ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

BÁO CÁO DỰ ÁN

LỰA CHỌN ĐƯỜNG ĐI TỐI ƯU THÔNG QUA HAI THUẬT TOÁN DIJKSTRA VÀ FORD-FULKERSON

Nhóm thực hiện: Nhóm 03
CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN
(MAT3514)

Hà Nội - 2023

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

BÁO CÁO DỰ ÁN
LỰA CHỌN ĐƯỜNG ĐI TỐI ƯU THÔNG QUA HAI THUẬT TOÁN DIJKSTRA VÀ FORD-FULKERSON

Nhóm thực hiện: Nhóm 03
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT
(MAT3514)

Hà Nội - 2023

LỜI NÓI ĐẦU

Tình trạng tắc nghẽn giao thông đường bộ hiện đang ngày càng xảy ra một cách nghiêm trọng. Hiện tượng này xảy ra khi số lượng phương tiện giao thông vượt quá khả năng chứa đựng của các cơ sở hạ tầng đường hiện tại. Tình trạng này không chỉ ảnh hưởng đến kinh tế, môi trường mà còn ảnh hưởng đến chất lượng cuộc sống của mỗi người dân.

Trong nội dung khuôn khổ của đề tài và môn học, nhóm thực hiện thu thập giữ liệu về lưu lượng giao thông và khoảng cách các cung đường có thể đi từ điểm gốc đến điểm đích cụ thể làm một mô hình ví dụ. Dựa trên cơ sở dữ liệu được thu thập, nhóm xây dựng một đồ thị có trọng số dựa trên lát cắt của một khu vực, sử dụng hai thuật toán kết hợp với giao diện để biểu thị con đường ngắn nhất, ít tắc nhất và tỉ lệ tắc của các đường còn lại.

Tuy nhiên, do nhiều yếu tố hạn chế khách quan và chủ quan nên những phần tìm hiểu và sản phẩm của nhóm có thể có nhiều thiếu sót. Nhóm luôn cố gắng tiếp thu những nhận xét, sự đóng góp ý kiến để từ đó có thể tiến bộ hơn.

Xin chân thành cảm ơn!

Hà Nội, tháng 12 năm 2023

THÀNH VIÊN NHÓM

Thành viên 1:

Họ và tên: Phạm Ngọc Uy

Mã sinh viên: 21000713

Lớp: K66A2 Toán tin

Email: phamngocuy_t66@hus.edu.vn

Thành viên 2:

Họ và tên: Vũ Trịnh Hoàng

Mã sinh viên: 21000681

Lớp: K66A2 Toán tin

Email: vutrinhhoang_t66@hus.edu.vn

Thành viên 3:

Họ và tên: Trần Văn Luật

Mã sinh viên: 21000690

Lớp: K66A2 Toán tin

Email: tranvanluat_t66@hus.edu.vn

Thành viên 4:

Họ và tên: Nguyễn Hoàng Minh

Mã sinh viên: 22001273

Lớp: K67A5 Khoa Học Dữ Liệu

Email: nguyenhoangminh_t67@hus.edu.vn

MỤC LỤC

CHƯƠNG 1

ĐẶT VẤN ĐỀ

1.1 Bài toán đặt ra

Hiện nay ở Hà Nội tình trạng tắc đường đang vô cùng phổ biến, cùng với đó là hệ thống đường đi được thiết kế vô cùng phức tạp với nhiều cung đường đan xen và hệ thống cơ sở hạ tầng không đồng đều giữa các tuyến đường. Điều này vô tình dẫn đến việc tham gia giao thông của người dân trở nên vô cùng khó khăn.

1.2 Phương pháp giải quyết

Sau một thời gian bàn luận, nhóm đã hình thành nên ý tưởng, sử dụng hai thuật toán Dijkstra và Ford-Fulkerson để có thể đưa ra các lựa chọn về đường đi và biểu diễn các nội dung: Đường đi ngắn nhất, đường đi ít tắc nhất, và tỉ lệ tắc trong các đường đi còn lại từ điểm A đến điểm B bất kì

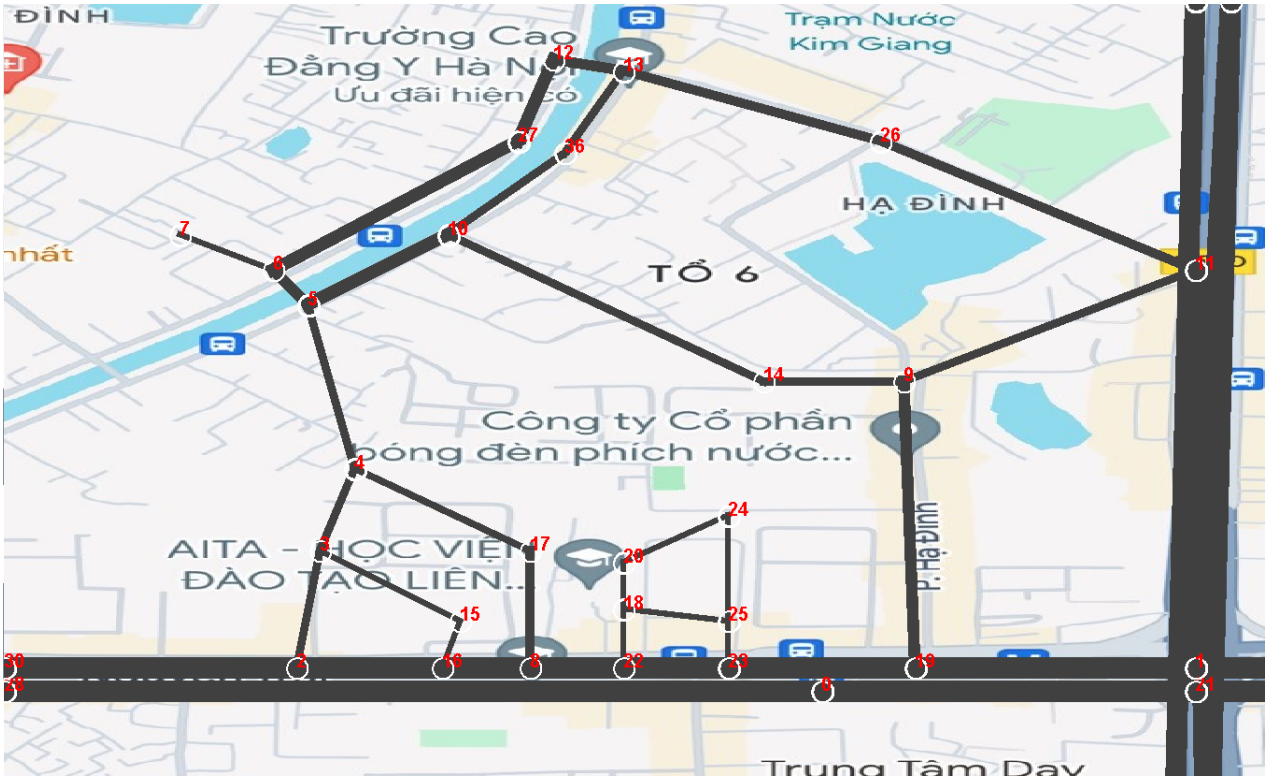
Do những hạn chế nhất định về mặt thời gian và dữ liệu, nhóm tiến hành thực hiện thử nghiệm trên một lát cắt bản đồ của khu vực Quận Thanh Xuân, Hà Nội. Nhóm đã thực hiện các nội dung: Tìm hiểu và xây dựng các chương trình lập trình cho các thuật toán đã chọn, thu thập dữ liệu về bản đồ khu vực đã chọn, xây dựng giao diện bản đồ trực quan để biểu diễn kết quả.

CHƯƠNG 2

XÂY DỰNG CẤU TRÚC DỮ LIỆU

2.1 Thu thập và xây dựng dữ liệu

Để tạo dữ liệu cho hệ thống, bước đầu tiên là xác định khu vực cần thu thập dữ liệu. Sau đó, tôi đánh dấu các điểm quan trọng trên khu vực đó, bao gồm các ngã ba, ngã tư và các điểm quan trọng khác. Với mỗi điểm quặt, ta sẽ đánh dấu là một đỉnh trên đồ thị hay một nút bằng các con số. Tiếp theo, đo khoảng cách giữa các điểm, cũng như ước tính sức chứa trung bình của mỗi con đường



Hình 2.1: Bản đồ đã được đánh dấu

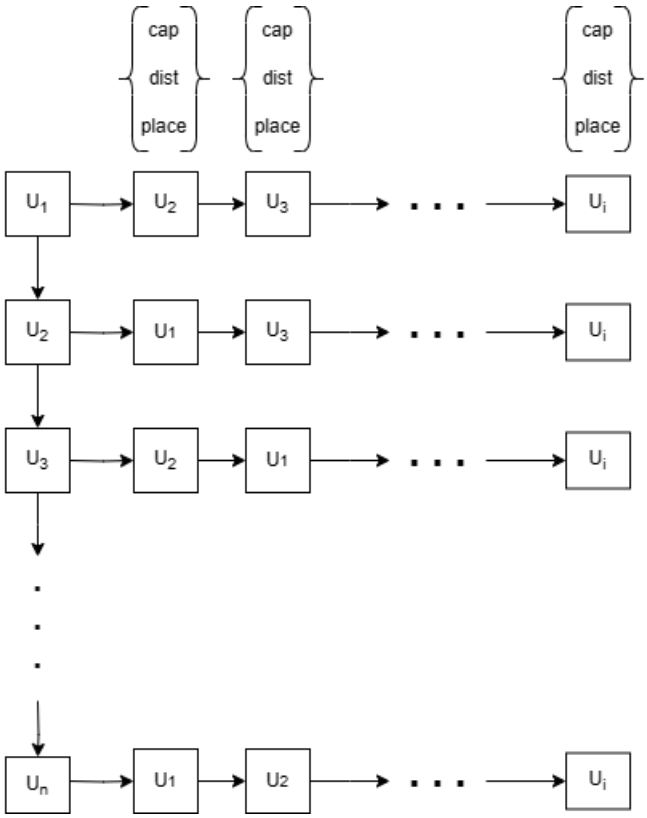
Sau khi có dữ liệu này, cần ghi chú liệu con đường là một chiều hay hai chiều. Nếu chỉ là một chiều, ta xác định chiều di chuyển trên con đường đó. Khi có đủ dữ liệu, tiến hành cập nhật chúng vào một file CSV và bắt đầu xây dựng bản đồ

