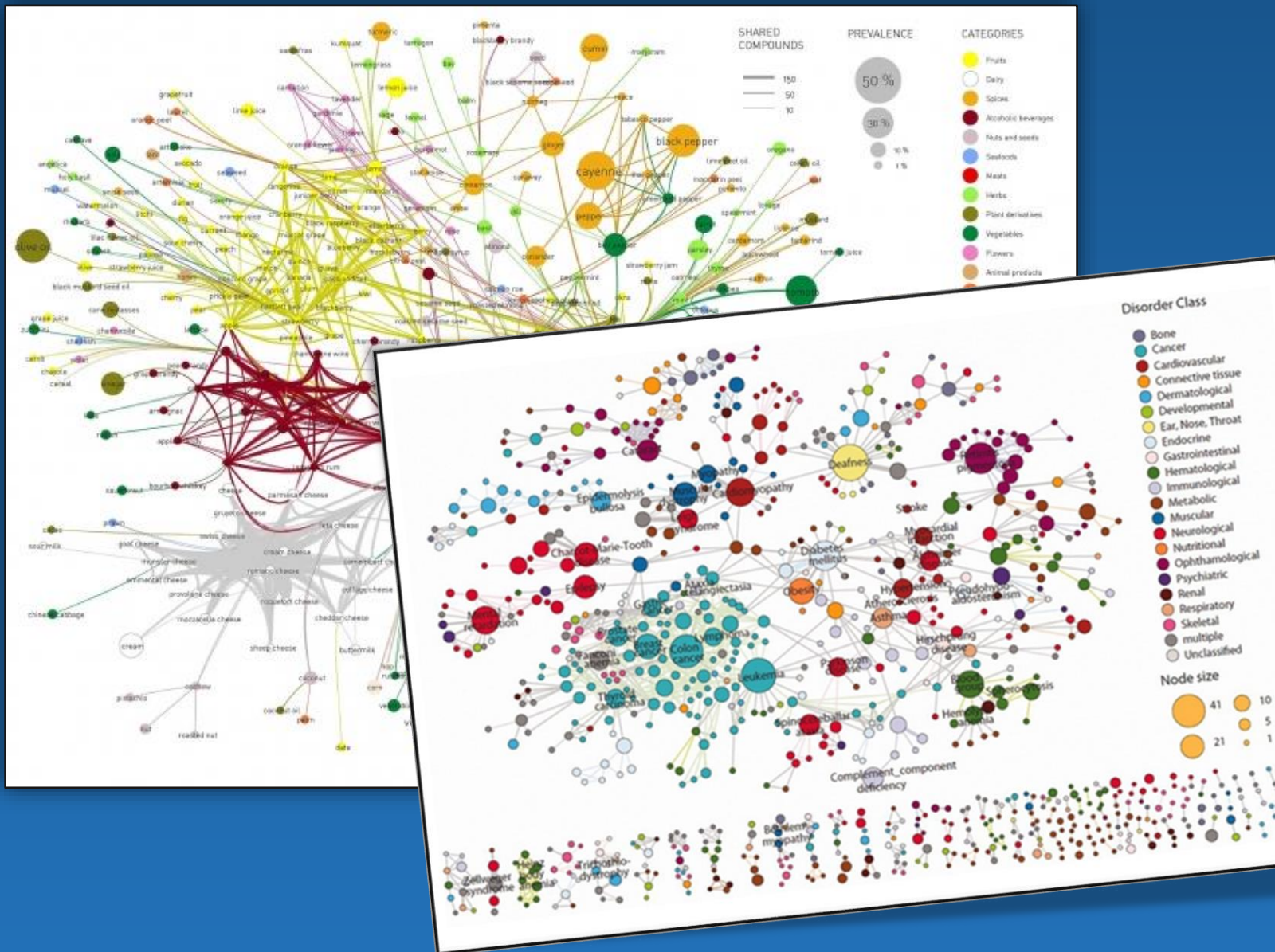


# Compte rendu



Nom

NAMOLARU

Prénom

Leonard

Numéro étudiant

51704115

Groupe de TD

4

<b>Table des matières .....</b>	<b>2</b>
<b>TP 4 • Arbre couvrant Minimal .....</b>	<b>3</b>
Prim sur une matrice d'adjacences .....	3
Kruskal sur tableau d'arêtes .....	4
<b>TP 5 • FIFO et LIFO en Java .....</b>	<b>5</b>
Structures : Ensembles .....	5
Structures : Dictionnaires .....	5
Structures : Files d'attente (FIFO) .....	5
Structures : Structures Pile (LIFO) .....	5
<b>TP 6 • Visualisation de Graphes .....</b>	<b>6</b>
Les moyens offerts en Java pour visualiser des graphes .....	6
Visualisation de façon plus large que Java .....	6
<b>TP 7 , TP 8 • Recherche du plus court chemin .....</b>	<b>7</b>
Algorithme de Dijkstra .....	7
L'algorithme A* .....	8
Les performances de Dijkstra et d'A* sur différentes cartes .....	8
<b>TP 9 • Extraction du min ou du max en Java .....</b>	<b>9</b>
Class PriorityQueue<E> .....	9
Méthode Quicksort en Java .....	9
Performances .....	10

## TP 4

## Arbre couvrant minimal

## # Prim sur une matrice d'adjacences

L'algorithme de Prim est un algorithme qui calcule un arbre couvrant minimal dans un graphe connexe valué et non orienté.

L'algorithme consiste à faire croître un arbre depuis un sommet. On commence avec un seul sommet puis à chaque étape, on ajoute une arête de poids minimum ayant exactement une extrémité dans l'arbre en cours de construction. En effet, si ses deux extrémités appartenaient déjà à l'arbre, l'ajout de cette arête créerait un deuxième chemin entre les deux sommets dans l'arbre en cours de construction et le résultat contiendrait un cycle.

Source : Wikipedia - CC BY-SA 3.0

Input : Un graphe connexe non-orienté valué  $G$  et un sommet  $v$

**Prim( $G, v$ )**

$T = \{v\}$

Initialiser  $T$  avec un sommet de  $G$  qu'on choisit

