Graph Algorithm:
Data Structure, Algorithms, and Applications

ACM-ICPC Preparing Session at Mahidol

ภิญโญ แท้ประสาทสิทธิ์

ภาควิชาคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยศิลปากร

(pinyotae at gmail dot com; pinyo at su.ac.th)

หัวข้อเนื้อหา

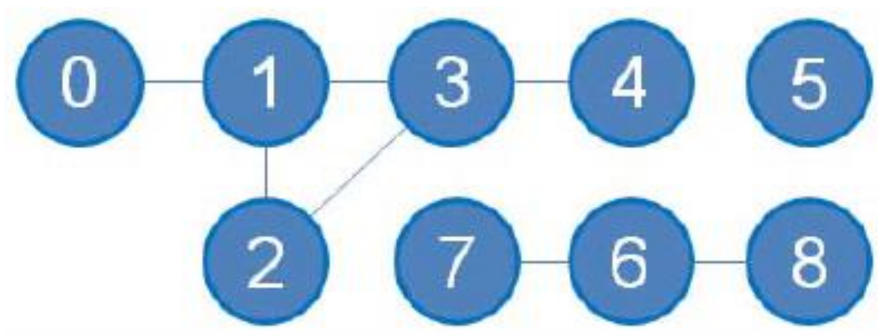
- รู้จักกราฟ
- การอธิบายโครงสร้างของกราฟ
- พื้นฐานการจัดการและใช้โครงสร้างข้อมูลกราฟ
- อัลกอริทึมเกี่ยวกับกราฟ
  - Bread-First Search, Depth-First Search, Topological Sort
  - Minimum Spanning Tree
  - Shortest Path Algorithm
  - Network Flow
- ตัวอย่างการประยุกต์ใช้ที่น่าสนใจ

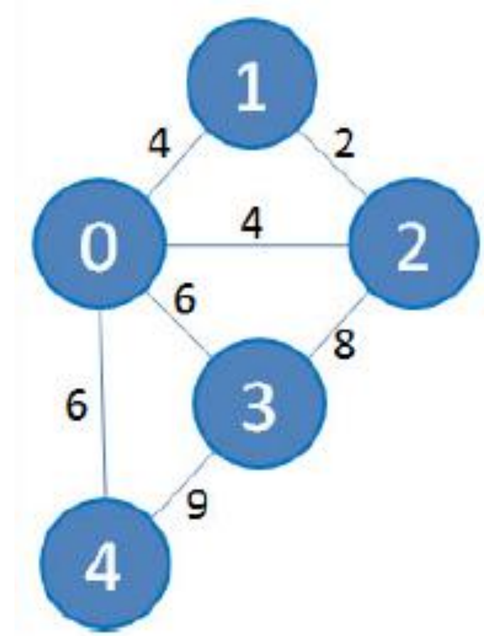
รู้จักกับกราฟ

- กราฟเป็นโครงสร้างข้อมูลแบบหนึ่ง
- โดยตัวของมันเองไม่ใช่อัลกอริทึม ซึ่งจุดนี้จะต่างกับ ...
  - Greedy Algorithm
  - Divide and Conquer
  - Dynamic Programming
- โครงสร้างข้อมูลของกราฟมักจะประกอบด้วยของสองส่วนหลักคือ
  - โหนด / จุดยอด (Node / Vertex)
  - เส้นเชื่อม / ขอบ (Edge)
    - เส้นเชื่อมอาจจะมีทิศทาง (directional edge)
    - เส้นเชื่อมอาจจะมีน้ำหนัก (weight) หรือค่าใช้จ่าย (cost) กำหนดไว้ด้วย

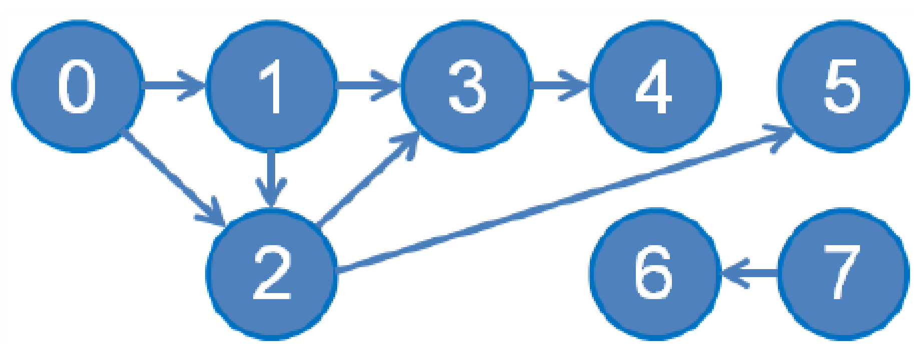
มโนภาพของโครงสร้างข้อมูลแบบกราฟ

- ตัวอย่างกราฟที่เส้นเชื่อมไม่มีทิศทาง





- ตัวอย่างกราฟที่เส้นเชื่อมมีทิศทางกำหนด



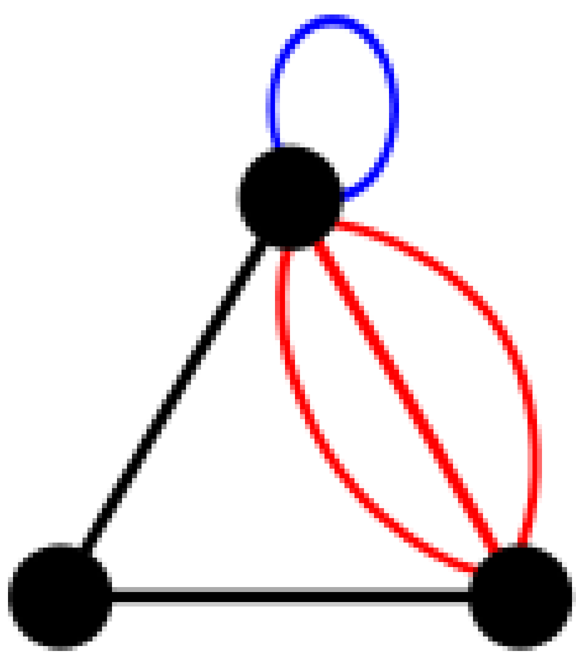
ภาพจาก Competitive Programming 2nd Edition

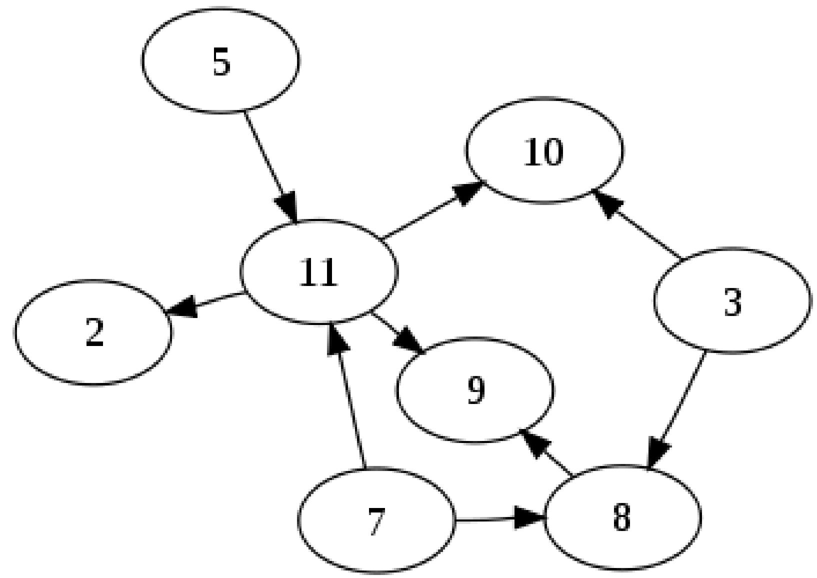
ประเภทของกราฟ

- เราสามารถแบ่งกราฟได้เป็นหลายประเภท
  - ขึ้นอยู่กับมุมมองที่เราเลือกใช้สำหรับการแบ่งประเภท
  - แต่ในระดับพื้นฐาน เรามักแบ่งตามลักษณะของเส้นเชื่อม
    - ➢ ถ้าเส้นเชื่อมมีทิศทางกำหนดเรามักเรียกกราฟว่ากราฟมีทิศทาง (directional graph) ไม่เช่นนั้นจะเรียกว่ากราฟไม่มีทิศทาง (undirectional graph)
- แต่วิธีแบ่งประเภทกราฟก็มีอีกหลากหลาย เช่น
  - ถ้ากราฟมีทิศทาง เราก็อาจแบ่งว่ามีวังวน (cycle) ในกราฟหรือไม่ ถ้าไม่มีเราเรียกว่า Directional Acyclic Graph (DAG)
  - โหนดคู่ใด ๆ ในกราฟมีเส้นเชื่อมได้มากกว่า 1 เส้นหรือไม่ ถ้ามีเราเรียกว่า Multigraph

ภาพตัวอย่างกราฟแบบต่าง ๆ

- กราฟบางแบบก็มีเส้นเชื่อมที่วกเข้าหาตัวเอง (self loop) [เส้นเชื่อมสีน้ำเงิน]
- ชุดเส้นเชื่อมที่ทำให้เป็น multigraph คือเส้นเชื่อมสีแดง





Directed Acyclic Graph (DAG)

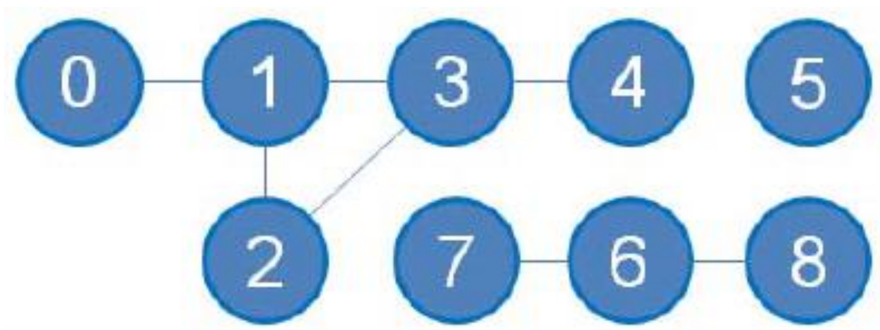
ภาพจาก wikipedia

การอธิบายโครงสร้างกราฟ

- เราสามารถแบ่งกราฟได้เป็นหลายประเภท
  - ขึ้นอยู่กับมุมมองที่เราเลือกใช้สำหรับการแบ่งประเภท
  - แต่ในระดับพื้นฐาน เรามักแบ่งตามลักษณะของเส้นเชื่อม
    - ถ้าเส้นเชื่อมมีทิศทางกำหนดเรามักเรียกกราฟว่ากราฟมีทิศทาง (directional graph) ไม่เช่นนั้นจะเรียกว่ากราฟไม่มีทิศทาง (undirectional graph)
- แต่วิธีแบ่งประเภทกราฟก็มีอีกหลากหลาย เช่น
  - ถ้ากราฟมีทิศทาง เราก็อาจแบ่งว่ามีวังวน (cycle) ในกราฟหรือไม่ ถ้าไม่มีเราเรียกว่า Directional Acyclic Graph (DAG)
  - โหนดคู่ใด ๆ ในกราฟมีเส้นเชื่อมได้มากกว่า 1 เส้นหรือไม่ ถ้ามีเราเรียกว่า Multigraph
- เรามักแทนเซ็ตของโหนดด้วย V และเซ็ตของเส้นเชื่อมด้วย E

การอธิบายโครงสร้างกราฟ

- กราฟเป็นโครงสร้างข้อมูลที่มีของสามอย่างเกี่ยวข้องด้วยบ่อย ๆ
  - โหนด, เส้นเชื่อม และ ค่าน้ำหนักของเส้นเชื่อม (ตัวท้ายสุดอาจไม่จำเป็น)
- เรามักจะระบุด้วยว่าเส้นเชื่อมมีทิศทางหรือไม่
- เรามักระบุการเชื่อมต่อกันของโหนดต่าง ๆ เช่น



Edge listing

```
0 1
1 3
1 2
2 3
3 4
6 7
6 8
```

Adjacency listing

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

ภาพจาก Competitive Programming 2^{nd} Edition

เรื่องที่ต้องคิดในทางปฏิบัติ

- เราเห็นการอธิบายกราฟด้วย Edge listing กับ Adjacency list มาแล้ว
  - แต่คำถามก็คือว่าเราจะเก็บข้อมูลพวกนี้ในหน่วยความจำอย่างไร (How to keep graph description in memory?)
  - แล้วเราจะทำอะไรกับข้อมูลพวกนี้บ้าง (What operations are we going to do with a graph?)
- ธรรมชาติของโครงสร้างข้อมูลที่ดีก็คือว่า มันต้องสามารถทำในสิ่งที่เราต้องการทำบ่อย ๆ ได้อย่างมีประสิทธิภาพ
  - เรามักถามว่าโหนดหมายเลข x มีเส้นเชื่อมต่อกับโหนดหมายเลข y หรือไม่
  - เรามักถามว่ามีเส้นทางจาก โหนดหมายเลข x ไปโหนดหมายเลข y หรือไม่

เปรียบเทียบการใช้ Edge listing กับ Adjacency list

- ถ้าเราถามว่าโหนดหมายเลข x มีเส้นเชื่อมต่อกับโหนดหมายเลข y หรือไม่
  - Edge listing: เราต้องวิ่งไปทั้งลิสต์เพื่อหาว่ามีคู่เส้นเชื่อม (x, y) หรือ (y, x) หรือไม่
    - จากตัวอย่าง เรามีเส้นเชื่อมทั้งหมด 7 เส้น ถ้าถามแบบนี้ 100 รอบ ในกรณีที่เลวร้ายที่สุดก็ต้องวิ่งหาเส้นเชื่อมทั้งหมด 700 ครั้ง
  - Adjacency list: เราสร้างลิสต์แยกของแต่ละโหนดออกมา แล้วตรวจเฉพาะลิสต์ของโหนดที่สนใจ เช่นถ้า x = 1 และ y = 2
    - เราก็ตรวจเฉพาะลิสต์ของโหนด 1 หรือโหนด 2 อย่างใดอย่างหนึ่ง
    - ถ้าเราเลือกตรวจลิสต์โหนด 1 เราก็จะใช้เวลาตรวจรอบละ 3 ครั้ง ถ้าต้องตรวจ 100 รอบ ก็เสียเวลาแค่ 300 ครั้ง
- Adjacency list เร็วกว่ามาก ดังนั้นเรามาดูเรื่องวิธีเก็บในหน่วยความจำ

การเก็บโครงสร้างกราฟในหน่วยความจำ

- อย่างที่บอกไว้ตั้งแต่ต้นว่ากราฟคือโครงสร้างข้อมูล

  → ถ้าเราไม่รู้ว่าจะเก็บโครงสร้างข้อมูลไว้ได้อย่างไรก็ไม่มีความหมาย

- สมมติว่าเรารู้จำนวนโหนดและเส้นเชื่อมของแต่ละโหนดว่าเป็นจำนวนที่แน่นอนตายตัวในกราฟที่เราสนใจ

