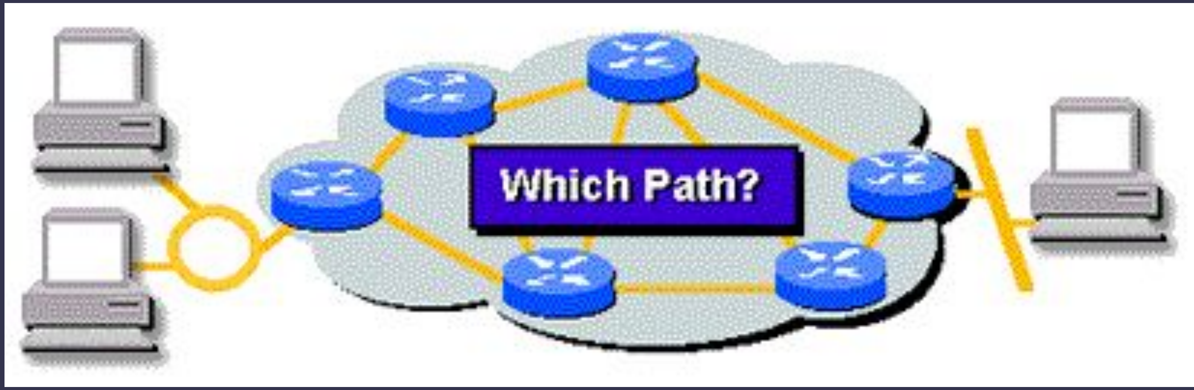
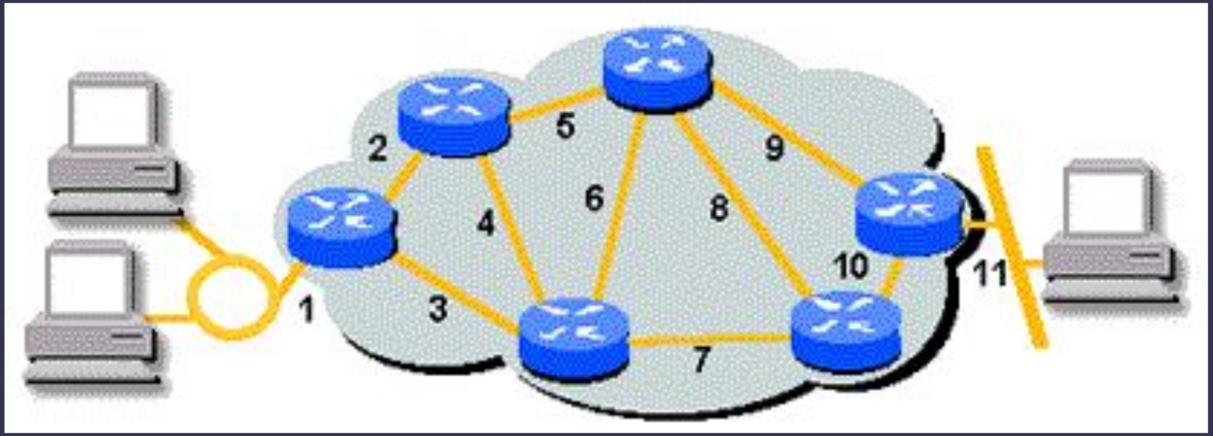
Shortest Paths (1/11)

- If there is more than one path from A to B, which path is the shortest?



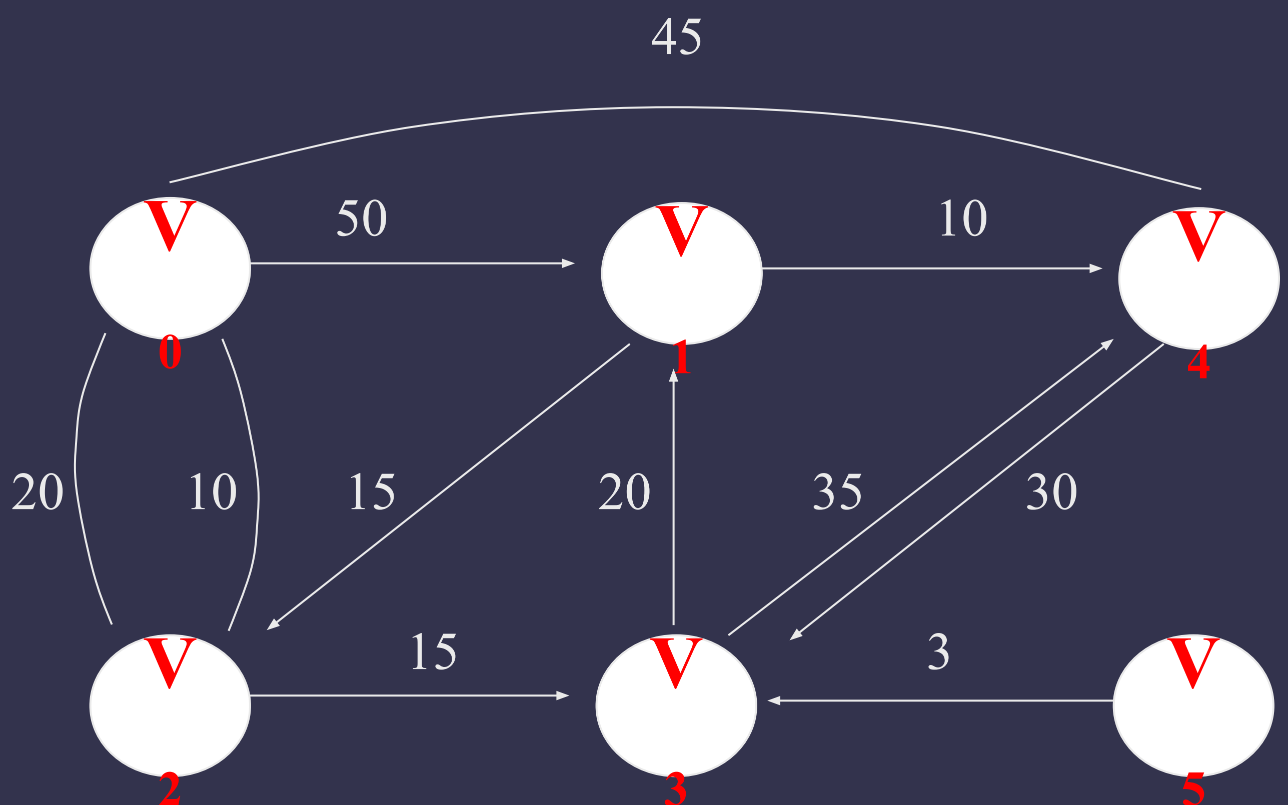


Shortest Paths (2/11)

- Single source/All destinations: nonnegative edge cost
  - **Problem:** given a directed graph $G = (V, E)$, a length function $length(i, j)$, $length(i, j) \geq 0$, for the edges of G, and a source vertex v.
  - **Need to solve**: determine a shortest path from v to each of the remaining vertices of G.

Shortest Paths (2/11)

- Single source/All destinations: nonnegative edge cost



Shortest Paths (2/11)

- Single source/All destinations: nonnegative edge cost
  - Idea:
    - Let S denote the set of vertices
    - ***dist*[*w*]:** the length of shortest path starting from *v*, going through only the vertices that are in *S*, ending at *w*.

Shortest Paths (3/11)

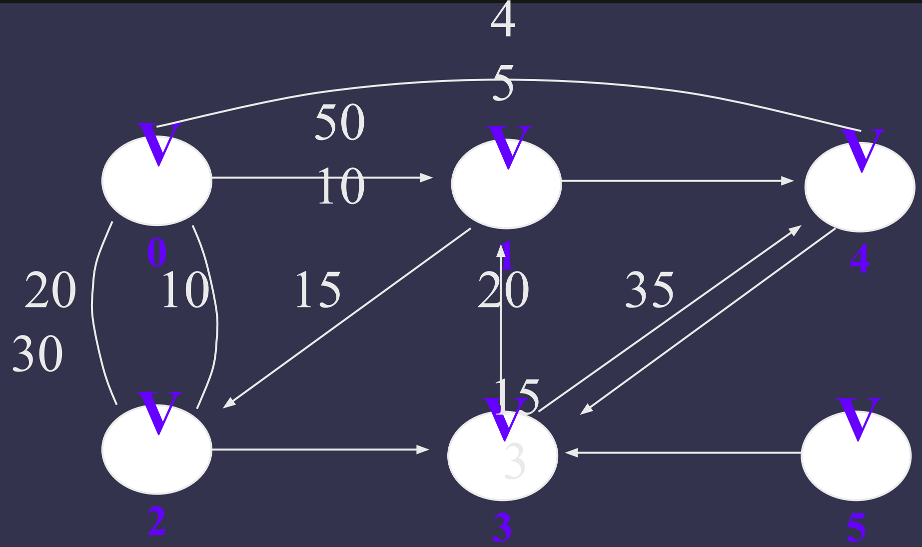
- Dijkastra's Algorithm

$S \leftarrow \{v_0\}$;
$dist[v_0] \leftarrow 0$;
for each v in V - $\{v_0\}$ do $dist[v] \leftarrow e(v_0, v)$;
while $S \neq V$ do
    choose a vertex w in V-S such that $dist[w]$ is a minimum;
    add w to S;
    for each v in V-S do
        $dist[v] \leftarrow \min(dist[v], dist[w]+e(w, v))$;

$O(N^2)$

- Declarations for the Shortest Path Algorithm

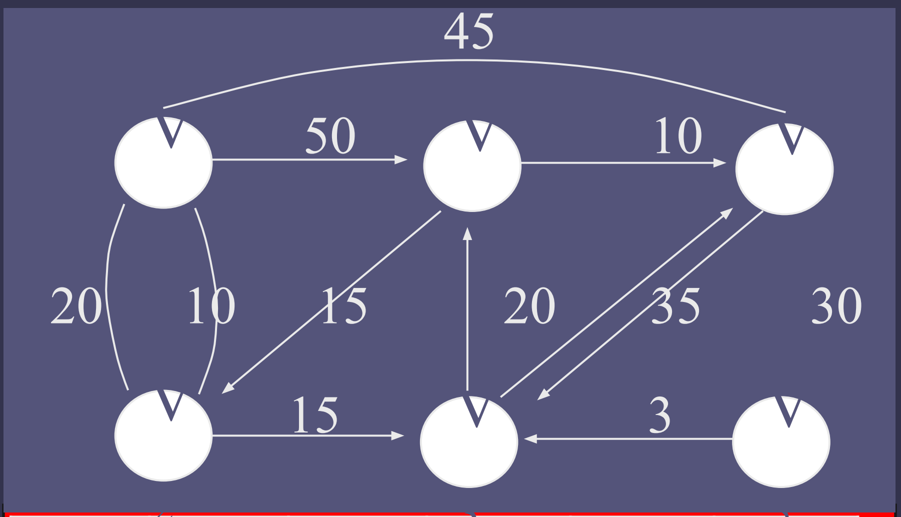
```
#define MAX_VERTICES 6
int cost[ ][MAX_VERTICES]=
{  {    0,    50,    10, 1000,    45, 1000},
    {1000,     0,    15, 1000,    10, 1000},
    {  20, 1000,     0,    15, 1000, 1000},
    {1000,    20, 1000,     0,    35, 1000},
    {1000, 1000,    30, 1000,     0, 1000},
    {1000, 1000, 1000,     3, 1000,     0}};

int dist[MAX_VERTICES];
short int found{MAX_VERTICES];
int n = MAX_VERTICES;
```



