Viaggio in Europa

Shortest Path Game

Marco Grisanti

Sommario

1. Problema

Viaggio in Europa è un gioco da tavola creato nel 1990 dalla Ravensburger. Si gioca su un tabellone che raffigura l'Europa con delle città che sono collegate da diversi percorsi via terra, via mare e via aerea.

Ad ogni giocatore vengono assegnate casualmente una città di partenza e sei città da visitare. In ogni turno si tira un dado, il quale indica di quanto si può spostare la pedina del giocatore in questione nel tabellone.

Ciascun percorso via terra o via mare ha un peso pari a 1. Invece, un percorso via aerea ha un peso che dipende dalla posizione della città. Ogni città che ha un aeroporto è associata ad un quadrante del tabellone. Valgono le seguenti regole:

- Volare per due città appartenenti allo stesso quadrante ha un costo pari a 2
- Volare per due città appartenenti a quadranti adiacenti ha un costo pari a 4.

L'obiettivo di ogni giocatore è quello di visitare tutte e sei le città che gli sono state assegnate e ritornare alla città di partenza nel minor numero di mosse possibili. In altri termini, ogni giocatore deve trovare il cammino minimo che collega la sua città di partenza con lei sei città che gli sono state assegnate. Si noti che la sorgente e la destinazione del cammino minimo devono coincidere.

Sia V l'insieme delle città (vertici) e sia E l'insieme dei percorsi (archi).

Data una sorgente s appartenente a V e dati v_1, v_2, v_3, v_4, v_5 e v_6 anch'essi appartenenti a V, si deve calcolare il percorso migliore che, partendo da s e tornando in s, passi per v_1, v_2, v_3, v_4, v_5 e v_6 non necessariamente in quest'ordine.

2. Lavoro Svolto

È stata sviluppata un'applicazione in Java in grado di risolvere il problema precedentemente illustrato e dare ad ogni giocatore il percorso migliore in modo che la vittoria o meno del gioco dipenda esclusivamente dalla fortuna, ovvero dal punteggio che si ottiene tirando il dado in ogni turno e dalla distanza delle città scelte casualmente.

L'applicazione è formata da quattro classi:

- *Vertex*
- *Edge*
- *WeighedGraph*
- *Main*

La classe Vertex rappresenta la singola città, ovvero il singolo vertice. Ogni vertice è rappresentato da:

- *name*: Stringa che indica il nome della città.
- *airportLocalization*: Intero che indica il quadrante di appartenenza di una città, nel caso in cui sia presente l'aeroporto.
- *adjs*: Lista di archi nella quale sono contenuti tutti gli archi che hanno come vertice sorgente il vertice in questione.
- *d*: Vettore di interi che indica la distanza di ogni altro vertice del tabellone relativamente al vertice in questione.

- *pred*: Vettore di vertici che, per ogni altro vertice v del tabellone, indica il precedente vertice in un eventuale cammino minimo che va dal vertice in questione a v.

```java
public class Vertex {
    private String name;
    private int airportLocalization;
    private LinkedList<Edge> adjs;

    public Integer[] d;
    public Vertex[] pred;

    public Vertex(String name) {
        this.name = name;
        this.airportLocalization = 0;
        d = new Integer[100000];
        pred = new Vertex[100000];
        adjs = new LinkedList<Edge>();
    }

    public Vertex(String name, int airportLocalization) {
        this.name = name;
        this.airportLocalization = airportLocalization;
        d = new Integer[100000];
        pred = new Vertex[100000];
        adjs = new LinkedList<Edge>();
    }

    public String getName() {
        return name;
    }

    public int getAirportLocalization() {
        return airportLocalization;
    }

    public LinkedList<Edge> getAdjs() {
        return adjs;
    }

}
```

Figura 1 - Classe Vertex

La classe *Edge* rappresenta il singolo percorso (arco) che collega una città ad un'altra. Ogni arco è rappresentato da:

- *sourceVertex*: Vertice che indica il vertice di partenza.
- *destinationVertex*: Vertice che indica il vertice di arrivo.
- *w*: Intero che indica il peso dell'arco in questione.

```java
public class Edge {
    private Vertex sourceVertex;
    private Vertex destinationVertex;
    private int w;

    public Edge(LinkedList<Vertex> vertices,
                String sourceVertexName,
                String destinationVertexName,
                int w) {

        for (Vertex v: vertices) {
            if (v.getName().equals(sourceVertexName))
                sourceVertex = v;
            if (v.getName().equals(destinationVertexName))
                destinationVertex = v;
        }
        if (sourceVertex == null || destinationVertex == null) {
            //Errore
            System.exit(1);
        }
        this.w = w;

    }

    public Vertex getSourceVertex() {
        return sourceVertex;
    }

    public Vertex getDestinationVertex() {
        return destinationVertex;
    }

    public int w() {
        return w;
    }

}
```

Figura 2 - Classe Edge

La classe *WeighedGraph* rappresenta un grafo pesato, ovvero un grafo (formato da vertici e archi) dove ad ogni arco è associato un peso. Ogni grafo pesato è rappresentato da:

- *vertices*: Lista di vertici.
- edges: Lista di archi (pesati).

Questa classe contiene anche altri metodi molto utili per l'applicazione che è possibile vedere in dettaglio nel file *WeighedGraph.java.*

La classe *Main* è la classe principale dell'applicazione nella quale viene risolto il problema precedentemente discusso.

