บทที่ 9 GRAPH

เป็นโครงสร้างข้อมูลที่มีความสำคัญมากในทางคอมพิวเตอร์ เนื่องจากกราฟจะถูกใช้แสดงความสัมพันธ์ ที่น่าสนใจระหว่างข้อมูลแล้ว กราฟยังสามารถที่จะสร้างโมเดลปัญหาที่มีขนาดใหญ่และซับซ้อน เพื่อแก้ปัญหาได้ง่ายขึ้น เช่น

1) ในระบบขนส่งมวลชน เราอาจจะโมเดลการเดินทางของยานพาหนะ เพื่อหาเวลาในการเดินทางที่สั้นที่สุด หรือ เส้นทางที่สั้นที่สุด โดยใช้กราฟ

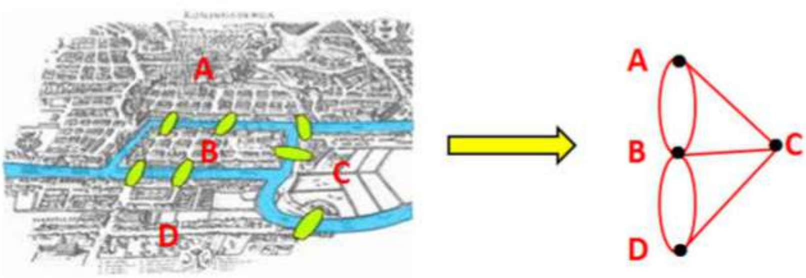
2) internet และ social netword กราฟสามารถแสดงความสัมพันธ์ระหว่างเว็บเพจหรือผู้ใช้คนอื่นๆ ได้

กราฟใช้สำหรับแสดงความสัมพันธ์ระหว่างข้อมูล

- Transportation
- Network or Internet
- Social Network

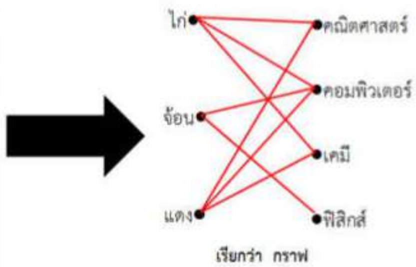
กราฟเป็นแบบจำลองทางคณิตศาสตร์ ซึ่งใช้สำหรับจำลองปัญหาบางอย่าง ด้วยแผนภาพที่ประกอบด้วยจุด และเส้นที่เชื่อมระหว่างจุด 2 จุด ตัวอย่างเช่น แผนภาพที่แสดงเส้นทางของรถไฟฟ้า, แผนภาพที่แสดงถนนที่เชื่อมเมืองต่างๆ, แผนภาพแสดงโครงสร้างวงจรไฟฟ้า, แผนภาพเครือข่ายคอมพิวเตอร์, แผนภาพแสดงเส้นทางการบิน, แผนภาพแสดงโครงสร้างทางเคมีของสารประกอบไฮโดรคาร์บอน เป็นต้น

เลออนฮาร์ด ออยเลอร์ (Leonhard Euler) นักคณิตศาสตร์ ชาวสวิส เป็นผู้ริเริ่มในการศึกษาทฤษฎีกราฟ เนื่องจากการตอบข้อคำถามเกี่ยวกับสะพานเมืองเคอนิกส์เบอร์ก โดยเมืองเคอนิกส์เบอร์กตั้งอยู่ริมฝั่งแม่น้ำพรีเกล (Pregel) ทั้งสองฝั่ง และบางส่วนเป็นเกาะ โดยมีสะพานเชื่อม 7 สะพาน คำถามคือ เป็นไปได้หรือไม่ ที่จะเดินทางรอบเมือง โดยเริ่มจากพื้นที่ใดพื้นที่หนึ่ง แล้วมาจบลงที่พื้นที่เริ่มต้น โดยการข้ามสะพานทั้ง 7 แห่ง แต่ละแห่งข้ามได้เพียงครั้งเดียว โดยจากคำถามนี้สามารถเขียนออกมาเป็นรูปได้ดังนี้



นอกจากกรณีสะพานเมืองเคอนิกส์เบอร์ก ความสัมพันธ์ของข้อมูลต่างก็สามารถเขียนออกมาในแบบกราฟดังตัวอย่างข้างล่างนี้ ด้านซ้ายมือคือตารางความสัมพันธ์ ด้านขวามือคือความสัมพันธ์ในรูปแบบกราฟ

ชื่อ	วิชา
ไก่	คณิตศาสตร์
จ้อน	คอมพิวเตอร์
แดง	คณิตศาสตร์
ไก่	เคมี
ไก่	คอมพิวเตอร์
จ้อน	ฟิสิกส์
แดง	เคมี
แดง	คอมพิวเตอร์



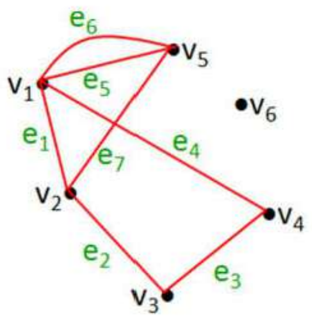
บทนิยาม กราฟ G = (V, E) ประกอบด้วย เซตจำกัด 2 เซต คือ

1. เซตที่ไม่เป็นเซตว่างของจุดยอด (Vertex) แทนด้วยสัญลักษณ์ V(G) หรือ Vertex (V) = เมือง ➔ $V \neq \varnothing$
2. เซตของเส้นเชื่อม (Edge) ที่เชื่อมระหว่างจุดยอด แทนด้วยสัญลักษณ์ E(G)

ข้อสังเกต <u>$V(G) \neq \varnothing$ แต่ E(G) อาจเป็นเซตว่างได้</u> หรือ Edge (E) = เส้น ➔ $E = \varnothing$

9.1 คำศัพท์ของทฤษฎีกราฟ

จากรูปข้างล่างนี้ เส้นที่ลากต่อระหว่างจุดหรือ**จุดยอด (Vertex)** เรียกว่า **เส้นเชื่อม (edge)**
กราฟ G = (V, E) ประกอบด้วยเซต V ซึ่งเป็นเซตของจุดต่างๆ และ E ซึ่งเป็นเซตของเส้นเชื่อมระหว่างจุด 2 จุด

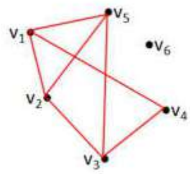


$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$
$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ หรือ
$E = \{(v_1,v_2),(v_2,v_3),(v_3,v_4),(v_1,v_4),(v_1,v_5),(v_1,v_5),(v_2,v_5)\}$

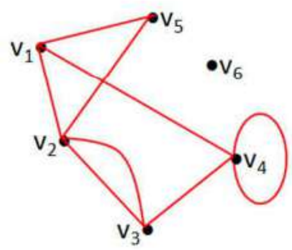
จากรูป เส้นเชื่อม **e = (a, b)** คือเส้นที่เชื่อมต่อระหว่างจุด a และจุด b จะเรียกว่าเส้นเชื่อม e
โดยที่จุด a และ จุด b เชื่อมกัน (adjacent) สามารถเขียน **e = (b, a)** ได้

จากรูป จุดประชิดกันมีเส้นเชื่อมมากกว่า 1 เส้น เรียกว่า **เส้นเชื่อมขนาน (parallel edges)**

จากรูป e =(a, a) คือเส้นเชื่อมที่ต่อจุดเดียวกัน เรียกว่า **เส้นเชื่อมวงวน (loop)**



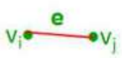
กราฟที่ไม่มีทั้งเส้นเชื่อมวงวนและเส้นเชื่อมขนาน เรียกว่า **กราฟเชิงเดียว (simple graph)**
จุดที่ไม่มีการเชื่อมกับใครเรียกว่า **จุดเอกเทศ (isolated point)** จากรูป ในกรณีนี้คือ V_6



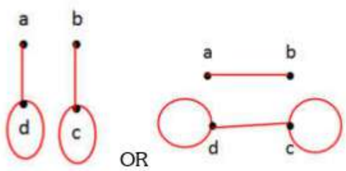
ดีกรีของจุด (Degree of Vertex) คือ ดีกรีของจุด v เขียนแทนด้วย deg(v) คือจำนวนเส้นเชื่อมที่กระทบกับจุด v สำหรับดีกรีเป็นเลขคู่ เรียก **จุดยอดคู่ (even vertex)** และดีกรีเป็นเลขคี่ เรียก **จุดยอดคี่ (odd vertex)** จากรูป

$deg(v_1) = 3$
$deg(v_2) = 4$
$deg(v_3) = 3$
$deg(v_4) = 4$ มีเส้นเชื่อมวงวน (loop) = +2
$deg(v_5) = 2$
$deg(v_6) = 0$

