Graph Algorithms II

Agenda

01 **Shortest path**

คืออะไร, Algorithm

02 **All pair shortest path**

คืออะไร, Algorithm

03 **Topological sort**

คืออะไร, Algorithm

04 **โจทย์**

ตัวอย่างข้อสอบกราฟ

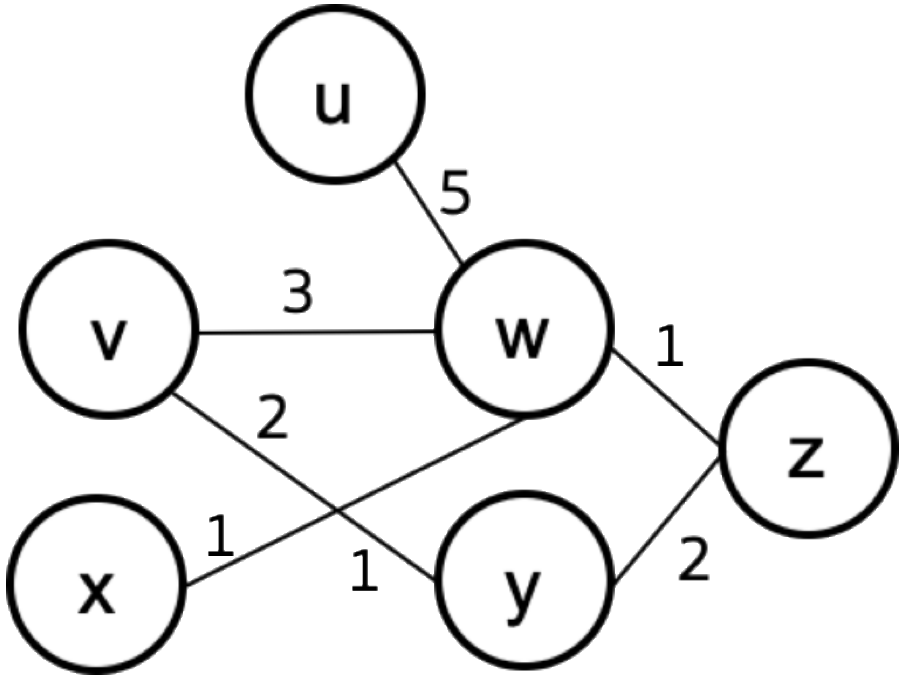
Shortest path

คืออะไร, Algorithm

Shortest path of Graph

คืออะไร, Algorithm

- **มีจุดเริ่มต้น (source) 1 จุด**
- ถามว่า **เส้นทางใด** ใกล้ที่สุด / ใช้ระยะทางน้อยสุด / weight ต่ำที่สุด ในการ**ไปถึง destination**

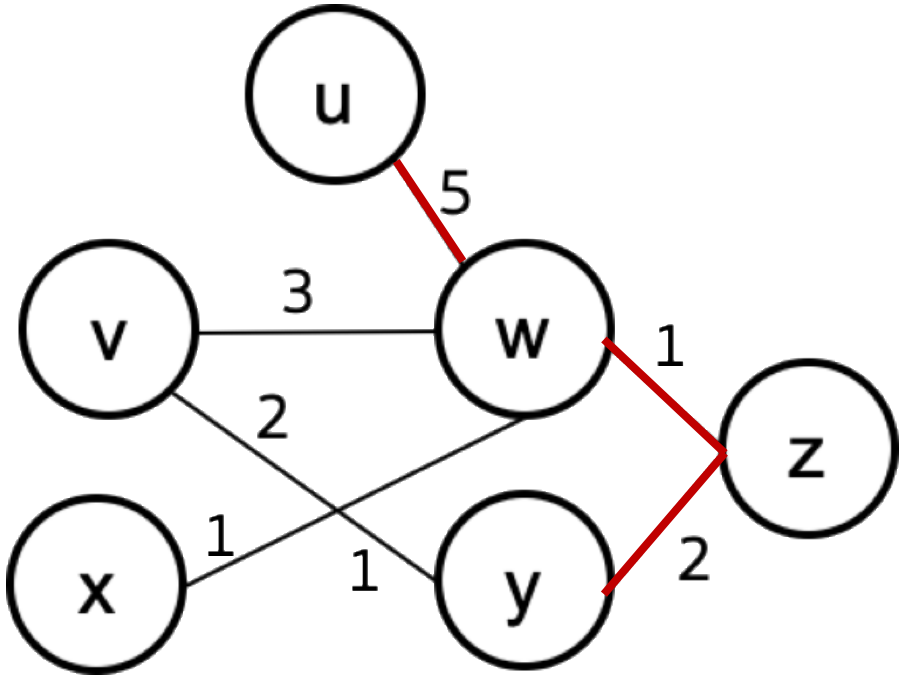


จงหาเส้นทางที่ใกล้ที่สุด จาก u ไป y

Shortest path of Graph

คืออะไร, Algorithm

- **มีจุดเริ่มต้น (source) 1 จุด**
- ถามว่า **เส้นทางใด** ใกล้ที่สุด / ใช้ระยะทางน้อยสุด / weight ต่ำที่สุด ในการ**ไปถึง destination**

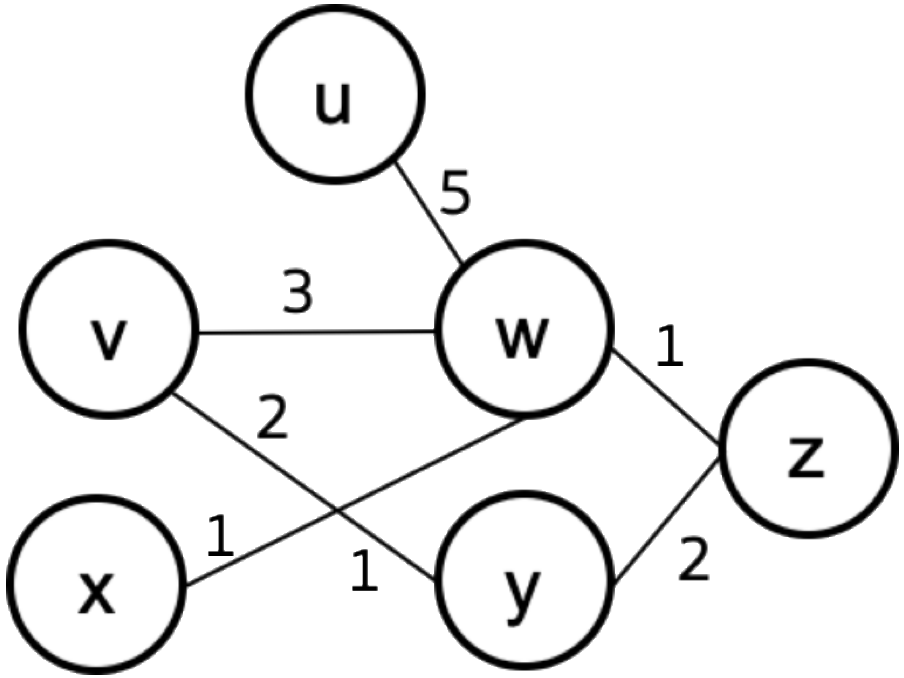


จงหาเส้นทางที่ใกล้ที่สุด จาก u ไป y

Shortest path of Graph

คืออะไร, Algorithm

- <u>หรือ</u> ถามว่า **ระยะทางที่ใกล้ที่สุด** จากจุดเริ่มต้น **ไปยังทุกจุด** เป็นเท่าใด **(สร้าง Shortest Path Tree)**



จงหาระยะทางที่สั้นที่สุด
ในการเดินไปให้ครบทุก node เมื่อเริ่มจาก u

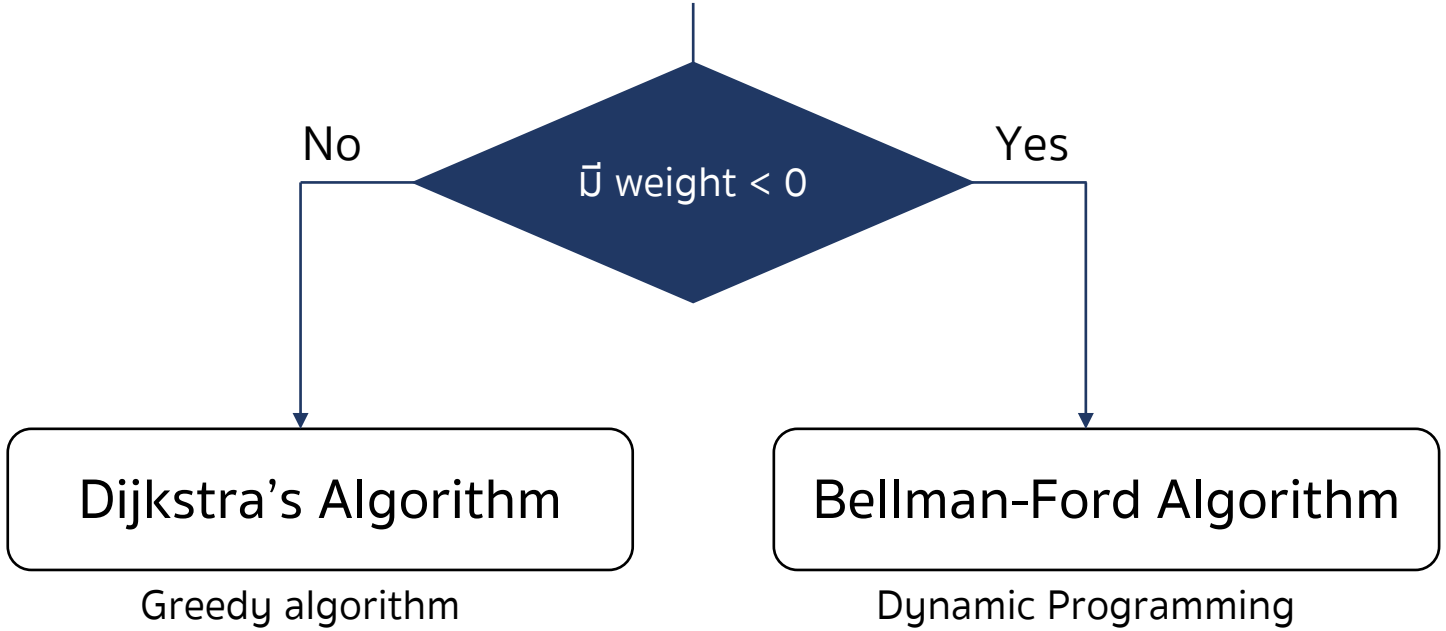
u -> v -
u -> w -
u -> x -
u -> y -
u -> z -

(ผลลัพธ์จากการสร้าง **Shortest Path Tree**)

Shortest path of Graph

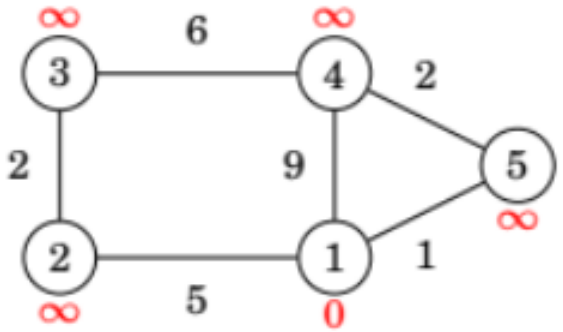
คืออะไร, Algorithm

- วิธีการหา Shortest Path / Shortest Path tree

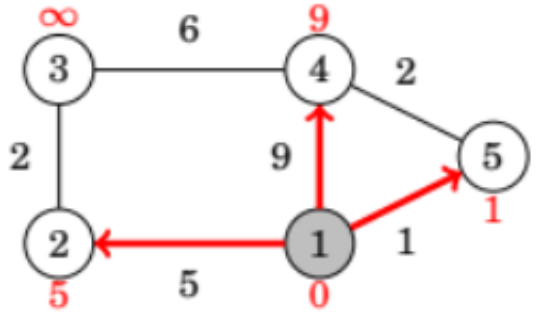


Shortest path of Graph

Dijkstra’s Algorithm



ในแต่ละรอบ Dijkstra's algorithm จะเลือกโหนดที่ยังไม่ถูกพิจารณาและมีระยะทางใกล้ที่สุด ในตัวอย่างโหนดแรกเป็นโหนด 1 ซึ่งมีระยะทางเป็น 0

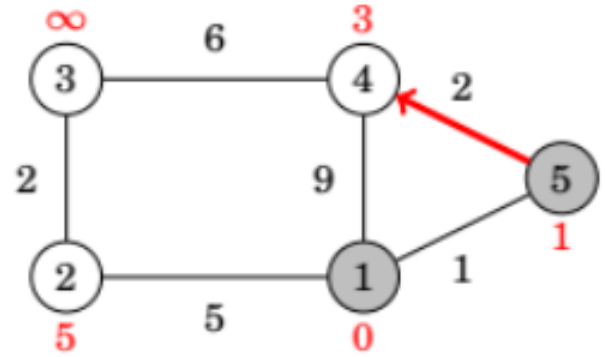


เมื่อโหนดถูกเลือก algorithm จะพิจารณาทุกเส้นเชื่อมที่เริ่มต้นที่โหนดนั้นและลดระยะทาง

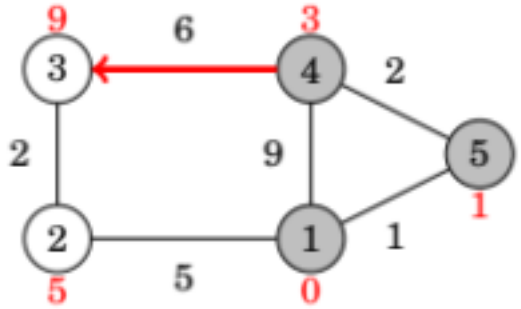
ในกรณีนี้เส้นเชื่อมจากโหนด 1 ลดระยะทางของโหนด 2, 4 และ 5 ซึ่งระยะทางปัจจุบันเป็น 5, 9 และ 1

Shortest path of Graph

Dijkstra's Algorithm



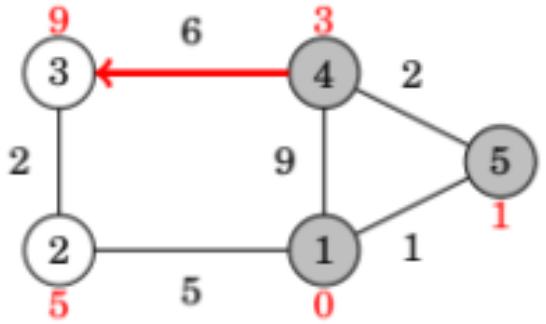
โหนดต่อไปที่จะประมวลผลคือโหนด 5 ที่มีระยะทางเป็น 1 ซึ่งจะไปลดระยะทางที่ไปยังโหนด 4 จาก 9 เป็น 3



หลังจากนั้นโหนดต่อไปคือโหนด 4 จะลดระยะทางจาก 3 เป็น 9

Shortest path of Graph

Dijkstra’s Algorithm

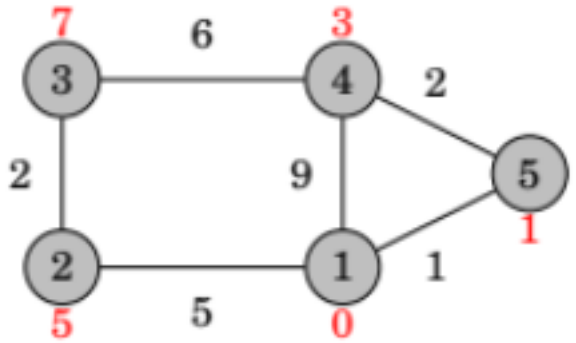


คุณสมบัติของ Dijkstra's algorithm คือเมื่อโหนดถูกเลือกแล้ว ระยะทางของมันจะถือว่าสิ้นสุดแล้ว

ตัวอย่างเช่นที่จุดนี้ ระยะทาง 0 1 และ 3 เป็นระยะทางสุดท้ายของโหนด 1, 5 และ 4

Shortest path of Graph

Dijkstra's Algorithm



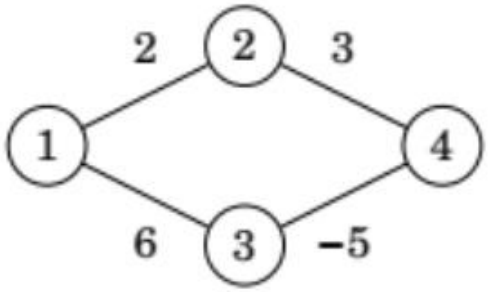
หลังจากนั้น algorithm จะทำงานกับ 2 โหนดที่เหลือ ซึ่งได้ระยะทางสุดท้ายดังรูป

