Advanced Competitive Programming

國立成功大學ACM-ICPC程式競賽培訓隊
nckuacm@imslab.org

Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan

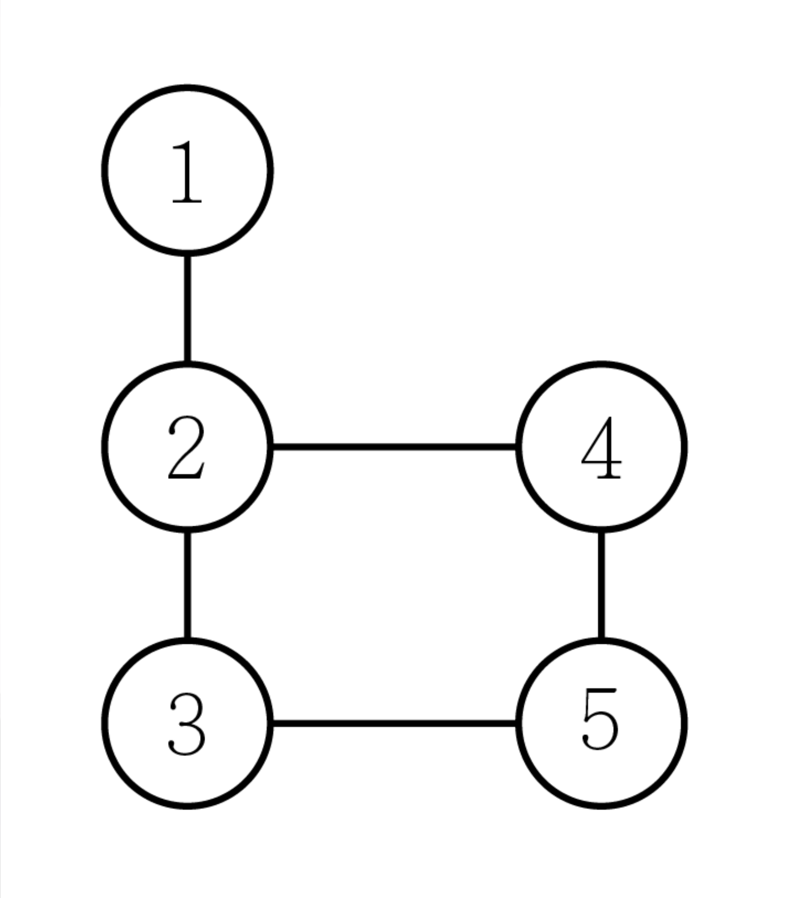
Bipartite Graph

二分圖

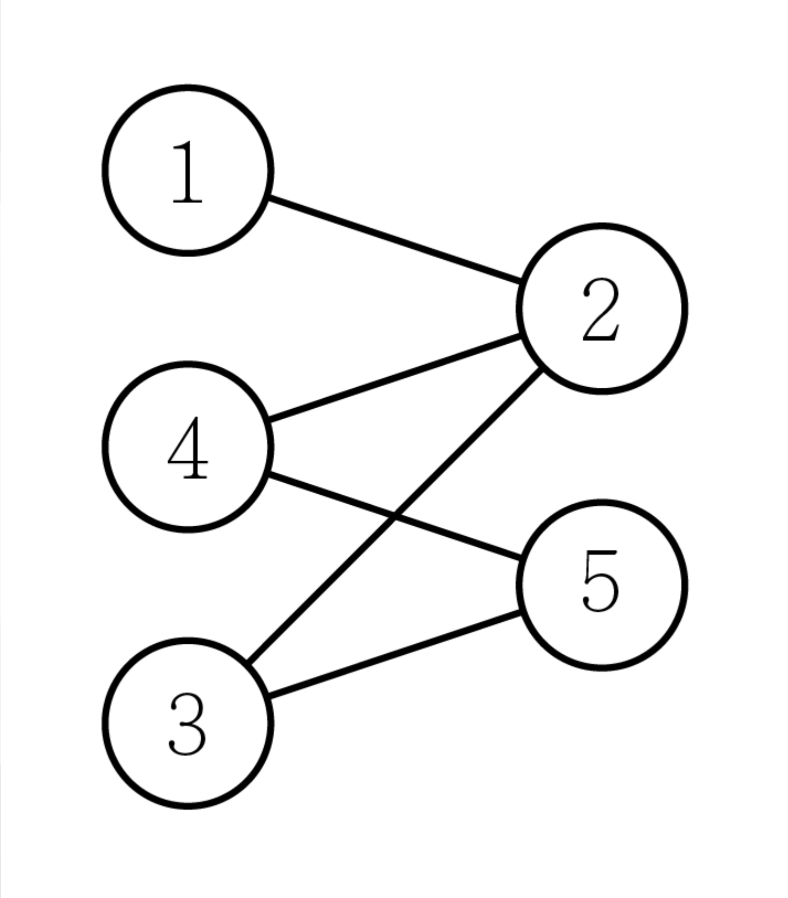
- 對於一張圖 G 中，存在一種方法可以將所有點分割成兩個不相交點集 U 與 V ，且對於所有的邊 $(u, v) \in G$ ， u 與 v 屬於不同點集。

- 換句話說，當所有點被分成兩個點集後，所有的邊都會「跨越點集」。

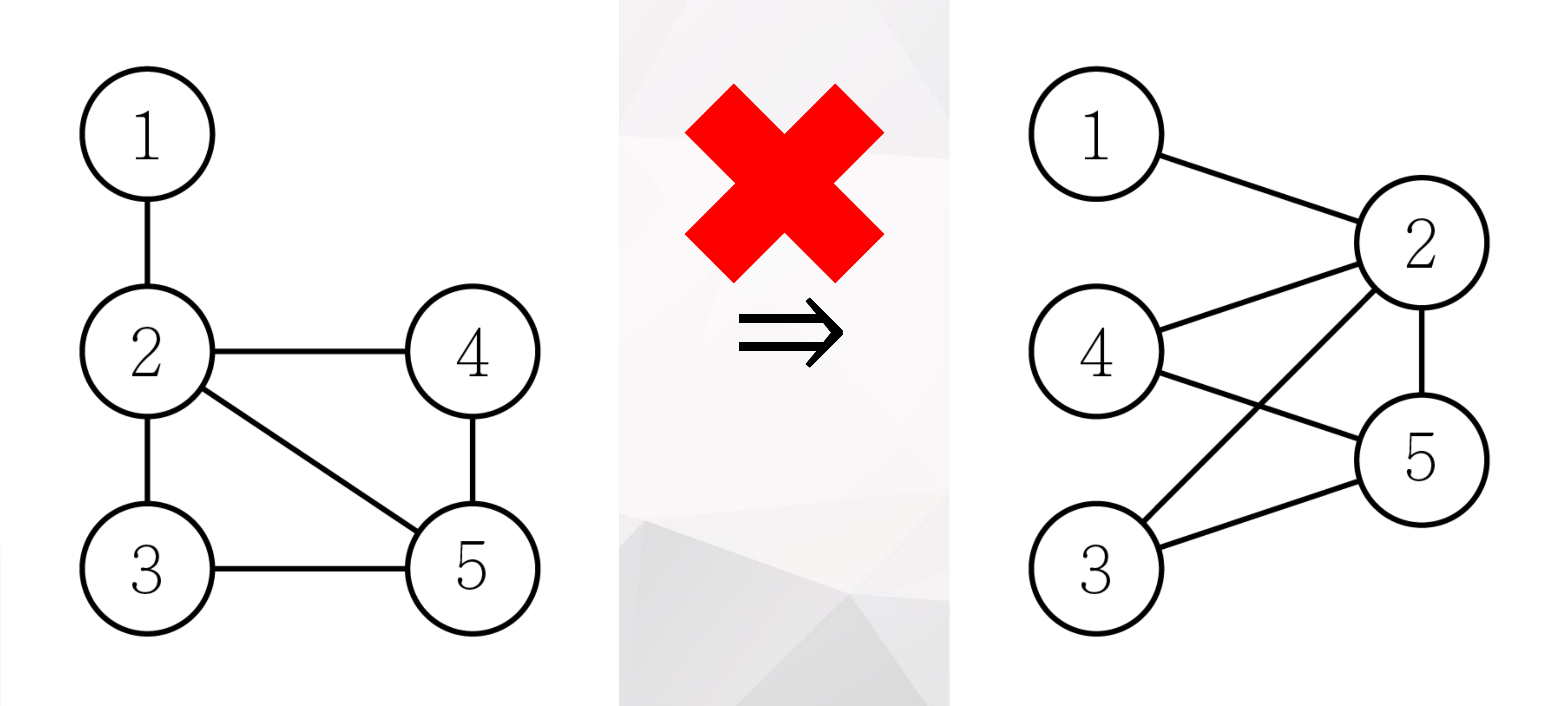
二分圖



$\Rightarrow$



二分圖



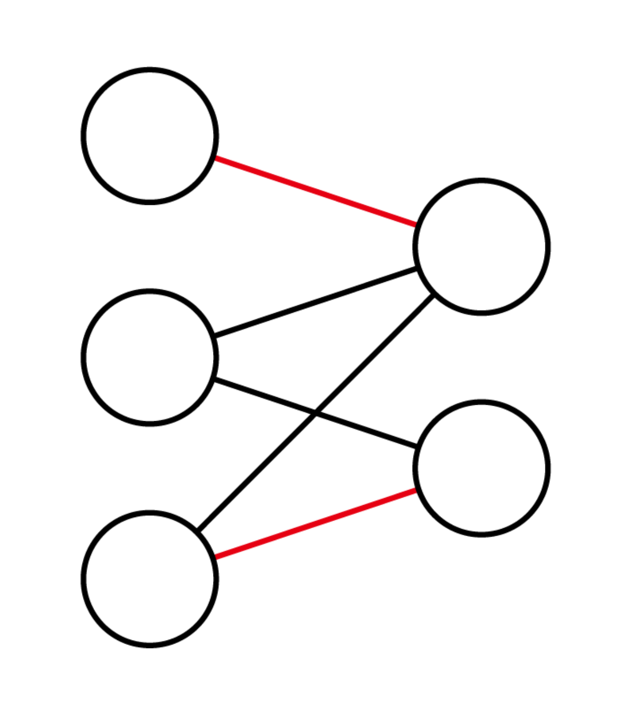
判別二分圖

- 假設一張圖是二分圖，那麼不妨將兩個點集分別著上不同的顏色
- 對於任意一條邊，其兩端點的顏色必不同
- 先將圖上其中一點著色，並對該點 DFS，若其相鄰點還沒被塗色，就將其著上與之不同的顏色，並繼續 DFS；若其相鄰點已經被著上相同的顏色，則表示這張圖不是二分圖
- 若圖不連通，須對圖中所有連通塊 DFS

Matching

匹配

- 對於一張無向圖，找到相異的 $2N$ 個點 $A_1, A_2, \cdots, A_N$ 及 $B_1, B_2, \cdots, B_N$ ，使得對於所有的 i ，都有一條邊 E_i 連接 A_i 與 B_i ，這個匹配的權重就是 E_i 的權重和

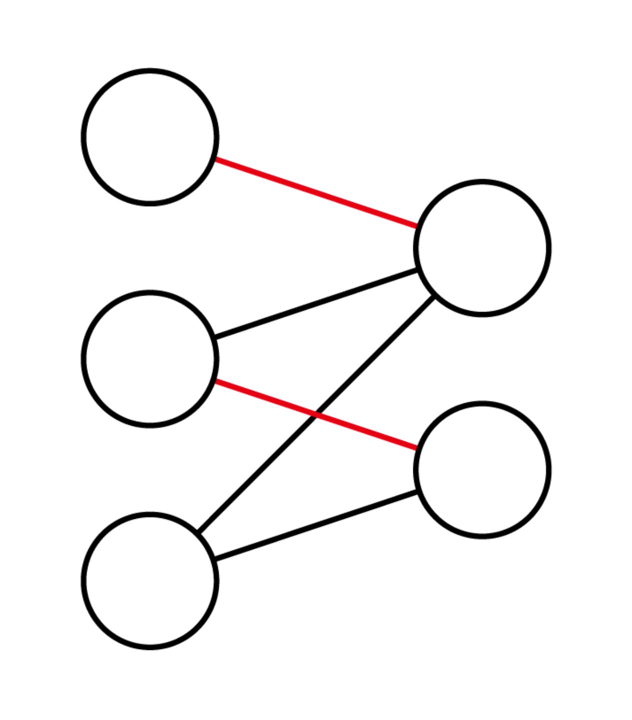


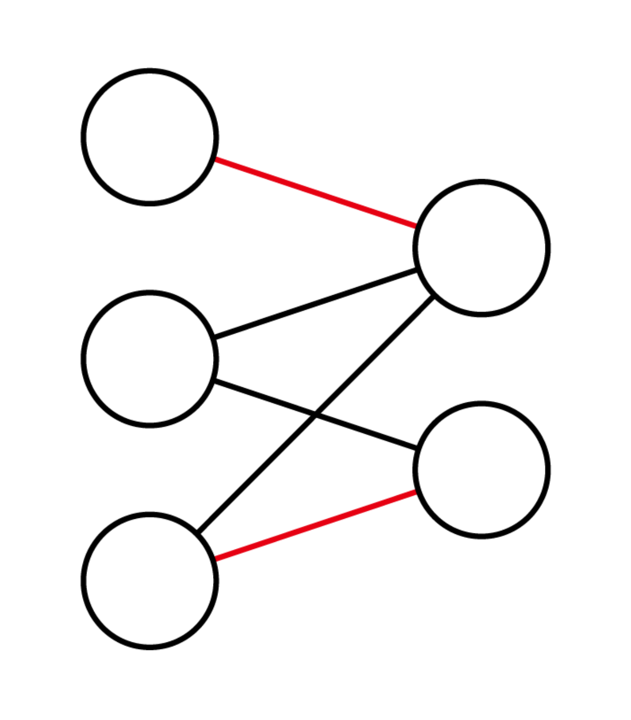
- 匹配有分為一般圖匹配與二分圖匹配

- 一般圖匹配較困難，故不在這次課程內容

二分圖最大匹配

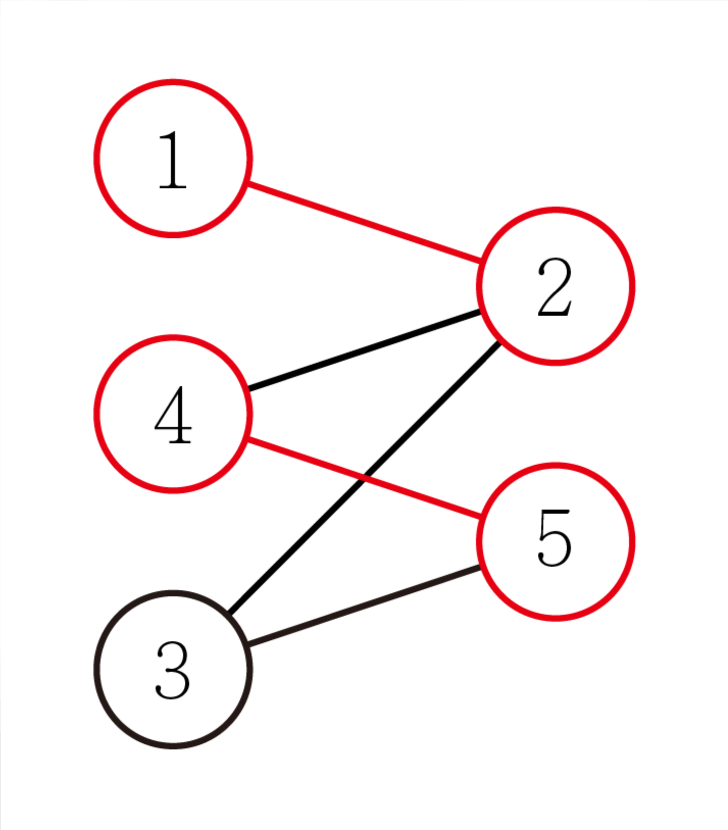
- 這次只會講二分圖最大匹配，也就是在二分圖上所有邊的權重都是 1 時的最大權匹配，如右圖同時也是這張圖的最大匹配
- 最大匹配並不唯一，右圖也是這張圖其中一個最大匹配





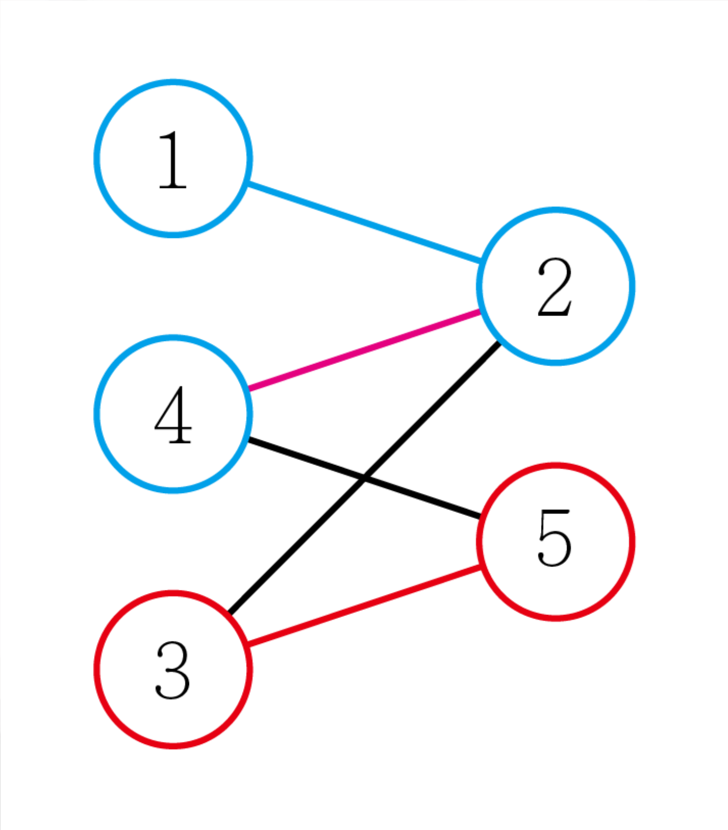
二分圖最大匹配

- 在一個圖當中，邊可以被分類為「匹配邊」與「未匹配邊」點可以分成「匹配點」與「未匹配點」
- 右圖中紅色的邊就是匹配邊，紅色的點是匹配點，反之則為未匹配邊與未匹配點



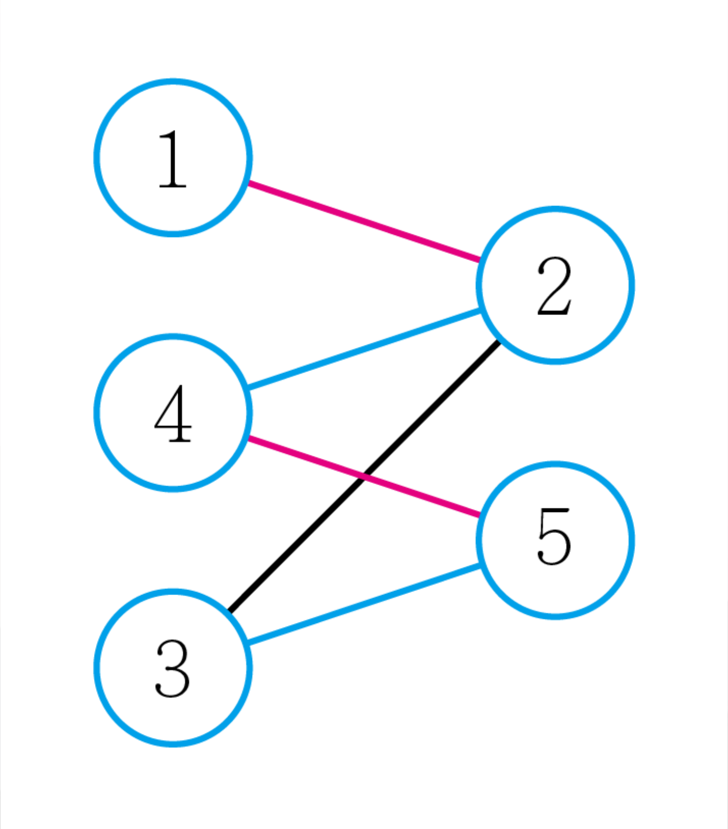
二分圖最大匹配

- 往後的圖示中
  藍色：選取的路徑、點
  粉色：選取路徑上的匹配邊
  紅色：不在選取路徑上的匹配邊、點
  黑色：不在選取路徑上的未匹配邊、點



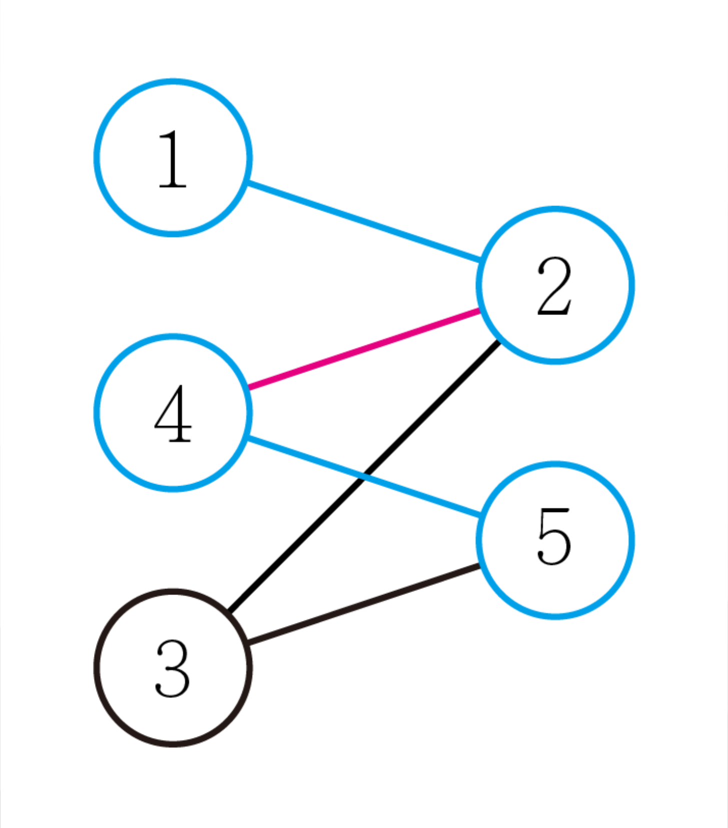
二分圖最大匹配

- 當一個非環簡單路徑(沒有經過重複頂點)所經過的邊是匹配邊與非匹配邊交替出現，則這個路經稱為「交錯路徑」



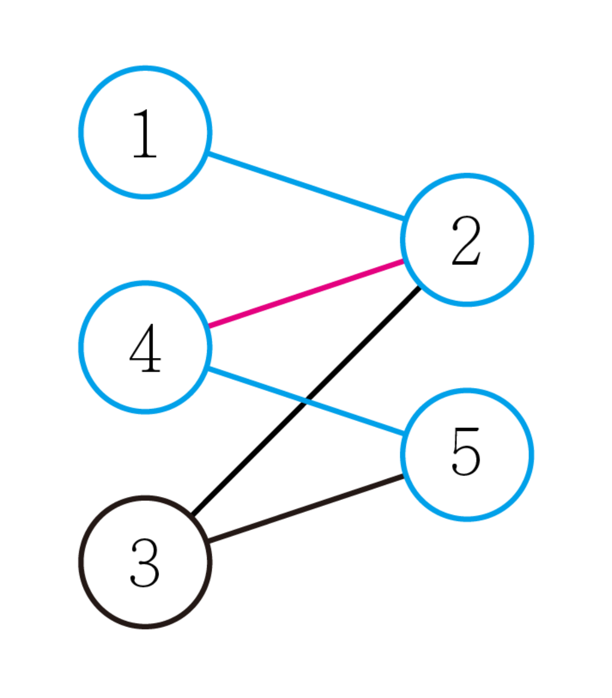
二分圖最大匹配

- 若交錯路徑的起點與終點均為未匹配點，且路徑上與之相鄰的邊均為未匹配邊，則這條路經稱為「擴充路徑」

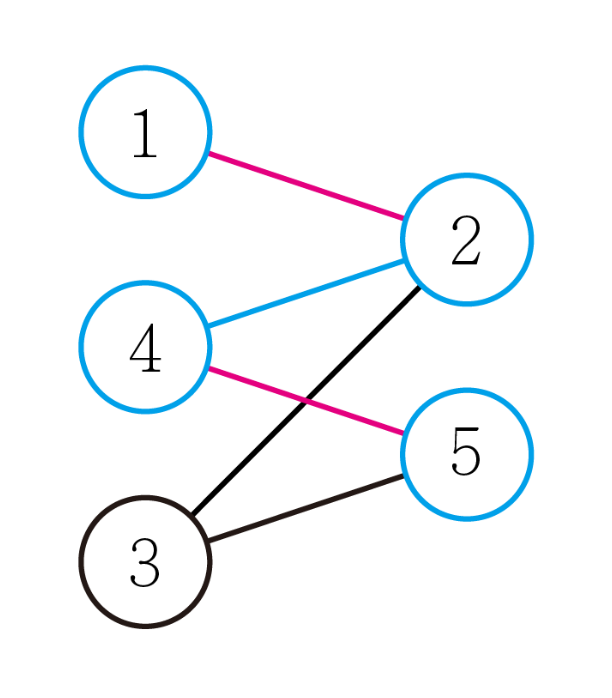


二分圖最大匹配

- 如果將擴充路徑中所有的匹配邊與非匹配邊反轉，則匹配數會 +1 ，這邊所說的反轉是指匹配邊變成未匹配邊，未匹配邊變成匹配邊



$\Rightarrow$



Berge’s lemma

- 一個匹配是最大匹配 $\Leftrightarrow$ 圖上找不到任何擴充路徑
- 假設存在任何擴充路徑，那麼就能將其匹配邊與未匹配邊反轉，並將匹配數 $+1$
- 此定理可以延伸得到：若從一個未匹配點 v 出發找不到任何的擴充路徑，則一定存在一種不包含點 v 的最大匹配

二分圖最大匹配

- 有了以上想法，便可以直接構造出演算法了

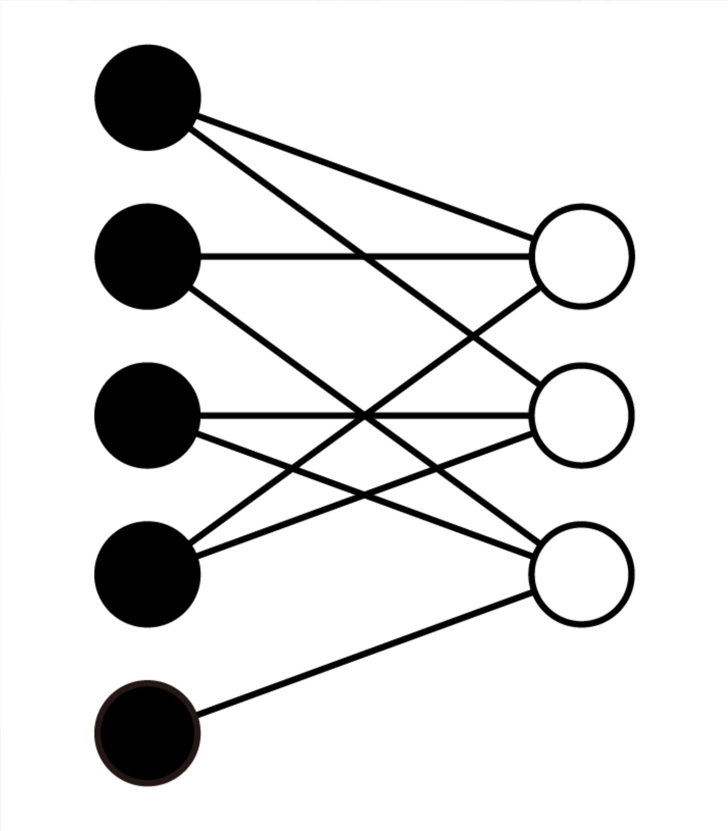
- 枚舉二分圖某一側的所有點，若從該點出發找不到任何的擴充路徑，則將其移出匹配，否則將擴充路徑上所有的邊反轉，匹配數 +1

如何找擴充路徑？

- 利用二分圖的性質
- 若將一個二分圖塗成黑白兩色，只要從二分圖的未匹配黑點開始 **DFS**，找到相鄰白點後檢查其是否為未匹配點，
- 若是未匹配點，則找到擴充路徑，匹配數 +1
- 否則從該白點的匹配黑點開始 **DFS**，繼續尋找擴充路徑，若找到擴充路徑則將其反轉，匹配數 +1 ，找不到匹配點則回傳 0 ，此時起點可以繼續選擇下一個白點
- 如果找不到任何擴充路徑，將該黑點移出可能匹配的點

