Rapport de Projet Tuteuré - PANDROIDE

Pas de jaloux, un jeu de partage équitable

Auteurs
Alexandre Bontems
Gualtiero Mottola
Hans Thirunavukarasu

Superviseurs
Nicolas Maudet
Aurélie Beynier

Université Pierre et Marie Curie, Paris 6, Département Informatique
4 place Jussieu 75252 Paris cedex 05, France

Dans le cadre du master informatique ANDROIDE de l'UPMC, un projet tuteuré doit être effectué par les étudiants et ce rapport en détaille les résultats. Le problème de partage équitable LEF, présenté en [**lef**], est étudié et un jeu puzzle en est dérivé. Pour répondre aux problématiques d'analyse de difficulté pour l'humain, des résolutions « à la main » sont observées et des outils sont développés pour tenter d'expliquer les ressentis. Le développement d'une application jeu pose également les problématiques liées à l'expérience utilisateur et de conception de tutoriel.

Table des matières

1. Introduction

Ce travail s'est effectué dans le cadre du projet tuteuré du Master ANDROIDE, proposé par la faculté des sciences de *Sorbonne Université*. Les principaux objectifs, donnés par l'équipe cliente (composée de Nicolas Maudet et Aurélie Beynier), étaient de développer un jeu basé sur le problème de satisfaction LEF et de pouvoir y proposer des niveaux de difficulté croissante. Pour répondre à ces besoins le projet s'est vu divisé en deux parties : l'analyse d'instances pour en évaluer la difficulté et, en parallèle, le développement d'une application permettant de jouer au jeu.

Définit dans [**lef**], le problème de satisfaction LEF est résumé ici ; n agents se trouvent liés par un réseau social dans lequel chaque agent peut percevoir un à deux autres voisins. Ils sont disposés en chaîne et sont liés à leurs voisins de gauche et de droite (voir **??**). C'est le seul type de réseau évoqué par ce projet mais tout autre type peut être envisagé pour analyse avec les outils proposés dans ce rapport. On dispose d'autant de biens indivisibles (ou objets) que d'agents. La résolution du problème consiste alors en la recherche d'une allocation équitable $\mathcal{A}$ d'un objet par agent. Une allocation est recevable si aucun agent n'éprouve de jalousie, c'est-à-dire si pour toute paire d'agents (a_i, a_j), voisins dans le réseau, la relation suivante est vérifiée : $\mathcal{A}(a_i) \succ_i \mathcal{A}(a_j)$. Comme on peut le voir dans la **??**, chaque agent est associé à une liste de préférences concernant tous les biens, triée de haut en bas. Par exemple, l'agent a_1 est définit par l'ordre de préférence

$o_2 \succ o_4 \succ o_1 \succ o_3$. Une solution possible de l'exemple est surlignée en jaune.

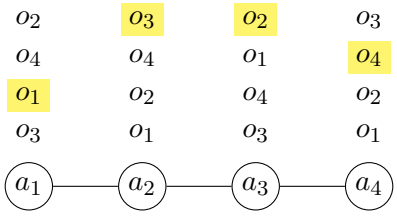


FIGURE 1 – Exemple d'instance étudiée du problème LEF

Ce sont les instances de cette variante du problème LEF qui ont été analysées au cours de ce projet. Seules les instances de trois à sept agents étaient concernées car celles de tailles inférieures à trois ne peuvent avoir qu'une solution et qu'une borne de taille était nécessaire à cause du support choisi pour le jeu. Afin de proposer des enjeux intéressants aux joueurs, le travail d'analyse s'est concentré sur l'obtention d'instances solvables et le développement d'outils permettant d'évaluer leur difficulté. Puisque la résolution au sein du jeu se fait « à la main » par un humain, obtenir les ressentis de difficulté des joueurs était primordial et l'application de jeu s'est révélée très utile pour récupérer ces informations. Il a ainsi été possible d'étudier les corrélations entre les caractéristiques d'une instance et la difficulté éprouvée par les joueurs. Ces résultats permettent d'intégrer de nouvelles instances dans le jeu et de déterminer un niveau de difficulté probable selon leurs caractéristiques.

Il a été décidé très tôt qu'une plateforme mobile se prêtait au mieux à ce type de jeu puzzle et l'application a donc été produite au sein de l'écosystème `Android`, permettant ainsi la distribution du jeu à un grand public. Un certain nombre de problématiques liées au développement d'application et d'interface ont été traitées, notamment la conception de tutoriel et d'une interface favorisant la compréhension et l'amusement.

Ce rapport détaille en premier lieu le processus d'analyse des instances, puis l'architecture logicielle ainsi que l'esthétique de l'application jeu en elle-même.

2. Analyse d'instances

Dans la partie théorique de ce projet, nous nous sommes attelés au calcul de métriques diverses permettant de résumer les caractéristiques d'une instance donnée. Puisque l'évaluation de la difficulté d'une instance se fait par rapport à une résolution humaine, plusieurs hypothèses ont été posées concernant les méthodes de résolution utilisées par un joueur. Elles dérivent directement d'observations réalisées grâce à l'application mobile développée dans le cadre de ce projet. Dans cette section, les différentes métriques conçues sont détaillées ainsi que les observations à leur source.

2.1. Solvabilité d'une instance

Afin d'obtenir des instances pertinentes pour notre analyse, il a d'abord été essentiel d'étudier leur solvabilité. Puisque le voisinage de chaque agent est connu, générer des instances comptant au moins une solution est relativement aisé. L'idée est de choisir une allocation aléatoire, c'est-à-dire l'indice pour chaque agent, dans leur liste de préférences respective, de l'objet qui leur sera alloué. On s'assure ensuite qu'un agent donné ne préfère pas les objets choisis pour ses voisins à sa propre allocation. Le pseudo-code suivant a été implémenté dans ce projet.

