

Compte rendu projet majeur

Soren LE MEITOUR, Mael MONTILLET

Octobre 2022

Table des matières

1 Concept et histoire du jeu	3
1.1 L'univers pokémon	3
1.2 La licence	3
1.3 Les créatures pokémon	3
2 Règles du jeu	4
2.1 Déroulé d'une partie	4
2.2 Calcul des dégât d'une capacité	4
3 Notre version du jeu	5
3.1 Les pokemons disponibles dans notre jeu	5
3.1.1 Boulbizarre	5
3.1.2 Carapuce	5
3.1.3 Etourmi	6
3.1.4 Pikachu	7
3.1.5 Osselait	7
3.1.6 Salamèche	8
3.2 Les statuts disponibles dans notre jeu	8
3.2.1 Brûlure	8
3.2.2 Poison	9
3.2.3 Vampigraine	9
3.2.4 Paralysie	9
3.2.5 Emprisonnement	9
3.2.6 Piège de rock	9
3.3 Tableau des types disponibles dans notre jeu	9
4 Architecture du code	11
4.1 Idée générale	11
4.2 Diagramme UML	11
4.3 Architecture du jeu	14
4.3.1 Jeu simultané	15
4.3.2 Jeu non déterministe	15

4.3.3	Le cas particulier de pokemon	16
4.3.4	Les fonctions suivants	17
5	Strategies	19
5.1	Presentation du problème	19
5.1.1	L'objectif	19
5.1.2	Difficultés liées au jeu pokemon	19
5.2	Evaluation de l'état	20
5.3	La classe joueur	21
5.4	Solution envisagées	21
5.4.1	Choix aléatoires	21
5.4.2	Règles simples	21
5.4.3	Minimax	25
5.4.4	Expectiminimax	30
5.5	Ordre des mouvements	32
5.6	Solutions Envisageables	33
5.6.1	Ordre des mouvements	34
5.6.2	Recherche de principale variation	34
6	interfaces	35
6.1	Introduction	35
6.2	Interface Textuelle	35
6.2.1	déroulé d'un tour	35
6.2.2	affichages en textuel :	36
6.3	Interface Graphique	37
6.3.1	déroulé d'un tour	38
6.3.2	affichages graphiques :	38
7	Manuel utilisateur	44
8	Sources	45

1 Concept et histoire du jeu

1.1 L'univers pokemon

Dans l'univers des Pokémons, les animaux du monde réel n'existent pas. Le monde est peuplé de Pokémons, des créatures qui vivent en harmonie avec les humains, mais possèdent des aptitudes quasiment impossibles pour des animaux du monde réel.

Certains dressent les Pokémons pour organiser des combats entre eux et nous vous proposons une émulation de ces combats.

1.2 La licence

Pokémon est une série de jeux vidéo créée par le Japonais Satoshi Tajiri et éditée par Nintendo, selon les statistiques de Nintendo en 2010, les jeux Pokémons se sont vendus à environ 250 millions d'unités

1.3 Les creatures pokemon

Les Pokémons sont des créatures intelligentes ayant un ou deux types et pouvant maîtriser des capacités. Le but de ces combats est simple : il faut mettre le Pokémon adversaire K.O. grâce aux capacités, le Dresseur guide son Pokémon en lui donnant des instructions d'attaque. Les Pokémons sont capturables par des Poké Balls qui permettent de transporter le Pokémon plus facilement s'il n'est pas utilisé. Un Dresseur ne peut porter avec lui que six Pokémons à la fois. Mais peut choisir ses sixs pokemons parmis les 924 pokemons crées à ce jours. Nous avons bien sur enlevé cette possibilité pour des raison pratiques. Dans notre version, les deux adversaires auront donc la même équipe composée de bulbizarre, carapuce, étourmi, pikachu, osselait et salamèche (pokemons qui appartiennent réellement à la licence). Nous avons simplifié le système de statistique des pokemons (qui présente normalement deux statistiques d'attaques et deux de defense). Chaque pokemon aura donc :

- Une statistique d'attaque.
- Une statistique de defense.
- Un nombre de PV
- Une vitesse qui détermine quel pokemon attaque en premier (outre la priorité des attaques et des changements).
- Quatre attaques qui infligent un nombre de dégât donné (et qui peuvent avoir des effets autre que les dégat comme ajouter des statuts, changer de pokemon...) et possèdent un type.
- Les points de pouvoir ou PP de ses attaques. C'est une liste d'entiers qui représente le nombre de fois que l'attaque i (i l'indice dans la liste) peut être utilisée. Chacun de ces entiers est décrémenté quand l'attaque correspondante est utilisée. Quand les PP sont à 0, l'attaque correspondante n'aura pas d'effet si elle est choisie. Si un pokemon n'a plus de PP

sur toutes ses attaques et qu'il reste sur le terrain, il perd 50 PV. Cette statistique est utile pour limiter la taille des parties et s'assurer que les agents ne rentrent pas dans une boucle.

2 Règles du jeu

Lors d'un affrontement, chaque dresseur envoie au combat un pokémon. À chaque tour, chaque pokémon utilise une capacité choisie ou est remplacé par un autre pokémon de l'équipe. Les choix des actions sont simultanés entre les deux joueurs et l'ordre des événements est régit par plusieurs règles : Les changements sont prioritaires donc effectués au début du tour. Les attaques prioritaires comme vive-attaque passent avant les autres quelque soit la vitesse de leur lanceur. Lorsque deux attaque ont la même priorité, c'est le pokémon le plus rapide qui attaque en premier.

Lorsqu'un pokémon est mis K.O., il ne peut pas effectuer d'attaque et un remplaçant est choisi par le dresseur concerné.

A la fin de chaque tour, les statuts des pokemons sur le terrain sont appliqués. Quand deux pokemons ont des statuts c'est le pokémon le plus lent qui subit les statuts en premier (c'est important car il peut y avoir la situation où les deux pokemons vont mourir des statuts et sont les derniers pokemons de l'équipe, le premier qui subit les statuts est donc celui qui perd la partie).

Un joueur a perdu quand tous ces pokemons sont K.O..*

2.1 Déroulé d'une partie

Une fois l'interface, les types et les pseudos des joueurs choisis, la partie débute. Tout d'abord, les deux joueurs choisissent leur pokémon de départ parmi les 6 pokemons disponibles dans notre jeu (se référer au point 3.1 pour plus d'information). Celui-ci est alors sur le terrain, c'est lui qui combattrra en premier. Comme expliqués ci-dessus, à chaque début de tour, les deux joueurs choisiront chacun leur tour entre attaquer le pokémon adverse et échanger le pokémon sur le terrain avec un de ses pokémon disponible, en pouvant jeter un œil aux informations de la partie pour prendre sa décision. Une fois que les deux joueurs ont effectué leur choix, ceux-ci s'appliqueront et le tour sera fini. Tant qu'un des deux joueurs n'a pas perdu tous ses pokemons, les tours se succéderont de cette façon.

2.2 Calcul des dégâts d'une capacité

Si on note a l'attaque du pokémon attaquant, d la défense du pokémon défenseur et c les dégâts de l'attaque, la formule pour calculer les dégâts (dans notre version simplifiée) est :

$$\frac{a}{d} * c$$

Avec un bonus de 10% si le type de l'attaque est le même que celui du pokémon qui la lance.

3 Notre version du jeu

3.1 Les pokemons disponibles dans notre jeu

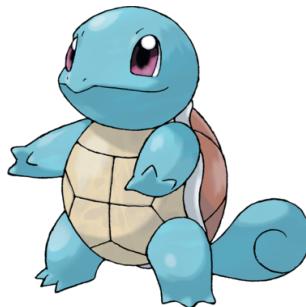
3.1.1 Boulbizarre



- Type : plante
- Attaque : 120
- Défense : 120
- PV : 230
- Vitesse : 110
- Listes des attaques : Fouet-Liane (de type plante, inflige 90 de dégât), Charge (de type normal, inflige 60 de dégât), Croissance (de type normal, inflige 0 dégâts et multiplie le bonus d'attaque 'a'[voir 2.1] par 1,25) et Vampigraine (de type plante, applique le statut vampigraine)

Remarque stratégique : L'attaque Vampigraine de bulbizarre est très utile sur le long terme grâce à son pourvoir de soin. Ce pokémon est un très bon tank avec son grand nombre de PV, capable d'encaisser les attaques tout en mettant des bons dégâts. Il est cependant un peu lent.

3.1.2 Carapuce



- Type : eau
- Attaque : 120
- Defense : 150
- PV : 230
- Vitesse : 110
- Listes des attaques : Exuviation (de type eau, inflige 0 dégâts et multiplie les bonus d'attaque et de defense 'a' et 'd' [voir 2.1] respectivement par 1,5 et 0,75), Ebullition (de type eau, inflige 80 de dégât et le statut brûlure avec une chance sur trois), Toxic (de type normal, inflige 0 dégâts et applique le statut poison) et Coud-Boule (de type normal, inflige 75 de dégât)

Remarque stratégique : Carapuce est un pokémon avec une grosse défense et qui a la possibilité d'appliquer plusieurs statuts. Il est donc très intéressant lorsqu'il s'agit de se placer, notamment grâce à son attaque éxuviation qui permet d'augmenter beaucoup son attaque, bien qu'en prenant le risque de baisser sa défense

3.1.3 Etourmi



- Type : vol
- Attaque : 130
- Defense : 90
- PV : 220
- Vitesse : 110
- Listes des attaques : Vive-attaque (de type normal, inflige 70 de dégât), Atterrissage (de type vol, inflige 0 dégâts et soigne étourmi à hauteur de la moitié de ses PV de base), Rapace (de type vol, inflige 100 de dégât) et Antibrume (de type vol, inflige 0 dégâts, retire le statut piège de rock si celui-ci est présent)

Remarque stratégique : Malgré son faible potentiel défensif, étourmi compense par une possibilité de soin et surtout par de gros dégâts et la possibilité de s'assurer d'attaquer en premier avec vive-attaque, très utile pour finir un pokémon avant qu'il soit changé par exemple.

3.1.4 Pikachu



- Type : electrik
- Attaque : 130
- Defense : 110
- PV : 210
- Vitesse : 200
- Listes des attaques : Tonerre (de type electrik, inflige 95 de dégât), Cage éclair (de type electrik, inflige 0 dégâts et applique le statut Paralysé si l'adversaire n'est pas de type electrik), Rugissement (de type normal, inflige 0 dégâts et multiplie le bonus d'attaque 'a'[voir 2.1] de l'adversaire par 0,75) et Change-éclair (de type electrik, inflige 70 de dégât et permet de changer de pokémon tout de suite après l'attaque)

Remarque stratégique : Pikachu est idéal pour faire mal rapidement. En effet, il a une bonne attaque et une vitesse qui lui assure, sauf cas particulier, de toujours attaquer en premier. Son attaque change-éclair permet, au lieu de simplement changer de pokémon, d'infliger des bons dégâts avant de le faire.

3.1.5 Osselait



- Type : sol
- Attaque : 120
- Defense : 200
- PV : 240
- Vitesse : 95

- Listes des attaques : Mimi-queue (de type normal, inflige 0 dégâts et multiplie le bonus de défense 'd' [voir 2.1] de l'adversaire par 0,75), Seisme (de type sol, inflige 100 de dégât), Piège de rock (de type sol, inflige 0 dégâts et applique le statut piège de rock) et Coud-Boule (de type normal, inflige 75 de dégât)

Remarque stratégique : Meilleur défenseur du jeu, Osselait peut résister très longtemps à ses adversaires. Son attaque seisme fait d'énorme dégâts et son attaque piège de rock permet de blesser les pokemons adverses progressivement, à chaque changement de pokemon adverses.