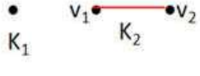
ถ้า G เป็นกราฟ มีผลรวมของดีกรีของทุกจุดในกราฟเท่ากับสองเท่าของจำนวนเส้นเชื่อม
ถ้า G คือ กราฟ แล้ว $deg(v_1)+deg(v_2)+...+deg(v_n) = 2$ (จำนวนเส้นเชื่อมของ G)
ผลรวมดีกรี = 16
จำนวนเส้นเชื่อม = 8



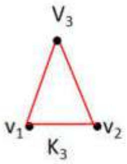
จากรูป เส้นเชื่อม e ระหว่างจุด v_i และ v_j มีค่าเป็น 1
v_i มีจำนวนดีกรี 1 และ v_j มีจำนวนดีกรี 1 ดังนั้นจำนวนดีกรีของ e ถูกนับเป็น 2
สรุป เส้นเชื่อม e ใดๆ จะถูกนับเป็นดีกรี 2 ครั้งเสมอ
ดังนั้น<u>ผลรวมดีกรีของกราฟ G = 2 x ของจำนวนเส้นเชื่อมของกราฟ G</u>

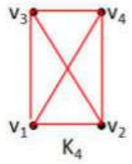


จากรูป กราฟที่มี 4 จุด แต่ละจุดมีดีกรี 1, 1, 3 และ 3 ซึ่งสามารถเขียนกราฟได้
<u>ผลรวมดีกรี = 1+1+3+3 = 8 (เลขคู่)</u>
แต่ถ้ากราฟที่มี 4 จุด แต่ละจุดมีดีกรี 1, 1, 2 และ 3 กรณีนี้ไม่สามารถเขียนเป็นกราฟได้
<u>ผลรวมดีกรี = 1+1+2+3 = 7 (เลขคี่)</u>



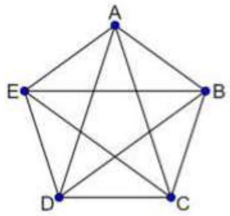
จากรูป กราฟที่มี n จุด จะเรียกกราฟว่า **กราฟบริบูรณ์ (Complete Graph)** เมื่อเป็น<u>กราฟเชิงเดียวและระหว่างจุดทุกคู่ต้องมีเส้นเชื่อม</u> หรืออาจกล่าวได้ว่า ทุกจุดได้มีเส้นเชื่อมโดยตรงกับจุดอื่นๆ ในกราฟทั้งหมด และต้องเชื่อมกันเพียงเส้นเดียวเท่านั้นดังรูป

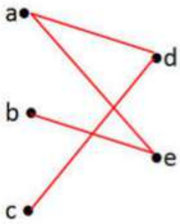




$$E = V \times \frac{V-1}{2}$$

กราฟที่เชื่อมกันทั้งหมด 4 + 3 + 2 + 1 = 10



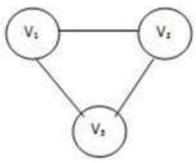


กราฟ G =(V,E) เป็น**กราฟสองส่วน (Bipartite Graph)** ถ้ามีเซตย่อย V_1 และ V_2 ของ V ซึ่งไม่เป็นเซตว่าง โดยที่ $V_1 \cap V_2 = \varnothing$ และ $V_1 \cup V_2 = V$ และแต่ละเส้นเชื่อมใน E เชื่อมระหว่างจุดหนึ่งใน V_1 และอีกจุดใน V_2 โดยหลักการคือการสมมุติให้กราฟแบ่งเป็นสองกลุ่ม และมีกฎว่าภายในกลุ่มของตัวเองจะต้องไม่มีเส้นเชื่อมถึงกันโดยตรง โดยพิจารณาดังนี้ จากรูป a เชื่อม d กับ e ดังนั้นถ้า a อยู่กลุ่มไหนห้ามมี d กับ e และ b เชื่อม e ดังนั้น b อยู่กลุ่มไหนห้ามมี e และ c เชื่อม d ดังนั้น c อยู่กลุ่มไหนห้ามมี d ซึ่ง a, b, c จึงจัดอยู่ในกลุ่มเดี่ยวกัน ให้จัดอยู่กลุ่มเดี่ยวกัน โดยพิจารณาจากตัวห้าม

V = {a, b, c, d, e}

V_1= {a, b, c} คือ a จะไม่เชื่อมกับเซตของตัวเองเสมือนแบ่งเป็นสองชุด

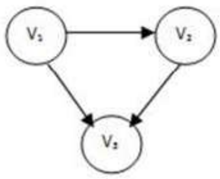
V_2 = {d, e} คือ d จะไม่เชื่อมกับเซตของตัวเองเสมือนแบ่งเป็นสองชุด



กราฟไม่มีทิศทาง (Undirected Graph)

Vertex = v1, v2, v3

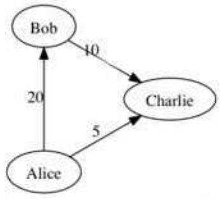
Edge = { (v1, v2) (v2, v1) (v2, v3) (v3, v2) (v1, v3) (v3, v1) }



กราฟมีทิศทาง (Directed Graph)

Vertex = v1, v2, v3

Edge = { (v1, v2) (v2, v3) (v2, v3) }

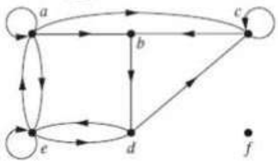


Weighted Graph

Bob = B, Charlie = C, Alice = A

Vertex = B, A, C

Edge = { (A, B, 20), (B, C, 10), (A, C, 5) }



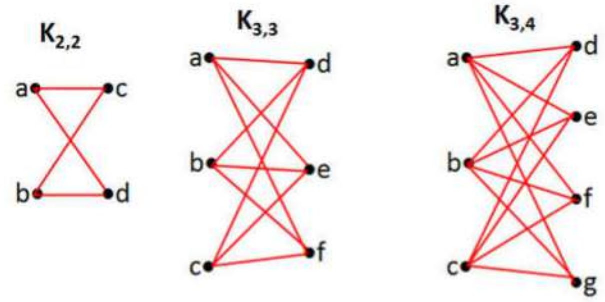
In and Out Degree (มีเฉพาะกราฟที่มีทิศทาง)

In degree = deg(a) = 2, deg(b) = 2, deg(c) = 3, deg(d) = 2, deg(e) = 3, deg(f) = 0

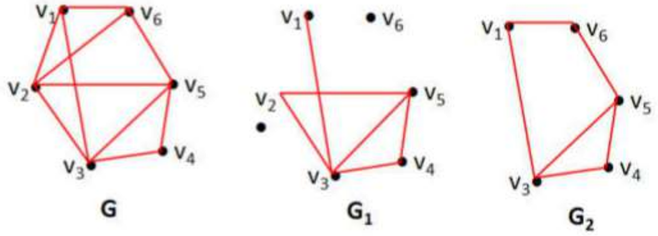
Out degree = deg(a) = 4, deg(b) = 1, deg(c) = 2, deg(d) = 2, deg(e) = 3, deg(f) = 0

Degree = In degree + Out degree = 12+ 12=24

จากรูปข้างล่าง กราฟ G = (V,E) เป็น**กราฟสองส่วนบริบูรณ์ (Complete bipartite graph)** เมื่อ G เป็นกราฟสองส่วน และแต่ละจุดในเซตย่อย V_1 ของ V ต้องมีเส้นเชื่อมไปยังทุกจุดในเซตย่อย V_2 ของ V เขียนแทนด้วย $K_{m,n}$ เมื่อ m และ n เป็นจำนวนสมาชิกของ V_1 และ V_2 ตามลำดับ โดยการพิจารณาคือ ชุดแรกต้องเชื่อมกับชุดที่สองให้ครบจึงเป็นกราฟสองส่วนบริบูรณ์ ตัวอย่างเช่น กราฟซ้ายมือ สมาชิกชุดแรก คือ a สามารถเชื่อม c กับ d ซึ่งเป็นสมาชิกในชุดที่สองทั้งหมด และสมาชิกชุดสอง คือ c สามารถเชื่อม a กับ b ซึ่งเป็นสมาชิกในชุดแรกทั้งหมด

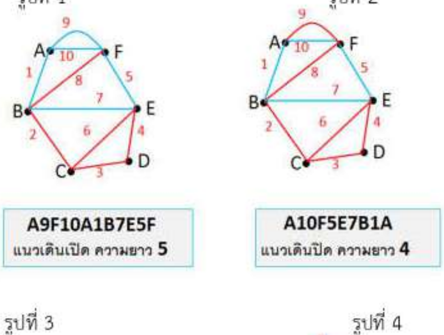


กราฟ $G_1 = (V_1,E_1)$ เป็น**กราฟย่อยของกราฟ** G =(V,E) ถ้า $V_1 \subset V$ และ $E_1 \subset E$ และ เส้นเชื่อม $e \in E_1$ มีจุดกระทบทั้งสองอยู่ใน V_1 ตัวอย่างเช่น กราฟ G ของรูปข้างล่างนี้ สามารถเขียนออกมาเป็นกราฟย่อย G_1 และ G_2 โดย G_1 และ G_2 เมื่อประกอบกันแล้วจะเป็นกราฟ G

