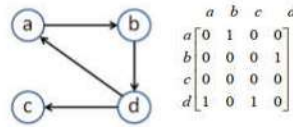


บทที่ 10 กราฟ (ต่อ)

10.1 การค้นหากราฟ Path ในกราฟ



มี path จาก a \rightarrow c หรือไม่ \rightarrow กราฟทิศทาง Directed graph

มี path จาก b \rightarrow d หรือไม่ \rightarrow กราฟทิศทาง Directed graph

Transitive Relation

สมบัติถ่ายทอด (Transitive) เมื่อมีคู่ลำดับที่มีความสัมพันธ์แบบเชื่อมกันเกิดขึ้น เช่น (1,2) (2,3) ต้องมีตัวที่ 3 ตามมา ถ้าไม่มีไม่เป็นไร แต่ถ้าต้องมีตัวที่สามตามมาเท่านั้น นั่นคือ (1,3)

$r_1 = \{ (1,1), (2,4) \}$

มีสมบัติถ่ายทอด

$r_2 = \{ (1,2), (2,4), (1,4), (1,3) \}$

มีสมบัติถ่ายทอด

$r_3 = \{ (1,2), (2,4), (1,3) \}$

ไม่มีสมบัติถ่ายทอด ไม่มี (1,4)

$r_4 = \{ (1,1), (2,2), (3,3), (1,2), (2,1) \}$

มีสมบัติถ่ายทอด

$r_5 = \{ (2,2), (3,3), (1,2), (2,1) \}$

ไม่มีสมบัติถ่ายทอด ไม่มี (1,1)

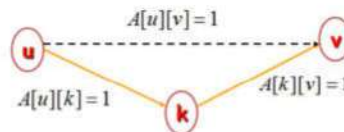
$r_6 = \{ (1,1), (3,3), (1,2), (1,3) \}$

มีสมบัติถ่ายทอด

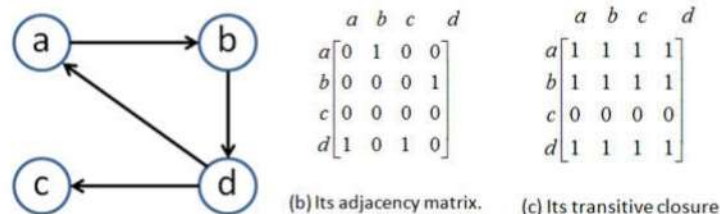
$r_7 = \{ (1,1), (2,2), (1,2), (2,3) \}$

ไม่มีสมบัติถ่ายทอด ไม่มี (1,3)

กำหนดให้ u และ v พาสจาก u ไป v ถ้ามี **direct edge** จาก u ไป v หรือ มี **intermediate node** k ซึ่งมี พาสจาก u ไป k และ พาส k ไป v หาก $R[u][v]=1$ เรียกความสัมพันธ์ที่เกิดขึ้นนี้ว่า Transitive relation คือ ไม่ได้ไปโดนตรง แต่ไปถึงได้โดยอาศัย node อื่นๆ เชื่อมเข้าหา



Transitive Closure Matrix



เมตริกซ์ Transitive Closure จะแสดงความสัมพันธ์ r_i ซึ่งเกิดขึ้นใน Digraph โดยแสดงว่า a สามารถไปหา b, c, d ได้หรือไม่ โดยถ้าเป็น 1 แสดงว่าไปได้ และถ้าเป็น 0 แสดงว่าไปไม่ได้ และ a ไปหา a คือวนกลับมาหาตัวเองได้ไหม

Transitive Closure using DFS

	0 1 1 0	0 0 0 0	เริ่มต้น	1 1 1 1
	0 0 1 0	0 0 0 0	หลังจากนั้นให้ทำ DFS ทุก node เริ่มต้นที่ 0	1 1 1 1
	1 0 0 1	0 0 0 0	ท่อง DFS จะได้มาเป็น 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 ก็ได้	1 1 1 1
	0 0 0 0	0 0 0 0	1 เข้าไปแถวแรกให้หมด	0 0 0 0
	โครงสร้างของ Graph	0 0 0 0	แถว 2 ได้ผลลัพธ์ คือ 2 \rightarrow 0 \rightarrow 1 \rightarrow 3 ใส่ 1 เข้าไปแถวสอง	ผลลัพธ์จากการทำ DFS

หลักการคือ ใช้ DFS กับทุก node โดยถ้า node ไหนเป็น node เริ่มต้น ให้ lock แถวของ node นั้นแล้วถ้าสามารถ access ได้ ให้ add เข้าไปใน column ของ node นั้น โดยต้องระวังกรณีเริ่มต้น คือ node ตัวเอง ต้องไม่ให้ access node ตัวเองจนกว่าจะ travel กลับมา node ตัวเองได้เองจึง add เข้าไป ข้อเสียของวิธีนี้คือ มีการคำนวณที่ซ้ำซ้อน เนื่องจากต้องท่อง Digraph หลายครั้ง

CODE

<pre> bool tc[1000][1000]; bool visited_dfs[1000]; int index_i; int first_time; </pre>	<p>คำตอบถ้าถูก access เป็น 1 แต่ default เป็น 0 ต้อง reset ทุกครั้ง เก็บแถว</p>	<p>0 : 1 2 1 : 2 2 : 0 3 3 :</p>
<pre> void DFSUtil(int start) { if(first_time > 0) { visited_dfs[start] = true; tc[index_i][start] = 1; } first_time++; for (int j=0 ; j < n_vertices ; j++) { if (edges[start][j] > 0 && visited_dfs[j] == false) { DFSUtil(j); } } } </pre>	<p>กรณีในการ run ครั้งแรกต้องไม่ visit node แรกแต่หลังจากท่องแล้วสามารถกลับมา visit node ตัวเองได้ เพราะถ้า visit node แรกเลยกรณีแถวสุดท้ายจะมีเลข 1 อยู่ที่ตัวแรกเสมอ ดังนั้น node แรกจะไม่ visit แต่ถ้าสามารถวนกลับมาได้ก็จะนับ node แรกไปด้วยเลย</p> <p>ทำ DFS ปกติ</p>	<p>----- 0 1 1 0 0 0 1 0 1 0 0 1 0 0 0 0 ----- 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0</p>
<pre> void transitiveClosure() { for (int i = 0; i < n_vertices ; i++) { for (int j=0; j<n_vertices ; j++){ cout << edges[i][j] << " "; cout << endl; } } for (int i = 0; i < n_vertices ; i++) { for (int j=0; j<n_vertices ; j++){ tc[i][j] = false; } } for (int i = 0; i < n_vertices ; i++) { for (int j=0; j<n_vertices ; j++) { visited_dfs[j] = false; } first_time = 0; index_i = i; DFSUtil(i); } for (int i=0; i < n_vertices; i++) { for (int j=0; j<n_vertices ; j++){ cout << tc[i][j] << " "; cout << endl; } } } </pre>	<p>ปรีงตาราง matrix ธรรมดา</p> <p>ดำเนินการ reset ตารางคำตอบเป็น 0 ให้หมด</p> <p>clear ตาราง visit เพื่อทำ DFS ใหม่หมด เริ่มต้น first_time = 0 เพื่อให้เริ่มต้นไม่ access ตัวเอง และ index คือ ตำแหน่งเริ่มต้น</p> <p>ปรีงตาราง matrix ธรรมดา</p>	
<pre> graph *g = new graph(); 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->print_graph(); g->transitiveClosure(); </pre>		

10.2 WARSHALL algorithm

WARSHALL algorithm เป็น iterative search ซึ่งสร้าง transitive closure matrix ขนาด $n \times n$ จาก matrix ที่มีการเชื่อมต่อกันของ digraph

$R^{(0)}$ คือ เมตริกซ์ของกราฟ

$R^{(1)}$ คือ เมตริกซ์ของ path ที่มี 1^{st} vertex เป็น intermediate vertex

$R^{(2)}$ คือ เมตริกซ์ของ path ที่มี 1^{st} และ 2^{nd} vertices เป็น intermediate vertices

.....

