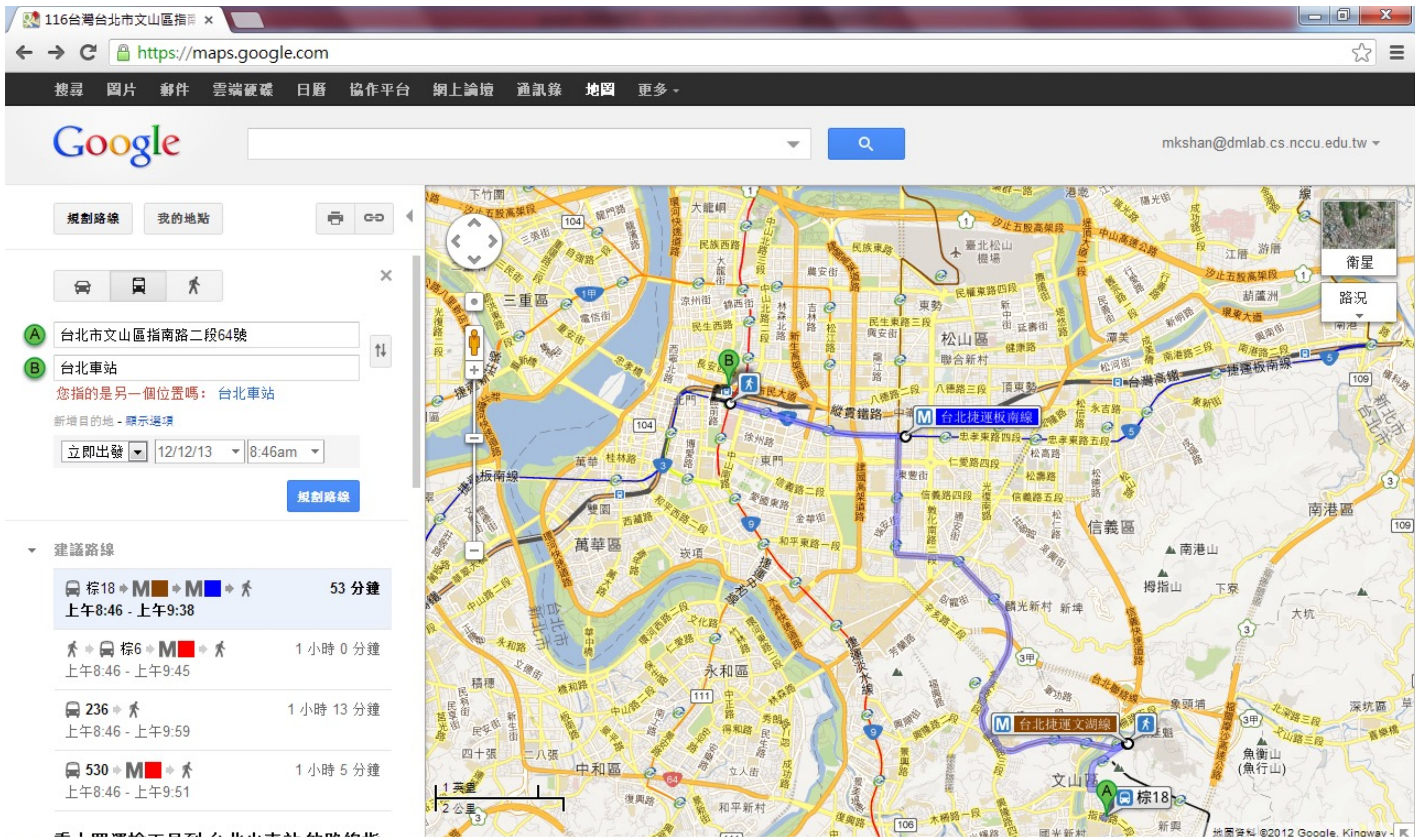
Algorithms

Graph-2

Single Source Shortest Path

(pp. 201~208)


116台灣台北市文山區指
https://maps.google.com
搜尋 圖片 郵件 雲端硬碟 日曆 協作平台 網上論壇 通訊錄 地圖 更多
Google
mkshan@dmlab.cs.nccu.edu.tw
規劃路線 我的地點
台北市文山區指南路二段64號
台北車站
您指的是另一個位置嗎： 台北車站
新增目的地 - 顯示選項
立即出發 12/12/13 8:46am
規劃路線
建議路線
棕18 53 分鐘
上午8:46 - 上午9:38
棕6 1 小時 0 分鐘
上午8:46 - 上午9:45
236 1 小時 13 分鐘
上午8:46 - 上午9:59
530 1 小時 5 分鐘
上午8:46 - 上午9:51
衛星
路況
台北捷運板南線
台北捷運文湖線
松山區
信義區
萬華區
永和區
中和區
三重區
南港區
文山區
棕18
1 英里
2 公里
地圖資料 ©2012 Google, Kingway

M. K. Shan, CS, NCCU

Shortest-Path Problem

- **Problem:**

  Given a graph $G = (V, E)$

  and a vertex v

  Find shortest path v to all other vertices of G
- **Single-Source All Destination Shortest-Path**
- **Types of problems**
  - **Unweighted shortest paths O(|V|+|E|)**
  - **Weighted shortest paths in acyclic graphs O(|E|+|V|)**
  - **Weighted shortest paths in cyclic graph without negative edges O(|E|log|V|)**
  - **Weighted Shortest Paths in cyclic graph with negative edges O(|E|*|V|)**

Unweighted Shortest Paths

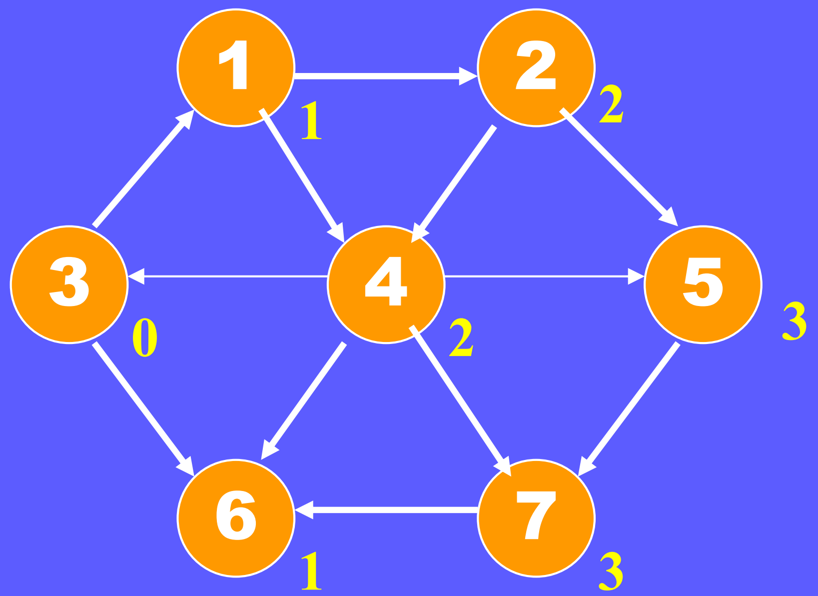
- **Problem:**

  Given a undirected or directed unweighted graph $G = (V, E)$

  and a vertex v

  Find shortest path v to all other vertices of G
- **Unweighted shortest paths: minimum no. of edges along path**
- **Strategy of algorithm:**
  - **breadth-first search, processing vertices in layers**
  - **the vertices closest to the start are evaluate first**
  - **≈ level order traversal of trees**
  - **Improved using queue**

Unweighted Shortest Paths (cont.)



Algorithm for Unweighted Shortest Paths

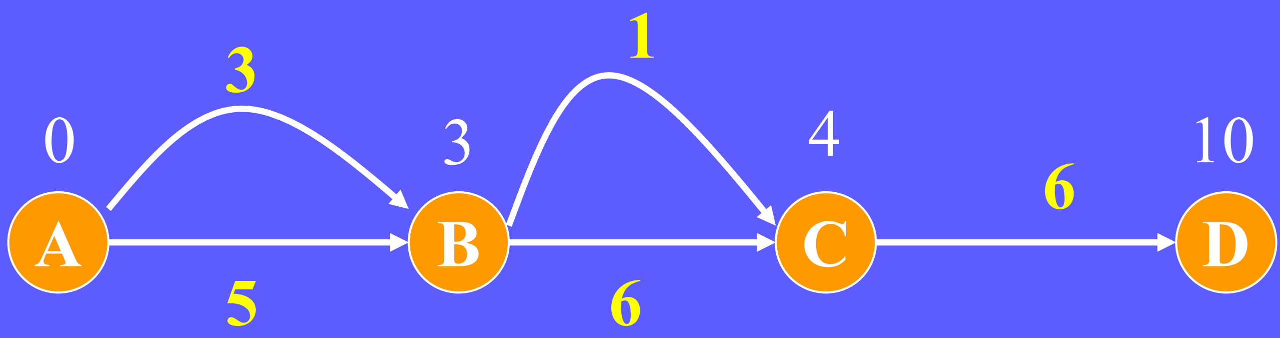
```
void Unweighted(Table T)   /* O(|V|^2)  */
{   int CurrDist;
    Vertex V,W;
     for (CurrDist=0; CurrDist < NumVertex; CurrDist++)
            for each vertex V
                  if (!T[V].know && T[V].Dist == CurrDist)
                  {   T[V].know=True;
                       for each W adjacent to V
                            if (T[W].Dist==infinity)
                            {   T[W].Dist=CurrDist+1;
                                 T[W].Path=V;
                             }
                  }
}
```

Improved Algorithm for Unweighted Shortest Paths

```
void Unweighted (Table T)    /* O(|E|+|V|)  */
{   Queue Q;
    Vertex V,W;
     Q = CreateQueue( NumVertex); MakeEmpty(Q);
     Enque(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);
}
```

Weighted Shortest Paths

- **Case 1**

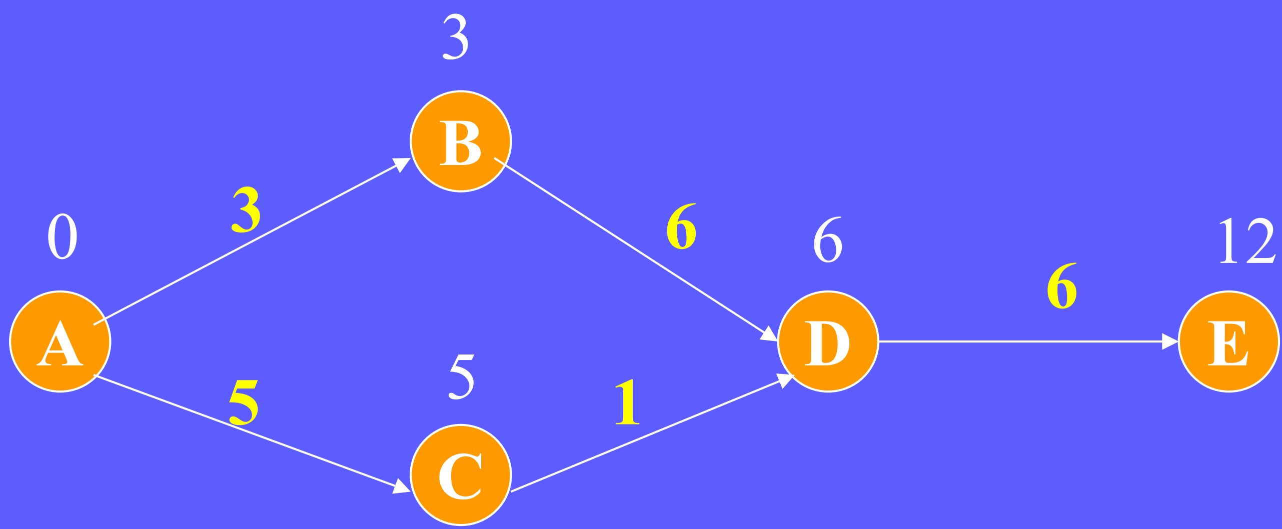


length of shortest path from A to D = 3 + 1 + 6 (greedy algorithm)

Stage	Selected	A	B	C	D
1	A	0	∞	∞	∞
2	B	0	3	∞	∞
3	C	0	3	4	∞
4	D	0	3	4	10

Weighted Shortest Paths (cont.)

- **Case 2**

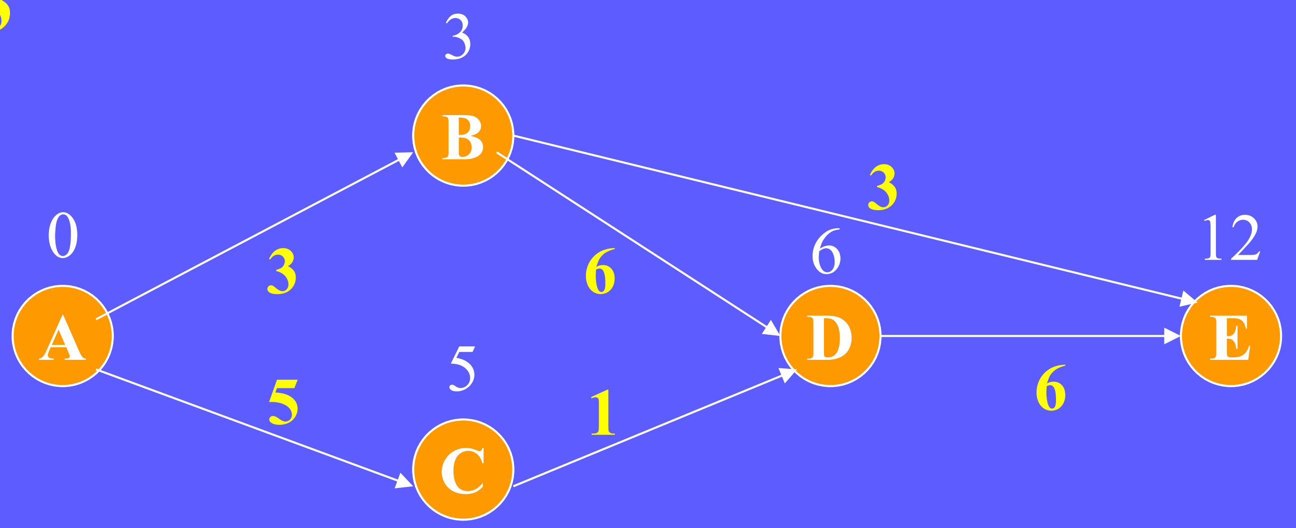


length shortest path from A to E = 5+1+6 (greedy algorithm)

Stage	Selected	A	B	C	D	E
1	A	0	∞	∞	∞	∞
2	B	0	3	∞	∞	∞
3	C	0	3	5	∞	∞
4	D	0	3	5	6	∞
5	E	0	3	5	6	12

Weighted Shortest Paths (cont.)

