

# Projet INFO3 S5 : ipolytech\_SI3\_algoprogr

Mikaël FOURIER, Jean LABAT, Vincent BONNEVALLE

# Table des matières

<b>Introduction</b>	<b>2</b>
<b>1 Architecture</b>	<b>3</b>
1.1 Stockage des données . . . . .	3
1.2 Parseur . . . . .	3
1.2.1 Structure générale du parseur . . . . .	3
1.2.2 Cas de parse_init et de parse_state . . . . .	3
1.2.3 Test . . . . .	4
1.3 Structure de données . . . . .	4
1.3.1 Introduction . . . . .	4
1.3.2 Objet Match . . . . .	4
1.3.3 Objet Cell . . . . .	4
1.3.4 Objet Movement . . . . .	4
<b>2 Stratégie</b>	<b>6</b>
2.1 Stratégies de base . . . . .	6
2.2 _less_worse_strat . . . . .	6
2.3 strat4 . . . . .	6
2.4 strat5 . . . . .	6
2.4.1 Fonctions utilisées dans la stratégie . . . . .	6
2.4.2 Fonctionnement de la stratégie . . . . .	7
2.5 strat6 . . . . .	7
<b>3 Répartition des tâches</b>	<b>8</b>
<b>4 Problèmes recontré</b>	<b>9</b>
<b>Conclusion</b>	<b>10</b>

# Introduction

Le jeu Little Stars For Little Wars est un jeu de stratégie en temps réel dans lequel chaque joueur a pour but de contrôler des planètes en gérant le déplacement de ses unités. Le but de ce projet est de réaliser une intelligence artificielle capable de jouer à ce jeu, et qui soit la meilleure possible. Pour servir cet objectif, nous avons donc dû étudier les meilleures façons de représenter le terrain ainsi que les unités dans une architecture adéquate, et nous avons également élaboré une stratégie d'action pour notre intelligence artificielle.

# Chapitre 1

## Architecture

### 1.1 Stockage des données

Comme le bot consiste en plusieurs fonctions appelées par le client, il a fallu choisir un moyen de conserver les données. Nous avons choisi d'utiliser deux variables globales : `UUID` et `MATCHES`. `UUID` contient l'uid du joueur. Normalement, on n'y accède qu'en lecture après initialisation. `MATCHES` est un dictionnaire associant un id de match à un objet `Match` (cf ci-dessous) , ceci afin de permettre à plusieurs matches de se dérouler en parallèle.

La boucle principale de jeu a été simplifiée au maximum, la complexité étant distribuée entre les autres modules. On commence par récupérer les changements sur le plateau (message `STATE`) qu'on parse. Ensuite, on récupère l'objet `Match` correspondant au match concerné, on le met à jour avec les nouvelles données puis on lui demande de calculer la stratégie. Finalement, on encode cette stratégie et on l'envoie au serveur. Chaque module est bien séparé : l'ajout des messages `GAMEOVER` et `ENDOFGAME` n'ont nécessités des changements que dans `protocol.py`, ainsi que de légers changements haut-niveau dans `lolipooo.py`. Si le format des ordres à envoyer changeait, le module `strategy.py` ne serait probablement pas modifié.

### 1.2 Parseur

#### 1.2.1 Structure générale du parseur

Chaque type de message (`INIT`, `STATE`,...) est parsé par une fonction nommée `'parse_<type>'`. Toutes ces fonctions sauf `parse_register` utilisent des regex définies en constantes globales, avec des groupes nommés pour les données intéressantes. L'utilisation de la méthode `'groupdict'` des regex permet de récupérer le résultat parsé sous forme de dictionnaire associant le groupe nommé ) la valeur. Le reste des fonctions consiste en divers bidouillages pour transformer certains str en int par exemple.

Nous avons également mis en place un log simple utilisant le module `'logging'`. Il consiste à afficher le message reçu puis le message parsé. Cela permet en cas de problème que les parsing a été exécuté correctement.

Dans la boucle principale, le bot peut recevoir des messages `STATE`, `ENDOFGAME` ou `GAMEOVER`. La fonction `'parse_message'` est utilisée pour appeler la bonne fonction de parsing. Le type de message est simplement reconnu à l'aide de la méthode `str.startswith`. Une condition `else` permet de gérer le cas où un nouveau type de message serait ajouté sans que le code du bot ne soit mis à jour. Le tout est enrobé dans un `try/catch` pour attraper les erreurs sans faire crasher le client.

#### 1.2.2 Cas de `parse_init` et de `parse_state`

Les messages de type `INIT` et `STATE` sont particuliers à traiter car ils comportent des listes de taille variable. Les regex python ne permettant pas d'avoir un nombre variable de groupes, nous avons dû diviser les listes et parser individuellement chaque item. Par exemple, dans `'parse_state'`, la string

des cellules est récupérée par la regex principale en un bloc. Celui-ci est ensuite coupé au niveau des virgules, puis chaque fragment est parsé à l'aide de la regex `REGEX_CELL_STATE`. La liste python résultante est ensuite réintroduite dans le dictionnaire parsé à la place de la string.

