CS 2351 Data Structures

Graphs (II)

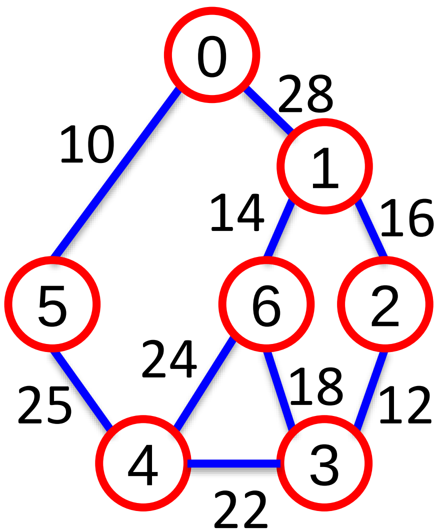
Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University

Outline

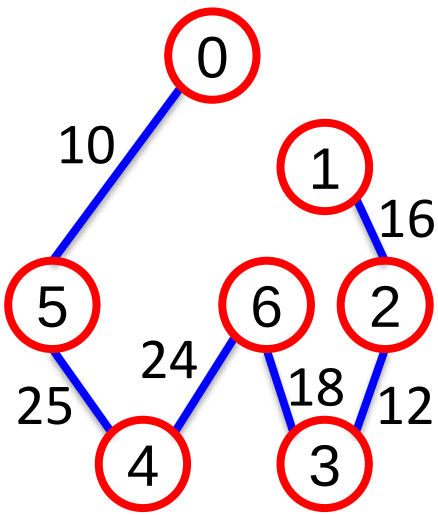
- Minimum cost spanning tree (Sec. 6.3)
  - Kruskal’s algorithm
  - Prims’s algorithm
  - Sollin’s algorithm
- Shortest path and transitive closure (Sec. 6.4)
  - Single source/all destination: non-negative edge costs
  - All-pairs shortest paths
  - Transitive closure
- Activity networks (Sec. 6.5)
  - Activity-on-vertex (AOV) networks

Minimum-Cost Spanning Trees

- For a weighted undirected graph, find a spanning tree with **the sum of the weights (costs) of the edges being minimum**
- Three greedy algorithms:
  - Kruskal's algorithm
  - Prims's algorithm
  - Sollin's algorithm



Spanning tree with cost 105



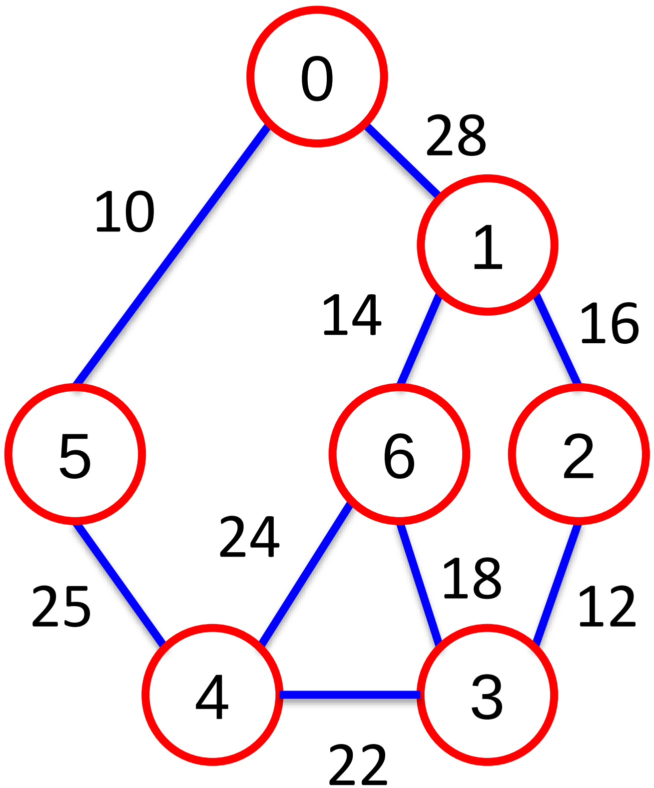
Kruskal's Algorithm

Idea: add edges to the tree one at a time according to their edge costs, from the smallest to the largest

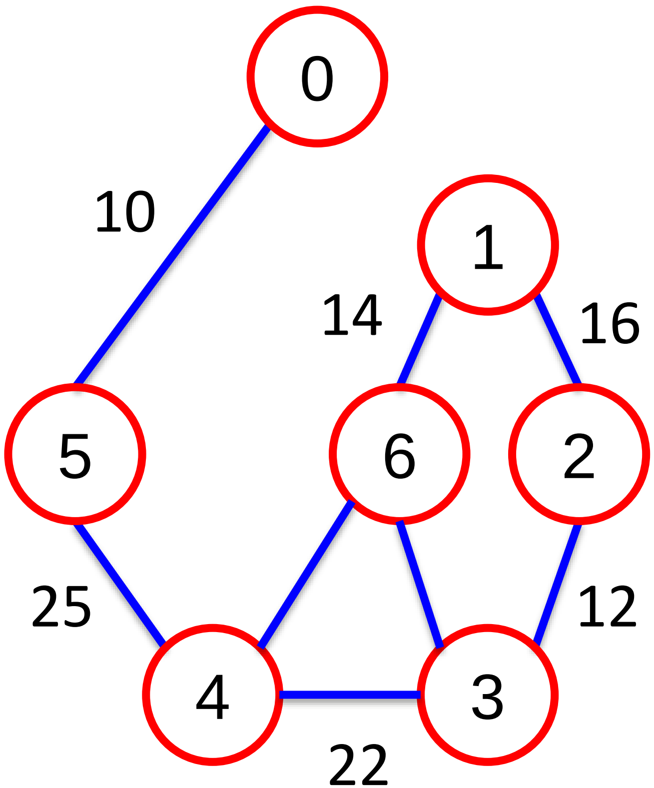
- Step 1: find an edge with the minimum cost
- Step 2: if it creates a cycle to the edges already selected, discard the edge; otherwise, select the edge
- Step 3: repeat steps 1 and 2 until we select n-1 edges

