Travail d'Etude et de Recherche
GM6

Recherche –intelligente !- du chemin le plus intelligent

Brameret Pierre-Antoine - Ibanez Aurélien - Kim Raphaël

CACHAN

Encadrant : Genet Martin

Année universitaire 2009-2010

Table des Matières

I. Contexte et Théorie

I.1. Première approche

i) Problème réel

Considérons une personne en vélo ; il veut parcourir le chemin le plus court et le moins fatiguant possible (pentes, etc.) d'un point A à un point B.
C'est ce chemin que nous appelons *chemin intelligent*.

S'il n'est en possession que d'un simple plan de ville, par exemple, aucun indice sur le dénivelé des différents itinéraires envisageables ne lui est donné.
En revanche, s'il est en possession d'une carte type IGN[1], il lui est possible –mais fastidieux !- d'appréhender le relief grâce à la lecture des courbes de niveaux.

Nous avons alors pour but de mettre à disposition un outil simple et rapide capable de calculer ce chemin intelligent selon les préférences (pentes raides mais courtes/pentes douces mais soutenues) de l'utilisateur.

ii) Abstraction : Représentation mathématique et résolution

La théorie des graphes est un outil permettant, par la forme de la représentation des données à traiter (cartes), la modélisation et la résolution de notre problème. L'histoire de la théorie des graphes débute avec les travaux d'Euler au XVIIIe siècle avec certains problèmes liés à la recherche de chemins dans une ville. Cette théorie semble donc adéquate et est loin d'être archaïque, s'étant considérablement étendue avec les progrès de l'informatique.

iii) Implémentation

La quantité de données à traiter suggère évidemment la mise en place d'un outil informatique.

- **Code**

La théorie des graphes offrant de nombreux algorithmes de résolution d'un problème de plus court chemin, cet outil sera écrit dans un langage jugé convenable.

- **Données**

L'outil traitera des données issues de sources libres adaptées par nos soins à notre problème et compilées.

[1] Institut Géographique National

I.2. Définition du problème

i) Tenants et aboutissants

Nous avons à notre disposition :
- données cartographiques, topologiques et routières formatées,
- algorithmes issus de la théorie des graphes.

Nous souhaitons permettre à une personne de trouver le trajet idéal pour aller à sa destination. Ce trajet idéal tient compte de la distance à parcourir et des capacités de la personne. Nous devrons donc adapter nos données tri-dimensionnelles à cette recherche.

ii) Analyse de l'existant

L'outil présenté dans ce rapport répond à un besoin réel.

En effet, comme l'écrit le journal *Le Monde*[2], certains endroits imposent un *«relief parfois douloureux pour les mollets des cyclistes ».*
Cependant les solutions actuelles ne sont que d'une faible utilité.

- **Outils existants**

Dans la même source citée précédemment, on peut lire :

> *« Certains sites spécialisés dans la recherche d'itinéraires […]* ***proposent certes une option "vélo"****. Mais le trajet affiché, s'il prend en compte les sens interdits et précise l'emplacement des radars automatiques pourtant* ***inutiles aux cyclistes****,* ***ignore totalement la rudesse des dénivelés****. »*

On voit donc que les solutions informatiques actuelles sont, bien que puissantes, rapides et très efficacement mises à jour, ne traitent qu'un aplanissement de l'abrupte réalité terrestre.

- **Cartographies existantes**

Les cartes topographiques actuelles sont précises mais réservées à un public averti. De plus, leur utilisation pour la recherche d'un chemin *intelligent* est quasiment impossible sur une *longue* distance.

Toujours dans l'article de *Le Monde*, on découvre la mise en place d'une carte[3] vulgarisée présentant sous forme de couleur le caractère raide ou plat de chaque rue de Paris que le photographe et le cartographe ont parcouru.
Le principal inconvénient de cette carte est le manque de personnalisation. En effet l'utilisateur ne peut définir lui-même à partir de quelle inclinaison une pente est considérée comme trop abrupte, par exemple (bien entendu, une solution informatique représente également un gain considérable en rapidité de traitement).

[2] *LeMonde.fr*, article du 04 Janvier 2010 : « Le plan qui épargne les mollets des cyclistes parisiens »
[3] Velo Pente, *Velopente.com*

iii) Définition du problème

- **Représentation du problème**

Le problème de plus court chemin que nous avons à résoudre consiste à traiter un ensemble données issus d'un support de type carte routières agrémentées de données topologiques. Les éléments d'intérêt sont, de manière évidente, les carrefours et les routes. Le parcours de publications diverses traitant de ce type de problème a rapidement orienté nos recherches sur une modélisation intuitivement adaptée de représentation de ces données : la théorie des graphes.

Il est ici nécessaire, avant d'aller plus loin, de définir les concepts de cette théorie et la terminologie qui y est associée, en nous restreignant aux issues pertinentes selon notre problème.

- **Eléments de Théorie des Graphes**

Voir annexe V.1

- **Paramètres**

Pour traiter des problèmes d'altitude, nous introduire également une pondération des distances. Nous aurons une matrice de distance qui peut varier avec chaque utilisateur. Puisqu'il est difficile d'estimer comment va varier le ressenti d'une personne pour une pente donnée et avec un moyen de locomotion donné, nous avons écrit une relation linéaire pour prendre en compte l'altitude.

$$m'_{ij} = m_{ij} + \alpha M * pM * lM + \alpha D * pD * lD$$

α : $coefficient\ de\ ressenti\ de\ la\ pente\ pour\ la\ personne$
αM : $coeff\ \alpha\ pondéré\ par\ 1{,}2$
αD : $coeff\ \alpha\ pondéré\ par\ 0{,}8$
lM : $la\ longueur\ de\ route\ en\ montée$
lD : $la\ longueur\ de\ route\ en\ descente$
pM : $la\ pente\ moyenne\ en\ phase\ de\ montée\ (sans\ unité)$
pD : $la\ pente\ moyenne\ en\ phase\ de\ descente$

Le coefficient alpha est de l'ordre de grandeur de 1. Plus il est élevé, plus on souhaite éviter les dénivelés importants.

I.2. Résolution

La théorie des graphes nous amène différents algorithmes permettant de résoudre un problème de plus court chemin. Si les applications sont extrêmement vastes, il sera nécessaire d'évaluer ceux-ci selon des critères adaptés à notre problème.

*i) Algorithme A**

- **Principe**

L'algorithme de A*, contrairement aux autres algorithmes, nécessite de préciser le nœud de départ et le nœud d'arrivé. C'est l'algorithme qui se rapproche le plus de l'intuition humaine. En effet, on emprunte le chemin permettant de se rapprocher de plus en plus du point d'arrivé.

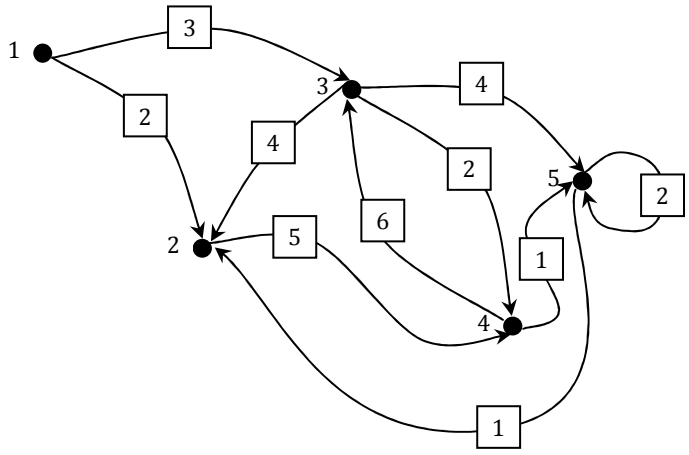
Soit x_0 le nœud de départ et x_f le nœud d'arrivé.
Soit x_{ij} si on a la possibilité d'aller de i vers j
Alors l'algorithme s'écrira :

Initialisation
Minimum = infini

Algorithme
Pour *i jusqu'au nombre de nœuds* ***Faire***
    Pour *j jusqu'au nombre de nœuds* ***Faire***
        Si *existe* x_{ij} ***Faire***
            Si minimum < $dist(x_i, x_j)$
            minimum = $dist(x_i, x_j)$
        Lorsque *la variable "minimum" est minimum* ***Relever*** *j*
        Liste[i]= x_i

La distance "*dist*" est ici définie de la manière suivante :
$dist(x_i, x_j)$ = *distance entre* x_i *et* x_j + *distance entre* x_j *et* x_f

- **Exemple sur un graphe orienté pondéré à partir du nœud 1 au nœud 5**



Matrice d'adjacence pondérée :

$$M = \begin{pmatrix} \infty & 2 & 3 & 0 & 0 \\ \infty & \infty & \infty & 5 & \infty \\ \infty & 4 & \infty & 3 & 4 \\ \infty & \infty & 6 & \infty & \infty \\ \infty & 1 & \infty & \infty & 2 \end{pmatrix}$$

Matrice des distances d'un nœud à un autre :

$$M' = \begin{pmatrix} 0 & 2 & 3 & 6 & 7 \\ 2 & 0 & 4 & 5 & 4 \\ 3 & 4 & 0 & 3 & 2 \\ 6 & 5 & 3 & 0 & 2 \\ 7 & 4 & 2 & 2 & 0 \end{pmatrix}$$

(les valeurs sont justes à titre d'exemple. Ce sont des distances en vol d'oiseau)

<u>itération i=1</u>
<u>nœud 1</u>
les chemins possibles sont : 2 et 3

aller au nœud 2?
dist(1,2)=de 1 à 2 puis de 2 à 5 = 2+4=6

aller au nœud 3?
dist(1,3)=de 1 à 3 puis de 3 à 5 = 3+2=5

minimum = 5

on va donc au nœud 3

<u>itération i=2</u>
<u>nœud 3</u>
les chemins possibles sont : 2 et 4 et 5

aller au nœud 2?
dist(3,2)=de 3 à 2 puis de 3 à 5 = 4+2=6

aller au nœud 4?
dist(3,4)=de 3 à 2 puis de 2 à 5 = 4+4=8

aller au nœud 5?
dist(3,5)=de 3 à 5 puis de 5 à 5 = 4+0=4

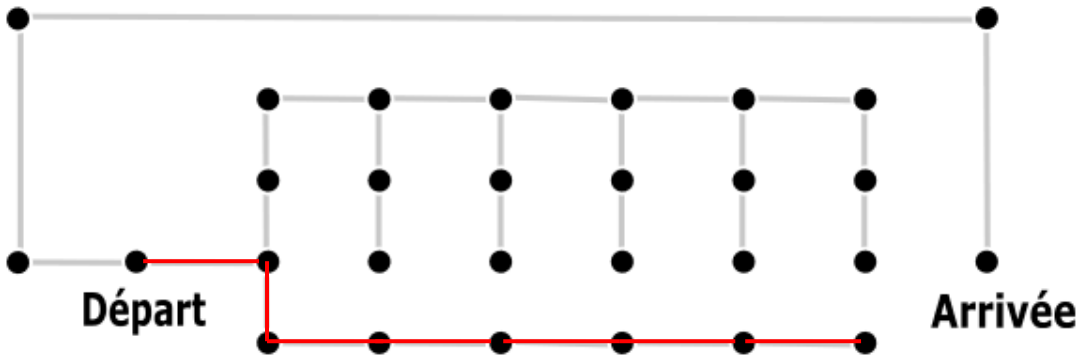
minimum = 4

on va donc au nœud 5
Arrivé!

La liste est donc : 1 - 3 - 5

- **Limite de l'algorithme A***

Le bon fonctionnement de cet algorithme dépend fortement du graphe.
Voici un exemple classique de graphe dans lequel cet algorithme échoue :



En rouge on voit le chemin qu'emprunterai l'algorithme A*.

Cependant nous pouvons noter que très souvent les graphes de type carte routière sont conçus de façon à obtenir un bon fonctionnement de cet algorithme.

- **Amélioration de l'algorithme A***

Voir annexe V.2.

ii) Algorithme de Floyd-Warshall

- **Principe**

L'algorithme de Floyd-Warshall établit tous les plus courts chemins entre tous les nœuds. C'est le mode d'établissement des chemins qui impose cette structure des données. On part d'une matrice des distances que l'on va compléter pour obtenir la matrice des distances pondérées. En parallèle de cet algorithme, on stockera les chemins parcourus.

Notations :

$$M \text{ la matrice des distances}$$
$$P \text{ la matrice des chemins}$$
$$m_{ij} \text{ les éléments de } M$$
$$p_{ij} \text{ les éléments de } P$$

La matrice des chemins P représente en ij l'antécédent de i sur la route menant à j. Son fonctionnement et un exemple sont présents dans la partie suivante, au sujet de l'algorithme de Dijkstra, avec le vecteur *Pred.*

La procédure est de chercher, pour chaque point k, si il est préférable, pour aller de i à j, d'aller directement de i à j, ou bien de passer par k. Dans ce cas :

$$m_{ij} = m_{ik} + m_{kj}$$
$$p_{ij} = k$$

i et j vont parcourir tous les points pour chaque k. On ne saura qu'à la fin, c'est-à-dire quand k aura parcouru tous les chemins, quel est le chemin préférable pour aller d'un point à un autre.

Initialisation
construire la matrice des distances M
construire la matrice des chemins P avec

$$p_{ij} = i$$

Algorithme
Pour k de 1 à n ***Faire***
  Pour i de 1 à n Faire
    Pour j de 1 à n ***Faire***

      Si m(i,j) > m(i,k) + m(k,j)
        on passe par le point k

- **Performances**

La complexité de cet algorithme est en $O(n^3)$, où n est le nombre de points. En effet, pour k variant de 1 à n, i varie de 1 à n, puis pour (k,i) donné, j varie de 1 à n.

L'encombrement en mémoire est principalement de $2*n^2$ cases mémoires pour les matrices des distances pondérées et matrice des chemins.

Après avoir codé l'algorithme de la sorte, nous sommes passés par une phase d'optimisation de l'implémentation qui a permis de diviser par 2 l e temps d'exécution. La réduction du temps d'exécution n'est pas prévisible par le calcul, c'est-à-dire que ces

optimisations n'ont pas changé l'ordre de complexité générale d'exécution de l'algorithme. L'optimisation nous a fait changer la structure des données pour les matrices. Nous avons opté pour des matrices « sparse », c'est-à-dire creuse. L'encombrement en mémoire est beaucoup plus faible dans la majorité des cas mais reste cependant non prévisible.

On peut observer en II.2.iii) un relevé d'une courbe réelle du temps d'execution en fonction du nombre de points.

iii) Algorithme de Dijkstra

- **Principe**

L'algorithme de Dijkstra permet d'obtenir un classement des nœuds d'un graphe selon leur distance à un nœud source choisi à l'avance. On obtient finalement l'ensemble des plus courts chemins à tous les nœuds accessibles d'un graphe depuis ce nœud source.

Notons

$$x_j \text{ le } j^{ème} \text{ noeud le plus proche du noeud source } x_0$$
$$V(x) \text{ l'ensemble des voisins du noeud } x$$

L'algorithme se base sur l'observation suivante :

$$\exists j < k \text{ tel que } x_k \in V(x_j)$$

Autrement dit, le $k^{ème}$ nœud le plus proche de x_0 est le voisin d'un x_j, $j^{ème}$ nœud le plus proche de x_0, pour un $j < k$.

Preuve :

Soit $SP_n = \{x_0, x_{SP1}, \dots, x_{SPn-2}, x_{k+1}\}$ *les plus court chemin de* x_0 *à* x_{k+1}.

Or nécessairement x_{SPn-2} *est plus proche de* x_0 *que* x_{k+1} *l'est (car les arcs sont de poids positif) :*

$$dist(x_0, x_{SPn-2}) < dist(x_0, x_{k+1})$$

Donc x_{SPn-2} *fait partie des* $k^{ème}$ *nœuds les plus proches de* x_0*, autrement dit :*

$$x_{spn-2} \in \{x_j\}_{j<k+1}$$

Finalement,

$$\forall j < k+1, x_{k+1} \in V(x_j).$$

Ainsi, pour trouver le $k^{ème}$ nœud le plus proche de la source, il suffit de connaître tous les $j^{ème}$ nœuds les plus proches quel que soit $j < k$.

En effet, x_k est alors tel que :

$$Distance(x_0, x_j) + Longueur(x_j, x_k) = \min_{i<k, x \in V(x_i)} [Distance(x_0, x_i) + Longueur(x_j, x)]$$

On en déduit la structure de l'algorithme :

Initialisation
Atteints = $\{x_0\}$
Pred(i)= 1 si *i* est voisin de x_0
= 0 sinon

Algorithme
Tant que non *Atteints* n'est pas vide ***Faire***
Trouver *v* le nœud le plus proche de x_0 dans non *Atteints*
Ajouter *v* à *Atteints*

Pour chaque voisin *w* de *v* ***Faire***
Si *w* n'appartient pas à *Atteints*
Si *dist(x_0,w) > dist(x_0,v) + dist(v,w)*
Définir *dist(x_0,w)* égal à *dist(x_0,v) + dist(v,w)*
Définir *Pred(w) = v*

- **Interprétation des résultats**

La reconstruction du plus court chemin de x_0 à un nœud se fait enfin grâce au vecteur *Pred.* Ce vecteur contient en effet le prédécesseur de chaque nœud sur l'arbre des plus courts chemins issu de x_0.

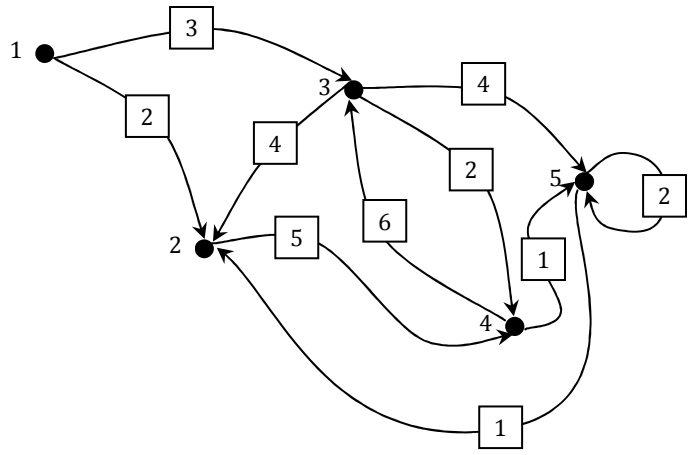
Ainsi, le plus court chemin de x_0 à y se retrouve en exécutant :

Initialisation
Chemin = $\{y\}$

Algorithme
Tant que *Point* ≠ x_0 ***Faire***
Définir *Point* égal à *Chemin(0)*
Insérer *Pred(Point)* en première position dans *Chemin*

Et la longueur de ce plus court chemin se trouve en rappelant la valeur de *dist(x_0, y).*

- **Exemple sur un graphe orienté pondéré: plus court chemin à partir du nœud 1**



Initialisation

$Atteints = \begin{pmatrix} 1 \\ / \\ / \\ / \\ / \end{pmatrix}$; $Pred = \begin{pmatrix} \# \\ 1 \\ 1 \\ / \\ / \end{pmatrix}$

dist(1,1) = 0
dist(1,2) = 2
dist(1,3) = 3
dist(1,4) = ∞
dist(1,5) = ∞

Première itération
v = 2
dist(1,4)⊡dist(1,2)+dist(2,4)
Pred(4)⊡2

$Atteints = \begin{pmatrix} 1 \\ 2 \\ / \\ / \\ / \end{pmatrix}$; $Pred = \begin{pmatrix} \# \\ 1 \\ 1 \\ 2 \\ / \end{pmatrix}$

dist(1,1) = 0
dist(1,2) = 2
dist(1,3) = 3
dist(1,4) = 7
dist(1,5) = ∞

Deuxième itération
v = 3
dist(1,4)⊡dist(1,3)+dist(3,4)
Pred (4)⊡3
dist(1,5)⊡dist(1,3)+dist(3,5)
Pred(5)⊡3

$Atteints = \begin{pmatrix} 1 \\ 2 \\ 3 \\ / \\ / \end{pmatrix}$; $Pred = \begin{pmatrix} \# \\ 1 \\ 1 \\ 3 \\ 3 \end{pmatrix}$

dist(1,1) = 0
dist(1,2) = 2
dist(1,3) = 3
dist(1,4) = 5
dist(1,5) = 7

Troisième itération
v = 4
dist(1,5)⊡dist(1,4)+dist(4,5)
Pred(5)⊡4

$Atteints = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ / \end{pmatrix}$; $Pred = \begin{pmatrix} \# \\ 1 \\ 1 \\ 3 \\ 4 \end{pmatrix}$

dist(1,1) = 0
dist(1,2) = 2
dist(1,3) = 3
dist(1,4) = 5
dist(1,5) = 6

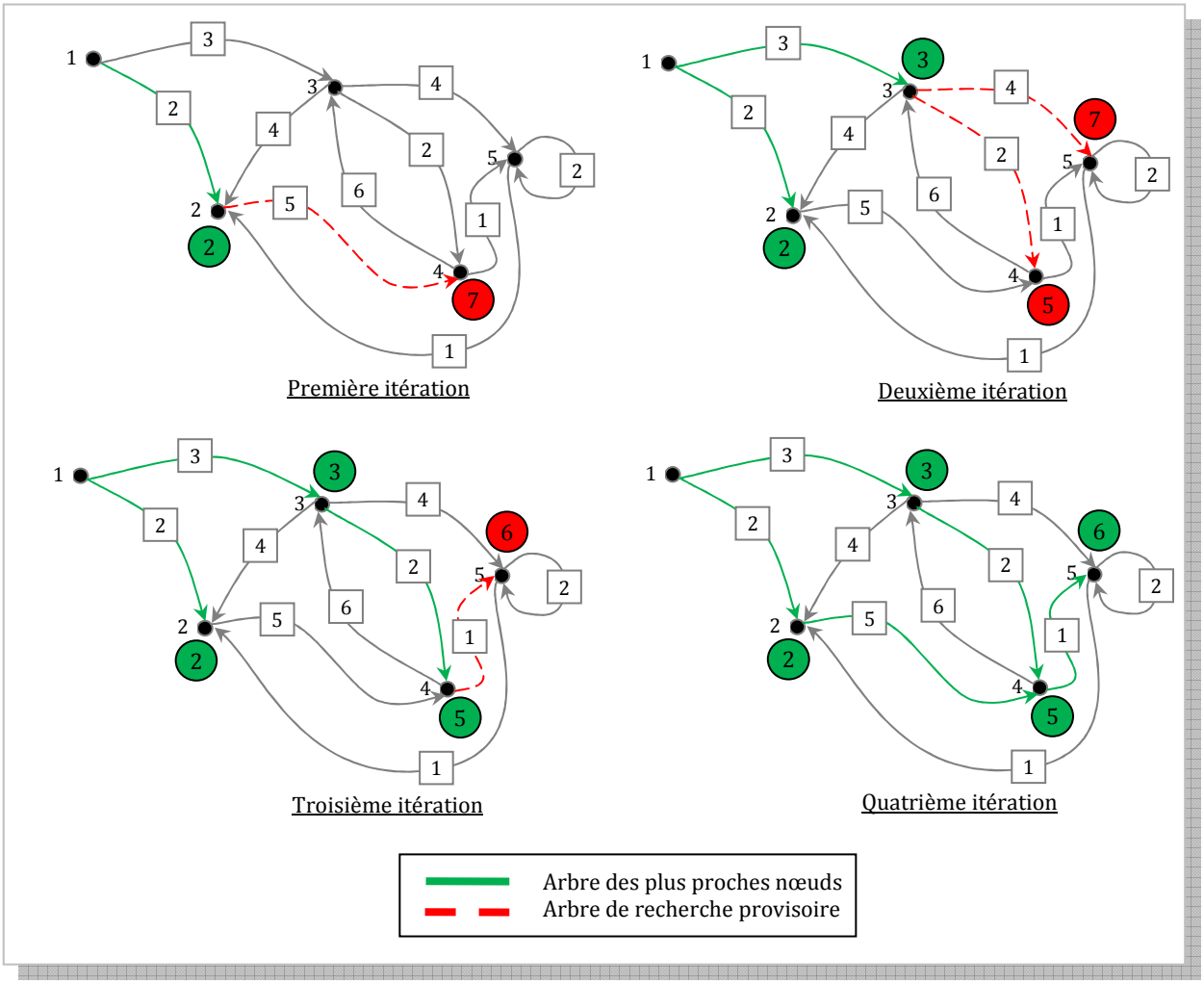
Quatrième itération
v = 5

$Atteints = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$ FIN.

- **Remarque**

On peut remarquer que cet algorithme contient deux arbres implicites : un arbre des plus proches nœuds, représenté par le vecteur *Atteints*, et un arbre de recherche provisoire, représenté par l'ensemble des x tels que $dist(x_0, x) \neq \infty$. Ce dernier marque les plus proches nœuds estimés. Notons que l'arbre de recherche provisoire contient l'arbre des plus proches nœuds.

L'évolution de ces deux arbres sur l'exemple ci-dessus peut être visualisée ici :



Cette remarque étant faite, on peut donc arrêter l'exécution de l'algorithme lorsque le point d'intérêt (arrivée) est ajouté à l'arbre des plus proches nœuds, soit le vecteur *Atteints.*
Ainsi, l'algorithme de recherche de plus court chemin entre les nœuds x_0 et y s'écrit, en utilisant la méthode de Dijkstra et la condition d'arrêt précédente:

Initialisation
Atteints = $\{x_0\}$
Pred(i)= 1 si *i* est voisin de x_0
= 0 sinon

Algorithme
Tant que $v \neq y$ ***Faire***
Trouver *v* le nœud le plus proche de x_0 dans non *Atteints*
Ajouter *v* à *Atteints*

Pour chaque voisin *w* de *v* ***Faire***
Si *w* n'appartient pas à *Atteints*
Si $dist(x_0,w) > dist(x_0,v) + dist(v,w)$
Définir $dist(x_0,w)$ égal à $dist(x_0,v) + dist(v,w)$
Définir *Pred(w) = v*

- **Validité de l'algorithme de Dijkstra**

Preuve

Supposons que la $k^{ème}$ itération de l'algorithme identifie correctement le $k^{ème}$ nœud x_k le plus proche de x_0.

On sait que x_{k+1} et voisin d'un x_j, $j < k+1$.
A l'itération $j < k+1$, l'algorithme définit dist(x_0, x_{k+1})=dist(x_0, x_j)+dist(x_j, x_{k+1}) qui est la vraie plus courte distance de x_0 à x_{k+1}.
A l'itération $(k+1)$, l'algorithme reconnait x_{k+1} comme étant le prochain nœud le plus proche de x_0 (les $x_i, i < k+1$ ont déjà été placés dans Atteints*, par hypothèse) en parcourant les dist(x_0, x) et a déjà calculé sa distance à x_0 à l'itération j.*
Donc

$$\left\{\begin{array}{c} la\ k^{ème}\ itération\ de\ l'algorithme \\ identifie\ correctement \\ le\ k^{ème}\text{nœud}\ x_k\ le\ plus\ proche\ de\ x_0 \end{array}\right\} \Rightarrow \left\{\begin{array}{c} la\ (k+1)^{ème}\ itération\ de\ l'algorithme \\ identifie\ correctement \\ le\ (k+1)^{ème}\text{nœud}\ x_{k+1}\ le\ plus\ proche\ de\ x_0 \end{array}\right\}$$

Vérifions le résultat vrai à l'itération 1 :
non Atteints *représente l'ensemble des nœuds du graphe privé du nœud de départ x_0.*
*L'opération « **Trouver** v le nœud le plus proche de x_0 dans non* Atteints *» trouve le plus proche voisin de x_0, qui est effectivement x_1, le premier nœud le plus proche de x_0.*

iv) Choix méthode (algorithme) – Un ou plusieurs - Justification

- **Démonstrations – Qu'il soit au moins admissible !**

La convergence de l'algorithme A* n'étant pas acquise quel que soit le chemin demandé, l'intégrer dans un programme de recherche de plus court chemin semble inapproprié. De plus, si elle converge, la solution donnée n'est qu'une approximation, elle n'est pas forcément admissible.

En revanche, les algorithmes de Floyd-Warshall et de Dijkstra convergent quelles que soient les conditions initiales et sont, bien que beaucoup moins intuitifs qu'A*, **admissibles.**

Nous choisirons donc de ne retenir que ces deux derniers admissibles, et robustes au sens de la convergence.

- **Avantages, possibilités – Définir des critères de choix et d'évaluation de performances, robustesse – Intuitif ?**

Il reste alors à définir des critères d'évaluation des performances des algorithmes afin de caractériser l'adéquation algorithme/données initiales.

Le critère prépondérant est bien entendu la rapidité d'exécution. Ensuite, en imaginant l'intégration de notre méthode de recherche dans des appareils mobiles, nous voyons émaner un second critère : la mémoire nécessaire à l'exécution des différents algorithmes.

	Floyd-Warshall	**Dijkstra**
Complexité	$O(n^3)$	$O(n^2)$
Mémoire demandée	$O(n^2)$	$O(n^2)$

v) Problèmes pratiques – transition

- **Temps de résolution/ Mémoire nécessaire**

Malgré la considérable évolution en passant en système de matrices creuses, le calcul avec l'algorithme de Floyd sur l'ile de France n'a pas pu aboutir, faute de place en mémoire sur les machines de calcul du LMT, après plus de **30h** de travail. Nous avons rempli 3,5 Go de mémoire pour aboutir à ce non résultat.

- **Réaction à la modification des données**

Il faut 15 heures pour formater une zone large comme l'ile de France. Le fichier de données initial pèse 270Mo, le fichier traité pèse 7Mo. Des mises à jour régulières semblent impossibles.

Pour une zone plus réduite, de 4.3 Mo pour le fichier initial, soit 3006 points, le temps de calcul pour l'algorithme de Floyd est de 25 minutes, et l'affichage des résultats pèse 1.4Go, ce qui semble impossible à stocker également. La zone couverte est de l'ordre du km^2. Pour information, il y a 142000 points définis dans l'Ile de France !

- **Tout ceci dépend également de la structure du code et des données !**

II. Implémentation

II.1. Justification

i) Langage

Nous avons eu besoin d'un language polyvalent, performant, libre et suffisamment de haut niveau pour que l'on puisse faire toutes nos tâches librement, que nous puissions communiquer entre nous et avec l'encadrant, et que nous ne perdions pas de performances dans des applications externes. En plus de cela, nous avons utilisé des morceaux de codes provenant d'autres auteurs pour le formatage des données. Nous nous sommes donc tourné vers le C++, qui regroupe tous les avantages cités.

ii) données

- **Structure**

Les données routières les plus précises et les plus exploitables ont été tirées de la base de données du projet OpenStreetMap[4] (*OSM*), distribuée sous licence libre.
Le format d'extraction est le format XML[5] : les différents *objets* entrés par les participants au projet sont représentés sous forme d'arborescence.
Parmi beaucoup d'autres, on peut citer les objets suivants :

- *node* : « nœud », point de mesure repéré par un couple (latitude, longitude) et une identité;
- *way* : « voie », route (ou chemin) décrit par les identités des nœuds qui la constituent et par son type (autoroute, route secondaire, etc.) ;

Exemple : aperçu du contenu d'une cartographie OSM au format XML

```
….
 <node id="475057044" lat="48.7735598" lon="2.3130706" version="2"
                changeset="2540635" user="wallclimber21" uid="81024" visible="true"
                timestamp="2009-09-20T03:44:16Z"/>
<way id="11288755" visible="true" timestamp="2009-09-02T11:48:20Z" version="9"
                changeset="2347390" user="pafnow" uid="157511">
        <nd ref="257802188"/>
        …
         <nd ref="100296871"/>
        <tag k="name" v="Rue Marcel Bonnet"/>
        <tag k="highway" v="tertiary"/>
        <tag k="ref" v="D68"/>
</way>
…
```

- **Cohérence données/utilisation**

La structure de la cartographie n'est ici pas adaptée à un traitement en direct. En effet, la lecture d'une telle arborescence est longue (chaque accès à une balise étant en $O(n)$, où n est le nombre de balises du même type) et beaucoup d'informations sont inutiles à notre problème. Nous avons donc conçu un outil de simplification permettant de n'extraire que les nœuds carrefours ainsi que les routes, disposés *ligne par ligne* dans un fichier texte ASCII[6] (ce dernier

[4] Projet collaboratif de cartographie mondiale sous licence libre, *www.openstreetmap.fr*
[5] Extensible Markup Language, langage informatique de balisage « personnalisable ».

choix n'est pas optimal mais est facilement mis en œuvre). Nous stockons d'abord l'ensemble des carrefours avec leurs coordonnées (X,Y,Z), puis les routes. Pour celles-ci, nous stockons leur longueur réelle, le dénivelé cumulé en montée, la longueur de route en montée (pour obtenir la pente moyenne en montée), puis le dénivelé en descente et la longueur en descente.

Le contenu est très fourni. Nous disposons ainsi de nombreuses informations qui peuvent être utiles pour la modélisation d'un outil plus développé (feux rouges, points d'intérêt, etc.).
Malheureusement, une information cruciale nous manque : l'altitude des nœuds.
Les données topographiques libres disponibles d'après nos recherches ne sont que trop peu précises (résolution minimale de la grille de l'ordre du kilomètre[7]). Nous avons donc simulé un relief sur les cartes que nous traitons. Pour cela, nous avons généré un polynôme en X et Y de degré suffisamment élevé pour obtenir des courbes intéressantes (typiquement 8), en faisant passer cette courbure par le nombre nécessaires de points pour la définir.

Pour passer des données de latitude et longitude aux données (X,Y) qui correspondent à un repérage « à plat », nous avons choisi un modèle de repérage tel que :

- Le point (0,0) a pour coordonnées GPS (lat=0°, lon=0°)
- L'axe Y est l'axe Nord-Sud local
- La coordonnée Y est donc la distance à l'équateur, en terme de longueur d'arc : R*latitude où latitude est exprimée en radians et R est le rayon moyen de la Terre (6 360 km).

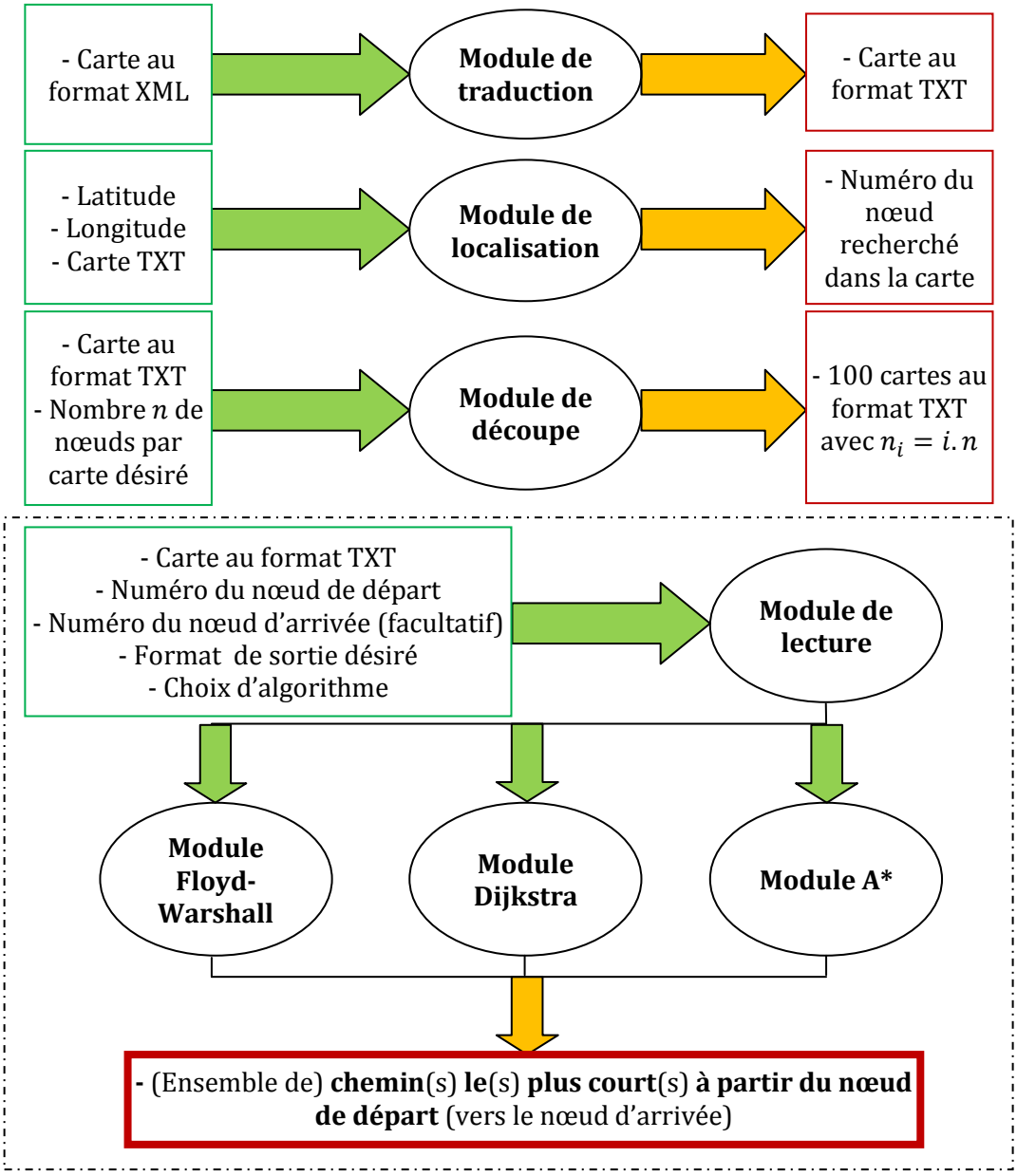
La coordonnée X est alors la distance au méridien en terme de longueur d'arc : R*cos(latitude)*longitude.

[6] American Standard Code for Information Interchange, norme informatique de codage de caractères.
[7] Modèle ETOPO1 du National Geophysical Data Center, résolution 1 arc-minute.

iii) code

- **Structure**

Nous avons divisé notre outil en plusieurs modules, présentés ci-dessous.



- **Transcription des algorithmes formels**

Voir annexe **Erreur ! Source du renvoi introuvable.**

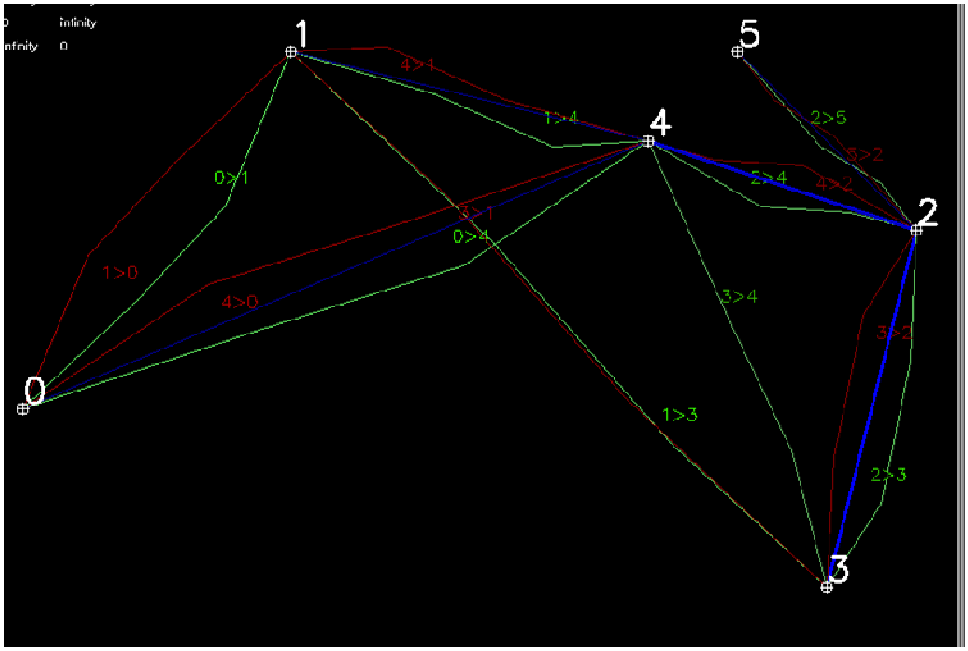
II.2. Evaluation des performances

i) Vérification de l'admissibilité

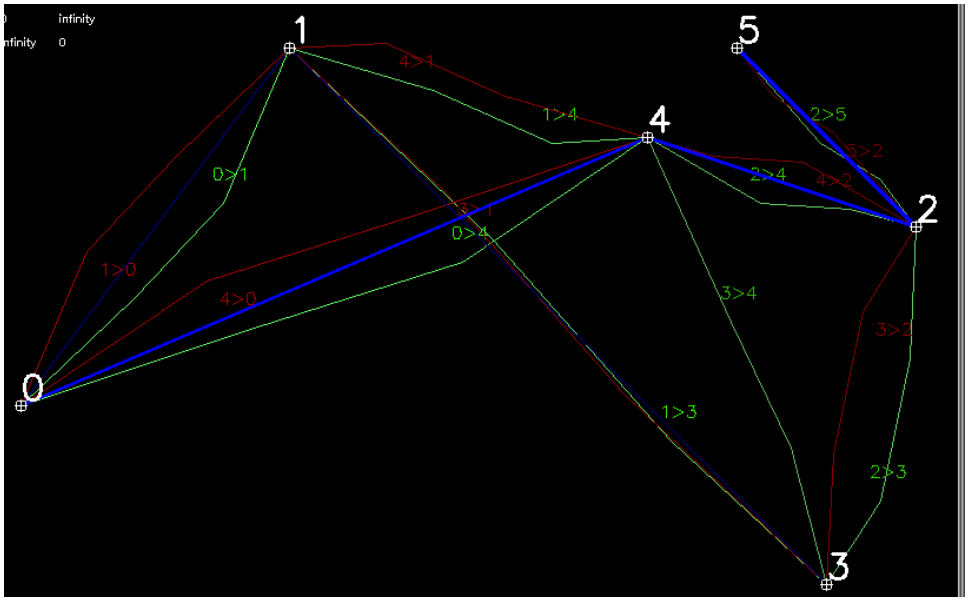
- **Visualisation des résultats**

Dans un premier temps, pour les phases de test de l'implémentation des algorithmes, nous avons construit une carte simple de 5 nœuds dont les routes ne sont pas toutes à double sens.
Le programme prend alors en entrée cette carte, un nœud source et un nœud d'arrivée.

Voici deux exemples de résultats :

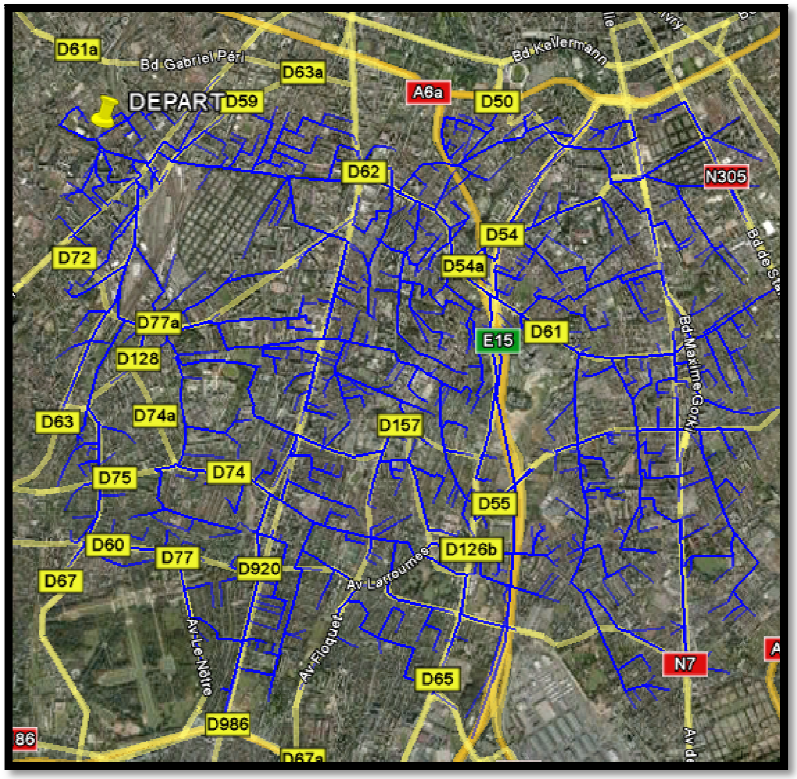


Chemin de 4 vers 3 (en bleu clair) – on observe que, une route directe n'existant pas de 4 vers 3 (seule 3>4 existe), l'algorithme choisit de passer par 2.

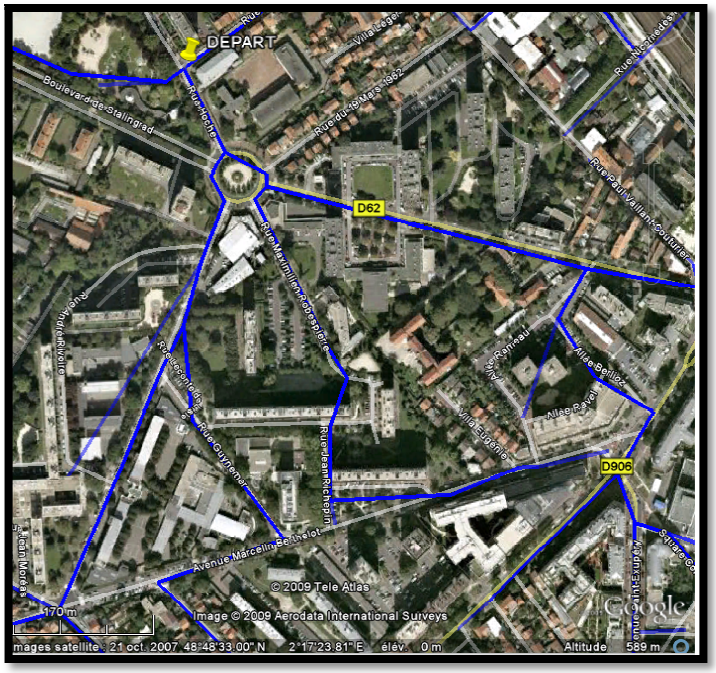


Chemin de 0 vers 5 (en bleu clair) – à première vue, l'algorithme semble trouver effectivement le plus court chemin de 0 vers 5

Dès lors que les résultats sont probants, nous nous sommes attachés à traiter des données réelles en passant par la conception du traducteur de fichiers XML.
L'export des résultats se fait alors au format standard KML[8] afin de contrôler le traducteur à l'aide de logiciels d'affichage cartographique.



Aperçu de l'ensemble des plus courts chemins à partir d'un carrefour source, algorithme de Dijkstra (Région de Cachan, 94 – images Google Earth)



Détail de la cartographie précédente (Région de Cachan, 94 – images Google Earth)

[8] KeyHole Markup Langage, langage basé sur la structure XML destiné à la gestion de l'affichage de données géospatiales.

Ces images permettent de vérifier le bon fonctionnement du traducteur de fichiers XML. En effet nous constatons ici que les routes et carrefours relevés par l'algorithme concordent avec le réel.

Les algorithmes permettent également de n'exporter qu'un chemin à destination unique. Cela nous sera utile pour contrôler plus en détail l'admissibilité de la solution trouvée, afin de vérifier l'intégration des distances par le traducteur.

Par exemple :

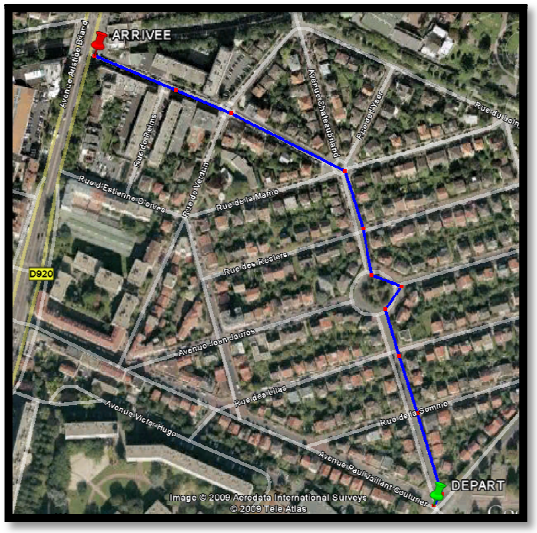
Chemin calculé le plus court de l'entrée de l'ENS Cachan vers la station RER Bagneux (image Google Earth)

Ici nous constatons que l'algorithme a choisi le chemin le plus proche du parcours à vol d'oiseau. Après vérification manuelle, nous confirmons que ce chemin calculé est le chemin réel le plus court.

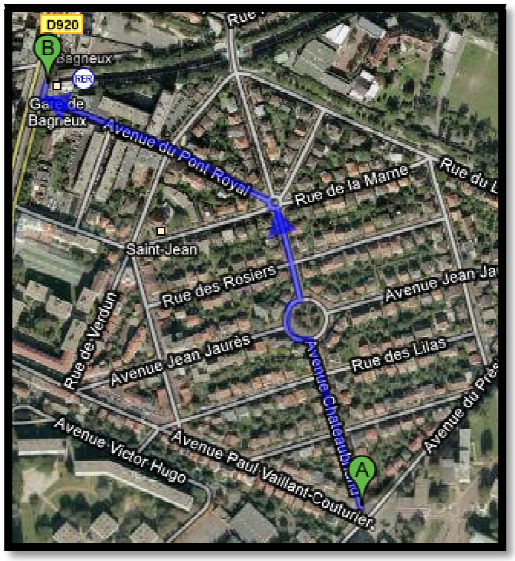
L'ensemble des résultats présentés dans cette partie 0 concerne les algorithmes de Dijkstra et de Floyd-Warshall. En effet, une étude comparative automatisée des résultats démontra que les résultats des deux algorithmes, une fois implémentés, sont parfaitement identiques.

ii) Comparaison avec moteurs de recherche existants

Reprenons l'exemple du chemin calculé le plus court de l'entrée de l'ENS Cachan vers la station RER Bagneux, et comparons-le au résultat donné par l'utilitaire *Google Maps*[9] :



Chemin calculé par nos algorithmes

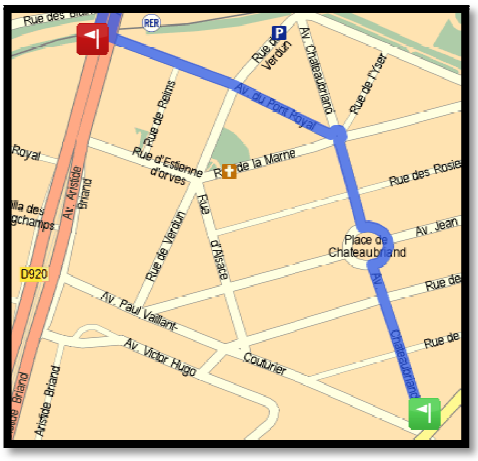


Chemin calculé par Google Maps (option « à pied »)

Nos résultats sont, si l'on se fie à Google Maps, corrects. Cependant, en utilisant l'option *« En voiture »*, Google Maps conseille de passer par l'avenue Paul Vaillant-Couturier.
Une explication peut-être que ce dernier prend en compte les limitations de vitesse, et donne ainsi un avantage à la route D920 (encadré jaune). La priorité est alors de rejoindre cette route.

Voyons ce qu'indique l'utilitaire *Mappy*[10] qui propose une option « vélo » :

Chemin calculé par nos algorithmes



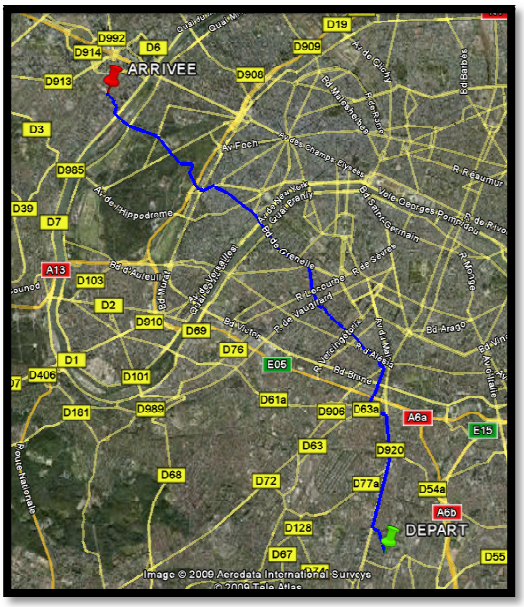
Chemin calculé par Mappy (option « Vélo »)

[9] Maps.google.fr
[10] Fr.mappy.com

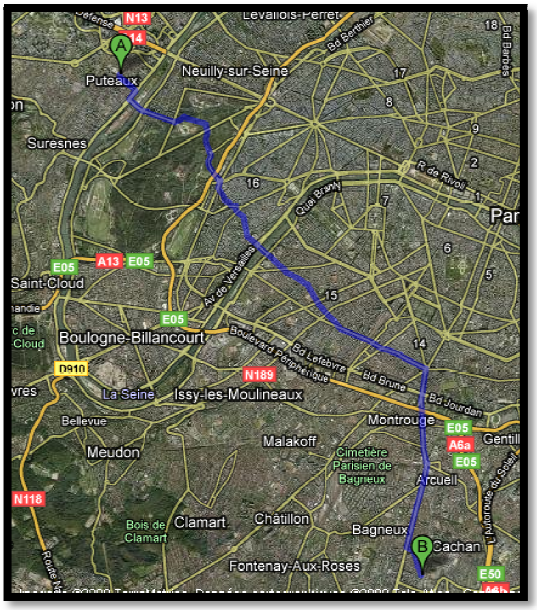
Fait intéressant, Mappy conseille le même itinéraire si l'on choisit l'option « Véhicule ».

Vérifions enfin nos résultats sur des étendues plus conséquentes.

Nous considérons maintenant un chemin de l'entrée de l'ENS Cachan à un carrefour dans le quartier de La Défense 11. Soit un chemin de près de 20km.



Chemin calculé par nos algorithmes



Chemin calculé par Google Maps (option « à pied »)

Le résultat est ici très probant !

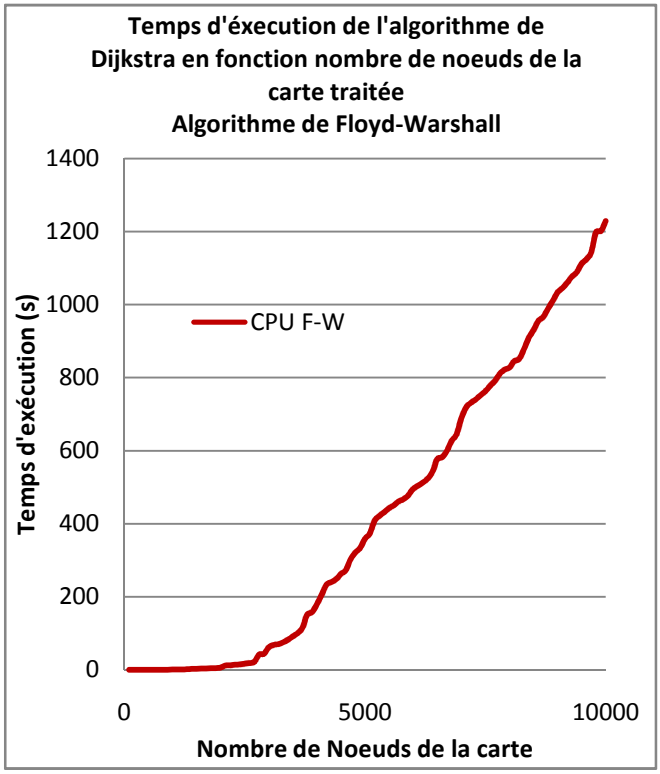
(En revanche, sous l'option « en voiture », Google Maps conseille d'emprunter le boulevard périphérique ou même l'A86….)

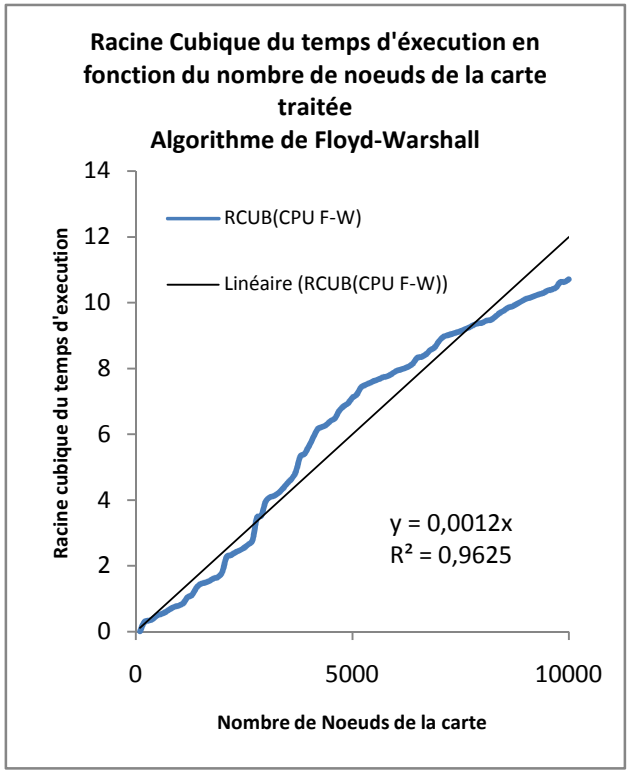
iii) Comparatif : théorie contre pratique, Floyd-Warshall contre Dijkstra

Nous avons jusqu'ici considéré les algorithmes de Floyd-Warshall et de Dijkstra comme équivalents en termes de résultats. Mais voyons maintenant le côté pratique : il semble impératif de comparer les performances de ces algorithmes en termes de vitesse d'exécution.

Pour cela, nous étudierons le temps d'exécution de chacun de ces deux algorithmes pour des tailles de carte différentes. Nous quantifierons cette taille par le nombre de carrefours qu'elles contiennent.

- **Algorithme de Floyd-Warshall**

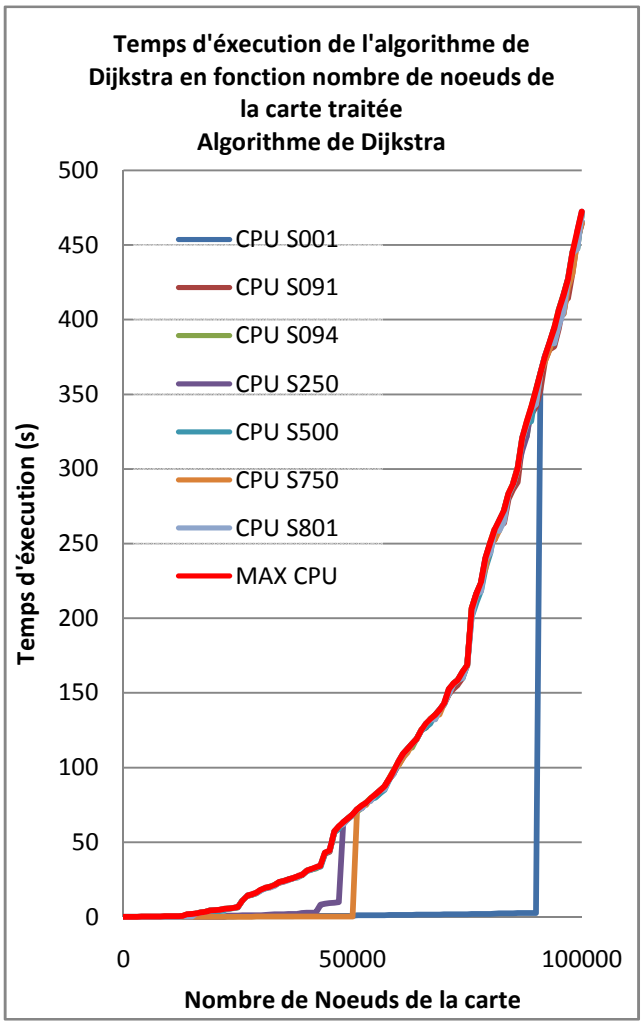


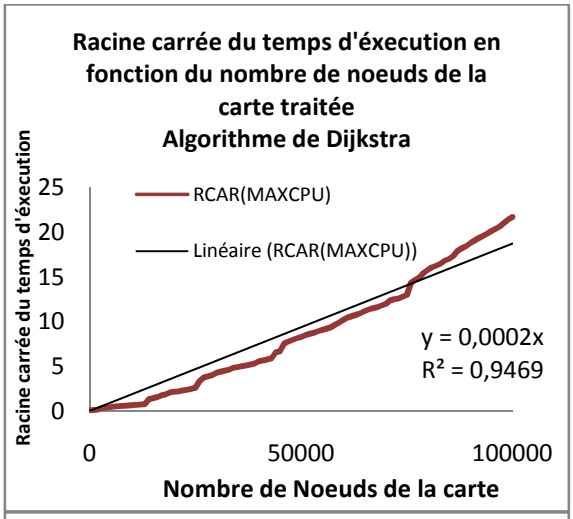


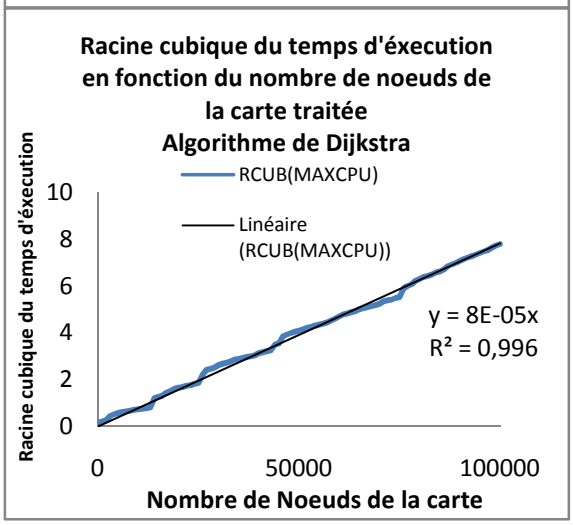
Ici, alors que l'algorithme de Floyd-Warshall est assez difficile à estimer.
Racine carrée → rien
Racine cubique → rien
Logarithme → rien

- **Algorithme de Dijkstra**

L'algorithme est exécuté à partir de divers points sources et calcule l'ensemble des plus courts chemins de la carte.







La théorie montre que l'algorithme de Dijkstra est en $O(n^2)$. Or ici, il apparaît clairement que le temps d'exécution de l'algorithme dépend linéairement du nombre de nœuds traités, en première approximation.

L'intérêt des différentes séries de mesure est montré sur le premier graphique.

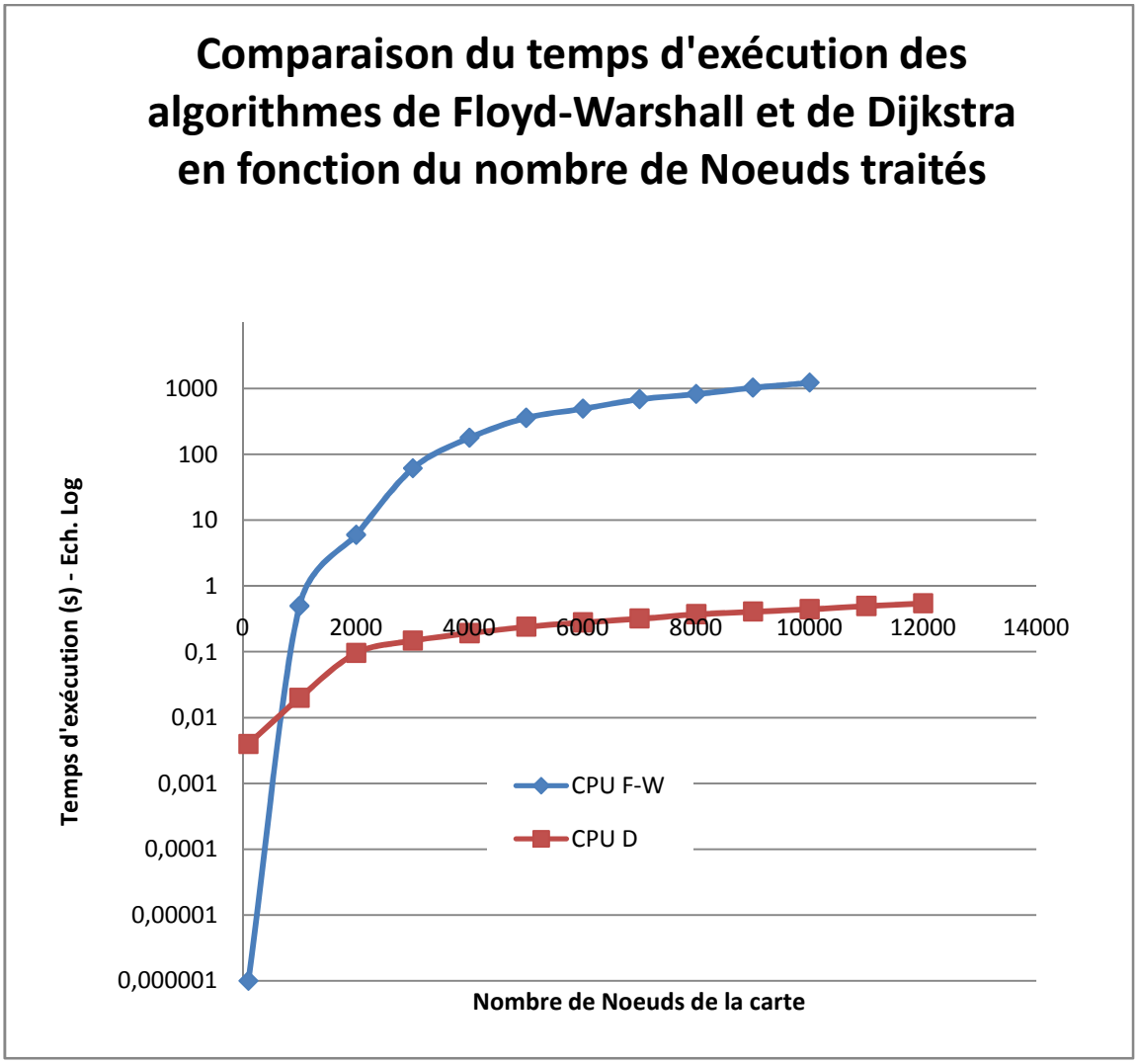
Des raisons possibles de ces écarts à la théorie sont développées en Annexe.

- **Comparaison**

	Floyd-Warshall		**Dijkstra**	
	Théorique	Application	Théorie	Application
Complexité	$O(n^2)$	$O(n^3)$	$O(n^2)$	$O(n^3)$
Mémoire demandée	$O(n^2)$	*???*	$O(n^2)$	*???*

Nous ne sommes pas parvenu à mesurer les valeurs d'occupation de la mémoire.

Comparons maintenant les temps d'exécution des deux algorithmes :



L'avantage est ici à l'algorithme de Dijkstra, nettement plus rapide que celui de Floyd-Warshall.

III. Conclusions

III.1. Perspectives : optimisation de la recherche

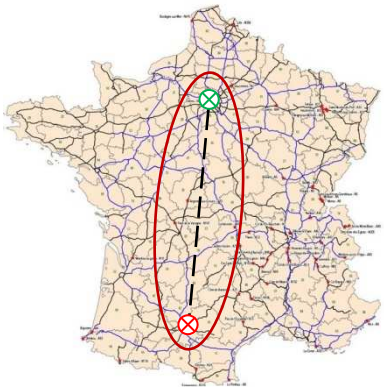
i) Faisceaux, élimination de nœuds

Une manière évidente d'augmenter la vitesse d'exécution de la recherche est de réduire le nombre de données traitées. Cette réduction doit-être faite en fonction du trajet demandé.

Ainsi, il est envisageable de définir un faisceau délimitant une zone géographique englobant les points de départ et d'arrivée, et de le centrer autour du parcours « à vol d'oiseau ». Les nœuds situés en dehors de se faisceau ne seront pas traités par l'algorithme.

Exemple : aperçu de l'optimisation par faisceau.

Carte complète avec nœuds de départ et d'arrivée et parcours à vol d'oiseau



Carte complète avec faisceau

Carte réduite avec nœuds de départ et d'arrivée

On peut raisonnablement penser qu'une telle méthode réduirait très sensiblement le coût de l'algorithme.
En effet, dans l'exemple précédent, le nombre nœuds traités n et divisé par 4 environ. Dans le cas d'un algorithme de complexité en $O(n^2)$, cela équivaut en première approximation à une division par 16 du temps d'exécution.

Divers tests seraient à effectuer afin de définir une forme et taille optimales pour ce faisceau. Le positionnement de ce dernier par rapport aux nœuds de départ et d'arrivée est également à étudier.

Il est important de noter que la réduction de carte par faisceau n'assure plus l'admissibilité du résultat fourni par l'algorithme.

ii) Changement d'algorithme

Si l'on arrivait à définir un critère de convergence de A*, il serait alors possible d'effectuer une pré-traitement de la carte afin de décider (ou non) de préférer cet algorithme aux autres admissible. En effet, l'algorithme A* est, en théorie, beaucoup plus rapide que les deux autres.

iii) Multi-résolutions

L'idée ici est très simple : constituer plusieurs cartes d'une même région, mais à des niveaux de précision différents.
Par exemple, l'une ne serait composée que des Autoroutes, l'autre des Autoroutes et des nationales, et ce jusqu'à une dernière carte comprenant l'ensemble du réseau routier.

Alors nous procèderons par étapes successives :
- Localiser les carrefours les plus proches des points demandés sur la carte la plus grossière,
- Effectuer le calcul sur cette carte ,
- Localiser sur une carte moins grossière les carrefours les plus proches de ceux trouvés en première étape ,
- Effectuer le calcul sur cette carte,
-etc.

iv) Dallage

Le dallage permet de réduire l'espace mémoire nécessaire. On divise une région en plusieurs cartes de même résolution. On détermine ensuite les cartes concernées par les points de départ et d'arrivée demandés.
On effectue alors le calcul sur ces dernières cartes seulement.

v) D'autre considérations

L'altitude est certes intéressante pour un cycliste, mais d'autres paramètres peuvent lui être utiles. Par exemple, la fréquentation des routes, certains points d'intérêt…

III.2. Conclusion

La démarche de notre recherche s'est portée sur trois grands points :
- modélisation formelle du problème : comment résoudre le problème ?
- recherche et traitement des données
- implémentation de la résolution

Au départ, nous avions pensé que la programmation des algorithmes serait la seule partie de ce T.E.R.
L'avancement de notre recherche nous a mené à découvrir de plus en plus de problématiques (notamment pratiques) , par exemple l'extraction de la base de donnée pour la traiter, la gestion de la mémoire due à la taille des matrices d'adjacences même sur une petite partie de la France, etc.

Ces problématiques étaient très intéressantes d'un point de vue de recherche de solutions, et pourront être réutilisable en tant que stratégie de calcul en mécanique, pour le traitement de nombreux nœuds en éléments finis par exemple.

IV. Bibliographie

Algorithmic Graph Theory - James A.McHugh

Graph Theory, An Algorithmic approach - Chrostofides

*Christofides, pardon

Planar Graphs: Theory and algorithms - T.Nishizeki et N.Chiba

ac-nancy-metz.fr - Eric Sigward

Eléments de Théorie des Graphes et Programmation Linéaire - Didier Maquin, INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

http://brassens.upmf-grenoble.fr/IMSS/mamass/graphecomp/

WIKIPEDIA

http://www.nimbustier.net/publications/djikstra/

Recherche de chemin par l'algorithme A* - Pierre Schwartz

V. Annexes

V.1. Eléments de théorie des graphes

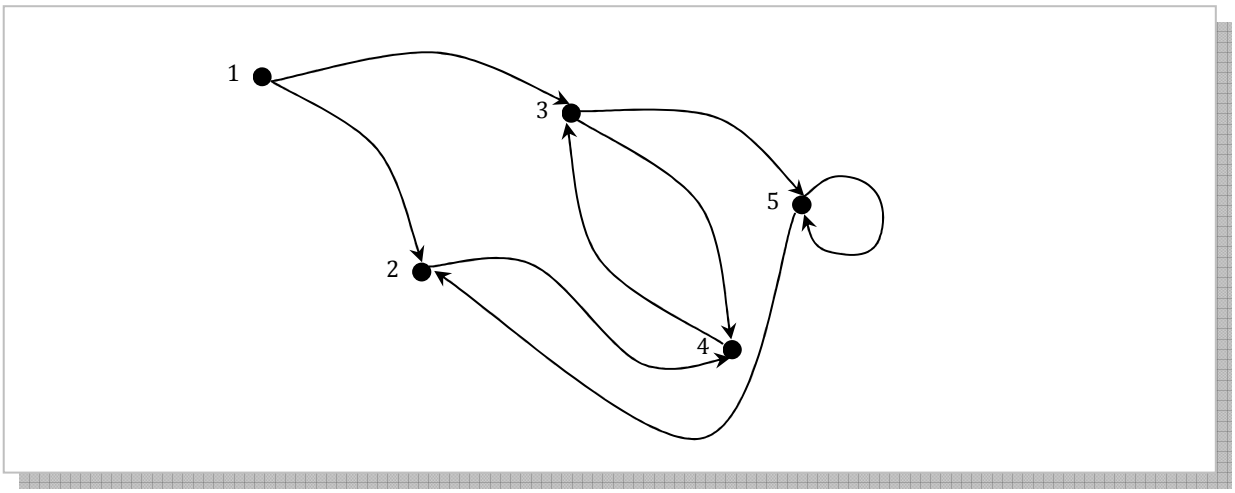
i) Définition 1 : Graphe orienté ou Digraphe

On appelle graphe orienté $G = (X, A)$ la donnée d'un ensemble X dont les éléments sont appelés sommets et d'une partie A de $X \times X$ dont les éléments sont appelés arcs ou arêtes.

En présence d'un arc $a = (x, y)$ qui peut être noté simplement xy, on dit que :
- *x est l'**origine** (ou extrémité initiale),*
- *y l'**extrémité** (terminale) de a*
- *a est **sortant** en x,*
- *a est **incident** en y,*
- *y est **successeur** de x,*
- *x est **prédécesseur** de y,*
- *x et y sont **adjacents**.*

Ainsi, dans une première modélisation, un nœud représentera un carrefour ($x \in \mathbb{R}^2$ défini par ses coordonnées, par exemple) alors qu'un arc sera l'abstraction d'une route à sens unique. Pour une route à double sens, il suffit d'introduire dans A les arcs xy et yx.

Exemple1 : exemple de graphe orienté



ii) Définition 2 : Matrice d'adjacence

Soit le graphe orienté $G = (X, A)$ *avec* $= \{x_1, x_2, \dots, x_n\}$ *et* $A = \{a_1, \dots, a_m\}$.
On appelle matrice d'adjacence la matrice $M = (m_{ij})$ *de dimension* $n \times m$ *telle que :*

$$m_{ij} = \begin{cases} 1 \; si \; x_i x_j \; \in A \\ 0 \; sinon \end{cases}$$

Cette représentation mathématique d'un graphe semble attrayante d'un point de vue algorithmique.

<u>*Dans l'exemple 1, on a :*</u>

Matrice d'adjacence :

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

iii) Définition 3 : Matrice des distances ou Matrice des couts minimums

Soit le graphe orienté $G = (X, A)$ *avec* $X = \{x_1, x_2, \dots, x_n\}$ *et* $A = \{a_1, \dots, a_m\}$.
On appelle matrice des distances la matrice $M = (m_{ij})$ *de dimension* $n \times m$ *telle que :*

$$m_{ij} = \begin{cases} min\{l(c)/c \; est \; un \; chemin \; de \; x \text{ à } y\} \; si \; x_i x_j \; \in A \\ \infty \; sinon \end{cases}$$

La résolution d'un problème de plus court chemin entre $x_i \; et \; x_j$ consiste à trouver un m_{ij} de cette matrice des distances, parfois appelées *matrice des couts minimums*. Cependant, dans notre problème, il sera nécessaire également de connaitre le chemin $c_0 \; tel \; que$:

$$l(c_0) = min\{l(c)/c \; est \; un \; chemin \; de \; x \text{ à } y\}$$

<u>*Dans l'exemple 1, on a :*</u>

Matrice des distances :

$$M = \begin{pmatrix} \infty & 1 & 1 & 2 & 2 \\ \infty & \infty & 2 & 1 & 3 \\ \infty & 2 & \infty & 1 & 1 \\ \infty & 3 & 1 & \infty & 2 \\ \infty & 1 & 3 & 2 & 1 \end{pmatrix}$$

On peut également introduire la notion de *matrice d'adjacence pondérée*.

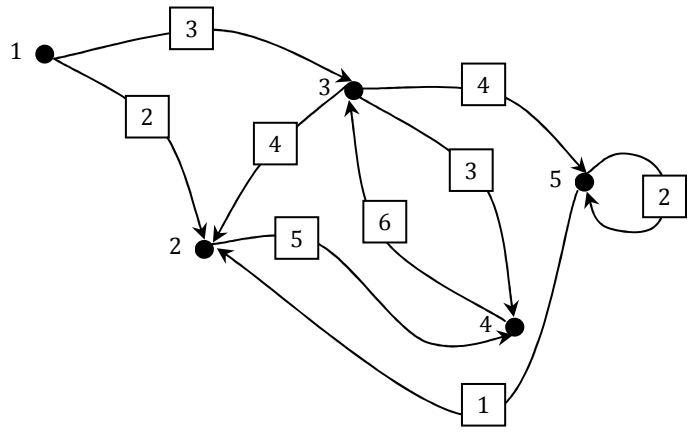
iv) Définition 4 : Matrice d'adjacence pondérée

Soit le graphe orienté $G = (X, A)$ *avec* $X = \{x_1, x_2, \dots, x_n\}$ *et* $A = \{a_1, \dots, a_m\}$.
On affecte une longueur l_j *à l'arc* a_j *et on définit la matrice d'adjacence pondérée* $M = (m_{ij})$ *de dimension* $n \times m$ *telle que :*

$$m_{ij} = \begin{cases} l_k \; si \; x_i x_j = a_k \; \in A \\ \infty \; sinon \end{cases}$$

Cette approche permet de s'affranchir de la prise en compte du profil plan de la route dans la matrice d'adjacence. En effet la sinuosité de la route n'est pertinente dans notre problème que dans l'augmentation de distance qu'elle amène, qui est alors incorporée dans l_j. Ainsi, après traitement, la quantité de données à manipuler en est considérablement réduite.
Il est alors aisé de définir une *matrice des distances pondérées*.

Exemple2 : exemple de graphe orienté pondéré



Matrice d'adjacence pondérée :

$$M = \begin{pmatrix} \infty & 2 & 3 & 0 & 0 \\ \infty & \infty & \infty & 5 & \infty \\ \infty & 4 & \infty & 3 & 4 \\ \infty & \infty & 6 & \infty & \infty \\ \infty & 1 & \infty & \infty & 2 \end{pmatrix}$$

Matrice des distances pondérées :

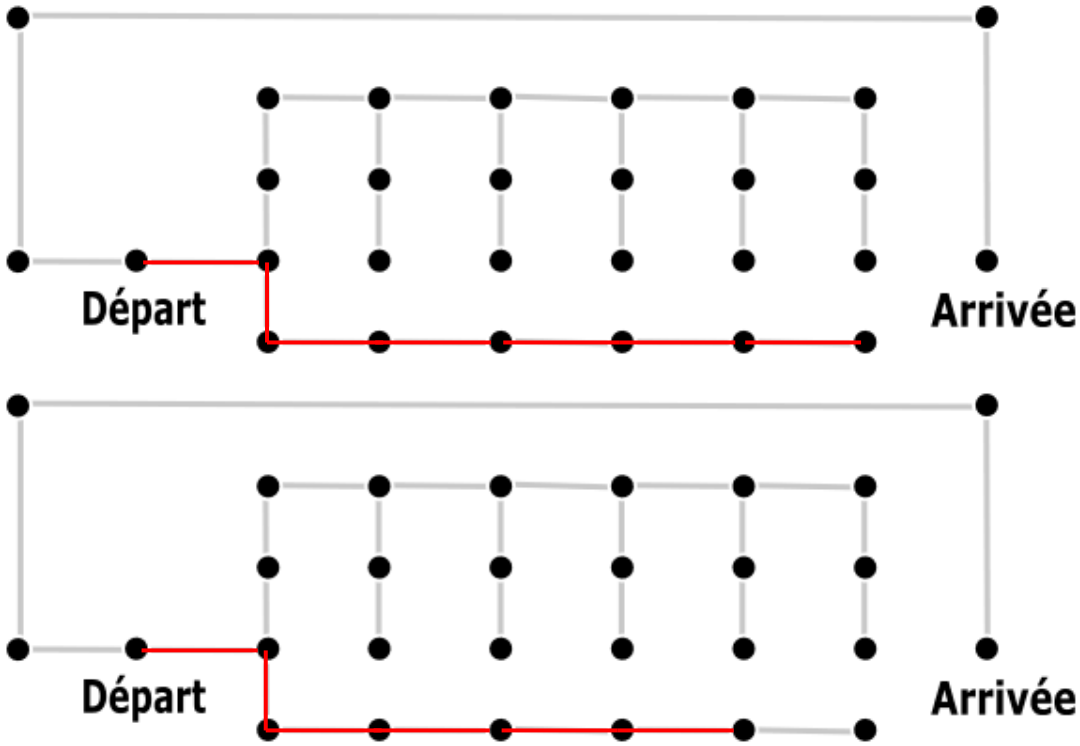
$$M' = \begin{pmatrix} \infty & 2 & 3 & 6 & 7 \\ \infty & 13 & 11 & 5 & 15 \\ \infty & 4 & 9 & 3 & 4 \\ \infty & 10 & 6 & 9 & 10 \\ \infty & 1 & 12 & 6 & 2 \end{pmatrix}$$

*V.2. Amélioration de l'algorithme A**

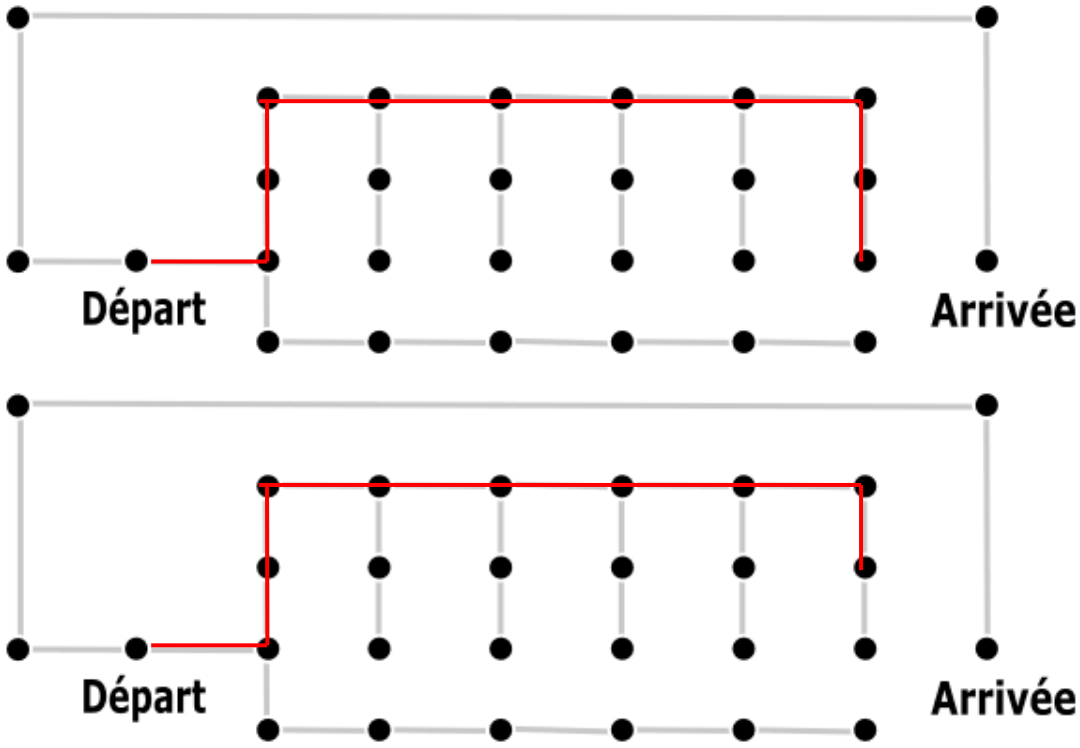
Initialisation
Minimum = infini

Algorithme
Pour *i jusqu'au nombre de nœuds* ***Faire***
Pour *j jusqu'au nombre de nœuds* ***Faire***
Si *existe* x_{ij} ***Faire***
Si minimum < dist(x_i ,x_j)
minimum = dist(x_i, x_j)
Lorsque *la variable "minimum" est minimum* ***Relever*** *j*
Liste[i]= x_i
Lorsque *Liste[i]= Liste[i-2]* ***Faire***
i = i-2
et
Supprimer *le chemin allant de i-2 à j*

On change l'algorithme tel que si l'on parcourt deux fois le même point, on considérera que le chemin menant à ce point n'existe plus. On illustre ci-dessous cet algorithme

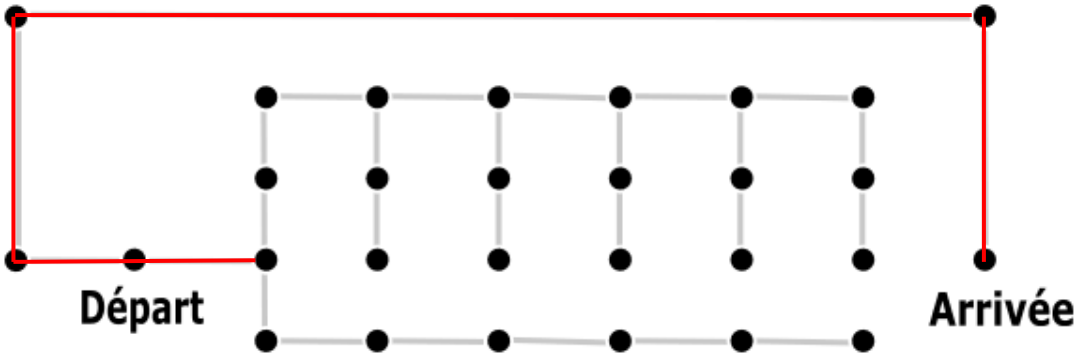


....etc...Jusqu'à :



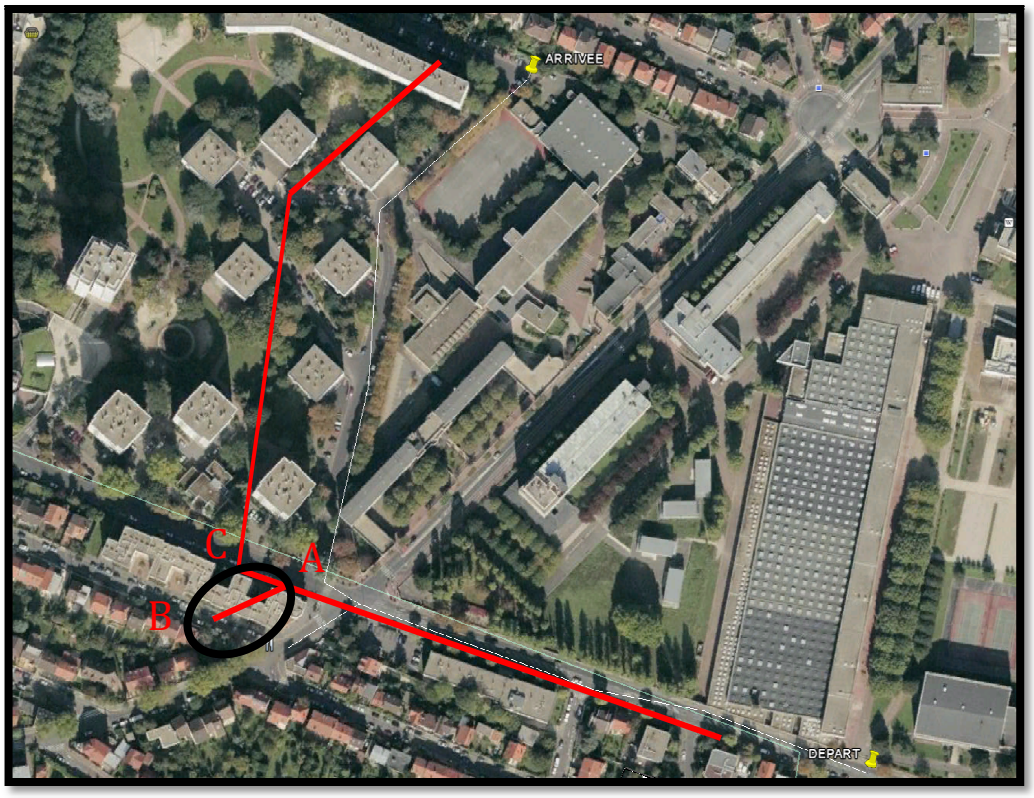
...etc...

Pour arriver, enfin :



- **Exemple réel**

Cas où l'algorithme fonctionne sans problème (image Google Earth)



Sans l'amélioration, on observe une oscillation (image Google Earth)

Lorsque que l'on est en A, la distance entre A à B + B à Arrive < entre A à C + C à Arrive donc on va en B.

Lorsque que l'on est en B, la distance entre B à A + A à Arrive < entre B à C + C à Arrive donc on va en A.

D'où l'oscillation. (Le dessin n'est pas convaincant!)

V.3. Théorie contre pratique

i) Code

- **Temps de test**

Nous avons rencontré beaucoup de problèmes pour réaliser notre code dus en particulier à la découverte de l'environnement Linux, des problèmes de traitements de la base de données, de l'équilibre difficile à trouver entre faire un test sur une petite zone et avoir des résultats peu convaincant, ou attendre que des tests plus larges soient effectués. On peut aussi mettre en avant les difficultés dues au codage en lui-même, et à la validation (tout du moins à la vérification) de nos résultats.

ii) Données

- **Connexité des morceaux**

Nous avons pu rencontrer des problèmes lors de nos tests dus au fait que nous coupons des sources de données de manière arbitraire, mais aussi par le fait que les données ne sont pas

complètes. En effet, celles-ci étant rentrées par des contributeurs bénévoles, elles ne sont pas en quantité suffisante.

Une méthode pour tester la pertinence de nos résultats est de calculer la connexité du réseau formé par nos données. On peut postuler que toutes routes sont normalement connexes. En effet, il paraît difficile d'aller sur une route, si il n'y a pas de route pour y aller.

Nous avons donc développé un outil qui nous permet de montrer la connexité des réseaux. Pour l'Ile de France, on trouve 421 réseaux pour les 140000 points définis, ce qui veut dire qu'on esquive environ 2000 points sur le total.

- **Temps de lecture pour l'algorithme de Dijkstra**

orsque nous avons découpé nos données pour tracer l'évolution du temps de calcul en fonction du nombre de points, nous n'avons pas eu le temps de découpé correctement les données en ne sélectionnant que celles qui sont connexes. Nous avons pu constaté que pour 10000 points répartis sur l'Ile de France, nous avions environ 500 réseaux différents. La résolution de l'algorithme de Dijkstra est considérablement accélérée lorsque les réseaux ne sont pas connexes. En effet, dès que l'on sait qu'on ne trouve plus de routes vers un point hors du réseau, on s'arrête. Ce qui fait que la complexité est réduite.