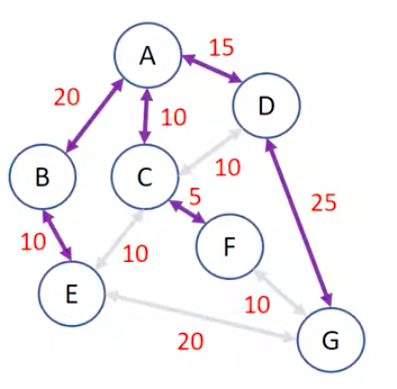
Ch14. 最短路徑

本章介紹最短路徑問題，首先介紹這個問題的定義，以及一些可用來解決此問題的技巧。

再來，介紹與最短路徑有關的重要演算法：Bellman-Ford、DAG、Dijkstra、Floyd-Warshall。

1. 用 BFS 求解最短路徑問題

在廣度優先搜尋的章節，有提過 BFS 可用來尋找最短路徑。但是 BFS 找到的是「邊數最少」、而非「各邊權重總和最小」的路徑，因此當各邊權重不同（比如開車走每條道路的時間成本不同），而欲求的是權重總和最小的路徑時，BFS 就不適用。

上圖中，從 A 點出發進行 BFS，欲求到 G 點的路徑。第一輪會從 A 點向外擴展到 B、C、D 三點，第二輪擴展到 E、F、G 三點，得到「邊數最少」的路徑為 。

但是事實上， 經過的兩條邊權重和為 ，而 雖然經過三條邊，但權重總和僅 ，可見 BFS 只有在每條邊權重看作相等時，才能解決定義為「權重和最小」的最短路徑問題，每條邊權重不同時，必須尋找其它方法。

1. 最短路徑問題簡介

（1）最短路徑問題的定義

在圖上尋找兩頂點的最短路徑，其中：

1. 路徑 ，由頂點與連接其間的邊組成
2. 邊 的權重以 表示
3. 路徑 的權重 ，即路徑上各邊權重的和
4. 的最短距離 定義為

上式表示當頂點 到 之間有路徑時，最短路徑是所有路徑之間權重最小的那個，反之，若 到 之間沒有路徑，則定義 的值為無限大。

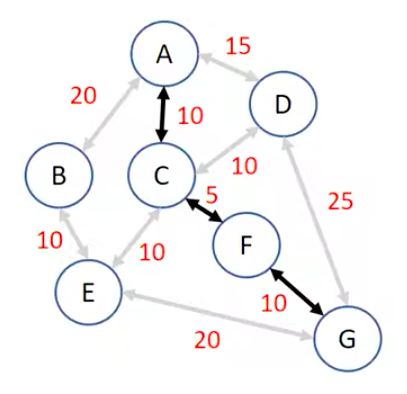
最短路徑的應用很多，諸如安排管線、縮短交通時間、寄信、瓦斯抄錶等等，都需要尋找最短路徑，以找到兩點之間最節省時間 / 金錢的往來方式。另外，在最近相當熱門的自然語言處理領域中，要透過知識圖譜或規則系統來產生過程最短的推論，一樣會需要尋找最短路徑。

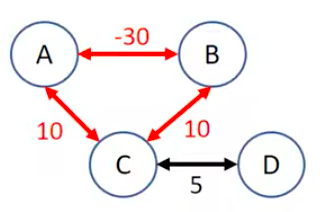
之前有提過，可以透過將無向圖上的每條邊視作兩條相反的邊來轉為有向圖，但有向圖則無法轉為無向圖，同時，邊上無權重時可以全部指定相同權重來轉換為有權重圖，但有權重的圖則無法轉換為無權重的圖。

因此，只要某個演算法解決了「有權重」的「有向圖」上最短路徑的問題，就同樣可以應用來尋找其他類型的圖上的最短路徑。

（2）最短路徑問題的分類

1. Single-Source：找尋從特定一個頂點到其餘所有頂點的最短路徑
2. Single-Pair：從特定一個頂點走到另一個特定頂點的最短路徑，屬於 Single-Source 的子問題（因為一個 Single-Source 的解會包含該頂點出發的所有 Single-Pair 的解）
3. Single-Destination：從所有其餘頂點到一個特定頂點的最短路徑，如果把圖上的所有邊方向倒過來，就與 Single-Source 相同
4. All-Pairs：所有頂點到所有其餘頂點的最短路徑，即求出所有頂點對之間的 Single-Pair

（3）最短路徑的性質

頂點 和 之間的最短路徑 中必不存在環：最短路徑會是一個以起點 為根節點的樹，而因為最多只會經過所有頂點一次（否則即有環），所以最多由 條邊組成。

為何最短路徑中一定不存在環呢？分成兩個情況討論：若環的各邊權重和為正，那麼走過這個環不僅回到原處沒有前進，而且成本還增加，所以不會比不走環時更有利；若環的各邊權重和為負，那麼不斷的在環當中繞圈，就可以讓成本降為負無限大，只要任何路徑可以走到這個環內，都同為最短路徑，並沒有求解的意義。

再來，若頂點 和 間存在最短路徑 ，則此路徑上所有的「子路徑」皆為最短，比如 是由 A 出發，依序經過 B、C 兩點再到達 D，那麼 A 與 C 之間的最短路徑一定是 ，B 與 D 之間的最短路徑一定是 ，這對其餘所有頂點對之間也都成立。



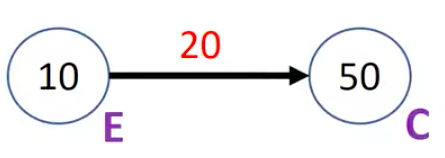
比如上圖中，如果 的成本比 和 的成本和來得大，代表 之間的最短路徑並非 ，而應該是 ，但若如此，則 的成本一定大於 ， 就不會是 A 到 D 的最短路徑。反過來說，若已知 是 A 與 D 間的最短路徑，那麼 B 和 C 點間不會存在比走邊 成本還低的路徑。

一般來說，若 ，則 的子路徑 也是 與 兩點間的最短路徑，即 。

（4）Relaxation

處理最短路徑問題時，以一個陣列 來記錄起點與每個其餘頂點的距離，另外，以一個陣列 來記錄每個頂點往起點方向的上一個頂點為何。

是比較走到同一個頂點的兩個路徑之間何者成本（權重和）較小，並依此更新該頂點對應的 。



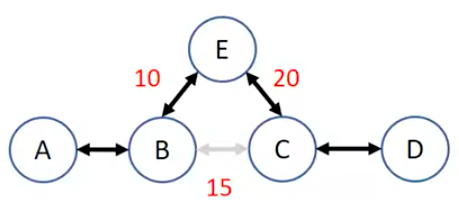
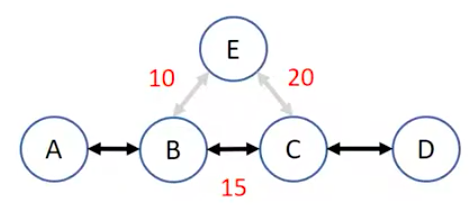
比如前幾輪處理中，發現從起點 A 出發要走到頂點 C 的成本至少需要 50 單位、走到頂點 E 的成本至少需要 10 單位，但是在這個前提下，又發現 的權重只有 20，也就是說，若不走先前找到的某個花費 50 由 A 到 C 的路徑，而改為先花費 10 由 A 走到 E，再花費 20 走 EC 這條邊，就可以把 A 到 C 的成本降為 。

