Báo cáo Project

Nguyễn Trung Hậu

Tháng 7, 2025

Mục lục

1 Project 4: Các Bài Toán Duyệt Đồ Thị & Cây

1.1 Duyệt Đồ thị

Mục tiêu của duyệt đồ thị là "đi thăm"tất cả các đỉnh của đồ thị theo một trật tự cụ thể để tìm kiếm thông tin, kiểm tra tính chất hoặc tìm đường đi.

1.1.1 Tìm kiếm theo chiều rộng (BFS - Breadth-First Search)

- **Ý tưởng:** Bắt đầu từ một đỉnh gốc, thuật toán "lan tỏa"ra để duyệt tất cả các đỉnh gần nhất, sau đó mới đến các đỉnh xa hơn.
- **Cơ chế:** Sử dụng một **hàng đợi (queue)**.
- **Ứng dụng chính:** Tìm đường đi ngắn nhất trong đồ thị không có trọng số.

1.1.2 Tìm kiếm theo chiều sâu (DFS - Depth-First Search)

- **Ý tưởng:** Đi sâu hết mức có thể theo một nhánh cho đến khi gặp ngõ cụt rồi mới quay lui (backtracking) để thử nhánh khác.
- **Cơ chế:** Sử dụng một **ngăn xếp (stack)** hoặc **đệ quy (recursion)**.
- **Ứng dụng chính:** Kiểm tra sự tồn tại của chu trình, sắp xếp Topo.

1.2 Cây

Cây là một dạng đồ thị liên thông, không chu trình, làm nền tảng cho nhiều bài toán tối ưu hóa mạng lưới.

1.2.1 Tìm Cây khung nhỏ nhất (Minimum Spanning Tree - MST)

- **Bài toán:** Cho một đồ thị liên thông có trọng số. Tìm một tập hợp các cạnh để nối tất cả các đỉnh lại với nhau sao cho tổng chi phí là thấp nhất và không tạo ra chu trình.
- **Thuật toán tiêu biểu:** Kruskal (chọn các cạnh rẻ nhất không tạo chu trình), Prim (nuôi cây lớn dần từ một đỉnh).

1.2.2 Tìm Đường đi ngắn nhất (Shortest Path)

- **Bài toán:** Cho một đồ thị có trọng số, tìm một đường đi giữa hai đỉnh sao cho tổng trọng số của các cạnh trên đường đi là nhỏ nhất.
- **Thuật toán tiêu biểu:** Dijkstra (tìm đường đi ngắn nhất từ một đỉnh đến mọi đỉnh khác trong đồ thị có trọng số không âm).

1.3 Bài toán 4: Chuyển đổi các Dạng biểu diễn Đồ thị và Cây

Viết chương trình C/C++, Python chuyển đổi giữa 4 dạng biểu diễn (adjacency matrix, adjacency list, extended adjacency list, adjacency map) cho 3 loại đồ thị (đơn đồ thị, đa đồ thị, đồ thị tổng quát) & 3 dạng biểu diễn của cây (array of parents, first-child next-sibling, graph-based).

Chứng minh. *Dây là một bài toán về thiết kế cấu trúc dữ liệu và cài đặt các thuật toán chuyển đổi. "Sẽ có* $3 \cdot A_4^2 + A_3^2 = 42$ *converter programs"*

1.3.1 Phân tích Yêu cầu

Tổng số 42 chương trình chuyển đổi được tính như sau:

- ***Chuyển đổi Đồ thị:*** *Có 4 dạng biểu diễn. Số cách chuyển đổi có thứ tự giữa 2 trong 4 dạng này là một chỉnh hợp chập 2 của 4:* $A_4^2 = \frac{4!}{(4-2)!} = 12$. *Quá trình này được yêu cầu cho 3 loại đồ thị, nên có tổng cộng* $3 \times 12 = 36$ *chương trình.*
- ***Chuyển đổi Cây:*** *Có 3 dạng biểu diễn. Số cách chuyển đổi có thứ tự giữa 2 trong 3 dạng này là:* $A_3^2 = \frac{3!}{(3-2)!} = 6$ *chương trình.*

Tổng cộng là $36 + 6 = 42$ *chương trình.*

1.3.2 Logic và Chiến lược Cài đặt cho Đồ thị

Một chiến lược hiệu quả là thiết kế một lớp `Graph` *sử dụng một dạng biểu diễn làm chuẩn (ví dụ:* ***danh sách kề****) và cung cấp các phương thức để chuyển đổi.*

- ***Lưu trữ chính:*** *Dùng danh sách kề, vì nó đủ linh hoạt để xử lý cả đơn đồ thị, đa đồ thị và đồ thị tổng quát.*
- ***Hàm khởi tạo (Constructor):*** *Cung cấp các hàm khởi tạo có thể nhận đầu vào từ các dạng khác, ví dụ* `Graph(matrix)`*. Logic của hàm này là duyệt ma trận và xây dựng danh sách kề tương ứng.*

- ***Phương thức xuất (Export Methods):*** *Cung cấp các phương thức để trả về các dạng khác, ví dụ* `toAdjacencyMatrix()`*. Logic của hàm này là duyệt danh sách kề và điền vào một ma trận mới.*

1.3.3 Logic và Chiến lược Cài đặt cho Cây

Tương tự, ta có thể thiết kế một lớp `Tree`*.*

- ***Lưu trữ chính:*** *Dùng danh sách kề (biểu diễn dạng đồ thị).*
- ***Các hàm chuyển đổi:***
  - ***Sang Mảng Cha:*** *Cần một đỉnh gốc. Thực hiện một thuật toán duyệt (BFS hoặc DFS) từ gốc. Khi đi từ đỉnh u đến v chưa thăm, ta đặt* `parent[v] = u`*.*
  - ***Từ Mảng Cha:*** *Duyệt mảng* `parent`*. Với mỗi đỉnh i (không phải gốc), thêm cạnh (*`parent[i]`*, i) vào danh sách kề.*
  - ***Sang First-Child Next-Sibling:*** *Dòi hỏi việc tạo các đối tượng nút và liên kết các con trỏ một cách đệ quy trong khi duyệt cây.*

1.4 Problems 1.1–1.6 Exercises 1.1–1.10

1.4.1 Bài toán 1.1: Kích thước Đồ thị Đầy đủ và Lưỡng phân Đầy đủ

Bài toán 1.1. Determine the size of the complete graph K_n on n vertices and the complete bipartite graph $K_{p,q}$ on $p + q$ vertices.

