Vampires vs Werewolves

Report AI Challenge

Nicolas GREVET
MSC AI at CentraleSupelec
nicolas.grevet@student-cs.fr

Zakariae EL ASRI
MSC AI at CentraleSupelec
zakariae.elasri@student-cs.fr

Charles BOY DE LA TOUR
MSC AI at CentraleSupelec
charles.boy-de-la-tour@student-cs.fr

Shawn SIDBON
MSC AI at CentraleSupelec
shawn.sidbon@student-cs.fr

Introduction

Qui des Loups-Garous ou des Vampires contrôleront la carte ? La stratégie utilisée pour cette bataille sera-t-elle celle qui mènera à la victoire finale ? Nous proposons dans ce rapport de détailler les choix que nous avons fait ainsi que l'algorithme implémentée afin d'atteindre l'objectif final : la victoire totale !

Nous sommes dans le cadre d'un jeu ouvert avec 2 joueurs et à somme nulle (1 gagnant, 1 perdant). Chaque joueur a toute l'information sur le jeu car l'environnement est déterministe et totalement observable. Le plan de jeu est séquentiel car chaque décision prise à un moment donné aura un impact sur celles à venir. De plus un ensemble de données récupérables nous permettent de connaître l'état du jeu à la fin de chaque tour.

A partir de ces informations, l'un des aspects importants du jeu est de prendre en compte les coups joués par l'adversaire et ne pas se contenter d'être une IA naïve. Plusieurs techniques de ***Adversarial search*** (pruning, fonction d'évaluation heuristique) associées aux algorithmes minimax ou alpha-beta doivent nous permettre de développer une solution au regard de cette contrainte. Ce rapport a pour vocation de décrire les étapes de construction de notre IA : ***zcsn*** qui va lui permettre de choisir le meilleur coup à jouer à chaque tour.

Les règles de jeu sont initialisées afin d'équilibrer le jeu. Elles sont précises et formalisables. Toutes les actions que notre IA effectuera devra respecter ces règles. Lorsqu'une Bataille Aléatoire se lance, notre IA doit aussi être capable de réagir en fonction de son résultat afin de conserver toutes ses chances de devenir l'espèce dominante.

Une contrainte supplémentaire dont notre IA doit faire face est la limite de temps de 2 secondes qu'elle ne doit pas dépasser pour faire son calcul et renvoyer le coup à jouer. L'optimisation de notre code doit donc aussi prendre en compte ce paramètre. C'est pourquoi nous avons fait le choix d'utiliser l'algorithme alpha-beta en langage Python et de l'optimiser de façon itérative.

Table des matières

1. Plan de jeu

1.1. Etat du jeu

Afin de suivre les changements du plan de jeu à chaque tour, nous avons modélisé la MAP sous forme de 3 dictionnaires : Humains, Alliés, Ennemis. Chaque dictionnaire contiendra l'ensemble des positions de chaque espèce sous forme (Coordonnées, Nombre).

Tout au début du jeu, nous allons récupérer notre position initiale puis un plan de jeu de départ complet indiquant, pour chaque espèce, les coordonnées et le nombre. En croisant les deux informations, nous pourrons déterminer notre espèce (Vampires ou Loups-garous) et ainsi remplir les 3 dictionnaires avec la situation du début.

Après chaque coup de l'adversaire, on recevra une update indiquant les changements survenus à la MAP après notre dernier coup et celui de l'adversaire. Cette update nous permettra de mettre à jour nos dictionnaire pour décrire le plan de jeu actuel.

1.2. Déplacements sur la carte

Pour se déplacer sur la carte, plusieurs étapes sont nécessaires :

Tout d'abord, à chaque tour, nous devons commencer par nous localiser ; i.e déterminer les positions des groupes alliés. Puis déterminer les mouvements possibles à partir de cet état qui seront regroupé dans une liste.

Nos mouvements dépendront fortement de la situation de des ennemis et des humains, d'où l'intérêt de localiser les espèces autours de nous (humains et/ou ennemis)

En fonction de l'espèce localisé et de son nombre nous avons définis deux options possibles : attaquer (***Move Toward)***, ou fuir (***Flee)***.

Les déplacements sur la carte sont gérés par un nombre de règles (cf. Règles du jeu). Afin de respecter ces règles, nous avons défini une fonction de contrôle ***isLegalMove*** que l'on va appliquer à chaque mouvement possible afin d'affiner notre liste de mouvements possibles. Après cette étape, nous pouvons évaluer chaque mouvement afin de trouver le meilleur.

