Môn: Toán Tổ Hợp và Lý Thuyết Đồ Thị

Giảng viên hướng dẫn: [Điền tên giảng viên]

Sinh viên Thực hiện: Phạm Minh Khoa
MSVV: 2201700189

Mục lục

Đồ án 4: Phân Hoạch Số Nguyên

Bài toán 1: In biểu đồ Ferrers và biểu đồ Ferrers chuyển vị

Toán học

Một **phân hoạch** của số nguyên dương n thành k phần là một cách viết n thành tổng của đúng k số nguyên dương. Theo quy ước, các số này (gọi là các phần) được sắp xếp theo thứ tự giảm dần để tránh lặp lại các cách viết giống nhau.

- **Biểu đồ Ferrers (F)** biểu diễn phân hoạch này bằng k hàng dấu $*$, với hàng thứ i có số dấu $*$ tương ứng với phần tử thứ i của phân hoạch.
- **Biểu đồ Ferrers chuyển vị ($F^\top$)** được tạo ra bằng cách hoán đổi vai trò của hàng và cột trong biểu đồ F. Phân hoạch tương ứng với $F^\top$ được gọi là phân hoạch liên hợp.

Thuật toán

1. Sử dụng thuật toán đệ quy quay lui để tìm tất cả các bộ số thỏa mãn điều kiện. Hàm đệ quy sẽ có các tham số như (tổng còn lại, số phần còn lại, giá trị tối đa cho phần tử tiếp theo) để đảm bảo thứ tự giảm dần và tránh lặp lại.
2. Với mỗi phân hoạch tìm được, duyệt qua từng phần tử và in ra số dấu $*$ tương ứng.
3. Từ phân hoạch gốc, tạo ra phân hoạch liên hợp. Phần tử thứ i của phân hoạch liên hợp bằng số phần tử trong phân hoạch gốc có giá trị lớn hơn hoặc bằng i. Sau đó, in biểu đồ Ferrers cho phân hoạch liên hợp này.

Ý tưởng

Ý tưởng chính là sử dụng đệ quy quay lui để duyệt qua tất cả các khả năng chia số n thành k phần một cách có hệ thống. Với mỗi khả năng hợp lệ tìm được, ta sẽ tiến hành vẽ biểu đồ.

Bước giải (Ví dụ: n=6, k=3)

1. Thuật toán đệ quy sẽ tìm ra các phân hoạch: ‘4+1+1‘, ‘3+2+1‘, ‘2+2+2‘.
2. Xét phân hoạch ‘3+2+1‘:
3. Vẽ biểu đồ Ferrers (F):

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

4. Vẽ biểu đồ chuyển vị ($F^\top$) bằng cách đọc theo cột của F:

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

Bài toán 2: So sánh số phân hoạch $p_k(n)$ và $p_{\max}(n, k)$

Toán học (Derivation)

- Số cách phân hoạch n thành k phần ($p_k(n)$) bằng số cách phân hoạch n có phần tử lớn nhất là k ($p_{\max}(n, k)$).
- **Chứng minh:** Phép chuyển vị biểu đồ Ferrers tạo ra một song ánh giữa hai tập hợp phân hoạch này. Một biểu đồ có k hàng khi chuyển vị sẽ có hàng đầu tiên dài k và ngược lại.
- **Xây dựng công thức truy hồi:** Để tính $p_k(n)$, ta sẽ chia các phân hoạch của n thành k phần ra làm hai loại không giao nhau:
  - **Loại 1:** Phân hoạch có chứa ít nhất một số 1. Nếu ta bỏ đi một số 1, ta sẽ còn lại một phân hoạch của số $n - 1$ thành $k - 1$ phần là $p_{k-1}(n - 1)$.
  - **Loại 2:** Phân hoạch không chứa số 1 nào. Nếu trừ 1 ở mỗi phần trong k phần đó, tổng sẽ giảm đi k. Ta sẽ thu được một phân hoạch mới của số $n - k$ và vẫn có đúng k phần là $p_k(n - k)$.
- Cộng hai trường hợp lại, ta có công thức truy hồi: $p_k(n) = p_{k-1}(n - 1) + p_k(n - k)$.

