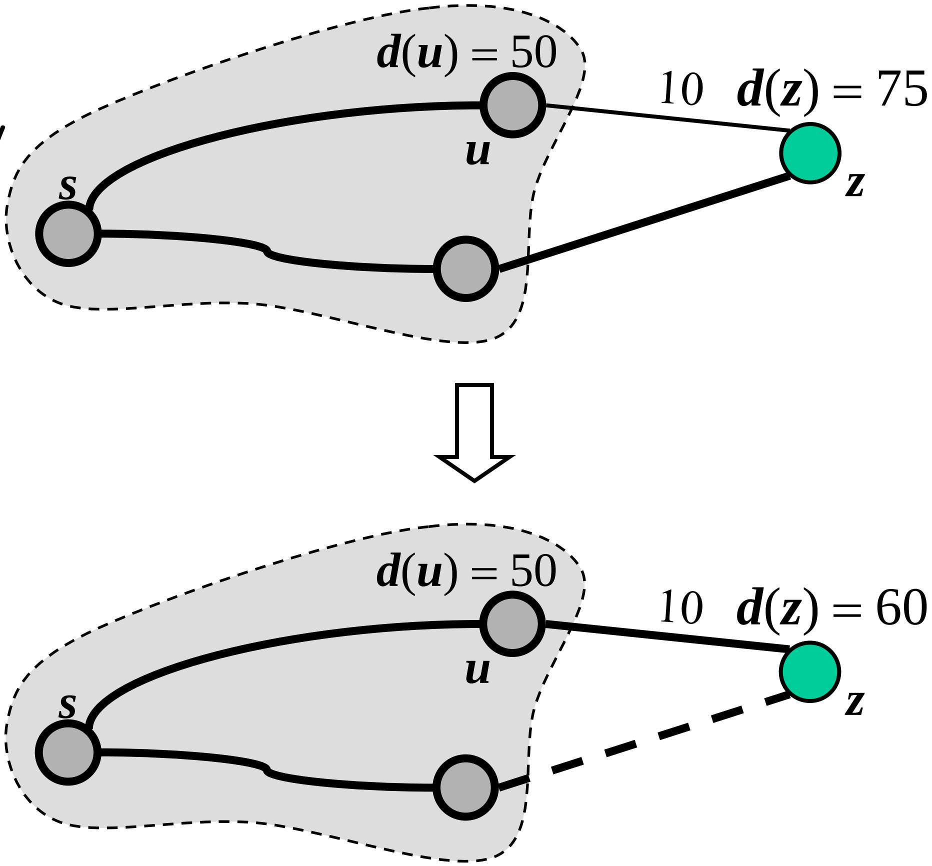
Dijkstra's Algorithm

- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

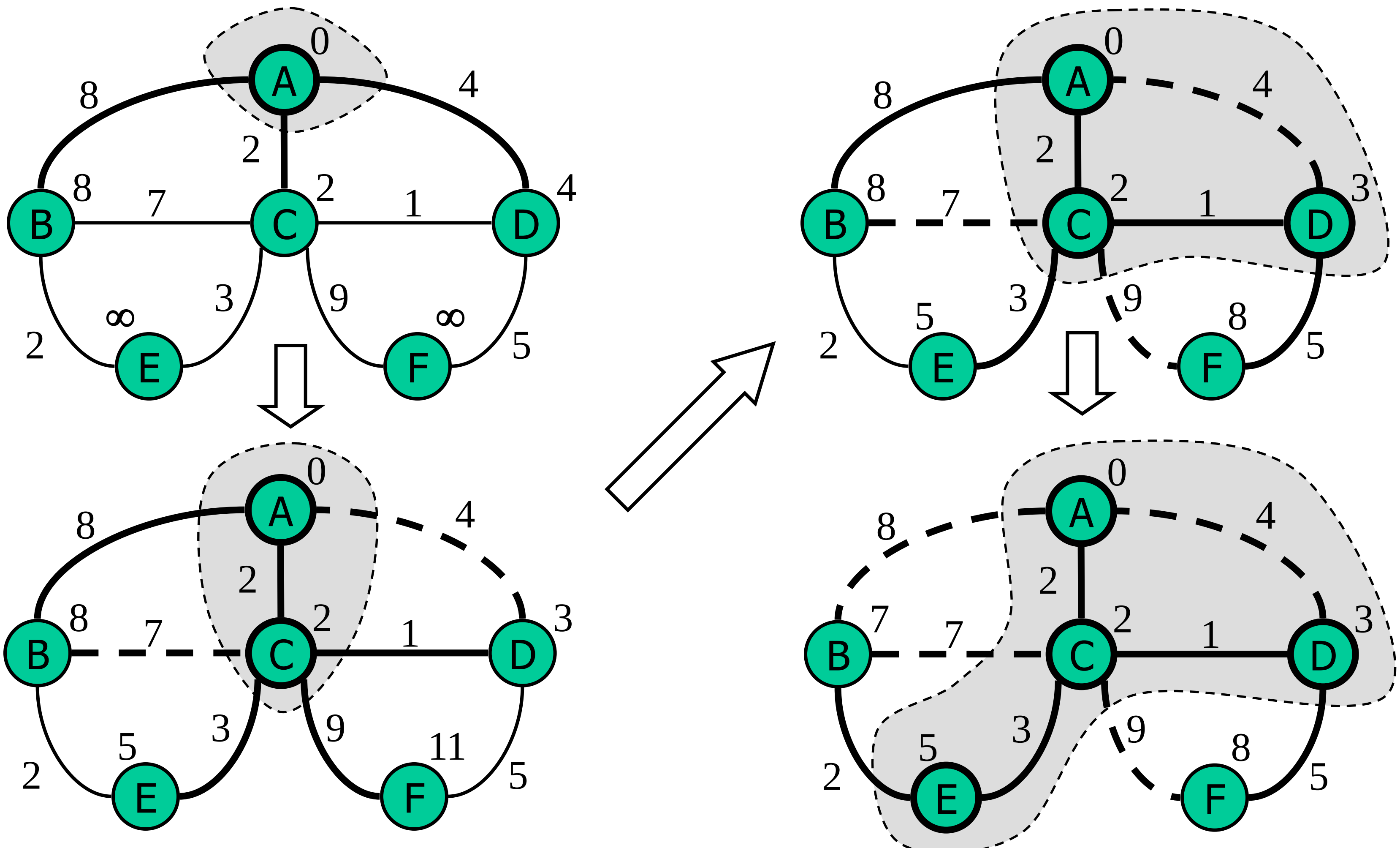
Edge Relaxation

- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows

$$d(z) \leftarrow \min(d(z), d(u) + weight(e))$$

