Etienne FONGUE
Thanh Lam TANG

Tuteur : Arnaud Carayol

Solveur de Ruzzle

Introduction

Le projet de Ruzzle s'inscrit dans l'option de projet tutoré de L3 informatique, proposée par notre tuteur Arnaud Carayol.
Ce projet a été développé en java et le but était de créer un solveur pour Ruzzle.
Nous commencerons par présenter le jeu en lui même, ainsi que ses subtilités, ensuite nous aborderons les différentes étapes de développement qui nous ont amenés à l'élaboration du projet. Nous continuerons par une comparaison entre les différentes démarches entreprises, et nous finirons par quelques mots sur l'interface graphique déployer.

1 - Présentation du jeu Ruzzle

Le jeu Ruzzle est un jeu composé d'une grille 4x4, chaque case contient une lettre, et le but est de trouver le plus de mots avec les 16 lettres disponibles et donc de réaliser le meilleur score possible en 2 minutes.

1.1 - Les Règles

Il y a cependant certaines règles à respecter:

- Les mots doivent être dans le dictionnaire.
- Les mots doivent être de 2 lettres à 16 lettres.
- On ne doit utiliser que les lettres adjacentes pour se déplacer.
- On peut se déplacer dans toutes les directions même en diagonales.
- On ne peut pas utiliser deux fois la même case.
- On ne peut pas faire 2 fois le même mot, même si l'on prend un autre chemin

1.2 - Les points

1.2.1 - Les points sur les lettres

Chaque lettre possède une valeur en points.
Par exemple:

A = 1 point.
C = 3 points.
Z = 10 points.

Le mot "CA" vaut donc 4 points.

1.2.2 - Les bonus sur les lettres

Chaque case de la grille peut contenir un bonus.
Il y a 4 types de bonus :

DL = Lettre compte double => les points sur cette lettre sont doublés.
TL = Lettre compte triple => les points sur cette lettre sont triplés.
DW = Mot compte double => les points du mot seront doublés.
TW = Mot compte triplé => les points du mot seront triplés.

1.2.3 - Les bonus sur la longueur des mots

Plus un mot est long plus il rapporte des points.

Longueur de 5 = +5 points
Longueur de 6 = +10 points
Longueur de 7 = +15 points
Etc.

1.2.4 - Le système de calcul des points

Les points sont calculés dans cet ordre:

On fait la somme des points de chaque lettre du mot, en prenant en compte les bonus effectifs sur les lettres, DL ou TL.

Ensuite, on multiplie ce nombre par 2 ou 3 selon si on a pris le bonus DW ou TW, sachant que les bonus se cumulent multiplicativement, par exemple si l'on prend le bonus DW ET TW, les points seront multiplier par 6!
Ensuite à ce résultat on ajoute le bonus sur la longueur.

Maintenant que les règles du jeu sont posées, passons au projet!

2 - Présentation des problèmes

Le but du projet est donc de réaliser un programme donnant l'ensemble des mots possibles sur une grille du jeu Ruzzle. On souhaite donc obtenir la liste des mots que l'on peut trouver, a l'aide d'un dictionnaire.

Remarque: On utilise actuellement le dictionnaire du jeu Scrabble, qui fût disponible; les dictionnaires n'étant pas parfaitement identiques, il n'est pas impossible que la fonction de recherche trouve des mots qui ne sont pas validés par le jeu Ruzzle.

Cette tâche n'est pas aussi facile que cela pourrait paraitre, en effet il y a énormément de combinaison possible, et les règles de déplacement compliquent la tâche.

On a donc cherché un moyen pour trouver tous les mots de la grille tout en obtenant un temps de recherche faible.

La problématique se décompose donc en 2 parties bien distinctes, qui sont la recherche de tous les mots dans une grille et le temps/complexité/poids de cette recherche.

À partir de là, on s'est posé la question: quelle structure de donnée utiliser?

Nous avons tenté de résoudre le problème en utilisant différentes structures de donnée:
listes, HashMap, HashSet, arbre, arbre hybride (arbre + liste).

Chacune ayant ses avantages (implémentation facile, rapide, algorithme simple) et ses inconvénients (recherche lente, structure de donnée trop volumineuse ...), nous avons donc implémenté ces différentes structures de données, le but ultime était de comparer ces implémentations afin de retenir l'implémentation qui s'accordait le mieux avec ce type de problème.