Afin de respecter la contrainte du temps de calcul, nous avons limité le nombre de split de nos groupes à 2 groupes seulement afin de bénéficier du temps de calcul dans l'exploration en profondeur

1.3. Distance sur la carte

Le calcul de distance entre deux positions sur la carte est primordial dans ce jeu afin de localiser les groupes les plus proches.

Il s'agit d'un jeu avec des éléments mobiles séquentiellement (une espèce bouge chaque tour), la distance doit exprimer ainsi le nombre de tours minimal requis pour se déplacer d'une position initiale à une position cible.

Vu que les déplacements sont autorisés en diagonale aussi et non seulement verticalement et horizontalement, la distance euclidienne ne sera pas une métrique intéressante dans notre cas. En effet, si on considère la MAP ci-après, il ne faudra 2 coups minimum pour se déplacer de C à A. Aussi, il nous faudra 2 coups minimum pour se déplacer de C à B. On peut dire ainsi que pour notre jeu, les cases A et B sont situés à la même distance de C.

A		
B		C

C'est pour cette raison que nous avons défini une version modifiée de la distance de Manhattan comme métrique :

$$D\_Manhattan\ (x1, y1), (x2, y2) = Max\ (|x1 - x2|\ ;\ |y1 - y2|)$$

2. Stratégie du jeu

Puisque ce jeu oppose deux joueurs et que les coups de l'adversaire ne sont pas connus à l'avance, nous devons implémenter un algorithme pour à la fois prédire les coups de l'adversaire et aussi répondre par le meilleur coup

qui nous donnera la meilleure situation à moyen terme. A cette fin, nous choisissons d'implémenter un algorithme minimax amélioré avec l'alpha-beta pruning.

A chaque tour, nous adoptons la stratégie décomposée en 4 étapes :

1. Détermination de la phase de jeu

   A partir de l'état du jeu, nous déterminons la phase de jeu. Nous avons défini deux phases de jeu :
   - *Phase de construction :* elle correspond au début de la partie où nous allons chercher à conquérir plus d'humain afin de renforcer nos rangs et devenir nombreux. Cette phase sera caractérisée par un nombre d'humain supérieur à un nombre de référence ***HUM_THRESH***.
   - *Phase d'attaque* : elle correspond à la phase finale de la partie où le nombre d'humain sera inférieur à ***HUM_THRESH*** et où le nombre d'ennemis relativement grand. Au cours de cette phase nous allons chercher des batailles contre l'ennemi.

2. Recherche des groupes cibles

   Ensuite, nous cherchons les groupes cibles les plus proches de l'espèce visée. Nous avons alors deux cas de figure : Humains ou Ennemis. Pour respecter la règle du temps de calcul, nous décidons de limier le nombre de groupes par espèce. Après plusieurs tests, nous décidons de limiter ce nombre en fonction des cas de figure :
   - 1ère phase - Humains :
     - 4 groupes d'humains lorsque nos alliés sont groupés dans une seule position.
     - 3 groupes d'humains lorsque nos alliés sont divisés en 2 positions.
   - 2ème phase - Ennemis :
     - 2 groupes d'ennemis.

3. Liste des mouvements possibles

   Après avoir déterminé les groupes visés, nous listons les mouvements possibles en fonction des cas de figure :
   - 1ère phase - Humains :
     - Mouvement 1 : faire bouger notre groupe en un seul bloc vers le bloc d'humains.
     - Mouvement 2 : faire bouger deux groupes alliés vers 2 blocs d'humains.
     - Mouvement 3 : regrouper les blocs alliés en 1 bloc (si séparation au préalable).
   - 2ème phase - Ennemis :
     - Mouvement 1 : avancer vers l'ennemi.
     - Mouvement 2 : fuir l'ennemi.

   Nous soumettons chacun des coups potentiels à une fonction de contrôle afin de s'assurer que ce n'est pas un mouvement interdit.

4. Détermination du meilleur mouvement

   L'enjeu maintenant est de sélectionner, parmi la liste définie à l'étape précédente, le mouvement qui nous permettra d'avoir la meilleure situation possible sur la carte.

   Nous faisons ici appel à l'algorithme Minimax optimisé par alpha-beta pruning.

   Nous déroulons les mouvements potentiels un par un et pour chaque mouvement nous faisons une exploration en profondeur de l'arbre de situations possibles jusqu'à atteindre la profondeur limite qu'on va définir, à cette profondeur, nous allons évaluer la situation du jeu selon une fonction heuristique (détaillée section 4). Ces valeurs d'heuristique calculées à la profondeur limite et rétro-propagées à la racine nous permettrons de sélectionner le meilleur mouvement à exécuter.

   Le gain en temps induit du pruning sera exploité pour étendre l'arbre plus en profondeur.

