# Rapport de soutenance du projet

Daniel MURRAY, Léo MALUBIER, Martin AUBLET, Hasti KHAYAMI

11 mai 2020

# 1 Tableau de Contenu

## Table des matières

# 2 Manuel d'utilisation

## 2.1 Lancement du programme

— Windows :
  — Naviguer dans le dossier "src", et double-cliquer sur le fichier "wargame.py" se-trouvant à la racine du dossier pour lancer le simulateur.
— Linux :
  — Ouvré un terminal dans la racine du dossier, et saisir le suivant :
  — python src/wargame.py

## 2.2 Utilisation

Lors du lancement du programme, 3 options sont proposés.
— "Simuler une bataille" - Simule un combat singulier entre deux armées à une vitesse ralenti, pour pouvoir visualiser les statistiques d'un combat tour-par-tour.
— "Optimiser une armée" - Générer une armée avec un pourcentage de victoire maximisé, basé sur un plateau de jeu et des simulations de combat.
— "Quitter" - Fermer le programme.

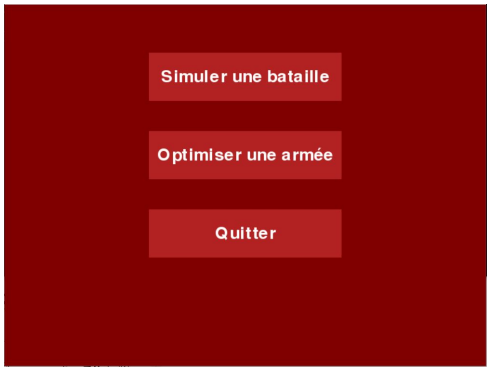


**Figure 1** – Le menu principale du programme.

### 2.2.1 Simuler une bataille

Dans cet section du programme, on simule un combat singulier, entre deux armées sur un plateau de jeu, à une vitesse ralenti, pour pouvoir visualiser les statistiques d'un combat tour-par-tour Quand cet section est ouvert, le programme commence par générer un plateau de jeu. Quand ceci est fait, le programme attend que l'utilisateur clique sur la barre d'espace pour commencer la simulation. Une simulation est alors effectué, un tour à la fois. A

la fin de chaque tour, plusieurs données sont visibles et accessibles à l'utilisateur, pour permettre la visualisation et analysé les résultat.
A la fin de la simulation, les résultats de la bataille sont affichés.

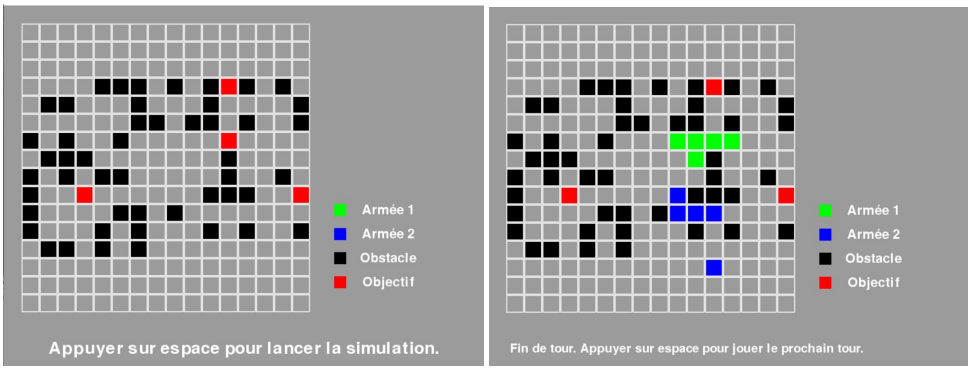


**Figure 2** – Captures d'écran d'une simulation.

### 2.2.2 Optimiser une armée

Dans cet section, on lance plusieurs simulations avec différents armées générés aléatoirement, pour optimiser et pouvoir générer une armée qui sera la meilleur. Ce processus est largement optimisé grâce à des algorithmes et fonctions, qui nécessite très peu d'action de la part de l'utilisateur.
A la fin du processus, le programme affiche la meilleur armée. On peux voir les statistique dans la console, et l'armée qui a remporté.

