Animation d'algorithmes de parcours de graphes

TER de Maîtrise d'Infomatique de l'Université Montpellier II

Olivier Beau Michaël Belheur Nicolas Rivoallan

25 avril 2001

Table des matières

Remerciements

Nous tenons à remercier :

Michel Habib (encadrant) pour ses explications et les nombreuses documentations qu'il nous a fourni, pour la partie algorithmique.

Marc Nanard (encadrant) pour les spécifications précises qu'ils nous a fourni tout au long du développement.

Jean-Francois Baget et Denis Payet pour nous avoir apporté de l'aide technique lors de nos questions.

Chapitre 1

Introduction

1.1 Cahier des charges

Le but de ce projet était de réaliser une animation de parcours de graphe par des algorithmes. Pour résoudre cela, on prendra un parcours générique simple que nous spécifierons afin d'obtenir différents algorithmes. Il faudra ensuite animer l'évolution du parcours, ce qui permettra de comprendre et de comparer les variantes.

Nous aurons bien entendu à déterminer ce qui est pertinent à animer, en fonction de chaque algorithme. L'ensemble de l'application sera en JAVA, tandis que les fichiers générés utiliserons le XML. Le projet sera extensible à d'autres algorithmes.

Initialememt prévu pour deux étudiants, les difficultés de répartitions des étudiants sur les projets nous ont contraint de nous retrouver à trois sur ce projet. Toutefois nous pouvons dire aujourd'hui, alors que la réalisation du projet touche à sa fin, que nous ne nous sommes pas du tout sentis en sûrnombre pour le réaliser.

1.2 Description des differentes parties du projet

Notre projet se subdivise en trois grandes parties : une partie dite éditeur, une partie algorithme et une partie animation. La partie éditeur permet de créer un graphe avec ses sommets et ses arcs, puis de le sauvegarder. Cette partie pourra être considérée comme un serveur de graphes. Vient ensuite la partie algorithme, c'est ici que s'effectuera l'exécution d'un algorithme sur un graphe. De cette exécution ressortira la trace de l'algorithme. Nous assimilerons cette partie à un serveur de traces d'algorithme. Enfin la dernière partie permet de visualiser l'exécution d'un algorithme à partir de sa trace, éventuellement pas à pas, voire à l'envers. Cette partie est assimilable à un client. Actuellement, notre application ne forme qu'un seul et unique bloc, cependant nous l'avons conçue pour qu'elle soit facilememt subdivisée en trois sous-applications. Le portage vers une architecture client/serveur permettrait une bonne adaptation au profil

pédagogique étudiants/professeur.

Chaque membre s'est occupé d'une partie du projet
- Michaël pour la structure du graphe et du XML
- Nicolas pour l'implémentation des algorithmes
- Olivier pour l'interface graphique

1.3 Description des différents types d'algorithmes traités

En ce qui concerne les algorithmes traités, nous trouvons actuellement les suivants : Dijkstra, A*, A* vol d'oiseau, Largeur et Lexicographique. Parmis ceux-ci, il faut noter que certains sont des algorithmes de recherche du plus court chemin, alors que d'autres sont des algorithme de parcours en largeur du graphe. Dans le premier groupe nous trouverons ainsi Dijkstra, A* et A* vol d'oiseau, alors que dans le second groupe nous utiliserons Largeur et Lexicographique. Tous ces algorithmes parcourent au moins une fois chaque sommet à l'exception des algorithmes de type A*. En effet ceux-ci se basent sur une heuristique associée à chacun des sommets du graphe. De cette manière l'algorithme ne passe pas par des sommets dont on sait à l'avance qu'ils n'ont pas d'intérêt (en supposant bien entendu que les heuristiques soient bien choisies !). Ces cinq algorithmes seront par la suite décrits plus en détail avec une mise en valeur des points qui leur sont communs et qui les différencient.

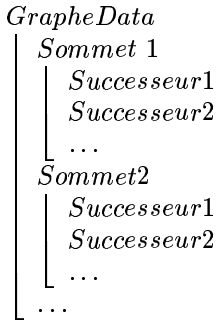
Chapitre 2

Structure de graphe employée

2.1 Graphe statique

Nous avons choisi le terme "graphe statique" lorsque nous nous référons à un graphe au sens mathématique du terme. C'est à dire un ensemble de sommets reliés entre eux par des arrêtes. Le terme statique a été choisi en opposition avec le terme "dynamique", dont nous verrons plus loin la signification.

Dans notre application, nous avons représenté un graphe (classe `GrapheData`) comme étant un ensemble de sommets (classe `Sommet`), chacun pouvant éventuellement posséder un ou plusieurs successeurs (classe `Successeur`). Un `GrapheData` peut donc se visualiser ainsi



Chacun de ces éléments présente des caractéristiques :

- Un sommet possède donc des successeurs, mais aussi un nom, une coloration et une heuristique
- Un successeur, quant à lui, pointe sur le sommet destination de l'arc, mais possède également une coloration et un poids

A partir d'une telle structure, il nous est donc possible de faire tourner des algorithmes.

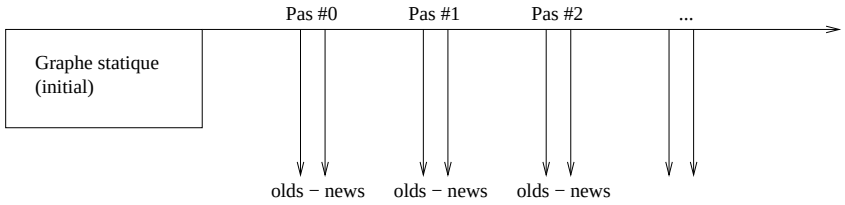
2.2 Graphe dynamique (trace d'algorithme)

Qu'est-ce qu'un graphe dynamique ? Dans le cadre de notre application, un graphe dit dynamique sera utilisé pour représenter l'évolution d'un algorithme sur un graphe (statique). Autrement dit, il s'agit là de conserver la trace de cet algorithme afin d'obtenir l'avancement de ce dernier à chaque instant.

Pour résoudre cela, nous avons considéré qu'un algorithme peut être divisé en une succession d'étapes. Ainsi, pour obtenir la trace de l'algorithme, il suffit de sauvegarder l'état du graphe (statique) à chaque étape. De plus, à des fins de clarté et d'optimisation, au lieu de sauvegarder la totalité du graphe à chaque étape, nous avons opté pour ne conserver que les modifications effectuées depuis l'étape précédente.

Au sein de notre application, un graphe dynamique (classe `DynamicGraphData`) va donc posséder un graphe statique qui sera le graphe à l'état initial, ainsi qu'une succession de pas (classe `DynamicData`). Pour chaque pas, nous générons une suite de sommets dédoublés, avant puis après modification par l'algorithme. Une arrête étant portée par un sommet parent, pour sauver une arrête il suffit de sauver son parent.

Un `DynamicGraphData` peut donc se visualiser ainsi



Certain algorithme demandent un sommet de départ et parfois un sommet de destination. Nous avons donc simplement ajouté deux propriétés qui portent le nom de ces sommets au graphe dynamique. A partir du moment où nous sommes dans un graphe dynamique, les sommets possèdent en plus une propriéte `poids` qui, dans la pluspart des algorithmes, represente la distance entre l'origine et ce sommet.

Il est intéressant de noter que lorsque nous sauvegardons l'état du graphe en cours d'exploration, nous sauvons également l'état **avant** la modification. De cette manière nous pouvons aussi dérouler la trace de l'algorithme dans le sens contraire du déroulement de l'algorithme.

