**PROJECT LÝ THUYẾT ĐỒ THỊ**

**KHTN – ĐTTX**

**---- ^\_^ ---**

1. **Mục lục**
   1. Định nghĩa lớp Graph
   2. Yêu cầu 1: Phân tích đồ thị
   3. Yêu cầu 2: Duyệt đồ thị
   4. Yêu cầu 3: Tìm cây khung nhỏ nhất
   5. Tìm đường đi ngắn nhất
   6. Tìm chu trình hoặc đường đi Euler
2. **Nội dụng**
   1. **Định nghĩa lớp Graph**
3. **public int V:** Trường này lưu trữ số lượng đỉnh trong đồ thị.
4. **public bool isSimpleGraph**: Trường này cho biết xem đồ thị có phải là đơn đồ thị (simple graph) hay không. Một đơn đồ thị là đồ thị không chứa cạnh song song hoặc cạnh bội.
5. **public bool isUndirected**: Trường này cho biết xem đồ thị có phải là đồ thị vô hướng (undirected graph) hay không. Đồ thị vô hướng là đồ thị mà các cạnh không có hướng, tức là nếu có một cạnh nối đỉnh A và B, thì cũng có một cạnh nối từ B đến A.
6. **public List<int>[ ,] adjMatrix**: Ma trận kề của đồ thị. Đây là một mảng hai chiều, trong đó mỗi phần tử là một danh sách (List) chứa các trọng số của các cạnh nối giữa các đỉnh.
7. **public int numEdges**: Số lượng cạnh trong đồ thị.
8. **public Graphs(string filePath):** Constructor của lớp "Graphs" được sử dụng để khởi tạo đối tượng đồ thị từ một tệp tin. Phương thức này đọc tệp tin chứa thông tin về đồ thị và xây dựng ma trận kề tương ứng. Nó cũng kiểm tra xem đồ thị có phải là đơn đồ thị và đồ thị vô hướng hay không.
9. **public void AddEdge(int u, int v, int w):** Phương thức này được sử dụng để thêm một cạnh từ đỉnh u đến đỉnh v với trọng số w vào ma trận kề.
10. **public void RemoveEdge(int u, int v, int w):** Phương thức này được sử dụng để loại bỏ một cạnh từ đỉnh u đến đỉnh v với trọng số w khỏi ma trận kề.
11. **public void PrintGraph()**: Phương thức này in ra ma trận kề của đồ thị.
12. **public int[] GetDegree():** Phương thức này tính và trả về bậc của mỗi đỉnh trong đồ thị vô hướng.
13. **public int[,] GetDegreeDirected():** Phương thức này tính và trả về bậc của mỗi đỉnh trong đồ thị có hướng.
14. **public int GetNumEdges()**: Phương thức này tính và trả về số lượng cạnh trong đồ thị.
15. **public bool CheckIsSimpleGraph()**: Phương thức này kiểm tra xem đồ thị có phải là đơn đồ thị hay không.
16. **public bool IsConnected()**: Phương thức này kiểm tra xem đồ thị có phải là đồ thị liên thông hay không bằng cách sử dụng thuật toán DFS.
    1. **Yêu cầu 1: Phân tích đồ thị**
17. **public Bai1(string filePath):** Constructor của lớp Bai1 được sử dụng để khởi tạo đối tượng đồ thị từ một tệp tin đầu vào. Nó sử dụng đối tượng của lớp Graphs để nạp đồ thị từ tệp tin được chỉ định.
18. **public void ShowBai1():** Phương thức này được sử dụng để hiển thị thông tin về đồ thị bao gồm:
    * In ra ma trận kề của đồ thị bằng cách gọi phương thức *g.PrintGraph().*
    * In ra thông tin về đồ thị vô hướng hoặc có hướng.
    * In ra số cạnh của đồ thị bằng cách gọi phương thức *g.GetNumEdges().*
    * In ra bậc của các đỉnh trong đồ thị bằng phương thức *PrintDegree().*
    * In ra số cạnh song song bằng cách gọi phương thức *CountParallelEdges().*
    * In ra số cạnh khuyên bằng phương thức *CountSelfLoops().*
    * In ra số đỉnh treo (leaf) bằng phương thức CountLeaf().
    * In ra số đỉnh cô lập (isolated vertices) bằng phương thức countIsolatedVertices().

* Kiểm tra xem đồ thị có phải là đồ thị liên thông hay không bằng cách gọi phương thức *g.IsConnected().*
* Kiểm tra xem đồ thị có phải là đơn đồ thị (simple graph) hay không bằng cách kiểm tra giá trị của *g.isSimpleGraph*.