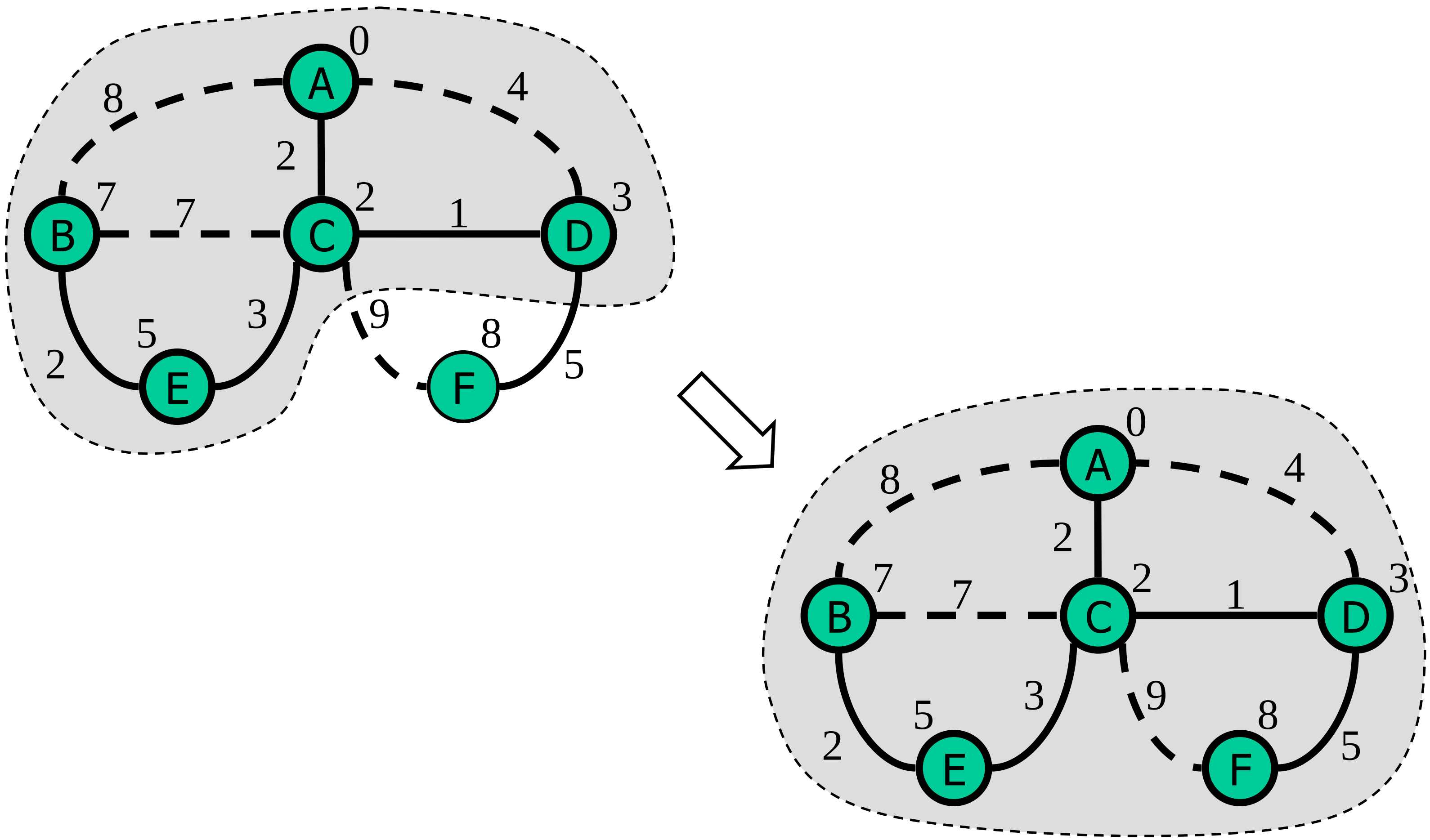


Example

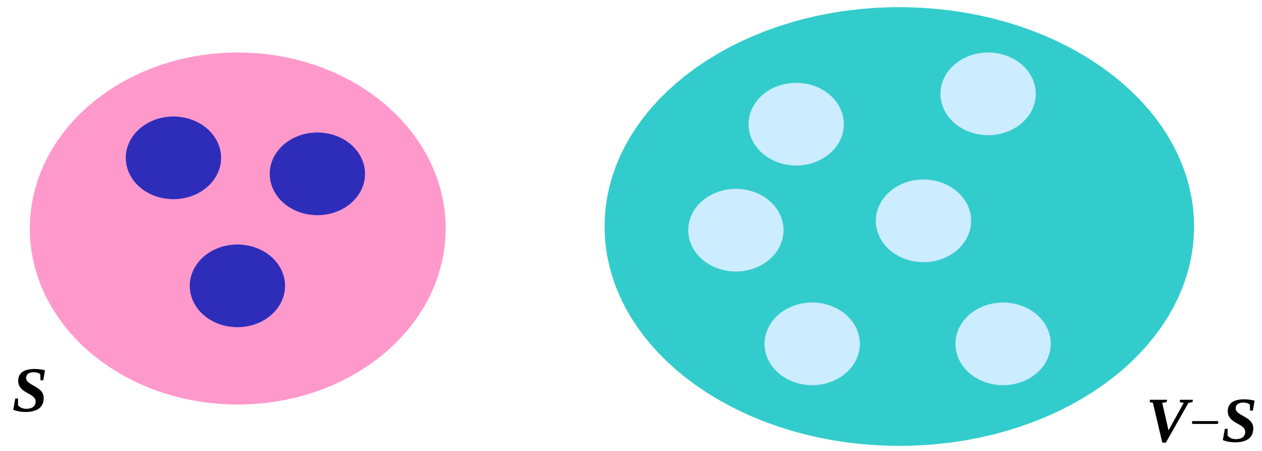


A B C D E F
0 8 4 2 8 7 2 1 4 ∞ 3 9 ∞ 2 5
A B C D E F
0 8 4 2 8 7 2 1 3 5 3 9 11 2 5
A B C D E F
0 8 4 2 8 7 2 1 3 5 3 9 8 2 5
A B C D E F
0 8 4 2 7 7 2 1 3 5 3 9 8 2 5

Example (cont.)