2.3 Description du graphe en XML

Les deux structures que nous avons décrite précédemment sont également exportables sur une unité de stockage au format XML. Ainsi un utilisateur lambda, peut générer un graphe ou une trace et les sauvegarder ou les transférer à un tier.

Le choix du format XML nous a été demandé dans le cachier des charges car c'est un format qui présente des avantages certains

- il est portable d'une plate forme à une autre
- il tend à devenir un standard dans le monde internet
- il est adapté aux structures arborescentes
- ce n'est pas un format propriétaire (lisible dans un editeur de texte classsique)

Nous devons cependant noter un inconvénient, et pas de moindre : plusieurs sociétés présentes sur Internet proposent des `packages` afin d'accéder au XML à partir du JAVA. Mais ils ne se sont pas normalisés entre-eux, ainsi lire ou écrire du XML sera plus facile chez *Apache* (avec `xerces`) que chez *Sun* (avec `Jaxp`). A retenir...

Lorsque nous exportons un graphe statique, il nous a semblé plus simple d'exporter un graphe dynamique avec uniquement le graphe initial (donc pas d'étapes). Ainsi, voici la représentation du squelette d'un graphe (qu'il soit statique ou dynamique) :

```
<Graphe Dynamique>
  <Graphe Statique>
    <Sommet 1>
      <Arrete 1>
      <Arrete 2>
      <...>
    <Fin Sommet 1>
    <...>
  <Fin Graphe Statique>
  <Etape 1> (si le graphe sauvegarde est dynamique)
    <Avant modifications>
      <Sommet x>
        <Arrete y>
        <...>
      <Fin Sommet x>
      <...>
    <Fin Avant modifications>
    <Apres modifications>
      <Sommet x>
        <Arrete y>
        <...>
      <Fin Sommet x>
      <...>
    <Fin Apres modifications>
  <Fin Etape 1>
  <...>
<Fin Graphe Dynamique>
```

Chapitre 3

Partie Algorithmique

3.1 Introduction

Nous allons travailler tout le long de cette partie avec cinq algorithmes construits tous à partir d'un algorithme plus général. C'est en leur appliquant différentes propriétés que nous allons les différencier. Cet algorithme plus général possède donc une structure commune basée sur deux ensembles que nous nommerons *OUVERTS* et *FERMES*, ils représenteront respectivement les sommets en cours de traitement, et les sommets déjà traités.

Nous allons montrer que le goulot d'étranglement de complexité provient de la gestion des *OUVERTS*, et que le choix effectué, dans cette ensemble à chaque étape, sera en $O(log(n))$ si l'on choisit la structure adéquate. Deux familles d'algorithmes seront étudiés ici

- Les algorithmes du plus court chemin
- Les algorithmes de parcours en largeur

Dans chacune de ces parties, nous allons donner différents cas de figures afin de nous faire une idée de l'utilisation de chacun, ainsi que de leur utilité. Nous allons d'abord commencer par expliquer l'algorithme plus général, (cité plus haut), nommé **algorithme de parcours générique**.

3.2 Algorithme de parcours générique

3.2.1 Présentation

On considère un graphe orienté $G = (X, U)$ et l'on suppose les arcs munis d'étiquettes c'est à dire une valuation $w : U \to T$, où T est un ensemble muni d'une relation d'ordre total $\ll$, et de deux éléments distingués M (resp. N) représentant un unique élément maximal (resp. minimal). La relation d'ordre total $\ll$ s'interprétant comme

- L'ordre usuel quand T est l'ensemble des entiers
- L'ordre lexicographique lorsque T est un langage

3.2.2 Algorithme de parcours générique

Voici donc la représentation de cet algorithme.

Donnée un graphe orienté $G = (X, U)$, une fonction de coût $w : U \to T$.

Résultat une arborescence des plus courts chemins issus de x.

$OUVERTS \leftarrow x$
$FERMES \leftarrow$
$Parent(x) \leftarrow NIL$
Pour *Tous les* $y \neq x$, **faire** $Parent(y) \leftarrow y$
$d(x) \leftarrow N(element\ minimal\ de\ T)$
Pour $y \neq x, d(y) \leftarrow M(element\ maximal\ de\ T)$
  tant que $OUVERTS \neq$ **faire**
    $z \leftarrow Choix(OUVERTS)$
    $Ajout(z, FERMES)$
    $Explorer(z)$
    $Virez(z, OUVERTS)$
  $Explorer(z)$
  Pour *Tous les voisins* y *de* z **faire**
    si $y \in FERMES$ **alors** *ne rien faire*
    si $y \in OUVERTS$ **alors**
      si $d(z) + w(z, y) < d(y)$ **alors**
        $Parent(y) \leftarrow z$
        $d(y) \leftarrow d(z) + w(z, y)$
      sinon
        $Ajout(y, OUVERTS)$
        $Parent(y) \leftarrow z$
        $d(y) \leftarrow d(z) + w(z, y)$

A l'issue de cet algorithme, on dispose d'une arborescence de racine x, définie à l'aide de la fonction *Parent* et de la fonction $d : X \to T$ qui associe à chaque sommet y, la distance d'un plus court chemin de x à y.

3.2.3 Quelques détails et conventions

On peut constater que l'algorithme s'arrête uniquement lorsqu'il a traité l'ensemble de ses *OUVERTS*. Il ne parcours donc pas forcément tous les sommets du graphe. C'est pourquoi, les graphes que nous traiterons par la suite seront de composante connexe égale à 1, c'est à dire que pour chaque sommet x, il existera un chemin reliant x à l'origine préalablement choisie.

On prendra comme convention M (distance maximale) égale à l'infini et N(distance minimale) égale à 0. A noter aussi que les distances traitées ici seront uniquement des entiers et leurs valeurs seront positives ou nulles afin de faciliter tous les types de calcul.

Lors de l'instanciation de ces algorithmes il faudra leur spécifier un sommet de départ, que l'on nommera *Origine*, et pour certain un sommet d'arrivée que l'on nommera *Arrivée*.

Les fonctions décrites ci-dessus vont être expliquées dans le paragraphe suivant.

Parent (x) renvoie le prédécesseur direct du sommet x

d (x) renvoie la distance actuelle du sommet x à l'*Origine*

Choix (OUVERTS) indique le prochain sommet x à traiter, x appartenant à l'ensemble *OUVERTS*, le choix s'effectue aléatoirement

Ajout (x, FERMES) ajoute l'élément x à l'ensemble des *FERMES*

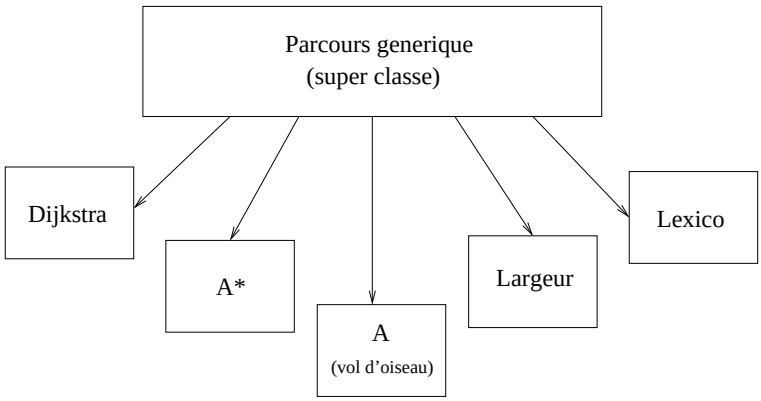
Explorer (x) effectue un parcours de tous les voisins du sommet x.

