Data Structures
資料結構

Image courtesy of www.escpeurope alumni.org

Graphs – Part II

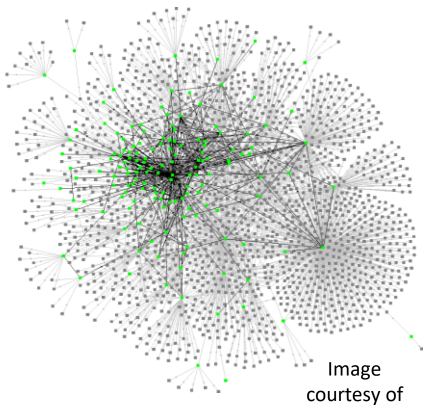
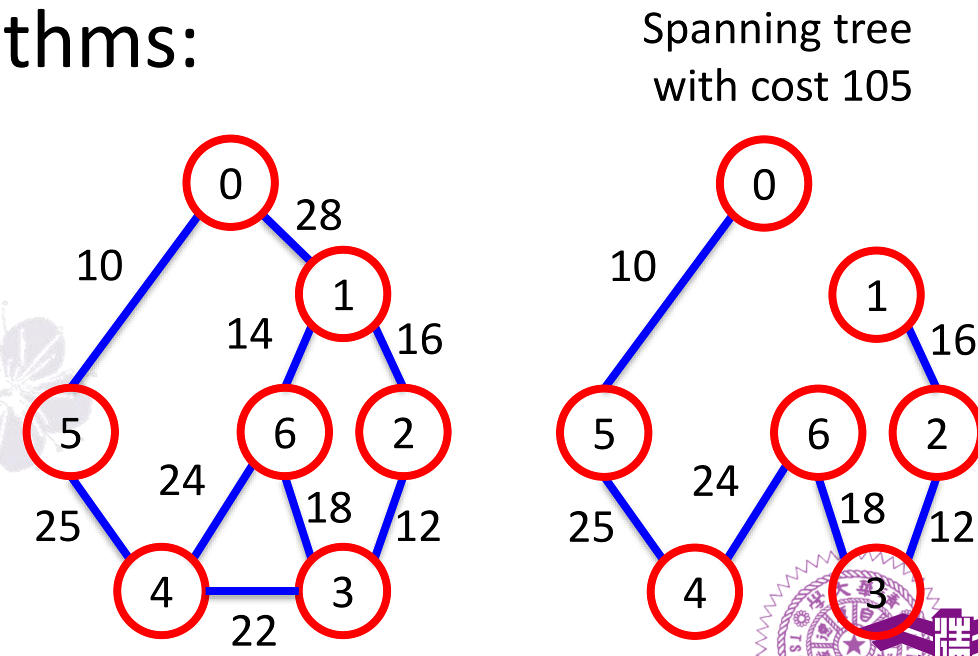


Image courtesy of Ross Mayfield

Department of Computer Science
National Tsing Hua University

Minimum-Cost Spanning Trees

- For a weighted undirected graph, find a spanning tree with **least cost of the sum of the edge weights**.
- Three greedy algorithms:
  - – Kruskal's algorithm
  - – Prims's algorithm
  - – Sollin's Algorithm

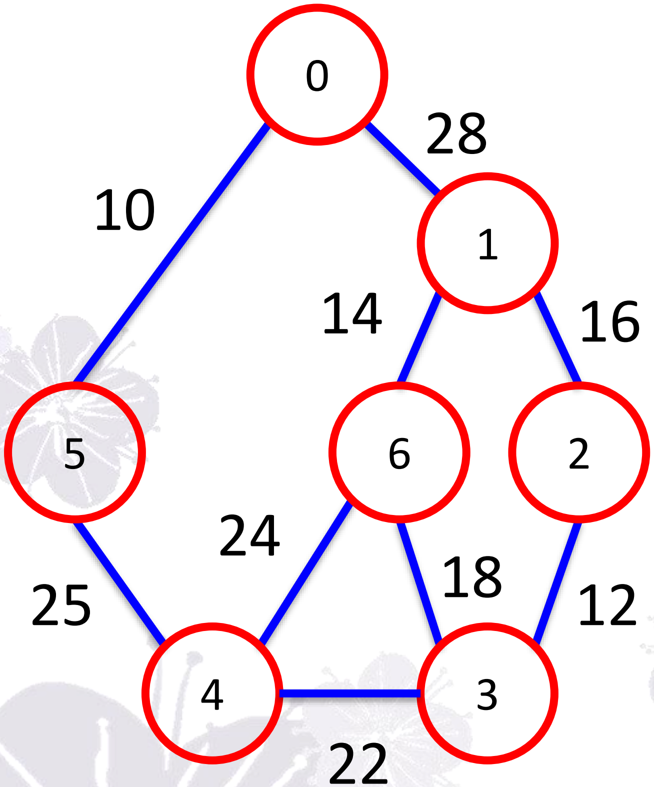


Kruskal's Algorithm

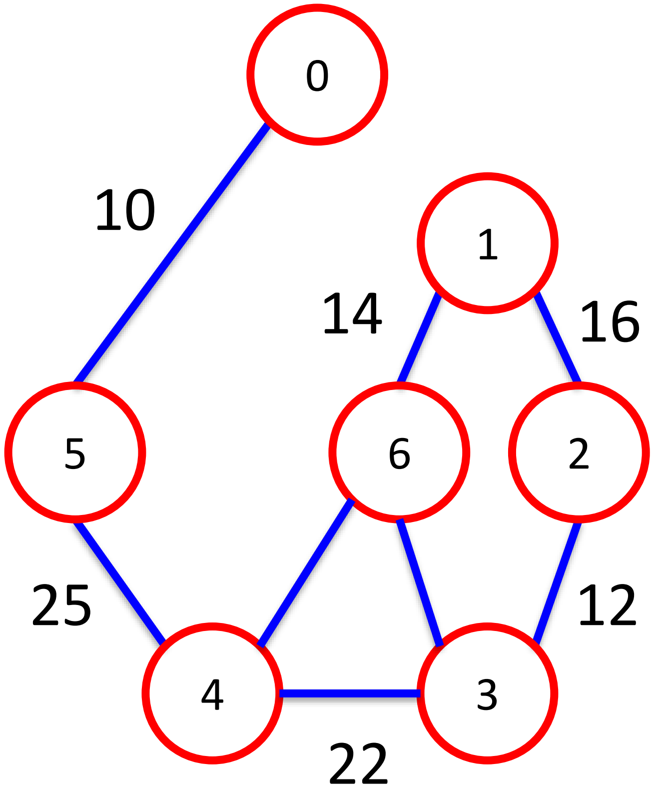
- Idea: Add edges with minimum edge weight to tree one at a time.
- **Step 1:** Find an edge with minimum cost.
- **Step 2:** If it creates a cycle, discard the edge.
- **Step 3:** Repeat step **1** and **2** until we find **n-1** edges.

Running Example

Refer to textbook for detailed steps!



Connected graph



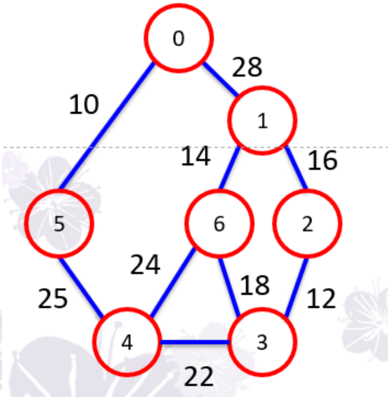
Spanning tree with cost 99

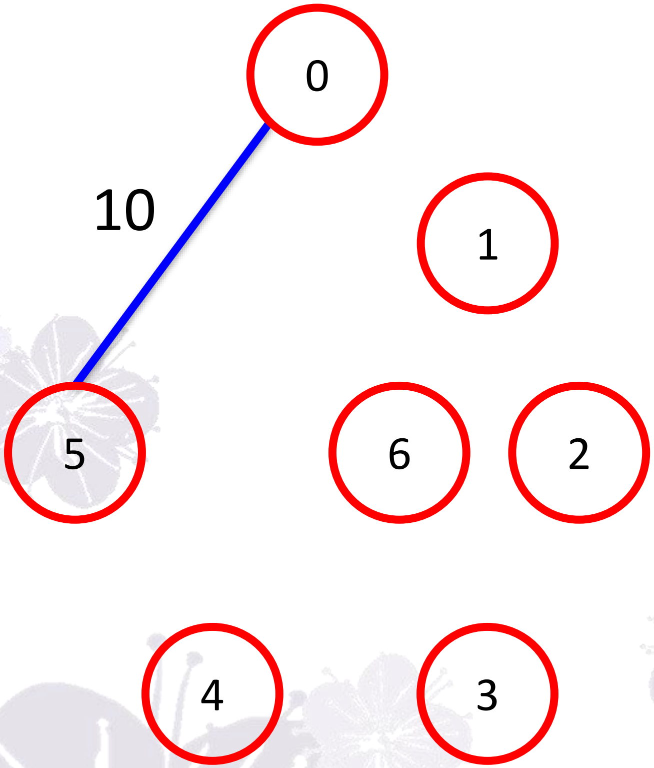
Kruskal's Algorithm

```
Kruskal's algorithm
1.  T = ϕ
2.  While((T contains less than n-1 edges)&&(E is not empty)){
3.    choose an edge (v,w) from E of lowest cost;
4.    delete (v,w) from E
5.    if((v,w) does not create a cycle) add (v,w) to T;
6.    else discard (v,w)
7.  }
8.  If(T contains less than n-1 edges)
9.    cout << "there is no spanning tree!" <<endl;
```

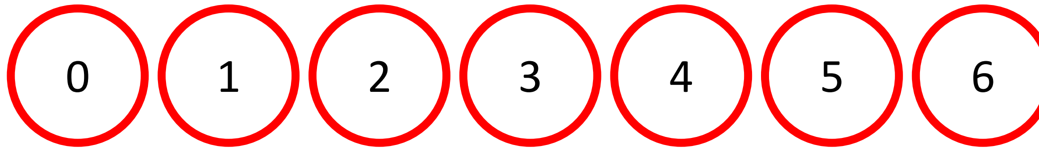
- Step 3 & 4: use **min heap** to store edge cost.
- Step 5: use **set representation** to group all vertices in the same connected component into a set. (see appendix)
  - For an edge (v,w) to be added, if vertices are in the same set, discard the edge, else merge two sets.

