확장트리 / 최단경로

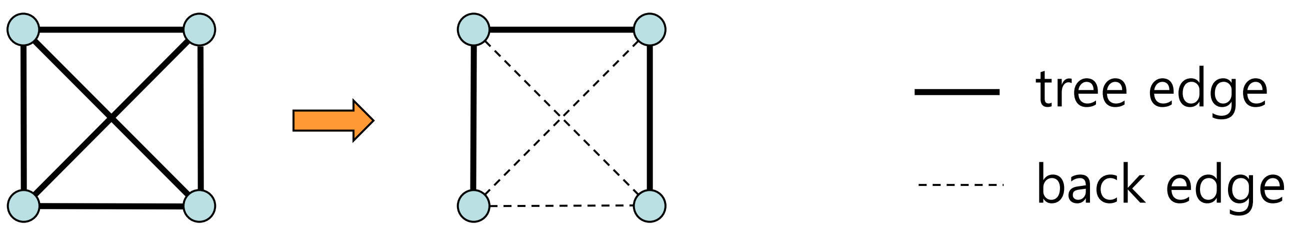
강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터공학과

SPANNING TREES

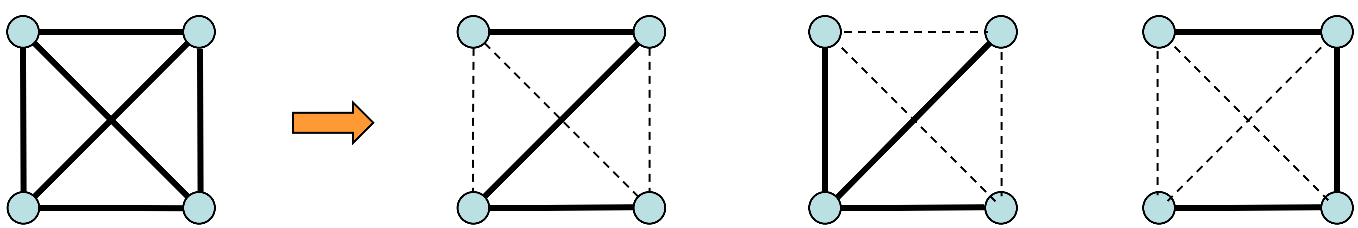
Spanning Trees
Depth First Spanning Trees
Bread First Spanning Trees
Minimum Cost Spanning Trees

Spanning Trees

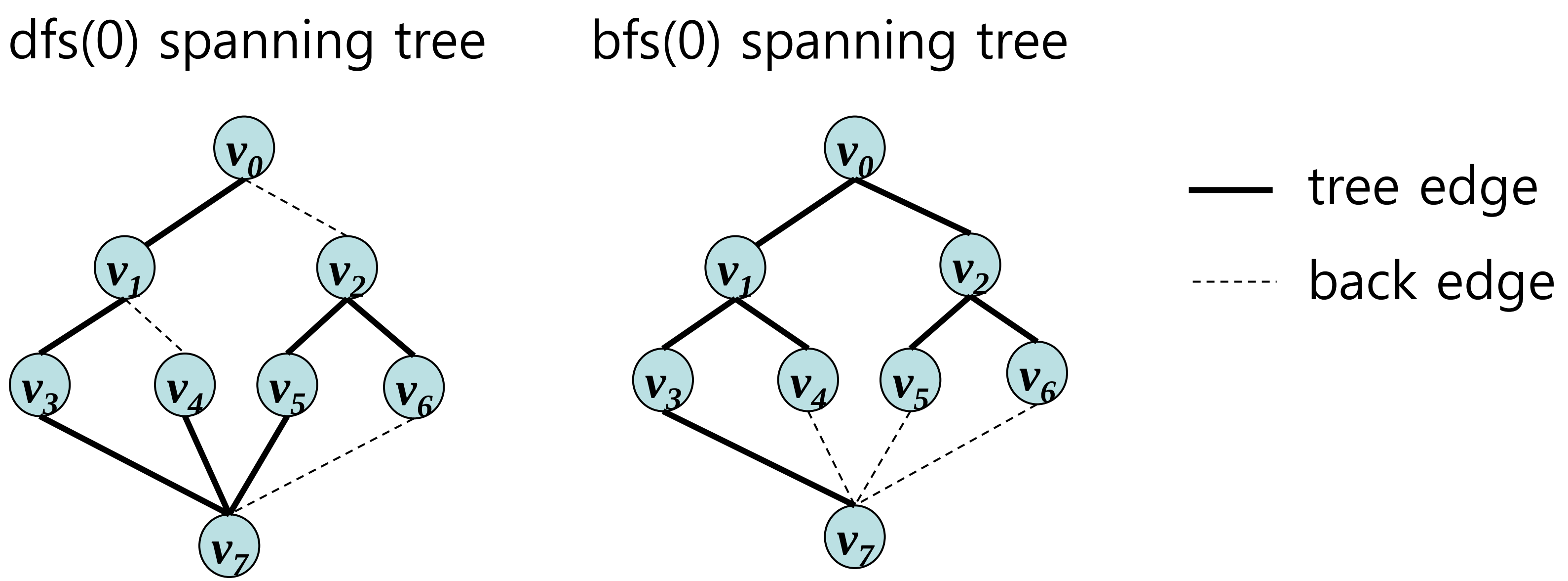
- $G = (V, E)$: a connected graph.
- A spanning tree of G is a tree $G' = (V', E')$ such that $V' = V$ and $E' \subseteq E$.
- Let $T = E'$: Tree edges
  $B = E - E'$: Back edges (Non-tree edges)



- Spanning trees are not unique.



