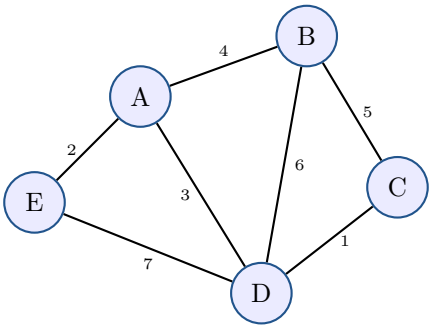
圖論演算法完整教材

Graph Algorithms —A Complete Guide

從圖的表示、走訪、最短路徑、最小生成樹、
網路流、進階結構，到真實世界的應用與解法



演算法教學系列

內附完整、可編譯的 C++17 實作（17 個核心演算法）

編譯方式：`xelatex` 連續執行兩次以解析目錄與交叉引用

Contents

前言：為什麼要學圖論演算法 **4**

1 圖論基礎與術語 **5**

1.1 什麼是圖 . . . 5
1.2 有向圖與無向圖 . . . 5
1.3 加權圖與無權圖 . . . 5
1.4 重要術語速覽 . . . 6
1.5 圖的稀疏性與演算法選擇 . . . 6

2 圖的表示法 **7**

2.1 鄰接矩陣 (Adjacency Matrix) . . . 7
2.2 鄰接串列 (Adjacency List) . . . 7
2.3 邊串列 (Edge List) . . . 8
2.4 三種表示法的比較 . . . 8
2.5 隱式圖 (Implicit Graph) . . . 9

3 圖的走訪：BFS 與 DFS **10**

3.1 廣度優先搜尋 (BFS) . . . 10
3.1.1 演算法流程 . . . 10
3.1.2 核心性質 . . . 10
3.1.3 BFS 的延伸應用 . . . 11
3.2 深度優先搜尋 (DFS) . . . 11
3.2.1 DFS 的時間戳與邊分類 . . . 11
3.3 BFS 與 DFS 的比較 . . . 12

4 連通性分析 **13**

4.1 連通分量 (Connected Components) . . . 13
4.2 二分圖判定 (Bipartite Check) . . . 14
4.3 強連通分量 (SCC, Tarjan 演算法) . . . 14
4.4 橋與割點 (Bridges & Articulation Points) . . . 15

5 拓撲排序 (Topological Sort) **16**

5.1 問題定義 . . . 16
5.2 Kahn 演算法（BFS 式） . . . 16
5.3 DFS 式拓撲排序 . . . 17
5.4 應用 . . . 17

6 環偵測 (Cycle Detection) 18
6.1 有向圖：DFS 三色法 . . . 18
6.2 無向圖：DFS 父節點法或並查集 . . . 18
6.3 偵測負權環 . . . 19
6.4 應用對照 . . . 19

7 最短路徑演算法 20
7.1 問題分類 . . . 20
7.2 Dijkstra 演算法 . . . 20
7.2.1 核心思想 . . . 20
7.2.2 以優先佇列實作 . . . 20
7.2.3 手算範例 . . . 21
7.2.4 重建路徑 . . . 21
7.3 0-1 BFS . . . 21
7.4 Bellman-Ford 演算法 . . . 22
7.4.1 核心思想 . . . 22
7.5 Floyd-Warshall 演算法 . . . 22
7.6 A* 搜尋 . . . 23
7.7 最短路演算法選擇指南 . . . 23

8 最小生成樹 (Minimum Spanning Tree) 24
8.1 問題定義 . . . 24
8.2 Kruskal 演算法 . . . 24
8.3 Prim 演算法 . . . 25
8.4 Kruskal vs. Prim . . . 25
8.5 應用 . . . 25

9 並查集 (Disjoint Set Union) 26
9.1 問題與動機 . . . 26
9.2 兩大優化：路徑壓縮 + 按秩合併 . . . 26
9.3 應用 . . . 27

10 網路流 (Network Flow) 28
10.1 最大流問題 . . . 28
10.2 Ford-Fulkerson 與 Edmonds-Karp . . . 28
10.3 最大流最小割定理 . . . 29
10.4 建模的藝術：問題化約 . . . 29

11 二分匹配 (Bipartite Matching) 30
11.1 問題定義 . . . 30
11.2 匈牙利演算法（增廣路） . . . 30
11.3 應用 . . . 31

12 進階主題 **32**
12.1 最近共同祖先 (LCA) . . . 32
12.2 歐拉路徑與歐拉迴路 . . . 32
12.3 最小生成樹的延伸 . . . 32
12.4 圖著色與最大團 . . . 32
12.5 現代延伸：圖神經網路 (GNN) . . . 33

13 解題策略：如何辨識題型 **34**
13.1 第一步：辨識圖的結構 . . . 34
13.2 第二步：題型 → 演算法對照表 . . . 34
13.3 第三步：常見陷阱檢查表 . . . 35

14 真實世界的應用 **36**
14.1 導航與地圖 . . . 36
14.2 社群網路 . . . 36
14.3 詐欺偵測與金融 . . . 36
14.4 推薦系統 . . . 36
14.5 基礎設施與供應鏈 . . . 37
14.6 編譯器、建置與排程 . . . 37
14.7 生物資訊與其他 . . . 37

15 完整 C++ 實作 **38**
15.1 編譯與執行 . . . 38
15.2 原始碼 . . . 38
15.3 執行輸出（節錄） . . . 51

16 練習題 **53**
16.1 入門（走訪與連通） . . . 53
16.2 進階（排序與最短路） . . . 53
16.3 挑戰（流、匹配、進階） . . . 53

17 總結與延伸閱讀 **54**
17.1 複雜度速查表 . . . 54
17.2 學習路徑建議 . . . 54
17.3 延伸閱讀 . . . 54

前言：為什麼要學圖論演算法

「圖」（Graph）是電腦科學中最具表達力的資料結構之一。任何牽涉到「物件」與「物件之間關係」的問題，幾乎都能用圖來建模：社群網路的朋友關係、地圖上的道路、網頁之間的超連結、程式的相依性、晶片上的線路、金融交易的金流、分子的原子鍵結……。一旦把問題抽象成圖，我們就能套用一整套成熟、強大且效率經過嚴格證明的演算法來求解。

本教材的目標，是帶你從零開始、系統性地掌握圖論演算法。我們會回答四個核心問題：

1. **如何表示圖？**——鄰接矩陣、鄰接串列、邊串列各自的取捨。
2. **如何走訪與分析圖？**——BFS、DFS 與其衍生的連通性、拓撲排序、環偵測、橋與割點。
3. **如何解決經典圖問題？**——最短路徑（Dijkstra、Bellman-Ford、Floyd-Warshall、A*）、最小生成樹（Kruskal、Prim）、網路流與二分匹配、最近共同祖先。
4. **真實世界如何應用？**——導航、社群推薦、詐欺偵測、排程、編譯器、網路路由，以及現代的圖神經網路（GNN）與圖資料庫。

本教材的特色

每個演算法都包含：直觀說明、嚴謹定義、虛擬碼或數學式、手算範例搭配 TikZ 圖解、複雜度分析、常見陷阱，以及對應的真實應用。第 14 章提供一份**完整、已實際編譯執行通過**的 C++17 程式碼（涵蓋全部 17 個演算法），可直接複製使用。

閱讀建議：若你是初學者，請依章節順序閱讀，先打穩第 1–3 章（表示法與走訪）的基礎；若你準備面試或競賽，可直接跳到對應主題，並搭配第 13 章「解題策略：如何辨識題型」與第 15 章的練習題。

Chapter 1

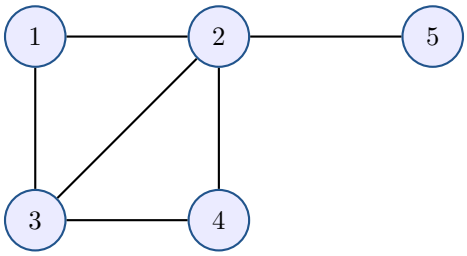
圖論基礎與術語

理解圖論演算法的前提，是先把術語講清楚。本章建立全書共用的語彙與數學記號。

1.1 什麼是圖

定義 1.1 (圖)**.** 一個**圖**（graph）是一個有序對 $G = (V, E)$，其中 V 是**頂點**（vertex / node）的有限集合，$E \subseteq V \times V$ 是**邊**（edge）的集合。我們通常以 $n = |V|$ 表示頂點數，$m = |E|$ 表示邊數。

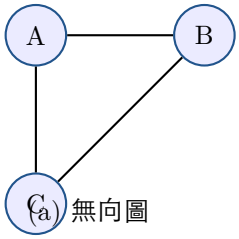
頂點代表「物件」，邊代表「物件之間的關係」。例如在社群網路中，頂點是使用者，邊是「好友」關係；在地圖中，頂點是路口，邊是道路。



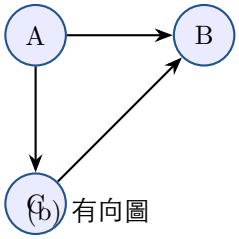
上圖是一個 $n = 5$、$m = 6$ 的無向圖。

1.2 有向圖與無向圖

- **無向圖**（undirected graph）：邊沒有方向，(u, v) 與 (v, u) 是同一條邊。例如 Facebook 的好友關係（互相的）。
- **有向圖**（directed graph / digraph）：邊有方向，(u, v) 表示「從 u 指向 v」。例如 Twitter 的追蹤關係、網頁超連結、工作相依性。



(a) 無向圖



(b) 有向圖

1.3 加權圖與無權圖

若每條邊都附帶一個數值 $w(u, v)$，稱為**加權圖**（weighted graph）。權重可表示距離、成本、時間、容量或機率。沒有權重（或視為權重皆為 1）的圖稱為**無權圖**（unweighted graph）。

常見陷阱

權重可以是**負數**（例如代表退款、放熱反應）。一旦出現負權邊，Dijkstra 演算法就會失效，必須改用 Bellman-Ford。請見第 7 章。

1.4 重要術語速覽

術語	意義
鄰接 (adjacent)	若 $(u,v)\in E$，稱 u 與 v 相鄰。
度數 (degree)	與頂點相連的邊數。有向圖再分**入度** (in-degree) 與**出度** (out-degree)。
路徑 (path)	一連串相鄰頂點 $v_0, v_1, \dots, v_k$。**簡單路徑**不重複經過頂點。
環 (cycle)	起點與終點相同的路徑（長度 ≥ 1，有向；≥ 3，無向簡單環）。
連通 (connected)	無向圖中任兩頂點間都存在路徑。
強連通 (strongly connected)	有向圖中任兩頂點 u, v 互相可達。
子圖 (subgraph)	由 $V'\subseteq V$、$E'\subseteq E$ 構成的圖。
完全圖 (complete graph)	任兩頂點都有邊，記為 K_n，邊數 $\binom{n}{2}$。
二分圖 (bipartite)	頂點可分為兩集合，邊只跨集合連接（無同側邊）。
樹 (tree)	連通且無環的無向圖，恰有 $n-1$ 條邊。
DAG	有向無環圖（Directed Acyclic Graph），可做拓撲排序。
稠密／稀疏	稠密圖 $m\approx n^2$；稀疏圖 $m\approx n$。決定資料結構選擇。