2.1 - Le 1er problème : La recherche du plus long mot

Le problème qui nous a été demandé de résoudre au tout début a été de rechercher l'ensemble de tous les mots (et le(s) plus long(s) mot(s)) que l'on pouvait faire à partir d'un ensemble de 16 lettres tiré au hasard, pour cela nous avons testé 2 implémentations présentées ci-dessous a savoir celle d'une ArrayList et celle d'un HashMap.

Ensuite un problème plus délicat nous a été posé que nous aborderons dans la seconde partie.

2.1.1 - L'implémentation en Arraylist

Pour répondre à ce problème, nous avons implémenté le dictionnaire (fichier texte) sous forme d'une liste de mot (ArrayList<String>).
Nous avions donc une liste de mots naturellement triés par ordre alphabétique et ensuite nous avons trié les lettres de chaque mot par ordre alphabétique.

Cela donne par exemple:

liste de mots		**liste de mots triés par lettre**
bonjour=>	bjnooru	
chien	=>	cehin
etc.		

Pour la recherche il nous suffisait de créer un mot aléatoire trié:
Si le mot aléatoire est "cbeihzn" nous obtenons "bcehinz".

Ensuite, nous pouvions facilement comparer "bcehinz" avec les mots de liste des mots triés par lettre.

L'implémentation de la recherche est effectuée par la méthode statique DicoList.wordCreatFromRand présente dans le fichier DicoList.java de Ruzzle_3.

Ainsi nous pouvions trouver les mots français correspondants, cependant cette méthode n'est pas optimal à cause du fait que les mots : "chien", "niche", "chine" se transforme tous en "cehin", (ce sont des anagrammes).
Nous avions donc plusieurs occurrences de "cehin" dans la liste.
Cette structure de donnée (ArrayList) ne nous permettait pas de résoudre sans redondance la recherche des mots, ce qui nous a amennés à l'implémentation d'une HashMap.

2.1.2 - L'implémentation en HashMap

Pour éviter ces problèmes d'anagramme, nous avons implémenté une HashMap qui à chaque mot, trié par lettres, associe une liste de mots.

Ce qui donne par exemple :

Clé **Liste associée**

cehin => [chien,chine,niche] // la clé cehin associe les mots chien, chine et niche.

etc.

La recherche de mot s'effectue de la même façon que dans l'implémentation ArrayList, mais cette fois en utilisant les clés.

L'implémentation de la recherche est effectuée par la méthode findKey présente dans le fichier DicoHashMap.java de Ruzzle_HashMap.

2.1.3 - Benchmark sur les structures ArrayList et HashMap

Voici le tableau des temps de la mise en place de la structure et de la recherche de tous les mots à partir d'un ensemble de lettres aléatoire.

Test 1	Test 2	Test 3	Test 4	**Moyenne**
1,1192976467	1,1199475848	1,1180406319	1,1202900179	1,11939397

Test 1	Test 2	Test 3	Test 4	**Moyenne**
2,2277768042	2,2309214245	2,2276403446	2,2234373205	2,22744397

On remarque que le temps moyen pour la structure de donnée ArrayList est de 1,12 seconde tandis qu'elle est de 2,23 seconds pour la HashMap.

Cela s'explique par le fait que la HashMap prend plus de temps pour être fonctionnelle, en effet on doit parcourir plusieurs fois la liste des mots pour trouver toutes les anagrammes, ce temps est compensé par le fait que les recherches sont plus rapides.

2.2 - Le 2nd problème : La recherche de tous les mots de la grille

Pour le second on dispose d'une matrice correspondant à la grille de lettres du jeu Ruzzle en plus du dictionnaire.

Le problème à résoudre fut plus ambitieux, car il consistait à trouver l'ensemble de tous les mots français faisable à partir de la grille, tout en appliquant les règles de déplacement du jeu.

Or pour effectuer cette recherche, nous devions apporter une attention particulière au temps de recherche et de mise en place de la structure de données.

Nous allons commencer par expliquer les structures de données qui ont été utilisées puis présenter un benchmark comparatif entre ces structures.

La recherche dans la grille et les optimisations pour rendre cette recherche plus rapide seront expliquées dans la partie suivante.

2.2.1 - La recherche dans la matrice et optimisation

Pour rechercher un mot dans la grille, nous devons passer par plusieurs étapes, voici l'enchainement des étapes de la recherche:

On commence par parcourir chaque lettre de la matrice.
Au départ, la chaîne contient uniquement la lettre de la case.

