Course Project: Combinatorics & Graph Theory

Vo Ngoc Tram Anh

July 26, 2025

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject

Contents

1 Integer Partition

1.1 Bài toán 1

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/1_IntegerPartition/Baitoan1

Mục tiêu: Cho 2 số nguyên dương n, k. Liệt kê tất cả các phân hoạch của n thành đúng k số nguyên dương sao cho:

- Tổng các phần tử bằng n
- Có đúng k phần tử
- Các phần tử không giảm (để đảm bảo các phân hoạch là phân biệt)

Số lượng các phân hoạch thoả mãn là $p_k(n)$.

Giải thuật: Gọi hàm đệ quy `genPartitions(cur, k, nRemain, part, res)`

- *cur*: chỉ số hiện tại trong phân hoạch (từ 0 đến $k-1$)
- *nRemain*: tổng còn lại cần chia
- *part*: mảng tạm lưu phân hoạch hiện tại
- *res*: danh sách chứa các phân hoạch hoàn chỉnh

Điều kiện dừng: Nếu đã điền đủ k phần tử ($cur = k$):

$$\text{Nếu } nRemain = 0 \Rightarrow \text{thêm phân hoạch vào danh sách kết quả}$$

Tại vị trí cur, ta xét các giá trị nguyên dương x sao cho:

$$x \in [\max(1, part[cur-1]),\ nRemain - (k - cur - 1)]$$

Ý nghĩa:

- Mỗi phần tử phải ≥ 1, nên $x \geq 1$
- Để đảm bảo dãy không giảm, ta cần $x \geq part[cur-1]$
- Để không bị thiếu tổng, cần chừa lại ít nhất $k - cur - 1$ đơn vị cho các phần tử còn lại (mỗi phần tử ít nhất là 1)

Với mỗi giá trị hợp lệ của x, thực hiện:

$$part[cur] \leftarrow x, \quad \texttt{genPartitions}(cur+1, k, nRemain - x, part, res)$$

Biểu diễn Ferrers và Ferrers chuyển vị: Với mỗi phân hoạch, ta biểu diễn Ferrers và Ferrers chuyển vị như sau

1. Ferrers: Cho phân hoạch $(a_1, a_2, \ldots, a_k)$. Ta vẽ k dòng, dòng thứ i có a_i dấu $*$.
2. Ferrers chuyển vị: Xét lại biểu đồ Ferrers theo cột. Dòng i của biểu đồ chuyển vị chứa số lượng dấu $*$ bằng số phần tử trong phân hoạch lớn hơn hoặc bằng i.

1.2 Bài toán 2

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/1_IntegerPartition/Baitoan2

Mục tiêu: Cho 2 số nguyên dương n, k. Cần tính:

- $p(n,k)$: Số phân hoạch của n mà tất cả các phần tử không lớn hơn k.
- $p_{\max}(n,k)$: Số phân hoạch của n mà phần tử lớn nhất bằng chính xác k.
- $p_k(n)$: Số phân hoạch của n thành đúng k phần tử.

Giải thuật:

1. Tính $p(n,k)$ Gọi $p(n,k)$ là số phân hoạch của n sử dụng các phần tử không lớn hơn k. Công thức quy hoạch động được sử dụng là:

$$p(n,k) = \begin{cases} 1 & \text{n} = 0 \\ 0 & \text{n} < 0 \text{ hoặc k} = 0 \\ p(n,k-1) + p(n-k,k) & \text{n, k} > 0 \end{cases}$$

   Giải thích:

   - $p(n,k-1)$: không sử dụng số k, chỉ sử dụng các số nhỏ hơn k.
   - $p(n-k,k)$: sử dụng ít nhất một số k, còn lại phân hoạch $n-k$ với phần tử không vượt quá k.

2. Tính $p_k(n)$ Gọi $pk(n,k)$ là số phân hoạch của n thành đúng k phần tử.

   Công thức quy hoạch động là:

$$pk(n,k) = \begin{cases} 1 & \text{n} = 0, \text{k} = 0 \\ 0 & \text{n} = 0, \text{k} > 0 \text{ hoặc k} = 0, \text{n} > 0 \\ pk(n-1,k-1) + pk(n-k,k) & \text{n, k} > 0 \end{cases}$$

   Giải thích:

   - $pk(n-1,k-1)$: thêm phần tử 1 vào phân hoạch $n-1$ có $k-1$ phần tử.
   - $pk(n-k,k)$: tăng mỗi phần tử trong phân hoạch $n-k$ thành k phần tử lên 1.

3. Tính $p_{\max}(n,k)$ Đây là số phân hoạch của n trong đó phần tử lớn nhất đúng bằng k.

   Ta có:

$$p_{\max}(n,k) = p(n,k) - p(n,k-1)$$

   Giải thích:

   - $p(n,k)$: tất cả phân hoạch với phần tử không vượt quá k.
   - $p(n,k-1)$: tất cả phân hoạch với phần tử không vượt quá $k-1$.
   - $p(n,k) - p(n,k-1)$ chính là những phân hoạch có phần tử lớn nhất là đúng k.

1.3 Bài toán 3

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/1_IntegerPartition/Baitoan3

Lý thuyết

- Phân hoạch tự liên hợp là phân hoạch có biểu đồ Ferrers đối xứng qua đường chéo chính.
- Phân hoạch thành các phần lẻ (odd parts) là phân hoạch mà mỗi phần là số lẻ.
- Số phân hoạch tự liên hợp của n đúng bằng số phân hoạch của n thành các phần lẻ phân biệt (Ref: Wolfram MathWorld).

Giải thuật

- Sinh phân hoạch tự liên hợp:
  - Kiểm tra tự liên hợp: Tạo phân hoạch liên hợp bằng cách đếm số phần tử lớn hơn hoặc bằng $i+1$ trong từng cột của biểu đồ Ferrers. Nếu phân hoạch gốc bằng phân hoạch liên hợp $\Rightarrow$ tự liên hợp.
  - Backtrack sinh phân hoạch: thêm vào danh sách kết quả nếu là tự liên hợp.
- Đếm và sinh phân hoạch thành các phần lẻ:
  - Gọi $dp[i]$ là số cách phân hoạch số i thành tổng các số lẻ, $dp[0] = 1$. Duyệt qua từng số lẻ odd từ 1 đến n, với mỗi odd: $dp[i] += dp[i - odd]$ (phân hoạch i bằng cách thêm odd vào phân hoạch $i - odd$, vẫn đảm bảo phân hoạch i thành các số lẻ)
  - Duyệt qua các số lẻ $odd = minodd, minodd + 2, ...$, với mỗi odd, nếu thêm vào phân hoạch hiện tại thì gọi đệ quy phân hoạch $n - odd$. Dừng khi $n == 0$ (tổng các số trong phân hoạch đang xét bằng đúng ban đầu). Sau khi đệ quy thì `pop_back()` để backtrack xét tiếp các phân hoạch khác.
- Sinh phân hoạch thành các phần lẻ phân biệt: Tương tự như sinh phân hoạch thành các phần lẻ, tuy nhiên ở mỗi bước ta luôn tăng $start$ lên $odd + 2$ để đảm bảo tính phân biệt.

2 Graph & Tree Traversing Problems

2.1 Bài toán 4

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan4

Graph

- From adjacency matrix: https://github.com/vntanh1406/Graph_SUM2025/blob/main/FinalProject/2_GraphAndTree/Baitoan4/FromAdjMat.cpp

  Input:
  - Một số nguyên n là số đỉnh trong đồ thị, đánh số từ 0 đến $n-1$.
  - Một biến $type$ xác định kiểu đồ thị:

$$type = \begin{cases} 0 & \text{đồ thị vô hướng} \\ 1 & \text{đồ thị có hướng} \end{cases}$$

- Ma trận kề kích thước $n \times n$, trong đó ô (u, v) chứa số nguyên k biểu diễn số lượng cạnh (hoặc trọng số tổng) từ đỉnh u đến đỉnh v.
- Nếu đồ thị vô hướng, ma trận kề sẽ đối xứng. Các cạnh khuyên thể hiện ở các phần tử trên đường chéo chính ($u = v$).
- Nếu đồ thị có hướng, ma trận kề không cần đối xứng.
- Ma trận kề cho phép đa cạnh (tức $adjMat[i][j] = k \Leftrightarrow$ từ i đến j có k cạnh).