1. **public void PrintDegree():** Phương thức này in ra thông tin về bậc của các đỉnh trong đồ thị. Nếu đồ thị là vô hướng, nó in ra bậc của mỗi đỉnh. Nếu đồ thị có hướng, nó in ra bậc vào và ra của mỗi đỉnh.
2. **public int CountParallelEdges():** Phương thức này tính và trả về số cạnh song song trong đồ thị. Nó lặp qua ma trận kề và đếm các cạnh có nhiều hơn một trọng số. Đối với đồ thị vô hướng, kết quả được chia cho 2.
3. **public int CountSelfLoops():** Phương thức này tính và trả về số lượng cạnh khuyên trong đồ thị, tức là các cạnh nối một đỉnh với chính nó. Nó lặp qua ma trận kề và đếm các cạnh có một đỉnh xuất phát và đích giống nhau.
4. **public int CountLeaf():** Phương thức này tính và trả về số đỉnh treo (leaf) trong đồ thị. Nếu đồ thị là vô hướng, thì đỉnh treo có bậc bằng 1. Đối với đồ thị có hướng, đỉnh treo là đỉnh có tổng bậc vào và bậc ra bằng 1.
5. **public int countIsolatedVertices():** Phương thức này tính và trả về số đỉnh cô lập (isolated vertices) trong đồ thị. Đỉnh cô lập là đỉnh không kết nối với bất kỳ đỉnh nào trong đồ thị. Nếu đồ thị là vô hướng, thì đỉnh cô lập có bậc bằng 0. Đối với đồ thị có hướng, đỉnh cô lập là đỉnh có tổng bậc vào và bậc ra bằng 0.
   1. **Yêu cầu 2: Duyệt đồ thị**
6. **public Bai2(string filePath):** Constructor của lớp Bai2 tương tự như lớp Bai1, được sử dụng để khởi tạo đối tượng đồ thị từ một tệp tin đầu vào.
7. **public void ShowBai2():** Phương thức này thực hiện các tác vụ sau:

* Kiểm tra xem đồ thị có là đồ thị liên thông không bằng cách gọi phương thức *g.IsConnected().*
* Nếu đồ thị là liên thông, nó sẽ thực hiện hai thuật toán duyệt đồ thị và in ra đường đi từ một đỉnh bắt đầu đến tất cả các đỉnh:
* Duyệt đồ thị bằng giải thuật DFS (Depth-First Search) và in ra đường đi bằng cách gọi DFS(0, new bool[g.V]).
* Duyệt đồ thị bằng giải thuật BFS (Breadth-First Search) và in ra đường đi bằng cách gọi BFS(0).
* Nếu đồ thị không liên thông, nó sẽ tính toán và in ra thông tin về các thành phần liên thông:
* Sử dụng phương thức ConnectedComponents() để tìm các thành phần liên thông.
* In ra số thành phần liên thông và liệt kê đỉnh thuộc mỗi thành phần.

1. **public List<List<int>> ConnectedComponents():** Phương thức này tìm và trả về danh sách các thành phần liên thông của đồ thị. Nó sử dụng giải thuật DFS để duyệt đồ thị và tạo danh sách các đỉnh thuộc cùng một thành phần liên thông.
2. **public void DFS(int u, bool[] visited, List<int> connectedComponent):** Phương thức DFS này được sử dụng trong phương thức ConnectedComponents(). Nó thực hiện duyệt đồ thị, đánh dấu các đỉnh đã được duyệt và thêm các đỉnh vào danh sách thành phần liên thông tương ứng.
3. **public void DFS(int start, bool[] visited):** Phương thức DFS này được sử dụng để in ra đường đi từ một đỉnh bắt đầu đến tất cả các đỉnh trong đồ thị. Nó sử dụng giải thuật DFS để thực hiện việc này và in ra đỉnh trong quá trình duyệt.
4. **public void BFS(int start):** Phương thức BFS thực hiện giải thuật duyệt đồ thị theo chiều rộng từ một đỉnh bắt đầu đến tất cả các đỉnh trong đồ thị. Nó sử dụng hàng đợi để duyệt và in ra các đỉnh trong quá trình này.
   1. **Yêu cầu 3: Tìm cây khung nhỏ nhất**
5. **public Bai3(string filePath):** Constructor của lớp Bai3 được sử dụng để khởi tạo một đối tượng đồ thị từ một tệp đầu vào, tương tự như trong các lớp Bai1 và Bai2.
6. **public void ShowBai3():** Phương thức này thực hiện các tác vụ sau:

* Kiểm tra xem đồ thị có là đồ thị liên thông vô hướng không và có thực hiện các thuật toán Prim và Kruskal không.
* Nếu đồ thị là đồ thị vô hướng và liên thông, nó sẽ thực hiện hai thuật toán tìm cây khung nhỏ nhất:
* Thuật toán Prim (thực hiện bằng phương thức Prim(int start)): Tìm cây khung nhỏ nhất sử dụng thuật toán Prim và in ra các cạnh và tổng trọng số của cây khung này.
* Thuật toán Kruskal (thực hiện bằng phương thức Kruskal()): Tìm cây khung nhỏ nhất sử dụng thuật toán Kruskal và cũng in ra các cạnh và tổng trọng số của cây khung này.