Running Example



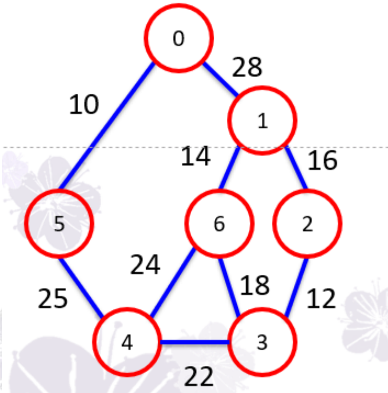


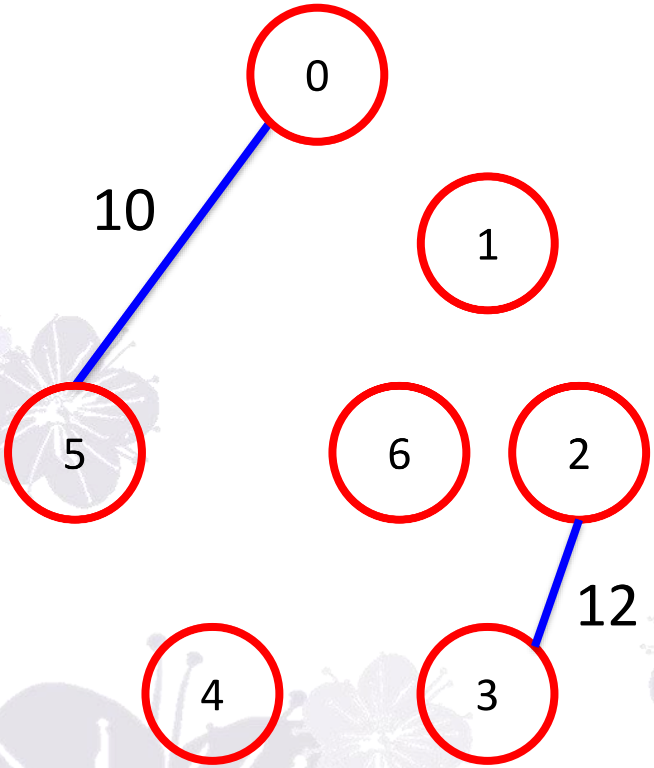
Disjoint set



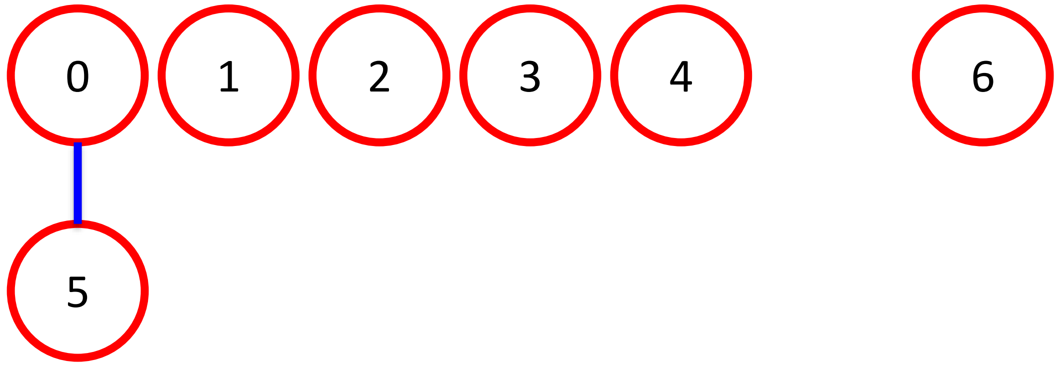
Spanning tree with cost 99

Running Example



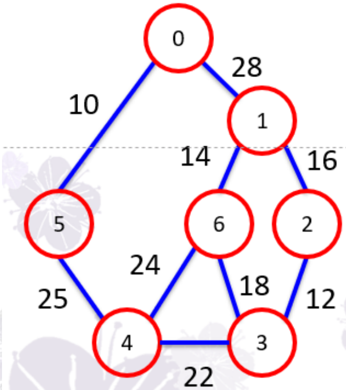


Disjoint set

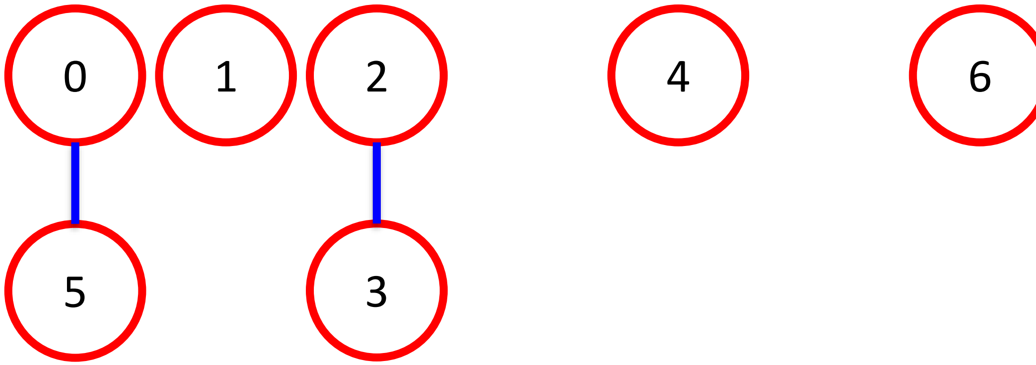


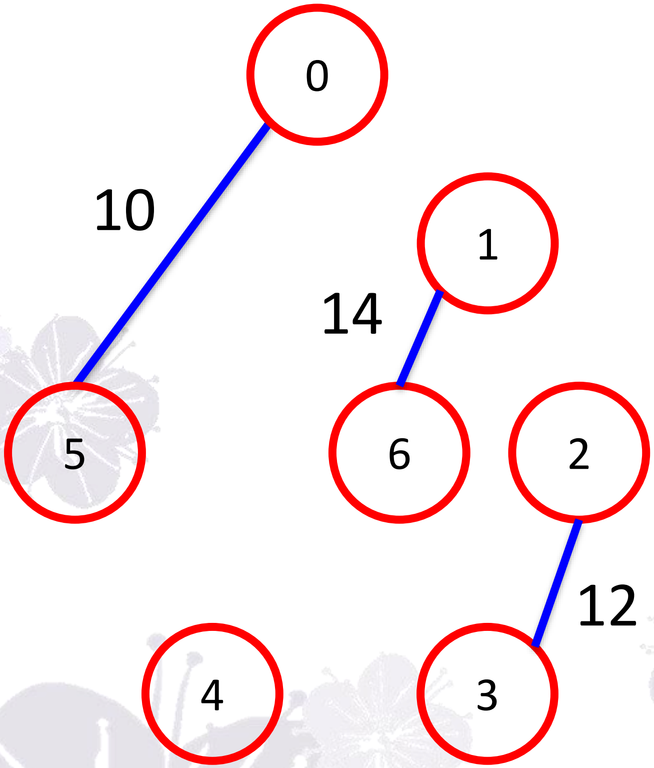
Spanning tree with cost 99

Running Example



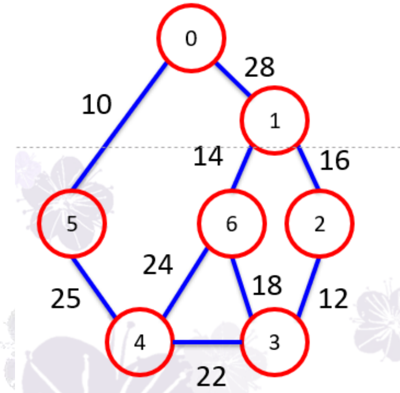
Disjoint set

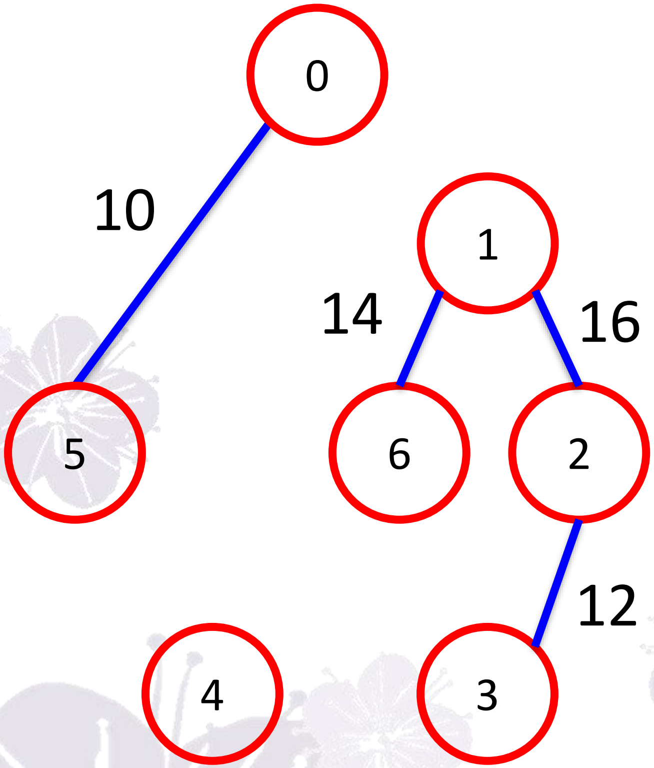




Spanning tree with cost 99

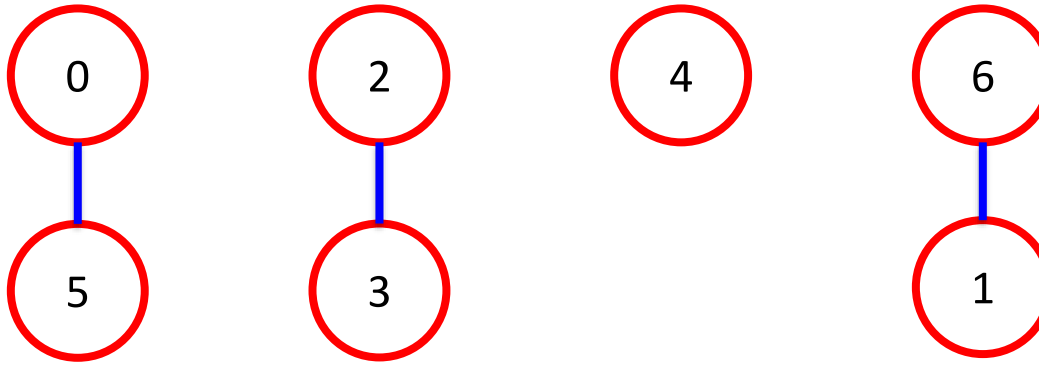
Running Example



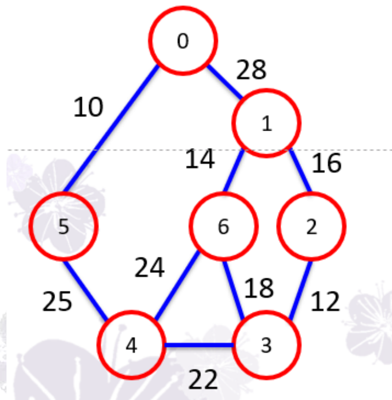


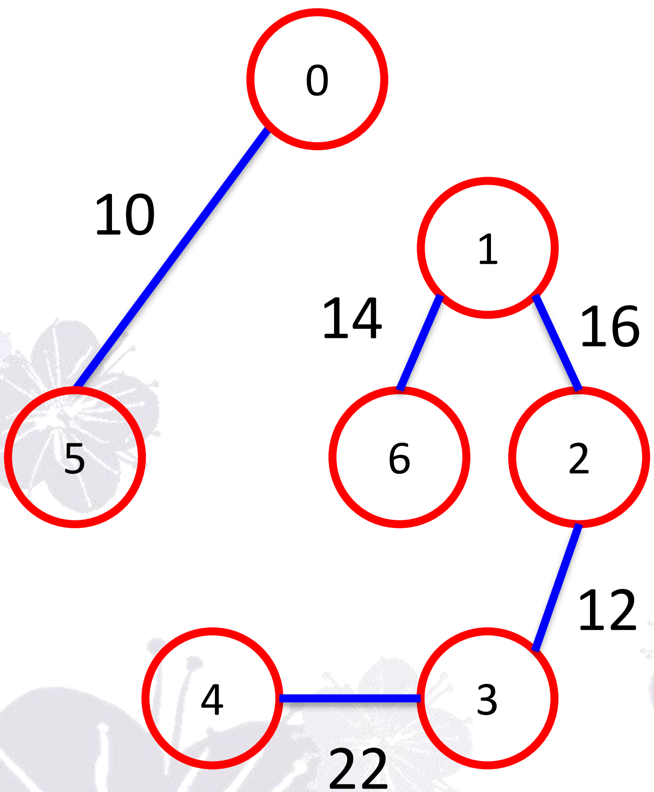
Spanning tree with cost 99

Disjoint set



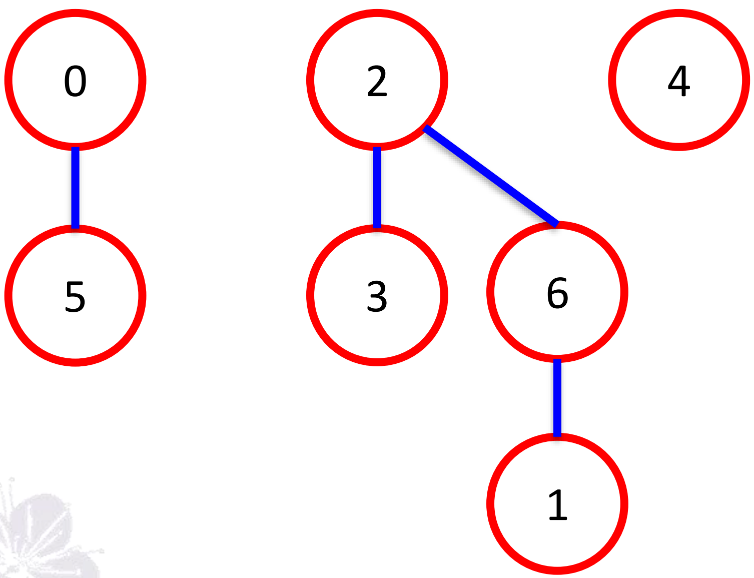
Running Example



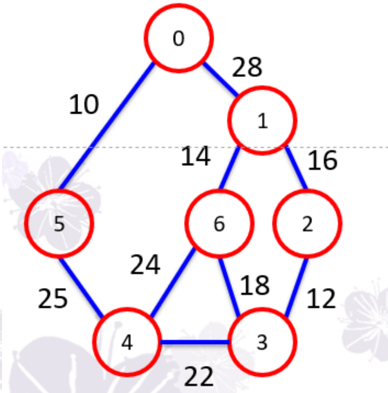


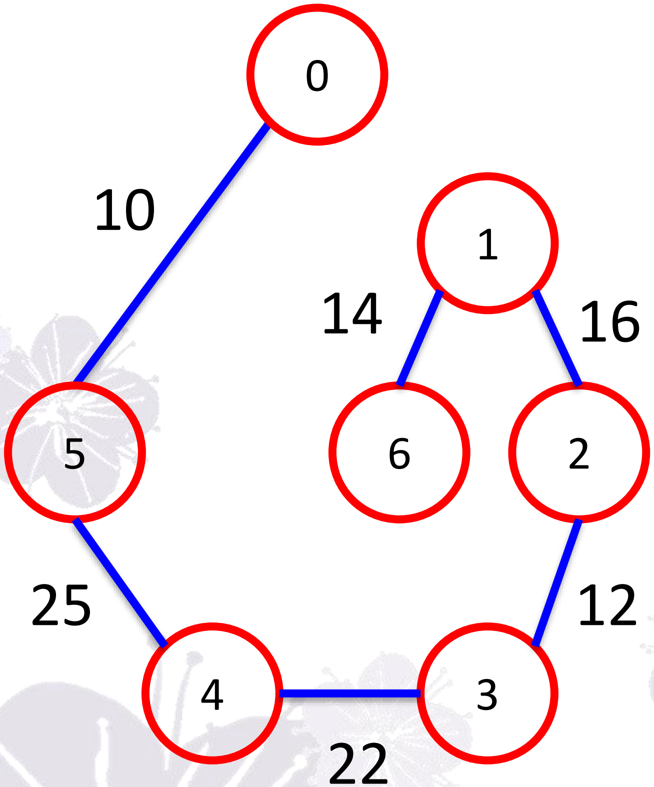
Spanning tree with cost 99

Disjoint set



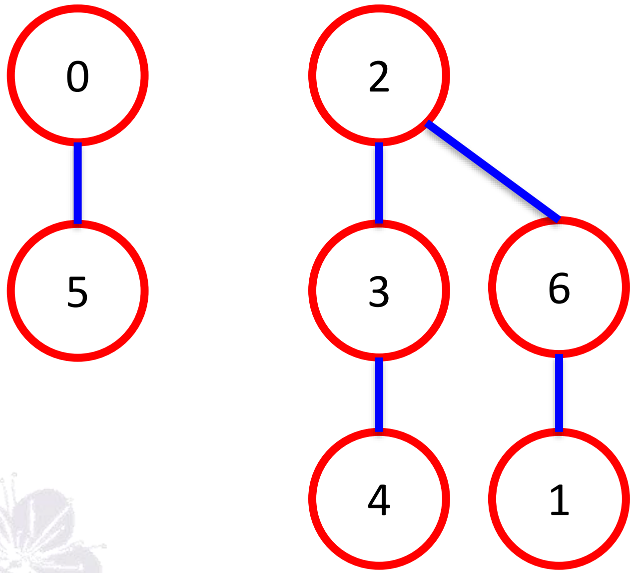
Running Example



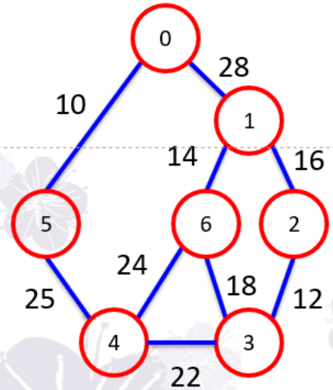


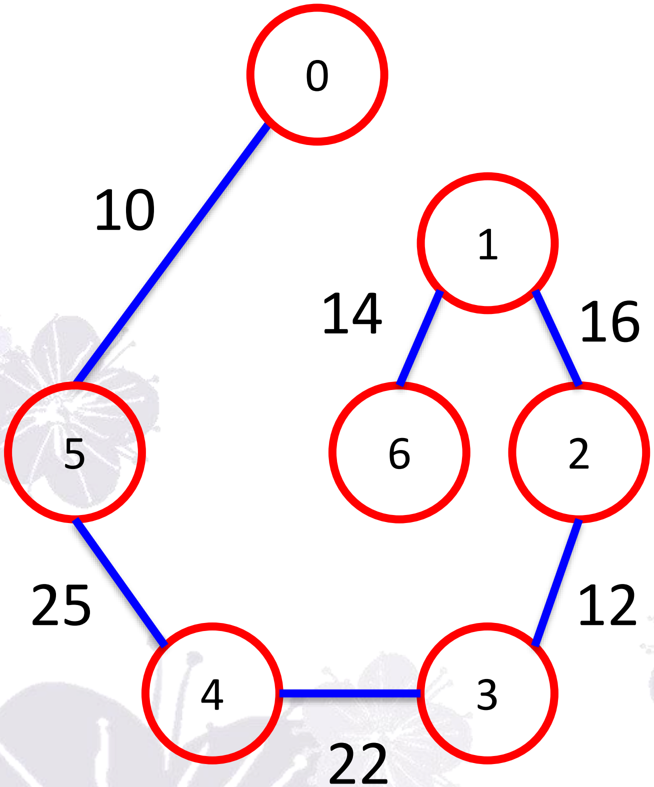
Spanning tree with cost 99

Disjoint set



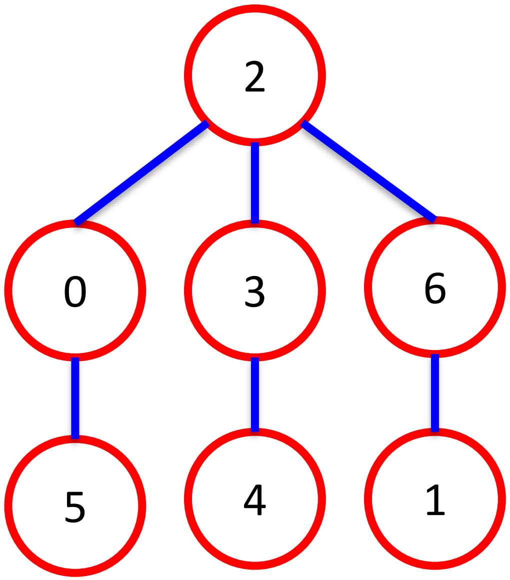
Running Example





Spanning tree with cost 99

Disjoint set



Time Complexity

```
Kruskal's algorithm
1. T = φ
2. While((T contains less than n-1 edges)&&(E is not empty)){
3.   choose an edge (v,w) from E of lowest cost;
4.   delete (v,w) from E
5.   if((v,w) does not create a cycle) add (v,w) to T;
6.   else discard (v,w)
7. }
8. If(T contains less than n-1 edges)
9.   cout << "there is no spanning tree!" <<endl;
```

- Min heap:
  - Step 3&4 : O(log e)
- Set:
  - Step 5: O(log e) -> see appendix
- At most execute e-1 rounds:
  - (e-1)·(log e + log e) = O(e log e)

Kruskal’s Algorithm

《Theorem 6.1》
Let G be any undirected connected graph.
Kruskal’s algorithm generates a minimum-cost spanning tree.