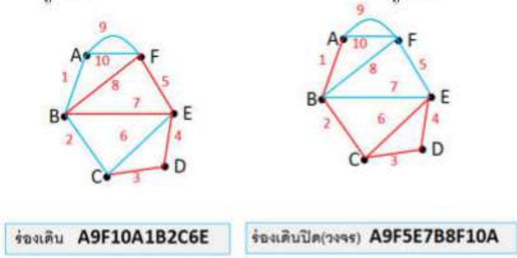


แนวเดิน (walk) คืออันดับจำกัดของจุดและเส้นเชื่อมสลับกัน ดังนี้ $v_0e_1v_1e_2...v_ne_n$ โดย $e_i = (v_{i-1}, v_i)$ และ $i \in \{1,2,...,n\}$ เรียก v_0 ว่า จุดเริ่มต้นและเรียก v_n ว่าจุดสิ้นสุดของแนวเดิน เรียก $v_1,v_2,...,v_{n-1}$ ว่าจุดภายในของแนวเดิน เรียกแนวเดินดังกล่าวว่า แนวเดิน v_0-v_n ตัวอย่างเช่น จากรูปข้างบนด้านซ้ายมือ A ไปหา F ด้วยเส้นทาง 10 และ F ไปหา E ด้วยเส้นทาง 5 และ E ไปหา B ด้วยเส้นทาง 7 ตามลำดับ เป็นต้น

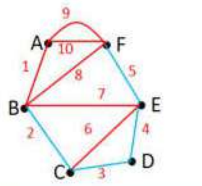
รูปที่ 1 รูปที่ 2



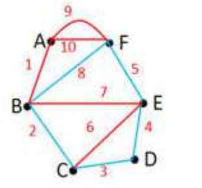
รูปที่ 3 รูปที่ 4



รูปที่ 5 รูปที่ 6

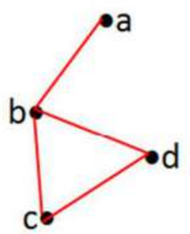


วิถี **B2C3D4E5F**

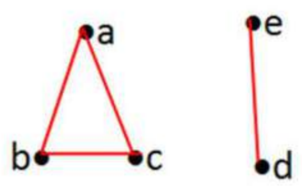


วิถีปิด(วัฏจักร) **B2C3D4E5F8B**

- แนวเดินที่มีเส้นเชื่อม n เส้น เรียกว่า **แนวเดินที่มีความยาวเท่ากับ n** ตัวอย่างเช่น รูปที่ 1 และ รูปที่ 2 มีความยาวเท่ากับ 5 และ 4 ตามลำดับ
- แนวเดินที่มีจุดเริ่มต้นกับจุดสิ้นสุดแตกต่างกัน เรียกว่า **แนวเดินเปิด**
  ตัวอย่างเช่น รูปที่ 1 จุดเริ่มต้นที่ A แล้วจุดสิ้นสุดที่ F
- แนวเดินที่มีจุดเริ่มต้นกับจุดสิ้นสุดเป็นจุดเดียวกัน เรียกว่า **แนวเดินปิด**
  ตัวอย่างเช่น รูปที่ 2 จุดเริ่มต้นที่ A แล้วจุดสิ้นสุดที่ A
- แนวเดินที่เส้นเชื่อมทุกเส้นในอันดับแตกต่างกัน (V ซ้ำได้แต่ E ห้ามซ้ำ) เรียกว่า **แนวเดินไม่ซ้ำหรือร่องเดิน (trail)**
  เพราะปกติแนวเดินนั้น สามารถซ้ำเส้นทางเดิมได้ (V ซ้ำได้และ E ซ้ำได้) ตัวอย่างเช่น A9F9A1B7E กรณีนี้คือ A ไป F และ F ไป A กรณีนี้มีเส้นทางที่ซ้ำเส้นทางเดิม ซึ่งกรณีนี้ถึงว่าไม่เป็นร่องเดิน แต่รูปที่ 3 นั้น เป็นร่องเดิน เพราะไม่มีเส้นเชื่อมที่ซ้ำเส้นเดิม
- ร่องเดินแบบปิดที่มีความยาวตั้งแต่ 3 ขึ้นไป (V ซ้ำได้แต่ E ห้ามซ้ำ และกลับที่จุดเริ่มต้น) เรียกว่า **วงจร (circuit)**
  ตัวอย่างเช่นรูปที่ 4 นั้น เป็นร่องเดิน ที่จุดเริ่มต้นกับจุดสิ้นสุดเป็นจุดเดี่ยวกัน คือจุดเริ่มต้นคือจุด A และจุดสิ้นสุดคือจุด A
- ร่องเดินที่จุดในอันดับทุกจุดแตกต่างกัน (V ห้ามซ้ำและ E ห้ามซ้ำ) เรียกว่า **วิถี (path)**
  ตัวอย่างเช่นรูปที่ 5 นั้น เป็นวิถี เพราะไม่มีเส้นเชื่อมที่ซ้ำเส้นเดิม และไม่มีจุดยอดใดซ้ำกันเลย
  เนื่องจากอันดับของจุดและเส้นเชื่อมที่เป็นวิถี ไม่ซ้ำกัน
  จึงสามารถเขียนแทนวิถีในรูปที่ 5 คือ B2C3D4E5F ด้วยอันดับของจุด BCDEF แบบนี้ได้
- วิถีแบบปิด (วิถีที่ยกเว้นจุดเริ่มต้นและจุดสิ้นสุดซ้ำกัน หรือ V ห้ามซ้ำและ E ห้ามซ้ำ และกลับที่จุดเริ่มต้น) ที่มีความยาวตั้งแต่ 3 ขึ้นไป เรียกว่า **วัฏจักร (cycle)**
  ตัวอย่างเช่นรูปที่ 6 เป็นวัฏจักร เพราะไม่มีเส้นเชื่อมที่ซ้ำเส้นเดิม และไม่มีจุดยอดใดซ้ำกันเลย มีจุดเริ่มต้นกับจุดสิ้นสุดเป็นจุดเดี่ยวกัน คือจุดเริ่มต้นคือจุด A และจุดสิ้นสุดคือจุด A
- ระยะห่างระหว่างจุด a และ b คือ **ความยาวของวิถีจากจุด a ไปยังจุด b ที่สั้นที่สุด**
  ตัวอย่างเช่น A ไป E ระยะห่างระหว่างจุด A และ E คือ 8 โดยวิถีคือ A1B7E ระยะห่างคือ 1 + 7 = 8

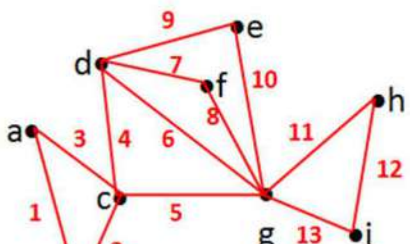


เชื่อมโยง



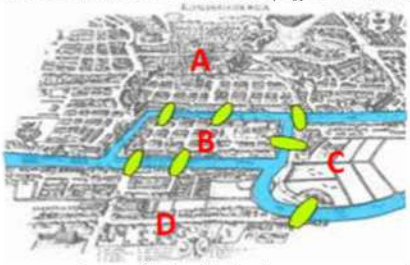
ไม่เชื่อมโยง

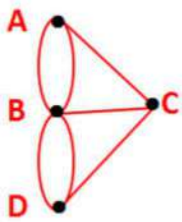
ให้ a และ b เป็นจุดในกราฟ จุด a เชื่อมโยง (connect) กับจุด b ก็ต่อเมื่อ มีแนวเดินจากจุด a ไปยังจุด b กราฟ G เป็น**กราฟเชื่อมโยง (connected graph)** ก็ต่อเมื่อทุกๆ คู่ของจุดในกราฟ G ต้องเชื่อมโยงกัน หรืออาจกล่าวได้ว่า จุดใดๆ บนกราฟสามารถไปถึงจุดใดบนกราฟได้ทุกจุด เช่น กราฟบนด้านซ้ายมือ a มีเส้นทางที่สามารถไปหา b, c, d ได้ จึงเป็นกราฟเชื่อมโยง แต่กราฟบนด้านขวามือ a ไม่มีเส้นทางที่สามารถไปหา e กับ d ได้ จึงไม่เป็นกราฟเชื่อมโยง