Star	End	Distance	Capacity	xCor1	yCor1	xCor2	yCor2
0	21	644	18	700	590	1020	590
21	1	250	17	1020	590	1020	570
1	19	220	18	1020	570	780	570
1	11	650	18	1020	570	1020	230
19	23	210	18	780	570	620	570
19	9	480	10	780	570	770	325
11	9	250	7	1020	230	770	325
11	26	250	8	1020	230	750	120
26	13	210	8	750	120	530	60
13	12	40	9	530	60	470	50
23	25	210	5	620	570	620	530
23	22	210	18	620	570	530	570
9	14	210	7	770	325	650	325
14	10	420	7	650	325	380	200
25	24	210	5	620	530	620	440
25	18	210	5	620	530	530	520
22	18	210	5	530	570	530	520
22	8	210	18	530	570	450	570
10	5	640	11	380	200	260	260
12	27	1140	11	470	50	440	120
27	6	250	11	440	120	230	230
24	20	210	5	620	440	530	480
18	20	210	5	530	520	530	480
8	17	420	8	450	570	450	470
5	6	60	10	260	260	230	230
6	7	600	5	230	230	150	200

Hình 2.2: File CSV dữ liệu bản đồ

2.2 Xây dựng cấu trúc dữ liệu

Để triển khai thuật toán Dijkstra, chúng ta sẽ cần một loại cấu trúc dữ liệu sao cho vừa thể hiện được tính đồ thị và cũng vừa có thể lưu trữ được các thông tin về khoảng cách giữa các điểm với nhau. Chính vì vậy, nhóm đề xuất cấu trúc **Adjacent List** kết hợp cùng với một cấu trúc mà chúng tôi xây dựng gọi là **Vertex**



Hình 2.3: Cấu trúc dữ liệu cho bài toán

Đầu tiên là chúng ta sẽ có một cấu trúc List tổng quát dùng để chứa một loạt những cấu trúc List con. Mỗi một List con này sẽ thể hiện mối quan hệ của đỉnh đang xét với tất cả các đỉnh mà nó kết nối. Nói cách khác, trong một List con, phần tử ở đầu danh sách sẽ là một đỉnh Ui bất kì và nếu có đỉnh nào khác kết nối với nó thì ta thêm đỉnh đó vào trong List con mà chúng ta đang xét để thể hiện rằng giữa chúng có sự kết nối. Như vậy, mỗi một phần tử Ui trong List con sẽ chứa các thông tin như là **distance** (khoảng cách giữa đỉnh đó và đỉnh ở đầu List con) hoặc **capacity** (lưu lượng xe của cạnh kết nối đỉnh đó và đỉnh đầu List con). Các phần tử Ui này chính là cấu trúc **Vertex** đã để cập ở trên.

Dưới đây là phần code của cấu trúc **Vertex**:

```
public class Vertex implements Comparable<Vertex>{
String place;
double distance;
double capacity;

public Vertex(String place, double distance, double capacity) {
    this.place = place;
    this.distance = distance;
    this.capacity = capacity;
}

@Override
public int compareTo(Vertex o) {
    if (this.distance <= o.distance) return -1;
    else return 1;
}
```

Nhóm đã cho lớp Vertex implements giao diện Comparable để có thể sử dụng được Priority Queue trên Vertex (sẽ được đề cập đến ở phần sau). Các biến của Vertex bao gồm **place**,

distance, **capacity** được giải nghĩa như ở phần trên.

Về phần cấu trúc của đồ thị, nhóm đã triển khai code về cơ bản là như sau:

```
LinkedList<LinkedList<Vertex>> adjacentList;

public AdjacentListGraph() {
    adjacentList = new LinkedList<>();
}

public void addVertex(String place) {
    // If the place already exist, return nothing
    for (LinkedList<Vertex> link : adjacentList) {
        if (link.get(0).getPlace().equals(place)) return;
    }
    // Add new place
    LinkedList<Vertex> link = new LinkedList<>();
    link.add(new Vertex(place));
    adjacentList.add(link);
    numOfVertices++;
}

public void addEdges(String u, double distance, double capacity, String v) {
    for (LinkedList<Vertex> link : adjacentList) {
        if (link.get(0).getPlace().equals(u)) {
            Vertex destination = new Vertex(v, distance, capacity);
            link.add(destination);
        }
        if (link.get(0).getPlace().equals(v)) {
            Vertex destination = new Vertex(u, distance, capacity);
            link.add(destination);
        }
    }
}
```

Như vậy là việc xây dựng cấu trúc dữ liệu đã hoàn thành. Tiếp theo chúng ta sẽ bàn đến thuật toán Dijkstra dùng để giải quyết bài toán tìm đường đi ngắn nhất trong một đồ thị.

CHƯƠNG 3

GIẢI QUYẾT BÀI TOÁN VỚI THUẬT TOÁN DIJKSTRA

3.1 Sử dụng thuật toán Dijkstra tìm đường đi ngắn nhất

3.1.1 Giới thiệu về thuật toán Dijkstra

Dijkstra là một thuật toán phổ biến cho việc giải quyết bài toán tìm đường đi ngắn nhất từ điểm này tới điểm khác trong một đồ thị, sao cho các cạnh của đồ thị đó có trọng số không âm. Đây là thuật toán được công bố bởi nhà khoa học máy tính Edsger W. Dijkstra vào năm 1956. Trong bài toán mà chúng tôi đang xét, chúng tôi muốn tìm đường đi ngắn nhất từ một điểm gốc tới tất cả các điểm còn lại.

3.1.2 Khái quát chung về thuật toán Dijkstra trên phương diện toán học.

Nhìn chung, thuật toán sẽ bao gồm **hai tập hợp** (Set). Một tập hợp dùng để lưu trữ các điểm **chưa được đi tới** và một tập hợp sẽ dùng để lưu trữ các điểm **đã được đi qua**. Nó sẽ bắt đầu từ một điểm gốc nào đó và chọn một điểm tới chưa được ghé thăm với khoảng cách dự tính là **nhỏ nhất** trong đồ thị. Sau đó, nó đi đến những điểm **nằm xung quanh** điểm tới đó, **cập nhật lại** các khoảng cách dự tính của các điểm này nếu như một đường đi ngắn hơn được tìm thấy. Thuật toán lặp đi lặp lại bước trên cho đến khi tìm được điểm cuối, hoặc tất cả những điểm mà nó có thể đi đến được (reachable) đã đều bị ghé thăm.

3.1.3 Các bước triển khai thuật toán Dijkstra.

1) Đánh dấu **điểm gốc** với khoảng cách là 0 và tất cả các điểm còn lại với khoảng cách là dương vô cùng.

2) Chọn một điểm chưa được ghé thăm với khoảng cách hiện tại tới điểm gốc nhỏ nhất là **điểm hiện tại**.

3) **Với mỗi hàng xóm N** của điểm hiện tại, cập nhật khoảng cách của điểm **N** bằng cách **cộng khoảng cách của điểm hiện tại với trọng số trên cạnh kết nối chúng**. Nếu khoảng cách này nhỏ hơn khoảng cách dự tính của điểm N thì cập nhật nó, còn không thì không cập nhật.

4) Đánh dấu điểm hiện tại là **đã ghé thăm**.

5) Quay lại bước 2 nếu vẫn còn điểm chưa ghé thăm.

3.1.4 Mã giả của thuật toán Dijkstra

Hiện nay, có nhiều cách để triển khai thuật toán Dijkstra. Tuy nhiên, hai phương pháp phổ biến nhất thường được nhắc đến là sử dụng **Priority Queue** và **Array**. Trong đó, sử dụng Priority Queue là phương pháp hiệu quả hơn đáng kể. Trong bài báo cáo này, chúng tôi cũng sẽ sử dụng Priority Queue để triển khai thuật toán.

Quy ước: Mỗi một đỉnh ở trong thuật toán của chúng tôi sẽ chứa đựng một biến gọi là "khoảng cách". Biến này nhằm lưu lại giá trị khoảng cách nhỏ nhất từ điểm nguồn tới điểm chứa nó. Như vậy, biến "khoảng cách" ở trong điểm nguồn mặc định là sẽ bằng 0.

```
 function Dijkstra(Graph, source):
/* Khởi tạo khoảng cách dự tính trên tất cả các
đỉnh là dương vô cùng, ngoại trừ điểm nguồn*/
distances = vô cùng tại tất cả các điểm
distances = 0 tại điểm nguồn

/* Khởi tạo một tập rỗng các đỉnh "đã ghé thắm" và
một Priority Queue để theo dõi các điểm "định ghé thăm" */
visited = empty set
queue = new PriorityQueue()
queue.enqueue(source, 0)

/* Lặp cho tới khi tất cả các đỉnh đã đều được ghé thăm*/
while queue is not empty:

    /* Dequeue đỉnh có khoảng cách là nhỏ nhất
    trong Priority Queue */
    current = queue.dequeue()

    // Nếu đỉnh này đã được ghé thăm thì bỏ qua nó
    if current in visited:
        continue

    // Cho đỉnh này vào tập "đã ghé thăm"
    visited.add(current)

    /* Kiểm tra tất cả các đỉnh hàng xóm xem liệu rằng chúng
    có cần phải cập nhật lại khoảng cách dự tính hay không */
    for neighbor in Graph.neighbors(current):

        /* Tính khoảng cách dự tính của điểm hàng xóm
        thông qua điểm mà chúng ta đang xét */
        tentative_distance = distances[current]
                        + Graph.distance(current, neighbor)
```

```
            /* Nếu khoảng cách dự tính nhỏ hơn khoảng cách
            trên điểm hàng xóm đó thì cập nhật lại */
            if tentative_distance < distances[neighbor]:
                distances[neighbor] = tentative_distance

                /* Thêm vào Priority Queue điểm hàng xóm đó
                cùng với khoảng cách mới của nó */
                queue.enqueue(neighbor, distances[neighbor])

    /* Trả về khoảng cách nhỏ nhất từ điểm nguồn tới tất cả các
    điểm khác trong đồ thị */
    return distances
```

