README & EXPLICATIONS

Lien GitLab Unistra

git@git.unistra.fr:cyrille.muller/Projet_SD.git

Explications globales

Nous avons programmé le programme Java ci-joint en ayant en tête une palteforme de simulation de partie de jeu, et l'exploitation de logs après la dite simulation. Pour ce faire, nous avons deux classes principales (contenant les fonctions main) :

- La classe *Client*, utilisée par la personne voulant faire la simulation. Elle est chargée de lire les paramètres de la partie désirée, de la lancer sur les serveurs et d'en récupérer les logs quand cette dernière est terminée. Le client peut modifier tous les paramètres de la partie , via un fichier de configuration dont le squelette est généré par un script. (Détails dans la section « Lancement »). En plus de cela, le Client peut choisir précisément sur quel serveur lancer quel acteur. Il lui suffit de renseigner le champs « adresse » dans la configuration. Pour faciliter les test locaux, un serveur peut héberger plusieurs Acteurs de la partie.

-La classe *Serveur*. Cette dernière met à disposition du client un objet « Setup ». Cet objet permet au client de créer joueurs, producteurs et coordinateurs sur le serveur. Il lui permet également de vérifier le déroulement de la partie, et de récupérer les logs. Il est possible de créer plusieurs Serveurs sur la même machine, avec des nom d'objet Setup différents. On peut donc, sur une même machine, crée deux objets setup, sur chacun desquels il est possible de créer plusieurs Acteurs.

Le Client se sert donc des objets Setup pour créer les différents acteurs nécessaires à la partie. Le processus est automatique. Comme dit précédemment, différent objets Setup peuvent être mis à disposition d'un Client sur une seule et même machine, et plusieurs Acteurs peuvent être lancés sur un même objet Setup. Le client doit donc préciser, en plus de l'adresse de la machine désirée, le nom de l'objet Setup sur lequel l'acteur devra être lancé. Ceci permet de choisir avec précision sur quel machine placé un joueur humain, par exemple. On admet que le client possède un fichier serveurs.csv contenant l'adresse des serveurs et le nom des objets setup disponibles.

La première «communication RMI » se fait donc entre le client et les serveurs, via les objets « Setup ». Lorsque tout est mis en place, la partie se lance.
A partir de ce moment, la communication n'est faite qu'entre serveurs. Chaque serveurs possèdent des Acteurs pouvant joindre d'autres Acteurs afin d'en prendre les ressources. La partie prend fin suivant la règle décidée dans le fichier de configuration (détails des règles plus bas). A chaque action, les acteurs notent leur état actuel (action réalisé, ressources possédées, etc). A la fin de la partie, les objets Setup permettent au client de récupérer ces logs, et des scripts R permettent de les analyser. Le tour par tour a également été implémenté. Il faut pour cela préciser que l'on veut un

coordinateur, et préciser que la partie est en tour par tour dans le fichier de configuration. De même pour les joueurs humains.

Pour résumer :

Le client paramètre la partie dans un fichier de configuration généré par un script python (*creer_template.py)*. Ce dernier est lu, et la partie mise en place sur les différents serveurs mis à disposition via RMI. La partie se lance, et les Acteurs des objets Setup des serveurs communiquent entre eux via RMI pour faire avancer la partie. A la fin de cette dernière, le client récupère les logs.

Note : les makefiles se trouvent dans les fichiers src de src/Client et src/Serveur. Il possède une fonction « clean »

DETAIL DES CLASSES

On distingue deux types d'interfaces RMI :

-Celle pour mettre en place la partie (Setup)

-Celles nécessaires au déroulement de la partie (Joueurs, Producteurs, Coordinateurs)

Setup

Comme expliqué précédemment, ces objets crées sur les serveur et mis à disposition du client permettent de créer les acteurs nécessaires à la partie, de lancer cette dernière et d'en récupérer les logs. Au lancement de chaque serveur, on précise le nom du fichier setup que l'on va mettre à disposition. Le client possède un fichier .csv liant les setup à l'adresse de leur serveur.