Depth First spanning trees and Breadth First spanning trees



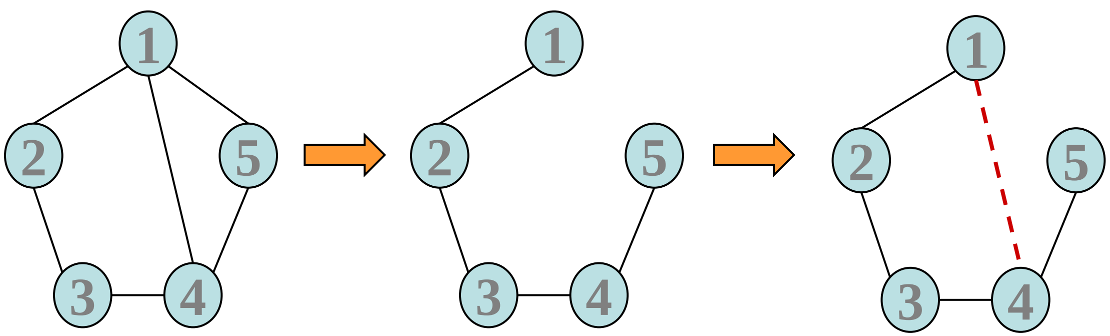
How to determine Tree edges?

- If (!visited[w]) {

  $T = T \cup \{ (v, w) \}$; /* Initially $T = \varnothing$ */

  }

- If any back edge is introduced in the spanning tree, then a cycle is formed.



- Path 1-2-3-4
- Back edge (4,1)
- A cycle 1-2-3-4-1

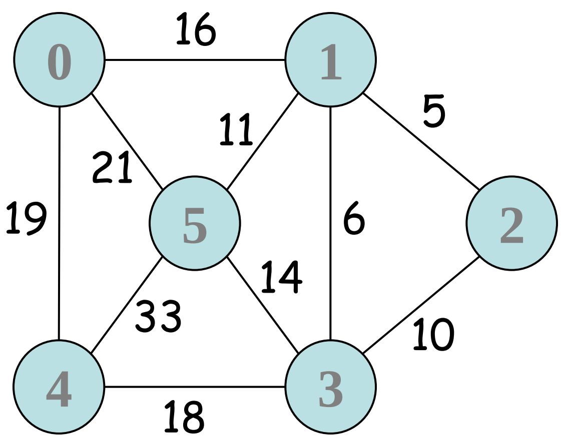
- Any connected graph with N vertices must have at least N-1 edges.
- Any connected graph with N vertices and N-1 edges is a (free) tree.

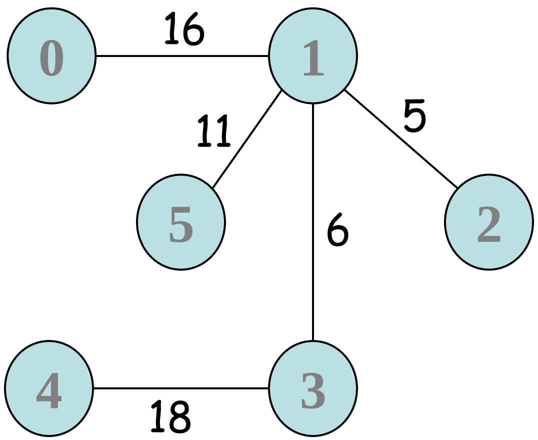
Minimum Cost Spanning Trees

Minimum Cost Spanning Trees ?

- The cost of a spanning tree
  = the sum of the costs of the edges in that tree.
- A minimum cost spanning tree of a weighted graph G is a spanning tree which has the minimum cost among all the spanning trees of G.
- The problem is:
  "How to find a minimum cost spanning tree?"
- Solutions:
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Sollin's Algorithm

Example





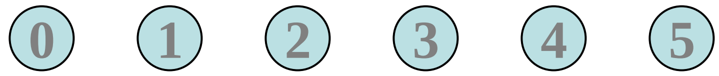
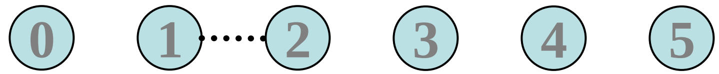
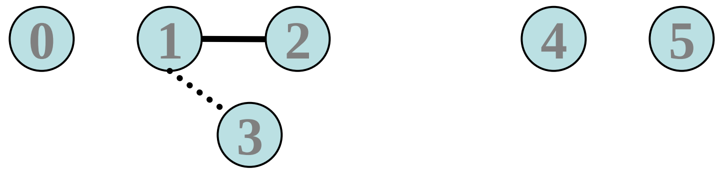
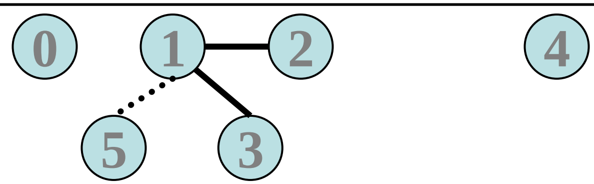
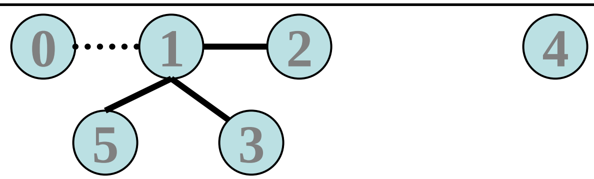
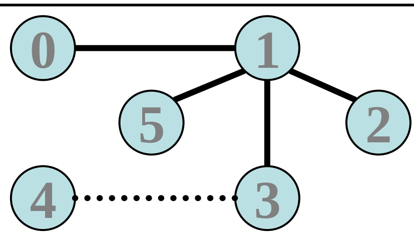
Kruskal's Algorithm