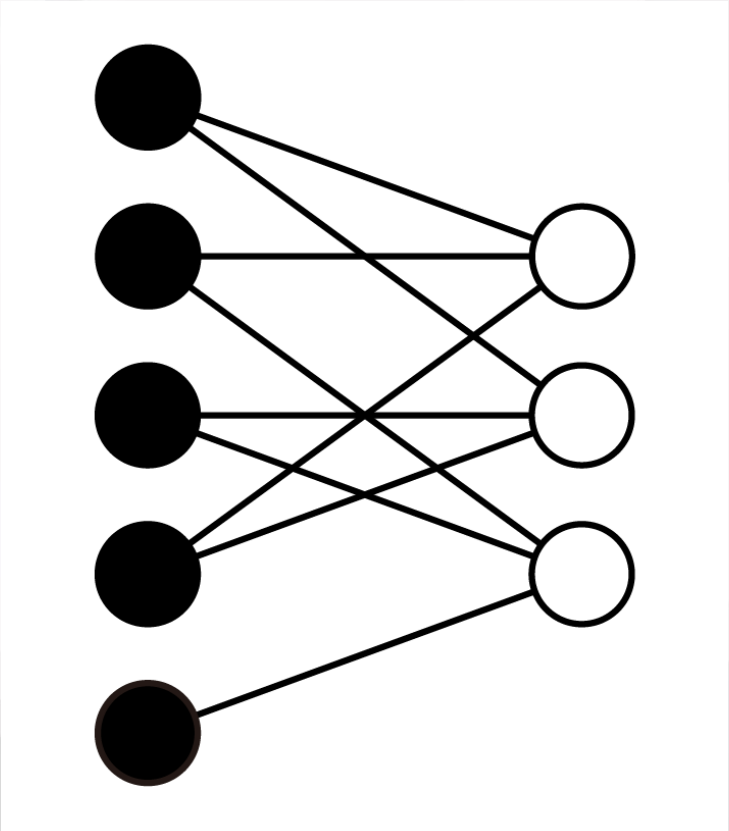
二分圖最大匹配

- 考慮這張二分圖，假設左方為黑點



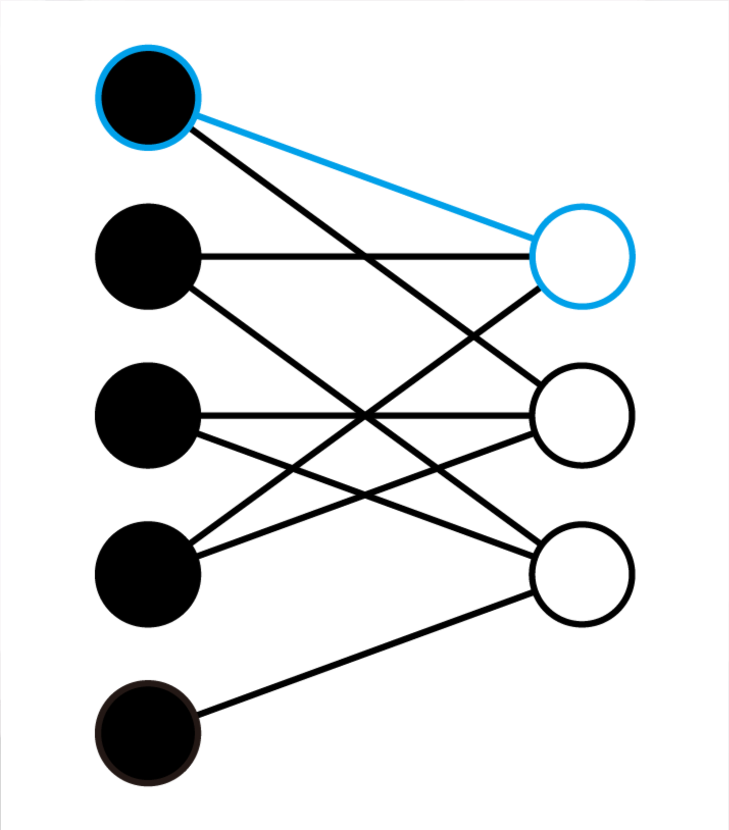
二分圖最大匹配

- 先從第一個黑點開始找擴充路徑



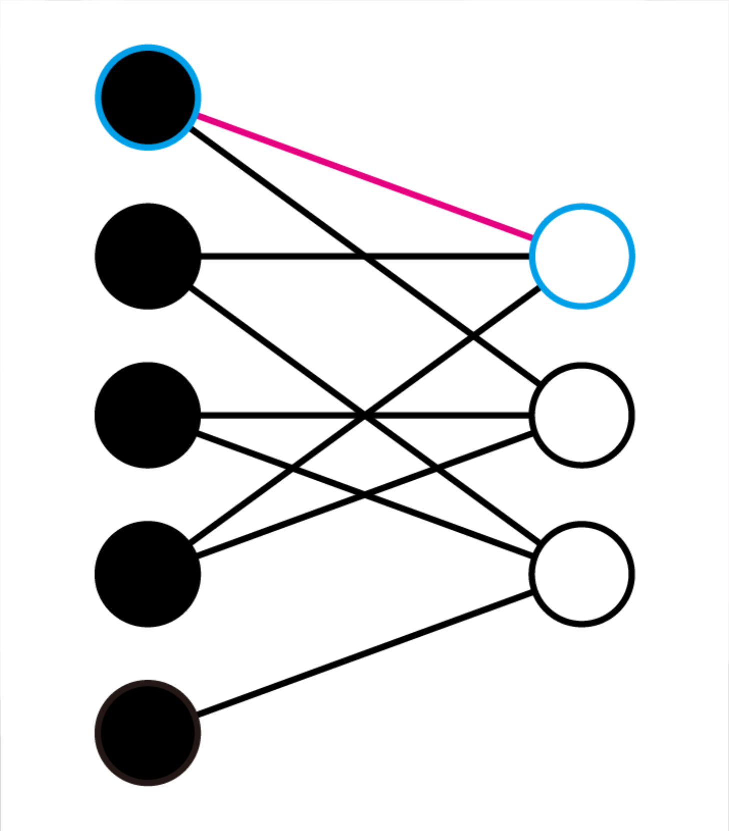
二分圖最大匹配

- 找到了擴充路徑



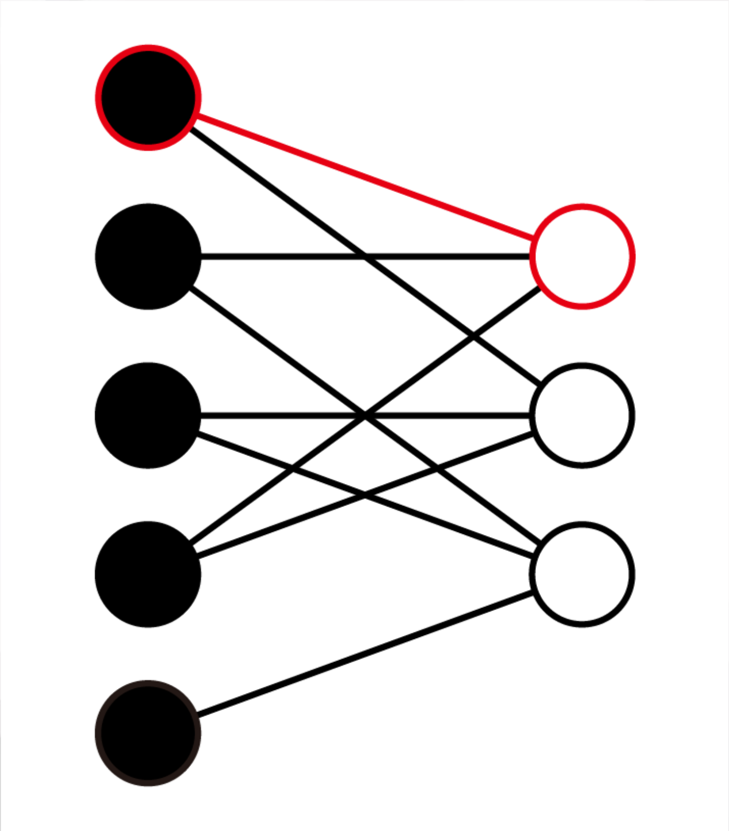
二分圖最大匹配

- 將其匹配邊與未匹配邊反轉，匹配數 +1



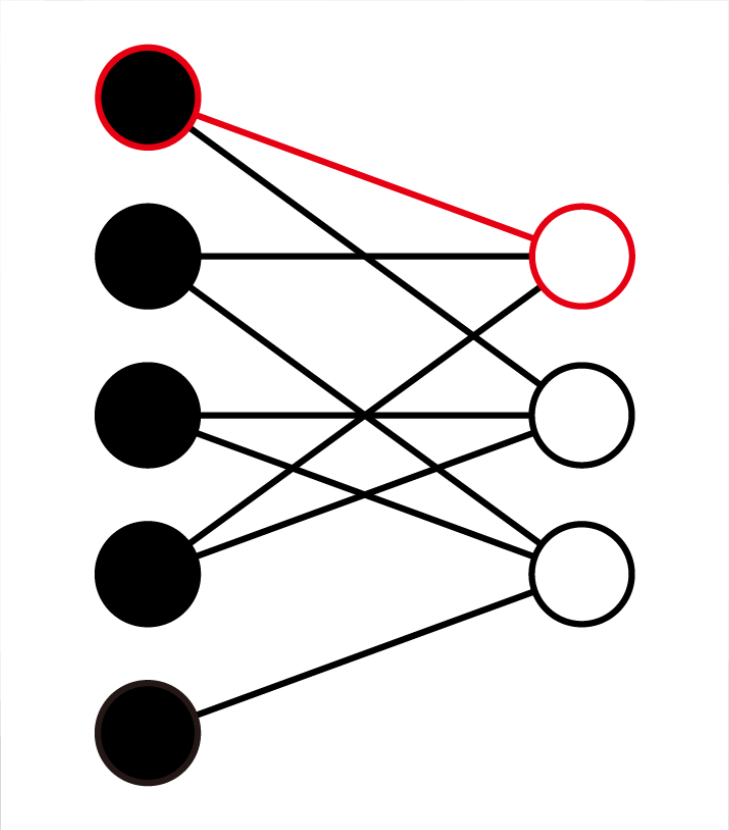
二分圖最大匹配

- 將其匹配邊與未匹配邊反轉，匹配數 +1



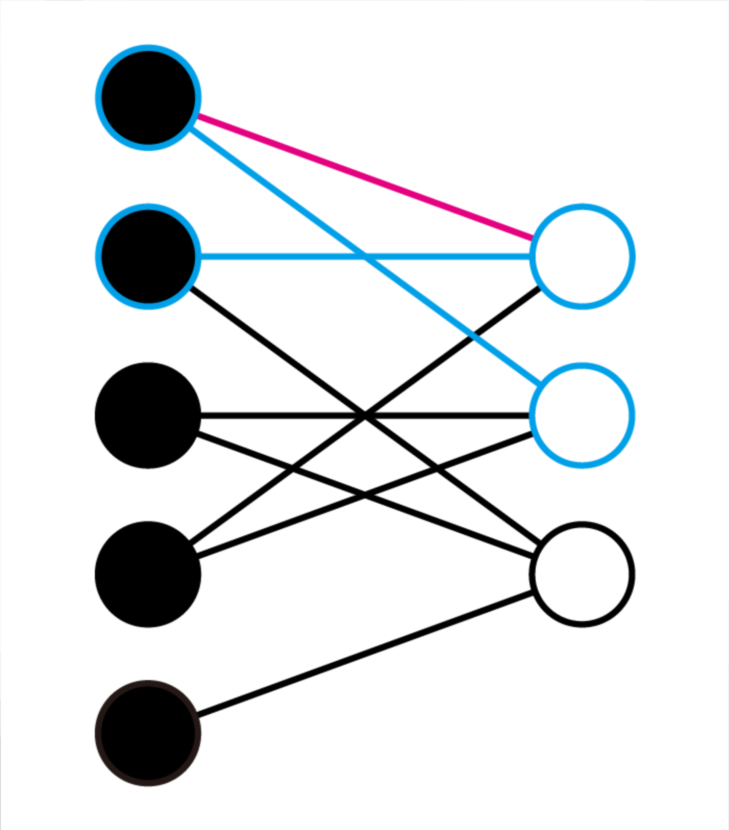
二分圖最大匹配

- 接著從第二個點開始找擴充路徑



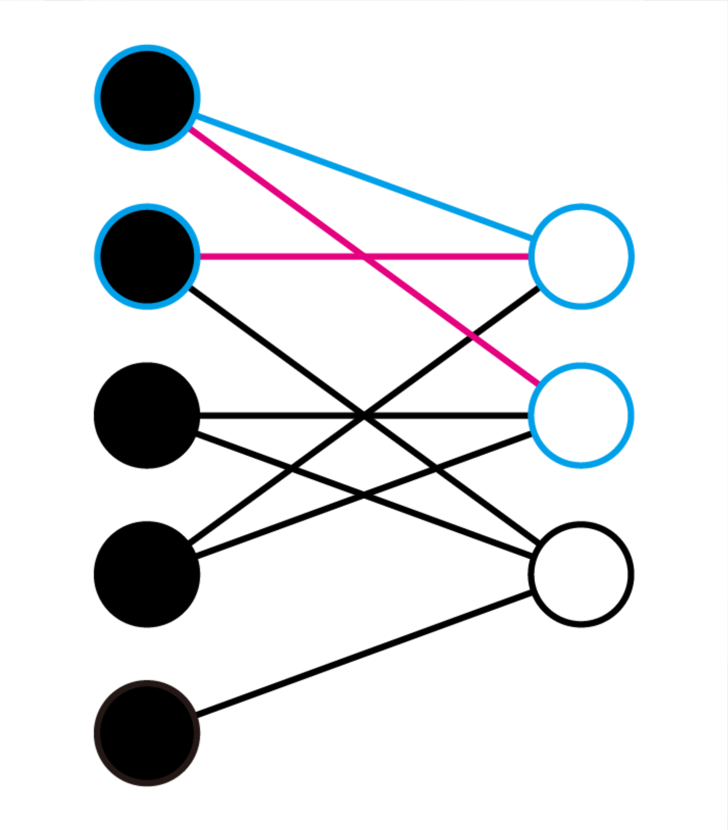
二分圖最大匹配

- 找到了擴充路徑



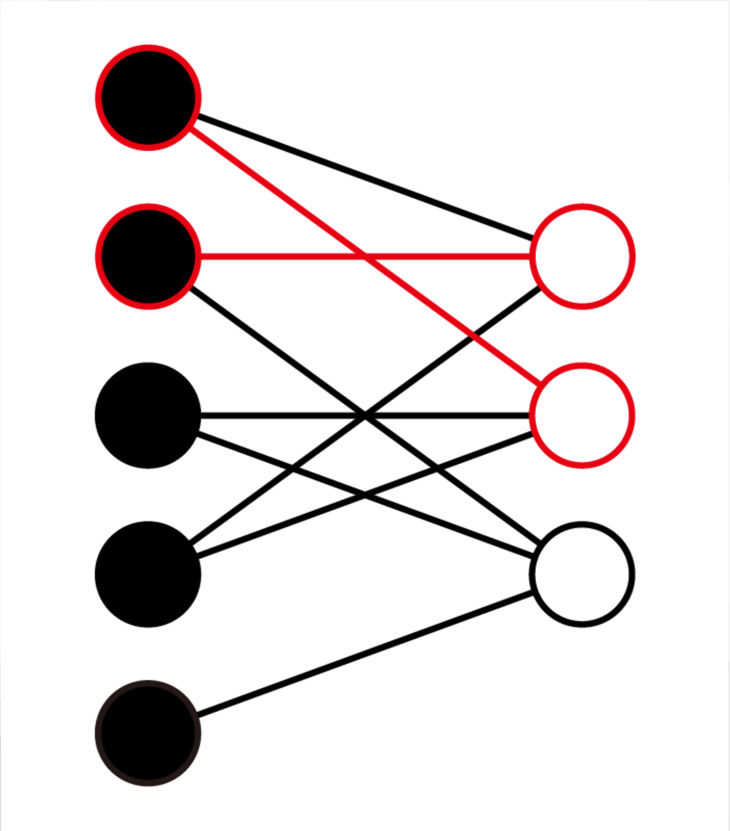
二分圖最大匹配

- 將其匹配邊與未匹配邊反轉，匹配數 +1



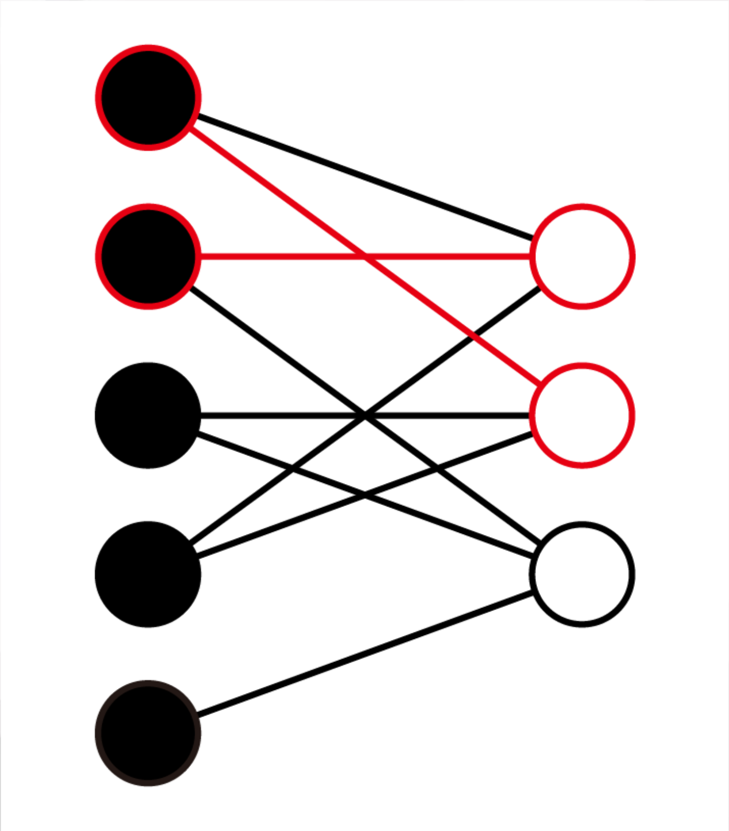
二分圖最大匹配

- 將其匹配邊與未匹配邊反轉，匹配數 +1



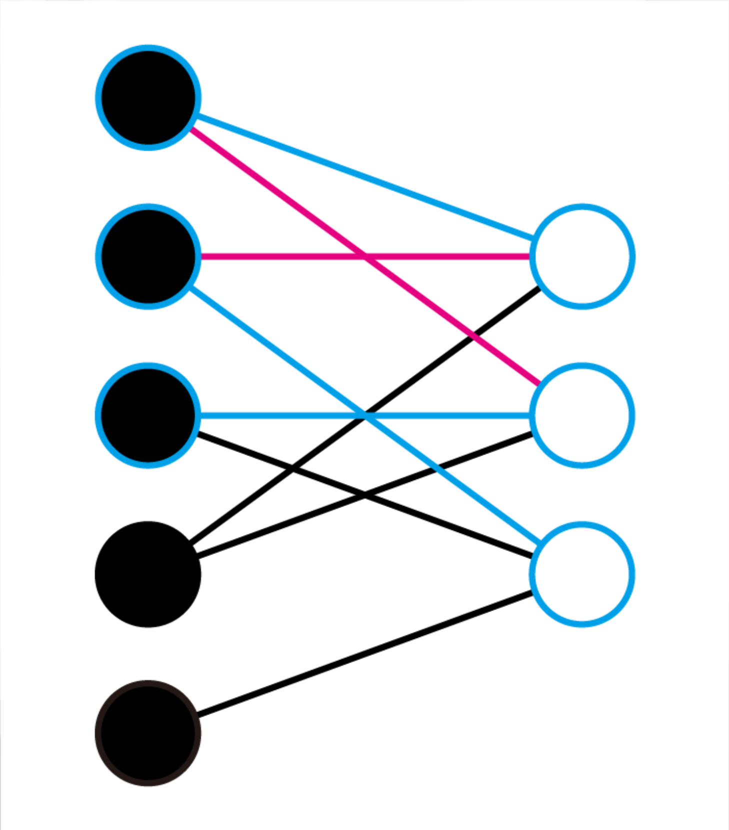
二分圖最大匹配

- 接著從第三個點開始找擴充路徑



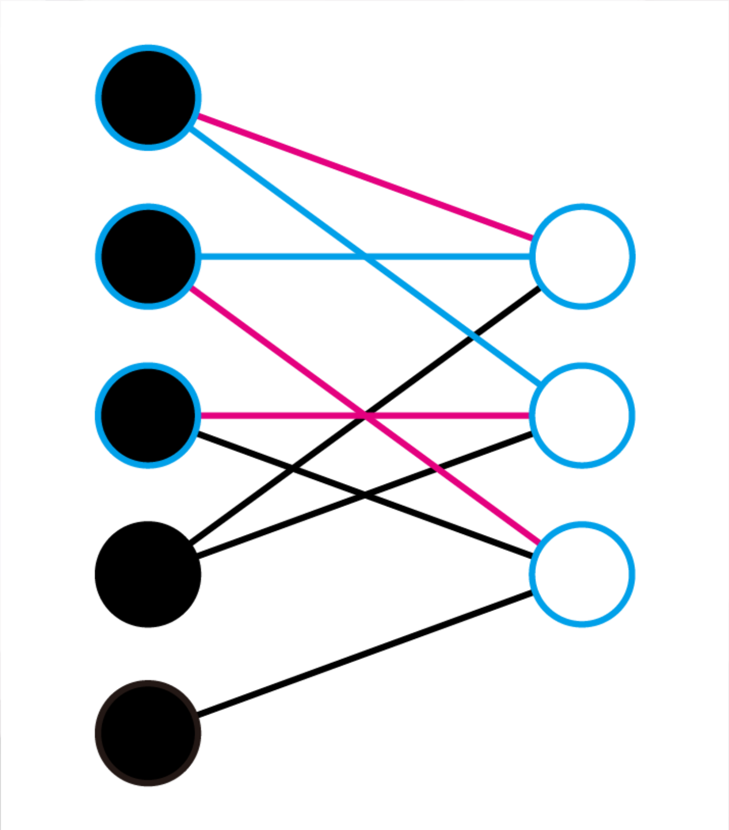
二分圖最大匹配

- 找到了擴充路徑



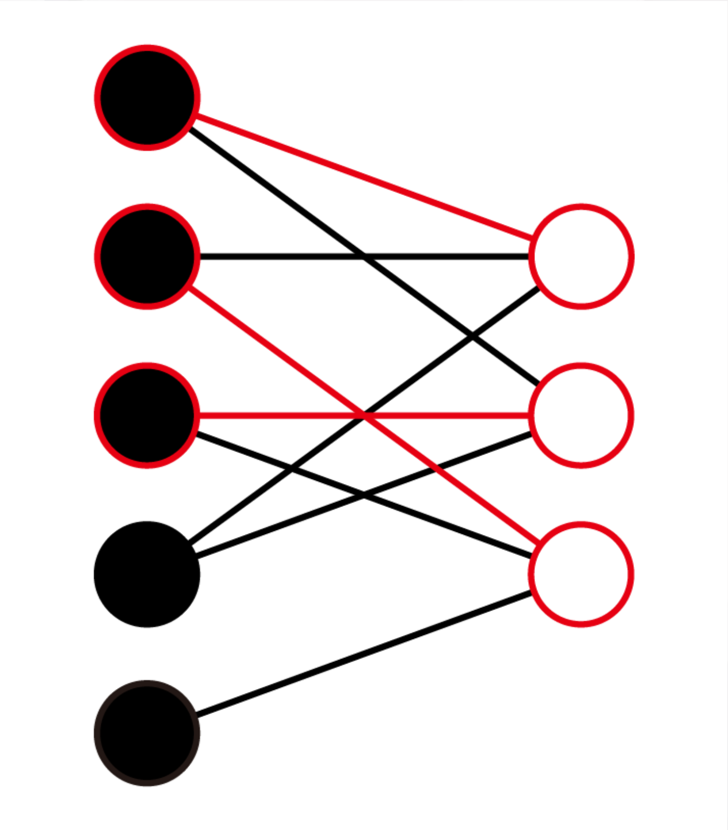
二分圖最大匹配

- 將其匹配邊與未匹配邊反轉，匹配數 +1



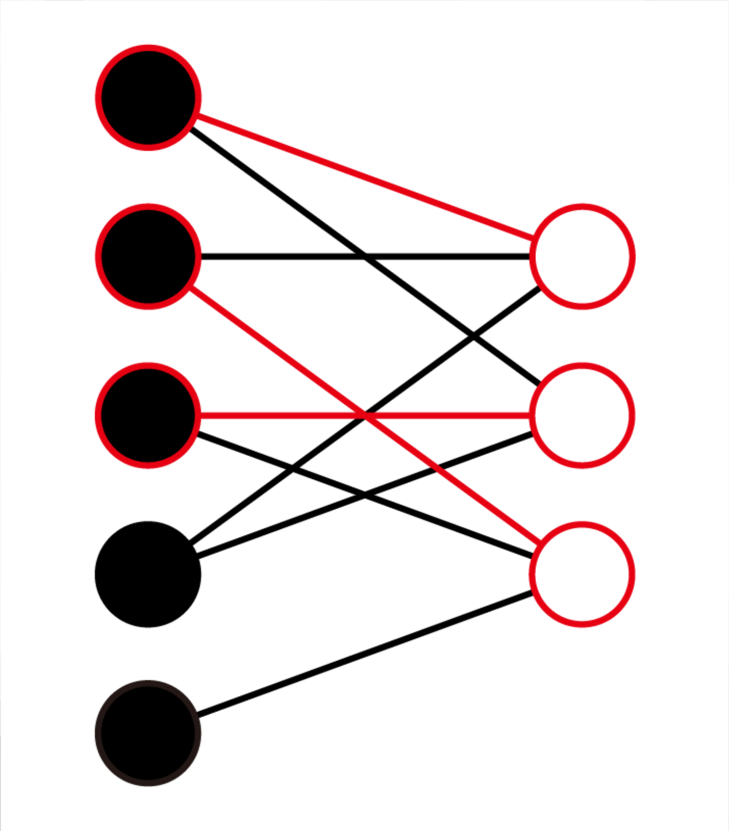
二分圖最大匹配

- 將其匹配邊與未匹配邊反轉，匹配數 +1



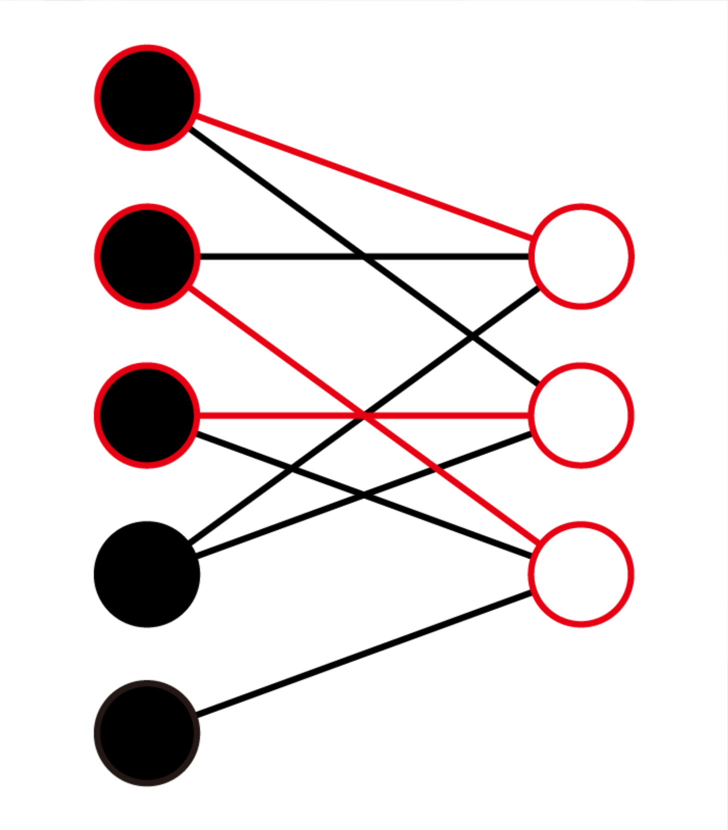
二分圖最大匹配

- 接著從第四個點開始找擴充路徑



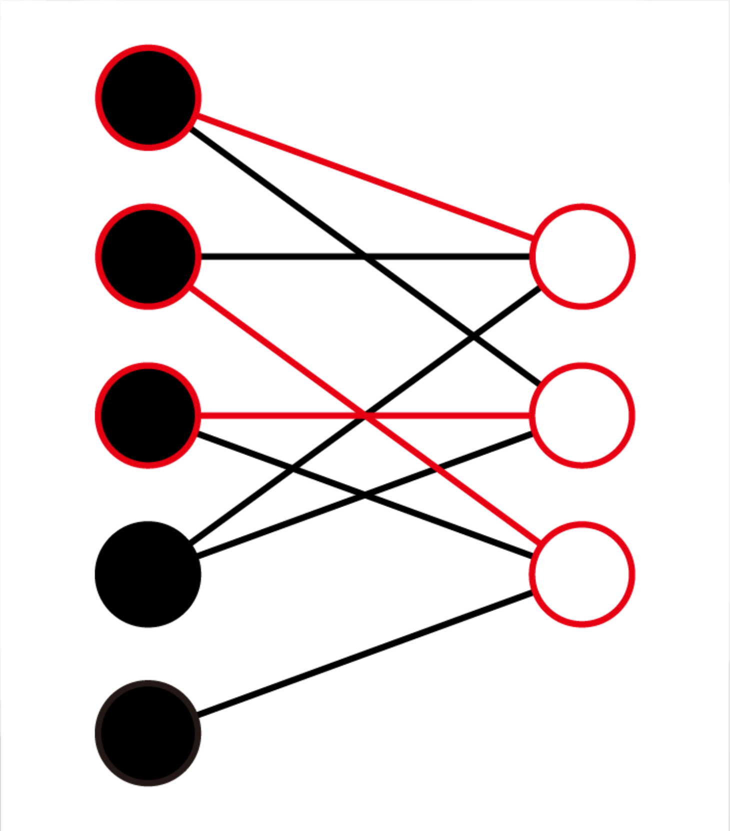
二分圖最大匹配

- 發現找不到擴充路徑
- 因此第四個點不可能是匹配點



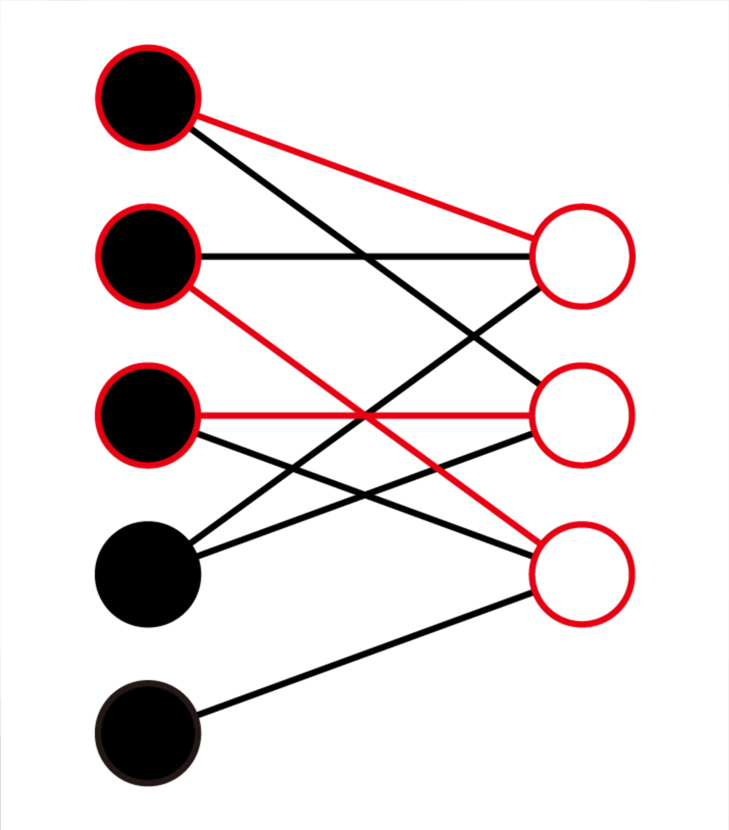
二分圖最大匹配

- 接著從第五個點開始找擴充路徑



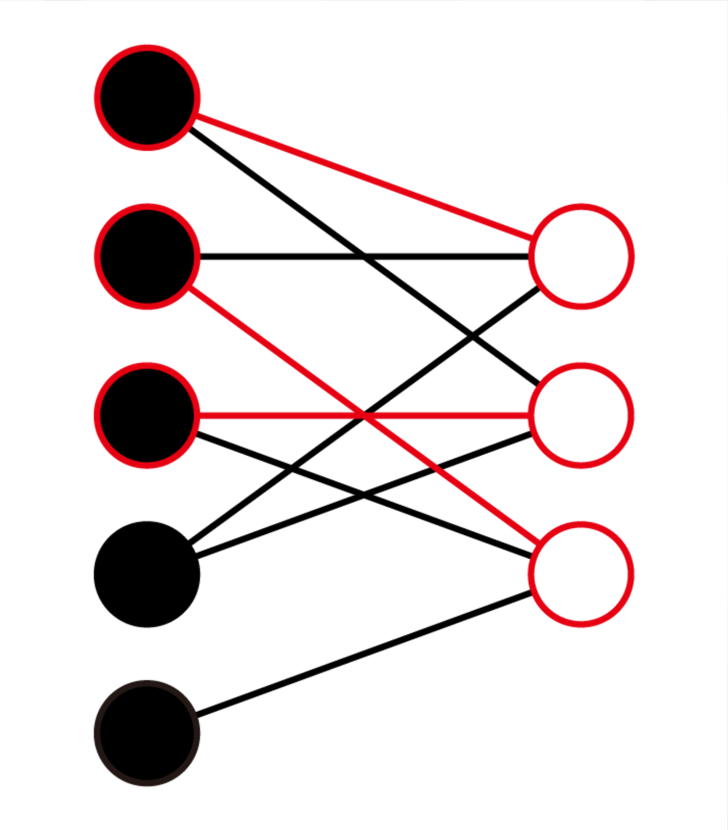
二分圖最大匹配

- 發現找不到擴充路徑
- 因此第五個點不可能是匹配點



二分圖最大匹配

- 所有的點都枚舉完了，演算法結束，此時的匹配數即為最大匹配



時間複雜度

- 每一次尋找擴充路徑，花費 $O(M)$ 的時間
- 至多枚舉 N 個黑點，每次枚舉完只會有增加匹配與移除點兩種情況，因此每個點至多 DFS 一次
- 總複雜度 $O(NM)$
- 如果一開始在圖上先胡亂匹配後，再執行此演算法，則可能讓整個演算法的執行時間縮短

Code

- 這邊做了一些空間上的優化，因為白點需要尋找的點只有當前匹配的黑點，因此只需要為黑點存邊就好
- https://paste.ofcode.org/cgtJGTtKiJtAkqUTjSYLAj

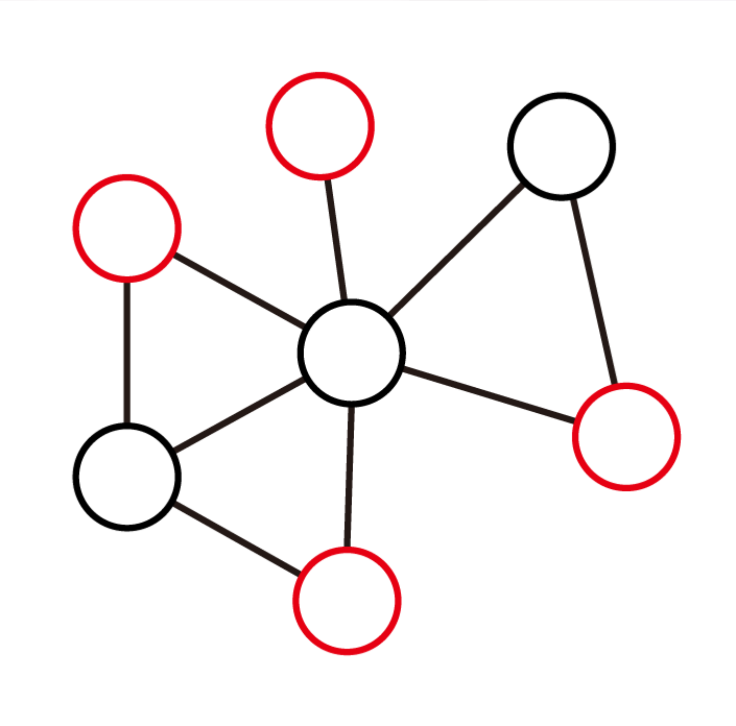
Independent Set and Cover Set

名詞介紹

- 以下所講的所有獨立集與覆蓋都不唯一
- 圖中所舉的例子是只是其中一種

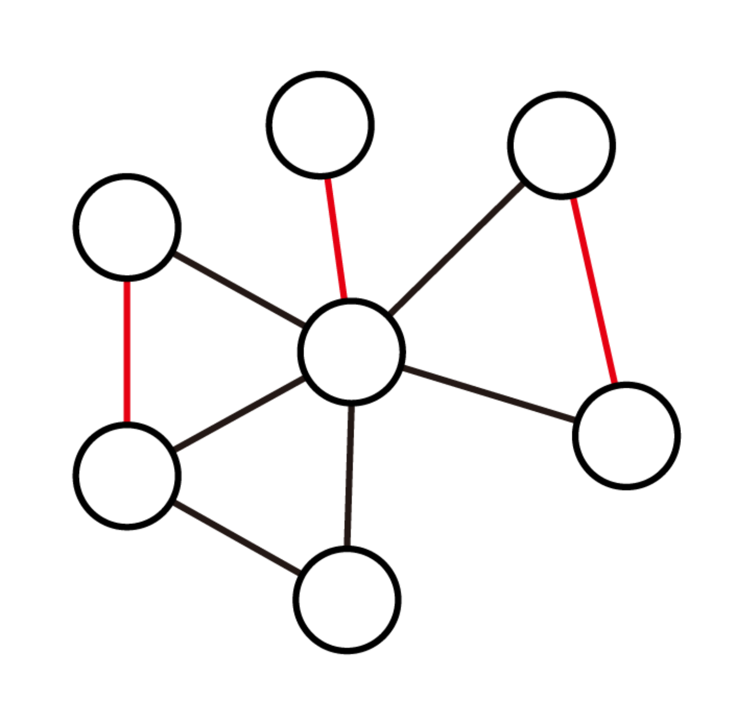
名詞介紹

- 最大點獨立集(Maximum independent set)：
  圖中最大的點集，使得點集中任兩點沒有直接的邊連接，
  在此簡稱此點集為 I
- 如右圖紅色的點集就是這張圖的最大點獨立集



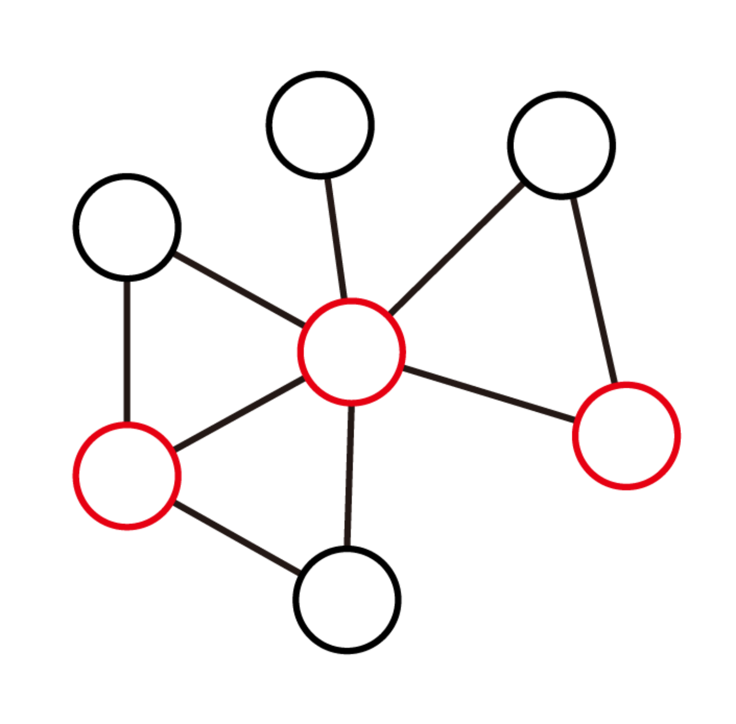
名詞介紹

- 最大邊獨立集(Maximum independent edge set)：圖中最大的邊集，使得邊集中任兩條邊沒有共同端點。且此獨立集等價於最大匹配，在此簡稱此邊集為 M
- 如右圖紅色的邊集就是這張圖的最大邊獨立集



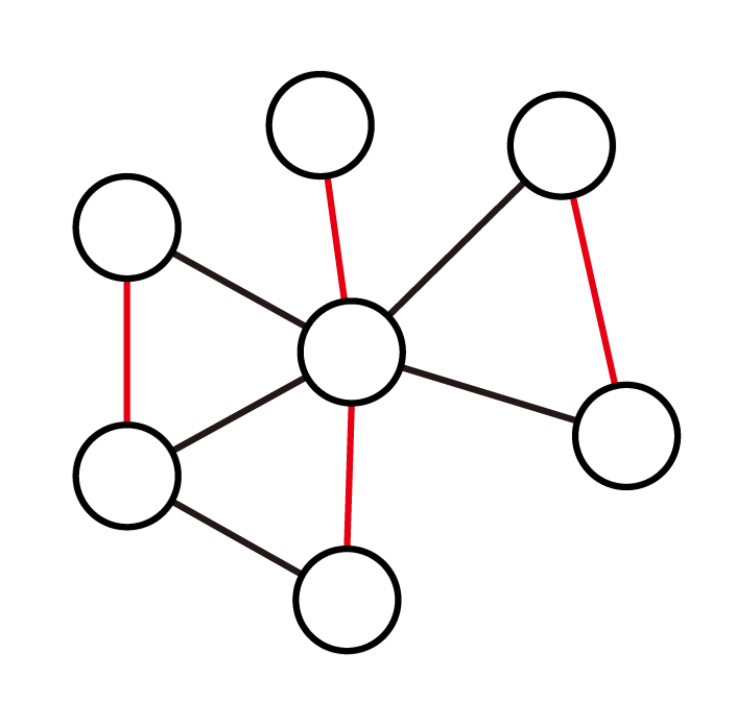
名詞介紹

- 最小點覆蓋(Minimum vertex cover)：
  圖中最小的點集，使得對於圖中任一條邊，至少有一端點存在此點集中，在此簡稱此點集為 C_v
- 如右圖紅色的點集就是這張圖的最小點覆蓋



名詞介紹

- 最小邊覆蓋(Minimum edge cover)：
  圖中最小的邊集，使得對於圖中任一點，至少有一條與該點相鄰的邊存在此邊集中，在此簡稱此邊集為 C_e
- 如右圖紅色的邊集就是這張圖的最小邊覆蓋



獨立集 & 覆蓋

- 根據以上的定義可以發現，對於一張圖 $G = (V, E)$ 中，任何的點獨立集 I' ，$\mathrm{V} - I'$ 都是一個點覆蓋，因此有以下公式：

$$|I| + |C_v| = |V|$$

獨立集 & 覆蓋

- 對於一個最大匹配，這個匹配已經覆蓋了 $2|M|$ 個點，因此至多只需要再 $|V|-2|M|$ 條邊就能覆蓋所有點，故 $|C_e| \le |V|-|M|$

獨立集 & 覆蓋

- 對於一個最小邊覆蓋，這個邊覆蓋與所有的點會形成一個森林，因此每個連通塊的點數會等於邊數加一
- 且連通塊數量會是 $|V| - |C_e|$ ，故 $|M| \geq |V| - |C_e|$

獨立集 & 覆蓋

- 綜合 $|C_e| \leq |V| - |M|$ 與 $|M| \geq |V| - |C_e|$ 得到：

$$|M| + |C_e| = |V|$$

獨立集 & 覆蓋

- 在一般圖上，最大點獨立集與最小點覆蓋都是 NPC 問題，但是在二分圖上，根據 Kőnig 定理得知 $|M| = |C_v|$ ，也就是以上四種問題都可以對應到最大匹配