Shortest path of Graph

Dijkstra’s Algorithm

ประสิทธิภาพของ Dijkstra's algorithm นั้นอยู่บนพื้นฐานของความจริง ที่ว่าในกราฟไม่มีเส้นเชื่อมที่มีน้ำหนักเป็นลบ (negative edges)

ถ้ามีเส้นเชื่อมเป็นลบ algorithm อาจจะให้ผลลัพธ์ที่ผิดได้ ตัวอย่างเช่น



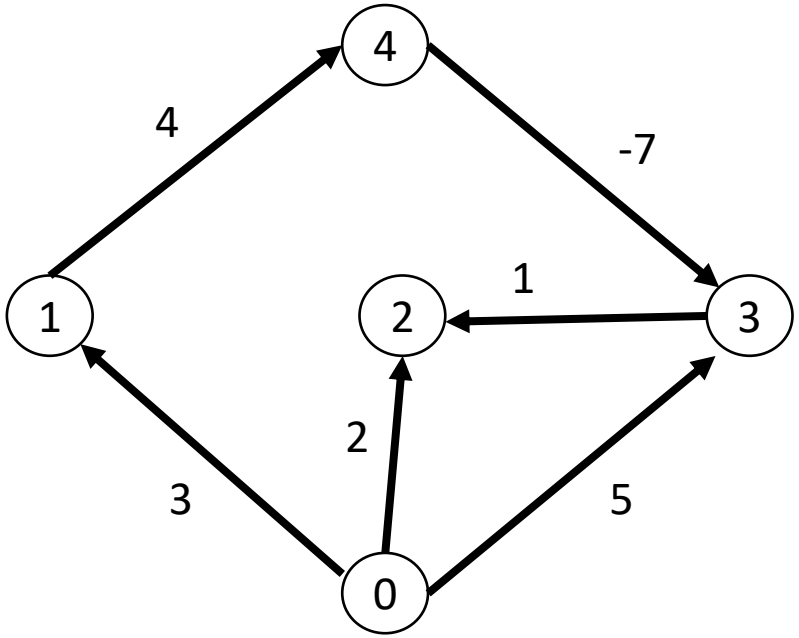
ระยะทางสั้นที่สุดจาก 1 ไปโหนด 4 คือ 1->3->4 ซึ่งความยาวเป็น 1 อย่างไรก็ตาม Dijkstra's algorithm หาเส้นทาง 1->2->4 ตามที่ค่าน้ าหนักน้อยที่สุดของเส้นเชื่อม

Shortest path of Graph

Dijkstra’s Algorithm

ประสิทธิภาพของ Dijkstra's algorithm นั้นอยู่บนพื้นฐานของความจริง
ที่ว่าในกราฟไม่มีเส้นเชื่อมที่มีน้ำหนักเป็นลบ (negative edges)

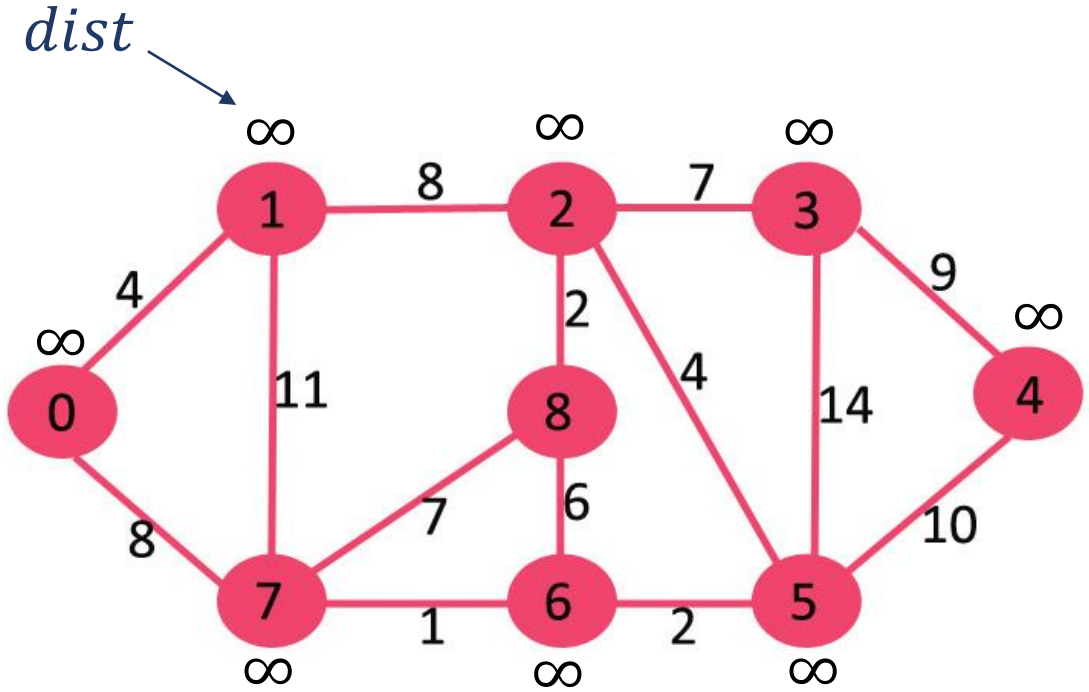
ถ้ามีเส้นเชื่อมเป็นลบ algorithm อาจจะให้ผลลัพธ์ที่ผิดได้ ตัวอย่างเช่น



Shortest path of Graph

Dijkstra's Algorithm

- วิธีการสร้าง Shortest Path Tree **(Dijkstra's Algorithm)**
  - Distance ที่สั้นที่สุด จาก src ถึงแต่ละ node
  - ใช้ **BFS** with **Priority queue**



- Set up dist ทุกตัวเป็น INF
- เริ่มต้นจาก src -> s = src
- dist[s] = 0

- ถ้ายัง visited ไม่ครบทุกตัว
  เลือก **s** ที่มี **dist** น้อยที่สุด
  mark ว่า visit **s** แล้ว
  For each **node** in Adj(s)
    ถ้า **dist[s] + w(s –> node) < dist[node]:**
      update **dist[node]**

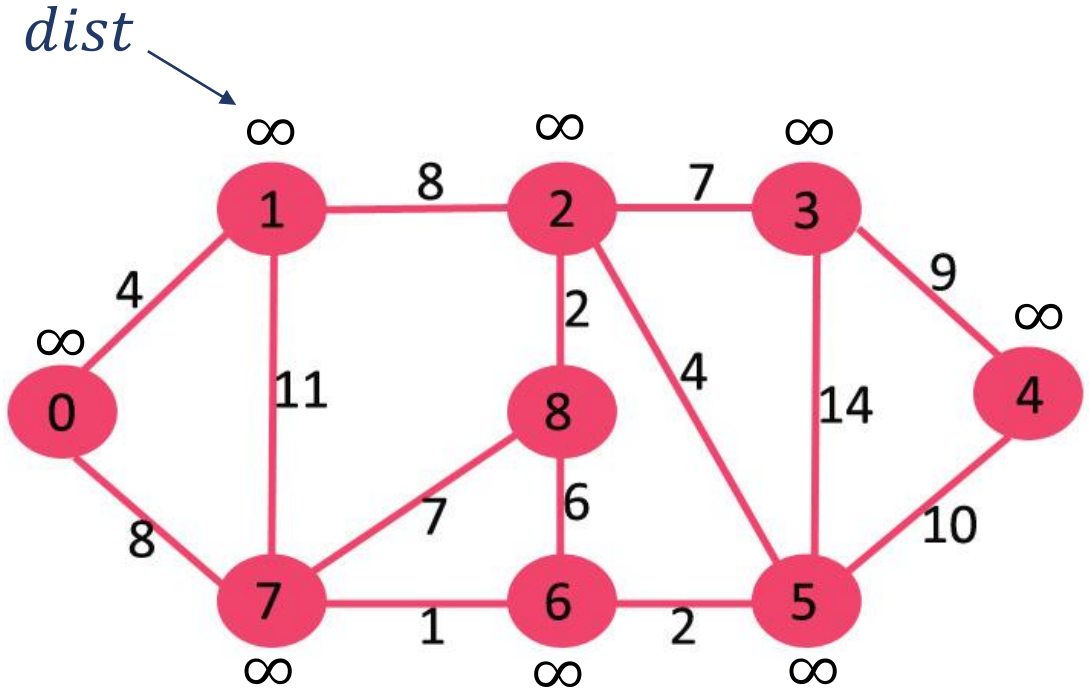
- ค่า **dist** ที่ได้ของแต่ละ node คือ shortest distance from src

Shortest path of Graph

Dijkstra's Algorithm

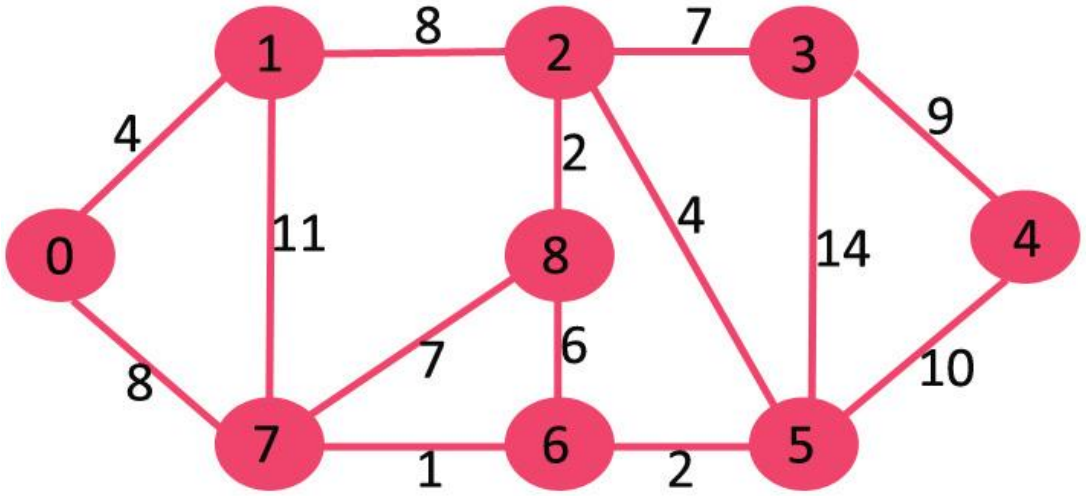
- วิธีการสร้าง Shortest Path Tree **(Dijkstra's Algorithm)**
  - Distance ที่สั้นที่สุด จาก src ถึงแต่ละ node
  - ใช้ **BFS** with **Priority queue**

- **เป็น BFS ยังไง?**
  - ค่อยๆ อัพเดท dist ทีละขั้นตามแนวกว้าง
  - ใช้ Priority queue เพื่อเก็บ node ที่ต้องไปต่อ แบบเรียงลำดับ dist จากน้อยไปมาก



Shortest path of Graph

Dijkstra's Algorithm



Let's code !!

Shortest path of Graph

Dijkstra's Algorithm

- **แบบฝึกหัด**
  - จงหาเส้นทางที่ใกล้ที่สุดในการเดินทางจากสถานที่เริ่มต้น ไปยังปลายทางที่กำหนด โดยมีกราฟเป็น input

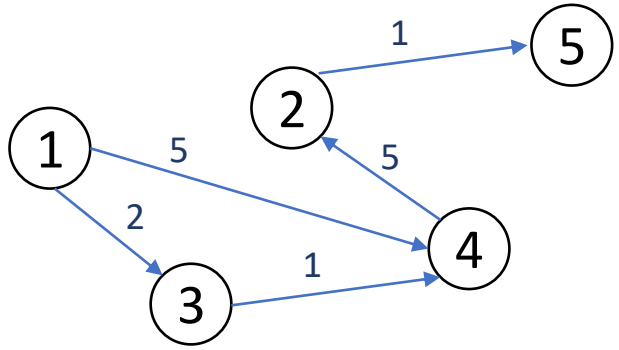
- **ข้อมูลนำเข้า (Input)**
  - จำนวนสถานที่ในแผนที่ (N)
  - จำนวนเส้นทางที่เชื่อมต่อแต่ละสถานที่ (E)
  - สถานที่ต้นทาง สถานที่ปลายทาง
  - ต้นทาง ปลายทาง ระยะทาง ของ เส้นทางที่ 1
  - ต้นทาง ปลายทาง ระยะทาง ของ เส้นทางที่ 2
  - ...
  - ต้นทาง ปลายทาง ระยะทาง ของ เส้นทางที่ E