Output:

- Edge List: Danh sách các cạnh được lưu dưới dạng các bộ ba (u, v, k):
  - u: đỉnh xuất phát
  - v: đỉnh đích
  - k: số lượng cạnh u đến v
- Adjacency List: Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
  - v: đỉnh kề với u
  - k: số lượng cạnh từ u đến v
- Adjacency Map Outgoing (og): Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
  - v: đỉnh có cạnh xuất phát từ u
  - k: số lượng cạnh từ u đến v
- Adjacency Map Incoming (ic): Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
  - v: đỉnh có cạnh nối đến u
  - k: số lượng cạnh từ v đến u

- From adjacency list: https://github.com/vntanh1406/Graph_SUM2025/blob/main/FinalProject/2_GraphAndTree/Baitoan4/FromAdjList.cpp

  Input:

  - Một số nguyên n là số đỉnh trong đồ thị, đánh số từ 0 đến $n - 1$.
  - Một biến *type* xác định kiểu đồ thị:

$$type = \begin{cases} 0 & \text{đồ thị vô hướng} \\ 1 & \text{đồ thị có hướng} \end{cases}$$

  - Danh sách kề gồm n dòng, mỗi dòng ứng với một đỉnh u:
    - Một số nguyên m là số đỉnh kề với u.
    - Sau đó là m cặp số (v, k):
      - v: đỉnh kề với u
      - k: số lượng cạnh từ u đến v
  - Dữ liệu cho phép cạnh khuyên là (u, u, k), và cho phép tồn tại nhiều hơn một cạnh giữa hai đỉnh.

  Output:

  - Edge List: Danh sách các cạnh được lưu dưới dạng các bộ ba (u, v, k):
    - u: đỉnh xuất phát
    - v: đỉnh đích

    - k: số lượng cạnh u đến v
  - Adjacency Matrix: Ma trận $n \times n$ với phần tử (u, v) chứa số nguyên k:
    - k: tổng số cạnh từ u đến v
    - Với đồ thị vô hướng, ma trận đối xứng; cạnh khuyên nằm trên đường chéo chính.
  - Adjacency Map Outgoing (og): Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
    - v: đỉnh có cạnh xuất phát từ u
    - k: số lượng cạnh từ u đến v
  - Adjacency Map Incoming (ic): Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
    - v: đỉnh có cạnh nối đến u
    - k: số lượng cạnh từ v đến u

- From edge list: https://github.com/vntanh1406/Graph_SUM2025/blob/main/FinalProject/2_GraphAndTree/Baitoan4/FromEdgeList.cpp

  Input:

  - Hai số nguyên n và m lần lượt là số đỉnh và số cặp đỉnh có cạnh nối, với các đỉnh được đánh số từ 0 đến $n - 1$.
  - Một biến *type* xác định kiểu đồ thị:

$$type = \begin{cases} 0 & \text{đồ thị vô hướng} \\ 1 & \text{đồ thị có hướng} \end{cases}$$

  - m dòng tiếp theo, mỗi dòng chứa bộ ba số nguyên (u, v, k):
    - u: đỉnh xuất phát của cạnh
    - v: đỉnh đích của cạnh
    - k: số lượng cạnh giữa u và v
  - Dữ liệu cho phép cạnh khuyên là (u, u, k), và cho phép tồn tại nhiều hơn một cạnh giữa hai đỉnh.

  Output:

  - Adjacency Matrix: Ma trận $n \times n$ với phần tử (u, v) chứa số nguyên k:
    - k: tổng số cạnh từ u đến v
    - Với đồ thị vô hướng, ma trận là đối xứng; các cạnh khuyên nằm trên đường chéo chính.
  - Adjacency List: Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
    - v: đỉnh kề với u
    - k: số lượng cạnh từ u đến v
  - Adjacency Map Outgoing (og): Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
    - v: đỉnh có cạnh xuất phát từ u
    - k: số lượng cạnh từ u đến v
  - Adjacency Map Incoming (ic): Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
    - v: đỉnh có cạnh nối đến u
    - k: số lượng cạnh từ v đến u

- From adjacency map: https://github.com/vntanh1406/Graph_SUM2025/blob/main/FinalProject/2_GraphAndTree/Baitoan4/FromAdjMap.cpp

  Input:

  - Một số nguyên n là số đỉnh trong đồ thị, đánh số từ 0 đến $n-1$.
  - Một biến *type* xác định kiểu đồ thị:

$$type = \begin{cases} 0 & \text{đồ thị vô hướng} \\ 1 & \text{đồ thị có hướng} \end{cases}$$

  - n dòng tiếp theo mô tả Outgoing Adjacency Map của từng đỉnh u:
    - Mỗi dòng bắt đầu với một số nguyên m là số đỉnh kề với u.
    - Sau đó là m cặp (v, k):
      - v: đỉnh kề với u
      - k: số lượng cạnh (hoặc tổng trọng số) từ u đến v
  - Dữ liệu cho phép cạnh khuyên và đa cạnh.

  Output:

  - Adjacency Map Incoming (ic): Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
    - v: đỉnh có cạnh nối đến u
    - k: số lượng cạnh từ v đến u
  - Edge List: Danh sách các cạnh được lưu dưới dạng các bộ ba (u, v, k):
    - u: đỉnh xuất phát
    - v: đỉnh đích
    - k: số lượng cạnh u đến v
  - Adjacency Matrix: Ma trận $n \times n$ với phần tử (u, v) chứa số nguyên k:
    - k: tổng số cạnh từ u đến v
    - Với đồ thị vô hướng, ma trận là đối xứng; các cạnh khuyên nằm trên đường chéo chính.
  - Adjacency List: Với mỗi đỉnh u, lưu danh sách các cặp (v, k):
    - v: đỉnh kề với u
    - k: số lượng cạnh từ u đến v

Tree

- From array of parents: https://github.com/vntanh1406/Graph_SUM2025/blob/main/FinalProject/2_GraphAndTree/Baitoan4/Tree_FromArrOfParents.cpp

  Input:

  - Một số nguyên n là số đỉnh trong cây, được đánh số từ 0 đến $n-1$.
  - Một mảng gồm n phần tử, trong đó phần tử thứ i chứa giá trị $parent[i]$ là cha của đỉnh i.
  - Nếu $parent[i] = -1$ thì đỉnh i là root.
  - Mỗi đỉnh (trừ root) có duy nhất một cha. Dữ liệu đảm bảo cấu trúc cây hợp lệ.

Output:

  - First-Child, Next-Sibling Representation: Mỗi đỉnh u có thể có:
    * $F[u]$: first child của u, hoặc `nil` nếu không có con nào.
    * $N[u]$: next sibling của u, hoặc `nil` nếu không có anh em kế.
  - Graph-Based Representation (Adjacency List):
    * Với mỗi đỉnh u, lưu danh sách các đỉnh con của u.
    * Dạng này tương tự danh sách kề trong đồ thị, nhưng đảm bảo không có chu trình và có đúng một root.

- From first-child & next-sibling: https://github.com/vntanh1406/Graph_SUM2025/blob/main/FinalProject/2_GraphAndTree/Baitoan4/Tree_FromFCNS.cpp

  Input:

  - Một số nguyên n là số đỉnh trong cây, được đánh số từ 0 đến $n-1$.
  - Hai mảng F và N, mỗi mảng gồm n phần tử:
    * $F[u]$: chỉ số first child của u, hoặc -1 nếu không có con nào.
    * $N[u]$: chỉ số next sibling của u, hoặc -1 nếu không có.

  Output:

  - Array of Parents:
    * Mỗi đỉnh i có $parent[i]$ là đỉnh cha của nó.
    * Nếu $parent[i] = -1$ thì i là root.
  - Graph-Based Representation (Adjacency List):
    * Với mỗi đỉnh u, lưu danh sách các đỉnh con của u.
    * Dạng này tương tự danh sách kề trong đồ thị, nhưng đảm bảo không có chu trình và có đúng một root.

