

Rapport de Projet Tuteuré - ANDROIDE

Alexandre Bontems, Gualtiero Mottola, Hans Thirunavukarasu

20 mai 2018

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	5
2.4	Analyse de Fitness Landscape	7
2.5	Apprentissage de la difficulté	7
3	Application mobile	7
3.1	Composants Android	8
3.2	Structure de l'application	8
3.3	Le modèle du jeu	8
3.4	Design interface	10
3.5	Interface GameView	11
3.6	Récupération des données	11
4	Conclusion	14

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 **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 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.

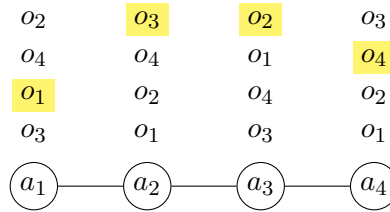


FIGURE 1 – Exemple d'instance étudiée du problème LEF

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 leur configuration.

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}

```

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. 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 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 $a_{i+1} \rightarrow a_{i+2} \rightarrow \dots$ pour trouver une solution puis dans la direction $a_{i-1} \rightarrow a_{i-2} \rightarrow \dots$ pour tenter d'en trouver une autre.

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 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 L'algorithme de backtracking permet de trouver l'ensemble de ces solutions pour une instance. Ce sont, d'après notre hypothèse de choix des objets par un joueur humain (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. Évidemment moins il existe de solutions, plus le temps moyen de recherche augmente et on devrait

donc voir apparaître une corrélation directe entre la difficulté ressentie et le nombre de solutions Pareto-optimales.

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,
- Le regret moyen dans les solutions Pareto-optimales,
- Le regret minimum pour les agents en extrémité puisque ce sont eux qui sont choisis en premier le plus fréquemment.

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 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.

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

	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

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 ci-dessous.

```

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.
```

Pour 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 suivante.