- Proof:
  - (a) Kruskal’s method results in a spanning tree whenever a spanning tree exists
  - (b) The generated spanning tree is of least cost

Step 1: Find an edge with minimum cost.
Step 2: If it creates a cycle, discard the edge.
Step 3: Repeat step **1** and **2** until we find **n-1** edges.

Kruskal's Algorithm

- Proof (a): it finds a spanning tree whenever a spanning tree exists
  - Only delete those **edges that form a cycle**.
  - Delete a cycle doesn't affect the connectivity of the graph.
  - Always result in a **connected graph with n-1 edges**, therefore create a spanning tree.

Step 1: Find an edge with minimum cost.
Step 2: If it creates a cycle, discard the edge.
Step 3: Repeat step **1** and **2** until we find **n-1** edges.

Kruskal's Algorithm

- Proof (b): The generated spanning tree is of least cost
  - Let **U** be another minimum-cost spanning tree.
  - If **T** = **U**, then **T** is a minimum-cost spanning tree.
  - If **T** ≠ **U**, let k, k > 0, be the number of edges in **T** not in **U**.
  - We shall see that there exists a way to transform **U** to **T** in k steps such that cost of **U** is not changed.

Step 1: Find an edge with minimum cost.
Step 2: If it creates a cycle, discard the edge.
Step 3: Repeat step **1** and **2** until we find **n-1** edges.

Kruskal's Algorithm

- Transform **U** to **T**:

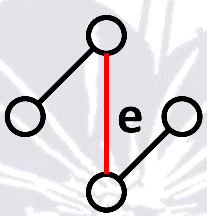
  (1) Let **e** be the least-cost edge in **T** that is <u>not in **U**</u>.

  (2) When **e** is added to **U**, a unique cycle **C** is created.

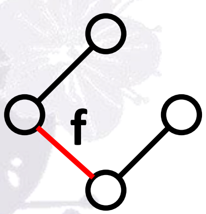
  (3) Let **f** be any edge on **C** that is not in **T**.
  (This edge must exists as **T** contains no cycle).

  – Now **U = U+{e}-{f}** is a spanning tree.

  – We need to proof that **cost(e) = cost(f).**



T

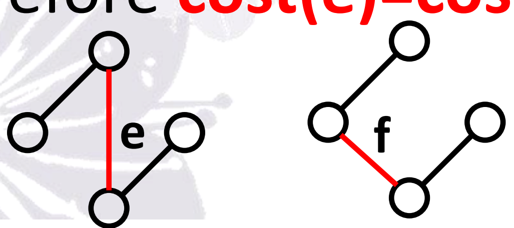


U

Step 1: Find an edge with minimum cost.
Step 2: If it creates a cycle, discard the edge.
Step 3: Repeat step **1** and **2** until we find **n-1** edges.

Kruskal’s Algorithm

- Case i : **cost(e) < cost(f)**
  - cost (U+{e}-{f}) < cost(U) => **Impossible!**
  - Because U is a minimum cost spanning tree.
- Case ii : **cost(e) > cost(f)**
  - **f** should be considered earlier than **e** in Kruskal’s algo.
  - **f** is not in **T** means **f** together with edges in **T** whose costs are less than or equal to **f** form the cycle **C**.
  - Those edges are also in **U (because as mentioned earlier, e is the least-cost-edge which is in T but not in U)**, hence **U** **(**which contains **f)** must also contain a cycle. **Contradiction!**
- Therefore **cost(e)=cost(f)**.



Step 1: Find an edge with minimum cost.

Step 2: If it creates a cycle, discard the edge.

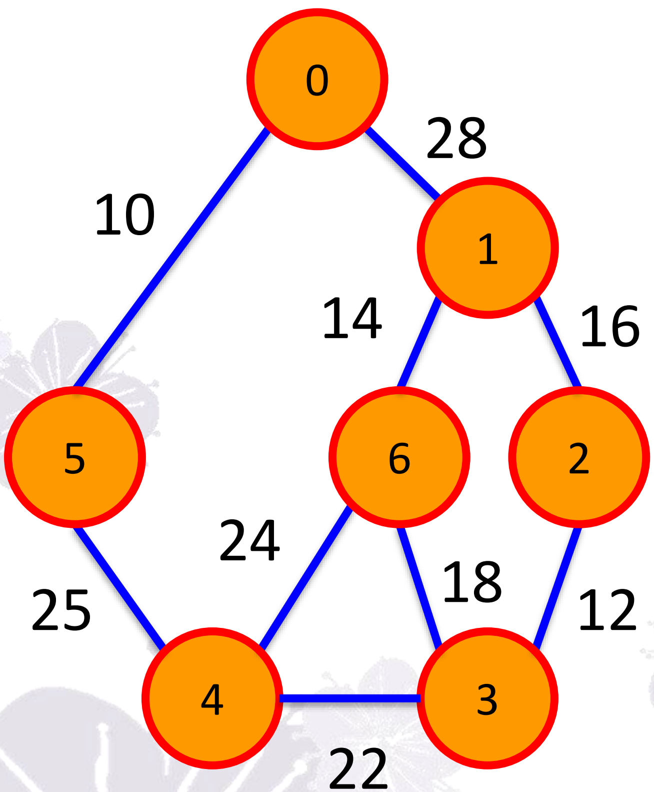
Step 3: Repeat step **1** and **2** until we find **n-1** edges.

Prim’s algorithm

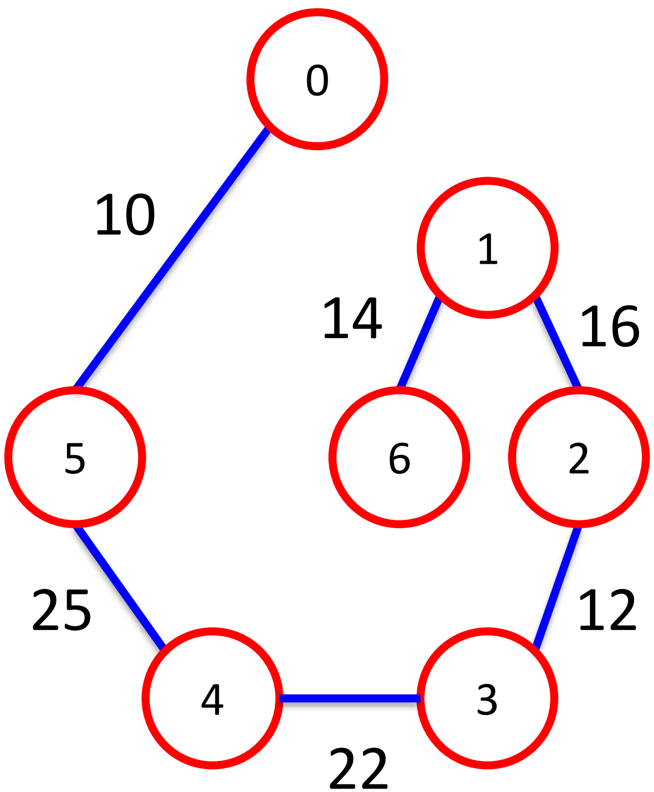
- Idea: Add edges with minimum edge weight to tree one at a time. **At all times during the algorithm, the set of selected edges form a tree.**
- Step 1: Start with a tree T contains a single arbitrary vertex.
- Step 2: Among all edges, add a least cost edge (u,v) to T such that T U (u,v) is still a tree.
- Step 3: Repeat step 2 until T contains n-1 edges.

Running Example

Refer to textbook for detailed steps!



Connected graph



Spanning tree with cost 99

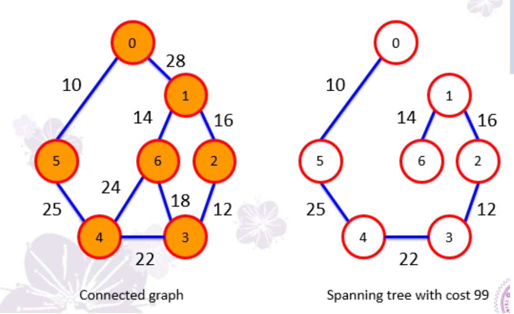
Prim’s Algorithm

```
Prim’s algorithm
1. V(T) = {0} // start with vertex 0
2. for(T=ϕ ; T contains less than n-1 edges; add (u,v) to T){
3.   Let (u,v) be a least cost edge such that u⊆V(T) and v∉V(T);
4.   if(there is no such edge) break;
5.   add v to V(T);
6. }
7. If(T contains fewer than n-1 edges)
8.    cout << “there is no spanning tree!” <<endl;
```

- Step 3: use a **near-to-tree** data structure
  - Create an array to record the nearest distance of vertices to T.
  - Only vertices not in V(T) and adjacent to T are recorded.

Running Example

near-to-tree	0	1	2	3	4	5	6
V(T)={0}	*	28	∞	∞	∞	$\boxed{10}$	∞
V(T)={0,5}	*	28	∞	∞	$\boxed{25}$	*	∞
V(T)={0,5,4}	*	28	∞	$\boxed{22}$	*	*	24
V(T)={0,5,4,3}	*	28	$\boxed{12}$	*	*	*	18
V(T)={0,5,4,3,2}	*	$\boxed{16}$	*	*	*	*	18
V(T)={0,5,4,3,2,1}	*	*	*	*	*	*	$\boxed{14}$
V(T)={0,5,4,3,2,1,6}							



Time Complexity

- Near-to-tree
  - Step 3 : O(n)
- At most execute n rounds: $O(n^2)$

Prim's Algorithm: Correctness

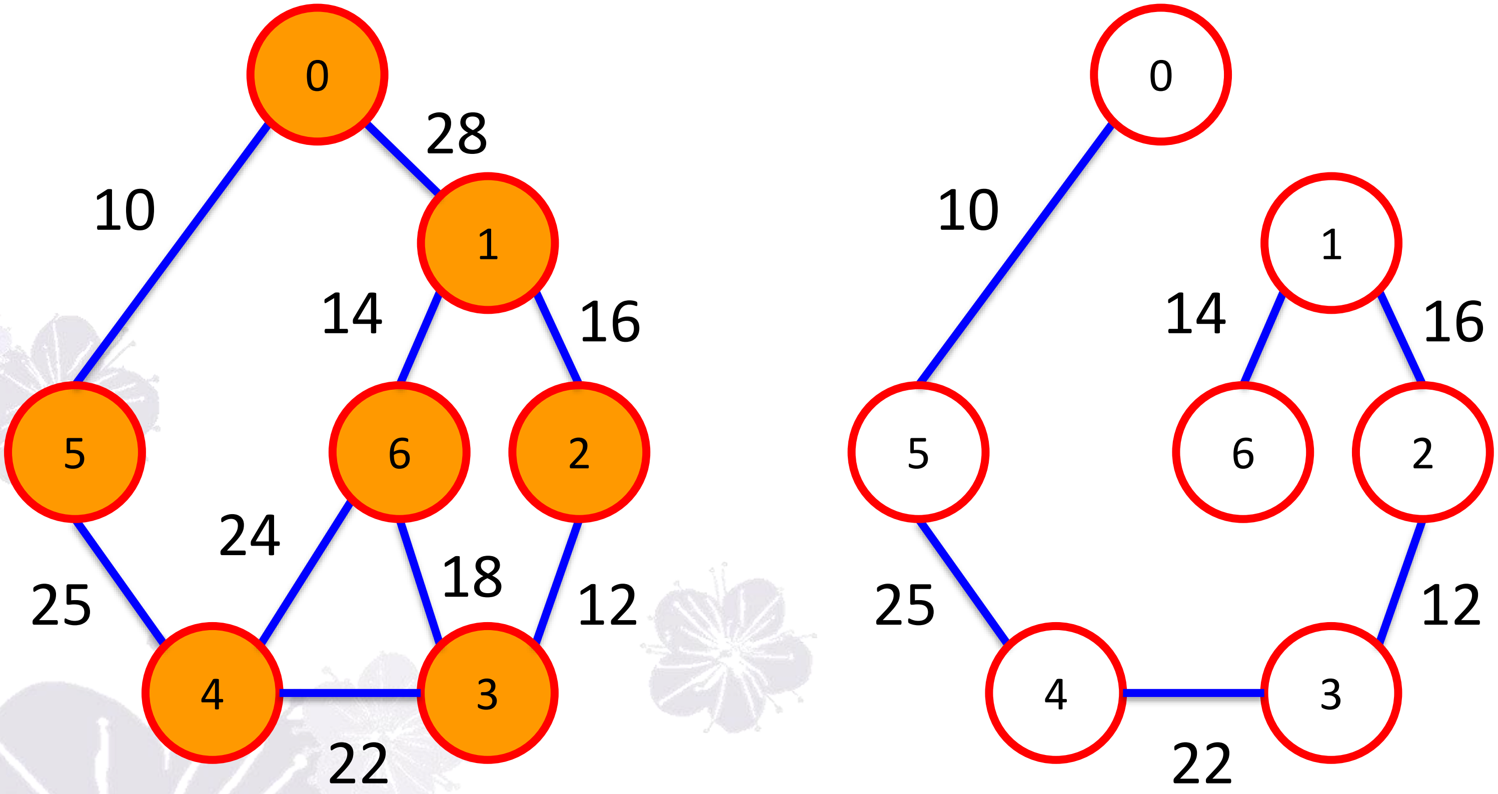
- See appendix

Sollin's Algorithm

- Idea: Select several edges at each stage.
- Step 1: Start with a forest that has n spanning trees (each has one vertex).
- Step 2: Select one minimum cost edge for each tree. This edge has exactly one vertex in the tree.
- Step 3: Delete multiple copies of selected edges and if two edges with the same cost connecting two trees, keep only one of them.
- Step 4: Repeat until we obtain only one tree.

Running Example

Refer to textbook for detailed steps!

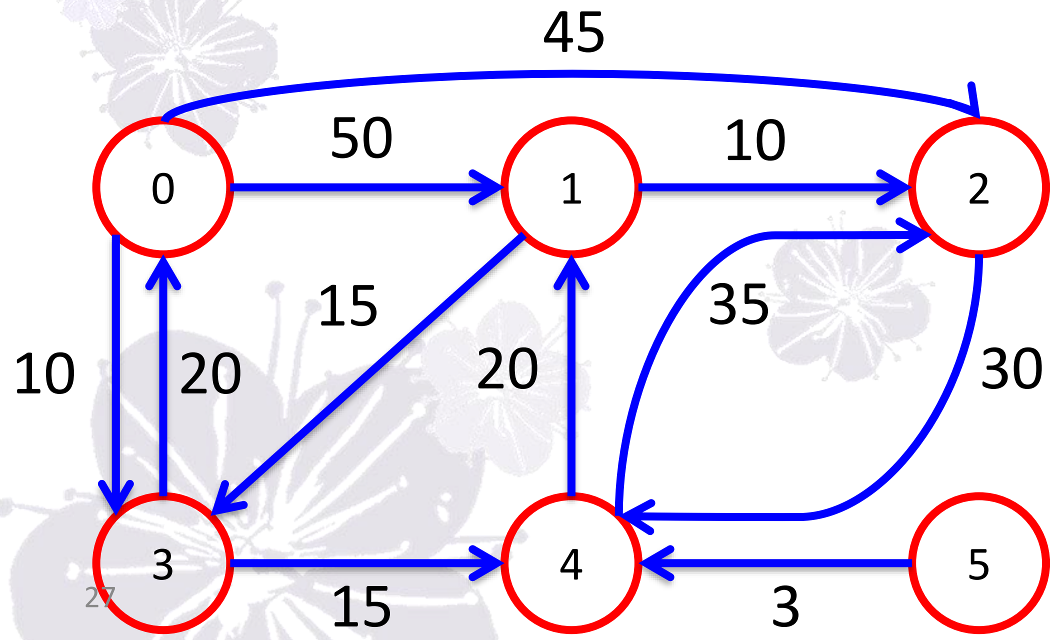


Connected graph

Spanning tree with cost 99

Single Source Shortest Paths

- Given a **digraph** with **nonnegative edge costs**, we want to compute the **shortest path** from a source vertex to all other vertices.
- **Single source/all destinations** problem.