- From graph-based representation (adjacency list): https://github.com/vntanh1406/Graph_SUM2025/blob/main/FinalProject/2_GraphAndTree/Baitoan4/Tree_FromGP.cpp

  Input:

  - Một số nguyên n là số đỉnh trong cây, được đánh số từ 0 đến $n-1$.
  - Danh sách kề gồm n dòng, mỗi dòng ứng với một đỉnh u:
    * Một số nguyên deg là số lượng đỉnh con của u.
    * Tiếp theo là deg số nguyên, lần lượt là các đỉnh con của u.
  - Dữ liệu đảm bảo là cấu trúc cây hợp lệ, tức không có chu trình và có đúng một root.

  Output:

  - Array of Parents:
    * Mỗi đỉnh i có $parent[i]$ là đỉnh cha của nó.
    * Nếu $parent[i] = -1$ thì i là root.
  - First-Child, Next-Sibling Representation: Mỗi đỉnh u có thể có:
    * $F[u]$: first child của u, hoặc `nil` nếu không có con nào.
    * $N[u]$: next sibling của u, hoặc `nil` nếu không có anh em kế.

2.2 Bài toán 5

2.2.1 Problem 1.1

- Complete graph K_n: Với n đỉnh, mỗi cặp đỉnh được nối với nhau bởi đúng một cạnh. Số cặp đỉnh là:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

  Vậy số cạnh của K_n là: $\dfrac{n(n-1)}{2}$

- Complete bipartite graph $K_{p,q}$: Gồm hai tập đỉnh rời có p và q đỉnh, mỗi đỉnh ở tập p nối với tất cả đỉnh ở tập q, nên:

$$\text{số cạnh} = p \cdot q$$

  Vậy số cạnh của $K_{p,q}$ là: $p \cdot q$

2.2.2 Problem 1.2

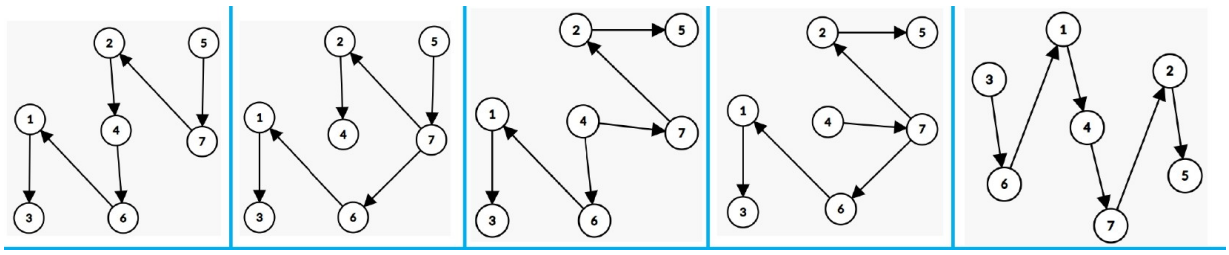
Một đồ thị là bipartite $\Longleftrightarrow$ tập đỉnh của nó có thể chia thành hai tập con rời nhau sao cho mọi cạnh đều nối một đỉnh ở tập này với một đỉnh ở tập kia. Vì vậy , bipartite graph không chứa chu trình nào có số cạnh lẻ.

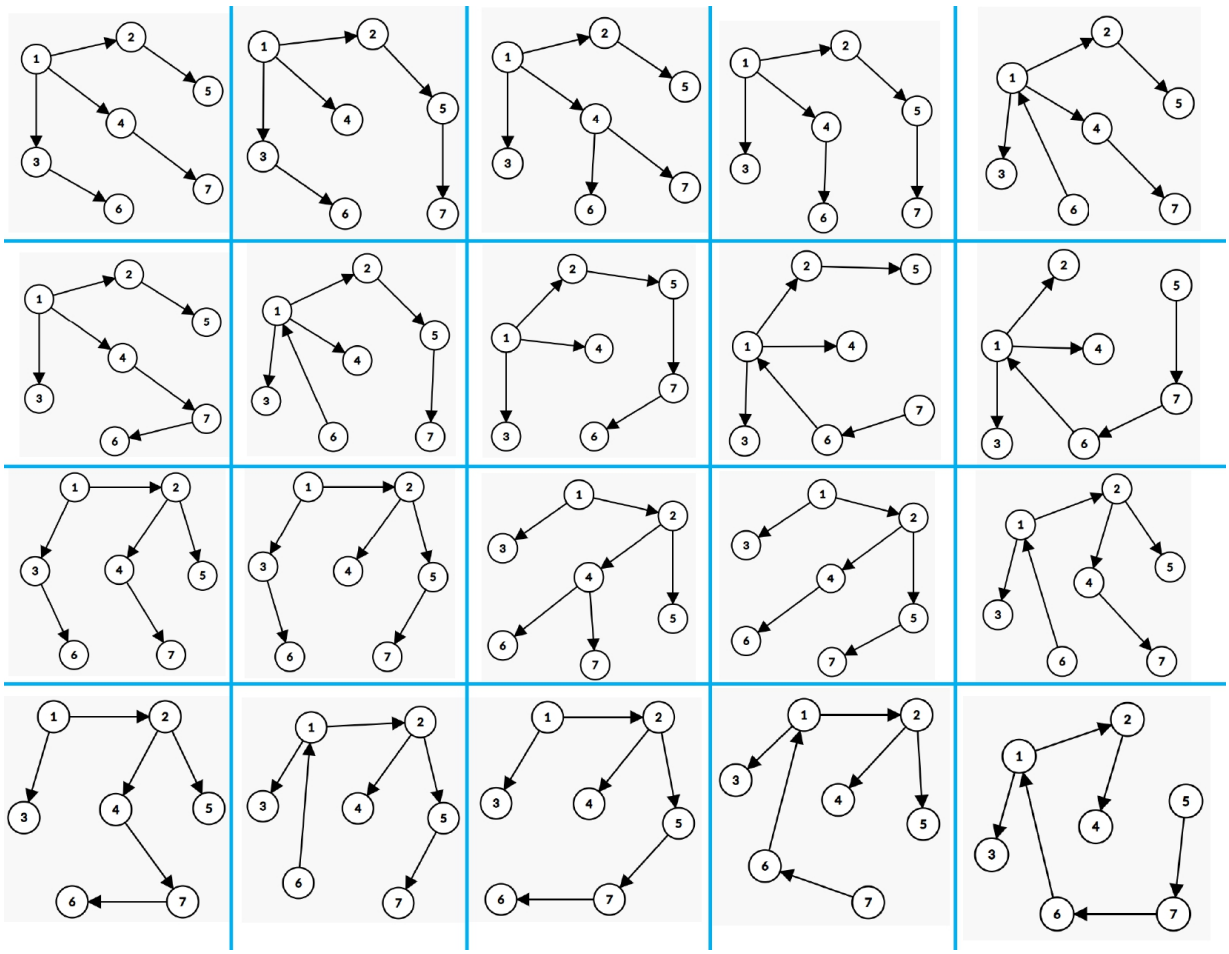
- Circle graph C_n gồm n đỉnh là một cycle n cạnh. Vì vậy, C_n là bipartite $\Longleftrightarrow$ n chẵn.
- Complete graph K_n:
  - K_1: chỉ có một đỉnh, không có cạnh $\Rightarrow$ bipartite.
  - K_2: chia hai đỉnh vào hai tập khác nhau $\Rightarrow$ bipartite.
  - K_3: có ba đỉnh và ba cạnh tạo thành chu trình lẻ (tam giác) $\Rightarrow$ không phải bipartite.
  - Với $n \geq 3$: tồn tại chu trình lẻ trong K_n $\Rightarrow$ không phải bipartite.