Un exemple d'instance générée se trouve en **??**. Les objets correspondants aux indices choisis en première étape sont encadrés et on se rend bien compte que les objets choisis pour les voisins sont bien placés dans les préférences de façon à ne pas générer de jalousie dans l'allocation.

```
Indices := []
Pour chaque agent a:
    % Il faut que les objets voisins ne soient pas préférés à l'indice
    % choisi donc on garde une place pour les agents en extrémités et
    % deux places pour les restants.
    Si a est le premier agent ou le dernier agent:
        Indices[a] := valeur aléatoire entre 1 et n-1
    Sinon:
        Indices[a] := valeur aléatoire entre 1 et n-2

Pour chaque agent a:
    Prefs[a] := []
    ValeursPossibles := {1, .., n}\(Indices[Voisins[a]] et Indices[a])

    Pour chaque indice i < Indices[a]:
        k := valeur aléatoire parmi ValeursPossibles
        Prefs[a, i] := k
        ValeursPossibles := ValeursPossibles\{k}

    Prefs[a, i] := Indices[a]
    ValeursPossibles := ValeursPossibles et Indices[Voisins[a]]

    Pour chaque indice i > Indices[a]:
        k := valeur aléatoire parmi ValeursPossibles
        Prefs[a, i] := k
        ValeursPossibles := ValeursPossibles\{k}
```

FIGURE 2 – Algorithme de génération d'instance solvable

1	3	1	4
3	4	2	3
2	2	3	0
4	1	4	2
a_1	a_2	a_3	a_4

TABLE 1 – Exemple d'instance générée

Chaque couleur correspond à un objet différent et les objets alloués aux voisins apparaissent en couleur.

2.2. Résolution par backtracking

Avec des instances résolvables, nous avons pu procéder à leur analyse et la première approche abordée a été de résoudre le problème à l'aide d'un algorithme de backtracking. En effet, il s'est rapidement montré évident qu'un processus similaire pouvait être utilisé comme méthode de résolution par un humain (**??**).

***Observation* 1.** *Un déroulement fréquemment observé est de commencer par choisir un premier agent pour lui affecter un objet (généralement parmi les extrémités car le voisinage est alors de taille* 1 *seulement et les contraintes sont par conséquent plus faciles à satisfaire). L'étape suivante est de choisir un objet pour cet agent et le plus facile est de commencer par l'objet préféré : plus un objet est apprécié par un agent, moins il est probable de laisser place à de la jalousie. On poursuit ensuite le processus d'allocation en choisissant un voisin de cet agent et en procédant de manière similaire de voisin en voisin (choix de l'objet le plus aimé parmi les restants). Lorsqu'un*

agent se montre jaloux, on revient sur le choix précédent. L'explication du problème au joueur peut cependant influer sur ce déroulement typique ; Par exemple si la visualisation des listes de préférences n'est pas bien comprise alors la procédure de choix des objets à affecter peut être altérée. Ce comportement atypique disparaît généralement si un joueur joue plusieurs fois.

C'est avec ces observations en tête que l'algorithme de backtracking a été conçu. Depuis un agent quelconque, l'algorithme tente d'affecter les objets préférés en premier tout en vérifiant les contraintes, et procède ainsi de voisin en voisin jusqu'à ce qu'une affectation soit trouvée. Si au cours de la recherche aucune affectation n'est possible dans la liste de préférence d'un agent sans générer de jalousie, alors un retour arrière sur l'agent précédent est opéré et une nouvelle affectation est tentée pour cet agent. De par l'heuristique de choix des objets à affecter, on s'assure de trouver des solutions Pareto-optimales (voir **??**). L'algorithme est lancé depuis tous les points de départs possibles et dans chaque direction possible afin de trouver toutes les solutions optimales. Donc pour chaque agent a_i, l'algorithme va dans la direction $a_{i+1} \rightarrow a_{i+2} \rightarrow ...$ pour trouver une solution puis dans la direction $a_{i-1} \rightarrow a_{i-2} \rightarrow ...$ pour tenter d'en trouver une autre. Puisque l'algorithme est volontairement naïf, sa complexité temporelle est élevée (il peut théoriquement considérer les $n!$ solutions possibles) mais la taille des instances étant limitée il est suffisamment rapide en pratique.

Définition* 1.** *Une solution* ***Pareto-optimale *est une solution telle que l'on ne peut affecter un meilleur objet à un agent sans devoir affecter un objet moins aimé à un ou plusieurs autres agents.*

Nombre d'affectations (`naff`) Une première mesure résultant de l'exécution de l'algorithme est le nombre de tentatives d'affectation qui est incrémenté à chaque fois que l'algorithme tente d'allouer un objet à un agent. Un déroulement sans retours arrière affiche donc un nombre d'affectation égal au nombre d'agents mais une instance plus difficile à résoudre pour l'algorithme engendrera un plus grand nombre d'essais à cause des retours arrière. On peut mesurer la moyenne de ce nombre sur toutes les exécutions de l'algorithme sur une instance et obtenir une estimation du nombre d'objets à considérer en moyenne lors d'une résolution.

Solutions Pareto-optimales (`npo`) L'algorithme de backtracking permet de trouver l'ensemble de ces solutions pour une instance. Ce sont, d'après notre hypothèse de choix (commencer par les préférés), les instances les plus facilement accessibles. Compter leur nombre donne donc une indication de la force des contraintes dans l'instance. S'il existe peu de ces solutions alors on peut, dans certains cas, inférer un temps de recherche plus grand mais il est important de combiner cette mesure avec la notion de regret détaillée plus bas. En effet, une instance dans laquelle tous les agents peuvent avoir leur objet préféré compte une seule solution Pareto-optimale mais ne peut être considéré comme difficile.

Regret associé à une solution (`rgrt`) Toujours en rapport avec l'hypothèse de choix des objets, on définit le regret global associé à une solution tout simplement comme la somme des indices de l'allocation dans les listes de préférences respectives. Soit $ind(.)$ la fonction qui a un objet associe son indice dans la liste de préférence de son agent alors le regret R peut s'écrire comme suit.

$$R = \sum_{i=1}^{n} ind(\mathcal{A}(a_i))$$

Ainsi, en utilisant une indexation à partir de 0, dans la **??**, la solution en jaune a un regret de 1 et la solution en bleu a un regret de $3 + 4 + 2 + 4 + 2 + 1 + 2 = 18$. La première est bien évidemment la plus facile à trouver.

De cette mesure de regret on peut tirer plusieurs métriques intéressantes :

Index							
0	2	3	2	5	7	4	6
1	1	4	6	6	3	5	1
2	3	2	1	7	2	6	3
3	6	5	3	3	1	7	7
4	5	7	4	4	6	1	5
5	3	1	7	2	4	2	2
6	7	6	5	1	5	3	4

TABLE 2 – Exemple de deux solutions pour une instance de 7 agents

— Le regret minimum à atteindre pour trouver une solution (`minr`),
— Le regret moyen dans les solutions Pareto-optimales (`avgr`),
— Le regret minimum pour les agents en extrémité puisque ce sont eux qui sont choisis en premier le plus fréquemment (`minr_ext`).

