Projet Final

NSI
Projet 5

Sommaire

1.Présentation et Structure

- Environnement d'exécution et But

- Cette Application à été développer en et pour le langage de programmation Python 3.X même si la majorité du code est compatible et des mesures on été prise pour que le code soit en partie compatible avec Python 2.X avec la version minimum 2.5 .

- Ce projet a pour But de démontrer l'application de l'algorithme appeler MinMax qui calcule toute les possibilités et les valeurs que un coups va rapporter ;

et ensuite toutes les possibilités de coups des adversaires après le premier tour (avec le coup précédent calculer)

et ainsi de suite jusqu'à ce que une limite soit atteint .

(NOTE 1.1 : aussi si le coups va directement gagner la partie)

(NOTE 1.2 : il y a la possibilité de ne pas mettre de limite ,**cela est déconseiller**)

- Ce projet à plusieurs parties et utilitaires pour l'accompagner (Voir 1.Fichiers pour savoir lesquels sont obligatoires et ce qu'ils font) .

- Ce projet est accompagner d'une interface graphique pour démontrer un exemple de jeux de Morpions et un jeux d'échecs .

- Une application concrète qui est déployer dans le monde est dans les sites de jeux d'échecs en ligne pour jouer contre un robot (même si il y a d'autre algorithme)

à ce but l'utilitaire server.py utilise la convention UCI .

- Architecture

Le projet utilise des Singleton et de l'inhéritance pour son architecture ;
le corps du projet est l'algorithme MinMax dans le fichier algorithm.py ;

Il y a ensuite différent utilisateur de algorithm.py :

cela est utiliser par Morpion.py et echecs.py avec leurs logique pour crée les jeux .

GUI_Morpion.py et GUI_echecs sont donc leurs interface graphique
GUI_launcher.py regroupe donc toute les interface graphiques

Toute les interfaces graphique utilise la librairie GUI_helper.py

NOTE 1.3 : l'utilitaire server.py qui utilise algorithm.py et la convention UCI pour communiquer sans interface graphique par stdin et stdout (il est facile de "pipe" sous linux et windows (utilitaire a part) le stdin et stdout avec le réseau).

- Fichiers

1 - Collection Implémentation du jeux de Morpion:

- algorithm.py
- Morpion.py

2 - Collection Implémentation du jeux de Morpion Graphique:

- algorithm.py
- Morpion.py
- GUI_helper.py
- GUI_Morpion.py

3 - Collection Implémentation du jeux d'échecs Graphique: [NON-FONCTIONEL]

- algorithm.py
- échecs.py
- GUI_helper.py
- GUI_echecs.py

4 - Collection Regroupement des Interface Graphique :

- algorithm.py
- GUI_helper.py
- Morpion.py
- GUI_Morpion.py
- échecs.py
- GUI_echecs.py (affiche quand même le jeux vide)
- GUI_launcher.py

Utilitaires (dossier Utilitaires - UCI):

5 - Collection pour faire un serveur : [démonstration seulement]

- algorithm.py

- server.py

- UCI_serialize.py

- echecs.py

(NOTE 1.4 : pour un serveur il est assumer que l'on joue au échecs mais cela peut être adapter pour n'importe quelle jeux avec une méthode de sérialisation)

6 - Collection pour visualiser un arbre :

- visual.py

- visual_DATA.py

(NOTE 1.5 : le format pour visual_DATA.py est DATA={} avec chaque clé du dict sera un nœud ; la valeur devrai être tuple(valeur , est_gagner , {sous_arbre}))

7 - Collection pour génération de documentation :

- gen_doc.py

2.L’Algorithme MinMax

- single thread

minmax(_board: algorithm.board, cache=None, max_depth=3, max_time=5):

Performs a minimax search on a given board object to find the optimal move.

Args:

_board (board): The board object representing the current state of the game.

cache (dict, optional): A dictionary to store the search tree for caching purposes. Defaults to None.

max_depth (int, optional): The maximum depth of the search tree. Defaults to 3.

max_time (int, optional): The maximum time in seconds allowed for the search. Defaults to 5.

Returns:

dict: The search tree representing the optimal moves and their values.

Implémentation

minmax_s(_board: algorithm.board, cache=None, max_depth=3, max_time=5):

Performs a minimax search on a given board object to find the optimal move.

Args:

_board (board): The board object representing the current state of the game.

cache (dict, optional): A dictionary to store the search tree for caching purposes. Defaults to None.

max_depth (int, optional): The maximum depth of the search tree. Defaults to 3.

max_time (int, optional): The maximum time in seconds allowed for the search. Defaults to 5.

Returns:

dict: The search tree representing the optimal moves and their values.

recursive_search(tree, depth):

Recursively searches through the tree and returns a list of winning moves.

Args:

tree (tuple): The search tree in the format (value, is_win, sub_tree).

depth (int): The current depth in the search tree.