Notations

- T : the current set of tree edges.
- |T| : number of edges in T.
- n : the number of vertices of the given graph
- E : the set of edges of the given graph

Algorithm

```
T = ∅ ; /* T is the set of tree edges. */
while ( (|T| < n-1) && (E is not empty) )  {
      choose an edge (v,w) from E of the lowest cost ;
      delete (v,w) from E ;
      if  ( (v, w) does not create a cycle in T )
            add (v,w) to T ;
      else
            discard (v,w) ;
}
 if (|T| < n-1)
      System.out.println("No spanning tree!") ;
```

Edge	Cost	Action	Tree Edges	Pairwise disjoint sets
-	-	-		{0} {1} {2} {3} {4} {5}
(1,2)	5	Add		{0} {1,2} {3} {4} {5}
(1,3)	6	Add		{0} {1,2,3} {4} {5}
(2,3)	10	Discard		{0} {1,2,3} {4} {5}
(1,5)	11	Add		{0} {1,2,3,5} {4}
(3,5)	14	Discard		{0} {1,2,3,5} {4}
(0,1)	16	Add		{0,1,2,3,5} {4}
(3,4)	18	Add		{0,1,2,3,4,5}

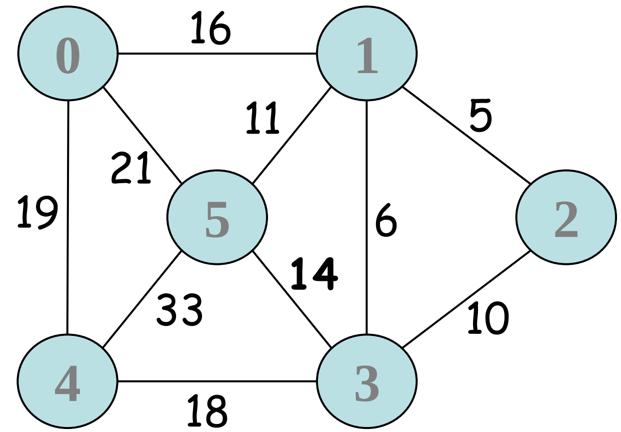
Time Complexity of Kruskal's Algorithm

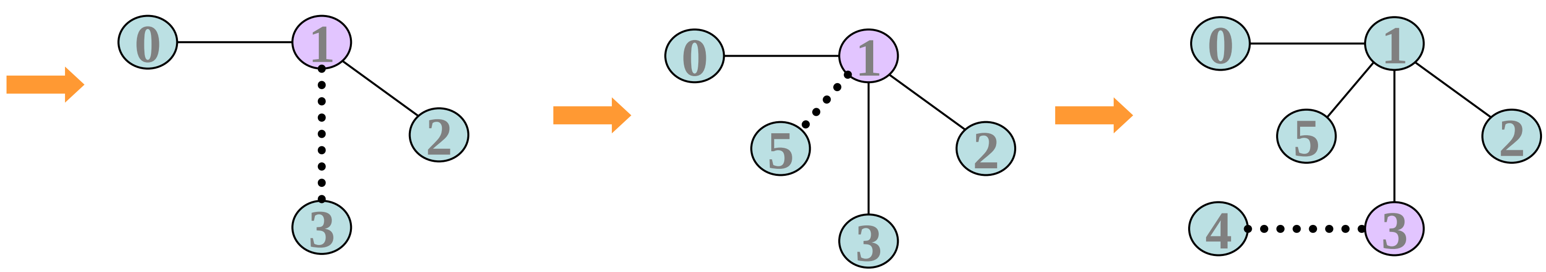
- For line 3 and 4,
  - Use a sorted list for E : $O(e \log e)$
  - Use Heap: Construction of Initial Heap : $O(e)$
    Each Action of deleteMin() : $O(\log e)$
    $\Rightarrow O(e) + e \bullet O(\log e) = O(e \log e)$
- For line 5 and line 6,
  - Use the Union-and-Find algorithm (See Chapter 5).

    $\Rightarrow O(e\, \alpha(e))$

    - Note that $O(\alpha(e)) < O(\log e)$ and
      the function $\alpha(e)$ is a very very slowly increasing function.
      In other words, $\alpha(e)$ is practically an almost constant function.
- The computing time is determined by line 3 and 4.
  - Therefore, the complexity is $O(e \log e)$.