Running Example

Refer to the textbook for detailed steps



Connected graph



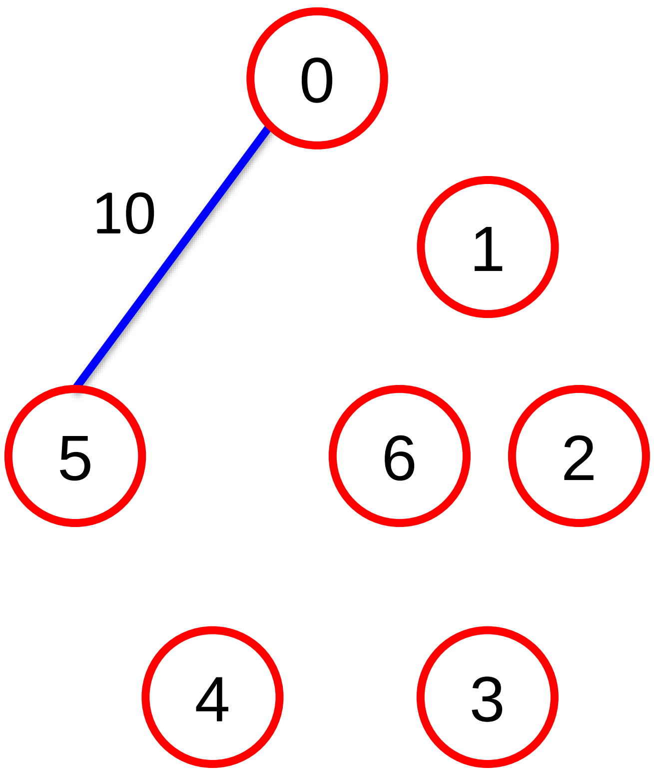
Spanning tree with cost 99

Kruskal's Algorithm

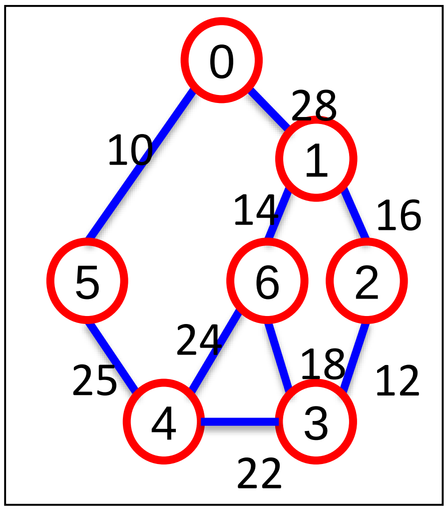
```
1. T = ϕ
2. While((T has fewer than n-1 edges)&&(E is not empty)){
3.    choose an edge (v,w) from E with the minimum cost;
4.    delete (v,w) from E;
5.    if((v,w) does not create a cycle) add (v,w) to T;
6.    else discard (v,w);
7. }
8. If(T contains fewer than n-1 edges)
9.    cout << "no spanning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

- Steps 3 and 4: use a *min heap* to store edge cost
- Step 5: use *disjoint sets* representation (Sec. 5.10) for intermediate trees, one set for each partial tree
  - For an edge (v,w) to be added, if v and w are in the same set, discard the edge; else union two corresponding sets

Running Example

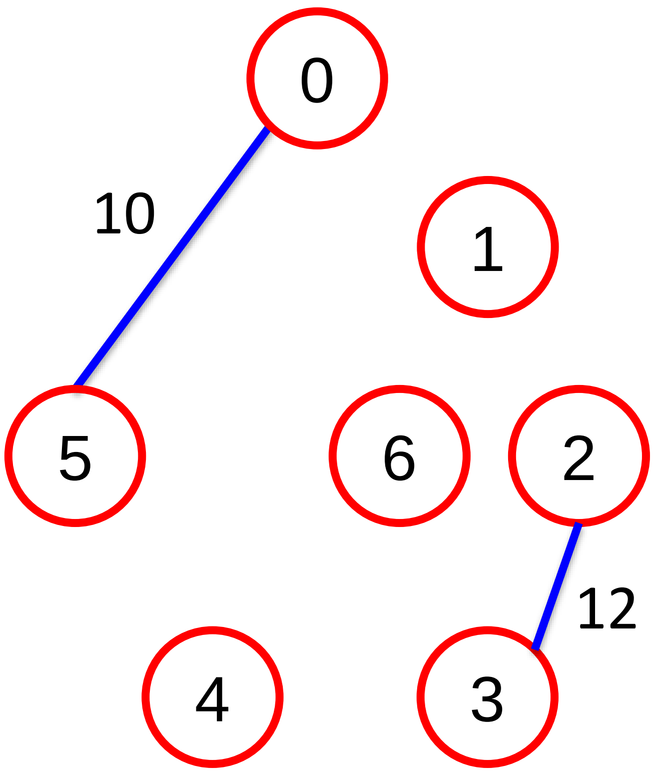


Disjoint sets



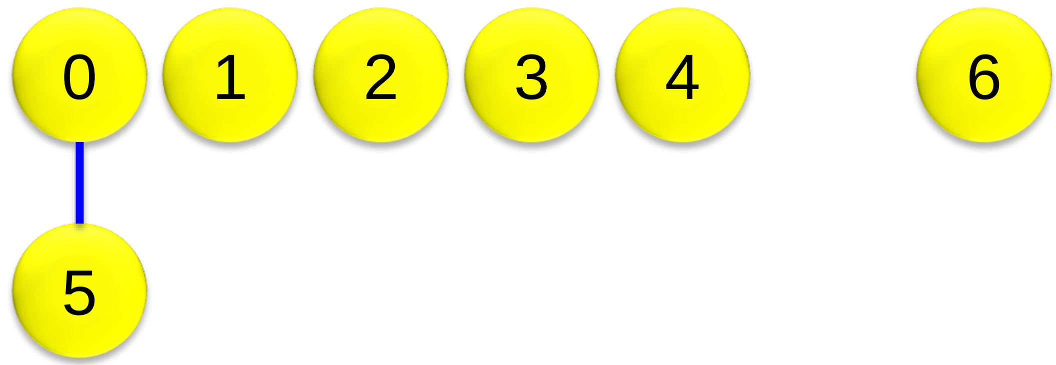
Spanning tree with cost 99

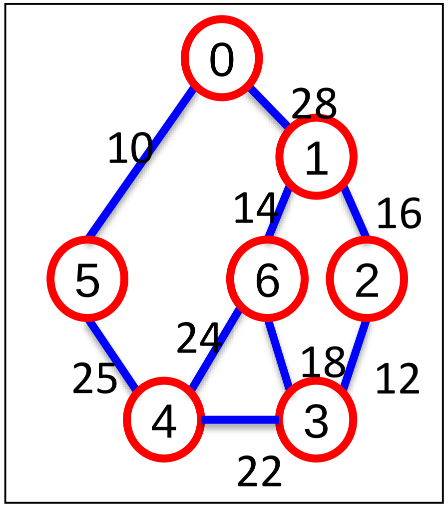
Running Example



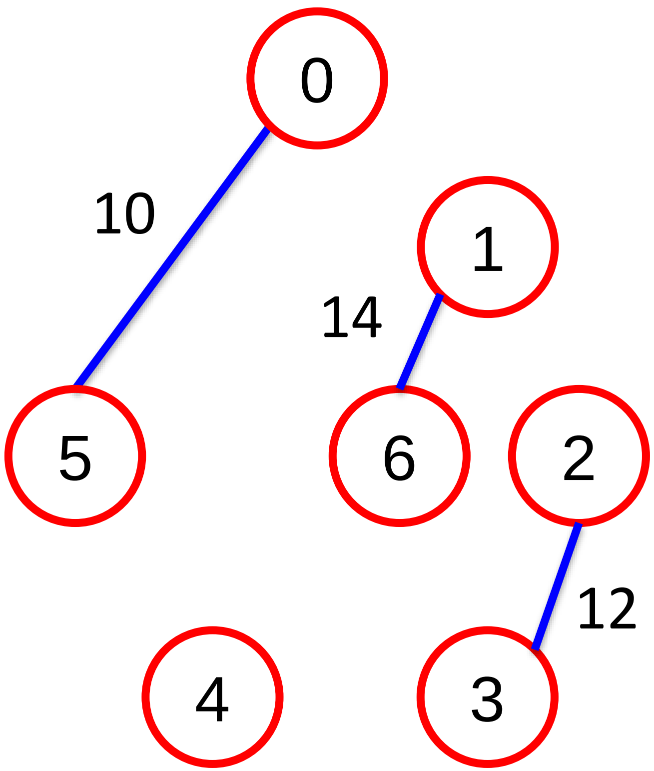
Spanning tree with cost 99

Disjoint sets



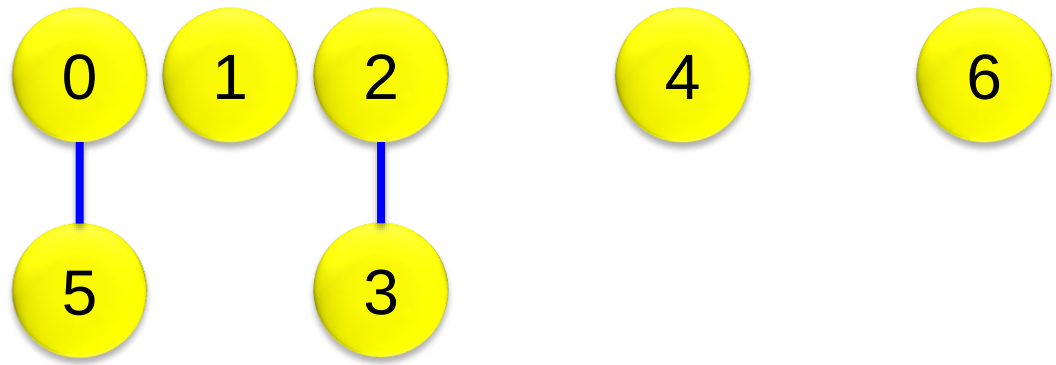


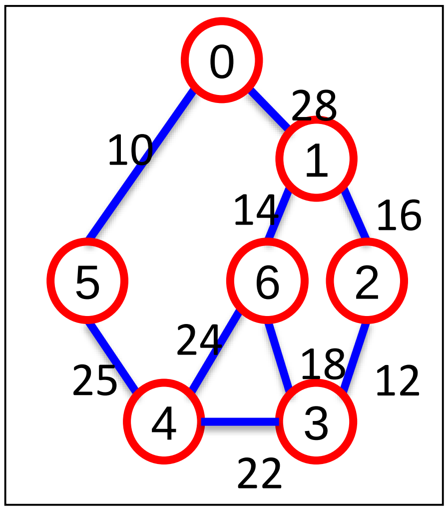
Running Example



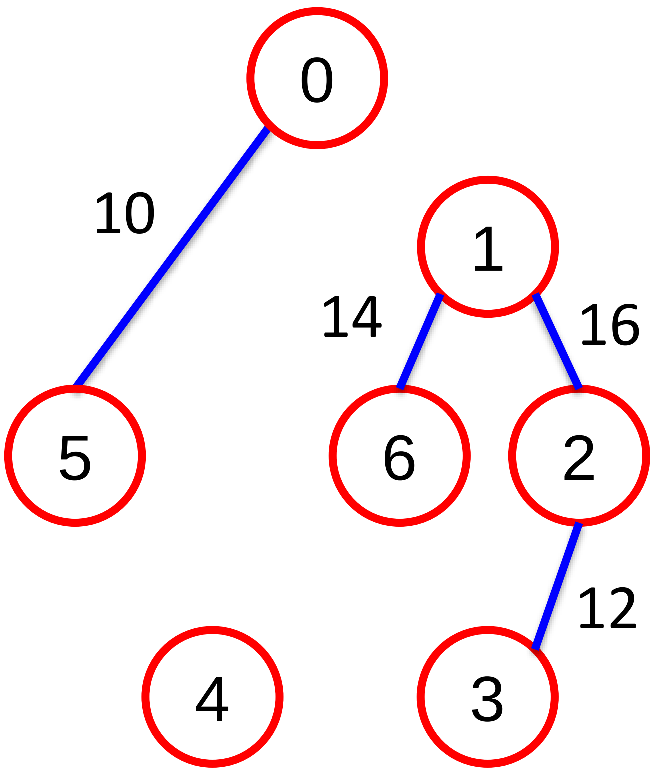
Spanning tree with cost 99

Disjoint sets



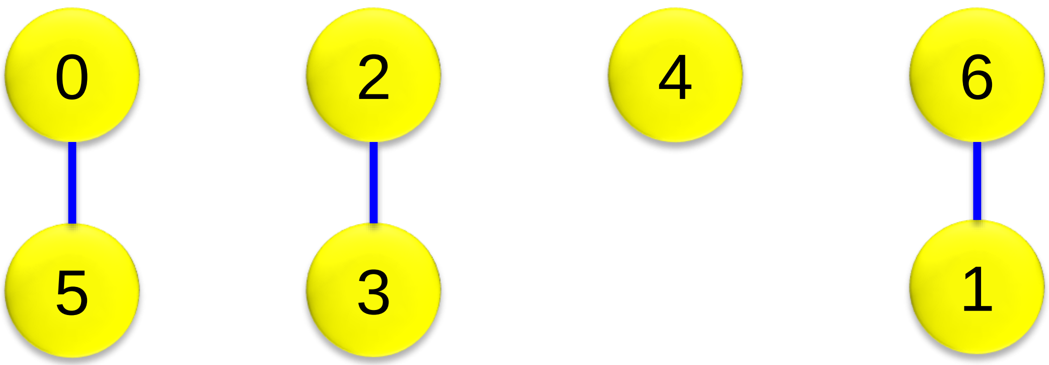


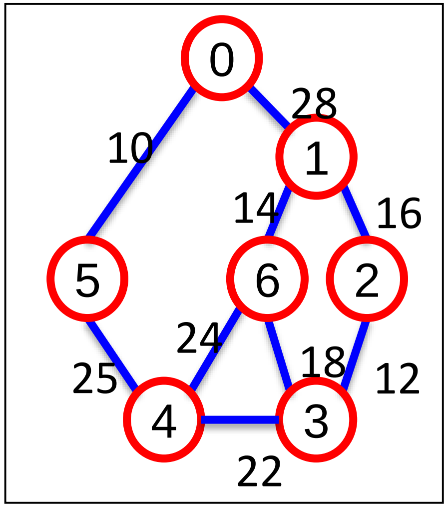
Running Example



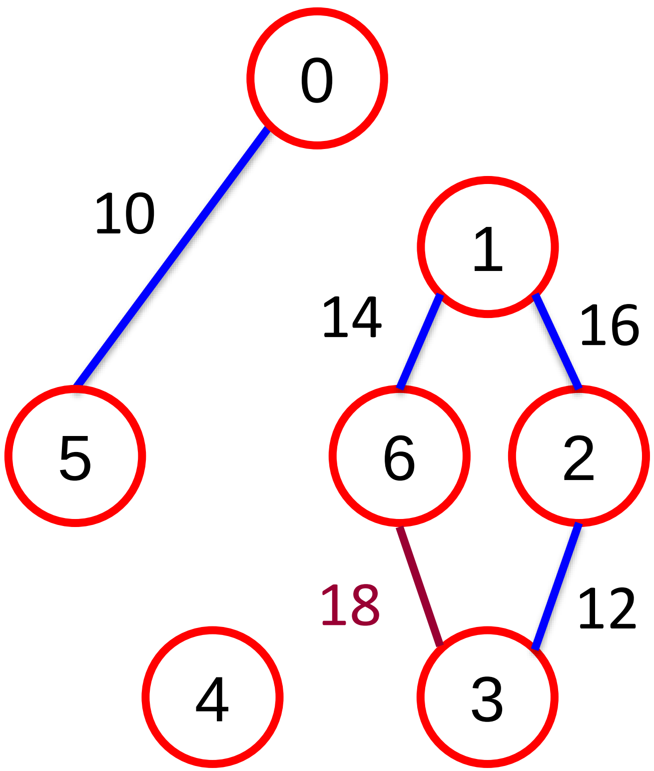
Spanning tree with cost 99

Disjoint sets

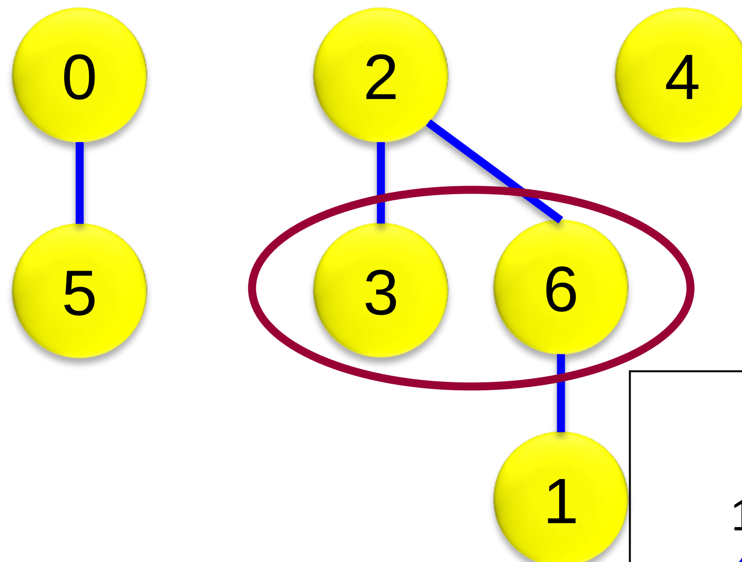


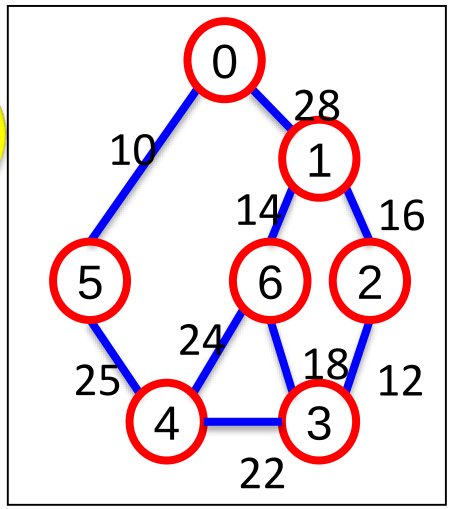


Running Example



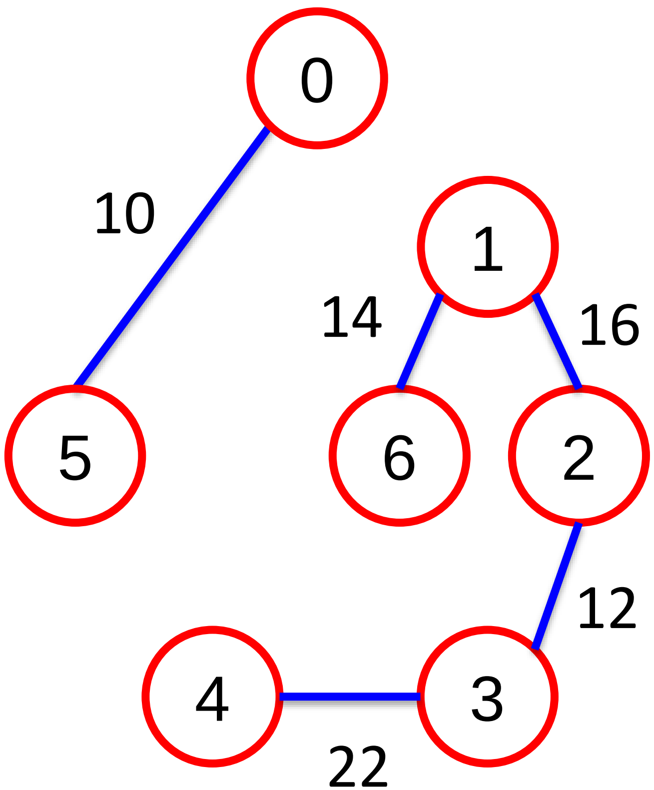
Disjoint sets





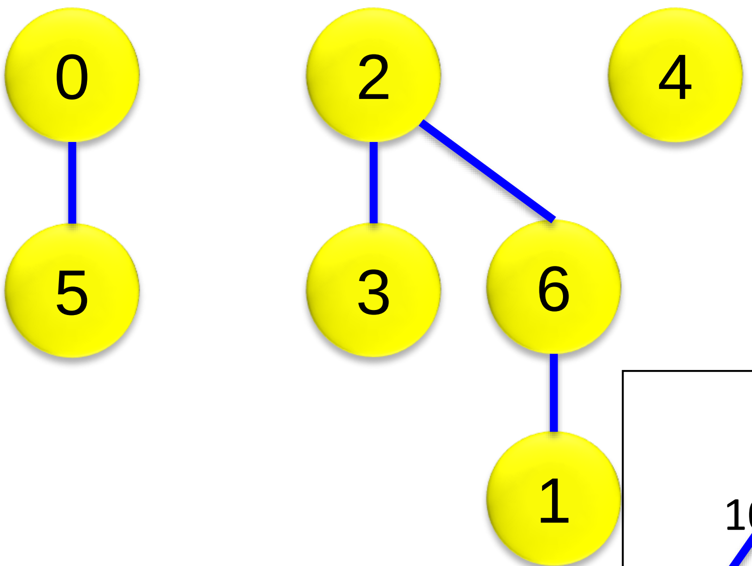
Spanning tree with cost 99

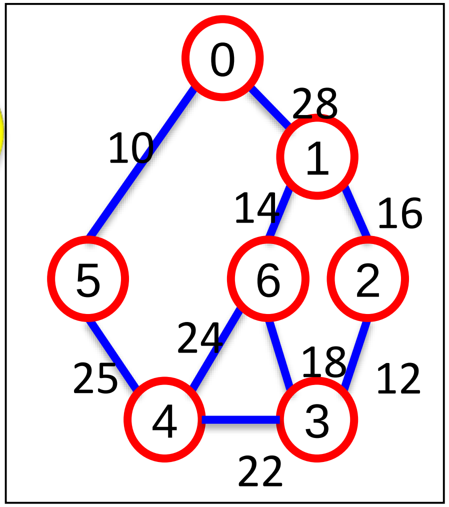
Running Example



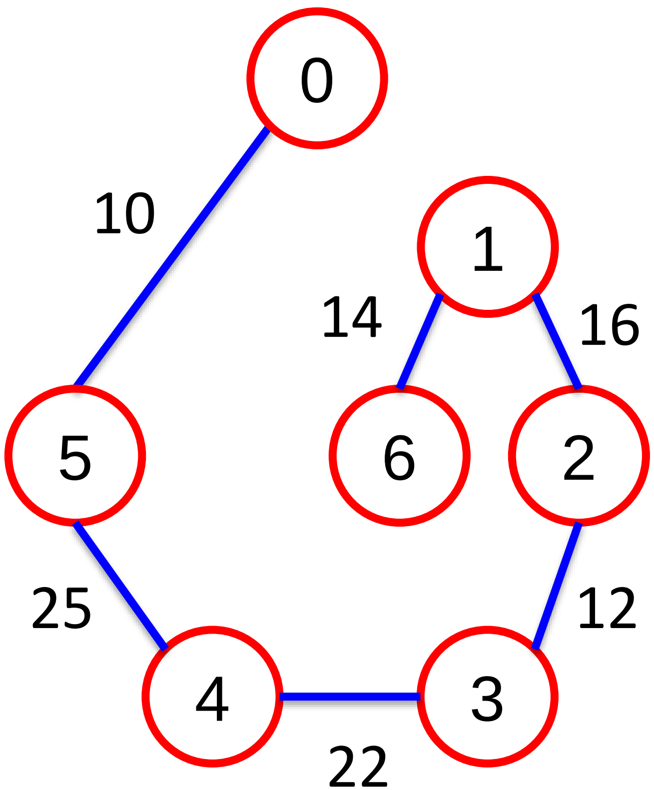
Spanning tree with cost 99

Disjoint sets



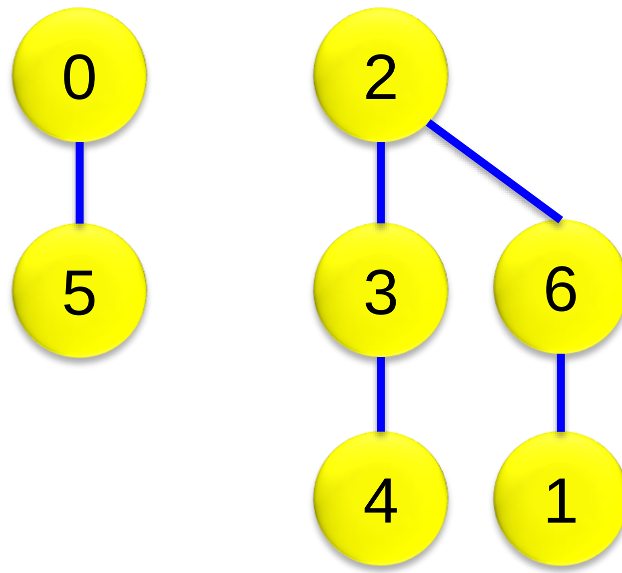


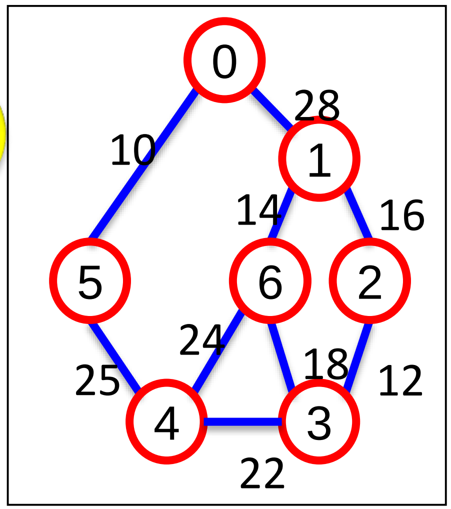
Running Example



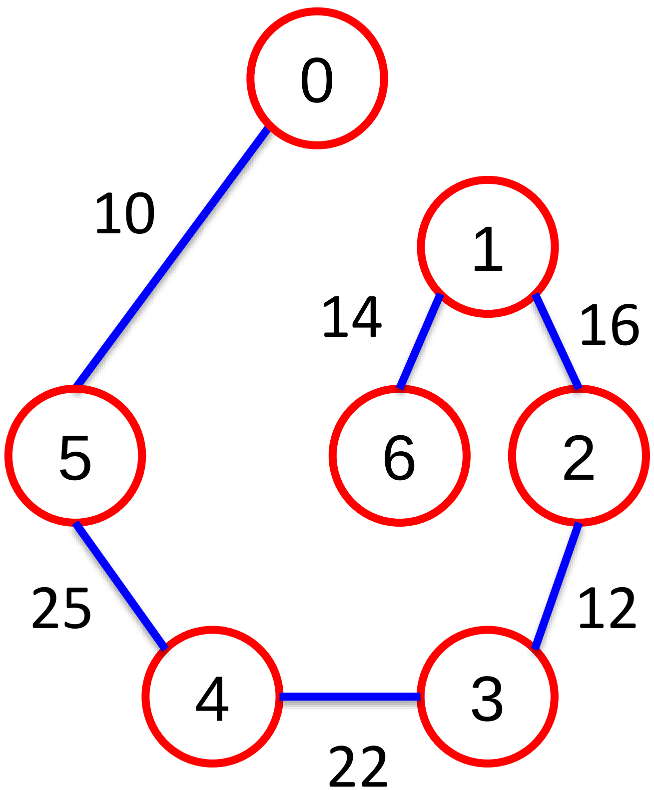
Spanning tree with cost 99

Disjoint sets

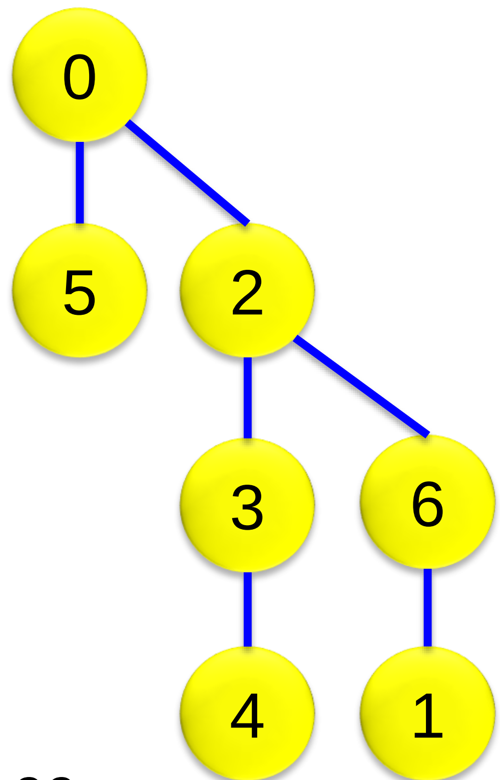


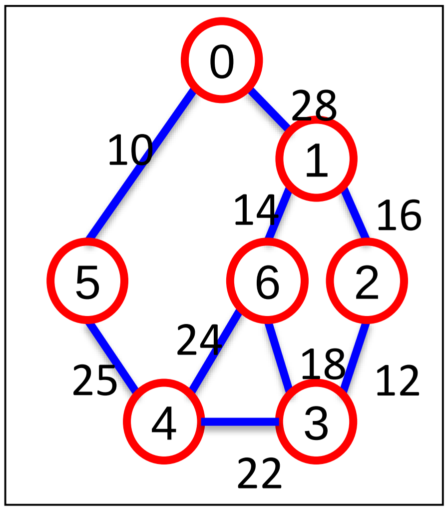


Running Example



Disjoint sets





Spanning tree with cost 99

Time Complexity

- Min heap: find minimum edge and delete from E
  - Steps 3 and 4: $O(\log e)$
- Set: find vertices in sets and union two sets
  - Step 5: $O(a(e))$
- At most execute e-1 rounds:
  - $(e-1)\times(\log e + a(e)) = O(e \log e)$

Optimality Proof of Kruskal's Algorithm

《Theorem 6.1》