> **Chứng minh.** *Đồ thị đầy đủ K_n:Mỗi đỉnh được nối với tất cả $n-1$ đỉnh còn lại. Số cạnh là số cách chọn 2 đỉnh từ n đỉnh:*
>
> $$\text{Số cạnh} = \binom{n}{2} = \frac{n(n-1)}{2}$$

- ***Đồ thị lưỡng phân đầy đủ*** $K_{p,q}$***:*** *Đồ thị có hai tập đỉnh P (p đỉnh) và Q (q đỉnh). Mỗi đỉnh trong P được nối với tất cả các đỉnh trong Q. Theo quy tắc nhân, số cạnh là:*

$$\text{Số cạnh} = p \times q$$

1.4.2 Bài toán 1.2: Tính chất Lưỡng phân

Bài toán 1.2. Determine the values of n for which the circle graph C_n on n vertices is bipartite, and also the values of n for which the complete graph K_n is bipartite.

Chứng minh. *Một đồ thị là lưỡng phân nếu nó không chứa chu trình độ dài lẻ.*

- ***Đồ thị vòng*** C_n***:*** *C_n là một chu trình độ dài n. Do đó, nó là lưỡng phân khi và chỉ khi n không lẻ, tức là* ***n là số chẵn****.*
- ***Đồ thị đầy đủ*** K_n***:*** *Nếu $n \geq 3$, K_n luôn chứa một chu trình tam giác (K_3), là một chu trình lẻ. Do đó, K_n chỉ là lưỡng phân khi* ***$n \leq 2$****.*

1.4.3 Bài toán 1.3: Cây khung của Đồ thị

Bài toán 1.3. Give all the spanning trees of the graph in Fig. 1.30, and also the number of spanning trees of the underlying undirected graph.

Chứng minh.

- ***Cây khung của đồ thị có hướng:*** *Một cây khung có hướng (cây phân nhánh) yêu cầu một đỉnh gốc có bậc vào bằng 0. Quan sát đồ thị, mọi đỉnh đều có bậc vào lớn hơn 0. Do đó, số cây khung của đồ thị có hướng này là* ***0****.*
- ***Số cây khung của đồ thị vô hướng:*** *Việc liệt kê là không khả thi do số lượng lớn. Phương pháp tính toán chính xác là sử dụng* ***Định lý ma trận cây (Matrix Tree Theorem)****.*

1.4.4 Bài toán 1.4: Mở rộng biểu diễn Ma trận kề

Bài toán 1.4. Extend the adjacency matrix graph representation to support various operations.

Chứng minh.

- `del_edge(v, w)`*: Đặt* `A[v, w] = 0` *(hoặc giảm đi 1 nếu là đa đồ thị).*
- `edges()`*: Duyệt toàn bộ ma trận. Nếu* `A[i, j] > 0`*, thêm cạnh (i, j) vào danh sách.*
- `incoming(v)`*: Duyệt cột v của ma trận.*
- `outgoing(v)`*: Duyệt hàng v của ma trận.*
- `source(e), target(e)`*: Ma trận kề không lưu trữ định danh cạnh riêng lẻ. Cần một cấu trúc dữ liệu phụ (ví dụ: danh sách cạnh) để thực hiện thao tác này.*

1.4.5 Bài toán 1.5: Mở rộng biểu diễn Cây

Bài toán 1.5. Extend the first-child, next-sibling tree representation to support T.root(), T.number_of_children(v), and T.children(v) in $O(1)$ time.

Chứng minh.
- *`T.root()`: Cấu trúc dữ liệu của cây `T` cần lưu một con trỏ trực tiếp đến nút gốc.*
- *`T.number_of_children(v)`: Mỗi nút `v` cần được mở rộng để chứa một trường số nguyên `v.childCount` lưu trực tiếp số con của nó.*
- *`T.children(v)`: Thao tác này đã là $O(1)$ nếu nó chỉ trả về con trỏ đến con đầu tiên (`v.firstChild`).*

1.4.6 Bài toán 1.6: Kiểm tra Cây

Show how to double check that the graph-based representation of a tree is indeed a tree, in time linear in the size of the tree.

Chứng minh. *Một đồ thị là cây nếu nó **liên thông** và **không có chu trình**. Ta có thể kiểm tra cả hai điều kiện trong thời gian tuyến tính $O(n+m)$ bằng một lần duyệt đồ thị duy nhất (BFS hoặc DFS).*

1. *Bắt đầu duyệt từ một đỉnh s bất kỳ.*
2. ***Kiểm tra chu trình:** Trong khi duyệt, nếu gặp một cạnh đi đến đỉnh đã được thăm nhưng không phải đỉnh cha trực tiếp, đồ thị có chu trình.*
3. ***Kiểm tra tính liên thông:** Sau khi duyệt xong, nếu số đỉnh đã thăm bằng tổng số đỉnh của đồ thị, đồ thị liên thông.*

Nếu cả hai điều kiện đều thỏa mãn, đó là một cây.