Per il calcolo dei cammini minimi si usa l'algoritmo di Dijkstra, il quale, preso in input un vertice, calcola i cammini minimi verso tutti gli altri vertici. A seguire viene riportata l'implementazione:

```
public static void initilizeSingleSource(WeighedGraph g, Vertex s) {
    for (int i = 0; i < g.getVertices().size(); i++) {
        s.d[i] = 100000;
        s.pred[i] = null;
    }
    s.d[g.getVertices().indexOf(s)] = 0;
}
```

Figura 3 – Funzione "initializeSingleSource"

```
public static void relax(WeighedGraph g, Edge e, Vertex s, HashMap<Vertex, Integer> hashMap) {
    if (s.d[g.getVertices().indexOf(e.getDestinationVertex())] > s.d[g.getVertices().indexOf(e.getSourceVertex())] + g.w(e)) {
        s.d[g.getVertices().indexOf(e.getDestinationVertex())] = s.d[g.getVertices().indexOf(e.getSourceVertex())] + g.w(e);
        s.pred[g.getVertices().indexOf(e.getDestinationVertex())] = e.getSourceVertex();
        hashMap.put(e.getDestinationVertex(), s.d[g.getVertices().indexOf(e.getDestinationVertex())]);
    }
}
```

Figura 4 - Funzione "relax"

```
public static void scan(WeighedGraph g, Vertex u, Vertex s, HashMap<Vertex, Integer> hashMap) {
    for (Edge e: u.getAdjs())
        relax(g, e, s, hashMap);
}
```

Figura 5 - Funzione "scan"

```
public static void Dijkstra(WeighedGraph g, Vertex s) {
    initilizeSingleSource(g, s);
    HashMap<Vertex, Integer> hashMap = new HashMap<Vertex, Integer>();
    for (Vertex v: g.getVertices()) hashMap.put(v, s.d[g.getVertices().indexOf(v)]);
    while (!hashMap.isEmpty()) {
        Integer minimumDistance = Collections.min(hashMap.values());
        Vertex v = getKeyByValue(hashMap, minimumDistance);
        hashMap.remove(v);
        scan(g, v, s, hashMap);
    }
}
```

Figura 6 - Funzione "Dijkstra"

Grazie all'algoritmo di Dijkstra siamo in grado di calcolare i cammini minimi di s, v_1, v_2, v_3, v_4, v_5 e v_6. Però, ciò non è sufficiente poiché è necessario capire qual è l'ordinamento migliore di v_1, v_2, v_3, v_4, v_5 e v_6 relativamente al costo. Per fare ciò effettuiamo le permutazioni dei cammini minimi e calcoliamo il costo di ogni permutazione. Successivamente scegliamo la permutazione con costo

minore. In questo modo, abbiamo calcolato il cammino minimo che ha come sorgente e destinazione s e passa per v_1, v_2, v_3, v_4, v_5 e v_6.

```
public static LinkedList<LinkedList<Vertex>> combinePaths(LinkedList<Vertex> vertices) {
    LinkedList<LinkedList<Vertex>> combinedPaths = new LinkedList<LinkedList<Vertex>>();
    combinedPaths.add((LinkedList<Vertex>) vertices.clone());
    int[] p = new int[vertices.size()];
    int i = 1;
    while (i < vertices.size()) {
        if (p[i] < i) {
            int j = ((i % 2) == 0) ? 0 : p[i];
            swap(vertices, i, j);
            combinedPaths.add((LinkedList<Vertex>) vertices.clone());
            p[i]++;
            i = 1;
        }
        else {
            p[i] = 0;
            i++;
        }
    }
    return combinedPaths;
}
```

Figura 7- Computazione delle permutazioni dei cammini minimi

Come si può vedere di seguito, si applica l'algoritmo di Dijkstra sia alla sorgente s che ai vertici dai quali si intende passare. Successivamente, si effettuano le permutazioni dei vertici ottenendo $n!$ cammini, posto che n sia il numero dei vertici da visitare. Ad ogni cammino si aggiunge la sorgente s sia all'inizio che alla fine. Per concludere, viene calcolato il cammino minimo fra tutti gli $n!$ cammini (minimi a loro volta).

```
public static void playGame(WeighedGraph g, Vertex s, LinkedList<Vertex> vertices) {
    Dijkstra(g, s);
    for (Vertex v: vertices) Dijkstra(g, v);

    LinkedList<LinkedList<Vertex>> combinedPaths = combinePaths(vertices);
    for (LinkedList<Vertex> path: combinedPaths) {
        path.addFirst(s);
        path.addLast(s);
    }

    LinkedList<Vertex> minimumPath = combinedPaths.get(0);
    int miniumDistance = 100000;
    for (int i = 0; i < combinedPaths.size(); i++) {
        int currentDistance = 0;
        for (int j = 0; j < combinedPaths.get(i).size() - 1; j++) {
            currentDistance = currentDistance + combinedPaths.get(i).get(j).d[g.getVertices().indexOf(combinedPaths.get(i).get(j + 1))];
        }
        if (currentDistance < miniumDistance) {
            minimumPath = combinedPaths.get(i);
            miniumDistance = currentDistance;
        }
    }

    for (int i = 0; i < minimumPath.size() - 1; i++) printPath(g, minimumPath.get(i), minimumPath.get(i + 1));
}
```

Figura 8 - Funzione "playGame"

Nel caso in cui si voglia compilare l'applicazione è possibile farlo con il comando:

$$javac\ *.java$$

L'applicazione può essere eseguita passandogli i parametri nel modo seguente:

$$java\ Main\ <Sorgente><Città1><Città2>\ \ldots\ <CittàN>$$

3. Conclusioni

To Do

4. Bibliografia

To Do