$R^{(n)}$ คือ เมตริกซ์ของ path ที่มี n vertices อยู่ใน intermediate vertices คือ transitive closure ของ digraph

กำหนดให้ Intermediate vertex เป็น vertex 1 จากนั้นทำการคำนวณ $R(1)$ จาก $R(0)$

$$R^{(1)}[i][j] = R^{(0)}[i][j] \vee (R^{(0)}[i][1] \wedge R^{(0)}[1][j])$$

ตัวอย่างที่ 1



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

จากรูปนี้ คือตารางต้นฉบับ

Step 1: เมื่อทำ $R1$ สร้างตาราง $(R0 \wedge R0$ แบบพิเศษ) $\cup R0$ ต้นฉบับ = $R1$

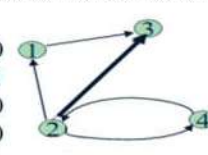
0	0	1	0
1			
0			
0			

หลักตรงไหน เป็นเลข 1 สีฟ้า ให้เอาแถวไปแปะไว้ตรงนั้น

0	0	1	0

ดูทีละแถว ถ้าแถวไหนเป็น 1 ให้เอาแถวนั้น ไปแปะเข้าไป ส่วนแถวที่เป็น 0 ใส่ 0 ทั้งหมด

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



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

ผลลัพธ์ครั้งที่ 1

Step 2: ต่อแบบเดิม

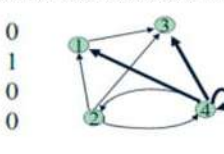
	0		
1	0	1	1
	0		
	1		

หลักตรงไหน เป็นเลข 1 สีฟ้า ให้เอาแถวไปแปะไว้ตรงนั้น เอาผลลัพธ์ที่ 1 มาคิดไม่ได้เอาจากโจทย์

1	0	1	1

ดูทีละแถว ถ้าแถวไหนเป็น 1 ให้เอาแถวนั้น ไปแปะเข้าไป ส่วนแถวที่เป็น 0 ใส่ 0 ทั้งหมด

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



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

ผลลัพธ์ครั้งที่ 2

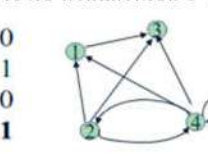
Step 3: ต่อแบบเดิม

		1	
		1	
0	0	0	0
		1	

ดูทีละแถว ถ้าแถวไหนเป็น 1 ให้เอาแถวนั้น ไปแปะเข้าไป ส่วนแถวที่เป็น 0 ใส่ 0 ทั้งหมด

0	0	0	0
0	0	0	0
0	0	0	0

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



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

ผลลัพธ์ครั้งที่ 3

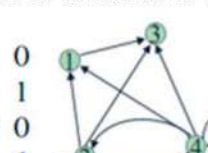
Step 4: ต่อแบบเดิม

			0
			1
			0
1	1	1	1

ดูทีละแถว ถ้าแถวไหนเป็น 1 ให้เอาแถวนั้น ไปแปะเข้าไป ส่วนแถวที่เป็น 0 ใส่ 0 ทั้งหมด

1	1	1	1
1	1	1	1

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

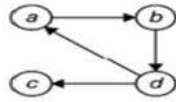


0	0	1	0
1	1	1	1
0	0	0	0
1	1	1	1

CODE

<pre> int reach[1000][1000]; void Warshall() { for (int i = 0; i < n_vertices; i++) { for (int j = 0; j < n_vertices; j++) { reach[i][j] = edges[i][j]; } } for (int k = 0; k < n_vertices; k++) { for (int ii=0 ; ii < n_vertices; ii++) { for (int jj=0 ; jj < n_vertices; jj++) { cout<<(reach[k][jj] && reach[ii][k])<<" "; } cout<<endl; } } for (int i = 0; i < n_vertices; i++) // X { for (int j = 0; j < n_vertices; j++) //Y { reach[i][j] = reach[i][j] (reach[i][k] && reach[k][j]); } } for (int ii=0 ; ii < n_vertices; ii++) { for (int jj=0 ; jj < n_vertices; jj++) { cout<<reach[ii][jj]<<" "; } cout<<endl; } } </pre>	<p>สร้างตาราง</p> <p>ดำเนินการ copy ตารางทั้งหมด</p> <p>R1 To R4 ได้ตามแถว</p> <p>ปริงข้อความเฉยๆ หลังจากการปรับปรุงตาราง</p> <p>วน Loop แก่ X กับ Y</p> <p>ส่วน code จริงๆ คือตรงนี้ โดยเปรียบเทียบแก่ X กับ Y ว่า match กันไหม โดย && คือ 1 กับ 1 เป็น 1 และ เปรียบเทียบ คือ 1 กับ 1 เป็น หรือ</p> <p>ปริงข้อความเฉยๆ หลังจากการปรับปรุงตาราง</p>	<pre> 0:12 1:2 2:03 3: ----- 0000 0000 0110 0000 ----- 0010 0000 0010 0000 ----- 1111 1111 1111 0000 ----- 0000 0000 0000 0000 ----- 1111 1111 1111 0000 </pre>
<pre> graph *g = new graph(); 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->print_graph(); g->Warshall(); </pre>		

ตัวอย่างที่ 2



	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

จากรูปนี้ คือตารางต้นฉบับ

STEP 1

<table><tr><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td></td><td></td><td></td></tr><tr><td>0</td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td></tr></table>	0	1	0	0	0				0				1				<table><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>													0	1	0	0	+	<table><tr><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	<table><tr><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	0	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0
0	1	0	0																																																																	
0																																																																				
0																																																																				
1																																																																				
0	1	0	0																																																																	
0	1	0	0																																																																	
0	0	0	1																																																																	
0	0	0	0																																																																	
1	0	1	0																																																																	
0	1	0	0																																																																	
0	0	0	1																																																																	
0	0	0	0																																																																	
1	1	1	0																																																																	

STEP 2

	1		
0	0	0	1
	0		
	1		

0	0	0	1
0	0	0	1

+

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

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

STEP 3

		0	
		0	
0	0	0	0
		1	

0	0	0	0

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

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

STEP 4

			1
			1
			0
1	1	1	1

1	1	1	1
1	1	1	1
1	1	1	1

+

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

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

คำตอบ

คำตอบ

10.3 การค้นหาเส้นทางที่สั้นที่สุดในกราฟแบบมีน้ำหนัก

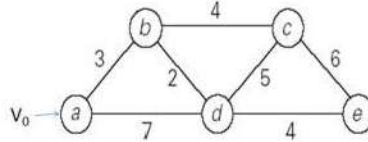
ปัญหาการหาเส้นทางที่สั้นที่สุด (Shortest path problem) ของการเดินทางจากจุดหนึ่งไปยังอีกจุดหนึ่ง โดยปัญหาการเดินทางของพนักงานขาย, ปัญหาการขนส่ง, ปัญหาต้นไม้แบบทอดข้ามน้อยสุด ถึงเป็นปัญหาประเภทนี้ สำหรับอัลกอริทึมที่นิยมใช้แก้ปัญหานี้ คือ Prim, Kruskal, Dijkstra

ปัญหาการหาเส้นทางที่สั้นที่สุดโดยทั่วไปเราสามารถแบ่งปัญหานี้ออกเป็น 2 ประเภท คือ

1. การหาเส้นทางที่สั้นที่สุดโดยกำหนดจุดเริ่มต้น เรียกปัญหานี้ว่า Single Source Shortest Path
2. การหาเส้นทางที่สั้นที่สุดที่จุดใดก็ได้ เรียกปัญหานี้ว่า All-Pair Shortest Path

1) Single Source Shortest Path

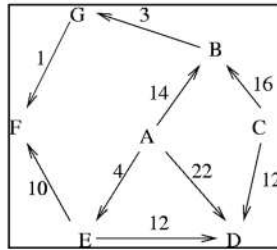
กำหนดให้ G เป็นกราฟแบบมีน้ำหนัก และ V_0 เป็น vertex เริ่มต้น (source vertex) ค้นหาทุกๆ shortest path จาก V_0 ไปยัง vertex ที่เหลือใน G



1.1) Dijkstra Algorithm

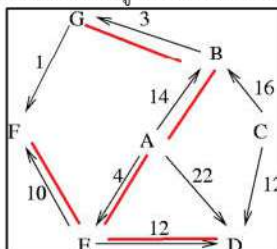
ใช้แก้ปัญหาการเดินทางจากจุดหนึ่งไปหาจุดอื่นๆ ทั้งหมดในระยะทางที่สั้นที่สุด (single-source shortest path) หลักการทำงานเป็นแบบกริดี้ โดยหลักการคือ ให้เลือกจุดที่เราต้องการเดินทางไปหาจุดอื่นๆ มาหนึ่งจุด (A) แล้วคำนวณผลรวมระยะจากจุดนั้น ถึงจุดอื่นๆ (B,C,D,E,F,G) โดยพิจารณาทุกเส้นทางที่จุดหนึ่ง (A) สามารถถึงอีกจุดหนึ่ง (B หรือ C หรือ D หรือ E หรือ F หรือ G) วิธีนี้ถูกเอามาใช้ในการทำ Routing Protocols (IS-IS, OSPF)