- **Case 3**



length shortest path from A to E = 3 + 3 = 6 (greedy algorithm)

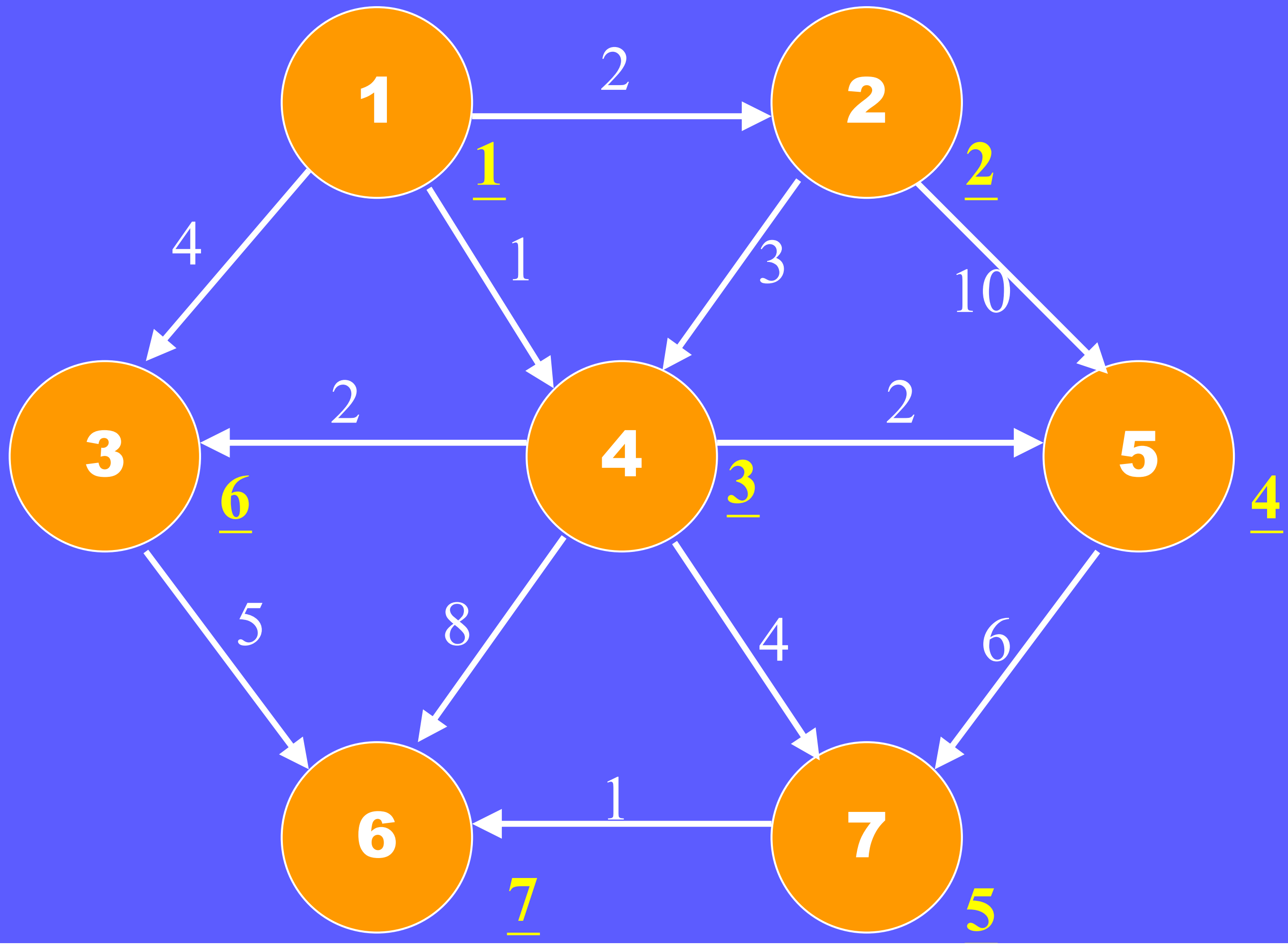
Stage	Selected	A	B	C	D	E
1	A	0	∞	∞	∞	∞
2	B	0	3	∞	∞	∞
3	C	0	3	5	∞	∞
4	D	0	3	5	6	∞
5	E	0	3	5	6	6

Shortest Path in Acyclic Graph

- **Problem:**

  Given a directed weighted acyclic graph *G* = (*V*, *E*) and a vertex *v*

  Find shortest path *v* to all other vertices of *G*



M. K. Shan, CS, NCCU

Shortest Path in Acyclic Graph

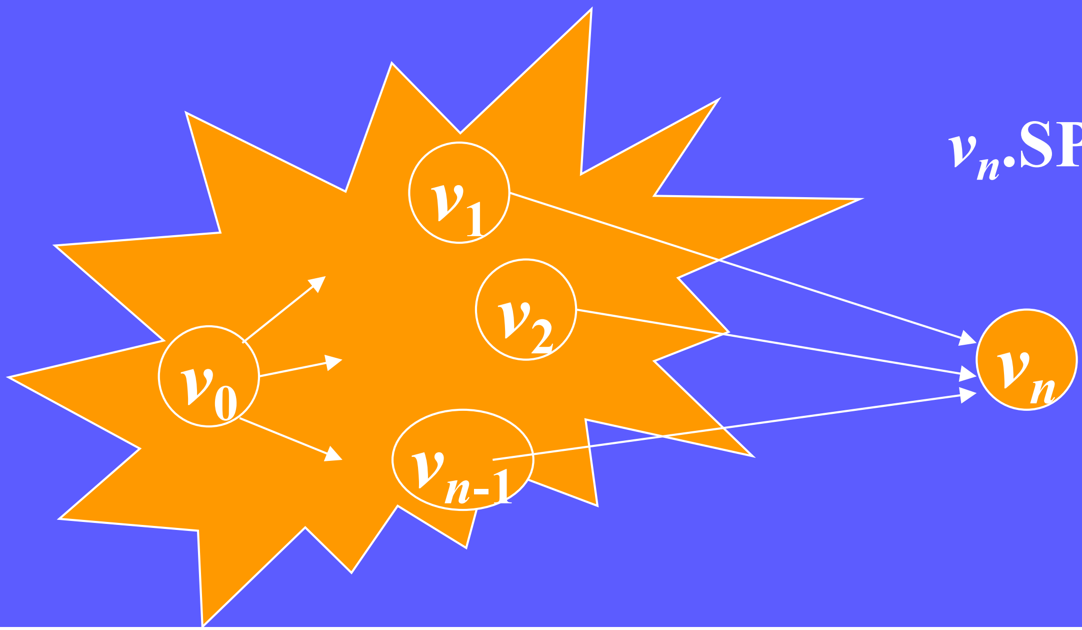
- **Induction hypothesis**

  Given a topological ordering,

  we know how to

  find the lengths of shortest paths v_i.SP from v_0

  to the first n-1 vertices $\{v_i \mid 1 <= i <= n-1\}$

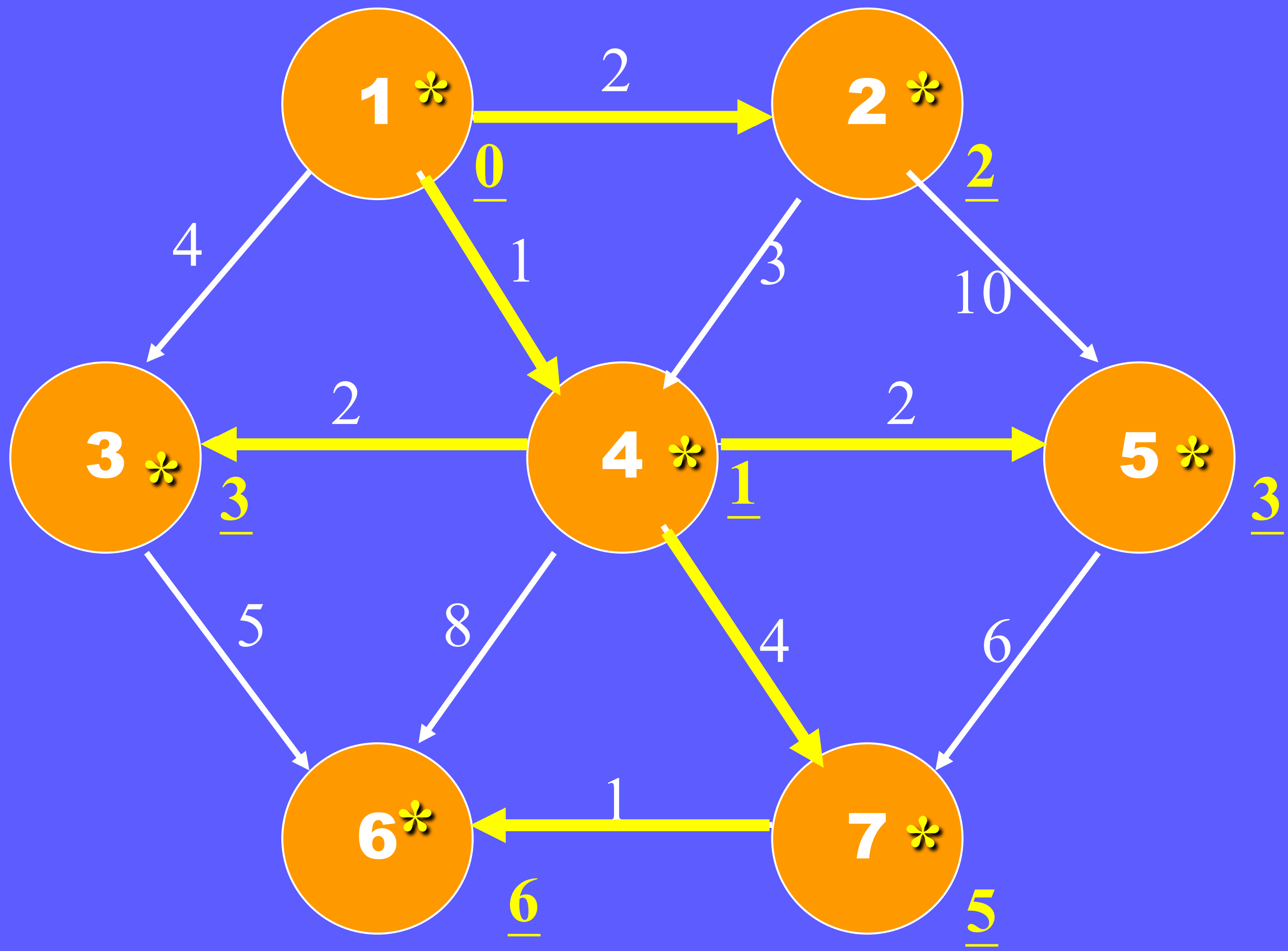


$$v_n.\text{SP} = \min_i \{v_i.\text{SP} + \text{length}(v_i, v_n)\}$$

Shortest Paths in Acyclic Graph (cont.)

- **Selection rule: select vertices in topological ordering (why?)**
- **A vertex *v* is selected, its distance d_v can no longer be lowered. (why?)**
- **Complexity: O(|E|+|V|)**

Shortest Paths in Acyclic Graph (cont.)



Algorithm of Acyclic Shortest Path

```
Algorithm Acyclic_Shortest_Path(G, v, n)
{assume topological ordering has been performed}
Begin
    let z be vertex labeled n
    if z <> v then
       Acyclic(G-z, v, n-1);
       for all w such that (w,z) belongs to E do
            if w.SP+length(w,z) < z.SP then
                z.SP:= w.SP+length(w,z)
            else v.SP:=0
End
```

Improved Algorithm of Acyclic Shortest Path

```
Algorithm Improved_Acyclic_Shortest_Path(G, v, n)
{includes topological ordering}
Begin
   for all vertices w do
      w.Sp=∞;
   initialize v.indegree for all vertices   /*DFS
   for i:=1 to n do
      if  v_i.indegree=0 then put v_i in Queue;
   v.Sp:=0;
   repeat
      remove vertex w from Queue;
      for all edges (w,z) do
         if w.SP+length(w,z) < z.SP then
            z.SP:=w.SP+length(w,z);
         z.indegree:=z.indegree-1;
         if z.indegree=0 then put z in Queue;
    until Queue is empty
End
```

Thinking

- Greedy algorithm: each stage, select the best choice
- Stage?
- Acyclic case: stage = topological ordering = v_0, v_1, …, v_n
- When v_k is considered in stage k

  (1) there are no paths from v_k to vertices with label < k

  (2) there are no paths from vertices with labels > k to v_k

  (shortest path to v_k can no longer be lowered)