1.4.7 Exercise 1.3: Tạo các đồ thị cơ bản

Implement algorithms to generate the path graph P_n, the circle graph C_n, and the wheel graph W_n on n vertices.

Chứng minh. *Giả sử các đỉnh được đánh số từ 0 đến $n-1$.*

- ***Đồ thị đường dẫn** P_n **(Path Graph):** Gồm các cạnh nối các đỉnh liền kề. Thuật toán: Dùng một vòng lặp `for i from 0 to n-2`, thêm cạnh $(i, i+1)$.*
- ***Đồ thị vòng** C_n **(Circle Graph):** Là một đồ thị P_n có thêm một cạnh nối hai đỉnh đầu mút. Thuật toán: Tạo đồ thị P_n trước, sau đó thêm cạnh $(n-1, 0)$.*

- ***Đồ thị bánh xe*** W_n ***(Wheel Graph):*** *Gồm một đỉnh trung tâm và một vòng* C_{n-1}*. Thuật toán: Chọn đỉnh* $n-1$ *làm tâm, tạo vòng* C_{n-1} *trên các đỉnh từ* 0 *đến* $n-2$*, sau đó thêm các cạnh* $(n-1, i)$ *với mọi* i *từ* 0 *đến* $n-2$*.*

1.4.8 Exercise 1.4: Tạo đồ thị đầy đủ

Implement an algorithm to generate the complete graph K_n and the complete bipartite graph $K_{p,q}$.

Chứng minh.
- ***Đồ thị đầy đủ*** K_n***:*** *Mỗi đỉnh nối với tất cả các đỉnh khác. Thuật toán: Dùng hai vòng lặp lồng nhau* `for i from 0 to n-2` *và* `for j from i+1 to n-1`*, thêm cạnh* (i, j)*.*

- ***Đồ thị lưỡng phân đầy đủ*** $K_{p,q}$***:*** *Các đỉnh chia thành hai tập* P *(cỡ* p*) và* Q *(cỡ* q*). Thuật toán: Coi các đỉnh từ* 0 *đến* $p-1$ *thuộc* P *và từ* p *đến* $p+q-1$ *thuộc* Q*. Dùng hai vòng lặp lồng nhau để thêm cạnh* (i, j) *với mọi* $i \in P$ *và* $j \in Q$*.*

1.4.9 Exercise 1.5: Cài đặt Lớp Ma trận kề mở rộng

Implement the extended adjacency matrix graph representation given in Problem 1.4, wrapped in a Python class.

Chứng minh. *Ta tạo một lớp* `AdjacencyMatrixGraph` *với các phương thức sau:*

- `__init__(self, n)`*: Khởi tạo một ma trận* $n \times n$ *chứa toàn số 0.*
- `add_edge(self, u, v)`*: Tăng giá trị của* `matrix[u][v]` *và* `matrix[v][u]`*.*
- `edges(self)`*: Duyệt qua ma trận và trả về danh sách các cạnh.*
- `incoming(self, v)`*: Duyệt cột* `v` *của ma trận.*
- `outgoing(self, v)`*: Duyệt hàng* `v` *của ma trận.*

1.4.10 Exercise 1.6: Cặp ghép hoàn hảo trong $K_{p,q}$

Enumerate all perfect matchings in the complete bipartite graph $K_{p,q}$.

Chứng minh. *Một cặp ghép hoàn hảo chỉ tồn tại khi hai tập đỉnh có cùng kích thước, tức là* $\mathbf{p = q}$*. Nếu* $p \neq q$*, không có cặp ghép hoàn hảo nào.*

Khi $p = q$*, một cặp ghép hoàn hảo là một song ánh từ tập đỉnh* P *sang tập đỉnh* Q*. Số lượng cặp ghép hoàn hảo là* $\mathbf{p!}$*. Ta có thể liệt kê chúng bằng thuật toán đệ quy quay lui (backtracking) để tìm tất cả các hoán vị ánh xạ các đỉnh từ* P *sang* Q*.*

1.4.11 Exercise 1.7: Tạo Cây nhị phân hoàn chỉnh

Implement an algorithm to generate the complete binary tree with n nodes.

Chứng minh. *Ta có thể sử dụng quy tắc đánh chỉ số của cấu trúc heap. Coi các nút được đánh số từ* 0 *đến* $n-1$*. Thuật toán: Duyệt qua mỗi nút* i *từ* 0 *đến* $n-1$*.*

- *Con trái của* i *là* $2i+1$*. Nếu* $2i+1 < n$*, thêm cạnh* $(i, 2i+1)$*.*
- *Con phải của* i *là* $2i+2$*. Nếu* $2i+2 < n$*, thêm cạnh* $(i, 2i+2)$*.*

1.4.12 Exercise 1.8: Tạo Cây ngẫu nhiên

Implement an algorithm to generate random trees with n nodes. Give the time and space complexity.

Chứng minh. *Một thuật toán đơn giản để tạo cây ngẫu nhiên:*

1. *Bắt đầu với một đồ thị chỉ có 1 đỉnh (đỉnh 0).*
2. *Dùng vòng lặp* `for i from 1 to n-1`*:*
   - *Thêm đỉnh mới* i *vào đồ thị.*
   - *Chọn một đỉnh* j *ngẫu nhiên trong số các đỉnh đã có (từ* 0 *đến* $i-1$*).*
   - *Thêm cạnh* (i, j)*.*

Độ phức tạp: *Thời gian là* $\mathbf{O(n)}$*, không gian là* $\mathbf{O(n)}$*.*

1.4.13 Exercise 1.9: Cài đặt `previous_sibling`

Bài toán 1.6. Give an implementation of operation `T.previous_sibling(v)` using the array-of-parents tree representation.

