**Báo cáo Project – Tuần 08**

**1. vector<vector<int>> convertMatrixToList(const string& filename):** Chuyển đổi từ ma trận kề sang danh sách kề.

\*Ý tưởng: Trong ma trận kề, mỗi dòng đại diện cho 1 đỉnh. Với mỗi dòng i, nếu như phần tử thứ j của dòng i bằng 1 tức là tồn tại cạnh nối từ dòng i-> j.

\*Cách thực hiện:

- Đọc từng dòng của file.

- Duyệt từng hàng i của ma trận. Với mỗi hàng, xem thử phần tử thứ j nào có giá trị là 1 -> Thêm đỉnh j vào danh sách kề của đỉnh i.

- Sau khi đã thêm xong tất cả các đỉnh j, đếm xem có bao nhiêu đỉnh đã được thêm vào -> thêm số lượng đã đếm được vào đầu danh sách kề.

**2. vector<vector<int>> convertListToMatrix(const string& filename):** Chuyển từ danh sách kề sang ma trận.

\*Ý tưởng: Trong danh sách kề, mỗi đỉnh i lưu số lượng đỉnh kề và các đỉnh kề với nó. Đỉnh i sẽ tương ứng với hàng thứ i trong ma trận, còn đỉnh j tương ứng với phần tử thứ j của đỉnh i. Nếu như đỉnh i kề với đỉnh j thì phần tử thứ [i][j] của ma trận sẽ bằng 1.

\*Cách thực hiện:

- Tạo 1 ma trận vuông với tất cả các phần tử bằng 0, số hàng là số lượng đỉnh.

- Duyệt từng dòng i. Với mỗi dòng i, ta duyệt từng phần tử thứ j. Nếu đỉnh i có cạnh nối với đỉnh j thì gán a[i][j] = 1.

**3.**

**- bool isDirected(const vector<vector<int>>& adjMatrix):** Kiểm tra đồ thị có hướng hay vô hướng.

\*Ý tưởng: Kiểm tra xem ma trận kề có đối xứng qua đường chéo hay không. Nếu có -> đồ thị vô hướng, ngược lại thì đồ thị có hướng.

\*Cách thực hiện: Kiểm tra xem phần tử adjMatrix[i][j] có bằng adjMatrix[j][i] (tức là phần tử ở vị trí (i, j) có bằng (j, i) không). Nếu có, trả về false, ngược lại trả về true.

**- int countVertices(const vector<vector<int>>& adjMatrix):** Đếm số đỉnh của đồ thị.

\*Ý tưởng và cách thực hiện: Số đỉnh của đồ thị cũng là số dòng của ma trận, vì thế hàm này trả về size của ma trận.

- **int countEdges(const vector<vector<int>>& adjMatrix):** Đếm số cạnh của đồ thị.

\* Ý tưởng và cách thực hiện:

- Với đồ thị có hướng: Với mỗi cạnh nối từ đỉnh i->j, trong ma trận, ở vị trí (i, j) sẽ có giá trị là 1. Vì thế, chỉ cần đếm xem có bao nhiêu số 1 ở ma trận thì đó cũng là số cạnh của đồ thị.

- Với đồ thị có hướng: Làm tương tự như đồ thị có hướng. Tuy nhiên, lúc này mỗi cạnh sẽ được đếm 2 lần, nên ta chỉ cần lấy kết quả chia 2.

- **vector<int> getIsolatedVertices(const vector<vector<int>>& adjMatrix):** Lập danh sách đỉnh cô lập của đồ thị.

\*Ý tưởng: Đỉnh cô lập là đỉnh không có cạnh nối với bất kì đỉnh nào khác. Vì thế để kiểm tra xem đỉnh đó có cô lập không, ta xem thử đỉnh đó có cạnh nào nối với đỉnh khác không, và các đỉnh khác có tồn tại cạnh nào nối với đỉnh đó không. Nếu không -> đỉnh đó bị cô lập, thêm vào vector.