- **ผลลัพธ์ (Output)**
  - v_1 v_2 v_3 ... v_k

  (vi แทนหมายเลขสถานที่ โดยคั่นด้วยเว้นวรรค)
  - ระยะทางที่ใช้

- **ตัวอย่าง input**

```
4
4
1 2
1 4 5
1 3 2
3 4 6
4 2 5
```



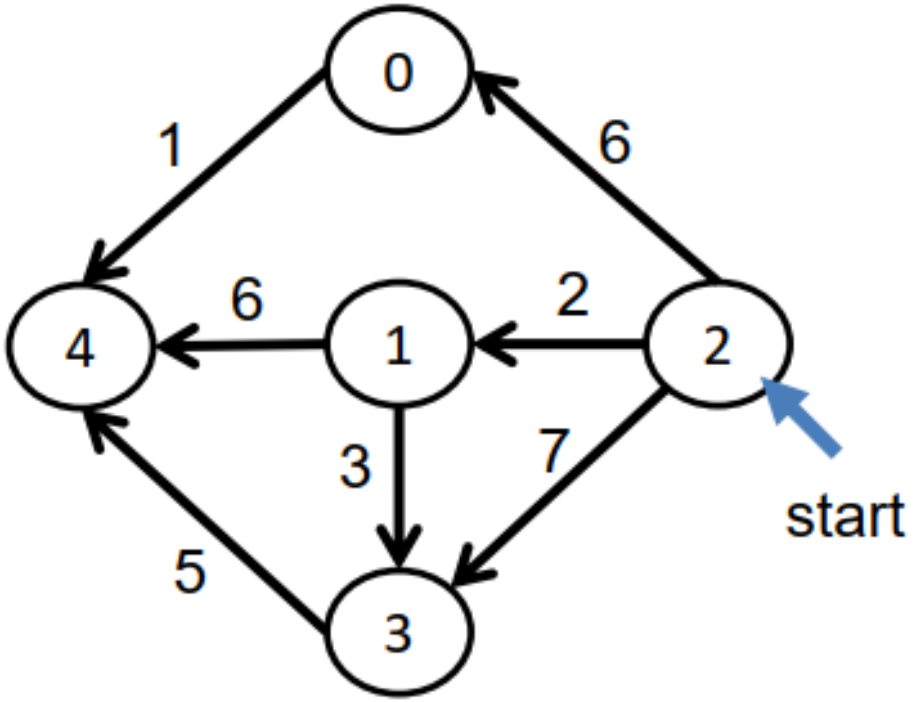
- **ตัวอย่าง output**

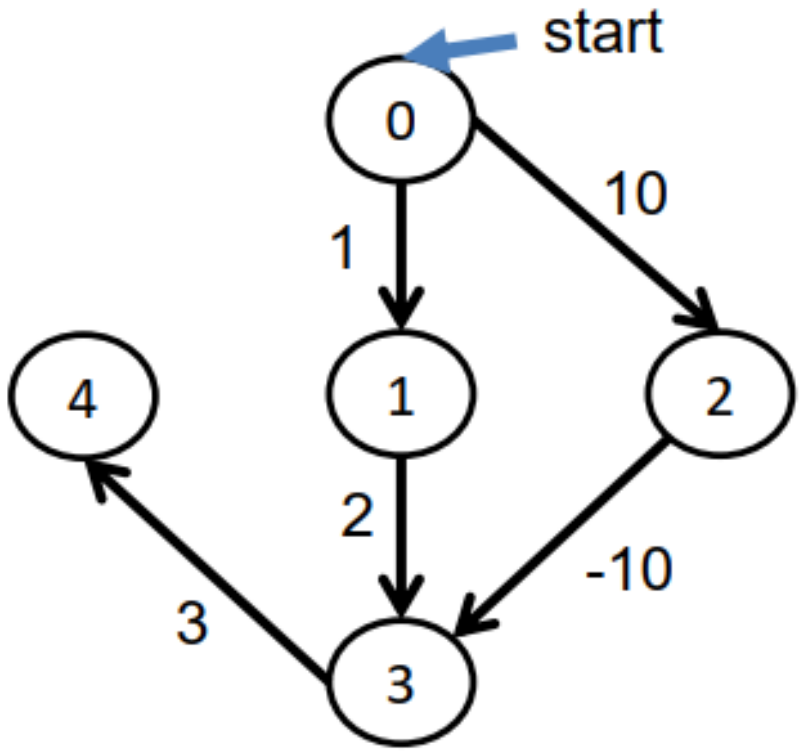
```
1 3 4 2
8
```

Shortest path of Graph

Dijkstra's Algorithm

ทดลองกราฟนี้





Shortest path of Graph

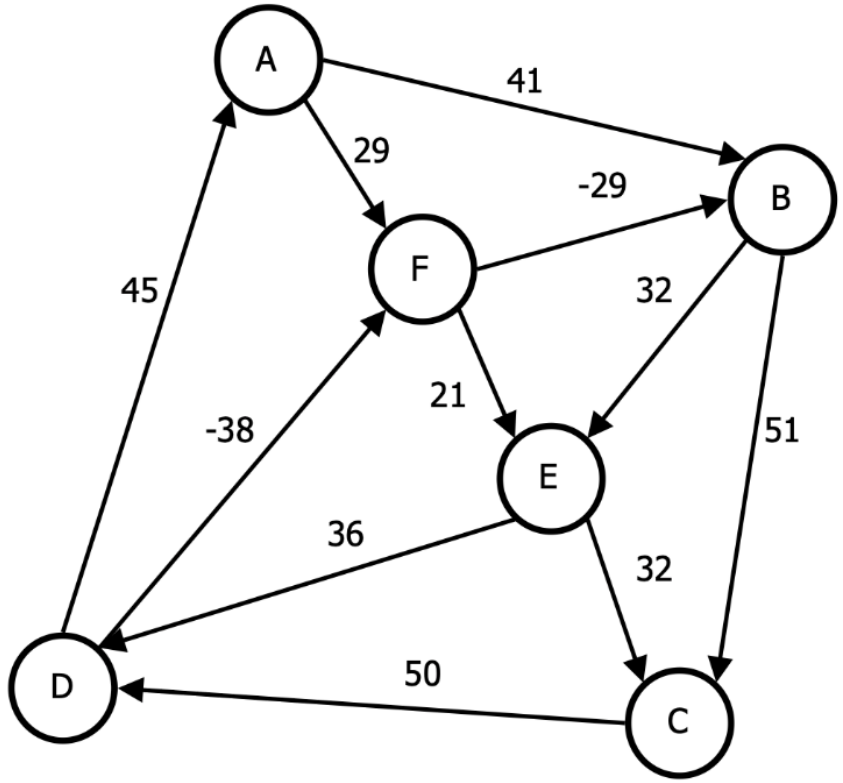
Bellman-Ford Algorithm

- ในกรณีที่กราฟมี weight เป็นค่าติดลบ ไม่สามารถรับประกันได้ว่า Dijkstra's จะถูก
- ให้ใช้ **Bellman–Ford Algorithm**
- (ต้องยอม ถึงแม้จะใช้เวลาเยอะกว่า)

Shortest path of Graph

Bellman-Ford Algorithm

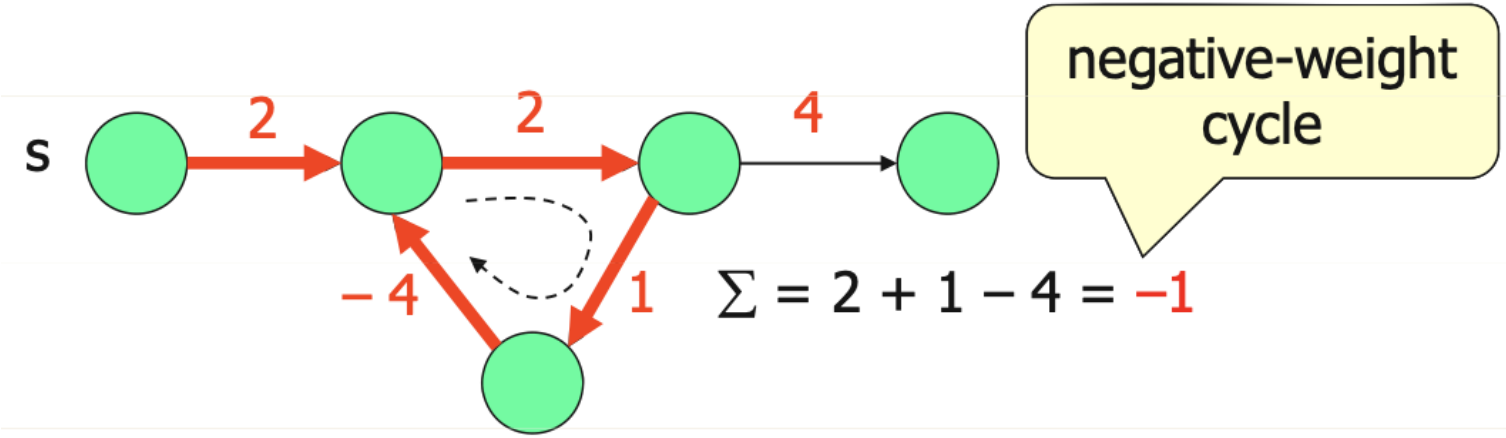
- **ตัวอย่างในชีวิตประจำวัน**
  - แต่ละ **Node** แทน**สินค้า**ทั้งหมดในรายการ
  - **Edge** (u, v) แทน**ราคา**เมื่อซื้อสินค้า u แล้วซื้อสินค้า v
    - เช่น เมื่อซื้อ A -> ซื้อ B จะเสียค่าใช้จ่าย 41 บาท
    - เมื่อซื้อ F -> ซื้อ B จะได้รับส่วนลด 29 บาท
  - การแก้ปัญหาว่า “ควรซื้อสินค้าแต่ละชนิดอย่างไร เพื่อให้ได้ราคาถูกที่สุด เมื่อซื้อสินค้า A เป็นชิ้นแรก”



Shortest path of Graph

Bellman-Ford Algorithm

- **Bellman–Ford Algorithm**
  - **Input:** source vertex
  - **Output:**
    - If no negative cycle detected -> returns **Shortest Path Tree** (Shortest path to all vertices)
    - If negative cycle detected -> the shortest distance will not be calculated, the **cycle** is reported.



คือ **cycle ที่ weight รวมมีค่า < 0**,
ซึ่งทำให้หาจุดสิ้นสุดไม่ได้ เพราะยิ่งเดินในวง ค่ายิ่งลดลงเรื่อย ๆ

Shortest path of Graph

Bellman-Ford Algorithm

- ขั้นตอนของ **Bellman–Ford Algorithm**

กำหนดให้ V = set of vertices, src = Vertex เริ่มต้น

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance โดย ทำด้านล่างนี้ทั้งหมด |V| - 1 ครั้ง
   - แต่ละ **edge (u, v)**

     if dist[v] > dist[u] + weight of edge(u,v):

     dist[v] = dist[u] + weight of edge(u,v)

   ** สรุปคือ วนทุก edge -> แต่ละ edge เก็บค่าน้อยที่สุดไว้ที่ node ปลายทาง และวนต่อจนครบ -> ทำไป |V| -1 รอบ

3. Report if there's negative cycle
   - แต่ละ **edge (u, v)**

     if dist[v] > dist[u] + weight of edge(u,v):

     Report "Graph contains negative weight cycle"

   ** ข้อ 2. เป็นการการันตีแล้วว่านั้นคือเส้นทางที่สั้นที่สุด ถ้าตรงนี้ยังเจออีก แสดงว่าเจอ negative weight cycle

4. If there's **no** negative weight cycle:

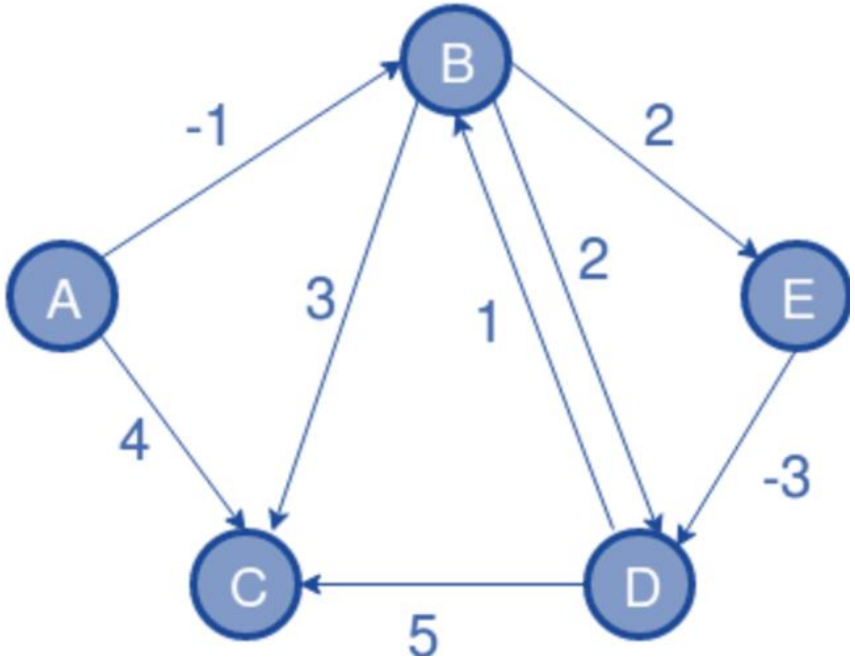
   dist[v] = shortest path จาก src มายัง v เมื่อ v ∈ V

Shortest path of Graph

Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
   For each **edge (u, v)**:
   if dist[v] > dist[u] + w(edge(u,v)):
   dist[v] = dist[u] + w(edge(u,v))
3. Report if there's negative cycle
   For each **edge (u, v)**:
   if dist[v] > dist[u] + weight of edge(u,v):
   "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
   dist[v] = shortest path tree



E = { }

dist =

A	B	C	D	E

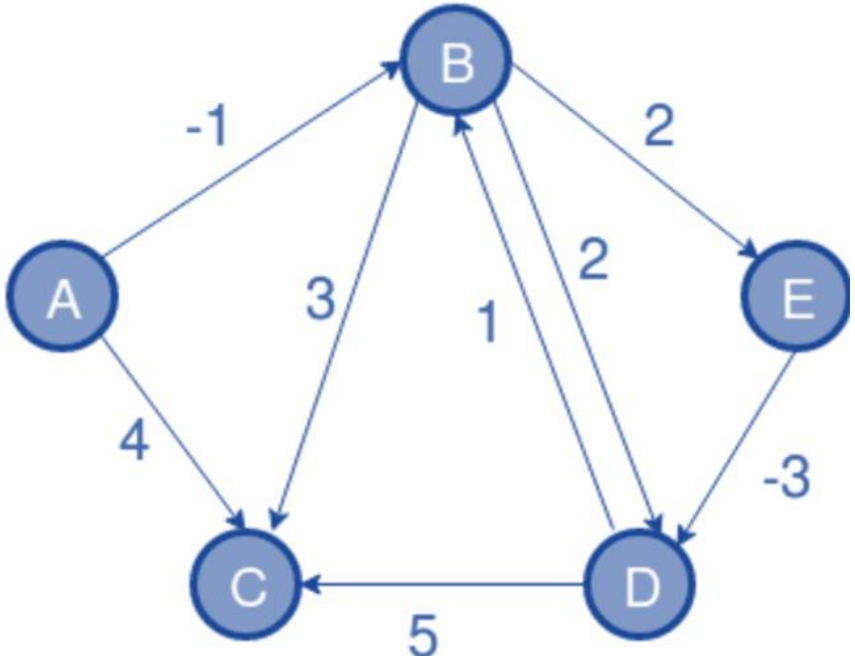
Shortest path of Graph

Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =					

Shortest path of Graph

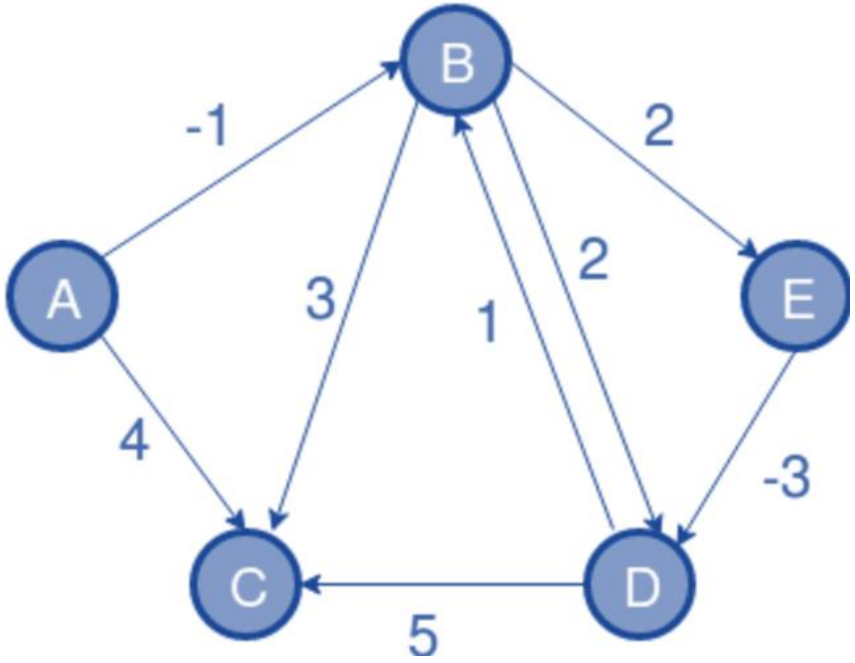
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"

4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	INF	INF	INF	INF

Shortest path of Graph

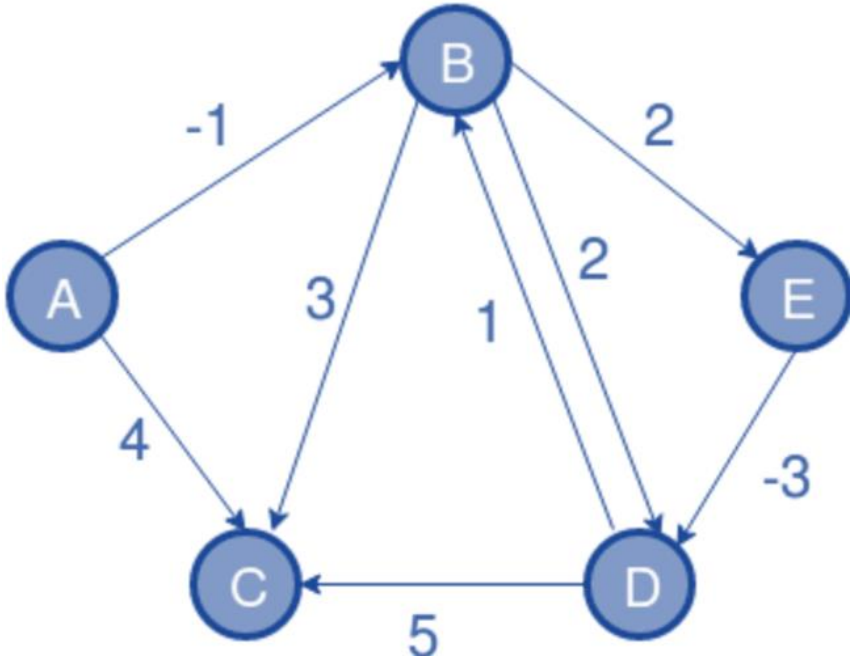
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
   dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	INF	INF	INF	INF

Shortest path of Graph

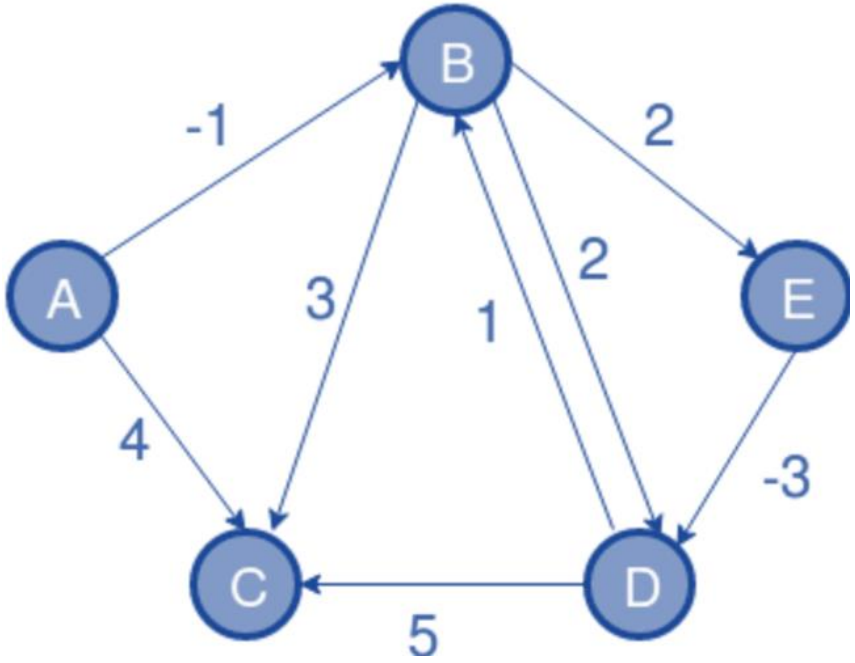
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	INF	INF	INF	INF