```

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 0, on définit l'indice p dans
7 % la liste de préférence:
8 position(A, 0, p).
9 [...]
```

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** 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" 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 est 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 la 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 ?, ?. Le but est souvent de jauger la difficulté de résolution d'une instance pour un algorithme donné et de pouvoir ainsi déterminer quel algorithme ou composante d'algorithme est la plus à même de résoudre le problème rapidement. Par la suite on considère le problème d'optimisation combinatoire qui correspond à la relaxation du problème LEF. Une fonction de fitness $f : \text{solution} \rightarrow \mathbb{N}$ évidente est alors le nombre d'agents jaloux dans une allocation donnée et le problème consiste en la minimisation de cette fonction.

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.*

Les mesures qui suivent se basent sur la méthode de résolution de l' **Observation 3**. On peut définir une fonction de distance entre deux solutions $d : s_1 \times s_2 \rightarrow \mathbb{N}$ qui correspond au 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 :

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

Bassin d'attraction Depuis les optima trouvées grâce au programme ASP, il est possible de calculer leur bassin d'attraction. Il convient tout d'abord de définir la notion de « chemin descendant » : un tel chemin est une séquence de solutions candidates présentant chacune une fitness moins grande que la précédente (comme définit dans ?, 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 :

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

La taille d'un tel bassin donne une idée de la probabilité de convergence vers un optimum depuis une solution candidate quelconque.

Définition 2. *La notion de **Landscape Walk** ? 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 est choisie pour poursuivre l'exploration.

Auto-corrélation

Fitness Distance Correlation

2.5. APPRENTISSAGE DE LA DIFFICULTÉ

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 l'utilisation de l'application par les utilisateurs. Ces données devaient pouvoir être exportées vers une base données Google Sheets.

3.1. COMPOSANTS ANDROID

Une application Android s'organise en plusieurs composants, nous allons en présenter certains dans la Section 3.2 pour pouvoir décrire l'application et ses fonctionnalités aisément. Nous allons notamment nous concentrer sur le concept d'activité, nécessaire pour pouvoir présenter 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 (par exemple, lancer un niveau). Le layout, chargé par une activité à sa création, est décrit dans un fichier XML.

3.2. STRUCTURE DE L'APPLICATION

Notre application est composée de six activités que nous allons présenter ici. Des impressions d'écrans sont visibles en Figure 2.

- **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 quatre 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.

3.3. 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

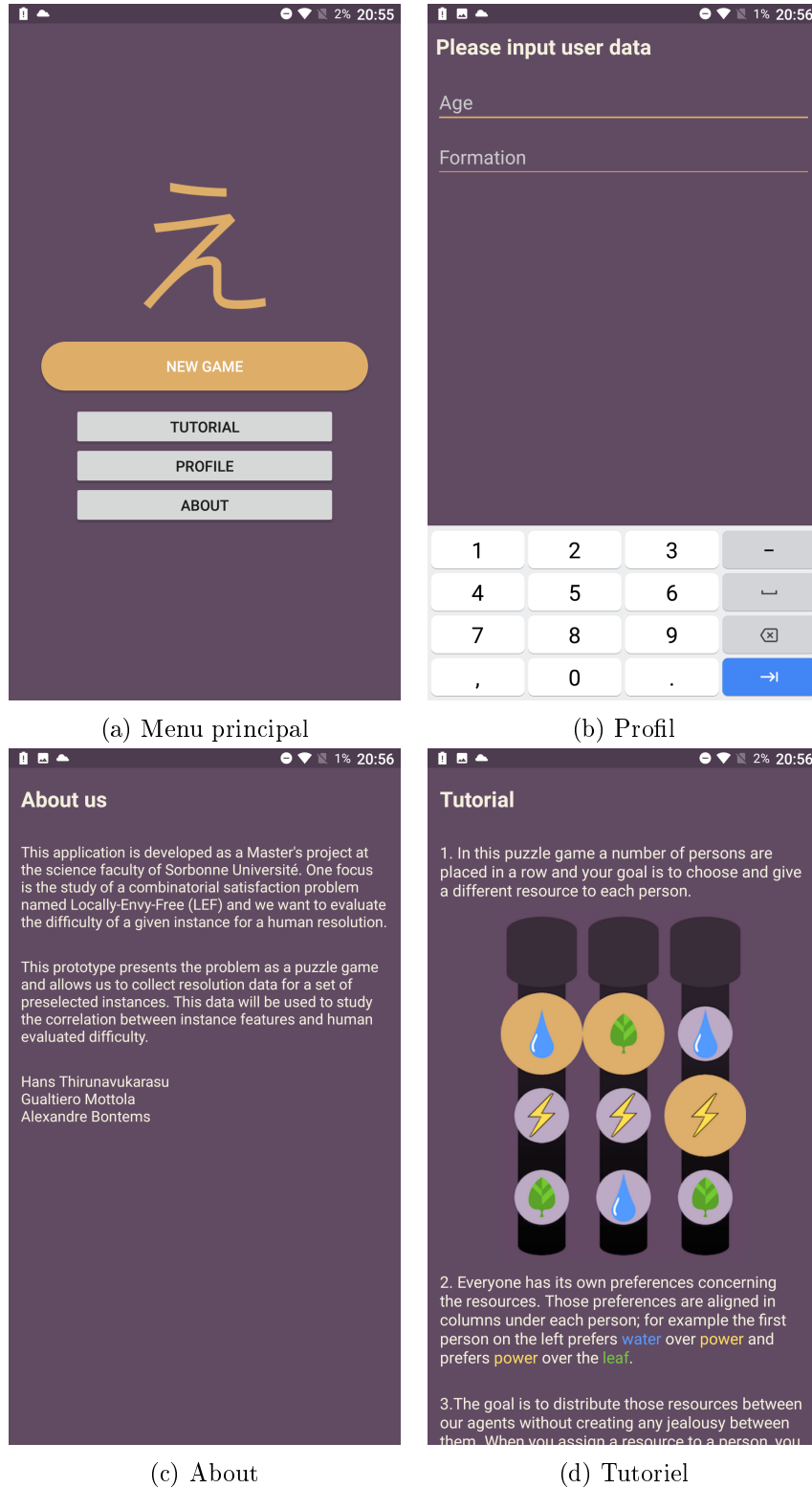


FIGURE 2 – Design final des menus de l'application

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`) : L'interface `IPiece` est composé de deux méthodes, la méthode `getId` qui retourne l'identifiant de la pièce et la méthode `getPosition`

qui 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**) : La classe Actor implémente l'interface IPiece et toute ses méthodes, elle représente les agents du jeu.
- définition d'une Préférence (Classe **Préférence**) : La classe Préférence implémente également l'interface IPiece, on y ajoute cependant là deux entier, valeur : 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 quel agent elle l'est.