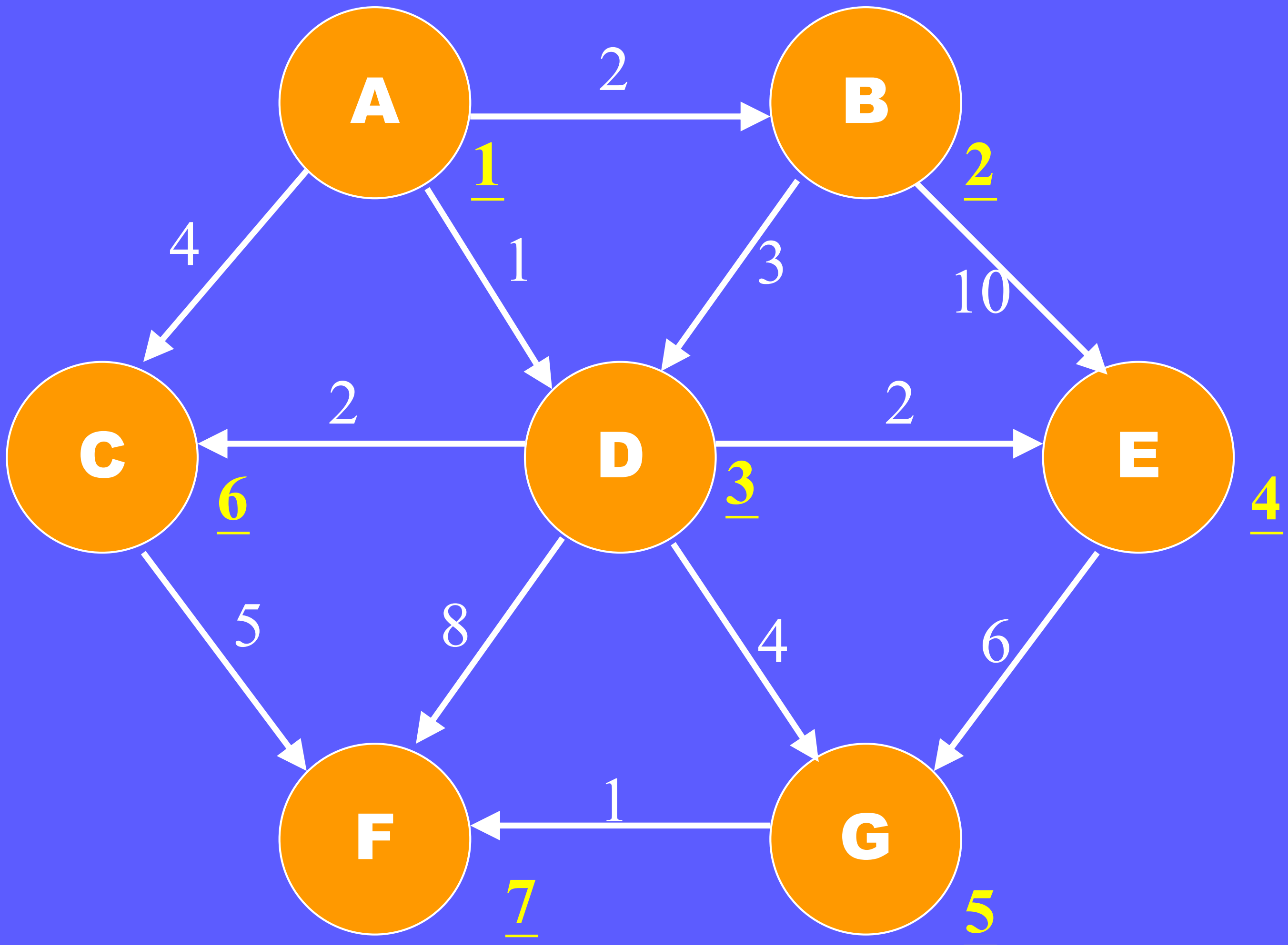
=> consider shortest path stage by stage

Shortest Path in Acyclic Graph

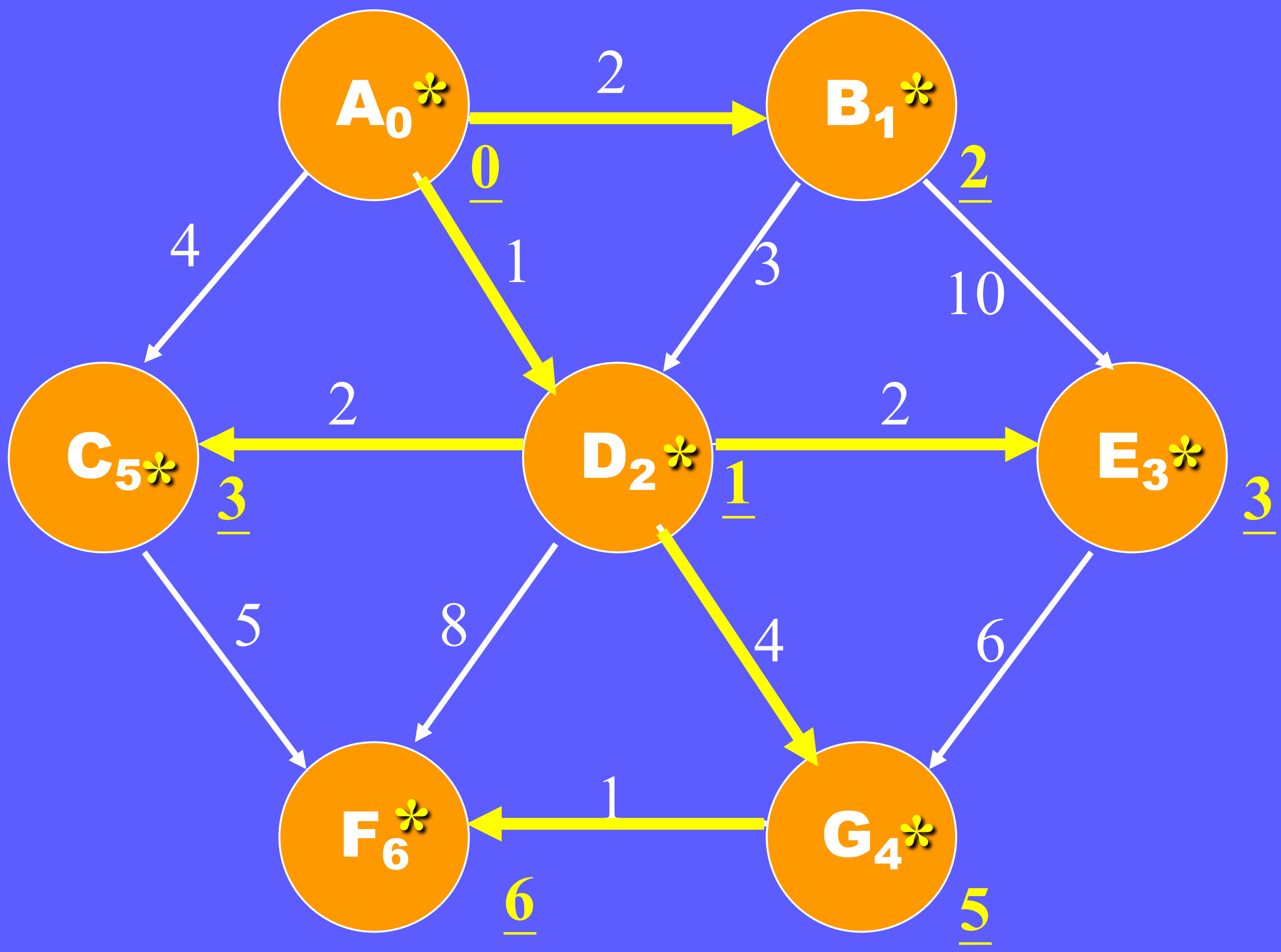
- **Problem:**

  Given a directed weighted acyclic graph $G = (V, E)$ and a vertex v

  Find shortest path v to all other vertices of G



M. K. Shan, CS, NCCU



If *E* is vertex with label 3, then

(1) there are no paths from E to vertices with label < 3 (i.e., A, B, D)

(2) there are no paths from vertices with labels > 3 to *E* (i.e., C, F, G)

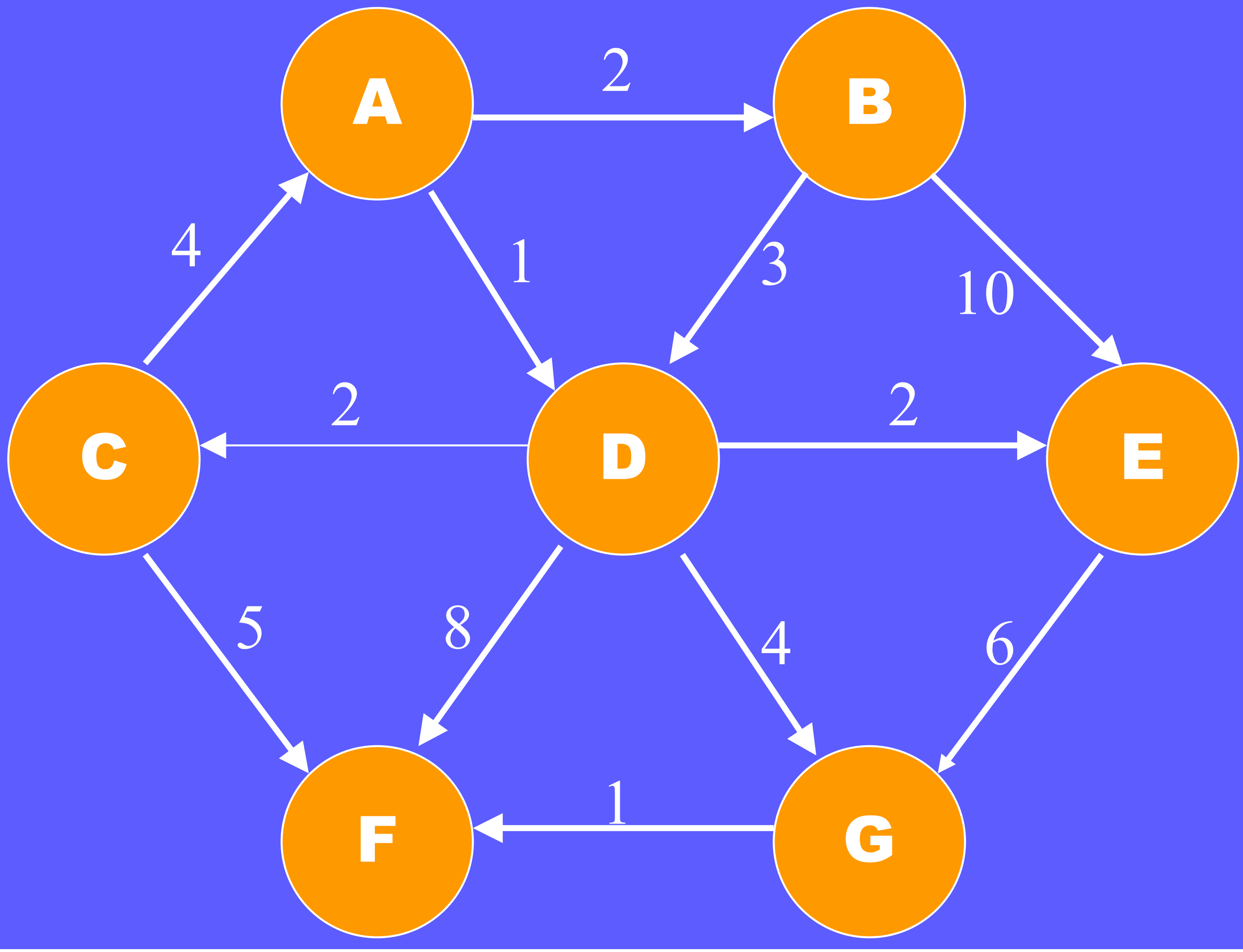
(shortest path to *E* can no longer be lowered)

Shortest Paths in Cyclic Graph

- **Problem:**

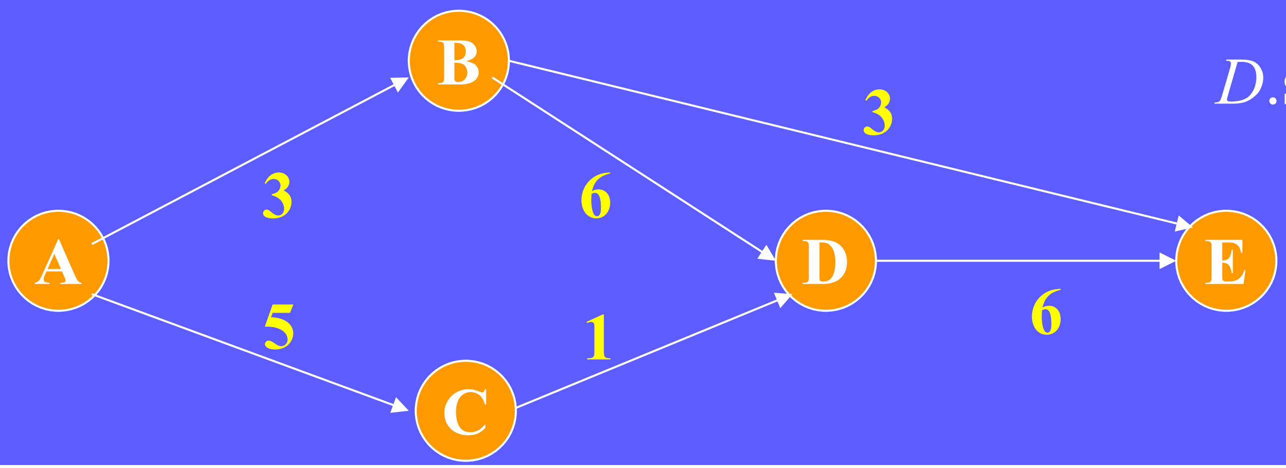
Given a weighted cyclic graph $G = (V, E)$ and a vertex v

