



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 (Local Envy Freeness), présenté en [2], 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	2
2	Analyse d'instances	3
2.1	Solvabilité d'une instance	3
2.2	Résolution par backtracking	4
2.3	Modélisation ASP	6
2.4	Analyse de Fitness Landscape	7
2.5	Apprentissage de la difficulté	8
3	Application mobile	9
3.1	Conception de l'interface	9
3.1.1	Les besoins utilisateur	9
3.1.2	Interface de jeu	10
3.1.3	Autres pages	12
3.1.4	Tutoriel	12
3.2	Structure de l'application	13
3.3	Les modèles du jeu	14
3.4	Récupération des données	15
4	Conclusion	17

Références	17
A LEF Solver	18
B Diagramme de classe d'Equity	18

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.

Défini dans [2], 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 Figure 1). 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 Figure 1, 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 défini 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.

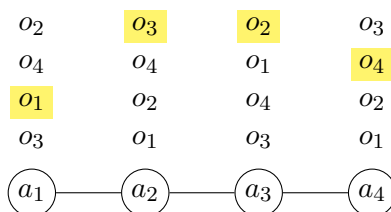


FIGURE 1 – Exemple d'instance étudiée du problème LEF

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.

```

1 Indices := []
2 Pour chaque agent a:
3     % Il faut que les objets voisins ne soient pas préférés à l'indice
4     % choisi donc on garde une place pour les agents en extrémités et
5     % deux places pour les restants.
6     Si a est le premier agent ou le dernier agent:
7         Indices[a] := valeur aléatoire entre 1 et n-1
8     Sinon:
9         Indices[a] := valeur aléatoire entre 1 et n-2
10
11 Pour chaque agent a:
12     Prefs[a] := []
13     ValeursPossibles := {1, ..., n} \ (Indices[Voisins[a]] et Indices[a])
14
15     Pour chaque indice i < Indices[a]:
16         k := valeur aléatoire parmi ValeursPossibles
17         Prefs[a, i] := k
18         ValeursPossibles := ValeursPossibles \ {k}
19
20     Prefs[a, i] := Indices[a]
21     ValeursPossibles := ValeursPossibles et Indices[Voisins[a]]
22
23     Pour chaque indice i > Indices[a]:
24         k := valeur aléatoire parmi ValeursPossibles
25         Prefs[a, i] := k
26         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

Un exemple d’instance générée se trouve en Table 1. 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. Chaque couleur correspond à un objet différent et les objets alloués aux voisins apparaissent en couleur.

2.2. RÉOLUTION PAR BACKTRACKING

Avec des instances résolubles, 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**).

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 influencer 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 Définition 1). L’algorithme est lancé depuis tous les points de départ 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 $a_{i+1} \rightarrow a_{i+2} \rightarrow \dots$ pour trouver une solution puis dans la direction $a_{i-1} \rightarrow a_{i-2} \rightarrow \dots$ 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 (avg_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 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 à 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 Table 2, 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.

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

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

- 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_extr** et **minr_extl**). Les deux regrets sont mesurés séparément.

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'**Observation 2**, 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 Table 3, 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 Table 3, 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 Figure 3. Pour

```

1 % Génération:
2 % On doit avoir au plus un objet par agent.
3 1{ aff(A, 0) : object(0) }1 :- agent(A).
4
5 % Un objet 0 ne peut être affecté qu'une seule fois.
6 :- aff(A1, 0), aff(A2, 0), A1 != A2.
7
8 % Pas de jalousie entre voisins.
9 :- aff(A1, 01), aff(A2, 02),
10     position(A1, 01, P1),
11     position(A1, 02, P2),
12     P2 < P1, |A1 - A2| == 1.

```

FIGURE 3 – Codage des contraintes de LEF en ASP

chaque 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 Figure 4. Après résolution, 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.

