UFR MATHÉMATIQUES ET INFORMATIQUE

Université de Paris

ALGORITHMIQUE AVANCÉE

IF05X040

Travaux Dirigés 6-7

Prof: Nicolas LOMÉNIE
Author: Letao WANG

November 29, 2021

Partie exercices

Exercice 2.7.

Déterminer un arbre couvrant de poids minimal pour le graphe de la figure 2.3

Réponse. On peut utiliser l'algo de Prim et Kruskal, les resultat sont meme, le poids minimal est 12.

Exercice 2.8.

On considère un graphe valué G et un ensemble U de sommets. Proposer un algorithme qui détermine un arbre couvrant de poids minimal parmi ceux tels que tous les sommets de U sont des feuilles.
Démontrer sa validité et préciser sa complexité.

Réponse. Au premier il faut verifier si c'est possible de determine un arbre couvrant de poids minimal avec les sommets de U sont des feuilles. Si graphe G - U est pas connexe, c'est impossible de rendre les sommets de U d'etre feuilles.
On suppose G - U est connexe:

Methode 1. On peut modifier l'algo Kruskal:

```
1  G un graphe
2  U ensemble de sommets
3  F vide
4
5  begin while G est pas vide:
6          E = la arete poids minimale de G
7          F = F U E
8          begin if F contient un cycle or
9                   un sommet de U avoir deuxieme arete dans F or
10                  les deux sommet de E sont dans U:
11                  F = F - E
12         end if
13         G = G - E
14 end while
```

Validation: on ajoute la situation limite, mais l'algo est juste comme Kruskal.

Complexite: $O(m \log m)$

Methode 2. On fait T est l'arbre couvrant de poids minimale de G - U.
Ensuite on relie les sommets de U et T avec poids minimale.

Validation: parce que les sommets de U sont relie finalement, evidement ce sont tout feuilles.
Et T est arbre couvrant poids minimale, les aretes entre U et T sont poids minimales aussi, le resultat est minimale.
Pour complexite: complexite de algo Kruskal + algo de relie

Complexite: $O(m \log m + m)$

Exercice 2.9.

Appliquer l'algorithme de Dijkstra au graphe de la figure 1.2 pour déterminer le chemin de poids minimal entre u et v.

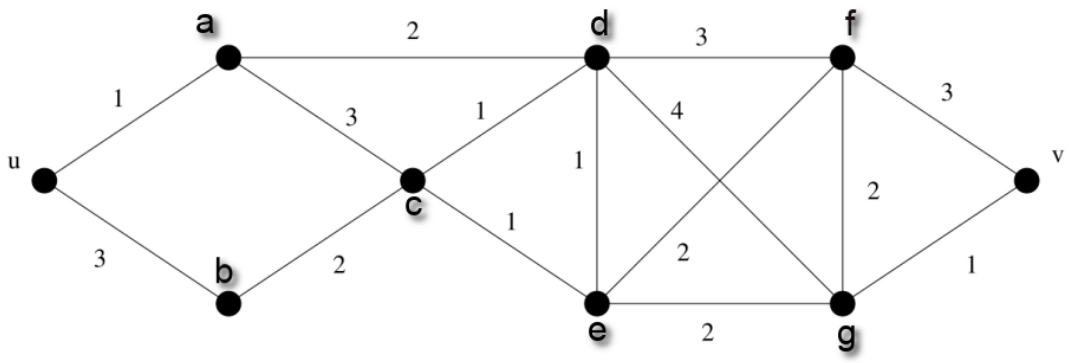


FIGURE 2.4 – .

Réponse.

l'algorithme de Dijkstra

Sommet	u	a	b	c	d	e	f	g	v
Distance	0	1	3	∞	∞	∞	∞	∞	∞
Distance	0	1	3	4	3	∞	∞	∞	∞
Distance	0	1	3	4	3	4	6	7	∞
Distance	0	1	3	4	3	4	6	6	∞
Distance	0	1	3	4	3	4	6	6	7
Distance	0	1	3	4	3	4	6	6	7

Exercice 2.10

A partir de l'algorithme de Dijkstra, construire un algorithme qui détermine le diamètre de chaque composante connexe d'un graphe. Quelle en est la complexité ?

Réponse. On fait l'algo Dijkstra pour tous les sommets dans chaque composante connexe, retourner le plus grand nombre.
Complexity : n * algo Dijkstra (on considere c'est $O(m + nlogn)$ à l'aide de tas de Fibonacci)

Donc c'est $O(n(m + nlogn))$

Exercice 3.1.