Kruskal's algorithm can generate a minimum-cost spanning tree for any undirected connected graph G

<Proof>:

(a) Kruskal's method results in a spanning tree whenever a spanning tree exists

(b) The generated spanning tree is of minimum cost

Optimality Proof of Kruskal's Algorithm

<Proof>: (a)

- Only delete those edges that form a cycle
- Delete a cycle does not affect the connectivity of G
- Since G is initially a connected graph, Kruskal's algorithm always results in a connected graph with n-1 edges, i.e., the algorithm cannot terminate with $E=\emptyset$ and $|T|<n-1$
- Therefore, Kruskal's algorithm always creates a spanning tree for an undirected connected graph

Optimality Proof of Kruskal's Algorithm

<Proof>: (b)

- Let U be another minimum-cost spanning tree of G
- 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 the cost of U is not changed

Optimality Proof of Kruskal's Algorithm

<Proof>: (b)

- Transform U to T:
  - (1) Let e be the least-cost edge in T that is not in U
  - (2) When e is added to U, a cycle C is created
  - (3) Let f be any edge on C that is not in T
    (This edge must exist as T contains no cycle)
  - –Now **U' = U+{e}-{f}** is a spanning tree
  - –We need to prove that **cost(e) = cost(f)**

Optimality Proof of Kruskal's Algorithm

<Proof>: (b)

- 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 algorithm, but why f is not in T?

    → f together with certain edges in T, whose costs must be smaller than or equal to f, form a cycle
  - Those edges are also in U, and thus U must contain a cycle → Contradiction!
- Therefore cost(e)=cost(f)

Because e is the least-cost edge in T that is not in U

Outline

- Minimum cost spanning tree (Sec. 6.3)
  - Kruskal’s algorithm
  - Prims’s algorithm
  - Sollin’s algorithm
- Shortest path and transitive closure (Sec. 6.4)
  - Single source/all destination: non-negative edge costs
  - All-pairs shortest paths
  - Transitive closure
- Activity networks (Sec. 6.5)
  - Activity-on-vertex (AOV) networks

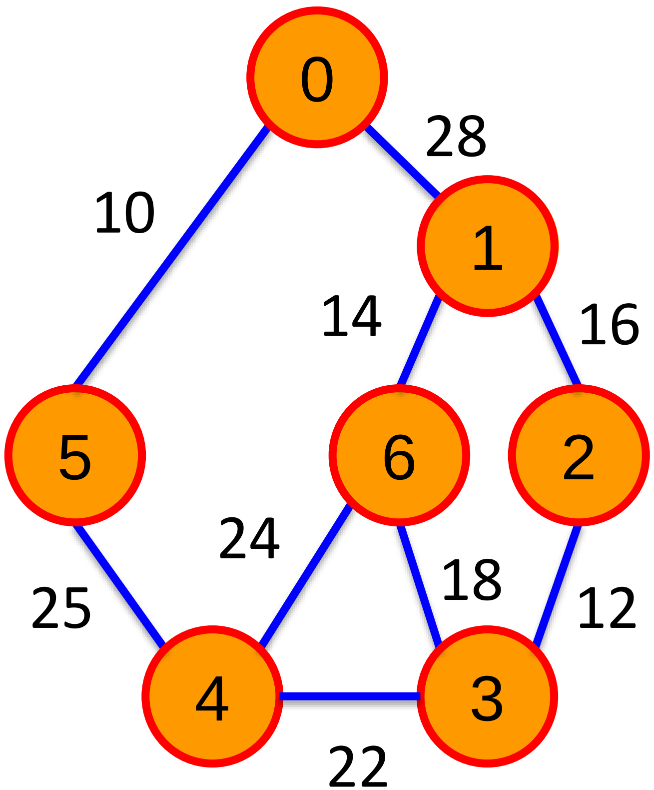
Prim's algorithm

Idea: add edges with minimum edge weight to the tree one at a time. **At all times during the algorithm, the set of selected edges forms a tree**

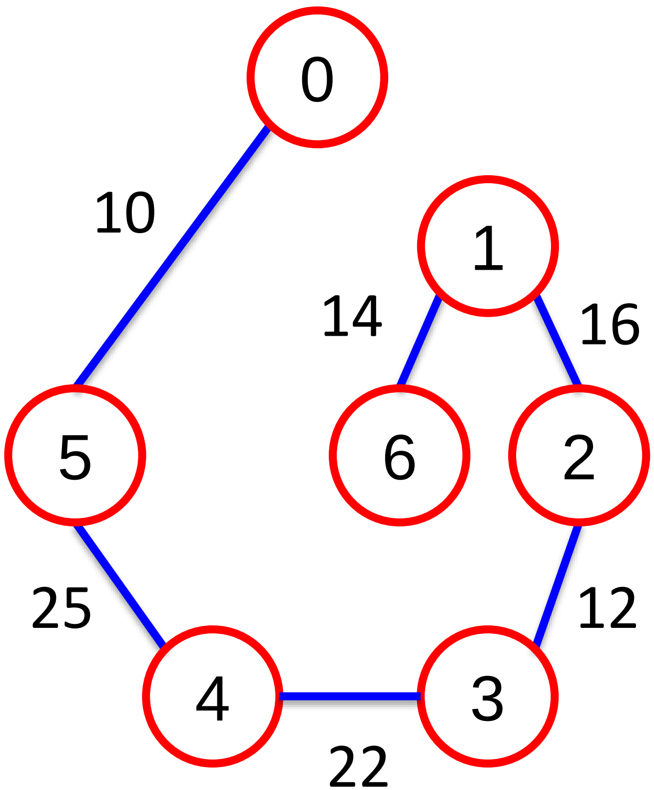
- Step 1: start with a tree T contains a single arbitrary vertex
- Step 2: among all edges, add a least cost edge (u,v) to T such that T U (u,v) is still a tree
- Step 3: repeat step 2 until T contains n-1 edges

Running Example

Refer to the textbook for detailed steps



Connected graph



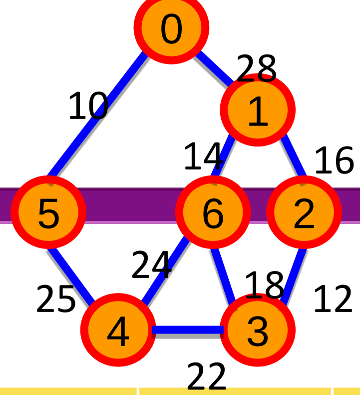
Spanning tree with cost 99

Prim’s Algorithm

