圖論

林伯禧

February 6, 2023

講義勘誤

p.115 當我們發現一個點的父節點是割點時，我們便從堆疊頂端取出在這個割點子節點上面的所有點

p.117 複雜度 $O((N+M)+(N+Q)\ \sout{log N^2}\ \log^2 N)$

p.121 Too Many ~~Queries~~ Constraints, Codeforces ~~1687~~1697

p.144 ~~完美~~最大匹配 M^* 的大小

擬陣章節的 Lemma 沒有編號，但是文字裡面有 Lemma ??.??

p.156 的是指基底的強交換性

其他都是 p.157 的引理

講師簡介

- 林伯禧
- 2021, 2022 TOI 二階
- 2022 ICPC 桃園站第六名

講師簡介

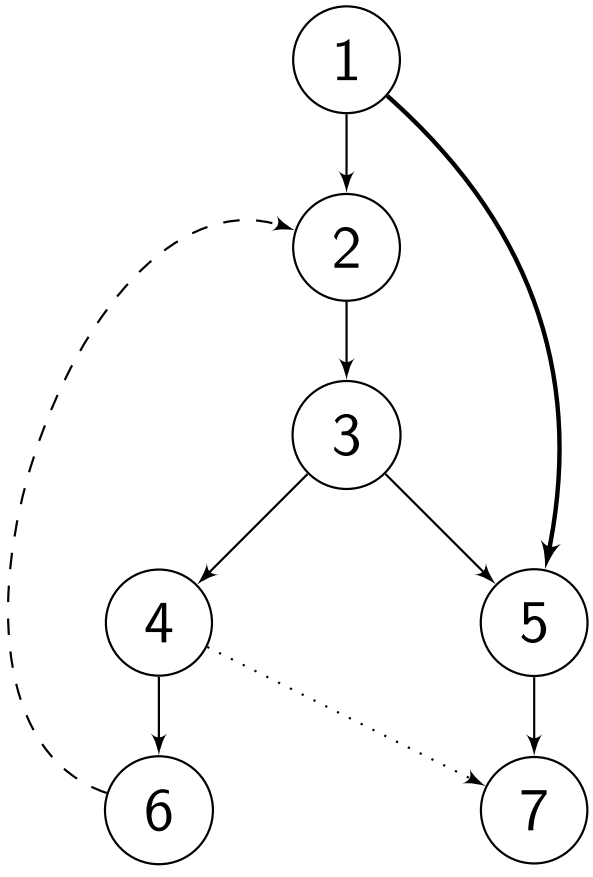
- 林伯禧
- 2021, 2022 TOI 二階
- 2022 ICPC 桃園站第六名
- 因為想學習圖論所以當圖論講師

大綱

圖的連通性

圖的連通性 – DFS 邊分類

- 樹邊：真正在 DFS 樹上的邊，從父親連往小孩
- 回邊：從子孫連回祖先的邊
- 前向邊：連向沒有直接親子關係的子孫的邊
- 交錯邊：連向非直系血親的邊



圖的連通性 – 無向圖的連通性

定義 (點連通度)

對於一個連通圖 $G = (V, E)$，定義它的點連通度 $\kappa(G)$ 為最少要移除幾個節點才可以使 G 不連通。

定義 (邊連通度)

對於一個連通圖 $G = (V, E)$，定義它的邊連通度 $\kappa'(G)$ 為最少要移除幾條邊才可以使 G 不連通。

我們稱一張圖 G 是**點-k-連通**若 $\kappa(G) \geq k$，同樣的，G 是**邊-k-連通**若 $\kappa'(G) \geq k$。在競賽中最常見的是 $k = 2$ 的情況，分別被稱為**點雙連通**以及**邊雙連通**。

圖的連通性 – 橋

定義

一條邊 e 被稱為**橋（bridge）**若圖 G 在移除 e 之後變得不連通。

圖的連通性 – 橋

定義

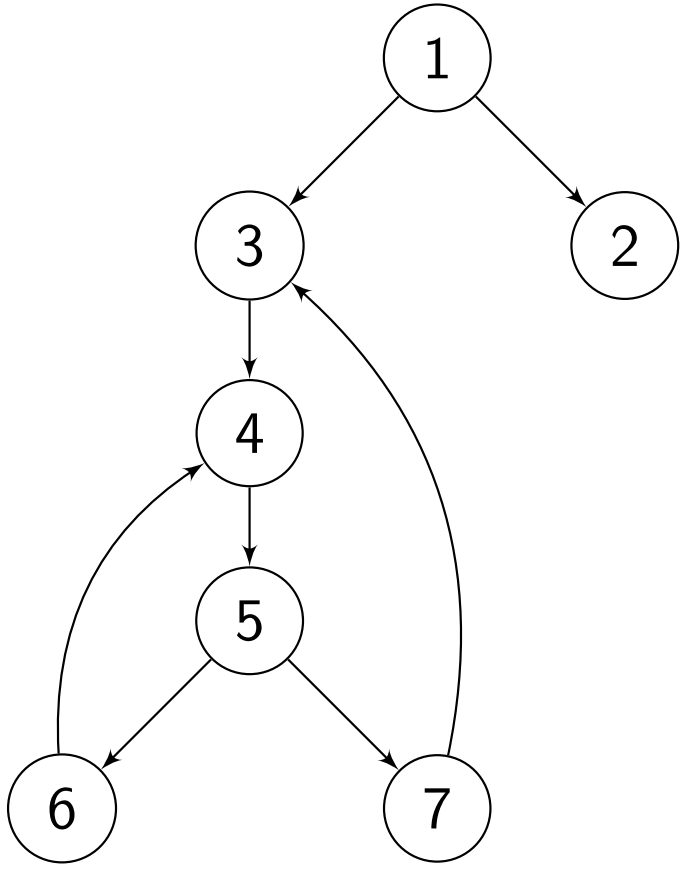
一條邊 e 被稱為**橋（bridge）**若圖 G 在移除 e 之後變得不連通。

Tarjan 算法：利用 DFS 樹找橋

圖的連通性 – 橋

找任一棵 DFS 樹

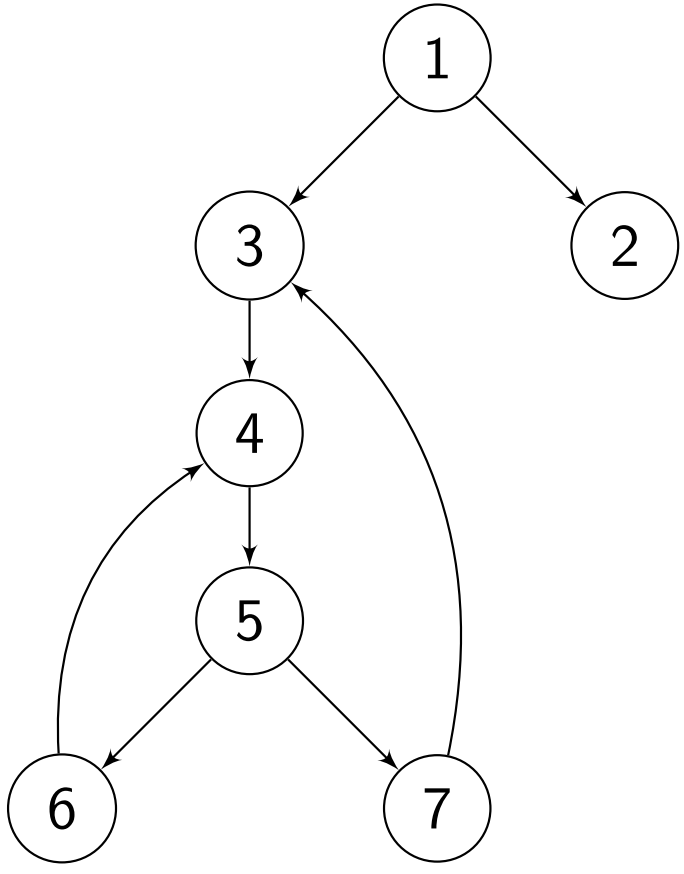
- 回邊：不是橋
- 樹邊：檢查兒子不透過父邊能不能到祖先



圖的連通性 – 橋

從節點 v 不經過父邊走到祖先的方法：

- v 直接有一條邊連到祖先
- v 先向下往 v 的小孩以及子孫走，再從子孫中爬回祖先



圖的連通性 – Tarjan 算法找橋

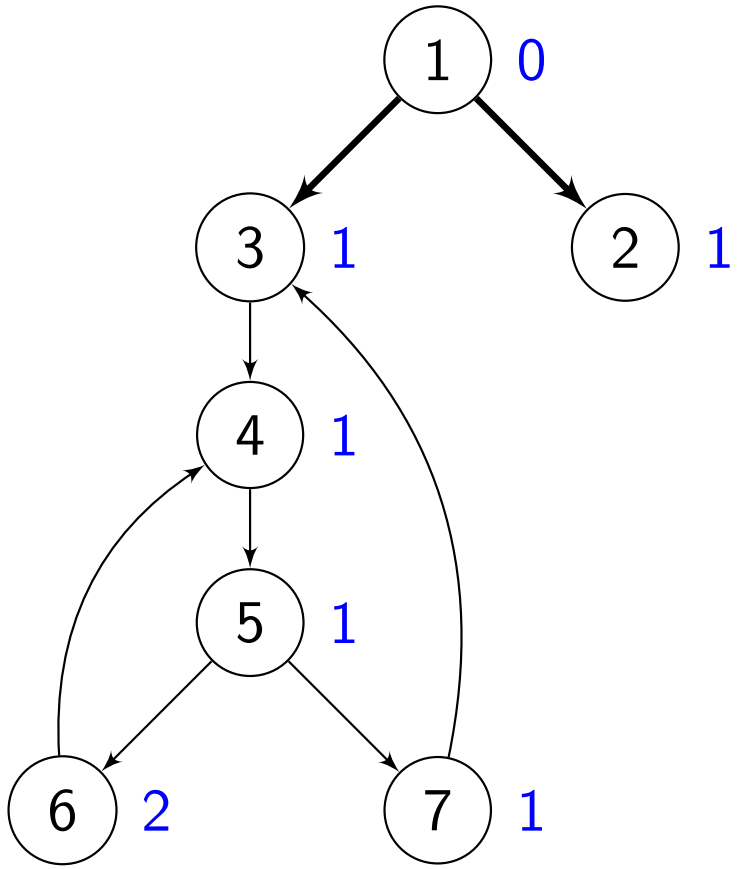
Tarjan 定義了以下的 low 函數：

定義

對於無向圖上的節點 v，$\mathrm{low}(v)$ 被定義為「在不透過父邊的情況下，最多經過一條回邊能夠到達**深度最淺**的祖先的深度」。

圖的連通性 – Tarjan 算法找橋

對於節點 v，如果 $\mathrm{low}(v) = \mathrm{depth}(v)$ 的話，就代表 v 在不依靠父邊的情況下永遠沒辦法走到它的祖先，此時 v 的父邊就會是橋



圖的連通性 – Tarjan 算法找橋

DFS 到邊 (v, u) 時：

1. u 還沒有被走過：此時 u 是 v 在 DFS 樹上的小孩，v 有機會透過 u 走到祖先，所以先對 u 做 DFS，再將 $\text{low}(v)$ 與 $\text{low}(u)$ 取 min
2. u 已經被走過了：此時 (v, u) 會是一條回邊 (因為無向圖只有樹邊與回邊)，那麼我們要檢查 u 的深度是否比 $\text{low}(v)$ 還要小，是的話就將 $\text{low}(v)$ 更新為 $\text{depth}(u)$

圖的連通性 – Tarjan 算法找橋

```
void dfs(int v, int p) {
  vis[v] = true;
  low[v] = depth[v] = ~p ? depth[p] + 1 : 0;
  for (int u : g[v]) {
    if (u == p) continue;
    if (!vis[u]) {
      dfs(u, v);
      low[v] = min(low[v], low[u]);
    }
    else {
      low[v] = min(low[v], depth[u]);
    }
  }
  if (low[v] == depth[v]) {
    // the edge (v, p) is a bridge if v is not the root vertex
  }
}
```

圖的連通性 – 邊雙連通

性質

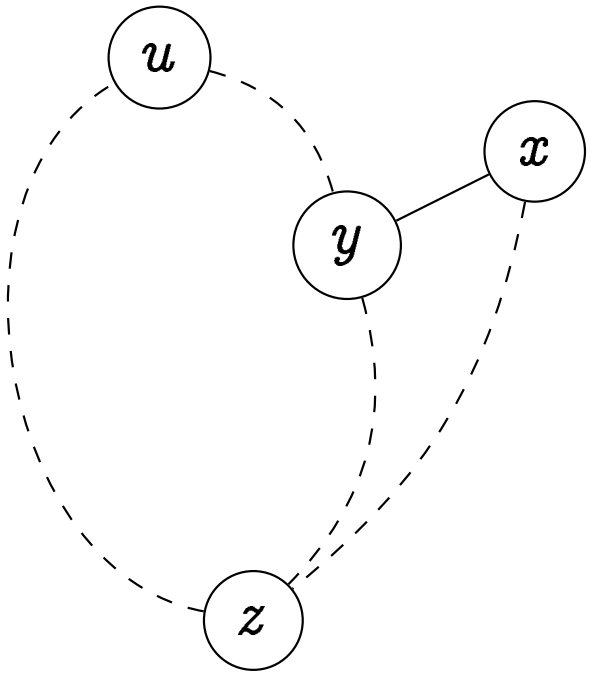
對於一張邊雙連通圖中的任意兩個節點 u, v，都存在至少兩條從 u 到 v 的路徑，且這兩條路徑沒有共用邊。

圖的連通性 – 邊雙連通

- 對於節點 u，我們顯然能找到一個包含它的簡單環，環上每一點都滿足這個性質。

圖的連通性 – 邊雙連通

- 對於節點 u，我們顯然能找到一個包含它的簡單環，環上每一點都滿足這個性質。
- 對於和這個環相鄰的所有點 x 也都存在一個包含 u, x 的簡單環。



圖的連通性 – 邊雙連通

- 對於節點 u，我們顯然能找到一個包含它的簡單環，環上每一點都滿足這個性質。
- 對於和這個環相鄰的所有點 x 也都存在一個包含 u, x 的簡單環。
- 和每個新的環相鄰的點又能再找到簡單環。以此類推，邊雙連通內的任意節點 v 都和 u 同屬於某個簡單環，環的兩側就是兩條沒有共用邊的路徑。

圖的連通性 – 邊雙連通

定理 (Robbin 定理)

對於一張無向圖 G，G 是邊雙連通若且唯若存在一種將 G 的邊定向的方式使得任兩個節點可以互相抵達。

圖的連通性 – 邊雙連通

題目 (Delivery Oligopoly)

有一張 N 點 M 邊的邊雙連通圖，現在要刪除一些邊，求最少要保留多少邊才能讓圖仍然保持邊雙連通，並輸出一組解。

- $N \leq 14$
- $M \leq \frac{N(N-1)}{2}$

圖的連通性 – 邊雙連通

枚舉點的子集，用 DP 計算需要幾條邊才會是邊雙連通。

圖的連通性 – 邊雙連通

枚舉點的子集，用 DP 計算需要幾條邊才會是邊雙連通。

假如我們先找圖中的一個小的邊雙連通子圖 (比如一個環)，每次多拿一些邊擴增子圖，直到所有點都被涵蓋，這個過程看起來就能用 DP 維護了。

圖的連通性 – 邊雙連通

枚舉點的子集，用 DP 計算需要幾條邊才會是邊雙連通。

假如我們先找圖中的一個小的邊雙連通子圖 (比如一個環)，每次多拿一些邊擴增子圖，直到所有點都被涵蓋，這個過程看起來就能用 DP 維護了。

每次要多拿的邊會是一條路徑，其中起點和終點位於原本的子圖上 (可能是同一點)，而其他點都在子圖以外，在選完這條路徑後，經過的點就會被加進邊雙連通子圖。

圖的連通性 – 邊雙連通

枚舉點的子集，用 DP 計算需要幾條邊才會是邊雙連通。

假如我們先找圖中的一個小的邊雙連通子圖 (比如一個環)，每次多拿一些邊擴增子圖，直到所有點都被涵蓋，這個過程看起來就能用 DP 維護了。

每次要多拿的邊會是一條路徑，其中起點和終點位於原本的子圖上 (可能是同一點)，而其他點都在子圖以外，在選完這條路徑後，經過的點就會被加進邊雙連通子圖。

所有的邊雙連通圖都能經由這個過程得到，因此我們一定能得到最佳解。

圖的連通性 – 邊雙連通

- 預處理每條從 u 到 v 並且包含 $mask$ 中所有點的路徑是否存在。複雜度 $O(N^3 2^N)$
- 枚舉每個點的子集和它最後加入的一條路徑，計算讓它是邊雙連通最少需要幾條邊。複雜度 $O(N^2 3^N)$

圖的連通性 – 邊雙連通分量

當原圖不是邊雙連通時，有時要找出每個「邊雙連通分量」

定義 (邊雙連通分量)

無向圖上「邊雙連通」的極大導出子圖（induced subgraph）稱為邊雙連通分量。

圖的連通性 – 邊雙連通分量

當原圖不是邊雙連通時，有時要找出每個「邊雙連通分量」

定義 (邊雙連通分量)

無向圖上「邊雙連通」的極大導出子圖（induced subgraph）稱為邊雙連通分量。

將圖上的所有橋移除後，每個連通分量都是邊雙連通分量

圖的連通性 – Tarjan 算法找邊雙連通分量

如果我們在使用 Tarjan 找橋的 DFS 過程中維護一個堆疊 (stack)，在每次進入一個節點的時候把該節點推入堆疊中，那當我們在節點 v 確定他的父邊是橋的時候，堆疊中從最上面一路取出，一直到節點 v，剛好就形成了 v 所在的邊雙連通分量！

圖的連通性 – Tarjan 算法找邊雙連通分量

```cpp
vector<vector<int>> bcc;
void dfs(int v, int p) {
  stk.push(v);
  vis[v] = true; low[v] = depth[v] = ~p ? depth[p] + 1 : 0;
  for (int u : g[v]) {
    // DFS and calculate low
  }
  if (low[v] == depth[v]) {
    bcc.emplace_back();
    while (stk.top() != v) {
      bcc.back().push_back(stk.top());
      stk.pop();
    }
    bcc.back().push_back(stk.top());
    stk.pop();
  }
}
```

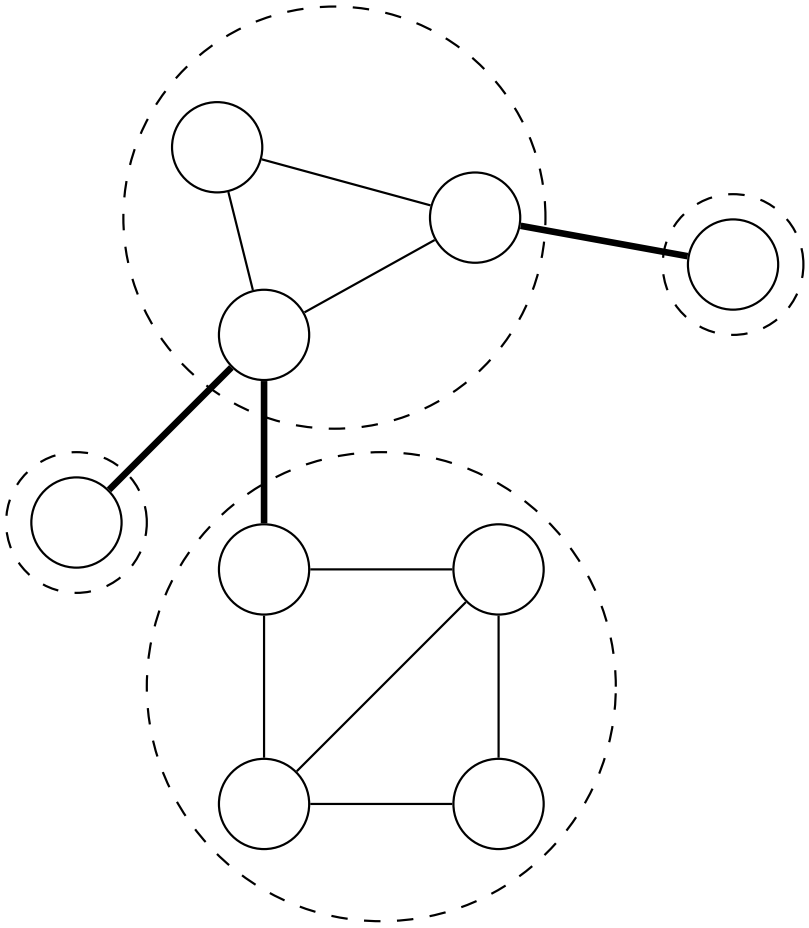
圖的連通性 – 邊雙連通分量

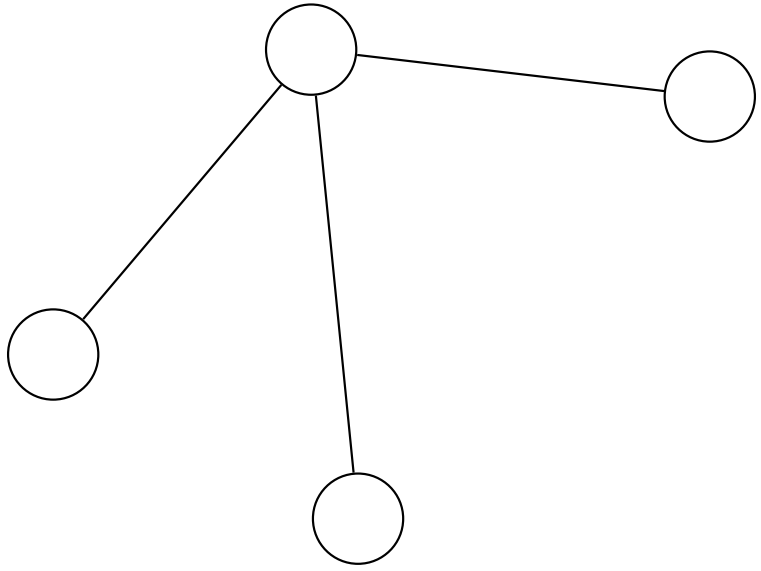
利用邊雙連通分量將連通圖簡化為樹

- 把每個邊雙連通分量都縮成點
- 對於原圖的每條橋，在新圖加入一條邊連接對應的兩個邊雙連通分量

新圖不會有環（否則會縮成一點）而且連通，因此就是一棵樹

圖的連通性 – 邊雙連通分量





圖的連通性 – 割點

定義

一個節點 v 被稱為**割點（cut vertex）**或**關節點（articulation vertex）**若圖 G 在移除 v 之後變得不連通。

圖的連通性 – 割點

定義

一個節點 v 被稱為**割點（cut vertex）**或**關節點（articulation vertex）**若圖 G 在移除 v 之後變得不連通。

修改 Tarjan 找橋的算法來找割點

圖的連通性 – 割點

一樣先找 DFS 樹。對於一個節點 v 以及其小孩 u，若 u 沒有辦法在不經過 v 的條件下到達 v 的祖先的話，那麼將節點 v 移除之後，u 與其子樹中的節點就沒有其他途徑能走到 v 之上，圖也就不連通了。

圖的連通性 – 割點

一樣先找 DFS 樹。對於一個節點 v 以及其小孩 u，若 u 沒有辦法在不經過 v 的條件下到達 v 的祖先的話，那麼將節點 v 移除之後，u 與其子樹中的節點就沒有其他途徑能走到 v 之上，圖也就不連通了。

沿用 low 函數的定義，當 $\text{low}(u) \geq \text{depth}(v)$ 的時候，就代表 u 的子樹最高也就爬到 v，也就表示 v 是一個割點。

圖的連通性 – 割點

一樣先找 DFS 樹。對於一個節點 v 以及其小孩 u，若 u 沒有辦法在不經過 v 的條件下到達 v 的祖先的話，那麼將節點 v 移除之後，u 與其子樹中的節點就沒有其他途徑能走到 v 之上，圖也就不連通了。

沿用 low 函數的定義，當 $\text{low}(u) \geq \text{depth}(v)$ 的時候，就代表 u 的子樹最高也就爬到 v，也就表示 v 是一個割點。

當 v 是根節點的時候，由於此時 v 並沒有祖先，因此 v 是割點的條件變成在 DFS 樹上有超過一個小孩 (移除 v 之後各個小孩皆不互相連通)。

圖的連通性 – Tarjan 算法找割點

```
void dfs(int v, int p) {
  vis[v] = true; low[v] = depth[v] = ~p ? depth[p] + 1 : 0;
  int ch_cnt = 0;
  for (int u : g[v]) {
    if (u == p) continue;
    if (!vis[u]) {
      ++ch_cnt;
      dfs(u, v);
      low[v] = min(low[v], low[u]);
      if (low[u] >= depth[v] && ~p) cut[v] = true;
    }
    else {
      low[v] = min(low[v], depth[u]);
    }
  }
  if (p == -1 && ch_cnt > 1) cut[v] = true;
}
```