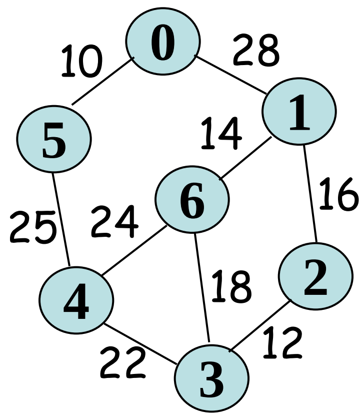
Prim's Algorithm



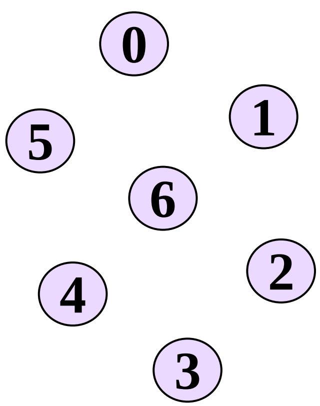


- Time Complexity: O(n^2)

Sollin's Algorithm: Example 1



Initial spanning forest

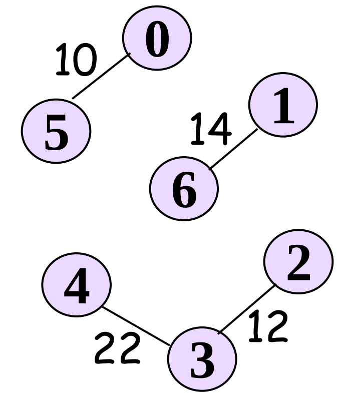


Tree(0) → (0,5)
Tree(1) → (1,6)
Tree(2) → (2,3)
Tree(3) → (3,2)
Tree(4) → (4,3)
Tree(5) → (5,0)
Tree(6) → (6,1)

Remove duplicated edges.

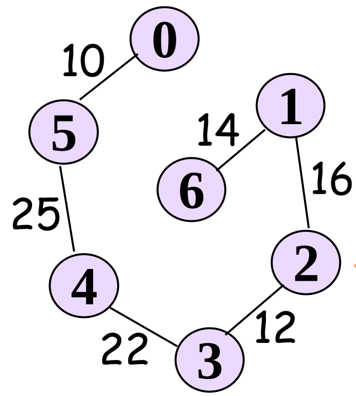
Tree(0) → (0,5)
Tree(1) → (1,6)
Tree(2) → (2,3)
Tree(4) → (4,3)

Add the edges to the spanning forest.

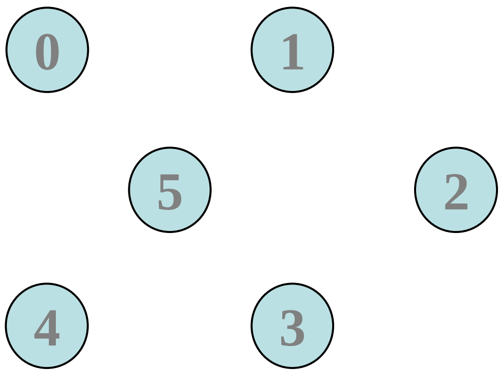


Tree(0) → (5,4)
Tree(1) → (1,2)
Tree(2) → (2,1)

Tree(0) → (5,4)
Tree(1) → (1,2)



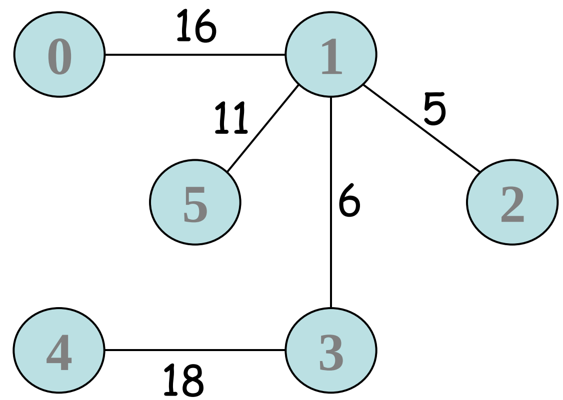
Sollin's Algorithm: Example 2



Tree(0) → (0,1)
Tree(1) → (1,2)
Tree(2) → (2,1)
Tree(3) → (3,1)
Tree(4) → (4,3)
Tree(5) → (5,1)

Remove duplicated edges.
→ No duplicated edges.

Add the edges to the spanning forest.

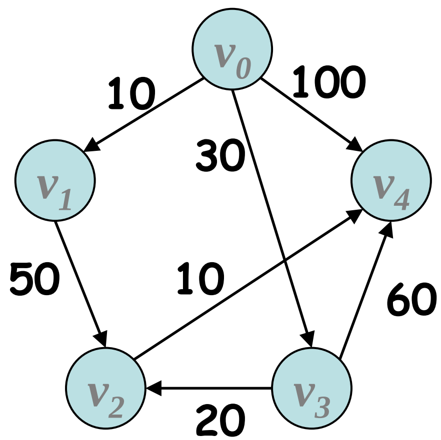


Shortest Paths

"Single Source All Destinations"
"All Pairs"

