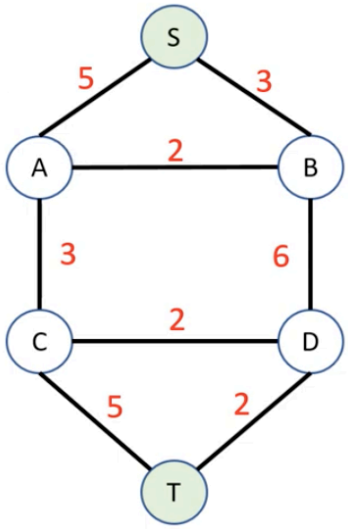
Ch13. 網路流 Flow Network

接下來進入網路流的介紹。本章一開始一樣會先簡介什麼叫網路流問題，以及重要的「最大流最小割」定理。

再來，會介紹產生網路流的演算法：Ford-Fulkerson 和 Edmonds-Karp 演算法。

1. 網路流問題簡介



在網路流問題中，圖一樣由頂點和邊組成。

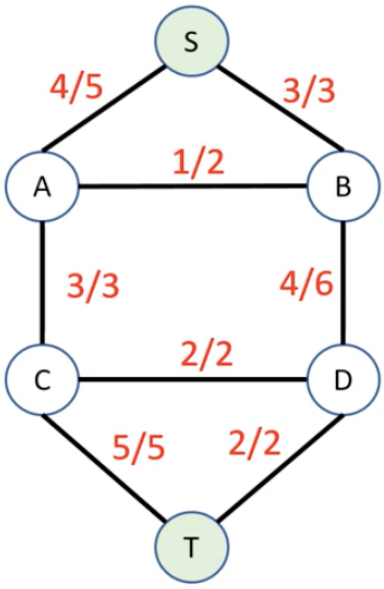
其中，邊可以想像成是「水管」，許多水管共同組成了一個水管分布圖。邊上的權重是水管的容量上限，通過該水管的水不能超過這個正數。

若頂點也有權重，代表水管接合處的容量上限，但一般來說，不考慮頂點的權重。有兩個特殊的頂點「源點 Source」和「匯點 Sink」，源點縮寫為 S、匯點縮寫為 T，要找到的就是從源點到匯點能夠支撐的最大流量。

也可以把 S 想像成是家裡的電腦，T 想像成是公司的電腦，而每個邊上的權重是頻寬限制，網路流問題要解決的就是如何能夠讓傳輸資料最快，達到頻寬的「最大化」。

或者把 S 當作「水庫」、T 當作「城市」，那麼尋找的就是怎麼樣可以讓水庫運到城市的水流最大化。

（1）網路流的例子



上圖中，從 S 到 T 的「水流」由四個「支流」構成：

$S->A->C->T$：3

$S->A->B->D->C->T$：1

$S->B->D->T$：2

$S->B->D->C->T$：1

三單位的水流從 S 經過 A、C 流到 T，而一單位的水流由 S 出發、經過 A 後，改走 B、D、C 到 T，因此經過 $\overline{SA}$ 這個水流總量為 3+1=4，小於該邊的權重 5，代表邊可以承受這個水流。

類似的，$\overline{BD}$ 總共有 3 個支流經過，水流量分別為 1、2、1，總水量不超過它的權重 6。

網路流的演算法就是要能夠自動算出整個網路流的「最大流量」。

（2）網路流名詞定義

A. 網路 Network：一個有向圖 G
B. 源點與匯點 Source and Sink：源點 S 為網路流的起點、匯點 T 為網路流的終點，其餘點為中間點
C. 流量 Flow：每條邊 / 弧上的數值，表示目前經過該邊 / 弧的流量，記為 $F(u,v)$

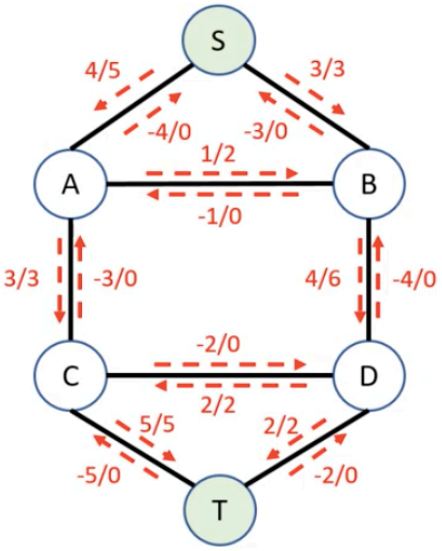
D. 容量 Capacity：每條邊 / 弧上的數值，表示該邊 / 弧的流量上限，記為 $C(u,v)$
E. 剩餘容量 Residual Capacity：每條邊 / 弧上的容量減去流量，有 $C_f(u,v) = C(u,v) - F(u,v)$
F. 剩餘網路 Residual Network：剩餘容量的集合
G. 網路流量 Flow of Network：由原點出發至匯點的總流量，當其為該網路能達到的最大流量時，稱為最大流 Maximum Flow

$$|f| = \sum f(s,v) = \sum f(v,t)$$

（3）弧 Arcs 的分類

A. 不飽和弧：弧上的流量小於其容量
$F(u,v) < C(u,v)$
B. 飽和弧：弧上的流量恰好等於其容量
$F(u,v) = C(u,v)$
C. 零流弧：弧上的流量為零
$F(u,v) = 0$
D. 非零流弧：弧上的流量不為零
$F(u,v) \neq 0$
E. 前向弧與後向弧：若 P 為源點 S 到匯點 T 的路徑，該路徑中，與路徑前進方向相同的，稱為前向弧，方向相反的，稱為後向弧（見後述）