Returns:

hash : winning moves.

search_empty(tree, queue, lv, max_depth):

Recursively find all empty path values in the tree.

- multi-process

[NON-FONCTIONEL]

3.Interface Graphique (GUI)

- librairie GUI_helper

Librairie pour simplifier les taches graphiques

build_reverse_index(dic: dict):

Construit un index inverse a partir d'un dictionnaire.

Args:

dic (dict): Le dictionnaire a partir duquel construire l'index inverse.

Returns:

dict: L'index inverse construit.

Raises:

ValueError: Si le parametre `dic` n'est pas un dictionnaire.

Example:

```
dictionary = {'apple': ['fruit'], 'carrot': ['vegetable'], 'banana': ['fruit', 'yellow']}
reverse_index = build_reverse_index(dictionary)
```

default_main(*args):

Fonction main par defaut qui cree une fenetre Tkinter et initialise l'etat global.

dispatch(state: GUI_helper.gstate, *args):

Handles the state and executes the appropriate function based on the current state stack.

Args:

state (gstate): The state object representing the current state.

*args: Variable length argument list.

Raises:

ValueError: If the provided state is not of type `gstate`.

Example:

state = gstate(...)

dispatch(state)

get_open_window():

handy_backbtn(state: GUI_helper.gstate):

Cree un bouton "Retour" dans la grille de l'objet `gstate` donne.

Args:

state (gstate): L'objet `gstate` representant l'etat global.

Returns:

tkinter.Button: Le bouton "Retour".

Raises:

ValueError: Si l'objet `state` fourni n'est pas de type `gstate`.

Example:

```
state = gstate(root, grid=grid)
back_button = handy_backbtn(state)
```

handy_clear(state: GUI_helper.gstate):

Efface le contenu de la grille dans l'objet `gstate` donne.

Args:

state (gstate / widget): L'objet `gstate` representant l'etat global.

Raises:

ValueError: Si l'objet `state` fourni n'est pas de type `gstate`.

Example:

```
state = gstate(root, grid=grid)
handy_clear(state)
```

handy_config(state: GUI_helper.gstate, row: int, column: int = None):

Configure la disposition de la grille dans l'objet `gstate` donne avec le nombre specifie de lignes et de colonnes.

Args:

state (gstate): L'objet `gstate` representant l'etat global.

row (int): Le nombre de lignes dans la grille.

column (int, optionnel): Le nombre de colonnes dans la grille. S'il n'est pas specifie, il est par defaut egal e la valeur de `row`.

Raises:

ValueError: Si les valeurs de ligne ou de colonne fournies ne sont pas des entiers.

Example:

```
state = gstate(root, grid=grid)
handy_config(state, 3, 4)
```

handy_grid(master=None, repeatewidget=None, repeatewidget_func: <built-in function callable> = None, rows: int = 3, cols: int = None, binding=None, *args, **kwargs):

Cree une grille de widgets en fonction des parametres fournis.

Args:

repeatewidget (function, optionnel): Une fonction qui cree un seul widget.

repeatewidget_func (function, optionnel): Une fonction qui cree un widget en fonction des indices de ligne et de colonne.

master: Le widget maetre representant la grille.

rows (int): Le nombre de lignes dans la grille.

cols (int, optionnel): Le nombre de colonnes dans la grille. S'il n'est pas specifie, il est par defaut egal e la valeur de `rows`.

binding (function): Une fonction pour lier des actions aux widgets crees.

*args: Liste d'arguments de longueur variable transmise e la fonction de creation de widget.

**kwargs: Arguments arbitraires passes e la fonction de creation de widget.

Raises:

ValueError: Si ni repeatewidget ni repeatewidget_func ne sont fournis.

NotImplementedError: Si la fonction de liaison n'est pas definie.

Returns:

list: Une liste e 2 dimensions representant la grille des widgets crees.

Example:

```
def create_label(master, row, col):
    label = tk.Label(master, text=f"Ligne {row}, Colonne {col}")
    return label

grid = handy_grid(repeatewidget_func=create_label, master=frame, rows=3, cols=4,
binding=bind_func)
```

handy_show_grid(master):

Affiche la disposition de la grille en creant des frame dans la grille avec marque la ligne et colone.

Args:

master (tk.Widget): Le widget maetre representant la grille.

Raises:

ValueError: Si l'objet `master` fourni n'est pas de type `tk.Widget`.