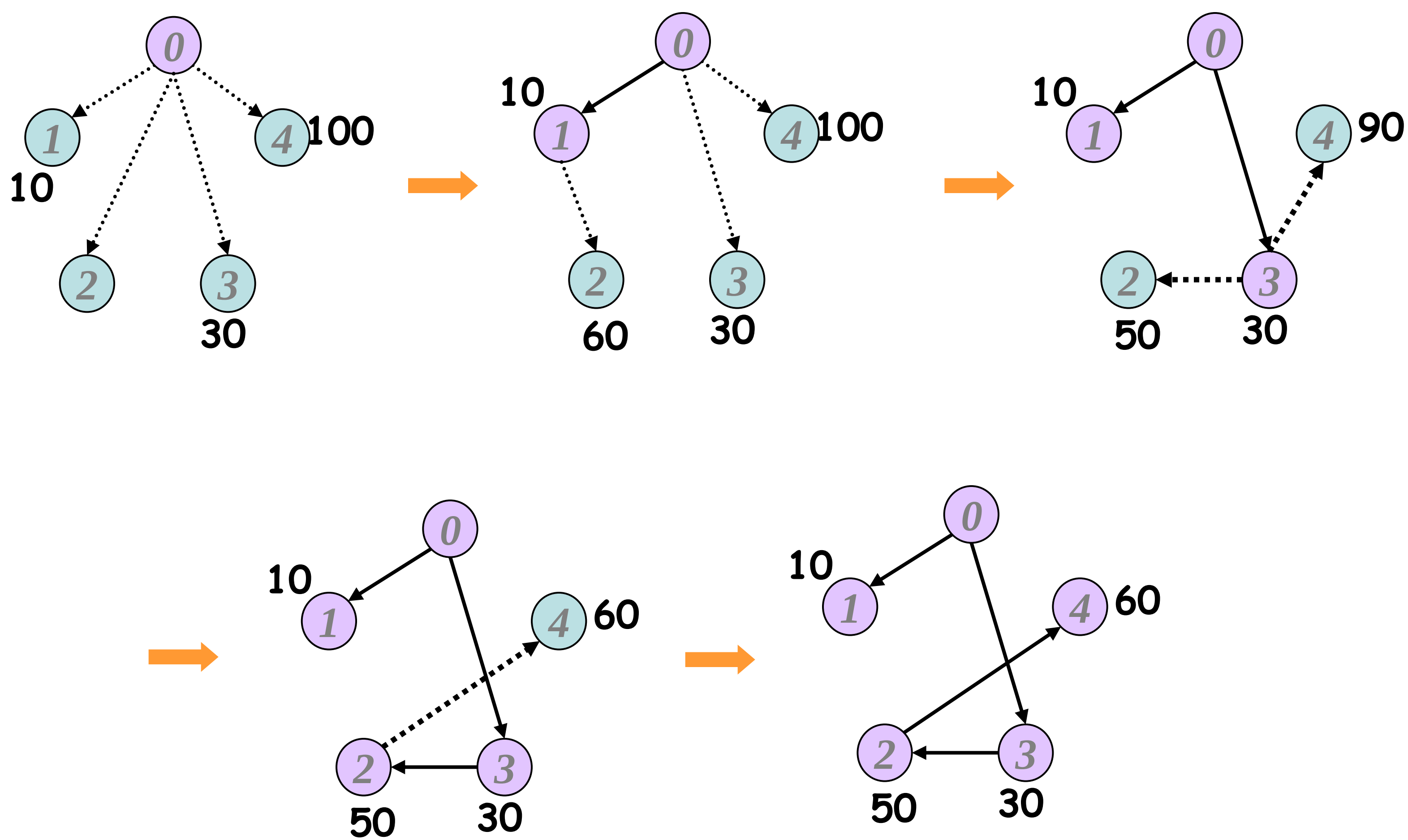
Single Source All Destinations

Dijkstra's Algorithm



Path	Length
$v_0 - v_1$	10
$v_0 - v_3$	30
$v_0 - v_3 - v_2$	50
$v_0 - v_3 - v_2 - v_4$	60

Iteration	S	u	distance[1]	distance[2]	distance[3]	distance[4]
초기화	{0}	-	**10**	∞	30	100
1	{0,1}	1	10	60	**30**	100
2	{0,1,3}	3	10	**50**	30	90
3	{0,1,3,2}	2	10	50	30	**60**
-	{0,1,3,2,4}	4	10	50	30	60


0
1
10
4
100
2
3
30
0
10
1
4
100
2
60
3
30
0
10
1
4
90
2
50
3
30
0
10
1
4
60
2
50
3
30
0
10
1
4
60
2
50
3
30

Proof Sketch of Dijkstra's Algorithm

- We attempt to devise an algorithm that generates the shortest paths in non-decreasing order of length.
- Notations

  v_0 : source vertex

  S : the set of vertices, including v_0, whose shortest paths have been found.

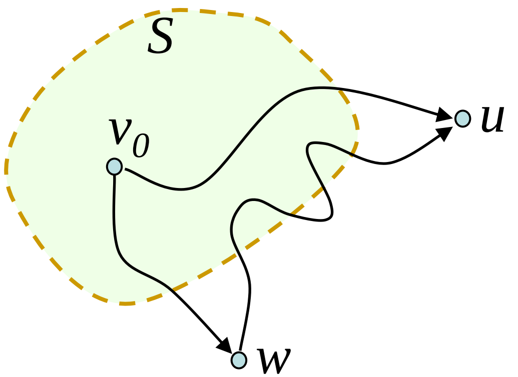
  $\text{distance}[w]$: the length of the shortest path starting from v_0, going through vertices only in S, and ending in w.

Observations

① If the next shortest path is to u, then the path begins at v_0, ends at u, and goes through only those vertices in S.

(Proof) Assume that there is a vertex on this path that is not in S and that w is the first such vertex not in S among the path.
Then $v_0 \rightarrow w$ is shorter than $v_0 \rightarrow u$.
The algorithm should generate the paths in non-decreasing order.
So, w should be in S before u.

Contradiction!
No such w exist.



② Vertex u is chosen so that it has the minimum distance, distance[u], among all the vertices not in S.
(This follows from the definition of distance[] and observation ①.)

- Several vertices with the same distance[] ?
  ⇒ any of them may be selected.