$G/T$  sommets qui se trouve dans  $G$  et pas dans  $T$

while il existe des arêtes de  $T$  vers  $G \setminus T$  do

tant que (  $|V(T)| < |V(G)|$  )

    | Trouver l'arête  $e$  de poids minimal de  $T$  vers  $G \setminus T$   
    |  $T = T + e$

$T = T \cup \{e\}$

$e = (u, v)$   
 = arête de poids min ayant  $u \in V(T)$  et  $v \in V(G) - V(T)$

end

return  $T$

Output :  $T$  l'arbre couvrant de  $G$  de poids minimal

Complexité de  $O(mn)$

## # Kruskal sur tableau d'arêtes

**L'algorithme construit un arbre couvrant minimum**  
en sélectionnant des arêtes par poids croissant.

Plus précisément,

L'algorithme considère toutes les arêtes du graphe par poids croissant  
(en pratique, on trie d'abord les arêtes du graphe par poids croissant)  
et pour chacune d'elles, il la sélectionne si elle ne crée pas un cycle.

**Source :** Wikipedia - CC BY-SA 3.0

**Kruskal( $G$ )**

**Input :** Un graphe connexe non-orienté valué  $G$

$L = E(G)$

Une liste des arêtes ordonnées par poids croissant

$F = \emptyset$

L'arbre résultant

**while**  $F$  n'est pas un arbre **do**

$e =$  première arête de  $L$

$e = head(L)$  ,  $e$  est une arête

**if** les extrémités de  $e$  ne sont pas dans le même arbre de  $F$  **then**

$F = F + e$

$F = F \cup \{e\}$

**end**

    Supprimer  $e$  de  $L$

$L = L \setminus \{e\}$

**end**

**return**  $F$

**Output :**  $F$  l'arbre couvrant de  $G$  de poids minimal

**Si** les extrémités de  $e$  ne  
sont pas dans le même arbre  
de  $F$

**Si**  $F \cup \{e\}$   
ne crée pas un cycle

Examiner dans l'ordre chacune des  
arêtes  $e = \{x, y\}$

**S'il n'existe pas de chaîne de  $x$  à  $y$  dans  $F$**

**Alors**  $F = F + e$

Complexité de  $O(m \cdot \log m)$

La fonction `kruskal()` dans le fichier `TP4_Kruskal.c` utilise le tri par sélection