\* Cách thực hiện:

- Với mỗi dòng trong ma trận, giả sử đỉnh đó bị cô lập (check = false). Ta kiểm tra xem phần tử (i, j) và phần tử (j, i) của dòng i. Nếu bất kì vị trí thứ j nào của dòng i mà (i, j) hoặc (j, i) = 1 -> Tồn tại đỉnh j có cạnh nối với i -> Đỉnh đó không bị cô lập (check = true). Nếu không tồn tại đỉnh j nào thoả mãn -> Đỉnh đó bị cô lập (check = false) -> Đẩy đỉnh đó vào vector.

- **bool isCompleteGraph(const vector<vector<int>>& adjMatrix):** Kiểm tra xem đồ thị này có phải là đồ thị đầy đủ không. ( xét đồ thị vô hướng)

\*Ý tưởng: Đồ thị đầy đủ là đồ thị có 1 đỉnh bất kì nối với tất cả các đỉnh còn lại. Để kiểm tra, ta chỉ cần xem thử có tồn tại 1 đỉnh nào không nối với tất cả các cạnh hay không.

Chuyển từ đồ thị qua ma trận, với mỗi phần tử (i, j) (i khác j) có giá trị 1 thì tồn tại cạnh nối giữa i và j; nếu có giá trị 0 thì không tồn tại cạnh nối giữa 2 đỉnh này. Vì thế, nếu như có giá trị 0 giữa đỉnh i với đỉnh j bất kì, thì đỉnh i không nối với tất cả các đỉnh còn lại của đồ thị -> đồ thị này sẽ không đầy đủ.

\*Cách thực hiện: Kiểm tra xem có phần tử (i, j) của ma trận mà phần tử này có giá trị 0 hay không.

- Nếu phần tử này nằm ở vị trí (i, j) mà i == j -> Bỏ qua, giữa đỉnh i và chính nó hiển nhiên không xét đến.

- Nếu phần tử này nằm ở vị trí (i, j) và i khác j -> Trả về false.

Nếu duyệt hết mà vẫn không tồn tại phần tử bằng 0 ngoài đường chéo chính -> trả về true.

- **bool isBipartite(const std::vector<std::vector<int>>& adjMatrix):** Kiểm tra xem đồ thị có phải là đồ thị 2 phía không.

\*Ý tưởng: Đồ thị 2 phía là đồ thị có thể chia được làm 2 tập đỉnh, sao cho mỗi tập đỉnh không có đỉnh nào có cạnh nối với nhau. Trong thuật toán này, ta sẽ dùng thuật toán bfs để kiểm tra.

- Ta sẽ tìm cách tô màu cho các đỉnh. Quy ước 0 là màu trắng, 1 là màu đen. Nếu như có 2 đỉnh kề nhau có màu giống nhau -> không thể chia ra làm 2 phía.

- Bắt đầu duyệt từ đỉnh bắt đầu, tô màu đỉnh này và dùng bfs để lan ra các đỉnh kề. + Nếu các đỉnh này chưa được tô, tô màu các đỉnh kề có màu khác đỉnh cha của nó.

+ Nếu đỉnh đã được tô rồi, xem thử đỉnh này có màu giống với màu của đỉnh kề nó không. Nếu có -> không thoả mãn đồ thị 2 phía.

- Nếu đã tô hết các đỉnh mà không gặp trường hợp đỉnh này trùng màu với đỉnh kề nó -> Đồ thị nầy có thể chia làm 2 phía.

\*Cách thực hiện:

- Tạo 1 hàng đợi queue để lưu các phần tử và phục vụ cho việc duyệt bfs.

- Đẩy phần tử đầu tiên của đồ thị vào queue để bắt đầu duyệt bfs. Tô màu cho đỉnh này.

- Cho tới khi hàng đợi này chưa rỗng, ta lấy phần tử front của hàng đợi ra. Xét tất cả các đỉnh kề với nó.

+ Nếu đỉnh này chưa được tô, tô màu cho đỉnh với màu khác màu của đỉnh cha, bằng cách lấy phần bù của màu đó (1 – color[u]).

