UNIVERSITÉ
CAEN
NORMANDIE

CONCEPTION LOGICIEL

Optimisateur de WarGame

21810781 - KABORI Hamza
21802795 - LUCCHINI Melvin
21804030 - LETELLIER Guillaume
21808205 - MOK William
21810530 - Pais Oliveira Lorenzo

L1 Informatique
PROMO 2018-2019

14 avril 2019

Table des matières

1 Objectifs du projet

1.1 Description du concept derrière l'application

Le but de l'application que nous avions à réaliser est de créer, si l'on reprend les modalités du sujet, un Optimisateur de WarGame. Tout d'abord, qu'est ce qu'un WarGame ? C'est tout simplement un jeu de stratégie où deux armées se rencontrent et s'affrontent. Une armée est composée de plusieurs unités qui peuvent changer du tout au tout selon l'époque dans laquelle se joue la partie. Pour notre part, nous avons choisi de mettre en place le WarGame durant l'époque médiévale, c'est-à-dire avec des archers, des cavaliers, des chevaliers, etc...

Le WarGame se joue généralement en multijoueur, mais dans le cadre de notre projet, on fera s'affronter un joueur réel contre une pseudo intelligence artificielle. Il peut être extrêmement complexe avec beaucoup de règles qu'elles soient au niveau du déplacement ou encore de l'attaque.

Bien évidemment, il nous fallait d'abord créer un WarGame avec des méthodes pour attaquer, pour se déplacer, pour générer deux armées, gérer leurs positions, etc... Mais le réel concept derrière l'application était l'optimisation. Lorsqu'on parle d'optimiser un WarGame, on ne parle pas d'optimiser le code pour qu'il s'exécute plus vite, mais plutôt d'optimiser l'armée de l'utilisateur en fonction de celle de l'ennemie pour la rendre optimale face à cette dernière. C'est cette optimisation qui est le réel but de l'application, même si l'on devait d'abord créer un WarGame fonctionnel.

1.2 Ce qu'il fallait faire

Comme dit précédemment, il fallait d'abord créer un WarGame fonctionnel pour pouvoir ensuite implémenter l'optimisation. Nous avons d'abord commencer par fixé l'époque et tout le contexte derrière. Nous avons donc choisi l'époque médiévale. Il nous fallait ensuite créer l'armée et toutes les unités qui la compose. Une armée est donc composée :

- d'un roi et un seul (King) qui est l'unité la plus forte ;
- de cavaliers (Horseman) ;
- de chevaliers (Knight) ;
- de archers (Bowman) ;
- et de guerriers (Warrior).

Il faut ensuite définir pour chaque unité, un certains nombre de paramètres qui permettront de les hiérarchiser, c'est-à-dire que selon leurs paramètres, une unités prendra le dessus sur une autre. Par exemple, on définit un nombre de points de vie différent pour chaque unité. On définit aussi :

- un nombre de points d'armure ;
- les dégâts infligé par l'unité en cas d'attaque ;
- une distance d'attaque ;
- une distance maximale de déplacement ;
- un nombre représentant le moral d'une troupe ;
- le coût en points d'armée de l'unité ;
- ...

Parallèlement à cela, on peut implémenter des malus et des bonus qui se répercuteront sur les paramètres cités juste avant. Par exemple, lorsqu'une unité, qu'elle soit allié ou ennemie, se retrouve entourée par des unités ennemies, on affecte un malus a cette unité. Ce malus est

définit par une baisse de moral. Autre exemple, lorsque le Roi meurt, on applique un malus au reste des unités faisant partie de l'équipe du Roi déchu, ce qui baissera les dégâts etc...

Il fallait aussi créer de nouvelles méthodes pour pouvoir déplacer une unité quelconque, qu'elle puisse attaquer, tout cela en fonction des paramètres de chaque unité.

Pour nous assurer que les méthodes de déplacement et d'attaque soient fonctionnelles, il faut pouvoir visualiser notre WarGame. Pour ce faire, nous avons décidé de "créer" un simulateur. Dans ce simulateur, on peut créer l'armée que l'on souhaite en fonction du nombre de points d'armée que l'on possède, puis nous pouvons choisir la position de chaque unité avec des coordonnées en x et en y (en px). Ensuite, dans le simulateur, on peut tester toutes les choses énoncées précédemment, c'est-à-dire, les différents malus/bonus, et les méthodes d'attaque et de déplacement. Pour ce qui est des méthodes d'attaque et de déplacement, le simulateur simule le combat entre deux armées (ordinateur VS joueur) et nous indique qui gagne à la fin du combat.

Parallèlement à tout ce travail, il faut aussi créer une interface graphique qui nous permettra de mieux visualiser notre WarGame en créant une carte et en représentant chaque unité avec un symbole unique. Cette interface sera réaliser avec la librairie PyGame.

