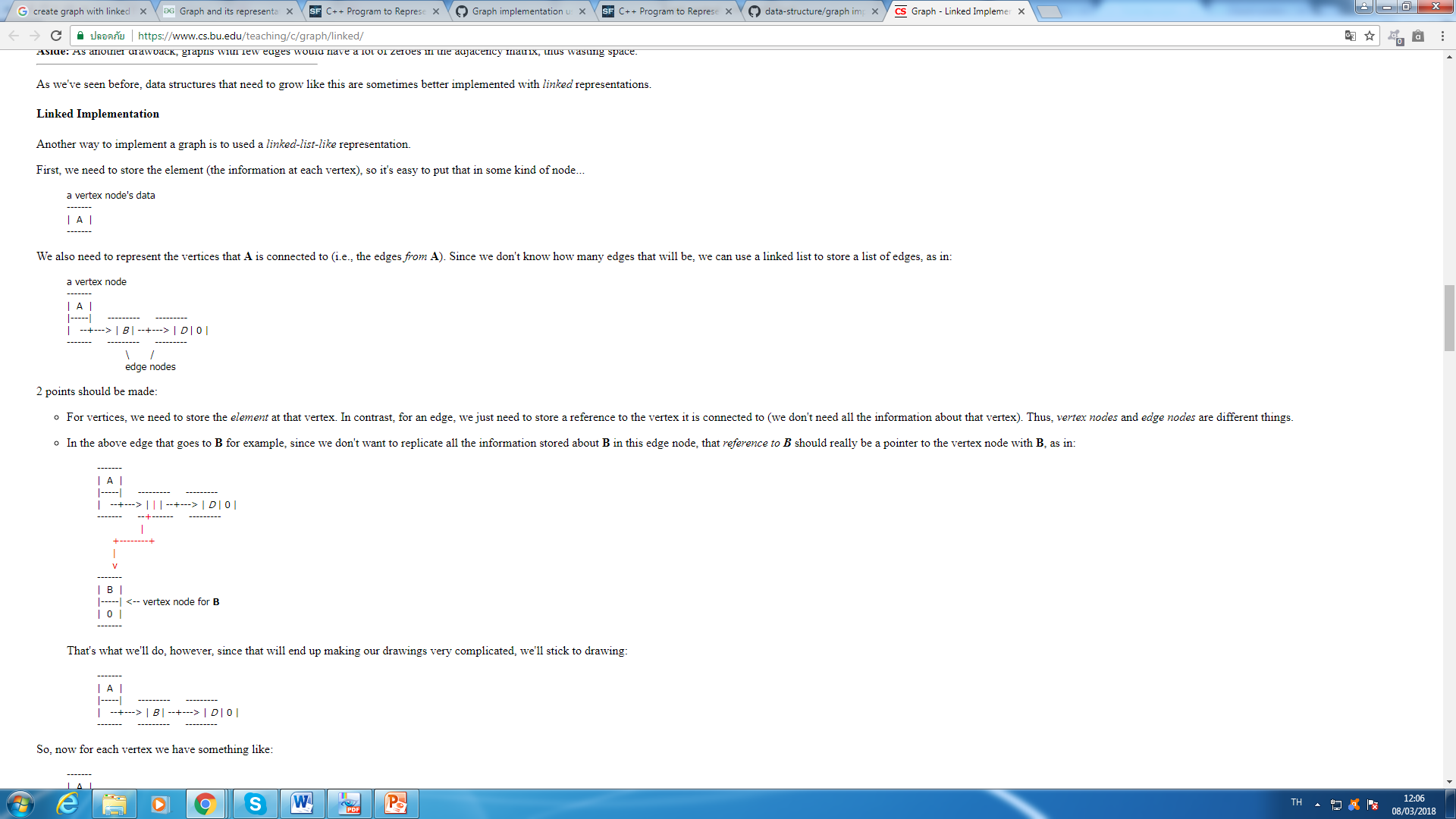
Graph Implementation

1. Array
   * ใช้โครงสร้างการเก็บแบบ adjacency matrix

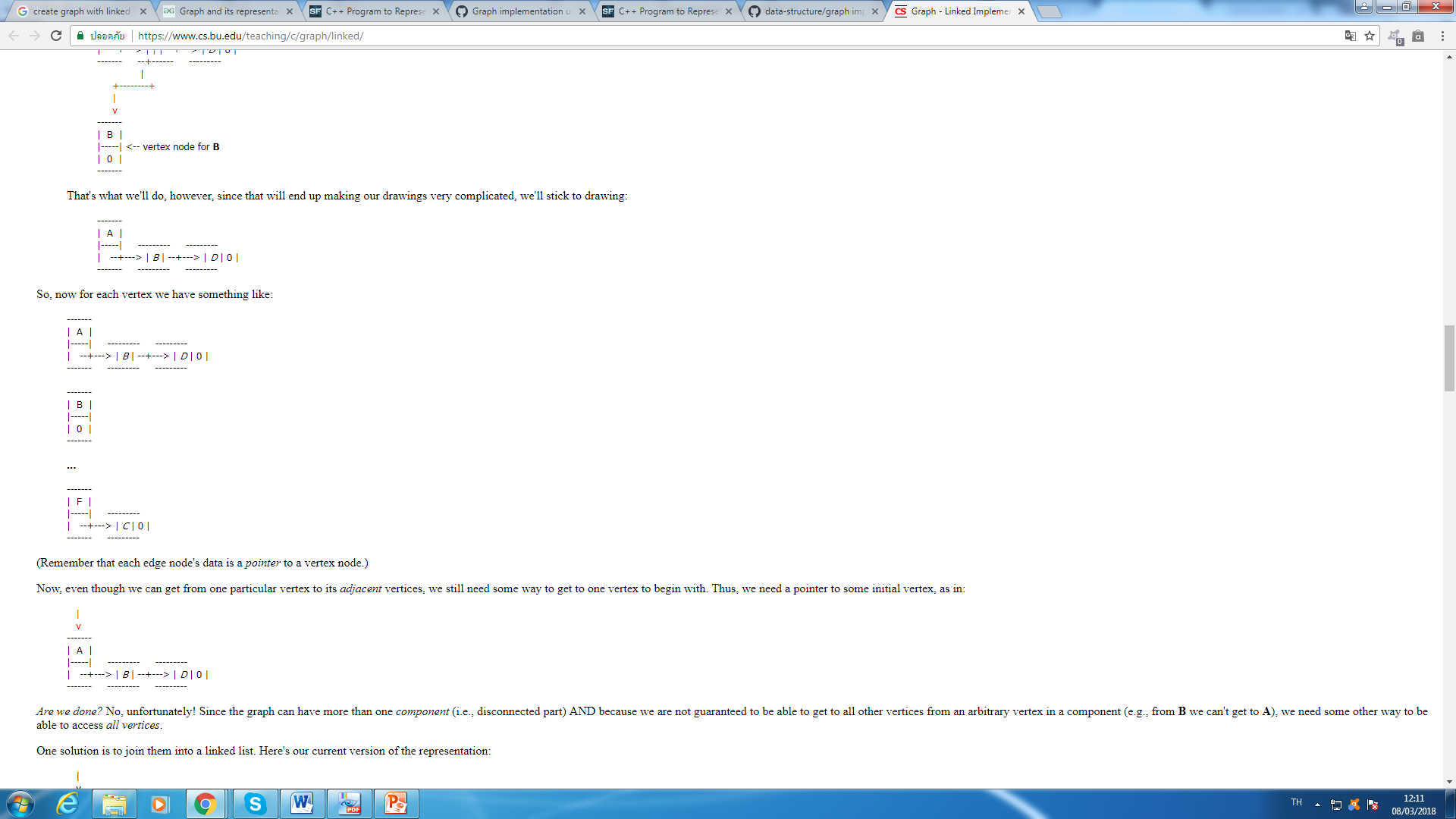
* ใช้ array 1 มิติสำหรับเก็บ vertex
* ใช้ array 2 มิติสำหรับเก็บ edges

