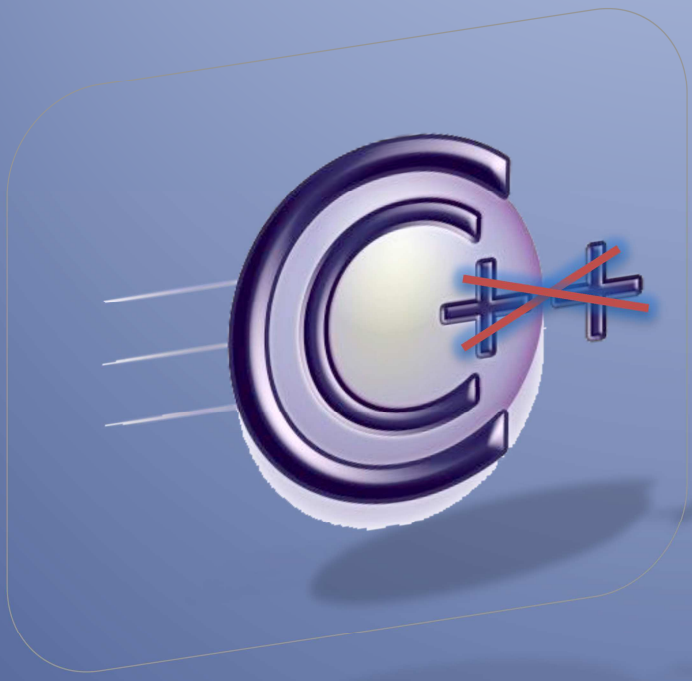


PROJET LI213

COQUART KEVIN | RYBAEV DANIIL

Ceci est notre compte-rendu du projet de li213. Le rendu contient 7 pages.

COQUART KEVIN 3000976
RYBAEV DANIIL 3000086

PROJECT - LI213 (PHASE 1)

CONTENU

PREAMBULE

Nous avons réussi à lire toutes les instances avec chacune des méthodes ("normale", table de hachage et arbre binaire).

Dans le dossier instance, les fichiers en .res seront créer avec la méthode de base, ceux _hachage.res avec la table de hachage et les _abr.res avec l'arbre binaire équilibré.

La longueur des chaînes a été évalué avec et sans le gamma, voir le tableau à la fin de ce rendu.

INTRODUCTION

Ce projet consiste à manipuler diverse structure de données ainsi que les comparer.

Le sujet comporte plusieurs parties :

- la 1er grande partie était de convertir le fichier lut dans un format plus facile à manipuler. Puis de construire un plan de l'instance.
- la 2ème étape était de reconstruire le réseau en utilisant 2 autres structures de données (table de hachage et arbre binaire) puis d'évaluer la rapide de chacune pour ensuite les comparer.
- la 3ème étape consistait à chercher si l'on pouvait réduire les tailles des instances en prenant des chemins plus cours.

LES STRUCTURES DE DONNEES MANIPULE

Les différentes structures de données utilisées dans ce projet sont :

- des listes simplement chaîné
- une table de hachage
- un arbre binaire de recherche équilibré
- un tas

LES ALGORITHMES DEPLOYES

Au niveau algorithmique, il y a de nombreux parcours de liste, tableau ou bien d'arbre de recherche, chose plutôt simple ainsi que quelque fonction d'implémentation sur les tas.

Mais on a aussi utilisé l'algorithme de dijkstra, ce dernier sert à rechercher la plus courte distance entre 2 points.
On définit des poids à chaque points de notre liste, au départ on les initialise tous à l'infini (en l'occurrence, le c ne proposant pas l'infini, on a choisi -1) et on met le point d'arrivés à 0. Ensuite on parcourt les points relié au précédant en ajustant leur poids (dans le projet, c'est la distance entre les 2 pts) jusqu'à être au moment où l'on regarde le pt de départ et l'algorithme se termine.