```
1. V(T) = {0};  // start with vertex 0
2. for(T=ϕ; T has fewer than n-1 edges; add (u,v) to T){
3.   Let (u,v) be a least cost edge that u∈V(T), 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 << “no spanning tree!” << endl;
```

- Step 3: use an array to record nearest distance of each vertex to T
  - Only vertices not in V(T) & adjacent to T are updated, O(n)
- At most execute n rounds → $O(n^2)$

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}							

Outline

- Minimum cost spanning tree (Sec. 6.3)
  - Kruskal's algorithm
  - Prims's algorithm
  - Sollin's algorithm
- Shortest path and transitive closure (Sec. 6.4)
  - Single source/all destination: non-negative edge costs
  - All-pairs shortest paths
  - Transitive closure
- Activity networks (Sec. 6.5)
  - Activity-on-vertex (AOV) networks

Sollin's Algorithm

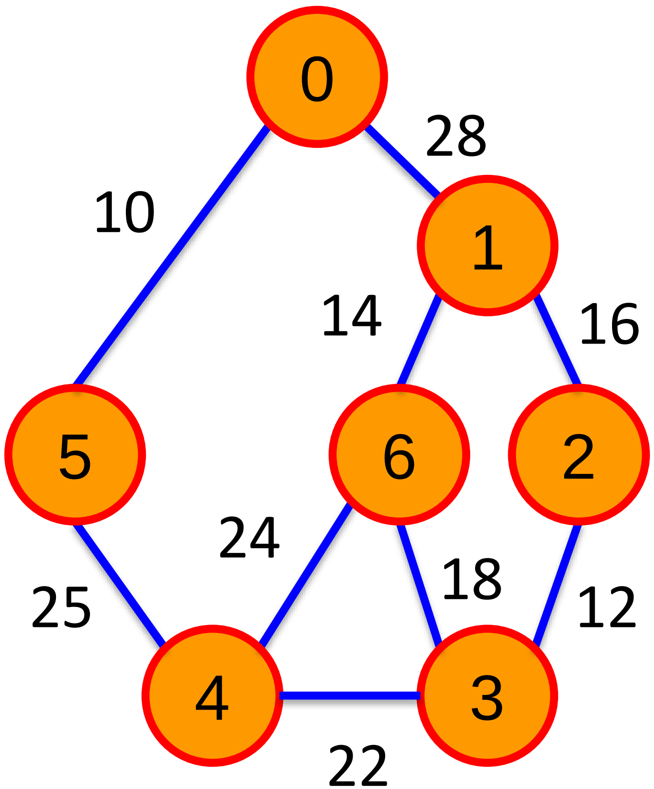
Idea: select several edges at each stage

- Step 1: start with a forest which has n spanning trees (each has one vertex)
- Step 2: select one minimum cost edge for each tree and 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

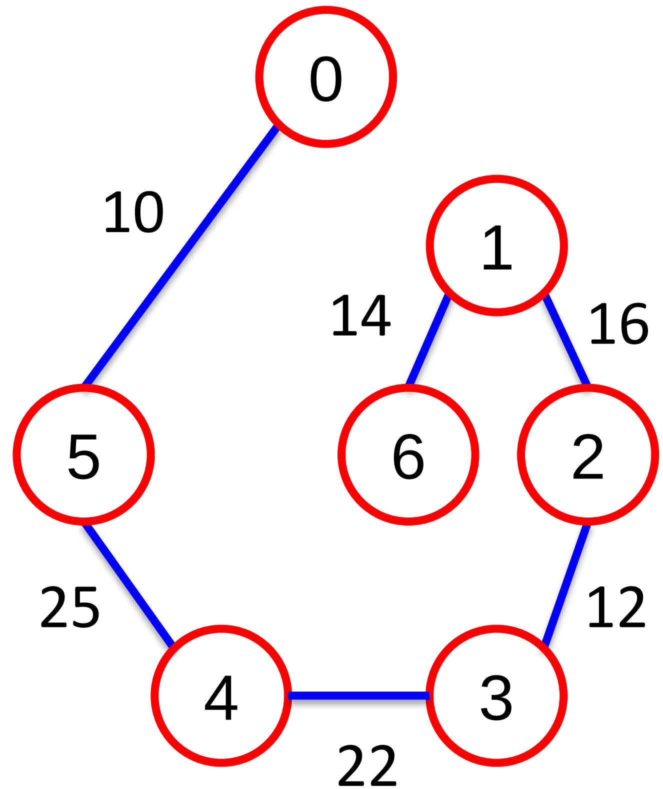
A parallel algorithm!

Running Example

Refer to the textbook for detailed steps



Connected graph



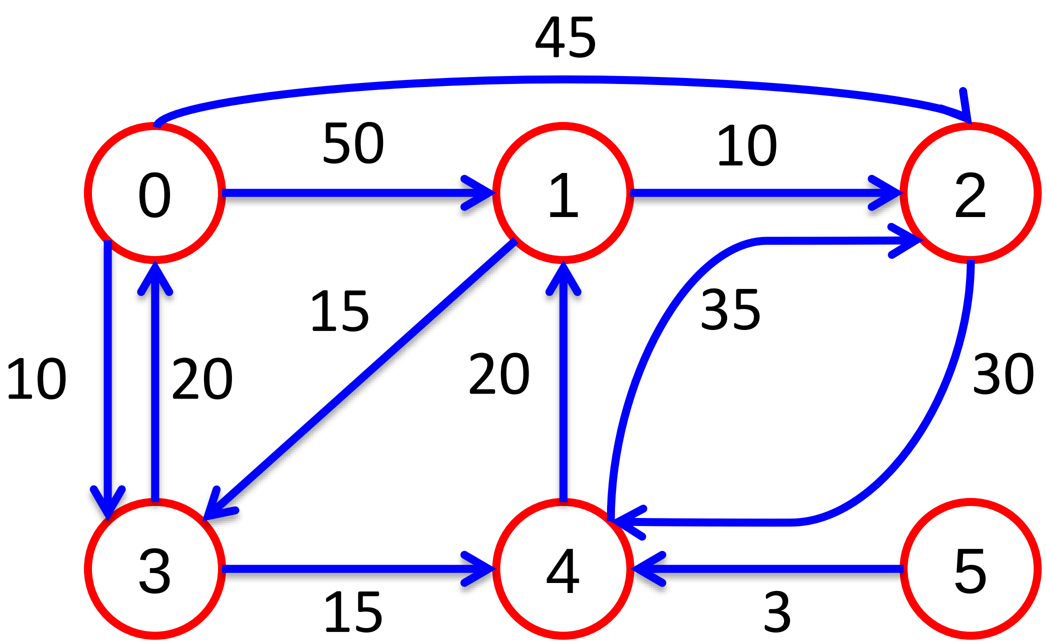
Spanning tree with cost 99

Outline

- Minimum cost spanning tree (Sec. 6.3)
  - Kruskal's algorithm
  - Prims's algorithm
  - Sollin's algorithm
- Shortest path and transitive closure (Sec. 6.4)
  - Single source/all destinations: non-negative edge costs
  - All-pairs shortest paths
  - Transitive closure
- Activity networks (Sec. 6.5)
  - Activity-on-vertex (AOV) networks

Single Source Shortest Paths

- Given a **digraph** with **nonnegative edge costs**, we want to compute a shortest path from a source vertex to each of the 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

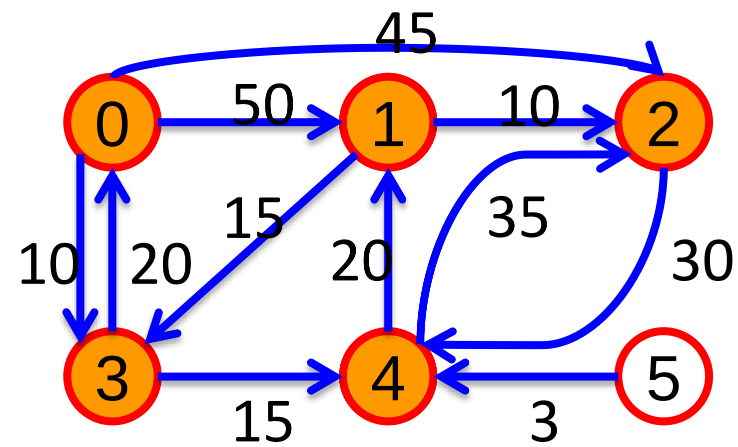
- Use a set **S** to store the vertices whose shortest path have been found
- An array **dist** stores the shortest distance found so far from source v to each of the other vertices
  - dist[w] = length of shortest path starting v, going through only vertices in S, and ending at w
- When a new vertex w is visited, update **dist**:

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

u is a vertex in S which is adjacent to w

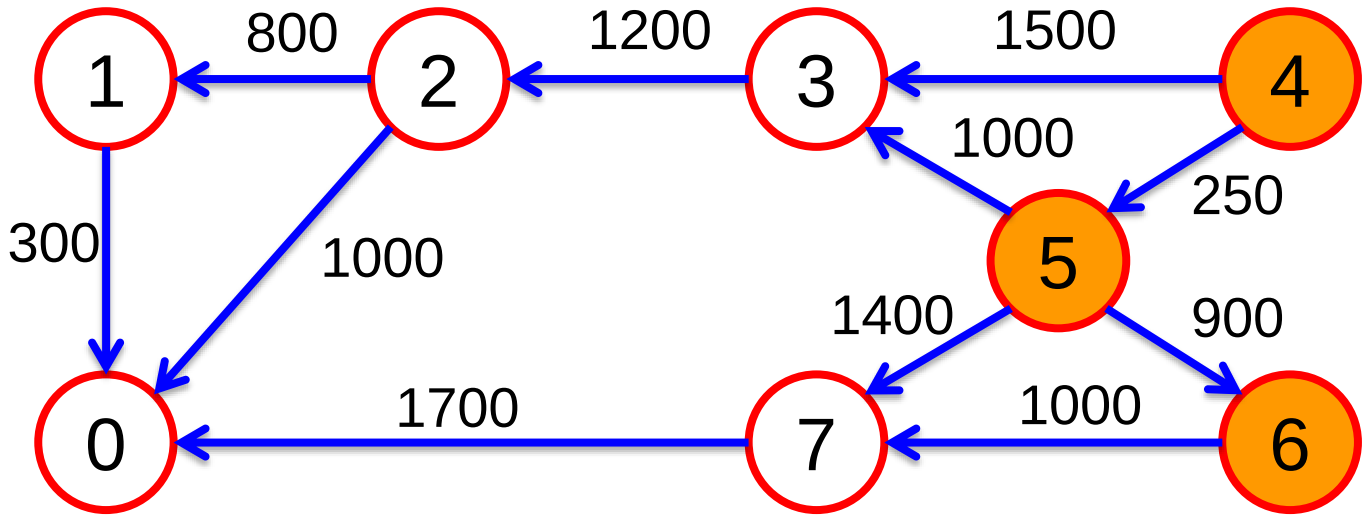
- Always select the vertex with smallest dist[w] into S

Running Example



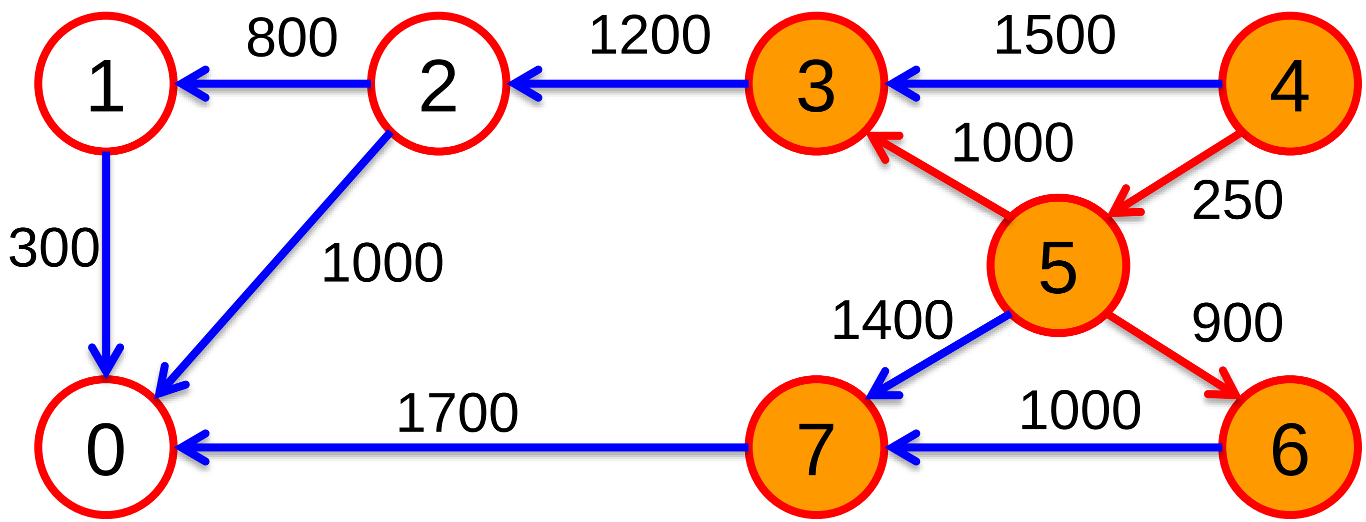
S	0	1	2	3	4	5
{0}	0	50	45	10	∞	∞
{0, 3}	0	50	45	10	25	∞
{0, 3, 4}	0	45	45	10	25	∞
{0, 3, 4, 1}	0	45	45	10	25	∞
{0, 3, 4, 1, 2}	0	45	45	10	25	∞

Running Example



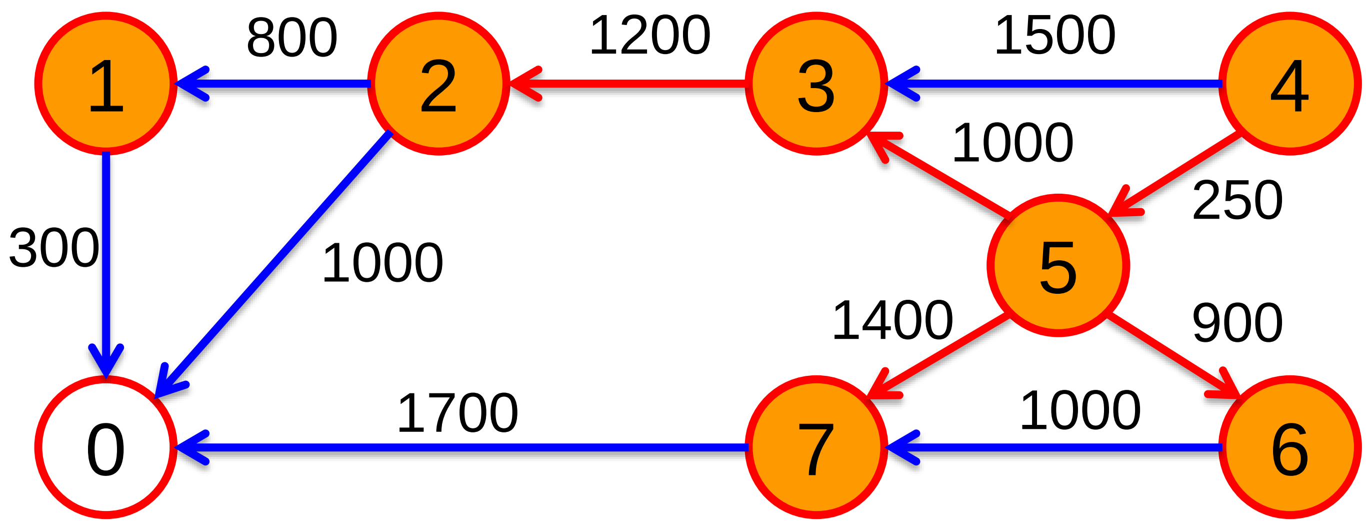
S	0	1	2	3	4	5	6	7
{4}	∞	∞	∞	1500	0	250	∞	∞
{4, 5}	∞	∞	∞	1250	0	250	1150	1650
{4, 5, 6}	∞	∞	∞	1250	0	250	1150	1650

Running Example



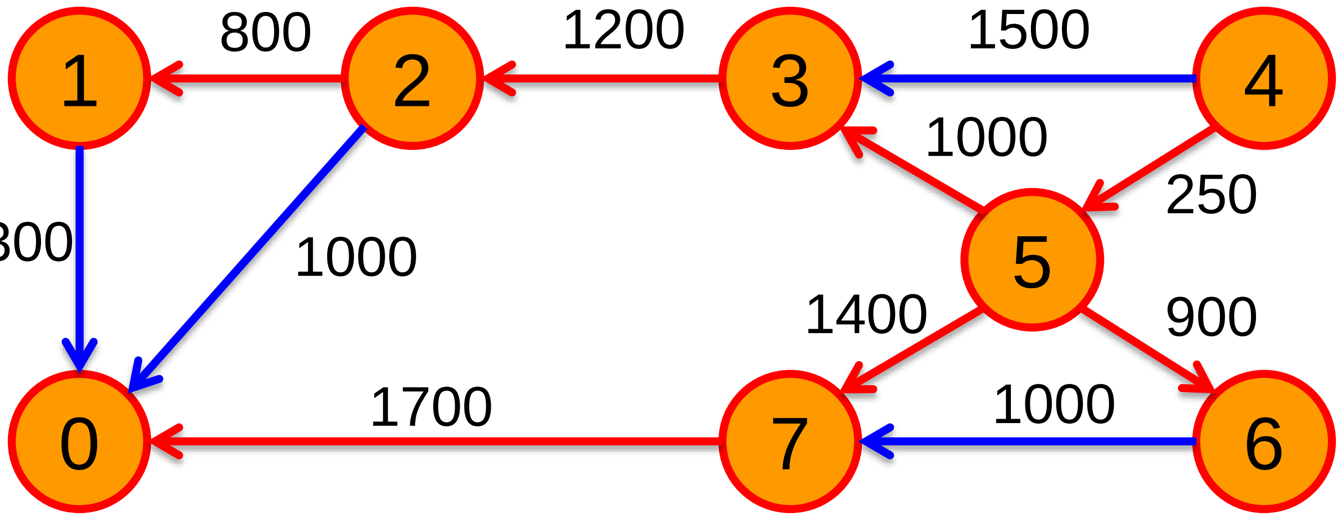
S	0	1	2	3	4	5	6	7
{4, 5, 6}	∞	∞	∞	1250	0	250	1150	1650
{4, 5, 6, 3}	∞	∞	2450	1250	0	250	1150	1650
{4, 5, 6, 3, 7}	3350	∞	2450	1250	0	250	115	1650

Running Example



S	0	1	2	3	4	5	6	7
{4, 5, 6, 3, 7}	3350	∞	2450	1250	0	250	1150	1650
{4, 5, 6, 3, 7, 2}	3350	3250	2450	1250	0	250	1150	1650
{4, 5, 6, 3, 7, 2, 1}	3350	3250	2450	1250	0	250	1150	1650

Running Example



S	0	1	2	3	4	5	6	7
{4,5,6,3,7,2,1}	3350	3250	2450	1250	0	250	1150	1650
{4,5,6,3,7,2,1,0}	3350	3250	2450	1250	0	250	1150	1650

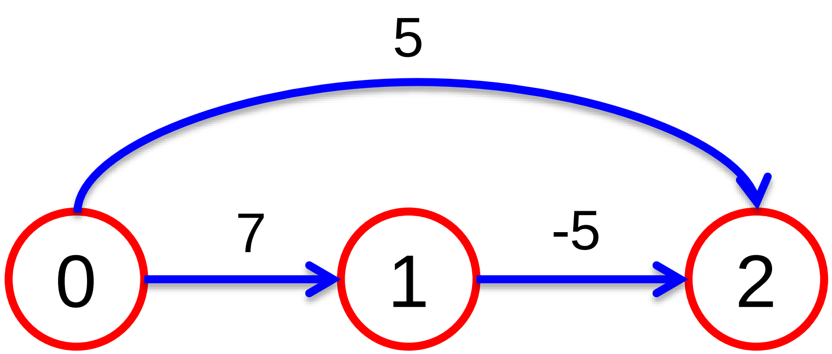
Dijkstra's Algorithm

