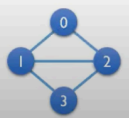
Graph, Vertex and Edge

- For a graph with n vertices, the maximum # of edges is:
  - $n(n-1)/2$ for undirected graph
  - $n(n-1)$ for directed graph
- Vertices u and v are **adjacent** if $(u, v) \in E$ and edge (u, v) is **incident** on vertices u and v.
- For a directed edge $< u, v >$, we say u is **adjacent to** v and v is **adjacent from** u, and edge (u, v) is **incident** on vertices u and v.

Path and Simple Path

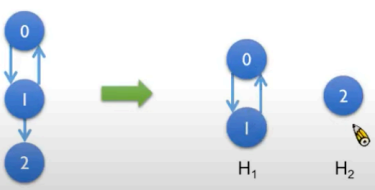
- **Path**:
  - A path from u to v represents a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$ such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in graph.
- **Simple path:**
  - A simple path is a path in which all vertices except possibly the first and the last are distinct.



Sequence	Path?	Simple path?
0,1,3,2	Yes	Yes
0,2,0,1	Yes	No
0,3,2,1	No	No

Strongly Connected Component

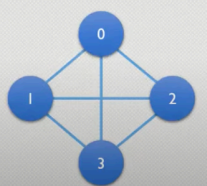
- A **strongly connected component** is a maximal subgraph that is strongly connected.



Two strongly connected components

Complete Graph

Complete undirected graph	Complete directed graph
Graph with n vertices has exactly $n(n-1)/2$ edges.	Graph with n vertices has exactly $n(n-1)$ edges.



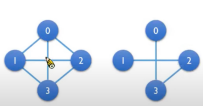
Strongly Connected

- Directed graph G is said to be **strongly connected** iff for **every pair of distinct vertices u and v**, there is a **directed path from u to v** and **also from v to u** in G.

Not a strongly connected digraph!
There is no directed path from 2 to 0

Subgraph

- G' is a subgraph of G such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.



① A cycle in a graph must be a simple path

② The strongly connected components of a graph can be identified by depth-first search

③ The sum of degree of nodes in a graph equals to twice of the number of edges

④ The minimum edges need for a n-node connected undirected graph : n-1

⑤ A connected graph with n vertices and n-1 edges must be a tree

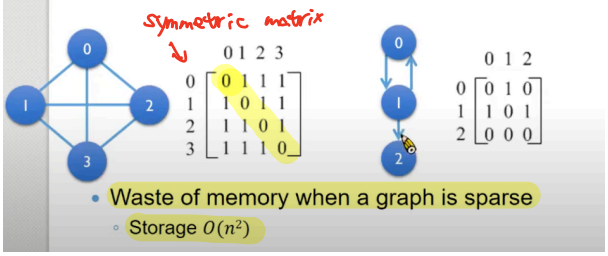
⑥ A spanning tree of a graph G is a maximal connected subgraph of G without cycle

⑦ A tree is a connected graph without cycles

⑧ A tree of n vertices has exactly n-1 edges

Adjacency Matrix

- A two dimensional array with the property that $a[i][j] = 1$ iff the edge (i, j) or $< i, j >$ is in $E(G)$.



- Waste of memory when a graph is sparse
  - Storage $O(n^2)$

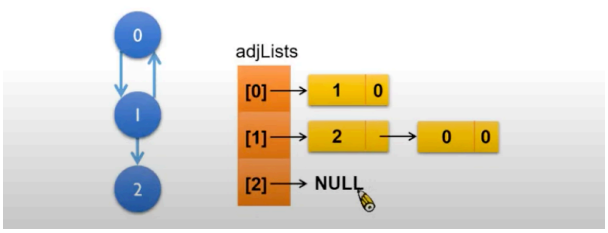
- Merits of Adjacency Matrix
  - The complexity of checking edge number or examining if *G* is connect
    - *G* is undirected: $O(n^2/2)$
    - *G* is directed: $O(n^2)$

Adjacency Lists

- Undirected graph: Use a chain to represent each vertex and its **adjacent** vertices.

Adjacency Lists

- Directed graph: Use a chain to represent each vertex and its **adjacent** vertices.