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

  - ดังนั้นโหนดแต่ละโหนดจะมีอาเรย์ของเลขจำนวนเต็มเพื่อเก็บว่ามันเชื่อมต่อกับใครอยู่บ้าง
  - เราสามารถนำอาเรย์ของแต่ละโหนดมามัดรวมกันเป็นอาเรย์สองมิติ
    - มิติแรกระบุว่าเป็นลิสต์ของโหนดไหน
    - มิติที่สองระบุว่าโหนดที่สนใจเชื่อมกับใคร
    - ขนาดของมิติที่สองของแต่ละโหนดต่างกันได้ (แปลกใจมั้ย)

ตัวอย่างการเก็บ Adjacency list

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

```
public class GraphStructDemo {
  int[][] arNode;

  private void prepareSpace() {
    arNode = new int[9][];

    arNode[0] = new int[1];
    arNode[1] = new int[3];
    arNode[2] = new int[2];
    arNode[3] = new int[3];
    arNode[4] = new int[1];      .
                                 .
    arNode[5] = new int[0];      .
    arNode[6] = new int[2];
    arNode[7] = new int[1];
    arNode[8] = new int[1];
  }
  ..........
}
```

ประกาศอาเรย์สำหรับเก็บข้อมูลการเชื่อมต่อไว้

เริ่มสร้างอาเรย์ แต่อย่าเพิ่งรีบบอกขนาดของมิติที่สอง เก็บไว้ทำทีหลังได้ (ทำแบบนี้ได้จริง ๆ นะ)

ระบุขนาดของลิสต์การเชื่อมต่อของแต่ละโหนดทีละอันตามปริมาณที่ต้องใช้จริง

ไฟล์ GraphStructDemo.java

ป้อนข้อมูลเข้า Adjacency list

```
private void insertData() {
    arNode[0][0] = 1;
    arNode[1][0] = 0;  arNode[1][1] = 2;  arNode[1][2] = 3;
    arNode[2][0] = 1;  arNode[2][1] = 3;
    arNode[3][0] = 1;  arNode[3][1] = 2;  arNode[3][2] = 4;
    arNode[4][0] = 3;

    //arNode[5][];  // No edge to insert
    arNode[6][0] = 7;  arNode[6][1] = 8;
    arNode[7][0] = 6;
    arNode[8][0] = 6;
}
```

ตรวจว่าโหนดเชื่อมกันหรือไม่

การตรวจดูว่าโหนด x เชื่อมกับโหนด y หรือไม่

```
public boolean isLinked(int x, int y) {
  int numNodes = arNode[x].length;

  for(int i = 0; i < numNodes; ++i) {
    if(arNode[x][i] == y) {
      return true;  // y is found
    }
  }

  return false;      // y not found
}
```

เนื่องจากอาเรย์ของแต่ละโหนดมีความยาวไม่เท่ากัน เราจึงต้องหาความยาวลิสต์ของโหนดมาเก็บไว้ก่อน

จากโครงสร้างของลูป เราเห็นได้ว่าการจะตรวจการเชื่อมต่อของโหนด ในกรณีที่แย่ที่สุดเกิดขึ้นเมื่อไม่พบการเชื่อมต่อ (return false) เพราะเราจะต้องหาตั้งแต่เริ่มจนจบ

มีวิธีที่ทำให้ฟังก์ชันนี้ทำงานเร็วขึ้นหรือไม่ ?

ทำอย่างไรจึงจะหาคำตอบเกี่ยวกับการเชื่อมต่อได้เร็ว ๆ

- จากฟังก์ชัน isLinked ลูปเป็นบริเวณที่ใช้เวลาในการค้นหานานที่สุด
- แต่เราก็รู้มาก่อนแล้วว่าวิธีที่ใช้ในการค้นหาที่เร็วนั้นมีอยู่
  - เช่น **การใช้ binary search จะย่นเวลาในการหาข้อมูลในแต่ละลิสต์**เหลือเพียง $O(\log M)$ เมื่อ M คือจำนวนโหนดในลิสต์ที่เราสนใจ
  - ดังนั้นถ้าเราจัดเรียงข้อมูลในลิสต์แต่ละอันก่อน เราก็จะสามารถใช้ binary search เพื่อทำให้การหาคำตอบเกี่ยวกับการเชื่อมต่อเป็นไปอย่างรวดเร็ว
- แล้วที่จริงยังมีทางทำให้เร็วกว่า $O(\log M)$ หรือไม่
  - มีเหมือนกัน แต่เราต้องเลิกใช้ Adjacency list แล้วไปใช้ของอย่างอื่นแทน
  - มาถึงจุดนี้ หลายคนคงสังเกตแล้วว่าในขณะที่กราฟเป็นโครงสร้างข้อมูลแบบหนึ่ง การเก็บข้อมูลมันก็ทำได้หลายแบบ ไม่ได้มีแบบเดียว

การอธิบายโครงสร้างกราฟด้วย Adjacency Matrix

- ก่อนหน้านี้เราพยายามเก็บรายการเส้นเชื่อมออกมาเป็นลิสต์
  - Edge listing จะจับของทุกอย่างมาเป็นกองเดียว (ไม่นิยมอย่างแรง)
  - Adjacency list จะจัดให้เป็นหมวดหมู่ตามโหนด (นิยมพอสมควร โดยเฉพาะตอนที่ลิสต์ค่อนข้างสั้น)
- แต่เราก็สามารถเก็บข้อมูลการเชื่อมต่อได้ในรูปแบบของตาราง (ในที่นี้เรียกชื่อวิชาการว่าเป็น matrix)
  - ตารางแต่ละแถวและคอลัมน์จะแทนหมายเลขโหนด
  - ถ้ามีการเชื่อมต่อกันจริง ช่องตารางที่เป็นคู่กันตามแถวและคอลัมน์จะเป็นเลข 1 (true) ถ้าไม่เชื่อมต่อกันจะเป็นเลข 0 (false)

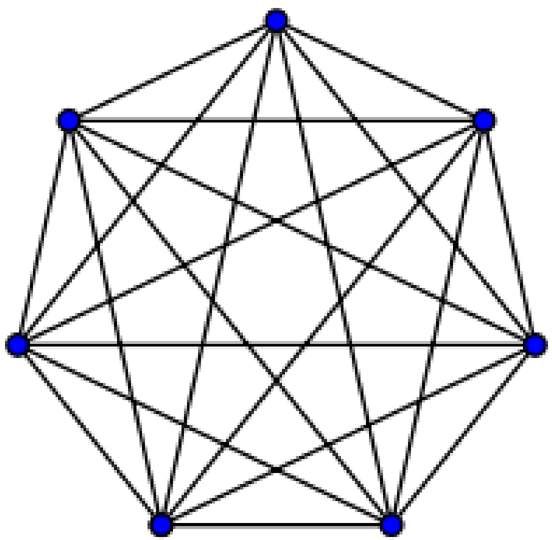
ตัวอย่าง Adjacency Matrix

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

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

หน่วยความจำที่ต้องใช้ในการอธิบายโครงสร้างกราฟ

- การใช้ Edge listing หรือ Adjacency list เหมาะกับกราฟที่มีเส้นเชื่อมน้อยเมื่อเทียบกับจำนวนโหนด (เหมาะกับ Sparse Graph)
- ลองพิจารณากราฟที่มีเส้นเชื่อมสมบูรณ์ (Complete Graph)



กราฟมี 7 โหนด 21 เส้นเชื่อม

Edge listing ใช้พื้นที่เก็บเลขจำนวนเต็ม 2 ตัวต่อเส้นเชื่อม

→ ใช้ integer 42 ตัว (168 bytes)

→ ถ้าเส้นเชื่อมมาก ปริมาณหน่วยความจำที่ต้องใช้จะเพิ่มขึ้นมากตาม

→ กราฟจำนวนมากมีปริมาณเส้นเชื่อมเป็น $O(|V|^2)$ [V คือเซ็ตของโหนด $|V|$ ก็คือจำนวนโหนดในเซ็ต]

ดังนั้นปริมาณเส้นเชื่อมและหน่วยความจำที่ต้องใช้จึงอยู่ในระดับ $O(|V|^2)$ ไปด้วย

ภาพจาก wikipedia

หน่วยความจำที่ต้องใช้ในการอธิบายโครงสร้างกราฟ (2)

- แล้วถ้าเป็น Adjacency matrix ล่ะ
  - เนื่องจากเป็นอาเรย์สองมิติ เราเห็นได้ชัดว่าใช้หน่วยความจำ $O(|V|^2)$
  - แต่ว่าเราไม่จำเป็นต้องเก็บเลขจำนวนเต็มเหมือนกับการใช้ลิสต์
  - ถ้าเราใช้ boolean เราจะเสียพื้นที่ต่อช่อง 1 ไบต์ (ถ้าเป็นจำนวนเต็มแบบลิสต์จะใช้ 4 ไบต์)
  - ถ้าเราใช้ **java.util.BitSet** เราจะใช้พื้นที่ต่อช่องแค่ 1 บิต
- ด้วยเทคนิคด้านการจัดการหน่วยความจำ ถ้ากราฟมีเส้นเชื่อมมาก ๆ การใช้ Adjacency matrix เป็นทางออกที่ดีที่สุดอย่างไม่ต้องสงสัย
  - ตอบคำถามเรื่องการเชื่อมต่อได้เร็วกว่า เข้าใจง่ายกว่า
  - ถ้าเส้นเชื่อมมากพอ มักจะใช้หน่วยความจำน้อยกว่าด้วย

สร้างกราฟด้วยวิธีอื่นได้อีกหรือไม่

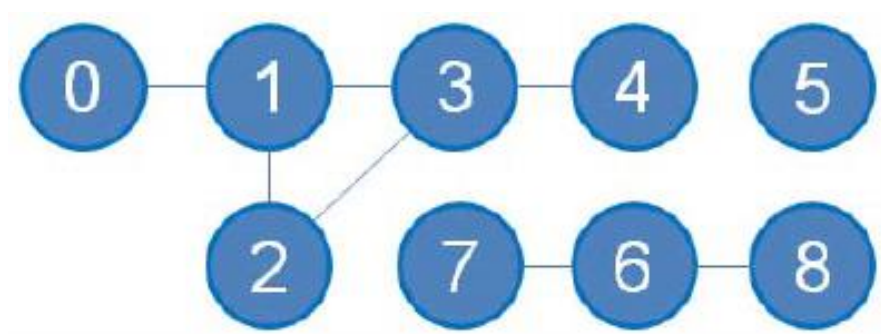
- แน่นอนว่ามีวิธีสร้างกราฟด้วยวิธีอื่น ๆ อีก
- วิธีอื่นที่นิยมก็มีจำพวกที่สร้างกราฟออกมาเป็นคลาสของโหนด
  - ภายในคลาสใช้ลิงค์ลิสต์ไปถึงกราฟที่ติดกัน
  - เนื่องจากมันติดกันได้มากกว่าหนึ่ง ลิงค์ลิสต์จึงมักอยู่ในรูปของอาเรย์
  - การนำข้อมูลการเชื่อมต่อมาเก็บไว้ในคลาสของโหนดแต่ละโหนดทำให้คนเขียนโปรแกรมไม่ต้องใช้อาเรย์สองมิติโดยตรง
  - วิธีนี้จะสอดคล้องกับมโนภาพของตัวกราฟ ทำให้มีการใช้งานจริงพอสมควร
  - เป็นอีกร่างหนึ่งของแนวคิดในกลุ่ม Adjacency list แต่ถ้าจะทำแบบนี้ก็ควรจะทราบว่าลิงค์ลิสต์เป็น reference / pointer ซึ่งใช้พื้นที่ 4 ไบต์ในระบบ 32 บิต และ 8 ไบต์ในระบบ 64 บิต

หัวข้อเนื้อหา

- รู้จักกราฟ
- การอธิบายโครงสร้างของกราฟ
- **พื้นฐานการจัดการและใช้โครงสร้างข้อมูลกราฟ**
- อัลกอริทึมเกี่ยวกับกราฟ
  - Bread-First Search, Depth-First Search, Topological Sort
  - Minimum Spanning Tree
  - Shortest Path Algorithm
  - Network Flow
- ตัวอย่างการประยุกต์ใช้ที่น่าสนใจ

การไปถึงกันได้ระหว่างโหนด

- ก่อนหน้านี้เราพูดถึงการเชื่อมกันของโหนดแบบเชื่อมโดยตรง
- แต่กราฟมักไม่ได้ถูกสร้างมาเพื่อถามเกี่ยวกับการเชื่อมต่อโดยตรง ที่เรามักถามกันบ่อยจริง ๆ คือถามว่า เราสามารถเดินทางจากโหนด x ไปโหนด y ได้หรือไม่
  - เช่นถามว่า เราสามารถเดินทางจากโหนด 0 ไปโหนด 4 ได้หรือไม่
  - หรือ เราสามารถเดินทางจากโหนด 2 ไปโหนด 5 ได้หรือไม่



- เราเรียกการไปถึงกันได้ของโหนดสองโหนดว่า reachable

ภาพจาก Competitive Programming 2nd Edition

แล้วจะรู้ได้ไงว่ามันไปถึงกันได้หรือเปล่า

- ใช้วิธีการค้นหา ซึ่งวิธีพื้นฐานก็คือการใช้ Breadth-First Search (BFS) หรือ Depth-First Search (DFS) จากโหนด x แล้วคอยตรวจดูว่าในระหว่างการค้นหามันเคยเจอโหนด y หรือเปล่า
- ถ้าต้องเขียนโค้ดเอง BFS มีแนวโน้มจะเขียนง่ายกว่า เข้าใจง่ายกว่า
- DFS มักถูกอธิบายในทางหลักการออกมาเป็นแบบ recursive
  - ทำให้มือใหม่งงจนทำอะไรไม่ถูก
  - แต่ที่จริงไม่ต้องทำเป็นรีเคอร์ซีฟก็ได้ (แต่ก็ทำให้งงและยุ่งยากกว่าเดิม)
- เราจะเรียนวิธีเขียนโปรแกรมกับทั้งสองแบบเพราะ
  - BFS เป็นพื้นฐานของ Dijkstra’s algorithm (หนึ่งในอัลกอริทึมที่ฮิตที่สุด)
  - DFS เป็นพื้นฐานของ Topological sorting

หัวข้อเนื้อหา

- รู้จักกราฟ
- การอธิบายโครงสร้างของกราฟ
- พื้นฐานการจัดการและใช้โครงสร้างข้อมูลกราฟ
- **อัลกอริทึมเกี่ยวกับกราฟ**
  - **Bread-First Search, Depth-First Search, Topological Sort**
  - Minimum Spanning Tree
  - Shortest Path Algorithm
  - Network Flow
- ตัวอย่างการประยุกต์ใช้ที่น่าสนใจ

เรื่องที่พบบ่อยในการจัดการกราฟ

- ก่อนจะเริ่มอธิบายตัว BFS เราควรเข้าใจธรรมชาติของการทำงานเกี่ยวกับกราฟอยู่สองประการ
  - ตัวของกราฟเป็นโครงสร้างพื้นฐานทางแนวคิด ตัวของมันเองไม่ค่อยมีประโยชน์กับการใช้งานจริงจนกว่าเราจะขยาย (Augment) โครงสร้างข้อมูลเพื่อให้เข้ากับการใช้งาน
  - การขยายโครงสร้างข้อมูลเป็นสิ่งที่ต้องทำบ่อยมากในกราฟ คนที่ไม่ค่อยเข้าใจแต่เน้นจำจะติดปัญหาตรงนี้มาก
- ท่าไม้ตายที่ใช้บ่อยในการขยายโครงสร้างข้อมูลของกราฟก็คือ การเสริมตัวบันทึกสถานะของโหนด โดยเฉพาะ **การเยี่ยมโหนด (Node-visit status)**