1. Linked List

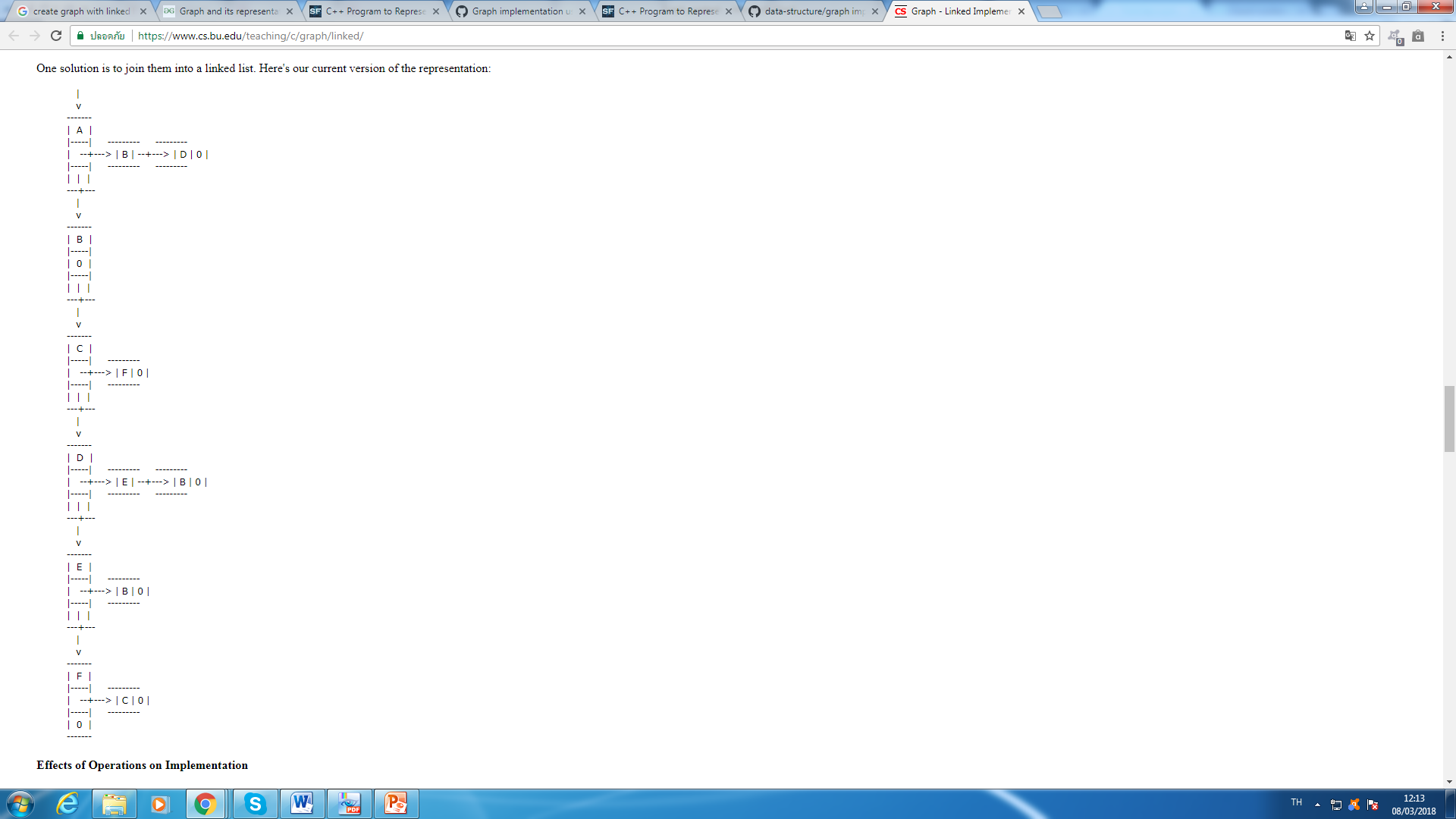
* ใช้โครงสร้างการเก็บแบบ adjacency List
* ต้องมีการแทนว่าโหนด A มีการเชื่อมต่อไปยังโหนดใดบ้าง



* ดังนั้นแต่ละโหนด จะมีโครงสร้างลักษณะนี้



* จะทำอย่างไรให้สามารถท่องไปยังทุกโหนดได้จากโหนด วิธีที่ทำได้ ต้องให้โครงสร้างข้อมูลเป็นดังนี้



1. Vector

การใช้ vector ในการสร้างกราฟเพื่อช่วยให้การสร้างเร็วขึ้น ซึ่ง vector จะอยู่ใน STL ไลบรารี่

**ตัวอย่างที่ 1** undirected and unweighted graph

vector จะถูกใช้ในการสร้างกราฟด้วยโครงสร้างการจัดเก็บข้อมูลในกราฟจะเป็นแบบ Adjacency List กำหนดให้มีกราฟดังรูปด้านล้างนี้ เป็น ที่มีโหนดทั้งหมด 5 โหนด

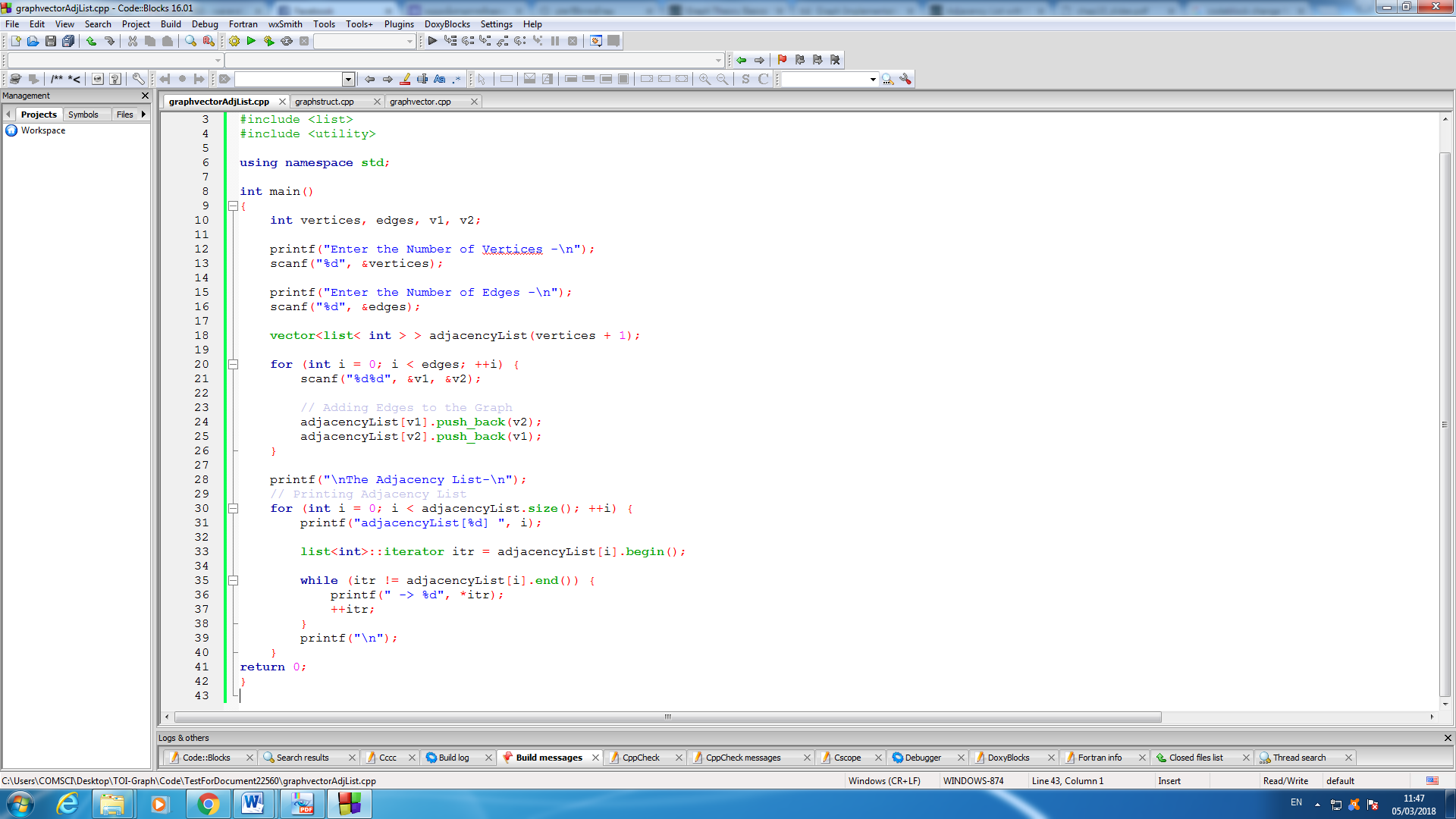
8

จะได้ กราฟที่แทนด้วย adjacency list ดังนี้

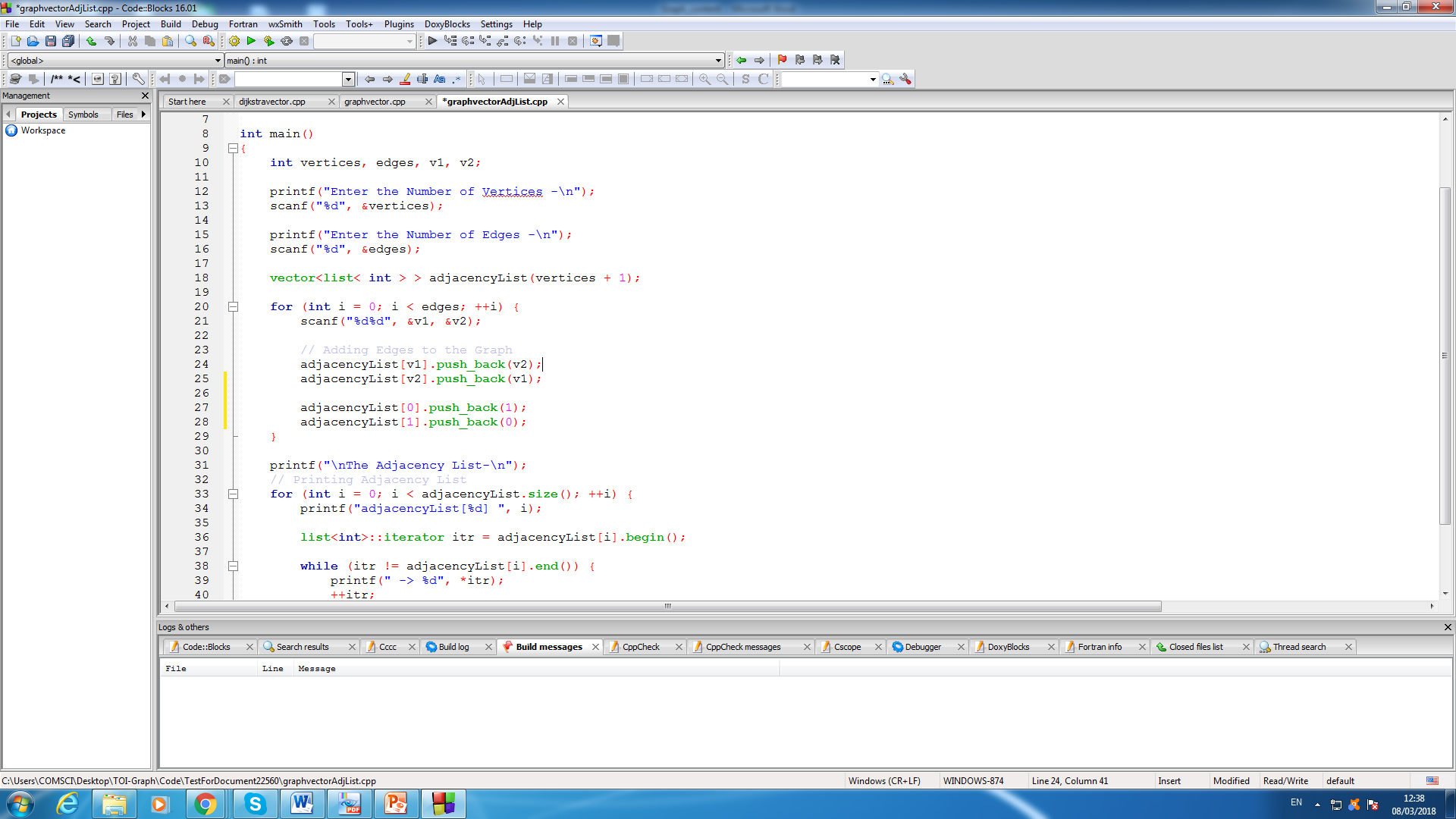
9 (1)