Une fois que le WarGame est fonctionnel. On peut implémenter l'optimisation. Comme nous l'avions dit, l'optimisation servira a créer l'armée optimale pour le joueur en fonction de l'armée ennemie. Cette optimisation va aussi positionner chaque unité de façon optimale. Pour mettre en place cela, il nous a fallut apprendre ce qu'était un algorithme génétique puis le réadapter à notre jeu. Cette algorithme fera une série de test, jusqu'à avoir la meilleure armée. Chaque unité sera positionner de façon optimale. Grâce à cet algorithme, l'ordinateur n'a quasiment aucune chance de gagner face au joueur.

1.3 Ce qui existe déjà

Bien évidemment, il existe déjà beaucoup beaucoup de WarGame et donc il existe déjà des règles prédéfinies pour un WarGame. Pour nous aider, nous avons bien sûr regarder comment fonctionner les autres wargames. Ce qui a pu nous donner des idées pour faire le notre comme l'idée des bonus et des malus. Mais nous voulions tout de même créer notre WarGame qui reprend les règles de base mais tout en adaptant un peu celles-ci à notre guise et en fonction des idées que nous avions en tête.

Ce qui nous a permis de faire l'optimisation de l'armée alliée est ce qu'on appelle un algorithme génétique. Ce type d'algorithme reprend en quelque sorte les règles de la génétique et de la biologie. Voici les différents termes liés à cet algorithme et qui la plupart du temps, sont utilisés pour tout type de problème, à condition bien sûr de les adapter.

- Un individu : c'est rien de plus que le résultat attendu. Dans notre problème, le résultat attendu est un groupe d'individus. (Armée optimale)
- Une population : c'est une collection ou regroupement d'individus.
- Une génération : c'est une étape dans l'évolution d'une population.
- Une sélection : c'est une étape clef dans le fonctionne des algorithmes génétiques (GA). Cette étape attribue un score à chaque individu en fonction de son adaptation au problème donnée. Ce score est nommé fitness (= aptitude pour...) et en fonction des scores

obtenus par chaque individu de la population, certains seront conservés dans la génération suivante. Les autres seront oubliés. C'est le principe biologique de la sélection naturelle.
- Un croisement : c'est une étape qui consiste à prendre deux individus différents et de les faire se reproduire pour générer deux nouveaux individus.
- Une mutation : La mutation est une étape qui consiste à introduire aléatoirement des modifications dans les constituants d'un individu.

En adaptant tout cela à notre problème, on réussit à obtenir en moins de 30 secondes avec 25 générations sur une population de 10 individus, une armée optimale positionnée et prête à attaquer l'ennemi.

2 Fonctionnalités implémentées

2.1 Description des fonctionnalités

En ouvrant notre jeu **WarGame**, nous pouvons entendre sa musique d'ambiance dont nous pouvons couper le son en appuyant sur la touche "u" ou bien avec un simple clic sur l'image son située en haut à droite de la fenêtre du jeu. Également, nous arrivons sur le menu qui propose plusieurs fonctionnalités qui sont :

- **QUIT** : Quitter le jeu.
- **ABOUT** : Toutes les informations nous concernant y sont référés (Nos noms, la version de PyGame ainsi que le but de la réalisation du jeu).
- **PLAY** : Le joueur souhaite s'aventurer, combattre son adversaire (qui est un IA) et va choisir entre ces deux options sachant qu'il connait l'armée ennemie générée aléatoirement :
  - **CREATE** : La création de son armée est prévue ici. Le coût maximal de l'équipe ainsi que son coup actuel sont affichés en dessous de la liste des unités. Ainsi le joueur choisit le nombre de chaque unité avec les touches directionnelles exceptées le nombre de Roi qui est d'office à 1 et qui ne peut être modifié.
  - **OPTIMIZE** : L'optimisateur directement implémenté dans le jeu va concevoir une armée la plus optimisée pour combattre celle de l'opposant. Il faudra alors attendre 15 à 20 secondes pour récupérer son armée. Cela est dû au temps d'exécution de l'algorithme génétique. (cf. 3.1.6).
  - **VALIDATE** : Une fois l'armée engendrée, le joueur accepte le combat et est redirigé vers la carte où ont lieu les combats entre l'ordinateur (le camp gauche rouge) et l'utilisateur (le camp droit bleu).

Toutefois, si le prétendant a des remords avant d'appuyer sur **VALIDATE**, alors il peut retourner en arrière en cliquant sur la flèche arrière.

