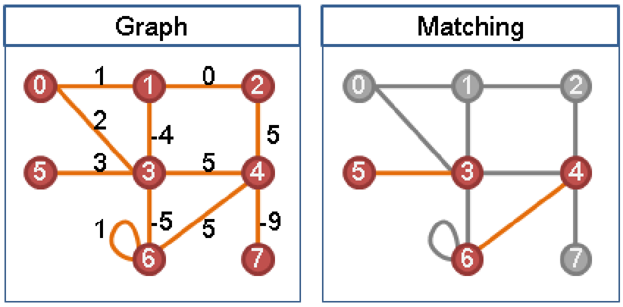
Matching

導讀

因為 Matching 的演算法有點複雜，所以我們同時介紹 Matching 和它的特例 Bipartite Matching 。每當要講解一個演算法時，就先提出 Bipartite Matching 的演算法，再進一步提出 Matching 的演算法，以循序漸進的方式進行講解。

Matching

給定一張無向圖，當圖上兩點以邊相連時，這兩點就可以配成一對 —— 但是呢，一個點最多只能與一個鄰點配成一對，寧可孤家寡人，萬萬不可三妻四妾。雙雙對對之間的邊，整體成為一個「匹配」。



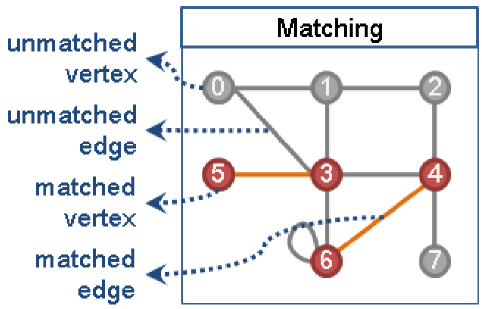
更簡單的說法是：令圖上各點僅連接著零條邊或一條邊，這些邊構成的集合稱作一個「匹配」。

Matched Vertex 與 Unmatched Vertex

一個點要嘛就是和另一個點比翼雙飛，要嘛就是孑然一身 —— 前者為「匹配點」，後者為「未匹配點」。

Matched Edge 與 Unmatched Edge

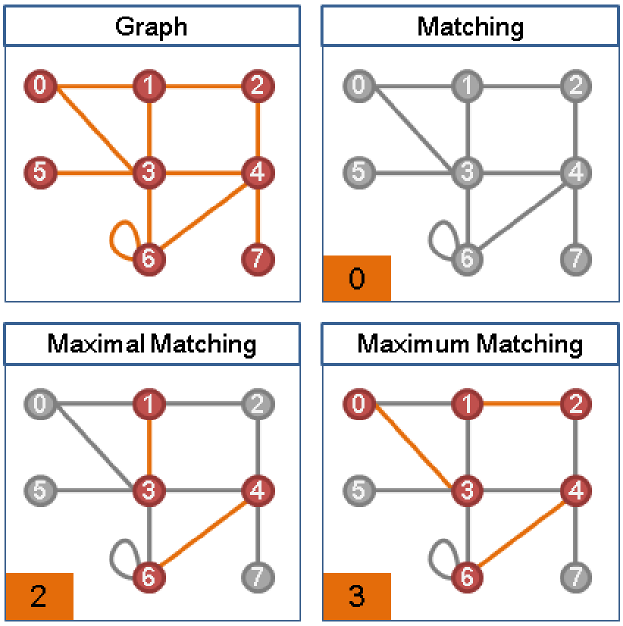
出雙入對的兩點之間的邊為「匹配邊」，除此以外則為「未匹配邊」。一個匹配是由許多匹配邊所組成的。



Cardinality

一個匹配擁有的匹配邊數目，也就是配對的數目，稱作 Cardinality ，尚無適當中譯。

◀ Index

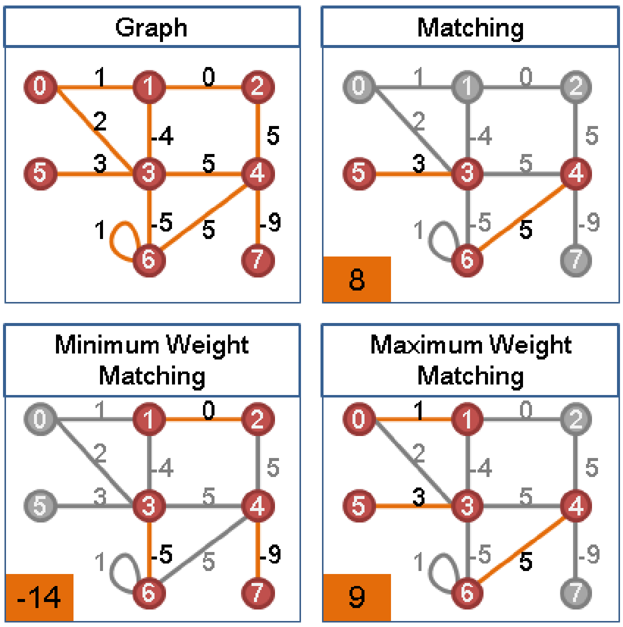


順便介紹一些特別的匹配：

maximal matching：一張圖中，沒有辦法直接增加配對數的匹配。
maximum matching：一張圖中，配對數最多的匹配。也是maximal matching。
perfect matching：一張圖中，所有點都送作堆的匹配。也是maximum matching。

Weight

當圖上的邊都有權重，一個匹配的權重是所有匹配邊的權重總和。



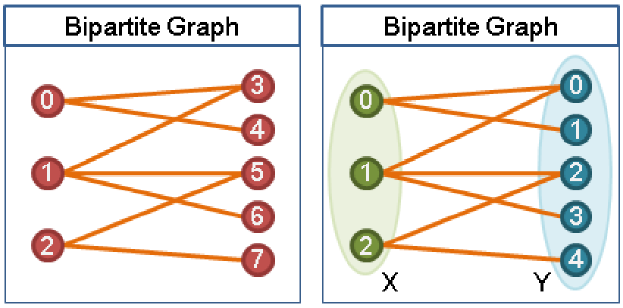
順便介紹一些特別的匹配：

maximum weight matching：
一張圖中，權重最大的匹配。
maximum weight maximum cardinality matching：
一張圖中，配對數最多的前提下，權重最大的匹配。
maximum weight perfect matching：
一張圖中，所有點都送作堆的前提下，權重最大的匹配。

Bipartite Matching

Bipartite Graph

「二分圖」是圖的一種特例。一張二分圖的結構是：兩群點（通常標記作 X 集合與 Y 集合）、橫跨這兩群點的邊（ X 與 Y 之間）。至於兩群點各自之內是沒有邊的（ X 與 X 、 Y 與 Y 間）。



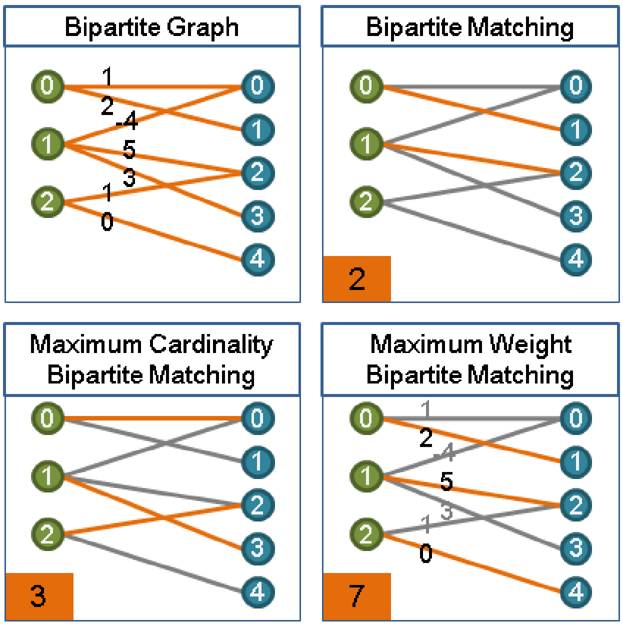
順帶一提，二分圖構造較單純，其資料結構可以進行精簡：

	0	1	2	3	4	5	6	7
0	0	0	0	1	1	0	0	0
1	0	0	0	1	0	1	1	0
2	0	0	0	0	0	1	0	1
3	1	1	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0
5	0	1	1	0	0	0	0	0
6	0	1	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0

X \ Y	0	1	2	3	4
0	1	1	0	0	0
1	1	0	1	1	0
2	0	0	1	0	1

Bipartite Matching

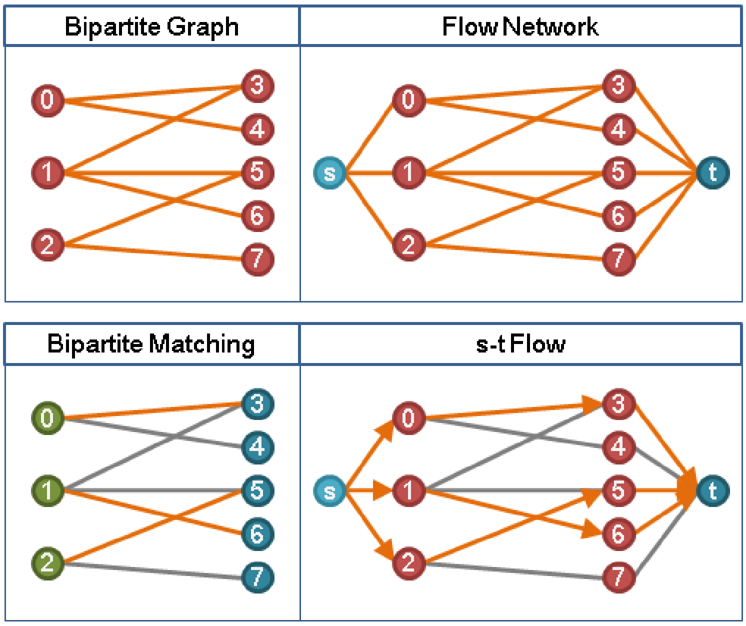
「二分匹配」。一張二分圖上的匹配稱作二分匹配，理所當然所有的匹配邊都是這橫跨這兩群點的邊，就像是連連看一樣。