Chứng minh. *Biểu diễn bằng mảng cha* `parent[i]` *không lưu trực tiếp thông tin về thứ tự các nút con.*

1. *Tìm cha của* v*:* $p =$ `parent[v]`*.*
2. *Tạo một danh sách các con của* p *bằng cách duyệt toàn bộ mảng* `parent` *(độ phức tạp* $O(n)$*).*
3. *Tìm vị trí của* v *trong danh sách con này.*
4. *Nếu* v *không phải là con đầu tiên, trả về phần tử đứng ngay trước nó.*

Thao tác này có độ phức tạp thời gian là $\mathbf{O(n)}$*.*

1.4.14 Exercise 1.10: Cài đặt Lớp Cây mở rộng

Bài toán 1.7. Implement the extended first-child, next-sibling tree representation of Problem 1.5, wrapped in a Python class.

Chứng minh. *Ta sẽ cài đặt hai lớp là* `Node` *và* `Tree`*.*

- ***Lớp*** `Node`***:*** *Chứa các thuộc tính* `data`*,* `firstChild`*,* `nextSibling`*, và trường mở rộng* `childCount = 0`*.*
- ***Lớp*** `Tree`***:*** *Chứa thuộc tính* `root = None`*.*
- *Cần cài đặt một phương thức* `add_child(parent, new_node)` *để cập nhật các con trỏ và trường* `childCount` *một cách chính xác.*

Với cấu trúc này, các thao tác `root()`*,* `number_of_children(v)`*, và* `children(v)` *đều có thể được thực hiện trong thời gian* $O(1)$*.*

1.5 Bài toán 6

Viết chương trình C/C++, Python để giải bài toán tree edit distance problem bằng cách sử dụng: (a) Backtracking, (b) Branch-&-bound, (c) Divide-&-conquer, (d) Dynamic programming.

Chứng minh. *Báo cáo này trình bày và phân tích các phương pháp tiếp cận khác nhau để giải bài toán tính* ***Khoảng cách Chỉnh sửa Cây****.*

Giới thiệu Bài toán

Khoảng cách chỉnh sửa cây giữa hai cây T_1 *và* T_2 *là chi phí tối thiểu để biến đổi* T_1 *thành* T_2 *bằng cách sử dụng một chuỗi các thao tác cơ bản:* ***Thêm (Insert)****,* ***Xóa (Delete)****, và* ***Thay đổi nhãn (Relabel)*** *của các nút. Đây là một bài toán nền tảng trong so sánh cấu trúc.*

1.5.1 Phương pháp (a): Quay lui (Backtracking)

- ***Ý tưởng:*** *Đây là phương pháp* ***vét cạn****, thử tất cả các khả năng. Thuật toán sẽ khám phá một cách đệ quy mọi chuỗi thao tác chỉnh sửa có thể có để biến cây* T_1 *thành* T_2 *và tìm ra chuỗi có chi phí nhỏ nhất.*
- ***Logic:*** *Tại mỗi bước, thuật toán sẽ thử cả 3 lựa chọn (Xóa nút, Thêm nút, Thay đổi nhãn) và gọi đệ quy để giải quyết bài toán con còn lại.*
- ***Nhược điểm:*** *Số lượng chuỗi thao tác cần xét tăng theo cấp số nhân với kích thước của cây. Phương pháp này có độ phức tạp thời gian cực kỳ lớn và không khả thi trong thực tế.*

1.5.2 Phương pháp (b): Nhánh và Cận (Branch-and-Bound)

- ***Ý tưởng:*** *Đây là một phiên bản cải tiến của Quay lui. Nó cũng tìm kiếm trong không gian trạng thái nhưng thông minh hơn bằng cách* ***cắt tỉa*** *những nhánh tìm kiếm chắc chắn không dẫn đến kết quả tốt hơn.*

- ***Logic:*** *Thuật toán duy trì chi phí tốt nhất đã tìm thấy (min_cost). Trước khi đi sâu vào một nhánh, nó tính một* ***cận dưới (lower bound)*** *cho chi phí còn lại. Nếu tổng chi phí hiện tại và cận dưới lớn hơn hoặc bằng min_cost, nhánh đó sẽ bị loại bỏ.*

- ***Nhược điểm:*** *Mặc dù tốt hơn Backtracking, hiệu quả của nó phụ thuộc rất nhiều vào chất lượng của hàm tính cận dưới và vẫn có thể rất chậm.*

1.5.3 Phương pháp (c): Chia để trị (Divide-and-Conquer)

- ***Ý tưởng:*** *Chia một bài toán lớn thành các bài toán con* ***độc lập****, giải các bài toán con rồi kết hợp kết quả.*

- ***Phân tích:*** *Bài toán Tree Edit Distance có cấu trúc đệ quy, nhưng các bài toán con của nó (ví dụ: tính khoảng cách giữa các rừng cây con) lại* ***trùng lặp (overlapping)*** *rất nhiều lần. Chúng không độc lập.*

- ***Kết luận:*** *Do các bài toán con không độc lập, "chia để trị"không phải là phương pháp phù hợp. Đây chính là tình huống mà Quy hoạch động được thiết kế để giải quyết.*

1.5.4 Phương pháp (d): Quy hoạch động (Dynamic Programming)

- ***Ý tưởng:*** *Đây là phương pháp chuẩn và hiệu quả nhất. Nó giải quyết triệt để vấn đề* ***bài toán con chồng chéo*** *bằng cách lưu lại kết quả của các bài toán con đã giải vào một bảng và sử dụng lại khi cần.*

- ***Logic (Thuật toán Zhang-Shasha):*** *Thuật toán xây dựng một bảng 2D* `dp[i][j]` *để lưu khoảng cách chỉnh sửa giữa các cây con gốc tại nút i của T_1 và nút j của T_2. Bảng được điền từ các bài toán con nhỏ nhất (lá cây) đến các bài toán lớn hơn, dựa trên một công thức truy hồi phức tạp nhưng hiệu quả.*

- ***Ưu điểm:*** *Quy hoạch động giảm độ phức tạp từ cấp số nhân xuống đa thức, giúp bài toán trở nên khả thi trong thực tế.* □

1.6 Bài toán 7: Duyệt cây (Tree Traversal)

Viết chương trình C/C++, Python để duyệt cây: (a) preorder traversal, (b) postorder traversal, (c) top-down traversal, (d) bottom-up traversal.