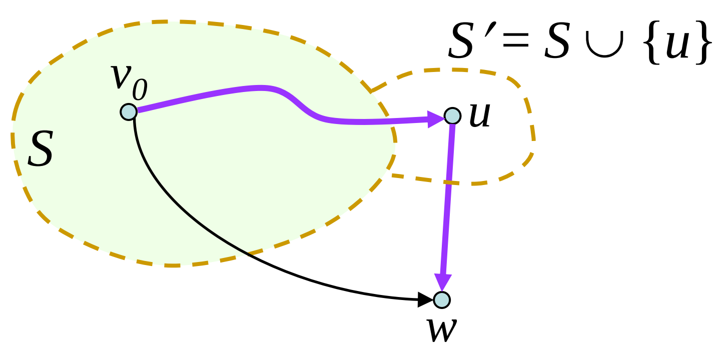
③ Adding the new vertex u into S.

- Once we have selected and generated the shortest path from v_0 to u, u becomes a member of S.
- If distance[w] changes, then it must be due to a shorter path $v_0 \rightarrow u \rightarrow w$, where all intermediate vertices must be in S.
- The subpath $u \rightarrow w$ can be chosen so as to have no intermediate vertices.

if ((old distance[w]) > distance [u] + cost[u,w]) {

(new distance[w]) = distance[u] + cost[u,w] ;

}



Dijkstra's Algorithm

```
public void  shortestPaths (int sourceVertex) {
    int i, u, w ;
    boolean[] found = new boolean[this._numOfVertices] ;
    for (i = 0; i < this._numOfVertices ; i++) {
        found[i] = false ;
        this._distance[i] = this._cost[sourceVertex][i] ;
     }
     found[sourceVertex] = true ;
     this._distance[sourceVertex] = 0 ;
     for (i=0; i < this._numOfVertices-2; i++) {
        u = choose(found);
        this._found[u] = true;
        for (w = 0; w < this._numOfVertices ; w++) {
            if ( !found[w] ) {
                if ( this._distance[w] > this._distance[u] + this._cost[u][w] )
                    this._distance[w] = this._distance[u] + this._cost[u][w];
            }
        }
    }
}
```

Time Complexity : $O(n^2)$

Generation of Vertex Sequences

- Use another array path[] of vertices.

  path[u] ≡ the vertex immediately before u in the shortest path

- Initialize path[u] = v for all u ≠ v (v is the source) ;

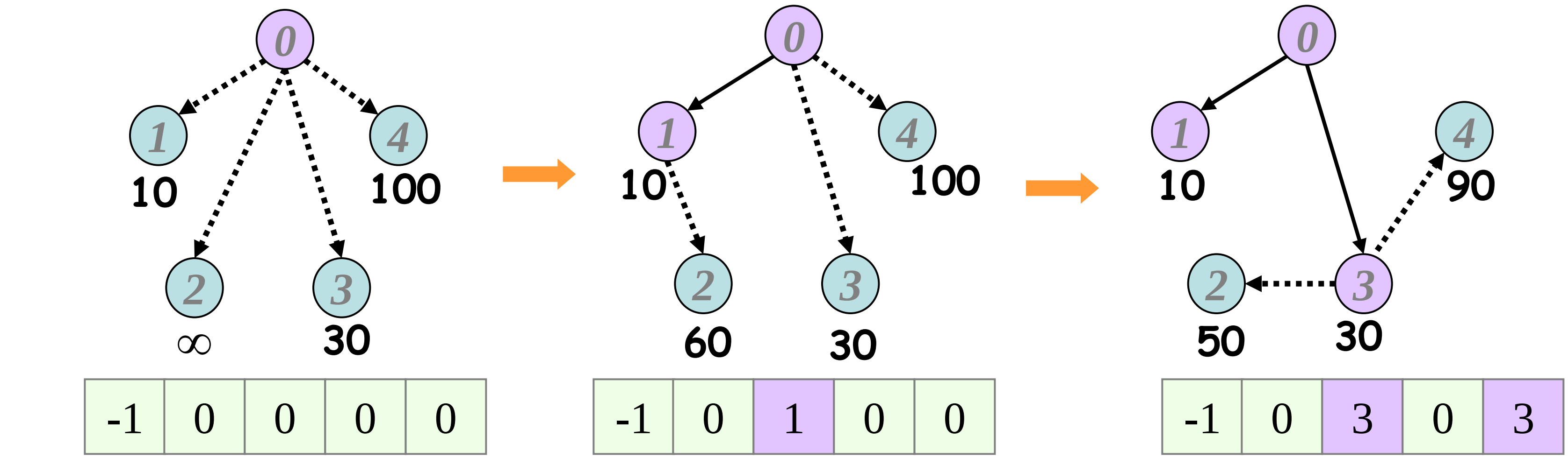
  path[v] = -1 ;