```
void MatrixWDigraph::ShortestPath(const int n, const int v)
{ // dist[j], 0 ≤ j < n, stores 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];}
  s[v] = true;
  dist[v] = 0;
  // find n - 1 paths starting from v
  for(int i=0; i<n-2; i++){                         O(n)
  // Choose() returns u that dist[u] min. & 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];
   } // end of for (i = 0; ...)
}
```

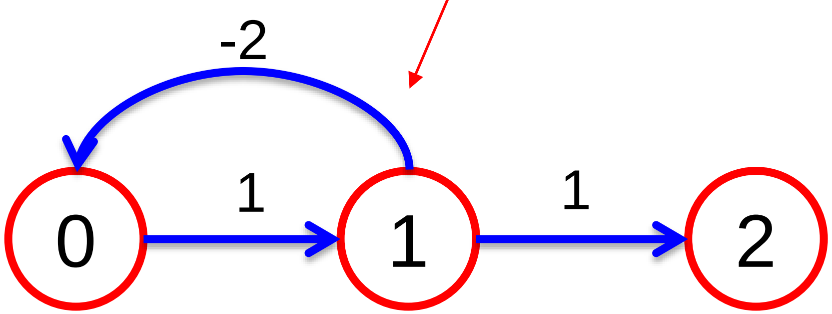
Time complexity: $O(n^2)$

Digraph with Negative Costs

- A similar algorithm can be applied to **digraph with negative cost edges** (*Bellman and Ford Algorithm*)
- However, the digraph **MUST NOT** contain cycles of negative length, e.g., shortest path from 0 to 2 is -∞



Digraph with a negative cost edge
(Dijkstra's Algorithm won't work)



Digraph with a cycle of negative cost

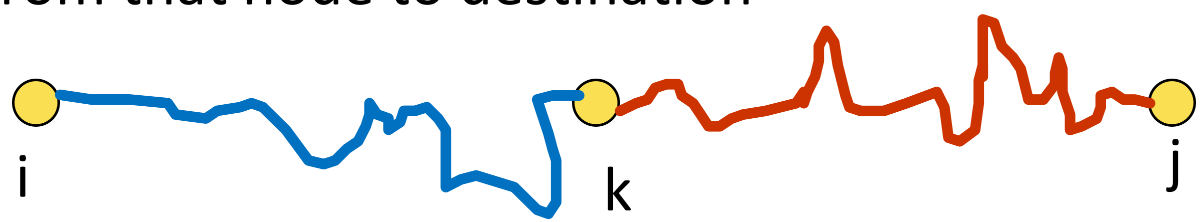
Outline

- Minimum cost spanning tree (Sec. 6.3)
  - Kruskal's algorithm
  - Prims's algorithm
  - Sollin's algorithm
- Shortest path and transitive closure (Sec. 6.4)
  - Single source/all destination: non-negative edge costs
  - All-pairs shortest paths
  - Transitive closure
- Activity networks (Sec. 6.5)
  - Activity-on-vertex (AOV) networks

All-Pairs Shortest Paths

- Intuitive idea: apply single source shortest path to each of n vertices → $O(n^3)$
- Alternative: use an idea similar to induction
  - Suppose we have found all-pairs shortest paths using only a set of k-1 vertices as the intermediate vertices
  - By adding one more vertex into this set, can we further reduce all-pairs shortest paths?

    → only need to consider paths from source to that node and from that node to destination



Floyd-Warshall's Algorithm

- Assumption: G has no cycles with negative length

  → Any shortest path must have at most n-1 edges

- Represent G using a length-adjacency matrix A:
  - $A^{-1}[i][j]$: just length[i][j]
  - $A^{n-1}[i][j]$: the length of the shortest path from i to j in G
  - $A^{k}[i][j]$: the length of the shortest path from i to j going through no intermediate vertex of index greater than k

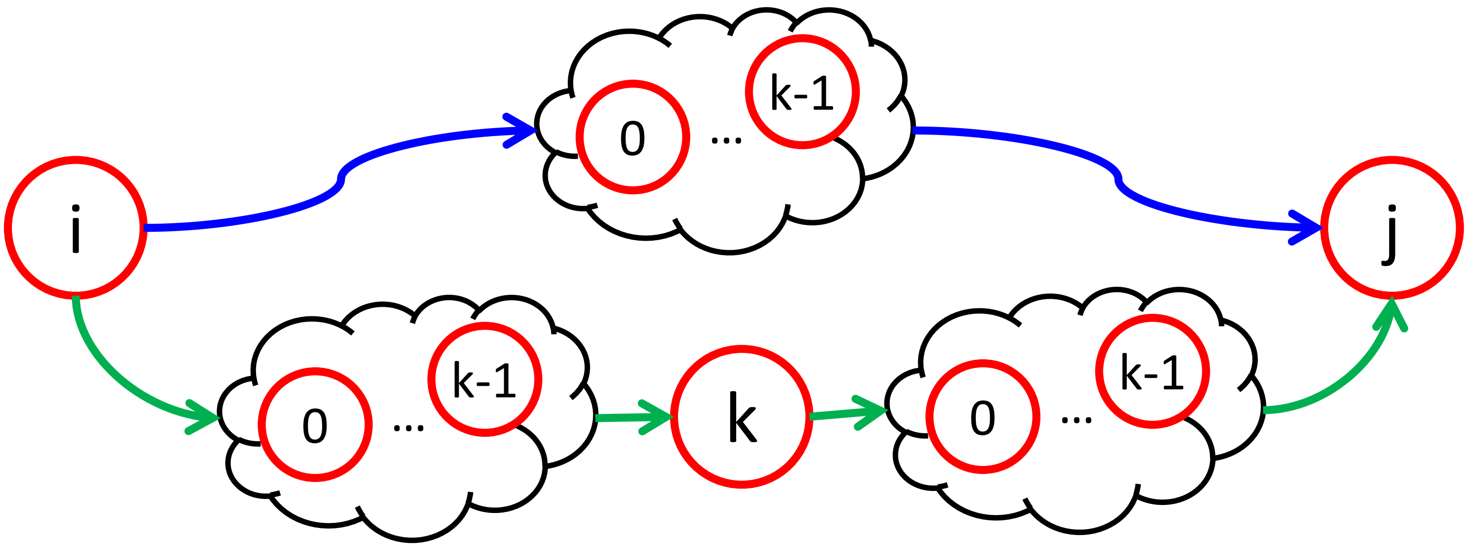
    → i.e., use only a set of k vertices as intermediate vertices

Run at most n-1 rounds

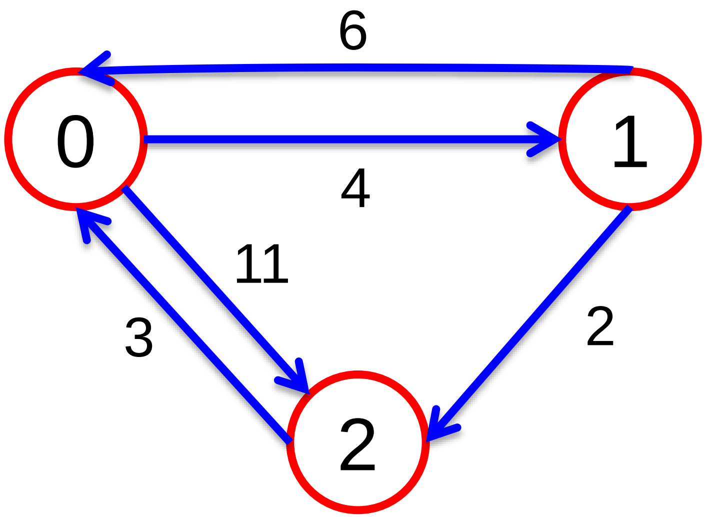
Floyd-Warshall's Algorithm

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

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



Running Example



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

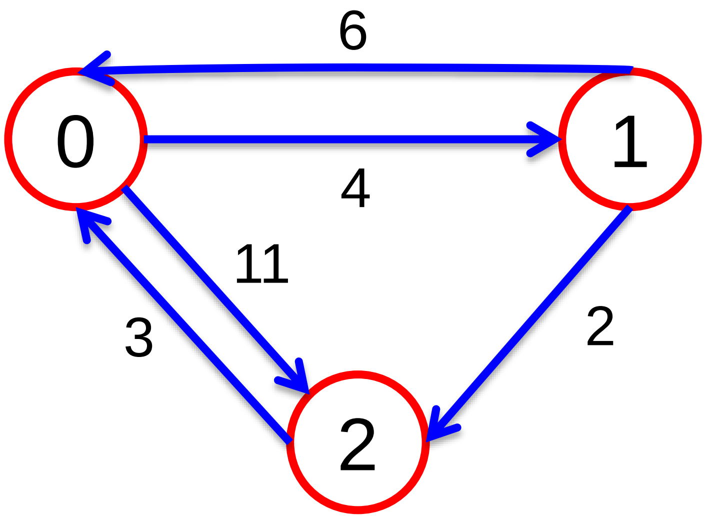
$A^0[2][1] = \min(A^{-1}[2][1], A^{-1}[2][0]+A^{-1}[0][1])$

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

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

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

Running Example



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

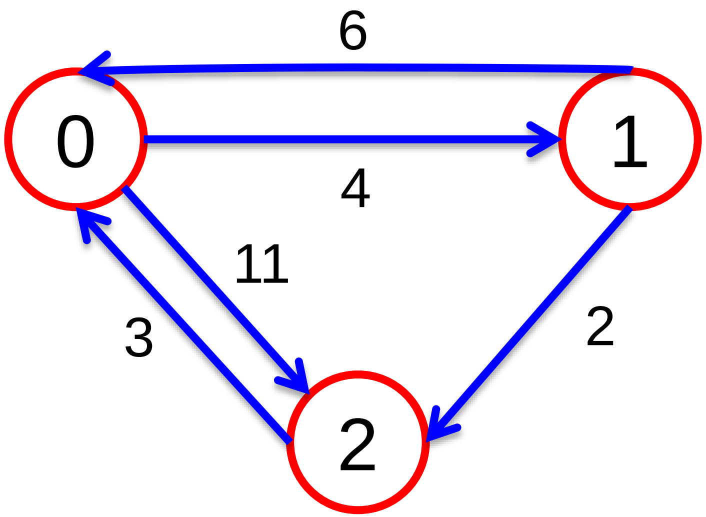
$A^1[2][0] = \min(A^0[2][0], A^0[2][1]+A^0[1][0])$

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

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

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

Running Example

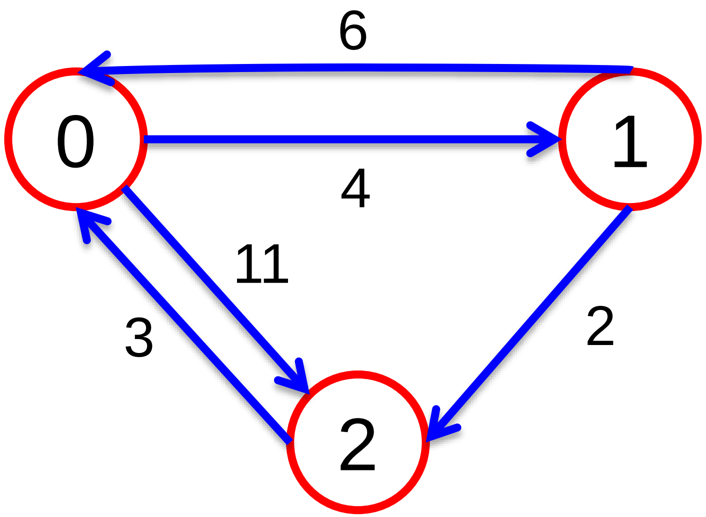


A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

$A^2[0][1] = \min(A^1[0][1], A^1[0][2]+A^1[2][1])$
$A^2[0][1] = \min(4, 6+3) = 4$

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

Running Example



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

Floyd-Warshall's Algorithm