（4）可行流 Positive Flow

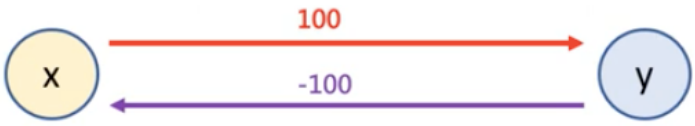


若一個流符合以下三個限制，就稱為一個可行流：

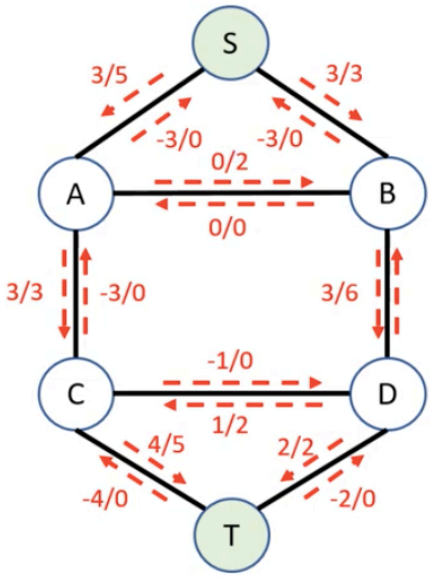
A. 容量限制 Capacity Constraints
每條邊上，流量不大於容量，即 $F(u,v) \leq C(u,v)$

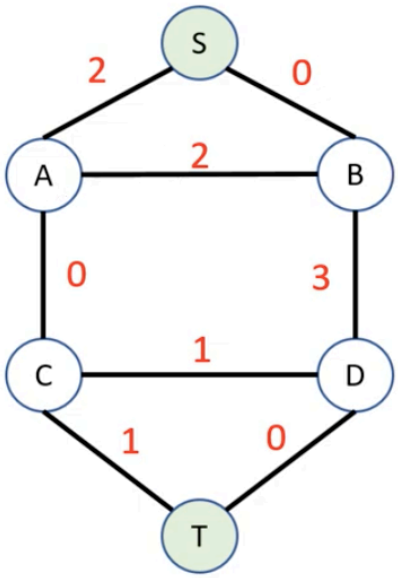
B. 流量守恆 Flow Conservation
任意節點的流入量必等於流出量，因此從原點流出的總流量必等於流入匯點的總流量：$\sum F(i,u) = \sum F(u,j)$

C. 斜對稱性 Skew Symmetry
u 到 v 的淨流量加上 v 到 u 的淨流量必為 0，$F(u,v) + F(v,u) = 0$。
比如由 x 向 y 的流量是 100，那麼 y 向 x 的流量就視為 -100。



（5）剩餘容量

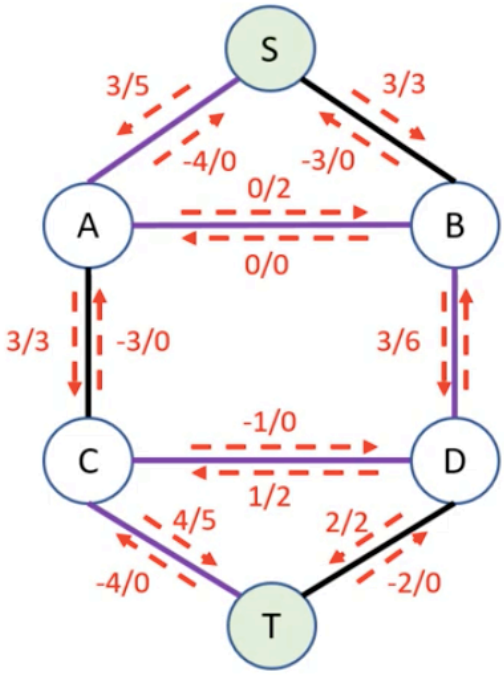




每條邊 / 弧上的容量減去流量，就是該邊 / 弧的剩餘容量，即 $C_f(u,v) = C(u,v) - F(u,v)$。

透過把每條邊的剩餘容量畫出，就可以得到「殘餘網路 Residual Network」，觀察右上圖中，每條邊上標示的數字都是左上圖中該邊的容量減去流量。

（6）增廣路徑 Augmenting Path



若 f 是一個可行流、P 是由原點到匯點的一條路徑，且 P 滿足以下條件，就稱 P 為可行流 f 的一條增廣路徑：P 上的每條前向弧都是非飽和弧，也就是說，「順著」P 的方向上走，不會遇到流量已經等於容量的弧。

因為 P 上的前向弧都是非飽和弧，有 $C_f(u_k, u_{k+1}) > 0$，因此還可以再加上多餘的流量，使得整個流仍是可行流，網路流中的總流量增加。

比如上圖中，一個增廣路徑是 $S -> A -> B -> D -> C -> T$，因為中間的每個前向弧剩餘容量至少都大於等於 1，因此還可以透過該路徑增加流量 1。

透過殘餘網路，可以很容易看出是否存在增廣路徑。

2. 網路流問題的演算法簡介

（1）Ford-Fulkerson method

Ford-Fulkerson 的概念，是在殘餘網路中隨意找一條增廣路徑，並利用該增廣路徑來增加流量，隨後修正殘餘網路，不斷重複前述，直到找不到增廣路徑為止。

一條增廣路徑能夠增加的流量，會是該路徑中「還可以增加的流量最少」的邊的流量。這類似經濟學中的木桶原理，即一個木桶能夠裝的水的高度，由共同組成該木桶的木板中「長度最短」者決定，如果水面超過該最短木板，水就會向外溢出。