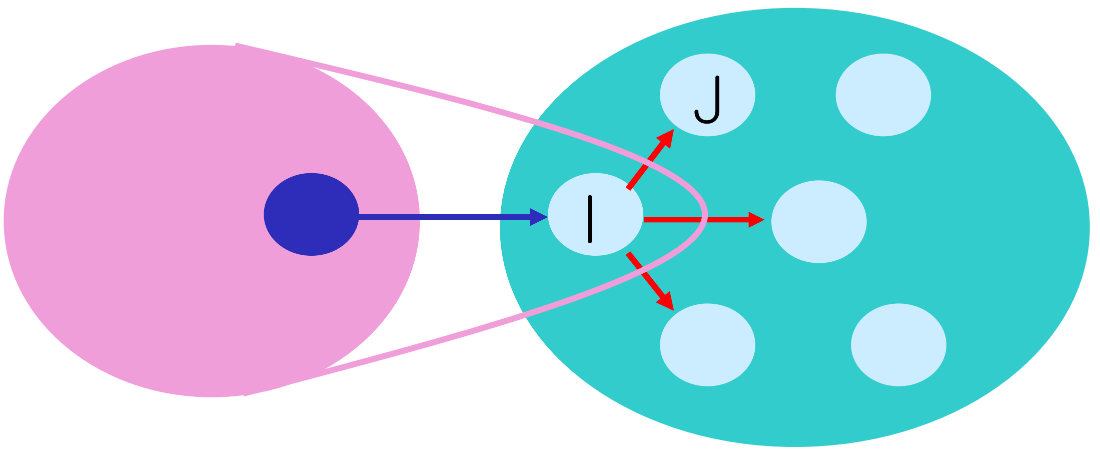
Single-source shortest path



If $v \in S$, $\boldsymbol{Best}(v)$ is the length of the shortest path between source and v

If $v \in S$, $\boldsymbol{Best}(v)$ is the length of the shortest path between source and v when using only the vertices in S as intermediate vertices

Dijkstra's algorithm



S $V-S$

Dijkstra’s algorithm

S = Φ; Best[1] = 0; Best[all the other vertices] = ∞ ;

BuildHeap(***v***) for each vertex *v* with value Best[***v***] ;

while S has fewer than n nodes **do** {

1) Select the vertex ***i*** with the smallest Best[***i***] in V-S ;
2) S ← S ∪ {***i***} ;
3) for each vertex ***j*** in V-S adjacent to ***i***

    If Best[***j***] > Best[***i***] + C[***i***, ***j***] {

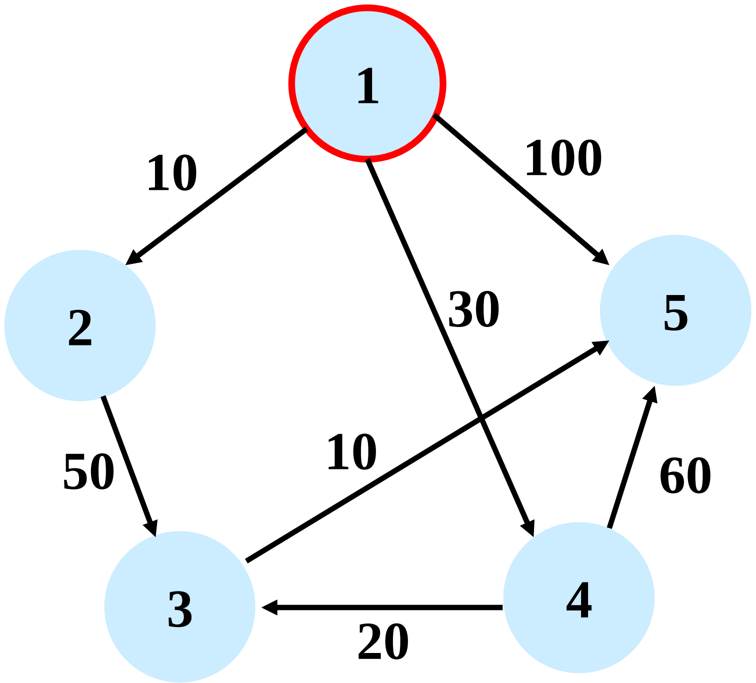
        Best[***j***] = Best[***i***] + C[***i***, ***j***] ;

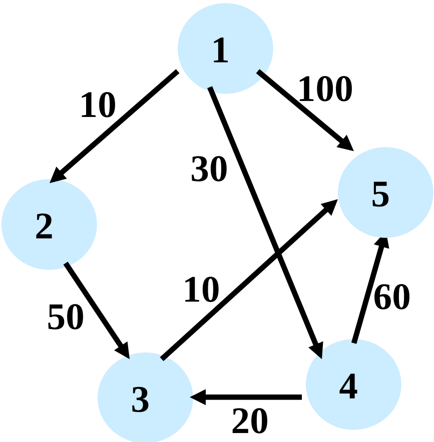
        P[***j***] = ***i***;

