REPORT_24120184

1.

Hàm vector<vector<int>> convertMatrixToList(const string& filename):

- Hàm trên có mục đích từ 1 ma trận kề thu được 1 danh sách kề.
- Đầu tiên ta đọc lấy kích thước của ma trận, cũng chính là số đỉnh.
- Duyệt qua từng dòng (đếm số lượng cạnh mà đỉnh đó nối đến và lưu lại những đỉnh được nối)
- Lưu chúng vào danh sách kề.

2.

Hàm vector<vector<int>> convertListToMatrix(const string& filename):

- Hàm trên có mục đích từ 1 danh sách kề ta thu được 1 ma trận kề.
- Đầu tiên ta đọc lấy số đỉnh , tạo một ma trận với kích thước n x n (n là số đỉnh) với các phần tử bằng 0.
- Duyệt qua từng dòng (duyệt lần lượt các các đỉnh được đỉnh đang xét nối đến), tương ứng với từng đỉnh ta chọn cột tương ứng của ma trận cho bằng 1.
- Lưu lại và trả về ma trận kề.

3.

Hàm bool isDirected(const vector<vector<int>>& adjMatrix)

- Hàm kiểm tra xem ma trận kề đó có phải là 1 đồ thị có hướng hay không?
- Chạy vòng lặp duyệt ma trận với i là tham số duyệt dòng và j là tham số duyệt cột của ma trận.
- Ta kiểm tra lần lượt các cạnh nối từ đỉnh i đến đỉnh j , nếu có cạnh nối từ đỉnh i đến j mà không tồn tại cạnh nối từ đỉnh j đến đỉnh i thì ta trả về true do đó là đồ thị có hướng.
- Nếu kiểm tra tất cả các cạnh đều thỏa thì ta trả về false (là đồ thị vô hướng).

Hàm int countVertices(const vector<vector<int>>& adjMatrix):

- Hàm có mục đích trả về số đỉnh của đồ thị được biểu diễn bằng ma trận kề.
- Ta chỉ trả về số dòng của ma trận thì đó cũng chính là số đỉnh của đồ thị.

Hàm int countEdges(const vector<vector<int>>& adjMatrix):

- Hàm có mục đích đếm số cạnh của 1 đồ thị dựa trên ma trận kề.
- Chạy vòng lặp duyệt ma trận với i là tham số duyệt dòng và j là tham số duyệt cột của ma trận:

  -Khai báo biến đếm cho giá trị bằng 0.

-Nếu tại vị trí [i][j] của ma trận bằng 1 tức là có cạnh nối từ đỉnh i đến đỉnh j, cho biến đếm cộng thêm 1.

- Trả về số cạnh đếm được.

Hàm vector<int> getIsolatedVertices(const vector<vector<int>>& adjMatrix):

- Hàm có mục đích lưu lại những đỉnh cô lập (tức là không có cạnh nối từ đỉnh này tới đỉnh khác, hoặc không từ đỉnh khác nối với đỉnh này).
- Chạy vòng lặp duyệt ma trận với i là tham số duyệt dòng và j là tham số duyệt cột của ma trận:

  -Khai báo biến check bằng false

  -Nếu vị trí [i][j] hoặc [j][i] của danh sách kề bằng 1 tức là có cạnh nối với i cho biến check bằng true, nếu sau khi duyệt hết cột check vẫn bằng false thì thêm đỉnh vào vector để lưu.

- Trả về vector chứa các đỉnh cô lập.

Hàm bool isCompleteGraph(const vector<vector<int>>& adjMatrix)

- Hàm kiểm tra có phải là đồ thị đầy đủ hay không
- Đồ thị đầy đủ là đồ thị vô hướng trong đó tử 1 đỉnh và nối với tất cả đỉnh còn lại.
- Chạy vòng lặp duyệt ma trận với i là tham số duyệt dòng và j là tham số duyệt cột của ma trận:

  -Tại vị trí i bằng j thì bỏ qua ta không xét đến, vì chỉ xét cạnh nối từ đỉnh này với đỉnh khác , nếu tồn tại vị trí [i][j] khác 1 thì không có cạnh từ đỉnh này đến đỉnh khác trả về false, hoặc đỉnh i nối với đỉnh j khác với đỉnh j nối với đỉnh i thì ta trả về false.