不斷對所有頂點與它們的所有出邊進行 Relax，就可以找到頂點對之間的最短路徑，而根據前述性質，起點到目標頂點的最短路徑就是由這些子路徑中的某些共同組成。

|  |  |
| --- | --- |
| Relax 的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Relax (E, C, weight){    // 比較原先 C 的成本，與先走到 E 再走邊 EC 何者較高  if (Distance[C] > Distance[E] + weight of E-C){  // 若先到 E 再走 EC 較好，更新 C 與頂點的距離 Distance[C]  Distance[C] = Distance[E] + weight of E-C;  // 記錄路徑上頂點 C 的上一個頂點是 E  Predecessor[C] = E;  }  } |

由 Relax 的概念，可以看出幾個性質：

1. Triangle inequality

，可由 Relax 的條件推知

比如經過 Relax 後，

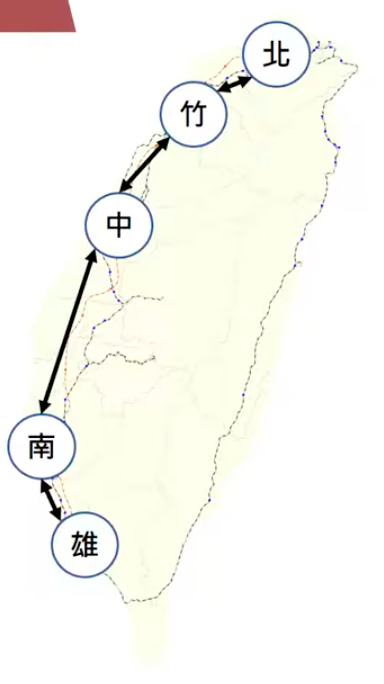
1. Convergence property

如果由 A 到 C 的最短路徑 會先經過頂點 B，且 ，代表已經找到了 AB 之間的最短路徑，那麼此時對 B 到 C 的所有路徑進行 Relax，就可以讓 ，亦即就可以找到 AC 之間的最短路徑。

舉例來說，若「已知」由高雄出發到紐約的最短路徑中包含「左營高鐵 -> 桃園機場」這段，那麼對從桃園機場到紐約的所有可能路徑進行 Relax 後，就可以找到高雄到紐約整個行程的最短路徑（會由「左營高鐵 -> 桃園機場」與「桃園機場到紐約的最短路徑」兩段子路徑組成）。

（5）Path-relaxation

根據 Convergence property，如果「已知」 到 之間的最短路徑必定經過 、、、...、 這些點，那麼依序執行 、、...、 之後，就會得到 到 的最短路徑。



比如「已知」從台北開車到高雄，中間必定經過新竹、台中、台南幾個城市，那麼依序對「台北 -> 新竹」、「新竹 -> 台中」、「台中 -> 台南」、「台南 -> 高雄」進行 Relax，就會得到台北到高雄的最短路徑距離。

1. Bellman-Ford Algorithm

Bellman-Ford 演算法是每一輪都對「所有邊」進行 Relax，因為沒有按照剛才提過的「正確順序」（如新竹、台中、台南的順序）來進行，所以必須進行共 輪以確保每個邊會在適當的條件下被 Relax 過，虛擬碼如下：

|  |  |
| --- | --- |
| Bellman-Ford 的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Bellman-Ford(G,w,s){  initialize(G,s)  // 對所有邊 Relax |V|-1 輪  for i=1 to |V|-1  for edge(u,v) E  Relax(u,v,w)  // 檢查負迴圈  for edge(u,v) in Graph  if d[v]>d[u]+w(u,v)  return false  return true  } |