- Single Source Shortest Paths Program (v=0)

visited

[0]	[1]	[2]	[3]	[4]	[5]



dist:

0	50 45	10	1000 25	45	1000

Shortest Paths (5/11)

- Choosing the least cost edge

```
int choose(int distance[], int n, short int found[])
{
/* find the smallest distance not yet checked */
   int i, min, minpos;
   min = INT_MAX;
   minpos = -1;
   for (i = 0; i < n; i++)
      if (distance[i] < min && !found[i]) {
         min = distance[i];
         minpos = i;
      }
   return minpos;
}
```

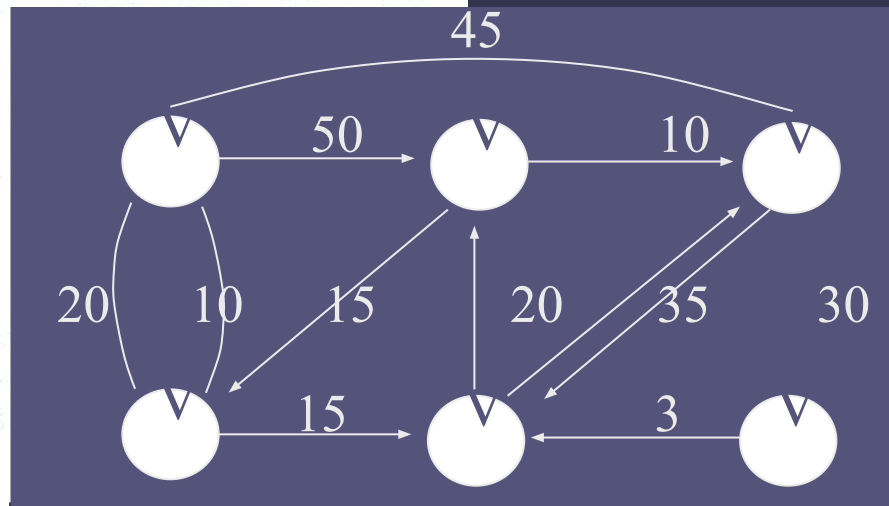
▪ Single Source Shortest Paths Program (v=0)

```
void shortestpath(int v, int cost[][MAX_VERTICES],
int distance[], int n, short int found[])
{
  int i,u,w;
  for (i = 0; i < n; i++) {
    found[i] = FALSE;
    distance[i] = cost[v][i];
  }
  found[v] = TRUE;
  distance[v] = 0;
  for (i = 0; i < n-2; i++) {
    u = choose(distance,n,found);
    found[u] = TRUE;
    for (w = 0; w < n; w++)
      if (!found[w])
        if (distance[u] + cost[u][w] < distance[w])
          distance[w] = distance[u] + cost[u][w];
  }
}
```

Program 6.10: Single source shortest paths

visited

[0]	[1]	[2]	[3]	[4]	[5]



dist:

0	45	10	25	45	1000

i

u

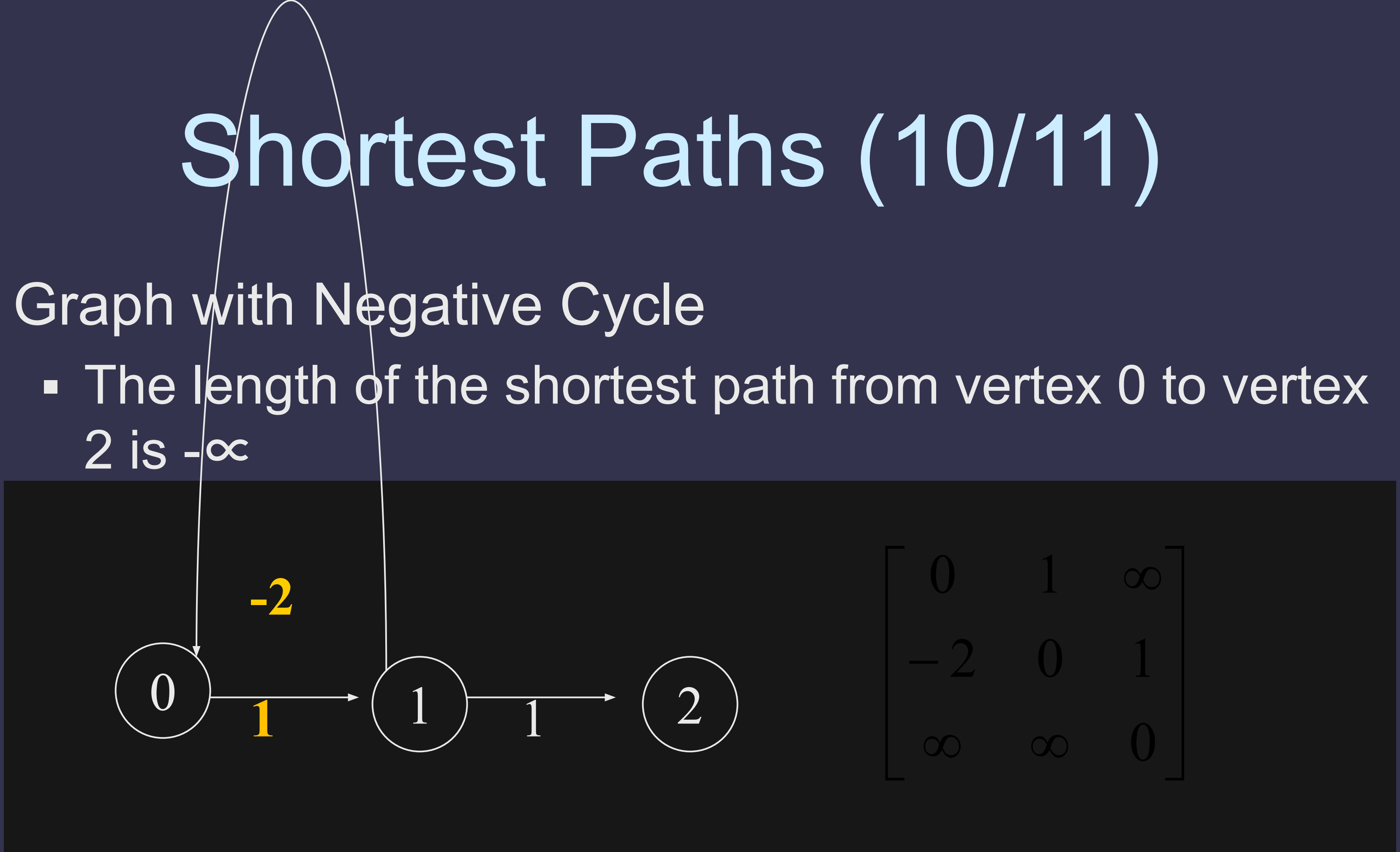
w

Shortest Paths (7/11)

- All Pairs Shortest Paths
  - we could solve this problem using *shortestpath* with each of the vertices in $V(G)$ as the source. (O(n^3))
  - we can obtain a conceptually simpler algorithm that works correctly even if some edges in G have negative weights, require G has no cycles with a negative length (still O(n^3))

Shortest Paths (10/11)

- Graph with Negative Cycle
  - The length of the shortest path from vertex 0 to vertex 2 is -∞



(a) Directed graph (b) cost

0, 1, 0, 1, 0, 1, …, 0, 1, 2

Shortest Paths (8/11)

- Another solution
  - Use dynamic programming method.
  - Let $A^k[i][j]$ be the cost of shortest path from i to j, using only those intermediate vertices with an index $\leq k$.
  - The shortest path from i to j is $A^{n-1}[i][j]$ as no vertex in G has an index greater than n - 1.
  - Represent the graph G by its cost adjacency matrix with cost[i][j].
    - If $i = j$, $cost[i][j] = 0$.
    - If <i, j> is not in G, $cost[i][j]$ is set to some sufficiently large number.

Shortest Paths (9/11)

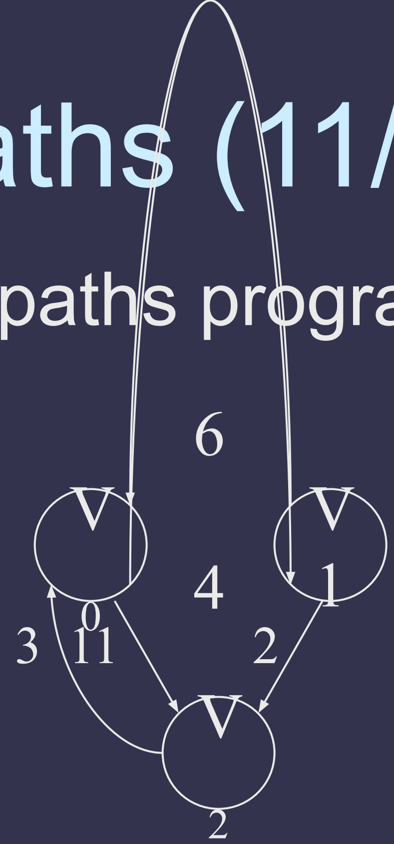
- Algorithm concept
  - The basic idea in the all pairs algorithm is begin with the matrix A^{-1} and successively generated the matrices A^{-1}, A^{0}, A^{2}, …, A^{n}
  - $A^{-1}[i][j]=cost[i][j]$
  - Calculate the A^0, A^1, A^2, ..., A^{n-1} from A^{-1} iteratively
  - $A^k[i][j]=\min\{A^{k-1}[i][j], A^{k-1}[i][k]+A^{k-1}[k][j]\}$, $k>=0$

Shortest Paths (11/11)

- All pairs shortest paths program

cost:

0	4	11
6	0	2
3	1000	0



final distance:

0	4	11
6	0	2
3	7	0

K 0

I 2

J 1

dist[i][j] < dist[i][k] + dist[k][j] ????