- 但是這個定理證明複雜，在此略過。

重點整理

- 在所有圖上：
- $|I| + |C_v| = |V|$
- $|M| + |C_e| = |V|$

- 在二分圖上：
- $|M| = |C_v|$
- $|I| + |M| = |V|$

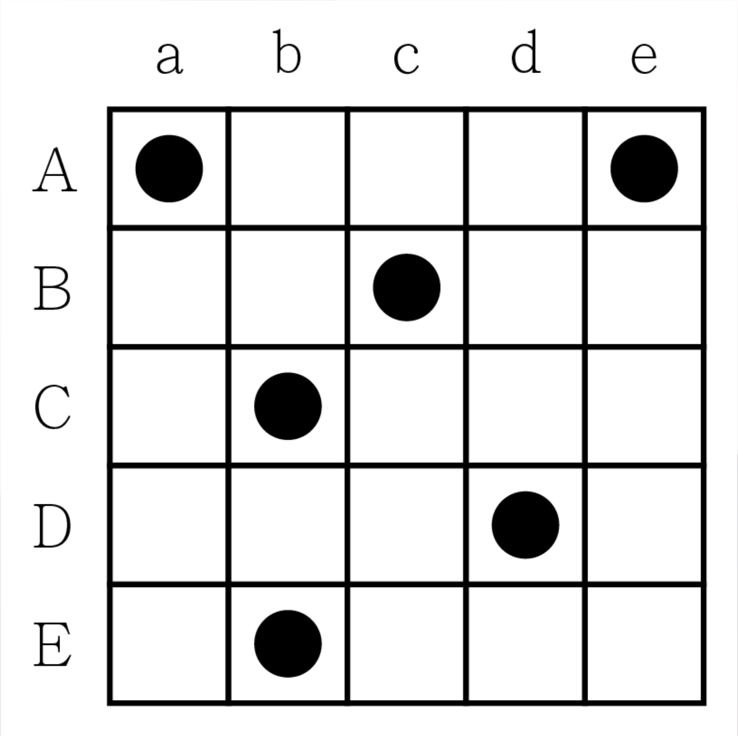
Example Problem

TIOJ 1089 Asteroids

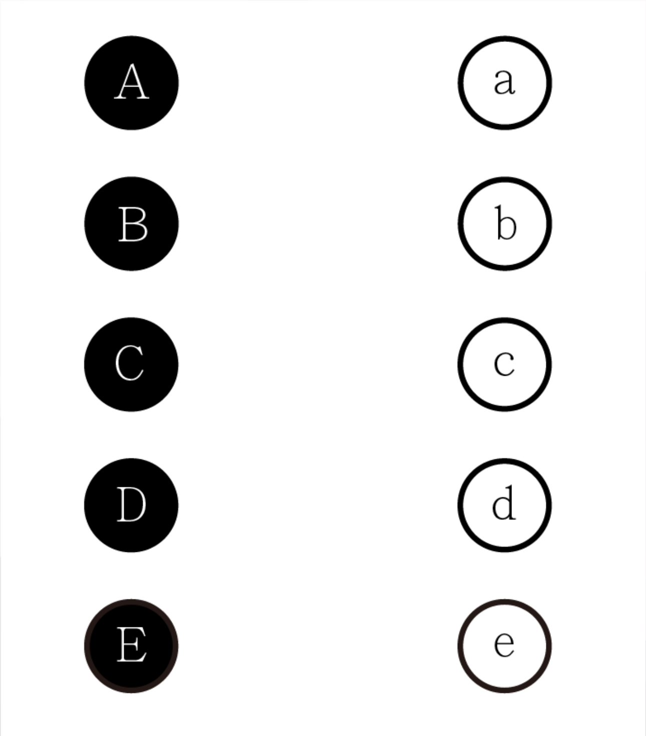
- 有一個 $N \times N$ 的方格紙，在某些格子中有隕石，你現在有一種東西可以一次消滅一整行或一整列的隕石，請問最少要幾次才能將隕石全部消滅

TIOJ 1089 Asteroids

- 將每列編號轉換為二分圖的黑點
- 將每行編號轉換為二分圖的白點

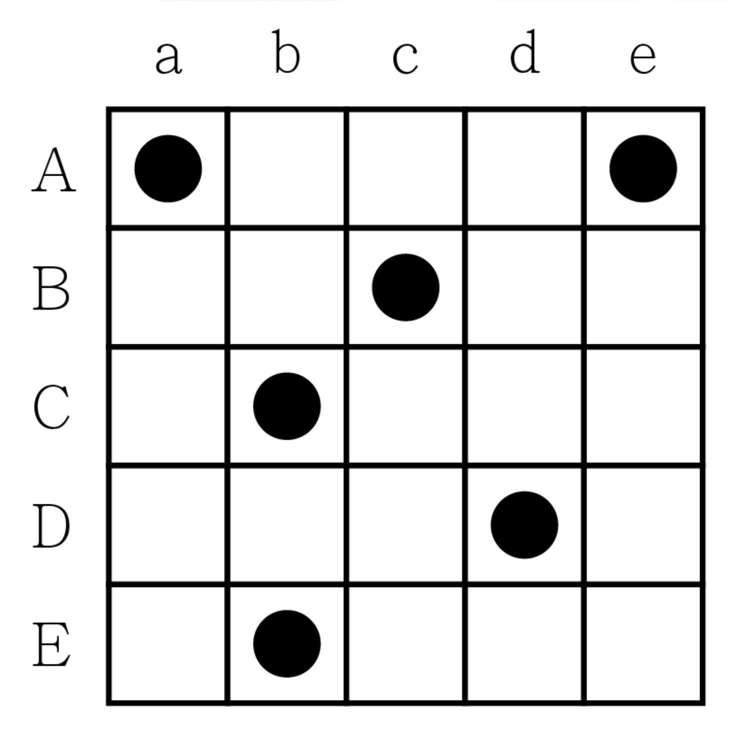


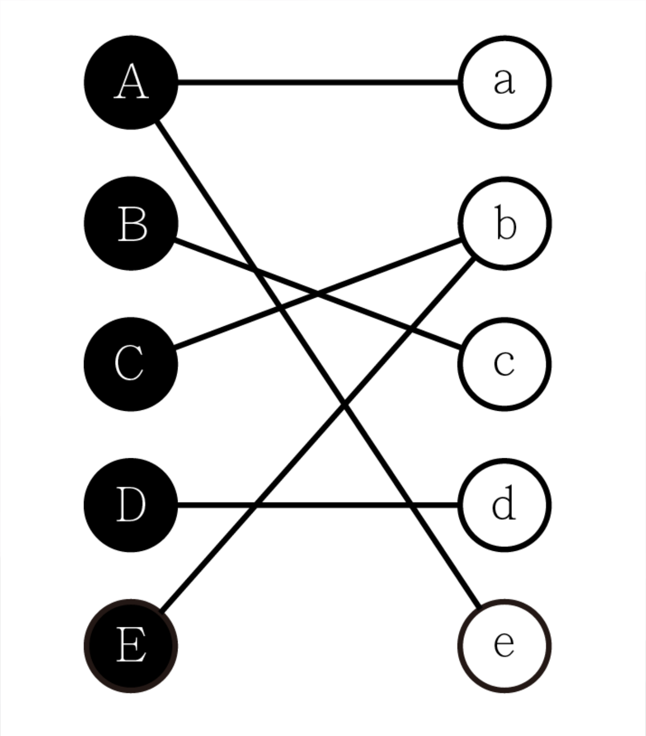
$\Rightarrow$



TIOJ 1089 Asteroids

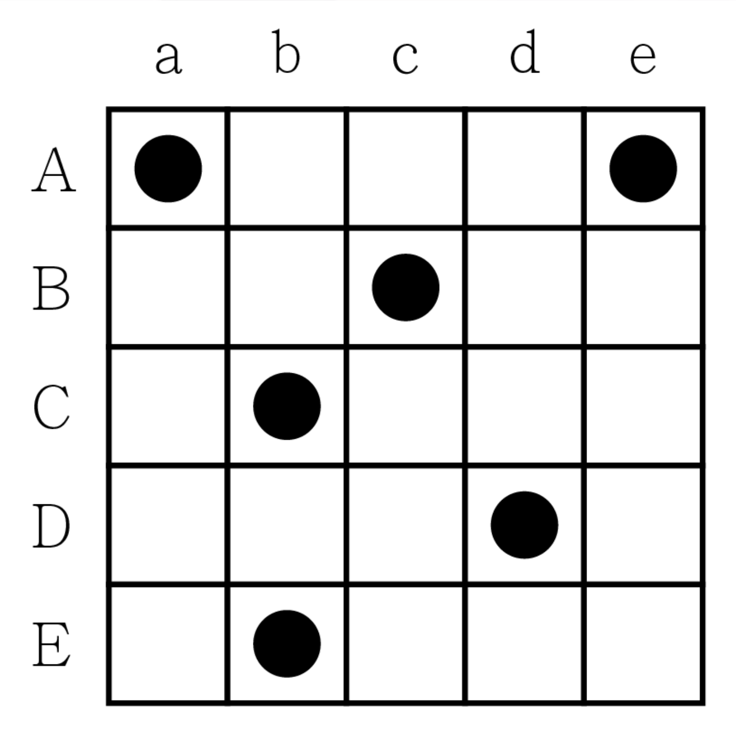
- 有隕石所在的地方，將其所在的行與列連起來



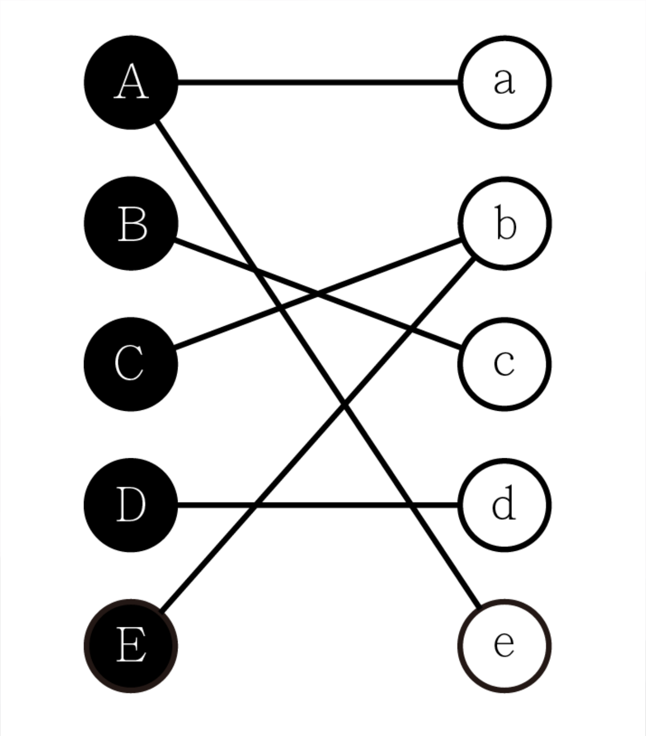


TIOJ 1089 Asteroids

- 接著你會發現當你要射某一行(列)時，這行(列)上所有隕石都會被消除

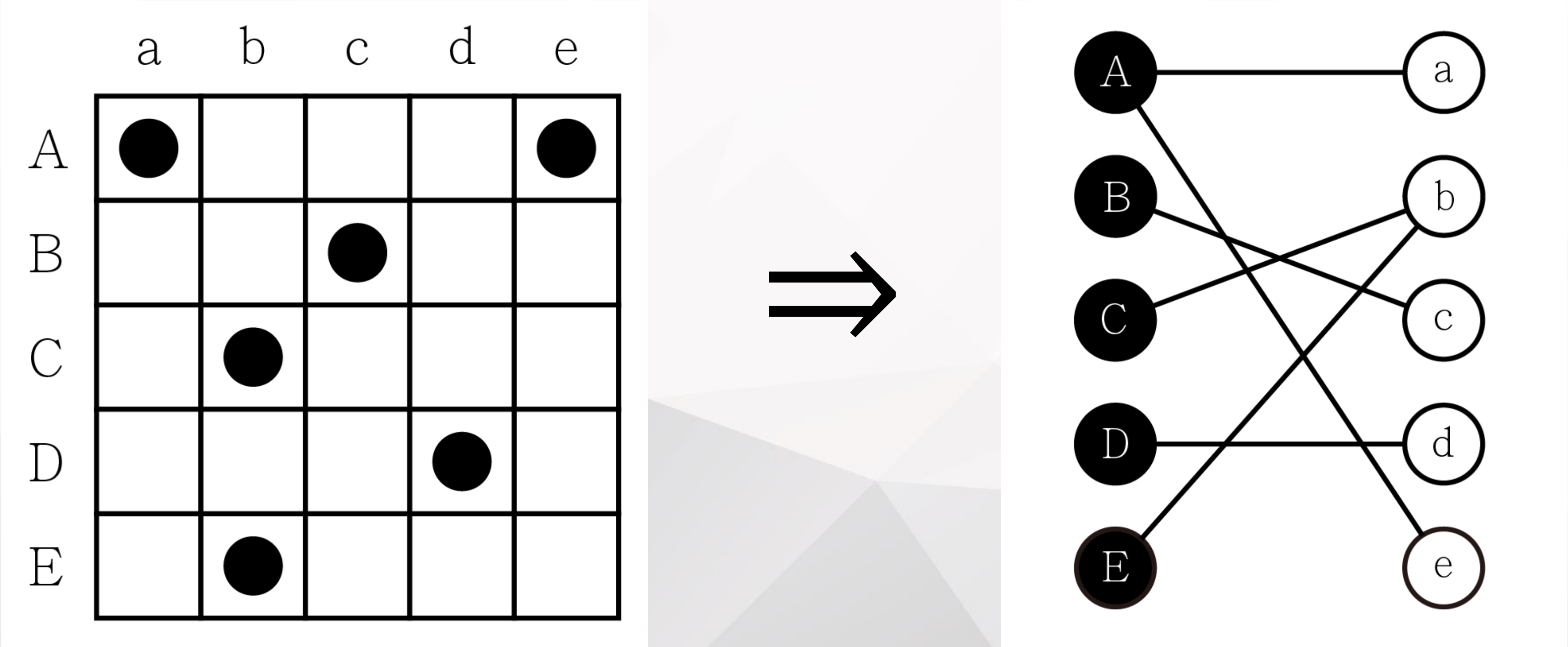


$\Rightarrow$



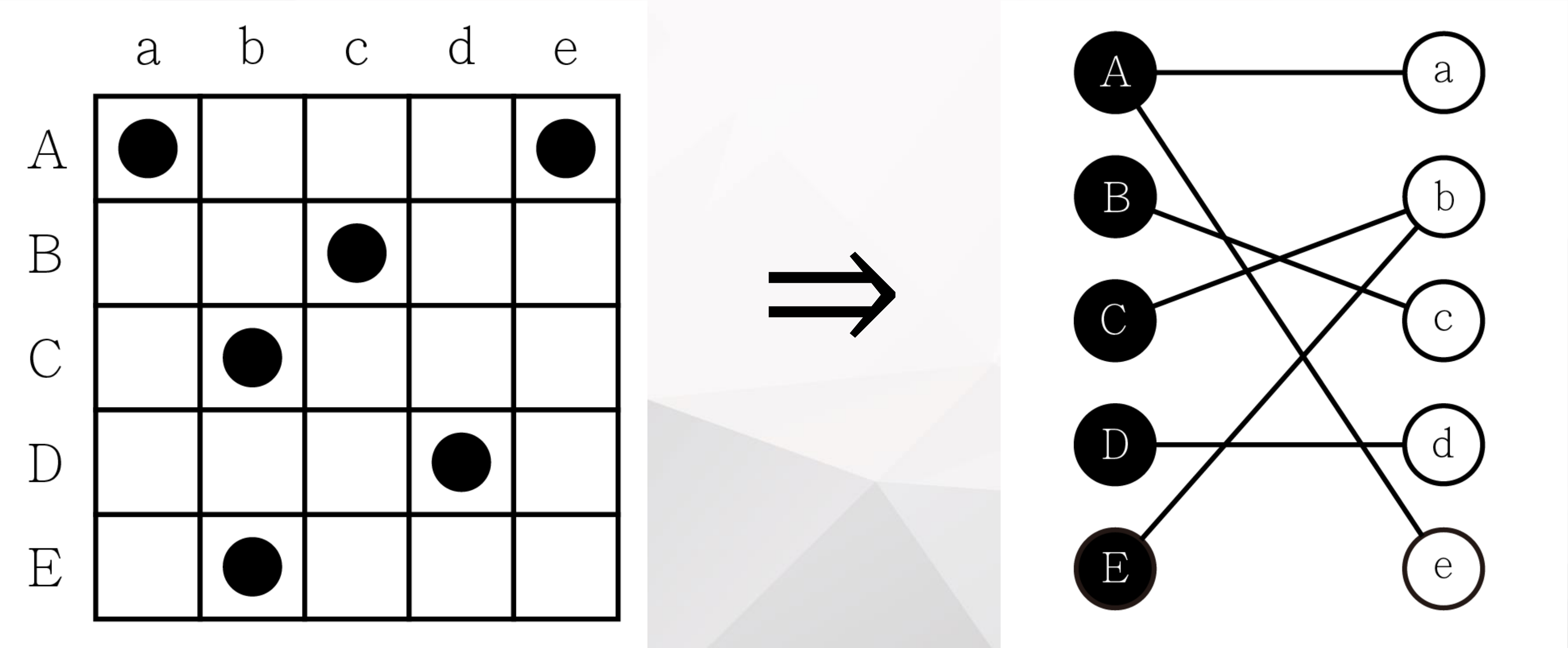
TIOJ 1089 Asteroids

- 也就是選取某個點時，會消去其所有相鄰邊
- 題目要求要選取最少的點，且讓每一條邊都消失



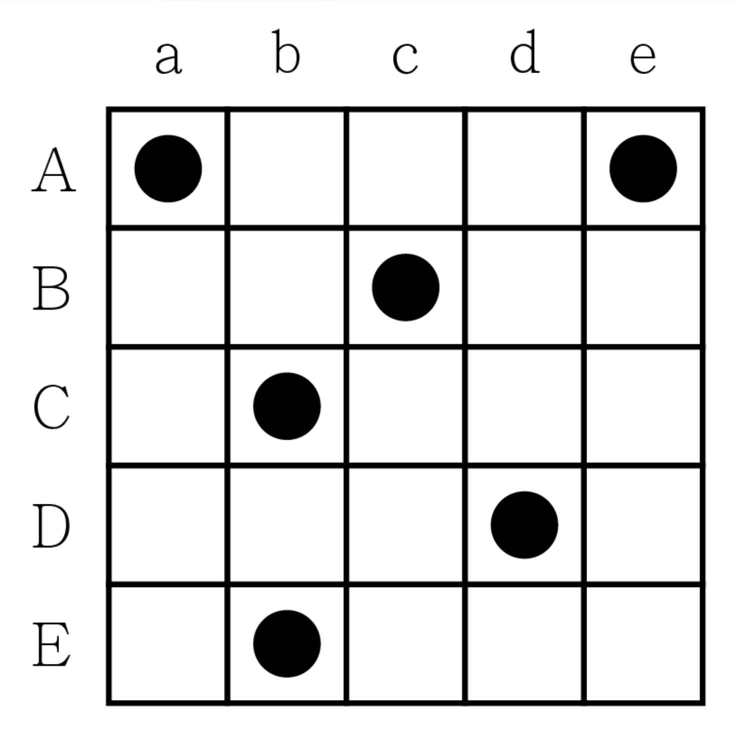
TIOJ 1089 Asteroids

- 此時題目轉化為二分圖上的最小點覆蓋數
- 又因二分圖上最大匹配數等於最小點覆蓋數

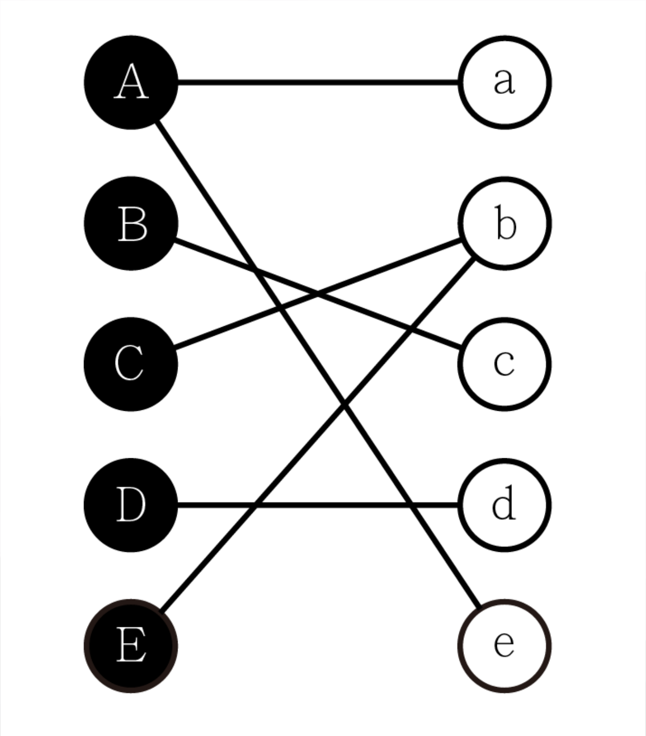


TIOJ 1089 Asteroids

- 因此本題題目要求二分圖上的最大匹配數



⇒



Network Flow

網路流

- 給定一個有向圖，其中每條邊都有自己的容量限制，試問從 A 點到 B 點最大流是多少？

名詞介紹

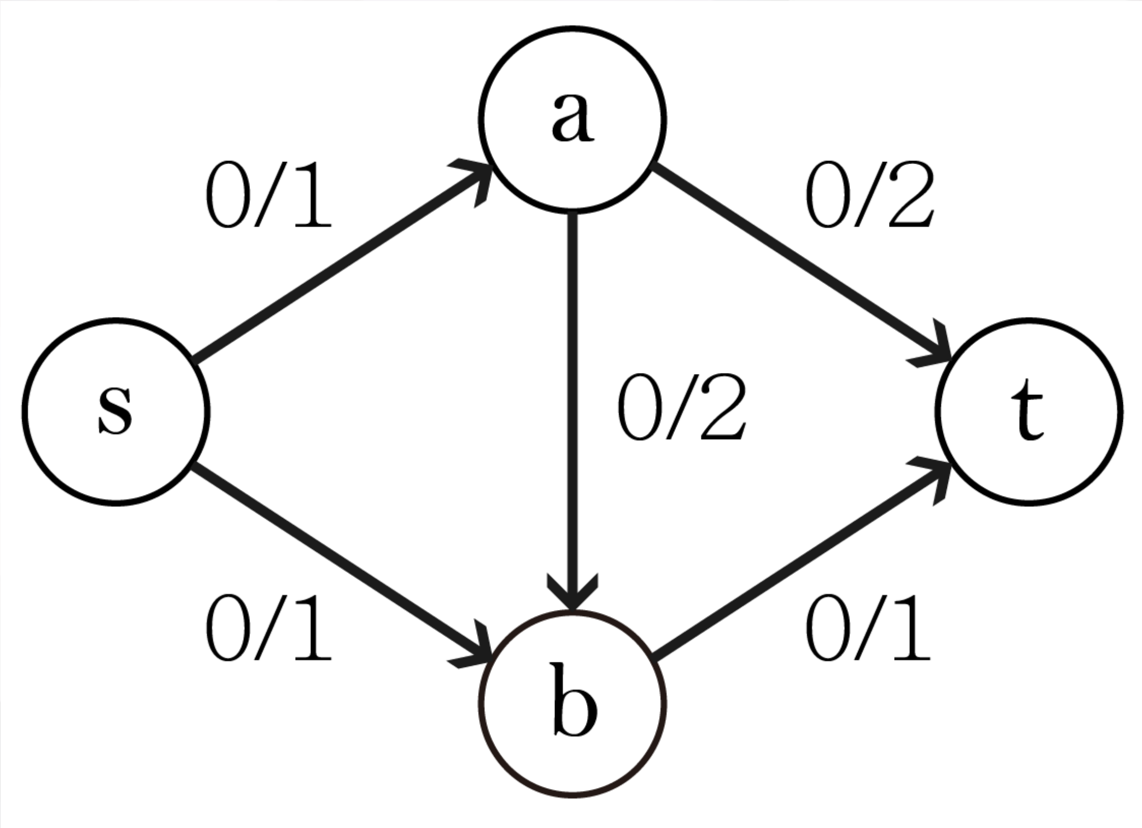
- 管線(Pipe)：就像圖論中的單向邊一樣
- 容量(Capacity)：一條邊的最大流量限制
- 源點(Source)：相當於起點，通常寫作 s
- 匯點(Sink)：相當於終點，通常寫作 t

網路流

- 在一張網路圖中，只有源點能夠供應水流，只有匯點能夠收集水流
- 除了源點及匯點以外的點都只能夠「轉送水流」，也就是說流入的量會等於流出的量
- 從源點供應的水流量會等於匯點收集到的水流量，也就是中途不會有漏水或是增加水