Find shortest path v to all other vertices of G

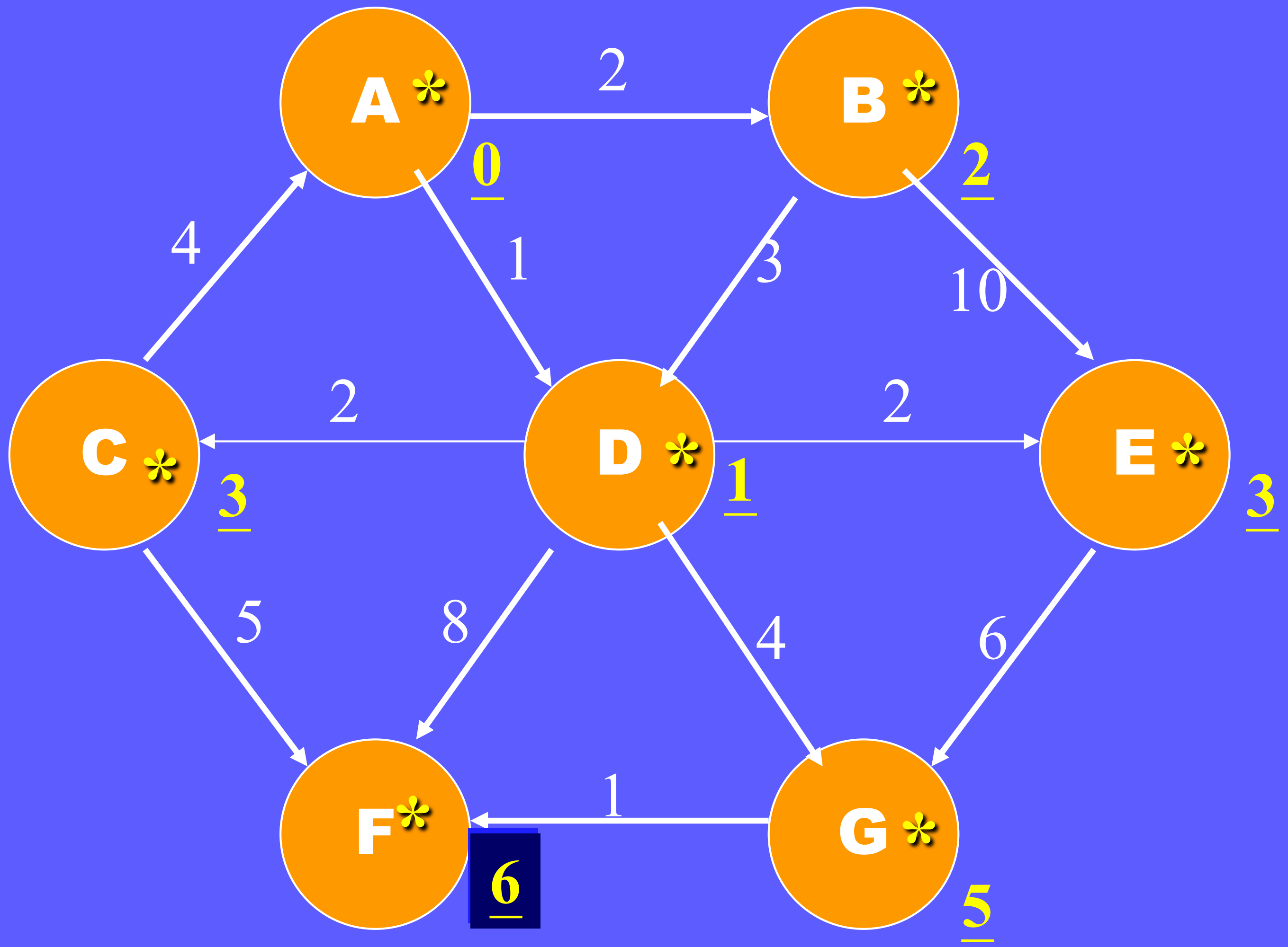


Dijkstra's algorithm

- Greedy algorithm
- *v*.sp: the tentative shortest path length from source *S* to *v*
- Initially, *v*.sp = ∞, for each vertex *v*
- Each stage of Dijkstra's algorithm
  - selects a vertex *v* which has the smallest *v*.sp among unknown vertices and declare *v* as known
  - update *w*.sp, for each edge (*v*, *w*)
- In stage k, after vertex *v* is selected, *v*.sp is determined.



D.sp : tentative sp from *A* to *D*



* Consider the vertices in the order of
vertex A, D, B, E, C, G, F
whose shortest path from source is 0, 1, 2, 3, 3, 5, 6 respectively

Dijkstra's algorithm (cont.)

- **Stage: consider vertices in the order imposed by lengths of shortest paths from *S***
- **Induction hypothesis**

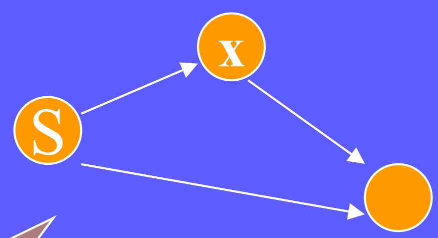
we know

(1) the *k* vertices that are closest to S

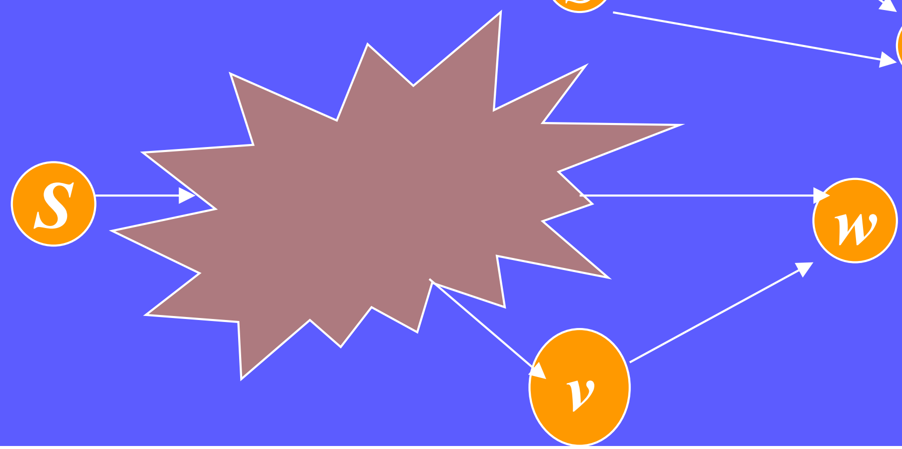
(2) lengths of the shortest paths to them

- **Base: (1) x=min {<S,x>}**

(2)



- **Induction**



Dijkstra's algorithm (cont.)

- *v*.sp: the tentative shortest path length from source *S* to *v*
- In stage k, after vertex *v* is selected, *v*.sp is determined.
- In stage k, if vertex *v* is selected, the shortest path from *S* to *v* can go thru only the vertices in S_k

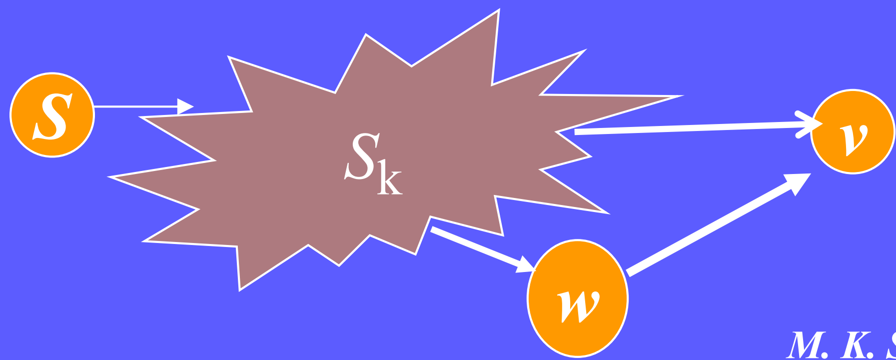
  * S_k: *k*-th nearest vertices from *S*

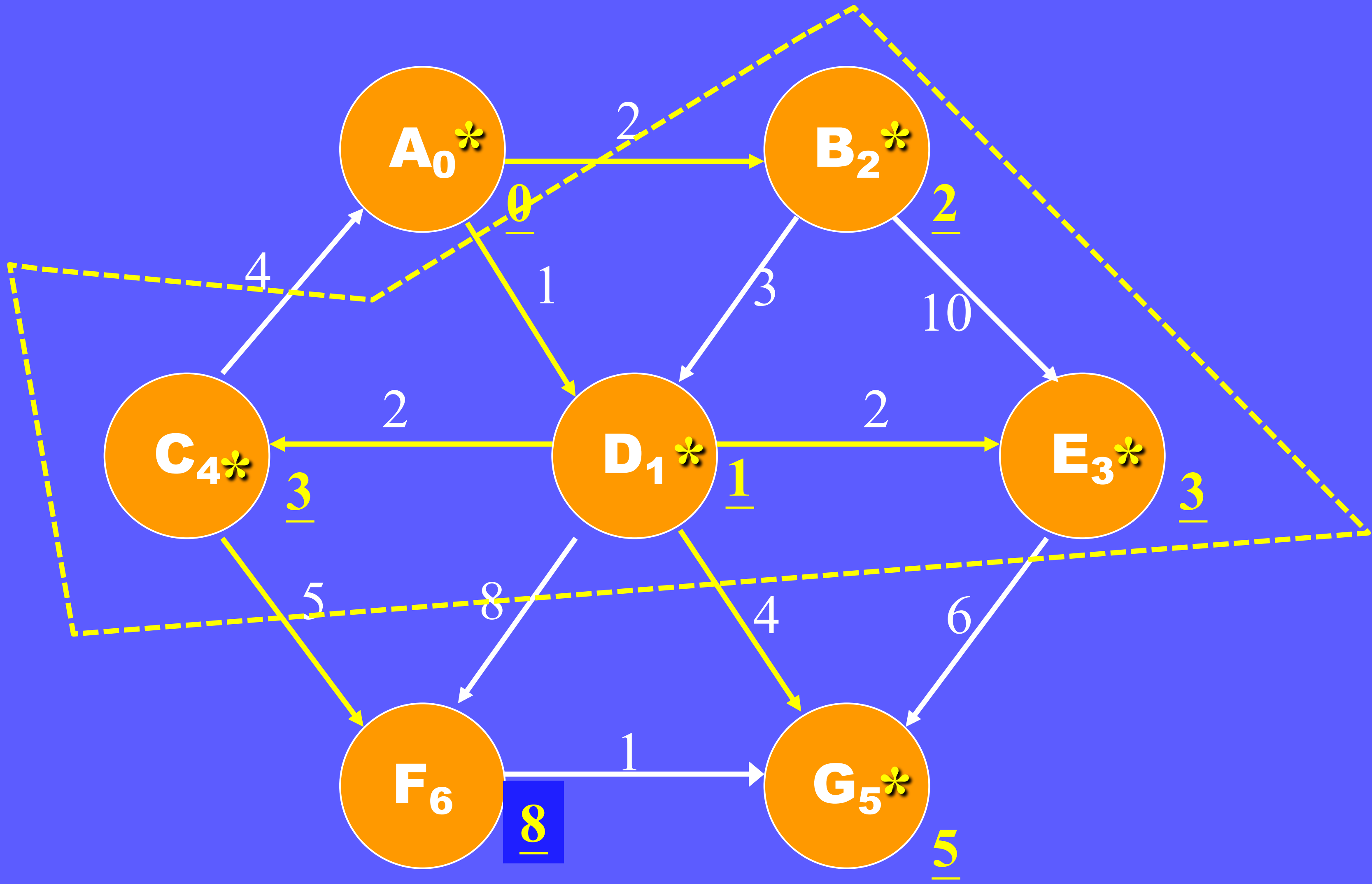
  Proof: by contradiction

  □ if there exists vertices not in S_k, say *w*, such that

  $$w.\text{sp} + \text{length}(w,v) < v.\text{sp}$$

  then stage *k* should select vertex *w*, rather than *v*.





- In stage 5, after *G* is selected, *G*.sp is determined.
  the shortest path from *A* to *G* can go thru only vertices in
  S_5 = {A, D, B, E, C}}

Dijkstra's Algorithm for Weighted Shortest Paths (version 1)

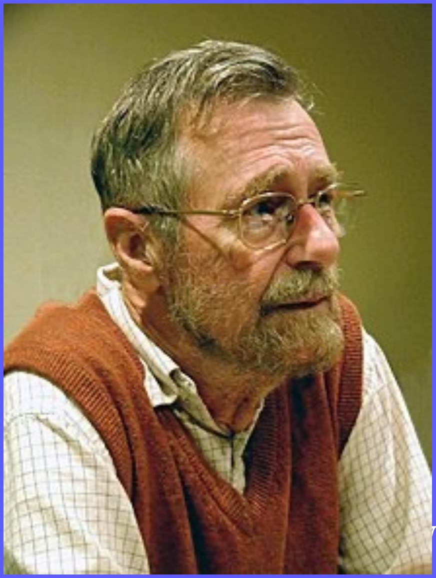
```
Algorithm General_Case
Begin
    for all vertices w do
        w.mark:=false;
        w.sp:=∞;
    v.sp:=0;
    while there exists an unmarked vertex do
        let w be an unmarked vertex such that w.sp in minimal;
        w.mark:=true;
        for all edges (w,z) such that z is unmarked do
          if w.SP + length (w,z) < z.SP then
            z.SP:=w.SP+length(w,z)
End
```

Dijkstra's Algorithm for Weighted Shortest Paths (version 2)

```
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[V].Dist+Cvw < T[W].Dist)
            {   Descrease  T[W].Dist  to  (T[V].Dist+Cvw);
                T[W].Path=V;
            }
    }
}
```

Complexity of Dijkstra's Algorithm

- Complexity depends on how the table is implemented
  - unsorted table: $O(|E|+|V|^2)$, $O(|V|)$ find minimum, $O(|V|^2)$total of find minimum, $O(|E|)$update
  - priority queue: $O(|E|\log|V| + |V|\log|V|)$
- invented by Edsger W. Dijkstra in 1956 and published three years later.



Currently, there is no algorithm
in which finding the
single-source single-destination shortest paths
is faster than
single-source all-destination shortest paths.

