2025

PYTHSTONE

RAPPORT DÉVELOPPEMENT EFFICACE

Marwane El Boumyaoui - Farouk Bendeddouche

SOMMAIRE

Marwane El Boumyaoui - Farouk Bendeddouche

DESCRIPTION DU PROJET

Le projet s'inspire du célèbre jeu de cartes stratégique Hearthstone. Il s'agit de développer une version simplifiée et textuelle de ce jeu en Python, mettant en œuvre des concepts fondamentaux de structures de données.

L'objectif principal est de simuler un affrontement entre deux joueurs dans un environnement textuel, en gérant les différentes mécaniques essentielles du jeu. Chaque joueur dispose d'un deck de cartes, d'un plateau où il peut invoquer des serviteurs, et de points de vie qu'il doit protéger.

Le projet vise également à démontrer une gestion efficace des données et des événements grâce à l'utilisation de structures telles que les piles, les files, et les listes chaînées.

DESCRIPTION

FONCTIONNALITÉS

- Système de Deck et de Pioche

- Les decks des joueurs sont implémentés à l'aide de structures de files pour représenter la pioche. Les cartes sont piochées dans l'ordre où elles ont été ajoutées.

- Gestion des Serviteurs

- Les serviteurs invoqués sur le plateau sont gérés via une liste chaînée. Chaque serviteur possède des caractéristiques spécifiques (points de vie, points d'attaque, effets).

- Les actions des joueurs (attaque, pioche, utilisation de pouvoirs ou de cartes).

- Gestion des Points de Vie et de la Condition de Victoire

- Chaque joueur commence avec un total de points de vie. L'objectif est de réduire les points de vie de l'adversaire à zéro en utilisant des serviteurs ou des sorts.

FONCTIONNALITÉS

STRUCTURE UTILISÉES

Le projet que nous avons développé repose sur plusieurs structures de données fondamentales, utilisées pour gérer et manipuler les différents éléments du jeu.

Pile

Dans notre projet, la pile est utilisée pour représenter à la fois le deck de cartes et le cimetière des serviteurs. Lors des actions de jeu, les joueurs "piochent" des cartes en dépilant les éléments de la pile.

En cas de mort d'un serviteur, celui-ci est également ajouté à la pile, permettant ainsi de récupérer facilement le dernier serviteur mort pour appliquer des effets ou le réanimer si nécessaire.

Liste chaînée

Dans notre projet, la liste chaînée est utilisée pour représenter à la fois le plateau (board) et la main du joueur. Chaque élément de la liste correspond à une carte ou à un serviteur.

Cette structure permet une gestion dynamique des éléments, permettant d'ajouter ou de retirer des cartes facilement. Pour le plateau, la liste chaînée permet de gérer l'ordre des serviteurs en jeu, tandis que pour la main du joueur, elle permet de gérer l'ajout et le retrait des cartes à mesure qu'elles sont jouées.

File

La file est utilisée pour gérer à la fois le cimetière des serviteurs morts et les effets associés aux serviteurs. Lorsqu'un serviteur meurt, il est ajouté à la file, préservant l'ordre des éliminations.

Cette structure permet également de gérer les effets qui doivent être traités dans l'ordre, comme les effets de fin de vie des serviteurs. En cas de réanimation ou d'activation d'effets spécifiques, la file offre un moyen efficace de récupérer et traiter ces serviteurs ou effets au moment opportun.

COMPLEXITÉ

Dans cette section, nous analyserons la complexité des principales méthodes implémentées dans le projet, en évaluant leur efficacité en termes de performance.

Complexité de la méthode attaque :

```
def attaque(self):  1 usage (1 dynamic)  Marwane El +1
    if self.board.taille() == 0 :
        print(f"{self.name} n'a pas de serviteurs.")
        print("#-------------------------------------------------------------------------#")
        return
    if self.ennemy.board.taille() == 0:
        print(f"Votre adversaire n'a pas de serviteurs.")
        print("#-------------------------------------------------------------------------#")
        return

    print("Serviteurs disponibles pour attaquer :")
    self.getBoard()

    # Choisir un attaquant
    choix_attaquant = input("Choisissez un serviteur pour attaquer (-1 pour passer le tour) : ")
    try:
        choix_attaquant = int(choix_attaquant)
    except ValueError:
        print("Entrée invalide. Action annulée.")
        return

    while choix_attaquant == 0:
        choix_attaquant = input("Choisissez un serviteur et non un héros (-1 pour passer le tour) : ")
        try:
            choix_attaquant = int(choix_attaquant)
        except ValueError:
            print("Entrée invalide. Action annulée.")
```

Complexité totale :

En résumé, les parties principales de la méthode attaque qui influent sur la complexité sont les appels à taille(), getBoard(), attaquer(), et nettoyerPlateau().