หมายเหตุ ค้นหาระยะทางต่ำสุดจากโหนด i ถึงโหนดทั้งหมด จะได้ต้นไม้ระยะทางต่ำสุดจากโหนด i



เริ่มต้นที่จุด A เดินทางไปหาจุดอื่นๆ (B,C,D,E,F,G)		
ที่ A มีช่องทางเชื่อมต่อ B, D, E	$A \rightarrow B = 14$ $A \rightarrow D = 22$ $A \rightarrow E = 4$	เลือก E
ที่ E มีช่องทางเชื่อมต่อ F, D	$A \rightarrow B = 14$ $A \rightarrow E \rightarrow F = 14$ $A \rightarrow D = 22$ สามารถไปหา D มีเส้นทางที่สั้นกว่าเลยตัดออก $A \rightarrow E \rightarrow D = 16$	เลือก B
ที่ B มีช่องทางเชื่อมต่อ G	$A \rightarrow E \rightarrow F = 14$ $A \rightarrow E \rightarrow D = 16$ $A \rightarrow B \rightarrow G = 17$	เลือก F
ที่ F ไม่มีตัวเชื่อมต่อ	$A \rightarrow E \rightarrow D = 16$ $A \rightarrow B \rightarrow G = 17$	เลือก D
ที่ D ไม่มีตัวเชื่อมต่อ	$A \rightarrow B \rightarrow G = 17$	เลือก G

เมื่อเลือกแล้วทำให้การเชื่อมต่อเส้นทางเข้ากับทุกโหนดเสร็จสมบูรณ์ เพราะจาก รูป C ไม่สามารถเดินทางไปได้ โดยผลลัพธ์จะเป็นดังรูปข้างล่าง



CODE Dijkstra

<pre>void Dijkstra(int point) {</pre>	
<pre> int ListNode[100][100]; double DistNode[100]; int ContNode[100]; bool Compnode[100]; int NumCity = n_vertices; int DistN[100][100]; int fix_point = point; for(int i=0 ; i < NumCity ; i++) { for(int j=0 ; j < NumCity ; j++) { DistN[i][j] = edges[i][j]; if(i == j DistN[i][j] == 0){ DistN[i][j] = 99999; } } } for(int i=0;i<NumCity;i++){ ListNode[i] = 99999; } for(int i=0;i<NumCity;i++){ ListNode[i][0] = point; } for(int i=0;i<NumCity;i++){ ContNode[i] = 1; } for(int i=0;i<NumCity;i++){ Compnode[i] = false; }</pre>	<p>//initial value โดยการ reset ค่าต่างๆ</p> <p>fix_point คือตำแหน่งเริ่มต้นต้องไม่มีการวนกลับ เอาไว้ block การวนกลับ</p> <p>สร้างตาราง Distance Matrix</p> <p>ListNode คือ คำตอบ เช่น 1 ไป 2 DistNode คือ ระยะทาง ของแต่ละคำตอบ เช่น 1 ไป 2 คือ 15 หน่วย ContNode คือ ความยาวของ ListNode Compnode คือ ตัว check สำหรับหยุดโปรแกรม โดยถ้าเป็น true หมดก็หยุดโปรแกรม</p>
<pre> for(int i=0 ; i < NumCity ; i++) { for(int j=0 ; j < NumCity ; j++) { cout<<DistN[i][j]<<" "; } cout<<endl; }</pre>	<p>แสดงผล ตาราง Distance Matrix</p> <pre>99999 4 99999 99999 99999 99999 99999 8 99999 4 99999 8 99999 99999 99999 99999 11 99999 99999 8 99999 7 99999 4 99999 99999 2 99999 99999 7 99999 9 14 99999 99999 99999 99999 99999 99999 9 99999 10 99999 99999 99999 99999 99999 4 14 10 99999 2 99999 99999 99999 99999 99999 99999 99999 2 99999 1 6 8 11 99999 99999 99999 99999 1 99999 7 99999 99999 2 99999 99999 99999 6 7 99999</pre>
<pre> double Max = 99999; int select = point; for(int i=0 ; i < NumCity ; i++) { if(99999 > DistN[point][i]) { DistNode[i] = DistN[point][i]; ListNode[i][ContNode[i]] = i; ContNode[i] = ContNode[i] + 1; } if(Compnode[i] == false && Max > DistNode[i]) { Max = DistNode[i]; select = i; } } Compnode[select] = true; point = select;</pre>	<p>//first node เลือก 0</p> <p>A,B,C,D,E,F,G</p> <pre>99999 4 99999 99999 99999 99999 99999 8 99999</pre> <p>A ไป B และ F ได้</p> <p>ดำเนินการแตกออกตามเส้นทางที่แตกไปได้ และนำผลลัพธ์ไปเก็บไว้ในคำตอบ</p> <p>A->B = 4 A->F = 8</p> <p>Max ในที่นี้ ใช้สำหรับ บังคับให้เลือกตัวน้อยสุดเป็นตัวต่อไปในกรณีนี้ เลือก B Select = B</p> <p>node แรกโดนเลือกไปแล้ว</p>
<pre> cout<<"["<<select<<"\n"; for(int i=0 ; i < NumCity ; i++) { for(int j=0 ; j < ContNode[i] ; j++) { cout<<"["<<ListNode[i][j]; } cout<<"] = "<<DistNode[i]<<"\n"; } cout<<"-----\n";</pre>	<p>แสดงผล เลือก 1 = B</p> <pre>[1] 0 = 99999 0 1 = 4 0 = 99999 0 = 99999 0 = 99999 0 = 99999 0 = 99999 0 7 = 8 0 = 99999</pre>

```

while(true)
{
    Max = 99999;
    for(int i=0 ; i < NumCity ; i++)
    {
        if( 99999 > DistN[point][i] && i != fix_point )
        {
            double TDistNode = DistNode[point] + DistN[point][i];
            if(DistNode[i] > TDistNode)
            {
                for(int j=0 ; j < ContNode[point] ; j++)
                {
                    ListNode[i][j] = ListNode[point][j];
                }
                DistNode[i] = TDistNode;
                ContNode[i] = ContNode[point];
                ListNode[i][ContNode[i]] = i;
                ContNode[i] = ContNode[i] + 1;
            }
        }
        if( Compnode[i] == false && Max > DistNode[i] )
        {
            Max = DistNode[i];
            select = i;
        }
    }
    Compnode[select] = true;
    point = select;
}

```

//second to final node

บังคับไม่ให้วนกลับ

การบวกกับตำแหน่งจากปัจจุบันไปตำแหน่งใหม่
 ถ้ากรณีที่ตำแหน่งเก่าไปชนกันและมีค่าต่ำกว่าตำแหน่งเก่าก็เอาไปแทนที่ทั้งเส้น
 สมมุติ
 A->B->C = 20 แตกใหม่
 A->C = 30 ของเดิม
 ก็ให้เอาของใหม่คือ A-B-C แทน A-C
 เดิม default จะเป็น 99999 หมด อย่างไรก็ตามก็น้อยกว่าถ้าเป็น path ใหม่
 ที่เหลือเหมือนกรณี โหนดแรก

```

cout<<"<<select<<"\n";
for(int i=0 ; i < NumCity ; i++)
{
    for(int j=0 ; j < ContNode[i] ; j++)
    {
        cout<<"*<<ListNode[i][j];
    }
    cout<<"| = "<<DistNode[i]<<"\n";
}
cout<<"-----\n";

```

[7]	[6]	[4]
0 = 99999	0 = 99999	0 = 99999
0 1 = 4	0 1 = 4	0 1 = 4
0 1 2 = 12	0 1 2 = 12	0 1 2 = 12
0 = 99999	0 = 99999	0 1 2 3 = 19
0 = 99999	0 = 99999	0 7 6 5 4 = 21
0 = 99999	0 = 99999	0 7 6 5 = 11
0 = 99999	0 7 6 = 9	0 7 6 = 9
0 7 = 8	0 7 = 8	0 7 = 8
0 = 99999	0 7 8 = 15	0 1 2 8 = 14
	

```

bool BreakProgram = true;
for(int i=0 ; i < NumCity ; i++)
{
    if( Compnode[i] == false && DistNode[i] < 99999 )
    {
        BreakProgram = false; break;
    }
}
if(BreakProgram){ break; }
}
}