Shortest path of Graph

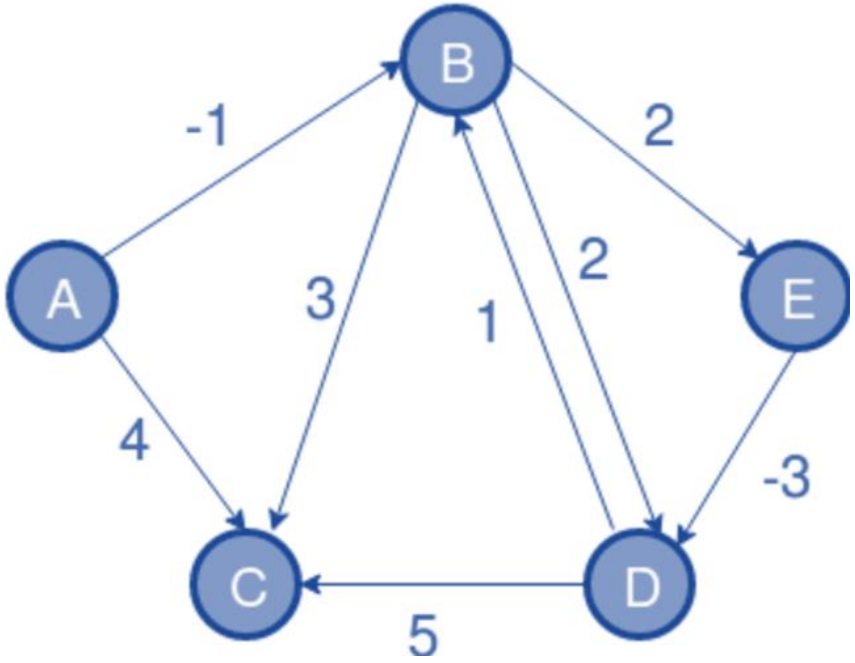
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	INF	INF	INF

Shortest path of Graph

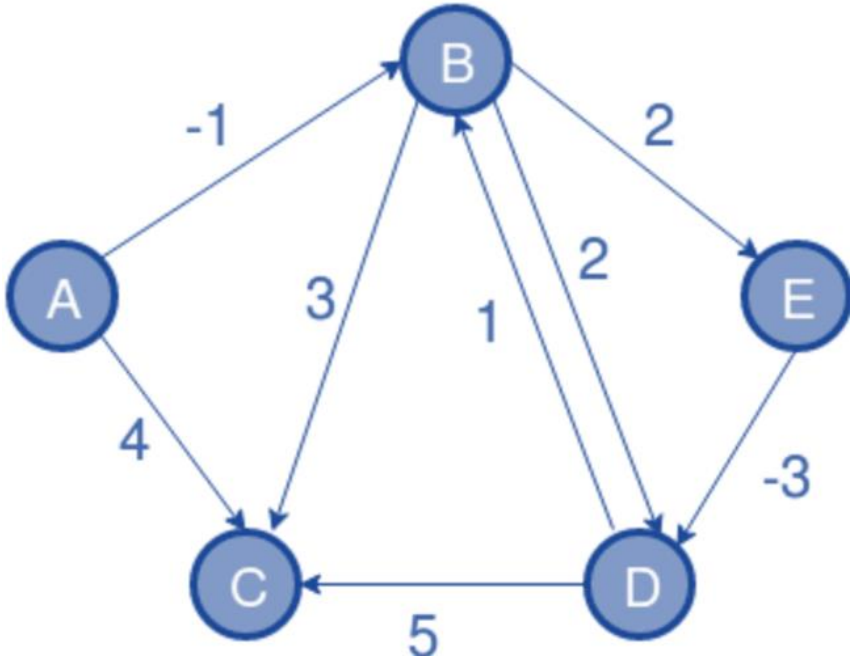
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	INF	INF	INF

Shortest path of Graph

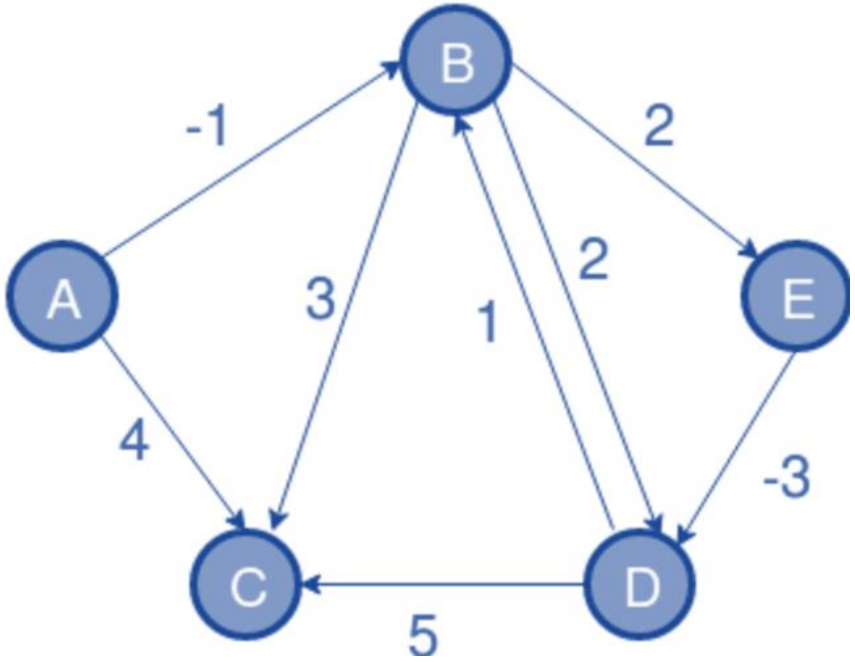
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	4	INF	INF

Shortest path of Graph

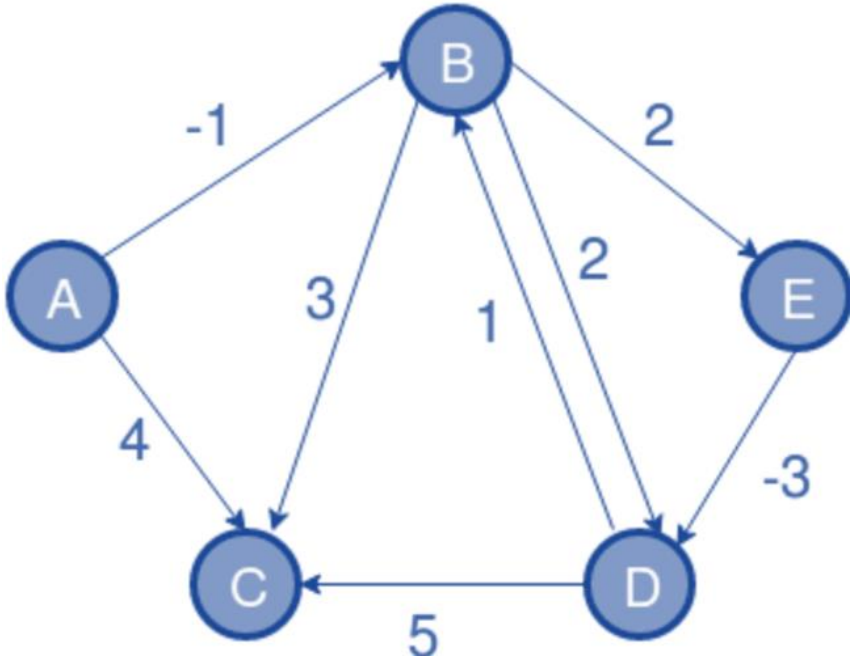
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	4	INF	INF

Shortest path of Graph

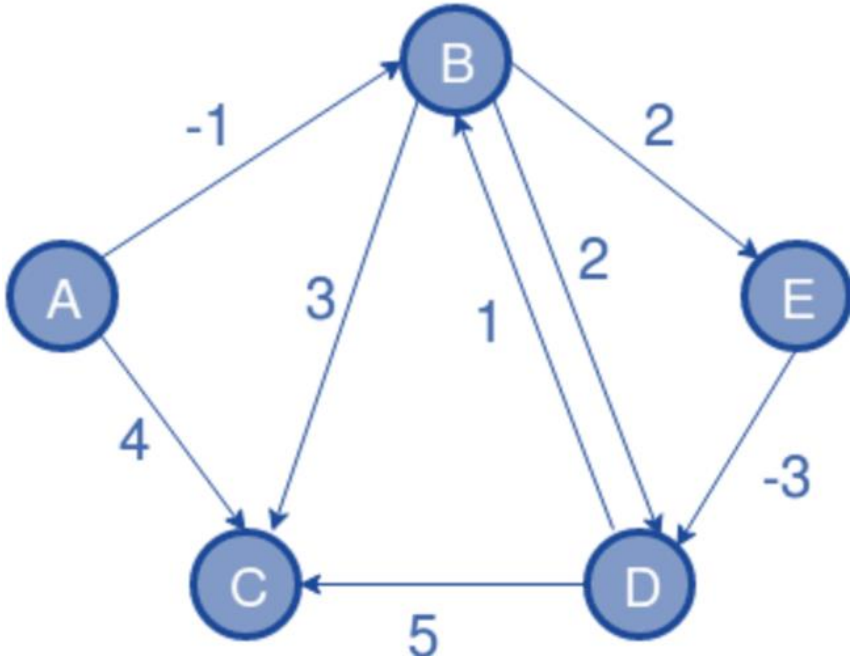
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	INF	INF

Shortest path of Graph

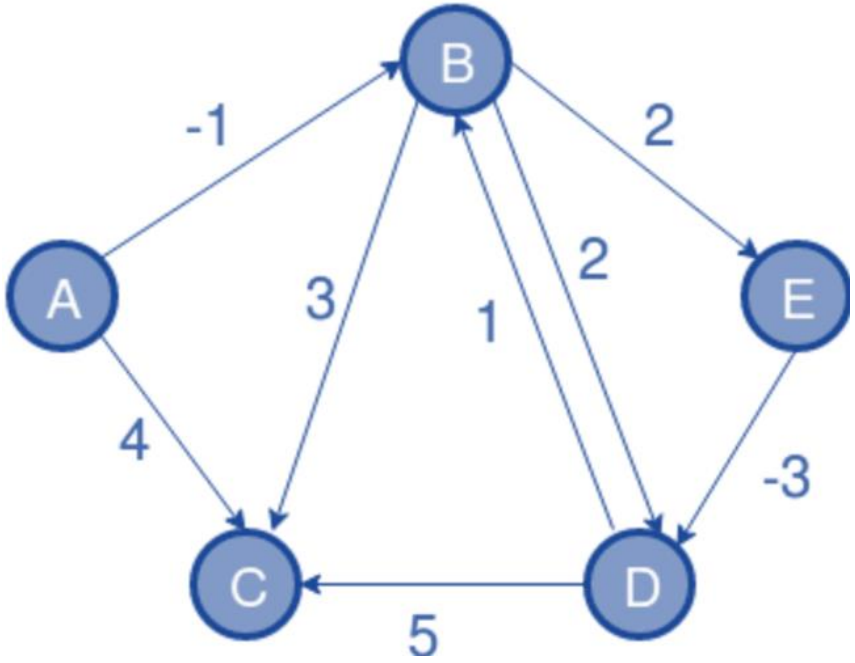
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	INF	INF

Shortest path of Graph

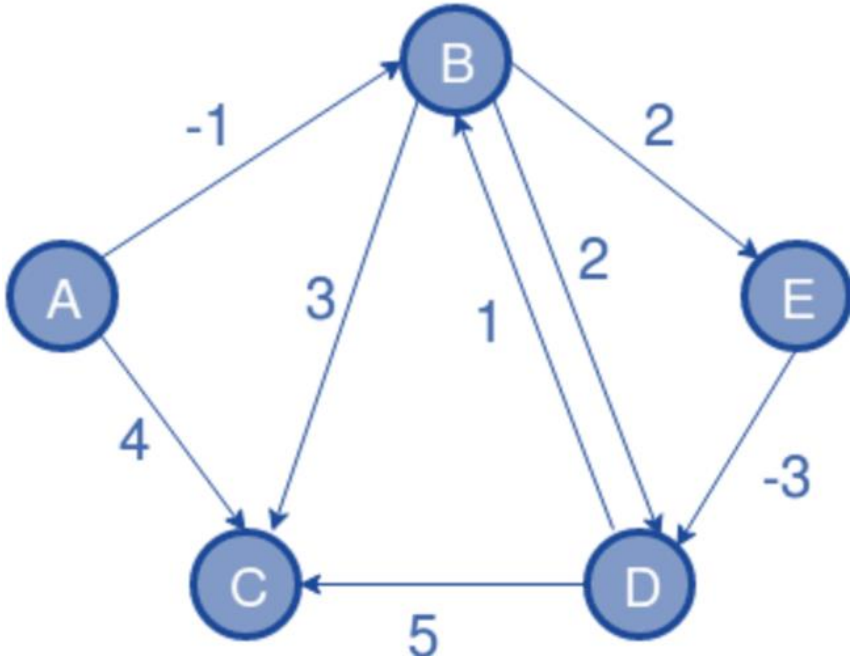
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	1	INF

Shortest path of Graph

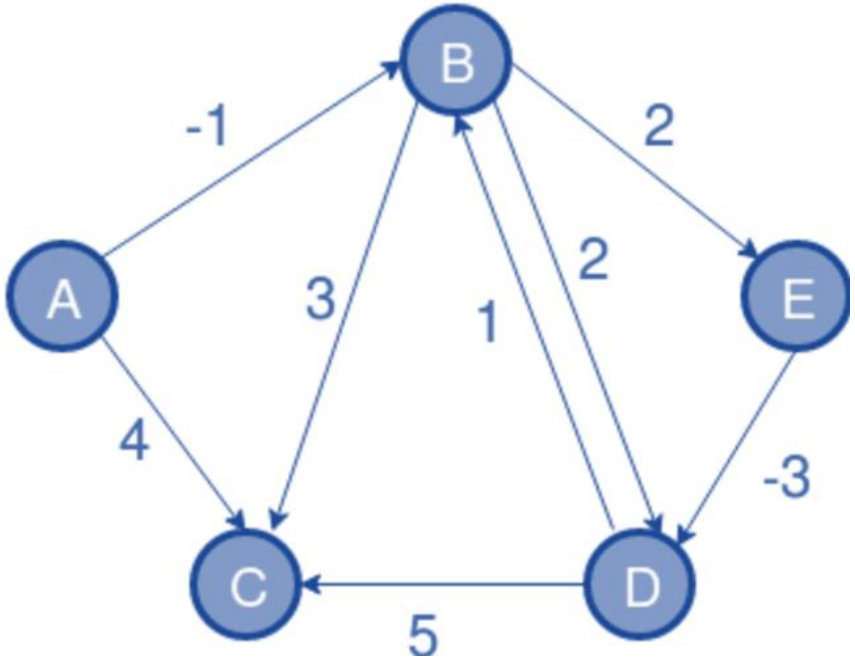
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	1	INF

Shortest path of Graph

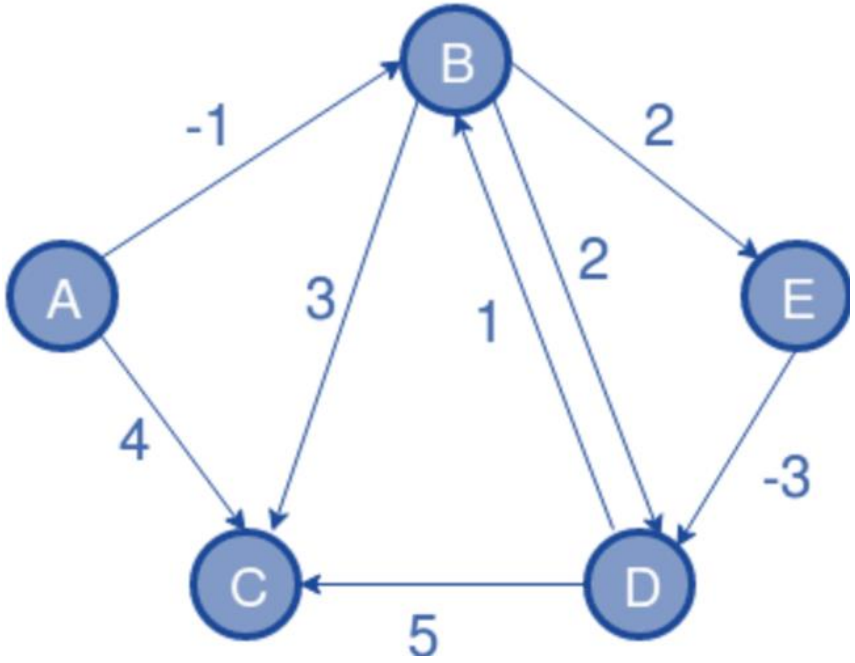
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	1	1