Virez (x, OUVERTS) enlève le sommet x, x appartenant aux *OUVERTS*

w (z, y) renvoie la valuation de l'arrête entre le sommet z et le sommet y, attention car $w(zy)$ peut être différent de $w(y, z)$ car le graphe est orienté ! !

3.2.4 Point de vue programmation

Le corps de cet algorithme étant le même que les autres que nous allons aborder, la redondance de certaines fonctions nous a poussé à utiliser une représentation hierarchisée. Ainsi l'algorithme de parcours générique sera notre super classe et les algorithmes A*, A , Dijkstra, Largeur et Lexicographique seront nos sous-classes.



3.3 Algorithme du plus court chemin

3.3.1 Algorithme de Dijkstra

Présentation

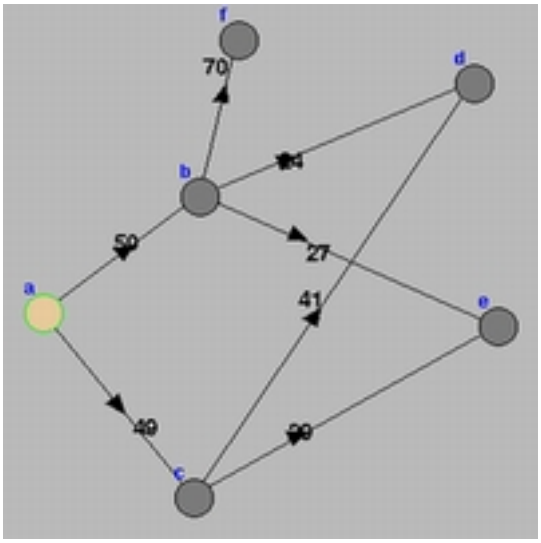
Grâce à la structure décrite ci-dessus, nous allons tranformer l'algorithme du parcours générique en algorithme de Dijkstra. Pour cela on prendra

- T, l'ensemble des réels positifs
- La fonction *Choix ()* fournit un sommet z vérifiant $d(z) = min\ d(y) \mid y \in$ *OUVERTS*

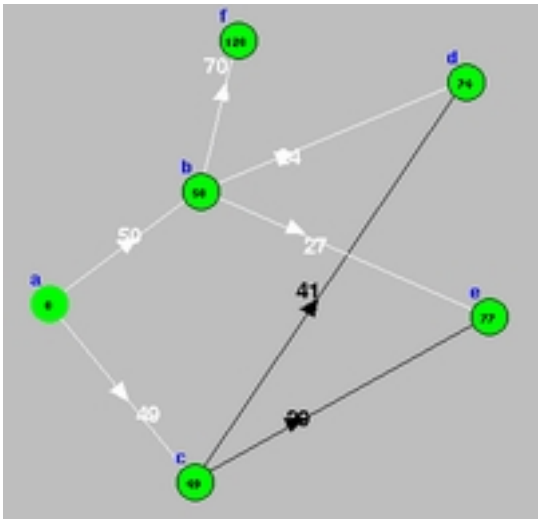
L'algorithme de Dijkstra donne une arborescence des plus courts chemins issus de l'*Origine*. Les sommets ne sont traités qu'une et une seule fois. A chaque étape, l'algorithme détermine le plus court chemin **potentiel** entre son *Origine*

et les points du graphe. Cette remarque est très importante car cela signifie que l'ensemble des plus courts chemins issus de l'*Origine* ne sont obtenus qu'après l'exploration complète des sommets du graphe.

Démonstration de Dijkstra



Début



Fin

Point de vue programmation

Difficultés

– La difficulté du problème provient de la gestion des *OUVERTS*. Ainsi la structure, que nous avons choisie en langage JAVA, est de type *vecteur* qui

se réorganisera à chaque étape par insertion dichotomique en fonction de la distance des sommets
- Pour chaque nouvel élément x modifié, une supression s'effectura suivi d'une réinsertion afin d'obtenir le nouvel ensemble des *OUVERTS* [1]
- Deux variables booléennes vont être utilisées afin de faciliter la recherche de l'appartenance d'un sommet à l'ensemble des *OUVERTS* ou des *FERMES*

complexité
- L'ensemble des *OUVERTS* sera donc classé par ordre croissant à chaque étape. De ce fait, la fonction *Choix* se réalisera avec une complexité minimale en $O(1)$, puisque l'on selectionnera toujours le premier élément du tableau
- Les fonctions *Virez ()* et *Ajout ()* ont une complexité en $O(log(n))$ du fait de leur recherche dichotomique
- Savoir si un sommet appartient à l'ensemble des *OUVERTS* ou à l'ensemble des *FERMES* sera réalisé en $O(n)$, n étant le nombre de sommets de chacun des ensembles

3.3.2 Algorithme A*

Présentation

Nous allons définir certaines propriétés de l'algorithme générique afin de représenter ce nouvel algorithme. Pour cela on prendra
- T, l'ensemble des réels positifs
- La fonction *Choix ()* fournit un sommet z vérifiant $d(z)+h(z) = min\ d(y) + h(y)$ tel que $y \in OUVERTS$, et où $h(y)$ est l'information heuristique de la distance qu'il reste à parcourir
- de même dans la fonction *Explorer*, l'instruction *ne rien faire* est remplacés par

si $d(z) + w(z, y) < d(y)$ **alors**
  $Parent(y) \leftarrow z$
  $d(y) \leftarrow d(z) + w(z, y)$
  $Ajout(y, OUVERTS)$

L'algorithme A* est très intéressant à étudier. La présence de l'heuristique perturbe quelque peu car elle représente une estimation de la distance qui reste à parcourir. Dans les exemples ci-dessous, nous allons montrer toute l'importance du choix de cette heuristique et de son impact dans le déroulement de l'algorithme. Selon le choix, nous nous retrouverons dans l'un des trois cas suivants.
- Soit une distance minimale fausse
- Soit un parcours très long et fastidieux
- Soit une distance exacte et immédiate

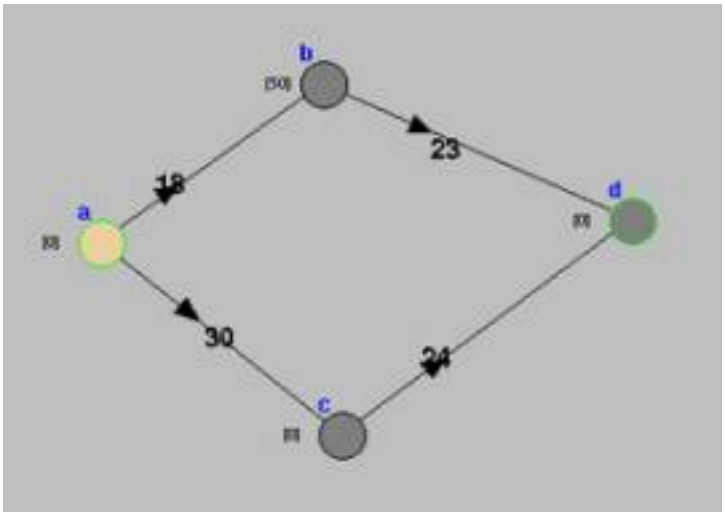
On aura besoin au préalable de choisir en plus d'un sommet *Origine*, un sommet *Arrivée*. La différence essentielle s'effectue dans son arrêt : une fois que le sommet sélectionné sera le sommet d'arrivée, alors l'algorithme stoppe. La lecture du chemin se fera grâce à la fonction *Parent()*, en partant du sommet

[1] A la fin de l'étape l'ensemble des *OUVERTS* sera de nouveau trié.