Joueurs, Producteurs & Coordinateurs

Ces derniers descendent tous de Acteurs, une classe abstraite. Ce sont les Acteurs des parties. Ils existent sur les serveurs, et communiquent avec des Acteurs d'autres Serveurs.

Les Joueurs ont des objectifs, un Comportement, caractérisant sa manière de jouer, et ils possèdent des Ressources. Un joueur peut être humain ou non. Si il l'est, lors de son tour, un prompt lui demandera quelles actions il choisit d'effectuer.

Les Producteurs possèdent et produisent des Ressources à une certaine intervalle (renseigné en ms).

Les Coordinateurs régulent la partie si cette dernière est en tour par tour.

Les Summary

Trois classes possèdent un nom finissant par Summary. Leur but est simple. Afin de pouvoir communiquer avec les autres Acteurs, un Acteur a besoin de connaître leur nom, leur adresse, et selon certains cas, leur ressources (pour demander chez le bon producteur, par exemple). Les Summary résument les informations nécessaires à ces actions, et sont envoyé à chaque Acteurs de la partie pour permettre une bonne communication.

La Partie

La classe Partie contient toutes les informations de la partie, le Summary des acteurs, les règles, etc. C'est cette dernière qui est transmise aux Acteurs à leur création. Elle contient une classe Regle, étant simplement là pour structurer les différentes options relative à la partie (punition en cas de vol, etc).

Les Ressources

Les ressources sont l'objectifs de chaque joueurs. Elles sont au nombre de trois. Petrole, Or, et Fer. Elles sont toutes produites différement. La rapidité de production, par contre, est propre au producteur.

Les Comportements

Les comportements guident les actions des joueurs. On en distingue trois : Gentil, Average, et Agressif. Agressif va souvent voler d'autre joueurs, et applique une méthode « greedy » : il va soit prendre le maximum de ce dont il a le plus besoin, soit prendre le maximum de la ressource la plus rare. Average va voler assez rarement, et prendre le maximum de la ressource dont il a le plus besoin. Gentil ne vole jamais, et prend la Ressource dont il a le plus besoin. Cependant, il ne prend pas le maximum ; il prend le nombre d'exemplaires disponibles divisé par le nombre de joueurs dans la Partie.

Lancement

Le lancement du programme se fait en plusieurs étapes. Premièrement, le client doit lancer le script create_template.py avec les arguments suivants :

-i [nombre d'ias]

-j [nombre de joueurs]

-p [nombre de producteurs]

-c [nombre de coordinateurs]

-o [fichier output de la configuration]

-f [fichiers contenant les informations des serveurs]

Il n’est pas nécessaire d’utiliser chaque arguments. Ils possèden des valeurs par défaut (0 pour le nombre d’acteur, *conf_partie* pour -o, et *serveurs.csv* pour -f.)

Cela fait, un fichier template sera crée à l’endroit du script (dans le dossier Client). Le client doit compléter ce dernier suivant les paramètres de partie qu’il désire. Néanmoins, le fichier tel qu’il est généré est déjà opérationnel. Chaque acteurs aura été donné un nom, et une adresse. Les adresses données aux acteurs ont été choisies aléatoirement dans le fichiers contenant les informations de chaque serveurs. Si l’on souhaite quand même changer certaines informations (notamment les objectifs des joueurs), il est impératif de suivre exactement la syntaxe utilisée (pas d’espace en plus, etc).

Grâce à cette méthode, il est donc possible de personnaliser, pour chaque partie, son nombre d’acteurs, le serveur sur lequel les acteurs doivent être lancé, leur objectifs, et les paramètres de la partie.

Dans le type « Partie » du fichier de configuration , MAXTAKE représente le maximum demandable en terme de ressources par un joueur pendant son action. VICTOIRE représente le type de règle utilisée pour définir la victoire (First , où un seul joueur doit terminer, ou All, où tous les joueurs doivent terminer.). PUNVOL est la quantité de ressources qu’un joueur se voit retirer en cas d’échec d’une tentative de vol, et TOURPTOUR définit si la partie se jouera au tour par tour ou non (false ou true).