ตัวแปรเก็บสถานะการเยี่ยมโหนด

- ตัวแปรสถานะการเยี่ยมโหนดมักถูกใช้เพื่อป้องกันไม่ให้เราทำเรื่องเดิมซ้ำไม่รู้จบ (ที่จริงเราเอาไว้บันทึกว่าโปรแกรมคำนวณโหนดใดไปแล้วนั่นเอง)
- เนื่องจากเราบันทึกทุกโหนด และโหนดมีเป็นจำนวนมาก

  → ตัวแปรนี้จึงถูกจัดทำในรูปของอาเรย์

```
boolean[] arVisit;

private void prepareSpace() {
   ……
   arVisit = new boolean[9];
}

private void insertData() {
    ……
    Arrays.fill(arVisit, false);
}
```

ตัวแปรเก็บสถานการณ์เยี่ยมโหนด สามารถใช้บูลีนเป็นตัวเก็บข้อมูลได้

ใช้ตัวอย่างเดิมมี 9 โหนด

กำหนดให้ค่าสถานะเริ่มต้นเป็น false คือยังไม่ถูกเยี่ยม (คำสั่งนี้สามารถเติมค่าในอาเรย์ได้เร็วมาก มาจาก java.util)

BFS กับการตรวจหาการไปถึงกันได้ของโหนด

```
int[] arIndex; // Keep pending nodes
```

เอาไว้ใช้บันทึกว่าโหนดไหนบ้างที่ถูกนำมาพิจารณาในการค้นหาครั้งนี้
โหนดไหนที่ถูกนำมาพิจารณาก็คือโหนดที่เข้าถึงได้จากโหนดที่กำหนด

```
public void listReachableNodes(final int src) {
  // Reset visit status
  Arrays.fill(arVisit, false);

  // Init the list of pending nodes
  int pending = src;
  int index0 = 0;    // Start index
  int index1 = 0;    // End index
  arIndex = new int[9];
  arIndex[index1] = pending;
  arVisit[pending] = true;
  ++index1;
  ......
}
```

เก็บอินเด็กซ์โหนดที่กำลังจะพิจารณา

เอาไว้เก็บขอบเขตอินเด็กซ์ของโหนดที่ยังไม่ได้พิจารณา (ค่าเปลี่ยนไปเรื่อย ๆ)

ไฟล์ BfsDemo.java

BFS กับการตรวจหาการไปถึงกันได้ของโหนด (2)

```
public void listReachableNodes(final int src) {
  ......

  while(index0 < index1) {
    // Explore neighboring nodes
    int nNeighbors = arNode[pending].length;
    for(int i = 0; i < nNeighbors; ++i) {
      int id = arNode[pending][i];
      if(arVisit[id] == false) {
        arIndex[index1] = id;
        ++index1;
        arVisit[id] = true;
      }
    }
    ++index0;
    pending = arIndex[index0];
  }
  ......
}
```

BFS กับการตรวจหาการไปถึงกันได้ของโหนด (3)

ส่วนนี้แสดงให้เห็นว่าอาเรย์ที่สร้างขึ้นมาเอาไปใช้ระบุโหนดที่เข้าถึงได้ได้อย่างไร

```
public void listReachableNodes(final int src) {
  ……
  ……

  // List reachable nodes
  System.out.println("Source node = " + src);
  for(int i = 0; i < index1; ++i) {
    System.out.println(arIndex[i]);
  }
}
```

คำถามชวนคิด

ถ้าอยากรู้ว่ากราฟแบบไม่มีทิศทางถูกแบ่งออกเป็นกี่ส่วน ควรจะทำอย่างไร ?

(ส่วนเดียวกันคือโหนดที่เชื่อมต่อถึงกันไปมาได้หมด)

กราฟที่การเชื่อมต่อมีรูปแบบที่แน่นอนเป็นกฎที่ชัดเจน

- ถ้ามีขั้นตอนที่แน่นอนในการระบุการเชื่อมต่อของโหนด เราจะไม่จำเป็นต้องใช้ Adjacency list หรือ matrix ในการอธิบายโครงสร้าง
- เราเรียกกราฟพวกนี้ว่ากราฟโดยนัย (Implicit Graph)
- ใช้มากในวิชา Image Processing เพื่อหา Connected Component (Flood-fill algorithm จะเป็นของคู่กับกราฟพวกนี้)
- ตัวอย่างการใช้ก็คือหาว่าพื้นที่ (region) ในรูปภาพมีทั้งหมดเท่าใด อยู่ตำแหน่งไหน (ข้อมูลพวกนี้จะถูกนำมาใช้ในการวิเคราะห์อื่น ๆ ต่อไป)

โจทย์ทดสอบระบบ ACM ภาคกลางปี 2012

ข้อ Island Survey: นับว่ามีเลข 1 ที่เชื่อมต่อกันเป็นพื้นที่แยกกันกี่พื้นที่

0	0	0	0	0	0	0	0	0
0	0	0	**1**	**1**	**1**	0	**1**	0
0	**1**	**1**	0	**1**	**1**	0	**1**	0
0	0	**1**	**1**	0	0	0	**1**	0
0	0	0	0	0	0	0	0	0

กำหนดให้การเชื่อมต่อกันของพื้นที่เป็นไปได้ 8 ทิศทาง ดังนั้นพื้นที่หมายเลข 1 ที่เชื่อมต่อกันจะแยกเป็นพื้นที่ได้ทั้งหมด 2 พื้นที่

- ข้อกำหนดเกี่ยวกับการเชื่อมต่อทำให้เราไม่จำเป็นต้องให้ข้อมูลเส้นเชื่อมแยกต่างหาก แต่คิดได้จากตำแหน่งที่กำลังพิจารณาแต่ละตำแหน่งเลย
- กราฟถูกมองได้เป็นอาเรย์สองมิติ และการสำรวจการเชื่อมต่อสามารถทำได้โดยการใช้ตัวเก็บสถานะการเยี่ยมโหนดที่เป็นอาเรย์สองมิติ

ระเบิดกำแพงเขาวงกต (TOI 2012)

มีระเบิดหนึ่งลูก ต้องใช้ระเบิดกำแพงในจุดที่ทำให้เกิดทางที่สั้นที่สุดระหว่างจุดเริ่มต้นไปถึงทางออก

นักผจญภัยเดินได้เฉพาะบนช่องเลข 1 ในแนวตั้งฉาก จุดเริ่มต้นคือวงรีแดง ทางออกคือสามเหลี่ยมเลข 1 แต่มีกำแพงเลข 0 ขวางไว้ ให้หาว่าจะวางระเบิดซึ่งระเบิดช่องเลข 0 ได้แค่ช่องเดียวอย่างไรจึงจะได้ทางที่สั้นที่สุด

0	0	1	1	0	0	0	0
1	0	1	1	0	1	1	1
1	0	1	1	1	0	0	1
1	1	0	0	1	0	0	1
0	0	1	1	0	1	1	1

DFS กับการตรวจหาการไปถึงกันได้ของโหนด

- ต่อให้เป็น DFS ก็ต้องใช้ตัวแปรเก็บสถานะเพื่อป้องกันการทำงานซ้ำ

  → ตัวแปรเก็บสถานะการเยี่ยมโหนดเป็นของคู่ชีพการจัดการกราฟจริง ๆ

```
int[] arIndex; // Keep pending nodes
int index;     // Keep current cursor of arIndex

public void listReachableNodes(final int src) {
  // Reset visit status
  Arrays.fill(arVisit, false);

  // Init the list of pending nodes
  arIndex = new int[9];
  index = 0;
  arIndex[index] = src;
  arVisit[src] = true;
  ++index;
  dfs(src);
  ……
}
```

ไฟล์ DfsDemoRecursive.java

ส่วนหลักของ DFS

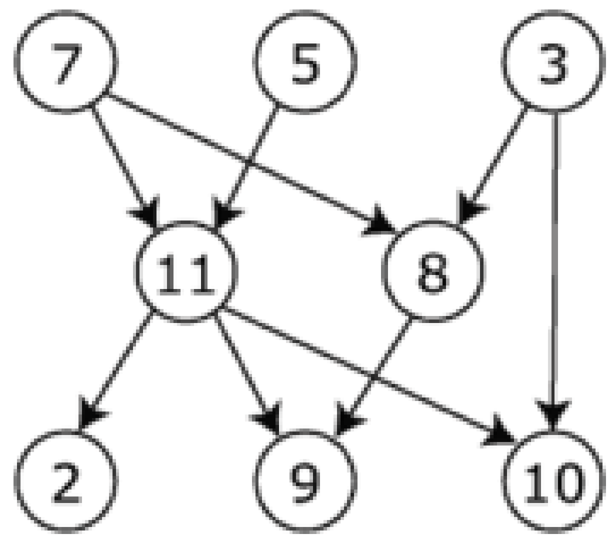
```
private void dfs(final int src) {
  int nNeighbors = arNode[src].length;
  for(int i = 0; i < nNeighbors; ++i) {
    int id = arNode[src][i];
    if(arVisit[id] == false) {
      arIndex[index] = id;
      arVisit[id] = true;
      ++index;
      dfs(id);
    }
  }
}
```

พอพบโหนดใหม่ เราก็รีบเจาะลึกค้นหามันต่อไปทันที ไม่รอดูเพื่อนบ้านคนอื่นของโหนด src ปัจจุบัน

- ถ้าไม่ทำเป็นแบบรีเคอซีฟเรื่องจะยุ่งขึ้นเล็กน้อย เพราะโหนดที่เราจดจ่ออยู่จะเปลี่ยนไปเร็วมากทำให้เราต้อง (1) อ่านลิสต์เพื่อนบ้านใหม่เมื่อย้อนกลับมาซ้ำ ๆ หรือ (2) ต้องคอยจำค่าว่าอ่านลิสต์เพื่อนบ้านแต่ละคนไปถึงไหนแล้ว และมีรายละเอียดจุกจิกอื่น ๆ ด้วย

Topological Sort (toposort)

- เป็นการจัดลำดับโหนดในกราฟที่มีทิศทางและไม่มีลูป (Directed Acyclic Graph / DAG)
- มักใช้กับการจัดแผนงาน (Job Scheduling) ที่ขั้นตอนของงานมีทั้งส่วนที่ขึ้นต่อกันและเป็นอิสระต่อกัน
- ขั้นตอนงานที่ขึ้นต่องานอื่นหมายความว่า จะทำขั้นตอนนั้นได้ถ้างานอื่นที่ว่าทำเสร็จไปก่อนแล้ว
- จากภาพ ขั้นตอนงาน 9 จะทำได้ก็ต่อเมื่อขั้นตอนงาน 8 และ 11 ทำเสร็จแล้ว ในทำนองเดียวกัน ขั้นตอนงาน 11 จะทำได้ก็ต่อเมื่อขั้นตอนงาน 5 และ 7 ถูกเสร็จไปก่อน



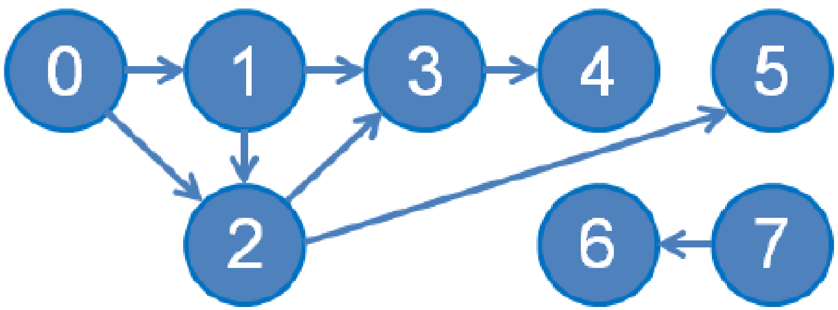
ภาพจาก wikipedia

ถ้าขั้นตอนงานมีมากและวุ่นวายไปหมด

- การวิเคราะห์การขึ้นต่อกันของขั้นตอนช่วยให้เรารู้ได้ว่างานขั้นตอนใดบ้างที่สามารถทำพร้อมกันได้และจะจัดสรรทรัพยากรอย่างไรเพื่อให้งานทั้งหมดเสร็จเร็วที่สุด (วิศวกรอุตสาหการใช้การวิเคราะห์พวกนี้ค่อนข้างมาก)
- มันไม่ใช่เรื่องง่ายนักที่จะวิเคราะห์ขั้นตอนงานในผังงานขนาดใหญ่ให้เห็นถึงการขึ้นต่อกันได้อย่างเป็นระบบ → ต้องหาอัลกอริทึมที่เหมาะสมมาใช้
- วิธีที่ง่ายและมีประสิทธิภาพก็คือการดัดแปลง DFS ให้ใส่โหนดที่อยู่ลึกที่สุดลงไปทางด้านใต้ โหนดที่มาก่อนหน้าจะอยู่ตื้นลงมา
- ค่อย ๆ ทำอย่างนี้กับทุกโหนดที่ยังไม่ได้ถูกเยี่ยม เดี๋ยวดีเอง (เหลือเชื่อมาก)
  - รับประกันได้ว่างานที่ต้องถูกทำทีหลังจะตกไปอยู่ทางด้านใต้ของงานที่ต้องทำให้เสร็จก่อน
  - งานที่ทำได้พร้อมกันอาจจะสลับตำแหน่งกันก็ได้ ใครอยู่เหนือใครก็ไม่เป็นไร

มาลองดูไอเดีย Toposort ดีกว่าว่ามันถูกจริงหรือเปล่า

- จากกราฟจะทำให้ดูว่าถ้าเราพยายามเลือกพิจารณาโหนดในลำดับ 1 0 3
  - โหนดในกลุ่มนี้จะถูกเรียงลำดับอย่างถูกต้อง
  - งานที่มาทีหลังจะถูกใส่ลงไปทางด้านใต้ (ใช้ stack ก็ได้)



- จะเห็นได้ว่าเมื่อพิจารณาโหนด 0 มันจะถูกใส่ไว้ด้านบนก่อนโหนด 1 โดยปริยาย
- เมื่อพิจารณาโหนด 3 เราจะพบว่ามันอยู่ใน stack เรียบร้อยแล้ว

ภาพจาก Competitive Programming 2nd Edition

Toposort Implementation

ส่วนที่ดัดแปลงมาจาก dfs

```
void dfs2(int src) {
  arVisit[src] = true;

  final int nNeighbors = arNode[src].length;
  for(int i = 0; i < nNeighbors; ++i) {
    int id = arNode[src][i];
    if(arVisit[id] == false) {
      // No node insertion here, keep it for later.
      dfs2(id);
    }
  }

  //System.out.println("Insert " + src);
  arOrder[index] = src;
  ++index;
}
```

ไฟล์ ToposortDemo.java

Toposort Implementation (2)