Observation 2. *Une grande proportion de joueurs, après plusieurs résolutions, exhibe une méthode de réduction des domaines de variables basée sur la présence de certains objets en top préférence.*

Nombre de positions possibles pour un objet (`npstn`) Grâce à l'**??**, il a été remarqué que l'on pouvait, dans certaines instances, déterminer des positions impossibles pour un objet donné. En effet, la présence d'un objet en top préférence dans plusieurs listes voisines empêche son affectation aux agents concernés. Par exemple dans la **??**, aucun des agents a_4, a_5 ou a_6 ne peut se voir affecté l'objet 7 sous peine de rendre au moins un de ses voisins jaloux. D'une manière générale, si un objet est en top d'une liste de préférence, il ne peut pas être affecté à un agent voisin sans créer de jalousie chez l'agent l'ayant en top. De plus, on ne peut affecter à un agent que les $(n - nbvoisins)$ premiers objets de sa liste de préférences car il est sinon impossible de ne pas envier au moins un de ses voisins. Dans la **??**, on a donc seulement deux positions possibles pour l'objet 7, surlignées en bleu. Les positions des agents a_3, a_4, a_5 et a_6 sont prohibés par la présence de l'objet en top chez un voisin et on ne peut pas affecter l'objet à l'agent a_2 car il serait forcément jaloux de ses voisins.

	7	2	6	2	7	7	7
	2	1	4	6	2	1	5
	4	3	7	4	1	6	6
	3	4	2	5	3	5	3
	6	5	3	1	6	4	2
	5	6	5	3	5	2	1
	1	7	1	7	4	3	4
Agents	a_1	a_2	a_3	a_4	a_5	a_6	a_7

TABLE 3 – Exemple des positions possibles d'un objet

2.3. MODÉLISATION ASP

Acquérir l'ensemble des solutions possibles d'une instance était intéressant pour comprendre ses contraintes ainsi que pour certaines métriques détaillées dans la suite de cette section. Pour cela, le problème a été modélisé sous la forme d'un programme d'**Answer Set Programming** (ASP), particulièrement efficace pour la résolution de problèmes NP-difficiles. La génération de modèle ainsi que les contraintes écrites dans le formalisme ASP sont visibles en **??**. Pour chaque

```
% Génération:
% On doit avoir au plus un objet par agent.
1{ aff(A, O) : object(O) }1 :- agent(A).

% Un objet O ne peut être affecté qu'une seule fois.
:- aff(A1, O), aff(A2, O), A1 != A2.

% Pas de jalousie entre voisins.
:- aff(A1, O1), aff(A2, O2),
        position(A1, O1, P1),
        position(A1, O2, P2),
        P2 < P1, |A1-A2|==1.
```

FIGURE 3 – Codage des contraintes de LEF en ASP

instance à résoudre, il suffit donc de générer l'encodage des données, c'est-à-dire les listes de préférences. Pour une instance à n agents, cela prend la forme visible en **??**. Après résolution,

```
% On a les agents 1 à n.
agent(1..n).
% On a les objets 1 a n.
objets(1..n).
% On définit les positions des objets dans les listes de préférences.
% Pour chaque agent A, pour chaque objet O, on définit l'indice p dans
% la liste de préférence:
position(A, O, p).
[...]
```

FIGURE 4 – Codage d'une instance de LEF en ASP

les valeurs vérifiant le prédicat `aff/2` donnent les affectations possibles pour chaque modèle. On peut ainsi récupérer le **nombre total de solutions** (`nsols`) pour une instance. Certaines de ses solutions sont complètement dominées au sens de Pareto par celles trouvées via l'algorithme de backtracking mais il est tout de même important de les prendre en compte car elles donnent une indication sur la difficulté à vérifier les contraintes du problème.

Nombre de variables dites "frozen" (`nfrozen`) Des concepts de la programmation logique découlent les frozen variables. Ce sont les variables qui ne peuvent prendre qu'une seule valeur dans l'ensemble des solutions. Un grand nombre de ces variables implique généralement un faible nombre de solution et donc un problème très contraint. Cela peut-être bénéfique pour un joueur car la seule position possible est potentiellement déductible de la façon décrite dans le paragraphe sur les positions possibles d'un objet (voir sous-section précédente).

2.4. Analyse de Fitness Landscape

Lors des recherches bibliographiques dédiées à ce projet, beaucoup de résultats concernant l'analyse de *fitness landscape* sont apparus [**pitzer**, **garnier**, **stadler**]. Le but est souvent de jauger la difficulté de résolution d'un problème et de pouvoir ainsi déterminer quel algorithme ou composante d'algorithme est la plus à même de résoudre le problème rapidement. Un certain nombre d'outils tirés de cette littérature ont été explorés dans l'espoir d'expliquer les ressentis humains. Ici le problème n'est pas de sélectionner un algorithme comme dans ces articles mais de déterminer les algorithmes utilisés par un joueur. Par la suite on considère le problème

d'optimisation combinatoire correspondant à la relaxation du problème LEF et le paysage étudié découle de l'**??**.

***Observation* 3.** *Lorsqu'une allocation complète est atteinte mais qu'un ou plusieurs agents sont jaloux, certains joueurs ont tendance à échanger les objets de deux agents dans l'espoir de réduire le nombre de jaloux.*

Définition* 2.** *Pour définir le fitness landscape, il faut tout d'abord en définir les composantes. D'après nos observations, une fonction de fitness (f : solution $\to \mathbb{N}$) évidente est le nombre d'agents jaloux dans une allocation donnée et le problème consiste alors en la minimisation de cette fonction. On peut ensuite définir une fonction de distance entre deux solutions ($d : s_1 \times s_2 \to \mathbb{N}$) qui compte le nombre minimum d'échanges nécessaires pour passer de s_1 à s_2. Soit $\mathcal{S}$ l'espace des solutions candidates, le* ***fitness landscape *$\mathcal{F}$ est alors la structure :*

$$\mathcal{F} = (\mathcal{S}, f, d)$$

***Définition* 3.** *Le voisinage $\mathcal{N}(x, \epsilon)$ d'une solution x dans un tel paysage comprend toutes les solutions situées à une distance ϵ de x.*

$$\mathcal{N}(x, \epsilon) = \{x' \mid d(x, x') = \epsilon\}$$

Définition* 4.** *La notion de* ***Landscape Walk *[**pitzer**] est un outil très utilisé dans l'analyse de fitness landscape. On s'intéresse ici à deux types distincts :*

- ***Random walk*** *dans lequel on se déplace au sein de l'espace des solutions de manière aléatoire, passant de voisin en voisin,*
- ***Adaptative walk*** *où à chaque pas de temps une solution de meilleure fitness dans le voisinage est choisie pour poursuivre l'exploration.*

Bassin d'attraction (`bs`) Depuis les optimums trouvés grâce au programme ASP, il est possible de calculer des bassins d'attraction. Cela implique le calcul des « chemin descendants » jusqu'à un optimum : un tel chemin est une séquence de solutions candidates présentant chacune une fitness moins grande que la précédente (comme définit dans [**pitzer**], un chemin descendant P entre x_0 et x_n est $\{x_i\}_{i=0}^{n}$ avec $(\forall i < j)\ f(x_i) \geq f(x_j), f(x_0) > f(x_n)$ et $d(x_{i+1}, x_i) = 1$). Alors on note le bassin d'attraction faible d'un optimum o :

$$B(o) = \{x | x \in S, P(x, o)\}$$

La taille d'un tel bassin donne une idée de la probabilité de convergence vers un optimum depuis une solution candidate quelconque. Un petit bassin autour d'un optimum global entraîne un temps de recherche probablement plus long.

