資料結構 C 演算法

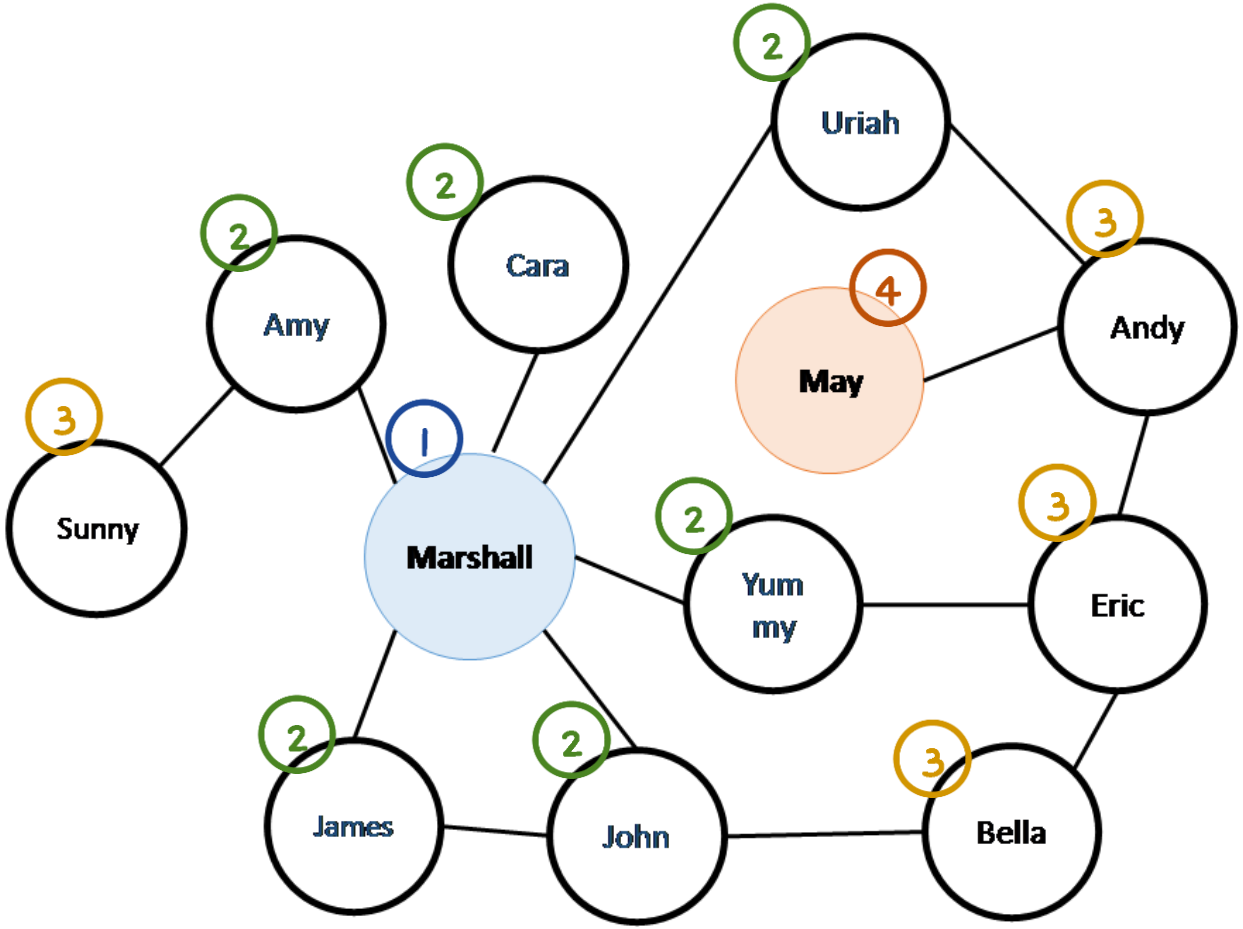
最短路徑 Shortest Path

最短路徑 Shortest Path

透過幾個人介紹可以認識？

- 如果 Marshall 想要認識 May，最少要透過幾個人介紹才可以認識？

- 將 Marshall 視為起始點，執行廣度優先搜尋 BFS，就可以知道這兩點的**最短路徑**！

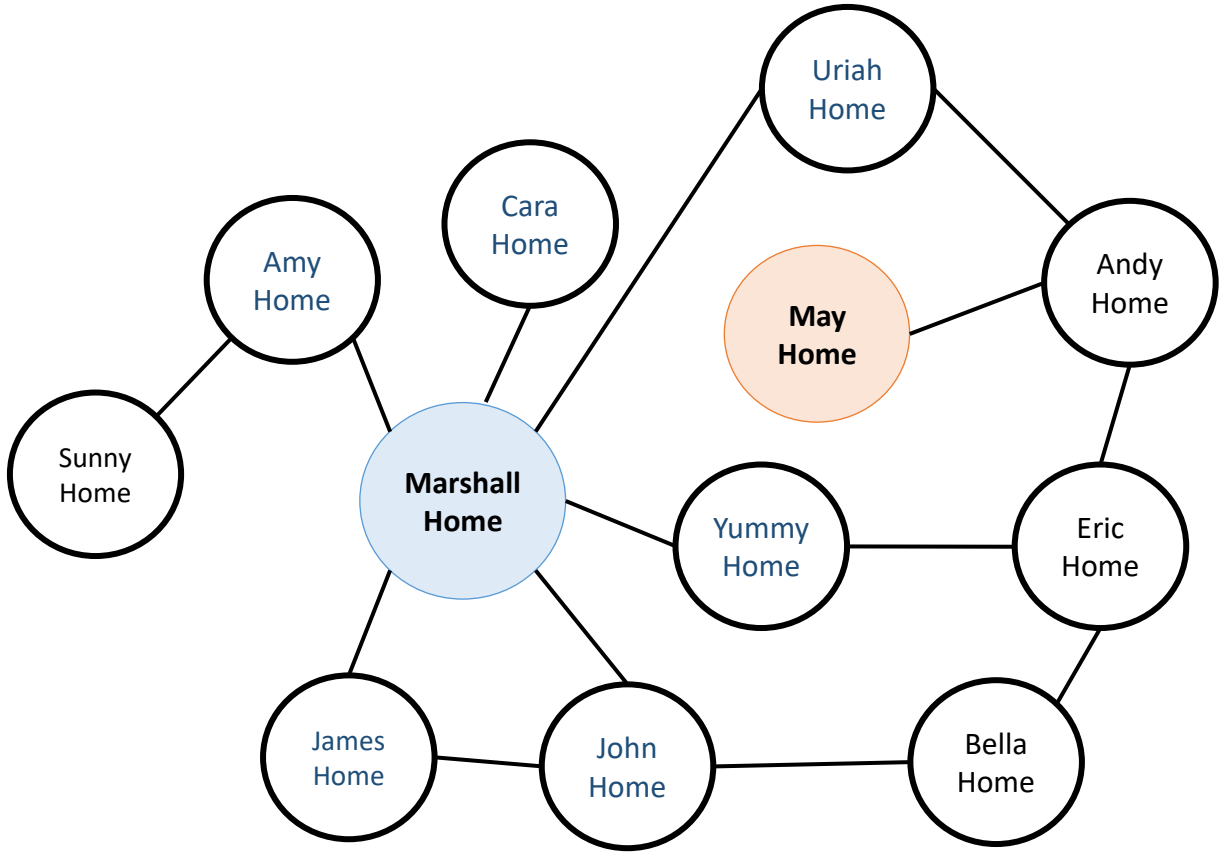


最短路徑： Marshall -> Uriah -> Andy -> May

如果將人際關係圖改成交通路線圖呢？

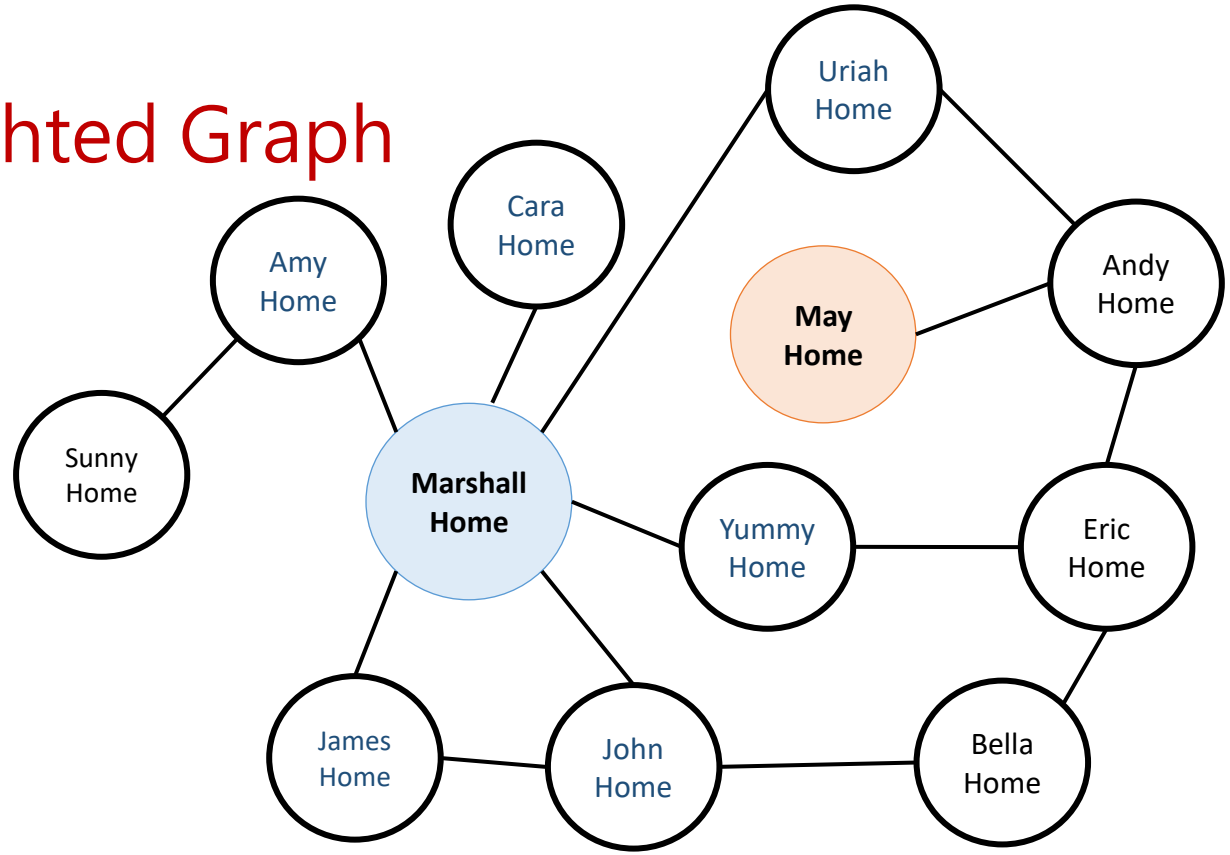
- Marshall Home -> Uriah Home-> Andy Home -> May Home 仍然會最快嗎？