Đẩy đỉnh này vào queue.

+ Nếu đã được tô, kiểm tra xem đỉnh này có màu giống với đỉnh cha không (color[i] == color[u]). Nếu có -> Không thể chia thành đồ thị 2 phía, trả về false.

- Nếu đã tô hết mà vẫn chưa trả về false -> trả về true.

- **bool isCompleteBipartite(const vector<vector<int>>& adjMatrix):** Kiểm tra xem đồ thị này có phải là đồ thị 2 phía đầy đủ không.

\*Ý tưởng: Đồ thị 2 phía đầy đủ là đồ thị 2 phía và mỗi đỉnh của tập này nối với tất cả các đỉnh của tập kia. Để kiểm tra xem đồ thị này có 2 phía đầy đủ không, ta cần:

- Bước 1: Kiểm tra xem đồ thị có 2 phía không.

- Bước 2: Mỗi đỉnh tập này có nối với tất cả các đỉnh tập kia không. Giả sử bước 1 đã đúng, ta sẽ có 2 tập đỉnh với 2 tập màu khác nhau. Giả sử tập đỉnh A gồm các đỉnh tô màu đen, tập đỉnh B gồm các đỉnh tô màu trắng. Ta sẽ xét xem: nếu tồn tại 1 đỉnh tô màu đen bất kỉ không có cạnh nối giữa 1 đỉnh nào đó tô màu trắng -> đỉnh này không nối với tất cả các đỉnh của tập kia -> không thoả đồ thị 2 phía đầy đủ.

\*Cách thực hiện:

- Nếu đồ thị không phải là đồ thị 2 phía -> trả về false.

- Tạo vector v1 để lưu các đỉnh có màu 1, vector v2 để lưu các đỉnh có màu 2.

- Với mỗi phần tử i trong v1, duyệt từng phần tử j trong v2. Nếu như giữa phần tử i và j không có cạnh nối -> trả về false.

- Nếu đã thực hiện hết các bước trên mà chưa trả về false -> trả về true.

**4. vector<vector<int>> convertToUndirectedGraph(const vector<vector<int>>& adjMatrix):**

Chuyển từ đồ thị có hướng sang vô hướng.

\*Ý tưởng: Đồ thị vô hướng là đồ thị mà các phần tử trong ma trận sẽ đối xứng nhau qua đường chéo chính. Vì thế, việc chuyển từ có hướng sang vô hướng là làm cho ma trận đang xét trở nên đối xứng qua đường chéo chính.

\*Cách thực hiện: Kiểm tra xem nếu phần tử ở (i, j) == 1, thì gán cho phần tử đối xứng với nó, tức là (j, i) = 1.

**5. vector<vector<int>> getComplementGraph(const vector<vector<int>>& adjMatrix):** Chuyển thành đồ thị bù.

\*Ý tưởng: Đồ thị bù là đồ thi ngược với đồ thị ban đầu: Giữa đỉnh i và đỉnh j, nếu chưa có cạnh nối thì tạo ra cạnh nối, nếu có cạnh nối thì bỏ cạnh này đi.

\*Cách thực hiện: Với ma trận, ngoại trừ đường chéo chính, những phần tử nào có giá trị 1 sẽ trở thành 0, ngược lại 0 chuyển thành 1 (bỏ các cạnh đang nối và thêm vào cạnh mà ban đầu chưa được nối).

**6. vector<int> findEulerCycle(const vector<vector<int>>& adjMatrix):** Chu trình Euler.

\*Ý tưởng: (dùng thuật toán Hierholzer)

Đồ thị có chu trình Euler là đồ thị:

- Với đồ thị vô hướng:

+ Không có đỉnh cô lập.

+ Bậc ở mỗi đỉnh đều là bậc chẵn.

- Với đồ thị có hướng:

+ Không có đỉnh cô lập.

+ Bán bậc vào bằng bán bậc ra ở mọi đỉnh.