Calculer ces bassins exhaustivement peut se révéler coûteux dans certaines instances et on pourra donc procéder à une estimation via des parcours adaptatifs [**garnier**] (longueur des parcours). La complexité est alors O(nombre de parcours $\times$ longueur max $\times$ $|\mathcal{N}|$).

***Définition* 5.** *Un optimum local x^* est une solution telle que toute solution voisine x présente une fitness inférieure à celle de x^*.*

$$\forall x \in \mathcal{N}(x^*),\ f(x) \leq f(x^*)$$

Nombre d'optimums locaux (`nlo`) Le nombre d'optimums locaux dans l'espace de recherche donne une indication de la rugosité du paysage. Selon l'**??**, les joueurs peuvent exécuter des parcours adaptatif et cette mesure se révèle alors intéressante. Pour l'estimer deux méthodes ont été considérées : échantillonnage aléatoire et échantillonnage via parcours adaptatifs. Dans

la méthode proposée par [**alyahya**], des solutions candidates sont tirées uniformément aléatoirement et sont évaluées. On obtient ainsi une procédure de complexité $O(n \times |\mathcal{N}|)$ avec n la taille de l'échantillon souhaité. La fiabilité de cet estimateur est largement dépendant sur cette taille choisie. Dans la méthode basée sur des parcours adaptatifs, des solutions candidates sont choisies uniformément aléatoire mais on procède à un parcours adaptatif dessus afin de trouver un optimum.

Auto-corrélation (`ac`) Une autre mesure permettant l'estimation de la rugosité est l'auto-corrélation de la fonction fitness dans des parcours de l'espace de recherche. Si elle est haute alors la fitness ne varie pas beaucoup de voisin en voisin : le paysage est dit « plat ». On s'est intéressé dans ce projet à mesurer cette corrélation dans le voisinage des optimums. L'intuition est que si un optimum a un grand bassin d'attraction mais que la corrélation est grande, alors il sera difficile pour un joueur de naviguer jusqu'à l'optimum. On procède à des parcours aléatoires en partant d'optimums et la mesure `ac` est donnée par la formule suivante [**pitzer**]

$$\rho(n) = \frac{E\left[f(x_i) - \bar{f})(f(x_{i+n} - \bar{f})\right]}{\sigma_f^2}$$

avec $\sigma_f^2 = \bar{f^2} - \bar{f}^2$ la variance de fitness et E la fonction d'espérance mathématique. On calcule ici $\rho(1)$ qui permet de calculer l'auto-corrélation le long du parcours.

2.5. Apprentissage de la difficulté

Fort de ces mesures, il s'est agit ensuite d'établir la relation entre les caractéristiques d'une instance et la difficulté ressentie par les utilisateurs. Pour cela, l'application mobile (détaillée en **??**) était très utile et il a été demandé à un grand nombre de personne de résoudre des instances. À chaque niveau résolu l'application enregistre les informations suivantes : le *nombre de coups*, le *temps de résolution* et une *note de difficulté* donnée par le joueur. Un « coup » correspond à l'action de donner un objet à un agent.

3. Application mobile

L'application Android a été développée sous l'IDE Android Studio 3, avec l'API Android 16, supportant ainsi près de 99% des appareils Android en circulation. Le code de l'application a été entièrement géré avec Git et est d'ailleurs disponible à l'adresse suivante : `https://github.com/tndnc/pandroid/tree/master/equity`.

L'application devait répondre aux spécifications suivantes : permettre aux utilisateurs de sélectionner des niveaux, de les résoudre et d'en noter la difficulté. Pour les administrateurs il était également important de pouvoir facilement mettre à jour la liste des niveaux ainsi que de pouvoir récupérer les notes et meta-données issues de son utilisation par les utilisateurs. Ces données devaient pouvoir être exportées vers une base de données.

3.1. Conception de l'interface

L'interface centrale de l'application, l'aire de jeu d'Equity, est un des composants principaux du projet et a donc suivit un certain nombre d'itérations qui seront décrites dans la partie suivante. Le premier objectif de cette interface était de pouvoir représenter le jeu sur un écran de smartphone en mode portrait. En outre, il était important de réfléchir aux différents menus de l'application et au tutoriel permettant d'expliquer le jeu.

Ayant suivi en parallèle l'UE d'IHM pendant ce semestre, nous avons pu acquérir les connaissances et pratiques appropriées pour la conception de notre interface. Ces connaissance nouvellement acquises ont été appliquées tout au long du processus de réalisation de notre interface.

Il fallait tout d'abord identifier l'utilisateur qui était visé par notre application, mais nous nous sommes très rapidement rendu compte que cette notion était intrinsèque au sujet de notre projet. En effet, rappelons nous qu'il fallait évaluer la difficulté de nos niveaux et comparer nos prédictions à de vraies données. Or la notion de difficulté est subjective à n'importe quel être humain. Il nous fallait donc des données comprenant un maximum de diversité c'est-à-dire des utilisateurs de tous âges, de toutes formations, etc, afin de minimiser le biais de nos résultats. Nous nous sommes donc donnés pour but de créer une application pour tout type d'utilisateur et l'interface devait aller dans ce sens aussi. Notre premier objectif était d'établir les besoins utilisateurs et tâches de notre système.