- **วงจรออยเลอร์ (Euler Circuit)** คือ วงจรที่ผ่านเส้นเชื่อมทุกเส้นแต่เพียงหนึ่งครั้งเท่านั้น และทุกจุดยอด โดยเริ่มจากจุดหนึ่งไปตามเส้นเชื่อมต่างๆ แล้วสามารถกลับมาที่จุดเริ่มต้นได้ (V ซ้ำได้แต่ E ห้ามซ้ำ และกลับที่จุดเริ่มต้น และต้องผ่าน E ให้ครบ)
- **ทางเดินออยเลอร์หรือเส้นทางออยเลอร์ (Euleria Tail)** คือ อันดับเส้นเชื่อม โดยเส้นทางที่ลากผ่านเส้นต่างๆ ในกราฟ โดยแต่ละเส้นลากผ่านได้เพียงครั้งเดียว จากรูปข้างบน คือ 1 2 5 13 12 11 8 7 9 10 6 4 3 โดยเริ่มที่จุด a จบลงที่จุด a และเส้นเชื่อมไม่ซ้ำกันทุกเส้นเชื่อมในกราฟ
- **กราฟออยเลอร์ (Eulerian Graph)** คือ กราฟที่มีวงจรออยเลอร์ ซึ่งกราฟออยเลอร์ นั้นเมื่อนับดีกรีของจุดจะได้คราวละ 2 เสมอ เพราะส่วนจุดเริ่มต้นเมื่อมีเส้นเชื่อมออกไป ในที่สุดจะมีเส้นเชื่อมกลับเข้ามา ซึ่งทำให้ทุกๆ จุดของกราฟเป็นจุดยอดคู่ ตัวอย่างเช่น deg(a) = 2, deg(b) = 2, deg(c) = 4, deg(d) = 4, deg(e) = 2, deg(f) = 2, deg(g) = 6, deg(h) = 2, deg(i) = 2 ดังนั้นกราฟนี้เป็นกราฟออยเลอร์ เพราะดีกรีของจุดมีค่าเป็นจุดยอดคู่ทั้งหมด

คำถามเกี่ยวกับสะพานเมืองเคอนิกส์เบอร์ก ตอบโดยทฤษฎีกราฟ พื้นที่เมืองแบ่งเป็น 4 ส่วน แทนด้วยจุด A B C D และแทนสะพานด้วยเส้นเชื่อม





การที่จะเดินทางรอบเมือง โดยเริ่มจากพื้นที่ใดพื้นที่หนึ่ง แล้วมาจบลงที่พื้นที่เริ่มต้น โดยการข้ามสะพานทั้ง 7 แห่ง แต่ละแห่งข้ามได้เพียงครั้งเดียว คือการหาวงจรที่ผ่านเส้นเชื่อมทุกเส้นนั่นคือ วงจรออยเลอร์ เนื่องจากกราฟที่ใช้แทนมีจุดยอดคี่ เช่น deg(C) = 3 ดังนั้นกราฟนี้ไม่เป็นกราฟออยเลอร์ จึงไม่มีวงจรออยเลอร์ คำตอบคือ เป็นไปไม่ได้

8.2 โครงสร้างข้อมูลสำหรับกราฟ

1) Adjacency Matrix (เมตริกซ์ประชิด)

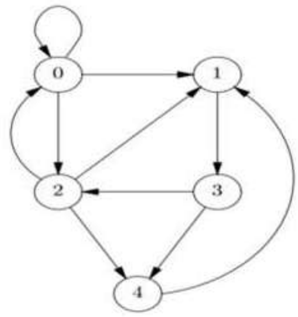
- Index of Array = Vertex
- Value of Array = Edge
- Benefit = easy, speed
- Bans = use a lot of memory
- มักใช้ Array 2 มิติ แทน Vertex ที่เชื่อมต่อกันด้วย Edge
- ข้อดีคือจัดการง่ายและทำงานได้รวดเร็ว
- ข้อเสียคือสิ้นเปลื่องหน่วยความจำ โดยเฉพาะอย่างยิ่ง sparse graph

เมตริกประชิดสำหรับกราฟ

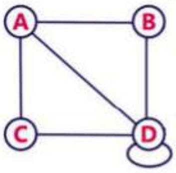
เวอร์เท็กซ์

เวอร์เท็กซ์	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

edges[][]

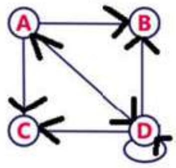


1.1) Undirected Graph [ขาไป = ขากลับ] + ไม่มีน้ำหนัก



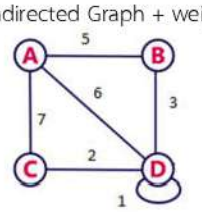
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	1

1.2) Directed Graph [ขาไป ≠ ขากลับ] + ไม่มีน้ำหนัก



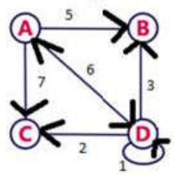
	A	B	C	D
A	0	1	1	1
B	0	0	0	0
C	0	0	0	0
D	1	1	1	1

1.3) Undirected Graph + weight [ขาไป = ขากลับ] + มีน้ำหนัก



	A	B	C	D
A	0	5	7	6
B	5	0	0	3
C	7	0	0	2
D	6	3	2	1

1.4) Directed Graph + weight [ขาไป ≠ ขากลับ] + มีน้ำหนัก



	A	B	C	D
A	0	5	7	6
B	0	0	0	0
C	0	0	0	0
D	6	3	2	1

CODE

class graph { public: int edges[1000][1000]; int n_vertices; int n_edges;	//Node //Line

void print_graph()
{
for(int i=0 ; i < n_vertices ; i++)
{
cout<<i<<" : ";
for(int j=0 ; j <= n_vertices ; j++){if(edges[i][j] > 0){cout<<j<<" ";}}
cout<<endl;
}
}
	แสดงกราฟ
	//0 : 1 2 3
	//1 : 0 3
	//2 : 0 3
	//3 : 0 1 2 3

void initial_graph(int n)
{
n_vertices = n;
n_edges = 0;
for(int i=0;i<n;i++)
{
for(int j=0;j<n;j++)
{
edges[i][j] = 0;
}
}
}
	สร้าง array ขนาด 4 * 4 ให้ทุกช่องเป็น 0

void insert_graph(int x, int y)
{
edges[x][y] = 1;
edges[y][x] = 1;
n_edges++;
}
	กำหนดช่องนั้นเป็น 1แต่เนื่องจากมีช่องเชื่อม 2 ทางเลยต้องทำทั้ง x ไป y และ y ไป x
	Undirected Graph

void insert_graph(int x, int y)
{
edges[x][y] = 1;
n_edges++;
}
	กำหนดช่องนั้นเป็น 1แต่เนื่องจากมีช่องเชื่อม 2 ทางเลยต้องทำทั้ง x ไป y เท่านั้น
	Directed Graph

void insert_graph(int x,int y,int v)
{
edges[x][y] = v;
edges[y][x] = v;
n_edges++;
}
	Undirected Graph + weight และใส่ น้ำหนักให้มัน

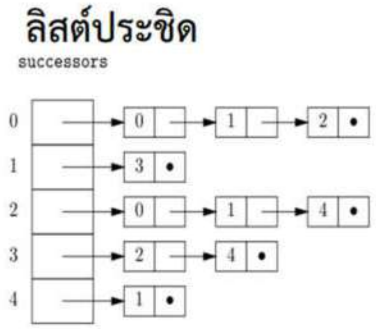
void insert_graph(int x,int y,int v)
{
edges[x][y] = v;
n_edges++;
}
	กำหนดช่องนั้นเป็น 1แต่เนื่องจากมีช่องเชื่อม 2 ทางเลยต้องทำทั้ง x ไป y เท่านั้น
	Directed Graph + weight และใส่ น้ำหนักให้มัน

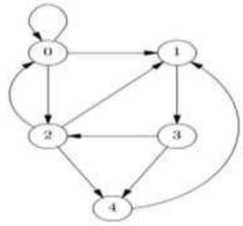
};	

graph *g = new graph();
g->initial_graph(4);
g->insert_graph(0, 1);
g->insert_graph(0, 2);
g->insert_graph(0, 3);
g->insert_graph(1, 3);
g->insert_graph(2, 3);
g->insert_graph(3, 3);
g->print_graph();
	กำหนดตรงช่องนั้นเป็น 1 ถ้าเป็น 1 แสดงว่า สามารถเชื่อมต่อกันได้
	ถ้ากรณีมี weight ก็จะใส่ ช่องเชื่อมด้วย weight

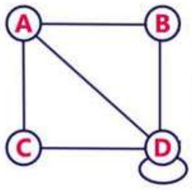
2) Adjacency List (ลิสต์ประชิด)

- Link-List + array 1 dimension
- Benefit = save memory
- Bans = hard manage
- ใช้ลิสต์ร่วมกับอาร์เรย์ขนาด 1 มิติเพื่อเก็บ egde ระหว่าง Vertex
- ข้อดีคือ ประหยัดหน่วยความจำที่ใช้ในการแสดงกราฟขนาดใหญ่
- ข้อเสีย คือ ยุ่งยากในการจัดการกว่าเมตริกซ์ประชิด