Dans le type Producteur, on trouvera de quoi changer les ressources produites par chaque producteur, l’intervalle de temps entre chaque vague de production (en ms) , et le nombre de ressources au départ de chaque producteur.

Le type Joueur ne possède pas beaucoup d’argument, car il référence un joueur humain. Son comportement doit toujours etre calibré sur « humain »

Le type Coordinateur ne possède pas beaucoup d’arguments non plus, ne servant qu’à réguler le tour par tour.

Enfin, le type IA permet de régler les objectfifs de chaque IA, mais surtout leur comportement. Comme dit précédemment , il existe trois comportements : « Gentil », « Agressif », et « Average ».

Les serveurs doivent avoir été lancés, et sont lancés en renseignant le nom de l’objet setup.(Java Serveur nom_setup) Le fichier setup chez le Client doit quant à lui avoit été renseigné sous le format suivant.

adresse,nomsetup

adresse2,nomsetup2

Il faut avoir lancé rmiregistry au préalable. Une des forces de cette implémentation est que une fois les serveurs lancés, ils peuvent servir de « Cloud » indéfiniment. Il est possible d’enchaîner les parties sans se soucier des serveurs. Il est alors simple de faire un script lancant plusieurs fois le Client sur différentes configuration de partie, pour créer des logs conséquents simplement.

Si en cas d’erreur de configuration, ou dans un cas particulier, une partie ne se finit pas, il n’est pas nécessaire d’arrêter le serveur. Envoyer un SIGINT au client suffit. Ce dernier, avant de se couper,

va dire aux serveurs d'arrêter la partie, et tenter de récupérer des logs. Il suffit alors de relancer un client sur une nouvelle configuration.

La partie se déroulera toute seule après avoir lancé le client. Ce dernier se lance via « java Client [nom_fichier serveur] [nom fichier configuration partie]». Comme leur nom l'indique, ces fichiers sont respectivement ceux contenant les informations des setups, et les informations de configuration de la partie.

Bien que certains outputs sont visibles en temps réel pour témoigner du lancement de la partie, ils ne témoignent pas précisémment du déroulement de cette dernière. Ceci peut être étudié via les scripts R, permettant de tracer des graphes grâce aux logs des Parties terminées. Les logs de chaque partie sont classés dans une arborescence de cette forme : ./src/Client/Logs/NomPartie. Le nom de la partie est à renseigner dans la configuration. Dans ces Logs, on trouvera un log reprenant les informations de la partie, et un log par acteurs. Ces logs sont donc exploitables facilement via l'utilisation des scripts R fournis (exemple ci dessous).

Lancement pré-configuré :

Pour lancer une partie avec les configurations fournies :

rmiregistry &

Dans srs/Serveur/src :

make

Puis dans trois terminaux :

java Serveur stup

java Serveur stup2

java Serveur stup3

Et enfin dans src/Client/src

make

java Client ../serveurs.csv ../FICHIER_CONFIG

FICHIER_CONFIG = {conf_partie_tpt, conf_partie_normale} ;

Les serveurs n'ont pas besoin d'etre relancés entre différentes partie.