```
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 }
```

Fonction de détection de la jalousie

3.4. DESIGN INTERFACE

L'interface centrale de l'application, l'aire de jeu de 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.

Il a été déterminé très tôt dans le projet qu'une visualisation sous forme de grille serait nécessaire, chaque case de la grille représentant une préférence d'un des agents. Il a ensuite été décidé que les agents, représentés par des carrés, serait placé sur la gauche de la grille et leur préférences, représenté par des cercles, sur la même ligne que l'agent avec son objet préféré sur la gauche le plus proche de lui.

Il a ensuite été décidé que pour afficher la sélection d'une préférence par un des agents, un cercle de la couleur de l'agent serait dessiné derrière le cercle représentant la préférence en question, l'utilisateur peut donc choisir de sélectionner une certaine préférence en touchant l'icône correspondante. On note que le processus de sélection des ressources empêche de sélectionner plusieurs ressources pour un seul agent. A ce stade l'application est fonctionnelle.

La dernière modification structurelle de l'IHM consiste à optimiser l'espace qui est alloué à l'affichage des préférences des agents : L'affichage de la grille de jeu a été pivoté pour que les agents se trouvent en haut de l'écran, l'alignement des préférences se faisait donc de haut en bas, ce changement a donc permis d'utiliser une plus grande partie de la hauteur de l'écran du smartphone. En effet cette rotation de la grille a permis d'économiser une colonne en ajoutant une ligne ce qui est plus adapté à l'affichage en mode portrait.

les modifications suivantes de l'activité ont été soit esthétiques soit relevant de l'ajout de feature : tout d'abord l'interface a suivis un redesign complet, une harmonisation des couleurs et des formes pour la représentation de la plupart des éléments à dessiner. L'ajout de barres verticales pour chacun des agents, ce qui facilite la lisibilité et l'ajout de l'identifiant du niveau.

Après avoir testé cette interface nous avons décidé que pour améliorer l'expérience de jeu nous devons rendre la détection d'affectation erroné plus aisée. Nous avons donc implémenté les solutions suivantes : l'affichage d'une couleur différente sur les ressources sélectionnés par deux agents ce qui permet au joueur de remarquer immédiatement si une assignation des ressources est impossible. La possibilité d'afficher en rouge les colonnes des agents jaloux, notons que cette fonctionnalité facilite grandement le jeu, elle est donc désactivé par défaut et le joueur a la possibilité d'appuyer sur un bouton pour activer cet affichage.

Ces deux fonctionnalités facilitent grandement le gameplay mais nous pensons que ce compromis permet une expérience de jeu plus fluide et agréable pour l'utilisateur, un point qui nous semble essentiel pour la distribution de l'application.

3.5. 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 suivante : les agents sont placés 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 alignés les préférences

3.6. 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 basées 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 avons donc décidé de la faire tester sur un public de taille variée et diverse, pour recueillir assez de données. Mais nous avons 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.

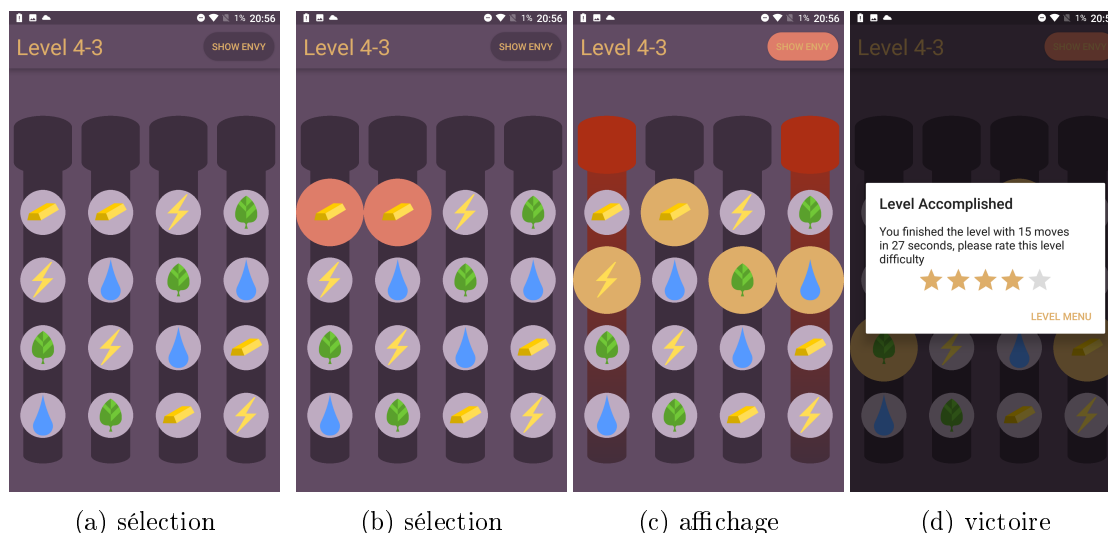


FIGURE 3 – Exemple Gameview

Googlesheets Nous avons donc intégré dans l'application mobile, une solution permettant l'envoi 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.getRessources()` ;). 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 récupérerons 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'œil 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
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_2!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 }
```

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 renvoi 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 vérifie si `failedEvaluation` et `failedUserInfo` sont vides 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.

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

Méthode `WriteUserEvaluation`

4. CONCLUSION