ส่วนเตรียมตัวเรียกการใช้งาน

```
public void listTopoOrder() {
  // Reset visit status
  Arrays.fill(arVisit, false);

  // Init the list of pending nodes
  arOrder = new int[8]; // This example has Nodes 0 to 7.
  index = 0;
  for(int src = 0; src < 8; ++src) {
    if(arVisit[src] == false)
      dfs2(src);
    }

    // Write topological order
    System.out.println("Topological order:");
    for(int i = index - 1; i >= 0; --i) {
      System.out.println(arOrder[i]);
    }
}
```

เนื่องจากใช้อาเรย์มาเก็บลำดับ
ไม่ได้ใช้แสต็ค ลำดับจึงย้อนหลัง

ประยุกต์ใช้กับ Job Scheduling

- สมมติว่าข้อจำกัดมีเพียงว่าต้องทำงานที่จำเป็นก่อนหน้าให้เสร็จก่อน
  - งานขั้นตอนงานทุกอย่างใช้เวลาเท่ากัน
  - งานที่ไม่ขึ้นต่อกันทำพร้อมกันกี่อันก็ได้ ขอแค่ขั้นตอนก่อนหน้าเสร็จไปแล้ว
  - อยากตอบให้ได้ว่างานจะเสร็จเร็วที่สุดได้ต้องทำกี่ขั้น และควรจัดงานอย่างไร
- เราตอบคำถามนี้ได้ไม่ยากถ้าเราทำ topological sort ไปแล้ว
- จากโหนดที่อยู่บนสุด ซึ่งเรามั่นใจว่าสามารถเริ่มงานได้ทันที
  - ทำ BFS จากโหนดบนสุดแล้วบันทึกความลึกจากโหนดบนสุดไว้
  - จุดต่างมีอยู่ว่าโหนดที่ถูกเยี่ยมแล้วเยี่ยมซ้ำได้ถ้าความลึกจากเส้นทางที่ตามมามันมากกว่าเดิม
  - ทำ BFS จากโหนดที่ยังไม่ได้เยี่ยมจากรอบที่แล้ว (แสดงว่าเป็นโหนดเริ่มต้นเหมือนกัน) และทำในลักษณะเดิมไปเรื่อย ๆ จนหมด

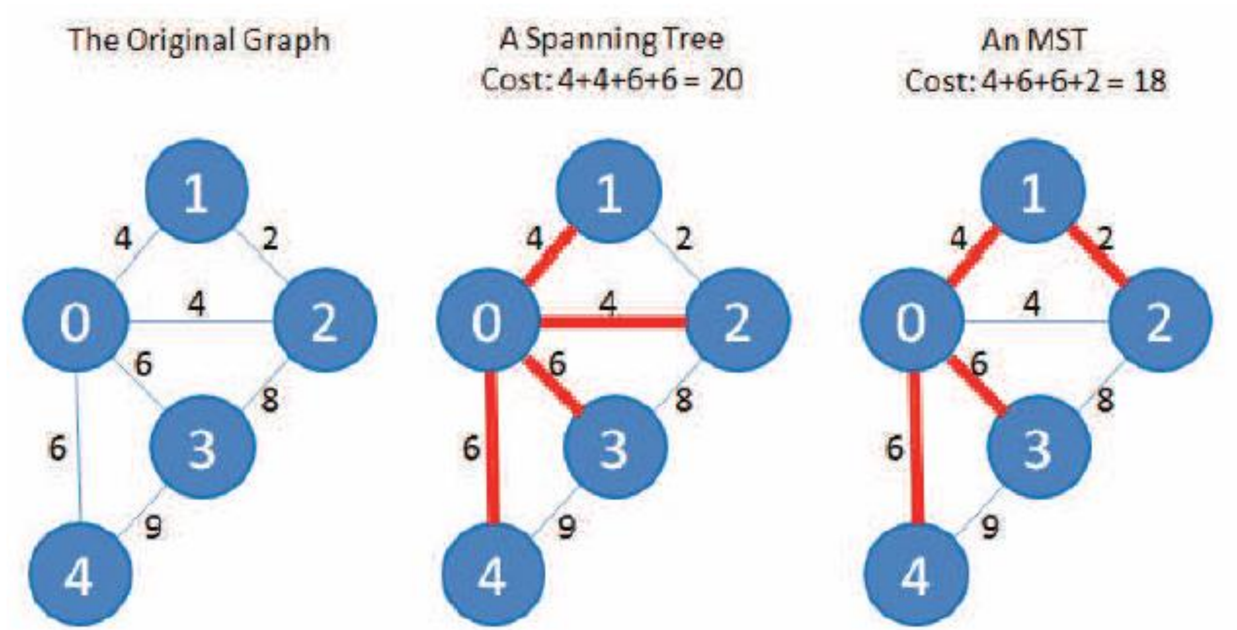
หัวข้อเนื้อหา

- รู้จักกราฟ
- การอธิบายโครงสร้างของกราฟ
- พื้นฐานการจัดการและใช้โครงสร้างข้อมูลกราฟ
- **อัลกอริทึมเกี่ยวกับกราฟ**
  - Bread-First Search, Depth-First Search, Topological Sort
  - **Minimum Spanning Tree**
  - Shortest Path Algorithm
  - Network Flow
- ตัวอย่างการประยุกต์ใช้ที่น่าสนใจ

Minimum Spanning Tree (MST)

หากเรามีกราฟไม่มีทิศทางและเส้นเชื่อมมีค่าน้ำหนักกำกับอยู่ เราต้องการหาต้นไม้ในกราฟที่เชื่อมโหนดทุกโหนดเข้าด้วยกันได้ และมีผลรวมของค่าน้ำหนักของเส้นเชื่อมในต้นไม้น้อยที่สุด

- ต้นไม้ที่เชื่อมโหนดทุกโหนดเข้าด้วยกันได้เรียกว่า Spanning Tree
- ส่วน Spanning Tree ที่มีผลรวมค่าน้ำหนักน้อยที่สุดคือ Minimum Spanning Tree
- มักถูกใช้ในการหาวิธีวางท่อหรือสายเคเบิลที่เชื่อมจุดสำคัญและเสียค่าใช้จ่ายน้อยที่สุด

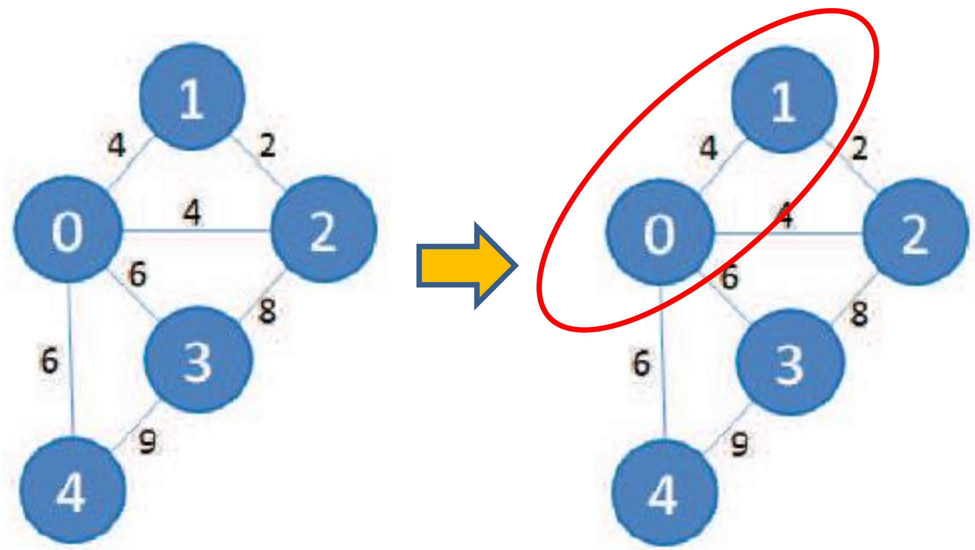


ภาพจาก Competitive Programming 2^{nd} Edition

Prim's Algorithm

แนวคิดพื้นฐาน

- ในเมื่อต้องเชื่อมทุกโหนด ดังนั้นเริ่มคิดจากโหนดไหนก็ได้
- จากโหนดเดียวให้ขยายไปโหนดที่ติดกันด้วยเส้นเชื่อมที่มีน้ำหนักน้อยที่สุด (ถ้ามีหลายเส้นที่เบาที่สุด เลือกเส้นไหนก็ได้)
- สมมติว่าเราเริ่มที่โหนด 0 เราจะเลือกเส้นเชื่อมไปโหนด 1

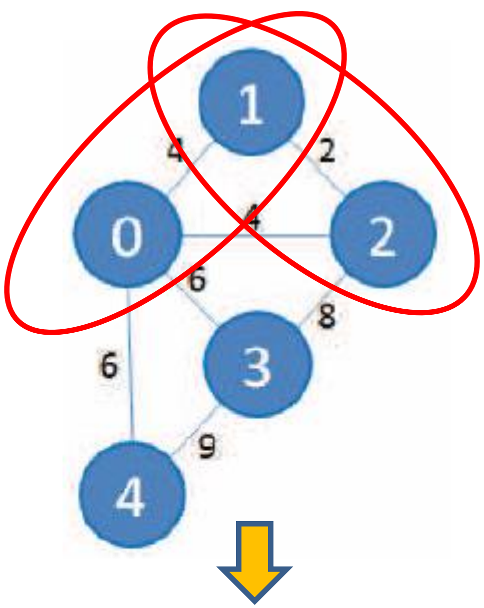


มองโหนดที่เชื่อมถึงกันไปแล้วเป็น Super Node (รวมโหนดเข้าด้วยกัน)

จากนั้นทำแบบเดิมอีก คราวนี้เราเลือกเส้นเชื่อมไปโหนด 2 ที่มีน้ำหนัก 2 เพราะเบาที่สุด จากตัวเลือกที่มีอยู่

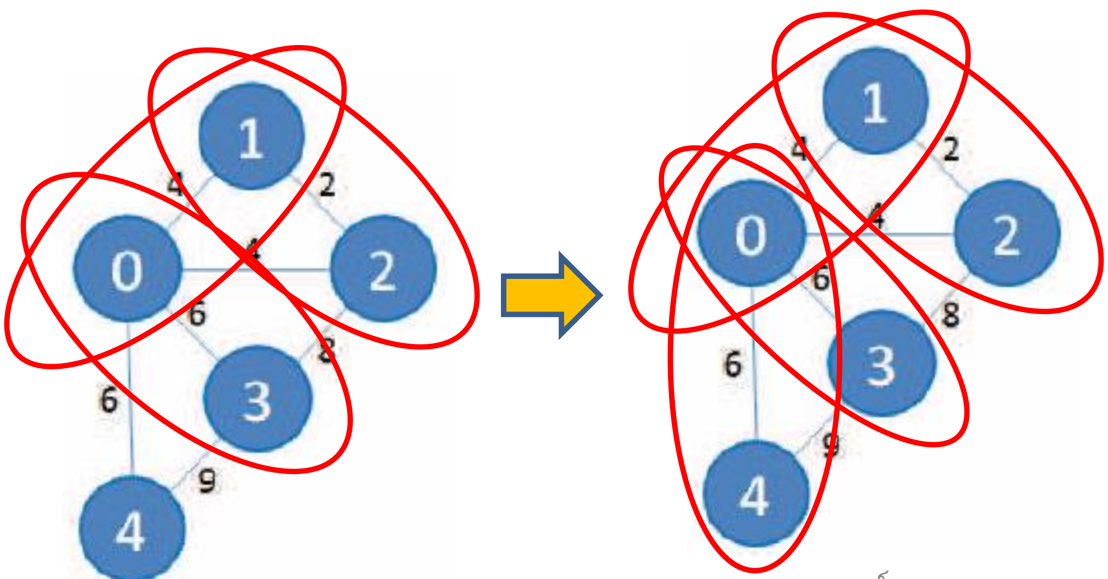
ภาพจาก Competitive Programming 2nd Edition

การทำงานของ Prim's Algorithm



ตอนนี้ Super Node ของเรามีโหนดย่อยสามโหนดอยู่ด้วยกัน คือโหนด 0, 1 และ 2

เส้นเชื่อมที่อยู่ภายใน Super Node นี้ไม่ถือว่าเป็นเส้นเชื่อมไปข้างนอกหาเพื่อนบ้าน ดังนั้นไม่นับ (เช่นเส้นเชื่อมระหว่างโหนด 0 กับโหนด 2 เป็นเส้นเชื่อมภายใน Super Node เราไม่นับ)



เมื่อรวมโหนดทุกโหนดมาหมดแล้ว ก็เป็นอันจบการทำงาน

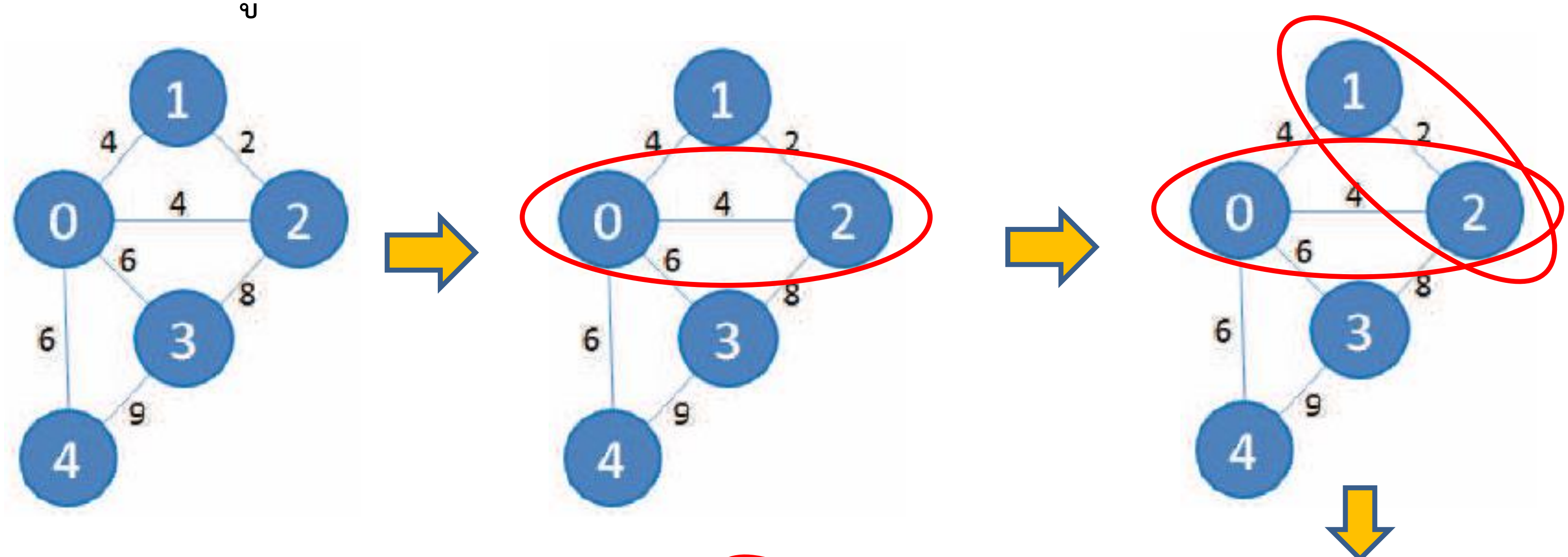
รวมค่าน้ำหนักทั้งหมดได้

4 + 2 + 6 + 6 = 18

ภาพจาก Competitive Programming 2nd Edition

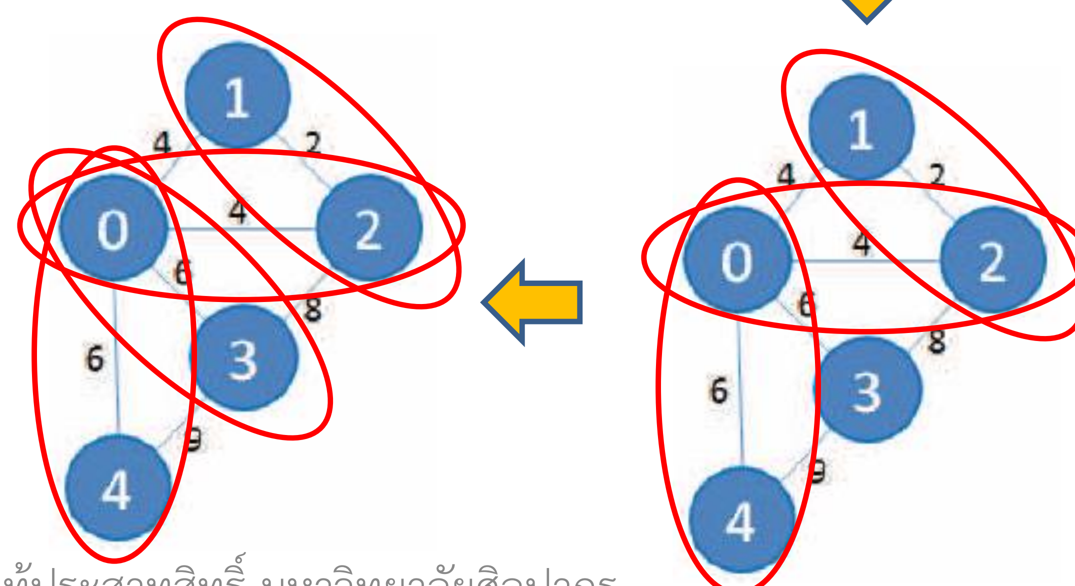
แล้วถ้าเลือกเส้นเชื่อมอีกทางหนึ่งที่หนักน้อยสุดเท่ากันล่ะ