```
void MatrixWDigraph::AllLengths(const int n)
{// length[i][j]: edge length between i and j
 // a[i][j]: 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];
 }
```

Time complexity: $O(n^3)$

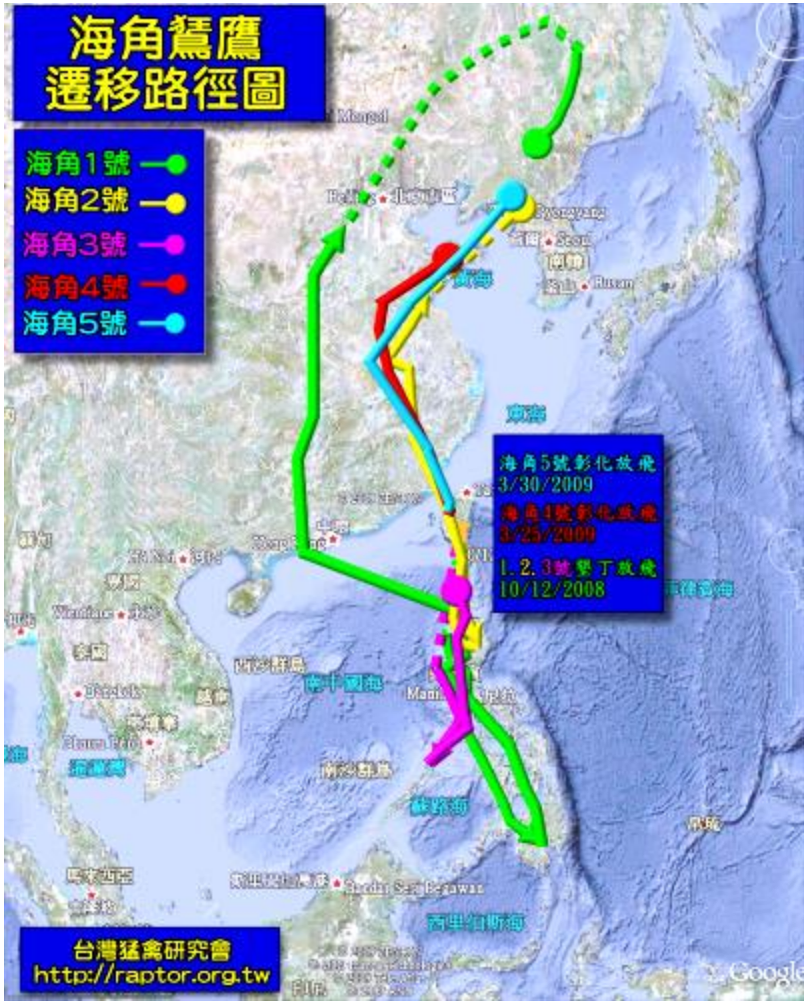
Outline

- Minimum cost spanning tree (Sec. 6.3)
  - Kruskal's algorithm
  - Prims's algorithm
  - Sollin's algorithm
- Shortest path and transitive closure (Sec. 6.4)
  - Single source/all destination: non-negative edge costs
  - All-pairs shortest paths
  - Transitive closure
- Activity networks (Sec. 6.5)
  - Activity-on-vertex (AOV) networks

Migration of Gray-faced Buzzard Eagle

- Gray-faced Buzzard Eagle (灰面鵟鷹)

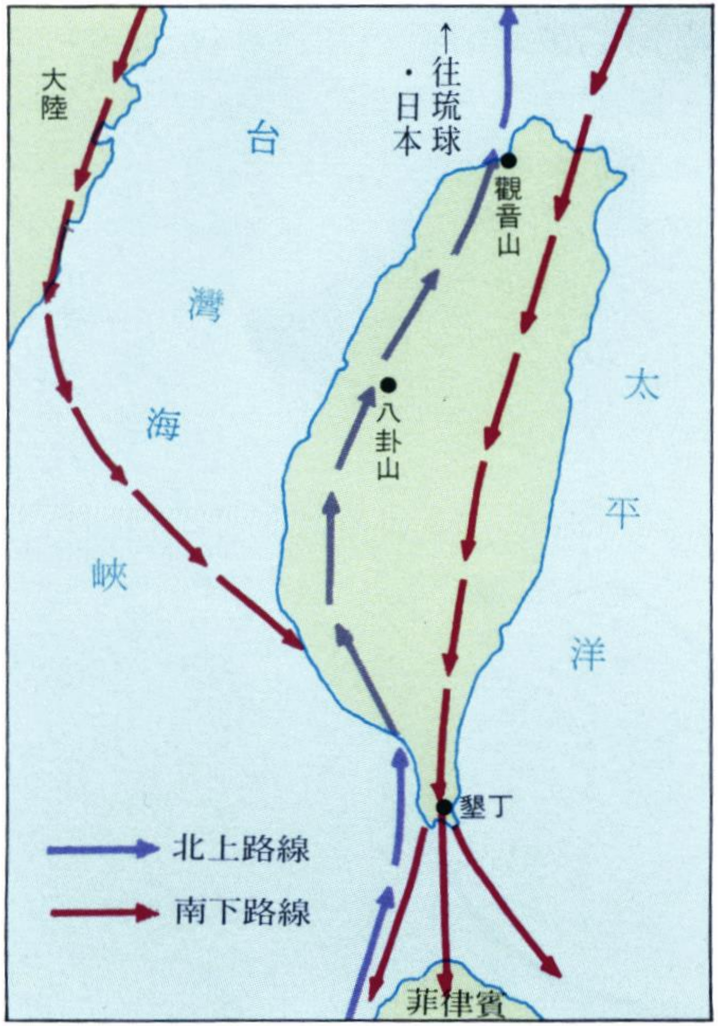
(http://www.ktnp.gov.tw/cht/notes02.aspx?print=1&ecologyContentID=20)



(http://raptor.org.tw/grey-faced-buzzard-satellite-tracking/origin.html)

Migration of Gray-faced Buzzard Eagle

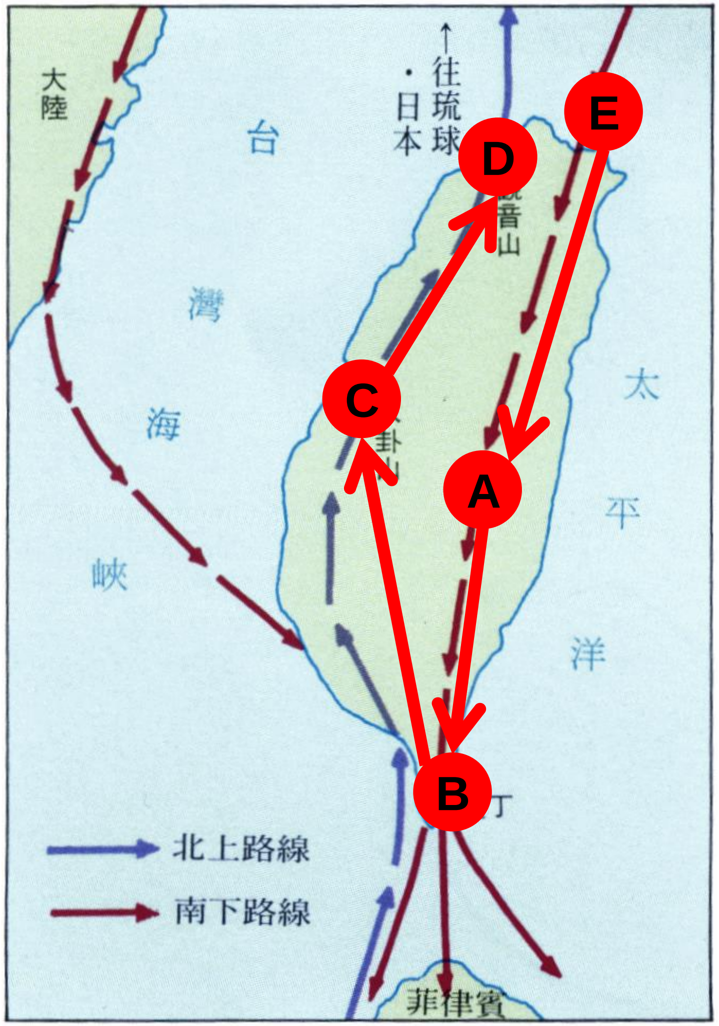
- Migrating routes thru Taiwan



(https://tw.knowledge.yahoo.com/question/question?qid=1612050905363)

Migration of Gray-faced Buzzard Eagle

- Resting sites (assumed)
- Let *x R y* denote “eagles flight directly from site *x* to *y*“
- If *x R y* and *y R z*, can we imply *x R z*?
- The relation *R* over the set of sites *S* is *non-transitive*



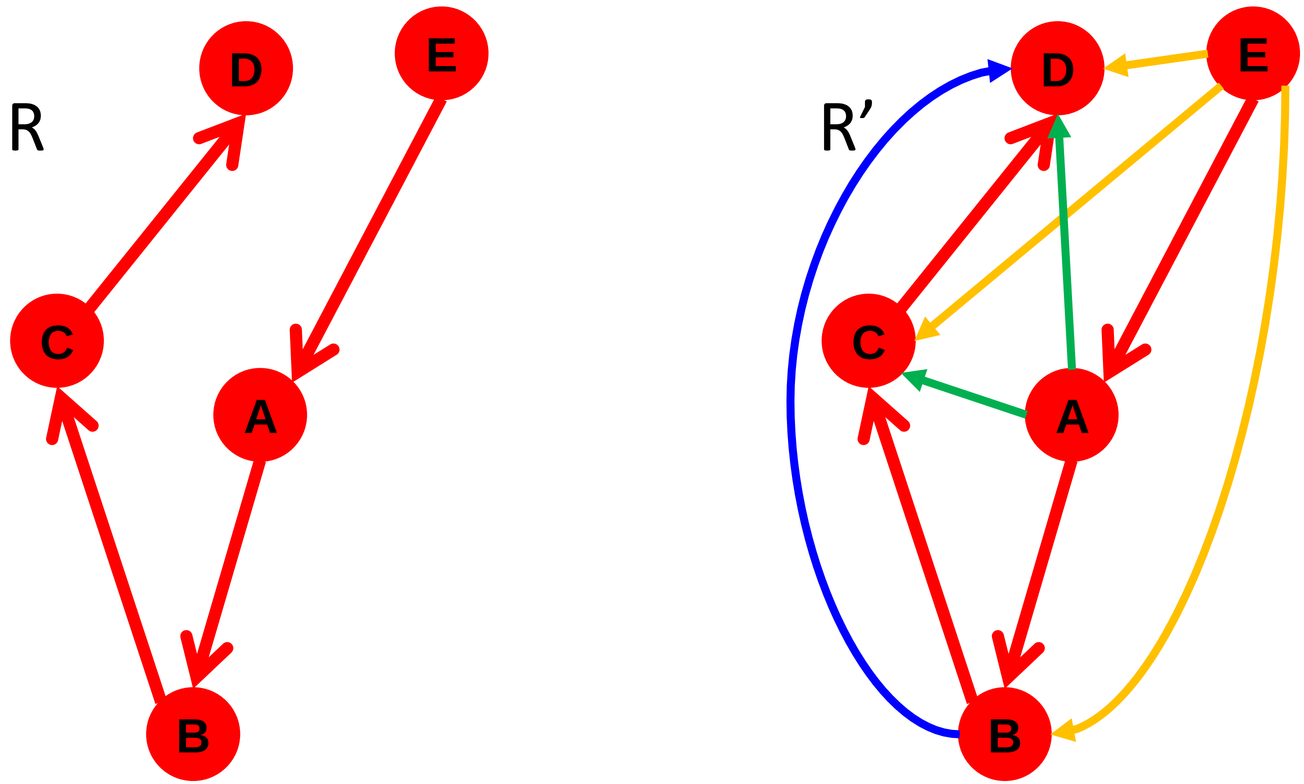
(https://tw.knowledge.yahoo.com/question/question?qid=1612050905363)

Transitive Relations

- A relation *R* on a set *S* is *transitive* if, for all *x*, *y*, and *z* in *S*, whenever *x R y* and *y R z* then *x R z*. (Definition in page 376 of textbook)
  - Example: equality, arrive-before, is-ancestor-of, …
  - Example of non-transitive relation: flight directly from site x to y, is-parent-of, …
- Can we extend a non-transitive relation into a transitive relation?
  - Example: from is-parent-of to is-ancestor-of?
  - Can you give an extended relation R' for the relation R = "eagles flight directly from x to y" that is transitive?

Extended Relations for Transitivity

- An example of an extended relation R'
  "eagles starting at site x may rest at site y"



Extended Relations for Transitivity