3.1.6 Salamèche



- Type : feu
- Attaque : 130
- Défense : 110
- PV : 220
- Vitesse : 150
- Listes des attaques : Crocs feu (de type feu, inflige 65 de dégât et le statut brûlure avec une chance sur trois), danse flammes (de type feu, inflige 35 de dégât et le statut emprisonne), tranche (de type normal, inflige 70 de dégât) et groz yeux (de type normal, inflige 0 dégâts et multiplie le bonus de défense 'd' [voir 2.1] de l'adversaire par 0,75 tout en multipliant son bonus d'attaque 'a' [voir 2.1] par 1,25)

Remarque stratégique : L'attaque danse flammes est utile pour bloquer le pokémon adverse sur le terrain et mettre face à lui un pokémon qui le contre afin de le mettre KO ou de se placer. La vitesse élevée du salamèche lui permet aussi de facilement finir des pokemons affaiblis.

3.2 Les statuts disponibles dans notre jeu

3.2.1 Brûlure

Statut qui inflige des dégâts à hauteur d'un huitième des PV totaux du pokémon à chaque tour pendant 6 tours et multiplie son bonus d'attaque 'a' [voir 2.1] par 2/3,

3.2.2 Poison

Statut qui inflige des dégâts à hauteur de 1/16, 2/16, 3/16 puis 4/16 des Pv totaux du pokémon, respectivement pendant le 1er, 2ème, 3ème et 4ème tour durant lesquels il s'applique.

3.2.3 Vampigraine

Le pokémon affecté voit sa vitesse être réduite de moitié et se fait 'vampiriser', Il perd des PV à hauteur de 1/8 de ses PV totaux et le pokémon qui a lancé vampigraine récupère ces PV. Ce statut s'applique pendant 5 tours.

3.2.4 Paralysie

Le pokémon affecté par ce statut a 1 chance sur 3 de ne pas pouvoir attaquer au tour suivant et voit sa vitesse être réduite de moitié. Ce statut s'applique pendant 10 tours

3.2.5 Emprisonnement

Le pokémon affecté ne peut pas être échangé avec un autre pokémon pendant 6 tours.

3.2.6 Piège de rock

Le dresseur étant coincer sous les pierres pendant 30 tours, le pokémon arrivant sur le terrain en cas de changement de pokémon de sa part reçoit directement des dégâts en fonction de son type :

- 18/100 de ses PV totaux si il est en faiblesse par rapport au type sol
- 12/100 de ses PV totaux si il n'est ni en faiblesse ni en résistance par rapport au type sol
- 6/100 de ses PV totaux si il est en résistance par rapport au type sol
- rien si il est immunisé au type sol

3.3 Tableau des types disponibles dans notre jeu

	ATTAQUE	VOL	SOL	FEU	PLANTE	EAU	NORMAL	ELECTRIK
DEFENSE								
VOL			1	0	1	0.5	1	1
SOL			1	1	1	2	2	1
FEU			1	2	0.5	0.5	2	1
PLANTE			2	0.5	2	1	0.5	1
EAU			1	1	0.5	2	0.5	1
NORMAL			1	1	1	1	1	1
ELECTRIK			0.5	2	1	1	1	0.5

Voici un tableau présentant les rapports entre les différents types :

- Se lit " le type 1 (colonne) est en faiblesse, résistance, immunité ou normal par rapport au type 2 (ligne) si le coefficient est respectivement 2, 0.5, 0 ou 1

4 Architecture du code

Dans cette partie nous allons voir comment nous avons structuré nos programmes, c'est un point important qui définit les performances, la robustesse et la facilité à s'adapter à de nouvelles situations de l'application produite.

4.1 Idée générale

Nous avons cherché le plus possible à utiliser les classes dès que c'était nécessaire, car la programmation orienté objet permet une meilleure lecture du code et plus de souplesse et de facilité dans son développement.

Pour la plupart des problèmes nous avons cherché à produire une **class abstraite** mère qui correspondait à l'idée générale de ce que nous voulions coder et plusieurs classe filles, plus spécifiques qui en héritait. Cette manière de procéder à l'avantage de rendre plus clair le code et de faciliter l'ajout de nouvelle fonctionnalité. Par exemple pour ajouter un pokémon il suffit de créer une nouvelle classe qui hérite de la classe abstraite Pokemon.

Ensuite, un autre avantage de l'orienté et de l'héritage des classes abstraites c'est de cloisonner les problèmes. En effet, quand par exemple un jeu discute avec une interface, il lui fait des requêtes (comme lui demander une action, ou appeler une animation) sans se soucier de si c'est une interface texte ou graphique.

Aussi, nous avons utilisé de nombreuses property, soit pour stocker une information d'une manière différente qu'elle est utilisée, soit pour être sûr que des attributs ne sont pas modifiées en dehors de leur classe. Par exemple, pour être sûr que le pokémon courant d'un dresseurs soit un des pokemons de son dictionnaire de pokémon, self._courant stocke le nom d'un pokémon puis un property fait en sorte que self.courant soit le dictionnaire indexé par le nom stocké dans self._courant.

Enfin l'orienté objet permet d'utiliser des solutions de code qui sont générales et réutilisables dans d'autres situation. Par exemple, les solution utilisant l'algorithme du minimax sont des solution générales appliquées à notre jeu.

4.2 Diagramme UML

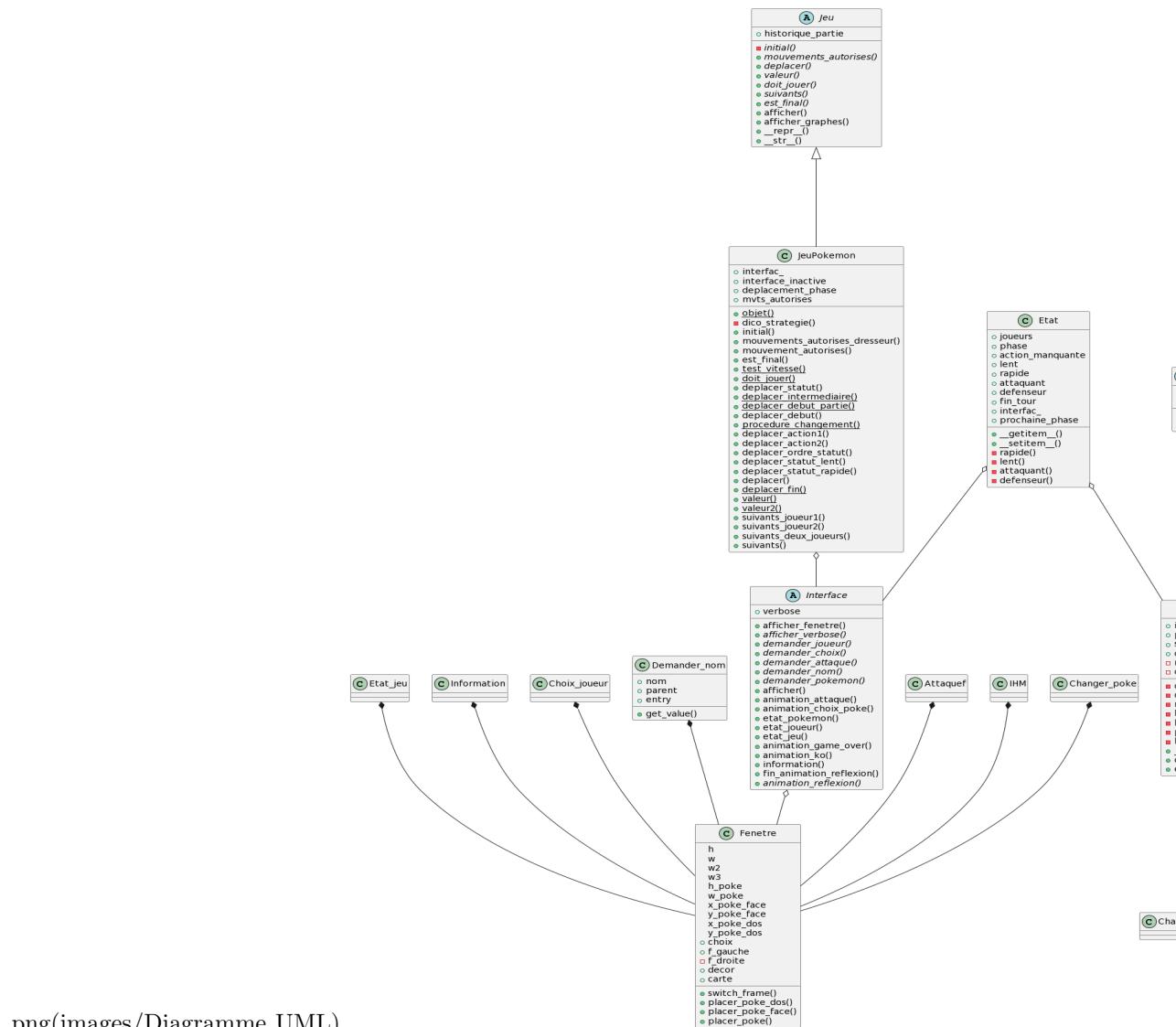
Voici le diagramme UML de notre jeu, réalisé avec PlantText UML. Etant donné du grand nombre de classes présentent dans notre jeu, nous avons décidé de limiter le diagramme UML aux classes principales en représentant leurs classes dérivées par une seule classe Exemple. Par exemple, notre jeu comporte une classe abstraite Attaque, qui se dérive en 23 sous-classes pour les 23 différentes attaques présentent dans notre jeu. Celles-ci sont représentées dans le diagramme par la classe Exemple_attaque.

Dans ce diagramme, les attributs et les méthodes sont représentés respectivement dans les parties hautes et basses des classes. Voici une légende pour mieux comprendre les signes qui peuvent apparaître devant ceux-ci.

- Une classe portant la mention 'C' à côté de son nom est une classe normale
- Une classe portant la mention 'A' à côté de son nom, qui est écrit en italique dans ce cas, est une classe abstraite.
- Un rond vert non plein représente un attribut publique d'instance.
- Un carré rouge non plein représente un attribut privé d'instance.
- L'absence de signe représente un attribut de classe.
- Un rond vert plein représente une méthode basique.
- Un carré rouge plein une méthode avec une property.
- Une méthode soulignée représente une méthode statique.
- Une méthode en italique représente une méthode abstraite.

Remarque : pour mieux voir le diagramme, veuillez regarder directement le

Diagramme UML



png(images/Diagramme_UML)

4.3 Architecture du jeu

Afin de pouvoir appliquer des solutions générales comme l'algorithme du minimax à notre jeu, il fallait que notre jeu soit une classe JeuPokemon qui hérite de la classe Jeu présentée dans le cours et qu'il soit lancé via joueur_jeu. Pour ce qui est de la classe mère, elle a été reprise tel quel dans notre code, cependant la fonction joueur_jeu était plus problématique.

En effet la fonction joueur_jeu qui nous a été présenté en cours s'appliquait à un jeu :

- à somme nulle (les intérêts des joueurs sont strictement opposés).
- déterministe (pas d'intervention de la chance).
- séquentiel (l'ordre des décisions est prédéfini à l'avance)
- à information complètes (on connaît ses possibilités d'action, les possibilités d'action des autres joueurs, les gains résultants de ces actions et les motivations des autres joueurs).

Cependant le jeu pokémon que nous vous proposons est à somme nulle, **non déterministe, simultané** (les joueurs décident en même temps de leur stratégie) et à information complètes. Nous avons donc modifié joueur_jeu pour qu'elle s'applique à tout les jeux qui ont ces dernières propriétés.