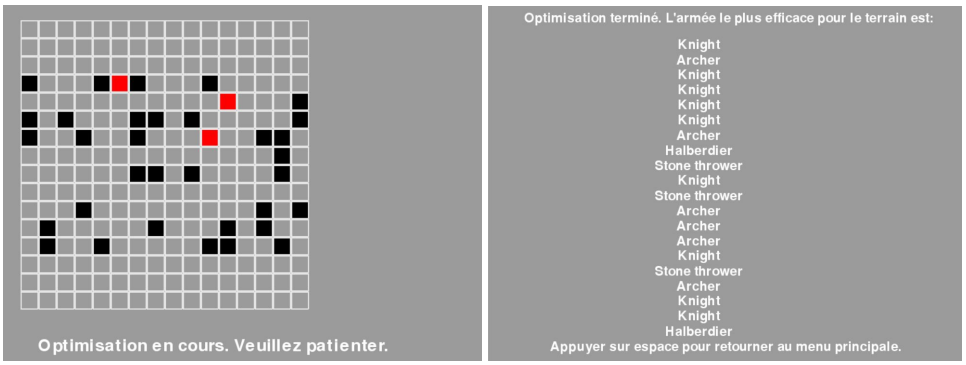


**Figure 3** – Captures d'écran d'une optimisation.

# 3 Organisation

## 3.1 Répartition des tâches

| | |
|---|---|
| Codage des objets et modèles,visualisation simulation | Daniel MURRAY |
| Combat, optimisation, traitement des données simulation | Léo MALUBIER |
| Interface graphique, sprites | Martin AUBLET |
| Codage graphique menu, PyGame | Hasti KHAYAMI |

## 3.2 Ecriture des fichiers

| | |
|---|---|
| modeles.py,optimisateur.py | Daniel MURRAY |
| combat.py,combat.py | Léo MALUBIER |
| mainmenu.py, unites.py | Martin AUBLET |
| mainmenu.py, modeles.py | Hasti KHAYAMI |

## 3.3 Contact

| | |
|---|---|
| Daniel MURRAY | 07 52 05 17 13 |
| Léo MALUBIER | 06 38 01 14 29 |
| Martin AUBLET | |
| Hasti KHAYAMI | 07 79 46 62 24 |

# 4 Objectifs du projet

## 4.1 Description du concept

Le but de ce projet est de réalisé un simulateur et optimisateur de "wargame", qui permet de construire une armée qui aura un pourcentage maximum de victoire, en relation du terrain sur le plateau du jeu, et contre un autre armée quelconque.

## 4.2 Les éléments existants

La pratique du "wargame" est principalement appliquée par le militaire pour des entraînements stratégiques, mais il existe aussi une vaste variété de jeux de société qui ont pour but de recréer cet expérience de manière amusant.

Des différentes variantes existent avec leurs propres règles et procédures. Pour maximiser la simplicité et la compréhension du côtés programme et utilisateur, nous avons choisi d'utilisé un l'ensemble de règles trouvés dans "Grimdark Future: Firefight" comme base. Nous avons aussi modifié ou supprimé certains règles et procédures ci-trouvant pour gardé la simplicité du

déroulement.

Notre langage de programmation choisi est Python, pour ses capacités d'écriture simple et sa robustesse. De plus, nous intégrons aussi le bibliothèque "PyGame", qui rend possible des fonctions et méthodes pour l'affichage graphique.Un module math est aussi utilisé, et deepcopy.

### 4.3 Les éléments à développer

Pour pouvoir créer un optimiseur de wargame, notre but est divisé en quatre taches :

1. Recréer l'ensemble des règles choisis de wargame dans python.
2. Simuler le combat entre les armés en respectant l'ensemble de règles choisis.
3. Développer un optimiseur qui générera une armée efficace en relation avec le système de combats lors de l'étape précédente.
4. Afficher le tout dans un interface graphique simple et facile à comprendre et utiliser.


**Figure 4** – Logo de Grimdark Future : Firefight.


**Figure 5** – Logo de PyGame.

# 5 Fonctionnalités implémentées

## 5.1 Description des fonctionnalités

Notre projet peut être divisé en quatre parties principales. Chaque résultat d'une partie peut contribuer ou fournir les variables nécessaires pour une autre.