複雜度分析：

1. 初始化：
2. 一輪中，對所有邊進行 Relax：
3. 重複 輪：
4. 總和：

（1）Bellman-Ford 演算法的步驟

1. 初始化所有 Vertex（initialize 函式）

* 設為 -1，代表目前沒有 Predecessor
* 設為 ，起點的 設為 0

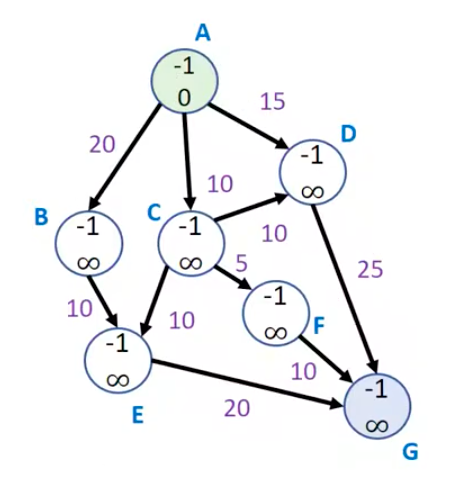
B. 對所有 Edge 進行 Relax

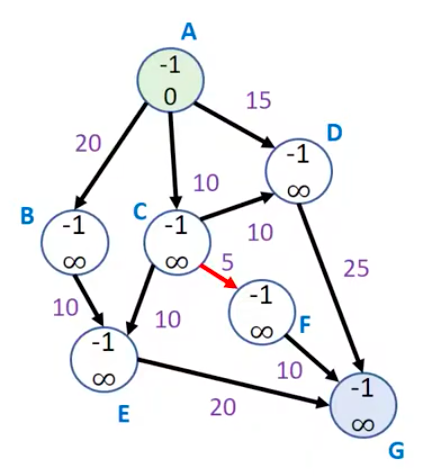
C. 重複上一步驟 次，以確保得到最短路徑

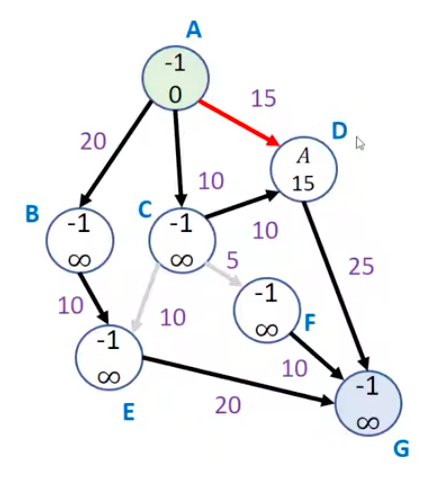
D. 檢查沒有負迴圈

因為 Edge relax 沒有按照最短路徑的順序，所以必須進行 輪，是最短路徑可能有的最長邊數。也就是說，Bellman-Ford 演算法類似一種暴力解。

（2）Bellman-Ford 演算法的進行過程

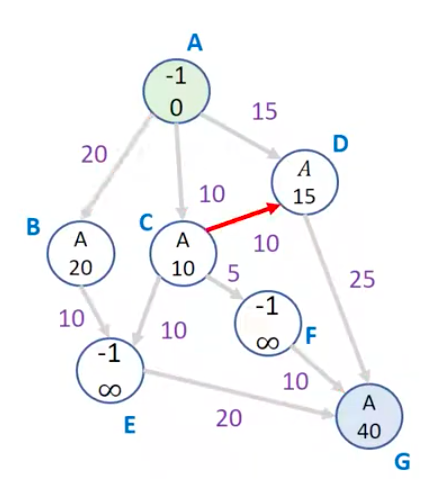


第一輪 Relax 中，會對上圖的所有邊進行一次 Relax，但順序是任意選定，假設首先對 這條邊進行 Relax，因為 C 點的 是無限大、加上 的成本 5 後，並沒有比此時 E 點的 無限大來的小，因此 E 點的距離不會被更新，類似的，如果接下來對 進行 Relax，一樣不會更新 G 點的 。



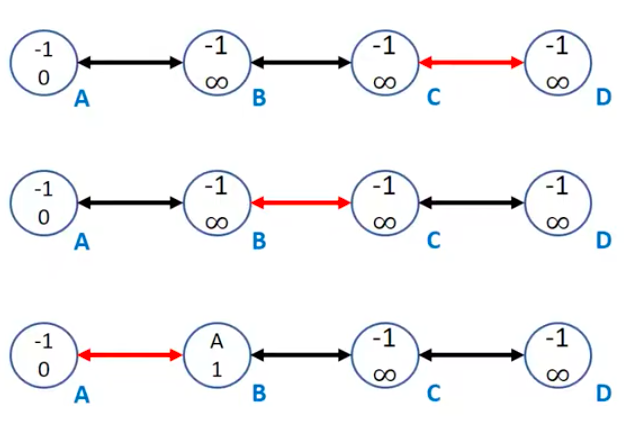
等到第一輪中對邊 進行 Relax 時，因為 A 的 在初始化時設為 0，加上邊 的成本 15 後，仍然比 D 點的 無限大來得小，因此 D 點的 會被更新為 15。

類似的，第一輪中，B、C、D 點的距離會分別被更新為 20、10、15，其他頂點的距離是否被更新則視任意選定的 Relax 順序而定。



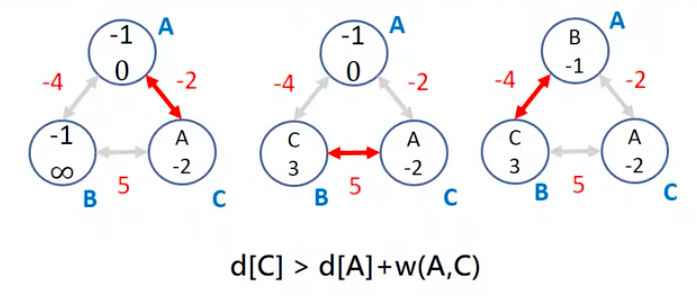
比如第一輪中，如果對 進行 Relax 的時間點在邊 之後，則對 的 Relax 會使點 G 的 被更新為 40，然而 40 並不是實際的最短路徑成本 25，需要接著往下進行，直到進行 （頂點數-1）輪後，才能確保 就是最短路徑的成本。

Worst Case 下，每一輪對所有邊 Relax 的順序正好與實際最短路徑的順序相反，這會導致一輪 Relax 後只會完成最短路徑上一個邊的更新，而整個最短路徑最多可能有 個邊，就使得對所有邊的 Relax 必須進行 輪。



上圖中，如果實際的最短路徑是 ，但是每一輪中都是按照 、、 的順序來 Relax，那麼第一輪只有對 Relax 時會更新到 B 的距離，第二輪對 Relax 時才會更新到 C 的距離，第三輪對 Relax 時才會更新到 D 的距離。

完成 Relaxation 後，還要檢查圖中是否有負環。沒有負環時，根據 Triangle inequality，每個邊都會滿足 ，意即先走到 u 再走 到 v，一定會比 v 的距離來得大，否則圖中必有負環。



上圖中存在負環 ABCA，依序對 、、 三條邊進行 Relax 後，發現 ，因此發現負環存在，讓函式回傳 false 以利後續處理。

（3）實作 Bellman\_Ford 演算法

|  |  |
| --- | --- |
| Relax 函式 | |
| 1  2  3  4  5  6  7  8 | bool graph::relax(int from, int to, float weight){  if (distance[to] > distance[from] + weight){  distance[to] = distance[from]+weight;  predecessor[to] = from;  return true;  }  return false;  } |

|  |  |
| --- | --- |
| Bellman\_Ford 演算法 | |
| 1  2  3  4  5  6  7  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | void graph::initialize\_shortest\_path(int initial){  distance.clear();  predecessor.clear();  for (int i=0 ; i<number\_vertex ; i++){  // 距離設為無限大，要 include <limits>  distance[i] = numeric\_limits<float>::max();  // predecessor 設為 -1  predecessor[i] = -1;  }  // 起點的距離初始化為 0  distance[initial] = 0;  }  bool graph::Bellman\_Ford(int initial){  // 傳入的 initial 是起點編號  initialize\_shortest\_path(initial);  list<edge\*>::iterator it;  // 要重複 |V|-1 輪  for (int i=0 ; i<number\_vertex-1 ; i++){  // 每一輪對所有邊進行 relax  for (int j=0 ; j<number\_vertex ; j++){  for (it=edges[j].begin() ; it!=edges[j].end() ; it++){  relax(j, (\*it)->to, (\*it)->weight);  }  }  }  // 檢查負迴圈  for (int j=0 ; j<number\_vertex ; j++){  for (it=edges[j].begin() ; it!=edges[j].end() ; it++){  if (distance[(\*it)->to] > distance[j] + (\*it)->weight)  return false;  }  }  return true;  } |

1. SPFA (Shortest Path Faster Algorithm)

因為 Bell-Ford 演算法會浪費很多時間計算不會被更新的點（比如距離為無限大的點指向另一個距離為無限大的點的情形），因此一種改良方式就是挑出先前有被更新過距離的頂點，只對這些頂點的出邊進行 Relax。