เส้นเชื่อมที่ถูกเลือกก็เปลี่ยนไป แต่ค่าน้ำหนักรวมก็ยังคงเดิม



บางทีเส้นที่เลือกก็เปลี่ยน

บางทีก็แค่เปลี่ยนลำดับที่ถูกเลือก



ภาพจาก Competitive Programming 2nd Edition

Naïve Implementation of Prim’s Algorithm

- แนวคิด: เมื่อรวมโหนดเข้ามา ให้รวมเส้นเชื่อมของมันเข้ามาในลิสต์สำหรับพิจารณาในภายหลัง
  - จากนั้นก็ตามหาเส้นเชื่อมในลิสต์ที่มีน้ำหนักน้อยที่สุด
  - ดูว่าปลายทั้งสองของเส้นเชื่อมอยู่ภายใน Super Node หรือไม่ ถ้าใช่แสดงว่าเป็นเส้นเชื่อมภายใน ไม่สามารถใช้ได้ ให้หาต่อไปจนกว่าจะเจอเส้นเชื่อมที่ต่อออกไปข้างนอก Super Node

การเก็บเส้นเชื่อมคู่กับค่าน้ำหนัก

- ตอนนี้กราฟของเราเริ่มมีค่าน้ำหนักเส้นเชื่อมเข้ามาเกี่ยวข้อง
  - เราต้องขยายโครงสร้างข้อมูลขึ้นไปอีกระดับ
  - เราต้องการผูกค่าน้ำหนักกับเส้นเชื่อมเข้าด้วยกัน
    → นิยามคลาสสำหรับเก็บข้อมูลพวกนี้ไว้จะสะดวกขึ้น
  - ถ้าเราใช้ออปเจ็คในจาวามาเก็บข้อมูลมันจะทำให้เราใช้พวกคลาสอำนวยความสะดวกอย่าง Vector ได้ (vector ใน C++ ใช้กับข้อมูลอย่าง int ได้)
- ลองหัดใช้คลาสพวก Vector/ArrayList จัดการกับปัญหากราฟจะช่วยเราได้มาก เพราะกราฟที่มีประโยชน์มักมีโครงสร้างข้อมูลที่ซับซ้อน

คลาสสำหรับเส้นเชื่อมใน Adjacency list

```
class Edge {
  int node0, node1;
  int weight;
  boolean valid;

  public Edge(int node0, int node1, int weight) {
    this.node0 = node0;
    this.node1 = node1;
    this.weight = weight;
    valid = true;
  }
}
```

ไฟล์ NaivePrimDemo.java

Vector

- มาจาก java.util.Vector;
- เป็นอาเรย์ที่ขยายขนาดได้ ทำให้เราไม่จำเป็นต้องกังวลว่าเราต้องเผื่อพื้นที่สำหรับเก็บสิ่งต่าง ๆ ไว้ในอาเรย์มากเกินไป
- เก็บได้เฉพาะ Object Reference
- ในตัวอย่างนี้เราจะใช้มันเก็บเส้นเชื่อมที่ต่อกับ Super Node

```
Vector<Edge> vEdge = new Vector<Edge>();
```

- ถ้าอยากใส่ของเพิ่มเข้าไปต่อทางด้านท้าย เช่น `vEdge.add(e);`
- ถ้าอยากได้ของที่ช่องข้อมูลใด ให้ใช้เมธอด elementAt เช่น

```
Edge e = vEdge.elementAt(i);
```

ทำความคุ้นเคยกับคลาส Edge ที่เราสร้างขึ้นมา

```
Edge[][] arNode;
boolean[] arVisit;

private void insertData() {
   arNode[0][0] = new Edge(0, 1, 4);
   arNode[0][1] = new Edge(0, 2, 4);
   arNode[0][2] = new Edge(0, 3, 6);
   arNode[0][3] = new Edge(0, 4, 6);

   arNode[1][0] = new Edge(1, 0, 4);
   arNode[1][1] = new Edge(1, 2, 2);

   arNode[2][0] = new Edge(2, 0, 4);
   arNode[2][1] = new Edge(2, 1, 2);
   arNode[2][2] = new Edge(2, 3, 8);
   .........
}
```

node0 = หมายเลขโหนดด้านต้นเส้นเชื่อม

node1 = หมายเลขโหนดด้านปลายเส้นเชื่อม

weight = น้ำหนักเส้นเชื่อม

ที่จริงเป็นเส้นเดียวกับ arNode[0][0]

ใจความของ Naïve Implementation (1)

```
Vector<Edge> vMst;
int weightSum;
public void mst(int src) {
  vMst = new Vector<Edge>();
  weightSum = 0;
  while(true) {
    arVisit[src] = true;

    // Add edges to vector
    int nNeighbors = arNode[src].length;
    for(int i = 0; i < nNeighbors; ++i) {
      Edge e = arNode[src][i];
      int id = e.node1;
      if(arVisit[id] == false) {
        vEdge.add(e);
      }
    }
    ……
  }
}
```

ใจความของ Naïve Implementation (2)

```
while(true) {
  ......
  // Explore the vector for minimum-weight edge
  int numEdges = vEdge.size();
  int minWeight = Integer.MAX_VALUE;
  int minIndex = -1;
  for(int i = 0; i < numEdges; ++i) {
    Edge e = vEdge.elementAt(i);
    if(e.valid && e.weight < minWeight) {
      if(arVisit[e.node0] == false ||
         arVisit[e.node1] == false) {
        minWeight = e.weight;
        minIndex = i;
      } else {
        e.valid = false;
      }
    }
  }
  ......
}
```

ใจความของ Naïve Implementation (3)

```
while(true) {
  ……
  if(minIndex == -1) {  // No valid edge left, finish
    break;
  } else {
    weightSum += minWeight;
    Edge e = vEdge.elementAt(minIndex);
    vMst.add(e);
    src = e.node1; // เปลี่ยน src ไปเป็นโหนดที่เพิ่งใส่เข้ามาใน Super Node
    e.valid = false;
  }
}
```

ทำให้ Prim's Algorithm เร็วขึ้นกว่าเดิม

- จากวิธีที่ใช้ เราจะเห็นได้ว่า อัลกอริทึมต้องใช้เวลาในการหาเส้นเชื่อมที่มีค่าน้ำหนักน้อยที่สุดนานมาก (เป็นลูปด้านในที่ต้องวิ่งไปตลอดทั้งเวคเตอร์)
- เนื่องจากต้องวิ่งเป็นจำนวนรอบตามจำนวนโหนดใน MST และเส้นเชื่อมมีได้มากถึง $|E|$ จำนวนเวลาที่ต้องใช้จึงเป็น $O(|V||E|)$
- ในเมื่อเป้าหมายที่แท้จริงของการทำงานในแต่ละรอบใหญ่ คือการหาเส้นเชื่อมที่เบาที่สุด ที่เชื่อมออกไปนอก Super Node
  - ถ้าเราใช้โครงสร้างข้อมูลที่สามารถ Find Min ได้เร็ว ๆ คงจะดีไม่น้อย
  - เมื่อได้ค่า Min แต่ละตัวมาแล้ว ถ้าเราเพียงทดสอบว่าเส้นเชื่อมค่า Min ดังกล่าวเป็นเส้นเชื่อมที่ถูกต้องหรือไม่ ถ้าใช่ก็เยี่ยม ถ้าไม่ใช่ก็หาต่อไป
  - ถ้าการ Find Min แต่ละครั้งทำให้เรากำจัดเส้นเชื่อมที่ไม่ถูกต้องออกไปได้อย่างรวดเร็ว มันก็จะเยี่ยมมาก

และผู้ที่มีคุณสมบัติดังกล่าวครบก็คือ Priority Queue

- รู้จักกันในอีกชื่อคือ (Min) Heap
- เราสามารถ extract / remove min และ add เส้นเชื่อมได้เร็วพอสมควร → $O(\log(|M|))$ เมื่อ M คือจำนวนเส้นเชื่อม**ใน Heap**
- เพราะจำนวนเส้นเชื่อมสูงสุดเป็น $O(|E|)$ ดังนั้นความเร็วในการทำงานเมื่อเปลี่ยนมาใช้ฮีปคือ $O(|E| \log |E|)$ → $O(|E| \log |V|)$
- เอ๋ ในเมื่อลูปด้านนอกมันวนตามจำนวนโหนด แล้วทำไมความเร็วมันไม่เป็น $O(|V| \log |E|) = O(|V| \log |V|)$?
  - ที่เป็นแบบนี้ก็เพราะว่าแต่ก่อนเราใส่เส้นเชื่อมเข้าไปในเวคเตอร์ด้วยเวลา $O(1)$
  - แต่เมื่อใส่เข้าไปในฮีป มันจะใช้เวลา $O(\log |E|)$ และใส่ไปทั้งหมด $|E|$ ครั้ง

ข้อคิดดี ๆ จากเทคนิคที่ใช้ใน Prim’s Algorithm

- เริ่มเห็นแล้วใช่หรือไม่ว่ากราฟเป็นแหล่งรวมการสร้างสรรค์และใช้งานโครงสร้างข้อมูลสารพัดรูปแบบจริง ๆ
- เรื่องที่น่าสนใจและได้ใช้บ่อยมากก็คือว่า ในงานที่เราต้องหาค่าที่น้อยหรือมากที่สุดในข้อมูลที่วิ่งเข้าวิ่งออกอยู่ตลอด การใช้ Priority Queue จะช่วยได้มาก
- ดังนั้นในการพัฒนาอัลกอริทึม เราไม่ควรมองข้ามโครงสร้างข้อมูลต่าง ๆ ที่เราได้เรียนมา โดยเฉพาะพวก balance tree
    - อัลกอริทึมชื่อดังอย่าง Dijkstra’s ก็ใช้ Priority Queue เช่นกัน

มาลองใช้ Priority Queue (PQ) กันเลยดีกว่า

- PriorityQueue เป็นคลาสที่มีเมธอดที่สำคัญดังนี้
  - add(E e) เติมค่าเข้าไปใน PQ
  - poll( ) เป็นการสกัดค่าที่น้อยที่สุดออกมาจาก PQ
  - size( ) ดูจำนวนค่าที่เหลืออยู่ใน PQ
- PQ เป็น Generic Container สามารถเก็บวัตถุจากคลาสได้ตามที่เรากำหนด (เป็นแบบเดียวกับ Vector เช่น ถ้าอยากให้เก็บ Edge ก็เขียนว่า PriorityQueue<Edge>
- แต่การใช้เมธอด add นั้น ตัว PQ จะต้องตัดสินได้ว่าความมากน้อยของวัตถุที่อยู่ข้างในเป็นอย่างไรเพื่อที่จะได้จัดลำดับได้ถูก

  → วัตถุจึงควร implements interface Comparable ด้วย

ทำคลาส Edge ให้รองรับอินเตอร์เฟซ Comparable

เราจะต้องประกาศเมธอด int compareTo(Edge e)

- เมธอดนี้ควรคืนค่าลบถ้าหากเราถือว่าวัตถุของเรามีค่าน้อยกว่าวัตถุ e ถ้าค่าเท่ากันคืนเลข 0 ถ้าค่ามากกว่าคืนเลขบวก
- ถ้าเราต้องการให้วัตถุที่มีค่ามากขึ้นไปอยู่บนสุดของ PQ ให้ทำกลับกันกับที่บอกตรงนี้ เช่น ถ้าเราต้องการให้ poll คืนค่าสูงสุดมาให้ เราก็ควรทำให้เมธอด compareTo คืนค่าลบเมื่อวัตถุเรามีค่ามากกว่า e
- วิธีที่ใช้บอกว่าค่าน้อยกว่า เท่ากัน หรือมากกว่าจะแตกต่างกันไปตามแต่ข้อมูลในคลาสที่เกี่ยวข้องกับการเปรียบเทียบ
- ในกรณีที่ตัดสินด้วยค่า weight เดียวแบบนี้ วิธีที่คลาสสิคที่สุดก็คือ

```
return this.weight - e.weight;
```

โฉมหน้าของคลาส Edge ที่เป็น Comparable ในตัว

สภาพหลังจากตัดตัวแปรที่ไม่จำเป็นออกไปคือ node0 และ valid

```
class Edge implements Comparable<Edge> {
  int node1; int weight;

  public Edge(int node1, int weight) {
    this.node1 = node1;
    this.weight = weight;
  }

  public int compareTo(Edge e) {
    return this.weight - e.weight;
  }
}
```

ไฟล์ PqPrimDemo.java

เอาไปใช้กับ PriorityQueue

```
PriorityQueue<Edge> pq = new PriorityQueue<Edge>();

while(true) {
  arVisit[src] = true;

  // Add edges to PriorityQueue
  int nNeighbors = arNode[src].length;
  for(int i = 0; i < nNeighbors; ++i) {
    Edge e = arNode[src][i];
    int id = e.node1;
    if(arVisit[id] == false) {
      pq.add(e);
    }
  }
  ……
}
```

ขั้นตอนการหาค่าน้ำหนักน้อยที่สุดง่ายนิดเดียว

```
while(true) {
  ......
  Edge minEdge = null;

  while(pq.size() > 0) {

    Edge e = pq.poll();

    // Check if edge connects outside of super node.
    if(arVisit[e.node1] == false) {
      minEdge = e;
      break;
    }
  }
  ......
}
```