以 Flow 解 Bipartite Matching

一側接上源點，一側接上匯點，即可利用網路流來解決最大二分匹配問題、最大（小）權二分匹配問題。

◀ Index



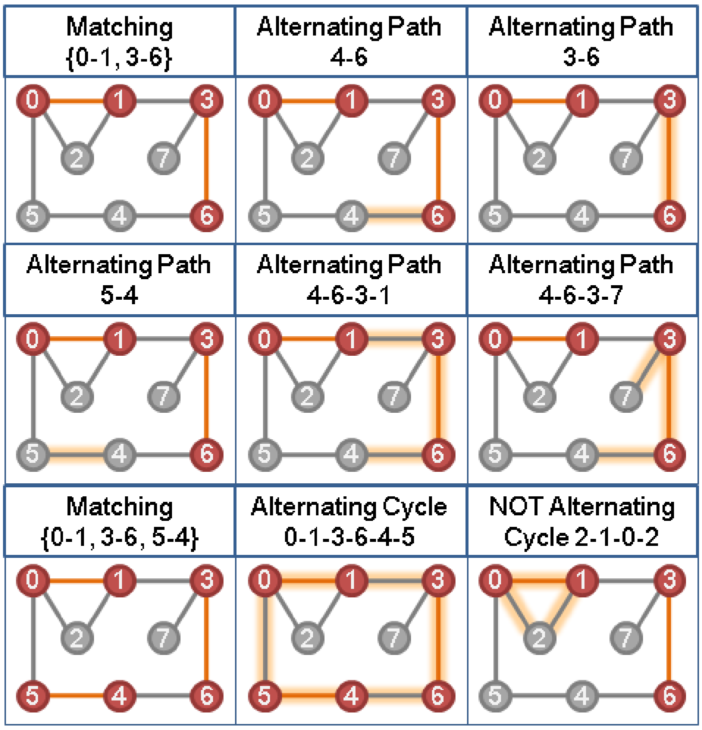
Augmenting Path Theorem (Berge's Theorem)

本章提要

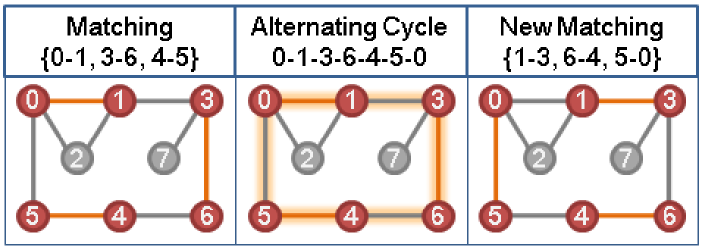
Berge's Theorem 是尋找最大匹配的一個重要理論。在這個章節中，將會講解匹配的相關知識，並證明 Berge's Theorem ，最後提出一種計算最大匹配的手段。

Alternating Path 與 Alternating Cycle

「交錯路徑」與「交錯環」，在一張存在匹配的圖上，匹配邊和未匹配邊彼此相間的一條路徑與一只環。

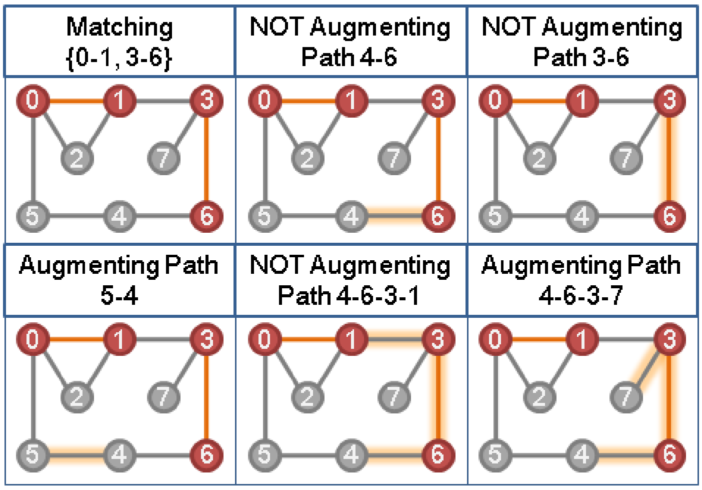


交錯環有個有趣的特性：顛倒交錯環上的匹配邊和未匹配邊，可以改變匹配，但不影響 Cardinality 。

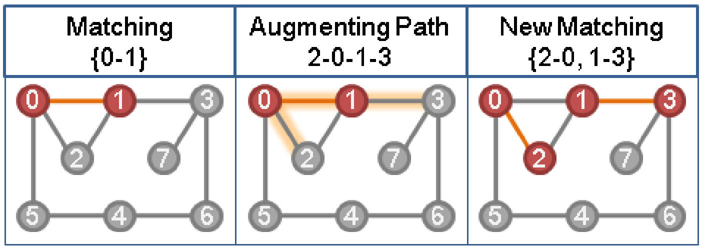


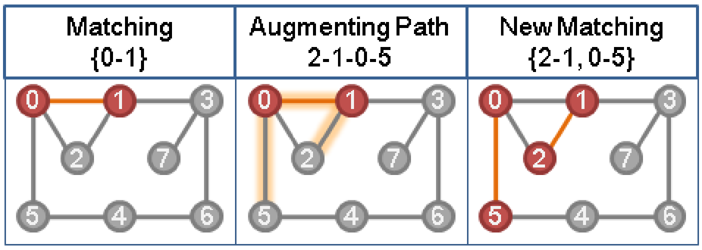
Augmenting Path

「擴充路徑」，是第一個點和最後一個點都是未匹配點的一條交錯路徑，因此第一條邊和最後一條邊都是未匹配邊。



擴充路徑有個更有趣的特性：顛倒擴充路徑上的匹配邊和未匹配邊，可以改變匹配，並且讓 Cardinality 增加一。



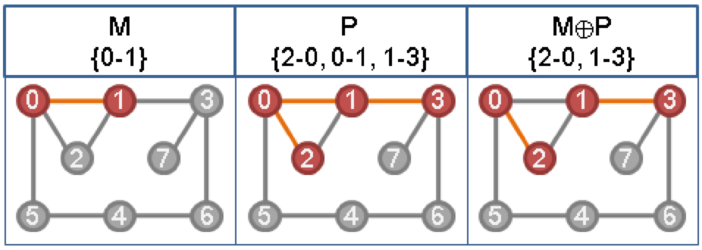


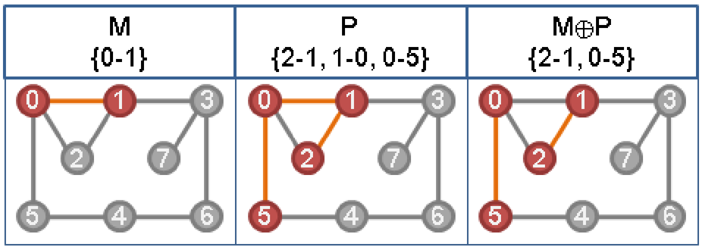
Symmetric Difference

兩個集合 A 和 B 的「對稱差集」定義為 A⊕B = (A∪B) - (A∩B)。例如 A = {1,3,4,5} 、 B = {2,4,5,7} 、 A⊕B = {1,2,3,7} ，沒有重複出現的元素將會留下，重複出現的元素將會消失。

對稱差集非常適合用來描述「顛倒擴充路徑上的匹配邊與未匹配邊」這件事情。現在有一個匹配 M ，和一條擴充路徑 P （拆開成邊），那麼 M⊕P 會等於新匹配。

◀ Index

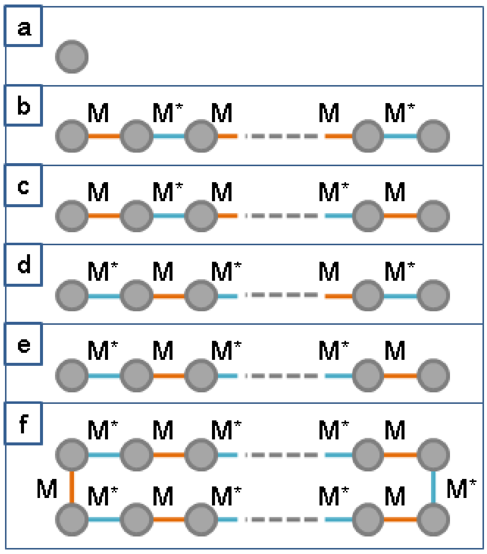




坊間書籍常以對稱差集來表述匹配相關理論。在此特別將對稱差集的概念介紹給各位，希望各位往後遇到 ⊕ 這個符號時，不會下意識地認為它艱深晦澀。

Symmetric Difference of Matching

同一張圖上的兩種匹配 M 和 M* 也可以計算對稱差集 M⊕M* ，總共會產生六大類 connected component ，都是交錯路徑或者交錯環，各位若是不信可以親自實驗看看：