Thuật toán

Ta sẽ sử dụng phương pháp **Dynamic Programming** để cài đặt công thức truy hồi. Một bảng 2D `dp[i][j]` sẽ được dùng để lưu giá trị của $p_j(i)$.

Ý tưởng

Ý tưởng là xây dựng một bảng lưu kết quả các bài toán con. Giá trị của một ô trong bảng được tính dựa trên các giá trị đã được tính trước đó, giúp tránh việc tính toán lặp lại mà phương pháp đệ quy thông thường gặp phải.

Bước giải (Ví dụ: tính $p_3(6)$)

1. Ta có $p_3(6) = p_2(5) + p_3(3)$.
2. Tính $p_3(3)$: chỉ có ‘1+1+1‘ -> $p_3(3) = 1$.
3. Tính $p_2(5) = p_1(4) + p_2(3)$:
   - $p_1(4)$: chỉ có ‘4‘ -> $p_1(4) = 1$.
   - $p_2(3)$: chỉ có ‘2+1‘ -> $p_2(3) = 1$.
   - Suy ra $p_2(5) = 1 + 1 = 2$.
4. Kết quả cuối cùng: $p_3(6) = 2 + 1 = 3$.

Bài toán 3: Số phân hoạch tự liên hợp

Toán học

- Một phân hoạch là **tự liên hợp** nếu biểu đồ Ferrers của nó đối xứng qua đường chéo chính.
- Số phân hoạch tự liên hợp của n bằng số phân hoạch của n thành các phần tử lẻ và phân biệt.

Thuật toán

- **(a):** sinh tất cả các phân hoạch của n thành k phần, sau đó với mỗi phân hoạch, kiểm tra tính tự liên hợp của nó bằng cách so sánh với phân hoạch liên hợp.
- **(b):** viết một hàm khác để sinh tất cả các phân hoạch của n, và chỉ đếm những phân hoạch có số lượng phần tử là một số lẻ.
- **(c):** Hướng tiếp cận khả thi là cài đặt thuật toán dựa trên định lý tương đương (phân hoạch thành các phần tử lẻ và phân biệt).

Ý tưởng

Ý tưởng là liệt kê các trường hợp và kiểm tra điều kiện. Với câu (a), ta liệt kê các phân hoạch có k phần rồi kiểm tra tính tự liên hợp. Với câu (b), ta liệt kê tất cả các phân hoạch rồi kiểm tra xem số lượng phần có phải là số lẻ không.

Bước giải (Ví dụ: n=8, k=4)

1. **Giải câu (a):**
   - Liệt kê các phân hoạch của 8 thành 4 phần: ‘5+1+1+1‘, ‘4+2+1+1‘, ‘3+3+1+1‘, ‘3+2+2+1‘, ‘2+2+2+2‘.
   - Kiểm tra từng cái. Ví dụ ‘4+2+1+1‘: chuyển vị của nó cũng là ‘4+2+1+1‘ -> Hợp lệ.
   - Kết quả: Chỉ có 1 phân hoạch là ‘4+2+1+1‘.
2. **Giải câu (b):**
   - Liệt kê các phân hoạch của 8 thành 1, 3, 5, 7 phần.
   - Đếm tổng số lượng: 1 (1 phần) + 5 (3 phần) + 3 (5 phần) + 1 (7 phần) = 10.
3. **So sánh:** $p_4^{\text{selfcjg}}(8) = 1$ và số phân hoạch thành lẻ phần là 10.

Đồ án 5: Các Bài Toán Duyệt Đồ Thị & Cây

Bài toán 4: Chuyển đổi giữa các dạng biểu diễn đồ thị

Thuật toán

Bài toán yêu cầu viết các chương trình chuyển đổi. Chúng đều dựa trên việc duyệt qua cấu trúc dữ liệu nguồn và điền thông tin vào cấu trúc dữ liệu đích. Em lấy ví dụ là trường hợp Ma trận kề sang Danh sách kề cho đồ thị vô hướng:

1. Khởi tạo một danh sách kề rỗng.
2. Dùng hai vòng lặp i và j để duyệt qua ma trận kề M.
3. Nếu $M[i][j]$ bằng 1, điều đó có nghĩa là có cạnh nối từ đỉnh i đến đỉnh j.
4. Thêm j vào danh sách kề của i.