   Lorsque l'ordre de jouer du serveur est donné, notre algorithme Minimax commence par extraire l'état actuel de la MAP, en fait une copie pour y simuler le déroulement de chaque mouvement potentiel et les mouvements qui suivent.

   A la fin, nous avons le meilleur mouvement à jouer ainsi que son score relatif et nous envoyons au serveur le meilleur mouvement identifié.

3. Le choix de l'heuristique

Du fait de la combinatoire importante du jeu, l'exploration de la totalité de l'arbre est impossible. Nous devons donc limiter notre IA dans la profondeur de l'arbre. Il devient alors nécessaire de sélectionner une bonne heuristique qui évalue bien chaque situation de la MAP et reflète la chance de gagner.

Nous avons commencé par l'implémentation d'une heuristique simple sur laquelle nous avons itéré après avoir effectué des tests. Notre heuristique finale est composé de plusieurs évaluations :

- **Distance entre alliés et humains : Dist**
  Evalue la distance qui sépare les alliés aux 2 groupe d'humain les plus proches.
- **Distance entre groupe d'alliés : Intra_dist**
  Evalue la distance qui sépare les deux groupes d'alliés. En soustrayant l'inverse cela met en valeur les groupes éloignés et donc pousse les groupes à se séparer (essentiellement au début).
- **Nombre d'humains sur la carte : human.values()**
  Evalue le nombre d'humain présents.
- **Cumul d'évolution des alliés : UsIncrease**
  Evalue l'évolution de nombre le gain (/ perte) du nombre d'alliés auquel on applique un discount factor basé sur la profondeur à laquelle l'action a été jouée.
- **Cumul d'évolution des ennemis : EnnemyIncrease**
  Evalue l'évolution de nombre le gain (/ perte) du nombre d'ennemis auquel on applique un discount factor basé sur la profondeur à laquelle l'action a été jouée.

Comme pour la recherche de mouvements potentiels, nous avons défini aussi deux heuristiques pour chaque cas de figure :

- 1ère phase - Humains :
  - Nous avons 1 groupe :
    *UsIncrease + 0.9*EnemyIncrease – 1.1*sum(human.values()) – 0.1*dist – 1/intra_dist*
  - Nous avons 2 groupes :
    *UsIncrease + 0.9*EnemyIncrease – 1.1*sum(human.values()) – 0.1*dist*
- 2ème phase - Ennemis (il n'y a plus d'humains sur la carte) :
  *UsIncrease – 1.2*EnemyIncrease – 2*n_groups – dist*

La décision des poids a été mené de façon itérative en simulant plusieurs combats et en ajustant au fur et à mesure. Un Script en SSH nous aurait permis de simuler plusieurs parties (cf. groupe JPG), cependant faute de temps nous faisions les tests manuellement.

4. Algorithme

Nous avons implémenté notre stratégie de jeu en langage Python. Nous nous sommes appuyés sur une architecture de classes :

- **Client**
  Similaire aux autres groupes.
  Ce fichier nous permet de nous connecter au client ainsi que de recevoir et envoyer des messages.
- **Main**
  Ce fichier fait appel aux autres classes et fichiers. Il s'agit du fichier que l'on exécute pour instancier notre IA. On y trouve la boucle *while* du jeu qui continue temps que le jeu tourne.
- **Config**
  Contient des paramètres utiles à tous les fichiers :
  - **IDX_US, IDX_EN** : Il s'agit de l'index dans les messages d'update, ce qui simplifie la tâche de lecture des messages et de l'update des états des joueurs.
  - **X_LIM, Y_LIM** : Il s'agit des dimensions limite de la carte.
  - **MAX_DEPTH** : Il s'agit de la profondeur maximale autorisée à l'algorithme.

- **MAX_GROUPS** : Il s'agit du nombre maximum de groupes autorisés. Pour éviter de dépasser le temps limite.

- **Gamestate**
  Fichier où se trouve la class **GAME_STATE** qui nous permet de mettre à jour nos états de manière simple.
  La fonction principale est ***update_treatment*** qui permet de vérifier si l'on doit supprimer une clef, en ajouter une nouvelle ou si l'on doit modifier la valeur associée à une clef dans notre dictionnaire.