圖的連通性 – 點雙連通

與邊雙連通相仿，點雙連通圖有以下的性質：

性質

對於一個點雙連通圖中的任兩個節點 u 以及 v，都存在至少兩條從 u 到 v，且這兩條路徑沒有共用點。

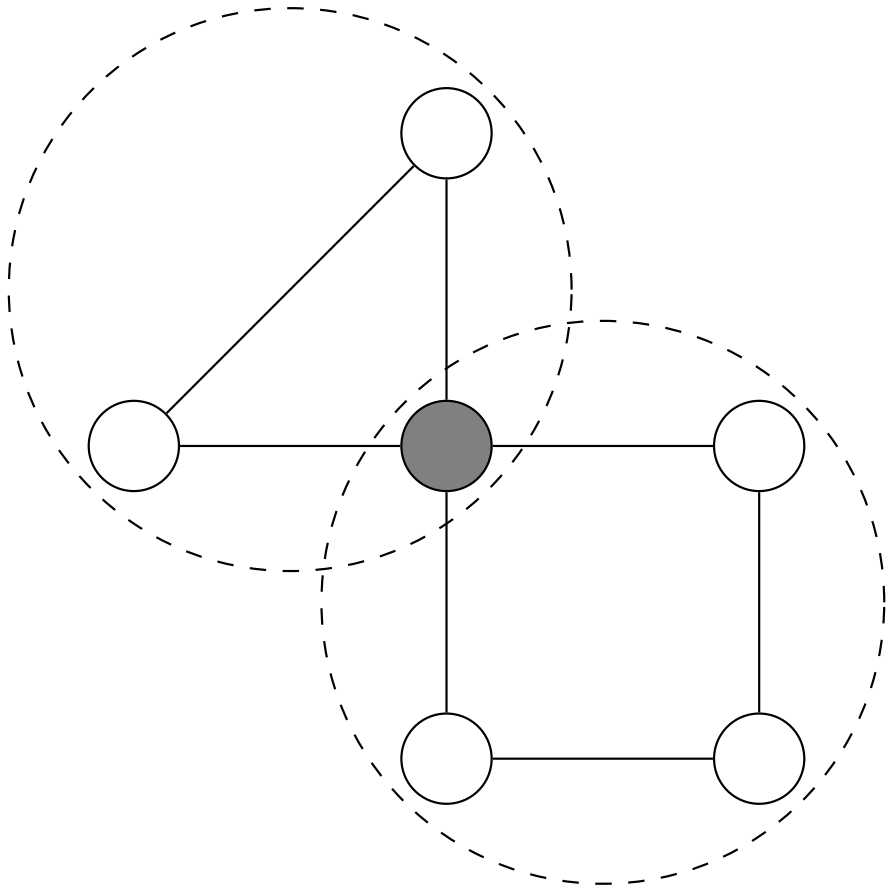
圖的連通性 – 點雙連通分量

定義 (點雙連通分量)

無向圖上「點雙連通」的極大導出子圖（induced subgraph）稱為點雙連通分量。

圖的連通性 – 點雙連通分量

與邊雙連通分量不同，不同點雙連通分量可能互相重疊



圖的連通性 – 點雙連通分量

一個點可能同時存在於很多點雙連通分量內，不過這種情形只會發生在割點。

一樣在 DFS 的時候維護一個堆疊，每次遇到一個新的點時便將它推入堆疊。當我們發現一個點的父節點是割點時，我們便從堆疊頂端取出在這個子節點上面的所有點，則這些點和這個割點就會剛好形成一個點雙連通分量。

圖的連通性 – Tarjan 算法找點雙連通分量

```
dfs(u, v);
low[v] = min(low[v], low[u]);
if (low[u] >= depth[v]) {
  bcc.emplace_back();
  while (stk.top() != u) {
    bcc.back().push_back(stk.top());
    stk.pop();
  }
  bcc.back().push_back(stk.top());
  stk.pop();
  bcc.back().push_back(v);
}
```

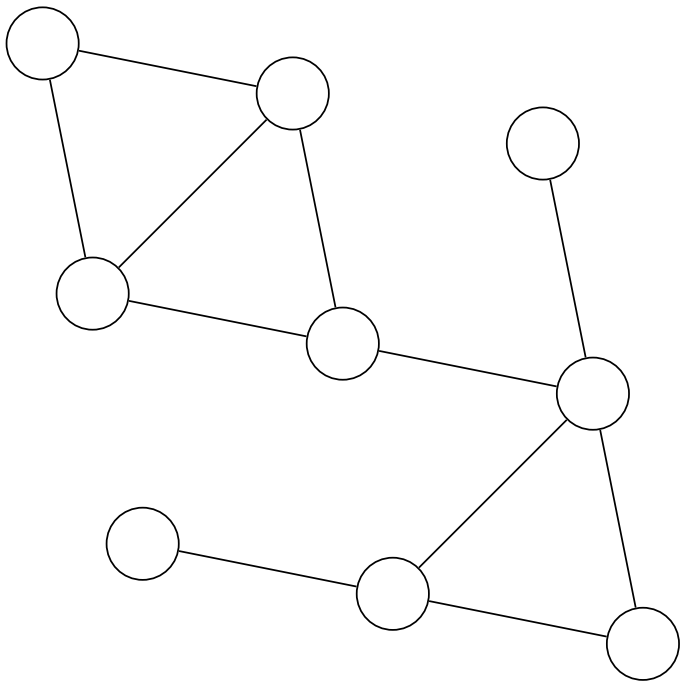
圖的連通性 – 點雙連通分量

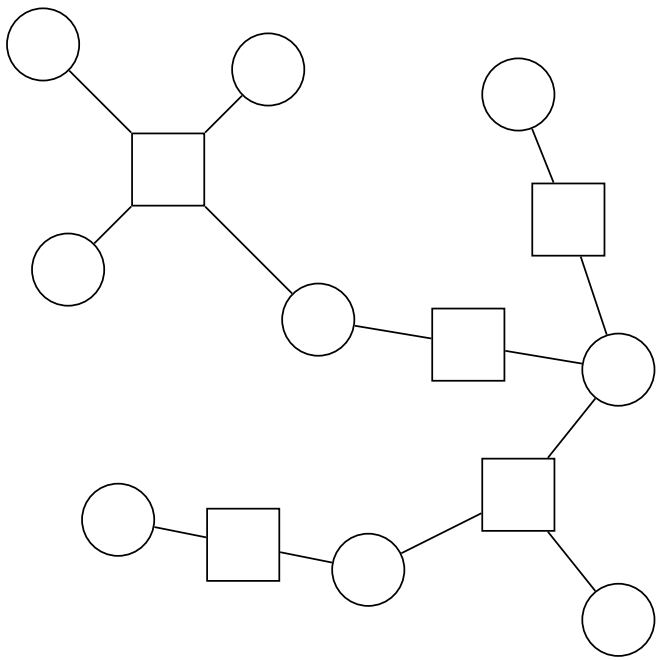
由於一個點可能會出現在很多分量之中，直接將圖轉換成樹不太容易，較便利的方法是建一棵「圓方樹」。

- 由圓點、方點和一些一端是圓點一端是方點的邊組成
- 新圖的每個圓點對應到原圖中的一個節點
- 新圖的每個方點對應到原圖中的一個點雙連通分量
- 對於每個點雙連通分量，把它對應的方點連到分量中每個節點的圓點

這樣一來，新圖就會是一棵樹，而一個圓點的鄰居就是包含它的每個連通分量。

圖的連通性 – 點雙連通分量





圖的連通性 – 點雙連通分量

題目 (Tourists)

有一張 N 點 M 邊的無向連通圖，每個節點 v 有權重 w_v。請支援下列兩種操作 (共 Q 筆):

- 將節點 a 的權重改為 b
- 詢問節點 a 到節點 b 的所有簡單路徑上，能經過的最小的點權。
- $N, M, Q \leq 10^5$

圖的連通性 – 點雙連通分量

- 從 a 走到 b 的路徑會依序經過特定的一些割點和點雙連通分量
- 在一個點雙連通分量中從 x 走到 y 時，對於同一個分量中的任一個節點 z，都一定存在 x 經過 z 再到 y 的簡單路徑，因此經過某個點雙連通分量時的最小點權就會是這個分量的節點之中的最小點權

圖的連通性 – 點雙連通分量

- 從 a 走到 b 的路徑會依序經過特定的一些割點和點雙連通分量
- 在一個點雙連通分量中從 x 走到 y 時，對於同一個分量中的任一個節點 z，都一定存在 x 經過 z 再到 y 的簡單路徑，因此經過某個點雙連通分量時的最小點權就會是這個分量的節點之中的最小點權

先對原圖建圓方樹，將圓點的點權設為對應的節點點權，方點的點權設為對應的連通分量中的最小點權，以 multiset 維護好這些值，操作就會變成詢問樹上一條路徑的最小點權，可以簡單的用後面會提到的樹鏈剖分處理這種詢問

圖的連通性 – 點雙連通分量

- 從 a 走到 b 的路徑會依序經過特定的一些割點和點雙連通分量
- 在一個點雙連通分量中從 x 走到 y 時，對於同一個分量中的任一個節點 z，都一定存在 x 經過 z 再到 y 的簡單路徑，因此經過某個點雙連通分量時的最小點權就會是這個分量的節點之中的最小點權

先對原圖建圓方樹，將圓點的點權設為對應的節點點權，方點的點權設為對應的連通分量中的最小點權，以 multiset 維護好這些值，操作就會變成詢問樹上一條路徑的最小點權，可以簡單的用後面會提到的樹鏈剖分處理這種詢問

複雜度 $O((N+M)+(N+Q)\log^2 N)$

圖的連通性 – 有向圖的連通性

定義

一個有向圖為**弱連通（weakly connected）**若將所有的有向邊都換為無向邊後圖為連通。

定義

一個有向圖為**強連通（strongly connected）**若對於任兩個節點 u, v 都存在一條從 u 走到 v 的路徑以及一條從 v 走到 u 的路徑。

圖的連通性 – 強連通分量

定義 (強連通分量)

有向圖上「強連通」的極大導出子圖（induced subgraph）稱為強連通分量。

圖的連通性 – 強連通分量

定義 (強連通分量)

有向圖上「強連通」的極大導出子圖（induced subgraph）稱為強連通分量。

- Tarjan 算法
- Kosaraju 算法

圖的連通性 – Kosaraju 演算法

用好的順序在圖上 DFS 的話，每次走到的節點們就會各自形成強連通分量

以有向無環圖（Directed Acyclic Graph, DAG）為例，根據定義，每個節點會自成一個強連通分量。此時如果照著拓樸順序的反序進行 DFS 的話，每次都只會搜集到恰一個節點（前面的人都被拜訪過了），因此，這樣所求出的 SCC 就是我們想得到的結果

圖的連通性 – Kosaraju 演算法

對於一般的有向圖上的節點 x 與 y，如果 x 能到達 y 且 y 不能到達 x 的話，那麼 x 與 y 一定屬於不同的強連通分量，如果這時 x 比 y 早 DFS 到的話，那就會誤把 y 與 x 放在同一個 SCC 裡。

因此我們希望對於所有這樣的點對，y 都要在 x 之前被 DFS 拜訪到。

圖的連通性 – Kosaraju 演算法

1. 建立有向圖 G 的**反圖（transpose graph）** G'
2. 在 G' 上進行 DFS 並記錄下所有節點離開的時間
3. 將節點由離開時間大到小排列，按照這個順序在原圖上 DFS
4. 收集每次走到的節點形成一個 SCC

圖的連通性 – Kosaraju 演算法

對於 x 可以走到 y 但 y 不能走到 x 的點對，在 G' 上會變成 y 可以走到 x 而 x 不能走到 y。考慮在反圖上 DFS 到 y 時 x 的狀態：

- 如果 x 已經被拜訪過了：那顯然 x 的離開時間小於 y 的離開時間。
- x 還未被拜訪：那從 y 開始 DFS 會經過 x，由於 y 要等 x 離開才能離開，因此 x 的離開時間依然小於 y 的離開時間。

因此 x, y 會被正確分到不同的 SCC

圖的連通性 – Kosaraju 演算法

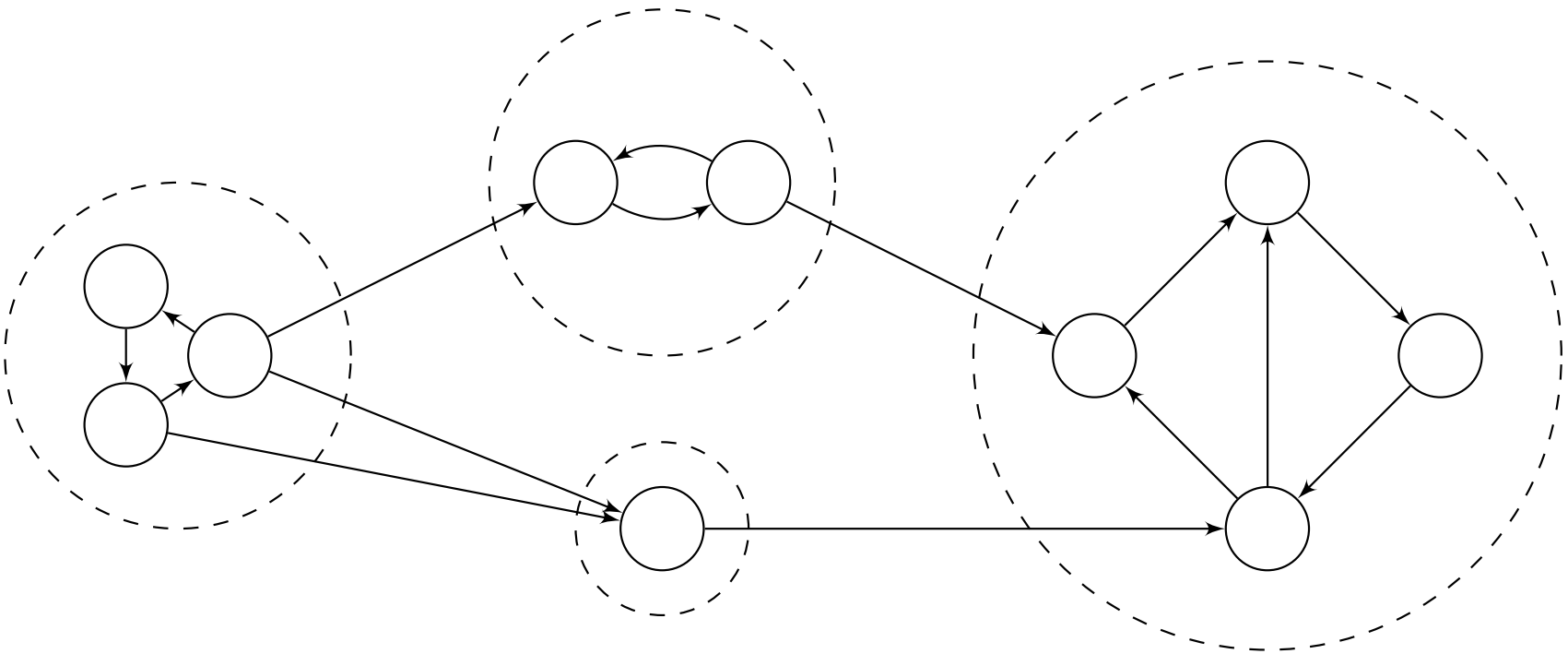
```
void rev_dfs(int v) {
  vis[v] = true;
  for (int u : rev_g[v]) {
    if (!vis[u])
      rev_dfs(u);
  }
  order.push_back(v);
}
void dfs(int v, int sid) {
  scc[v] = sid;
  for (int u : g[v]) {
    if (scc[u] == -1)
      dfs(u, sid);
  }
}
```

圖的連通性 – Kosaraju 演算法

```
void Kosaraju(int n) {
  for (int i = 0; i < n; ++i) {
    if (!vis[i])
      rev_dfs(i);
  }
  int scc_cnt = 0;
  for (int i = n - 1; i >= 0; --i) {
    int v = order[i];
    if (scc[v] == -1) {
      dfs(v, scc_cnt);
      ++scc_cnt;
    }
  }
}
```

圖的連通性 – 強連通分量

如果把強連通分量當作節點，那麼形成的圖會是一個 DAG



圖的連通性 – 2-SAT

題目 (2-SAT(2-satisfiability))

有 N 個布林變數 $X_1, X_2, \ldots, X_N$ 以及一條形如

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \ldots \wedge (a_M \vee b_M)$$

的式子，其中 a_i 與 b_i 都是某個布林變數 X_j 或它取 not 的結果 $\neg X_j$。請將 $X_1, X_2, \ldots, X_N$ 都賦值使得前式的結果為真。

圖的連通性 – 2-SAT

對於每個 $(a_i \vee b_i)$ 的句子 (clause)，如果 a_i 為 `False` 的話，那 b_i 一定要是 `True`，反之亦然。

構造一個 $2N$ 個點的有向圖，每個節點分別代表 X_i 以及 $\neg X_i$，對於 $(a_i \vee b_i)$，我們連接一條 $\neg a_i \to b_i$ 以及 $\neg b_i \to a_i$ 的有向邊。那麼對於圖上的兩個節點 x 與 y，如果 x 可以走到 y 的話，在原本的世界裡就代表 $x \implies y$。

圖的連通性 – 2-SAT

如果發現 X_i 與 $\neg X_i$ 位於同一個強連通分量，就代表沒有任何解，否則一定有解。

由於同一個強連通分量裡的變數一定同時為真或同時為假，可以將 SCC 縮點後對它們賦值。找解時可以根據拓樸排序由後往前將還不確定的 SCC 設為 `True`，同時把該 SCC 中變數取 `not` 所在的 SCC 也賦值，以此類推。由於是由拓樸排序後面的開始做，因此不會發生解到一半產生矛盾的情形。

樹論

樹論 – 樹壓平

- 在 DFS 的時候記下每個節點 v 進入的時間點 L_v 以及結束的時間點 R_v
- 對於 v 子樹中的節點 u，由於 u 會在 v 之後才被拜訪到，所以 $L_v \leq L_u$
- v 會在整棵子樹拜訪後才結束，所以 $L_u < R_v$
- 如果以 L_v 當作 v 的新編號的話，v 的子樹恰好包含所有編號在 L_v 到 R_v 中的節點，可以將子樹操作轉換為序列上的區間操作
- 由於每個點只會在序列中出現一次，樹壓平通常比尤拉迴路更適合處理修改的操作

樹論 – 樹壓平

- 記錄一個全域變數 `cnt`，初始為 0，由根開始 DFS
- 遞迴到節點 v 時紀錄 `L[v] = cnt++;`
- 離開前紀錄 `R[v] = cnt;`

就可以得到每個節點左閉右開的子樹區間

樹論 – 樹壓平

題目 (子樹加值&子樹和)

給定一棵有根樹，每個點有預設為 0 的點權，請支援下列兩種操作：

1. 將以節點 x 為根的子樹裡每個點的點權加上 k
2. 詢問以節點 x 為根的子樹的點權和

樹論 – 樹壓平

題目 (子樹加值&子樹和)

給定一棵有根樹，每個點有預設為 0 的點權，請支援下列兩種操作：

1. 將以節點 x 為根的子樹裡每個點的點權加上 k
2. 詢問以節點 x 為根的子樹的點權和

照著前面的方法，將第一個操作轉換為序列上的區間加值，第二個操作轉換為序列上的區間和詢問，就變成我們所熟悉的區間操作問題了

樹論 – 全方位木 DP

題目 (Alternating Tree)

給定一棵 N 個節點的樹，每個點有個權值 a_i。假如 u 到 v 的最短路徑依序經過 $v_1, v_2, \ldots, v_k$，定義 $f(u,v) = \sum_{i=1}^{k} (-1)^{i+1} \cdot a_{v_i}$，請計算所有點對的 $f(u,v)$ 的總和。

- $N \leq 2 \cdot 10^5$

樹論 – 全方位木 DP

先看 $\sum_{v \in V} f(1, v)$

以 1 為根，令 dp_v 為從 v 往子樹中每個點走的路徑的總和，$size_v$ 為 v 的子樹大小，利用 $dp_v = a_v \cdot size_v - \sum_{pa_u = v} dp_u$ 可以在一次 DFS 算完

樹論 – 全方位木 DP

接下來處理根的兒子 x。x 的子樹都不需要重複 DP，只需要計算從 x 往上走的那些路徑就好

計算 $dp'_1 = dp_1 - (a_1 \cdot size_x - dp_x)$，把父節點當作一個一般的子樹合併

繼續往下 DFS，每到一個點就把它父親的貢獻加進去，完成後就得到從每個點出發的路徑價值和了！

樹論 – 全方位木 DP

- 需要計算一個點到其他所有點的某個值
- 對於根節點可以用 DP 求值
- 對於一個點可以快速扣掉某個子樹的貢獻
- 對於一個點可以快速新增來自父節點的貢獻

樹論 – 樹背包

題目 (Werewolves)

給定一棵 N 個點的樹，每個點有一個顏色，請問有多少個連通子圖中存在絕對眾數，也就是某個顏色的出現次數大於子圖大小的一半。

- $N \leq 3000$

樹論 – 樹背包

對每個顏色分別計算他是眾數的連通子圖數量，令該顏色的節點權重為 1，其他點為 -1，那麼相當於找出權重和是正數的連通子圖數量

樹論 – 樹背包

對每個顏色分別計算他是眾數的連通子圖數量，令該顏色的節點權重為 1，其他點為 -1，那麼相當於找出權重和是正數的連通子圖數量

暴力：任取一點當根，設 $dp_{i,j}$ 表示取了 i 及子樹中一些連通的點後權重為 j 的方法數，用背包的方式轉移

樹論 – 樹背包

```
int dp[N][2 * N + 1], tmp[2 * N + 1], zro = N;
void dfs(int v, int p) {
  dp[v][zro + w[v]] = 1;
  for (int u : g[v]) {
    if (u == p) continue;
    dfs(u, v);
    for (int i = 0; i <= 2 * N; ++i)
      tmp[i] = dp[v][i];
    for (int i = -N; i <= N; ++i)
      for (int j = max(-N - i, -N); j <= min(N - i, N); ++j)
        tmp[zro + i + j] += dp[v][zro + i] * dp[u][zro + j];
    for (int i = 0; i <= 2 * N; ++i)
      dp[v][i] = tmp[i];
  }
}
```

樹論 – 樹背包

最多可能有 N 種顏色，而每次需要花 $O(N^3)$ 的時間，複雜度是 $O(N^4)$，顯然是不夠快的

樹論 – 樹背包

大小為 x 的節點總和只會在 $\pm x$ 之間，枚舉總和時可以只枚舉這個範圍

每次合併兩塊時花的次數是兩塊的大小相乘，換句話說，每個點對只會在合併的時候花到常數次的次數，而樹上的點對數就只有 $O(N^2)$ 個，因此複雜度變成 $O(N^3)$，但還是不夠快

樹論 – 樹背包

假設某個顏色的節點只有 x 個，對這個顏色有用的總和範圍就只有 $\pm x$，可以再減少枚舉總和的時間

樹論 – 樹背包

假設某個顏色的節點只有 x 個，對這個顏色有用的總和範圍就只有 $\pm x$，可以再減少枚舉總和的時間

複雜度會是 $O(N^2)$！

樹論 – 樹背包

```
for (int i = -min(num_cur_color, sz[v]); \
  i <= min(num_cur_color, sz[v]); ++i) {
  for (int j = max({-num_cur_color, \
    -num_cur_color - i, -sz[u]}); \
    j <= min({num_cur_color, num_cur_color - i, \
    sz[u]}); ++j) {
    tmp[zro + i + j] += dp[v][zro + i] * dp[u][zro + j];
  }
}
sz[v] += sz[u];
```

樹論 – 樹背包

假設這次 DFS 的顏色包含 K 個節點，把子樹大小不到 K 的當作白點，其餘的當作黑點，分別來檢查複雜度

樹論 – 樹背包

假設這次 DFS 的顏色包含 K 個節點，把子樹大小不到 K 的當作白點，其餘的當作黑點，分別來檢查複雜度

- 發生在白點的更新採用前面的分析手法，因為每個節點只會和少於 K 個點合併，所以這部分總共花了 $O(NK)$ 的時間

樹論 – 樹背包

假設這次 DFS 的顏色包含 K 個節點，把子樹大小不到 K 的當作白點，其餘的當作黑點，分別來檢查複雜度

- 發生在白點的更新採用前面的分析手法，因為每個節點只會和少於 K 個點合併，所以這部分總共花了 $O(NK)$ 的時間
- 每個節點往上爬的過程中最多只會有一次從白色走到黑色，在這個祖先的更新次數也至多 K 次，所以也花了 $O(NK)$ 的時間

樹論 – 樹背包

接著看只包含黑點的部分，現在每個白點會讓自己第一個黑色祖先多儲存一個值，所有黑點中一共儲存了 N 個值

當我們將一個黑點和父節點合併時，子節點儲存了 K 個值，假如父親原本有 x 個值，那需要花 $O(xK)$ 的時間將他們合併成 K 個存進父節點，而子節點就不用管了

樹論 – 樹背包

接著看只包含黑點的部分，現在每個白點會讓自己第一個黑色祖先多儲存一個值，所有黑點中一共儲存了 N 個值

當我們將一個黑點和父節點合併時，子節點儲存了 K 個值，假如父親原本有 x 個值，那需要花 $O(xK)$ 的時間將他們合併成 K 個存進父節點，而子節點就不用管了