Le champ `CELLS` du message `INIT` est plus délicat à parser, car il comporte une virgule au sein même des items. La string est donc coupée au niveau de `"I,"`, puis `"I"` est rajouté à la fin de chaque cellule avant parsing. Ce n'est pas très élégant, mais ça fonctionne.

### 1.2.3 Test

Pour chaque type de message, un message d'exemple ainsi que le dictionnaire parsé sont inclus dans le fichier source. Nous avons utilisé la fonctionnalité `doctest` de python pour intégrer ces tests dans la docstring des fonctions. Les docstrings sont assez limitées, le résultat attendu est récupéré par python sous forme de texte. Comme les dictionnaires ne sont pas déterministes pour l'ordre des clés, nous avons dû comparer le retour de la fonction avec le résultat attendu et vérifier que cette comparaison est vraie, au lieu de simplement mettre le dictionnaire en résultat.

## 1.3 Structure de données

### 1.3.1 Introduction

Nous avons choisi d'utiliser des classes pour l'encapsulation qu'elles proposent. Voir le fichier `loli-poop.py` pour la simplicité de la boucle principale.

### 1.3.2 Objet Match

L'objet `Match` contient les informations relatives à un match. Celles-ci sont un mélange de données statiques, comme les cellules et les liaisons entre elles, et de données dynamiques, comme le nombre d'unités par cellule. De plus, chaque objet `Match` contient une référence à la fonction calculant la stratégie. Cela permet par exemple de choisir la stratégie à appliquer en fonction de l'organisation des cellules.

Les données basiques, comprenant l'id du match et du joueur, la vitesse de jeu et le nombre de joueurs, sont stockées sous leur forme d'origine, à savoir `int` ou `str`. Le membre `cells` est plus complexe, il est stocké sous forme d'un tableau associant un id de cellule à un objet `Cell` (cf ci-dessous). Ce tableau permet de retrouver facilement l'objet `Cell` à l'aide de son identifiant.

Cet objet contient peu de logique. L'initialisateur se contente d'initialiser les membres, les détails d'instanciation des cellules étant délégué à la classe `Cell`. La mise à jour dynamique se fait simplement en parcourant les cellules et en appelant la méthode `update` correspondante. Quand au calcul de la stratégie, il est délégué à la fonction de stratégie choisie à l'initialisation.

### 1.3.3 Objet Cell

Un objet `Cell` est instancié pour chaque cellule du plateau. Il centralise les données relatives à cette cellule. Il contient des données simples, comme la quantité maximale d'unités offensives, et des données plus complexes comme la liste des mouvements à destination de la cellule ou un dictionnaire associant l'id des cellules voisines à la distance les séparant de la cellule courante.

La méthode `update` permet de mettre à jour les données dynamiques à partir des données parsées dans la boucle principale.

### 1.3.4 Objet Movement

L'objet `Movement` modélise un mouvement d'unités vers une cellule. Il stocke le nombre d'unités en déplacement et leur propriétaire, la cellule de départ ainsi que le temps restant avant arrivée des unités.

Nous avons choisi de stocker les déplacement dans la cellule d'arrivée car c'est ce qui nous a paru être le plus utile.

# Chapitre 2

## Stratégie

Pour choisir la stratégie, il suffit de changer la valeur de la variable `strategy_name`

### 2.1 Stratégies de base

Les stratégies `_strat_base` et `_strat_base2` sont des stratégies basique : on attaque la cellule ennemie adjacente la plus faible ou on aide la cellule adjacente la plus faible s'il n'y a pas de cellule ennemie adjacente

### 2.2 `_less_worse_strat`

Bien qu'assez basique, cette stratégie est un peu plus complexe que les stratégies de base. Le principe de cette stratégies est de partir des cellules ayant une cellule ennemie adjacente, de leurs faire attaquer la cellule ennemie adjacente la plus faible. Puis les cellules adjacentes aux cellules auxquelles on vient de donner des ordres aident ces cellules. Puis on recommence jusqu'à que toutes les cellules aient des ordres

### 2.3 `strat4`

Le principe de cette stratégie est de lister les cellules par propriétaire et par danger (calculé par `unit_needed`). On parcourt la moitié la moins en danger de nos cellules puis on liste les actions possible (grâce à `possible_action`) pour ces cellules. Puis on envoie un nombre d'unité correspondant au maximum entre 75% des unités de la cellule source et le nombre d'unité attendu par la cellule cible.

