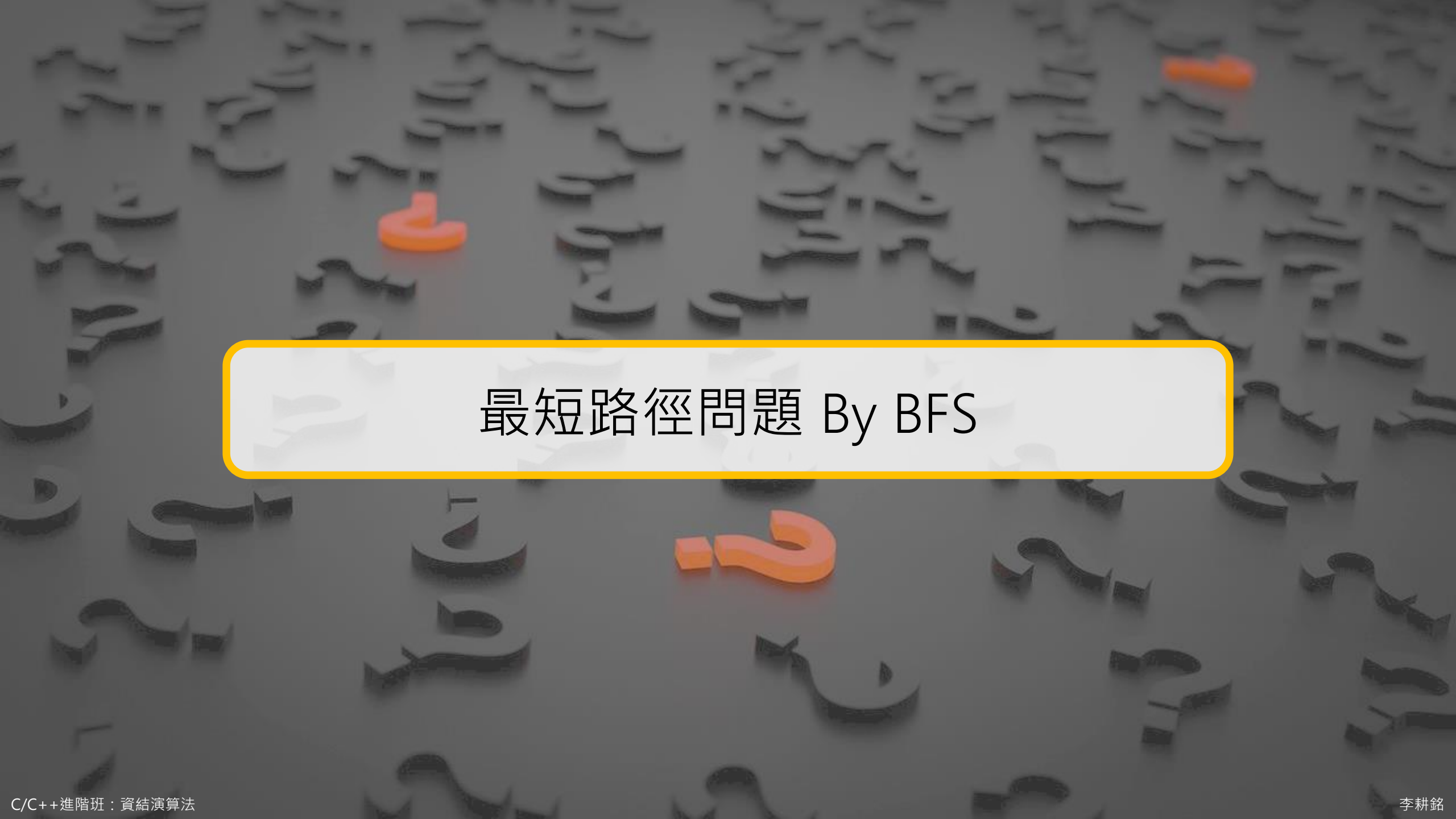
C/C++ 進階班
演算法

最短路徑
(Shortest Path)

李耕銘

課程大綱

- 最短路徑問題 By BFS
- 最短路徑問題簡介
- 最短路徑問題的基礎
- Bellman-Ford Algorithm
- DAG Algorithm
- Dijkstra's Algorithm
- Floyd-Warshall Algorithm
- 最短路徑總結

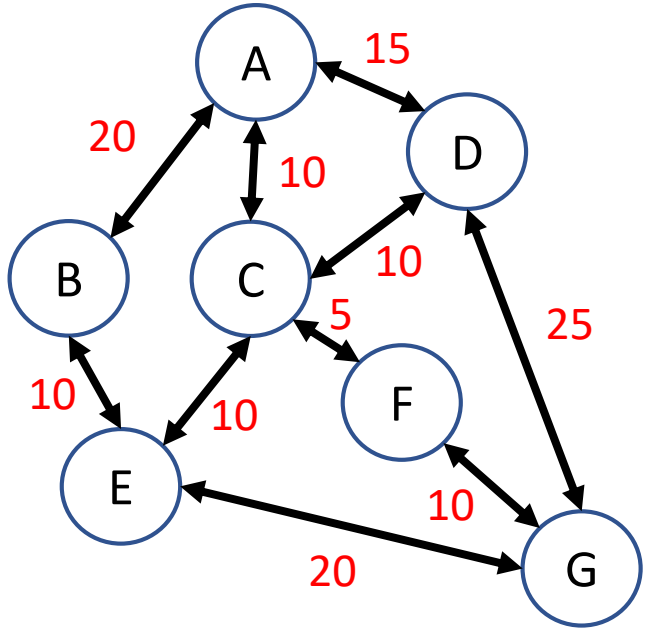


最短路徑問題 By BFS

最短路徑問題

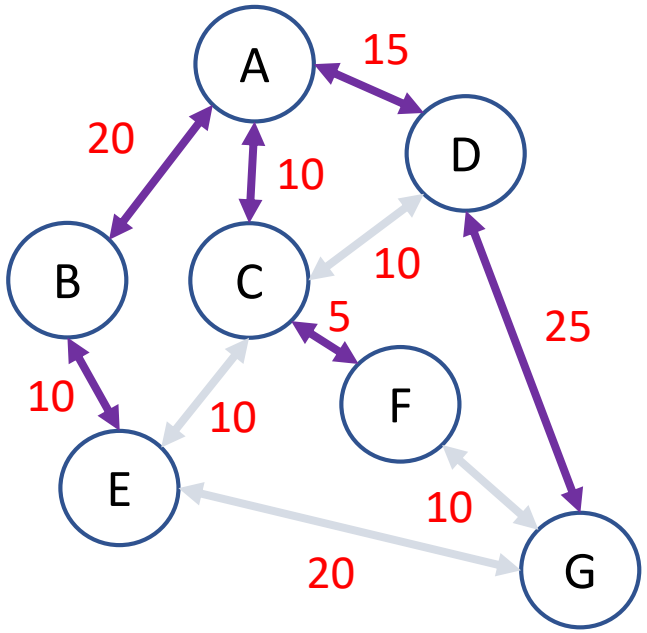
Mission

給定城市的數目與連接的道路，假設每一條道路所需要的交通時間不相等，試著找出從 A 到其他城市間的最短路徑。(Weighed)



為什麼不能用 BFS找最短路徑？

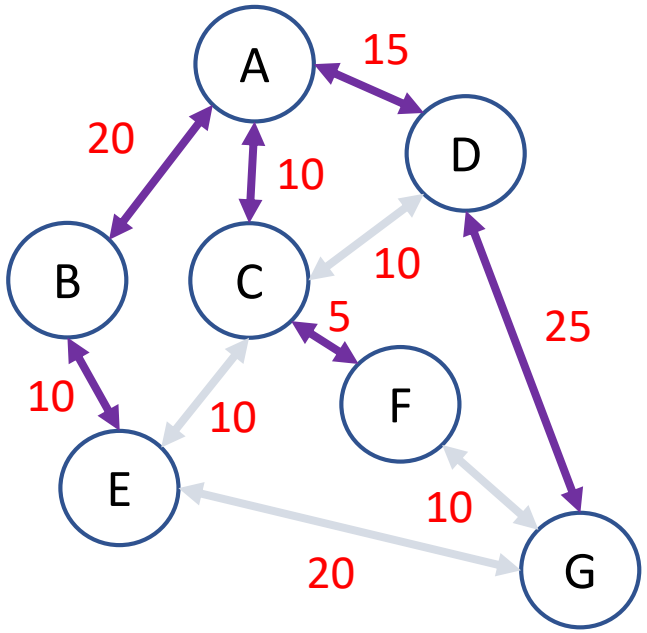
最短路徑問題



Vertex	Distance from vertex(A)
A	0
B	20
C	10
D	15
E	30
F	15
G	40

為什麼不能用 BFS找最短路徑？

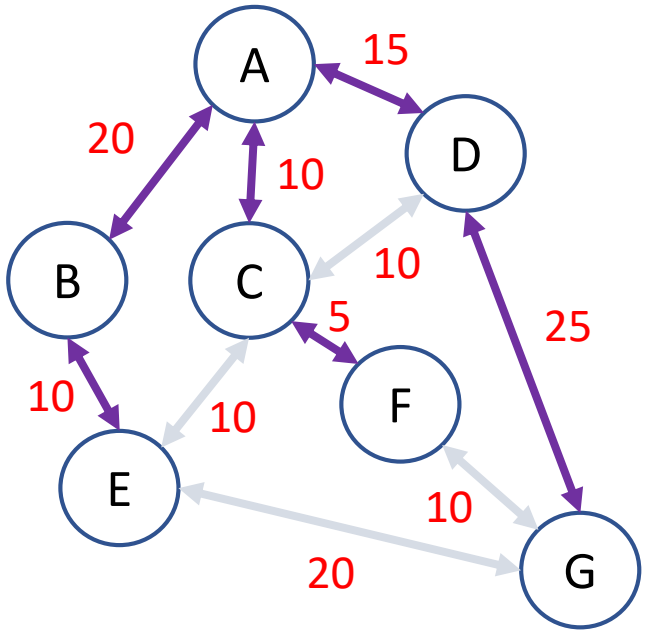
最短路徑問題



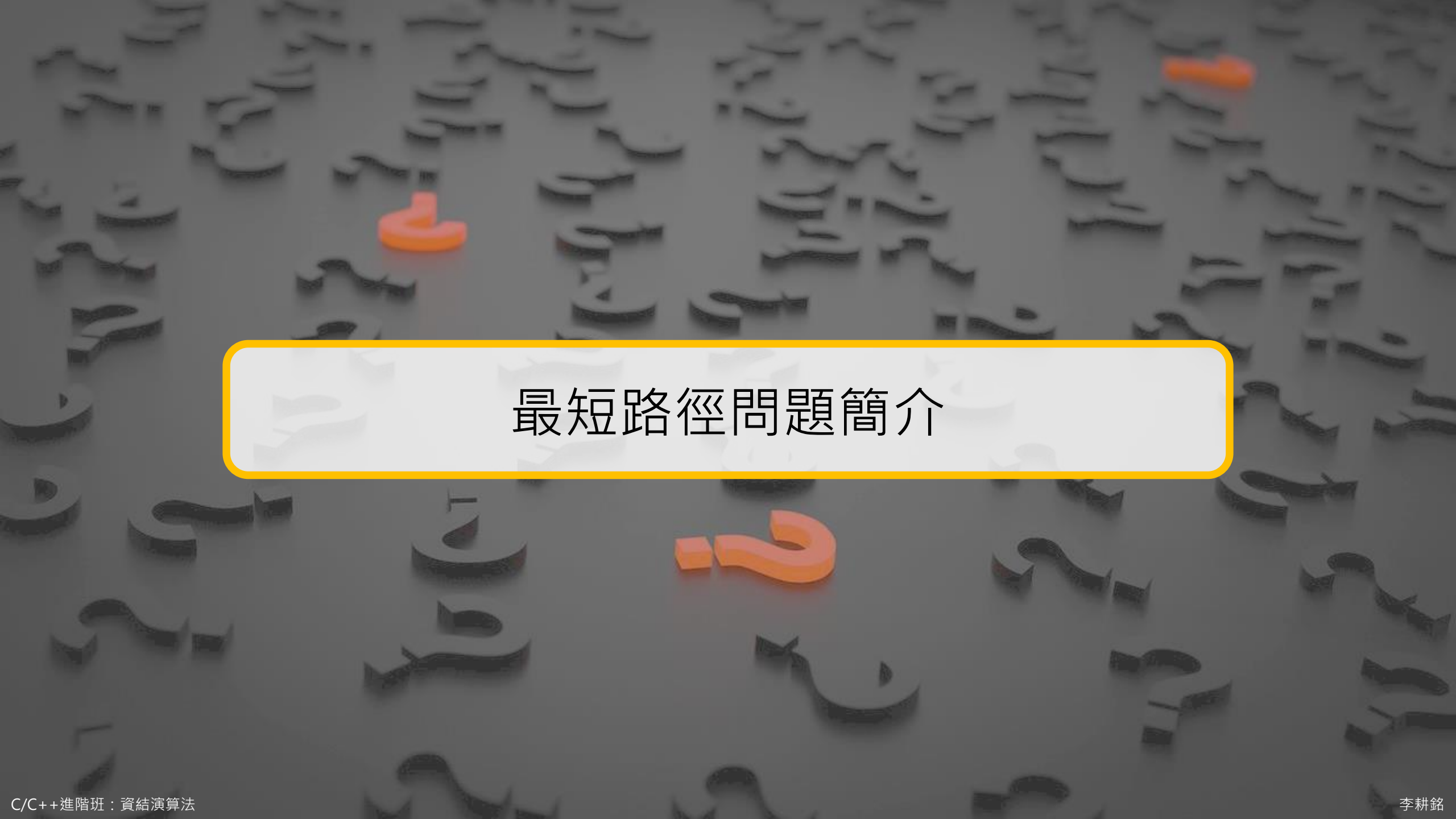
Vertex	Distance from vertex(A)
A	0
B	20
C	10
D	15
E	**20**
F	15
G	**25**

為什麼不能用 BFS找最短路徑？

最短路徑問題

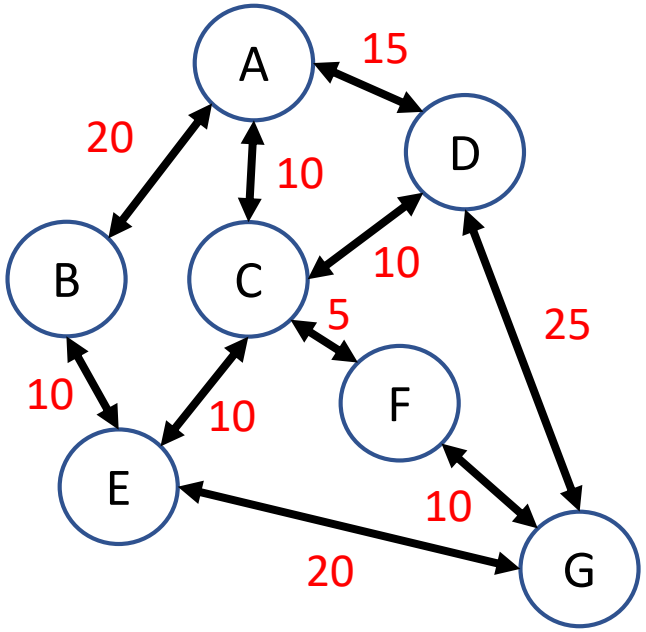


- BFS 尋找的最短路徑是經過最少的 Edge
- 跑遍所有 Vertex，但不跑遍所有 Edge
- 只適用於 Unweighted Edge
- 不適用於 Weighted Edge



最短路徑問題簡介

最短路徑問題

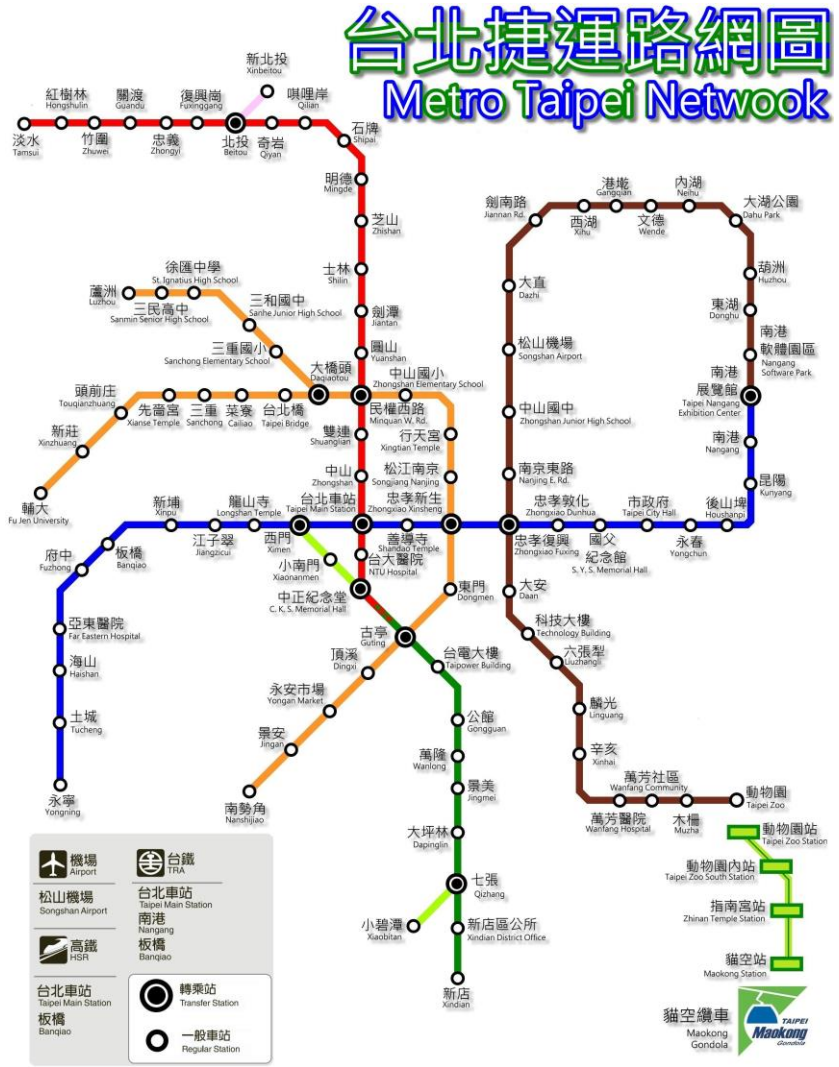


在圖上找出兩頂點間的最短路徑：

- 路徑 $p_{(v_1,v_n)} = (v_1, e_1, v_2, e_2, v_3 \cdots\cdots, e_{n-1}, v_n)$
- 邊 e_i 的權重 $= w(e_i)$
- 路徑 p 的權重 $w(p_{(v_1,v_n)}) = \sum_{i=1}^{n-1} w(e_i)$
- $p_{(v_1,v_n)}$ 的最短距離 $\delta(v_1, v_n)$：

$$\delta(v_1, v_n) = \begin{cases} min\left(w(p_{(v_1,v_n)}) : v_1 \xrightarrow{p_{(v_1,v_n)}} v_n\right), \exists\ path\ from\ v_1\ to\ v_n \\ \infty; otherwise \end{cases}$$

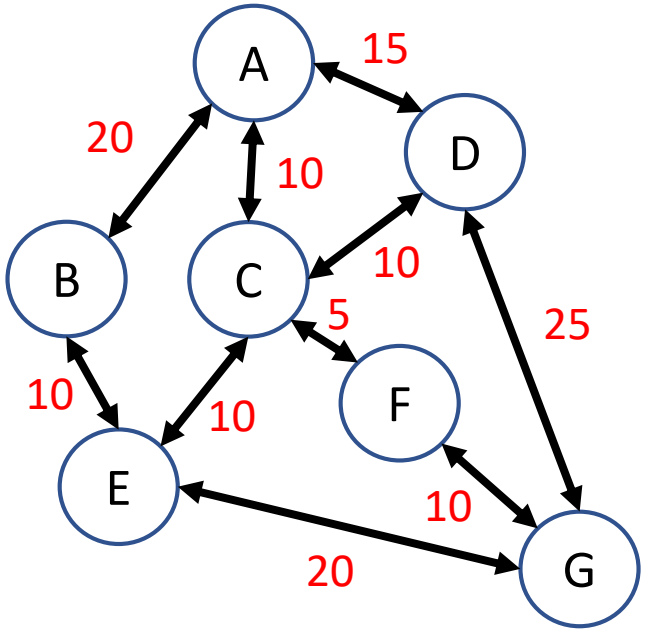
最短路徑問題



最短路徑的應用

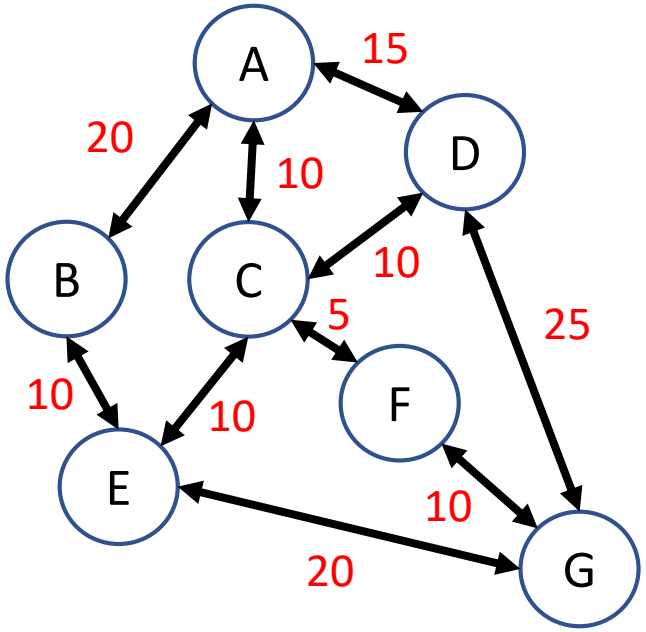
- 修路問題
- 瓦斯管線
- 交通時間
- 郵差寄信
- 瓦斯抄錶.....

最短路徑問題



- Weight 的和便是該路徑的成本
- Undirected Edge 可以視作兩條對向的 Directed Edege
  - Undirected Graph 能當作 Directed Graph 解決
  - 反之無法
- Unweighted Edge 可以視作所有 Edge 擁有相同的 Weight
  - Unweighted Graph 能當作 Weighted Graph 解決
  - 反之無法
- 聚焦在解決 Directed and Weighted Graph 上

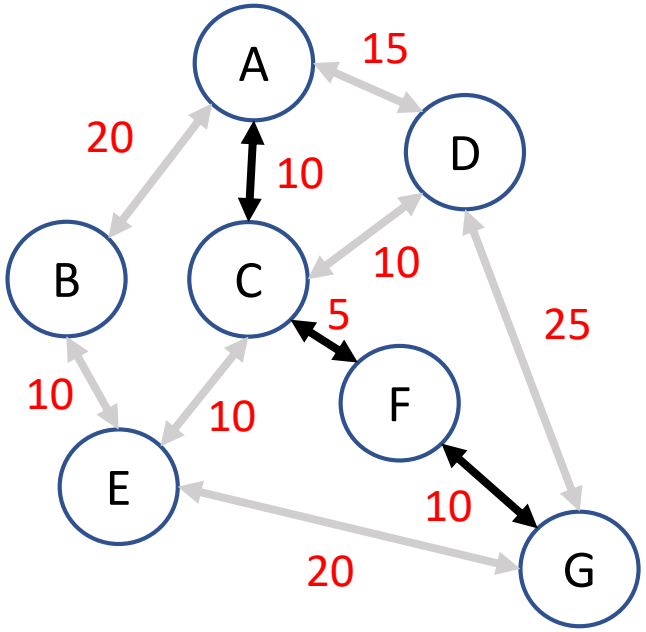
最短路徑問題



最短路徑問題的分類

1. Single-Source Shortest Path
   - 從單一頂點到其餘所有頂點的最短路徑
2. Single-Pair Shortest Path
   - 從單一頂點到單一頂點的最短路徑
   - Single-Source Shortest Path 的子問題
3. Single-Destination Shortest Path
   - 從所有頂點到特定頂點的最短路徑
   - 把 Edge 方向倒過來來，即是 Single-Pair Shortest Path
4. All-Pairs Shortest Path
   - 所有頂點到其餘所有頂點的最短路徑
   - 把所有頂點用 Single-Pair Shortest Path 算過一次即是

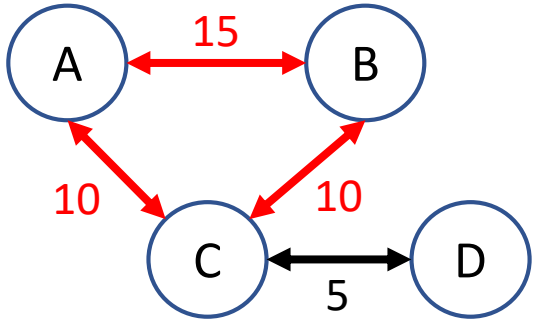
最短路徑問題

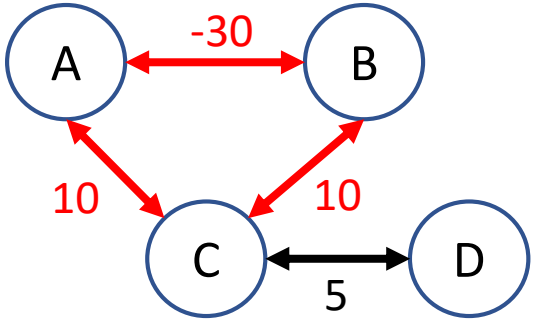


v_1, v_n 間的最短路徑 $p_{(v_1,v_n)}$ 必不存在cycle

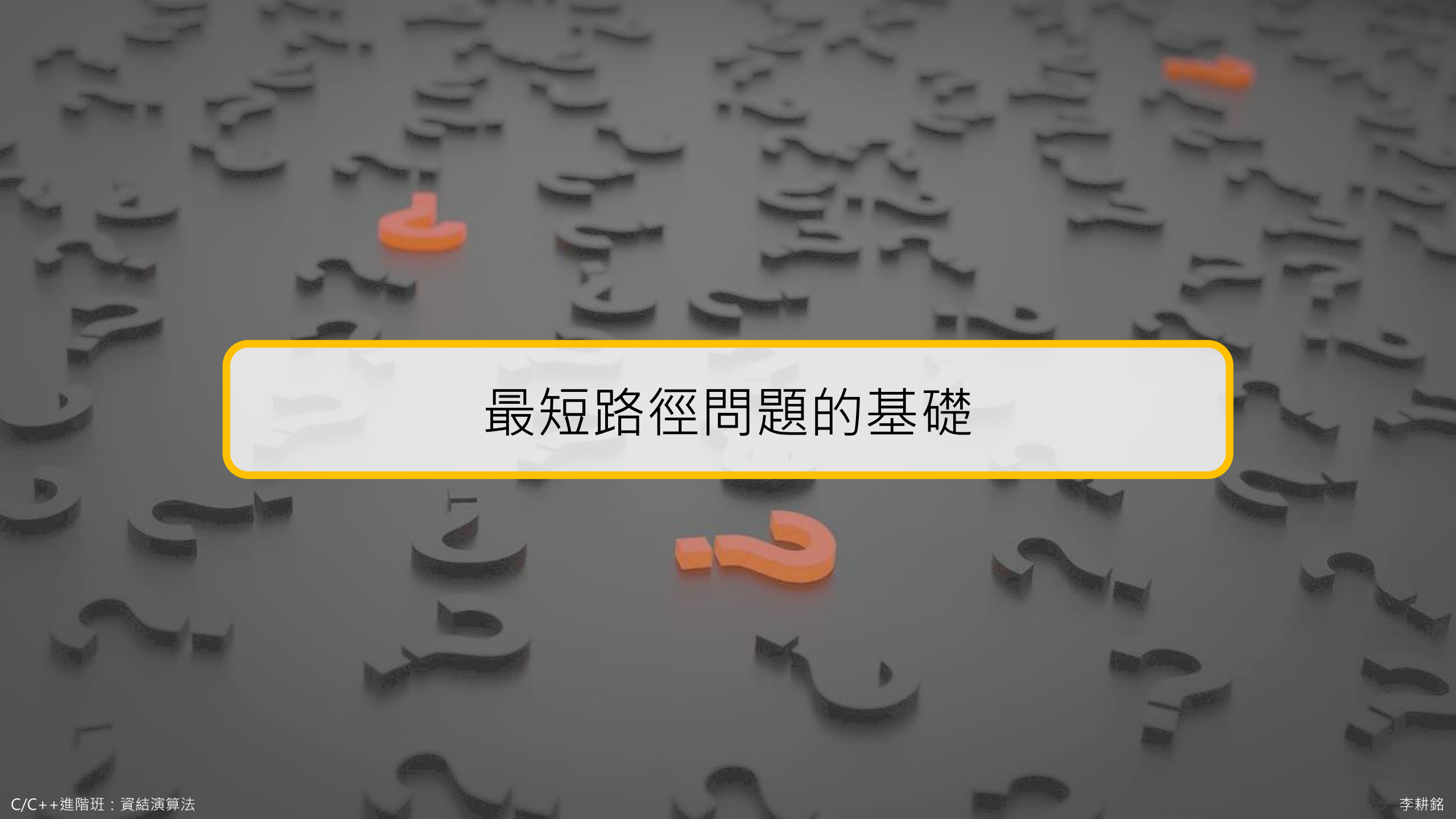
- 會是一棵 Shortest-Path Tree
- 起點 v_1 即為根節點
- 最多只會有 $|V| - 1$ 條 Edge
  - ✓ 即經過所有頂點一次

