เอกสารประกอบการอบรม
Graph Algorithms

ค่ายคอมพิวเตอร์โอลิมปิก สอวน. ค่าย 2 2/2565
ศูนย์โรงเรียนสามเสนวิทยาลัย - มหาวิทยาลัยธรรมศาสตร์
ระหว่างวันที่ 20 มีนาคม – 4 เมษายน 2566

โดย

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี

มหาวิทยาลัยธรรมศาสตร์

26 มีนาคม 2567

<u>อัลกอริทึม (Day 8):</u>

- Graph Algorithms

ผศ.ดร.ฐาปนา บุญชู

26 มีนาคม 2567

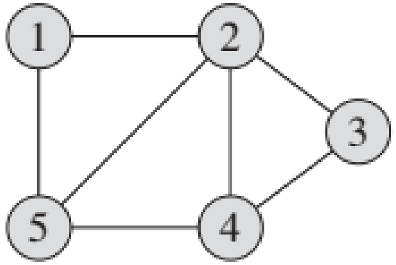
Graph Fundamentals and Traversal Algorithms

Representing a Graph

- We can choose between two standard ways to represent a graph $G = (V, E)$ as a collection of **adjacency lists** or as an **adjacency matrix**.
- Because the adjacency-list representation provides a compact way to represent **sparse** graphs—those for which $|E|$ is much less than $|V|^2$—it is usually the method of choice.
- We may prefer an adjacency-matrix representation, however, when the graph is **dense**—$|E|$ is close to $|V|^2$—or when we need to be able to tell quickly if there is an edge connecting two given vertices.

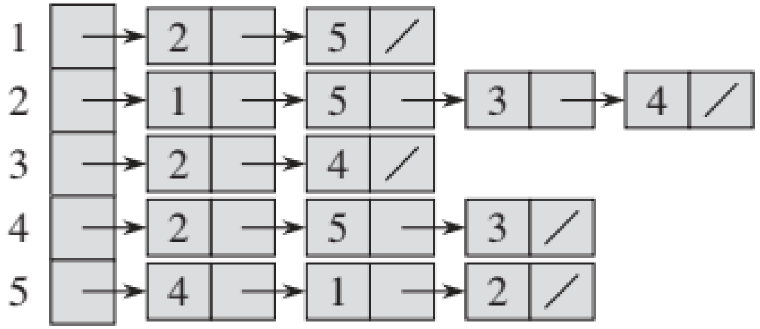
Representing a Graph

Graph



(a)

Represented using Adjacency Lists.



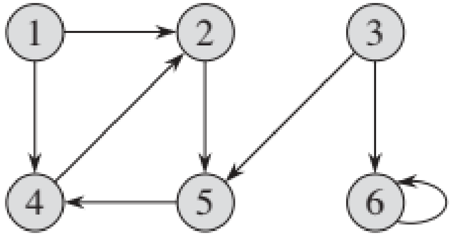
(b)

Represented using Adjacency Matrix.

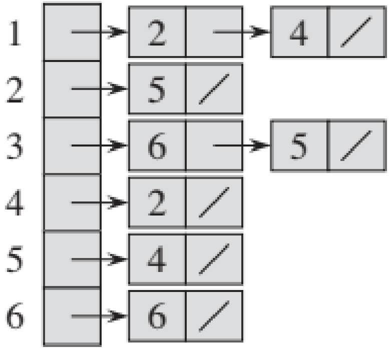
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Undirected graph



(a)



(b)

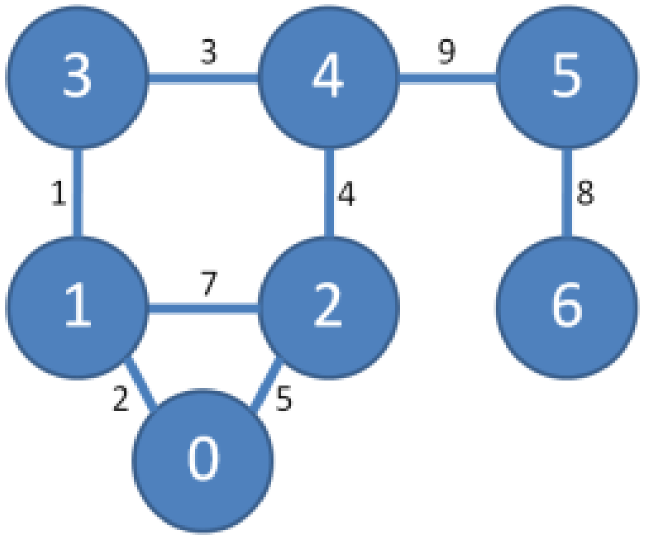
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Directed graph

Representing a Graph

- Weighted Graph



A

	0	1	2	3	4	5	6
0		2	5				
1	2		7	1			
2	5	7			4		
3		1			3		
4			4	3		9	
5					9		8
6						8	

B

0 → (1,2) → (2,5)
1 → (0,2) → (2,7) → (3,1)
2 → (0,5) → (1,7) → (4,4)
3 → (1,1) → (4,3)
4 → (2,4) → (3,3) → (5,9)
5 → (4,9) → (6,8)
6 → (5,8)

C

w	v1	v2
1	1	3
2	0	1
3	3	4
4	2	4
5	0	2
7	1	2
8	5	6
9	4	5

ข้อดี/ข้อเสีย ของ Graph representations

- Adjacency matrix:
  - Space complexity: $O(V^2)$
    - ในกราฟที่ Sparse => Element ส่วนใหญ่จะเป็น 0
    - ในการแข่งขันหาก V >= 1000 ควรพิจารณาใช้วิธีการ Represent แบบอื่น
  - เวลาที่ใช้ในการเข้าถึงเพื่อนบ้าน (Neighbors) คือ O(V) *** ใช้ค่อนข้างบ่อย
- Adjacency list:
  - ในภาษา C++ สามารถแทนด้วย **Vector of vector pairs**
  - กล่าวคือ `vector<vii> AdjList;`
  - และ `vii` สามารถนิยามด้วย `typedef pair<int, int> ii; typedef vector<ii> vii;`
  - Space complexity: O(V + E)
  - เวลาที่ใช้ในการเข้าถึง Neighbors คือ O(k) โดยที่ k คือจำนวนเพื่อนบ้าน k <<= V

ข้อดี/ข้อเสีย ของ Graph representations

- Edge list:
  - vector< pair<int, ii> > EdgeList; (Triple)
  - เป็นการเก็บ List ของ Vertex โดยปรกติจะต้องอยู่ใน Sorted order
  - มีประโยชน์ในการ Implement Kruskal’s algorithm สำหรับ Minimum Spanning Tree (MST)

Implicit graphs!

- 2D Grid:
  - Vertices => Cells;
  - Edge => Adjacent cells; (Neighbours)
- Graph ที่ Edges ถูกสร้างจากกฎบางอย่าง
  - เช่น กราฟที่ถูกสร้างขึ้นมาจาก N Vertices (นั่นคือ [0…N-1]) และ Edges เกิดขึ้นระหว่าง 2 Vertices i, j เมื่อ i+j = เลขจำนวนเฉพาะ

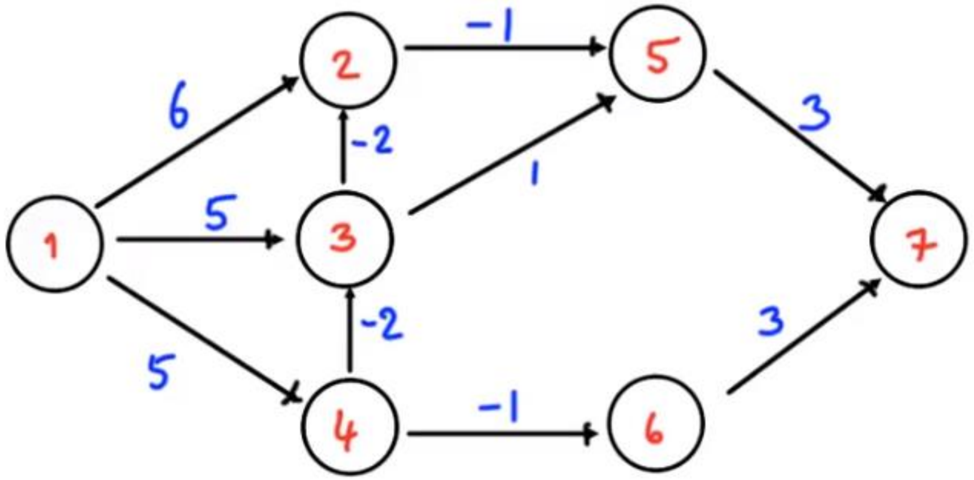
Check yourself..

Vertices/Nodes	Edges	Set V; size $\|V\|$	Set E; size $\|E\|$	Graph $G(V, E)$
Un/Weighted	Un/Directed	Sparse	Dense	In/Out Degree
Path	Cycle	Isolated	Reachable	Connected
Self-Loop	Multiple Edges	Multigraph	Simple Graph	Sub-Graph
DAG	Tree/Forest	Eulerian	Bipartite	Complete

List of Important Graph Terminologies

แบบฝึกหัด

- ให้นักเรียนเขียนโปรแกรมเพื่อ**แทน**กราฟในรูปด้านล่างต่อไปนี้ ด้วยวิธี Adjacency Matrix, Adjacency list และ Edge list



Solution – Adjacency List

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
```

```
int main() {
  int V, E, s, u, v, w;
  vector<vii> AdjList;

  scanf("%d %d %d", &V, &E, &s);

   AdjList.assign(V, vii());
   // assign blank vectors of
   // pair<int, int>s to AdjList

  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjList[u].push_back(ii(v, w));
  }
}
```

Test input#1

5 5 0

0 1 1

0 2 10

1 3 2

2 3 -10

3 4 3

Solution – Edge List

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
```

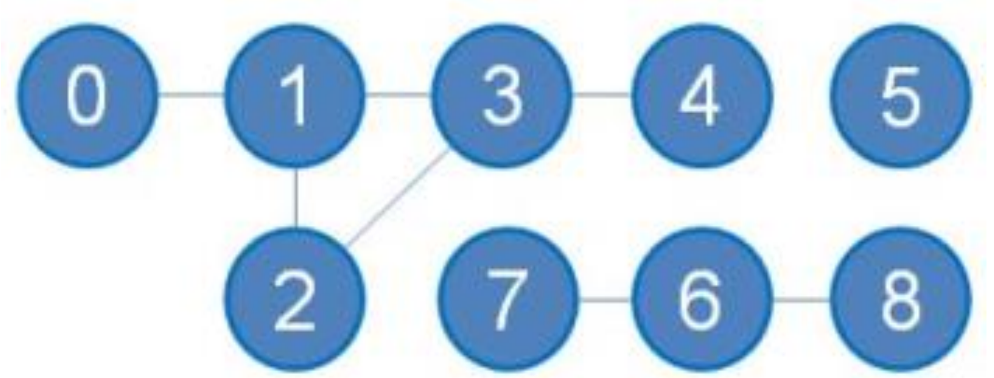
```
int main (){

  vector< pair<int, ii> > EdgeList;
  scanf("%d %d %d", &V, &E, &s);

  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    // read the triple: (u, v, w)
    EdgeList.push_back(make_pair(w, ii(u, v)));
  }

}
```

ทำไมถึง w เป็น first element ใน pair

BFS และ DFS

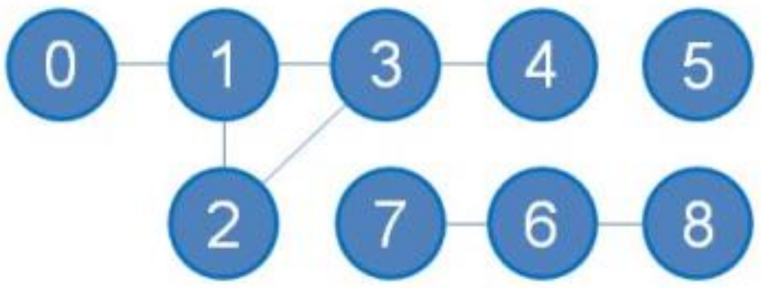
Depth First Search

- DFS เริ่มจาก Source vertex และทำการท่อง (Traverse) ลงไปในกราฟ
- เมื่อการท่องถึง Branching point (จุดที่มีเพื่อนบ้านมากกว่า 1) DFS จะเลือกไป Visit ยัง Nodes ที่ยังไม่เคยถูก Visited มาก่อน
- DFS จะทำกระบวนการนี้ซ้ำ ๆ และท่อง**ลึก**ลงไปเรื่อย ๆ จนกระทั่งไม่สามารถไปต่อได้ (เจอเจอที่เพื่อบ้านทุกคนถูก Visited ไปหมดแล้ว)
  - DFS จะทำการ Backtrack กลับไป State ก่อนหน้าเพื่อไป Visit Vertices ที่ยังไม่เคยถูก Visited มาก่อน