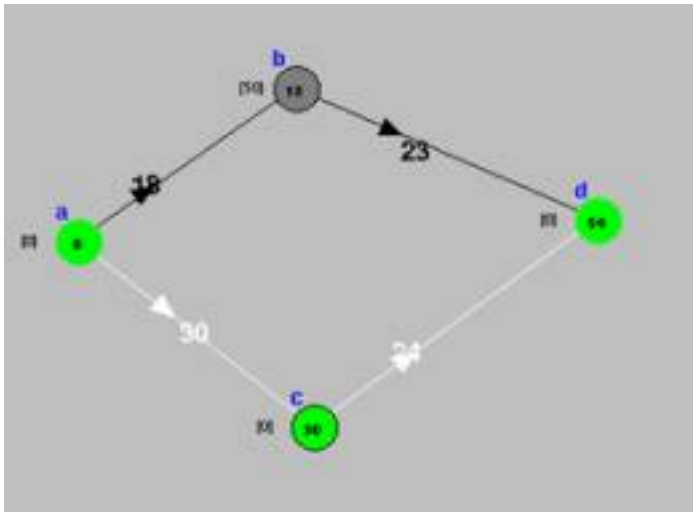
d'arrivée et en remontant jusqu'au sommet de départ. [2]

Démonstration de A*

Distance minimale fausse



Début



Fin

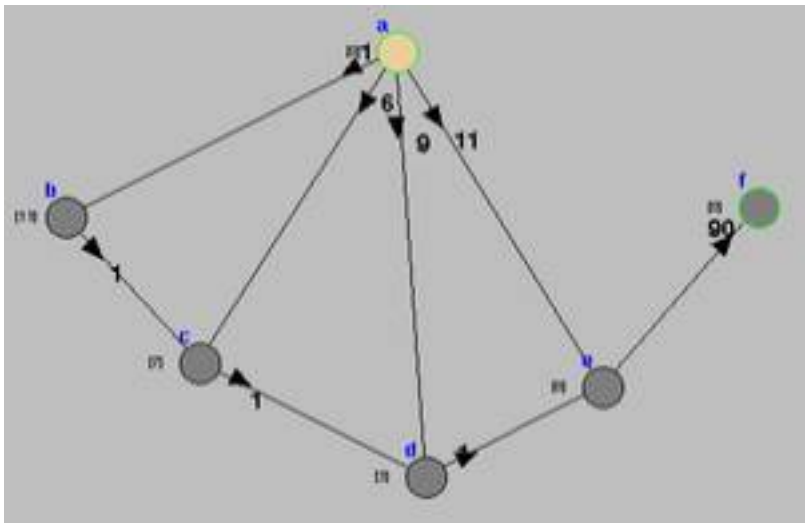
Comme on peut le constater sur cet exemple la distance finale est fausse. La raison pour laquelle on obtient ce résultat provient d'une heuristique attribuée au sommet b trop importante. L'heuristique a donc été mal choisie.

Cependant, on peut le voir diffèremment, en effet supposons deux routes menant au même endroit, on suppose l'une des deux routes plus courte que

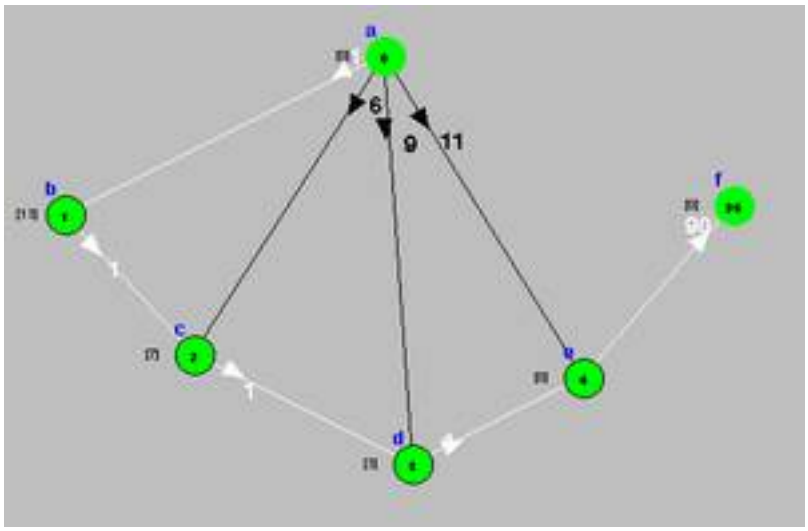
[2] On obtient l'algorithme de Dijkstra on mettant la même heuristique sur chacun des sommets du graphe.

l'autre. Si sur cette même route il y a eu un accident de voiture, la circulation sera très mauvaise, on peut supposer dans ce cas là que l'heuristique est très forte, ainsi l'ordinateur de bord ayant ces informations donnera l'autre chemin comme chemin le plus court.

Parcours très long et fastidieux



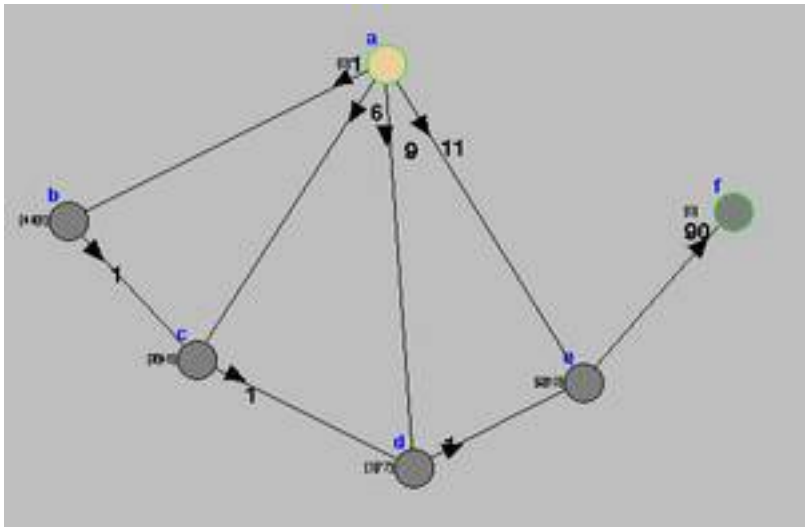
Début



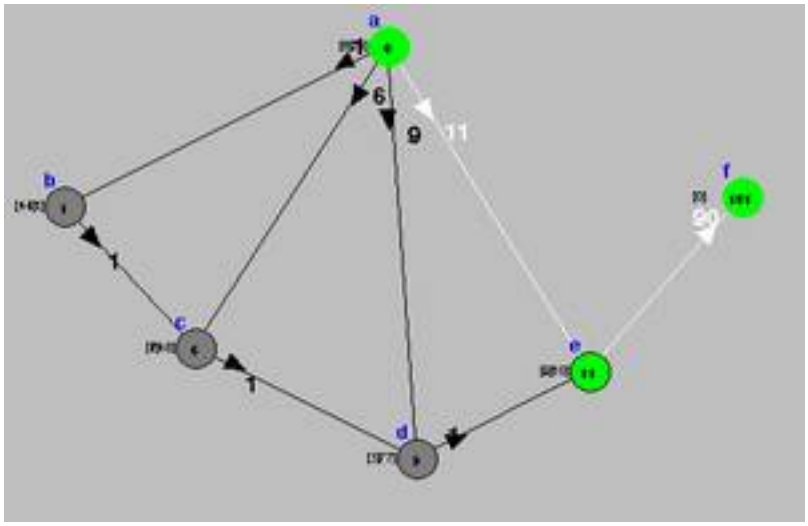
Fin

Dans ce cas de figure, le choix de l'heuristique est tel que les sommets traités, se réouvrent à l'étape suivante. De ce fait le parcours s'effectue en un temps bien supérieur comparé à l'algorithme de Dijkstra classique.