最短路徑問題



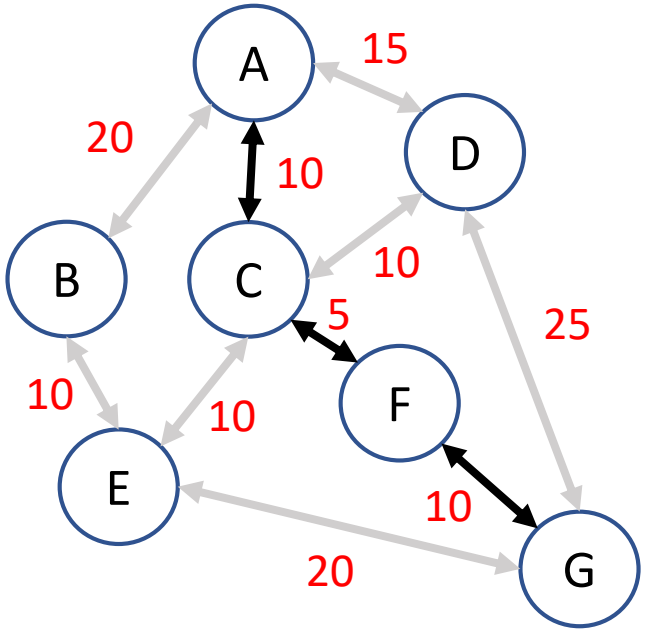


- 最短路徑裡一定不存在 cycle嗎？
  - ➢ Cycle 的權重總和為正
    - ✓ 多個 Cycle 成本必定增加
  - ➢ Cycle 的權重總和為負
    - ✓ 可無限制走該 Cycle
    - ✓ $\delta(v_1, v_n) = -\infty$，不存在唯一路徑



最短路徑問題的基礎

最短路徑問題



處理最短路徑問題

- Distance
  - 記錄某頂點到其餘頂點路徑上的權重總和
- Predecessor
  - 目前行該頂點成本最低的方向

最短路徑問題

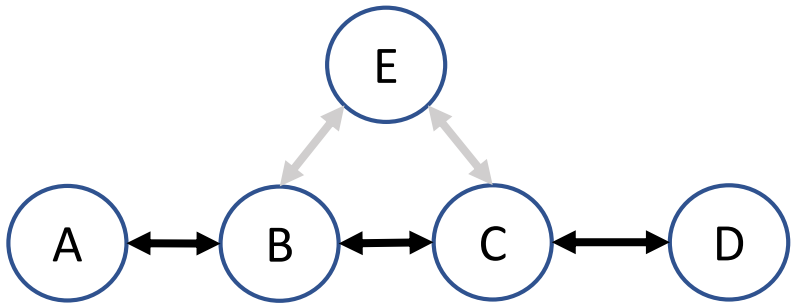
若 v_1, v_n 間存在最短路徑路徑 C ，則此路徑上的所有子路徑皆為最短

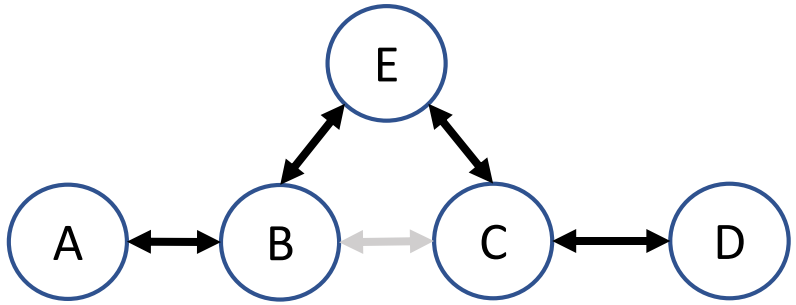
- ✓ 若 $0 \leq i \leq j \leq n$
- ✓ $(v_i, e_i, v_{i+1}, e_{i+1}, \ldots\ldots, e_{j-1}, v_j)$ 亦為 v_i, v_j 間的最短路徑 $p_{(v_i, v_j)}$

➢ 最短路徑必由所有經過頂點間的最小路徑所組成

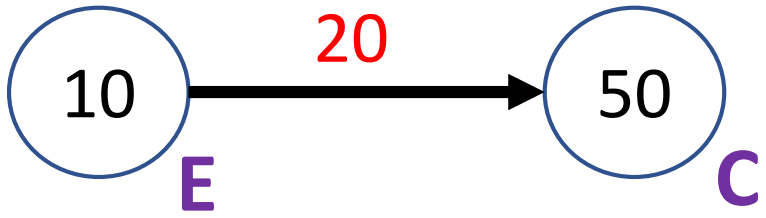
➢ 如圖

1. 若 B-C 間的成本 > B-C-E：採用 B-E-C
2. 若 B-C 間的成本 < B-C-E：採用 B-C





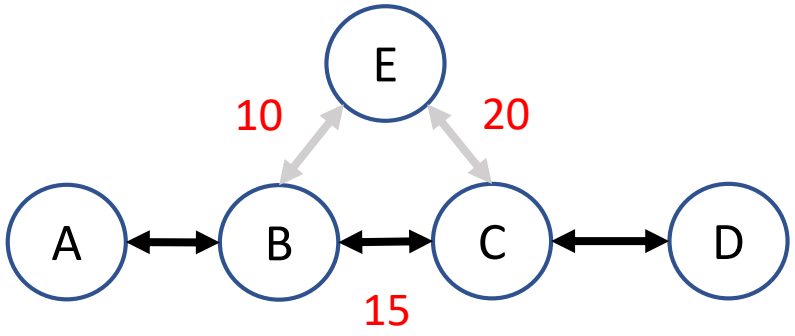
最短路徑問題

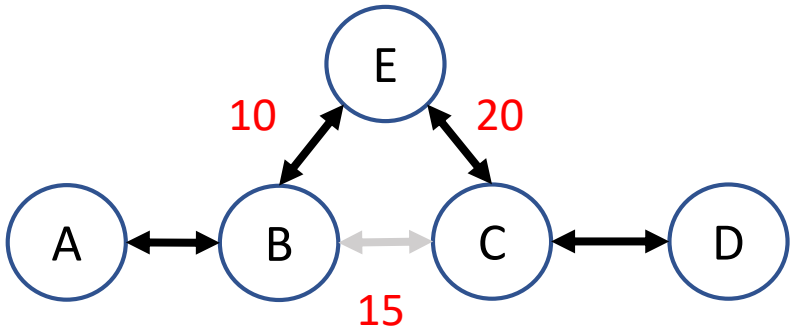


- Relaxation
  - 比較兩不同路徑的成本後再更新就會得到較短路徑
    1. C：50
    2. E-C：10+20=30
    3. 之後都以 E-C 這條路徑走到 C

```
Relax(E, C, weight){
    if (Distance[C] > distance[E] + weight of E-C)
{
        Distance[C] = distance[E] + weight of E-C;
        Predecessor[C] = E;
    }
}
```

最短路徑問題

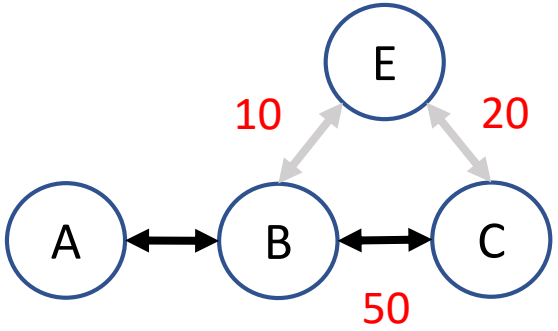


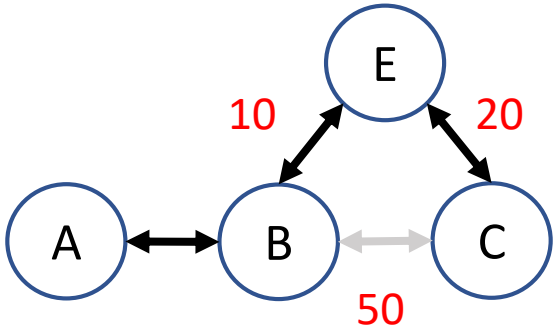


- Triangle inequality
  - $\delta(v_1, v_n) \leq \delta(v_1, v_x) + w(v_x, v_n)$
  - $\delta(v_B, v_C) \leq \delta(v_B, v_E) + w(v_E, v_C)$
- Upper-Bound property
  - 若 $p_{(v_1,v_n)}$ 已是 v_1, v_n 間的最短路徑
  - $p_{(v_1,v_n)}$ 無須再作任何更動

最短路徑問題

- Convergence property
  - 如果 A-C 的最短路徑包含 (A, B) 且 d[B]=δ(A,B)，則此時對 B→C的所有路徑進行 Relax 會讓 distance[C] = δ(A,C)
  - Ex: 已知高雄到紐約的最短路徑必會包括：
    - ✓ 左營高鐵→桃園機場
    - ✓ 則對桃園機場→紐約進行 Relax 後會使該路徑為最短





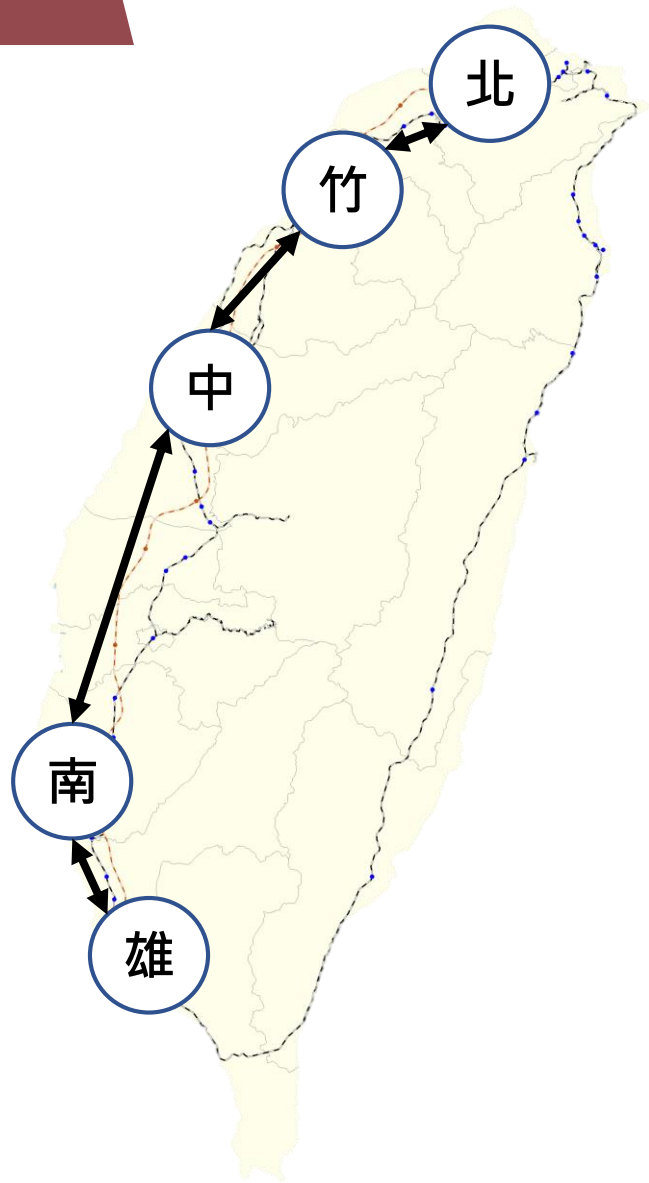
最短路徑問題

- Path-relaxation

➢ 若 $p_{(v_1,v_n)} = (v_1, v_2, \ldots\ldots, v_n)$ 是 v_1, v_n 間的最短路徑， 則依序執行 $Relax(v_1, v_2, w_1)$、$Relax(v_2, v_3, w_2)$、...、$Relax(v_{n-1}, v_n, w_{n-1})$ 最後便會得到最短路徑。

➢ 已知台北騎機車到高雄的中間必會經過新竹、台中、台南

- ✓ 對台北→新竹、新竹→台中、台中到台南、台南到高雄 Relax
- ✓ 便會得到台北→高雄的最短路徑



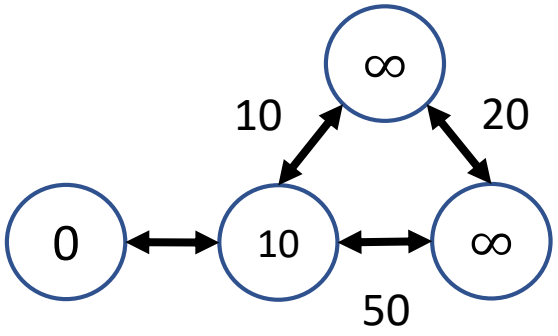
Bellman-Ford Algorithm

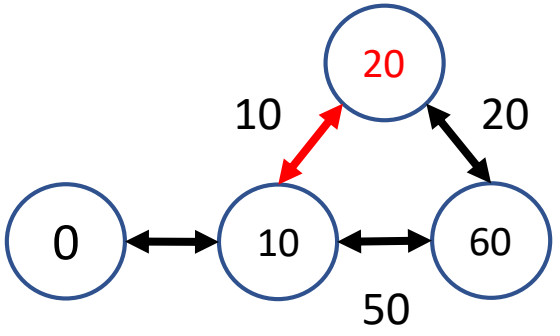
Bellman-Ford Algorithm

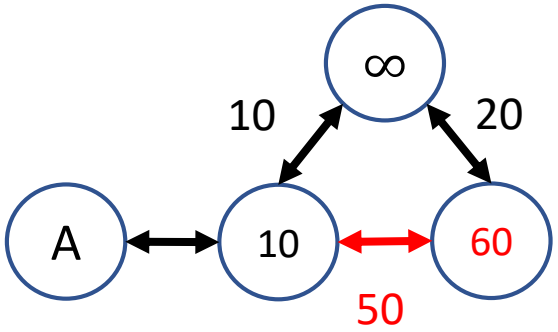
Bellman-Ford Algorithm

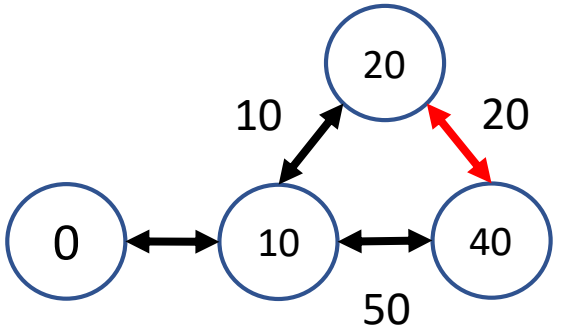
```
1  Bellman-Ford(G,w,s){
2        initialize (G,s)
3        for i = 1 to |V-1|
4              for edge (u,v)∈E
5                    Relax(u,v,w)
6        for edge(u,v) in Graph
7              if d[v]>d[u]+w(u,v)
8                    return false
9        return true
```

1. 依序對所有 Edge 進行 Relax
2. 但必須執行 |V| − 1 次迴圈，以確保最短路徑
3. 最後須檢查沒有負迴圈！

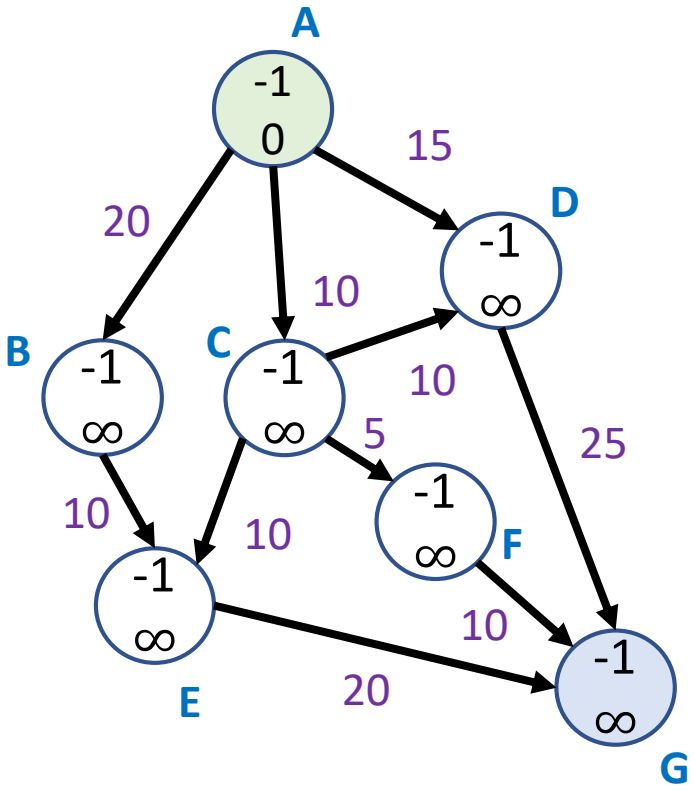






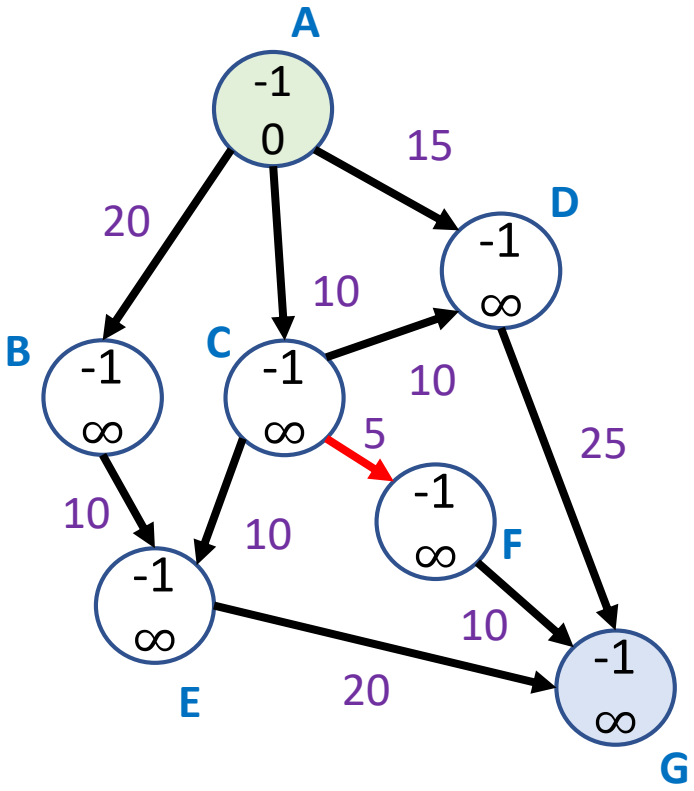


Bellman-Ford Algorithm



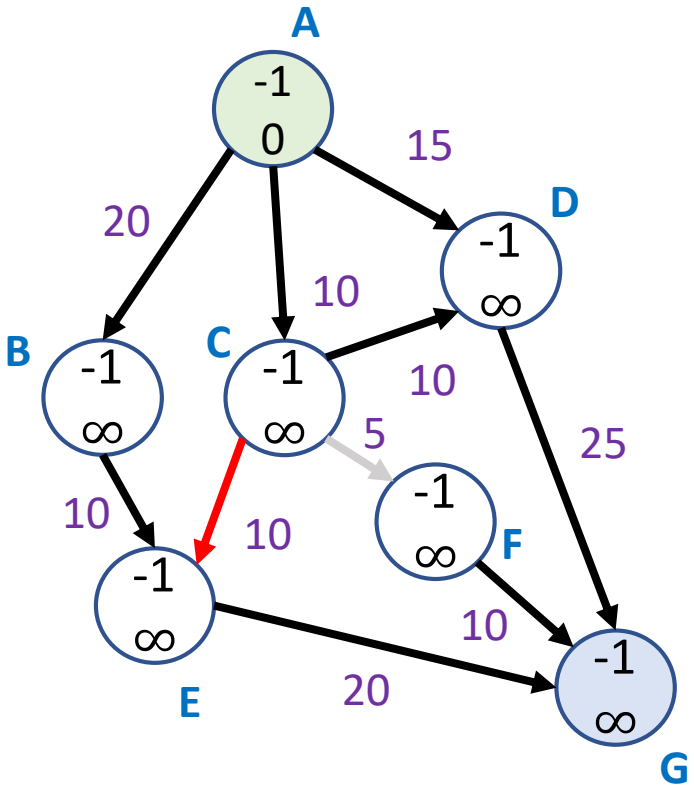
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



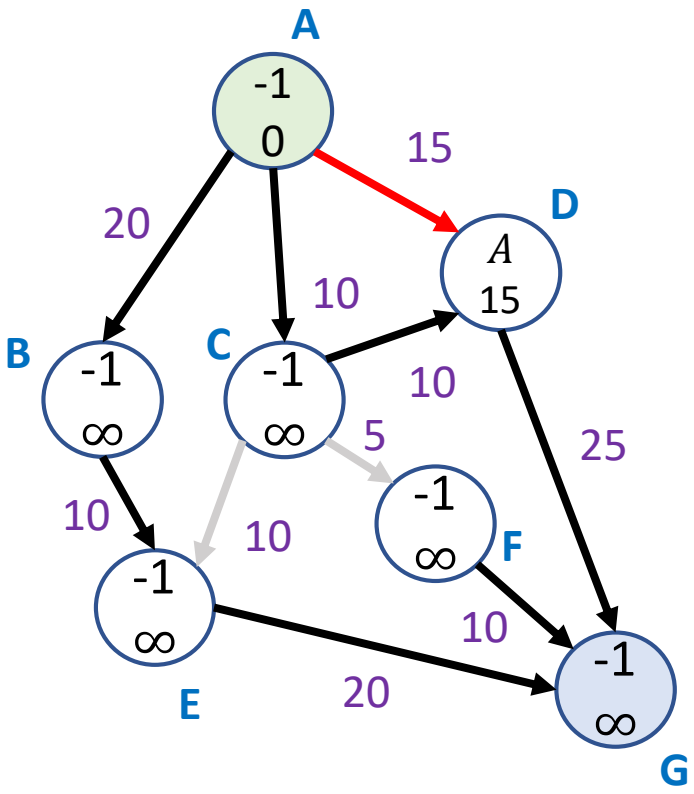
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



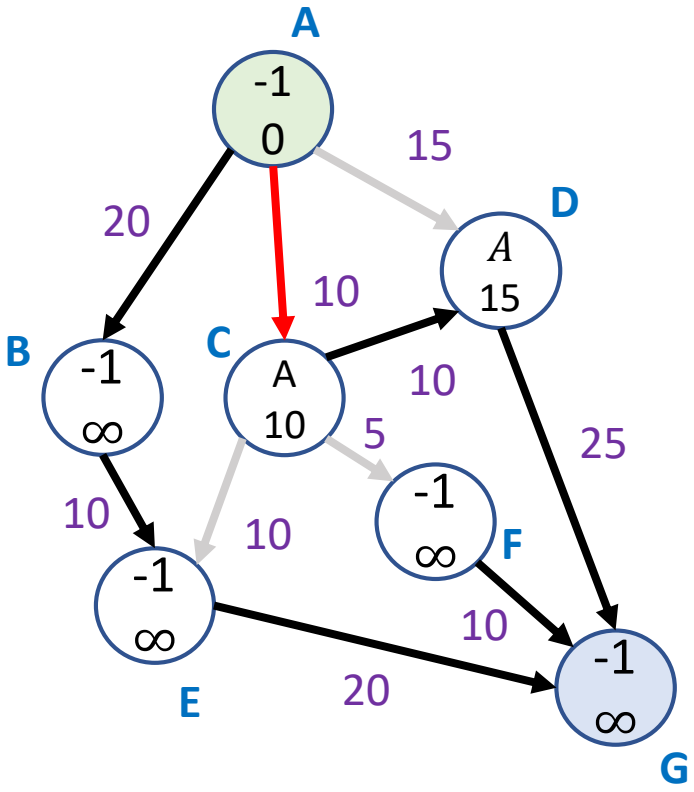
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



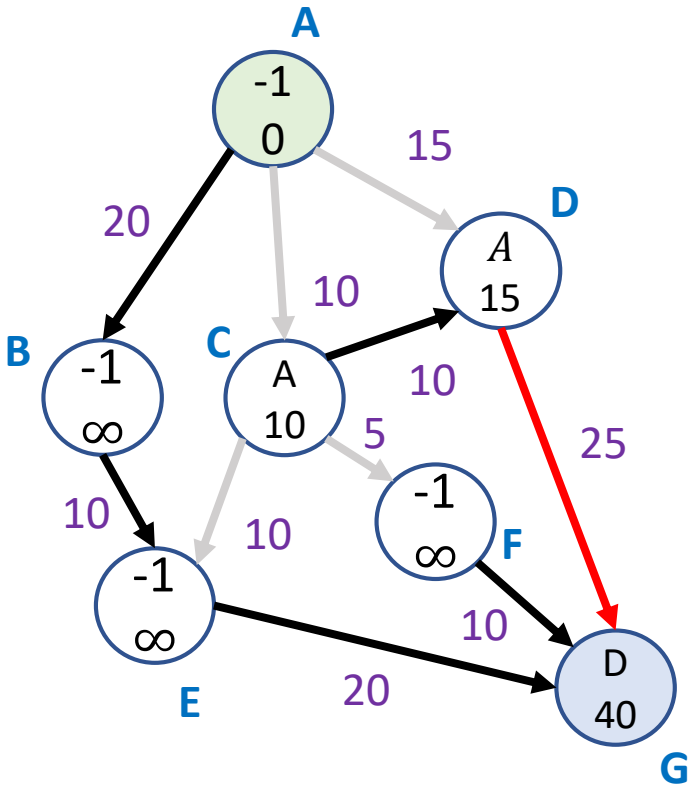
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



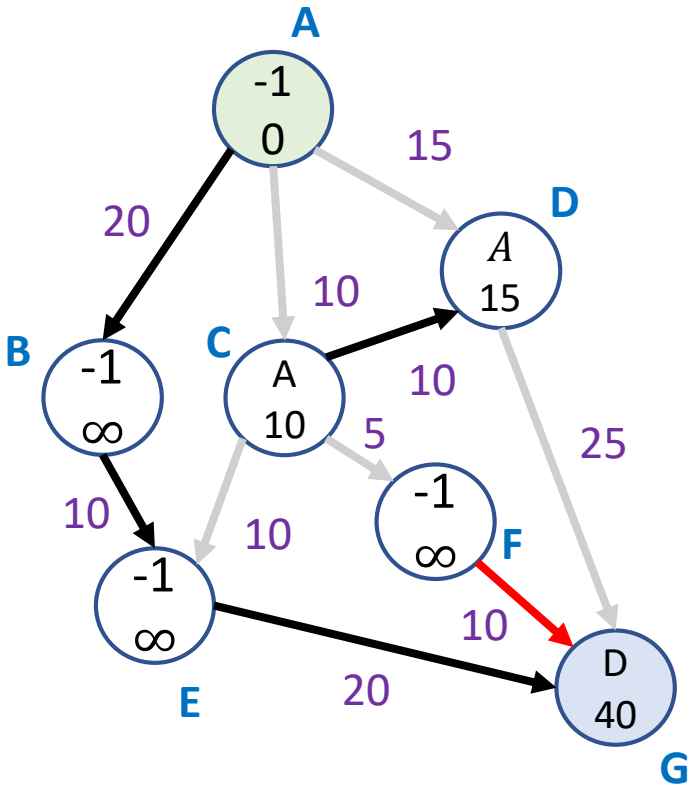
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



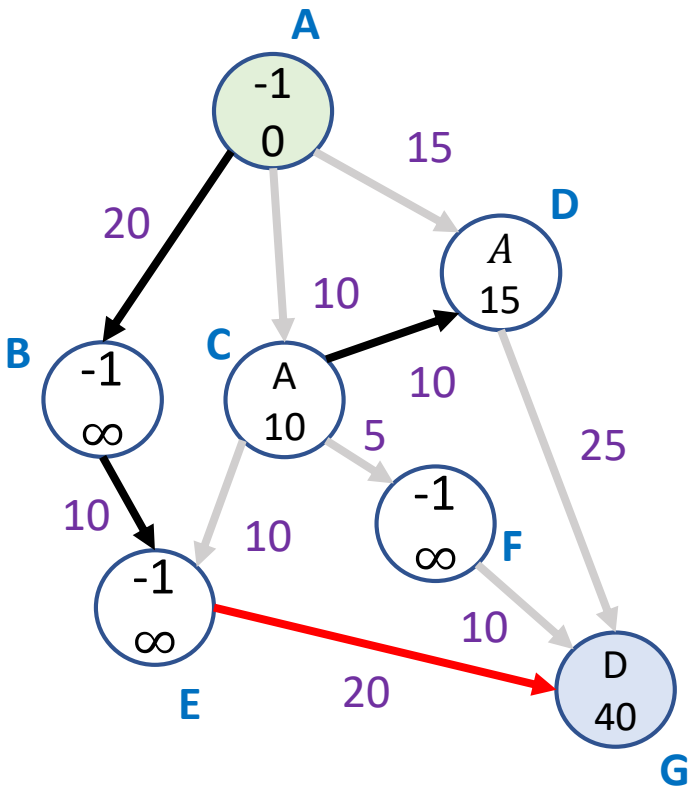
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