```

1 % On a les agents 1 à n.
2 agent(1..n).
3 % On a les objets 1 à n.
4 objets(1..n).
5 % On définit les positions des objets dans les listes de préférences.
6 % Pour chaque agent A, pour chaque objet O, on définit l'indice p dans
7 % la liste de préférence:
8 position(A, O, p).
9 [...]

```

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

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 [4, 3, 5]. 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**.

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 ($f : \text{solution} \rightarrow \mathbb{N}$) 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 ($d : s_1 \times s_2 \rightarrow \mathbb{N}$) 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** [4] 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,
- **Adaptive 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 [4], 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 [3] (longueur des parcours). La complexité est alors $O(\text{nombre de parcours} \times \text{longueur max} \times |\mathcal{N}|)$.

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) La modalité de l'espace de recherche donne une indication de la rugosité du paysage. Selon l'**Observation 3**, les joueurs peuvent exécuter des parcours adaptatif et cette mesure se révèle alors intéressante. Pour l'estimer plusieurs méthodes ont été considérées : échantillonnage aléatoire et échantillonnage via parcours adaptatifs. Dans la méthode proposée par [1], 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 de la 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. Ce rapport n'étudie que des instances de taille n'excédant pas 7 agents ce qui veut dire un maximum de $7! = 5040$ solutions possibles. Ce faible nombre permet de compter exhaustivement le nombre d'optimums locaux.

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 [4]

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

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 agi 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 Section 3) 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, nommée Equity, 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 suivi 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 connaissances nouvellement acquises ont été appliquées tout au long du processus.

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. L'idée était 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 personnes ont été consultées et confrontées au problème, initialement sous la forme visualisée en Section 2 (c'est-à-dire un graphe liant les agents avec des listes de préférences triées de haut en bas). Puisqu'aucun problème de compréhension n'était apparent à ce stade, les premiers prototypes de l'interface se sont donc vu largement inspirés 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 plaints de la difficulté à « entrer » dans un jeu dès lors que la prise en main devenait difficile et que les règles 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 côté 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. INTERFACE DE JEU

Une fois ce travail en amont fait, nous devons 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 Figure 5.

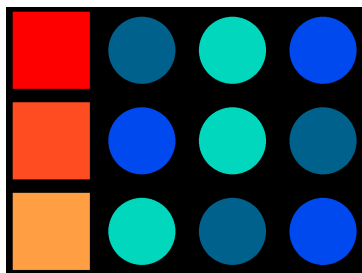


FIGURE 5 – Premier prototype d’interface de jeu