Distance exacte et immédiate



Début



Fin

Ce cas de figure représente un cas de figure optimal, effectué en un minimim de temps. Nous en reparlerons peu après lorsque nous présenterons l'algorithme A.

Point de vue programmation

D'un point de vue programmation la structute employée est la même que précédement.

Difficultés

Les difficultés rencontrées ont été similaires à celles de l'algorithme de Dijkstra. Les fonctions *Ajout* et *Virez* ont été réutilisées dans le même état d'esprit.

complexité
La complexité est très variable suivant les cas
- Si tous les sommets ont la même heuristique, la complexité équivaux à l'algorithme de Dijkstra
- Si le graphe réouvre les sommets déjà traités alors sa complexité est de l'ordre de l'exponentiel

Nous allons voir maintenant une spécificité de cet algorithme, qui lorsqu'on donne certaines propriétés à son heuristique, trouve toujours le chemin le plus court en un minimum d'étapes.

3.3.3 Algorithme A* à vol d'oiseau

Présentation
Cet algorithme repose sur les mêmes conditions que celui présenté auparavant. Son originalité se situe dans le choix de son heuristique. Dans ce projet, on prendra comme heuristique la distance de Manathan. C'est à dire pour chaque sommet du graphe, on affectera la distance à vol d'oiseau du sommet d'*Arrivée*.

Par cette méthode, on obtient toujours le plus court chemin entre deux points. La condition nécessaire est d'avoir la distance de Manathan plus petite que le chemin lui-même.

Caractéristiques
Par comparaison avec les autres algorithmes, il est à noter également que l'algorithme ne traite pas les sommets dont la distance de manathan est bien supérieure à la distance de manathan du sommet de départ, en supposant tout de même que les valuations des arrêtes restent cohérentes.
De ce fait, si l'on souhaite obtenir le plus court chemin entre Montpellier et Paris par exemple, on peut être sûr qu'il ne traitera pas les villes se trouvant géographiquement aux alentours du sud de Montpellier, alors qu'un algorithme de Dijkstra, calculera tous les chemins avant de vous montrer le chemin le plus court, ce qui est beaucoup plus coûteux.

3.3.4 Conclusion de cette partie

Suivant le cas de figure, le type d'algorithme employé sera très performant ou dans le cas contraire se déroulera lentement.
L'algorithme A* a d'autres utilités dans d'autres problèmes en utilisant une heuristique adéquate, par contre on a bien vu toute l'importance du choix de cette heuristique car dans le pire des cas on peut obtenir une complexité exponentielle ! !

3.4 Autres types d'algorithmes

Dans cette deuxième partie nous allons développer des algorithmes appartenant à une autre famille, celle du parcours en largeur.
Nous examinerons en second lieu, le parcours en largeur lexicographique.

3.4.1 Parcours en largeur

Présentation

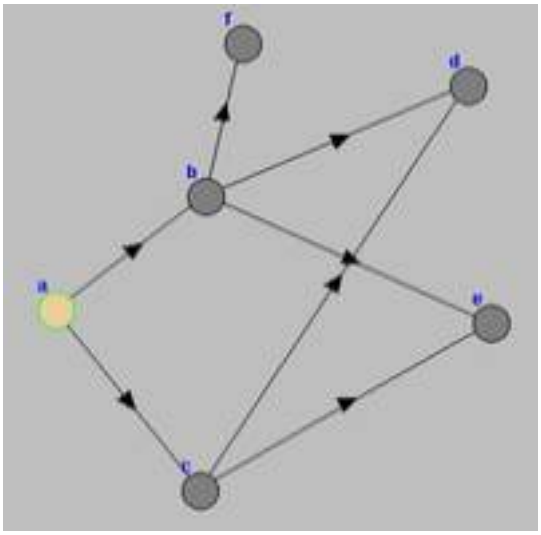
Le parcours en largeur se déduit du parcours générique. Il suffit pour cela de
- mettre un coût constant unitaire sur les arcs
- prendre T l'ensemble des entiers positifs
- gérer les *OUVERTS* comme une file *FIFO* (premier entré, premier sorti)

La compréhension de cet algorithme est simple : à chaque étape on traite tous les voisins du sommet choisi par la fonction *choix ()*, cela va constituer la première couche, on recommence l'étape suivante avec le premier voisin parcouru du précédent sommet, et jusqu'à ce que tous les sommets soient traités.

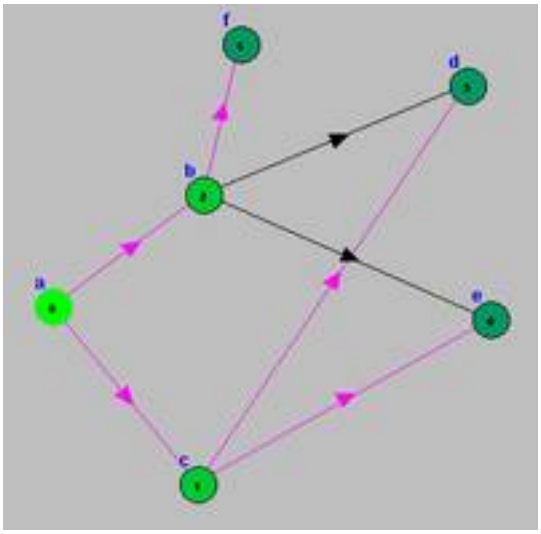
Au final, le parcours en largeur admettra une numérotation compatible qui sera visible sur le graphe ainsi que les différentes couches du graphe représentées par des couleurs différentes. Le choix reste le même cependant l'insertion des sommets dans la gestion des *OUVERTS* est totalement différente.
On prendra donc la nouvelle fonction *Ajout (z, OUVERTS)* on insère le sommet z à la fin de l'ensemble des *OUVERTS*
La fonction *Explorer* devient

Pour *tout voisin y de z* **faire**
  si $y \in auxFERMES$ **alors** *ne rien faire*
  si $y \in auxOUVERTS$ **alors** *ne rien faire*
  sinon
    $Ajout(y, OUVERTS)$
    $Parent(y) < -z$

Démonstration



Début



Fin

Point de vue programmation

Difficulté

Les règles étant encore plus simplifiées qu'auparavant, la gestion des *OUVERTS* s'est déroulée sans encombre.

Complexité

La fonction *Choix()* s'effectuant en $O(1)$, de même que la fonction *Ajout ()*, la complexité de l'algorithme s'effectue en $O(n)$ ce qui représente le parcours de tous ses sommets.