（1）SPFA 的步驟

A. 把起點 Push 到 Queue 中

B. 從 Queue 中 Pop 出一筆資料處理

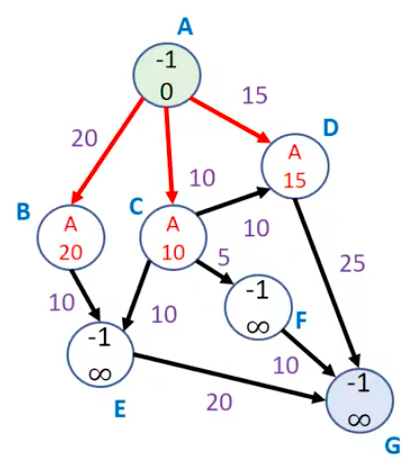
C. 對該資料的所有出邊進行 Relax

D. 把被出邊指到，因此距離被更新的頂點放到 Queue 中

E. 重複步驟 ，直到 Queue 為空

F. 同樣需要檢查負環

|  |  |
| --- | --- |
| SPFA 的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | SPFA(G,w,s){  initialize(G,s)  push s to Queue(Q)  while Q is not empty:  // 從 queue 中取出有被更新過距離的點  u = Q.pop()  // 針對 u 的出邊進行 relax  for each edge in Adj[u]:  if (Relax(u,v,w))  update[v] += 1  // 有負環時會導致無窮迴圈  // 某個頂點被 Relax 的總次數 >= |V| 時，就代表有負環  if update[v] == |V|  return false  push v into Q  return true  } |

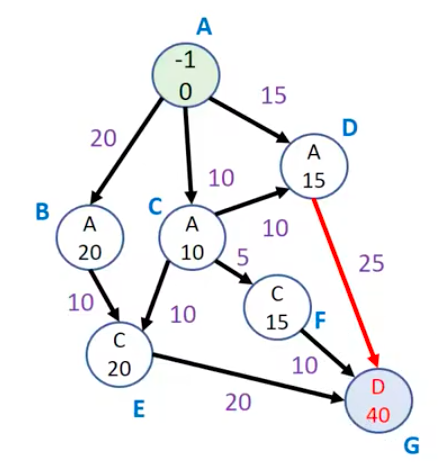
（2）SPFA 的進行過程

初始化之後，起點 A 被放到 中，取出 A 並對 A 的所有出邊作 Relax，這會使 B、C、D 三點的距離被更新且被放到 Queue 中，使得 。

再來，取出 Queue 中的 B 並對出邊作 Relax，使得 E 點的距離被更新，且 變為 。

取出 中的 C 並對出邊作 Relax，E、F 兩點的距離被更新，並被放到 中，CD 邊則不會使 D 點的距離變小，因此 D 點不會被加到 中，此時 。

再對 最前方 D 的所有出邊作更新，G 點的距離被更新並放到 中，。



繼續往下處理，對 E 的出邊作 Relax 時不會更新，對 F 的出邊作 Relax 時，G 點的距離被更新為 25，且 ，而 G 點沒有出邊，因此處理完後 為空，跳出迴圈。

（3）SPFA 的複雜度

Worst Case 發生在處理一些權重特殊的完全圖時，因為對於 中的每個頂點，都要考慮所有出邊，而對這些出邊進行 Relax 又會導致許多頂點被加入 中，這時 SPFA 的表現不會比一般的 Bellman-Ford 來得好多少，時間複雜度同樣為 。

不過在處理一般的圖時，平均來說 SPFA 的複雜度為 。

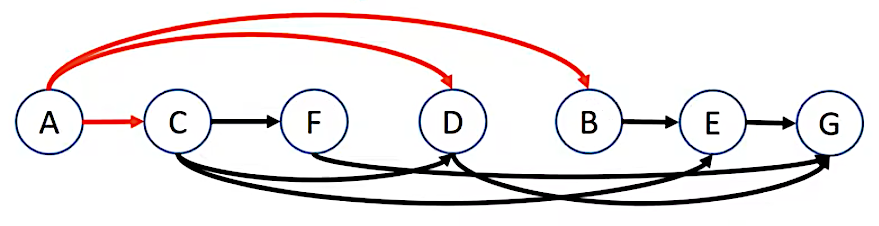
（4）實作 SPFA

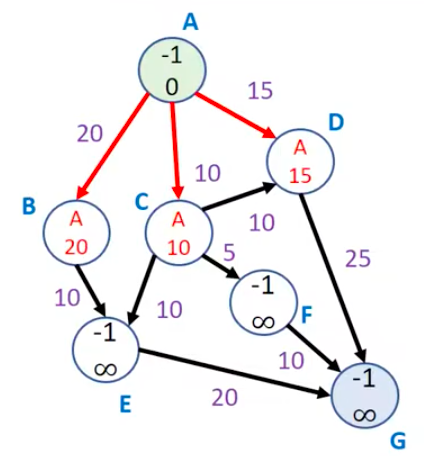
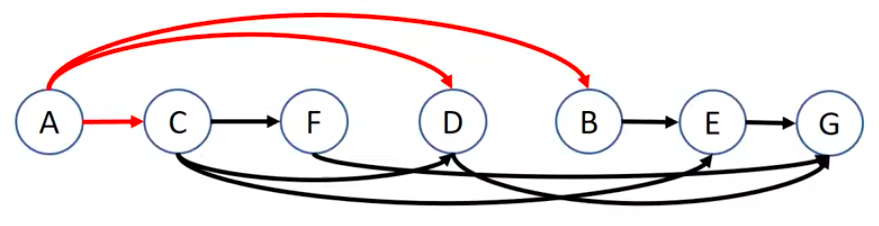
|  |  |
| --- | --- |
| SPFA | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | bool graph::SPFA(int initial){  initialize\_shortest\_path(initial);  queue<int> candidate;  unordered\_map<int, int> update;  for (int i=0 ; i<number\_vertex ; i++){ update[i] = 0; }  // 起點放入 queue 中  candidate.push(initial);  while (!candidate.empty()){  int vertex\_now = candidate.front();  candidate.pop();  list<edge\*>::iterator it=edges[vertex\_now].begin();  // 對所有 vertex\_now 的出邊進行 relax  for ( ; it!=edges[vertex\_now].end() ; it++){  bool relaxed = relax(vertex\_now, (\*it)->to, (\*it)->weight);  // 如果某個出邊指到的點被更新  if (relaxed && (\*it)->to!=candidate.back()){  // 該頂點被更新的次數+1  update[(\*it)->to] += 1;  // 存在負環時回傳 false  if (update[(\*it)->to]==number\_vertex){  return false;  }  // 頂點放到 queue 中  candidate.push((\*it)->to);  } // end of if  } // end of for  } // end of while  // 找到路徑時回傳 true  return true;  } |

