BÀI BÁO CÁO THUẬT TOÁN DSA – WEEK 8

Nguyễn Đình Trung Kiên – 24120195

1. convertMatrixToList:

- Dùng *ifstream* để đọc ma trận từ file.
- Chuyển ma trận kề thành danh sách kề (từ các chỉ số và vị trí trong ma trận) rồi trả về.

2. convertListToMatrix:

- Dùng *ifstream* để đọc danh sách kề từ file.
- Dùng thuật toán ngược lại với *convertMatrixToList* để tạo thành danh sách kề rồi trả về.

3. isDirected:

- Kiểm tra một ma trận kề có phải là dạng biểu diễn cho một đồ thị có hướng hay không bằng cách kiểm tra từng ô từ đường chéo chính trở lên.
- Nếu cả ô đó và ô đối xứng với nó trong ma trận đều là ‘1’ thì đồ thị đang xét là vô hướng, ngược lại nếu không có ô nào như vậy thì là có hướng.

4. countVertices:

- Đếm số lượng đỉnh bằng cách lấy kích thước ma trận kề trừ đi 1.
- Vì một dòng là một đỉnh mà ta còn dùng thêm một dòng cho số lượng đỉnh.

5. countEdges:

- Duyệt từng ô từ đường chéo chính trở lên của ma trận kề rồi lấy đối xứng của nó, nếu một trong hai là ‘1’ thì đếm là 1.
- Vì nếu có hướng hay vô hướng thì hai ô đối xứng đều chỉ đại diện cho 1 cạnh, nếu đếm toàn ma trận thì sẽ bị dư.

6. getIsolatedVertices:

- Một đỉnh bị cô lập là một đỉnh không có đỉnh nào nối tới và ngược lại.
- Vì vậy ta duyệt hàng và cột của từng đỉnh, nếu đỉnh nào có cả 2 đều bằng 0 thì đó là đỉnh cô lập.

7. isCompleteGraph:

- Nếu một đồ thị là hoàn chỉnh (tất cả các đỉnh nếu nối với nhau) thì số cạnh đếm được ở một đỉnh là *n – 1* cạnh (vì ứng với mỗi đỉnh khác nó nối tới là một cạnh).
- Vì vậy nếu tất cả đỉnh đều như thế và để không đếm trùng cạnh, số cạnh của một đồ thị hoàn chỉnh là tổng của 1, 2... (n – 1).

8. isBipartite:

- Sử dụng cách tô màu từng đỉnh của đồ thị và lưu vào một vector, duyệt theo DFS để có thể tô hết toàn bộ đồ thị.
- Nếu có đỉnh nào bị tô 2 lần với 2 màu khác nhau thì chứng tỏ đồ thị đó không phải đồ thị hai phía.

9. isCompleteBipartite:

- Từ thuật toán được dùng ở *isBipartite* ta có màu của các đỉnh được lưu vào một vector.
- Ta duyệt các vector và tìm ra hai tập hợp các đỉnh có màu 1 và 2, kiểm tra các đỉnh màu 1 có nối với tất cả màu 2 hay không.
- Nếu có đỉnh nào không thỏa, thì đồ thị đó không phải đồ thị hai phía hoàn chỉnh.

10. convertToUndirectedGraph:

- Biến đổi Directed Graph bằng cách ở mỗi cạnh được nối thì nối thêm vào theo chiều ngược lại.

11. getComplementGraph:

- Khởi tạo đồ thị phần bù bằng việc nối hết tất cả các đỉnh lại với nhau.
- Biến đối từ đồ thị gốc, ở tất cả các cạnh có ở đồ thị gốc thì xóa đi ở đồ thị phần bù.

12. findEulerCycle:

- *Hàm phụ:*

  + DFSToCheckVisited: Hàm sử dụng DFS để đánh dấu các đỉnh đã được duyệt đến hay chưa.

  + vertexDegree: Hàm xác định bậc của một đỉnh (số đỉnh khác mà đỉnh đó nối tới).

  + checkDirectedEulerCycle: Hàm xác định một đồ thị có hướng có tồn tại chu trình Euler hay không. Hàm sẽ kiểm tra việc các đỉnh có bậc khác 0

trong đồ thị có được liên thông bằng *DFSToCheckVisited* và bậc vào, bậc ra của các đỉnh có bằng nhau hay không.

+ checkUndirectedEulerCycle: Hàm kiểm tra xem một đồ thị vô hướng có tồn tại chu trình Euler hay không, bằng cách kiểm tra sự liên thông các đỉnh có bậc khác không (*DFSToCheckVisited)* và bậc các đỉnh có chẵn hay không.

+ convertToVector: Chuyển một stack thành vector theo thứ tự.

+ deletePath: Xóa một cạnh của graph trong ma trận liền kề.

+ getFirstAdjacent: Hàm tìm đỉnh đầu tiên kết nối với đỉnh đang xét.