All Pairs Shortest Path

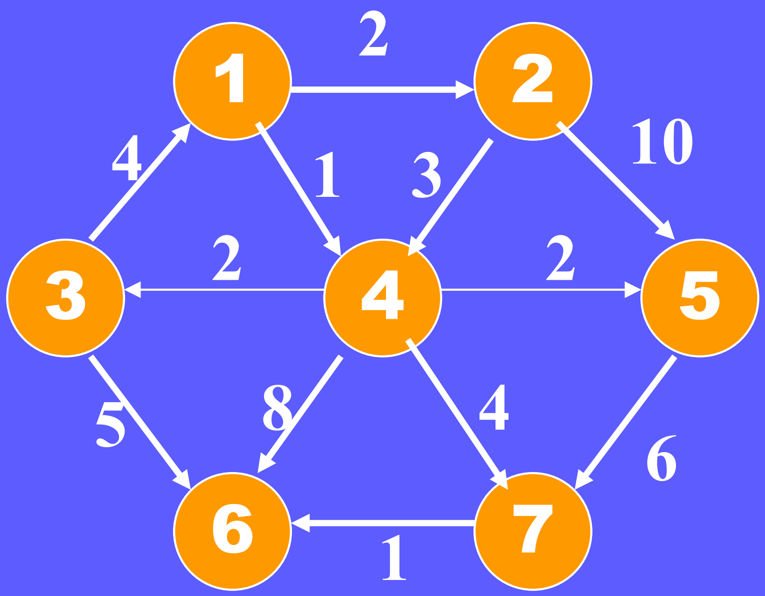
(pp. 212~214)

All-Pairs Shortest Paths

Problem:

Given a weighted graph $G = (V, E)$ with non-negative weights

Find shortest paths between all pair of vertices.

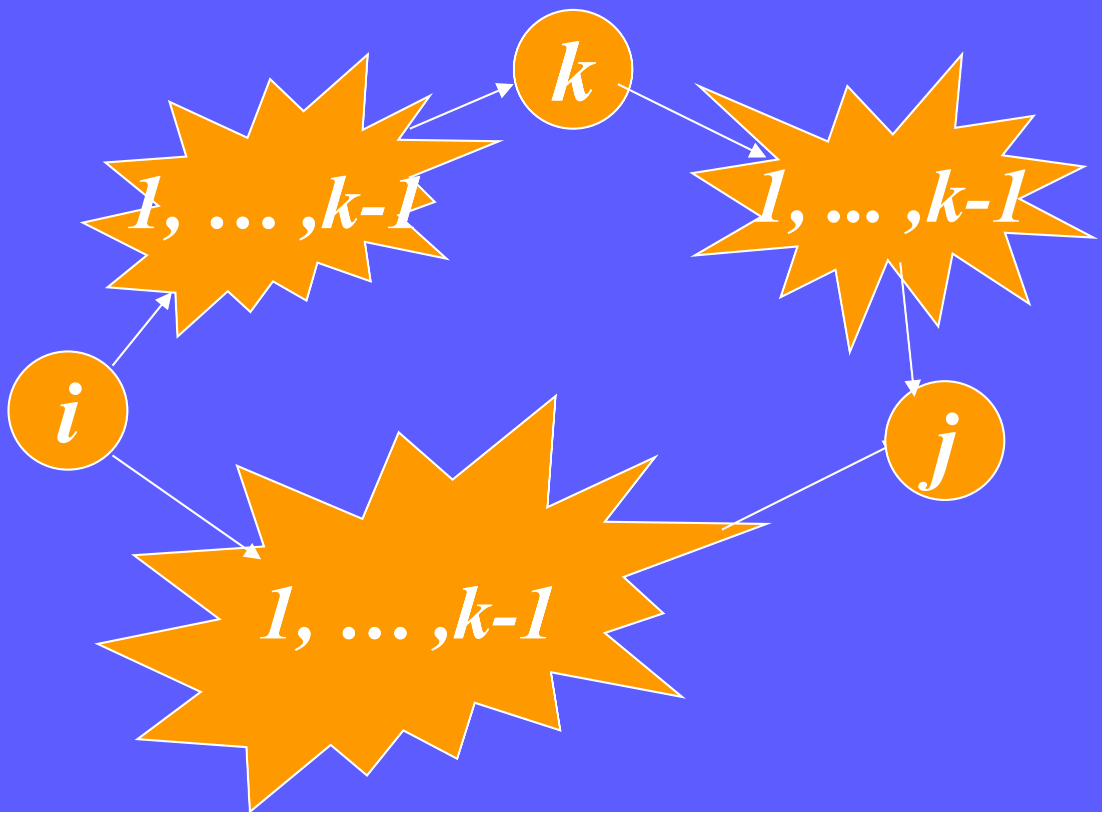


D_0	1	2	3	4	5	6	7
1	0	2	∞	1	∞	∞	∞
2	∞	0	∞	3	10	∞	∞
3	4	∞	0	∞	∞	5	∞
4	∞	∞	2	0	2	8	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

D_7	1	2	3	4	5	6	7
1	0	2	3	1	3	6	5
2	9	0	5	3	10	8	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	5	4
5	∞	∞	∞	∞	0	7	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

Recurrence Relations for All-Pairs Shortest Paths

$$D_k(i,j) = \begin{cases} D_{k-1}(i,k) + D_{k-1}(k,j) \\ D_{k-1}(i,j) \end{cases}$$



Algorithm of All Pairs Shortest Path

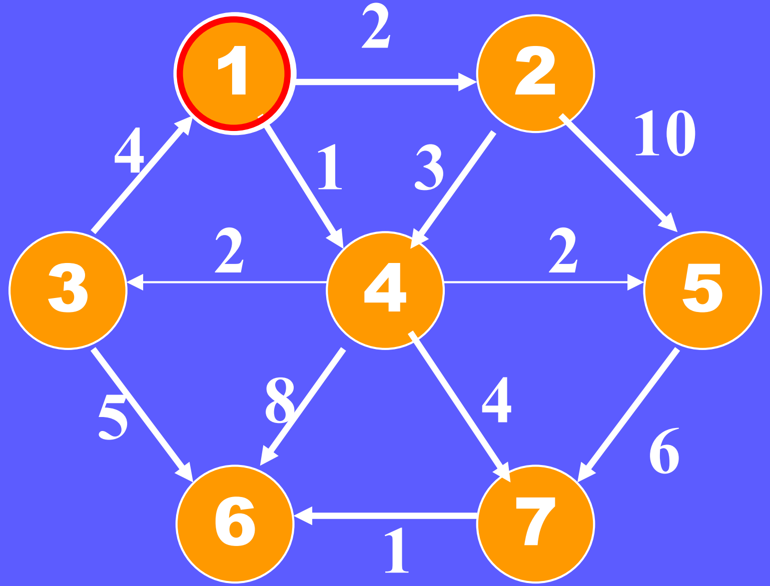
```
void  Allpairs(Const TwoDimArray A, TwoDimArray D,
                              TweoDimArray Path, int N)
{   int i, j, k;
    for (i=0; i <n; i++)
        for (j=0; j <N; j++)
        {       D[i][j] = A[i][j];
                Path[i][j] = -1;
        };
    for (k=0; k <N; k++)
        for (i=0; 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][j]=k;
                   }
}
```

D_0	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

$$D_1(3,2) = \begin{cases} D_0(3,1) + D_0(1,2) \\ D_0(3,2) \end{cases} = 6$$

$$D_1(3,4) = \begin{cases} D_0(3,1) + D_0(1,4) \\ D_0(3,4) \end{cases} = 5$$

…



D_1	1	2	3	4	5	6	7
1	0	2	∞	1	∞	∞	∞
2	∞	0	∞	3	10	∞	∞
3	4	<u>6</u>	0	<u>5</u>	∞	5	∞
4	∞	∞	2	0	2	8	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

M. K. Shan, CS, NCCU

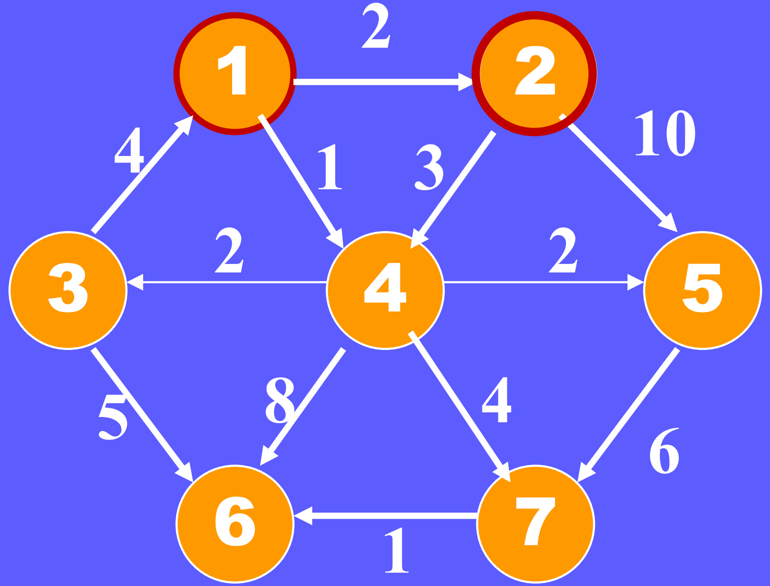
D_1	1	2	3	4	5	6	7
1	0	2	∞	1	∞	∞	∞
2	∞	0	∞	3	10	∞	∞
3	4	6	0	5	∞	5	∞
4	∞	∞	2	0	2	8	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

$$D_2(1,5) = \begin{cases} D_1(1,2) + D_1(2,5) \\ D_1(1,5) \end{cases} = 12$$

$$D_2(3,5) = \begin{cases} D_1(3,2) + D_1(2,5) \\ D_1(3,5) \end{cases} = 16$$

$$D_2(1,4) = \begin{cases} D_1(1,2) + D_1(2,4) \\ D_1(1,4) \end{cases} = 1$$

...



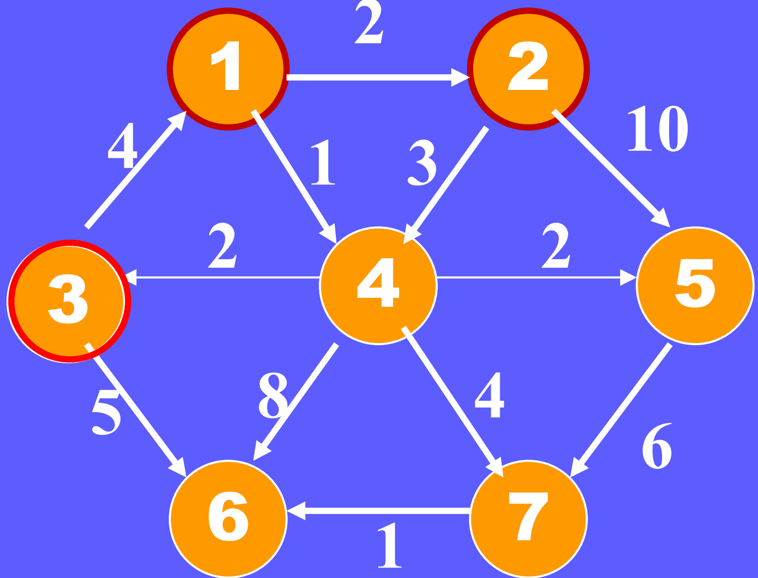
D_2	1	2	3	4	5	6	7
1	0	2	∞	1	<u>12</u>	∞	∞
2	∞	0	∞	3	10	∞	∞
3	4	6	0	5	<u>16</u>	5	∞
4	∞	∞	2	0	2	8	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

D_2	1	2	3	4	5	6	7
1	0	2	∞	1	12	∞	∞
2	∞	0	∞	3	10	∞	∞
3	4	6	0	5	15	5	∞
4	∞	∞	2	0	2	8	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

$$D_3(4,1) = \begin{cases} D_2(4,3) + D_2(3,1) \\ D_2(4,1) \end{cases} = 6$$

$$D_3(4,6) = \begin{cases} D_2(4,3) + D_2(3,6) \\ D_2(4,6) \end{cases} = 7$$

…



D_3	1	2	3	4	5	6	7
1	0	2	∞	1	12	∞	∞
2	∞	0	∞	3	10	∞	∞
3	4	6	0	5	15	5	∞
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

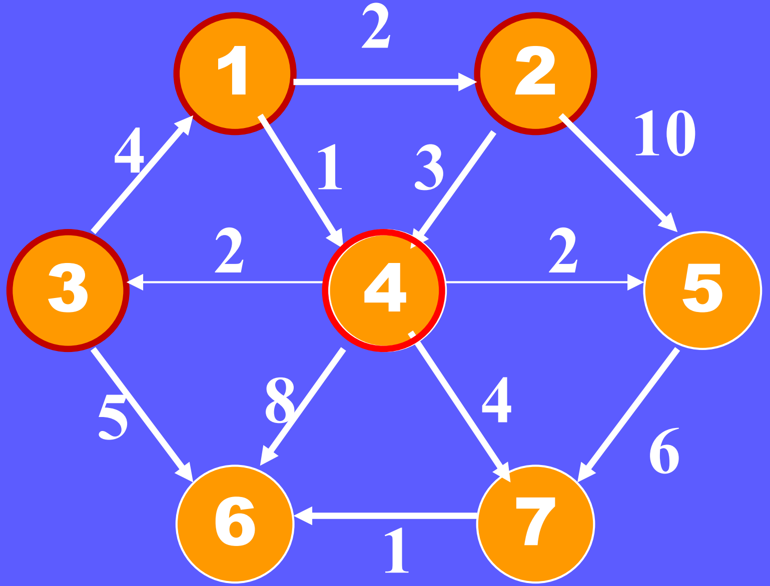
D_3	1	2	3	4	5	6	7
1	0	2	∞	1	12	∞	∞
2	∞	0	∞	3	10	∞	∞
3	4	6	0	5	15	5	∞
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

$$D_4(1,3) = \begin{cases} D_3(1,4) + D_3(4,3) \\ D_3(1,3) \end{cases} = 3$$

$$D_4(2,3) = \begin{cases} D_3(2,4) + D_3(4,3) \\ D_3(2,3) \end{cases} = 5$$

$$D_4(3,7) = \begin{cases} D_3(3,4) + D_3(4,7) \\ D_3(3,7) \end{cases} = 9$$

$$\cdots$$

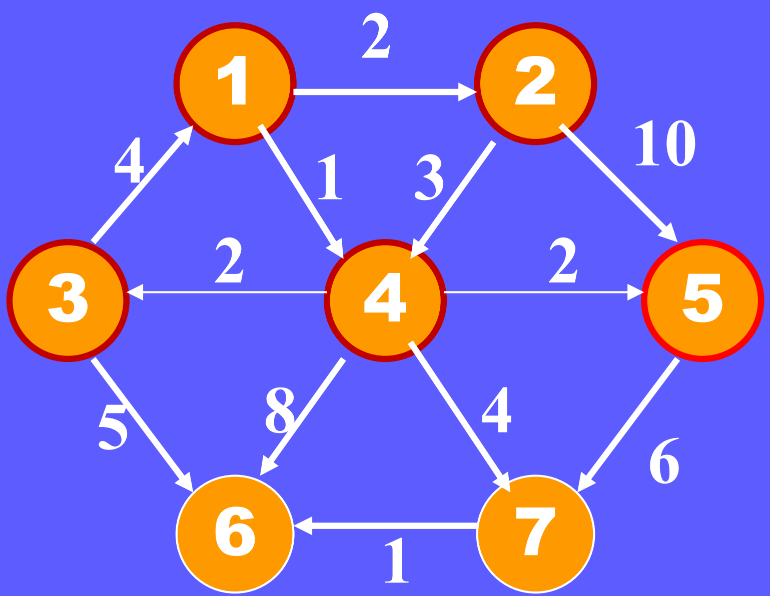


D_4	1	2	3	4	5	6	7
1	0	2	3	1	3	9	5
2	9	0	5	3	10	11	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