- Sau khi duyệt đều thỏa thì ta trả về true.

Hàm bool isBipartite(const std::vector<std::vector<int>>& adjMatrix):

- Hàm kiểm tra xem 1 đồ thị có phải là đồ thị 2 phía hay không dựa trên ma trận kề.
- Đồ thị 2 phía là đồ thị mà tập đỉnh chia thành 2 tập rời sao cho mọi cạnh đều nối giữa 2 tập đó, không có cạnh nối 2 đỉnh trong cùng 1 tập.
- Khai báo n là số lượng đỉnh trong đồ thị và temp1, temp2 là 2 tập chưa 2 đỉnh thuộc 2 phần khác nhau.
- Chạy vòng lặp duyệt ma trận với i là tham số duyệt dòng và j là tham số duyệt cột của ma trận:

  -Trường hợp đầu tiên gặp cạnh phân cho i vào temp1 , j vào temp2:

+Phân loại vị trí của i và j , kiểm tra duyệt các phần tử trong temp 1, sau vòng lặp nếu check1 bằng true thì i có trong temp1, check2 bằng true thì j có trong temp2

+Tương tự cho temp2.

-Ta kiểm tra nếu check1 và check 2 đều bằng true hoặc check3 hoặc check4 đều bằng true tức là i và j nằm trong cùng 1 tập, có cạnh nội bộ không phải là bipartite nên trả về false:

+Trường hợp đó là 1 tập mới chia i và j vào 2 tập khác nhau.

+Trưởng hợp 1 đỉnh đã biết tập: nếu i đã nằm trong temp1 thì j phải nằm trong temp2 và ngược lại.

- Nếu duyệt tất cả mà không gặp mâu thuẫn thì trả về true là đồ thị bipartite.

Hàm bool isCompleteBipartite(const vector<vector<int>>& adjMatrix):

- Hàm kiểm tra có phải là đồ thị 2 phía đầy đủ hay không
- Đồ thị 2 phía đầy đủ có tính chất như đồ thị 2 phía nhưng thêm điều kiện 1 đỉnh bất kì từ tập bên này phải có cạnh nối với tất cả đỉnh bên tập kia và ngược lại.
- Ta kiểm tra liệu có phải là đồ thị 2 phía hay không qua hàm isPipartite khi này có 2 tập temp1 và temp2, lần lượt duyệt kiểm tra tương ứng đảm bảo rằng 1 đỉnh bất kì ở tập temp1 sẽ nối với tất cả các đỉnh ở tập temp2 và ngược lại, nếu phát hiện trường hợp không thỏa trả về false.
- Nếu duyệt tất cả mà không gặp mâu thuẫn trả về true.

4.

Hàm vector<vector<int>> convertToUndirectedGraph(const vector<vector<int>>& adjMatrix):

- Hàm giúp chuyển từ 1 đồ thị có hướng sang đồ thị vô hướng.
- Khai báo 1 ma trận kề mới với tất cả phần tử bằng 0.
- Chạy vòng lặp duyệt ma trận với i là tham số duyệt dòng và j là tham số duyệt cột của ma trận:

  -Nếu phát hiện có cạnh nối từ i đến j hoặc cạnh nối từ j đến i thì ta cho ma trận kề mới có cạnh nối từ i đến j và có cạnh nối từ j đến i.

- Trả về ma trận mới là ma trận kề biểu diễn đồ thị vô hướng.

5.

Hàm vector<vector<int>> getComplementGraph(const vector<vector<int>>& adjMatrix)

- Hàm giúp thu được 1 ma trận bù biểu diễn cho đồ thị bù, trái với lại ma trận kề cho trước
- Đồ thị bù thu được có những cạnh mà đồ thị gốc không có.
- Khai báo 1 ma trận kề mới với tất cả phần tử bằng 0.(Chính là ma trận bù)
- Chạy vòng lặp duyệt ma trận với i là tham số duyệt dòng và j là tham số duyệt cột của ma trận:

  -Tại vị trí i bằng j thì ra bỏ qua, với những cạnh mà đồ thị gốc không có tức là không có cạnh nối từ đỉnh i đến đỉnh j thì ta cho vị trí [i][j] của ma trận bù bằng 1.