這一個操作也可以想成我們把 x 個值各花 $O(K)$ 的時間「消滅」了，因此全部消滅掉頂多也花 $O(NK)$ 的時間

樹論 – 樹背包

接著看只包含黑點的部分，現在每個白點會讓自己第一個黑色祖先多儲存一個值，所有黑點中一共儲存了 N 個值

當我們將一個黑點和父節點合併時，子節點儲存了 K 個值，假如父親原本有 x 個值，那需要花 $O(xK)$ 的時間將他們合併成 K 個存進父節點，而子節點就不用管了

這一個操作也可以想成我們把 x 個值各花 $O(K)$ 的時間「消滅」了，因此全部消滅掉頂多也花 $O(NK)$ 的時間

總複雜度是 $O(\sum NK_i) = O(N \cdot \sum K_i) = O(N^2)$，就能通過這題了

樹論 – 樹上啟發式合併

有些時候，子樹中擁有的資訊太多，我們沒辦法只用幾個數字來保留這些訊息，所以只好暴力的多跑幾次

但為了節省時間，我們可以沿用一個子樹存好的資訊，並且很直覺的選擇最大的那棵子樹，以節省更多時間，這個技巧就稱作樹上啟發式合併（DSU on Tree）

樹論 – 樹上啟發式合併

題目 (Lomsat gelral)

給定一棵有根樹，每個節點有一個顏色，求樹中每個子樹出現最多次的顏色編號的和。

樹論 – 樹上啟發式合併

對每個點 DFS 一次他的子樹，用陣列存下每種顏色的出現次數，計算答案。複雜度 $O(N^2)$

樹論 – 樹上啟發式合併

對每個點 DFS 一次他的子樹，用陣列存下每種顏色的出現次數，計算答案。複雜度 $O(N^2)$

如果在 DFS 一個子樹前，先把那個點的兒子都算好，並把子樹大小最大的兒子留到最後計算，算完後不要清空次數，就不用再 DFS 一次那棵子樹了

樹論 – 樹上啟發式合併

對每個點 DFS 一次他的子樹，用陣列存下每種顏色的出現次數，計算答案。複雜度 $O(N^2)$

如果在 DFS 一個子樹前，先把那個點的兒子都算好，並把子樹大小最大的兒子留到最後計算，算完後不要清空次數，就不用再 DFS 一次那棵子樹了

這樣做之後，複雜度將會變成 $O(N \log N)$！

樹論 – 樹上啟發式合併

將所有樹邊都歸類為「輕邊」或是「重邊」，每個節點會往其**有最大子樹大小的小孩**連一條重邊，其他的則連輕邊

從一個節點往根走時，每遇到一條輕邊代表子樹大小至少翻倍，因此對於一棵有 N 個節點的樹，其根節點到樹上任意節點的輕邊數量不會超過 $\log N$ 條

樹論 – 樹上啟發式合併

將所有樹邊都歸類為「輕邊」或是「重邊」，每個節點會往其**有最大子樹大小的小孩**連一條重邊，其他的則連輕邊

從一個節點往根走時，每遇到一條輕邊代表子樹大小至少翻倍，因此對於一棵有 N 個節點的樹，其根節點到樹上任意節點的輕邊數量不會超過 $\log N$ 條

根據我們前面的作法，一個節點只有在往上爬遇到輕邊時才會被重複 DFS 到，因此複雜度為 $O(N \log N)$

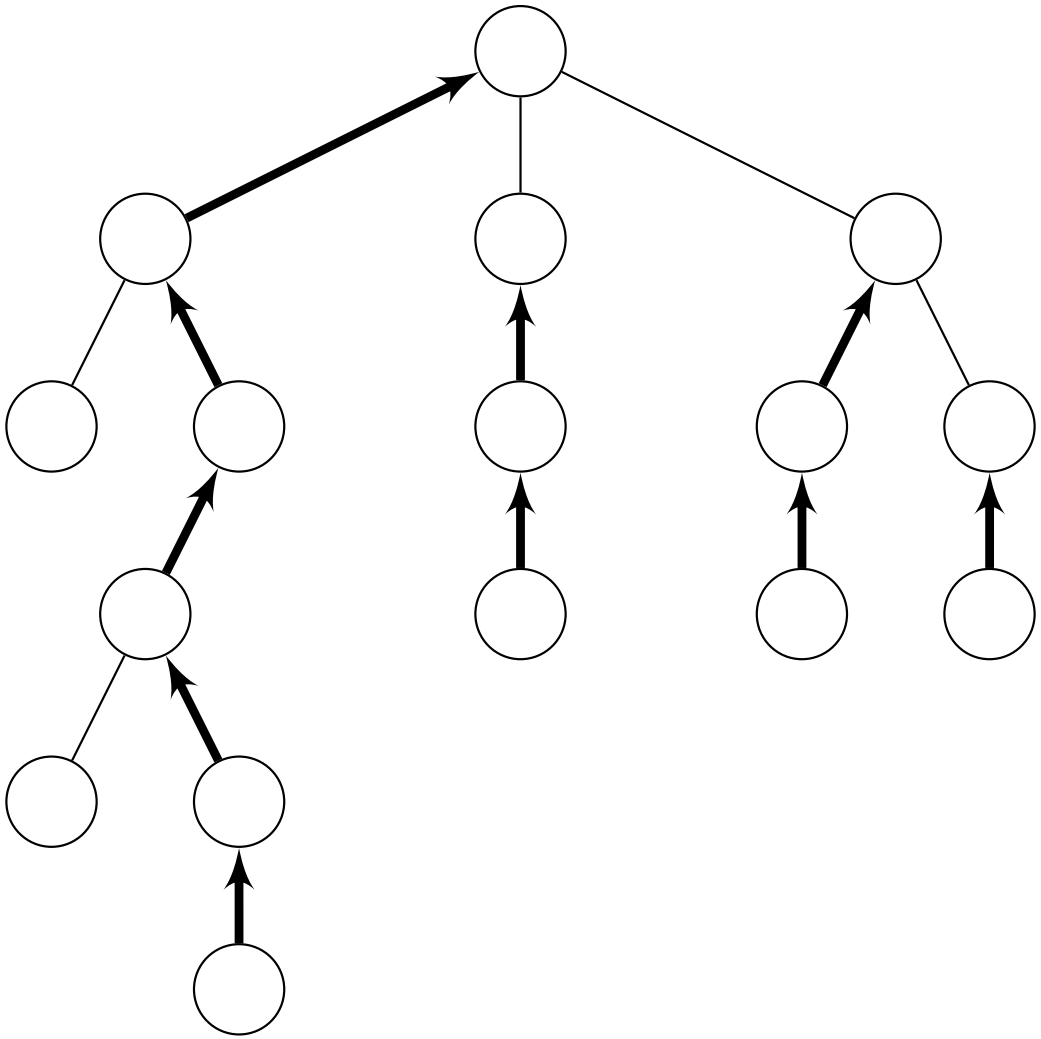
樹論 – 樹鏈剖分

樹鏈剖分主要用來處理帶修改的路徑問題

核心想法是將樹上的點分成若干個由祖先到子孫的「鏈」，只要 DFS 的時候優先順著鏈走，鏈上的節點與邊在樹壓平序列上就會形成一個**連續的區間**

只要可以保證從點 u 到點 v 所經過的「鏈」不會太多，就能把修改與查詢的複雜度變成
$O(u$ 到 v 鏈的數量$) \times$ 資料結構的複雜度

樹論 – 樹鏈剖分



樹論 – 輕重鏈剖分

輕重鏈剖分（Heavy-Light Decomposition, HLD）將所有樹邊都歸類為「輕邊」或是「重邊」，輕重邊的定義和樹上啟發式合併時相同

一個節點 v 到根節點所需要經過的輕邊數量為 $O(\log N)$，且只有在經過輕邊時才會發生「跳鏈」的情形

樹論 – 輕重鏈剖分

利用輕重鏈剖分可以將樹上的路徑操作轉為序列的區間操作

- 修改單一邊權 $\rightarrow$ 單點改值
- 修改一條路徑 $\rightarrow$ $O(\log N)$ 次區間修改
- 查詢一條路徑 $\rightarrow$ $O(\log N)$ 次區間詢問

樹論 – 重心

定義

對於一棵 N 個節點的樹，一個節點 v 被稱作重心（centroid）若移除 v 之後最大連通塊的大小不超過 $\lfloor \frac{N}{2} \rfloor$。

如果令 $d(v)$ 為樹上每一個節點到 v 的距離和的話，那 $d(v)$ 最小的節點 v 就會是重心。否則存在一個把 v 拔掉之後的連通塊大小超過 $\lfloor \frac{N}{2} \rfloor$，此時如果將 v 往那個連通塊走一步的話 $d(v)$ 就會變小

基於這個原因一棵樹上也至多只有兩個重心，而且如果有的話那兩個重心就會相鄰。

樹論 – 重心分治

處理一棵樹的問題時，有時可以先考慮所有跟節點 v 有關的答案，接著將 v 移除之後遞迴計算剩下的連通塊

樹論 – 重心分治

處理一棵樹的問題時，有時可以先考慮所有跟節點 v 有關的答案，接著將 v 移除之後遞迴計算剩下的連通塊

令 $T(N)$ 為計算一個 N 個節點的樹的答案所需的時間、$f(N)$ 為計算跟 v 有關的答案所需的時間，那麼有

$$T(N) = \sum T(s_i) + f(N)$$

其中 s_i 為移除 v 之後的子樹大小

樹論 – 重心分治

處理一棵樹的問題時，有時可以先考慮所有跟節點 v 有關的答案，接著將 v 移除之後遞迴計算剩下的連通塊

令 $T(N)$ 為計算一個 N 個節點的樹的答案所需的時間、$f(N)$ 為計算跟 v 有關的答案所需的時間，那麼有

$$T(N) = \sum T(s_i) + f(N)$$

其中 s_i 為移除 v 之後的子樹大小

如果每次選擇重心當作 v，在 $f(N) = O(N)$ 時，根據主定理 $T(N)$ 就會是 $O(N \log N)$

樹論 – 重心剖分

更常見的應用是建造**重心樹**：

對於一個 N 個節點的樹我們可以在 $O(N \log N)$ 的時間內找出它的重心樹：

1. 先找出樹重心 c 設為重心樹的根
2. 將重心移除之後遞迴每個連通塊並將 c 設為連通塊的重心在重心樹上的父節點
3. 重複到每個點都加到父節點上

樹論 – 重心剖分

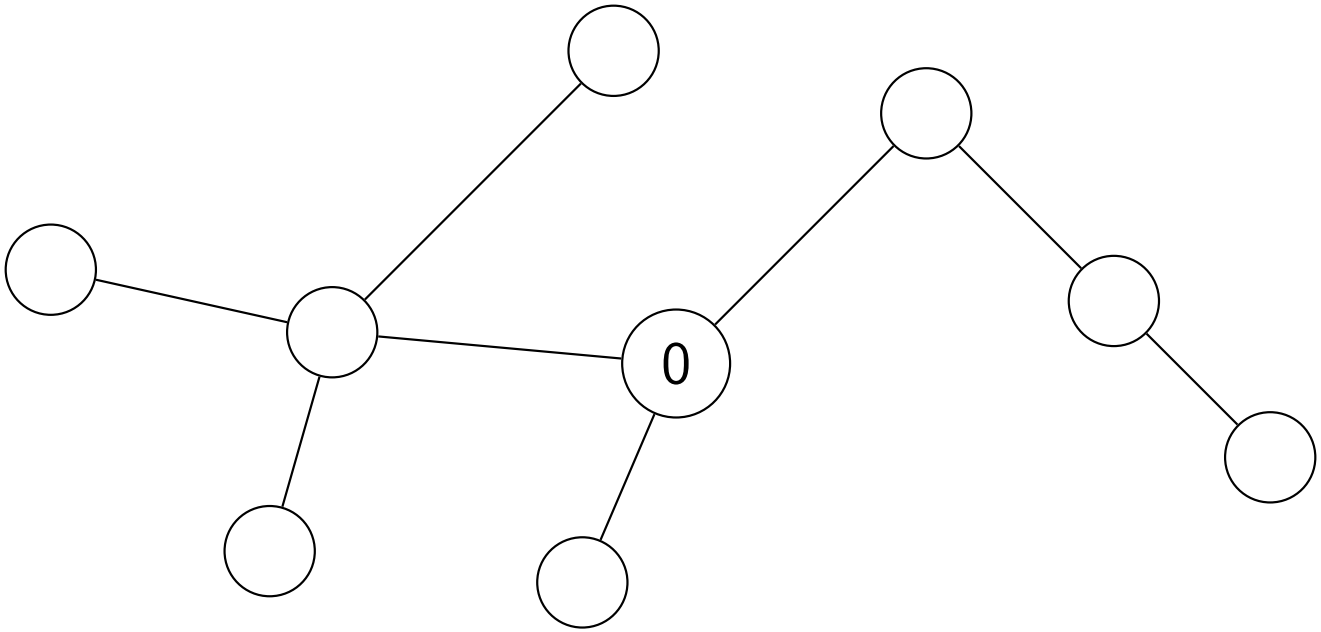
更常見的應用是建造**重心樹**：

對於一個 N 個節點的樹我們可以在 $O(N \log N)$ 的時間內找出它的重心樹：

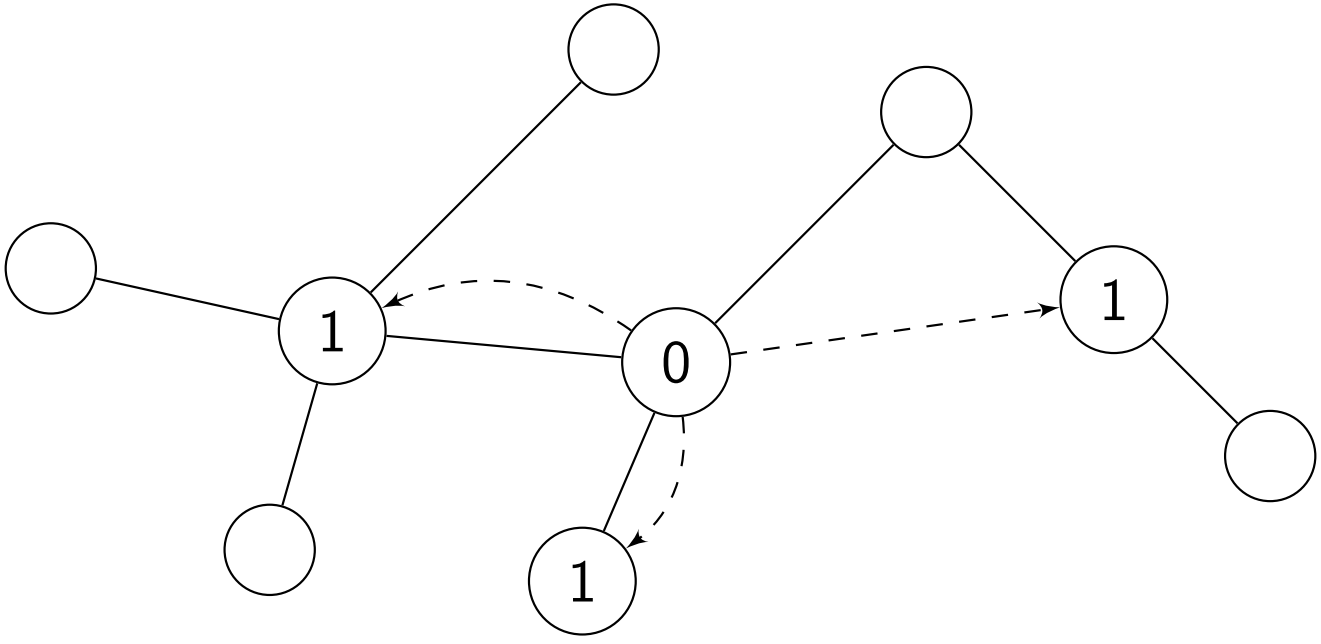
1. 先找出樹重心 c 設為重心樹的根
2. 將重心移除之後遞迴每個連通塊並將 c 設為連通塊的重心在重心樹上的父節點
3. 重複到每個點都加到父節點上

重心樹的好處就是由於每次遞迴連通塊的點數都會至少減半，因此它的深度是 $O(\log N)$。維護每個重心樹子樹的資訊，在查詢某個節點到其他點的資訊時沿著重心樹的父節點慢慢往上爬，通常能有好的複雜度

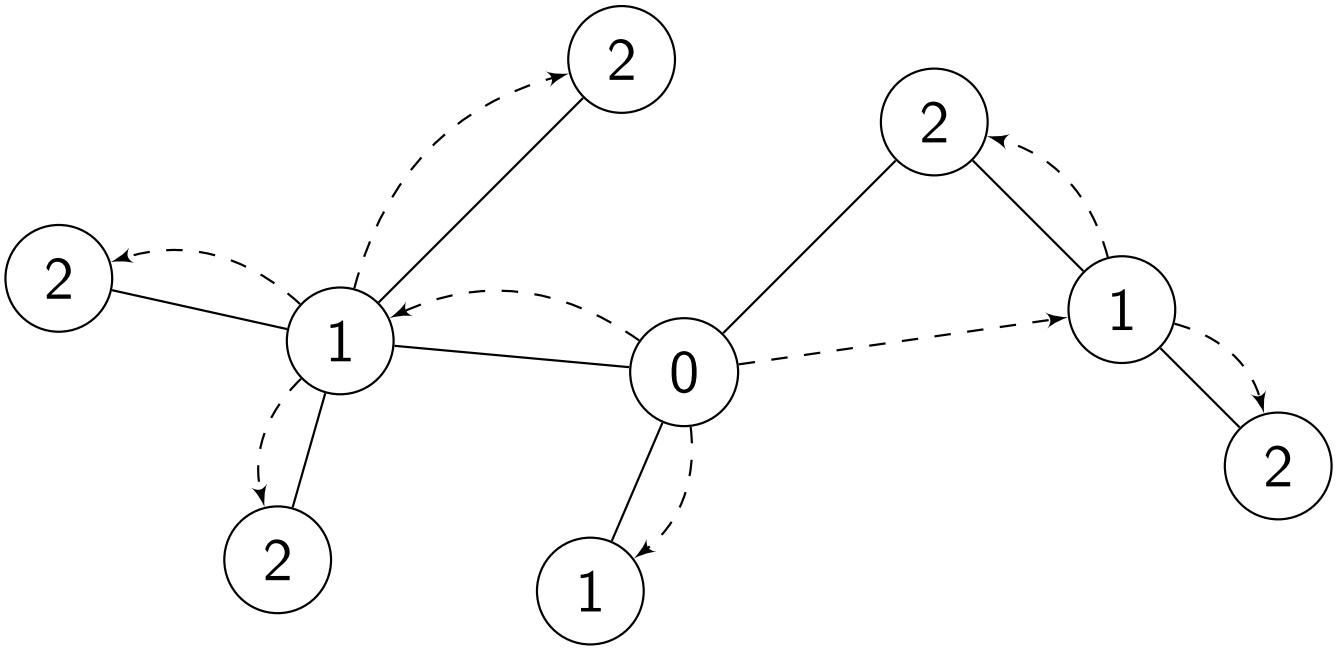
樹論 – 重心剖分



樹論 – 重心剖分



樹論 – 重心剖分



樹論 – 重心剖分

```
void centroid_decomp(int v, int p) {
  dfs_sz(v, -1);
  int c = dfs_cen(v, -1, sz[v]);
  vis[c] = true;
  pa[c] = p; // parent on centroid tree
  for (int u : g[c]) {
    if (vis[u]) continue;
    centroid_decomp(u, c);
  }
}
```

樹論 – 重心剖分

```
void dfs_sz(int v, int p) {
  sz[v] = 1;
  for (int u : g[v]) {
    if (vis[u] || u == p) continue;
      dfs_sz(u, v);
      sz[v] += sz[u];
  }
}
int dfs_cen(int v, int p, int tot) {
  for (int u : g[v]) {
    if (vis[u] || u == p) continue;
    if (sz[u] * 2 > tot)
      return dfs_cen(u, v, tot);
  }
  return v;
}
```

樹論 – 重心剖分

題目 (Xenia and Tree)

給定一棵 N 個節點的樹，進行 Q 筆操作，操作有兩種：

- 將節點 x 塗成紅色
- 詢問節點 x 到最近的紅點的距離

- $N \leq 10^5$
- $Q \leq 10^5$

樹論 – 重心剖分

兩個節點之間的最短路徑一定會經過他們在重心樹上的 LCA，所以在重心樹上的每個節點儲存離他最近的紅點距離，詢問時沿著重心樹往上爬，就能算答案了

樹論 – 重心剖分

兩個節點之間的最短路徑一定會經過他們在重心樹上的 LCA，所以在重心樹上的每個節點儲存離他最近的紅點距離，詢問時沿著重心樹往上爬，就能算答案了

每個點到祖先的距離可以在找到每一層的重心時順便 DFS 一次該連通塊記錄下來，所以複雜度是 $O(N \log N)$

樹論 – 虛樹

虛樹（Virtual Tree）又稱為輔助樹（Auxiliary Tree），用來處理每次操作只會和樹上一部份的點有關的題目

題目 (Kingdom and its Cities)

給一棵 N 個節點的樹和 Q 次詢問，第 i 次詢問包含 k_i 個特殊點，問最少要拔掉幾個非特殊點才能讓特殊點兩兩互不連通。

- $N \leq 10^5$
- $\sum k_i \leq 10^5$

樹論 – 虛樹

先處理單筆詢問。如果有特殊點相鄰肯定無解，否則 DFS 整棵樹，每次讓當前節點的子樹滿足條件

- 當 DFS 完特殊點 v，且某個子樹中還有和 v 連通的特殊點，那就在路徑上任意拔掉一點
- 當 DFS 完非特殊點 v，v 需要拔掉若且唯若兩個以上的子樹還有連通的特殊點