3.1.1. Les besoins utilisateur

Afin de définir les besoins, il est de convention d'interviewer des utilisateurs potentiels pour comprendre leur réaction face au jeu. Un petit nombre de personne ont été consultées et confrontée au problème, initialement sous la forme visualisée en **??** (c'est-à-dire un graphe liant les agents avec des listes de préférences triées de haut en bas). Puisqu'aucune problème de compréhension n'était apparent à ce stade, les premiers prototypes de l'interface se sont donc vu largement inspiré par cette visualisation. Nous avons aussi étudié les interfaces de jeux mobiles populaires telle que Candy Crush ou encore 2048 pour mieux comprendre les caractéristiques d'une bonne interface. Ces données une fois regroupées nous ont permis de dégager plusieurs lignes directrices pour notre interface.

Lors de nos interviews nous avons demandé quelles étaient les caractéristiques d'un bon jeu d'après nos utilisateurs potentiels et l'un des points positifs le plus souvent pointé du doigt était la facilité avec laquelle un utilisateur comprenait les règles. Beaucoup se sont en effet plaint de la difficulté à « entrer » dans un jeu dès lors que la prise en main devenait difficile et que les règles du jeu étaient complexes. Or la notion de jalousie évoquée précédemment n'est pas forcément naturelle, il était donc crucial pour notre application d'avoir une interface la plus claire possible pour la compréhension du jeu. Toujours en rapport avec le premier souci, nous avons également remarqué au cours de nos recherches, que la quantité d'information à l'écran était un facteur important. Il apparaît que le sentiment de confusion est amplifié lorsqu'un nombre important de données s'affiche en même temps : il fallait prendre en compte ce critère pendant le développement de notre application. Enfin les personnes interviewées nous ont souvent mentionné le coté fluide et rapide qu'ils trouvaient nécessaire à une application de ce type. L'optimisation de l'interface a donc été une préoccupation certaine durant le développement.

Tout au long du processus, nous nous sommes surtout concentrés sur les critères d'usabilité suivants qui était à nos yeux les plus importants : la facilité d'apprentissage (easy to learn), la facilité d'usage (easy to use), la satisfaction (qui reste tout de même une notion très subjective), et la robustesse. Après tout, il ne faut pas oublier que le but final est que l'utilisateur passe un moment agréable en jouant à notre application.

3.1.2. Les fonctionnalités de notre système

Une fois ce travail en amont fait, nous devions établir notre système et puisqu'il est relativement facile de produire des prototypes Android, nous nous sommes directement attelés au développement de la première itération d'interface. La visualisation en forme de grille a été déterminée très tôt : chaque case de la grille représente une préférence d'un des agents. Il a ensuite été décidé que les agents, représentés par des carrés de couleurs différentes, seraient placés sur la gauche de la grille et leur préférences, représentées par des cercles de couleurs différentes, sur la même ligne que l'agent. Les préférences se lisaient de gauche à droite donc l'objet préféré se trouvait le plus à gauche. Cette première interface est visible en **??**.

La sélection d'une préférence pour un des agents (l'allocation d'un objet à un agent) dessine un cercle de la couleur de l'agent derrière le cercle représentant la préférence en question.

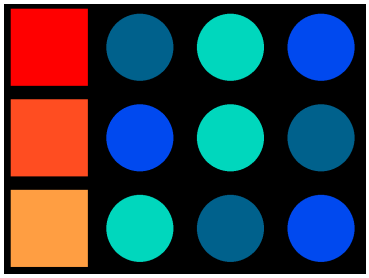


FIGURE 5 – Premier prototype d'interface de jeu

L'utilisateur peut donc choisir de sélectionner une certaine préférence en touchant le cercle correspondant. On note que le processus de sélection des ressources empêche de sélectionner plusieurs ressources pour un seul agent. À ce stade l'application est fonctionnelle.

Dans un soucis d'optimisation de l'espace et de lisibilité, la dernière modification structurelle de l'IHM consistait en la rotation de la grille de façon à ce que les agents se trouvent en haut et les préférences soient lues de haut en bas. Ce changement a permis l'utilisation d'une plus grande partie de la hauteur de l'écran et de pouvoir afficher de manière optimale les instances de plus grandes tailles (jusqu'à 7 agents) sur un grand nombre d'appareils.

L'étape suivante était de rendre l'interface agréable à l'œil et prête pour une distribution. Tous les changements sont visibles en **??**. En premier lieu une harmonisation des couleurs a été opérée offrant une palette réduite pour favoriser la lisibilité du problème. Les données de l'instance en revanche ont pu revêtir une palette plus complète de couleurs et de formes pour que les joueurs puissent facilement distinguer entre les différents objets. Plusieurs symboles, variants en formes et en couleurs, ont été dessinés avec un faible niveau de détail pour ne pas fatiguer l'œil. Des barres verticales ont été ajouté pour chacun des agents ce qui facilite la distinction des préférences par agent et l'identifiant du niveau est montré en haut de l'écran. Nous avons voulu ainsi minimiser la quantité d'information à assimiler. Par exemple les agents ne montrent plus des couleurs différentes, ils sont déjà différenciés par leur position dans l'interface.