兩個匹配的對稱差集，提供了這兩個匹配互相變換的管道：對其中一個匹配來說，只要顛倒整個對稱差集中的匹配邊與未匹配邊，就可以變成另一個匹配。寫成數學式子就是：令 M⊕M* = P ，則 M⊕P = M* 、 M*⊕P = M 。

Augmenting Path Theorem

從圖上任取一個未匹配點，如果找不到以此點作為端點的擴充路徑，那麼這張圖會有一些最大匹配不會包含此點。更進一步來說，就算從這張圖上刪除此點（以及相連的邊），以剩餘的點和邊，還是可以找到原本那張圖的其中一些最大匹配。

證明不困難，利用一下先前所學到的東西，便可以推理出來：

```
令當下的匹配M找不到以未匹配點p作為端點的擴充路徑，
並令M*是該圖的其中一個最大匹配。
1. 如果p不在M*上：
```

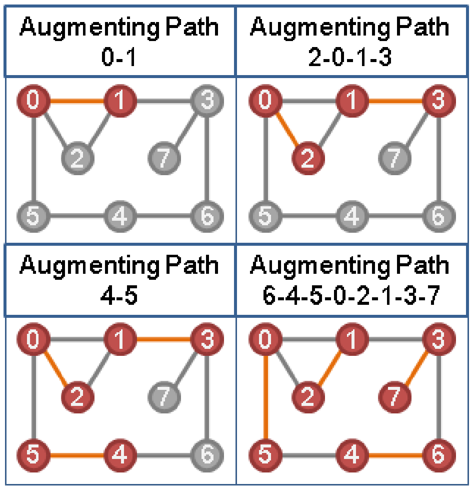
```
    刪除此點完全不會對M和M*有任何影響，定理成立。
2. 如果p在M*上：
  2-1. p對於M來說是未匹配點。理所當然p不在M上。
  2-2. 考慮M⊕M*的六種情形。p不在M上，且p在M*上，所以只有d或e符合條件。
  2-3. M找不到以p作為端點的擴充路徑，所以d不符合條件，只有e符合條件。
  2-4. 對於M*來說，只要照著e顛倒匹配邊和未匹配邊，
       就可以製造出另一個不會包含p的最大匹配，
       成為1.的情形，定理還是成立。
```

這個理論相當的重要，它表明了一個找最大匹配的手段：

```
1. 一開始圖上所有點都是未匹配點。
2. 將圖上每一個未匹配點都嘗試作為擴充路徑的端點：
  甲、如果找得到擴充路徑：
      沿著擴充路徑修改現有匹配，以增加Cardinality。
      （此未匹配點變成了匹配點。）
  乙、如果找不到擴充路徑：
      直接刪除此點。繼續下去仍然可以找到原圖的其中一個最大匹配。
      （此未匹配點被刪除。）

所有的未匹配點要嘛變成匹配點，要嘛被刪除，
因此未匹配點最後會盡數消失，同時產生一個最大匹配。
```

其要點在於：反覆利用 Augmenting Path Theorem 。儘管圖上的點不斷在減少，匹配也一直在改變，依然能找到原圖的其中一個最大匹配。



Augmenting Path Theorem ，另一種形式

一個匹配若無擴充路徑，就是最大匹配。

要是圖上所有未匹配點都不能當作擴充路徑的端點，就代表著圖上根本就沒有擴充路徑； Cardinality 無法增加，就代表著當下的匹配就是最大匹配囉！

```
不斷找擴充路徑，直到找不到為止。此時的匹配就是最大匹配。
```

Maximum Cardinality Bipartite Matching: Augmenting Path Algorithm

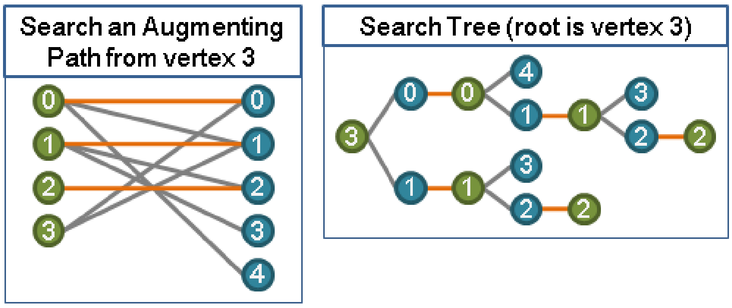
用途

找出一張二分圖的其中一個最大二分匹配。

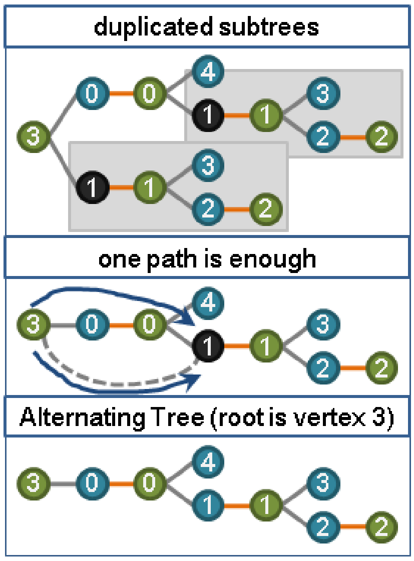
Alternating Tree

前面的章節以 Augmenting Path Theorem 提出了一個找最大匹配的方式，但是癥結在於我們選定一個未匹配點之後，還不知道如何

有效率的尋找擴充路徑。我們可以選定一個未匹配點作為樹根，然後建立搜尋樹，嘗試列出所有的交錯路徑，從樹根出去的路徑都是交錯路徑 —— 藉此找擴充路徑。



有個重要的發現是：在搜尋樹當中，當兩條交錯路徑撞在同一個點，將來還是只能選擇其中一條路徑來進行擴充，所以現在只要留下一條路徑就夠了。



根據這個重要的發現，圖上的每個點、每條邊只需經過一次，就能判斷出擴充路徑。我們得以用 Graph Traversal 來找一條擴充路徑，並得到一棵樹。

如此得到的樹稱作「交錯樹」，從樹根出去的路徑仍都是交錯路徑。很幸運的，二分圖中的每條交錯路徑都是在 X 與 Y 之間來回，交錯樹很容易建立，亦可以很快的看出擴充路徑在哪裡：若我們選定的未匹配點、擴充路徑的端點是在 X 上，它會是交錯樹的樹根；擴充路徑的另一個端點、未匹配點就一定會在 Y 上，它會是交錯樹的樹葉。

演算法

```
1. 一開始圖上所有點都是未匹配點。
2. 將X的每個未匹配點依序嘗試作為擴充路徑的端點，
   並以Graph Traversal建立交錯樹，以尋找擴充路徑。
  （X的未匹配點都處理過的話，Y的未匹配點就不會再有擴充路徑，故只需找X側。）
  甲、如果找得到擴充路徑：
      沿著擴充路徑修改現有匹配，以增加Cardinality。
     （此未匹配點變成了匹配點。）
  乙、如果找不到擴充路徑：
      直接刪除此點。繼續下去仍然可以找到原圖的其中一個最大匹配。
     （此未匹配點被刪除。）
```

這個演算法運作起來，實際上就跟接上了源點與匯點再進行 Ford-Fulkerson Algorithm （ Augmenting Path Algorithm ）一樣。

時間複雜度

時間複雜度為 O(V) 次 Graph Traversal 的時間。圖的資料結構為 adjacency matrix 的話，便是 $O(V^3)$ ；圖的資料結構為 adjacency lists 的話，便是 O(VE) 。

找出一個最大二分匹配（精簡過的 adjacency matrix ）

以 DFS 來找擴充路徑，程式碼變得相當精簡。

```
int nx, ny;             // X的點數目、Y的點數目
int mx[100], my[100];   // X各點的配對對象、Y各點的配對對象
bool vy[100];           // 記錄Graph Traversal拜訪過的點
bool adj[100][100];     // 精簡過的adjacency matrix

// 以DFS建立一棵交錯樹
bool DFS(int x)
{
    for (int y=0; y<ny; ++y)
        if (adj[x][y] && !vy[y])
        {
            vy[y] = true;

            // 找到擴充路徑
            if (my[y] == -1 || DFS(my[y]))
            {
                mx[x] = y; my[y] = x;
                return true;
            }
        }
    return false;
}

int bipartite_matching()
{
    // 全部的點初始化為未匹配點。
    memset(mx, -1, sizeof(mx));
    memset(my, -1, sizeof(my));

    // 依序把X中的每一個點作為擴充路徑的端點，
    // 並嘗試尋找擴充路徑。
    int c = 0;
    for (int x=0; x<nx; ++x)
//      if (mx[x] == -1)    // x為未匹配點，這行可精簡。
        {
            // 開始Graph Traversal
            memset(vy, false, sizeof(vy));
            if (DFS(i)) c++;
        }
    return c;
}
```