- 當圖代表的是交通路線時，我們會想知道的是任兩個點之間如何最快到達，至於中間是否能經過最少點就不是那麼重要了！



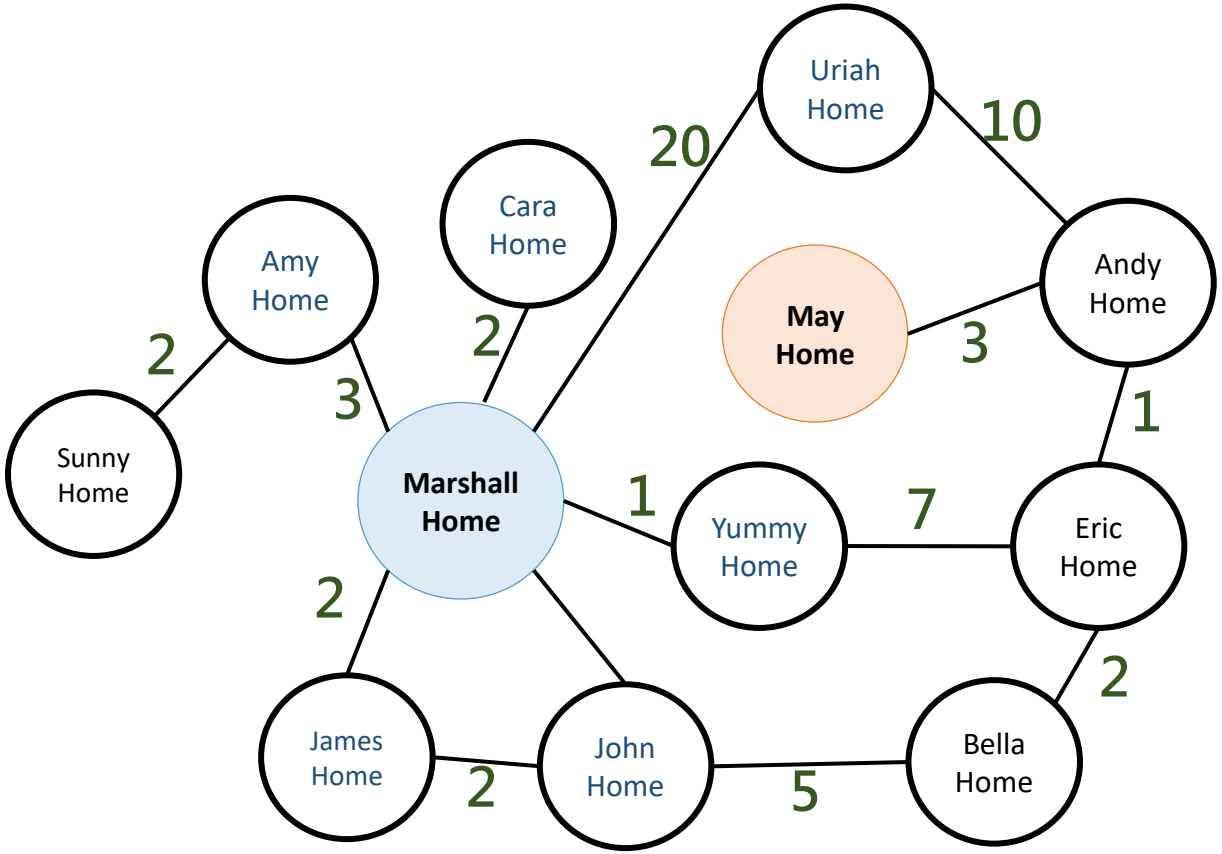
加權圖形 Weighted Graph

- 使用**權重** weight 代表相鄰兩點之間的距離感。
- 有權重的圖稱為加權圖形 Weighted Graph
- 依照圖的種類，權重可以代表：
  - 兩點之間相距幾公里。
  - 兩點之間相距幾分鐘。
  - 兩點之間相差多少錢。
  - ……



權重 weight

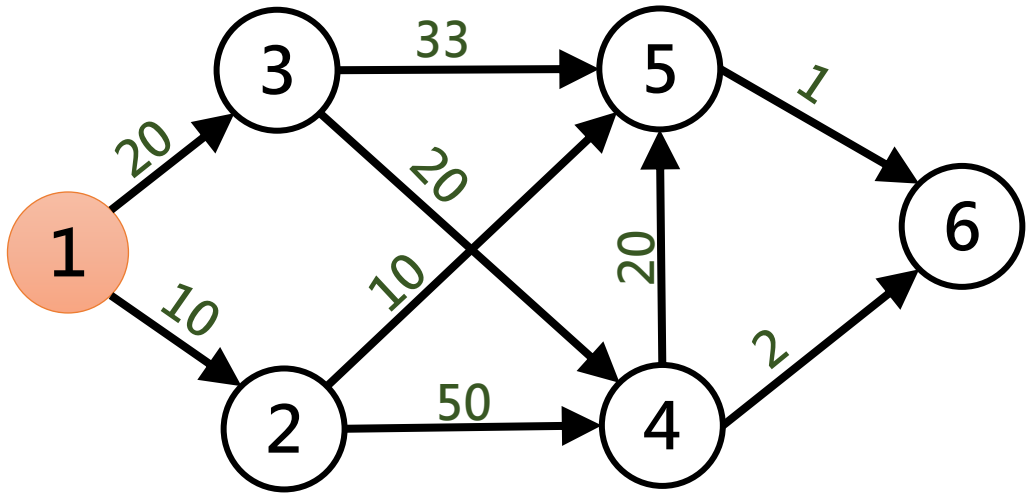
- 當邊上沒有任何權重值時，預設每邊的權重值就是 1 。
- 當每邊權重值是 1 時，經過最少點的路徑就會是最短路徑。
- 如果邊上有權重值，那最短路徑又該如何取得呢？




Dijkstra

Dijkstra 演算法

- Dijkstra 演算法是用來處理單源最短路徑問題：計算圖上某一點到其他所有點的最短路徑。



Dijkstra 演算法 – 概念

- 任兩點之間的最短路徑有兩種情況：
  - 此兩點之間有邊相連，是相鄰的兩點
  - 此兩點會經過其它點，產生最短路徑

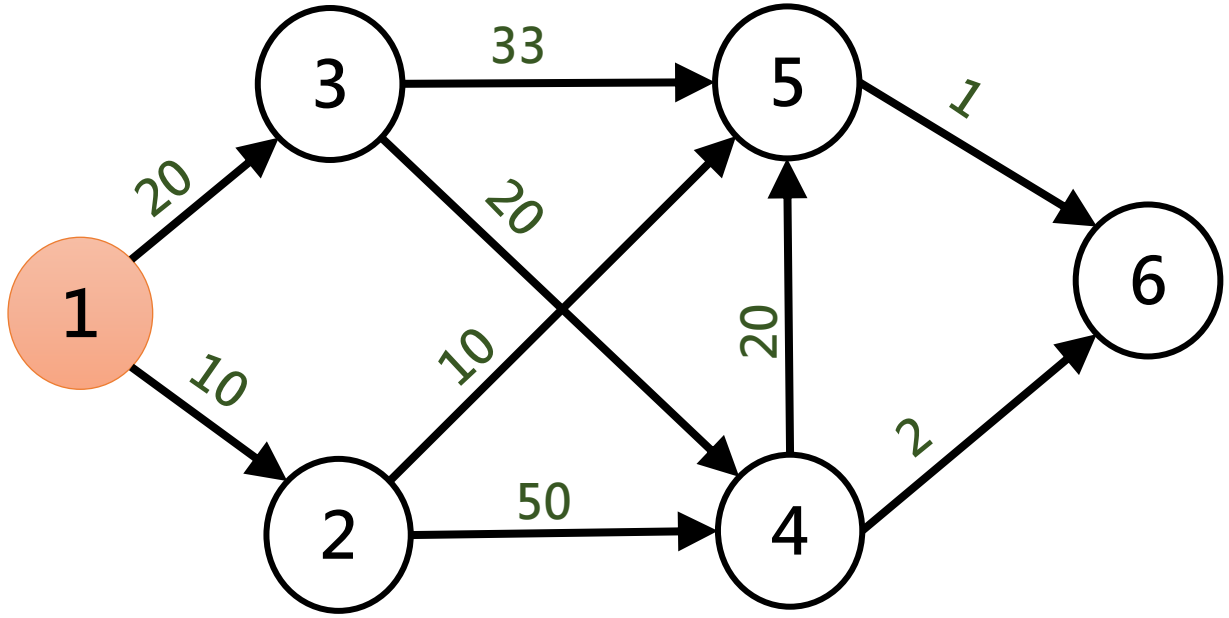
- 若是相鄰兩點產生的最短路徑，只要列出所有的邊就可得到答案
- 若是兩點會經過其它點，產生最短路徑，就需要先找出指定點與另一點的最短路徑，畢竟只有最短路徑才能讓另一個路徑也是最短的！

Dijkstra 演算法 – 流程

- 步驟 1：列出初始距離
- 步驟 2：設定兩個節點集合：
  - 不確定節點集合: 除了指定節點外的所有節點
  - 確定節點集合: 空集合
- 步驟 3：從不確定節點集合裡面找出有最短路徑的節點移到確定節點集合
- 步驟 4：反覆執行步驟 3，直到所有節點都放到確定節點集合。