Cụ thể, thuật toán của chúng tôi sẽ được triển khai như sau trong Java:

```java
public ArrayList<String> findShortestPath(String origin, String destination) {
    ArrayList<String> path = new ArrayList<>();
    boolean[] visited = new boolean[numOfVertices];
    HashMap<String, Vertex> map = new HashMap<>();
    PriorityQueue<Vertex> queue = new PriorityQueue<>();
    map.put(origin, new Vertex(origin, 0.0));
    queue.add(new Vertex(origin, 0.0));

    while (!queue.isEmpty()) {
        Vertex temp = queue.poll();
        String v = temp.getPlace();
        double distance = temp.getDistance();
        visited[findLinkIndexInAdjacentMatrix(v)] = true;

        LinkedList<Vertex> link = getLink(v);
        for (Vertex next : link) {
            if (!visited[adjacentList.indexOf(getLink(next.getPlace()))]) {
                if (!map.containsKey(next.getPlace())) {
                    map.put(next.getPlace(), new Vertex(v, distance + next.getDistance()));
                } else {
                    Vertex sn = map.get(next.getPlace());
                    if (distance + next.getDistance() < sn.getDistance()) {
                        sn.setPlace(v);
                        sn.setDistance(distance + next.getDistance());
                    }
                }
                queue.add(new Vertex(next.getPlace(), distance + next.getDistance()));
            }
        }
    }
    path.add(destination);
    while (!path.get(0).equals(origin)) {
        String temp = map.get(path.get(0)).getPlace();
        path.add(0, temp);
    }

    return path;
}
```

Nếu đối chiếu so với phần mã giả, ccó thể thấy được rằng nếu triển khai thuần theo mã giả, chúng ta sẽ chỉ thu được khoảng cách ngắn nhất từ điểm nguồn tới tất cả các điểm khác mà không truy vết được đường đi ngắn nhất mà thuật toán đã đi qua. Chính vì vậy, khi triển khai cụ thể, nhóm đã thêm một cấu trúc dữ liệu Map để có thể lưu lại đường đi của thuật toán. Mỗi một phần tử ở trong map sẽ lưu trữ vị trí của một đỉnh N, khoảng cách ngắn nhất

từ đỉnh N đó đến điểm nguồn, và một đỉnh khác là đỉnh mà thuật toán phải đi qua ngay trước đó để đến điểm N thì khoảng cách kia mới là ngắn nhất. Như vậy khi kết thúc thuật toán, công việc còn lại là truy ngược lại đường đi ngắn nhất. Chúng ta sẽ bắt đầu từ điểm cuối cùng và truy ngược lại Map tới điểm đầu.

Ngoài ra, trong phần triển khai cụ thể, nhóm có sử dụng hai helper method để có thể lấy ra được một List con trong Adjacent List và vị trí của List con đó:

```
public int findLinkIndexInAdjacentMatrix(String s) {
    LinkedList<Vertex> link = new LinkedList<>();
    for (LinkedList<Vertex> l : adjacentList) {
        if (l.get(1).getPlace().equals(s)) {
            link = l;
            break;
        }
    }

    return adjacentList.indexOf(link);
}
```

```
public LinkedList<Vertex> getLink(String s) {
    for (LinkedList<Vertex> l : adjacentList) {
        if (l.get(0).getPlace().equals(s)) {
            return l;
        }
    }

    return null;
}
```

Như vậy là việc triển khai thuật toán Dijkstra đã hoàn tất, phần tiếp theo chúng ta sẽ xem xét hướng phát triển của chính thuật toán Dijkstra trên một bài toán khác đó là tìm đường với lưu lượng giao thông thoáng nhất.

3.2 Phát triển thuật toán Dijkstra tìm đường đi ít tắc nhất

Có lẽ chúng ta đều đã quen thuộc với việc sử dụng thuật toán Dijkstra cùng các biến thể của nó trong việc tìm kiếm các đường đi tối ưu. Tuy nhiên, ta có thể thử bổ sung thuộc tính **capacity** vào đối tượng **Vertex** và điều chỉnh một chút ở phần thuật toán để từ bài toán tìm đường đi ngắn nhất ta sẽ tìm ra một đường đi với lưu lượng giao thông thông thoáng nhất.

Ở đây thuộc tính kiểu int là **capacity** mang ý nghĩa là dung tích chứa tối đa của đường đi, hay nói cách khác là số lượng xe tối đa đi qua mà đường đi có thể chịu được để không gây ra tình trạng ùn tắc.

Sau đó ta tiến hành xây dựng lại các bước của thuật toán để đi tìm capacity nhỏ nhất từ điểm gốc đến các điểm còn lại trong đồ thị như sau:

1) Đánh dấu **điểm gốc** với capacity là dương vô cùng và tất cả các điểm còn lại với capacity là âm vô cùng.

2) Chọn một điểm chưa được ghé thăm với capacity hiện tại tới điểm gốc là **điểm hiện tại**.

3) **Với mỗi hàng xóm N** của điểm hiện tại, tìm ra gía trị nhỏ nhất giữa hai giá trị là capacity của điểm gốc, và trọng số trên cạnh nối liền giữa chúng. Nếu giá trị này này lớn hơn capacity của điểm N thì cập nhật nó, còn không thì không cập nhật.

4) Đánh dấu điểm hiện tại là **đã ghé thăm**.

5) Quay lại bước 2 nếu vẫn còn điểm chưa ghé thăm.

Ta có thể biểu diễn thuật toán trên bằng đoạn code dưới đây:

```
public static ArrayList<Integer> findMinCongestionPath (int origin , int destination ) {
   ArrayList<Integer> path = new ArrayList<>();
   boolean[] visited = new boolean[numOfVertices];
   HashMap<Integer, Vertex> map = new HashMap<>();
   int[] maxCapacity = new int[numOfVertices];
   Arrays.fill(maxCapacity, Integer.MIN_VALUE);
   maxCapacity[origin] = Integer.MAX_VALUE;

   PriorityQueue<Vertex> pq = new PriorityQueue<>(Comparator.comparingInt(vertex ->
       -maxCapacity[vertex.getPlace()]));
   pq.add(new Vertex(origin, Integer.MAX_VALUE));

   while (!pq.isEmpty()) {
       Vertex uVertex = pq.poll();
       int u = uVertex.getPlace();
       if (visited[u]) {
           continue;
       }
       visited[u] = true;
       LinkedList<Vertex> link = getLink(u);
       for (Vertex next : link) {
           int v = next.getPlace();
           int weight = (int) next.getCapacity();
           int minCapacity = Math.min(maxCapacity[u], weight);
           if (minCapacity > maxCapacity[v]) {
               maxCapacity[v] = minCapacity;
               map.put(v, new Vertex(u, next.getCapacity())); //
               pq.add(new Vertex(v, minCapacity));
           }
       }
   }

   path.add(destination);
   while (path.get(0) != origin) {
       int temp = map.get(path.get(0)).getPlace();
       path.add(0, temp);
   }

   return path;
}
```

Trong đó **origin** là điểm xuất phát, **destination** là đích đến. Phương thức trên sẽ trả về một tập các đỉnh biểu thị đường đi mà ở đó giao thông sẽ thông thoáng nhất

CHƯƠNG 4

GIẢI QUYẾT BÀI TOÁN VỚI THUẬT TOÁN FORD-FULKERSON

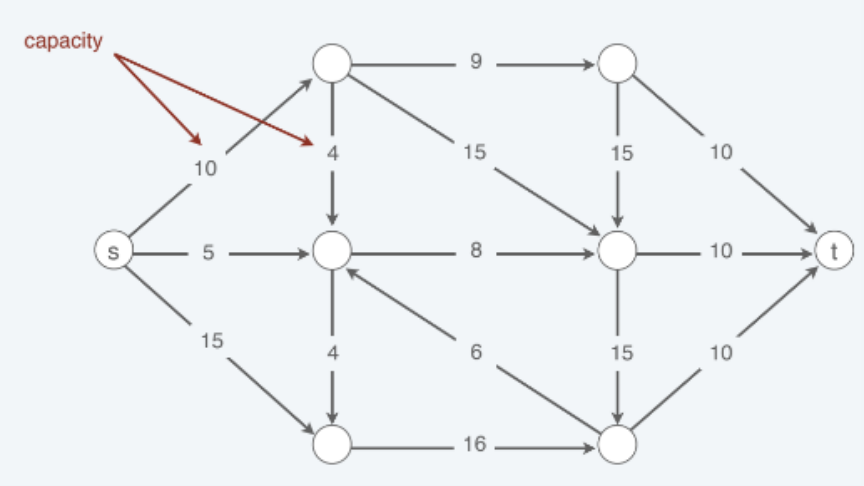
4.1 Giới thiệu về bài toán max-flow, min-cut

4.1.1 Luồng trên mạng

Một luồng trên mạng **(a flow network)** là một đồ thị G = (V, E, s, t, c)). Trong đó:

- Đồ thị có hướng G (V, E) với đỉnh nguồn $s \in V$ và nút thu $t \in V$
- Dung tích capacity (không âm) $c(e)$ với mỗi cạnh $e \in E$

Phát biểu bài toán luồng trên mạng: Một lượng vật chất được gửi qua mạng truyền tải, từ đỉnh nguồn đến đỉnh thu.