Puis par un parcours préfixe l'arbre du dictionnaire, on teste si la chaîne est un préfixe d'un des mots du dictionnaire, si ce n'est pas le cas, on "return".
Puis on teste si la chaîne est un mot du dictionnaire, si c'est vrai on l'affiche.

Ensuite, pour chaque voisin de la dernière lettre de la chaîne, on ajoute la lettre voisine à la fin de la chaîne et on effectue les mêmes tests jusqu'à ne plus trouver de voisins déjà parcourus , ainsi, on procède à la recherche de tous les mots pour toutes les cases de la matrice.

Cependant la recherche en procédant par toutes les combinaisons possibles, possède une complexité élévée (difficilement évaluable) qui ne serait pas résolue en un temps raisonnable, pour remédier a cela il a fallût effectuer des optimisations lors de la recherche.

Notamment lorsqu'on a un mot qui n'est le préfixe d'aucun autre mot du dictionnaire :

```
if (!(ArbreH.estPrefixe(sb.toString(),aH))){ return;}
```

Ce test de condition nous fait gagner énormément de temps dans la recherche et cette optimisation est utilisée pour toutes les implémentations qui s'en suivent.

2.2.2 - L'implémentation en Arbre Hybride

Nous allons commencer par expliquer la structure la plus complexe à savoir l'arbre hybride, car les autres implémentations qui s'ensuivent seront relatives à celle-ci.

Qu'est ce que l'arbre hybride?

L'arbre hybride c'est une structure mi-arbre, mi-liste, qui se comporte comme un arbre, puis à partir d'une certaine hauteur, tous les fils sont regroupés dans une liste.

Le but de cette structure est donc de faire varier la hauteur de l'arbre afin d'en tirer des conclusions sur les performances, et les hauteurs optimales, pour résoudre le problème de recherche.

Voici comment se présente la structure d'un arbre hybride :

```
private ArrayList<ArbreH> arrayFils;          // Liste d'arbre fils
private ArrayList<String> arrayChaine;// Liste des mots
                                              (peut être une chaine ou une lettre selon la hauteur)
private boolean estFeuille;                   // Indique si ce noeud est une feuille
```

Chaque noeud en partant de la racine jusqu'à la hauteur choisie contient 1 lettre, au-delà, les mots de longueur supérieure à cette hauteur sont découpés et positionné dans une liste.

Voici ce que cela donne avec une hauteur de 4 :
(Les feuilles sont entre accolades)

```
<[.]>
  |  <[A]>
  |    |  {[A]}
  |    |    |  <[L]>
  |    |    |    |  <[E]>
  |    |    |    |    |  {[NIEN, NIENNE, NIENNES, NIENS]}
  |    |    |  {[S]}
  |    |  <[B]>
  |    |    |  <[A]>
  |    |    |    |  <[C]>
  |    |    |    |    |  {[A, AS, OST, OSTS, ULE, ULES]}
Etc.
```

(En modifiant la hauteur, on peut passer d'un arbre "entier" à une liste chainée et vice versa)

2.2.3 - L'implémentation en Arbre (Arbre Hybride de hauteur 15)

En imposant une hauteur d'arbre de 15, nous obtenons un arbre uniquement composé de noeud contenant une lettre, un arbre complet en somme.

Cela donne :

```
<[.]>
  |  <[A]>
  |    |  {[A]}
  |    |    |  <[L]>
  |    |    |    |  <[E]>
  |    |    |    |    |  <[N]>
  |    |    |    |    |    |  <[I]>
  |    |    |    |    |    |    |  <[E]>
  |    |    |    |    |    |    |    |  {[N]}
  |    |    |    |    |    |    |    |    |  <[N]>
  |    |    |    |    |    |    |    |    |    |  {[E]}
  |    |    |    |    |    |    |    |    |    |    |  {[S]}
  |    |    |    |    |    |    |    |    |  {[S]}
  |    |    |  {[S]}
  |    |  <[B]>
  |    |    |  <[A]>
  |    |    |    |  <[C]>
  |    |    |    |    |  {[A]}
  |    |    |    |    |    |  {[S]}
```

etc.

Cette implémentation devient lourde à cause du nombre de noeuds que l'on doit construire.

2.2.4 - L'implémentation en Liste (Arbre Hybride de hauteur 0)

En imposant une hauteur d'arbre de 0, nous obtenons une liste composée de tous les mots du dictionnaire.