```

ถ้าทุกเมืองถูกเลือกหมดแล้วก็ให้หยุดการค้นหา


```

graph *g = new graph();
g->initial_graph(9);
g->insert_graph(0, 1, 4);
g->insert_graph(0, 7, 8);
g->insert_graph(1, 0, 4);
g->insert_graph(1, 2, 8);
g->insert_graph(1, 7, 11);
g->insert_graph(2, 1, 8);
g->insert_graph(2, 3, 7);
g->insert_graph(2, 5, 4);
g->insert_graph(2, 8, 2);
g->insert_graph(3, 2, 7);
g->insert_graph(3, 4, 9);
g->insert_graph(3, 5, 14);
g->insert_graph(4, 3, 9);
g->insert_graph(4, 5, 10);
g->insert_graph(5, 2, 4);
g->insert_graph(5, 3, 14);
g->insert_graph(5, 4, 10);
g->insert_graph(5, 6, 2);
g->insert_graph(6, 5, 2);
g->insert_graph(6, 7, 1);
g->insert_graph(6, 8, 6);
g->insert_graph(7, 0, 8);
g->insert_graph(7, 1, 11);
g->insert_graph(7, 6, 1);
g->insert_graph(7, 8, 7);
g->insert_graph(8, 2, 2);
g->insert_graph(8, 6, 6);
g->insert_graph(8, 7, 7);
g->Dijkstra(0);

```

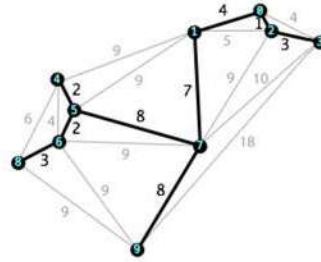
```

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

```

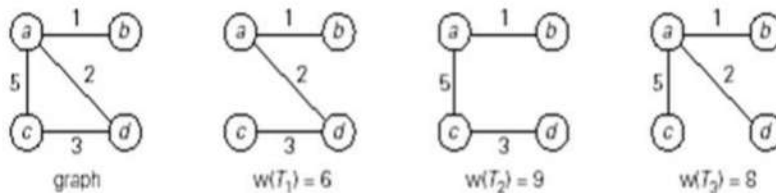
2) All-Pair Shortest Path

Acyclic Graph คือ กราฟที่ไม่มี Cycle เช่น Spanning tree (vertex ทุกอันต้องเชื่อมกันต่อในกราฟและมีจำนวนที่เชื่อมมน้อยที่สุด)



ปัญหาการค้นหา Minimum Spanning Tree (MST) คือ เส้นที่เชื่อมกันแล้วได้น้ำหนักน้อยสุด Spanning Tree คือ acyclic graph ซึ่งเชื่อมกับทุกๆ vertex สำหรับกราฟแบบมีน้ำหนัก

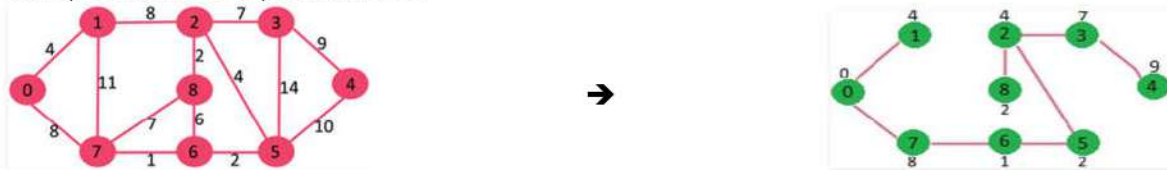
หมายเหตุ algorithm 2 คือ Prim Algorithm และ Kruskal Algorithm ต้นไม้ที่เชื่อมต่อโหนดทั้งหมดในขณะที่ผลรวมของค่าใช้จ่ายทั้งหมดเป็นค่าต่ำสุดที่เป็นไปได้



การใช้ brute force search or exhaustive search เพื่อสร้าง minimum spanning tree อาจเจอปัญหาสำคัญ 2 ปัญหา ได้แก่

- 1) จำนวน spanning trees เติบโตแบบ exponential เมื่อเทียบกับขนาดของกราฟ 2^n เพราะเป็น binary ลักษณะเป็นตารางแล้วเลือกไม่เลือก
- 2) การสร้าง spanning trees ทั้งหมดของ graph ไม่ใช่เรื่องง่าย

Example Minimum spanning tree



2.1) Prim Algorithm

Prim's algorithm (Prim's Minimum Spanning Tree, MST)

สิ่งที่ต้องเก็บคือ Index of array = node / Value of array = weight

สำหรับช่องที่ไม่มีการเชื่อมต่อกัน ให้กำหนดช่องนั้นเป็นค่ามากๆ ตั้งแต่สร้าง distance matrix

สำหรับช่องที่เชื่อมกับตัวเอง ให้กำหนดช่องนั้นเป็นค่ามากๆ ตั้งแต่สร้าง distance matrix

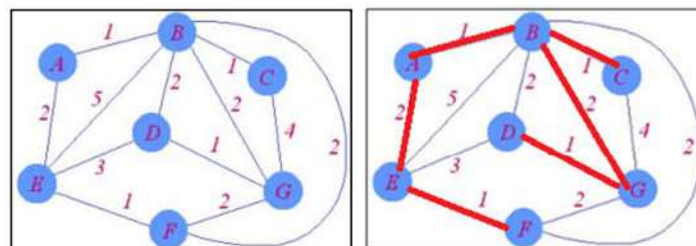
ถ้าเจออะไรมาพิจารณากับ Array ชุดนี้

- เปรียบตัวที่แตกมาใหม่แล้วเอาที่ค่าใน Array ตัวที่ visit จะไม่พิจารณาแล้ว
- แล้วเลือกกระยะที่น้อยสุด แล้วกำหนดเป็น visit โดยจะไม่เลือกตัว visit แล้ว เพื่อเอาไปแตกต่อ

สร้าง array เก็บ node ที่ access ป้องกันการ access ซ้ำ node เดิม และตรวจสอบการหยุดการทำงาน กับ ค่าของ node ที่ access ต้องค่าที่น้อยที่สุดแต่ละรอบหา ค่าที่น้อยที่สุด

ใช้แก้ปัญหาต้นไม้แบบทอดข้ามน้อยสุด และสามารถเอาไปใช้กับปัญหาการเดินทางของพนักงานขาย หลักการทำงานเป็นแบบกรีดี โดยแบ่งโหนดออกเป็น 2 ส่วน คือ ส่วนที่เชื่อมต่อแล้ว กับส่วนที่ยังไม่ได้เชื่อมต่อ โดยพิจารณาจากโหนดไล่เชื่อมไปเรื่อยๆ จนทุกโหนดเชื่อมต่อกัน

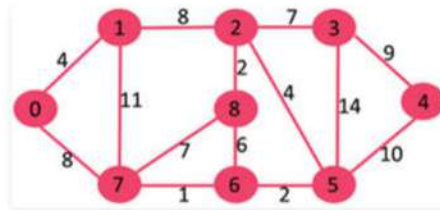
ตัวอย่างที่ 1



- จากรูปเริ่มที่ A (เริ่มต้นสุ่มโหนดใดโหนดหนึ่งที่อยู่ในรูป) แล้วพิจารณา ว่า A เชื่อมกับอะไรบ้าง แล้วเลือกตัวที่เชื่อมต่อน้อยสุด $A \rightarrow 1 \rightarrow B$ และ $A \rightarrow 2 \rightarrow E$ เลือกน้อยสุด {A,B} ดังนั้น A, B เป็นส่วนที่เชื่อมต่อกัน
- ในรอบต่อมาพิจารณาส่วนที่ไม่ได้เชื่อมต่อกัน ได้แก่ $A \rightarrow 2 \rightarrow E$, $B \rightarrow 1 \rightarrow C$, $B \rightarrow 2 \rightarrow G$, $B \rightarrow 2 \rightarrow D$, $B \rightarrow 5 \rightarrow E$, $B \rightarrow 2 \rightarrow F$ เลือกน้อยสุด {B,C} ดังนั้น A, B, C เป็นส่วนที่เชื่อมต่อกัน