D_4	1	2	3	4	5	6	7
1	0	2	3	1	3	9	5
2	9	0	5	3	10	11	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

$$D_5(2,7) = \begin{cases} D_4(2,5) + D_4(5,7) \\ D_4(2,7) \end{cases} = 7$$

$$\cdots$$

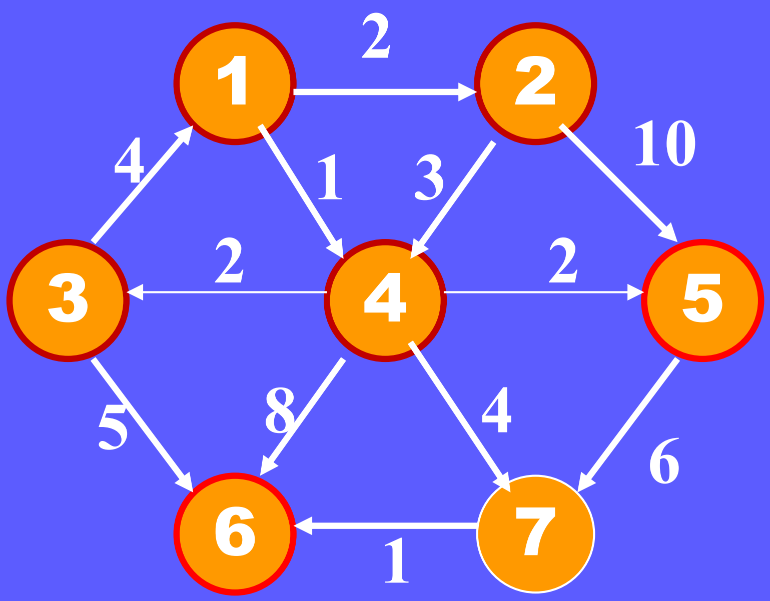


D_5	1	2	3	4	5	6	7
1	0	2	3	1	3	9	5
2	9	0	5	3	10	11	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

D_5	1	2	3	4	5	6	7
1	0	2	3	1	3	9	5
2	9	0	5	3	10	11	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

$$D_6(3,7) = \begin{cases} D_5(3,6) + D_5(6,7) \\ D_5(3,7) \end{cases} = 9$$

…



D_6	1	2	3	4	5	6	7
1	0	2	3	1	3	9	5
2	9	0	5	3	10	11	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

D_6	1	2	3	4	5	6	7
1	0	2	3	1	3	9	5
2	9	0	5	3	10	11	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	7	4
5	∞	∞	∞	∞	0	∞	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

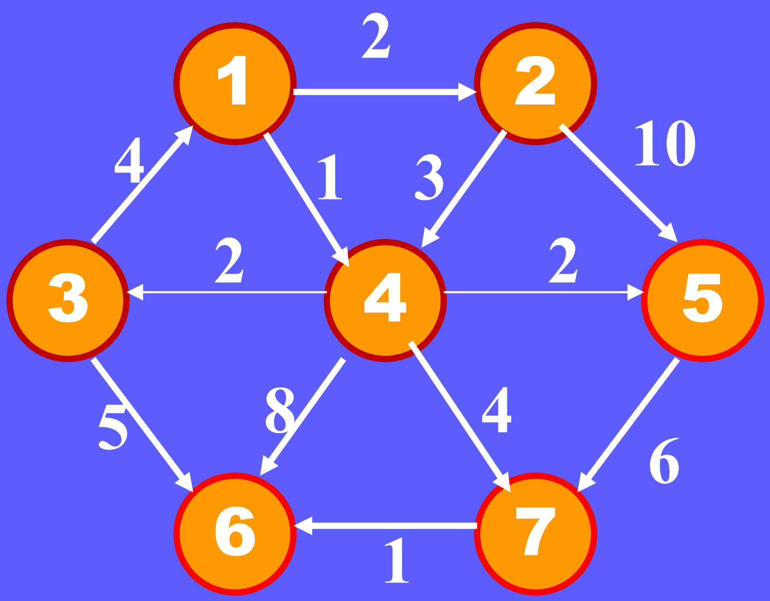
$$D_7(1,6) = \begin{cases} D_6(1,7) + D_6(7,6) \\ D_6(1,6) \end{cases} = 6$$

$$D_7(2,6) = \begin{cases} D_6(2,7) + D_6(7,6) \\ D_6(2,6) \end{cases} = 8$$

$$D_7(4,6) = \begin{cases} D_6(4,7) + D_6(7,6) \\ D_6(4,6) \end{cases} = 5$$

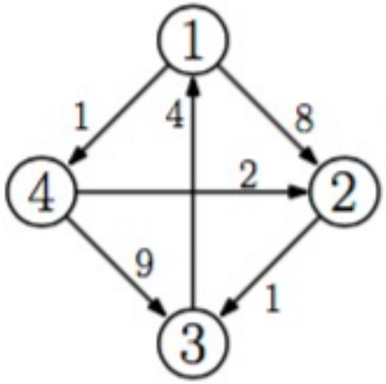
$$D_7(5,6) = \begin{cases} D_6(5,7) + D_6(7,6) \\ D_6(5,6) \end{cases} = 7$$

$$\cdots$$

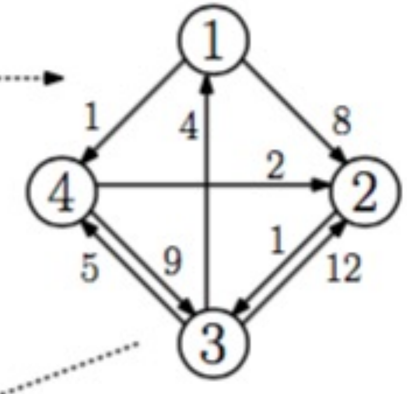


D_7	1	2	3	4	5	6	7
1	0	2	3	1	3	<u>6</u>	5
2	9	0	5	3	10	<u>8</u>	7
3	4	6	0	5	7	5	9
4	6	8	2	0	2	<u>5</u>	4
5	∞	∞	∞	∞	0	<u>7</u>	6
6	∞	∞	∞	∞	∞	0	∞
7	∞	∞	∞	∞	∞	1	0

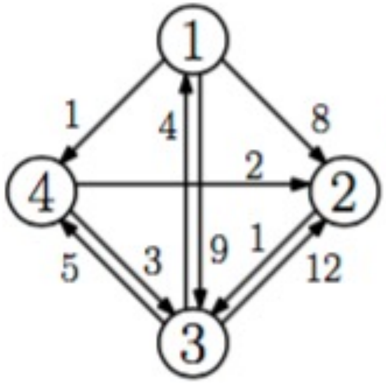
$$D_{k,i,j} = \min\begin{cases} D_{k-1,i,j} \\ D_{k-1,i,k} + D_{k-1,k,j} \end{cases}$$



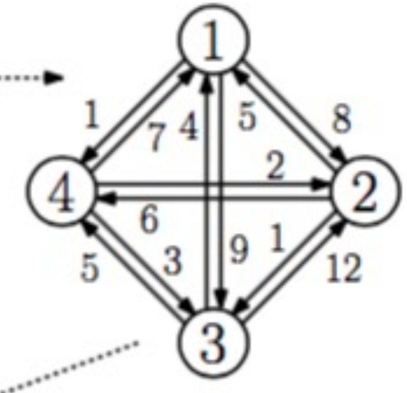
$$d^{(0)} \quad \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$



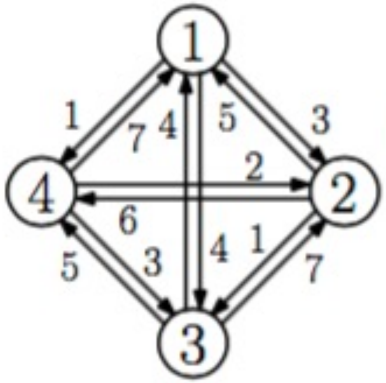
$$d^{(1)} \quad \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$



$$d^{(2)} \quad \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$



$$d^{(3)} \quad \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$



$$d^{(4)} \quad \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$\text{final} \quad \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

Floyd–Warshall algorithm

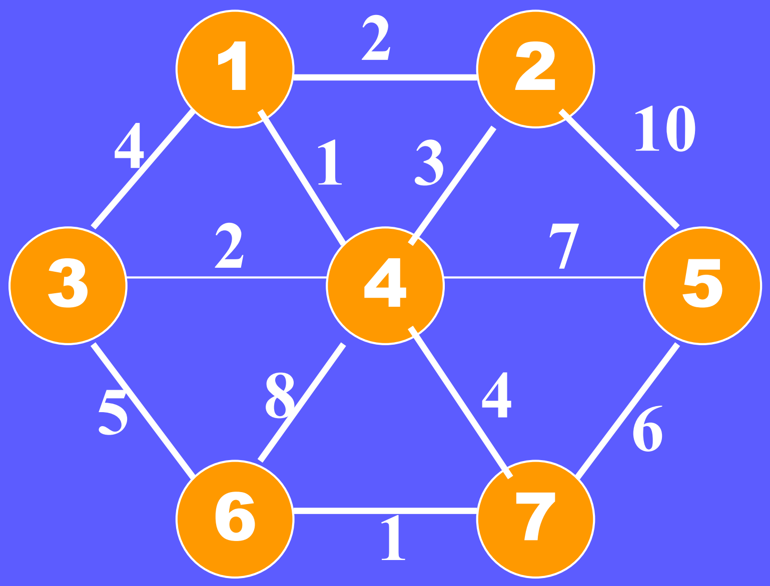
- also known as Floyd's algorithm, the Roy–Warshall algorithm, the Roy–Floyd algorithm, or the WFI algorithm
- published by Robert Floyd in 1962.
- However, it is essentially the same as algorithms previously published by Bernard Roy in 1959 and also by Stephen Warshall in 1962 for finding the transitive closure of a graph

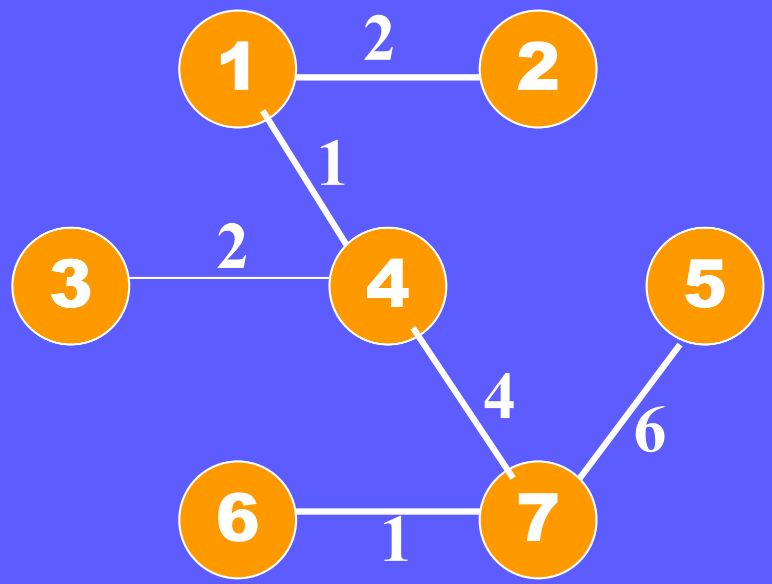
Minimum Spanning Tree

(pp. 208~212)

Minimum Spanning Tree

- Spanning Tree: a tree formed from graph edges that connects all the vertices
- Minimum Spanning Tree: spanning tree with lowest total cost
- A minimum spanning tree exists iff G is connected



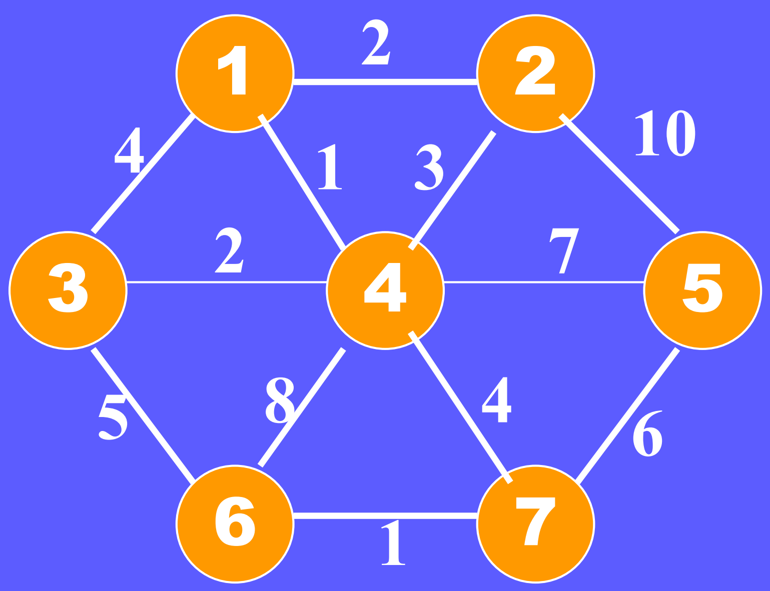


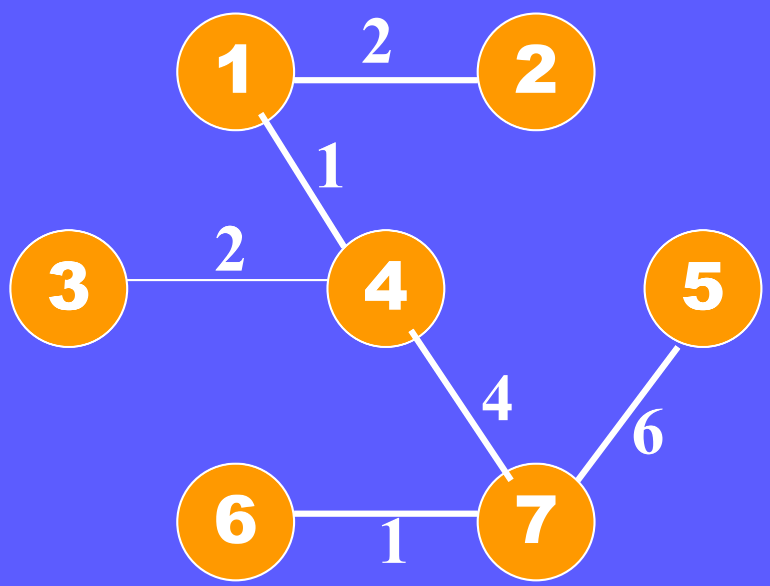
Minimum Spanning Tree (cont.)