C++ Implementation for DFS

```
typedef pair<int, int> ii; // In this chapter, we will frequently use these
typedef vector<ii> vii; // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but they are useful in competitive programming

vi dfs_num;   // global variable, initially all values are set to UNVISITED

void dfs(int u) {      // DFS for normal usage: as graph traversal algorithm
  dfs_num[u] = VISITED;          // important: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) { // default DS: AdjList
    ii v = AdjList[u][j];                     // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == UNVISITED)    // important check to avoid cycle
      dfs(v.first);    // recursively visits unvisited neighbors of vertex u
} }  // for simple graph traversal, we ignore the weight stored at v.second
```



- DFS ควรจะ Traverse ลำดับอย่างไร

- ให้นักเรียนสังเกต**<u>โครงสร้างข้อมูล</u>**ที่ใช้

Backtrack version!

```
void backtrack(state) {
  if (hit end state or invalid state)              // we need terminating or
    return;    // pruning condition to avoid cycling and to speed up search
  for each neighbor of this state                    // try all permutation
    backtrack(neighbor);
}
```

Breath First Search

- BFS เริ่มจาก Source vertex และทำการท่อง (Traverse) ลงไปในกราฟ
- BFS จะไป Visit ยังเพื่อนบ้านในเลเวลที่ใกล้กับ Vertex ที่ Expand ให้ครบก่อนไปแล้วจึงท่องลงไปในเลเวลต่อไปเรื่อย ๆ (Implement โดยใช้ Queue)

```
// inside int main()---no recursion
  vi d(V, INF); d[s] = 0;                     // distance from source s to s is 0
  queue<int> q; q.push(s);                                  // start from source

  while (!q.empty()) {
    int u = q.front(); q.pop();                           // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                               // for each neighbor of u
      if (d[v.first] == INF) {          // if v.first is unvisited + reachable
        d[v.first] = d[u] + 1;           // make d[v.first] != INF to flag it
        q.push(v.first);              // enqueue v.first for the next iteration
  } } }
```

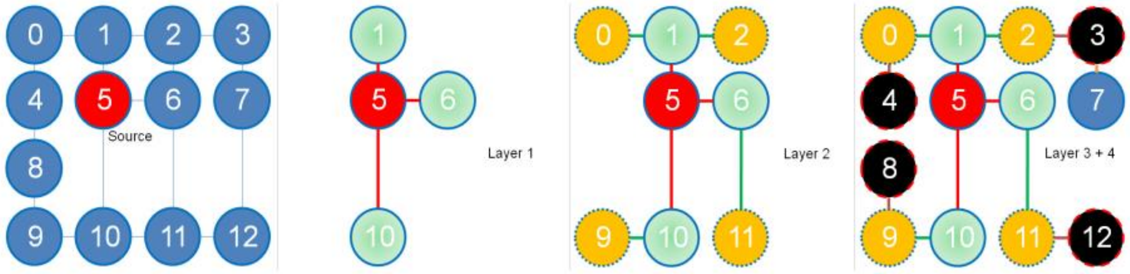
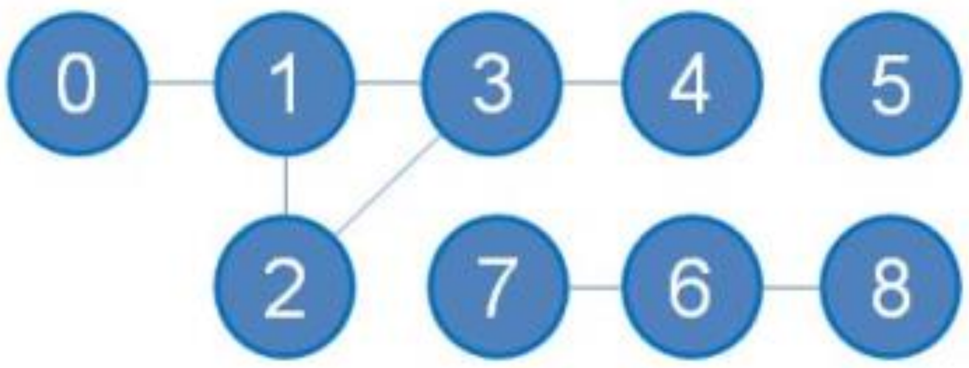


Figure 4.3: Example Animation of BFS

```
Layer 0:, visit 5
Layer 1:, visit 1, visit 6, visit 10
Layer 2:, visit 0, visit 2, visit 11, visit 9
Layer 3:, visit 4, visit 3, visit 12, visit 8
Layer 4:, visit 7
```

แบบฝึกหัด

- **Problem**: ให้นักเรียนเขียนลำดับการ visit nodes ตามความเข้าใจ เมื่อทำเสร็จแล้ว
- ให้เช็คคำตอบกับผลลัพธ์จากการรันโปรแกรมของตนเอง (ทั้ง DFS และ BFS)



Finding Connected Components (Undirected Graph)

Finding Connected Components (Undirected Graph)

- การหา Connected Components ของกราฟ

```
// inside int main()---this is the DFS solution
  numCC = 0;
  dfs_num.assign(V, UNVISITED);    // sets all vertices' state to UNVISITED
  for (int i = 0; i < V; i++)                // for each vertex i in [0..V-1]
    if (dfs_num[i] == UNVISITED)             // if vertex i is not visited yet
      printf("CC %d:", ++numCC), dfs(i), printf("\n");      // 3 lines here!

// For the sample graph in Figure 4.1, the output is like this:
// CC 1: 0 1 2 3 4
// CC 2: 5
// CC 3: 6 7 8
```

- ปัญหานี้สามารถใช้ Union-find data structure แก้ได้เช่นเดียวกัน

Flood Fill - Labeling/Coloring the Connected Components

Flood Fill - Labeling/Coloring the Connected Components

- ใช้สำหรับ Label (To color), นับจำนวน Connected components, หา Component sizes ของแต่ละ Connected component
- รู้จักในชื่อ “flood fill”

```
int dr[] = {1,1,0,-1,-1,-1, 0, 1};  // trick to explore an implicit 2D grid
int dc[] = {0,1,1, 1, 0,-1,-1,-1};         // S,SE,E,NE,N,NW,W,SW neighbors

int floodfill(int r, int c, char c1, char c2) {   // returns the size of CC
  if (r < 0 || r >= R || c < 0 || c >= C) return 0;          // outside grid
  if (grid[r][c] != c1) return 0;                         // does not have color c1
  int ans = 1;   // adds 1 to ans because vertex (r, c) has c1 as its color
  grid[r][c] = c2;    // now recolors vertex (r, c) to c2 to avoid cycling!
  for (int d = 0; d < 8; d++)
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans;                      // the code is neat due to dr[] and dc[]
}
```

	NW	N	NE	
	W	r, c	E	
	SW	S	SE	

	r-1, c-1	r-1, c+0	r-1, c+1	
	r+0, c-1	r, c	r+0, c+1	
	r+1, c-1	r+1, c+0	r+1, c+1	

Flood Fill - Labeling/Coloring the Connected Components

UVa 469 - Wetlands of Florida

```
// inside int main()
  // read the grid as a global 2D array + read (row, col) query coordinates
  printf("%d\n", floodfill(row, col, 'W', '.'));  // count size of wet area
                                                   // the returned answer is 12
// LLLLLLLLL           LLLLLLLLL
// LLWWLLWLL           LL..LLWLL   //          The size of connected component
// LWWLLLLLL (R2,C1) L..LLLLLL   //                          (the connected 'W's)
// LWWWLWWLL           L...L..LL   // with one 'W' at (row 2, column 1) is 12
// LLLWWWLLL ======> LLL...LLL
// LLLLLLLLL           LLLLLLLLL   //     Notice that all these connected 'W's
// LLLWWLLWL           LLLWWLLWL   //  are replaced with '.'s after floodfill
// LLWLWLLLL           LLWLWLLLL
// LLLLLLLLL           LLLLLLLLL
```

Topological Sort (Directed Acyclic Graph)

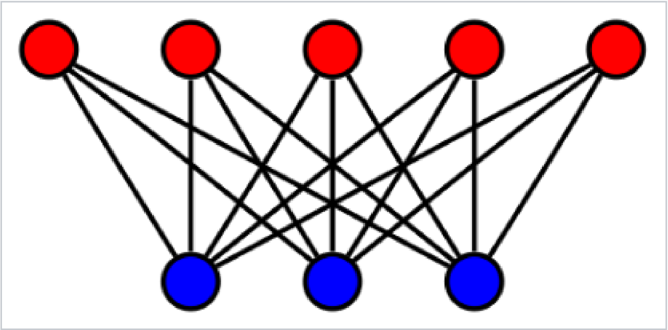
```
vi ts;                    // global vector to store the toposort in reverse order

void dfs2(int u) {  // different function name compared to the original dfs
  dfs_num[u] = VISITED;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      dfs2(v.first);
  }
  ts.push_back(u); }                   // that's it, this is the only change
```

```
// inside int main()
  ts.clear();
  memset(dfs_num, UNVISITED, sizeof dfs_num);
  for (int i = 0; i < V; i++)          // this part is the same as finding CCs
    if (dfs_num[i] == UNVISITED)
      dfs2(i);
               // alternative, call: reverse(ts.begin(), ts.end()); first
  for (int i = (int)ts.size() - 1; i >= 0; i--)           // read backwards
    printf(" %d", ts[i]);
  printf("\n");

// For the sample graph in Figure 4.4, the output is like this:
// 7 6 0 1 2 5 3 4   (remember that there can be >= 1 valid toposort)
```

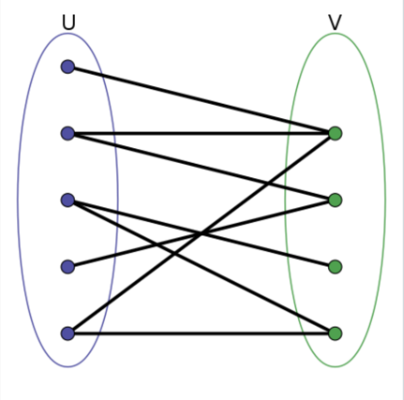
Bipartite Graph Check

Bipartite Graph Check

- ต้องการตรวจสอบว่ากราฟเป็น Bipartite Graph หรือไม่
- อาจจะใช้ BFS หรือ DFS ก็ได้ แต่เราจะได้ BFS ในที่นี้
  - เริ่มจากให้สี Source vertex (First layer) ด้วยค่า 0
  - และให้ค่า Neighbor (Second layer) ด้วยสีที่แตกต่างกันคือค่า 1
  - และให้ค่า Neighbor (Third layer) ด้วยสีที่แตกต่างกันคือค่า 0
  - ทำต่อไปเรื่อย ๆ
- ในระหว่าง Coloring นี้เมื่อใดก็ตามพบว่า Vertices ใด ๆ ที่เป็น Neighbours กันมีสีเดียวกัน จะตอบว่า **ไม่ใช่ Bipartite Graph**