Le simulateur simule des combats. Si nous souhaitons simuler le combat entre 15 armées, on effectue 25*26 combats. On prend une armée qui va combattre les 25 autres armées, puis repeter ceci 26 fois pour tester les autres armées. Chacune des armées à ses propres unités crées à partir du format trouvé dans l'ensemble des règles, sur un plateau de jeu généré aléatoirement avec des objectifs et des obstacles.

Chaque résultat de chaque combat est enregistré, et l'armée avec le plus de victoire est considéré comme gagnant, ou pourra passer a l'étape suivant qui est l'optimisation.

L'intégralité de notre programme sera encadré par une interface graphique, à travers les fonctions d'affichage et rendus grâce à PyGame.

## 5.2 Organisation du projet

Notre groupe de projet est constitué de 4 personnes. Nous nous somme donc répartis les taches que nous voulions réalisé.

— Une personne code et transforme l'ensemble de règles utilisés en fonctions, classes et objets dans Python.

— Un personne interprètent les instructions et règles pour concevoir un arbre de décision pris par l'ordinateur, les transforment en pseudo-code/algorithmes, puis ensuite le traduit et écrit en language de programmation.

— Une personne développe une système d'optimisation pour pouvoir gérer les résultats des combats simulés dans le programme, et ensuite analyse et développe une armée avec des taux maximisé.

— Une personne développe un système de combat qui sera utilisé lors de la simulation. Elle prendra en charge plusieurs fonctions de gestion de variables, attaque et défense, les points de vie d'une unité...

— Une personne prend en charge les graphiques nécessaires. Ils créent et dessinent tous les sprites et images nécessaires, ensuite ils développe un système pour pouvoir gérer l'affichage graphique, et faire afficher les sprites et objets nécessaires.

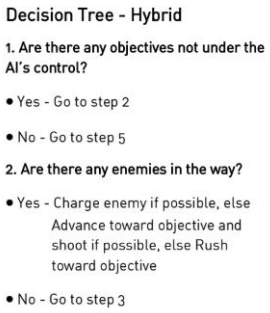


**Figure 6** – Un extrait de l'arbre de décision prise par les armés controllés par l'ordinateur. Cet arbre sera ensuite traduit en langage de programmation.

# 6 Elements techniques

## 6.1 Algorithmes

### 6.1.1 Génération du plateau de jeu

Avant de pouvoir se lancer dans un combat entre deux armées, il est nécessaire de créer un plateau de jeu, où la simulation pourra se dérouler. Ce plateau sera constitué d'une grille 16*16, pour un totale de 256 cases. Le plateau de jeu contiendra aussi des objectifs, que les armées doivent capturer et défendre, et des obstacles, ou les armées ne peuvent pas se déplacer directement dessus, mais ils peuvent néanmoins les utiliser comme protection contre les attaques.

Pour générer notre plateau de façon aléatoire, nous suivons l'algorithme suivant :

- Créer une grille vide avec 16 lignes et 16 colonnes.
- Choisir une nombre aléatoire entre 10 et 15, ce nombre correspondra au nombre d'obstacles qui sera mis sur le plateau.
- Choisir une nombre aléatoire entre 1 et 3, et lui ajouter 2 - ce nombre correspondra au nombre d'objectifs qui sera mis sur le plateau.
- Itérer à travers chaque case dans les colonnes de la grille, et génerer une nombre aléatoire. Si ce nombre se trouve dans un seuil, cet case sera un objectif ou un obstacle.
- Répéter l'étape précédant jusqu'à ce que le nombre d'objectifs et d'obstacles définis est atteint.
- Effacer les trois premiers et trois derniers lignes de la grille - ces espaces seront utilisés pour le déploiement d'armée, elles doivent être alors vides pour que les armées puissent se déployer sans problème.

### 6.1.2 Arbre de décision des unités

Lors des combats entre chaque armée, l'ordinateur doit calculer quel décision prendre - se déplacer, attaquer une autre unité, etc. On appelle ceci "arbre de décision". Il existe déjà un arbre de décision dans les documents des ensembles de règles (voir Figure n.3).