CONTENU DU CODE

- Abr.h : contient la structure de donné utilisé pour les arbres, ainsi que les fonctions de création d'arbres à partir d'un réseau et d'affichage d'un arbre ->ex5
- Chaine.h : contient les fonctions pour réduire la taille des listes (basé sur dijkstra) -> ex8
- Dijkstra.h : contient l'algorithme de dijkstra -> ex7
- Hachage.h : contient la structure de donné de notre table de hachage, les fonctions de créations et d'affichage d'une table, ainsi que la clé de hachage -> ex4
- Liste.h : contient la structure de donné de notre liste chaîné, les fonctions de créations et d'affichage -> ex1
- Noeud.h : contient les fonctions de comparaison sur les nœuds, la création et l'ajout de voisin et enfin les fonctions de créations d'un réseau -> ex3
- Reseau.h : contient la structure de donné de notre réseau, les fonctions de créations, d'affichage, d'écriture sur le disque -> ex2
- Tas.h : contient la structure du tas ainsi que les éléments qui le remplissent. Il y a aussi les fonctions qui vont avec.

MES EXECUTABLES

Mon Makefile créés 5 exécutables, un seul est à utiliser directement :

- menu : le lanceur « graphique », il demande l'instance à traité lors de son exécution
- main : exécutable personnelle, qui sert lors d'évolution du code
- creerR : prend en paramètre l'instance .cha, l'emplacement de sauvegarde .res ainsi que si l'on veut celui du dessin .fig. Fait la conversion de .cha a .res. -> ***EX3***
- creerH : prend en paramètre l'instance .cha, l'emplacement de sauvegarde .res ainsi que si l'on veut celui du dessin .fig. Fait la conversion de .cha à .res en se servant d'une table de hachage comme support. -> ***EX4***
- creerA : prend en paramètre l'instance .cha, l'emplacement de sauvegarde .res ainsi que si l'on veut celui du dessin .fig. Fait la conversion de .cha à .res en se servant d'un arbre binaire de recherche comme support. -> ***EX5***
- creerC : à ne pas utiliser de manière directe. Sert lors de l'exécution du script Shell pour la création du PostScript avec les courbes. -> ***RENDU_S8***
- creerL : prend en paramètre l'instance .cha, l'emplacement de sauvegarde du dessin .fig. Créer le dessin de l'instance (en passant directement par la liste). -> ***EX1***
- dijk : prend en paramètre l'instance .cha, l'emplacement de sauvegarde du dessin .fig ainsi que si l'on veut celui avec la représentation de dijkstra. Calcule dijkstra entre le nœud 0 et le nœud de fin, cela est paramètre dans le main en changeant la valeur des 2 premiers paramètres de la fonction dijkstra. Renvoie sur le terminal la distance entre ces 2 nœuds et si l'on demande les figures, affiche sur la seconde le chemin idéal en rouge. -> ***EX7***
- dijkListe : prend en paramètre une instance .cha. Calcule la longueur de la chaîne initiale, applique dijkstra dessus (avec et sans gamma) et renvoie la nouvelle longueur (sans chercher à sauvegarder la nouvelle liste obtenue pour ne pas interférer avec des tests ultérieurs). -> ***EX8***