3.4.2 Parcours en largeur lexicographique

Présentation

Afin d'obtenir l'algorithme en largeur lexicographique, on opère sur l'algorithme de parcours générique comme suit

- on prend $T = 1, 2, 3, n*$ (l'ensemble des mots construits avec les nombres 1,2 jusqu'a n) et on interprète l'ordre sur ces mots lexicographiquement.
- La fonction *Explorer* devient alors

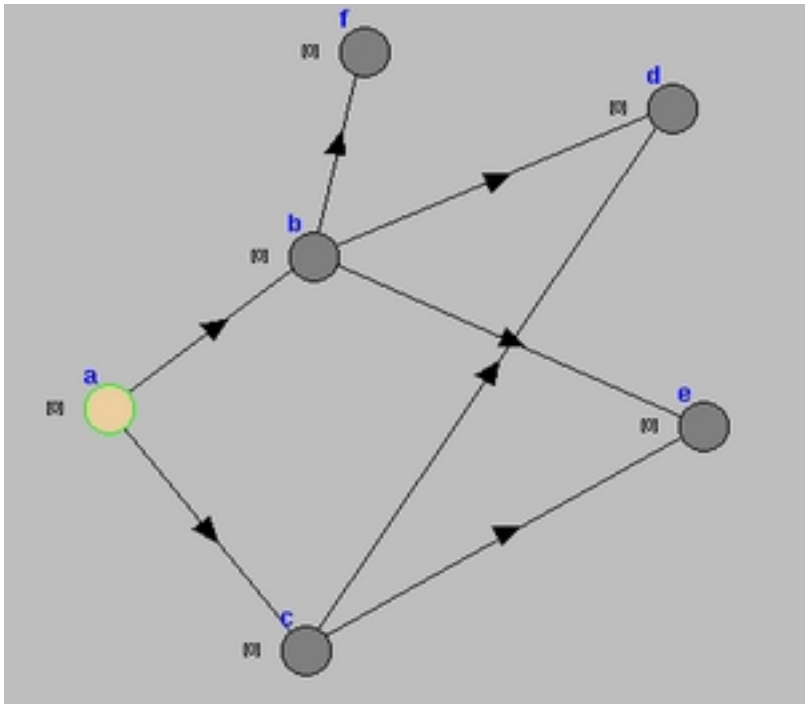
$numerocourant \leftarrow numerocourant - 1$
Pour *tout voisins y de z* **faire**
  si $y \epsilon FERMES$ **alors** *ne rien faire*
  si $y \epsilon OUVERTS$ **alors**
    $d(y) \leftarrow d(y) \bullet numero(z)$
  sinon
    $d(y) \leftarrow d(y) \bullet numero(z)$
    $Ajout(y, OUVERTS)$

- *numero (z)* la fonction renvoie la variable `numerocourant`

- *Choix ()* la fonction choix renvoie toujours le premier élément
- *Ajout ()* insère les sommets en fonction de leur distance lexicographiquement numerocourant est initialisé au départ à n (n représentant le nombre de sommets du graphe)

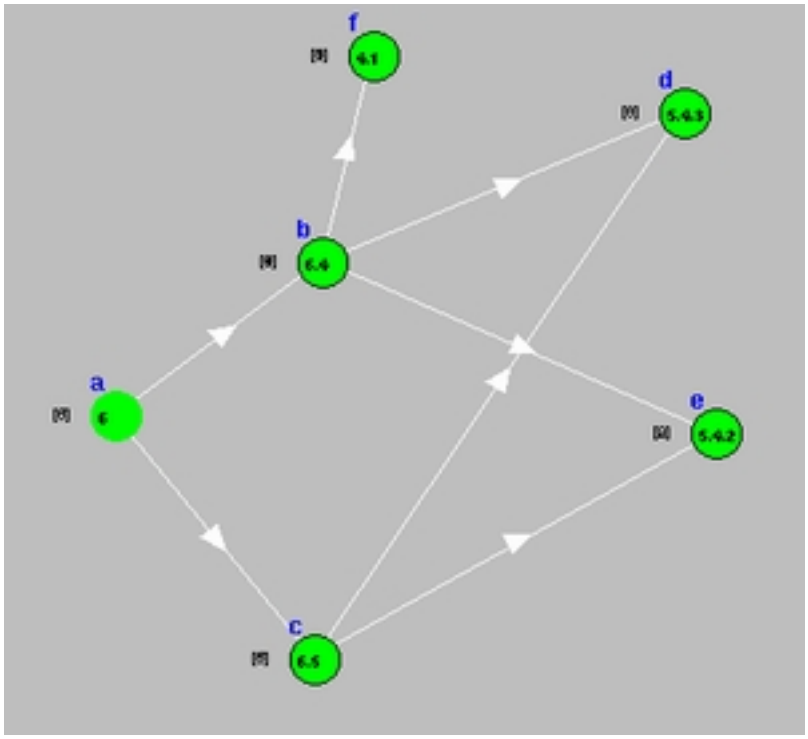
Cet algorithme doit parcourir tous les sommets avant de s'arrêter et il ne les parcours qu'une et une seule fois.
A la fin, nous obtenons une numérotation lexicographiquement compatible. A noter que le parcours en largeur lexicographique est un parcours en largeur classique recevant une relation de priorité quant au choix du sommet suivant. [3]

Démonstration



Début

[3] lorsque le graphe est complet, alors le dernier sommet traité obtient tous les numéros de sommet concaténés à lui. Ainsi un graphe complet à 5 sommets aura un sommet comportant la numérotation 5.4.3.2.1.



Fin

Point de vue programmation

Difficulté

La gestion des *OUVERTS* a était nettement plus délicate ici. La comparaison entre sommets ne se faisait pas automatiquement et la mise en oeuvre de cette fonction capable de déterminer lequel était le plus grand n'a pas été facile à réaliser.

Complexité

Le choix de la fonction s'effectue toujours en $O(1)$ mais la fonction *Virez ()* s'est effectués en $O(n)$, ainsi que la fonction *Ajout ()*, de ce fait l'algorithme se réalise en $O(n)$.
Une structure plus appropriée aurait certaiement diminué cette complexité mais par manque de temps nous n'avons pas pu le réaliser.

3.5 Choix des couleurs, enregistrements des étapes et commentaires

Cette dernière partie constitue le travail effectué sur chacun des algorithmes afin d'avoir le résultat attendu. Chaque algorithme est passé par ces trois étapes.

3.5.1 Choix des couleurs

Le choix des couleurs nous a révelé un problème plus fondamental, celui de décrire les étapes importantes de l'algorithme. Il fallait que certaines étapes fassent apparaître une spécificité de l'algorithme. Afin de faire comprendre en détail les subtilités des différents algorithmes, nous avons sélectionné 5 à 6 types de couleurs faisant apparaître

- les sommets appartenants aux *OUVERTS*
- le sommet choisi
- le sommet traité
- le sommet modifié
- les voisins du sommet sélectionné
- l'arrête modifiée
- l'arrête représentant le plus court chemin (potentiel ou réel)

Quand au choix des couleurs, elles ont été choisies avec un maximum de contraste afin d'obtrenir une meilleure visualisation du graphe.

3.5.2 Enregistrement des étapes

La deuxième partie correspond à l'enregistrement des différents sommets et arrêtes pour chaque étape. La visualisation d'une étape nous a posé un second problème, car certaines étapes nétaient pas nécessaire visuellement alors que certaines étaient primordiales.

Un exemple typique d'étape négligeable était qu'il ne fallait pas traiter les sommets les uns à la suite des autres lorsque ceux-ci n'appartenaient ni aux *OUVERTS*, ni aux *FERMES*, il fallait qu'en une seule étape on regroupe ces petites étapes.
On a essayé de mettre en valeur les étapes importantes en les décomposant si cela était nécessaire, comme lors d'une réouverture d'un sommet dans l'algorithme A*.