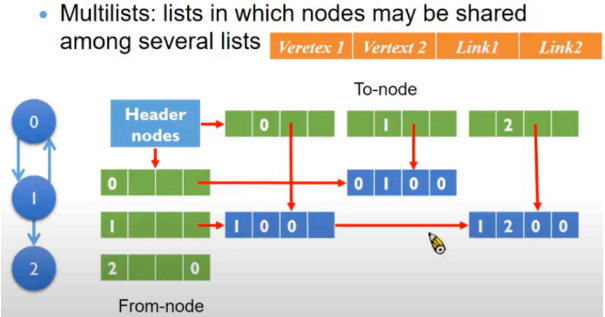


Inverse Adjacency Lists

- Directed graph: Use a chain to represent each vertex and its **adjacent from** vertices
  - Length of list = in-degree of v

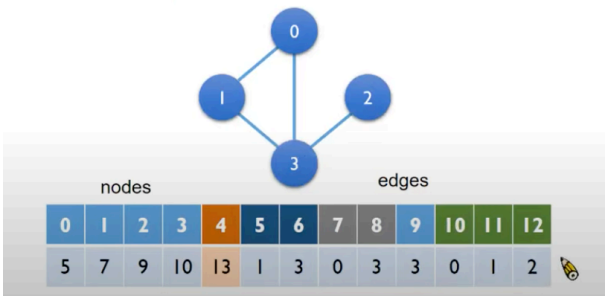
Adjacency Multilists

- Multilists: lists in which nodes may be shared among several lists

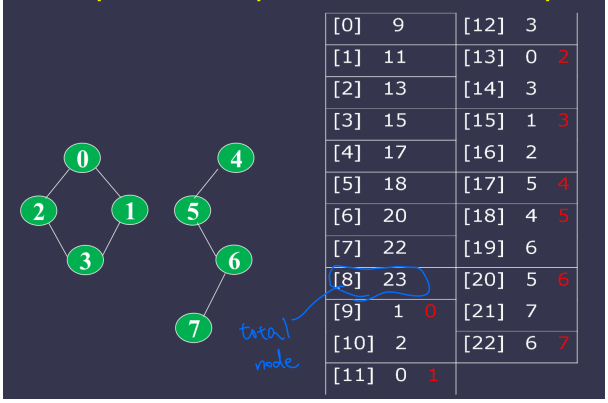


Sequential Representation

- Example: $n = 4, e = 4$
- Int nodes[$n + 2e + 1$] => nodes[13]



Sequential Representation of Graph



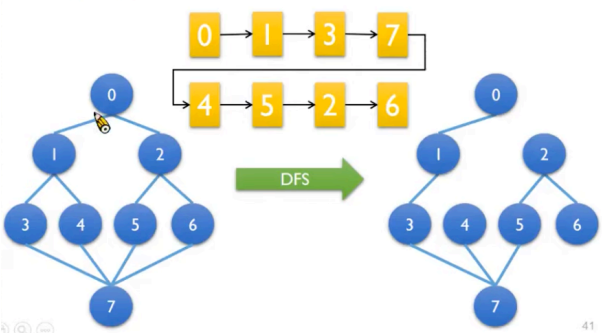
Sequential Representation (Undirected Graph) : Time Complexity O(n+e)

DFS Complexity

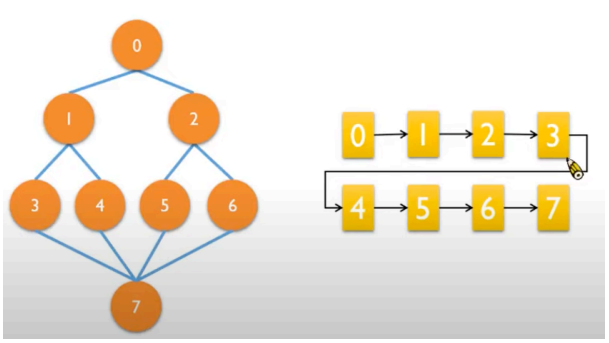
- Adjacency matrix
  - Time to determine all adjacent vertices: $O(n)$
  - At most n vertices are visited:
    $$O(n \cdot n) = O(n^2)$$
- Adjacency lists
  - There are $n + 2e$ chain nodes
  - Each node in the adjacency list is examined at most once. Time complexity = $O(e)$

DFS Spanning Tree

- Tree edges are those edges met during the traversal.

