

École Polytechnique

INF411

Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

Bloc 8 / lundi 16 octobre 2017

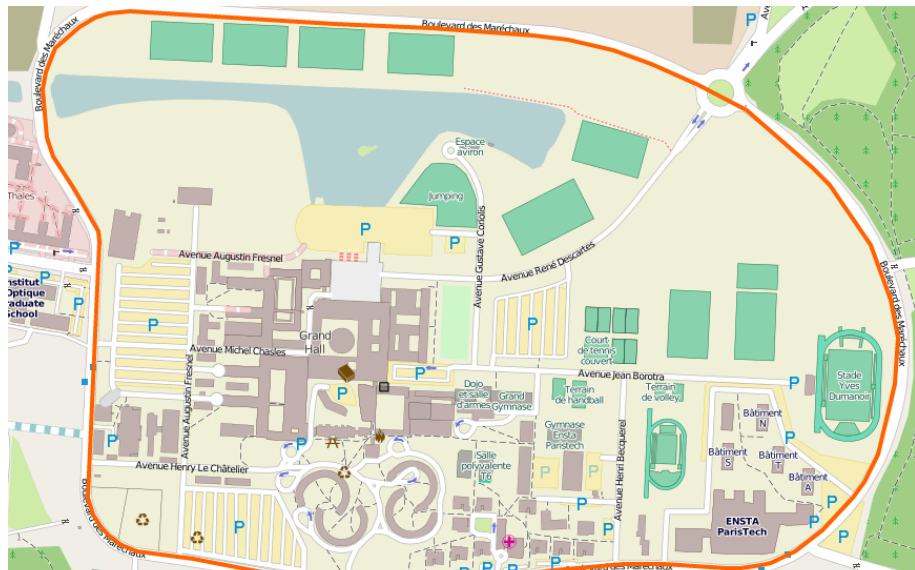
une carte routière est un exemple de graphe

aujourd'hui, on va programmer un (embryon de) navigateur GPS

cartographie libre : www.openstreetmap.org



cartographie libre : www.openstreetmap.org



les données cartographiques sont au format XML

elles décrivent des sommets, comme

```
<node id="243494495" lat="48.9319569" lon="2.1654713">  
</node>
```

ou encore

```
<node id="80953646" lat="48.8582320" lon="2.2945504">  
  <tag k="name" v="Tour Eiffel"></tag>  
  <tag k="tourism" v="attraction"></tag>  
</node>
```

chaque sommet est identifié par un numéro unique (attribut id)

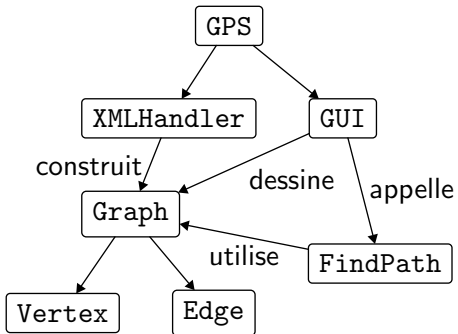
et des chemins

```
<way id="22815079">  
  <nd ref="243494495"></nd>  
  <nd ref="245446883"></nd>  
  <nd ref="245447534"></nd>  
  <nd ref="245446753"></nd>  
  <nd ref="245446884"></nd>  
  <tag k="highway" v="residential"></tag>  
  <tag k="name" v="rue de la Paix"></tag>  
</way>
```

ici 4 arcs reliant 5 sommets identifiés par leur numéro (attribut ref)

c'est un graphe (orienté)





construction du graphe

```
class Vertex {  
    final int id;  
    final double lat, lon;  
    ...  
}
```

```
class Edge {  
    final Vertex src, dst;  
    final String name;  
    ...  
}
```

```
class Graph {  
    private static final HashMap<Vertex, List<Edge>> adj;  
    ...  
}
```

on utilise la bibliothèque Java `org.xml.sax`

le principe est de fournir des méthodes à appeler sur chaque

- balise ouvrante (`<node>`, `<way>`, etc.)
- balise fermante (`</node>`, `</way>`, etc.)

on les fournit en redéfinissant les méthodes d'une classe `DefaultHandler`

```
class XMLHandler extends DefaultHandler {  
    @Override  
    public void startElement(...) { ... }  
  
    @Override  
    public void endElement(...) { ... }  
}
```

une instance sera passée à la fonction de lecture

```
parser.parse(file, new XMLHandler());
```

```
<node id="80953646" lat="48.8582320" lon="2.2945504">
```

```
public void startElement(..., String name, Attributes a) {  
    if (name.equals("node")) {  
        int id      = Integer.parseInt (a.getValue("id" ));  
        double lat  = Double.parseDouble(a.getValue("lat"));  
        double lon  = Double.parseDouble(a.getValue("lon"));  
        Vertex.create(id, lat, lon);  
    } else ...  
}
```

dans la classe Vertex, on fournit

```
// enregistre le sommet de numéro id  
static void create(int id, double lat, double lon)
```

et

```
// retrouve le sommet de numéro id  
static Vertex fromId(int id)
```

```
<way id="22815079">  
  ...
```

on se donne une liste pour stocker les sommets

```
LinkedList<Vertex> way = null;
```

qu'on initialise quand on voit la balise ouvrante <way>

```
public void startElement(..., String name, Attributes a) {  
  ...  
} else if (name.equals("way")) {  
  way = new LinkedList<Vertex>();  
} else ...
```

(on ignore la valeur de id)

```
<way id="22815079">  
  <nd ref="243494495"></nd>  
  <nd ref="245446883"></nd>  
  ...
```

tout sommet rencontré ensuite est mis dans cette liste

```
public void startElement(..., String name, Attributes a) {  
  ...  
} else if (name.equals("nd") && way != null) {  
  int ref = Integer.parseInt(a.getValue("ref"));  
  way.add(Vertex.fromId(ref));  
} else ...
```

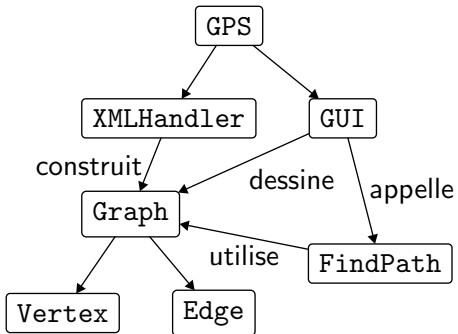


```
<way id="22815079">  
  ...  
</way>
```

quand on rencontre la balise fermante, on construit les arcs

```
public void endElement(..., String name) {  
    if (name.equals("way") && way != null && way.size() > 1) {  
        Vertex v = way.removeFirst();  
        for (Vertex w: way) {  
            Graph.addEdge(new Edge(v, w));  
            v = w;  
        }  
        way = null;  
    }  
}
```

interface graphique



on lance l'interface graphique avec

```
new GUI()
```

où

```
class GUI extends JFrame {
    GUI() {
        this.add(new Window()); // ajoute un composant JPanel
        this.setSize(800, 600);
        ...
    }
}
```

et

```
class Window extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        ...
    }
}
```

on accède au graphe par les méthodes statiques
Graph.vertices et Graph.edges

```
public void paintComponent(Graphics g) {  
    // dessiner chaque sommet  
    for (Vertex p : Graph.vertices())  
        drawVertex(g, p);  
    // puis chaque arc  
    for (List<Edge> l : Graph.edges())  
        for (Edge e : l)  
            drawEdge(g, e);  
}
```

```
// on dessine un sommet comme un disque
void drawVertex(Graphics2D g, Vertex p) {
    g.fillOval(x(p) - 2, y(p) - 2, 4, 4);
}
```

où les méthodes `x` et `y` convertissent les coordonnées sphériques en coordonnées cartésiennes (par exemple avec la projection Mercator)

idem pour `drawEdge`

cela convient ici, mais pas sur une carte plus grande

on pourrait organiser la carte selon un **quad tree** (cf TD 4)
et ne dessiner alors que les sous-arbres qui intersectent la fenêtre graphique

comment se programme l'**interaction** avec l'utilisateur ?

en fournissant des méthodes qui seront appelées
quand des **événements** se produisent :

- touche du clavier enfoncée, relâchée
- souris déplacée
- bouton de la souris enfoncé, relâché
- molette de la souris activée
- etc.

on donne un objet destiné à traiter les événements clavier

```
class Window extends JPanel {  
    Window() {  
        this.addKeyListener(new MyKeyListener());  
    }  
}
```

il **redéfinit** les méthodes appropriées

```
class MyKeyListener extends KeyListener {  
    @Override  
    public void keyTyped(KeyEvent e) {  
        // réagir ici à l'événement clavier e  
        ...  
    }  
}
```

idem pour la gestion des événements souris

```
class Window extends JPanel {
    Window() {
        ...
        this.addMouseListener(new MyMouseListener());
    }
}
```

```
class MyMouseListener extends MouseListener {
    @Override
    public void mousePressed(MouseEvent e) {
        // réagir ici à l'événement souris e
        ...
    }
}
```

ainsi on peut

- déplacer la carte en la faisant glisser
- zoomer avec la molette de la souris
- sélectionner deux points en cliquant sur la carte
- lancer un calcul de chemin en appuyant sur une touche

cf le code complet sur la page du cours

plus court chemin

on s'intéresse au problème du **plus court chemin** dans un graphe

non pas en terme de nombre d'arcs
(cf parcours en largeur la semaine dernière)

mais en fonction d'une **distance** d associée à chaque arc $u \rightarrow v$

$$u \xrightarrow{d} v$$

cette distance peut être

- la distance euclidienne
- la distance orthodromique
- le temps de trajet en voiture, à pieds
- le coût du transport
- etc.

la longueur du chemin

$$x_0 \xrightarrow{d_1} x_1 \xrightarrow{d_2} x_2 \cdots x_{n-1} \xrightarrow{d_n} x_n$$

est la somme des distances

$$\sum_{1 \leq i \leq n} d_i$$

on note $u \xrightarrow{d}^* v$ l'existence d'un chemin de u à v de longueur d

- on cherche les plus courts chemins à partir d'une *source* donnée
- le plus court chemin $source \rightarrow^* v$ n'est pas nécessairement unique
- la distance d de chaque arc $u \xrightarrow{d} v$ est supposée **positive ou nulle**

on se donne un graphe g

```
Graph<V> g;
```

et une distance entière associée à chaque arc

```
int weight(V u, V v) { ... }
```

la méthode `weight` ne sera appelée que pour u et v tels que $u \rightarrow v$

l'objectif est de remplir une table

```
HashMap<V, Integer> distance;
```

avec la longueur du plus court chemin depuis la source

- au départ, `distance` est vide
- à la fin, `distance` contient exactement les sommets atteignables

avec la méthode

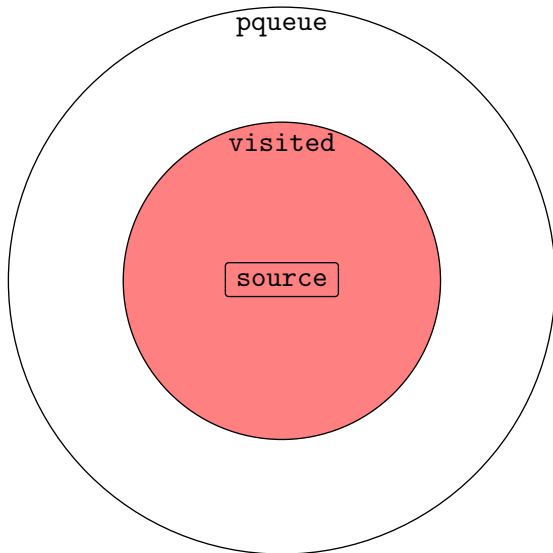
```
void shortestPaths(V source) { ... }
```

une solution à ce problème est l'**algorithme de Dijkstra**

le principe, comme pour le parcours en largeur, consiste à découvrir les plus courts chemins par ordre croissant de longueur

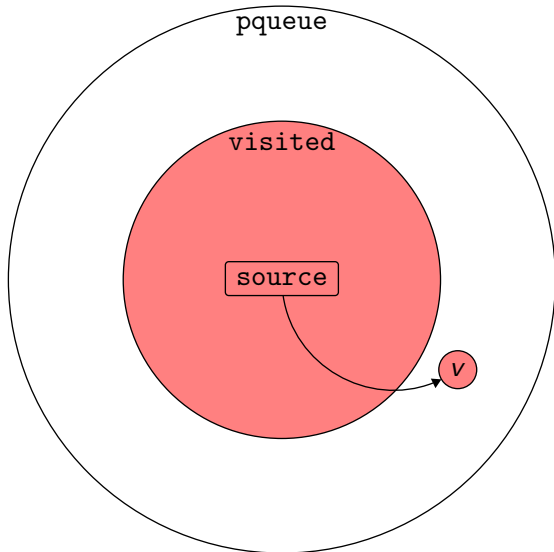
un ensemble (`visited`)
contient des sommets pour
lesquels on a déjà trouvé le
plus court chemin

un autre (`pqueue`) contient
des sommets déjà atteints



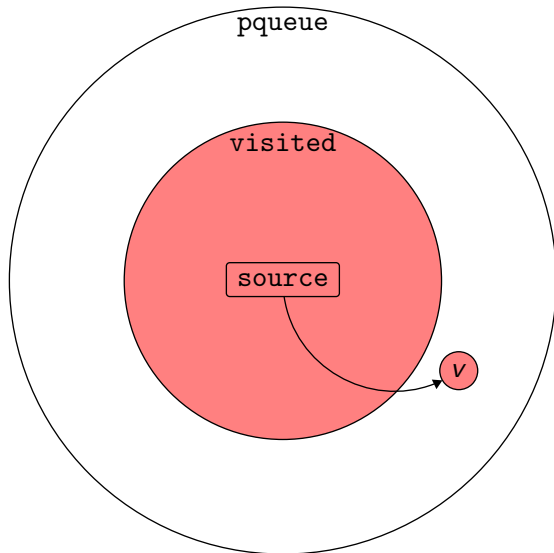
on sélectionne le sommet v
dans $pqueue$ à distance
minimale de la source

il est ajouté à $visited$



on considère tous les arc
 $v \rightarrow w$

il y a plusieurs cas de figure

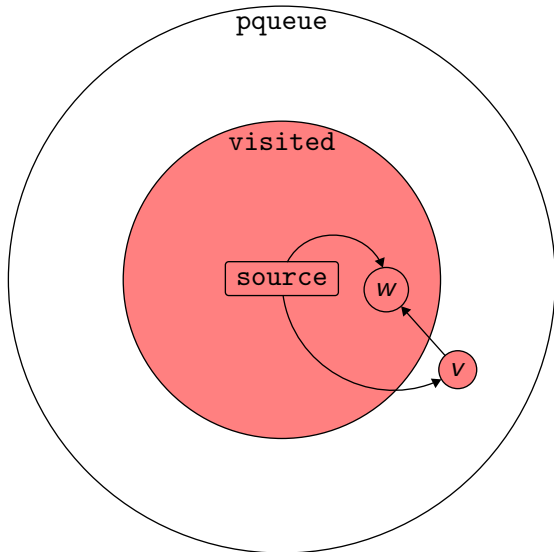


cas 1

w est déjà dans visited

le chemin trouvé n'est pas meilleur
(car $\text{weight}(v,w) \geq 0$)

il est ignoré



w est dans pqueue

cas 2

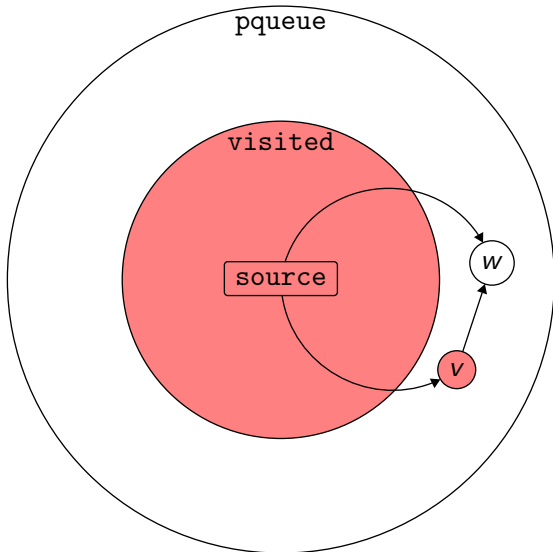
le chemin trouvé n'est pas amélioré

il est ignoré

cas 3

le chemin trouvé est plus court

la distance de w est mise à jour

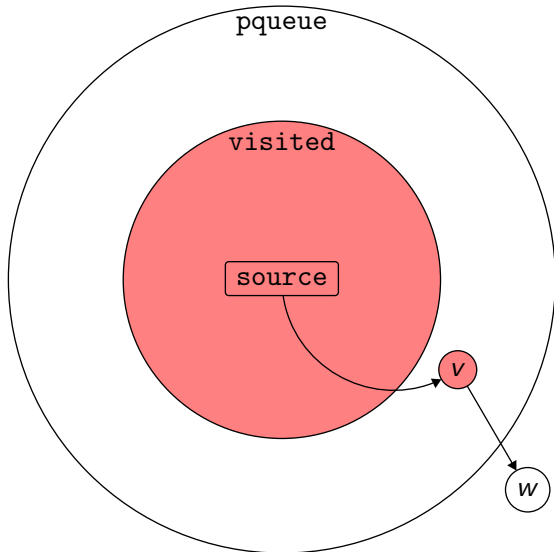


cas 4

w n'est pas dans pqueue

on vient de découvrir un
premier chemin menant à w

w est ajouté à pqueue



les sommets pour lesquels on a déjà trouvé le plus court chemin

```
HashSet<V> visited = new HashSet<V>();
```

une **file de priorité** (cf amphi 5) contient des sommets déjà atteints avec leur distance à la source

```
PriorityQueue<Node<V>> pqueue =  
    new PriorityQueue<Node<V>>();
```

la priorité est la distance

un sommet et sa distance à la source

```
class Node<V> implements Comparable<Node<V>> {  
  
    final V    node;  
    final int  dist;  
  
    Node(V node, int dist) {  
        this.node = node;  
        this.dist = dist;  
    }  
  
    @Override  
    public int compareTo(Node<V> that) {  
        return this.dist - that.dist;  
    }  
}
```

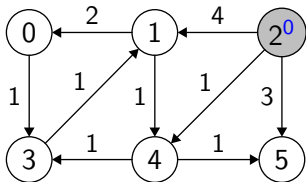
la source elle-même est à distance 0

```
distance.put(source, 0);
```

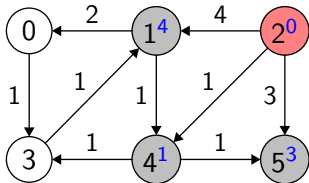
on la met dans la file de priorité avec cette distance

```
pqueue.add(new Node<V>(source, 0));
```

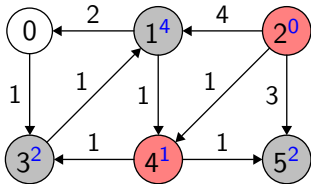
```
while (!pqueue.isEmpty()) {
    Node<V> n = pqueue.poll();
    // ne rien faire si on a déjà trouvé le plus court chemin
    if (visited.contains(n.node)) continue;
    // on vient de trouver le plus court chemin pour n.node
    visited.add(n.node);
    for (V w: g.successors(n.node)) {
        int d = n.dist + weight(n.node, w);
        // est-ce mieux d'emprunter n.node -> w?
        if (!distance.containsKey(w) || d < distance.get(w)) {
            distance.put(w, d);
            pqueue.add(new Node<V>(w, d));
        }
    }
}
```



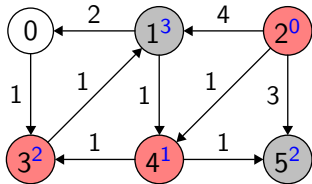
visited	pqueue
	2 ⁰



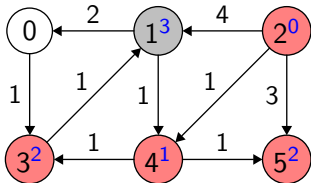
visited	pqueue
2	4 ¹
	1 ⁴
	5 ³



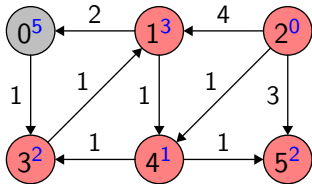
visited	pqueue
2	
4	
	1 ⁴
	5 ³
	3 ²
	5 ²



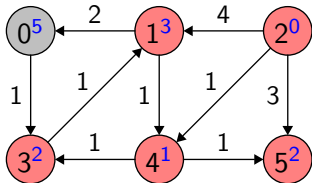
visited	pqueue
2	
4	
3	1 ⁴
	5 ³
	5 ²
	1 ³



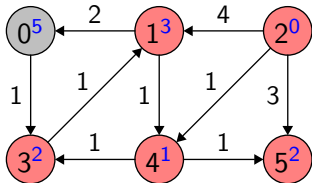
visited	pqueue
2	
4	
3	1 ⁴
5	5 ³
	1 ³



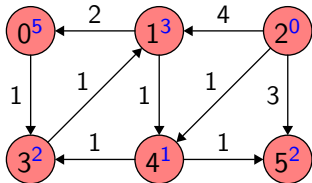
visited	pqueue
2	
4	
3	1 ⁴
5	5 ³
1	
	0 ⁵



visited	pqueue
2	
4	
3	1 ⁴
5	
1	
	0 ⁵



visited	pqueue
2	
4	
3	
5	
1	
	0 ⁵



visited	pqueue
	2
	4
	3
	5
	1
	0

si on souhaite **construire** le plus court chemin trouvé
on procède comme pour DFS et BFS (cf amphi 7) :

une table supplémentaire donne, pour chaque sommet v ,
l'arc $u \rightarrow v$ qui permet de l'atteindre dans le chemin optimal

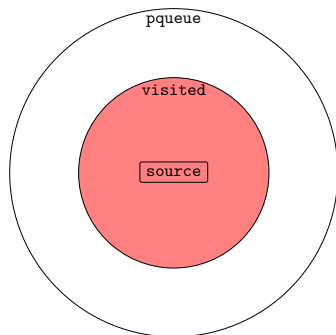
montrons la **correction** de cet algorithme

c'est-à-dire, qu'à la fin de la boucle **while**,

- `visited` contient exactement les sommets atteignables
- `distance` donne la longueur du plus court chemin

on le fait en établissement des **invariants de boucle**

c'est-à-dire des propriétés vraies à chaque tour de boucle



- (1) $\text{source} \in \text{visited} \cup \text{pqueue}$
- (2) $\text{distance}[\text{source}] = 0$
- (3) $\forall v \in \text{visited} \cup \text{pqueue},$
 $\text{source} \xrightarrow{\text{distance}[v]}^* v$
- (4) $\forall v \in \text{visited},$
 $\text{distance}[v]$ est la longueur d'un plus court chemin
- (5) $\forall v \in \text{visited}, \forall w$ t.q. $v \xrightarrow{d} w,$
 $w \in \text{visited} \cup \text{pqueue}$ et $\text{distance}[w] \leq \text{distance}[v] + d$
- (6) $\forall v,$ si $\text{source} \xrightarrow{d}^* v$ et $d < \min(\text{pqueue})$ alors $v \in \text{visited}$

on prouve que ces invariants

- sont vrais avant la boucle `while`
(lorsque `visited = ∅` et `pqueue = {source}`)
- sont préservés par toute itération de la boucle `while`

en particulier, ces invariants seront vrais à la sortie de la boucle,
lorsque `pqueue = ∅`

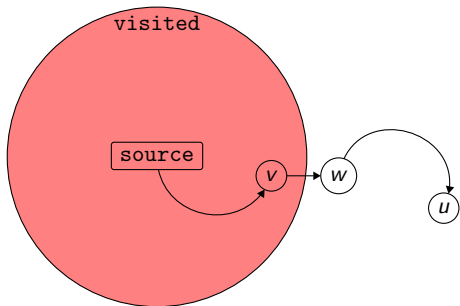
Invariant (4)

$\forall u \in \text{visited}$, $\text{distance}[u]$ est la longueur d'un plus court chemin

considérons l'instant où $\text{distance}[u]$ est fixée, c'est-à-dire quand u sort de la file

un chemin $\text{source} \rightarrow^* u$ strictement plus court sortirait de visited par $v \rightarrow w$

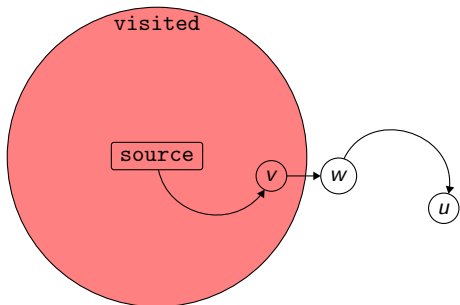
mais alors on aurait $\text{distance}[w] < \text{distance}[u]$ ce qui contredit le choix de u



Propriété

à la fin, `visited` contient exactement les sommets atteignables

- (3) garantit que les sommets de `visited` sont atteignables
- si `u` est atteignable et n'est pas dans `visited`, le premier arc sortant de `visited` $v \rightarrow w$ contredit (5)



- en **temps**

chaque arc est examiné au plus une fois

dans le pire des cas, chaque arc examiné entraîne une insertion dans la file $\Rightarrow E$ tours de boucle

chaque tour de boucle est dominé par le coût de la file de priorité, qui est au pire $O(\log E)$ c'est-à-dire $O(\log V)$ car $E \leq V^2$

d'où un total $O(E \log V)$

- en **espace**

clairement $O(E)$ dans le pire des cas

non, car il existe une structure de file de priorité
où la priorité d'un élément peut être modifiée en temps constant
(tas de Fibonacci)

on a alors

- V insertions/suppressions dans la file $\Rightarrow O(V \log V)$
- E modifications de priorité $\Rightarrow O(E)$

d'où un total $O(V \log V + E)$

mais **en pratique** la solution en $O(E \log V)$ est tout à fait satisfaisante

prenons deux points sur la carte, pour lesquels il existe un chemin

parcours en largeur

```
...  
103 arcs  
distance totale = 10.90 km
```

nombre d'arcs minimal

algorithme de Dijkstra

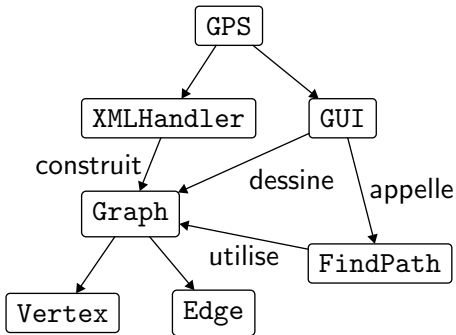
```
...  
132 arcs  
distance totale = 8.76 km
```

distance minimale

si la distance associée à chaque arc est la même,
alors l'algorithme de Dijkstra est inutile :
un simple parcours en largeur suffit

```

30 GPS.java
64 XMLHandler.java
50 Vertex.java
26 Edge.java
58 Graph.java
103 FindPath.java
17 VertexDist.java
248 GUI.java
-----
596 total
    
```



version **bidirectionnelle**
de l'algorithme de Dijkstra



dans les rues de New-York



- lire le **poly**, chapitre 16.2 plus court chemin

il y a des **exercices** dans le poly

- CC le **lundi 23 octobre** 14h–17h
 - documents autorisés : poly, transparents, notes personnelles
 - dictionnaire papier autorisé pour les élèves étrangers
 - sujets et corrigés des années précédentes sur la page du cours

que retenir de 411 ?

- l'ordre n'importe pas

	insertion	suppression	
table de hachage	$O(1)$	$O(1)$	(amorti)

- l'ordre importe

	insertion	suppression
arbre binaire de recherche	$O(\log N)$	$O(\log N)$
file de priorité	$O(\log N)$	$O(\log N)$

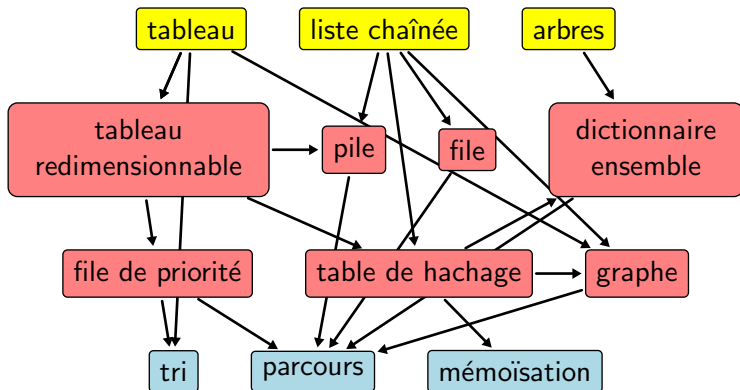
- **mémoïser**, c'est ne pas calculer deux fois la même chose

- **trier** N valeurs par comparaisons
ne peut pas être fait en moins que $O(N \log N)$

le tri pas tas (*heapsort*) est optimal, en temps et en espace

- on peut **parcourir un graphe** de V sommets et E arcs,
à partir d'un sommet donné,
en temps $O(V + E)$ et en espace $O(V)$, ce qui est optimal

les briques de base sont peu nombreuses



8.3 tas auto-équilibrés (*skew heaps*)

9 classes disjointes (*union-find*)

12 rebroussement (*backtracking*)

beaucoup de données librement accessibles,
utiles pour vos projets

- projet Gutenberg
53000 livres sur <https://www.gutenberg.org/>
- Wikipedia
<https://dumps.wikimedia.org/>
- OpenStreetMap
<https://openstreetmap.fr/>

si vous voulez tout savoir sur la manière dont les langages de programmation et les compilateurs sont faits, le cours

INF564 Compilation

est fait pour vous

en particulier, vous écrirez un compilateur optimisant pour un fragment du langage C vers l'assembleur x86-64

bonne continuation à tous,
avec ou sans informatique