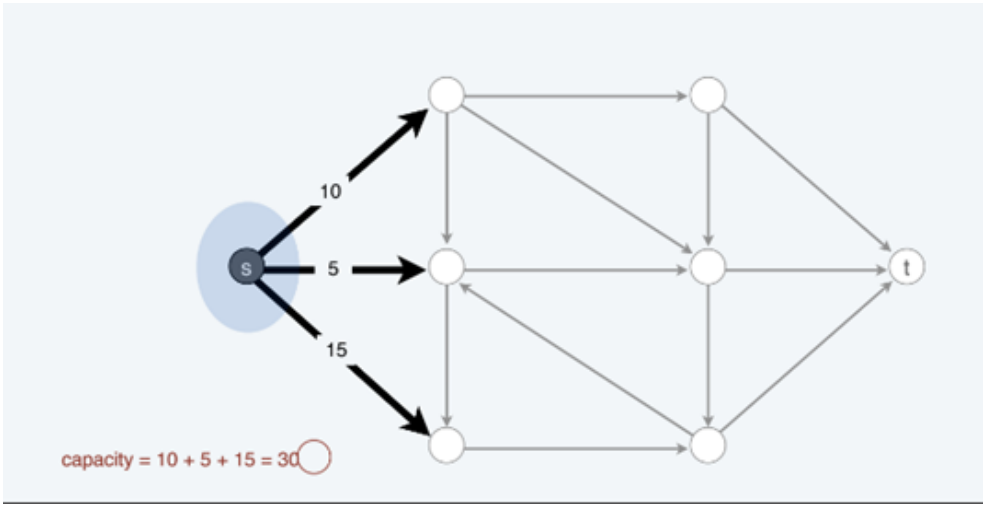


Hình 4.1: Flow Network

4.1.2 Bài toán lát cắt cực tiểu (min-cut)

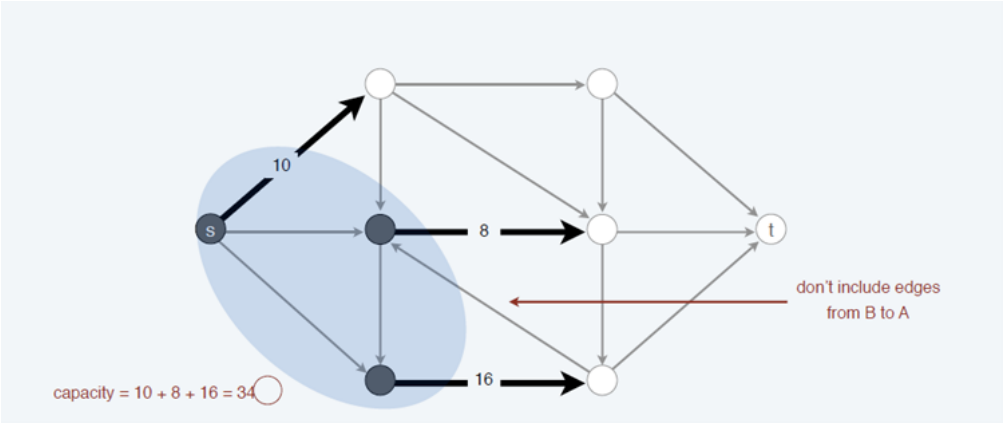
- **Định lý 1:** Một lát cắt là một phân hoạch (A, B) của tập cạnh với $s \in A$, $t \in B$
- **Định lý 2:** Dung tích (capacity) là tổng của năng lực (capacities) của các cạnh từ A đến B.

$$cap(A, B) = \sum_{eoutofA} c(e)$$



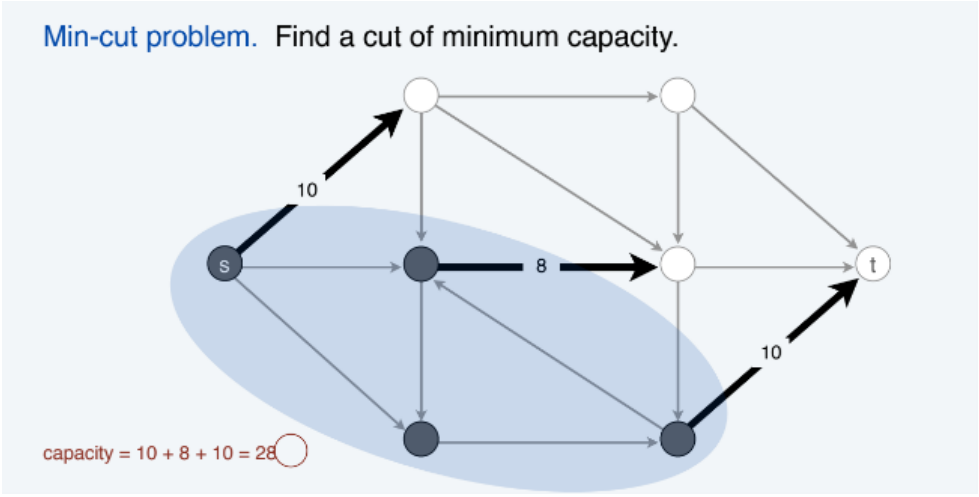
Hình 4.2: A = {s}, B = {V - s}

Dung tích của một lát cắt là tổng năng lực của các cạnh **đi ra từ lát cắt A đến lát cắt B**, không bao gồm cạnh với chiều ngược lại từ B đến A.



Hình 4.3

Bài toán min-cut: Tìm ra một lát cắt sao cho dung tích là nhỏ nhất.



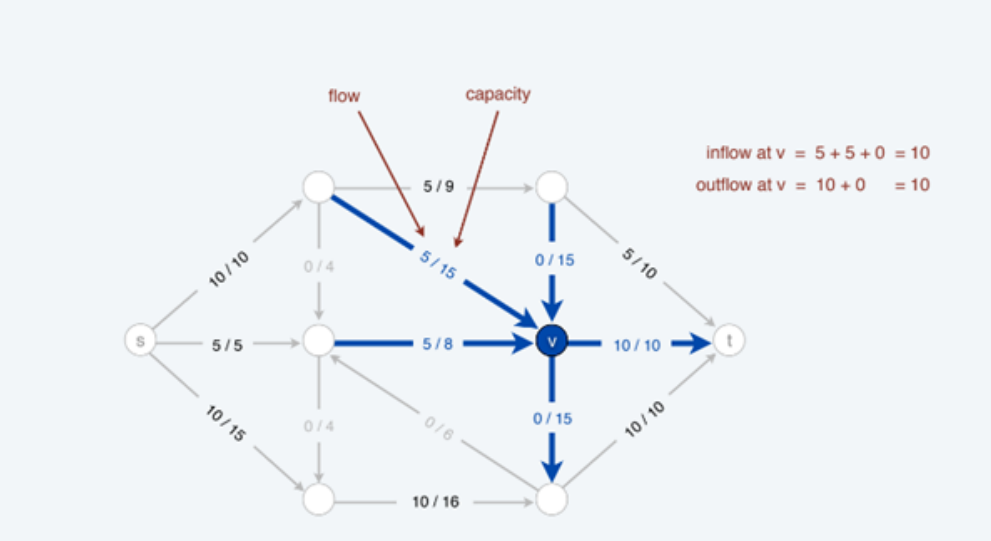
Hình 4.4: Bài toán min-cut

4.1.3 Bài toán luồng cực đại (max-flow)

Định lý 3: Một luồng (flow) f là một hàm thỏa mãn:

- Với mỗi cạnh $e \in E : 0 \leqslant f(e) \leqslant c(e)$.
- Với mỗi đỉnh $v \in V - \{s, t\}$:

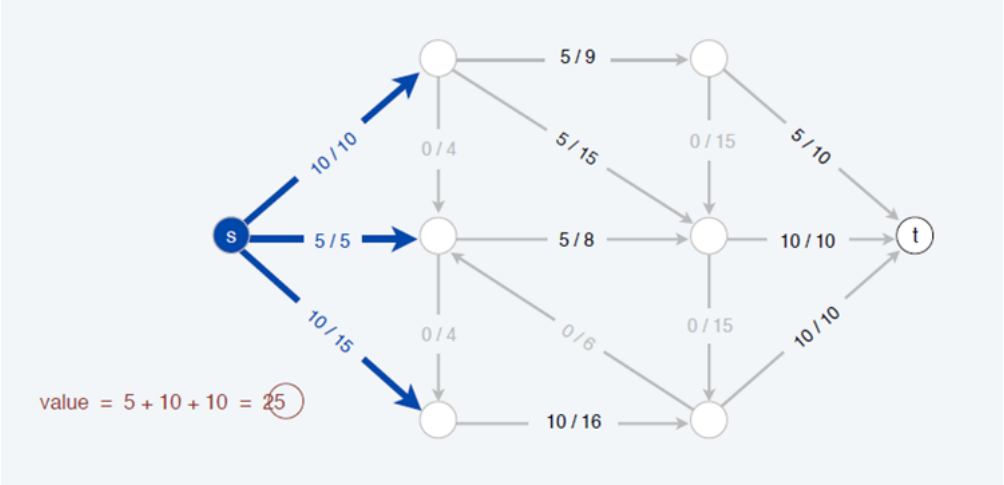
$$\sum_{e\ in\ to\ V} f(e) = \sum_{e\ out\ of\ V} f(e)$$



