

**Projet 2I006**

*Jean de Bodinat*

*Yannis Karmim*

*Rapport*

27/04/2017

|  |
| --- |
| SOMMAIRE |

I. SUJET DU PROJET 2

II. LES STRUCTURES MANIPULÉES ET STRUCTURE DU CODE. 3

I. Structure du Code. 3

1. 1ère Partie du Projet. 3

2. 2ème Partie du Projet. 4

II. Structures de données utilisées. 6

1. 1ère Partie du Projet. 6

2. 2ème Partie du Projet. 6

III. JEUX D’ESSAIS 7

I. Chaînes : 7

II. Réseau : 7

III. Nouvelles Chaînes : 8

IV. ANALYSE DES JEUX D’ESSAIS 8

1 – Analyse des performances des structures de données (1ère partie) 8

2 – Performance Parcours en Largeur. 9

3 – Performance Algorithme de Djisktra. 9

|  |
| --- |
| **I.** SUJET DU PROJET |

Dans le cadre de l’étude des structures de données ainsi que des algorithmes parcourant ces mêmes données, nous avons travaillé sur des réseaux allant de 12 et 15 liaisons à 9000 nœuds et 11000 liaisons.

Dans une première partie nous avons développé des graphes à partir de listes de chaînes en utilisant des tables de hachages, des arbres et puis donc des graphes.

Ensuite, nous avons développé des algorithmes calculant les plus courts chemins pour essayer de reconstituer ces chaines à partir de graphes.

|  |
| --- |
| II. LES STRUCTURES MANIPULÉES ET STRUCTURE DU CODE. |

## Structure du Code.

### 1. 1ère Partie du Projet.

Dans la première partie du code, nous avons décidé d’utiliser un fichier .c pour chaque structure donnée particulière utilisée, chacun des fichiers \*.c est accompagné d’un fichier \*.h référençant les fonctions utilisées et définissant les structures données utilisées.

* Arbre.c :

Les fonctions principales de ce fichier sont :

* ArbreQuat**\* insererNoeudArbre**(Noeud\* n, ArbreQuat\* a, ArbreQuat\* parent) ;

Comme son nom l’indique, cette fonction insere un nœud à sa place dans l’arbre. Sa position sera déterminée en fonction de ses coordonnées et de l’état des branches parentes (si elle existe, si elle comporte un nœud ou non).

* Noeud\* **chercherNoeudArbre**(CellPoint\* pt, Reseau\* R, ArbreQuat\*\* aptr, ArbreQuat\* parent) ;
* Reseau\* **recreeReseauArbre**(Chaines\* C) ;
* ArbreQuat\* **orientationArbre**( double x, double y, ArbreQuat\* a ) ;

Cette fonction factorise le code en retournant le sous arbre de a correspondant aux coordonnées x et y.

Malheureusement, la fonction recreeReseauArbre ne fonctionne pas. Un manque de rigueur en est surement à l’origine.

* Chaine.c :

Les fonctions principales de ce fichier sont :

* void **lectureChaine**( FILE \*f , Chaines\* chaines ) ;
* void **ecrireChaineTxt**(Chaines\* C,FILE \*f1) ;
* void **afficheChaineSVG**( Chaines \*C, char\* nomInstance ) ;
* Hachage.c :

Les fonctions principales de ce fichier sont :

* Noeud**\* rechercheCreeNoeudHachage**( Reseau \*R, TableHachage\* H, double x, double y) ;
* TableHachage\* **creerTableHachage**( Chaines \*C ) ;
* Reseau\* **recreeReseauHachage**(Chaines \*C) ;

On utilise ici la rapidité de recherche des tables de hachages pour créé un réseau à parti d’un table de hachage. Nous allons utilisé un boolean afin de savoir si nous visitons l’extremité A ou B de la liaison.

* Reseau.c :
* Noeud\* **rechercheCreeNoeudListe**(Reseau\* R,double x,double y) ;
* Reseau\* **reconstitueReseauListe**(Chaines\* C)
* void **min\_max\_r**(Reseau \*R,double\* minx,double\* miny,double\* maxx,double\* maxy)

### 2. 2ème Partie du Projet.

Dans la deuxième partie du code, nous avons décidé d’utiliser un fichier \*.c pour chaque méthode de recherche, ces fichiers sont aussi accompagné d’un fichier \*.h :

* Djikstra.c :

Djisktra.c contient le code nécessaire à l’algorithme de Djikstra.