Aussi, nous avons transformé le while qui était dans la fonction en récursivité pour pouvoir animer le jeu avec la méthode after de tkinter mais cette modification ne change rien à ce que fait la fonction. Comme nous allons y faire référence dans les prochaines sections, voici la fonction joueur_jeu :

```
def joueur_jeu(jeu, strategies, nb_partie, noms, verbose = True):
    """fonction pour jouer au jeu donn en param tre.
    Param tres:
    strategie : dictionnaire qui      chaque joueur associe une instance de Joueur.
    nb_partie : int
    noms : liste de deux str qui stocke les noms des joueurs
    verbose : vrai si on souhaite afficher l'etat des tours      la fin de ceux-ci.
    """
    sys.setrecursionlimit(7000)
    etat = jeu.initial(noms)
    tour(jeu, etat, strategies, nb_partie, noms, verbose)
    """Pas de while mais de la r cursivit
    car tkiter est en multithreading et il faut passer par racine.after
    pour ne pas cr er de prob mes."""

def tour(jeu, etat, strategies, nb_partie, noms, verbose):
    """Effectue un tour du jeu et app le le tour suivant pas r cursivit ."""
    deplacement = {}#dictionnaire qui va stocker les action des joueurs qui jouent.
    #liste des joueurs qui doivent jouer.
    liste_joueur = jeu.doit_jouer(etat)
    #On r cup re la strategie de tout les joueurs.
    for joueur in liste_joueur:
        deplacement[joueur] = strategies[joueur](jeu, etat)
    etats = jeu.deplacer(etat, deplacement)
    etat = choix(etats)#On choisit un etat parmi les etats possibles.
    if verbose:
        #mise jour les infos sur le tour.
        etat.interface.etat_jeu(etat)
        etat.interface.afficher(f'\nvaleur du jeu estim e : {jeu.valeur(etat)}')
        #affichage des infos via l'interface.
        etat.interface.afficher_verbose()
```

```

jeu.historique_partie[-1].append(etat)
if not jeu.est_final(etat):
    etat.interface.rappel(lambda : tour(jeu, etat, strategies, nb_partie, \
                                         noms, verbose))
else:
    nb_partie -= 1
    if nb_partie > 0:
        etat = jeu.initial(noms)
        etat.interface.rappel(lambda : tour(jeu, etat, strategies, nb_partie, \
                                         noms, verbose))
    else:
        #affichage des graphes (1 graphe par partie pour la valeur et un pour
        #toutes les partie pour le temps)
        jeu.afficher_graphes(jeu.valeur, strategies['joueur1'], strategies['joueur2'])
        strategies['joueur1'].afficher_historique()
        strategies['joueur2'].afficher_historique()

```

4.3.1 Jeu simultané

Un jeu simultané est un jeu dans lequel l'ordre de prise de décision n'est pas prédéfini. Selon la situation, un joueur doit prendre une décision, l'autre ou les deux en même temps. Une fois les décisions prises, un système de priorité des actions permet de définir quelle action est exécutée en premier ou si même l'action est exécutée. Ainsi, la méthode `doit_jouer` d'un tel jeu ne renvoie pas le joueur qui joue dans un état mais la liste des joueurs qui doivent jouer.

Ensuite `joueur_jeu` doit demander la stratégie adoptée par chaque joueur qui doit jouer et la stocker dans `deplacements` qui est donc un dictionnaire qui prend comme clés les joueurs et comme valeur les actions des joueurs. `Deplacement` est ensuite fourni à `deplacer` pour déplacer l'état.

4.3.2 Jeu non déterministe

Pour pouvoir prendre en compte le facteur chance, `deplacer` doit renvoyer non pas un état mais l'ensemble des couples des états possibles avec leur probabilité d'apparition.

Ensuite la fonction `choix` est utilisée pour choisir un état parmi ceux qui sont possibles.

```

def choix(liste_etats):
    """Permet de choisir un état parmi les états possibles en respectant
    leur probabilité."""
    proba_totale = 0
    liste_proba = []
    dico_etaut = {}
    for proba, etat in liste_etats:
        proba_totale += proba
        liste_proba.append(proba_totale)
        dico_etaut[proba_totale] = etat
    choix = random()
    i = 0
    """on découpe [0,1] en segments de taille correspondant à la probabilité
    de chaque état. Un état est choisi si random() tombe dans son segment."""
    while i < len(liste_proba) - 1 and liste_proba[i] <= choix:
        i += 1
    return dico_etaut[liste_proba[i]]

```

4.3.3 Le cas particulier de pokemon

Pour que la classe JeuPokemon puisse respecter ce qui a été dit ci-dessus, cela a été un vrai défit de programmation.

Tout d'abord un premier challenge a été de faire en sorte que la fonction deplacer s'arrête à chaque fois qu'il faut demander un choix a un des joueurs, et reprendre ensuite là où elle en était. Dans pokemon, un choix est demandé à chaque début de tours aux deux joueurs, puis à chaque fois qu'un pokemon est mit ko au dresseur à qui appartient le pokemon. Ainsi, nous avons découpé le jeu en plusieurs phases, stockées dans l'état, qui se suivent et représentent des blocs entiers sans choix posés aux joueurs. A chaque phase correspond une fonction deplacement.

De ce fait, la fonction déplacer correspond à une boucle while dans laquelle on effectue la fonction de deplacement de la phase tant que l'état ne nous indique pas que le tour est finit.

Les blocs sont les suivants :

- "debut de partie" : en tout début de la partie, quand les dresseurs n'ont pas de pokemon courant et qu'on ne peut pas faire de test de vitesse.
- "debut" : en début de tour, permet de définir qui attaque en premier.
- "action1" : on effectue l'action du joueur qui joue en premier.
- "action2" : on effectue l'action du joueur qui joue en second
- "ordre statut" : Si les deux joueurs ont des statuts, on définit l'ordre dans lequel ils les subiront. Si il n'y en a qu'un on les effectue directement (car pas besoin de différencier le lent et le rapide).
- "statut lent" : On effectue les effets de statuts du pokemon le plus lent.
- "statut rapide" : On effectue les statut du joueur le plus rapide.
- "fin" : on vérifie si on doit enlever des statuts des pokemons.

Solution au problème des interruption dans le tour : A chaque fin de bloc, la phase de l'état est mise soit à la prochaine phase soit à une phase intermédiaire quand un pokemon a été mit KO. L'ordre du déroulé n'est bien sûr pas linéaire et ne suit pas la liste précédente. Par exemple si aucun des joueurs n'a de statut, on passe directement de "ordre statut" à "debut".

Quand la phase est mise à intermédiaire, on stocke dans l'état la prochaine phase et on signale au while que c'est la fin du tour. Ensuite le pokemon de remplacement est demandé au joueur qui a un pokemon ko et la partie reprend par le deplacement de la phase intermédiaire qui effectue le changement demandé par le joueur. Puis la phase prend pour valeur la phase qui avait été stockée dans l'état comme étant la prochaine phase et le déroulé de la partie reprend.

Bien sûr, la phase qui est stockée avant une phase intermédiaire comme étant la prochaine phase depend des événements dans la partie. Par exemple lors du deplacement de "action1", si c'est le joueur qui a fait l'action qui est mit KO en la faisant (par exemple si un pokemon qui a peu de PV est placé en

remplacement sur des pièges de rock, il peut être mit KO), alors la prochaine phase sera "action2". Cependant si c'est le joueur qui attaque en second qui est mit KO pendant "action2", alors, la prochaine phase sera "ordre statut" car un pokémon qui est mit KO ne peut pas attaquer dans le tour.

Enfin pour que la partie puisse reprendre correctement après une interruption, on est parfois amené à stocker dans l'état des informations comme l'attaque du second attaquant ou le pokémon qui doit subir les statuts lent... Ce stockage est fait dans l'état qui est donc une instance de la classe Etat. La classe Etat possède de nombreux attributs dont ces informations intermédiaires, la phase, les dresseurs... (voir abstract_jeu.py)

Deplacement de tout les Etats possibles : Le déroulé présenté dans le paragraphe précédent correspond à la fonction deplacer telle qu'elle était avant qu'on ne prenne en compte l'aléatoire.

Ainsi deux changements majeurs ont été faits pour prendre en compte les branches liées à la chance :

- Toutes les actions renvoient des couples d'état possibles avec la probabilité associée, ainsi, toutes les fonctions de déplacement renvoient elles aussi le même type de couple.

La fonction deplacer travaille avec des listes qui stockent tous les états possibles qui ne sont pas encore finis, et à chaque tour du while, elle parcourt la liste pour déplacer la phase suivante de tous ces états. À chaque fois qu'un état est déplacé, ce qui est récupéré est une liste d'états possibles liés au déplacement de cet état particulier, la probabilité de tous les états de cette liste est donc multipliée par la probabilité de l'état dont ils sont issus.

À chaque fois qu'un état est fini, il est stocké avec sa probabilité dans la liste finale, sinon il est stocké avec sa probabilité dans une liste des états qui restent à déplacer. La boucle while s'arrête quand il ne reste plus d'états possibles qui ne sont pas finis.

4.3.4 Les fonctions suivants

Pour les jeux simultanés il y a une petite subtilité par rapport aux états suivants, c'est que si l'autre joueur doit jouer dans l'état suivant, il faut préciser quelle action il fait pour réellement, c'est ce qui a été fait dans notre jeu. Ainsi, il y a deux fonctions suivantes qui ont principalement été utilisées : suivants_joueur1 et suivants_joueur2. On leur fournit deux informations : l'état et si l'autre joueur joue avec dans ce cas-là, son action. La fonction renvoie l'ensemble des triplets action du joueur 1, action du joueur 2, liste des états possibles avec leur probabilité, sachant que les actions de l'autre joueur seront donc soit toutes les mêmes si il joue soit None si il ne joue pas.

Voici le code :

```
def suivants_joueur1(self, etat, mvt_j2 = None, doit_jouer_j2 = False, trier = False):
    """Renvoie la liste des états suivants les actions disponibles du joueur1 sachant si le
```

```
joueur2 joue et quel action il fait dans ce cas."""
liste_suivant = []
deplacement={}
for mvt_j in self.mouvements_autorises_dresseur(etat, etat[ 'joueur1' ], trier):
    deplacement[ 'joueur1' ] = mvt_j
    if doit_jouer_j2:
        deplacement[ 'joueur2' ] = mvt_j2
    liste_suivant.append((mvt_j, mvt_j2, self.deplacer(etat, deplacement)))
return liste_suivant
```

5 Strategies

5.1 Presentation du problème

5.1.1 L'objectif

Dans cette partie, nous allons analyser les stratégies envisageables pour jouer à Pokemon. Notre objectif sera de créer une IA capable de jouer à pokemon de la manière la plus performante possible sans que le temps de calcul ne soit trop grand.

Pour cela nous envisagerons d'abord des solutions simples qui s'appuient sur l'aléatoire et sur des règles simples, puis nous travaillerons sur des IA plus performantes qui reposent sur les algorithmes généraux présentés en cours et enfin nous essayerons d'améliorer ces algorithmes pour qu'ils s'adaptent mieux à notre jeu.

5.1.2 Difficultés liées au jeu pokemon

Dans le papier de recherche **Learning complex games through self play - Pokémons battles** de *Miquel Llobet Sanchez*, l'auteur cherche à créer une IA qui joue à pokemon via des algorithmes de parcours de graphe utilisant du deeplearning (solution qui est généralement celle utilisée pour le jeu pokemon).

Ce jeu y est présenté comme un jeu qui représente un challenge pour les intelligences artificielles (même modernes) pour les raisons suivantes :

L'atomicité des tours : A chaque tour, les deux joueurs soumettent une action, et la résolution des actions se fait en même temps via un système de priorité des actions (voir règles du jeu). Il en résulte que le jeu ne peut pas réellement être représenté comme un arbre et que le plus souvent, les joueurs exécutent leurs actions dans des états différents de celui dans lequel l'action à été choisie.