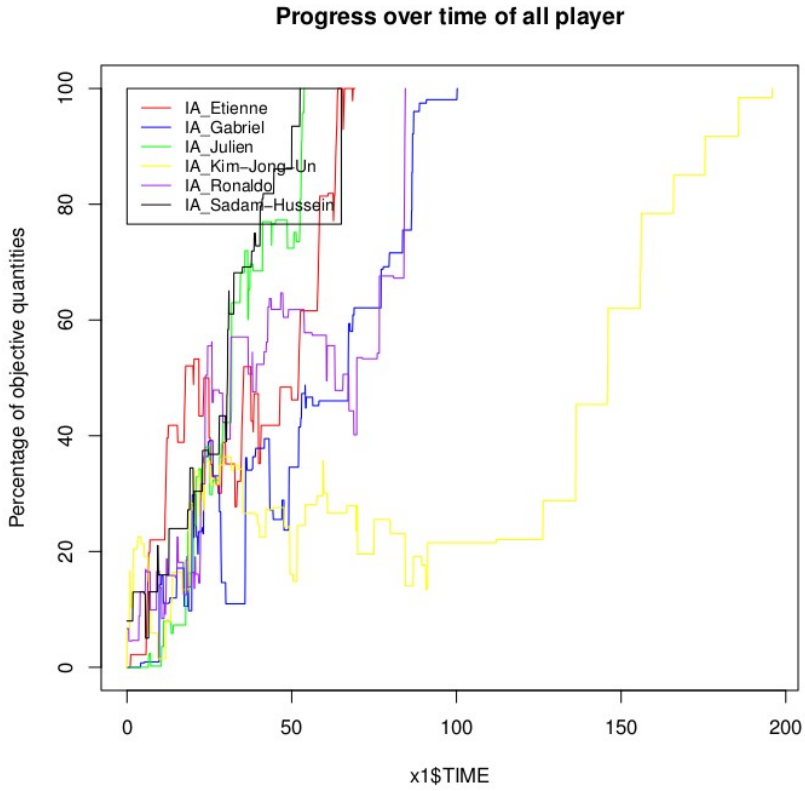
Pour pouvoir utiliser « ./make_plots NOM_PARTIE » il faut que les noms des joueurs commencent par JOUEUR, ceux des ia par IA et ceux des producteurs par PROD.

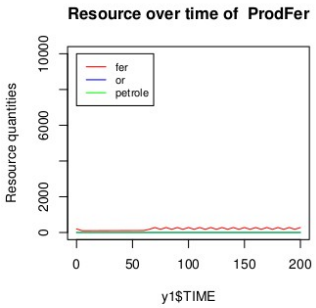
Exemple de Parties

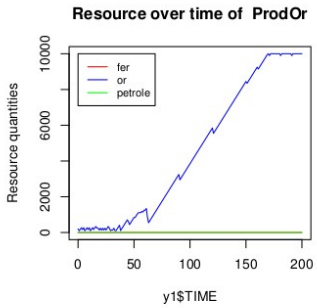
Les Parties sont visualisables à posteriori en utilisant « ./make_plot.sh NOM_PARTIE »

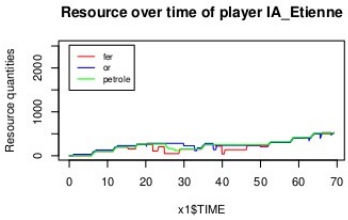
Voici la représentation d’une partie normale ou les personnalitées et objectifs sont uniformément répartis :

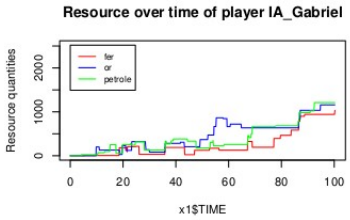
On peut y discerner une différence entre les joueurs et deviner lesquels sont altruistes ou égoïstes.