Ford-Fulkerson 演算法的時間複雜度為 $O(Ef)$，其中 E 為邊數、f 為最大流流量：因為每次搜尋增廣路徑的時間為 $V+E$（頂點數加上邊數），而搜尋到一個增廣路徑後，最少可能只能增加 1 單位流量，因此最多需要花費 $O((E+V)f)$ 的時間，又通常 $E \geq V$，因此 $O((E+V)f) = O(Ef)$。

（2）Edmonds-Karps algorithm

Ford-Fulkerson 並沒有指定找到增廣路徑的方式，而 Edmonds-Karps 演算法是採用「廣度優先搜尋 BFS」來實作 Ford-Fulkerson 方法。

由於廣度優先搜尋的特性，每次找到的增廣路徑必定會經過最少的邊 / 弧。

每次找出增廣路徑並修正殘餘網路後，其中一條路徑就被「消除」掉了，而可以證明網路中最多只有 $O(VE)$ 條增廣路徑、尋找一次增廣路徑的複雜度為 $O(E)$，因此 Edmonds-Karps 演算法的總複雜度為 $O(E(VE)) = O(VE^2)$。

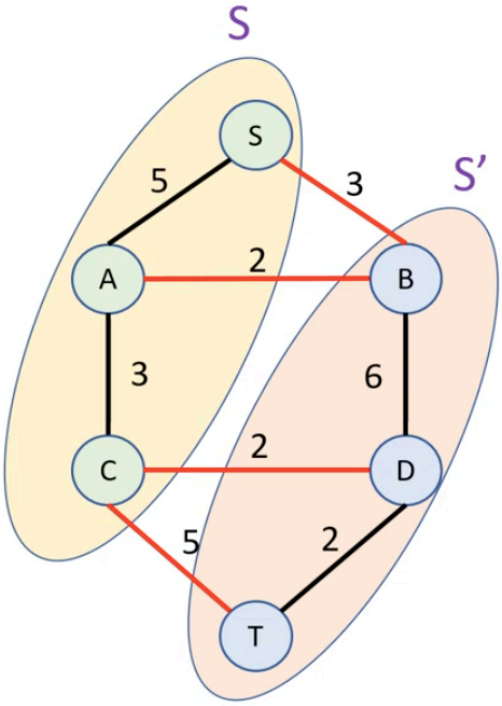
（3）最小割

在一個流量網路 $G=(V,A)$ 中（V 是頂點、A 是弧），可以把所有頂點 V 分為不相交（不相連）的兩個集合 S 與 S'，且源點在 S 中、匯點在 S' 中。

如果 A' 是 A 的最小子集，使得 G 中去除 A' 後，就成為兩個不相交的子圖 $G_1(S,A_1)$ 與 $G_2(S',A_2)$，則稱 A' 是 S 與 S' 的割集。

也就是說，A' 是 G 的其中一些邊，把這些邊拿掉後，G 裡面的所有頂點就被分成兩個集合 S 與 S'，兩個集合間沒有任何邊相連，而 A' 只含有需要達到這個目的所必須的邊，不會含有多餘的邊，因此是「最小子集」。

A' 中含有的弧容量總和，就是這個割的容量。然而符合上述條件的割集可能有複數個，當 A' 是這些割集中容量最小者時，就稱 A' 為該圖的最小割。



舉例來說，上圖中所有頂點被分為 $S = \{S, A, C\}$、$S' = \{B, D, T\}$，則 $A' = \{\overline{SB}, \overline{AB}, \overline{CD}, \overline{CT}\}$。

A' 的容量為四條邊的容量和，為 $3 + 2 + 2 + 5 = 12$。

（4）最大流最小割定理 Maximum Flow Minimum Cut Theorem

最大流最小割定理說明在一個流量網路 $G = (V, A)$ 中，下面三個條件等價：

A. 一個流 f 為 G 的最大流
B. G 的殘餘網路中沒有增廣路徑
C. 存在一個割 C，其容量為流量 f

如果從 G 的殘餘網路中還可以找到增廣路徑，那麼流量就還可以增加，因此前兩個條件必定等價。

而因為源點和匯點分別在兩個集合 S 和 S' 中，能夠產生的最大流必定要經過這個割集中的邊，即上頁圖中的 $\overline{SB}$、$\overline{AB}$、$\overline{CD}$、$\overline{CT}$。

已知 C 為圖中的最小割，產生最大流量時，C 中的每個邊的流量必定與容量相等，即 $C_f(u,v) = C(u,v) - F(u,v) = 0$，否則可以產生增廣路徑而使流量更大。

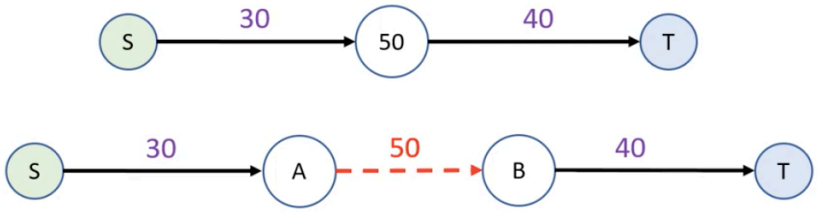
如此可以導出最大流最小割定理：尋找網路流中的最大流，與尋找網路流中的最小割是等價的，也就是說，最小割的容量就是該網路能夠產生的最大流量。