1. DAG Algorithm

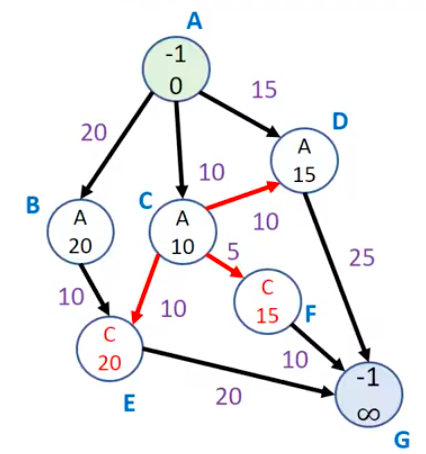
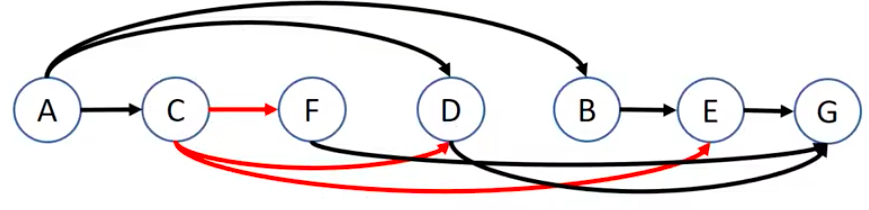
Bellman-Ford 的 relax 過程並沒有按照最短路徑的順序，因此效率不佳，如果能夠事先知道最短路徑中各頂點的順序，就可以減少進行 relax 的次數。

1. DAG 演算法的基本概念



先對頂點進行拓樸排序（見圖論章節），接著，按照得出的順序來依序對各頂點的出邊進行 relax，比如上圖中，按照 A、C、F、D、B、E、G 的頂點順序進行。

首先，對頂點 A 的出邊 、、 進行 Relax，更新 B、C、D 點的距離。



接著，對 C 點的出邊 、、 進行 Relax，同樣更新各點距離。

依此類推，接下來會依序對 、、、 等邊進行 Relax。

拓樸排序只能應用在有向無環圖上，也因此只有「有向無環圖」的最短路徑才能應用 DAG 演算法。

|  |  |
| --- | --- |
| DAG 的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10 | DAG(G,w,s){  // 先進行拓樸排序  Topological Sort(G,s)  initialize(G,s)  // 按照拓樸排序的順序進行 relax  for vertex(u) in Topological Sort:  for vertex(v)Adj[u]  Relax(u,v,w)  return true  } |

1. DAG 的複雜度
2. 拓樸排序的複雜度（DFS）：
3. 按照拓樸排序 Relax 的複雜度：
4. 總計：

（3）實作 DAG 演算法

|  |  |
| --- | --- |
| DAG | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | void graph::DAG(int N){  initial\_shortest\_path(initial);  // 呼叫之前寫的 DFS 會產生一個 stack<int> topological\_sort  // 當中有拓樸排序的頂點順序  DFS(initial);  while(!topological\_sort.empty()){  // 從拓樸排序中拿出一個頂點  int index\_now = topological\_sort.top();  topological\_sort.pop();  list<edge\*>::iterator it = edges[index\_now].begin();  // 對該頂點的所有出邊進行 relax  for (it ; it!=edges[index\_now].end() ; it++){  relax(index\_now, (\*it)->to, (\*it)->weight);  }  }  } |

1. Dijkstra's Algorithm
2. Dijkstra 演算法的基本概念

Dijkstra 演算法假定圖中沒有負邊，則經過邊數越多成本必定增加。

因此，可以把頂點分成「已經找到最短路徑」和「還未找到最短路徑」兩組，並利用 Convergence Property，從已經找到最短路徑的組別，往剩下還沒找到最短路徑的頂點進行 Relax。

因為要取出「未找到」組中 最短的頂點，頂點之間要有順序，通常會以 Priority queue 實作。

而且 Dijkstra 演算法每輪都以當前的最佳選擇（未找到最短路徑組中，距離最短的頂點）為下一步，是一種貪婪演算法。

（2）Dijkstra 演算法的步驟

A. 初始化 Distance（無限大）與 Predecessor（-1）

B. 把頂點分成「已經找到最短路徑」和「還未找到最短路徑」兩組

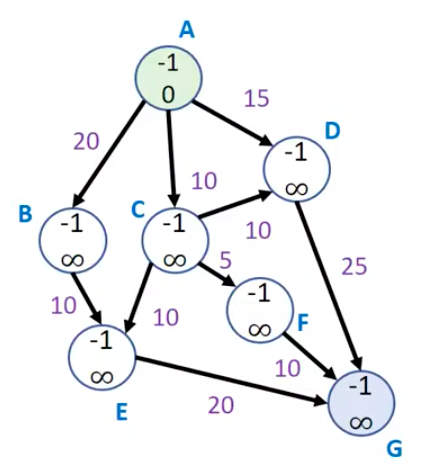
C. 從還沒找到最短路徑的組別中，找到 Distance 最小的頂點 V

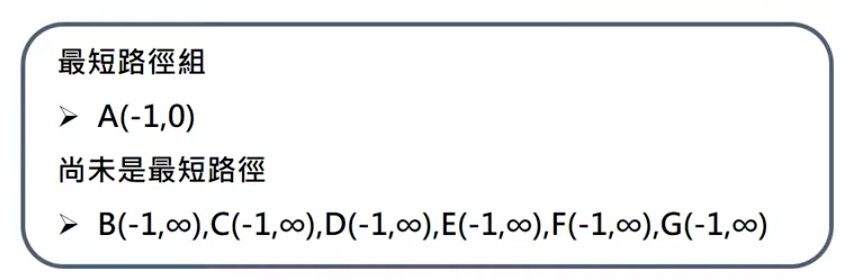
D. 對頂點 V 進行 relax 並放到已經找到最短路徑組

E. 重複前兩步驟共 次

F. 所有頂點都被放到已經找到最短路徑組時完成

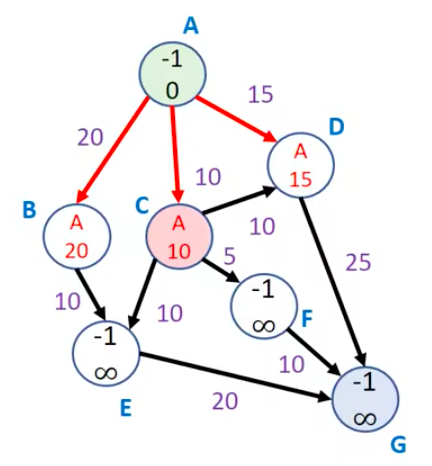
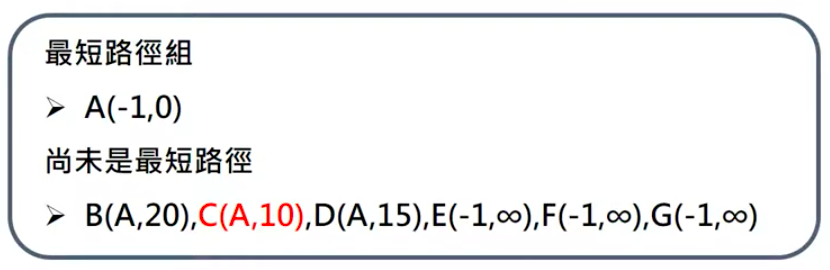
（3）Dijkstra 演算法的進行過程