Shortest path of Graph

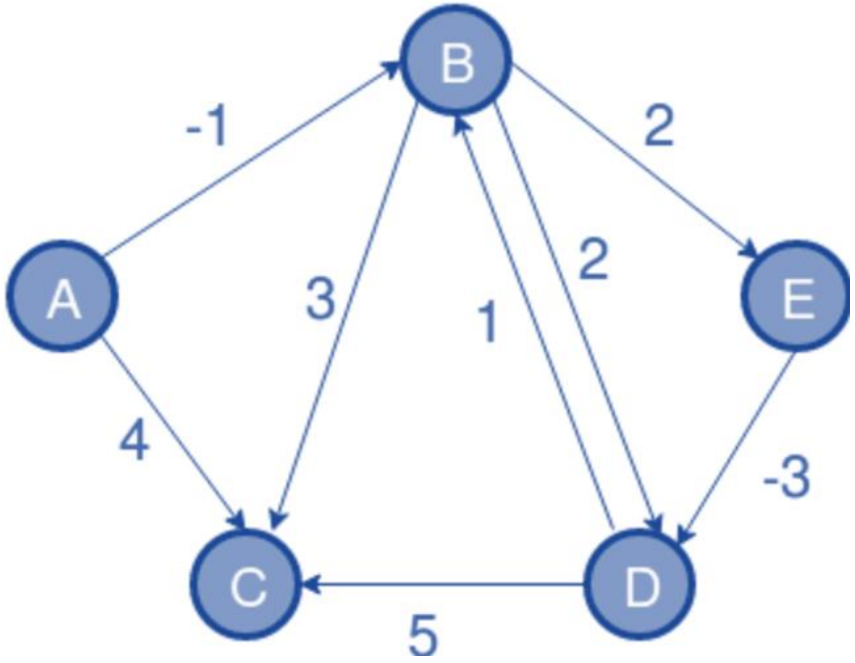
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	1	1

Shortest path of Graph

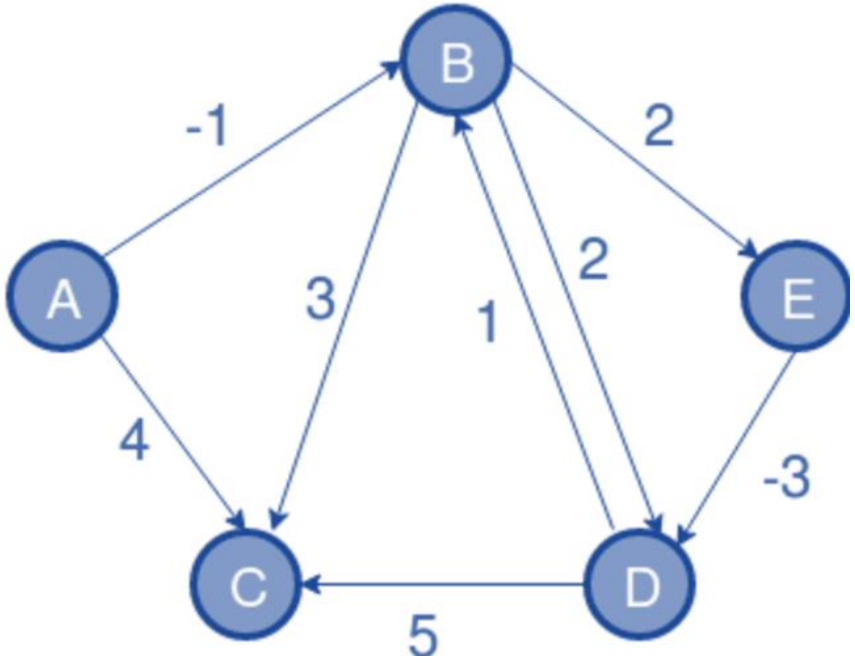
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	1	1

Shortest path of Graph

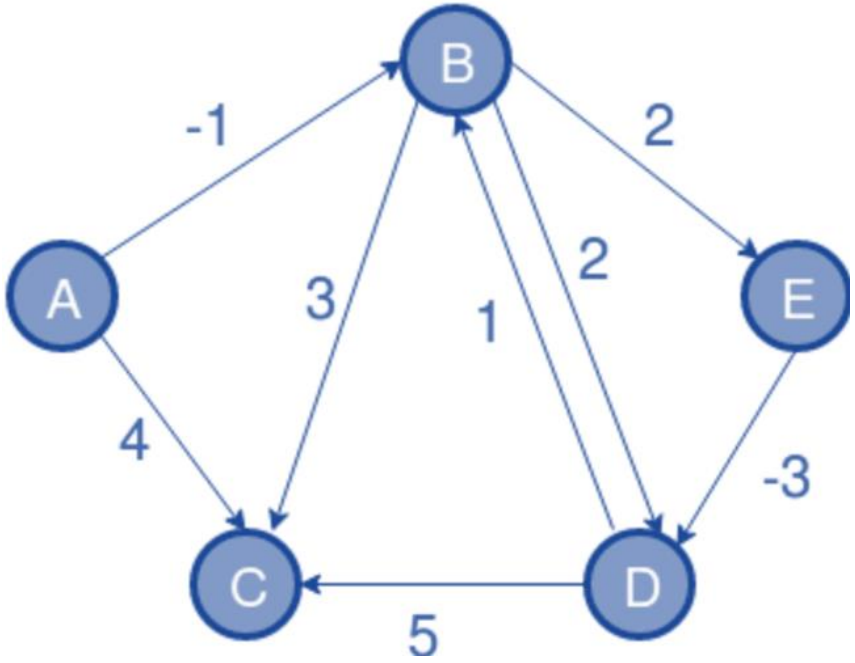
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	1	1

Shortest path of Graph

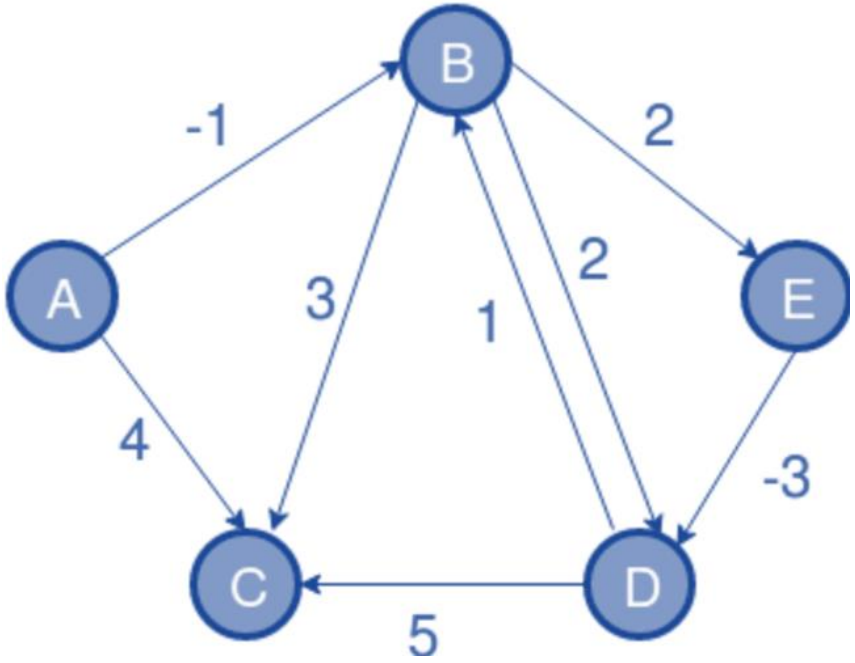
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

i = 1

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	-2	1

Shortest path of Graph

Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance

   For i = 0 to |V| - 1:

   i = 2

      For each **edge (u, v)**:

         if dist[v] > dist[u] + w(edge(u,v)):

            dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle

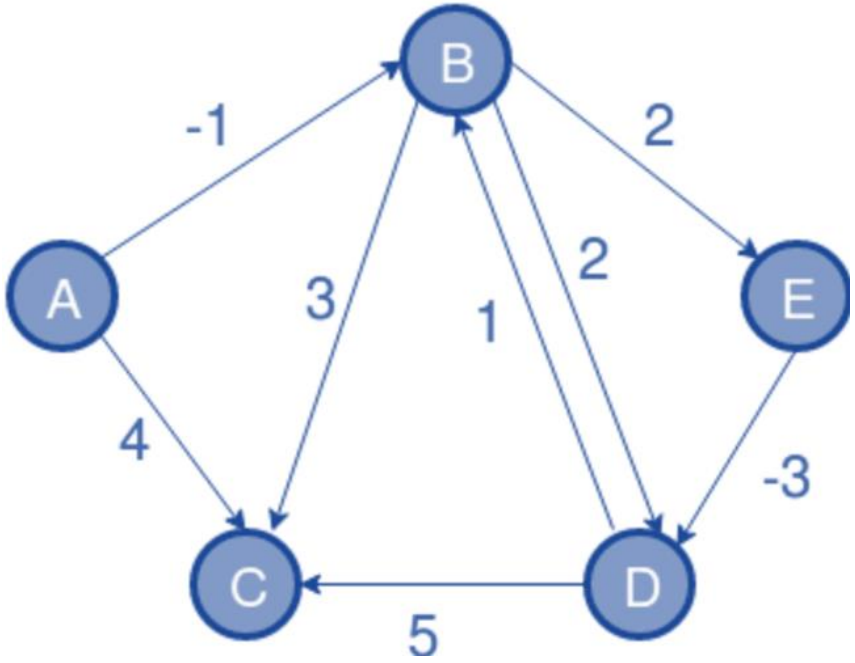
   For each **edge (u, v)**:

      if dist[v] > dist[u] + weight of edge(u,v):

         "**Graph contains negative weight cycle**"

4. If there's **no** negative weight cycle:

      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	-2	1

Shortest path of Graph

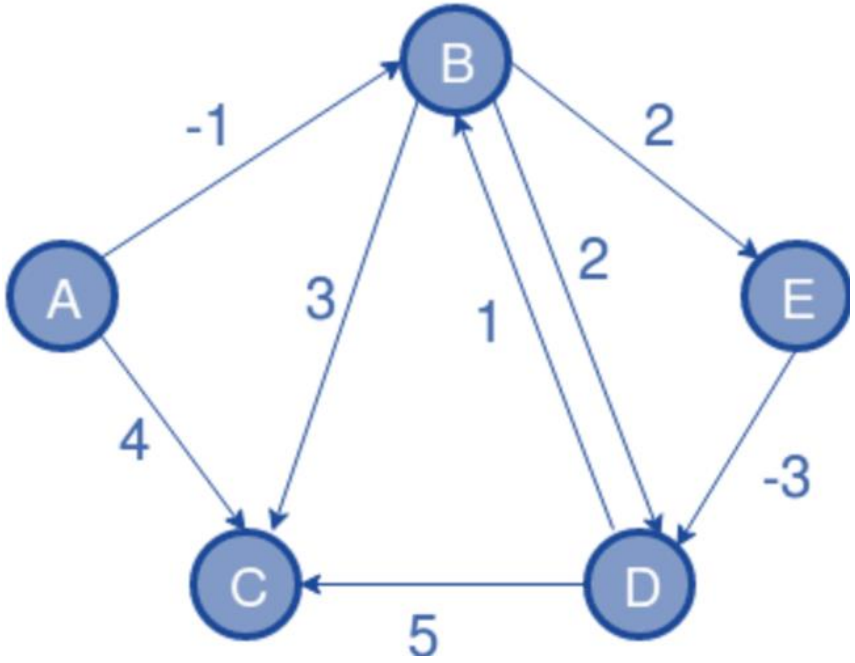
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree

i = 2



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	-2	1

Shortest path of Graph

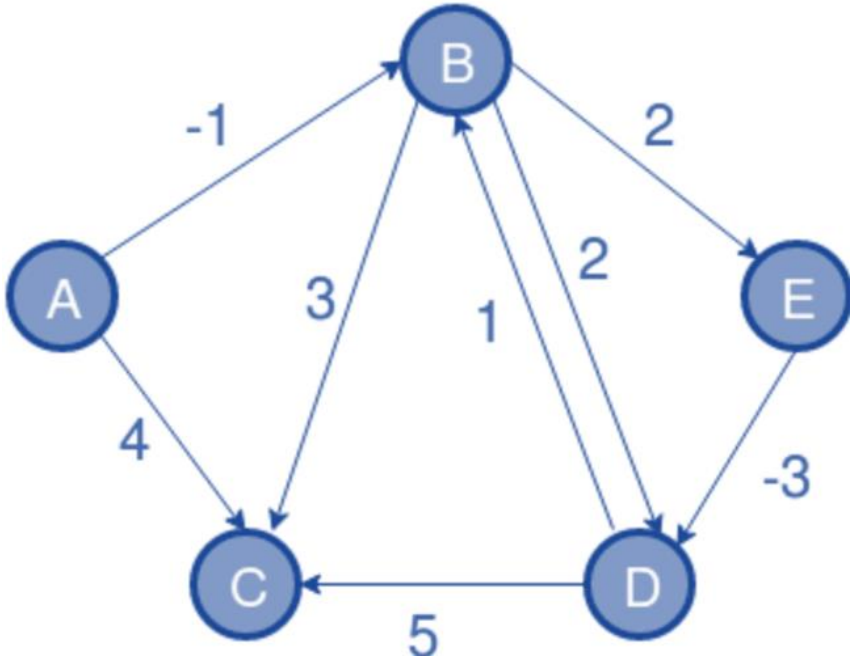
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree

i = 3



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	-2	1

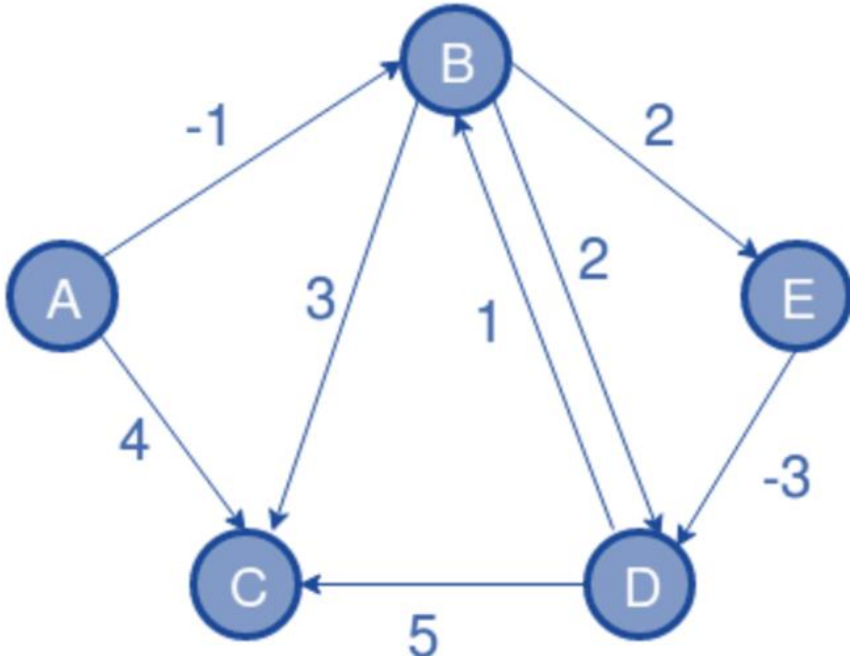
Shortest path of Graph

Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1: **i = 4**
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"
4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	-2	1