La carte est constituée d'une multitude de cases :

- vertes claires (formant un carré) représente le champ de bataille, les plaines ; à l'intérieur se trouvent :
  - verte foncée : la forêt
  - bleue : les surfaces d'eaux (impossible à se positionner dessus)
  - jaune : le désert
  - grise : la montagne

Le jeu au final se termine lorsque l'une des deux armées est annihilée. Le joueur a la possibilité de sauvegarder la partie. S'il effectue ceci, alors lors de la prochaine réouverture du jeu et qu'il

appuyera sur le bouton **PLAY**, alors s'affichera une proposition disant si le joueur souhaite continuer la partie ou bien écraser la partie en commençant une autre nouvelle.

2.2 Organisation du projet

Nous avons tous, tout d'abord, mené une réflexion sur les bases fondamentales de notre jeu **WarGame** à savoir les règles de combat, la création des unités (de classes), leurs performances et les tactiques. Suite à cela, durant ces vingtaines de séances, nous nous sommes répartis le travail principalement en deux groupes. D'une part, Hamza et William se sont investis sur l'interface graphique pour permettre une bonne accessibilité du jeu et le rendre esthétique afin d'attirer l'œil des joueurs. D'autre part, Guillaume, Melvin et Lorenzo se sont attaqués sur la partie brute du projet, c'est-à-dire le codage de notre jeu **WarGame** afin qu'il y ait une fluidité optimale du système et une mise en main du jeu rapide et facile.

3 Éléments techniques

3.1 Algorithmes implémentées

3.1.1 Fichier 'units.py'

Dans la classe *Unit* de ce fichier, plusieurs algorithmes ont été utilisés, notamment pour la recherche dans les différentes méthodes de la classe ou encore la vérification.

Nous avons tout d'abord utilisé la formule de distance entre 2 points :

$$AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2} \tag{1}$$

Elle permet de vérifier si l'ennemi se trouve dans la portée d'attaque de l'unité (méthode "can_attack") ou si la case se trouve dans la portée d'attaque (méthode "attack_boxes_valid") ou dans la portée de mouvement (méthode "movement_boxes_valid").

Ensuite, un algorithme important a été utilisé. Il sert à déterminer le déplacement d'une unité d'une case à une autre. Elle se constitue de multiples conditions liées aux mouvements d'une part et d'autres part, l'assignation des nouvelles coordonnées à l'unité.

```
1: x est la coordonnée en abscisse de l'unité
2: if x ne se trouve pas dans la carte then
3:    if x est inférieur à la coordonnées en x de carte then
4:       x ← x+ vitesse de mouvement de l'unité
5:    else
6:       x ← x− vitesse de mouvement de l'unité
7:    end if
8: else
9:    x_a position en x de l'ennemi choisi
10:   if x_a < x and |x_a − x| >portée d'attaque de l'unité then
11:      x ← x− vitesse de mouvement de l'unité
12:   else if x_a > x and |x_a − x| >portée d'attaque de l'unité then
13:      x ← x+ vitesse de mouvement de l'unité
14:   end if
15: end if
```

lined 1 – Algorithme de mouvement des unités

Dans le code précédent, nous copions les valeurs des attributs de coordonnées de *Unit* et nous comparons les valeurs à celles de la position de la carte, si la valeur est trop grande, on la diminue et inversement. Si la valeur se trouve bien dans la carte, on fait bouger l'unité en fonction de la position de l'ennemi. L'algorithme ne montre que pour x mais il y a la même base pour y mais en ordonnées au lieu des abscisses.

3.1.2 Fichier 'mapping.py'

Dans la première partie du fichier correspondant aux définitions des différentes cases (rivières, forêts, ...), aucun algorithme n'a été utilisé. Dans la seconde partie, il y a deux algorithmes majeurs. L'un permet la création de la carte de combat et l'autre la recherche d'une ou plusieurs cases de la carte.

L'algorithme de création fait appelle à de nombreuses boucles. On définit tout d'abord le nombre de cases qui peuvent être placées horizontalement et verticalement. Puis nous créons les 2 villes (alliée et ennemie) :

```
1: allBox = []
2: boxWidth correspond au nombre de cases en largeur
3: boxHeight correspondant au nombre de cases en hauteur
4: boxSize correspondant à la longueur du côté de la case
5: mapPosWidth position de la carte horizontalement
6: mapPosHeight position de la carte verticalement
7: t représente le nombre de case 'Ville' en largeur
8: for i ← 0, t − 1 do
9:    for j ← 0, boxHeight − 1 do
10:      Ajouter ('townType', (i ∗ boxSize + mapPosWidth, j ∗ boxSize + mapPosHeight)) à
         allBox
11:   end for
12: end for
```

lined 2 – Algorithme de création (viles)

Ensuite, on met toutes les cases restantes dans une liste vide *possibleBoxes*. Pour chaque type et dans cet ordre (rivières, montagnes, forêts et déserts), on applique l'algorithme suivant :

```
1: Nb < type > le nombre de cases du 'type'
2: for i ← 0, Nb < type > −1 do
3:    posChoice ← randomBox {randomBox est une case choisie aléatoirement}
4:    Enlever posChoice de possibleBoxes
5:    Ajouter (' < type > ', posChoice) à allBox
6: end for
```

lined 3 – Algorithme de création (rivières, montagnes ...)