    }

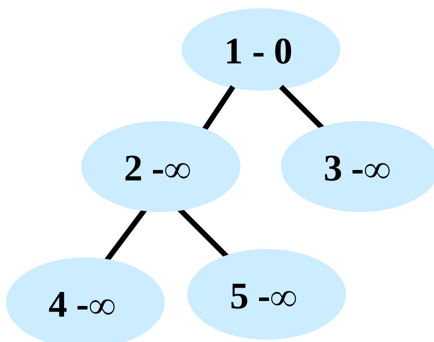
}

Example

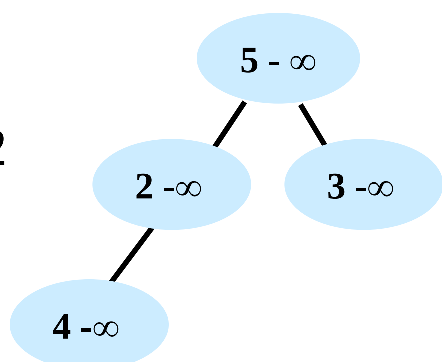




Step 1



Step 2



		S		V-S
Step 1		Φ		1 2 3 4 5
	Best		Best	0 ∞ ∞ ∞ ∞
Step 2		1		2 3 4 5
	Best	0	Best	10 ∞ 30 100
	p	-	p	1 - 1 1
Step 3		1 2		3 4 5
	Best	0 10	Best	60 30 100
	p	- 1	p	2 1 1
Step 4		1 2 4		3 5
	Best	0 10 30	Best	50 90
	p	- 1 1	p	4 4
Step 5		1 2 4 3		5
	Best	0 10 30 50	Best	60
	p	- 1 1 4	p	3
Step 6		1 2 4 3 5		
	Best	0 10 30 50 60		
	p	- 1 1 4 3		

Graph

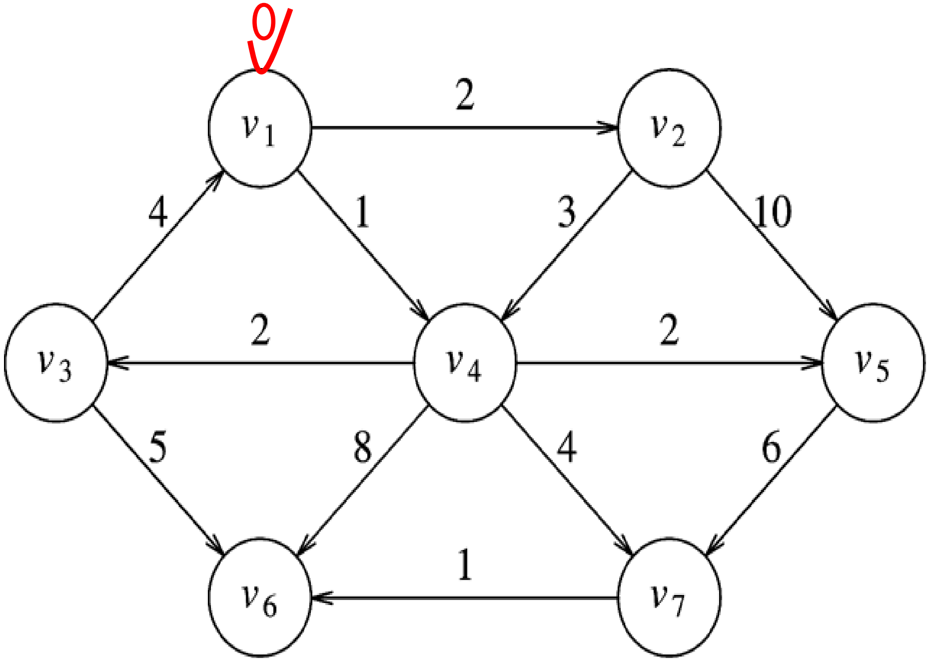
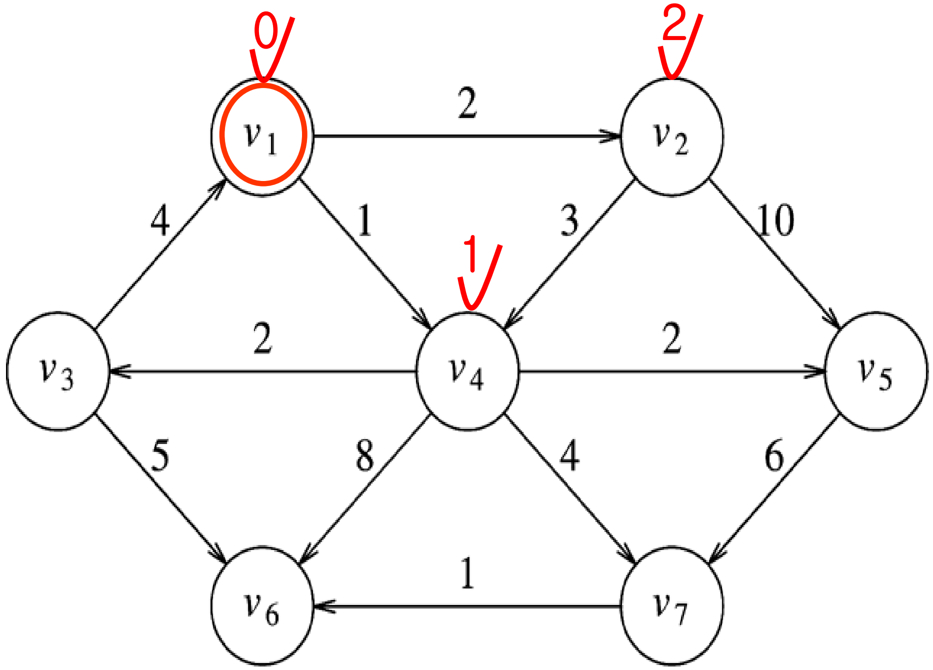


Figure 9.20 The directed graph G

v	Known	d_v	p_v
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figure 9.21 Initial configuration