A complete bipartite graph with m = 5 and n = 3



Example of a bipartite graph without cycles

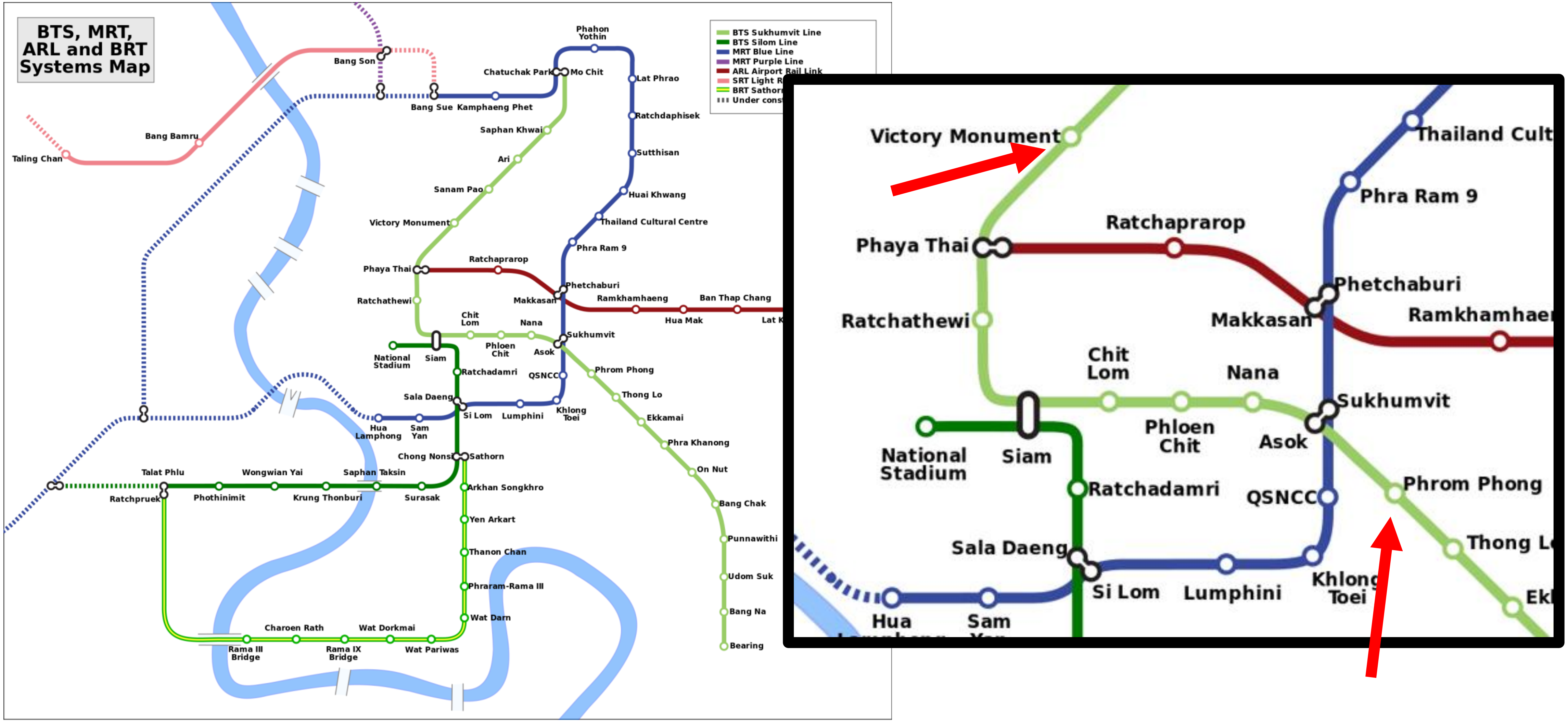
Bipartite Graph Check

```
// inside int main()
  queue<int> q; q.push(s);
  vi color(V, INF); color[s] = 0;
  bool isBipartite = true;  // addition of one boolean flag, initially true
  while (!q.empty() & isBipartite) { // similar to the original BFS routine
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (color[v.first] == INF) {   // but, instead of recording distance,
        color[v.first] = 1 - color[u];  // we just record two colors {0, 1}
        q.push(v.first); }
      else if (color[v.first] == color[u]) {  // u & v.first has same color
        isBipartite = false; break; } } }    // we have a coloring conflict
```

Try to solve!! => UVa 10004 - Bicoloring

Single-Source Shortest Paths

Single-Source Shortest Paths



Single-Source Shortest Paths (SSSP)

- Given a weighted graph G and a starting source vertex s, what are the shortest paths from s to every other vertices of G?
- For a **unweighted** graph (or all edges have equal or constant weight)
  - Use the efficient **O(V + E) BFS algorithm.**
- For a general **weighted** graph,
  - BFS does not work correctly.
  - Use algorithms like the **O((V + E) log V) Dijkstra's** algorithm or
  - Use **O(VE) Bellman Ford**'s algorithm (SSSP on Graph with Negative Weight Cycle).

Shortest-paths problem

Note: Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that we would want to minimize

- In a shortest-paths problem, we are given a weighted, directed graph $G(V, E)$, with weight function $w: E \rightarrow R$ mapping edges to real-valued weights.
- The weight $w(p)$ of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- We define the shortest-path weight $\delta(u, v)$ from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

- A **shortest path** from vertex u to vertex v is then defined as any path p with weight $\mathrm{w}(p) = \delta(u, v)$.

Optimal substructure of a shortest path

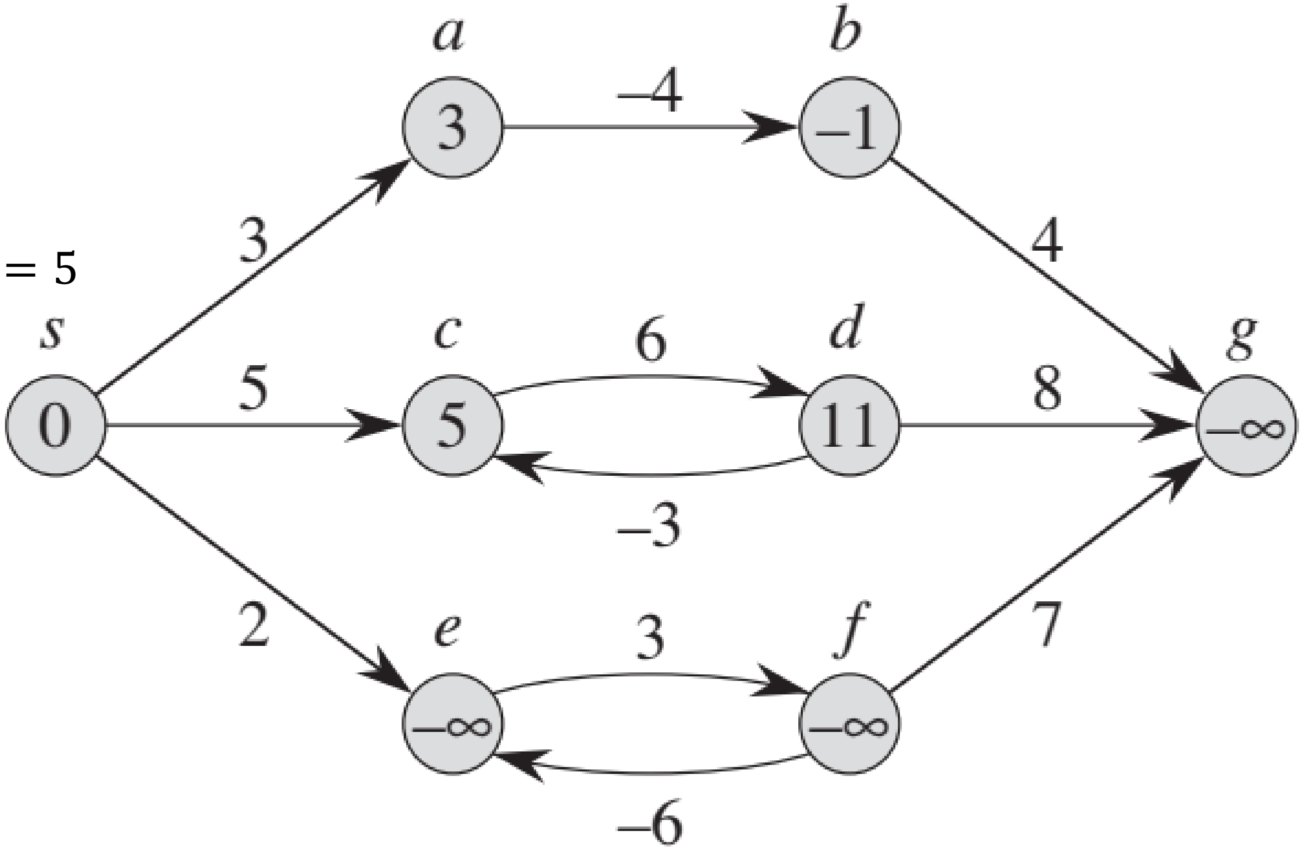
- Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.
- Recall that optimal substructure is one of the key indicators that **dynamic programming** and the **greedy method** might apply.
- **Dijkstra's algorithm** is a greedy algorithm.

Negative-weight edges

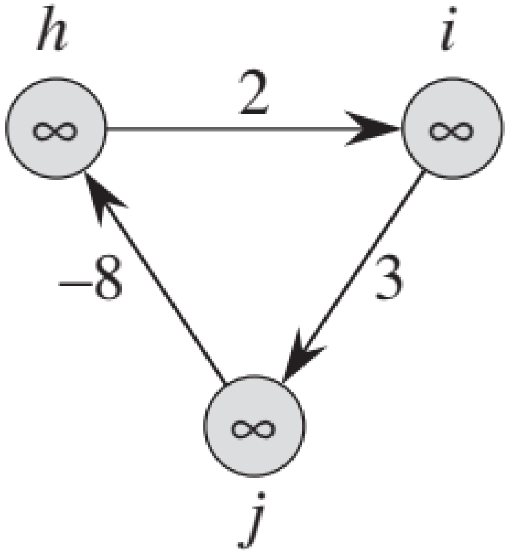
- Some instances of the single-source shortest-paths problem may include edges whose weights are **negative**.
- If the graph $G(V, E)$ contains **no negative weight cycles** reachable from the source S, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
- If the graph contains a **negative-weight cycle** reachable from s, however, shortest-path weights are not well defined.

Negative-weight edges

$\delta(s,c) = 5$



$\delta(s,e) = -\infty$



Cycles

- Can a shortest path contain a cycle? As we have just seen, it cannot contain a **negative-weight cycle**.
- How about **positive-weight cycle?**
  - Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.
- Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles, i.e., they are simple paths.

Relaxation

- For each vertex $v \in V$, we maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v.
  - We call $v.d$ a **shortest-path estimate**.
- We initialize the **shortest-path estimates** and **predecessors** by the following $\Theta(V)$-time procedure:

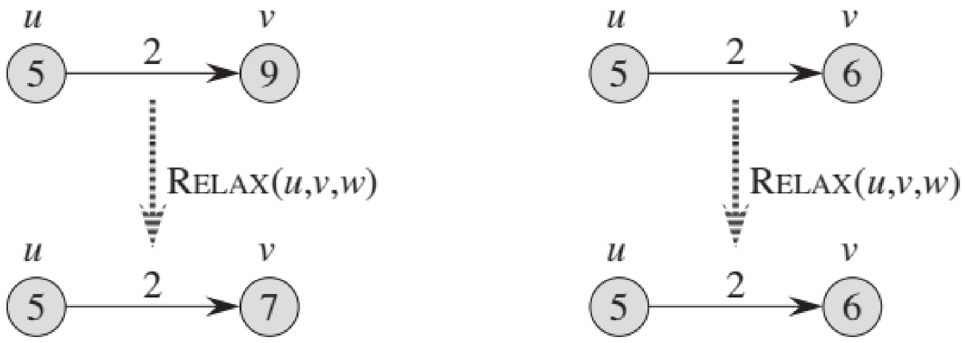
INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** each vertex $v \in G.V$
2 $\quad v.d = \infty$
3 $\quad v.\pi = \text{NIL}$
4 $s.d = 0$

Relaxation

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$.
- The following code performs a relaxation step on edge (u, v) in $O(1)$ time:

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```



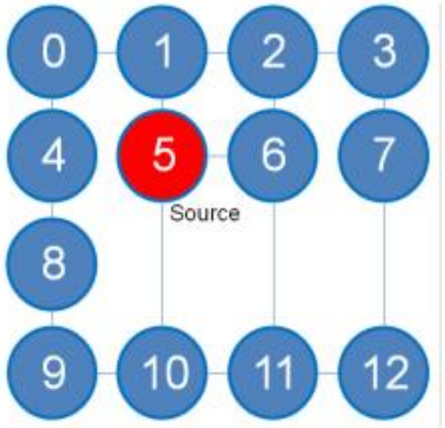
(a) (b)

SSSP on Unweighted Graph

Using BFS

SSSP on Unweighted Graph

- Some programming problems require us to reconstruct the actual shortest path, not just the shortest path length.
  - For example, in Figure 4.3, the shortest path from 5 to 7 is $5 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 7$.
  - This can be easily done using vector of integers `vi p`.
  - Each vertex `v` remembers its parent `u (p[v] = u)` in the shortest path spanning tree.