Concept

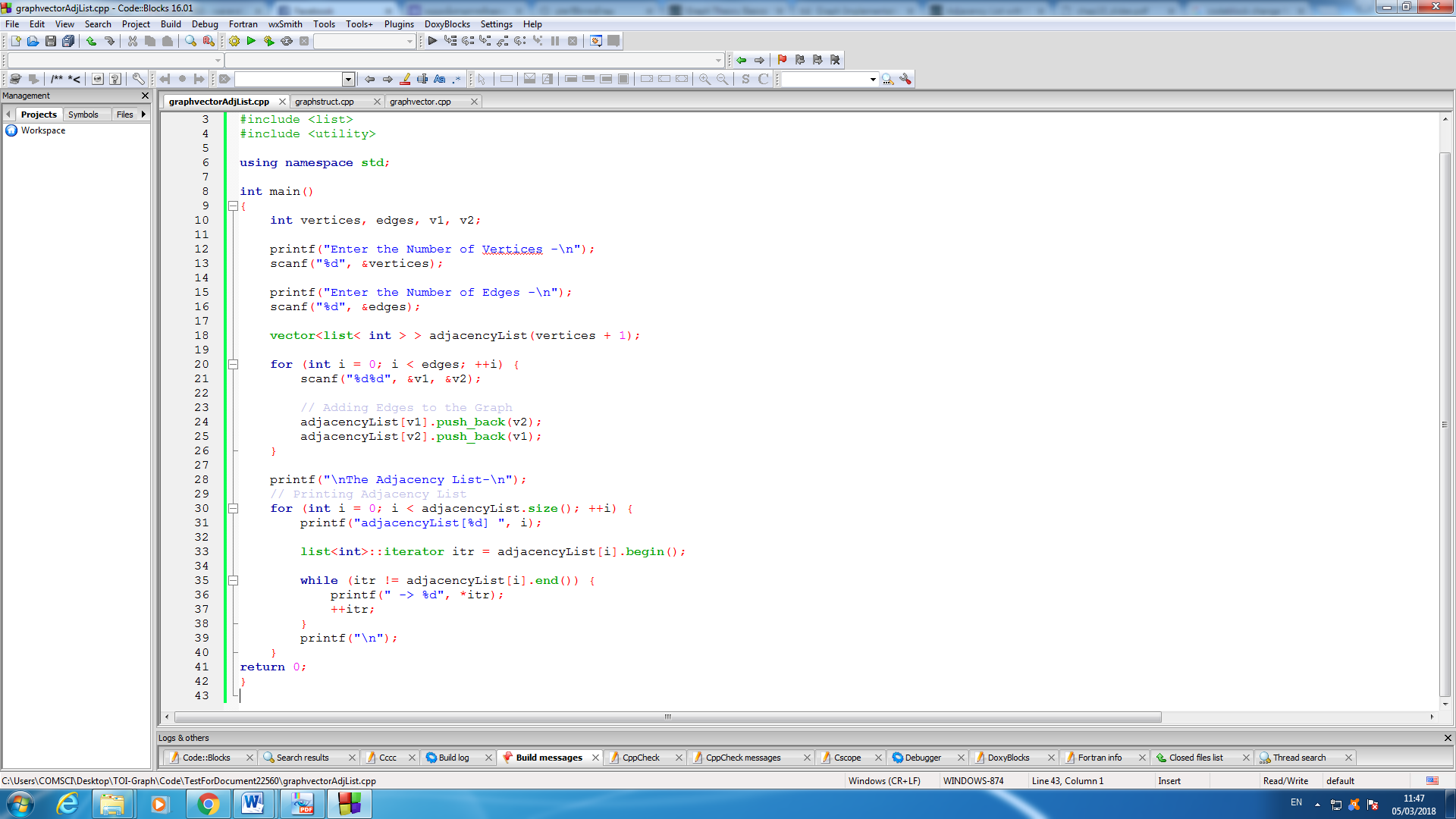
1. vector of list of nodes

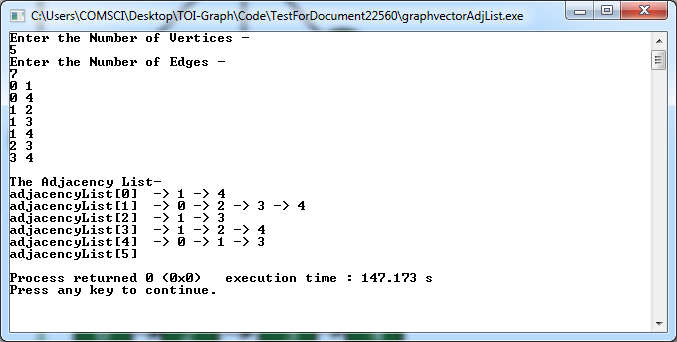


2. ในแต่ละโหนดมี list ของ neighbours



การพิมพ์ข้อมูลจาก adjacecyList สามารถทำได้ดังนี้

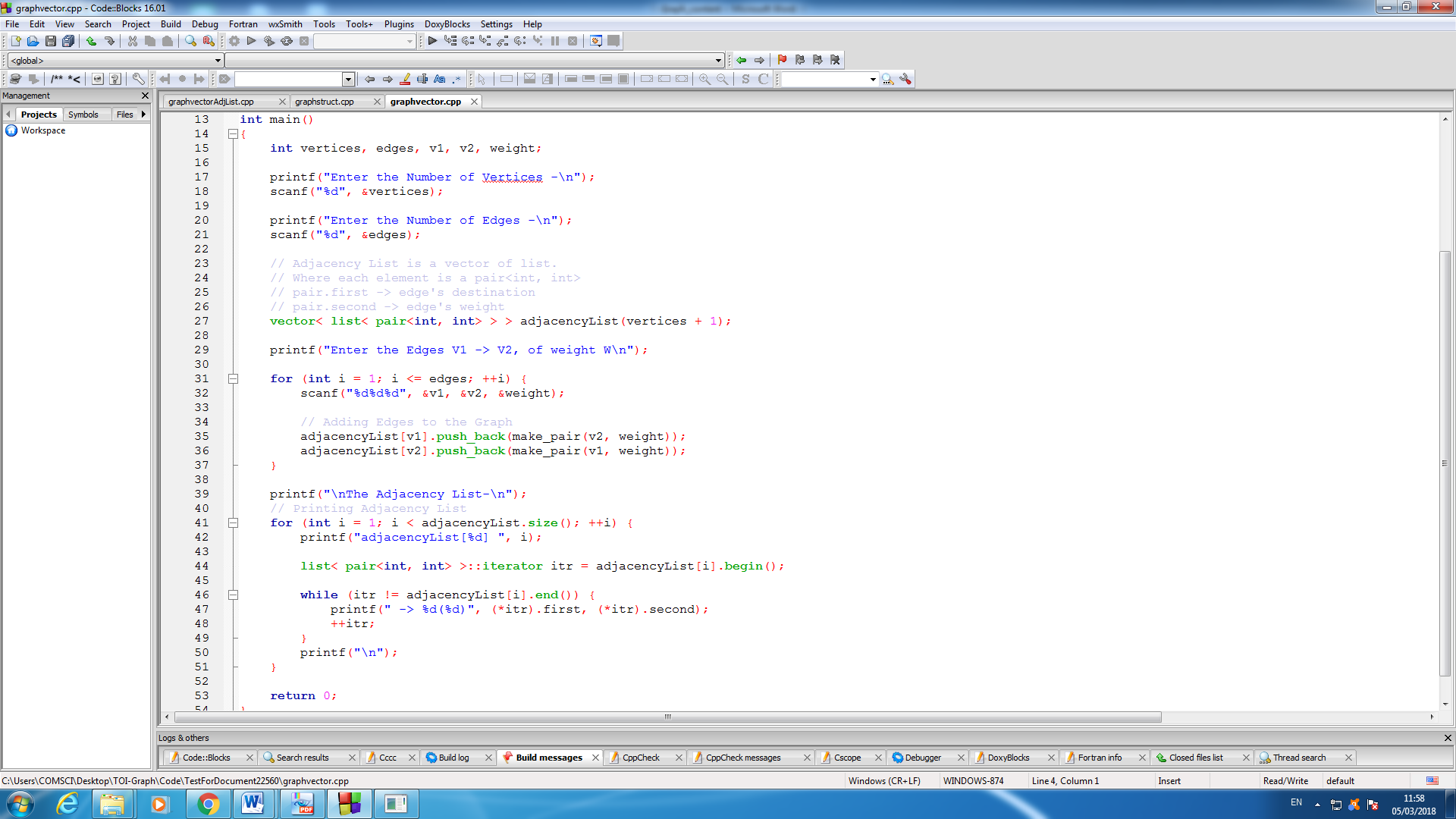




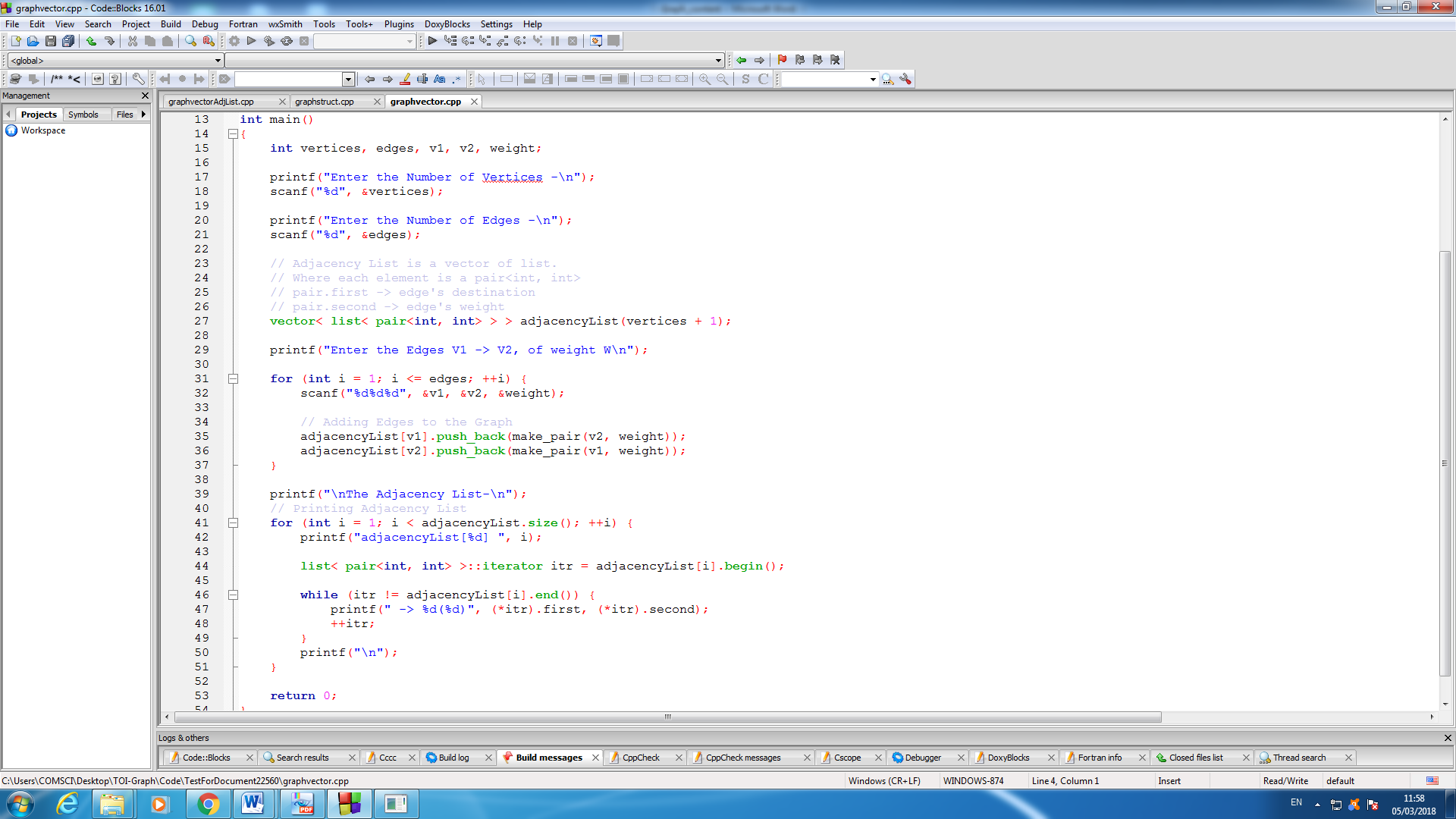
**ตัวอย่างที่** 2 undirected and weighted graph

Concept

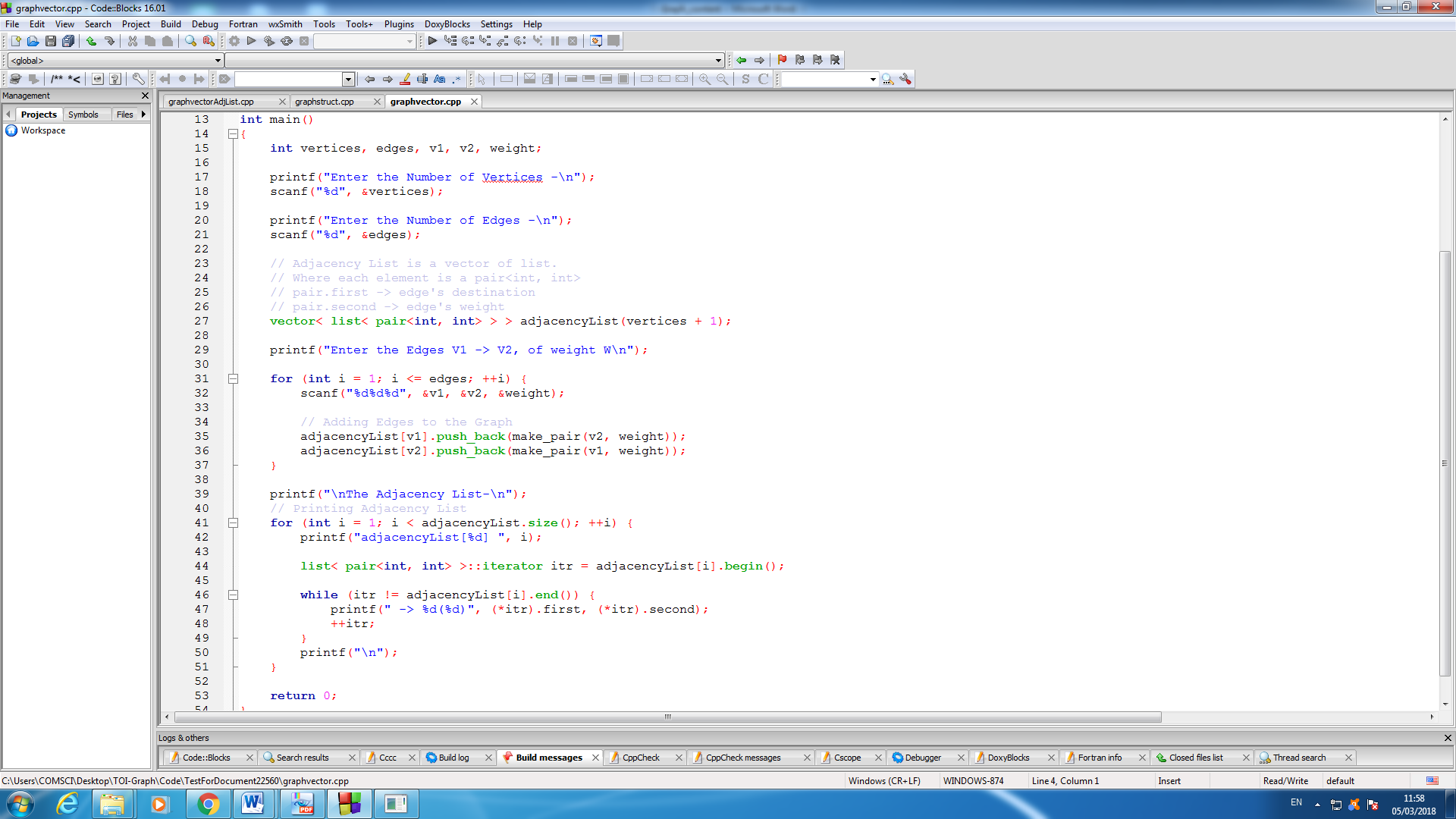
1. vector of list of nodes

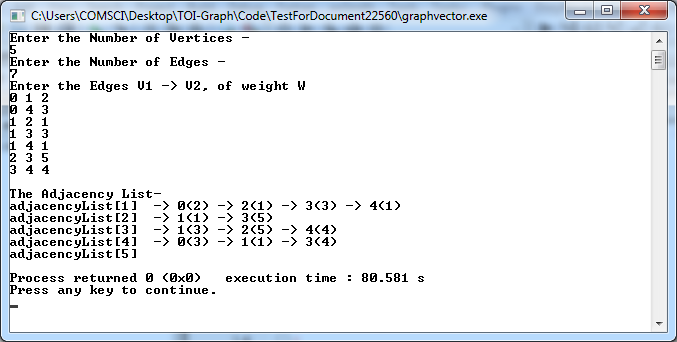


2. ในแต่ละโหนดมี list ของ neighbours



การพิมพ์ข้อมูลจาก adjacecyList สามารถทำได้ดังนี้





**การท่องเข้าไปในกราฟ (Traverse Graph)**

เมื่อเราสร้างกราฟที่มีโหนดและมีการเชื่อมต่อโหนดเหล่านี้เข้าด้วยกันแล้ว อีกสิ่งหนึ่งที่เรา จะขาดไม่ได้คือ การเข้าถึงโหนดต่าง ๆ เหล่านี้ การเข้าหากราฟโดยทั่วไปทำได้ 2 วิธี คือ การการเข้าถึงแบบที่เรียกว่า การท่องเข้าไปในกราฟแบบแนวลึก (depth-first search: DFS) และการท่องเข้าไปในกราฟแบบแนวลึก (breadth-first search: BFS)