Shortest path of Graph

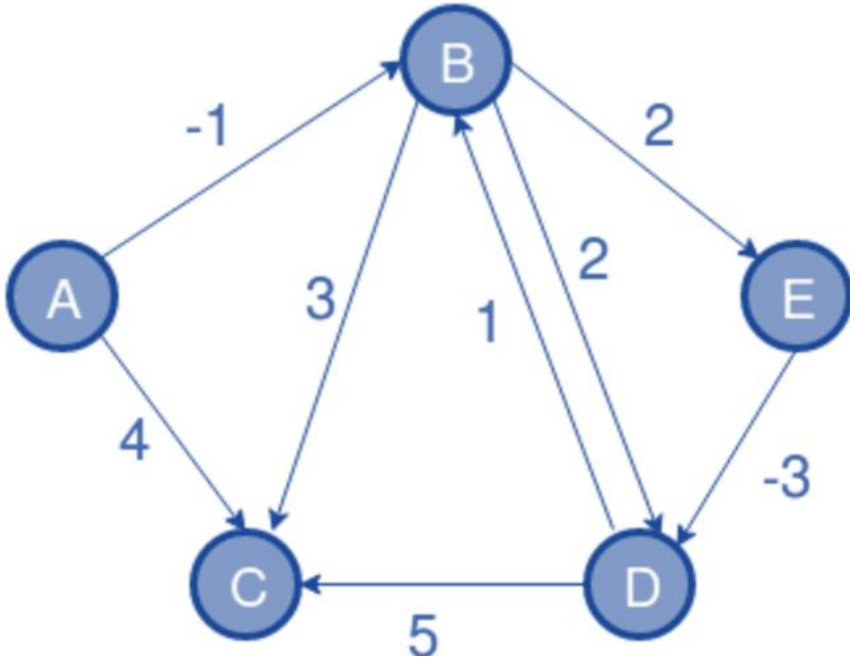
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
      For each **edge (u, v)**:
         if dist[v] > dist[u] + w(edge(u,v)):
            dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle
   For each **edge (u, v)**:
      if dist[v] > dist[u] + weight of edge(u,v):
         "**Graph contains negative weight cycle**"

4. If there's **no** negative weight cycle:
      dist[v] = shortest path tree



E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	-2	1

Shortest path of Graph

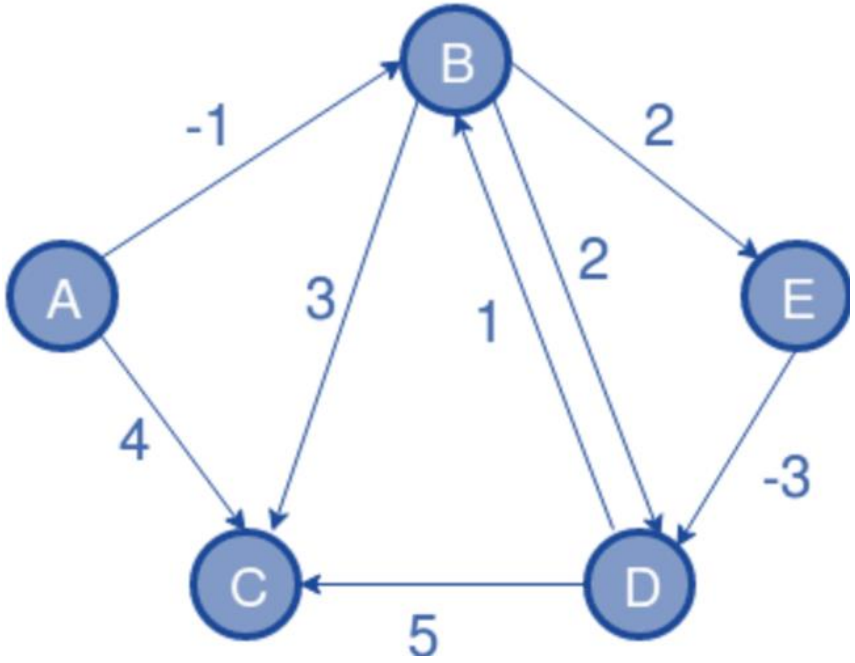
Bellman-Ford Algorithm

- **ตัวอย่าง** (src = A)

1. Initial array **dist** size |V| with **INF**, except for **dist[src] = 0**
2. Calculate shortest distance
   For i = 0 to |V| - 1:
   For each **edge (u, v)**:
   if dist[v] > dist[u] + w(edge(u,v)):
   dist[v] = dist[u] + w(edge(u,v))

3. Report if there's negative cycle
   For each **edge (u, v)**:
   if dist[v] > dist[u] + weight of edge(u,v):
   "**Graph contains negative weight cycle**"

4. If there's **no** negative weight cycle:
   dist[v] = shortest path tree

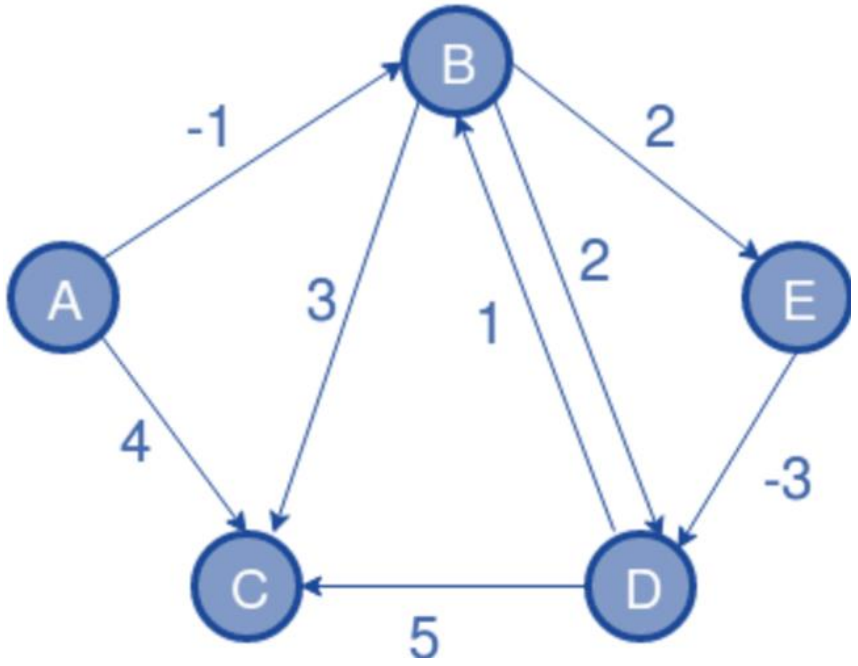


E = {(A, B), (A, C), (B, C), (B, D), (B, E), (D, C), (D, B), (E, D)}

	A	B	C	D	E
dist =	0	-1	2	-2	1

Shortest path of Graph

Bellman-Ford Algorithm



Let’s code !!

[06-Bellman-Ford.cpp]

adjMatrix https://pastebin.com/e8gtvpad

adjList https://pastebin.com/S7ciAxii

All-pair Shortest path

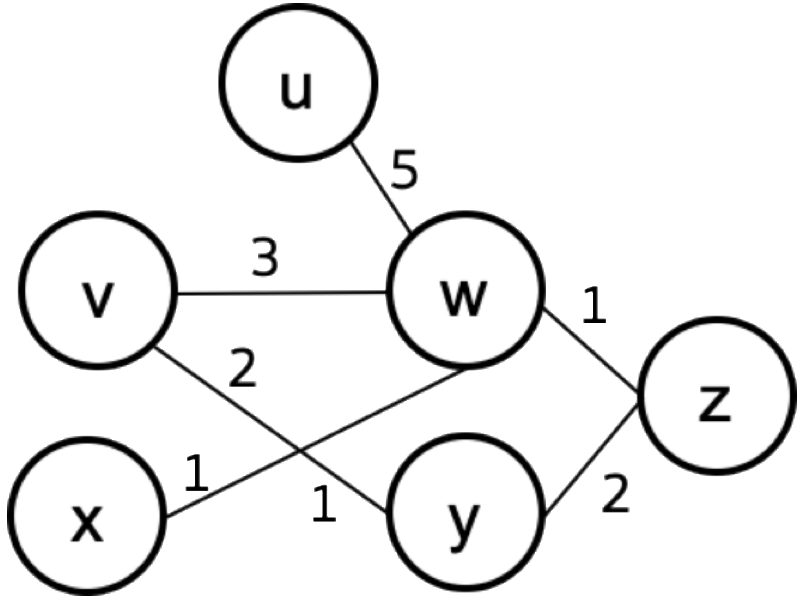
คืออะไร, Algorithm

All-pair Shortest path of Graph

คืออะไร

- **Shortest Path (แบบแรก)**
  - **มีจุดเริ่มต้น (source) 1 จุด ->** เส้นทางใดสั้นที่สุด

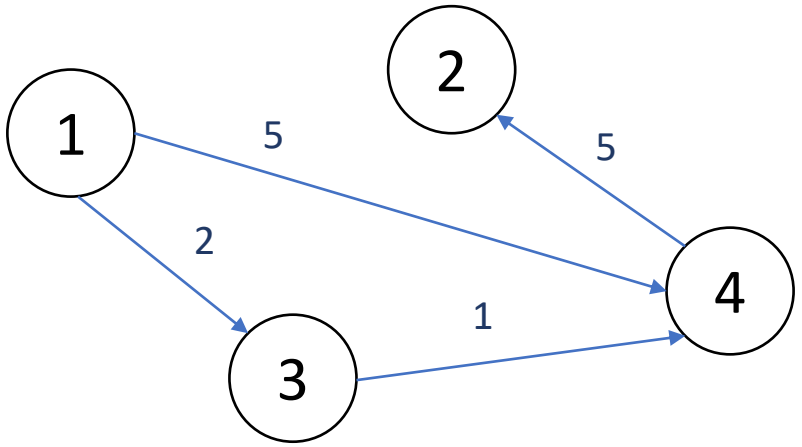
- **All-pair Shortest Path**
  - **ทุก ๆ Node u, v ที่ไปถึงกันได้** -> หาเส้นทางที่ใกล้ที่สุด จาก u ไป v



All-pair Shortest path of Graph

Floyd-Warshall Algorithm

- **Floyd-Warshall Algorithm**
  - ทุก ๆ Node u, v ที่ไปถึงกันได้ -> ระยะทางที่ใกล้ที่สุด จาก u ไป v เป็นเท่าไหร่บ้าง



Node	distance
1, 2	
1, 3	
1, 4	
3, 2	
3, 4	
4, 2	

All-pair Shortest path of Graph

Floyd-Warshall Algorithm

- **ขั้นตอนของ Floyd-Warshall Algorithm**

  “ ในการหาเส้นทางจาก i -> j ถ้ามีเส้นทางที่อ้อมไป (i -> k -> j) แล้วสั้นกว่า จะเก็บไว้เรื่อย ๆ ”

กำหนดให้ input graph G = (V, E)

1. Initialize array **dist** ขนาด V x V

2. **dist = G**
   dist[i][j] = 0 if i = j และ dist[i][j] = INF if no (i, j) in E

3. for (k=0; k<V; k++):
   for (i=0; i<V; i++):
   for (j=0; j<V; j++):
   if (**dist[i][k] + dist[k][j] < dist[i][j]**)
   dist[i][j]) = dist[i][k] + dist[k][j]