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. 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 à plus grande échelle. Tous les changements décrits sont visibles en Figure 6a. 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, variés 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 ajoutées 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 car ils sont déjà différenciés par leur position dans l'interface.

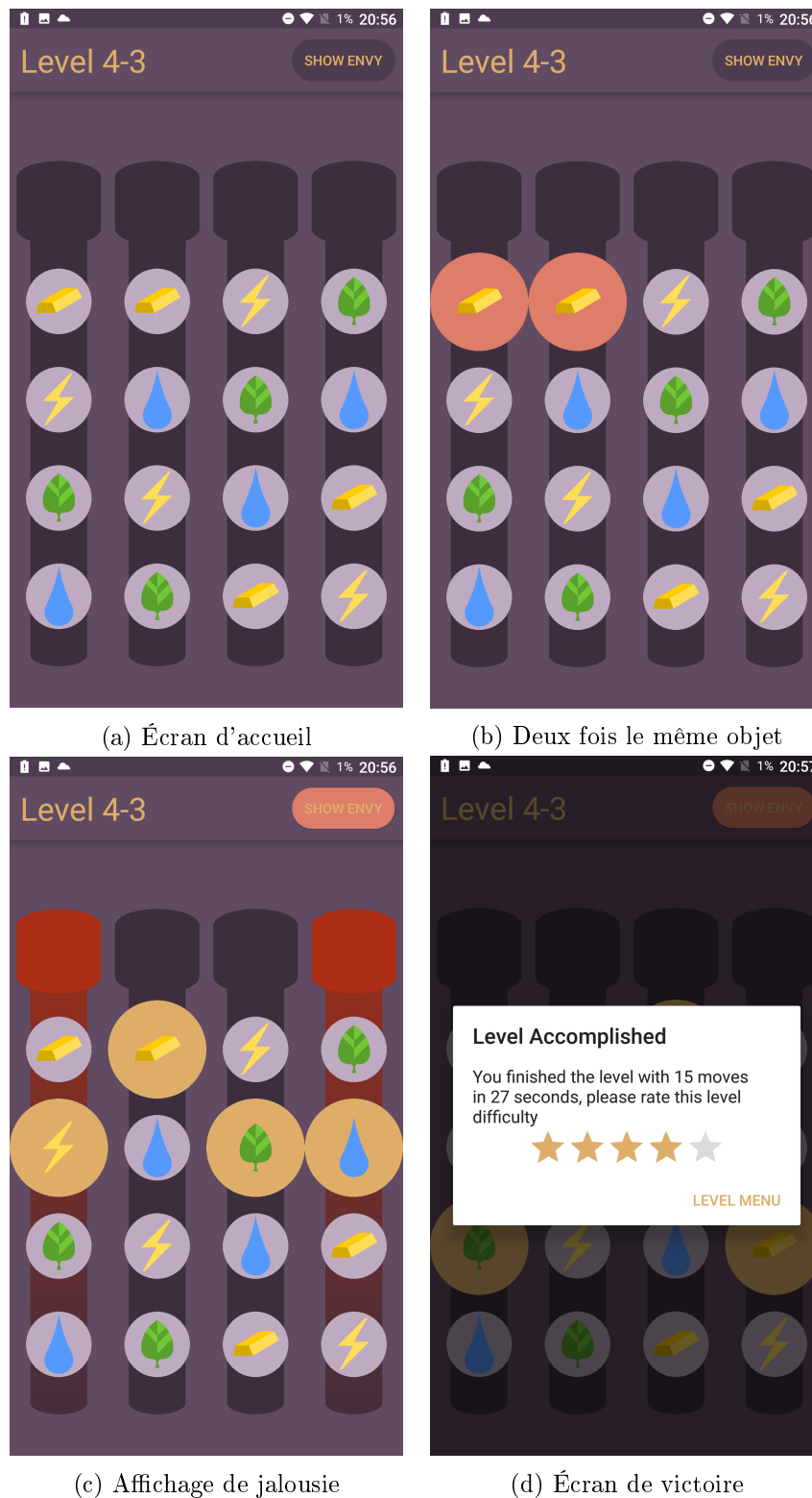


FIGURE 6 – Interface de jeu finale

Afin d'évaluer notre application, nous avons fait tester cette interface à nos proches. De mul-

tiples retours relataient ne pas comprendre quand et pourquoi une solution était 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 ou plus permet au joueur de remarquer immédiatement qu'une telle assignation des ressources est impossible (Figure 6b). La possibilité d'afficher en rouge les colonnes des agents jaloux améliore le potentiel d'amusement (Figure 6c). 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 additions 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. Lorsqu'un niveau est résolu une fenêtre flottante annonce la victoire accompagnée du temps de résolution et du nombre de coups réalisés (Figure 6d). Elle demande également à l'utilisateur de noter la difficulté ressentie pour le niveau en question. Ces données sont sauvegardées en ligne lorsque le joueur quitte la fenêtre.

3.1.3. AUTRES PAGES

Le reste de l'application devait comprendre un menu de sélection des niveaux, une page profil, un tutoriel et une page « à propos » expliquant le projet. Ainsi dans la version finale de l'application, le **menu principal** (Figure 7a) est la première page sur laquelle arrive un utilisateur en ouvrant l'application. Il permet d'accéder à la liste des niveaux et aux autres pages citées ci-dessus. Le **menu de sélection des niveaux** (Figure 7e) catégorise les niveaux par nombre d'agents. L'utilisateur a la possibilité de jouer à tous les niveaux présentés et une pastille de texte indique si un niveau a été terminé ou commencé. Le **page de profil** s'affiche au lancement de l'application lors de la première utilisation et est accessible depuis le menu principal. Elle permet, au moyen de deux zones de texte, de spécifier ou modifier les informations relatives au profil d'un joueur (âge, formation). Si un utilisateur ne souhaite pas divulguer ses informations il lui est possible de laisser les champs vides. Enfin la **page « à propos »** présente un bref résumé de l'objectif du projet et des raisons pour lesquelles l'application a été développée.

3.1.4. TUTORIEL

Une composante très importante de l'application était le tutoriel permettant d'expliquer les règles du jeu. Une version interactive a d'abord été considéré qui aurait expliqué les actions nécessaires pour vérifier les contraintes tout en demandant à l'utilisateur de les effectuer. Par manque de temps cependant, une version combinant texte et images d'exemple a finalement été implémentée (Figure 7c). Il se déroule en trois étapes. Tout d'abord le but du jeu est explicité : trouver une allocation des objets pour les agents, un unique objet par agent. Ensuite les préférences sont abordées et les joueurs apprennent quel est l'ordre pour chaque agent. Enfin la notion de jalousie est montrée et expliquée à l'aide d'un exemple sous forme d'image. Le niveau servant d'exemple peut être retrouvé dans la liste des niveaux et peut être joué pour parfaire la compréhension.