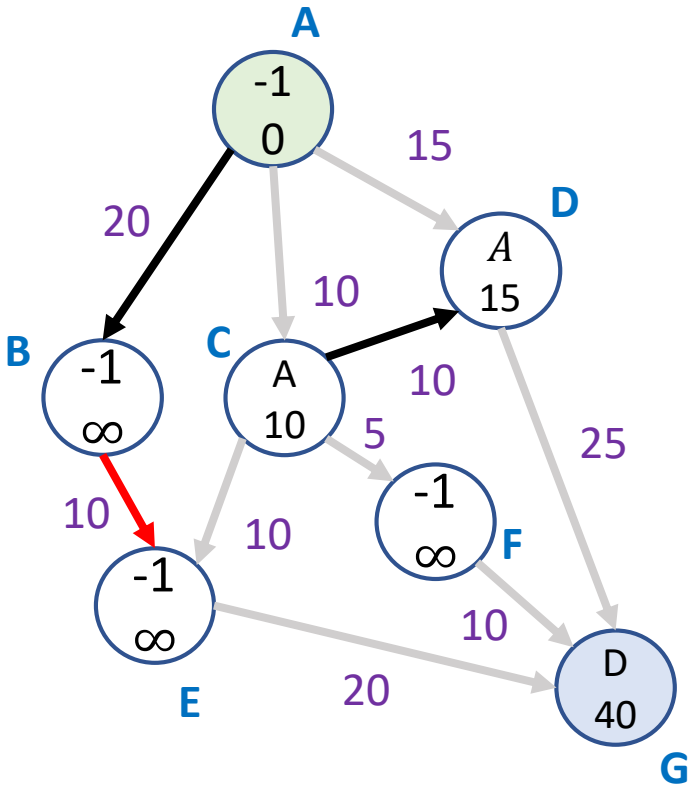
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



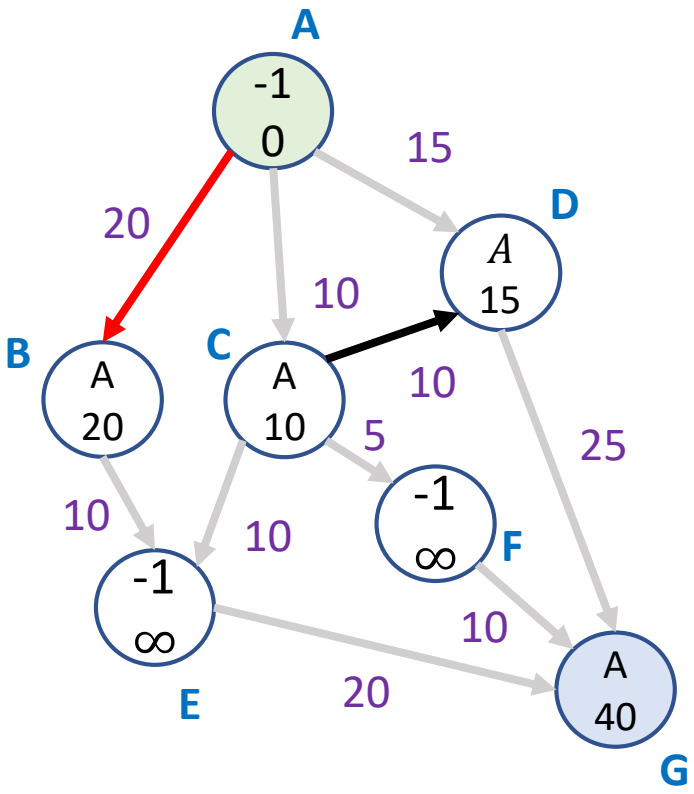
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



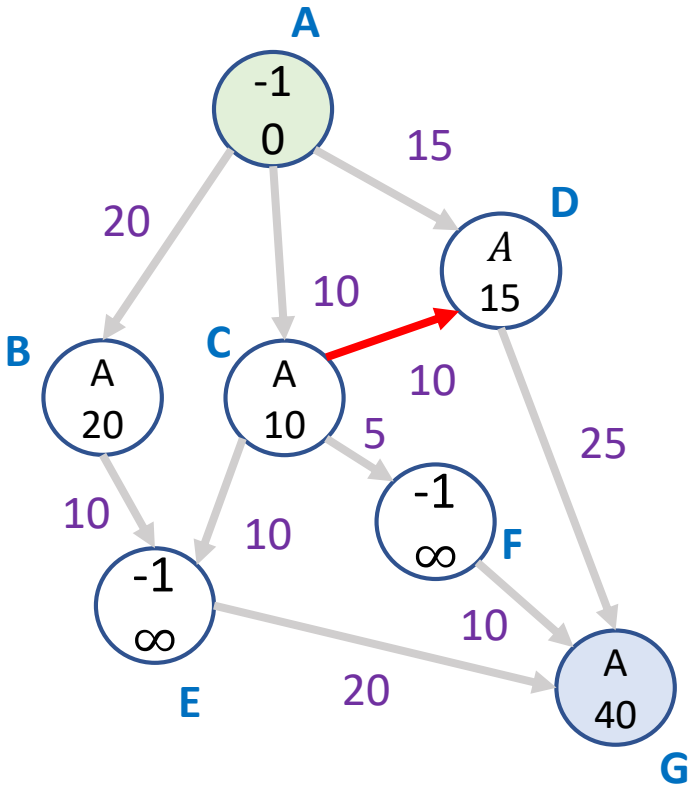
1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm



1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm

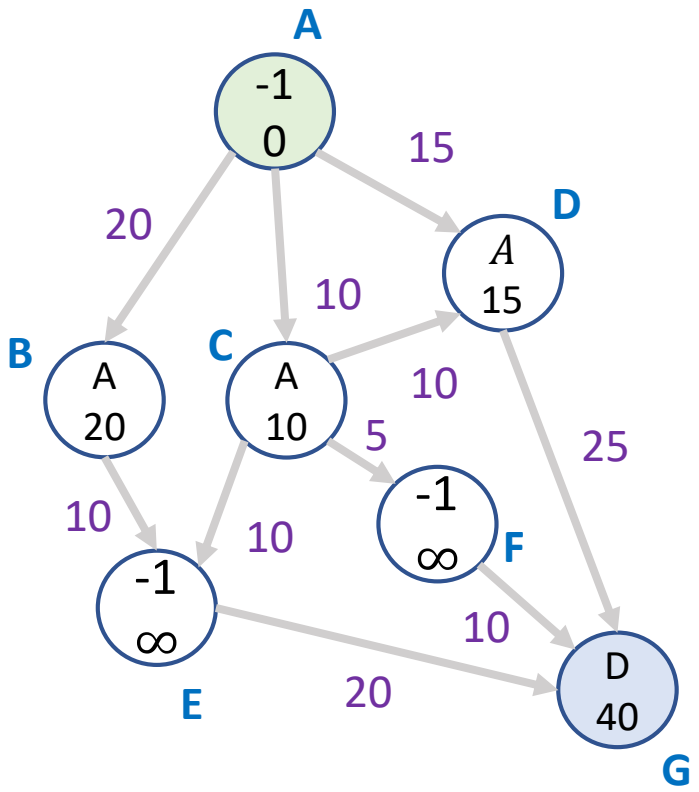


1. 初始化所有 Vertex
   - ✓ Predecessor 設為 -1，表目前沒有 Predecessor
   - ✓ Distance 設為 ∞，起點的 Distance 設成 0
2. 對 10 個 Edge 進行 Relax
3. 重複執行上一步驟 6 (|V| − 1) 次，以確保最短路徑
4. 最後檢查沒有負迴圈！

Bellman-Ford Algorithm

為什麼要進行 |V|-1 次？

因為Edge relax的過程並沒有依照最短路徑的順序！

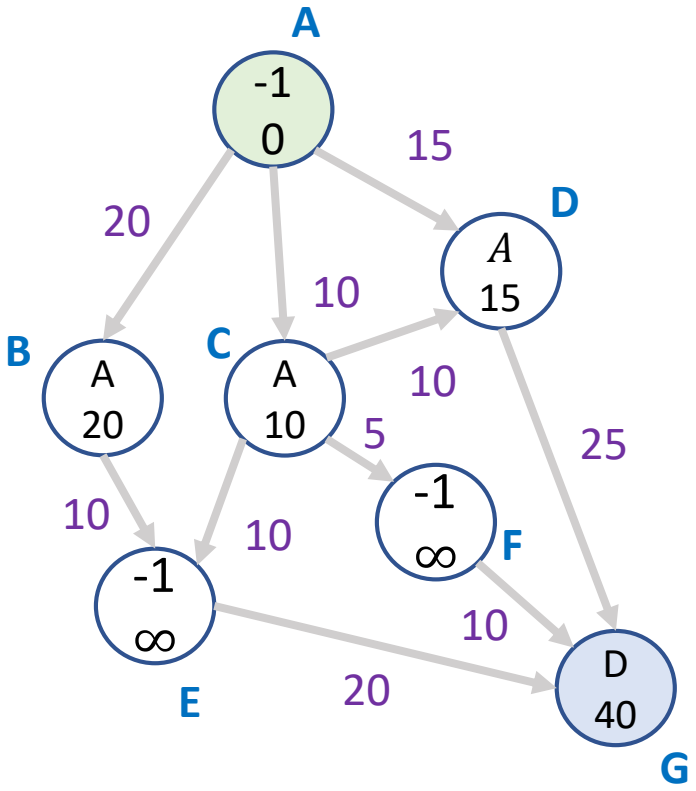


Bellman-Ford Algorithm

||

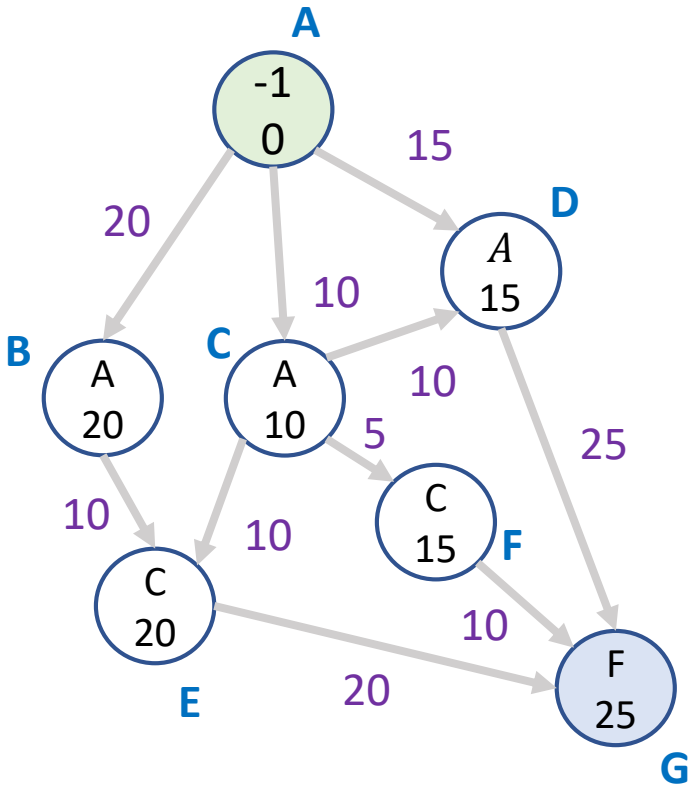
暴力法

Bellman-Ford Algorithm



Relax Order	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Iteration #1	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #2	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #3	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #4	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #5	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #6	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)

Bellman-Ford Algorithm



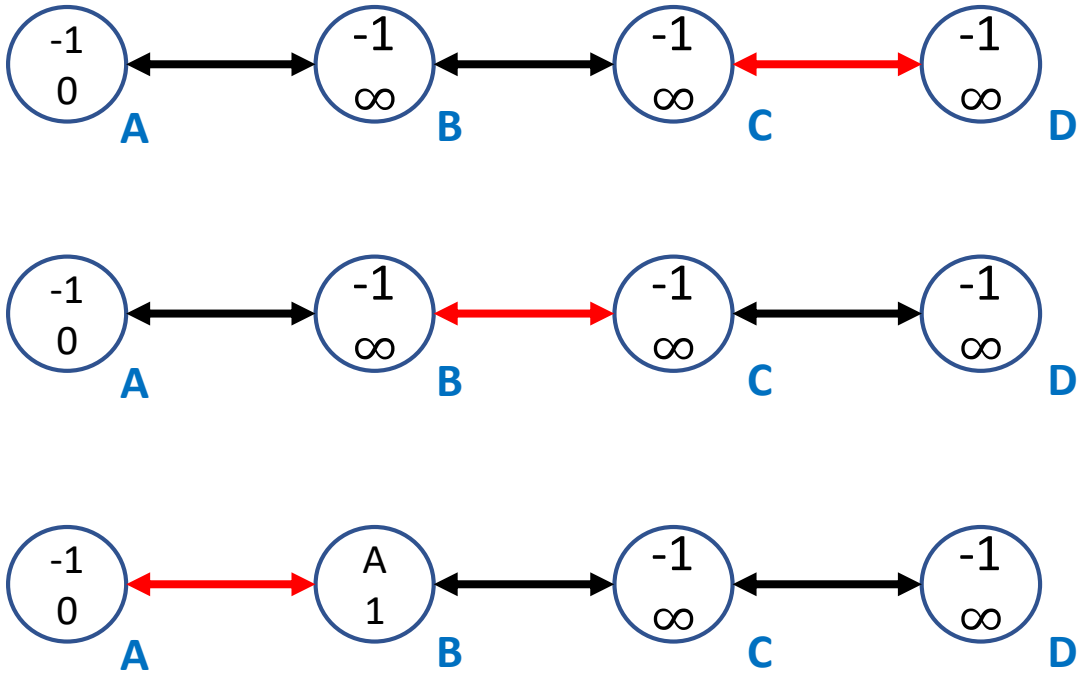
Relax Order	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Iteration #1	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #2	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #3	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #4	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #5	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)
Iteration #6	e(C,F)	e(C,E)	e(A,D)	e(A,C)	e(D,G)	e(F,G)	e(E,G)	e(B,E)	e(A,B)	e(C,D)

Bellman-Ford Algorithm

Worst Case：

- Relax 順序剛好跟最短路徑的順序相反
- 每次 Relax 後只能優化單一子路徑
- 最多要經過 |V| 個頂點，需要有 |V|-1 條子路徑
- 至少需要重複 |V|-1 次

<u>若改善 Relax 的 Edge 順序就能夠大幅優化複雜度</u>



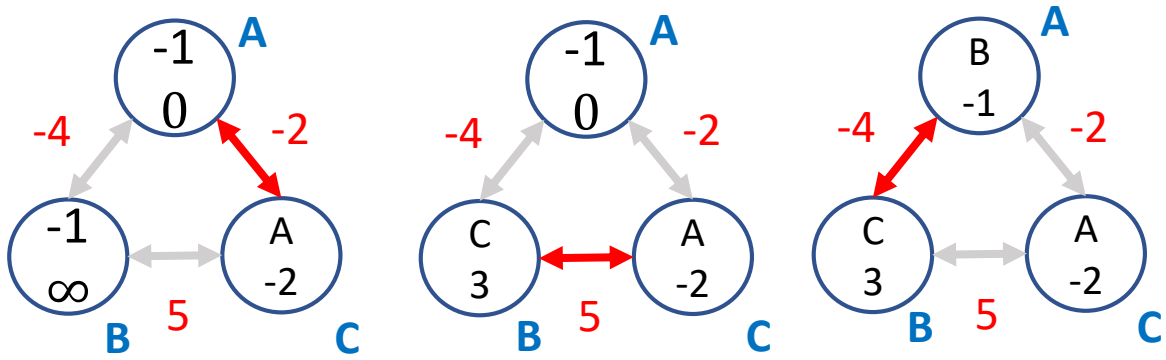
Bellman-Ford Algorithm

Bellman-Ford Algorithm

```
1  Bellman-Ford(G,w,s){
2      initialize (G,s)
3      for i = 1 to |V-1|
4          for edge (u,v)∈E
5              Relax(u,v,w)
6      for edge(u,v) in Graph
7          if d[v]>d[u]+w(u,v)
8              return false
9      return true
10 }
```

負迴圈的檢查

- 最後所有的 Edge 應該要滿足 $d[v] \leqq d[u]+w(u,v)$
- 若出現 $d[v] > d[u]+w(u,v)$，代表出現負迴圈！



d[C] > d[A]+w(A,C)

Bellman-Ford Algorithm

Bellman-Ford Algorithm

```
1  Bellman-Ford(G,w,s){
2        initialize (G,s)
3        for i = 1 to |V-1|
4              for edge (u,v)∈E
5                    Relax(u,v,w)
6        for edge(u,v) in Graph
7              if d[v]>d[u]+w(u,v)
8                    return false
9        return true
10 }
```

O((|V|-1)*|E|)

O(|E|)

效能分析

- 初始化：O(|V|)
- 對所有邊進行 Relax： O(|E|)
- 重複 |V|-1 次： O(|V-1||E|)
- 總和：O(|V||E|)

Example Code

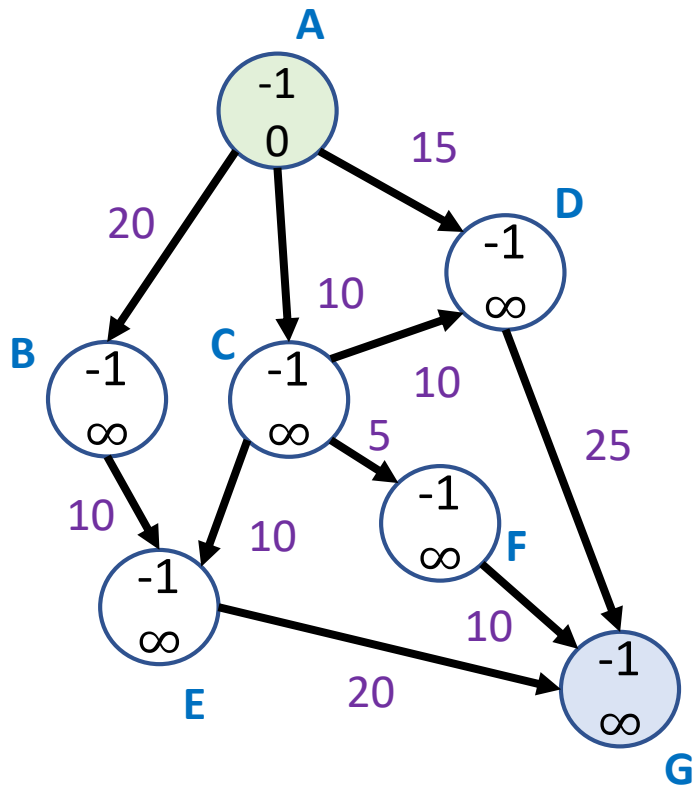
Mission

完成 Relax 函式

Practice 1

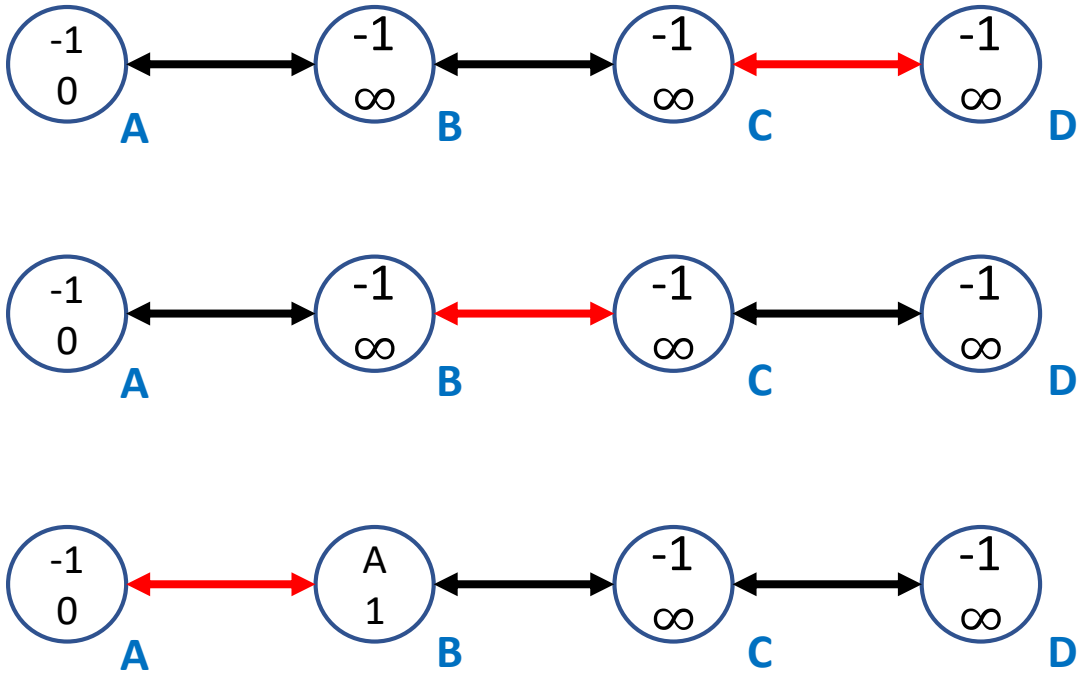
Mission

完成 Bellman-Ford 演算法



SPFA

- Bellman-Ford
  - 浪費時間更新圖上的 ∞ 或沒變過的地方
- SPFA(Shortest Path Faster Algorithm)
  - 每次只 Relax 那些有更新過的點



SPFA

- SPFA(Shortest Path Faster Algorithm)
  - 每次只 Relax 那些有更新過的點
  - 步驟
    1. 把起點 Push 到 Queue
    2. 從 Queue 裡 Pop 出一筆資料
    3. 該筆資料的所有邊進行 Relax
    4. 有更新到的頂點再 Push 到 Queue
    5. 重複步驟 2 ~ 4，直到 Queue 為空
  - 但需要檢查負環

SPFA Algorithm

```
1  SPFA(G,w,s){
2      initialize (G,s)
3      push s to Queue(Q)
4      while Q is not empty:
5          u = Q.pop()
6          for each edge in Adj[u]:
7              if (Relax(u,v,w))
8                  update[v] += 1
9                  if update[v] == |V|
10                     return false
11                 push v into Q
12     return true
13 }
```

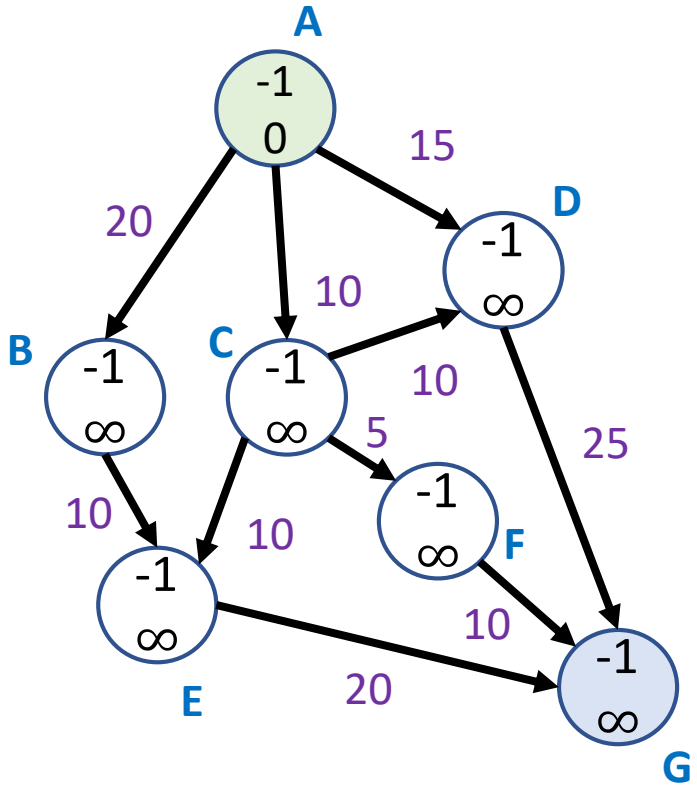
SPFA

- Bellman-Ford
  - 固定迴圈次數為 |V|-1 次
- SPFA(Shortest Path Faster Algorithm)
  - 有負環時，會出現無窮迴圈
  - 如果某點 Relax 次數 ≧ |V|，表有負迴圈
  - 複雜度：O(|V-1||E|)

SPFA Algorithm

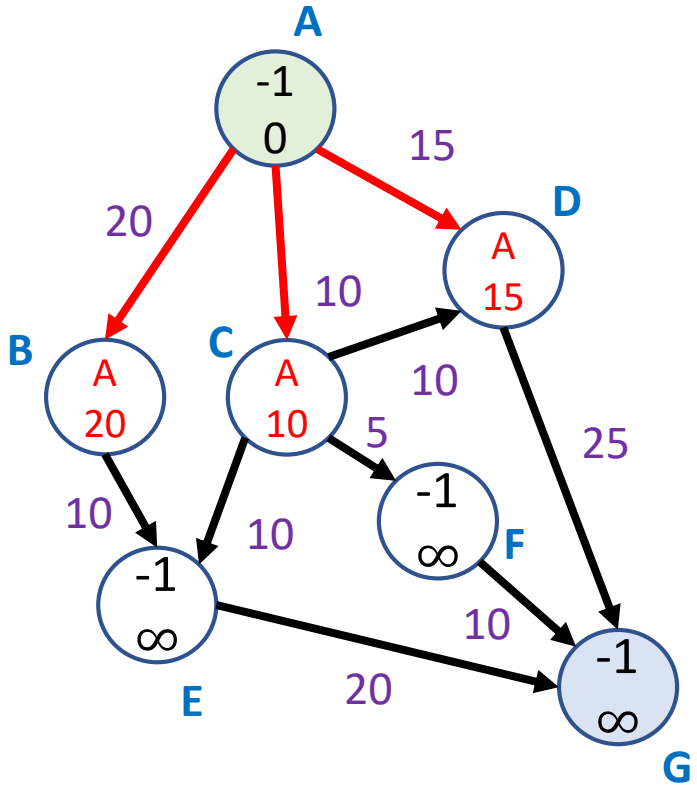
```
1  SPFA(G,w,s){
2      initialize (G,s)
3      push s to Queue(Q)
4      while Q is not empty:
5          u = Q.pop()
6          for each edge in Adj[u]:
7              if (Relax(u,v,w))
8                  update[v] += 1
9                  if update[v] == |V|
10                     return false
11                 push v into Q
12     return true
13 }
```