另外採用 BFS 也是可以的，這裡就不贅述了。

Greedy Matching

這裡介紹一個加速的手段：一開始就把一些明顯可以配對的點給配對起來，這樣就不必替每一個未匹配點建立交錯樹了。這個手段在

圖很龐大的時候可以發揮作用。

```
int greedy_matching()
{
    int c = 0;
    for (int x=0; x<nx; ++x)
        if (mx[x] == -1)
            for (int y=0; y<ny; ++y)
                if (my[y] == -1)
                    if (adj[x][y])
                    {
                        mx[x] = y; my[y] = x;
                        c++;
                        break;
                    }
    return c;
}

int bipartite_matching()
{
    memset(mx, -1, sizeof(mx));
    memset(my, -1, sizeof(my));

    int c = greedy_matching();  // 能連的先連一連

    for (int x=0; x<nx; ++x)
        if (mx[x] == -1)    // 這行記得補上來
        {
            memset(vy, false, sizeof(vy));
            if (DFS(i)) c++;
        }
    return c;
}
```

它的時間複雜度僅為一次 Graph Traversal 的時間，不太會影響原演算法的運行效率。

UVa 259 670 753 10080 10092 10243 10418 10984 663 11148

Maximum Cardinality Matching: Blossom Algorithm

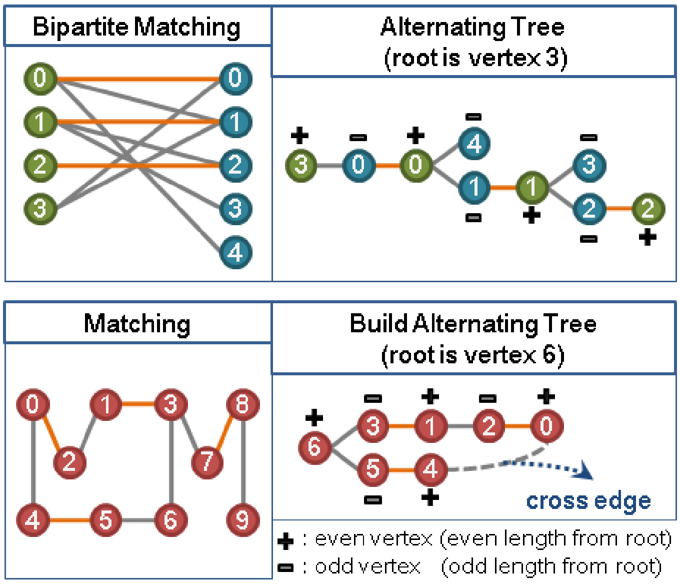
用途

找出一張無向圖的其中一個最大匹配。

Alternating Tree ： Cross Edge

嘗試利用二分圖的 Augmenting Path Algorithm ，不斷選定未匹配點作為交錯樹的樹根，然後尋找擴充路徑。

這裡定義距離樹根偶數條邊的點稱作「偶點」，距離樹根奇數條邊的點稱作「奇點」 —— 可以發現一般的 Graph 建立交錯樹時，會有個與 Bipartite Graph 不一樣的地方，就是會有偶點到偶點的邊。

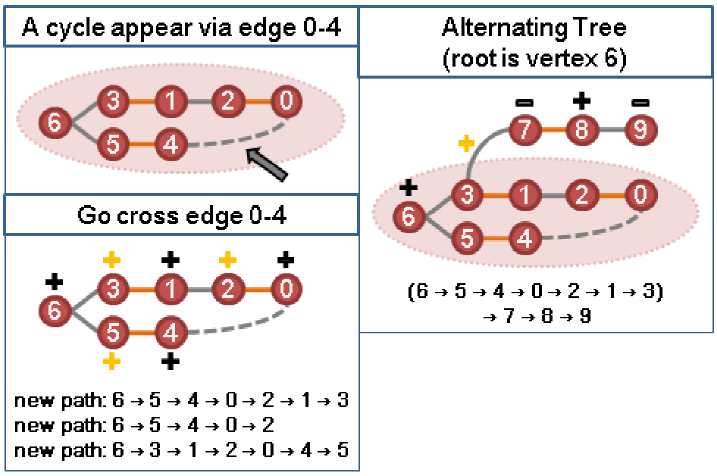


二分圖的交錯樹：
1. 偶點到奇點：一定是未匹配邊。
2. 奇點到偶點：一定是已匹配邊。
3. 偶點到偶點：二分圖不會有這種邊。
4. 奇點到奇點：二分圖不會有這種邊。

一般圖的交錯樹：
1. 偶點到奇點：一定是未匹配邊。
2. 奇點到偶點：一定是已匹配邊。
3. 偶點到偶點：一定是未匹配邊，且會形成「花」。
4. 奇點到奇點：交錯樹不會有這種邊，因為不會形成交錯路徑。

Alternating Tree ： Cycle

偶點到偶點的邊，在交錯樹上會形成一個環。只要穿越這條偶點到偶點的邊，以繞遠路的方式，環上所有奇點都能夠成為偶點，而且將來可以延伸出更多條交錯路徑。



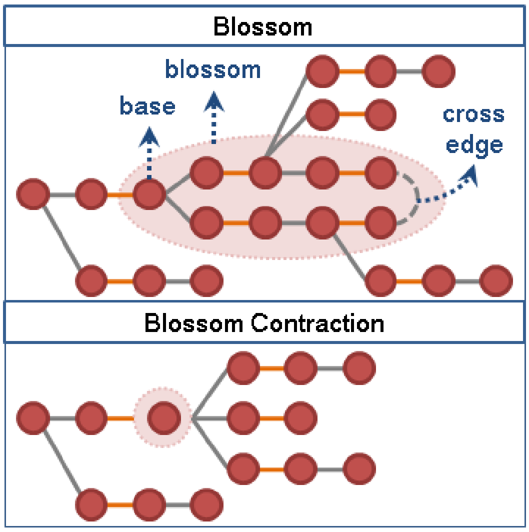
原本奇點只能以匹配邊連到偶點，無法額外延伸出其他交錯路徑；現在一般圖的交錯樹中，多了偶點到偶點的邊，奇點因此活躍了。環上的所有奇點，可以搖身一變成為偶點，然後重新延伸出交錯路徑！

Blossom

在交錯樹上，分岔的兩段交錯路徑，接上一條偶點到偶點的邊，所形成的奇數條邊的環，就稱作「花」。花上兩條未匹配邊的銜接點，則稱作「花托」，宛如花開在交錯樹上。

Blossom Contraction

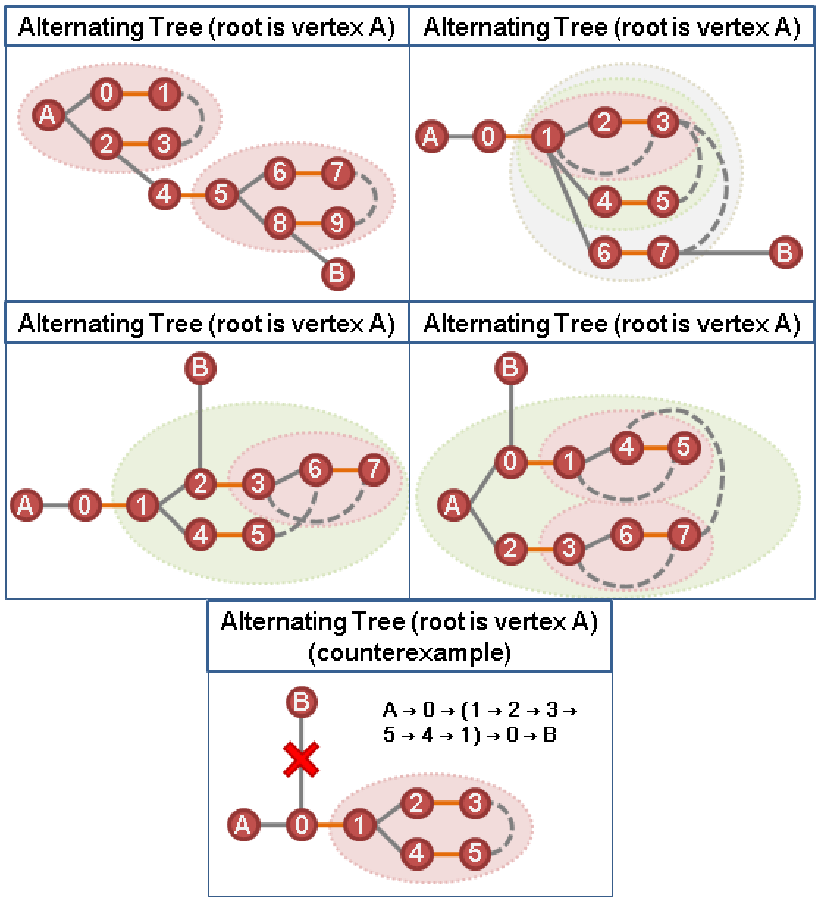
既然花上的點都可以成為偶點，那麼乾脆把花直接縮成一個偶點，會讓交錯樹變得更簡潔明白。



交錯樹上可能會有許多偶點到偶點的邊，形成許多朵重重疊疊的花，我們可以用任意順序縮花。實作時，為了容易找到花，可以在建立交錯樹的途中，一旦發現偶點到偶點的邊就立即縮花。一句話，一旦發現花就立即縮花。

