

# Directed Graphs (Part II)

CSE 373

Data Structures

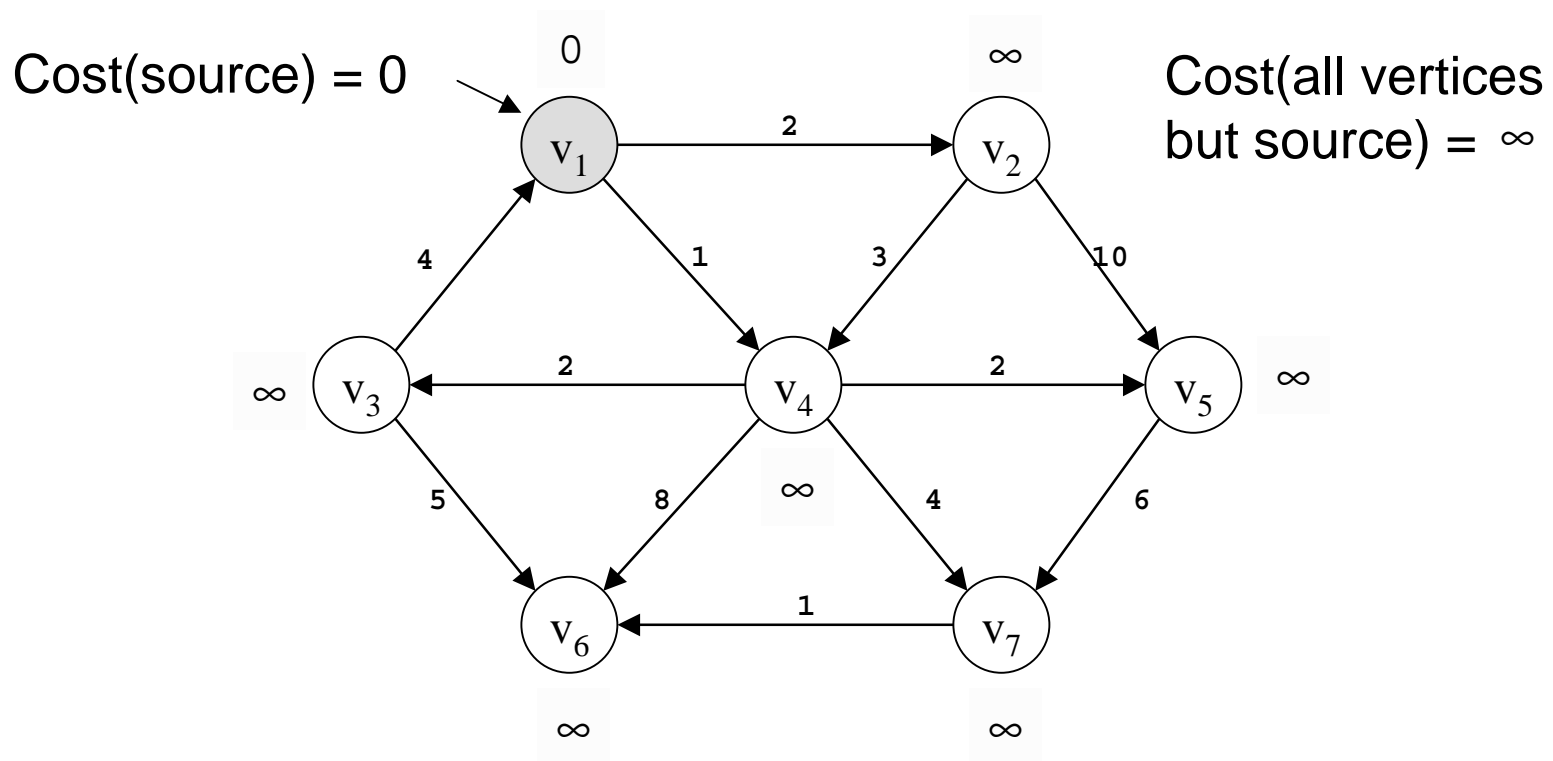
Lecture 15

# Dijkstra's Shortest Path Algorithm

---

- Initialize the cost of  $s$  to 0, and all the rest of the nodes to  $\infty$
- Initialize set  $S$  to be  $\emptyset$ 
  - ›  $S$  is the set of nodes to which we have a shortest path
- While  $S$  is not all vertices
  - › Select the node  $A$  with the lowest cost that is not in  $S$  and identify the node as now being in  $S$
  - › for each node  $B$  adjacent to  $A$ 
    - if  $\text{cost}(A) + \text{cost}(A,B) < B$ 's currently known cost
      - set  $\text{cost}(B) = \text{cost}(A) + \text{cost}(A,B)$
      - set  $\text{previous}(B) = A$  so that we can remember the path