Dijkstra 演算法 – 範例

- 請找出節點 1 到其它各節點的最短路徑



Dijkstra 演算法 – 步驟 1

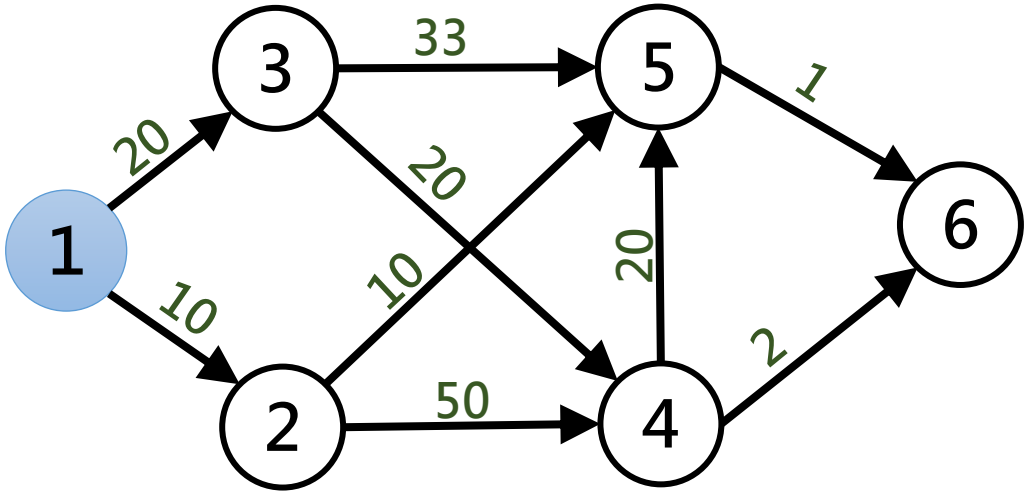
- 步驟 1：先列出指定點(節點1)到其它點的初始距離

節點 1 只有跟節點2與節點3相鄰，
因此只有這兩點有初始路程，其
它都先將路程設為無限大

	1	2	3	4	5	6
距離	0	10	20	∞	∞	∞

自己到自己
的距離設為 0

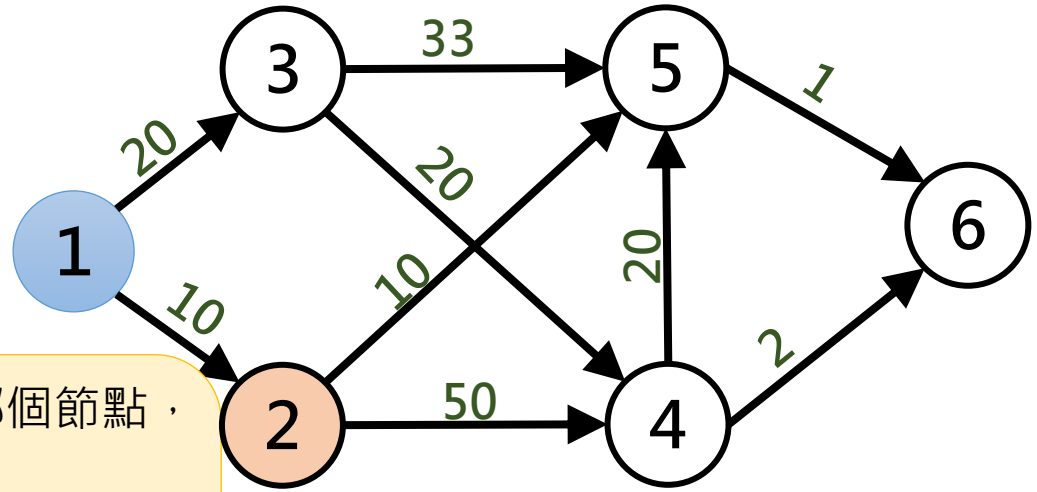
與指定點沒有相鄰的點，
都先將距離設為無限大。



Dijkstra 演算法 – 步驟 2 <第1個確定點>

- 步驟 2：在初始距離中找出距離是最短的點做為**第 1 個確定點**，之後此點的最短路徑就不會再被更動。
  - 如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從初使距離中決定一個點的最短路徑。

	1	2	3	4	5	6
距離	0	10	20	∞	∞	∞



• 節點 1 只有跟節點 2 與節點 3 相鄰，所以之後不論節點 1 是要到哪個節點，一定是先經過這兩點的其中一點。

• 節點 1 到節點 2 的距離是 10，節點 1 到節點 3 的距離是 20，10 < 20，此時就可以得到節點 1 到節點 2 的最短路徑必定是 10。

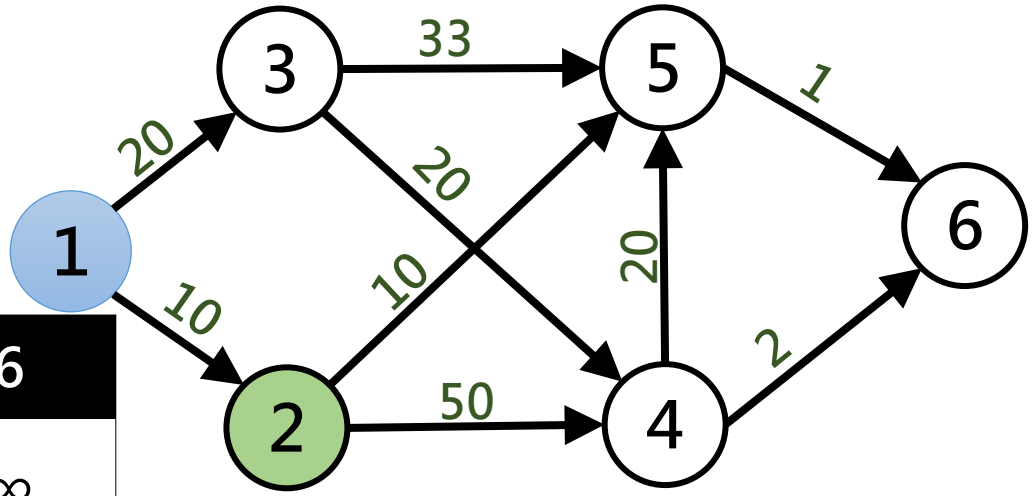
•若是先通過節點 3，那距離至少會大於20，所以是不可能的！

Dijkstra 演算法 – 步驟 3<第2個確定點>

- 步驟 3：在剩下尚未決定最短距離的節點中，判斷若是經過已經確定最短路徑的**節點 2**，是否可降低各點的最短路徑，若有降低，就更改該點的最短距離值，若沒有，就維持原距離值。並在這些點中找出距離最短的頂點最為**第 2 個確定點**。
  - 同樣的，第 2 個確定點產生的原因也是因為如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從已知的距離中找出一個最短距離。

	1	2	3	4	5	6
原距離	0	10	20	∞	∞	∞

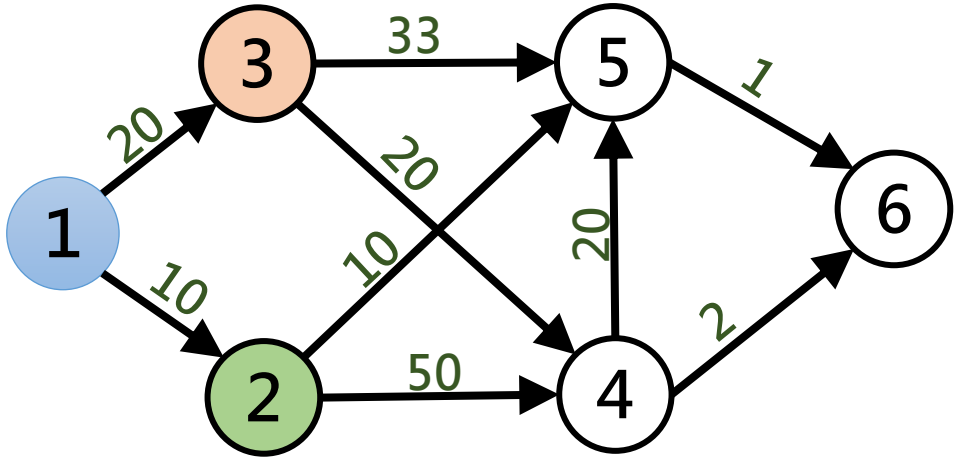
	1	2	3	4	5	6
經過節點 2 的新距離	0	10	20	60	20	∞



Dijkstra 演算法 – 步驟 3 <第2個確定點>

- 從節點 1 經過節點 2 可以讓節點 4 與節點 5 的距離降低，所以就先這兩個節點的距離值更新。
- 節點 3、節點 4、節點 5、節點 6中距離最短的是節點 3 與節點5，選擇節點 3 最為**第 2 個確定點**。
  - 當有兩個節點的距離同時都是最低的值時，可任意選一點。

	1	2	3	4	5	6
距離	0	10	20	60	20	∞

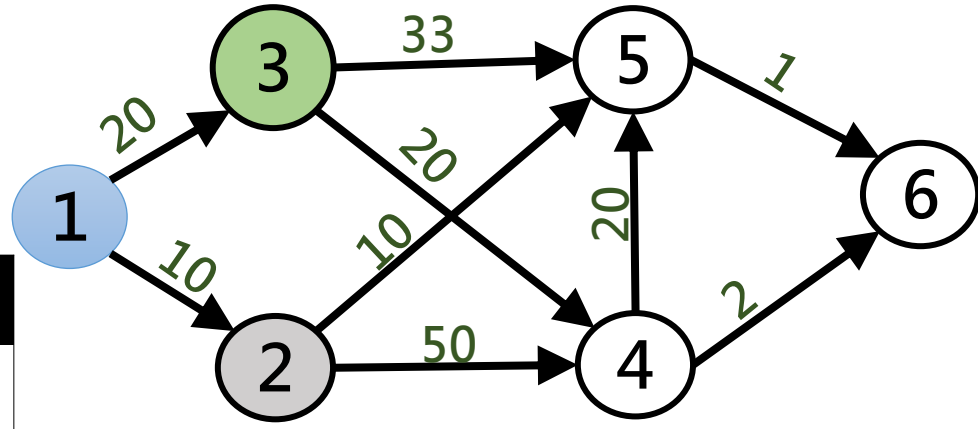


Dijkstra 演算法 – 步驟 4<第3個確定點>

- 步驟 4：在剩下尚未決定最短距離的節點中，判斷若是經過前一個步驟產生的確定點(**節點3**) ，是否可降低各點的最短路徑，若有降低，就更改該點的最短距離值，若沒有，就維持原距離值。並在這些點中找出距離最短的頂點最為**第 3 個確定點**。
  - 同樣的，第 3 個確定點產生的原因也是因為如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從已知的距離中找出一個最短距離。

	1	2	3	4	5	6
距離	0	10	20	60	20	∞

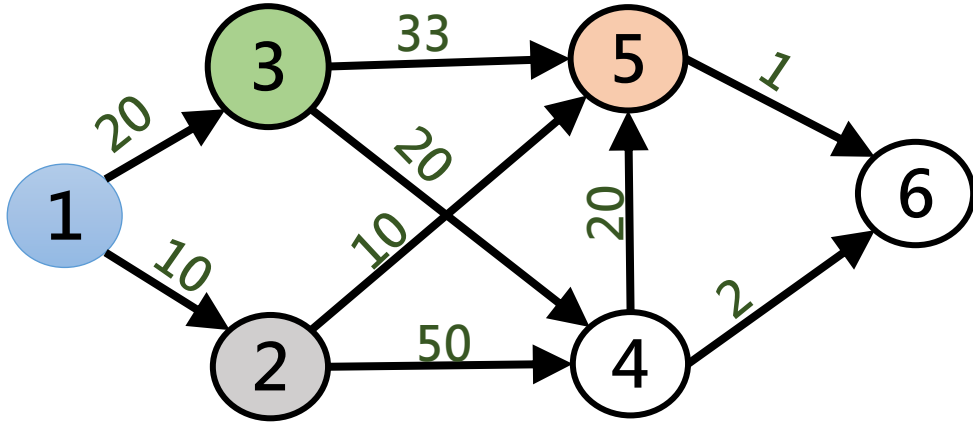
	1	2	3	4	5	6
經過節點 3 的新距離	0	10	20	40	20	∞



Dijkstra 演算法 – 步驟 4 <第3個確定點>

- 從節點 1 經過節點 3 可以讓節點 4 的距離降低，所以要將節點4的距離值更新。
- 節點 4、節點 5、節點 6中距離最短的是節點5，選擇節點 5 最為**第 3 個確定點**。