Hình 4.5: Định lý 3

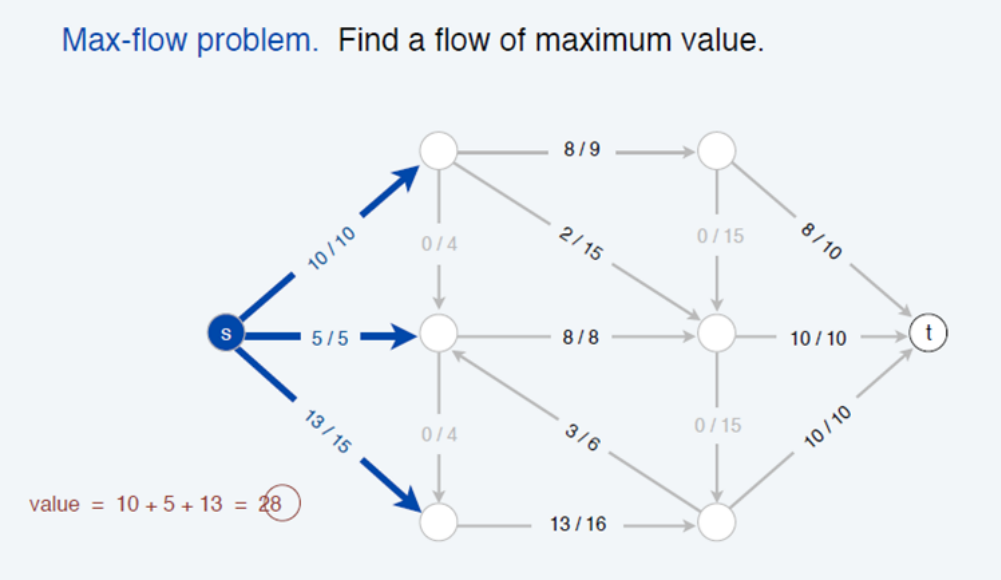
Định lý 4: Giá trị (value) của một luồng f là:

$$val(f) = \sum_{e\ out\ to\ V} f(e) - \sum_{e\ in\ of\ V} f(e)$$



Hình 4.6: Định lý 4

Bài toán max-flow: Tìm ra một flow sao cho value lớn nhất

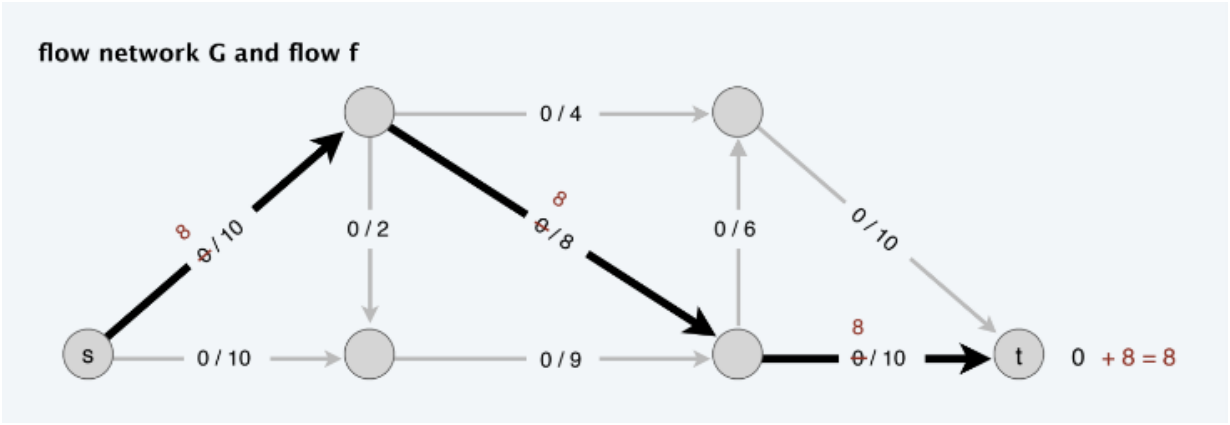


Hình 4.7: Bài toán max-flow

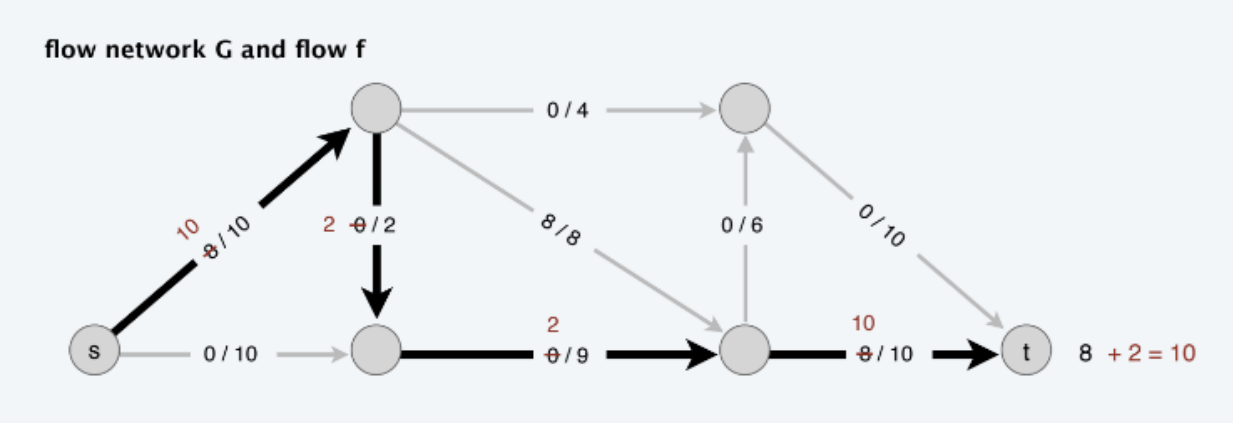
4.2 Thuật toán Ford-Fulkerson

4.2.1 Thuật toán tham lam (greedy algorithm)

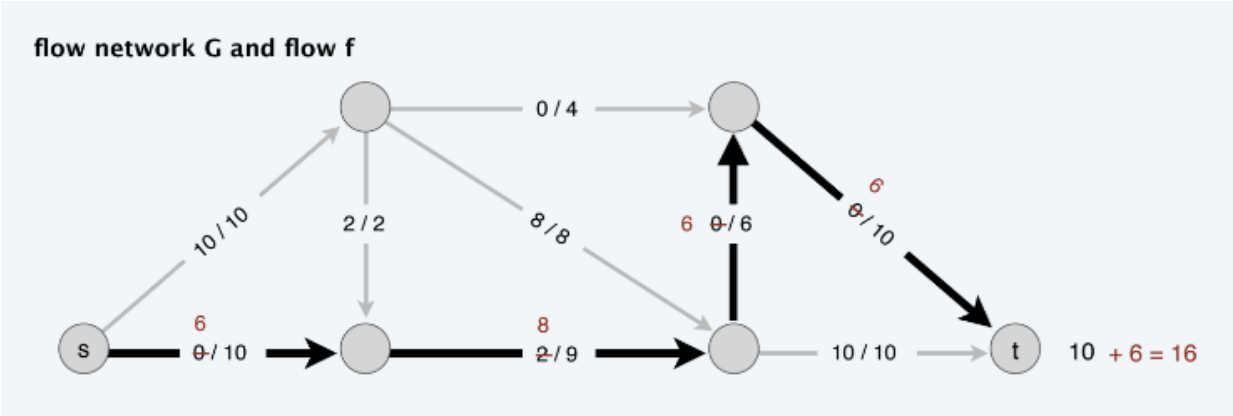
- Bắt đầu với $f(e) = 0$ với mỗi cạnh $e \in E$
- Tìm một đường đi từ s đến t, nơi có mỗi cạnh e thỏa mãn: $f(e) < c(e)$
- Tăng luồng (augment) dọc theo đường đi vừa chọn.
- Lặp lại cho đến khi không thể tiếp tục.



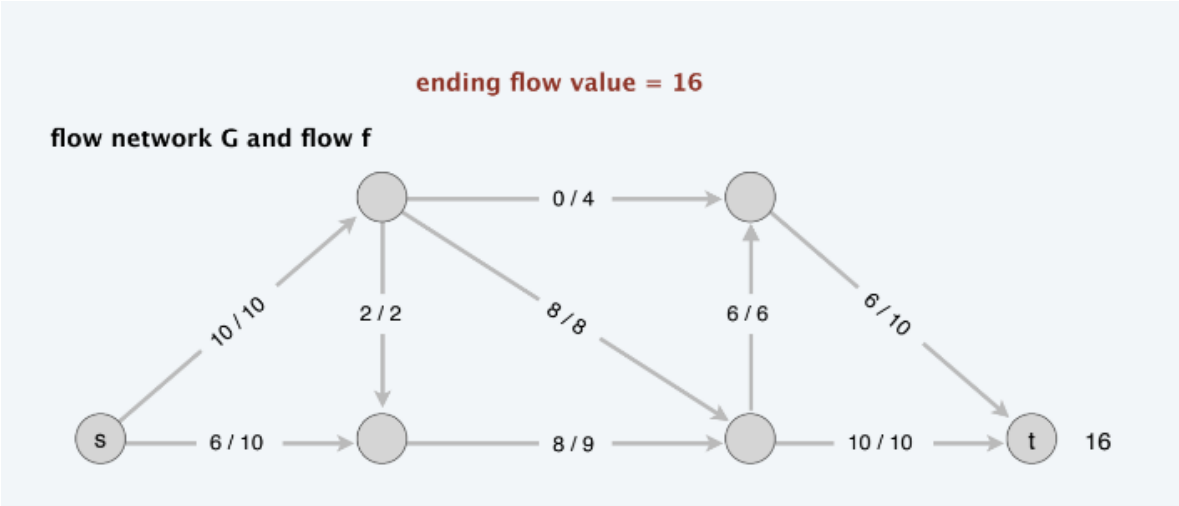
Hình 4.8: Step 1



Hình 4.9: Step 2

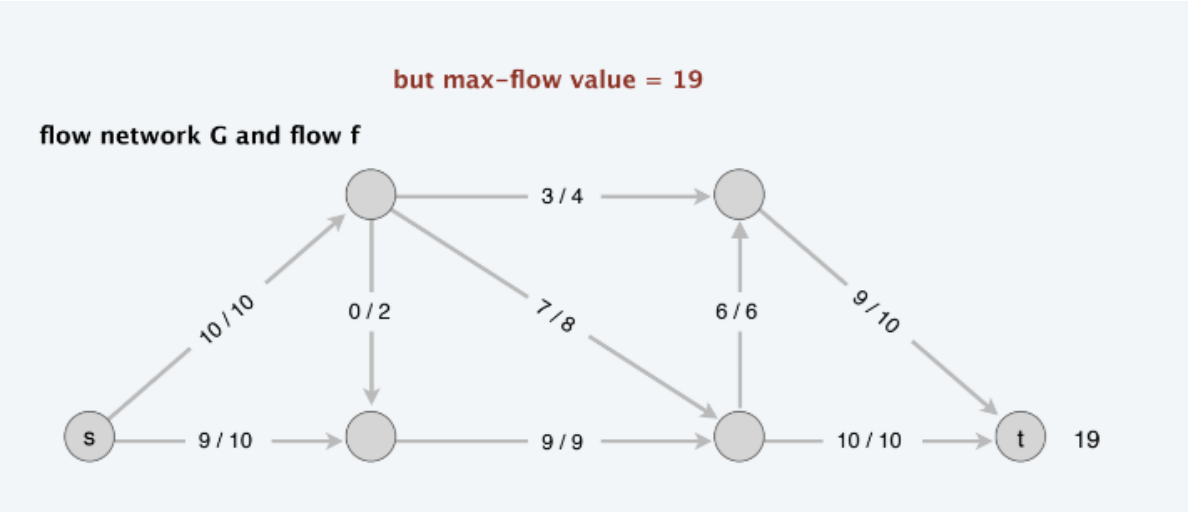


Hình 4.10: Step 3



Hình 4.11: Step 4

Tuy nhiên ta có thể tìm ra một flow với giá trị lớn hơn:



Hình 4.12: Step 5

Vậy vấn đề ở đây là gì? Thuật toán tham lam chọn luồng chưa tối ưu. Khi chọn một luồng đi qua cạnh (u, v), dung tích của cạnh không bao giờ được cập nhật lại. Nói cách khác dung tích của cạnh đó luôn tăng. Ta cần xây dựng thuật toán để có thể chọn lại được luồng tối ưu hơn khi đã chọn luồng ban đầu chưa tối ưu.