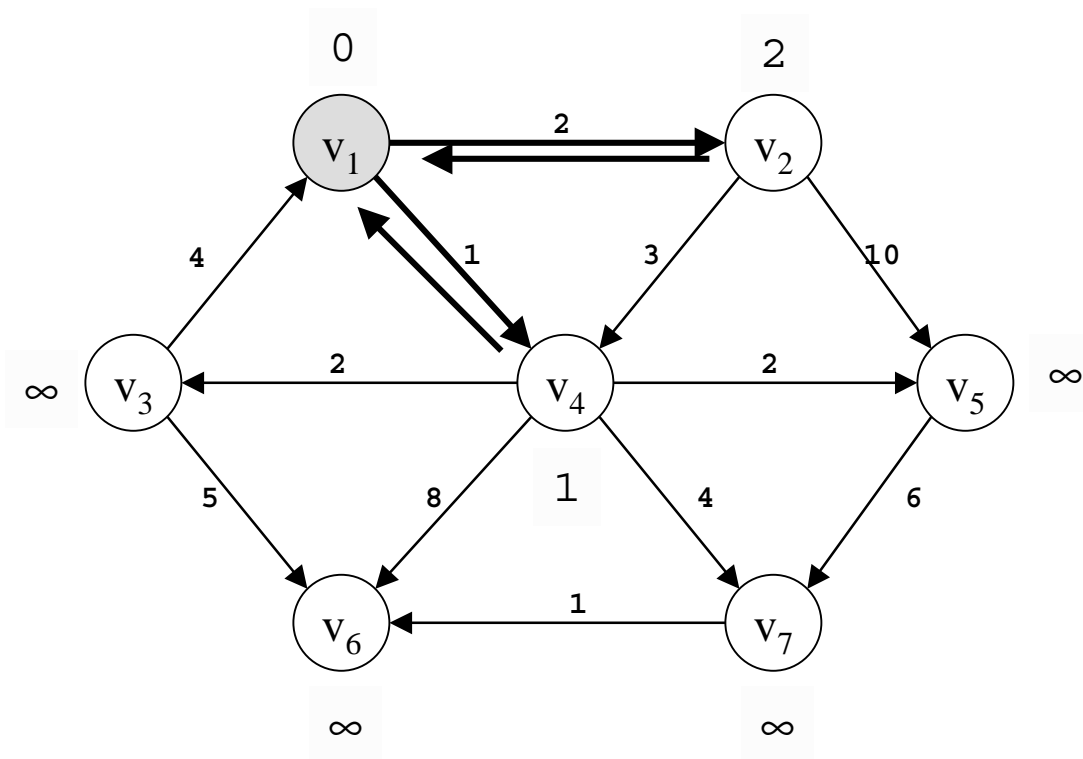
# Example: Initialization



Pick vertex not in  $S$  with lowest cost.