- A minimum spanning tree exists iff G is connected
- Problem

  Given an undirected connected weighted graph $G = (V, E)$

  Find a spanning tree T of G of minimum cost.

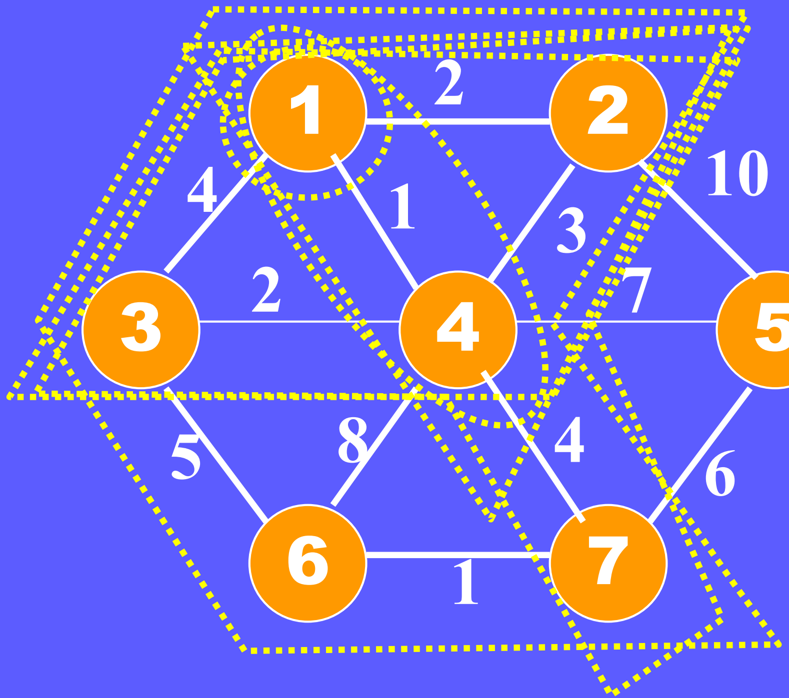


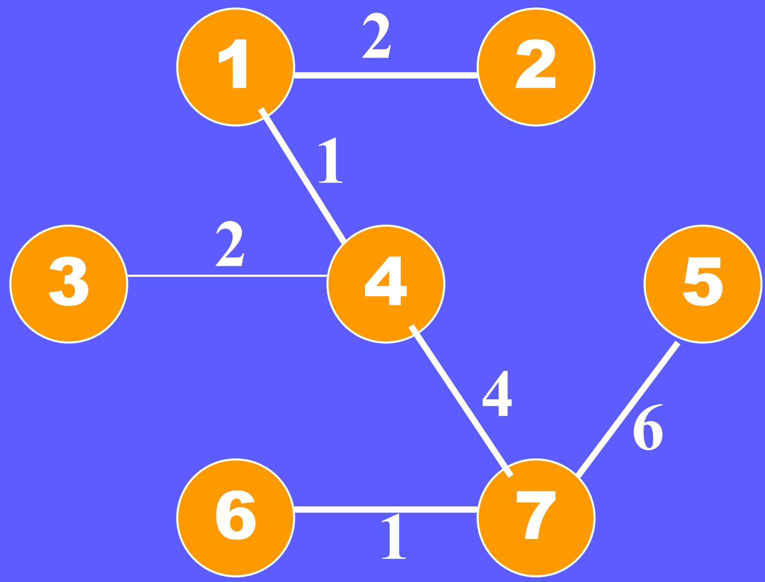


M. K. Shan, CS, NCCU

Principle of Prim's Algorithm

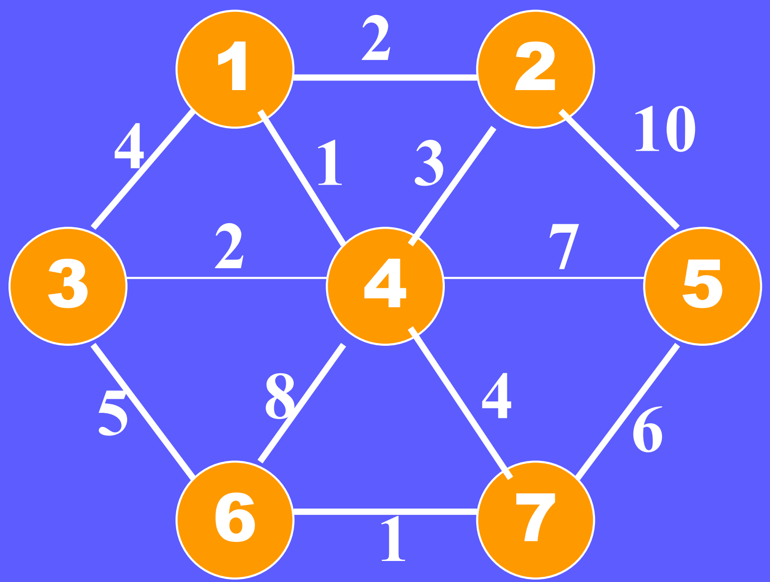
- **Greedy algorithm (invented by Vojtěch Jarník in 1930 and rediscovered by Prim in 1957 and Dijkstra in 1959)**
- **At each stage, a new vertex is added to the tree**
- **The new selected vertex *v*: smallest cost (*u*,*v*) where *u* in the tree, *v* is not.**

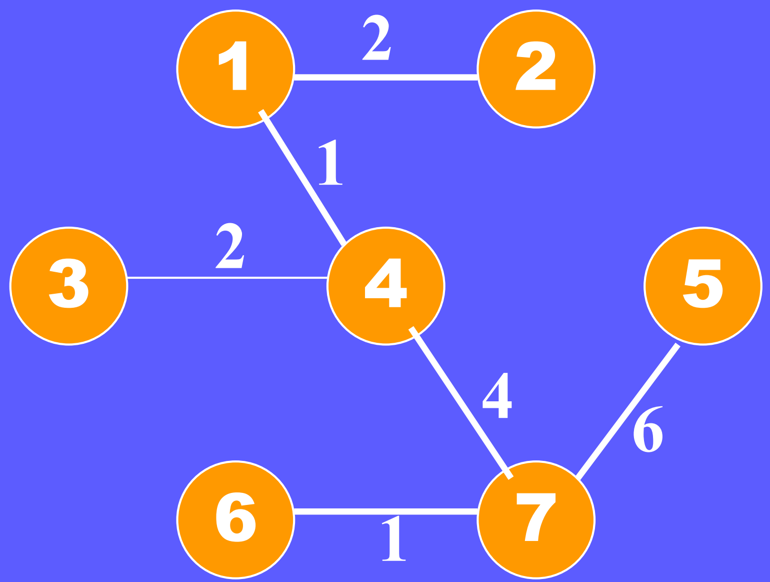




Principle of Kruskal's Algorithm

- Greedy algorithm
- At each stage, a new edge is added to the tree
- The selected edge is the smallest cost among unselected edge and it does not cause a cycle.



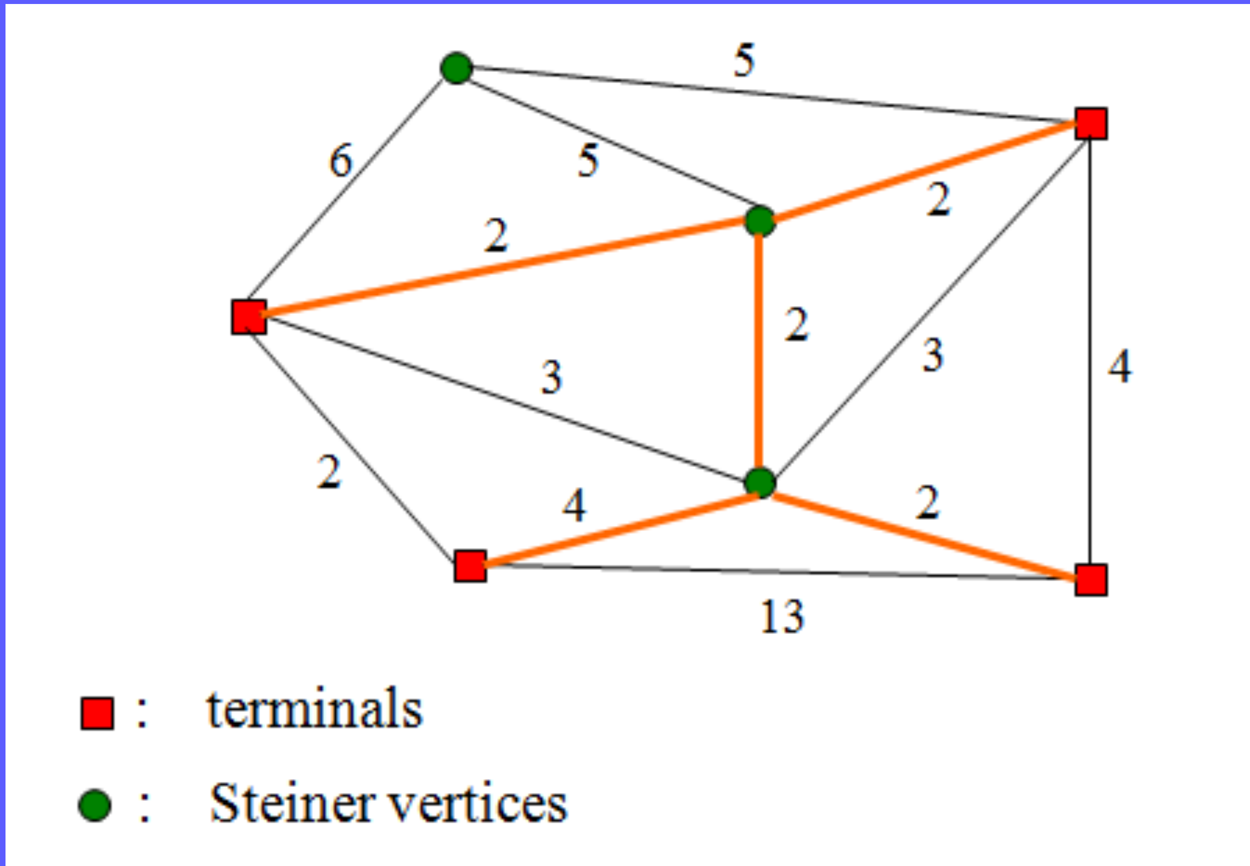


Kruskal's Algorithm

```
void kruskal(Graph G)
{    int EdgesAccepted;     DisjSet S;          PriorityQueue H;
     Vertex u,v;      SetType uset, vset;     Edge E;
     Initialize(S);
     ReadGraphIntoHeapArray(G,H);      BuildHeap(H);
     EdgesAccepted=0;
     While (EdgeAccepted < NumVertex -1)
     {     E = DeleteMin(H);   /*  E = (u,v)   */
          uset=Find(u,S);   /* find the set which contains U */
          vset=Find(v,S);
          if (uset != vset)   /* if not cycle */
          {  EdgesAccepted++;
             SetUnion(S, uset, vset);
           }
      }
}
```

Variations of Minimum Spanning Tree

- ***k*-Minimum Spanning Tree**
- **Steiner Tree**



Matching

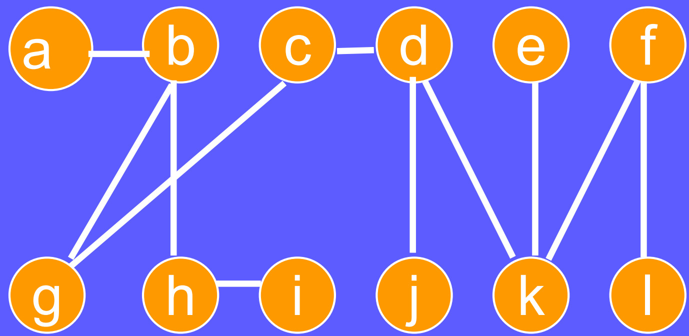
(pp. 234~238)

COOL3C.COM | 作者：MASH YANG

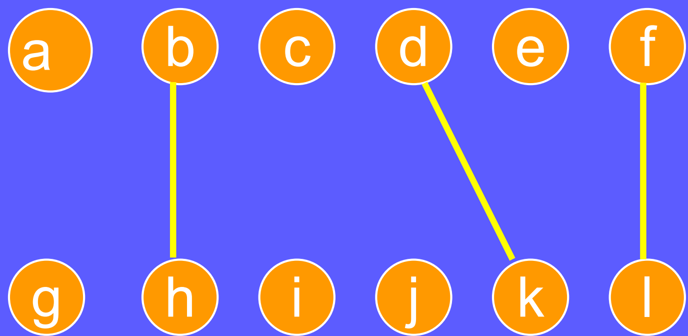
臉書推出好友暗戀配對功能 Secret Crush：市場所有交友APP非常害怕 - 瘾科技 Cool3c

Matching

- **Given an undirected graph G=(V, E)**
  - **matching: a set of edges no two of which have a vertex in common**

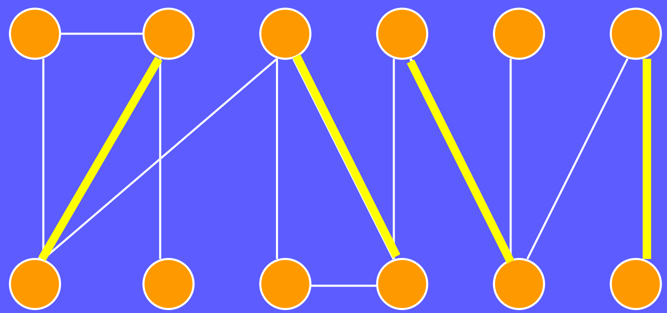


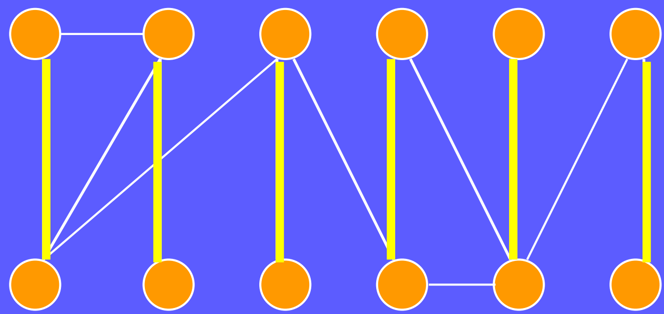
matching



Matching (cont.)