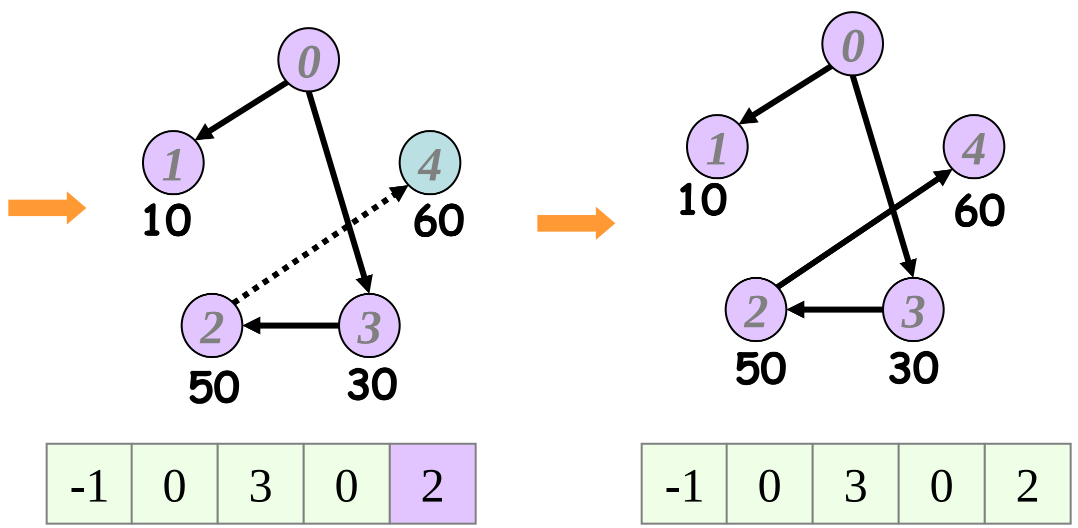
- And then,

```
If ( this._distance[w] > this._distance[u]+ this._cost[u][w] ) {
      this._distance[u] = this._distance[u]+ this._cost[u][w] ;
      this._path[w] = u ;
}
```

- Upon termination,
  the paths can be found by tracking backward.

Finding Vertex Sequences of Paths:





The shortest paths in reverse vertex order

- 1-0
- 2-3-0
- 3-0
- 4-2-3-0

All Pairs Shortest Paths

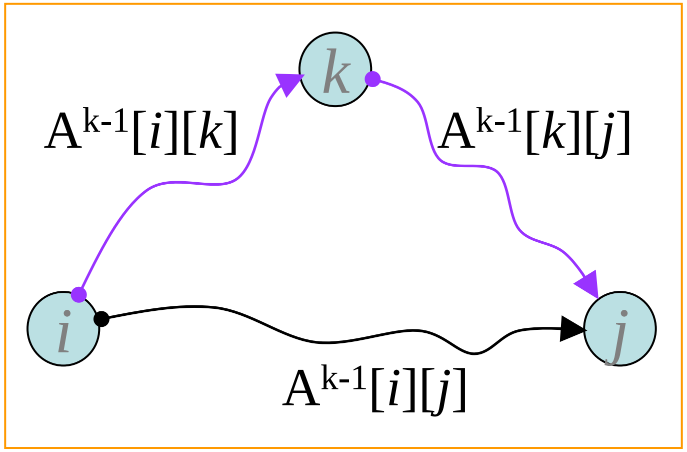
- Apply the Dijkstra's Algorithm n times : $O(n^3)$
- A simpler algorithm using Cost Adjacency Matrix.

$$\text{cost}[i][i] = 0$$
$$\text{cost}[i][j] = \text{cost of edge } <i, j> \in E$$
$$\text{cost}[i][j] = \infty \text{ if } <i, j> \notin E$$

- Define $A^k[i][j]$ to be the cost of the shortest path from i to j going through no intermediate vertex of index greater than k.
- Then $A^{n-1}[i][j]$ will be the cost of the shortest path from i to j in G.
- $A^{-1}[i][j]$ is just $\text{cost}[i][j]$.

Basic Idea

- Successively generate the matrices $A^0, A^1, \ldots, A^{n-1}$ starting from A^{-1}.
- Assume we have already generated A^{k-1}.
  Then we may generate A^k as follows:
  For any pair of vertices i and j,
  either
  (*a*) the shortest path from i to j
  does not go through k,
  $\Rightarrow$ its cost is $A^{k-1}[i][j]$
  or
  (*b*) the path goes through k.
  $\Rightarrow$ its cost is $A^{k-1}[i][k] + A^{k-1}[k][j]$



- Thus,
  $A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k]+A^{k-1}[k][j]\}$, $k \geq 0$, and
  $A^{-1}[i][j] = \text{cost}[i][j]$

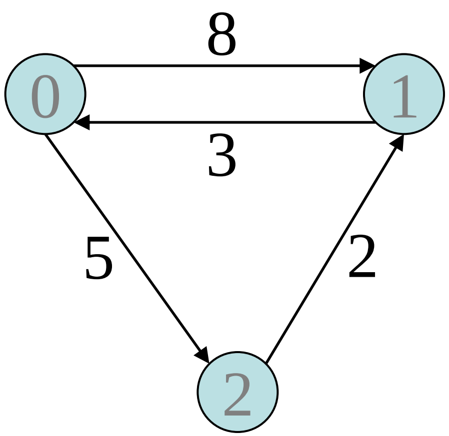
Floyd's Algorithm

```
private void  allcosts ( int[][] cost, int[][] distance,  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][j] > distance[i][k] + distance[k][j])
                        distance[i][j] = distance[i][k] + distance[k][j];
}
```

Time Complexity : $O(n^3)$

Example of Floyd's Algorithm

$$\text{cost [] } = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$$



A^{-1}	0	1	2
0	0	8	5
1	3	0	∞
2	∞	2	0

Example of Floyd's Algorithm (Cont'd)