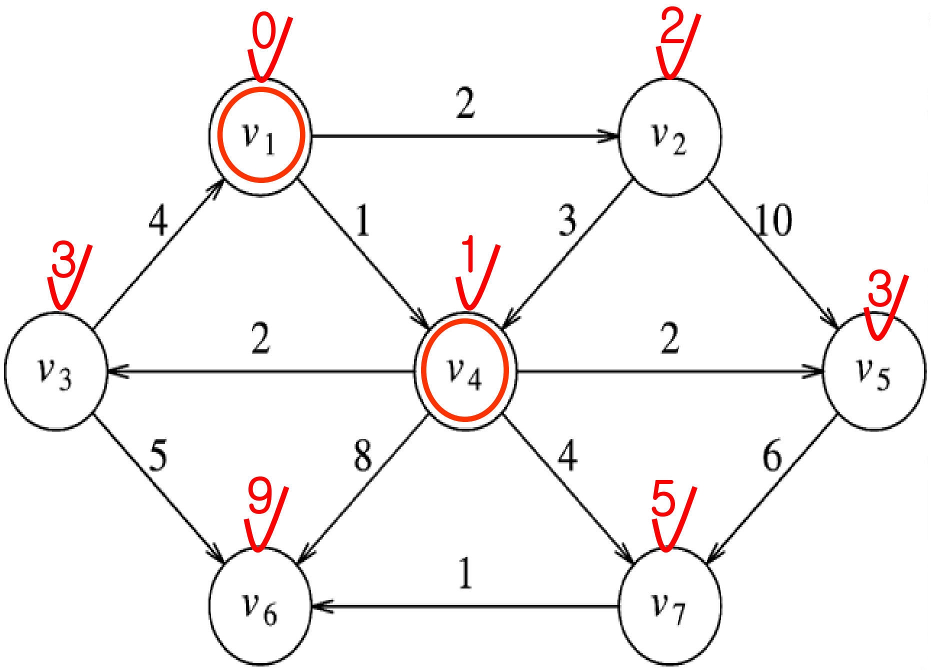
Graph



v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	∞	0
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Figure 9.22 After v_1 is declared known