Les graphes de la figure 2.4 contiennent-ils un chemin eulérien, un cycle eulérien, un chemin hamiltonien, un cycle hamiltonien ?

Réponse.

	G1	G2	G3
Chemin eulérien	true	true	false
Cycle eulérien	false	true	false
Chemin hamiltonien	true	false	true
Cycle hamiltonien	true	false	false

Exercice 3.2.

Résoudre le problème du postier sur le graphe de la figure 3.2

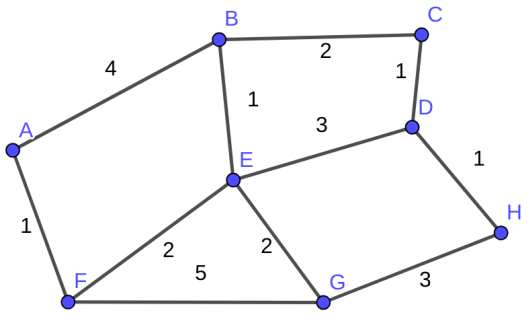


FIGURE 3.2

Réponse. Au premier, il faut trouver les sommets qui degre est impair. Donc on a sommet F, B, G, D. Ensuite on construire le Figure 3.2a

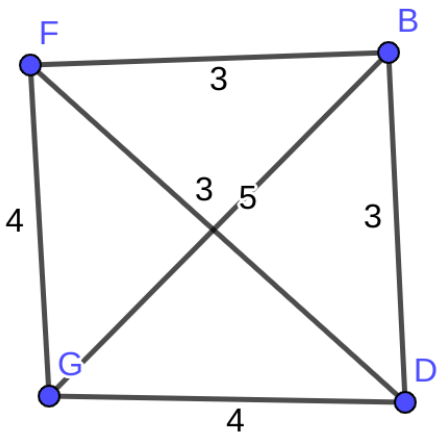


Figure 3.2a

Selon le Figure 3.2a, on veut connecter les 4 sommets avec le minimal poid, on peut choisir Edge(F, G) et Edge(B, D).

On peut ajouter un Edge(B, D) dans le Figure 3.2, donc il y a seulement 2 sommets impaire, on peut trouver un chemin eulerien.

Le poids minimal total est l'addition de tous les poids de Figure3.2 et Edge(F,G), Edge(B,D).

Tous les poids de Figure3.2 : $1+4+2+1+1+3+2+2+5+1+3=25$
Avec les 2 autre edges : $25+3+4=32$

Donc le poids minimal total est **32**.

Partie codes

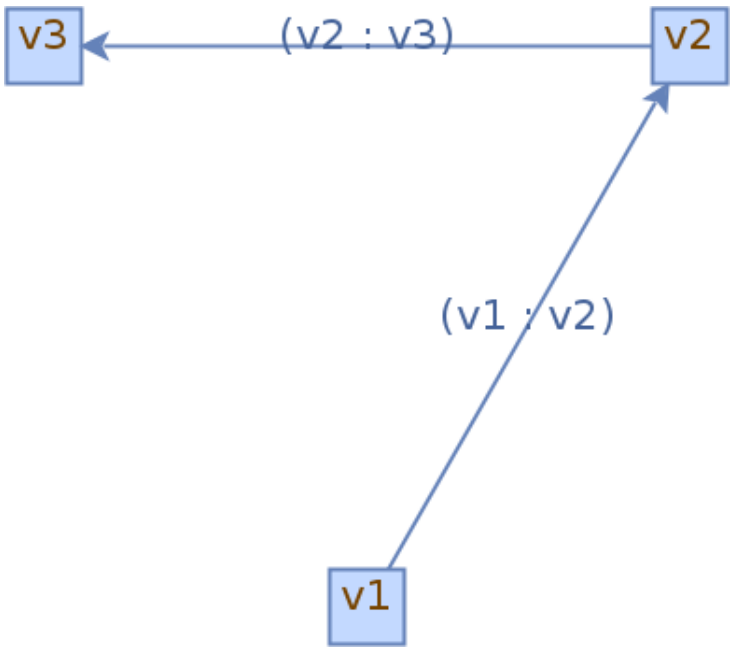
TP6.B. Visualisation de Graphes

JGraphT. Je utilise le bibliotheque JGraphT pour visualiser graphe. Une partie de mon code d'affichage.