Paths from 0 to 1:

0->1 : 50

0->2->4->1 : 95

…

0->3->4->1 : 45

Dijkstra's Algorithm

"DIKE-stra" ([ˈdaɪk.stɹə])

- Similar to Prim's algorithm
- Use a set **S** to store the vertices whose shortest path have been found
- An array **dist** is used to store the shortest distances from source V to all vertices so far
- An array **π** is used to store the vertex's predecessor
- When a new vertex w is visited, update **dist** as:

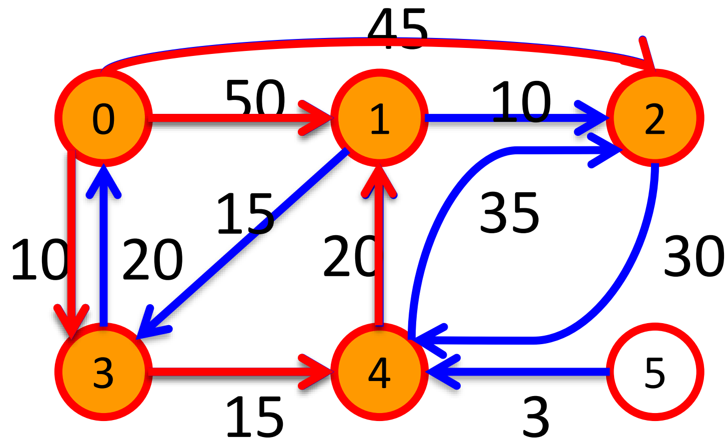
dist[w] = min(dist[u]+length(<u,w>),dist[w])

u is the previously visited vertex which is adjacent to w

Dijkstra's Algorithm

- Initialization: for **i** ϵ V, set **dist**[i]=length[v][i], **dist**[v]=0, **π**[i]=NULL
- Steps:
  - Choose vertex **u** such that i) **dist**[u] is minimum and ii) vertex **u** is not in **S**; Add **u** to **S**
  - Pick a vertex **w** not in **S**,
    if **dist**[u]+**length**[u][w]< **dist**[w],
    then update:
    - **dist**[w] = **dist**[u]+**length**[u][w]
    - **π**[w] = u
- Repeat the above steps n-1 times.

Running Example



vertex	π
0	NULL
1	NULL 4
2	NULL 0
3	NULL 0
4	NULL 3
5	NULL

S	0	1	2	3	4	5
{0}	0	50	45	10	∞	∞
{0, 3}	0	50	45	10	25	∞
{0, 3, 4}	0	45	45	10	25	∞
{0, 3, 4, 1}	0	45	45	10	25	∞
{0, 3, 4, 1, 2}	0	45	45	10	25	∞

Dijkstra - How to Find the Path

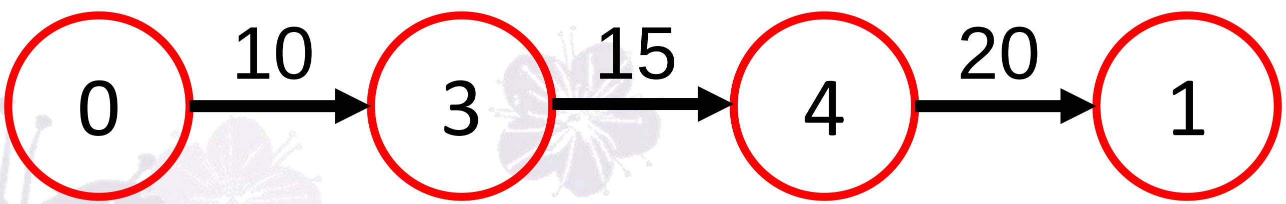
- Retrieve the path from the source vertex to any vertex **w** with the help of array **π**
- Lookup **w**'s predecessor with **π**[w] (suppose vertex **u**), and **u**'s predecessor **π**[u] and so on, until we reach the source vertex.

Dijkstra - Finding the Path

Suppose we want to find the shortest path from **0** to **1**

vertex	π
0	NULL
1	4
2	0
3	0
4	3
5	NULL

$\pi[4]=3$



Dijkstra's Algorithm

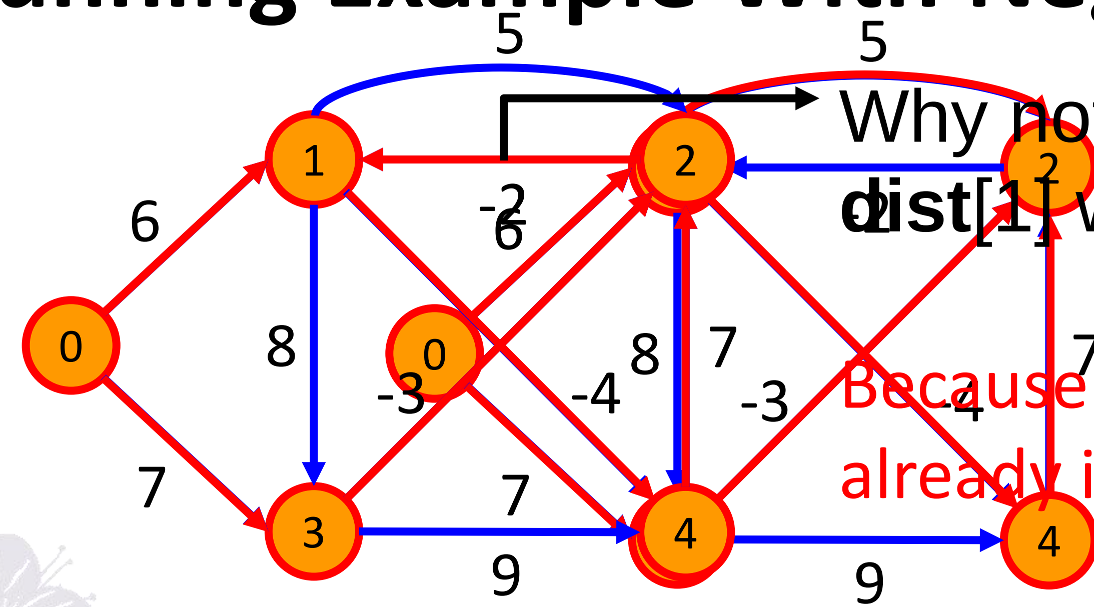
```
void MatrixWDigraph::Dijkstra(const int n, const int v)
{ // dist[j], 0 ≤ j < n, stores the shortest path from v to j
  // length[i][j] stores length of edge <i, j>
  for(int i=0; i<n; i++){ s[i]=false; dist[i]=length[v][i];
    π[i]=NULL;}
  s[v] = true;
  dist[v] = 0;
  // find n - 1 paths starting from v
  for(int i=0; i<n-1 ;i++){                            O(n)
    // Choose a vertex u, such that dist[u]
    // is minimum and s[u] = false
    int u = Choose(n);                                 O(n)
    s[u] = true;
    for(int w=0; w<n; w++){                            O(n)
      if(!s[w] && dist[u] + length[u][w] < dist[w]){
        dist[w] = dist[u] + length[u][w];
        π[w]=u;
  } // end of for (i = 0; ...)
}
```

Time complexity: $O(n^2)$

For Dijkstra algorithm, we assumed there is no edge with negative weight

What if such edges exist?

Running Example With Negative Edge



Why not update
dist[1] with this edge?

Because vertex 1 is
already in the set S.

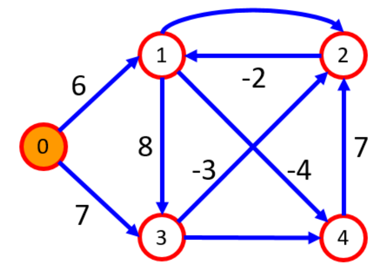
S	0	1	2	3	4
{0}	0	6	∞	7	∞
{0, 1}	0	6	11	7	2
{0, 1, 4}	0	6	9	7	2
{0, 1, 4, 3}	0	6	4	7	2
{0, 1, 4, 3, 2}	0	6	4	7	2

vertex	π
0	NULL
1	NULL 0
2	NULL 4
3	NULL 0
4	NULL 1

Steps:

- Choose vertex **u** such that i) **dist**[u] is minimum and ii) vertex **u** is not in the **S**; Add **u** to **S**
- Pick a vertex **w** not in the S, if **dist**[u]+**length**[u][w]< **dist**[w], then update:
  - **dist**[w] = **dist**[u]+**length**[u][w]
  - π[w] = u

Dijkstra Went Wrong



Dijkstra finds shortest path from 0 to 4 as:

$$0 \xrightarrow{6} 1 \xrightarrow{-4} 4$$

vertex	π
0	NULL
1	0
2	3
3	0
4	1

The correct shortest path from 0 to 4 should be:

$$0 \xrightarrow{7} 3 \xrightarrow{-3} 2 \xrightarrow{-2} 1 \xrightarrow{-4} 4$$

Dijkstra can’t handle graphs with negative edges

Bellman-Ford Algorithm

- Works when edge weights may be negative
- An array **dist** is used to store the shortest distances from source to all vertices so far
- An array **π** is used to store the vertex's predecessor
- Relaxes all edges at most |V|-1 times
- Ability to detect negative cycles
- update **dist[]** using the equation:

dist[w] = min(dist[u]+length(<u,w>),dist[w])

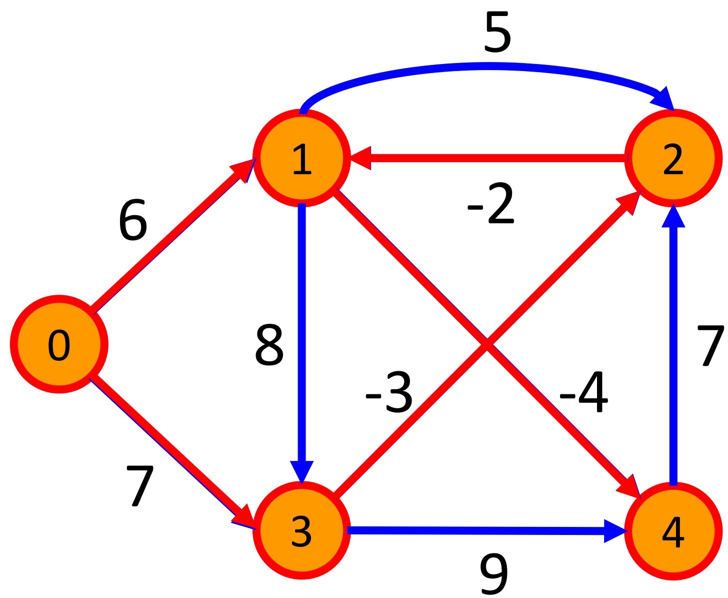
u is the vertex adjacent to w

Bellman-Ford Algorithm

- Initialize: for **i** ϵ V, set **dist**[i]=∞, **π**[i]=NULL
- For source **v**, **dist**[v]=0
- Step:
  - For each edge **<u,w>** ϵ E,
    if **dist**[u] + **length**[u][w] < **dist**[w], then update
    - **dist**[w] = **dist**[u]+**length**[u][w]
    - **π**[w] = u
- Repeat the above step |V|-1 times
- Check whether the graph has a negative cycle

Running Example

i=4



vertex	π
0	NULL
1	NULL 2
2	NULL 3
3	NULL 0
4	NULL 1

0	1	2	3	4
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

<u,w> = <1,4>
dist[1] + **length**[1][4] < **dist**[4]?
2 + -4 < 2 **YES!** Update **dist**[4] = -2

- Step:
  - For each edge **<u,w>** ϵ E, if **dist**[u] + **length**[u][w] < **dist**[w], then update
    - **dist**[w] = **dist**[u]+**length**[u][w]
    - **π**[w] = u
- Repeat the above step |V|-1 times

Bellman-Ford - How to Find the Path

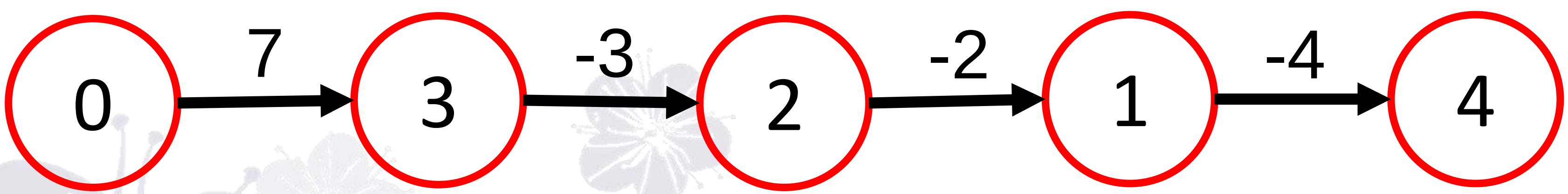
- After the algorithm, we can find the shortest path from the source vertex to a vertex **w** with the array **π**
- We use **π**[w] to find vertex **w**’s predecessor (suppose vertex **u**) and **u**’s predecessor and so on, until the source vertex is reached

Bellman-Ford - Find the Path (Similar to Dijkstra)

Suppose we want to find the shortest path from 0 to 4

$\pi[4]=1$

vertex	π
0	NULL
1	2
2	3
3	0
4	1



Bellman-Ford Algorithm

```
bool MatrixWDigraph::Bellman_Ford (const int n, const int v)
{ // dist[j], 0 ≤ j < n, stores the shortest path from v to j
  // length[i][j] stores length of edge <i, j>
  // π[i] stores the predecessor of i
  for(int i=0; i<n; i++){ π[i]=NULL; dist[i]=∞;}// initialize
  dist[v] = 0;
  // find n - 1 paths starting from v
  for(int i=1; i<=n-1 ;i++){                    O(n)
    for each edge <u,w> ϵ E                     O(|E|)
      if(dist[u] + length[u][w] < dist[w]){
        dist[w] = dist[u] + length[u][w];
        π[w]=u;
      }
  } // end of for (i = 1; …)
  for each edge <u,w> ϵ E                       O(|E|)
      if(dist[u] + length[u][w] < dist[w])
        return false; // have a negative cycle
  return true;
}
```

Time complexity: $O(n|E|)$

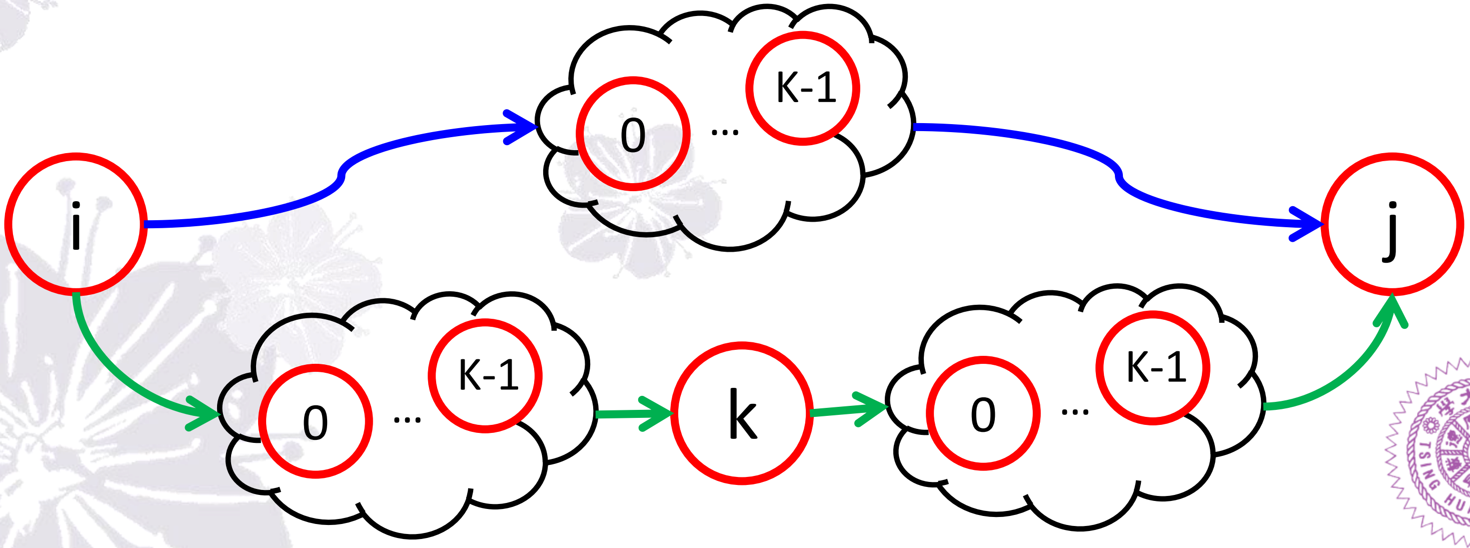
All-Pairs Shortest Paths

- One approach: Applying single source shortest path to each of n vertices
- Another approach: Floyd-Warshall's algorithm
- We number the vertices from 0 to n-1, and maintain an array **A**
  - $\mathbf{A}^{-1}[i][j]$: is just the length[i][j]
  - $\mathbf{A}^{n-1}[i][j]$: the length of the shortest i-to-j path in G
  - $\mathbf{A}^{k}[i][j]$: the length of the shortest path from i to j going **through no intermediate vertex of index greater than k**
- $\mathbf{A}^{k}[i][j] = \min\{\mathbf{A}^{k-1}[i][j], \mathbf{A}^{k-1}[i][k] + \mathbf{A}^{k-1}[k][j]\}, k \geq 0$

Floyd-Warshall's Algorithm

- There are only two possible paths for $A^k[i][j]$!
  - The path dose not pass vertex k.
  - The path dose pass vertex k.