而要尋找最小割，可以從源點開始，沿著殘餘網路的前向弧搜索，並且找到每條路徑中第一條容量為 0 的弧，這些弧已經不能再加入額外流量，因此這些弧的集合就是最小割。

（5）網路流問題的變化

A. 點容量 Capacity of Vertices

一般的網路流問題只限制邊上的流量，但若頂點上亦有流量限制，要如何處理呢？



事實上，只要把該頂點拆成兩個點和一條邊，使得邊上的流量限制與原先頂點上的流量限制相同，就可以使其變回一般的網路流問題了。

B. 多個源點與匯點 Multiple Source / Sink Vertices

一般的網路流問題中，只有一個源點與一個匯點，如果同時給定多個源點與匯點，則可以把源點間同時連到一個新增的頂點，匯點間也同時連到一個新增的頂點，過程中加上的邊容量都設定為無限大，這樣一來，就可以使其變為一般的網路流問題。

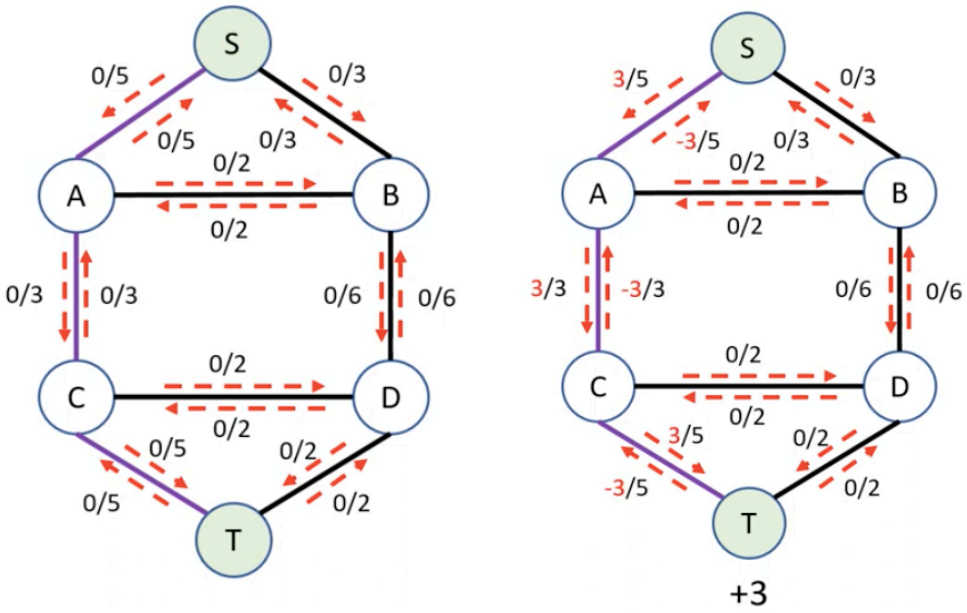
上圖中，原先有 3 個源點 S_1、S_2、S_3，把它們各自用一條容量為無限大的邊連到新增的源點 S 上，匯點 T_1、T_2、T_3 也連接到新增的匯點 T 上，就成為一般的網路流問題。

3. 實作網路流的演算法

（1）Ford-Fulkerson method

Ford-Fulkerson 方法是不斷在殘餘網路中尋找增廣路徑，並利用該增廣路徑增加流量，直到找不到增廣路徑時，累積的流量就是該網路的最大流量。也就是說，最大流量是由許多小細流匯聚而成。

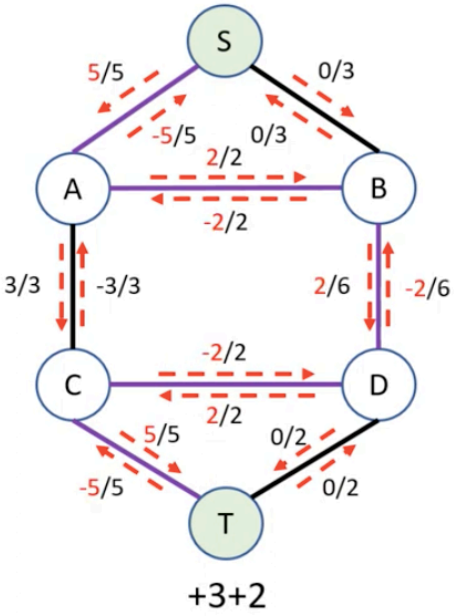
A. Ford-Fulkerson 方法的進行過程

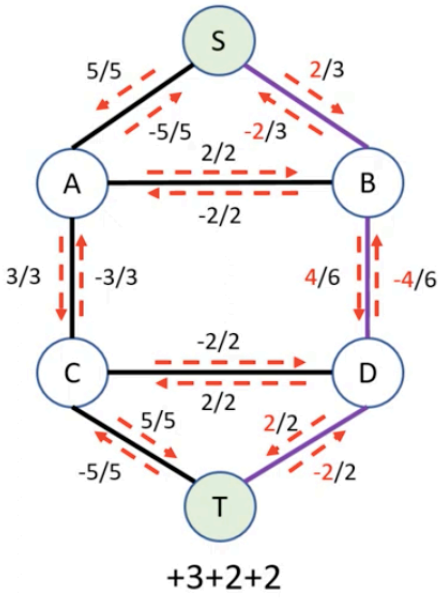


一開始，在左上圖中隨意尋找一條增廣路徑，比如 $S->A->C->T$，因為路徑中剩餘容量最小的弧 AC 只能再增加 3 單位流量，因此把 3 單位流量加到累積的流量中，並且修正各弧上目前的流量，如右上圖。