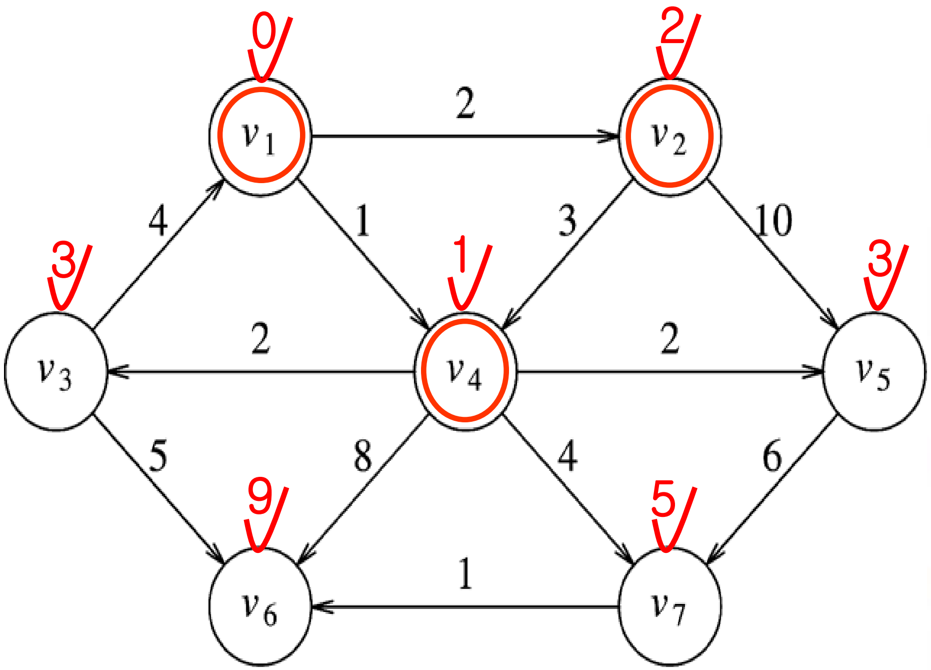
Graph



v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

Figure 9.23 After v_4 is declared known

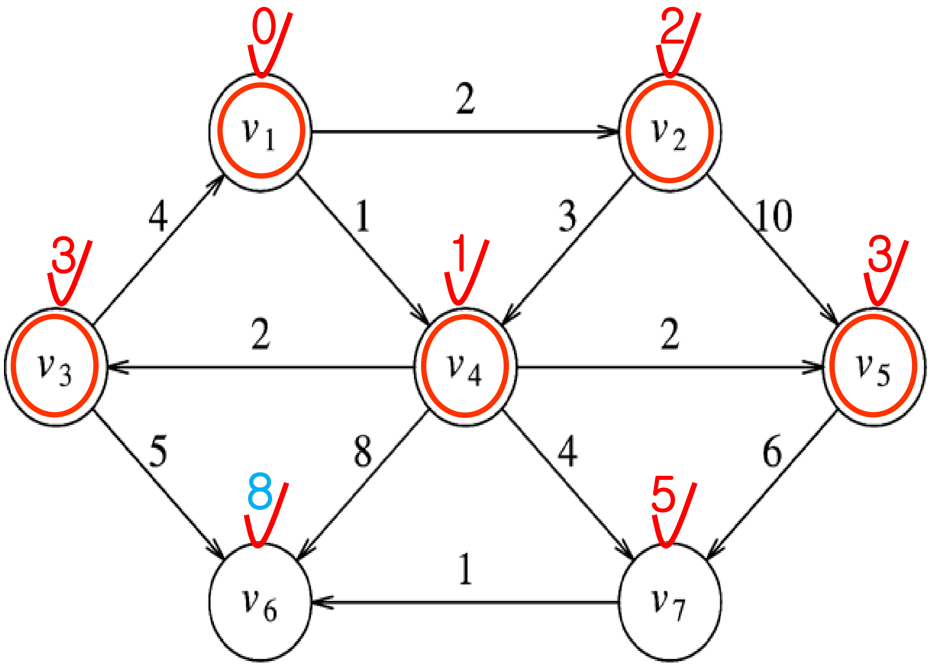
Graph



v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

Figure 9.24 After v_2 is declared known

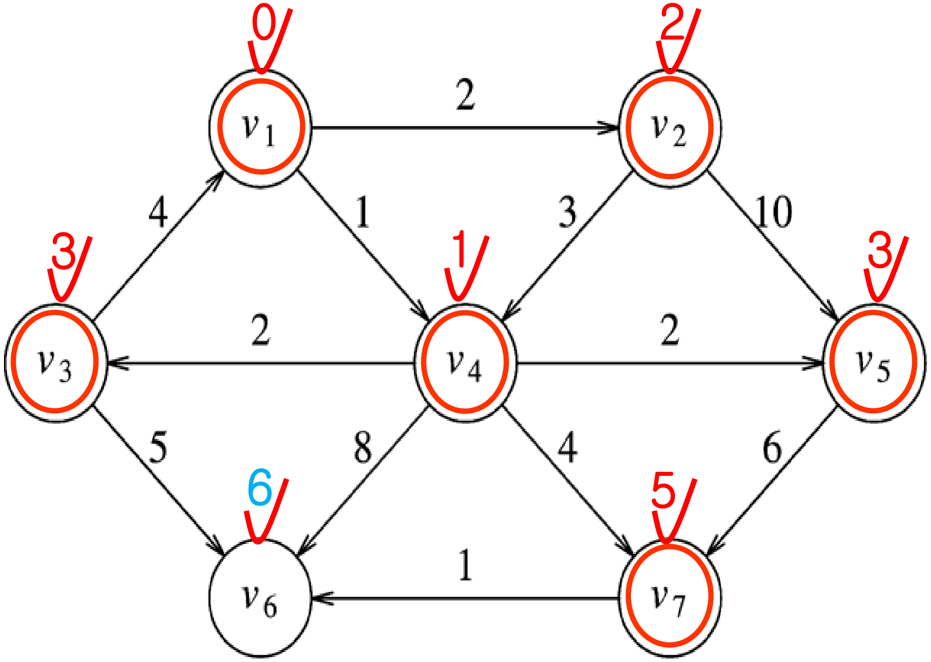
Graph



v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	8	v_3
v_7	0	5	v_4

Figure 9.25 After v_5 & v_3 are declared known

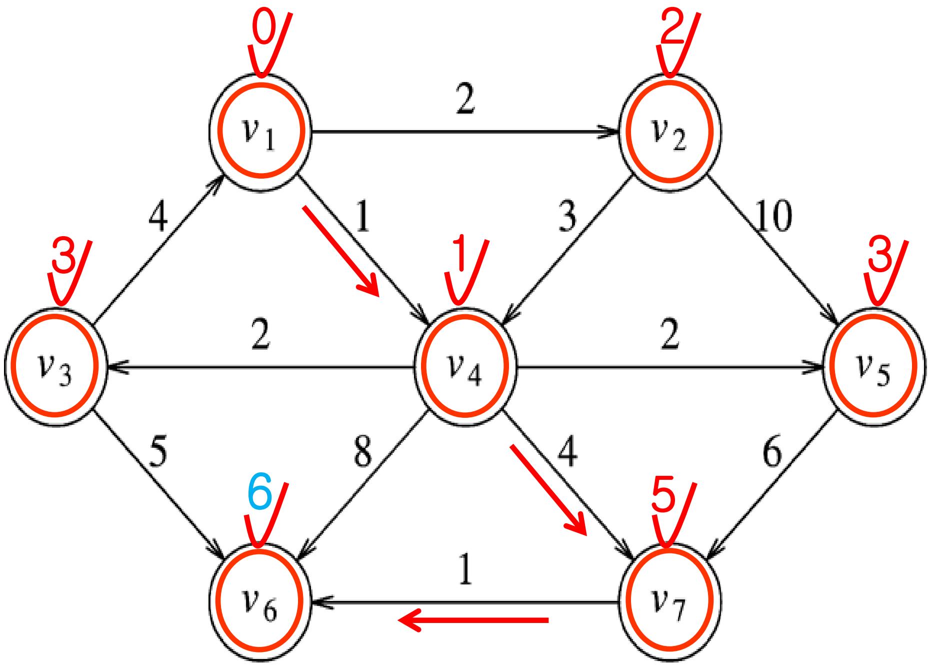
Graph



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

Figure 9.26 After v_7 is declared known

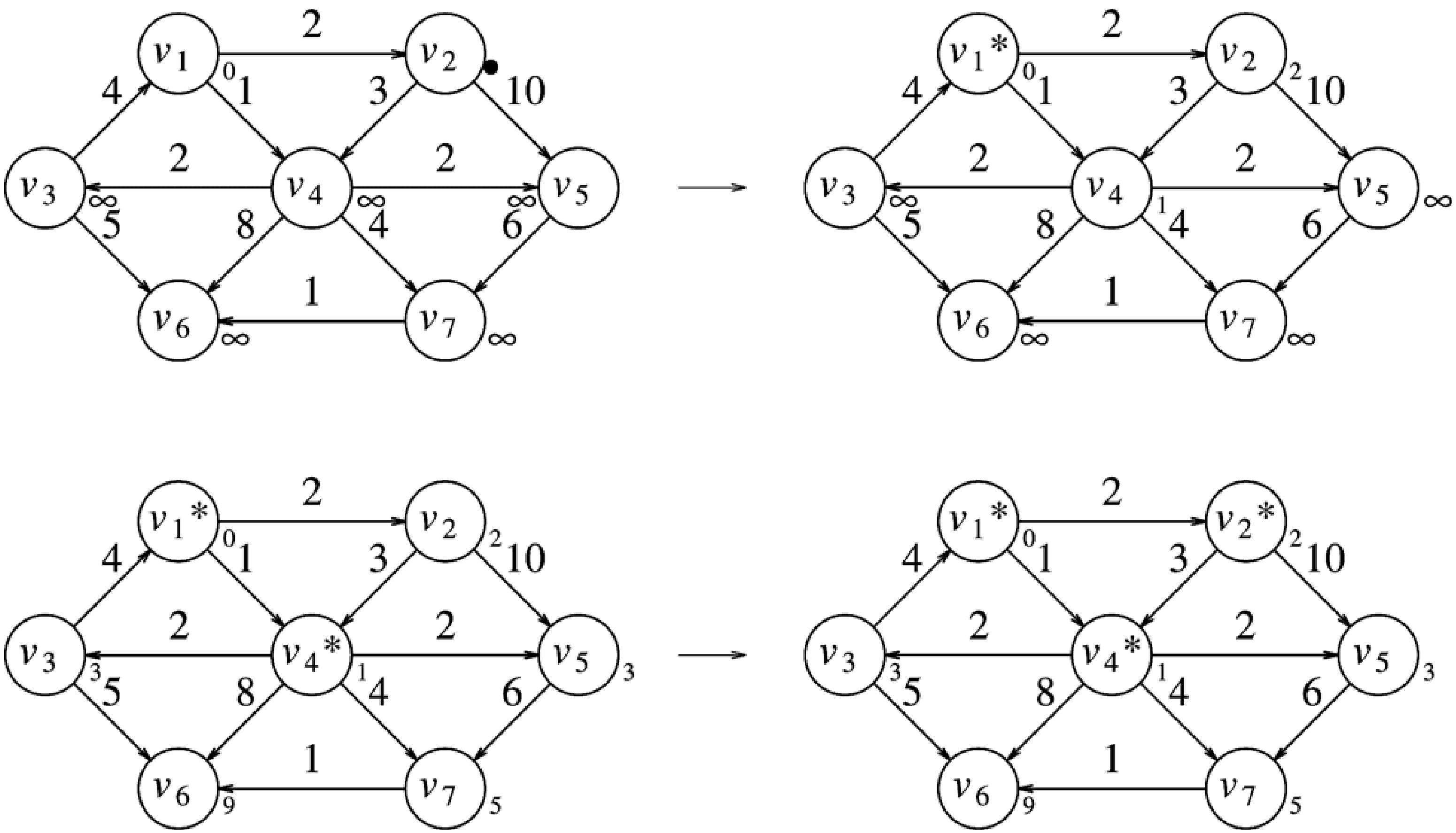
Graph



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

Figure 9.27 After v_6 is declared known and algorithm terminates

Stages of Dijkestra's algorithm



Stages of Dijkstra's algorithm

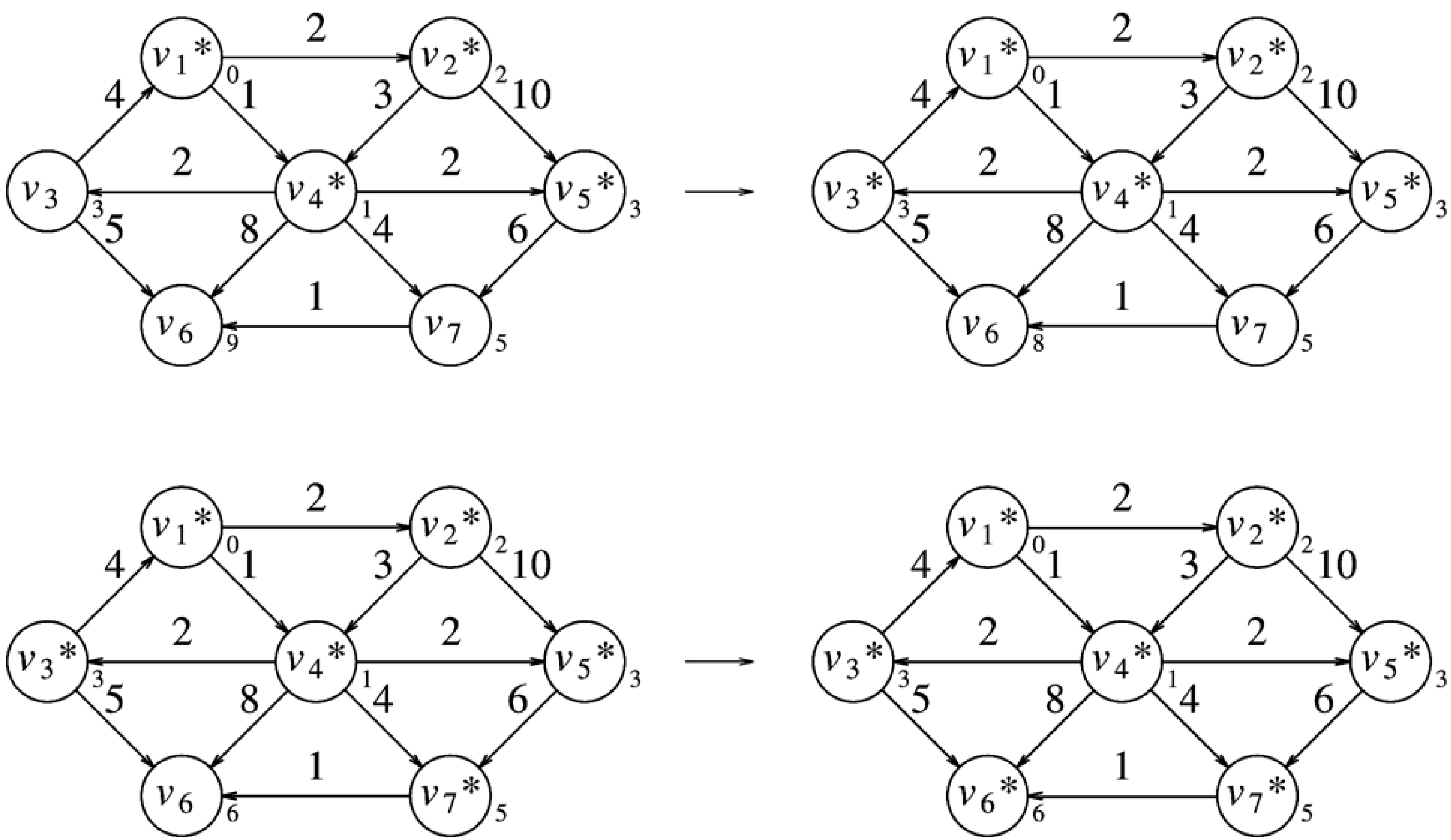


Table Structure

```
typedef int Vertex;

struct TableEntry
{
    List     Header; /* Adjacency list */
    int      Known;
    DistType Dist;
    Vertex   Path;
};

/* Vertices are numbered from 0 */
#define NotAVertex (-1)
typedef struct TableEntry Table[ NumVertex ];
```

Table Initializer

```
        void
        InitTable( Vertex Start, Graph G, Table T )
        {
            int i;

/* 1*/      ReadGraph( G, T );  /* Read graph somehow */
/* 2*/      for( i = 0; i < NumVertex; i++ )
            {
/* 3*/          T[ i ].Known = False;
/* 4*/          T[ i ].Dist  = Infinity;
/* 5*/          T[ i ].Path  = NotAVertex;
            }
/* 6*/      T[ Start ].dist = 0;
        }
```

Path Printing Algorithm

```
/* Print shortest path to V after Dijkstra has run */
/* Assume that the path exists */

void
PrintPath( Vertex V, Table T )
{
    if( T[ V ].Path != NotAVertex )
    {
        PrintPath( T[ V ].Path, T );
        printf( " to" );
    }
    printf( "%v", V );  /* %v is pseudocode */
}
```