1. การท่องเข้าไปในกราฟแบบแนวลึก (Depth-First Traversal)

ในการเข้าถึงแบบ depth-first search นั้นเป็นการเข้าหาตามความลึกของเส้นทางจาก node ที่ มีอยู่ในกราฟไปจนกว่าจะไปไม่ได้แล้วในเส้นทางนั้น แล้วจึงเข้าหาเส้นทางอื่น ๆ ที่เหลืออยู่ โดยจะใช้ความลึกเป็นหลักเช่นเดียวกัน ทำไปเรื่อย ๆ จนกว่าจะครบทุกเส้นทาง สำหรับ โครงสร้างที่ใช้เก็บโหนดที่มีอยู่ในกราฟ เราจะใช้ stack เป็นตัวเก็บทั้งนี้ก็เนื่องจากว่า stack มี การเก็บข้อมูลแบบ LIFO ซึ่งทำให้การเรียกโหนดมาตรวจสอบเป็นไปได้ง่ายขึ้น วิธีการท่องกราฟแบบ depth-first search มีวิธีดังนี้

เริ่มต้นด้วยโหนดใด ๆ ที่มีอยู่ในกราฟโดยจะทำกระบวนการต่อไปนี้

กฎข้อที่ 1

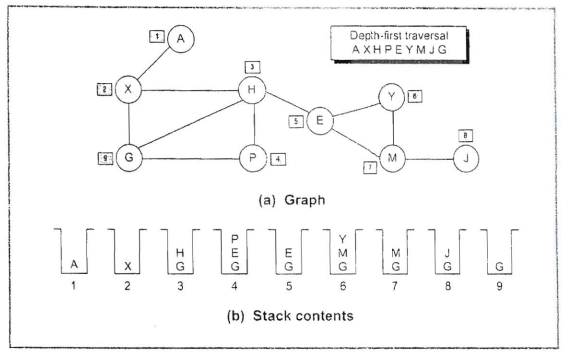
* + Visit โหนดนี้
  + เก็บโหนดนี้ไว้ใน stack
  + กำหนดให้โหนดนี้ได้ถูก visit แล้ว เพื่อไม่ให้เกิดการ visit ที่ซ้ำซ้อน
  + หลังจากนั้นเราก็ไปยังโหนดที่เป็นเพื่อนบ้าน (adjacent) ของโหนดนี้ ที่เรายังไม่ได้ visit ทำการเหมือนกับที่ทำก่อนหน้านี้ คือ
    - Visit vertex นี้
    - เก็บ vertex นี้ไว้ใน stack
    - กำหนดให้ vertex นี้ได้ถูก visit แล้ว เพื่อไม่ให้เกิดการ visit ที่ซ้ำซ้อน
  + ทำไปเรื่อย ๆ จนทำไม่ได้แล้ว

กฎข้อที่ 2

* + ถ้าทำตามกฎข้อที่ 1 ไม่ได้ ให้ดึงข้อมูลออกจาก stack (ถ้า stack ยังมีข้อมูลอยู่)

กฎข้อที่ 3

* + ถ้าไม่สามารถทำได้ทั้ง กฎข้อที่ 1 และ กฎข้อที่ 2 แสดงว่ากระบวนการท่องกราฟได้สิ้นสุดลงแล้ว ก็สามารถยุติการท่องกราฟได้



โดยรวมแล้ว DFS มีขั้นตอนการทำงานดังนี้

1. เริ่มจาก vertex ใด vertex หนึ่งเช่น vertex A ในภาพ

2. ทำการ visit vertex นี้

3. นำ vertex นี้เข้าสู่ stack

4. เก็บ vertex นี้ไว้ใน array (สำหรับการแสดงผล)

5. loop จนกว่าจะหมดข้อมูลใน stack

5.1. เริ่มจาก vertex v

5.2. ค้นหา vertex w ที่เชื่อมกับ v

5.3. ถ้า w ยังไม่มีการ visit ให้ทำการ visit พร้อมทั้งนำ vertex w เข้าสู่ stack พร้อมกับเก็บ w ไว้ ใน array (สำหรับการแสดงผล)

5.4. ถ้า w ถูก visit แล้วให้ดึง w ออกจาก stack

6. จบ loop

ตัวอย่างโปรแกรมของ Depth-First Search ของ undirected and unweighted graph ที่แสดงให้ดูต่อไปนี้มีการ trace การทำงานด้วยการแสดงข้อมูลที่อยู่ใน stack ณ เวลาต่าง ๆ กันของการท่องโหนดที่มีอยู่ในกราฟ

ตัวอย่างโค้ด : DFSVector.cpp

1. การท่องเข้าไปในกราฟแบบแนวกว้าง (Breadth -First Traversal: BFS)

Breadth-first search ใช้การเข้าหา โหนดที่เป็นเพื่อนบ้านของโหนดที่กำหนดไว้จนหมดเสียก่อน แล้วจึงทำการ visit โหนดอื่น ๆ ที่เหลืออยู่ด้วยวิธีเดียวกัน สมมติว่าเรากำหนดให้โหนดใด ๆ เป็น โหนดแรก เจากนั้นเริ่มต้นด้วยการ visit โหนดนี้และ mark โหนดนี้ว่าได้ผ่านการ visit แล้ว และเริ่มกระบวนการต่อไปนี้

กฎข้อที่ 1

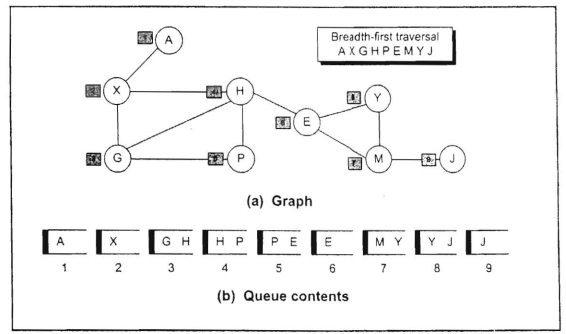
* visit โหนดที่ยังไม่ได้ visit ที่เชื่อมต่อกับโหนดนี้
* กำหนดให้โหนดนี้มีการ visit แล้ว
* นำโหนดนี้เข้าสู่ queue

กฎข้อที่ 2

* ถ้าไม่สามารถทำตามกฎข้อที่ 1 ได้ให้ดึงโหนดนี้ออกจาก queue และกำหนดให้โหนดนี้เป็น โหนด ในการทำงานต่อไป

กฎข้อที่ 3

* ถ้าทำตามกฎทั้งสองไม่ได้แล้วนั่นหมายความว่าได้ท่องในกราฟเสร็จสมบูรณ์แล้ว



จากรูป แสดงขั้นตอนการ visit ของกราฟที่เริ่มจากโหนด A เห็นว่าโหนดที่ visit คือ A X G H P E M Y J ซึ่งเป็นไปตามกฎของการท่องกราฟแบบ Breadth-First Search ที่ได้กล่าวถึง ถ้าสังเกตให้ดีจะเห็นว่า การ visit เกิดขึ้นกับโหนดที่อยู่ในระดับเดียวกันก่อนที่จะย้ายไป visit โหนดในระดับ อื่นๆ ต่อไป

ขั้นตอนที่ BFS ทำงานโดยรวมแล้วมีดังนี้

1. เริ่มจาก vertex ใด node หนึ่งเช่น vertex A ในภาพ

2. นำโหนดนี้เข้าสู่ queue

3. เก็บโหนดนี้ไว้ใน array (สำหรับการแสดงผล)

4. loop จนกว่าจะไม่มีข้อมูลอยู่ใน queue

4.1. ดึงเอา vertex v ออกจาก queue

4.2. กำหนดให้โนด v นี้ถูก visit แล้ว

4.3. loop จนกว่าเพื่อนบ้านของโหนด v นี้ได้รับการ visit จนหมด

4.3.1. ทำการ visit โหนด w ที่เป็นเพื่อนบ้านของ v

4.3.2. นำโหนดเข้าสู่ queue

4.3.3. เก็บ w ไว้ใน array (สำหรับการแสดงผล)

4.4. จบ loop

5. จบ loop

ตัวอย่างการเขียน Code ของ BFS ในกราฟ directed and unweighted graph

ตัวอย่างโค้ด :BFSVector.cpp

**แบบฝึกหัด**

1. ในหนังสือพิมพ์หรือนิตยสารมักมีเกมปริศนาที่เรียกว่า word tramsformation โดยจะกำหนดรายการของคำทั้งหมดมา ตัวอย่างเช่น

dip

lip

mad

map

maple

may

pad

pip

pod

pop

sap

sip

slice

slick

spice

stick

stock

จากนั้นกำหนดคำเริ่มต้นที่และคำที่ต้องการจะให้คำเริ่มต้นเปลี่ยนไป ให้หาว่าจากคำเริ่มต้นที่กำหนดมาให้นั้น ต้องเปลี่ยนกี่ครั้งถึงจะคำที่ต้องการสุดท้าย โดยมีเงื่อนไขว่า ในการเปลี่ยนแต่ละครั้งสามารถเปลี่ยนตัวอักษรได้ทีละตัวเท่านั้นและความยาวยังคงเท่าเดิมตลอด