Shortest Paths (11/11)

- All pairs shortest paths program

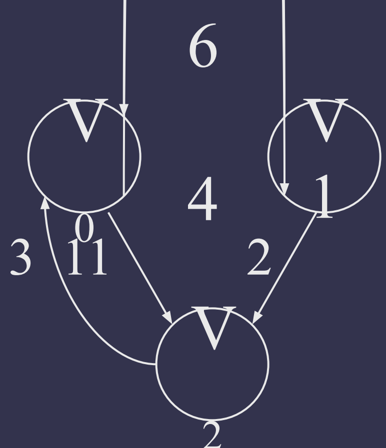
final distance:

0	4	11
6	0	2
3	7	0

cost:

0	4	11
6	0	2
3	1000	0

K 0
I
J




```
void allcosts(int cost[][MAX_VERTICES],
              int distance[][MAX_VERTICES], int n)
{
  int i,j,k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      distance[i][j] = cost[i][j];
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if (distance[i][k] + distance[k][j] <
                                distance[i][j])
          distance[i][j] =
          distance[i][k] + distance[k][j];
}
```

Program 6.12: All pairs, shortest paths function

Topological Sorts (1/19)

- Activity on vertex (AOV) networks
  - All but the simplest of projects can be divided into several subprojects called *activities*.
  - The successful completion of these activities results in the completion of the entire project

Topological Sorts (1/19)

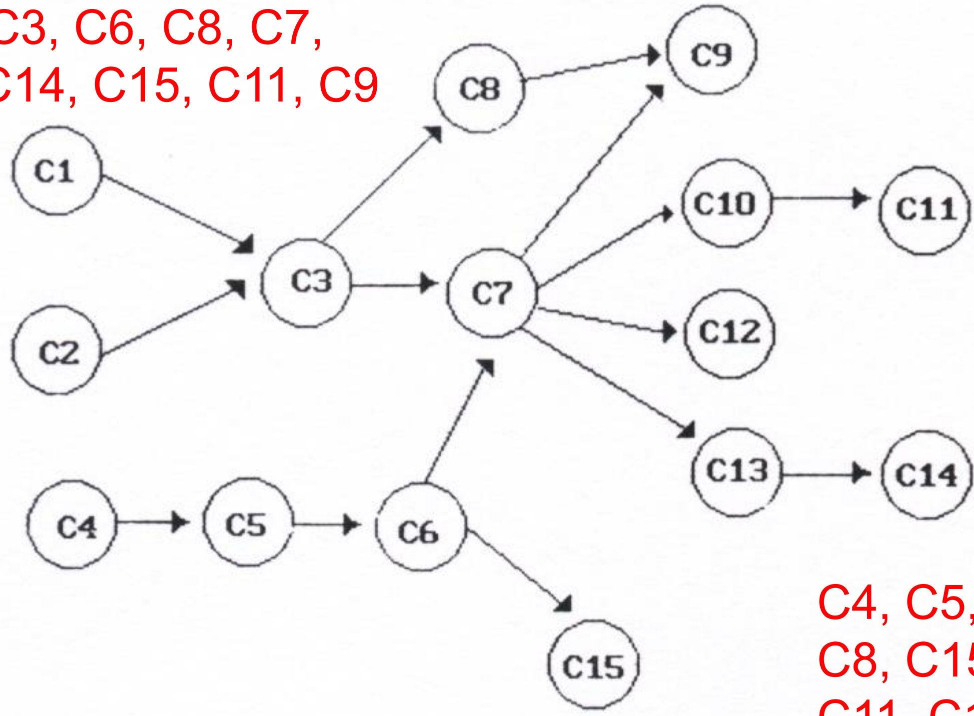
- Activity on vertex (AOV) networks
  - Example: A student working toward a degree in computer science must complete several courses successful

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university

Topological Sorts (3/19)

C1, C2, C4, C5, C3, C6, C8, C7,
C10, C13, C12, C14, C15, C11, C9



C4, C5, C2, C1, C6, C3,
C8, C15, C7, C9, C10,
C11, C13, C12, C14

Topological Sorts (2/19)

- Definition:
  - Activity On Vertex (AOV) Network:
    a directed graph in which the vertices represent tasks or activities and the edges represent precedence relations between tasks.
  - predecessor (successor):
    vertex *i* is a predecessor of vertex *j* iff there is a directed path from *i* to *j*. *j* is a successor of *i*.

Topological Sorts (2/19)

- Definition:
  - partial order:
    a precedence relation which is both transitive ($\forall$ i, j, k, i · j & j · k $\rightarrow$ i · k) and irreflexive
  - acylic graph:
    a directed graph with no directed cycles

Topological Sorts (3/19)

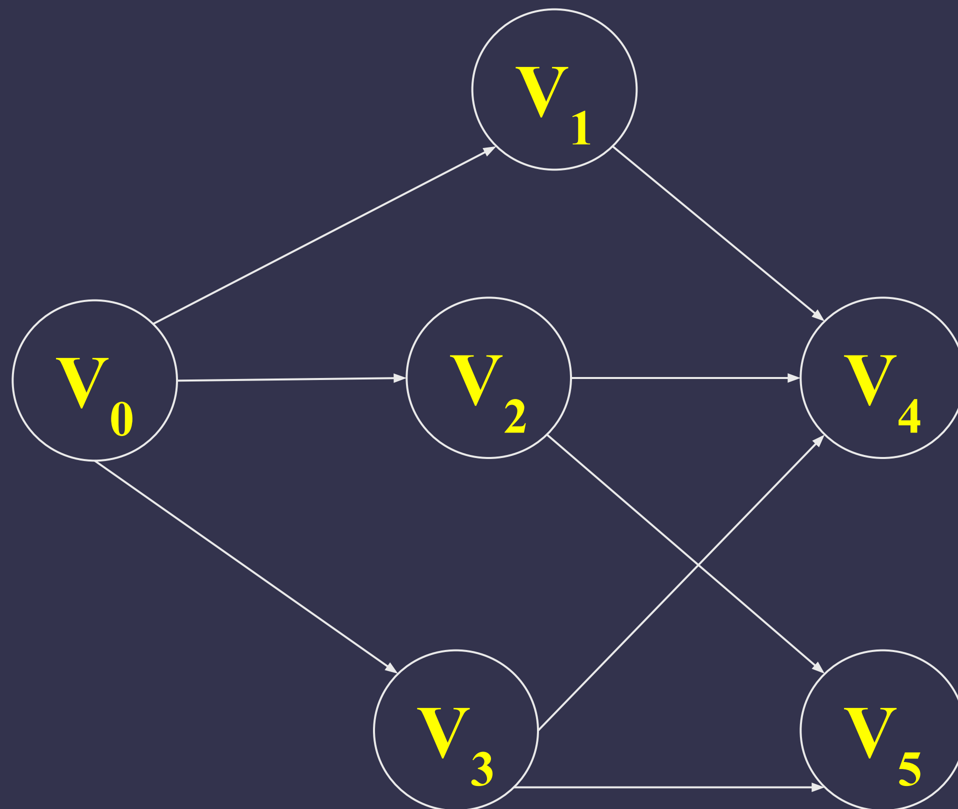
- Definition: Topological order
  - linear ordering of vertices of a graph
  - $\forall$ *i*, *j* if *i* is a predecessor of *j*, then *i* precedes *j* in the linear ordering

Topological Sorts (4/19)

- Topological sort Algorithm

```
for (i = 0; i <n; i++) {
  if every vertex has a predecessor {
    fprintf (stderr, “Network has a cycle. \n “ );
    exit(1);
  }
  pick a vertex v that has no predecessors;
  output v;
  delete v and all edges leading out of v from the network;
}
```

Topological Sorts (4/19)



Topological order generated: v_0 v_3 v_2 v_5 v_1 v_4

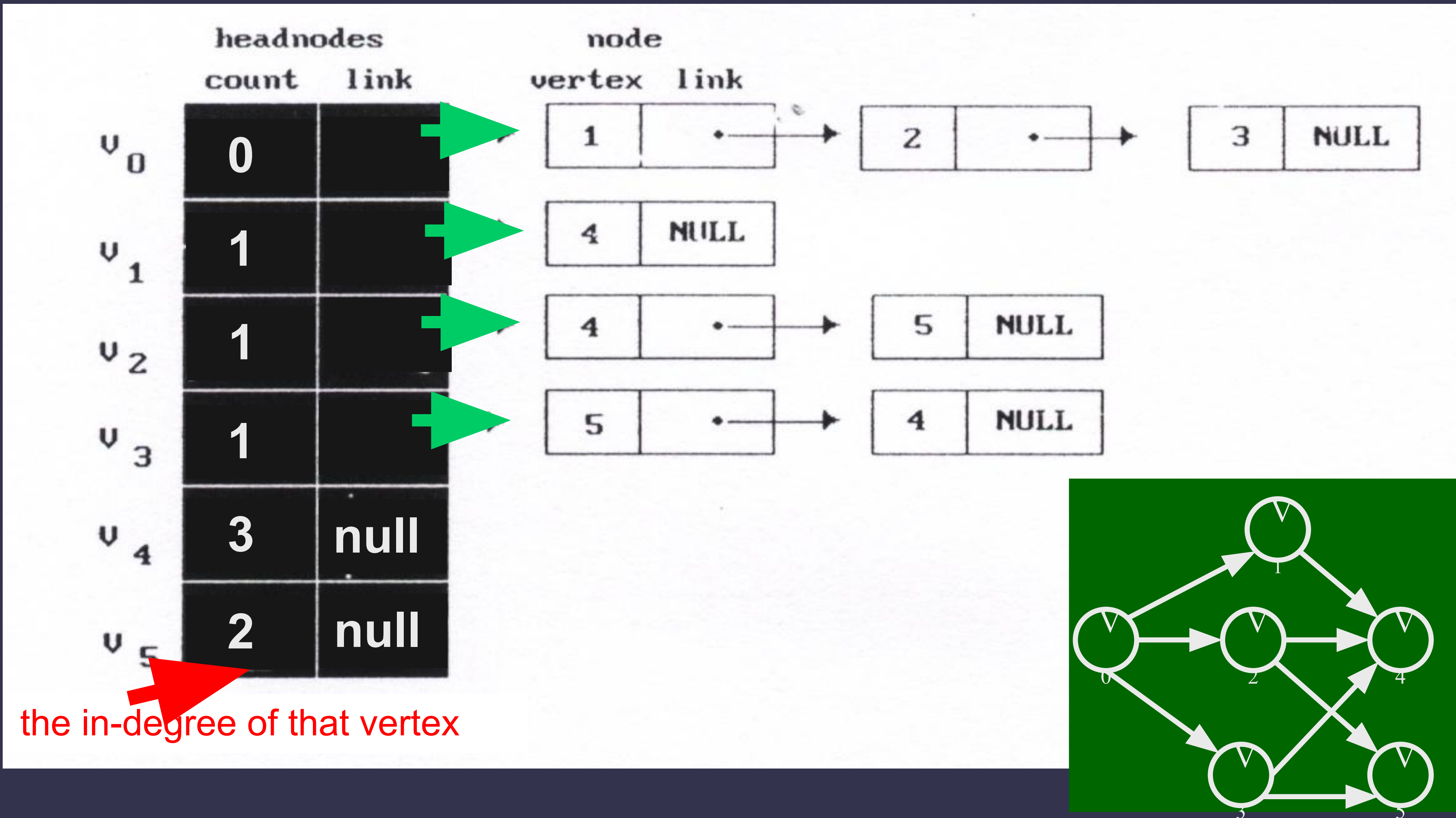
Topological Sorts (5/19)