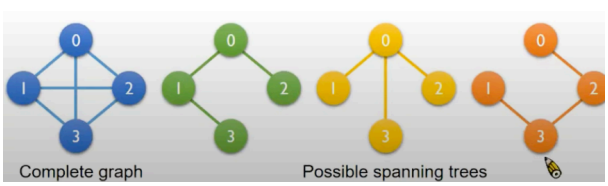


Breadth-First Search (BFS)



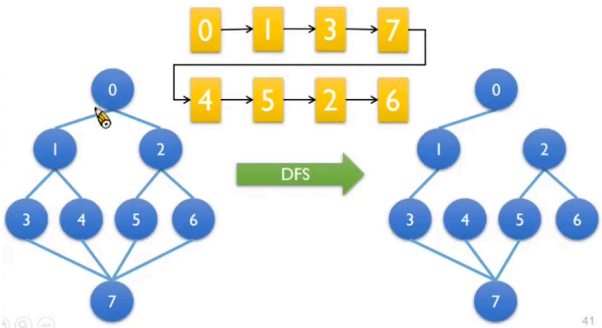
Spanning Trees

- Definition: Any tree consists of solely of edges in $E(G)$ and including all vertices of $V(G)$.
- Number of tree edges is $n - 1$.
- Add a non-tree edge will create a **cycle**.



DFS Spanning Tree

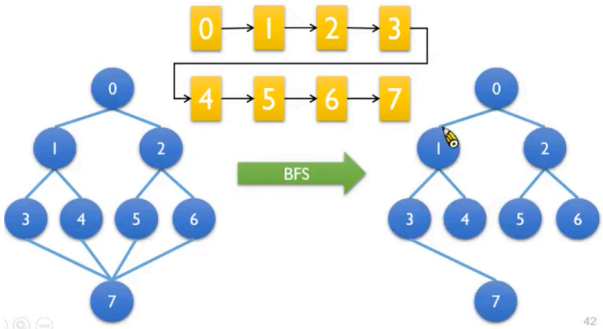
- Tree edges are those edges met during the traversal.



41

BFS Spanning Tree

- Tree edges are those edges met during the traversal.



42

Connected Components

- How to determine whether a graph is connected or not?
  - Call DFS or BFS once and check if there is any unvisited vertices, if Yes, then the graph is not connected.
- How to identify connected components
  - Make a repeated calls to DFS or BFS.
  - Each call will output a connected component.
  - Start next call at an unvisited vertex.

Kruskal's Algorithm

Idea: Add edges with minimum edge weight to tree one at a time.

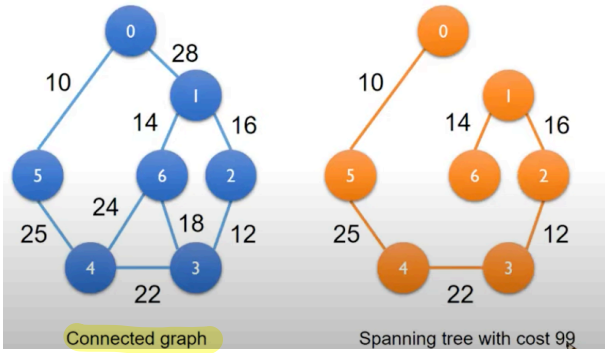
1. Find an edge with minimum cost.
2. If it creates a cycle, discard the edge.
3. Repeat step 1 and 2 until we find $n-1$ edges.

Time Complexity

- Min heap: $O(\log e)$
- Set: $O(a(e))$
- At most execute e rounds:
  - $e \cdot (\log e + a(e)) = O(e \log e)$

Example for Kruskal's Alg.

Refer to the textbook for detailed steps!



Prim's algorithm

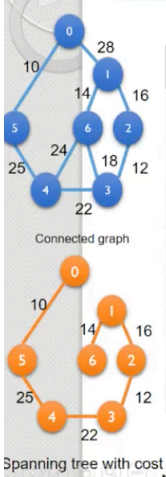
Idea: Add edges with minimum edge weight to tree one at a time. **At all times during the algorithm, the set of selected edges form a tree.**

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

Time Complexity

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

Example for Prim's Alg.