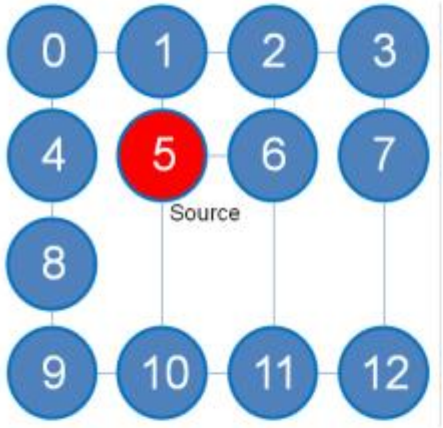


```
void printPath(int u) {                          // extract information from 'vi p'
  if (u == s) { printf("%d", s); return; }       // base case, at the source s
  printPath(p[u]);   // recursive: to make the output format: s -> ... -> t
  printf(" %d", u); }
```

```
// inside int main()
  vi dist(V, INF); dist[s] = 0;              // distance from source s to s is 0
  queue<int> q; q.push(s);
  vi p;                                 // addition: the predecessor/parent vector
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] == INF) {
        dist[v.first] = dist[u] + 1;
        p[v.first] = u;        // addition: the parent of vertex v.first is u
        q.push(v.first);
  } } }
  printPath(t), printf("\n");      // addition: call printPath from vertex t
```

แบบฝึกหัด

- เขียนโปรแกรมเพื่อหา SSSP โดย Source node = 5 ของกราฟด้านล่างต่อไปนี้



SSSP on Weighted Graph

SSSP on Weighted Graph

- If the given graph is weighted, **BFS does not work**.
- To solve the SSSP problem on weighted graph, we use a greedy Edsger Wybe **Dijkstra's algorithm**.

Dijkstra’s algorithm

- Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are **nonnegative**.
- As we shall see, with a good implementation, the running time of Dijkstra’s algorithm is lower than that of the Bellman-Ford algorithm.
- Dijkstra’s algorithm maintains a set S of vertices whose final shortest-path weights from the source S have already been determined.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the **minimum shortest-path estimate**, adds u to S, and relaxes all edges leaving u.

Dijkstra's algorithm

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX(u, v, w)

INITIALIZE-SINGLE-SOURCE(G, s)

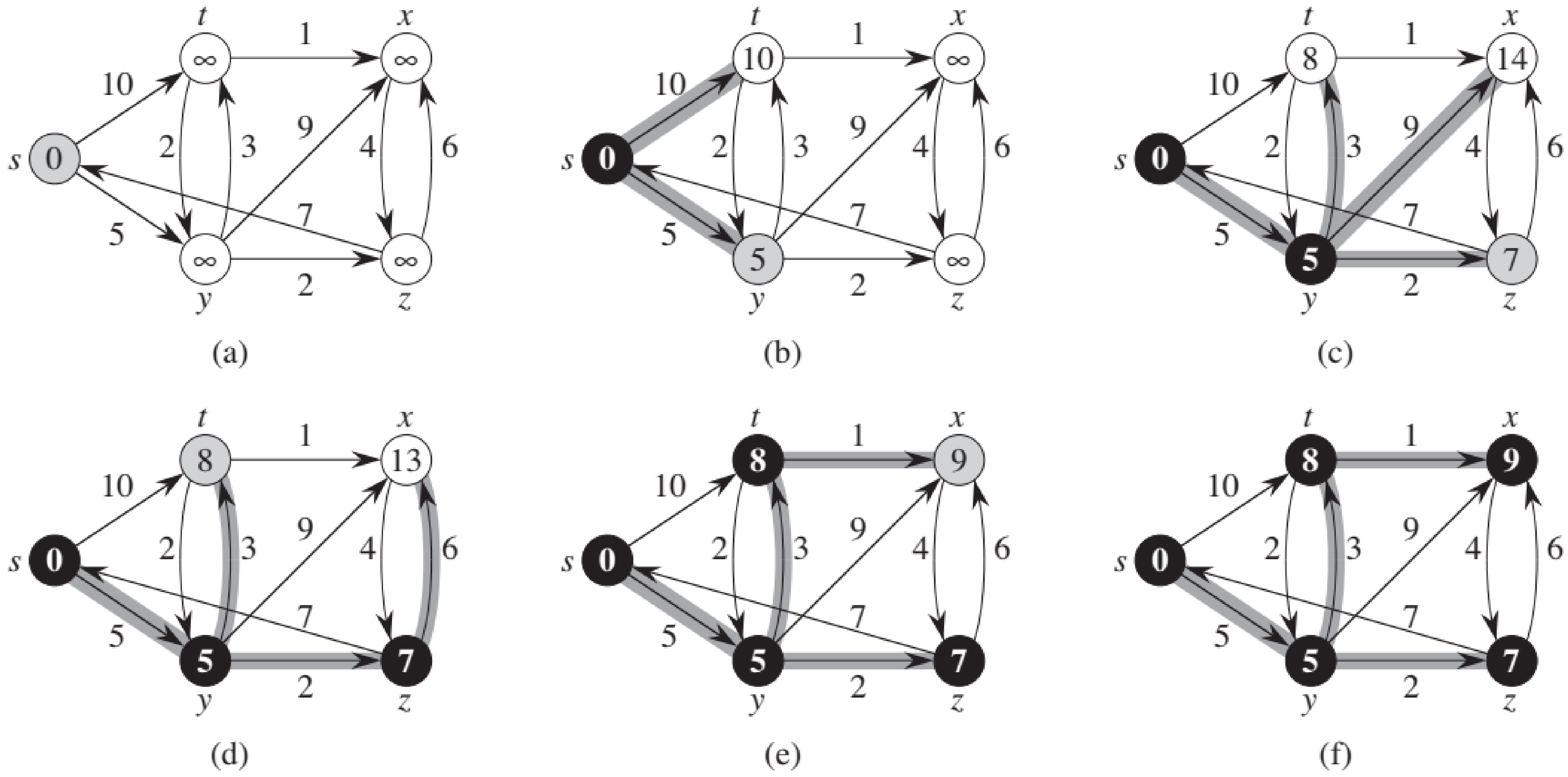
for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi =$ NIL
$s.d = 0$

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

เมื่อไหร่ก็ตามที่เราเพิ่ม u เข้าไปยัง Set S จะหมายถึง $u.d = \delta(s, u)$

Dijkstra’s algorithm

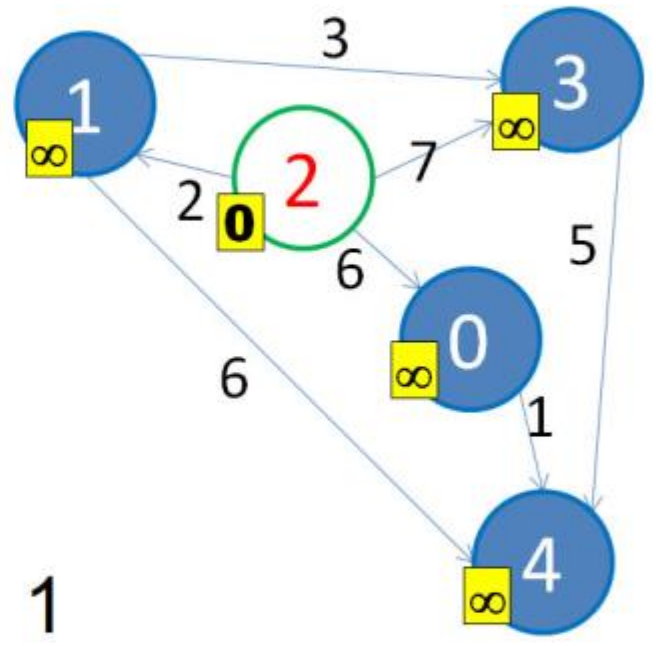


Dijkstra's algorithm (C++ with priority queue)

```
vi dist(V, INF); dist[s] = 0;                      // INF = 1B to avoid overflow
priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
while (!pq.empty()) {                                              // main loop
  ii front = pq.top(); pq.pop(); // greedy: get shortest unvisited vertex
  int d = front.first, u = front.second;
  if (d > dist[u]) continue;                 // this is a very important check
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                          // all outgoing edges from u
    if (dist[u] + v.second < dist[v.first]) {
      dist[v.first] = dist[u] + v.second;                   // relax operation
      pq.push(ii(dist[v.first], v.first));
} } }     // this variant can cause duplicate items in the priority queue
```

แบบฝึกหัด

- เขียนโปรแกรมเพื่อหา SSSP โดย Source node = 2 ของกราฟด้านล่างต่อไปนี้



SSSP on Graph with Negative Weight Cycle

SSSP on Graph with Negative Weight Cycle

- If the input graph has negative edge weight, typical Dijkstra's implementation can produce wrong answer.
- To solve the SSSP problem in the potential presence of negative weight cycle(s), the more generic (but slower) Bellman Ford's algorithm must be used.

The Bellman-Ford algorithm

- The **Bellman-Ford algorithm** solves the single-source shortest-paths problem in the general case in which edge weights may be **negative**.
- Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w: E \to R$, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a **negative-weight cycle** that is reachable from the **source**.
  - If there is such a cycle, the algorithm indicates that no solution exists.
  - If there is no such cycle, the algorithm produces the shortest paths and their weights.
- The Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.
- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $\delta(s, v)$ until it achieves the actual shortest-path weight

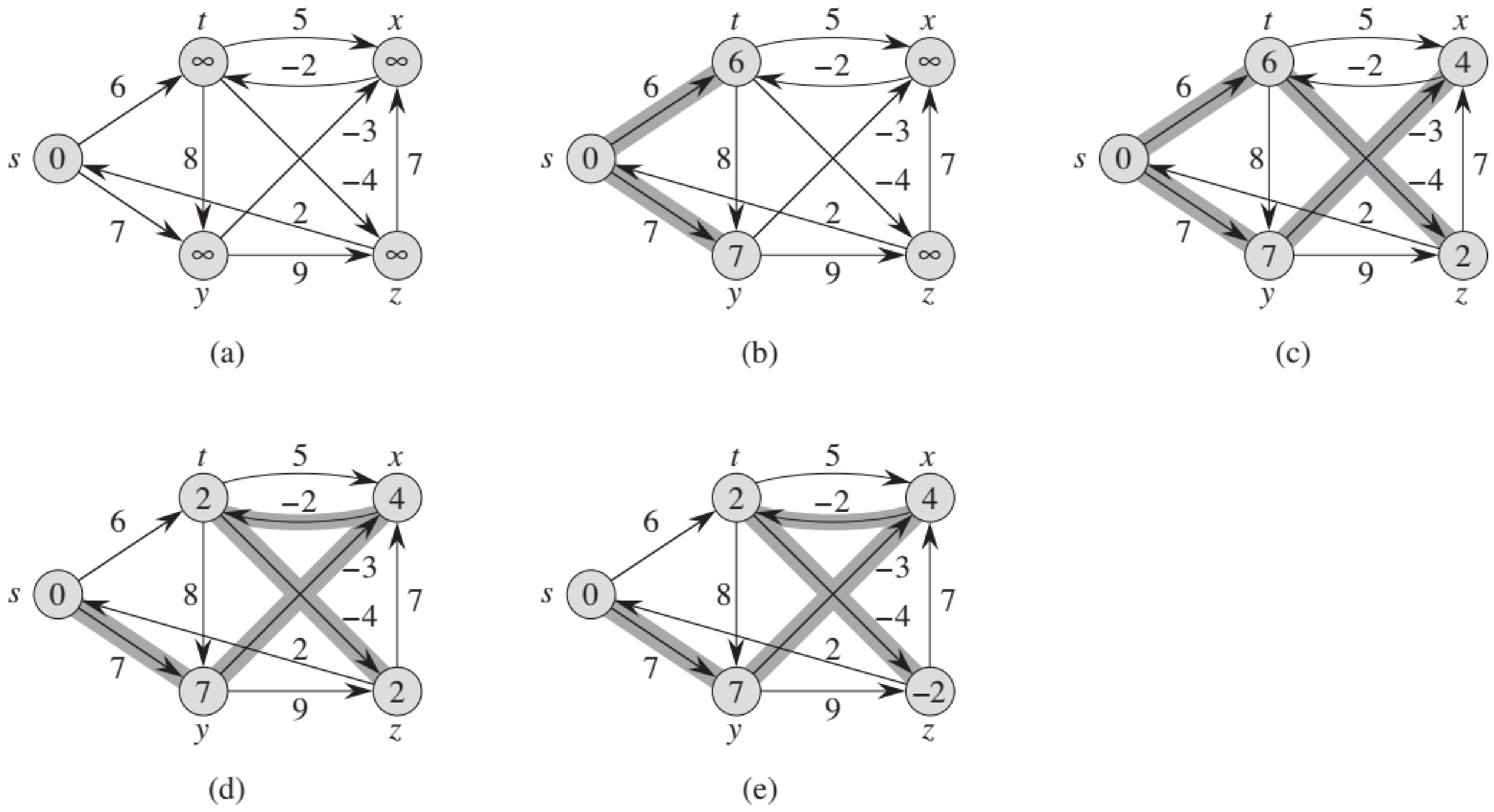
The Bellman-Ford algorithm

```
BELLMAN-FORD(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| − 1
        for each edge (u, v) ∈ G.E
            RELAX(u, v, w)
    for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            return FALSE
    return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex v ∈ G.V
        v.d = ∞
        v.π = NIL
    s.d = 0
```

```
RELAX(u, v, w)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
```