縮花的次數呢？一朵花最少有三個點，縮花後成為一個點，前前後後少了兩個點。由此推得： V 個點的圖建立一棵交錯樹，最多縮花 V/2 次；如果再多縮幾朵花，樹上就沒有點了。

路徑沿著花繞來繞去，繞得你暈頭轉向。



演算法

```
1. 一開始圖上所有點都是未匹配點。
2. 將圖上每個未匹配點依序嘗試作為擴充路徑的端點，
   並以Graph Traversal建立交錯樹，以尋找擴充路徑。
   甲、走到未拜訪過的點：
      a. 如果是已匹配點，則延伸交錯樹，一條未匹配邊再加一條已匹配邊。
      b. 如果是未匹配點，則找到擴充路徑。
   乙、走到已拜訪過的點：
      a. 如果是偶點，形成花。做花的處理。
      b. 如果是奇點，根據只需留一條路徑的性質，什麼都不必做。
```

```
花的處理：
1. 找出花托，即是x與y的Lowest Common Ancestor，
2. 設定一下到達花上各奇點之交錯路徑。
   一定會經過cross edge。注意花托別重複經過。
3. 把花上面的點全部當作偶點。
   或者，乾脆把花直接縮成一個偶點。
   縮花可用Disjoint Sets資料結構。
```

找出一個最大匹配（ adjacency matrix ）

下面程式碼採用 BFS 建立交錯樹，不縮花。

總共進行 V 次 Graph Traversal ，每次 Graph Traversal 需要花 $O(V^2)$ 時間建立樹根至圖上各點的交錯路徑，用 deque 資料結構記錄之。

圖的資料結構為 adjacency matrix 的話，便是 $O(V^4)$ ；圖的資料結構為 adjacency lists 的話，便是 $O(V^2(V+E))$ ，可簡單寫成 $O(V^2E)$ 。

```
const int V = 50;   // 圖的點數，編號為0到V-1。
bool adj[50][50];   // adjacency matrix
deque<int> p[50];   // p[x]記錄了樹根到x點的交錯路徑。
int m[50];          // 記錄各點所配對的點，值為-1為未匹配點。
int d[50];          // 值為-1未拜訪、0偶點、1奇點。
int q[50], *qf, *qb;    // queue，只放入偶點。

// 設定好由樹根至花上各個奇點的交錯路徑，並讓奇點變成偶點。
// 只處理花的其中一邊。
// 邊xy是cross edge。bi是花托的索引值。
void label_one_side(int x, int y, int bi)
{
    for (int i=bi+1; i<p[x].size(); ++i)
    {
        int z = p[x][i];
        if (d[z] == 1)
        {
            // 設定好由樹根至花上奇點的交錯路徑。
            // 會經過cross edge。
            p[z] = p[y];
            p[z].insert(p[z].end(), p[x].rbegin(), p[x].rend()-i);

            d[z] = 0;   // 花上的奇點變偶點
            *qb++ = z;  // 將來可以延伸出交錯路徑
        }
    }
}

// 給定一個未匹配點r，建立交錯樹。
bool BFS(int r)
{
```

```
    for (int i=0; i<V; ++i) p[i].clear();
    p[r].push_back(r);                      // 交錯樹樹根

    memset(d, -1, sizeof(d));
    d[r] = 0;                               // 樹根是偶點

    qf = qb = q;
    *qb++ = r;                              // 樹根放入queue

    while (qf < qb)
        for (int x=*qf++, y=0; y<V; ++y)
            if (adj[x][y] && m[y] != y) // 有鄰點，點存在。
                if (d[y] == -1)             // 沒遇過的點
                    if (m[y] == -1)         // 發現擴充路徑
                    {
                        for (int i=0; i+1<p[x].size(); i+=2)
                        {
                            m[p[x][i]] = p[x][i+1];
                            m[p[x][i+1]] = p[x][i];
                        }
                        m[x] = y; m[y] = x;
                        return true;
                    }
                    else                    // 延伸交錯樹
                    {
                        int z = m[y];

                        p[z] = p[x];
                        p[z].push_back(y);
                        p[z].push_back(z);

                        d[y] = 1; d[z] = 0;
                        *qb++ = z;
                    }
                else
                    if (d[y] == 0)          // 形成花
                    {
                        // 從交錯路徑中求得LCA的索引值
                        int bi = 0;
                        while (bi < p[x].size()
                            && bi < p[y].size()
                            && p[x][bi] == p[y][bi]) bi++;
                        bi--;

                        // 兩條路徑分開標記
                        // 不必擔心x與y在同一朵花上
                        label_one_side(x, y, bi);
                        label_one_side(y, x, bi);
                    }
                    else                    // 只需留一條路徑
                        ;
    return false;
}

int match()
{
    memset(m, -1, sizeof(m));
```

◀ Index


```

    int c = 0;
    for (int i=0; i<V; ++i)
        if (m[i] == -1)
            if (BFS(i))
                c++;        // 找到擴充路徑，增加匹配數
            else
                m[i] = i;   // 從圖上刪除此點
    return c;
}

int main()
{
    cin >> V;

    int x, y;
    while (cin >> x >> y)
        adj[x][y] = adj[y][x] = true;

    cout << "匹配數為" << match();
    for (int i=0; i<V; ++i)          // 印出所有的匹配邊
        if (i < m[i])
            cout << i << ' ' << m[i] << endl;

    return 0;
}
```

找出一個最大匹配（ adjacency matrix ）

下面程式碼採用 BFS 建立交錯樹，以 disjoint forest 實作縮花。縮花時可將花托設定為 disjoint forest 的樹根，這樣程式碼會比較簡潔。

附帶一提，求 lowest common ancestor 的過程中，縮花縮掉的點不會再度出現。因此，建一棵交錯樹的過程中，算 least common ancestor 的時間複雜度總共僅為 O(V) 。

時間複雜度為 V 次 Graph Traversal 的時間，再乘上維護 disjoint forest 的時間。圖的資料結構為 adjacency matrix 的話，便是 $O(V^3\alpha(E,V))$ ；圖的資料結構為 adjacency lists 的話，便是 $O(VE\alpha(E,V))$ 。

```
const int V = 50;    // 圖的點數，編號為0到V-1。
bool adj[50][50];    // adjacency matrix
int p[50];           // 交錯樹
int m[50];           // 記錄各點所配對的點，值為-1為未匹配點。
int d[50];           // 值為-1未拜訪、0偶點、1奇點。
int c1[50], c2[50]; // 記錄花上各奇點所經過的cross edge
int q[50], *qf, *qb;     // queue，只放入偶點。

/* disjoint-sets forest */
int pp[50];
int f(int x) {return x == pp[x] ? x : (pp[x] = f(pp[x]));}
// void u(int x, int y) {pp[f(x)] = f(y);}
void u(int x, int y) {pp[x] = y;}

```

◀ **Index**


```
/* lowest common ancestor */
int v[50];

// 繞一繞花，找出擴充路徑，並擴充。源頭是r，末梢是x。
// 最初以偶點作為末端，每次往回走一條匹配邊加一條未匹配邊，
// 如果遇到花上奇點，就要繞花，以cross edge拆成兩段路徑。
void path(int r, int x)
{
    if (r == x) return;

    if (d[x] == 0)          // 還是偶點，繼續往回走。
    {
        path(r, p[p[x]]);
        int i = p[x], j = p[p[x]];
        m[i] = j; m[j] = i;
    }
    else if (d[x] == 1)     // 遇到花上奇點，就要繞花。
    {
        // 往回走會經過cross edge c1[x]-c2[x]
        path(m[x], c1[x]);  // 頭尾要顛倒
        path(r, c2[x]);
        int i = c1[x], j = c2[x];
        m[i] = j; m[j] = i;
    }
}

// 邊xy是cross edge，同時一起往上找LCA。
// 找一次的時間複雜度是O(max(x->b, y->b))，
// 不會超過縮花所縮掉的點的兩倍，縮掉的點也不會再算到，
// 故可推得：建一棵交錯樹，算LCA總共只有O(V)。
int lca(int x, int y, int r)
{
    int i = f(x), j = f(y);
    while (i != j && v[i] != 2 && v[j] != 1)
    {
        v[i] = 1; v[j] = 2;
        if (i != r) i = f(p[i]);
        if (j != r) j = f(p[j]);
    }
    int b = i, z = j; if (v[j] == 1) swap(b, z);

    for (i = b; i != z; i = f(p[i])) v[i] = -1;
    v[z] = -1;
    return b;
}

/*
// 一次就要O(V)的LCA演算法
int lca(int x, int y, int r)
{
    int v[50] = {0};
    v[r] = 1;
    for (x = f(x); x != r; x = f(p[x])) v[x] = 1;
    for (y = f(y); !v[y]; y = f(p[y])) ;
    return y;
}
*/

// 找到花時，弄好到達花上各奇點之交錯路徑，並讓奇點變成偶點。
```

```
// 只弄半邊。
// 邊xy為cross edge。b為花托。
void contract_one_side(int x, int y, int b)
{
    for (int i = f(x); i != b; i = f(p[i]))
    {
        u(i, b);     // 縮花，花托成為disjoint forest的樹根
        if (d[i] == 1) c1[i] = x, c2[i] = y, *qb++ = i;
    }
}

// 給定一個未匹配點r，建立交錯樹。
bool BFS(int r)
{
    for (int i=0; i<V; ++i) pp[i] = i;  // d. f.: init
    memset(v, -1, sizeof(v));           // lca: init

    memset(d, -1, sizeof(d));
    d[r] = 0;                           // 樹根是偶點

    qf = qb = q;
    *qb++ = r;                          // 樹根放入queue

    while (qf < qb)
        for (int x=*qf++, y=0; y<V; ++y)
            // 有鄰點。點存在。縮花成同一點後則不必處理。
            if (adj[x][y] && m[y] != y && f(x) != f(y))
                if (d[y] == -1)         // 沒遇過的點
                    if (m[y] == -1)     // 發現擴充路徑
                    {
                        path(r, x);
                        m[x] = y; m[y] = x;
                        return true;
                    }
                    else                // 延伸交錯樹
                    {
                        p[y] = x; p[m[y]] = y;
                        d[y] = 1; d[m[y]] = 0;
                        *qb++ = m[y];
                    }
                else
                    if (d[f(y)] == 0)   // 形成花
                    {
                        int b = lca(x, y, r);
                        contract_one_side(x, y, b);
                        contract_one_side(y, x, b);
                    }
                    else                // 只需留一條路徑
                        ;
    return false;
}
```