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

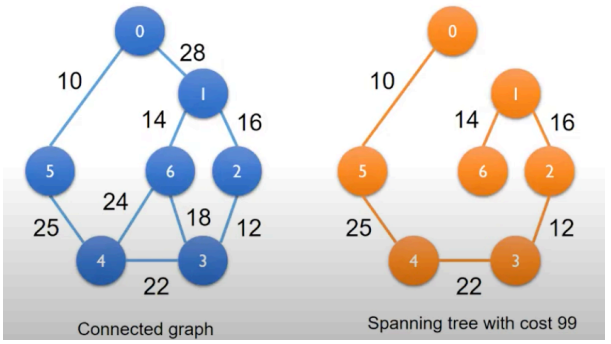
Sollin's Algorithm

Idea: Select several edges at each stage.

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

Example for Sollin's Alg.

Refer to the textbook for detailed steps!

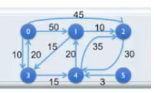


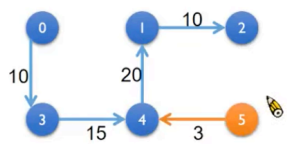
Dijkstra's Algorithm

- Similar to Prim's algorithm
- Use a set *S* store the vertices whose shortest path have been found
- Use an array *dist* store the shortest distances from source v to all visited vertices
- The algorithm
  1. Let *S*={*v*}, all entries in *dist* = ∞
  2. For each vertex *w* not in *S*, update *dist*
     $dis[w]=min(dist[u]+length(\langle u,w\rangle),dist[w])$
     u is the newly added vertex to *S* adjacent to w
  3. Add to *S* the vertex *x* not in *S* but of the minimum cost in *dist*.
  4. Repeat last two steps until *S* include all vertices.

Time Complexity : $O(n^2)$

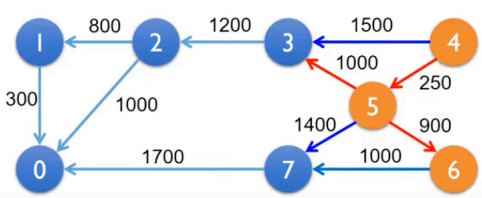
Running Example





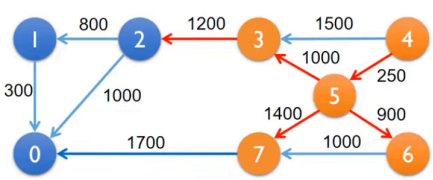
S	0	1	2	3	4	5
{0}	0	50	45	10	∞	∞
{0, 3}	0	50	45	10	25	∞
{0, 3, 4}	0	45	45	10	25	∞
{0, 3, 4, 1}	0	45	45	10	25	∞
{0, 3, 4, 1, 2}	0	45	45	10	25	∞

Running Example 2: 1/3



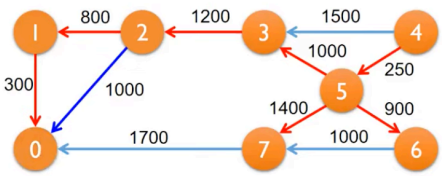
S	0	1	2	3	4	5	6	7
{4}	∞	∞	∞	1500	0	250	∞	∞
{4, 5}	∞	∞	∞	1250	0	250	1150	1650
{4, 5, 6}	∞	∞	∞	1250	0	250	1150	1650

Running Example 2: 2/3



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	1150	1650

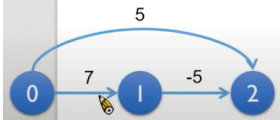
Running Example 2: 3/3



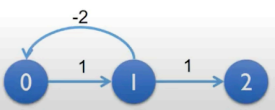
S	0	1	2	3	4	5	6	7
{4, 5, 6, 3, 7}	3350	∞	2450	1250	0	250	1150	1650
{4, 5, 6, 3, 7, 2}	3350	3250	2450	1250	0	250	1150	1650
{4, 5, 6, 3, 7, 2, 1}	3350	3250	2450	1250	0	250	1150	1650
{4, 5, 6, 3, 7, 2, 1, 0}	3350	3250	2450	1250	0	250	1150	1650

Digraph with Negative Costs

- This algorithm also applies to **digraph with negative cost edges**.
- However, the digraph **MUST NOT** contain cycles of negative length.