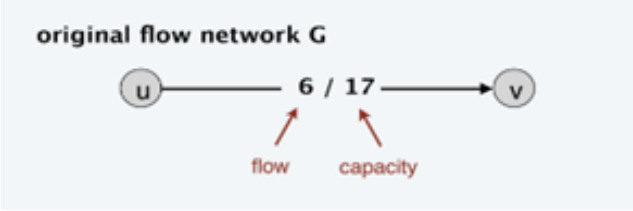
4.2.2 Thuật toán Ford-Fulkerson

Ta cần một giải pháp để có thể "undo" luồng đã chọn ban đầu nếu luồng đó chưa tối ưu.

Mạng còn dư (Residual network) Ý tưởng của residual network là khởi tạo một cạnh ảo có chiều ngược lại với cạnh ban đầu. Nhằm mục đích ta có thể gửi lại luồng từ t đến s trong trường hợp luồng ta vừa gửi chưa tối ưu.

Original edge $e = (u, v) \in E$

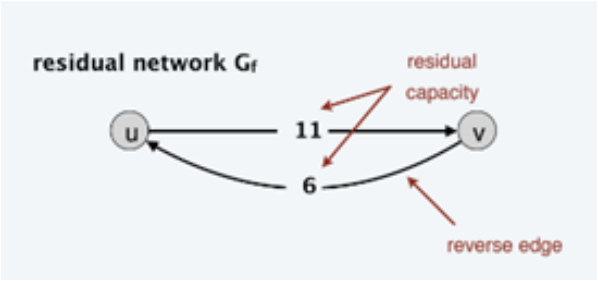
- Flow $f(e)$
- Capacity $c(e)$



Hình 4.13: Original edge

Reverse edge $e^{reverse} = (v, u)$

- "Undo" flow đã gửi.



Hình 4.14: Residual Network G_f

Dung tích còn dư (Residual Capacity).

$$C_{f(e)} = \begin{cases} c(e) - c(f) & \text{if } e \in E \\ f(e) & \text{if } e^{\text{reverse}} \in E \end{cases}$$

Mạng còn dư (Residual Network G_f) $G_f = (V, E_f, s, t, c_f)$

- $E_f = \{e : f(e) < c(e)\} \cup \{e^{\text{reverse}} : f(e) > 0\}$
- f' là một flow trong đồ thị G_f khi và chỉ khi $f' + f$ là một flow trong G.

Đường tăng (Augment path)

Định lý 4.2.1. *Một đường tăng là một đường đi đơn giản từ s đến t trên đồ thị còn dư G_f.*

Định lý 4.2.2. *Dung lượng cổ chai (bottleneck capacity) của một đường tăng P là dung lượng còn dư nhỏ nhất trên bất kỳ cạnh nào trong P.*

Định lý 4.2.3. *Gọi f là một luồng và gọi P là một đường tăng trong đồ thị còn dư G_f. Khi đó, sau khi tìm thêm đường tăng, luồng f' là một luồng kết quả và $val(f') = val(f) + bottleneck(G_f, P)$.*

AUGMENT (f, c, P)

$b \leftarrow$ bottleneck capacity of path P.

FOREACH edge $e \in P$

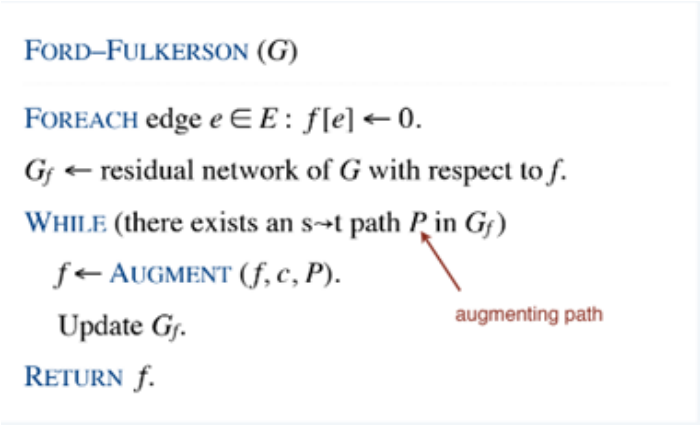
  IF $(e \in E)$ $f[e] \leftarrow f[e] + b$.

  ELSE $f[e^{\text{reverse}}] \leftarrow f[e^{\text{reverse}}] - b$.

RETURN f.

Hình 4.15: Mã giả đường tăng augment path

Như vậy ta đã giải quyết vấn đề của thuật toán tham lam ban đầu. Dưới đây là mã giả của thuật toán Ford-Fulkerson:



Hình 4.16: Mã giả thuật toán Ford-Fulkerson

4.3 Triển khai thuật toán

Để phù hợp với đề tài, chúng em đã có một số cải thiện trong thuật toán gốc ban đầu như sau:

- Trả về giá trị max-flow khi biết điểm đầu s và đỉnh đích t
- Sau khi chạy thuật toán, giá trị max-flow là duy nhất. Khi đó mỗi cạnh sẽ có một lượng flow đi qua. Chúng em dựa trên lượng flow đi qua mỗi cạnh để đánh giá tỉ lệ $\frac{flow}{capacity}$ để đưa ra kết luận cạnh đó có gây hiện tượng "tắc đường" hay không. Từ đó phân loại và hiển thị kết quả những cạnh đang "tắc đường", "không tắc" và "tắc vừa phải".
- Kết hợp cùng thuật toán Dijkstra để đưa ra tình trạng giao thông hiện tại. Từ đó giúp người dùng đánh giá và lựa chọn đường đi phù hợp.

Một số đối tượng trong chương trình

- **Edge** - lưu đối tượng cạnh của đồ thị G
- **graph** - lưu dưới dạng danh sách kề của cạnh
- **visited**[] - mảng đánh dấu đỉnh đã được thăm hay chưa.
- **flow** - biến lưu flow sau mỗi vòng lặp. Cuối cùng trả về giá trị maximum flow

Triển khai các bước thuật toán

4.3.1 Tìm một luồng bất kỳ từ đỉnh s đến t

Sử dụng thuật toán duyệt đồ thị DFS (Depth First Search) để tìm một đường đi bất kỳ từ đỉnh s đến t. Trong quá trình tìm, cập nhật các thuộc tính.

```
private long dfs(int node, long flow) {
    if (node == t) return flow;

    List<Edge> edges = graph[node];
```

```
    visit(node);

    for (Edge edge : edges) {
        long rcap = edge.remainingCapacity();
        if (rcap > 0 && !visited(edge.to)) {
            long bottleNeck = dfs(edge.to, min(flow, rcap));
                // Augment flow with bottleneck value
                if (bottleNeck > 0) {
                    edge.augment(bottleNeck);
                    return bottleNeck;
                }
            }
        }
        return 0;
    }
```

Thuật toán sẽ duyệt qua tất cả các cạnh hàng xóm của một đỉnh `for (Edge edge : edges)` theo thứ tự đỉnh đã được đánh thứ tự trong bộ data.
Ở mỗi vòng lặp, *rcap* (remain capacity) `long rcap = edge.remainingCapacity()` sẽ luôn được cập nhật như sau:

```
    public long remainingCapacity() {
        return capacity - flow;
    }
```

Thuật toán sẽ chỉ duyệt đến cạnh tiếp theo khi thỏa mãn hai điều kiện: cạnh đó vẫn còn khả năng thông qua ($rcap > 0$) và đỉnh tiếp theo nối với cạnh đó chưa được thăm `if(rcap > 0 &&!visited(edge.to))`.
Khi đó, tiếp tục duyệt đệ quy đến đỉnh tiếp theo. Cập nhật giá trị bottleneck. Nếu giá trị bottleneck dương, điều đó chứng tỏ cạnh vừa duyệt có thể gửi thêm một lượng bottleneck. Cập nhật đường tăng augment path trên đồ thị còn dư G_f.

```
    public void augment(long bottleNeck) {
        flow += bottleNeck;
        residual.flow -= bottleNeck;
    }
```

Tiếp tục đệ quy đến đỉnh tiếp theo:

```
    long bottleNeck = dfs(edge.to, min(flow, rcap));

        // Augment flow with bottleneck value
        if (bottleNeck > 0) {
        edge.augment(bottleNeck);
        return bottleNeck;
        }
```

Sau khi duyệt hết tất cả các đỉnh thỏa mãn, trả về giá trị bottleneck.
Kết luận: phương thức dfs() tìm đường tăng từ s đến t và trả về một giá trị bottleneck của đường tăng đó. Đương nhiên đường tăng đó phải phù hợp và có khả năng gửi một lượng xe cộ từ s đến t.