SPFA



Queue = {A}

SPFA

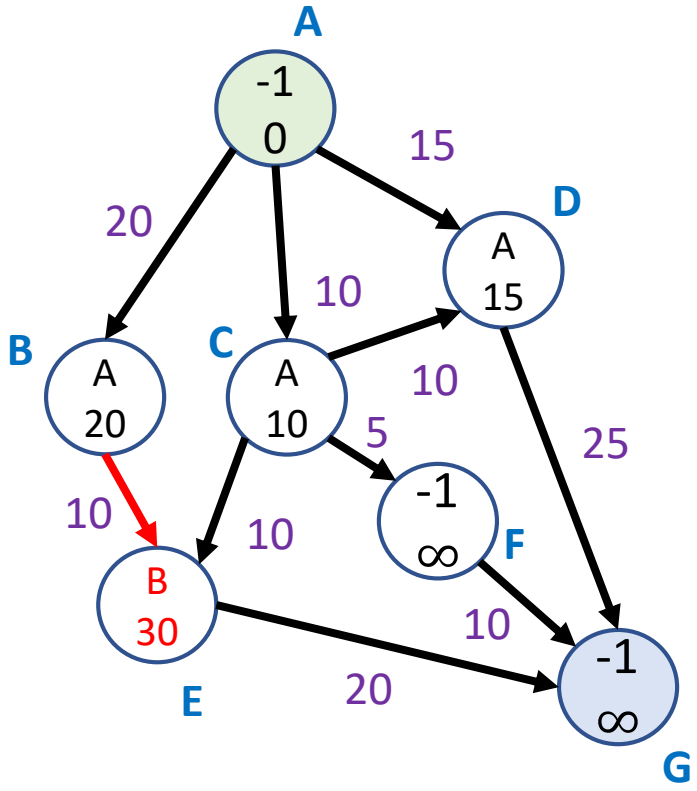


Queue = {A}

1. 對 A 的所有出邊做 Relax
2. 把有更新的 BCD 放入 Queue

Queue = {B,C,D}

SPFA

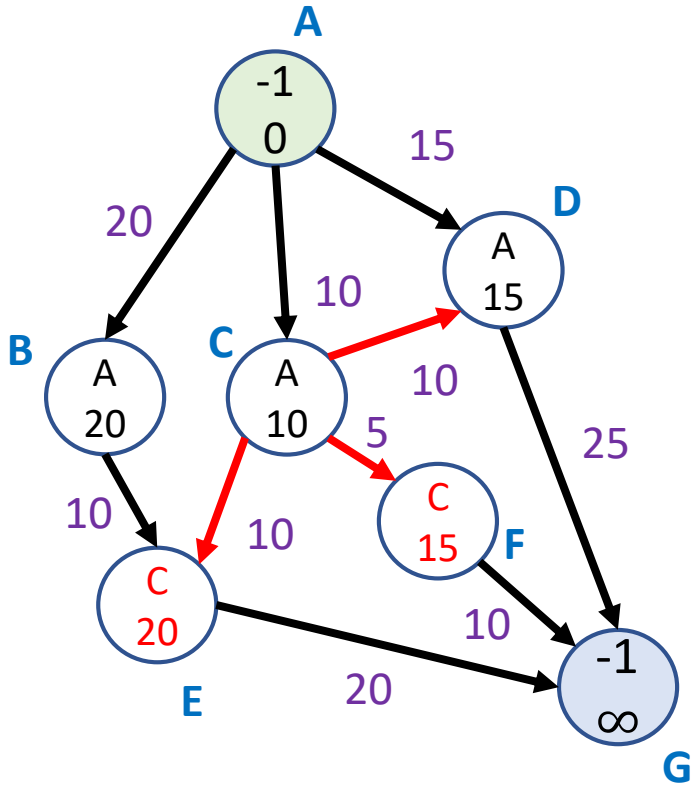


Queue = {B,C,D}

1. 對 B 的所有出邊做 Relax
2. 把有更新的 E 放入 Queue

Queue = {C,D,E}

SPFA

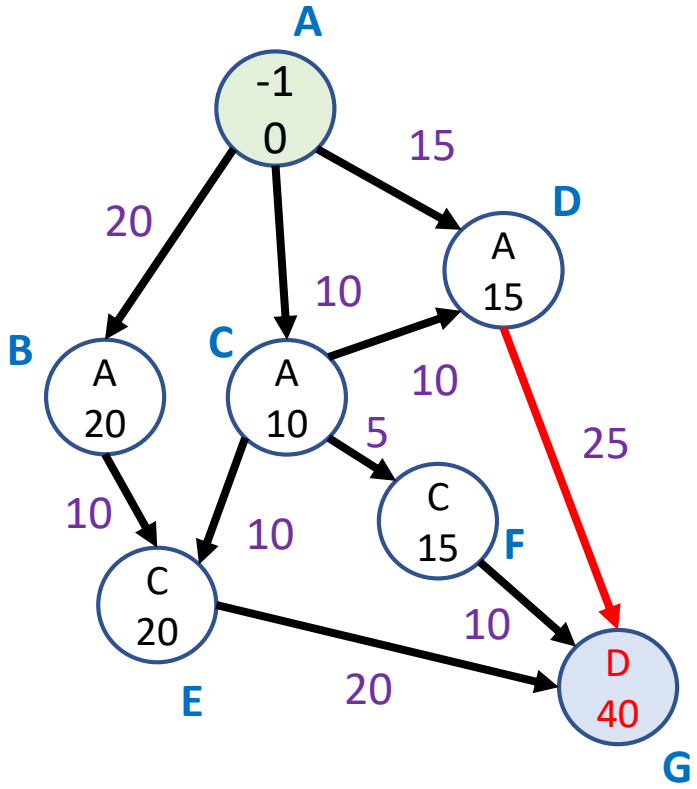


Queue = {C,D,E}

1. 對 C 的所有出邊做 Relax
2. 把有更新的 E,F 放入 Queue

Queue = {D,E,E,F}

SPFA

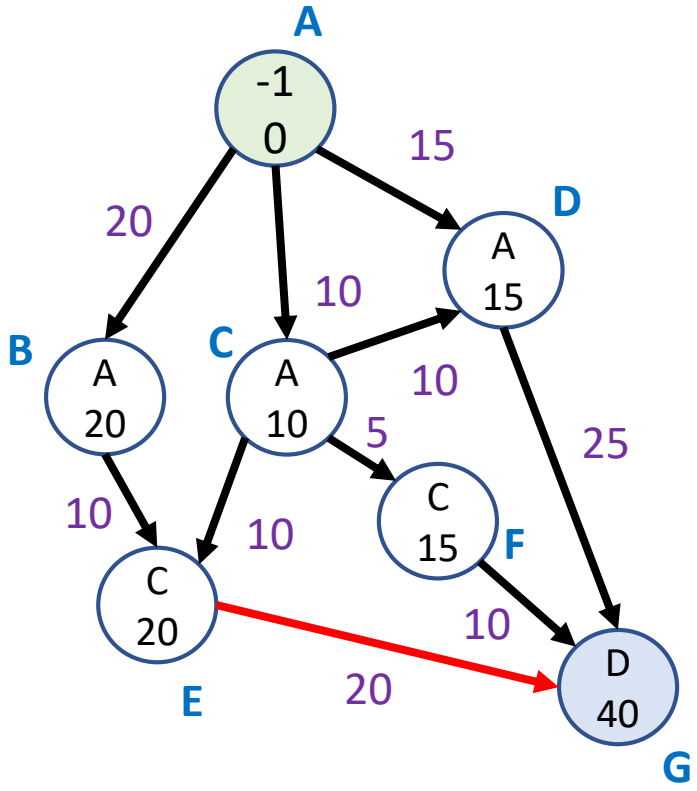


Queue = {D,E,E,F}

1. 對 D 的所有出邊做 Relax
2. 把有更新的 G 放入 Queue

Queue = {E,E,F,G}

SPFA

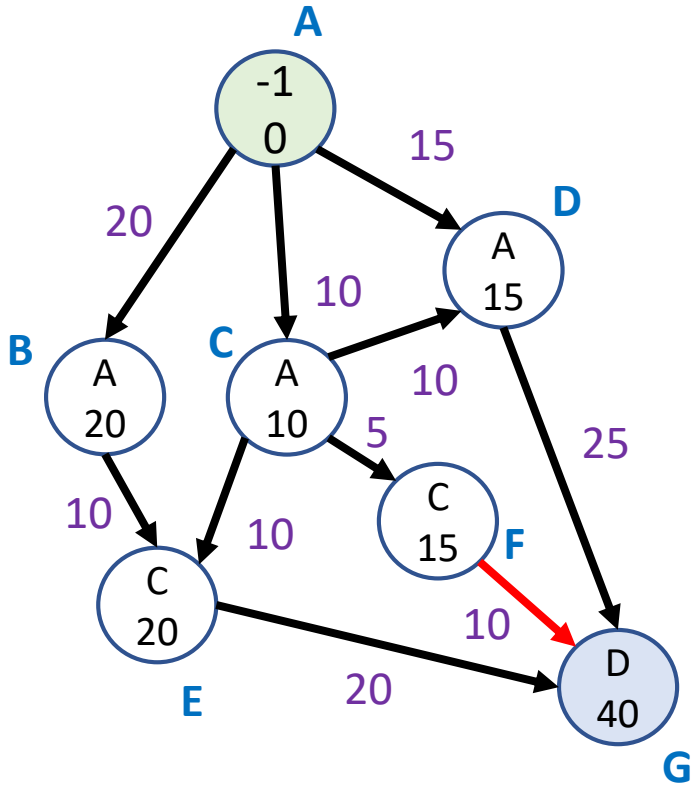


Queue = {E,E,F,G}

1. 對 E 的所有出邊做 Relax
2. 無任何更新
3. 因有兩個 E，故重複兩次

Queue = {F,G}

SPFA

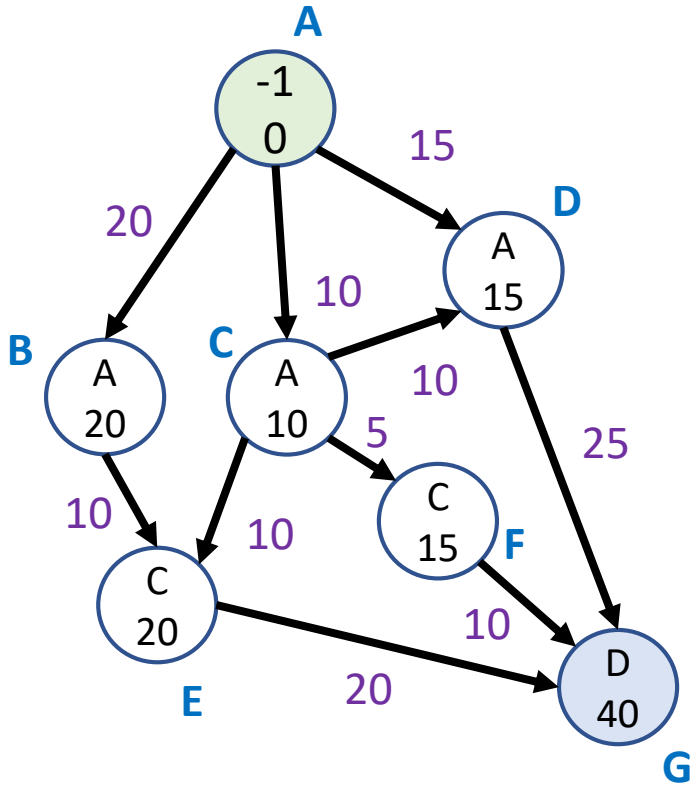


Queue = {F,G}

1. 對 F 的所有出邊做 Relax
2. 把有更新的 G 放入 Queue

Queue = {G,G}

SPFA



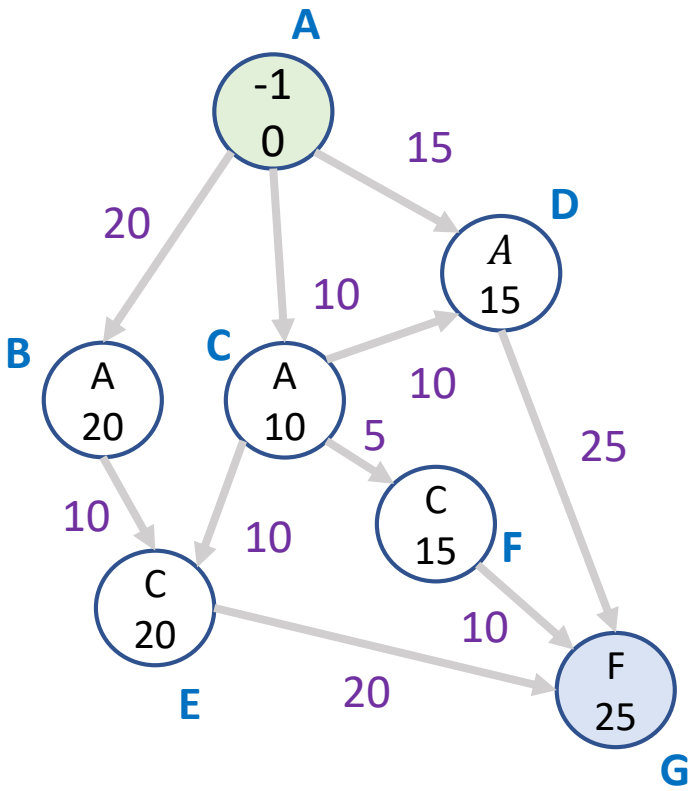
Queue = {G,G}

1. 對 G 的所有出邊做 Relax
2. 無任何更新
3. 因有兩個 G，故重複兩次

Queue = {}

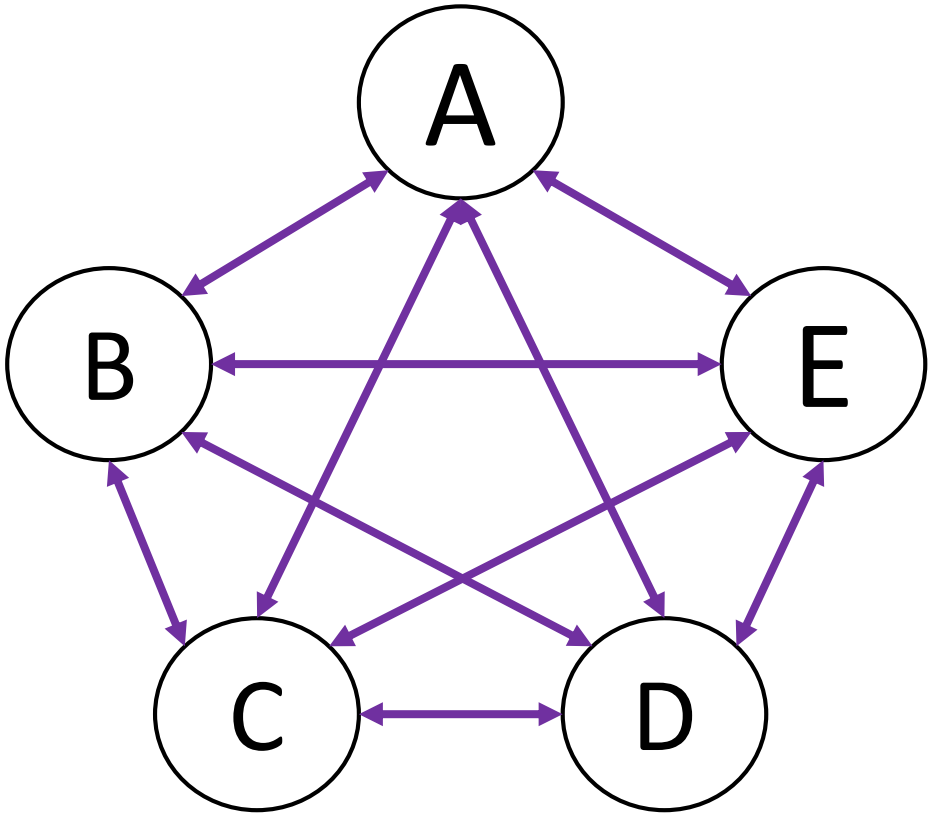
結束

SPFA



把起點 A 放入 Queue，Queue = {A}										
Iteration #1	e(A,B)	e(A,C)	e(A,D)							
pop出 A ，更新 A 的相鄰頂點 BCD，把有更動的 BCD 放入 Queue，Queue = {B,C,D}										
Iteration #2	e(B,E)									
pop出 B ，更新 B 的相鄰頂點 E，把有更動過 E 放入 Queue，Queue = {C,D,E}										
Iteration #3	e(C,E)	e(C,D)	e(C,F)							
pop出 C ，更新 C 的相鄰頂點 DEF，把有更動的 EF 放入 Queue，Queue = {D,E,E,F}										
Iteration #4	e(D,G)									
pop出 D ，更新 D 的相鄰頂點 G，把有更動的 G 放入 Queue，Queue = {E,E,F,G}										
Iteration #5	e(E,F)									
pop出 E ，更新 E 的相鄰頂點 G，未更新任何資料，Queue = {F,G} (重複兩次)										
Iteration #6	e(F,G)									
pop出 F ，更新 F 的相鄰頂點 G，把有更動的 G 放入 Queue，Queue = {G,G} (重複兩次)										
Iteration #6										
pop出 G ，更新 G的相鄰頂點，未更新任何資料，Queue = {} (重複兩次)										

SPFA



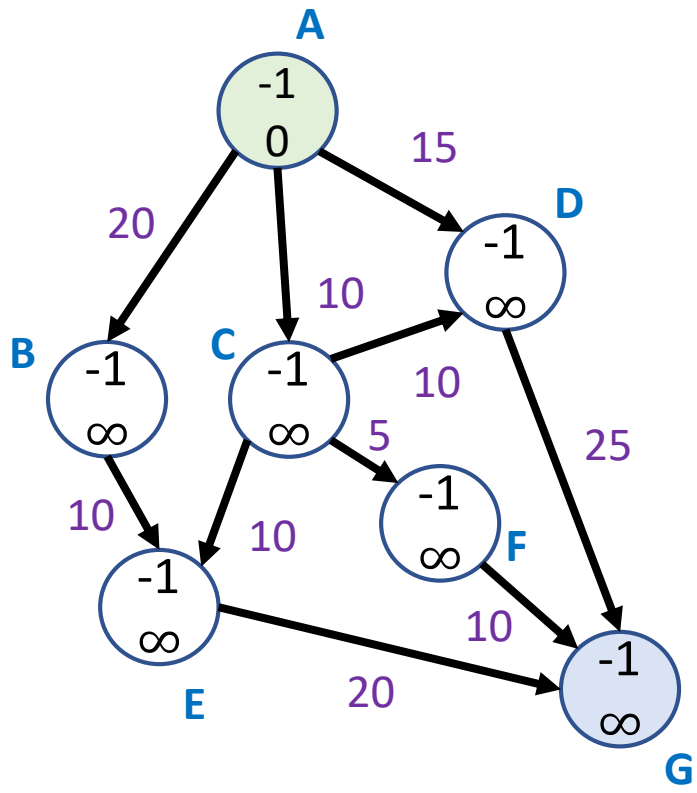
SPFA複雜度
Worst Case：完全圖

1. **relax |E|/2 個邊時，所有頂點會被更新**
2. **更新 |V|次才能知道有環**
3. **時間複雜度：|V| |E|**

Practice 2

Mission

完成 SPFA 演算法

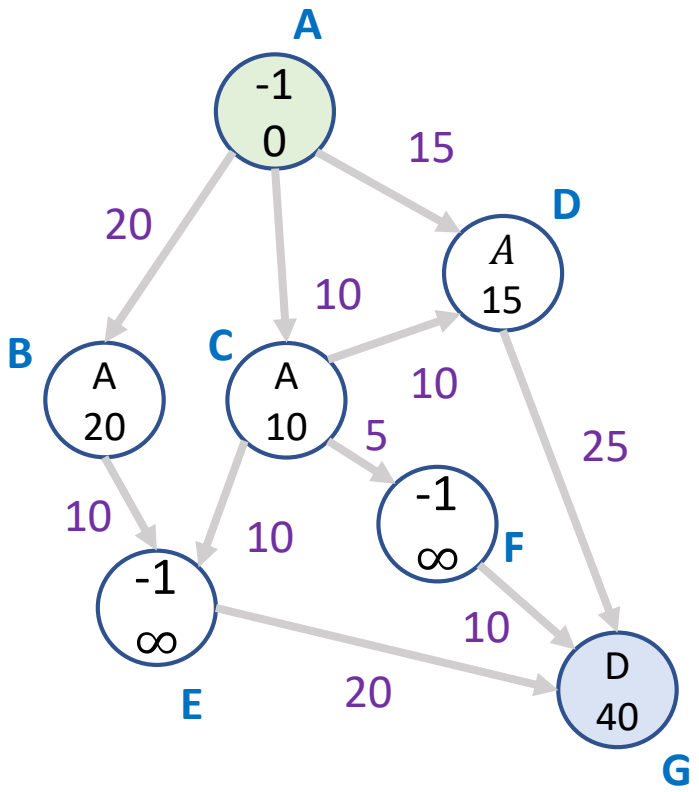


DAG Algorithm

Bellman-Ford Algorithm

為什麼要進行 |V|-1 次？

因為Edge relax的過程並沒有依照最短路徑的順序！



Bellman-Ford Algorithm

||

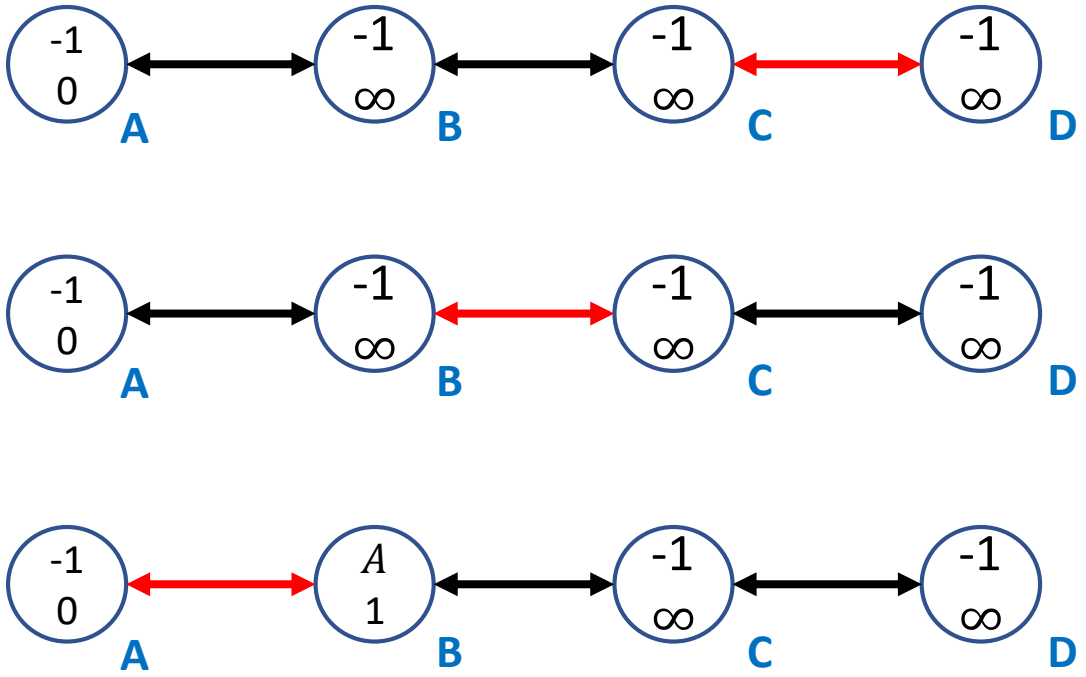
暴力法

Bellman-Ford Algorithm

Worst Case：

- Relax 的順序剛好跟最短路徑的順序相反
- 每次 Relax 後只能優化單一子路徑
- 最多要經過 |V| 個頂點，需要有 |V|-1 條子路徑
- 至少需要重複 |V|-1 次

若改善 Relax 的 Edge 順序就能夠大幅優化複雜度



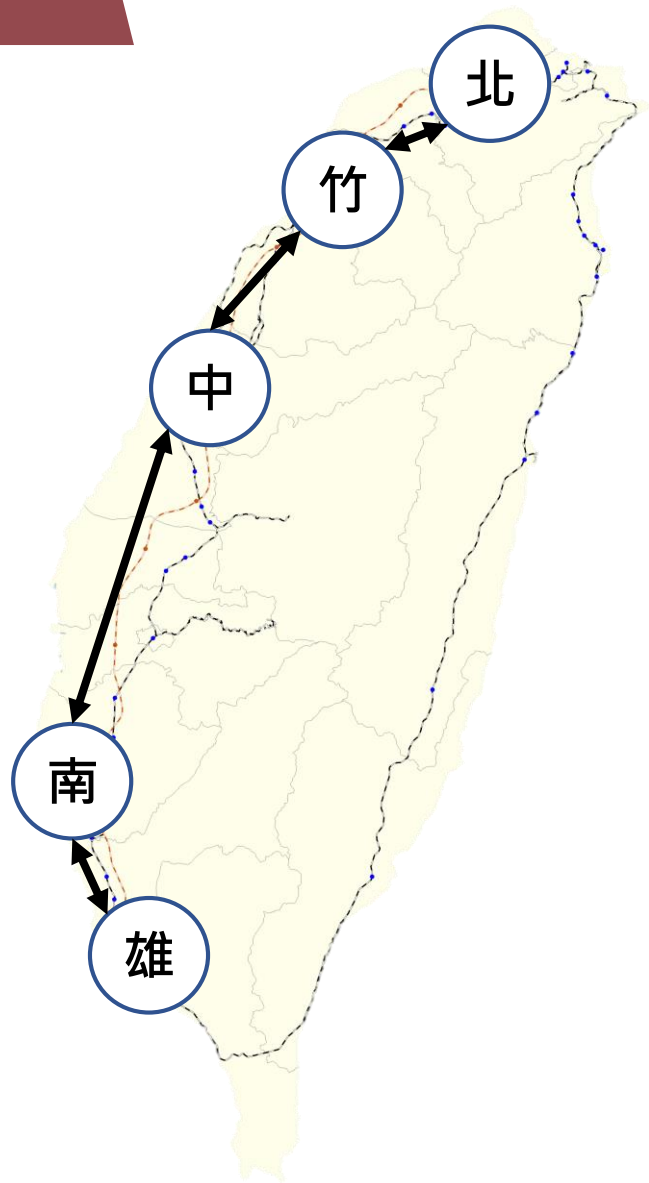
最短路徑問題

- Path-relaxation

➢ 若 $p_{(v_1,v_n)} = (v_1, v_2, \ldots\ldots, v_n)$ 是 v_1, v_n 間的最短路徑， 則依序執行 $Relax(v_1, v_2, w_1)$、$Relax(v_2, v_3, w_2)$、...、$Relax(v_{n-1}, v_n, w_{n-1})$ 最後便會得到最短路徑。

➢ 已知台北騎機車到高雄的中間必會經過新竹、台中、台南

- ✓ 對台北→新竹、新竹→台中、台中到台南、台南到高雄 Relax
- ✓ 便會得到台北→高雄的最短路徑

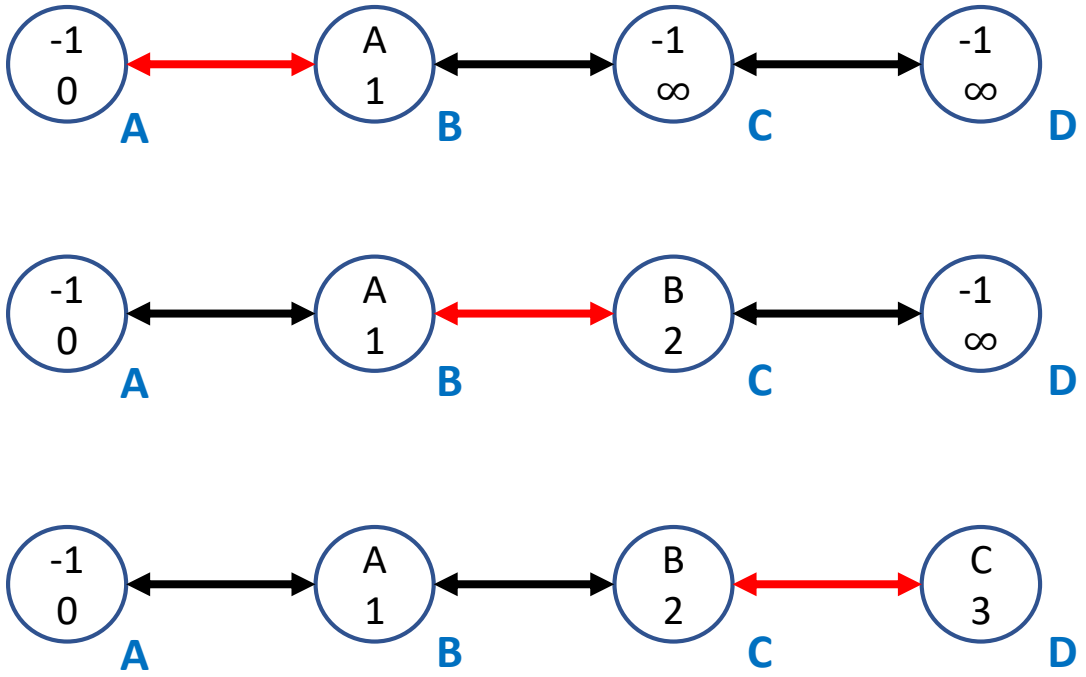


DAG Algorithm

Path-relaxation

- 若事先依照 Edge 的經過順序進行排序
- 只需把經過的 Edge 分別做一次 Relax 就可以
- 複雜度： O(|E|)
- Bellman-Ford Algorithm：O(|V||E|)