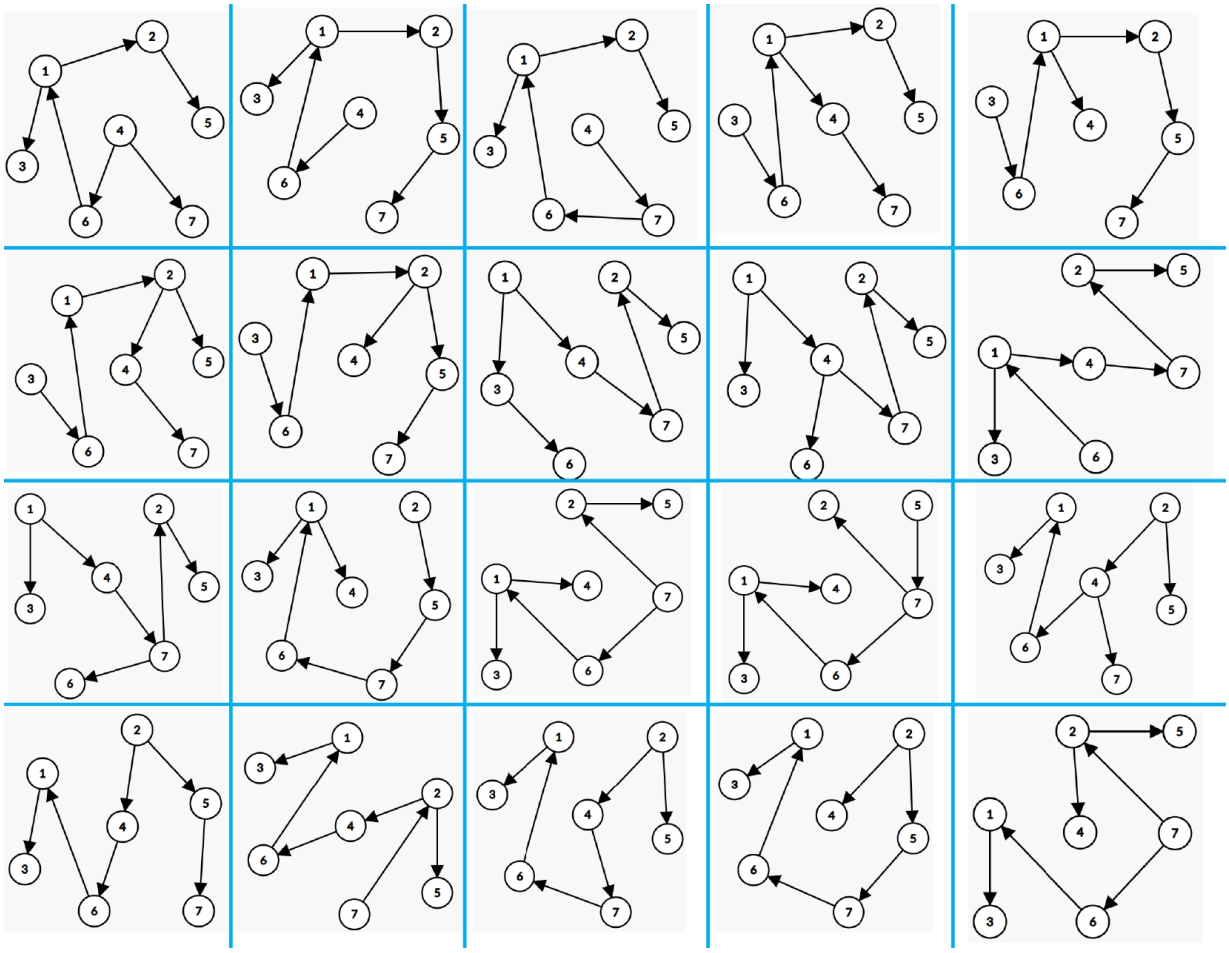
  Vậy K_n là bipartite graph khi và chỉ khi $n \leq 2$

2.2.3 Problem 1.3

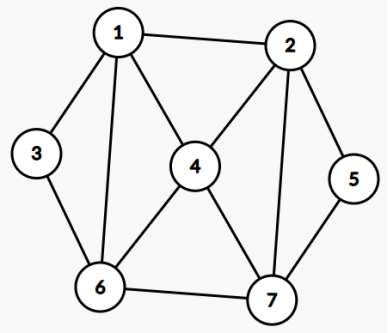
https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Prob13







Tổng số lượng spanning tree (với đồ thị vô hướng): (Ref: Wiki - Kirchhoff's Theorem)



- Xét ma trận bậc $D = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 \end{bmatrix}$,ma trận kề $A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$
- Xét ma trận Laplacian

$$L = D - A = \begin{bmatrix} 4 & -1 & -1 & -1 & 0 & -1 & 0 \\ -1 & 4 & 0 & -1 & -1 & 0 & -1 \\ -1 & 0 & 2 & 0 & 0 & -1 & 0 \\ -1 & -1 & 0 & 4 & 0 & -1 & -1 \\ 0 & -1 & 0 & 0 & 2 & 0 & -1 \\ -1 & 0 & -1 & -1 & 0 & 4 & -1 \\ 0 & -1 & 0 & -1 & -1 & -1 & 4 \end{bmatrix}$$

- Xóa 1 hàng và 1 cột của L (chọn xóa hàng 1, cột 1) được ma trận L_{minor}. Sau đó tính $det(L_{minor})$ ta được số spanning trees của đồ thị ban đầu là 288.

```
import numpy as np
L = np.array([
    [ 4, -1, -1, -1,  0, -1,  0],
    [-1,  4,  0, -1, -1,  0, -1],
    [-1,  0,  2,  0,  0, -1,  0],
    [-1, -1,  0,  4,  0, -1, -1],
    [ 0, -1,  0,  0,  2,  0, -1],
    [-1,  0, -1, -1,  0,  4, -1],
    [ 0, -1,  0, -1, -1, -1,  4]
])
L_minor = np.delete(np.delete(L, 0, axis=0), 0, axis=1)
print(round(np.linalg.det(L_minor)))
```

2.2.4 Problem 1.4

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Prob14

Với đồ thị được biểu diễn bằng ma trận kề, không tồn tại đối tượng cạnh riêng biệt. Do đó, mọi thao tác có liên quan đến cạnh (edge object) sẽ được chuyển đổi sang dạng thao tác với cặp đỉnh (v, w).

Ta xây dựng lớp Graph cho đồ thị n đỉnh như sau:

- Sử dụng ma trận kề $adj[n][n]$ với $adj[v][w] = 1$ nếu tồn tại cạnh nối từ v đến w.
- `del_edge`(v, w): Xóa cạnh nối từ v đến w bằng cách đặt $adj[v][w] := 0$.
- `edges`(): Duyệt toàn bộ ma trận kề, thêm mọi cặp đỉnh (v, w) mà $adj[v][w] = 1$ vào danh sách kết quả. Kết quả trả về là danh sách các cạnh dưới dạng các cặp đỉnh (v, w).
- `incoming`(v): Tìm các đỉnh u sao cho có cạnh nối từ u đến v, tức $adj[u][v] = 1$.Kết quả trả về là danh sách các cạnh dưới dạng các cặp đỉnh (u, v) (hoặc danh sách các đỉnh u).
- `outgoing`(v): Tìm các đỉnh w sao cho có cạnh nối từ v đến w, tức $adj[v][w] = 1$. Kết quả trả về là danh sách các cạnh dưới dạng các cặp đỉnh (v, w) (hoặc danh sách các đỉnh w).
- `source`(v, w): Với cạnh được xác định bằng cặp đỉnh (v, w) thì đỉnh nguồn chính là v nếu cạnh tồn tại.
- `target`(v, w): Với cạnh được xác định bằng cặp đỉnh (v, w) thì đỉnh đích chính là w nếu cạnh tồn tại.

2.2.5 Problem 1.5

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Prob15

Giải thích và ý tưởng: Cấu trúc *First-Child, Next-Sibling* biểu diễn mỗi nút trong cây bằng hai con trỏ:

- $first\_child$: trỏ tới con đầu tiên của nút.
- $next\_sibling$: trỏ tới anh em kế tiếp của nút.

Tuy nhiên, trong biểu diễn truyền thống, để lấy số con của nút v hoặc tập con của v, ta phải duyệt danh sách các anh em kế tiếp, mất thời gian $\mathcal{O}(k)$ với k là số con của v. Để đạt được phép toán $\mathcal{O}(1)$, ta mở rộng như sau:

- Dùng 1 biến riêng lưu *root* cho cây, cho phép lấy *root* trong $\mathcal{O}(1)$.
- Mỗi node v được mở rộng thêm trường $num\_children$ lưu số lượng con hiện tại của v. Việc cập nhật số con được thực hiện mỗi khi thêm hoặc xóa con.
- Giữ nguyên con trỏ $first\_child$ và $next\_sibling$: Giúp truy xuất tập con của nút v bằng cách truy cập con trỏ $first\_child$ (trả về danh sách con dạng liên kết). Việc trả về con trỏ này là $\mathcal{O}(1)$.