Ý tưởng

Ý tưởng chung là lặp qua toàn bộ cấu trúc dữ liệu đầu vào, và với mỗi thông tin về cạnh tìm được (ví dụ `matrix[i][j] == 1`), ta sẽ thêm thông tin tương ứng vào cấu trúc dữ liệu đầu ra (ví dụ `adjList[i].push_back(j)`).

Bước giải (Ví dụ: Ma trận sang Danh sách kề)

1. Cho ma trận:

```
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
```

2. Khởi tạo 4 danh sách kề rỗng: `adj[0]`, `adj[1]`, `adj[2]`, `adj[3]`.
3. Duyệt hàng 0: thấy `M[0][1]=1` và `M[0][2]=1`. Thêm 1 và 2 vào `adj[0]`. Kết quả: `adj[0] = {1, 2}`.
4. Duyệt hàng 1: thấy `M[1][0]=1` và `M[1][3]=1`. Thêm 0 và 3 vào `adj[1]`. Kết quả: `adj[1] = {0, 3}`.
5. Làm tương tự cho các hàng còn lại.

Bài toán 5: Làm Problems 1.1-1.6 và Exercises 1.1-1.10 [Val21]

Toán học và Thuật toán

Problems:

- **Problem 1.1:** Số cạnh của K_n là $n(n-1)/2$. Số cạnh của $K_{p,q}$ là $p \times q$.
- **Problem 1.2:** Một đồ thị là hai phía khi và chỉ khi nó không chứa chu trình lẻ, do đó C_n là hai phía khi n chẵn. K_n là hai phía khi $n \leq 2$.
- **Problem 1.6:** Một đồ thị vô hướng là một cây nếu nó liên thông và có $V-1$ cạnh (với V là số đỉnh). Thuật toán sẽ kiểm tra hai điều kiện này. Tính liên thông có thể được kiểm tra bằng BFS hoặc DFS.

Exercises:

- **Exercise 1.3 & 1.4:** Thuật toán để sinh các đồ thị đặc biệt:

- P_n: Tạo n đỉnh, sau đó thêm các cạnh $(i, i+1)$ với i từ 0 đến $n-2$.
- K_n: Tạo n đỉnh, sau đó thêm cạnh giữa mọi cặp đỉnh (i, j) với $i < j$.
- $K_{p,q}$: Tạo $p+q$ đỉnh, chia thành hai tập. Thêm cạnh giữa mọi đỉnh của tập thứ nhất với mọi đỉnh của tập thứ hai.

- **Exercise 1.6:** Đếm số cặp ghép hoàn hảo trong $K_{p,q}$. Giả sử $p \leq q$. Số cách ghép là số chỉnh hợp chập p của q: $P(q,p) = q!/(q-p)!$.
- **Exercise 1.7:** Sinh cây nhị phân hoàn chỉnh. Với n nút, nút cha của nút i (với $i > 0$) là nút $\lfloor (i-1)/2 \rfloor$. Dựa vào công thức này để thêm các cạnh.

Ý tưởng

Các bài tập này nhằm củng cố các định nghĩa và tính chất cơ bản của các loại đồ thị và cây đặc biệt. Ý tưởng là áp dụng trực tiếp các công thức toán học hoặc các thuật toán duyệt đồ thị cơ bản để giải quyết.

Bước giải (Ví dụ: Problem 1.6 - Kiểm tra cây)

1. Cho một đồ thị G có V đỉnh và E cạnh.
2. **Kiểm tra tính liên thông:** Dùng BFS hoặc DFS bắt đầu từ một đỉnh bất kỳ (ví dụ đỉnh 0). Sau khi duyệt xong, đếm số đỉnh đã thăm. Nếu số đỉnh đã thăm nhỏ hơn V, đồ thị không liên thông -> không phải là cây.
3. **Kiểm tra số cạnh:** Nếu đồ thị liên thông, đếm số cạnh của nó. Nếu số cạnh đúng bằng V-1 -> là cây. Ngược lại -> không phải là cây.