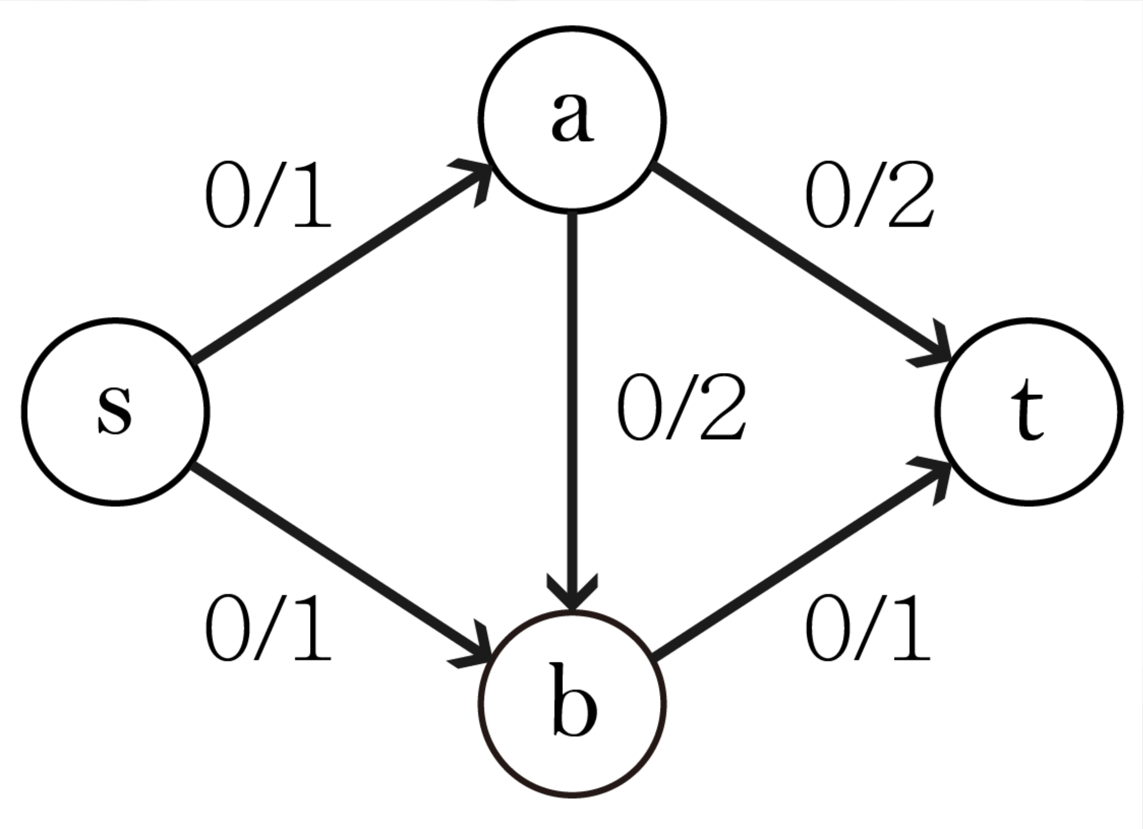
Greedy

- 考慮以下的圖，圖中的數字是 當前流量/容量



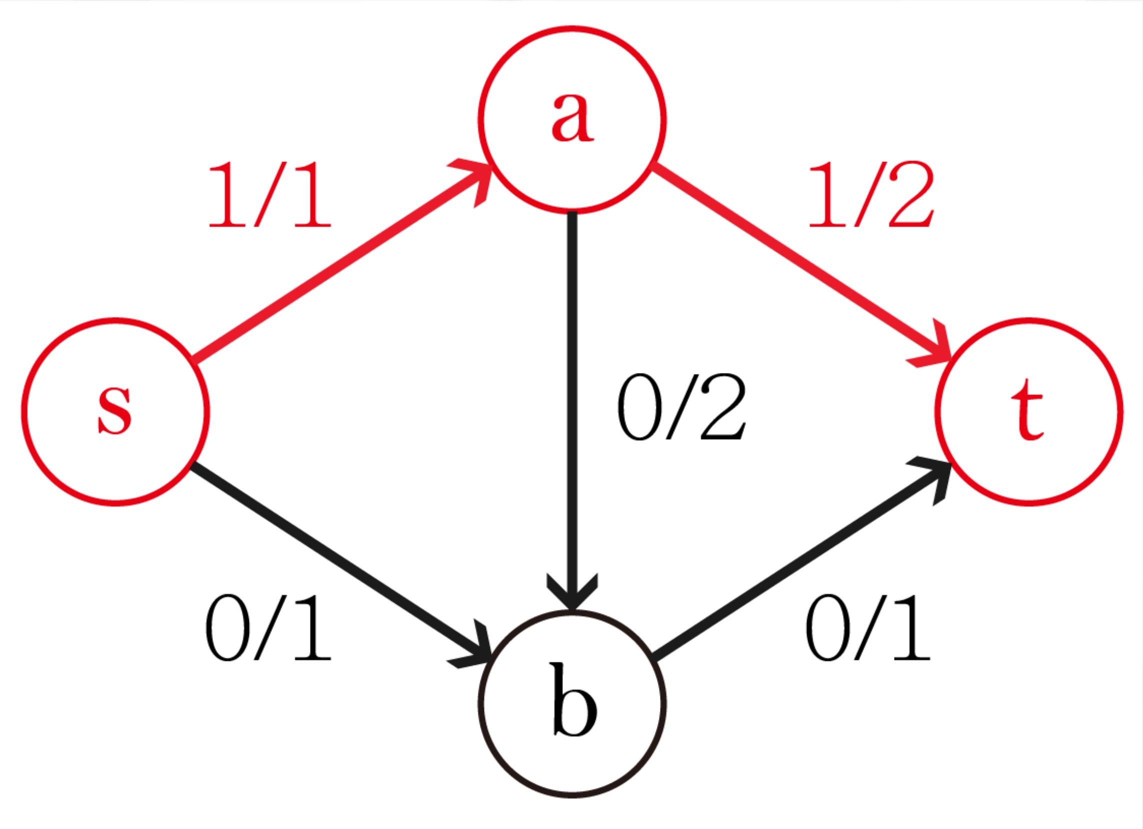
Greedy

- 一直尋找從源點到匯點還沒流滿的邊，然後用那些邊增加流量



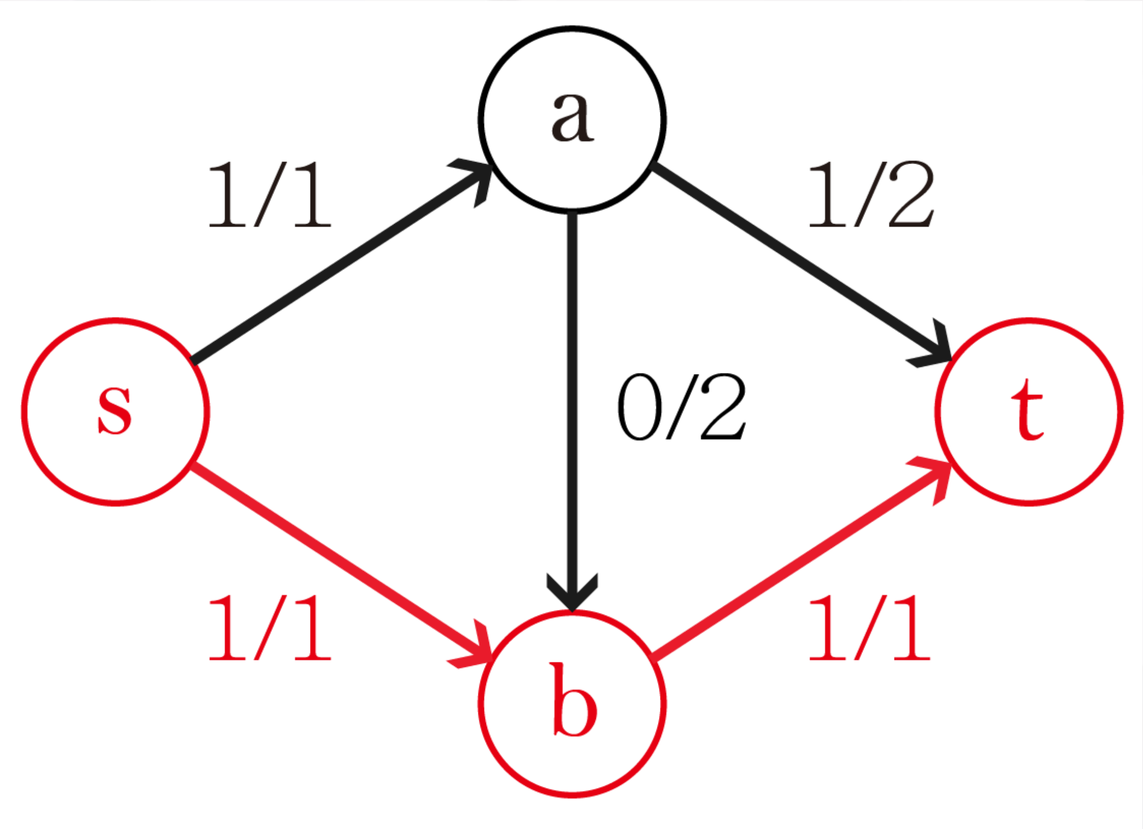
Greedy

- 先依著 $s \to a \to t$ 將流量增加
- 增加後當前流量為 1



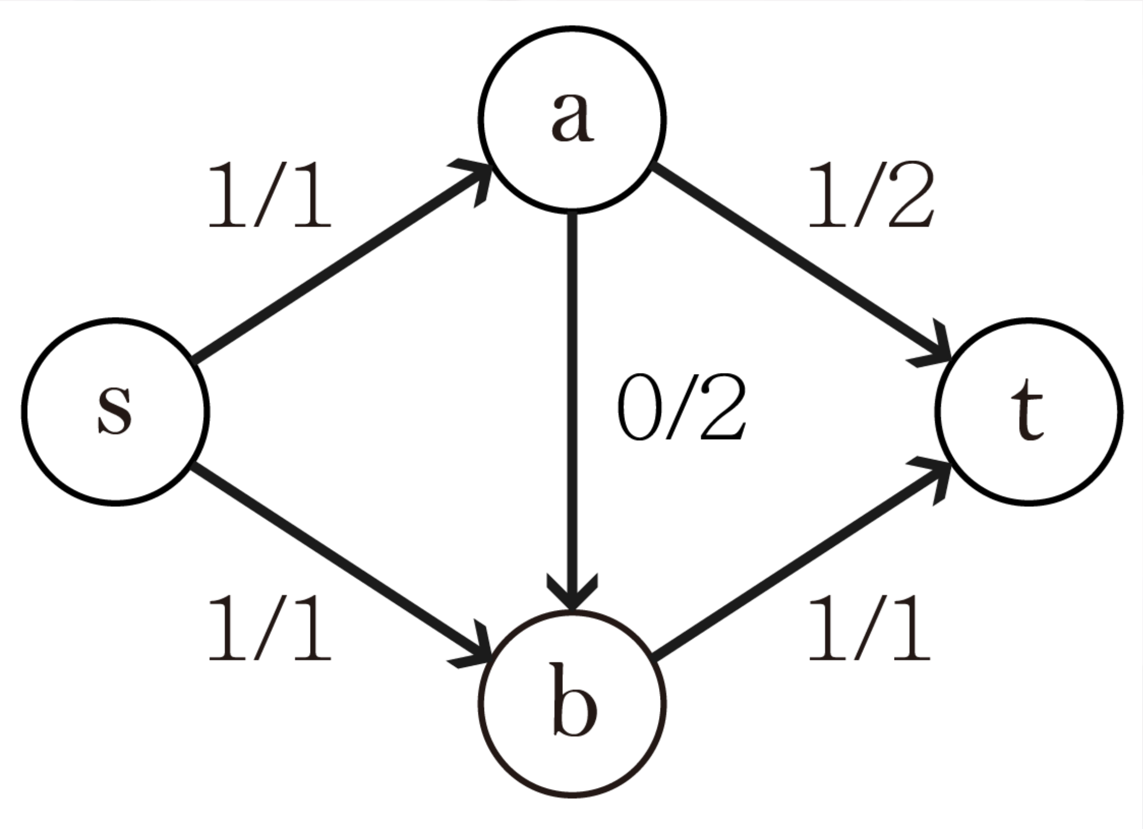
Greedy

- 接著依 $s \to b \to t$ 將流量增加
- 增加後當前流量為 2



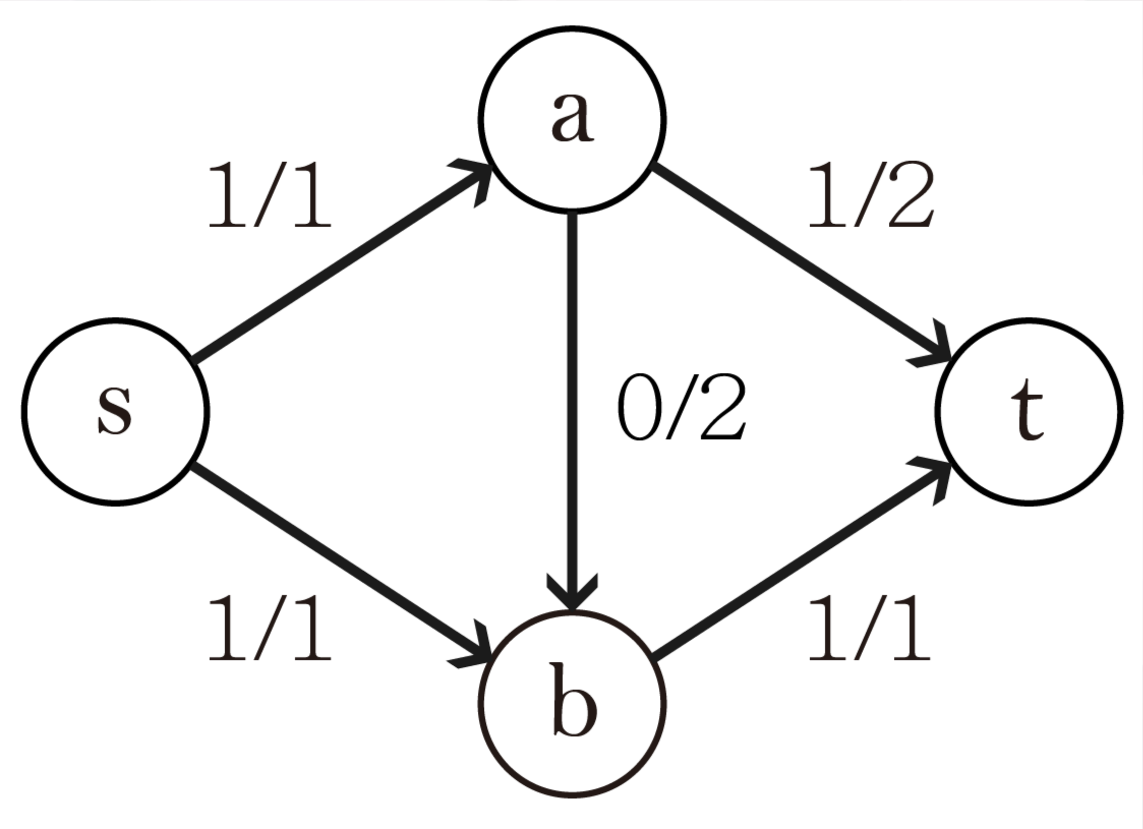
Greedy

- 發現已經沒有邊可以讓流量增加，因此流量為 2 ，經檢查發現其為最大流



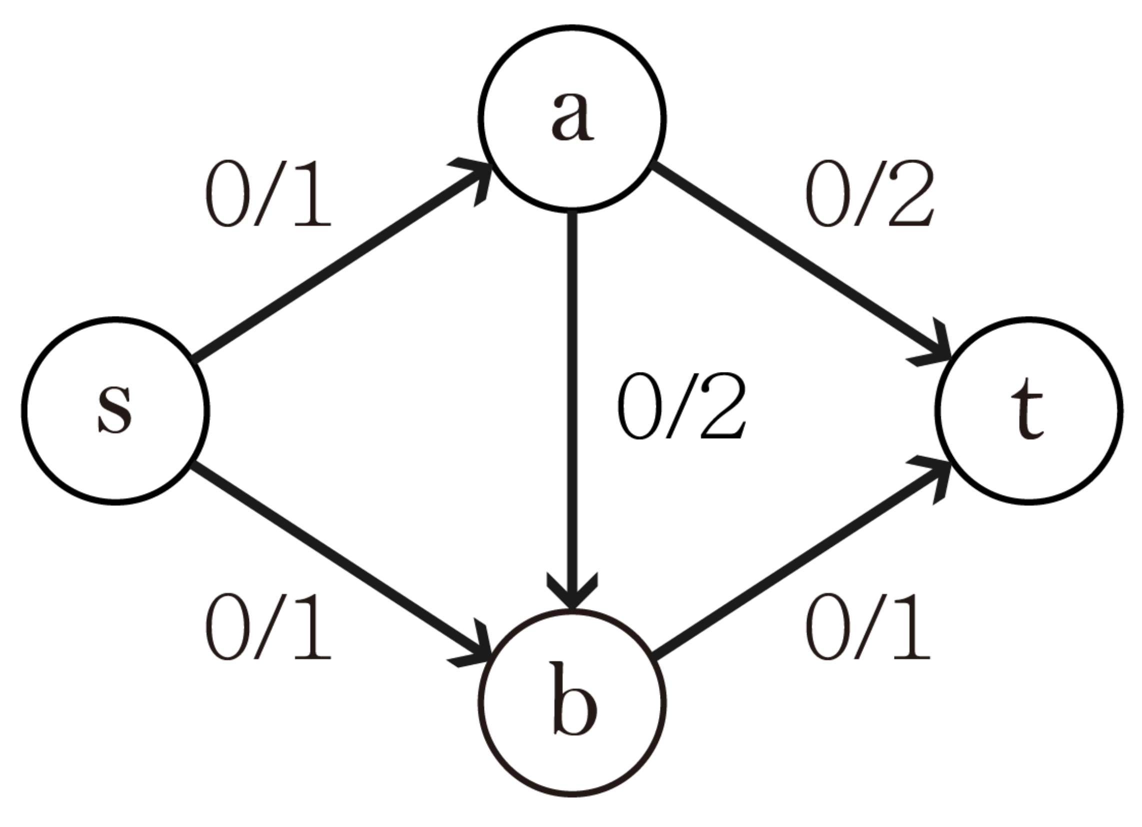
Greedy

- 這種方法「有時」可以找到最大流，但是有時卻會失敗



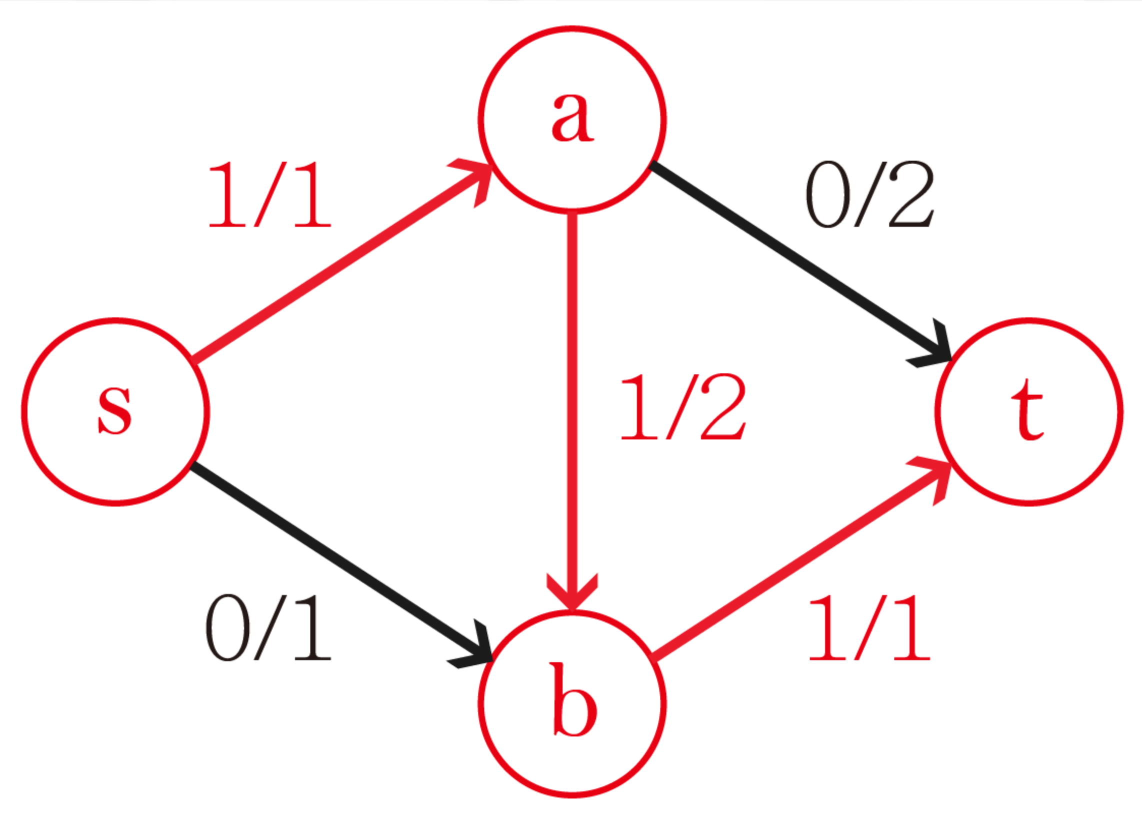
Greedy

- 我們用同一張圖再來跑一次



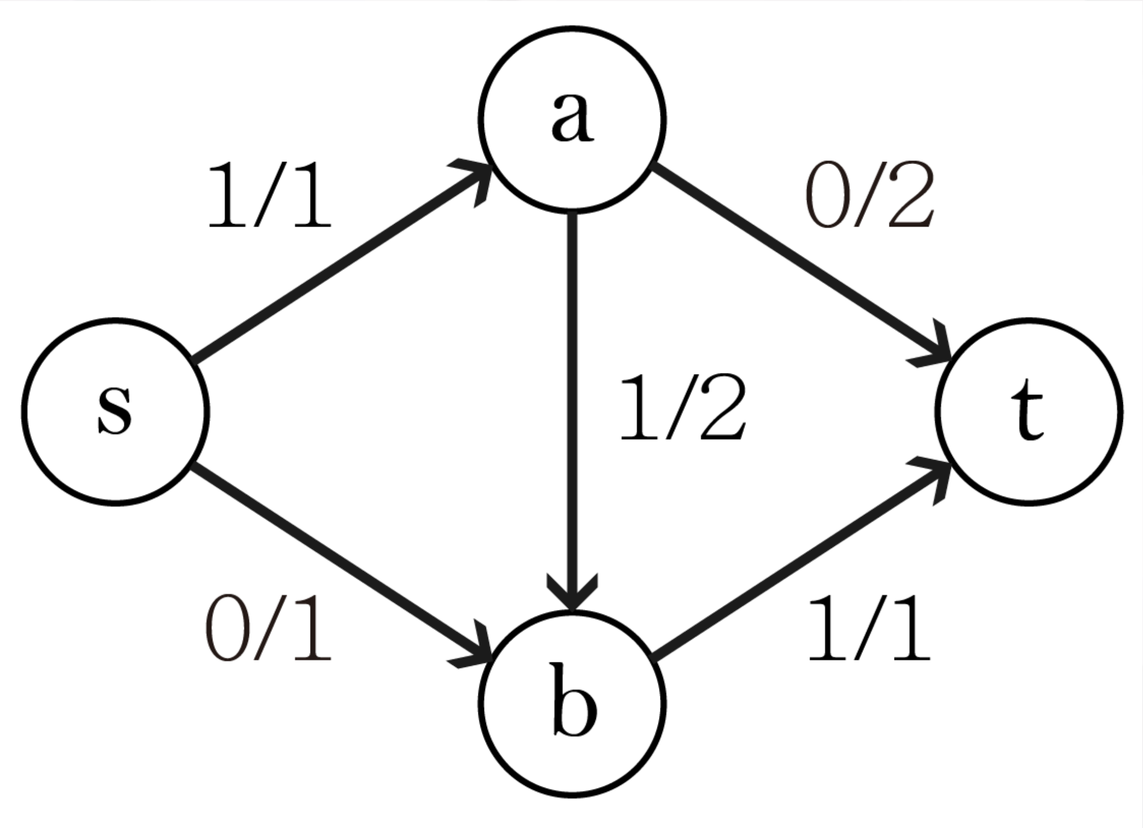
Greedy

- 依著 $s \to a \to b \to t$ 將流量增加
- 增加後當前流量為 1



Greedy

- 發現已經沒有邊可以讓流量增加，因此流量為 1 ，但是這不是最大流



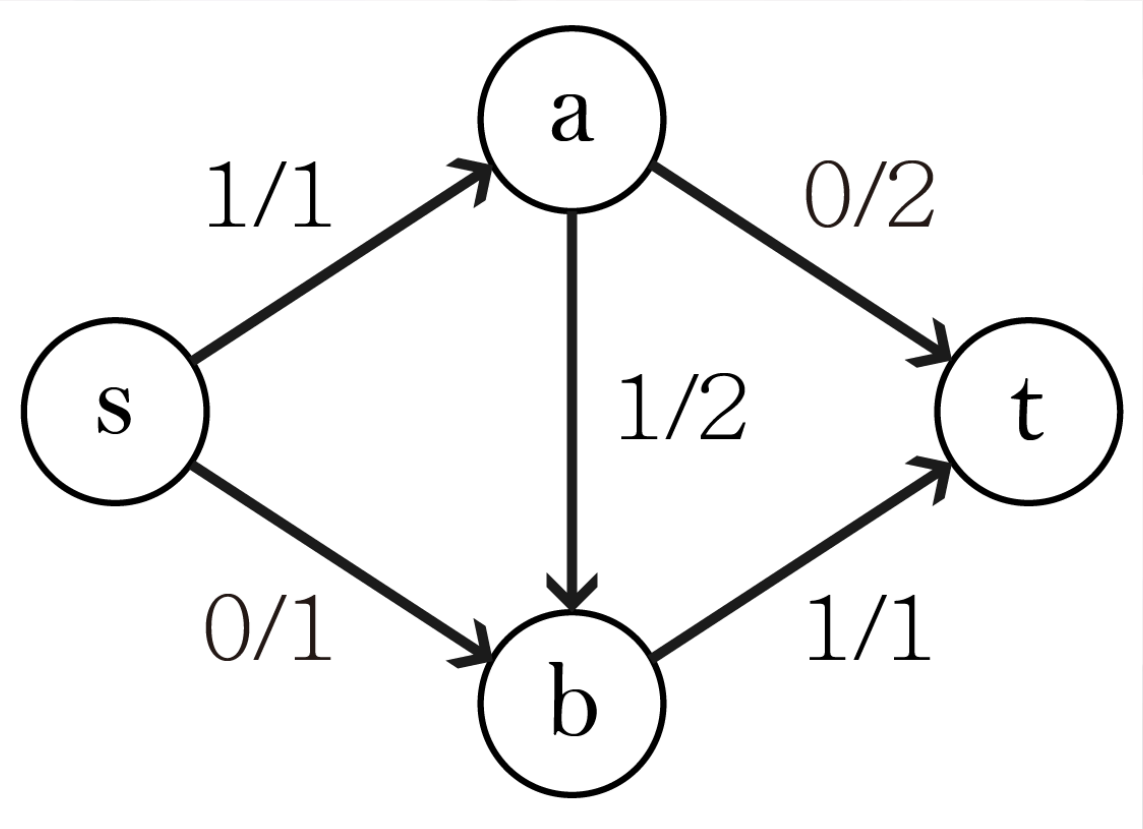
Residual Network

剩餘網路

- 為了解決 Greedy 的問題，我們需要讓他能「逆流」回去抵銷已經流過的地方

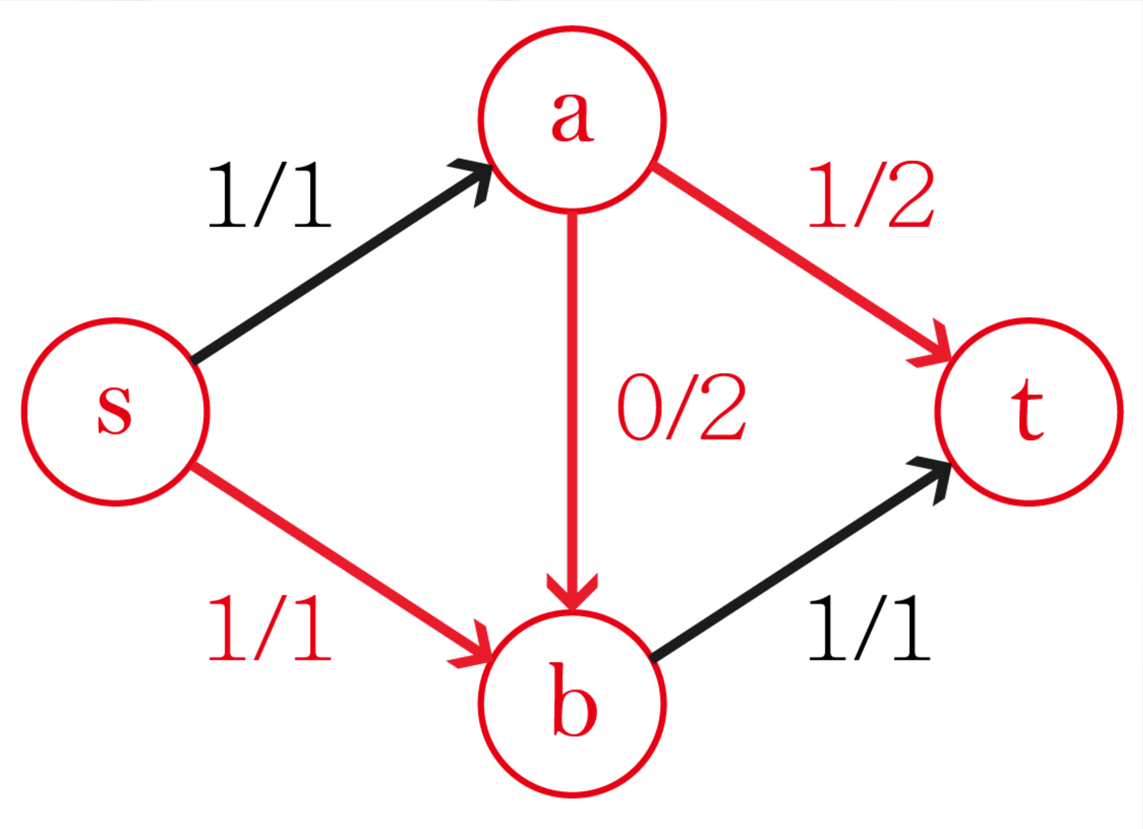
剩餘網路

- 以剛剛這張圖來看，若能讓其由 $s \to b \to a \to t$ 逆流回去抵銷 $a \to b$ 已經流過的流量的話，就可以得到最大流了



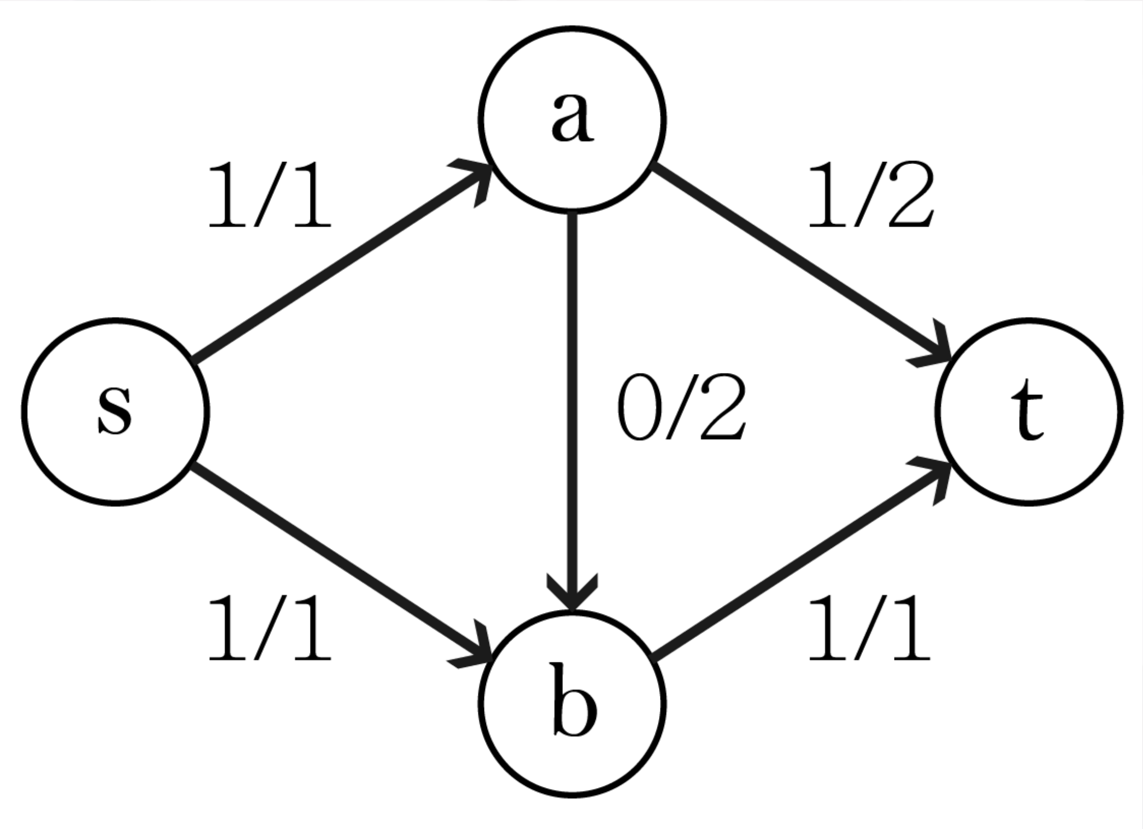
剩餘網路

- 以剛剛這張圖來看，若能讓其由 $s \to b \to a \to t$ 逆流回去抵銷 $a \to b$ 已經流過的流量的話，就可以得到最大流了



剩餘網路

- 以剛剛這張圖來看，若能讓其由 $s \rightarrow b \rightarrow a \rightarrow t$ 逆流回去抵銷 $a \rightarrow b$ 已經流過的流量的話，就可以得到最大流了



剩餘網路

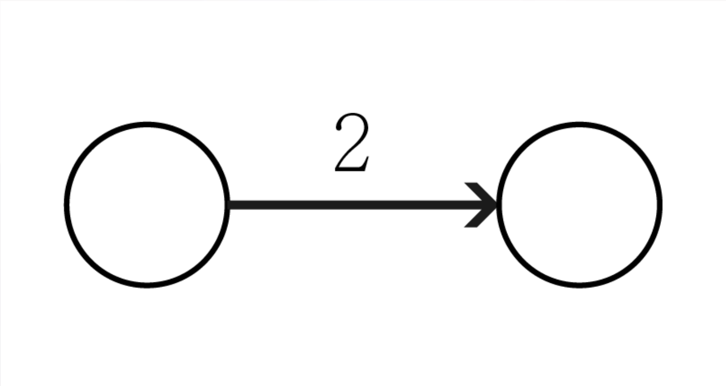
- 由剛剛的範例可知，每次在增加流量時，會做兩種事：

1. 將沒流滿的邊增加流量
2. 將已有流量的邊「逆流」回去抵銷原本的流量

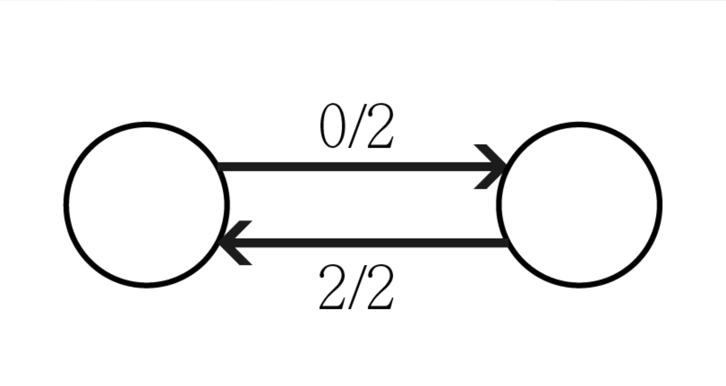
剩餘網路

- 為了方便處理，我們對原圖進行一些變化：

1. 對每條邊紀錄其流量
2. 每條邊都增加一條容量相同的反向邊，並且在一開始將其流量設為原本那條邊的容量



$\Rightarrow$



剩餘網路

- 經轉換後我們定義剩餘流量 $r(u,v)$

$$r(u,v) = c(u,v) - f(u,v)$$

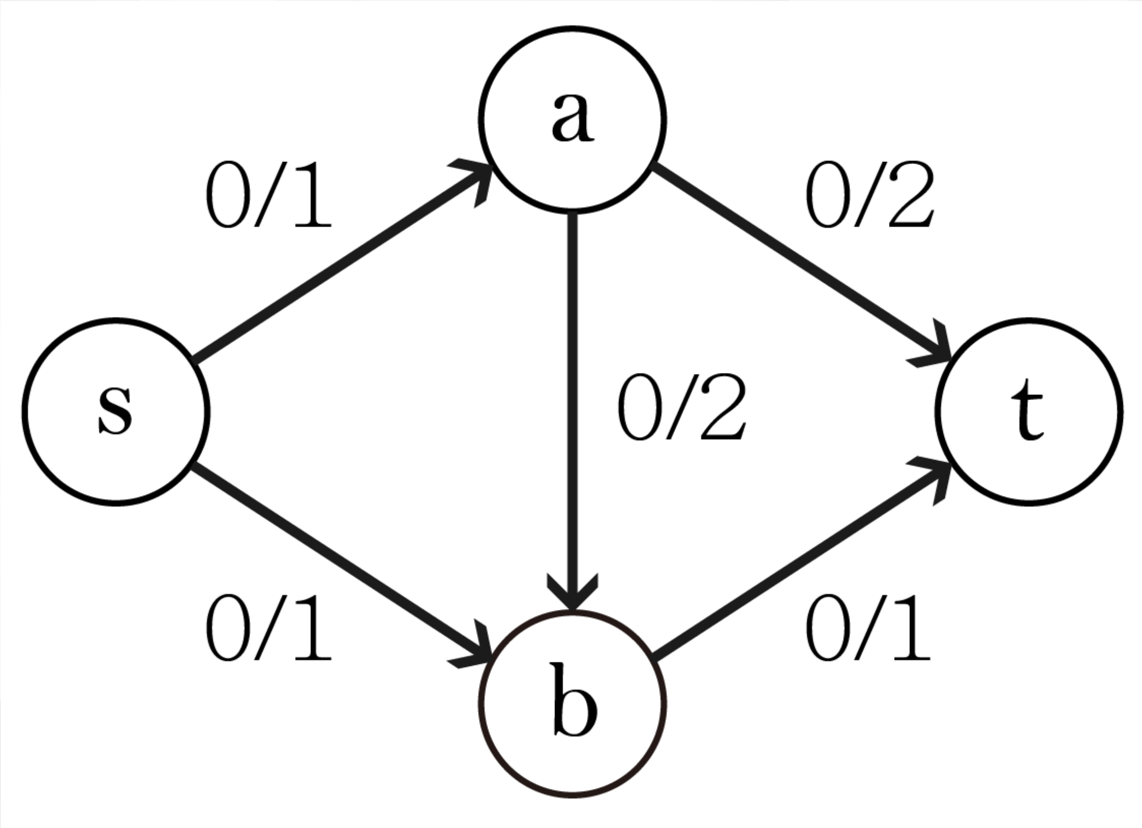
- $c(u,v)$ 代表從 u 到 v 這條邊的容量
- $f(u,v)$ 代表從 u 到 v 這條邊當前的流量

剩餘網路

- 轉換過後的圖稱之為「剩餘網路」
- 在剩餘網路中會將 $r(u,v)=0$ 的邊視為不存在
- 如果存在一條從 x 走到 y 的路徑且路徑上每條邊的剩餘流量均大於零，則我們將其稱為「增廣路」