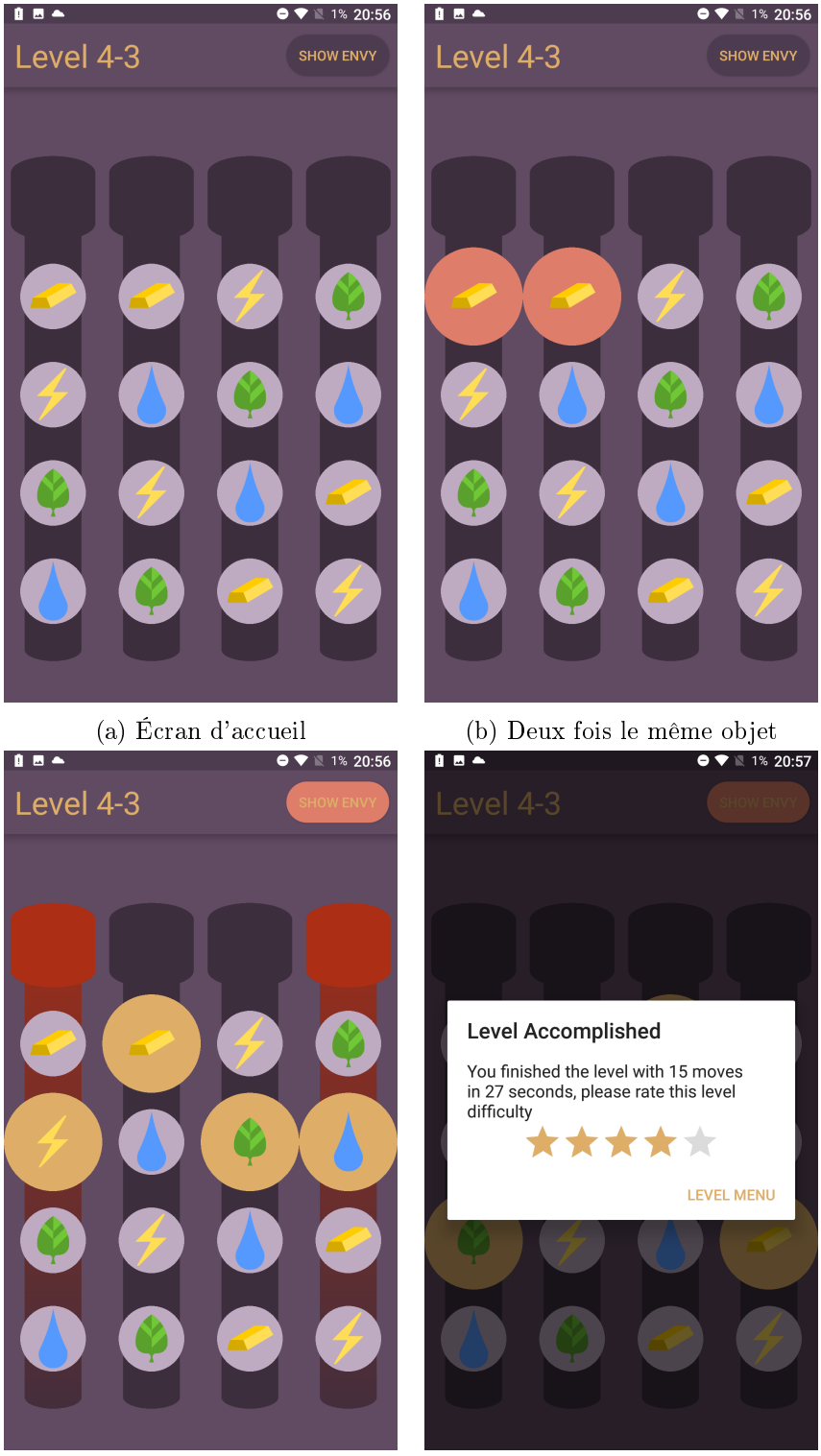
Afin d'évaluer notre application, nous avons fait tester cette interface à nos proches. De multiples retours nous ont cité l'incompréhension d'une solution erronée. Il était donc évident que pour améliorer l'expérience de jeu nous devions rendre la détection d'affectation non recevable plus aisée. Deux solutions ont été implémentées pour pallier ce problème. L'affichage d'une couleur différente lorsqu'une ressource est sélectionnée par deux agents permet au joueur de remarquer immédiatement qu'une telle assignation des ressources est impossible (**??**). La possibilité d'afficher en rouge les colonnes des agents jaloux améliore le potentiel d'amusement (**??**). Les joueurs passent moins de temps à rechercher pourquoi une solution n'est pas valide et plus de temps à chercher une meilleure solution. Notons que cette fonctionnalité facilite grandement le jeu, elle est donc désactivée par défaut et le joueur a la possibilité d'appuyer sur un bouton pour activer cet affichage.

Ces deux fonctionnalités rendent les résolutions plus aisées mais nous pensons que ces compromis permettent une expérience de jeu plus fluide, agréable et sans sentiment de confusion pour l'utilisateur, un point qui nous le rappelons, semble essentiel pour la distribution de l'application.

3.2. Composants Android

Une application Android s'organise en plusieurs composants et nous allons en présenter certains pour pouvoir décrire l'application et ses fonctionnalités aisément. Nous allons notamment concentrer notre attention sur le concept d'activité, nécessaire à la présentation de l'interface.

Une activité Android correspond à une fenêtre d'application et chaque application est donc composée d'au moins une activité. La navigation à l'intérieur d'une application se fait en chan-



(a) Écran d'accueil

(b) Deux fois le même objet

(c) Affichage de jalousie

(d) Écran de victoire

FIGURE 6 – Interface de jeu finale

geant d'activité. Leur apparence visuelle est principalement définie par leur *layout* qui structure l'agencement des vues (images, boutons, etc) et peut lier certains événements, initiés par l'utilisateur (comme par exemple un « clic » de bouton), à des actions prédéfinies (par exemple, lancer un niveau). Le layout, chargé par une activité à sa création, est décrit dans un fichier XML.

3.3. Structure de l'application

Notre application est composée de six activités que nous allons présenter ici. Des impressions d'écrans sont visibles en **??**.

- `MainMenuActivity` : l'activité principale de l'application, lorsque celle-ci est ouverte, l'utilisateur arrive sur l'écran qui correspond au menu principal, il a ensuite la possibilité de sélectionner un des quatres boutons du layout pour effectuer les actions suivantes : ouvrir le menu de sélection des niveaux, lancer le tutoriel, modifier son profil utilisateur ou ouvrir la page « à propos ».
- `LevelSelectActivity` : l'activité de sélection des niveaux dans laquelle la liste des niveaux est affichée, catégorisée par le nombre d'agents de chacun des niveaux. L'utilisateur a la possibilité de lancer chacun des niveaux présentés.
- `UserProfileActivity` : Cette activité remplace la `MainMenuActivity` lors du premier lancement de l'application. Elle comporte deux zones d'entrées de texte qui permettent à l'utilisateur de rentrer sa formation et son âge, données utiles lors de l'analyse des niveaux. On note que l'utilisateur a la possibilité de revenir sur cette activité depuis le menu principal ou de ne pas remplir les champs de texte s'il ne désire pas partager ces informations.
- `AboutActivity` : un bref résumé de l'objectif de notre projet et des raisons pour laquelle l'application a été développé est visible sur cette page.
- `TutorialActivity` : le tutoriel du jeu, on y trouve une description de l'interface de jeu, la définition de ce qu'est un agent jaloux, les modalités de victoire d'une partie ainsi que des captures d'écran pour illustration.
- `GameActivity` : l'activité où se déroule la partie. Elle est composée principalement d'un canevas sur lequel est affiché l'interface de jeu. Deux autres vues permettent d'afficher le nom du niveau et un bouton activant la visualisation de l'envie chez les agents.