- *Chức năng hàm:*

- Sử dụng thuật toán Hierholzer để tìm chu trình Euler.
- Tìm đỉnh đầu tiên có bậc khác 0, rồi bắt đầu duyệt từ đỉnh đó.
- Duy trì hai stack, một là đường đi và hai là chu trình Euler.
- Ban đầu stack chỉ có một đỉnh là đỉnh khác 0 đầu tiên, duyệt cho đến khi stack rỗng.
- Ở mỗi lần duyệt, ta tìm đến một đỉnh bất kỳ được nối với đỉnh đứng đầu của stack bằng hàm *getFirstAdjacent.* Nếu còn đỉnh liền kề (có nghĩa là còn cạnh có thể đi được) thì ta xóa cạnh từ đỉnh đầu của stack tới đỉnh liền kề đó rồi cập nhật đỉnh liền kề đó vào stack để duyệt tiếp.
- Nếu không còn đỉnh liền kề, nghĩa là ở đỉnh đó ta không thể đi đâu được nữa thì ta cho vào chu trình Euler.
- Chuyển stack chu trình Euler thành vector rồi trả về.

13. dfsSpanningTree:

- *Hàm phụ:*

  + DFSTree: Sử dụng DFS để duyệt qua các cạnh của đồ thị, đánh dấu các đỉnh được duyệt qua và đánh dấu các cạnh được đưa vào Spanning Tree.

- *Chức năng chính:*

- Sử dụng *DFSTree* để duyệt theo DFS và tìm ra cây khung, đi theo các đỉnh chưa được duyệt qua đến khi không thể đi tiếp được nữa.
- Sau khi duyệt tìm, kiểm tra xem có đỉnh nào có bậc khác 0 mà chưa được duyệt tới hay không.

14. bfsSpanningTree:

- Duyệt từng đỉnh của đồ thị để tạo thành Spanning Tree nhưng khác với hàm *dfsSpanningTree,* ở hàm này ta dùng BFS với queue để duyệt.
- Duyệt qua các đỉnh theo thứ tự BFS, duyệt qua rồi thì đánh dấu lại cho đến khi không duyệt được đến đỉnh nào nữa thì dừng.
- Tìm xem có đỉnh nào trong đồ thị chưa được duyệt tới hay không, nếu có thì cây khung không thể hoàn thành.

15. isConnected:

- *Hàm phụ:*

- dfsCheckConnected: Sử dụng DFS và quay lui để duyệt tất cả cạnh của đồ thị để tìm xem u có đi được tới v hay không.

- *Chức năng chính:*

- Sử dụng hàm *dfsCheckConnected* để duyệt tìm và trả về kết quả.

16. Dijkstra:

- Hàm sử dụng thuận toán Dijkstra để tìm đường đi ngắn nhất trong một đồ thị weighted.
- Sử dụng thêm ba vector là *distance* để lưu lại khoảng cách nhỏ nhất để đi đến của từng đỉnh, *prev* để lưu lại đỉnh liền trước để đi đến từng đỉnh và *explored* để lưu lại từng đỉnh đã được đi qua hay chưa.
- Ở mỗi bước duyệt, ta tìm tới đỉnh chưa được *visited* có khoảng cách nhỏ nhất tính từ đỉnh *start* đến đó (lưu trong *distance*) rồi cập nhật khoảng cách cho các đỉnh liền kề với nó, lưu lại đỉnh đó vào *prev* của các đỉnh liền kề. Nghĩa là nếu đi theo đường có trọng số ngắn nhất thì phải đi qua đỉnh này mới tới được các đỉnh liền kề nó. Đồng thời nếu ở các bước duyệt sau, ta tìm được đường khác ngắn hơn thì ta cũng có thể cập nhật lại đường đi đến đỉnh này thông qua *prev.*
- Duyệt n lần (n là số đỉnh), vì cứ mỗi lần duyệt ta sẽ đánh dấu được 1 đỉnh là *visited.*
- Sau khi duyệt xong, ta truy xuất đường đi từ *end* về *start* thông qua *prev,* sau khi đã truy xuất xong mà đỉnh đầu tiên không phải *start* nghĩa là việc duyệt từ *start* không hề đến được *end* và ta trả về mảng rỗng.
- Ngược lại ta trả về đường đi hợp lệ từ *start* đến *end.*

17. bellmanFord:

- Sử dụng cho các đồ thị có trọng số âm và có thể kiểm tra các chu trình âm.

- Ở mỗi lần lặp, ta duyệt qua toàn bộ cạnh của đồ thị, nếu có thể cập nhật khoảng cách mới nhỏ hơn cho đỉnh nào thì ta cập nhật vào *distance* và *prev* để tìm lại đường đi từ *start* đến đó.
- Ở mỗi lần lặp, ta cập nhật một biến *updated,* nghĩa là ở lần lặp đó có đỉnh nào được cập nhật nữa không, nếu không ta có thể kết thúc vòng lặp ở đây vì tất cả các đỉnh đã được cập nhật.
- Tối đa vòng lặp là n – 1 lần với n là số đỉnh của đồ thị.
- Sau khi kết thúc vòng lặp, ta cần kiểm tra lại lần nữa xem có cạnh nào có thể cập nhật hay không, nếu có thì ta trả về rỗng vì đồ thị tồn tại chu trình âm, nếu tiếp tục tìm thì nó sẽ luôn tìm tới vô tận.
- Duyệt tìm như hàm *dijkstra* để tìm lại đường đi từ *start* tới *end.*