En général, l'algorithme pour l'arbre de décision (fonction computeOptions fichier modeles.py )est défini par le ceci :

- Regarder s'il y a des objectifs non contrôles. Si oui, avancer vers l'objectif, et attaquer un ennemi si possible.
- Si non, analyser s'il y aura des ennemis en portée si l'unité avance. Si oui, avancer et attaquer l'ennemi. Si non, charger dans la direction de l'ennemi.
- Répéter les étapes précédentes pour tous les unités dans une armée.

```
Table = 16*16 list
MaxNumOfObstacles = RandNum (10,15)
MaxNumOfObjectives = (RandNum(1,3)) + 2
NumOfObstaclesLeft = MaxNumOfObstacles
NumOfObjectivesLeft = MaxNumOfObjectives
Seed = 0

While NumOfObjectivesLeft > 0:

For a in list:


For b in list[a]:
seed = RandNum(0,3)
If Seed == 0:

If List[a][b] == Empty:

List[a][b] = Objective

NumOfObjectivesLeft = NumOfObjectivesLeft - 1

While NumOfObstaclesLeft > 0:

For a in list:

For b in list[a]:

seed = RandNum(0,3)
If Seed == 0:

If List[a][b] == Empty:

List[a][b] = Obstacle
NumOfObstaclesLeft = NumOfObstaclesLeft - 1
ShowGrid
```

**Figure 7** – L'algorithme transformé en pseudo-code.

```python
# Ajouter l'emplacement des objectifs dans le champ.
def setUpObjectives(self):

    print("Generation de ", self.numOfObjectives,   " objectifs...")
    numOfObjectivesLeft = self.numOfObjectives
    seed = 0
    alreadyPlaced = 0

    while numOfObjectivesLeft > 0:
        for a in range(3,len(self.grid)-3):
            for b in range(len(self.grid[a])):
                seed = random.randint(0,50)
                if seed == 1:
                    if alreadyPlaced == 0:
                        if numOfObjectivesLeft > 0:
                            self.grid[a][b] = "X"
                            alreadyPlaced = 1
                            numOfObjectivesLeft = numOfObjectivesLeft - 1
                            b = len(self.grid[a])+1

                alreadyPlaced = 0
                seed = 0

# Fonction pour génerer des obstacles sur le terrain.
def putObstaclesInMap(self, n):

    #Declaration des variables nécessaires.
    numOfObstaclesLeft = n
    seed = 0
    alreadyPlaced = 0

    #Algorithme principale pour le terrain.
    #Le terrain est representé par le caractère suivant: *
    while numOfObstaclesLeft > 0:
        for a in range(len(self.grid)):
            for b in range(len(self.grid[a])):
                seed = random.randint(0,3)
                if seed == 1:
                    if alreadyPlaced == 0:
                        self.grid[a][b] = "*"
                        alreadyPlaced = 1
                        numOfObstaclesLeft = numOfObstaclesLeft - 1
                        b = len(self.grid[a])-1

                    alreadyPlaced = 0
                    seed = 0
```

**Figure 8** – L'algorithme transformé en langage Python.

Chaque partie de l'algorithme aura un sous-algorithme, qu'on expliquera en détail plus tard.

### 6.1.3 Boucle de jeu

Notre boucle de jeu est un des algorithmes le plus important dans notre programme( fonction BestArmies fichier combat.py), puisque c'est elle qui gère la plupart des évènements, et sert à faire la gestion entre tous les autres sous-composantes (menus, fonctions, classes, objets...). Lors du lancement d'un simulation de combat, la boucle de jeu est mise en fonction.

En général, le boucle de jeu suit l'algorithme suivant :

- Générer le plateau du jeu.
- Appelle la création d'un certain nombre d'armées aléatoire.
- On prend une armée, et on la fait combattre toutes les autres. (On prend l'armée qui va combattre toutes les autre, on l'envoie dans la fonction boucle qui va poursuivre l'attaque tant qu'une des deux armées n'est pas battue.) Renvoyer l'armée victorieuse.
- Dans le combat, on va simuler une lancer de pièce pour décider quel armée pourra déployer ses unités sur le plateau en premier.
- Déployer les armées sur le plateau.
- Pendant qu'il y a au moins une unité de vivant dans une armée, simuler l'arbre de décision pour chaque unité dans une armée, le déplacer sur le plateau, et, si disponible, faire appel aux fonctions de combat.
- Quand il n'y a plus d'unités de vivant dans une armée, terminer la boucle, puis passer au test de l'armée suivante.

```
GenerateTerrain()

CoinToss = RandNum(1,2)
If CoinToss == 1:
FirstArmy = Army1
Else If CoinToss == 2:
FirstArmy = Army2

DeployArmies(FirstArmy)

While True:
For a in range(0,Army1.length):
ComputeUnitTurn(Army1.length[a])
For b in range(0,Army2.length):
ComputeUnitTurn(Army2.length[b])
if (Army1.length == 0) or (Army2.length == 0):
EndGame()
```