Expliquer la notion d'envie à l'aide d'un court texte s'est révélé être un défi et plusieurs itérations se sont succédées avant que le tutoriel puisse être considéré fonctionnel. Bien sûr il est probablement très améliorable.



FIGURE 7 – Design final des menus de l'application

3.2. STRUCTURE DE L'APPLICATION

Une application Android s'organise en plusieurs composants et nous allons en présenter certains pour pouvoir décrire l'implémentation de l'application 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 changeant 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 (lancer un niveau par exemple). Le layout, chargé par une activité à sa création, est décrit dans un fichier XML.

Notre application est ainsi composée de six activités correspondant aux pages décrites plus haut :

- `MainMenuActivity`,
- `LevelSelectActivity`,
- `UserProfilActivity`,
- `AboutActivity`,
- `TutorialActivity`,
- `GameActivity`.

Un bref Schéma de Navigation d'Interface (SNI) est disponible en Figure 8 (les conventions MACAO ne sont pas suivies).

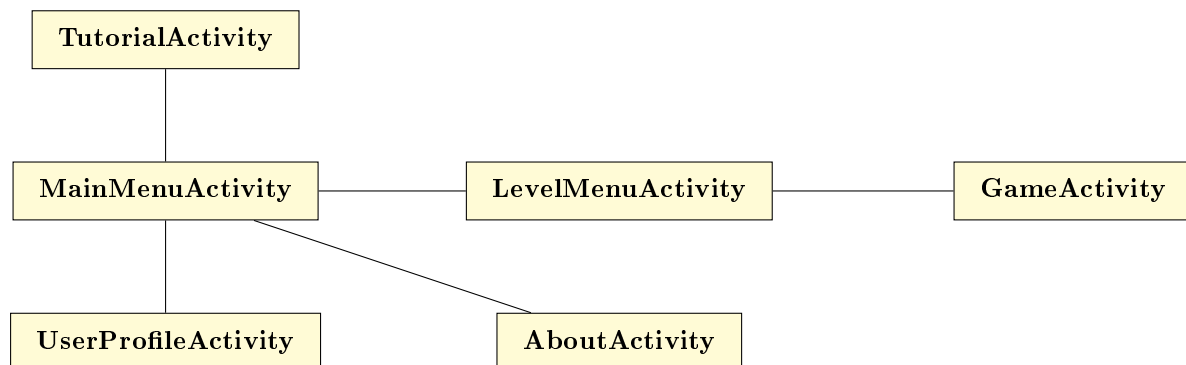


FIGURE 8 – SNI

3.3. LES MODÈLES DU JEU

Nous allons ici décrire la partie back-end de l'application, en particulier les objets et classes utilisées pour représenter le jeu. Toutes les instances sont stockées dans un fichier XML et une classe `LevelLoader` permet de le *parser* et charger les niveaux en mémoire vive. Le menu de sélection des niveaux qui est implémenté grâce à un `RecyclerView` peut alors être peuplé.

Le package `models` contient les classes nécessaires à la représentation du jeu dans l'application. On y trouve la classe `Level` construite à partir du XML et qui définit le nom d'un niveau, les préférences associées aux agents et sa taille. On y trouve aussi 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). Cette classe est utilisée lorsqu'un joueur joue pour ne pas affecter la représentation en mémoire des niveaux. L'écran de jeu est rendu à partir d'un canevas pour faciliter l'ajout futur d'animations et autres éléments esthétiques. C'est pourquoi il a été nécessaire d'implémenter des classes pouvant être dessinées et représentant les différents éléments de la grille de jeu. Le package contient donc 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 composée 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 toutes 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).

Nous utilisons ensuite ces trois classes dans la classe **Model** décrite précédemment. Une des fonctionnalités que nous souhaitons mettre en valeur grâce à cette représentation est le calcul de la jalousie entre les agents. Il est très facile de récupérer des colonnes de la grille et de les comparer pour savoir si l'agent correspondant à cette ligne est jaloux. La fonction prends en arguments deux suites de préférences **P1** et **P2** et permet de détecter si l'agent **P2** est jaloux de l'agent **P1**. On commence par récupérer les préférences de chaque colonne et on vérifie qu'un objet a été alloué. Puis on compare la position de l'objet alloué à l'agent **P1** avec celle de **P2** ; si l'objet de l'agent **P1** est préférée par l'agent **P2** a celui qui lui est assigné alors cette fonction retourne vrai. On note que cette méthode permet la détection de la victoire du jeu en calculant la jalousie de toute les paires d'agents côte à côte.