- It is always possible to extend a relation R to derive another relation R' that contains R and is transitive
- In fact, there are many such extended relations
- Among all such extended relations, the smallest one is called the *transitive closure* of R
  - May help to answer questions such as *reachability* of a statements in a program

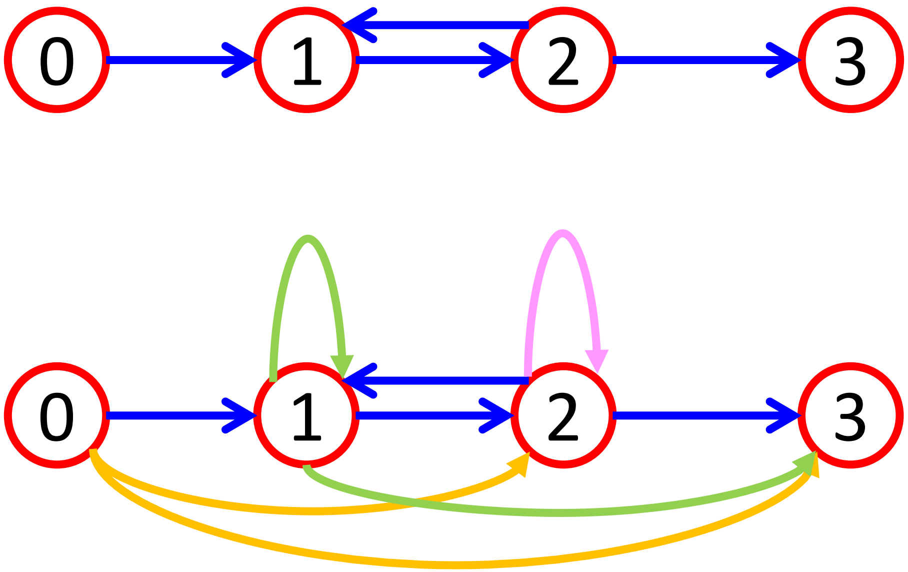
Transitive Closure

For a graph G with unweighted edges:

- 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
  - A^+[i][i] = 1 iff there is a cycle of length > 1 containing i
- 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
  - A relation *R* on a set *S* is *reflexive* if, for every *x* in *S*, *x R x* is true

Transitive Closure

- Use Floyd-Warshall's algorithm to get A^+
  - $A^k[i][j] = A^{k-1}[i][j] \,||\, (A^{k-1}[i][k] \,\&\&\, A^{k-1}[k][j]);$

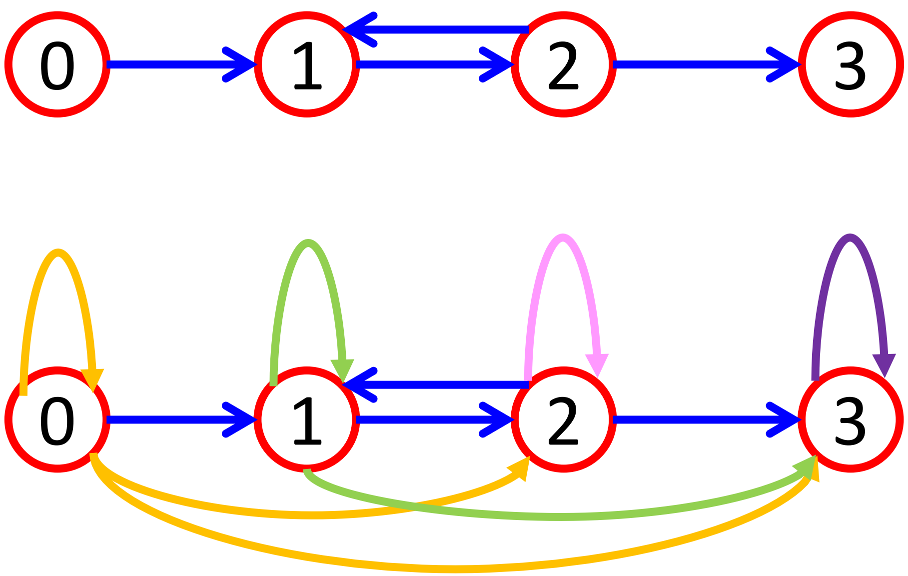


A^+	0	1	2	3
0	0	1	1	1
1	0	1	1	1
2	0	1	1	1
3	0	0	0	0

Transitive closure matrix

Reflexive Transitive Closure

- $A^+[i][i] \leftarrow 1$ for all i in A^+



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

Outline

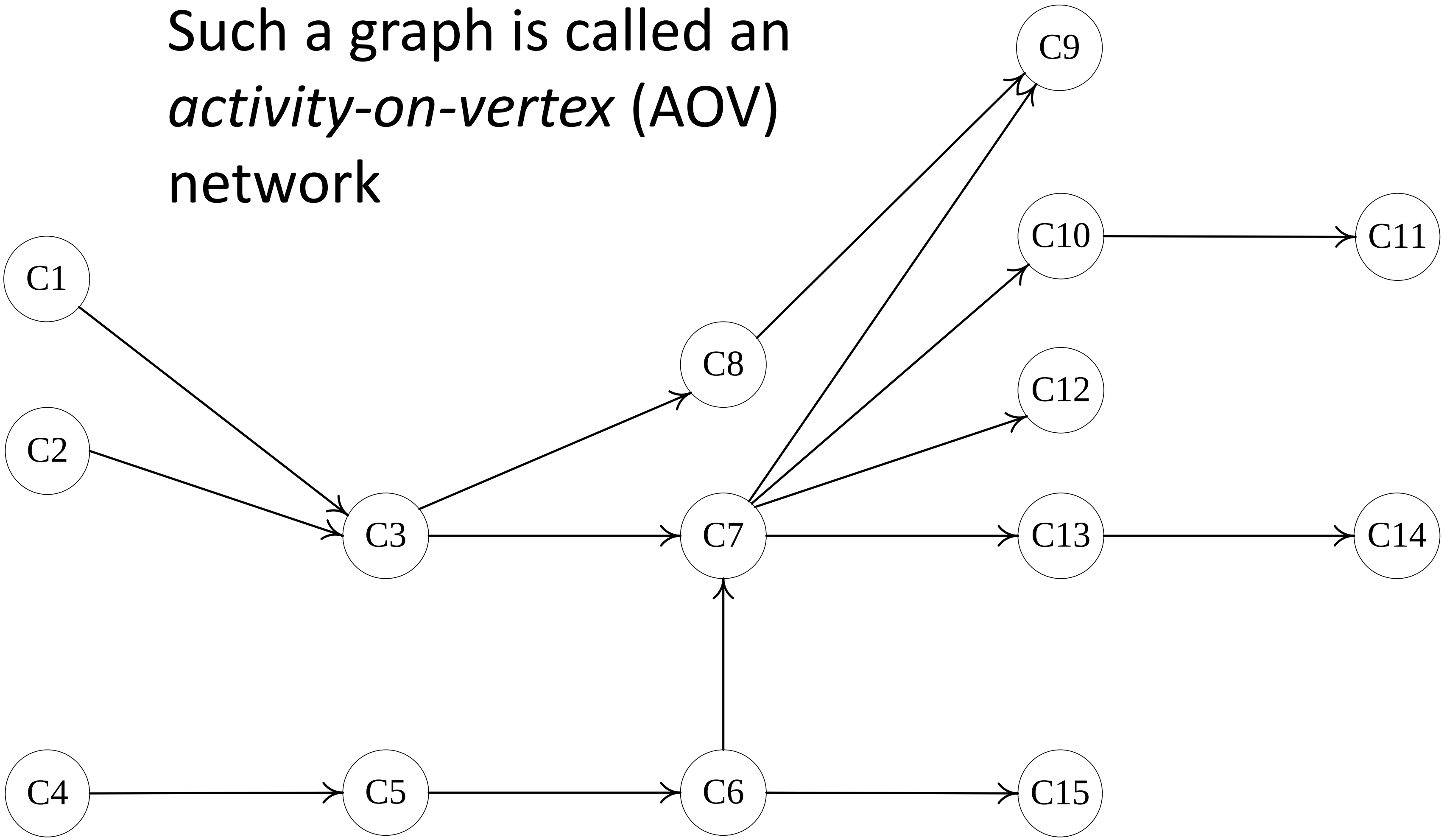
- Minimum cost spanning tree (Sec. 6.3)
  - Kruskal's algorithm
  - Prims's algorithm
  - Sollin's algorithm
- Shortest path and transitive closure (Sec. 6.4)
  - Single source/all destination: non-negative edge costs
  - All-pairs shortest paths
  - Transitive closure
- Activity networks (Sec. 6.5)
  - Activity-on-vertex (AOV) networks

Courses and Their Prerequisites

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

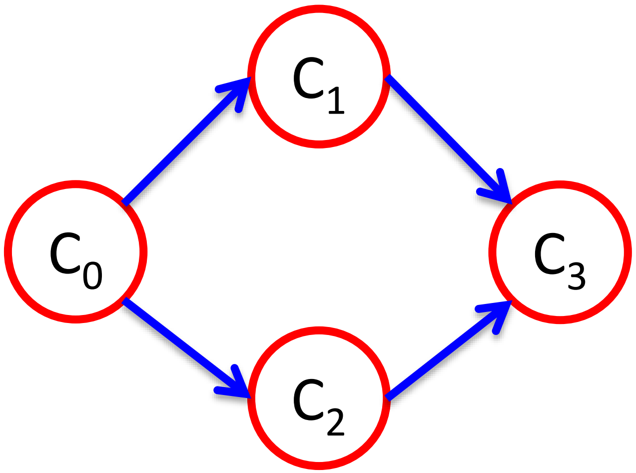
Prerequisite Relationship as a Graph

Such a graph is called an *activity-on-vertex* (AOV) network



Activity-on-Vertex (AOV) Networks

- A digraph G with the vertices representing tasks or activities and the edges representing 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

Precedence relation is transitive?

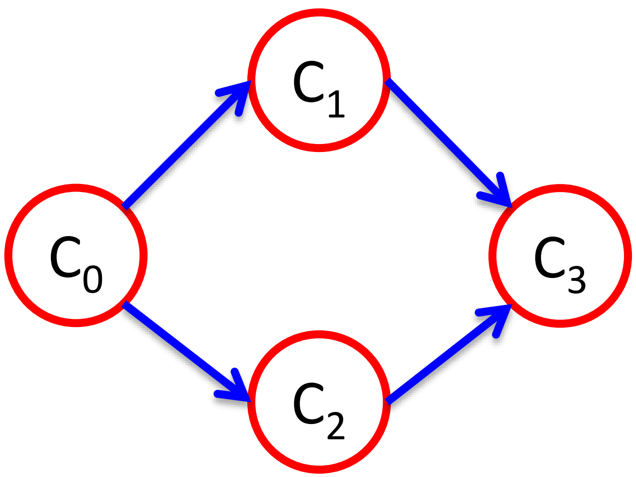
reflexive?

(Definition in page 376 of textbook)

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 graph, then i precedes j in the linear ordering