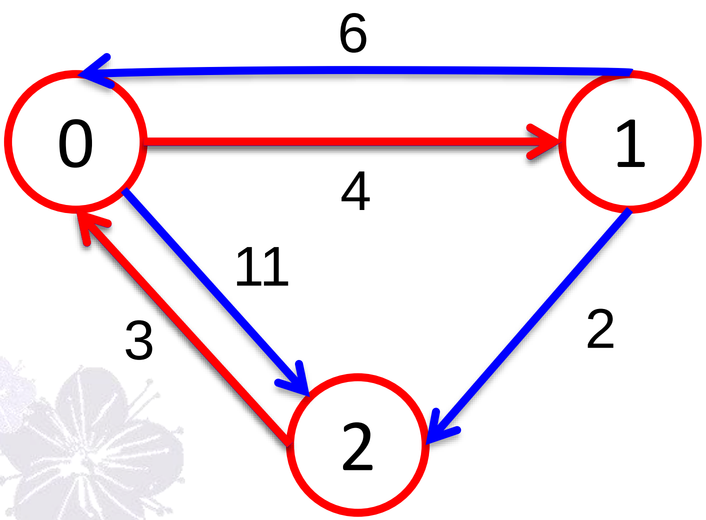
$$\boxed{A^k[i][j] = \min\{ A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j] \}, k \geq 0}$$



Floyd-Warshall's Algorithm

- Array **A** stores the shortest distance between vertex **i** and **j** in **V**
- Array **p** stores the vertices in the path from vertex **i** to **j**
- Initialize: Set $\mathbf{A}^{-1}$[i][j] = **length**[i][j], **p**[i][j]=-1
- For **k**=0 to n-1, if $\mathbf{A}^{k-1}$[i][k]+ $\mathbf{A}^{k-1}$ [k][j] < $\mathbf{A}^{k-1}$[i][j], update $\mathbf{A}^{k}$[i][j] = $\mathbf{A}^{k-1}$[i][k]+ $\mathbf{A}^{k-1}$ [k][j], **p**[i][j] = k
- Finally $\mathbf{A}^{n-1}$[i][j] is the shortest distance from vertex **i** to **j**

Running Example



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

p	0	1	2
0	-1	-1	-1
1	-1	-1	-1
2	-1	-1	-1

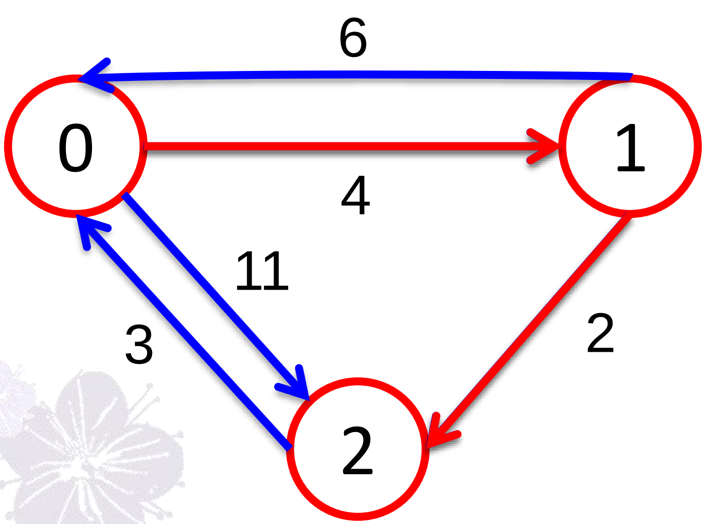
$A^0[2][1] = \min(A^{-1}[2][1], A^{-1}[2][0]+A^{-1}[0][1])$

$A^0[2][1] = \min(\infty, 3+4) = 7$

$A^0[1][2] = \min(A^{-1}[1][2], A^{-1}[1][0]+A^{-1}[0][2])$

$A^0[1][2] = \min(2, 6+11) = 2$

Running Example



A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

p	0	1	2
0	-1	-1	-1
1	-1	-1	-1
2	-1	0	-1

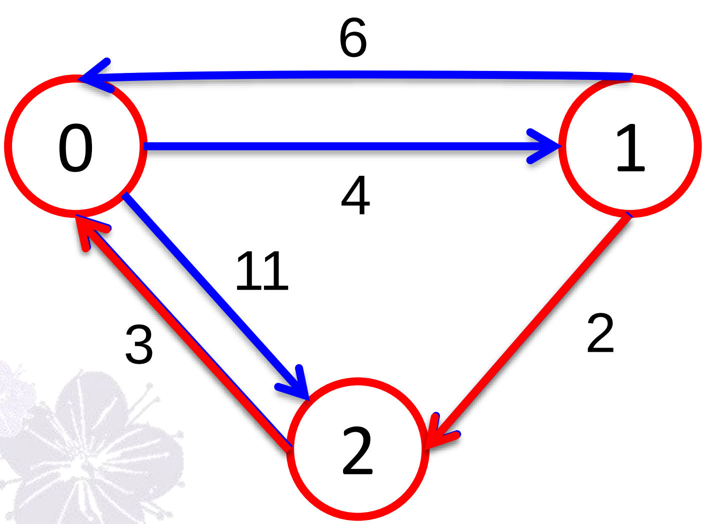
$A^1[2][0] = \min(A^0[2][0], A^0[2][1]+A^0[1][0])$

$A^1[2][0] = \min(3, 7+6) = 3$

$A^1[0][2] = \min(A^0[0][2], A^0[0][1]+A^0[1][2])$

$A^1[0][2] = \min(11, 4+2) = 6$

Running Example



A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

p	0	1	2
0	-1	-1	1
1	-1	-1	-1
2	-1	0	-1

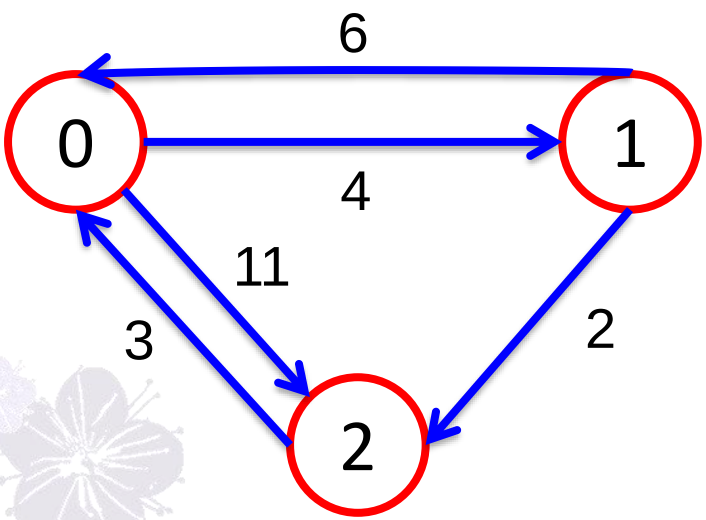
$A^2[0][1] = \min(A^1[0][1], A^1[0][2]+A^1[2][1])$

$A^2[0][1] = \min(4, 6+7) = 4$

$A^2[1][0] = \min(A^1[1][0], A^1[1][2]+A^1[2][0])$

$A^2[1][0] = \min(6, 2+3) = 5$

Running Example



A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

p	0	1	2
0	-1	-1	1
1	2	-1	-1
2	-1	0	-1

Floyd-Warshall - How to Find the Path

- With the help of array **p**
- If **p**[i][j] = -1, no vertex is needed to go through for the shortest path from **i** to **j**
- Otherwise, lookup **p**[i][j] to find vertex required to go thorugh (suppose vertex **k**), and then find the shortest path from **i** to **k** and from **k** to **j**

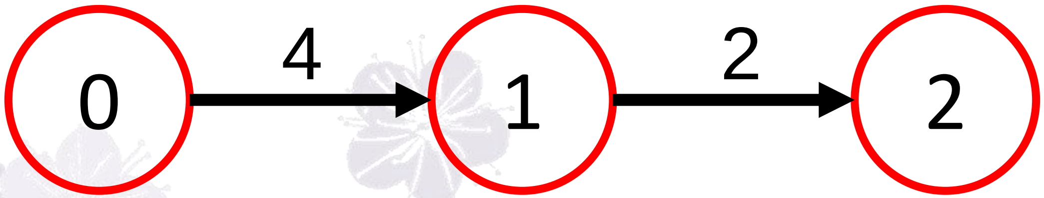
Floyd-Warshall find the path

Suppose we want to find the shortest path from 0 to 2

p[0][2]=1

p	0	1	2
0	-1	-1	1
1	2	-1	-1
2	-1	0	-1

p[0][1]=-1 p[1][2]=-1

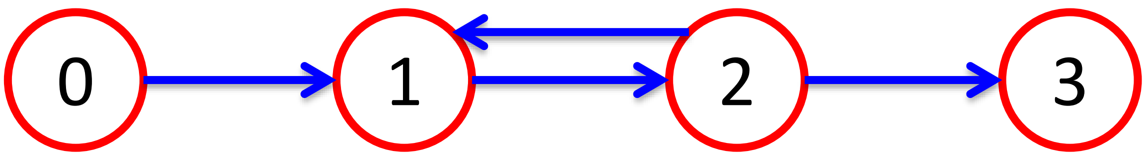


Floyd-Warshall's Algorithm

```
void MatrixWDigraph::AllLengths(const int n)
{// length[n][n] stores edge length between
 // adjacent vertices
 // a[i][j] stores the shortest path from i to j
 for (int i = 0; i<n; i++)------------> O(n)
  for (int j = 0; j<n; j++)-----------> O(n)
      a[i][j]= length[i][j];
 // path with top vertex index k
 for (int k= 0; k<n; k++) ------------> O(n)
  // all other possible vertices
  for (int i= 0; i<n; i++)------------> O(n)
   for (int j= 0; j<n; j++) ----------> O(n)
    if((a[i][k]+a[k][j])<a[i][j]){
      a[i][j] = a[i][k] + a[k][j];
      p[i][j] = k;
    }
 }
```

Time complexity: $O(n^3)$

Transitive Closure



A^+	0	1	2	3
0	0	1	1	1
1	0	1	1	1
2	0	1	1	1
3	0	0	0	0

Transitive closure matrix

A^*	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	1	1	1
3	0	0	0	1

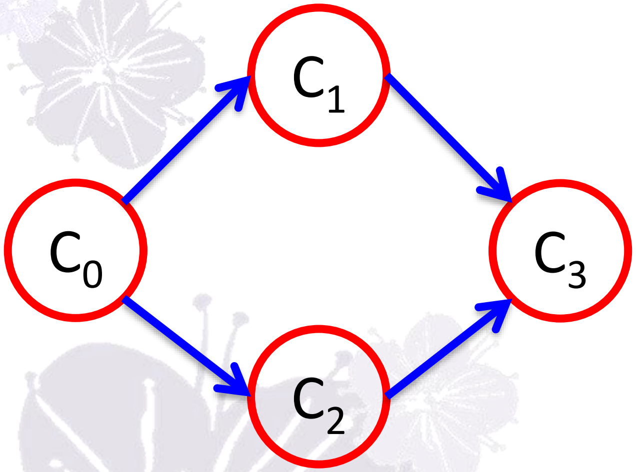
Reflexive transitive closure matrix

Transitive Closure

- The **transitive closure matrix A^+**:
  - **A^+** is a matrix such that **A^+**[i][j] = 1 if there is **a path of length > 0 from i to j** in the graph; otherwise, **A^+**[i][j] = 0.
- The **reflexive transitive closure matrix A***:
  - **A*** is a matrix such that **A***[i][j] = 1 if there is **a path of length >= 0 from i to j** in the graph; otherwise, **A***[i][j] = 0.
- Use Floyd-Warshall's algorithm!
  - A^k[i][j] = A^{k-1}[i][j] || (A^{k-1}[i][k] && A^{k-1} [k][j]);

Activity-on-Vertex (AOV) Networks

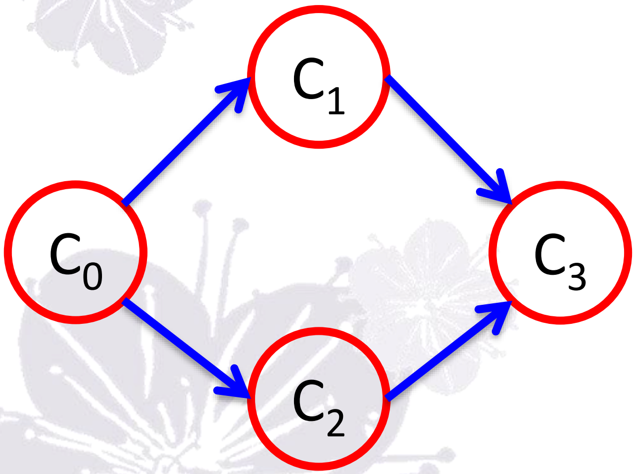
- A digraph G where the vertices represent tasks or activities and the edges represent precedence relations between tasks.



Predecessor :
Vertex i is a predecessor of vertex j, iff there is a directed path from vertex i to vertex j.

AOV Network

- **Topological order**:
  - A **linear ordering** of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering.