```
1 private boolean isJealous(Preference[] P1,Preference[] P2){
2     if (P1[0] == null){
3         return false;
4     }
5     Preference P2pref = null;
6     Preference P1pref = null;
7     for (Preference pref:P1){
8         if(pref.getSelectedby() != -1){
9             P1pref = pref;
10        }
11    }
12    for(Preference pref:P2){
13        if(pref.getSelectedby() != -1){
14            P2pref = pref;
15        }
16    }
17    if(P1pref == null || P2pref == null){
18        return false;
19    }
20    for(Preference pref:P2){
21        if(pref.getValue() == P1pref.getValue()){
22            if(pref.getPos().getCol() < P2pref.getPos().getCol()){
23                return true;
24            }
25        }
26    }
27    return false;
28 }
```

FIGURE 9 – Fonction de détection de la jalousie

Une fonctionnalité de sauvegarde était nécessaire pour l'application. En effet lorsqu'un joueur commence un niveau sans finir de le résoudre il était d'une part important de conserver le temps passé sur ce niveau et pour rendre l'expérience plus agréable, de sauvegarder l'avancement dans la résolution. Un tel système a été implémenté à l'aide du système de **SharedPreferences** présent dans les applications Android. Il permet d'enregistrer des données persistantes et ainsi de garder en mémoire certaines informations même si l'application est quittée/fermée.

3.4. RÉCUPÉRATION DES DONNÉES

Une fois l'application mobile réalisée, il nous fallait la faire tester sur un public de grande taille et offrant une certaine diversité pour recueillir des données pertinentes. Pour pouvoir

réaliser l'apprentissage de difficulté décrit en Section 2, il était demandé de pouvoir sauvegarder des données de résolutions sur une base de donnée distante. Le processus devait être automatique et silencieux de façon à simplifier au plus l'utilisation de l'application. Les serveurs Google et plus précisément le service GoogleSheets a été choisi pour cette tâche.

Nous avons donc intégré dans l'application mobile, une solution permettant l'envoi de diverses données utilisateurs qui étaient enregistrées lors de la résolution d'une instance. L'API Google Sheets v4 permet l'écriture/lecture de données sur une feuille de calcul sur différentes plateformes : application mobile, site web, programme python, etc. La communication entre la feuille et notre application est géré et sécurisée via l'API. En effet elle 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. L'application demande alors la permission d'interagir avec la feuille sous le compte de l'utilisateur. Nous avons donc opté pour la seconde solution qui use d'un compte de service dédié à notre application. Celle-ci interagira via ce compte sans demander la confirmation de l'utilisateur, ce qui est crucial pour favoriser le côté « easy to use » de l'application. L'utilisateur est plus à même de vouloir tester notre application sur une courte durée/longue durée. Quelque soit la méthode, un fichier JSON contenant les identifiants est créé.

L'intégration de cette solution dans notre application se traduit donc par deux classes : `SheetsServiceUtil` et `GoogleSheetsWriteUtil`. La première contient une unique méthode `getSheetsService` qui permet la création de l'objet `Sheet`. C'est l'intermédiaire pour écrire et lire avec l'API. Cet objet contiendra les identifiants nécessaire à l'autorisation de la communication entre l'application et la feuille.

La deuxième classe `GoogleSheetsWriteUtil` contient toutes les méthodes qui seront appelées pour l'écriture de nos données. D'abord la méthode `setup` permet la création d'un objet `Credential` à partir de notre fichier JSON. Cette instance est ensuite utilisée dans la méthode `getSheetsService`.

Puisque les opérations réseaux sont asynchrones par nature, Android prohibe ce type d'action sur le thread principal de l'application. Ce thread est généralement dédiée à la mise à jour et affichage de l'interface. C'est pourquoi Android propose une classe `AsyncTask` utile pour effectuer des actions courtes sur un thread séparé. C'est la solution que nous utilisons pour envoyer nos données. Cela reste invisible à l'œil de l'utilisateur ce qui est à nouveau crucial pour le côté agréable et fluide de son expérience.

Nous avons donc trois classes dans `SheetsServiceUtil` qui héritent toutes de la classe `AsyncTask` :

- `WriteUserInfoAsyncTask`,
- `ModifyUserInfoAsyncTask`,
- `WriteUserEvalAsyncTask`.

Ces trois classes implémentent ainsi une méthode `doInBackground`, qui prend en paramètre une liste de chaîne de caractère, les données à envoyer, et qui réalise l'appel à l'API en tâche de fond. Nous appelons donc à l'intérieur les méthodes de l'API de Google Sheets pour mettre à jour (`ModifyUserInfoAsync`) ou écrire de nouvelles données (`WriteUserEvalAsyncTask` et `ModifyUserInfoAsyncTask`) dans notre feuille. Le code visible en Figure 10 montre un exemple d'utilisation de l'API.

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 la planification d'un renvoi plus tard. Il existe donc pour cela deux