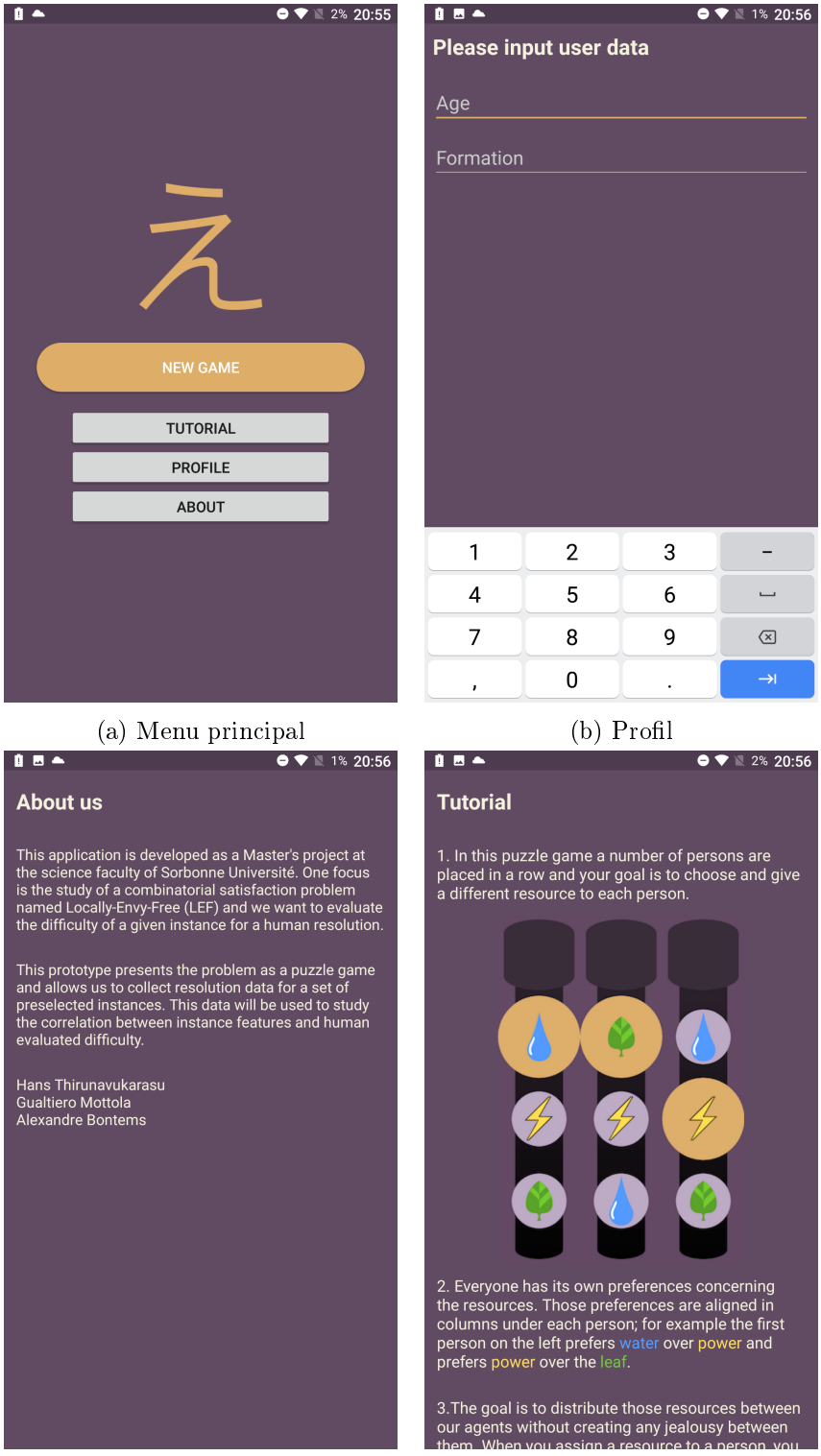
3.4. Le modèle du jeu

Le package `models` contient les classes nécessaires à la représentation du jeu dans l'application. On y trouve la classe `Model`, appelée dès lors qu'il est demandé de faire une modification du modèle (assignation d'un objet à un des agents par exemple) ou encore de récupérer des informations du jeu comme le nombre d'agents d'un niveau. Cette classe stocke en effet la grille qui représente un niveau, comprenant les agents et les préférences. Le package contient également la définition de l'interface `IPiece`, implémentée par les classes `Preference` et `Actor` qui représentent respectivement les préférences et les agents.

- Définition d'une Pièce (interface `IPiece`) : cette interface est composé de deux méthodes ; la méthode `getId` retourne l'identifiant de la pièce et la méthode `getPosition` permet de retourner la position de la pièce sous forme d'un objet position qui contient les indices de la ligne et de la colonne dans lesquelles se trouve la pièce.
- Définition d'un Acteur (classe `Actor`) : cette classe implémente l'interface `IPiece` et toute ses méthodes. Elle représente les agents du jeu.
- Définition d'une Préférence (Classe `Préférence`) : cette classe implémente également l'interface `IPiece`. On y ajoute cependant là deux entiers : `value` (qui permet de déterminer de quel type est cette préférence) et `selectedby` (qui permet de savoir si cette préférence est sélectionnée par un des agents et par qui).

3.4.1. Interface GameView

L'interface centrale de l'application, l'ère de jeu de equity est un des composants principaux du projet, le canevas utilisé pour dessiner les agents et leur préférence s'organise de la façon



(a) Menu principal (b) Profil

(c) About (d) Tutoriel

FIGURE 7 – Design final des menus de l'application

suivante : les agents sont placé en haut de la surface sur une ligne horizontale, et leurs préférences sont arrangés verticalement, leur objet préféré étant le premier en partant du haut, derrière chaque agent est dessiné une bande verticale sur laquelle sont aligné les préférences

```
private boolean isJealous(Preference[] P1,Preference[] P2){
    if (P1[0] == null){
        return false;
    }
    Preference P2pref = null;
    Preference P1pref = null;
    for (Preference pref:P1){
        if(pref.getSelectedby() != -1){
            P1pref = pref;
        }
    }
    for(Preference pref:P2){
        if(pref.getSelectedby() != -1){
            P2pref = pref;
        }
    }
    if(P1pref == null || P2pref == null){
        return false;
    }
    for(Preference pref:P2){
        if(pref.getValue() == P1pref.getValue()){
            if(pref.getPos().getCol() < P2pref.getPos().getCol()){
                return true;
            }
        }
    }
    return false;
}
```

FIGURE 8 – Fonction de détection de la jalousie

3.5. RÉCUPÉRATION DES DONNÉES

L'analyse de diverses instances du problème nous a permis donc de dégager plusieurs métriques, qui peuvent être utilisées pour essayer de prédire la difficulté de nos niveaux. Cependant, il nous fallait confronter ces prédictions avec des données réelles , qui seraient justement basé sur ces critères, et ce afin de trouver si oui ou non, il existait une corrélation entre la difficulté d'un niveau et ces derniers.