Chứng minh. *Báo cáo này trình bày logic của bốn phương pháp duyệt cây cơ bản.*

1.6.1 Duyệt Tiền thứ tự (Preorder Traversal)

- ***Ý tưởng:*** *"Thăm gốc trước, con sau". Phương pháp này ưu tiên xử lý nút cha trước khi đi xuống các nút con của nó.*
- ***Thứ tự duyệt:***
  1. *Xử lý* ***nút gốc*** *hiện tại.*
  2. *Duyệt đệ quy các cây con từ* ***trái sang phải****.*
- ***Ví dụ ứng dụng:*** *Sao chép một cây, in cây thư mục, tạo mục lục cho sách.*
- ***Logic Thuật toán:*** *Thường được cài đặt bằng đệ quy.*

1.6.2 Duyệt Hậu thứ tự (Postorder Traversal)

- ***Ý tưởng:*** *"Thăm con trước, gốc sau". Một nút gốc chỉ được xử lý sau khi tất cả các cây con của nó đã được duyệt xong.*
- ***Thứ tự duyệt:***
  1. *Duyệt đệ quy các cây con từ* ***trái sang phải****.*
  2. *Xử lý* ***nút gốc*** *hiện tại.*
- ***Ví dụ ứng dụng:*** *Tính toán các giá trị phụ thuộc vào nút con (ví dụ: dung lượng thư mục), giải phóng bộ nhớ của cây.*
- ***Logic Thuật toán:*** *Thường được cài đặt bằng đệ quy.*

1.6.3 Duyệt từ trên xuống (Top-down / Level-order Traversal)

- ***Ý tưởng:*** *Đây là phương pháp* ***duyệt theo mức****, tương đương với thuật toán* ***BFS*** *trên cây. Nó duyệt cây theo từng tầng, từ trên xuống dưới.*
- ***Thứ tự duyệt:***
  1. *Thăm nút gốc (mức 0).*

2. *Thăm tất cả các nút ở mức 1 (từ trái sang phải).*
3. *Thăm tất cả các nút ở mức 2, và cứ thế tiếp tục.*

- ***Ví dụ ứng dụng:*** *Hiển thị sơ đồ tổ chức của một công ty theo từng cấp bậc.*
- ***Logic Thuật toán:*** *Sử dụng một* ***hàng đợi (queue)****.*

1.6.4 Duyệt từ dưới lên (Bottom-up Traversal)

- ***Ý tưởng:*** *Duyệt ngược lại với duyệt theo mức. Nó sẽ thăm các nút ở mức sâu nhất trước, sau đó đi dần lên các mức cao hơn, và thăm nút gốc cuối cùng.*
- ***Thứ tự duyệt:***
  1. *Thăm tất cả các nút ở mức sâu nhất (từ trái sang phải).*
  2. *Thăm tất cả các nút ở mức ngay trên đó, và cứ thế tiếp tục cho đến gốc.*
- ***Logic Thuật toán:*** *Có thể cài đặt bằng cách sửa đổi BFS:*
  1. *Thực hiện duyệt BFS như bình thường.*
  2. *Thay vì xử lý nút ngay khi lấy ra khỏi hàng đợi, ta đẩy nó vào một* ***ngăn xếp (stack)****.*
  3. *Sau khi BFS kết thúc, lấy lần lượt các nút ra khỏi ngăn xếp và xử lý chúng.*

1.7 Mục 5.1: Thuật toán tìm kiếm theo chiều rộng (BFS)

Lời giải chung cho các **Bài toán 8, 9, và 10**,

1.7.1 Đơn đồ thị (Simple Graph)

Đây là loại đồ thị cơ bản và có nhiều ràng buộc nhất.

× **Không có khuyên** (cạnh nối một đỉnh với chính nó).

× **Không có cạnh song song** (nhiều hơn một cạnh nối cùng hai đỉnh).

Ví dụ: Sơ đồ mạng lưới bạn bè, trong đó hai người hoặc là bạn hoặc không, không có "nhiều mức độ"bạn bè trực tiếp.

1.7.2 Đa đồ thị (Multigraph)

Loại đồ thị này nới lỏng hơn một chút so với đơn đồ thị.

✓ **Cho phép** có cạnh song song.

× **Không cho phép** có khuyên.

Ví dụ: Bản đồ giao thông, có thể có nhiều con đường hoặc tuyến bay khác nhau cùng nối trực tiếp hai thành phố.

1.7.3 Đồ thị tổng quát (General Graph / Pseudograph)

Đây là loại đồ thị tự do và tổng quát nhất.

✓ **Cho phép** có cạnh song song.

✓ **Cho phép** có khuyên.

Ví dụ: Sơ đồ luồng của một chương trình máy tính, trong đó một trạng thái có thể quay trở lại chính nó (tạo thành khuyên).

1.7.4 Ý tưởng cốt lõi

Thuật toán BFS khám phá đồ thị theo từng "lớp"hoặc "mức"khoảng cách, giống như cách sóng nước lan tỏa. Bắt đầu từ một đỉnh nguồn, nó sẽ thăm tất cả các đỉnh hàng xóm trực tiếp, sau đó mới đến các hàng xóm của các đỉnh đó. Cách tiếp cận này đảm bảo rằng các đỉnh gần nguồn sẽ luôn được duyệt trước các đỉnh ở xa hơn.