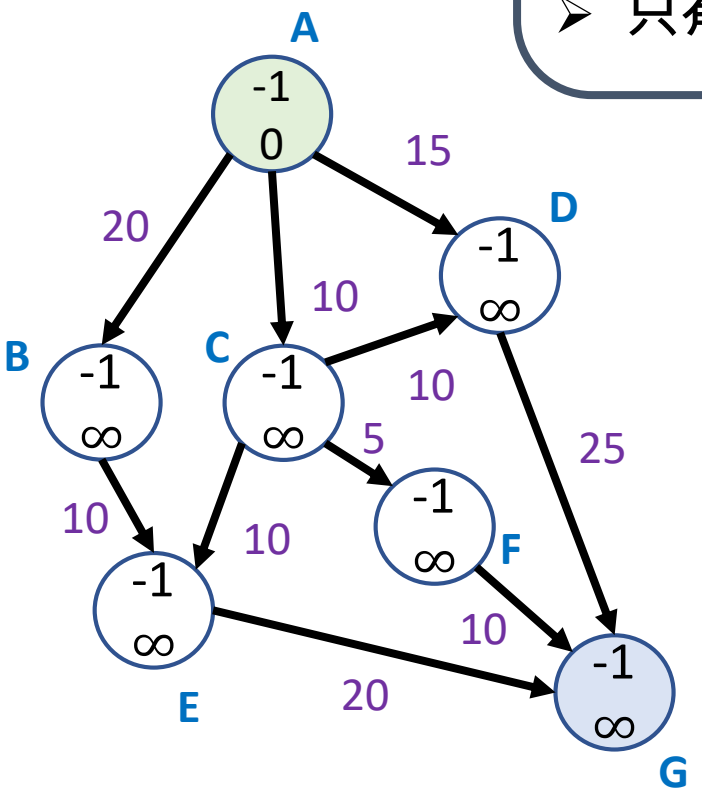
怎麼依照經過順序將 Vertex 與 Edge 進行排序？

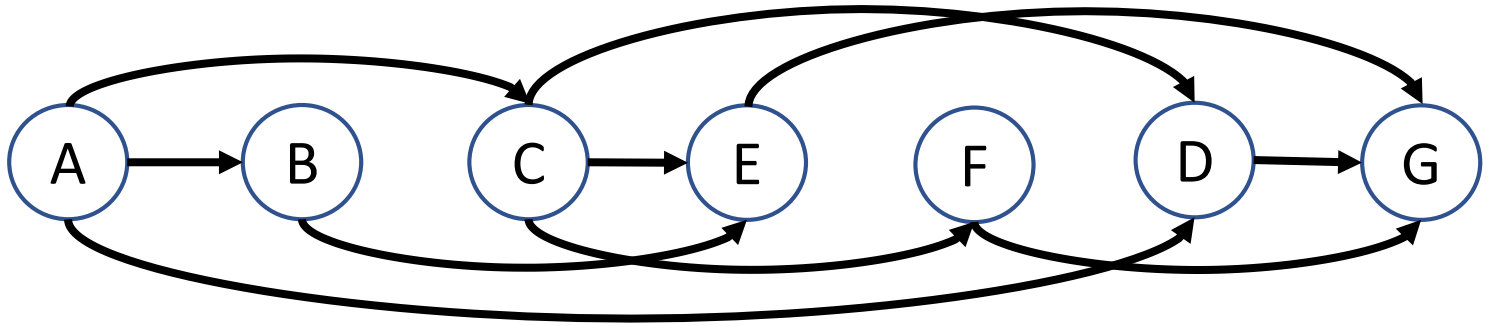


Topological Sort

Topological Sort(拓撲排序)

- 每條在有向無環圖的 Edge(A,B)
  - ✓ 拓樸排序必是 Vertex(A) 在 Vertex(B) 之前
- 只有有向無環圖的拓樸排序才有意義

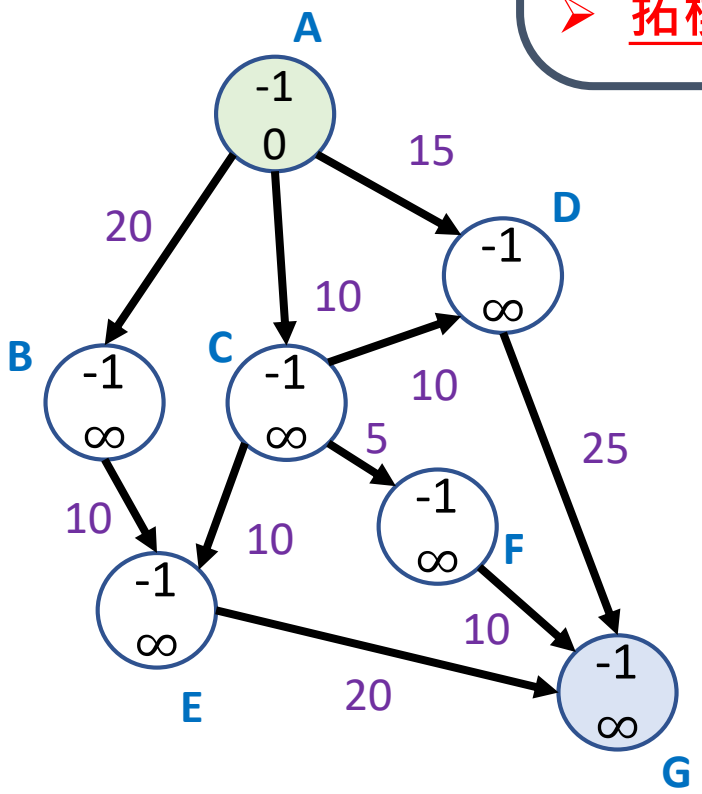




Topological Sort

如何產生 Topological Sort(拓撲排序)

- 進行一次 DFS 便可以把路過次序記錄下來
- 依照離開的時間戳記就可以進行拓樸排序
- 拓樸排序不是唯一解！

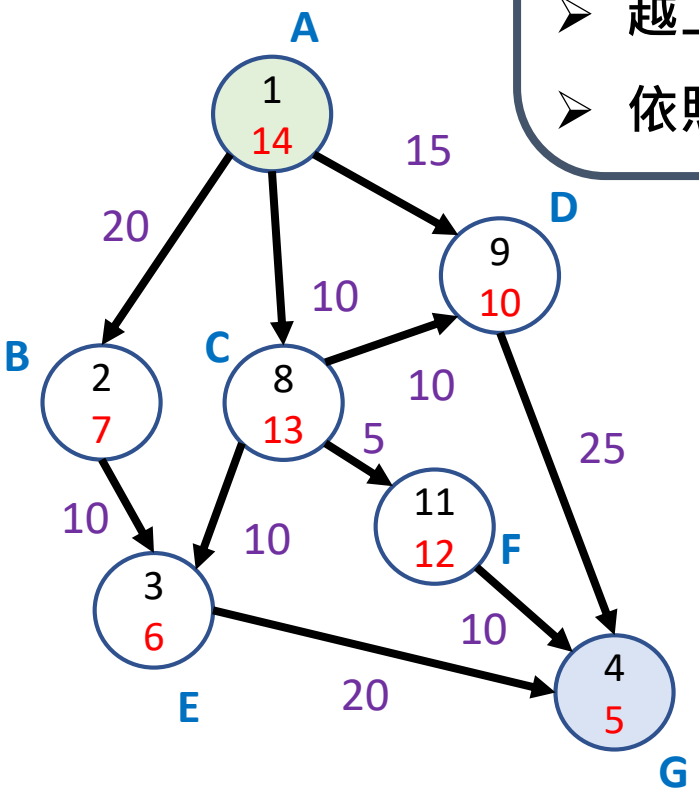


Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

Topological Sort

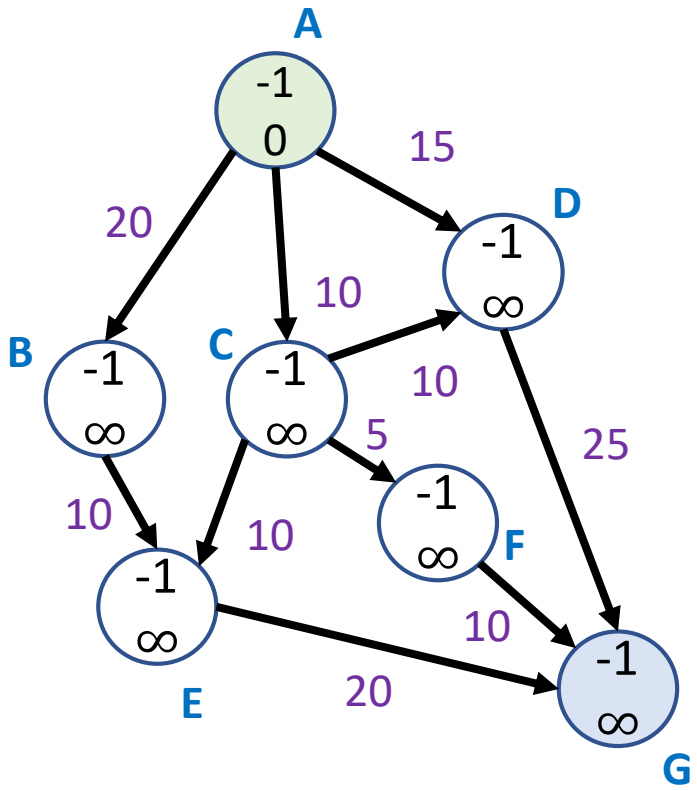
如何產生 Topological Sort(拓撲排序)

- 離開的時間戳記代表處理完該頂點的時間點
  - ✓ 也就是從 stack 取出的時間點
- 越上游的頂點，離開時間戳記越大
- 依照離開時間戳記的大小便可以得到拓樸排序



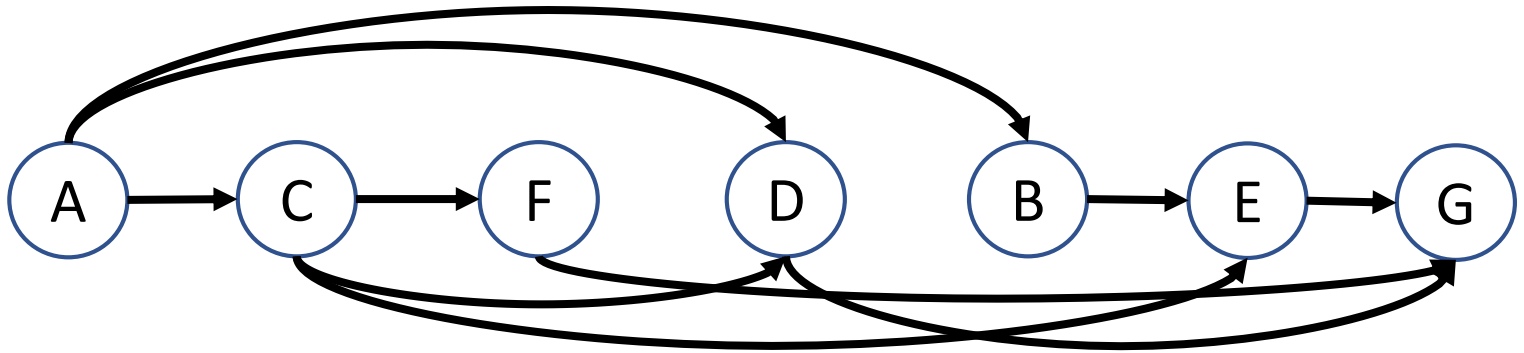
Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

Topological Sort

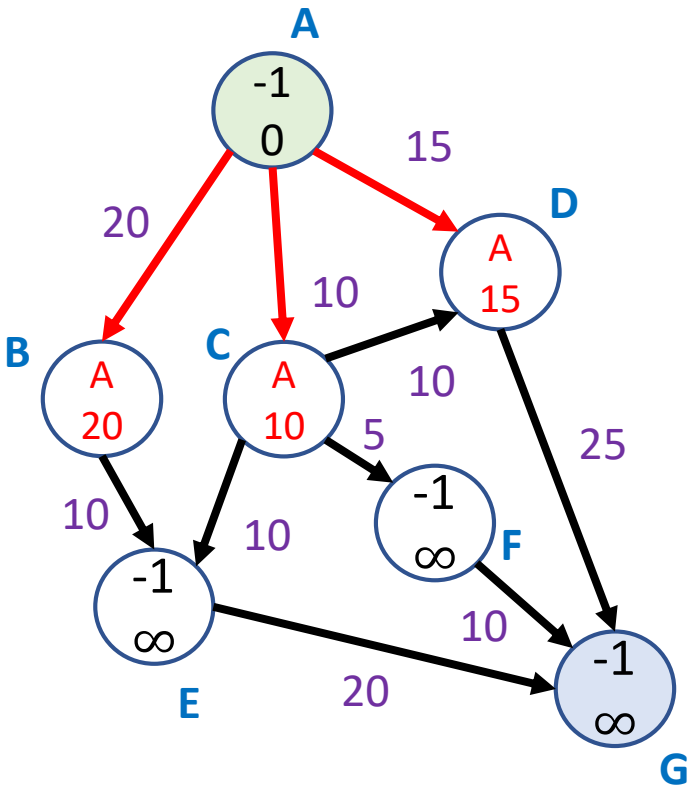


Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

ACFDBEG

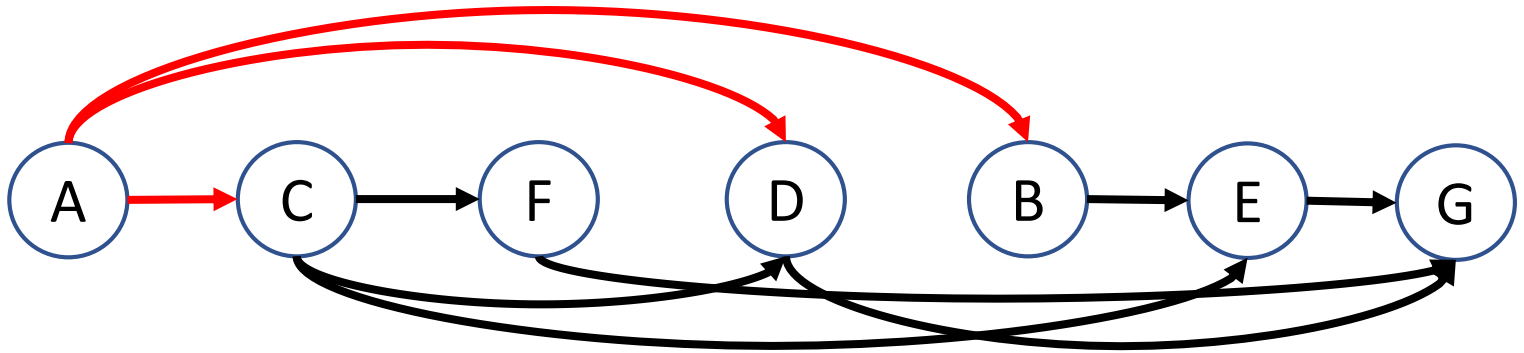


Topological Sort

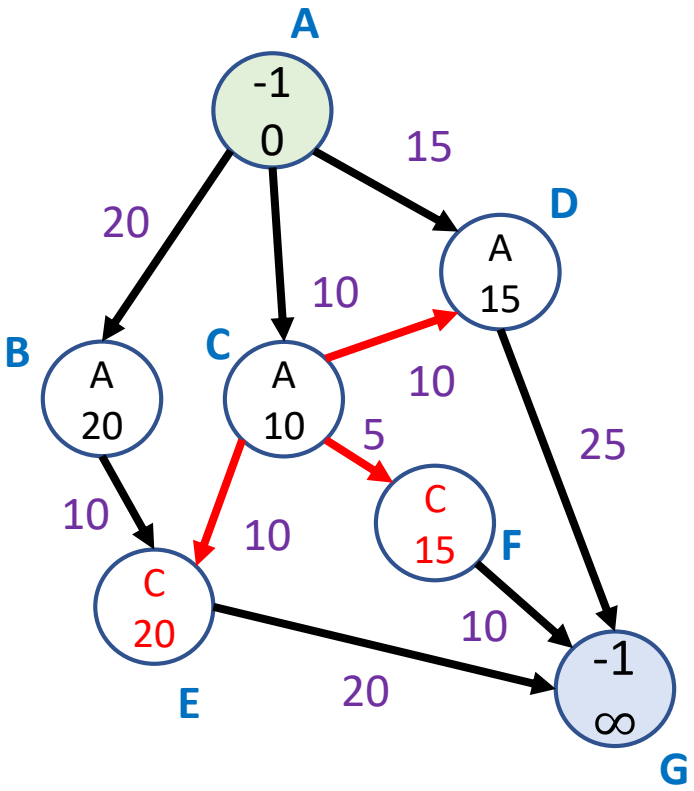


Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

ACFDBEG

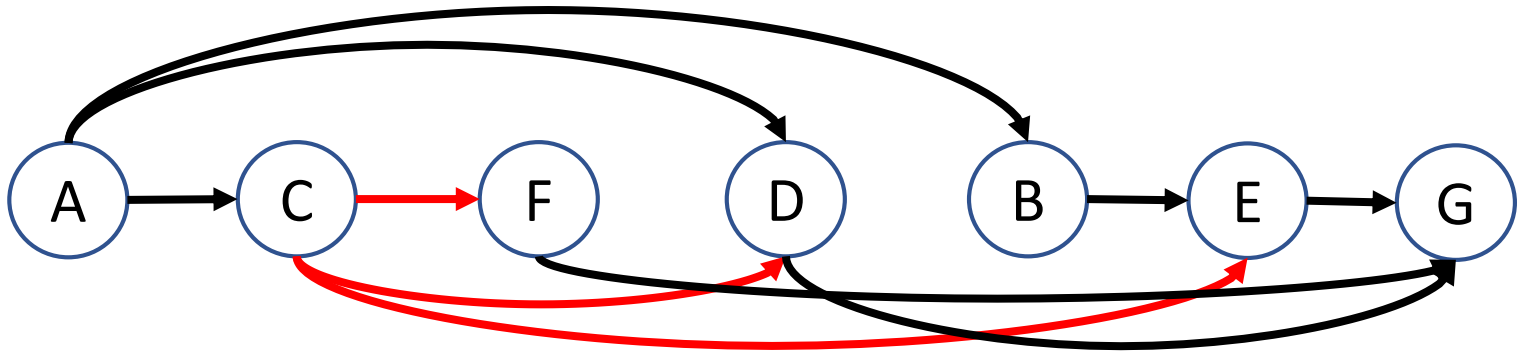


Topological Sort

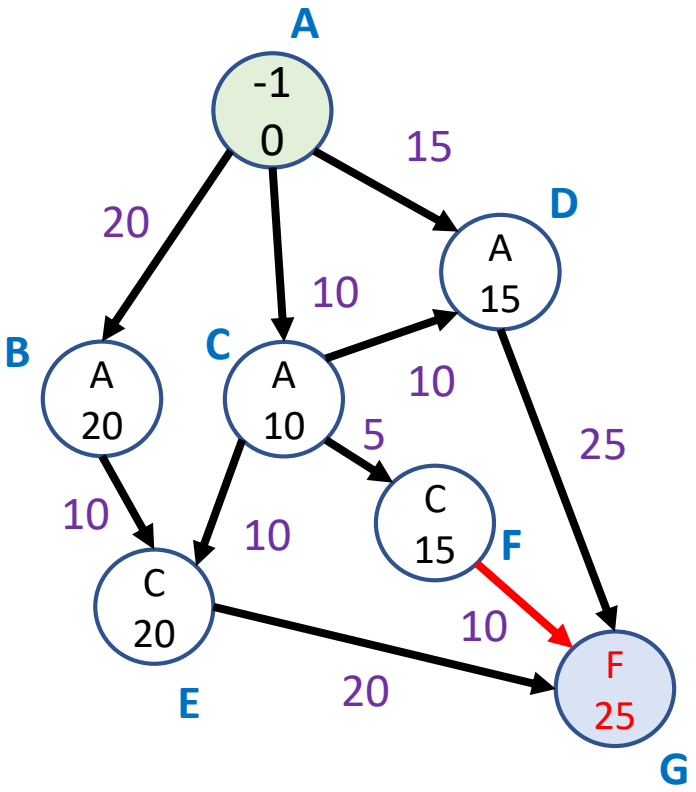


Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

ACFDBEG

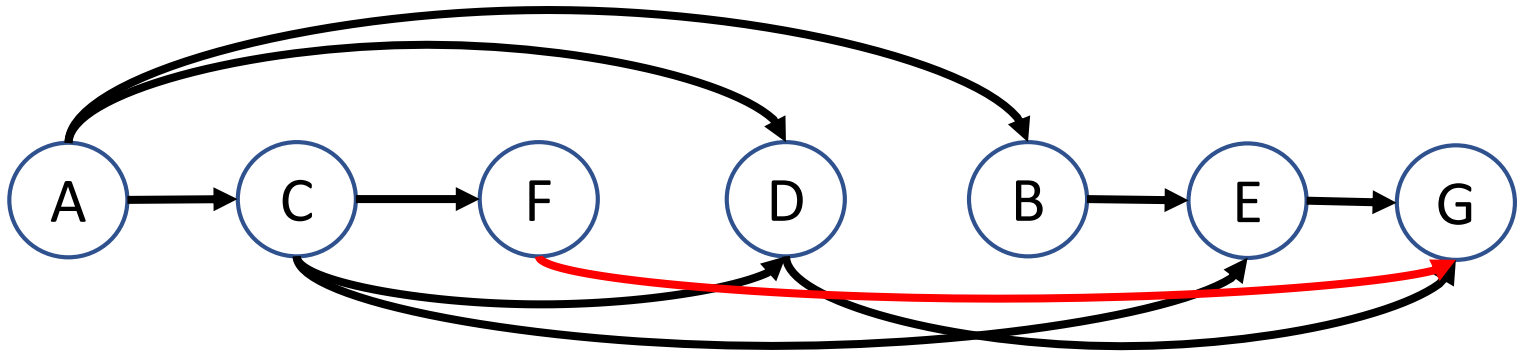


Topological Sort

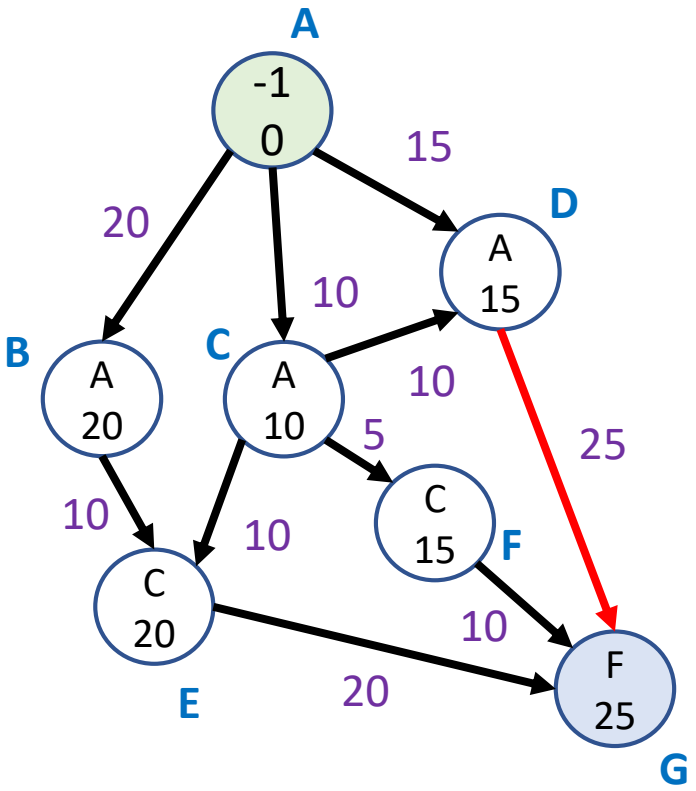


Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

ACFDBEG

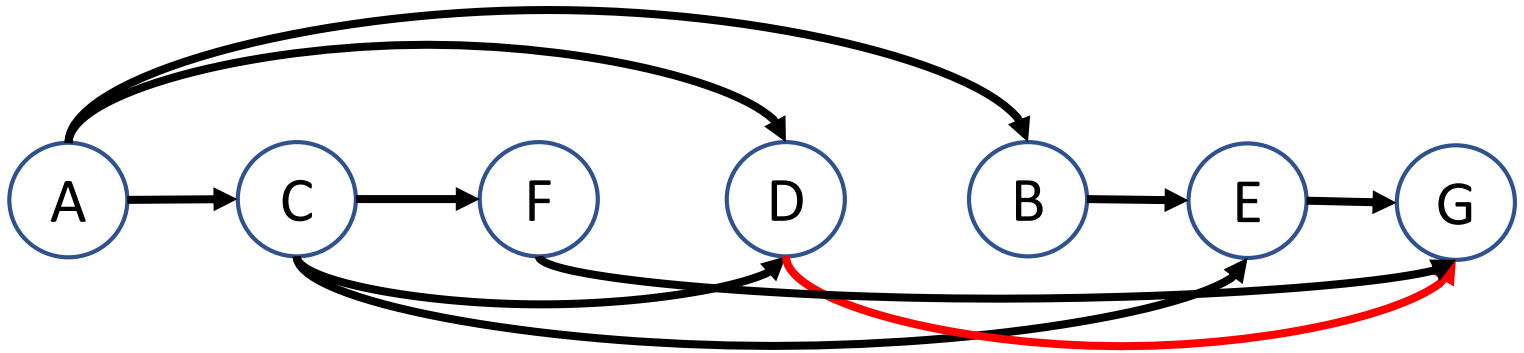


Topological Sort

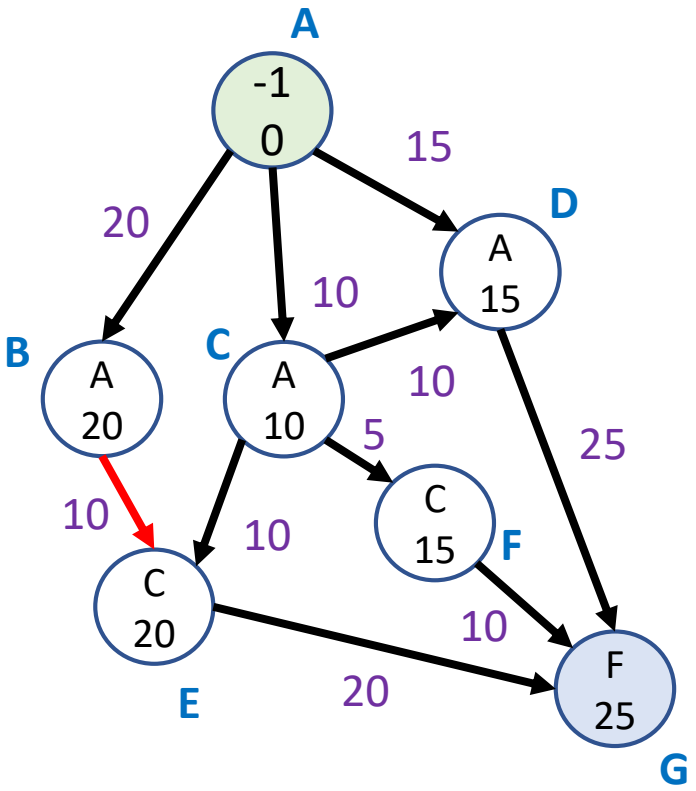


Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

ACFDBEG

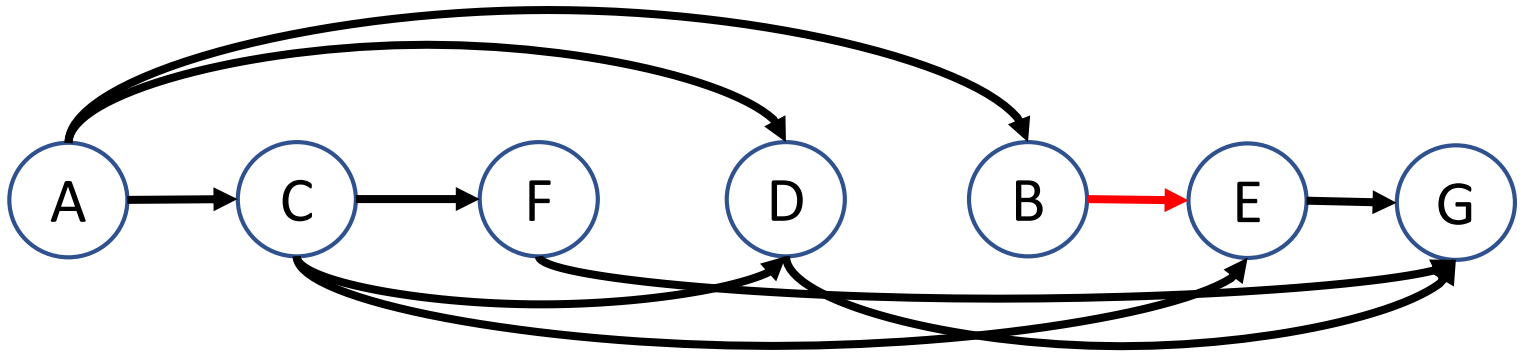


Topological Sort

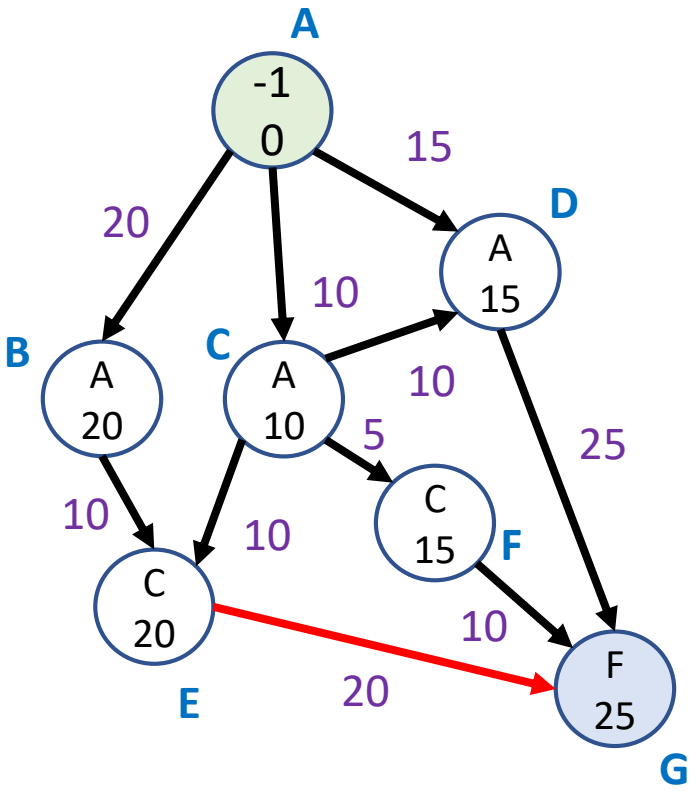


Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

ACFDBEG

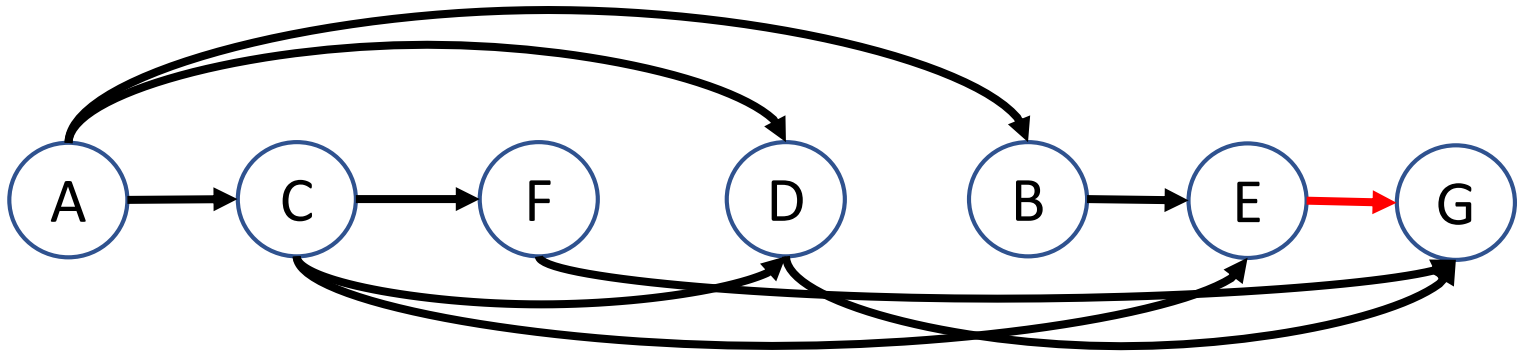


Topological Sort



Vertex	進入	離開
A	1	14
B	2	7
C	8	13
D	9	10
E	3	6
F	11	12
G	4	5

ACFDBEG



DAG Algorithm

```
1  DAG(G,w,s){
2        Topological Sort(G,s)
3        initialize (G,s)
4        for vertex(u) in Topological Sort:
5             for vertex(v) ∈ Adj[u]
6                  Relax(u,v,w)
7        return true
8  }
```

