Documentation du Code du Jeu *Sixteen-Soldiers*

Bienvenue dans la documentation complète du code pour le projet *Sixteen-Soldiers*. Cette documentation vous guidera à travers les étapes d'installation, de compréhension de l'architecture du projet, et de l'utilisation des classes et fonctions principales.

Table des matières

Installation et Débogage du Code 🚀

Prérequis

1. **Disposer de Python 3.12 ou version supérieure**
   Si Python n'est pas installé sur votre machine, vous pouvez le télécharger et l'installer depuis le site officiel :
   👉 Télécharger Python.

   ⚙️ Installation de Python sous différents systèmes d'exploitation

   - **Windows** :
     - Téléchargez l'installateur correspondant à votre architecture (32 ou 64 bits).
     - Cochez l'option `Add Python to PATH` avant d'installer.
     - Suivez les étapes de l'installateur.
   - **macOS** :
     - Téléchargez l'installateur `.pkg` et suivez les instructions.
     - Alternativement, vous pouvez utiliser `Homebrew` :

       ```
       brew install python
       ```
   - **Linux** :
     - Utilisez votre gestionnaire de paquets. Par exemple, pour Ubuntu/Debian :

       ```
       sudo apt update
       sudo apt install python3.12
       ```

2. **Vérification de la version de Python installée**
   Après installation, ouvrez un terminal ou une invite de commande, et exécutez :

   ```
   python --version
   ```

   ou

   ```
   python3 --version
   ```

   Vous devriez obtenir une version égale ou supérieure à `3.12`.

Installation des Dépendances 📦

Le projet utilise un fichier `requirements.txt` pour gérer ses dépendances.

1. Assurez-vous d'avoir `pip`, le gestionnaire de paquets Python. Vous pouvez le vérifier en exécutant :

   ```
   pip --version
   ```

   Si `pip` n'est pas installé, vous pouvez l'ajouter via :

   ```
   python -m ensurepip
   python -m pip install --upgrade pip
   ```

2. Placez-vous dans le répertoire du projet, puis installez les dépendances avec la commande suivante :

   ```
   pip install -r requirements.txt
   ```

   ☑ **Cette commande installe tous les packages nécessaires pour exécuter le projet.**

Exécution du Code 🏁

1. Assurez-vous que l'installation s'est déroulée correctement en lançant le projet :

   ```
   python main.py
   ```

   ou

```
python3 main.py
```

2. **Débogage** :
   Si des erreurs surviennent, vérifiez :
   - Que tous les packages sont installés correctement.
   - Que la version de Python est compatible.
   - Consultez les logs générés pour identifier les problèmes spécifiques.

🎉 Une fois le programme exécuté avec succès, vous êtes prêt à découvrir les fonctionnalités du jeu !

Architecture des Fichiers 🗂️

L'architecture du projet est organisée pour faciliter la compréhension, l'extension et la maintenance du code. Chaque dossier a une responsabilité spécifique. Voici un aperçu détaillé :

Dossier `agents` 🤖

Ce dossier contient les agents d'intelligence artificielle capables de jouer au jeu :

- `base_agent.py` : Fournit une structure de base pour tous les agents.
- `random_agent.py` : Un agent simple qui joue de manière aléatoire.
- `main_ai.py` : L'agent principal utilisé pour les compétitions.

💡 **Note Importante :**

- Si vous développez une IA, créez un nouveau fichier pour chaque agent. Chaque fichier doit implémenter une classe retournant une action valide à chaque appel.
- Lorsque votre IA atteint un bon niveau, sauvegardez-la dans `main_ai.py`. Ce fichier sera utilisé comme référence pour les compétitions réelles.

Dossier `models` 🏗️

Ce dossier contient les modèles de données utilisés dans l'application :

- `board.py` : Implémente le plateau de jeu.
  🔑 **Astuce** : Ce fichier est crucial pour l'implémentation de vos IA. Consultez la **class Board** pour mieux comprendre ses fonctionnalités.
- `move.py` : Représente les coups joués dans le jeu.

Dossier `actions` 🎯

Ce dossier regroupe toutes les actions possibles dans le jeu :

- `board_actions.py` : Gère les déplacements des pions.
- `time_actions.py` : Gère la gestion du temps.