Bài toán 6: Tree Edit Distance

Toán học

Khoảng cách sửa cây (Tree Edit Distance) giữa hai cây có thứ tự (ordered tree) là chi phí tối thiểu để biến đổi cây thứ nhất (T1) thành cây thứ hai (T2) thông qua một chuỗi các phép toán cơ bản:

1. **Chèn (Insert):** Thêm một nút mới vào cây.
2. **Xóa (Delete):** Xóa một nút khỏi cây.
3. **Thay thế (Substitute/Rename):** Thay đổi nhãn (label) của một nút.

Mỗi phép toán này thường được gán một chi phí, ví dụ là 1.

Thuật toán

Phương pháp tiêu chuẩn để giải quyết bài toán này là quy hoạch động (Dynamic Programming).

- **Bài toán con:** Ta định nghĩa `dp[i][j]` là chi phí nhỏ nhất để biến đổi cây con gốc tại nút i của T1 thành cây con gốc tại nút j của T2.
- **Công thức truy hồi:** Giá trị của `dp[i][j]` được tính bằng cách lấy giá trị nhỏ nhất của 3 trường hợp:

1. **Xóa nút** i: Chi phí bằng $1 + \text{dp}[\text{rừng con của } i][j]$.
2. **Chèn nút** j: Chi phí bằng $1 + \text{dp}[i][\text{rừng con của } j]$.
3. **Thay thế** i **bằng** j: Chi phí bằng $\text{cost}(i, j) + \text{dp}[\text{rừng con của } i][\text{rừng con của } j]$.

Ý tưởng

Ý tưởng là xây dựng lời giải cho bài toán lớn từ các lời giải tối ưu của các bài toán con. Cụ thể là tính khoảng cách giữa hai cây lớn dựa trên khoảng cách đã biết của các cây con nhỏ hơn của chúng.

Bước giải

Thuật toán này rất phức tạp để giải tay. Tuy nhiên, ý tưởng cơ bản là điền vào một bảng quy hoạch động. Giá trị tại mỗi ô `dp[i][j]` được tính bằng cách xem xét 3 lựa chọn (xóa, chèn, thay thế) và chọn ra phương án có chi phí thấp nhất.

Bài toán 7: Duyệt cây

Thuật toán

- **Pre-order:** Gốc -> Trái -> Phải.
- **In-order:** Trái -> Gốc -> Phải.
- **Post-order:** Trái -> Phải -> Gốc.

Các thuật toán này được cài đặt một cách tự nhiên bằng đệ quy. Hàm đệ quy sẽ thăm nút hiện tại và gọi chính nó cho các cây con theo đúng thứ tự.

Ý tưởng

Sự khác biệt giữa 3 kiểu duyệt nằm ở thời điểm ta "thăm"(xử lý/in ra) nút gốc so với thời điểm ta duyệt các cây con của nó.

Bước giải (Ví dụ với cây có gốc A, con trái B, con phải C)

1. **Pre-order:** Thăm A -> Duyệt cây con B -> Duyệt cây con C. Kết quả: A, B, C.
2. **In-order:** Duyệt cây con B -> Thăm A -> Duyệt cây con C. Kết quả: B, A, C.
3. **Post-order:** Duyệt cây con B -> Duyệt cây con C -> Thăm A. Kết quả: B, C, A.

Bài toán 8: Breadth-first search (BFS) trên đồ thị đơn

Thuật toán

Duyệt một đồ thị vô hướng, không có cạnh song song hay khuyên, bắt đầu từ một đỉnh cho trước.

- BFS duyệt đồ thị theo từng mức. Nó khám phá tất cả các đỉnh hàng xóm ở mức hiện tại trước khi đi xuống mức sâu hơn.

- **Cấu trúc dữ liệu:** Thuật toán sử dụng một hàng đợi (Queue) để lưu các đỉnh sẽ được thăm và một tập hợp `visited` để đánh dấu các đỉnh đã được thăm, tránh việc duyệt lặp lại.

