JUNIA
2 rue Norbert Ségard
59014 Lille cedex

Semestre 2

Projet Sprouts

M1

Pour la validation du de l'unité d'enseignement : Projet d'application
Professeur référent : BAUDEL Manon

DANEL Théo
DE POORTER Maxence
FORDELONE Marco
PRUVOT Quentin
STORDEUR Théo
VANGHELUWE Théo

Fonctionnement Général

1/ Backend Django (**Sprouts/** et **Game/**)

Utilisation du framework Django pour ces différentes fonctionnalités:

Fonctionnalités:

- Authentification (**Game/forms.py** , **Game/views.py**)
- Modèle de donnée de la BDD SQLite (via **Game/models.py**)
- API REST (via **Game/views.py**), utilisé pour la communication avec le frontend react
- Différents scripts de jeu (via **Game/utils**)

Structure de la BDD SQLite:

Voici les tableaux importants pour comprendre l'utilisation de la BDD.

- **auth_user :** login/register
- **Game_game_players :** sauvegarde de l'id d'une partie et des joueurs concernés (une game produit deux entités dans ce tableau avec deux fois la même game_id mais deux user_id différents)
- **Game_game :** sauvegarde d'une partie afin de pouvoir redessiner l'historique des coups joués

2/ Frontend React (**frontend/**)

Utilisation du framework React et stylisation avec Tailwind.

Structure:

- **src/components/** : éléments UI réutilisables
- **src/pages/** : pages du site
- **src/assets/** : images/logo

src/components/

- **game/** : un ensemble de scripts pour le fonctionnement du jeu

- **online/** : un ensemble de scripts spécifiquement pour le jeu en ligne
- **replay/** : script de replay pour GameSummaryPage
- **Footer** : footer personnalisé déployé sur toutes les pages
- **Header** : header personnalisé déployé sur toutes les pages
- **MoveHistory** : simple script pour AIGamePage

src/pages/

- **AIGamePage** : page de jeu en local contre IA
- **GameSummaryPage** : rejoue les coups d'une partie (id de la partie spécifiée dans l'url)
- **HistoricPage** : page répertoire de l'historique des parties d'un utilisateur
- **HomePage** : page d'accueil
- **LegalPage** : page des mentions légales du site/projet
- **MenuPage** : menu de jeu
- **MenuProfilPage** : menu du profil
- **OnlinePage** : page de jeu en ligne joueur contre joueur
- **PVEGamePage** : page de jeu en local joueur contre joueur
- **RulesPage** : page des règles
- **TechnicPage** : page depuis laquelle accéder à ce document, intégration PDF

Outils de chaîne de caractères

PVEUtils.js

isFrontiereInRegion = (frontiere, region, curveMap, points) => {}

Renvoie vrai/faux en fonction de si une frontière est dans une région en prenant en argument:

- La frontière concernée,
- La région concernée,
- La liste des courbes associées,
- La liste des points.

isPointInPolygon = (point, polygon) => {}

Renvoie vrai/faux en fonction de si un point est dans un polygone en prenant en argument:

- Le point concerné,
- Le polygone concerné

Cette fonction est appelée dans **isFrontiereInRegion.**

updateCurveMap = (curveMap, startPoint, endPoint, newCurve) => {}

Renvoie la liste des courbes associées en prenant en argument :

- La liste des courbes associées actuelles,
- Le point de départ,
- Le point de fin,
- La courbe créée.

generateInitialGraphString = (points) => {}

Renvoie la première chaîne de caractères à l'initialisation du jeu en prenant en argument:

- La liste des points.

generateGraphString = (startPoint, addedPoint, endPoint, currentGraphString, curveMap, points) => {}

Renvoi la chaîne de caractère modifié en prenant en argument :

- Point de départ,
- Point ajouté sur la courbe,
- Point de fin,
- La chaîne de caractère actuel,
- La liste des courbes associé,
- La liste des points.

move_verification.py

parse_boundaries(chain: str) -> list[list[str]]

Découpe la chaîne de caractères en **frontières**, chacune étant une liste de sommets.
Les frontières sont séparées par **.** ou **}**.

Exemple : "AL.BF}MC.}!" → [['A', 'L'], ['B', 'F'], ['M', 'C']]

parse_regions(chain: str) -> list[list[str]]

Découpe la chaîne en **régions**.
Chaque région contient une **liste unique de sommets** (pas de doublons), construite entre deux symboles **}**.

Exemple : "AB.GF}AC.}!" → [['A', 'B', 'G', 'F'], ['A', 'C']]

get_vertex_degrees(chain: str) -> dict[str, int]

Retourne un dictionnaire associant à chaque sommet son **degré**, calculé à partir des frontières.
Si un sommet apparaît une seule fois dans une frontière de taille 1 → degré = 0.

Exemple : {'A': 2, 'B': 1, 'C': 0}

playable_vertices(chain: str) -> list[str]

Renvoie la liste des sommets jouables, c'est-à-dire ayant un **degré < 3**.

is_valid_move(old_chain: str, new_chain: str) -> bool

Vérifie si la transformation entre **old_chain** et **new_chain** correspond à un **coup valide** selon les règles de Sprouts :

- Un seul nouveau sommet est ajouté
- Ce sommet a un degré 2
- La somme des degrés a augmenté de 4
- Aucun sommet n'a un degré > 3
- Le nouveau sommet est connecté à **exactement deux anciens sommets**
- Si les sommets sont différents, ils doivent appartenir à la **même région**
- Si les **régions** créés par le coup sont correctement créés

move_generator.py

generate_possible_moves(chain: str) -> list[tuple[str, str]]

Génère tous les **coups possibles** à partir d'une chaîne :

- entre **deux sommets de degré ≤ 2** dans une même région
- **self-loop** sur un sommet de degré ≤ 1

Retour : Liste de tuples **(v1, v2)** représentant les sommets à connecter

choose_move(chain: str) -> tuple[str, str] | None

Choisit **aléatoirement** un coup parmi ceux générés par **generate_possible_moves**.
Retourne **None** si aucun coup n'est possible.

move_over.py

is_game_over(chain: str) -> bool

Retourne **True** si **aucun coup n'est encore possible**, **False** sinon.

Un coup est encore possible dans une région si :

- il existe **au moins 1 sommet** de degré 0 ou 1 → self-loop possible

- ou **au moins 2 sommets** de degré ≤ 2 → coup classique possible