**Complexité du tri par sélection :**

Comparaisons  $O(m^2)$  | Echanges  $O(m)$

## TP 5

## FIFO et LIFO en Java

**FIFO** - First in, First out ; **LIFO** - Last in, First out

## mini doc

## Structures Ensembles

Interface Collection<E>  
Interface List<E>

→ Class LinkedList<E>  
→ Class ArrayList<E>  
→ Class Vector<E>

Interface Collection<E>  
Interface Set<E>

→ Class HashSet<E>

## Structures Dictionnaires

Interface  
Map<K,V>

→ Class HashMap<K,V>

## Structures Files d'attente (FIFO)

Interface Collection<E>

Interface Queue<E>

## Structures Pile (LIFO)

Interface Collection<E>

Interface List<E>

Class Vector<E>

Class Stack<E>



## TP 6


## Visualisation de Graphes

## Les moyens offerts en Java pour visualiser des graphes

<b>GraphStream</b> <a href="https://graphstream-project.org/">https://graphstream-project.org/</a>	→ Bibliothèque Java de gestion de graphes. → Se concentre sur les aspects dynamiques des graphes. → Modélisation de réseaux d'interactions dynamiques de différentes tailles.
<b>graphviz-java</b> <a href="https://github.com/nidi3/graphviz-java">https://github.com/nidi3/graphviz-java</a>	→ Graphviz est un logiciel de visualisation de graphes. → Graphviz-java : Ecrire du code JAVA et le convertir en graphes.
<b>JUNG</b> <a href="http://jrtom.github.io/jung/">http://jrtom.github.io/jung/</a>	→ JUNG - fournit un langage commun pour la modélisation, l'analyse et la visualisation de données pouvant être représentées sous forme de graphes ou de réseau. → La base en Java permet aux applications basées sur JUNG d'utiliser l'API Java.
<b>JavaFX SmartGraph</b> <a href="https://github.com/brunomnsilva/JavaFXSmartGraph">https://github.com/brunomnsilva/JavaFXSmartGraph</a>	→ JavaFX : API Java pour la construction d'interfaces graphiques. → JavaFX SmartGraph : une bibliothèque de visualisation de graphes. → Peuvent être stylisés via CSS.

## Visualisation de façon plus large que Java

Pourrait fournir un fichier de sortie exploitable par d'autres plateformes/environnement de développement.

<b>Neo4j</b> <a href="https://neo4j.com/">https://neo4j.com/</a> 	→ Un système de gestion de base de données de graphes.) → Implémenté en Java → Accessible à partir de logiciels écrits dans d'autres langages.
------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------

# Algorithme de Dijkstra

Pseudo-code // Cours Prof. Birmelé ; Commentaires : Wikipedia - CC BY-SA 3.0

## Dijkstra( $G, u$ )

$V(T) = \text{noeud\_visites}$

```

T = {u}
for v ∈ V(G) do
  dist_prov(v) := ∞; pere(v) := ∅
end
dist_finale(u) := 0; dernier_ajout := u

```

Pour chaque sommet  $v$  de  $G$  faire

distance provisoire

distance finale

dernier ajout à  $T$

Initialisation

Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle.

while  $V(T) \neq V(G)$  do      Tant que  $V(T)$  [ = noeuds\_visites ] ne contient pas tous les nœuds de  $G$  faire

```

for v voisin de dernier_ajout do
  if dist_finale(dernier_ajout) + ω(dernier_ajout, v) < dist_prov(v)
  then
    dist_prov(v) := dist_finale(dernier_ajout) + ω(dernier_ajout, v)
    pere(v) := dernier_ajout
  end
end

```

On met à jour les distances des sommets voisins de celui ajouté

La nouvelle distance du sommet voisin est le minimum entre la distance existante et celle obtenue en ajoutant le poids de l'arc entre sommet voisin et sommet ajouté à la distance du sommet ajouté.

```

end
Selectionner v ∉ V(T) tel que dist_prov est minimum
Ajouter (pere(v), v) à T
dist_finale(v) := dist_prov(v)
dernier_ajout := v

```

Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe

On considère un graphe valué  $G$  et un sommet  $u$  de  $G$ . Construire un arbre couvrant  $T_u$  tel que, pour tout sommet  $v$ , la distance de  $u$  à  $v$  est la même dans  $G$  et dans  $T_u$ .