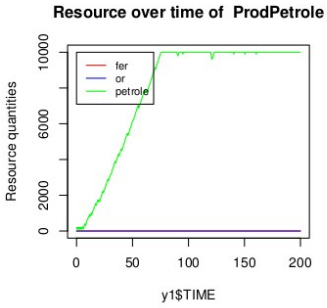


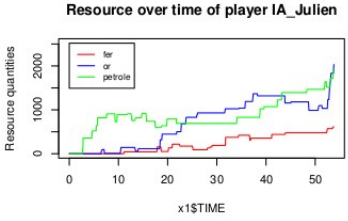


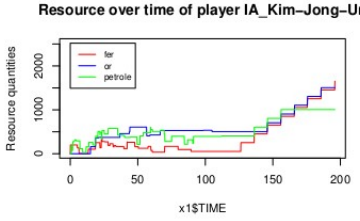


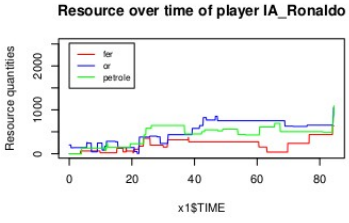


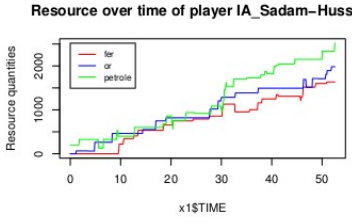




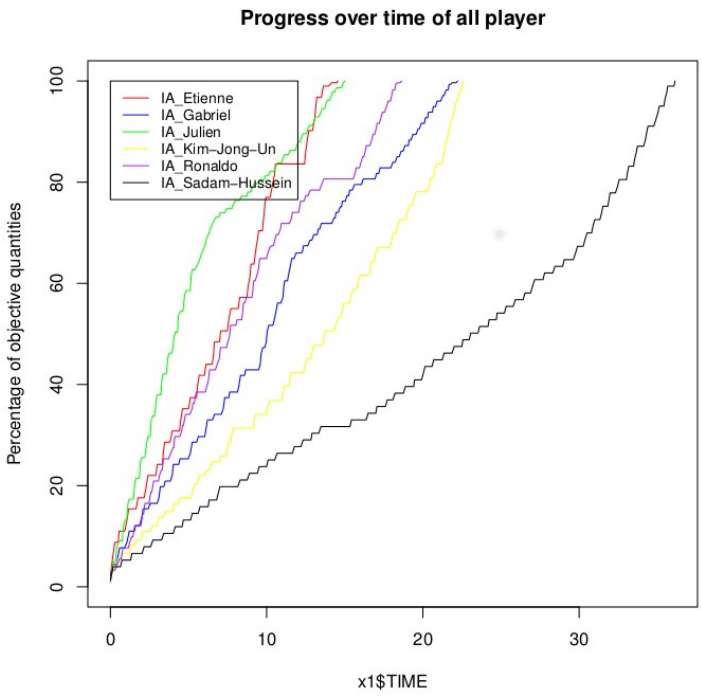


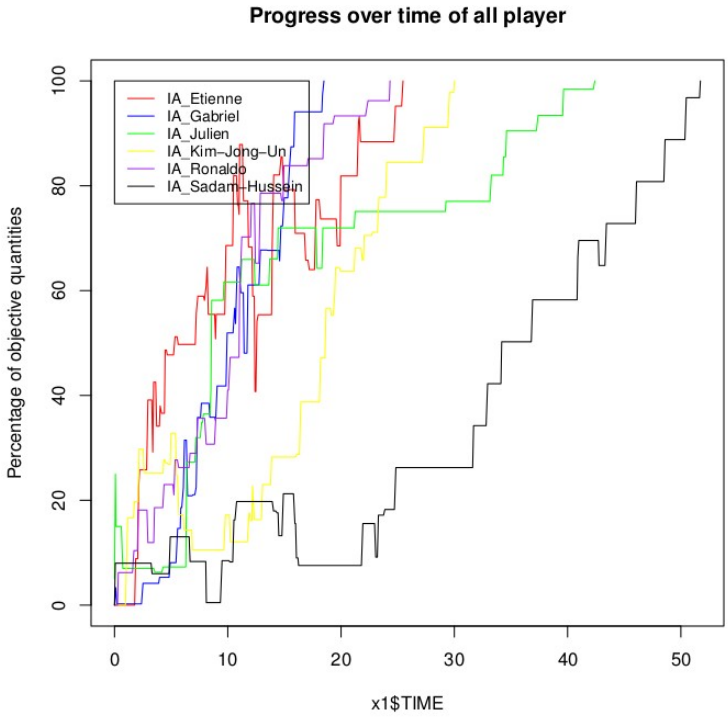


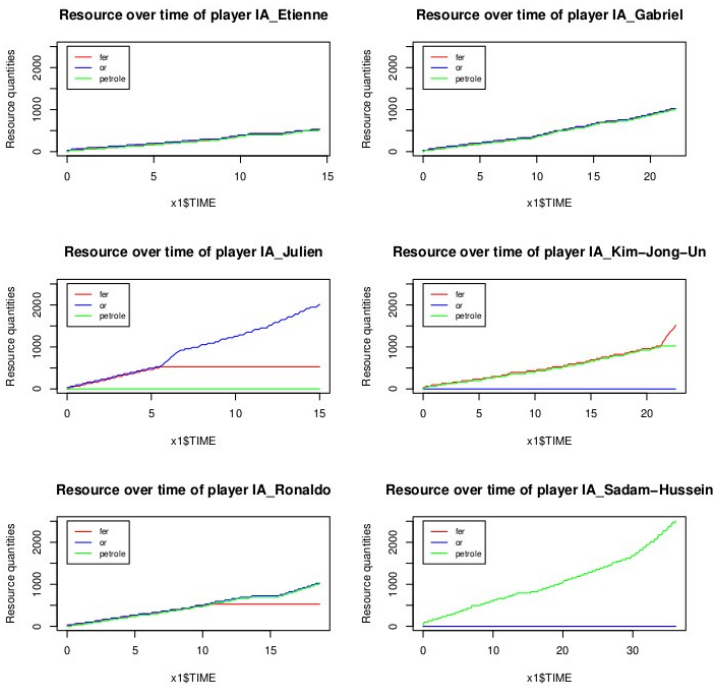


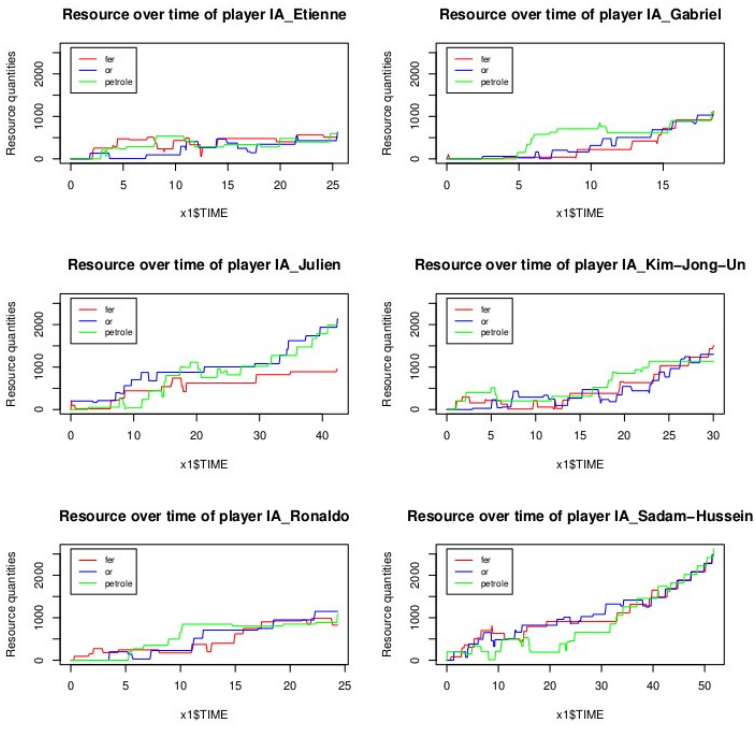


Ci-dessous deux parties dont la seule différence est la personnalitée de tous les joueurs. Dans le premier cas tout le monde est avare, dans le deuxieme tout le monde est altruiste.









On peut clairement voir que lorsqu’ils sont isolés les égoïstes profitent des autres, mais que lorsque tout le monde est égoïste les joueurs prennent quasiment deux fois plus de temps que s’ils étaient tous altruistes.