The Bellman-Ford algorithm



The Bellman-Ford algorithm

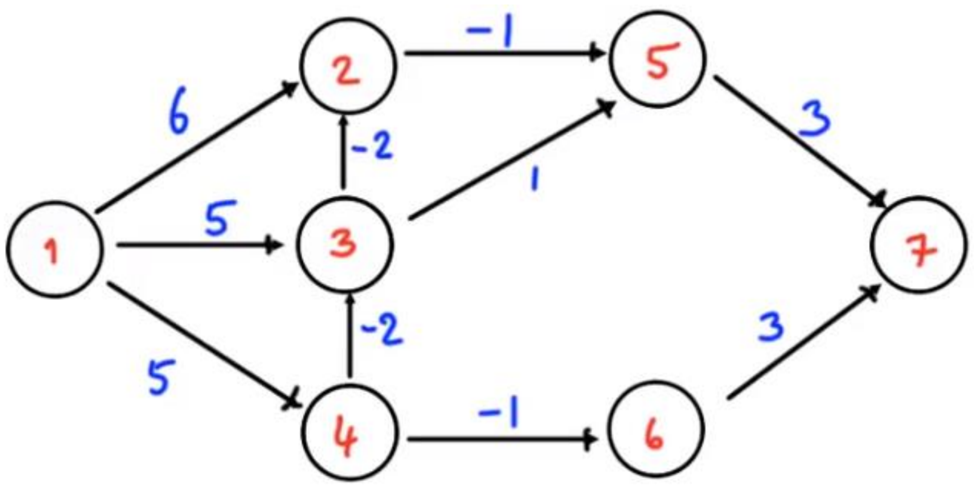
```
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++)                    // relax all E edges V-1 times
  for (int u = 0; u < V; u++)     // these two loops = O(E), overall O(VE)
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                 // record SP spanning here if needed
      dist[v.first] = min(dist[v.first], dist[u] + v.second);     // relax
    }
```

```
// after running the O(VE) Bellman Ford's algorithm shown above
  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                            // one more pass to check
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] > dist[u] + v.second)   // if this is still possible
        hasNegativeCycle = true;              // then negative cycle exists!
    }
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
```

The presence of a negative cycle causes the vertices reachable from this negative cycle to have ill-defined shortest paths information.

แบบฝึกหัด

- เขียนโปรแกรมเพื่อหา SSSP โดย Source node = 1 ของกราฟด้านล่างต่อไปนี้

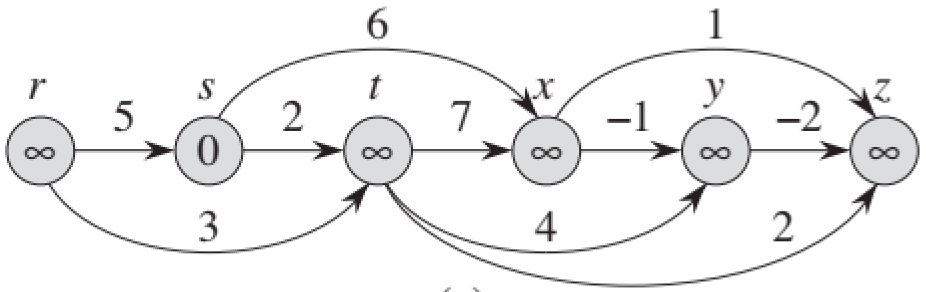


Complexity Analysis

- The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges takes $\mathrm{O}(E)$ time.

Single-source shortest paths in directed acyclic graphs

- By relaxing the edges of a weighted **dag** (directed acyclic graph) $G = (V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V + E)$ time.
- Shortest paths are always well defined in a dag, since even if there are negative-weight edges, **no negative-weight cycles can exist**.
- The algorithm starts by **topologically sorting** the dag to impose a linear ordering on



(a)

```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

All-Pairs Shortest Paths

All-Pairs Shortest Paths

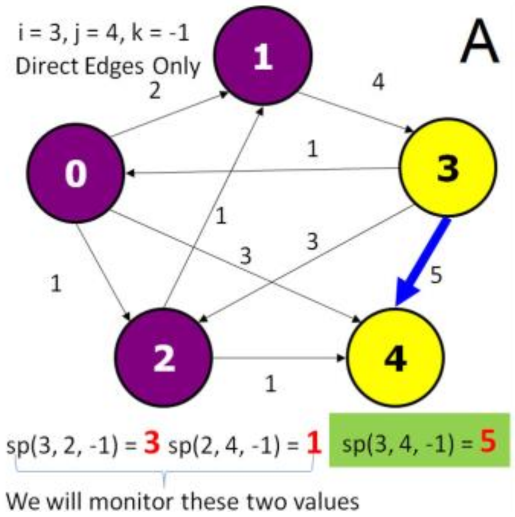
- Motivating Problems (See also UVa 11463 Commandos):
  - กำหนด Connected, weighted Graph G ที่มี Vertices V น้อยกว่าเท่ากับ 100 ให้หา Maximum value ของ dist[s][i] + dist[i][d] ทุก ๆ ค่าที่ $i \in [0 \ldots V-1]$
  - ปัญหานี้ต้องการข้อมูล Shortest paths จากทุก ๆ Sources ใน G (ซึ่งสามารถแก้ได้ด้วย Dijkstra's Algorithm)
  - แต่เราสามารถแก้ด้วยวิธีที่ง่ายกกว่านั้นคือ Floyd Warshall's algorithm