# Example: Update Cost neighbors

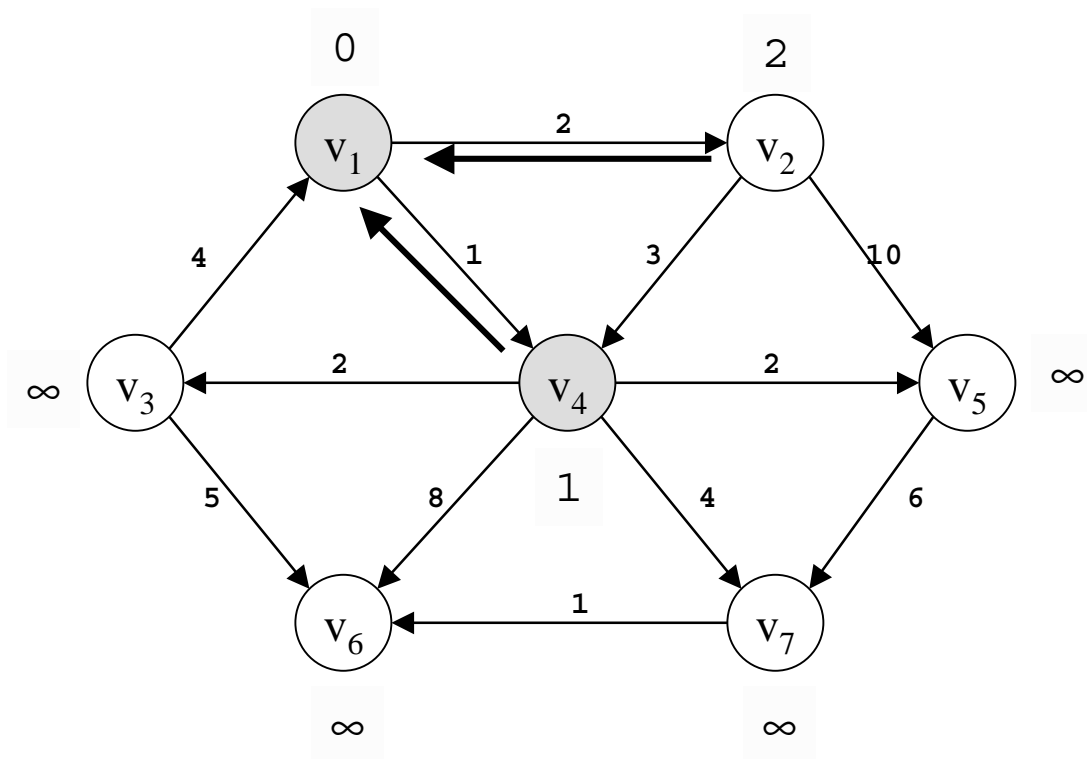
---



$\text{Cost}(v_2) = 2$   
 $\text{Cost}(v_4) = 1$

# Example: pick vertex with lowest cost and add it to S

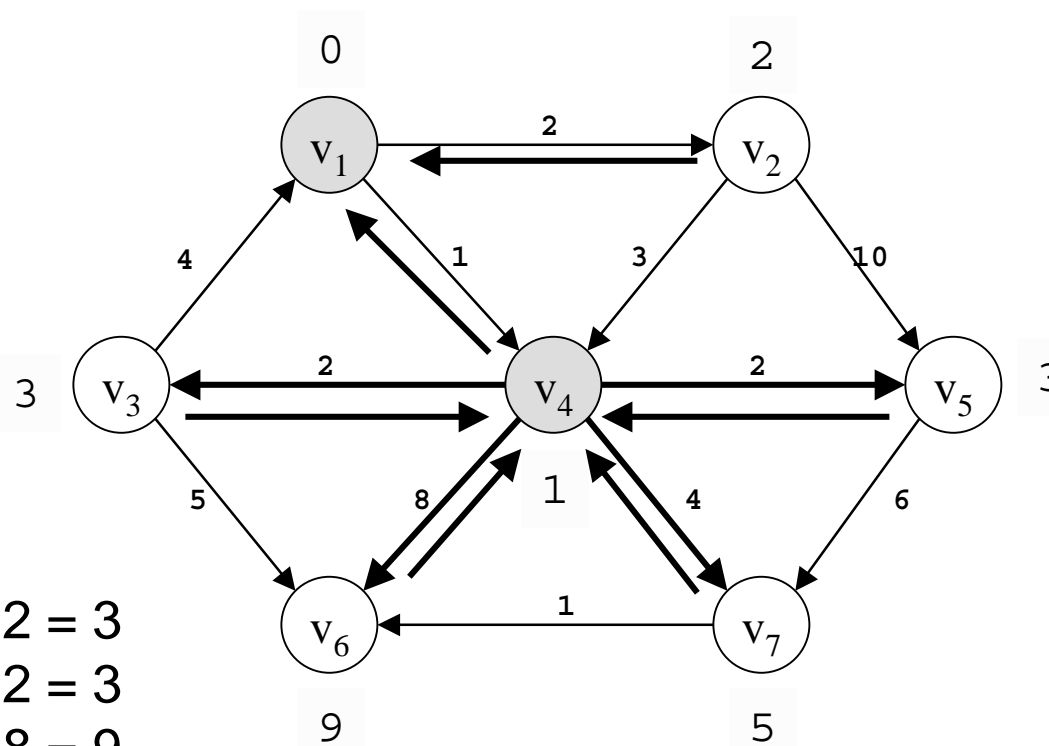
---



Pick vertex not in  $S$  with lowest cost, i.e.,  $v_4$

# Example: update neighbors

---



$$\text{Cost}(v_3) = 1 + 2 = 3$$

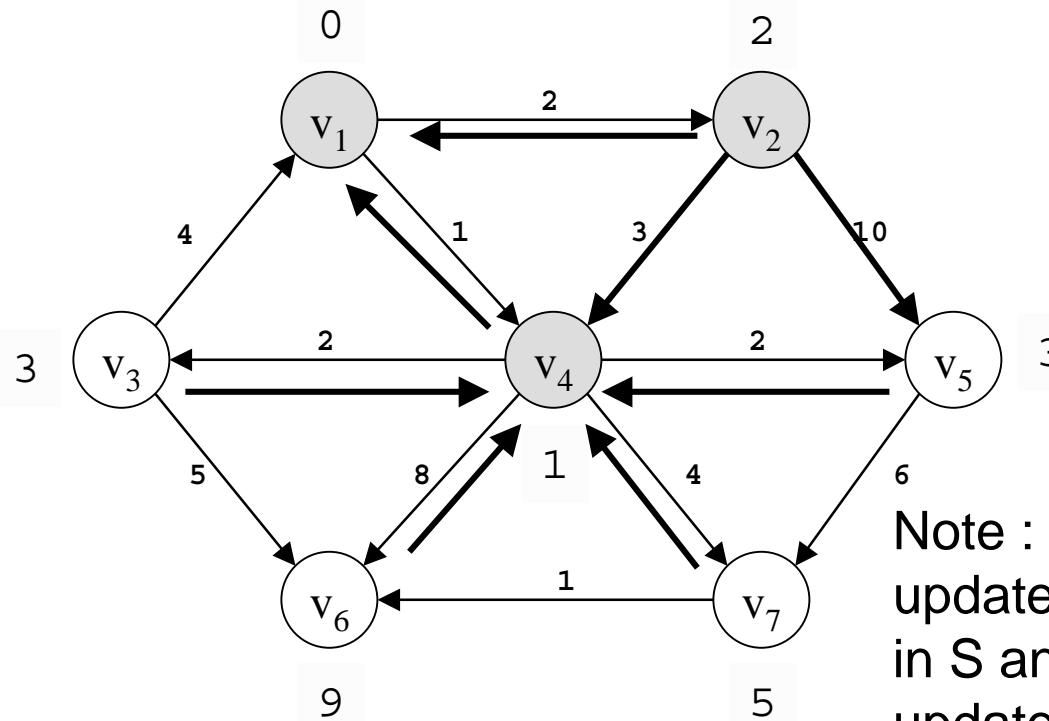
$$\text{Cost}(v_5) = 1 + 2 = 3$$

$$\text{Cost}(v_6) = 1 + 8 = 9$$

$$\text{Cost}(v_7) = 1 + 4 = 5$$

# Example (Ct'd)

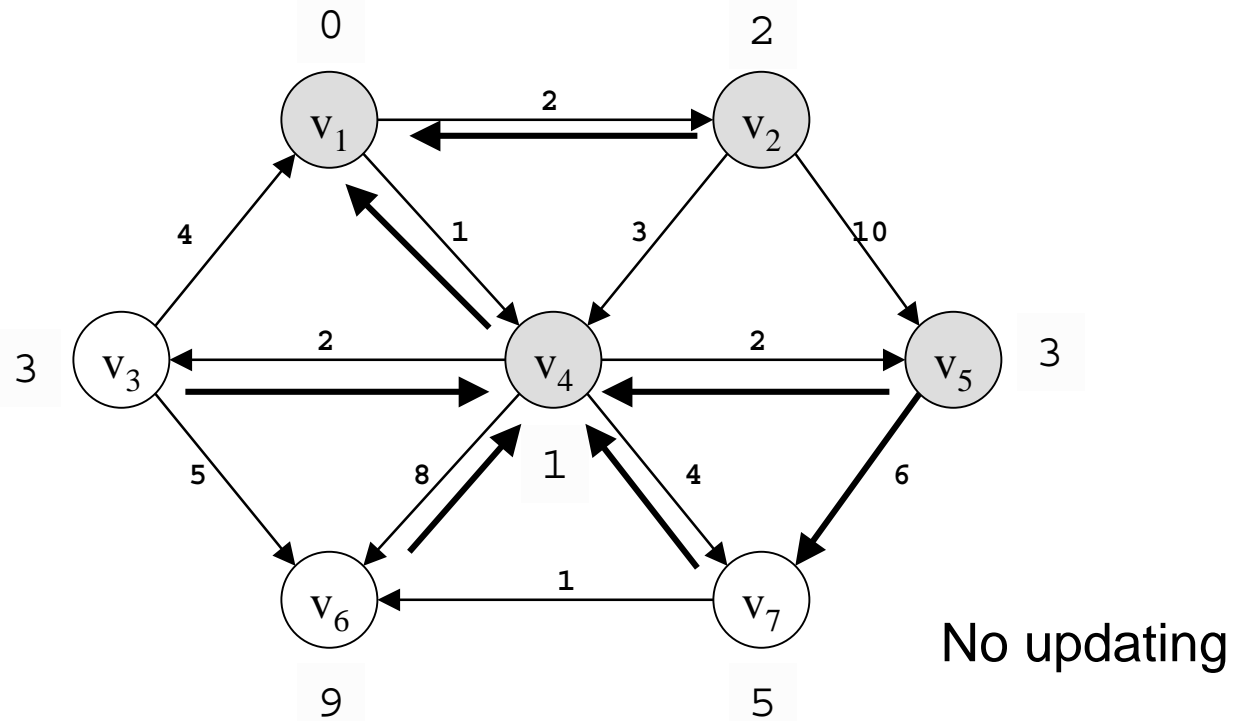
Pick vertex not in S with lowest cost ( $v_2$ ) and update neighbors