Cela donne :

```
<[.]>
  |  {[AA, AALENIEN, AALENIENNE, AALENIENNES, AALENIENS, AAS, ABACA, ABACAS, ABACOST, ...
]}
```

Cette implémentation est très lourde et lente, car la liste contient 386 264 mots.

2.2.5 - L'implémentation en HashSet (dites naïve)

L'implémentation en HashSet est un ensemble d'objets sans doublons.

Nous utilisons pour cela deux HashSet, un pour représenter le dictionnaire et l'autre pour représenter les préfixes de chaque mot.

Le problème de cette implémentation est qu'elle ne prend pas en compte les mots trouvés de plusieurs manières différentes.

De plus, cette implémentation prend beaucoup d'espace en mémoire que l'implémentation en arbre hybride.

Récapitulatif des performances pour l'implémentation HashSet (en nanosecondes) :

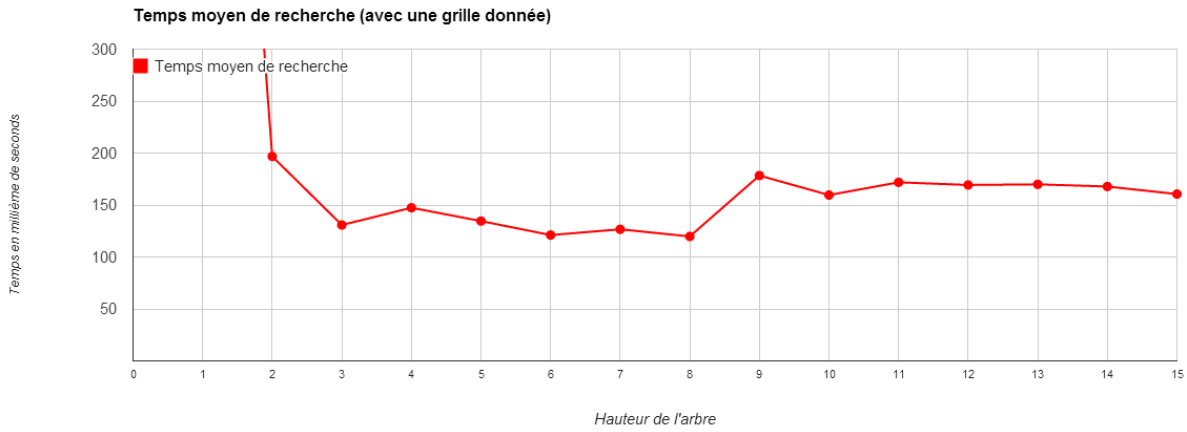
Temps moyen d'initialisation (HashSet)	Temps moyen de recherche (HashSet)	Temps moyen total (HashSet)
665 501 022	42 066 733	707 567 756

Alors que l'arbre hybride possède un temps moyen total d'environ 450 000 000 nanoseconds pour les meilleures hauteurs.

3 - Benchmark

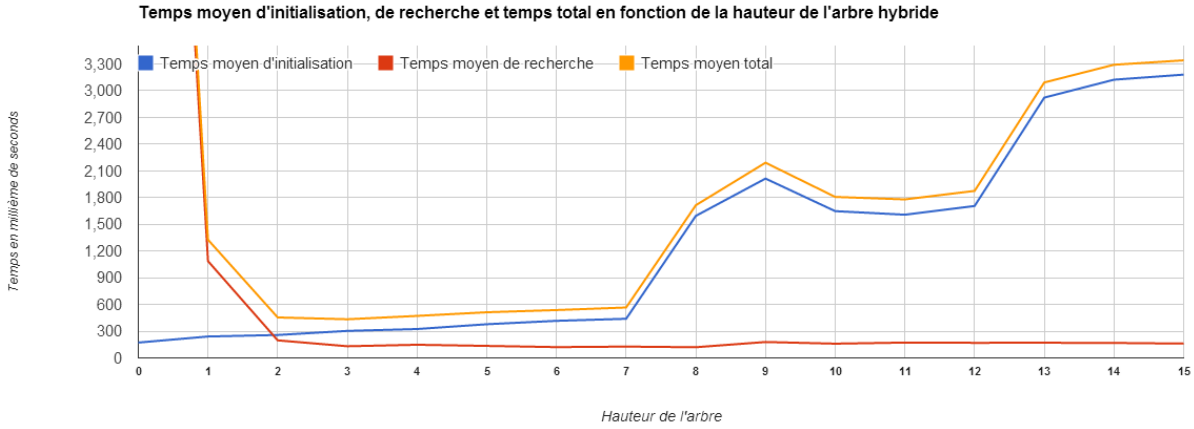
3.1 Benchmark de l'arbre Hydride

Voici le graphique de temps moyen de recherche



D'après le graphique, on constate que la hauteur optimale pour la recherche serait entre 3 et 8.