- `history_actions.py` : Gère l'historique des actions réalisées.

Dossier `reducers` 🔄

Les reducers sont responsables de la mise à jour de l'état global du jeu en fonction des actions reçues :

- `game_reducer.py` : Gère l'état général du jeu.
- `board_reducer.py` : Met à jour l'état du plateau.
- `time_reducer.py` : Traite la gestion des joueurs et du temps.
- `history_reducer.py` : Met à jour l'historique des coups joués.

Dossier `store` 📊

Le store contient l'état global de l'application et les outils nécessaires pour gérer cet état :

- **Fonctions incluses** :
  - *Dispatch* : Envoie des actions au reducer.
  - *Subscribe* : Permet de surveiller les changements d'état.

Dossier `views` 🖼️

Les vues sont responsables de l'affichage du jeu. Chaque sous-vue est dédiée à une partie spécifique de l'application :

- `base_view.py` : Définit une interface commune pour toutes les vues.

```
class BaseView:
    def subscribe(self, store): ...
    def update(self, state): ...
```

- `main_view.py` : Coordonne la fenêtre principale et les sous-vues.
- Sous-vues spécialisées :
  - `game_board.py` : Gère l'affichage et l'interaction avec le plateau.
  - `player_view.py` : Affiche les informations des joueurs.
  - `history_view.py` : Affiche l'historique des coups joués.

Dossier `utils` 🛠️

Ce dossier contient des fonctions auxiliaires pour le projet :

- `validator.py` : Valide les actions effectuées par les joueurs ou les agents.
- `audio.py` : Gère la musique et les sons.
- `const.py` : Regroupe les constantes globales utilisées dans le projet.

Dossier `saved_game` 💾

Ce dossier contient les fichiers de sauvegarde des parties jouées. Vous pouvez y retrouver toutes vos parties enregistrées pour les analyser ou les rejouer.

👉 **Suivant : Classes Utiles** 🔧

Points à Retenir 🎯

1. La présentation de l'architecture est purement à but informatif ou pour une meilleure prise en main du code.
2. Implémentez votre IA dans le dossier `agents`.
3. Ne modifiez en aucun cas les dossiers mentionnés sauf (`agents`) au risque de faire crasher l'interface et vos IA. Tout crash, est par ailleurs, disqualificatif.
4. Le fichier `main_ai.py` doit toujours contenir votre meilleure IA.

Classes Utiles 🔧

Classe `Board`

La classe `Board` représente le plateau de jeu et gère les mouvements et les captures des soldats.

Attributs

- `battle_field` (Dict[str, Set[str]]): Un dictionnaire représentant les positions adjacentes pour chaque position sur le plateau.
- `soldiers` (Dict[str, Soldier]): Un dictionnaire représentant la valeur du soldat à chaque position sur le plateau.
- `last_action` (Dict): Le dernier mouvement effectué.
- `is_multiple_capture` (bool): Indique si une capture multiple est possible.
- `logger` (logging.Logger): Un logger pour enregistrer les événements du jeu.

Méthodes

```
__init__(self)
```

Initialise le plateau de jeu avec les positions des soldats rouges et bleus.

```
get_neighbors(self, position: str) -> Dict[str, List[str]]
```

Retourne les positions voisines d'une position donnée, classées par valeur (RED, BLUE, EMPTY).

- **Paramètres**:
  - `position` (str): La position pour laquelle obtenir les voisins.
- **Retourne**:
  - Un dictionnaire avec les clés `RED`, `BLUE`, et `EMPTY`, chacune contenant une liste de positions voisines.

Exemple :

```
{
    "RED": ["a3", "b2"],
    "BLUE": [],
    "EMPTY": ["c3"]
}
```

`get_empty_positions(self) -> List[str]`

Retourne une liste de toutes les positions vides sur le plateau.

- **Retourne**:
  - Une liste de positions vides.

`get_soldier_positions(self, soldier_value: Soldier) -> List[str]`

Retourne une liste de toutes les positions occupées par les soldats d'une certaine valeur (RED ou BLUE).

