Groupe 3
A1

INSTITUT DE TECHNOLOGIE
UNIVERSITÉ Clermont Auvergne

SAÉ 1.02 : Comparaison d'approches algorithmiques

GRIMAUD Ange

COUDEL-KOUMBA Gabriel

1 Fonctionnalités du logiciel

1.1 Jouer une partie prédéfinie

L'utilisateur choisit un fichier correspondant à une partie existante et joue en tant que chevalier.

1.2 Créer une nouvelle partie

Permet de générer une nouvelle partie personnalisée.

1.3 Afficher la liste des joueurs triée par nom

Trie et montre la liste des chevaliers en ordre alphabétique.

1.4 Afficher la liste des joueurs triée par meilleur score

Trie et affiche les chevaliers en fonction de leurs scores les plus élevés.

1.5 Afficher les statistiques d'un joueur

Recherche et affiche les détails d'un chevalier spécifique, tels que ses scores et performances.

1.6 Quitter

Met fin au programme.

2 Structures de données

2.1 Chevalier (Joueur)

Pour le stockage d'un chevalier, nous avons opté pour un fichier binaire. Ce format est plus compact, réduisant ainsi l'espace disque requis. De plus, il permet un accès rapide et direct à des données spécifiques grâce à l'utilisation de la fonction fseek.

Pour le fichier binaire contenant les chevaliers, nous avons choisi d'écrire en première position le nombre total de chevaliers enregistrés. Ensuite, nous sauvegardons successivement les informations de chaque chevalier.

En ce qui concerne le joueur en lui-même, il est contenu dans la structure "Chevalier" contenant ces champs :

- **nom** qui est une chaine de caractères de 31 caractères maximum.
- **niveau** un entier représentant le niveau du chevalier.
- **xp** un entier représentant l'expérience accumulée.
- **pv** un entier représentant les points de vie.
- **dgts_infliges** un entier représentant les dégâts totaux infligés.

- **nb_parties** un entier représentant le nombre de parties jouées.
- ***tab_scores** un tableau dynamique d'entier, nous n'avons pas fait un tableau dynamique de pointeurs sur un entier car un entier prend 4 octets de stockage, tandis qu'un pointeur peut occuper 4 ou 8 octets selon l'architecture du processeur. Cela optimise l'espace mémoire utilisé.

2.2 Monstres

Les monstres sont eux stockés dans la structure "Monstre", la structure contient ces champs :

- **nom** qui est une chaine de caractères de 15 caractères maximum.
- **niveau** un entier représentant le niveau du monstre.
- **pv** un entier représentant les points de vie.
- **pts_dgts** un entier représentant les dégâts à infligés.
- **nb_armes** un entier représentant le nombre d'armes qu'il possède.

Les groupes de monstres sont représentés par une file chaînée, définie avec les champs suivants:

- **tete** un pointeur vers le premier monstre de la file.
- **queue** un pointeur vers le dernier monstre de la file.

Elle permet une gestion dynamique de la taille des groupes sans gaspillage de mémoire.

2.3 Contextes

Le contexte d'une partie est stocké dans une pile chaînée, avec la structure suivante :

- **contexte** une chaîne de caractères représentant le contexte actuel (par exemple, l'état du jeu).
- **grp1** un pointeur vers un groupe de monstres (file).
- **grp2** un pointeur vers un autre groupe de monstres (file).
- **suiv** un pointeur vers le contexte suivant dans la pile.

Nous avons choisi une pile chaînée, car elle permet de gérer les étapes du jeu en mode LIFO (Last In, First Out), elle adapté à la navigation entre les niveaux.

2.4 Attaques

Les attaques sont représentées par une liste chaînée, définie comme suit :

- **attaque** un caractère représentant une attaque (par exemple, P, F, C, O ou #).
- **suiv** un pointeur vers l'attaque suivante.

Ce choix est justifié par la gestion dynamique et efficace des séquences d'attaques, dont la longueur peut varier selon les situations de jeu.

Nous avons choisi de gérer les structures de cette manière pour leur efficacité en termes de mémoire et de gestion dynamique, permettant une manipulation rapide des données tout en minimisant le coût en mémoire.

2.5 Fichiers de données

2.5.1 Fichiers textes

Nous avons choisi d'utiliser des fichiers textes pour la sauvegarde et le chargement des parties car les données sont simples et structurées de manière à être lisible par l'humain. Ceci permet de pouvoir toujours vérifier que les données ne sont pas corrompues. De même pour les fichiers de données contenant les noms possibles de monstres et les contextes de batailles, la portabilité de ses données cruciales pour le déroulement d'une partie. Ce choix offre un bon équilibre entre la simplicité, lisibilité et portabilité.

2.5.2 Fichiers binaires

3 Algorithmes de tri

3.1 Tri rapide (tri_chevalier_rapide_nom())

3.1.1 Principe

Utilise un pivot pour diviser le tableau en deux parties, les éléments inférieurs au pivot à gauche et les éléments supérieurs au pivot à droite.

3.1.2 Complexité

$O(n^2)$ dans le pire cas.

3.1.3 Utilisation

Tri les chevaliers selon leur nom.

3.1.4 Avantages

- Très efficace pour un grand nombre de données.
- Fonctionne sans avoir à créer d'autres tableaux.

3.1.5 Inconvénients

- Consomme un peu plus de mémoire à cause des appels récursifs.
- Performances non régulières.

3.2 Tri fusion (tri_chevalier_fusion_score())

3.2.1 Principe

Divise le tableau en deux moitiés, le tri récursivement, puis fusionne les deux moitiés triées.

3.2.2 Complexité

O(nlog(n)) dans tous les cas.

3.2.3 Utilisation

Tri les chevaliers selon leur score maximum.

3.2.4 Avantages

- Performances régulières.
- Adapté pour un grand nombre de données.

3.2.5 Inconvénients

- Nécessite de l'espace de stockage pour les tableaux auxiliaires.
- Plus lent que le tri rapide pour un petit nombre de données.

4 Algorithmes de recherche

4.1 Recherche dichotomique (estPresent())

4.1.1 Principe

Fonctionne sur un tableau trié. Divise l'espace de recherche par deux à chaque itération.

4.1.2 Complexité

O(log(n)) dans tous les cas.

4.1.3 Utilisation

Recherche si un nom est présent dans un tableau trié de noms.

4.1.4 Avantages

- Très rapide, surtout quand il y a beaucoup de données.
- Limite l'espace mémoire occupé.

4.1.5 Inconvénient

- Nécessite que les données soient triées.

4.2 Recherche linéaire (chevalier_existe()) :

4.2.1 Principe

Parcours séquentiellement un fichier binaire pour trouver un chevalier ayant un nom spécifique.

4.2.2 Complexité

O(n) dans tous les cas.

4.2.3 Utilisation

Recherche un chevalier dans un fichier binaire non trié.

4.2.4 Avantages

- Facile à implémenter.
- Fonctionne même sur les données non triées.

4.2.5 Inconvénient

Inefficace pour un grand nombre de données car on doit parcourir tous les éléments.