Kết quả:

- T.`root`(): Trả về biến *root* ngay lập tức, thời gian $\mathcal{O}(1)$.
- T.`number_of_children(v)`: Trả về $num\_children$ của node v mà không phải duyệt danh sách con, thời gian $\mathcal{O}(1)$.
- T.`children(v)`: Trả về con trỏ $first\_child$ của nút v, cho phép duyệt hoặc truy cập tập con theo liên kết anh em kế tiếp, thời gian trả về con trỏ là $\mathcal{O}(1)$.

2.2.6 Problem 1.6

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Prob16

Phân tích: Để kiểm tra xem một đồ thị có phải là cây hay không, ta dựa vào định nghĩa:

- Đồ thị phải liên thông.
- Không chứa chu trình.
- Có đúng $n-1$ cạnh với n đỉnh.
- Với đồ thị có hướng (cây có gốc - rooted directed tree) thì cần xét thêm:
  - Có đúng một đỉnh gốc (*root*) với $indegree = 0$.
  - Mọi đỉnh còn lại đều có $indegree = 1$.

Giải thuật: Độ phức tạp $\mathcal{O}(n+m)$, tuyến tính theo kích thước đồ thị (với n là số đỉnh, m là số cạnh).

1. Kiểm tra số cạnh: không phải $n-1$ thì không phải cây.
2. Kiểm tra tính liên thông và chu trình:
   - Với đồ thị vô hướng:
     - Duyệt DFS hoặc BFS từ một đỉnh bất kỳ.
     - Tổng số đỉnh đã thăm phải bằng n (đảm bảo liên thông).
     - Nếu trong quá trình duyệt gặp lại đỉnh đã thăm mà không phải là cha thì có chu trình.
   - Với đồ thị có hướng:
     - Kiểm tra tồn tại đúng một đỉnh gốc ($indegree = 0$), các đỉnh còn lại có $indegree = 1$.
     - Duyệt DFS từ đỉnh gốc.
     - Nếu tồn tại chu trình (phát hiện trong DFS bằng phương pháp 3 trạng thái), hoặc không thăm hết các đỉnh thì không phải cây.

2.2.7 Exercise 1.1

Định dạng DIMACS: Dùng cho đồ thị vô hướng.

- Dòng định nghĩa vấn đề: `p edge n m` với n là số đỉnh, m là số cạnh.
- m dòng miêu tả các cạnh: `e i j`, với i, j là chỉ số các đỉnh (từ 1 đến n).
- Các dòng bắt đầu bằng `c` là dòng chú thích, có thể bỏ qua.

Yêu cầu: Cài đặt hai hàm.

- `read_dimacs()` để đọc dữ liệu đồ thị từ định dạng DIMACS và lưu vào cấu trúc dữ liệu.
- `write_dimacs()` để xuất đồ thị hiện tại ra định dạng DIMACS.

Giải pháp :

- Khi đọc bỏ qua các dòng comment, lấy số đỉnh và số cạnh từ dòng `p edge n m`.
- Lưu các cạnh vào danh sách kề, chuyển chỉ số đỉnh từ 1-based sang 0-based để thuận tiện xử lý trong code.
- Khi ghi, in ra dòng `p edge n m` và các dòng cạnh `e i j`, chuyển lại sang 1-based index.

2.2.8 Exercise 1.2

Định dạng SGB: Định dạng Stanford GraphBase (SGB) biểu diễn đồ thị (có hướng hoặc vô hướng) theo cấu trúc:

- Dòng 1: `* GraphBase graph (utiltypes..., nV, mA)`
  Với nV số đỉnh, mA số cung (arcs).
- Dòng 2: Chuỗi nhận diện đồ thị (tên hoặc mô tả).
- Dòng 3: `* Vertices` đánh dấu bắt đầu phần đỉnh.
- Dòng 4 đến $nV + 3$: mỗi dòng mô tả một đỉnh: `label, Ai, 0, 0`
  Trong đó:
  - `label` là nhãn đỉnh (chuỗi).
  - `Ai` là chỉ số cạnh đầu tiên đi ra từ đỉnh, trong khoảng $[0, mA - 1]$. Nếu đỉnh không có cạnh đi ra, `Ai` $= 0$.
  - Hai số 0 cuối dòng là các trường không dùng.
- Dòng $nV + 4$: `* Arcs` đánh dấu bắt đầu phần cung.
- Dòng $nV + 5$ đến $nV + mA + 4$: mỗi dòng mô tả một cung: `Vj, Ai, label, 0`
  Trong đó:
  - `Vj` là chỉ số đỉnh đích (từ 0 đến $nV - 1$).
  - `Ai` là chỉ số cạnh kế tiếp cùng xuất phát từ đỉnh nguồn; nếu đây là cung cuối cùng từ đỉnh nguồn thì `Ai` $= 0$.
  - `label` là nhãn cung (thường là số nguyên).
  - Số 0 cuối dòng là trường không dùng.
- Dòng cuối: `* Checksum ...` để kiểm tra tính toàn vẹn định dạng.

Yêu cầu: Cài đặt hai hàm

- `read_sgb()` để đọc dữ liệu đồ thị từ file văn bản theo định dạng SGB chuẩn, bao gồm các phần:
  - Dòng đầu: `* GraphBase graph (utiltypes..., nV, mA)`
  - Dòng nhận diện đồ thị
  - Dòng `* Vertices` và n dòng mô tả đỉnh: `label, Ai, 0, 0`
  - Dòng `* Arcs` và m dòng mô tả cung: `Vj, Ai, label, 0`
  - Dòng `* Checksum ...`
- `write_sgb()` để ghi dữ liệu đồ thị ra file văn bản theo đúng cấu trúc định dạng SGB chuẩn như trên.

Giải pháp:

- Trong hàm `read_sgb()`, tiến hành đọc tuần tự từng dòng từ file hoặc luồng nhập, phân tích cú pháp các phần theo định dạng đã mô tả (dòng đầu, đỉnh, cung, checksum).
- Lưu thông tin đỉnh và cung vào cấu trúc dữ liệu thích hợp, ví dụ: danh sách đỉnh với nhãn và chỉ số cạnh đầu tiên, danh sách cung với đỉnh đích, chỉ số cạnh kế tiếp cùng nguồn, và nhãn cạnh.
- Trong hàm `write_sgb()`, xuất lần lượt các phần gồm: dòng tiêu đề, dòng nhận diện đồ thị, phần `* Vertices` với mô tả đỉnh, phần `* Arcs` với mô tả cung, và cuối cùng dòng `* Checksum` để hoàn thiện định dạng.

2.2.9 Exercise 1.3

Mục tiêu: Sử dụng bộ 32 phép toán trừu tượng trên đồ thị (phần 1.3) để xây dựng các đồ thị tiêu chuẩn gồm:

- Path graph P_n với n đỉnh.
- Circle graph C_n với n đỉnh.
- Wheel graph W_n với n đỉnh.

Ý tưởng giải thuật:

- Khởi tạo đồ thị rỗng.
- Dùng phép `new_vertex()` thêm lần lượt n đỉnh.
- Với P_n, thêm cạnh nối liền các đỉnh liên tiếp: $\forall i = 0 \dots n-2$, `new_edge`(v_i, v_{i+1}).
- Với C_n, xây dựng P_n rồi thêm cạnh nối từ đỉnh cuối về đỉnh đầu: `new_edge`(v_{n-1}, v_0).
- Với W_n, thêm đỉnh trung tâm v_0, tạo vòng tròn trên các đỉnh còn lại $v_1 \dots v_{n-1}$ như C_{n-1}, rồi nối đỉnh trung tâm tới tất cả các đỉnh trên vòng tròn.