Note :  $\text{cost}(v_4)$  not updated since already in S and  $\text{cost}(v_5)$  not updated since it is larger than previously computed

# Example: (ct'd)

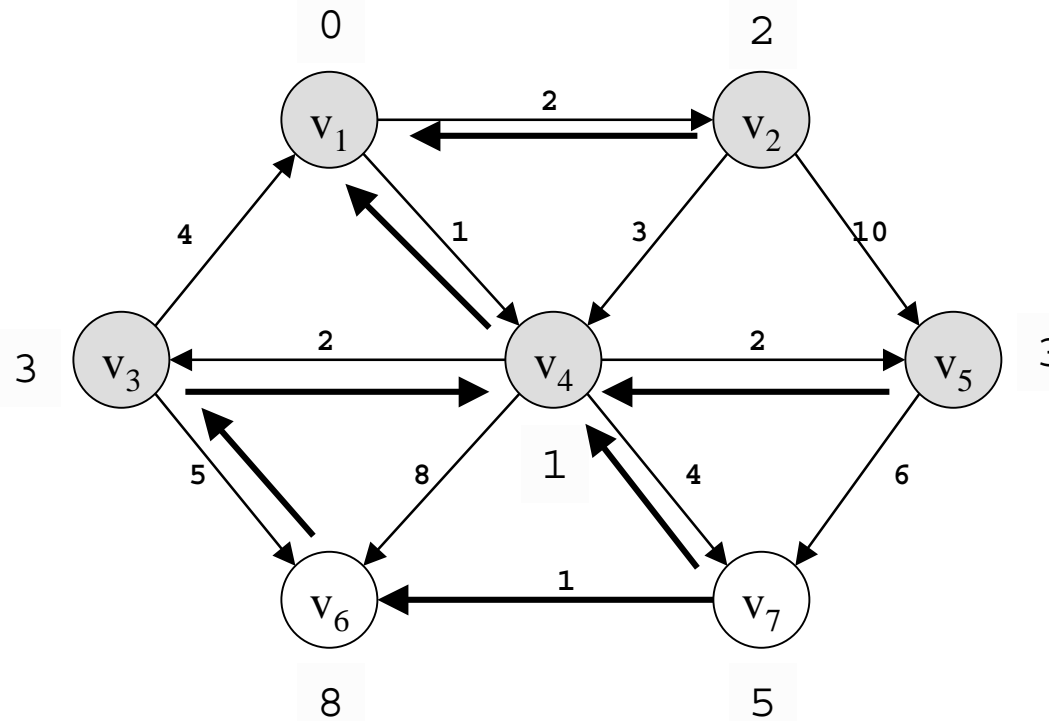
Pick vertex not in  $S$  ( $v_5$ ) with lowest cost and update neighbors





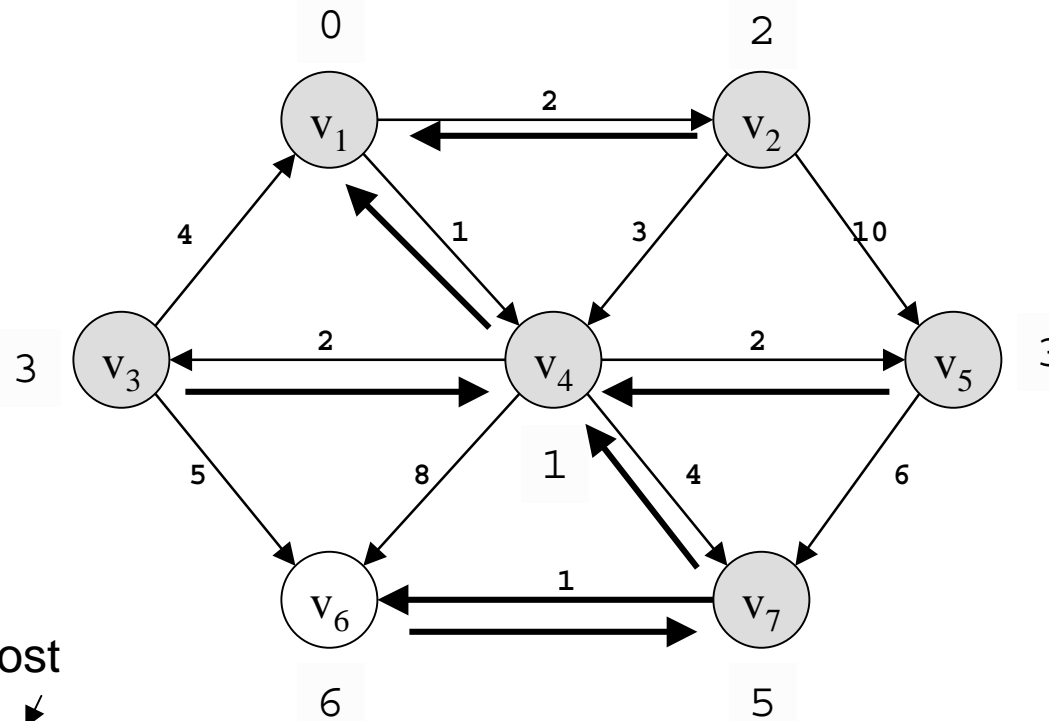
# Example: (ct'd)