Le coût de simulation : Il faut 5 à 40 ms aux moteurs de simulation de bataille pokemons opens sources pour simuler un tour. Sachant que notre version sera vraisemblablement moins optimisée, le coût de simulation sera un facteur très limitant dans le nombre d'états qui pourront être explorés par nos agents.

Stochasticité : Le jeu pokemon n'est pas un jeu déterministe, la chance influence fortement l'issue des tours. Elle intervient dans le système de priorité : deux pokemons qui ont la même vitesse auront une chance sur deux d'attaquer en premier si les attaques ont la même priorité (sachant que dans notre version, seule vive-attaque a une priorité différente). Certaines attaques incorporent des facteurs chances comme Ebullition qui possède une chance sur trois de brûler l'adversaire, et les statuts aussi (exemple : un pokemon paralysé a une chance

sur trois de ne pas attaquer). Un agent qui saura prendre en compte le facteur chance aura donc un avantage certain sur les autres agents.

Le facteur de branchement : Sans prendre en compte la Stochasticité, le facteur de branchement est de 4 à 9 (4 attaques et 5 changements potentiels). Ce n'est pas comparable au facteur de branchement des échecs qui est en général autour de 35-38 mais cela reste non négligeable. D'autant plus que le facteur chance peut multiplier le nombre de branches par un coefficient compris entre 2 et 4 (ou plus dans de rares cas).

Difficulté d'appréciation des états Certaines attaques comme piège de rock (pose des pièges qui inflige des dégâts à chaque changement de pokémon de l'adversaire) ont un effet sur le long terme qui sera difficile à apprécier pour les agents car ceux-ci seront forcément limités en terme de profondeur. De plus, les pokemons peuvent avoir un certain nombre de statut en même temps et de bonus qui vont être difficiles à apprécier.

5.2 Evaluation de l'état

Une fonction d'évaluation de l'état de bonne qualité est nécessaire pour créer des agents performants pour deux raisons principales. D'abord, c'est un outil très intéressant pour évaluer les performances d'un agent. On pourra tracer le graphe de la fonction d'évaluation en fonction du nombre de tour lors d'un combat entre deux agents pour évaluer leurs performances tout au long de la partie. Ensuite, la fonction d'évaluation est nécessaire à certains agents (ceux qui utilisent le minimax) pour avoir une approximation de l'utilité (les parties faisant en général plus de 30 tours, il est irréalisable de simuler la partie jusqu'à la fin pour ces agents).

La fonction d'évaluation doit être rapide à calculer et représenter le mieux possible l'état de la partie. Nous avons donc choisi de prendre la différence des PV de tout les pokémon du joueur 1 et de tout les pokémon du joueur 2. Le joueur max sera donc toujours le joueur 1 et le min le joueur 2 L'avantage c'est que cette fonction est rapide de calcul et représente bien ce qui est la clef de la victoire, les PV des pokemons. Le default c'est quelle ne prend pas en compte les types des pokémon , leurs bonus et statuts, et leurs statistiques. Cependant tout ces éléments rajoutent du temps de calcul et demandent une connaissance du jeu que nous n'avons pas. Nous avons donc décidé de ne pas les prendre en compte. La prochaine étape pour améliorer nos agent serait d'entrainer un réseaux de neurone à évaluer chaque position. Qui sait c'est peut-être l'idée d'un futur projet, quand nous aurons les outils mathématiques nécessaires...

Implementation : (source : jeu.py) :

```
def valeur(etat : Etat) -> int:  
    PV_total_j1 = sum([poke.PV for poke in etat[ 'joueur1 '].liste_non_ko])  
    PV_total_j2 = sum([poke.PV for poke in etat[ 'joueur2 '].liste_non_ko])  
    return PV_total_j1 - PV_total_j2
```

5.3 La classe joueur

Voici la classe joueur. Tout les agents sont des classes qui en héritent ce qui leur permet de connaitre quel dresseur de l'état ils representent (self.id) et de stocker les temps de réponse.

```
class Joueur(metaclass = abc.ABCMeta):
    """D finit un joueur"""
    def __init__(self, joueur, nom):
        self.id = joueur #identifiant
        self.historique_temps = []#historique des temps de r ponse
        self.nom = nom

    def __call__(self, jeu, etat):
        t = time()
        mvt = self.strategie(jeu, etat)#strat gie sp cifique un joueur.
        self.historique_temps.append(time() - t)#mesure du temps de r ponse
        return mvt

    @abc.abstractmethod
    def strategie(self, jeu, etat):
        pass

    def afficher_historique(self):
        x = range(len(self.historique_temps))
        y = self.historique_temps
        plt.plot(x, y, 'r')
        plt.axline((0,0), (10**-7,0), color="black", linewidth=1)
        plt.axline((0,0), (0,10**-7), color="black", linewidth=1)
        plt.xlabel('numero_du_choix')
        plt.ylabel('temps_de_reflexion_(en_s)')
        plt.title(f'Temps_de_r ponse_de_{self.nom}')
        plt.show()

    def __str__(self):
        return self.nom
```

5.4 Solution envisagées

5.4.1 Choix aléatoires

Une première solution serait de choisir aléatoirement une action parmi les mouvements autorisés. C'est une solution simple pratique pour tester le code avant d'envisager des solutions plus compliquées. Ce n'est bien sur pas une solution viable étant donné que pokemon est un jeu dans lequel seulement une petite partie des actions sont interressantes dans un etat donné. La probabilité de tomber aléatoirement sur une action viable est donc trop faible.

5.4.2 Règles simples

On donne à un agent des règles simples à suivre :

- Le premier pokemon est choisi au hasard
- Si le pokemon courant de l'agent n'a plus de pp sur ses attaques, il change de pokemon si possible.
- Si il y a un désavantage de type en faveur de l'adversaire, l'agent change de pokemon si possible.

- Quand il change de pokémon il essaye en premier de choisir un pokémon qui a un avantage de type par rapport à l'adversaire. Si il n'y en a pas, il choisit un pokémon qui n'a pas de désavantage de type. Si il n'y en a pas, elle essaye d'attaquer et si ce n'est pas possible elle change de pokémon au hasard.
- Quand il attaque, l'agent fait toujours l'attaque qui fait le plus de dégât parmi les attaques qui ont des PP.

Ce sont des règles simples qui prennent comme priorité les PVs et les types des pokemons.

Implementation : (source : joueurs.py) :

```

def avantage_type(avantage : Pokemon, desavantage : Pokemon):
    """ Vrai si le pokémon avantage un avantage de type sur
    le pokémon desavantage
    Remarque : deux pokémon peuvent se résister l'un
    l'autre l'autre en même temps et il n'y a alors pas
    d'avantage """
    return (desavantage.type.nom in avantage.type.resistance and not \
            avantage.type.nom in desavantage.type.resistance) or \
            (desavantage.type.nom in avantage.type.imunité) or \
            avantage.type.nom in desavantage.type.faiblesse


class JoueurBasique(Joueur):
    """Joueur qui suit un ensemble de règles prédéfinies."""
    def __init__(self, joueur):
        super().__init__(joueur, 'Basique')

    def changer(self, dresseur, adversaire, attaque_possible = True):
        """ Sélection de l'agent si l'agent décide de changer de
        pokémon en priorité
        Précondition : dresseur peut changer de pokémon c'est dire
        dresseur.pokemons_dispo != [] """
        # choix en priorité d'un pokémon qui a un avantage de type
        liste_changement = [poke for poke in dresseur.pokemons_dispo \
                            if avantage_type(poke, adversaire.courant)]
        if liste_changement != []:
            return ChangerPoke(choice(liste_changement))
        else:
            # Sinon un pokémon qui n'a pas de désavantage de type
            liste_changement = [poke for poke in dresseur.pokemons_dispo \
                                if not avantage_type(adversaire.courant, poke)]
            if liste_changement != []:
                return ChangerPoke(choice(liste_changement))
            else:
                # A défaut de changement convenable, on attaque.
                if self.attaque_possible:
                    return self.atttaquer(dresseur, adversaire)
                """ si on a plus de PP ou si on doit changer malgré
                qu'on ne peut faire que des changements désavantageux
                on fait un changement au hazard """
                return ChangerPoke(choice(dresseur.pokemons_dispo))

    def attaquer(self, dresseur, adversaire):
        """ Sélection de l'agent si l'agent décide d'attaquer
        en priorité """
        attaques = [attaque for attaque in dresseur.courant.tuple_attaque \
                    if dresseur.courant.liste_pp[attaque.index] > 0]
        if attaques != []:
            # On effectue en priorité l'attaque qui fait le plus de dégât parmi
            # celles qui ont des PP.
            attaques.sort(key = lambda attaque : attaque.degat, reverse = True)

```

```

        return attaques[0]
    else:
        #si il n'y a plus de PP dans toutes les attaques, l'ordre importe peu.
        return dresseur.courant.attaque_1

def strategie(self, jeu, etat):
    #recuperation des r les des joueurs.
    dresseur = etat[self.id]
    if self.id == 'joueur1':
        adversaire = etat['joueur2']
    else:
        adversaire = etat['joueur1']
    if etat.phase == 'debut_partie':
        #choix de pokemon aleatoire en d but de partie
        return choice(list(jeu.mouvements.autorisés_dresseur(etat, dresseur)))
    #Verification du nombre de PP dans les attaques
    self.attaque_possible = not dresseur.courant.verifier_pp()
    if dresseur.doit_changer:
        self.attaque_possible = False
        return self.changer(dresseur, adversaire)
    if (not self.attaque_possible) and dresseur.pokemons_dispo != []:
        #si il n'y a plus de PP on change de pokemon si possible
        return self.changer(dresseur, adversaire)
    if "emprisonne" not in dresseur.statut and dresseur.pokemons_dispo != [] \
        and avantage.type(adversaire.courant, dresseur.courant):
        #si il y a un desavantage de type, on change de pokemon
        return self.changer(dresseur, adversaire)
    return self.atttaquer(dresseur, adversaire)

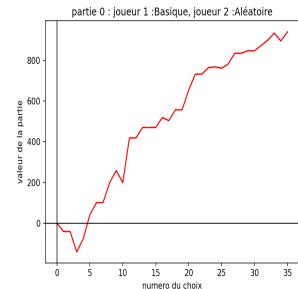
```

Résultat : L'IA obtenue bat largement l'IA qui joue aléatoirement mais ne représente pas un grand défaut pour un humain qui connaît un peu le jeu. Elle bat largement l'IA aleatoire avec un temps de réponse de l'ordre de 10^{-9} avec de rares pics à 10^{-3} (tout les ordres grandeurs de temps sont pris avec le même ordinateur dans le même environnement de travail pour pouvoir les comparer). Exemple de partie jouées contre l'aléatoire :

Sur 30 parties l'agent en gagne 30 et voici les valeurs des 2 premières parties :

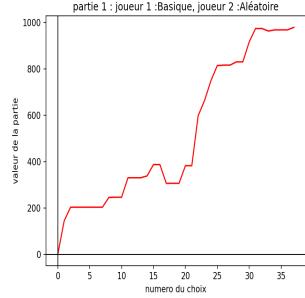
Première partie :

Evolution de la valeur (l'agent qui utilise les règles est l'agent nommé basique en joueur 1 donc en max) : Remarque : On peut voir le gagnant sur le graphique en regardant le signe du dernier point).

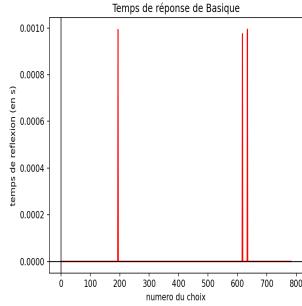


Seconde partie :

Evolution de la valeur (l'agent qui utilise les règles est l'agent nommé basique en joueur 1 donc en max) :



Le temps de réponse ne semble pas dépendre de la phase de la partie. Voici le temps de réponse au cours des 30 parties mises bout à bout :



Avantages : Le temps de réponse est très court. L'ia est très forte contre des agents de bas niveau car elle cherche le plus possible à faire des dégâts.