2.2.10 Exercise 1.4

Mục tiêu: Sử dụng bộ 32 phép toán trừu tượng trên đồ thị (phần 1.3) để xây dựng các đồ thị sau:

- Complete graph K_n gồm n đỉnh, trong đó mỗi cặp đỉnh đều có cạnh nối.
- Complete bipartite graph $K_{p,q}$ gồm hai tập đỉnh rời nhau P và Q với kích thước p và q, trong đó mỗi đỉnh của tập P nối với mọi đỉnh của tập Q.

Ý tưởng giải thuật:

- Khởi tạo đồ thị rỗng.
- Dùng `new_vertex()` thêm các đỉnh cần thiết:
  - Với K_n, thêm n đỉnh.
  - Với $K_{p,q}$, thêm $p+q$ đỉnh, chia làm 2 tập P và Q.
- Với K_n, duyệt tất cả các cặp đỉnh (v_i, v_j) với $0 \le i < j < n$, thêm cạnh `new_edge`(v_i, v_j) và nếu là đồ thị vô hướng, thêm cả `new_edge`(v_j, v_i) nếu cần thiết.
- Với $K_{p,q}$, với mỗi đỉnh $u \in P$ và $v \in Q$, thêm cạnh `new_edge`(u, v).

2.2.11 Exercise 1.5

Mục tiêu: Triển khai lớp *Graph* biểu diễn đồ thị sử dụng ma trận kề mở rộng (theo Promblem 1.4), sử dụng Python lists để lưu trữ và quản lý các đỉnh, cạnh theo số thứ tự nội bộ (internal numbering).

Mô tả và yêu cầu:

- Sử dụng ma trận kề adj kích thước $n \times n$, với $adj[v][w] = 1$ nếu tồn tại cạnh từ đỉnh v đến đỉnh w, ngược lại 0.
- Cài đặt các phép toán chính:
  - `del_edge(v, w)`: xóa cạnh nối từ v đến w.
  - `edges()`: trả về danh sách các cạnh dưới dạng cặp đỉnh (v, w).
  - `incoming(v)`: trả về danh sách các đỉnh có cạnh đi vào v.
  - `outgoing(v)`: trả về danh sách các đỉnh có cạnh đi ra từ v.
  - `source(v, w)`: trả về đỉnh nguồn v của cạnh (v, w) nếu tồn tại, ngược lại `None`.
  - `target(v, w)`: trả về đỉnh đích w của cạnh (v, w) nếu tồn tại, ngược lại `None`.
- Quản lý đỉnh với số thứ tự từ 0 đến $n - 1$ (internal numbering).
- Cài đặt: Problem 1.4

2.2.12 Exercise 1.6

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Ex16

Mục tiêu: Liệt kê tất cả các perfect matching trong complete bipartite graph $K_{p,q}$.

Phân tích:

- Để tồn tại perfect matching, hai tập đỉnh phải có cùng số lượng: $p = q$.
- Mỗi perfect matching tương ứng với một hoán vị của tập Q theo thứ tự tập P.
- Do đó, nhiệm vụ liệt kê các perfect matching tương đương với việc liệt kê tất cả các hoán vị của tập Q.

Giải thuật: Sử dụng sinh hoán vị (permutation) của tập Q. Với mỗi hoán vị π, tạo ra tập cạnh $\{(P_i, Q_{\pi(i)})\}_{i=1}^{p}$.

2.2.13 Exercise 1.7

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Ex17

Mục tiêu: Sử dụng bộ 13 phép toán trừu tượng trên cây để tạo complete binary tree n node.

Ý tưởng:

- Gán số thứ tự các node từ 0 đến $n-1$ theo thứ tự BFS.
- 0 là root.
- Với node i, nếu tồn tại, node con trái là $2i+1$, node con phải là $2i+2$.
- Thêm cạnh từ node cha i đến các con trái và phải nếu chỉ số con nhỏ hơn n.

Thuật toán:

1. Tạo mảng $nodes[0 \dots n-1]$.
2. Với $i = 0 \to n-1$, thực hiện: $nodes[i] = T.\texttt{new\_node()}$
3. Gán node gốc: $T.\texttt{root()} = nodes[0]$.
4. Với $i = 0 \to \lfloor \frac{n-2}{2} \rfloor$:
   - Nếu $2i+1 < n$: $T.\texttt{children}(nodes[i]).\texttt{append}(nodes[2i+1])$
   - Nếu $2i+2 < n$: $T.\texttt{children}(nodes[i]).\texttt{append}(nodes[2i+2])$

2.2.14 Exercise 1.8

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Ex18

Mục tiêu: Sử dụng bộ 13 phép toán trừu tượng trên cây để tạo cây ngẫu nhiên n node.

Ý tưởng: Dựa trên một tính chất cơ bản: Mỗi cây có n đỉnh thì có đúng $n-1$ cạnh. Để sinh một cây ngẫu nhiên với n đỉnh, ta gán lần lượt mỗi đỉnh từ 1 đến $n-1$ làm con của một đỉnh đã được tạo trước đó (từ 0 đến $i-1$). Điều này đảm bảo không tạo chu trình và tạo nên một cây hợp lệ.

Thuật toán:

1. Khởi tạo cây rỗng T và gán $T.\texttt{root()} \leftarrow 0$.
2. Với mỗi i từ 1 đến $n-1$:
   - Gọi $T.\texttt{number\_of\_nodes()} = i$.
   - Sinh ngẫu nhiên một node $p \in \{0, 1, \dots, i-1\}$.
   - Thêm node i làm con của node p thông qua phép toán trừu tượng: $T.\texttt{children}(p).\texttt{append}(i)$.
   - Gán ngược lại: $T.\texttt{parent}(i) \leftarrow p$.

Độ phức tạp:

- Thời gian: Thuật toán lặp qua từng node từ 1 đến $n-1$, mỗi bước thực hiện các thao tác như sinh ngẫu nhiên, thêm con, gán cha đều là $\mathcal{O}(1)$. Do đó tổng thời gian là $\mathcal{O}(n)$.
- Bộ nhớ: Cần lưu thông tin cha con cho tất cả n node, tổng cộng bộ nhớ dùng là $\mathcal{O}(n)$.

2.2.15 Exercise 1.9

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Ex19

Mô tả bài toán: Cho cây T được lưu trữ dưới dạng mảng cha *parent*, trong đó $parent[v]$ là cha của node v (nếu v là *root* thì $parent[v]$ là `None` hoặc giá trị đặc biệt). Yêu cầu cài đặt phép toán `T.previous_sibling(v)` trả về previous sibling của v trong cây, nếu tồn tại; ngược lại trả về `None`.

Ý tưởng:

- Tìm node cha $p = parent[v]$.
- Tìm tất cả các node con của p, tức các node u sao cho $parent[u] = p$.
- Xác định vị trí của v trong danh sách các con của p.
- Nếu v không phải là node con đầu tiên, trả về node đứng ngay trước nó trong danh sách, ngược lại trả về `None`.

2.2.16 Exercise 1.10

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan5/Ex110

Mục tiêu: Xây dựng cấu trúc cây theo mô hình *First-Child, Next-Sibling* mở rộng, cho phép truy cập số con của mỗi node trong thời gian $\mathcal{O}(1)$ và duyệt tập con theo danh sách liên kết.

Ý tưởng:

- Mỗi node được biểu diễn bằng ba trường chính:
  - $first\_child$: trỏ đến con đầu tiên của node (hoặc `None` nếu không có con).
  - $next\_sibling$: trỏ đến anh em kế tiếp của node (hoặc `None` nếu là anh em cuối).
  - $num\_children$: lưu số lượng con hiện tại của node, cập nhật khi thêm hoặc xóa con.
- Bổ sung trường *parent* để thuận tiện trong các phép toán với cây.
- Cây có biến *root* lưu chỉ số nút gốc, cho phép truy cập root nhanh chóng.
- Việc thêm con được thực hiện bằng cách chèn node con vào đầu danh sách con của node cha, cập nhật liên kết và số con.
- Duyệt con của một nút sẽ bắt đầu từ $first\_child$ và theo chuỗi $next\_sibling$ đến hết.