Digraph with a negative cost edge

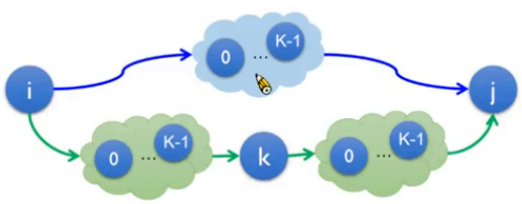


Digraph with a cycle of negative cost

Floyd-Warshall's Algorithm

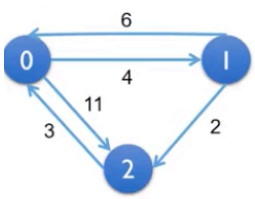
- There are only two possible paths for $A^k[i][j]$!
  - The path dose not pass vertex k.
  - The path dose pass vertex k.

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



Time Complexity : $O(n^3)$

Example

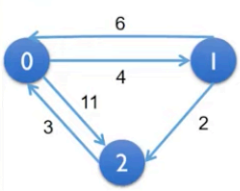


A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

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

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

Example

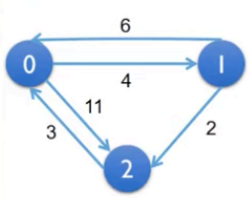


A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

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

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

Example



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

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

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

對角線全設為1，其餘不變

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!

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

Example: 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

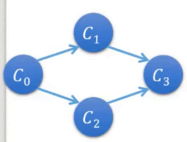
Transitive closure matrix

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

Topological Order

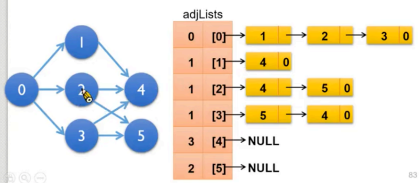
- A **linear ordering** of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering.



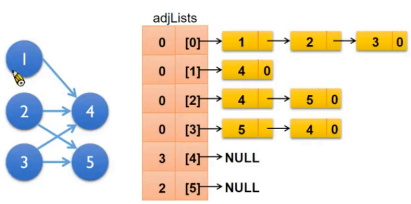
$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow C_3$ (O)
$C_0 \rightarrow C_2 \rightarrow C_1 \rightarrow C_3$ (O)
$C_0 \rightarrow C_2 \rightarrow C_3 \rightarrow C_1$ (X)

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.

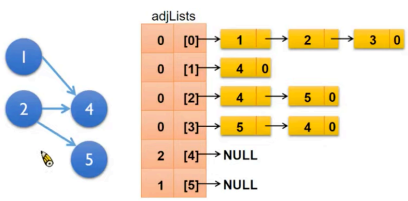


Running Example



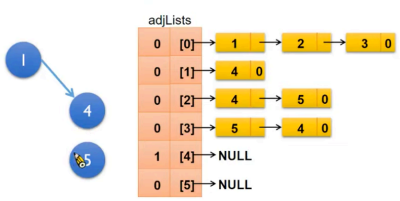
Ordered list: 0

Running Example



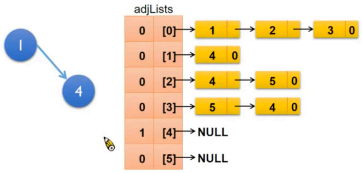
Ordered list: 0 3

Running Example



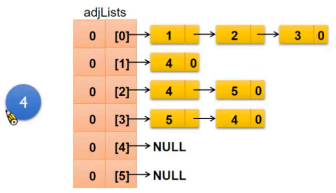
Ordered list: 0 3 2

Running Example



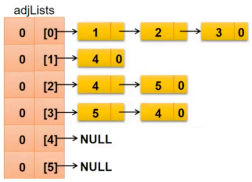
Ordered list: 0 3 2 5

Running Example



Ordered list: 0 3 2 5 1

Running Example



Ordered list: 0 3 2 5 1 4