1. **Prim(int start) method:** Phương thức này thực hiện thuật toán Prim để tìm cây khung nhỏ nhất của đồ thị, bắt đầu từ đỉnh start. Sử dụng mảng parent để lưu trữ cây khung và mảng key để lưu trọng số của các đỉnh. Trạng thái của các đỉnh đã được thêm vào cây khung được theo dõi bằng mảng mstSet. Các đỉnh gốc được chọn từ start. Phương thức MinKey được sử dụng để tìm đỉnh có trọng số nhỏ nhất chưa được thêm vào cây khung. Kết quả cuối cùng là các cạnh của cây khung và tổng trọng số của cây khung được in ra bằng phương thức PrintMST.
2. **int MinKey(int[] key, bool[] mstSet) method**: Phương thức này tìm và trả về đỉnh có trọng số nhỏ nhất chưa được thêm vào cây khung.
3. **PrintMST(int[] parent, int[] key) method:** Phương thức này in ra các cạnh của cây khung nhỏ nhất và tổng trọng số của cây khung này.
4. **Kruskal() method**: Phương thức này thực hiện thuật toán Kruskal để tìm cây khung nhỏ nhất của đồ thị. Danh sách các cạnh được sắp xếp theo trọng số tăng dần. Sử dụng hai mảng parent và rank để thực hiện việc tìm đỉnh gốc và kết hợp các cây con. Kết quả cuối cùng là các cạnh của cây khung và tổng trọng số của cây khung được in ra bằng phương thức PrintMST.
5. **Edge class**: Lớp này đại diện cho một cạnh trong đồ thị. Các trường src, dest và weight lưu trữ thông tin về nguồn, đích và trọng số của cạnh. Lớp này triển khai giao diện IComparable để có thể so sánh các cạnh dựa trên trọng số.
   1. **Yêu cầu 4: Tìm đường đi ngắn nhất**
6. **public Bai4(string filePath):** Constructor của lớp Bai4 được sử dụng để khởi tạo một đối tượng đồ thị từ một tệp đầu vào, tương tự như trong các lớp Bai1, Bai2, và Bai3.
7. **public void ShowBai4():** Phương thức này thực hiện các tác vụ sau:

* Kiểm tra xem đồ thị có chứa cạnh có trọng số âm hay không bằng cách gọi phương thức isNegativeWeight().
* Nếu có cạnh có trọng số âm, nó sẽ thực hiện thuật toán Bellman-Ford từ mọi đỉnh đến tất cả các đỉnh khác.
* Nếu không có cạnh có trọng số âm, nó sẽ thực hiện thuật toán Dijkstra từ mọi đỉnh đến tất cả các đỉnh khác.

1. **bool** **isNegativeWeight**() method: Phương thức này kiểm tra xem đồ thị có cạnh có trọng số âm hay không. Nếu đồ thị là đồ thị vô hướng, nó sẽ kiểm tra tất cả các cạnh. Nếu đồ thị là đồ thị có hướng, nó cũng kiểm tra tất cả các cạnh. Trả về true nếu có ít nhất một cạnh có trọng số âm, ngược lại trả về false.
2. **void Dijkstra(int start, int end) method**: Phương thức này thực hiện thuật toán Dijkstra để tìm đường đi ngắn nhất từ đỉnh start đến đỉnh end. Sử dụng ba danh sách distance, visited, và pre để lưu trữ thông tin về khoảng cách, trạng thái đã thăm, và đỉnh trước của mỗi đỉnh. Phương thức MinDistance được sử dụng để tìm đỉnh có khoảng cách nhỏ nhất chưa được thăm. Kết quả cuối cùng là đường đi ngắn nhất và độ dài của đường đi, được in ra bằng phương thức PrintPath.
3. **int MinDistance(List<int> distance, List<bool> visited) method**: Phương thức này tìm và trả về đỉnh có khoảng cách nhỏ nhất chưa được thăm.
4. **void PrintPath(int start, int end, List<int> distance, List<int> pre) method**: Phương thức này in ra đường đi ngắn nhất từ đỉnh start đến đỉnh end. Nếu không có đường đi, nó sẽ in ra thông báo không có đường đi.
5. **void BellmanFord(int start, int end) method:** Phương thức này thực hiện thuật toán Bellman-Ford để tìm đường đi ngắn nhất từ đỉnh start đến đỉnh end. Sử dụng ba danh sách distance, pre, và rank để lưu trữ thông tin về khoảng cách, đỉnh trước, và rank của mỗi đỉnh. Kết quả cuối cùng là đường đi ngắn nhất và độ dài của đường đi, được in ra bằng phương thức PrintPath.
   1. **Yêu cầu 5: Tìm chu trình và đường đi Eluer**
6. **Constructor** **public** Bai5(string filePath): Constructor này tạo một đối tượng đồ thị từ một tệp đầu vào, tương tự như trong các lớp Bai1, Bai2, Bai3, và Bai4.
7. public **void** ShowBai5(): Phương thức này thực hiện các tác vụ sau:

* Kiểm tra xem đồ thị có phải là đơn đồ thị (simple graph) và liên thông (connected) bằng cách sử dụng thuộc tính isSimpleGraph và phương thức IsConnected() của đối tượng đồ thị.
* Nếu đồ thị là đơn đồ thị và liên thông, nó sẽ kiểm tra xem đồ thị có chu trình Euler hoặc đường đi Euler không bằng cách gọi phương thức isEulerCycle() và isEulerPath().
* Nếu có chu trình Euler, nó sẽ gọi FindEulerCycle() để tìm và in ra chu trình Euler.
* Nếu có đường đi Euler, nó sẽ gọi FindEulerPath() để tìm và in ra đường đi Euler.

1. bool **isEulerCycle**() method: Phương thức này kiểm tra xem đồ thị có chu trình Euler không. Đối với đồ thị vô hướng, nó kiểm tra xem tất cả các đỉnh có bậc chẵn không (bậc của một đỉnh là số cạnh kết nối đến nó). Đối với đồ thị có hướng, nó kiểm tra xem tất cả các đỉnh có bậc vào bằng bậc ra. Trả về true nếu đồ thị thỏa mãn yêu cầu chu trình Euler, ngược lại trả về false.
2. bool **isEulerPath**() method: Phương thức này kiểm tra xem đồ thị có đường đi Euler không. Đối với đồ thị vô hướng, nó kiểm tra xem có chính xác 2 đỉnh có bậc lẻ (số lẻ các cạnh kết nối đến chúng). Đối với đồ thị có hướng, nó kiểm tra xem có chính xác 2 đỉnh u và v sao cho deg+(u) - deg-(u) = 1 và deg+(v) - deg-(v) = -1 (bậc ra của u ít hơn bậc vào, và bậc vào của v ít hơn bậc ra). Trả về true nếu đồ thị thỏa mãn yêu cầu đường đi Euler, ngược lại trả về false.
3. void **FindEulerCycle**(int start) method: Phương thức này tìm chu trình Euler bắt đầu từ đỉnh start trong đồ thị vô hướng. Sử dụng một danh sách path để lưu trữ đường đi và giảm dần danh sách cạnh của đồ thị. In ra chu trình Euler sau khi tìm thấy nó.
4. void **FindEulerPath**() method: Phương thức này tìm đường đi Euler trong đồ thị vô hướng và liên thông. Tìm hai đỉnh u và v có bậc lẻ (được sử dụng làm đỉnh bắt đầu và kết thúc). Sử dụng một danh sách path để lưu trữ đường đi và giảm dần danh sách cạnh của đồ thị. In ra đường đi Euler sau khi tìm thấy nó.
5. **Ghi Chú**
   1. Mở tệp `Program.cs`.
   2. Sửa biến `filePath` trong hàm `Main` để chỉ định đường dẫn đến tệp đồ thị đầu vào. Định dạng tập tin input được qui định như sau:

* Dòng đầu tiên chứa số nguyên n (n > 2) thể hiện số đỉnh của đồ thị.
* n dòng tiếp theo lần lượt chứa thông tin danh sách kề của đỉnh 0 đến đỉnh n-1.
* Danh sách kề của mỗi đỉnh i được biểu diễn bằng 2\\*mi + 1 số nguyên. Số nguyên đầu tiên là số lượng đỉnh có cạnh nối xuất phát từ đỉnh i (tức là mi). Các cặp số nguyên tiếp theo là chỉ mục của đỉnh kề j (chỉ mục tính từ 0) và trọng số tương ứng của liên kết i-j.
* Các số nguyên kế cận đều cách nhau bằng một khoảng trắng.