性質 1.1 (握手定理)**.** 無向圖中所有頂點的度數總和等於邊數的兩倍：

$$\sum_{v\in V}\deg(v) = 2|E|.$$

因為每條邊恰好替它的兩個端點各貢獻 1 度。由此可推得：度數為奇數的頂點必為偶數個。

1.5 圖的稀疏性與演算法選擇

「稀疏」還是「稠密」會直接影響你該用哪種表示法與演算法。一個直覺的判準：

$$\text{稠密} \iff m=\Theta(n^2), \qquad \text{稀疏} \iff m=O(n).$$

大多數真實世界的圖（社群網路、道路網、網頁圖）都是**稀疏**的——每個節點的鄰居數遠小於 n。這也是為什麼鄰接串列（第 2 章）幾乎是預設選擇。

Chapter 2

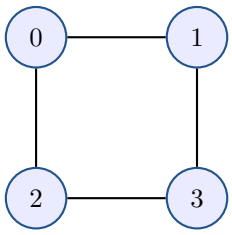
圖的表示法

「如何在記憶體中表示一張圖」是所有圖演算法的起點。選錯表示法，輕則浪費記憶體，重則讓演算法慢上好幾個數量級。本章比較三種主流表示法。

2.1 鄰接矩陣 (Adjacency Matrix)

用一個 $n \times n$ 的二維陣列 A 表示，其中

$$A[u][v] = \begin{cases} 1 & \text{若} (u,v) \in E \text{(無權圖)} \\ w(u,v) & \text{若加權圖} \\ 0 \text{ 或} \infty & \text{否則} \end{cases}$$



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

優點：查詢「u 與 v 是否相鄰」為 $O(1)$；實作簡單；適合稠密圖與需要頻繁查邊的演算法（如 Floyd-Warshall）。

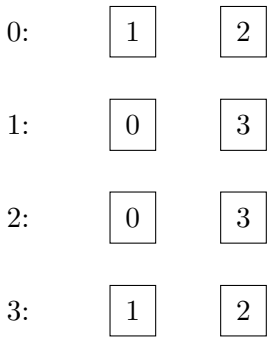
缺點：空間固定為 $O(n^2)$，對稀疏圖極度浪費；列舉某頂點的所有鄰居需 $O(n)$。

```cpp
int n;                                      // 頂點數
vector<vector<int>> A(n, vector<int>(n, 0));
A[u][v] = 1;                                // 加一條無向邊
A[v][u] = 1;
```

程式碼 2.1: 鄰接矩陣的建立

2.2 鄰接串列 (Adjacency List)

每個頂點維護一個「鄰居清單」。這是**最常用**的表示法，因為真實圖多為稀疏。



優點：空間為 $O(n+m)$；列舉鄰居只花 $O(\deg(v))$；適合稀疏圖與絕大多數走訪／最短路演算法。
缺點：查詢「u, v 是否相鄰」需 $O(\deg(u))$（非 $O(1)$）。

```cpp
int n;
vector<vector<pair<int,int>>> adj(n);  // adj[u] = {(鄰居, 權重), ...}
adj[u].push_back({v, w});              // 有向邊 u -> v
adj[v].push_back({u, w});              // 無向圖再加反向
```

程式碼 2.2: 鄰接串列（加權）

2.3 邊串列 (Edge List)

直接存所有邊 $\{(u, v, w)\}$。本身不利於查鄰居，但對某些演算法（Kruskal 需要把邊依權重排序、Bellman-Ford 逐邊鬆弛）反而最自然。

```cpp
struct Edge { int u, v, w; };
vector<Edge> edges;
edges.push_back({u, v, w});
```

程式碼 2.3: 邊串列

2.4 三種表示法的比較

操作 / 性質	鄰接矩陣	鄰接串列	邊串列
空間複雜度	$O(n^2)$	$O(n+m)$	$O(m)$
查 (u, v) 是否相鄰	$O(1)$	$O(\deg u)$	$O(m)$
列舉 u 的鄰居	$O(n)$	$O(\deg u)$	$O(m)$
加一條邊	$O(1)$	$O(1)$	$O(1)$
適合圖型	稠密	稀疏（最常用）	視演算法而定
代表演算法	Floyd-Warshall	BFS, DFS, Dijkstra, Prim	Kruskal, Bellman-Ford

實務技巧

預設用**鄰接串列**。只有在 (1) 圖很稠密、(2) 演算法需要 $O(1)$ 查邊、或 (3) 要做矩陣運算（如以矩陣冪計算路徑數）時，才考慮鄰接矩陣。

2.5 隱式圖 (Implicit Graph)

很多題目不會明確給你一張圖，而是「狀態空間」本身就是圖：

- **格子地圖**：每個格子是頂點，上下左右相鄰格是鄰居（牆壁除外）。
- **字串轉換**（Word Ladder）：每個單字是頂點，差一個字母的單字相連。
- **棋盤狀態**：每個盤面是頂點，合法走步連到下一盤面。

這類問題不需建出整張圖，只要在走訪時**即時生成**鄰居即可。這是 BFS/DFS 最常見的實戰形式。

Chapter 3

圖的走訪：BFS 與 DFS

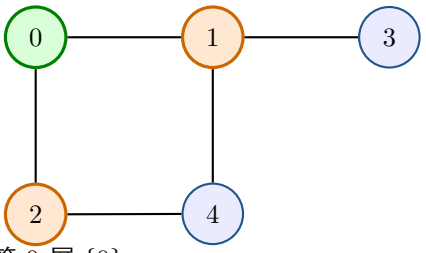
走訪（traversal）是「有系統地拜訪圖中每個頂點」的過程，是幾乎所有圖演算法的骨架。兩大基石是廣度優先搜尋（BFS）與深度優先搜尋（DFS）。

3.1 廣度優先搜尋 (BFS)

BFS 從起點出發，**一層一層**向外擴張：先拜訪所有距離為 1 的頂點，再拜訪距離為 2 的，依此類推。它用一個**佇列**（queue, FIFO）來維護「待拜訪」的頂點。

3.1.1 演算法流程

1. 將起點 s 放入佇列，標記 $dist[s] = 0$，其餘為未拜訪。
2. 當佇列非空：取出隊首 u，對每個未拜訪鄰居 v，設 $dist[v] = dist[u] + 1$，並入列。



由 0 出發：第 0 層 {0}，
第 1 層 {1,2}，第 2 層 {3,4}

3.1.2 核心性質

定理 3.1 (BFS 最短路)**.** 在**無權圖**中，BFS 求出的 $dist[v]$ 就是從 s 到 v 的最短路徑（邊數最少）長度。

直覺證明：BFS 按距離遞增的順序拜訪頂點，因此第一次抵達某頂點時，必定走的是最短路。

```cpp
vector<int> bfs(const vector<vector<int>>& adj, int s) {
    vector<int> dist(adj.size(), -1);
    queue<int> q;
    dist[s] = 0; q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u])
            if (dist[v] == -1) {          // 尚未拜訪
                dist[v] = dist[u] + 1;
                q.push(v);
            }
    }
```

```
    return dist;
}
```

程式碼 3.1: BFS 單源最短距離

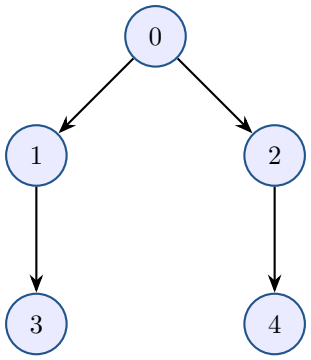
複雜度：每個頂點入列／出列一次、每條邊檢查一次，故時間 $O(V+E)$，空間 $O(V)$。

3.1.3 BFS 的延伸應用

- **多源 BFS**：把所有起點一次放入佇列（如「腐爛的橘子」、火勢蔓延）。
- **0-1 BFS**：邊權只有 0 或 1 時，用雙端佇列取代優先佇列，達 $O(V+E)$（見第 7 章）。
- **二分圖判定**：BFS 交錯著色，若發現相鄰同色則非二分圖（見第 4 章）。

3.2 深度優先搜尋 (DFS)

DFS 採取「**一路走到底，再回頭**」的策略：從某頂點出發，盡量往深處探索，直到無路可走才回溯。它天然以**遞迴**（或顯式堆疊）實作。



走訪順序（前序）：
$0 \to 1 \to 3 \to 2 \to 4$

```
void dfs(const vector<vector<int>>& adj, int u, vector<bool>& vis) {
    vis[u] = true;
    // ... 前序處理 u ...
    for (int v : adj[u])
        if (!vis[v]) dfs(adj, v, vis);
    // ... 後序處理 u ...
}
```

程式碼 3.2: DFS 遞迴版

常見陷阱

遞迴 DFS 在頂點數達數十萬時可能**堆疊溢位**（stack overflow）。深圖請改用顯式 `stack` 的迭代版本，或調大遞迴堆疊上限。

3.2.1 DFS 的時間戳與邊分類

DFS 過程中為每個頂點記錄「進入時間」$tin[u]$ 與「離開時間」$tout[u]$，可把邊分成四類（有向圖）：

- **樹邊**（tree edge）：DFS 樹中實際走的邊。
- **回邊**（back edge）：指向祖先——**回邊的存在等價於有環**。

- **前向邊**（forward edge）：指向已完成的子孫。
- **橫向邊**（cross edge）：指向其他子樹。

這套「時間戳 + 邊分類」是後續環偵測、拓撲排序、強連通分量、橋與割點的共同基礎。

3.3 BFS 與 DFS 的比較

	BFS	**DFS**
資料結構	佇列 (FIFO)	堆疊 / 遞迴 (LIFO)
探索順序	由近到遠，逐層	一路到底再回溯
無權最短路	**可以**（保證最短）	不保證
記憶體	可能存整層，最差 $O(V)$	與遞迴深度相關 $O(V)$
典型用途	最短路、層序、最近目標	連通性、環、拓撲、SCC、回溯
時間複雜度	$O(V+E)$	$O(V+E)$

實務技巧

口訣：「找**最短**用 BFS，找**結構**（環、連通、相依）用 DFS」。

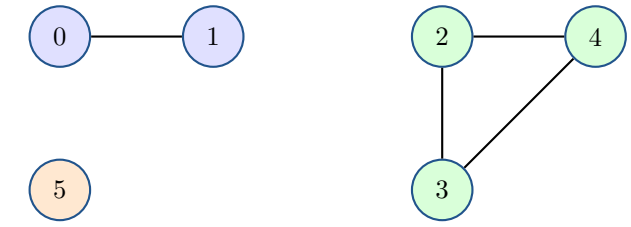
Chapter 4

連通性分析

「圖的哪些部分彼此相連？」是最基本的結構問題。本章涵蓋無向圖的連通分量、有向圖的強連通分量，以及二分圖判定。

4.1 連通分量 (Connected Components)

在無向圖中，**連通分量**是一個極大的頂點子集，集合內任兩點都互相可達。找出所有連通分量只需反覆從未拜訪的頂點啟動一次 DFS 或 BFS，並為它們標上同一個編號。



三個連通分量：{0,1}、{2,3,4}、{5}

```cpp
vector<int> components(const vector<vector<int>>& adj) {
    int n = adj.size(), cid = 0;
    vector<int> comp(n, -1);
    for (int s = 0; s < n; ++s) {
        if (comp[s] != -1) continue;
        stack<int> st; st.push(s); comp[s] = cid;
        while (!st.empty()) {
            int u = st.top(); st.pop();
            for (int v : adj[u])
                if (comp[v] == -1) { comp[v] = cid; st.push(v); }
        }
        ++cid;
    }
    return comp;
}
```

程式碼 4.1: 連通分量標記（迭代 DFS）

時間 $O(V+E)$。連通分量的計數在「朋友圈數量」、「島嶼數量」、「網路是否分裂」等問題中無所不在。

Union-Find 也能做

連通分量也可用第 9 章的並查集（DSU）增量維護——特別適合「邊會動態加入」的情境，例如逐步建線判斷網路何時連通。

4.2 二分圖判定 (Bipartite Check)