1.7.5 Cơ chế Hoạt động và Mã giả

Để thực hiện ý tưởng trên, BFS sử dụng một **hàng đợi (queue)** (hoạt động theo nguyên tắc Vào-trước-Ra-trước) và một cấu trúc để ghi nhớ các **đỉnh đã thăm** (`visited`).

Algorithm 1 Mã giả Tìm kiếm theo chiều rộng (BFS)

```
1: procedure BFS(G, S)                          ▷ G: Đồ thị, S: Đỉnh bắt đầu
2:     Khởi tạo hàng đợi Q và tập visited
3:     Q.enqueue(S)                             ▷ Thêm đỉnh bắt đầu vào hàng đợi
4:     visited.add(S)                           ▷ Đánh dấu đã thăm
5:     while Q không rỗng do
6:         u ← Q.dequeue()                      ▷ Lấy đỉnh hiện tại ra để xử lý
7:         Process(u)                           ▷ Ví dụ: In tên đỉnh
8:         for mỗi đỉnh kề v của u do
9:             if v chưa có trong visited then
10:                visited.add(v)               ▷ Đánh dấu v đã thăm
11:                Q.enqueue(v)                 ▷ Thêm v vào hàng đợi
12:            end if
13:        end for
14:    end while
15: end procedure
```

1.7.6 Cách Thuật toán Giải quyết Bài toán

Logic của BFS đủ mạnh để giải quyết cả ba bài toán trên các loại đồ thị khác nhau:

- **Đơn đồ thị (Bài toán 8):** Là trường hợp áp dụng chuẩn.
- **Đa đồ thị (Bài toán 9):** Các cạnh song song không gây ra vấn đề. Nhờ có tập `visited`, một đỉnh kề chỉ được thêm vào hàng đợi đúng một lần duy nhất, dù có nhiều cạnh nối đến nó.
- **Đồ thị tổng quát (Bài toán 10):** Các khuyên (loop) cũng không gây ra vấn đề. Khi duyệt một đỉnh u, nó sẽ thấy chính nó là hàng xóm. Tuy nhiên, vì u đã được đánh dấu là `visited`, thuật toán sẽ bỏ qua.

1.8 Mục 5.2: Thuật toán tìm kiếm theo chiều sâu (DFS)

[Bài toán 11, 12, 13] Implement the depth-first search on a finite simple graph, multigraph, and general graph.

Thuật toán DFS có thể được cài đặt một cách tổng quát để hoạt động trên cả ba loại đồ thị. Dưới đây là phần trình bày chi tiết.

Ý tưởng và Cơ chế

Thuật toán Tìm kiếm theo chiều sâu (DFS) hoạt động bằng cách khám phá một nhánh của đồ thị **sâu hết mức có thể**. Khi gặp một "ngõ cụt"(đỉnh không còn hàng xóm nào chưa được thăm), thuật toán sẽ **quay đầu (backtracking)** để khám phá các nhánh khác. Cơ chế này thường được cài đặt một cách tự nhiên bằng **đệ quy (recursion)** và sử dụng một tập `visited` để ghi nhớ các đỉnh đã thăm.

Mã giả

Algorithm 2 Tìm kiếm theo chiều sâu (DFS)

Khởi tạo: `visited` là một tập rỗng
procedure DFS_Main(G, S) ▷ Hàm chính để bắt đầu từ đỉnh S
    DFS_Recursive(G, S)
end procedure

procedure DFS_Recursive(G, u) ▷ Hàm đệ quy thực hiện duyệt
    `visited`.add(u) ▷ Đánh dấu đỉnh hiện tại đã thăm
    Process(u) ▷ Ví dụ: In tên đỉnh
    for mỗi đỉnh kề v của u **do**
        if v chưa có trong `visited` **then**
            DFS_Recursive(G, v) ▷ Đi sâu vào nhánh mới
        end if
    end for
end procedure

Cách Thuật toán Giải quyết Bài toán

Cơ chế của DFS đủ mạnh để xử lý các loại đồ thị khác nhau mà không cần thay đổi logic:

- **Đơn đồ thị (Bài toán 11):** Là trường hợp áp dụng chuẩn.
- **Đa đồ thị (Bài toán 12):** Các cạnh song song không gây ra vấn đề. Nhờ có tập `visited`, một đỉnh kề sẽ chỉ được gọi đệ quy đúng một lần.
- **Đồ thị tổng quát (Bài toán 13):** Các khuyên (loop) cũng không gây ra vấn đề. Khi duyệt một đỉnh u, nó sẽ thấy chính nó là hàng xóm, nhưng vì u đã được đánh dấu là `visited`, lệnh gọi đệ quy vô hạn sẽ không xảy ra.

2 Project 5: Shortest Path Problems on Graphs

2.1 Mục 6.1: Thuật toán Dijkstra

Bài toán 2.1 (Bài toán 14, 15, 16)**.** Implement the Dijkstra's algorithm to find the shortest path problem on a finite simple graph, multigraph, and general graph.