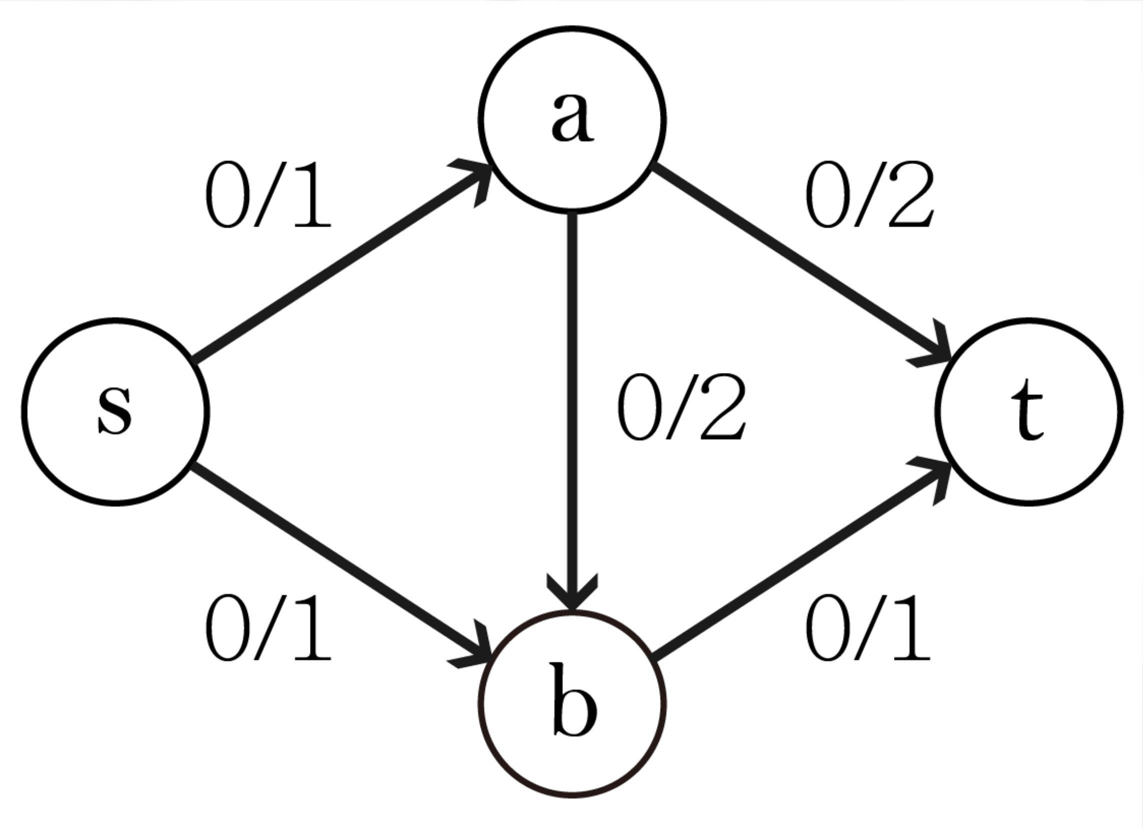
剩餘網路

- 再以剛剛的圖為例



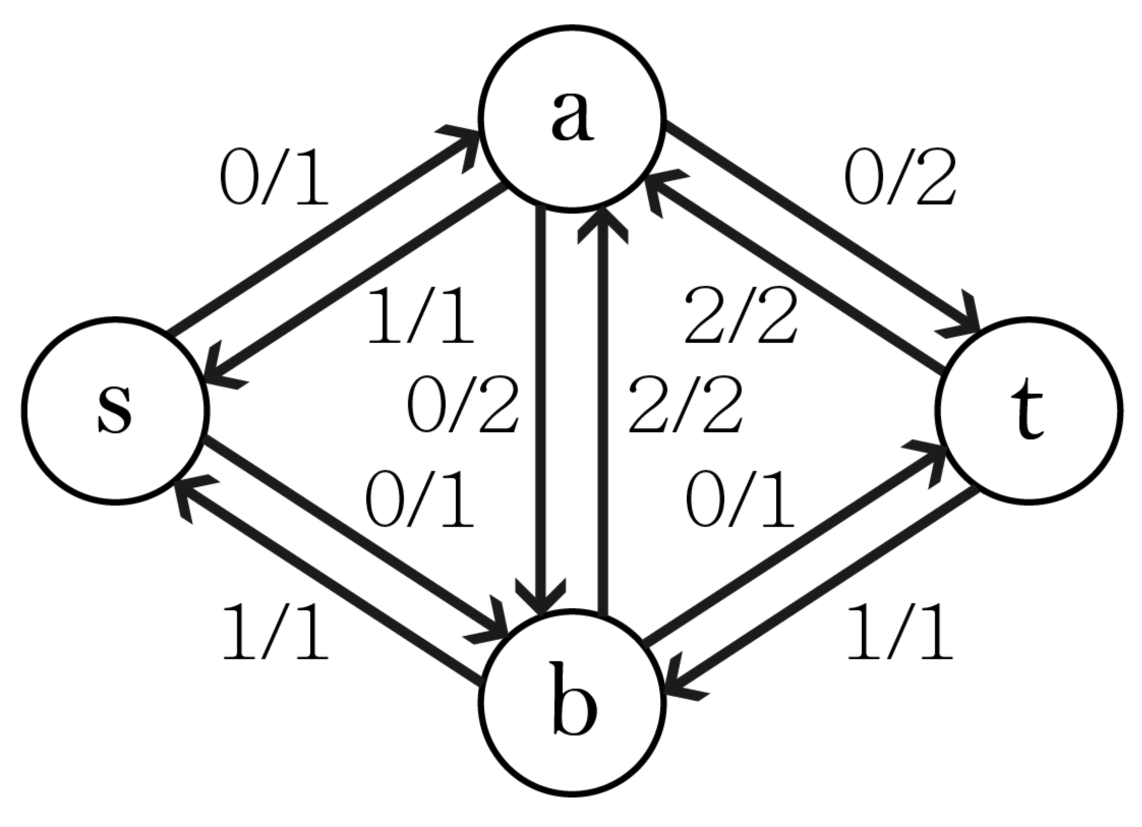
剩餘網路

- 先將其轉換為剩餘網路



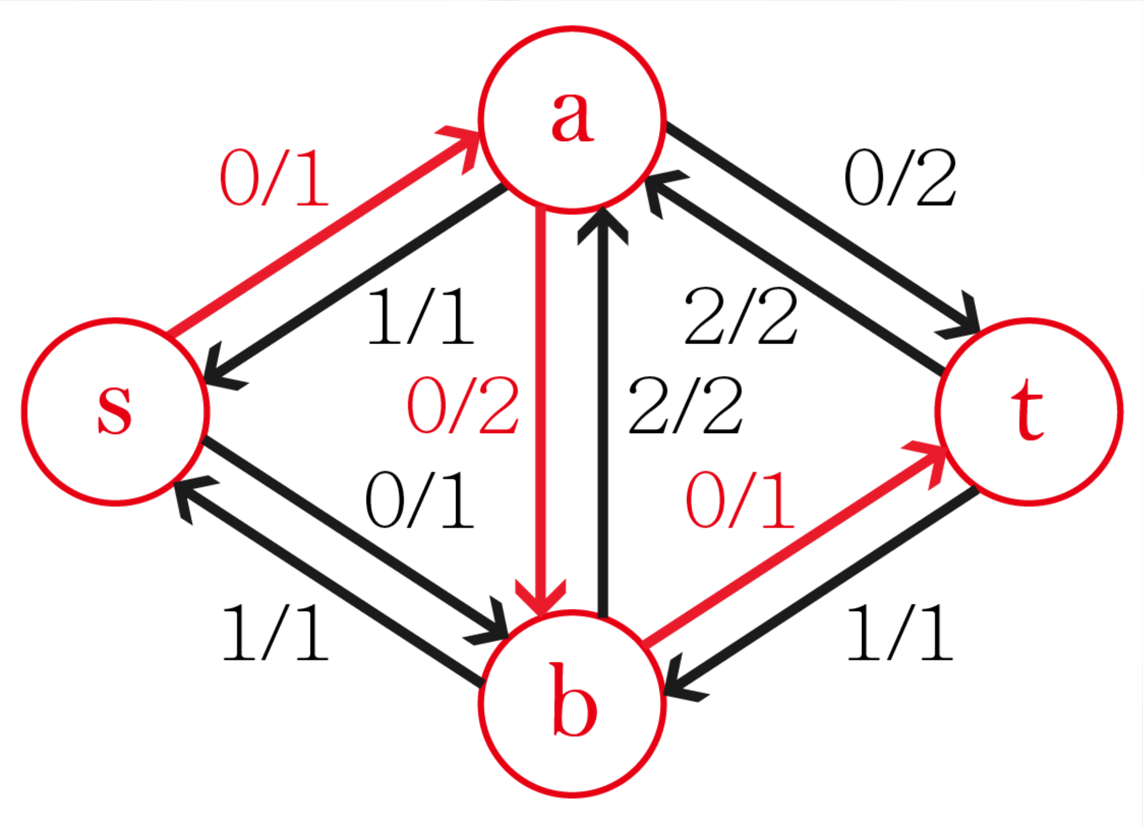
剩餘網路

- 先將其轉換為剩餘網路



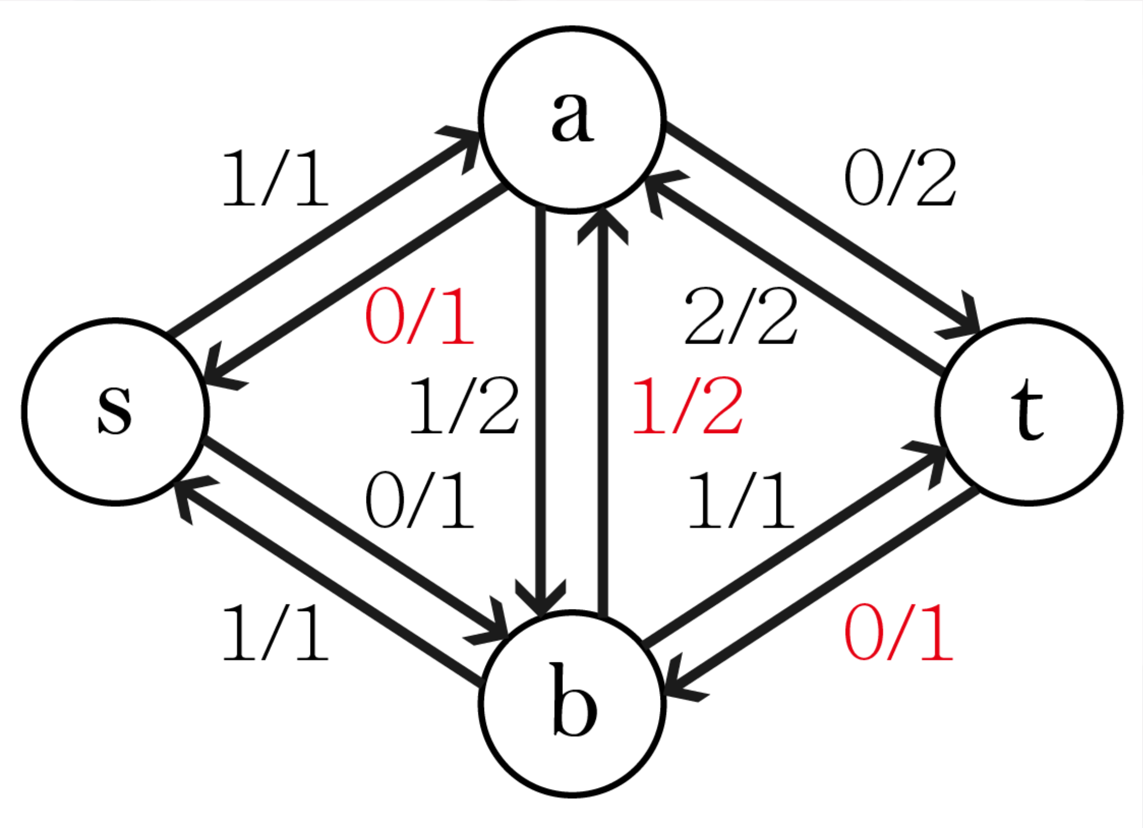
剩餘網路

- 接著找出一條從 s 到 t 的增廣路



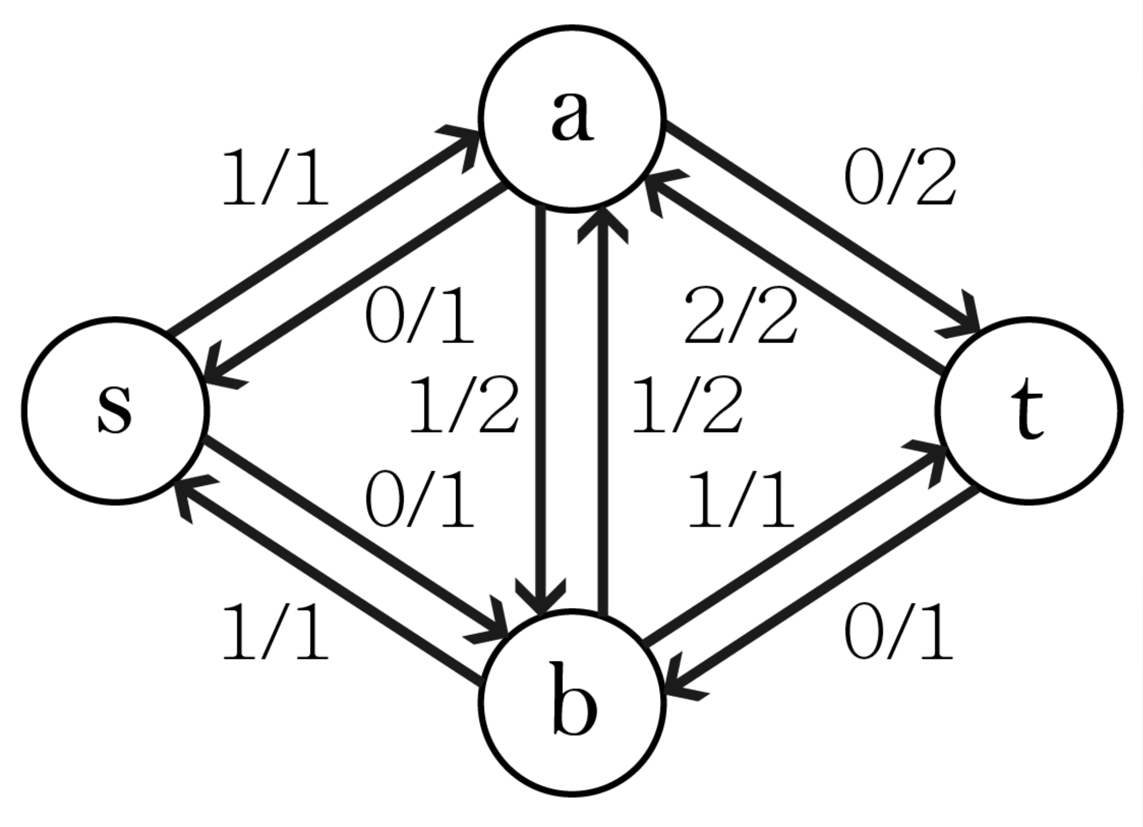
剩餘網路

- 將該增廣路增廣，總流量增加 1
- 注意增廣後會使反向邊的當前流量也減少相同數值



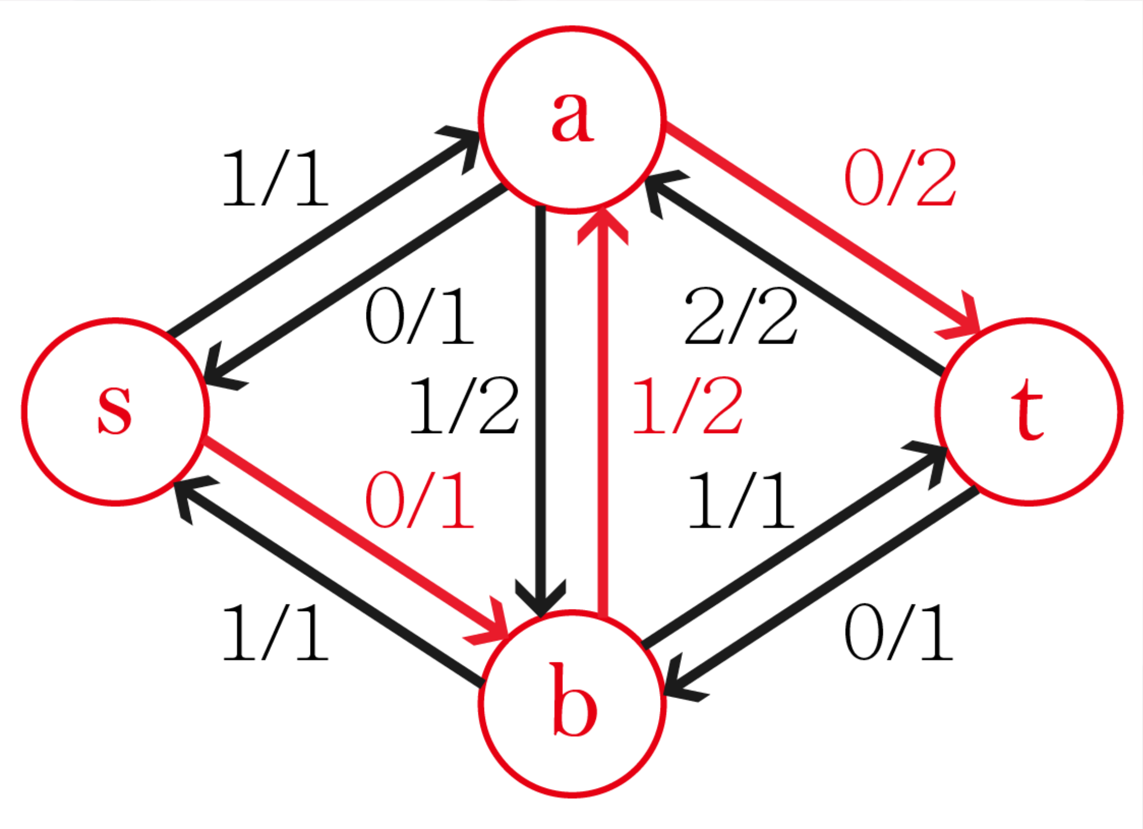
剩餘網路

- 繼續尋找增廣路並增廣



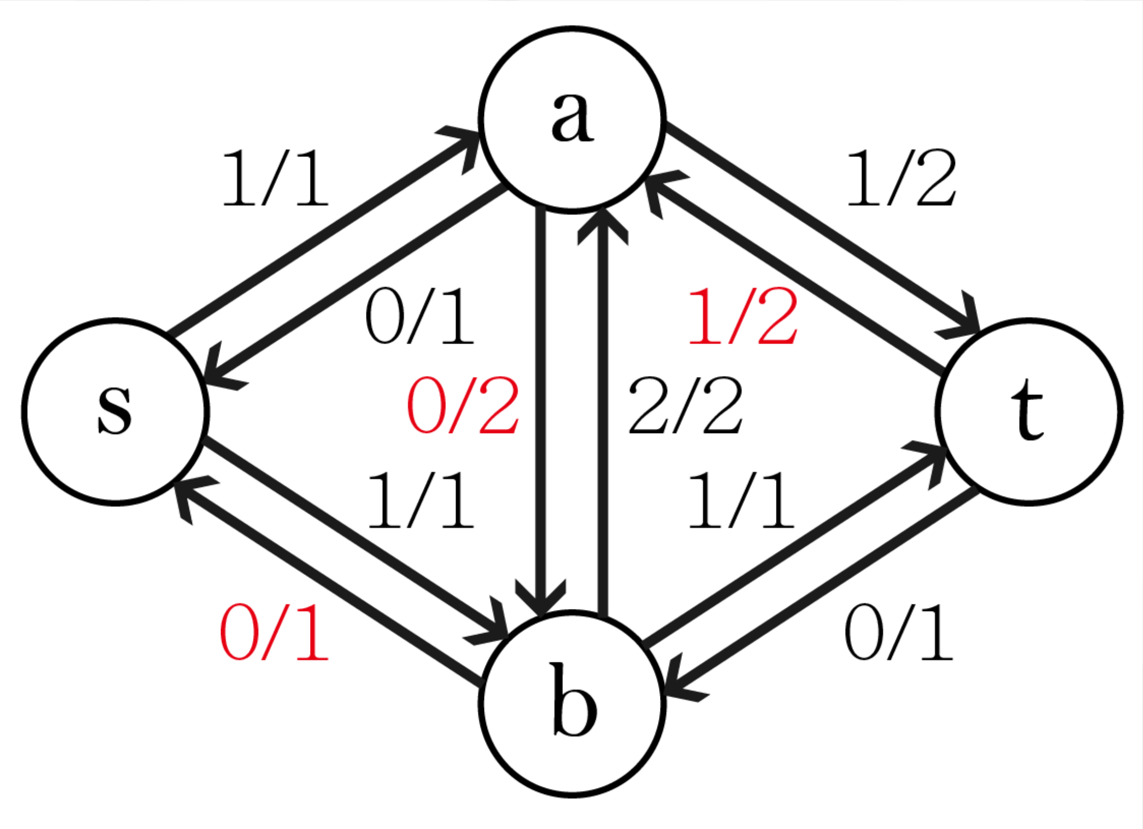
剩餘網路

- 繼續尋找增廣路並增廣



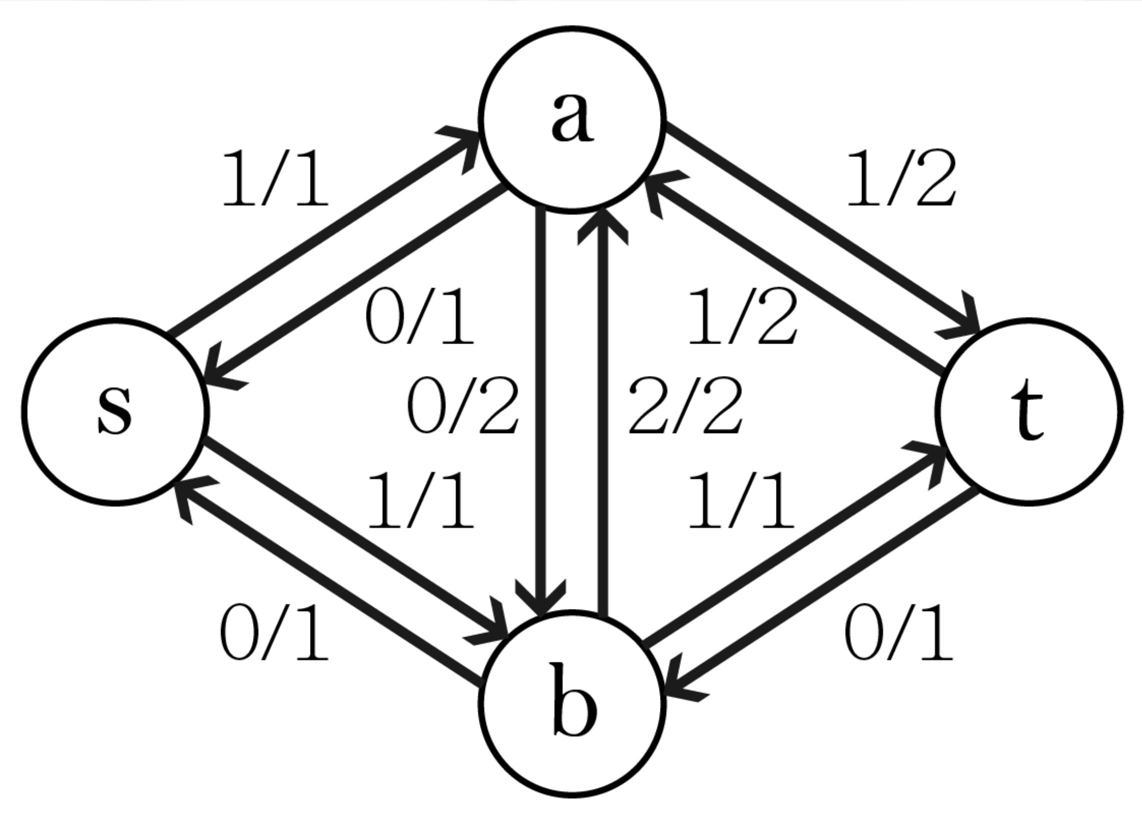
剩餘網路

- 總流量再增加 1
- 要記得反向邊也要減少相等的流量



剩餘網路

- 發現在剩餘網路上已經找不到增廣路了，此時得到的總流量 2 為最大流



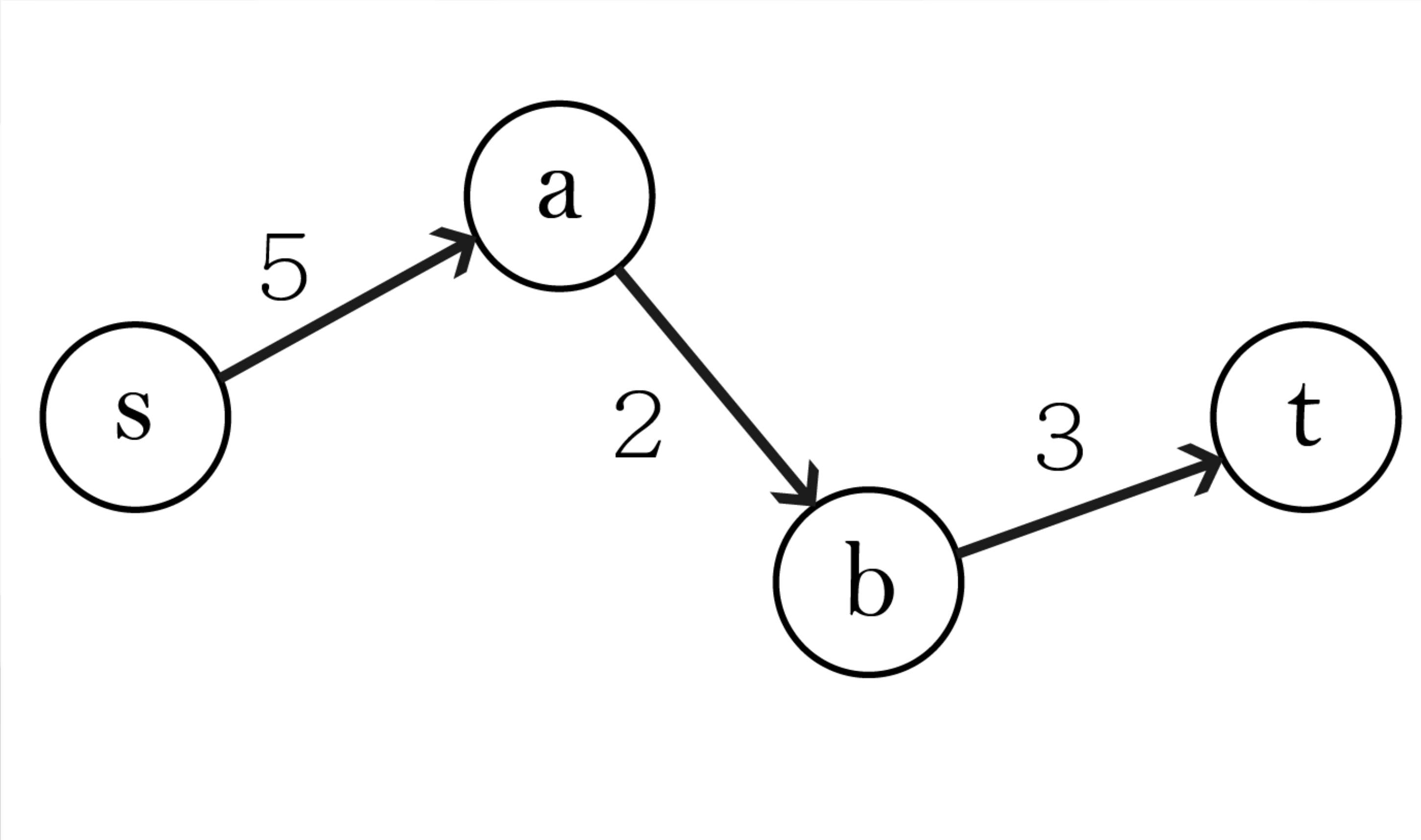
剩餘網路

- 問題來了，每次進行增廣時要增加多少總流量呢？

- 你會發現，假設你選了一條路徑 $s \to v_1 \to v_2 \to \cdots \to v_k \to t$
  則增加的總流量為 $\min\{r(s, v_1), r(v_1, v_2), \cdots, r(v_k, t)\}$
  也就是路徑上剩餘流量的最小值

剩餘網路

- 如下圖的增廣路可以增加 2 單位的總流量



Max Flow

如何存圖

- 在最大流中，每一條邊會需要儲存
  起點、終點、容量、當前流量
- 通常在維護剩餘網路時會需要維護反向邊才能快速找到反向邊的位置，因此會多存反向邊的「位置」

如何存圖

- 如圖

```
struct Edge{
    int from, to, cap, flow, rev;
}
```

- 在 rev 的部份我們存反向邊所在的 index 值
- 使用 vector<Edge> 就能存圖了

如何存圖

- 通常筆者會使用另外一種方式，因為既然使用 vector 了，那也沒必要存起點資訊，最終簡化成這樣

```
struct Edge{
    int to, cap, flow, rev;
}
```

- 當要增加一條從 u 到 v ，容量為 cap 的邊時，做法如下：

```
void add_edge(int u, int v, int cap){
    G[u].push_back(Edge{v, cap, 0, G[v].size()});
    G[v].push_back(Edge{u, 0, 0, G[u].size() - 1});
}
```

如何存圖

- 當要將某條邊 e 的流量增加 df 單位時如下：

```
e.flow += df;
G[e.to][e.rev].flow -= df;
```

如何存圖

- 欸？怎麼把反向邊的容量及初始流量都設為 0 ，這樣某條邊增加流量時反向邊的流量不就變負的了嗎
- 在剩餘網路中，我們只關心剩餘容量是否為 0 ，也就是 cap – flow 的值，只要注意初始值的 cap 要等於 flow，因此就算寫成下面這樣也沒問題

```cpp
void add_edge(int u, int v, int cap){
    G[u].push_back(Edge{v, cap, 0, G[v].size()});
    G[v].push_back(Edge{u, 0, 0, G[u].size() - 1});
}
```

```cpp
void add_edge(int u, int v, int cap){
    G[u].push_back(Edge{v, cap, 0, G[v].size()});
    G[v].push_back(Edge{u, 500, 500, G[u].size() - 1});
}
```

```cpp
void add_edge(int u, int v, int cap){
    G[u].push_back(Edge{v, cap, 0, G[v].size()});
    G[v].push_back(Edge{u, 1e9, 1e9, G[u].size() - 1});
}
```

Ford-Fulkerson

- 這個做法就如同剛剛講的，每次找到增廣路，並將其增廣

- 我們先假設這個增廣路算法會是正確的，詳細證明留到後面再說

Ford-Fulkerson 時間複雜度

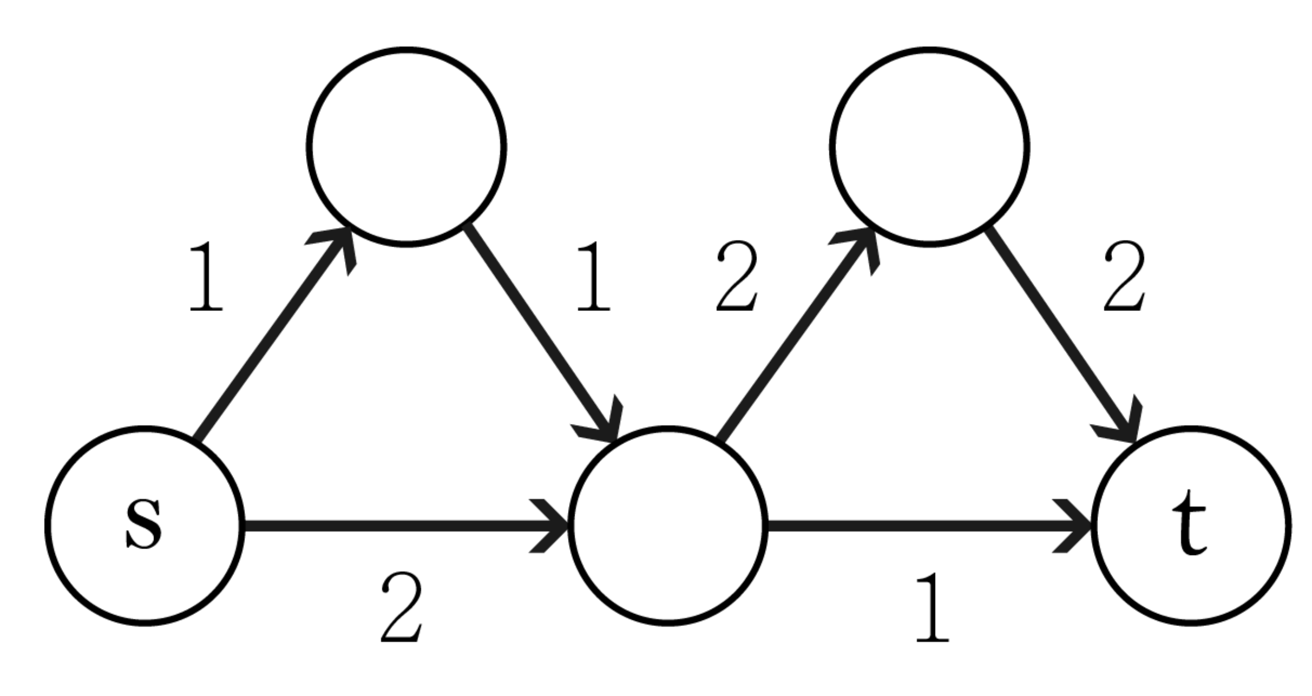
- 每次找尋增廣路時，會需要尋找所有從源點到匯點的路徑，複雜度 $O(m)$ ，找到增廣路後會需要對增廣路上所有的邊進行修改，從源點到匯點至多經過 $n-1$ 條邊，因此修改邊的複雜度為 $O(n)$ ，每次增廣至少增加 1 單位的流量，如果最大流為 F ，則最多增廣 F 次
- 總複雜度 $O\big(\mathrm{F}\times(m+n)\big)=O(mF)$