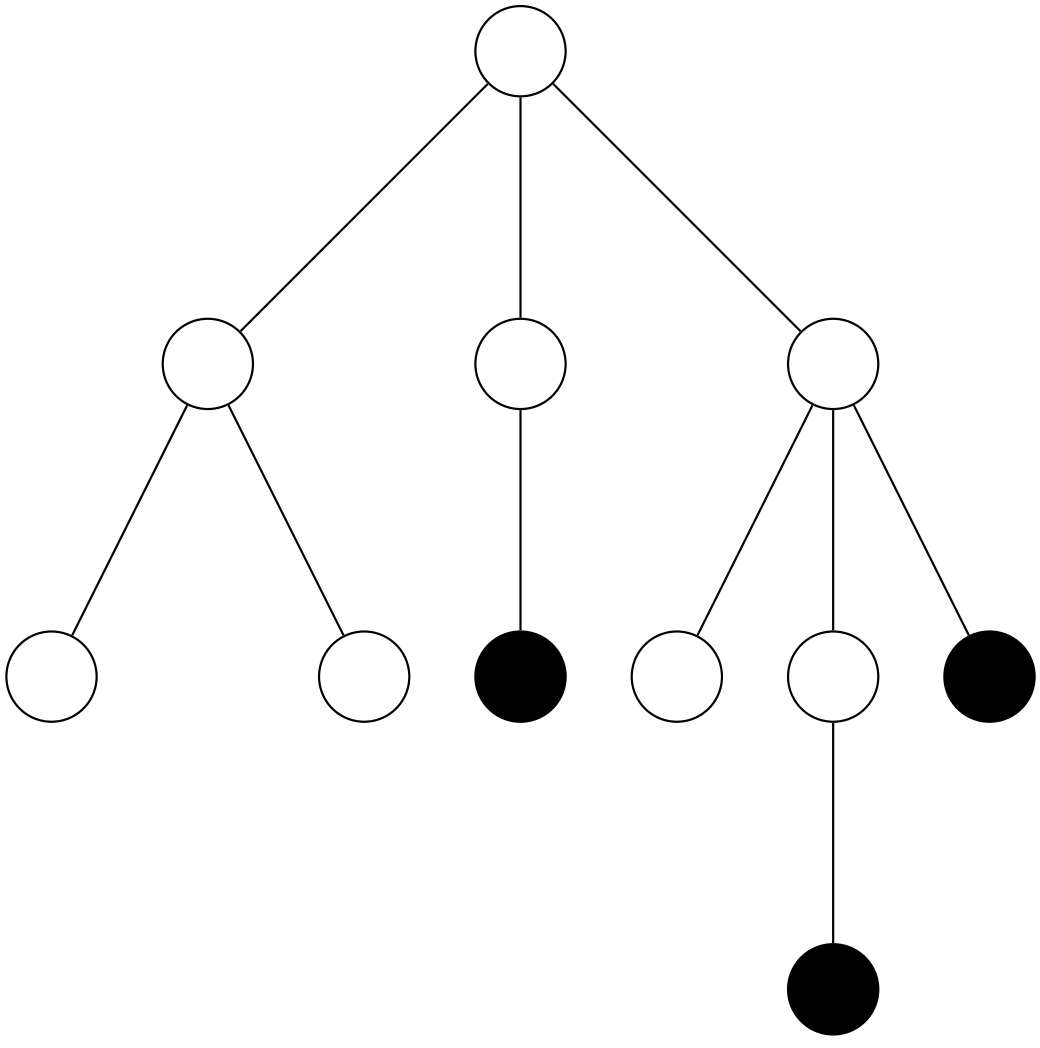
樹論 – 虛樹

先處理單筆詢問。如果有特殊點相鄰肯定無解，否則 DFS 整棵樹，每次讓當前節點的子樹滿足條件

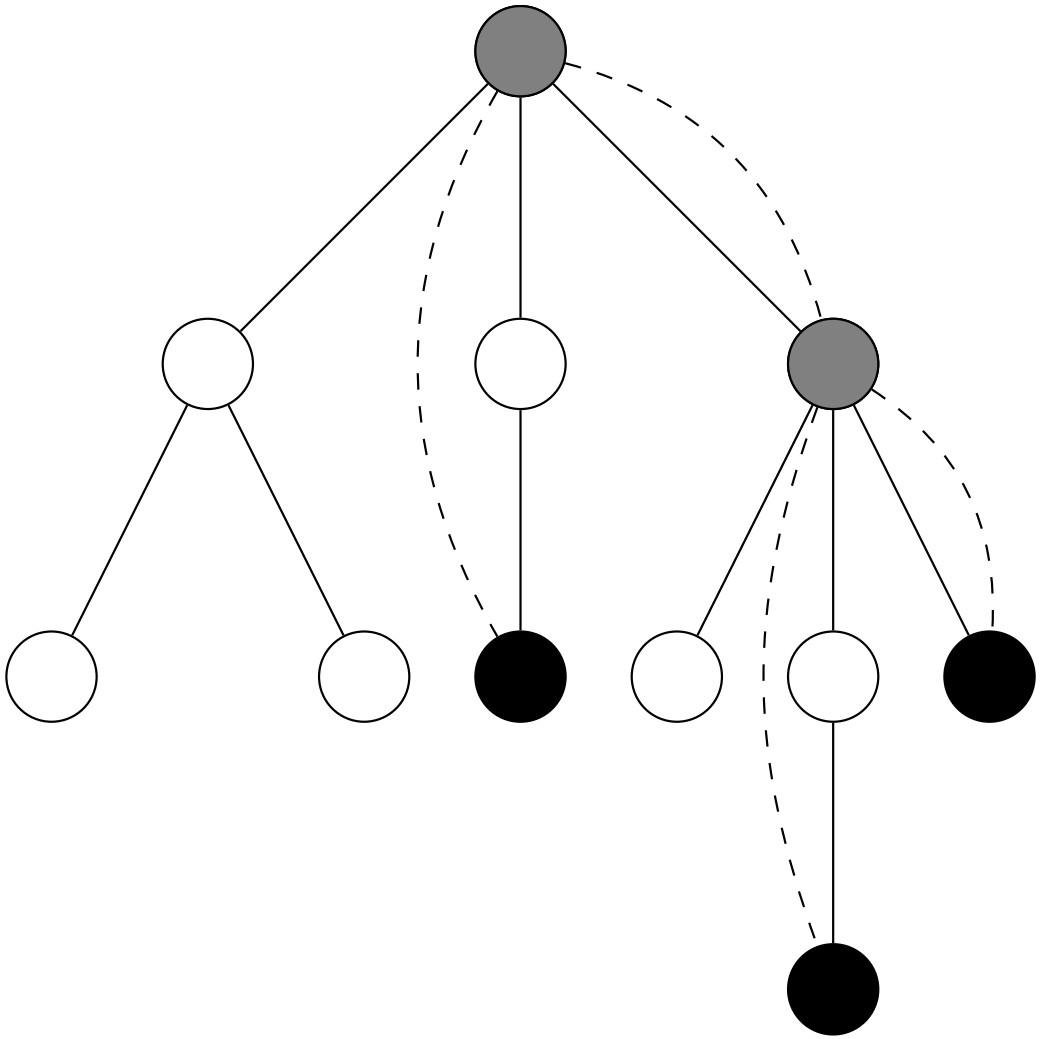
- 當 DFS 完特殊點 v，且某個子樹中還有和 v 連通的特殊點，那就在路徑上任意拔掉一點
- 當 DFS 完非特殊點 v，v 需要拔掉若且唯若兩個以上的子樹還有連通的特殊點

其實不一定要 DFS 到整棵樹，只要保留特殊點和它們之間的 LCA，知道它們的祖孫關係，就能得到正確的答案了，而這張輔助圖就是虛樹

樹論 – 虛樹



樹論 – 虛樹



樹論 – 虛樹

在對某 k 個點建立虛樹時，如果我們先把 k 個點按照 DFS 序排序好，這時候任兩點的 LCA 一定會是 **DFS 序相鄰的兩個點的 LCA**，而相鄰的點對只有 k 對，所以虛樹中至多有 $2k$ 個點

樹論 – 虛樹

從給定的點集建構虛樹：

先照剛才說的找出虛樹上的點，將這些點照 DFS 序排好，接下來分別尋找他們在虛樹上的父節點

樹論 – 虛樹

記錄進入和離開時間以判斷兩個點的祖孫關係，在 v 節點的祖先之中，DFS 序最大的就是父節點

樹論 – 虛樹

記錄進入和離開時間以判斷兩個點的祖孫關係，在 v 節點的祖先之中，DFS 序最大的就是父節點

維護一個單調 stack，每當 stack 的頂端不是現在拜訪的節點的祖先時就 pop 掉，因為他不會是後面任何一點的祖先，留在 stack 頂端的就是當前節點的父親，找到後把當前節點推到 stack 裡

樹論 – 虛樹

記錄進入和離開時間以判斷兩個點的祖孫關係，在 v 節點的祖先之中，DFS 序最大的就是父節點

維護一個單調 stack，每當 stack 的頂端不是現在拜訪的節點的祖先時就 pop 掉，因為他不會是後面任何一點的祖先，留在 stack 頂端的就是當前節點的父親，找到後把當前節點推到 stack 裡

找 LCA 會花 $O(k \log N)$ 的時間，排序花的時間是 $O(k \log k)$，最後 DFS 的時間是 $O(k)$，總複雜度是 $O(k \log N)$

最小生成樹

最小生成樹

- Kruskal：從整個邊集 greedy 挑
- Prim：從一個連通塊相鄰的邊 greedy 挑
- Borůvka：從每個連通塊相鄰的邊 greedy 挑

最小生成樹 – Borůvka

對於每個連通塊用最少花費擴增連通塊大小

1. 一開始沒有任何邊，每個點是一個連通塊
2. 找出每個連通塊連接出去的最小邊
3. 將這些邊加到最小生成樹（跳過形成環的邊）
4. 重複加邊直到剩下一個連通塊

最小生成樹 – Borůvka

- 一個連通塊旁邊的最小邊一定在最小生成樹上 (假設邊權相異)
- 每進行一輪，連通塊數量至少減半，最多 $O(\log N)$ 輪

最小生成樹 – Borůvka

題目 (MST and Rectangles)

給一個 $N \times N$ 的矩陣 A，一開始所有元素都是 0，接著進行 M 次矩形加值操作：

- 給定 X_1, X_2, Y_1, Y_2, W，對於所有 $X_1 \leq i \leq X_2, Y_1 \leq j \leq Y_2$，將 $A_{i,j}$ 和 $A_{j,i}$ 加上 W。

全部操作完後建立一張 N 個點的無向完全圖 G，(i, j) 的邊權為 $A_{i,j}$，求這張圖的最小生成樹。

- $N, M \leq 10^5$

最小生成樹 – Borůvka

- 找到一個連通塊的最小鄰邊：枚舉點 i，詢問第 i 列上和他屬於不同連通塊的最小值

最小生成樹 – Borůvka

- 找到一個連通塊的最小鄰邊：枚舉點 i，詢問第 i 列上和他屬於不同連通塊的最小值
- 維護一列上最小的值以及和最小值不同連通塊的次小值，再根據 i 所屬的連通塊選擇

最小生成樹 – Borůvka

- 找到一個連通塊的最小鄰邊：枚舉點 i，詢問第 i 列上和他屬於不同連通塊的最小值
- 維護一列上最小的值以及和最小值不同連通塊的次小值，再根據 i 所屬的連通塊選擇
- 將加值用差分拆成兩筆操作，使用掃描線線段樹可以在 $O(\log N)$ 的時間處理每個加值或詢問的操作

最小生成樹 – Borůvka

- 找到一個連通塊的最小鄰邊：枚舉點 i，詢問第 i 列上和他屬於不同連通塊的最小值
- 維護一列上最小的值以及和最小值不同連通塊的次小值，再根據 i 所屬的連通塊選擇
- 將加值用差分拆成兩筆操作，使用掃描線線段樹可以在 $O(\log N)$ 的時間處理每個加值或詢問的操作
- 每一輪的操作進行完後，因為每個點所在的連通分量可能會改變，要把線段樹整棵重新建構

最小生成樹 – Borůvka

- 找到一個連通塊的最小鄰邊：枚舉點 i，詢問第 i 列上和他屬於不同連通塊的最小值
- 維護一列上最小的值以及和最小值不同連通塊的次小值，再根據 i 所屬的連通塊選擇
- 將加值用差分拆成兩筆操作，使用掃描線線段樹可以在 $O(\log N)$ 的時間處理每個加值或詢問的操作
- 每一輪的操作進行完後，因為每個點所在的連通分量可能會改變，要把線段樹整棵重新建構

複雜度是 $O((N+Q)\log^2 N)$。

最短路

最短路 – 差分約束

- N 個變數 $x_1, x_2, \ldots, x_N$ 以及 M 個約束條件
- 每個約束條件是對於其中兩個變數的差的限制，通常長得會像是 $x_i - x_j \leq c_k$
- 目標是找出一組 $x_1, x_2, \ldots, x_N$ 的解來滿足所有的約束條件

最短路 – 差分約束

- 差分約束系統：$x_i \leq x_j + c_k$
- 最短路：$\texttt{dis}[y] \leq \texttt{dis}[x] + z$

最短路 – 差分約束

把每個變數 x_i 看成一張圖中的節點，對於每個條件 $x_i - x_j \leq c_k$，從節點 j 到節點 i 連一條長度為 c_k 的有向邊。

最短路 – 差分約束

把每個變數 x_i 看成一張圖中的節點，對於每個條件 $x_i - x_j \leq c_k$，從節點 j 到節點 i 連一條長度為 c_k 的有向邊。

設 `dis[0]` $= 0$ 並向每個點連一條權重為 0 的邊，之後跑一遍單源點最短路，若存在負環則代表無解，否則 $x_i =$ `dis[`i`]` 即為這個差分約束系統的一組解。

最短路 – 差分約束

把每個變數 x_i 看成一張圖中的節點，對於每個條件 $x_i - x_j \leq c_k$，從節點 j 到節點 i 連一條長度為 c_k 的有向邊。

設 `dis[0]` $= 0$ 並向每個點連一條權重為 0 的邊，之後跑一遍單源點最短路，若存在負環則代表無解，否則 $x_i =$ `dis[`i`]` 即為這個差分約束系統的一組解。

實作上通常會利用 Bellman-Ford 或 SPFA 來判斷是否存在負環，最差時間複雜度為 $O(NM)$。

歐拉迴路

歐拉迴路

題目 (一筆畫問題)

對於一張給定的無向圖，請找到一條路徑使每條邊都剛好被經過一次。

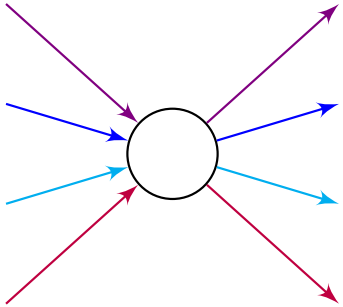
這條路徑稱作歐拉路徑（Euler path）

起點跟終點在同一個頂點的話又叫歐拉迴路（Euler circuit）

歐拉迴路

存在歐拉路徑的圖會滿足：

- 連通圖
- 非起終點的度數一定是偶數



歐拉迴路

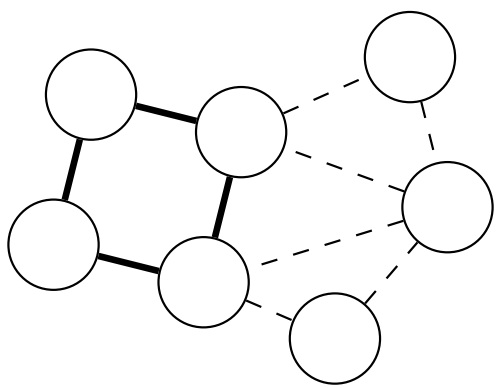
性質

連通無向圖 G 存在歐拉路徑的充要條件是：G 中奇頂點 (點度為奇數的點) 恰有 0 個或 2 個。
連通無向圖 G 存在歐拉迴路的充要條件是：G 中所有頂點的度都是偶數。

歐拉迴路

假設 G 中沒有奇頂點

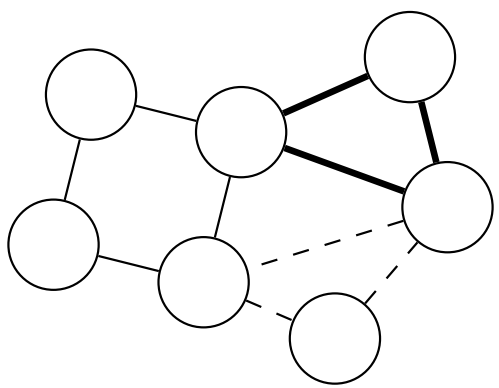
- 從任意的點 v 出發，走到不能走為止，根據奇偶性一定會停在 v，也就是形成一個迴路



歐拉迴路

假設 G 中沒有奇頂點

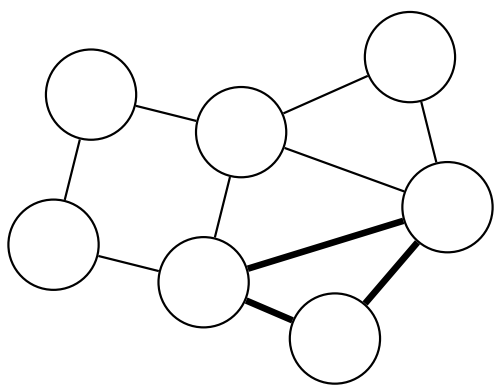
- 從任意的點 v 出發，走到不能走為止，根據奇偶性一定會停在 v，也就是形成一個迴路
- 假如還沒結束，在剛才的迴路中找到某個頂點 u 旁邊還有沒用過的邊，從點 u 出發再找到一個迴路
- 把這兩個迴路在 u 這邊接起來，得到一個更長的迴路



歐拉迴路

假設 G 中沒有奇頂點

- 從任意的點 v 出發，走到不能走為止，根據奇偶性一定會停在 v，也就是形成一個迴路
- 假如還沒結束，在剛才的迴路中找到某個頂點 u 旁邊還有沒用過的邊，從點 u 出發再找到一個迴路
- 把這兩個迴路在 u 這邊接起來，得到一個更長的迴路
- 重複這個動作，一定能把所有邊串成一條迴路。



歐拉迴路

有奇頂點？

- 從奇頂點出發
- 用連一條邊把他們接起來，找到歐拉迴路後，再拔掉多的那條邊

歐拉迴路

雙向鏈接串列？

改成確定一段路徑中間不需要再插入後才儲存這段路徑，用一次 DFS 找完歐拉迴路

歐拉迴路

- 先從某個點出發走到沒路，把最後一條邊加到答案

歐拉迴路

- 先從某個點出發走到沒路，把最後一條邊加到答案
- 回頭檢查上一個點 v，如果 v 旁邊的邊也都走過了，就把前一條邊也加進答案，再繼續往回檢查
- 如果 v 旁邊還有邊可以走，就先走過去繞一圈回到 v，把多走的邊也從後面一條一條檢查後加進答案，再把一開始走到 v 的邊也加進去

歐拉迴路

- 先從某個點出發走到沒路，把最後一條邊加到答案
- 回頭檢查上一個點 v，如果 v 旁邊的邊也都走過了，就把前一條邊也加進答案，再繼續往回檢查
- 如果 v 旁邊還有邊可以走，就先走過去繞一圈回到 v，把多走的邊也從後面一條一條檢查後加進答案，再把一開始走到 v 的邊也加進去
- 重複這個動作直到每條走過的邊都檢查完，根據前面的觀察一定會形成歐拉迴路。

歐拉迴路

```cpp
// the neighbor and the index of edge
vector<vector<pair<int, int>>> g;
vector<int> ans;
vector<bool> used;
void dfs(int v) {
  while (!g[v].empty()) {
    int u, eid;
    tie(u, eid) = g[v].back(); g[v].pop_back();
    if (used[eid]) continue;
    used[eid] = true;
    dfs(u);
    ans.push_back(eid);
  }
}
```

歐拉迴路

一個點會被檢查好幾次，可以使用鄰接串列（adjacency list）儲存這張圖，每次用完一條邊就把他移除掉。

記得在刪除 (v, u) 這條邊的同時也要刪除 (u, v)，一種方法是用 set 儲存，不過這邊的做法是幫每條邊儲存一個刪除標記。

複雜度是 $O(M)$。

歐拉迴路

題目 (Fair Share)

給 M 個長度為偶數的陣列，請把每個陣列中恰一半的元素丟到 multiset L 裡面，另外一半的元素丟到 multiset R 裡面，且最後 L 和 R 要長一樣 (每一種值的出現個數都一樣)。

- 總元素個數 $\leq 2 \cdot 10^5$

歐拉迴路

分兩邊讓兩邊長一樣

歐拉迴路：對每個點來說入邊和出邊一樣多

歐拉迴路

- 每個陣列被分到兩邊的元素要一樣多：幫每個陣列建一個點連到它的所有元素
- 每個值被分到兩邊的數量要一樣多：幫每個值建立一個點連到所有這個值的元素

歐拉迴路

- 每個陣列被分到兩邊的元素要一樣多：幫每個陣列建一個點連到它的所有元素
- 每個值被分到兩邊的數量要一樣多：幫每個值建立一個點連到所有這個值的元素

可以看成一張二分圖

- 左邊每個點代表一個陣列 a
- 右邊每個點代表一個值 x
- 為每個 $a_i = x$ 建立一條 (a, x) 的邊。

歐拉迴路

假如這張圖能找到一條歐拉迴路，那麼把每條邊定向，從左到右的放到第一個 multiset，從右到左的放到第二個 multiset，就能滿足那兩個條件

如果找不到歐拉迴路呢？如果有某個連通塊存在奇度數的點，意思就是有個數字出現奇數次，那顯然無解。否則對每個連通塊分別去找找看，然後直接合併答案就好了。

歐拉迴路 – 有向圖

有向圖存在歐拉迴路的充要條件：

- 連通圖
- 對於每個頂點，相鄰的出邊、入邊一樣多

歐拉迴路 – 有向圖

有向圖存在非迴路的歐拉路徑的充要條件：

- 連通圖
- 起點的出邊比入邊多一條
- 終點的入邊比出邊多一條
- 其餘的點兩種邊一樣多

歐拉迴路 – 有向圖

有向圖存在非迴路的歐拉路徑的充要條件：

- 連通圖
- 起點的出邊比入邊多一條
- 終點的入邊比出邊多一條
- 其餘的點兩種邊一樣多

找解的方式與無向圖相同

歐拉迴路 – K 筆畫問題

找到最小的 K，使得存在 K 條路徑滿足圖上每條邊都恰好被其中一條路徑經過一次

歐拉迴路 – K 筆畫問題

- 當一個連通圖中有 $2k$ 個奇頂點時，因為每用一條路徑只有起終點的度數奇偶性可能改變，所以至少要 k 條

歐拉迴路 – K 筆畫問題

- 當一個連通圖中有 $2k$ 個奇頂點時，因為每用一條路徑只有起終點的度數奇偶性可能改變，所以至少要 k 條
- 多加一個點，把奇頂點各連一條邊過去，此時這張圖存在歐拉迴路，拔掉那些邊後恰好剩下 k 條路徑，因此至多只需要 k 條

歐拉迴路 – K 筆畫問題

- 當一個連通圖中有 $2k$ 個奇頂點時，因為每用一條路徑只有起終點的度數奇偶性可能改變，所以至少要 k 條
- 多加一個點，把奇頂點各連一條邊過去，此時這張圖存在歐拉迴路，拔掉那些邊後恰好剩下 k 條路徑，因此至多只需要 k 條

每個連通塊需要 $\max(1, \text{奇頂點數量}/2)$ 條路徑，每個連通塊的答案加總就是這張圖需要的路徑數

加完邊後跑歐拉迴路的演算法找解

匹配

匹配

- 匹配（matching）：一個邊的子集，滿足每個頂點最多和子集中的一條邊相鄰
- 最大匹配：包含最多條邊的匹配
- 最大權匹配：每條邊有邊權，邊權總和最大的匹配
- 完美匹配（perfect matching）：每個頂點恰好和一條邊相鄰的匹配

匹配 – 二分圖匹配

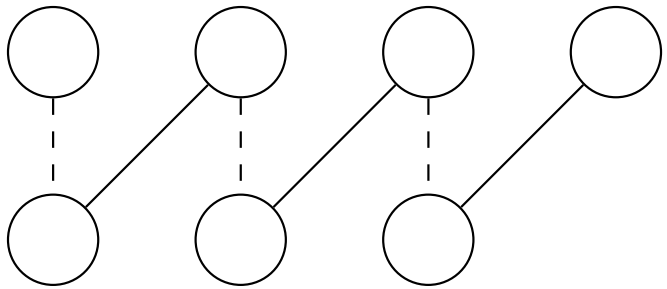
- 圖中頂點可分為兩個子集，使每條邊的兩個端點都屬於不同子集
- 可以用最大流的方式解決

匹配 – 二分圖匹配

定義

對於一組匹配 M，圖上的一個路徑
$P = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n, x_{n+1})$ 被稱作**交錯路徑**
(alternating path) 若以下條件滿足：

- 所有的 (x_i, y_i) 以及 (y_i, x_{i+1}) 皆是圖上的邊
- x_1 不位於 M 上
- 所有的 (y_i, x_{i+1}) 皆屬於 M
- 所有的 (x_i, y_i) 皆不屬於 M

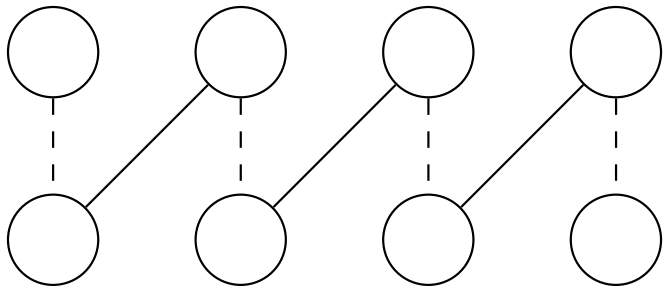


匹配 – 二分圖匹配

定義

對於一組匹配 M，圖上的一個路徑 $P = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$ 被稱作**增廣路徑（augmenting path）**若以下條件滿足：

- 所有的 (x_i, y_i) 以及 (y_i, x_{i+1}) 皆是圖上的邊
- x_1 以及 y_n 都不在 M 上
- 所有的 (y_i, x_{i+1}) 皆屬於 M
- 所有的 (x_i, y_i) 皆不屬於 M



匹配 – 二分圖匹配

把增廣路徑上所有未匹配邊都改成匹配邊、匹配邊改成未匹配邊，新的邊集還是一組匹配，且由於增廣路徑的長度是奇數，所以新產生的匹配比原本的大小多一

匹配 – 二分圖匹配

把增廣路徑上所有未匹配邊都改成匹配邊、匹配邊改成未匹配邊，新的邊集還是一組匹配，且由於增廣路徑的長度是奇數，所以新產生的匹配比原本的大小多一

定理 (Berge's Lemma)

一組匹配 M 為圖 G 中最大匹配若且唯若圖上不存在對於 M 來說的增廣路徑。

匹配 – 二分圖匹配

把增廣路徑上所有未匹配邊都改成匹配邊、匹配邊改成未匹配邊，新的邊集還是一組匹配，且由於增廣路徑的長度是奇數，所以新產生的匹配比原本的大小多一

定理 (Berge's Lemma)

一組匹配 M 為圖 G 中最大匹配若且唯若圖上不存在對於 M 來說的增廣路徑。

假設圖上存在更大的匹配 M'，對稱差 $M \oplus M'$ 上的某個連通塊就會是對於 M 的增廣路徑

匹配 – 二分圖匹配

不斷在圖上找增廣路徑，如果找到的話就翻轉路徑上的邊的角色，直到找不到為止

匹配 – 二分圖匹配

對於每個在左邊的頂點 $x \in X$ 尋找從 x 開始的增廣路徑

- 從 x 開始 DFS。考慮 x 的所有鄰居 $y \in Y$，如果 y 沒有被匹配的話，那我們就找到了 $P = (x, y)$ 這個路徑，將 y 與 x 匹配。
- 對於一個匹配過的鄰居 y，考慮他的匹配點 $z \in X$，如果我們可以找到從 z 開始的增廣路徑 P' 的話，那麼把 x 跟 y 接到 P' 前面就得到了一組從 x 開始的增廣路徑。因此這時我們遞迴計算 z。