Pick vertex not in S with lowest cost ( $v_7$ ) and update neighbors



# Example: (ct'd)

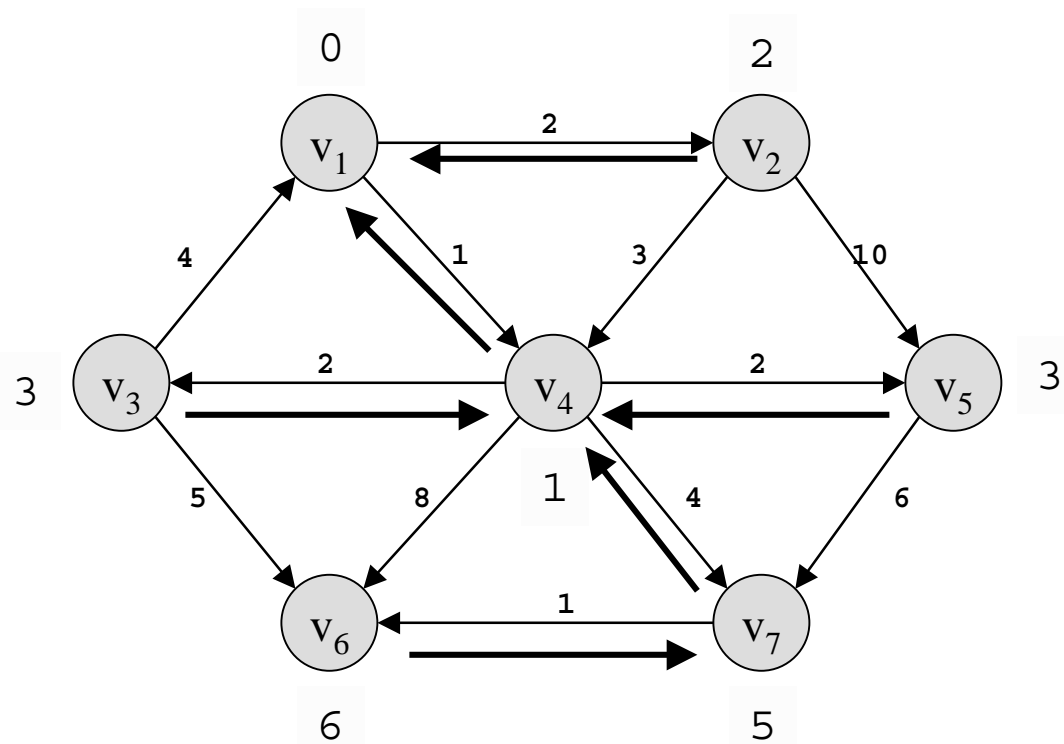
Pick vertex not in S with lowest cost ( $v_7$ ) and update neighbors



Previous cost  
↙  
 $\text{Cost}(v_6) = \min(8, 5+1) = 6$

# Example (end)

---



Pick vertex not in  $S$  with lowest cost ( $v_6$ ) and update neighbors

# Data Structures

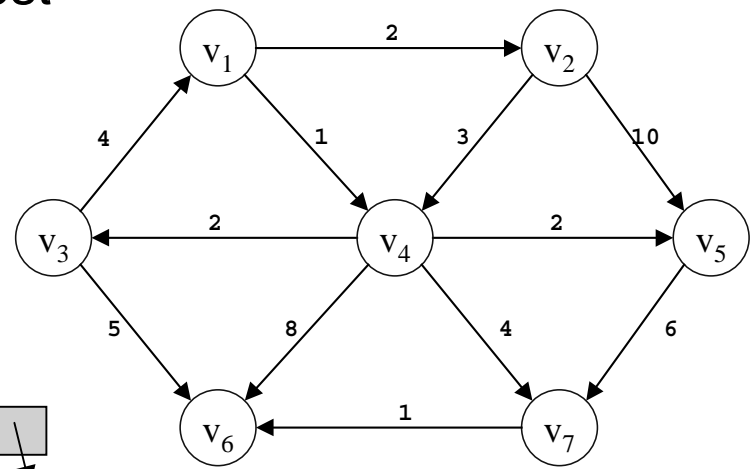
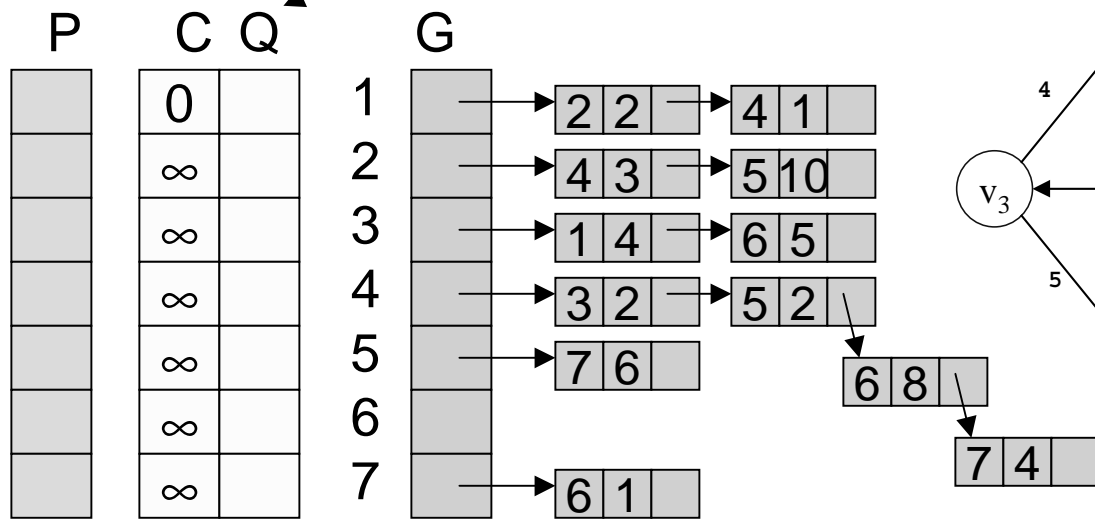
- Adjacency Lists

previous cost priority queue pointers

adj 

--	--	--

 next  
cost



Priority queue for finding and deleting lowest cost vertex and for decreasing costs (Binary Heap works)

# Time Complexity

---

- $n$  vertices and  $m$  edges
- Initialize data structures  $O(n+m)$
- Find min cost vertices  $O(n \log n)$ 
  - ›  $n$  delete mins
- Update costs  $O(m \log n)$ 
  - › Potentially  $m$  updates
- Update previous pointers  $O(m)$ 
  - › Potentially  $m$  updates
- Total time  $O((n + m) \log n)$  - very fast.