Edmonds-Karp

- $O(mF)$ 這個時間複雜度，在大部分情況下是不能被接受的，假設最大流非常大，那麼這個算法不足以解決問題，那麼要如何優化呢？

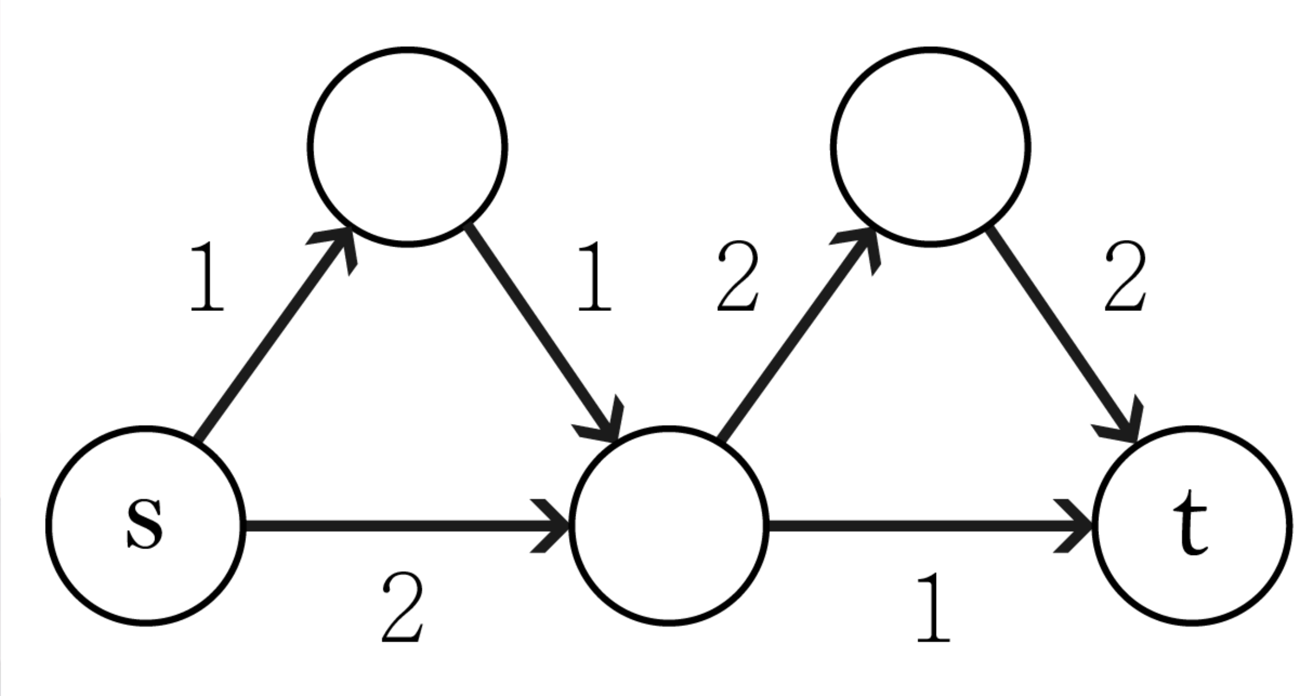
Edmonds-Karp

- 每次在找尋增廣路時，都找距離最短，也就是**經過邊數最少**的增廣路，以下圖為例



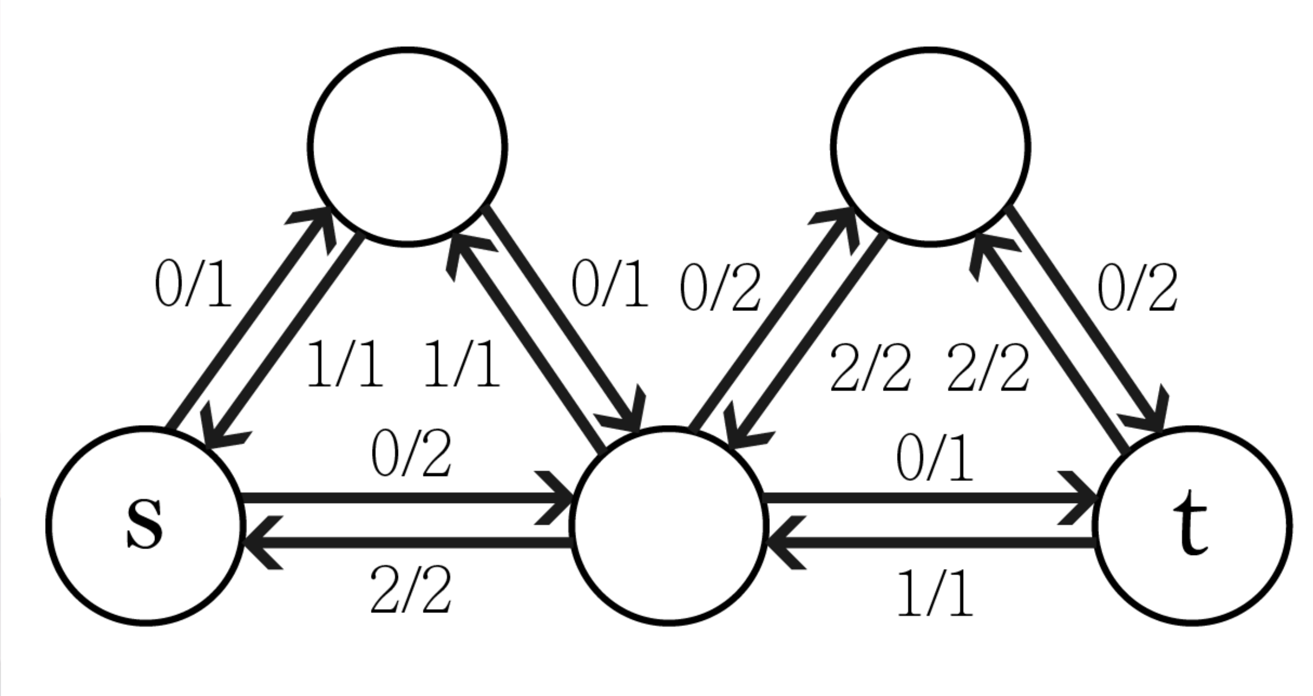
Edmonds-Karp

- 將其轉換為剩餘網路



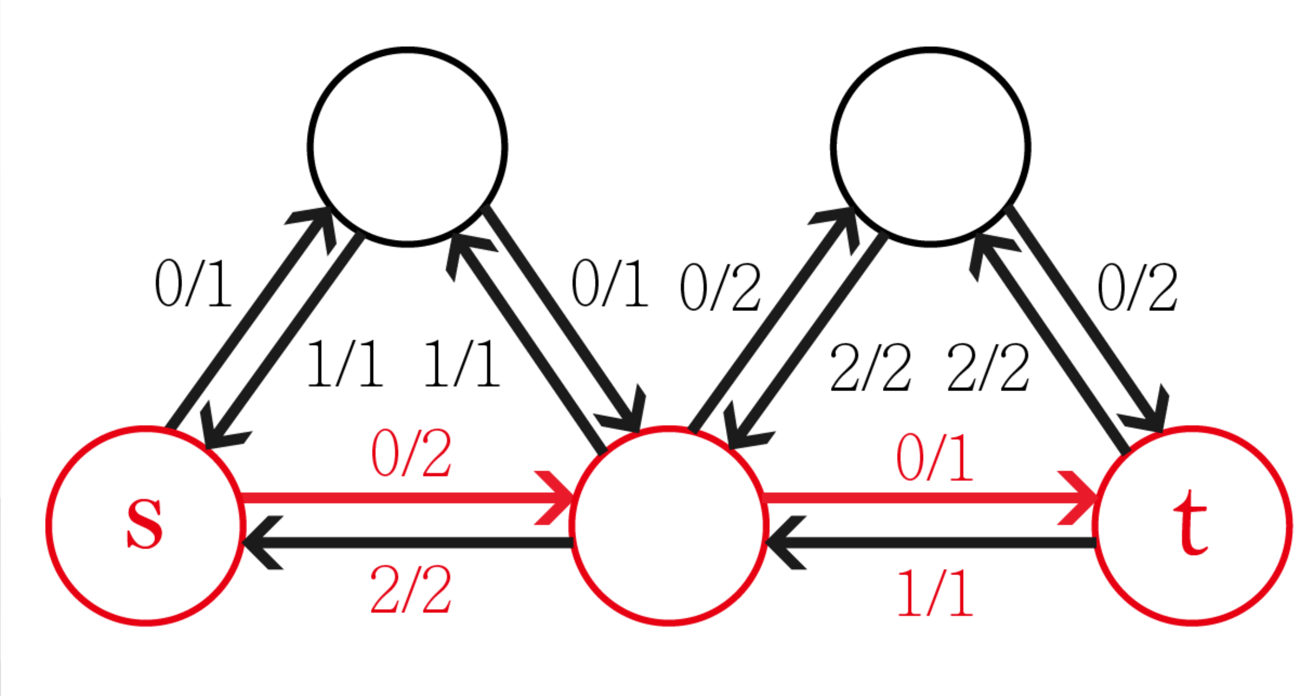
Edmonds-Karp

- 將其轉換為剩餘網路



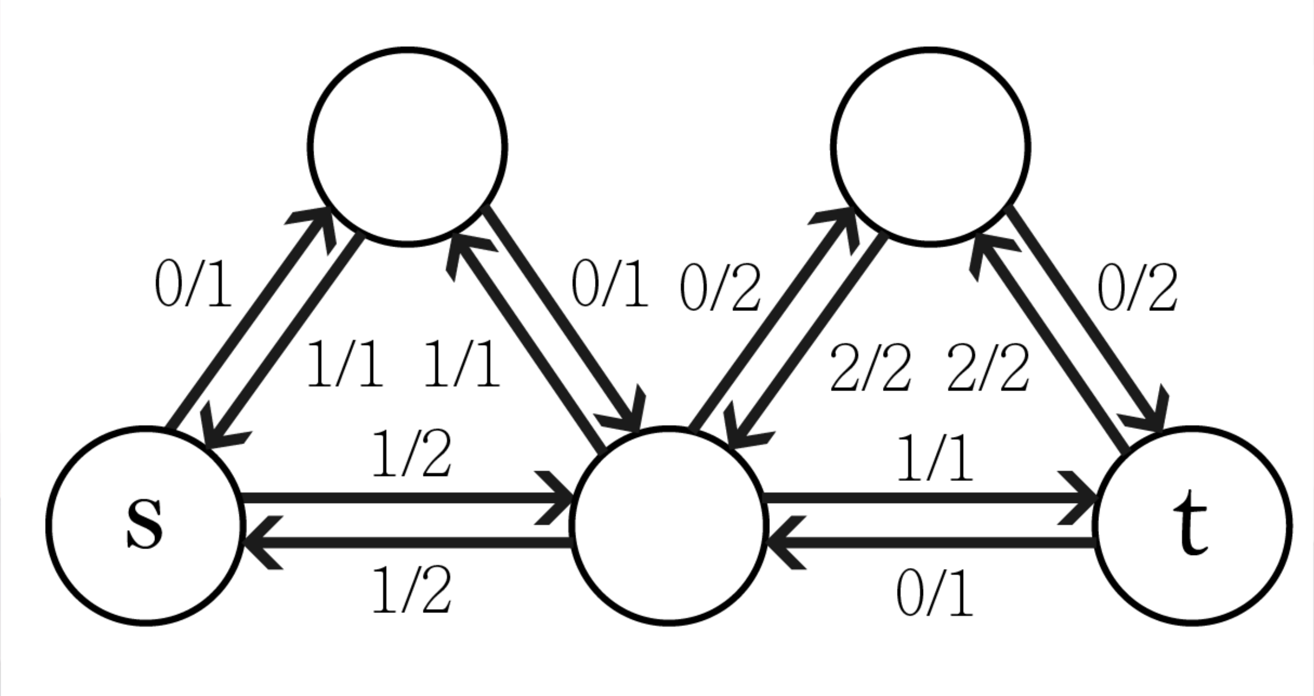
Edmonds-Karp

- 找尋最短的增廣路



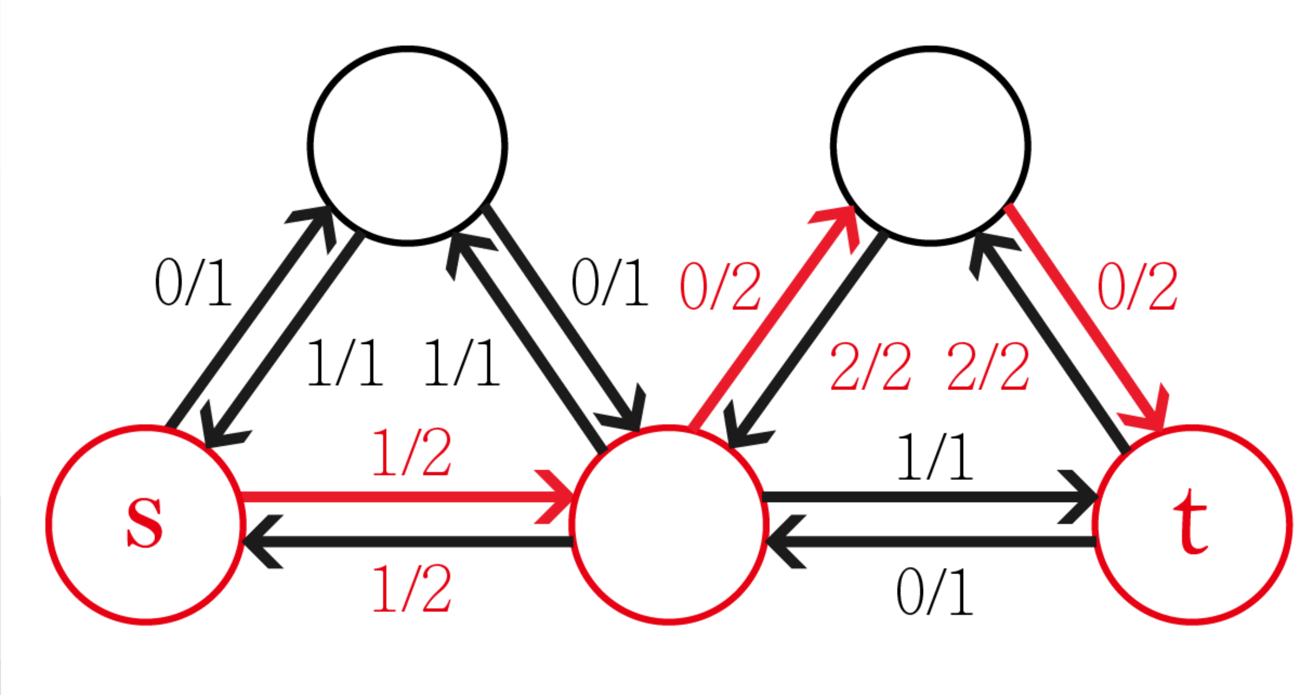
Edmonds-Karp

- 找尋最短的增廣路
- 總流量增加 1



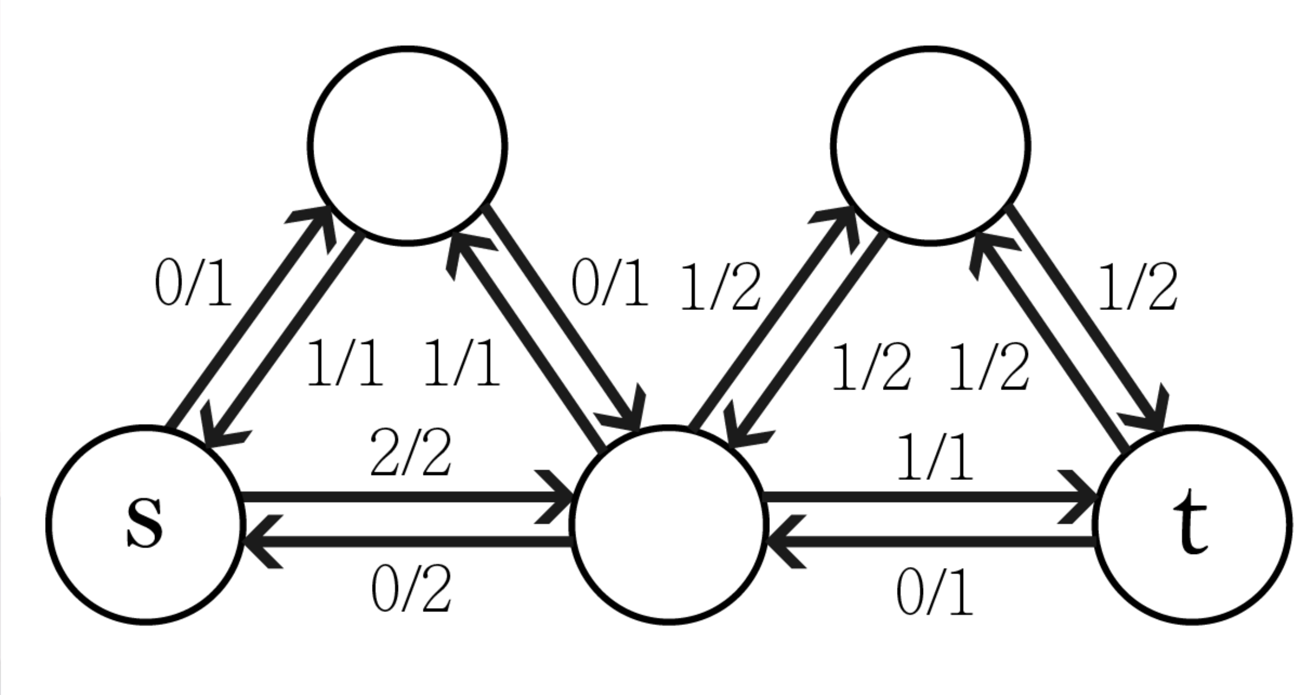
Edmonds-Karp

- 繼續找尋最短的增廣路



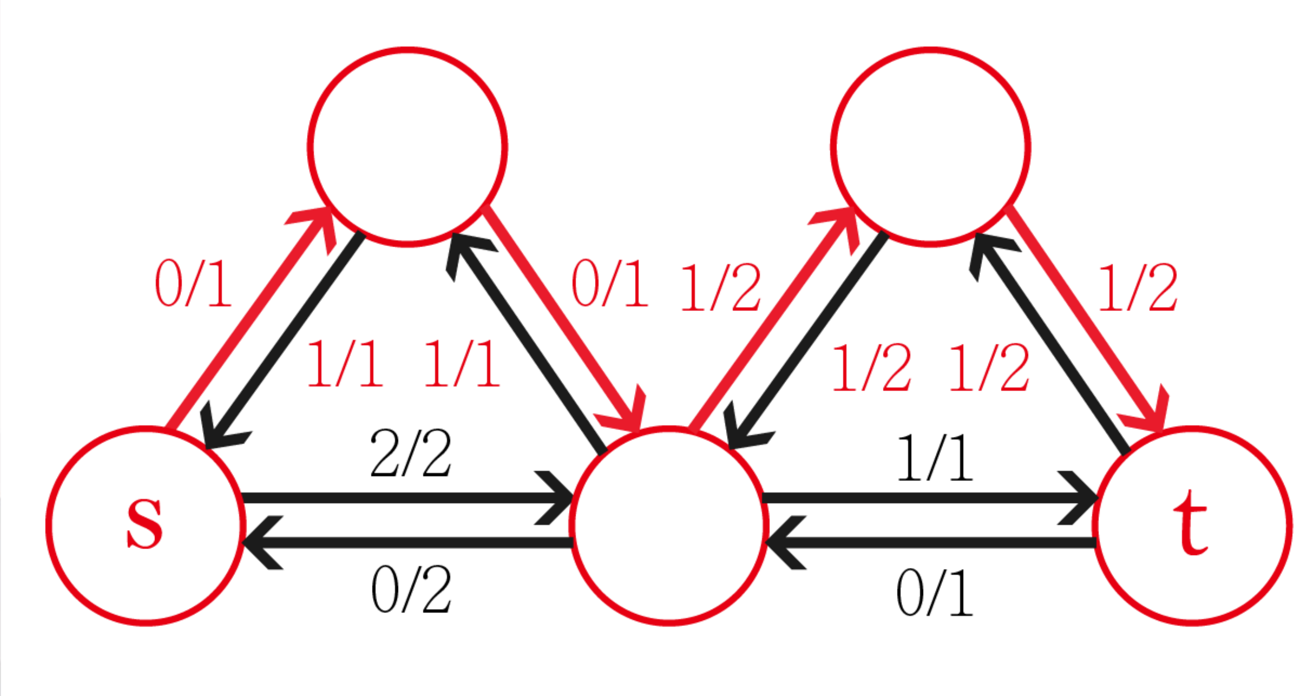
Edmonds-Karp

- 繼續找尋最短的增廣路
- 總流量增加 1



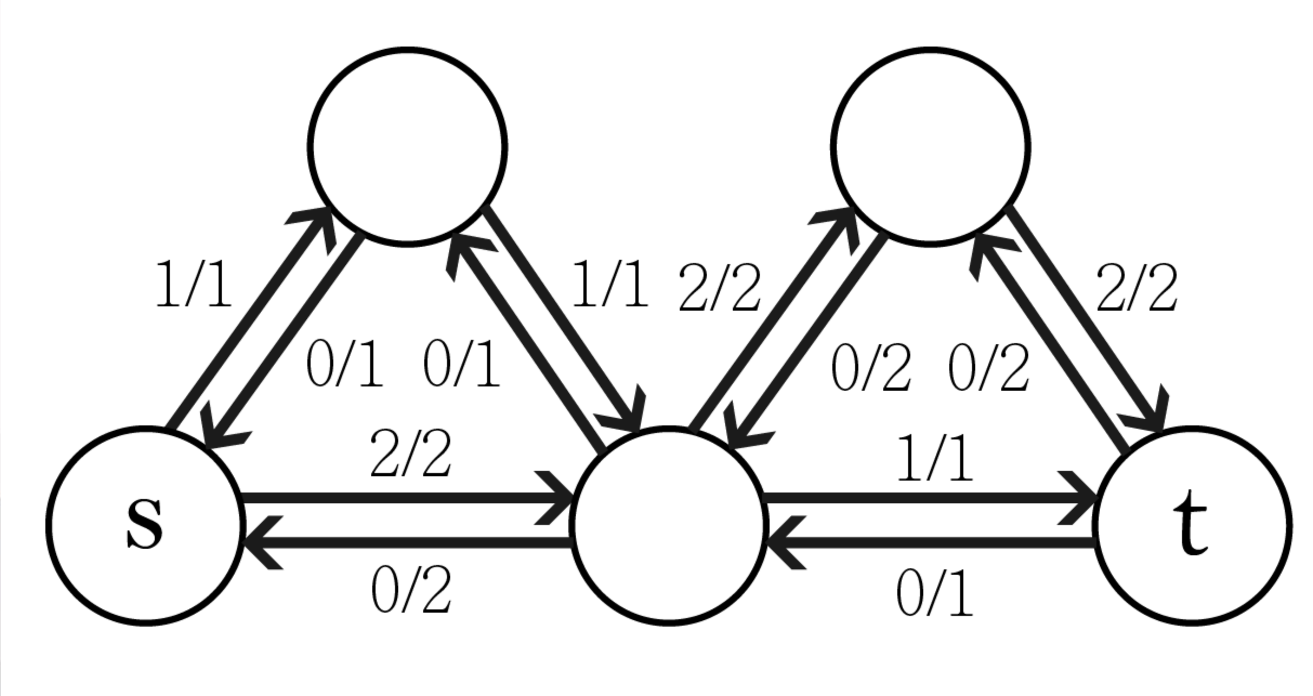
Edmonds-Karp

- 繼續找尋最短的增廣路



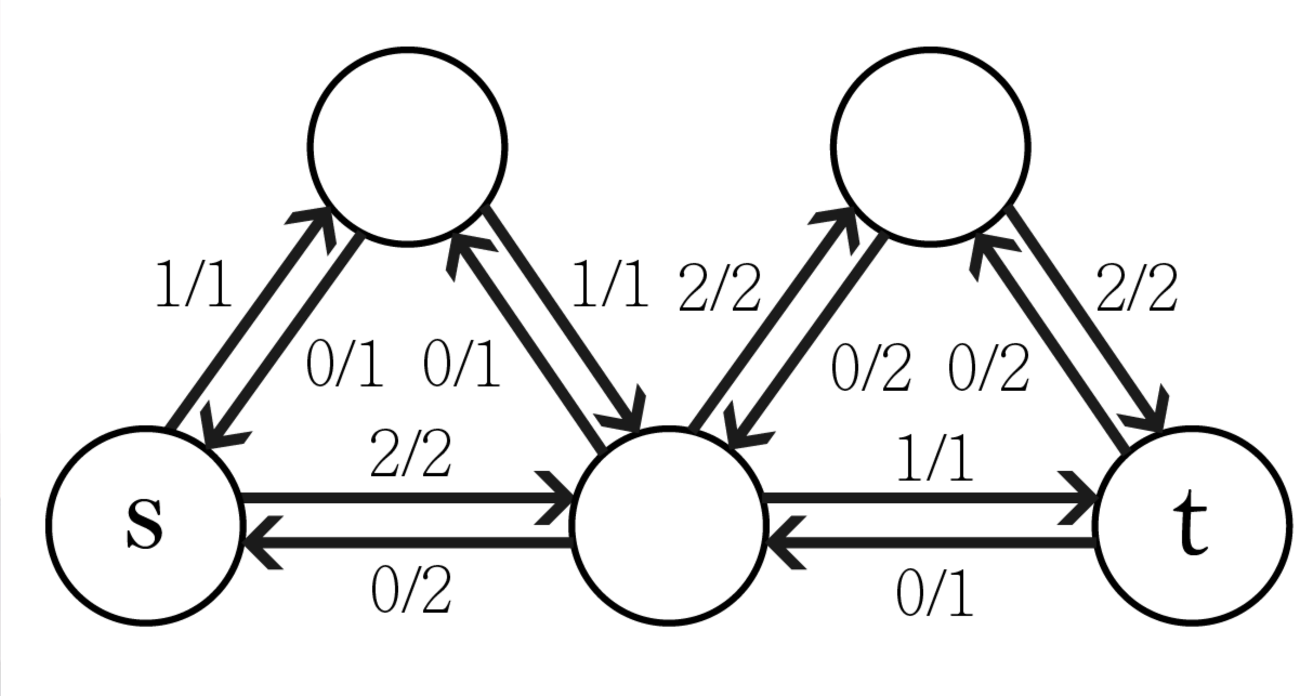
Edmonds-Karp

- 繼續找尋最短的增廣路
- 總流量增加 1



Edmonds-Karp

- 發現沒有增廣路了，因此最大流為 3



Edmonds-Karp 時間複雜度

- 設 $\delta_f(s,x)$ 為增廣**前**的剩餘網路中，源點到 x 的最短距離。
- 令 v 是在某次增廣**後** $\delta_f(s,v)$ 變小的點中距離源點最近的點
- 設 $\delta_{f'}(s,x)$ 為增廣**後**的剩餘網路中，源點到 x 的最短距離。則可以得到 $\delta_{f'}(s,v) < \delta_f(s,v)$
- 令 u 是在增廣**後**的剩餘網路中，從源點到 v 之最短路徑的前一個節點，則 $\delta_{f'}(s,v) = \delta_{f'}(s,u) + 1$

Edmonds-Karp 時間複雜度

- 又因為我們選擇 v 的方式，因此

$$\delta_f(s,u) \leq \delta_{f'}(s,u)$$
$$\Rightarrow \delta_f(s,u)+1 \leq \delta_{f'}(s,u)+1 = \delta_{f'}(s,v) < \delta_f(s,v)$$
$$\Rightarrow \delta_f(s,u)+1 < \delta_f(s,v)$$

- 也就是說 (u,v) 邊沒有剩餘流量，因為如果 (u,v) 邊還有剩餘流量的話代表 $\delta_f(s,v) \leq \delta_f(s,u)+1$

Edmonds-Karp 時間複雜度

- (u,v) 邊在擴充前沒有剩餘流量，但擴充後有剩餘流量，代表在這次增廣時有通過 (v,u) 邊，所以 $\delta_f(s,v)+1=\delta_f(s,u)$ ，但是這與 $\delta_f(s,u)+1<\delta_f(s,v)$ 矛盾，因此不存在這樣的 v 點

$\Rightarrow$ 最短增廣路的距離非遞減

Edmonds-Karp 時間複雜度

- 令 (u, v) 邊為某次增廣中路徑上容量最低的邊，則增廣後 (u, v) 會消失於剩餘網路，若之後某次增廣後，(u, v) 又出現於剩餘網路上，代表該次增廣時有通過 (v, u)
- 使 (u, v) 消失的那次增廣中，$\delta_f(s, v) = \delta_f(s, u) + 1$
  使 (u, v) 再次出現的增廣中，$\delta_{f'}(s, v) + 1 = \delta_{f'}(s, u)$
- 由於 $\delta_{f'}(s, v) \geq \delta_f(s, v)$，因此

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2$$

$$\delta_{f'}(s, u) \geq \delta_f(s, u) + 2$$

Edmonds-Karp 時間複雜度

- 最短的增廣路長度至多為 $n-1$ ，因此一條邊成為該路徑上容量最小的邊，其次數不超過 $O(n)$ 次
- 每次增廣至少會有一條邊為容量最少的邊，由剩餘網路定義可知，剩餘網路的邊數小於等於兩倍的邊數
- 至多每條邊都被增廣 $O(n)$ 次，增廣次數最多 $O(nm)$ 次
- 增廣一次需耗時 $O(m)$ ，總複雜度

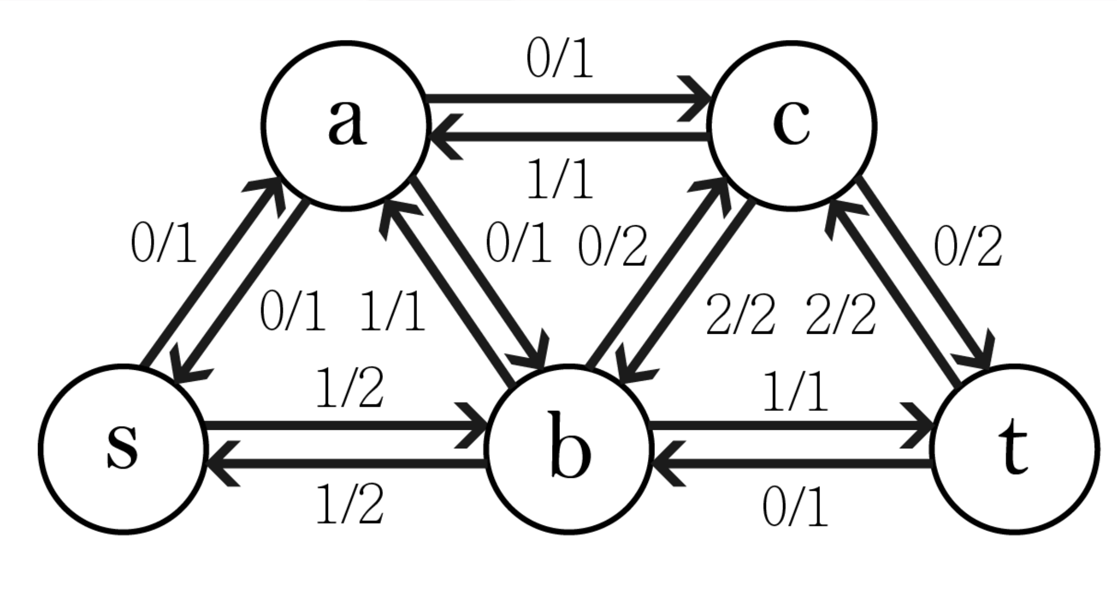
$$O(m \times nm) = O(nm^2)$$

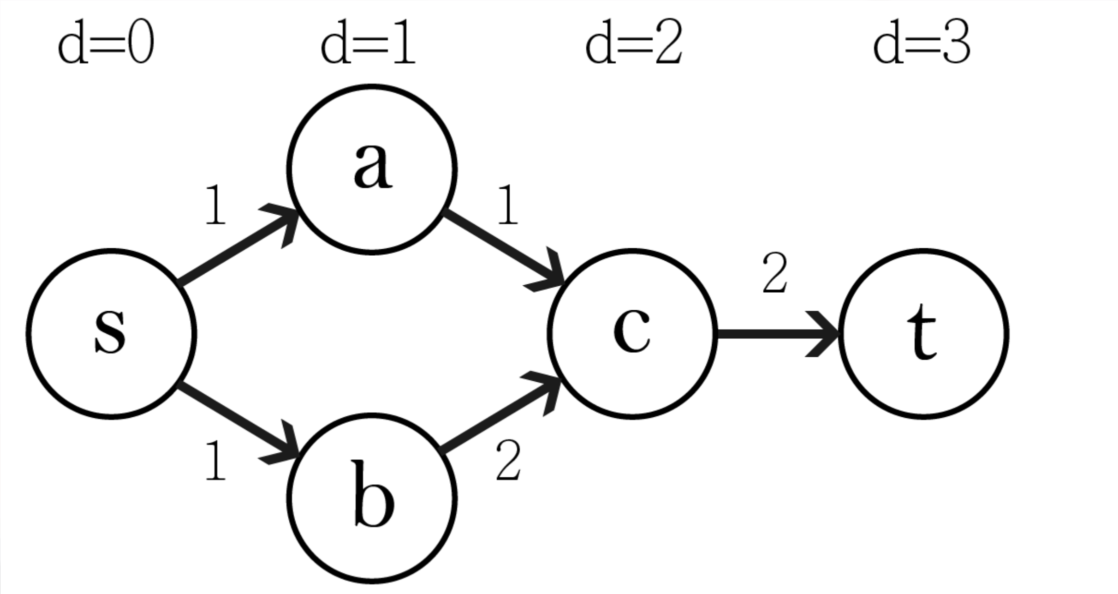
Dinic

- 如果每次擴充時將所有距離為 k 的增廣路全部找出來呢？
- 距離為 k 之增廣路上的頂點一定是從 1 慢慢累加到 k ，那麼只要保留那些距離差是 1 的邊，這些邊所構成的增廣路必為 k
- 這些距離差為 1 的邊所構成的圖稱為層次圖(level graph)