A^0	*0*	*1*	*2*
0	0	8	5
1	3	0	8
2	∞	2	0

$A^0[1][2] = \min\{\infty, 3+5\} = 8$

$A^0[2][1] = \min\{2, \infty+8\} = 2$

A^1	*0*	*1*	*2*
0	0	8	5
1	3	0	8
2	5	2	0

$A^1[0][2] = \min\{5, 8+8\} = 5$

$A^1[2][0] = \min\{\infty, 2+3\} = 5$

A^2	*0*	*1*	*2*
0	0	7	5
1	3	0	8
2	5	2	0

$A^2[0][1] = \min\{8, 5+2\} = 7$

$A^2[1][0] = \min\{3, 8+5\} = 3$

Recovering the Paths

- path[i][j] means that the shortest path from *i* to *j* goes through path[i][j].
- Initially,

```
path[i][j] = -1;
```

- In the innermost loop,

```
if (distance[i][j] > distance[i][k] + distance[k][j]) {
    distance[i][j] = distance[i][k] + distance[k][j] ;
    path[i][j] = k;
}
```

- In order to print out the shortest path from *i* to *j* :

```
private void showPath (int i, int j)
{
    int  k ;
    k = path[i][j] ;
    if (k >= 0) {
        showPath (i, k);
        System.out.print (k);
        showPath (k, j);
    }
}
```

Example: Recovering the paths

$path^{-1}$	0	1	2
0	-1	-1	-1
1	-1	-1	-1
2	-1	-1	-1

$path^{0}$	0	1	2
0	-1	-1	-1
1	-1	-1	**0**
2	-1	-1	-1

$path^{1}$	0	1	2
0	-1	-1	-1
1	-1	-1	0
2	**1**	-1	-1

$path^{2}$	0	1	2
0	-1	**2**	-1
1	-1	-1	**0**
2	**1**	-1	-1

$A^0[1][2] = \min\{\infty, \underline{3+5}\} = 8$
$path^0[1][2] = \mathbf{0}$

$A^0[2][1] = \min\{\underline{2}, \infty+8\} = 2$
$path^0[2][1]$: No change

$A^1[0][2] = \min\{\underline{5}, 8+8\} = 5$
$path^1[0][2]$: No change

$A^1[2][0] = \min\{\infty, \underline{2+3}\} = 5$
$path^1[2][0] = \mathbf{1}$

$A^2[0][1] = \min\{8, \underline{5+2}\} = 7$
$path^2[0][1] = \mathbf{2}$

$A^2[1][0] = \min\{\underline{3}, 8+5\} = 3$
$path^2[1][0]$: No change

path[0][1] = 2
$\left.\begin{array}{l}\text{path[0][2]} = -1 \\ \text{path[2][1]} = -1\end{array}\right\}$ $0 \rightarrow 2 \rightarrow 1$

path[1][2] = 0
$\left.\begin{array}{l}\text{path[1][0]} = -1 \\ \text{path[0][2]} = -1\end{array}\right\}$ $1 \rightarrow 0 \rightarrow 2$

path[2][0] = 1
$\left.\begin{array}{l}\text{path[2][1]} = -1 \\ \text{path[1][0]} = -1\end{array}\right\}$ $2 \rightarrow 1 \rightarrow 0$

Transitive Closures

Transitive Closures

- The existence problem of a path $i \rightarrow j$.
- Transitive closure : A^+
  - All path lengths are required to be positive.

$$A^+[i][j] = \begin{cases} 1 & \text{if there is a path } i \rightarrow j \text{ of length} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Reflexive transitive closure : A^*
  - Path lengths are to be nonnegative.

$$A^*[i][j] = \begin{cases} 1 & \text{if there is a path } i \rightarrow j \text{ of length} >= 0 \\ 0 & \text{otherwise} \end{cases}$$

- The only difference between A^+ and A^*:
  - the terms on the diagonal.
    - $A^+[i][i] = 1$ iff there is a cycle of length > 1 containing vertex i.
    - $A^*[i][i] = 1$ always.

Use Floyd's Algorithm for A^+ or A^*

Let $\text{Cost}[i][j] = \begin{cases} 1 & \text{if } < i, j > \in E \\ \infty & \text{if } < i, j > \notin E \end{cases}$

Then, the final matrix becomes A^+ by letting

$$A^+[i][j] = \begin{cases} 1 & \text{if } A[i][j] < +\infty \\ 0 & \text{otherwise} \end{cases}$$

A^* can be obtained from A^+ by setting

$$A^*[i][i] = 1 \text{ for all } i = 1, \ldots, n$$

Simple Modification using Boolean Matrix for A+:

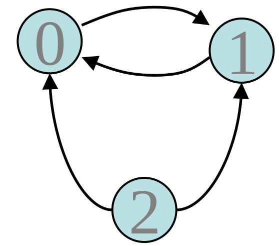
Let

$$\text{Cost}[i][j] = \begin{cases} \text{true} & \text{if } <i, j> \in E \\ \text{false} & \text{if } < i, j > \notin E \end{cases}$$

Then

$$A^k[i][j] = A^{k-1}[i][j] \mathbin{||} (A^{k-1}[i][k] \mathbin{\&\&} A^{k-1}[k][j]);$$

Example for A^+ and A^*



Using Cost Adjacency Matrix

$$\text{Cost} = \begin{pmatrix} \infty & 1 & \infty \\ 1 & \infty & \infty \\ 1 & 1 & \infty \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 1 & \infty \\ 1 & 2 & \infty \\ 2 & 1 & \infty \end{pmatrix} \Rightarrow A^+ = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} \Rightarrow A^* = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Using Boolean Matrix

$$\text{Cost} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \Rightarrow A = A^+ = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} \Rightarrow A^* = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

End of Spanning Trees & Shortest Paths