- ในรอบต่อมาพิจารณาส่วนที่ไม่เชื่อมต่อกัน ได้แก่ $A \rightarrow 2 \rightarrow E$, $B \rightarrow 2 \rightarrow G$, $B \rightarrow 2 \rightarrow D$, $B \rightarrow 5 \rightarrow E$, $B \rightarrow 2 \rightarrow F$, $C \rightarrow 4 \rightarrow G$, เลือกน้อยสุด A,E หรือ B,G หรือ B,D จากกรณีนี้ {A,E} ดังนั้น A, B, C, E เป็นส่วนที่เชื่อมต่อกัน
- ในรอบต่อมาพิจารณาส่วนที่ไม่เชื่อมต่อกัน ได้แก่ $B \rightarrow 2 \rightarrow G$, $B \rightarrow 2 \rightarrow D$, $B \rightarrow 5 \rightarrow E$, $B \rightarrow 2 \rightarrow F$, $C \rightarrow 4 \rightarrow G$, $E \rightarrow 3 \rightarrow D$, $E \rightarrow 1 \rightarrow F$ เลือกน้อยสุด จากกรณีนี้ {E,F} ดังนั้น A, B, C, E, F เป็นส่วนที่เชื่อมต่อกัน
- ในรอบต่อมาพิจารณาส่วนที่ไม่เชื่อมต่อกัน ได้แก่ $B \rightarrow 2 \rightarrow G$, $B \rightarrow 2 \rightarrow D$, $B \rightarrow 5 \rightarrow E$, $B \rightarrow 2 \rightarrow F$, $C \rightarrow 4 \rightarrow G$, $E \rightarrow 3 \rightarrow D$, $F \rightarrow 2 \rightarrow G$ เลือกน้อยสุด จากกรณีนี้ {B,G} ดังนั้น A, B, C, E, G, F เป็นส่วนที่เชื่อมต่อกัน
- ในรอบต่อมาพิจารณาส่วนที่ไม่เชื่อมต่อกัน ได้แก่ $B \rightarrow 2 \rightarrow D$, $B \rightarrow 5 \rightarrow E$, $B \rightarrow 2 \rightarrow F$, $C \rightarrow 4 \rightarrow G$, $E \rightarrow 3 \rightarrow D$, $F \rightarrow 2 \rightarrow G$, $G \rightarrow 1 \rightarrow D$ เลือกน้อยสุด จากกรณีนี้ {G,D} ดังนั้น A, B, C, E, G, D, F เป็นส่วนที่เชื่อมต่อกัน แล้วก็เชื่อมต่อกันหมดแล้ว ดังรูปข้างบน

ตัวอย่างที่ 2



	<p>เริ่มจาก 0 แล้วใช้ greedy แตกมาเป็น 1 กับ 7 เอา 1 เพราะน้อยกว่า ไม่มีการบวกตั้งแต่ตำแหน่งแรก เอาน้อยไว้ก่อนอะไรน้อยเลือกเลย ถ้ามีช่องทางอื่น access ได้ เอาช่องทางที่น้อยที่สุดเก็บไว้อย่างเดียวหรือแทนที่ช่องทางเดิม</p>
	<p>เมื่อแตก 1 ต่อ เป็น 2 กับ 7 ไ้ 1 มีไปเชื่อม 7 = 11 ไม่เอา เพราะ ยาวกว่าไม่มีการคิดย้อนจุด คิดแค่ค่าของช่องปัจจุบัน และ 1 ไปเชื่อมกับ 2 ณ ตอนนี้อยู่ไม่มีใครเชื่อม 2 เอาไว้ก่อน แข่งกันระหว่าง 7 ของ 8 กับ 2 ของ 8 เลือกอะไรก็ได้ * dijkstra คิดจากจุด แต่ prim เปรียบเทียบกับปัจจุบันเท่านั้น จากตรงนี้ไม่ได้คิดจาก 0-1-2 แต่คิดแค่ 1-2 เท่านั้น # แตกแล้วเลือกตัวที่น้อยที่สุดที่มองเห็น</p>
	<p>สมมุติเลือก 7 ตัว 7 แตกต่อเป็น 8 กับ 6 ทั้งหมดมี 2=8, 8=7, 6=1 เลือก 6 ทำแบบนี้ไปเรื่อยๆ</p>
	<p>สุดท้ายก็ออกมาเป็นรูปนี้</p>

CODE PRIM

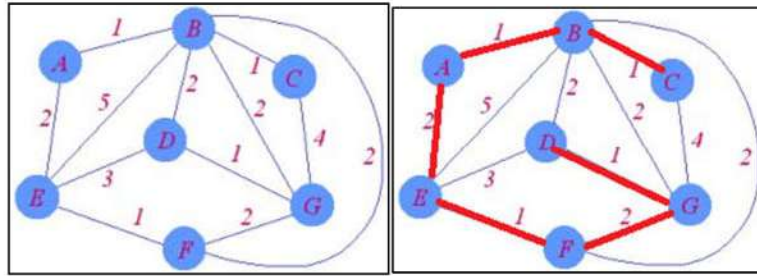
<pre>void Prim() { int path_connect[1000][2]; int List_Node[100]; bool Ans_node[100]; int NumCity = n_vertices; int DistN[100][100];</pre>	<p>ใช้ในการเขียนกราฟ เป็นคำตอบเป็นคู่ ค่าที่ต่ำที่สุดถ้าไปถึงโน้ตนั้น และ index คือหมายเลข node คือใช้ index เดียวพอ ใช้หยุดโปรแกรม จำนวน Node ตาราง distance matrix</p>
<pre>for(int i=0 ; i < NumCity ; i++){ Ans_node[i] = false; List_Node[i] = 99999; } for(int i=0 ; i < NumCity ; i++) { for(int j=0 ; j < NumCity ; j++) { DistN[i][j] = edges[i][j]; if(i == j DistN[i][j] == 0){ DistN[i][j] = 99999;} } }</pre>	<p>กำหนดค่าเริ่มต้นทั้งหมด</p> <p>สร้างตาราง distance matrix</p>
<pre>int min = 99999; int select = 0; Ans_node[select] = true; cout<<select<<" "; for(int i=0 ; i < NumCity ; i++) { if(99999 > DistN[select][i]) { if(List_Node[i] > DistN[select][i] && Ans_node[i] == false) { List_Node[i] = DistN[select][i]; path_connect[i][0] = select; path_connect[i][1] = i; } } } min = 99999; for(int i=0 ; i<NumCity ; i++) { if(min > List_Node[i] && Ans_node[i] == false) { min = List_Node[i]; select = i; } } Ans_node[select] = true; cout<<select<<" ";</pre>	<p>สำหรับ NODE แรกแนวคิดแบบ Dijkstra เริ่มที่ 0 select node 0 start โน้ตเริ่มต้น โน้ตที่ถูกเลือกเป็น 0 แสดงโน้ตที่ถูกเลือก คำตอบ จะมีรูปแบบ เริ่ม 0 แดกไปที่ 1 2 8 5 6 7 3 4 ค้นหาจากโน้ตที่เชื่อมต่อกันไล่ไปเรื่อยๆ</p> <p>ถ้าไปหากันได้ น้อยกว่า 99999 จากรูปคือ 0 ไปเชื่อม 1 กับ 7</p> <p>โน้ตนั้นต้องไม่ถูกเลือกไปแล้ว และ List_Node เก็บค่า node ที่ต่ำที่สุดเอาไว้</p> <p>เลือกเอาตัวที่น้อยที่สุดในรอบการค้นหานี้</p> <p>แสดงตัวที่ถูกเลือก และไป set ตัวที่ถูกเลือกเป็น จริงคือ access ไปแล้ว</p>