1. 先把圖進行拓樸排序
2. 根據 Path-relaxation
   - ✓ 依拓樸排序的順序進行 Relax 一次即可！

DAG Algorithm

```
1  DAG(G,w,s){
2        Topological Sort(G,s)
3        initialize (G,s)
4        for vertex(u) in Topological Sort:
5              for vertex(v) ∈ Adj[u]
6                    Relax(u,v,w)
7        return true
8  }
```

O(|V|+|E|)

O(|E|)

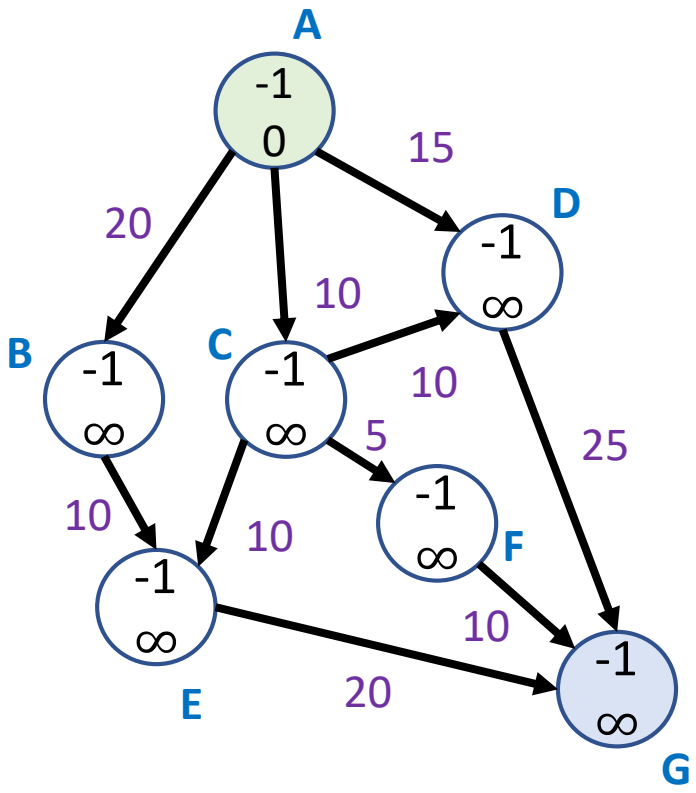
複雜度分析

- ✓ 拓樸排序的複雜度： O(|V|+|E|)
- ✓ 依序 Relax 時的複雜度： O(|E|)
- ✓ 總計：O(|V|+|E|)

Practice 3

Mission

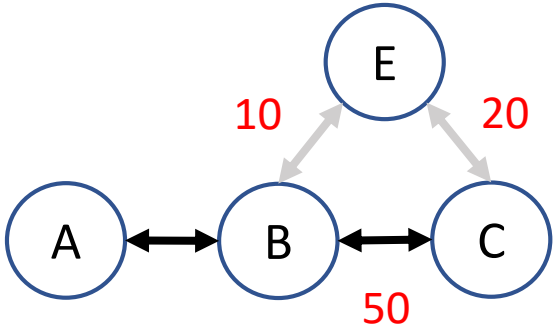
完成 DAG 演算法

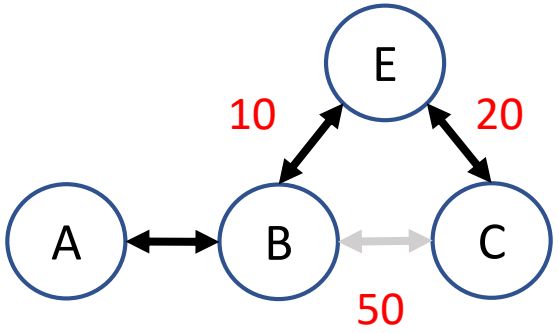


Dijkstra's Algorithm

Convergence Property

- Convergence property
  - 如果 A-C 的最短路徑包含 (A, B) 且 d[B]=δ(A,B)，則此時執行 Relax(B, C, w) 會讓 distance[C] = δ(A,C)
  - Ex: 已知高雄到紐約的最短路徑必會包括：
    - ✓ 左營高鐵→桃園機場
    - ✓ 則對桃園機場→紐約進行 Relax 後會使該路徑為最短



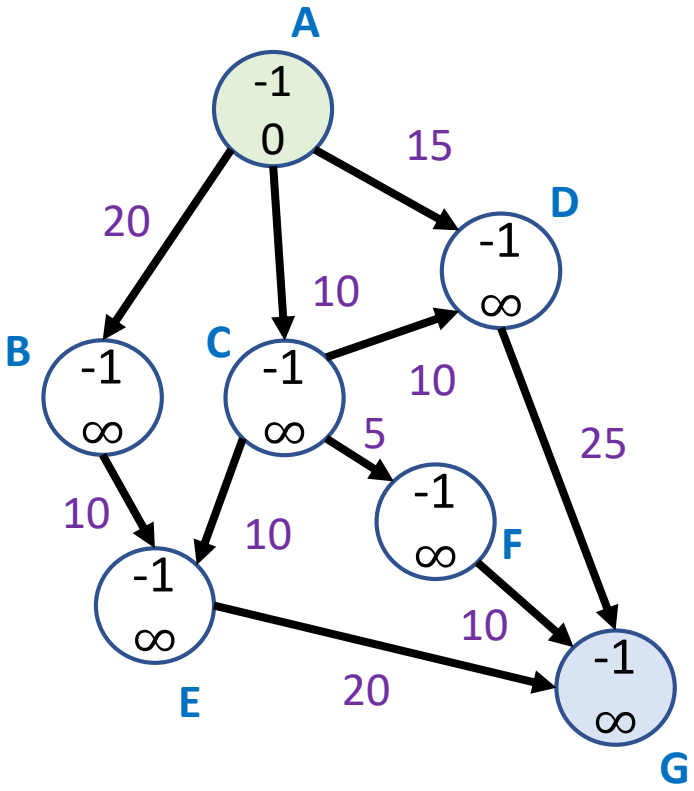


Dijkstra's Algorithm

Dijkstra's Algorithm

1. 假定沒有負邊，Edge 越多成本必定增加
2. 每輪都以當前最佳選擇為下一步的貪婪演算法
3. 分成已經找到最短路徑跟還沒找到最短路徑兩組
4. 利用 Convergence Property
   - ✓ 從已經找到最短路徑那組出發，只需要對剩下還沒找到最小路徑那組進行 Relax
5. 通常以 Priority queue 實做。

Dijkstra's Algorithm



1. 初始化 Distance 與 Predecessor
2. 分成已經找到最短路徑跟還沒找到最短路徑兩組
3. 從還沒找到最短路徑中找到 Distance 最小的頂點 (V)
4. Relax V，V 就可放入已經找到最短路徑組
5. 重複步驟 2~4 共 |V| 次
6. 直到所有頂點都被放入已經找到最短路徑組

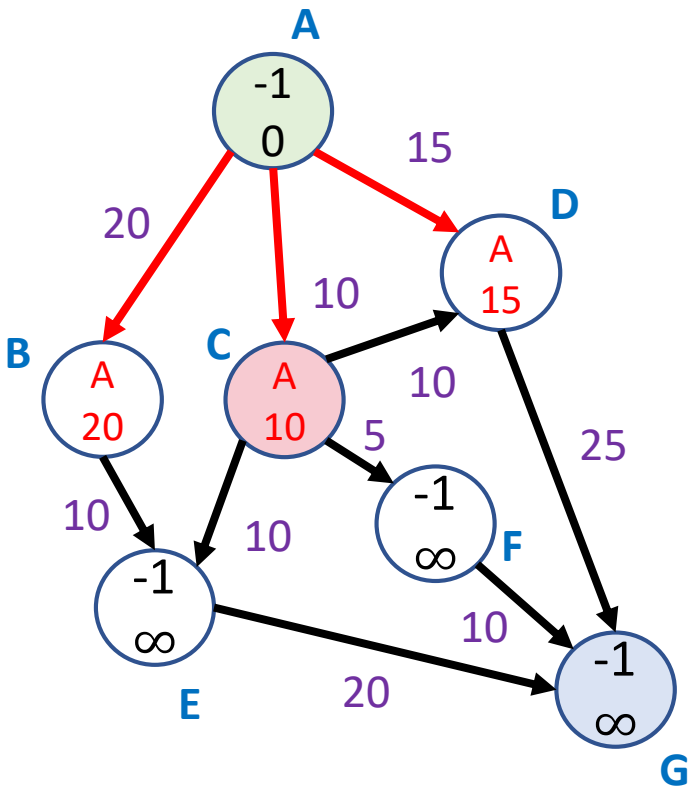
最短路徑組

- A(-1,0)

尚未是最短路徑

- B(-1,∞),C(-1,∞),D(-1,∞),E(-1,∞),F(-1,∞),G(-1,∞)

Dijkstra's Algorithm



1. 把起點 A 放入已找到最短路徑組
2. 其餘頂點放入還沒找到最短路徑組
3. 從 A 出發， Relax 所有 A 出發的邊
   - ✓ B(A,20)、C(A,10)、D(A,15)
4. 從還沒找到最短路徑組選最小放入已找到最短路徑組

C 必定最小，因沒有負邊，其他路徑都會比目前距離大

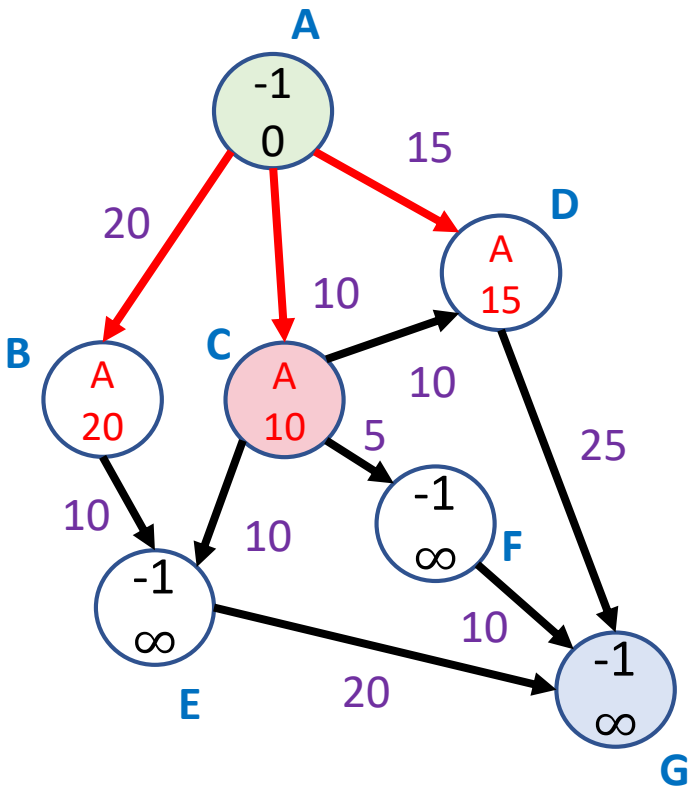
最短路徑組

- A(-1,0)

尚未是最短路徑

- B(A,20),C(A,10),D(A,15),E(-1,∞),F(-1,∞),G(-1,∞)

Dijkstra's Algorithm



1. 把起點 A 放入已找到最短路徑組
2. 其餘頂點放入還沒找到最短路徑組
3. 從 A 出發， Relax 所有 A 出發的邊
   - ✓ B(A,20)、C(A,10)、D(A,15)
4. 從還沒找到最短路徑組選最小放入已找到最短路徑組

C 必定最小，因沒有負邊，其他路徑都會比目前距離大

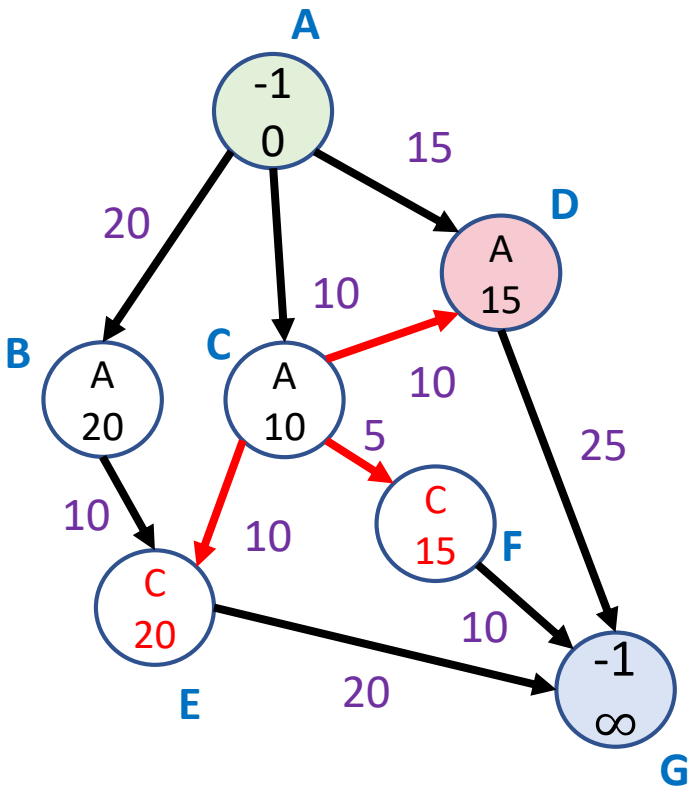
最短路徑組

- A(-1,0),C(A,10)

尚未是最短路徑

- B(A,20),D(A,15),E(-1,∞),F(-1,∞),G(-1,∞)

Dijkstra's Algorithm



1. 從 C 出發， Relax 所有 C 出發的邊
   - ✓ D(A,15)、 E(C,20)、F(C,15)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

D 必定最小，因沒有負邊，其他路徑都會比目前距離大

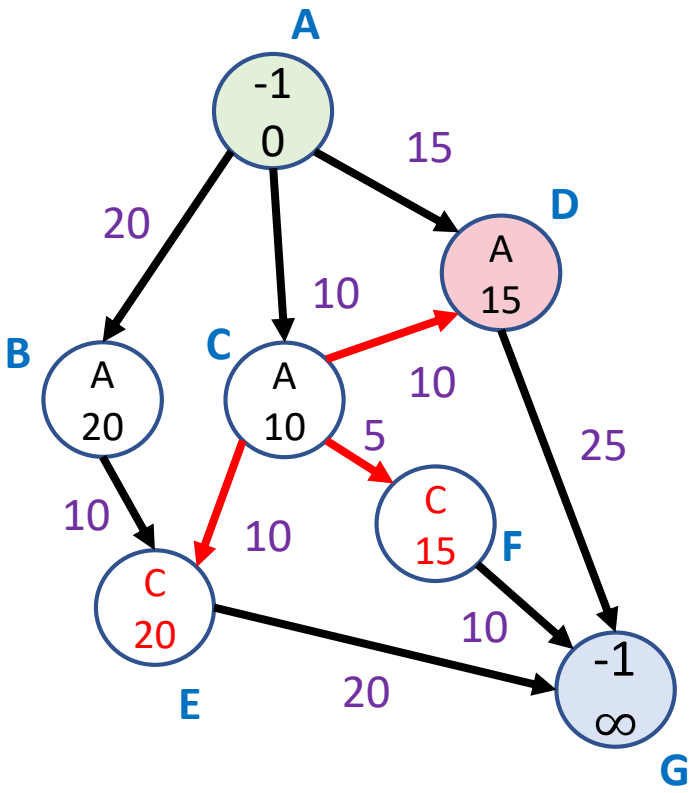
最短路徑組

- A(-1,0),C(A,10)

尚未是最短路徑

- B(A,20),D(A,15),E(C,20),F(C,15),G(-1,∞)

Dijkstra's Algorithm



1. 從 C 出發， Relax 所有 C 出發的邊
   - ✓ D(A,15)、 E(C,20)、F(C,15)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

D 必定最小，因沒有負邊，其他路徑都會比目前距離大

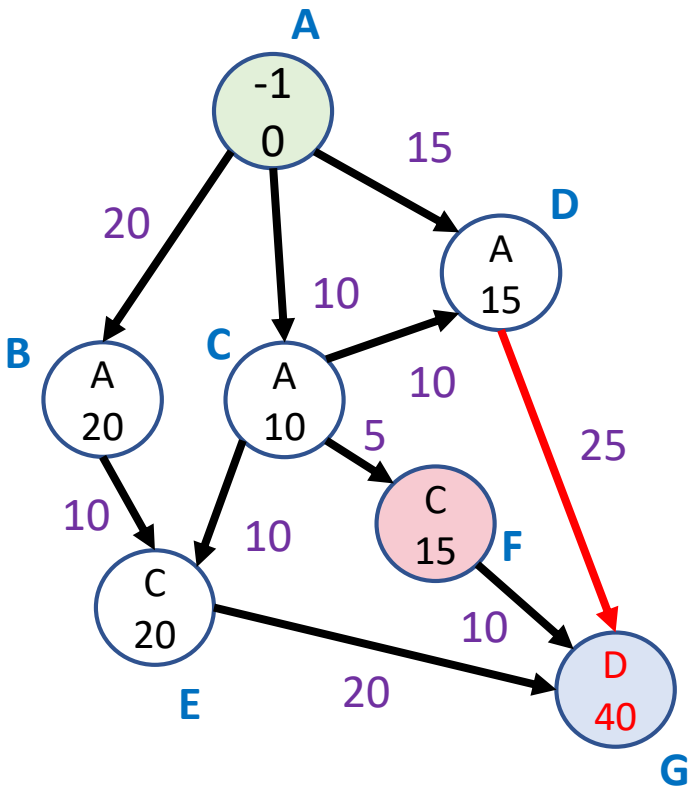
最短路徑組

- A(-1,0),C(A,10),D(A,15)

尚未是最短路徑

- B(A,20),E(C,20),F(C,15),G(-1,∞)

Dijkstra's Algorithm



1. 從 D 出發， Relax 所有 D 出發的邊
   - ✓ G(D,40)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

F 必定最小，因沒有負邊，其他路徑都會比目前距離大

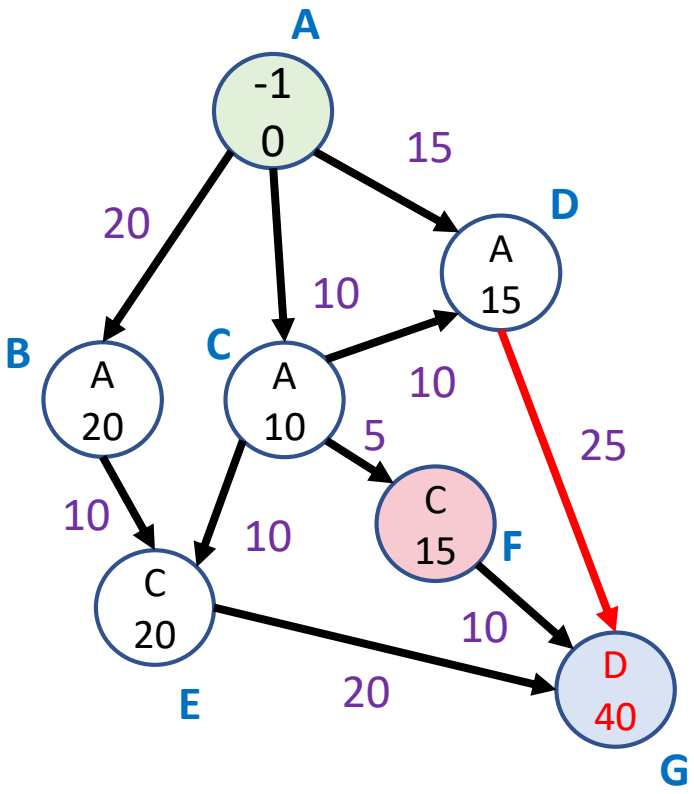
最短路徑組

- A(-1,0),C(A,10),D(A,15)

尚未是最短路徑

- B(A,20),E(C,20),F(C,15),G(D,40)

Dijkstra's Algorithm



1. 從 D 出發， Relax 所有 D 出發的邊
   - ✓ G(D,40)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

F 必定最小，因沒有負邊，其他路徑都會比目前距離大

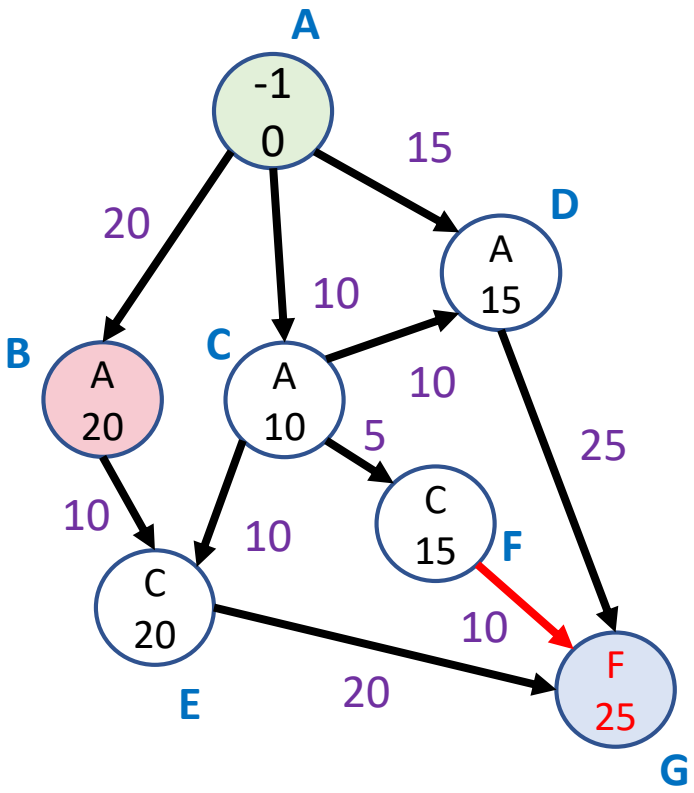
最短路徑組

- A(-1,0),C(A,10),D(A,15),F(C,15)

尚未是最短路徑

- B(A,20),E(C,20),G(D,40)

Dijkstra's Algorithm



1. 從 F 出發， Relax 所有 F 出發的邊
   - ✓ G(F,25)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

B 必定最小，因沒有負邊，其他路徑都會比目前距離大

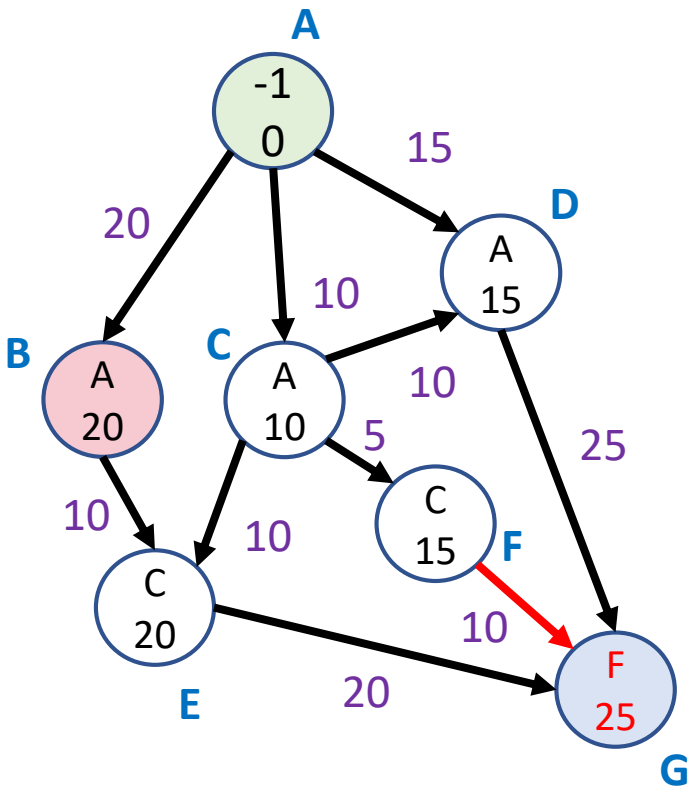
最短路徑組

- A(-1,0),C(A,10),D(A,15),F(C,15)

尚未是最短路徑

- B(A,20),E(C,20),G(F,25)

Dijkstra's Algorithm



1. 從 F 出發， Relax 所有 F 出發的邊
   - ✓ G(F,25)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

B 必定最小，因沒有負邊，其他路徑都會比目前距離大

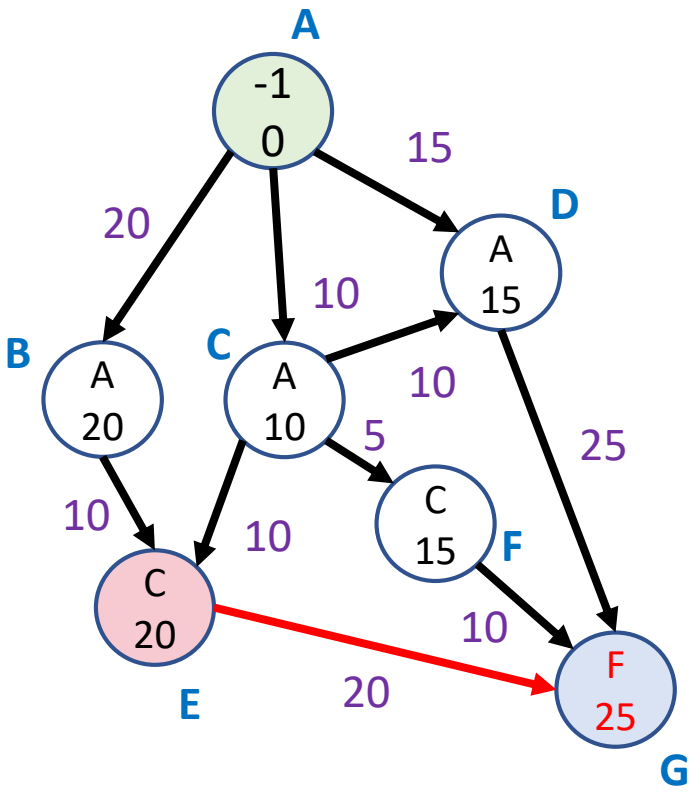
最短路徑組

- A(-1,0),C(A,10),D(A,15),F(C,15),B(A,20)