Example:

```
grid = tk.Frame(root)
handy_show_grid(grid)
```

```

menu_debug(state: GUI_helper.gstate):

menu_demo(state: GUI_helper.gstate):
une simple demo pour demontrer l'apparence et les differente capabilite

menu_list(state: GUI_helper.gstate, *args):
Cree un menu e partir des arguments fournis et l'affiche dans l'etat global.

Args:
state (gstate): L'etat global representant l'etat de l'application.
*args: Liste d'arguments variables contenant des fonctions.

Raises:
ValueError: Si l'un des arguments dans `args` n'est pas une fonction.

Example:
state = gstate(root, grid=grid)
menu_list(state, function1, function2, function3)

menu_main(state: GUI_helper.gstate):
Genere un menu principal e partir de l'objet `state` de type `gstate`.

Args:
state (gstate): L'objet `state` representant l'etat global.
```

Raises:

ValueError: Si l'objet `state` fourni n'est pas de type `gstate`.

menu_main_make(*args):

Creates a custom `menu_main` function that generates a menu based on the provided arguments.

Args:

*args: Variable length argument list of callable objects.

Raises:

ValueError: If any argument in `args` is not callable.

Returns:

function: The generated `menu_main` function.

menu_option(state: GUI_helper.gstate):

le menu de base pour les option renvoie a API.option

pretty_error(ex, exclude=['dispatch']):

quick_ask(txt: str):

Pose une question e l'utilisateur qui necessite une reponse rapide (oui ou non).

Args:

txt (str): Le texte de la question.

Returns:

int: 1 si la reponse de l'utilisateur est "y" (oui), 0 si la reponse est "n" (non).

Example:

```
answer = quick_ask("Do you want to continue? (y/n): ")
```

quick_dir(obj, _global=None, _local=None):

Affiche les attributs et les methodes d'un objet donne, ainsi que leur valeur si l'objet est appelable.

Permet ensuite e l'utilisateur d'evaluer des expressions Python e l'aide de la fonction `quick_eval`.

Args:

obj: L'objet dont les attributs et les methodes doivent etre affiches.

_global (dict, optionnel): Le dictionnaire global e utiliser pour l'evaluation des expressions dans `quick_eval`.

_local (dict, optionnel): Le dictionnaire local e utiliser pour l'evaluation des expressions dans `quick_eval`.

Returns:

None

Example:

```
quick_dir(obj, globals(), locals())
```

quick_eval(_global=None, _local=None):

evalue et affiche le resultat d'une expression Python saisie par l'utilisateur.

Args:

_global (dict, optionnel): Le dictionnaire global e utiliser pour l'evaluation de l'expression.

_local (dict, optionnel): Le dictionnaire local e utiliser pour l'evaluation de l'expression.

Returns:

None

Example:

```
quick_eval(globals(), locals())
```

textU(txt: str):

Utilise le dossier "language" pour traduire le texte en utilisant des fichiers de traduction.

chaque text est associer un numeros et ;

Chaque fichier de traduction est une liste de numero correspondant e chaque text traduit.

Args:

txt (str): Le texte e traduire.

Returns:

str: Le texte traduit s'il existe une traduction correspondante dans le fichier de traduction.

Si aucune traduction n'est disponible, le texte d'origine est renvoye.

Raises:

ValueError: Si le parametre `txt` n'est pas une chaene de caracteres.

Example:

```

translated_text = textU("Hello")
```

try_find_module(module_name, nickname=None, exit_if_not_found=True, ask=True):

Tente de trouver et importer un module specifie par son nom.

Args:

module_name (str): Le nom du module e trouver et importer.

nickname (str, optionnel): Le surnom e utiliser pour le module importe.

exit_if_not_found (bool, optionnel): Indique s'il faut quitter le programme si le module n'est pas trouve.

ask (bool, optionnel): Indique s'il faut demander une confirmation e l'utilisateur avant d'importer le module.

Returns:

int: 1 si le module est trouve et importe avec succes, 0 sinon.

Raises:

None

Exemple:

```

try_find_module("math", "m", exit_if_not_found=True, ask=True)
```

- Implémentation

Le minimum est démontrer avec default_GUI.py
GUI_launcher va scanner tout les fichier GUI_*.py pour en faire un menu
GUI_morpion implémente a lui seul l’interface graphique pour le morpion
GUI_echecs impelemtn la partie graphique pour le jeux d’échecs [vide]

4.Utilitaire

- server

Ce fichier est une démonstration comment ses algorithmes peuvent être utiliser pour faire tourner un serveur pour un site d'échecs par exemple

> (NOTE 1.3 : pour un serveur il est assumer que l'on joue au échecs mais cela peut être adapter pour n'importe quelle jeux avec une méthode de sérialisation)

- gen_doc

Outils pour scanner tout les fichier donner
importe les fichier
donne dans un dossier -o (ou stdout avec -p) toute les functions du fichier avec leurs signature et leur docstring

- visual

Petit programme graphique pour visualiser l'arbre de choix de algorithm.py
peut facilement être transformer pour n'importe quelle donné

> (NOTE 1.5 : le format pour visual_DATA.py est DATA={} avec chaque clé du dict sera un nœud ; la valeur devrai être tuple(valeur , est_gagner , {sous_arbre}))

il y a aussi des fichier .bat comme raccourcie pour directement éxecuter le programme