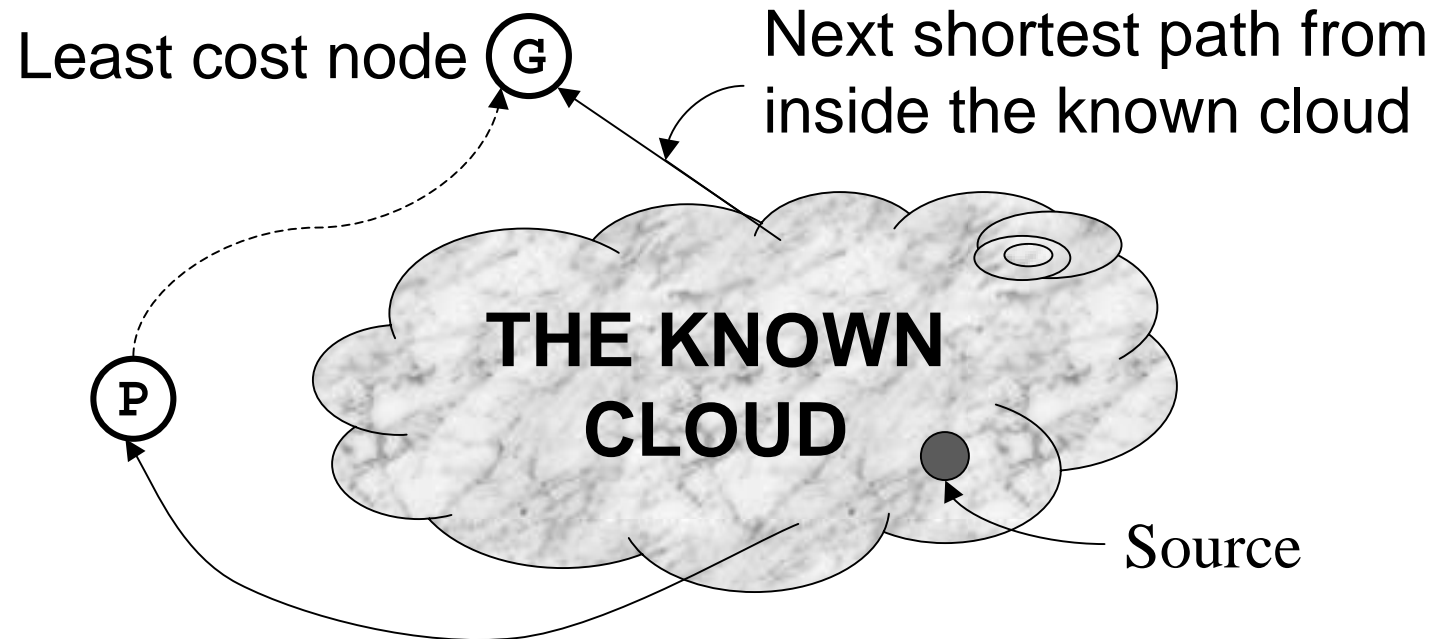
# Correctness

---

- Dijkstra's algorithm is an example of a greedy algorithm
- Greedy algorithms always make choices that currently seem the best
  - › Short-sighted – no consideration of long-term or global issues
  - › Locally optimal does not always mean globally optimal
- In Dijkstra's case – choose the least cost node, but what if there is another path through other vertices that is cheaper?

# “Cloudy” Proof: The Idea

---



- If the path to **G** is the next shortest path, the path to **P** must be at least as long. Therefore, any path through **P** to **G** cannot be shorter!

# Inside the Cloud (Proof)

---

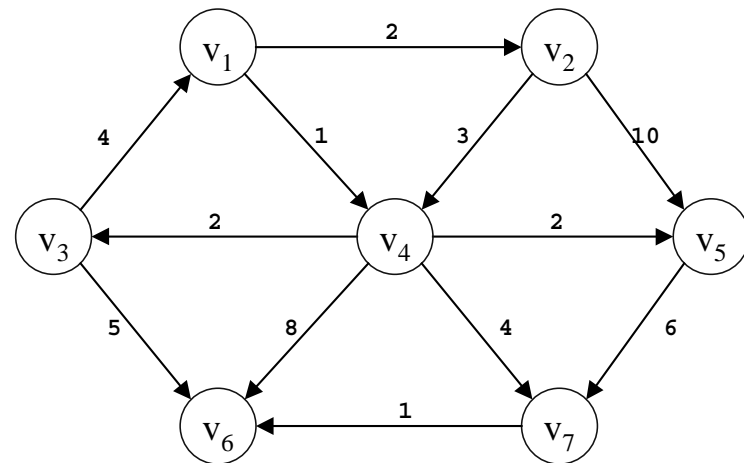
- Everything inside the cloud has the correct shortest path
- Proof is by induction on the number of nodes in the cloud:
  - › Base case: Initial cloud is just the source with shortest path 0
  - › Inductive hypothesis: cloud of  $k-1$  nodes all have shortest paths
  - › Inductive step: choose the least cost node  $G \rightarrow$  has to be the shortest path to  $G$  (previous slide). Add  $k$ -th node  $G$  to the cloud



# All Pairs Shortest Path

- Given a edge weighted directed graph  $G = (V, E)$  find for all  $u, v$  in  $V$  the length of the shortest path from  $u$  to  $v$ . Use matrix representation.