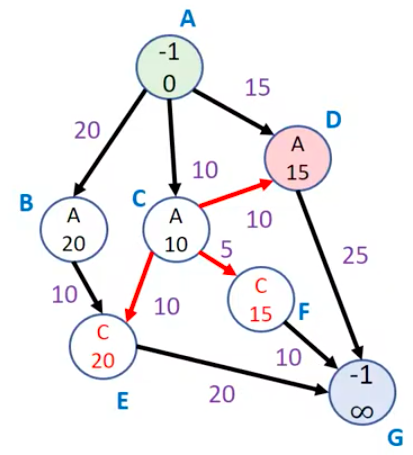
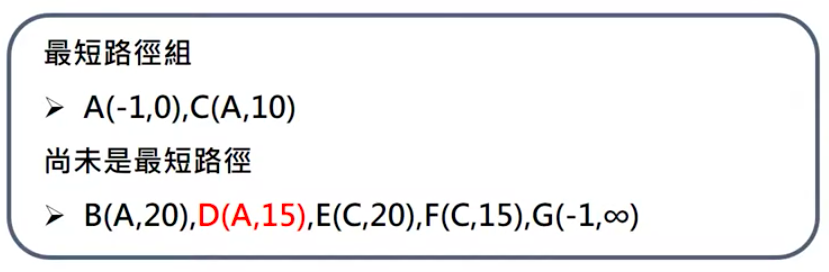
一開始只有起點 A 點在最短路徑組中， 是 -1，距離是 0，其他頂點的距離都是無限大。

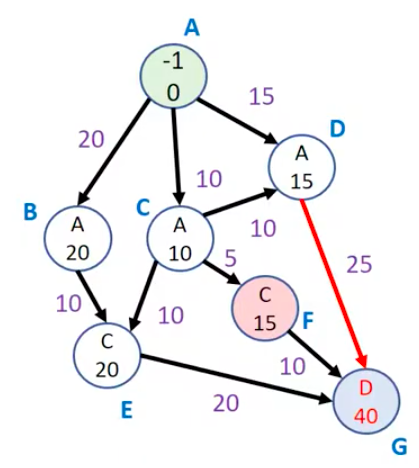
對 A 點的所有出邊作 relax，完成後 B、C、D 點的 都是 A，距離分別為 20、10、15。

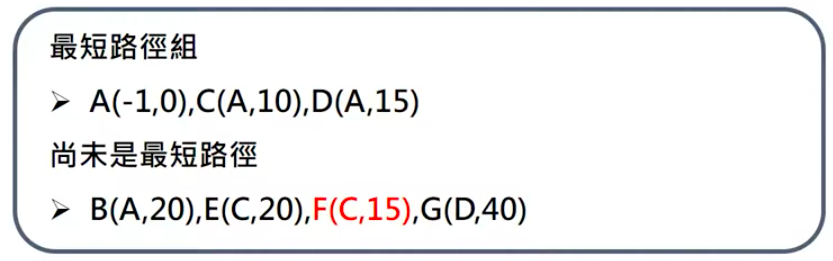


接下來，從「未找到最短路徑組」取出目前距離最短的 C，放到最短路徑組中，代表已經找到由 A 到 C 的最短距離。

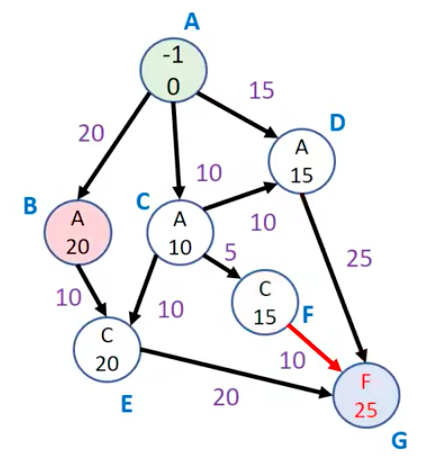
之所以可以確定由 A 到 C 的最短距離就是 的權重 10，是因為 Dijkstra 演算法假設圖上沒有負邊，既然如此，從 A 出發先經過 B 或 D 再走到 C，光是經過的第一個邊 或 權重就已經比 的權重 10 來得大了，也就是說，從 A 出發繞路到 B 或 D 再到 C，一定不會比直接走 到 C 來得好。

對 C 點的出邊作 relax，更新 E 和 F 點的距離為 和 ，且 設定為 C（雖然也處理 ，但 D 點的距離沒有變短，所以 D 點的 仍為 A）。

此時尚未是最小路徑組中距離最短的是 D 和 F，距離同為 15。任意選擇 D 點放到最短路徑組，並對出邊進行 relax，使 G 點的距離被更新為 、 為 D。



接下來，選到 F 加入最短路徑組，更新 G 點的距離為 ，且此時 值改為 F。



隨後依序挑選到 B、E、G 進行處理，直到所有頂點都被加到最短路徑組中，就完成演算法的過程。

（4）Dijkstra 演算法的虛擬碼

因為要一路取出目前「未找到」組中距離最小的頂點，所以通常使用 priority\_queue 來存放所有頂點，並依照距離排序。

|  |  |
| --- | --- |
| Dijkstra 演算法的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Dijkstra(G,w,s){  initialize(G,s)  // 把頂點按照 Distance 排列  priority\_queue = V[G]    while priority\_queue is not empty:  // 取出 Distance 最小的頂點 u  u = priority\_queue.pop()  for vertex(v)Adj[u]  Relax(u,v,w)  return true  } |

（5）Dijkstra 演算法的複雜度

取決於 priority\_queue 的資料結構而有不同，可以拆成兩個部分來看，一個部分是建立 queue 並依序把所有頂點取出，另一部分是對每個邊進行一次 relax。