	1	2	3	4	5	6
距離	0	10	20	40	20	∞

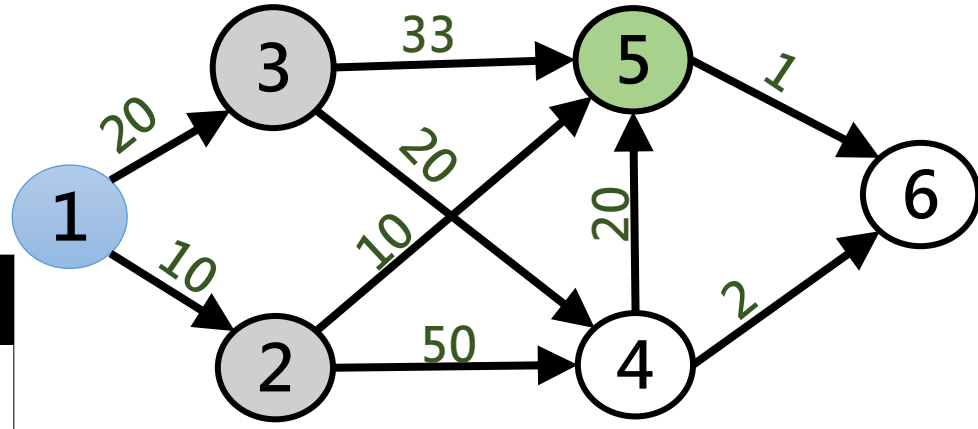


Dijkstra 演算法 – 步驟 5 <第4個確定點>

- 步驟 5：在剩下尚未決定最短距離的節點中，判斷若是經過前一個步驟產生的確定點(**節點5**)，是否可降低各點的最短路徑，若有降低，就更改該點的最短距離值，若沒有，就維持原距離值。並在這些點中找出距離最短的頂點最為**第 4 個確定點**。
  - 同樣的，第 4 個確定點產生的原因也是因為如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從已知的距離中找出一個最短距離。

	1	2	3	4	5	6
距離	0	10	20	40	20	∞

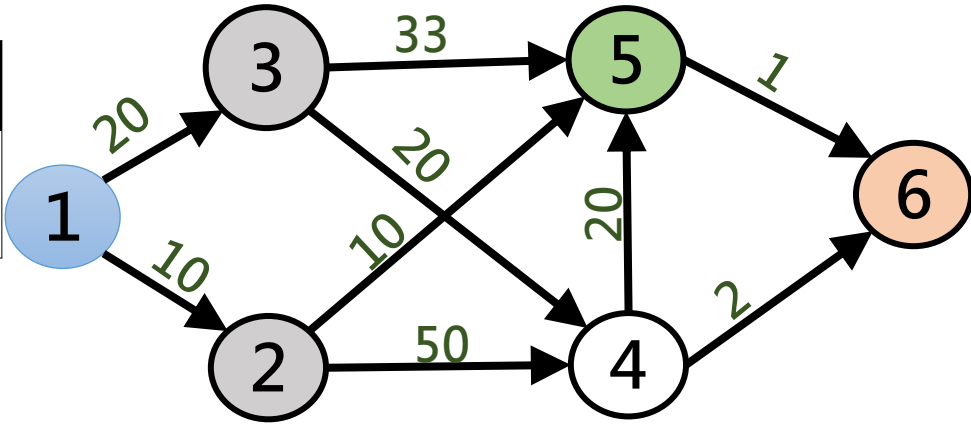
	1	2	3	4	5	6
經過節點 5 的新距離	0	10	20	40	20	21



Dijkstra 演算法 – 步驟 5<第4個確定點>

- 從節點 1 經過節點 5 可以讓節點 6 的距離降低，所以要將節點6的距離值更新。
- 節點 4、節點 6中距離最短的是節點6，選擇節點 6 最為**第 4 個確定點**。

	1	2	3	4	5	6
距離	0	10	20	40	20	21

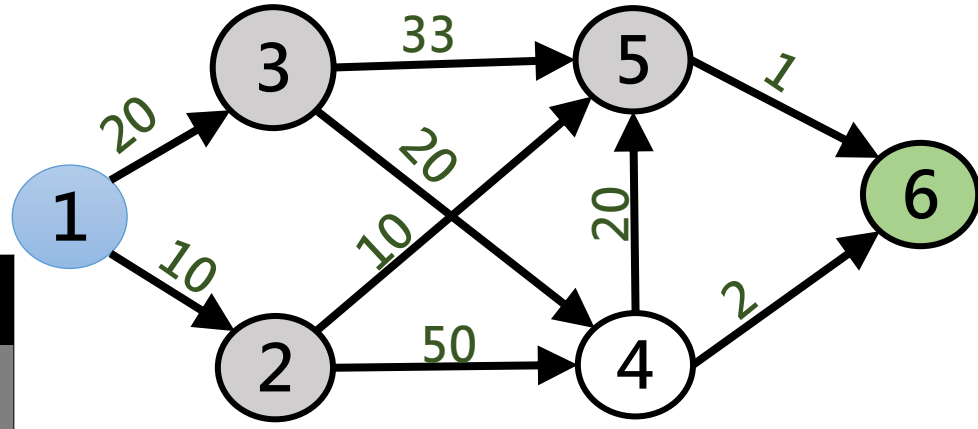


Dijkstra 演算法 – 步驟 6 <第5個確定點>

- 步驟 6：最後只剩下節點 4 尚未確定最短路徑，因此只要計算節點4如果經過**節點6**是否可以降低距離，有降低就更新記錄值。不論是否有降低，節點 4 一定會是**第 5 個確定點**。

	1	2	3	4	5	6
距離	0	10	20	40	20	21

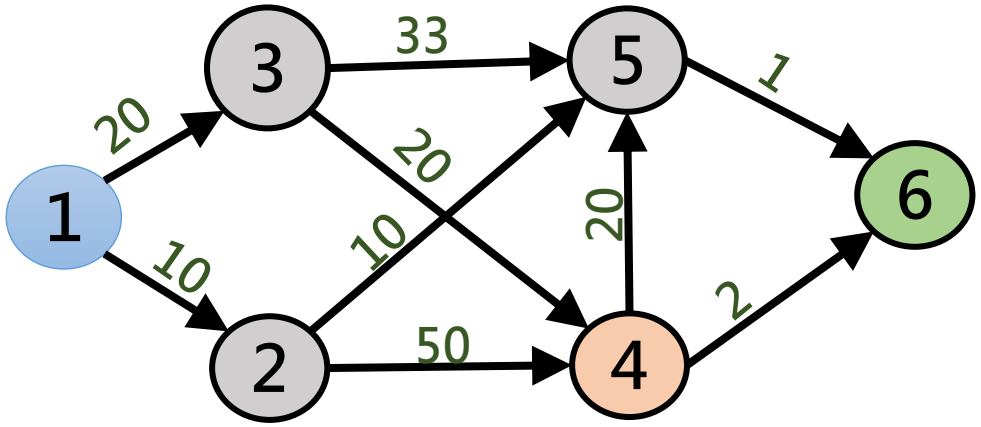
	1	2	3	4	5	6
經過節點 6 的新距離	0	10	20	40	20	21



Dijkstra 演算法 – 步驟 6 <第5個確定點>

- 節點 4 成為**第 5 個確定點**。

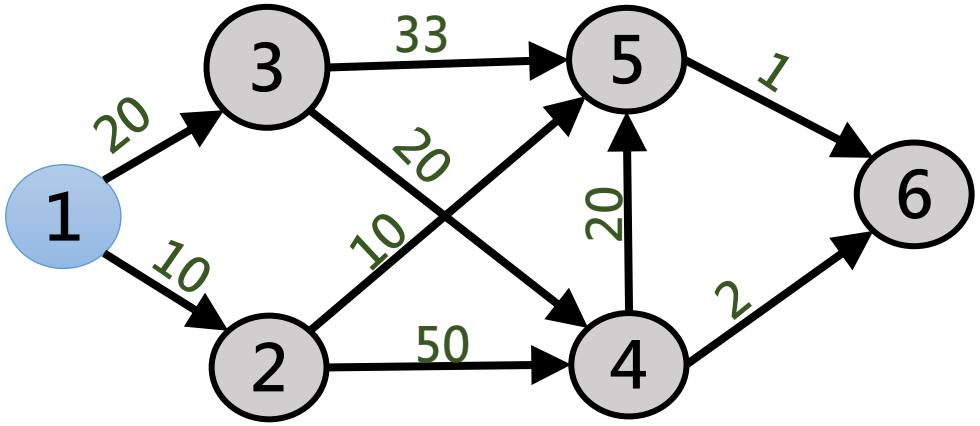
	1	2	3	4	5	6
距離	0	10	20	40	20	21



Dijkstra 演算法 – 步驟 7 <完成>

- 步驟 7：已經完成節點 1 到各節點的最短路徑，最後只要根據需求回傳所需要的路徑長度或是路徑資料就完成了！

	1	2	3	4	5	6
距離	0	10	20	40	20	21



Dijkstra

Dijkstra 實作

若是使用相鄰矩陣記錄圖型結構，那矩陣內指定節點對應的那行就會是初始距離。

若仍有不確定節點，就會持續執行

找出不確認集合裡面最短的距離

找出下一個要從不確定節點集合移到確定節點集合的結點。也就是找出可以經過剛剛確定的最短距離點而縮點距離的點

done 陣列代表的是確定節點集合
doneVerticesCount則是記錄確定節點數量

```
void Dijkstra(int vertex, int verticesCount, int graph[][MAX_VERTICES])
{
 int distance[MAX_VERTICES], done[MAX_VERTICES];
 int i, lastVertex, doneVerticesCount, shortestLength;
 for (i = 0; i < verticesCount; i++) {
  distance[i] = graph[vertex][i];
  done[i] = 0;
 }
 done[vertex] = 1;
 doneVerticesCount = 1;
 while(doneVerticesCount < verticesCount) {
  shortestLength = 99999;
  for (i = 0; i < verticesCount; i++) {
   if (done[i] == 1) continue;
   if (distance[i] < shortestLength) {
    shortestLength = distance[i];
    lastVertex = i;
   }
  }
  done[lastVertex] = 1;
  doneVerticesCount++;
  for (i = 0; i < verticesCount; i++) {
   if (done[i] == 1) continue;
   if (distance[i] > distance[lastVertex] + graph[lastVertex][i])
    distance[i] = distance[lastVertex] + graph[lastVertex][i];
  }
 }
 displayDistance(verticesCount, distance);
}
```

Dijkstra 演算法

- Dijkstra 演算法的主軸思想就是只有前面是最短路徑，後面才會是最短路徑，因此處理的圖絕對**不能有負權重邊**！

邊的權重通常是代表距離、時間等資訊，因此權重值大多為正值，但圖型結構中是沒有強制要求權重值不能為負值，因此處理圖型結構時，一定要注意權重值的正負值情況。

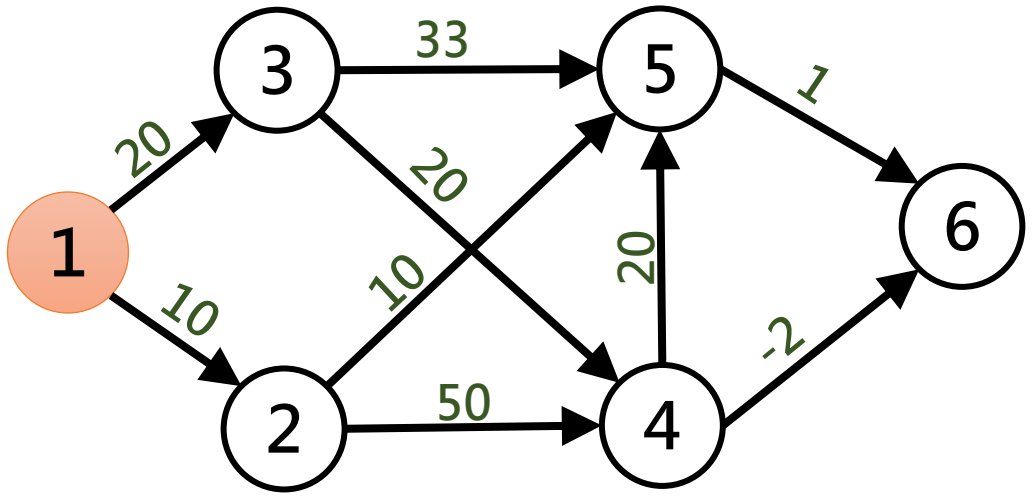
問題來了...