Dans cet algorithme, on choisi une case aléatoirement et on l'enlève de la liste initiale. On l'ajoute à la nouvelle liste avec le type de la case. Enfin, pour les cases de type plaines, on prend toutes les dernières cases possibles.

L'algorithme de recherche n'est pas compliqué mais peut nécessiter quelques explications. Prenons l'exemple de la méthode "research_one_valid_box" :

```
Require: boxes liste comportant toutes les cases possibles
 1: for all box ∈ boxes and box.type ≠' River' do
 2:    if box.centerBox = position then
 3:       if box.object = None then
 4:          return box
 5:       else
 6:          Cassage de la boucle {car chaque case est unique}
 7:       end if
 8:    end if
 9: end for
10: return None
```

lined 4 – Algorithme de recherche de case valide

Ici, nous recherchons dans toutes les cases de la carte si la position donnée en argument de la méthode correspond à la position du centre de la case. Si elles correspondent, on vérifie s'il y a une unité sur cette case, s'il n'y en a pas, on retourne la case sinon on casse la boucle. Ceci nous permet de terminer l'itération de la boucle et donc d'optimiser la durée d'exécution du code. Si on sort de la boucle et qu'aucune case n'a été trouvé, on retourne *None*.

3.1.3 Fichier 'utils.py'

Dans ce fichier, un algorithme de sélection des unités possibles a été utilisé avec des variantes dues par qui fait le choix (le joueur ou l'ordinateur). Ceci permet d'encadrer les joueurs dans les choix de l'armée et donc de ne pas dépasser la limite des points disponibles pour créer l'armée. L'algorithme suivant est une version simplifiée des deux variantes :

Require: *points* points encore disponibles pour créer l'armée
1: **if** $points \geq 40$ **then**
2: Toutes les unités peuvent être choisies
3: **end if**
4: **if** $points < 40$ **then**
5: Le Horseman ne peut plus être choisi
6: **end if**
7: **if** $points < 30$ **then**
8: Horseman et Knight ne peuvent plus être choisis
9: **end if**
10: **if** $points < 20$ **then**
11: Seul le Warrior peut être choisi
12: **end if**
13: $unitType \leftarrow$ Choix de l'unité ... (explications après)
14: **return** $unitType$

lined 5 – Algorithme de sélection d'unité

Il y a cette base logique dans les deux fonctions qui permettent de choisir une unité. Pour celle du joueur, on applique une vérification pour voir si l'unité existe et si le joueur entre autre chose qu'un nombre, ceci renvoie une erreur stoppant l'exécution du programme. Pour l'ordinateur, on fait un choix aléatoire dans les unités disponibles (pas besoin de vérifier).

3.1.4 Fichier 'simulator.py'

Ce fichier permet de faire le combat de deux armées en console. Il y a deux algorithmes importants. Le premier permet à l'ordinateur de choisir une unité adverse (pseudo Intelligence Artificielle) et l'autre permet à une unité d'attaquer un adversaire choisi par l'algorithme précédent. Nous allons nous pencher vers les algorithmes pour l'ordinateur car pour les joueurs, la base est la même est les choix ne sont fait que par un humain et donc par une logique humaine et pas algorithmique ou mathématique.

L'algorithme suivant permet la sélection de l'unité ennemie que l'ordinateur attaquera :

Require: *enemyList* liste des ennemis
1: $HPMin = 6000$
2: $unit2 = None$
3: **for all** $enemy \in enemyList$ **do**
4: **if** $enemy.hp + enemy.armor < HPMin$ **then**
5: $unit2 \leftarrow enemy$
6: $HPMin \leftarrow enemy.hp + enemy.armor$
7: **end if**
8: **end for**
9: **if** $unit2 = None$ **then**
10: **return** Ennemi tiré aléatoirement de la liste {Si aucune unité n'a été trouvée}
11: **end if**
12: **return** Ennemi qui a l'index de $unit2$

lined 6 – Algorithme de sélection d'unité ennemie à attaquer

On peut voir que l'algorithme sélectionne l'unité qui a le moins de points de vie et d'armure

additionnés. S'il n'en trouve pas, il en tire un au hasard.