Ta sẽ dùng bfs để duyệt các đỉnh. Tuy nhiên, sau mỗi lần thêm các đỉnh kề, ta sẽ không xoá đỉnh cũ cho tới khi đỉnh này không còn đỉnh kề. Với mỗi lần duyệt qua đỉnh kề, ta sẽ xoá cạnh nối giữa 2 đỉnh này.

\*Cách thực hiện:

- Tạo 1 vector EC (Euler Cycle) để lưu chu trình, 1 stack rỗng để lưu các đỉnh, 1 ma trận phụ sao chép từ ma trận chính (vì ma trận chính không thay đổi được), các mảng bậc của đỉnh, bán bậc vào, bán bậc ra.

- Nếu như ở vị trí (i, j) == 1, tức là đỉnh i có cạnh nối qua j: tăng bậc của i lên 1 (với đồ thị vô hướng), tăng bán bậc ra của i, tăng bán bậc vào của j (với đồ thị có hướng).

- Tìm đỉnh bắt đầu chu trình: Tìm đỉnh đầu tiên có giá trị ở ô (i, j) là 1.

- Xét xem đồ thị trên có hướng hay vô hướng:

+ Nếu vô hướng: Kiểm tra xem nếu đỉnh nào có bậc lẻ -> trả về vector rỗng.

Push đỉnh đầu tiên vào stack. Lặp cho đến khi stack rỗng, ta sẽ lần lượt lấy phần tử đầu (u) của stack ra và tìm đỉnh kề với nó (i). Nếu tìm được, ta sẽ xoá cạnh nối giữa u và i và đẩy đỉnh i vào stack, lặp lại các bước trên. Nếu không tìm được, đẩy phần tử này ra khỏi stack và thêm vào chu trình Euler.

+ Nếu có hướng: Kiểm tra xem nếu 1 đỉnh bất kì có bán bậc vào khác bán bậc ra -> trả về false.

Làm tương tự như đồ thị có hướng, nhưng chỉ khác phần kiểm tra xem có chu trình Euler không.

- Sau khi đã lưu hết các phần tử vào EC, ta đảo ngược EC lại (vì thuật toán đang làm cho EC bị ngược).

**7. \*vector<vector<int>> dfsSpanningTree(const vector<vector<int>>& adjMatrix, int start):** Duyệt đồ thị theo thuật toán dfs.

\*Ý tưởng: Giả sử các đỉnh trong đồ thị đều chưa được thăm. Bắt đầu từ đỉnh đầu tiên, đặt cho đỉnh này là đã được thăm. Duyệt tất cả các đỉnh kề của đỉnh này. Nếu đỉnh này chưa được thăm -> tiếp tục đệ quy gọi dfs đỉnh này. Tiếp tục như vậy thì tất cả các đỉnh liên thông với nhau sẽ được thăm.

\*Cách thực hiện:

- Tạo 1 hàm phụ dfs để phục vụ cho hàm chính. Trong hàm phụ này, sẽ có 1 ma trận vector phụ để lưu kết quả sau khi duyệt dfs.

- Đặt 1 mảng bool visited[] với toàn bộ giá trị trong mảng là false.

- Giả sử u là đỉnh bắt đầu. Cho đỉnh bắt đầu này có giá trị visited[u] = true, tức là đã được thăm. Với mỗi đỉnh kề của u, ta xem thử nếu đỉnh này chưa được thăm thì lưu đường đi của đỉnh này với đỉnh kia vào ma trận phụ, sau đó đệ quy tiếp tới đỉnh này.

- Qua hàm chính, ta chỉ cần gọi lại hàm phụ với kết quả được lấy từ ma trận hàm phụ.

**\*vector<vector<int>> bfsSpanningTree(const vector<vector<int>>& adjMatrix, int start):** Duyệt đồ thị theo bfs.

\*Ý tưởng: Giả sử các đỉnh của đồ thị đều chưa được thăm. Bắt đầu từ đỉnh đầu tiên, ta sẽ thăm từng đỉnh kề với nó. Với mỗi đỉnh kề nó, ta tiếp tục thăm các đỉnh kề khác cho tới khi hết đồ thị. Ta sẽ sử dụng queue để thăm.