ให้เขียนโปรแกรมเพื่อช่วยคำนวนจำนวนครั้งที่น้อยที่สุดในการ เปลี่ยนคำไปเป็นคำที่ต้องการ

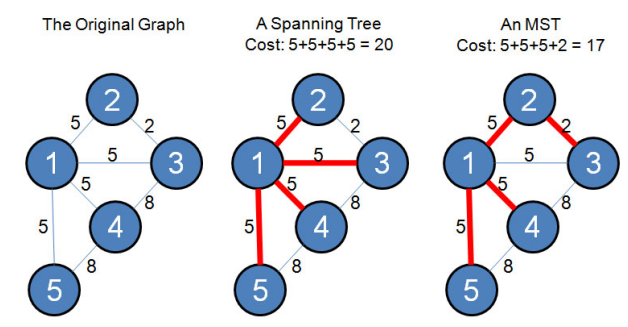
ตัวอย่าง

กำหนดคำเริ่มต้นและคำสุดท้ายคือ spice และstock

spice -- > slice -- > slick -- > stick -- >stock

Minimum Spanning Tree (MST)

หากมีกราฟไม่มีทิศทางและเส้นเชื่อมมีค่าน้ำหนักกํากับอยู่ ถ้าต้องการหาต้นไม้ในกราฟที่เชื่อมโหนดทุกโหนดเข้าด้วยกันได้ และมีผลรวมของค่าน้ำหนักของเส้นเชื่อมในต้นไม้น้อยที่สุด ต้นไม้ที่เชื่อมโหนดทุกโหนดเข้าด้วยกันได้เรียกว่า Spanning Tree ส่วน Spanning Tree ที่มีผลรวมค่าน้ำหนักน้อยที่สุดคือ Minimum Spanning Tree มักถูกใช้ในการหาวิธีวางท่อหรือสายเคเบิลที่เชื่อมจุดสําคัญเพื่อให้เสียค่าใช้จ่ายน้อยที่สุด



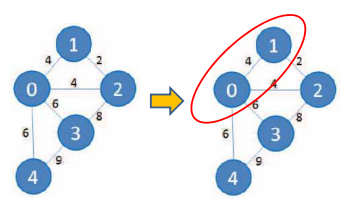
รูปที่ ตัวอย่าง Minimum Spanning Tree

ปัญหา Minimum Spanning Tree สามารถแก้ไขได้ด้วย หลายอัลกอริทึก อย่างเช่น Prim and Kruskal algorithm

Prim’s Algorithm

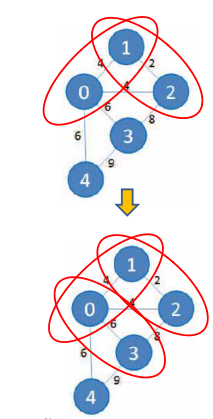
แนวคิดพื้นฐาน คือ ในเมื่อต้องเชื่อมทุกโหนด ดังนั้นเริ่มคิดจากโหนดไหนก็ได้ จากโหนดเดียวให้ขยายไปโหนดที่ติดกันด้วยเส้นเชื่อมที่มีน้ำหนักน้อยที่สุด (ถ้ามีหลายเส้นที่เบาที่สุดเท่ากันให้ เลือกเส้นไหนก่อนก็ได้)

ตัวอย่าง สมมติว่าเริ่มที่โหนด 0 มีเส้นเชื่อมสองเส้นไปยังโหนด 1 และโหนด 4 แต่เส้นเชื่อมไปยังโหนด 1 มีน้ำหนักน้อยกว่า ดังนั้นจึงเลือกเส้นเชื่อมไปโหนด 1

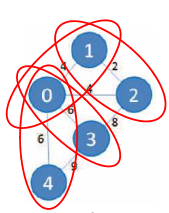


มองโหนดที่เชื่อมถึงกันไปแล้วเป็น Super Node (รวมโหนดเข้าด้วยกัน) จากนั้นทําแบบเดิมอีก คราวนี้เลือก เส้นเชื่อมไปโหนด 2 ที่มีน้ำหนักเป็น 2 เพราะมีน้ำหนักน้อยที่สุด จากตัวเลือกที่มีอยู่

ตอนนี้ Super Node ของเรามีโหนดย่อยสามโหนดอยู่ด้วยกัน คือโหนด 0, 1 และ 2 เส้นเชื่อมที่อยู่ภายใน Super Node นี้ไม่ถือว่าเป็นเส้นเชื่อมไป ข้างนอกหาเพื่อนบ้าน ดังนั้นไม่นับ (เช่นเส้นเชื่อมระหว่างโหนด 0 กับโหนด 2 เป็นเส้นเชื่อมภายใน Super Node ไม่นับ)



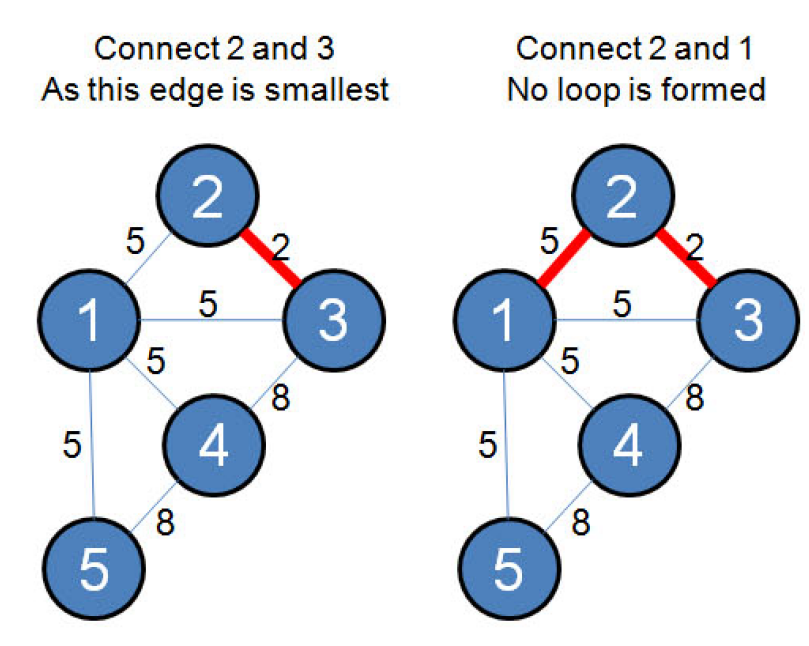
เมื่อรวมโหนดทุกโหนดมาหมดแล้ว ก็เป็นอันจบการทํางาน รวมค่าน้ําหนักทั้งหมดได้ 4 + 2 + 6 + 6 = 18



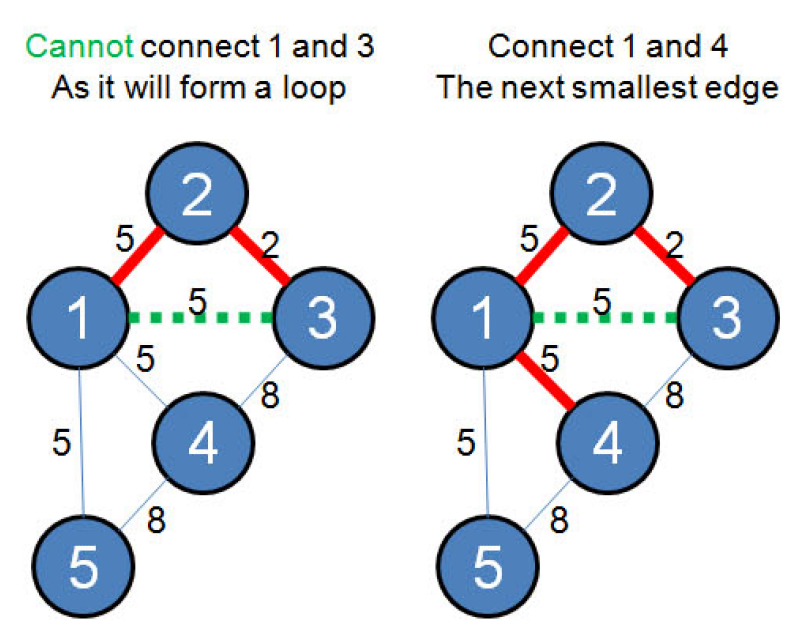
Kruskal’s Algorithm

แนวคิดพื้นฐาน คือ ในเมื่อต้องเชื่อมทุกโหนด ดังนั้นเริ่มคิดจากเส้นเชื่อมไหนก่อนก็ได้ที่มีค่าที่น้อยที่สุด โดยไม่จำเป็นต้องมองเส้นที่เชื่อมต่อกันแล้วเป็น super node ในการเลือกเส้นเชื่อมที่ต่อกันจะต้องไม่ทำให้เกิด cycle