<pre> while(true) { for(int i=0 ; i < NumCity ; i++) { if(99999 > DistN[select][i]) { if(List_Node[i] > DistN[select][i] && Ans_node[i] == false) { List_Node[i] = DistN[select][i]; path_connect[i][0] = select; path_connect[i][1] = i; } } } min = 99999; for(int i=0 ; i<NumCity ; i++) { if(min > List_Node[i] && Ans_node[i] == false) { min = List_Node[i]; select = i; } } Ans_node[select] = true; cout<<select<<" "; </pre>	<p>เหมือนกรณี dijista</p>
<pre> bool BreakProgram = true; for(int i=0 ; i < NumCity ; i++) { if(Ans_node[i] == false){ BreakProgram = false; break;} } if(BreakProgram){break;} } </pre>	<p>ทุกโหนดถูกเลือกแล้วก็ให้ปรับให้หยุดการค้นหา</p>
<pre> cout<<endl; for(int i=1 ; i < NumCity ; i++) { cout<<path_connect[i][0] <<" " << path_connect[i][1] <<endl; } cout<<endl; </pre>	<p>แสดงคู่เชื่อมใช้สำหรับเขียนกราฟ โดยเริ่มจาก node ใหญ่ไม่ต้องปริง node นั้นโปรแกรมนี้เริ่มจาก Node 0 จึงไม่ print node 0</p>
<pre> g->initial_graph(9); g->insert_graph(0, 1, 4); g->insert_graph(0, 7, 8); g->insert_graph(1, 0, 4); g->insert_graph(1, 2, 8); g->insert_graph(1, 7, 11); g->insert_graph(2, 1, 8); g->insert_graph(2, 3, 7); g->insert_graph(2, 5, 4); g->insert_graph(2, 8, 2); g->insert_graph(3, 2, 7); g->insert_graph(3, 4, 9); g->insert_graph(3, 5, 14); g->insert_graph(4, 3, 9); g->insert_graph(4, 5, 10); g->insert_graph(5, 2, 4); g->insert_graph(5, 3, 14); g->insert_graph(5, 4, 10); g->insert_graph(5, 6, 2); g->insert_graph(6, 5, 2); g->insert_graph(6, 7, 1); g->insert_graph(6, 8, 6); g->insert_graph(7, 0, 8); g->insert_graph(7, 1, 11); g->insert_graph(7, 6, 1); g->insert_graph(7, 8, 7); g->insert_graph(8, 2, 2); g->insert_graph(8, 6, 6); g->insert_graph(8, 7, 7); g->print_graph(); g->Prim(); </pre>	<pre> 0 : 1 7 1 : 0 2 7 2 : 1 3 5 8 3 : 2 4 5 4 : 3 5 5 : 2 3 4 6 6 : 5 7 8 7 : 0 1 6 8 8 : 2 6 7 </pre> <p>ผลลัพธ์ 0 1 2 8 5 6 7 3 4</p> <pre> 0 1 1 2 2 3 3 4 2 5 5 6 6 7 2 8 </pre> <p>สำหรับคู่นี้เอาไว้เขียนกราฟ</p>

2.2) Kruskal Algorithm

ใช้แก้ปัญหาต้นไม้แบบทอดข้ามน้อยสุด และสามารถเอาไปใช้กับปัญหาการเดินทางของพนักงานขาย หลักการทำงานเป็นแบบกรีดี โดยเริ่มเลือกจากเส้นทางที่สั้นที่สุด แล้วให้เส้นทางเชื่อมต่อกันให้หมด

ตัวอย่างที่ 1

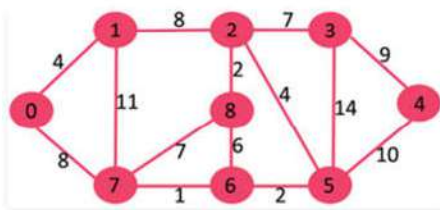


- จากรูปเลือกเส้นทางที่สั้นที่สุด คือ 1 มีคู่อันดับดังนี้ A-B, B-C, D-G, E-F แล้วทำการเชื่อมต่อกันได้ผลลัพธ์เป็น A-B-C, D-G, E-F
- จากรูปเลือกเส้นทางที่สั้นเป็นอันดับต่อมา คือ 2 มีคู่อันดับดังนี้ A-E, B-D, B-F, E-G, G-F แล้วทำการเชื่อมต่อกันได้ผลลัพธ์เป็น D-G-F-E-A-B-C แล้วก็เชื่อมต่อกันหมดแล้ว ดังรูปข้างบน

ตัวอย่างที่ 2

- Sort แล้วเก็บเส้นทางเป็นคู่ๆ ลงไปใน array ตาม index
- สร้าง Array 2 มิติ เอา Weight มาใส่ตามลำดับ เอาไว้เก็บผลการ sort ข้างบน
- สร้าง Array 2 มิติ เป็น array ที่เอาไว้เก็บกราฟใหม่ โดย Add เข้าไปใน array ของกราฟใหม่ โดยการ add แต่ละครั้งต้องพิจารณาว่าเกิด loop ในกราฟไหม ถ้าเกิด loop ในกราฟไม่ต้อง add เข้าไป

โจทย์



Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

	เลือก 7-6 น้อยสุด ยาว 1		เลือก 8-2 ยาว 2
	เลือก 6-5 ยาว 2 เริ่มเชื่อมกันแล้ว		เลือก 0-1 ยาว 4
	เลือก 2-5 ยาว 4 ให้เชื่อมกัน ต่อมา เลือก 8-6 เชื่อมไปแล้ว ดังนั้นจึงไม่สนใจเส้นนี้		เลือก 2-3 ยาว 7 ต่อมา เลือก 7-8 เชื่อมไปแล้ว ดังนั้นจึงไม่สนใจเส้นนี้
	เลือก 0-7 ยาว 8 ให้เชื่อมกัน ต่อมา เลือก 1-2 เชื่อมไปแล้ว ดังนั้นจึงไม่สนใจเส้นนี้		เลือก 3-4 ให้เชื่อมกัน ยาว 9 เชื่อมครบหมดแล้ว

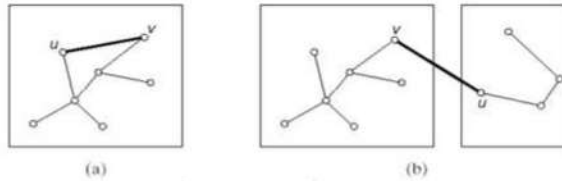
CODE KRUSKAL

<pre>int my_path[100][3]; int New_egde[100][100]; int size_my_path; bool connect; bool visited_k[100][0]; int pointA; int pointB;</pre>	<p>สร้างตัวแปรสำหรับให้ เกิดการท่องของ DFS แล้ว New_egde คือตารางคำตอบ</p>
<pre>void check_connect(int start) { visited_k[start] = true; for (int j=0 ; j < n_vertices ; j++) { if (New_egde[start][j] > 0 && visited_k[j] == false) { if(j == pointB){ connect = true; } check_connect(j); } } }</pre>	<p>เป็น DFS ท่องไปจบเจออีกตัว ถ้าสามารถหาอีกตัวได้แสดงว่ามันเชื่อมกันแล้ว ก็ไม่ต้องเพิ่มตัวใหม่เข้าไป แต่ถ้าไม่เชื่อมก็เพิ่มตัวใหม่เข้าไป โดยการท่องจะท่องบน " New_egde " หรือแผนภาพคำตอบ</p>
<pre>void Kruskal() { size_my_path = 0; for(int i=0 ; i < n_vertices ; i++){ for(int j=0 ; j < n_vertices ; j++){ New_egde[i][j] = 0; } } for(int i=0 ; i < n_vertices ; i++) { for(int j=0 ; j < n_vertices ; j++) { if(edges[i][j] > 0) { my_path[size_my_path][0] = edges[i][j]; my_path[size_my_path][1] = i; my_path[size_my_path][2] = j; size_my_path++; } } } }</pre>	<p>New_egde สำหรับเก็บเส้นที่เชื่อมต่อกันที่เลือกเป็นคำตอบ</p> <p>กำหนดค่าเริ่มต้น my_path[0] ขนาดของ node ระหว่าง 2 จุด my_path[1] และ [2] เก็บ node เชื่อมระหว่าง 2 ที่</p>
<pre>for(int i=0 ; i < size_my_path ; i++) { for(int j=0 ; j < size_my_path ; j++) { if (my_path[i][0] < my_path[j][0]) { int T1 = my_path[i][0]; int T2 = my_path[i][1]; int T3 = my_path[i][2]; my_path[i][0] = my_path[j][0]; my_path[i][1] = my_path[j][1]; my_path[i][2] = my_path[j][2]; my_path[j][0] = T1; my_path[j][1] = T2; my_path[j][2] = T3; } } }</pre>	<p>ดำเนินการ Sort จากน้อยไปหามาก</p>