加速

之前提到的 greedy matching 在此處也是適用的。

另外，可以一口氣把圖上所有未匹配點作為樹根，建立交錯森林，當兩棵交錯樹碰到的時候，就是有擴充路徑了。這麼做可以稍微降低縮花的次數。

Timus 1099 UVa 11439 ICPC 3820

Maximum Cardinality Bipartite Matching: Hopcroft-Karp Algorithm

用途

找出一張二分圖的其中一個最大二分匹配。

演算法

每次都一口氣找出所有目前最短的擴充路徑進行擴充，直到找不到為止。正確性證明與時間複雜度證明，請參考 CLRS 的習題。【待補文字】

```
1. 一開始圖上所有點都是未匹配點。
2. 重複下列動作，直到無法增加匹配（最多執行sqrtV次）：
   甲、以X的所有未匹配點同時作為樹根，
       採用BFS建立交錯森林，一次僅延展一整層，
       直到發現所有目前最短的擴充路徑。
       （時間複雜度為一次Graph Traversal的時間。）
   乙、對於各個Y的未匹配點，
       若為交錯森林的樹葉（最短擴充路徑的端點），就往樹根方向找擴充路徑。
       注意一旦拜訪過的點就不再拜訪。
       （時間複雜度為一次Graph Traversal的時間。）
       註：也可以由樹根往樹葉找。
```

時間複雜度

時間複雜度為 O(sqrtV) 次 BFS 的時間。圖的資料結構為 adjacency matrix 的話，便是 $O(V^2\text{sqrt}V) = O(V^{2.5})$；圖的資料結構為 adjacency lists 的話，便是 O((V+E)sqrtV)，可簡單寫成 O(EsqrtV)。

找出一個最大二分匹配（精簡過的 adjacency matrix）

```
int nx, ny;             // X的點數目、Y的點數目
int mx[100], my[100];   // X各點的配對對象、Y各點的配對對象
int px[100], py[100];   // 交錯森林
bool adj[100][100];     // 精簡過的adjacency matrix

// 由樹葉往樹根找擴充路徑，並擴充。
int trace(int y)
{
    int x = py[y], yy = px[x];
    py[y] = px[x] = -1; // 一旦拜訪過的點就不再拜訪
    if (mx[x] == -1 || (yy != -1 && trace(yy)))
    {
        mx[x] = y; my[y] = x;
        return 1;
    }
    return 0;
}

int bipartite_matching()
{
    memset(mx, -1, sizeof(mx));
    memset(my, -1, sizeof(my));

```

◀ Index


```
    int q[100], *qf, *qb;
    int c = 0;
    while (true)    // 如果還能找到擴充路徑就繼續
    {
        memset(px, -1, sizeof(px));
        memset(py, -1, sizeof(py));
        qf = qb = q;

        // 把X的未匹配點，作為交錯森林的樹根。
        for (int x=0; x<nx; ++x)
            if (mx[x] == -1)
            {
                *qb++ = x;
//              px[x] = -2;
            }

        // 採用BFS建立交錯森林，一次僅延展一整層，
        // 直到發現所有目前最短的擴充路徑。
        bool ap = false;    // 是否存在擴充路徑
        for (int* tqb = qb; qf < tqb && !ap; tqb = qb)
            for (int x=*qf++, y=0; y<ny; ++y)
                if (adj[x][y] /*&& mx[x] != y*/ && py[y] == -1)
                {
                    py[y] = x;
                    if (my[y] == -1) ap = true;
                    else *qb++ = my[y], px[my[y]] = y;
                }
        if (!ap) break;

        // 由樹葉往樹根找擴充路徑，並擴充。
        for (int y=0; y<ny; ++y)
            if (my[y] == -1 && py[y] != -1)
                c += trace(y);
    }
    return c;
}
```

Maximum Cardinality Matching: Micali-Vazirani Algorithm

用途

找出一張無向圖的其中一個最大匹配。

演算法

每次都同時找出所有目前最短的擴充路徑，並進行擴充，直到找不到為止。

http://www.cc.gatech.edu/~vazirani/new-proof.pdf

時間複雜度

O(EsqrtV) 。

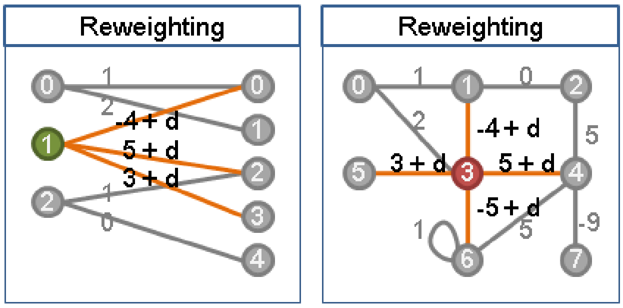
Maximum Weight Perfect Bipartite Matching: Hungarian Algorithm (Kuhn-Munkres Algorithm)

用途

匈牙利演算法是幾位匈牙利學者所發明的，用來求出一張二分圖的最大（小）權完美二分匹配。稍做修改，也能用來求出最大（小）權最大二分匹配、最大（小）權二分匹配。

調整權重

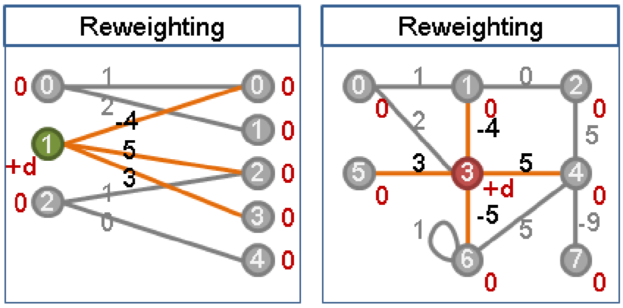
一個點連接的所有邊，等量增加權重、等量減少權重，都不會影響最大權完美匹配的位置。



此性質二分圖和一般圖都成立。

建立 vertex labeling

幫各個點都創造一個變數，直接在點上調整權重，代替在邊上調整權重，藉此減少調整權重的時間。



轉換問題：最小化所有點的權重總和 = 最大化所有匹配邊的權重總和

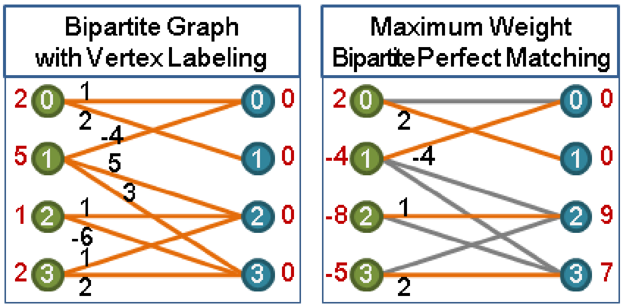
建立一組 vertex labeling ：令圖上每一條邊，其兩端點的權重總和，大於等於邊的權重。

由於最大完美匹配必定用到每一個點：所有點的權重總和，必定大於等於所有匹配邊的權重總和（最大權完美匹配的權重）。

現在只要盡力降低所有點的權重總和，就能逼近最大權完美匹配的權重。

```
令l(x)是vertex labeling，x是圖上任意一個點。
令vertex labeling讓圖上所有邊(x,y)都滿足l(x)+l(y)>=adj(x,y)。

想辦法降低l(x)，讓 ∑ l(x) =      ∑ adj(x,y) 為最大權完美二分匹配的權重。
                x為圖上一點  (x,y)為匹配邊
```



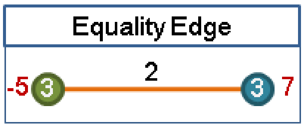
設定上限，然後不斷降低上限，降低到極限就碰到最大值了。原本一個求最大值的問題，變成了一個求最小值的問題。這是很實用的數學轉換。

【註：以 Linear Programming 的觀點來看，這個轉換正是 primal problem 與 dual problem 之間的轉換。】

這個轉換有個重要目的：操作 vertex labeling 而不操作 edge labeling ，藉此減少調整權重的時間。

現在只要求出一組總和最小的 vertex labeling ，就得到最大權完美二分匹配。

Equality Edge （ Admissible Edge ）



一條邊，兩端點的點權重相加，恰好等於邊權重，稱為「等邊」。當 vertex labeling 的總和降低到極限的時候，可以發現最大權完美二分匹配的所有匹配邊都是「等邊」。