Voici le code utilisé pour attaquer l'unité ennemie choisie par l'algorithme précédent :

```
try:
    enemy_index = enemyChoiceByComputer(left_army.full_army[i],
       right_army.full_army)
    left_army.full_army[i].attack(right_army.full_army[enemy_index])
    if right_army.full_army[enemy_index].die():
        right_army.full_army.remove(right_army.full_army[enemy_index
           ])
except IndexError:
    if right_army.full_army == [] or left_army.full_army == []:  #
       Si l'une des armée est vide
        in_progress = False
    else:
        left_army.full_army[i].attack(right_army.full_army[-1])
        if right_army.full_army[-1].die():
                        right_army.full_army.remove(right_army.
                           full_army[-1])
```

Ici, nous faisons une gestion d'erreur. On exécute les instructions qu'il y a dans le *try* puis s'il y a une exception, on la capte (principalement la liste est soit vide, soit l'unité trouvé n'est plus dans la liste). Dans le *try*, on fait attaquer l'unité alliée sur l'unité ennemie et on vérifie si l'unité ennemie est morte. Si elle l'est, on la retire de la liste des ennemis. Dans le *except*, on vérifie la l'exception est due à une des liste est vide. Si aucune des armées est vide, on refait ce qu'il y a dans le *try* mais on attaque la dernière unité de la liste des ennemis.

3.1.5 Fichier 'thread_simulation.py'

Un algorithme de calcul est utilisé pour calculer le *fitness* qui servira dans l'algorithme génétique. Voici l'algorithme :

Require: $enemyList$ liste des ennemis
Require: $allyList$ liste des alliés
Require: $len(allyList)$ longueur de l'armée alliée
1: $sum = [len(allyList) * \sum_{ally \in allyList}(ally.hp + ally.armor) * ally.damage] - sum(enemyList)$
{$sum(enemyList)$ la même somme mais pour l'armée ennemie}

lined 7 – Algorithme de calcul du *fitness*

Ceci nous permet d'avoir qu'une seule valeur en sortie ce qui va simplifier grandement la sélection pour l'algorithme génétique.

3.1.6 Fichier 'ga_wargame.py'

Dans ce fichier, plusieurs algorithmes de création d'armée se trouve ici mais ils sont similaires aux algorithmes de création vus plus hauts. Un algorithme différent des autres est utilisé pour la mutation.

```
for single_army in armies:
    for index, old_unit in enumerate(single_army.army_base):  # Énum
       ération des unités de l'armée qu'on veut muter
```

```
            if uniform(0, 1.0) <= 0.05:  # Probabilité de mutation de 5%
                position = unit_position_choice(single_army.
                    position_available)
                new_unit = (old_unit[0], position.center_box, old_unit
                    [2])
                single_army.army_base = single_army.army_base[0:index] +
                     [new_unit] + single_army.army_base[index + 1:]
                break  # Permet de muter seulement une unité dans l'armé
                    e
```

Le code cidessus permet de faire muter une seule unité dans la liste des armées qui sont actuellement itérées. L'algorithme est différent car nous utilisons de la probabilité pour simuler la sélection naturelle. Dans le code, on cherche à faire une mutation avec une probabilité de 5%.

3.2 Structures de données

3.2.1 Qu'est ce qu'une structure de données ?

C'est une manière d'organiser les données dans le but de les traiter le plus rapidement et plus facilement. Cela permet un traitement automatique des données.

3.2.2 Comment l'a-t-on appliqué ?

Dans les fichiers 'mapping.py' et 'army.py', pour créer l'armée ou la carte, nous avons du utiliser deux listes différentes pour chaque classe (Army et Map). Par exemple, la première sert à garder l'armée de base et la seconde, permet d'avoir l'armée sous forme d'objet, donc avec les différentes unités et leurs propres attributs. Nous avons été obligé d'utiliser ceci quand nous devions mettre en place l'algorithme génétique avec la recréation systématique de l'armée ennemie ou de la carte pour que nous simulions toujours la même armée ennemie sur la même carte mais avec des armée alliées différentes. Cette organisation nous permet de traiter plus rapidement afin que la durée de l'exécution de l'algorithme génétique soit amoindrit.

Dans d'autres cas, nous avons préféré utiliser des dictionnaires car cela nous semblait beaucoup plus facile à utiliser. Par exemple, dans le fichier 'WarGame.py', nous avons utilisé des dictionnaires pour faire passer les différents arguments dans les fonctions (utilisé pour les boutons). On utilise donc la syntaxe $**kwargs$ pour "key word arguments". Ceci a donc été une bonne idée car dans les dictionnaires, c'est un ensemble de clés et de valeurs et dans les arguments d'une fonction, c'est la même chose donc l'utilisation de dictionnaire dans ce cas est justifié.