把頂點放到 queue 中，隨後依序把每個頂點取出

* 陣列：
* 紅黑樹：

對圖上的每個邊進行一次 relax

* 陣列：
* 紅黑樹：

總計

* 陣列：
* 紅黑樹：

（6）Johnson’s Algorithm

前面提到 Dijkstra 演算法假設沒有負邊，Johnson 提出了修改權重的方式來增加 Dijkstra 演算法的適用範圍，使其可以處理有負邊，但不存在負環的情形。

應用公式 ，把圖上每個邊原本的權重 都換成新的權重 後，所有負權重的邊都會變為非負權重的邊，其中 和 是由起點到 u 和 v 的最短距離，可由 Bellman-Ford 演算法得到。

（7）實作 Dijkstra 演算法

|  |  |
| --- | --- |
| Dijkstra’s Algorithm | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | // 每個頂點 vertex 有編號、predecessor 和 distance  typedef struct{  int index;  int predecessor;  float distance;  }vertex;  void graph::Dijkstra(int initial){  // queue 中存放 <vertex\*>  // 其中第二個參數代表指定以 vector 實作  priority\_queue<vertex\*, vector<vector\*>, compare> candidates;  // 記錄已經處理過的頂點  unordered\_set<int> already\_shortest;  vertex\* vertex\_now;  initialize\_shortest\_path(initial);  // 把起點加入 candidates 中  // predecessor 為 -1，距離為 0  candidates.push(new vertex{initial, -1, 0});  // 在 queue 為空之前繼續處理  while(!candidates.empty()){  int index\_now = candidates.top()->index;  // 取到的頂點在「已找到」組時忽略  if (already\_shortest.find(index\_now) != already\_shortest.end()){  candidates.pop();  continue;  }  // 把這個頂點加到「已找到」組  already\_shortest.insert(index\_now);    // 對所有出邊進行 relax  list<edge\*>::iterator it=edges[index\_now].begin()  for (it ; it!=edges[index\_now].end() ; it++){    bool relaxed = relax(index\_now,(\*it)->to,(\*it)->weight);  // 出邊指到的頂點距離改變時，加到 candidates 中  if (relaxed){  candidates.push(new vertex{(\*it)->to,  predecessor[(\*it)->to], distance[(\*it)->to]});  }  }  candidates.pop();  }  } |

1. Floyd-Warshall 演算法

Floyd-Warshall 是 All-Pairs Shortest Path 的演算法，會一次算出所有頂點到其餘所有頂點的最短路徑，屬於一種動態規劃的應用。

（1）Floyd\_Warshall 的核心概念

Floyd\_Warshall 是依序加入中繼點到某個頂點對間，看距離會不會縮短。如果經過中繼點的距離比原距離短，則把該頂點對間的距離更新（即 relax）。

Floyd\_Warshall 演算法屬於動態規劃，藉由最小化頂點與頂點對間的距離，得到最小距離矩陣，觀察得到的矩陣，會發現最短路徑必由經過中間所有頂點對的每段最短路徑共同組成。

原本頂點 和 之間的距離是 ，如果發現加上 後，由 先走到 ，再由 走到 的距離 比 更短，那就把新的距離 設為 。



上圖中，原本還未加入（還沒考慮）E 點時，B 到 C 的距離為經過 的 50，但加入 E 點後，發現可以走 ，兩段權重和只有 ，因此把 B 到 C 的距離更新為 30，即 。

（2）Floyd\_Warshall 的步驟

A. 初始化一個 Adjacent matrix

B. 記錄每個頂點的 predecessor

C. 加入一個點為中繼點，進行 次更新

D. 重複上個步驟 次，直到把每個頂點都加入作為中繼點

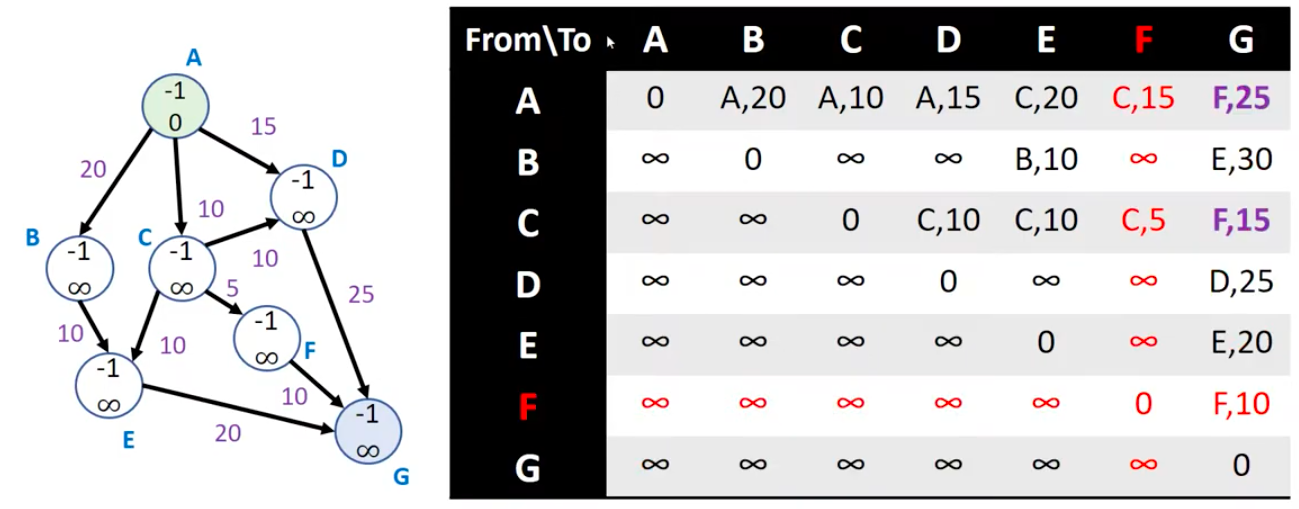
E. 得到最短距離矩陣

|  |  |
| --- | --- |
| Floyd\_Warshall 演算法的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | Floyd\_Warshall(G,w){  Initialize D  // 加入一個點 k 作為中繼點  for k = 1~|V|:  // 更新 |V|^2 大小的矩陣  for i = 1~|V|:  for j = 1~|V|:  = min(, +)  // 檢查是否存在負環  for i=1~|V|:  if D[i,i]<0:  return False  return D  } |