เอาไว้เก็บเส้นเชื่อมที่ค่าน้อยที่สุด การใส่ค่า null เอาไว้ใช้ตรวจว่า ‘ตกลงเจอเส้นเชื่อมที่ถูกกฎหรือไม่’

ต้องระวังว่าจะไม่ดึงของใน pq ออกมาตอนที่มันไม่มีอะไรอยู่

ใช้คำสั่ง poll ทีเดียวออกฤทธิ์

ถ้า e.node1 ยังไม่ถูกเยี่ยม แสดงว่าเส้นเชื่อมนี้ต่อออกไปนอก Super Node

ปัดกวาดเช็ดถู บันทึกผลการค้นหาให้เรียบร้อย

```
while(true) {
  ……
  if(minEdge == null) {    // No valid edge left, finish
    break;
  } else {
    weightSum += minEdge.weight;
    vMst.add(minEdge);
    src = minEdge.node1;
  }
}

System.out.println("Edge found = " + vMst.size());
System.out.println("Weight sum = " + weightSum);
```

หัวข้อเนื้อหา

- รู้จักกราฟ
- การอธิบายโครงสร้างของกราฟ
- พื้นฐานการจัดการและใช้โครงสร้างข้อมูลกราฟ
- **อัลกอริทึมเกี่ยวกับกราฟ**
  - Bread-First Search, Depth-First Search, Topological Sort
  - Minimum Spanning Tree
  - **Shortest Path Algorithm**
  - Network Flow
- ตัวอย่างการประยุกต์ใช้ที่น่าสนใจ

Shortest Path Algorithm

- Single Source Shortest Path (SSSP)
  - บนกราฟที่เส้นเชื่อมไม่มีน้ำหนัก (Unweighted graph)
  - บนกราฟที่เส้นเชื่อมมีน้ำหนักเป็นบวกหรือศูนย์ (Non-negative edge)
  - บนกราฟที่เส้นเชื่อมติดลบได้ แต่ไม่มีลูปที่ติดลบ
  - บนกราฟที่มีลูปที่ติดลบ
- Single Pair Shortest Path
  - ประยุกต์จาก SSSP แล้วตัดเอาเฉพาะคู่ที่ต้องการ
  - A* Heuristic Search
- All-Pair Shortest Path
  - Floyd Warshall's Dynamic Programming Algorithm

SSSP บนกราฟที่เส้นเชื่อมไม่มีน้ำหนัก

- หากเส้นเชื่อมไม่มีน้ำหนัก เราจะถือว่าจำนวนเส้นเชื่อมที่ต้องเดินทางผ่านคือระยะทาง
  → ดังนั้นเส้นทางที่สั้นที่สุดก็คือเส้นทางที่ผ่านเส้นเชื่อมน้อยที่สุดนั่นเอง
- เราใช้ Breadth-First Search มาแก้ปัญหานี้ได้ทันที
- เพราะ BFS จะสำรวจโหนดที่ใกล้ที่สุดจนหมดก่อนที่จะย้ายไปสำรวจโหนดที่ไกลออกไป
- เมื่อสิ้นสุด BFS จากโหนด Source เราจะรู้ระยะทางที่สั้นที่สุดจาก Source ดังกล่าวไปยังโหนดทุกโหนดที่สามารถไปถึงได้จาก Source

SSSP บนกราฟที่ไม่มีเส้นเชื่อมค่าน้ำหนักติดลบ

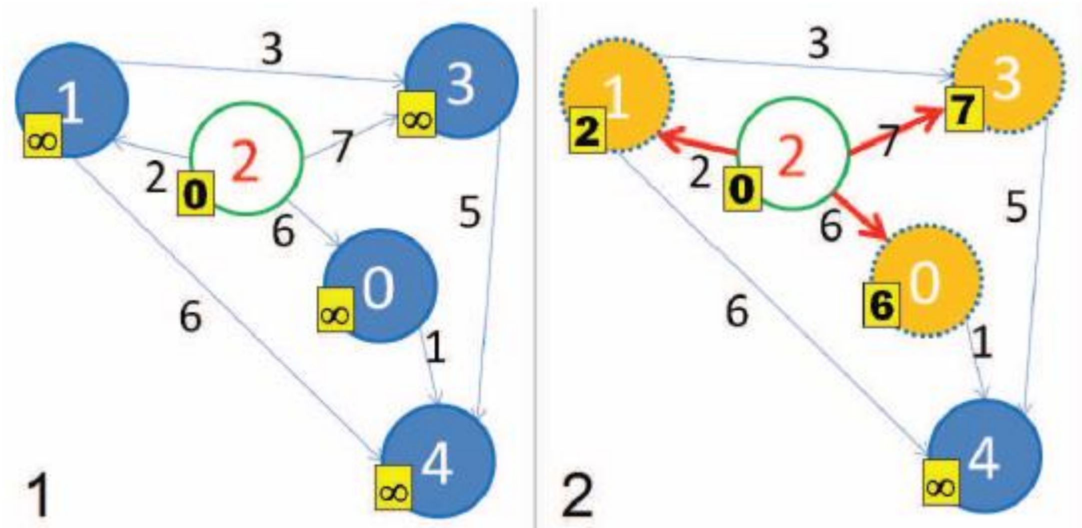
เราใช้ BFS ไม่ได้ แต่เราจะใช้อัลกอริทึมที่ชื่อว่า Dijkstra’s Algorithm

แนวคิด

- เนื่องจากค่าน้ำหนักในเส้นเชื่อมไม่มีค่าติดลบ แสดงว่ายิ่งเดินทางผ่านโหนดมากเท่าไหร่ ค่าน้ำหนักก็จะยิ่งเพิ่มมากขึ้นเท่านั้น
- ดังนั้นจากโหนด Source ถ้ามองออกไปรอบ ๆ ที่เพื่อนบ้าน หากเลือกเส้นเชื่อมที่มีค่าน้อยที่สุด เราเลือกเส้นเชื่อมนั้นได้เลยนั่นจะเป็นทางที่สั้นที่สุดจาก Source แน่ ๆ
    - ➢ อย่างที่บอกไว้ว่าเส้นเชื่อมไม่มีค่าติดลบ ดังนั้นใครที่เป็นผู้ชนะแล้วจะไม่มีทางถูกแย่งชิงตำแหน่งไปได้แน่นอน
- ผนวกโหนดที่ไปถึงแล้วเข้ามาเป็น Super Node พร้อมกับเส้นเชื่อมของมัน จากนั้นทำไปเรื่อย ๆ จนไม่พบโหนดใหม่ในกราฟอีก

วิธีทำงานของ Dijkstra’s Algorithm

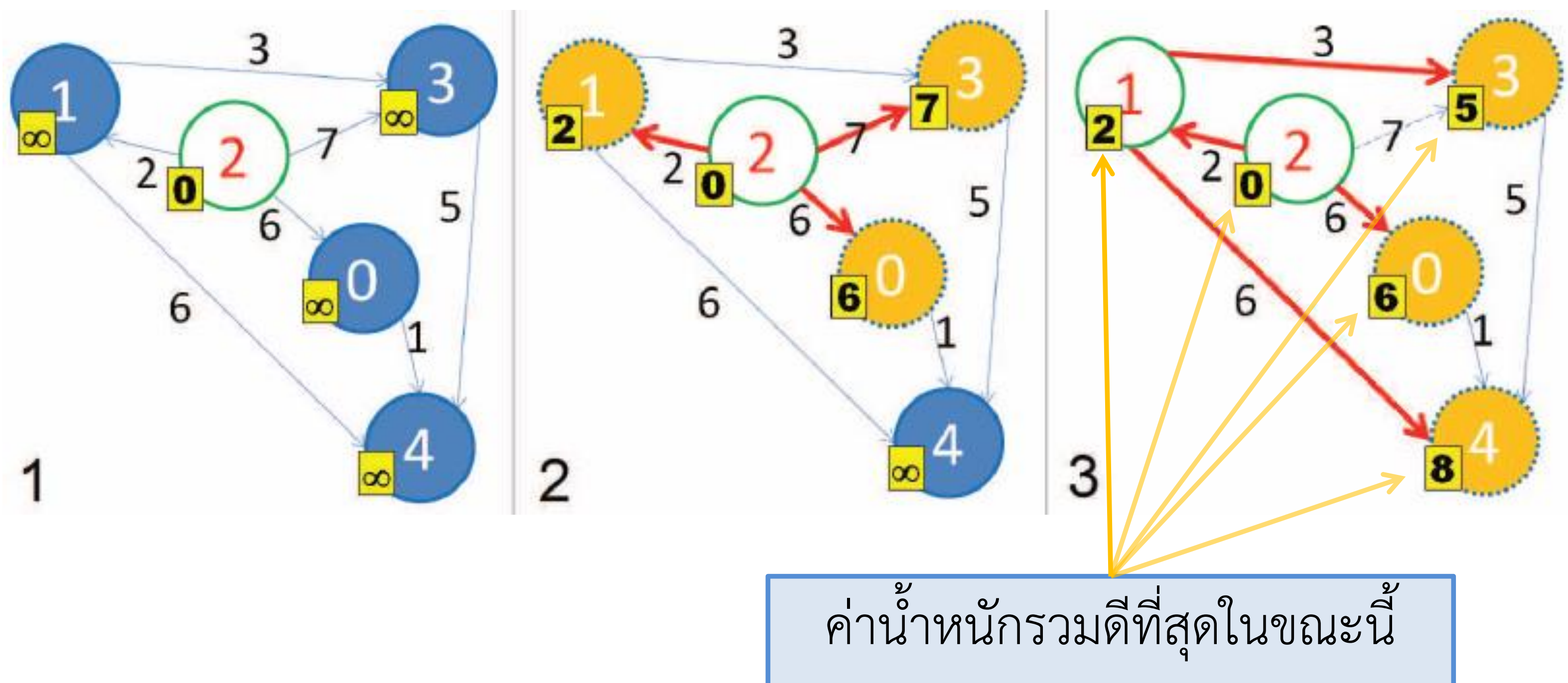
- จากจุดเริ่มต้น (ในตัวอย่างคือโหนด 2) เราจะถือว่า ณ ขณะนี้มีโหนดในกลุ่มเพียงโหนดเดียวและมีค่าน้ำหนักในการไปถึงโหนดตัวเองเท่ากับศูนย์
- โหนดอื่น ๆ ที่ยังไม่ถูกแวะเยี่ยมถือว่ามีระยะทางจากโหนดเริ่มต้นเป็นอนันต์ (ค่านี้จะถูกเปลี่ยนเป็นค่าจริงเมื่อโหนดถูกแวะเยี่ยม)
- จากโหนดเริ่มต้นให้สำรวจเส้นเชื่อมทั้งหมดแล้วเก็บไว้ในลิสต์ จากนั้นมองหาเส้นเชื่อมที่มีค่าน้ำหนักน้อยที่สุดในลิสต์ (ได้เส้นเชื่อมที่ต่อไปโหนด 1)



ภาพจาก Competitive Programming 2nd Edition

วิธีทำงานของ Dijkstra's Algorithm (2)

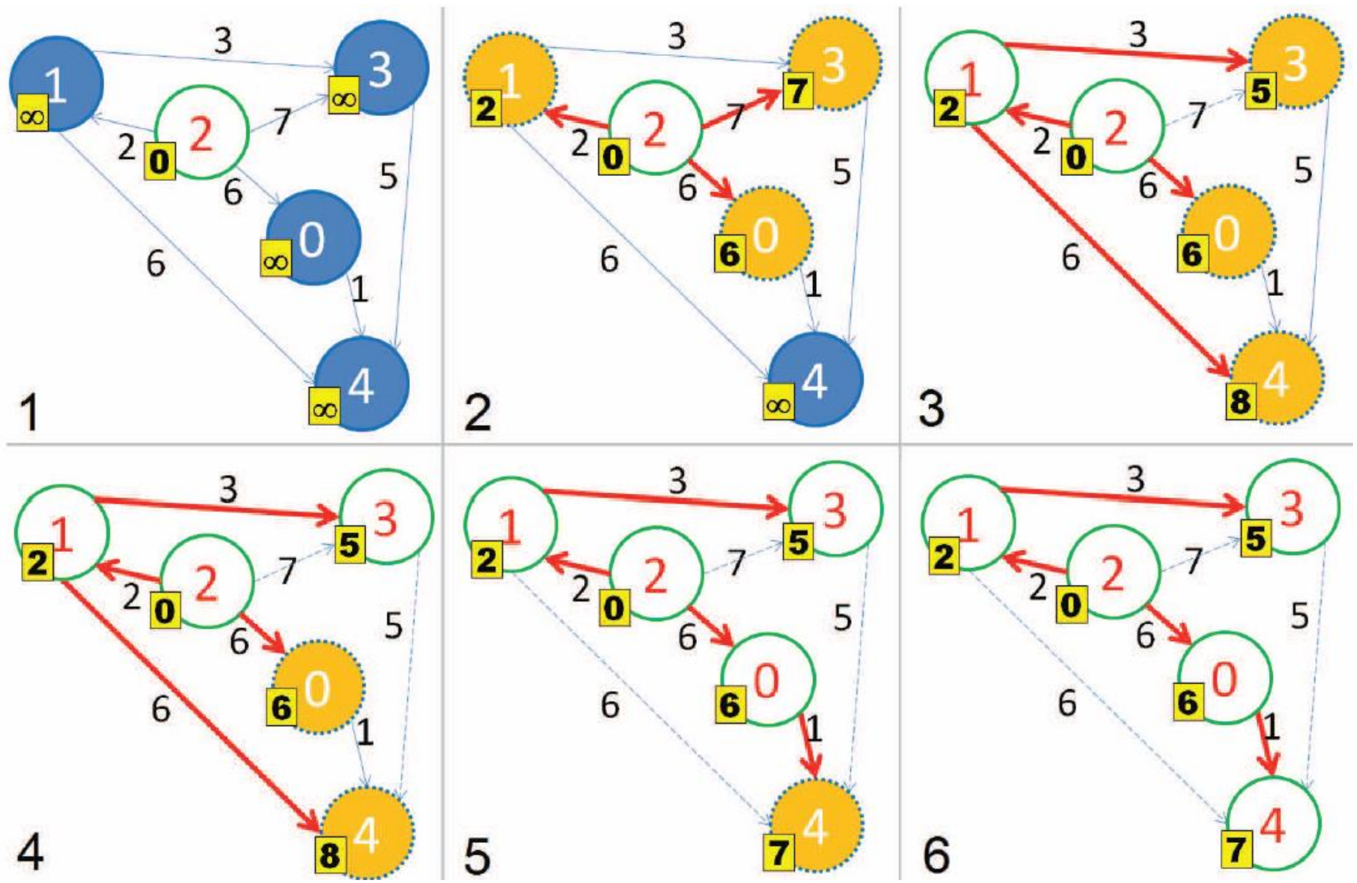
- เลือกโหนดที่ต่อกับเส้นเชื่อมนั้นเข้ามาในกลุ่ม (คือโหนดสีขาวทั้งหมด) ให้ถือว่าโหนดพวกนี้ถูกแวะเยี่ยมและได้ค่าน้ำหนักรวมเท่ากับค่าน้ำหนักจากโหนดที่มาถึงมันบวกกับเส้นเชื่อที่ใช้ ดังนั้นค่าน้ำหนักของโหนด 1 จึงเป็น 0 + 2 = 2
- เติมเส้นเชื่อมของโหนด 1 เข้าไปในลิสต์ (เส้นเชื่อมที่มีสีแดงคืออยู่ในลิสต์)
- เลือกเส้นเชื่อมที่สร้าง**ค่าน้ำหนักรวมน้อยที่สุด** และต้องเชื่อมไปยังโหนดนอกกลุ่ม