Chaque opération dans la méthode attaque (à l'exception des choix de l'utilisateur) a une complexité de O(n).

Ainsi, la complexité totale de la méthode attaque est O(n), où n est le nombre de serviteurs sur le plateau du joueur ou de l'ennemi, car chaque opération principale dépend de cette taille de plateau.

COMPLEXITÉ

Complexité de la méthode JouerCarte():

```
def jouerCarte(self):  1 usage (1 dynamic)  Marwane El Boumyaoui
    if self.hand.taille() == 0 :
        return
    print("#--------------------------------------------------------------------------#")
    print("#\tAffichage de la main :")
    print("#--------------------------------------------------------------------------#")
    self.hand.afficher()
    print("#--------------------------------------------------------------------------#")
    try:
        num = input("#Veuillez jouer une carte (par son numéro) (-1 pour ne pas jouer) : ")
        if num == "-1" :
            print("#--------------------------------------------------------------------------#")
            return
        carteJouee = self.hand.getNoeud(int(num)-1)
        if carteJouee is None:
            print("#Indice invalide. Aucune carte n'a été jouée.\n")
            return

        # Accéder à la valeur pour l'ajouter au plateau
        self.board.ajouter(carteJouee.valeur)

        # Supprimer le nœud de la main
        self.hand.supprimerNoeud(carteJouee)
        print(f"#La carte '{carteJouee.valeur.getName()}' a été jouée.")

        print("#--------------------------------------------------------------------------#")
```

• • •

Complexité :

1. Affichage de la main :
2. Récupération d'une carte par son indice :
3. Ajout de la carte au plateau :
4. Suppression de la carte de la main

La complexité globale de cette méthode jouerCarte est donc dominée par les opérations qui parcourent la liste, ce qui nous donne une complexité O(n), où n est la taille de la main (self.hand).

COMPLEXITÉ

Complexité de la méthode démarrer et ses sous méthodes:

```
def demarrer(self):  1 usage  Marwane El Boumyaoui
    while not self.verifPartieTermine():
        print("\n#----------------------------------------------------------------")
        print("#C'est au tour de", self.tourActuel.name)
        self.gererPhase()
        self.changerTour()
    print("Fin de la partie.")

def verifPartieTermine(self):  1 usage  Marwane El Boumyaoui
    # Vérifie si le joueur actuel ou son adversaire a perdu
    if self.tourActuel.hero.hp <= 0:
        print(f"La partie est terminée ! {self.tourActuel.name} a perdu.")
        return True
    adversaire = self.getAdversaire()
    if adversaire.hero.hp <= 0:
        print(f"La partie est terminée ! {adversaire.name} a perdu.")
        return True
    return False

def gererPhase(self):  1 usage  Marwane El Boumyaoui
    joueurActif = self.tourActuel
    self.phaseDebut(joueurActif)
    print("#\tPhase principale :")
    self.phasePrincipale(joueurActif)
    print("#\tPhase d'attaque :")
    self.phaseAttaque(joueurActif)
```

1. **demarrer()** :
    a. La boucle s'exécute jusqu'à la fin de la partie, appelant verifPartieTermine() et gererPhase(). Cela donne une complexité O(T * n), où T est le nombre de tours et n est la taille des mains ou des serviteurs.
2. **verifPartieTermine()** :
    a. Vérification des points de vie des héros, ce qui est une opération O(1).
3. **gererPhase()** :
    a. phaseDebut() : O(1) pour l'ajout et la régénération du mana.
    b. phasePrincipale() : Appel de jouerCarte().
    c. phaseAttaque() : Appel de attaquer().

Complexité globale :

La complexité globale de demarrer() est **O(T * n)**, avec T le nombre de tours et n la taille des mains ou des serviteurs.