Ý tưởng

Tưởng tượng như một viên sỏi ném xuống mặt nước, BFS lan ra theo các vòng tròn đồng tâm. Nó dùng hàng đợi để đảm bảo các đỉnh gần hơn được xử lý trước các đỉnh xa hơn.

Bước giải (Ví dụ: A-B, A-C, B-D)

1. **Khởi tạo:** Queue = '[A]'. Visited = 'A'.
2. **Lần 1:** Lấy 'A' ra. In 'A'. Thêm các hàng xóm (B, C) vào queue. Queue = '[B, C]'. Visited = 'A, B, C'.
3. **Lần 2:** Lấy 'B' ra. In 'B'. Thêm hàng xóm (D) vào queue. Queue = '[C, D]'. Visited = 'A, B, C, D'.
4. **Lần 3:** Lấy 'C' ra. In 'C'. Không có hàng xóm mới. Queue = '[D]'.
5. **Lần 4:** Lấy 'D' ra. In 'D'. Không có hàng xóm mới. Queue = '[]'.
6. **Kết thúc.** Kết quả: A, B, C, D.

Bài toán 9: Breadth-first search (BFS) trên đa đồ thị

Thuật toán

Duyệt một đồ thị vô hướng có thể có nhiều cạnh nối giữa hai đỉnh (cạnh song song).

- Thuật toán BFS hoạt động hoàn toàn tương tự như trên đồ thị đơn. Logic của thuật toán chỉ quan tâm đến việc một đỉnh đã được thăm hay chưa, chứ không quan tâm có bao nhiêu cạnh nối đến nó.
- Vẫn sử dụng hàng đợi (Queue) và mảng `visited`.

Bài toán 10: Breadth-first search (BFS) trên đồ thị tổng quát

Thuật toán

Duyệt một đồ thị vô hướng có thể có cạnh song song và cạnh khuyên (cạnh nối một đỉnh với chính nó).

- Logic của BFS vẫn không thay đổi. Khi duyệt các đỉnh kề của một đỉnh u, nếu có cạnh khuyên (u, u), thuật toán sẽ thấy rằng đỉnh u đã được thăm và bỏ qua nó.
- Vẫn sử dụng hàng đợi (Queue) và mảng `visited`.

Bài toán 11: Depth-first search (DFS) trên đồ thị đơn

Thuật toán

Duyệt một đồ thị vô hướng, không có cạnh song song hay khuyên, bắt đầu từ một đỉnh cho trước.

- DFS đi sâu vào một nhánh cho đến khi không thể đi tiếp, sau đó quay lui (backtrack) và thử nhánh khác.
- **Cấu trúc dữ liệu:** Thuật toán thường được cài đặt bằng đệ quy, tận dụng ngăn xếp (stack) của hệ thống. Nó cũng cần một tập hợp `visited` để tránh các chu trình vô hạn.

Ý tưởng

Tưởng tượng như bạn đang đi trong một mê cung, DFS luôn đi thẳng cho đến khi gặp ngõ cụt, sau đó mới quay lại và thử một lối rẽ khác.

Bước giải (Ví dụ: A-B, A-C, B-D)

1. **Bắt đầu tại A:** Thăm A. Đi sâu vào hàng xóm đầu tiên là B.
2. **Tại B:** Thăm B. Đi sâu vào hàng xóm đầu tiên là D.
3. **Tại D:** Thăm D. Hết đường đi mới. Quay lui về B.
4. **Tại B:** Hết đường đi mới. Quay lui về A.
5. **Tại A:** Đi sâu vào hàng xóm tiếp theo là C.
6. **Tại C:** Thăm C. Hết đường đi mới. Quay lui về A.
7. **Kết thúc.** Kết quả: A, B, D, C.

Bài toán 12: Depth-first search (DFS) trên đa đồ thị

Thuật toán

Duyệt một đồ thị vô hướng có thể có cạnh song song.