- **Utils**
  Le fichier le plus lourd qui se décompose en 3 parties :

  1. Le calcul de distance : `distance_player(player1,player2)`

  La fonction principale retourne une liste de liste de la distance entre les 2 joueurs, la position des ennemis (ou humains) ainsi que leur quantité.

```
def distance_player(player1, player2):

    """
    :param player1: dictionary of player 1 states
    :param player2: dictionary of player 2 states
    :return: distance (sorted) - list of list of ints,
            positions - list of list of positions,
            quantities - list of list of ints
    """

    total_distances = []
    total_positions = []
    total_quantities = []

    for (xP1, yP1) in player1.keys():
        distances = []
        positions = []
        quantities = []

        for (xP2, yP2) in player2.keys():
            distances.append(manhattan_distance(xP1, xP2, yP1, yP2))
            positions.append((xP2, yP2))
            quantities.append(player2[xP2,yP2])

        # sorting with respect to distances

        if distances:
            distances, positions, quantities = zip(*sorted(zip(distances,
positions, quantities)))

            total_distances.append(list(distances))
            total_positions.append(list(positions))
            total_quantities.append(list(quantities))

        else:
            total_distances.append([0])
            total_positions.append([])
            total_quantities.append([0])


    return total_distances, total_positions, total_quantities
```

2. La vérification des cellules : `enemy_in_cell(us, enemy, x, y, x_update, y_update, q1)`

3 questions se posent : y a-t-il un ennemi ? y-a-t-il un humain ? y-a-t-il un allié ? (et implicitement la case est-elle vide ?)

Nous allons ici détailler la fonction vérifiant la présence d'un ennemi dans la case car elle présente les mêmes caractéristiques que les 2 autres fonctions et en plus à une condition supplémentaire dus aux batailles aléatoires.

On vérifie d'abord que si l'on est 1.5 fois supérieur ou inférieur au nombre d'ennemis et dans ce cas le résultat est simple. Un des deux est éliminés et les positions sont mises à jour.

Si ce n'est pas le cas nous rentrons dans une bataille aléatoire, si l'on est plus, on considère que l'on gagne et on met à jour l'état du groupe en y soustrayant l'espérance de perte d'humains. Et inversement si l'ennemi est supérieur.

```
def enemy_in_cell(us, enemy, x, y, x_update, y_update, q1):
    """
    :param us: our state
    :param enemy: the enemy's state
    :param x: the initial position
    :param y: the initial position
    :param x_update: the position of interest
    :param y_update: the position of interest
    :param q1: quantity displaced by us
    :return:
    """

    p = 1

    q2 = enemy[(x_update, y_update)]

    # If the difference is 1.5 times higher or 1.5 times lower then no
randomness
    if 1.5*q2 <= q1:
        us[(x_update, y_update)] = q1
        del enemy[(x_update, y_update)]

        if q1 == us[(x,y)]:
            del us[(x, y)]
        else:
            us[(x, y)] -= q1

    elif q2 >= 1.5 * q1:

        if q1 == us[(x,y)]:
            del us[(x, y)]
        else:
            us[(x, y)] -= q1

    # Else, a random battle starts:
    # If we are more we win and keep p*initial number of creatures
    # If we are less we lose and the enemy keeps (1-p) * inital number of
creatures

    else:
        (q1_temp, q2_temp), p = random_battle(q1, q2)
        if q1_temp != 0:
            us[(x_update, y_update)] = q1_temp
            del enemy[(x_update, y_update)]

            if q1 == us[(x,y)]:
                del us[(x, y)]

            else:
                us[(x, y)] -= q1

        else:
            enemy[(x_update, y_update)] = q2_temp

            if q1 == us[(x,y)]:
                del us[(x, y)]

            else:
                us[(x, y)] -= q1

    return us, enemy, p
```

3. Le calcul des mouvements possibles : `compute_possible_moves(us, enemy, human, isEnemy = False)`

Plus compliqué mais l'idée et que l'on considère que l'on en peut que peut bouger vers un nombre arbitraire d'humains, que l'on peut bouger vers / et à l'opposé d'un groupe arbitraire d'ennemis et que l'on peut se séparer (uniquement en fonction des humains i.e. seule la position des humains dicte s'il est intéressant de se séparer) et si l'on a déjà 2 groups, la possibilité de merge.

Pour cela nous avons identifié plusieurs cas :

- Au début on se concentre essentiellement sur les humains mais la partie minimisation elle continue de prendre en compte les alliés
- Une fois qu'il reste moins de 2 groups d'humains, on commence à prendre en compte les ennemis dans nos mouvements possibles

