Etude de Cas

Réalisation du jeu Ruzzle

Sommaire

Introduction :

La réalisation de ce projet s'est effectuée via le pattern MVC (Modèle-Vue-Contrôleur), ce qui nous a permis de mieux répartir les tâches et d'avancer de manière synchrone. La partie Modèle est celle qui va être détaillée dans les pages suivantes, nous considérons que les parties Vue et Contrôleur sont similaires entre les différents projets, une brève description sera tout de même faite. Nous allons donc détailler les structures utilisées, les algorithmes ainsi que leur complexité.

Structures de données :

Nous utilisons simplement 3 structures de données pour représenter d'une part le dictionnaire (2 structures), et de l'autre la grille (1 structure).

Dictionnaire :

Les principales contraintes quand on décide de représenter un dictionnaire de plus de 300 000 mots en mémoire est d'abord l'efficacité de stockage, et ensuite la possibilité de recherche qui ne doit pas être exponentiellement longue. Ceci à réduit nos choix à 2 structures de données potentielles, les tables de hashage et les trie. Notre choix s'est porté sur l'utilisation de Trie (appelé arbre prefixe), qui nous semble le plus efficace et le plus facile à implémenter. En effet en temps d'accès aux données, les tables de hashage et les trie sont égaux, cependant l'implémentation du premier est plus délicat. En effet les tables de hashage demandent de créer des fonctions qui permettent de convertir une chaîne de caractères en nombre qui sera ensuite stocké. Le principal inconvénient réside dans la gestion des collisions, qui peut arriver dans le cas où 2 chaînes de caractères différentes sont représentées par le même nombre. Ceci complexifie donc la mise en place de tables de hashage, en effet pour être efficace il aurait fallu implémenter des fonctions de hashage efficaces, donc complexes.

Reste donc les Trie, qui nous permettent de représenter un dictionnaire en mémoire. Chaque mot est stocké dans l'arbre grâce aux nœuds de celui-ci, qui contiennent un caractère du mot en question. Chaque descendant d'un nœud possède donc le même préfixe, ce qui nous permet de stocker des mots similaires au même endroit sans devoir utiliser 2 espaces mémoires pour stocker le même mot au singulier et pluriel (exemple des mots singuliers/pluriels). De plus la recherche s'en retrouve facilitée, en effet elle est fonction de la longueur de la chaîne de caractère, et non pas de la position de ses lettres dans l'alphabet. Voici donc une représentation du Trie dans notre programme :

```
// Struct for a node in the Trie
typedef struct sNode {

    int isEnd;
    struct sNode *child[ALPHABET];

} Node;

// Struct of the Trie
typedef struct sTrie {

    Node *root;
} Trie;
```

Nous modélisons le Trie par une racine (root), qui est ensuite le point de départ de chaque nœud. Un nœud possède un champ pour savoir s'il s'agit d'une feuille ou non, mais aussi de pointeurs vers les nœuds suivants. Nous pouvons voir cela comme étant un élément dans un tableau de 26 caractères (longueur de l'alphabet), pointant vers le prochain caractère du mot lui aussi stocké dans un tableau de 26 caractères, et ainsi de suite. L'avantage de cette représentation est que la taille du Trie est fonction du nombre de mots à ajouter, il n'y a pas de mémoire non utilisée dans le Trie.

Grille :

La représentation en mémoire de la grille est relativement simpliste, nous avons utilisé un tableau bi-dimensionnel pour représenter la représenter. Nous créons en mémoire chaque cellule ou case, et nous déclarons un double tableau de ces éléments. Voilà la structure qui représente chaque case :

```
// Struct for a Cell
typedef struct sCell {

    char letter;
    int score;
    int isVisited;
    char bonus[2];
} Cell;
```

Une case est donc renseignée par la lettre qu'elle contient, en informations annexes nous avons aussi son score (le score de la lettre, sans modificateur), un champ (détaillé plus tard) qui va nous permettre de savoir si la case a été visitée ou non, et un stockant le potentiel bonus de la case. Ainsi toutes les informations concernant une seule case se retrouvent au même endroit. La grille en elle-même est donc constituée de 4*4 cellules.

Algorithmes :

A ce stade du projet (création du dictionnaire faite ainsi que celle de la grille entière), il nous a fallu s'attaquer à l'exploration de ces structures, plus particulièrement celle de la grille. Plusieurs contraintes sont apparues :

- Comment détecter les voisins pour continuer le parcours du mot dans la grille ?
- Comment ne pas re-sélectionner la même lettre ou une déjà saisie ?
- Comment récupérer les informations d'une case précise ?
- Comment ne pas re-sélectionner le même mot ?