如果邊有負權重，
要怎麼找最短路徑呢？


Bellman-Ford

Bellman-Ford 演算法

- Bellman-Ford 演算法是用來處理單源最短路徑問題：計算圖上某一點到其他所有點的最短路徑。



Bellman-Ford 演算法 – 概念

- 若在任兩節點之間的可能路徑中加入一個邊，可以讓此兩點的路徑距離降低，那就將這個邊加到此兩點的距離中，反之，就忽略這個邊。

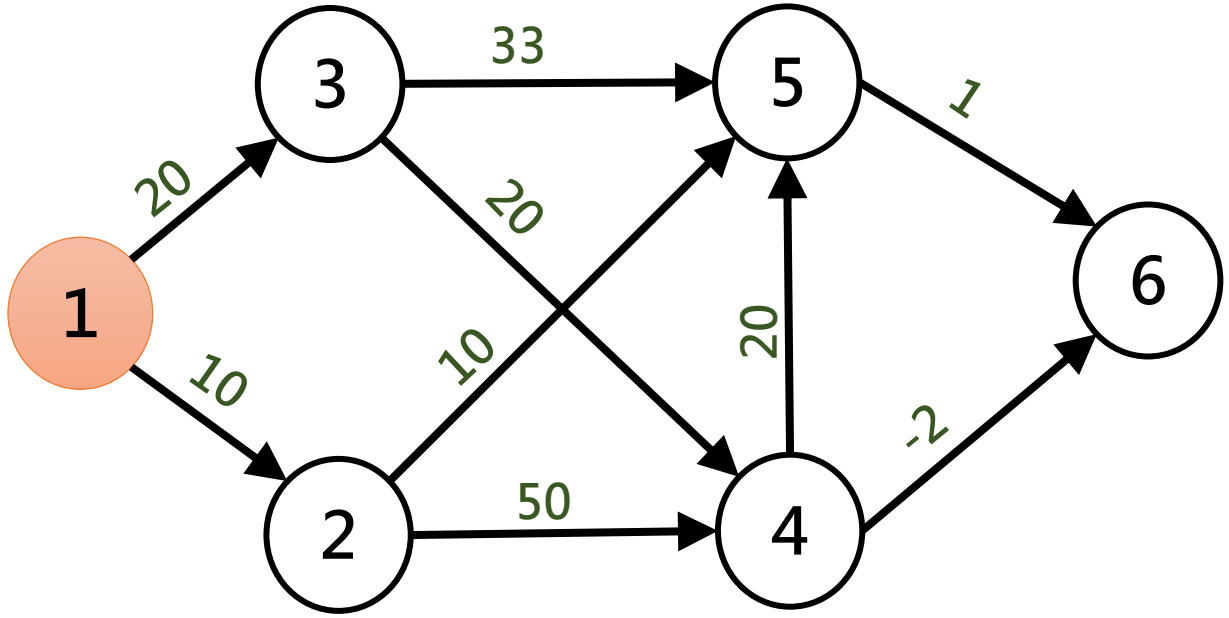
- 只要每個頂點都各別考慮過加入各個邊後是否會造成影響，就可以產生最短路徑。
  - 若有 N 個頂點，M 條邊，最多執行 N * M 的路徑距離大小判斷就可以得到結果。
  - 但是，若 M 條邊的任一條邊加入後都沒有路徑會更改距離，那就可以提早結束判斷。

Bellman-Ford 演算法 – 流程

- 步驟 1：將指定節點到其他任一個節點的距離長度設為無限大。
- 步驟 2：依序取出每一條邊，找出因為這一條邊而可以降低最短距離的兩節點。
- 步驟 3：為了避免步驟 2 中有任兩節點因為加入新邊而降低距離，會間接對其它點的最短路徑造成影響，因此需要重複執行步驟 2，直到所有邊都有對每一個節點檢查。
- 步驟 4：除了完成所有邊對每一個節點檢查以外，如果加入任何一條邊都不會更改任一點的最短路徑時，就可以結束。

Bellman-Ford 演算法 – 範例

- 請找出節點 1 到其它各節點的最短路徑

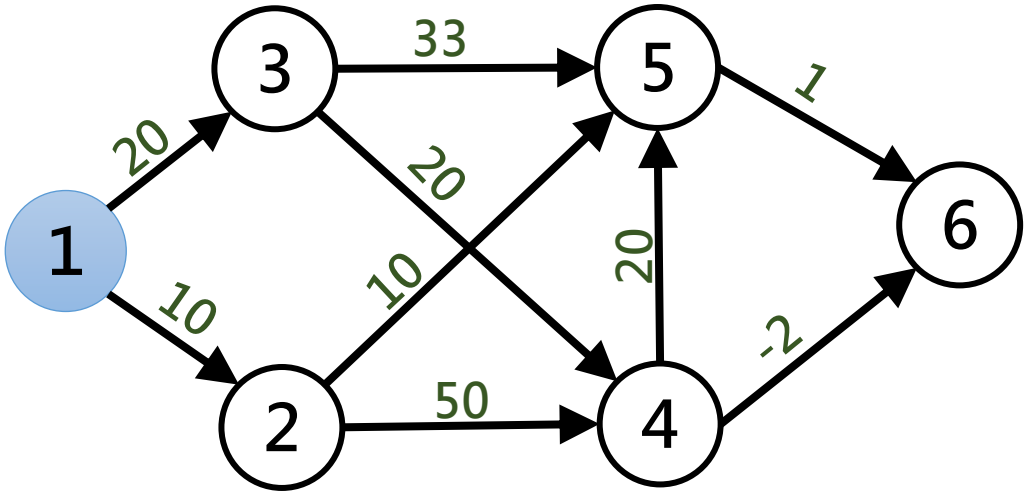


Bellman-Ford 演算法 – 步驟 1

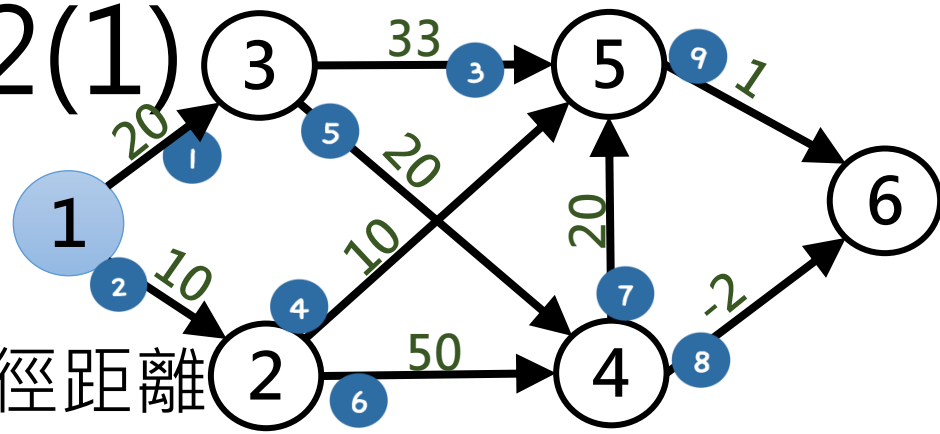
- 步驟 1：先將指定點(節點1)到其它點的初始距離都設為無限大

	1	2	3	4	5	6
距離	0	∞	∞	∞	∞	∞

自己到自己
的距離設為 0



Bellman-Ford 演算法 – 步驟 2(1)



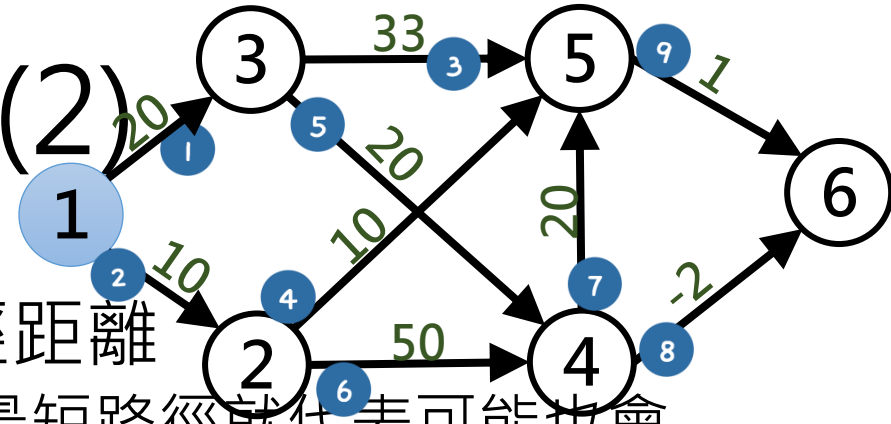
- 加入第 1 條邊：降低節點 1 到節點 3 的路徑距離

	1	2	3	4	5	6
距離	0	∞	20	∞	∞	∞

- 加入第 2 條邊：降低節點 1 到節點 2 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	∞	∞	∞

Bellman-Ford 演算法 – 步驟 2(2)