匹配 – 二分圖匹配

假如從左邊的頂點 x 開始找不到增廣路徑，那之後不管怎麼翻它也不會產生增廣路徑，因此對於每個左邊的頂點，我們只會進行一次 DFS，總複雜度為 $O(V(V+E)) = O(VE)$

匹配 – 二分圖匹配

```
bool dfs(int v) {
  vis[v] = true;
  for (int u : g[v]) {
    if (match[u] == -1 || (!vis[match[u]] && dfs(match[u]))) {
      match[u] = v;
      return true;
    }
  }
  return false;
}
```

匹配 – Hopcroft-Karp

Hopcroft-Karp 演算法：

- 每次找多條增廣路徑一起擴充
- 每一輪先用 BFS 找出最短的增廣路徑
- 從最短的增廣路徑中找出極大（maximal）的子集來擴充

複雜度 $O(E\sqrt{V})$

匹配 – Hopcroft-Karp

引理

若匹配 M 中最短的增廣路徑長度為 l，$\{P_1, P_2, \ldots, P_k\}$ 是一些沒有共用頂點的最短增廣路徑的極大集合，令 $M' = M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k)$，則 M' 中的最短增廣路徑長度大於 l。

引理

若匹配 M 中最短的增廣路徑長度為 l，則最大匹配 M^* 的大小 $|M^*| \leq |M| + \frac{|V|}{l+1}$。

匹配 – Hopcroft-Karp

在進行 $\sqrt{V}$ 次迭代後，圖上的最短增廣路徑的長度至少是 $\Omega(\sqrt{V})$，因此最多再擴增 $O(\sqrt{V})$ 次就會變成最大匹配，總共進行 $O(\sqrt{V})$ 次迭代

每次迭代會在原圖上先透過 BFS 找出從未匹配點沿著交錯的邊走到每個點的最短距離，再用 DFS 找增廣路徑，且只考慮 BFS 時有用的邊 (兩端的距離剛好差一)，確保找出來的每條增廣路徑都是最短的

匹配 – Hopcroft-Karp

在進行 $\sqrt{V}$ 次迭代後，圖上的最短增廣路徑的長度至少是 $\Omega(\sqrt{V})$，因此最多再擴增 $O(\sqrt{V})$ 次就會變成最大匹配，總共進行 $O(\sqrt{V})$ 次迭代

每次迭代會在原圖上先透過 BFS 找出從未匹配點沿著交錯的邊走到每個點的最短距離，再用 DFS 找增廣路徑，且只考慮 BFS 時有用的邊 (兩端的距離剛好差一)，確保找出來的每條增廣路徑都是最短的

每個迭代複雜度為 $O(E)$，最終的時間複雜度 $O(E\sqrt{V})$

匹配 – Hopcroft-Karp

```
bool bfs(int n1) {
  // ... 把左邊未匹配點設為 BFS 的起點推入 queue ...
  while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : g[v]) {
      if (match[u] == -1) {
        l = dis[v];
        return true;
      }
      if (dis[match[u]] == INF) {
        dis[match[u]] = dis[v] + 1; q.push(match[u]);
      }
    }
  }
  return false;
}
```

匹配 – Hopcroft-Karp

```
int HopcroftKarp(int n1, int n2) {
  match.assign(n2, -1);
  dis.resize(n1); used.assign(n1, false); vis.resize(n1);
  int ans = 0;
  while (bfs(n1)) {
    fill(vis.begin(), vis.end(), false);
    for (int i = 0; i < n1; ++i) {
      if (!used[i] && dfs(i)) {
        used[i] = true;
        ++ans;
      }
    }
  }
  return ans;
}
```

匹配 – 二分圖最大權匹配

Kuhn-Munkres（KM）算法：在帶權的完全二分圖上找最大權的完美匹配

透過加入空點或是將不存在的邊權設為 0 或負無限大來用在一般情況

匹配 – Kuhn-Munkres

KM 算法的關鍵就在於它給每個二分圖上的頂點 v 設了一個頂標 l_v，並透過調整兩邊頂點的頂標來找到最大完美匹配

對於左邊的點 x，右邊的點 y，頂標隨時必須滿足 $l_x + l_y \geq W_{xy}$，其中 W_{xy} 為 x 與 y 之間的邊權

匹配 – Kuhn-Munkres

定理

令 M 為任意一組完美匹配的權值總和。對於任意一組滿足 $l_x + l_y \geq W_{xy}$ 的頂標，都有

$$m = \sum_{x \in X} l_x + \sum_{y \in Y} l_y \geq M$$

更進一步地，令 P 為一組最大權完美匹配且 M_P 為 P 的權值總和，則存在頂標 l 滿足 $M_P = \sum_{x \in X} l_x + \sum_{y \in Y} l_y$ 且 $l_x + l_{P_x} = W_{xP_x}$。

匹配 – Kuhn-Munkres

對於一組頂標 l，我們稱 $l_x + l_y = W_{xy}$ 的那些邊為可行邊，並假設 G' 為 G 移除掉非可行邊所形成的子圖

如果只用可行邊能找到一組完美匹配，它就是最大權的完美匹配

匹配 – Kuhn-Munkres

先初始化頂標，對於所有右邊的頂點 y，設 $l_y = 0$，左邊的頂點 x 的頂標就可以設為 $l_x = \max_{y \in Y} W_{xy}$

接著我們嘗試為左邊的頂點 v 一一找到增廣路徑，並且限制只能使用可行邊。當找不到的時候，就代表需要修改一下頂標，才能增加匹配數

為了不破壞前面找好的匹配，所有匹配邊依然要是可行邊，而增廣路徑上的非匹配邊當然也要是可行邊，之後才能把他翻過來

匹配 – Kuhn-Munkres

- 若找到從 v 出發的增廣路徑，完成這次增廣。
- 否則，找出從 v 出發的交錯路徑樹，搜集經過的頂點，左邊的放到集合 S 中，右邊的放到集合 T 中。如果此時我們將所有 S 中的頂點頂標減 d，T 中的頂點頂標加 d，那麼：
  1. 若 $x \in S$ 且 $y \in T$：一加一減抵銷，頂標和不變，因此可行性也不變。
  2. 若 $x \notin S$ 且 $y \notin T$：頂標和不變，可行性也不變。
  3. 若 $x \in S$ 且 $y \notin T$：頂標和變小，可能導致 (x, y) 原本不在 G' 中而更新後進入 G'。
  4. 若 $x \notin S$ 且 $y \in T$：頂標和變大，可能是一直都不在 G' 中，或是原本在而更新後被移除。

所有匹配邊及交錯路徑上的邊依然可行。只需注意第三種邊的頂標和不能變太小

匹配 – Kuhn-Munkres

從所有第三種邊 ($x \in S$ 且 $y \notin T$) 中挑出 $l_x + l_y - w_{xy}$ 的最小值當做 d，這樣就不會有頂標和過小的邊，而且至少會有一條邊加入 G'，同時 T 中也會至少多一個節點

在進行 $O(N)$ 次後，T 中就會有非匹配點，此時就找到增廣路徑了

匹配 – Kuhn-Munkres

找一條增廣路徑時，會修改 $O(N)$ 次頂標，每次要 $O(N^2)$ 找到 d，因此複雜度是 $O(N^3)$，而總共要找 $O(N)$ 條，所以最終複雜度是 $O(N^4)$

如果要降低複雜度，可以好好對每個 Y 中的頂點維護 d 是多少才能讓他進入 T，並且不要在一輪增廣中重複 DFS 一個點，就能把複雜度壓到 $O(N^3)$

匹配 – 定理與應用

題目 (砲打皮皮)

有一個 $N \times N$ 的棋盤，其中有 K 格上有障礙物。一次操作可以選擇一行或一列並消滅該行或該列上所有的障礙物。請問最小操作次數為何？

- $1 \leq N \leq 1000$
- $1 \leq K \leq 20000$

匹配 – 定理與應用

建造一個左右各 N 個點的二分圖，左邊的點代表棋盤的列、右邊的點代表棋盤的行，並將位於 (r, c) 的障礙物轉換成 r 與 c 之間的一條邊

一次操作就等於選一個左邊的點（選一列）或選一個右邊的點（選一行），並且刪除該點連接的所有邊

問題被轉換成：對於一個二分圖，求出最少需要覆蓋幾個頂點才能讓每一條邊都至少有一個端點被覆蓋

匹配 – 定理與應用

建造一個左右各 N 個點的二分圖，左邊的點代表棋盤的列、右邊的點代表棋盤的行，並將位於 (r, c) 的障礙物轉換成 r 與 c 之間的一條邊

一次操作就等於選一個左邊的點（選一列）或選一個右邊的點（選一行），並且刪除該點連接的所有邊

問題被轉換成：對於一個二分圖，求出最少需要覆蓋幾個頂點才能讓每一條邊都至少有一個端點被覆蓋

這個問題被稱作二分圖上的**最小點覆蓋**問題

匹配 – 二分圖最小點覆蓋

定理 (König 定理)

對於連通二分圖 G，G 的最大匹配的邊數恆等於 G 的最小點覆蓋的點數。

匹配 – 二分圖最小點覆蓋

任意一組點覆蓋的大小一定不小於最大匹配的大小，因為一個點至多覆蓋一條匹配邊。接著來構造一個與匹配同樣大小的點覆蓋

考慮在最大匹配下，左邊的未匹配點 L 以及從 L 延伸出去的交錯路徑上的點集 U。令 K_L 為交錯路徑上在左邊的頂點、K_R 為在交錯路徑上右邊的頂點

令 $K = (X \setminus K_L) \cup K_R$，也就是所有在左邊不在交錯路徑上的頂點以及所有在右邊且在交錯路徑上的頂點

匹配 – 二分圖最小點覆蓋

對於所有邊 e 有以下兩種情形：

- e 的左端點在 K_L 中：根據尋找交錯路徑的方式，e 的右端點一定在 K_R 中，此時 e 的右端點被覆蓋。
- e 的左端點不在 K_L 中：此時 e 的左端點被覆蓋。

因此 K 是一個點覆蓋

匹配 – 二分圖最小點覆蓋

由於 $L \subseteq K_L$ 且 L 是所有左邊的未匹配點，$X \setminus K_L$ 中所有的點都是匹配點。而根據交錯路徑的定義，K_R 中的點一定也都是匹配點

不過因為沒有匹配的邊同時左端點在 $X \setminus K_L$ 中、右端點在 K_R 中，因此 K 的大小不大於最大匹配的大小，得知最小點覆蓋和最大匹配一樣大

匹配 – 二分圖最小點覆蓋

因此將一組最大匹配轉成一組最小點覆蓋的算法如下：

1. 為了找出交錯路徑，將所有非匹配邊改成由左指到右的有向邊、所有匹配邊改為由右指向左的有向邊。
2. 從所有左邊的未匹配點在新的圖上進行 DFS。
3. 最小點覆蓋即為「在左邊且沒被拜訪的點」以及「在右邊且被拜訪的點」。

匹配 – 定理與應用

題目 (密室逃脫)

給 $N \times N$ 的矩陣 A，每次操作可以選一行或一列把數字全部減一，求把所有數字變成 ≤ 0 的最少操作次數，並輸出一組解。

- $N \leq 500$
- $A_{i,j} \leq 2 \cdot 10^6$

匹配 – 定理與應用

題目 (密室逃脫)

給 $N \times N$ 的矩陣 A，每次操作可以選一行或一列把數字全部減一，求把所有數字變成 ≤ 0 的最少操作次數，並輸出一組解。

- $N \leq 500$
- $A_{i,j} \leq 2 \cdot 10^6$

一樣把行跟列建成兩邊的頂點，每個格子建成一條邊，假設第 i 行選的次數是 $cnt1_i$，第 j 列選的次數是 $cnt2_j$，那麼必須對所有 i, j 滿足 $cnt1_i + cnt2_j \geq a_{i,j}$

匹配 – 定理與應用

題目 (密室逃脫)

給 $N \times N$ 的矩陣 A，每次操作可以選一行或一列把數字全部減一，求把所有數字變成 ≤ 0 的最少操作次數，並輸出一組解。

- $N \leq 500$
- $A_{i,j} \leq 2 \cdot 10^6$

一樣把行跟列建成兩邊的頂點，每個格子建成一條邊，假設第 i 行選的次數是 $cnt1_i$，第 j 列選的次數是 $cnt2_j$，那麼必須對所有 i, j 滿足 $cnt1_i + cnt2_j \geq a_{i,j}$

KM 的頂標！求最大權完美匹配，得到的頂標就是一組解

匹配 – 二分圖最小權點覆蓋

題目 (二分圖最小權點覆蓋)

給一張二分圖，每個點有點權，請覆蓋一些點使得每一條邊都至少有一端點被覆蓋，且選擇的點集點權總和最小

匹配 – 二分圖最小權點覆蓋

題目 (二分圖最小權點覆蓋)

給一張二分圖，每個點有點權，請覆蓋一些點使得每一條邊都至少有一端點被覆蓋，且選擇的點集點權總和最小

二分圖最小點覆蓋問題可以用二分圖最大匹配解，而二分圖最大匹配可以用網路流解，事實上二分圖最小權點覆蓋也能用網路流的角度來解

匹配 – 二分圖最小權點覆蓋

- 建立超級源點 S 與超級匯點 T
- 對於所有左邊的點 x，連一條 S 到 x 容量為 W_x 的邊
- 對於所有右邊的點 y，連一條 y 到 T 容量為 W_y 的邊
- 對於所有二分圖上的邊 (x, y)，連一條 x 到 y 容量為 ∞ 的邊

匹配 – 二分圖最小權點覆蓋

在一張有源點、匯點的有向圖上，如果將頂點分為兩堆，一堆包含源點，一堆包含匯點，此時挑出所有從第一堆指向第二堆的邊，就稱為一組 $S-T$ **割**。而邊權和最小的割就是最小割

如果在剛才建的圖上找到一組最小割，並且覆蓋那些被割掉的邊對應的點，就會是一組最小權點覆蓋，因為此時割中沒有 ∞ 的邊，也就是原圖的每條邊都有一端被覆蓋

而最小割等價最大流，因此就解決二分圖最小權點覆蓋了

匹配 – 最小邊覆蓋

各個覆蓋相關問題的難度

- 二分圖最小權點覆蓋 $\rightarrow$ P
- 一般圖最小點覆蓋 $\rightarrow$ NP-hard
- 一般圖最小權邊覆蓋 $\rightarrow$ P

匹配 – 最小邊覆蓋

先找出一組最大匹配，再把未匹配點旁邊各選一條

如果存在更小的覆蓋，代表能有更多匹配點，與最大匹配矛盾

匹配 – 最小權邊覆蓋

同樣要找到某個匹配，再選擇每個未匹配點相鄰的最小邊

匹配 – 最小權邊覆蓋

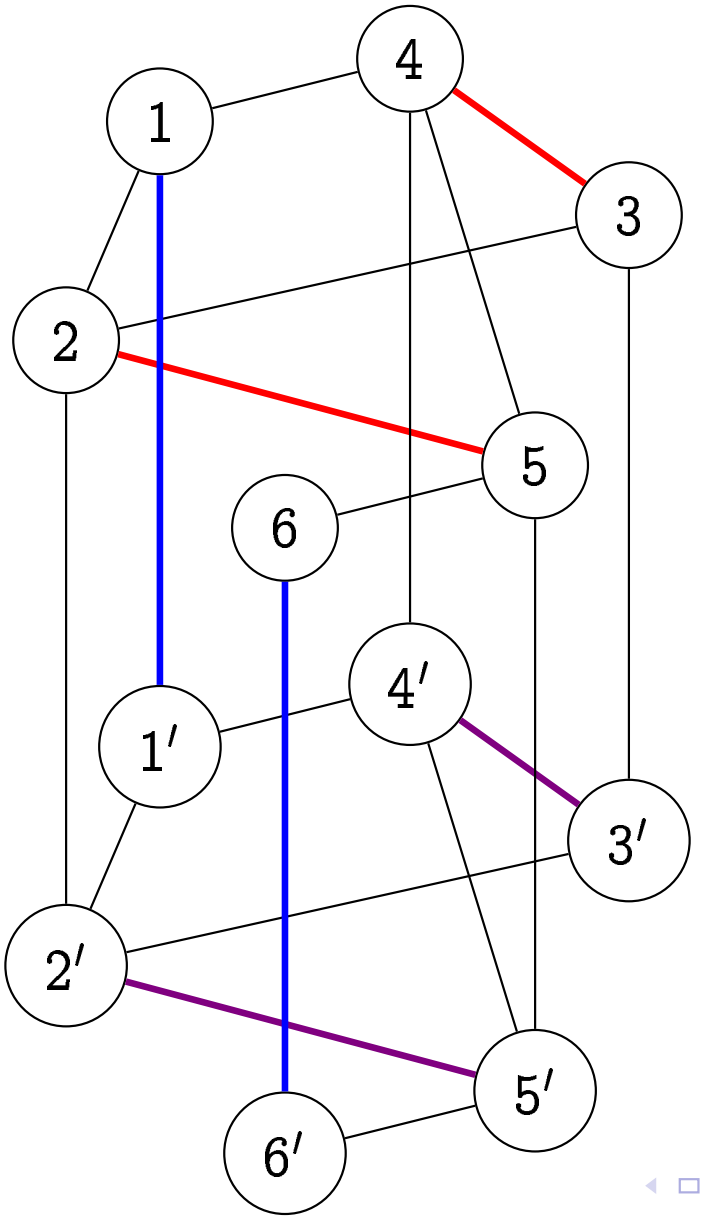
同樣要找到某個匹配，再選擇每個未匹配點相鄰的最小邊

在圖上加一些點和邊：

- 把 G 的點集複製一次，也就是對於 G 中的節點 v，G' 上都有 v 與 v'
- 對於原圖的每條邊 (u, v, w)，在 G' 中新增 $(u', v', 0)$ 這條邊
- 把所有的 v 與 v' 各用一條邊連起來，並且設邊權為 v 相鄰的最小邊權

此時 G' 上的最大匹配（完美匹配）就會與 G 上的邊覆蓋相互對應，而最小權的最大匹配也能在多項式時間找到

匹配 – 最小權邊覆蓋



匹配 – Hall 定理

定理 (Hall 定理 (Hall's marriage theorem))

令 $G = (X, Y)$ 為一二分圖。對於 X 的子集 S，定義 $N_G(S)$ 為 S 的鄰居，也就是所有在 Y 中與 S 中的點有連邊的頂點們。則 G 存在一個對 X 而言的完美匹配若且唯若

$$|S| \leq |N_G(S)| \tag{1}$$

對於所有 $S \subseteq X$ 都成立。

匹配 – Hall 定理

題目 (Allowed Letters)

你有一個由 a 到 f 組成的字串 s 以及 M 個限制。每個限制可以表示成 (p_i, t_i)，其中 p_i 為 1 到 $|s|$ 的位置，而 t_i 是一個 a 到 f 所組成的字串，代表位置 p_i 只能放 t_i 內的字元。請將 s 排列成字典序最小又不違反限制的字串。

- $|s| \leq 10^5$
- $m \leq |s|$

匹配 – Hall 定理

字典序最小：從頭開始，由 a 到 f 枚舉要放什麼字元，使得在這個位置放它之後後面存在至少一種合法的排法

利用 Hall 定理判斷能不能放

匹配 – Hall 定理

考慮一個二分圖，左邊為所有的位置，右邊為所有的字元 (如果有多個的話就有多個頂點)。位置 i 可以連到所有可以放在位置 i 的字元，那麼存在一組合法排列的充要條件就是這個二分圖上存在完美匹配

因為 a 到 f 只有六種字元，因此總共也只有 2^6 種限制，所以剛才的二分圖其實可以把左邊的節點濃縮成 2^6 個、右邊的節點濃縮成 6 個。

匹配 – Hall 定理

根據 Hall 定理，完美匹配存在的充要條件就是「對於每個字元的子集 X，有與 X 內的點連邊的限制數量都必須不少於 X 內的字元數量」

維護 $f(X)$ 為 X 內的字元個數、$c(X)$ 是限制為 X 的子集的限制數量，每次檢查對於所有 X，$f(X) \leq N' - c(\overline{X})$ 有沒有成立即可。

匹配 – 擬陣

定義

令 E 為一個有限集合，我們稱一個二元組 $M = (E, \mathcal{I})$ 為一個擬陣 (Matroid)，其中 $\mathcal{I} \subseteq 2^E$ 為 E 的子集所形成的**非空**集合，若：

- 遺傳特性 (hereditary property)：若 $S \in \mathcal{I}$ 以及 $S' \subsetneq S$，則 $S' \in \mathcal{I}$
- 擴充特性 (augmentation property)：對於 $S_1, S_2 \in \mathcal{I}$ 滿足 $|S_1| < |S_2|$，存在 $e \in S_2 \setminus S_1$ 使得 $S_1 \cup \{e\} \in \mathcal{I}$

注意到 $\mathcal{I}$ 非空代表 $\emptyset \in \mathcal{I}$

匹配 – 擬陣

定義

除此之外，我們有以下的定義：

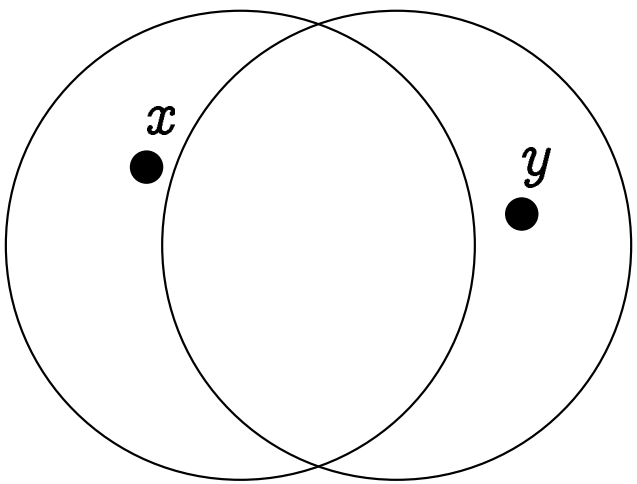
- 位於 $\mathcal{I}$ 中的集合我們稱之為獨立集 (independent set)，反之不在 $\mathcal{I}$ 中的我們稱為相依集 (dependent set)
- 極大的獨立集為基底 (base)、極小的相依集為迴路 (circuit)
- 一個集合 Y 的秩 (rank) $r(Y)$ 為該集合中最大的獨立子集的大小，也就是 $r(Y) = \max\{|X| \mid X \subseteq Y \text{ 且} X \in \mathcal{I}\}$

我們以 $\mathcal{B}_M$ 來代表 M 的基底集合，以 $\mathcal{C}_M$ 來代表 M 的迴路集合

匹配 – 擬陣

引理 (基底的強交換性 (Strong Basis Exchange))

令 B_1 與 B_2 是 M 中的基底且 $B_1 \neq B_2$，則對所有 $x \in B_1 \setminus B_2$ 都存在 $y \in B_2 \setminus B_1$ 使得 $(B_1 \setminus \{x\}) \cup \{y\}$ 與 $(B_2 \setminus \{y\}) \cup \{x\}$ 都是基底。



匹配 – 擬陣

對於一個圖 $G = (V, E)$，我們定義他的**圖擬陣（Graphic Matroid）**為 $M_G = (E, \mathcal{I}_G)$，其中 E 的子集 E' 屬於 $\mathcal{I}_G$ 若 E' 中的邊不會形成任何的環

1. 若 E' 不包含環，則任何 E' 的子集當然不包含環
2. 若 E_1 與 E_2 皆不包含環且 $|E_1| < |E_2|$，則一定找得到一條在 E_2 中的邊 e 使得 e 連接 E_1 中兩個不連通的點。這就代表 $E_1 \cup \{e\}$ 也沒有環。

匹配 – 擬陣

對於一個圖 $G = (V, E)$，我們定義他的**圖擬陣（Graphic Matroid）**為 $M_G = (E, \mathcal{I}_G)$，其中 E 的子集 E' 屬於 $\mathcal{I}_G$ 若 E' 中的邊不會形成任何的環

1. 若 E' 不包含環，則任何 E' 的子集當然不包含環
2. 若 E_1 與 E_2 皆不包含環且 $|E_1| < |E_2|$，則一定找得到一條在 E_2 中的邊 e 使得 e 連接 E_1 中兩個不連通的點。這就代表 $E_1 \cup \{e\}$ 也沒有環。

對於連通圖而言，圖擬陣的基底就是生成樹

匹配 – 擬陣

- **塗色擬陣 (Colored Matroid)**: 對於一個大小為 n 的集合 E 以及一個塗色函式 $c : E \to \{1, 2, \ldots, k\}$，其中 $c(e)$ 代表元素 e 被塗了什麼顏色。考慮塗色擬陣 $M = (E, \mathcal{I})$，其中 $\mathcal{I} = \{E' \subseteq E \mid E'$ 中沒有兩個元素被塗了相同的顏色$\}$。一樣可以簡單驗證 M 滿足擬陣的條件。這可以推廣到在給定限制 $p_1, p_2, \ldots, p_k$ 下，定義
  $\mathcal{I} = \{E' \subseteq E \mid E'$ 中 $c(e) = t$ 的元素不超過 p_t 個$\}$。
- **截斷擬陣 (Truncated Matroid)**: 對於一個擬陣 $M = (E, \mathcal{I})$ 以及非負整數 k，定義截斷擬陣 $M_k = (E, \mathcal{I}_k)$，其中 $\mathcal{I}_k = \{E' \subseteq I \mid |E'| \leq k\}$，也就是只保留 M 中大小不超過 k 的獨立集。