> 定義「等邊」(x,y)是滿足l(x)+l(y)=adj(x,y)的邊。

Equality Edge + Augmenting Path Algorithm

以「等邊」的概念，結合之前的 Augmenting Path Algorithm ，得到一個演算法：以「等邊」構成的擴充路徑，不斷進行擴充。由於用來擴充的邊全是「等邊」，最後一定得到的最大權匹配當然全是「等邊」。

> 一、X的每一個未匹配點，依序尋找擴充路徑，擴充路徑必須都是「等邊」。
> 　換句話說，運用Graph Traversal建立交錯樹，交錯樹必須都是「等邊」。
> 　（X的未匹點都處理過的話，Y的未匹配點就不會再有擴充路徑，故只需找X側。）
> 　回、如果找到「等邊」擴充路徑：進行擴充。
> 　回、如果找不到「等邊」擴充路徑：？？？？？

當找不到「等邊」，只好想辦法調整 vertex labeling 了。

調整 vertex labeling

該如何調整呢？要注意 vertex labeling 仍要維持大於等於的性質，而且既有的「等邊」不能被改變。

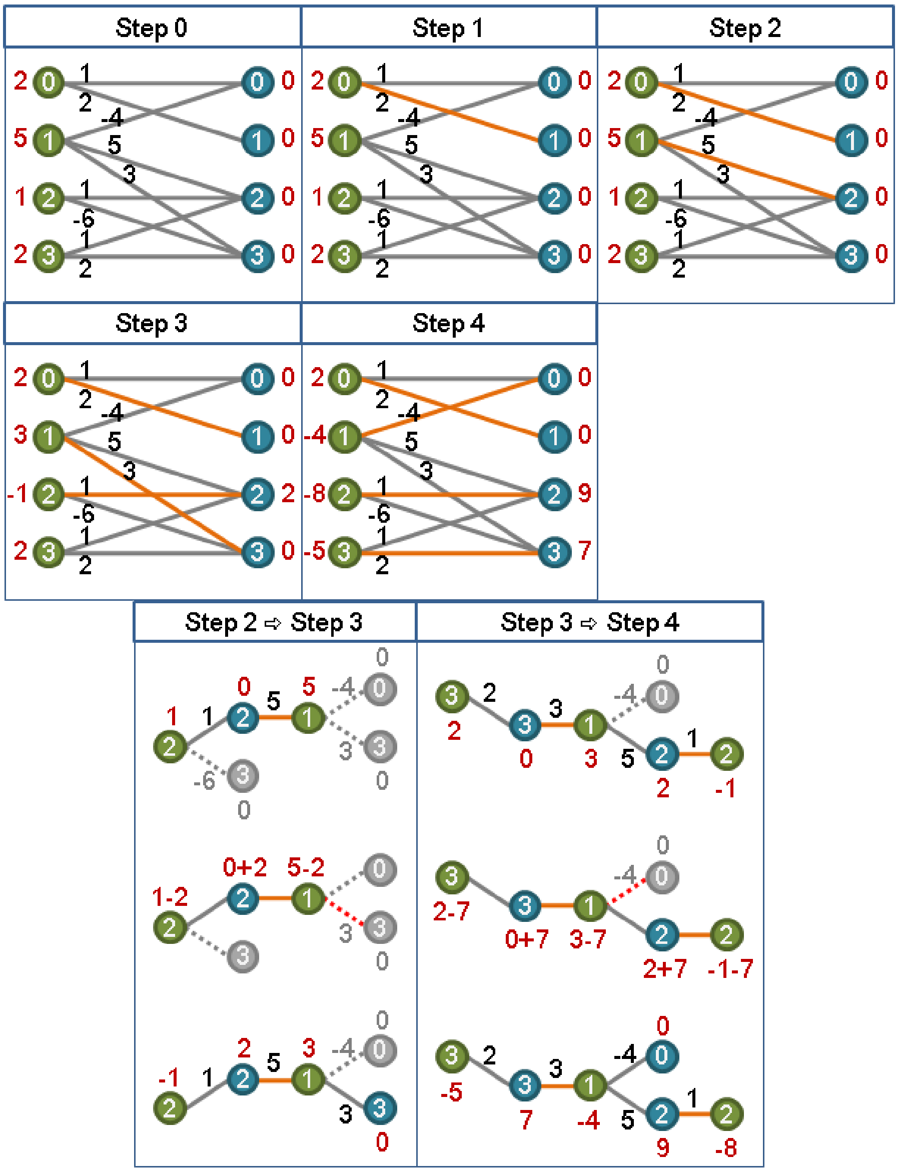
先把「等邊」構成的交錯樹延伸到極限，再把交錯樹末稍不是「等邊」的邊，取其兩端點的權重總和、減掉邊的權重，找此值最小者，交錯樹上偶點減此值、奇點加此值。

一加一減後，交錯樹內、外既有的「等邊」依然保持不動，交錯樹末稍則增添了新的「等邊」。整張圖的「等邊」只增不減。

```
令 d = min( l(x) + l(y) - adj(x,y) )，
      x為「等邊」交錯樹上一點，y為「等邊」交錯樹外一點。

讓 l(x) -= d，讓 l(x) += d。
   x為樹上偶點    x為樹上奇點

如此便製造了一條（以上）的等邊，而既有等邊保持不動，
而且維持了每一條邊的大於等於性質。
```



演算法

```
一、一開始圖上所有點都是未匹配點。
二、建立vertex labeling，使之滿足前述的大於等於性質。
三、X的每一個未匹配點，
   依序尋找「等邊」構成的擴充路徑。
   以Graph Traversal建立「等邊」構成的交錯樹。
   （X的未匹配點都處理過的話，Y的未匹配點就不會再有擴充路徑，故只需找X側。）
  甲、如果形成「等邊」構成的擴充路徑：
     沿著擴充路徑修改現有匹配，增加Cardinality。
  乙、如果找不到「等邊」，則製造新的「等邊」：
     所有交錯樹末梢的邊（都不是等邊），算最小差值，偶點減，奇點加，
     便在交錯樹末稍增加一條以上的等邊，而且既有等邊保持不動。
```

```
1. 為了節省時間，使用vertex labeling轉換問題。
2. 只用等邊延展交錯樹。修正label以利延展：偶點減，奇點加。
3-1. label總和會逐漸減少乃至收斂：修正label時，偶點總是比奇點多一個，然後看2
3-2. 等邊數量會逐漸增加乃至收斂：每次修正label就會產生一條以上的等邊。
4. 收斂後，得最大權完美匹配。匹配邊都是等邊。權重為label總和。
```

時間複雜度：一、 O(V) 個未匹配點建立交錯樹。二、一個未匹配點建立交錯樹時，最多調整 O(V) 次 label 、進行 O(V) 次 Graph Trversal 。三、每次調整 label ，都要花 $O(V^2)$ 時間找到最小差值。總時間複雜度為 $O(V^4)$ 。

```
const int X = 50;    // X的點數目，等於Y的點數目
const int Y = 50;    // Y的點數目
int adj[X][Y];       // 精簡過的adjacency matrix
int lx[X], ly[Y];    // vertex labeling
int mx[X], my[Y];    // X各點的配對對象、Y各點的配對對象
bool vx[X], vy[Y];   // 記錄是否拜訪過

// 以DFS建立一棵「等邊」交錯樹
bool DFS(int x)
{
    vx[x] = true;
    for (int y=0; y<Y; ++y)
        if (!vy[y])
            // 遇到等邊，延展交錯樹。
            if (lx[x] + ly[y] == adj[x][y])
            {
                vy[y] = true;
                if (my[y] == -1 || DFS(my[y]))
                {
                    mx[x] = y; my[y] = x;
                    return true;
                }
            }
    return false;
}

int Hungarian()
{
    // 初始化vertex labeling
//  memset(lx, 0, sizeof(lx));  // 任意值皆可
    memset(ly, 0, sizeof(ly));
    for (int x=0; x<X; ++x)
        for (int y=0; y<Y; ++y)
            if (adj[x][y] != 1e9)
                lx[x] = max(lx[x], adj[x][y]);

    // X側每一個點，分別建立「等邊」交錯樹。
    memset(mx, -1, sizeof(mx));
    memset(my, -1, sizeof(my));
    for (int x=0; x<X; ++x)
        while (true)
        {
            // 建立「等邊」交錯樹
            memset(vx, false, sizeof(vx));
            memset(vy, false, sizeof(vy));
            if (DFS(x)) break;

            // 計算最小差值（有人稱作slack）
            // 枚舉交錯樹內的X、交錯樹外的Y，
            // 以得到交錯樹末稍。
            int d = 1e9;
            for (int x=0; x<X; ++x) if (vx[x])
```

```
                for (int y=0; y<Y; ++y) if (!vy[y])
                    if (adj[x][y] != 1e9)
                        d = min(d, lx[x] + ly[y] - adj[x][y]);

            // 未新增等邊。無擴充路徑。無完美匹配。
            if (d == 1e9) return -1e9;

            // 調整交錯樹上的label
            for (int x=0; x<X; ++x)
                if (vx[x])
                    lx[x] -= d;
            for (int y=0; y<Y; ++y)
                if (vy[y])
                    ly[y] += d;
        }

    // 計算最大權完美匹配的權重
    int weight = 0;
    for (int x=0; x<X; ++x)
        weight += adj[x][mx[x]];
    return weight;
}
```