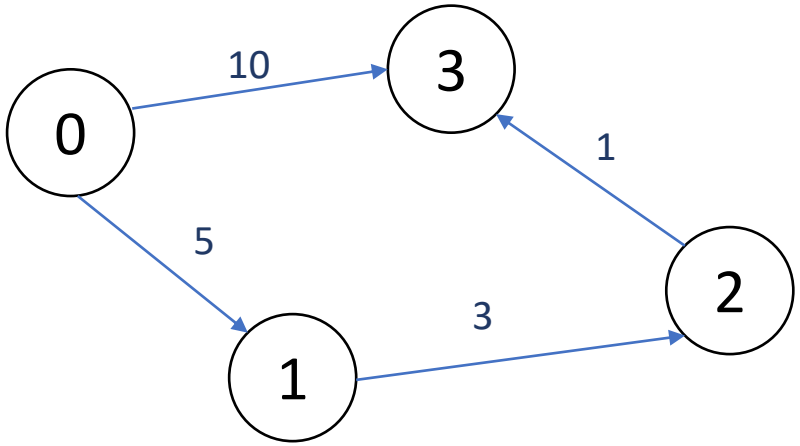
All-pair Shortest path of Graph

Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E




```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```

All-pair Shortest path of Graph

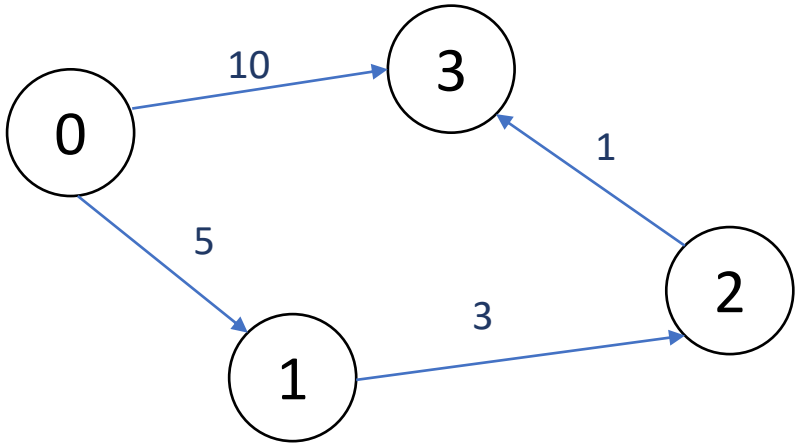
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



dist

	0	1	2	3
0				
1				
2				
3				

All-pair Shortest path of Graph

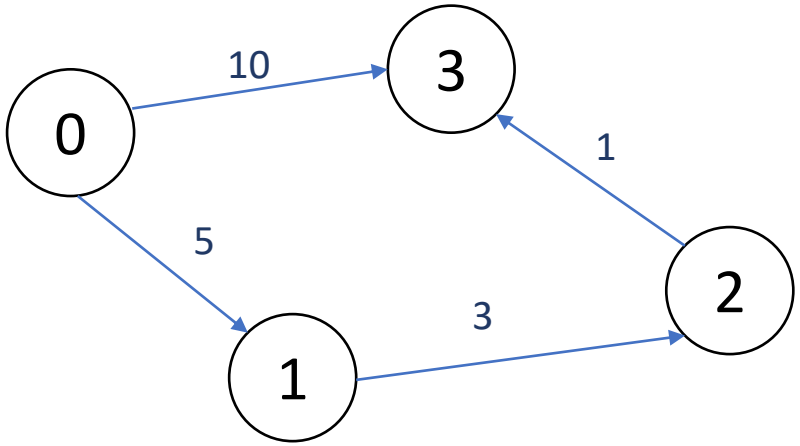
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



dist

	0	1	2	3
0				
1				
2				
3				

All-pair Shortest path of Graph

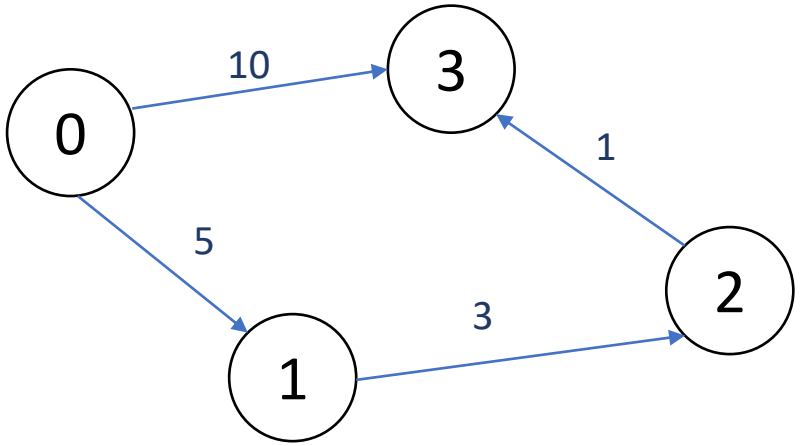
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



dist

	0	1	2	3
0	**0**	**5**	**INF**	**10**
1	**INF**	**0**	**3**	**INF**
2	**INF**	**INF**	**0**	**1**
3	**INF**	**INF**	**INF**	**0**

All-pair Shortest path of Graph

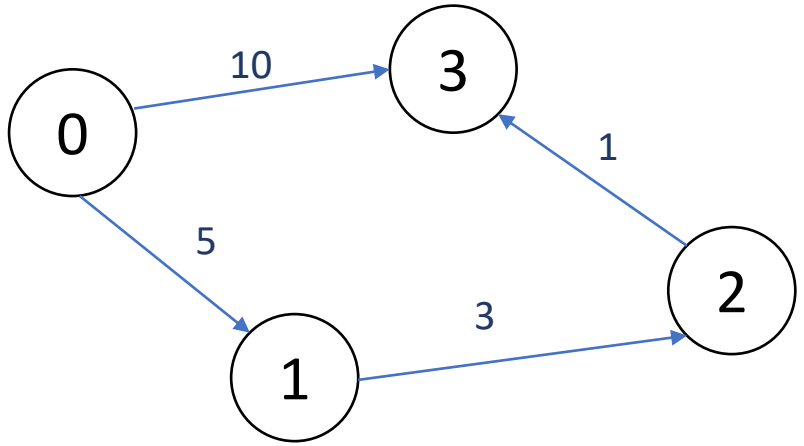
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



i = 0 j = 0 k = 0 →

dist

	0	1	2	3
0	0	5	INF	10
1	INF	0	3	INF
2	INF	INF	0	1
3	INF	INF	INF	0

All-pair Shortest path of Graph

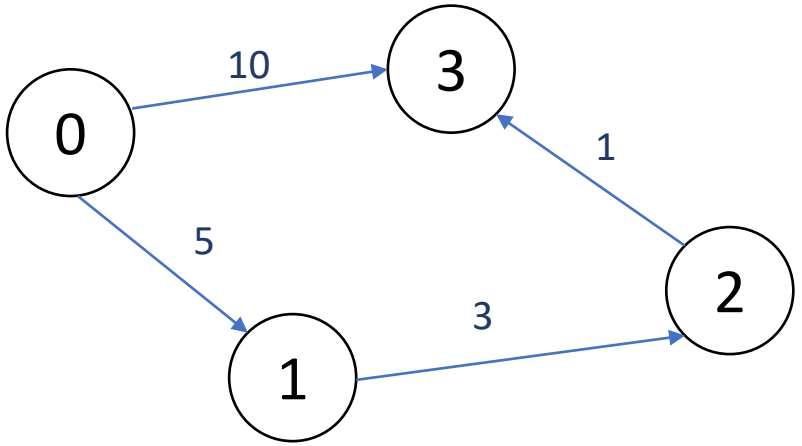
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



dist

	0	1	2	3
0	0	5	INF	10
1	INF	0	3	INF
2	INF	INF	0	1
3	INF	INF	INF	0

i = 0 j = 0 k = 0 → False

All-pair Shortest path of Graph

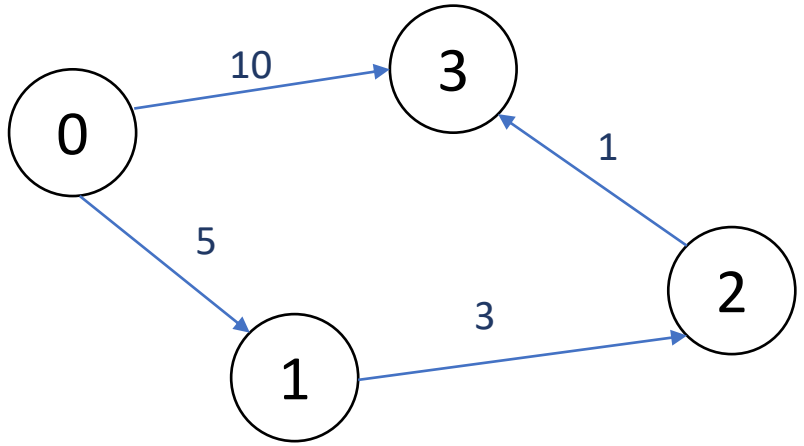
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



i = 0 j = 1 k = 0 →

dist

	0	1	2	3
0	0	5	INF	10
1	INF	0	3	INF
2	INF	INF	0	1
3	INF	INF	INF	0

All-pair Shortest path of Graph

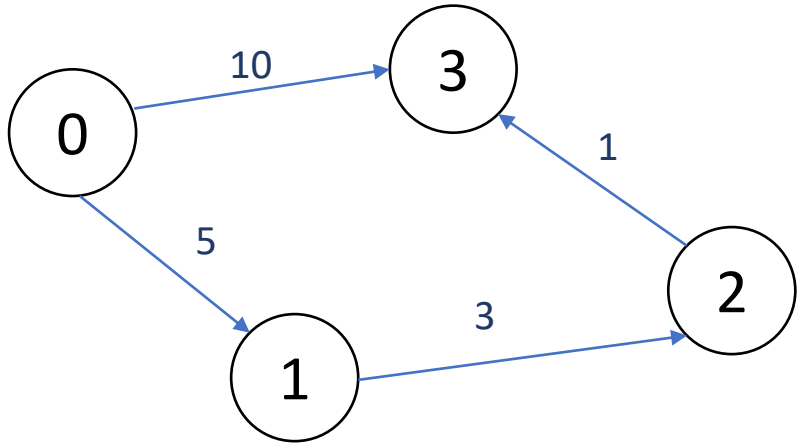
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



i = 0 j = 2 k = 0 →

dist

	0	1	2	3
0	0	5	INF	10
1	INF	0	3	INF
2	INF	INF	0	1
3	INF	INF	INF	0

All-pair Shortest path of Graph

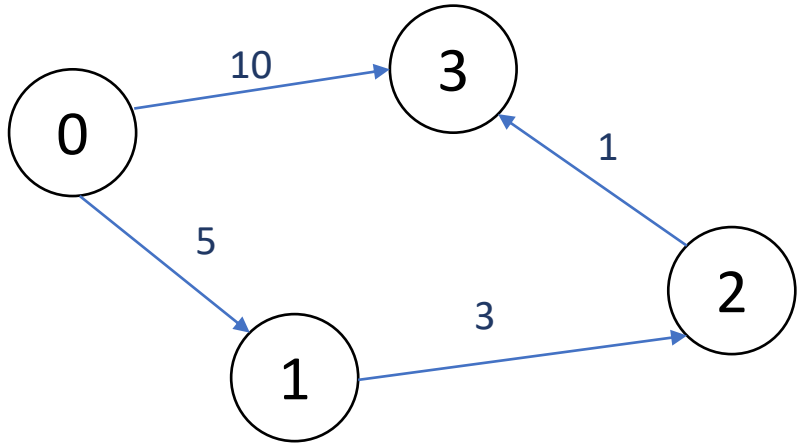
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



i = 0 j = 3 k = 0 →

dist

	0	1	2	3
0	0	5	INF	10
1	INF	0	3	INF
2	INF	INF	0	1
3	INF	INF	INF	0

All-pair Shortest path of Graph

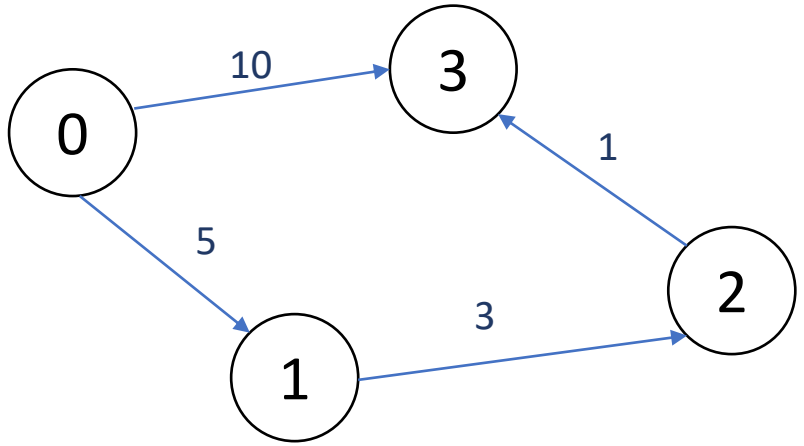
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



i = 1 j = 0 k = 0 →

dist

	0	1	2	3
0	0	5	INF	10
1	INF	0	3	INF
2	INF	INF	0	1
3	INF	INF	INF	0

All-pair Shortest path of Graph

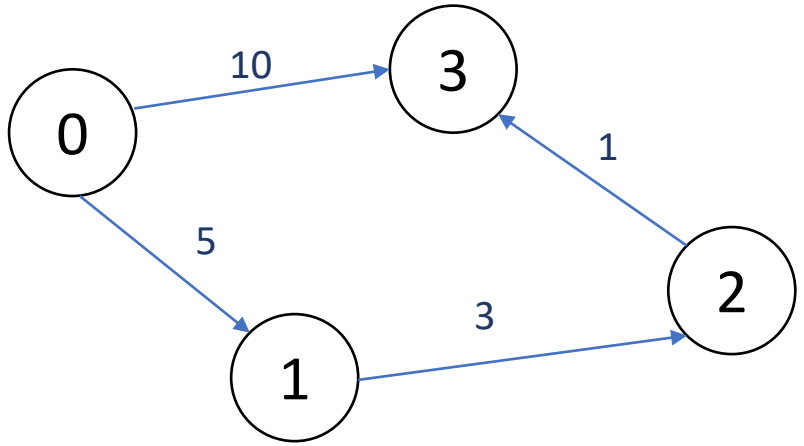
Floyd-Warshall Algorithm

กำหนดให้ input graph G = (V, E)

Initialize array **dist** ขนาด V x V

dist = **G**
dist[i][j] = 0 if i = j
dist[i][j] = INF if no (i, j) in E

```
for (k=0; k<V; k++):
    for (i=0; i<V; i++):
        for (j=0; j<V; j++):
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j]) = dist[i][k] + dist[k][j]
```



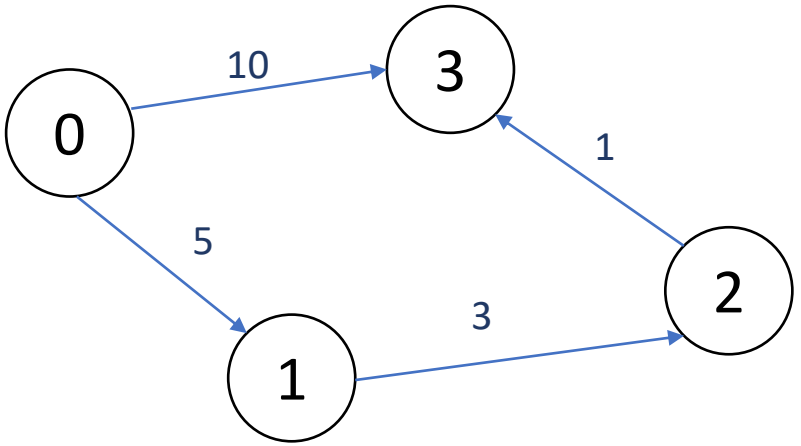
i = j = k = →

dist

	0	1	2	3
0	0	5	INF	10
1	INF	0	3	INF
2	INF	INF	0	1
3	INF	INF	INF	0

All-pair Shortest path of Graph

Floyd-Warshall Algorithm



Let’s code !!

[07-Floyd-Warshall.cpp]

https://pastebin.com/WMgwUzVy

All-pair Shortest path of Graph

Floyd-Warshall Algorithm

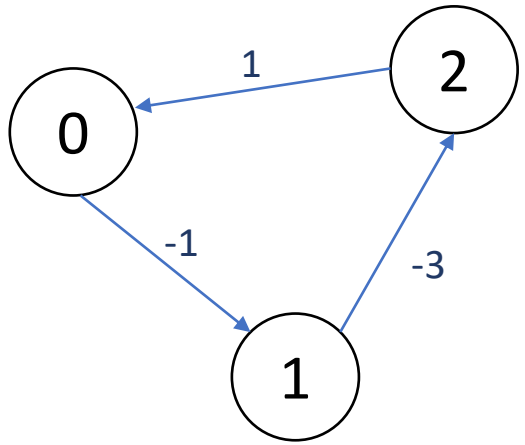
- **Dijkstra's Algorithm vs. Floyd-Warshall Algorithm**

Dijkstra's	Floyd-Warshall
Shortest Path (Single source)	All-pair Shortest Path
O(ElogV) → all nodes = $O(V^3 logV)$	$O(V^3)$
Not Support negative weights	Support negative weights (if no negative weight cycle detected)

All-pair Shortest path of Graph

Floyd-Warshall Algorithm

- How to detect **negative weight cycle** using **Floyd-Warshall Algorithm**
  - ปกติแล้ว dist[i][j] = 0 เสมอ เมื่อ i == j
  - ถ้าพบว่า **dist[i][j] < 0** แสดงว่าพบ **negative weight cycle**



Sum of edges in this cycle = -1 + -3 + 1 = -3
(จึงเป็น Negative Weight Cycle)

dist

	0	1	2
0	**0**	**-1**	**INF**
1	**INF**	**0**	**-3**
2	**1**	**INF**	**0**

i = j = k =

if (**dist[i][k] + dist[k][j] < dist[i][j]**)
dist[i][j]) = dist[i][k] + dist[k][j]

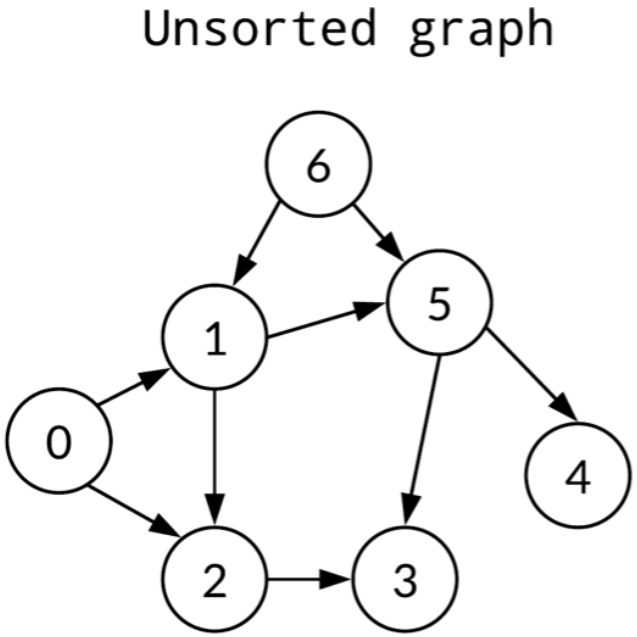
Topological sort

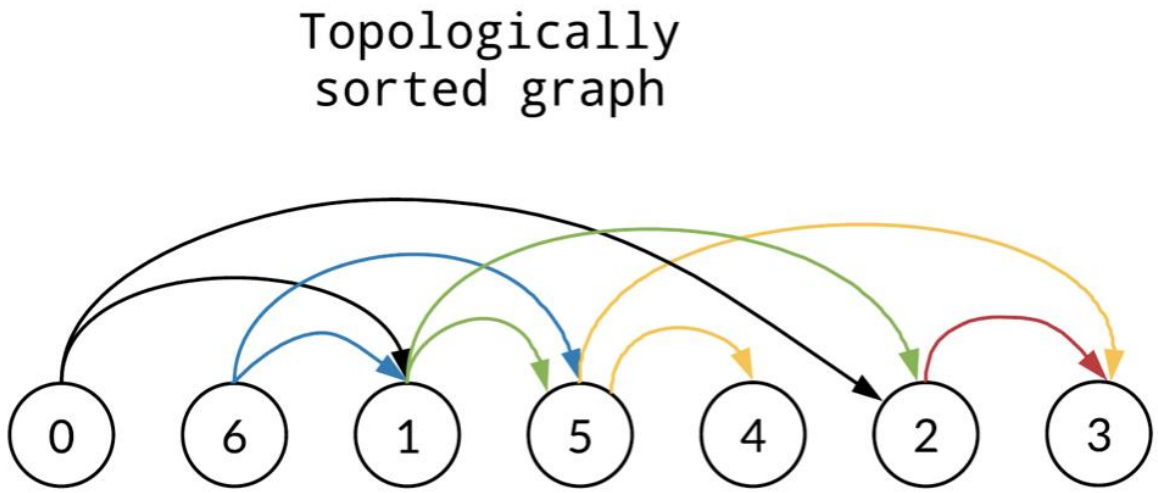
คืออะไร, Algorithm

Topological sort of Graph

คืออะไร, Algorithm

- จาก Directed Acyclic Graph (DAG) G = (V, E)
- Directed Acyclic Graph คือ กราฟแบบมีทิศทาง ที่ไม่มี cycle อยู่ข้างใน
- **Topological sort of G คือ ลำดับของ Vertex ที่ทุก ๆ edge(u, v) u จะมาก่อน v เสมอ**

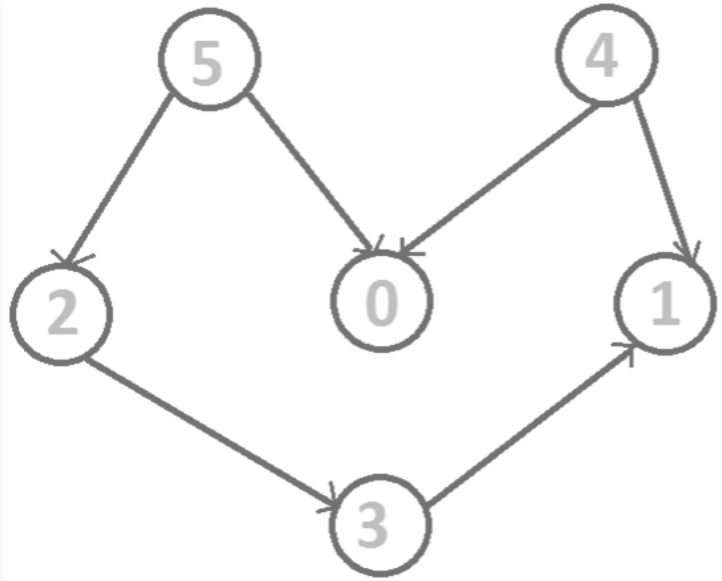




Topological sort of Graph

คืออะไร, Algorithm

- ข้อใดเป็น **Topological sort** ของกราฟด้านล่างบ้าง?

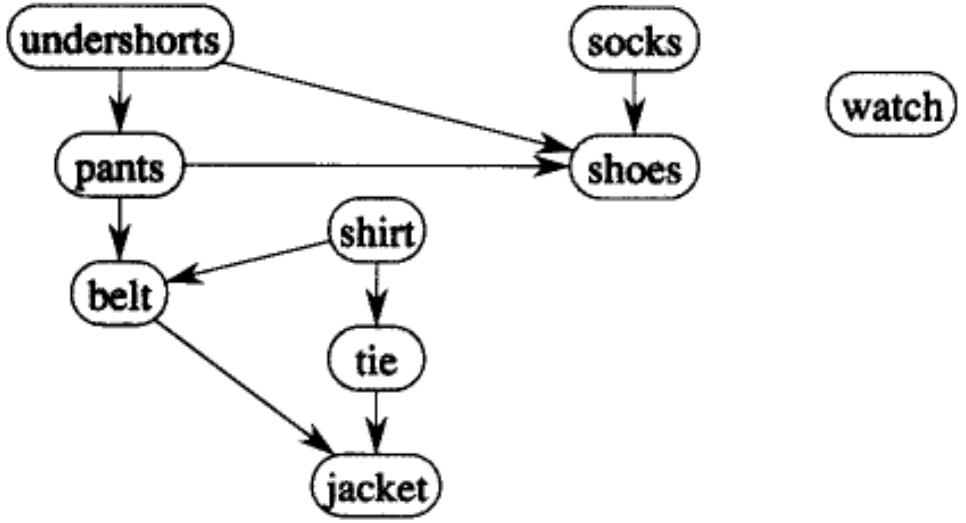


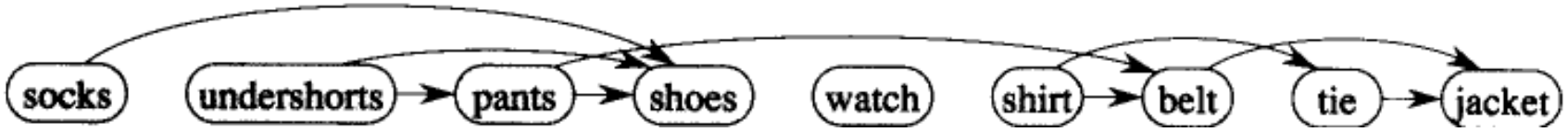
- ☐ 5 4 2 3 1 0
- ☐ 4 5 2 3 1 0
- ☐ 5 2 3 1 4 0
- ☐ 4 2 3 1 5 0

Topological sort of Graph

คืออะไร, AIgorithm

- ประโยชน์

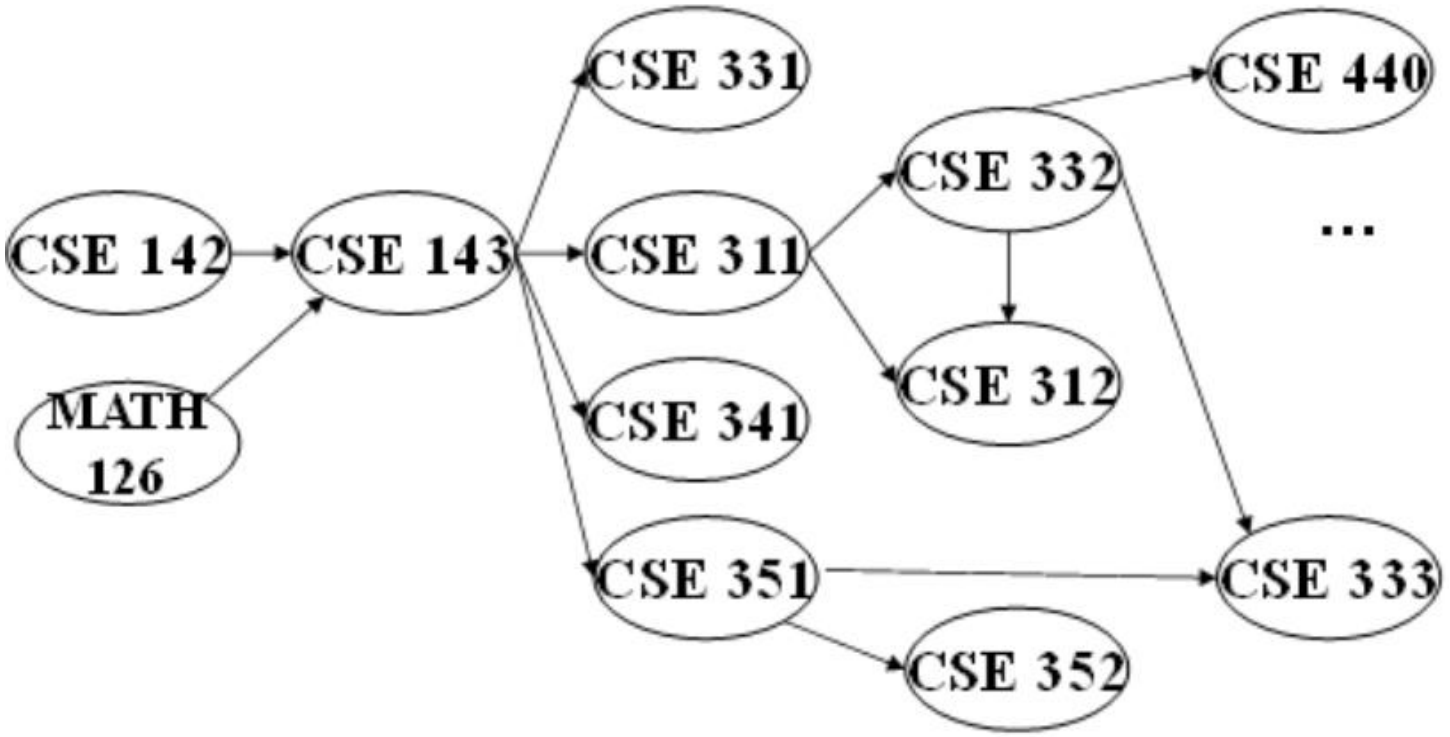




Topological sort of Graph

คืออะไร, Algorithm

- ประโยชน์



Topological sort of Graph

คืออะไร, Algorithm

- วิธีหา **Topological sort** “ประยุกต์จาก DFS แต่เพิ่ม stack เพื่อเก็บ node ที่เป็นลำดับก่อนหน้าให้ออกมาก่อน ”