- Trả về ma trận bù.

6.

Hàm vector<int> findEulerCycle(const vector<vector<int>>& adjMatrix):

- Hàm giúp phát hiện chu trình trong đồ thị thông qua ma trận kề.
- Ta cần tạo 2 vector để lưu tại 1 đỉnh thì số lượng cạnh đi ra là bao nhiêu từ đỉnh đó thông qua outDegree và cạnh đi vào đỉnh đó thông qua inDegree.
- Nếu là đồ thị vô hướng mỗi cạnh được tính là vừa đi vào vừa đi ra tức inDegree và outDegree sẽ giống nhau.
- Ta xét 2 trường hợp:

  -Trường hợp 1: Đồ thị có hướng : nếu số cạnh đi vào và đi ra của 1 đỉnh là khác nhau thì trả về rỗng do điều kiện để tồn tại chu trình Euler trong đồ thị có hướng thì số cạnh đi vào phải bằng số cạnh đi ra của 1 đỉnh.

  -Trường hợp 2: Đồ thị vô hướng: nếu số cạnh đi vào không chia hết cho 2 thì trả về false, điều kiện cần để tồn tại chu trình Euler là tất cả các đỉnh phải có bậc chẵn.

- Đầu tiên ta chọn đỉnh bắt đầu, tìm đỉnh phù hợp để đi (nếu tất cả đỉnh đều không có cạnh nào outDegree bằng 0) thì đồ thị rỗng -> trả về rỗng {}.
- Duyệt bằng Hierholzer:

  -Sử dụng stack để giữ lại những đỉnh chưa hoành thành, duyệt cho đến khi stack rỗng:

  -Khai báo temp lưu đỉnh cần xét , duyệt qua tất cả các cạnh có thể của temp:

  +Nếu tìm được cạnh thì thêm vào stack, xóa cạnh đó để tránh lần sau duyệt lại.

  +Nếu là đồ thị vô hướng phải xóa cạnh ngược lại.

  -Nếu trong quá trình duyệt không tìm được cạnh nữa thì thêm temp vào kết quả (lưu lại đường đi).

- Vì ta lưu kết quả theo hướng ngược thông qua mỗi lần rút stack nên ta cần đảo lại để đúng thứ tự đường đi Euler.

7.

Hàm vector<vector<int>> dfsSpanningTree(const vector<vector<int>>& adjMatrix,int start)

- Tạo biến visited để đánh dấu những đỉnh đã được thêm nhằm đảo bảo (số cạnh bằng số đỉnh – 1) Hàm có mục đích trả về cây khung (spanning tree) bằng cách duyệt DFS duyệt theo chiều sâu bắt đầu tử đỉnh start.
- Ta duyệt tạo cây khung bằng DFS (duyệt theo chiều sâu) bằng stack:

  -Khai báo biến temp lưu đỉnh cần xét (sau đó pop khỏi stack)

  -Sau đó duyệt theo cột để kiểm tra đinh temp có cạnh và cạnh đó có thỏa điều kiện đã visited hay chưa:

  +Nếu thỏa:
  +Thì cập nhật cạnh và cạnh ngược lại vào spanning tree (vì là đồ thị vô hướng).

  +Gán đỉnh đã thăm sau đó thêm đỉnh đó vào stack cho lần duyệt tiếp theo.

- Trả về cây khung duyệt theo DFS.

Hàm vector<vector<int>> bfsSpanningTree(const vector<vector<int>>& adjMatrix, int start):

- Hàm có mục đích trả về cây khung (spanning tree) bằng cách duyệt BFS duyệt theo chiều rộng bắt đầu tử đỉnh start.
- Tạo biến visited để đánh dấu những đỉnh đã được thêm nhằm đảo bảo (số cạnh bằng số đỉnh – 1).
- Ta duyệt vào tạo cây khung bằng BFS bằng queue:

  -Khai báo biến temp lưu đỉnh cần xét.

