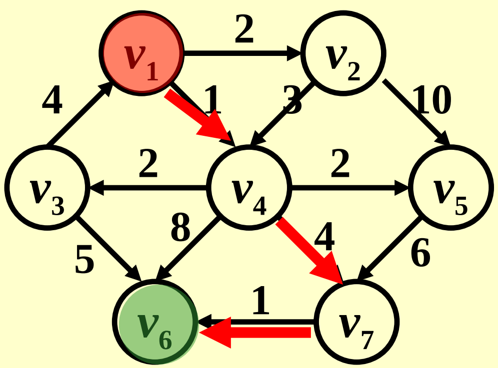
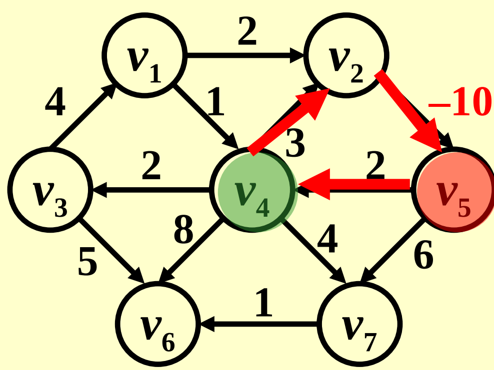
# §3 Shortest Path Algorithms

Given a digraph G = ( V, E ), and a cost function $c( e )$ for $e \in$ E( G ). The **length** of a path $P$ from **source** to **destination** is $\sum_{e_i \subset P} c(e_i)$ (also called **weighted path length**).

## 1. 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.

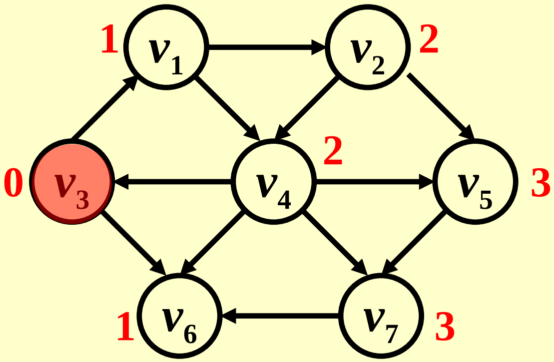



**Note:** If there is no negative-cost cycle, the shortest path **from $s$ to $s$** is defined to be **zero**.

# ➢ Unweighted Shortest Paths

## ❖ Sketch of the idea




0: ☝ $v_3$

1: ▯ $v_1$ and $v_6$

2: ▯ $v_2$ and $v_4$

3: ▯ $v_5$ and $v_7$

**Breadth-first search**

## ❖ Implementation

**Table[ i ].Dist ::= distance from *s* to $v_i$ /* initialized to be ∞ except for *s* */**

**Table[ i ].Known ::= 1 if $v_i$ is checked; or 0 if not**

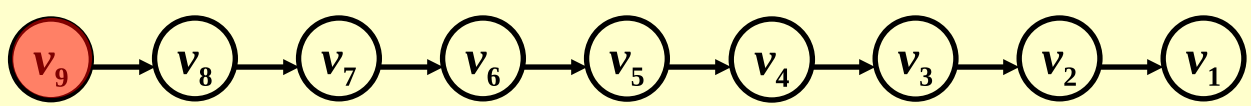
**Table[ i ].Path ::= for tracking the path /* initialized to be 0 */**

```
void Unweighted( Table T )
{   int  CurrDist;
    Vertex  V, W;
    for ( CurrDist = 0; CurrDist < NumVertex; CurrDist ++ ) {
        for ( each vertex V )
            if ( !T[ V ].Known && T[ V ].Dist == CurrDist ) {
                T[ V ].Known = true;
                for ( each W adjacent to V )
                    if ( T[ W ].Dist == Infinity ) {
            T[ W ].Dist = CurrDist + 1;
            T[ W ].Path = V;
                    } /* end-if Dist == Infinity */
            } /* end-if !Known && Dist == CurrDist */
    }  /* end-for CurrDist */
}
```

If V is unknown yet has Dist < Infinity, then Dist is either CurrDist or CurrDist+1.