定義 4.1 (二分圖)**.** 若頂點集可分割成兩個互斥集合 L, R，使得每條邊都連接一個 L 點與一個 R 點（沒有同側邊），則 G 為**二分圖**。

判定方法：用 BFS/DFS 做**二著色**。給起點塗色 0，鄰居塗相反色 1，再相反……若途中發現某條邊兩端同色，則不是二分圖。

定理 4.1. 一個圖是二分圖 $\iff$ 它不含**奇數長度的環**。

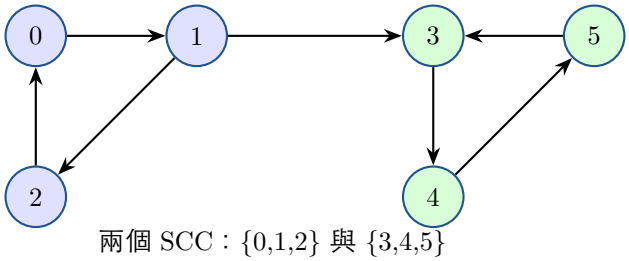
```cpp
bool isBipartite(const vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> color(n, -1);
    for (int s = 0; s < n; ++s) {
        if (color[s] != -1) continue;
        queue<int> q; q.push(s); color[s] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int v : adj[u]) {
                if (color[v] == -1) { color[v] = color[u] ^ 1; q.push(v);
                    }
                else if (color[v] == color[u]) return false; // 同色衝突
            }
        }
    }
    return true;
}
```

程式碼 4.2: 二分圖判定（BFS 二著色）

應用：排班（人 vs. 班次）、配對（求職者 vs. 職缺）、衝突偵測（兩群互斥），都先確認是否為二分圖，再套用第 11 章的二分匹配。

4.3 強連通分量 (SCC, Tarjan 演算法)

在**有向圖**中，「連通」要分得更細。**強連通分量**（Strongly Connected Component）是極大的頂點子集，集內任兩點 u, v 都**雙向可達**（$u \to v$ 且 $v \to u$）。



兩個 SCC：{0,1,2} 與 {3,4,5}

Tarjan 演算法用一次 DFS 求出所有 SCC。核心是兩個時間戳：

- $disc[u]$：u 的「發現時間」。
- $low[u]$：u 經由 DFS 子樹 + 至多一條回邊，能回到的最早 $disc$ 值。

搭配一個堆疊存放「目前尚未歸屬」的頂點。當 $disc[u] = low[u]$ 時，u 是一個 SCC 的「根」，將堆疊彈出到 u 為止即構成一個 SCC。

定理 4.2. SCC 縮點後（把每個 SCC 收縮成一個超級節點）所得的圖必為 **DAG**（稱為凝聚圖 / condensation）。這讓「先分層再處理」的策略成為可能。

複雜度：$O(V+E)$。完整實作見第 14 章 `TarjanSCC` 類別。

應用：2-SAT 判定、編譯器的相互遞迴函式分組、社群網路的緊密群偵測、死結（deadlock）偵測。

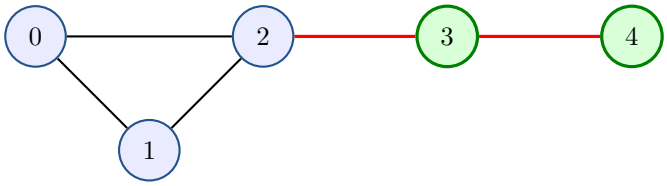
4.4 橋與割點 (Bridges & Articulation Points)

在無向圖中，有些邊或點是「網路的咽喉」：

- **割點**（articulation point / cut vertex）：移除後會使連通分量增加的頂點。
- **橋**（bridge / cut edge）：移除後會使連通分量增加的邊。

它們衡量網路的**脆弱性**——一座橋斷了，網路就分裂。同樣用 $disc/low$ 時間戳，一次 DFS 即可求出：

- 邊 (u,v) 是橋 $\iff low[v] > disc[u]$（v 的子樹無法繞過 u 回到更早的點）。
- 非根頂點 u 是割點 $\iff$ 存在子節點 v 使 $low[v] \geq disc[u]$；根頂點是割點 $\iff$ 它有 ≥ 2 個 DFS 子樹。



紅色為橋；綠色 3 為割點

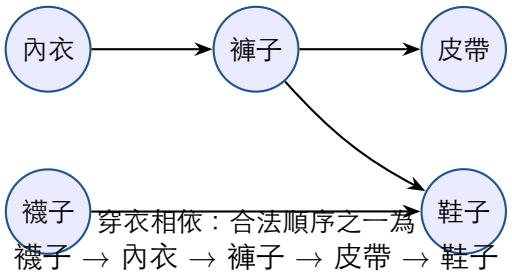
應用：電信骨幹網路的單點故障分析、道路網的關鍵橋樑、社群網路的「橋接者」（連接兩個社群的關鍵人物）。完整實作見第 14 章 `ArticulationBridge`。

Chapter 5

拓撲排序 (Topological Sort)

5.1 問題定義

給定一個**有向無環圖**（DAG），**拓撲排序**是把所有頂點排成一個線性序列，使得對每條邊 (u, v)，u 都排在 v 之前。直覺上就是「相依關係的合法執行順序」。



常見陷阱

拓撲排序**只對 DAG 存在**。若圖中有環，就不存在合法順序——這也是「偵測相依是否成循環」的方法。

5.2 Kahn 演算法（BFS 式）

核心觀念：**入度為 0** 的頂點沒有任何前置相依，可以最先處理。

1. 計算每個頂點的入度，把入度 0 者全部入列。
2. 反覆取出一個頂點 u 加入結果，並把它所有出邊「移除」（鄰居入度減 1）；若鄰居入度降為 0 則入列。
3. 若最終排序的頂點數 $< n$，代表有環。

```cpp
vector<int> topoSort(const vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> indeg(n, 0), order;
    for (int u = 0; u < n; ++u)
        for (int v : adj[u]) ++indeg[v];
    queue<int> q;
    for (int i = 0; i < n; ++i) if (indeg[i] == 0) q.push(i);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        order.push_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0) q.push(v);
    }
```

```
14      return (int)order.size() == n ? order : vector<int>{}; // 空 = 有環
15  }
```

程式碼 5.1: Kahn 拓撲排序

複雜度：$O(V+E)$。

5.3 DFS 式拓撲排序

另一種做法：對每個頂點做 DFS，在**離開**（後序）時把頂點推入堆疊，最後將堆疊反轉即為拓撲序。直覺是：一個頂點所有的後繼都處理完，它才被加入，故它必排在後繼之前。

實務技巧

若題目要求**字典序最小**的拓撲排序，把 Kahn 的普通佇列換成**最小堆**（priority_queue）即可。

5.4 應用

- **建置系統 / 編譯相依**：Makefile、Maven、npm 決定編譯順序。
- **課程先修**（LeetCode「課程表」）：判斷能否修完所有課。
- **試算表重算**：儲存格公式相依的計算次序。
- **任務排程**：專案管理的工序安排（搭配關鍵路徑法）。

Chapter 6

環偵測 (Cycle Detection)

「圖中有沒有環？」依有向／無向而有不同做法。環偵測是相依分析、死結偵測、資料一致性檢查的核心。

6.1 有向圖：DFS 三色法

為每個頂點維護三種狀態：

- **白色**（0，未拜訪）
- **灰色**（1，在當前 DFS 路徑上 / 遞迴堆疊中）
- **黑色**（2，已完成）

DFS 過程中，若遇到一條指向**灰色**頂點的邊（即**回邊**），就代表存在環。

```cpp
bool dfsCycle(const vector<vector<int>>& adj, int u, vector<int>& color) {
    color[u] = 1;                          // 灰：進入遞迴
    for (int v : adj[u]) {
        if (color[v] == 1) return true;   // 回邊 -> 有環
        if (color[v] == 0 && dfsCycle(adj, v, color)) return true;
    }
    color[u] = 2;                          // 黑：離開
    return false;
}
bool hasCycle(const vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> color(n, 0);
    for (int i = 0; i < n; ++i)
        if (color[i] == 0 && dfsCycle(adj, i, color)) return true;
    return false;
}
```

程式碼 6.1: 有向圖環偵測（三色 DFS）

常見陷阱

不能只用「是否拜訪過」的二元旗標！指向**黑色**（已完成）頂點的邊是前向／橫向邊，**不構成環**。必須區分灰與黑。

6.2 無向圖：DFS 父節點法或並查集

無向圖中，DFS 時若遇到一個**已拜訪且非父節點**的鄰居，即代表有環：

```cpp
bool dfsU(const vector<vector<int>>& adj, int u, int parent,
    vector<bool>& vis) {
    vis[u] = true;
    for (int v : adj[u]) {
        if (!vis[v]) { if (dfsU(adj, v, u, vis)) return true; }
        else if (v != parent) return true;   // 連到非父的已訪點 -> 環
    }
    return false;
}
```

程式碼 6.2: 無向圖環偵測（記錄父節點）

另一種優雅做法是**並查集**：逐邊處理 (u, v)，若 u, v 已在同一集合則成環，否則合併。這正是 Kruskal 避免成環的機制（第 9 章）。

6.3 偵測負權環

在加權有向圖中，**負權環**會讓最短路徑無下界（繞圈無限變小）。Bellman-Ford 在第 V 次仍能鬆弛任何邊，即代表存在負權環（第 7 章）。

6.4 應用對照

情境	圖型	方法
死結偵測（資源等待圖）	有向	三色 DFS
試算表循環參照	有向	三色 DFS / 拓撲失敗
電路短路 / 冗餘連線	無向	並查集 / 父節點 DFS
套利機會（匯率）	加權有向	Bellman-Ford 找負權環

Chapter 7

最短路徑演算法

最短路徑是圖論最重要、應用最廣的主題：導航軟體、網路路由、遊戲尋路、物流規劃都依賴它。本章依「圖的特性」介紹四種主力演算法，並給出選擇指南。

7.1 問題分類

圖的特性	適用演算法	時間複雜度
無權圖	BFS	$O(V+E)$
邊權僅 0/1	0-1 BFS（雙端佇列）	$O(V+E)$
非負權，單源	Dijkstra（二元堆）	$O((V+E)\log V)$
含負權，單源	Bellman-Ford	$O(VE)$
全源最短路	Floyd-Warshall	$O(V^3)$
單點對 + 啟發	A*	視啟發函數

7.2 Dijkstra 演算法

7.2.1 核心思想

Dijkstra 解決**非負權**圖的單源最短路。它是**貪婪**演算法：維護一個「已確定最短距離」的集合，每次從「尚未確定」的頂點中挑出**目前距離最小**者，確定它，並用它去**鬆弛**（relax）鄰居。

定義 7.1 (鬆弛 Relaxation)**.** 對邊 (u,v) 權重 w，若 $dist[u]+w < dist[v]$，則更新 $dist[v] \leftarrow dist[u]+w$。這是所有最短路演算法的共同基本操作。

定理 7.1 (正確性的關鍵)**.** 當所有邊權**非負**時，一旦某頂點被取出（其暫定距離在堆中最小），這個距離就是最終最短距離，不會再被更小的路徑更新。負權邊會破壞此假設，故 Dijkstra 不適用。

7.2.2 以優先佇列實作

用一個**最小堆**（min-heap）維護候選頂點，每次彈出距離最小者。

```
vector<long long> dijkstra(const vector<vector<pair<int,int>>>& adj, int
    s) {
    const long long INF = LLONG_MAX / 4;
    vector<long long> dist(adj.size(), INF);
    priority_queue<pair<long long,int>,
                   vector<pair<long long,int>>,
                   greater<>> pq;          // (距離, 頂點) 小根堆
    dist[s] = 0; pq.push({0, s});
    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
```