Nous avons aussi utilisé des listes par compréhension ce qui nous a permit de réduire la taille des fichiers. Nous l'avons utilisé dans le fichier 'mapping.py' à quelques reprises car les boucles for à faire étaient simples et on pouvait donc les réduire à une simple liste par compréhension.

Enfin, nous avons utilisé des tuples afin de ne pas pouvoir modifier le contenu de celui-ci. Cela nous a été utile lors de la création de l'armée quand une unité est choisie, son type, sa position et son coût ne puisse pas être changé mais que l'on puisse quand même y accéder rapidement et facilement (comme dans une liste).

3.3 Bibliothèques

3.3.1 Modules standards de Python

Le module *random* de Python a été utilisé à de nombreuses reprises notamment pour la sélection des unités avec les fonctions *choice* et *randint*, mais aussi pour la mutation des armées dans le fichier 'ga_wargame.py' où l'on utilise la fonction $uniform$.

Le module *math* est exclusivement utilisé dans le fichier 'units.py' pour calculer la distance entre deux points avec la fonction *sqrt*.

Le module *threading* est très important et est utilisé dans 'thread_simulation.py'. Pour le WarGame, il nous sert à améliorer la vitesse d'exécution de l'algorithme génétique en exécutant la même instruction en parallèle et indépendamment au lieu d'une exécution normale les unes après les autres. On s'en sert pour faire une simulation incluant la recréation de l'objet *Army* correspondant à l'armée ennemie. Ce fichier nous a permit d'augmenter la vitesse d'exécution et ces chiffres sont données dans la partie 5, dans la sous-partie sur les mesures de performance.

Le module *time* nous a principalement servi pour regarder la vitesse d'exécution ou pour mettre en pause le programme.

Les modules *os* et *system* ont été utilisé à des fins techniques et esthétiques. Techniques tout d'abord pour le chargement des fichiers qui ne sont pas en '.py' et esthétiques, pour nous permettre de mieux afficher lors du travail en console.

Le module *pickle* est utilisé dans le fichier 'WarGame.py'. Il nous permet de sauvegarder les données de la partie pour que le joueur puisse continuer de jouer plus tard s'il le souhaite.

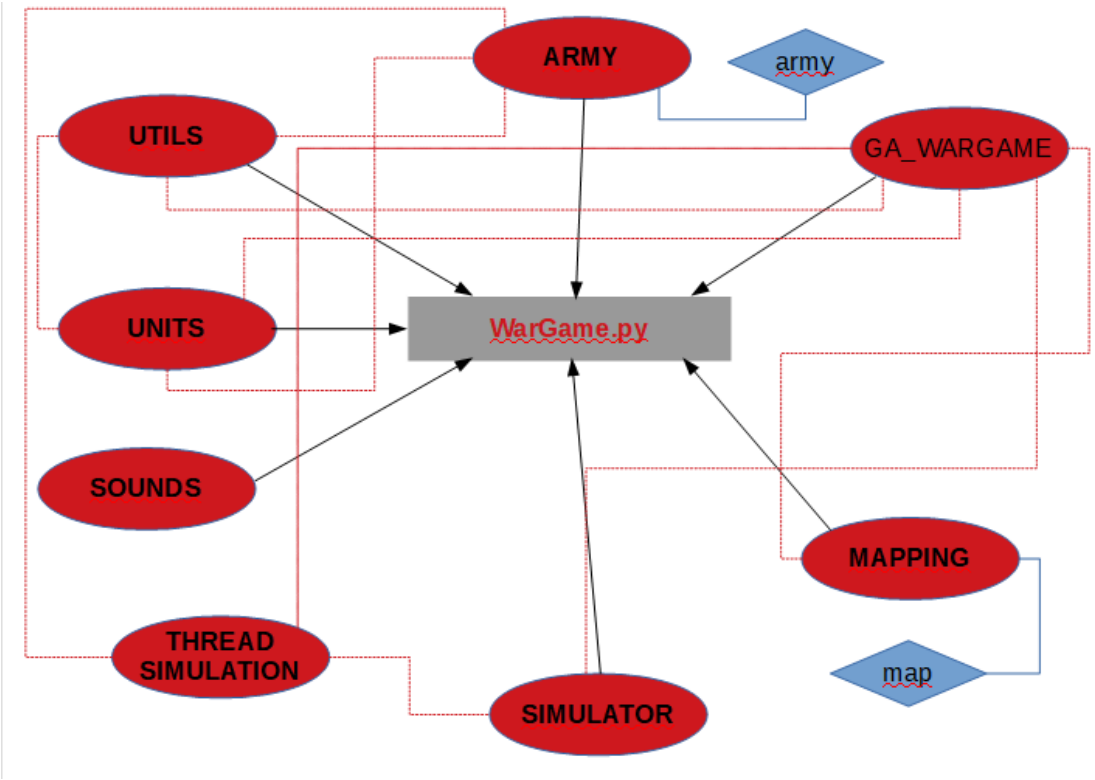
3.3.2 Modules non natifs de Python

Le seul module non natif qui a été utilisé est 'PyGame'. Il nous permet de créer l'interface graphique du jeu en y intégrant facilement des images, des musiques, ...