- data representation

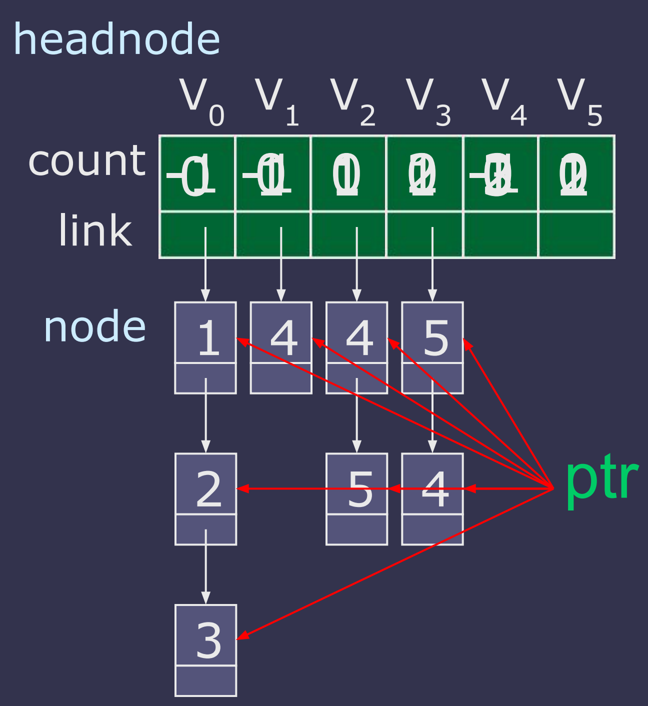
```
typedef   struct  node *node_pointer;
typedef   struct  node {
               int vertex;
               node_pointer link;
                }
typedef  struct {
               int count;
               node_pointer link;
               } hdnodes;
hdnodes  graph[max_vertices];
```

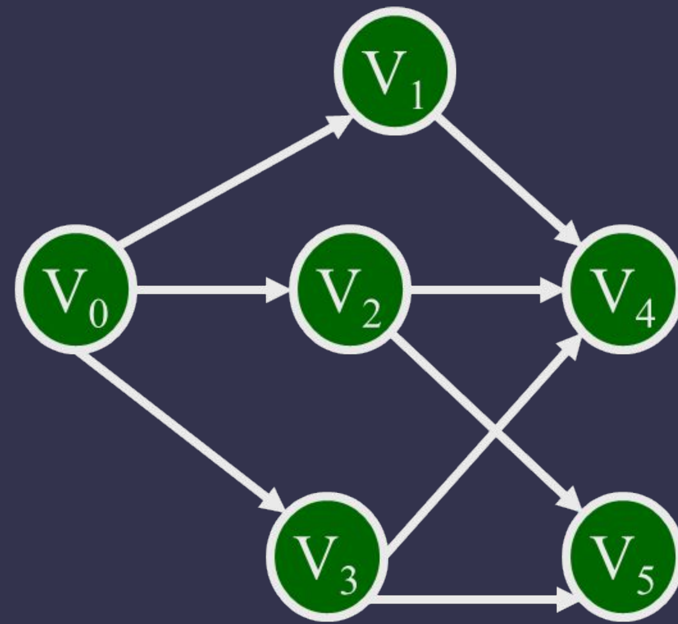
decide whether a vertex
has any predecessors

Topological Sorts (5/19)



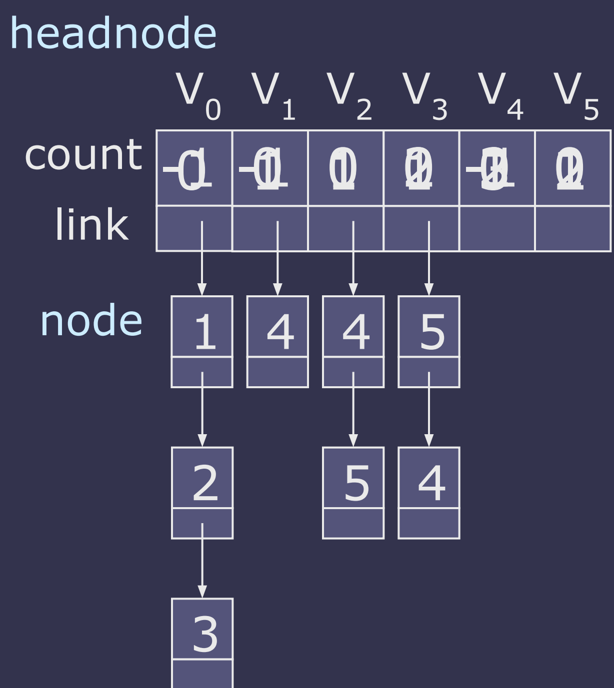
- Topological sort Program



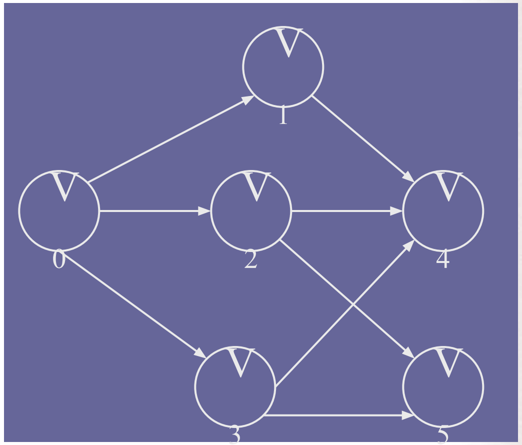


Time complexity: O(n+e)

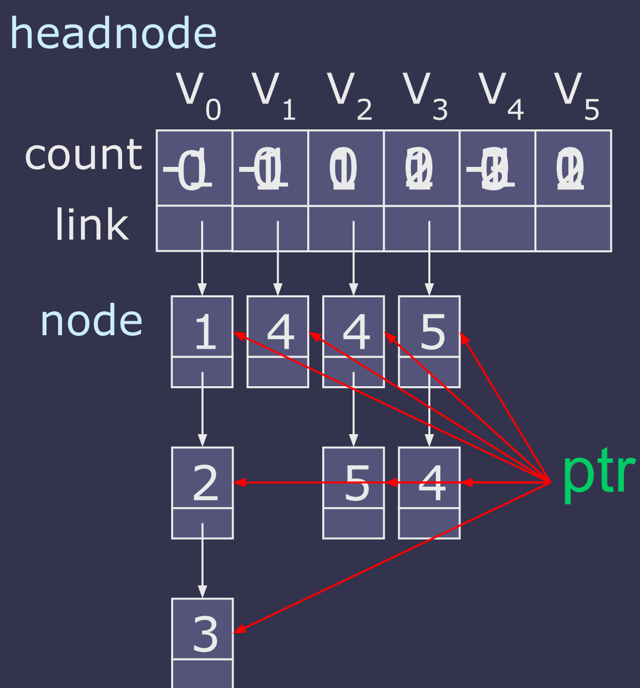
- Topological sort Program




```
void topsort(hdnodes graph[], int n)
{
   int i,j,k,top;
   node_pointer ptr;
   /* create a stack of vertices with no predecessors */
   top = -1;
   for (i = 0; i < n; i++)
      if (!graph[i].count) {
         graph[i].count = top;
         top = i;
   }
   for (i = 0; i < n; i++)
      if (top == -1) {
         fprintf(stderr,"\nNetwork
        terminated. \n");
         exit(1);
      }
      else {
         j = top;    /* unstack a vertex */
         top = graph[top].count;
         printf("v%d, ",j);
         for (ptr = graph[j].link; ptr; ptr = ptr->link) {
         /* decrease the count of the successor vertices
         of j */
            k = ptr->vertex;
            graph[k].count--;
            if (!graph[k].count) {
            /* add vertex k to the stack */
               graph[k].count = top;
               top = k;
            }
         }
      }
}
```



Topological sort Program

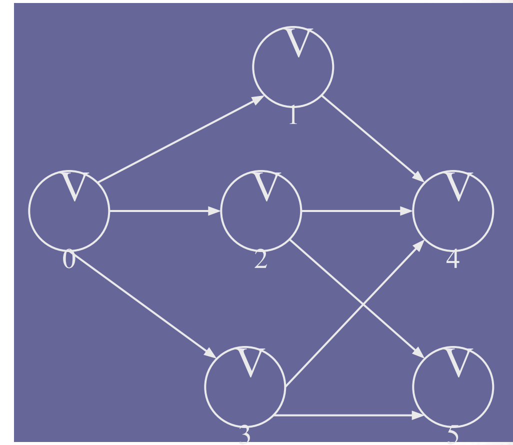


top: j: k:

output:

v0 v3 v2 v5 v1 v4

```
void topsort(hdnodes graph[], int n)
{
   int i,j,k,top;
   node_pointer ptr;
   /* create a stack of vertices with no predecessors */
   top = -1;
   for (i = 0; i < n; i++)
      if (!graph[i].count) {
         graph[i].count = top;
         top = i;
      }
   for (i = 0; i < n; i++)
      if (top == -1) {
         fprintf(stderr,"\nNetwork
        terminated. \n");
         exit(1);
      }
      else {
         j = top;    /* unstack a vertex */
         top = graph[top].count;
         printf("v%d, ",j);
         for (ptr = graph[j].link; ptr; ptr = ptr->link) {
         /* decrease the count of the successor vertices
         of j */
            k = ptr->vertex;
            graph[k].count--;
            if (!graph[k].count) {
            /* add vertex k to the stack */
               graph[k].count = top;
               top = k;
            }
         }
      }
}
```



Time complexity: O(*n*+*e*)

Topological Sorts (7/19)

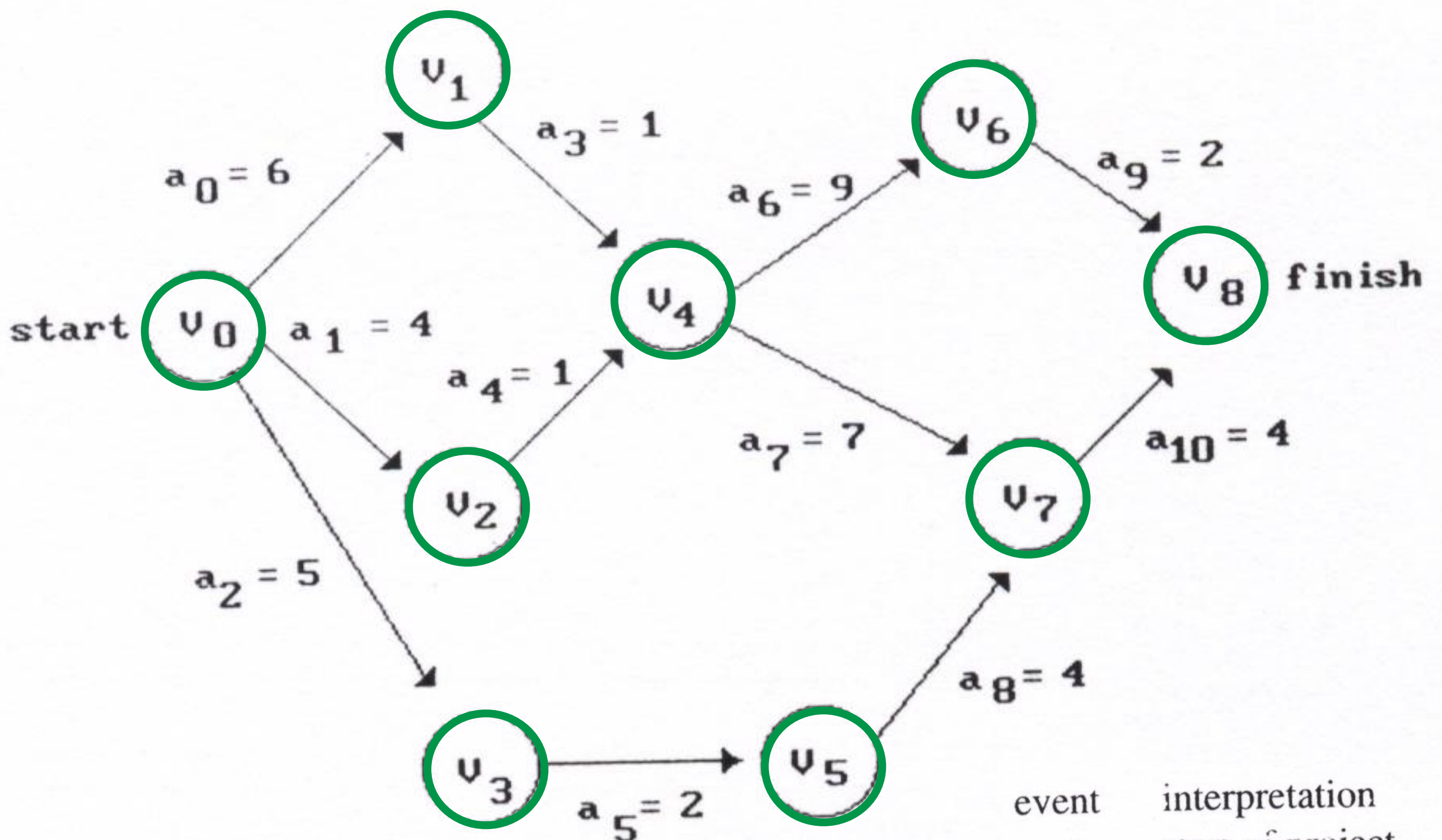
- Activity on Edge (AOE) Networks
  - AOE networks have proved very useful for evaluating the performance of many types of projects.
- What is the least amount of time in which the project may be complete (assuming there are no cycle in the network)?
- Which activities should be speeded to reduce project length?