Défauts : Le jeu pokémon est trop compliqué pour qu'un jeu de règles soit suffisant pour y être performant. Ces règles ne permettent pas de prendre en compte les spécificités de l'état du jeu à un moment donné de la partie. Une règle qui est valable dans certaines situations peut être une grande erreur dans d'autres. Par exemple, changer de pokémon à chaque fois qu'il y a un désavantage de type n'est pas une bonne idée. Si le pokémon de l'adversaire a très peu de PV et qu'on est plus rapide que lui, il vaut mieux attaquer pour le mettre KO car ainsi il ne pourra pas riposter (étant plus lent, l'adversaire attaque en second et ne pourra pas attaquer si il est mit KO). L'agent résultant a donc un comportement très prévisible qui peut être anticipé par un adversaire humain et il a une très mauvaise adaptation aux situations.

L'agent cherche en priorité à mettre des dégâts mais ne profite pas des avantages que représentent les bonus et les statuts.

Il faut avoir une bonne connaissance du jeu pour trouver des règles convenables. Donner des règles sur l'utilisation des statuts et des bonus est par exemple diffi-

cile commes ces éléments sont très situationnels. Le choix du premier pokémon est aussi difficile.

L'ia obtenue n'est pas généralisable et ne peut pas être réemployée dans des jeux similaires.

5.4.3 Minimax

L'algorithme du minimax est un algorithme pour les jeux à deux joueurs en tour par tour à choix alternés. Dans cet algorithme on considère que un joueur le max cherche à maximiser la valeur du jeu et un autre cherche à la minimiser. Dans cet algorithme, l'ordinateur visite l'arbre du jeu sur une certaine profondeur puis remonte les valeurs associées à chaque coup de la manière suivante :

- Si on est à la profondeur p ou si le jeu est fini, on renvoie une approximation de la valeur du jeu (celle de la fonction d'évaluation).
- Si c'est au min de jouer, il choisit l'action qui minimise la valeur du jeu.
- Si c'est au max de jouer, il choisit l'action qui maximise la valeur du jeu.

Une amélioration de cet algorithme est l'algorithme alpha beta qui stocke en mémoire deux entiers, alpha et beta, qui stockent respectivement la plus grande valeur trouvée par le max et la plus petite trouvée par le min et permet d'élaguer l'arbre c'est à dire de ne pas visiter les branches qu'il est inutile de visiter (celle qui quoi qu'il arrive ne seront pas choisies par l'adversaire.)

Notre objectif dans cette section va être de discuter d'une adaptation de l'algorithme alpha beta au jeu pokémon.

Problème de l'ordre de jeu et de l'atomicité des tours : L'ordre de jeu dans pokémon n'est pas alternatif car un choix peut être demandé aux deux joueurs ou à un seul (en cas de KO) sans ordre prédéfini. Il faut donc passer par une fonction intermédiaire qui regarde qui doit jouer et :

- Si un seul joueur doit jouer, on applique l'algorithme du minimax habituel.
- Si deux joueurs doivent jouer, le choix de l'agent sera pour celui qui est *le plus optimal quel que soit le choix de l'adversaire*.

Pour déterminer quel est le choix *le plus optimal quel que soit le choix de l'adversaire*, on parcourt les mouvements de disponibles pour l'agent et on applique l'algorithme du minimax habituel en considérant cette action comme choisie. Pour chaque mouvement l'adversaire va donc choisir l'action qui est la meilleure sachant que notre agent à choisi ce mouvement, il va choisir le meilleur contre à notre action. On va récupérer une valeur d'état associée à chaque mouvement et son meilleur contre (comme les 2 joueurs jouent, il faut deux actions pour déplacer l'état) et en déduire la meilleure action.

Par exemple pour le max : On parcourt l'ensemble des actions disponibles pour le max. Pour chaque action, le min choisit l'action qui minimise la valeur de l'état sachant l'action qui est jouée par le max. On récupère toutes les valeurs associées aux couples mouvements du max, meilleur contre du min et on choisit

le couple qui a la valeur la plus grande. Ainsi on a choisi l'action qui maximise la valeur de l'état même si le min choisit le meilleur contre de cette action.

L'algorithme sera développé de la même manière que le minimax habituel mais avec trois fonction qui s'entre-appellent au lieu de deux : il y aura la *fonction interface*, la fonction *min* et la fonction *max*.

Problème de la Stochasticité Comme nous l'avons vu dans la section 5.1.2, le jeu pokémon n'est pas un jeu déterministe. Cependant, dans un premier temps, nous considéreront que le facteur chance est négligeable. Cette agent traitera donc l'aléatoire de la manière suivante : lorsqu'un état peut mener à plusieurs états différents via un facteur chance, un de ces états sera choisi en respectant les probabilités qui sont associées à chaque état. L'état choisi sera considéré comme le seul état possible, ce qui mènera forcement à des erreurs dans l'évaluation de la valeur des futurs états.

Implementation : La classe *JoueurMinimax* (tout les agents qui utilisent le minimax en héritent)(source : joueurs.py) :

```
class JoueurMinMax(Joueur):
    """Definit un joueur qui applique une strat gie utilisant le minimax.
    Voir minimax pour plus de pr cision sur les param tres."""
    def appliquer_minimax(self, jeu, etat, profondeur, fonction_valeur,\ 
        traitement_alea, condition_tri):
        dresseur = etat[self.id]
        etat.interface.animation_reflexion(dresseur)
        if self.id == 'joueur1':
            mouvement = minimax_joueur1(jeu, etat, profondeur, fonction_valeur,\ 
                traitement_alea, condition_tri)[0]
        else:
            mouvement = minimax_joueur2(jeu, etat, profondeur, fonction_valeur,\ 
                traitement_alea, condition_tri)[0]
        etat.interface.fin_animation_reflexion()
    return mouvement
```

L'agent (source : joueurs.py) :

```
class JoueurAlphaBeta(JoueurMinMax):
    """Joueur qui applique l'algorithme du minimax avec elagage alpha beta."""
    def __init__(self, joueur):
        super().__init__(joueur, 'Minimax')

    def strategie(self, jeu, etat):
        mvt = super().appliquer_minimax(jeu, etat, 2, jeu.valeur, choix_aleatoire, lambda p : False)
        return mvt
```

L'algorithme minimax (source : minimax.py) : La fonction interface (on ne met que celle du joueur 1, celle du joueur 2 étant similaire mais inversée) :

```
def minimax_joueur1(jeu, etat, profondeur, fonction_valeur, traitement_alea, \
    condition_tri, alpha = -10**5, beta = 10**5):
    """Fonction qui effectue l'interface entre le fonction minvaleur et
    maxvaleur afin d'effectuer l'algorithme du minimax pour le joueur 1
    (le joueur max).
    param :
        jeu : instance de Jeu.
        etat : instance d'Etat
```

```

profondeur : nombre de choix sur lequel on fait le minimax
fonction_valeur : callable qui prend en paramètre un état et
renvoie une estimation de son utilité (int)
traitement_alea : fonction de traitement de l'aleatoire, un
ensemble d'état possibles, elle associe une valeur entre 0 et 1.
condition_tri : condition pour trier les mouvements via une
recherche superficielle, c'est un callable qui prend en entrée
un entier et renvoie un booléen.
alpha, beta : entiers qui stockent respectivement la plus grande
valeur trouvée par le max et la plus petite trouvée par le min,
sert à l'algo.
Renvoie :
Si le joueur 1 doit jouer, le mouvement qui maximise la valeur et \
la valeur associée.
Si le joueur 1 ne doit pas jouer, le mouvement du joueur 2 qui minimise
la valeur et la valeur associée.
"""
if profondeur == 0 or jeu.est_final(etat):
    return None, fonction_valeur(etat)
profondeur -= 1
liste_joueur = jeu.doit_jouer(etat)
if len(liste_joueur) == 2:
    v = -10**5
    mvt = None
    """On choisit l'action qui est la meilleure même si l'adversaire
    choisissait le meilleur contre"""
    for mvt_j1 in jeu.mouvements_authorized(etat, etat['joueur1'], \
                                                condition_tri(profondeur)):
        """régulation du meilleur contre : c'est l'action qui minimise l'utilité
        sachant que le joueur 1 joue mvt_j1."""
        new_v = minvaleur(jeu, etat, profondeur, minimax_joueur1, fonction_valeur, \
                           traitement_alea, condition_tri, alpha, beta, mvt_j1, True)[1]
        #On garde la valeur maximale de toute les valeurs issue du meilleur contre.
        if new_v > v:
            v = new_v
            mvt = mvt_j1
        if v >= beta:
            return mvt, v
    alpha = max(v, alpha)
return mvt, v
#dans le cas où un seul joueur doit jouer c'est un minimax normal
elif liste_joueur[0] == 'joueur1':
    return maxvaleur(jeu, etat, profondeur, minimax_joueur1, fonction_valeur, \
                     traitement_alea, condition_tri, alpha, beta)
else:
    return minvaleur(jeu, etat, profondeur, minimax_joueur1, fonction_valeur, \
                     traitement_alea, condition_tri, alpha, beta)

```

La fonction max (source : minimax.py) :

```

def maxvaleur(jeu, etat, profondeur, fonction_interface, fonction_valeur,\ 
              traitement_alea, condition_tri, alpha, beta, mvt_min = None,\ 
              doit_jouer_min = False):
    """ Trouve le mouvement qui maximise la valeur et la valeur associée
    sachant quel mouvement est choisi par le joueur min si celui-ci doit
    aussi jouer.
    Précondition : Le joueur 1 doit jouer dans le tour.
    Si le joueur 2 doit aussi, le mouvement qu'il joue est
    donné par mvt_min.
    param :
        jeu, etat, profondeur, fonction_valeur, traitement_alea,
        condition_tri, alpha et beta : voir minimax_joueur1
        fonction_interface : fonction qui fait l'interface entre
        minvaleur et maxvaleur.
        mvt_min : mouvement choisi par le min
        doit_jouer_min : vrai si le min doit aussi jouer dans
        le tour.
    """

```

```

v = -10**5 #quoi qu'il arrive la valeur est au dessus de -1500
mvt = None
"""jeu.suivant_joueur1 renvoie les triplets représentant dans l'ordre :
l'ensemble des mouvement suivant du joueur1, du joueur 2 et l'état associe
sachant que joueur2 choisit mvt_min si il doit jouer"""
for (a, _, s) in jeu.suivants_joueur1(etat, mvt_min, doit_jouer_min, condition_tri(profondeur)):
    """On passe par l'interface et on value la valeur de l'état comme l'ensemble des
    états possibles traités par traitement_alea"""
    new_v = traitement_alea(s, lambda etat_ : fonction_interface(jeu, etat_, \
profondeur, fonction_valeur, traitement_alea, condition_tri, alpha, beta)[1])
    if new_v > v:
        v = new_v
        mvt = a
    if v >= beta:
        return mvt, v
alpha = max(alpha, v)
return mvt, v

```

La fonction de traitement de l'aleatoire (source : minimax.py) :

```

def choix_aleatoire(liste_etats, fonction):
    """ Renvoie la valeur d'un des états pris au hazard."""
    return fonction(choix(liste_etats))

```

Résultats : Le coût des calculs et le facteur de branchement limite la recherche à 2 de profondeur ce qui reste intéressant étant donné que dans la plupart des coups, il faut regarder les mouvements possibles pour les deux joueurs (à cause de l'atomicité des tours).

L'agent obtenu est plus performant que tout les agents précédents.