4.3.2 Tìm giá trị max-flow

Sử dụng vòng lặp `for (long f = dfs(s, INF); f !=0; f = dfs(s, INF))` để duyệt qua tất cả các giá trị bottleneck trên mỗi đường tăng mà phương thức dfs() tìm ra. Tổng của tất cả các giá trị bottleneck đó chính là tổng dung tích lớn nhất có thể truyền qua mạng. Cụ thể đây chính là tổng lượng xe lớn nhất có thể di chuyển từ s đến t trên đồ thị giao thông.

```
public void solve() {
    // Find max flow by adding all augmenting path flows.
    for (long f = dfs(s, INF); f != 0; f = dfs(s, INF)) {
        markAllNodesAsUnvisited();
        maxFlow += f;
    }
}
```

4.3.3 Đánh giá mức độ tắc nghẽn của các cạnh

Sau khi chạy thuật toán, giá trị flow trên mỗi cạnh thay đổi. Dựa trên tỉ lệ $\frac{flow}{capacity}$ của mỗi cạnh, chúng em đánh giá mức độ tắc nghẽn giao thông trên từng cạnh của mạng giao thông.

Tạo ra ba loại cạnh để đánh giá:

- **edgesFree** - các cạnh không tắc nghẽn
- **edgedMedium** - các cạnh tắc nghẽn vừa phải
- **edgesFullCapacity** - các cạnh tắc nghẽn

```
public static List<Edge> edgesFree = new ArrayList<>();
static List<Edge> edgesMedium = new ArrayList<>();
static List<Edge> edgesFullCapacity = new ArrayList<>();
```

Tiến hành phân loại các cạnh sau khi chạy thuật toán.

```
private static void run_test() {
    FordFulkersonDfsSolverAdjacencyList graph = ImportData.getFordFulkersonDfsSolverAdjacencyList(n, s,
        t);
    System.out.println("Max flow:" + graph.getMaxFlow());

    List<Edge>[] g = graph.getGraph();

    for (List<Edge> edges : g) {
        for (Edge e : edges) {
            if (!e.isResidual()) {
                if (e.flow >= 0 && e.flow < (e.capacity / 4)) {
                    edgesFree.add(e);
                } else if (e.flow >= (e.capacity / 4) && e.flow < (e.capacity - e.capacity / 4)) {
                    edgesMedium.add(e);
                } else {
                    edgesFullCapacity.add(e);
                }
            }
        }
    }
}
```

4.3.4 Ưu và nhược điểm

Ưu điểm

- Việc lưu đồ thị **graph** dưới dạng `List<Edge>[] graph` giúp việc truy cập đến từng đỉnh chỉ mất độ phức tạp O(1). Việc này giúp phương thức addEdge() trở nên dễ dàng.
- Đồng thời việc lấy ra đồ thị sau khi chạy thuật toán cũng trở nên dễ dàng hơn vì truy cập tới các đỉnh rất nhanh O(1).

Nhược điểm

- Việc duyệt các đỉnh trong phương thức dfs() có độ phức tạp lớn vì phải truy cập tới tất cả các đỉnh, đồng thời truy cập tới tất cả các cạnh.

CHƯƠNG 5

THIẾT KẾ GIAO DIỆN VỚI JAVA SWING

5.1 Giới thiệu về Java Swing

Java Swing là một bộ công cụ GUI (Graphical User Interface) được cung cấp bởi Java để phát triển các ứng dụng desktop. Được sử dụng rộng rãi, Swing cung cấp các thành phần giao diện người dùng như nút, hộp văn bản, cửa sổ, menu và các thành phần khác để tạo ra giao diện người dùng trực quan. Java Swing được dùng trong dự án này mà không phải công cụ khác vì Swing có những ưu điểm đa nền tảng, cấu trúc linh hoạt, thư viện rộng, ...

5.2 Tạo giao diện bản đồ

5.2.1 Class Maps

Để vẽ được bản đồ, ta cần trích những dữ liệu trong file data và thêm vào class Maps. Ở class Maps, ta tiến hành in điểm, đường lên trên bằng đoạn code sau:

```
protected void paintComponent( Graphics g){
        super.paintComponents ( g );
        Graphics2D g2d = ( Graphics2D ) g;
        try {
            Font font = new Font("Arial", Font.BOLD, 16);
            File file1 = new File ( "F:\\DataStructure&Algorithms\\FinalExam\\src\\Add\\Location.csv" );
            scanner = new Scanner ( file1 );
            while ( scanner.hasNextLine ( ) ) {
                String line = scanner.nextLine ( );
                String[] values = line.split ( "," );
                int lx1 =Integer.parseInt(values[0]) ;
                int lx2 =Integer.parseInt(values[1]) ;
                int x1 = Integer.parseInt ( values[ 4 ] );
                int y1 = Integer.parseInt ( values[ 5 ] );
                int x2 = Integer.parseInt ( values[ 6 ] );
                int y2 = Integer.parseInt ( values[ 7 ] );
                int weight = Integer.parseInt ( values[ 3 ] );
                g2d.setStroke(new BasicStroke(weight));
                g2d.setColor(Color.darkGray);
                g2d.draw(new Line2D.Double(x1,y1,x2,y2));
                g2d.setColor(Color.WHITE);
                g2d.drawOval ( x1,y1,2,2 );
                g2d.drawOval ( x2,y2,2,2);
                g2d.setColor(Color.RED);
                g2d.setFont(font);
                g2d.drawString( String.valueOf ( lx1 ) , x1, y1);
                g2d.drawString( String.valueOf ( lx2 ) , x2, y2);
                }
        } catch (FileNotFoundException ex) {
            throw new RuntimeException ( ex );
        }
```

Ta sẽ được 1 bản đồ cơ bản bao gồm các đỉnh, đoạn đường với độ dài, độ rộng khác nhau, tên thứ tự của đỉnh và đỉnh sẽ được hiện lên.
Tiếp theo ta sẽ chuẩn bị sẵn đoạn code để kích hoạt khi sử dụng Button trong class Model.

```
if(activeFindFF){
\\TODO
}
if(activeFindDT){
\\TODO
}
```

Ta cũng cần hàm để nhận lệnh từ button:

```
public void onFindFFClicked() {
    this.activeFindFF = true;
}
public void onFindDTClicked() {
    this.activeFindDT = true;
}
```

5.2.2 Class View

Ở Class View, ta sẽ tạo hầu hết các nút, ô hiển thị và bố cục của giao diện.

```
public class View extends JFrame {
private JTextField jTextField_start;
private JTextField jTextField_end;
private JTextArea jTextArea_findFF;
private JTextArea jTextArea_findDT;

private JLabel jLabel_answer;

public View () throws HeadlessException {
    super();
    this.model = new Model ();
    this.init();
}
```

Sau khi khởi tạo các đối tượng ban đầu cần có, ta sẽ tiến hành gán chức năng của các đối tượng:

```
private void init ( ) {
    // Name, size of Jframe
    this.setTitle("Map");
    this.setSize(1300,700);
    this.setLocationRelativeTo (null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Font style, font, font size
    Font font = new Font("Times New Roman",Font.BOLD,20);

    //Assign names to Buttons
    JLabel jLabel_start = new JLabel("Start point");
    jLabel_start.setFont ( font );
    jTextField_start = new JTextField ( );
    jTextField_start.setFont ( font );
    //Assign names to Labels
    JButton jButton_findFF = new JButton("Find ways using FF");
    jButton_findFF.setFont(font);

    //Create a menu and add Labels and buttons to the menu
    JPanel jPanel_menu = new JPanel();
```

```
        jPanel_menu.setLayout ( new GridLayout(8,1,1,1));
        jPanel_menu.add( jLabel_start);
        jPanel_menu.add(jTextField_start);
        jPanel_menu.add(jButton_findFF);
        jPanel_menu.add(jTextArea_findFF);

        //layout of the interface
        map = new Maps (this);
        this.setLayout(new BorderLayout());
        this.add ( jPanel_menu, BorderLayout.WEST );
        this.add( map,BorderLayout.CENTER);
        this.setVisible (true);

        //create action listener
        FindWay findWayFF = new FindWay(this);
        jButton_findFF.addActionListener ( findWayFF );
        FindWay findWayDT = new FindWay(this);
        jButton_findDT.addActionListener ( findWayDT );
```

Để trích xuất giá trị đầu vào, tạo giá trị đầu ra, Và truyền giá trị đầu vào cho các hàm thuật toán, ta có đoạn code sau:

```
public int getStart(){
        return Integer.parseInt(jTextField_start.getText ()) ;
    }
public int getEnd(){
    return Integer.parseInt(jTextField_end.getText ());
}
public void findDTs(){
    String finalString1 = this.model.findDT(this.getStart (),this.getEnd());
    String finalString2 = this.model.findDTMin(this.getStart (),this.getEnd());
    this.jTextArea_findDT.setText(finalString1+"\n"+finalString2);
    this.map.onFindDTClicked ();
    repaint();
}
```

5.2.3 Class Model

Sau khi chuẩn bị xong phần giao diện, ta cần hàm kết nối giữa phần Back End và Front End là Class Model.

```
public class Model {}

    //List to paint on map
    public ArrayList <Integer> findDTs(int start, int end){
        Path path = new Path("F:\\DataStructure&Algorithms\\FinalExam\\src\\Add\\Location.csv");
        return path.shortestPath(start, end);
    }

    //String to output
    public String findDTMin(int start,int end) {
    ArrayList<Integer> paths = findDTsMin ( start,end );
    String path = "";
    for(int i = 0; i< paths.size ();i++){
        path+=paths.get ( i )+" ";
    }
    this.ketquaDT = paths;
    return path;
    }

}
```