Topological Sorts (8/19)

- An AOE network
  - directed edge: tasks or activities to be performed
  - vertex: events which signal the completion of certain activities
  - number: associated with each edge (activity) is the time required to perform the activity

Topological Sorts (8/19)

- An AOE network



(a) AOE network. Activity graph of a h

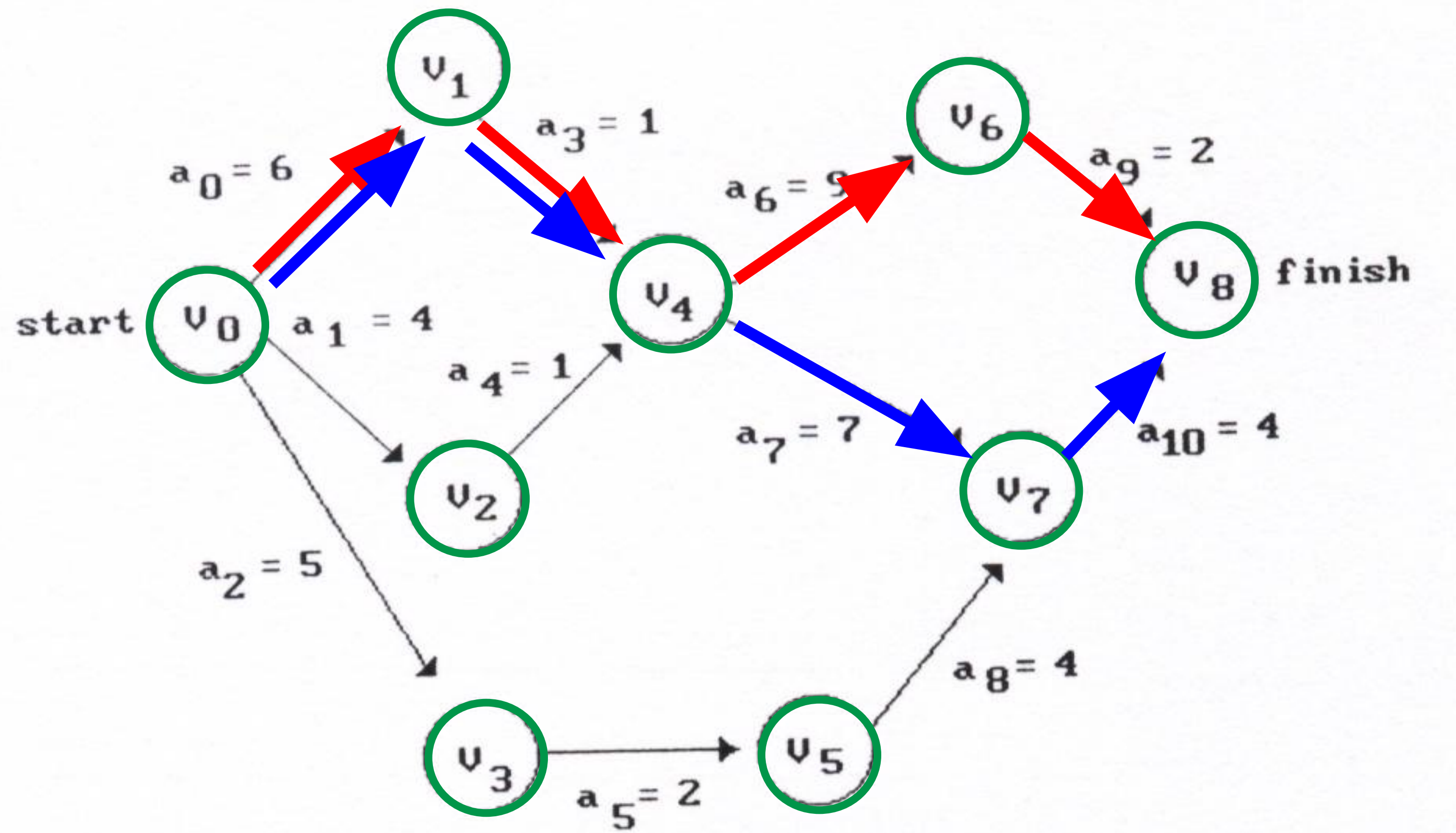
event	interpretation
v_0	start of project
v_1	completion of activity a_0
v_4	completion of activities a_3 and a_4
v_7	completion of activities a_7 and a_8
v_8	completion of project

Topological Sorts (9/19)

- Application of AOE Network
  - Evaluate performance
    - minimum amount of time
    - activity whose duration time should be shortened
    - …
  - Critical path
    - a path that has the longest length
    - minimum time required to complete the project

Topological Sorts (9/19)

- Critical path v_0, v_1, v_4, v_6, v_8 or v_0, v_1, v_4, v_7, v_8



(a) AOE network. Activity graph of a hypothetical project

Topological Sorts (10/19)

- Critical-path analysis
  - The purpose of critical-path analysis is to identify critical activities so that resource may be concentrated on these activities in an attempt to reduce a project finish time.
  - Critical-path analysis can also be carried out with AOV network
- Determine Critical Paths
  - Delete all noncritical activities
  - Generate all the paths from the start to finish vertex

Topological Sorts (11/19)

- Various Factors
  - The *earliest time* of an event v_i
    - the length of the longest path from v_0 to v_i (Ex. 7 for v_4)
  - *early*(*i*): Earliest activity time,
    i.e., earliest start time of activity a_i
  - *late*(*i*): Latest activity time,
  - i.e., latest start time of activity a_i
  - ***late*(*i*)-*early*(*i*)**
    - measure of how critical an activity is (ex. *late*(5)-*early*(5)=8-5=3)
  - Critical activity
    - an activity for which *early*(*i*)=*late*(*i*) (ex. *early*(7)=*late*(7))

Topological Sorts (12/19)

- Various Factors (cont'd)
  - *earliest*[*j*]:
    earliest event occurrence time for event *j* (vertex *j*)
  - *latest*[*j*]:
    latest event occurrence time for event *j* (vertex *j*) in the network
  - If activity a_i is represented by edge $<k, j>$
    - *early*(*i*) = *earliest*[*k*]
    - *late*(*i*) = *latest*[j] - *duration of activity* a_i

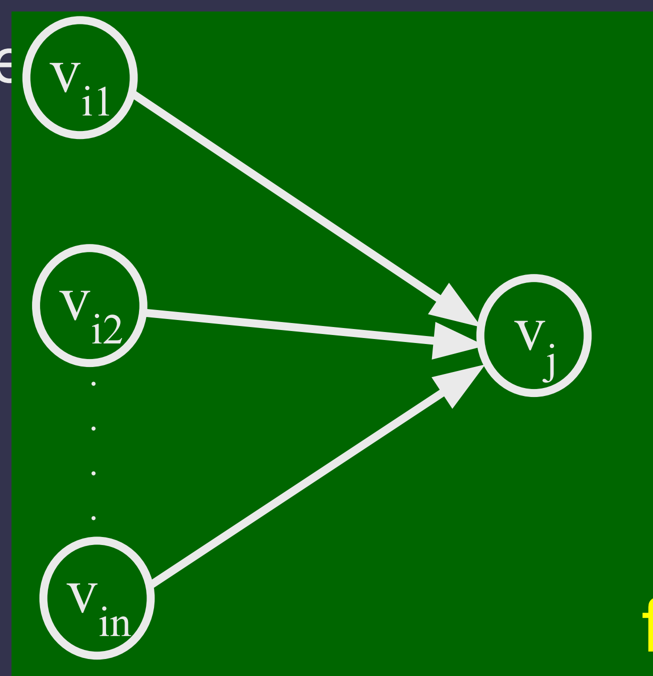
  - We compute the times *earliest*[*j*] and *latest*[*j*] in two stages: a forward stage and a backward stage

Topological Sorts (13/19)