MODE D'EMPLOI

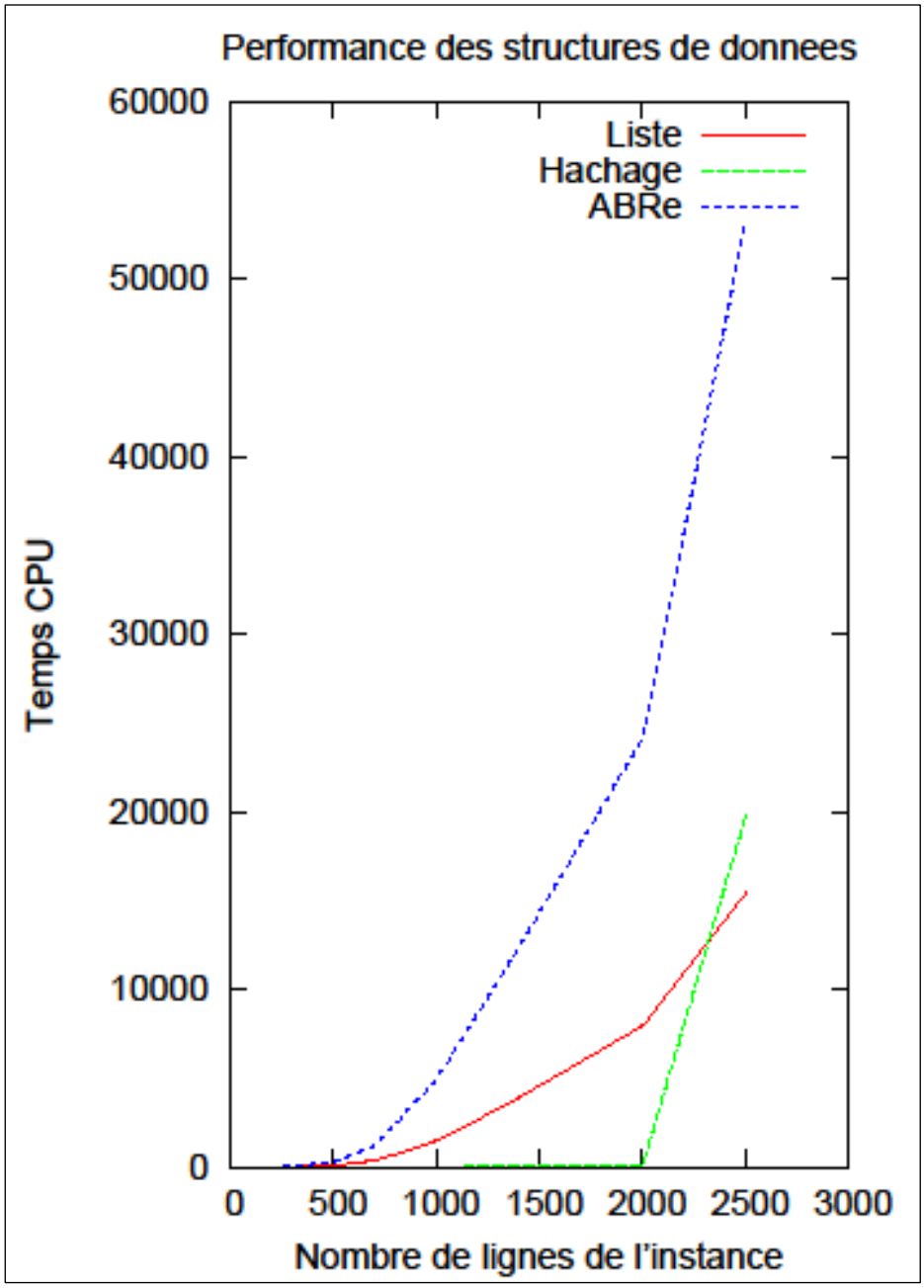
Pour exécuter les différents programmes, on peut au choix :

- Se servir du menu
- Lancer un des exécutables (creerL/R/H/A – dijk - dijkListe).
- Lancer le script creation_courbes.sh avec au choix
  - Un fichier .cha en paramètre et l'endroit ou écrire les tps CPU, il créer alors le rendu[1] et écrit le temps mis dans le fichier.
  - Un dossier contenant des .cha, il créer alors le rendu pour chaque instance et écrit le tps CPU de chaque instance, ceci sert pour créer les courbes.

[1] Correspond aux fichiers suivant : .res, _hachage.res, _abr.res, _liste.fig, .fig.

EXPERIMENTAL

Voici les courbes obtenu avec un test portant sur le temps CPU qu'il a fallu pour récréer chaque instance en fonction de la structure de données utilisé.



L'on remarque assez clairement que l'ABRe n'est pas adaptés à cette tâche.

La table de hachage est performante bien que pour l'instance de fin, elle soit plus lente que la méthode de base. Il faudrait d'autre instance plus grande et essayer avec différentes tailles de table de hachage pour voir si ce problème persiste.

L'on peut pour le moment conclure sur le fait que la structure de donnés la mieux adaptés et la table de hachage.

LA COMPLEXITE

Ω: meilleur des cas Θ: cas constant O: pire des cas

Soit n, le nombre de points dans le fichier initial. Le parcourt de ce fichier sera donc en Θ(n), on ces points dans le réseau.

Soit n2, le nombre de points mis progressivement dans le réseau, n2 égale n moins les doublons et n2 <= n (en général largement inférieur à n).

RESEAU DE BASE

Avec la fonction recherche_noeud, nous allons reparcourir le réseau de taille croissante n2, n fois.

La fonction ajouter_voisin va reparcourir deux fois la liste des voisins des deux nœuds crées pour mettre à jouer leurs voisins.
Or, un nœud à bien souvent entre 2 et 5 voisins (on prendra 3 en moyenne pour les calculs).

Donc au final nous avons Θ(n)*(O (n2) +O (3))

Considérons que n est égale ou proche de n2 pour le pire des cas, nous obtenons : C(x)=O ($3n^2$) environ.
Donc une complexité en O (n^2).