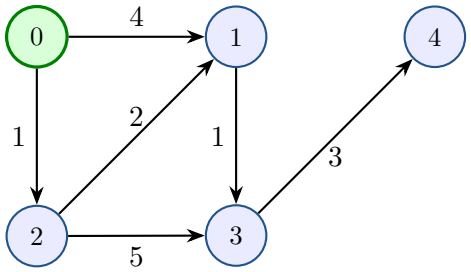
```
        if (d > dist[u]) continue;          // 過期項目，跳過
        for (auto [v, w] : adj[u])
            if (d + w < dist[v]) {
                dist[v] = d + w;
                pq.push({dist[v], v});
            }
    }
    return dist;
}
```

程式碼 7.1: Dijkstra（min-heap）

「過期項目」技巧

我們不刪除堆中的舊項目，而是在彈出時用 `if (d > dist[u]) continue;` 略過。這比實作「decrease-key」簡單得多，且不影響漸進複雜度。

7.2.3 手算範例



由 0 出發，依序確定：$dist[0]{=}0 \to dist[2]{=}1 \to dist[1]{=}3$（經 0→2→1，比直接 0→1 的 4 更短）$\to dist[3]{=}4 \to dist[4]{=}7$。最終 $dist = [0, 3, 1, 4, 7]$。

7.2.4 重建路徑

只要在鬆弛時記錄前驅 $parent[v] = u$，最後從終點沿 $parent$ 回溯即可得到實際路徑。

7.3 0-1 BFS

當邊權**只有 0 或 1** 時，可用**雙端佇列**（deque）取代堆，達到 $O(V + E)$：權重 0 的邊把鄰居推到隊**首**（同層），權重 1 的推到隊**尾**（下一層）。

```
vector<int> zeroOneBFS(const vector<vector<pair<int,int>>>& adj, int s) {
    vector<int> dist(adj.size(), INT_MAX);
    deque<int> dq;
    dist[s] = 0; dq.push_front(s);
    while (!dq.empty()) {
        int u = dq.front(); dq.pop_front();
        for (auto [v, w] : adj[u])
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (w == 0) dq.push_front(v);
                else        dq.push_back(v);
            }
    }
    return dist;
```

```
}
```

程式碼 7.2: 0-1 BFS

應用：格子地圖中「有些移動免費、有些要花費 1」的最短代價問題。

7.4 Bellman-Ford 演算法

7.4.1 核心思想

Bellman-Ford 能處理**負權邊**，並能**偵測負權環**。它不貪婪，而是「暴力鬆弛」：對所有邊重複鬆弛 $V-1$ 次。

定理 7.2. 在沒有負權環的圖中，任何最短路徑最多含 $V-1$ 條邊。因此鬆弛所有邊 $V-1$ 輪後，所有 $dist$ 必收斂到最短值。若第 V 輪仍能鬆弛，代表存在**負權環**。

```cpp
struct Result { vector<long long> dist; bool negCycle; };
Result bellmanFord(int n, const vector<array<int,3>>& edges, int s) {
    const long long INF = LLONG_MAX / 4;
    vector<long long> dist(n, INF); dist[s] = 0;
    for (int i = 0; i < n - 1; ++i) {
        bool relaxed = false;
        for (auto& e : edges) {              // e = {u, v, w}
            if (dist[e[0]] == INF) continue;
            if (dist[e[0]] + e[2] < dist[e[1]]) {
                dist[e[1]] = dist[e[0]] + e[2];
                relaxed = true;
            }
        }
        if (!relaxed) break;                 // 提早收斂
    }
    bool neg = false;
    for (auto& e : edges)
        if (dist[e[0]] != INF && dist[e[0]] + e[2] < dist[e[1]]) neg =
            true;
    return {dist, neg};
}
```

程式碼 7.3: Bellman-Ford（含負權環偵測）

複雜度：$O(VE)$。雖比 Dijkstra 慢，但能處理負權。SPFA 是其佇列優化版，平均較快但最差仍 $O(VE)$。

匯率套利

把貨幣當頂點、匯率取對數負值當邊權，則「負權環」對應到一連串兌換後**錢變多**的套利機會——Bellman-Ford 正好能偵測。

7.5 Floyd-Warshall 演算法

求**所有頂點對**之間的最短路。核心是極簡的動態規劃：考慮「只允許經過前 k 個頂點作中繼」的最短路，逐步放寬 k。

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right).$$

```cpp
// d[i][j] 初始化為邊權，無邊為 INF，d[i][i]=0
for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (d[i][k] + d[k][j] < d[i][j])
                d[i][j] = d[i][k] + d[k][j];
```

程式碼 7.4: Floyd-Warshall

常見陷阱

三層迴圈中，k **必須在最外層**！順序錯了會得到錯誤答案。另外鬆弛前要檢查 `d[i][k]` 與 `d[k][j]` 非 INF，避免溢位。

複雜度：時間 $O(V^3)$、空間 $O(V^2)$。適合頂點數較小（$V \lesssim 400$）但需要全源資訊的情境，也能用來求**傳遞閉包**（哪些點互相可達）。

7.6 A* 搜尋

A* 是 Dijkstra 的「有方向性」版本，常用於遊戲與地圖尋路。它用一個**啟發函數** $h(v)$ 估計 v 到終點的距離，以 $f(v) = g(v) + h(v)$ 排序（g 為已知起點距離）。

- 若 h 為**可採納**（admissible，從不高估），A* 保證找到最短路。
- 常見啟發：格子地圖用曼哈頓距離或歐氏距離。
- $h \equiv 0$ 時，A* 退化為 Dijkstra。

A* 透過優先探索「看起來更接近終點」的方向，大幅減少展開的節點數，但答案品質取決於啟發函數的設計。

7.7 最短路演算法選擇指南

實務技巧

決策樹：無權 → BFS；權重 0/1 → 0-1 BFS；非負權單源 → Dijkstra；有負權 → Bellman-Ford；要全源且 V 小 → Floyd-Warshall；單點對且有好啟發 → A*。

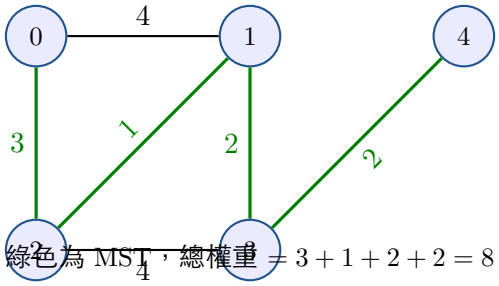
Chapter 8

最小生成樹 (Minimum Spanning Tree)

8.1 問題定義

定義 8.1 (生成樹與最小生成樹)**.** 給定一個**連通**、**無向**、**加權**圖 G，一棵**生成樹**（spanning tree）是包含全部 n 個頂點、且恰有 $n-1$ 條邊的無環子圖。權重總和最小的生成樹稱為**最小生成樹**（MST）。

直覺：用最少的「成本」把所有點連起來。例如鋪設電纜連通所有城市、讓總線長最短。



性質 8.1 (兩個關鍵性質)**. 割性質（Cut Property）**：對任一將頂點分成兩半的「割」，橫跨割的**最小權邊**必屬於某棵 MST。這保證了貪婪選擇的正確性。
環性質（Cycle Property）：任一環中**權重最大**的邊必**不**屬於 MST。

8.2 Kruskal 演算法

策略：把所有邊**依權重由小到大排序**，依序嘗試加入；若一條邊的兩端點**尚未連通**（用並查集判斷）就加入，否則會成環，跳過。加滿 $n-1$ 條邊即完成。

```
struct Edge { int u, v, w; };
long long kruskal(int n, vector<Edge> edges) {
    sort(edges.begin(), edges.end(),
         [](const Edge& a, const Edge& b){ return a.w < b.w; });
    DSU dsu(n);
    long long total = 0; int used = 0;
    for (auto& e : edges) {
        if (dsu.unite(e.u, e.v)) {          // 不成環才加入
            total += e.w;
            if (++used == n - 1) break;
        }
    }
    return total;
}
```

程式碼 8.1: Kruskal（需搭配第 9 章 DSU）

複雜度：排序主導，$O(E \log E)$。Kruskal 用**邊串列**最自然，特別適合**稀疏圖**。

8.3 Prim 演算法

策略：類似 Dijkstra，從任一頂點開始「長樹」。維護一個優先佇列，每次挑出「連接樹與樹外、權重最小」的邊，把對應的新頂點納入樹中。

```cpp
long long primMST(const vector<vector<pair<int,int>>>& adj) {
    int n = adj.size();
    vector<bool> inTree(n, false);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    long long total = 0; int taken = 0;
    pq.push({0, 0});                          // (權重, 頂點)
    while (!pq.empty() && taken < n) {
        auto [w, u] = pq.top(); pq.pop();
        if (inTree[u]) continue;
        inTree[u] = true; total += w; ++taken;
        for (auto [v, wt] : adj[u])
            if (!inTree[v]) pq.push({wt, v});
    }
    return total;
}
```

程式碼 8.2: Prim（min-heap）

複雜度：$O((V+E)\log V)$。Prim 用**鄰接串列**，較適合**稠密圖**（搭配鄰接矩陣的 $O(V^2)$ 版本在稠密時甚至更快）。

8.4 Kruskal vs. Prim

	Kruskal	**Prim**
策略	全域挑最小邊（排序）	從一點長樹（局部擴張）
資料結構	並查集 + 邊串列	優先佇列 + 鄰接串列
複雜度	$O(E\log E)$	$O((V+E)\log V)$
最適圖型	稀疏圖	稠密圖
易於處理「動態加邊」	是	否

8.5 應用

- **網路 / 電纜 / 管線佈建**：以最低總成本連通所有節點。
- **群聚分析**：刪除 MST 中最長的 $k-1$ 條邊，可得到 k 個群（單連結聚類）。
- **近似演算法**：MST 是旅行推銷員問題（TSP）2-近似解的基礎。
- **影像分割**：以像素相似度為權重，用 MST 切割影像區域。

Chapter 9

並查集 (Disjoint Set Union)

9.1 問題與動機

並查集（Disjoint Set Union, DSU / Union-Find）維護一組「不相交集合」，支援兩個操作：

- `find(x)`：回傳 x 所屬集合的代表元（root）。
- `unite(x, y)`：合併 x 與 y 所在的兩個集合。

它是 Kruskal、連通性維護、環偵測的幕後功臣，也是動態連通問題的標準解。

9.2 兩大優化：路徑壓縮 + 按秩合併

樸素實作的樹可能退化成鏈，使 `find` 變 $O(n)$。兩個優化讓它幾乎變成常數：

- **路徑壓縮**（path compression）：`find` 時把沿途節點直接掛到根下。
- **按秩 / 按大小合併**（union by rank/size）：把較矮的樹掛到較高的樹下，避免長高。

定理 9.1 (近乎常數)**.** 同時採用兩種優化後，m 次操作的總時間為 $O(m\,\alpha(n))$，其中 α 是反阿克曼函數，在所有實際輸入下 $\alpha(n) \le 4$。可視為「攤還常數時間」。

```cpp
class DSU {
    vector<int> parent, rank_;
public:
    DSU(int n) : parent(n), rank_(n, 0) {
        iota(parent.begin(), parent.end(), 0);  // 各自成集
    }
    int find(int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];       // 路徑壓縮（折半）
            x = parent[x];
        }
        return x;
    }
    bool unite(int x, int y) {
        int rx = find(x), ry = find(y);
        if (rx == ry) return false;              // 已在同集（成環）
        if (rank_[rx] < rank_[ry]) swap(rx, ry);
        parent[ry] = rx;
        if (rank_[rx] == rank_[ry]) ++rank_[rx];
        return true;
    }
    bool same(int x, int y) { return find(x) == find(y); }
};
```