**Figure 9** – Le boucle de jeu transformé en pseudo-code.

```
# Commencer notre boucle de jeu.
def beginGameLoop(self,screen):
    font_obj = pygame.font.Font('freesansbold.ttf', 20)
    print("Démarrage du boucle du jeu.")
    print("Decision du premier armee pour le deploiement...")
    beginningPlayer = self.rollOff()
    print("L'armée", beginningPlayer, "a gagné le roulement.")

    print("Deploiement des armees...")
    self.deployArmies(self.gameBattlefield.grid)
    updateGraphicsGrid(screen,self.gameBattlefield.grid,self.gameBattlefield.gridCoordinates)


    # Le jeu se lance a partir d'ici.
    gameLoop = True
    while gameLoop == True:
        for a in range(1,3):
            print(a)
            if a == 1:
                #Armée 1
                print(" ++++++++ Tour du armée 1 ++++++++")
                for b in range(0,len(self.gameArmy1.armyUnits)):
                    self.unitActionPrompt(self.gameArmy1, self.gameArmy1.armyUnits[b],self.gameBattlefield.grid,1)
                    updateGraphicsGrid(screen,self.gameBattlefield.grid,self.gameBattlefield.gridCoordinates)
                    time.sleep(.250)
                    pygame.event.get()
            elif a == 2:
                print("++++++++ Tour du armée 2 +++++++++")
                for b in range(0,len(self.gameArmy2.armyUnits)):
                    self.unitActionPrompt(self.gameArmy2, self.gameArmy2.armyUnits[b],self.gameBattlefield.grid,2)
                    updateGraphicsGrid(screen,self.gameBattlefield.grid,self.gameBattlefield.gridCoordinates)
                    time.sleep(.250)
                    pygame.event.get()
            self.endGame(2,screen)
```

**Figure 10** – Un extrait du boucle de jeu transformé en langage Python.

### 6.1.4 Optimisation d'armée

Le simulateur de combat est le premier des parties majeurs de notre programme ; l'optimisateur d'armée étant le deuxième. Elle sert à pouvoir générer une armée à partir d'un plateau de jeu, et des résultats de plusieurs combats simulés entre différents armées. Le résultat est une armée, qui possèdera un taux de victoire plus élevé.

L'optimisation d'armées suit l'algorithme suivant :

— Une armée a été choisis comme étant la plus forte (fonction BestArmies).

— On passe par la fonction CalculStat qui permet de comparer les statistiques des deux armées les plus faible et les deux armées les plus puissantes, pour savoir quelle est le point d'écart le plus élevé entre les armé perdantes et gagnants (on compare la vie, le coût l'agilité, le mouvement et la porté) et on compte combien de chaque unité, il y a dans les armées (des chevaliers, des archers, etc).

— Ensuite, on a un arbre de décision qui permet de modifier les armées, selon les condition que l'on a déduit grâce au calcule de la fonction CalculStat. Ainsi,on définit les point a maximiser (agilité par exemple) et on applique des changements. Cependant, on vérifie que l'ensemble des unités que l'on change ne contiens pas un nombre significatif pour l'armé d'unité correspondant au changement. (ici défini par le nombre total d'unité fois 1/5, ce qui est généralement autour de 4.

— Une simulation va nous permettre de savoir si les modification apporté rend meilleur l'armée ou non. Pour cela on l'affrone x fois l'armé la plus faible, et on fait de même avec sa version précédente. Puis l'on compare qui a le plus haut taux de victoire en pourcentage.

— Enfin on affiche l'armée avec le plus haut taux de victoire suite au modifications. Si l'armée modifié a un taux de réussite plus haut que l'armé de départ, alors elle est meilleur. Au contraire, elle peux être moins bonne, ce qui signifie que la meilleur armée était celle de départ et que l'on ne peux pas l'améliorer.