匹配 – 最小權基底

最小生成樹 $\rightarrow$ 擬陣的最小權基底

- 對於擬陣 $M = (E, \mathcal{I})$ 以及非負權重函數 w，求出 $B \in \mathcal{B}$ 使得 $w(B)$ 最小。其中 $w(B)$ 為 B 中所有元素的權重相加

匹配 – 最小權基底

定理

若 M 是一個擬陣，則用貪心演算法可求出最小權基底。

匹配 – 最小權基底

- 將 E 中的元素照權重由小到大排成 $e_1, e_2, \ldots, e_n$
- 初始化 $B = \emptyset$
- 對於所有 i 從 1 到 n，考慮 e_i。若 $B \cup \{e_i\} \in \mathcal{I}$，那就將 e_i 放入 B 中
- 最後得到的 B 即為權重最小的基底

匹配 – 最小權基底

題目

有 n 個數字 $a_1, a_2, \ldots, a_n$，以及 $w_1, w_2, \ldots, w_n$。請找出 a 的一個子集，使得這個子集不管怎麼 XOR 都湊不出 0，且使得對應的 w_i 總和最大。

- $n \leq 10^5$
- $0 \leq a_i < 2^{30}$

匹配 – 最小權基底

題目

有 n 個數字 $a_1, a_2, \ldots, a_n$，以及 $w_1, w_2, \ldots, w_n$。請找出 a 的一個子集，使得這個子集不管怎麼 XOR 都湊不出 0，且使得對應的 w_i 總和最大。

- $n \leq 10^5$
- $0 \leq a_i < 2^{30}$

把每個數字當成 $\mathbb{F}_2^{30}$ 中的向量，XOR 湊不出 0 就代表選到的子集是線性獨立，可以令這樣的子集為獨立集，構造出擬陣，因此用 Greedy 作法就會是對的

使用線性基（高斯消去）等方法來判斷獨立性，複雜度 $O(n \log n + 30n)$

匹配 – 擬陣交

擬陣交 (Matroid Intersection): 對於集合 E，在 E 上定義兩種擬陣 $M_1 = (E, \mathcal{I}_1)$ 以及 $M_2 = (E, \mathcal{I}_2)$，求最大的 $I \in \mathcal{I}_1 \cap \mathcal{I}_2$

匹配 – 擬陣交

題目 (Faulty System)

給定兩張 N 點 M 邊的無向圖，第 i 條邊在第 j 張圖連接點 a_{ij} 與點 b_{ij}。請找出 $\{1, 2, \ldots, M\}$ 中最小的子集 T，使得加入編號在 T 中的邊之後，兩張圖都變得連通。

- $1 \leq N, M \leq 300$

匹配 – 擬陣交

題目 (Faulty System)

給定兩張 N 點 M 邊的無向圖，第 i 條邊在第 j 張圖連接點 a_{ij} 與點 b_{ij}。請找出 $\{1, 2, \ldots, M\}$ 中最小的子集 T，使得加入編號在 T 中的邊之後，兩張圖都變得連通。

- $1 \leq N, M \leq 300$

將題目當成拔掉最多邊讓圖依然連通。分別用兩張圖定義擬陣：邊集 $E' \subseteq E(G)$ 是獨立若且唯若 $G - E'$ 是連通的

匹配 – 對偶擬陣

對於一個擬陣 $M = (E, \mathcal{I})$，構造一個對應的 $M^* = (E, \mathcal{I}^*)$，其中 $E' \in \mathcal{I}^*$ 若存在 M 中的基底 B 使得 $E' \cap B = \emptyset$。M^* 稱為 M 的對偶 (Dual)

性質

對於擬陣 M，M 的對偶 M^* 也是擬陣，稱為 M 的對偶擬陣 (Dual Matroid)。

匹配 – 對偶擬陣

拔掉 E' 之後圖還是連通

$\rightarrow$ 在 $G - E'$ 中存在生成樹（基底）B，因此為圖擬陣的對偶

例題轉為擬陣交

匹配 – 擬陣交

將擬陣交與二分圖最大匹配進行聯想

對於一個二分圖 $G = (V, E)$，若 X 與 Y 是 G 的兩部分點集，我們建構以下兩個擬陣：

- $M_X = (E, \mathcal{I}_X)$，其中 $E' \in \mathcal{I}_X$ 若每個在 X 中的節點都至多與一條在 E' 中的邊相鄰
- $M_Y = (E, \mathcal{I}_Y)$，其中 $E' \in \mathcal{I}_Y$ 若每個在 Y 中的節點都至多與一條在 E' 中的邊相鄰

匹配 – 擬陣交

應用增廣路徑的想法，維護一個獨立集 I，每次將 I 裡的一些元素換成不在 I 裡面的元素

- 對於一個獨立集 I，構造它的**交換圖（exchange graph）** $\mathcal{D}_M(I)$ 為一個二分圖
- 以 I 以及 $E \setminus I$ 為兩邊的點集
- 對於 $x \in I$ 以及 $y \in S \setminus I$，$\mathcal{D}_M(I)$ 中有 (x, y) 這條邊若 $(I \setminus \{x\}) \cup \{y\}$ 亦是獨立集

也就是說，一條邊代表一對可以交換的元素

匹配 – 擬陣交

引理

對於兩個相同大小的獨立集 I_1 以及 I_2，則 $I_1 \setminus I_2 \subseteq I_1$ 以及 $I_2 \setminus I_1 \subseteq E \setminus I_1$ 於 $\mathcal{D}_M(I_1)$ 中有完美匹配。

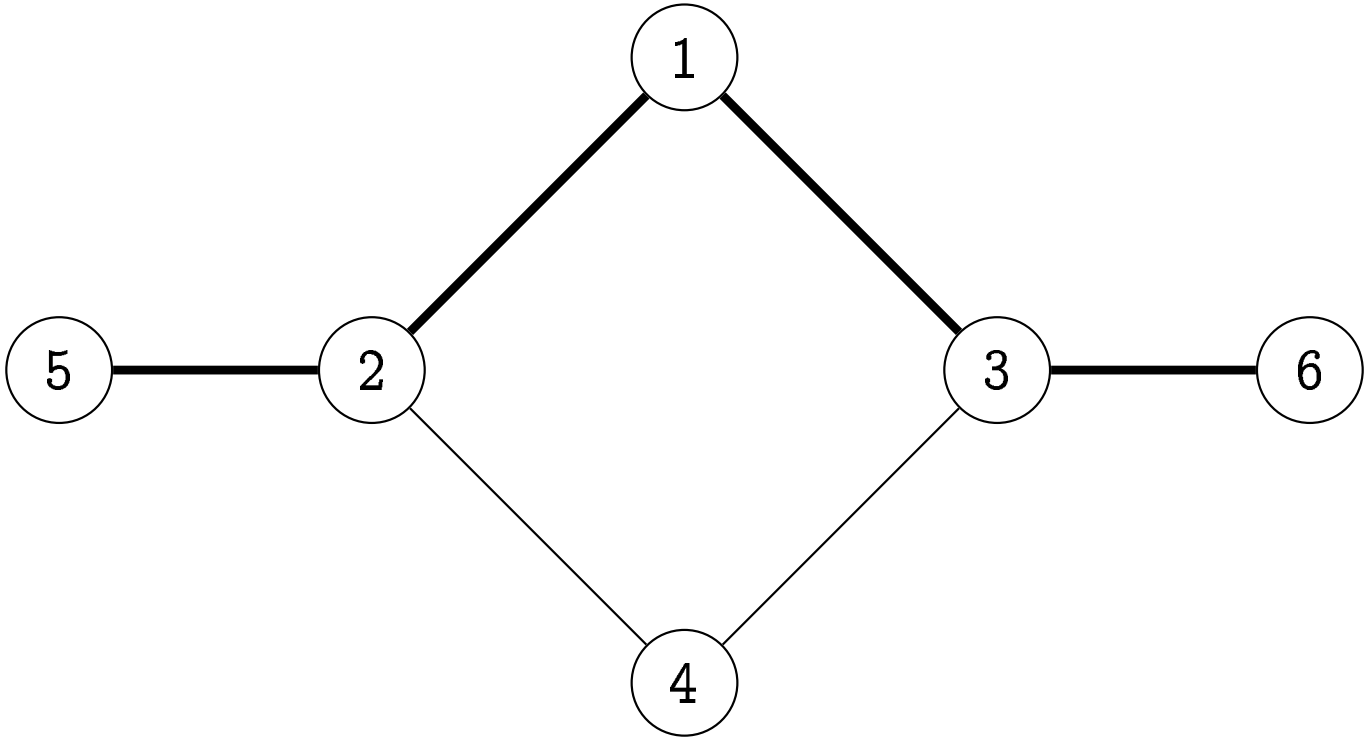
匹配 – 擬陣交

引理

對於兩個相同大小的獨立集 I_1 以及 I_2，則 $I_1 \setminus I_2 \subseteq I_1$ 以及 $I_2 \setminus I_1 \subseteq E \setminus I_1$ 於 $\mathcal{D}_M(I_1)$ 中有完美匹配。

這個性質的反方向不一定會成立：對於 $|I_1| = |I_2|$，I_1 是獨立集而且 $\mathcal{D}_M(I_1)$ 中有 $I_1 \setminus I_2$ 與 $I_2 \setminus I_1$ 的完美匹配並不能保證 I_2 是獨立集

匹配 – 擬陣交



考慮上圖的圖擬陣 M，$(2,5)$ 可以單獨被換到 $(2,4)$、$(3,6)$ 可以單獨被換到 $(3,4)$，但兩個一起換就會出現環（迴路）。不過，注意到這組交換圖的完美匹配並不是唯一的：把 $(2,5)$ 換到 $(3,4)$ 再把 $(3,6)$ 換到 $(2,4)$ 也是另一組匹配

匹配 – 擬陣交

引理

若 I_1 是獨立集且 $\mathcal{D}_M(I_1)$ 中 $I_1 \setminus I_2$ 以及 $I_2 \setminus I_1$ 有**唯一**的完美匹配的話，則 I_2 也是獨立集。

匹配 – 擬陣交

試著將一個擬陣的交換圖推廣到兩個擬陣的情況

- 對於 $M_1 = (E, \mathcal{I}_1)$ 與 $M_2 = (E, \mathcal{I}_2)$ 以及 $I \in \mathcal{I}_1 \cap \mathcal{I}_2$，定義交換圖 $\mathcal{D}_{M_1, M_2}(I)$ 為一個**有向**二分圖
- 兩個點集分別是 I 以及 $E \setminus I$
- 對於 $x \in I$ 以及 $y \in E \setminus I$，有 x 到 y 的邊若 $(I \setminus \{x\}) \cup \{y\} \in \mathcal{I}_1$、有 y 到 x 的邊若 $(I \setminus \{x\}) \cup \{y\} \in \mathcal{I}_2$

也就是說，跟一個擬陣時的交換圖一樣，只是現在邊的方向取決於將 x 與 y 交換後，I 是在 M_1 中獨立還是在 M_2 中獨立（有可能兩者皆成立，這樣兩個方向的邊就會都存在）

匹配 – 擬陣交

試著從 $I = \emptyset$ 開始，每次嘗試將 I 的大小增加一，直到不能再增加為止

令 $X_1 = \{e \in E \setminus I \mid I \cup \{e\} \in \mathcal{I}_1\}$，
$X_2 = \{e \in E \setminus I \mid I \cup \{e\} \in \mathcal{I}_2\}$

匹配 – 擬陣交

如果 X_1 與 X_2 有共同的元素 e 的話，直接將 e 加入 I 中

如果 X_1 與 X_2 有一個是空的話，那代表 I 已經是其中一個擬陣的基底

否則要想辦法擴增

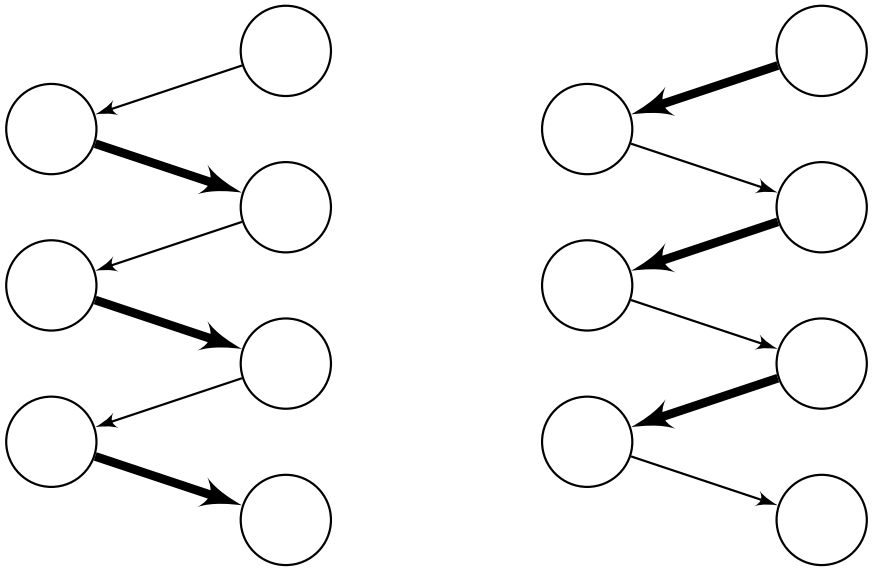
匹配 – 擬陣交

考慮交換圖 $\mathcal{D}_{M_1,M_2}(I)$，並且找一條從 X_1 中的點到 X_2 中的點且**沒有捷徑的路徑** P

由於交換圖是有向二分圖，左往右的邊與右往左的邊一定會在 P 中交互出現，只看同一方向的邊的話，這些邊將會形成一組匹配

再加上 P 沒有捷徑，因此對於任一個方向都有唯一的完美匹配

匹配 – 擬陣交



不過由於 X_1 與 X_2 都位於右邊，因此 P 的長度其實是奇數。這就代表，只看右到左的邊的話，形成的匹配會少了 v_k，而只看左到右的話，形成的匹配會少了 v_1，所以不能直接套用引理說明正確性

匹配 – 擬陣交

增加一個新的元素 t 到 E 裡面。這個 t 在 M_1 以及 M_2 中都與其他元素獨立，也就是說有新的兩個擬陣：

$$M_1' = (E \cup \{t\}, \{E' \subseteq E \cup \{t\} \mid E' \setminus \{t\} \in \mathcal{I}_1\})$$
$$M_2' = (E \cup \{t\}, \{E' \subseteq E \cup \{t\} \mid E' \setminus \{t\} \in \mathcal{I}_2\})$$

匹配 – 擬陣交

將 t 也暫時加入 I，由於 $v_1 \in X_1, v_k \in X_2$，交換圖上會有 $(t, v_1), (v_k, t)$ 兩條邊，這時會形成一個有向環，而且對於其中一種方向的邊都有唯一的完美匹配

套用先前的引理，我們可以將左邊的元素（包含 t）與右邊的元素作交換得到一組大小與 I 相同且在 M_1' 中與 M_2' 中都獨立的 I'

由於 I' 中的 t 已經被換走了，根據定義 I' 會在 M_1 與 M_2 中都獨立，且大小比 I （不包含 t）多一，這也就說明了上面直接交換 P 中的點的正確性

匹配 – 擬陣交

$I \leftarrow \emptyset$
$X_1 \leftarrow \{e \in E \setminus I \mid I \cup \{e\} \in \mathcal{I}_1\}$
$X_2 \leftarrow \{e \in E \setminus I \mid I \cup \{e\} \in \mathcal{I}_2\}$
while $X_1 \neq \emptyset$ 且 $X_2 \neq \emptyset$ **do**
  if $e \in X_1 \cap X_2$ **then**
    $I \leftarrow I \cup \{e\}$
  else
    構造交換圖 $\mathcal{D}_{M_1,M_2}(I)$
    在交換圖上找到一條 X_1 到 X_2 且沒有捷徑的路徑 P
    if $P = null$ **then**
      break
    end
    $I \leftarrow I \triangle P$
  end
  $X_1 \leftarrow \{e \in E \setminus I \mid I \cup \{e\} \in \mathcal{I}_1\}$
  $X_2 \leftarrow \{e \in E \setminus I \mid I \cup \{e\} \in \mathcal{I}_2\}$
end

匹配 – 擬陣交

在找不到新的路徑時，就代表我們得到最大的擬陣交了

假設答案的大小是 r，算法會在 r 次之後結束，每次建交換圖要檢查 $O(rn)$ 條邊是否存在，找 X_1, X_2 需要檢查 $O(n)$ 個元素能不能加入，找路徑 P 可以用 BFS 在 $O(rn)$ 時間找到，總複雜度是 $O(r^2 n T_{\text{ind}})$，其中 T_{ind} 是檢查一個子集是否為獨立集的時間

如果用類似 Hopcroft-Karp 的分層 BFS 加 DFS，時間複雜度會被優化成 $O(r^{1.5} n T_{\text{ind}})$

匹配 – 帶權擬陣交

現在每個元素 e 都有非負權重 $w(e)$，帶權擬陣交希望找到 $I \in \mathcal{I}_1 \cap \mathcal{I}_2$ 使得 I 中的元素權重總和最大

一般來說 $M' = (E, \mathcal{I}_1 \cap \mathcal{I}_2)$ 並不會是個擬陣，所以不能簡單地套用 Rado-Edmonds 演算法來得到答案。

匹配 – 帶權擬陣交

稍微更改一般擬陣交的算法，在每次把 I 的大小增加一的時候，維持 I 是所有大小為 $|I|$ 且位於 $\mathcal{I}_1 \cap \mathcal{I}_2$ 的獨立集中權重最大的

在交換圖中，我們給每個點 e 點權 $l(e)$：若 e 在 I 中，則 $l(e) = w(e)$，若 e 不在 I 中，則 $l(e) = -w(e)$。接著找交換圖上 X_1 到 X_2 中，經過點權和最小且沒有捷徑的一條路徑並翻轉路徑上元素所在的集合

在這個點權函式的定義下，最短的路徑其實就對應到將路徑翻轉後 I 的權重增加最大的一種選法，持續這個操作直到沒辦法找到新元素為止，就得到最大權的擬陣交了

最短路

最短路 – 同餘最短路

利用同餘來構造狀態，以最短路優化轉移的時間複雜度

- 狀態的轉移：$f((i+x) \bmod N) \leq f(i) + y$
- 最短路：$f(v) \leq f(u) + \mathsf{edge}(u, v)$

最短路 – 同餘最短路

題目 (跳樓機)

給定 a, b, c，求 $[0, h)$ 之間有多少個相異數字可以寫成 $ax + by + cz$，其中 x, y, z 為非負整數。

- $1 \leq a, b, c \leq 10^5$
- $1 \leq h \leq 2^{63} - 1$

最短路 – 同餘最短路

假設先使用 a, b 湊出了數字 i，那所有 $j \geq i$ 且 $j \equiv i \pmod{c}$ 就都能得到

最短路 – 同餘最短路

假設先使用 a, b 湊出了數字 i，那所有 $j \geq i$ 且 $j \equiv i \pmod{c}$ 就都能得到

對模 c 的每一種餘數判斷 a, b 湊出的最小數

最短路 – 同餘最短路

假設先使用 a, b 湊出了數字 i，那所有 $j \geq i$ 且 $j \equiv i \pmod{c}$ 就都能得到

對模 c 的每一種餘數判斷 a, b 湊出的最小數

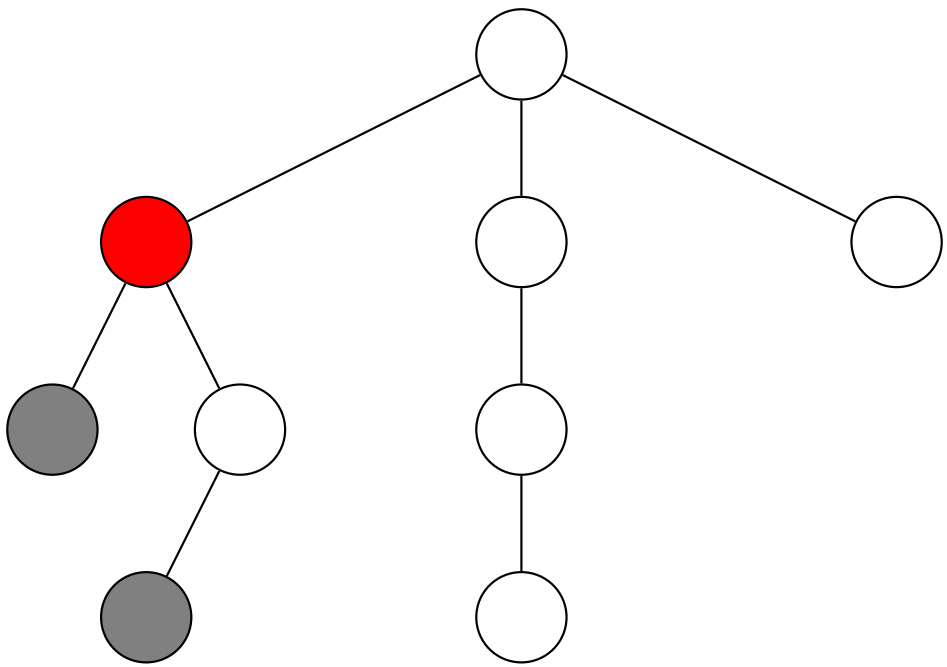
對每個餘數 i 建立一個節點，分別連一條權重 a, b 的邊到 $(i + a) \bmod c$ 和 $(i + b) \bmod c$，再從 0 開始跑最短路，`dis[`i`]` 就會是餘 i 的最小數字

樹論

樹論 – 最低共同祖先

定義

樹上兩點 u 以及 v 的最低共同祖先 (Lowest Common Ancestor, LCA) 為 u, v 的共同祖先中，深度最深的節點。



樹論 – 最低共同祖先

- 倍增法
- 樹上尤拉迴路
- Tarjan 離線算法

樹論 – 最低共同祖先

求 u, v 兩點的 LCA

- 如果兩點深度不一樣，那麼深度較深的顯然不會是 LCA，因此可以把那個點沿著父邊往上跳，直到詢問的兩點深度相同
- 如果還沒找到 LCA，就讓兩個點一起往上跳，直到他們的父節點相同時，那個父節點就會是 LCA。

樹論 – 倍增法

跳大步一點

假如我們知道每個節點的所有 2^i 輩的祖先，每次要往上跳時直接跳到最大可以跳的 2^i 輩祖先的位置，那就可以在 $O(\log N)$ 的時間內跳到該去的位置

樹論 – 倍增法

把較深的節點 u 跳到和 v 一樣深

- 比較 u 的 $2^{\lfloor \log N \rfloor}$ 輩祖先和 v 的深度
- 如果 v 較深，代表會跳太高，所以繼續檢查 $2^{\lfloor \log N \rfloor - 1}$ 輩祖先
- 如果 v 較淺或是一樣深，把 u 移動到 $2^{\lfloor \log N \rfloor}$ 輩祖先的位置，再繼續檢查新的 u 的 $2^{\lfloor \log N \rfloor - 1}$ 輩祖先
- 依序檢查 $2^{\lfloor \log N \rfloor - 2}, \ldots, 2^0$ 輩，最後 u 就會跳到和 v 一樣高

樹論 – 倍增法

如果 $u \neq v$，把一樣深的節點 u 和 v 跳到最低共同祖先的下面一層

- 比較 u 的 $2^{\lfloor \log N \rfloor}$ 輩祖先和 v 的 $2^{\lfloor \log N \rfloor}$ 輩祖先是否相同
- 如果相同，代表會跳太高，所以繼續檢查 $2^{\lfloor \log N \rfloor - 1}$ 輩祖先
- 如果不同，把 u, v 分別移動到 $2^{\lfloor \log N \rfloor}$ 輩祖先的位置，再繼續檢查新的 u, v 的 $2^{\lfloor \log N \rfloor - 1}$ 輩祖先
- 依序檢查 $2^{\lfloor \log N \rfloor - 2}, \ldots, 2^0$ 輩
- 此時 u 和 v 的父親皆為它們的最低共同祖先

樹論 – 倍增法

預處理 v 的第 i 輩祖先：

- 先找到每個點的父節點 (1 輩祖先)
- 2 輩祖先會是父節點的父節點
- 4 輩祖先會是 2 輩祖先的 2 輩祖先
- ...