Dijkstra's algorithm

```
        Dijkstra( Table T )
        {
            Vertex V, W;

/* 1*/      for( ; ; )
            {
/* 2*/          V = smallest unknown distance vertex;
/* 3*/          if( V == NotAVertex )
/* 4*/              break;

/* 5*/          T[ V ].Known = True;
/* 6*/          for each W adjacent to V
/* 7*/              if( !T[ W ].Known )
/* 8*/                  if( T[ V ].Dist + Cvw < T[ W ].Dist )
                        {   /* Update W */
/* 9*/                      Decrease( T[ W ].Dist to
                                      T[ V ].Dist + Cvw );
/*10*/                      T[ W ].Path = V;
                        }
            }
        }
```

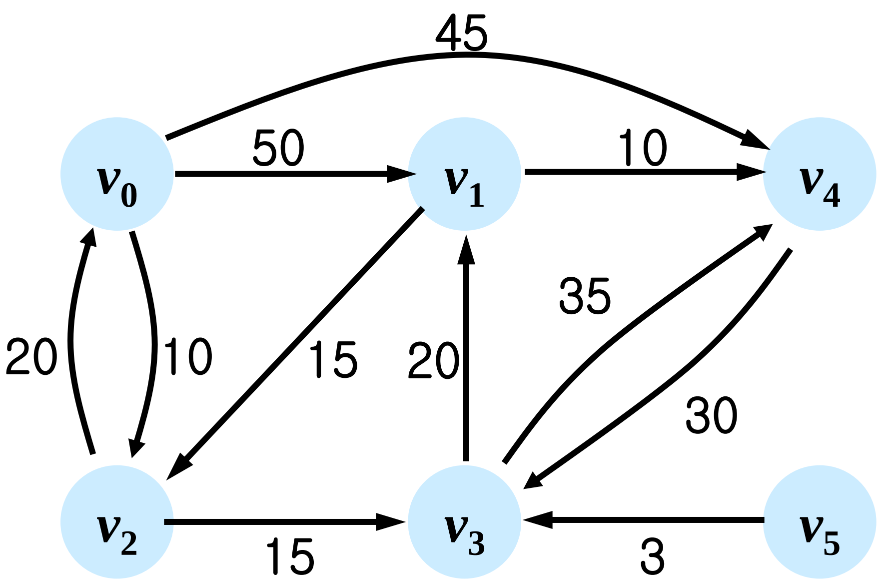
Dijkstra's algorithm

```
          WeightedNegative( Table T )
          {
              Queue Q;
              Vertex V, W;

/* 1*/        Q = CreateQueue( NumVertex ); MakeEmpty( Q );
/* 2*/        Enqueue( S, Q ); /* Enqueue the start vertex S */

/* 3*/        while( !IsEmpty( Q ) )
              {
/* 4*/            V = Dequeue( Q );
/* 5*/            for each W adjacent to V
/* 6*/                if( T[ V ].Dist + Cvw < T[ W ].Dist )
                      {
                          /* Update W */
/* 7*/                    T[ W ].Dist = T[ V ].Dist + Cvw;
/* 8*/                    T[ W ].Path = V;
/* 9*/                    if( W is not already in Q )
/*10*/                        Enqueue( W, Q );
                      }
              }
/*11*/        DisposeQueue( Q );
          }
```

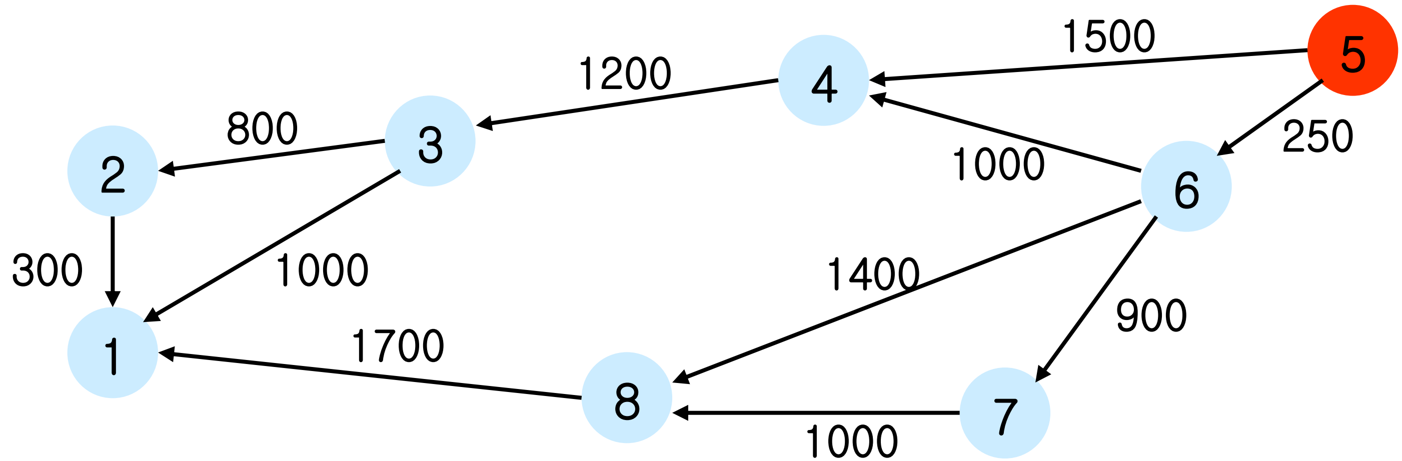
More Example



	path	length
1	$v_0 v_2$	10
2	$v_0 v_2 v_3$	25
3	$v_0 v_2 v_3 v_1$	45
4	$v_0 v_4$	45

– Graph and Shortest Paths from v_0 to All Destination

More Example



i	S	*u*	1	2	3	4	5	6	7	8
			∞	∞	∞	1500	0	250	∞	∞
1	5	6	∞	∞	∞	1250	0	250	1150	1650
2	5 6	7	∞	∞	∞	1250	0	250	1150	1650
3	5 6 7	4	∞	∞	2450	1250	0	250	1150	1650
4	5 6 7 4	8	3350	∞	2450	1250	0	250	1150	1650
5	5 6 7 4 8	3	3350	3250	2450	1250	0	250	1150	1650
6	5 6 7 4 8 3	2	3350	3250	2450	1250	0	250	1150	1650