    → from *partial ordering* to *total ordering*



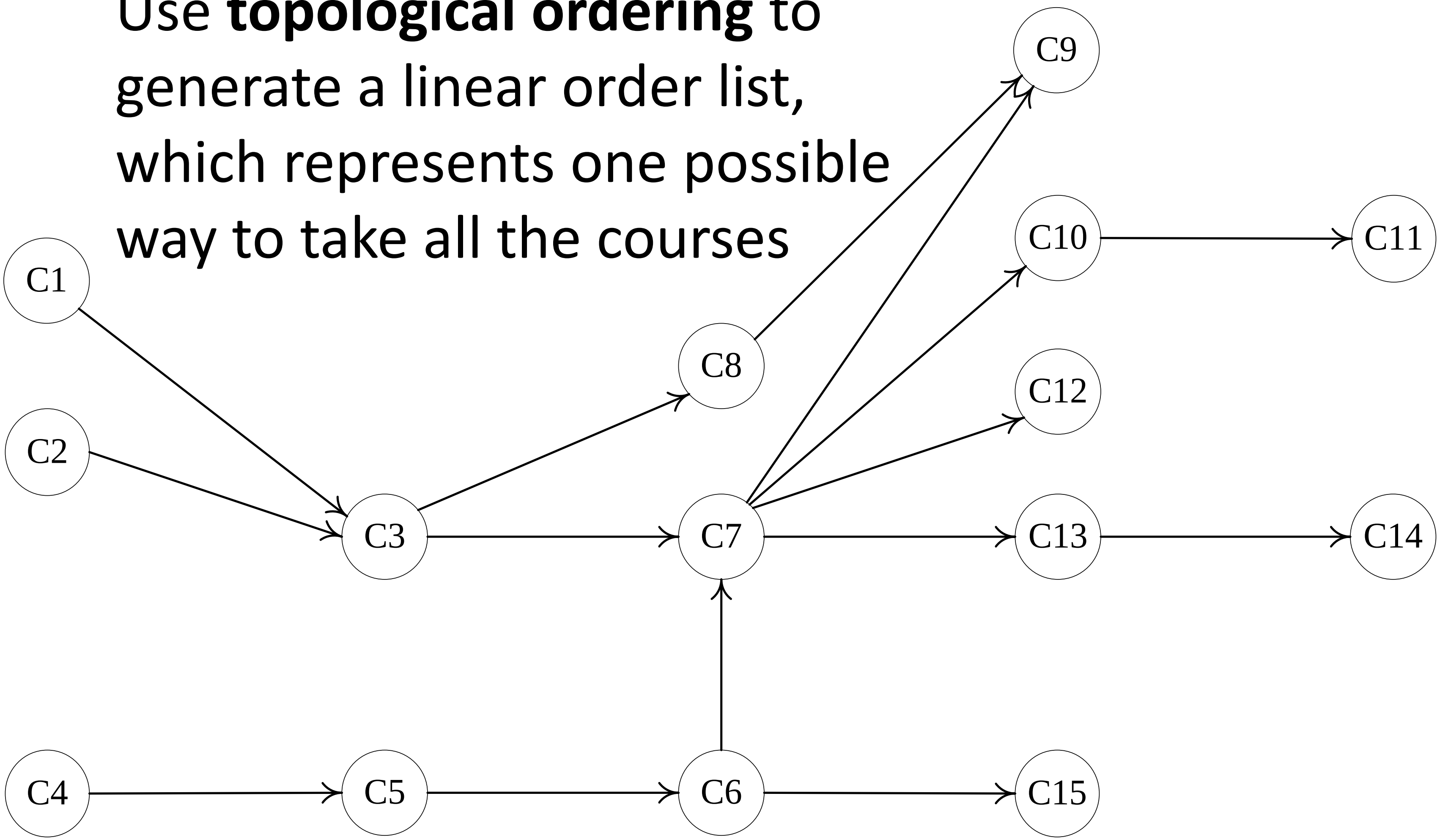
$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow C_3$ (√)

$C_0 \rightarrow C_2 \rightarrow C_1 \rightarrow C_3$ (√)

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

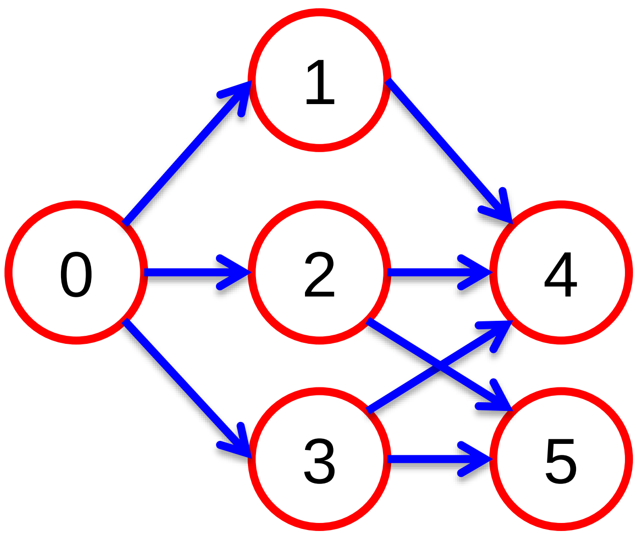
AOV Network of Courses

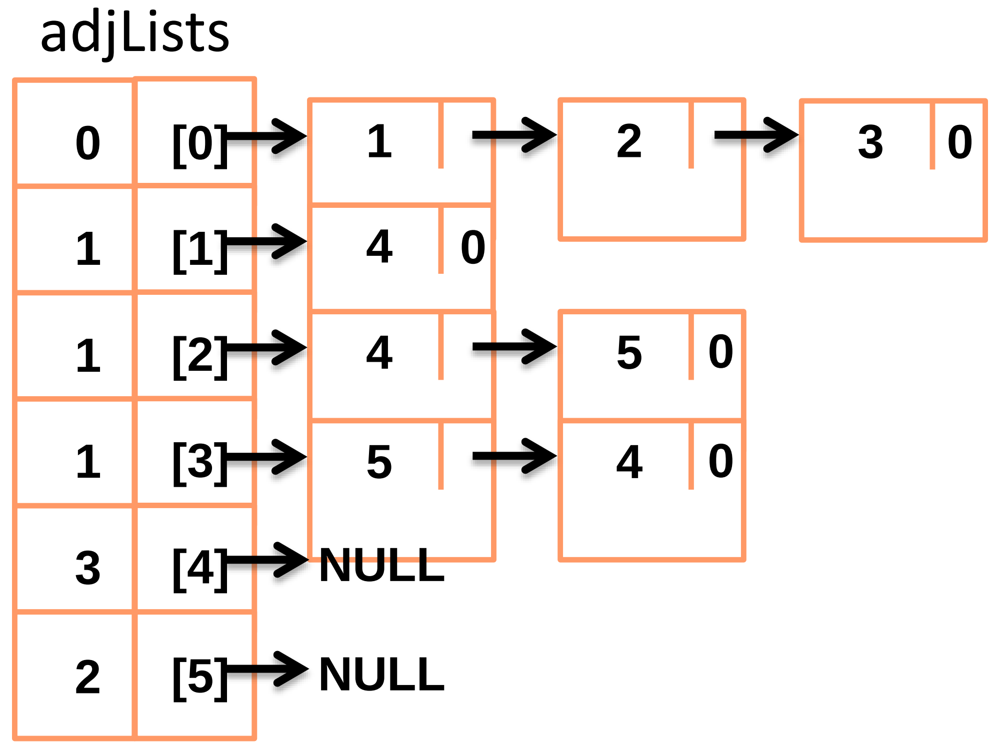
Use **topological ordering** to generate a linear order list, which represents one possible way to take all the courses



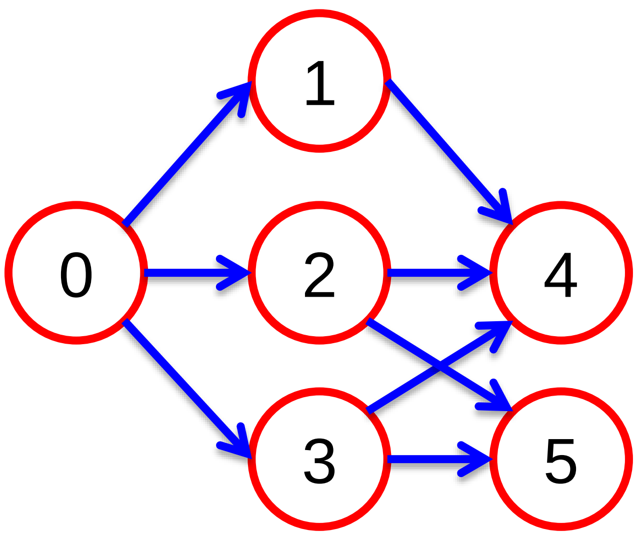
Topological Ordering

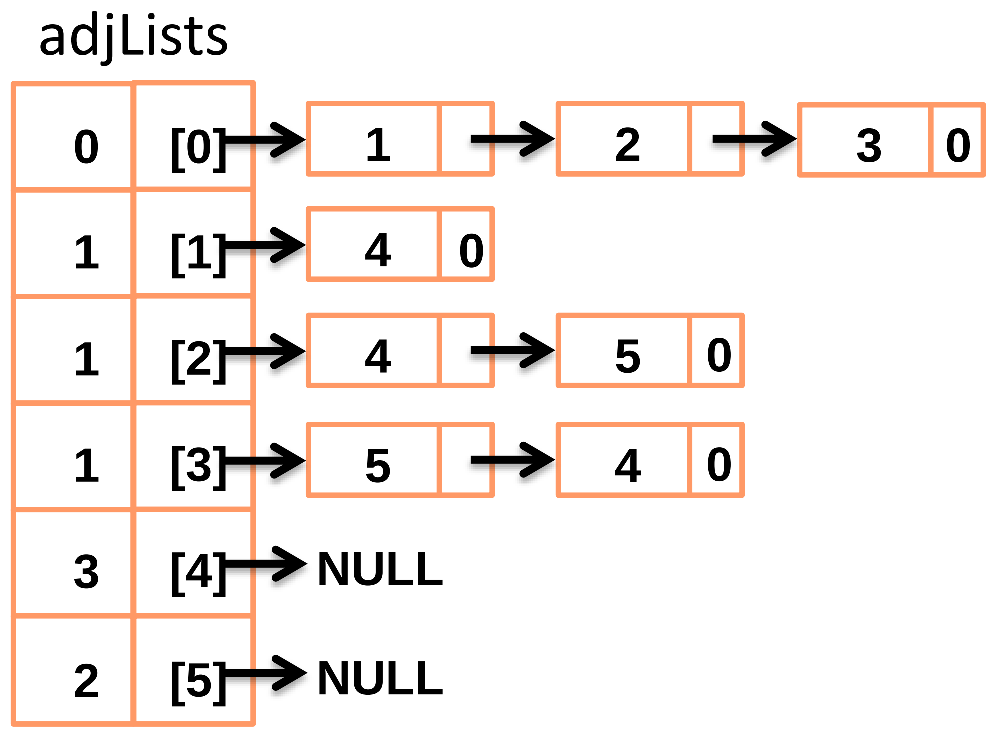
- Iteratively pick a vertex v that has no predecessor
  - Use a “count” field to record “in-degree” of each vertex
- Remove that vertex with all out-edges





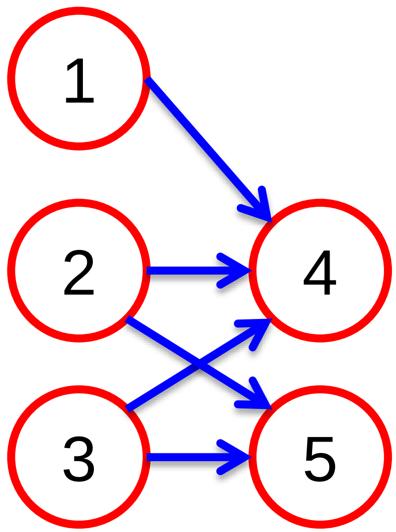
Running Example

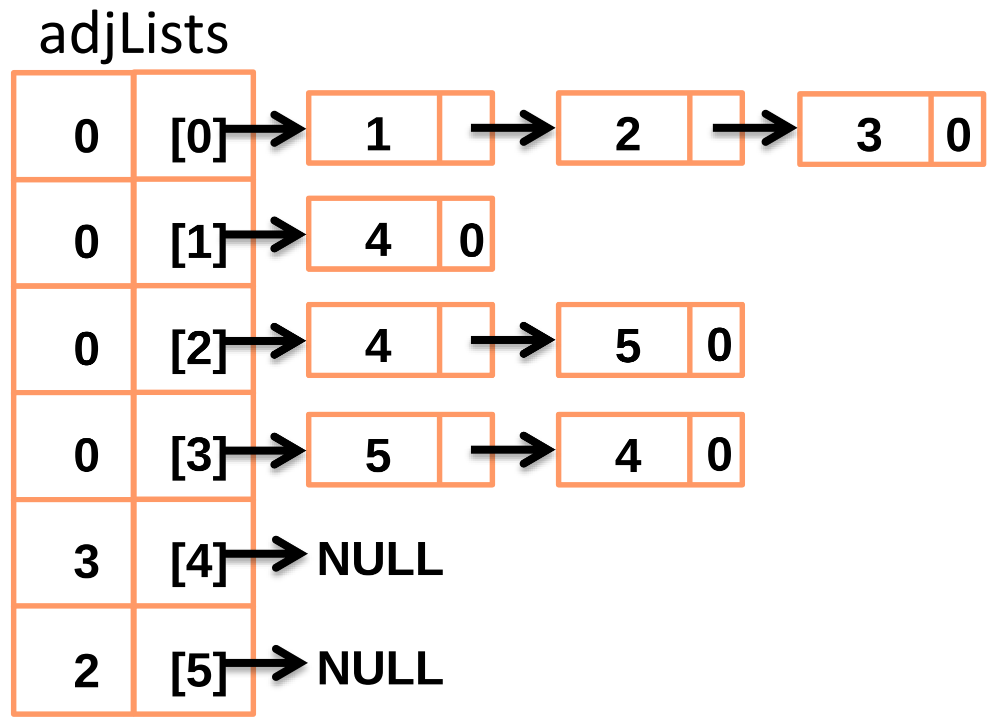




Ordered list:

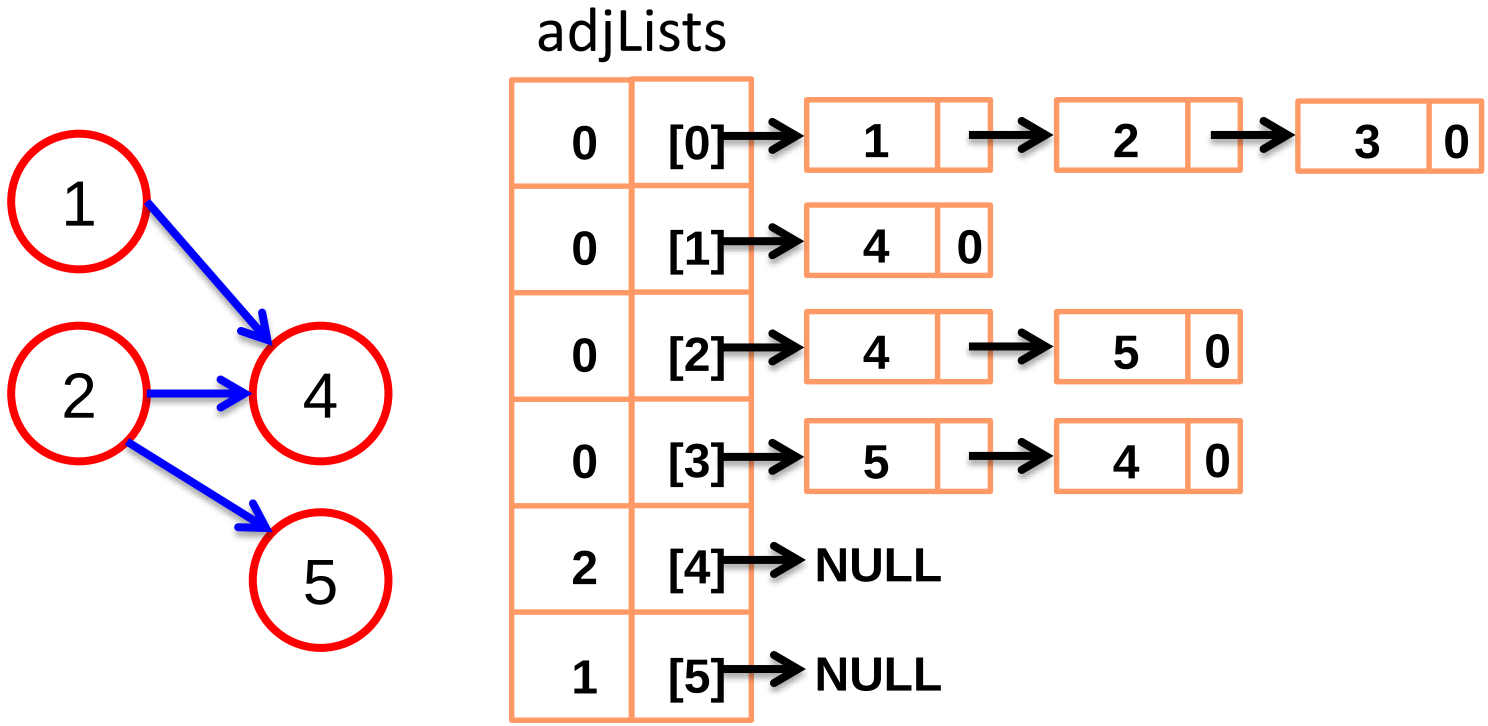
Running Example





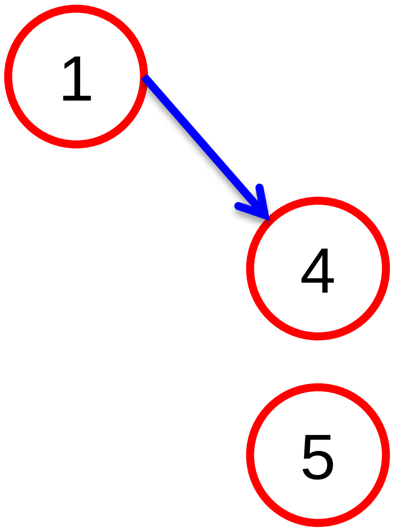
Ordered list: 0

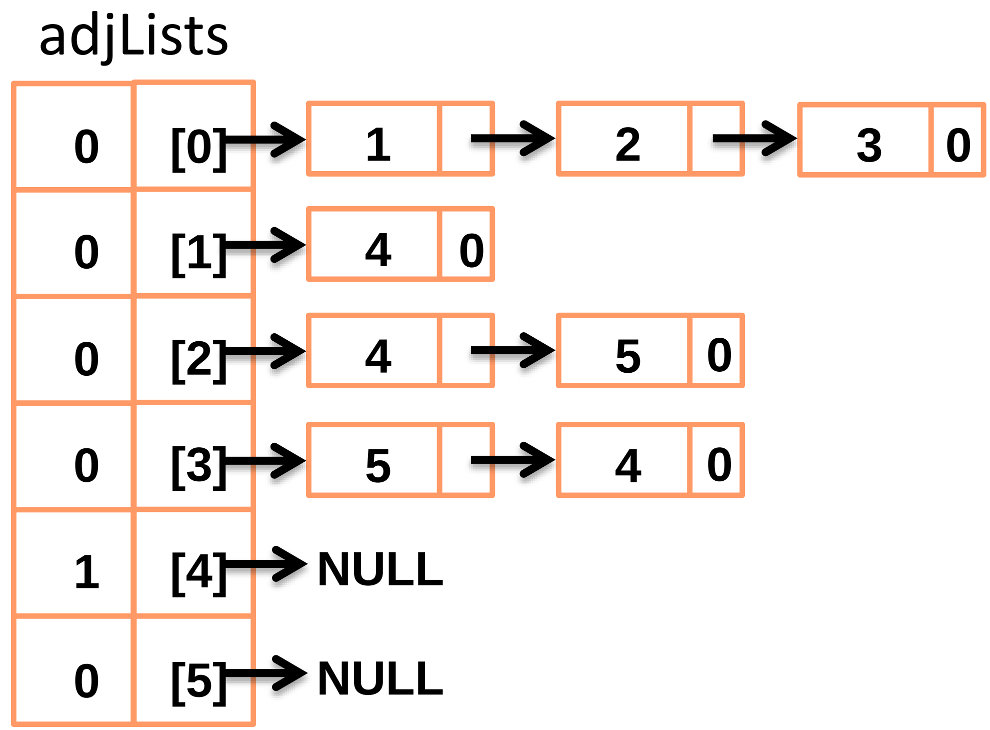
Running Example



Ordered list: 0 3

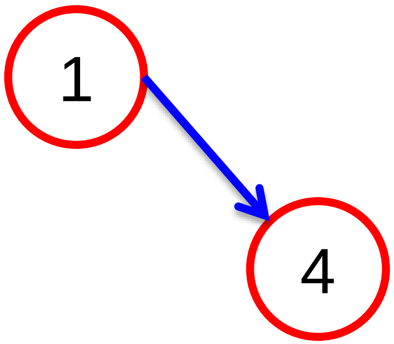
Running Example

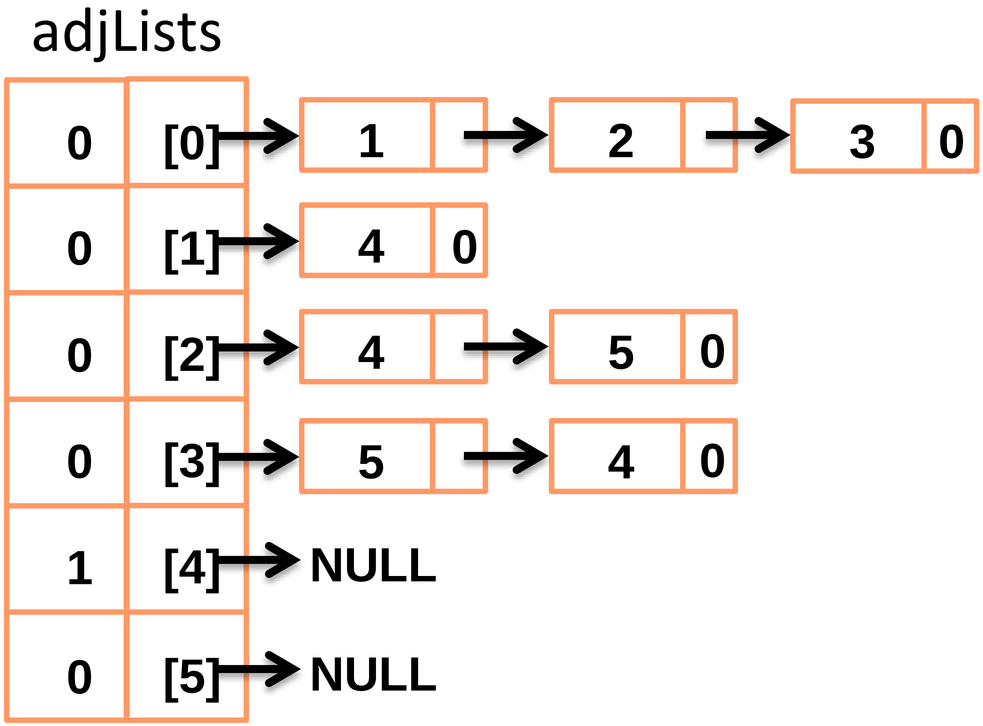




Ordered list: 0 3 2

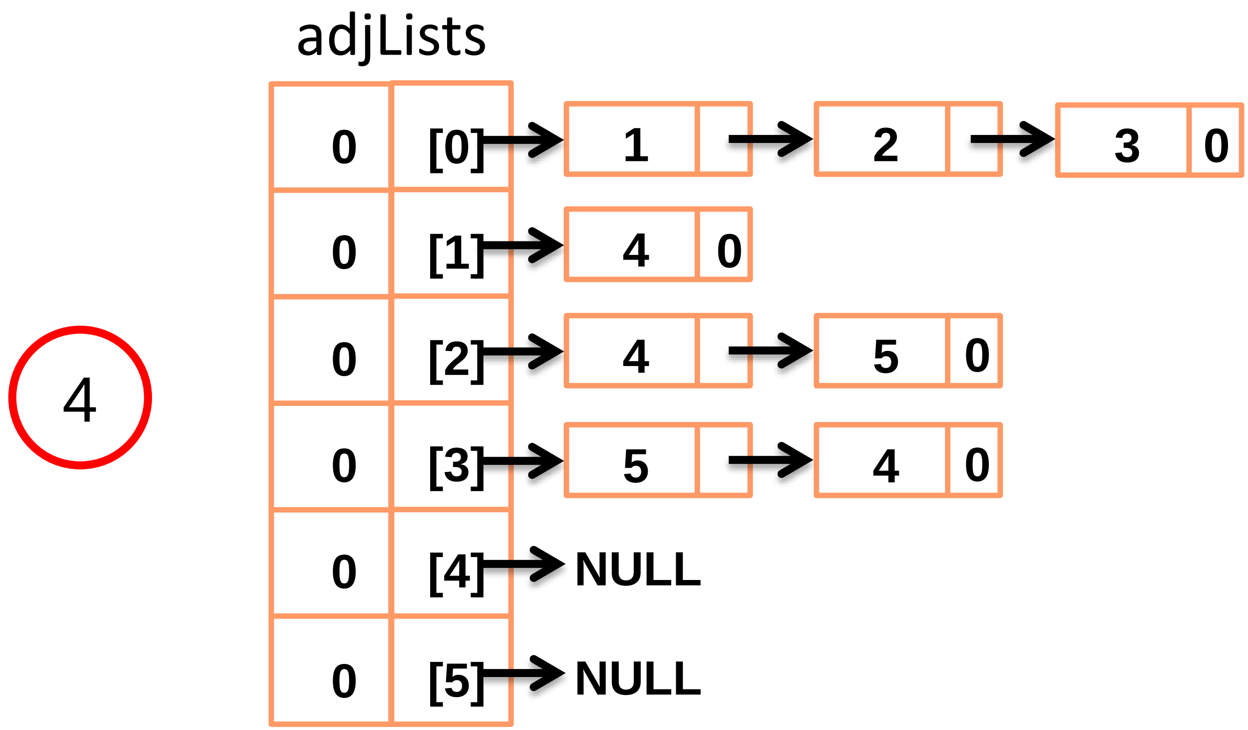
Running Example





Ordered list: 0 3 2 5

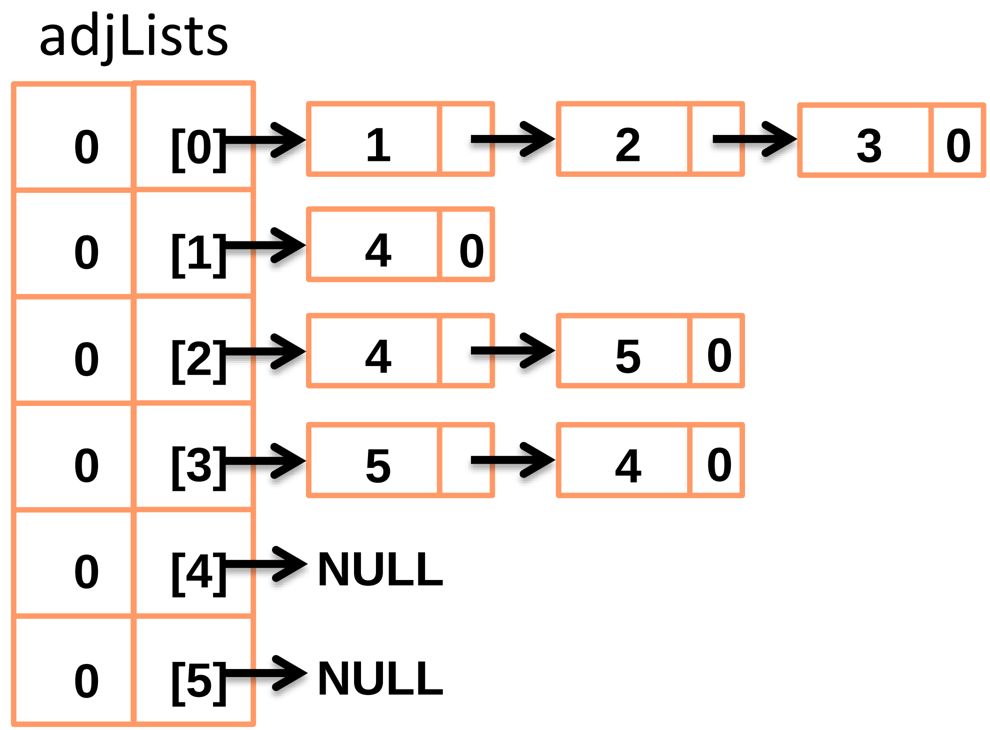
Running Example



Ordered list: 0 3 2 5 1

Running Example

If there are cycles in G, then the algorithm will end with some vertices still having predecessors and not being removed
→ digraph with no directed cycles is an *acyclic graph*



Ordered list: 0 3 2 5 1 4

Summary

- Finding the minimum cost spanning tree of a graph
  - Kruskal's, Prims's, and Sollin's algorithm
- Finding the shortest path and transitive closure
  - Dijkstra's Algorithm for single source/all destination
  - Floyd-Warshall's Algorithm for all-pairs shortest paths
- Activity-on-vertex (AOV) networks
  - Topological ordering
- Self-study topics
  - Single source shortest path:
    Digraph with negative edge costs
  - Activity-on-edge (AOE) networks: critical path analysis