程式碼 9.1: 並查集（路徑壓縮 + 按秩合併）

9.3 應用

- **Kruskal MST**：判斷加邊是否成環。
- **動態連通性**：逐步加邊，隨時查詢兩點是否連通。
- **連通分量計數**：合併同時維護分量數。
- **離線查詢**：如「島嶼數量 II」逐格加陸地。
- **帶權並查集**：可維護到根的相對關係（如「食物鏈」種類推理）。

實務技巧

DSU 擅長「**加邊**」與「查連通」，但**不支援高效刪邊**。若題目會刪邊，常見手法是「時間倒流」——離線把操作反轉成加邊。

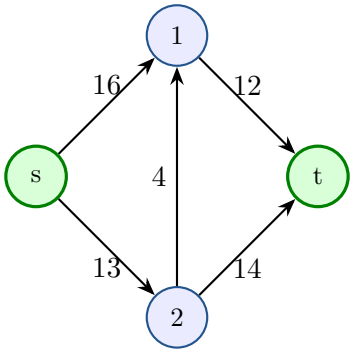
Chapter 10

網路流 (Network Flow)

網路流是圖論中極具威力的建模工具：許多看似無關的問題（匹配、分配、切割、排程）都能化約成「最大流」來求解。

10.1 最大流問題

定義 10.1 (流網路)**.** 一個**流網路**是有向圖 $G = (V, E)$，每條邊 (u, v) 有非負**容量** $c(u, v)$，並指定**源點** s 與**匯點** t。一個**流** f 需滿足：(1) 容量限制 $0 \le f(u, v) \le c(u, v)$；(2) 流量守恆：除 s, t 外，每個頂點流入 $=$ 流出。**最大流**即從 s 送到 t 的最大總流量。



10.2 Ford-Fulkerson 與 Edmonds-Karp

核心觀念：反覆在**殘餘網路** (residual graph) 中尋找一條從 s 到 t 的**增廣路徑** (augmenting path)，沿途增加流量，直到找不到為止。

- **殘餘容量**：邊 (u, v) 還能再送的量 $= c(u, v) - f(u, v)$。每條正向邊都伴隨一條**反向邊**，容量初始為 0，用來「反悔」先前的流。
- **Ford-Fulkerson**：用 DFS 找增廣路，整數容量下會終止，但複雜度依賴流值。
- **Edmonds-Karp**：固定用 **BFS** 找**最短**增廣路，複雜度 $O(VE^2)$，與容量大小無關。

```cpp
struct MaxFlow {
    struct E { int to, rev, cap; };
    vector<vector<E>> g;
    MaxFlow(int n) : g(n) {}
    void addEdge(int u, int v, int cap) {
        g[u].push_back({v, (int)g[v].size(), cap});
        g[v].push_back({u, (int)g[u].size()-1, 0}); // 反向邊
    }
    int run(int s, int t) { /* BFS 找增廣路並更新殘餘圖，見第 14 章 */ }
};
```

程式碼 10.1: Edmonds-Karp 最大流（核心）

10.3 最大流最小割定理

定理 10.1 (Max-Flow Min-Cut)**.** 在任一流網路中，從 s 到 t 的**最大流值**，恰等於分離 s 與 t 的**最小割**（移除後使 s 無法到 t 的邊容量總和最小者）的容量。

這個對偶關係極為深刻：求「最多能送多少」等價於求「最便宜的切斷方式」。它把流問題與割問題（影像分割、可靠度分析）連結起來。

10.4 建模的藝術：問題化約

網路流真正的威力在於**把其他問題翻譯成流**：

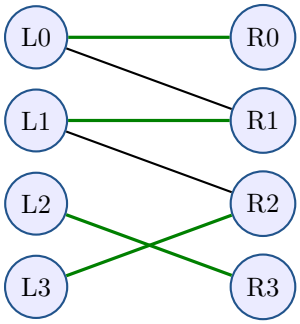
- **二分匹配**：源點連左部、右部連匯點，所有容量設 1，最大流即最大匹配（見下章）。
- **帶下界 / 多源多匯**：加超級源匯轉換。
- **最小成本最大流**：邊再附成本，用 SPFA/Bellman-Ford 找最便宜增廣路——用於指派、運輸問題。
- **專案選擇 / 最大權閉合子圖**：化為最小割。

Chapter 11

二分匹配 (Bipartite Matching)

11.1 問題定義

給定二分圖（左部 L、右部 R），**匹配**是一組沒有共用端點的邊。**最大匹配**即邊數最多的匹配。典型情境：把求職者配到職缺、學生配到專題、計程車配到乘客。



綠色為一組完美匹配（4 條）

11.2 匈牙利演算法（增廣路）

核心是**增廣路徑**：一條交替走「未匹配邊、已匹配邊」、且兩端都是未匹配點的路徑。沿此路徑翻轉所有邊的匹配狀態，匹配數就 $+1$。對每個左部點嘗試找增廣路即可。

```cpp
bool tryMatch(int u, vector<vector<int>>& adj,
              vector<int>& matchR, vector<bool>& used) {
    for (int v : adj[u]) {
        if (used[v]) continue;
        used[v] = true;
        if (matchR[v] == -1 || tryMatch(matchR[v], adj, matchR, used)) {
            matchR[v] = u;                  // 配對 / 重新配對
            return true;
        }
    }
    return false;
}
```

程式碼 11.1: 二分匹配（增廣路 DFS）

複雜度：$O(V \cdot E)$。資料量大時可用 Hopcroft-Karp 達 $O(E\sqrt{V})$。

定理 11.1 (König 定理)**.** 在二分圖中，**最大匹配數 = 最小頂點覆蓋數**。又由 König 可推得最大獨立集 $= n-$ 最大匹配。這些對偶關係讓匹配能解決覆蓋與獨立集問題。

11.3 應用

- **指派問題**：員工　任務、課程　教室、機器　工件。
- **排班**：醫師　值班時段。
- **推薦 / 廣告**：在限制下把廣告位配給廣告主（常用最小成本流的加權版）。

Chapter 12

進階主題

本章彙整幾個常在競賽與實務出現的進階結構，著重觀念與應用，完整實作見第 14 章。

12.1 最近共同祖先 (LCA)

在有根樹中，兩節點 u, v 的**最近共同祖先**是同時為兩者祖先、且深度最大的節點。它能高效回答「樹上兩點距離」、「路徑查詢」等問題。

倍增法（Binary Lifting）：預處理 $up[k][v]$ =「v 的第 2^k 個祖先」。查詢時先把較深的點上提到同深度，再一起往上跳。

- 預處理 $O(V \log V)$，每次查詢 $O(\log V)$。
- 樹上兩點距離 $= depth[u] + depth[v] - 2\,depth[\mathrm{lca}(u, v)]$。

應用：階層分類、family tree、網路拓撲的共同上游、版本控制的共同祖先 commit。

12.2 歐拉路徑與歐拉迴路

歐拉路徑是「不重複地走過每一條邊恰好一次」的路徑；若起終點相同則為**歐拉迴路**。這正是著名的「七橋問題」。

- 無向連通圖存在歐拉迴路 $\iff$ 所有頂點度數皆**偶數**。
- 存在歐拉路徑（非迴路） $\iff$ 恰有**兩個奇數度**頂點。
- Hierholzer 演算法可在 $O(E)$ 內構造。

應用：DNA 片段重組、郵差路線、一筆畫問題。

12.3 最小生成樹的延伸

- **次小生成樹**：在 MST 基礎上換一條邊。
- **有向最小生成樹（樹形圖）**：朱劉演算法（Chu-Liu/Edmonds）。
- **Steiner 樹**：只需連通指定子集（NP-hard，常用近似）。

12.4 圖著色與最大團

圖著色：給頂點塗色使相鄰異色，最少用幾種色（色數）是 NP-hard。應用於排程、暫存器配置、頻率分配。**最大團**（互相皆相鄰的最大頂點集）同為 NP-hard，但在實務上常用啟發式或位元優化暴搜。

12.5 現代延伸：圖神經網路 (GNN)

近年圖機器學習快速崛起。**圖神經網路**（GNN）透過**訊息傳遞**（message passing）學習節點 / 邊 / 整圖的向量表示：每個節點反覆「聚合鄰居資訊 → 更新自身狀態」，多輪後即捕捉到局部結構。

- 代表架構：GCN（圖卷積）、GraphSAGE（取樣聚合）、GAT（注意力）。
- 任務型態：節點分類、連結預測、整圖分類。
- 應用：分子性質預測、推薦系統、詐欺偵測、交通預測、知識圖譜補全。

傳統演算法 vs. GNN

傳統圖演算法靠**明確規則**求精確解（如最短路）；GNN 則從**資料學習**模式，擅長有雜訊、需預測的場景。實務上兩者常**互補**：先用圖演算法萃取特徵（中心性、社群、距離），再餵給機器學習模型。

Chapter 13

解題策略：如何辨識題型

面對一道新題，最關鍵的是**把問題翻譯成圖**並**辨識題型**。本章提供一套系統化的思考流程。

13.1 第一步：辨識圖的結構

問自己四個問題：

1. **頂點與邊是什麼？**物件是頂點，關係是邊。格子？狀態？單字？
2. **有向還是無向？**
3. **有權還是無權？權重可為負嗎？**
4. **圖是明確給定，還是隱式狀態空間？**

13.2 第二步：題型 → 演算法對照表

當你看到……	多半要用……
無權圖最短路 / 最少步數	BFS
邊權 0/1 的最短代價	0-1 BFS
非負權最短路	Dijkstra
含負權 / 偵測套利	Bellman-Ford
所有點對距離（V 小）	Floyd-Warshall
連通分量 / 島嶼數 / 朋友圈	DFS/BFS 或並查集
相依排序 / 課程表 / 編譯順序	拓撲排序
偵測（有向）環 / 死結	三色 DFS
最小成本連通所有點	MST（Kruskal / Prim）
動態加邊查連通	並查集
指派 / 配對 / 排班	二分匹配
最大吞吐 / 最小切斷	最大流 / 最小割
樹上兩點距離 / 共同祖先	LCA（倍增）
互斥兩群判定	二分圖著色

13.3 第三步：常見陷阱檢查表

常見陷阱

- 對**負權**誤用 Dijkstra（會給錯但不報錯）。
- 遞迴 DFS 在大圖**堆疊溢位**——改迭代。
- Floyd-Warshall 的 k 沒放**最外層**。
- 環偵測只用二元 visited，沒區分**灰/黑**。
- 無向圖環偵測忘了排除**父節點**。
- 最大流忘了加**反向邊**。
- 距離用 `int` 而**溢位**——大圖改 `long long`，且 INF 取 `LLONG_MAX/4` 避免相加溢位。

Chapter 14

真實世界的應用

圖演算法不是學術玩具——它們是現代科技基礎建設的核心。本章蒐集 2026 年產業實例。

14.1 導航與地圖

Google Maps、Waze 的路線規劃本質是**加權最短路**。實務上不直接跑 Dijkstra（太慢），而是用**雙向搜尋**、**A* + 地標啟發**、以及**收縮層級**（Contraction Hierarchies）等預處理技術，把全國路網的查詢壓到毫秒級。即時路況則把邊權動態更新。

14.2 社群網路

- **好友推薦**：「你可能認識的人」用 BFS 找二度人脈、共同好友數。
- **影響力 / 排名**：PageRank（特徵向量中心性的變體）衡量節點重要性。
- **社群偵測**：用連通分量、Louvain、標籤傳播找出緊密群體。
- **Netflix** 以自研「Graph Abstraction」服務社交圖與服務拓撲，尖峰處理近**每秒千萬次**操作、橫跨 650 TB 圖資料。