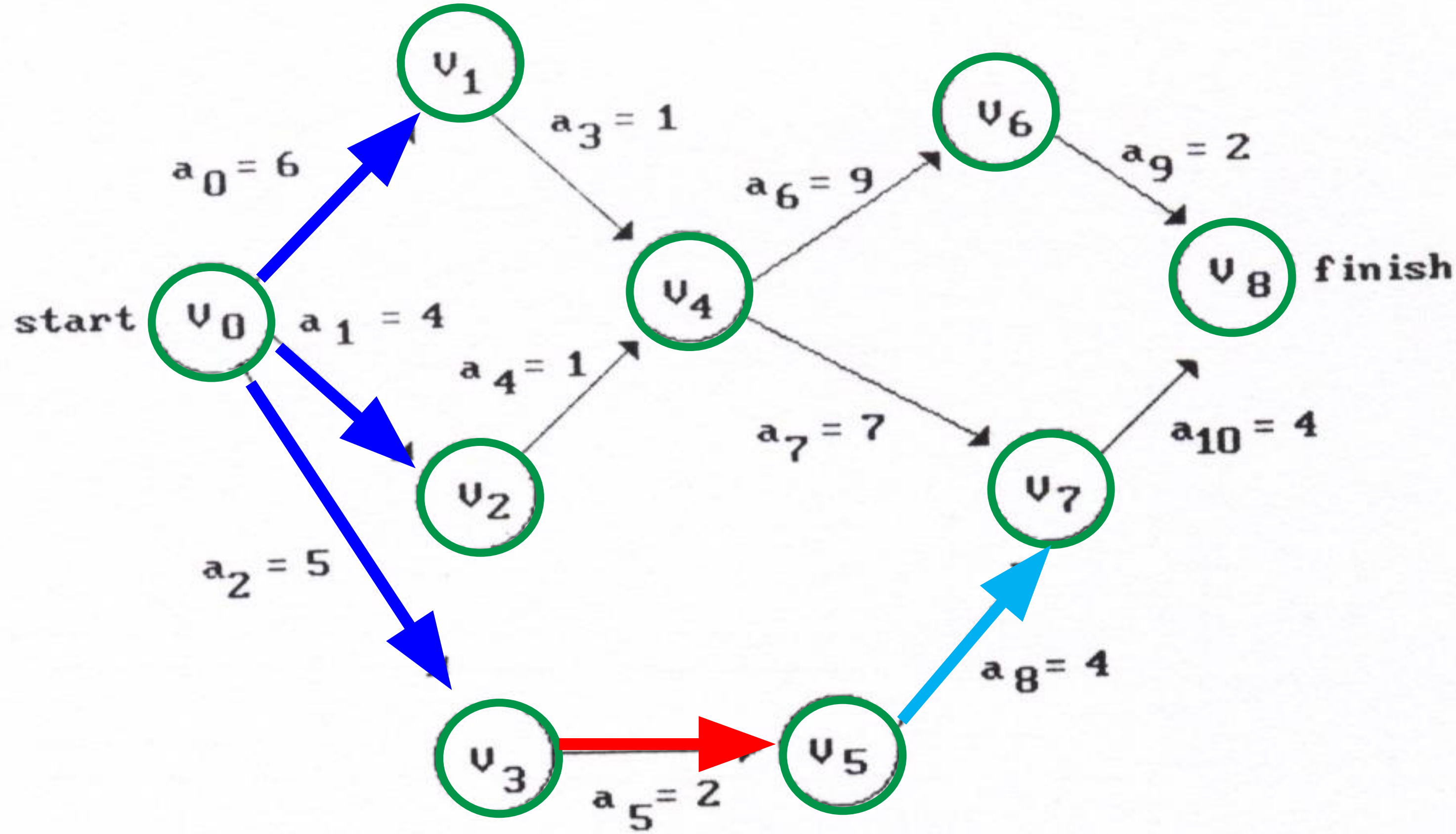
- Calculation of Earliest Times
  - During the forwarding stage, we start with *earliest*[0] = 0 and compute the remaining start times using the formula:

$$earliest[j] = \max_{i \in p(j)} \{earliest[i] + \text{duration of} < i, j >\}$$

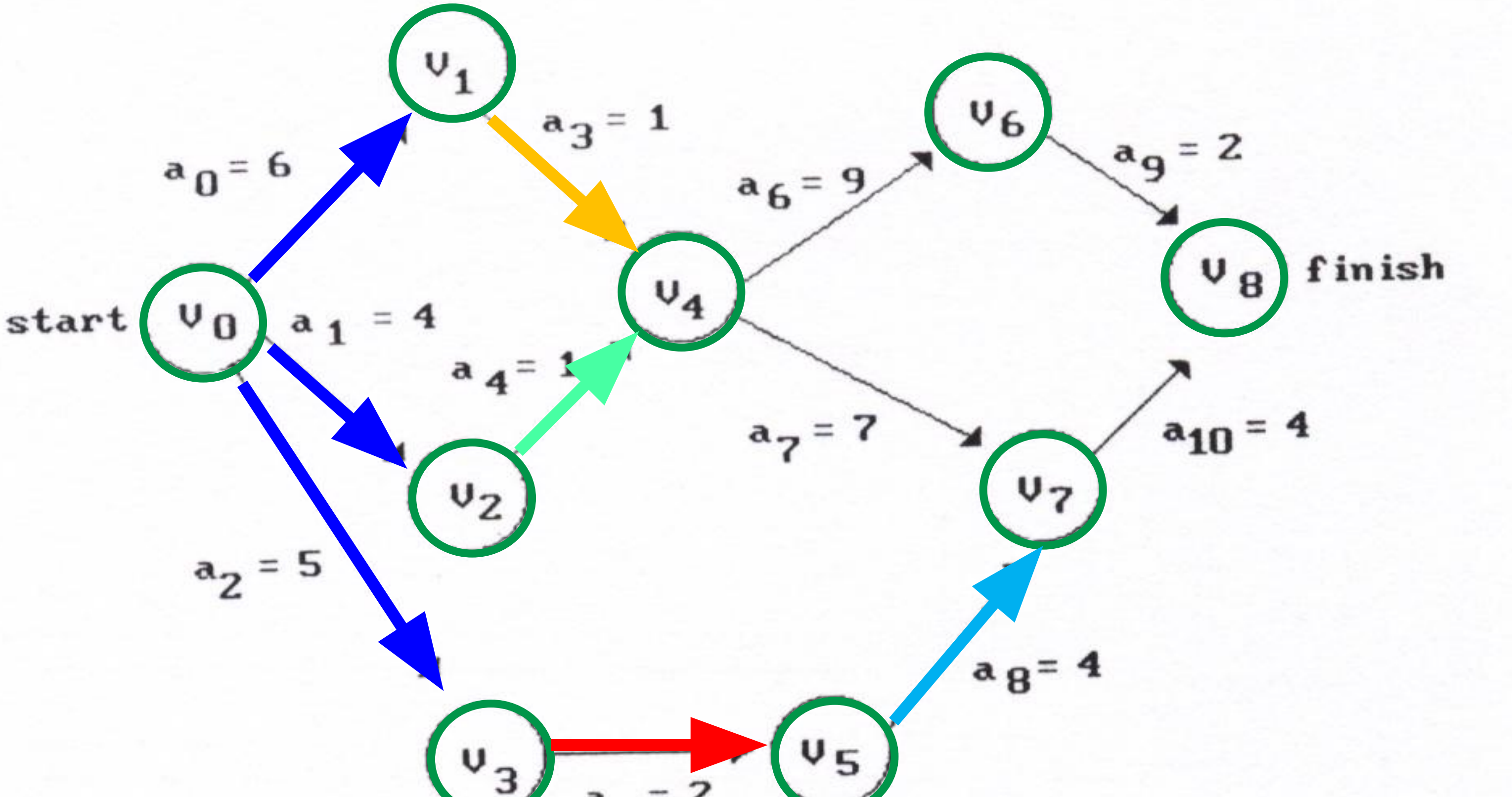
Where $P(j)$ is the ecessors of j



forward stage

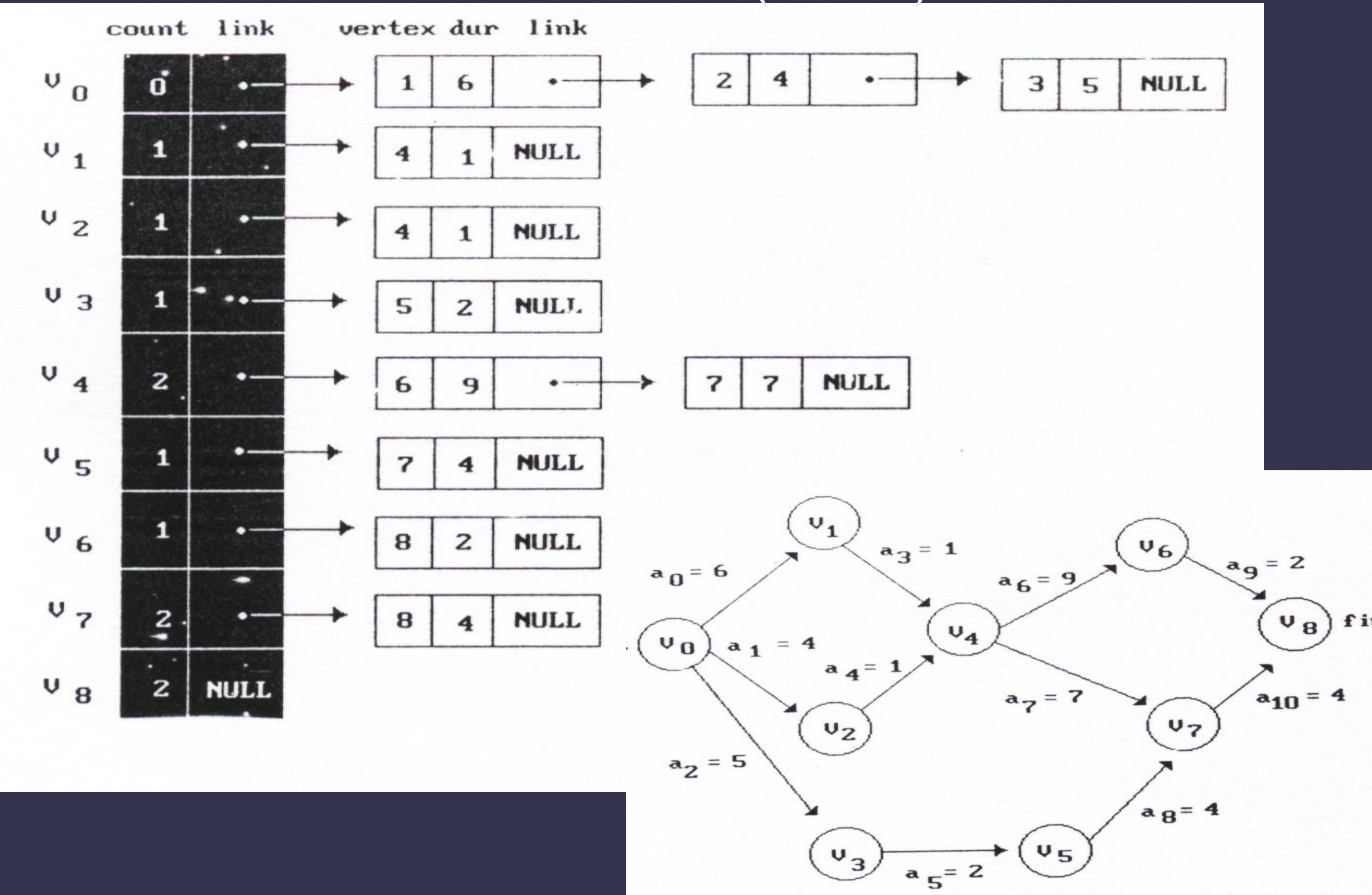


Earliest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
initial	0	0	0	0	0	0	0	0	0
output v_0	0	6	4	5	0	0	0	0	0
output v_3	0	6	4	5	0	7	0	0	0
output v_5	0	6	4	5	0	7	0	11	0

