Shortest Path Algorithms

Definitions

Weighted graph

Weighted path length

Unweighted path length

Single Source Shortest Path Problem

Given input weighted graph, G=(V,E) and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G

Negative Cost Cycle

Eg. Flight routes

Shortest Path Algorithms

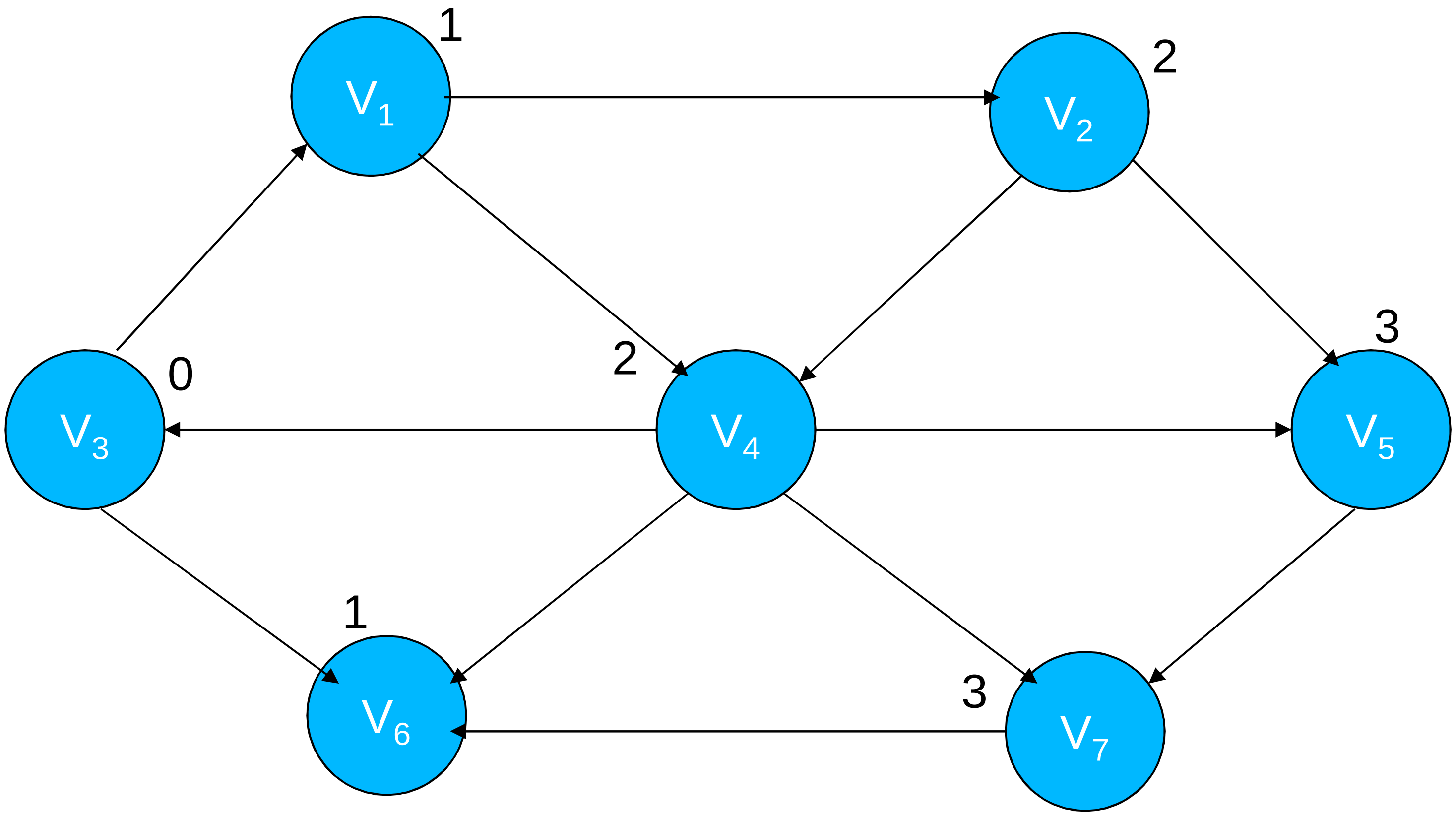
Unweighted shortest paths

Dijkstra’s Algorithm

Acyclic Graphs

All-Pairs Shortest Path

Unweighted Shortest Path



Initial Configuration Table

V	Known	Dv	Pv
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

Pseudocode for unweighted shortest path algorithm

```
void unweighted(table t)
{
    queue q;
    vertex v,w;
    q=createqueue(numvertex);
    makeempty(q);
    enqueue(s,q);
    while(!isempty(q))
    {
      v=dequeue(q);
      t[v].known=true;
      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);
          }
    }
    disposequeue(q);
}
```