Son temps de réponse est bien plus élevé que ceux des agent précédents(entre 3 en 20s). Contrairement aux agents précédents, son temps de réponse depend de la phase de la partie. En fin de partie il y a moins de choix possibles donc moins de calculs à faire.

Son comportement est très cohérent ce qui est une bonne réussite car on ne lui a pas donné d'instruction préalable. Dans la plupart des cas il change de pokémon quand il y a un désavantage de type mais pas quand son adversaire est affaibli et qu'il peut le finir parce qu'il est plus rapide. Quand il a un avantage il attaque. Il lui arrive d'utiliser des statuts comme la paralysie et vampigraine, et des bonus.

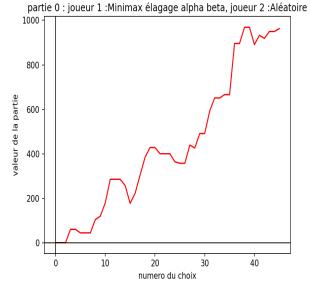
Même si il bat l'IA basique, certaines partie contre cet agent sont très serrées car il lui arrive de faire des erreurs à cause du facteur chance.

Le pokémon qu'il choisit en premier est pikachou. C'est un très bon choix car il est rapide et il possède l'attaque charge éclair qui lui permet de changer de pokémon tout en faisant des dégâts au cas où il serait en désavantage de type.

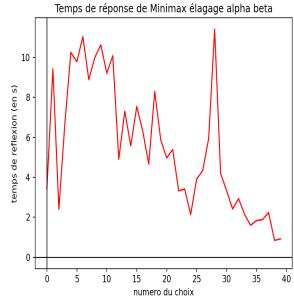
Face à un humain, l'agent commence à être intéressant : il bat les débutants et peut mettre en difficulté les joueurs intermédiaires (Exemple la première partie faite par un des concepteur contre cette ia n'a été gagné avec un seul pokémon restant).

Voici un exemple de partie contre l'aléatoire :

La valeur de la partie (l'agent est en position de max) :

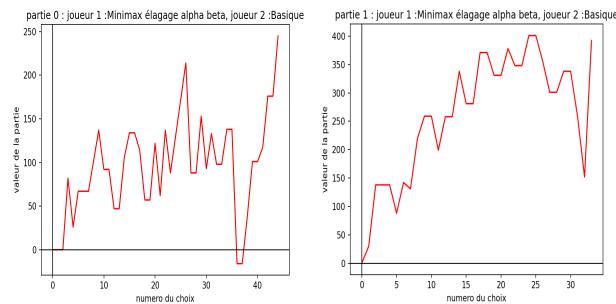


Le temps de réponse : On voit bien sur le graphique qu'il diminue en fin de partie.

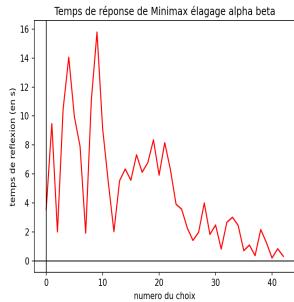


Voici des exemples de partie contre l'ia basique (agent avec un jeu de règle) :

Deux exemples de la valeur de la partie (l'agent est en position de max) :



Un exemple de temps de réponse contre l'IA basique :



Avantages : L'agent n'est pas prévisible comme il ne suit pas un jeu de règles.

Il ne nécessite pas de connaissance particulières dans le jeu et est généralisable à tout les jeux similaires à pokémon (jeu avec atomicité des tours et Stochasticité).

Il est plus performant que tout les agents précédents et surtout en fin de partie.

L'agent n'utilise pas que des attaques qui font des dégâts à court terme, il utilise aussi des statuts et des bonus.

Desavantages : Le temps de réponse de l'agent est assez important. Si on voulait créer une application dans laquelle des joueurs jouent contre notre ia, il serait difficile de leur faire accepter ce délai.

L'agent à du mal à traiter l'aléatoire ce qui l'amène à faire des erreurs.

La faible profondeur de recherche limite la compréhension réelle de l'état du jeu et empêche l'utilisation d'attaques à très long terme comme piège de rock

L'agent est très prévisible en début de partie, sachant qu'il prend toujours pi-kachou en premier, l'adversaire peut choisir son contre.

5.4.4 Expectiminimax

Dans cette section on va apporter une amélioration à l'algorithme alpha beta afin de mieux traiter le facteur chance : l'expextiminimax. Pour ce faire, nous allons simplement changer la fonction de traitement de la chance : à chaque fois qu'il y aura plusieurs états possibles pour une action, la valeur de cette action sera l'espérance des valeurs de tout les états.

Implementation : Seule la fonction de traitement de l'aléatoire change :

L'agent (source : joueurs.py) :

```
class JoueurAlphaBeta2(JoueurMinMax):
    """Joueur qui applique l'algorithme du minimax avec élagage alpha beta."""
    def __init__(self, joueur):
        super().__init__(joueur, 'Expectiminimax')

    def strategie(self, jeu, etat):
```

```
mvt = super().appliquer_minimax(jeu, etat, 2, jeu.valeur, esperance, lambda p : False)
return mvt
```

La fonction de traitement de l'aleatoire (minimax.py) :

Résultats : Le temps de reponse est plus élevé car il faut traiter en plus les branches liées à la chance. Il faut en général entre 10 et 30 secondes pour repondre. Cependant contrairement à l'agent précédent, il arrive que le temps de réponse l'ia expectiminimax stagne voire augmente en fin de partie. L'explication c'est qu'en fin de partie il y a plus de branches liées à la chance. L'augmentation du nombre de branches liées à la chance s'explique par l'augmentation du nombre de situation ou les deux pokemon sur le terrain ont des statuts. En effet, dans ce cas là, si les deux pokemons ont la même vitesse, la chance est à utilisée une première fois pour determiner qui attaque en premier et une seconde fois pour determiner qui subit les statuts en premier. Il y a donc plus de quatres etats possibles.

Ensuite, les performances de l'agent sont supérieures. En effet, grâce à sa maîtrise du facteur chance cet agent bat tout les agents précédents.

Les agents alea et basique sont battus avec un écart de PV très élevé mais en plus de tour qu'il n'en fallait à l'agent minimax pour les battre. Cela montre que l'agent expectiminimax prend moins de risque mais au final est plus précis.

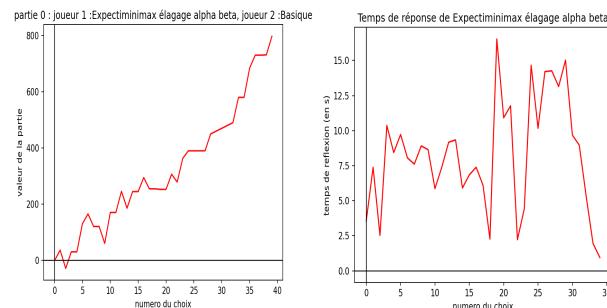
Comme pour l'agent précédent, l'agent expectiminimax choisit pikachou en premier ce qui semble confirmer que c'est un bon choix.

Son comportement est plus cohérent que tout les agents précédents car il ne fait pas d'erreurs liées au facteur chance.

Face à un humain, l'agent est intéressant : il bat les débutants et peut mettre en difficulté voire battre les joueurs intermédiaires.

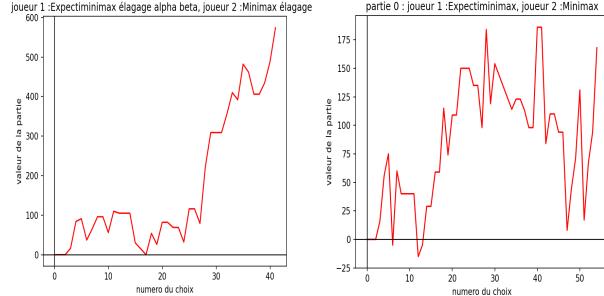
Voici un exemple de partie contre l'ia basique (agent avec un jeu de règle) :

La valeur de la partie (l'agent est en position de max) est à droite et le temps de réponse à gauche (On est dans le cas où il ne diminue pas en fin de partie) :

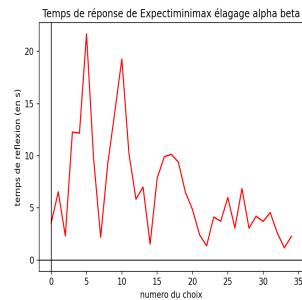


Voici deux exemples de partie contre l'ia alpha beta classique :

Les valeur de la partie (l'agent expectiminimax est en position de max) :



Un exemple de temps de réponse : Cette fois il diminue bien en fin de partie.



Avantages : L'agent prend mieux en compte le facteur chance et fait moins d'erreurs.

La solution proposée est toujours aussi générale, elle peut s'appliquer à tout les jeux à information parfaite, avec atomicité des tours.

Desavantages : Cette amélioration s'est faite au prix d'une augmentation du temps de réponse.

La faible profondeur de recherche reste une limite importante.

L'agent est très prévisible en début de partie, sachant qu'il prend toujours pikkachou en premier, l'adversaire peut choisir son contre.

5.5 Ordre des mouvements

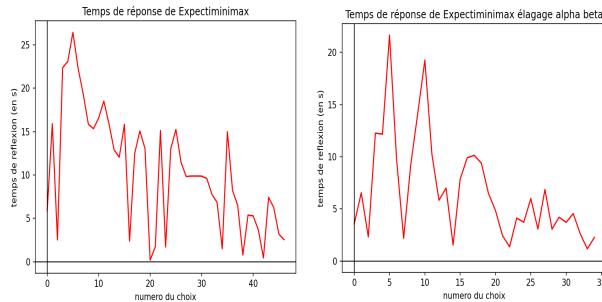
L'ordre de traitement des mouvements a une importance capitale pour l'élagage alpha beta. Plus un noeud correspondant à une valeur élevée est visité tôt, plus il entraînera de coupures. Une pratique habituelle pour trier les mouvements est de faire une recherche superficielle (c'est à dire utiliser l'algorithme du minimax avec une profondeur de 1), avant d'appliquer l'algorithme du minimax.

```

def mouvements_authorized_dresseur(self, etat, dresseur, trier = False):
    """Permet de connaitre les mouvement possibles d'un joueur."""
    try:
        """On stocke les r sultats dans un dictionnaire pour ne pas avoir
        recalculer si on recroise le m me etat"""
        return self.mvts_authorized[dresseur.id][etat]
    except KeyError:
        mvts = []
        #r cup ration des mouvement possibles
        if dresseur.doit_changer:
            mvts = dresseur.liste_changements
        elif etat.phase == 'debut':
            liste_changements = dresseur.liste_changements
            liste_attaques = [attaque for attaque in dresseur.courant.tuple_attaque]
            mvts = liste_attaques + liste_changements
        if trier:
            #tri superficiel ou swallow search
            if dresseur.id == 'joueur1':
                mvts.sort(key = lambda mvt : minimax_joueur1(self, etat, 1, \
                    self.valeur, choix_aleatoire, lambda p : False)[1])
            else:
                mvts.sort(key = lambda mvt : minimax_joueur2(self, etat, 1, \
                    self.valeur, choix_aleatoire, lambda p : False)[1], reverse = True)
        self.mvts_authorized[dresseur.id][etat] = mvts
    return mvts

```

Résultat : Dans le cas de notre jeu, une recherche superficielle demande trop de calculs par rapport au temps gagné par les coupures supplémentaires qu'elle entraîne. Ainsi, on observe pour tout les agents utilisant le minimax, une augmentation du temps de recherche lorsqu'on trie au préalable à la profondeur 0. (Pour trier à la profondeur 0, il faut donner à la variable condition_tri l'expression : lambda profondeur : profondeur j=1 au lieu de lambda p : False). Exemple : Voici le temps de réponse de l'agent expectiminimax contre l'agent minimax, avec tri à droite et sans tri à gauche.