<b>C</b>	1	2	3	4	5	6	7
1	0	2	:	1	:	:	:
2	:	0	:	3	10	:	:
3	4	:	0	:	:	5	:
4	:	:	2	0	2	8	4
5	:	:	:	:	0	:	6
6	:	:	:	:	:	0	:
7	:	:	:	:	:	1	0



# A (simpler) Related Problem: Transitive Closure

---

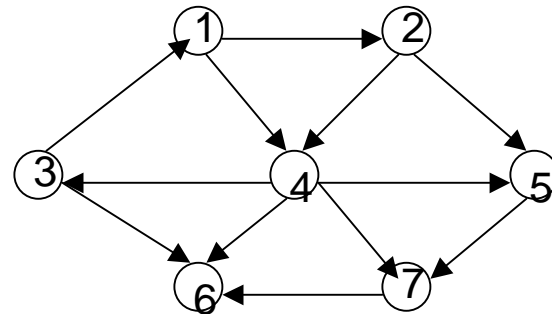
- Given a digraph  $G(V, E)$  the transitive closure is a digraph  $G'(V', E')$  such that
  - ›  $V' = V$  (same set of vertices)
  - › If  $(v_i, v_{i+1}, \dots, v_k)$  is a path in  $G$ , then  $(v_i, v_k)$  is an edge of  $E'$

# Unweighted Digraph Boolean Matrix Representation

- C is called the connectivity matrix

1 = connected  
0 = not connected

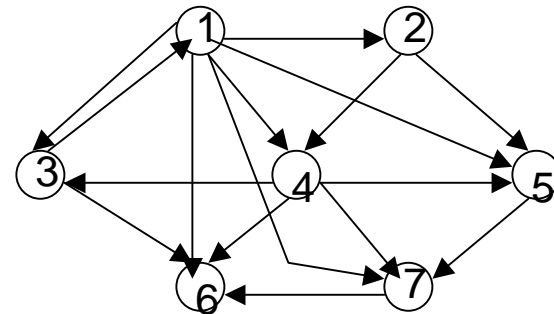
C	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	0	0	0	1	1	0	0
3	1	0	0	0	0	1	0
4	0	0	1	0	1	1	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



# Transitive Closure

---

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



On the graph, we show only the edges added with 1 as origin. The matrix represents the full transitive closure.

# Finding Paths of Length 2

---

```
// First initialize C2 to all zero //
```

```
Length2 {
```

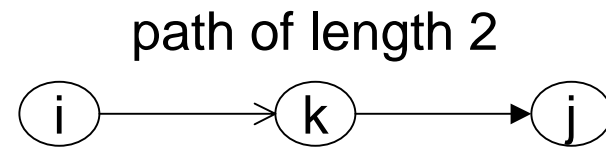
```
for k = 1 to n
```

```
  for i = 1 to n do
```

```
    for j = 1 to n do
```

```
      C2[i,j] := C2[i,j]  $\cup$  (C[i,k]  $\cap$  C[k,j]);
```

```
}
```



where  $\cap$  is Boolean And (&&) and  $\cup$  is Boolean OR (||)

This means if there is an edge from  $i$  to  $k$

AND an edge from  $k$  to  $j$ , then there is a path of length 2 between  $i$  and  $j$ .

Column  $k$  ( $C[i,k]$ ) represents the predecessors of  $k$

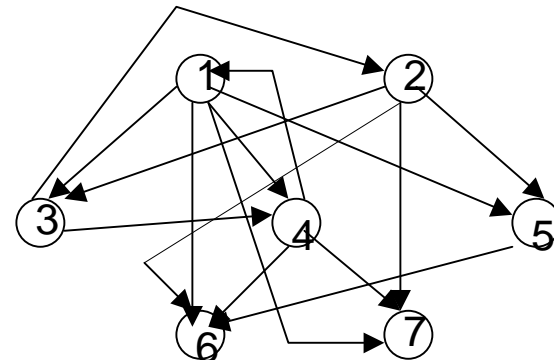
Row  $k$  ( $C[k,j]$ ) represents the successors of  $k$

# Paths of Length 2

**C**

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

Time  $O(n^3)$



**C2**

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

5/10/03

# Transitive Closure

---

- Union of paths of length 0, length 1, length 2, ..., length  $n-1$ .
  - › Time complexity  $n * O(n^3) = O(n^4)$
- There exists a better ( $O(n^3)$ ) algorithm: Warshall's algorithm

# Warshall Algorithm

---

```
TransitiveClosure {
for k = 1 to n do // k is the step number //
  for i = 1 to n do
    for j = 1 to n do
      C [i,j] := C[i,j]  $\cup$  (C[i,k]  $\cap$  C[k,j]);
}
```

where  $C[i,j]$  starts as the original connectivity matrix and  $C[i,j]$  is updated after step  $k$  if a new path from  $i$  to  $j$  through  $k$  is found.