尚未是最短路徑

- E(C,20),G(F,25)

Dijkstra's Algorithm



1. 從 E 出發， Relax 所有 E 出發的邊
   - ✓ G(F,25)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

E 必定最小，因沒有負邊，其他路徑都會比目前距離大

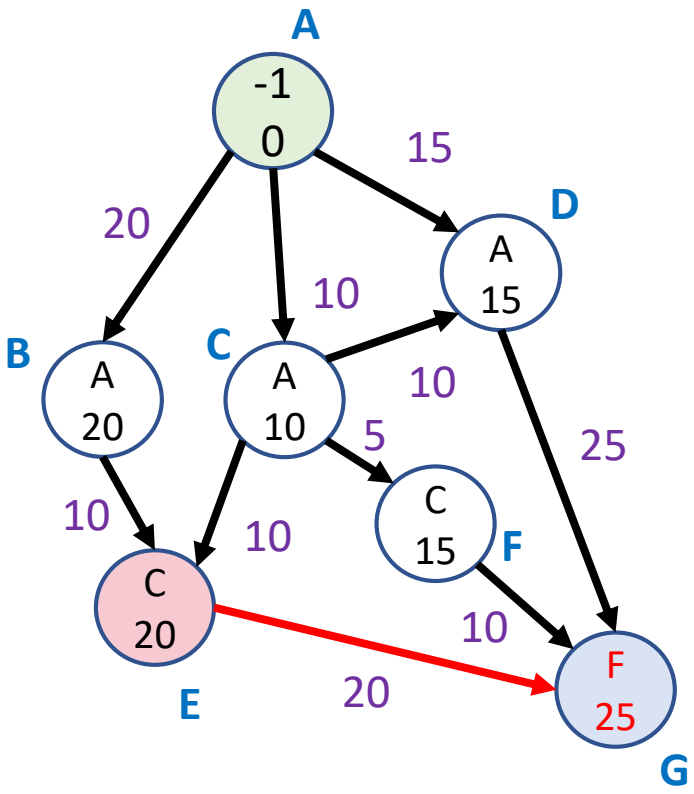
最短路徑組

- A(-1,0),C(A,10),D(A,15),F(C,15),B(A,20)

尚未是最短路徑

- E(C,20),G(F,25)

Dijkstra's Algorithm



1. 從 E 出發， Relax 所有 E 出發的邊
   - ✓ G(F,25)
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

E 必定最小，因沒有負邊，其他路徑都會比目前距離大

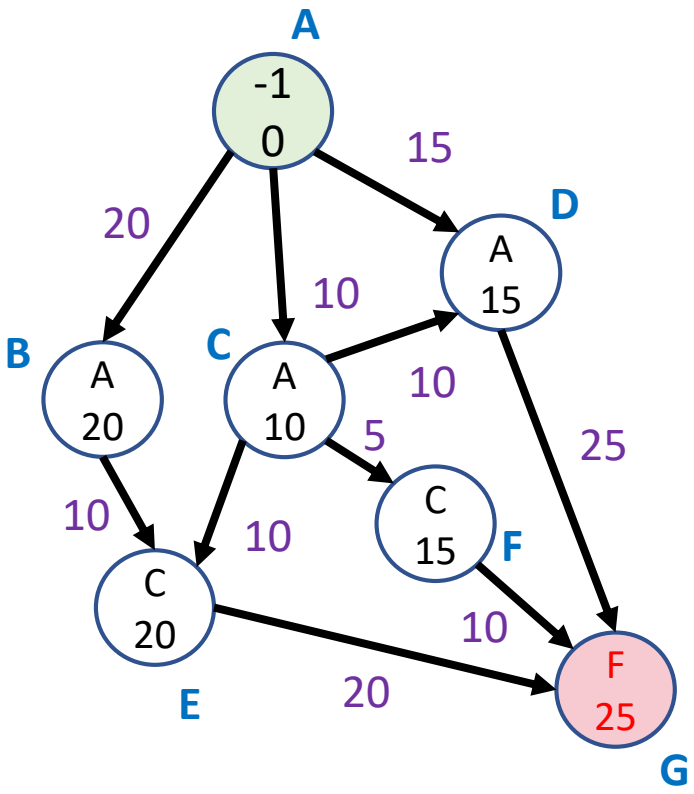
最短路徑組

- A(-1,0),C(A,10),D(A,15),F(C,15),B(A,20),E(C,20)

尚未是最短路徑

- G(F,25)

Dijkstra's Algorithm



1. 從 G 出發， Relax 所有 G 出發的邊
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

G 必定最小，因沒有負邊，其他路徑都會比目前距離大

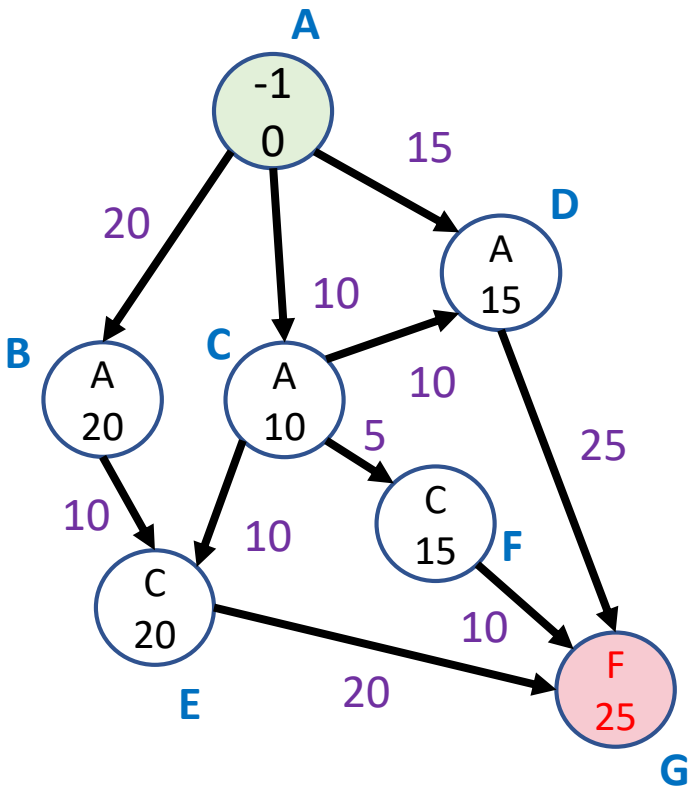
最短路徑組

- A(-1,0),C(A,10),D(A,15),F(C,15),B(A,20),E(C,20)

尚未是最短路徑

- G(F,25)

Dijkstra's Algorithm



1. 從 G 出發， Relax 所有 G 出發的邊
2. 從還沒找到最短路徑組選最小放入已找到最短路徑組

G 必定最小，因沒有負邊，其他路徑都會比目前距離大

最短路徑組

- A(-1,0),C(A,10),D(A,15),F(C,15),B(A,20),E(C,20),G(F,25)

尚未是最短路徑

Dijkstra's Algorithm

```
1  Dijkstra(G,w,s){
2        initialize (G,s)
3        priority_queue = V[G]
4        while priority_queue is not empty:
5              u = priority_queue.pop()
6              for vertex(v) ∈ Adj[u]
7                    Relax(u,v,w)
8        return true
9  }
```

1. priority_queue 會把 Distance 小的優先 pop
2. 根據 Convergence property
   - ✓ 對 pop 出來的頂點相連的 Edge 進行 Relax

Dijkstra's Algorithm

```
1  Dijkstra(G,w,s){
2        initialize (G,s)
3        priority_queue = V[G]
4        while priority_queue is not empty:
5            u = priority_queue.pop()
6            for vertex(v) ∈ Adj[u]
7                Relax(u,v,w)
8        return true
9  }
```

?

O(|V|)

O(1)

O(|E|$log_2|V|$)

複雜度分析

- ✓ 取決於 priority_queue 的資料結構
  1. Linear Array：$O(|V|)$
  2. Red Black Tree：$O(log_2|V|)$
- ✓ 總複雜度
  - ✓ Linear Array：
    - $O(|V|^2 + |E|log_2|V|)$
  - ✓ Red Black Tree：
    - $O(|V|log_2|V| + |E|log_2|V|)$

Johnson's Algorithm

Johnson 提出修改權重的方式以增大 Dijkstra’s algorithm 的適用範圍

1. 存在 negative weight edge
2. 不存在負環 (即存在一最佳解)

讓原本的 negative weight edge 轉換成 non-negative weight edge

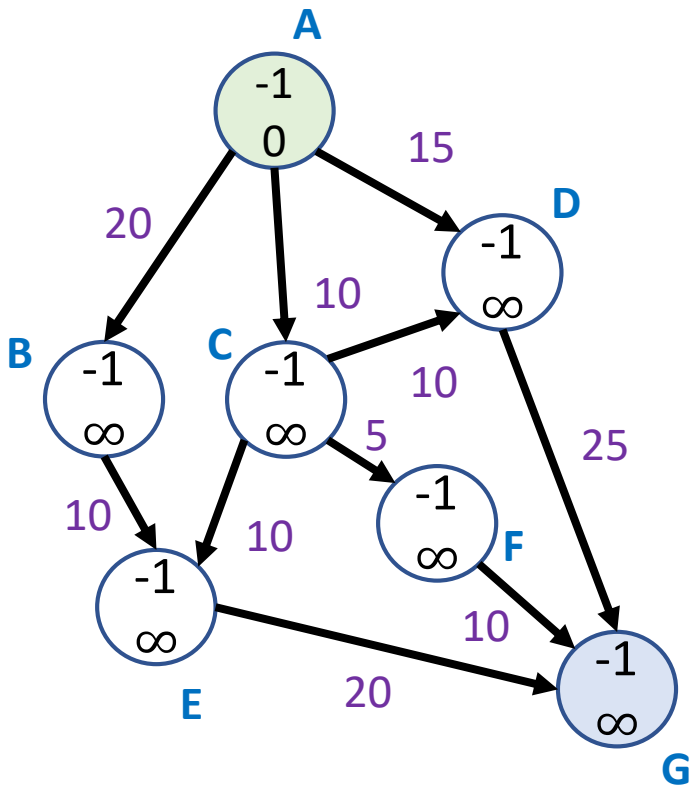
$$W'(u,v) = W(u,v) + \big(h(u) - h(v)\big)$$

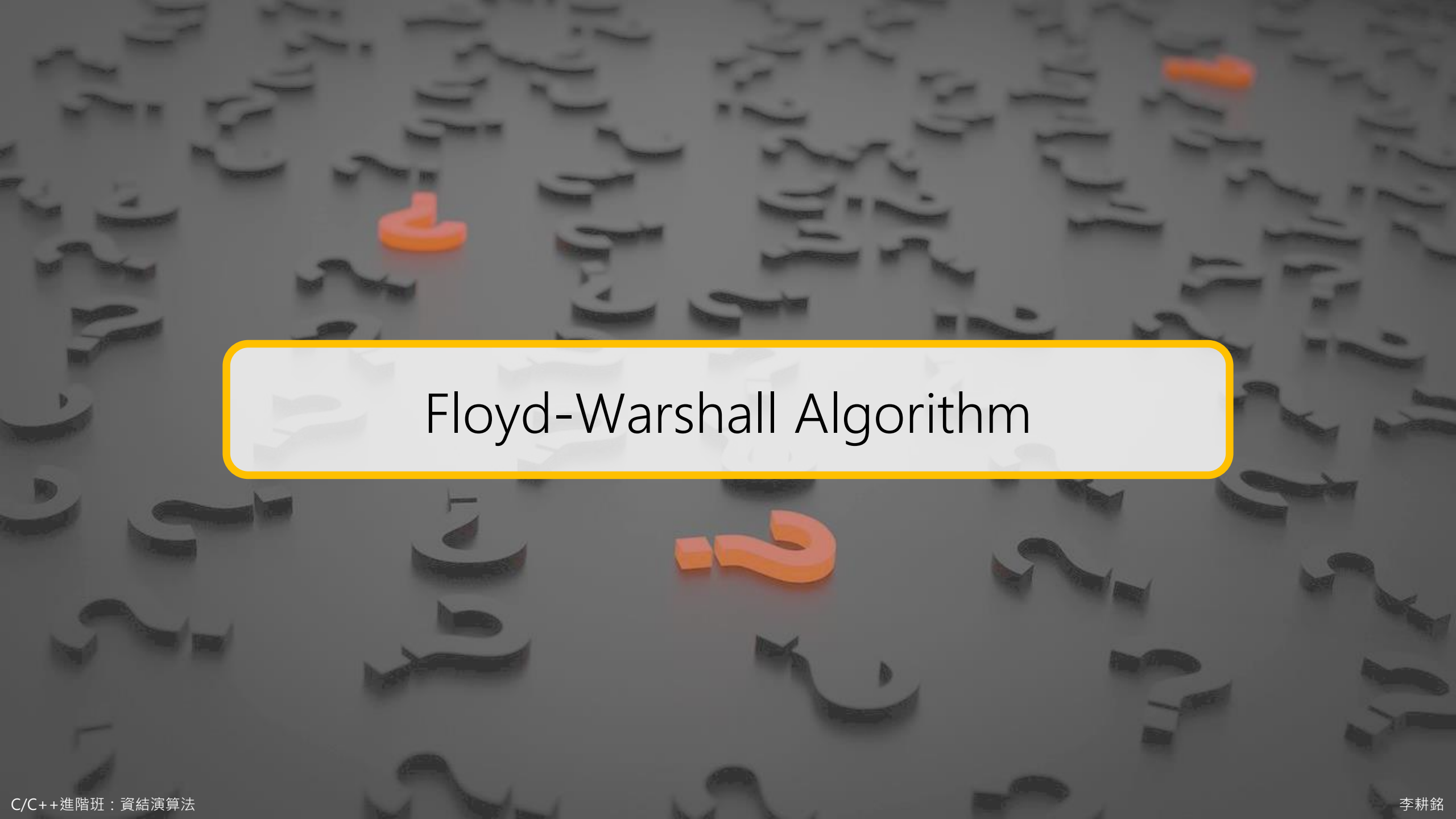
$h(u)\ and\ h(u)\ are\ the\ shortest\ paths\ from\ start\ point\ to\ u\ and\ v$
$h(u)\ and\ h(u)\ were\ obtained\ from\ Bellman-Ford\ Algorithm$

Practice 4

Mission

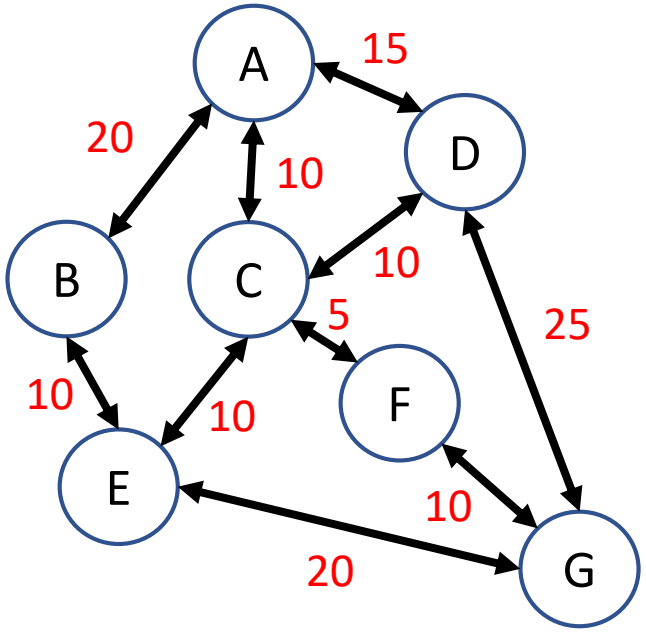
完成 Dijkstra‘s 演算法





Floyd-Warshall Algorithm

最短路徑問題



最短路徑問題的分類

1. Single-Source Shortest Path
   - 從單一頂點到其餘所有頂點的最短路徑
2. Single-Pair Shortest Path
   - 從單一頂點到單一頂點的最短路徑
   - Single-Source Shortest Path 的子問題
3. Single-Destination Shortest Path
   - 從所有頂點到特定頂點的最短路徑
   - 把 Edge 方向倒過來來，即是 Single-Pair Shortest Path
4. All-Pairs Shortest Path
   - 所有頂點到其餘所有頂點的最短路徑
   - 把所有頂點用 Single-Pair Shortest Path 算過一次即是

最短路徑問題

	BFS & DFS	Bellman-Ford	SPFA	DAG	Dijkstra's
Single-Source	$O(V+E)$	$O(VE)$	$O(VE)$	$O(V+E)$	$O(V log_2 V + \|E\|)$
All-Pairs	$O(V^2+VE)$	$O(V^2E)$	$O(V^2E)$	$O(V^2+VE)$	$O(V^2 log_2 V + VE)$
$E = O(V^2)$	$O(V^2+V^3)$	$O(V^4)$	$O(V^4)$	$O(V^3)$	$O(V^3)$
				不能處理環	不能處理負邊

Floyd-Warshall Algorithm

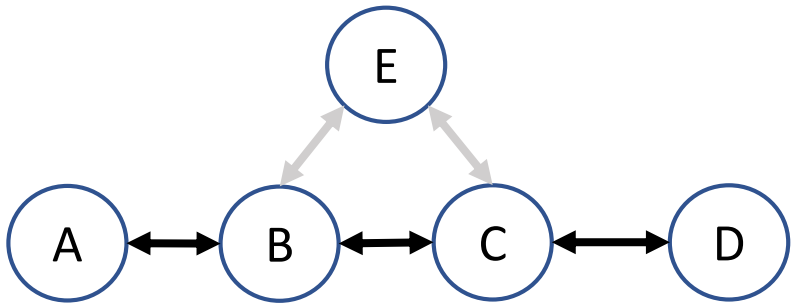
若 v_1, v_n 間存在最短路徑路徑 C ，則此路徑上的所有子路徑皆為最短

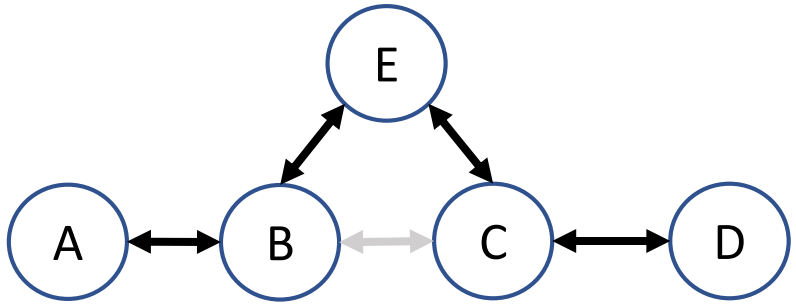
- ✓ 若 $0 \leq i \leq j \leq n$
- ✓ $(v_i, e_i, v_{i+1}, e_{i+1}, \ldots\ldots, e_{j-1}, v_j)$ 亦為 v_i, v_j 間的最短路徑 $p_{(v_i, v_j)}$

➢ 最短路徑必由所有經過頂點間的最小路徑所組成

➢ 如圖

1. 若 B-C 間的成本 > B-C-E：採用 B-E-C
2. 若 B-C 間的成本 < B-C-E：採用 B-C

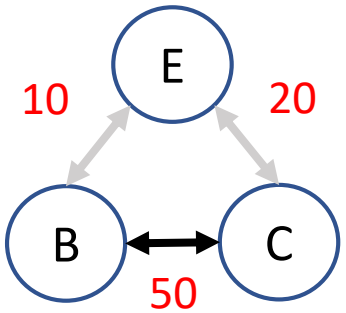


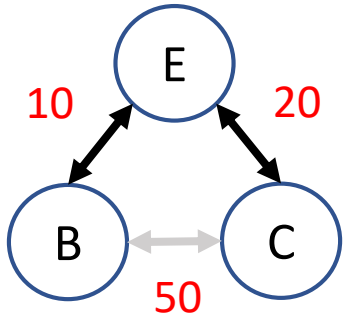


Floyd-Warshall Algorithm

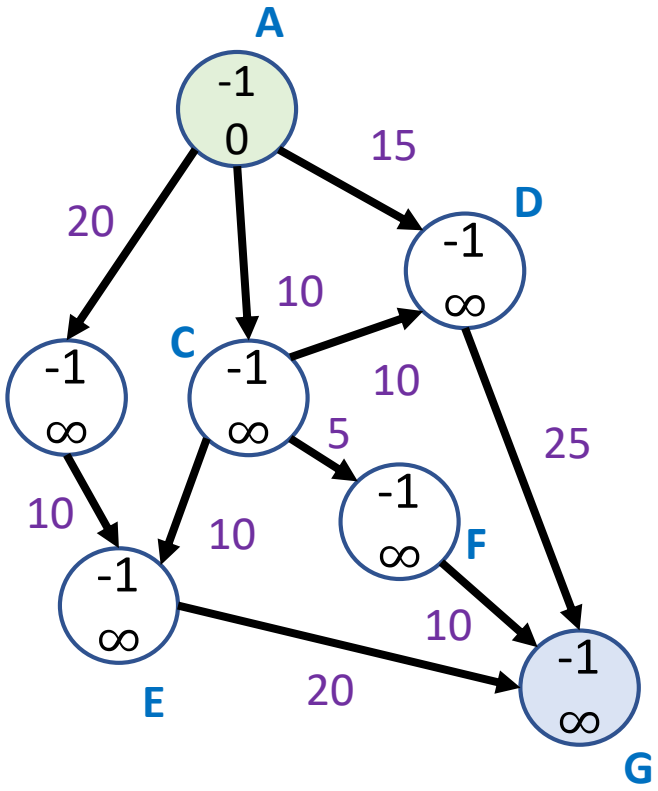
核心觀念：

- 依序加入中繼點，看距離會不會比較快
- 如果比原路徑短，就取代之 (與 Relax 概念相同)
  - ✓ $d_{ij}^{k} = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$
  - ✓ $d_{BC}^{k} = \min\left(d_{BC}^{k-1}, d_{BE}^{k-1} + d_{EC}^{k-1}\right) = \min(50, 10 + 20) = 30$
- 解法屬於動態規劃！
- 藉由最小化每個相異頂點-頂點間的距離，得到最小距離矩陣
- 最短路徑必由所有經過頂點間的最小路徑所組成





Floyd-Warshall Algorithm

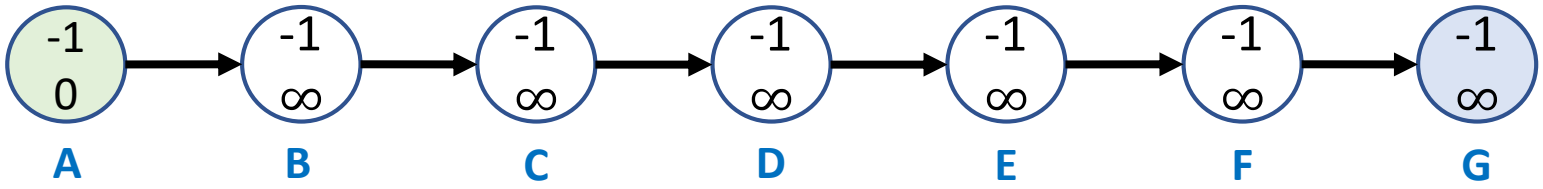


實作步驟：

1. 初始化一個 Adjacent matrix
2. 記錄每個頂點的 Predcessor
3. 加入一個點為中繼點，進行 $|V|^2$ 次更新
4. 重複步驟 3. |V| 次，依序加入每個點充當中繼點
5. 得到最短距離矩陣

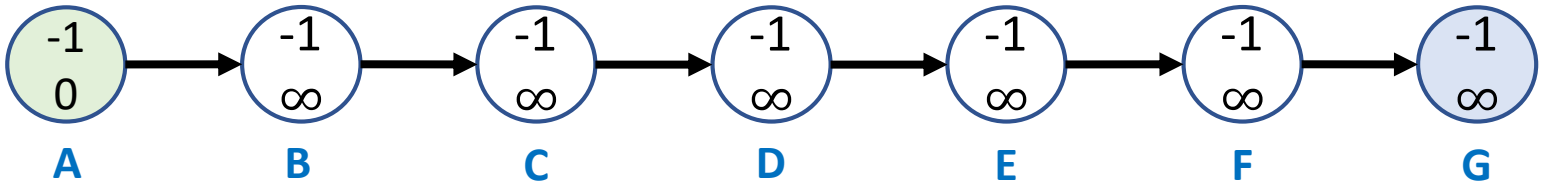
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	∞	∞	∞	∞	∞
B	∞	0	B,1	∞	∞	∞	∞
C	∞	∞	0	C,1	∞	∞	∞
D	∞	∞	∞	0	D,1	∞	∞
E	∞	∞	∞	∞	0	E,1	∞
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



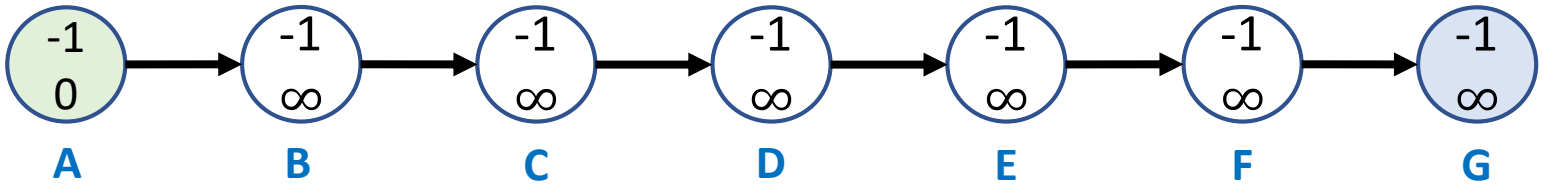
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	∞	∞	∞	∞	∞
B	∞	0	B,1	∞	∞	∞	∞
C	∞	∞	0	C,1	∞	∞	∞
D	∞	∞	∞	0	D,1	∞	∞
E	∞	∞	∞	∞	0	E,1	∞
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



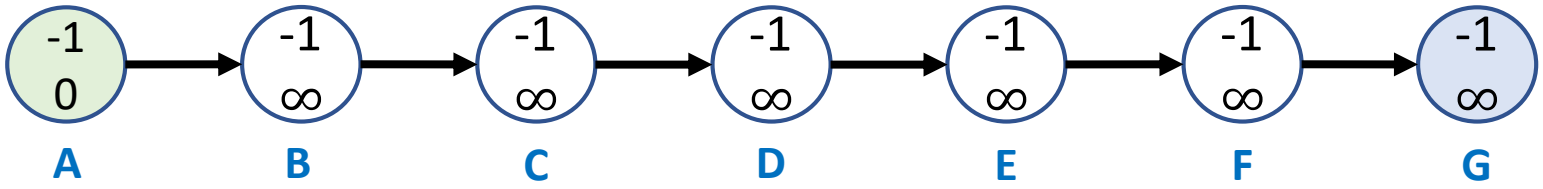
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	∞	∞	∞	∞	∞
B	∞	0	B,1	∞	∞	∞	∞
C	∞	∞	0	C,1	∞	∞	∞
D	∞	∞	∞	0	D,1	∞	∞
E	∞	∞	∞	∞	0	E,1	F,2
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



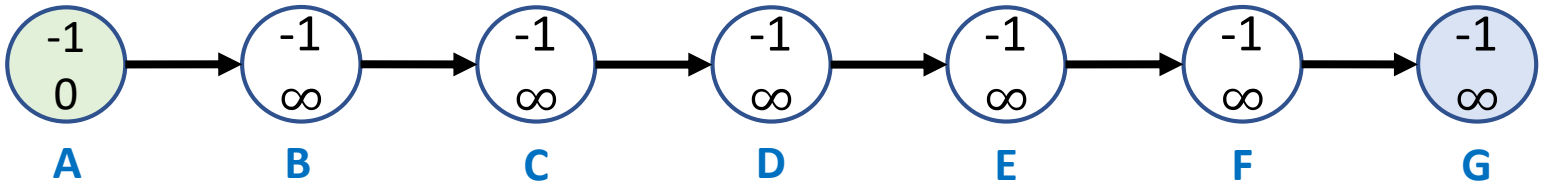
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	∞	∞	∞	∞	∞
B	∞	0	B,1	∞	∞	∞	∞
C	∞	∞	0	C,1	∞	∞	∞
D	∞	∞	∞	0	D,1	E,2	F,3
E	∞	∞	∞	∞	0	E,1	F,2
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