**Output :** Un arbre couvrant  $T$

et la distance  $D(v) = d_G(u, v)$  pour tout  $v$

*La complexite est polynomiale.*

*Elle est en  $O(n^2)$  pour un code naif,  
peut etre reduite a  $O((m+n)\log n)$  a l'aide de tas binaire  
et meme  $O(m + n\log n)$  a l'aide de tas de Fibonacci*

## L'algorithme A\* [ à prononcer A star ou A étoile ]

Peut être vu comme une extension de l'algorithme de Dijkstra.

La différence se fait dans le choix du nœud à explorer. Au lieu de choisir celui ayant la plus petite distance temporaire, on sélectionne en fonction de la somme entre la distance temporaire et une heuristique (par exemple, une distance estimée au nœud d'arriver) à définir

**Source :** Consignes.pdf

*La complexité temporelle de A\* dépend de l'heuristique.*

**Dans notre cas**

*l'heuristique est la distance estimée au nœud d'arriver ( distance euclidienne)*

Wikipedia - CC BY-SA 3.0

### Les performances de Dijkstra et d'A\* sur différentes cartes

	A*		Dijkstra	
	Total time of the path	Number of nodes explored	Total time of the path	Number of nodes explored
graph_test1.txt	56.87	628	56.87	1000
graph_test2.txt	60.385	257	60.385	765
graph_test3.txt	56.87	244	56.87	943
graph.txt	302.610	4130	302.610	4156



## TP 9

## Extraction du min ou du max en Java

## Class PriorityQueue&lt;E&gt;

File de priorité

- Les éléments d'une file de priorité sont classés selon leur ordre naturel, ou par un comparateur de type *Comparator*.
- La tête de la file de priorité est l'élément le plus petit.

```
import java.util.PriorityQueue;
public class PriorityQueueTest {
    public static void main(String[] args) {
        // Creation d'une file de priorité avec une capacité initiale de 3.
        PriorityQueue<Integer> fileDePriorite = new PriorityQueue<Integer>(3);

        // Ajout d'un nouvel élément une file de priorité.
        fileDePriorite.add(500);
        fileDePriorite.add(1);
        fileDePriorite.add(200);

        // Extraction du min
        System.out.println( "min = " + fileDePriorite.poll() );

        // Extraction du max
        Integer element = null, elementPrecedent = null;
        while( (element = fileDePriorite.poll()) != null )
            elementPrecedent = element;
        System.out.println( "max = " + elementPrecedent );
    }
}
```

## Méthode Quicksort en Java

Tri rapide

## Arrays.sort(int[] a)

- L'algorithme de tri est un tri rapide à double pivot. Généralement plus rapide que les implémentations de tri rapide traditionnelles (à un pivot).

[ Source : <https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html> ]

```
import java.util.Arrays;
public class QuickSortTest {
    public static void main(String[] args) {
        // Creation d'un tableau.
        int[] tableau = {500 , 1 , 200};
        // Tri rapide.
        Arrays.sort( tableau );
        // Extraction du min
        System.out.println( "min = " + tableau[0] );
        // Extraction du max
        System.out.println( "max = " + tableau[tableau.length - 1] );
    }
}
```

# Performances

## Extraction du min ou du max

	Arrays.sort	Class PriorityQueue<E>
	$O(n \cdot \log(n))$ L'algorithme de tri est un tri rapide à double pivot. Cet algorithme offre des performances $O(n \log(n))$ sur de nombreux ensembles de données et est généralement plus rapide que les implémentations traditionnelles du tri rapide (à un pivot).	$O(\log(n))$ pour les méthodes d'ajout et de retrait de la file.
Extraction que du min	$O(n \cdot \log(n))$	$O(\log(n))$
Extraction que du max	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$
Extraction du min ET du max	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$

Source 1 : <https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>

Source 2 : <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/PriorityQueue.html>