2.3 Bài toán 6

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan6

Các giải thuật tính Tree Edit Distance giữa hai cây có thứ tự (ordered trees), với giả định:

- Các thao tác chỉnh sửa bao gồm xóa, thêm và thay thế node.
- Chi phí của các thao tác được định nghĩa qua hàm γ:
  - Với γ_{sub1}: Chi phí thay thế node khác nhau bằng 1, tương đương tính tổng số thao tác sửa đổi (thêm, xóa và thay đổi đều có chi phí 1).
  - Với γ_{sub0}: Chi phí thay thế node khác nhau bằng 0, nghĩa là chỉ tính chi phí cho xóa và thêm
- Chỉ cho phép thực hiện thao tác trên các nút lá.
- Giữ nguyên thứ tự con trong mỗi node.

Backtracking

- Nếu một trong hai node là `null`, tính chi phí xóa hoặc thêm toàn bộ cây con còn lại.
- Nếu cả hai node đều không rỗng:
  - Tính chi phí thay thế giữa hai node gốc theo γ.
  - Xác định cách ghép các children của hai node sao cho chi phí tổng là nhỏ nhất, bằng cách thử tất cả khả năng:
    - Ghép con i của cây 1 với con j của cây 2 (substitute).
    - Xóa con i của cây 1.
    - Thêm con j của cây 2.
- Chi phí tối thiểu trong tất cả các trường hợp được chọn làm kết quả cho node hiện tại.

Branch and Bound

Thuật toán Branch and Bound là một cải tiến của phương pháp backtracking nhằm giảm số lượng trường hợp phải xét khi tính Tree Edit Distance. Ý tưởng chính là sử dụng một lower bound (cận dưới) cho chi phí còn lại để cắt tỉa (prune) các nhánh mà chắc chắn không thể cho kết quả tối ưu hơn kết quả hiện tại.

- Với mỗi cặp node (t_1, t_2), tính chi phí thay thế node gốc theo hàm γ.
- Xét dãy con children của t_1 và t_2, tương tự như backtracking, thử các lựa chọn:
  1. Substitute child $c1[i]$ với $c2[j]$.
  2. Delete child $c1[i]$.
  3. Insert child $c2[j]$.
- Tại mỗi bước đệ quy, tính lower bound chi phí tối thiểu còn lại từ vị trí (i, j) bằng hàm `computeLowerBound`.
- Nếu tổng chi phí hiện tại cộng với lower bound này lớn hơn hoặc bằng chi phí tối thiểu tìm được (`minCost`), nhánh đó bị cắt tỉa (không tiếp tục mở rộng).

Hàm `computeLowerBound`: Ước lượng chi phí tối thiểu còn lại cần thiết để xử lý các con còn lại của hai node hiện tại:

- Nếu một trong hai dãy con rỗng, lower bound là tổng chi phí xóa hoặc thêm tất cả các node còn lại.
- Nếu cả hai đều không rỗng, lower bound là độ chênh lệch số lượng node con còn lại (đơn giản nhưng có hiệu quả cắt tỉa).

Divide and Conquer

Thuật toán sử dụng chiến lược quy hoạch động để giải bài toán khoảng cách chỉnh sửa cây, bằng cách xử lý việc chuyển đổi cây con của mỗi node sang cây con tương ứng trong cây kia. Ý tưởng chính là chia bài toán lớn (so sánh hai cây) thành các bài toán nhỏ hơn (so sánh từng cây con), giải từng bài toán nhỏ này một cách đệ quy, sau đó tổng hợp kết quả để xây dựng lời giải tổng thể. Cụ thể:

- Nếu cả hai node đều rỗng, chi phí là 0.
- Nếu một trong hai node rỗng, thì chi phí bằng số thao tác chèn hoặc xóa toàn bộ cây còn lại (được tính đệ quy).
- Nếu cả hai node đều tồn tại, trước tiên cần so sánh hai node gốc (chi phí thay thế), sau đó xử lý phần cây con:
  - Giả sử node t_1 có n con và node t_2 có m con.
  - Duyệt toàn bộ cách ánh xạ có thứ tự giữa các cây con của t_1 và t_2.
  - Dùng bảng quy hoạch động $dp[i][j]$ để tính chi phí thấp nhất để biến i con đầu tiên của t_1 thành j con đầu tiên của t_2.
  - Công thức quy hoạch động:

$$dp[i][j] = \min \begin{cases} dp[i-1][j-1] + \text{cost}(c_1[i-1], c_2[j-1]) & \text{(thay thế)} \\ dp[i-1][j] + \text{cost}(c_1[i-1], \varnothing) & \text{(xóa)} \\ dp[i][j-1] + \text{cost}(\varnothing, c_2[j-1]) & \text{(thêm)} \end{cases}$$

    Giải thích:

    * Thay thế: biến cây con $c_1[i-1]$ thành $c_2[j-1]$ rồi tính tiếp phần còn lại.
    * Xóa: xóa cây con $c_1[i-1]$, giữ nguyên c_2.
    * Thêm: thêm cây con $c_2[j-1]$, giữ nguyên c_1.
- Kết quả cuối cùng là tổng chi phí của root và chi phí quy hoạch động giữa các cây con.

Dynamic Programming

- Ý tưởng tương tự như phương pháp chia để trị: tính chi phí biến đổi giữa hai cây bằng cách so sánh node gốc và xử lý các cây con.
- Tuy nhiên, khác với chia để trị, thuật toán này sử dụng bảng nhớ (memoization) để lưu trữ kết quả trung gian, tránh việc gọi lại các cặp cây con giống nhau nhiều lần.
- Thuật toán sử dụng memoization thông qua một bảng nhớ (gọi là `memo`) để lưu kết quả của các phép tính $dp(t_1, t_2)$.

- Cụ thể, trước khi tính toán chi phí biến đổi giữa hai node t_1 và t_2, thuật toán sẽ kiểm tra trong `memo`:

$$\text{nếu } dp(t_1, t_2) \text{ đã được tính trước đó thì trả về kết quả trong } \texttt{memo}(t_1, t_2)$$

- Nếu chưa có trong `memo`, thuật toán sẽ tính chi phí theo công thức, sau đó lưu kết quả vào `memo` trước khi trả về:

$$\texttt{memo}(t_1, t_2) \leftarrow dp(t_1, t_2)$$

- Việc sử dụng `memo` giúp tránh gọi lại đệ quy cho cùng một cặp node nhiều lần, giảm thiểu thời gian tính toán đáng kể.
- Hàm quy hoạch động $dp(t_1, t_2)$ được định nghĩa là chi phí thấp nhất để biến cây t_1 thành cây t_2, với chi phí thay thế được định nghĩa qua hàm γ.
- Nếu một trong hai cây rỗng, thì chi phí là tổng chi phí thêm hoặc xóa toàn bộ cây còn lại (đệ quy trên tất cả các node con).
- Nếu cả hai đều tồn tại:
  - Tính chi phí thay thế giữa hai node gốc.
  - Gọi $c_1[0 \dots n{-}1]$ là các con của t_1, và $c_2[0 \dots m{-}1]$ là các con của t_2.
  - Tạo bảng phụ $subdp[i][j]$ để lưu chi phí thấp nhất để biến i con đầu tiên của t_1 thành j con đầu tiên của t_2.
  - Công thức quy hoạch động:

$$subdp[i][j] = \min \begin{cases} subdp[i{-}1][j{-}1] + dp(c_1[i{-}1], c_2[j{-}1]) & \text{(thay thế)} \\ subdp[i{-}1][j] + dp(c_1[i{-}1], \varnothing) & \text{(xóa)} \\ subdp[i][j{-}1] + dp(\varnothing, c_2[j{-}1]) & \text{(thêm)} \end{cases}$$

  - Sau đó, tổng chi phí sẽ bằng:

$$dp(t_1, t_2) = \gamma(t_1, t_2) + subdp[n][m]$$

- So sánh với Divide and Conquer:
  - Cả hai đều có cùng công thức tính toán và chiến lược ánh xạ các cây con.
  - Tuy nhiên, phương pháp chia để trị gọi đệ quy trùng lặp nhiều lần và không lưu kết quả trung gian, dẫn đến thời gian chạy cao. Điều này khắc phục bằng việc sử dụng memoization trong thuật toán quy hoạch động.