ภาพจาก Competitive Programming 2^{nd} Edition

วิธีทำงานของ Dijkstra’s Algorithm (3)

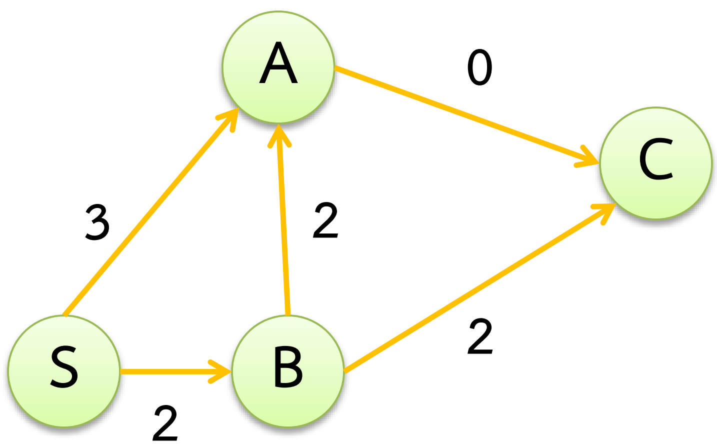
- เลือกโหนดและขยายลิสต์ของเส้นเชื่อมต่อไปเรื่อย ๆ ในทำนองเดิม จนกว่าจะไม่สามารถเพิ่มโหนดใหม่เข้ามาในกลุ่มได้อีก (ปรกติจะทดสอบจนลิสต์เส้นเชื่อมว่าง)



ภาพจาก Competitive Programming 2nd Edition

เรื่องที่คนมักเข้าใจผิดเกี่ยวกับ Dijkstra's Algorithm

- เข้าใจผิดไปว่าจะหาเส้นเชื่อมถัดไปจากเส้นเชื่อมของโหนดที่เพิ่มเติมเข้ามา
  - ➢ เรื่องนี้เกิดขึ้นบ่อย ๆ เพราะตัวอย่างจำนวนมากมีการเลือกเส้นที่ดีที่สุดแล้วไปตรงกับเส้นเชื่อมจากโหนดใหม่ที่เพิ่งได้มาพอดี (โดยเฉพาะขั้นตอนแรก ๆ)
- เข้าใจผิดไปว่าให้เลือกเส้นเชื่อมในลิสต์ที่มีค่าน้อยที่สุด (ที่จริงแล้วต้องเลือกเส้นเชื่อมที่สร้างค่าน้ำหนักรวมน้อยที่สุดถึงจะถูก)



จากตัวอย่างถ้าจุดเริ่มคือโหนด S แล้วเราหาเส้นเชื่อมที่ค่าน้อยที่สุด เราจะคำนวณเส้นทางไป A และ C ผิดพลาด โดยจะได้ค่าเป็น 4 ในขณะที่ค่าที่ควรจะเป็นคือ 3

ทำให้อัลกอริทึมมีความเร็วที่คู่ควรกับชื่อเสียง

- เราพบว่าอัลกอริทึมนี้มีการเก็บรวบรวมลิสต์ของเส้นเชื่อมไว้จำนวนมาก
- เราต้องการหาเส้นเชื่อมที่ต่อไปโหนดข้างนอกที่ทำให้เกิดน้ำหนักรวมน้อยที่สุด
  - คล้ายกับกรณีที่เราทำมาก่อนหน้าใน Prim’s Algorithm
  - ยิ่งไปกว่านั้น เส้นเชื่อมที่ใช้ไปแล้ว หรือเส้นเชื่อมที่ผิดกฎเพราะไม่เชื่อมออกไปข้างนอก Super Node ก็ควรถูกเอาออกไป
- แบบนี้ใช้ Priority Queue มาจัดการและค้นหาเส้นเชื่อมในลิสต์จะดีมาก
  - ➢ การเปรียบเทียบเพื่อจัดลำดับในลิสต์จะไม่ทำด้วย weight แต่จัดตามค่าน้ำหนักรวมที่เส้นเชื่อมทำได้

Bellman-Ford Algorithm

- เป็นอัลกอริทึมที่สามารถใช้กับกราฟที่มีค่าน้ำหนักติดลบได้
  - แต่ถ้ากราฟไม่มีค่าน้ำหนักติดลบ ใช้ Dijkstra’s จะเร็วกว่า
- ใช้หลักการ relaxation คือในตอนแรกค่าที่ได้จะเป็นค่าประมาณ แต่ค่าจะได้รับการปรับเรื่อย ๆ จนในที่สุดจะได้ค่าที่ถูกต้องและดีที่สุดสำหรับการคำนวณเส้นทางที่สั้นที่สุด
  - การ relaxation จะทำโดยพิจารณาการเปลี่ยนแปลงค่าน้ำหนักรวมที่เกิดจากเส้นเชื่อมทุกเส้น → ใช้เวลา $O(|E|)$
  - แต่การทำ relaxation จะต้องทำหลายรอบ รวมทั้งหมดเป็นตามจำนวนโหนด นั่นก็คือ $|V|$ → รวมการใช้เวลาเป็น $O(|V|\,|E|)$

หน้าตาของ Bellman-Ford Algorithm

```
// Step 1: initialize graph
// ตรงนี้เหมือน Dijkstra's algorithm คือทำให้ค่าน้ำหนักเป็นอนันต์ก่อน
// ในที่นี้ค่าน้ำหนักรวมจากจุดเริ่มต้นถึงโหนด v ถูกเก็บไว้ใน distance
for each vertex v in vertices:
   if v is source then distance[v] := 0
   else distance[v] := infinity
        predecessor[v] := null

// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
   for each edge (u, v) with weight w in edges:
      if distance[u] + w < distance[v]:
         distance[v] := distance[u] + w
         predecessor[v] := u
```

ลูปตัวด้านนอกแต่คือการ relaxation หนึ่งครั้ง

Credit: wikipedia

งานด้านในจะพิจารณาทีละเส้นเชื่อม ถ้าพบว่าเส้นเชื่อมที่ต่อจากโหนด u ไป v ให้ค่าที่ลดลงก็จะทำการอัพเดตค่า (กระบวนการอัพเดตค่าจะทำให้ได้ค่าใกล้เคียงกับคำตอบมากขึ้น)

แล้วถ้ามี Negative Cycle ล่ะ

- Bellman-Ford ได้พิสูจน์ว่า กระบวนการ relaxation จะสำเร็จและให้ค่าคำตอบที่ optimal เป็นแน่ ถ้าหากไม่มี negative cycle
- ดังนั้นจากขั้นตอนที่ 2 ถ้าเราลอง relax อีกครั้งแล้วพบว่าค่ายังลดลงได้อีก → มี negative cycle ที่สามารถไปได้จากโหนด source
- ถ้าเป็นเช่นนี้จะถือว่า Bellman-Ford Algorithm ทำงานไม่สำเร็จ เพราะไม่มีทางหาค่า optimal บางส่วนของกราฟได้
  - ➢ เพราะถ้าวนซ้ำอยู่ใน negative cycle ไปเรื่อย ๆ ค่าน้ำหนักรวมก็เบาไปเรื่อย ๆ ไม่รู้จบ

```
// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges:
   if distance[u] + w < distance[v]:
      error "Graph contains a negative-weight cycle"
```

Credit: wikipedia

All Pair Shortest Path (APSP)

- อัลกอริทึมที่ใช้บ่อยคือ Floyd-Warshall Algorithm ซึ่งเป็นไดนามิกโปรแกรมมิง
- ใช้ได้กับกราฟที่มีเส้นเชื่อมที่ติดลบ แต่ห้ามมี Negative cycle
- แนวคิด
  - ค่อย ๆ สร้างคำตอบของส่วนย่อยขึ้นมาเก็บไว้ในตาราง เช่นคำตอบของค่าน้ำหนักรวมจากโหนด u ถึง v
  - นำคำตอบของส่วนย่อยที่ได้ไปประกอบเป็นคำตอบย่อยของส่วนอื่นที่ใหญ่ขึ้น
  - การทำงานจะเริ่มจากการยอมให้ใช้เฉพาะเส้นที่ติดกับโหนดตัวเองก่อน
  - จากนั้นจะเริ่มให้ใช้ ‘ตัวช่วย’ คือเส้นทางที่เกี่ยวข้องกับโหนด 0 ก่อน และจะขยายต่อไปคือให้ใช้ของโหนด {0, 1} ต่อมาก็จะให้ใช้โหนด {0, 1, 2} เป็นเช่นนี้ไปเรื่อย ๆ จนยอมให้ใช้ตัวช่วยได้ทั้งหมด

คณิตศาสตร์ใน Floyd-Warshall Algorithm

แนวคิดที่กล่าวมา ใช้หลักการที่ว่าเส้นทางที่สั้นที่สุดจากโหนด i ไป j เมื่อยอมให้ใช้เส้นทางในหมู่โหนด {0, 1, 2, ..., k+1} จะเป็นเส้นทางที่ได้มาจากแนวทางใดแนวทางหนึ่งด้านล่างเท่านั้น

1. เส้นทางมีการใช้เฉพาะโหนดใน {0, 1, 2, ..., k}
2. นอกจากจะใช้เส้นทางในหมู่โหนด {0, 1, 2, ..., k} ยังเกี่ยวข้องกับเส้นทางที่มาจากโหนด i ไป k + 1 จากนั้นต่อด้วยเส้นทางจากโหนด k + 1 ไปถึงโหนด j

ถ้าเป็นแบบนี้เราสามารถบรรยายเส้นทางที่สั้นที่สุดจากโหนด i ไป j เมื่อยอมให้ใช้เส้นทางในหมู่โหนด {0, 1, 2, ..., k+1} ได้เป็น

$$\begin{aligned} sp(i,j,k+1) = \min(&sp(i,j,k), \\ &sp(i,k+1,k) + sp(k+1,j,k)) \end{aligned}$$

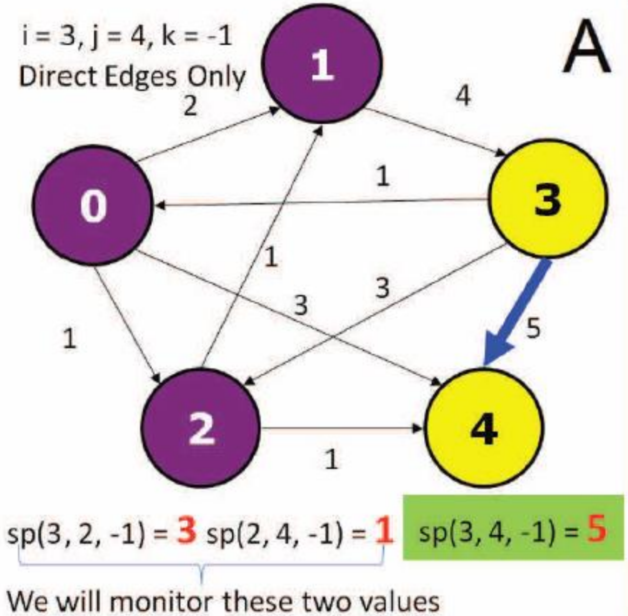
ถอดรหัส Floyd-Warshall Algorithm

- ความหมายของ Recurrence Relation ดังกล่าวก็คือ
  1. เส้นทางที่ได้มาก่อนหน้าเป็นเส้นทางที่ดีที่สุดอยู่แล้ว
  2. ลองเอาเส้นทางที่สั้นที่สุดสองอันที่เชื่อมกับโหนดตัวใหม่มาต่อกัน ถ้าเอามาต่อกันแล้วมันดีขึ้น ก็ให้ใช้อันที่มันวิ่งผ่านโหนดตัวใหม่

เนื่องจากในแต่ละรอบของการขยาย k เราต้องจับคู่โหนด i และ j ทุกรูปแบบที่เป็นไปได้ซึ่งทุกรูปแบบที่ว่าเป็น $O(|V|^2)$ ดังนั้นเมื่อต้องขยาย k จากเริ่มต้นไปจนถึง k = |V| เวลาที่ต้องใช้จึงออกมาเป็น

$$O(|V|\,|V|^2) = O(|V|^3)$$

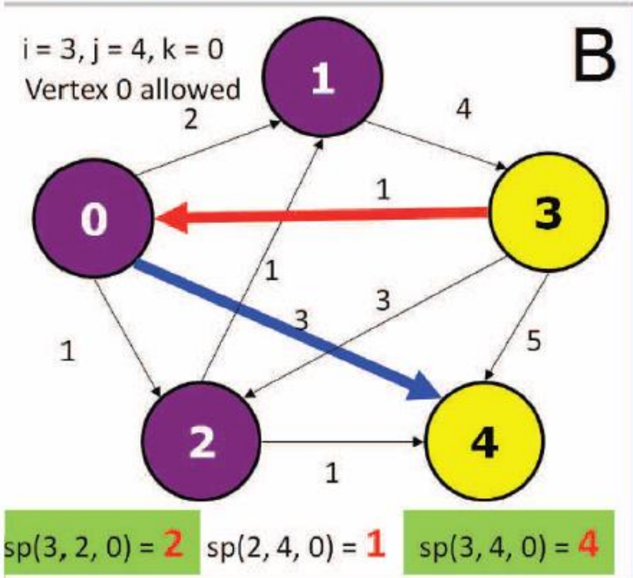
ดูตัวอย่างการทำงานของ Floyd Warshall เลยดีกว่า



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

สภาพเริ่มต้น k = -1 คือ ห้ามใช้ใครนอกจากตัวเอง และเส้นเชื่อมของมันเอง



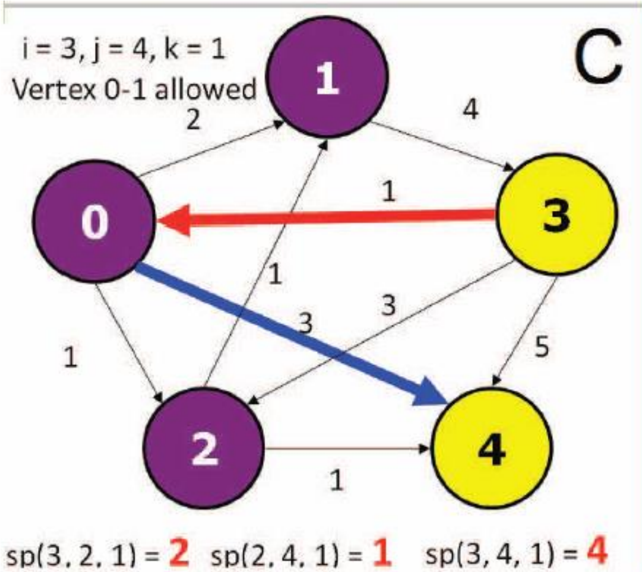
The current content of Adjacency Matrix D at **k = 0**