HACHAGE

Avec la fonction recherche_cree_noeud_hachage nous calculons l'indice où l'élément va se placer dans le tableau et on fait quelque petit test.
Suivant la taille du tableau choisi mais généralement grand, on a très peu de case remplie et souvent avec 1 seul éléments. Donc l'insertion se fera en O(1) environ.

La fonction ajouter_voisin va reparcourir deux fois la liste des voisins des deux nœuds crées pour mettre à jouer leurs voisins.
Or, un nœud à bien souvent entre 2 et 5 voisins (on prendra 3 en moyenne pour les calculs).

L'insertion dans le réseau se fait toujours en O(1) car nous l'insérons toujours au début.

Nous arrivons donc à C(x)=O(n).

ARBRE

La fonction insere_noeud va parcourir l'arbre, créer le nœud et l'insérer à la bonne position s'il n'existe pas, et renvoyer un pointeur dessus.

Par conséquent, étant donné que nous parcourons l'arbre, la complexité de ce parcours est en O (log2(n)).
Idem pour la recherche de nœud dans un arbre.

Le rééquilibrage de l'arbre et la mise à jour des hauteurs ne sont que des instructions, donc on les considère en O(1)

La fonction ajouter_voisin va reparcourir deux fois la liste des voisins des deux nœuds crées pour mettre à jouer leurs voisins.
Or, un nœud à bien souvent entre 2 et 5 voisins (on prendra 3 en moyenne pour les calculs).

L'insertion dans le réseau se fait toujours en O(1) car nous l'insérons toujours au début.

Nous arrivons donc à C(x)=O (n*(2log(n))) = O (n*(log(n)).

CONCLUSION

D'après les complexités, les tables de hachage sont bien mieux adaptés pour la création des réseaux, l'insertion et la recherche étant quasiment en O(1), cela permet d'ajouter des nouveaux nœuds ainsi que de trouver les doublons très rapidement.

LES LONGUEURS DES CHAINES

Longueur des chaînes				
Instance	**Longueur initiale**	**Longueur sans gamma**	**Longueur gagné en %**	**Longueur avec gamma**
00014_burma.cha	105	ERREUR*		ERREUR*
00022_ulysses.cha	337	124	63,2	124
00048_att.cha	122965	74175	39,7	74175
00052_berlin.cha	45553	18591	59,2	18591
00076_eil.cha	2141	1125	47,5	1125
00101_eil.cha	2818	1470	47,8	1470
00144_pr.cha	519326	296660	42,9	296661
00150_ch.cha	51544	23095	55,2	23095
00280_a.cha	23389	10839	53,7	10839
00417_fl.cha	355532	176467	50,4	176467
00493_d.cha	607003	270628	55,4	270628
00575_rat.cha	102327	53818	47,4	53818
00666_fr.cha	55127	28661	48,0	28661
00783_rat.cha	179150	85762	52,1	85762
01400_fl.cha	1768183	807189	54,3	807189
02392_pr.cha	13817889	6571161	52,4	6571162
03795_fl.cha	3785234	1232460	67,4	1232460
04461_fnl.cha	8167113	4742015	41,9	4742015
05934_rl.cha	48710747	20572554	57,8	20572554
07397_pla.cha	3309558964	1841825288	44,3	1841825288

L'on remarque assez clairement que les chaînes peuvent être réduite assez significativement, l'ajout du gamma ne change à priori rien (ce qui me pousse à me demander si ma fonction marche réellement, mais je ne vois pas ou serait l'erreur).

CONCLUSION

Ce projet nous a permis de nous perfectionner dans un langage de programmation quasiment indispensable pour la suite de nos études. Nous avons découvert et implémenter des structures de données qu'à ce jour nous ne connaissions peu.

Ce projet nous a montré un cas bien réels de l'utilité de l'informatique dans la vie « courante », car bien que l'on sache (et heureusement puisqu'on a choisi cette voie) que l'informatique est utile et indispensable, nous ne le voyons guère à notre niveau (je ne pense pas que programmer un morpion en VBA soit d'une utilité indispensable à la société [je ne remets pas en cause l'aspect pédagogique]).

Puis nous pouvons rajouter que ce projet a favorisé l'esprit collaboratifs des membres de ce binôme et aussi avec les autres personnes de l'UE, échanger autour de quelque point qui des fois nous bloqué nous, des fois les bloqués eux.