$C_0 \to C_1 \to C_2 \to C_3$ (O)

$C_0 \to C_2 \to C_1 \to C_3$ (O)

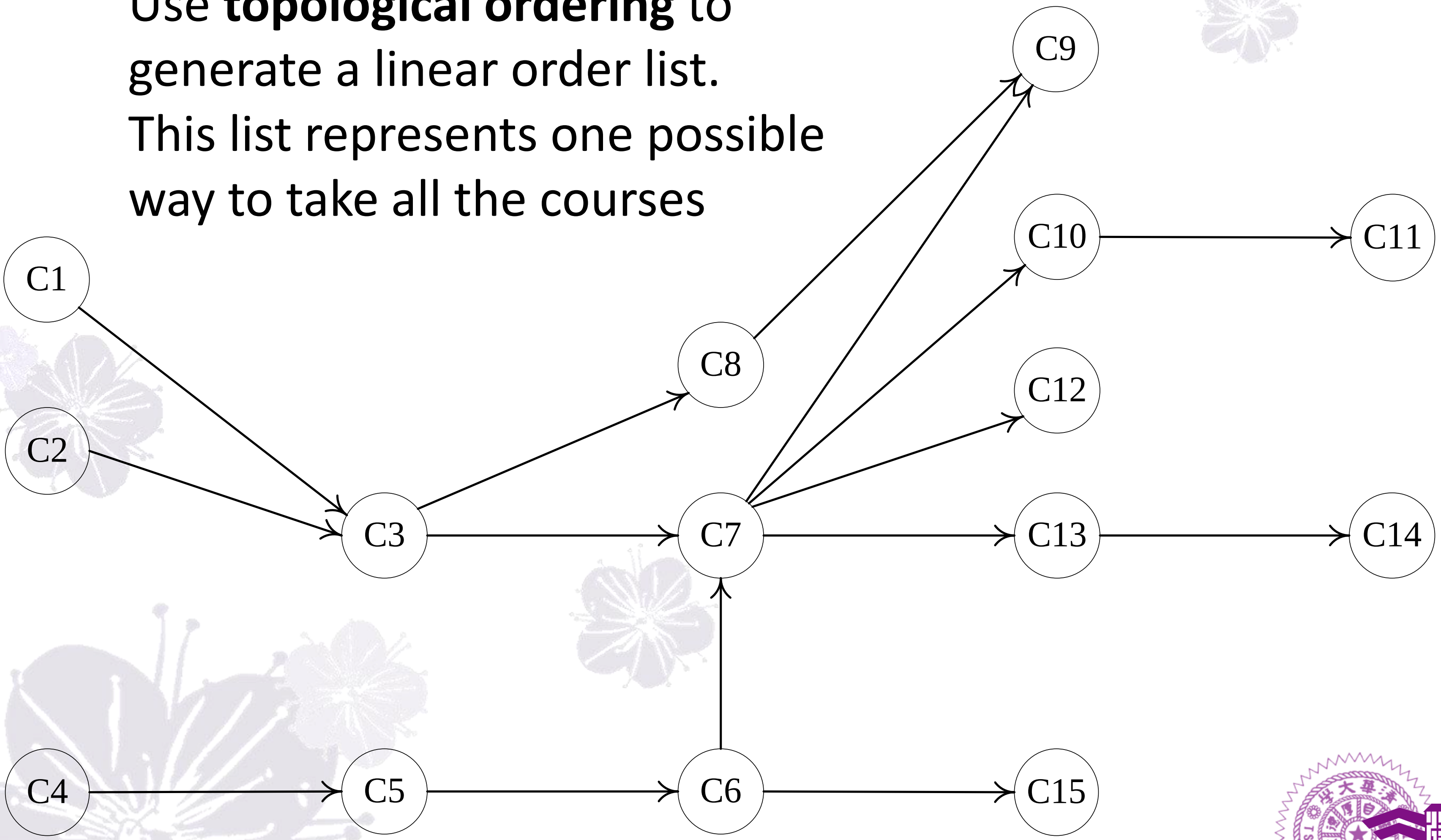
$C_0 \to C_2 \to C_3 \to C_1$ (X)

Application

Course No.	Course	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

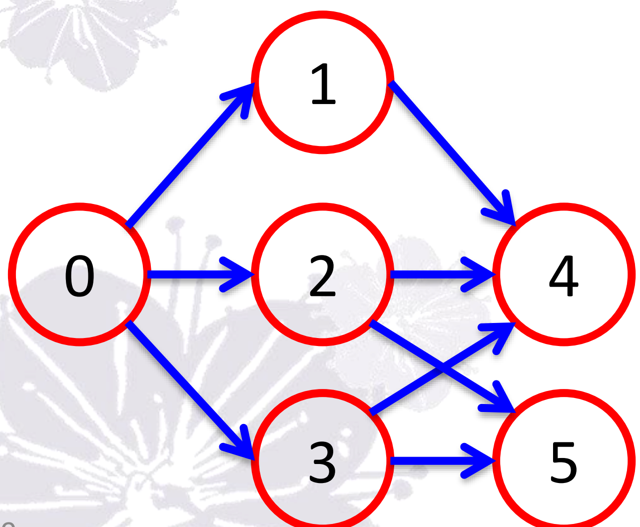
AOV Network of Courses

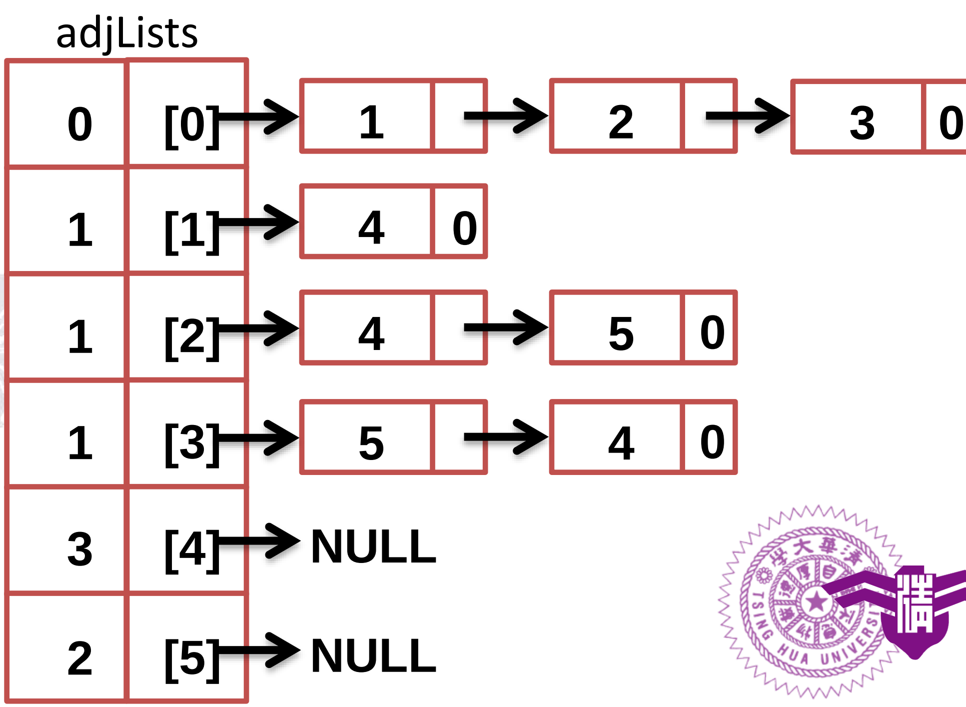
Use **topological ordering** to generate a linear order list. This list represents one possible way to take all the courses



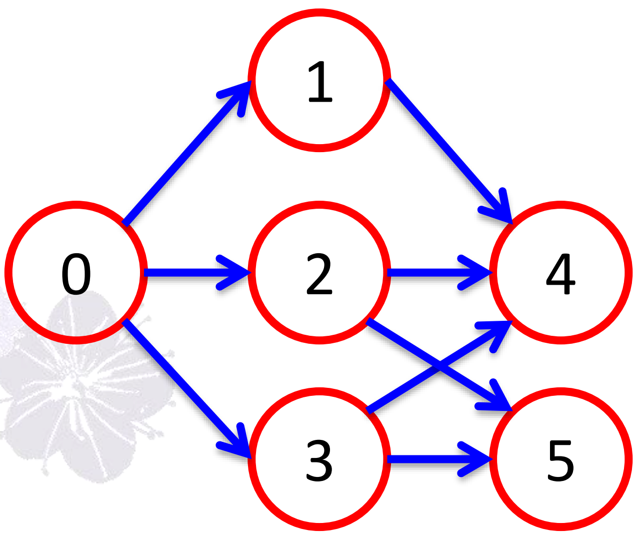
Topological Ordering

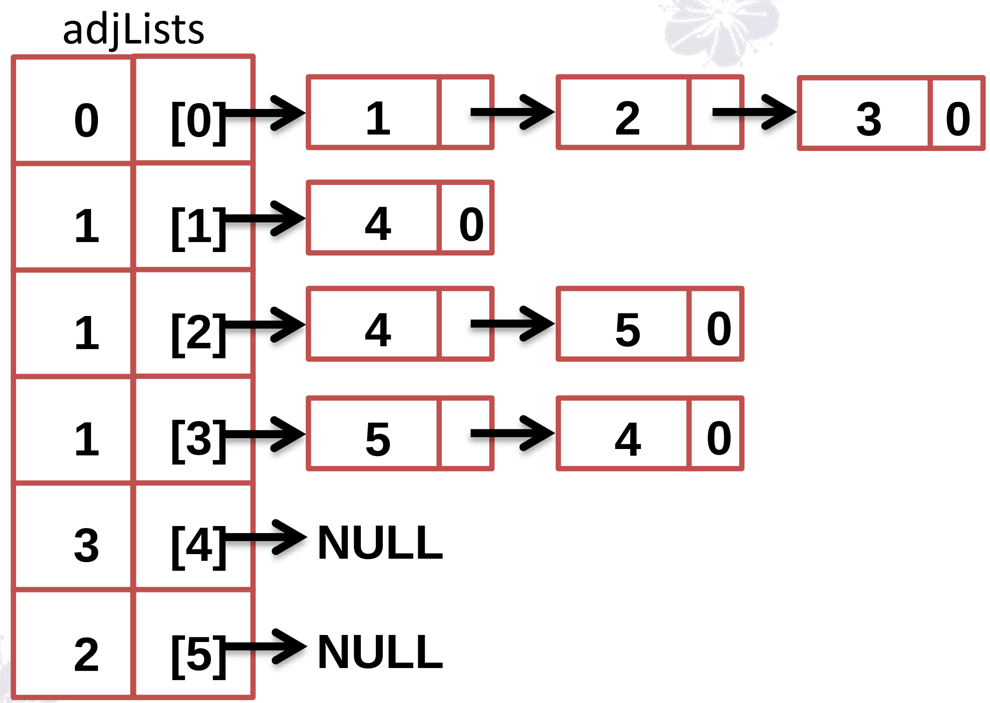
- Iteratively pick a vertex v that has no predecessors.
  - Use an additional field “count” to record the “in-degree” value of each vertex.





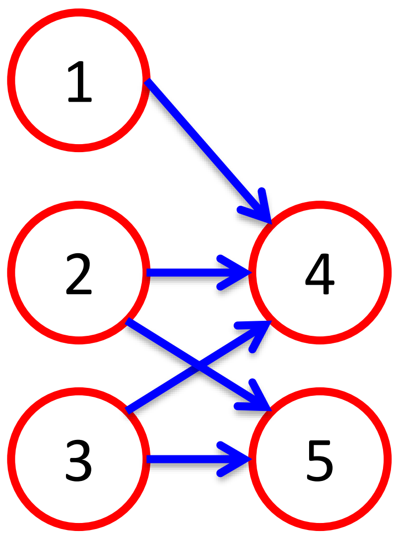
Running Example

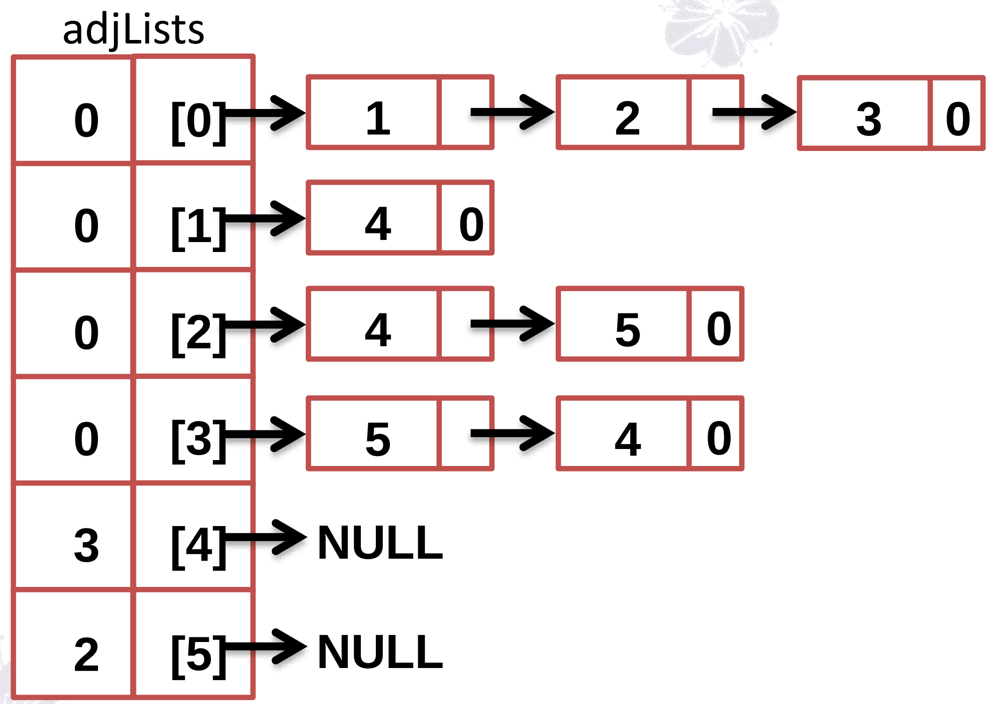




Ordered list:

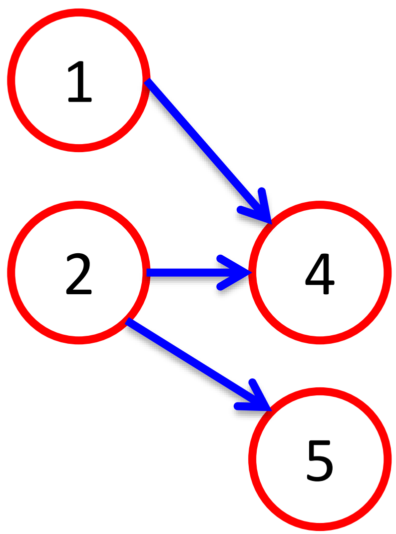
Running Example

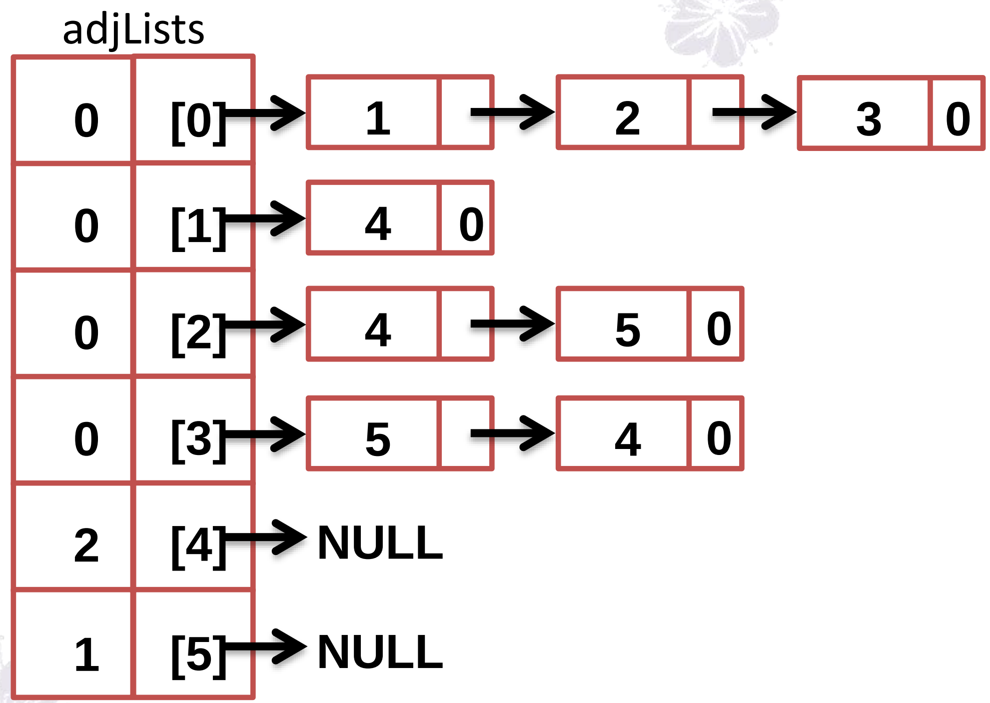




Ordered list: 0

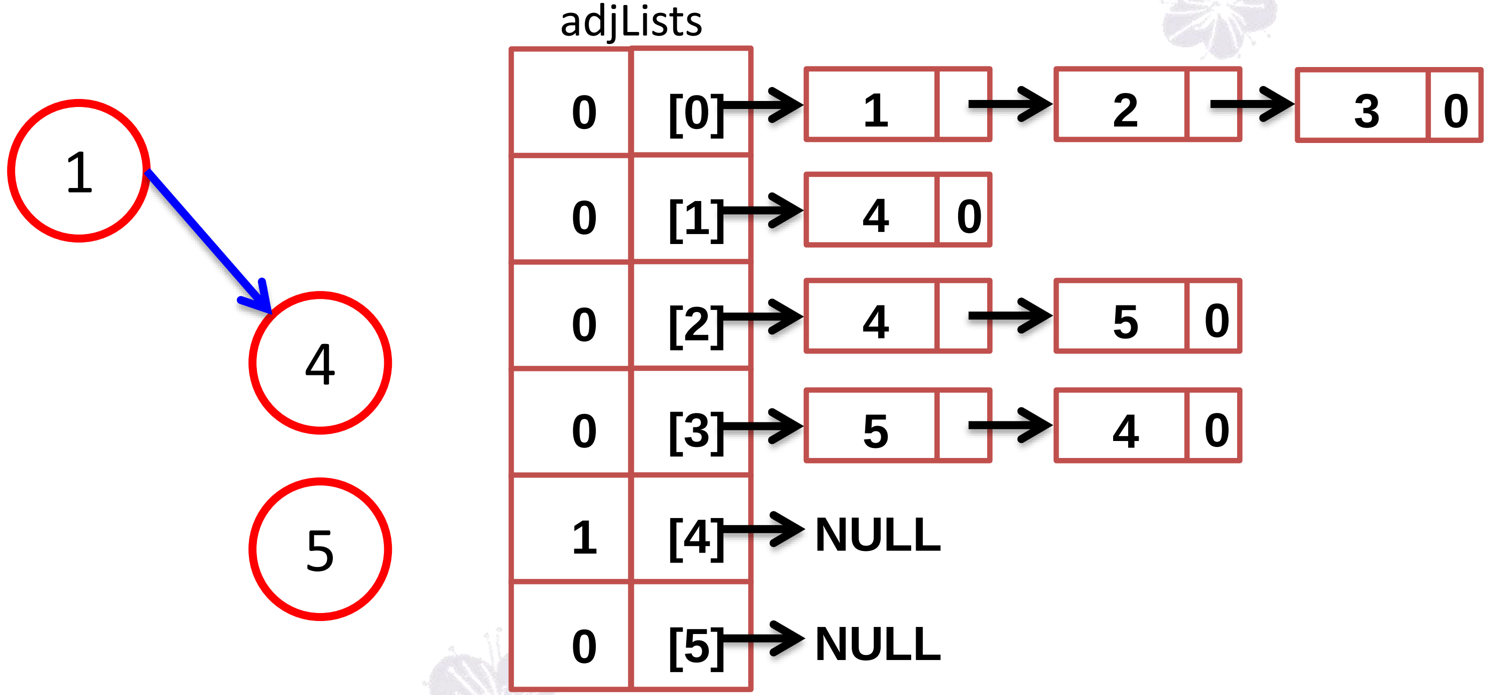
Running Example





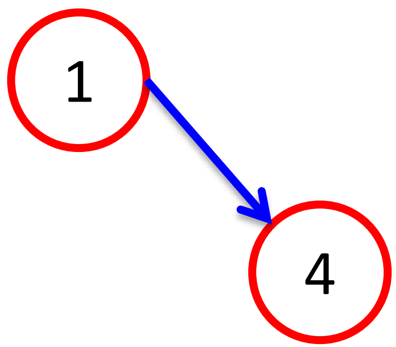
Ordered list: 0 3

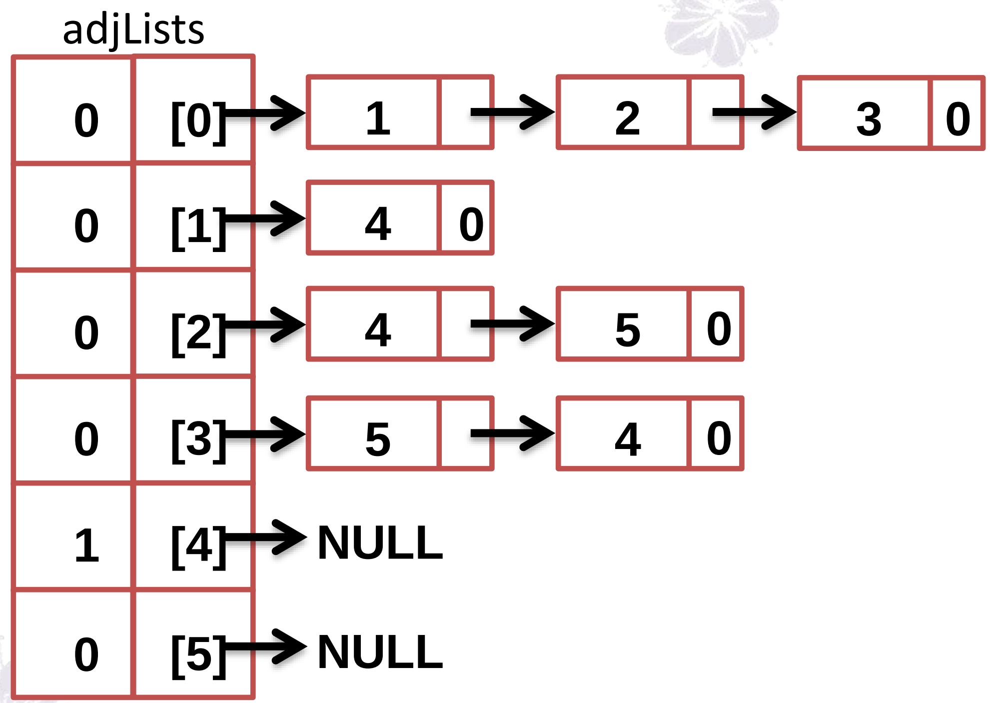
Running Example



Ordered list: 0 3 2

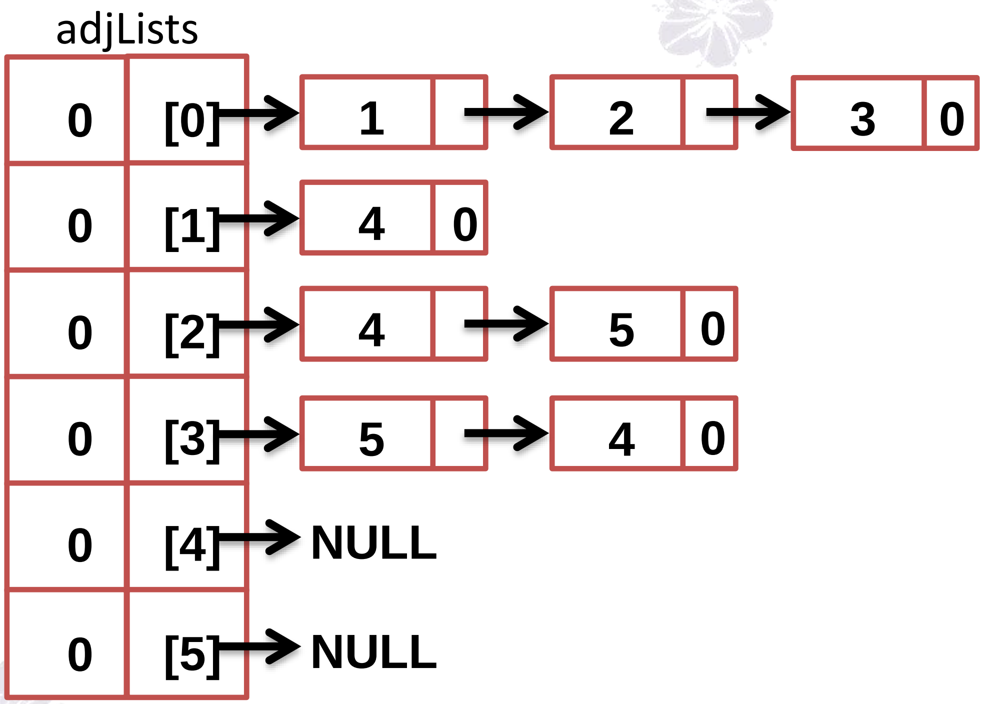
Running Example





Ordered list: 0 3 2 5

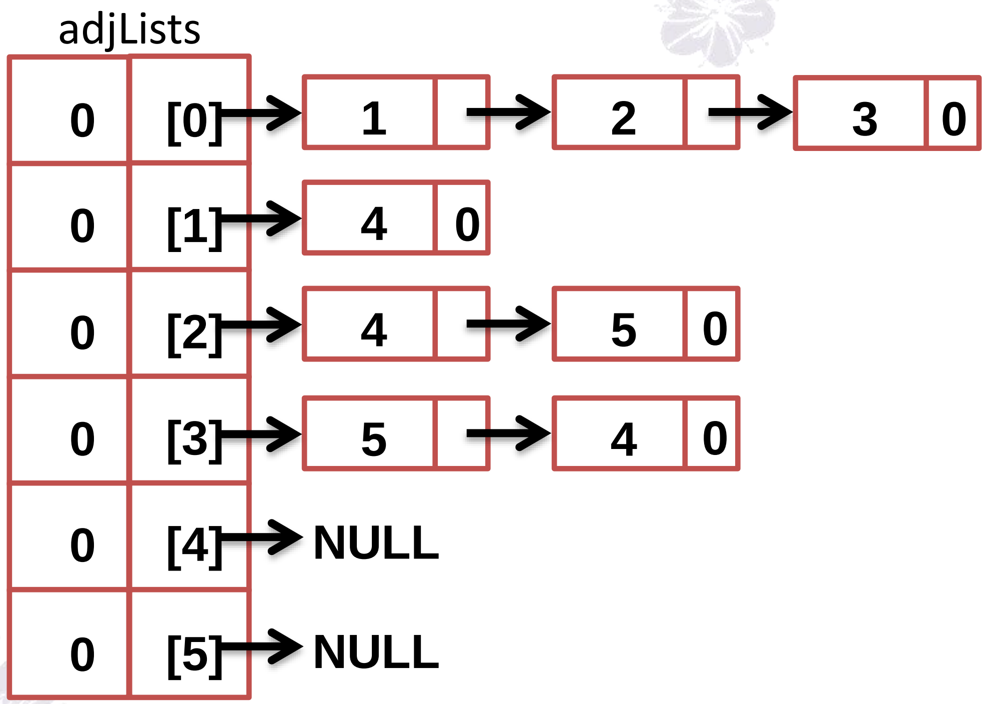
Running Example



4

Ordered list: 0 3 2 5 1

Running Example

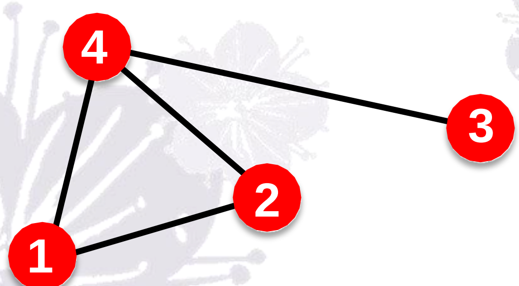


Ordered list: 0 3 2 5 1 4

Intuition: Powers of Adj Matrices

- Computing #paths between two nodes
  - **Recall:** $\mathbf{A}_{\mathbf{uv}} = 1$ if $u \in N(v)$
  - Let $\boldsymbol{P}_{\boldsymbol{uv}}^{(\boldsymbol{K})} =$ #paths of length $\boldsymbol{K}$ between $\boldsymbol{u}$ and $\boldsymbol{v}$
  - We will show $\boldsymbol{P}^{(\boldsymbol{K})} = \boldsymbol{A}^{\boldsymbol{k}}$
  - $\boldsymbol{P}_{\boldsymbol{uv}}^{(\mathbf{1})} =$ #paths of length 1 (direct neighborhood) between u and $v = \boldsymbol{A}_{\boldsymbol{uv}}$

$$\boldsymbol{P}_{\mathbf{12}}^{(\mathbf{1})} = \boldsymbol{A}_{\mathbf{12}}$$



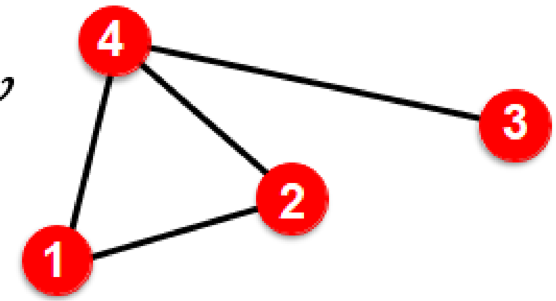
$$A = \begin{pmatrix} 0 & \boxed{1} & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Intuition: Powers of Adj Matrices

- **How to compute $P_{uv}^{(2)}$?**
  - **Step 1:** Compute **#paths** of length 1 **between each of u's neighbor and v**
  - **Step 2**: **Sum up** these #paths across u's neighbors
  - $P_{uv}^{(2)} = \sum_i A_{ui} * P_{iv}^{(1)} = \sum_i A_{ui} * A_{iv} = A^2{}_{uv}$

Intuition: Powers of Adj Matrices

$$P^{(2)}_{uv} = \boxed{\sum_i A_{ui} * P^{(1)}_{iv}} = \sum_i A_{ui} * A_{iv} = A^2{}_{uv}$$



從u到某個node i是否存在path (either 1 or 0) 乘上 從node i到v的path個數。把所有i的情況加起來

Example: we'd like to compute $P^{(2)}_{12}$, i.e., *u=1, v=2*

$$P^{(2)}_{12} = \sum_i A_{1i} * P^{(1)}_{i2} = A_{11} * P^{(1)}_{12} + A_{12} * P^{(1)}_{22} + A_{13} * P^{(1)}_{32} + A_{14} * P^{(1)}_{42}$$

= 0*1+1*0+0*0+1*1 = 1

Node 1's neighbors

#paths of length 1 between Node 1's neighbors and Node 2

$$P^{(2)}_{12} = A^2_{12}$$

$$A^2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 3 \end{pmatrix}$$

Power of adjacency

Global Neighborhood Overlap

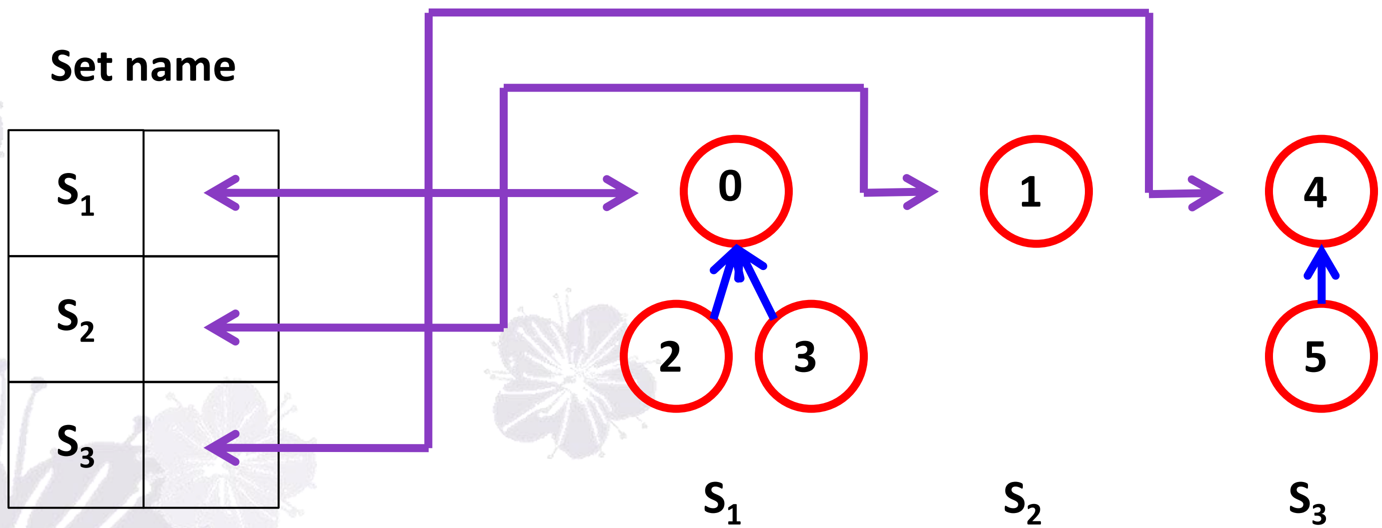
- **Katz index:** count the number of paths of all lengths between a pair of nodes.