Listing 1: GrapheGenerate

```
public class GraphGenerate {

    DefaultDirectedWeightedGraph<String, DefaultEdge> g;

    public GraphGenerate() {}

    public void createGraph() {
        g = new DefaultDirectedWeightedGraph<>
        (DefaultEdge.class);

        g.addVertex("v1");
        g.addVertex("v2");
        g.addVertex("v3");

        g.addEdge("v1", "v2");
        g.addEdge("v2", "v3");
    }

    public void visualiser() throws IOException {
        JGraphXAdapter<String, DefaultEdge> graphAdapter =
        new JGraphXAdapter<>(g);

        mxIGraphLayout layout =
        new mxCircleLayout(graphAdapter);

        layout.execute(graphAdapter.getDefaultParent());
        BufferedImage image =
        mxCellRenderer.createBufferedImage
        (graphAdapter, null, 2, Color.WHITE, true, null);

        File imgFile = new File("src/main/java/graph.png");
        ImageIO.write(image, "PNG", imgFile);
    }
}
```

Voici le resultat du code:



Neo4j. Neo4j est une base de données non relationnelle, une base de données graphique native. Dans une base de données relationnelle normale, la recherche d'un élément de données nécessite souvent l'interrogation de plusieurs tables de la base de données, en particulier pour les grands projets qui nécessitent de nombreuses et longues requêtes sautantes. Les performances de ce type de base de données sont réduites dans ce scénario. La base de données de graphes introduit le concept de Edge entre les Vertex, ce qui améliore considérablement les performances dans ce scénario.

Neo4j Java

Listing 2: Neo4j avec Java exemple

```
...
firstNode = tx.createNode();
firstNode.setProperty( "message", "Node1 " );
secondNode = tx.createNode();
secondNode.setProperty( "message", "Node2" );

relationship = firstNode.createRelationshipTo
( secondNode, RelTypes.KNOWS );

relationship.setProperty( "message", "relation12 " );
...
```

Voici le resultat dans le terminale:
Node1 relation12 Node2

Neo4j browser Neo4j browser est un interface graphique pour modifier les datas de database. Il utilise Cypher au lieu de SQL language.

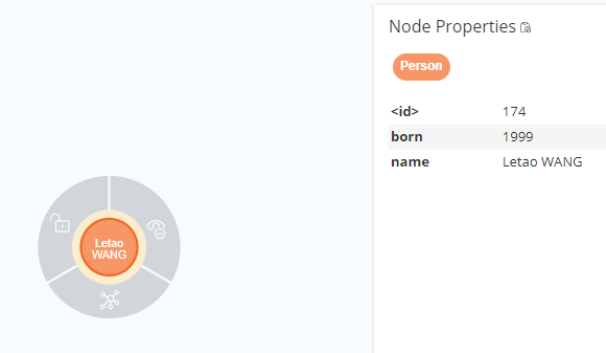
Create record:

CREATE (Letao:Person name:'Letao WANG', born:1999)

```
CREATE (UniversityOfParis:University
{title:'University of Paris', released:2019,
        tagline:'Welcome to the University of Paris'})
CREATE (Taole:Person {name:'Taole WANG', born:1999})
CREATE (Stan:Person {name:'Stan Ma', born:1999})
CREATE (Paul:Person {name:'Paul Hu', born:1999})

CREATE (Taole)-[:STUDIED_IN {roles:['Student']}]->
(UniversityOfParis)
```

Query:

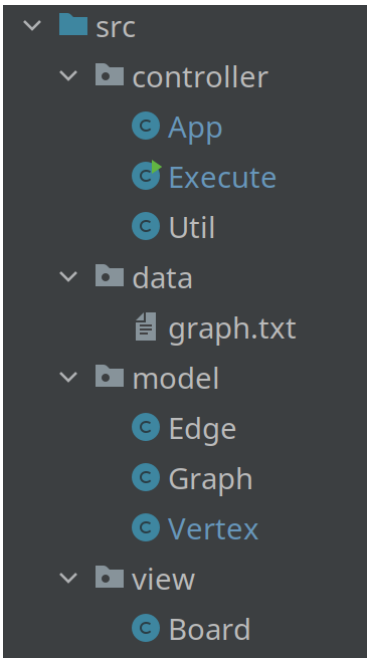
MATCH (Letao:Person name: "Letao WANG") RETURN Letao



Et on peut identifier la relation entre Person et University aussi, par exemple on cherche les personnes qui etudier a Universite de Paris

TP7 PartieB

Algo Dijkstra. Apres j'ai telecharge les fichiers, j'ai realise que le projet est ecrire par Java, mais le style est tres comme language C. Donc au premier je reformule un peu la structure du projet, un peu plus comme MVC (Bien sur je vais continuer le modifier).