注意到，反向的弧上流量是負的，比如 SA 的流量是 3 時，反向的 AS 流量就是 -3。

接下來，再任意尋找一條增廣路徑，如 $S->A->B->D->C->T$，因為該路徑上包括 SA、AB、DC、CT 等弧的剩餘流量都只有 2，因此把累積流量加上 2 單位，並且修正各弧上的流量。





繼續尋找增廣路徑，如 $S->B->D->T$ 還可以增加 2 單位流量，因此累積流量加上 2 單位，並修正各弧上流量。

這時，在殘餘網路上已經找不到增廣路徑，因此最大流量就是 $3+2+2=7$。

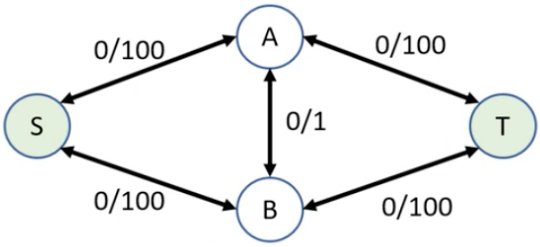
B. Ford-Fulkerson 方法的步驟

a. 初始化所有 flow，$f(u,v) = 0$
b. 找出一個從 $S->T$ 的增廣路徑
   - 該路徑上所有弧都滿足 $C_f(u,v) > 0$
   - 找出路徑上殘餘容量最小的弧
   - $C_f(p) = min\{C_f(u,v) : (u,v) \in p\}$
c. 對路徑上的所有弧 $e(u,v) \in p$，更新流量
   - $f(u,v) = f(u,v) + C_f(p)$
   - $f(v,u) = f(v,u) - C_f(p)$

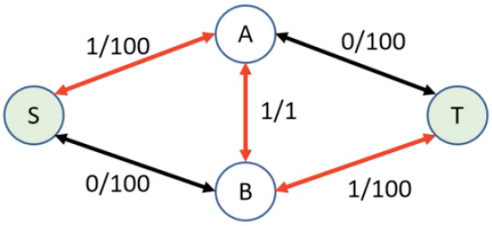
透過上述步驟累積的總流量就是最大流量，時間複雜度為 $O(Ef)$。

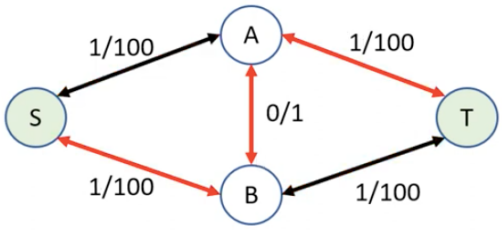
C. Ford-Fulkerson 方法的限制

Ford-Fulkerson 方法並沒有指定一種找到增廣路徑的方式，因此視採用的尋找方法而定，有時會需要重複尋找非常多次。

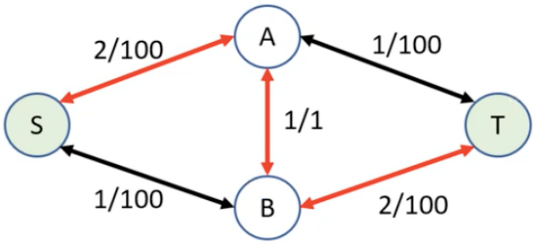


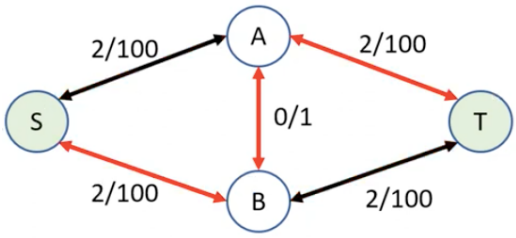
比如上圖中，如果第一次找到的路徑是 $S->A->B->T$，因為弧 AB 只能增加 1 單位流量，因此把累積流量增加 1，並修正剩餘流量。





接下來，可以再找到一個增廣路徑 $S->B->A->T$（注意這次走的是 BA，為 AB 的反向弧，所以 AB 的「流量/容量」由上一輪處理完時的 1/1 重新變為 0/1，BA 的流量由上一輪的 -1/1 變為 0/1），同樣只能再增加 1 單位累積流量。





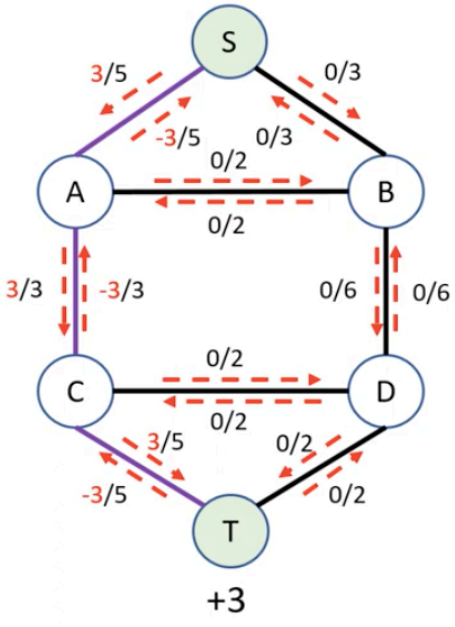
之後重複許多次這個步驟，直到 SA、AT、SB、BT 都變為 100/100 為止，因為每次都只增加一單位累積流量，而最後發現最大流量為 200，所以總共尋找了 200 次增廣路徑，時間複雜度為 $O(Ef)$，因為找尋一次增廣路徑要花費 $O(E)$ 的時間，而總共尋找了 $O(f)$ 次。