14.3 詐欺偵測與金融

詐欺本質是**連結問題**——詐騙集團以「環狀金流、共用裝置、共用身分」運作，傳統逐筆分析難以察覺。圖資料庫（如 Neo4j）用以下技巧揭露詐欺網：

- **連通分量 / 社群偵測**找出詐欺環。
- **PageRank / 中心性**找出異常具影響力的帳戶（人頭中樞）。
- **節點相似度（Jaccard）**偵測「共用 SSN / Email / 電話」的模仿帳戶。
- **多跳遍歷**追蹤洗錢的資金流向。

Neo4j 宣稱以圖比關聯式資料庫快上千倍揭露此類模式。

14.4 推薦系統

電商與串流以**二分圖**（使用者　商品）建模。協同過濾透過圖遍歷找「和你相似的人也喜歡的東西」。一個典型 Cypher 查詢的精神：找「我喜歡的電影 → 也喜歡它的其他人 → 他們喜歡但我還沒看的電影」，依出現次數排序推薦。

14.5 基礎設施與供應鏈

- **網路路由**：OSPF 用 Dijkstra 計算最短路徑樹；BGP 處理跨自治系統路由。
- **供應鏈**：以圖建模多階供應商相依，分析**斷點風險**（割點 / 橋）與替代路徑。Google BigQuery Graph 讓企業在 PB 級資料上做多跳相依分析。
- **電網 / 電信**：割點分析找單點故障；最大流評估容量。

14.6 編譯器、建置與排程

- **相依解析**：套件管理（npm、Maven）、建置系統用**拓撲排序**決定順序、用環偵測抓循環相依。
- **暫存器配置**：以**圖著色**決定變數該放哪個暫存器。
- **指令排程 / 關鍵路徑**：DAG 上的最長路徑決定專案最短工期。

14.7 生物資訊與其他

DNA 定序的片段重組用**歐拉路徑**（de Bruijn 圖）；蛋白質交互作用網路、代謝路徑分析、藥物分子的性質預測（GNN）都以圖為核心。此外，知識圖譜（Knowledge Graph）支撐了搜尋引擎與大型語言模型的 GraphRAG 檢索。

Chapter 15

完整 C++ 實作

本章提供一份**完整、可直接編譯執行**的 C++17 程式碼，涵蓋全書 17 個核心演算法。此程式碼已實際編譯（`g++ -std=c++17 -O2`）並執行驗證通過。

包含的演算法

走訪與結構：BFS、DFS、拓撲排序（Kahn）、有向環偵測、連通分量、強連通分量（Tarjan）、割點與橋。
最短路：Dijkstra、0-1 BFS、Bellman-Ford、Floyd-Warshall。
生成樹與集合：Kruskal、Prim、並查集（DSU）。
流與匹配與樹：Edmonds-Karp 最大流、二分匹配、LCA（倍增）。

15.1 編譯與執行

```
g++ -std=c++17 -O2 -Wall -Wextra -o graph_algorithms graph_algorithms.cpp
./graph_algorithms
```