複雜度為跑完三層迴圈所需的 。

要檢查是否存在負環，只要檢查矩陣中對角線的值是否為負即可，因為沿著負環走一圈，會回到原來的頂點，且得到的距離小於 0。

進行過程中產生的矩陣如：

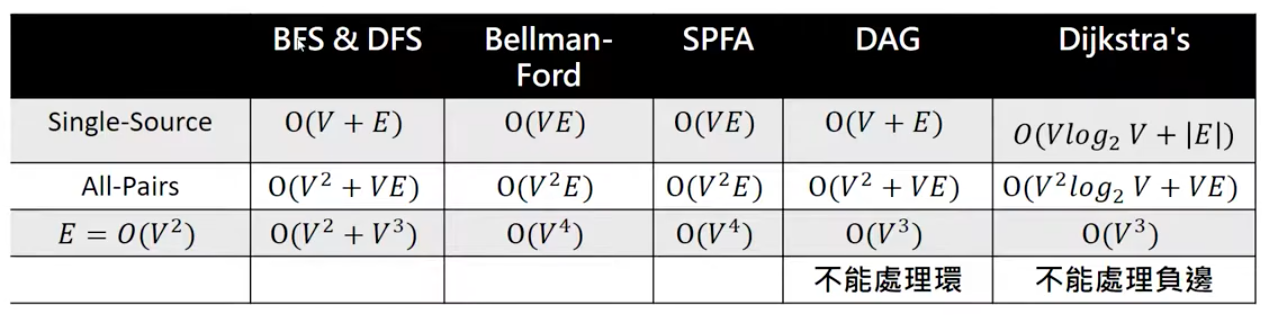


（3）實作 Floyd-Warshall 演算法

|  |  |
| --- | --- |
| Floyd-Warshall’s Algorithm | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  5051  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  7172  73  74  75  76 | #include <iostream>  #include <limits>  #include <stdlib.h>  using namespace std;  class graph{  private:  int vertex;  int \*\*distance\_matrix;  int \*\*predecessor\_matrix;  public:  graph(int);  // void print\_distance\_matrix();  // void print\_predecessor\_matrix();  void floyd\_warshall();  bool add\_edge(int, int, int=1);  }  graph::graph(int v){  vertex = v;  // 初始化頂點對頂點距離的矩陣 distance\_matrix  distance\_matrix = (int\*\*) malloc(sizeof(int\*)\*vertex);  for (int i=0 ; i<vertex ; i++){  \*(distance\_matrix+i) = (int\*) malloc(sizeof(int)\*vertex);  }  // 頂點到自身的距離初始化為 0，到其餘頂點距離為無限大  for (int i=0 ; i<vertex ; i++){  for (int j=0 ; j<vertex ; j++){  \*(\*(distance\_matrix+i)+j) = (i==j) ? 0 :  numeric\_limits<int>::max();  }  }  // 初始化頂點對間 predecessor 的矩陣  predecessor\_matrix = (int\*\*) malloc(sizeof(int\*)\*vertex);  for (int i=0 ; i<vertex ; i++){  \*(predecessor\_matrix+i) = (int\*) malloc(sizeof(int)\*vertex);  }  // 頂點到自身的 predecessor初始化為 0，其餘頂點對間為 -1  for (int i=0 ; i<vertex ; i++){  for (int j=0 ; j<vertex ; j++){  \*(\*(predecessor\_matrix+i)+j) = (i==j) ? 0 : -1;  }  }  }  void graph::floyd\_warshall(){  for (int k=0 ; k<vertex ; k++){  for (int i=0 ; i<vertex ; i++){  for (int j=0 ; j<vertex ; j++){  // i 無法走到 k 時不處理  if (distance\_matrix[i][k] == numeric\_limits<int>::max() ){ continue; }  // k 無法走到 j 時不處理  else if (distance\_matrix[k][j]==numeric\_limits<int>::max()){ continue; }  // i->k->j 比 i->j 來得短時，更新距離  else if (distance\_matrix[i][j] > distance\_matrix[i][k]+distance\_matrix[k][j]){  distance\_matrix[i][j] = distance\_matrix[i][k] + distance\_matrix[k][j];  predecessor\_matrix[i][j] = k;  }    } // end of for(j)  } // end of for(i)  } // end of for(k)  } // end of floyd\_warshall |

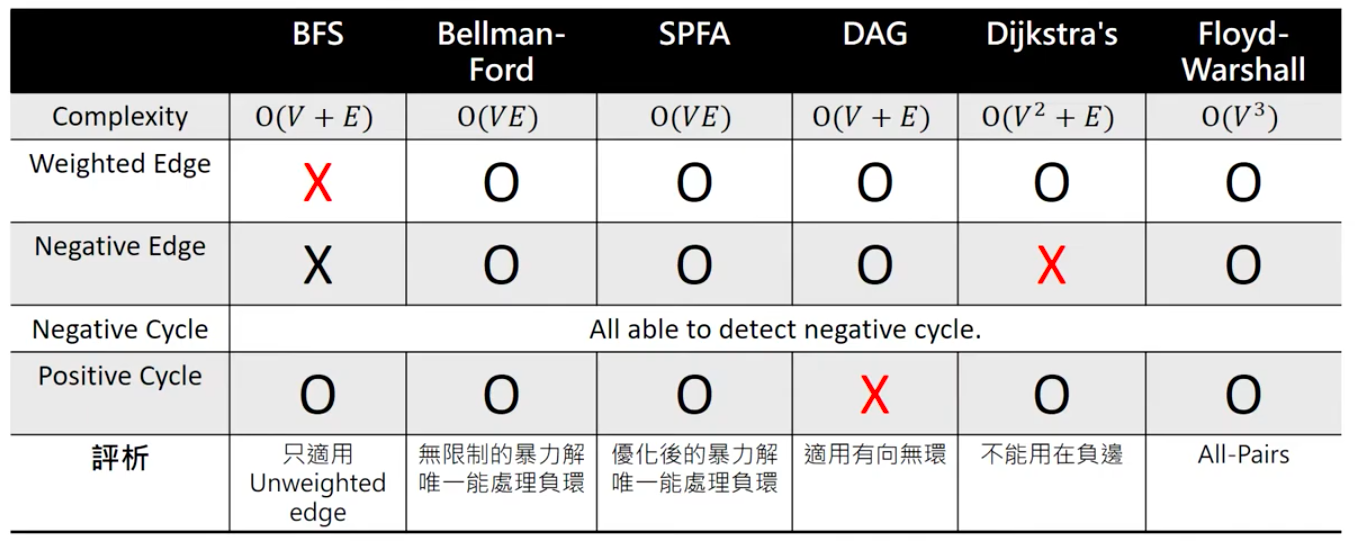
1. 最短路徑問題總結

（1）複雜度比較



注意到上表中，Single-Source 是求解由一個特定頂點到所有其他頂點的最短距離，對全部 個頂點作 Single-Source，就可以得到 All-Pairs，因此 All-Pairs 的複雜度都是 Single-Source 乘上 倍。

（2）各情況下的演算法適用



1. 各邊無權重或權重相同：BFS
2. 權重不同，但皆為正值：Dijkstra
3. 權重有負，但無環：DAG
4. 權重有負，有環：SPFA
5. 需要印出負環的順序：Bellman-Ford
6. All-pairs 問題：Floyd-Warshall