```
// inside int main()
  // precondition: AdjMat[i][j] contains the weight of edge (i, j)
  // or INF (1B) if there is no such edge
  // AdjMat is a 32-bit signed integer array
  for (int k = 0; k < V; k++)          // remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k] + AdjMat[k][j]);
```

เป็น DP ที่รันด้วย $O(V^3)$

Floyd Warshall’s algorithm

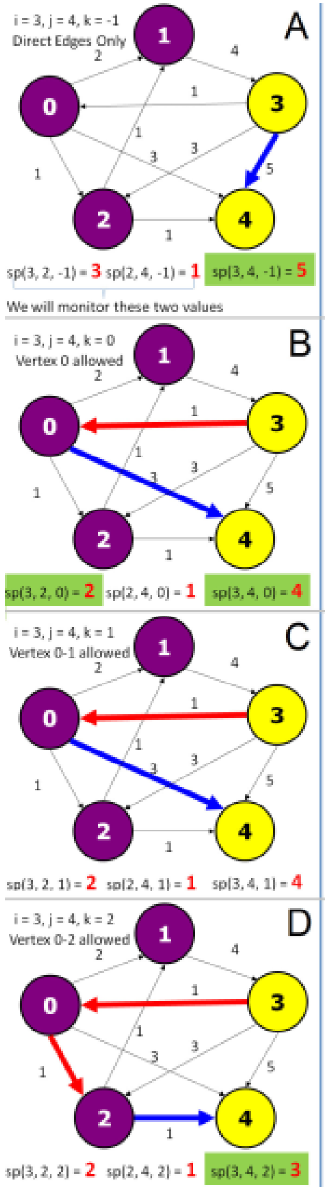
- Floyd Warshall’s ต้องใช้ Adjacency Matrix เพื่อให้การเข้าถึง Weight ของ Edge นั้นด้วย O(1)
- แนวคิดของ Floyd Warshall’s คือต้องการใช้ Intermediate vertices เพื่อสร้าง Shortest paths
  - สมมติให้ Vertices ถูก Labeled ด้วยหมายเลข [0…V-1]
  - เริ่มจาก Direct edges เท่านั้น ดังนั้น Shortest path จาก vertex i -> j คือ **sp(i, j, -1)** = weight(i,j)
  - k=-1 คือรอบแรกสำหรับ Direct edges (Initialization)
  - หลังจากนั้นเราจึงหา Shortest path ระหว่าง 2 Vertices จากตั้งแต่ [0, …, k]
  - เริ่มที่ k = 0, k=1, …, k=V-1 (ตามในลูป)



The current content of Adjacency Matrix D at **k = -1**

k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	**5**
4	∞	∞	∞	∞	0

Floyd Warshall’s algorithm (ต่อ)



The current content of Adjacency Matrix D at **k = -1**

k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0

The current content of Adjacency Matrix D at **k = 0**

k = 0	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

The current content of Adjacency Matrix D at **k = 1**

k = 1	0	1	2	3	4
0	0	2	1	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

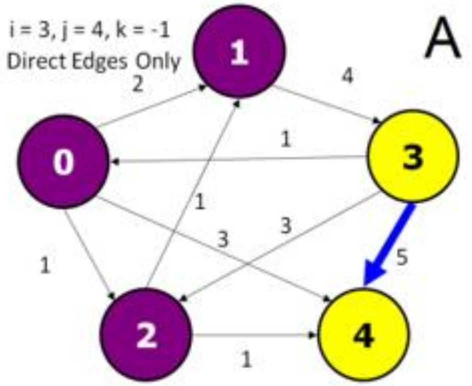
The current content of Adjacency Matrix D at **k = 2**

k = 2	0	1	2	3	4
0	0	2	1	6	2
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

- ในรูป Shortest path เราต้องการหา Final answer เราต้องการหา sp(3,4,4)=3; k=4; (3->0->2->4)
- หากพิจารณา current state ในรอบ k=-1 จะได้ว่า sp(3,4,-1) = 5 (ยังไม่ใช่ Optimal solution)
- เรารู้ว่า Solution สุดท้ายมันต้องมาจาก sp(3,2,2)+sp(2,4,2) = 3 แต่ ณ ปัจจุบันเรามีเพียงแค่ sp(3,2,-1)+sp(2,4,-1) = 4 (k=-1 คือรอบ direct edges)
- เมื่อเรา Update k=0; คราวนี้ vertex 0 **ถูกเห็น** และพร้อมที่จะเป็น Intermediate แล้วดังนั้น sp(3,4,0) จึงอัปเดตเปน sp(3,4,0) = sp(3,0,-1) + sp(0,4,-1) = 1+3 = 4 (ดูในรูป B)

แบบฝึกหัด

- ให้นักเรียนเขียนโปรแกรมเพื่อแก้ปัญหา All-Pairs Shortest Paths โดยใช้
  - Floyd Warshall’s Algorithm



สรุป

Graph Criteria	BFS $O(V+E)$	Dijkstra's variant $O((V+E)\log V)$	Bellman Ford's $O(VE)$	Floyd Warshall's $O(V^3)$
Max Graph Size	$V, E \leq 1M$	$V, E \leq 50K$	$VE \leq 1M$	$V \leq 200$
Unweighted	Best	Ok	Bad	Bad
Weighted	WA	Best	Ok	Bad
Negative weight	WA	Our variant Ok	Ok	Bad
Negative cycle	Cannot detect	Cannot detect	Can detect	Can detect
Small graph	WA if weighted	Overkill	Overkill	Best

WA = Wrong Answer

Union-Find Disjoint Sets

Union-Find Disjoint Sets

- The Union-Find Disjoint Set (UFDS) เป็นโครงสร้างข้อมูลที่โมเดล Collection of disjoint sets ที่มีความสามารถ (ใน O(1))
  - ในการระบุว่า Item หนึ่งเป็นสมาชิกของ Set ใด หรือ 2 Items อยู่ใน Set เดียวกันหรือไม่
  - ในการรวม 2 Sets เป็นเซตที่ใหญ่กว่า
- UFDS เป็นโครงสร้างข้อมูลที่สามารถใช้แก้ปัญหาการหา Connected components ใน Undirected graph ได้

Union-Find Disjoint Sets

- แนวคิดหลักของ UFDS คือการเลือก Parent item เพื่อแทนเซตแต่ละเซต (เป็น Identifier ของเซต)
  - UFDS จะสร้างโครงสร้างป่า (Forest) โดยแต่ละเซตจะหมายถึง ต้นไม้แต่ละต้นในป่า
  - Root ของต้นไม้แต่ละต้นคือ Representative ของ Set ดังนั้นการดำเนินการที่ต้องการระบุว่า Item หนึ่งเป็นสมาชิกของเซตใด ให้ทำการไต่ขึ้นไปจนถึง Root

Union-Find Disjoint Sets

- การเก็บข้อมูลของ UFDS
  - Index ของ parent item ใน `vi p`
  - ความสูงของต้นไม้ `vi rank`
  - **`vi`** คือ **`vector<int>`**
  - `p[i]` stores the immediate parent of item `i`.
  - ถ้า i เป็น Representative ของ Set จะได้ว่า `p[i] = i`

Union-Find Disjoint Sets

- การรวม 2 Disjoint sets
  - เริ่มต้นแต่ละ item จะถือเป็น disjoint set ที่มี rank = 0
  - เพื่อที่จะรวม 2 Disjoint sets ให้เป็นเซตเดียวกัน เราจะทำให้ item หนึ่งเป็น parent ใหม่ของอีก disjoint set หนึ่ง
  - **unionSet(i, j)** will cause both items 'i' and 'j' to have the same representative item—directly or indirectly.
  - For efficiency, we can use the information contained in vi rank to set the representative item of the disjoint set with higher rank to be the new parent of the disjoint set with lower rank, thereby minimizing the rank of the resulting tree.

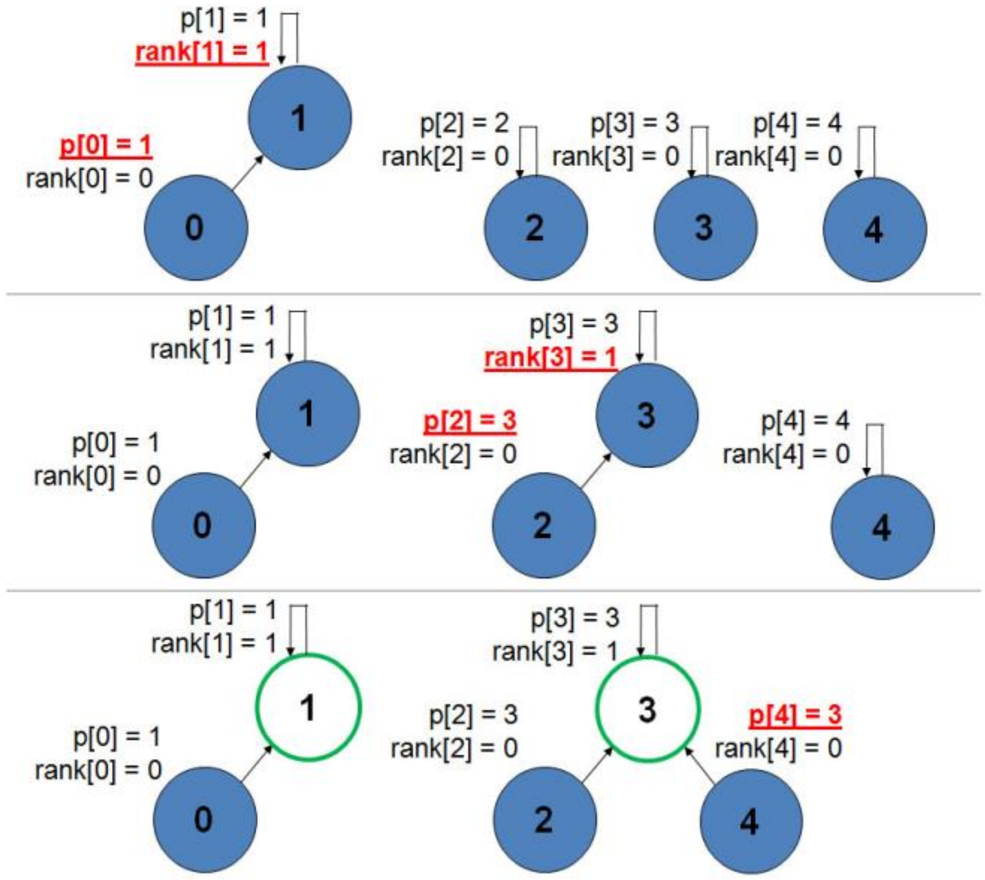


Figure 2.6: unionSet(0, 1) → (2, 3) → (4, 3) and isSameSet(0, 4)

Union-Find Disjoint Sets