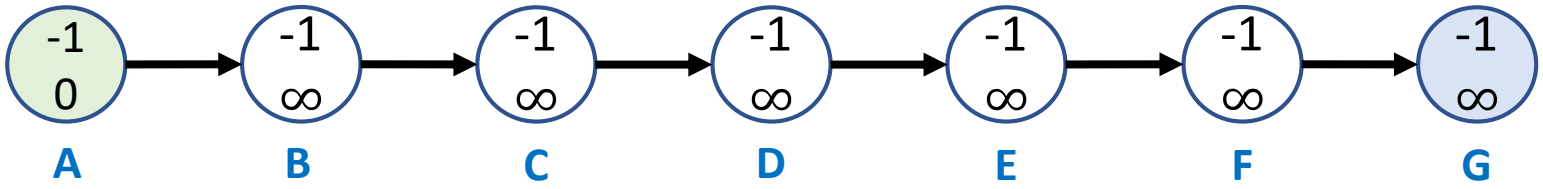
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	∞	∞	∞	∞	∞
B	∞	0	B,1	∞	∞	∞	∞
C	∞	∞	0	C,1	D,2	E,3	F,4
D	∞	∞	∞	0	D,1	E,2	F,3
E	∞	∞	∞	∞	0	E,1	F,2
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



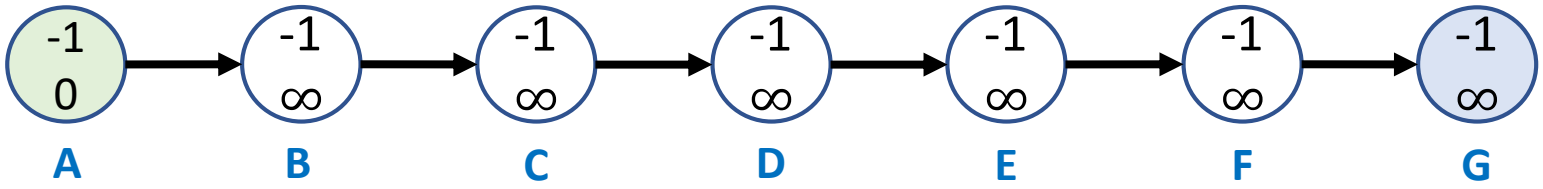
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	∞	∞	∞	∞	∞
B	∞	0	B,1	C,2	D,3	E,4	F,5
C	∞	∞	0	C,1	D,2	E,3	F,4
D	∞	∞	∞	0	D,1	E,2	F,3
E	∞	∞	∞	∞	0	E,1	F,2
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



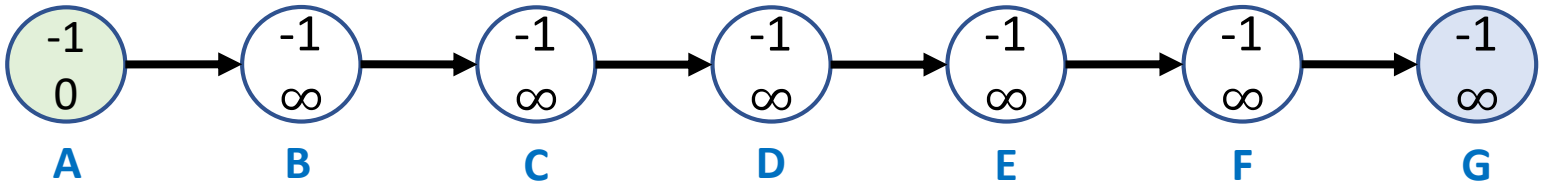
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	B,2	C,3	D,4	E,5	F,6
B	∞	0	B,1	C,2	D,3	E,4	F,5
C	∞	∞	0	C,1	D,2	E,3	F,4
D	∞	∞	∞	0	D,1	E,2	F,3
E	∞	∞	∞	∞	0	E,1	F,2
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	B,2	C,3	D,4	E,5	F,6
B	∞	0	B,1	C,2	D,3	E,4	F,5
C	∞	∞	0	C,1	D,2	E,3	F,4
D	∞	∞	∞	0	D,1	E,2	F,3
E	∞	∞	∞	∞	0	E,1	F,2
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0



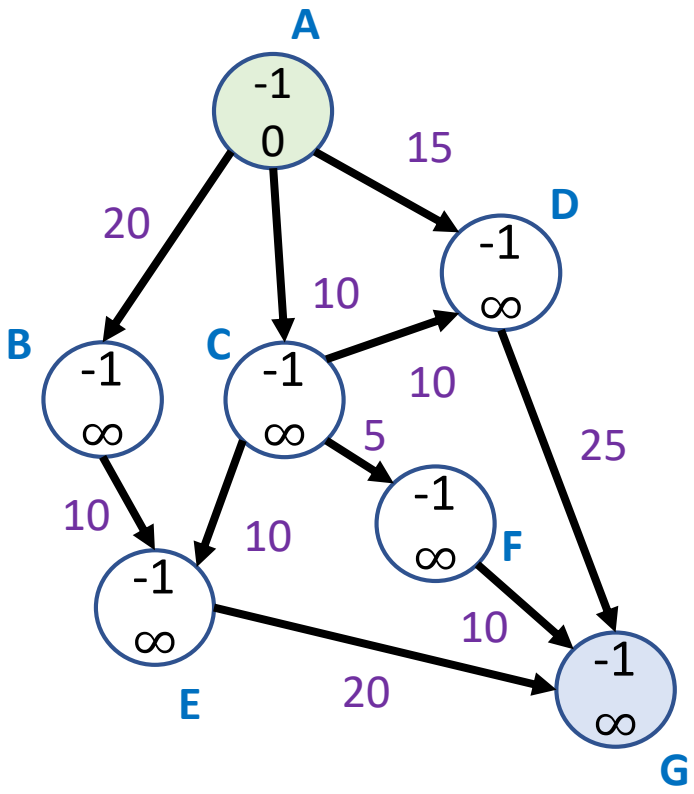
Floyd-Warshall Algorithm

From\To	A	B	C	D	E	F	G
A	0	A,1	B,2	C,3	D,4	E,5	F,6
B	∞	0	B,1	C,2	D,3	E,4	F,5
C	∞	∞	0	C,1	D,2	E,3	F,4
D	∞	∞	∞	0	D,1	E,2	F,3
E	∞	∞	∞	∞	0	E,1	F,2
F	∞	∞	∞	∞	∞	0	F,1
G	∞	∞	∞	∞	∞	∞	0

每次都會更新整個表格，複雜度：$\mathbf{O}(|\mathbf{V}|^2)$

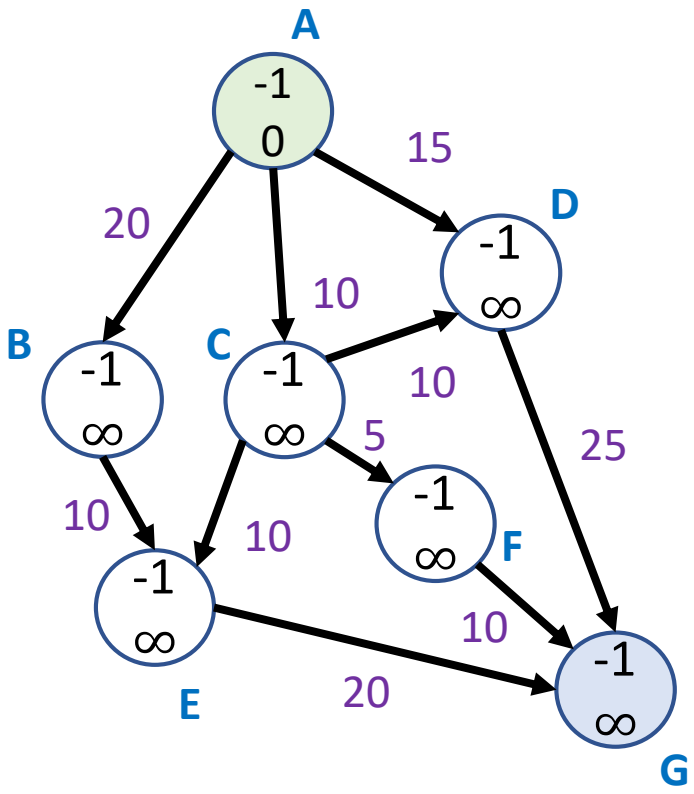
共更新 $|\mathbf{V}|$ 次，複雜度：$\mathbf{O}(|\mathbf{V}|^3)$

Floyd-Warshall Algorithm



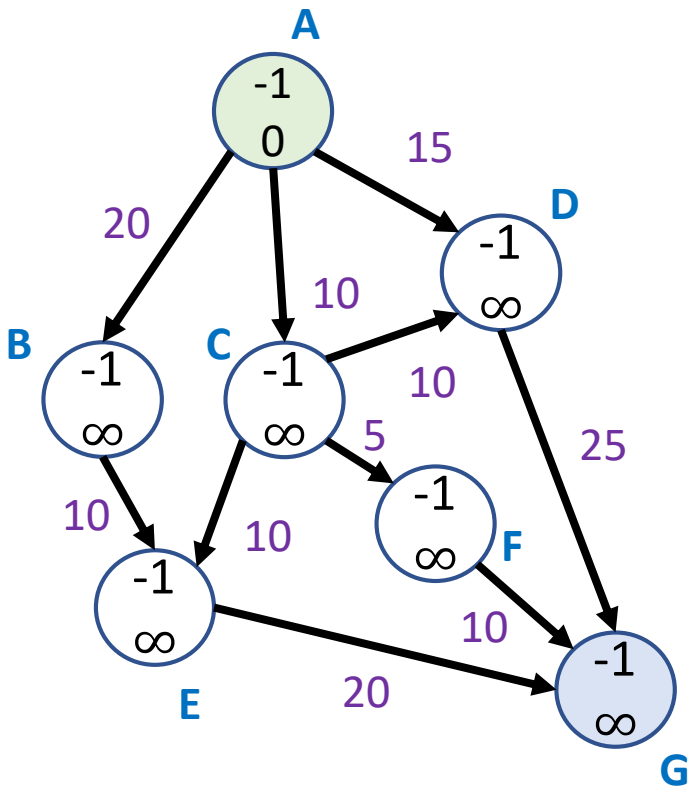
From\To	A	B	C	D	E	F	G
A	0	A,20	A,10	A,15	∞	∞	∞
B	∞	0	∞	∞	B,10	∞	∞
C	∞	∞	0	C,10	C,10	C,5	∞
D	∞	∞	∞	0	∞	∞	D,25
E	∞	∞	∞	∞	0	∞	E,20
F	∞	∞	∞	∞	∞	0	F,10
G	∞	∞	∞	∞	∞	∞	0

Floyd-Warshall Algorithm



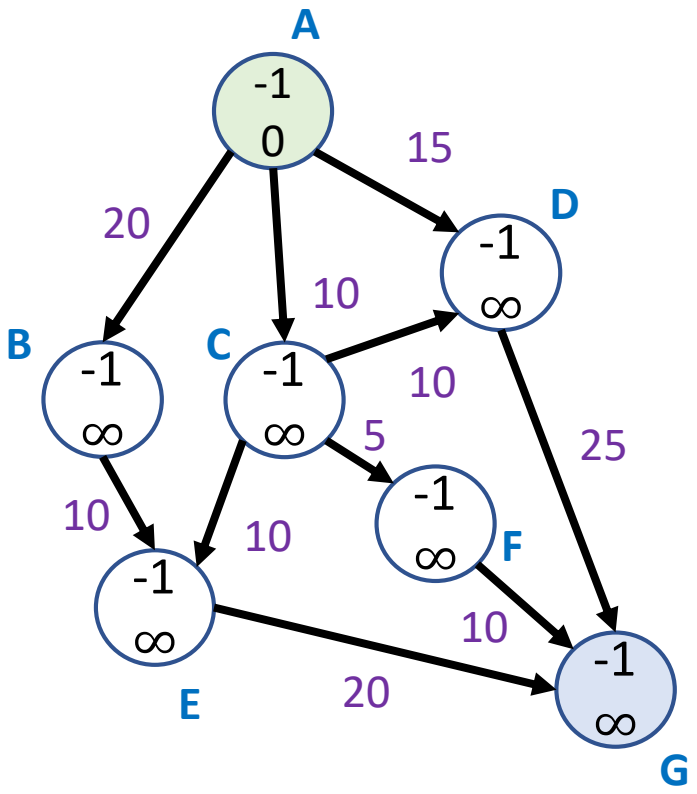
From\To	A	B	C	D	E	F	G
A	0	A,20	A,10	A,15	B,30	∞	∞
B	∞	0	∞	∞	B,10	∞	∞
C	∞	∞	0	C,10	C,10	C,5	∞
D	∞	∞	∞	0	∞	∞	D,25
E	∞	∞	∞	∞	0	∞	E,20
F	∞	∞	∞	∞	∞	0	F,10
G	∞	∞	∞	∞	∞	∞	0

Floyd-Warshall Algorithm



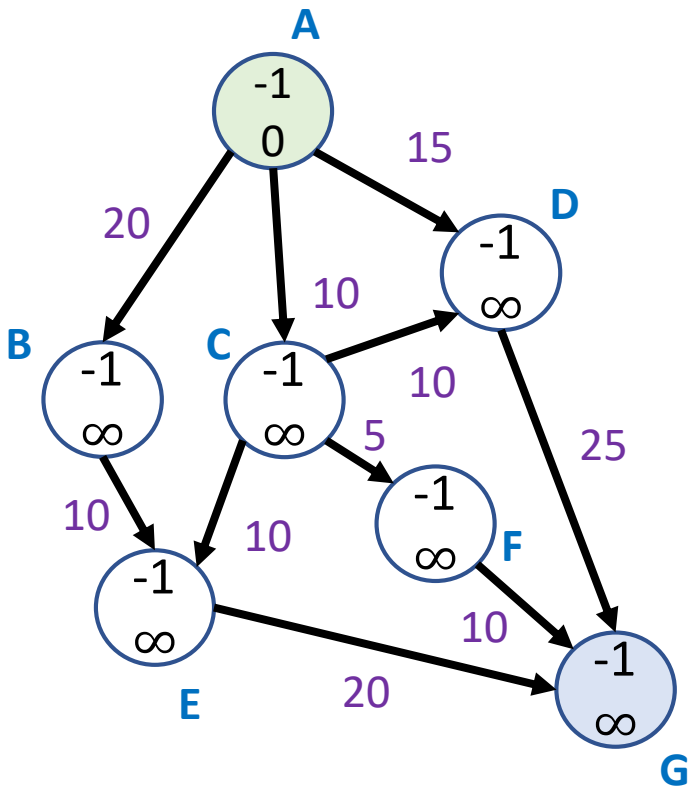
From\To	A	B	C	D	E	F	G
A	0	A,20	A,10	A,15	C,20	C,15	∞
B	∞	0	∞	∞	B,10	∞	∞
C	∞	∞	0	C,10	C,10	C,5	∞
D	∞	∞	∞	0	∞	∞	D,25
E	∞	∞	∞	∞	0	∞	E,20
F	∞	∞	∞	∞	∞	0	F,10
G	∞	∞	∞	∞	∞	∞	0

Floyd-Warshall Algorithm



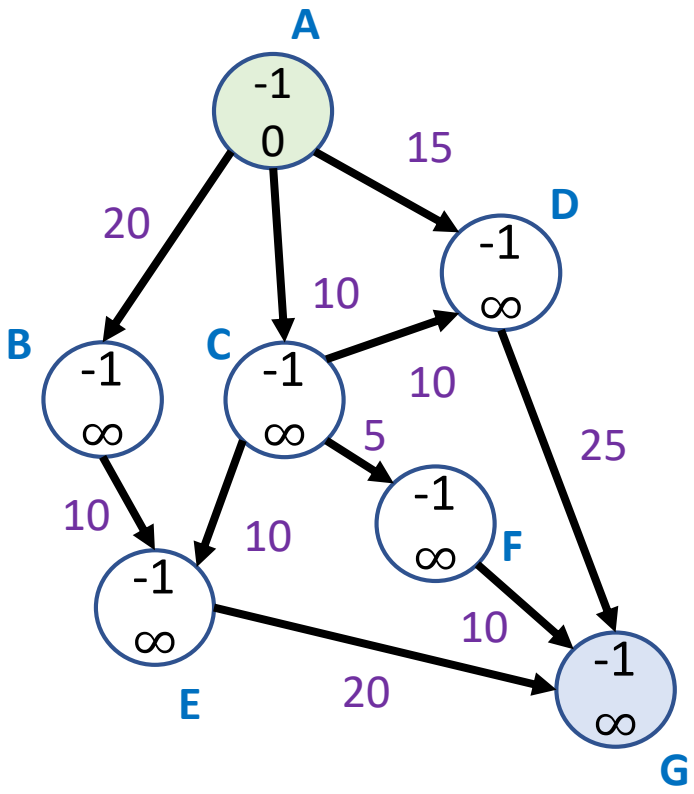
From\To	A	B	C	D	E	F	G
A	0	A,20	A,10	A,15	C,20	C,15	D,40
B	∞	0	∞	∞	B,10	∞	∞
C	∞	∞	0	C,10	C,10	C,5	D,35
D	∞	∞	∞	0	∞	∞	D,25
E	∞	∞	∞	∞	0	∞	E,20
F	∞	∞	∞	∞	∞	0	F,10
G	∞	∞	∞	∞	∞	∞	0

Floyd-Warshall Algorithm



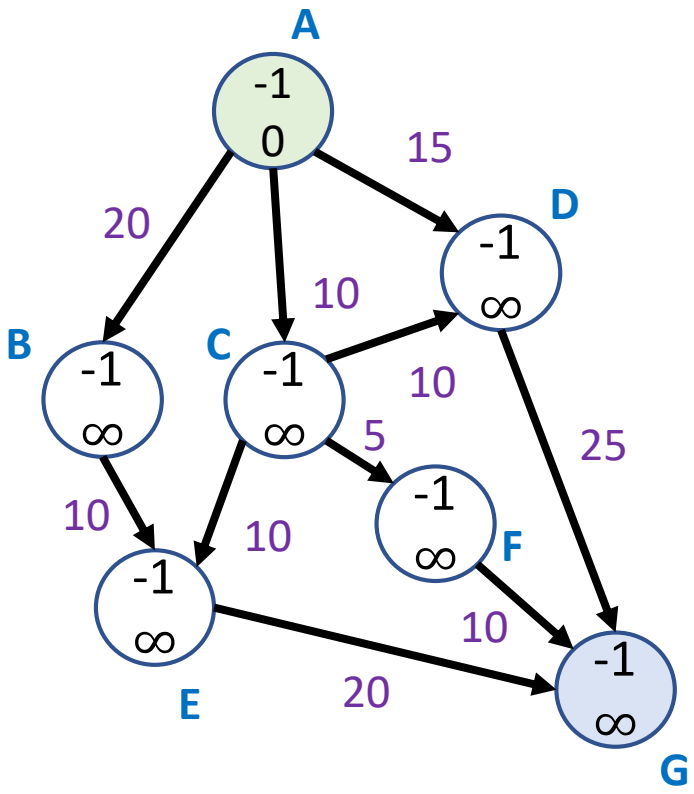
From\To	A	B	C	D	E	F	G
A	0	A,20	A,10	A,15	C,20	C,15	D,40
B	∞	0	∞	∞	B,10	∞	**E,30**
C	∞	∞	0	C,10	C,10	C,5	**E,30**
D	∞	∞	∞	0	∞	∞	D,25
E	∞	∞	∞	∞	0	∞	E,20
F	∞	∞	∞	∞	∞	0	F,10
G	∞	∞	∞	∞	∞	∞	0

Floyd-Warshall Algorithm



From\To	A	B	C	D	E	F	G
A	0	A,20	A,10	A,15	C,20	C,15	F,25
B	∞	0	∞	∞	B,10	∞	E,30
C	∞	∞	0	C,10	C,10	C,5	F,15
D	∞	∞	∞	0	∞	∞	D,25
E	∞	∞	∞	∞	0	∞	E,20
F	∞	∞	∞	∞	∞	0	F,10
G	∞	∞	∞	∞	∞	∞	0

Floyd-Warshall Algorithm



From\To	A	B	C	D	E	F	G
A	0	A,20	A,10	A,15	C,20	C,15	F,25
B	∞	0	∞	∞	B,10	∞	E,30
C	∞	∞	0	C,10	C,10	C,5	F,15
D	∞	∞	∞	0	∞	∞	D,25
E	∞	∞	∞	∞	0	∞	E,20
F	∞	∞	∞	∞	∞	0	F,10
G	∞	∞	∞	∞	∞	∞	0

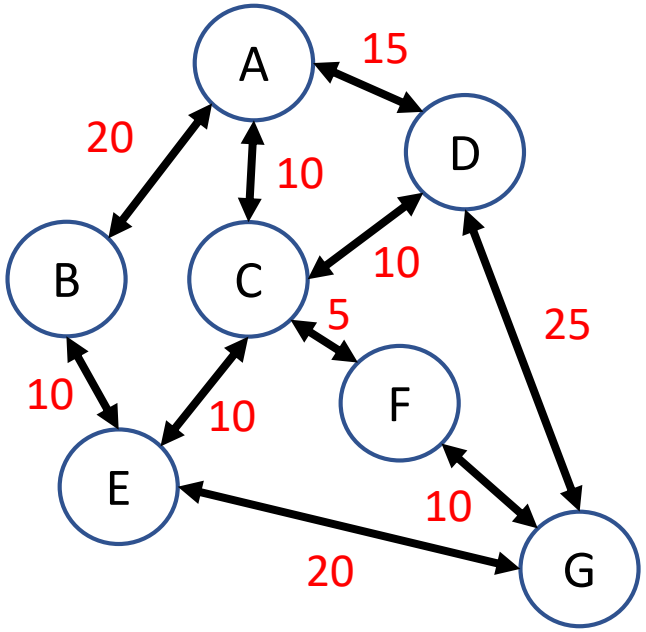
Floyd-Warshall Algorithm

```
1  Floyd-Warshall(G,w,s){
2      Initialize D
3      for k = 1~|V|:
4          for i = 1~|V|:
5              for j = 1~|V|:
6                  d_ij^k = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7      return D
8  }
```

$O(|V|^3)$

複雜度：$O(|V|^3)$

Floyd-Warshall Algorithm



偵測負環：檢查矩陣中對角的值是否為負！

- 從該點出發，回到該點的距離<0
- 代表該點中存在負環
- 可無限重複此一負環，不存在最短路徑！

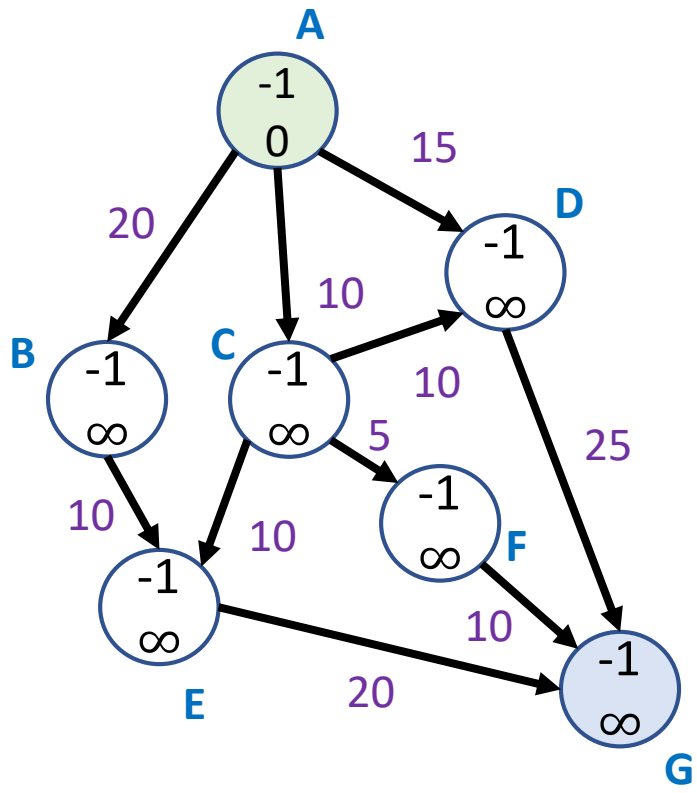
Floyd-Warshall Algorithm

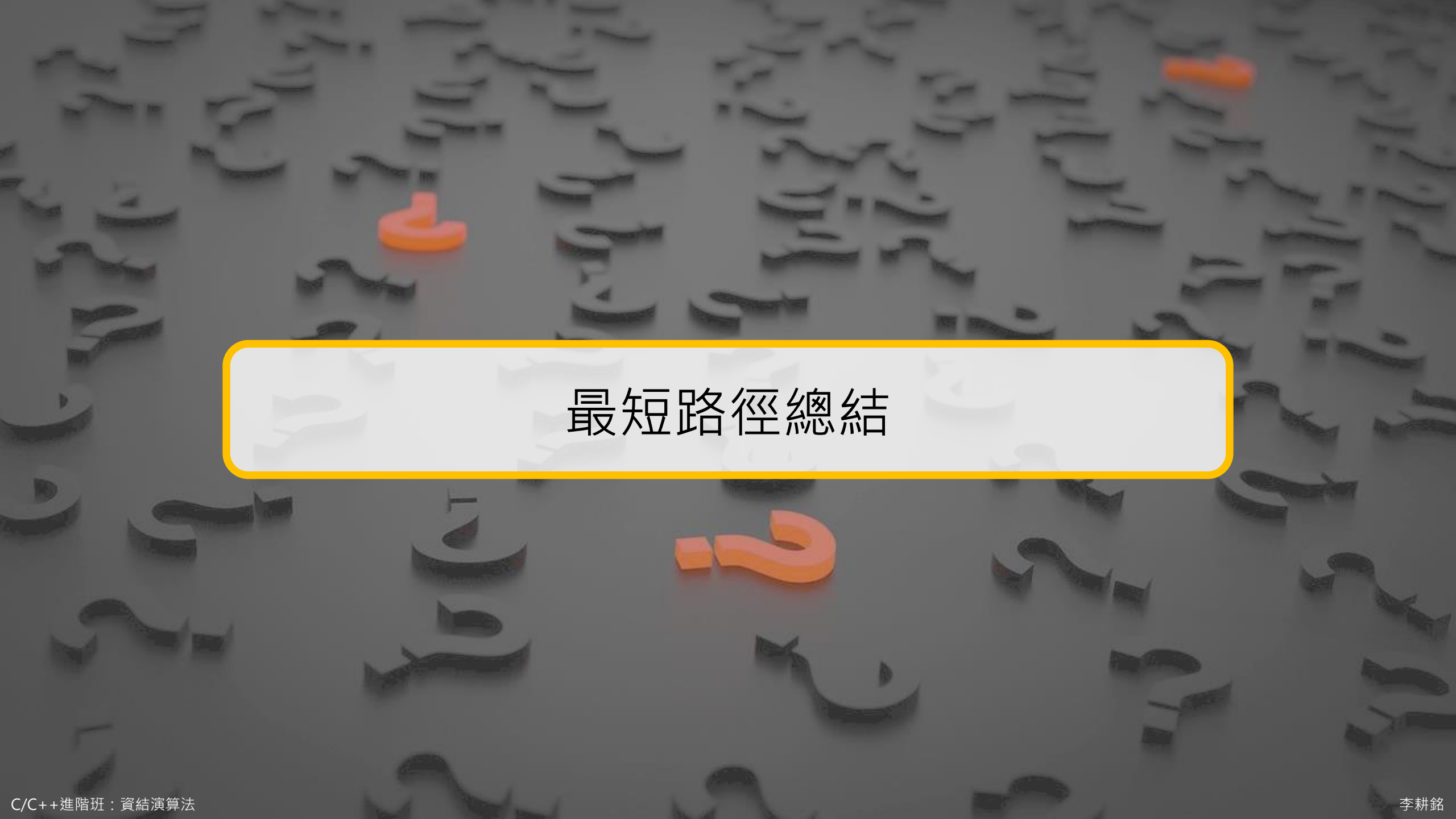
```
1  Floyd-Warshall(G,w,s){
2        Initialize D
3        for k = 1~|V|:
4              for i = 1~|V|:
5                    for j = 1~|V|:
6                          d_ij^k = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7        for i = 1~|V|:
8              if D[i,i] < 0
9                    return False
10       return D
11 }
```

Practice 5

Mission

完成 Floyd-Warshall 演算法





最短路徑總結

最短路徑問題

	BFS	Bellman-Ford	SPFA	DAG	Dijkstra's	Floyd-Warshall
Complexity	$O(V+E)$	$O(VE)$	$O(VE)$	$O(V+E)$	$O(V^2+E)$	$O(V^3)$
Weighted Edge	X	O	O	O	O	O
Negative Edge	X	O	O	O	X	O
Negative Cycle	All able to detect negative cycle.					
Positive Cycle	O	O	O	X	O	O
評析	只適用 Unweighted edge	無限制的暴力解 唯一能處理負環	優化後的暴力解 唯一能處理負環	適用有向無環	不能用在負邊	All-Pairs

最短路徑問題

1. 當無權重或權重一樣，使用 BFS
2. 當權重不一樣，但皆為正，使用 Dijkstra.
3. 當權重有負的，但沒有環，使用 DAG
4. 當權重有負的，但有環，使用 SPFA
5. 當需要印出負環的順序，使用 Bellman-Ford
6. All-pairs問題，使用 Floyd-Warshall