On voit que avec le tri les délais sont plus longs. On trouve des résultats similaires pour l'agent minimax.

5.6 Solutions Envisageables

Deux mois de projet c'est une longue période pour un projet scolaire, mais avec un sujet aussi vaste, il reste toujours des pistes à creuser, des détails à peaufiner.

Cette section sera donc dédiée à des pistes que nous aurions voulu explorer sans pour autant avoir le temps de le faire.

5.6.1 Ordre des mouvements

Un des plus grand défauts des ia les plus performantes que nous avons réussit à coder est le temps de reponse. Avec un temps de réponse plus court, on pourrait soit améliorer l'expérience utilisateur, soit augmenter la profondeur de recherche et avoir une ia plus performante. Ainsi, pour les améliorer nos ia, un bon tri des mouvements avant les recherches serait intéressant. Pour cela, on pourrait utiliser deux choses :

- Un tableau de hashage qui stocke les mouvements les plus efficaces qui sont à tester en premier.
- Lorsque deux états ne diffèrent que par un facteur chance et qu'on a déjà fait l'algorithme du minimax pour le premier, on peut stocker le mouvement résultant et l'explorer en premier. Souvent deux états qui ne diffèrent que d'un facteur chance sont très proches et le mouvement stocké sera le bon créant donc de nombreuses coupures.

5.6.2 Recherche de principale variation

Toujours dans l'idée de réduire le temps de calcul, on pourrait se pencher sur un amélioration de l'algorithme alpha beta qui est nommé recherche de principale variation ou PVS (Principal Variation Search). C'est un algorithme qui repose sur un bon tri préalable des mouvements c'est pourquoi le point précédent sera important. Il se déroule de la manière suivante :

- On tri les mouvements disponibles.
- Pour le premier mouvement on fait une recherche complète avec l'algorithme minimax
- Pour tous les autres on fait une recherche de fenêtre nulle (zero window search). C'est une recherche pour laquelle $\beta = \alpha + 1$. Ce type de recherche produit bien plus de coupure que l'élagage alpha beta et ne renvoie qu'une information : est-ce que le mouvement améliore la valeur actuelle.
- Pour tous les mouvements qui ont amélioré la valeur actuelle, on effectue une recherche complète.
- On remonte la valeur de la recherche complète qui est la plus grande pour le max et la plus petite pour le min.

Ainsi, si l'ordre des mouvements est bon on aura fait très peu de recherches complètes et une majorité de recherches à fenêtre nulle et on aura économiser un temps précieux.

6 interfaces

6.1 Introduction

Pour représenter notre jeu, nous avons donc créer 2 interfaces : une interface textuelle et une interface graphique. Afin de faciliter la transition entre ces deux interfaces et le programme du jeu, qui est donc le même pour toute l'interface, nous avons décidé de faire en sorte qu'elles aient toutes les deux la même structure. Elles sont ainsi simplement différentes dans la façon dont elles récupèrent/affichent les informations. C'est au début de la partie que l'une ou l'autre seront choisie par l'utilisateur via la console.

6.2 Interface Textuelle

Lorsque c'est l'interface textuelle qui est choisie, l'entièreté de la partie se déroulera alors dans la console, que ce soit les choix des joueurs ou les informations sur le déroulé du jeu. Au cours de la partie, les différents choix à effectuer par le/les joueur humain(s) si il y'en a seront à taper directement dans la console (pour les seuls résultats de leurs choix s'afficheront). Dans le cas de l'interface textuelle, une fonction 'input sécurisé' assure la bonne compréhension des réponses des utilisateurs. Celle-ci ne laissez passer que des réponses pré définies (ex : un nom de pokémon, un nom d'attaque) et permet ainsi d'éliminer les erreurs dues aux réponses non-conformes des utilisateurs (nom mal orthographié ou inexistant).

6.2.1 déroulé d'un tour

Dans un premier temps, il faudra taper dans la console la nature des deux joueurs ainsi que leurs nom et leur pokémon de départ, en répondant aux questions correspondantes dans la console comme explicité ci-contre :(cf image 1 au point 6.2.2)

Pour chaque tour (après un récapitulatif des choix respectifs des deux joueurs si il s'agit du premier tour), les deux joueurs sont appelé à choisir ce qu'ils vont faire, en choisissant entre attaquer ou changer de pokémon, avec la possibilité d'avoir accès aux informations de la partie avant de choisir. Les différents choix d'attaque ou de pokémon possible seront rappelés au joueur avant qu'il ne choisisse. Voici des exemples :(cf image 2 au point 6.2.2)

A la fin du tour, un résumé des différents actions ayant eu lieu durant ce tour sera affiché dans la console, accompagné d'un bilan global de l'état du jeu à ce tour, avec les informations sur les pokemons des deux joueurs comme présenté ci-contre :(cf image 3 au point 6.2.2)

Enfin, lorsque qu'un joueur n'a plus de pokémon en état de combattre, un message vient annoncer le vainqueur et la fin de la partie.(cf image 4 au point 6.2.2)

6.2.2 affichages en textuel :

Touts les affichages de l'interface textuelle sont gérés par des print appliqués par les différentes fonction de la classe InterfaceTexte ou par le fichier jeu (voir architecture du programme) pour ce qui concerne les informations destinées aux utilisateur et par la fonction 'input securise' pour ce qui concerne les informations données par l'utilisateur au programme.

Voici quelques capture d'écran d'une partie avec l'interface textuelle.

Image 1 : début de partie

```
In [1]: runfile('C:/Users/Soren/Documents/Info prepa/majeure info/projet jeu/Projet-majeure/main.py',  
              wdir='C:/Users/Soren/Documents/Info prepa/majeure info/projet jeu/Projet-majeure')  
  
Quelle interface choisissez-vous (texte / graphique)?texte  
  
Le joueur 1 (ia alea, ia alpha beta 1, ia alpha beta 2, ia basique, humain) : humain  
  
Le joueur 2 (ia alea, ia alpha beta 1, ia alpha beta 2, ia basique, humain) : humain  
  
nom du joueur 1:Test1  
  
nom du joueur 2:Test2  
  
Test1 Choisissez un pokémon parmi pikachou, boulbizarre, osselait, salamèche, carapuce, étourmi, ou  
menu pour revenir au menu: pikachou  
  
Test2 Choisissez un pokémon parmi pikachou, boulbizarre, osselait, salamèche, carapuce, étourmi, ou  
menu pour revenir au menu: ossela
```

Image 2 : Choix lors d'un tour

```
Test1 Voulez-vous attaquer ou changer de pokémon ? (attaquer, changer, information)attaquer  
  
Test1 Choisissez une attaque parmi tonnerre, cage éclair, rugissement, change-éclair, ou menu pour revenir  
au menu: tonnerre  
  
Test2 Voulez-vous attaquer ou changer de pokémon ? (attaquer, changer, information)changer  
  
Test2 Choisissez un pokémon parmi pikachou, boulbizarre, osselait, carapuce, étourmi, ou menu pour revenir  
au menu: étour
```

Image 3 : Fin du tour

```
Test2 choisit étourmi
le pikachou de Test1 attaque tonnerre (PP restantes: 14)
étourmi de Test2 perd 274 PV.
étourmi est KO
Etat du jeu :

Test1 :
Pokemon Ko :
Pokemon courant :
Nom : pikachou de Test1 :
PV : 210
Statuts : aucun
Bonus : attaque : 1, defense : 1, vitesse : 1
Statut de dresseur : aucun
Test2 :
Pokemon Ko :étourmi
Pokemon courant :
Nom : étourmi de Test2 :
PV : 0
Statuts : aucun
Bonus : attaque : 1, defense : 1, vitesse : 1
Statut de dresseur : aucun
valeur du jeu estimée : 220
```

Image4 : Fin de la partie

```
le étourmi de Test2 attaque rapace (PP restantes: 5)
osselait de Test1 perd 65 PV,
osselait est KO
Test1 a perdu la partie.

valeur du jeu estimée : -949
```

6.3 Interface Graphique

L'interface graphique reprends donc la même structure que l'interface textuelle. Cependant, dans le cas de l'interface graphique, le joueur effectue ses choix via des boutons ou des zones de texte apparaissant sur la fenêtre du jeu et non plus directement via la console. Le jeu se joue donc sur une fenêtre générée par le programme avec, sur la partie gauche, un decor ou apparaissent les pokemons actuels des deux joueurs ainsi qu'une partie sur la droite servant pour effectuer les choix/afficher les informations relatives au jeu. La nature de ces parties vous sera détaillée dans le point 6.3.2. Afin de pouvoir récupérer les choix des joueurs, on utilise les fonctions waitvariable et waitwindow qui nous permettent, via les

fonctions liées aux boutons qui affectent ainsi une certaine valeur à la variable récupérée par le programme, de mettre celui-ci en suspens tant que le bouton n'est pas cliqué.

6.3.1 déroulé d'un tour

Comme pour l'interface textuelle, il faut d'abord définir le type, le nom et le premier pokémon des deux joueurs. Pour le type et le premier pokémon, un simple clic sur le bouton voulu suffit. Pour ce qui est du nom, une zone de texte apparaît avec 'nom du joueur' en valeur par défaut. Il suffit de supprimer ce texte, de taper le pseudo voulu et d'appuyer sur la touche 'entrée' du clavier pour valider.(cf image 1, 2, 3 au point 6.3.2)

Pour chaque tour (un récapitulatif des choix respectifs des deux joueurs apparaît quelques secondes si il s'agit du premier tour), comme pour l'interface textuelle, les deux joueurs choisissent entre attaquer, changer, ou avoir des informations avant de faire ce choix. Ici, ces choix se gèrent grâce à des boutons à cliquer.(cf image 4, 5, 6, 7 au point 6.3.2)

Une fois ces choix effectués, ils s'appliquent et un récapitulatif des actions réalisées et des nouvelles informations apparaît pendant quelques secondes.(cf image 8 au point 6.3.2)

Enfin, lorsque qu'un joueur n'a plus de pokémon en état de combattre, les informations récapitulant le dernier tour apparaissent avec un message annonçant le vainqueur et la fin de la partie.(cf image 9 au point 6.3.2)

6.3.2 affichages graphiques :

La fenêtre générée est créée via le module Tkinter. Elle est composée de 2 frames, une grande à gauche et une autre plus petite à droite. Dans notre programme, la fenêtre est définie dans le fichier fenêtre avant d'être initialisée en tant qu'attribut de la classe InterfaceGraphique. Nous tenons à préciser que nous avons essayé de reprendre au mieux l'interface de combat graphique des jeux pokemons originaux ainsi que des images issus des premiers jeux de la license pour ce qui concerne les pokemons et le décor notamment.

- La grande frame de gauche est dédiée au décor du jeu. Elle contient un grand canva qui la recouvre entièrement, dans lequel on implémente l'image servant de paysage de fond à notre jeu. Sur ce canva sont disposés, lors du choix des pokémons, 2 canvas contenant les images (sprites) des pokémons des deux joueurs. Le joueur 1 se situe en bas à droite, de dos, tandis que le joueur 2 se situe en haut à gauche, de face. Lors d'un changement de pokémon, le canva du pokémon concerné est détruit et remplacé par un nouveau contenant le nouveau pokémon.

- La frame de droite, elle, sert d'interface entre l'utilisateur et le jeu et d'endroit pour afficher les informations du jeu. Cette frame est en réalité un objet d'une des classes ci-dessous :
 - IHM, frame permettant de faire le choix attaquer, changer ou information.
 - Attaquef, frame correspondant au choix attaquer.
 - Changerpoke, frame correspondant au choix de changer de pokémon.
 - Information, frame correspondant au choix information.
 - Demandernom, frame correspondant au choix de pseudo en début de partie.
 - Choixjoueur, frame correspondant au choix du type de joueur en début de partie.
 - Etatjeu, frame affichant l'état du jeu entre les tours.