## 6.2 Structures de données

La majorité des composantes qui font partie d'un wargame (exemples : le plateau du jeu, chaque unité d'une armée, etc) peuvent être répliqués sur Python avec la notion des "objets" dans Python. Chaque objet aura la possibilité de porter ses propres données, variables et fonctions. De plus, chaque objet peut être "instantané" pour pouvoir en créer plusieurs objets d'une même classe. Ceci est très utile pour la gestion de plusieurs éléments en même temps. (Exemple : tous les unités dans une armée sont tous des objets)

Dans cet section, nous entrons en détail dans le structure de quelque objets critiques dans notre programme.

### 6.2.1 Les armées

Chaque armée peut être considéré comme objet Python. Pour faciliter la compréhension et gestion de chaque unité dans notre armée, tous les objets unité seront enregistré dans une liste, qui sera ensuite stocké par notre objet armée. Lors du création de l'armée, l'objet fait appel à un fonction qui génère des objets pour les unités aléatoire, puis les stocke dans une liste dans notre objet.

— maxPoints : Le cout maximum que tous les unités d'un armée peuvent y être.
— maxArmyUnits : Le nombre maximum d'unités dont l'armée peut y être.
— armyUnits : Une liste qui stockera tous les objets Unité lors du création de l'armée.

### 6.2.2 Les unités

Chaque unité d'une armée a plusieurs variables qu'on peut facilement manipuler lors de la simulation de combat. Lors de l'initialisation de cet objet, certaines variables sont passés en argument, pour structurer l'objet (la santé, endroit sur le plateau du jeu, etc...)

Variables :

— Health : Le points de vies de l'unité.
— Wounds : Le nombre de blessures dont l'unité porte. Si cet variable est égale ou supérieure a Health, l'unité est alors supprimé de l'armée et du plateau du jeu.
— Location : Les coordonnées x et y de l'unité sur le plateau du jeu. Elle est représente comme une liste [x,y].
— Name : Le type de l'unité. Representé avec un String.
— Cost : Le cout de l'unité.

### 6.2.3 Le système de jeu

Le système utilisé pour suivre et gérer chaque simulation est un des composants le plus importants dans notre programme (voir partie 4.1.3). Il dirige la boucle principale de notre programme, et sert à faire la gestion entre tous les autres sous-composantes (menus, fonctions, classes, objets...). Cet objet est rempli de plusieurs fonctions et variables nécessaires pour le déroulement correct de la programme.

— test

## 6.3 Bibliothèques

### 6.3.1 PyGame

Pour permettre une visualisation graphique du simulation et l'optimisation, nous aurons besoin d'une bibliothèque Python qui permet l'utilisation des fonctions liées à l'affichage d'une interface graphique interactif. Pour satisfaire ces besoin, nous utiliserons la bibliothèque PyGame, qui donne l'accès aux fonctions pour manipuler des aspects graphiques de notre programme.

# 7 Architecture du projet

### 7.0.1 Fichiers du projet

Pour permettre une organisation plus nette et plus claire de notre projet, chaque partie d'une programme (simulateur, menu principale, etc..), et ses fonctions correspondants seront reparti, chacun dans son propre fichier. Exemple : Un fichier pour les fonctions du menu principale...

Notre fichiers principales sont les suivants :

- combat.py : Fonctions de combat utilisés lors du simulation.(permet de faire l'optimisation
- mainmenu.py : Fonctions pour le menu principale.
- modeles.py : Fonctions principales du programme, boucle de programme, etc. Elle contient aussi tous les classes et fonctions principales.
- optimisateur.py : Fonctions pour l'optimisation d'armée.(du menu interactif)
- unites.py : Classes des unités utilisés lors du simulation.
- wargame.py : Le ficher exécuté pour lancer le simulateur ; le point d'entrée du programme.