依照這個順序，每次可以 $O(1)$ 找到祖先，全部找完要 $O(N \log N)$

樹論 – 倍增法

由 u 走到 v 的簡單路徑一定是 $u \to LCA(u, v) \to v$

在倍增的過程同時維護那一段路的資訊，可能可以快速查詢 u 到 v 的路徑的資訊

樹論 – 倍增法

題目 (路徑最小值)

給定一棵帶點權的樹以及多筆詢問，每次詢問兩個點路徑上的點權最小值。

樹論 – 倍增法

題目 (路徑最小值)

給定一棵帶點權的樹以及多筆詢問，每次詢問兩個點路徑上的點權最小值。

u, v 路徑上的最小值就是「u 到 $LCA(u, v)$ 的最小值」與「v 到 $LCA(u, v)$ 的最小值」中較小的那個

在維護 2^i 輩祖先的同時維護 v **到** v **的** 2^i **輩祖先路徑上的最小值**，在查詢 LCA 的時候一邊往上爬、一邊更新答案。

樹論 – 倍增法

題目 (路徑最大連續和)

給定一棵帶邊權的樹以及多筆詢問，每次詢問兩個點路徑上邊權序列所能找到的最大連續和。

樹論 – 倍增法

題目 (路徑最大連續和)

給定一棵帶邊權的樹以及多筆詢問，每次詢問兩個點路徑上邊權序列所能找到的最大連續和。

維護 v **到** v **的** 2^i **輩祖先路徑上的最大連續和、最大前綴、最大後綴、邊權和**（和區間查詢最大連續和要維護的資訊一模一樣）

樹論 – 樹上尤拉迴路

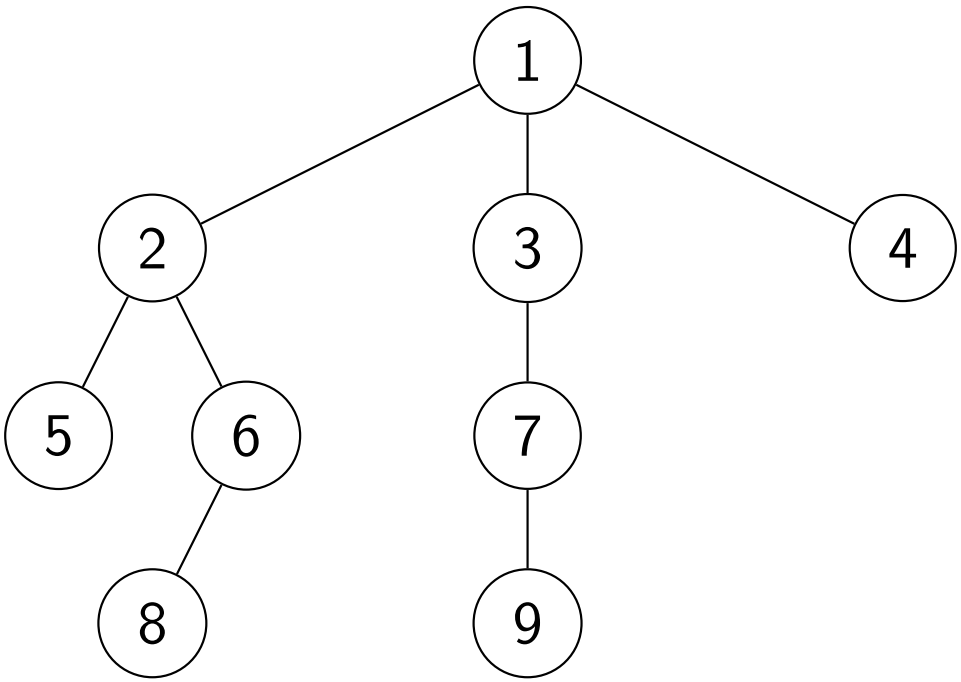
樹上尤拉迴路是一種將樹的結構轉化成序列的技巧。如果將雙向邊改為兩條來回的單向邊，從根開始 DFS 完整棵樹再回到根，恰好是一個尤拉迴路

每當拜訪到一個節點或是拜訪完它的一個子節點，把它紀錄下來，會得到一個長度為 $2N - 1$ 的序列

樹論 – 樹上尤拉迴路

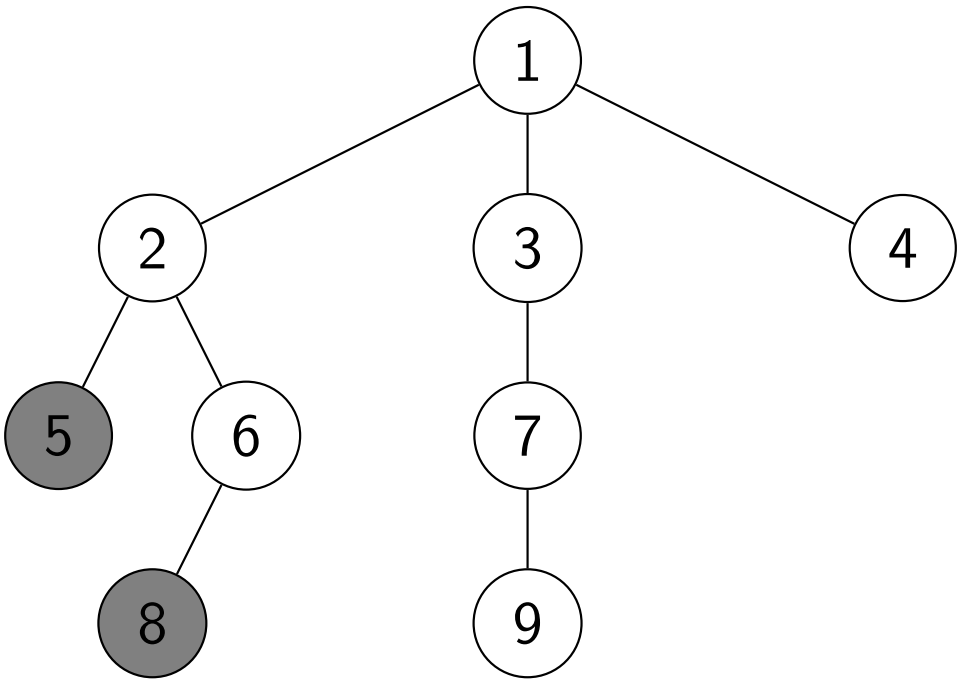
對於兩個節點 u 與 v，如果標出它們在序列上的任意出現位置，這兩個位置所形成的區間中深度最小的節點就會是 u 與 v 的最低共同祖先，因此可以將 LCA 轉為靜態的 RMQ 問題

樹論 – 樹上尤拉迴路



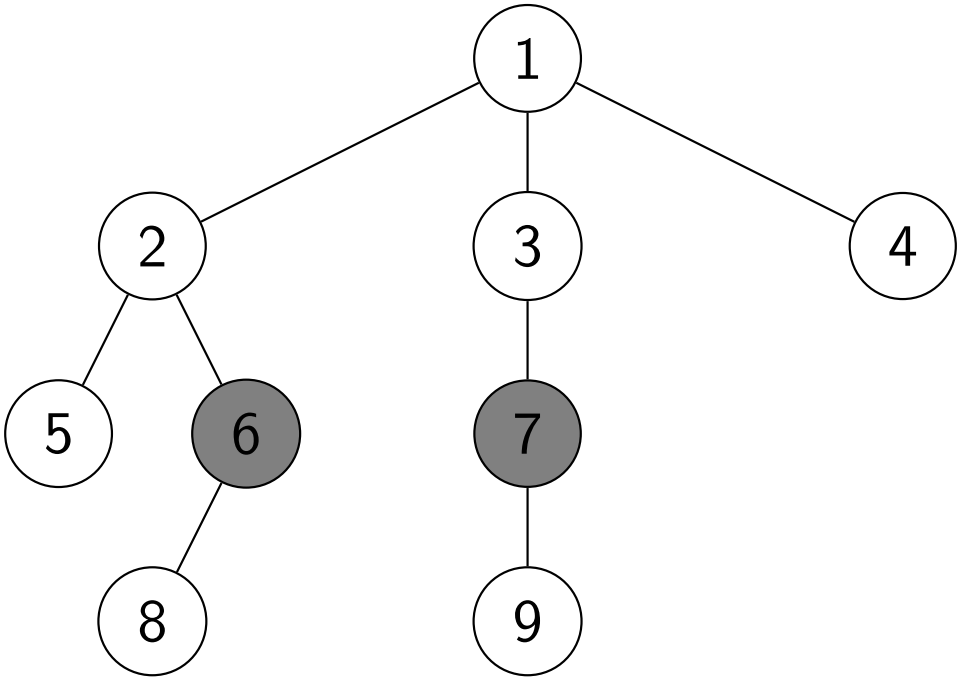
1 2 5 2 6 8 6 2 1 3 7 9 7 3 1 4 1

樹論 – 樹上尤拉迴路



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

樹論 – 樹上尤拉迴路

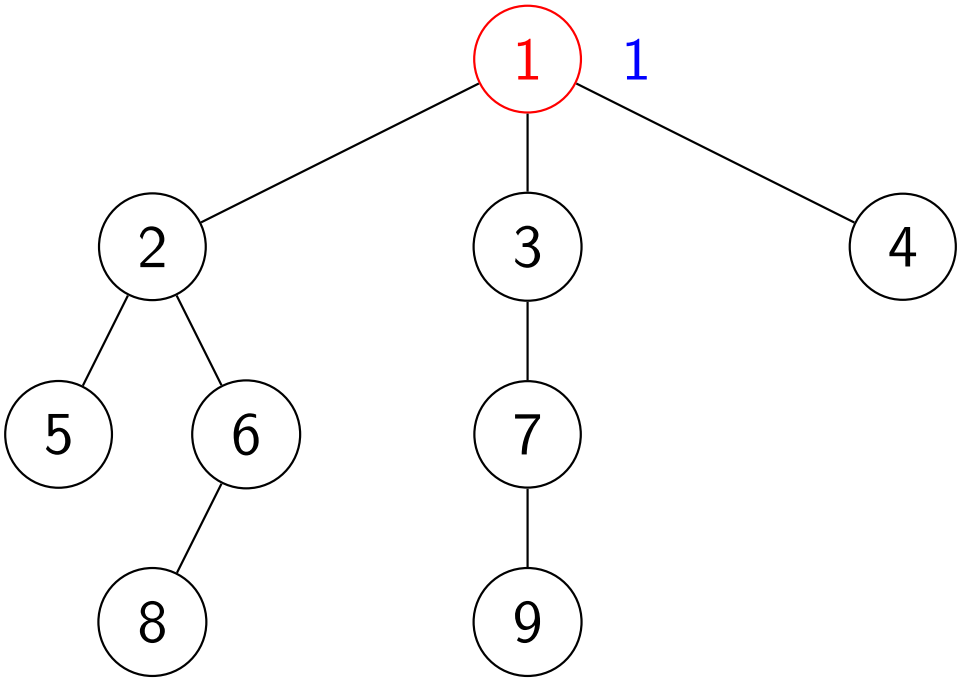


1 2 5 2 **6 8 6 2 1 3 7** 9 7 3 1 4 1

樹論 – Tarjan 離線算法

- 在 DFS 過程維護好所有前面已經經過的點和下一個要拜訪的點的 LCA
- 對於每個詢問的點對，在 DFS 到比較晚遇到的那個點時回答
- 當我們 DFS 完點 v 之後，接下來可能會去拜訪 v 的父節點 u 的其他子樹，這時候因為 v 的子樹中的點和 u 其他子樹中的點的 LCA 會是 u，因此必須把 v 子樹中所有點標記的祖先改成 u，這個操作可以使用並查集來維護