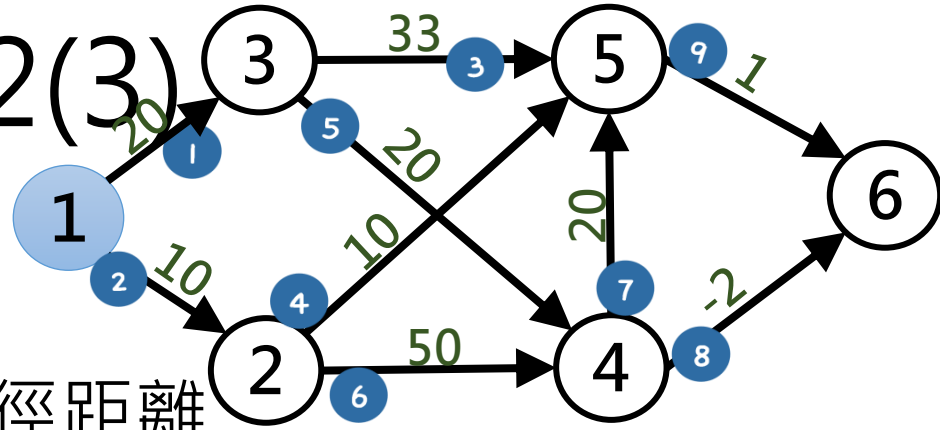
- 加入第 3 條邊：降低節點 3 到節點 5 的路徑距離
  - 節點 1 可到節點3，因此更新節點 3 到節點5的最短路徑就代表可能也會更新節點 1 到節點5的最短路徑

	1	2	3	4	5	6
距離	0	10	20	∞	53	∞

- 加入第 4 條邊：降低節點 2 到節點 5 的路徑距離
  - 節點 1 可到節點2，因此更新節點 2 到節點5的最短路徑就代表可能也會更新節點 1 到節點5的最短路徑

	1	2	3	4	5	6
距離	0	10	20	∞	20	∞

Bellman-Ford 演算法 – 步驟 2(3)



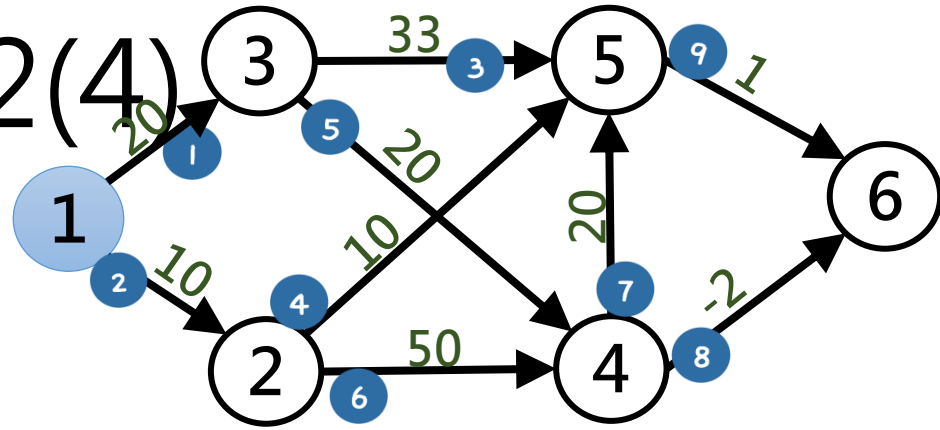
- 加入第 5 條邊：降低節點 3 到節點 4 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	40	20	∞

- 加入第 6 條邊：降低節點 2 到節點 2 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	40	20	60

Bellman-Ford 演算法 – 步驟 2(4)



- 加入第 7 條邊：沒有影響任兩點的距離，不用更改記錄值

	1	2	3	4	5	6
距離	0	10	20	40	20	60

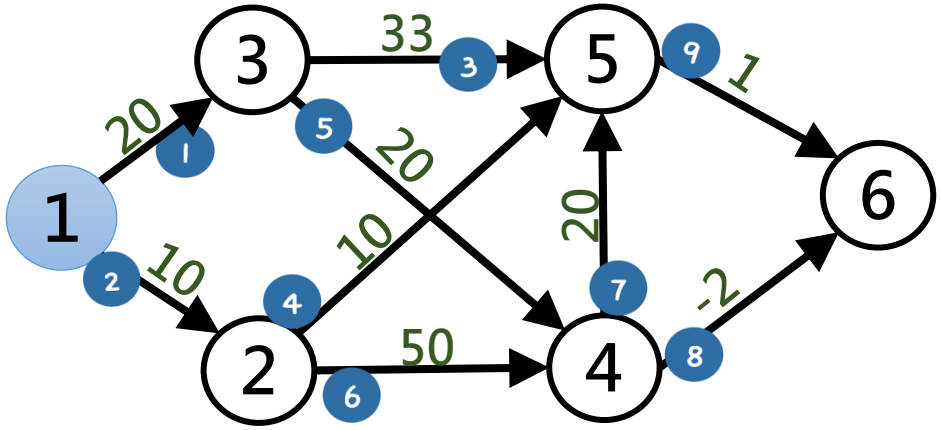
- 加入第 8 條邊：降低節點 4 到節點 6 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	40	20	38

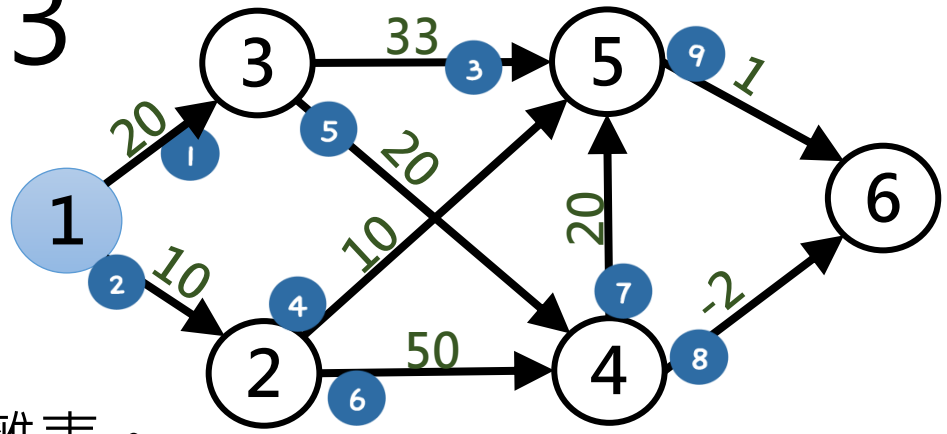
Bellman-Ford 演算法 – 步驟 2(5)

- 加入第 9 條邊：降低節點 5 到節點 6 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	40	20	21



Bellman-Ford 演算法 – 步驟 3



- 重複步驟2，再度更新距離表
- 反覆執行，值到：
  - 全部 9 條邊不論加入哪一條都不會更改到距離表。
  - 最多執行 6 (頂點數目)次

	1	2	3	4	5	6
距離	0	10	20	40	20	21

Bellman-Ford 演算法 – 實作

將指定點到任一點的距離初始值設為無限大

```
void BellmanFord(int vertex, int verticesCount, int edgesCount, int edgesList[][3])
{
 int i, j, isChange;
 int distance[MAX_VERTICES], predecessor[MAX_VERTICES];

 for (i = 0; i<verticesCount; i++) {
  distance[i] = 99999;
  predecessor[i] = i;
 }
 distance[vertex] = 0;
 for (i = 0; i<verticesCount; i++) {
  isChange = 0;
  for (j = 0; j<edgesCount; j++) {
   if (distance[edgesList[j][0]] + edgesList[j][2] < distance[edgesList[j][1]]) {
    distance[edgesList[j][1]] = distance[edgesList[j][0]] + edgesList[j][2];
    predecessor[edgesList[j][1]] = edgesList[j][0];
    isChange = 1;
   }
  }
  if (isChange == 0) break;
 }
}
```

每次檢查一個邊，看加入此邊後會不會降低現有的最短距離

最多執行 N (頂點數) 次，但如果加入任一邊都不會有變化的話，就不需要再反覆檢查了。

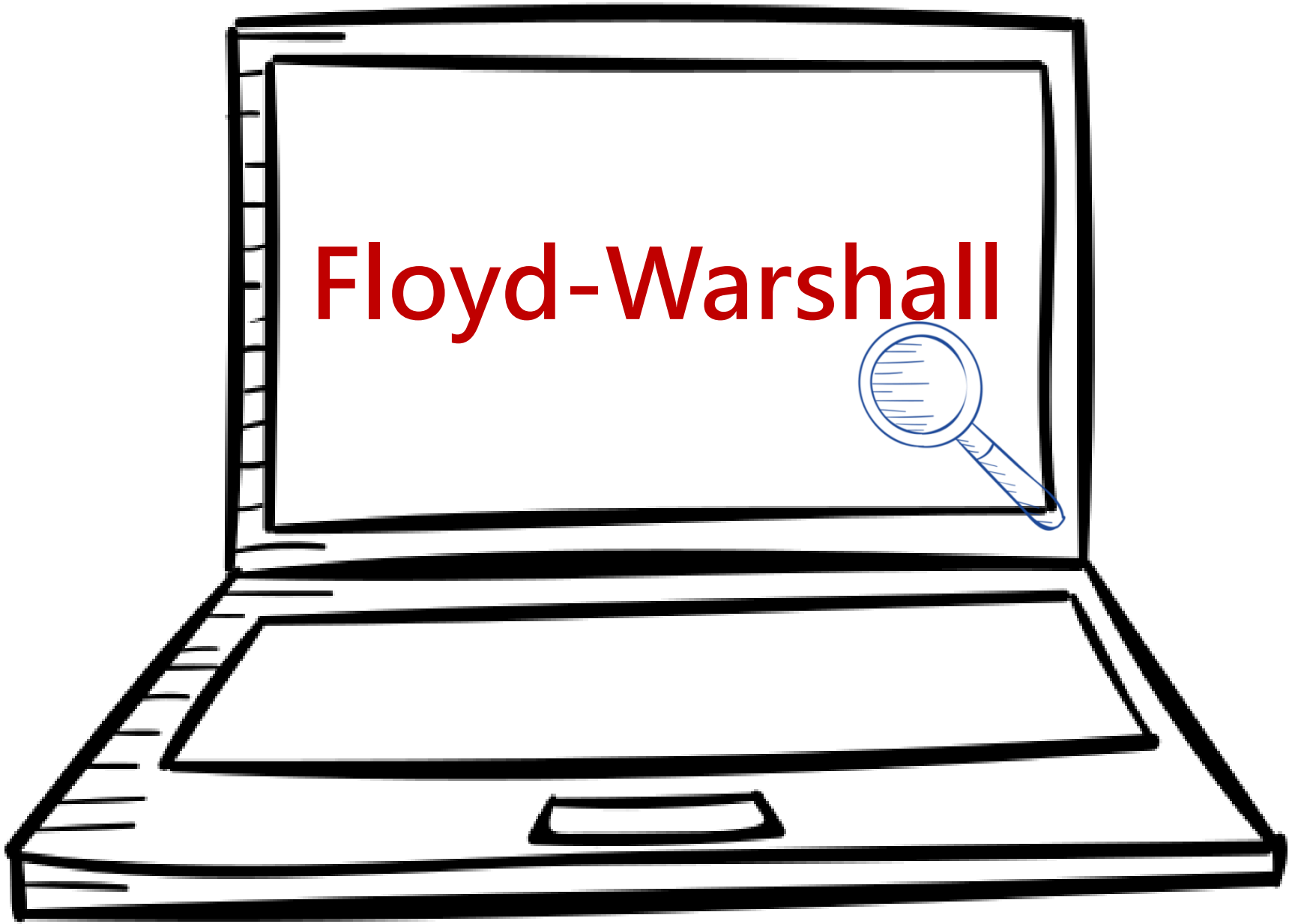
Bellman-Ford 演算法 – 檢查是否有負迴圈

```
isChange = 0;
 for (j = 0; j<edgesCount; j++) {
  if (distance[edgesList[j][0]] + edgesList[j][2] < distance[edgesList[j][1]]) {
   distance[edgesList[j][1]] = distance[edgesList[j][0]] + edgesList[j][2];
   predecessor[edgesList[j][1]] = edgesList[j][0];
   isChange = 1;
  }
 }
```

如果已經執行 N (頂點數) 次的各邊判斷後，再多加一次執行各邊判斷時，仍然有最短距離被更改，就代表圖內有負迴圈，才會一直造成距離變動

還是有問題...