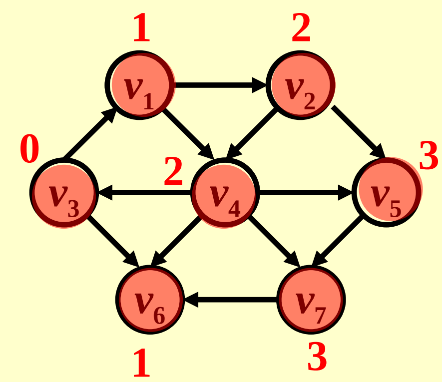
$T = O( |V|^2 )$

**The worst case:**



## ❖ Improvement

```
void Unweighted( Table T )
{   /* T is initialized with the source vertex S given */
    Queue  Q;
    Vertex  V, W;
    Q = CreateQueue (NumVertex );  MakeEmpty( Q );
    Enqueue( S, Q ); /* Enqueue the source vertex */
    while ( !IsEmpty( Q ) ) {
        V = Dequeue( Q );
        T[ V ].Known = true; /* not really necessary */
        for ( each W adjacent to V )
            if ( T[ W ].Dist == Infinity ) {
                T[ W ].Dist = T[ V ].Dist + 1;
                T[ W ].Path = V;
                Enqueue( W, Q );
            } /* end-if Dist == Infinity */
    } /* end-while */
    DisposeQueue( Q ); /* free memory */
}
```




| | Dist | Path |
|---|---|---|
| $v_1$ | 1 | $v_3$ |
| $v_2$ | 2 | $v_1$ |
| $v_3$ | 0 | 0 |
| $v_4$ | 2 | $v_1$ |
| $v_5$ | 3 | $v_2$ |
| $v_6$ | 1 | $v_3$ |
| $v_7$ | 3 | $v_4$ |

$T = O( |V| + |E| )$

## ➤ *Dijkstra'*s Algorithm (for weighted shortest paths)

Let $S$ = { $s$ and $v_i$'s whose shortest paths have been found }

For any $u \notin S$, define distance [ u ] = minimal length of path { $s \rightarrow ( v_i \in S ) \rightarrow u$ }. If the paths are generated in non-decreasing order, then

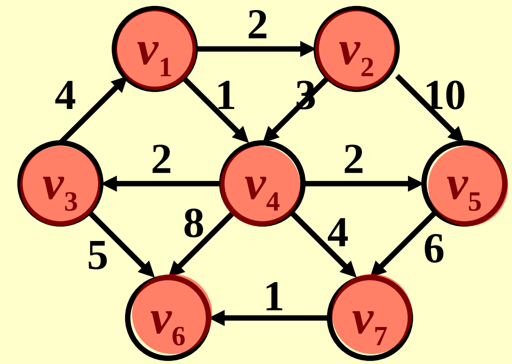
① the shortest path must go through ONLY $v_i \in S$ ;

② $u$ is chosen so that distance[ u ] = min{ $w \notin S$ | distance[ w ] } (If $u$ is not unique, then we may select any of them) ; /* Greedy Method */

Why? If it is not true, then there must be a vertex $w$ on this path that is not in $S$. Then...

③ if distance [ $u_1$ ] < distance [ $u_2$ ] and we add $u_1$ into $S$, then distance [ $u_2$ ] may change. If so, a shorter path from $s$ to $u_2$ must go through $u_1$ and distance' [ $u_2$ ] = distance [ $u_1$ ] + length(< $u_1$, $u_2$>).

```
void Dijkstra( Table T )
{   /* T is initialized by Figure 9.30 on p.303 */
    Vertex  V, W;
    for ( ; ; ) {     /* O( |V| ) */
        V = smallest unknown distance vertex;
        if ( V == NotAVertex )
                break;
        T[ V ].Known = true;
        for ( each W adjacent to V )
                if ( !T[ W ].Known )
                    if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {
                            Decrease( T[ W ].Dist  to
                 T[ V ].Dist + Cvw );
                T[ W ].Path = V;
                    } /* end-if update W */
    } /* end-for( ; ; ) */
}
    /* not work for edge with negative cost */
```




| | Dist | Path |
|---|---|---|
| $v_1$ | 0 | 0 |
| $v_2$ | 2 | $v_1$ |
| $v_3$ | 3 | $v_4$ |
| $v_4$ | 1 | $v_1$ |
| $v_5$ | 3 | $v_4$ |
| $v_6$ | 6 | $v_7$ |
| $v_7$ | 5 | $v_4$ |