Avec cette bibliothèque, nous avons décidé d'intégrer directement des images pour les différents menus. Nous utilisons une sous-bibliothèque 'gfxdraw' que nous devons importer du module 'pygame'. Ceci nous permet de créer des figures géométriques ce qui nous a permit de dessiner les cases de la carte ou encore de représenter les unités par des cercles ou encore une couronne.

4 Architecture du projet

4.1 Diagrammes des modules et des classes



4.2 Cas d'utilisation

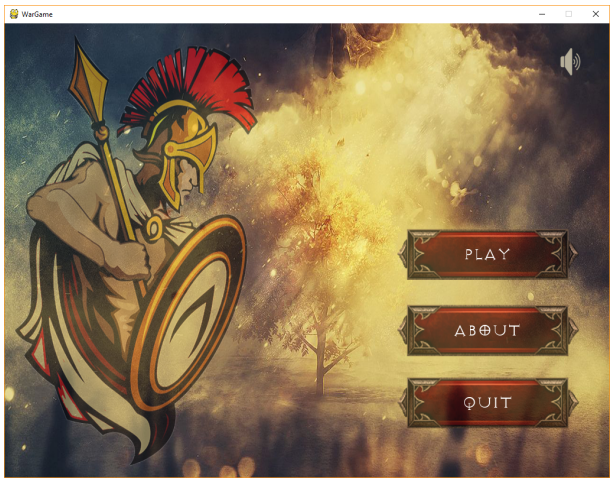
Pour subvenir au bon fonctionnement de notre application, nous avons utilisé le programme de PyGame, pour ce qui va être de l'interface graphique. Ceci donne à l'utilisateur une meilleur approche lorsqu'il utilise l'application. Pour traiter plusieurs tâches en même temps, nous sommes partis sur le threading, ce qui permet au programme de s'exécuter avec efficacité, en exécutant ses tâches parallèlement plutôt qu'une par une, ce qui donne à l'utilisateur un temps d'attente quasiment inexistant.

4.3 Chaîne de traitement

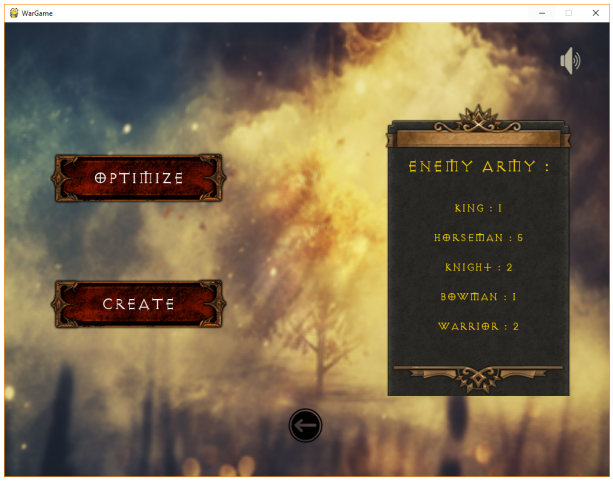
La chaine de traitement du programme va commencer par le lancement d'une interface graphique à l'aide PyGame sur laquelle on pourra exécuter plusieurs programmes en fonction du choix de l'utilisateur. Celui-ci aura le choix entre exécuter la suite du programme ou bien s'arrêter ici. Si l'utilisateur choisi de poursuivre le programme, celui-ci va donc enchaîner sur une nouvelle interface qui celle ci choisira entre se servir de l'optimisateur ou non, peut importe le choix que l'utilisateur va faire, celui-ci va se conclure sur le lancement du programme principal écris en python. Dedans vous allez pouvoir trouver le threading qui va permettre de faire fonctionner le tout efficacement et rapidement. Arriver à ce point, ça va être de nouveau PyGame qui va revenir en proposant une interface dynamique sur laquelle l'utilisateur va pouvoir poursuivre ses activités jusqu'à ce que fin s'en suive.