How the data changes during the unweighted shortest path algorithm

V	Initial State		
	Known	Dv	Pv
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
Q:	V_3		

V	V_3 Dequeued		
	Known	Dv	Pv
V_1	0	1	V_3
V_2	0	∞	0
V_3	1	0	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	1	V_3
V_7	0	∞	0
Q:	V_1, V_6		

Contd.

V	V_1 Dequeued		
	Known	Dv	Pv
V_1	1	1	V_3
V_2	0	2	V_1
V_3	1	0	0
V_4	0	2	V_1
V_5	0	∞	0
V_6	0	1	V_3
V_7	0	∞	0
Q:	V_6 , V_2 , V_4		

V	V_6 Dequeued		
	Known	Dv	Pv
V_1	1	1	V_3
V_2	0	2	V_1
V_3	1	0	0
V_4	0	2	V_1
V_5	0	∞	0
V_6	1	1	V_3
V_7	0	∞	0
Q:	V_2 , V_4		

Contd.

V	V_2 Dequeued		
	Known	Dv	Pv
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	0	2	V_1
V_5	0	3	V_2
V_6	1	1	V_3
V_7	0	∞	0
Q:	V_4 , V_5		

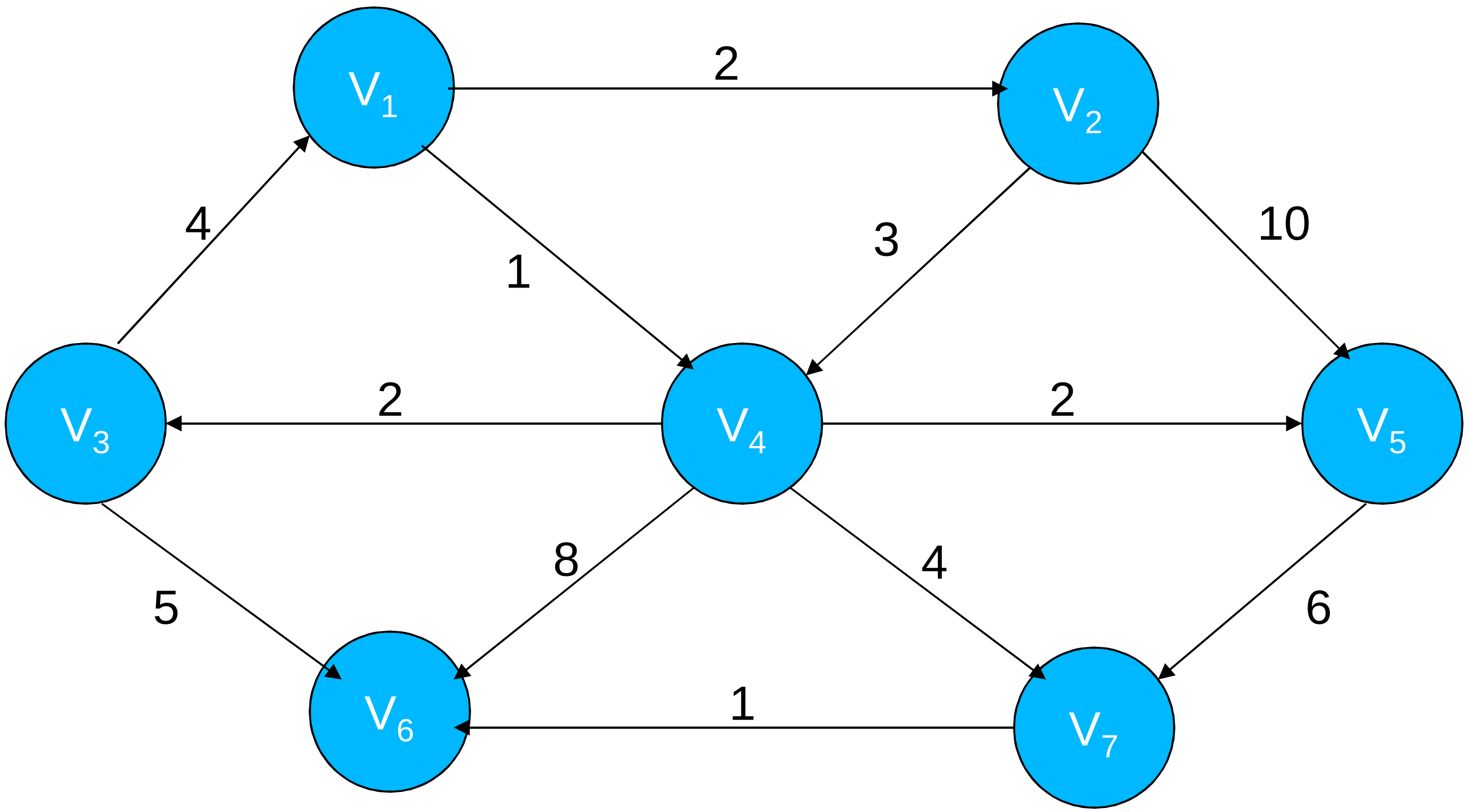
V	V_4 Dequeued		
	Known	Dv	Pv
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	1	2	V_1
V_5	0	3	V_2
V_6	1	1	V_3
V_7	0	3	V_4
Q:	V_5 , V_7		