- `findSet(p[i])` จะทำงานแบบ Recursive เพื่อหา Repetitive item ของเซต
  - จะทำงานด้วยการเรียก `findSet(p[i])` หาก p[i] != i และคืน i หาก p[i] = i
- `isSameSet(i, j)` จะเปรียบเทียบ findSet(i) และ findSet(j) ว่ามี Root เดียวกันหรือไม่
  - ถ้าใช้แปลว่า items i และ j อยู่ใน Set เดียวกัน

C++ Implementation for Union-find Data Structure

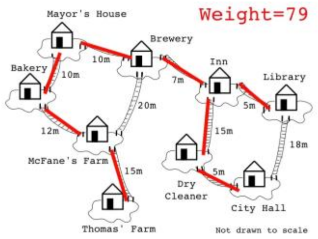
```
class UnionFind {                                        // OOP style
private: vi p, rank;                    // remember: vi is vector<int>
public:
  UnionFind(int N) { rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {                      // if from different set
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) p[y] = x;         // rank keeps the tree short
      else {                 p[x] = y;
                             if (rank[x] == rank[y]) rank[y]++; }
} } };
```

แบบฝึกหัด

- Given 8 disjoint sets: {0, 1, 2, . . . , 7}, please create a sequence of unionSet(i, j) operations to create a tree with rank = 3

Minimum Spanning Trees

Minimum Spanning Trees

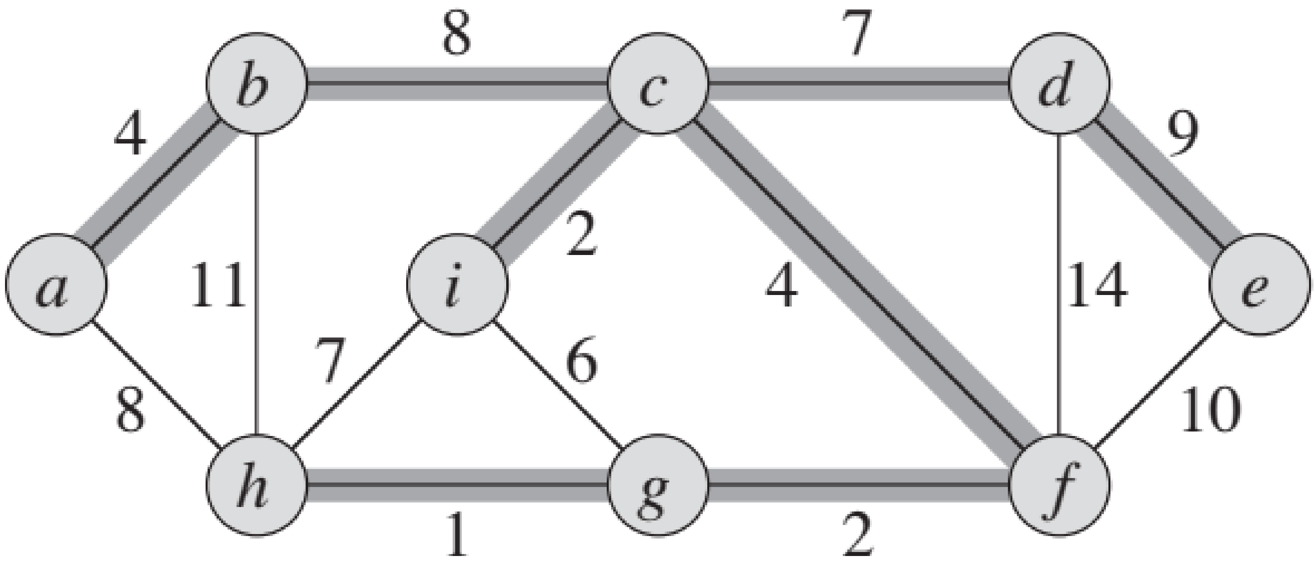


- Given a connected, undirected, and weighted graph G select a subset of edges $E' \in G$ such that the graph G is (still) connected and the total weight of the selected edges E' is **minimal**!
- Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
- To interconnect a set of n pins, we can use an arrangement of $n-1$ wires, each connecting two pins.
- Of all such arrangements, the one that uses the **least amount of wire** is usually the most desirable.

Minimum Spanning Trees

- We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the **set of pins**. E is the set of **possible interconnections between pairs of pins**, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.
- In other words, We then wish to find an acyclic subset $T \subseteq E$, that connects **all of the vertices** and whose total weight $w(T) = \sum_{(u,v)} T$
- Since T is **acyclic** and **connects** all of the vertices, it must form a tree, which we call a spanning tree since it “spans” the graph G.
- We call the problem of determining the tree T the **minimum-spanning-tree** problem.

Minimum Spanning Trees



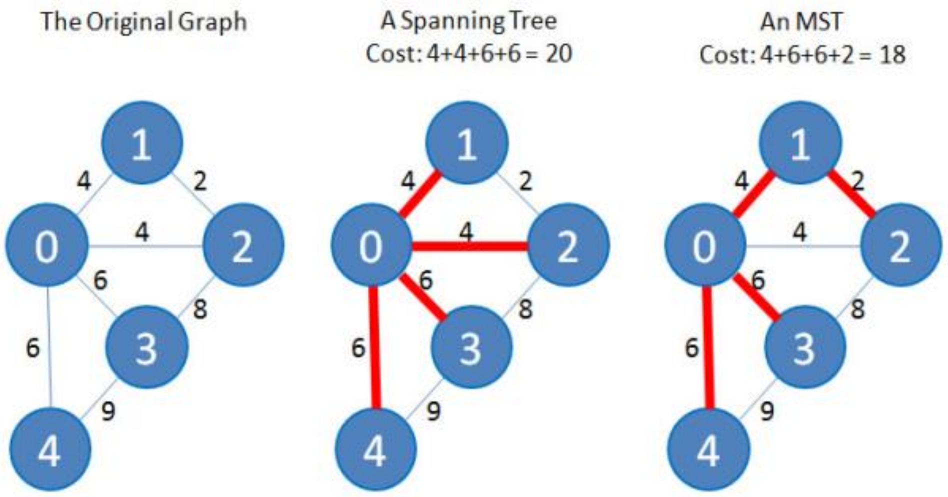


Figure 4.10: Example of an MST Problem

Minimum Spanning Trees

- In this lecture, we shall examine two algorithms for solving the minimum spanning-tree problem: **Kruskal**'s algorithm and **Prim**'s algorithm.
- We can easily make each of them run in time $O(E \lg V)$ using ordinary binary heaps.

Growing a minimum spanning tree

Growing a minimum spanning tree

- Assume that we have a *connected, undirected* graph $G = (V, E)$ with a weight function $w: E \rightarrow R$, and we wish to find a **minimum spanning tree** for G.
- Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, we determine an edge (u, v) that we can add to A without violating this invariant, so $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.
  - We call such an edge a safe edge for A, since we can add it safely to A while maintaining the invariant.

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

We use the loop invariant as follows:

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

Termination: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

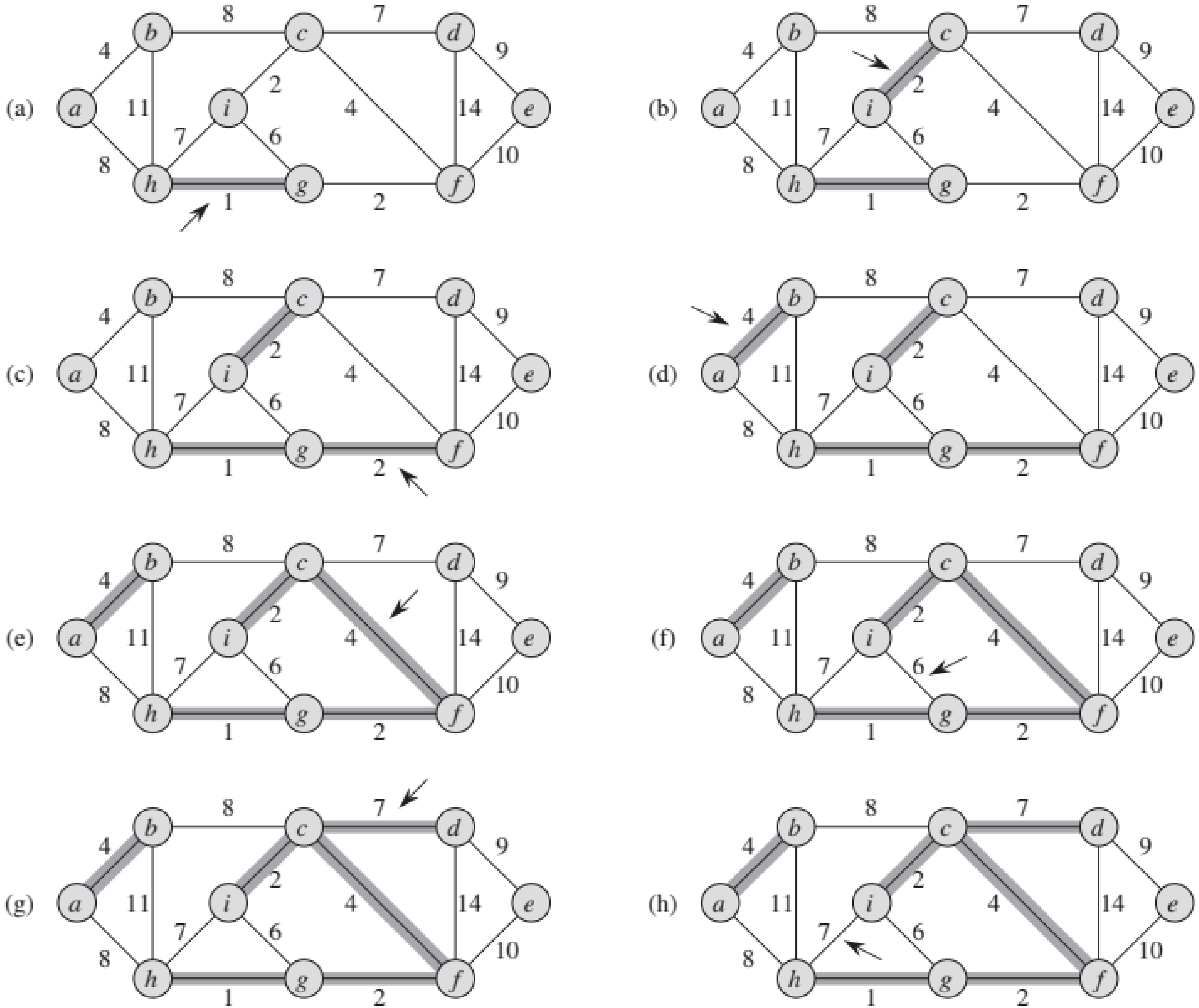
Kruskal’s algorithm

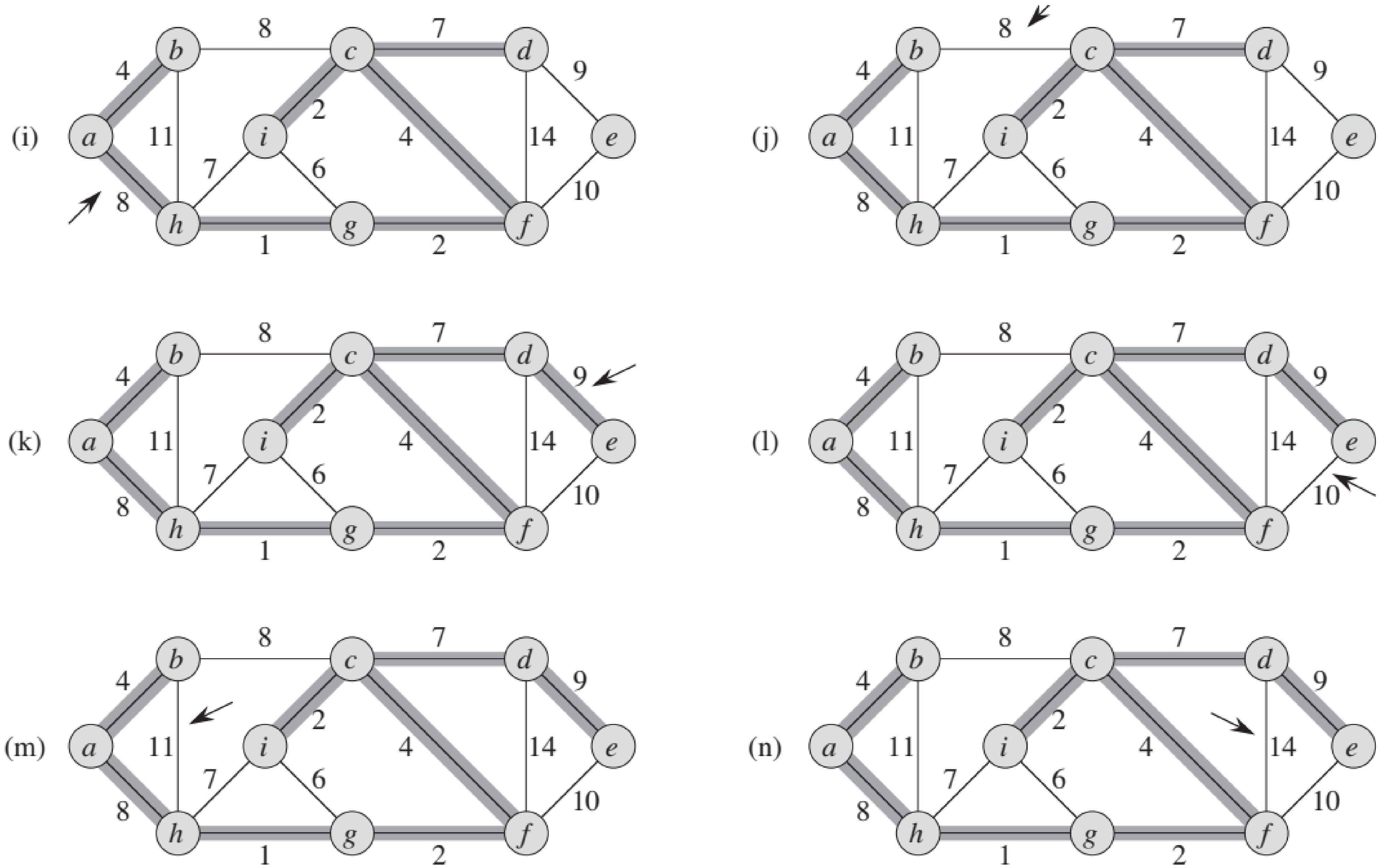
Kruskal's algorithm

- Kruskal's algorithm finds a safe edge to add to the growing forest by finding, **of all the edges that connect any two trees in the forest**, an edge (u, v) of least weight.
- Let C_1 and C_2 denote the two trees that are connected by (u, v).
- Since (u, v) must be a **light edge** connecting C_1 to some other tree, the corollary introduced previously (u, v) is a safe edge for C_1
- Kruskal's algorithm qualifies as a **greedy algorithm** because at each step it adds to the forest an edge of least possible weight.

Kruskal's algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

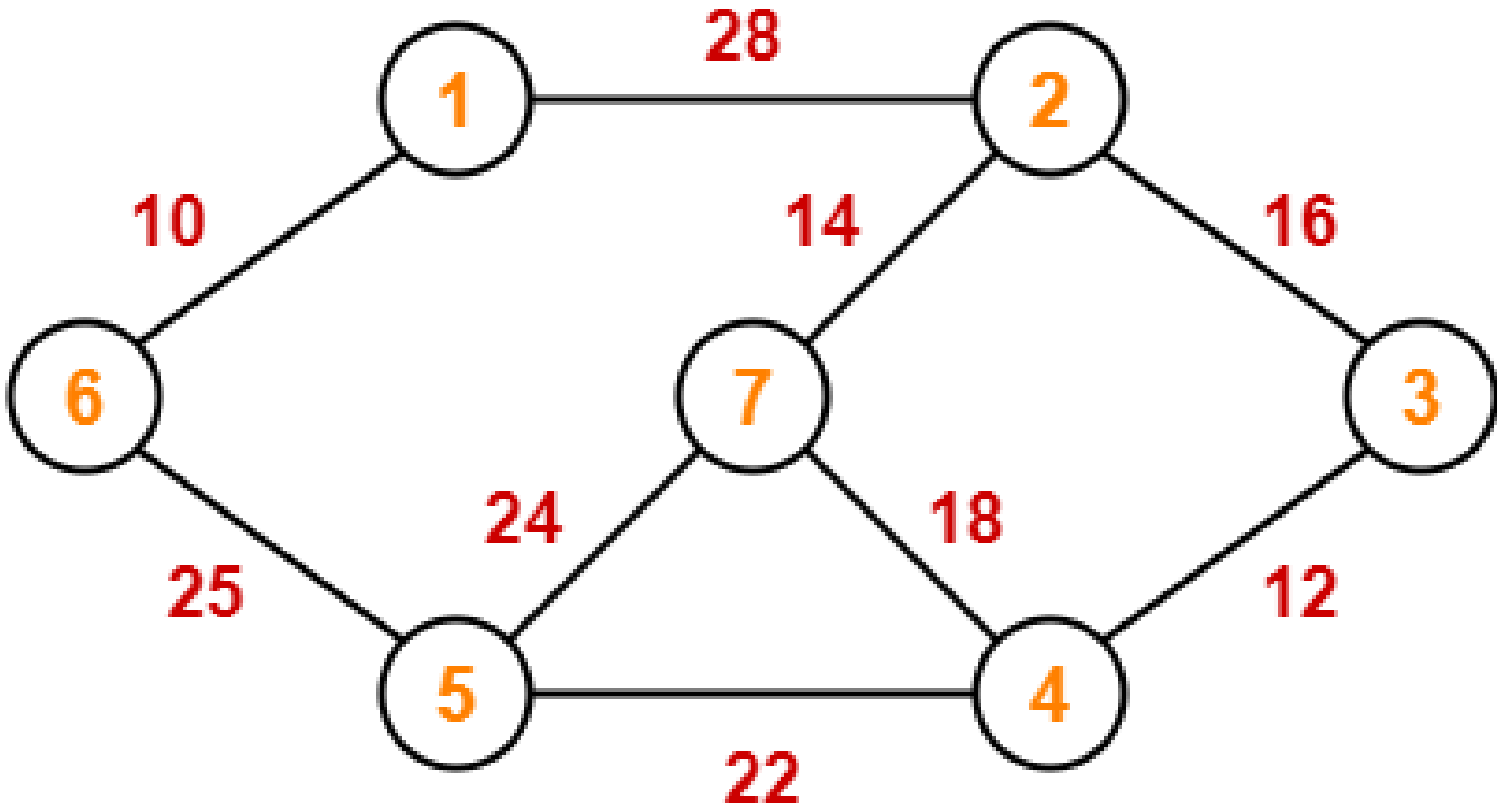

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)


(i)
(j)
(k)
(l)
(m)
(n)
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

Complexity Analysis

- The running time of Kruskal's algorithm is $O(E \lg V)$.

แบบฝึกหัด



Using the Kruskal's algorithm to find The minimum spanning tree.

C++ Implementation for Kruskal's algorithm