2.4 Bài toán 7

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan7

- Preorder: Thuật toán bắt đầu từ node gốc, thực hiện in giá trị của node này trước tiên. Sau đó, đệ quy lần lượt duyệt từng cây con theo thứ tự từ trái sang phải. Nói cách khác, với mỗi node, ta xử lý node đó trước rồi mới đi vào các cây con của nó.
- Postorder: Thuật toán đệ quy đi sâu vào từng cây con trước, lần lượt từ trái sang phải. Sau khi tất cả các cây con của node được duyệt xong, ta mới in node gốc.

- Top-down Level Order: Thuật toán dùng BFS, bắt đầu từ node gốc, đưa node này vào một hàng đợi (queue). Sau đó, lần lượt lấy node đầu hàng đợi ra, in giá trị node và đưa toàn bộ các cây con của node đó vào cuối hàng đợi. Quá trình lặp lại cho đến khi hàng đợi rỗng, tức là tất cả các node đã được duyệt theo thứ tự từng tầng từ trên xuống dưới.
- Bottom-up Level Order: Thuật toán tương tự Top-down Level Order sử dụng BFS, nhưng thay vì in giá trị node ngay khi duyệt, ta lưu lại các node theo từng tầng vào một mảng. Sau khi duyệt hết cây, ta đảo ngược thứ tự các tầng này rồi in ra, tức là in tầng thấp nhất trước, tầng cao nhất sau. Kết quả là duyệt cây theo từng tầng nhưng theo chiều từ dưới lên trên.

2.5 Bài toán 8 - 9 - 10

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan8_9_10

Ý tưởng chính: Sử dụng một hàng đợi để lưu trữ các đỉnh chờ được khám phá :

- Bắt đầu từ đỉnh nguồn, đánh dấu đỉnh này là đã thăm và đẩy nó vào hàng đợi.
- Lặp lại: lấy đỉnh đầu tiên ra khỏi hàng đợi, in đỉnh đó.
- Với mỗi đỉnh kề chưa được thăm của đỉnh hiện tại, đánh dấu nó đã thăm và đẩy vào hàng đợi.
- Quá trình tiếp tục cho đến khi hàng đợi rỗng.

Đặc điểm thuật toán:

- BFS khám phá các đỉnh theo thứ tự số cạnh từ đỉnh nguồn, tức là duyệt theo tầng.
- Đảm bảo tất cả các đỉnh trong cùng một tầng được xử lý trước khi chuyển sang tầng kế tiếp.
- BFS có thể áp dụng cho cả đồ thị vô hướng và đồ thị có hướng.
- Dùng được cho cả simple graph, multigraph và general graph vì thuật toán không phụ thuộc vào việc có cạnh khuyên hay có bao nhiêu cạnh song song, mà chỉ dựa trên các cạnh liền kề.

Pseudocode:

Input: Graph $G = (V, E)$, start vertex s
Initialize $visited[v] \leftarrow$ false $\quad \forall v \in V$
Initialize queue Q
$visited[s] \leftarrow$ true
Q.push(s)
while (Q is not empty) do
  $u \leftarrow Q$.front(); $\quad Q$.pop()
  Process u
  for each $v \in adj[u]$ such that $visited[v] =$ false do
    $visited[v] \leftarrow$ true
    Q.push(v)

Xử lý đồ thị không liên thông: Để đảm bảo duyệt hết tất cả các đỉnh trong đồ thị (có thể gồm nhiều thành phần liên thông), ta chạy BFS lần lượt từ từng đỉnh chưa được thăm.

2.6 Bài toán 11 - 12 - 13

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/2_GraphAndTree/Baitoan11_12_13

Ý tưởng chính: Sử dụng đệ quy (hoặc ngăn xếp) để thăm sâu từng nhánh của đồ thị:

- Bắt đầu từ đỉnh nguồn, đánh dấu đỉnh này là đã thăm và xử lý.
- Đệ quy thăm từng đỉnh kề chưa được thăm của đỉnh hiện tại.
- Quá trình tiếp tục cho đến khi không còn đỉnh kề nào chưa thăm.

Đặc điểm thuật toán:

- DFS thăm các đỉnh theo chiều sâu, đi hết một nhánh đến tận cùng trước khi quay lui.
- Thuật toán phù hợp cho cả đồ thị vô hướng và đồ thị có hướng.
- Dùng được cho simple graph, multigraph và general graph vì thuật toán chỉ dựa vào danh sách kề, không phụ thuộc vào dạng cạnh hay số lượng cạnh song song.

Pseudocode:

Input: Graph $G = (V, E)$, start vertex u
Procedure $DFS(u)$:
  $visited[u] \leftarrow$ true
  Process u
  for each $v \in adj[u]$ such that $visited[v] =$ false do
    $DFS(v)$
Initialize $visited[v] \leftarrow$ false $\forall v \in V$
for each $u \in V$ do
  if $visited[u] =$ false then
    $DFS(u)$

Xử lý đồ thị không liên thông: Để đảm bảo duyệt hết tất cả các đỉnh trong đồ thị (có thể gồm nhiều thành phần liên thông), ta chạy DFS lần lượt từ từng đỉnh chưa được thăm.

3 Shortest Path Problems on Graphs

3.1 Bài toán 14 - 15 - 16

https://github.com/vntanh1406/Graph_SUM2025/tree/main/FinalProject/3_ShortestPath/Baitoan14_15_16

Ý tưởng chính: Thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh nguồn đến tất cả các đỉnh còn lại trong đồ thị có trọng số không âm bằng cách:

- Khởi tạo khoảng cách từ đỉnh nguồn đến tất cả các đỉnh khác là vô cực, khoảng cách đến chính nó là 0.
- Dùng một tập các đỉnh chưa được xử lý, mỗi lần chọn đỉnh có khoảng cách nhỏ nhất chưa xử lý để "mở rộng".
- Cập nhật khoảng cách tới các đỉnh kề của đỉnh vừa chọn nếu tìm được đường đi ngắn hơn.
- Lặp lại đến khi tất cả đỉnh được xử lý hoặc không còn đỉnh nào có thể cải thiện khoảng cách.

Đặc điểm thuật toán:

- Áp dụng được cho cả simple graph, multigraph và general graph vì thuật toán chỉ dựa vào danh sách kề và trọng số các cạnh.
- Cạnh phải có trọng số không âm (non-negative weights).
- Có thể áp dụng cho cả đồ thị vô hướng và có hướng.
- Có nhiều cách triển khai, ví dụ:
  - Triển khai truyền thống dùng mảng để tìm đỉnh khoảng cách nhỏ nhất trong tập chưa xử lý (độ phức tạp $O(n^2)$).
  - Triển khai dùng priority queue (hàng đợi ưu tiên) giúp tăng tốc độ, đạt độ phức tạp $O(m \log n)$ với n đỉnh và m cạnh.

Pseudocode (priority queue):

Input: Graph $G = (V, E)$ with real edge weights $w \geq 0$, source s
Initialize:
  $dist[v] \leftarrow +\infty \quad \forall v \in V; \quad dist[s] \leftarrow 0$
  $prev[v] \leftarrow \text{null} \quad \forall v \in V$
  Priority queue Q storing pairs $(dist[v], v)$
  Q.push$((0, s))$
while (Q is not empty) do
  $(d, u) \leftarrow Q$.pop()
  if $d > dist[u]$ then continue
  for each $(v, w) \in adj[u]$ do
    $alt \leftarrow dist[u] + w$
    if $alt < dist[v]$ then
      $dist[v] \leftarrow alt$
      $prev[v] \leftarrow u$
      Q.push$((alt, v))$

Tìm đường đi ngắn nhất: Sau khi thuật toán kết thúc, ta có thể truy vết đường đi ngắn nhất từ nguồn đến đỉnh v bằng cách lần ngược theo mảng $prev$ từ v về s.

Xử lý đồ thị không liên thông: Để đảm bảo tìm đường đi đến tất cả các đỉnh trong đồ thị (có thể có nhiều thành phần liên thông), ta có thể chạy thuật toán từ mỗi đỉnh nguồn cần thiết hoặc xử lý từng thành phần liên thông riêng biệt.