整個演算法的過程，是建立 V 次交錯樹。建立一棵交錯樹的過程，類似於 Label Setting Algorithm ：以一個未匹配點做為起點，逐次把一個奇點及一個偶點（連帶著邊、且是等邊）移入交錯樹。

也就是說，只要運用 DP 、 Priority Queue 、 Bucket Sort ，就得到更低的時間複雜度。例如仿照 Dijkstra's Algorithm ，建立 DP 表格記錄最小差值，時間複雜度為 $O(V^3)$ 。

```
const int X = 50;    // X的點數目，等於Y的點數目
const int Y = 50;    // Y的點數目
int adj[X][Y];       // 精簡過的adjacency matrix
int lx[X], ly[Y];    // vertex labeling
int mx[X], my[Y];    // X各點的配對對象、Y各點的配對對象
int q[X], *qf, *qb;  // BFS queue
int p[X];            // BFS parent，交錯樹之偶點，指向上一個偶點
bool vx[X], vy[Y];   // 記錄是否在交錯樹上
int dy[Y], pdy[Y];   // DP表格

// relaxation
void relax(int x)
{
    for (int y=0; y<Y; ++y)
        if (adj[x][y] != 1e9)
            if (lx[x] + ly[y] - adj[x][y] < dy[y])
            {
                dy[y] = lx[x] + ly[y] - adj[x][y];
                pdy[y] = x; // 記錄好是從哪個樹葉連出去的
            }
}

// 調整交錯樹上的label、調整DP表格
void reweight()
{
```

```
    int d = 1e9;
    for (int y=0; y<Y; ++y) if (!vy[y]) d = min(d, dy[y]);
    for (int x=0; x<X; ++x) if ( vx[x]) lx[x] -= d;
    for (int y=0; y<Y; ++y) if ( vy[y]) ly[y] += d;
    for (int y=0; y<Y; ++y) if (!vy[y]) dy[y] -= d;
}

// 擴充路徑
void augment(int x, int y)
{
    for (int ty; x != -1; x = p[x], y = ty)
    {
        ty = mx[x]; my[y] = x; mx[x] = y;
    }
}

// 延展交錯樹：使用既有的等邊
bool branch1()
{
    while (qf < qb)
        for (int x=*qf++, y=0; y<Y; ++y)
            if (!vy[y] && lx[x] + ly[y] == adj[x][y])
            {
                vy[y] = true;
                if (my[y] == -1)
                {
                    augment(x, y);
                    return true;
                }

                int z = my[y];
                *qb++ = z; p[z] = x; vx[z] = true; relax(z);
            }
    return false;
}

// 延展交錯樹：使用新添的等邊
bool branch2()
{
    for (int y=0; y<Y; ++y)
    {
        if (!vy[y] && dy[y] == 0)
        {
            vy[y] = true;
            if (my[y] == -1)
            {
                augment(pdy[y], y);
                return true;
            }

            int z = my[y];
            *qb++ = z; p[z] = pdy[y]; vx[z] = true; relax(z);
        }
    }
    return false;
}
```

```

int Hungarian()
{
    // 初始化vertex labeling
//  memset(lx, 0, sizeof(lx));  // 任意值皆可
    memset(ly, 0, sizeof(ly));
    for (int x=0; x<X; ++x)
        for (int y=0; y<Y; ++y)
            lx[x] = max(lx[x], adj[x][y]);

    // X側每一個點，分別建立「等邊」交錯樹。
    memset(mx, -1, sizeof(mx));
    memset(my, -1, sizeof(my));
    for (int x=0; x<X; ++x)
    {
        memset(vx, false, sizeof(vx));
        memset(vy, false, sizeof(vy));
        memset(dy, 0x7f, sizeof(dy));
        qf = qb = q;
        *qb++ = x; p[x] = -1; vx[x] = true; relax(x);
        while (true)
        {
            if (branch1()) break;
            reweight();
            if (branch2()) break;
        }
    }

    // 計算最大權完美匹配的權重
    int weight = 0;
    for (int x=0; x<X; ++x)
        weight += adj[x][mx[x]];
    return weight;
}
```

延伸閱讀：另一種調整 vertex labeling 的方式

```
令 d = min( l(x) + l(y) - adj(x,y) )，
       x為「等邊」交錯樹上一點，y為「等邊」交錯樹外一點。

讓 l(x) -= 0.5d，讓 l(x) += 0.5d。
   x為X側的點          x為Y側的點

如此可製造一條（以上）的等邊，而且既有等邊保持不動。
```

UVa 11383 1411 ICPC 3276

最小權完美二分匹配

有兩種方式。第一種是把所有邊的權重變號即可。

第二種是把前述大於等於的性質改成小於等於、 vertex labeling 降低總和改為升高總和。

最大權最大二分匹配

vertex labeling 額外增加一個限制：每一點的權重不得小於零，值為零的點最後將成為未匹配點。證明就省略了。

最大權二分匹配

同上。

矩陣形式的匈牙利演算法

匈牙利演算法的原始論文，是以矩陣為視角：把圖上每一條邊，取其兩端 label 和、再減掉權重的值，放到二分圖的 adjacency matrix ，針對 adjacency matrix 進行操作。宣稱時間複雜度為 $O(V^3)$ 。

Maximum Weight Perfect Matching: Blossom Algorithm

用途

用來求出一張圖的最大（小）權最大匹配。

演算法

http://www.arl.wustl.edu/~jst/cse/542/lec/lec15.pdf

http://www.arl.wustl.edu/~jst/cse/542/lec/lec16.pdf

基於匈牙利演算法，仍然以等邊來建立交錯樹，但是要額外考慮花的問題。

當形成花的時候，就把花上所有點標記為偶點，並進行縮花。調整權重時，偶點減 d 、奇點加 d ，此舉造成花上的每條匹配邊，皆與實際上的權重值少了 2d 。

所以，每當調整權重，就必須記錄這失去的 2d ，因此，另外再建立一組 blossom labeling ，每當剛形成花時，其值為零，之後若調整權重，就加上 2d 。

調整權重改成：

```
讓 l(x) -= d，讓 l(x) += d。
   x為樹上偶點    x為樹上奇點

讓 b(B) += 2d，讓 b(B) -= 2d。
   B為最上層偶花   B為最上層奇花

如此可製造一條（或者一條以上）的等邊，且既有等邊保持不動。
```

等邊的定義改成：

```
定義「等邊」(x,y)是滿足l(x) + l(y) + ∑ b(B) = adj(x,y)的邊。
                                      B是花，且邊(x,y)在B上。

令 d = min( l(x) + l(y) + ∑ b(B) - adj(x,y) )，
       x為「等邊」交錯樹上一點，y為「等邊」交錯樹外一點。
```

擴充時，不拆花，仍將花暫時留著。當花變成

```
令l(x)是vertex labeling，x是圖上任意一個點。
令b(B)是blossom labeling，B是建立交錯樹時形成的任意一朵花。
令vertex labeling與blossom labeling讓圖上所有邊(x,y)都滿足：
l(x) + l(y) + ∑ b(B) = adj(x,y)
               B是花，且邊(x,y)在B上。
```

演算法改成：

```
1. 一開始圖上所有點都是未匹配點。
2. 建立vertex labeling，使之滿足前述的大於等於性質。
3. 將X的每個未匹配點依序嘗試作為「等邊」構成的擴充路徑的端點，
   並以Graph Traversal建立「等邊」交錯樹，以尋找「等邊」構成的擴充路徑。
  （X的未匹配點都處理過的話，Y的未匹配點就不會再有擴充路徑，故只需找X。）
  甲、如果形成「等邊」構成的擴充路徑：
      沿著擴充路徑修改現有匹配，以增加Cardinality。
  乙、如果找不到「等邊」，製造新的「等邊」：
      所有交錯樹末梢的邊（不會是等邊），算適當值，偶點減，奇點加，
      便在交錯樹末稍增加一條以上的等邊，且既有等邊保持不動。
```

```
z(B)加上2d的原因是，
縮花時花內每個點都被標成正點，  （正點減d、負點加d）
然後花內每條匹配邊的兩個端點當然都是正點都各減d，所以一條匹配邊就少2d。
故花內每條匹配邊都必須補2d回來。
注意到補2d的性質，經過擴充路徑反轉匹配邊/未匹配邊之後，還是依然如此。

只有最外層的花z(B)需要加上2d，因為z(u,v)被設計成累加所有z(B):u,v屬於B。
如此一來，調權重會簡單些，程式碼比較好寫。

因為要拆花，所以不能用Disjoint Forest，只能用普通的樹。
所以就算用DP也不能達到O(VEalpha(E,V))，還是只有O(VElogV)。
```

各位也可以嘗試建立交錯森林。但是要小心在不同樹之間、偶點與偶點之間的邊，調整權重時切記要滿足 vertex labeling 的大於小於性質。

```
1. http://vfleaking.blog.uoj.ac/blog/339
```

延伸閱讀：求最小權最大匹配

和匈牙利演算法相似，改為升高 vertex labeling 與 blossom labeling 的和。

或者把所有邊的權重變號，然後求最大權最大匹配。