一個圖如果有 10 個頂點，
就要執行 10 次的 Dijkstra
或 Bellman-Ford 才可以得
到圖上任兩點的最短路徑？


Floyd-Warshall

多源最短路徑

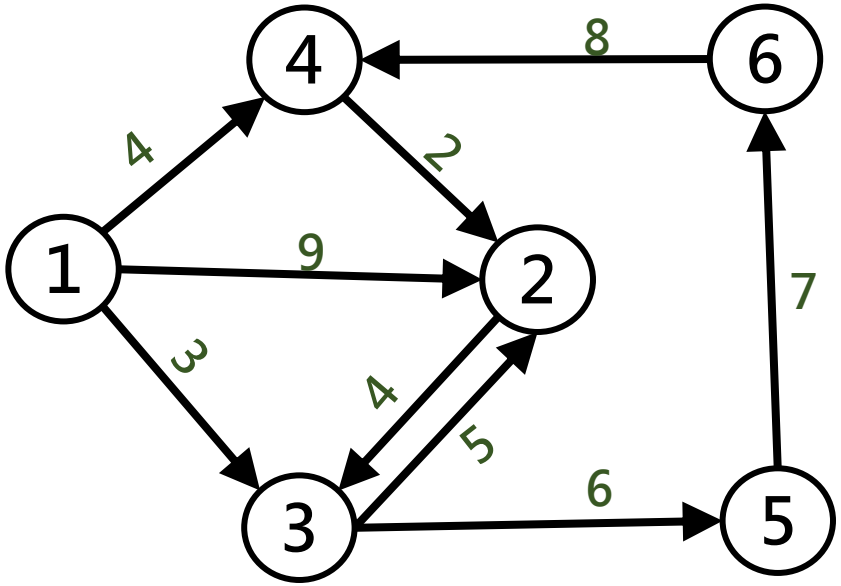
- 指定點到其他任一點的最短路徑問題稱為**單源最短路徑問題**
  - Dijkstra, Bellman-Ford

- 任兩點的最短路徑問題稱為**多源最短路徑問題**
  - Floyd-Warshall

Floyd-Warshall 演算法

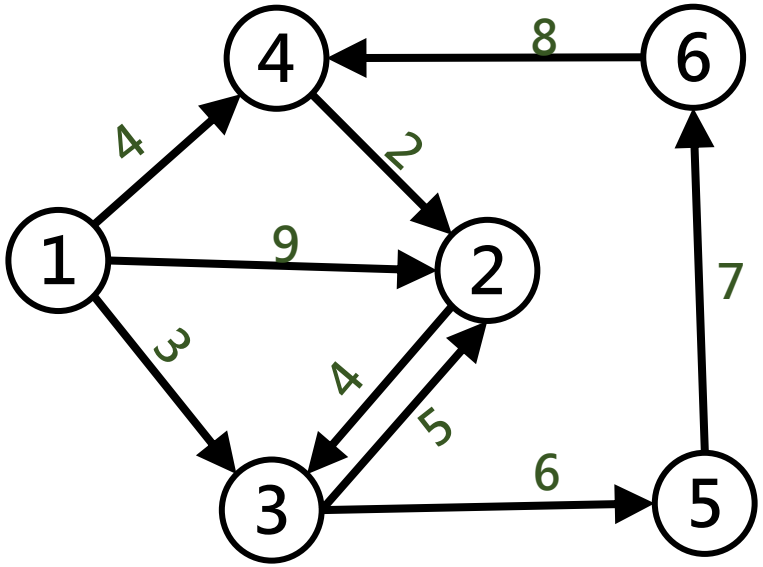
- Floyd-Warshall 是用來處理多源最短路徑問題：計算圖上任兩點的最短路徑。

- Floyd-Warshall 與 Dijkstra 一樣，不處理負權重的圖型問題



Floyd-Warshall 演算法 – 概念

- Floyd-Warshall 與 Dijkstra 類似，都認為任兩點之間的最短路徑有兩種情況：
  - 此兩點之間有邊相連，是相鄰的兩點
  - 此兩點會經過其它點，產生最短路徑



	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	∞	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0

Floyd-Warshall 演算法 – 流程

- 步驟 1: 複製一份圖型的相鄰矩陣資料，將這份矩陣資料做為最短路徑的初始值。

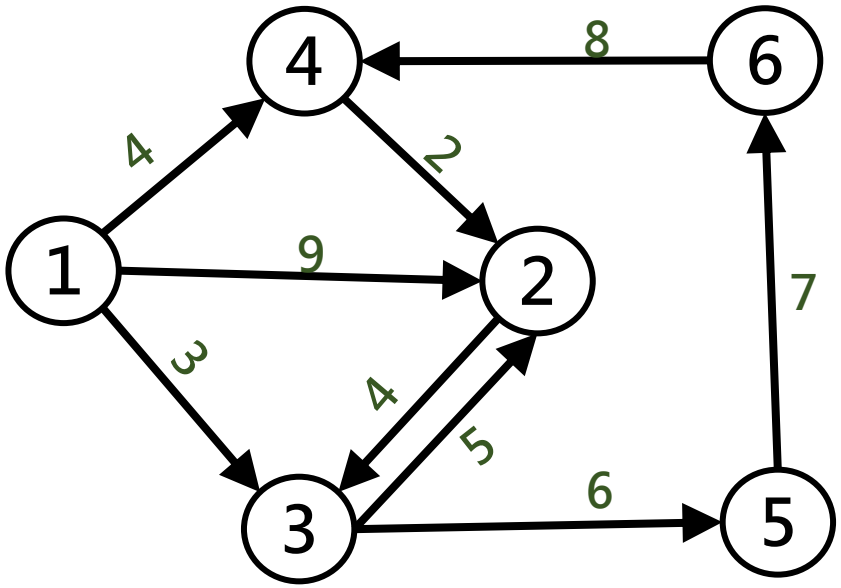
- 步驟 2: 在任兩點之間加入其它節點，確認距離是否有縮短，若有，就更新矩陣內對應的元素值，重複此步驟，直到所有節點都有嘗試放入任兩點節點之間。

- 步驟 3: 全部確認完成後，經由步驟 2 調整後的矩陣資料就是圖中任兩點的最短路徑。

Floyd-Warshall 演算法 – 步驟 1

- 步驟 1: 複製一份圖型的相鄰矩陣資料，將這份矩陣資料做為最短路徑的初始值。

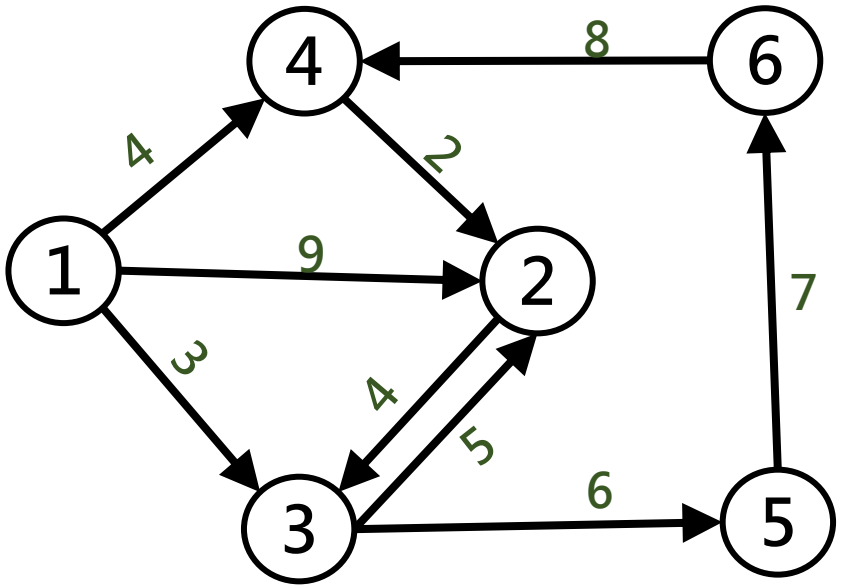
	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	∞	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法 – 步驟 2

- 步驟 2: 在任兩點之間加入節點 1，確認距離是否有縮短

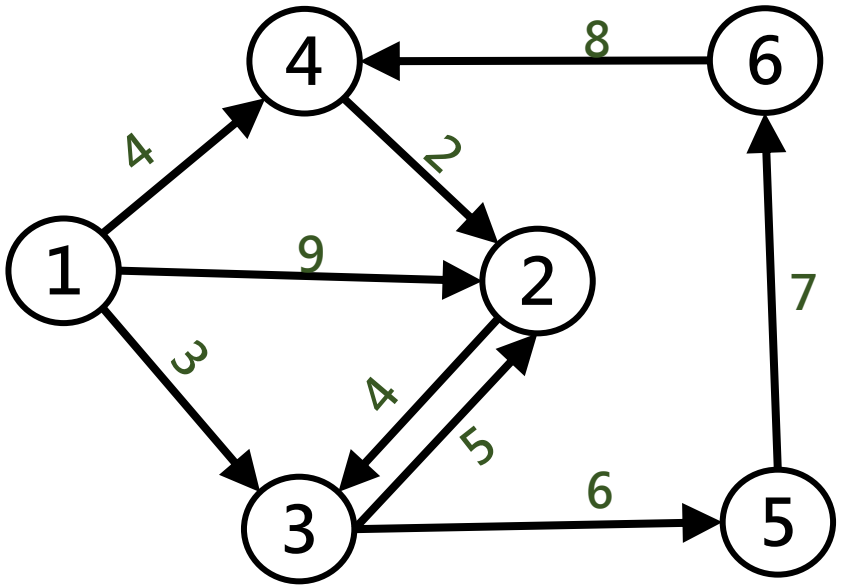
	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	∞	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法 – 步驟 3