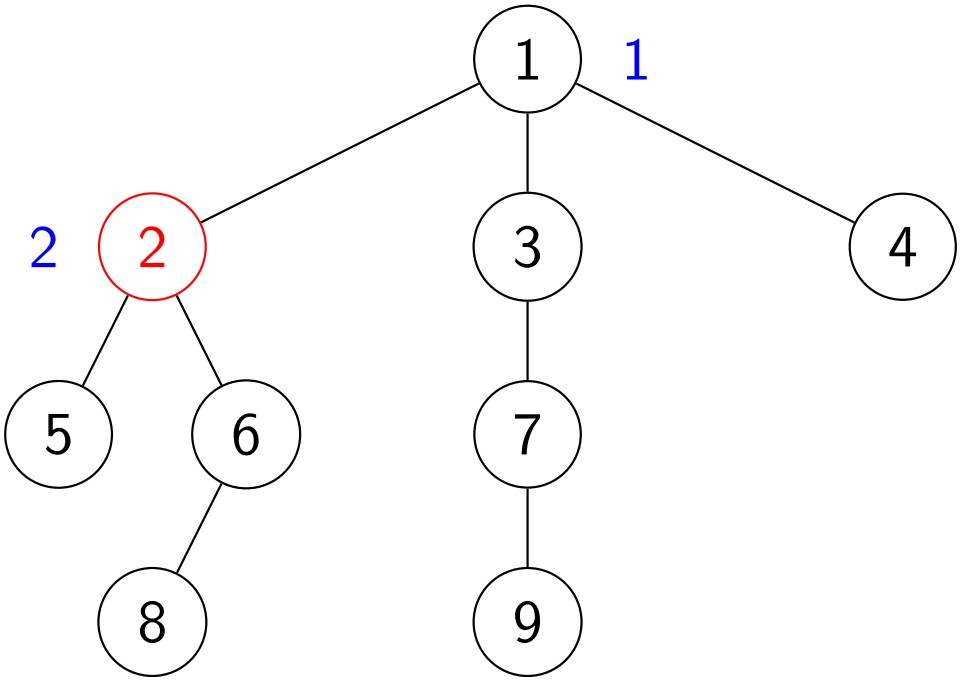
樹論 – Tarjan 離線算法



LCA(5, 8) =

LCA(6, 7) =

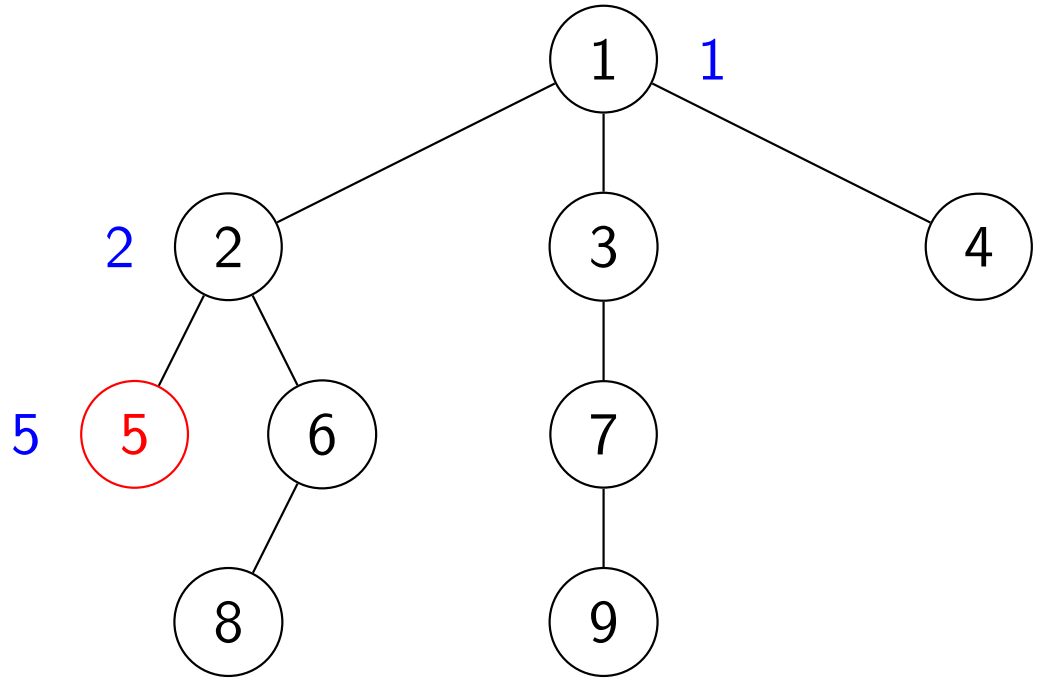
樹論 – Tarjan 離線算法



LCA(5, 8) =

LCA(6, 7) =

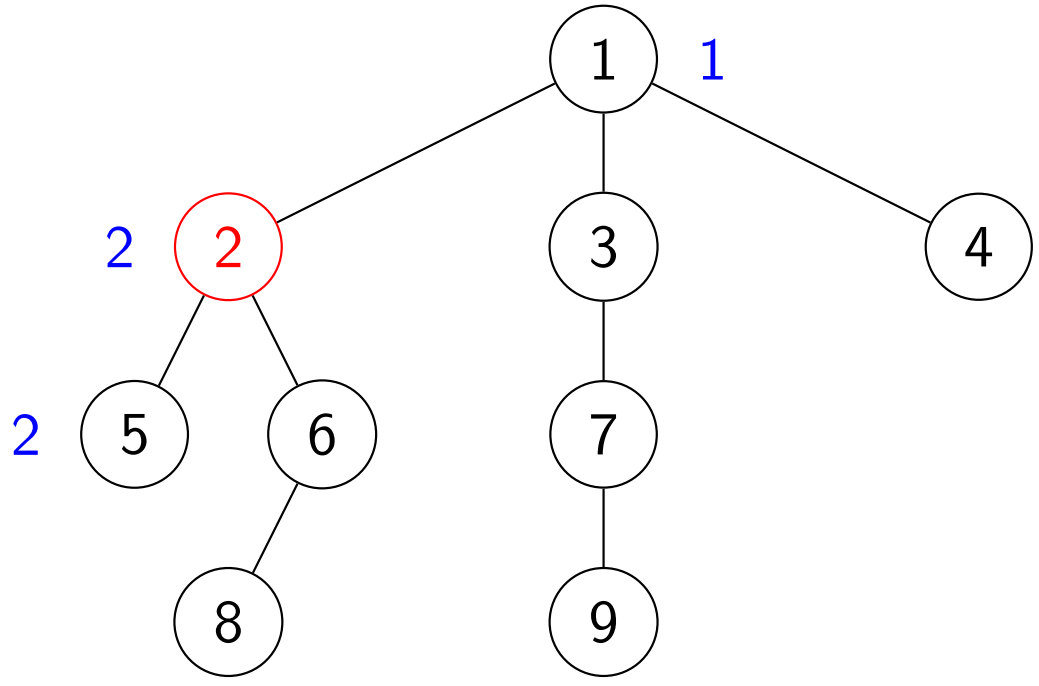
樹論 – Tarjan 離線算法



LCA(5, 8) =

LCA(6, 7) =

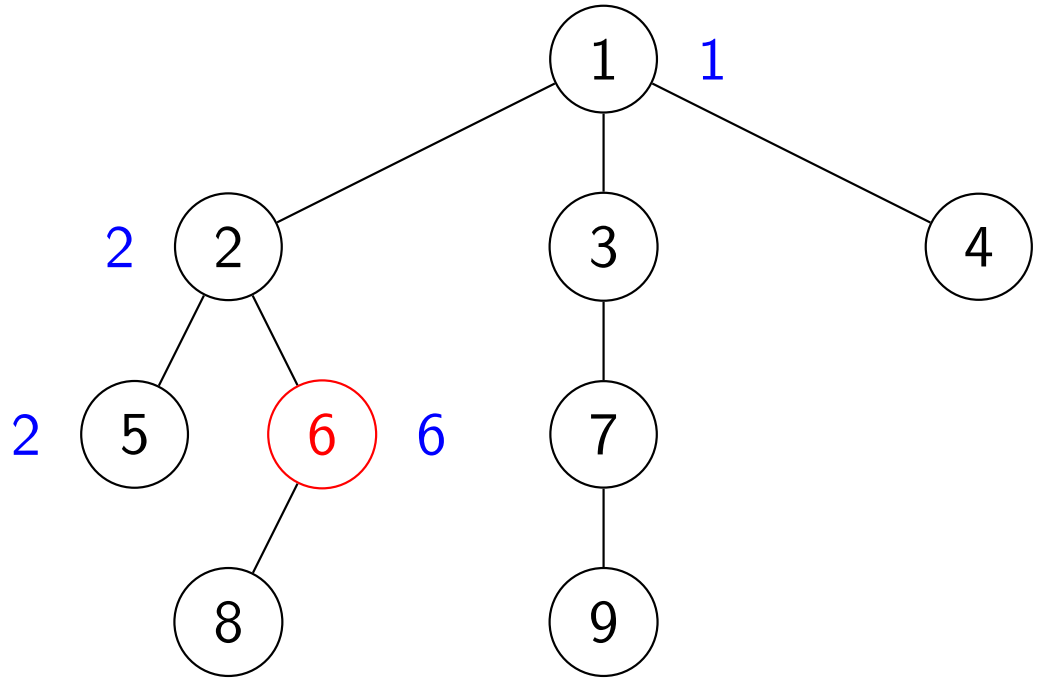
樹論 – Tarjan 離線算法



LCA(5, 8) =

LCA(6, 7) =

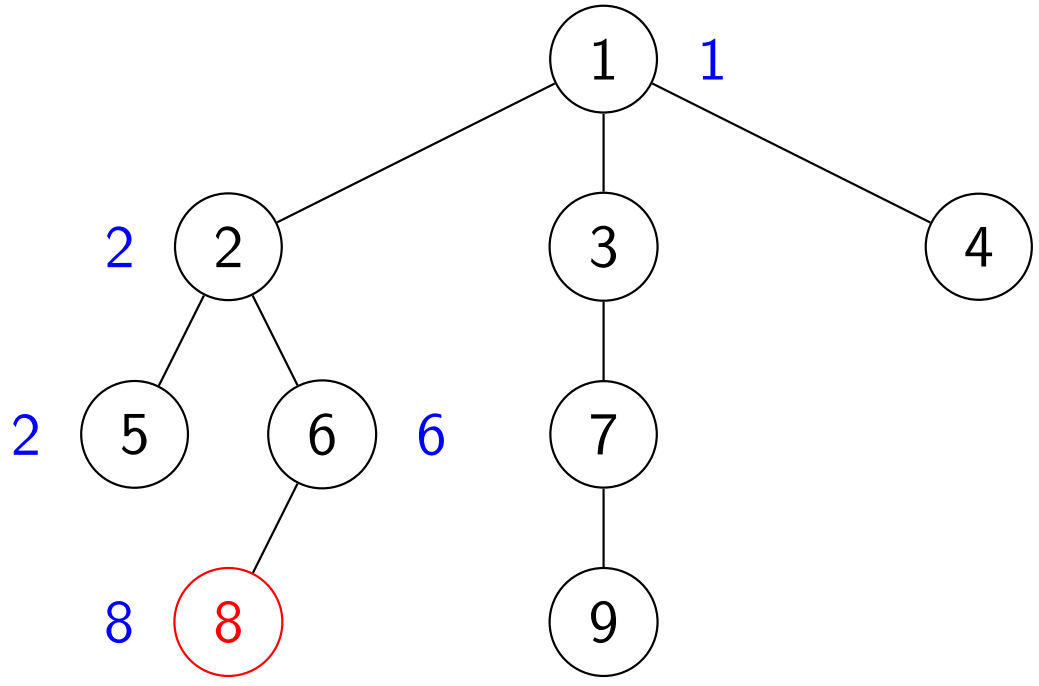
樹論 – Tarjan 離線算法



LCA(5, 8) =

LCA(6, 7) =

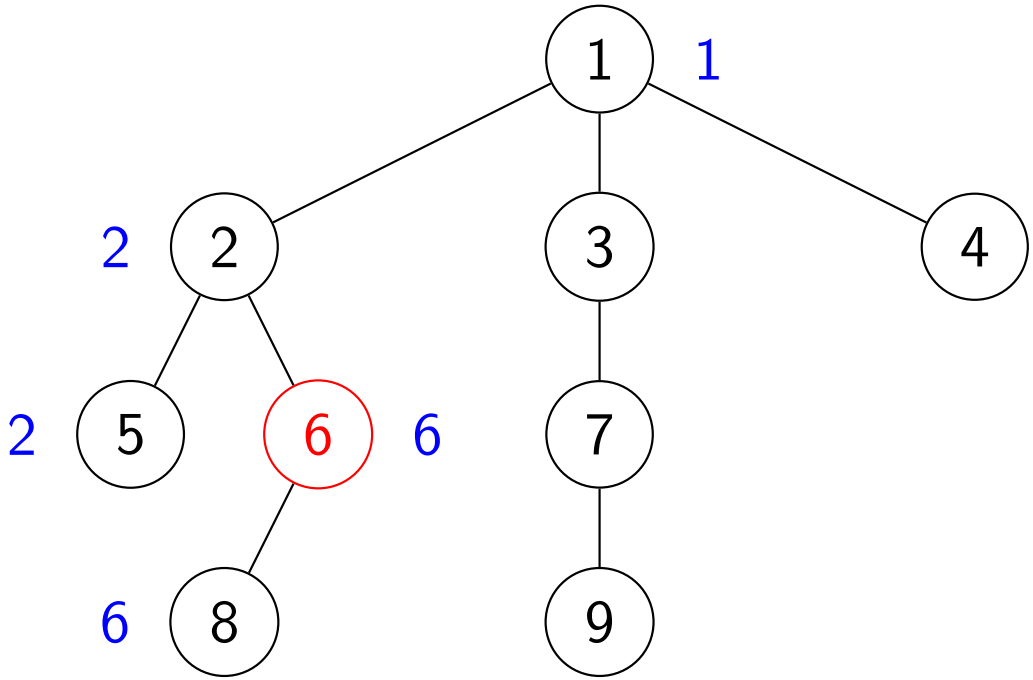
樹論 – Tarjan 離線算法



LCA(5, 8) = 2

LCA(6, 7) =

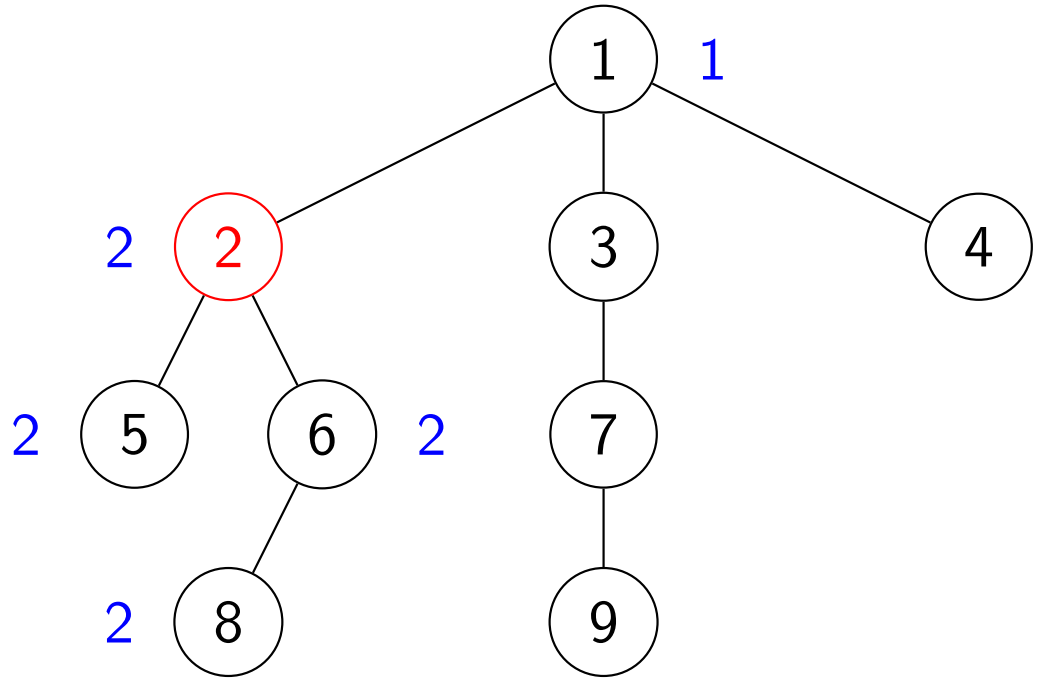
樹論 – Tarjan 離線算法



LCA(5, 8) = 2

LCA(6, 7) =

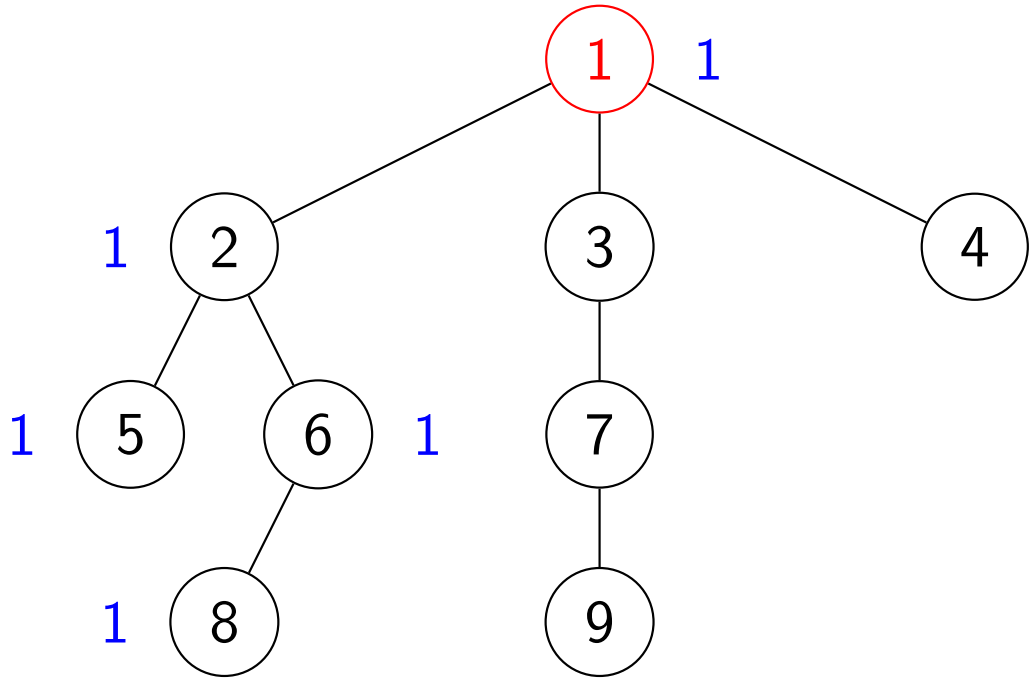
樹論 – Tarjan 離線算法



LCA(5, 8) = 2

LCA(6, 7) =

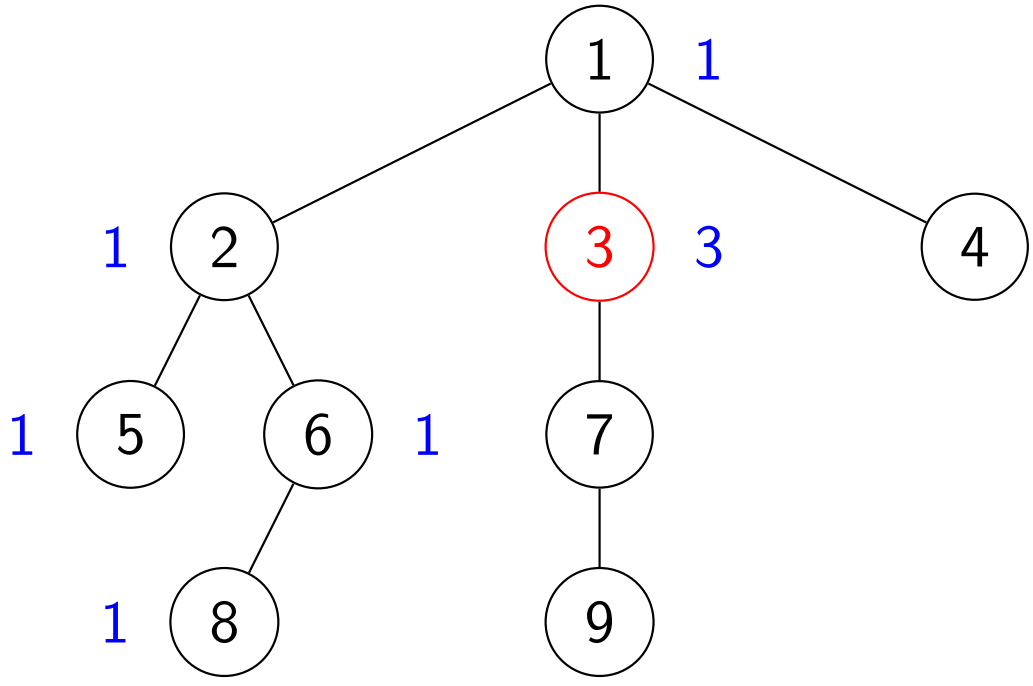
樹論 – Tarjan 離線算法



LCA(5, 8) = 2

LCA(6, 7) =

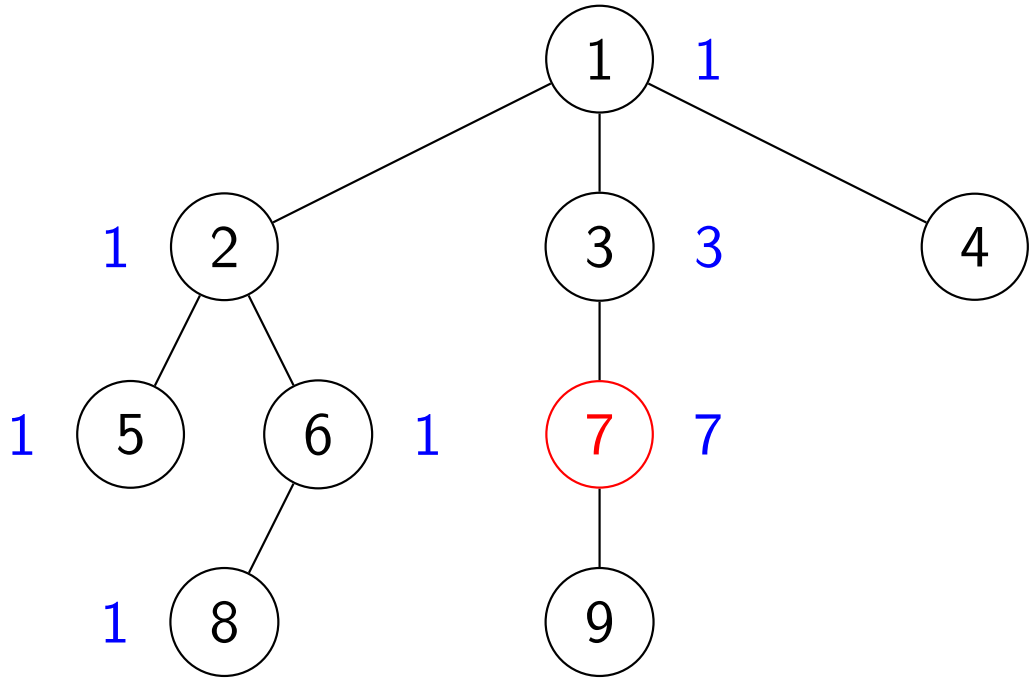
樹論 – Tarjan 離線算法



LCA(5, 8) = 2

LCA(6, 7) =

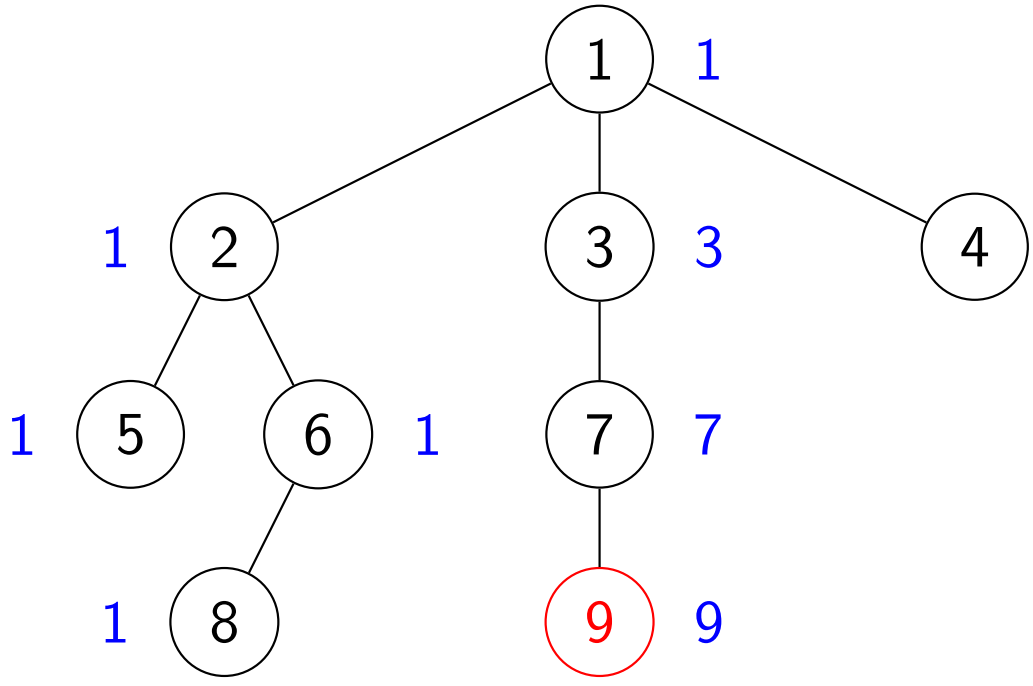
樹論 – Tarjan 離線算法



$\mathrm{LCA}(5,\ 8) = 2$

$\mathrm{LCA}(6,\ 7) = 1$

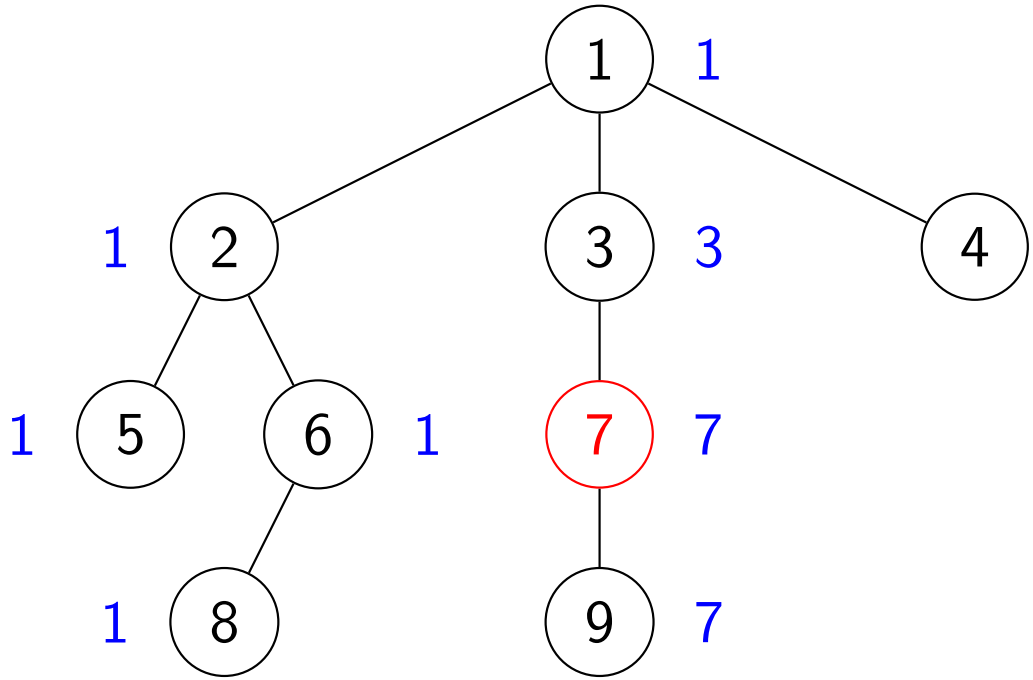
樹論 – Tarjan 離線算法



$\mathrm{LCA}(5,\ 8) = 2$

$\mathrm{LCA}(6,\ 7) = 1$

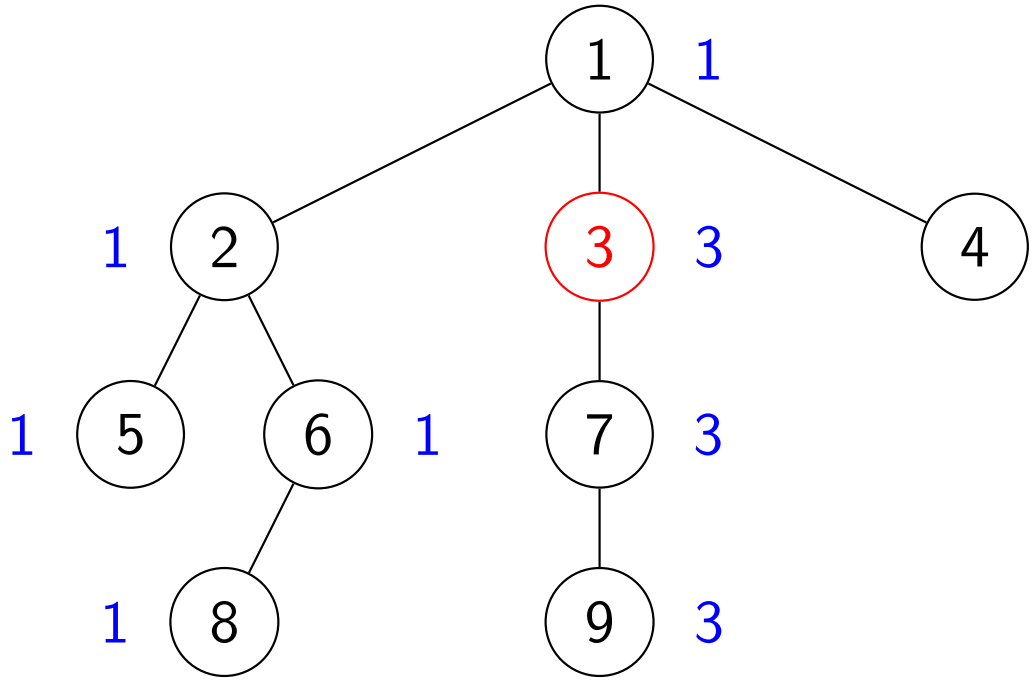
樹論 – Tarjan 離線算法



$\mathrm{LCA}(5,\ 8) = 2$

$\mathrm{LCA}(6,\ 7) = 1$

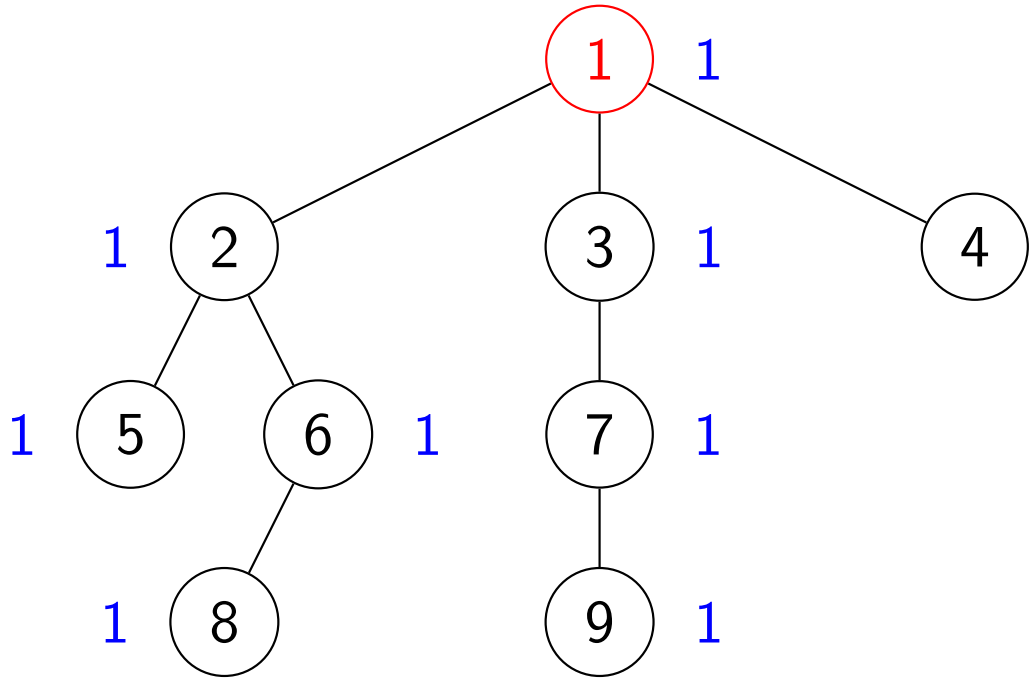
樹論 – Tarjan 離線算法



$\mathrm{LCA}(5,\ 8) = 2$

$\mathrm{LCA}(6,\ 7) = 1$

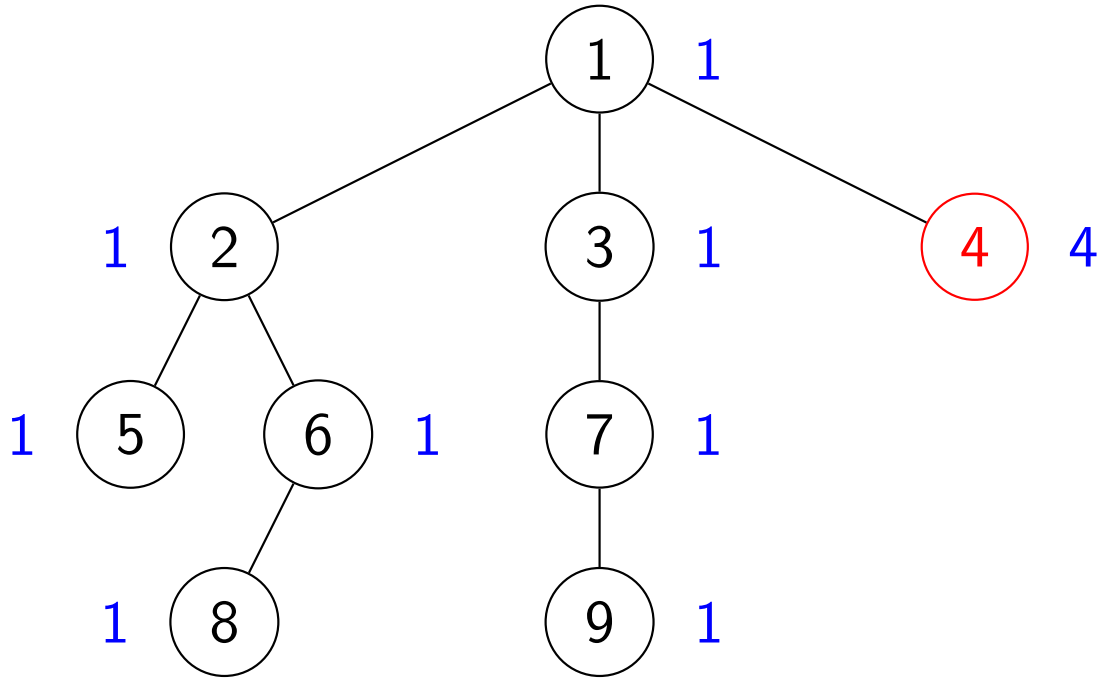
樹論 – Tarjan 離線算法



$\mathrm{LCA}(5,\ 8) = 2$

$\mathrm{LCA}(6,\ 7) = 1$

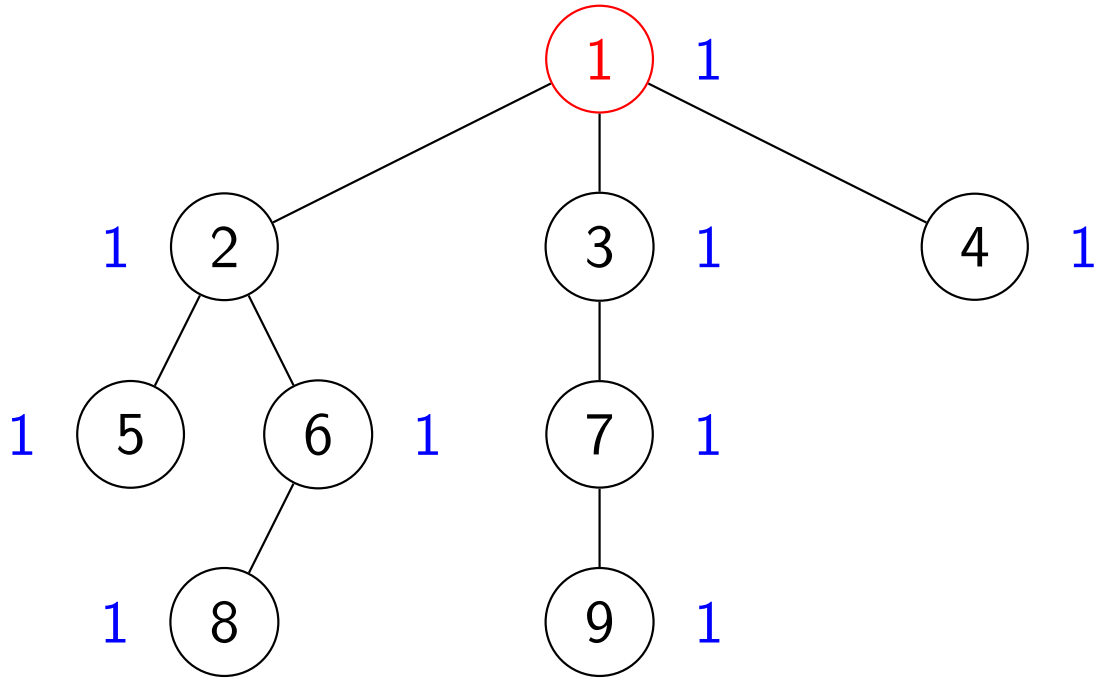
樹論 – Tarjan 離線算法



$\mathrm{LCA}(5,\ 8) = 2$

$\mathrm{LCA}(6,\ 7) = 1$

樹論 – Tarjan 離線算法



$\mathrm{LCA}(5,\ 8) = 2$

$\mathrm{LCA}(6,\ 7) = 1$

樹論 – Tarjan 離線算法

```
void dfs(int v, int p) {
  vis[v] = true;
  for (auto [u, qid] : queries[v]) {
    if (!vis[u]) continue;
      ans[qid] = ancestor[Find(u)];
  }
  for (int u : g[v]) {
    if (u == p) continue;
    dfs(u, v);
    Union(u, v);
    ancestor[Find(v)] = v;
  }
}
```

樹論 – Tarjan 離線算法

複雜度是 $O(N + Q\alpha(N))$，還能進一步優化到 $O(N + Q)$