當 f 非常大時，使用不適當的方式來找尋增廣路徑，可能導致花費大量時間。

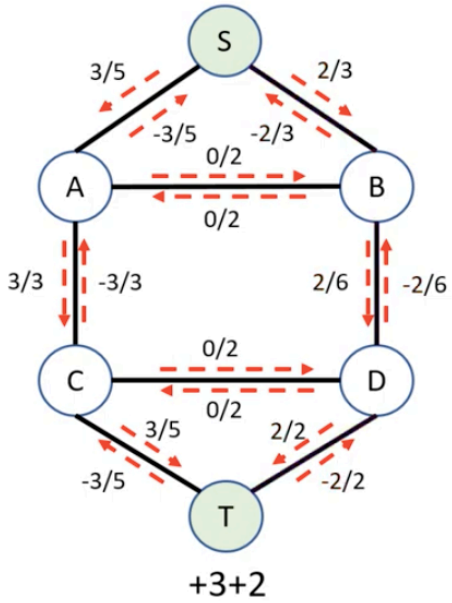
（2）Edmonds-Karp Algorithm

Edmonds-Karp 演算法採用「廣度優先搜尋 BFS」來尋找增廣路徑，因為 BFS 會優先找到邊數最少的路徑，可以避免尋找非常多次增廣路徑的情況（見後述）。

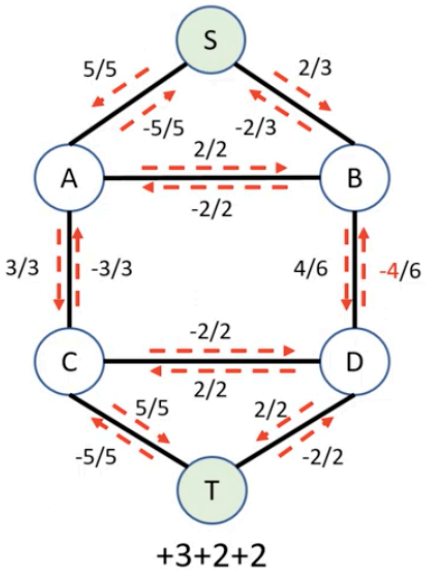
A. Edmonds-Karp 演算法的進行過程



首先，在上圖中透過 BFS 來尋找增廣路徑，會找到邊數最少的增廣路徑之一 $S->A->C->T$，使得累積流量增加 3 單位，並修改目前流量。



再來，BFS 會找到此時邊數最少的增廣路徑 $S->B->D->T$，使得累積流量增加 2 單位。

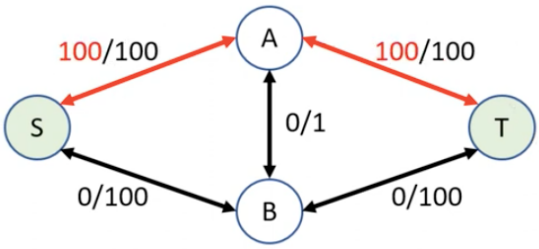


最後，BFS 找到此時剩下的增廣路徑 $S->A->B->D->C->T$，使累積流量增加 2 單位，且發現圖中沒有其他增廣路徑，因此最大流量為 $3+2+2=7$。

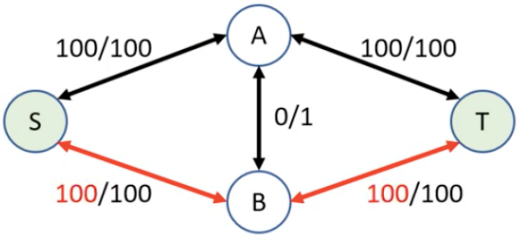
B. Edmonds-Karp 方法的步驟
   a. 初始化所有 flow，$f(u,v) = 0$
   b. 以「廣度優先搜尋 BFS」找出一個從 $S->T$ 的增廣路徑
      - 該路徑上所有弧都滿足 $C_f(u,v) > 0$
      - 找出路徑上殘餘容量最小的弧
      - $C_f(p) = min\{C_f(u,v) : (u,v) \in p\}$
   c. 對路徑上的所有弧 $e(u,v) \in p$，更新流量
      - $f(u,v) = f(u,v) + C_f(p)$
      - $f(v,u) = f(v,u) - C_f(p)$

總時間複雜度為 $O(VE^2)$，$O(V+E) = O(E)$ 為每次搜尋增廣路徑的時間，增廣路徑最多可能有 $O(VE)$ 條。

C. 用 Edmonds-Karps 演算法來避免 Ford-Fulkerson 方法的可能問題

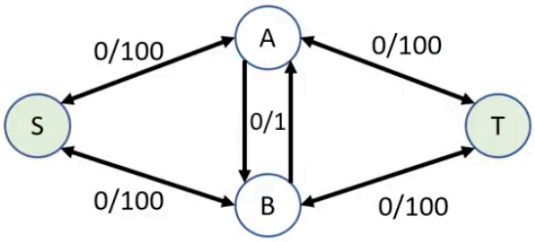


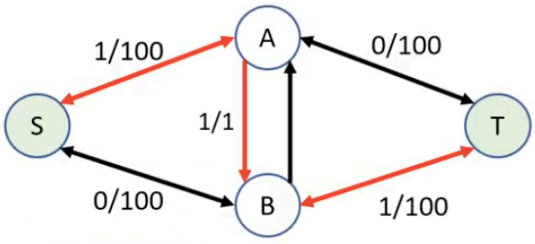
在上例中，因為 BFS 會優先找到邊數最短的增廣路徑，所以先找到路徑 $S->A->T$，並增加累積流量 100。



再來，找到此時邊數最少的增廣路徑 $S->B->T$，增加累積流量 100，得到最大流量為 $100+100=200$，避免了選擇增廣路徑方法不當導致花費時間過多的問題。