Chứng minh. *Thuật toán Dijkstra có thể được cài đặt một cách tổng quát để giải bài toán tìm đường đi ngắn nhất trên cả ba loại đồ thị, với điều kiện tiên quyết là đồ thị* ***không có cạnh với trọng số âm****.*

Ý tưởng cốt lõi

Thuật toán Dijkstra hoạt động dựa trên tư tưởng ***tham lam (greedy)****. Bắt đầu từ một đỉnh nguồn S, nó duy trì một tập các đỉnh đã được "chốt"khoảng cách ngắn nhất. Ở mỗi bước, nó chọn đỉnh u chưa được chốt mà có khoảng cách tạm thời ngắn nhất đến S, chốt khoảng cách đó, và cập nhật lại khoảng cách đến các đỉnh hàng xóm của u nếu tìm thấy đường đi mới ngắn hơn.*

Cơ chế Hoạt động và Mã giả

Để hoạt động hiệu quả, Dijkstra sử dụng một mảng `dist` *để lưu khoảng cách, và một* ***hàng đợi ưu tiên (priority queue)*** *để nhanh chóng tìm ra đỉnh có khoảng cách nhỏ nhất.*

Algorithm 3 Thuật toán Dijkstra

```
1:  procedure DIJKSTRA(G, S)                    ▷ G: Đồ thị, S: Đỉnh nguồn
2:      Khởi tạo dist[v] ← ∞ cho mọi đỉnh v; dist[S] ← 0.
3:      Khởi tạo hàng đợi ưu tiên PQ chứa tất cả các đỉnh.
4:      while PQ không rỗng do
5:          u ← PQ.extract_min()                ▷ Lấy đỉnh u có dist[u] nhỏ nhất
6:          for mỗi đỉnh kề v của u do
7:              new_dist ← dist[u] + weight(u, v)
8:              if new_dist < dist[v] then
9:                  dist[v] ← new_dist
10:                 PQ.decrease_key(v, new_dist)
11:             end if
12:         end for
13:     end while
14: end procedure
```

Cách Thuật toán Giải quyết Bài toán

Logic của Dijkstra đủ mạnh để xử lý các loại đồ thị khác nhau:

- ***Đơn đồ thị (Bài toán 14):*** *Là trường hợp áp dụng chuẩn.*
- ***Đa đồ thị (Bài toán 15):*** *Các cạnh song song được xử lý một cách tự nhiên. Thuật toán sẽ xét từng cạnh và cuối cùng sẽ chọn cạnh có trọng số nhỏ nhất để tối ưu hóa đường đi.*
- ***Đồ thị tổng quát (Bài toán 16):*** *Các khuyên (loop) có trọng số không âm sẽ luôn bị bỏ qua, vì đường đi qua một khuyên (u, u) sẽ có độ dài $dist[u] + weight(u, u) \geq dist[u]$ và không bao giờ cải thiện được khoảng cách.*

3 Project: Integer Partition – Đồ Án: Phân Hoạch Số Nguyên

Bài toán 1: Biểu đồ Ferrers & Chuyển vị

Nhập $n, k \in \mathbb{N}$. Viết chương trình C/C++, Python để in ra $p_k(n)$ biểu đồ Ferrers F & biểu đồ Ferrers chuyển vị F^T cho mỗi phân hoạch $\lambda = (\lambda_1, \ldots, \lambda_k) \in (\mathbb{N}^*)^k$ có định dạng các dấu chấm được biểu diễn bởi dấu $*$.

1. Logic Tìm kiếm Tất cả các Phân hoạch

Một **phân hoạch** của số nguyên n thành k phần là một bộ số nguyên dương $(\lambda_1, \ldots, \lambda_k)$ sao cho $\sum \lambda_i = n$. Để tránh lặp lại các hoán vị, ta dùng quy ước sắp xếp không tăng: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k > 0$.

Chiến lược để tìm tất cả các phân hoạch này là sử dụng **đệ quy quay lui (recursive backtracking)**. Ta sẽ xây dựng phân hoạch từng phần tử một, từ λ_1 đến λ_k, đồng thời đảm bảo điều kiện sắp xếp. Quá trình này sẽ tìm ra tất cả các phân hoạch duy nhất.

2. Logic Vẽ Biểu đồ Ferrers

Biểu đồ Ferrers là cách trực quan hóa một phân hoạch. Với một phân hoạch $\lambda = (\lambda_1, \ldots, \lambda_k)$, biểu đồ sẽ có k hàng, trong đó hàng thứ i có λ_i dấu sao.

Ví dụ: Phân hoạch $(4, 2, 1)$ của số 7 được biểu diễn là:

```
* * * *
* *
*
```

3. Logic Tìm và Vẽ Biểu đồ Ferrers Chuyển vị

Biểu đồ chuyển vị F^T được tạo ra bằng cách lật biểu đồ gốc F qua đường chéo chính, biến hàng thành cột. Về mặt thuật toán, ta sẽ tìm **phân hoạch liên hợp (conjugate partition)** λ' rồi vẽ biểu đồ của nó.

Phần tử thứ j của phân hoạch liên hợp, λ'_j, được tính bằng số lượng các phần tử trong phân hoạch gốc λ có giá trị lớn hơn hoặc bằng j.

Ví dụ: Với phân hoạch $\lambda = (4, 2, 1)$:

- λ'_1 (số phần tử trong $\lambda \geq 1$) $= 3$
- λ'_2 (số phần tử trong $\lambda \geq 2$) $= 2$
- λ'_3 (số phần tử trong $\lambda \geq 3$) $= 1$
- λ'_4 (số phần tử trong $\lambda \geq 4$) $= 1$

Phân hoạch liên hợp là $\lambda' = (3, 2, 1, 1)$. Biểu đồ chuyển vị tương ứng là:

```
* * *
* *
*
*
```

Bài toán 2

Nhập $n, k \in \mathbb{N}$. Đếm số phân hoạch $p_{max}(n, k)$ của n sao cho phần tử lớn nhất là k. Viết chương trình C/C++, Python để đếm số phân hoạch này. So sánh $p_k(n)$ (số phân hoạch của n thành k phần) và $p_{max}(n, k)$.