<pre> for(int i=0 ; i < size_my_path ; i++) { connect = false; for(int i=0 ; i < n_vertices ; i++){visited_k[i] = false;} pointA = my_path[i][1]; pointB = my_path[i][2]; check_connect(pointA); if(!connect) { New_egde[my_path[i][1]][my_path[i][2]] = my_path[i][0]; New_egde[my_path[i][2]][my_path[i][1]] = my_path[i][0]; } } </pre>	<p>กำหนด connect = false คือไม่เชื่อมกัน</p> <p>visited_k reset ไว้ก่อน</p> <p>กำหนดจุดเชื่อม A กับ B</p> <p>ถ้าเชื่อมกันแล้วไม่ add เข้าไป</p> <p>ถ้าไม่เชื่อมให้ add เข้าไปในตารางใหม่</p>
<pre> for(int i=0 ; i < n_vertices ; i++) { for(int j=0 ; j < n_vertices ; j++){ cout<<New_egde[i][j]<<" "; } cout<<endl; } </pre>	<p>แสดงตารางการเชื่อมต่อ print ตารางใหม่</p> <pre> 0 4 0 0 0 0 8 0 4 0 0 0 0 0 0 0 0 0 7 0 4 0 0 2 0 0 7 0 9 0 0 0 0 0 0 9 0 0 0 0 0 0 4 0 0 0 2 0 0 0 0 0 0 0 2 0 1 0 8 0 0 0 0 0 1 0 0 0 0 2 0 0 0 0 0 0 </pre>
<pre> graph *g = new graph(); g->initial_graph(9); g->insert_graph(0, 1, 4); g->insert_graph(0, 7, 8); g->insert_graph(1, 0, 4); g->insert_graph(1, 2, 8); g->insert_graph(1, 7, 11); g->insert_graph(2, 1, 8); g->insert_graph(2, 3, 7); g->insert_graph(2, 5, 4); g->insert_graph(2, 8, 2); g->insert_graph(3, 2, 7); g->insert_graph(3, 4, 9); g->insert_graph(3, 5, 14); g->insert_graph(4, 3, 9); g->insert_graph(4, 5, 10); g->insert_graph(5, 2, 4); g->insert_graph(5, 3, 14); g->insert_graph(5, 4, 10); g->insert_graph(5, 6, 2); g->insert_graph(6, 5, 2); g->insert_graph(6, 7, 1); g->insert_graph(6, 8, 6); g->insert_graph(7, 0, 8); g->insert_graph(7, 1, 11); g->insert_graph(7, 6, 1); g->insert_graph(7, 8, 7); g->insert_graph(8, 2, 2); g->insert_graph(8, 6, 6); g->insert_graph(8, 7, 7); g->print_graph(); g->Kruskal(); </pre>	<pre> 0 : 1 7 1 : 0 2 7 2 : 1 3 5 8 3 : 2 4 5 4 : 3 5 5 : 2 3 4 6 6 : 5 7 8 7 : 0 1 6 8 8 : 2 6 7 </pre>

10.4 การตรวจสอบ CYCLE ในต้นไม้

ปกติการเพิ่ม Edge u และ v เข้าไปเพื่อเชื่อมระหว่างต้นไม้ย่อย 2 ต้น ต้องทำการเช็ค ว่า vertex u และ v อยู่ในต้นไม้ย่อยเดียวกันหรือไม่ เพราะเกิดการเชื่อมนั้นเป็นการเชื่อมกันเองภายในต้นไม้เดียวกันเท่ากับต้นไม้ทั้งสองไม่ได้เชื่อมกัน ดังรูป a สำหรับ รูป b คือต้นไม้ย่อยทั้งสองสามารถเชื่อมกันได้ ดังนั้นต้องมีการตรวจสอบ cycle ของต้นไม้ เพื่อป้องกันต้นไม้มาเชื่อมกันเอง แต่ต้นไม้ทั้งสองต้องเป็นต้นไม้ที่แบบไม่มี loop อยู่แล้ว



Disjoint-set data structure หรือ union-find data structure หรือ merge-find set เป็น อัลกอริทึมตรวจสอบว่า Vertex ใดอยู่ในต้นไม้ย่อยเดียวกันบ้าง Disjoint sets คือ เซตย่อยซึ่งไม่มีสมาชิกซ้ำกันเลย หรือไม่มี loop เป็น acyclic graph

คุณสมบัติของ disjoint sets

- หากหีบเซตย่อยใดมา intersect กันจะได้เซตว่างเสมอ
- หาก union ทุกๆ เซตย่อยจะได้ universal set

Operation ของ disjoint sets

- Make (i) ทำหน้าที่สร้างเซตย่อยจากเซตทั้งหมดโดยมี i เป็นสมาชิก
- Union (i, j) ทำหน้าที่ในการ union เซต i และ เซต j เข้าด้วยกัน

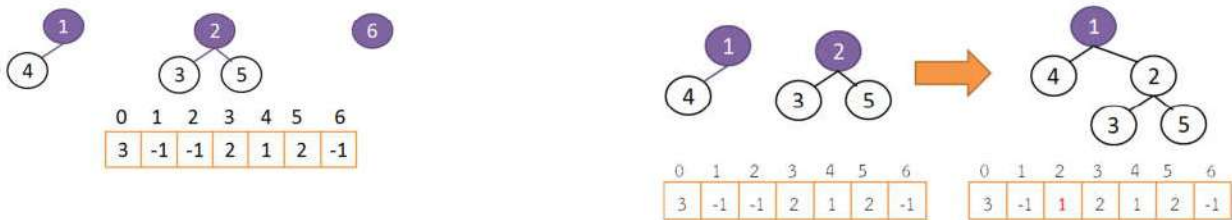
ตัวอย่าง $S = \{1, 2, 3, 4, 5, 6\}$ ใช้ make(i) เพื่อสร้าง set {i} บนสมาชิกแต่ละตัวของ S เพื่อให้ได้ 6 เซต {1}, {2}, {3}, {4}, {5}, {6}

จากนั้นทำ union (1, 4) และ union (5, 2) = {1, 4}, {5, 2}, {3}, {6} และถ้า union(4, 5) และ union (3, 6) ต่ออีก = {1, 4, 5, 2}, {3, 6}

การแสดง Disjoint sets = { {1,4}, {2,3,5}, {6} } ดังรูปซ้ายมือ

Parent = -1 คือ root

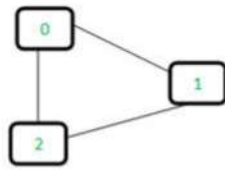
ถ้ามีตัวเลขอื่นตัวนั้นเป็น Parent โดยต้นไม้ย่อยต้องเลือกสมาชิก 1 ตัวเป็น Root



การ Union ระหว่าง 2 เซต จะต้องทำการเปลี่ยน root ของต้นไม้ต้นหนึ่งให้กลายเป็น child ของ root ในต้นไม้ต้นหนึ่ง เช่น Union (1, 2) รูปขวามือ

Algorithm Union-Find

ตัวอย่างที่ 1



	0	1	2
0	X	1	1
1	1	x	1
2	1	1	x

1) เริ่มต้น Parent ของทุกตัวเป็น -1

หลักการทํางาน คือ จะสร้าง Array 1 มิติ โดย index คือ node / value คือ parent

0	1	2
-1	-1	-1

2) เชื่อม 0-1

ให้ 1 เป็นพ่อของ 0

$0 = \text{find}(0)$; วิ่งหาพ่อที่ปลายทางเช่น 0 มี พ่อเป็น 2 และ 2 มีพ่อเป็น -1 ตอบ 2 แต่กรณีนี้ไม่มีพ่อเลยตอบตัวเอง

$1 = \text{find}(1)$;

แล้วดำเนินขั้นตอน Union (0, 1) เป็นตัวเปลี่ยน array โดยถ้าเป็น -1 ให้เอาอีกตัวใส่เป็นพ่อ กรณีนี้คือ เอา 1 ใส่เป็นพ่อของ 0

อธิบายคำสั่งฟังก์ชัน Find

0	1	2
1	2	-1

คำสั่ง Find คือหา Parent ตัวสุดท้าย หรือจนกว่าเจอ "-1" $\text{Find}(0) = 2$ และ $\text{Find}(2) = 2$ ได้คำตอบเดียวกัน ถ้าเชื่อม 2 กับ 0 โดยเรา find 0 กับ 2 เพื่อดูว่าถ้าเชื่อม 2 ตัวนี้จะ loop ไหม

0	1	2
1	-1	-1

คือหา Parent ตัวสุดท้าย หรือจนกว่าเจอ "-1" $\text{Find}(0) = 1$ และ $\text{Find}(1) = 1$ ได้คำตอบเดียวกัน กรณีเชื่อมไม่ได้ เพราะมันซ้ำกัน

0	1	2
-1 → 1	-1	-1

3) เชื่อม 1-2

ให้ 2 เป็นพ่อของ 1

$1 = \text{find}(1)$;

$2 = \text{find}(2)$;

แล้วดำเนินขั้นตอน Union (1, 2) เป็นตัวเปลี่ยน array โดยถ้าเป็น -1 ให้เอาอีกตัวใส่เป็นพ่อ กรณีนี้คือ เอา 2 ใส่เป็นพ่อของ 1