```

1 private static class WriteUserInfoAsyncTask extends
2 AsyncTask<String, Void, Void>{
3     @Override
4     protected Void doInBackground(String... data) {
5         ValueRange body = new ValueRange()
6             .setValues(Arrays.<List<Object>>asList(
7                 new List[]{Arrays.asList(data)}
8             ));
9         try {
10             AppendValuesResponse result = sheetsService.spreadsheets()
11                 .values().append(SPREADSHEET_ID, "User_Info!A1", body)
12                 .setValueInputOption("USER_ENTERED")
13                 .execute();
14             Object userPos = result.getUpdates().get("updatedRange");
15             prefs.edit().putString("userPos", (String) userPos).apply();
16         } catch (IOException e) {
17             Log.w("GoogleSheet", "Error_during_user_info_write;
18             rescheduling");
19             failedUserInfo = data;
20             e.printStackTrace();
21         }
22         return null;
23     }
24 }

```

FIGURE 10 – Code permettant l'écriture des informations utilisateur

variables listes dans la classe `SheetsServiceUtil` : `failedEvaluation` si l'écriture des données relative à une résolution échoue et `failedUserInfo` si l'écriture/modification du profil utilisateur échoue.

Avant toute nouvelles écritures, il faut donc vérifier qu'il n'existe pas de données non envoyées. Pour cette raison, les trois méthodes gérant l'envoi de donnée font cette vérification. `writeUserInfo`, `modifyUserInfo` et `writeUserEval` appellent toutes, au début de leur exécution, une méthode `checkFailures` qui vérifie que les listes d'échecs soient bien vides. Si elles ne le sont pas, alors on crée de nouvelles `AsyncTask` pour renvoyer ces données avant de procéder à l'envoi des nouvelles données.

```

1 public void writeUserEvaluation(String... data) {
2     this.checkFailures();
3     new WriteUserEvaluationAsyncTask().execute(data);
4 }

```

FIGURE 11 – Méthode `writeUserEvaluation`

4. CONCLUSION

RÉFÉRENCES

- [1] Khulood ALYAHYA et Jonathan E. ROWE. « Simple Random Sampling Estimation of the Number of Local Optima ». In : *Parallel Problem Solving from Nature – PPSN XIV*. Sous la dir. de Julia HANDL et al. Cham : Springer International Publishing, 2016, p. 932-941. ISBN : 978-3-319-45823-6.

- [2] Aurélie BEYNIER et al. « Local Envy-Freeness in House Allocation Problems ». In : *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018)*.
- [3] Josselin GARNIER et Leila KALLEL. « Efficiency of Local Search with Multiple Local Optima ». In : *SIAM Journal on Discrete Mathematics* 15.1 (2001), p. 122-141. DOI : 10.1137/S0895480199355225. eprint : <https://doi.org/10.1137/S0895480199355225>. URL : <https://doi.org/10.1137/S0895480199355225>.
- [4] Erik PITZER et Michael AFFENZELLER. « A Comprehensive Survey on Fitness Landscape Analysis ». In : *Recent Advances in Intelligent Engineering Systems*. Sous la dir. de János FODOR, Ryszard KLEMPOUS et Carmen Paz SUÁREZ ARAUJO. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 161-191. ISBN : 978-3-642-23229-9. DOI : 10.1007/978-3-642-23229-9_8. URL : https://doi.org/10.1007/978-3-642-23229-9_8.
- [5] Peter F. STADLER. « Landscapes and their correlation functions ». In : *Journal of Mathematical Chemistry* 20.1 (mar. 1996), p. 1-45. ISSN : 1572-8897. DOI : 10.1007/BF01165154. URL : <https://doi.org/10.1007/BF01165154>.

A. LEF SOLVER

B. DIAGRAMME DE CLASSE D'EQUITY