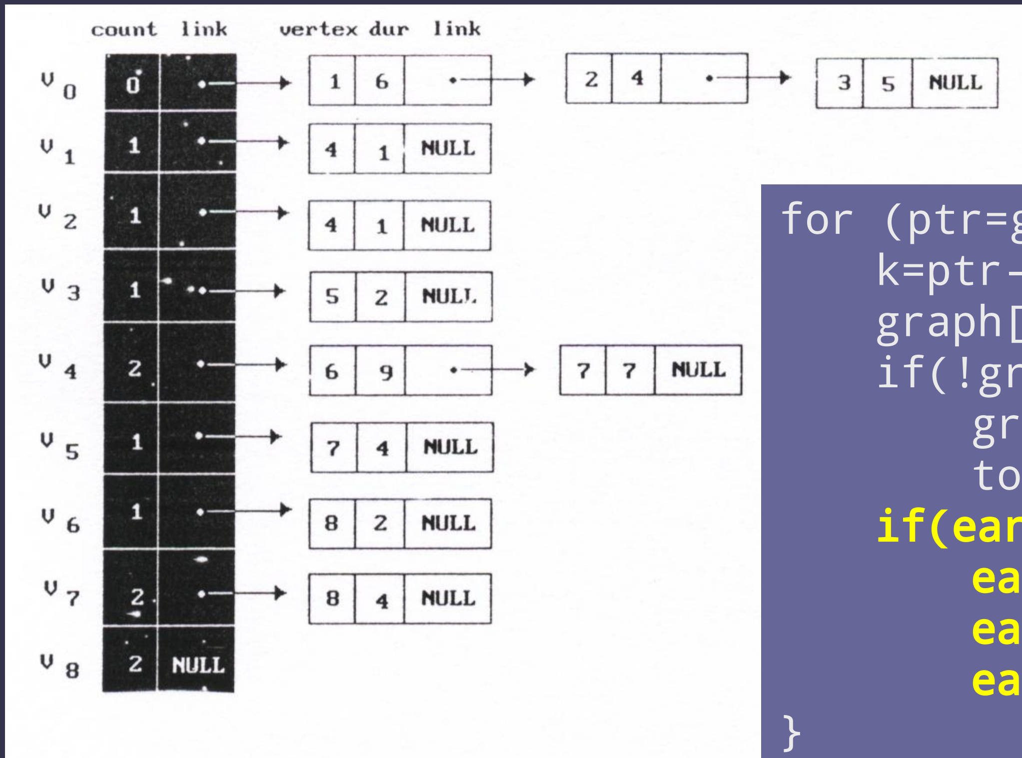


Earliest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
output v_5	0	6	4	5	0	7	0	11	0
output v_2	0	6	4	5	5	7	0	11	0
output v_1	0	6	4	5	7	7	0	11	0
output v_4	0	6	4	5	7	7	16	14	0
output v_7	0	6	4	5	7	7	16	14	16
output v_6	0	6	4	5	7	7	16	14	18

Calculation of Earliest Times (cont'd)



Calculation of Earliest Times (cont'd)




```
for (ptr=graph[j].link;ptr;ptr=ptr->link){
    k=ptr->vertex;
    graph[k].count--;
    if(!graph[k].count){
        graph[k].count=top;
        top=k;}
    if(earliest[k]<
        earliest[j]+ptr->duration)
        earliest[k] =
        earliest[j]+ptr->duration;
}
```

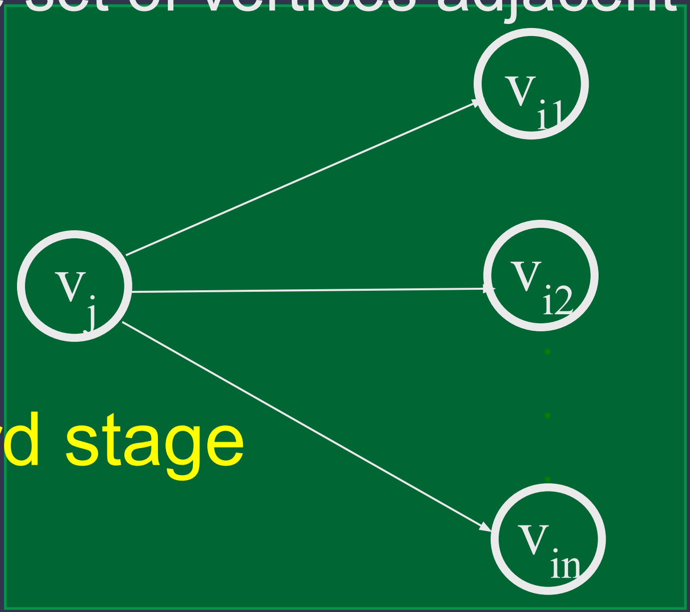
Earliest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	0	0	0	0	0	0	0	0	0	[0]
output v_0	0	6	4	5	0	0	0	0	0	[3, 2, 1]
output v_3	0	6	4	5	0	7	0	0	0	[5, 2, 1]
output v_5	0	6	4	5	0	7	0	11	0	[2, 1]
output v_2	0	6	4	5	5	7	0	11	0	[1]
output v_1	0	6	4	5	7	7	0	11	0	[4]
output v_4	0	6	4	5	7	7	16	14	0	[7, 6]
output v_7	0	6	4	5	7	7	16	14	16	[6]
output v_6	0	6	4	5	7	7	16	14	18	[8]
output v_8										

Topological Sorts (15/19)

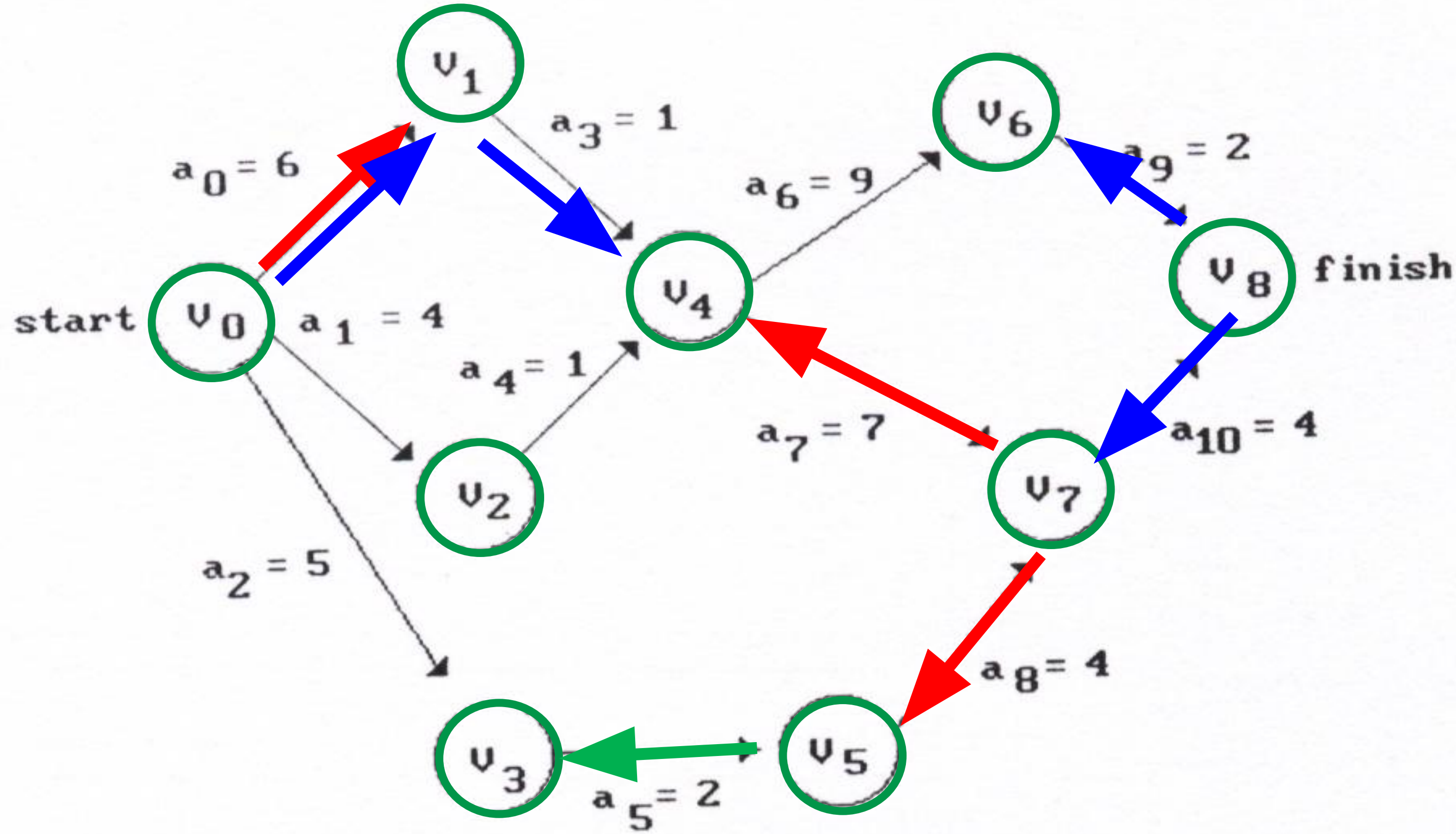
- Calculation of latest times
  - In the backward stage, we compute the values of *latest*[*i*] using a procedure analogous to that used in the forward stage.
  - We start with *latest*[*n*-1] = *earliest*[*n*-1] and use the formula:

$$latest\,[\,j\,] = \min_{i \in S(j)} \{latest\,[\,i\,] - \text{duration of} < j, i >\}$$