En outre la profondeur peut être légèrement modifié en fonction des différentes conditions. Par exemple dans le cas où il y'a moins de 2 groupes d'humains et qu'il y'a plus de 2 groupes d'ennemis nous avons baissé la profondeur maximale d'1 niveau pour rester sous les 2 secondes.

```
def compute_possible_moves(us, enemy, human, isEnemy = False):

    """
    :param us: player 1 state
    :param enemy: player 2 state
    :param human: human state
    :param isEnemy: refers to wether it is the minimization turn to play as
we always want to be wary of getting close to ennemies
    :return: list of possible moves
    """
    HUM_THRESH = 4
    ENEMY_THRESH = 2

    num_e = len(enemy)
    num_h = len(human)
    num_u  = len(us)

    possible = False

    for elem in list(human.values()):
        if elem < sum(us.values()):
            possible = True


    # if there are few humans then consider the ennemies

    if ((num_h < HUM_THRESH or possible == False) and (num_e <
ENEMY_THRESH)) or (isEnemy == True and num_h < HUM_THRESH):
        if ~isEnemy:
            cfg.MAX_DEPTH = 7
        if num_u == cfg.MAX_GROUPS:
            possibleMoves = move_2_groups(us, enemy, human)
        else:
            possibleMoves = move_1_group(us, enemy, human)

    # if there are few humans and many enemies lower the depth
    elif (num_h < HUM_THRESH or possible == False or isEnemy == True) and
num_e >= ENEMY_THRESH:
        if ~isEnemy:
            cfg.MAX_DEPTH = 6

        if num_u == cfg.MAX_GROUPS:
            possibleMoves = move_2_groups(us, enemy, human)
        else:
            possibleMoves = move_1_group(us, enemy, human)


    # when many humans just consider humans and go deeper
    else:
        cfg.MAX_DEPTH = 7

        if num_u == cfg.MAX_GROUPS:
            possibleMoves = move_2_groups_start(us, human)
        else:
            possibleMoves = move_1_group_start(us, human)


    possibleMoves = list(possibleMoves)
    random.shuffle(possibleMoves)

    return possibleMoves
```

- **Minimax**
  Finalement, le fichier **minimax** regroupe le minimax avec alpha beta pruning.

5. Optimisation

Bien qu'il serait possible de rechercher tous les mouvements possibles nous serions rapidement bloqués par la profondeur. De la même manière qu'un humain jouerait, c'est-à-dire en évaluant les conséquences d'un nombre limité de coups parmi l'ensemble des coups possibles. La décision de l'IA ne sera pas parfaite au regard de tous les coups possibles, mais c'est un dilemme entre performance et rapidité. Ce choix est donc assumé.

Un autre équilibre à trouver est celui entre la précision et la rapidité. Notre IA doit évaluer avec précision les positions qui lui donne le plus de chance de gagner et plus ce calcul est précis plus le coup joué aura de chance d'être le bon. Cependant ce calcul se fait au détriment de la vitesse de notre IA. L'alphabeta est en ce sens d'une redoutable efficacité.

Zobrist hashing : lors du développement de l'algorithme, une priorité a été de développer rapidement une IA fonctionnelle puis d'itérer dessus. La problématique liée au temps de calcul a été prédominante dans nos choix de développement. Une solution pour résoudre ce problème aurait été de mettre en place des tables de transposition qui enregistrent les évaluations des branches de l'arbre à chaque fois qu'elles sont calculées. Cela permet d'éviter de recalculer ces résultats à chaque fois que notre IA se retrouve dans une situation qu'elle a déjà vu. Cependant nous n'avons pas eu le temps de mettre cette solution en place. En outre elle présente des limites : être à la position (1,1) n'aura pas toujours le même score et donc la rapidité aurait été au détriment d'une compréhension de l'état actuel.

6. Limites

Il y a quelques limitations majeures à notre arbre de jeu.

1. Temps

Comme mentionné ci-dessus, les arbres de jeu sont rarement utilisés dans des scénarios en temps réel (lorsque l'ordinateur n'a pas beaucoup de temps pour réfléchir). La raison en est que la méthode nécessite beaucoup de traitement par l'ordinateur, et cela prend du temps. Cette méthode est performante que pour les jeux à tour par tour.

Ainsi, nous atteignons un maximum de 7 de profondeur. Une méthode pour améliorer cela aurait été de mettre en place un timer pour couper arrêter d'aller en profondeur lorsque nous n'avons plus de temps (C'est ce qui a plus ou moins été fait de manière manuelle avec nos règles de jeu). En outre, le multiprocessing aurait également permis d'évaluer des actions sur différents cœur et ainsi augmenter la profondeur possible.

2. Nécessite une connaissance complète de la façon de se déplacer.

Les jeux avec incertitude ne font généralement pas bon ménage avec les arbres. Ils peuvent être mis en œuvre dans de tels jeux, mais ce sera très difficile et les résultats pourraient s'avérer moins que satisfaisants.

3. Inefficaces pour déterminer avec précision les meilleurs choix dans des scénarios avec de nombreux choix possibles.