- **Paramètres**:
  - `soldier_value` (Soldier): La valeur du soldat (RED ou BLUE).
- **Retourne**:
  - Une liste de positions occupées par les soldats de la valeur spécifiée.

`get_soldier_value(self, position: str) -> Soldier`

Retourne la valeur du soldat à une position donnée.

- **Paramètres**:
  - `position` (str): La position pour laquelle obtenir la valeur du soldat.
- **Retourne**:
  - La valeur du soldat à la position spécifiée.

`count_soldiers(self, soldier_value: Soldier) -> int`

Compte le nombre de soldats d'une certaine valeur sur le plateau.

- **Paramètres**:
  - `soldier_value` (Soldier): La valeur du soldat (RED ou BLUE).
- **Retourne**:
  - Le nombre de soldats de la valeur spécifiée.

`move_soldier(self, action: Dict)`

Déplace un soldat en fonction du dictionnaire d'action.

- **Paramètres**:

- `action` (Dict): Un dictionnaire contenant les détails du mouvement (`from_pos`, `to_pos`, `soldier_value`).

`capture_soldier(self, action: Dict)`

Capture un soldat en fonction du dictionnaire d'action.

- **Paramètres**:
  - `action` (Dict): Un dictionnaire contenant les détails de la capture (`from_pos`, `to_pos`, `captured_soldier`, `soldier_value`).

`get_last_action(self) -> Dict`

Retourne le dernier mouvement effectué.

- **Retourne**:
  - Le dernier mouvement effectué.

`check_multi_capture(self, soldier_value: Soldier, current_position: str) -> bool`

Vérifie si une capture multiple est possible.

- **Paramètres**:
  - `soldier_value` (Soldier): La valeur du soldat (RED ou BLUE).
  - `current_position` (str): La position actuelle du soldat.
- **Retourne**:
  - `True` si une capture multiple est possible, `False` sinon.

`get_valid_actions(self) -> List[Dict]`

Retourne une liste de toutes les actions valides pour le joueur actuel.

- **Retourne**:
  - Une liste de dictionnaires représentant les actions valides.

    Exemple :

```
[
    {
    "from_pos": "a1",
    "to_pos": "a3",
    "type": "MOVE",
    "soldier_value": "RED"
    },

    {
        ...
    }
]
```

`get_valid_actions_for_position(self, position: str) -> List[Dict]`

Retourne une liste de toutes les actions valides pour une position spécifique.

- **Paramètres**:
  - `position` (str): La position pour laquelle obtenir les actions valides.
- **Retourne**:
  - Une liste de dictionnaires représentant les actions valides pour la position spécifiée.

`is_valid_move(self, from_pos: str, to_pos: str, soldier_value: Soldier) -> bool`

Vérifie si un mouvement de `from_pos` à `to_pos` est valide pour un soldat donné.

- **Paramètres**:
  - `from_pos` (str): La position de départ.
  - `to_pos` (str): La position d'arrivée.
  - `soldier_value` (Soldier): La valeur du soldat (RED ou BLUE).
- **Retourne**:
  - `True` si le mouvement est valide, `False` sinon.

`is_game_over(self) -> Soldier`

Vérifie si le jeu est terminé (un joueur n'a plus de soldats).

- **Retourne**:
  - La valeur du soldat gagnant (RED ou BLUE) si le jeu est terminé, `None` sinon.

Utilisation

Pour utiliser ces attributs et méthodes, vous pouvez créer une instance de la classe `Board` et appeler les méthodes appropriées pour gérer le jeu. Par exemple :

```
board = Board()
print(board.get_neighbors('a1'))
print(board.get_soldier_positions(Soldier.RED))
print(board.get_valid_actions())
```

Classe `Agent`

La classe `Agent` est une implémentation d'un agent IA qui joue des mouvements valides dans le jeu Sixteen-Soldiers. Cette classe hérite de `BaseAgent` et peut être utilisée comme point de départ pour créer des agents IA plus sophistiqués.

Dans le fichier `random_agent.py`, elle est implémentée pour choisir une action aléatoire parmi les actions valides. Ce fichier implémente une classe Agent qui est donc principalement utilisée comme un point de départ ou pour des tests rapides. Elle n'est pas optimisée pour jouer stratégiquement, mais elle offre une base fonctionnelle pour tester les interactions entre les composants du jeu.