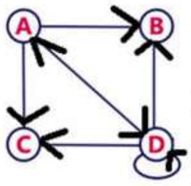


2.1) Undirected Graph [ขาไป = ขากลับ] + ไม่มีน้ำหนัก



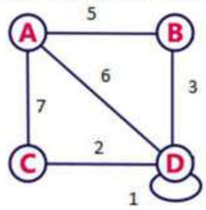
A → B → C → D → NULL
B → A → D → NULL
C → A → D → NULL
D → A → B → C → D → NULL

2.2) Directed Graph [ขาไป ≠ ขากลับ] + ไม่มีน้ำหนัก



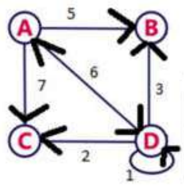
A → B → C → D →NULL
B → NULL
C → NULL
D → A → B → C → D →NULL

2.3) Undirected Graph + weight [ขาไป = ขากลับ] + มีน้ำหนัก



A → B=5 → C=7 → D=6 NULL
B → A=5 → D=3 NULL
C → A=7 → D=2 NULL
D → A=6 → B=3 → C=2 → D=1 NULL

2.4) Directed Graph + weight [ขาไป ≠ ขากลับ] + มีน้ำหนัก



A → B=5 → C=7 → D=6 NULL
B → NULL
C → NULL
D → A=6 → B=3 → C=2 → D=1 NULL

```
class node
{
   public:
   int item;
   node *next;
   node(int i) { item = i; next = NULL; }
   node() { item = 0; next = NULL; }
};
```

```
class linkgraph
{
   public:
   int n_vertices;
   int n_edges;
   node* edges[100];
```

```
   void print_graph()
   {
      node *r;
      for( int i=0 ; i < n_vertices ; i++)
      {
         r = edges[i];
         cout<<i<<" : ";
         while( r->next != NULL ){cout<<r->next->item<<" "; r = r->next;}
         cout<<endl;
      }
   }
```

```
//0 : 1 2 3
//1 : 3 0
//2 : 3 0
//3 : 0 1 2 3
```

```
   void initial_graph(int n)
   {
      n_vertices = n;
      n_edges = 0;
      for(int i=0 ; i<n ; i++){ edges[i] = new node(); }
   }
```

เป็นแถวหลักที่ใช้สำหรับเชื่อมต่อ

```
   void insert_graph( int x, int y )
   {
      node *p, *r;
      p = new node(y);
      r = edges[x];
      while( r->next != NULL ){r = r->next;}
      r->next = p;
   }
```

```
   void insert_graph( int x, int y, int z )
   {
      node *p, *r;
      p = new node(y,z);
      r = edges[x];
      while( r->next != NULL )
      {
         r = r->next;
      }
      r->next = p;
   }
```

ใส่ขนาด กรณีกราฟมีขนาด

```
};
```

```
   linkgraph *g = new linkgraph();
   g->initial_graph(4);
   g->insert_graph( 0, 1);  g->insert_graph( 1, 0);
   g->insert_graph( 0, 2);  g->insert_graph( 2, 0);
   g->insert_graph( 0, 3);  g->insert_graph( 3, 0);
   g->insert_graph( 1, 3); g->insert_graph( 3, 1);
   g->insert_graph( 2, 3); g->insert_graph( 3, 2);
   g->insert_graph( 3, 3);
   g->print_graph();
```

กรณีกราฟไม่มีทิศทาง ต้อง add ไปกลับ

```
//   g->insert_graph( 0, 1);
//   g->insert_graph( 0, 2);
//   g->insert_graph( 0, 3);
//   g->insert_graph( 3, 0);
//   g->insert_graph( 3, 1);
//   g->insert_graph( 3, 2);
//   g->insert_graph( 3, 3);
```

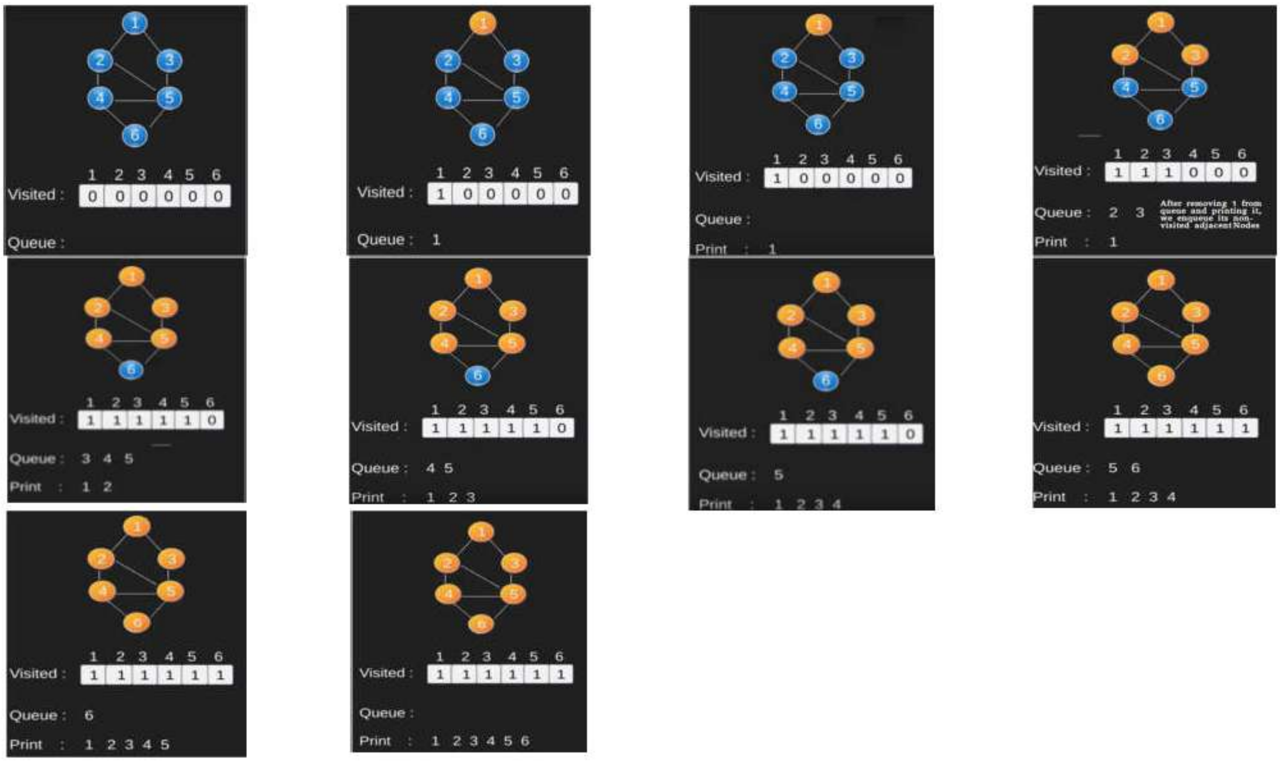
กรณีกราฟมีทิศทาง

8.3) การสำรวจกราฟ (Graph Traversal)

- การเข้าถึงทุก Node ของกราฟ
- ปัญหาพื้นฐานที่สำคัญที่สุดของกราฟ คือ ทำอย่างไรจึงจะเข้าถึงแต่ละโหนดในกราฟ หรือที่รู้จักกันว่าเป็นการเยี่ยมโหนด
- 2 เทคนิคที่นิยมใช้ในการสำรวจกราฟ

1) Breadth first search (BFS)

เริ่มต้น = เอาตัวเริ่มต้นใส่เข้าไปใน queue + set เป็น visit แล้วแตกไปทุก node ที่เชื่อม ถ้าเจอตัวไหนที่ยังไม่ visit ให้ใส่เข้าไปใน queue + set เป็น visit เมื่อออกจาก loop ให้เอาตัวออกจาก queue เมื่อตัวไหนถูกเอาออกให้แตก node นั้นต่อ



BFS สามารถประยุกต์หา<u>เส้นทางที่สั้นที่สุด</u>ที่เชื่อมต่อทุก node ในกรณี Graph no weight โดยจากการที่ vertex แตกไปทุกๆ โนด Shortest path

a->b =1

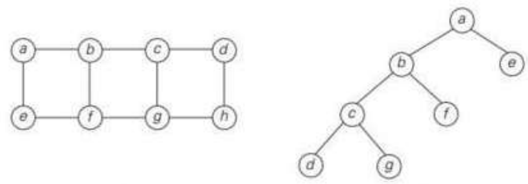
a->e=1

a->c=2

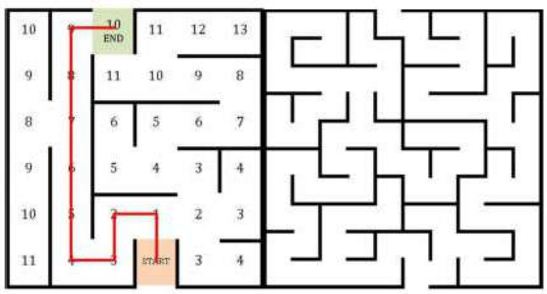
a->d=3

a->g=3

BFS สามารถนำมาใช้แก้ปัญหา <u>Maze exploration</u>



(a) (b)