- **Given an undirected graph G=(V, E)**
  - **maximal matching: a matching that cannot be extended by the addition of an edge** (極大值)
  - **maximum matching: a matching with maximum number of edges** (最大值)
  - **perfect matching: a matching in which all vertices are matched**





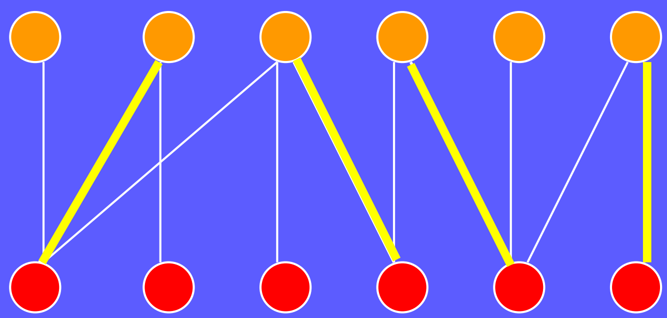
Matching (cont.)

- **Matching in general graphs is a difficult problem**
- **Two specific matching problem**
  - **finding perfect matching in a very dense graph**
  - **finding matching in bipartite graph**

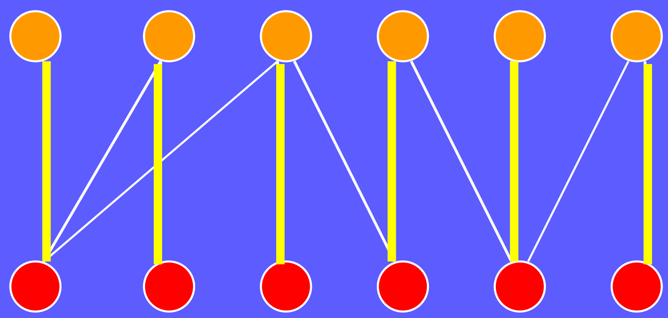
Bipartite Matching

- **Bipartite graph G= (V, E, U),**
  U, V: two disjoint sets of vertices,
  E: a set of edges connecting vertices between V & U

maximal matching

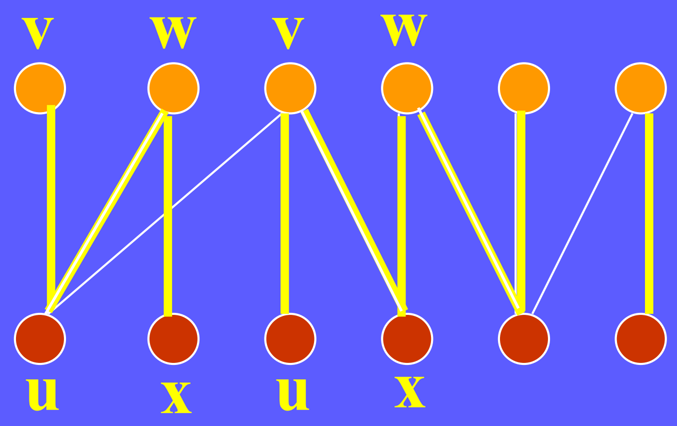


maximum matching



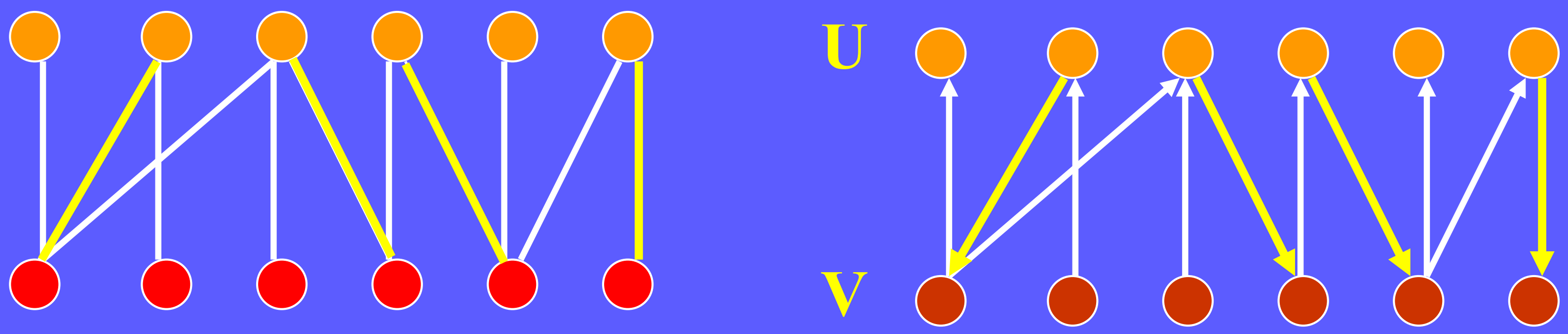
Observation

- start with an unmatched vertex v
- all v's neighbors are matched (maximal), try to break up a match.
- choose another vertex u, adjacent to v, matched with w
- break up match between u & w, match between u & v
- if w is adjacent to a unmatched vertex x, match w with x
  else continue by breaking matches



Formalization: Alternating Path

- Alternating Path for a given matching M:
  - a path from an unmatched vertex v in V to an unmatched vertex u in U
  - edges of P are alternatively in E-M and in M
- Transform undirected graph G to a directed graph G' by
  - directing edges in M to point from U to V
  - directing edges not in M to point from V to U
- Alternating path can be found by graph traversal



Alternating-Path Theorem

- **a matching is maximum iff it has no alternating paths**

Algorithm for Bipartite Matching

- Start with greedy algorithm, adding as many edges to the matching as possible, until maximal matching
- Repeat
  - search for an alternating path
  - modify the matching

  Until no more alternating paths

- complexity: O(|V| * (|V| + |E|))
  - DFS: O(|V| + |E|)
  - Iteration: |V| (Each stage extend one matching, at most |V|/2 matching)

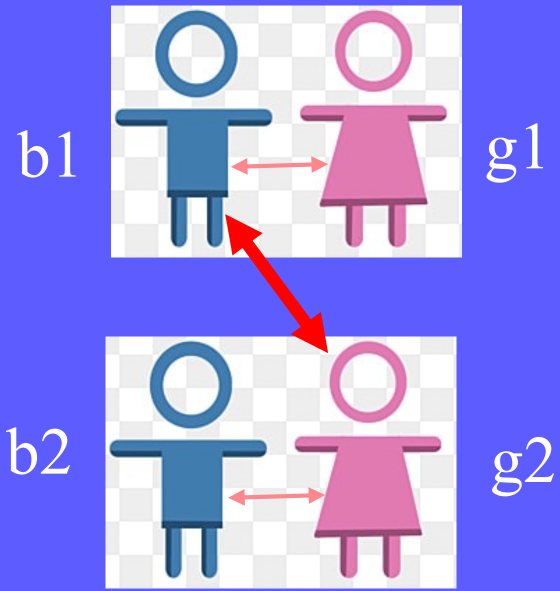
Variations of Bipartite Matching

- Maximum weighted matching (vs. maximum cardinality matching): Hungarian algorithm $O(n^3)$
- One-to-Many Matching: Integer Programming
- Matching on general graph, rather than bipartite.

Variations of Bipartite Matching (cont.)

■ Stable Marriage

- ☐ Bipartite graph where each boy has ranked each girl & each girl do the same to each boy.
- ☐ Stable
  - No parties can be unhappy enough to seek to break the matching
  - There are no marriage of the form (b1, g1) & (b2, g2), where b1 & g2 in fact prefer each other to their own spouses.

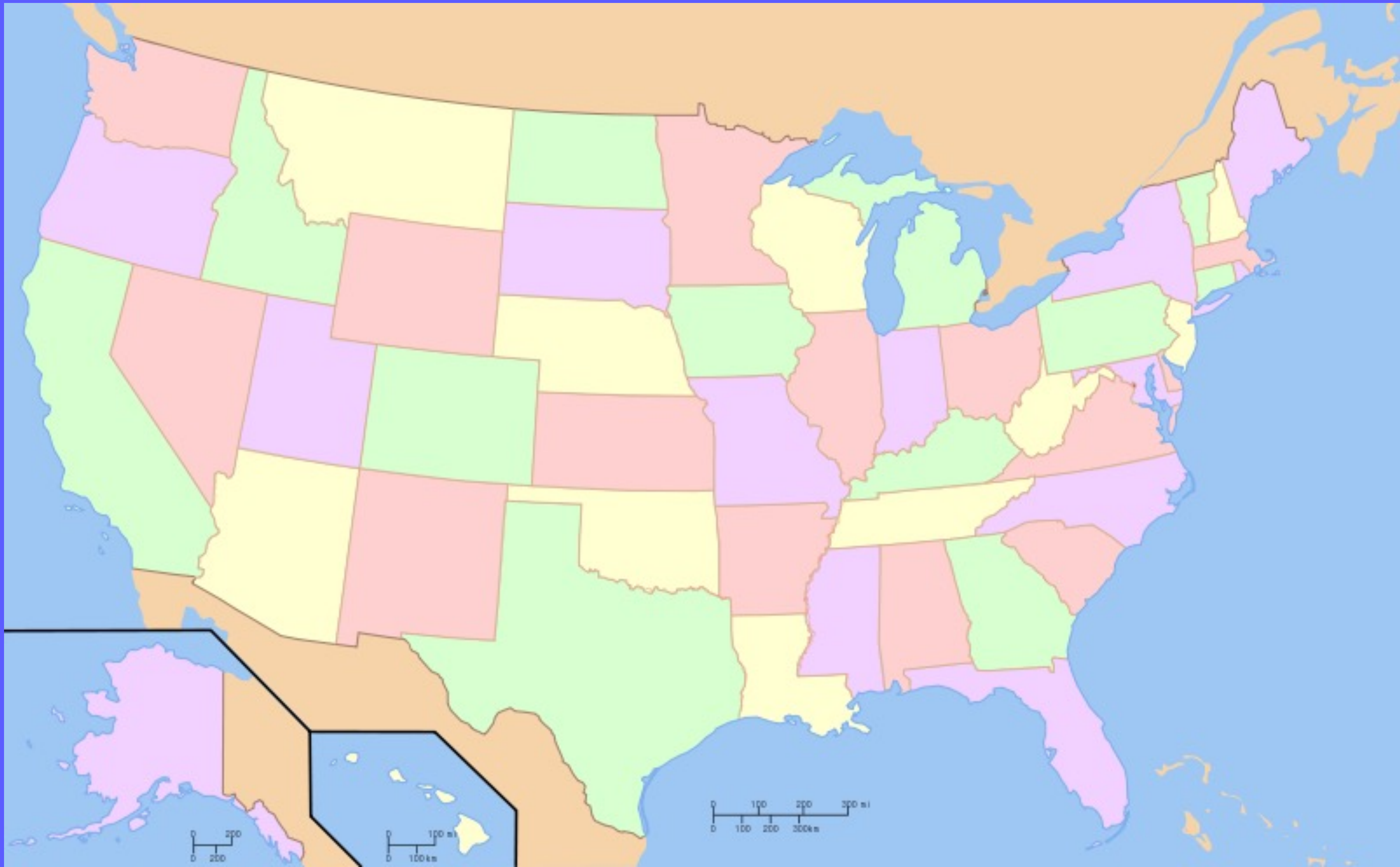


Graph Coloring

Two-Coloring Graphs

- Vertex coloring
  - assign a color to each vertex of a graph G such that no edge links two vertices of the same color.
  - the goal is to use as few colors as possible.
- Bipartite graph coloring
  - A graph which can be colored without conflicts while using only two colors.
- Two coloring problem
  - DFS (or BFS) the graph so that a new vertex is visited, we color it the opposite of its parent.
  - For each non-discovery edge, check whether it links two vertices of the same color.
  - Conflict: the graph cannot be two-colored.


WA
OR
CA
NV
ID
MT
WY
UT
AZ
CO
NM
ND
SD
NE
KS
OK
TX
MN
IA
MO
AR
LA
WI
IL
MS
MI
IN
KY
TN
AL
OH
WV
VA
GA
FL
SC
NC
PA
NY
VT
NH
ME
MA
RI
CT
NJ
DE
MD
AK
HI


0 100 200 300 mi
0 100 200 300km