### 7.0.2 Aborescence du fichier

Pour permettre la gestion simplifié du projet, chaque dossier, sous-dossier et fichier de notre programme doit porter un nom qui est en relation avec son contenu. De plus, le nombre de dossiers et sous dossiers devront être limités, pour éviter de partir dans tous les directions pour faire la recherche d'un ficher spécifique.
Pour pouvoir respecter ces besoins, nous avons construit le dossier du projet du manière suivante :

- Le racine du dossier contiendra deux dossiers, appelés "src" et "docs" respectivement.
- Le dossier src contiendra tous nos fichiers Python (.py).
- Le dossier docs contiendra tous les fichier liées au rapport LaTex, pour pouvoir le générer.
- Un dossier caché appelé ".svn" est parfois visible dans la racine du dossier. Cet dossier sert au comme paramètrage pour le répositoire en ligne.
- Un fichier texte appelé "README.txt" est aussi disponible dans la racine du dossier.

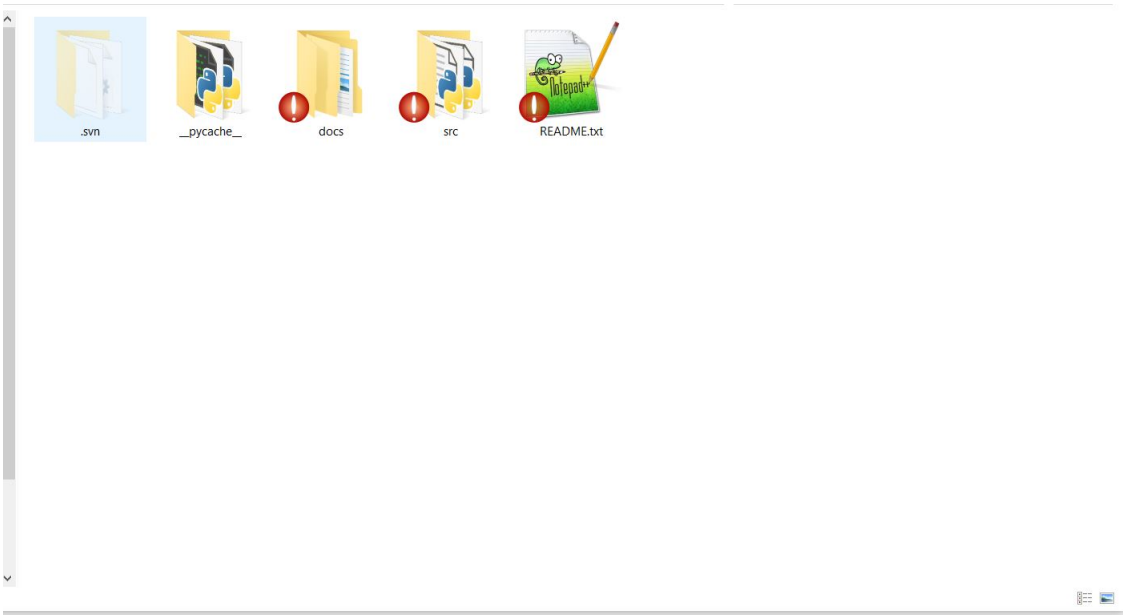


**Figure 11** – Une capture d'écran du racine du dossier.

# 8 Conclusion

Grâce aux algorithmes et fonctions implémentes dans notre projet, les objectifs principales de notre projet ont été atteints sans des soucis majeurs. Il y a certains éléments de la programme que nous avons souhaités réaliser mieux (exemple : les aspects graphiques), mais en cause du COVID-19 nous forcant à travailler à notre foyers, nous n'avions pas pu récuperer certains fichiers stockés sur les ordinateurs de l'université. A la suite de ceci, nous étions contraints de réaliser certains graphiques d'une manière moins précis. Le temps d'exécution de la boucle principale, et de la simulation nous contrains à devoir réduire le nombre d'armées. Nous en avons mis 26 armées, en raison que une des nombres les plus élevées, qui prend le moins de temps à avoir un résultat. Nous devons faire combattre 25*26 armées, soit 650 simulations de comba,t plus une moyenne de 19 unités par armée, soit plus de 12 350 interaction sans compter la partie optimisation qui prend moins de temps. L'optimisation se basant sur des statistiques, l'algorithme d'optimisation n'est pas sur a 100% d'avoir une optimisation réussis de l'armée. Nous considérons que, si une armée optimisé n'a pas passé le test de réussite, alors l'armée la plus puissante reste celle de base et sur l'échantillon d'armée, elle est la plus puissante.