0	1	2
1	-1 → 2	-1

4) เชื่อม 0-2

ให้ 2 เป็นพ่อของ 1

$2 = \text{find}(0)$;

$2 = \text{find}(2)$; กรณีนี้ถ้าเท่ากันให้ แสดงค่าเป็น loop ไม่ต้องหาต่อแล้ว

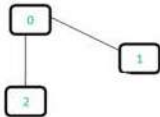
พยายามเชื่อมให้เป็น Loop จะสังเกตว่า พยายามเริ่มจาก 0 ไป 1, 1 ไป 2, 2 ไป 0 เป็นลักษณะ Loop ถ้าเกิดเป็น Loop แสดงว่าเกิด Loop

กรณีที่มี LOOP คือ 1 node เมื่อเชื่อมไปถึง root ต้องมีแค่ตัวเดียวเท่านั้น

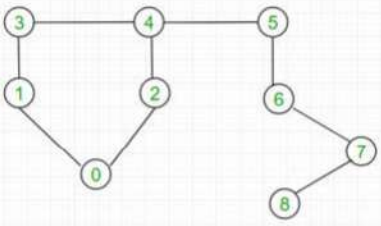
กรณี 0 root 3, กรณี 1 root 3, กรณี 2 root 3, กรณี 3 root 3 กรณีนี้เชื่อมกันหมด จะมี root ตัวเดียวกันหมด

0	1	2
1	2	-1

ตัวอย่างที่ 2

	<table><tr><td></td><td>0</td><td>1</td><td>2</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>2</td><td>1</td><td>0</td><td>0</td></tr></table>		0	1	2	0	0	1	1	1	1	0	0	2	1	0	0
	0	1	2														
0	0	1	1														
1	1	0	0														
2	1	0	0														
<p>1) เริ่มต้น Parent ของทุกตัวเป็น -1</p>	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>-1</td><td>-1</td><td>-1</td></tr></table>	0	1	2	-1	-1	-1										
0	1	2															
-1	-1	-1															
<p>2) เชื่อม 0-1 มีเส้นเชื่อมระหว่าง 0 กับ 1</p> <p>ให้ 1 เป็นพ่อของ 0</p> <p>0 = find(0);</p> <p>1 = find(1);</p> <p>Union (0, 1) กรณีนี้คือ เอา 0 ใส่เป็นพ่อของ 1</p>	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>-1</td><td>0</td><td>-1</td></tr></table>	0	1	2	-1	0	-1										
0	1	2															
-1	0	-1															
<p>3) เชื่อม 0-2 มีเส้นเชื่อมระหว่าง 0 กับ 2</p> <p>ให้ 2 เป็นพ่อของ 0</p> <p>1 = find(0);</p> <p>2 = find(2); พ่อแม่ไม่ตรงกัน</p> <p>Union (0, 2) กรณีนี้คือ เอา 0 ใส่เป็นพ่อของ 2</p> <p>หมดการเชื่อมต่อแล้ว ดังนั้นเป็นกราฟเชิงเดี่ยว</p>	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>-1</td><td>0</td><td>0</td></tr></table>	0	1	2	-1	0	0										
0	1	2															
-1	0	0															

ตัวอย่างที่ 3

																			
1) เริ่มต้น Parent ของทุกตัวเป็น -1	<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	0	1	2	3	4	5	6	7	8	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8											
-1	-1	-1	-1	-1	-1	-1	-1	-1											
2) เชื่อม 0-1 เส้นเชื่อม 0 กับ 1 0 = find(0) 1 = find(1) Union (0, 1)	<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	0	1	2	3	4	5	6	7	8	-1	0	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8											
-1	0	-1	-1	-1	-1	-1	-1	-1											
3) เชื่อม 0-2 เส้นเชื่อม 0 กับ 2 1 = find(0) 2 = find(2) Union (0, 2)	<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>0</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	0	1	2	3	4	5	6	7	8	-1	0	0	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8											
-1	0	0	-1	-1	-1	-1	-1	-1											
4) เชื่อม 1-3 เส้นเชื่อม 1 กับ 3 0 = find(1) 3 = find(3) Union (1, 3)	<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>0</td><td>0</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	0	1	2	3	4	5	6	7	8	-1	0	0	0	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8											
-1	0	0	0	-1	-1	-1	-1	-1											
5) เชื่อม 3-4 เส้นเชื่อม 3 กับ 4 0 = find(3) 4 = find(4) Union (3, 4)	<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr><tr><td>-1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>	0	1	2	3	4	5	6	7	8	-1	0	0	0	0	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8											
-1	0	0	0	0	-1	-1	-1	-1											
6) เชื่อม 2-4 เส้นเชื่อม 2 กับ 4 0 = find(2) 0 = find(4) ถ้าได้คำตอบเดียวกันแสดงว่าเชื่อมกันแล้ว																			

CODE UNION-FIND

<pre>int parent[1000]; int alledge[1000][1000]; int size_alledge;</pre>													
<pre>int find(int i) { if (parent[i] == -1){ return i; } return find(parent[i]); }</pre>	<p>วิ่งไปเรื่อยๆจนกว่าจะเจอ -1 เช่น</p> <table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>1</td><td>2</td><td>-1</td></tr></table> <p>พ่อของ 0 คือ 2 โดยวิ่งจาก 0 ไป 1 และจาก 1 ไป 2 และหยุดที่ 2 เพราะเป็น -1</p>	0	1	2	1	2	-1						
0	1	2											
1	2	-1											
<pre>void Union(int x, int y) { parent[x] = y; }</pre>	<p>การรวมคือการชี้ตัวถัดไปเช่น</p> <table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>-1</td><td>-1</td><td>-1</td></tr></table> <p>Union(0,1) คือ พ่อของ 0 คือ 1 กลายเป็น</p> <table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>1</td><td>-1</td><td>-1</td></tr></table>	0	1	2	-1	-1	-1	0	1	2	1	-1	-1
0	1	2											
-1	-1	-1											
0	1	2											
1	-1	-1											
<pre>int isCycle() { for(int i=0 ; i <1000 ; i++){ parent[i] = -1;} size_alledge = 0; for(int i=0 ; i < n_vertices ; i++) { for(int j=i+1 ; j < n_vertices ; j++) { if(edges[i][j] > 0) { alledge[size_alledge][0] = i; alledge[size_alledge][1] = j; size_alledge++; } } } }</pre>	<p>//reset</p> <p>//find all path</p> <p>การคำนวณจะคิดแค่ส่วนด้านบนของตารางทั้งหมด</p> <p>0 : 1 2 1 : 0 2 2 : 0 1 ดังนั้นจึงได้ผลลัพธ์ดังนี้</p> <p>0 1 0 2 1 2</p>												
<pre>for(int i = 0 ; i < size_alledge ; i++) { int x = find(alledge[i][0]); int y = find(alledge[i][1]); cout<<x<<" "<<y<<endl; if (x == y) { cout<<"graph contains cycle"; return 0; } Union(x, y); for(int ii=0 ; ii < n_vertices ; ii++){ cout<<parent[ii]<<" "; } cout<<endl; } cout<<"graph doesn't contain cycle"; return 0; }</pre>	<p>พิจารณาเฉพาะเส้นเชื่อมกันเท่านั้น ส่วนเส้นไม่เชื่อมไม่พิจารณา Alledge คือ กรองข้อมูลออก ให้หา x กับ y</p> <p>ถ้าได้คำตอบเดียวกันแสดงว่าเชื่อมกันแล้ว คือ มีพ่อเดียวกันแสดงว่าเกิดปัญหา loop</p> <p>ถ้าหาจนหมดแล้วไม่เจอพ่อแม่เดียวกัน แสดงว่าไม่มี loop</p>												
<pre>graph *g = new graph(); g->initial_graph(3); g->insert_graph(0, 1, 1); g->insert_graph(0, 2, 1); g->insert_graph(1, 0, 1); g->insert_graph(1, 2, 1); g->insert_graph(2, 0, 1); g->insert_graph(2, 1, 1); g->print_graph(); g->isCycle();</pre>													

แบบฝึกหัด

- 1) จงเขียนโปรแกรมกราฟ Transitive Closure
- 2) จงเขียนโปรแกรมกราฟ WARSHALL
- 3) จงเขียนโปรแกรมกราฟ Dijkstra
- 4) จงเขียนโปรแกรมกราฟ PRIM
- 5) จงเขียนโปรแกรมกราฟ Kruskal
- 6) จงเขียนโปรแกรมกราฟ Union-Find