Une fois l'application mobile réalisée, nous avions donc décidé de la faire tester sur un public de taille variante et diverse, pour recueillir assez de données. Mais nous avions besoin d'un moyen de les écrire quelque part , la contrainte étant que l'utilisateur allait jouer sur son téléphone. Il nous fallait donc envoyer ces informations sur une sorte de base de données. Notre regard s'est alors porté sur l'API Google Sheets qui nous offrait une relaxation de cette contrainte.

Googlesheets Nous avons donc intégré dans l'application mobile, une solution permettant l'envois de diverses données joueurs qui étaient enregistrées lors de la résolution d'une instance. l'API Google Sheets permet l'écriture/lecture de données sur une Google Sheet via différentes plateformes : application mobile, site web, programme python, etc... En ce qui nous concerne, notre but est d'envoyer les différentes métriques abordées précédemment sur une Google Sheet avec diverses informations sur l'utilisateur pour ensuite les analyser et appliquer une régression dessus.

La communication entre la feuille et notre application est géré via l'API, et elle est sécurisé. En effet l'API utilise le protocole OAuth 2.0 pour autoriser les communications. Nous avons donc besoins d'identifiants , deux moyen sont disponibles :

— l'usage d'une Clé API
— l'usage d'un compte de service Google

Néanmoins, la première méthode requiert une confirmation de l'utilisateur à chaque fois qu'il envoie des données via son téléphone car l'application demandera la permission d'interagir avec la feuille sous le compte de l'utilisateur. Nous avons donc opté pour la seconde solution qui user d'un compte de service dédié à notre application. L'application interagira via ce compte sans demander la confirmation de l'utilisateur, ce qui est crucial pour l'envie de l'utilisateur à vouloir tester notre application sur une courte durée/longue durée. Dans les deux cas, un fichier JSON est créé et il contiendra les informations nécessaire pour créer nos identifiants.

L'intégration de cette solution dans notre application se traduit donc par deux classes :

— La classe SheetsServiceUtil
— La classe GoogleSheetsWriteUtil

La première classe java SheetsServiceUtil contient une unique méthode getSheetsService qui permet la création d'une instance de l'objet Sheet qui est l'intermédiaire pour écrire et lire via l'API. Cette instance contiendra les identifiants nécessaire à l'autorisation de la communication entre l'application et la feuille.

La deuxième classe java GoogleSheetsWriteUtil contient toutes les méthodes qui seront appelées pour l'écritures de nos données. La méthode setup() permet entre autre la création d'une instance de l'objet Credential, à partir de notre fichier JSON qui est contenu dans les ressources de notre applications (pour rappel, on accède aux ressources de nos applications via ctx.getRessourses() ;). Cette instance est ensuite passée en paramètre dans la méthode getSheetsService de notre classe SheetsServiceUtil qui retournera un objet Sheets, sur lequel toute nos méthodes agiront par la suite. Par ailleurs, nous récupérons aussi les données du profil utilisateur enregistré si elles existent pour permettre de détecter la modification d'un profil.

Nous utilisons des AsyncTask pour envoyer nos données. En effet il est impossible sous android de faire des opérations réseaux (transfert de données) , sous le thread principale, on utilise donc des AsyncTask pour le faire en background. Cela nous évite des lags au niveau de l'interface qui est entièrement gérée par le main thread. et ça reste invisible à l'oeil de l'utilisateur ce qui est à nouveau crucial quant au côté agréable et fluide de son expérience.

Nous avons donc trois static private Class dans SheetsServiceUtil qui héritent tous de la class AsyncTask , une classe d'Android qui permet de lancer des threads aisément :

— WriteUserInfoAsyncTask
— ModifyUserInfoAsyncTask
— WriteUserEvalAsyncTask

Ces trois classes implémentent ainsi la méthode protected doInBackground , qui prend en paramètre un string data, c'est à dire les données à envoyer, et qui permet leur exécution en tâche de fond. Nous appelons donc à l'intérieure les méthodes de l'API de Google Sheets pour update (pour modifyUserInfoAsync) ou écrire de nouvelles données (pour WriteUserEvalAsyncTask et ModifyUserInfoAsyncTask) dans notre feuille en utilisant l'instance de la classe Sheets que l'on avait créé au préalable.

```
1 private static class WriteUserInfoAsyncTask extends
```

```
AsyncTask<String, Void, Void>{
   @Override
   protected Void doInBackground(String... data) {
      ValueRange body = new ValueRange()
                     .setValues(Arrays.<List<Object>>asList(
                              new List[]{Arrays.asList(data)}
                     ));
      try {
         AppendValuesResponse result = sheetsService.spreadsheets()
         .values().append(SPREADSHEET_ID, "User_Info_2!A1", body)
         .setValueInputOption("USER_ENTERED")
         .execute();
         Object userPos = result.getUpdates().get("updatedRange");
         prefs.edit().putString("userPos", (String) userPos).apply();
      } catch (IOException e) {
         Log.w("GoogleSheet", "Error during user info write;
␣␣␣␣␣␣␣␣rescheduling");
         failedUserInfo = data;
         e.printStackTrace();
       }
      return null;
   }
}
```

Fonction WriteUserInfoAsyncTask

Comme il est possible que l'écriture des données sur la feuille puisse échouer (problème de réseau, téléphone non connecté à internet, etc...) , il est nécessaire de sauvegarder les données non envoyées pour permettre une planification d'un renvois plus tard. Il existe donc deux variables dans notre classe SheetsServiceUtil pour cela : failedEvaluation si l'écriture d'une évaluation échoue failedUserInfo si l'écriture/modification du profil utilisateur échoue

Avant toute nouvelles écritures, il faut donc vérifier s'il n'en existe pas qui ont échouées. Pour cette raison, nous avons créé trois méthodes : WriteUserInfo(String data), ModifyUserInfo(String data) , WriterUserEval(String data), qui appelle toujours au début une méthode "CheckFailures" qui verifie si failedEvaluation et failedUserInfo sont vide ou non. Si elles ne le sont pas, alors on crée de nouvelles asyncTask pour renvoyer ces données. Et dans tous les cas, on crée de nouvelles AsyncTask correspondant à la méthode qui est appelé (par exemple WriteUserInfoAsyncTask pour la méthode WriteUserInfo) pour les nouvelles données passées en paramètre de ces méthodes.

```
public void writeUserEvaluation(String... data) {
   this.checkFailures();
   new WriteUserEvaluationAsyncTask().execute(data);
}
```

Méthode WriteUserEvaluation

4. Conclusion

A. LEF Solver

B. Diagramme de classe d'Equity