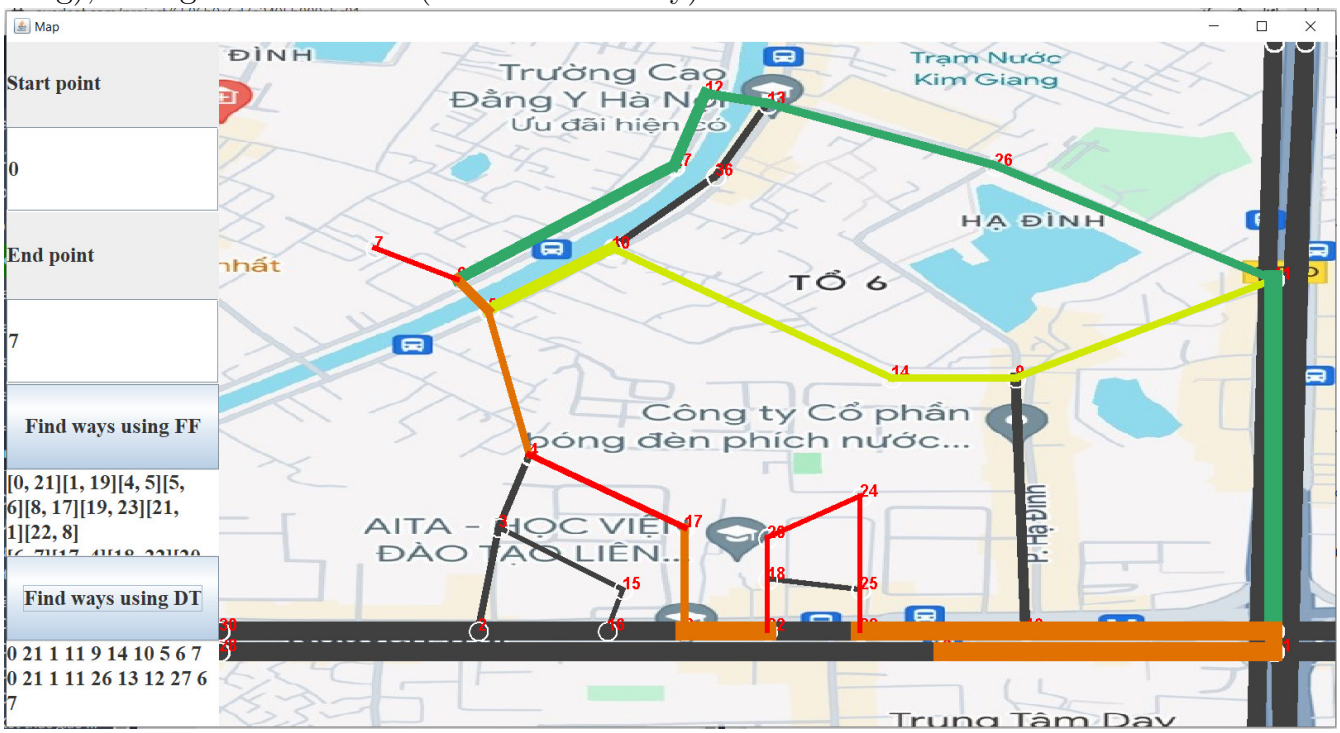
5.2.4 Class FindWay

Bước cuối cùng là tạo hàm để khi ấn nút, chức năng được gán với nút sẽ được thực hiện.

```
    public class FindWay implements ActionListener {
    private View view;
    public FindWay ( View view ) {
        super();
        this.view = view;
    }
    @Override
    public void actionPerformed ( ActionEvent e ) {
        String button = e.getActionCommand ();
        if(button.equals("Find ways using DT")) {
            this.view.findDTs();
        } else if (button.equals("Find ways using FF")) {
            this.view.findFFs();
        }
    }
}
```

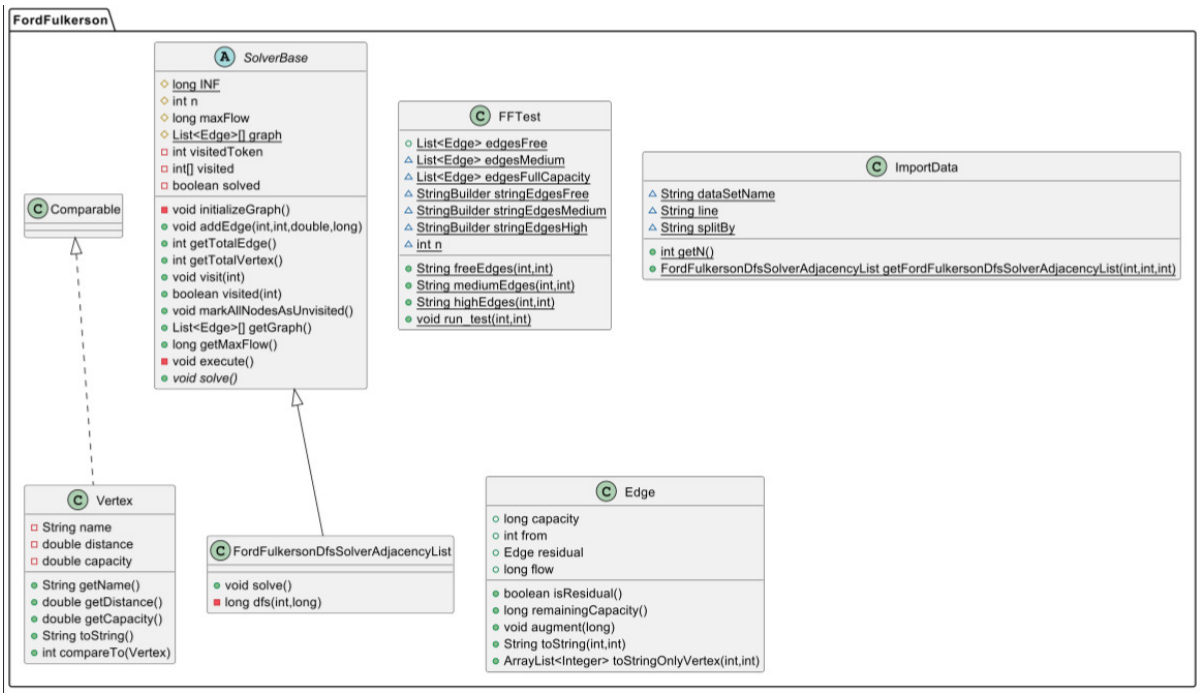
5.2.5 Kết thúc giao diện

Sau khi thêm các Class cơ bản như trên, ta sẽ được một bản đồ gần như hoàn chỉnh. Tiếp theo chỉ cần thêm những đối tượng khác, chỉnh sửa cỡ chữ, kiểu chữ, ...
Ở chương trình này, ô đầu tiên sẽ nhập điểm xuất phát, ô thứ 2 sẽ nhập điểm cần đến. 2 nút kế tiếp sẽ hiện đường đi tắc nhất(màu đỏ), tắc nhì(màu cam), đường đi ngắn nhất(màu vàng), đường đi nhanh nhất(màu xanh lá cây).

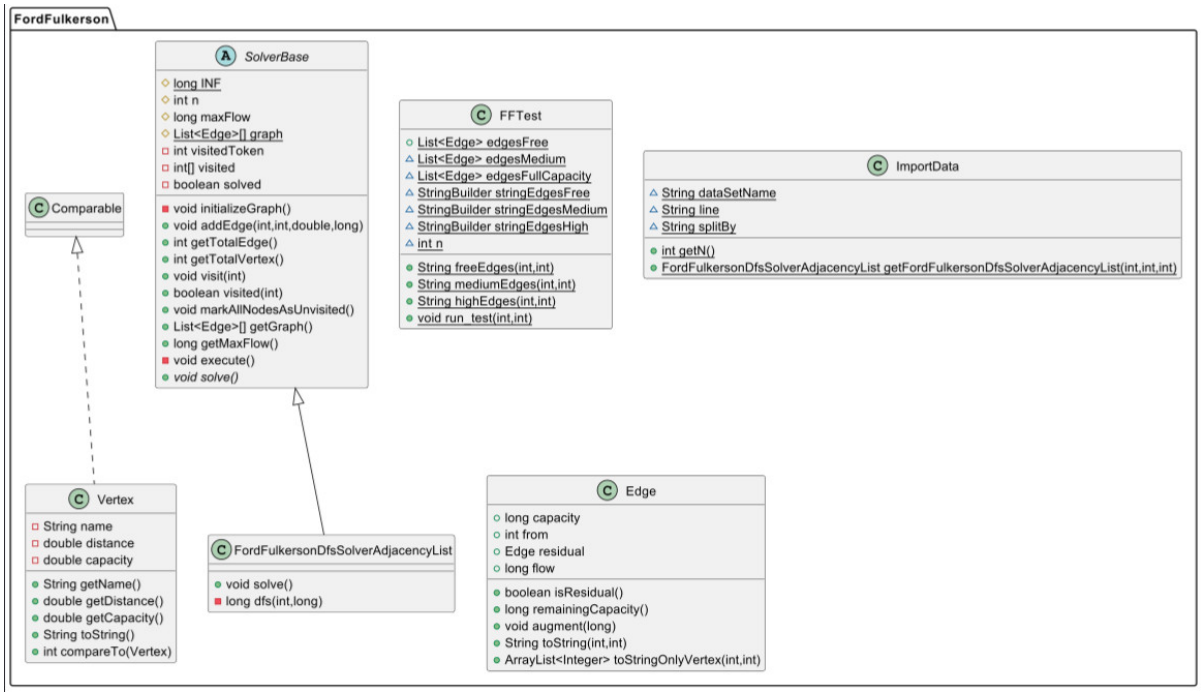


PHỤ LỤC

Hình 5.1: Đồ thị UML cho Package của thuật toán Dijkstra



Hình 5.2: Đồ thị UML cho Package của thuật toán Ford-Fulkerson



Hình 5.3: Đồ thị UML cho Package của giao diện và class chạy chính

TÀI LIỆU THAM KHẢO

[**1**] J. Kleinberg and E. Tardos, Algorithm Design, Pearson, 2005.

[**2**] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser - Data Structures and Algorithms in Java-Wiley (2014).

[**3**] Drake, Peter - Data structures and algorithms in Java-Prentice Hall (2008).

[**4**] A. K. Baruah, N. Baruah, "Minimum Cut Maximum Flow of Traffic in a Traffic Control Problem", International Journal of Mathematical Archive, vol. 4, no. 1, pp. 171-175, 2013.

[**5**] Noraini Abdullah1, Ting Kien Hua, "The Application of the Shortest Path and Maximum Flow with Bottleneck in Traffic Flow of Kota Kinabalu".

[**6**] Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. McGraw Hill, 2006.

[**7**] S. Skiena, "Dijkstra's algorithm". Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.Reading, MA: Addison-Wesley, pp. 225-227, 1990.

[**8**] https://www.cs.princeton.edu/ wayne/kleinberg-tardos/

[**9**] https://iraj.in/