\*Cách thực hiện:

- Đặt 1 mảng bool visited[] có tất cả giá trị trong mảng là false.

- Tạo 1 queue rỗng và đẩy đỉnh đầu tiên vào queue, đặt cho đỉnh này đã được thăm.

- Cho tới khi queue này chưa rỗng:

+ Lấy phần tử đầu tiên của queue(u) và xoá phần tử này ra khỏi queue.

+ Duyệt từng đỉnh kề của u . Nếu đỉnh kề này chưa được thăm, push đỉnh này vào queue, đánh dấu đỉnh này đã được thăm, lưu cạnh nối giữa đỉnh u và đỉnh này vào ma trận kết quả.

+ Làm tiếp tục các bước trên cho tới khi queue này rỗng.

- Trả về kết quả.

**8. bool isConnected(int u, int v, const vector<vector<int>>& adjMatrix):** Kiểm tra xem đỉnh u và v có đường đi tới nhau không.

\*Ý tưởng: Dùng thuật toán bfs để kiểm tra. Nếu như từ đỉnh u, qua thuật toán này có thể lan ra đỉnh v -> có đường đi.

\*Cách thực hiện:

Làm tương tự như thuật toán bfs để từ đỉnh u có thể lan ra hết các đỉnh có thể liên thông. Sau đó, kiểm tra xem đỉnh v có được thăm hay chưa. Nếu chưa -> u không thể đi tới v.

**9. Tìm đường đi ngắn nhất:**

**- vector<int> dijkstra(int start, int end, const vector<vector<int>>& adjMatrix):** Tìm đường đi dựa trên thuật toán dijkstra.

\*Cách thực hiện:

- Xuất phát từ đỉnh start, gán đỉnh này có khoảng cách là 0, các khoảng cách từ đỉnh khác tới đỉnh này là vô cực.

- Tạo 1 mảng parent nhằm lưu đỉnh trước của đỉnh đang xét, nhằm việc sau này lưu đường đi; 1 mảng visited để đánh dấu đã thăm đỉnh đó hay chưa.

- Tạo 1 hàng đợi ưu tiên nhằm lưu khoảng cách nhỏ nhất từ đỉnh chưa thăm tới start. Hàng đợi này có kiểu phần tử pair, lưu 2 giá trị theo cặp, trong đó giá trị đầu lưu khoảng cách từ đỉnh đó tới đỉnh start, giá trị sau là tên đỉnh.

- Đầu tiên, ta push đỉnh đầu vào queue. Đỉnh đầu có khoảng cách từ đỉnh tới start là 0.

- Lặp cho tới khi queue rỗng, ta làm như sau:

+ Lấy đỉnh đầu tiên của hàng đợi ra. Đỉnh này có khoảng cách từ đỉnh tới start là nhỏ nhất.

+ Kiểm tra xem đỉnh này có được ghé thăm chưa. Nếu đã được ghé thăm (từng được duyệt qua), bỏ qua đỉnh này. Nếu chưa, đánh dấu là đã duyệt qua.

+ Kiểm tra các đỉnh kề với đỉnh này (đỉnh u): Nếu đỉnh kề với nó chưa được ghé thăm (giả sử là đỉnh i) và khoảng cách là vô cực-> chưa từng duyệt qua đỉnh này. Nếu như vậy, ta so sánh khoảng cách của (i, start) với khoảng cách của (u, start) + khoảng cách (u, i). Nếu d(u, start) + d(i) < d(i, start) -> Cập nhật lại d(i, start) = d(u, i) + d(u, start). Sau đó, gán cha của đỉnh i là đỉnh u rồi push đỉnh i và khoảng cách của nó tới start vào queue.

+ Tiếp tục làm như vậy cho tới khi kết thúc vòng lặp.

- Kiểm tra đường đi: Duyệt ngược mảng parent từ parent.end về parent.begin, sau đó đảo lại, đó là đường đi cần tìm.