# Proof of Correctness

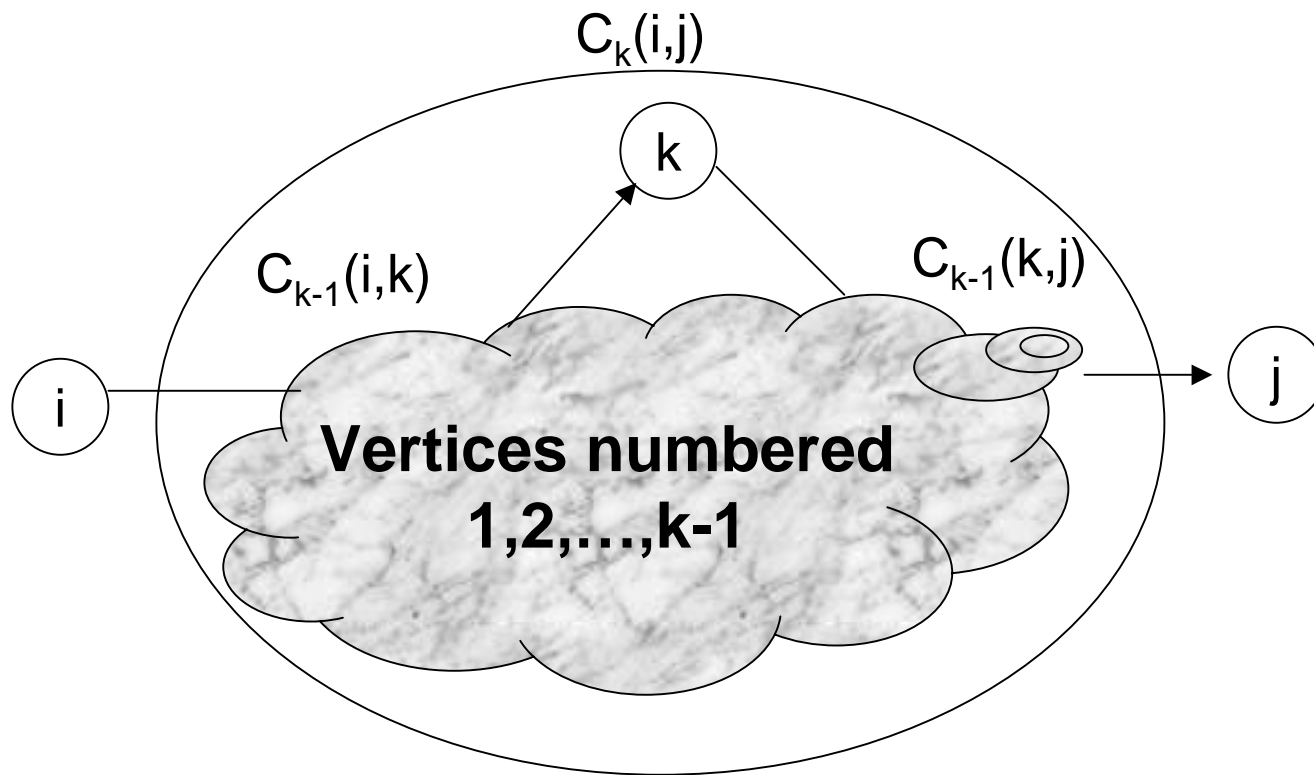
---

Prove: After the  $k$ -th time through the loop,  $C[i,j] = 1$  if there is a path from  $i$  to  $j$  that only passes through vertices numbered  $1, 2, \dots, k$  (except for the initial edges)

- Base case:  $k = 1$ .  $C[i,j] = 1$  for the initial connectivity matrix (path of length 0) and  $C[i,j] = 1$  if there is a path  $(i, 1, j)$

# Cloud Argument

---



# Inductive Step

---

- Inductive Hypothesis: Suppose after step  $k-1$  that  $C[i,j]$  contains a 1 if there is a path **from**  $i$  to  $j$  through vertices  $1, \dots, k-1$ .

- Induction: Consider step  $k$ , which does

$$C[i,j] := C[i,j] \cup (C[i,k] \cap C[k,j]);$$

Either  $C[i,j]$  is already 1 or there is a new path through vertex  $k$ , which makes it 1.

# Back to Weighted graphs: Matrix Representation

---

- $C[i,j]$  = the cost of the edge  $(i,j)$ 
  - ›  $C[i,i] = 0$  because no cost to stay where you are
  - ›  $C[i,j] = \text{infinity } (:)$  if no edge from  $i$  to  $j$ .

$$C \begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \left( \begin{array}{cccccc} 0 & 2 & : & 1 & : & : & : \\ : & 0 & : & 3 & 10 & : & : \\ 4 & : & 0 & : & : & 5 & : \\ : & : & 2 & 0 & 2 & 8 & 4 \\ : & : & : & : & 0 & : & 6 \\ : & : & : & : & : & 0 & : \\ : & : & : & : & : & 1 & 0 \end{array} \right) \end{matrix}$$

# Floyd – Warshall Algorithm

---

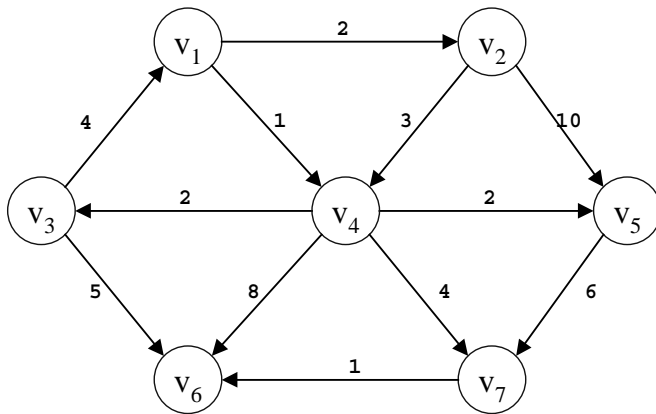
```
All_Pairs_Shortest_Path {  
  for k = 1 to n do  
    for i = 1 to n do  
      for j = 1 to n do  
        C[i,j] := min(C[i,j], C[i,k] + C[k,j]);  
      }  
}
```

Note  $x + \infty = \infty$  by definition ( $\infty$  is infinity)

On termination  $C[i,j]$  is the length of the shortest path from  $i$  to  $j$ .

# The Computation

$$\begin{array}{c} \mathbf{C} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \end{array} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \left( \begin{array}{cccccc} 0 & 2 & : & 1 & : & : & : \\ : & 0 & : & 3 & 10 & : & : \\ 4 & : & 0 & : & : & 5 & : \\ : & : & 2 & 0 & 2 & 8 & 4 \\ : & : & : & : & 0 & : & 6 \\ : & : & : & : & : & 0 & : \\ : & : & : & : & : & 1 & 0 \end{array} \right)
 \end{array}$$


$$\begin{array}{c} \mathbf{C} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \end{array} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \left( \begin{array}{cccccc} 0 & 2 & 3 & 1 & 3 & 6 & 5 \\ 9 & 0 & 5 & 3 & 5 & 8 & 7 \\ 4 & 6 & 0 & 5 & 4 & 5 & 6 \\ 6 & 8 & 2 & 0 & 2 & 5 & 4 \\ : & : & : & : & 0 & 7 & 6 \\ : & : & : & : & : & 0 & : \\ : & : & : & : & : & 1 & 0 \end{array} \right)
 \end{array}$$


# Time Complexity of All Pairs Shortest Path

---

- $n$  is the number of vertices
- Three nested loops.  $O(n^3)$ 
  - › Shortest paths can be found too (see the book).
- Repeated Dijkstra's algorithm
  - ›  $O(n(n+m)\log n)$  ( $= O(n^3 \log n)$  for dense graphs).
  - › Run Dijkstra starting at each vertex.
  - › But, Dijkstra also gives the shortest paths not just their lengths.