C'est ici que la liaison entre la partie graphique et la partie algorithmique est importante, la première va nous aider dans la deuxième.

Détecter les voisins d'une case dans la grille :

L'idée première était de calculer le score d'un mot formé par l'utilisateur, une chaîne de caractère était alors constituée au fur et à mesure et l'on utilisait cette dernière pour calculer le score du mot, tout en vérifiant que chaque lettre était voisine de l'autre. Cependant l'algorithme utilisé recherchait les voisins en effectuant le trajet le plus court, ce qui dans le cas où, d'une même lettre, nous pouvions aller vers 2 endroits différents pour la même lettre, posait problème. Pour «coller» au chemin parcouru par l'utilisateur, nous nous sommes servis des positions des lettres dans la grille, données par l'interface graphique. Ainsi à chaque clic, un tableau se remplit avec les coordonnées exactes des lettres. Cependant pour que le clic sur une case soit valide, celle-ci doit être voisine de la précédente, soit en diagonale, soit sur un des 4 côtés. On vérifie pour cela si la différence de coordonnées n'est pas supérieure à 1 sur chaque axe. Si cette différence est plus grande que 1 alors la case n'est pas voisine. Ce qui nous garantie de récupérer des positions de lettres voisines.

Ne pas re-sélectionner la même lettre ou un déjà saisie :

Pour éviter le double clic sur une case, on utilise le champ isVisited présent dans la structure grid. Ce champ se met à 1 lors du clic et est reset au clic droit. Ce qui permet de recliquer sur une lettre après validation d'un mot. Nous utilisons en quelque sorte des «flags» pour éviter de ré-exécuter la même action.

Récupérer les informations d'une case :

Une fois les coordonnées récupérées, nous pouvons retrouver le mot correspondant grâce un algorithme qui parcours le tableau de coordonnées et retrouve chaque lettre associée. Ensuite grâce au Trie nous nous assurons que le mot est dans le dictionnaire, puis nous récupérons le score de chaque lettre. L'algorithme parcourt le tableau de coordonnées, et comme il a pu le faire pour la lettre, récupère le score ainsi que le bonus. Nous récupérons dans des variables distinctes les données extraites, les regroupons et obtenons le score total du mot.

Ne pas re-sélectionner le même mot :

Nous ré-utilisons le même procédé que pour le Trie du dictionnaire, à savoir qu'à chaque première saisie d'un mot nous l'ajoutons dans un Trie, puis nous vérifions les prochaines fois s'il n'est pas dedans, auquel cas le score n'est pas calculé et l'utilisateur est invité à re-saisir un nouveau mot.

Partie graphique :

Pour la partie graphique, il nous a été demandé d'utiliser la librairie SDL version 1.2. Dans un premier temps il a été nécessaire de se familiariser avec cette même librairie. Dans le cadre de ce projet, nous avons décidé de découper le travail sur la partie graphique en trois grands points :

- D'une part l'interface du menu, ensuite celle du jeu en lui-même (la grille plus le score) et enfin l'interface permettant de relancer une partie si on le souhaite. Pour l'interface de départ (menu), nous avons mis en place une structure PrincipalWindow, avec des pointeurs, qui permet de regrouper tous les éléments nécessaires (éléments de la SDL : Surfaces, Rect, Font avec la TTF… etc).

- Ensuite, pour la fenêtre de jeu, nous avons implémenté une structure GridWindow, qui comme son nom l'indique va regrouper les éléments en rapport avec le jeu Ruzzle lui-même. Comme pour la structure précédente, ici nous allons retrouver les différents éléments permettent de gérer tout cela.

- Et enfin la dernière structure ScoreWindow, qui permet de rejouer, va se construire en suivant le même schéma que précédemment.

Méthode de réalisation :

Pour réaliser la partie graphique, nous suivons donc le même schéma pour la création de chaque interface. On commence par la créer, c'est à dire que l'on va chercher à définir tout les éléments qui la composent (image, background, textes… etc), dans une fonction et ensuite nous faisons en sorte d'afficher de tout afficher grâce aux différentes fonctions proposées par la SDL. Bien sûr, le travail sur la partie graphique ne se limite pas à cela, il faut faire en sorte de gérer les événements tels que le clic sur une surface ou même sur la croix pour quitter le jeu, mais d'une manière générale, on crée la fenêtre, on ajoute tout ce qu'il faut dessus et on affiche.