**Please read Figure 9.31 on p.304 for printing the path.**

## ❖ Implementation 1

**V = smallest unknown distance vertex;**
**/* simply scan the table – O( |V| ) */**

$T = O(\ |V|^2 + |E|\ )$

Good if the graph is dense

## ❖ Implementation 2

**V = smallest unknown distance vertex;**
**/* keep distances in a priority queue and call DeleteMin – O( log|V| ) */**

**Decrease( T[ W ].Dist to T[ V ].Dist + Cvw );**

Good if the graph is sparse

**/* Method 1: DecreaseKey – O( log|V| ) */**

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

**/* Method 2: insert W with updated Dist into the priority queue */**

**/* Must keep doing DeleteMin until an unknown vertex emerges */**

$T = O(\ |E| \log|V|\ )$ but requires |E| DeleteMin with |E| space

## ❖ Other improvements: Pairing heap (Ch.12) and Fibonacci heap (Ch. 11)

## Graphs with Negative Edge Costs

$T = O( |V| \times |E| )$

```
void  WeightedNegative( Table T )
{   /* T is initialized by Figure 9.30 on p.303 */
    Queue  Q;
    Vertex  V, W;
    Q = CreateQueue (NumVertex );  MakeEmpty( Q );
    Enqueue( S, Q ); /* Enqueue the source vertex */
    while ( !IsEmpty( Q ) ) {   /* each vertex can dequeue at most |V| times */
        V = Dequeue( Q );
        for ( each W adjacent to V )
            if ( T[ V ].Dist + Cvw < T[ W ].Dist ) {   /* no longer once per edge */
                T[ W ].Dist = T[ V ].Dist + Cvw;
                T[ W ].Path = V;
                if ( W is not already in Q )
                    Enqueue( W, Q );
            } /* end-if update */
    } /* end-while */
    DisposeQueue( Q ); /* free memory */
}
  /* negative-cost cycle will cause indefinite loop */
```

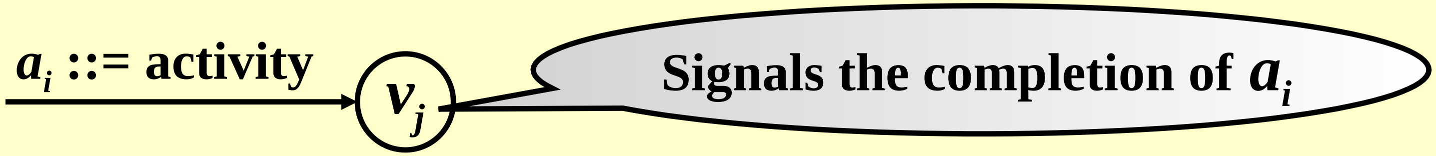
# Acyclic Graphs

If the graph is acyclic, vertices may be selected in topological order since when a vertex is selected, its distance can no longer be lowered without any incoming edges from unknown nodes.

$T$ = O( |E| + |V| ) and no priority queue is needed.

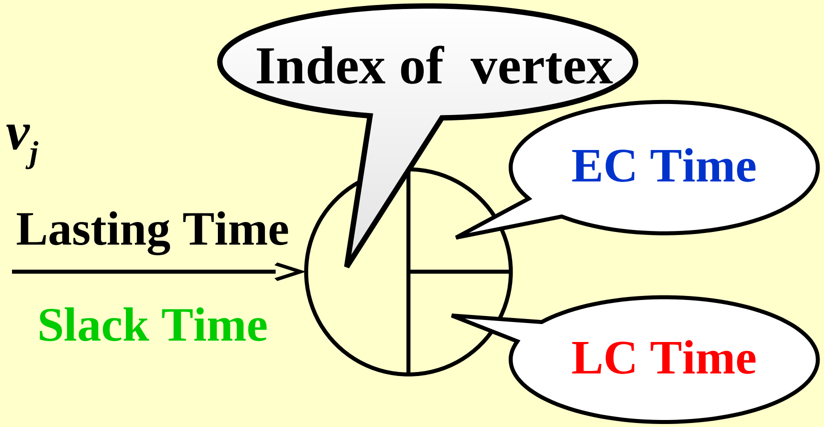
- Application: AOE ( Activity On Edge ) Networks