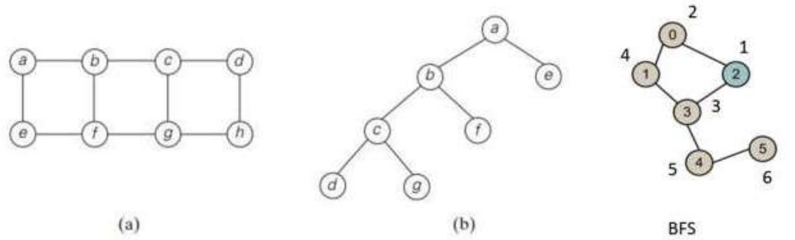
void BFS (int start)	
{	
vector <int> queue;	Clear queue
bool *visited = new bool[1000];	Clear visit
for(int i = 0 ; i < 1000 ; i++){ visited[i] = false; }	
visited[start] = true;	เริ่มเมือง 1 visited[0] = true ไปแล้ว index ไหนที่ถูก visit แล้วเป็น true
queue.push_back(start);	ใน queue ใส่ 0 เข้าไป โดยรูปแบบของต้นไม้ดังนี้
	0 : 1 2
	1 : 2
	2 : 0 3
	3 : 3
while(queue.empty() == false)	
{	
start = queue.front();	ปริ้งตัวแรกและลบตัวแรก
cout << start << " ";	เริ่มต้น node ที่ปริ้งวิ่งหมด path เช่น เริ่มที่ 0 วิ่งไป 1 กับ 2
queue.erase(queue.begin());	
node *r = edges[start];	
while(r->next != NULL)	
{	
if (!visited[r->next->item])	ไม่เคย visit ใส่เข้าไป
{	
visited[r->next->item] = true;	กลายเป็น visit แล้ว
queue.push_back(r->next->item);	ใส่ใน queue
}	
r = r->next;	
}	
}	
}	
linkgraph *g = new linkgraph();	
g->initial_graph(4);	
g->insert_graph(0, 1);	
g->insert_graph(0, 2);	
g->insert_graph(1, 2);	
g->insert_graph(2, 0);	
g->insert_graph(2, 3);	
g->insert_graph(3, 3);	
g->print_graph();	
g->BFS(0);	เริ่มต้นที่ 2 ผลลัพธ์ 2 0 3 1

การประยุกต์ของ BFS

ตรวจสอบ conectivity ของกราฟ

- เนื่องจาก BFS สิ้นสุดการทำงานเมื่อทุกๆ vertex ที่เชื่อมต่อกับ vertex ตั้งต้นได้รับการพิจารณา การตรวจสอบ connectivity สามารถทำได้โดยใช้ BFS traversal ซึ่งเริ่มจาก vertex ใดก็ได้ เมื่อ algorithm สิ้นสุดการทำงานให้ตรวจสอบว่า ทุกๆ vertex ใน graph ได้รับการพิจารณาแล้วหรือไม่ graph จะถือว่า connected ถ้าทุกๆ vertex ถูก visited
- ค้นหาพาสที่สั้นที่สุด (shortest path) สำรวจต้นไม้ BFS จะทำให้ได้ path ที่มีความยาวที่สั้นที่สุดจาก source vertex ไปยังทุกๆ vertex

Finding minimum-edge path



BFS

visit	q
-	f เริ่มจาก f
f	e,b,g โดย f เชื่อม e b g เลือก e (ซ้ายไปขวา บนลงล่าง)
e	b,g,a เอา e ออกไปใน list เพิ่ม a ต่อแถว
b	g,a,c
g	a,c,h
a	c,h
c	h,a
h	d
d	-

BFS

visit	q
-	f เริ่มจาก a
a	b,e
b	e,f,c
e	f,c
f	c,g
c	g,d
g	d,h
h	-

shortest path

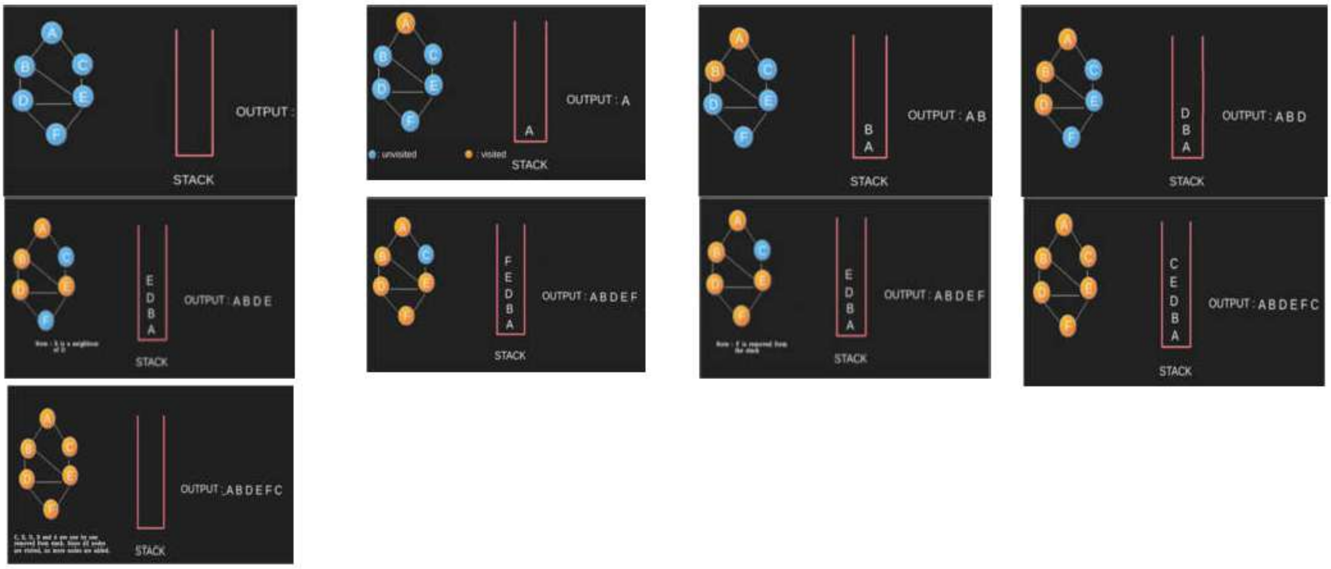
a->b = 1
a->e = 1
a->c = 2
a->f = 2
a->d = 3
a->g = 3
a->h = 4

การเก็บข้อมูล BFS ลงใน Array

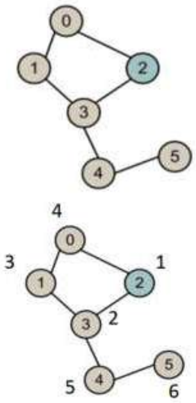
Index	0	1	2	3	4	5	6	7
Node	a	b	c	d	e	f	g	h
parent	-1	0	1	2	0	1	2	3

2) Depth first search (DFS)

หลักการคิด Print + Visit แล้วเริ่มท่องจากจุดนั้น ถ้าหากเจอตัวที่ไม่เคย visit เริ่มท่องจากจุดใหม่ (recursive) การท่องจะเจาะลงไปเรื่อยๆ เมื่อทำไม่ได้จะย้อนกลับมา เอาตัวที่ก่อนหน้า ถ้าเกิดยังไม่ซ้ำ ถ้าซ้ำก็ข้ามไป ดังนั้น OUTPUT ขึ้นอยู่กับ structure ถ้าในกรณีของ Link-list แต่ไม่มีผลในกรณี Array ข้อดีคือ Save memory, easy implementation

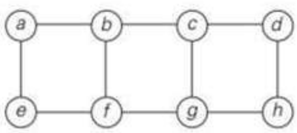


ตัวอย่าง DFS



DFS

stack		
2	Push 2 เชื่อมกับ 3 และ 0 เอาอันเดี่ยว คือ 3	
3 2	Push 3 เชื่อม 1 กับ 4 เอา 1 ตัวเดียว	
1 3 2	Push 1	
0 1 3 2	Push 0	
1 3 2	Pop 0 ไม่มีเส้นเชื่อมแล้ว ตัวที่อยู่ stack ไม่คิดอีก	0
3 2	Pop 1 ไม่มีเส้นเชื่อมแล้ว ตัวที่อยู่ stack ไม่คิดอีก	1
4 3 2	Push 4 จะ pop 3 ต้นเชื่อมกับ 4	
5 4 3 2	Push 5 จะ pop 4 ต้นเชื่อมกับ 5	
4 3 2	Pop 5	5
3 2	Pop 4	4
2	Pop 3	3
-	Pop 2	2