```
// inside int main()
  vector< pair<int, ii> > EdgeList;   // (weight, two vertices) of the edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);            // read the triple: (u, v, w)
    EdgeList.push_back(make_pair(w, ii(u, v))); }                // (w, u, v)
  sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)
                      // note: pair object has built-in comparison function
  int mst_cost = 0;
  UnionFind UF(V);                              // all V are disjoint sets initially
  for (int i = 0; i < E; i++) {                            // for each edge, O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)) {  // check
      mst_cost += front.first;                      // add the weight of e to MST
      UF.unionSet(front.second.first, front.second.second);     // link them
  } }                         // note: the runtime cost of UFDS is very light
  // note: the number of disjoint sets must eventually be 1 for a valid MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);
```

Prim’s algorithm

Prim's algorithm

- Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method from Section the previous slide.
- Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph.
- Prim's algorithm has the property that the edges in the set A always form a **single tree**.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V.

Prim’s algorithm

MST-PRIM(G, w, r)

```
1   for each u ∈ G.V
2       u.key = ∞
3       u.π = NIL
4   r.key = 0
5   Q = G.V
6   while Q ≠ ∅
7       u = EXTRACT-MIN(Q)
8       for each v ∈ G.Adj[u]
9           if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

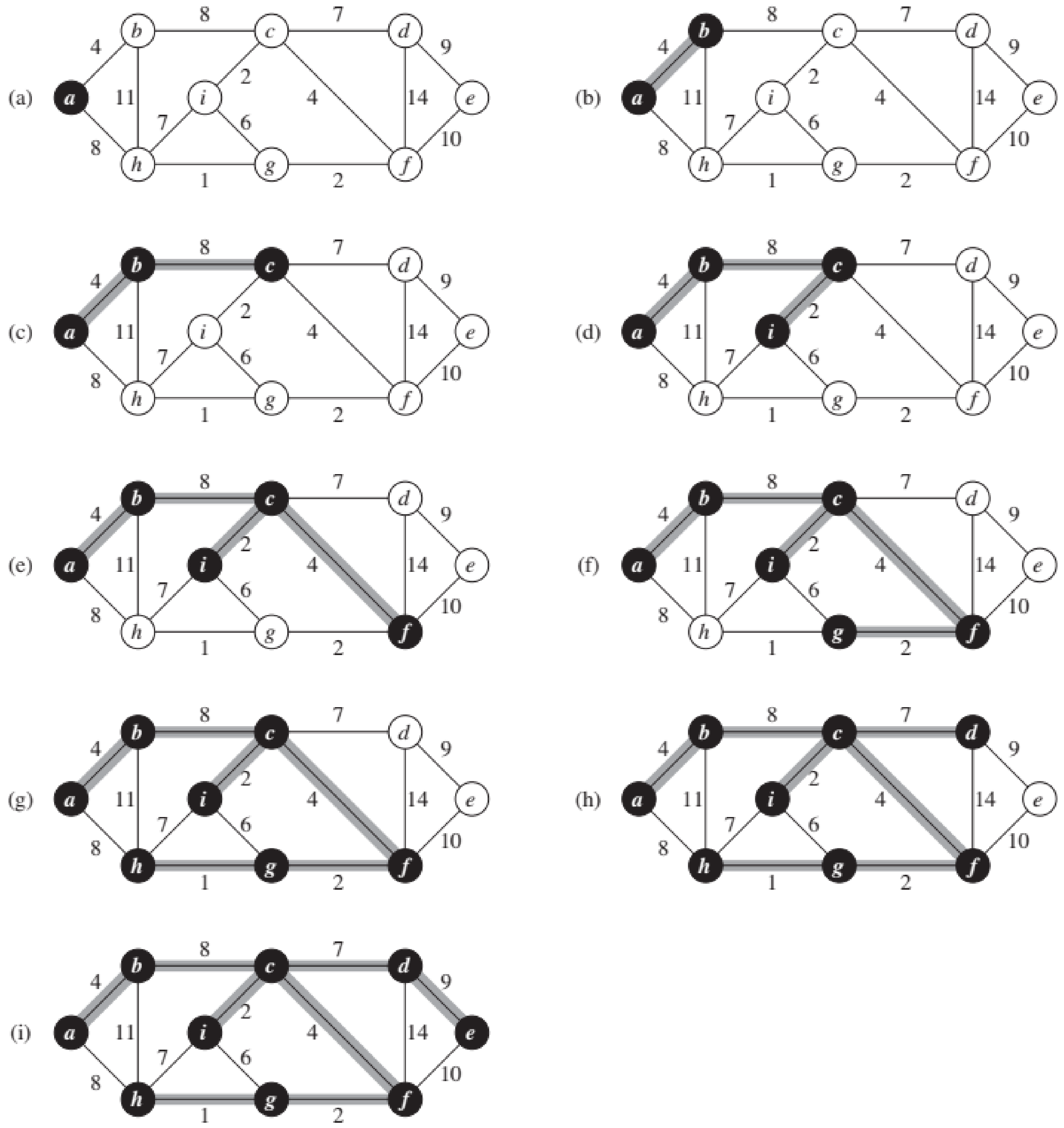
Prior to each iteration of the **while** loop of lines 6–11,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq$ NIL, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus

$$A = \{(v, v, \pi) : v \in V - \{r\}\}$$

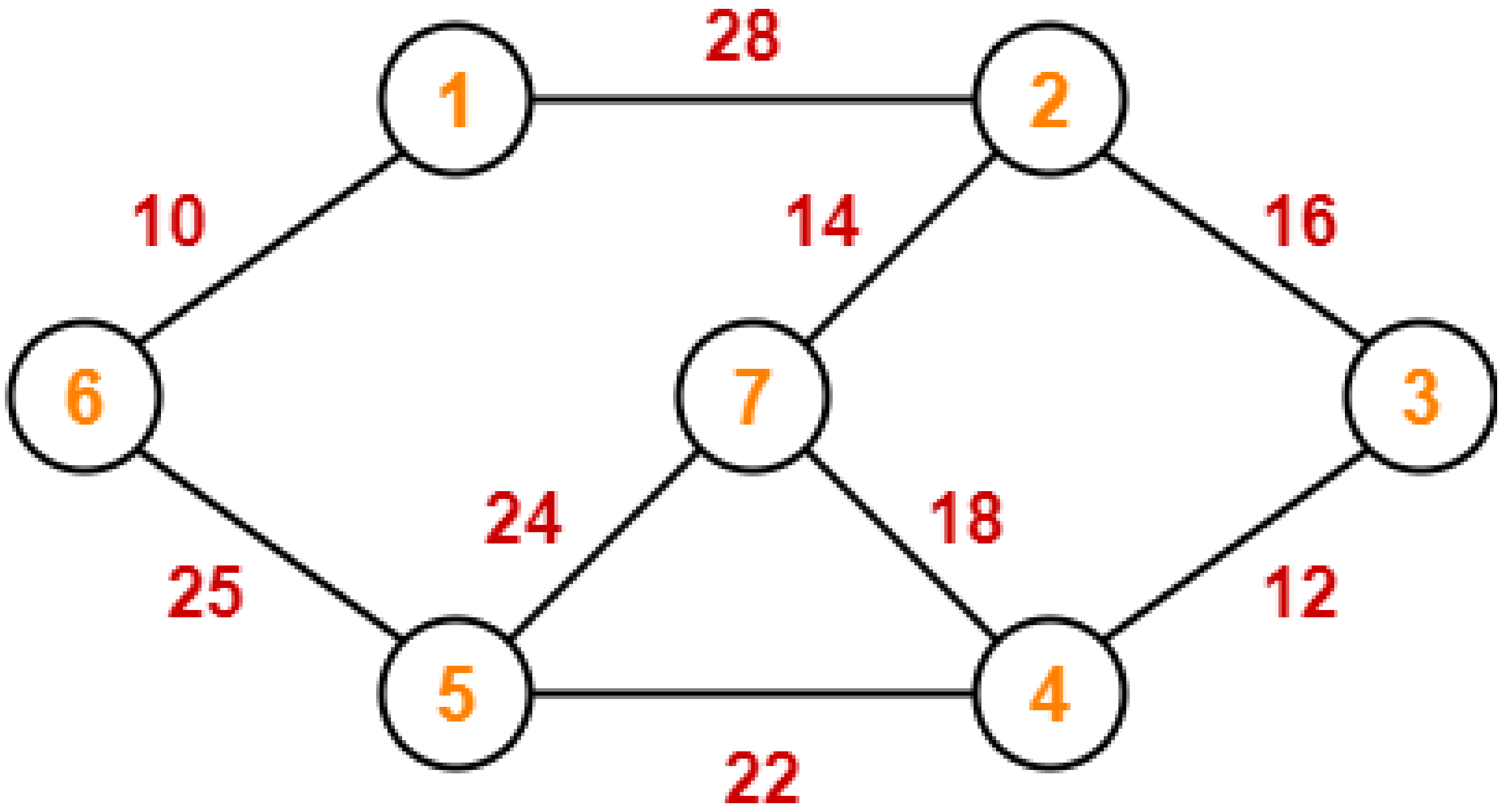
Example



Complexity Analysis

- The running time of Prim's algorithm is $O(E \lg V)$. (Same as Kruskal's algorithm)

แบบฝึกหัด



Using the Prim's algorithm to find The minimum spanning tree.

```cpp
vi taken;                                   // global boolean flag to avoid cycle
priority_queue<ii> pq;           // priority queue to help choose shorter edges
          // note: default setting for C++ STL priority_queue is a max heap
void process(int vtx) {    // so, we use -ve sign to reverse the sort order
  taken[vtx] = 1;
  for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
} }                                  // sort by (inc) weight then by (inc) id

// inside int main()---assume the graph is stored in AdjList, pq is empty
  taken.assign(V, 0);                         // no vertex is taken at the beginning
  process(0);   // take vertex 0 and process all edges incident to vertex 0
  mst_cost = 0;
  while (!pq.empty()) {  // repeat until V vertices (E=V-1 edges) are taken
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;  // negate the id and weight again
    if (!taken[u])                       // we have not connected this vertex yet
      mst_cost += w, process(u); // take u, process all edges incident to u
  }                                               // each edge is in pq only once!
  printf("MST cost = %d (Prim's)\n", mst_cost);
```

ch4 03 kruskal prim.cpp

แบบฝึกหัด

- ให้นักเรียนเขียนโปรแกรมเพื่อแก้ไขปัญหา Minimum Spanning Tree (MST) โดย
  - Kruskal’s algorithm หรือ Prim’s algorithm (MST.CPP)

Notes

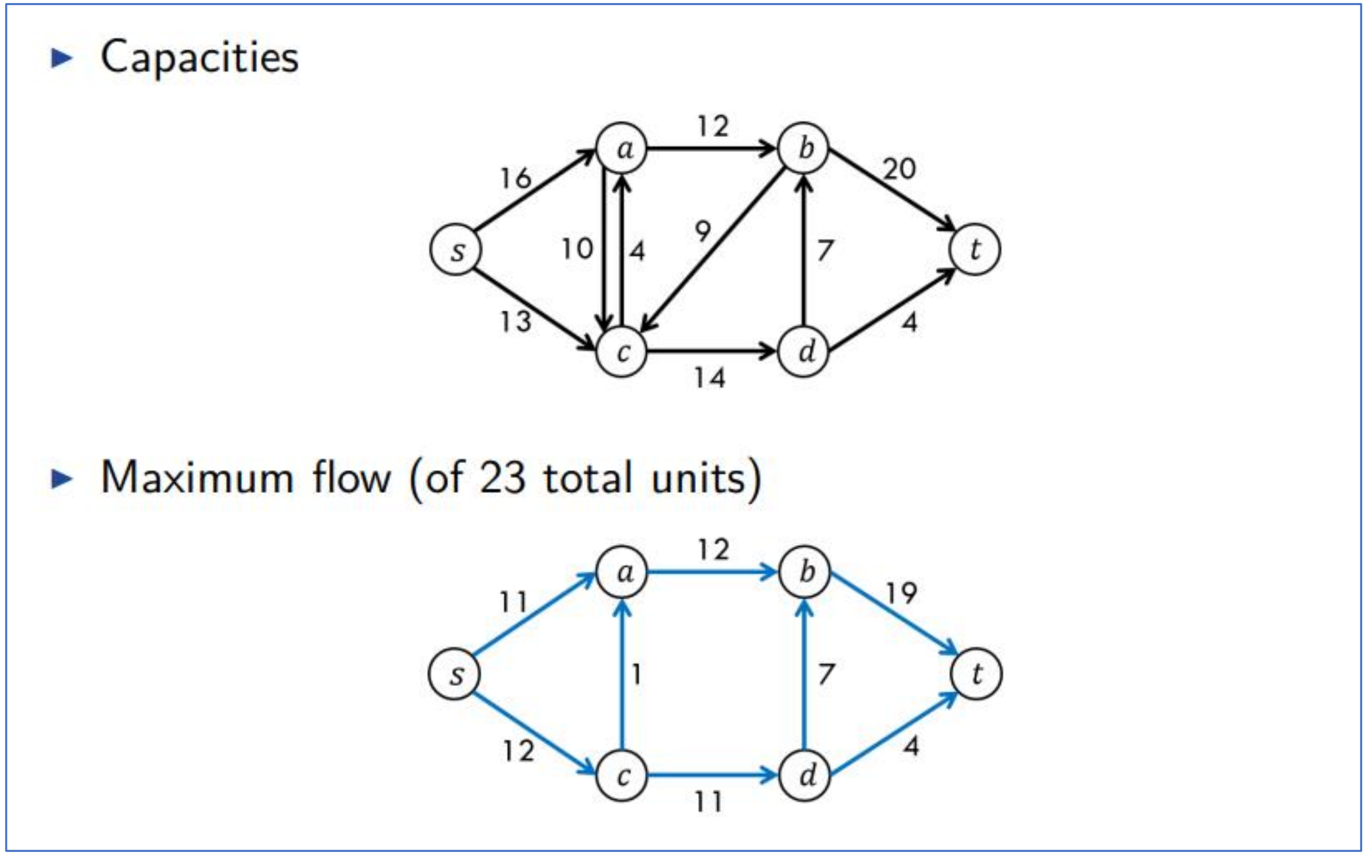
- แนะนำให้ใช้ Kruskal’s algorithm ในการหา MST
- ลองพิจารณา **<u>Max</u>**imum Spanning Tree

The Other Interesting Graph Problems

Not fully covered here.

Network Flow (Max Flow)

- กำหนดให้ Directed, weighted, directed graph คือเครือข่ายท่อน้ำ (Pipe network) โดยที่ท่อถูกแทนด้วย Edges และจุดแยก (Splitting points) คือ Vertices
- น้ำหนักของ Edges คือ Capacities ของท่อน้ำ
- โดยท่อน้ำมี 2 ชนิดคือ
  - Source s และ Sink t
- **โจทย์:** การไหลที่มากที่สุด (Maximum flow) จาก s -> t คือเท่าใด
- Algorithms:
  - Ford Fulkerson’s Method
  - Edmonds Karp’s



From: Network Flow Problems Jaehyun Park CS 97SI Stanford University

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.
- https://www.otfried.org/courses/cs500/slides-approx.pdf
- Competitive Programming 3
- https://web.stanford.edu/class/cs97si/08-network-flow-problems.pdf