程式碼 15.1: 編譯指令

15.2 原始碼

```
// graph_algorithms.cpp
// Compile: g++ -std=c++17 -O2 -Wall -Wextra -o graph_algorithms
    graph_algorithms.cpp
// Run:     ./graph_algorithms

#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <deque>
#include <algorithm>
#include <numeric>
#include <climits>
#include <functional>
#include <iomanip>
#include <utility>

using namespace std;

// ============================================================
// 1. Graph class — adjacency list with weighted edges
// ============================================================
class Graph {
```

```cpp
public:
    int n;
    vector<vector<pair<int, int>>> adj;  // adj[u] = {(neighbor, weight), ...}
    bool directed;

    Graph(int n_, bool directed_ = false) : n(n_), adj(n_), directed(directed_) {}

    void addEdge(int u, int v, int w = 1) {
        adj[u].push_back({v, w});
        if (!directed) adj[v].push_back({u, w});
    }

    int size() const { return n; }
};

// ============================================================
// 2. Breadth-First Search — single-source distance in unweighted graph
// ============================================================
vector<int> bfs(const Graph& g, int s) {
    vector<int> dist(g.n, -1);
    queue<int> q;
    q.push(s);
    dist[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (const auto& e : g.adj[u]) {
            int v = e.first;
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
    return dist;
}

// ============================================================
// 3. Depth-First Search — iterative + recursive variants
// ============================================================
void dfsRec(const Graph& g, int u, vector<bool>& visited, vector<int>& order) {
    visited[u] = true;
    order.push_back(u);
    for (const auto& e : g.adj[u])
        if (!visited[e.first]) dfsRec(g, e.first, visited, order);
}

vector<int> dfs(const Graph& g, int s) {
    vector<bool> visited(g.n, false);
    vector<int> order;
    dfsRec(g, s, visited, order);
    return order;
}

// ============================================================
// 4. Topological Sort — Kahn's algorithm (BFS-based)
// ============================================================
```

```cpp
vector<int> topoSort(const Graph& g) {
    vector<int> inDeg(g.n, 0);
    for (int u = 0; u < g.n; ++u)
        for (const auto& e : g.adj[u]) ++inDeg[e.first];
    queue<int> q;
    for (int i = 0; i < g.n; ++i) if (inDeg[i] == 0) q.push(i);
    vector<int> order;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        order.push_back(u);
        for (const auto& e : g.adj[u])
            if (--inDeg[e.first] == 0) q.push(e.first);
    }
    return (int)order.size() == g.n ? order : vector<int>();  // empty -> cycle
}

// ============================================================
// 5. Cycle Detection (directed graph) — DFS three colors
//    color[v]: 0 = unvisited, 1 = on stack, 2 = finished
// ============================================================
bool dfsCycle(const Graph& g, int u, vector<int>& color) {
    color[u] = 1;
    for (const auto& e : g.adj[u]) {
        int v = e.first;
        if (color[v] == 1) return true;
        if (color[v] == 0 && dfsCycle(g, v, color)) return true;
    }
    color[u] = 2;
    return false;
}

bool hasCycleDirected(const Graph& g) {
    vector<int> color(g.n, 0);
    for (int i = 0; i < g.n; ++i)
        if (color[i] == 0 && dfsCycle(g, i, color)) return true;
    return false;
}

// ============================================================
// 6. Connected Components (undirected) — DFS labelling
// ============================================================
vector<int> connectedComponents(const Graph& g) {
    vector<int> comp(g.n, -1);
    int cid = 0;
    for (int s = 0; s < g.n; ++s) {
        if (comp[s] != -1) continue;
        stack<int> st;
        st.push(s);
        comp[s] = cid;
        while (!st.empty()) {
            int u = st.top(); st.pop();
            for (const auto& e : g.adj[u])
                if (comp[e.first] == -1) {
                    comp[e.first] = cid;
                    st.push(e.first);
                }
        }
        ++cid;
```

```cpp
    }
    return comp;
}

// ============================================================
// 7. Strongly Connected Components — Tarjan's algorithm
// ============================================================
class TarjanSCC {
    const Graph& g;
    int timer = 0, sccCount = 0;
    vector<int> disc, low, sccId;
    vector<bool> onStack;
    stack<int> stk;

    void dfs(int u) {
        disc[u] = low[u] = ++timer;
        stk.push(u);
        onStack[u] = true;
        for (const auto& e : g.adj[u]) {
            int v = e.first;
            if (!disc[v]) { dfs(v); low[u] = min(low[u], low[v]); }
            else if (onStack[v]) low[u] = min(low[u], disc[v]);
        }
        if (disc[u] == low[u]) {
            while (true) {
                int v = stk.top(); stk.pop();
                onStack[v] = false;
                sccId[v] = sccCount;
                if (v == u) break;
            }
            ++sccCount;
        }
    }
public:
    TarjanSCC(const Graph& g_)
        : g(g_), disc(g_.n, 0), low(g_.n, 0), sccId(g_.n, -1),
          onStack(g_.n, false) {
        for (int i = 0; i < g.n; ++i) if (!disc[i]) dfs(i);
    }
    int count() const { return sccCount; }
    const vector<int>& component() const { return sccId; }
};

// ============================================================
// 8. Articulation Points & Bridges — Tarjan
// ============================================================
class ArticulationBridge {
    const Graph& g;
    int timer = 0;
    vector<int> disc, low;
public:
    vector<bool> isArticulation;
    vector<pair<int, int>> bridges;

    ArticulationBridge(const Graph& g_)
        : g(g_), disc(g_.n, 0), low(g_.n, 0), isArticulation(g_.n, false)
          {
        for (int i = 0; i < g.n; ++i) if (!disc[i]) dfs(i, -1);
    }
```

```cpp
private:
    void dfs(int u, int parent) {
        disc[u] = low[u] = ++timer;
        int children = 0;
        for (const auto& e : g.adj[u]) {
            int v = e.first;
            if (!disc[v]) {
                ++children;
                dfs(v, u);
                low[u] = min(low[u], low[v]);
                if (low[v] > disc[u]) bridges.push_back({u, v});
                if (parent != -1 && low[v] >= disc[u]) isArticulation[u]
                    = true;
            } else if (v != parent) {
                low[u] = min(low[u], disc[v]);
            }
        }
        if (parent == -1 && children > 1) isArticulation[u] = true;
    }
};

// ============================================================
// 9. Disjoint Set Union (Union-Find) with path compression + rank
// ============================================================
class DSU {
    vector<int> parent, rnk;
public:
    DSU(int n) : parent(n), rnk(n, 0) {
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }
    bool unite(int x, int y) {
        int rx = find(x), ry = find(y);
        if (rx == ry) return false;
        if (rnk[rx] < rnk[ry]) swap(rx, ry);
        parent[ry] = rx;
        if (rnk[rx] == rnk[ry]) ++rnk[rx];
        return true;
    }
    bool same(int x, int y) { return find(x) == find(y); }
};

// ============================================================
// 10. Dijkstra — Single-source shortest path (non-negative weights)
// ============================================================
vector<long long> dijkstra(const Graph& g, int s) {
    const long long INF = LLONG_MAX / 4;
    vector<long long> dist(g.n, INF);
    using P = pair<long long, int>;
    priority_queue<P, vector<P>, greater<P>> pq;
    dist[s] = 0;
    pq.push({0, s});
    while (!pq.empty()) {
```

```cpp
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;
        for (const auto& e : g.adj[u]) {
            long long nd = d + e.second;
            if (nd < dist[e.first]) {
                dist[e.first] = nd;
                pq.push({nd, e.first});
            }
        }
    }
    return dist;
}

// ============================================================
// 11. 0-1 BFS — shortest path when weights are only 0 or 1
// ============================================================
vector<int> zeroOneBFS(const Graph& g, int s) {
    vector<int> dist(g.n, INT_MAX);
    deque<int> dq;
    dist[s] = 0;
    dq.push_front(s);
    while (!dq.empty()) {
        int u = dq.front(); dq.pop_front();
        for (const auto& e : g.adj[u]) {
            int nd = dist[u] + e.second;
            if (nd < dist[e.first]) {
                dist[e.first] = nd;
                if (e.second == 0) dq.push_front(e.first);
                else dq.push_back(e.first);
            }
        }
    }
    return dist;
}

// ============================================================
// 12. Bellman-Ford — handles negative weights, detects negative cycle
// ============================================================
struct BellmanFordResult {
    vector<long long> dist;
    bool hasNegativeCycle;
};

BellmanFordResult bellmanFord(const Graph& g, int s) {
    const long long INF = LLONG_MAX / 4;
    vector<long long> dist(g.n, INF);
    dist[s] = 0;
    for (int i = 0; i < g.n - 1; ++i) {
        bool relaxed = false;
        for (int u = 0; u < g.n; ++u) {
            if (dist[u] == INF) continue;
            for (const auto& e : g.adj[u]) {
                if (dist[u] + e.second < dist[e.first]) {
                    dist[e.first] = dist[u] + e.second;
                    relaxed = true;
                }
            }
        }
        if (!relaxed) break;
```

```
    }
    bool neg = false;
    for (int u = 0; u < g.n && !neg; ++u) {
        if (dist[u] == INF) continue;
        for (const auto& e : g.adj[u])
            if (dist[u] + e.second < dist[e.first]) { neg = true; break; }
    }
    return {dist, neg};
}

// ============================================================
// 13. Floyd-Warshall — all-pairs shortest paths O(V^3)
// ============================================================
vector<vector<long long>> floydWarshall(const Graph& g) {
    const long long INF = LLONG_MAX / 4;
    vector<vector<long long>> d(g.n, vector<long long>(g.n, INF));
    for (int i = 0; i < g.n; ++i) d[i][i] = 0;
    for (int u = 0; u < g.n; ++u)
        for (const auto& e : g.adj[u])
            d[u][e.first] = min(d[u][e.first], (long long)e.second);
    for (int k = 0; k < g.n; ++k)
        for (int i = 0; i < g.n; ++i)
            for (int j = 0; j < g.n; ++j)
                if (d[i][k] < INF && d[k][j] < INF
                    && d[i][k] + d[k][j] < d[i][j])
                    d[i][j] = d[i][k] + d[k][j];
    return d;
}

// ============================================================
// 14. Kruskal — Minimum spanning tree via DSU
// ============================================================
struct EdgeWUV { int u, v, w; };

pair<long long, vector<EdgeWUV>> kruskal(int n, vector<EdgeWUV> edges) {
    sort(edges.begin(), edges.end(),
         [](const EdgeWUV& a, const EdgeWUV& b) { return a.w < b.w; });
    DSU dsu(n);
    long long total = 0;
    vector<EdgeWUV> mst;
    for (const auto& e : edges) {
        if (dsu.unite(e.u, e.v)) {
            total += e.w;
            mst.push_back(e);
        }
    }
    return {total, mst};
}

// ============================================================
// 15. Prim — Minimum spanning tree via priority queue
// ============================================================
long long primMST(const Graph& g) {
    long long total = 0;
    vector<bool> inTree(g.n, false);
    using P = pair<int, int>;  // (weight, node)
    priority_queue<P, vector<P>, greater<P>> pq;
    pq.push({0, 0});
    int taken = 0;
```

```cpp
    while (!pq.empty() && taken < g.n) {
        auto [w, u] = pq.top(); pq.pop();
        if (inTree[u]) continue;
        inTree[u] = true;
        total += w;
        ++taken;
        for (const auto& e : g.adj[u])
            if (!inTree[e.first]) pq.push({e.second, e.first});
    }
    return total;
}

// ============================================================
// 16. Edmonds-Karp — max flow via BFS-augmenting paths
// ============================================================
class MaxFlow {
public:
    struct Edge { int to, rev, cap; };
    int n;
    vector<vector<Edge>> g;

    MaxFlow(int n_) : n(n_), g(n_) {}

    void addEdge(int u, int v, int cap) {
        g[u].push_back({v, (int)g[v].size(), cap});
        g[v].push_back({u, (int)g[u].size() - 1, 0});
    }

    int bfsAug(int s, int t, vector<int>& parent, vector<int>&
        parentEdge) {
        parent.assign(n, -1);
        parentEdge.assign(n, -1);
        parent[s] = s;
        queue<pair<int, int>> q;
        q.push({s, INT_MAX});
        while (!q.empty()) {
            auto [u, flow] = q.front(); q.pop();
            for (int i = 0; i < (int)g[u].size(); ++i) {
                const auto& e = g[u][i];
                if (parent[e.to] == -1 && e.cap > 0) {
                    parent[e.to]     = u;
                    parentEdge[e.to] = i;
                    int newFlow = min(flow, e.cap);
                    if (e.to == t) return newFlow;
                    q.push({e.to, newFlow});
                }
            }
        }
        return 0;
    }

    int run(int s, int t) {
        int totalFlow = 0;
        vector<int> parent, parentEdge;
        int pushed;
        while ((pushed = bfsAug(s, t, parent, parentEdge)) > 0) {
            totalFlow += pushed;
            int cur = t;
            while (cur != s) {
```

```
                int prev = parent[cur];
                int ei = parentEdge[cur];
                g[prev][ei].cap -= pushed;
                g[cur][g[prev][ei].rev].cap += pushed;
                cur = prev;
            }
        }
        return totalFlow;
    }
};

// ============================================================
// 17. Bipartite Matching — Hungarian-style augmenting paths
// ============================================================
class BipartiteMatching {
    int nL, nR;
    vector<vector<int>> adj;  // adj[L-side u] = list of R-side neighbors
    vector<int> matchR;       // matchR[R] = L matched to R
    vector<bool> usedL;
    bool tryMatch(int u) {
        for (int v : adj[u]) {
            if (usedL[v]) continue;
            usedL[v] = true;
            if (matchR[v] == -1 || tryMatch(matchR[v])) {
                matchR[v] = u;
                return true;
            }
        }
        return false;
    }
public:
    BipartiteMatching(int nL_, int nR_) : nL(nL_), nR(nR_), adj(nL_),
       matchR(nR_, -1) {}
    void addEdge(int u, int v) { adj[u].push_back(v); }
    int run() {
        int total = 0;
        for (int u = 0; u < nL; ++u) {
            usedL.assign(nR, false);
            if (tryMatch(u)) ++total;
        }
        return total;
    }
    const vector<int>& matchingR() const { return matchR; }
};

// ============================================================
// 18. LCA — Lowest Common Ancestor via binary lifting
// ============================================================
class LCA {
    int n, LOG;
    vector<vector<int>> up;   // up[k][v] = 2^k-th ancestor of v
    vector<int> depth;
public:
    LCA(int n_, const vector<vector<int>>& tree, int root)
        : n(n_), LOG(1), depth(n_, 0) {
        while ((1 << LOG) < n) ++LOG;
        up.assign(LOG + 1, vector<int>(n, root));
        function<void(int, int)> dfs = [&](int u, int par) {
            up[0][u] = par;
```

```cpp
            for (int v : tree[u]) {
                if (v == par) continue;
                depth[v] = depth[u] + 1;
                dfs(v, u);
            }
        };
        dfs(root, root);
        for (int k = 1; k <= LOG; ++k)
            for (int v = 0; v < n; ++v)
                up[k][v] = up[k - 1][up[k - 1][v]];
    }
    int query(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);
        int diff = depth[u] - depth[v];
        for (int k = 0; k <= LOG; ++k)
            if ((diff >> k) & 1) u = up[k][u];
        if (u == v) return u;
        for (int k = LOG; k >= 0; --k)
            if (up[k][u] != up[k][v]) {
                u = up[k][u];
                v = up[k][v];
            }
        return up[0][u];
    }
    int dist(int u, int v) { return depth[u] + depth[v] - 2 *
        depth[query(u, v)]; }
};

// ============================================================
// Demonstrations
// ============================================================
static void hr() { cout << string(60, '-') << "\n"; }

int main() {
    cout << fixed << setprecision(2);

    // ---- BFS demo ----
    hr(); cout << "1. BFS shortest distance (unweighted)\n";
    {
        Graph g(6);
        g.addEdge(0, 1); g.addEdge(0, 2);
        g.addEdge(1, 3); g.addEdge(2, 3);
        g.addEdge(3, 4); g.addEdge(4, 5);
        auto d = bfs(g, 0);
        cout << "  dist from 0 = ";
        for (int x : d) cout << x << " ";
        cout << "\n";
    }

    // ---- DFS demo ----
    hr(); cout << "2. DFS pre-order from 0\n";
    {
        Graph g(6);
        g.addEdge(0, 1); g.addEdge(0, 2);
        g.addEdge(1, 3); g.addEdge(2, 4); g.addEdge(4, 5);
        auto o = dfs(g, 0);
        cout << "  order: ";
        for (int x : o) cout << x << " ";
        cout << "\n";
```

```
545     }
546
547     // ---- Topological sort ----
548     hr(); cout << "3. Topological sort (Kahn): 0->1, 0->2, 1->3, 2->3,
            3->4, 2->5\n";
549     {
550         Graph g(6, true);
551         g.addEdge(0, 1); g.addEdge(0, 2);
552         g.addEdge(1, 3); g.addEdge(2, 3);
553         g.addEdge(3, 4); g.addEdge(2, 5);
554         auto t = topoSort(g);
555         cout << "  order: ";
556         for (int x : t) cout << x << " ";
557         cout << "\n";
558     }
559
560     // ---- Cycle detection ----
561     hr(); cout << "4. Cycle detection (directed)\n";
562     {
563         Graph g(4, true);
564         g.addEdge(0, 1); g.addEdge(1, 2); g.addEdge(2, 0); g.addEdge(2,
                3);
565         cout << "  has cycle = " << boolalpha << hasCycleDirected(g) <<
                "\n";
566     }
567     {
568         Graph g(4, true);
569         g.addEdge(0, 1); g.addEdge(1, 2); g.addEdge(2, 3);
570         cout << "  has cycle (DAG) = " << boolalpha <<
                hasCycleDirected(g) << "\n";
571     }
572
573     // ---- SCC (Tarjan) ----
574     hr(); cout << "5. SCC (Tarjan): 0->1->2->0, 1->3, 3->4, 4->5, 5->3\n";
575     {
576         Graph g(6, true);
577         g.addEdge(0, 1); g.addEdge(1, 2); g.addEdge(2, 0);
578         g.addEdge(1, 3);
579         g.addEdge(3, 4); g.addEdge(4, 5); g.addEdge(5, 3);
580         TarjanSCC scc(g);
581         cout << "  components count = " << scc.count() << "\n";
582         cout << "  comp[v] = ";
583         for (int x : scc.component()) cout << x << " ";
584         cout << "\n";
585     }
586
587     // ---- Articulation & Bridges ----
588     hr(); cout << "6. Articulation points & bridges\n";
589     {
590         Graph g(7);
591         g.addEdge(0, 1); g.addEdge(1, 2); g.addEdge(2, 0);  // triangle
592         g.addEdge(2, 3); g.addEdge(3, 4); g.addEdge(4, 5);  // chain
593         g.addEdge(5, 6); g.addEdge(6, 4);                   // small cycle
594         ArticulationBridge ab(g);
595         cout << "  articulation = ";
596         for (int i = 0; i < g.n; ++i)
597             if (ab.isArticulation[i]) cout << i << " ";
598         cout << "\n  bridges = ";
599         for (auto& p : ab.bridges) cout << "(" << p.first << "," <<
```

```
            p.second << ") ";
600         cout << "\n";
601     }
602
603     // ---- Connected components ----
604     hr(); cout << "7. Connected components\n";
605     {
606         Graph g(6);
607         g.addEdge(0, 1); g.addEdge(2, 3); g.addEdge(3, 4);
608         auto cc = connectedComponents(g);
609         cout << "  comp = ";
610         for (int x : cc) cout << x << " ";
611         cout << "\n";
612     }
613
614     // ---- Dijkstra ----
615     hr(); cout << "8. Dijkstra (single-source shortest path)\n";
616     {
617         Graph g(5, true);
618         g.addEdge(0, 1, 4); g.addEdge(0, 2, 1);
619         g.addEdge(2, 1, 2); g.addEdge(1, 3, 1);
620         g.addEdge(2, 3, 5); g.addEdge(3, 4, 3);
621         auto d = dijkstra(g, 0);
622         cout << "  dist from 0 = ";
623         for (auto x : d) cout << x << " ";
624         cout << "\n";
625     }
626
627     // ---- 0-1 BFS ----
628     hr(); cout << "9. 0-1 BFS\n";
629     {
630         Graph g(5, true);
631         g.addEdge(0, 1, 0); g.addEdge(0, 2, 1);
632         g.addEdge(1, 2, 0); g.addEdge(2, 3, 1); g.addEdge(3, 4, 0);
633         auto d = zeroOneBFS(g, 0);
634         cout << "  dist from 0 = ";
635         for (auto x : d) cout << x << " ";
636         cout << "\n";
637     }
638
639     // ---- Bellman-Ford (negative edge demo) ----
640     hr(); cout << "10. Bellman-Ford (with negative edge)\n";
641     {
642         Graph g(5, true);
643         g.addEdge(0, 1, 6);  g.addEdge(0, 2, 7);
644         g.addEdge(1, 2, 8);  g.addEdge(1, 3, 5);  g.addEdge(1, 4, -4);
645         g.addEdge(2, 3, -3); g.addEdge(2, 4, 9);
646         g.addEdge(3, 1, -2);
647         g.addEdge(4, 0, 2);  g.addEdge(4, 3, 7);
648         auto r = bellmanFord(g, 0);
649         cout << "  dist from 0 = ";
650         for (auto x : r.dist) cout << x << " ";
651         cout << "\n  has negative cycle = " << boolalpha <<
                r.hasNegativeCycle << "\n";
652     }
653
654     // ---- Floyd-Warshall ----
655     hr(); cout << "11. Floyd-Warshall (all pairs)\n";
656     {
```

```
657         Graph g(4, true);
658         g.addEdge(0, 1, 5);  g.addEdge(0, 3, 10);
659         g.addEdge(1, 2, 3);  g.addEdge(2, 3, 1);
660         const long long INF = LLONG_MAX / 4;
661         auto d = floydWarshall(g);
662         for (int i = 0; i < g.n; ++i) {
663             cout << "  ";
664             for (int j = 0; j < g.n; ++j) {
665                 if (d[i][j] >= INF) cout << setw(5) << "INF";
666                 else                cout << setw(5) << d[i][j];
667             }
668             cout << "\n";
669         }
670     }
671
672     // ---- Kruskal MST ----
673     hr(); cout << "12. Kruskal MST\n";
674     {
675         vector<EdgeWUV> edges = {
676             {0, 1, 4}, {0, 2, 3}, {1, 2, 1}, {1, 3, 2},
677             {2, 3, 4}, {3, 4, 2}, {4, 5, 6}
678         };
679         auto [w, mst] = kruskal(6, edges);
680         cout << "  total weight = " << w << "\n  MST edges:";
681         for (auto& e : mst) cout << " (" << e.u << "," << e.v << "," <<
                e.w << ")";
682         cout << "\n";
683     }
684
685     // ---- Prim MST ----
686     hr(); cout << "13. Prim MST\n";
687     {
688         Graph g(6);
689         g.addEdge(0, 1, 4); g.addEdge(0, 2, 3);
690         g.addEdge(1, 2, 1); g.addEdge(1, 3, 2);
691         g.addEdge(2, 3, 4); g.addEdge(3, 4, 2);
692         g.addEdge(4, 5, 6);
693         cout << "  total weight = " << primMST(g) << "\n";
694     }
695
696     // ---- DSU ----
697     hr(); cout << "14. Disjoint Set Union\n";
698     {
699         DSU d(6);
700         d.unite(0, 1); d.unite(2, 3); d.unite(1, 2);
701         cout << "  same(0,3)? " << boolalpha << d.same(0, 3) << "\n";
702         cout << "  same(0,4)? " << boolalpha << d.same(0, 4) << "\n";
703     }
704
705     // ---- Edmonds-Karp Max Flow ----
706     hr(); cout << "15. Edmonds-Karp max flow (CLRS classic)\n";
707     {
708         MaxFlow mf(6);
709         // 0=s, 5=t
710         mf.addEdge(0, 1, 16); mf.addEdge(0, 2, 13);
711         mf.addEdge(1, 2, 10); mf.addEdge(2, 1, 4);
712         mf.addEdge(1, 3, 12); mf.addEdge(3, 2, 9);
713         mf.addEdge(2, 4, 14); mf.addEdge(4, 3, 7);
714         mf.addEdge(3, 5, 20); mf.addEdge(4, 5, 4);
```

```
        cout << "  max flow s->t = " << mf.run(0, 5) << "\n";
    }

    // ---- Bipartite Matching ----
    hr(); cout << "16. Bipartite matching (4 jobs, 4 workers)\n";
    {
        BipartiteMatching m(4, 4);
        m.addEdge(0, 0); m.addEdge(0, 1);
        m.addEdge(1, 1); m.addEdge(1, 2);
        m.addEdge(2, 0); m.addEdge(2, 3);
        m.addEdge(3, 2); m.addEdge(3, 3);
        cout << "  max matching = " << m.run() << "\n";
        cout << "  R-side match: ";
        for (int v : m.matchingR()) cout << v << " ";
        cout << "\n";
    }

    // ---- LCA ----
    hr(); cout << "17. LCA (binary lifting)\n";
    {
        // Tree:
        //        0
        //       / \
        //      1   2
        //     / \   \
        //    3   4   5
        //   /
        //  6
        vector<vector<int>> tree(7);
        auto addT = [&](int u, int v){ tree[u].push_back(v);
            tree[v].push_back(u); };
        addT(0, 1); addT(0, 2); addT(1, 3); addT(1, 4);
        addT(2, 5); addT(3, 6);
        LCA lca(7, tree, 0);
        cout << "  LCA(6,4) = " << lca.query(6, 4)
             << " dist = " << lca.dist(6, 4) << "\n";
        cout << "  LCA(6,5) = " << lca.query(6, 5)
             << " dist = " << lca.dist(6, 5) << "\n";
        cout << "  LCA(3,4) = " << lca.query(3, 4)
             << " dist = " << lca.dist(3, 4) << "\n";
    }

    return 0;
}
```