So sánh $p_k(n)$ và $p_{max}(n, k)$

Kết luận: Hai cách đếm này luôn cho cùng một kết quả, tức là:

$$p_k(n) = p_{max}(n, k)$$

Lập luận bằng Biểu đồ Ferrers

Ta có thể chứng minh đẳng thức trên một cách trực quan bằng cách sử dụng **biểu đồ Ferrers** và phép **chuyển vị** (transpose).

- Một phân hoạch của n thành k phần (được đếm bởi $p_k(n)$) sẽ có biểu đồ Ferrers với đúng k hàng. Ví dụ, phân hoạch $4+2+1$ của $n = 7$ thành $k = 3$ phần có biểu đồ:

```
* * * *
* *
*
```

- Khi ta chuyển vị biểu đồ này (lật qua đường chéo chính), số hàng trở thành số cột và ngược lại. Biểu đồ chuyển vị sẽ là:

```
* * *
* *
*
*
```

- Biểu đồ mới này tương ứng với phân hoạch $3+2+1+1$. Ta thấy rằng phần tử lớn nhất của nó là 3, chính bằng số hàng của biểu đồ ban đầu ($k = 3$).

Phép chuyển vị này tạo ra một **song ánh** (tương ứng 1-1) giữa tập các phân hoạch của n thành k phần và tập các phân hoạch của n có phần tử lớn nhất là k. Vì tồn tại một song ánh, số phần tử của hai tập hợp phải bằng nhau.

Logic để viết chương trình đếm $p_{max}(n, k)$

Để tính $p_{max}(n, k)$, ta dựa vào định nghĩa:

- Mọi phân hoạch được đếm bởi $p_{max}(n, k)$ đều phải chứa phần tử lớn nhất là k.
- Do đó, mỗi phân hoạch có dạng $n = k+\lambda_2+\lambda_3+\dots$ với điều kiện $k \geq \lambda_i > 0$.
- Bài toán trở thành tìm số cách phân hoạch cho số còn lại là $\mathbf{n-k}$, với điều kiện là tất cả các phần tử trong phân hoạch mới phải **nhỏ hơn hoặc bằng** k.
- Đây là một bài toán quy hoạch động kinh điển. Ta có thể định nghĩa một hàm `count(target, max_val)` để đếm số phân hoạch của `target` với các phần tử không lớn hơn `max_val`. Khi đó, $p_{max}(n, k) = \text{count}(n-k, k)$. □

3.1 Bài toán 3: Phân hoạch Tự liên hợp

Nhập $n, k \in \mathbb{N}$. (a) Đếm số phân hoạch tự liên hợp của n có k phần, ký hiệu $p_k^{\text{selfcjg}}(n)$, rồi in ra các phân hoạch đó. (b) Đếm số phân hoạch của n có k phần lẻ, phân biệt, rồi so sánh với $p_k^{\text{selfcjg}}(n)$. (c) Thiết lập công thức truy hồi cho $p_k^{\text{selfcjg}}(n)$, rồi implementation bằng đệ quy và quy hoạch động.

Định nghĩa và Logic cốt lõi

Một phân hoạch được gọi là **tự liên hợp (self-conjugate)** nếu biểu đồ Ferrers của nó đối xứng qua đường chéo chính. Một tính chất quan trọng của các phân hoạch này là **phần tử lớn nhất của nó bằng số phần tử của nó**. Tức là, một phân hoạch $\lambda = (\lambda_1, \ldots, \lambda_k)$ là tự liên hợp có k phần khi và chỉ khi $\lambda_1 = k$.

Hệ thức truy hồi (Câu c)

Ta có thể xây dựng một hệ thức truy hồi bằng cách "bóc"lớp ngoài cùng (hook) của biểu đồ Ferrers đối xứng. Lớp này gồm hàng đầu và cột đầu, chứa tổng cộng $2k-1$ ô. Phần còn lại cũng là một phân hoạch tự liên hợp của số $n' = n-(2k-1)$. Phân hoạch con này có thể có j phần, với j bất kỳ từ 0 đến $k-1$. Do đó, số cách phân hoạch tự liên hợp của n thành k phần bằng tổng số cách phân hoạch tự liên hợp của $n-(2k-1)$ thành $j < k$ phần.

$$p_k^{\text{selfcjg}}(n) = \sum_{j=0}^{k-1} p_j^{\text{selfcjg}}(n-(2k-1))$$

Hệ thức này có thể được cài đặt bằng đệ quy (có ghi nhớ) hoặc quy hoạch động.

So sánh với phân hoạch có các phần tử lẻ, phân biệt (Câu b)

Kết luận: Nói chung, hai giá trị này **không bằng nhau** khi xét một k cụ thể.

Định lý Euler chỉ ra rằng *tổng số* phân hoạch tự liên hợp của n (trên mọi k) bằng *tổng số* phân hoạch của n thành các phần tử lẻ và phân biệt. Đẳng thức này không đúng khi cố định số phần tử k.

Phản ví dụ: Với $n = 8, k = 3$.

- $p_3^{\text{selfcjg}}(8) = 1$. Phân hoạch duy nhất là $(3, 3, 2)$.
- Số phân hoạch của 8 thành 3 phần tử lẻ, phân biệt là 0 (vì tổng nhỏ nhất $1+3+5 = 9 > 8$).

Vì $1 \neq 0$, hai cách đếm này không tương đương cho một k cố định.

Về việc in các phân hoạch (Câu a)

Để in ra các phân hoạch, ta cần dùng thuật toán đệ quy quay lui dựa trên hệ thức truy hồi đã tìm ra. Khi hàm đệ quy tìm thấy một phân hoạch con (ví dụ của $n-(2k-1)$), ta sẽ tái tạo lại phân hoạch lớn ban đầu bằng cách "bọc"thêm lớp hook $2k-1$ vào và in ra kết quả. □