Where *S*(*j*) is the set of vertices adjacent from vertex *j*

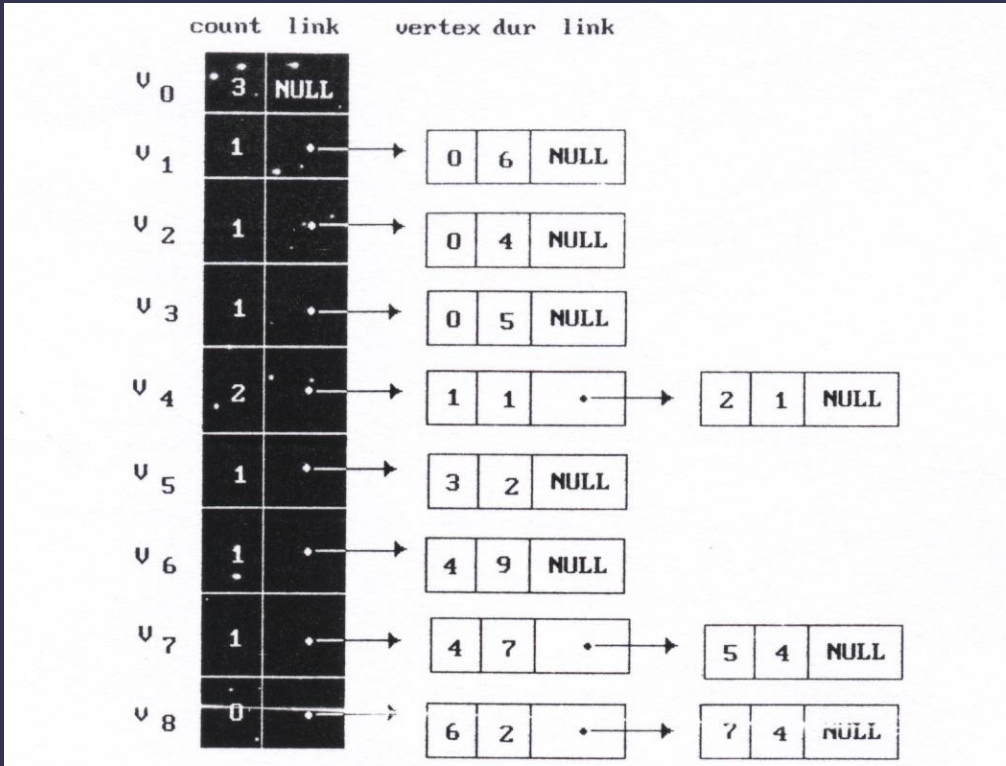


backward stage

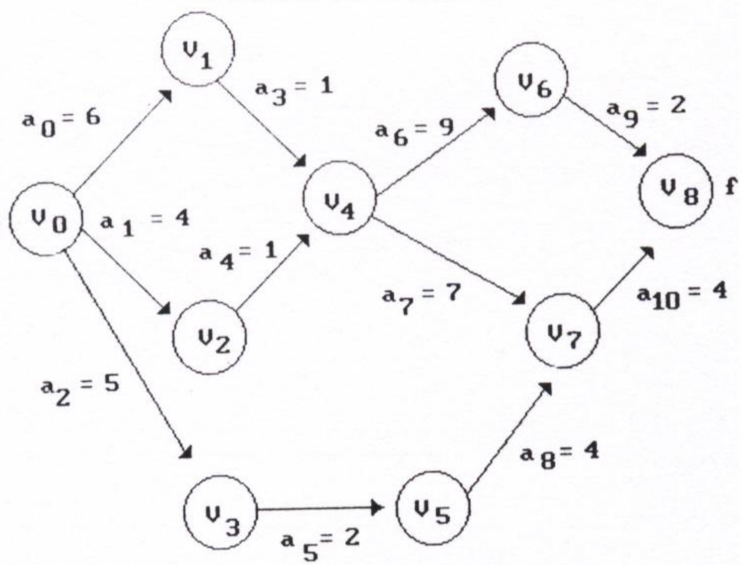


Latest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
initial	18	18	18	18	18	18	18	18	18
output v_8	18	18	18	18	18	18	16	14	18
output v_7	18	18	18	18	7	10	16	14	18
output v_5	18	18	18	8	7	10	16	14	18
output v_3	3	18	18	8	7	10	16	14	18
output v_6	3	18	18	8	7	10	16	14	18
output v_4	3	6	6	8	7	10	16	14	18
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

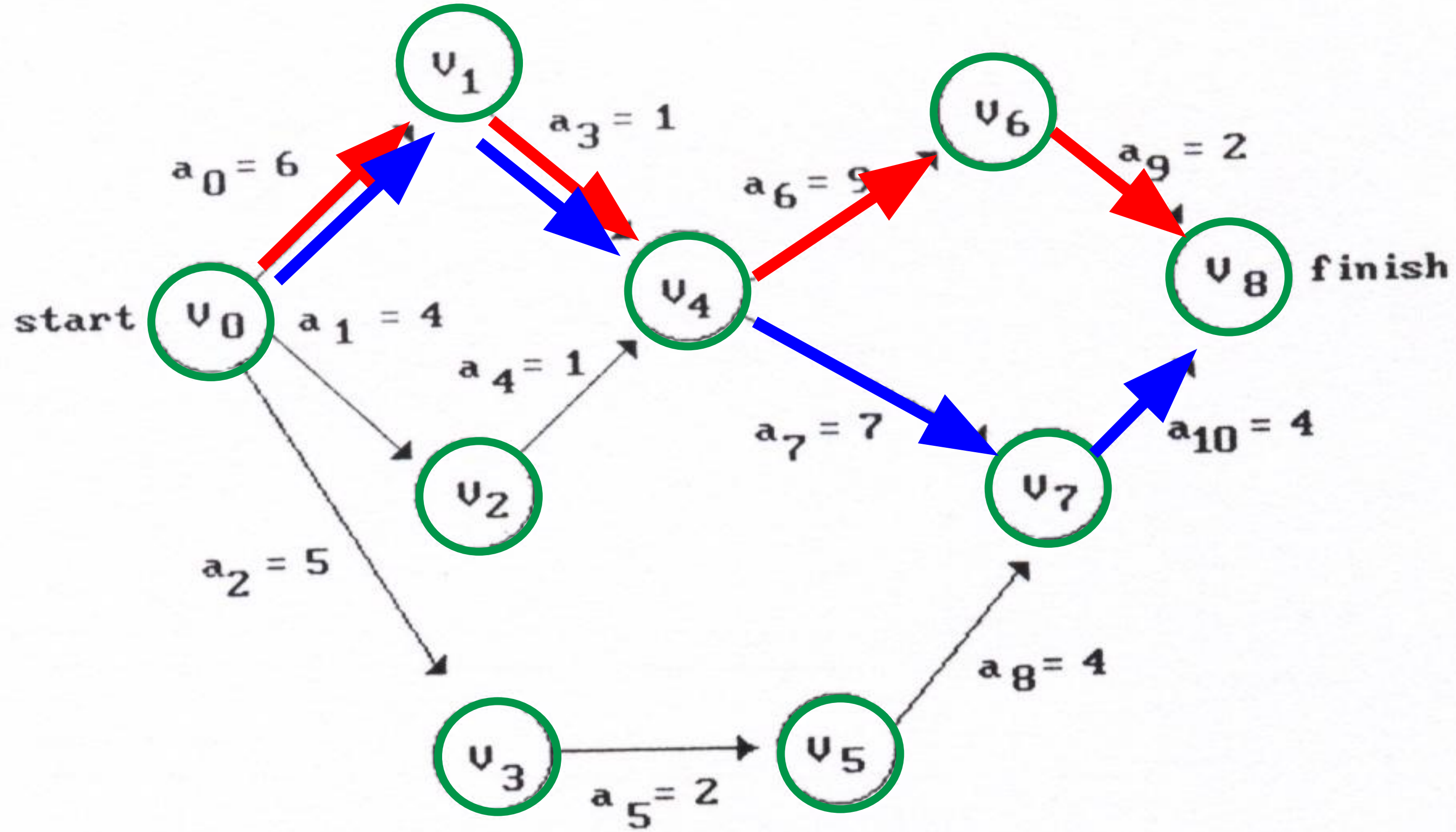
Calculation of latest Times (cont'd)



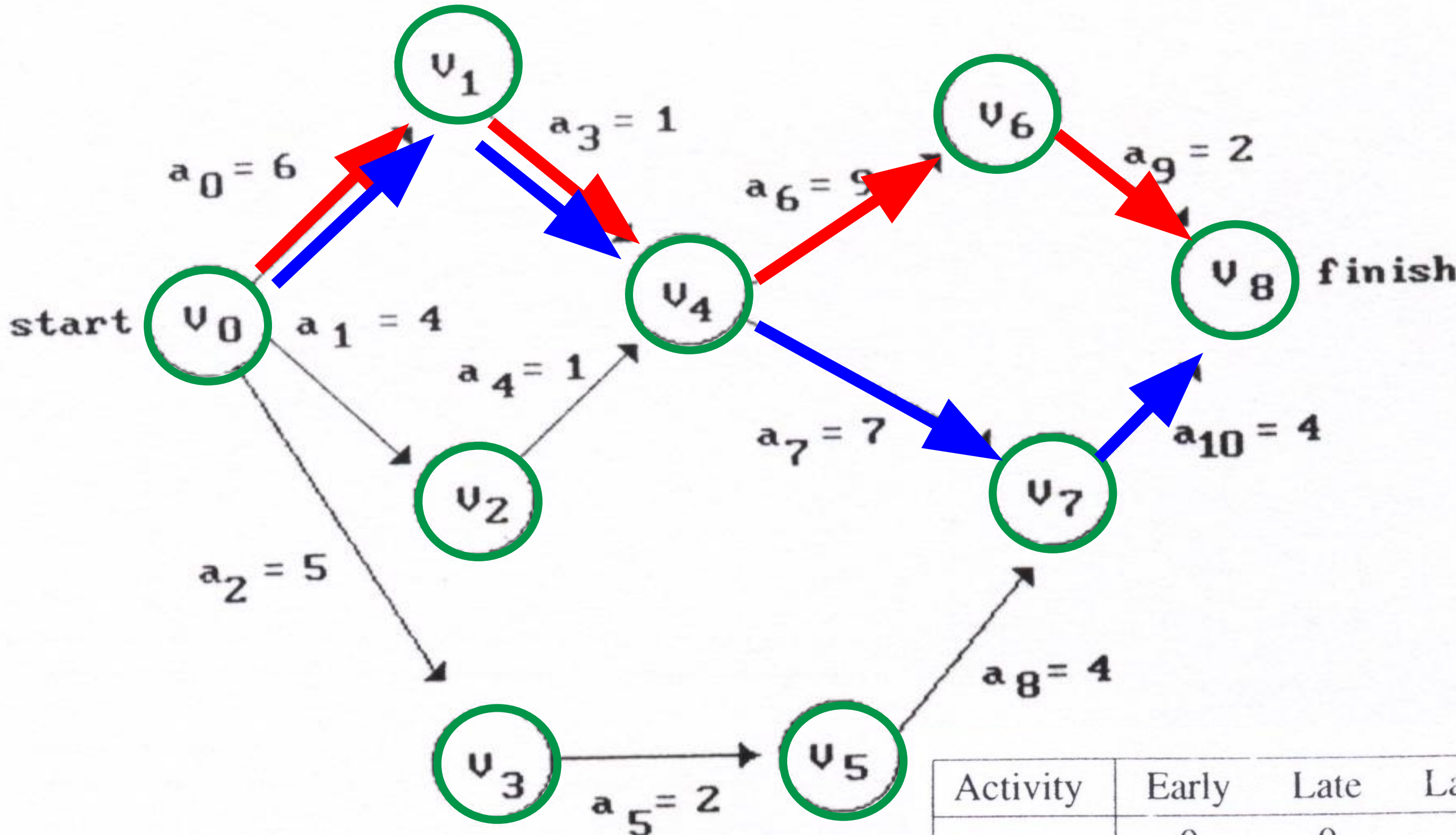

```
for (ptr=graph[j].link;ptr;
    ptr=ptr->link){
    k=ptr->vertex;
    graph[k].count--;
    if(!graph[k].count){
        graph[k].count=top;
        top=k;}
    if(latest[k]>
        latest[j]-ptr->duration)
        latest[k] =
        latest[j]-ptr->duration;
}
```



Latest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	18	18	18	18	18	18	18	18	18	[8]
output v_8	18	18	18	18	18	18	16	14	18	[7, 6]
output v_7	18	18	18	18	7	10	16	14	18	[5, 6]
output v_5	18	18	18	18	7	10	16	14	18	[3, 6]
output v_3	3	18	18	8	7	10	16	14	18	[6]
output v_6	3	18	18	8	7	10	16	14	18	[4]
output v_4	3	6	6	8	7	10	16	14	18	[2, 1]
output v_2	2	6	6	8	7	10	16	14	18	[1]
output v_1	0	6	6	8	7	10	16	14	18	[0]



Activity	Early	Late	Late – Early	Critical
a_0	0	0	0	yes
a_1	0	2	2	no
a_2	0	3	3	no
a_3	6	6	0	yes
a_4	4	6	2	no
a_5	5	8	3	no
a_6	7	7	0	yes



(a) AOE network. Activity graph o

Activity	Early	Late	Late – Early	Critical
a_0	0	0	0	yes
a_1	0	2	2	no
a_2	0	3	3	no
a_3	6	6	0	yes
a_4	4	6	2	no
a_5	5	8	3	no
a_6	7	7	0	yes
a_7	7	7	0	yes
a_8	7	10	3	no
a_9	16	16	0	yes
a_{10}	14	14	0	yes