Voici le graphique de temps moyen d'initialisation, de recherche et la somme des deux temps



D'après le graphique, on constate que la hauteur optimale pour le temps total serait entre 2 et 7.

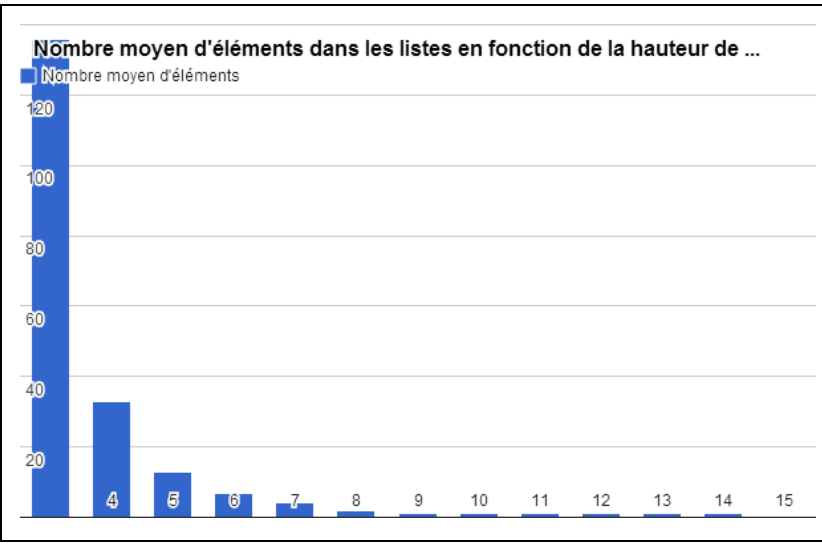
3.2 Moyenne et taille maximale des listes

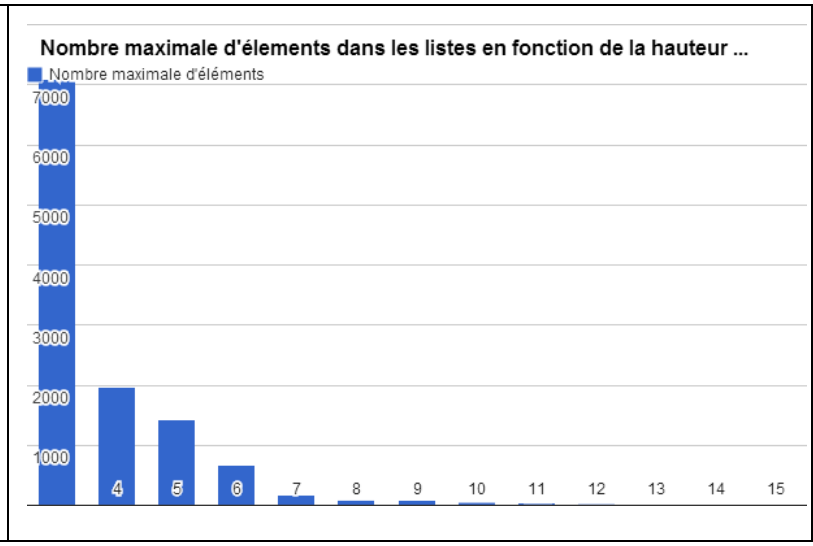
Nous avons effectué des calculs de moyenne et maximum sur les tailles des listes.
Pour cela, nous avons utilisé des variables statiques dans la classe ArbreHybrid :

```
//Variable utilisée pour l'affichage textuel de l'arbre
private static int compteurEspace = 0;

//Nombre de mots contenus dans toutes les arrayChaine de l'arbre
private static int nbElementList = 0;

//Nombre d'arrayChaine
private static int nbListe = 0;

//Taille maximale des listes dans tout l'arbre
private static int tailleMaxListe = 0;
```