5 Expérimentations et usages

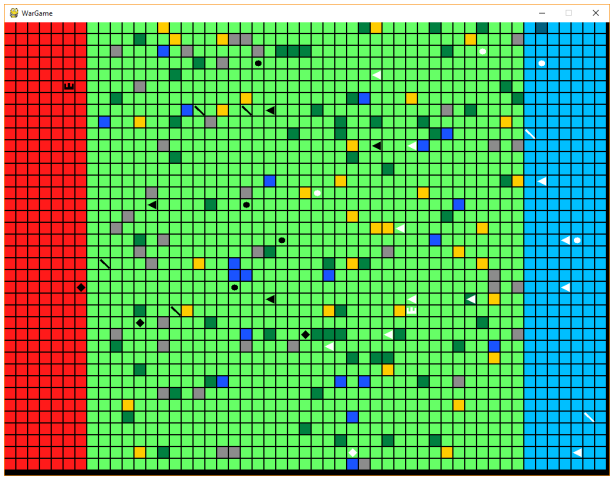
En ce qui concerne l'usage du jeu, tout d'abord le menu principal donne un impact d'un jeu de guerre. Premièrement en coté graphique, la charte graphique est choisie avec des couleurs chaudes (rouges, jaunes et orange) pour donner l'impact du feu et l'atmosphère de guerre. La police (DIABLO.ttf) donne une force à l'écriture amplifiant l'atmosphère guerrière.



Après que l'utilisateur rencontre le menu principal, il doit commencer le jeu. Il a deux choix. Il peux choisir lui même ses unités pour construire l'armée ou bien, il peut faire le choix de l'optimisation. Dans ce cas précis, c'est l'ordinateur lui même, d'une façon aléatoire et répétant la sélection naturelle, qui choisit pour l'utilisateur une armée avec des unités en ne dépassant pas la limite de points choisie aléatoirement par le jeu.



Le jeu fonctionne d'une façon optimisée pour un wargame. Cela signifie que ce n'est pas un vrai jeu de guerre, mais avec une carte et des représentations pour chacune des unités, alors dans tout le jeu, l'utilisateur doit éliminer toutes les unités de l'armée ennemie pour gagner le jeu. Mais en regardant le jeu d'un coté statistique.



Qu'est ce qu'il rend ce jeu plus performant ?

D'abord, pour la création des simulations, normalement cela l'exécution d'une seule simulation durait 1 seconde. Après une première optimisation, la durée d'exécution s'est réduite à 0.4 seconde.

Ensuite, lors de la création de l'algorithme génétique, pour une population de 10, une génération de 25 et une limite de 300 points disponibles, la durée était de 100 secondes en moyenne (cette durée est variable car elle dépendait de la résistance des armées).

Puis, de nouvelles optimisations du simulateur ont été faites ce qui a diminué les valeurs données ci-dessus. Pour une seule simulation, nous sommes arrivés à une durée de 0.22 seconde en moyenne. Pour l'algorithme génétique avec les mêmes paramètres, nous étions descendus à 55 secondes environ.

Enfin, nous avons décidé de créer des threads pour l'exécution parallèle. Ceci nous a permis d'obtenir les résultats actuels. Pour une simulation, nous sommes à 0.22 seconde en moyenne. Pour l'algorithme génétique, nous sommes actuellement à 10 secondes environ.

6 Conclusion

Pour récapituler, ce jeu est optimisé. Le joueur (qui a choisi de créer son armée ou a choisi l'armée optimisée) joue contre l'ordinateur qui est muni d'une pseudo intelligence artificielle créée par notre groupe. Celui qui élimine toutes les unités de l'armée ennemie est gagnant. Ce jeu est un mixte entre un jeu de stratégie et un wargame. De plus, quelques idées originales se trouve dans le projet. Par exemple, la possibilité de créer une infinité de carte toute différente, le menu principal qui a été créé avec PhotoShop et chargé par python en ajoutant des animations. Pour donner un exemple, lorsqu'on passe la souris au dessus d'un bouton, on passe un filtre sur celui-ci donnant l'impression de diminution d'opacité, tout cela à l'aide de l'indication des cordonnées de ce bouton et en utilisant PyGame...

Notre projet est créé à l'aide de tous les membres du groupe qui ont aidé a développer ce jeu, toutes ces semaines, en proposant des idées et en testant plusieurs méthodes pour dépasser les difficultés. Mais, on avait des idées et des propositions pour améliorer le jeu. Voici quelques idées pour améliorer le jeu :

- passer d'un jeu solo contre l'ordinateur à un jeu multijoueur en faisant une liaison online entre les deux joueurs pour jouer à distance (comme un jeu d'échecs) ;
- ajouter de nouvelles unités ;
- implémenter un système de niveau et d'accomplissement dans le jeu.

Nous avions pensé créer des vraies cartes en 2D à l'aide d'un programme se nommant titled au lieu de générer une carte avec PyGame. Nous avons du nous résoudre à abandonner cette idée car nous avons rencontré des problèmes d'ordre théoriques comme trouver comment appliquer et faire fonctionner ceci avec la partie cachée de notre jeu.