```
DFS:
    initialize all in visited[V] = false

    //call recursive function
    DFS_util(source, visited);
```

```
DFS_util(v, visited):
    show v;
    mark visited[v] = true;
    For each node in adjacency(v):
        if visited[node] == false:
            DFS(node, visited);
```

```
TopologicalSort:
    initialize all in visited[V] = false
    empty stack S

    //call recursive function for each Vertex v
    for each v:
        if visited[v] != true
            TopologicalSortUtil(v, visited, S)

    // print from stack
```

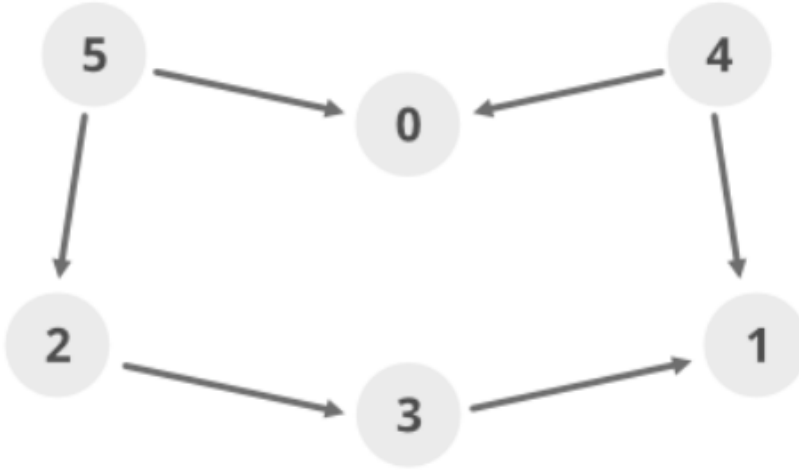
```
TopologicalSortUtil(v, visited):
    visited[v] = true // ** Not print yet
    For each node in adjacency(v):
        if visited[node] == false:
            TopologicalSortUtil(node, visited, S)

    push v to stack S
```

Topological sort of Graph

คืออะไร, Algorithm

```
TopologicalSort:
    initialize all in visited[V] = false
    empty stack S

    //call recursive function for each Vertex v
    for each v:
        if visited[v] != true
            TopologicalSortUtil(v, visited, S)

    // print from stack
```

```
TopologicalSortUtil(v, visited):
    visited[v] = true
    for each node in adjacency(v):
        if visited[node] == false:
            TopologicalSortUtil(node, visited, S)

    push v to stack S
```



Adja cent list (G)

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

	0	1	2	3	4	5
visited =	F	F	F	F	F	F

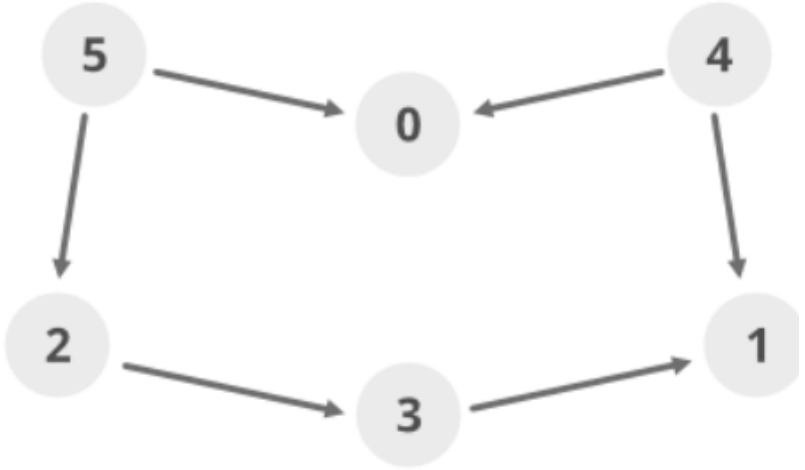
stack S = [ ]

Topological sort of Graph

คืออะไร, Algorithm

```
TopologicalSort:
    initialize all in visited[V] = false
    empty stack S

    //call recursive function for each Vertex v
    for each v:
        if visited[v] != true
            TopologicalSortUtil(v, visited, S)

    // print from stack
```

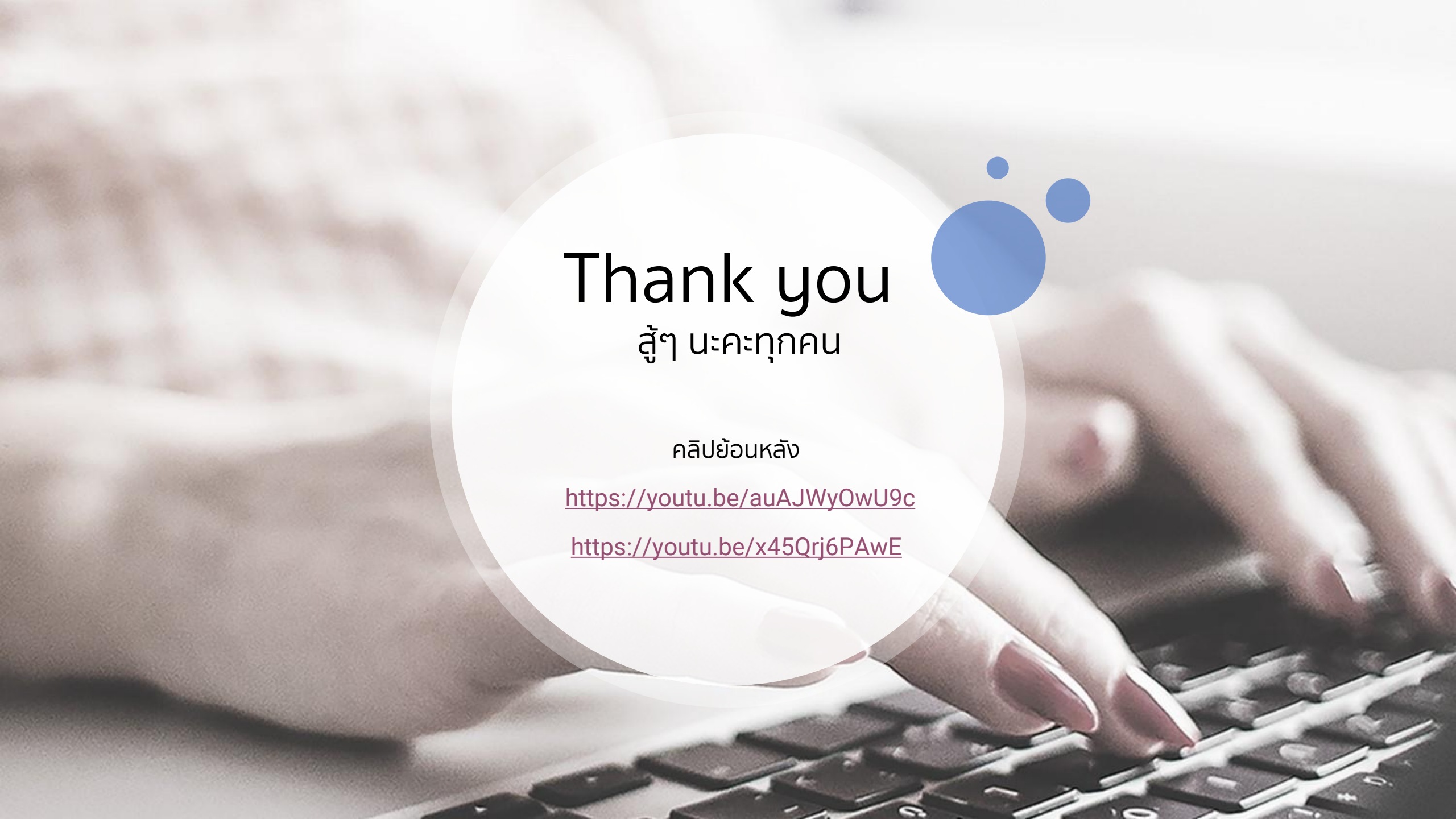
```
TopologicalSortUtil(v, visited):
    visited[v] = true
    for each node in adjacency(v):
        if visited[node] == false:
            TopologicalSortUtil(node, visited, S)

    push v to stack S
```



Adja cent list (G)

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

“ 5 4 2 3 1 0 ”



Thank you

สู้ๆ นะคะทุกคน

คลิปย้อนหลัง

https://youtu.be/auAJWyOwU9c

https://youtu.be/x45Qrj6PAwE

Appendix

Shortest path of Graph

Dijkstra's Algorithm

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, u, v, w;
  vector<vii> AdjList;
  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii());
// assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjList[u].push_back(ii(v, w));
// directed graph
 }
```

Shortest path of Graph

Dijkstra's Algorithm

```
vi dist(V, INF);
 dist[s] = 0;                        // INF = 1B to avoid overflow
 priority_queue< ii, vector<ii>, greater<ii> > pq;
 pq.push(ii(0, s));
                              // ^to sort the pairs by increasing distance from s
 while (!pq.empty()) {                                          // main loop
   ii front = pq.top();
   pq.pop();      // greedy: pick shortest unvisited vertex
   int d = front.first, u = front.second;
   if (d > dist[u]) continue;              // this check is important, see the explanation
   for (int j = 0; j < (int)AdjList[u].size(); j++) {
       ii v = AdjList[u][j];                            // all outgoing edges from u
       if (dist[u] + v.second < dist[v.first]) {
         dist[v.first] = dist[u] + v.second;                  // relax operation
         pq.push(ii(dist[v.first], v.first));
  }
 }
}
  for (int i = 0; i < V; i++)
    printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

  return 0;
}
```

Shortest path of Graph

Bellman Ford Algorithm

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

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, u, v, w;
  vector<vii> AdjList;
  scanf("%d %d %d", &V, &E, &s);

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

Shortest path of Graph

Bellman Ford Algorithm

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

bool hasNegativeCycle = false;
for (int u = 0; u < V; u++)                            // one more pass to check
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dist[v.first] > dist[u] + v.second)                  // should be false
      hasNegativeCycle = true;     // but if true, then negative cycle exists!
  }
printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

if (!hasNegativeCycle)
  for (int i = 0; i < V; i++)
    printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

return 0;
```

```
/*
 // Graph has negative weight, but no negative cycle
 5 5 0
 0 1 1
 0 2 10
 1 3 2
 2 3 -10
 3 4 3

 // Graph with negative cycle exists
 3 3 0
 0 1 1000
 1 2 15
 2 1 -42
 */
```