Dinic

- 剩餘網路轉換為層次圖， d 代表從源點 s 到達該點的距離



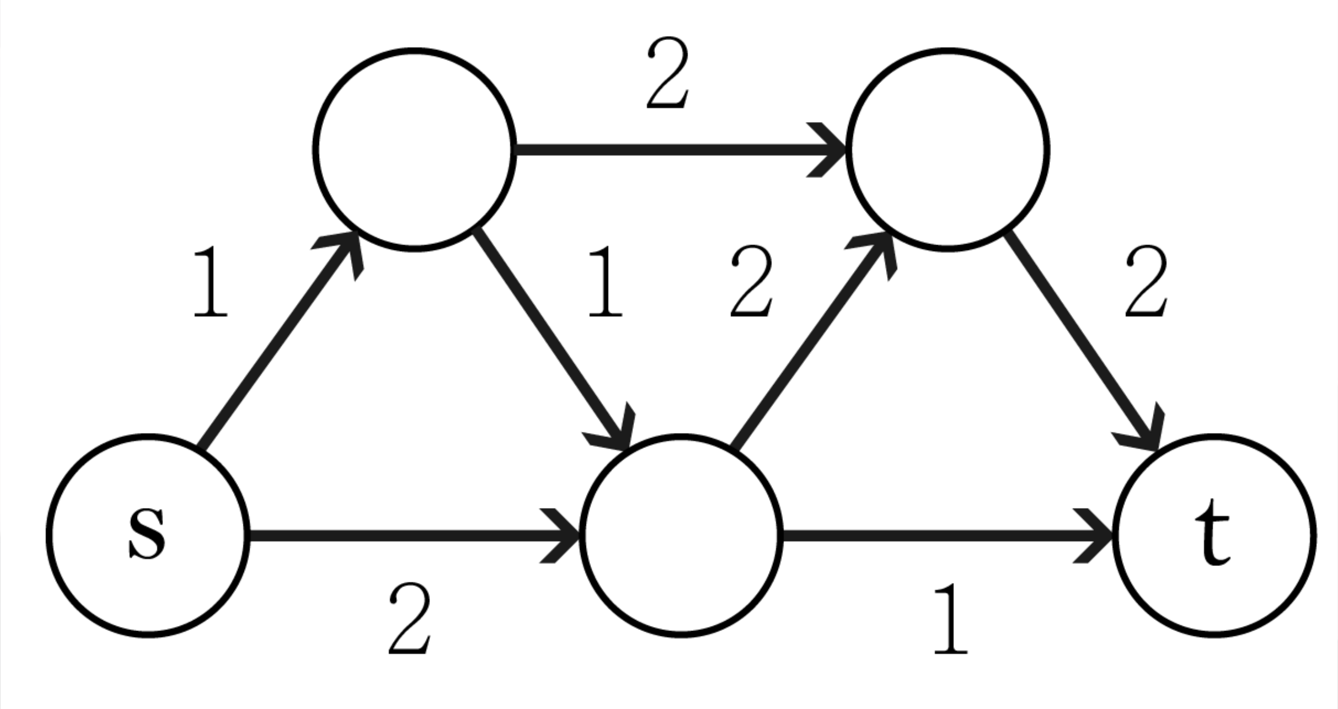


Dinic

- 每次先建構出源點 s 到匯點 t 的層次圖
- 將所有最短的增廣路進行增廣
- 重複以上動作直到沒有增廣路為止

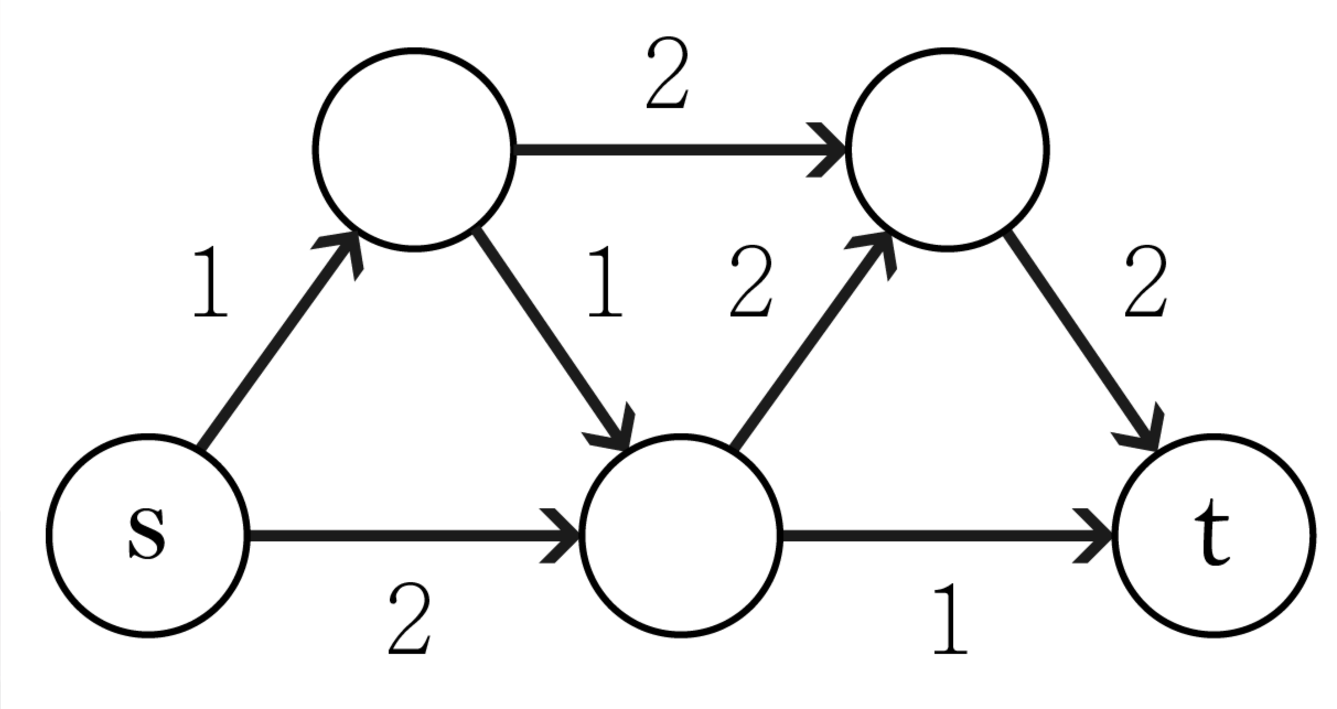
Dinic

- 以下圖為例



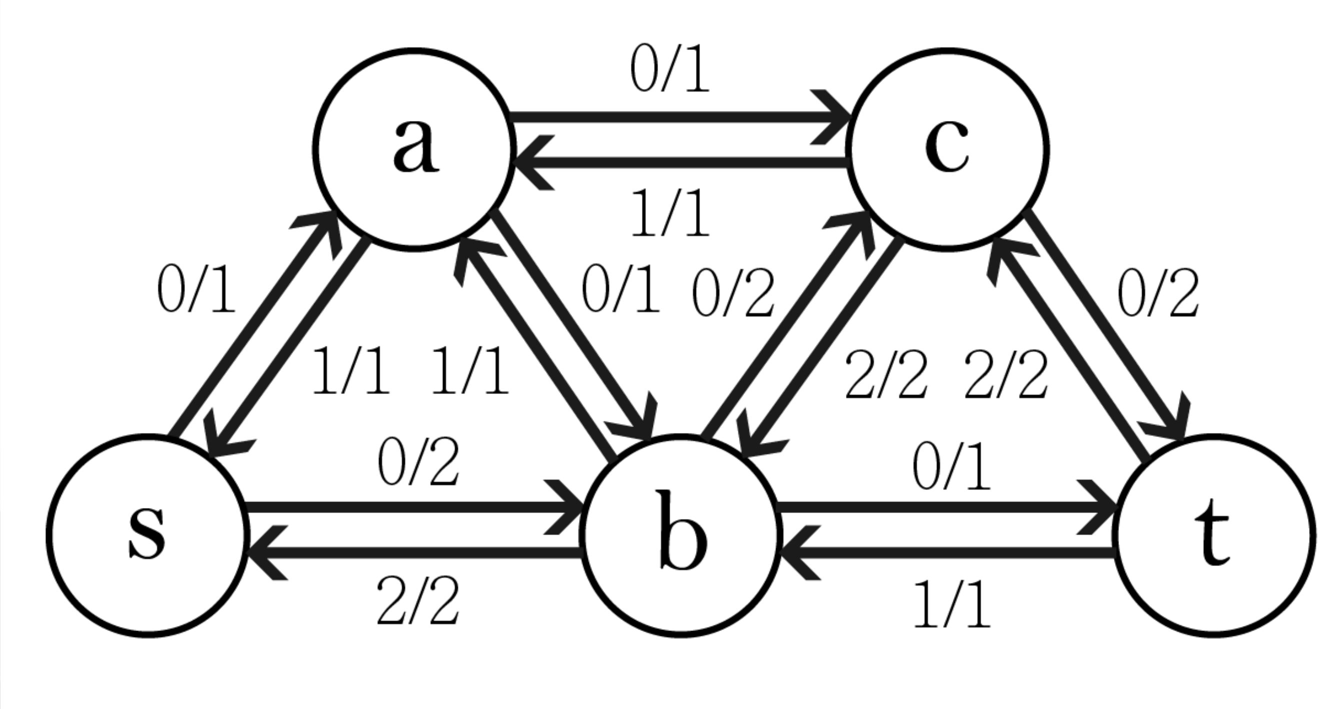
Dinic

- 轉換為剩餘網路



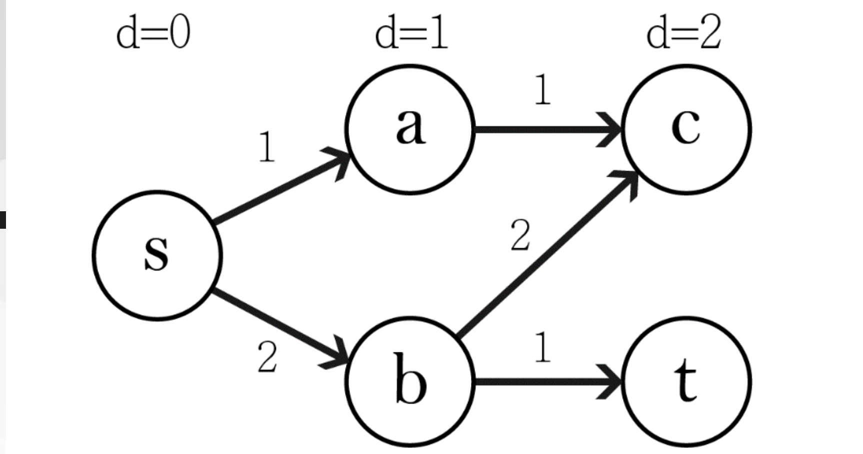
Dinic

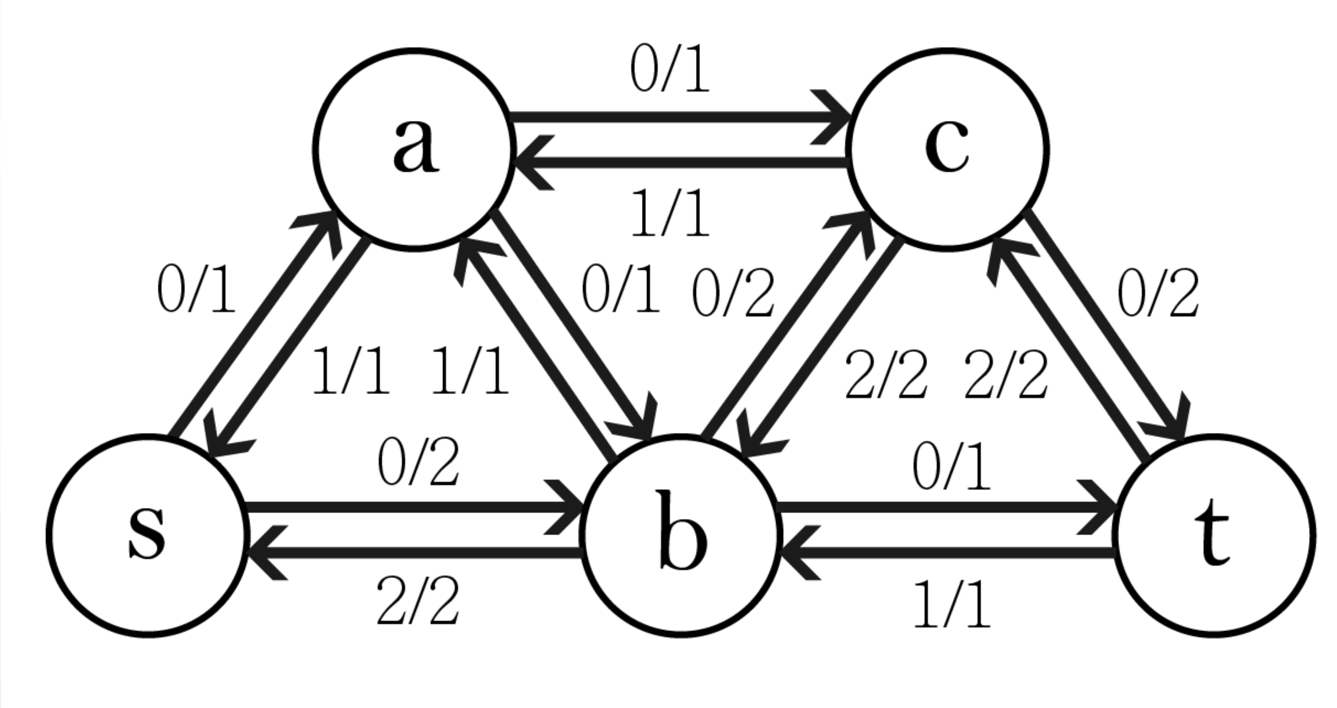
- 轉換為剩餘網路



Dinic

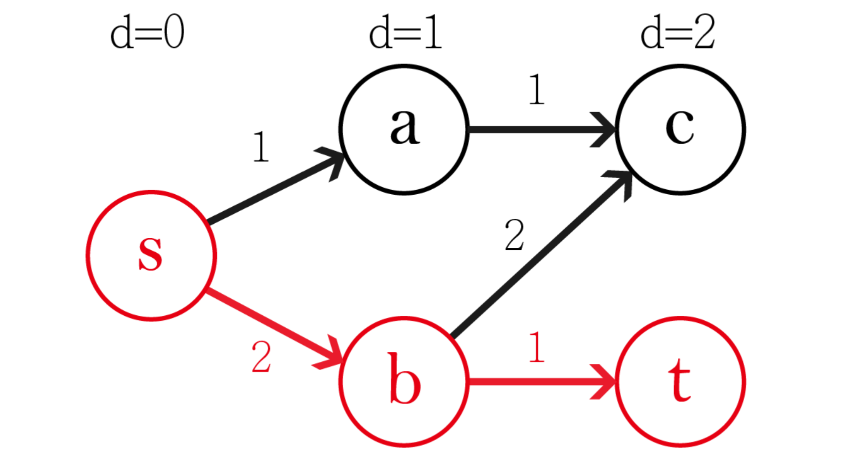
- 建構層次圖

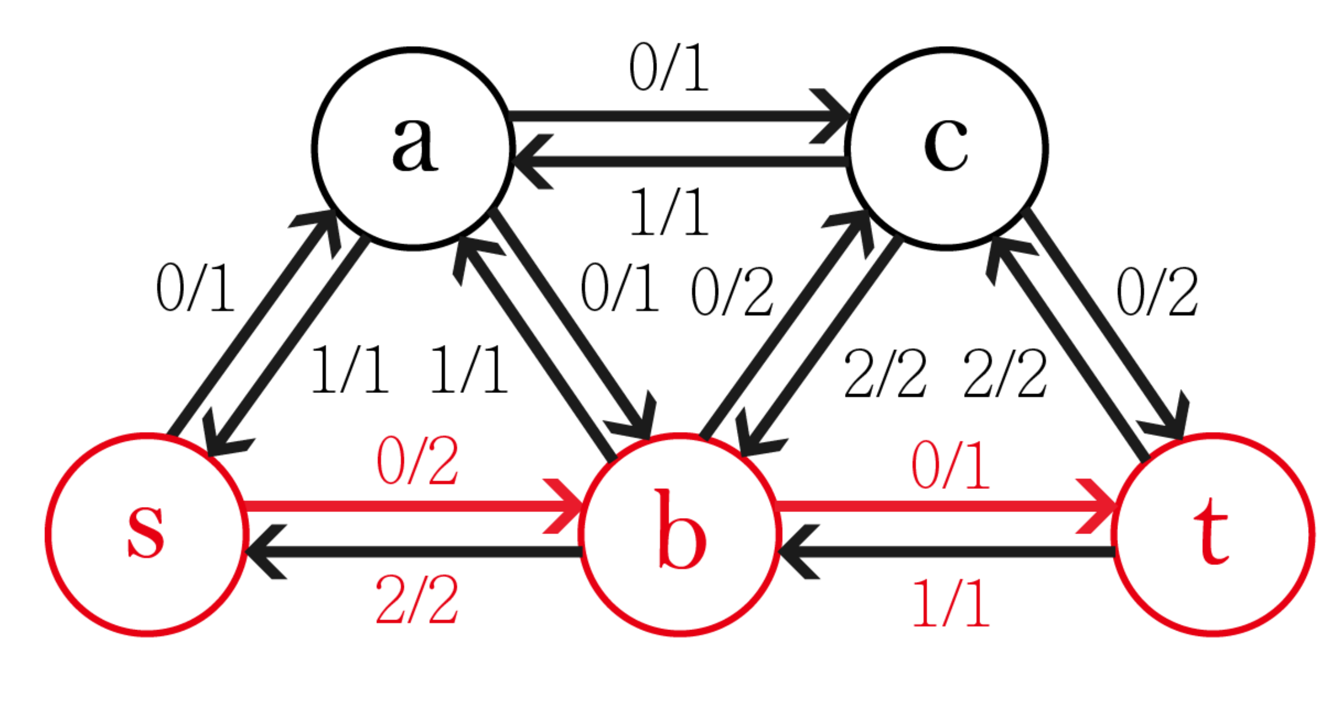




Dinic

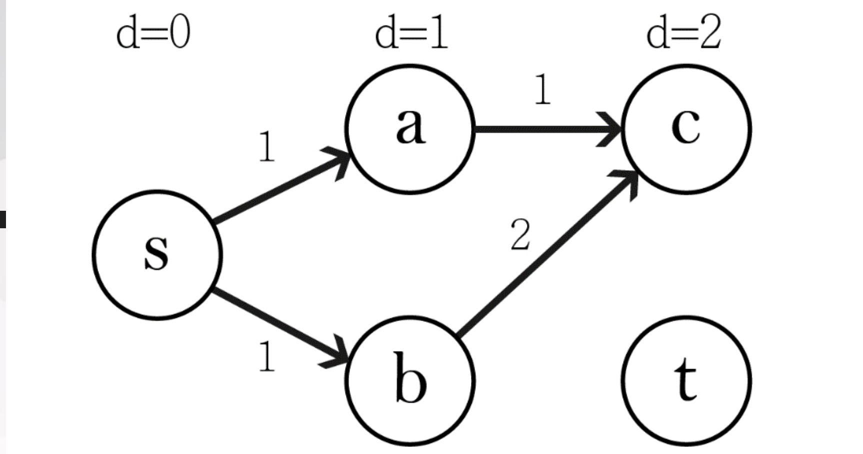
- 將層次圖所有 s 到 t 的增廣路進行增廣

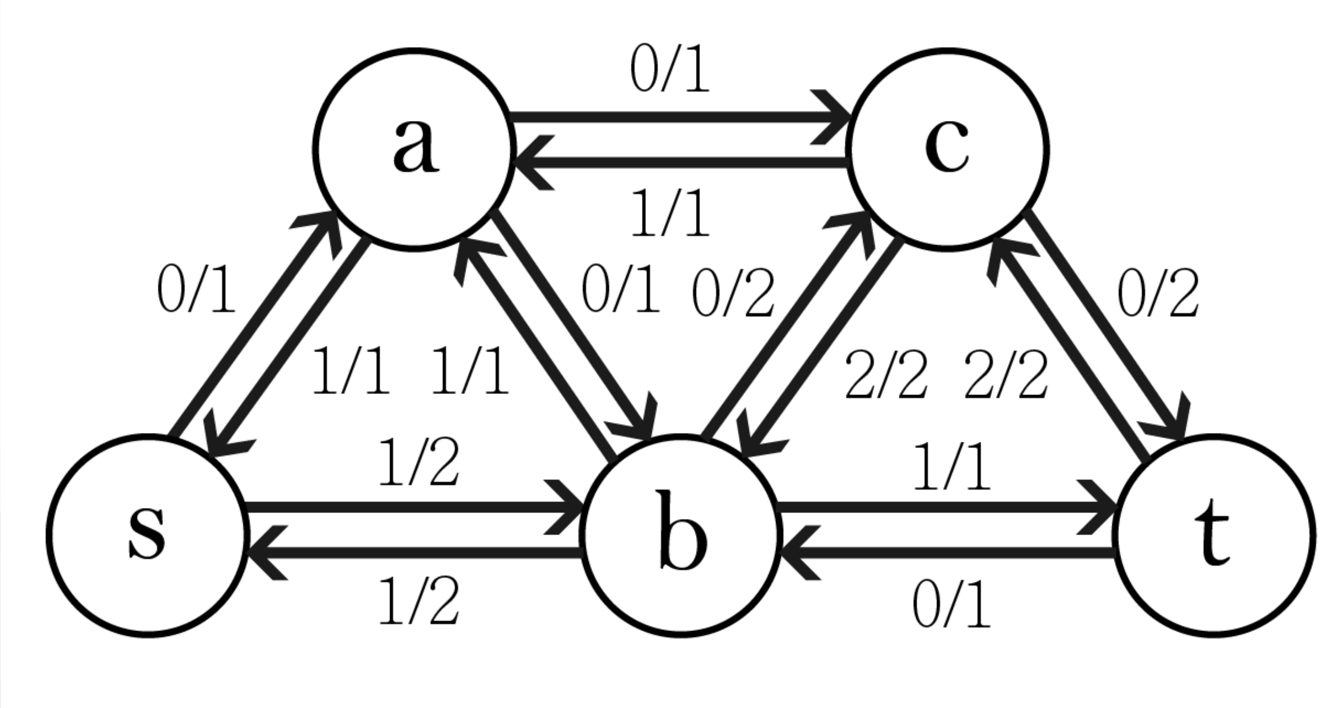




Dinic

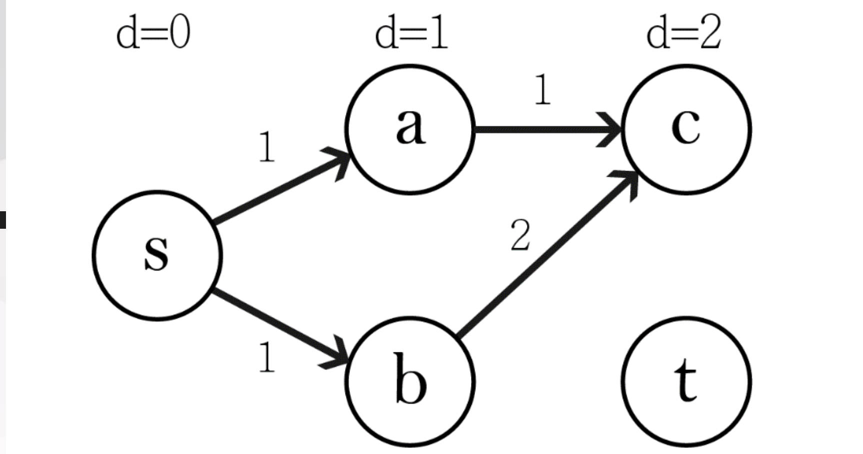
- 將層次圖所有 s 到 t 的增廣路進行增廣

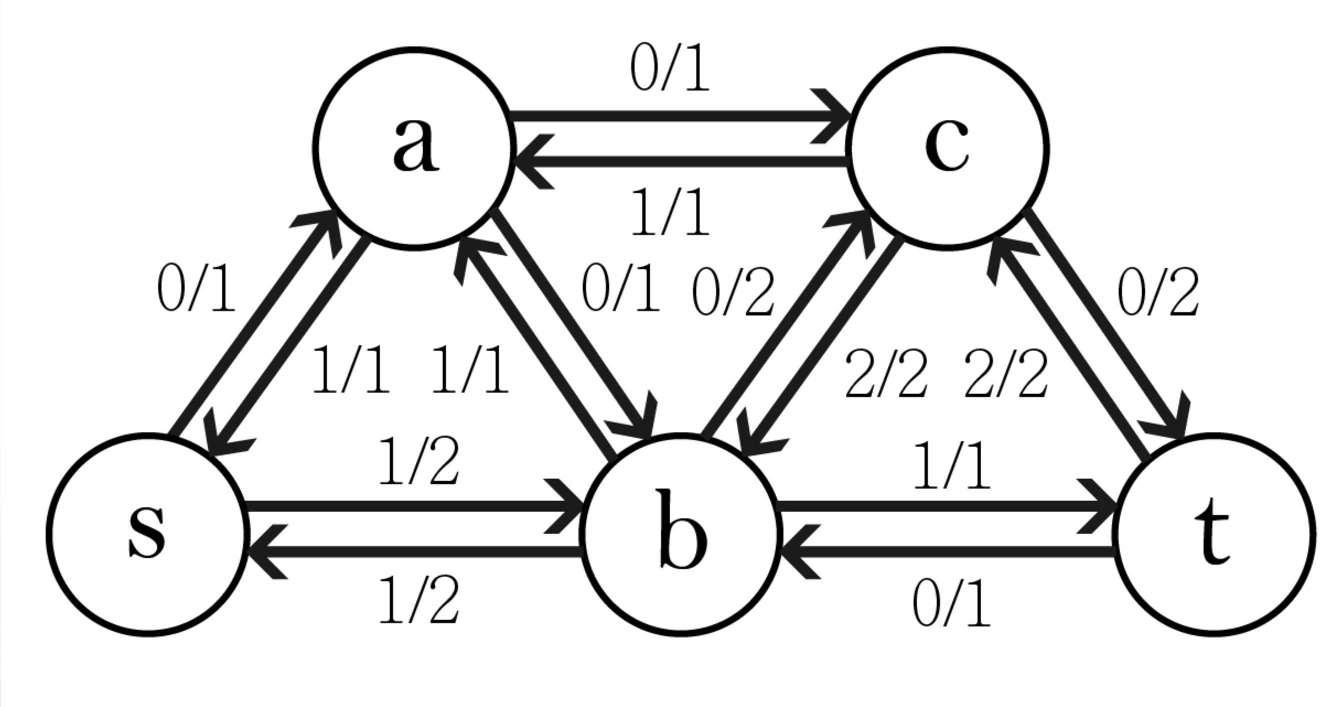




Dinic

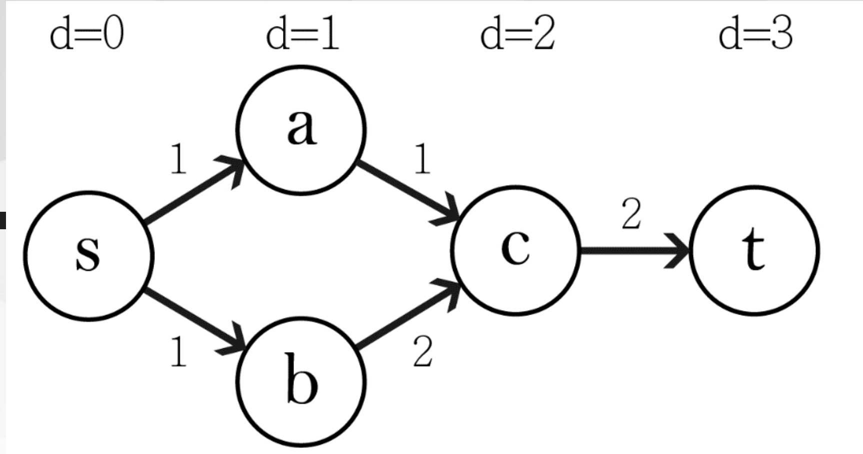
- 發現層次圖上沒有增廣路了

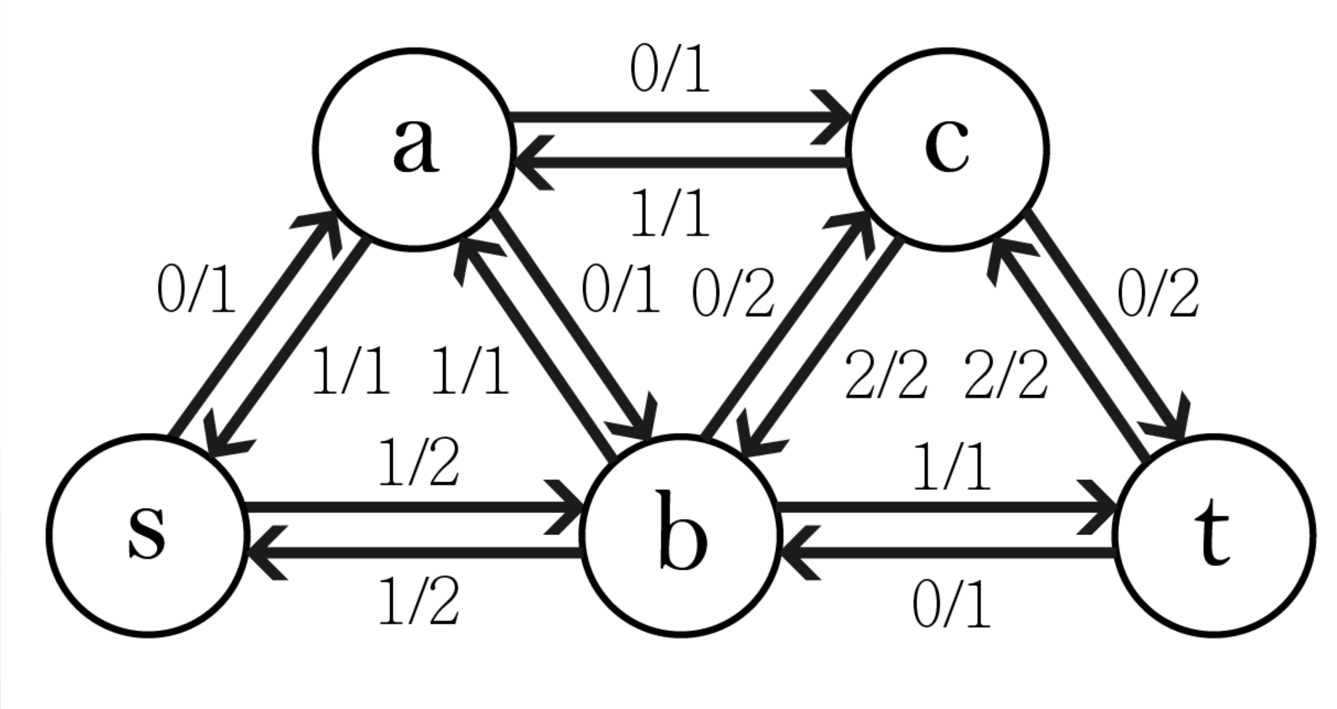




Dinic

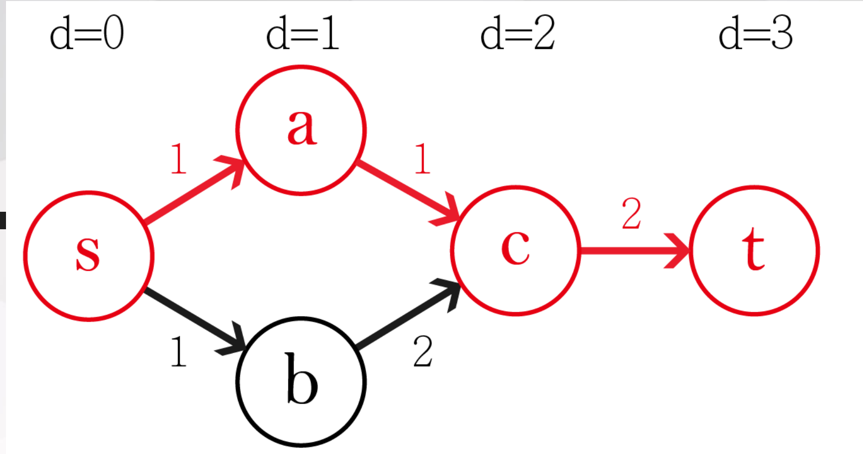
- 再次建構層次圖

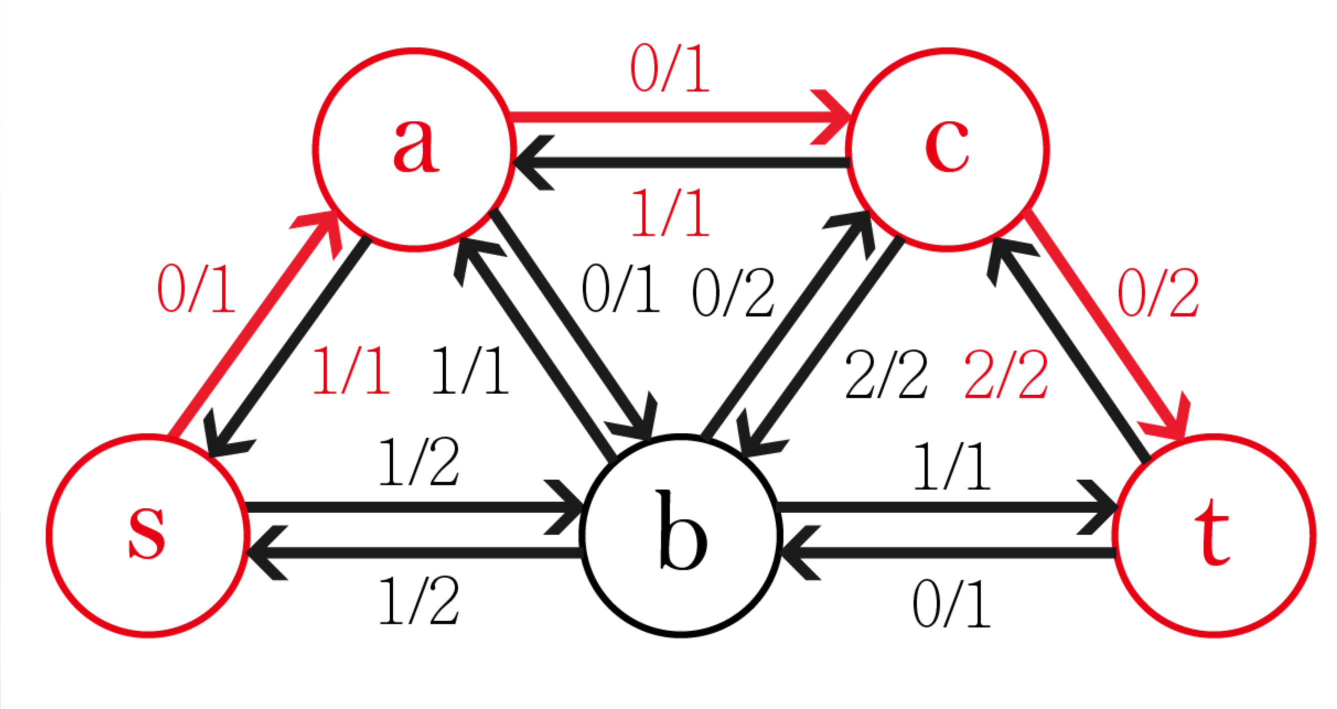




Dinic

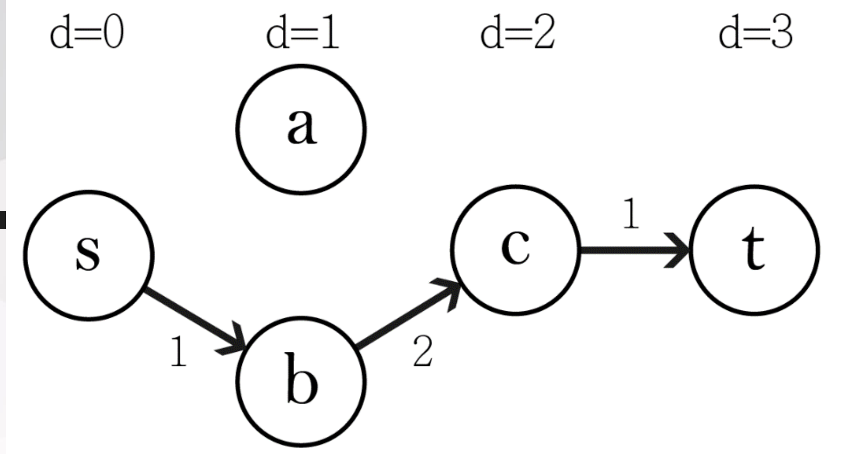
- 將層次圖所有 s 到 t 的增廣路進行增廣

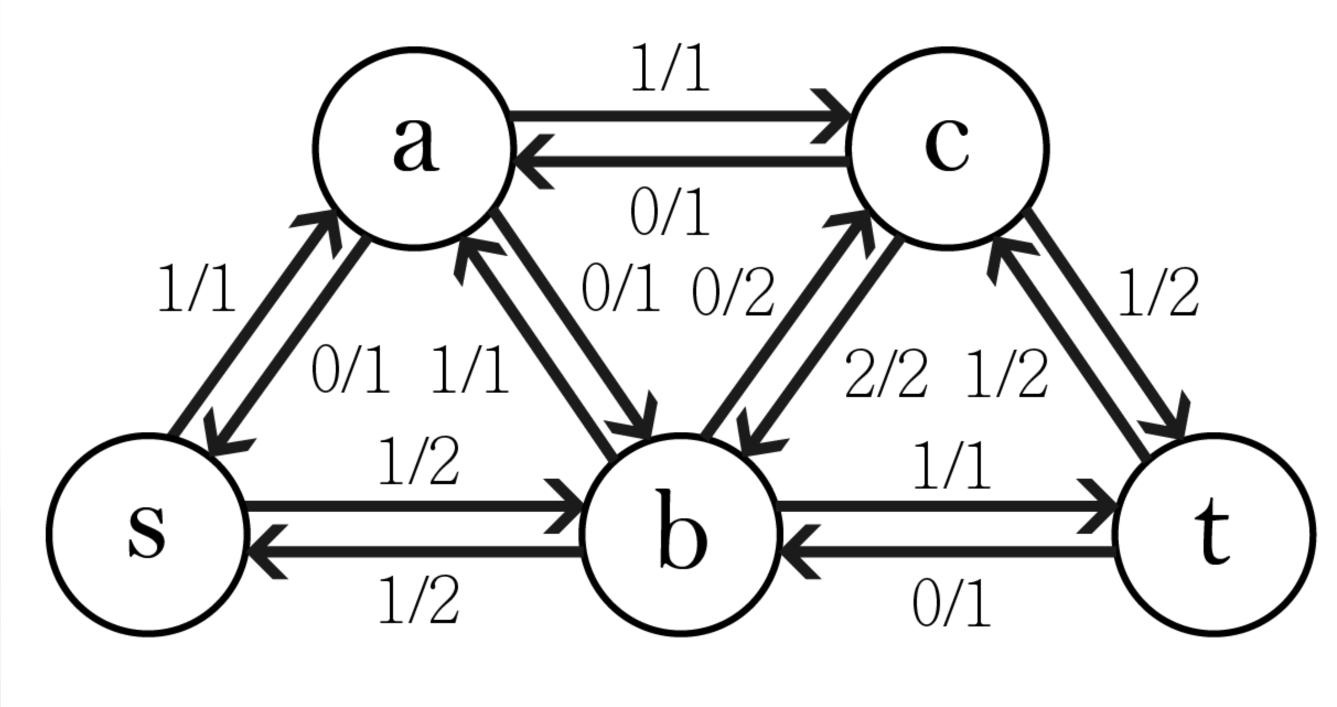




Dinic

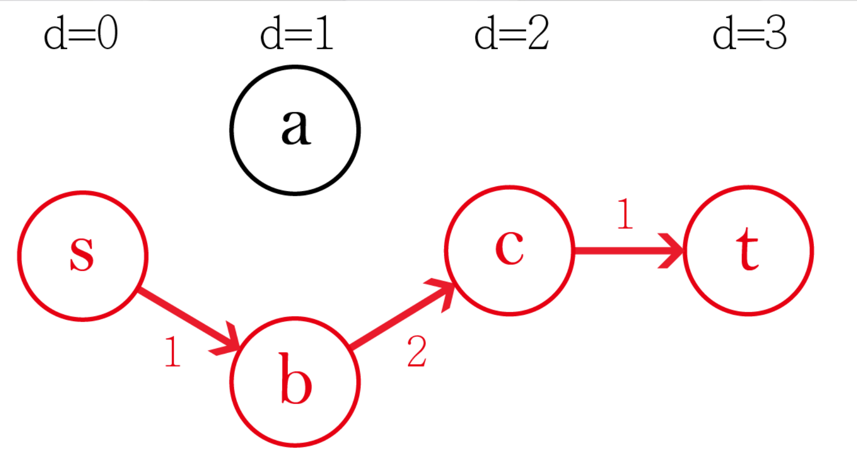
- 將層次圖所有 s 到 t 的增廣路進行增廣

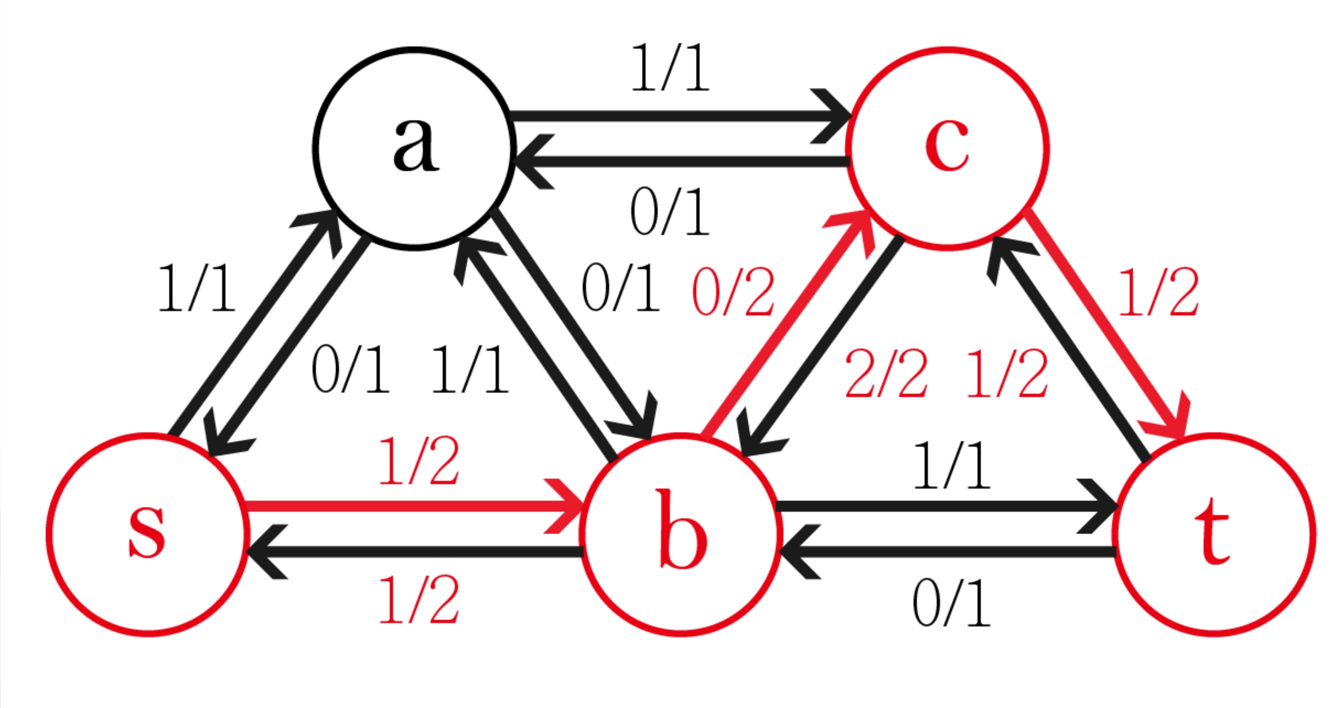




Dinic

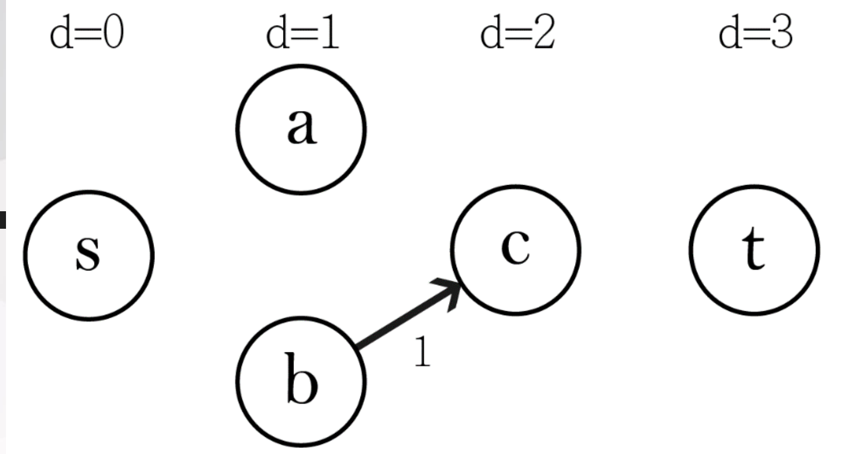
- 將層次圖所有 s 到 t 的增廣路進行增廣

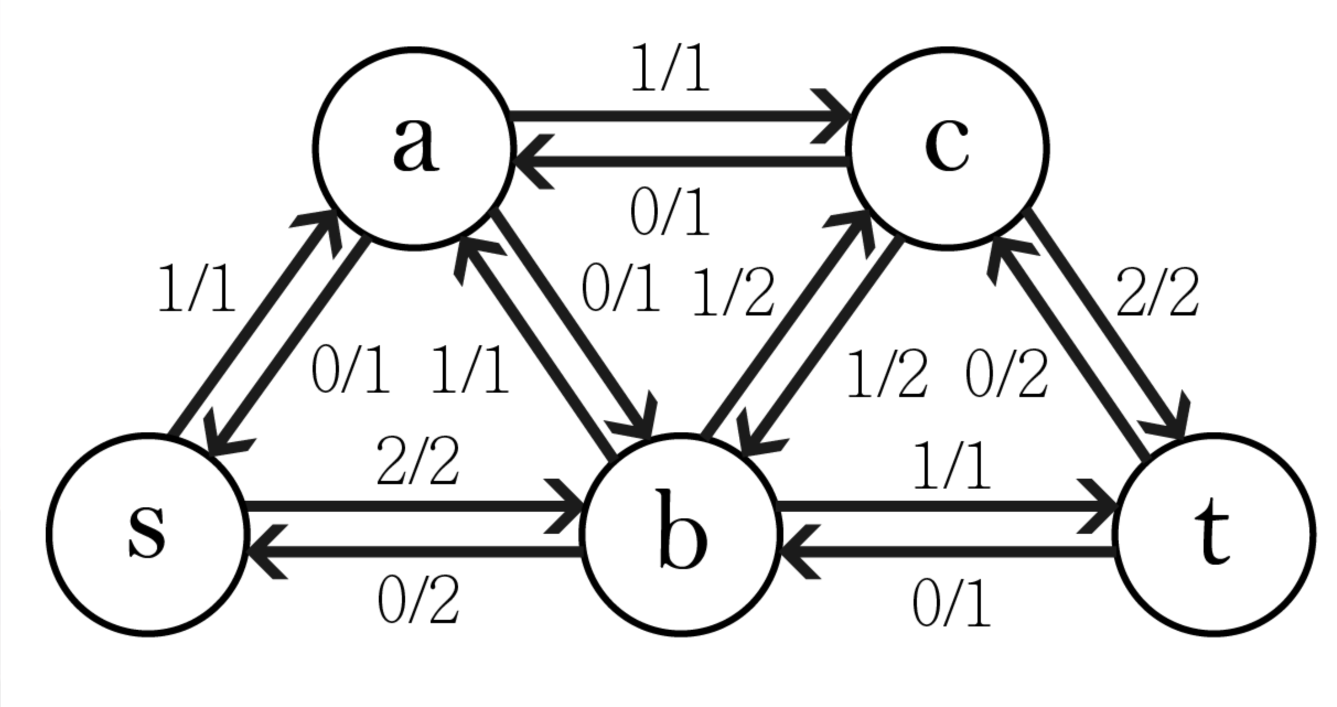




Dinic

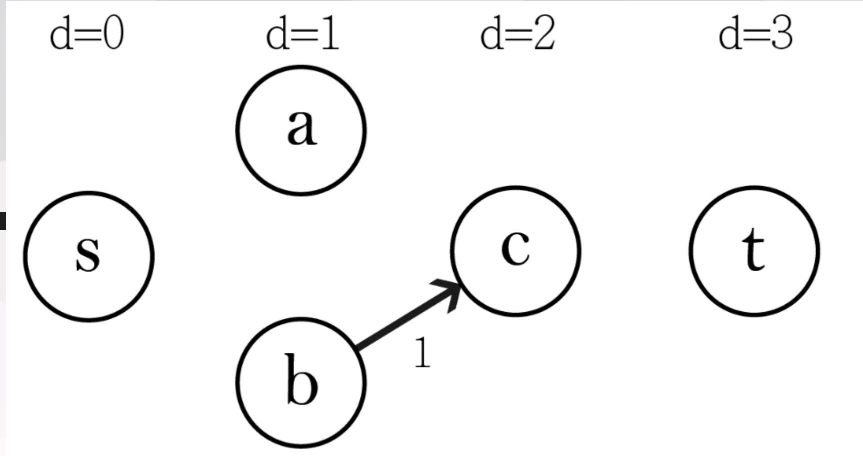
- 將層次圖所有 s 到 t 的增廣路進行增廣

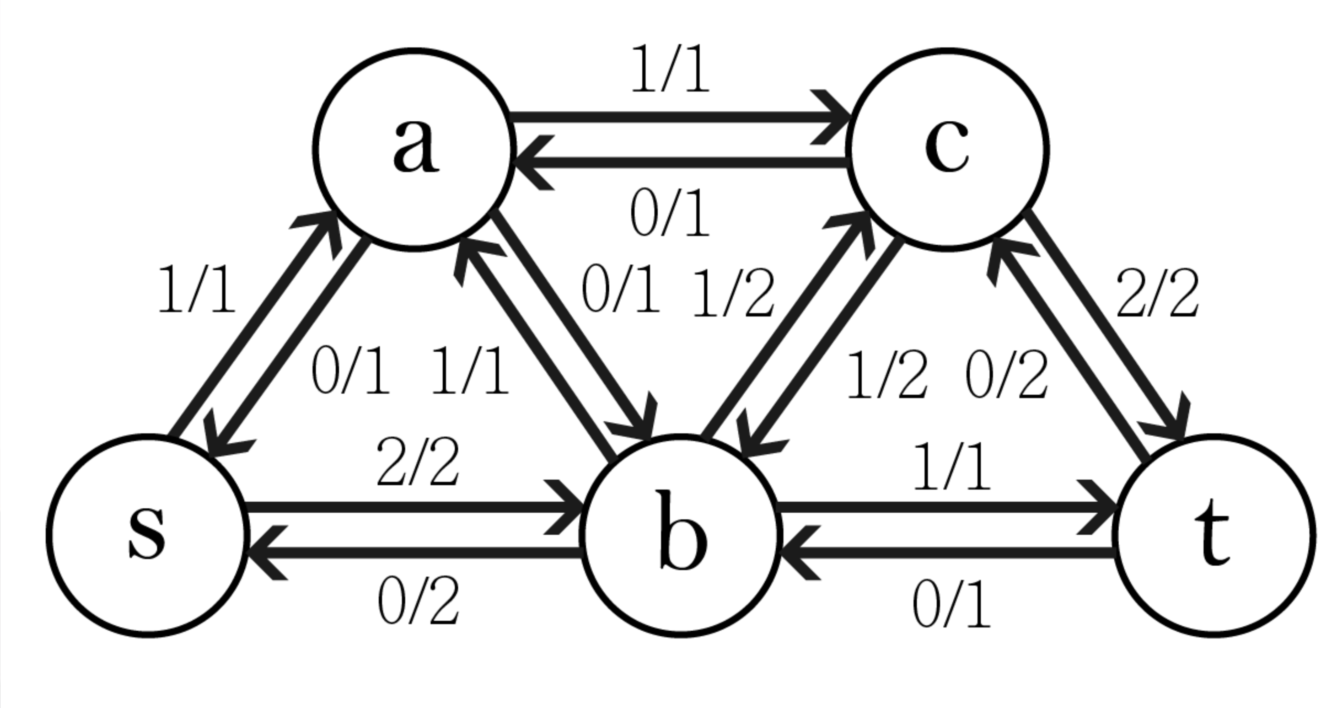




Dinic

- 發現層次圖上沒有增廣路了，且剩餘網路也不存在從源點走到匯點的路徑，因此演算法到此結束，
  得到最大流為 3





Dinic 時間複雜度

- 層次圖距離落在 $0 \sim n-1$ ，而增廣後的最短距離一定會大於增廣前，因此最多需要建構 $O(n)$ 次層次圖
- 每次增廣後至少會有一條邊「滿流」，滿流的邊會在層次圖上消失；最多消失 m 條邊，因此在層次圖上最多增廣 $O(m)$ 次，每次增廣路的長度至多 $n-1$ ，因此維護剩餘網路的複雜度為 $O(nm)$
- 總複雜度 $O(n \times nm) = O(n^2m)$
- 通常邊數會大於點數，因此 Dinic 的 $O(n^2m)$ 會比 Edmonds-Karp 的 $O(nm^2)$ 來的好