- Logic của DFS không bị ảnh hưởng bởi sự tồn tại của các cạnh song song. Khi duyệt các đỉnh kề của một đỉnh, nó sẽ đi sâu vào đỉnh kề đầu tiên chưa thăm, bất kể có bao nhiêu cạnh nối đến đó.
- **Cấu trúc dữ liệu:** Vẫn sử dụng đệ quy và mảng `visited`.

Bài toán 13: Depth-first search (DFS) trên đồ thị tổng quát

Thuật toán

Duyệt một đồ thị vô hướng có thể có cạnh song song và cạnh khuyên.

- Logic của DFS vẫn không thay đổi. Một cạnh khuyên (u, u) sẽ được xem là một cạnh nối đến một đỉnh đã được thăm (u), do đó thuật toán sẽ không đi vào và không gây ra vòng lặp vô hạn.
- Vẫn sử dụng đệ quy và mảng `visited`.

Bài toán 14: Thuật toán Dijkstra trên đồ thị đơn

Thuật toán

Tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong một đồ thị vô hướng có trọng số không âm.

- Nó duy trì một tập các đỉnh đã có đường đi ngắn nhất được xác định. Ở mỗi bước, nó chọn đỉnh u chưa có trong tập này nhưng có khoảng cách tạm thời `dist[u]` nhỏ nhất. Sau đó, nó "kết nạp"u vào tập và cập nhật (relax) khoảng cách đến các đỉnh kề của u.
- **Cấu trúc dữ liệu:** Thuật toán sử dụng Min-Priority Queue để luôn lấy ra đỉnh có khoảng cách tạm thời nhỏ nhất một cách hiệu quả.

Ý tưởng

Đây là một thuật toán tham lam. Nó luôn ưu tiên khám phá đỉnh "gần nhất"mà nó chưa khám phá xong. Bằng cách này, nó đảm bảo rằng một khi khoảng cách đến một đỉnh đã được xác định, đó là khoảng cách ngắn nhất.

Bước giải (Ví dụ: A-(1)-B, B-(2)-C, A-(4)-C)

1. **Khởi tạo:** `dist[A]=0`, `dist[B]=inf`, `dist[C]=inf`. Priority Queue = `[(0,A)]`.
2. **Lần 1:** Lấy `(0,A)` ra. Cập nhật hàng xóm: `dist[B]=1`, `dist[C]=4`. Thêm `(1,B)` và `(4,C)` vào PQ.
3. **Lần 2:** Lấy `(1,B)` ra (vì $1 < 4$). Cập nhật hàng xóm của B: đường đi đến C qua B có chi phí `dist[B]` + `cost(B,C)` $= 1 + 2 = 3$. Vì $3 < 4$ (khoảng cách cũ đến C), cập nhật `dist[C]=3`. Thêm `(3,C)` vào PQ.
4. **Lần 3:** Lấy `(3,C)` ra. Không có gì cập nhật.
5. **Kết thúc.** Kết quả: `dist[A]=0`, `dist[B]=1`, `dist[C]=3`.

Bài toán 15: Thuật toán Dijkstra trên đa đồ thị

Thuật toán

Tìm đường đi ngắn nhất trên đa đồ thị có trọng số không âm.

- Thuật toán Dijkstra vẫn hoạt động chính xác. Nếu có hai cạnh song song nối giữa u và v với trọng số khác nhau, khi cập nhật khoảng cách đến v từ u, thuật toán sẽ tự động xem xét cả hai cạnh và chọn cạnh có trọng số nhỏ hơn để tính toán.
- Vẫn sử dụng Min-Priority Queue.

Bài toán 16: Thuật toán Dijkstra trên đồ thị tổng quát

Thuật toán

Tìm đường đi ngắn nhất trên đồ thị tổng quát có trọng số không âm.

- Một cạnh khuyên (u, u) với trọng số w sẽ được xem xét. Nếu `dist[u]` $+ w <$ `dist[u]`, nó sẽ cố gắng cập nhật khoảng cách đến chính nó. Tuy nhiên, vì điều kiện của Dijkstra là trọng số không âm, nên điều kiện này không bao giờ đúng.
- Vẫn sử dụng Min-Priority Queue.