Il a pour fonctions principales :

* … : Cette fonction fait…
* Largeur.c :

Largeur.c contient le code nécessaire au parcours en largeur du graphe.

Ses fonctions principales sont :

* int **nbarcsmin**(Graphe \*G, int u, int v) ;

On utilise un tableau dans lequel chaque case référence un point, on va ajouter 1 au nombre du prédecesseur de chaque nœud dans le parcours, afin que chaque tour soit comptabilisé.

* ListeEntier **cheminuv**( Graphe \*G, int u, int v ) ;

Nous utilisons le même algorithme que précédemment mais ici nous référençons dans chaque case du tableau (nœud) le numéro du père, ainsi, lorsque le point voulu est atteint, nous pouvons retrouver le chemin qui a été suivi pour atteindre le nœud.

* Liste\_chemin\* **chemin\_commodites**( Graphe\* G ) ;

Cette fonction va exécuter la précédente pour retourner dans une structure de données créée pour l’occasion, la liste des chemins entre chaque commodité.

* void **ecrire\_commodites**( Graphe\* G, char\* nomfic ) ;
* Arete\* **recherche\_arrete**( Graphe\* G, int u, int v ) ;

Afin de faciliter les fonctions calcul\_largeur et calcul\_longueur, cette fonction retourne l’arête d’extremité u et v appartenant au graphe.

* int **calcul\_largeur**( char\* nomfic, Graphe\* G ) ;

Nous allons lire les fichiers nouvellement créés par ecrire\_commodites, afin d’en extraire la largeur maximale du réseau en tout points. Pour ce faire, nous lisons les points un par un en associant chacun des points avec son prédécesseur, chacun de ces arêtes est recherchée dans le graphe et nous allons ajouter une occurrence à son paramètre gamma. Nous garderons le gamma maximal finalement trouvé.

* double **calcul\_longueur**( char\* nomfic, Graphe\* G ) ;

Cette fonction a le même fonctionnement que la précédente, à la différence que l’on ajouter à un total la distance euclidienne de chaque arête.

## II. Structures de données utilisées.

### 1. 1ère Partie du Projet.

* Arbre.c :

Nous utilisons ici simplement un arbre quaternaire contenant les nœuds du réseau.

* Chaine.c :

Nous utilisons trois structures, Un liste chaînée de points, un liste de chaînée de chaine contenant chacune un liste de points, et enfin l’ensemble des chaînes contenant les caractéristiques de cet ensemble ainsi qu’une liste chaînée de chaines.

* Hachage.c :

La seule structure utilisée sera une table de hachage, qui contient donc un tableau de pointeurs vers des listes chaînées de points, ainsi que la taille du tableau et le nombre points contenus.

### 2. 2ème Partie du Projet.

* Djikstra.c :
* Largeur.c :

Nous reprendrons les structures de données définies dans le Graphe.c. Afin de faciliter la mise en place des fichiers \*.ncha, nous avons développé une nouvelle structure. Cette structure est simplement une liste chaînée de liste chaînées de points. Elle permettra de faciliment accéder aux chaînes de points des commodités obtenus par le parcours en largeur du réseau.

Aussi, afin de pouvoir compter la largeur maximales du réseau, nous avons initialisé à zéro un entier cal\_gamma dans la structure arête.

|  |
| --- |
| III. JEUX D’ESSAIS |

## I. Chaînes :

La première version des jeux d’essai était sous la forme d’une liste de chaînes avec quelques caractéristiques précisées.