- 步驟 3: 在任兩點之間加入節點 2，確認距離是否有縮短

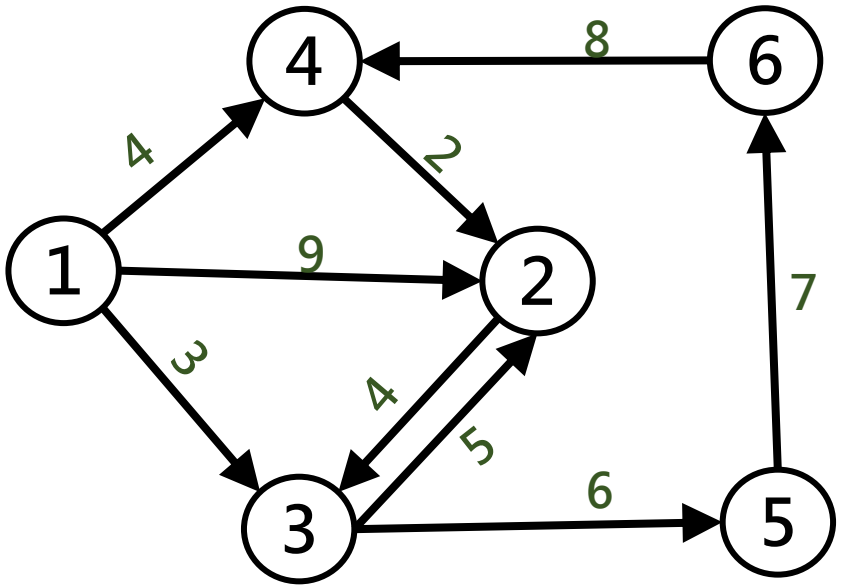
	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	**6**	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法 – 步驟 4

- 步驟 4: 在任兩點之間加入節點 3，確認距離是否有縮短

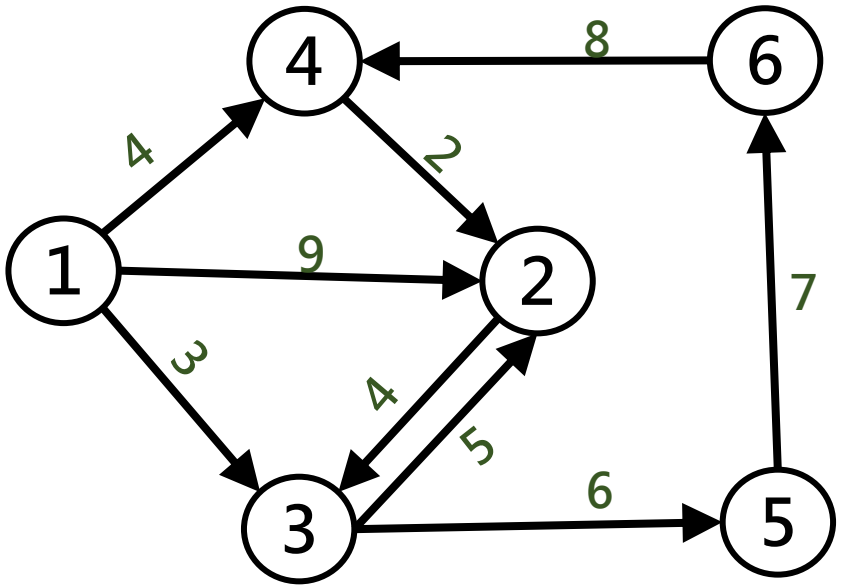
	1	2	3	4	5	6
1	0	**8**	3	4	**9**	∞
2	∞	0	4	∞	**10**	∞
3	∞	5	0	∞	6	∞
4	∞	2	6	0	**12**	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法 – 步驟 5

- 步驟 5: 在任兩點之間加入節點 4，確認距離是否有縮短

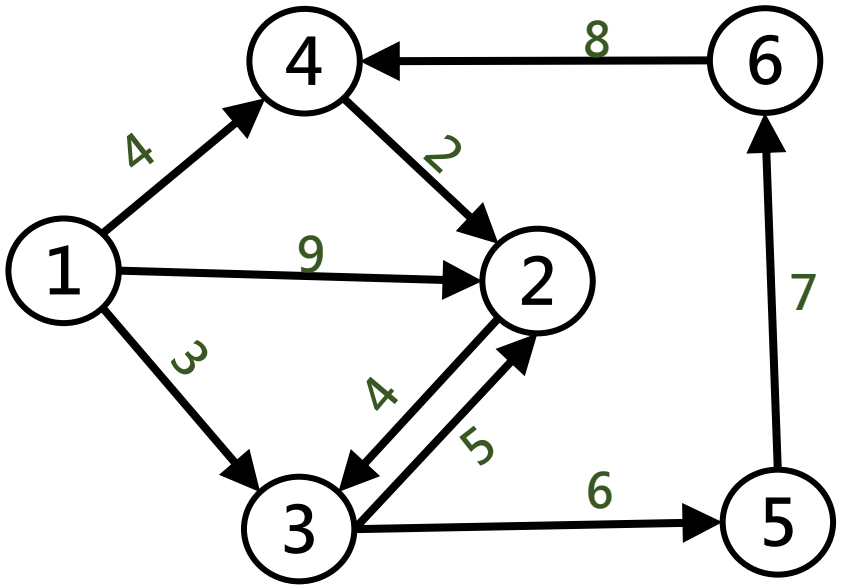
	1	2	3	4	5	6
1	0	**6**	3	4	9	∞
2	∞	0	4	∞	10	∞
3	∞	5	0	∞	6	∞
4	∞	2	6	0	12	∞
5	∞	∞	∞	∞	0	7
6	∞	**10**	**14**	8	**20**	0



Floyd-Warshall 演算法 – 步驟 6

- 步驟 6: 在任兩點之間加入節點 5，確認距離是否有縮短

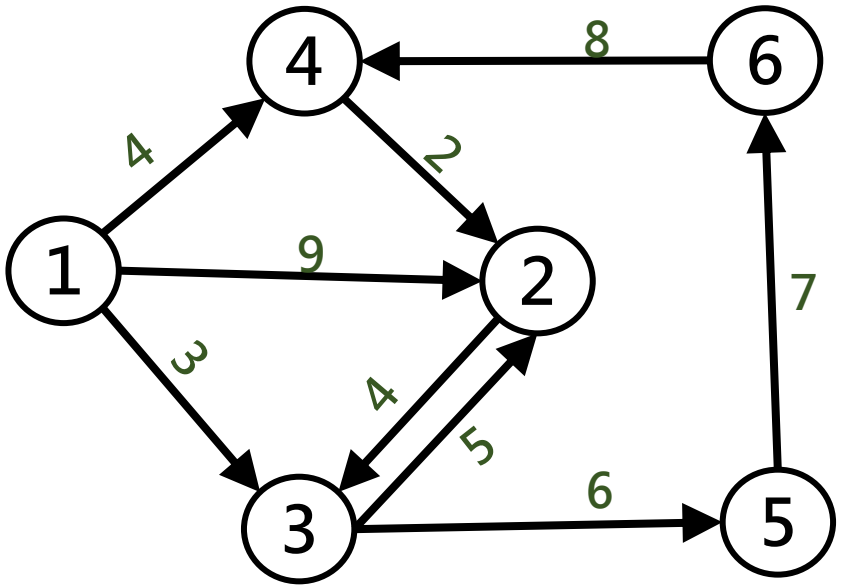
	1	2	3	4	5	6
1	0	6	3	4	9	**16**
2	∞	0	4	∞	10	**17**
3	∞	5	0	∞	6	**13**
4	∞	2	6	0	12	**19**
5	∞	∞	∞	∞	0	7
6	∞	10	14	8	20	0



Floyd-Warshall 演算法 – 步驟 7

- 步驟 7: 在任兩點之間加入節點 6，確認距離是否有縮短

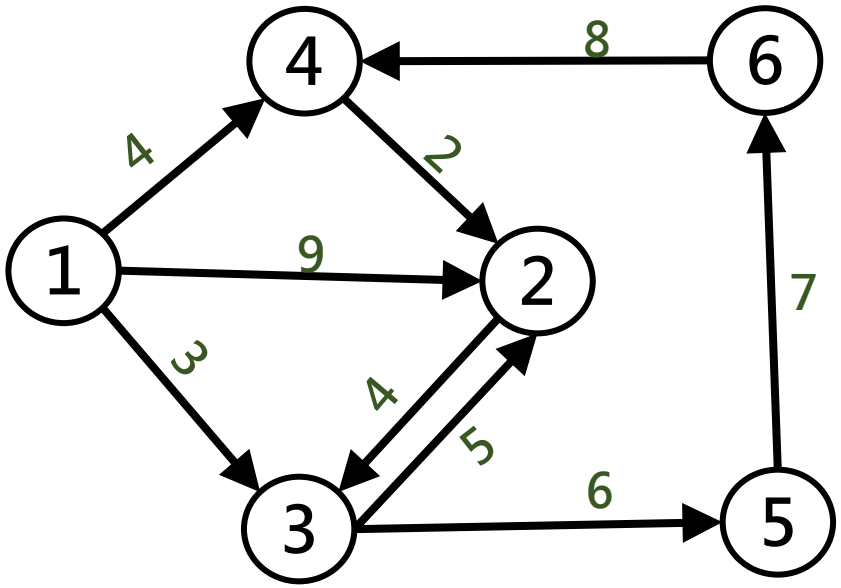
	1	2	3	4	5	6
1	0	6	3	4	9	16
2	∞	0	4	**25**	10	17
3	∞	5	0	**21**	6	13
4	∞	2	6	0	12	19
5	∞	**17**	**21**	**15**	0	7
6	∞	10	14	8	20	0



Floyd-Warshall 演算法 – 步驟 8

- 步驟 8: 全部確認完成後，最後的矩陣資料就是圖中任兩點的最短路徑。

	1	2	3	4	5	6
1	0	6	3	4	9	16
2	∞	0	4	25	10	17
3	∞	5	0	21	6	13
4	∞	2	6	0	12	19
5	∞	17	21	15	0	7
6	∞	10	14	8	20	0



Floyd-Warshall實作

使用相鄰矩陣的表示方式記錄圖讓任兩點的最短距離

比較任兩點之間的最短路徑，是否會因為多經過某節點就減少距離。

```
void FloyeWarshall(int verticesCount,int graph[][MAXVERTICES])
{
 int distance[MAXVERTICES][MAXVERTICES];
 int predecessor[MAXVERTICES][MAXVERTICES];
 int i, j, k;
 for (i = 0; i < verticesCount; i++) {
  for (j = 0; j < verticesCount; j++) {
   distance[i][j] = graph[i][j];
   predecessor[i][j] = -1;
  }
 }
 for (k = 0; k < verticesCount; k++) {
  for (i = 0; i < verticesCount; i++) {
   for (j = 0; j < verticesCount; j++) {
    if (i == j) {continue;} // 不處理自己到自己的情況
    if (i == k || j == k) {continue;} // 中繼點與兩端點一定要不一樣
    if (distance[i][k] + distance[k][j] < distance[i][j]) {
     distance[i][j] = distance[i][k] + distance[k][j];
     predecessor[i][j] = k;
    }
   }
  }
 }
 displayDistance(verticesCount, distance);
}
```

延伸的概念

概念 1: 最短路徑

- 單源最短路徑
  - 無權重：廣度優先搜尋 DFS
  - 正權重：Dijkstra 演算法
  - 負權重：Bellman-Ford 演算法

- 多源最短路徑
  - 正(或無)權重：Floyd-Warshall