Aussi encapsulation, c'est a dire ajouter "pivate" avant les attributs

Certaines methodes importants

Get Neighbors

```
1      /**
2      * For getting all max to 8 neighbors of this vertex
3      * @param line line of source
4      * @param col col of source
5      * @param nlines number of lines
6      * @param ncols number of cols
7      * @return list of neighbors vertex
8      */
9     public static ArrayList<Integer> getNeighborDest
10        (int line, int col, int nlines, int ncols)
```

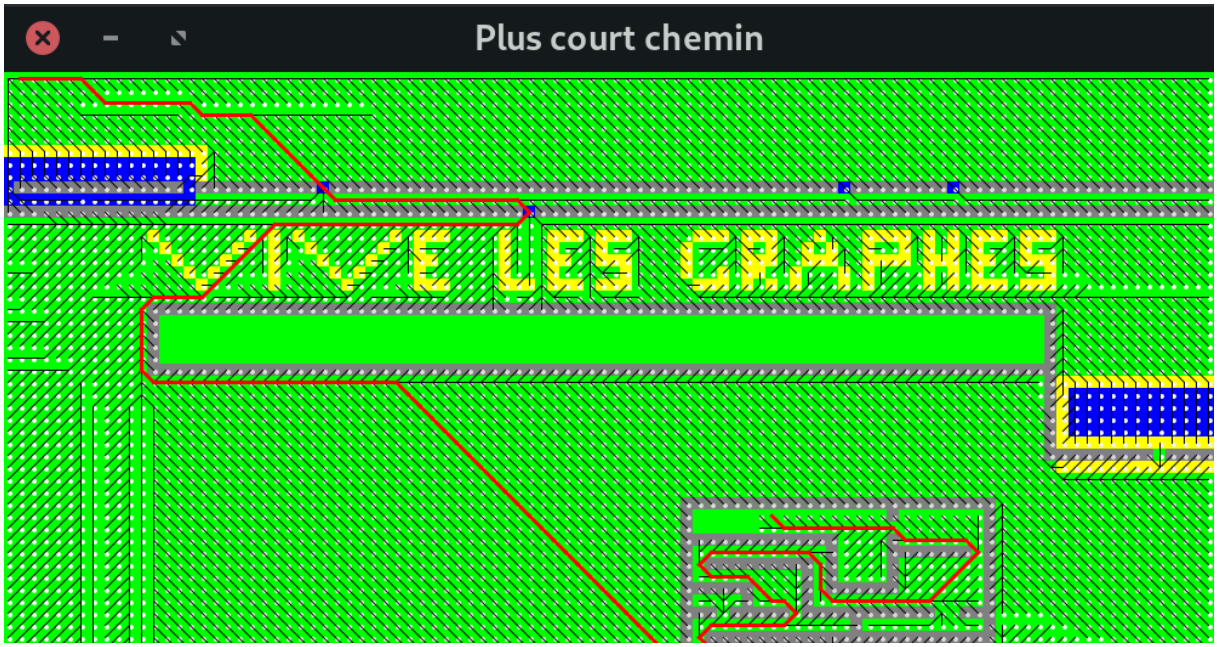
Set weights of Edges

```
        /* direction horizontally or vertically */
        if (Math.abs(source - dest) == 1 ||
                Math.abs(source - dest) == ncols) {
                weight = (timeDest + timeSrc) / 2;
        }
        /*  direction  diagonal */
        double weight12 = Math.sqrt(Math.pow(weight1, 2)
        + Math.pow(weight2, 2));

        double weight34 = Math.sqrt(Math.pow(weight3, 2)
         + Math.pow(weight4, 2));

        weight = Math.min(weight12, weight34);
```

Partie de algo Dijkstra

```
        /* pour tous ses voisins, on verifie si
                on est plus rapide en passant par ce noeud. */
        for (int i = 0; i < graph.getVertexlist().get(min_v).
                        getAdjacencylist().size(); i++) {
                        int to_try =
                graph.getVertexlist().get(min_v).
                getAdjacencylist().get(i).getDestination();

                /* to_try node timeFromSource += weight */
                /* code etc. */
        }
```

Resultat
Parce que la partie graphe de projet est deja fini, je n'ai pas besoin d'ecrire les interface graphique.



les lines noir sont des Edges, et le line rouge est le chemin minimal distance entre Vertex start a Vertex end.

Et dans le terminale

```
Done! Using Dijkstra:
    Number of nodes explored: 4156
    Total time of the path: 296.86466840874294
```

Partie Algo Dijkstra fini.