NbChain: 8

Gamma: 3

0 3 25.23 97.24 14.05 98.12 16.47 94.44

1 3 14.05 98.12 16.47 96.1 20.09 92.54

2 3 16.3 97.38 16.53 97.38 25.23 97.24

3 4 16.47 96.1 20.09 94.55 22.39 93.37 25.23 97.24

4 4 22.39 93.37 20.09 94.55 17.2 96.29 16.3 97.38

5 5 14.05 98.12 16.47 94.44 20.09 92.54 22.39 93.37 21.52 95.59

6 5 14.05 98.12 16.47 94.44 20.09 92.54 22.39 93.37 22 96.05

7 3 22.39 93.37 20.09 92.54 16.47 96.1

Comme pour chaque jeu de fichiers tests, nous avons des tailles variables de fichiers qui nous permettent de tester et d’analyser les performances de nos codes.

## Réseau :

NbNoeuds: 12

NbLiaison: 15

NbCommodite: 8

Gamma: 3

v 3 22.390000 93.370000

v 2 20.090000 92.540000

v 1 16.470000 96.100000

l 2 3

l 1 2

1 3

## Nouvelles Chaînes :

1 2 3 4 5 -1

6 7 -1

3 2 8 -1

3 4 9 10 30 13 14 15 16 -1

Ces fichiers sont créé par la fonction écrire\_comodités, c’est un liste des points de chaque chaînes.

Ils seront utilisés pour tester l’efficacité de

|  |
| --- |
| IV. ANALYSE DES JEUX D’ESSAIS |

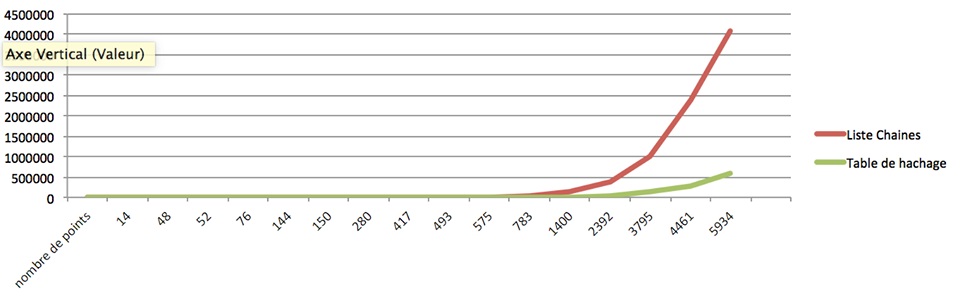
## 1 – Analyse des performances des structures de données (1ère partie)

Ci –dessous les performances de reconstitutions des réseaux à l’aide de différentes structures.

Les performances des arbres quaternaires n’ont pas pu être réalisée sachant que la fonction buggait. Mais il est évident que les arbres quaternaires auraient été les plus performants.

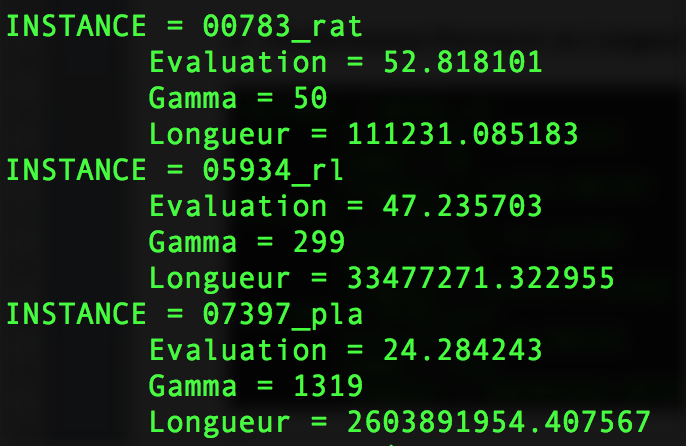
On peut voir sur l’axe des ordonnées des 1. 10(-6) secondes.

Nous reconstituons ici des réseaux à l’aide de la structure de données numéro 1, les chaînes.



## 2 – Performance Parcours en Largeur.

Ci-dessous un Screenshot du terminal sur les performances du parcours en largeur pour trouver les plus courts chemins.



Il est intéressant de voir que les performances du parcours en largeur suivent un fonction affine en fonction de gamma\*nbliaisons, gamma étant le gamma trouvé par l’algorithme et nbliaisons le nombre de liaisons du réseau.

## 3 – Performance Algorithme de Djisktra.