  -Sau đó duyệt theo cột để kiểm tra đinh temp có cạnh và cạnh đó có thỏa điều kiện đã visited hay chưa:

  +Nếu thỏa:
  +Thì cập nhật cạnh và cạnh ngược lại vào spanning tree (vì là đồ thị vô hướng).

  +Gán đỉnh đã thăm sau đó thêm đỉnh đó vào stack cho lần duyệt tiếp theo.

- Trả về cây khung duyệt theo BFS.

8.

Hàm bool isConnected(int u, int v, const vector<vector<int>>& adjMatrix)

- Hàm nhằm mục đích kiểm tra 2 đỉnh u với v có nối với nhau hay không.
- Ta chỉ cần trả về vị trí của ma trận kề tại [u][v] hoặc [v][u].

9.

Hàm vector<int> dijkstra(int start, int end, const vector<vector<int>>& adjMatrix)

- Hàm có mục đích tìm đường đi ngắn nhất từ đỉnh start đến đỉnh end thông qua thuật toán dijkstra (đồ thị có trọng số).
- Ta tạo vector dVertices(n ,INF) đầu tiên gán cho đường đi đến mỗi đỉnh là vô cùng , vector pre nhằm lưu đường đi của ngắn nhất, vector visited để kiểm tra đã duyệt qua đỉnh đó hay chưa.
- Sử dụng hàng đợi ưu tiên priority_queue, chọn khoảng cách đỉnh start bằng 0 bắt đầu duyệt: (thuật toán dijkstra)

  -Sử dụng hàng đợi ưu tiên để mỗi khi duyệt 1 đỉnh thì sẽ lựa chọn đỉnh (có trong số nhỏ nhất) khi push vào queue lên đầu.

  -Xét đỉnh (có khoảng cách nhỏ nhất) i cần xét kiểm tra cạnh kề với nó :

  -Nếu đỉnh i chưa được duyệt:

  +Duyệt tất cả các đỉnh kề với i , cập nhật khoảng cách nếu tìm được đường đi ngắn hơn theo công thức dVertices[i] + w < dVertices[j].

  +Cập nhật khoảng cách dVerctices[j] , lưu lại đường đi , đưa j vào p_q để duyệt sau.

- Sau đó kiểm tra nếu không thể đến end thì trả về rỗng
- Và phục hồi đường đi ngắn nhất qua vector pre.
- Trả về đường đi ngắn nhất.

Hàm vector<int> bellmanFord(int start, int end, const vector<vector<int>>& adjMatrix):

- Hàm có mục đích tìm đường đi ngắn nhất từ đỉnh start đến đỉnh end trong đồ thị có trọng số, bao gồm cả trọng số âm, sử dụng thuật toán Bellman-Ford.
- Ta khởi tạo vector dVertices khởi tạo khoảng cách từ start đến các đỉnh là vô cùng ,vector pre lưu đỉnh trước đó để truy vết đường đi.
- Gán khoảng cách từ start đến chính nó bằng 0: dVertices[start] = 0.
- Bước lặp chính của Bellman-Ford:

  -Lặp n - 1 lần (với n là số đỉnh):

  +Với mỗi cặp đỉnh i -> j, nếu có cạnh (adjMatrix[i][j] != 0) và khoảng cách đến j thông qua i nhỏ hơn khoảng cách hiện tại:

  - Cập nhật dVertices[j] = dVertices[i] + adjMatrix[i][j].
  - Lưu lại đỉnh trước của j: pre[j] = i.
- Kiểm tra chu trình âm:

  -Sau khi lặp n - 1 lần, kiểm tra thêm 1 lần nữa:

  +Nếu vẫn có cạnh i -> j làm cho dVertices[i] + w < dVertices[j] thì:

  - Đồ thị có chu trình âm, không thể tìm đường đi chính xác → in ra "Đồ thị chứa chu trình âm" và trả về {}.
- Sau đó kiểm tra nếu không thể đến end thì trả về rỗng
- Và phục hồi đường đi ngắn nhất qua vector pre.

- Trả về đường đi ngắn nhất.

Hình ảnh minh chứng upload folder lên github: