# Vote Itéré de Comité

# ElectiSim

## Rapport de Projet


Par les étudiants

Loic Urien, Victor Racine et Yasmine Hamdani


pour

M. Nicolas Maudet et M. Paolo Vappiani



Projet-ANDROIDE

Master informatique, parcours ANDROIDE

Faculté d'ingénierie - Sorbonne Université

27 Mai 2018

# Résumé

Lors d'un choix collectif, il est intéressant de permettre aux agents de modifier leurs votes en fonction de l'évolution des résultats, et leur offrir ainsi la possibilité d'effectuer des choix stratégiques basés sur des compromis afin de tirer partie de la dynamique du processus d'élection et d'augmenter leur satisfaction finale. Ce type d'élection fait ressortir l'influence de l'opinion générale sur le choix d'un votant, comme c'est le cas lors des élections présidentielles où les citoyens se réfèrent aux sondages pour évaluer les chances de leurs favoris et voter en conséquence. Ce système s'avère également utile dans des plateformes telles que Facebook ou Doodle qui permettent aux utilisateurs de modifier leurs réponses à tout moment pour arriver à un consensus.

C'est dans ce contexte que s'inscrit notre projet, *vote itéré de comités*, ayant pour objectif d'implémenter un simulateur d'élections à plusieurs tours mettant en scène des agents avec des préférences initiales sur les candidats qui tentent tout au long du processus de "manipuler" les élections afin de faire élire un comité dont ils seront le mieux satisfaits. Cet outil vise avant tout à aider la recherche dans ce domaine.

# Table des matières

# 1. Etat de l'art

## 1.1. Introduction

Dans cette partie nous présentons le fonctionnement des votes itérés. Nous commençons par introduire le sujet en section 1.2, puis nous décrivons les deux principales caractéristiques des votes itérés sont ensuite exposées en 1.3 et 1.4.

## 1.2. Vote itéré de comité

Les votes itérés relèvent du domaine du choix social computationnel où sont étudiés les problèmes de décisions collectives d'un point de vue computationnel[1]. Cet axe de recherche dérive de la théorie du choix social, dont fait partie la théorie du vote et qui s'intéresse à l'étude de l'agrégation de préférences individuelles pour atteindre un choix collectif. L'introduction de l'informatique à ce domaine de recherche est arrivé dans le début des années 2000s et a permis l'élaboration d'algorithmes déterministes et approchés pour traiter les problèmes de choix sociaux et mieux les analyser. Les applications de cette théorie en informatique se sont avérées multiples, notamment dans les systèmes de décisions dans un cadre multi-agents homogène ou hétérogène ou encore dans les systèmes de recommandation groupées.

Au cours d'un vote itéré[1], les agents commencent par voter selon leurs préférences individuelles. Un candidat, ou un comité dans le cas de l'élection d'un comité, est ensuite élu en appliquant la règle de vote spécifiée et les résultats sont communiqués aux agents en fonction de leur niveau d'information, cela peut aller d'une simple énumération du comité élu jusqu'aux scores détaillés de tous les candidats/comités. Chaque agent applique alors sa stratégie de vote afin d'évaluer ses votes possibles selon les résultats récents, ses préférences initiales et sa fonction d'utilité. Il choisit ensuite, lorsqu'il existe, le vote qui améliore potentiellement son utilité, ce vote pouvant être basé sur différentes heuristiques. Dans le cas où il ne dispose pas de "coup améliorant", il conserve le même vote que celui fourni à l'instant *t-1*. De nouveau, la règle de vote élit un vainqueur et les agents sont mis au courant et votent en appliquant leurs stratégies, et ainsi de suite. L'élection s'arrête lorsqu'on atteint un équilibre de Nash, c'est à dire qu'aucun agent n'a intérêt à changer de vote. Un exemple de vote est expliqué dans la section 2.5.1.

Par soucis de convergence, la plupart des études sur les votes itérés portent sur des configurations où seul un agent est autorisé à modifier son vote à un instant *t*. Autrement dit, les cas de votes simultanés sont généralement évités [2][3][4].

Un mécanisme de vote itéré est donc défini par une élection spécifiant la configuration du vote qui donne les votants et les candidats, une règle de vote permettant d'élire le candidat (ou comité) vainqueur à chaque itération, et la stratégie suivie par les agents pour mettre à jour leurs votes. Nous noterons que nous partons du principe que tous les agents suivent la même stratégie. Bien qu'il puisse éventuellement être intéressant que les agents puissent voter en utilisant des stratégies variées, ceci sort du cadre de ce projet.

Une élection $E$ est définie[5] comme un couple $(A, R)$ où $A$ représente l'ensemble des candidats et $R$, le profil, énumère les préférences des votants sous forme de listes ordinales sur les candidats, qui est généralement un ordre total $\succ_v$ de l'agent $v$ sur les candidats, mais il peut également être partiel dans certains cas.

## 1.3. Règle de vote

Dans le cas de l'élection d'un comité, une règle de vote ou fonction de choix social[1][2] est une fonction qui prend en entrée une élection $E$ et un entier $k$ strictement positif $0 < k < |A|+1$, et qui retourne l'ensemble des comités de taille $k$ élus ex-aequo. Elle peut avoir recours à des procédés de tie-breaking afin de les départager et de n'en tirer qu'un seul vainqueur. En général, les procédés choisis sont déterministes (par exemple suivant l'ordre lexicographique) afin de réduire les cas de non-convergence[4][5].

Plusieurs règles de votes existent et diffèrent selon l'objectif de l'élection. On se pose alors la question de savoir si par exemple, l'on cherche à élire un comité constitué des "élites" ou bien un comité plus représentatif des minorités présentes dans la société.

Les règles de votes dépendent de la configuration de celui-ci, notamment de la manière dont les agents votent. Il existe trois principaux types de règles de votes de comités[6] : les règles attribuant des scores aux comités, les règles basées sur la notion d'approbation et les règles suivant le principe de Condorcet.

- **Règles basées sur les scores des comités :** Elles associent via une fonction d'utilité $f$ à chaque comité (une séquence croissante de $k$ candidats élus) un score de telle sorte à ce que si un comité $I$ domine un comité $J$, alors $f(I) >= f(J)$, on considère qu'un comité $I=(i_1, i_2, \ldots, i_k)$ domine un comité $J=(j_1, j_2, \ldots, j_k)$ si pour tout $t \in \{1, 2, .., k\}$ $i_t <= j_t$. La plupart se basent sur des adaptations de fonctions d'attribution de scores appliquées pour des élections à un seul gagnant, notamment *k-approval* qui attribue un point aux $t$ candidats préférés d'un agent et *Borda* où le score d'un candidat attribué par un agent correspond au nombre de candidats qui viennent après lui dans les préférences de cet agent.

Quelques exemples de fonctions *f* connues :

- ➢ **Bloc :** chaque votant révèle ses *k* candidats favoris et les *k* candidats cités le plus souvent sont élus.
- ➢ ***k*-Borda :** élit les comités avec les *k* candidats ayant les meilleurs scores de Borda.
- ➢ **Single Non-Transferable Vote (SNTV) :** un comité reçoit un point d'un votant si il contient le candidat préféré de ce votant.
- ➢ **Chamberlin-Courant (ß-CC):** chaque votant attribue à un comité un score correspondant au score de Borda du candidat qu'il préfère parmi ceux du comité.

On remarquera que les règles Bloc et K-Borda cherchent à élire un comité composé des *k* candidats préférés tandis que SNTV et Chamberlin-Courant tentent plutôt de sélectionner les candidats supportés par la majorité, comme c'est le cas lors des élections législatives par exemple.

- **Règles basées sur l'approbation :** sont destinées à des élections où les agents communiquent l'ensemble des candidats qu'ils approuvent. Parmi elles on cite :
  - ➢ **Approval Voting (AV) :** retourne les comités formés des *k* candidats cités le plus souvent.
  - ➢ **Approval-Based Chamberlin-Courant rule :** un comité reçoit le vote (un seul point) d'un agent s'il contient au moins un candidat qu'il approuve.
  - ➢ **Proportional Approval Voting (PAV) :** cherche à maximiser la somme des satisfactions des agents. La satisfaction d'un agent est calculée par la formule $1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$, n étant le nombre de candidats du comité approuvés par l'agent.

- **Règles basées sur les vainqueurs de Condorcet :** un comité *C* est un comité au sens du Condorcet si la majorité des votants le préfère à chacun des autres comités *D*. Lorsque les votants ont des préférences sur les candidats, il faut trouver un mécanisme pour se ramener à des préférences sur les comités (par exemple dans le modèle k-approval, on considère qu'un votant préfère le comité *C* s'il contient plus de candidats approuvés que le comité *D*. Dans le modèle ordinal, le comité *C* devrait avoir un score de Borda -correspondant à la somme des scores de Borda de ses membres- supérieur à celui de *D*). Parmi les règles :
  - ➢ **Number of External Defeats (NED) :** le score d'un comité *S* correspond au nombre de couples de candidats (*c,d*) tels que pour $c \in S$ et $d \in A \backslash S$, au moins la moitié des votants préfèrent *c* à *d.* Le comité ayant le plus grand score est élu.
  - ➢ **Minimum Size of External Opposition (SEO) :** le score d'un comité *S* correspond au plus petit nombre de votants qui préfèrent un des membres du comité *S* à un candidat qui n'en fait pas parti. Le comité ayant le plus grand score est élu.

## 1.4. Stratégies des agents

Une stratégie de vote[3] est une fonction qui, étant données les préférences d'un agent, les résultats de l'élection à l'instant *t-1* et une règle de vote, fournit le vote de cet agent à l'instant *t*. Elle dépend fortement du niveau de connaissance des agents qui

spécifie les informations concernant les résultats du vote auxquelles ils ont accès. Les alternatives au vote précédent sont évaluées en appliquant une fonction d'utilité sur le comité élu par la règle de vote en lui passant en paramètre le nouveau vote de l'agent et les votes fournis à l'instant *t-1* des autres agents.

En général, on considère qu'à la première itération les agents votent "honnêtement", c'est à dire qu'ils ne cherchent pas à tromper les autres agents en communiquant un vote qui ne correspond pas à leurs préférences réelles.

De plus, les agents sont souvent considérés comme étant myopes[2]; ils votent à chaque itération comme si c'était la dernière et cherchent donc à obtenir un résultat immédiat au lieu de planifier une stratégie qui pourrait donner des résultats intéressants sur le long terme. On suppose également que les agents sont rationnels[2] dans le sens où ils cherchent à maximiser leur fonction d'utilité. Ils ne changeront donc de vote à un instant *t* que si cela permet d'accroître strictement cette utilité. Il est cependant à noter que ce ne sont en aucun cas des restrictions imposées par notre application, nous avons cependant préféré nous concentrer sur des stratégies de la forme décrite précédemment.

Les stratégies les plus courantes sont les stratégies dites *Better Response* et *Best Response.* Lorsqu'un agent suit la première stratégie, il vote pour le premier comité considéré améliorant sa fonction d'utilité, tandis que la seconde oblige l'agent à explorer tous les votes possibles et à choisir la meilleure alternative. Ces stratégies se basent sur les préférences de l'agent et requièrent un minimum d'informations sur les derniers résultats (ce peut être le classement des candidats ou des comités, leurs scores, etc …). Il existe d'autres stratégies basées des heuristiques, notamment dans les cas des votes à un seul vainqueur[2], ou sur l'apprentissage des agents [7].

## 1.5. Conclusion

Le domaine de la théorie du vote est vaste, et les votes itérés de comité en sont une partie qui est loin d'avoir livré tous ses secrets. Cet état de l'art nous a permis de mieux cerner le problème posé et de réfléchir à une architecture intelligente pour concevoir notre application. De plus, bien que la documentation concernant les règles de vote soit exhaustive et que le domaine de recherche soit florissant, l'étude des stratégies de agents dans le contexte de votes itérés est encore nouveau, ce qui explique le peu de stratégies implémentées dans notre projet.

# 2. Projet ElectiSim

## 2.1. Introduction

Dans cette partie, nous nous intéressons au projet sur lequel nous avons travaillé. Nous commençons par décrire l'architecture logicielle établie puis nous présentons l'application implémentée et discutons quelques tests effectués.

## 2.2. Processus de conception

Une fois le cahier des charges réalisé, nous nous sommes mis à réfléchir à une manière de structurer notre application qui permettrait de répondre aux attentes de nos encadrants. Nous nous sommes d'abord intéressés au fonctionnement des votes itérés (cf état de l'art) afin d'avoir une idée générale des composantes clés de l'application et de leurs interactions. L'application étant destinée à être enrichie plus tard, son extensibilité s'est présentée comme un critère primordial que nous avons fait en sorte de respecter tout au long de notre travail. Ainsi, l'ajout futur de nouvelles configurations (typiquement de nouvelles règles de votes, stratégies mais aussi des niveaux de connaissances différents) devrait se faire sans difficulté.

L'architecture de l'application a été modélisée en utilisant le diagramme de classes tel que représenté dans la Figure 2.1. (une version plus nette est disponible dans le répertoire documents du projet).

## 2.3. Composantes de l'application

L'idée était donc de concevoir une base solide composée de plusieurs modules interagissant les uns avec les autres pour configurer, lancer et récupérer les résultats de simulations. On distinguera en particulier les packages suivants :

- org.upmc.electisim : le package principal, contenant tous les composants nécessaires au bon fonctionnement de l'application ainsi que les stratégies et les règles de vote.
- org.upmc.electisim.knowledge : les "knowledge provider", fournissant les connaissances aux agents à chaque itération du vote.
- gui.upmc.electisim : le package regroupant les éléments du GUI de l'application.

- org.upmc.electisim.utils : les différents utilitaires utilisés dans le code de l'application à l'intérieur de plusieurs packages.
- org.upmc.electisim.input : les différents utilitaires de lecture de fichiers.
- org.upmc.electisim.output : les différents utilitaires d'écriture de fichiers.

### 2.3.1. Configuration d'une simulation

La configuration d'une simulation passe par la spécification du profil de l'élection, c'est à dire la liste des candidats et les préférences des agents (cf section 1.2), de la taille du comité à élire, de la règle de vote et du niveau de connaissance des agents et de leurs stratégies. Dans l'architecture de l'application, cela se traduit par une classe "SimulationProfile" centrale à celle-ci :.

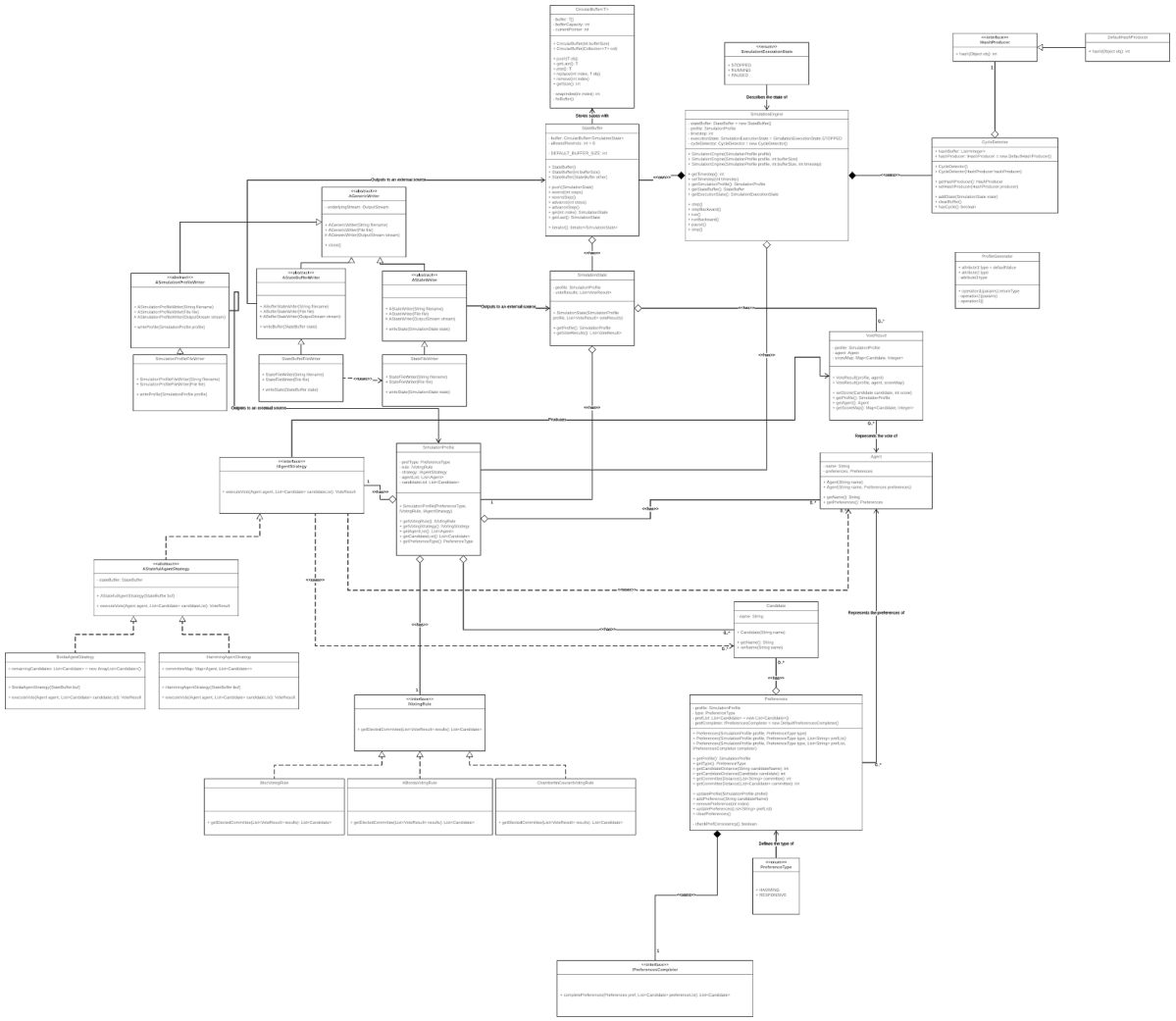


Figure 2.1 - Diagramme de classes de l'application

## Génération des préférences

Afin de permettre le lancement simulations mettant en scène plusieurs agents et candidats, nous avons permis la génération aléatoire de préférences des agents étant donnée la liste des candidats en lice. Bien entendu, l'utilisateur peut modifier les préférences générées ou les définir lui-même.

Deux types de préférences ont été implémentés : les préférences dites responsives qui correspondent un ordre total sur les candidats, et les préférences de Hamming qui suivent au modèle approval en considérant que chaque agent à un comité qu'il approuve. Une amélioration de notre modèle serait de modifier la génération des préférences de Hamming pour que les agents soient libres d'approuver autant de candidats qu'ils le souhaitent (pas forcément *k* candidats).

Il aurait été intéressant d'offrir aux utilisateurs la possibilité de paramétrer cette fonctionnalité pour pouvoir générer des préférences par paquets où certains candidats domineraient les préférences (avec des probabilités de tirage plus grandes), notamment en utilisant la méthode de Placket-Luce[8][9], ou encore de donner aux agents des préférences partielles. Nous avons commencé à travailler sur un "compléteur" de préférences mais il n'est pas encore fonctionnel, en l'état actuel, si un agent n'apparaît pas dans les préférences, le plus mauvais score possible lui est affecté.

## Niveau de connaissance

Le niveau de connaissance des agents est géré par les *Knowledge Dispenser*, des entités passées aux stratégies des agents qui permettent à ces dernières d'obtenir les informations auxquelles elles ont accès.

Nous avons implémenté quatre *Knowledge Providers*; l'un pour récupérer la règle de vote utilisée, un autre pour obtenir le classement des candidats/comités, un troisième communiquant les scores et le dernier qui est une agrégation des trois et qui fournit ainsi toutes les informations relatives aux dernières élections, c'est à dire la règle de vote utilisée, le comité élu ainsi que les scores de tous les candidats (dans le cas des règles de vote Bloc et K-Borda) ou tous les comités (dans le cas de Chamberlin Courant).

D'autres niveaux de connaissances peuvent être ajoutés facilement en se basant sur ceux implémentés et des classes permettent de limiter le niveau de connaissance d'une stratégie.

Le package "org.upmc.electisim.knowledge" regroupe tout les "providers" déjà implémentés

## Stratégie des agents

Nous avons implémenté une seule stratégie de type *Best Response* (qui évalue toutes les alternatives (cf. section 1.4) avec niveau de connaissance total. Elle explore les votes possibles et renvoie son meilleur vote qui n'est autre qu'un ordre linéaire sur les candidats.

Concrètement, elle initialise son vote à celui effectué à l'itération précédente. Elle génère ensuite toutes les alternatives possibles, lance la règle de vote pour chacune d'entre elles en supposant que les autres agents ne changent pas leurs votes et récupère ainsi les comités élus dans chaque cas. Elle estime ensuite la satisfaction, ou plutôt l'*insatisfaction* de l'agent en calculant la distance de Hamming de chaque comité élu par rapport aux préférences de l'agent, ce qui revient à faire la somme des distances de Hamming de chaque candidat du comité, sachant que le score de Hamming d'un candidat correspond à sa position dans le classement des préférences de l'agent. Lorsqu'un comité obtient un score de satisfaction supérieur à celui du vote initial de l'agent, la stratégie le sélectionne. Sinon, c'est l'ancien vote qui est gardé.

La stratégie telle que nous l'avons implémentée prend en compte toutes les permutations possibles de taille $k$ et ordonne les $m$-$k$ candidats restants selon les préférences de l'agent car cela suffit à déterminer le meilleur vote et réduit le temps de traitement.

### Règle de vote

La règle de vote prend en paramètre les votes des agents (liste d'ordres totaux sur les candidats) et retourne les scores des candidats/comités ainsi que le comité élu.

Les règles de votes que nous avons implémentées sont Bloc, K-Borda et Chamberlin Courant. Chacune commence par traduire les votes en scores sur les candidats. Bloc attribue simplement un point au $k$ premiers candidats de chaque vote ($k$ étant la taille du comité à élire), K-Borda et Chamberlin Courant les traduisent en scores de Borda. Elles somment ensuite les points de chaque candidat pour obtenir leurs scores finaux. A partir de là, chacune applique l'algorithme qui lui est propre : Bloc et K-Borda élisent les $k$ candidats ayant les plus hauts scores et Chamberlin Courant génère et évalue tous les comités possibles (rappelons qu'un agent attribue à un comité reçoit le score de Borda du candidat qu'il supporte le plus dans ce comité) et retourne celui ayant le score le plus élevé. L'ordre lexicographique permet de trancher dans les cas d'égalités.

## Importation et exportation des configurations

Les configurations peuvent s'effectuer via l'interface graphique, mais il est également possible de les sauvegarder sous format json afin de les recharger plus tard. Nous avons choisi ce format car il permet de décrire de manière claire et explicite les informations, il est de plus facilement extensible et est un format excessivement commun sur le web.

### 2.3.2. Exécution d'une simulation

Les composantes responsable du lancement de la supervision de la simulation sont principalement le *moteur* et le *buffer.*

#### Moteur de la simulation

Représenté par la classe "SimulationEngine", il est le *coeur* de l'application. Pour une configuration donnée, il est responsable du lancement et de la coordination de la simulation : il transmet aux stratégies des agents les résultats de l'itération précédente et récupère leurs nouveaux votes puis exécute la règle de vote et sauvegarde le résultat dans le buffer.

Il permet également l'avancement pas à pas (une itération à la fois) et le retour en arrière en chargeant un état précédent du buffer.

#### Suivi de la simulation

Le simulateur permet de naviguer entre les états de la simulation, c'est à dire le retour en arrière et l'avancement pas à pas. Ceci est réalisable grâce à un buffer circulaire contenant les *n* derniers états de la simulation, *n* étant un paramètre ajustable par l'utilisateur.

Un buffer circulaire correspond à une liste dont la fin est reliée au début. Lorsque la taille maximale est atteinte, les nouveaux éléments ajoutés remplacent les plus vieux.

#### Détection des cycles

La convergence des règles de votes n'étant pas assurée, il nous a été demandé d'implémenter un détecteur de cycles afin de repérer des *pattern* dans le comportement des agents. Pour cela, nous avons ajouté deux modules : un buffer et un détecteur de cycles.

Le buffer diffère de celui utilisé pour le suivi de la simulation. D'abord, sa taille est plus grande pour prévoir des cycles potentiellement longs, dans l'implémentation actuelle sa taille est illimitée. Ensuite, les états y sont stockés sous forme de hash afin d'économiser l'espace. La fonction de hachage utilisée est paramétrable en utilisant une classe implémentant l'interface "IHashProvider". Ainsi, deux états de la simulation identiques, c'est à dire deux états où les agents ont voté de la même manière, ont nécessairement le même code de hachage. La détection de cycles revient alors à trouver des répétitions de ces chaînes dans le buffer et ce, d'abord en explorant la liste en sens inverse à chaque nouvel état de la simulation et en comparant tout les codes de hachage, et dans un second temps via l'algorithme de recherche de Boyer Moore[10] afin de

détecter les cycles strictement similaires et donc éventuellement une non convergence de la simulation.

### Sauvegarde des résultats

Les résultats de l'élection à un instant *t* peuvent être exportés dans fichier au format *Comma Separated Values (CSV)*. Cela permet de sauvegarder la table résumant les scores des candidats/comités ainsi que le comité élu.

## 2.3.3. Interface

L'interface permet de configurer une simulation (ou d'en charger une), de la lancer et d'afficher les résultats sous forme de graphe avec en abscisses les candidats (ou les comités lorsque la règle de vote est Chamberlin Courant par exemple, voir tout autre type d'entité éligible) et en ordonnées les scores obtenus.

La démarche suivie pour réaliser l'interface est la méthode de conception centrée utilisateur étudiée en cours d'Interface Homme-Machine[11]. Nous avons commencé par interviewer les futurs utilisateurs (en l'occurrence, nos encadrants) pour déterminer leurs besoins et établir un cahier des charges. Nous avons ensuite échangé nos idées et avons créé des prototypes papiers que nous leur avons montrés. Nous avons pris en considération leurs remarques afin de raffiner notre prototype et d'en faire une maquette finale qui leur a été présentée.

Le principal critère à respecter était que l'interface soit simple et intuitive. Les clients étant habitués à l'outil informatique, il n'y avait pas lieu de prendre en compte des scénarios d'utilisation extrêmes, typiquement des scénarios mettant en scène des utilisateurs âgés non habitués aux nouvelles technologies. L'outil visant à être utilisé par des chercheurs, il est de la plus haute importance que ces derniers puissent se concentrer sur l'activité de recherche tout en ayant un outil facile d'utilisation.

Afin de réaliser ces objectifs, nous avons adopté un patron de conception dit MVC, model-view-controller directement implanté dans la bibliothèque JavaFX.

## 2.4. Implémentation

L'implémentation a été réalisée en Java. Notre choix s'est porté sur ce langage d'abord car notre architecture est basée sur le modèle orienté objet. Ensuite, parce qu'il facilite la portabilité des applications et ce, quelque soit le système d'exploitation. Par ailleurs, tous les membres du groupe y ont déjà eu recours ce qui a facilité l'implémentation.

Nous avons utilisé l'outil *Maven* pour gérer notre projet. Il nous a permis d'unifier la configuration du projet, notamment en regroupant les bibliothèques utilisées.

L'interface a été implémentée en utilisant la bibliothèque JavaFX.

Comme l'application sera probablement amenée à être modifiée pour y ajouter de nouvelles configurations, nous avons généré une documentation claire et détaillée de notre programme en utilisant JavaDoc.

Ci-dessous un schéma de l'exécution d'une itération dans notre application :

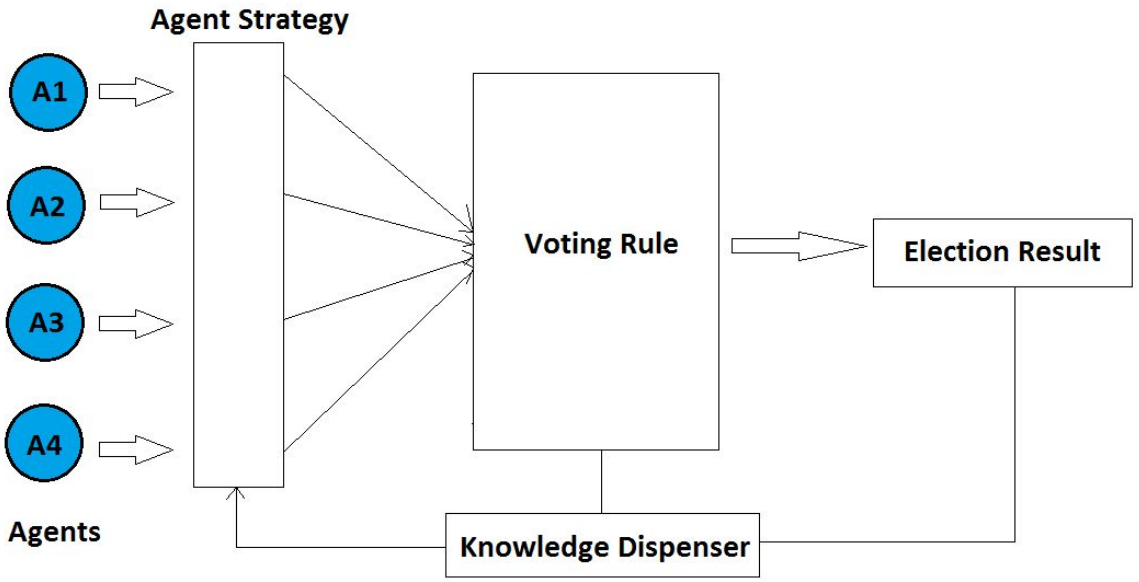



Figure 2.1 - schéma d'exécution d'une itération

Les agents ne sont ainsi utilisés que comme des "jetons" que la stratégie utilise pour émettre un vote (classe AgentVote). Une fois que la stratégie a consommé tous les agents, le simulateur agrège les votes et les envoie vers la règle de vote, une classe implémentant l'interface IVotingRule, qui produira un résultat d'élection (classe ElectionResult) qui contiendra les scores des différentes entités éligibles et le comité élu. Ce résultat sera ensuite utilisé par le "knowledge dispenser" pour les itérations suivantes, ainsi que la règle de vote potentiellement.

## 2.5. Tests

Une fois notre application fonctionnelle, nous avons effectué quelques tests, d'abord pour vérifier que l'exécution se faisait correctement mais aussi pour suivre le déroulement d'élections avec des profils différents. Nous commencerons par décrire le comportement des agents sur une petite instance puis nous présenterons quelques résultats obtenus.

### 2.5.1. Déroulement d'une instance

La simulation a été lancée sur une instance de *n=3* agents avec *m=4* candidats où on cherche à élire un comité de taille *k=2* avec des profils générés aléatoirement. La règle choisie est Bloc, la stratégie des agents est Best Response avec niveau de connaissance total.

| Itération | Votes | Scores des candidats | Comité élu | Insatisfaction (distance de Hamming) |
|---|---|---|---|---|
| i = 0<br>(les votes correspondent aux préférences des agents) | a1 : B>C>A>D<br>a2 : C>D>B>A<br>a3 : C>A>B>D | A : 1<br>B : 1<br>C : 3<br>D : 1 | [A,C] (par tie-break) | a1 : 3<br>a2 : 3<br>a3 : 1 |
| i = 1 | a1 : B>C>A>D<br>a2 : C>B>D>A<br>a3 : C>A>B>D | A : 1<br>B : 2<br>C : 3<br>D : 0 | [B,C] | a1 : 1<br>a2 : 2<br>a3 : 2 |
| i = 2 | inchangés | inchangés | inchangés | inchangés |

Tableau 2.1 - Déroulement d'une instance aléatoire

Les agents commencent par voter selon leurs préférences. La règle de vote Bloc traduit ces votes en affectant 1 point aux 2 premiers candidats de chaque vote et 0 aux autres. Elle somme ensuite les scores pour chaque candidat et sélectionne ceux ayant les meilleurs scores. Comme A, B et D sont à égalité, la règle de tie-break selon l'ordre lexicographique est appliquée et le comité [A,C] est élu.

A l'itération suivante, chaque agent évalue toutes les alternatives et vote pour celle lui offrant un indice d'insatisfaction inférieur à celui de l'itération précédente. a1 et a3 ne trouvent pas de meilleurs votes et conservent donc le même qu'à i=0. a2 cependant se rend compte qu'en interchangeant B et D dans son vote, il peut faire passer B devant A et élire le comité [B,C], réduisant ainsi son indice d'insatisfaction de 1.

A i=2, aucun agent ne trouve de coup améliorant, on a donc atteint un équilibre de Nash.

### 2.5.2. Quelques résultats

Les instances sur lesquelles notre programme a été testé sont de petites tailles afin de pouvoir suivre l'évolution des votes et les comités élus. Nous avons obtenu les résultats présentés dans le tableau ci-dessous :

| Instance | Préférences | Taille du comité | Bloc | K-Borda | Chamberlin Courant |
|---|---|---|---|---|---|
| Identiques | a1 : A>B>C>D<br>a2 : A>B>C>D<br>a3 : A>B>C>D | 2 | [A,B] | [A,B] | [A,B] (élu par tie-break) [A,C],[A,D] |
| 2 contre 1 | a1 : A>B>C>D<br>a2 : A>B>C>D<br>a3 : D>C>B>A | 2 | [A,B] | [A,B] | [A,D] |
| 2-symétrique | a1 : A>B>C>D<br>a2 : D>C>B>A | 2 | [A,B] (élu par tie-break) | ne converge pas, cycles de taille 6 | [A,D] |
| 4-symétrique | a1, a2 : A>B>C>D<br>a3, a4 : D>C>B>A | 2 | [A,B] (élu par tie-break) | [A,C] | [A,D] |
| 6-symétrique | a1, a2, a3 : A>B>C>D<br>a4, a5, a6 : D>C>B>A | 2 | [A,B] (élu par tie-break) | [A,D] | [A,D] |
| Circulaires | a1 : A>B>C>D<br>a2 : B>C>D>A<br>a3 : C>D>A>B<br>a4 : D>A>B>C | 2 | [A,B] (élu par tie-break) | [A,B] | [A,C] (elu par tie-break) [D,B] |
| Aléatoires | a1 : A>E>D>C>B>F<br>a2 : F>B>A>D>E>C<br>a3 : F>E>B>A>C>D<br>a4 : C>D>F>B>A>E | 3 | [A,B,F] | [A,B,D] | [F,C,A] |

Tableau 2.2 − Résultats obtenus sur les instances testées

En suivant l'évolution des différentes simulations nous avons remarqué un certain nombre de comportements intéressants :

- Chamberlin Courant cherche bien à élire les candidats supportés par la majorité tandis que les deux autres cherchent simplement les “meilleurs”, cela se voit bien sur l’instance “2 contre 1”.
- La manipulation des votes n’est pas évidente dans le cas Bloc car pour chaque candidat, un agent peut faire basculer le score au plus d’un point seulement. Sur la majorité des instances (y compris d’autres instances aléatoires) la convergence se faisait en une ou deux itérations.
- Dans le cas I-symétrique, à partir de I=6, les agents insatisfaits ne parviennent plus à améliorer leurs votes à partir de la 2e itération. Cela est lié au fait que les agents cherchent à améliorer leurs satisfactions individuellement. A *i*=0 les scores des agents sont tous égaux et le comité [A,B] est élu par tie break. Cela permet aux agents avec les préférences D>C>B>A de trouver un vote améliorant qui fait passer D devant B et parviennent ainsi à faire élire [A,D]. A partir de là, les autres agents ne trouvent pas un moyen de faire passer B devant, ni C (alors que pour I<6 c’était possible car chaque agent avait le poids nécessaire pour faire basculer l‘élection individuellement et ils parvenaient ainsi à faire élire leur comité préféré) et c’est donc le comité [A,D] qui est élu.
- En revanche, pour le même type d’instances avec la règle Bloc, le fait que d’une part, ce soit symétrique avec un nombre pair de candidats et que d’autre part, chaque agent vote en faveur de son comité préféré uniquement, restreint leur champ de manipulation. En effet, il y a exactement 50% des agents qui votent pour un comité et les 50% restants votent pour le complémentaire. Ainsi, aucun d’entre eux ne peut faire pencher les résultats d’un côté ou de l’autre et c’est la règle de tie break qui tranche.
- Chamberlin Courant converge toujours à la première itération car aucun agent ne parvient à manipuler les résultats obtenus en faveur de son comité préféré, et c’est tant mieux parce que plus le nombre de candidats augmente, plus les agents mettent du temps à évaluer leurs votes possibles. En effet, comme les stratégies sont amenées à lancer la règle de vote pour chaque permutation envisagée (soit chaque stratégie fait appelle à la règle $\frac{m!}{(m-k)!}$ fois, car nous ne testons que les préférences avec permutations sur les *k* premiers candidats et le reste est complété par ordre décroissant des préférences initiales de l’agent) et que Chamberlin Courant évalue toutes les combinaisons possibles de *k* candidats parmi *m,* le nombre d’alternatives envisagées croît très vite.
- Avoir des stratégies déterministes aide à la convergence des résultats. En effet, au cours de nos tests, il est arrivé qu’on modifie la stratégie pour qu’elle ne conserve pas son vote initial (pour être francs, c’était plutôt un oubli de notre part) et on a constaté que les instances convergeaient plus difficilement dans ce cas avec *Bloc* et *K-Borda*.
- Au cours de la simulation sur l’instance avec préférences circulaires, un état avec comme comité élu [A,C] (le même qu’avec Chamberlin Courant) est apparu. Cependant, ce n’était pas un équilibre de Nash et certains agents ont changé leurs votes à l’itération suivante. Finalement c’est le comité [A,B] qui a remporté l’élection avec un score d’insatisfaction allant de 1 (pour a1) à 5 (pour a3) tandis qu’avec [A,C] ça se situait plutôt entre 2 (a1 et a3) et 4 (a2 et a4).

- Le tie-break lexicographique n'est pas forcément le plus *juste*. En effet, sur l'instance 2-symétrique par exemple avec la règle Bloc, c'est le comité [A,B] qui est élu par tie-break. On voit bien que ce résultat est injuste vis-à-vis de l'autre agent qui se retrouve avec une insatisfaction de valeur 5 tandis que pour le premier elle est à 1. En choisissant le comité [A,D] on aurait pu répartir l'insatisfaction de manière plus homogène avec 3 pour chacun des deux agents. On pourrait donc penser à ajouter des tie-break qui tenteraient de minimiser l'insatisfaction max.

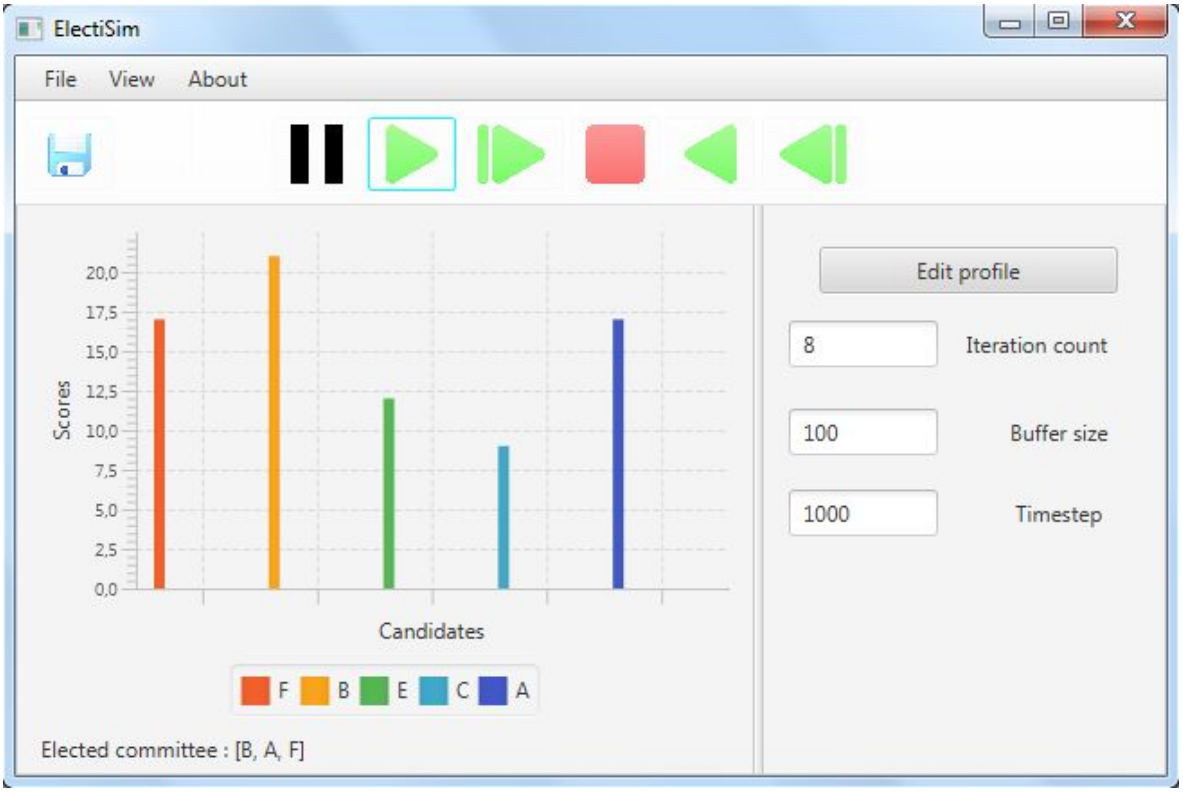



Figure 2.2 − Représentation graphique des résultats sur l'instance aléatoire

## 2.6. Conclusion

Nous sommes parvenus à implémenter une application complète permettant de configurer des simulations en combinant profils des votants, niveaux de connaissance, stratégies et règles de vote et de suivre leur évolution.

# 3. Conclusion générale

Notre projet, qui s'inscrit dans l'UE P-ANDROIDE, avait pour objectif de réaliser un simulateur pour les votes itérés de comité. Pour cela, nous avons suivi la démarche classique pour réaliser une application.

Nous avons d'abord discuté avec nos encadrants, M. Nicolas Maudet et M. Paolo Viappiani afin de comprendre l'idée générale du projet et les objectifs à atteindre. Cette phase de spécification des besoins a permis l'élaboration du cahier des charges qui nous a servi de référence tout au long de notre travail. Nous sommes ensuite passés à l'étude de l'existant en épluchant quelques articles sur le sujet, dont une partie nous a été communiquée par nos encadrants. Une fois les concepts de base du processus de vote acquis, nous nous sommes lancés dans la phase de conception en prenant garde à permettre l'extensibilité future de l'application qui était l'un des critères les plus importants du cahier des charges. Ainsi, nous avons modélisé l'architecture de base avec un diagramme de classes. En partant de cette base solide, nous nous sommes lancés dans la phase d'implémentation qui a fait ressortir certains inconvénients de notre architecture et certains aspects ont alors été révisés. Nous avons d'abord développé un simulateur de base avec une stratégie de type *Best Response* et la règle de vote Bloc, puis nous y avons intégré l'interface, le détecteur de cycles et nous y avons ajouté d'autres règles. Enfin, la phase de tests nous a permis de détecter les bugs majeurs de notre application et de les corriger.

Les fonctionnalités évoquées dans le cahier des charges que nous avons prises en compte sont reprises dans les trois tableaux ci-dessous (Tableaux 3.1., 3.2. et 3.3), chacune reprenant les fonctionnalité regroupées par priorité dans le cahier des charges (obligatoire, optionnelle et imaginée)

Nous sommes restés fidèles au cahier des charges et avons pu implémenter toutes les fonctionnalités obligatoires. Une bonne partie des fonctionnalités restantes ont été prises en compte lors de la conception de l'application ce qui facilitera grandement l'ajout de ces nouveaux modules.

| Fonctionnalité | Implémentation |
|---|---|
| Ecriture et chargement d'un fichier de configuration de simulation | réalisée |
| Stratégie basée sur la distance de Hamming et agents omniscients et bloc | réalisée |
| Représentation graphique de l'avancement de la simulation | réalisée |
| Export des résultats | réalisée |
| Reconnaissance des cycles et arrêt automatique | réalisée |
| Simulation pas à pas, définition de la taille du pas de temps et retour en arrière | réalisée |
| Génération aléatoire des préférences des agents | réalisée |

Tableau 3.1 − Récapitulatif des fonctionnalités obligatoires du cahier des charges

| Fonctionnalité | Implémentation |
|---|---|
| Implémentation de multiples stratégies, niveaux de connaissances, règles de votes et types de préférences | des niveaux de connaissances, des règles de vote et un autre type de préférence ont bien été ajoutés |
| Représentations graphiques alternatives | deux modes graphiques ont été implémentés : les scores des candidats dans le cas des votes Bloc et K-Borda et ceux des comités dans le cas Chamberlin Courant. Notons que l'actuel mode graphique supporte tout type conforme à l'interface "IElectable" |
| Recharger une simulation à un état donné | non implémentée mais prévue dans l'architecture car la sauvegarde d'un état de la simulation (notamment les votes |

| | |
|---|---|
| | des agents) est déjà faite |

Tableau 3.2 − Récapitulatif des fonctionnalités optionnelles du cahier des charges

| Fonctionnalité | Implémentation |
|---|---|
| Changement dynamique de stratégies sous conditions | non implémentée mais prévue également dans l'architecture de par l'organisation des niveaux de connaissance, stratégies et règle de vote en composants interagissant les uns avec les autres |
| Statistiques d'évolution de la simulation | non implémentée, mais prévue dans l'architecture afin de faciliter l'implémentation |
| Taille de comité automatique | non implémentée, hors du cadre du projet |
| Amélioration de la génération des préférences des agents | non implémentée, mais prévue dans l'architecture, une simple implémentation de l'interface "IPreferencesGenerator" permettant de définir un nouveau générateur de préférences |

Tableau 3.3 − Récapitulatif des fonctionnalités imaginées du cahier des charges

Nous regrettons néanmoins de n'avoir pas pu réalisé un certain nombre de tâches faute de temps, notamment de ne pas avoir implémenté d'autres stratégies, particulièrement des stratégies d'apprentissage qui auraient pu tirer partie de la dynamique du processus pour manipuler de manière plus intelligente les élections. Il y a encore pas mal à faire sur la manière de gérer les préférences, entre autres travailler sur la complétion des préférences (que nous avons prévue dans l'architecture mais qui n'est pas entièrement fonctionnelle), et la génération des préférences : il faudrait revoir les préférences de Hamming pour ne pas générer *k* candidats (*k* étant la taille du comité) mais un nombre quelconque que l'utilisateur peut entrer, et améliorer la génération des préférences pour permettre de définir des populations plus élaborées et faire tests plus approfondis. Par ailleurs, nous n'avons pas prévu de module spécifique pour la résolution des tie-break alors que ça aurait pu permettre l'élaboration de nouvelles règles intéressantes.

Néanmoins, nous sommes fiers du travail réalisé compte tenu du temps qui a été mis à notre disposition. Nous tenons à remercier nos encadrants, M. Nicolas Maudet et M. Paolo Viappiani pour avoir proposé un sujet aussi intéressant et assez nouveau dans

le domaine de la recherche ainsi que pour leur disponibilité tout au long du projet. Nous avons pris plaisir à travailler dessus et nous espérons que ce simulateur leur servira au cours de leurs recherches futures.

# 4. Bibliographie

[1] F. Brandt, V. Conitzer, U. Endriss, J. Lang, et A. D. Procaccia, *Handbook of Computational Social Choice*, Cambridge University Press. New York, 2016.

[2] R. Meir, « Iterative Voting », in *Trends in Computational Social Choice*, AI Access., U. Endriss, 2017, p. 69‑86.

[3] A. Loreggia, « Iterative Voting, Control and Sentiment Analysis », thèse de doctorat, Université de Padoue, Padoue, 2016.

[4] R. Meir, M. Polukarov, J. S. Rosenschein, et N. R. Jennings, « Convergence to Equilibria in Plurality Voting », présenté à Proceedings of the Twenty-fourth conference on Artificial Intelligence (AAAI-2010), 2010.

[5] O. Lev et J. S. Rosenschein, « Convergence of Iterative Voting », présenté à Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS-2012), 2012, p. 611‑618.

[6] P. Faliszewski, P. Skowron, A. Slinko, et N. Talmon, « Multiwinner Voting: A New Challenge for Social Choice Theory », in *Trends in Computational Social Choice*, AI Access., New York: U. Endriss, 2017, p. 27‑48.

[7] S. Airiau, U. Grandi, et F. S. Perotto, « Learning Agents for Iterative Voting », in *Algorithmic Decision Theory*, vol. 10576, J. Rothe, Éd. Cham: Springer International Publishing, 2017, p. 139‑152.

[8] R. D. Luce, *Individual choice behavior: A theoretical analysis*, Wiley. New York, 1959.

[9] R. L. Plackett, « The analysis of permutations », *Applied Stat.*, vol. 24, n° 2, p. 193‑202, 1975.

[10] R. S. Boyer et J. S. Moore, « A fast string searching algorithm », *CACM*, vol. 20, n° 10, p. 762‑772, 1977.

[11] J. Eagan, « Introduction à la Conception Centrée Utilisateur », 2016.

# Annexes

# A. Cahier des charges

## A.1. Introduction

### A.1.1. Contexte

Ce projet s'inscrit dans le cadre de l'UE P-ANDROIDE de 1ère année du master informatique parcours ANDROIDE de la faculté d'ingénierie de Sorbonne Université (du 15/01/2018 au 7/01/2018). Il est réalisé par une équipe composée de deux encadreurs enseignants-chercheurs au laboratoire d'informatique de Paris 6 (Lip6) ainsi que de trois étudiants de l'université.

Ce document représente le cahier des charges du projet "Vote itéré de comité" et a pour but d'expliciter et de regrouper la demande des clients, M. Nicolas Maudet et M. Paolo Vappiani, et l'offre du prestataire, le groupe du projet dont les membres sont Loic URIEN, Victor RACINE et Yasmine HAMDANI. Il servira de référence lors du développement du projet et permettra aux deux parties d'être en accord sur les différents points conceptuels avant de se lancer dans le développement de l'application.

Les clients, M. Maudet et M. Vappiani, proposent de créer une application permettant de simuler un vote itéré de comité, en laissant libre recours au groupe sur le choix de certaines fonctionnalités tout en définissant les fonctionnalités obligatoires, sans lesquelles le projet ne pourra être considéré comme complété. Ce document est donc un moyen de synthétiser la demande des clients, de définir les contraintes liées au développement ainsi qu'à l'utilisation du produit fini et d'établir le planning prévu pour réaliser ce projet.

### A.1.2. Description

Le programme, relevant du domaine de la prise de décisions collective, devra permettre la simulation d'un vote itéré de comité. Ceci nécessitera la mise en œuvre de notions de décision dans un cadre multi-agents dynamique.

L'intérêt de cette application est de permettre l'étude de l'évolution d'un vote itéré effectué par un certain nombre d'agents (électeurs) ayant à priori des préférences électorales différentes, l'objectif étant d'arriver à un consensus entre les agents afin d'élire les membres d'un comité. Dans la suite du document, nous référerons à chaque itération du vote comme étant un “pas de temps”.

Le programme devra offrir une interface simple permettant d'ajuster les paramètres de la simulation (notamment les préférences des votants, leur niveau d'information, et le nombre maximum d'itérations) et afficher le résultat du vote.

L'application se voudra modulaire afin de faciliter l'ajout/suppression et la modification de ses composants, en particulier les stratégies, les degrés de connaissances associés et les représentations graphiques de la simulation.

## A.2. Description de la demande

### A.2.1) Demande textuelle

L’étude efficace des différents comportements d’un groupe votant pour l’élection d’un comité représente un intérêt certain dans notre société actuelle, aussi bien dans le domaine de la politique que pour l’organisation et la planification. Dans cette optique, nous nous proposons de mettre en place un simulateur de ce type de comportements, capable de modéliser diverses stratégies et préférences.

Le programme ainsi créé devra permettre à l’utilisateur de configurer sa simulation via une interface simplifiée, ainsi que de la sauvegarder afin de permettre une réutilisation ou modification de celle-ci à une date ultérieure. L’application devra également permettre une visualisation simple des résultats de la simulation, plusieurs options d’avancement de la simulation (pause, pas à pas, avance rapide, etc …) ainsi que la possibilité d’exporter un rapport détaillé sous forme de document CSV à chaque pas de temps afin de pouvoir étudier les résultats.

## A.2.2) Fonctionnalités obligatoires

### 2.2.a) Ecriture et chargement d'un fichier de configuration de simulation

Afin de faciliter l'utilisation du simulateur, il est de la plus haute importance de proposer une manière simple de définir la simulation ainsi que de pouvoir y revenir plus tard. Dans ce cadre, nous nous proposons de donner à l'utilisateur la possibilité de sauvegarder sa simulation dans un fichier JSON, afin de pouvoir la charger plus tard à nouveau dans le simulateur.

Cette configuration se devra de sauvegarder les agents, leurs préférences, les candidats ainsi que le type de préférence utilisée et la stratégie mise en application. Elle ne gardera en aucun cas quelconque mémoire de la simulation à un temps donné.

### 2.2.b) Stratégie basée sur la distance de Hamming et agents omniscients et bloc

L'application devra implémenter au minimum une stratégie basée sur la distance de Hamming en utilisant des agents ayant une connaissance totale de leur environnement, à savoir, une connaissance du nombre de votes reçus par chaque candidat après chaque pas de temps. Le comité sera élu par la règle de vote bloc, à savoir les k candidats ayant reçu le plus de votes. S'il y a égalité, nous opterons pour un choix par ordre lexicographique.

### 2.2.c) Représentation graphique de l'avancement de la simulation

Une représentation graphique dynamique de la simulation devra être mise en place afin de permettre à l'utilisateur de visualiser la progression ou l'absence de progression, lui permettant d'arrêter manuellement la simulation si besoin.
Par défaut, la représentation sera un simple graphe ayant sur l'axe des abscisses les différents candidats et en ordonnées leurs scores. Pour plus d'informations, voir la section 3.1.1.

### 2.2.d) Export des résultats

L'utilisateur devra pouvoir être en mesure d'exporter le résultat de la simulation à chaque pas de temps sous forme d'un fichier CSV contenant les votants avec leurs préférences actuelles ainsi que le résultat des votes. Il n'est cependant pas prévu de pouvoir charger ce fichier dans l'application afin de reprendre à un état donné de la simulation (voir : Fonctionnalités optionnelles)

### 2.2.e) Reconnaissance des cycles et arrêt automatique

Nous devrons être en mesure de détecter les cycles et effectuer les actions nécessaires à la non stagnation de la simulation. Dans les fonctionnalités obligatoires, nous arrêterons la simulation immédiatement après avoir détecté un cycle ou une stagnation de durée N, indiquant l'arrivée probable à un état de stabilité, avec N un nombre de pas de temps configurable. L'application devra donc garder en mémoire un certain nombre d'états précédents de la simulation afin de pouvoir détecter les cycles efficacement.

### 2.2.f) Simulation pas à pas, définition de la taille du pas de temps et retour en arrière

Il est nécessaires de pouvoir mettre en pause la simulation à n'importe quel instant, la redémarrer ou l'avancer pas à pas. Il est également nécessaire de pouvoir régler la vitesse de la simulation, l'utilisateur devra donc être en mesure de définir la taille d'un pas de temps manuellement. L'application devra de plus garder en mémoire un certain nombre d'états précédents afin de permettre le retour en arrière de la simulation. De plus, à chaque étape, le comité élu devra être annoncé à l'utilisateur.

### 2.2.g) Génération aléatoire des préférences des agents

L'application devra pouvoir générer de manière aléatoire les préférences des agents étant donné une liste de candidats. Dans un premier temps, nous nous concentrerons uniquement sur une génération purement aléatoire et uniforme.

## A.2.3) Fonctionnalités optionnelles

### 2.3.a) Implémentation de multiples stratégies, niveaux de connaissances, règles de votes et types de préférences

Il serait extrêmement intéressant pour la simulation d'avoir la possibilité de changer la stratégie utilisée par les agents, afin de ne pas se limiter uniquement à la distance de Hamming. Il serait aussi préférable d'implémenter d'autres niveaux de connaissances que celui d'omniscience (i.e : les agents connaissent à tout moment le résultat des votes). Nous pouvons noter les alternatives suivantes :

- Préférence responsive
- Niveau de connaissance limité au classement
- Règle de vote Chamberlin Courant, Condorcet...

Toutes les stratégies ne peuvent être utilisées avec un niveau de connaissance quelconque, il est donc évident que le niveau de connaissance dépendra grandement de la stratégie utilisée. Il est cependant possible d'avoir plusieurs niveaux de connaissances pour une stratégie donnée.

### 2.3.b) Représentations graphiques alternatives

Nous pouvons envisager d'adopter d'autres modes de représentations graphiques que celui par défaut, afin de donner d'autres visions de la simulation à l'utilisateur. Nous pouvons également envisager un layout interactif permettant de visualiser plusieurs paramètres en même temps, donnant ainsi plus de précisions sur l'évolution du modèle.

### 2.3.c) Recharger une simulation à un état donné

Bien que non prévu dans les fonctionnalités de base de l'application, il serait intéressant de penser à un moyen de recharger une simulation à un instant donné. L'application pourra ainsi recharger un fichier de résultat CSV ainsi que la configuration correspondante (identification unique par code de hachage) afin de reprendre la simulation là où elle s'était arrêtée. Il est également possible de repenser le modèle d'export des résultats afin que la configuration soit inclue, évitant ainsi de ne plus pouvoir réutiliser le fichier de résultats si le fichier de configuration n'est plus présent.

## A.2.4) Fonctionnalités imaginées

### 2.4.a) Changement dynamique de stratégies sous conditions

Il serait intéressant de pouvoir effectuer un ou plusieurs changements de stratégie si une ou plusieurs conditions sont remplies. On peut imaginer par exemple qu'à partir d'un certain nombre d'itérations, ou après avoir détecté un cycle ou un état de stabilité avec un trop grand désaccord entre les agents, l'application change de stratégie avec d'essayer de trouver une meilleure solution. Les conditions pourraient être entrées par l'utilisateur ou bien simplement prédéfinies par l'application.

### 2.4.b) Statistiques d'évolution de la simulation

L'application pourrait donner des statistiques sur la vitesse de convergence et la distribution des votes.

### 2.4.c) Taille de comité automatique

Nous voulons être en mesure de pouvoir déterminer automatiquement une bonne taille de comité sans que l'utilisateur n'ait besoin de la définir, en utilisant par exemple une règle de type min-max. Précisons cependant que l'utilisateur devra être en mesure de remplacer cette valeur automatique par une valeur de son choix afin de ne pas limiter les possibilités de simulation.

### 2.4.d) Amélioration de la génération des préférences des agents

Il serait intéressant d'utiliser une génération plus "intelligente" et paramétrable pour la génération des préférences des agents, par exemple en utilisant la méthode de Plackett-Luce.

## A.2.5) Expression fonctionnelle des besoins

### 2.5.a) Exécution de la simulation

L'application devra pouvoir effectuer la simulation jusqu'à rencontrer une condition d'arrêt ou que l'utilisateur décide de lui-même d'y mettre fin. L'interface devra permettre de

facilement contrôler la taille du pas de temps ainsi que l'avancée de la simulation. L'utilisateur devra pouvoir :

- Mettre la simulation en pause
- Relancer la simulation en course
- Remettre la simulation à l'état initial
- Effectuer une avancée pas à pas (uniquement disponible en mode pause)
- Arrêter la simulation
- Revenir en arrière

A chaque pas de temps, sauf au pas de temps initial, l'utilisateur aura la possibilité d'enregistrer les résultats obtenus dans un fichier CSV. (voir 2.5.b)

L'échec lors du chargement d'une configuration à partir d'un fichier devra résulter en une simulation vide et un message d'erreur explicite devra être fourni en log. L'application ne devra en aucun cas produire une simulation erronée à partir d'un fichier de configuration invalide, toute donnée lue devra être ignorée.

Une erreur lors de la simulation devra être traitée comme fatale et les résultats obtenus invalides. L'utilisateur pourra choisir de sauvegarder les résultats obtenus à l'itération actuelle ou durant une itération précédente, mais le programme n'offrira alors aucune garantie de validité sur ceux-ci. De même, lors d'un échec il ne sera plus possible de relancer la simulation, de revenir en arrière ou de faire du pas à pas.

De plus, si la simulation ne peut pas effectuer une étape dans le pas de temps défini par l'utilisateur, l'application devra fournir un message clair concernant cette situation.

### 2.5.b) Sauvegarde des résultats

A chaque pas de temps, les résultats de la simulation (comité élu, score des candidats) devront pouvoir être sauvegardés dans un fichier au format CSV, pouvant être aisément chargé dans un logiciel de type tableur. Comme indiqué dans la section précédente, si l'application lance une erreur, les résultats ne pourront plus être considérés comme valides. L'utilisateur aura trois moyens de sauvegarder l'itération actuelle, soit par le raccourci ctrl+s, soit le bouton de sauvegarde dans la barre des contrôles (voir section 3.2) soit finalement par le menu "file" en cliquant sur "save results as" (voir section 3.3).

Par défaut, le nom du fichier sera de la forme $nomDuProfil$_$nombreIterationActuel$, où "nomDuProfil" et "nombreIterationActuel" sont des variables de la simulation.

L'utilisateur ne devra pas avoir la possibilité de sauvegarder les résultats de la simulation à l'instant initial, celle-ci n'ayant pas commencé et les résultats étant donc vides.

L'application devra lui signaler cette impossibilité par un message expliquant cet état de faits.

Si une erreur se produit lors de la sauvegarde des résultats, alors aucun fichier ne devra être créé et l'utilisateur devra être informé de l'erreur. Si l'application a rencontré une erreur provenant d'une autre partie de l'application que celle de l'écriture du fichier, alors il faudra tout de même sauvegarder les résultats, en informant l'utilisateur que ceux-ci ont de grandes chances d'êtres invalides.

### 2.5.c) Création et enregistrement d'un profil de simulation

L'application devra permettre à l'utilisateur de facilement créer, éditer, sauvegarder et recharger un profil de simulation, afin de permettre une plus grande aisance durant le paramétrage de la simulation. Pour créer et éditer un profil, il aura accès à un outil dédié intégré au programme via une interface graphique (voir section 3.2). Si jamais la configuration de l'utilisateur se trouve être erronée, elle ne devra en aucun cas pouvoir être sauvegardée.

L'utilisateur aura également comme possibilité de charger un profil de configuration déjà existant, afin de l'éditer ou de relancer une simulation antérieure. Si le fichier donné en entrée n'est pas un fichier de configuration de profil valide, l'application ne devra en aucun cas entrer dans un stade d'erreur total et ne devra changer aucune valeur que l'utilisateur aurait pu entrer précédemment, agissant comme si aucune action n'avait été effectuée.

# A.3. Maquettes graphiques

## A.3.1) Fenêtre principale

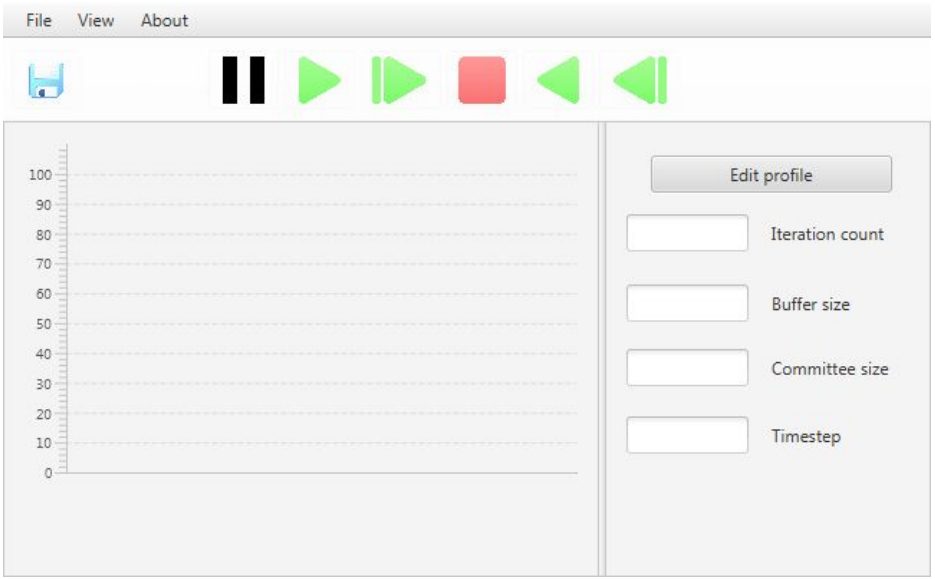



Figure A.3.1 - Fenêtre principale de l'application

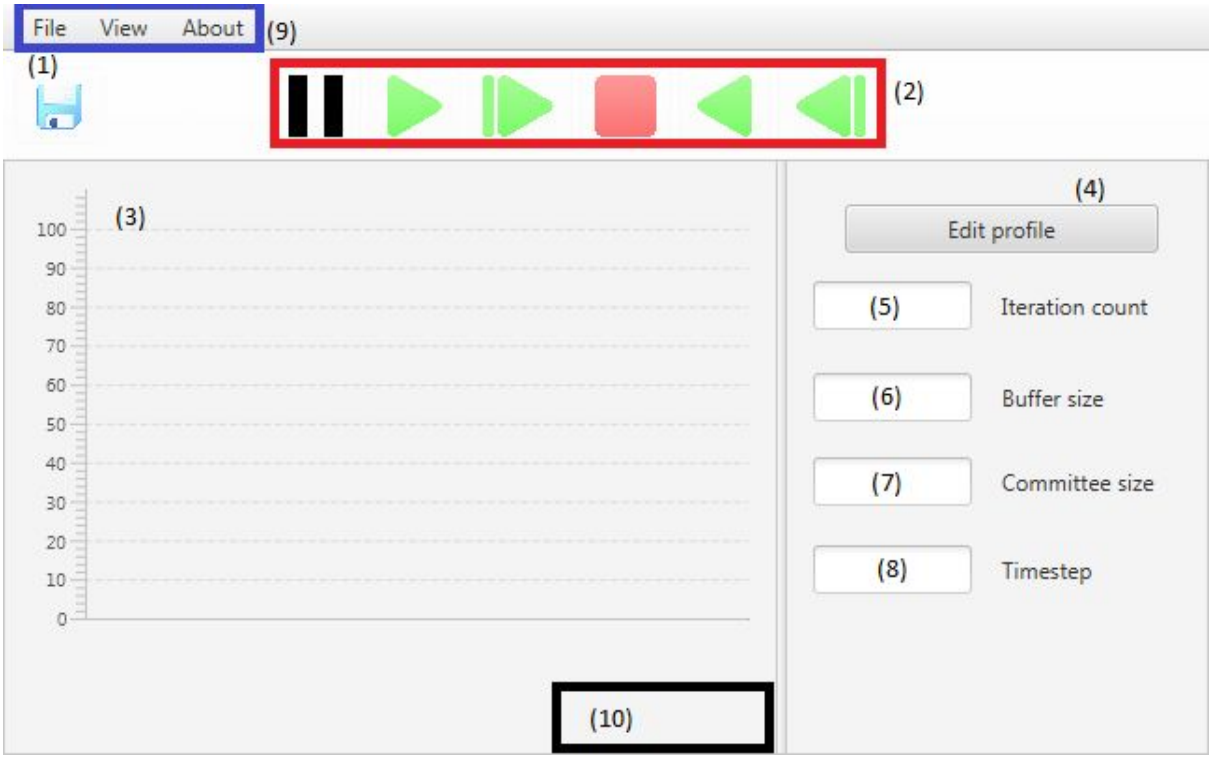



Figure A.3.2 - Fenêtre principale de l'application annotée

Nous décrivons ici les différentes parties composant l'interface principale de l'application :

(1) : Sauvegarde la trame actuelle de la simulation au format CSV. Pour plus d'information, se référer à la section 2.5.2

(2) : Panneau de contrôle de l'avancement de la simulation. De gauche à droite :

- (a) Met la simulation en pause. Cette option ne doit pas être disponible (bouton désactivé) si jamais l'application est arrêtée
- (b) Lance la simulation, ou la relance si celle-ci est en pause
- (c) Effectue une avancée pas à pas de la simulation
- (d) Arrête la simulation
- (e) Fait revenir la simulation sur ses pas
- (f) Retour en arrière pas à pas

(3) : Affichage des résultats de manière graphique

(4) : Edition du profil. Ouvre la fenêtre d'édition du profil de la simulation (voir section suivante)

(5) : Nombre d'itérations avant que la simulation ne s'arrête, infini si laissé vide

(6) : Taille de la mémoire tampon de l'application, définissant le nombre de "retours en arrière" pouvant être effectués. Le choix de la valeur par défaut sera laissée à la discrétion des développeurs

(7) : Taille du comité désirée, comité unitaire par défaut

(8) : Taille du pas de temps. Si ce champ est laissé vide, l'application essayera de compléter chaque itération le plus rapidement possible, le pas de temps ne sera donc pas constant

(9) : Barre de menus. Pour plus d'information, voir la section 3.3

(10) : Indication du comité élu actuellement, vide à l'étape initiale

### 3.1.1) Représentation graphique par défaut

Dans cette section nous démontrons la représentation par défaut de l'évolution de la simulation, représenté dans l'item (3) décrit dans la section précédente. Partons de principe qu'à une itération donnée nous ayons les résultats suivants pour l'élection :

| | |
|---|---|
| A | 54 |
| B | 23 |
| C | 76 |
| D | 23 |
| E | 4 |
| F | 19 |

Le graphe donné serait donc :

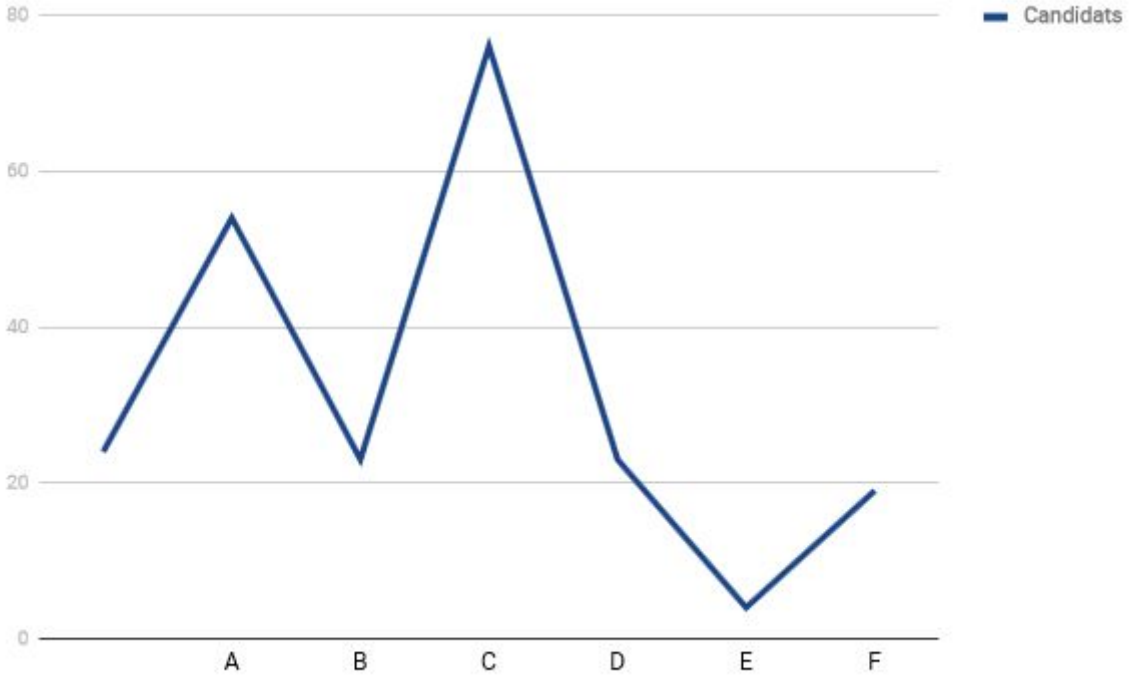



Figure A.3.3 - Graphe des scores des candidats

## A.3.2) Fenêtre de configuration du profil

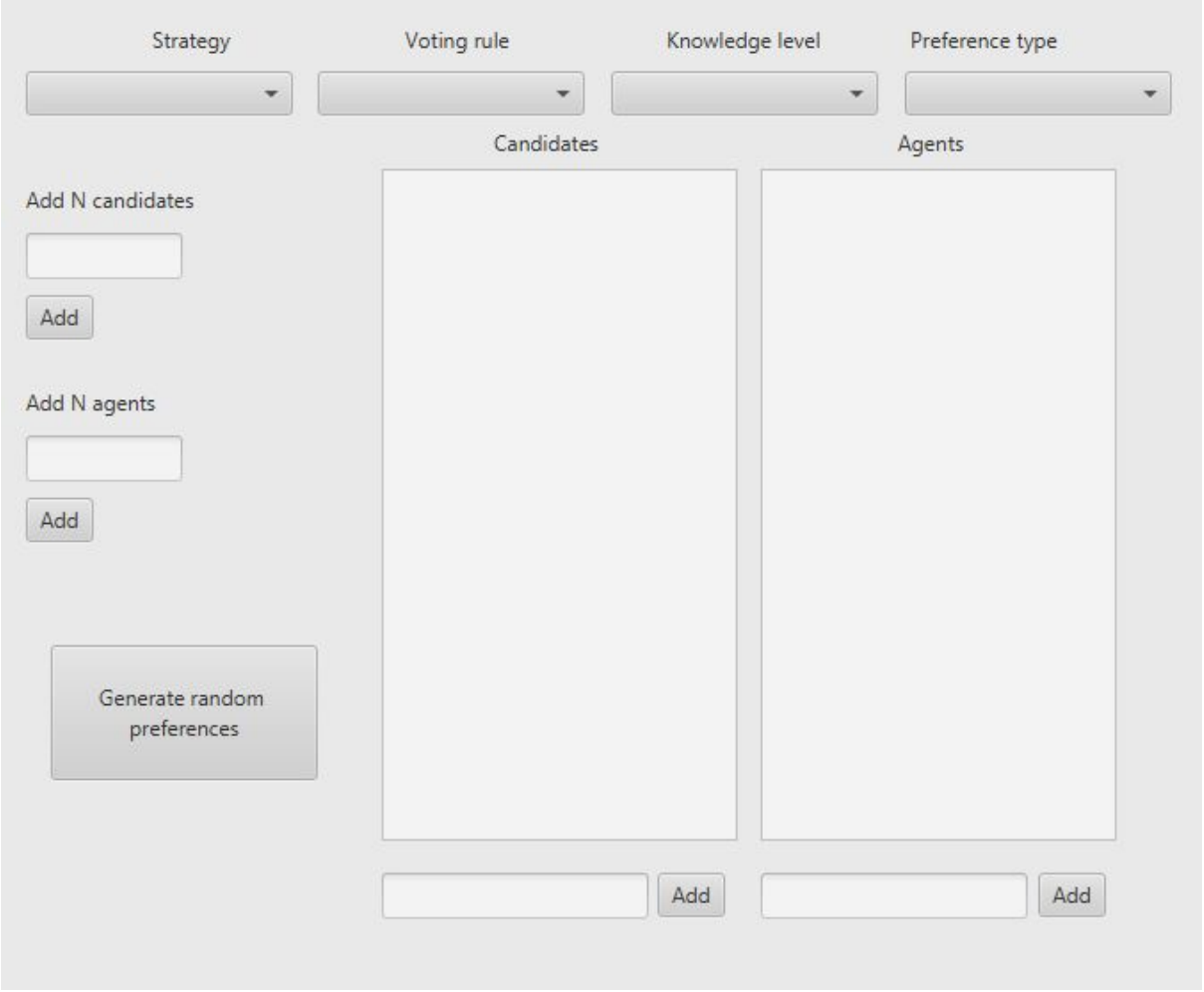



Figure A.3.4 - Fenêtre de configuration du profil

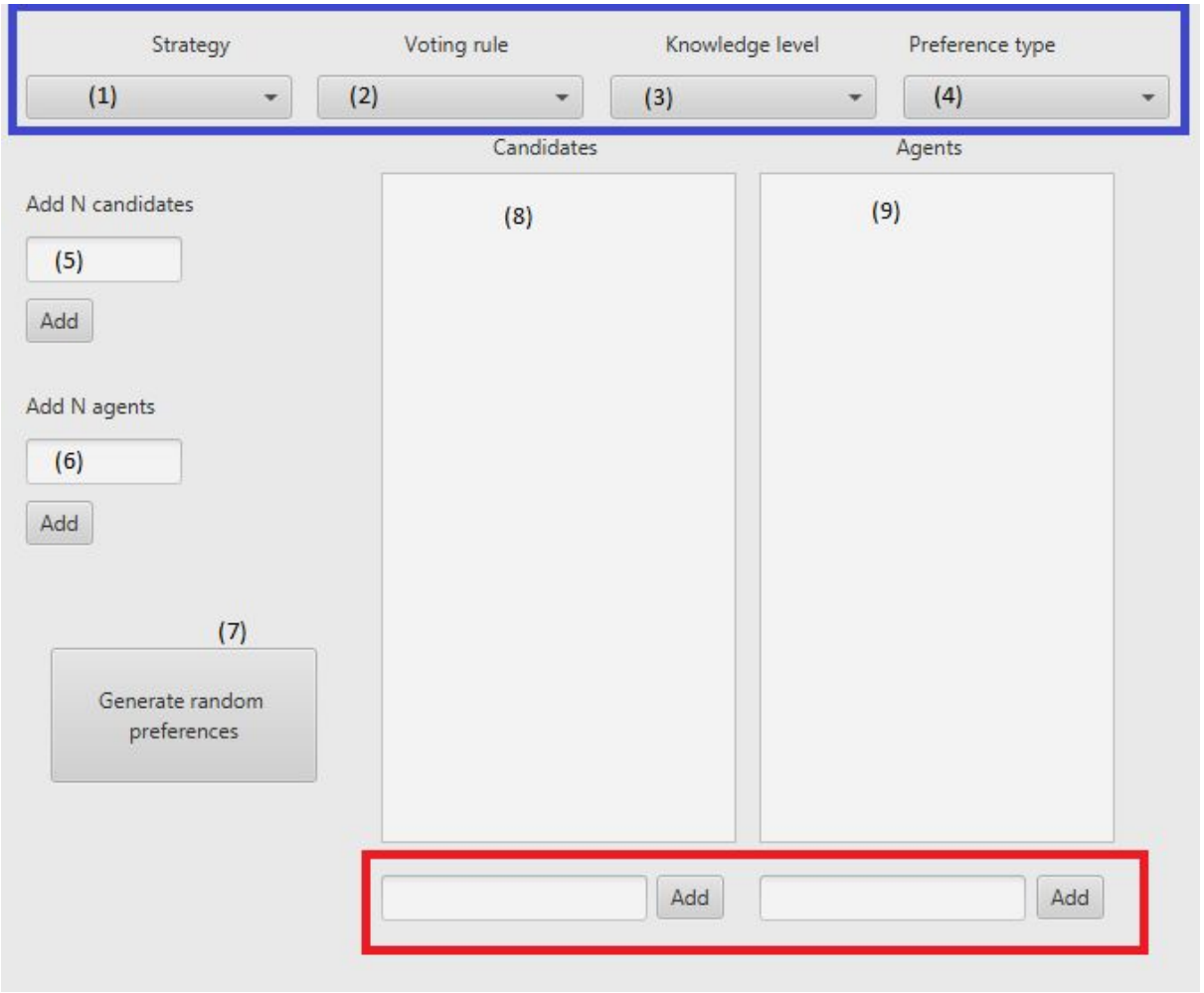



Figure A.3.5 - Fenêtre de configuration du profil marquée

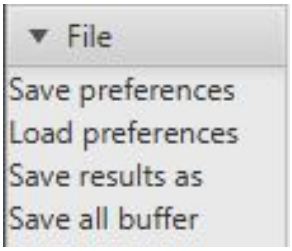
Cette fenêtre est ouverte via la fenêtre principale de l'application (voir section 3.1) en cliquant sur le bouton "Edit Profile". Elle permet à l'utilisateur de facilement configurer la simulation sans avoir à éditer le fichier de configuration. Ci-dessous nous décrivons les différentes parties de cette fenêtre. Très peu d'informations seront données pour les options dans la partie encadrée en bleu. Par défaut, seulement quelques options seront disponibles dans les menus déroulants. Pour plus d'informations, se référer aux sections contenues dans la description de la demande, particulièrement les sections 2.2.b et 2.3.a

(1) : Choix de la stratégie
(2) : Choix de la règle de vote
(3) : Choix du niveau de connaissance
(4) : Choix du type de préférences. Attention, changer cette option peut potentiellement invalider une configuration déjà existante si le nouveau format de préférences est différent de celui utilisé précédemment
(5) : Ajoute un nombre N de candidats en leur attribuant des noms génériques
(6) : Ajoute un nombre N d'agents en leur attribuant des noms génériques
(7) : Génère des préférences aléatoires pour les agents en se basant sur le format de préférences choisi

(8) : La liste des candidats. Il est possible de cliquer sur chacun des items afin d'éditer leurs noms. Chaque candidat n'apparaîtra que par son nom
(9) : La liste des agents. Il est possible de cliquer sur chacun des items afin d'éditer leurs noms et leurs préférences. Chaque agent apparaîtra par son nom et une brève description de ses préférences
(10) : Cette section permet l'ajout candidat par candidat et agent par agent

## A.3.3) Barre des menus

Dans cette section nous nous intéresserons à la barre des menus introduite dans la section 3.1.




Dans l'ordre de haut en bas :

- Sauvegarde les préférences de la simulation actuelle
- Charge des préférences sauvegardées précédemment, écrasant celles déjà définies dans l'application en même temps
- Sauvegarde les résultats de la trame actuelle dans un fichier au format .csv
- Sauvegarde l'entièreté du tampon de l'application