- How to compute #paths between two nodes?
- Use **adjacency matrix powers**!
  - $\boldsymbol{A}_{\mathbf{uv}}$ specifies #paths of length 1 (direct neighborhood) between u and v.
  - $\boldsymbol{A}^{2}_{\mathbf{uv}}$ specifies #paths of **length 2** (neighbor of neighbor) between u and v.
  - And, $\boldsymbol{A}^{\boldsymbol{l}}_{\mathbf{uv}}$ specifies #paths of **length $\boldsymbol{l}$**.

APPENDIX – RECAP OF SET UNION

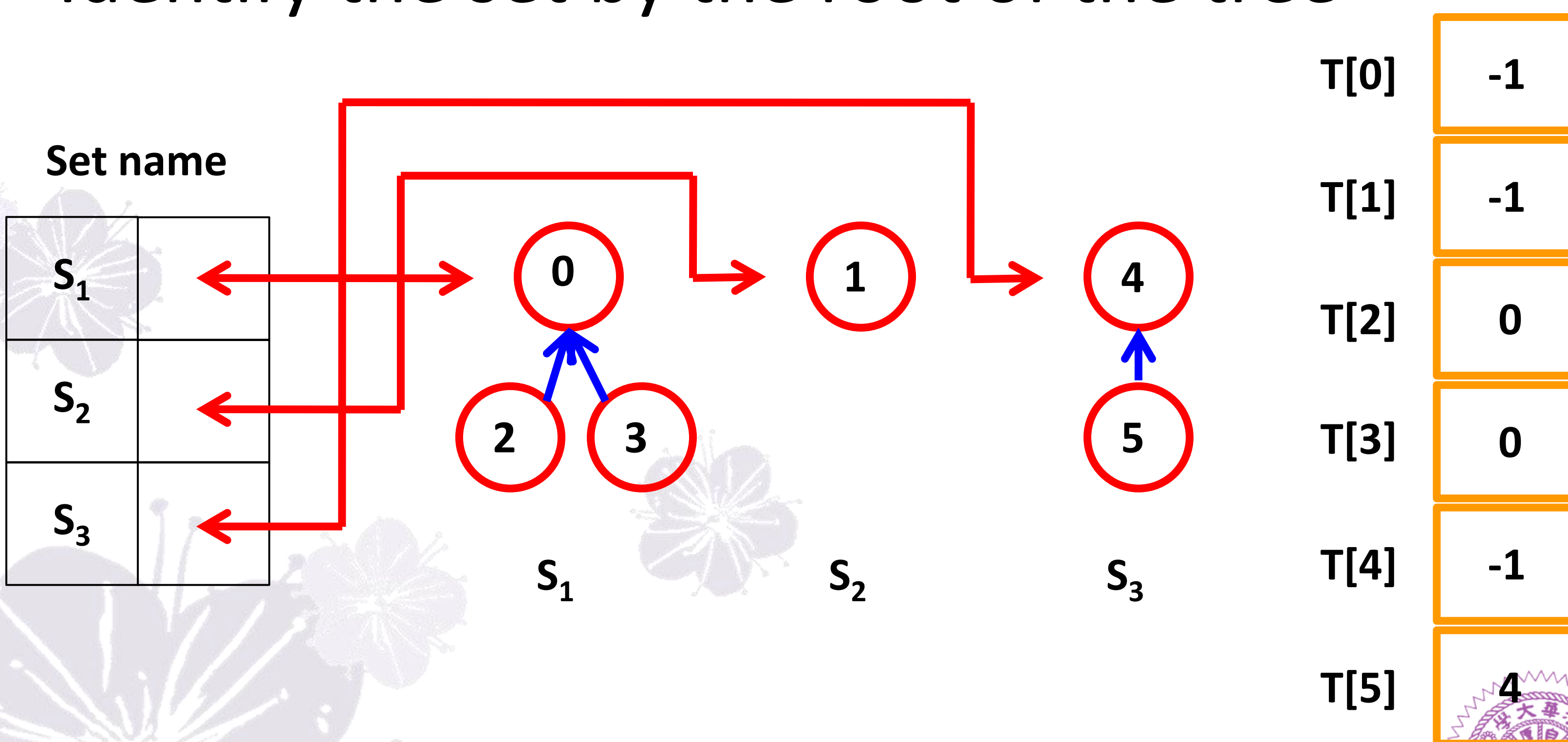
DS: Tree Representation

- Link elements of a subset to form a tree
  - Link children to root
  - Link root to set name



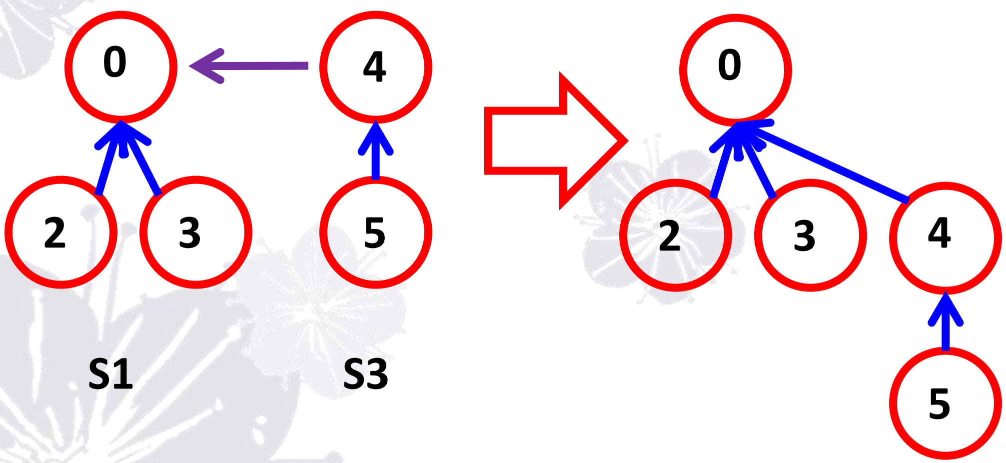
DS: Tree Representation

- Use an array to store the tree
- Identify the set by the root of the tree



DS Operation: Union(S_i, S_j)

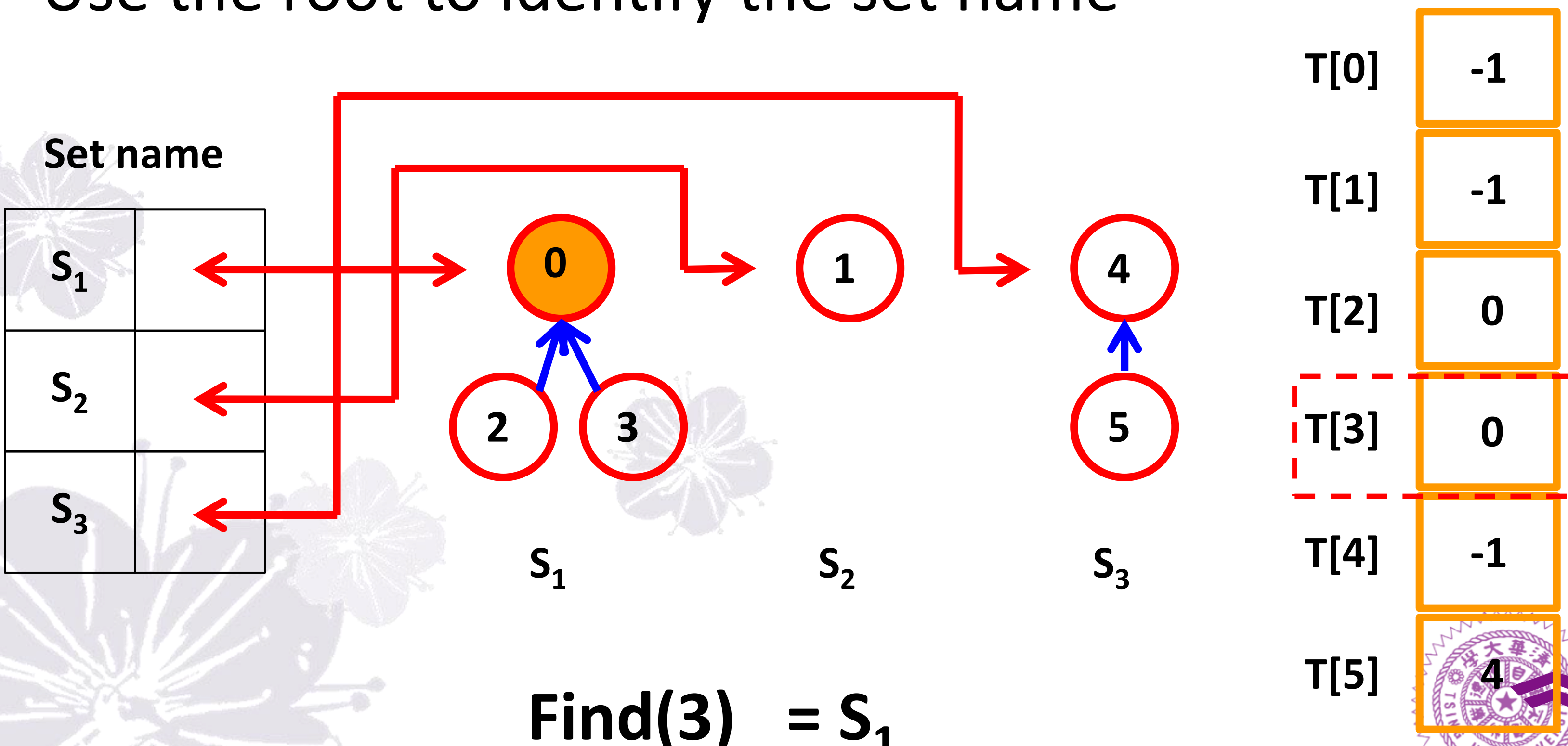
- Set the parent field of one of the root to the other root
  - S_1=Union(S_1, S_3)
  - Time complexity : O(1)



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

DS Operation: Find(x)

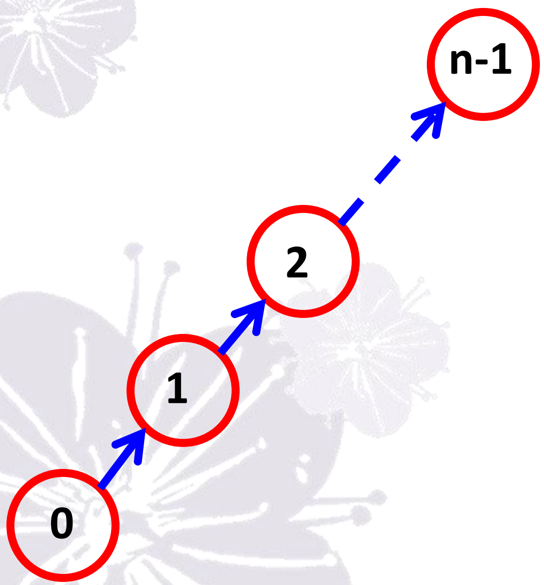
- Following the index starting at x and tracing the tree structure until reaching a node with parent value = -1
- Use the root to identify the set name



Find(3) = S_1

DS Time Complexity

- S = { 0, 1, 2, … , n-1 }
  - S_1 = { 0 }, S_2 = { 1 }, S_3 = { 2 }, … , S_n = { n-1 }
- Perform a sequence Union
  - Union(S_2, S_1), Union(S_3, S_2), …, Union(S_n, S_{n-1})

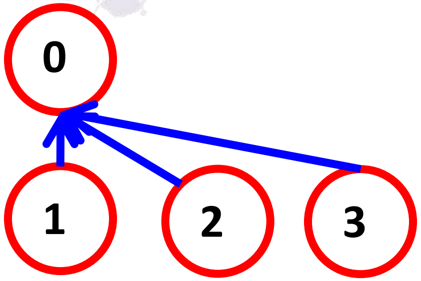


Followed by a sequence of Find
Find(0), Find(1), …, Find(n-1)

Time Complexity = $\sum_{i=1}^{n} i = O(n^2)$

Improved Union(S_i, S_j)

- Do not always merge two sets into the first set
- Adopt a ***Weighting rule*** to union operation
  - $S_i = S_i \cup S_j$, if $| S_i | >= | S_j |$
  - $S_j = S_i \cup S_j$, if $| S_i | < | S_j |$
- $S = \{ 0, 1, 2, \ldots , n \}$
  - $S_1 = \{ 0 \}$, $S_2 = \{ 1 \}$, $S_3 = \{ 2 \}$, … , $S_n = \{ n-1 \}$
  - Union (1, 2)->Union (1, 3)->Union (1, 4)



Maximum Tree Height

- Lemma 5.5
  - Let T be a tree with m nodes created by a sequence of weighting unions.
    The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$
- Proof with Induction:
  - 1) ***m=1*** is true
  - 2) Assume it is true for all trees with ***i*** nodes, ***i<=m-1***

- We'd like to show that it is also true for ***i=m***
- Let ***T*** be a tree with **m** nodes created by function *WeightedUnion*. Consider the last union operation performed on ***Union(k,j)***
- Let ***a*** be the number of nodes in tree ***j*** and ***(m-a)*** the number in ***k***. Wlog, we may assume ***1<=a<=m/2***.
- Then, the height of ***T*** is either 1) the same as that of ***k*** *(m-a>a)* or 2) is <u>one more</u> than that of ***j*** *(m-a=a)*
- For case 1, $height(T)<=floor(\log_2(m-a))+1<=floor(\log_2 m)$
- For case 2, $height(T)<=floor(\log_2 a)+2<=floor(\log_2 m/2)+2$ $<=floor(\log_2 m)+1$

Prim’s Algorithm - Correctness

- Prove with induction.
- **Hypothesis:** After each iteration, the tree **T** is a subgraph of some minimum spanning tree **M**.
- At iteration 1, this is trivially true because **T** is a single vertex.
- Suppose that at iteration **k**, we have **T** which is a subgraph of **M**, and Prim’s Algorithm tells us to add the edge e.
- We need to prove that **T U {e}** is also a subtree of some MST (not necessarily **M**).

Step 1: Start with a tree T contains a single arbitrary vertex.

Step 2: Among all edges, add a least cost edge (u,v) to T such that T U (u,v) is still a tree.

Step 3: Repeat step 2 until T contains n-1 edges.

Prim’s Algorithm - Correctness

- To prove: **T U {e}** is also a subtree of some MST.
- If $e \in M$ => this is clearly true
- If $e \notin M$. Then if we add **e** to **M**, we create a cycle. Since **e** has one endpoint in **T** and one endpoint not in **T**, there has to be some other edge **e’** in this cycle that has exactly one endpoint in **T**.

Step 1: Start with a tree T contains a single arbitrary vertex.
Step 2: Among all edges, add a least cost edge (u,v) to T such that T U (u,v) is still a tree.
Step 3: Repeat step 2 until T contains n-1 edges.

Prim's Algorithm - Correctness

- Therefore, Prim's Algorithm could have added **e'** but instead chose to add **e**, which means that **w(e')>=w(e)**. So if we add **e** to **M** and remove **e'**, we create a new tree **M'** whose total weight is at most the weight of **M**, and which contains **T U {E}**. This maintains the induction, so proves the theorem.

Bellman-Ford how to find the path

- (In fact, **w(e')=w(e)** must hold. Otherwise **M'** would have weight less than **M**, contradicting the assumption that **M** is an MST.

Step 1: Start with a tree T contains a single arbitrary vertex.

Step 2: Among all edges, add a least cost edge (u,v) to T such that T U (u,v) is still a tree.

Step 3: Repeat step 2 until T contains n-1 edges.