L'autre problème de l'enregistrement fût qu'il fallait enregistrer intelligement les sommets et arrêtes de chaque étapes afin de pouvoir visualiser avec la fonction *BACK* ou *NEXT* l'algorithme. La coordination des couleurs posa un problème.

3.5.3 Commentaires

La dernière et troisième partie constitue la partie compréhension. Le but des commentaires était d'afficher au bon moment les messages pour chacun des algorithmes et de représenter la partie de l'algorithme en cours de traitement. Il fallait de plus coordonner tout ceci avec les deux étapes précédentes afin que l'on puisse revenir à une certaine étape i de l'algorithme et avoir les renseignements propre à l'étape. La fenêtre des commentaires est divisée en deux parties

- la partie *Algorithme* indique la position de l'algorithme
- la partie *Commentaire* indique les caractéristiques de l'étape et de l'algorithme

Chapitre 4

Partie Graphique

4.1 Présentation et fonctionnalité des logiciels

L'objectif est de diviser les parties du logiciel pour pouvoir (un jour) utiliser une architecture client serveur.

Idéalement, il pourrait y avoir trois parties bien distinctes :
- L'éditeur de graphe, est une application seule qui permet de dessiner un graphe et de sauvegarder la description du graphe
- Un serveur d'algorithme prenant en donnée la description d'un graphe, et capable de renvoyer une description partielle du graphe à chaque étape de l'algorithme
- Un client animateur d'algorithme de parcours de graphe, qui (comme son nom l'indique) construirait une animation à partir des résultats du serveur d'algorithme

La programmation de l'éditeur et de l'animation a été entièrement faite à la maison. Plusieurs réunions ont été necessaires pour intégrer les algorithmes et la structure de graphe au format XML.

4.1.1 Editeur de graphe

Pour l'utilisateur, il fallait un programme permettant de rapidement dessiner un graphe (poser des sommets et relier ces sommets par des arcs), et de spécifier très précisement chaque détail du graphe :
- nom du sommet
- heuristique du sommet
- poids des arcs du sommet

De plus, l'utilisateur peut aussi modifier les proprietés du graphe, et enlever les sommets et les arcs existants. Enfin, l'utilisateur peut imprimer son graphe et le sauvegarder dans un fichier au format XML.

Le programme utilise la structure `DynamicGraphData` pour stocker le graphe que l'utilisateur construit. C'est cette même structure qui est directement utilisée pour afficher le graphe dans la `frame` - il n'y a donc pas de conversion faite

pour l'affichage, d'où un gain de temps à l'exécution.

On peut bien sûr modifier un graphe déjà sauvegardé.

4.1.2 Animation

L'animateur d'algorithme permet d'exécuter l'algorithme de son choix sur un graphe sauvegardé dans un fichier XML. Une fois que l'algorithme a tourné, on peut commencer l'animation propement dite. Les boutons `next` et `back` permettent respectivement d'aller à l'étape suivante, et l'étape précédente de l'algorithme, et l'utilisateur voit donc les changements de coloration du graphe dynamique pas à pas.

Il faut bien comprendre que l'algorithme n'est exécuté qu'une fois, et que l'animateur travaille uniquement sur le `DynamicGraphData` qui contient la trace de l'algorithme à chaque étape. Cette structure de graphe dynamique sauvegarde à chaque étape de l'algorithme les changements opérés, ce qui permet de faire tourner l'algorithme à l'envers visuellement. L'utilisateur peut ainsi sauvegarder le graphe dynamique (où l'exécution de l'algorithme a eu lieu) dans un fichier, et reprendre plus tard l'animation sans avoir à refaire tourner l'algorithme.

D'autres fonctionnalités pratiques pour l'utilisateur :
- Le `Scrollbar` *avancement* et le `Textfield` *avancementMin* permettent de se déplacer dans le graphe dynamique
- Le `Scrollbar` *vitesseS* définit pour la vidéo le temps entre chaque étape
- Le zoom (qui n'est pour l'instant pas pratique d'utilisation) permet d'animer juste une partie très précise du graphe dynamique

4.2 Affichage du graphe

La structure `DynamicGraphData` possède une fonction d'affichage. Cette fonction parcours entièrement le graphe pour afficher les sommets et les arcs. Pour un affichage net du graphe, il a fallu utiliser la trigonométrie et les nombres complexes qui facilitent l'affichage des fléches et la forte complexité entre deux sommets.

4.3 Programmation avec l'API Java

Nous avons utilisé l'API de base du JDK. Les objets AWT, très fonctionnels (bien que peu jolis), nous semblaient largement suffisante. De plus, l'application doit sûrement y gagner en rapidité d'exécution ; l'API SWING étant bien plus lourde que l'AWT.

La gestion des évènements est intuitive, on a eu aucun mal à relier les `Button, Scrollbar, Textfied, ...` avec les programmes. La documentation on-line est très pratique.

Cependant on peut reprocher à JAVA de ne pas supporter l'héritage multiple, qui aurait été utile pour la partie animation de ce projet.

4.4 Difficultés

On n'a pas rencontré de gros problèmes au niveau graphique, la gestion des évènements dans JAVA est assez intuitive, cependant la gestion des `Panel` et des `Layout` n'est pas très pratique (au moins au début du projet).

4.4.1 Problèmes liés à l'affichage

Le rafraichissement de la `frame` a été le principal problème rencontré. Lors d'un évènement (comme un clic de souris, pour poser un sommet), il faut rafraichir la fenêtre ; Denis PAYET nous a expliqué 2 solutions possibles

- Le clipping : qui consiste à rafraichir juste une partie de la frame (celle modifiée par le clic souris par exemple)
- Le double-buffering : cette technique utilise une image offscreen qui est recopiée à l'écran une fois qu'elle a été remplie

La solution la plus économique en temps machine est le clipping (puisqu'on ne rafraîchit qu'une petite partie de la `frame`, contrairement au double-buffering) ; c'est donc celle-ci qu'on a utilisé au début, mais on s'est vite rendu compte qu'il y avait une difficulté supplémentaire : **le calcul des coordonnées de la partie à rafraîchir**.

Finalement, on est passé progressivement à la seconde méthode. Et on a malheureusement remarqué un certain ralentissement de l'application sur nos ordinateurs "peu" puissants.