(a)

stack		
f	เริ่มจาก f	
e	e, b, g, a	
b	b g a c	
g	a c h	
a	c h	
c	h d	
h	d	
d	-	

<table>
<tr><td>
<pre>
bool visited_dfs[1000];
void reset_DFS()
{
   for(int i = 0 ; i < 1000 ; i++){visited_dfs[i] = false;}
}

void DFS (int start)
{
   cout<<start<<" ";
   visited_dfs[start] = true;
   node *r = edges[start];
   while(r->next != NULL)
   {
      if (!visited_dfs[r->next->item])
      {
         DFS (r->next->item);
      }
      r = r->next;
   }
}
</pre>
</td><td>
Reset ทั้งหมด คือไม่มี อะไร visit

เมืองเริ่มต้น เหมือน BFS โดยเริ่มจาก 0 ก็ปริ้ง 0 เลย
และไป start ที่เมือง 0
ไม่ได้ visit ก็ส่งเข้าไปแสดงต่อในกรณีนี้คือ 0 ไป 1 รอบต่อมาก็เป็น 2 แต่เนื่องจาก 2 โดย access ไปแล้วจาก recursive 2 ก็จะไม่ได้รับการปริ้งส่วนการ recursive คือ 0 ไป 1 และ 1 ไป 2 เมื่อถึง 2 แล้ว 2 ไป 0 และ 3 โดน 0 ไม่คิด คิดเฉพาะ 3 ถึง 3 ไปต่อไม่ได้ก็ย้อนกลับ
โดยรูปแบบของต้นไม้ดังนี้
0 : 1 2
1 : 2
2 : 0 3
3 : 3

0 1 2 3
</td></tr>
<tr><td>
<pre>
linkgraph *g = new linkgraph();
g->initial_graph(4);
g->insert_graph(0, 1);
g->insert_graph(0, 2);
g->insert_graph(1, 2);
g->insert_graph(2, 0);
g->insert_graph(2, 3);
g->insert_graph(3, 3);
g->print_graph();
g->reset_DFS();
g->DFS(2);
</pre>
</td><td>
เริ่มต้นที่ 2 ผลลัพธ์ 2 0 3 1
</td></tr>
</table>

8.3) การตรวจสอบ Connectivity ของกราฟ

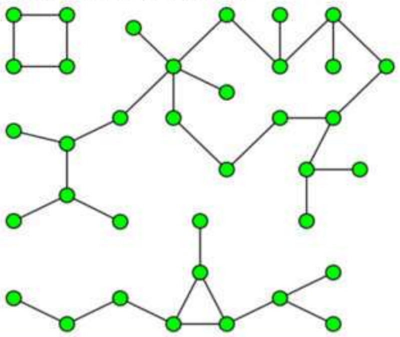
การตรวจสอบ Connectivity ของกราฟ หรือ การค้นหา Connected components (Connected sub graphs) หมายถึง เซตย่อยของ context ในกราฟซึ่งเชื่อมต่อซึ่งกันและกัน บางครั้งอาจเรียกว่าเป็นกราฟย่อย (sub graphs)

กราฟโดยทั่วไปจะมีเพียง 1 component ซึ่งหมายความว่าเราสามารถเข้าถึงทุกๆ vertex ในกราฟได้โดยผ่านการหาพาสในกราฟ

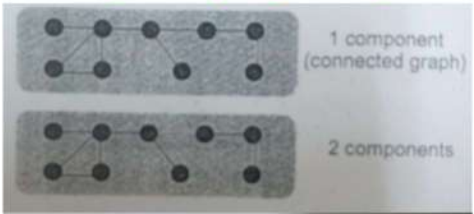
BFS สามารถตรวจสอบ connectivity โดยใช้ BFS เมื่อสิ้นสุดการทำงานตรวจสอบว่าทุกๆ vertex ใน graph ถูก Access ทุก vertex ถึงว่า connected

DFS สามารถหาจำนวน component ในกราฟ ให้ใช้ DFS ค้นหากราฟที่ 1 ได้มา 1 component ถ้ามี node เหลือตัวไหนที่ยังไม่ค้นหาอีก ให้ใช้ DFS ค้นหาอีกตัว node แล้วจะได้อีก component หาไปเรื่อยจนครบทุก node

การหากราฟย่อยว่า แตกออกไปกี่ส่วน

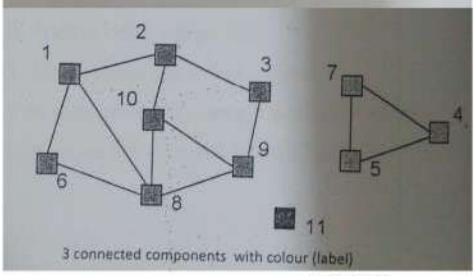


3 connected components = เรียกว่าเป็นกราฟย่อย

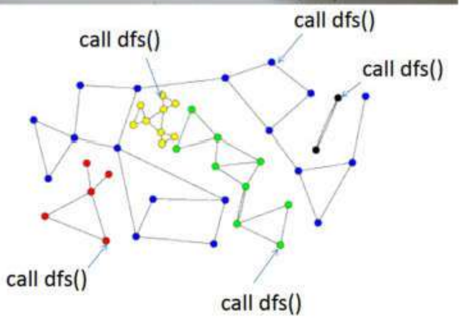


มีกราฟเพียงอันเดียว

มีกราฟ 2 อัน



มีกราฟ 3 อัน



5 connected components

void reset_DFS()……………………… void DFS (int start)……………	

void check_connect()
{
int label = 1;
reset_DFS();
for(int i = 0 ; i < n_vertices ; i++)
{
if(!visited_dfs[i])
{
cout<<"\nL = "<<label<<" : ";
DFS(i);
label = label + 1;
}
}
}
	จำนวน vertices ทั้งหมด = จำนวน node ทั้งหมด
	
	โดย DFS คือท่องให้หมด
	อะไรที่เชื่อมกันบ้าง
	L = 1 : 0 1 2
	L = 2 : 3 4 5
	L = 3 : 6 7 8

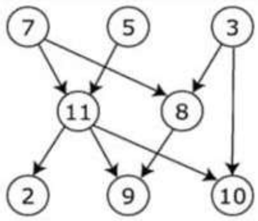
linkgraph *g = new linkgraph();
g->initial_graph(9);
g->insert_graph(0, 1);
g->insert_graph(0, 2);
g->insert_graph(1, 0);
g->insert_graph(1, 2);
g->insert_graph(2, 0);
g->insert_graph(2, 1);
g->insert_graph(3, 4);
g->insert_graph(3, 5);
g->insert_graph(4, 3);
g->insert_graph(4, 5);
g->insert_graph(5, 3);
g->insert_graph(5, 4);
g->insert_graph(6, 7);
g->insert_graph(6, 8);
g->insert_graph(7, 6);
g->insert_graph(7, 8);
g->insert_graph(8, 6);
g->insert_graph(8, 7);
g->check_connect();	

8.4) Topological Sorting

คือ ปัญหาของการเรียงลำดับ Node ของกราฟให้สอดคล้องกับทิศทางของ edge ซึ่งใช้ในกรณีจัดลำดับงาน โดยลำดับงานที่ 1 ต้องขึ้นอยู่กับลำดับงานที่ 2 ซึ่งกราฟต้องเป็น DAG (directed acyclic graph) ปัญหาของ Topological Sorting คือกรณีผังงานขนาดใหญ่ อาจจะเกิดความสับสน จำเป็นต้องมี อัลกอริทึมจัดการ โดยส่วนมาก algorithm สำหรับการหา topological sorting มี 2 ประเภท

1) การประยุกต์ DFS เพื่อค้นหาลำดับของ vectex

2) algorithm source delete



ขั้นตอนงาน 9 ทำได้ก็ต่อเมื่อขั้นตอน 8 และ 11 ทำเสร็จแล้ว ในทำนองเดียวกัน ขั้นตอน 11 จะทำได้ก็ต้อเมื่อ ขั้นตอน 7 และ 5 ทำเสร็จ

1) การประยุกต์ DFS เพื่อค้นหาลำดับของ vectex (Topological Sorting with DFS)

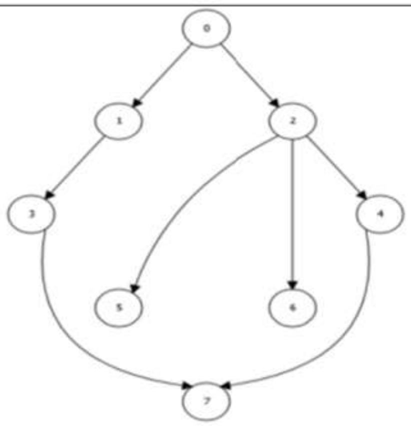
1 ท่องกราฟ DAG โดยใช้ algorithm DFS ปกติทั่วไป โดยเริ่มต้นที่ vectex ซึ่ง in-degree เป็น 0