Dinic 時間複雜度

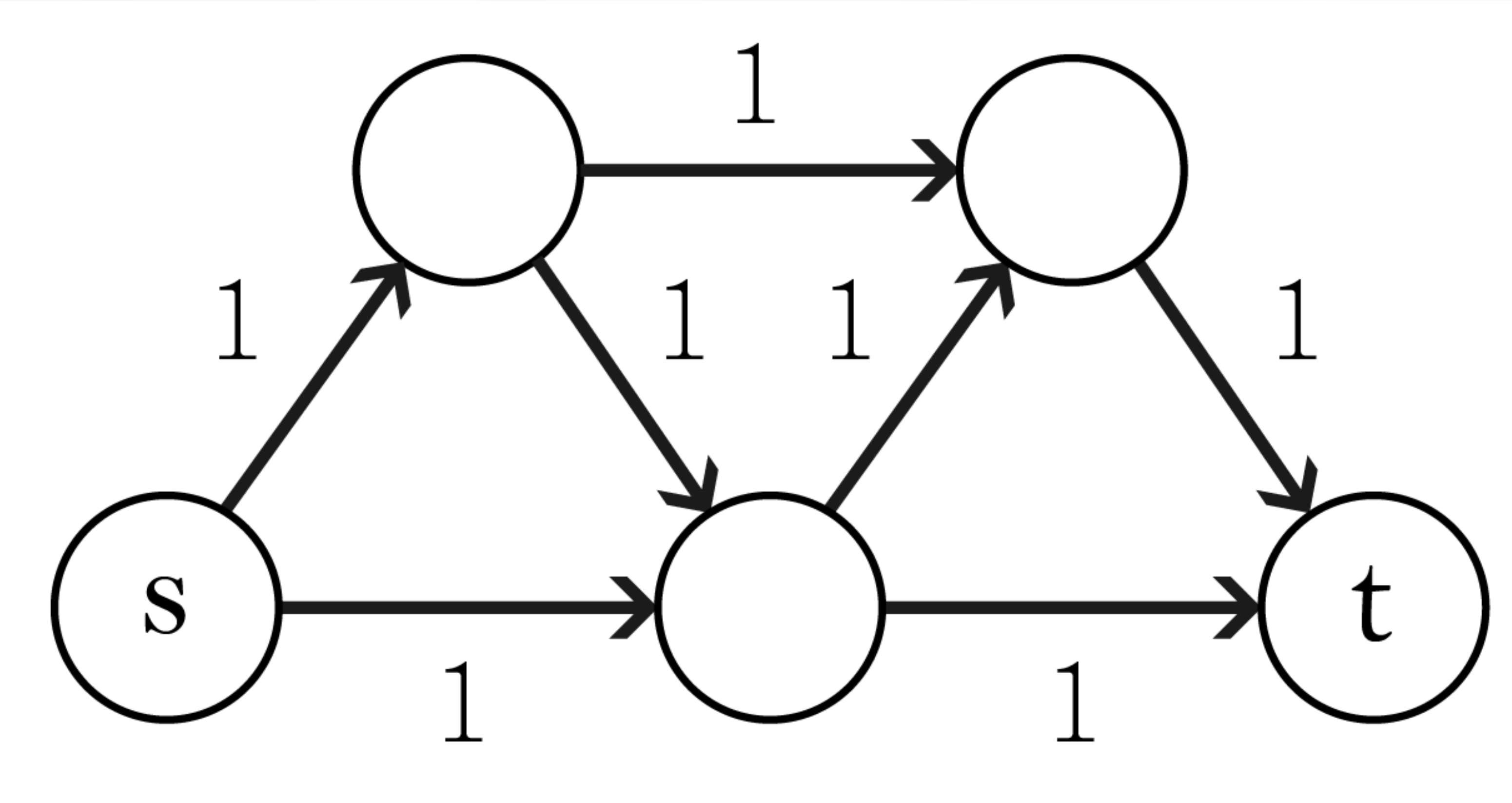
- 其實這是理論最差情況下的複雜度，實際應用上不會這麼差

- 另外如果是單位容量網路，也就是每條邊的容量都是 1 時，Dinic 的時間複雜度會是 $O(\min(n^{2/3}, m^{1/2})m)$

- 若在二分圖匹配的網路上，則 Dinic 的時間複雜度會是 $O(m\sqrt{n})$

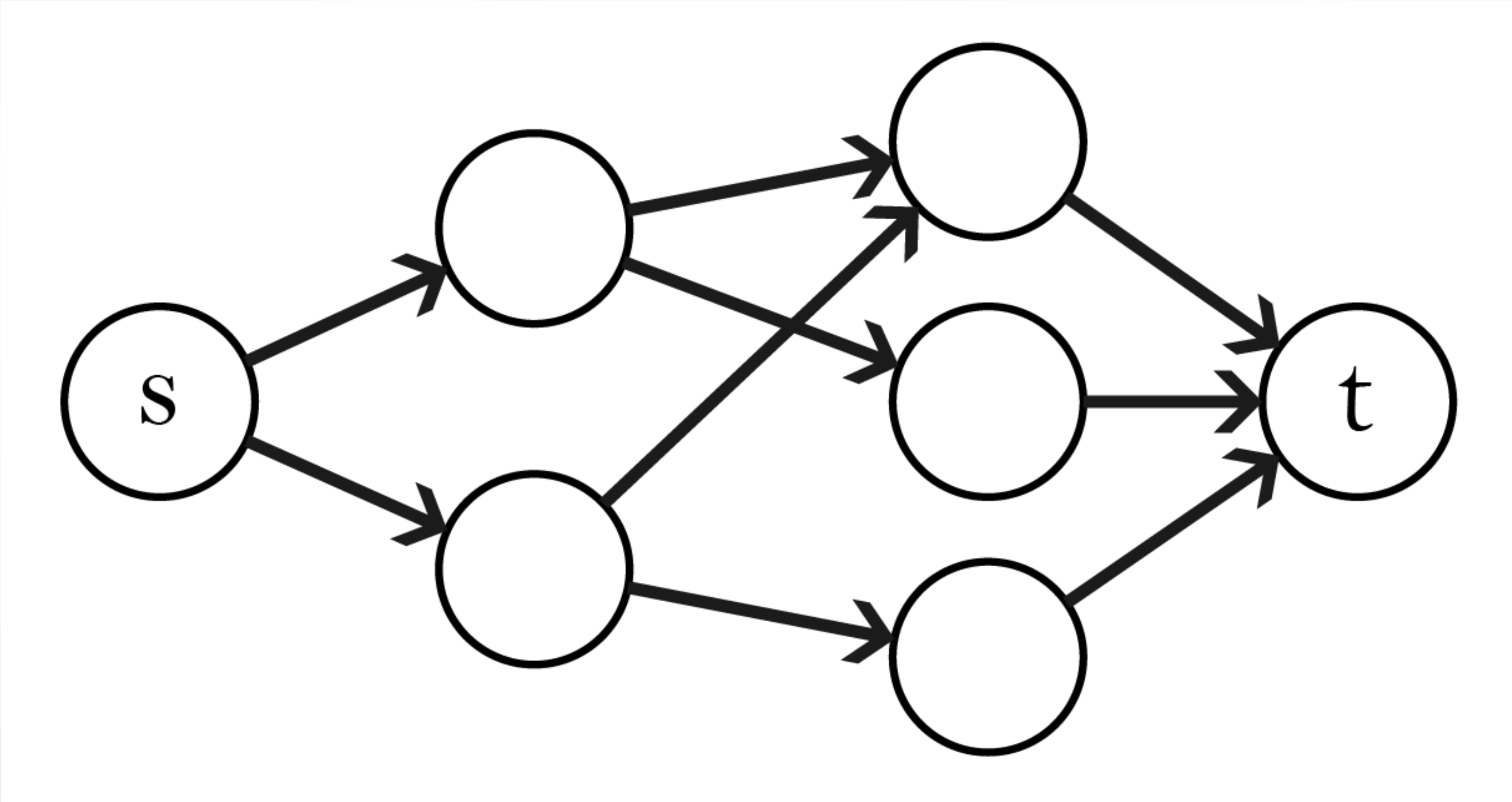
Dinic 時間複雜度

- 單位容量網路



Dinic 時間複雜度

- 二分圖匹配網路



Dinic 模板

- 因為 Dinic 效率最高，因此被許多人拿來當作網路流算法的模板，這邊也提供一份
- https://paste.ofcode.org/aisT6ShY95ewgpw5JYQPAW

Min Cut

割

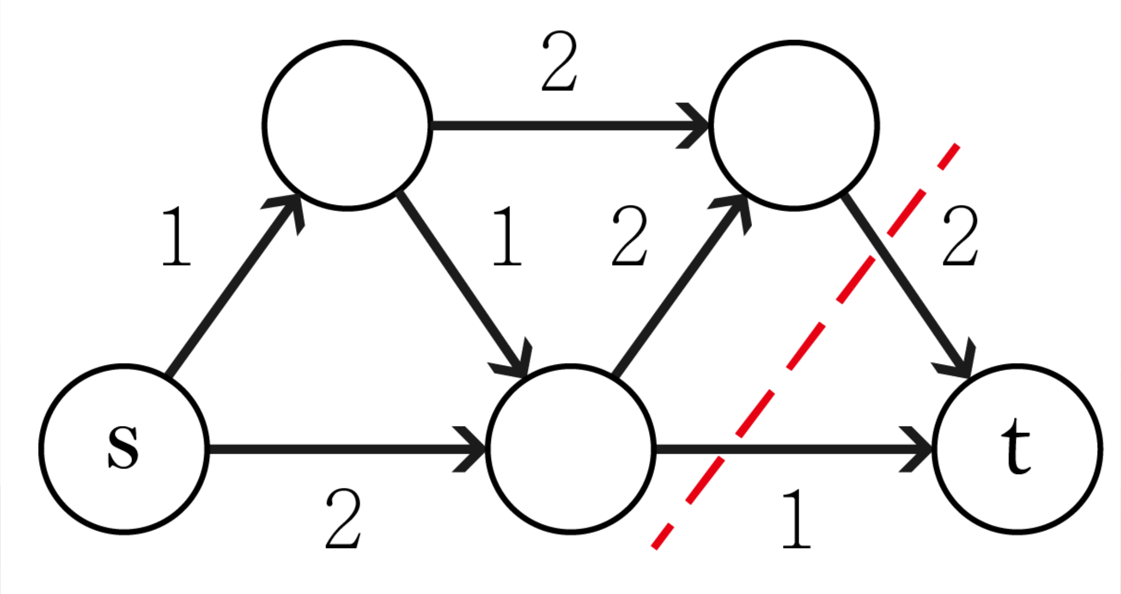
- 對於一張帶權圖中，一個割代表將圖 G 中所有的點分成兩個不相交點集 S 與 S'
- 對於一張 n 個點的圖有向圖，有 $2^n - 2$ 種割集
- 對於一張 n 個點的圖無向圖，有 $(2^n - 2)/2$ 種割集

s-t 割

- 網路流中，一個 s-t 割 $C(s,t)$ 是對於圖 G 中的所有點，將其劃分為兩個點集 S 與 T 且 $s \in S, t \in T$
- 則 $C(s,t) = \{(u,v) \in G \mid u \in S, v \in T\}$ ，也就是跨點集的邊所構成的集合，請注意這邊只計算從 S 跨到 T 的邊

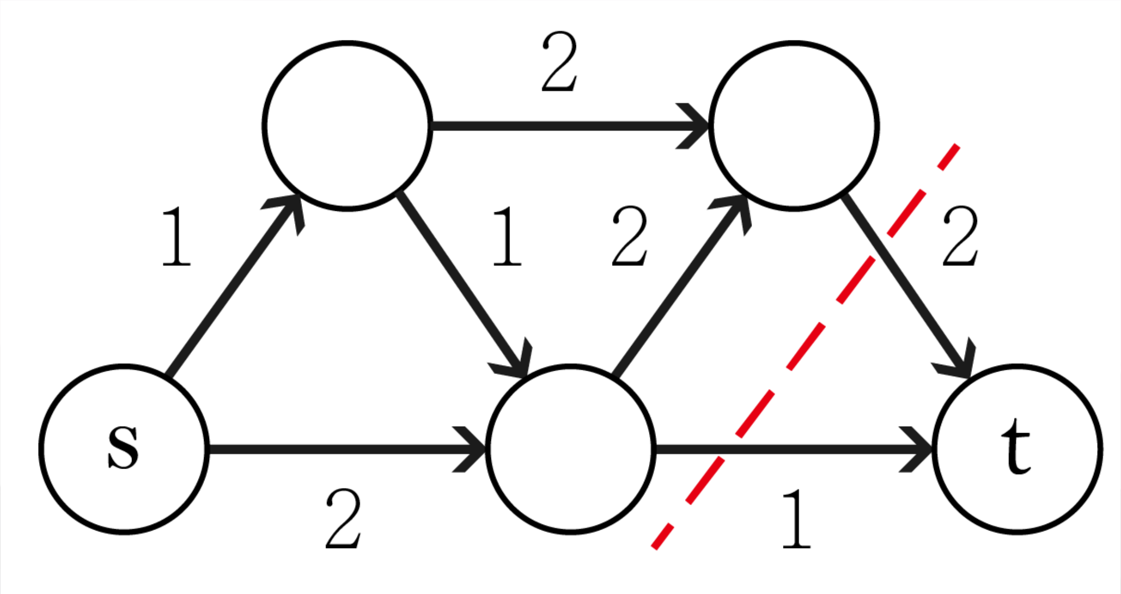
割的花費

- 一個割的花費是割集中所有的邊權和，網路流中割的花費就是割集中所有邊的容量和
- 如下圖中割的花費為 $2 + 1 = 3$



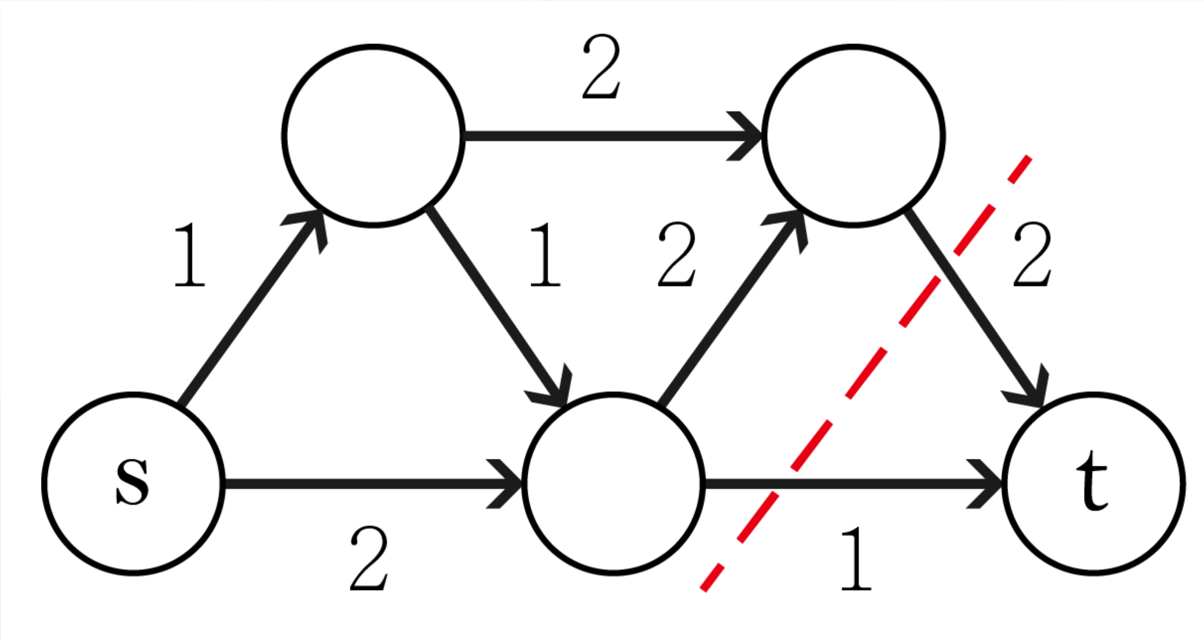
最小割

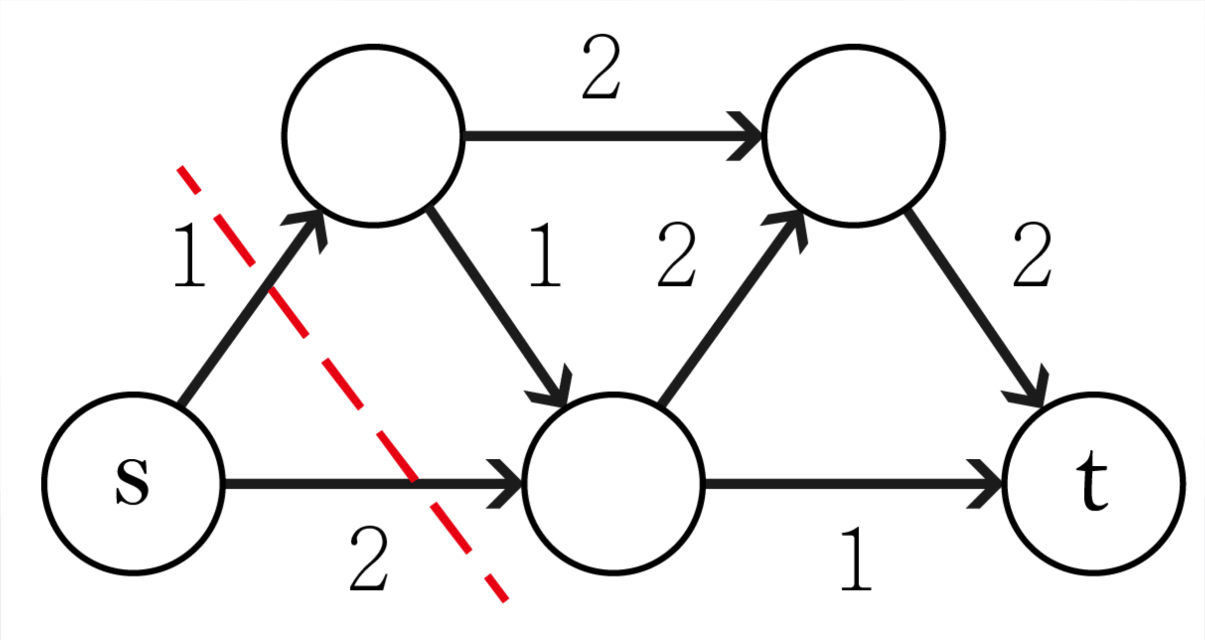
- 最小割顧名思義就是在所有割集中花費最小的割
- 下圖就是一個 s-t 最小割，往後在講最小割都是最小 s-t 割



最小割

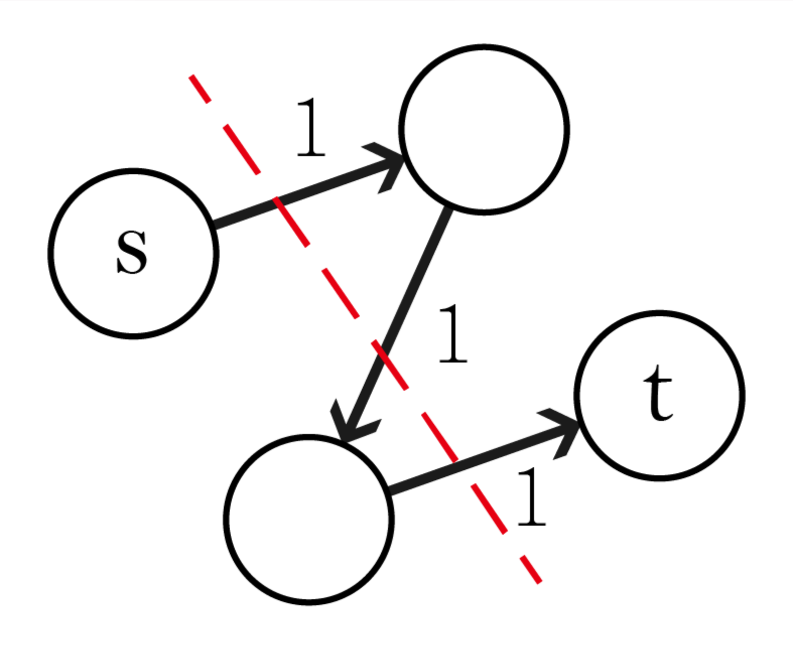
- 最小割不唯一，下面兩張圖同時是最小割





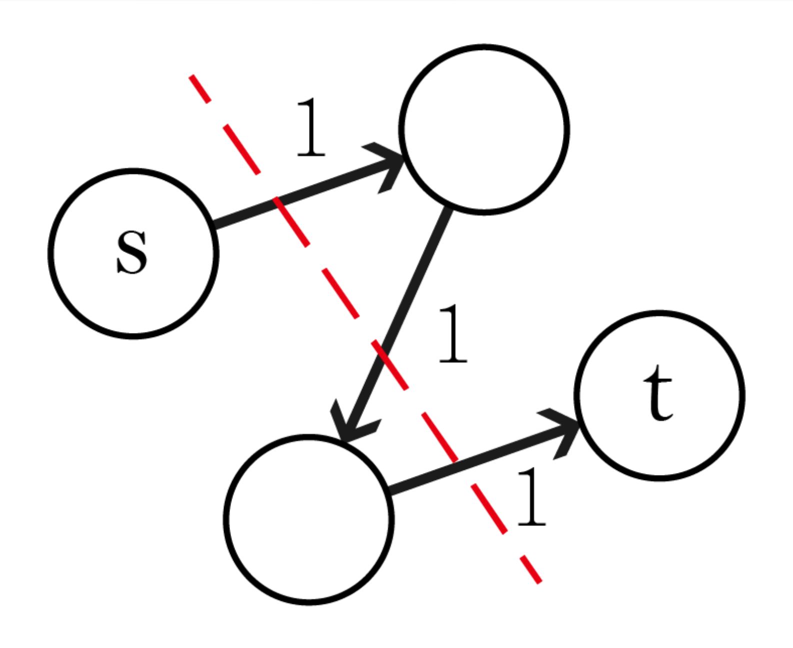
流-割對偶性

- 流-割對偶性屬於線性規劃的問題，因此我們用簡單一點的方式來說
- 以下用 S 表示源點這邊的割，用 T 表示匯點這邊的割
- 一張圖中的總流量等於 S 到 T 的流量減掉 T 到 S 的流量
- 如右圖的總流量是 $(1+1) - 1 = 1$



流-割對偶性

- 由於水流動時要符合容量限制，因此在任何割中，由 S 到 T 的總流量不會大於由 S 到 T 的總容量；反之由 T 到 S 的總流量不會大於由 T 到 S 的總容量
- 如果找到了 S 到 T 的總容量和最小的分割方法，則找到 s-t 的最小割

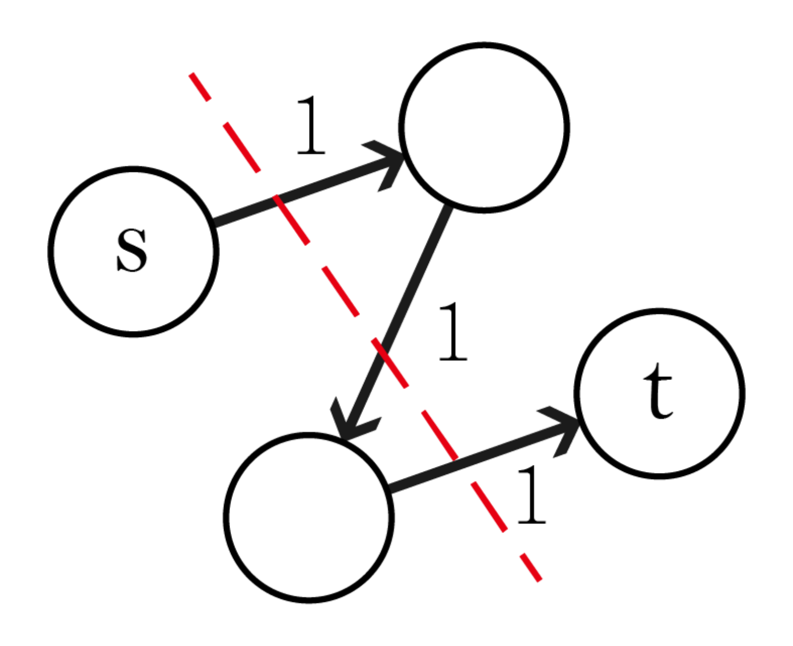


最大流最小割定理

- 如果容量大於流量，則水流可以繼續增加而不會超過限制
- 當無法繼續增加流量時，一定會有一些邊滿流，也就是整個網路的瓶頸
- 因此一個網路流的流量就是所有瓶頸的總容量，而總流量的瓶頸會出現在由 S 到 T 總容量最小的一種分割方式

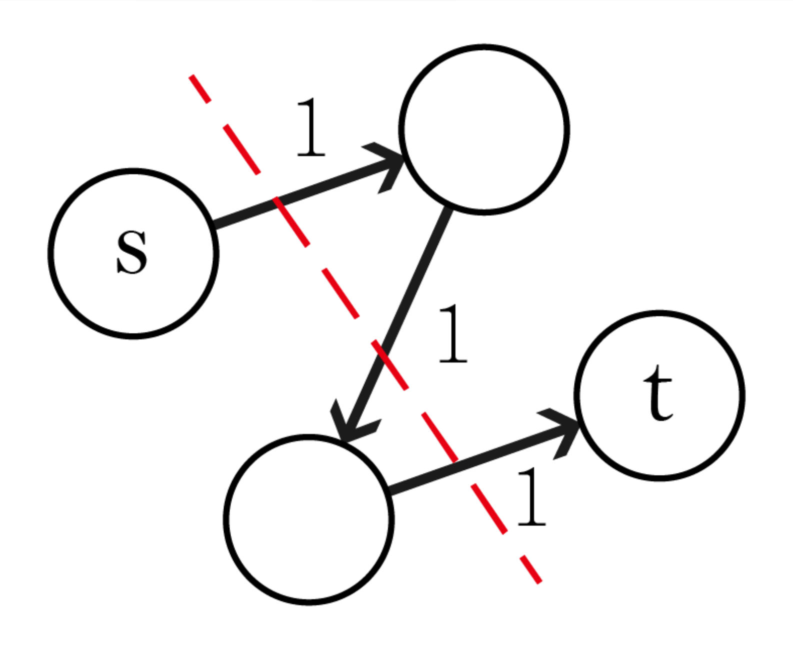
⇒ 最小割

⇒ 最大流流量 = 最小割花費



最大流最小割定理

- 你會發現，增廣路算法的中止條件是，在剩餘網路中源點及匯點不存在連通的路徑
- 如果不存在連通的路徑，代表網路中的瓶頸均滿流了，而瓶頸總容量就是最小割，且割的花費會等於最大流的流量，因此可以證明增廣路算法是正確的



Example Problem

題目

- 大部分網路流的題目都看不出這是網路流
- 舉個例子

TIOJ 2134 魔法藥水

- 有 N 個英雄與 M 個怪物，每個英雄只能殺特定幾種怪物，且每個英雄至多殺一隻怪物。現在有 K 瓶藥水，喝一瓶能使一位英雄多殺一隻怪物，每位英雄至多喝一瓶，請問在最好的策略下最多能殺幾隻怪物？

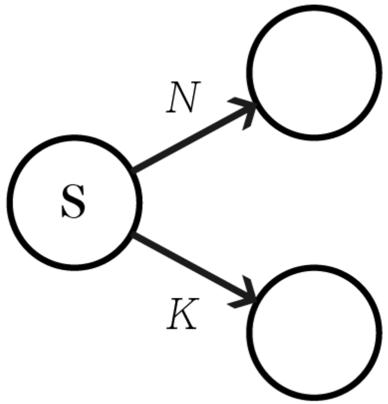
- 網路流？

建模

- 通常在網路流這類題目中，不會修改模板的 code，取而代之的會建構網路流的「模型」
- 可能是最大流的模型或是最小割的模型

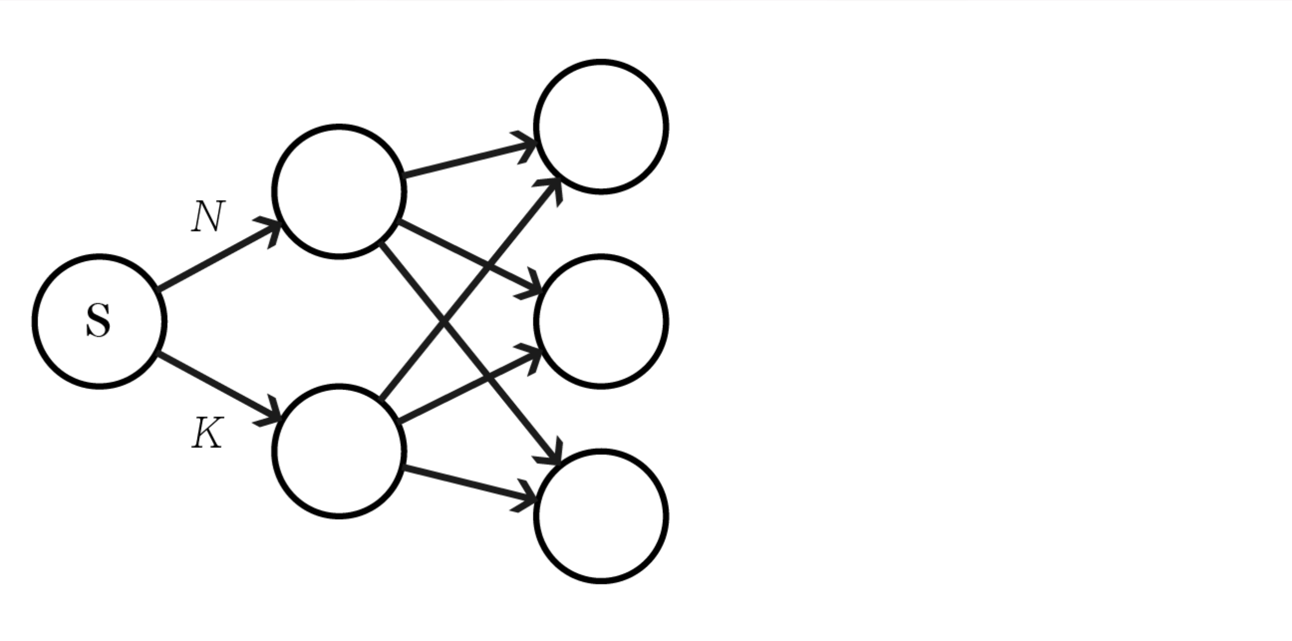
TIOJ 2134 魔法藥水

- 本題採用最大流的模型
- 先建立一個源點，並將源點指到兩個點，其容量分別是 N 與 K
- 代表 N 位英雄與 K 瓶藥水



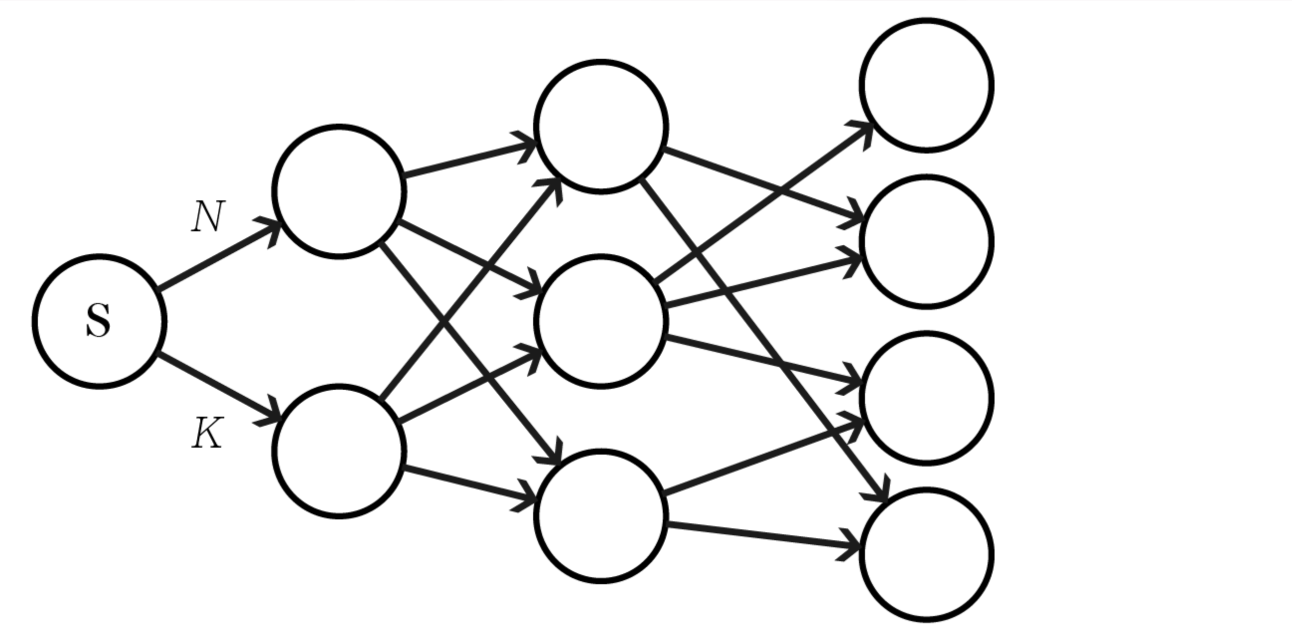
TIOJ 2134 魔法藥水

- 接著分別將這兩個點接到 N 位英雄，容量均為 1
- 分別代表每位英雄最多只能殺 1 隻怪物與喝 1 瓶藥水



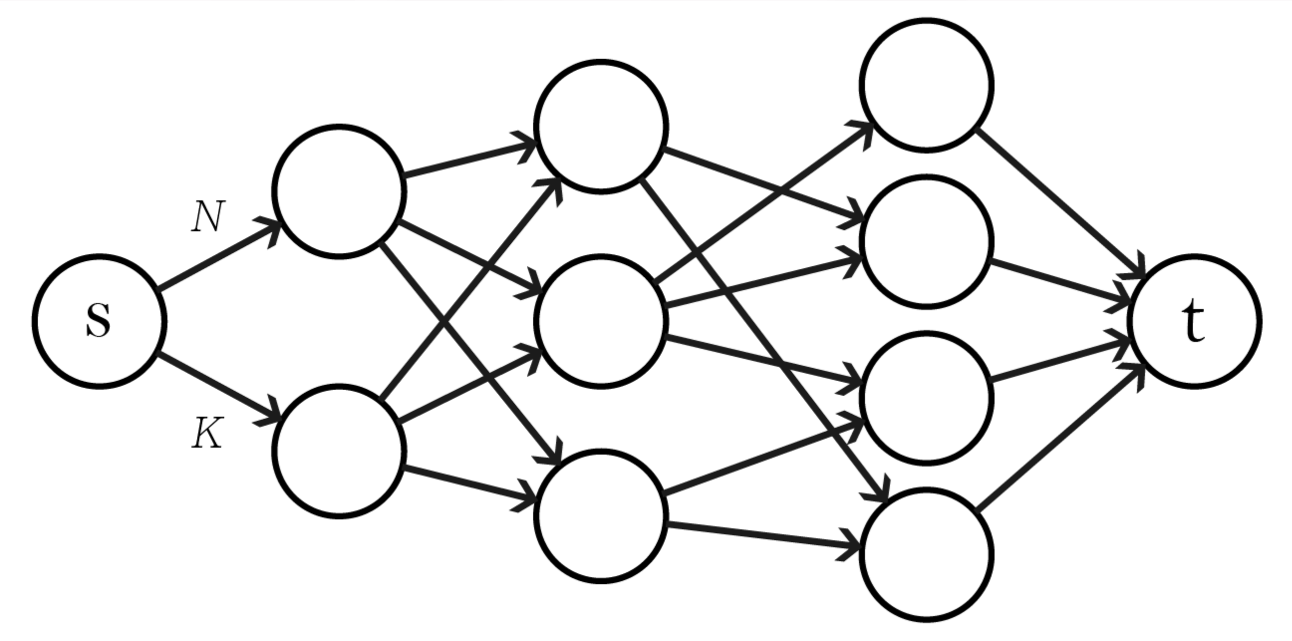
TIOJ 2134 魔法藥水

- 將每位英雄接到其能殺的怪物，容量均為 1
- 代表該英雄能殺死該怪物至多 1 次



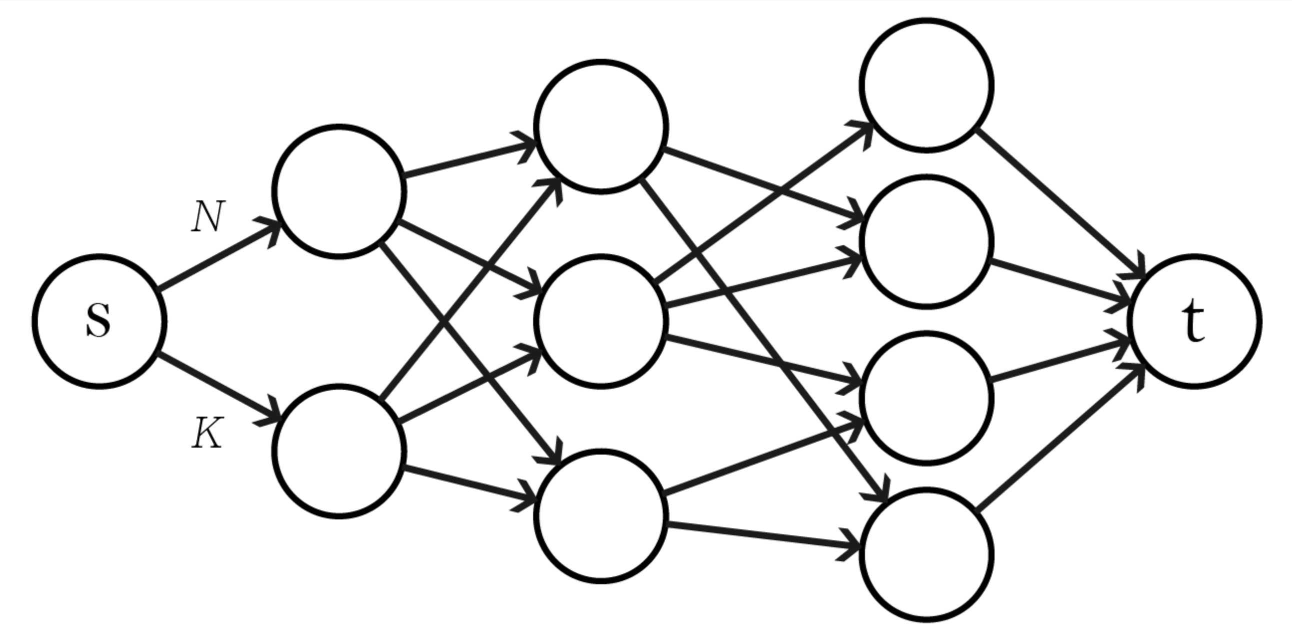
TIOJ 2134 魔法藥水

- 將所有怪物都接到匯點，容量為 1
- 代表每個怪物至多被擊殺 1 次



TIOJ 2134 魔法藥水

- 最後對這張圖跑一次最大流即為答案



Conclusion

總結

- 這兩種類型的題目常常會看不出來，只能靠多練習多寫題目來累積經驗

- 如果掌握了技巧的話，那麼看到這種類型的題目就能一眼看出來了

Questions?