D. 為何最多只有 $O(VE)$ 條增廣路徑



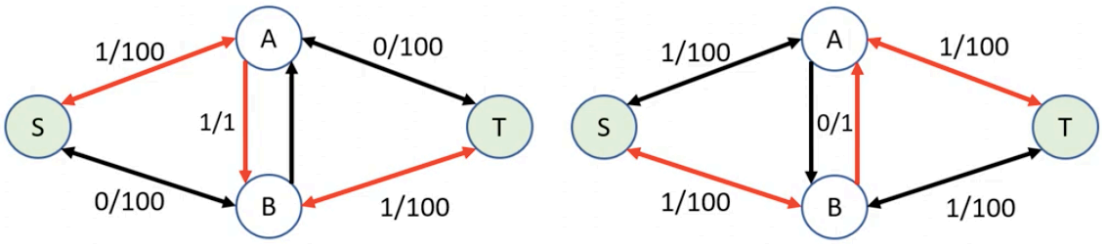


在 Edmonds-Karps 演算法中，首先尋找一條增廣路徑，如果這條路徑中的數個弧裡，由頂點 u 連到頂點 v 的弧 $e(u,v)$ 為路徑中剩餘容量最小的弧，那經過這輪處理之後，$e(u,v)$ 就變成了一個飽和弧，剩餘流量為 0，導致下一輪在尋找增廣路徑時，路徑中一定不會出現弧 $e(u,v)$。

比如上圖中，第一次找到的增廣路徑如果是 $S->A->B->T$，其中 $e(A,B)$ 為 SA、AB、BT 三者中剩餘流量最小的弧，所以經過這次增廣後，$e(A,B)$ 的

流量等於其容量，標示為 1/1，這導致下一輪一定不能走 $e(A,B)$。

但是這並不代表之後的任何增廣路徑都不會再出現 $e(A,B)$，因為一旦走了 $B->A$ 這個方向，就會使 $e(B,A)$ 的流量增加，相對的 $e(A,B)$ 的流量就會減少，比如若下一輪走 $S->B->A->T$，那 $e(A,B)$ 就從 1/1 重新變回 0/1，而可能出現在後續找到的增廣路徑中。



上面提到的一開始讓 $e(u,v)$ 變成飽和弧的那條增廣路徑裡，一定是由頂點 u 走到頂點 v，比如 $S->A->B->T$ 這條路徑中，是由頂點 A 經過 $e(A,B)$ 走到頂點 B，所以該路徑中，$\delta f(s,B)=2=\delta f(s,A)+1$。一般來說，$\delta f(s,v)=\delta f(s,u)+1$，其中 $\delta f(s,u)$ 代表的是增廣路徑中由源點 S 走到頂點 u 的邊數，$\delta f(s,v)$ 是由 S 走到頂點 v 的邊數。

走了 $S->A->B->T$ 後，AB 弧變成飽和弧，若之後有一條增廣路徑會使其變成不飽和弧，路徑中一定有 $B->A$，比如走 $S->B->A->T$ 使得 AB 弧由 1/1 變為 0/1，重新變成不飽和弧。

在 $S->B->A->T$ 中，$\delta f'(s,A)=2=\delta f'(s,B)+1$，$\delta f'(s,A)$ 表示由 S 走到 A 經過的邊數，$\delta f'(s,B)$ 表示由 S 走到 B 經過的邊數。一般來說，讓 $e(u,v)$ 由飽和弧變為不飽和弧的增廣路徑中，$\delta f'(s,v)=\delta f'(s,u)-1$，因為該路徑是由頂點 v 往頂點 u 方向走。

綜合上述，有兩個關係式

$$\delta f(s,v)=\delta f(s,u)+1$$
$$\delta f'(s,v)=\delta f'(s,u)-1$$

又因為 BFS 會優先找到邊數最少的增廣路徑（對整條增廣路徑中的一部分，如從 S 到頂點 v 也成立），所以後面找到的增廣路徑邊數一定與前面找到的增廣路徑邊數相同或更多。

這使得 $\delta f(s,u)+1\le\delta f'(s,u)-1$，整理一下，有 $\delta f(s,u)+2\le\delta f'(s,u)$。

又邊數最多的增廣路徑可能有 $|V|-1$ 個邊（走過全部共 $|V|$ 個頂點），所以既然每次增廣路徑經過 $e(u,v)$ 時，$\delta f(s,u)$ 會至少比前一次增加兩個邊，那麼 $e(u,v)$ 是增廣路徑中容量最小的弧（critical edge）的次數不會超過 $\frac{|V|-1}{2}$ 次。

比如找到的所有增廣路徑當中，有些含有 $e(u,v)$。當中第一個從 s 出發只走一個邊就到 u、第二個走三個邊才到 u、...，若圖中總共有 8 個頂點，最長的增廣路徑會有 7 個邊，那麼含有 $e(u,v)$ 的增廣路徑最多只能出現 $\frac{7-1}{2}=3$ 次，這三次中 $\delta f(s,u)$ 分別為 1、3、5。

再來，每次增廣時，一定至少有一個弧作為 critical edge，弧的總數（包含正向與反向）是邊數的兩倍，即 $2|E|$。有 $2|E|$ 個弧，每個弧成為 critical edge 的次數最多 $\frac{|V|-1}{2}$ 次，因此總增廣次數為 $O(2|E|\times\left(\frac{|V|-1}{2}\right))=O(|E|(|V|-1))=O(VE)$。