Contd.

V	V_5 Dequeued		
	Known	Dv	Pv
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	1	2	V_1
V_5	1	3	V_2
V_6	1	1	V_3
V_7	0	3	V_4
Q:	V_7		

V	V_7 Dequeued		
	Known	Dv	Pv
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	1	2	V_1
V_5	1	3	V_2
V_6	1	1	V_3
V_7	1	3	V_4
Q:	empty		

Dijkstra's Algorithm



Initial Configuration of table used in Dijkstra's algorithm

V	Known	Dv	Pv
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

After V_1 is declared known

V	Known	Dv	Pv
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

After V_4 is declared known

V	Known	Dv	Pv
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

After V_2 is declared known

V	Known	Dv	Pv
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

After V_5 and then V_3 are declared known

V	Known	Dv	Pv
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

After V_7 is declared known

V	Known	Dv	Pv
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

After V_6 is declared known and algorithm terminates

V	Known	Dv	Pv
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

Declarations for Dijkstra’s algorithm

```
typedef int vertex;
struct tableentry
{
  int known;
  disttype dist;
  vertex path;
};
#define notavertex -1
typedef struct tableentry table[numvertex];
```

Table Initialization Routine

```
void inittable(vertex start, graph g, table t)
{
   int i;
   readgraph(g,t);
   for(i=0;i<numvertex;i++)
   {
    t[i].known = false;
    t[i].dist = infinity;
    t[i].path = notavertex;
   }
   t[start].dist = 0;
}
```

Pseudocode for Dijkstra's Algorithm

```
void dijkstra(table t)
{
    vertex v,w;
    for(; ;)
    {
            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)
                 {
                      t[w].dist = t[v].dist + Cvw;
                      t[w].path = v;
                 }
    }
}
```

Acyclic Graphs

- Critical Path analysis
- Activity node graph
- Eg. Chemical reactions
- If EC_i is the earliest completion time for node i, then the applicable rules are