  —— scheduling a project

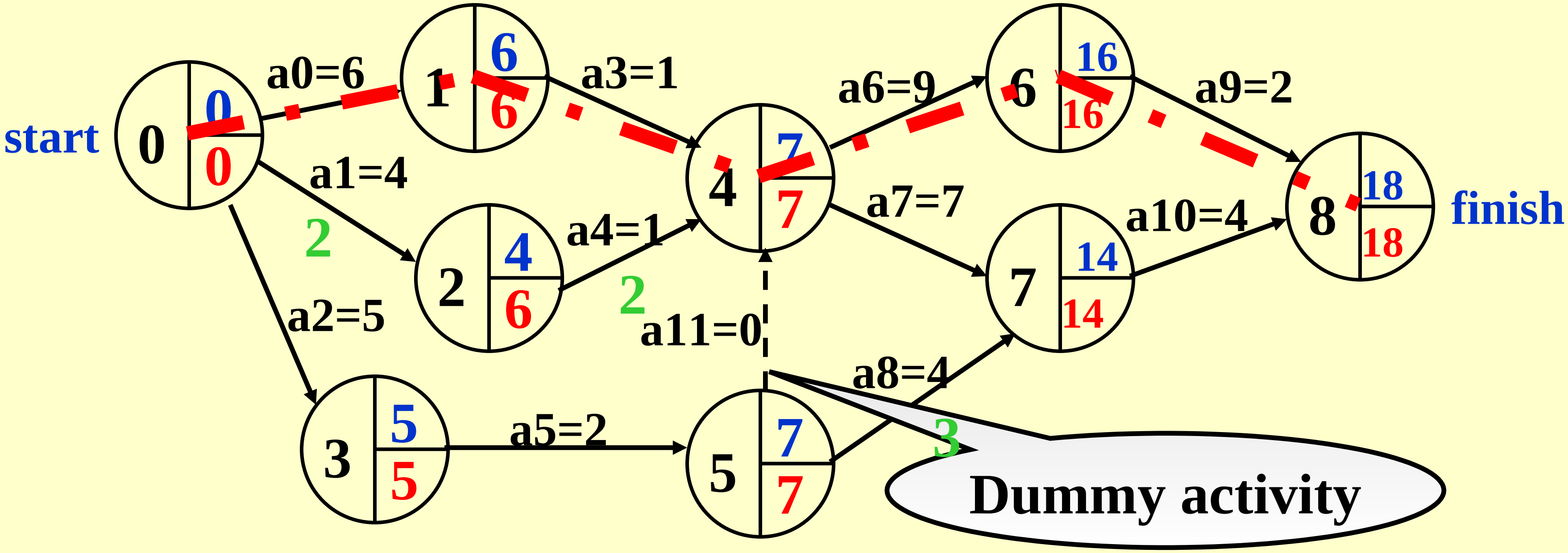



- EC[ $j$ ] \ LC[ $j$ ] ::= the earliest \ latest completion time for node $v_j$

- CPM ( Critical Path Method )



〖Example〗 AOE network of a hypothetical project




- **Calculation of EC: Start from v0, for any $a_i$ = <v, w>, we have**

$$EC[w] = \max_{(v,w)\in E}\{EC[v] + C_{v,w}\}$$

- **Calculation of LC: Start from the last vertex v8, for any $a_i$ = <v, w>, we have** $LC[v] = \min_{(v,w)\in E}\{LC[w] - C_{v,w}\}$

- **Slack Time of <v,w> =** $LC[w] - EC[v] - C_{v,w}$

- **Critical Path ::= path consisting entirely of zero-slack edges.**

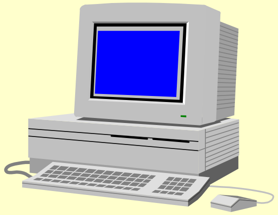
## 2. All-Pairs Shortest Path Problem

**For all pairs of $v_i$ and $v_j$ ( $i \neq j$ ), find the shortest path between.**

Method 1 **Use** single-source algorithm **for |V| times.**

**$T$ = O( |V|$^3$ ) – works fast on sparse graph.**

Method 2 **O( |V|$^3$ ) algorithm given in Ch.10, works faster on dense graphs.**



# Laboratory Project 3

**Normal: Ambulance Dispatch**

***Hard: The 2nd-shortest Path***

**Due: Monday, November 22nd, 2021 at 10:00pm**