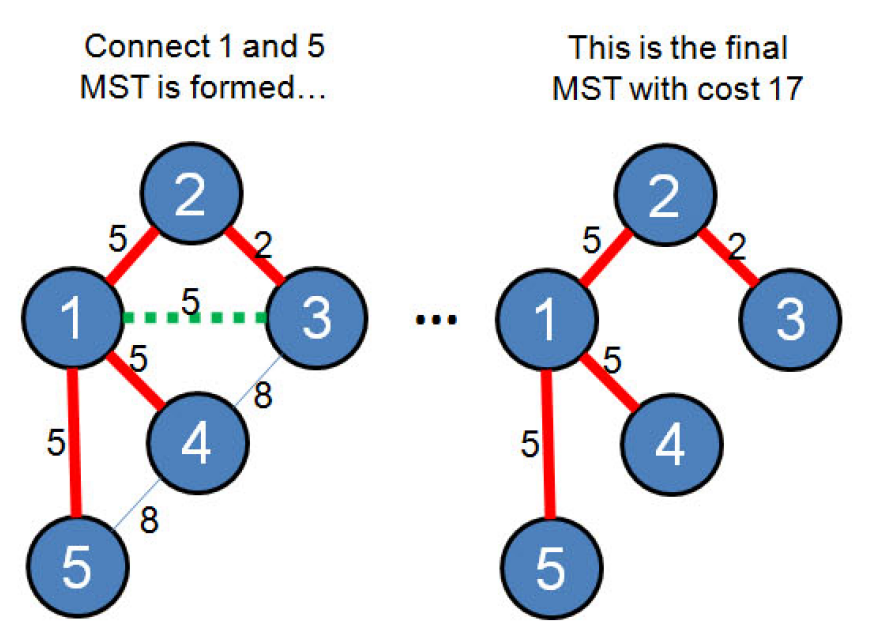
ตัวอย่าง สมมติมีกราฟดังรูป เส้นเชื่อมที่มีน้ำหนักน้อยที่สุดคือ เส้นเชื่อมระหว่างโหนด 2 และ 3 จากนั้นพิจารณาเส้นเชื่อมที่มีน้ำหนักถัดมา คือ เส้นเชื่อมที่มีน้ำหนักเป็น 5 ซึ่งมีหลายเส้น ให้เลือกเส้นไหนก็ได้ จากรูป เลือกเส้นที่เชื่อมระหว่างโหนด 2 และ 1



เส้นเชื่อมที่มีน้ำหนักถัดมายังคงเป็น 5 จากรูปเห็นว่า ถ้าเลือกเส้นเชื่อมระหว่างโหนด 1 และ 3 จะทำให้เกิด cyclic ดังนั้นไม่ควรเลือกเส้นนี้



เมื่อโหนดทุกโหนดถูก visit แล้ว ก็เป็นอันจบการทํางาน รวมค่าน้ําหนักทั้งหมดได้ 2 + 5 + 5 + 5 = 17



**Pseudo Code:**

// Initialize result

mst\_weight = 0

// Create V single item sets

**for** **each** vertex v

parent[v] = v;

rank[v] = 0;

**Sort** all edges into non decreasing

order by weight w

**for** each (u, v) taken from the sorted list E

**do if** **FIND-SET**(u) != **FIND-SET**(v)

**print** edge(u, v)

mst\_weight += weight of edge(u, v)

**UNION**(u, v)

ขั้นตอนการหา MST ด้วย Kruskal’s algorithm

1. เรียงลำดับเส้นเชื่อมจากน้ำหนักน้อยไปน้ำหนักมาก

2. เริ่มจากเส้นที่มีน้ำหนักน้อยที่สุด โดยพิจารณาดูว่าถ้าเลือกเส้นเชื่อมนี้ จะต้องไม่ทำให้กราฟเกิด cycle

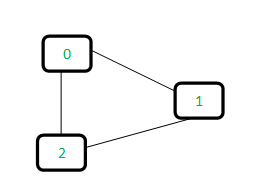
3. ทำซ้ำข้อที่ 2 จนกระทั่งครบทุกเส้นเชื่อม จะได้จำนวนเส้นเชื่อมทั้งหมดเท่ากับ V-1 เส้น

ในข้อที่ 2 สามารถใช้ union-find algorithm เพื่อพิจารณาว่าเส้นที่เลือกจะทำให้มี cycle หรือไม่

* Find ใช้กำหนดเพื่อดูว่า แต่ละอิลิเม้นต์อยู่ใน subset เดียวกันหรือไม่
* Union รวมสอง subset ให้เป็น subset เดียวกัน

วิธีนี้สมมติว่ากราฟไม่มีลูปในตัวเอง จากนั้นจะ track subsets ในรูปของอาเรย์ 1 มิติ กำหนดให้แทนด้วย parent[]

กำหนดให้มีกราฟดังรูปด้านล่างนี้

  
แต่ละเส้นเชื่อม ให้สร้าง subset ของโหนดทั้งสอง ถ้าโหนดทั้งสองอยู่ใน subset เดียวกันแสดงว่ามีลูป

* เริ่มต้นกำหนดให้ทุกช่องใน parent มีค่าเป็น -1 หมายถึงว่าในโหนดทั้งหมดอยู่ใน subset ที่แตกต่างกัน
* Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

0 1 2

-1 -1 -1

* เริ่มพิจารณาแต่ละเส้นเชื่อม
* *Edge 0-1:* หา subset ของโหนด 0 และ 1 พบว่าทั้งสองโหนดอยู่คนละ subset กัน จากนั้นทำการรวมสองโหนดนี้เข้าด้วยกัน ไม่ว่าจะให้โหนด 0 เป็นโหนดแม่ของโหนด 1 หรือ ให้โหนด 1 เป็นโหนดแม่ของโหนด 0 ก็ได้ สมมติให้ โหนด 1 เป็นโหนดแม่ของโหนด 0

0 1 2 <----- 1 is made parent of 0 (1 is now representative of subset {0, 1})

1 -1 -1

* *Edge 1-2:* โหนด 1 และ โหนด 2 อยู่คนละ subset กัน ดังนั้นจะรวมสองโหนดนี้ให้อยู่ subset เดียวกัน โดยกำหนดให้สองเป็นโหนดแม่ของโหนด 1

0 1 2 <----- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})

1 2 -1

* *Edge 0-2:* พิจารณาโหนด 0 พบว่า โหนดรากของโหนด 0 จริงๆ คือ โหนด 2 ถ้าพิจาณาโหนด 2 ค่าที่อยู่ใน parent เป็น -1 หมายถึงว่าโหนดสองมีโหนดรากเป็นตัวมันเอง ดังนั้นจะได้ว่า โหนด 0 และ 2 อยู่ใน subset เดียวกัน สรุปได้ว่า กราฟนี้มี cycle

Note: How subset of 0 is same as 2?  
0->1->2 // 1 is parent of 0 and 2 is parent of 1

วิธีหนึ่งที่สามารถใช้ใน union-find algorithm คือ Path Compression แนวคิด คือ สร้าง tree เพื่อหาโหนดรากของโหนดที่กำลังพิจารณาเพื่อว่าไม่จำเป็นต้องท่องยัง intermediate อีกครั้ง ถ้า X เป็นโหนดรากของ subtree แล้วเส้นทางจากทุกโหนดภายใต้ X จะถูกลดไปด้วย

ตัวอย่าง

กำหนดให้เซต {0, 1, .. 9} มีลักษณะดังรูปด้านล่าง

9

/ | \

4 5 6

/ \ / \

0 3 7 8

/ \

1 2

เมื่อเรียกใช้find() สำหรับโหนด 3 จะต้องท่องไปเพื่อหา โหนดรากของ 3 คือ 9 ด้วยการลดเส้นทาง สามารถที่จะสร้าง โหนด 3 ให้เป็นโหนดลูกของโหนด 9 ได้ เพื่อว่าครั้งถัดไป ในการหาโหนดรากของ โหนด 1 โหนด 2 หรือ โหนด 3 เส้นทางในการหาจะได้ลดลง

9

/ / \ \

4 5 6 3

/ / \ / \

0 7 8 1 2

ตัวอย่าง การหา MST ด้วย Kruskal’s Algorithm สมมติมีกราฟดังรูป ซึ่งมี 9 โหนดและ 14 เส้นเชื่อม ถ้าจะหา MST จะต้องได้เส้นเชื่อมทั้งหมดเท่ากับ V-1 คือ 8 เส้นเชื่อม



จากขั้นตอนวิธีของ Kruskal’s Algorithm

1. เรียงลำดับเส้นเชื่อมทั้งหมดจากน้ำหนักน้อยไปมาก

After sorting:

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

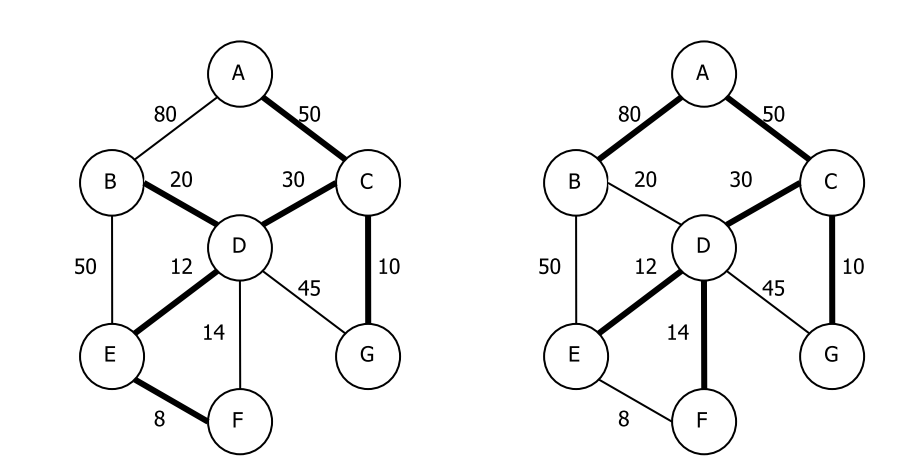
2. พิจารณาที่ละเส้นเชื่อมที่ได้เรียงลำดับไว้แล้ว

|  |  |
| --- | --- |
| เส้นเชื่อม | กราฟ |
| **1.** Pick edge 7-6: No cycle is formed, include it. | https://www.geeksforgeeks.org/wp-content/uploads/Fig-1.jpg |
| **2.** Pick edge 8-2: No cycle is formed, include it. | https://www.geeksforgeeks.org/wp-content/uploads/Fig-2-241x300.jpg |
| **3.** Pick edge 6-5: No cycle is formed, include it. | https://www.geeksforgeeks.org/wp-content/uploads/Fig-3-300x232.jpg |
| **4.** Pick edge 0-1: No cycle is formed, include it. | https://www.geeksforgeeks.org/wp-content/uploads/Fig-4-300x175.jpg |
| **5.** Pick edge 2-5: No cycle is formed, include it. | https://www.geeksforgeeks.org/wp-content/uploads/Fig-5-300x175.jpg |
| **6.** Pick edge 8-6: Since including this edge results in cycle, discard it. |  |
| **7.** Pick edge 2-3: No cycle is formed, include it | https://www.geeksforgeeks.org/wp-content/uploads/Fig-6-300x175.jpg |
| **8.** Pick edge 7-8: Since including this edge results in cycle, discard it. |  |
| **9.** Pick edge 0-7: No cycle is formed, include it. | https://www.geeksforgeeks.org/wp-content/uploads/Fig-7-300x175.jpg |
| **10.** Pick edge 1-2: Since including this edge results in cycle, discard it. |  |
| **11.** Pick edge 3-4: No cycle is formed, include it. | https://www.geeksforgeeks.org/wp-content/uploads/fig8new.jpeg |

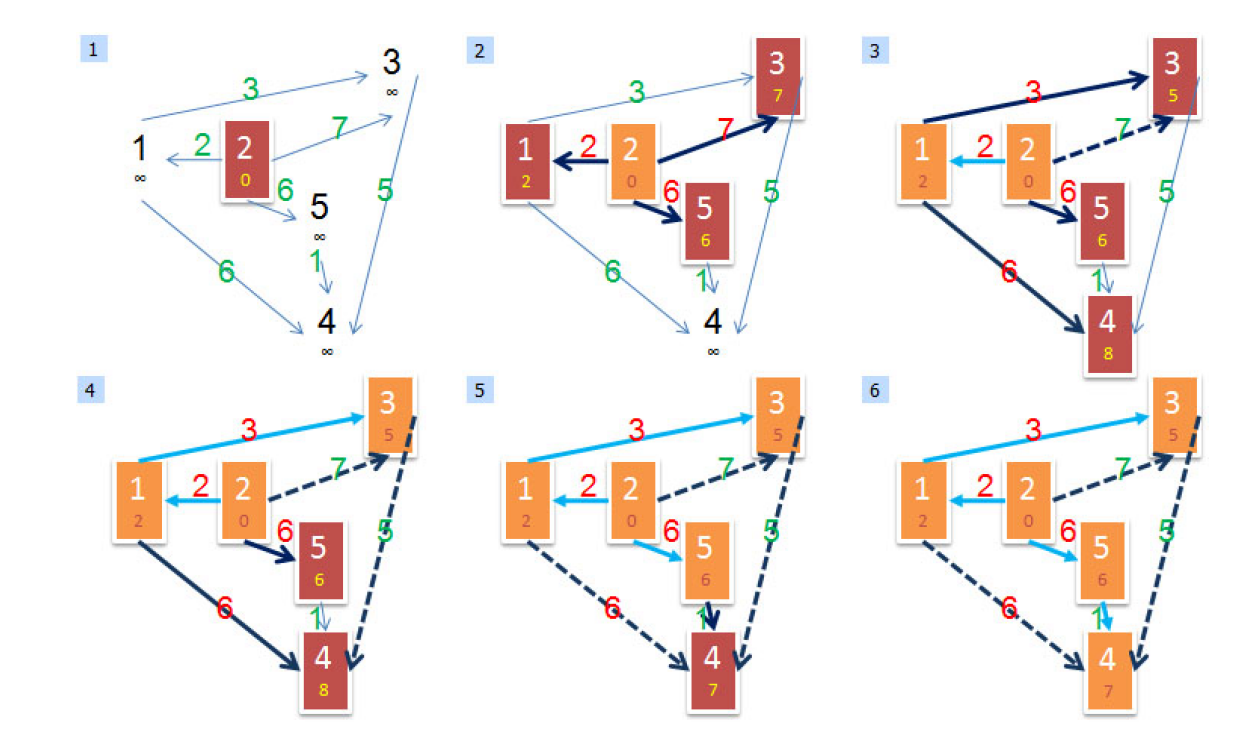
การหาระยะทางที่สั้นที่สุด (Shortest path) กับ Dijkstra's algorithm

minimum spanning tree กับ shortest path ต่างกัน หรือเหมือนกันอย่างไร

* minimum spanning tree เป็นการค้นหาระยะทางที่สั้นที่สุดใน graph ที่มีเพียงเส้นทางเดียวเท่านั้น (ภาพด้านซ้าย)
* ส่วน shortest path เป็นการหาระยะทางที่สั้นที่สุดใน graph จาก node ใด node หนึ่งใน graph ไปยัง node อื่น ๆ ที่มีอยู่ ซึ่งอาจมีมากกว่าหนึ่งเส้นทาง (ภาพด้านขวา)



มี algorithm หลายตัว ที่สามารถนำมาใช้ในการหาระยะทางที่สั้นที่สุด แต่วิธีที่จะแนะนำต่อไปนี้ เป็นวิธีที่เรียกว่า Dijkstra Algorithm เรามาลองดูขั้นตอนของการหาระยะทางที่สั้นที่สุด ด้วย algorithm นี้



* จากจุดเริ่มต้น (ในตัวอย่างตามรูปด้านบน คือโหนด 2) ณ จุดเริ่มต้นจะถือว่าขณะนี้มีโหนดในกลุ่มเพียงโหนดเดียวและมีค่าน้ำหนักในการไปถึงโหนดตัวเองเท่ากับศูนย์โหนดอื่น ๆ ที่ยังไม่ถูกแวะเยี่ยมถือว่ามีระยะทางจากโหนดเริ่มต้นเป็นอนันต์ (ค่านี้จะถูกเปลี่ยนเป็นค่าจริงเมื่อโหนดถูกแวะเยี่ยม)
* จากโหนดเริ่มต้นให้สํารวจเส้นเชื่อมทั้งหมดแล้วเก็บไว้ในลิสต์ จากนั้นมองหาเส้นเชื่อมที่มีค่าน้ําหนักน้อยที่สุดในลิสต์ (ได้เส้นเชื่อมที่ต่อไปโหนด 1)
* เลือกโหนดที่ต่อกับเส้นเชื่อมนั้นเข้ามาในกลุ่ม (คือโหนดสีส้มทั้งหมด) ให้ถือว่าโหนดพวกนี้ถูกแวะเยี่ยมและได้ค่าน้ําหนักรวมเท่ากับค่าน้ําหนักจากโหนดที่มาถึงมันบวกกับเส้นเชื่อมที่ใช้ ดังนั้นค่าน้ําหนักของโหนด 1 จึงเป็น 0 + 2 = 2
* เติมเส้นเชื่อมของโหนด 1 เข้าไปในลิสต์ (เส้นเชื่อมที่มีสีฟ้าคืออยู่ในลิสต์)
* เลือกเส้นเชื่อมที่สร้างค่าน้ําหนักรวมน้อยที่สุด และต้องเชื่อมไปยังโหนดนอกกลุ่ม
* เลือกโหนดและขยายลิสต์ของเส้นเชื่อมต่อไปเรื่อย ๆ ในทํานองเดิม จนกว่าจะไม่สามารถเพิ่มโหนดใหม่เข้ามาในกลุ่มได้อีก (ปรกติจะทดสอบจนลิสต์เส้นเชื่อมว่าง)

ขั้นตอนของ Dijkstra's algorithm

1. กำหนดให้ระยะทาง (dist) จากโหนดเริ่มต้นไปยังทุกจุดเป็น infinite

2. สร้าง set สำหรับเก็บโหนดที่กำลังจะถูก process โดยที่แต่ละ item ของ set ประกอบไปด้วย weight และ vertex

3. กำหนดให้ระยะทางของจุดเริ่มต้นเป็น 0 และ เพิ่มโหนดเริ่มต้นใน set

4. loop จนกว่า set จะไม่มี item

4.1 เอา item แรกออกจาก set และให้โหนดที่อยู่ใน item นี้แทนด้วย u

4.2 loop จนกว่า adjacent node (v) ของ u จะหมด

4.2.1 ถ้า dist[v] > dist[u] + weight(u,v)

4.2.1.1 update ระยะทางของ v : dist[v] = dist[u] + weight(u, v)

4.2.1.2 เพิ่ม v ใน set

4.2.1.3 ถ้าระยะทางของ v ไม่ได้เป็น infinite

i. ลบ v ออกจาก set

5. พิมพ์ระยะทางของทุกโหนดจาก dist

เอกสารอ้างอิง

1. steven halim, competitive programming 3rd edition, 2013

2. A Computer science for Geeks [Available online] https://www.geeksforgeeks.org/

3. Graph Link Implementation [Available online] https://www.cs.bu.edu/teaching/c/graph/linked/