***Exercises***

1. **The file contains an adjacency matrix. Read the file and output the corresponding adjacency list.** 
   1. **vector<vector<int>> convertMatrixToList(const string& filename);**

* Đọc ma trận kề và push vào theo VD: đỉnh thứ i kề với các đỉnh j thì đẩy j vào vector thứ i, cuối cùng đẩy vector i vào vector res và trả ra kết quả.

1. **The file contains an adjacency list. Read the file and output the corresponding adjacency matrix.**
   1. **vector<vector<int>> convertListToMatrix(const string& filename);**

* Khai báo 1 vector<vector <int>> với tất cả chỉ là số 0.
* Đọc danh sách theo cách đỉnh kề và gán adj[i][j] với các index tương ứng 1 là các cạnh, 0 là không có cạnh. Trả ra vector là ma trận kề.

1. **Implement functions to provide the following information about a given graph:**
   1. **Bool isDirected(const vector<vector<int>>& adjMatrix);**

* Nếu ma trận không đối xứng thì nó là đồ thị có hướng.
  1. **Int countVertices(const vector<vector<int>>& adjMatrix);**
* Số đỉnh chính là size của ma trận kề.
  1. **Int countEdges(const vector<vector<int>>& adjMatrix);**
* Đếm tất cả các giá trị 1 có trong ma trận. Nếu là đồ thì vô hướng thì số cạnh là số điểm chia 2 ( vì số cạnh đếm được bị lặp lại ) . Nếu là có hướng thì giữ nguyên ( chính là số cạnh luôn ).
  1. **Vector<int> getIsolatedVertices(const vector<vector<int>>& adjMatrix);**
* Gọi tổng n là tổng số đỉnh trong đồ thị là size của ma trận. cùng với danh sách rỗng để lưu các đỉnh cô lập. Sau đó duyệt 2 vòng for để tìm vòng lặp và trả ra
  1. **Bool isCompleteGraph(const vector<vector<int>>& adjMatrix);**
* Là đồ thị vô hướng có n đỉnh, => mỗi cặp đỉnh phân biệt đều có đúng 1 cạnh nối với nhau. Tổng số cạnh sẽ là n\*(n-1)/2 với n là sai.
* Kiểm tra điều kiện ma với mọi đỉnh khác nhau luôn tồn tại cạnh, mà không tồn tại đường đi tới chính nó ( self-loop)
  1. **Bool isBipartite(const std::vector<std::vector<int>>& adjMatrix);**
* Chia đồ thị thành 2 tập đỉnh rời nhau, ở đây xét là tập màu A và B, sao cho không có cạnh nào nối hai đỉnh trong cùng một tập. Ta sẽ tô một cách xen kẽ với một đỉnh bằng 0 và các đỉnh kề nó là 1, lan ra theo thuật toán BFS. Vì vậy ta sẽ chia đồ thị ra thành 2 tập A,B là đồ thị 2 phía. Duyệt quá trình, nếu nó thỏa thì trả về true, nếu không hợp lệ ( trong quá trình ) trả ra false.
  1. **Bool isCompleteBipartite(const vector<vector<int>>& adjMatrix);**
* Kiểm tra như câu trên cùng với các điều kiện thêm là:
  + Giữa 2 tập phải được nối với nhau đầy đủ.
  + Không tồn tại cạnh ở nội bộ các tập.
  + Kiểm tra đối xứng giữa 2 bên
* Trước tiên sẽ chia tập ra làm 2, push vào 2 vector a,b. Sau đó mỗi điều kiện ta kiểm tra tuần tự bằng vòng for. Nếu một trong 3 đièu kiện sai thì trả về false.

1. **Generate a base undirected graph from a given directed graph.**
   1. **vector<vector<int>> convertToUndirectedGraph(const vector<vector<int>>& adjMatrix);**

* Xét tất cả các cạnh của đồ thị, nếu giữa các đỉnh có đường nối thì ta cho nó nối ngược lại ( đủ 2 hướng ).

1. **Generate a complement graph from a given undirected graph and output its adjacency matrix (\*undirected graph).**
   1. **vector<vector<int>> getComplementGraph(const vector<vector<int>>& adjMatrix);**

* Tạo một ma trận kề mới và xét tương tự như câu 4, như chỉ bù cạnh còn lại của đồ thị có hướng.

1. **Determine the Euler cycle from a given graph using Hierholzer’s Algorithm.**
   1. **vector<int> findEulerCycle(const vector<vector<int>>& adjMatrix);**

* Điều kiện kiên quyết: Đồ thị phải liên thông, đối với đồ thị vô hướng thì mỗi đỉnh sẽ có số cạnh nối chẵn. Đối với đồ thị có hướng thì số cạnh vào phải bằng số cạnh ra và liên thông mạnh. ( có thể dùng hàm để kiểm tra đò thị có liên thông hay không trước khi thực hiện )
* Bài toán kiểm tra xem có chu trình hay không?
* Sử dụng stack để lưu dữ liệu, tạo bản phụ để dễ điều chỉnh, bắt đầu tại đỉnh đầu tiên ( hoặc đỉnh khác có trọng số lớn hơn 0 ), vừa duyệt vừa kiểm tra cạnh vào đẩy vào cycle, cùng vào đó là biến bool hasEdge để truy vết.

1. **Find the spanning tree of a given graph using(\*undirected graph):**
   1. **vector<vector<int>> dfsSpanningTree(const vector<vector<int>>& adjMatrix, int start);**

* Bắt đầu duyệt theo DFS từ nốt bắt đầu -> đánh dầu từng node đã visited hay chưa, chỉ thêm cạnh vào khi gặp một node mới hay tạo spanning tree. => Ma trận kề của spanning tree.
* Spanning tree có thể có nhiều loại khác nhau từ theo node viếng thăm đầu tiên ( ở đây ở cho là node đầu tiên ( node 0 ) để ouput ạ.
* Giả sử đồ thị liên thông, nếu không thì chỉ xét phần tử liên thông => không trọn vẹn bài toán
  1. **vector<vector<int>> bfsSpanningTree(const vector<vector<int>>& adjMatrix, int start);**
* Tương tự, chỉ duyệt trình tự theo BFS ( queue)

1. **Verify the connection between two vertices of a given graph.**
   1. **bool isConnected(int u, int v, const vector<vector<int>>& adjMatrix);**

* Duyệt tuần tự ( BFS hoặc DFS ) và dùng visited để xem thử đỉnh đã được đánh dấu hay chưa. Bắt đầu từ u, sau đó xem thử v có được đánh dấu chưa? Nếu chưa thì trả ra false, rồi thì trả ra true.

1. **Find the shortest path between two vertices of a given graph using (\*Weighted Graph):**
   1. **vector<int> dijkstra(int start, int end, const vector<vector<int>>& adjMatrix);**

* Trình bày ý tưởng, code được thực hiện theo trên mạng ạ.
* Tạo một mảng dist đẻ lưu vị trí ngắn nhất từ start tới điểm current đó. Sử dụng min\_heap-priority queue để xác định quá trình, và đỉnh gần nhất chưa được thăm.
* Sử dụng mảng Parents để đánh dáu đỉnh trả và lưu giá trị trọng số đã đi được, cơ sở để tạo ra đường đi ngắn nhất. Mảng visited để đánh dấu những đỉnh đã đi qua.
* Chỉ tìm đường đi ngắn nhất có trọng số ( là xét khoảng cách ) và nó dương nên ta có thể sử dụng thuật toán này
  1. **vector<int> bellmanFord(int start, int end, const vector<vector<int>>& adjMatrix);**