4.4.2 Compatibilité Java ?

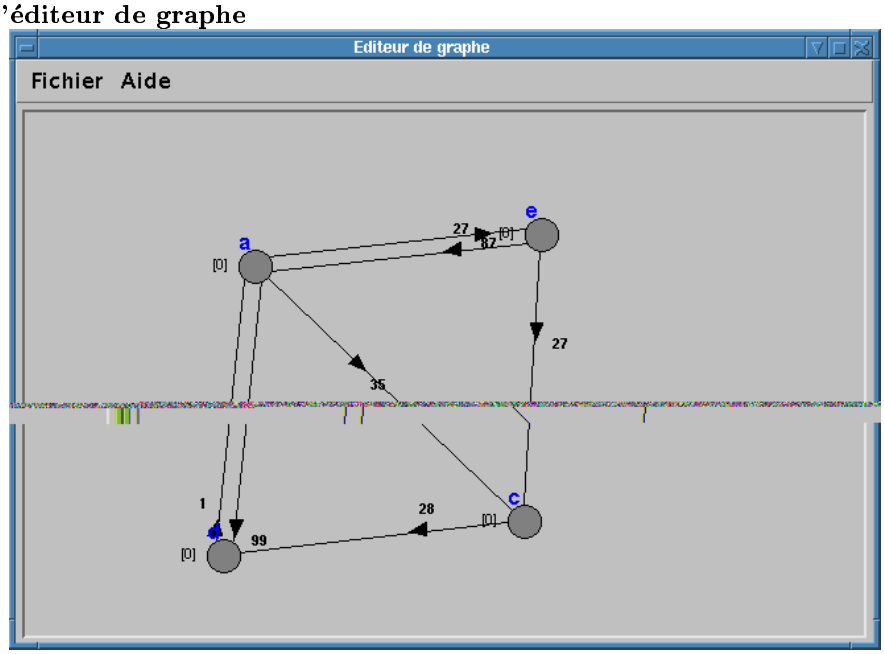
JAVA est supposé être un langage multi-plateforme et compatible ascendant. D'après notre petite expérience, on ne peut pas dire que ce soit le cas. Nous avons utilisé la version 1.1.8 du JDK pour développer le logiciel (suite au conseil de Jean-Francois BAGET). Nous avons constaté des lenteurs (et aussi des problèmes de font) sur les machines de la Fac (plus puissantes que les notres), qui utilisent le JDK 1.2.2. Et nous avons constaté, avec étonnement, stupeur et certes un petit moment d'émotion, une incompatiblité pure et simple avec la version 1.3.0 installée sur des machines au LIRMM.

4.5 Résultats

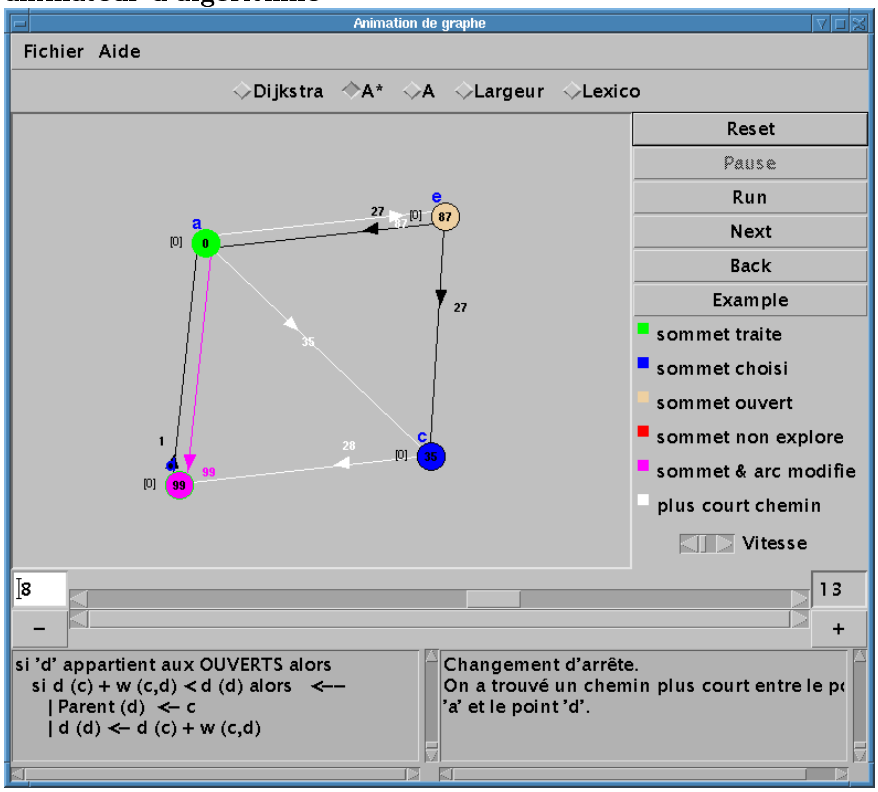
Notre logiciel fonctionne correctement (en tout cas sur le JDK 1.1.8), mais il n'est pas parfait :)

4.5.1 Maquettes d'écran

L'éditeur de graphe



L'animateur d'algorithme



4.5.2 Utilisation

Exécution Les sources et les bytes-codes sont accessibles sur la page `http://www.qalpit.com/~olivier/ter/index.html`.

Il faut ensuite rajouter dans l'environement le chemin exact pour l'API XML, le répertoire courant étant celui du projet :

- Sous UNIX avec *bash* : `export CLASSPATH=./xerces.jar:./`
- Sous UNIX avec *csh* : `setenv CLASSPATH ./xerces.jar:./`
- Sous WINDOWS : `set CLASSPATH .\xerces.jar;.\`

Maintenant on doit pouvoir lancer l'application avec `java anime`. Une petite compilation peut aussi être la bienvenue `javac *.java` si il y a des erreurs a l'exécution (dépendant de la version de votre JVM).

Description

On a essayé de faire une interface utilisateur simple ; une toute petite aide est disponible par le menu `a propos`. Les menus popup de l'éditeur et de l'animation sont crées dynamiquement en fonction de la localisation de la souris dans la `frame` - le menu popup sur un sommet du graphe permet de modifier les propriétés de ce sommet. Dans l'animation, des boutons de contrôle de l'animation (`reset, pause, run, next, back`) et du choix de l'algorithme sont disposś. L'interface du zoom n'est pas encore très pratique d'utilisation.

Chapitre 5

Conclusion

Chaque partie du rapport explique le travail effectué par chaque membre du trinôme de ce Travail d'Etude et de Recherche.

Ce TER nous a beaucoup appris, surtout pour la pratique. Il a montré certaines difficultés qu'il y a pour réaliser même un petit projet. Le travail en équipe est très motivant (surtout quand les membres s'entendent bien), permet de confonter l'avis de chacun pour faire émerger les meilleures idées.

La programmation JAVA n'a pas posé d'inconvéniants, nous regrettons cependant de ne pas avoir pu développer une application réseau (par manque de temps) ; cette idée nous semblait applicable pour le *Web*. Et il y a encore plusieurs bonnes ouvertures pour continuer ce projet, comme l'animation en VRML.

Au moment de la rédaction de ce rapport, il manque des fonctions dans la partie algorithmique, qui devraient être prête pour la soutenance.

Bibliographie

[1] *JDK 1.1.8 Documentation.* http ://java.sun.com/products/jdk/1.1/docs/index.html.

[2] *Xerces 1.3.1 API Documentation.* http ://xml.apache.org/apiDocs/index.html.

[3] David Flanagan. Java *in a nutshell.* O'Reilly, 1997.

[4] Michel Habib. Parcours générique dans un graphe. 2001.

[5] W3C, http ://www.w3.org/TR/REC-xml. *XML Reference Recommendation*, second edition edition, October 2000.

Résumé

Le TER du second semestre a pour objectif la mise en oeuvre des savoirs acquis jusque là : l'acquisition de compétences spécifiques, l'initiation à la recherche, l'analyse et la conception, ainsi que le travail en groupe.

Le développement des programmes JAVA d'un logiciel d'animation d'algorithmes de parcours de graphes nous a permis de mettre en pratique de nombreux enseignements (algorithmique, système, programmation, ...) dispensés, d'appréhender les atouts et les difficultés d'un tel langage.

On trouve dans ce rapport toutes les étapes d'analyse, de conception et de réalisation du projet. Les sources et les byte-codes des programmes, ainsi que nos adresses email sont disponibles sur la page web du TER
`http://www.qalpit.com/~olivier/ter/index.html`