15.3 執行輸出（節錄）

```
1. BFS shortest distance (unweighted)
  dist from 0 = 0 1 1 2 3 4
3. Topological sort (Kahn)
  order: 0 1 2 3 5 4
5. SCC (Tarjan): components count = 2
  comp[v] = 1 1 1 0 0 0
6. Articulation points & bridges
  articulation = 2 3 4
  bridges = (3,4) (2,3)
8. Dijkstra: dist from 0 = 0 3 1 4 7
```

```
10. Bellman-Ford: dist from 0 = 0 2 7 4 -2  (no negative cycle)
12. Kruskal MST: total weight = 14
13. Prim MST:    total weight = 14
15. Edmonds-Karp max flow s->t = 23
16. Bipartite matching = 4
17. LCA(6,4) = 1 dist = 3 ; LCA(6,5) = 0 dist = 5
```

程式碼 15.2: 程式實際輸出

Chapter 16

練習題

以下練習依難度分級。建議先獨立思考「該用哪個演算法、如何建圖」，再動手實作。多數題目可在 LeetCode、Codeforces 找到對應題。

16.1 入門（走訪與連通）

1. **島嶼數量**：給 0/1 格子地圖，求相連陸地塊數。（DFS/BFS 或並查集）
2. **無權最短路**：求迷宮起點到終點最少步數。（BFS）
3. **二分圖判定**：判斷是否可二著色。（BFS 二著色）
4. **朋友圈**：給朋友關係矩陣求圈數。（並查集）
5. **克隆圖**：深拷貝一張無向圖。（DFS + 雜湊表）

16.2 進階（排序與最短路）

6. **課程表 II**：輸出可行修課順序，或判斷不可能。（拓撲排序）
7. **網路延遲時間**：信號傳到所有節點的最短時間。（Dijkstra）
8. **K 站中轉最便宜航班**：帶限制的最短路。（修改版 Dijkstra / Bellman-Ford）
9. **路徑最小體力消耗**：邊權為高度差，最小化路徑最大邊。（Dijkstra 變體 / 二分 + BFS）
10. **連接所有點的最小費用**：曼哈頓距離為權的 MST。（Prim / Kruskal）

16.3 挑戰（流、匹配、進階）

11. **最大二分匹配**：求職者與職缺配對。（匈牙利 / 最大流）
12. **逃生問題**：多源到邊界的最大不相交路徑。（最大流 / 最小割）
13. **關鍵連接（橋）**：找出網路中所有橋。（Tarjan）
14. **樹上路徑查詢**：多次詢問兩點距離。（LCA 倍增）
15. **重新規劃路線**：使所有點能到達 0 的最少改向邊數。（DFS / BFS 建雙向圖）
16. **套利偵測**：給匯率表判斷是否存在套利。（Bellman-Ford 找負權環）

實務技巧

做題時養成習慣：先在紙上畫小範例、手動模擬演算法，再寫程式。對拍（brute force vs. 正解）能有效抓出邊界錯誤。

Chapter 17

總結與延伸閱讀

17.1 複雜度速查表

演算法	時間	空間
BFS / DFS	$O(V+E)$	$O(V)$
拓撲排序（Kahn）	$O(V+E)$	$O(V)$
連通分量	$O(V+E)$	$O(V)$
Tarjan SCC	$O(V+E)$	$O(V)$
割點 / 橋	$O(V+E)$	$O(V)$
Dijkstra（堆）	$O((V+E)\log V)$	$O(V)$
0-1 BFS	$O(V+E)$	$O(V)$
Bellman-Ford	$O(VE)$	$O(V)$
Floyd-Warshall	$O(V^3)$	$O(V^2)$
Kruskal	$O(E\log E)$	$O(V)$
Prim（堆）	$O((V+E)\log V)$	$O(V)$
並查集	$O(\alpha(n))$ ／操作	$O(V)$
Edmonds-Karp	$O(VE^2)$	$O(V+E)$
二分匹配（匈牙利）	$O(VE)$	$O(V+E)$
LCA（倍增）	預處理 $O(V\log V)$，查詢 $O(\log V)$	$O(V\log V)$

17.2 學習路徑建議

1. **打地基**：徹底搞懂鄰接串列、BFS、DFS。八成的圖題都建立在這三者之上。
2. **連通性家族**：連通分量、二分圖、拓撲排序、環偵測——它們都是 DFS/BFS 的變體。
3. **最短路家族**：先 Dijkstra，再 Bellman-Ford 與 Floyd-Warshall，理解何時用哪個。
4. **生成樹與並查集**：Kruskal/Prim/DSU 一起學，互相呼應。
5. **進階**：網路流、匹配、LCA、Tarjan，依需求深入。
6. **現代延伸**：圖資料庫（Neo4j、Cypher）、圖機器學習（GNN）。

17.3 延伸閱讀

- *Introduction to Algorithms*（CLRS），第 22–26 章——圖演算法的權威教科書。
- *Competitive Programming*（Halim 兄弟）——競賽導向、實作豐富。
- *Algorithm Design*（Kleinberg & Tardos）——強調建模與證明。

- *Graph Algorithms*（O'Reilly, Needham & Hodler）——以 Neo4j / Spark 講實務圖分析。
- *cp-algorithms.com*——線上免費、涵蓋完整的競賽演算法參考。
- PyTorch Geometric 官方教學——入門圖神經網路（GNN）的最佳起點。

結語

圖論演算法的迷人之處，在於它把五花八門的真實問題統一到「頂點 + 邊」的優雅框架下。一旦你能熟練地「把問題看成圖」，許多原本棘手的難題都會迎刃而解。祝你學習愉快，並在真實專案中發揮這套強大的工具！