Attributs

Attributs de l'Agent

- `soldier_value` *(Soldier)* : La valeur du soldat que l'agent contrôle (RED ou BLUE).
- `data` *(Dict, optionnel)* : Permet de passer des données supplémentaires à l'agent (par exemple, des paramètres ou des états persistants).
- `name` *(str)* : Nom de l'agent (défini comme `"Random Team"` dans le cas du RandomPlayer).

Méthodes Principales

`__init__(self, soldier_value: Soldier, data: Dict = None)`

Le constructeur initialise l'agent avec une valeur de soldat et des données optionnelles.

- **Paramètres :**
  - `soldier_value` : Identifie le type de soldat que cet agent contrôle (par exemple, un pion blanc ou noir).
  - `data` : Dictionnaire facultatif pour passer des informations supplémentaires.
- **Exemple d'usage :**

```
from src.utils.const import Soldier

agent = Agent(soldier_value=Soldier.RED)
print(agent.name)  # "Random Team"
```

`choose_action(self, board: Board) -> Dict`

Cette méthode sélectionne une action valide au hasard parmi toutes les actions possibles pour cet agent.

- **Paramètres :**
  - `board` : Instance de la classe `Board` représentant l'état actuel du jeu.
- **Retourne :**
  - Un dictionnaire représentant l'action choisie. Chaque action est une structure prédéfinie (voir la documentation de la méthode `get_valid_actions()` de la classe `Board` pour le format exact).
- **Exemple de fonctionnement :**
  Supposons que l'état du plateau permette trois actions valides :

```
[
    {"from_pos": (0, 0), "to_pos": (1, 0)},
    {"from_pos": (0, 0), "to_pos": (2, 0)},
```

```
    {"from_pos": (0, 0), "to_pos": (1, 1)},
]
```

La méthode choisira aléatoirement une des trois actions disponibles.

- **Code Simplifié :**

```
valid_actions = board.get_valid_actions()
return random.choice(valid_actions)
```

Implémentation de Nouveaux Agents

Pour implémenter un nouvel agent, vous pouvez hériter de la classe `BaseAgent` et redéfinir la méthode `choose_action` pour inclure votre logique personnalisée. Voici un exemple de comment créer un nouvel agent :

Exemple Pratique :

```
from src.agents.base_agent import BaseAgent
from src.models.board import Board
from src.utils.const import Soldier

class CustomAgent(BaseAgent):
    """AI agent with custom logic"""

    def __init__(self, soldier_value: Soldier, data: Dict = None):
        super().__init__(soldier_value, data)
        self.name = "Custom Team"

    def choose_action(self, board: Board) -> Dict:
        """
        Choose an action based on custom logic.
        Args:
            board: Current game board state
        Returns:
            Custom chosen valid action for the soldier_value
        """
        valid_actions = board.get_valid_actions()

        # Implémentez votre logique personnalisée ici
        best_action = self.custom_logic(valid_actions)

        return best_action

    def custom_logic(self, valid_actions: List[Dict]) -> Dict:
        """
        Custom logic to choose the best action.
        Args:
```

```
            valid_actions: List of valid actions
        Returns:
            Best action based on custom logic
        """
        # Implémentez votre logique personnalisée ici

        return action
```

Attention :

Ceci n'est qu'un exemple d'architecture de code pour l'implémentation de la classe `Agent`. Vous n'avez pas à implémenter une fonction `custom_logic` comme indiqué ici. Cependant, il est crucial de maintenir la fonction `choose_action`, qui sera appelée à votre tour de jeu pour retourner l'action que vous souhaitez exécuter. Vous pouvez personnaliser cette fonction en y intégrant votre propre logique de décision.

Si vous avez des questions ou des difficultés, contactez-nous dès que possible.

📧 Email : kyfaxgroup@gmail.com

📞 WhatsApp : https://chat.whatsapp.com/Lu4oj0uzr5g6lpG0HI54Xr

Bon codage et amusez-vous bien ! 🎉

KYFAX ✨