A chaque changement de frame, celle-ci est détruite puis remplacée par une nouvelle instance d'une des classes ci-dessus.

Pour ce qui est des boutons, ils sont créés via Tkinter et sont liés à une fonction qui permet d'affecter une certaine valeur à l'attribut 'choix' de la fenêtre qui est par la suite récupérée et traitée par l'interface. Le fonctionnement est similaire pour le champs de texte permettant de renseigner le pseudo des joueurs en début de partie, qui est un Widget Entry de Tkinter affectant la valeur reçue lors de la pression de la touche 'entrée' à l'attribut 'choix' de la fenêtre.

Voici quelques images pour l'interface graphique : Image 1 : début de partie

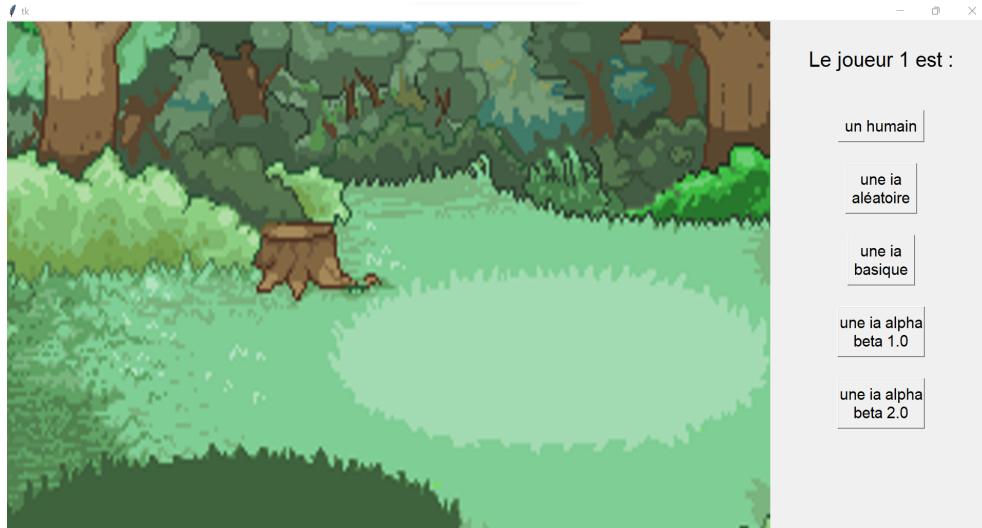


Image 2 : choix des noms :

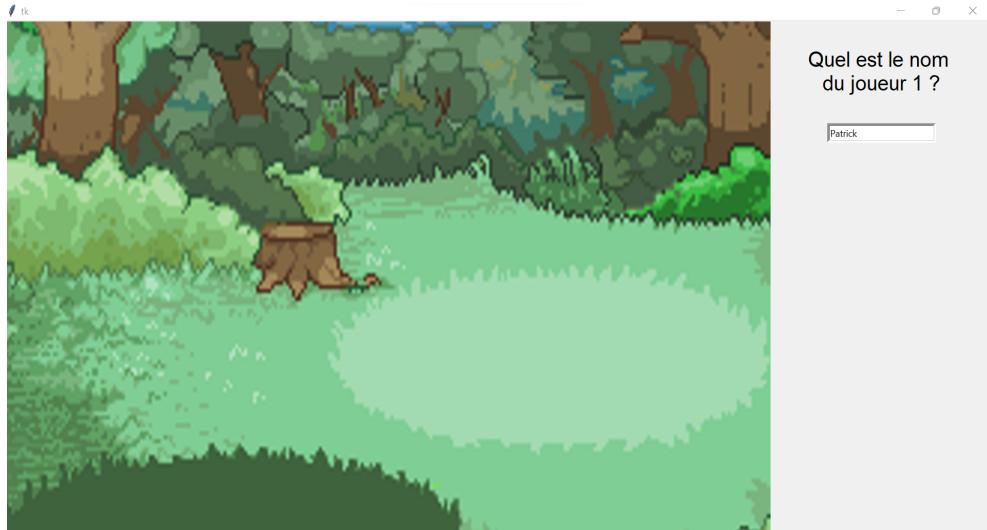


Image 3 : choix premier pokémon :



Image 4 : choix action :



Image 5 : choix des pokemons :

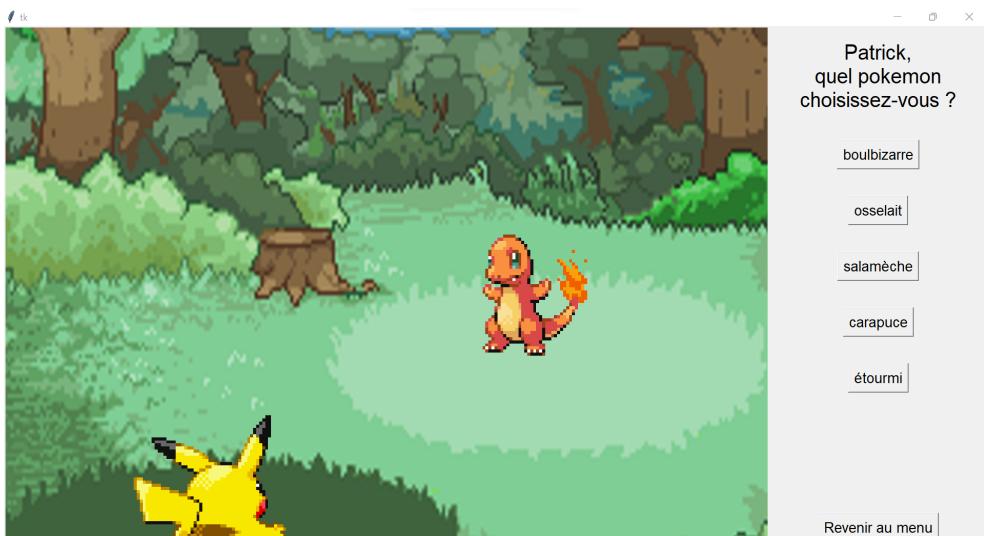
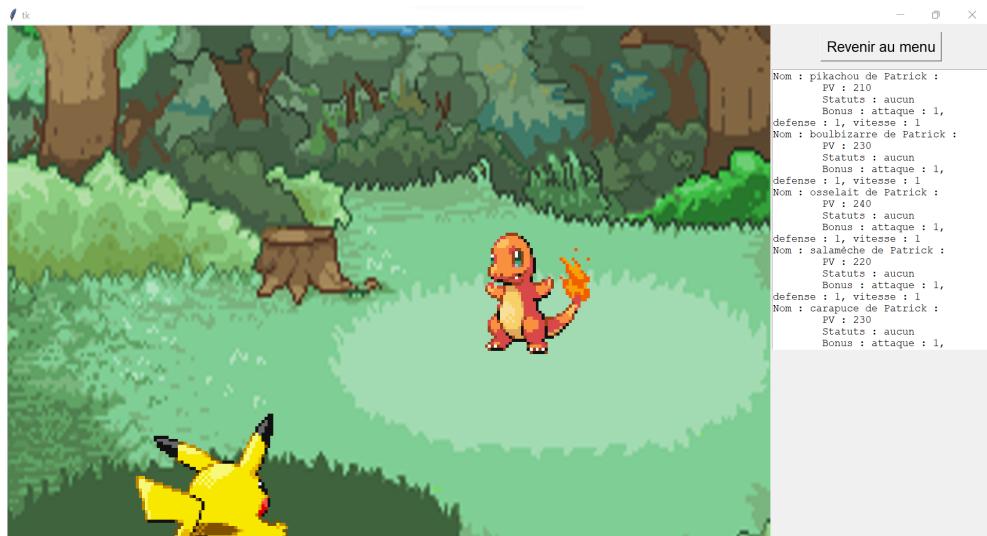


Image 6 : choix des attaques :



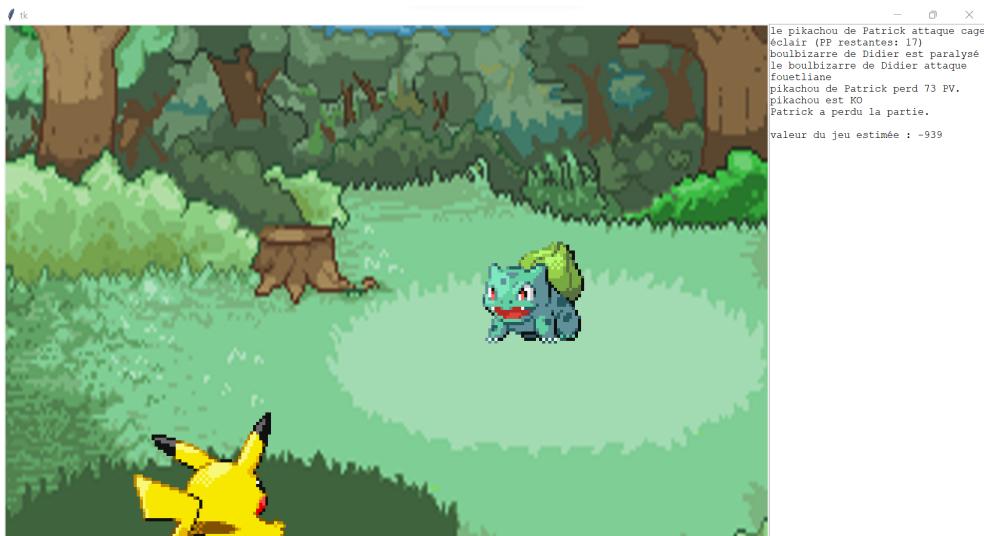
Images 7 : information :



Images 8 : état du tour :



Images 9 : Fin de la partie :



7 Manuel utilisateur

Afin de pouvoir lancer le jeu, il est nécessaire de télécharger tous les fichiers fournis ainsi que les images, fournies également. Celles-ci sont à ranger dans un dossier 'images' pour la bonne exécution du code. Une fois que tout est prêt, lancer le programme via le fichier 'main.py'. Une question portant sur le choix de l'interface vous sera alors posée. Une fois ce choix fait, la partie se lance. Suivez le déroulé des tours présenté dans les points 6.2.1 pour l'interface textuelle et 6.3.1 pour l'interface graphique. Ils détaillent les choix que vous serez amenés à faire ainsi que la méthode pour les donner, en plus d'indication pour pouvoir visualiser les différentes informations relatives à la partie en cours.

Si vous voulez lancer un nombre important de parties, il faudra augmenter la profondeur maximale de récursion ligne 130 de abstract_jeu.py

8 Sources

Le papier de recherche sur un ia deeplearning et pokemon : **Learning complex games through self play - Pokémon battles** de *Miquel Llobet Sanchez* : <https://upcommons.upc.edu/handle/2117/121655>

Des articles wikipedia qui nous ont été utiles :

- Minimax : <https://en.wikipedia.org/wiki/Minimax>
- Alpha Beta : https://en.wikipedia.org/wiki/Alpha_Beta_pruning
- Expectiminimax : <https://en.wikipedia.org/wiki/Expectiminimax>
- PVS : https://en.wikipedia.org/wiki/Principal_variation_search
- Théorie des jeux : https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_jeux

Pour toutes les informations/images concernant les pokemons : <https://www.pokepedia.fr/> Portail :Ac-cueil et <https://www.pokebip.com/> (a noter que nos pokemons ont les noms des jeux réels mise à part Pikachu (Pikachu) et Boulbizarre (Bulbizarre))

Un article sur l'ordre des mouvements : https://www.chessprogramming.org/Move_Ordering