2 สำรวจ vectex ต่างๆ ไปตามทิศทางของ edge

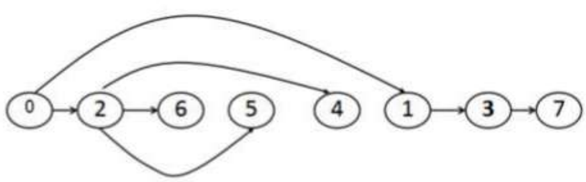
3 เมื่อใดก็ตามที่ DFS ไม่สามารถเข้าถึง vectex ถัดไปได้ ก็จะเกิดการ backtrack ขึ้น เราจะต้องบันทึก vectex ที่เกิดการ backtrack ลงใน temporary stack

4 เมื่อสำรวจกราฟจนครบ ลำดับของ vectex ใน temporary stack ก็จะถูกนำมาเขียนใหม่ โดยให้ vectex ที่อยู่ด้านบนของ stack ดังกล่าวอยู่ด้านซ้ายไล่ไปจนกระทั่งครบทุก vectex

ตัวอย่าง



ตามลำดับการ POP คือ 7 3 1 4 5 6 2 0 ทำ reverse จะได้ลำดับของ Topological order ดังนี้ คือ 0 2 6 5 4 1 3 7



0 push เริ่มต้น
1 push
3 push
7 push
7 pop
3 pop
1 pop
2 push
4 push
4 pop
5 push
5 pop
6 push
6 pop
2 pop
0 pop

2) Algorithm source delete

2.1) คำนวณ in-degree ของแต่ละ vertex ในกราฟทิศทาง

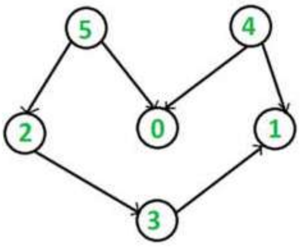
2.2) เริ่มต้นโดยการนำ vertex ซึ่งมี in-degree = 0 ไปใส่ในคิว

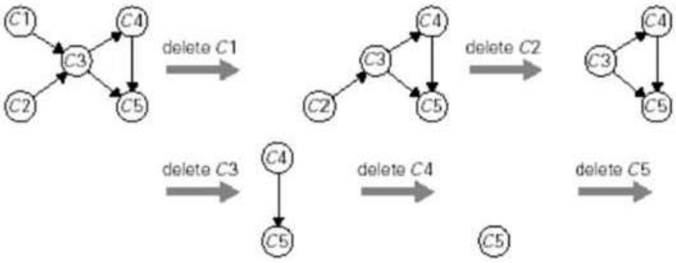
2.3) วนซ้ำหากพบว่ายังมี vertex ในคิว

2.3.1) เรียก dequeue และ output a vertex

2.3.2) ลด in-degree ของทุก vertex ที่มี edge ร่วมกับมันลง 1

2.3.3) นำ vertex ซึ่งมี in-degree เท่ากับ 0 ไปใส่ในคิว





The solution obtained is C1, C2, C3, C4, C5

5, 4, 2, 3, 1, 0 โดยการเรียงลำดับงาน โดยงานต้องเริ่มจาก 5 และ 4 ก่อน ต้องมาต้องทำ 2 และ 3 และ 1 ตามลำดับ สุดท้ายที่ 0

เริ่มที่ ตัวที่น้อยที่สุดคือ 0 และ 0 = visit → degree ออกไม่มี
ดูว่ามีตัวเชื่อมไหม ไม่มี เอาใส่ push → Stack
ถ้ามีให้แตกต่อ แตกแบบ DFS จนถึงตัวสุดท้าย
และเมื่อเจอตัวเลขไหน ให้ set ว่า visit แล้ว
กรณี 0 ไม่มี push → Stack

เริ่มที่ 1 และ 1 = visit → degree ออกไม่มี
กรณี 1 ไม่มี push → Stack

เริ่มที่ 2
กรณี 2 →2 = visit
มีตัวเชื่อม ไปที่ 3

เริ่มที่ 3 → 3 = visit มีตัวเชื่อม
ไปที่ 1 (แต่ visit แล้ว) ไม่มีตัวเชื่อมตัวอื่นแล้ว
ให้เอา 3 ใส่ใน Stack
แล้วกลับไปที่ 2 ให้เอา 2 ใส่ใน Stack

เริ่มที่ 4 → 4 = visit มีตัวเชื่อม
ไปที่ 0 (แต่ visit แล้ว)
ไปที่ 1 (แต่ visit แล้ว)
ให้เอา 4 ใส่ใน Stack

เริ่มที่ 5 → 5 = visit มีตัวเชื่อม
ไปที่ 0 (แต่ visit แล้ว)
ไปที่ 2 (แต่ visit แล้ว)
ให้เอา 5 ใส่ใน Stack

ใน Stack → บน 5, 4, 2 ,3, 1, 0 ล่าง
ให้ pop ออกมาให้หมด โดยเริ่มจากตัวบนสุด
จากรูปให้ทำงาน 5 และ 4 ก่อน ทำงาน 2 และ 3 และ 1 สุดท้ายคืองาน 0

bool tm_visited[1000]; int mypoint; int st[1000];	สำหรับทำหน้าที่เหมือน stack โดยใส่เข้าไปก่อนแล้วปริ้งย้อนกลับ

void topologicalSortUtil(int v)
{
tm_visited[v] = true;
node *r = edges[v];
while(r->next != NULL)
{
if (!tm_visited[r->next->item])
{
topologicalSortUtil (r->next->item);
}
r = r->next;
}
<u>st[mypoint] = v;</u>
<u>mypoint++;</u>
}
	ค้นหาแบบ DFS
	แต่เพิ่มส่วนนี้ ที่ขีดเส้นใต้ เข้ามา คือจะเก็บตัวที่ไหลไปจนสุดก่อนเช่น
	เริ่มจาก out degree เป็น 0 และ 1 ใส่เข้าไปเลย กลายเป็นตัวสุดท้ายเพราะไม่ได้เชื่อมกับใคร เลยไปหาใครไม่ได้
	แต่ถ้าเริ่มจาก 2 แล้ว 2 ไป 3 และไป 1 จากการค้นหาแบบ DFS ตามลำดับ
	มันจะกลายเป็นว่า 1 ถูกใส่ก่อน ตามด้วย 3 ถูกใส่ และ 2 ถูกใส่สุดท้ายใน Array จะมีหน้าตาแบบนี้
	[0] [1] [3] [2] ไม่ใส่ 1 ซ้ำ เพราะใส่ 1 ไปแล้วตามหลังของ DFS

void topologicalSort()
{
for (int i = 0 ; i < 1000 ; i++){ tm_visited[i] = false; }
mypoint = 0;
for (int i=0 ; i < n_vertices ; i++)
{
if (tm_visited[i] == false)
{
topologicalSortUtil(i);
}
}
for(int i=(mypoint-1) ; i >=0 ; i--){ cout<< st[i] << " ";}
}
	<u>เริ่มจากการ sort โดย out degree เป็น 0 ถึง n ต้อง Sort ก่อน</u>
	0 :
	1 :
	2 : 3
	3 : 1
	4 : 0 1
	5 : 2 0
	และทำการ reset visit และ mypoint
	ทำการค้นหาแบบ DFS ทุกอันโดยถูก visit ไปแล้วจะไม่สนใจอีก
	Print ย้อนหลัง เอาตัวสุดท้ายมา print
	เริ่มจาก 5 4 2, 3, 1 0 งานสุดท้ายคือ 0 กับ 1 ตามลำดับ

linkgraph *g = new linkgraph();
g->initial_graph(6);
g->insert_graph(5, 2);
g->insert_graph(5, 0);
g->insert_graph(4, 0);
g->insert_graph(4, 1);
g->insert_graph(2, 3);
g->insert_graph(3, 1);
g->topologicalSort();	

แบบฝึกหัด

1) จงเขียนโปรแกรมกราฟในรูปแบบ Adjacency Matrix ทั้งกราฟที่มีทิศทางและมีน้ำหนัก
2) จงเขียนโปรแกรมกราฟในรูปแบบ Adjacency List ทั้งกราฟที่มีทิศทางและมีน้ำหนัก
3) จงเขียนโปรแกรมกราฟ Breadth first search (BFS)
4) จงเขียนโปรแกรมกราฟ Depth first search (DFS)
5) จงเขียนโปรแกรมกราฟตรวจสอบ Connectivity ของกราฟ
6) จงเขียนโปรแกรมกราฟ Topological Sorting