$$EC_1 = 0$$

$$EC_w = \max_{(v,\, w)\, \varepsilon\, E}(EC_v + C_{v,w})$$

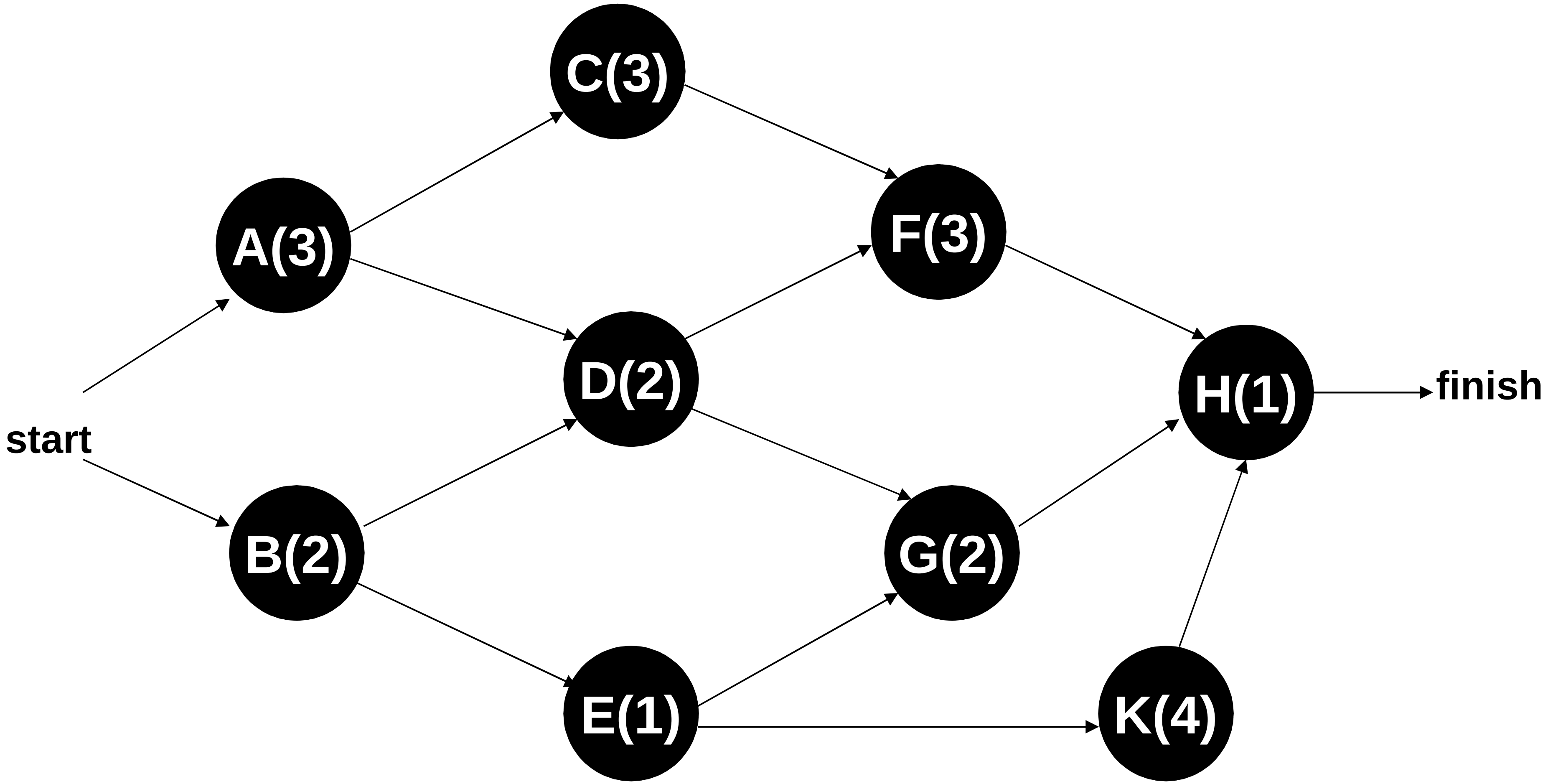
- Latest time, LC_i, each event can finish without affecting the final completion time

$$LC_n = EC_n$$

$$LC_v = \min_{(v,\, w)\, \varepsilon\, E}(LC_w - C_{v,w})$$

- $Slack_{(v,w)} = LC_w – EC_v - C_{v,w}$

Activity Node Graph



All-Pairs Shortest Path

- a[ ] contains the adjacency matrix
- d[ ] contains the values of the shortest path
- n is the number of vertices
- Actual path is computed using path[ ]

<u>Algorithm</u>

```
void allpairs(twodimarray A, twodimarray D, twodimarray path, int n)
{
      int i,j,k;
      for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
                d[i][j]=a[i][j];
                path[i][j]=notavertex;
        }
        for(k=0;k<n;k++)
        for(i=0;i<n;i++)
        for(j=0;j<n;j++)
                if(d[i][k]+d[k][j]<d[i][j])
                {
                        d[i][j] = d[i][k] + d[k][j];
                        path[i][k]=k;
                }
}
```