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

รอบต่อมา k = 0 คือยอมให้วิ่งผ่านโหนด 0 เพิ่มเติมได้ (ในที่นี้พบว่ามีคนใช้บริการโหนด 0 แล้วทำตัวดีขึ้น)

ภาพจาก Competitive Programming 2nd Edition

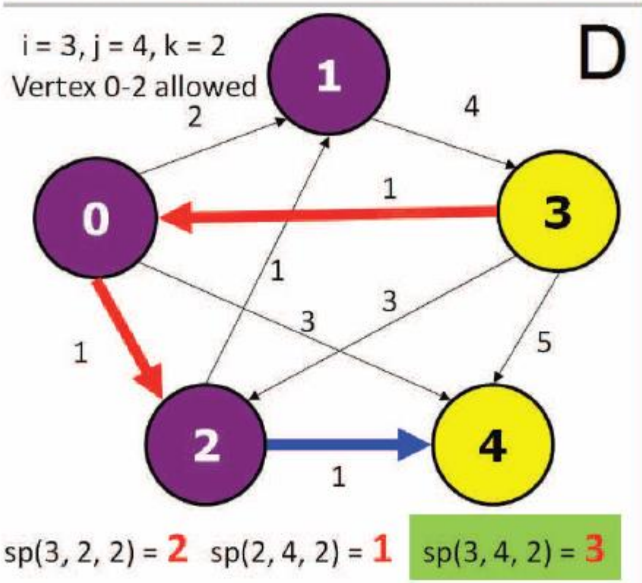
ดูตัวอย่างการทำงานของ Floyd Warshall (ต่อ)



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

คราวนี้ยอมให้วิ่งผ่านทั้งโหนด 0 และ 1



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

คราวนี้ยอมให้วิ่งผ่านทั้งโหนด 0 1 และ 2

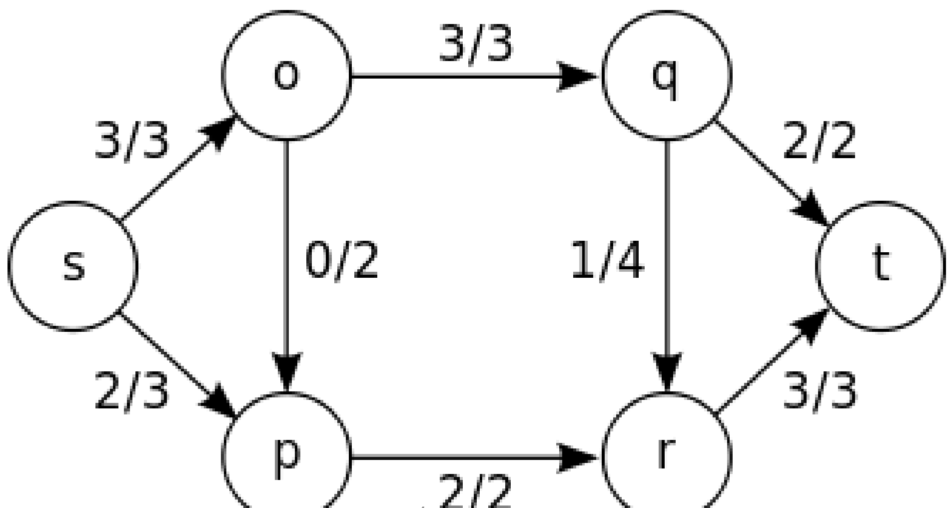
ภาพจาก Competitive Programming 2nd Edition

หัวข้อเนื้อหา

- รู้จักกราฟ
- การอธิบายโครงสร้างของกราฟ
- พื้นฐานการจัดการและใช้โครงสร้างข้อมูลกราฟ
- **อัลกอริทึมเกี่ยวกับกราฟ**
  - Bread-First Search, Depth-First Search, Topological Sort
  - Minimum Spanning Tree
  - Shortest Path Algorithm
  - Network Flow
- ตัวอย่างการประยุกต์ใช้ที่น่าสนใจ

Network Flow และ Maximum Flow

- เนื่องจากกราฟถูกมองเป็นเน็ตเวิร์คที่เชื่อมต่อกันไป
  - โหนดแต่ละโหนดอาจมองได้ว่าเป็นเครื่องคอม
  - เส้นเชื่อมที่มีค่าน้ำหนักคือขนาดข้อมูลที่ส่งข้ามไปมากันได้
  - เราต้องการทราบว่าเราสามารถส่งข้อมูลจาก source ไปยัง terminal ได้สูงสุดเท่าใดในคราวเดียว (พูดง่าย ๆ ก็คือคิดจะส่งช่วยกันหลายทาง แต่ว่ามันก็อาจจะขัดขากันเอง และดูยากว่าจะใช้เส้นทางไหนบ้างและต้องการแบนด์วิธเท่าใด)
  - ปัญหาแบบนี้เราเรียกว่า Maximum Flow



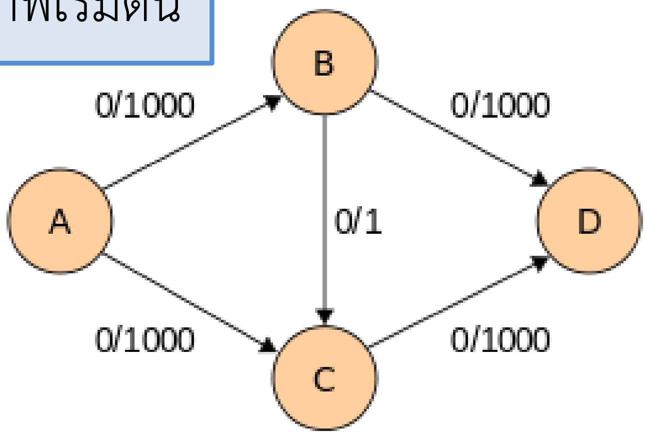
ภาพจาก wikipedia

Ford-Fulkerson’s Method

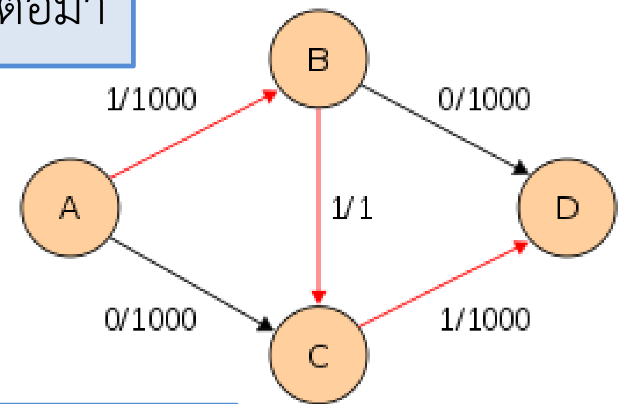
- เป็นอัลกอริทึมสำหรับการหา Maximum Flow ที่หลักการเข้าใจง่าย
- คือจากความจุของเส้นเชื่อม ถ้าเราพบว่ายังมีทาง ‘ปล่อยของ’ จาก ต้นทาง (source) ไปปลายทาง (terminal) ก็ให้ปล่อยเพิ่มไปเรื่อย ๆ จนเต็ม
- ปัญหาคือจะรู้ได้ไงว่าจะปล่อยของเพิ่มได้หรือเปล่า
  - ถ้าปล่อยได้แสดงว่ามีเส้นทางที่ยังมีความจุเหลืออยู่จากต้นทางถึงปลายทาง
  - ดังนั้นก็ให้ค่อย ๆ หาเส้นทางซ้ำ ๆ ไปเรื่อย ๆ จนกว่าจะเต็มความจุทุกเส้นทาง
- ปัญหาอีกอย่างก็คือทางที่เลือกไปก่อนหน้าอาจไม่ค่อยดี เราจึงต้องคอยปรับใหม่ ซึ่งขั้นตอนนี้จะดูงง ๆ ในตอนแรกว่ามันเป็นไปได้ยังไง

การทำงานของ Ford-Fulkerson's

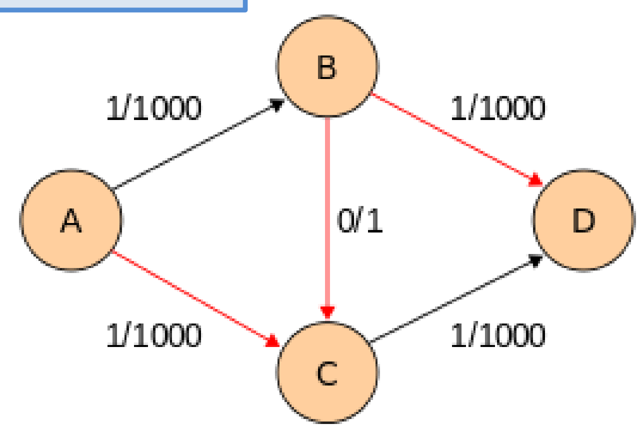
สภาพเริ่มต้น



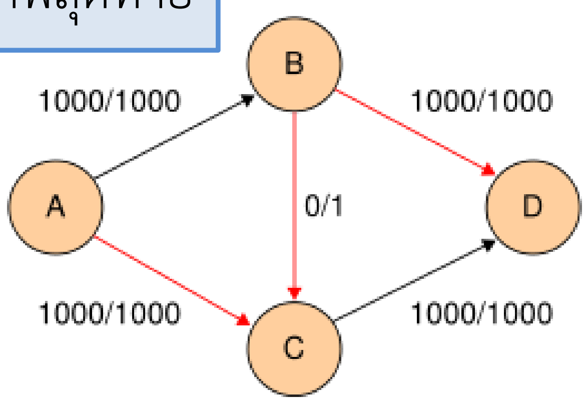
ในเวลาต่อมา



ปล่อยของเพิ่ม สังเกตด้วยว่ามีการตีกลับของเส้นทางได้ด้วย



สภาพสุดท้าย



อึม แล้วมันรู้ได้ไงว่าควรตีกลับ

- เพราะมันมีท่าไม้ตายที่ชื่อว่า Augmenting Path ซึ่งเวลาที่เราปล่อยของผ่านเส้นเชื่อม
  - มันไม่ได้แค่บันทึกว่าความจุของเส้นเชื่อมที่เหลือลดลง
  - แต่มันจะมีการเตรียมเส้นย้อนศรไว้ด้วยว่า ‘เส้นย้อนศรนี้ตีกลับได้เท่าไหร่’
  - เส้นย้อนศรนี้จะมีความจุเพิ่มขึ้นเรื่อย ๆ เมื่อเส้นเชื่อมปรกติที่คู่กับมันถูกใช้มากขึ้น
  - ดังนั้นเราไม่ต้องคิดอะไรมาก พยายามปล่อยของไปเรื่อย ๆ เดี๋ยวมันจะหาเองว่าจะใช้เส้นย้อนศรหรือเส้นปรกติเอง

แล้วไอ้เส้นทางปล่อยของนี่หากันไง

- แนวคิดของ Ford-Fulkerson โดยพื้นฐานจะจัดการกับความจุแบบจำนวนเต็ม และจะเพิ่มลดความจุของเส้นเชื่อมและเส้นย้อนศรทีละ 1
- นั่นคือปฏิบัติเหมือนกันเส้นต่าง ๆ มีความเท่าเทียมกันหมด
- เป็นแบบนี้แล้วเราก็ใช้อัลกอริทึมพื้นฐานพวก BFS หรือ DFS มาหาเส้นทางได้เลย
  - ได้ใช้เส้นทางใด ก็นำมาปรับความจุของเส้นเชื่อมและเส้นย้อนศรที่เกี่ยวข้อง
- เนื่องจากต้องปรับค่าทีละ 1 ถ้าความจุมันล้นหลาม อัลกอริทึมนี้คงพาเราไปพบกับจุดจบที่น่าเศร้า (โดยเฉพาะในการแข่งขัน)
- แต่ถ้าเราเลือกใช้ BFS และปรับวิธีเล็กน้อยมันจะก้าวผ่านข้อจำกัดนี้ได้

Edmonds-Karp's Method

- เป็นอัลกอริทึมที่เหมาะกับการหา Maximum Flow ใน Sparse Graph
- ตัวตนของมันก็คือ Ford-Fulkerson ที่ใช้ BFS ในการหาเส้นทางปล่อยของ และสามารถปล่อยของได้คราวละมาก ๆ
- ทำงานในเวลา $O(|V|\,|E|^2)$ ไม่ขึ้นกับความจุของเส้นเชื่อม
- เป็นอัลกอริทึมที่ถือว่าท้าทายมากถ้าเราจะเขียนเอง (โดยเฉพาะครั้งแรก) [แนะนำว่าให้ทำความเข้าใจแล้วพกติดตัวเข้ามาแข่ง ACM จะดีกว่า]

ก้าวข้ามขีดจำกัดของ Ford-Fulkerson

- ใช้การปรับ BFS ให้ทำอะไรบางอย่างเพิ่มไปจากเดิม
- นั่นคือในระหว่างการทำ BFS ให้เราเก็บไว้ด้วยว่าในเส้นทางการเดินทางไปถึงปลายทาง ความจุที่น้อยที่สุดคือเท่าใด
  - นั่นคือเมื่อเราทำ BFS แต่ละครั้งเสร็จ จะมีเส้นเชื่อมหรือเส้นย้อนอันใดอันหนึ่ง (หรือมากกว่า) ที่ถูกใช้จนหมดความจุทันที จุดนี้แตกต่างกับการที่ความจุจะค่อย ๆ ลดทีละหนึ่งแบบที่เกิดขึ้นใน Ford-Fulkerson
  - ถ้าหากไม่มีเส้นทางจากจุดเริ่มไปถึงปลายทางได้ แสดงว่าทุกเส้นทางที่เป็นไปได้อิ่มตัวแล้ว เราได้คำตอบสุดท้ายแล้ว
- ย้ำอีกครั้งว่าไม่ได้เขียนง่าย ๆ ควรทำความเข้าใจแล้วนำโค้ดไปดัดแปลงให้เข้ากับสถานการณ์ เพราะนี่เป็นธรรมชาติของกราฟอัลกอริทึม → ดัดแปลงโครงสร้างข้อมูลเพื่อให้เข้ากับงาน

ตัวอย่างประยุกต์ใช้งานที่น่าตื่นตาตื่นใจ

- Live-wire Algorithm for Interactive Image Segmentation
- Content-Aware Image Resizing

เรื่องอื่น ๆ

- ยังมีเนื้อหาสำคัญที่เราไม่ได้กล่าวถึงในที่นี้
  - Kruskal's algorithm (ใช้แทน Prim's ได้ และมักจะได้ผลดีใน Sparse graph รวมถึงการหา Second Best MST)
  - ใน Kruskal's algorithm มีการใช้โครงสร้างข้อมูลที่สำคัญอันหนึ่งก็คือ Set ซึ่งมีความสามารถในการ Union และ Find (ตรวจสอบว่าอยู่เซ็ตเดียวกันหรือไม่) ที่เร็วมาก (http://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf)
  - Tarjan's Strongly Connected Components ใช้ในการหาลูป (strongly connected component ในกราฟ) อัลกอริทึมตัวนี้แท้จริงมีการใช้หลายครั้งในการแข่ง ACM ทั้งที่ไทยและต่างประเทศ