### 2.4 `strat5`

#### 2.4.1 Fonctions utilisées dans la stratégie

Cette strategie repose sur la fonction `cell_value` qui permet de calculer "l'importance" d'une cellule pour de determiner quelles sont les cellules que nous souhaitons capturer en priorité. Cette fonction dépend de 2 parametres importants : -Le premier est éidement la production de la cellule. -Le second est la distance de la cellule à la cellule ennemie la plus proche en nombre de saut. Cette distance permet de determiner si la cellule est proche des ennemis ou pas, et donc si elle risque d'avoir besoin d'unités ou pas. Cela permet d'envoyer nos unités en direction du front en priorité. La deuxième fonction importante pour cette stratégie est la fonction `unit_needed`, qui calcule un "danger" pour une cellule, en fonction du nombre de cellules ennemies qu'il y a autour de la cellule, et des déplacements en direction de celle-ci. Ce taux de danger permet autant de savoir si une cellule ennemie est vulnérable que de savoir si une celue alliée elle permet donc de savoir le nombre de renfort dont pourrait avoir besoin une cellule, mais aussi de determiner si on peut attaquer une cellule ennemie.

### 2.4.2 Fonctionnement de la stratégie

En se basant sur l'importance des cellules définie ci-dessus, on crée donc une liste des cellules triée par ordre d'importance. On regarde ensuite chacune de nos cellules, et pour chacune d'entre elle, on va envoyer des unités aux cellules adjacentes qui en ont besoin en envoyant en priorité aux cellules les plus importantes. On sépare ici les cas selon le propriétaire de chaque cellule adjacente, car le nombre d'unités à envoyer peut varier selon le propriétaire. Il a fallu trouver un moyen d'actualiser le nombre d'unités dont une cellule avait besoin avant que les ordres ne soient transmis et qu'un nouveau state ne soit récupéré, c'est pourquoi nous avons ajouté l'attribut `unit_needed` à la classe cellule. Nous définissons donc le nombre au début de la boucle en l'actualisant pour toutes les cellules puis nous l'actualisons au fur et à mesure que les ordres sont donnés.

## 2.5 strat6

Tout d'abord, cette stratégie consiste à établir une table de routage pour chaque cellule, cette table de routage consiste en une liste de ligne contenant la cellule cible, la cellule vers laquelle aller vers la cible, ainsi que la distance à parcourir par ce chemin. La table de routage est créée une seule fois.

Ensuite, on sélectionne les cellules ennemis ou neutre pouvant être attaquées. Puis on les répartit entre nos cellules cibles.

Enfin, les cellules attaquent leurs cibles en prenant la route donnée par leurs tables de routage. Le calcul du nombre d'unités à envoyer se fait ainsi :

- Si la cible est neutre
  - Si les cellules adjacentes alliées ont assez d'unités pour conquérir la cellule et la garder on attaque avec la moitié des unités
  - Si sans nous, les cellules adjacentes alliées n'ont pas assez, on envoie les  $\frac{3}{4}$  des unités de la cellule
  - Sinon on envoie  $\frac{1}{4}$  des unités
- Si la cellule est alliée
  - Si on a pas assez pour l'aider, on envoie  $\frac{3}{4}$  de nos unités
  - Sinon on envoie le nombre d'unités dont elle a besoin
- Si la cellule est ennemie
  - Si on a un rapport nombre d'unités offensives / vitesse de production est le double de la cellule ennemie : on attaque en force
  - Si le rapport est supérieur à 1 et qu'on est aidé par les cellules alliées adjacentes, on fait de même.
  - Sinon on envoie  $\frac{1}{4}$



## Chapitre 3

# Répartition des tâches

Nous nous sommes réparti les tâches en deux sous groupes. Une personne (Mikaël) s'est occupé de la communication avec le serveur. Deux personnes (Jean et Vincent) se sont occupé des stratégies. Les stratégies de bases (`_strat_base`, `_strat_base2` ainsi que `_less_worse_strat`) on été créés à deux. Pour les autres stratégies, nous avons décidé de nous séparer les tâches (entre Jean et Vincent) pour explorer différentes options.

## Chapitre 4

# Problèmes rencontrés

Nous n'avons pas réussi à transmettre les ordres au serveur qui ne les recevais pas. Nous n'avons donc pas pu tester convenablement nos I.A.

# Conclusion

La programmation de cette IA a été une expérience très enrichissante, elle nous a permis de découvrir le travail en équipe et la répartition des tâches, il nous a également permis d'approfondir nos connaissances en python.

En effet, nous avons dû nous répartir les tâches de façon équitable : une personne ayant trop ou trop peu de travail aurait ralenti tout le groupe. De plus, le projet nous a forcé à travailler collectivement, à discuter souvent de l'avancement du projet et ainsi travailler en équipe et non chacun de notre côté.

Enfin, ce projet nous a permis de découvrir le gestionnaire de version git.