E. 實作 Edmonds-Karp 演算法

Edmonds-Karp 演算法

```
1  #include<iostream>
2  #include<stdlib.h>
3  #include<vector>
4  #include<list>
5  #include<queue>
6  using namespace std;
7
8  typedef struct{
9      int from;
10     int to;
11     // int weight;      // 邊上的權重 weight 改用 flow 來表示
12     int flow;           // 目前流量
13     int capacity;       // 容量
14 } edge;
15
16 class Graph{
17 private:
```

```
18      vector<list<edge*>> edges;
19      int vertex;
20  public:
21      // 網路流的函式，兩個參數為源點與匯點編號
22      int Network_Flow(int, int);
23      Graph(int);
24      void Print_Edges();
25      void Add_Edge(int, int, int=1);
26  };
27
28  int Graph::Network_Flow(int s, int t){
29
30      // 調整頂點編號與 index 對應
31      s--;
32      t--;
33
34      // 記錄最大流流量
35      int max_flow = 0;
36
37      while (true){
38
39          // Step1: 用 BFS 找到增廣路徑
40
41          queue<vector<int>> data;
42          bool found = false;       // 記錄是否找到增廣路徑
43          vector<int> initial_path, current, visited(vertex);
44          initial_path.push_back(s);
45          visited[s] = 1;           // 沒走過：0、有走過：1
46
47          data.push(initial_path);
48
49          while (!data.empty()){
50
51              // 取出 queue 中的第一個路徑
52              current = data.front();
53              data.pop();
54
55              // 檢查路徑 current 是否已經到達匯點 t
```

```
56                  int now = current.back();
57
58                  // 若已經到達匯點 t
59                  if (now == t){
60                      // BFS 完成並回傳路徑
61                      found = true;
62                      break;
63                  }
64                  // 還未到達匯點 t
65                  else {
66
67                      for (auto iter=edges[now].begin() ;
68                      iter!=edges[now].end() ; iter++){
69
70                          // 往有剩餘流量、還未造訪過的頂點方向移動
71                          if ((*iter)->capacity > (*iter)->flow &&
72                          visited[(*iter)->to]==0){
73
74                              vector<int> new_path = current;
75
76                              // 在原本的路徑 current 後面
77                              // 加上一個新造訪的頂點
78                              new_path.push_back((*iter)->to);
79
80                              // 記錄該頂點已經造訪過
81                              // 避免迴圈後續重複處理
82                              visited[(*iter)->to] = 1;
83                              data.push(new_path);
84
85                          } // end of inner if
86                      // end of for
87
88                  } // end of outer if
89              } // end of while
90
91              // 若沒有找到增廣路徑，結束迴圈
92              if (!found){ break; }
93
```

```
94          // Step2: 找到增廣路徑中剩餘容量最小的邊
95          int minimal = 2147483647; // INT_MAX
96
97          for (int i=0 ; i<current.size()-1 ; i++){
98
99              int u = current[i];
100             int v = current[i+1];
101
102             // 要得到邊(u,v)的剩餘容量，只能從 edges 中慢慢尋找
103             // 若改用矩陣方式儲存圖，會較方便
104             for (auto iter=edges[u].begin() ; iter!=edges[u].end() ; iter++){
105                 // 在 u 的出邊中找到指向 v 的那條
106                 if ((*iter)->to == v){
107                     // 比較 minimal 與邊的剩餘容量
108                     if ((*iter)->capacity - (*iter->flow)<minimal){
109                         minimal = (*iter)->capacity - (*iter)->flow;
110                     }
111                 }
112             } // end of inner for
113         } // end of outer for
114
115         // Step3: 更新殘餘網路
116         for (int i=0 ; i<current.size()-1 ; i++){
117
118             int u = current[i];
119             int v = current[i+1];
120
121             for (auto iter=edges[u].begin() ; iter!=edges[u].end() ; iter++){
122                 // 在 u 的出邊中找到指向 v 的那條
123                 if ((*iter)->to == v){
124                     // 邊的流量加上本次累積流量的增加值
125                     (*iter)->flow += minimal;
126                 }
127             }
128
129         }
130
131         // 累積流量增加
```

```
132                 max_flow += minimal;
133
134         }
135         return max_flow;
136 }
137
138 void Graph::Add_Edge(int from, int to, int capacity){
139         // 一開始流量為 0
140         edges[from-1].push_back(new edge{from-1, to-1, 0, capacity});
141         edges[to-1].push_back(new edge{to-1, from-1, 0, capacity});
142 }
143
144 // 把輸出的 weight 改成 capacity
145 void Graph::Print_Edges(){
146         for (int i=0 ; i<vertex ; i++){
147         ...
148                 for( ; iter!=edges[i].end() ; iter++){
149                         cout << "->" << (*iter)->to+1 << "," << (*iter)->capacity;
150                 }
151                 cout << endl;
152         }
153
154 }
155
156 int main(){
157
158         Graph(7);
159         g.Add_Edge(1,2,20);
160         g.Add_Edge(1,3,10);
161         g.Add_Edge(1,4,15);
162         g.Add_Edge(2,5,10);
163         g.Add_Edge(3,5,10);
164         g.Add_Edge(3,4,10);
165         g.Add_Edge(3,6,5);
166         g.Add_Edge(4,7,25);
167         g.Add_Edge(5,7,5);
168         g.Add_Edge(6,7,10);
169
```

```
170         int s=1, t=7;
171         cout << "Max Flow from " << s << "to" << t << ": " <<
172         g.Network_Flow(s,t);
173
174         return 0;
175     }
```

執行結果：

```
Max Flow from 1 to 7: 35
```