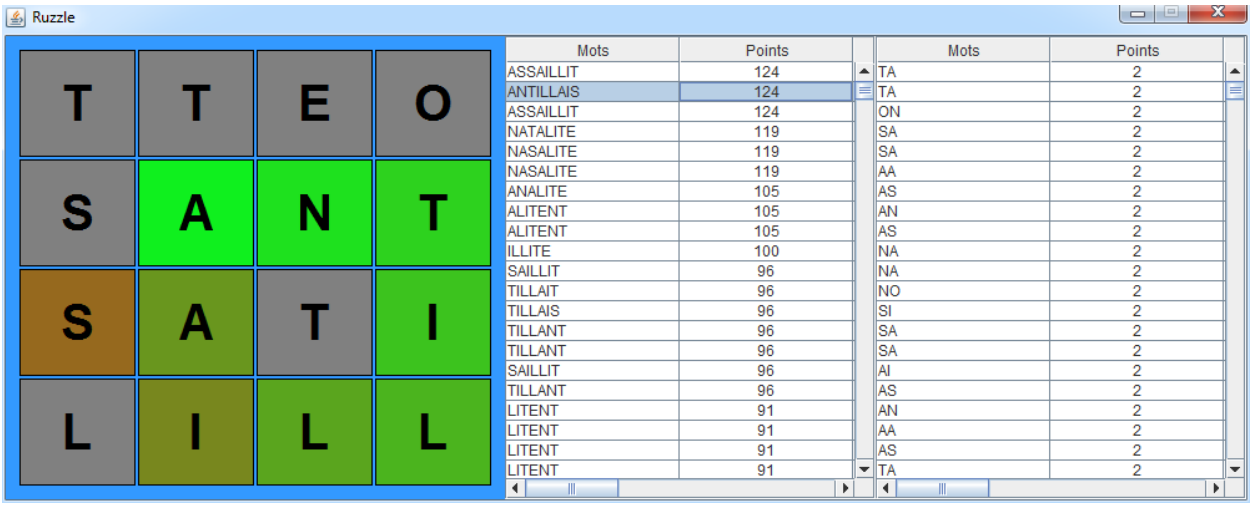
Tableau récapitulatif du nombre moyen et maximum d'éléments dans les listes, en fonction de la hauteur

Hauteur	Nombre moyen d'éléments	Nombre maximale d'éléments
0	386264	386264
1	14856	14856
2	1214	1214
3	136	7087
4	33	1972
5	13	1430
6	7	681
7	4	168
8	2	96
9	1	87
10	1	52
11	1	30
12	1	21
13	1	11
14	1	4
15	0	1





6 - L’interface graphique



L’interface graphique est séparée en 2 parties distinctes, à gauche nous avons la grille du jeu comportant les lettres, et à droite nous avons deux tableaux de résultats.

Lorsqu’on clique sur une ligne d’un tableau, le mot s’affiche automatiquement sur la grille avec un dégradé de couleur du vert vers le rouge pour indiquer le tracé a effectuer.

Le premier tableau est trié par points décroissants et le 2eme est trié par points croissants.
Ces 2 tableaux possèdent une scrollbar verticale qui permet d’afficher plus de mots, et une scrollbar horizontale qui permet d’afficher le parcours qui est masqué par défaut.

L’interface a été développée avec swing.

Nous avons un Jpanel principal contenant 2 Jpanel, MatricePanel et WordPanel.

La MatricePanel est constituée d’un tableau de 4x4 JToggleButton, dont la couleur est modifiable.
Le WordPanel est constitué de 2 JPanel WordPanelTable, dont chacun est constitué d’un JTable.

Conclusion

Pour conclure on peut dire que la meilleure implémentation que nous avons essayée est l'arbre hybride, pour une hauteur optimale comprise entre 2 et 7 par rapport à un temps moyen total, en comparaison avec les autres implémentations qui sont parfois très couteuses en temps, en espace et ne répondant pas aux exigences attendues.

Le projet nous a permis d'utiliser les bibliothèques de java et de Swing, ce qui nous a permis d'améliorer nos connaissances.

On a pris beaucoup de plaisir à travailler sur ce sujet, nous avons également acquis un enrichissement niveau algorithmique.

Remerciement

Avant de terminer, nous tenons à remercier tout particulièrement Mr Arnaud Carayol, notre tuteur de projet, pour l'aide et le temps qu'il nous a apportés durant tout le semestre de projet tutoré. Ses conseils nous ont beaucoup aidés à l'élaboration de notre projet.