Jean Barreto - 2311468

Agent intelligent pour le jeu Abalone

INF8175 – Intelligence artificielle : méthodes et algorithmes
Travaille présenté à Cappart quentin

Titre du projet : ABALONE
Nom d'équipe Challonge: BARRETO

Génie informatique et logiciel
École Polytechnique Montréal
7 décembre 2023

Introduction :

Ce rapport à pour but d'éclairer le déroulement du projet donné dans le cadre du cours INF8175 - Intelligence artificielle : méthodes et algorithmes. Ce projet consistait à réaliser un agent intelligent capable de jouer au jeu Abalone avec n'importe quelle configuration de départ.

Abalone est un jeu de plateau au tour par tour à 2 joueurs, déterministe et à information totale à somme nulle. Le but est de pousser les billes de l'adversaire hors du plateau (cf fig. 1). Avec des règles simples, mais une profondeur tactique surprenante, chaque mouvement compte. Sur un plateau hexagonal, les joueurs forment des groupes de billes pour renforcer leur position tout en cherchant à affaiblir celle de l'adversaire.

fig. 1 : plateau de jeu Abalone dans une configuration de départ classique

Nous allons voir au cours de ce rapport les différents choix de conceptions et d'améliorations qui ont été fait sur la base du cours afin de rendre un agent le plus compétent possible.

Agent de base :

1. Minimax

Nous somme partie sur l'idée d'un agent de base à l'architecture très simple afin de pouvoir l'améliorer en continue vers une version de plus en plus compétitive. En ce sens, il était évident pour nous de partir sur un algorithme de type minimax qui à la particularité d'être une bonne base pour des améliorations futures.

Le projet nous fournissant déjà bon nombre de fonctions auxiliaires (coups possible, transition d'état, etc…) Il a été aisé d'implémenter le pseudo code du minimax suivant pour notre agent de base :

```
maxValue(s) :
  if isTerminal(s) :
    return ⟨ utility(s, maxPlayer), ⊥ ⟩
  v* = −∞
  m* = ⊥
  for each a ∈ actions(s)
    s' = transition(s, a)
    ⟨v, _⟩ = minValue(s')
    if v > v* :
      v* = v
      m* = a
  return ⟨v*, m*⟩
```

```
minimaxSearch(s0) :
  ⟨v, m⟩ = maxValue(s0)
  return ⟨v, m⟩
```

```
minValue(s) :
  if isTerminal(s) :
    return ⟨ utility(s, maxPlayer), ⊥ ⟩
  v* = ∞
  m* = ⊥
  for each a ∈ actions(s)
    s' = transition(s, a)
    ⟨v, _⟩ = maxValue(s')
    if v < v* :
      v* = v
      m* = a
  return ⟨v*, m*⟩
```

Fig. 2 : pseudo code de l"algorithme minimax

On peut y lire les fonctions :

- isTerminal(s) : renvoie True si s est un état où l'un des joueurs gagne (6 points atteint) ou bien lorsqu'il ne reste plus de coup à jouer (limite de 50 coups) [déjà implémenté]
- transition (s,a) : renvoie l'état d'arrivée après exécution de l'action a sur l'état s [déjà implémenté]
- utility(s, maxPlayer) : renvoie l'utilité d'un état s pour le joueur maxPlayer[pas encore implémenté]

Cet algorithme est cependant inutilisable en l'état. En effet, on attend d'arriver au bout de l'arbre afin de renvoyer la valeur de l'état final. Au vu de la profondeur et de la largeur de l'arbre des états d'Abalone (jusqu'à 148 actions possible pour un état) ceci prendrait beaucoup trop de temps et n'est donc pas implémentable dans les conditions du projet

2. Heuristique

Nous allons donc implémenter une fonction heuristique qui se chargera d'estimer l'utilité d'un état quelconque pour un joueur afin de pouvoir arrêter la recherche minimax quand on le voudra.

Étant donné que l'on cherche ici à produire rapidement un agent de base capable d'assurer une partie du début à la fin, on se contentera d'une heuristique assez naïve. Puisque le gagnant est le premier arrivé à 6 points on utilisera le score comme première mesure d'heuristique et comme l'on a moins de chance de perdre nos billes si elles sont toutes aux centres ce sera notre deuxième mesure de l'heuristique. On a alors une heuristique simple définit comme suit :

$$h(s) = 0*h1(s) +*h2(s)$$

- h1(s) renvoie la différence de score entre les noirs et les blancs
- h2(s) renvoie la différence de valeurs des billes noires et blanches selon les poids des grilles de la fig. 3

Remarque : La fonction heuristique doit normalement être en tout tant comprise entre la valeur minimale et maximale de la fonction d'utilité. Dans toute la suite du projet on prend la fonction d'utilité comme étant égale à l'heuristique afin de respecter ce critère.

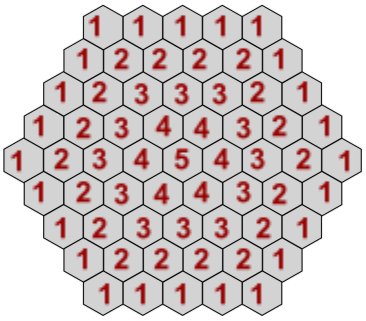

Fig. 3 : Répartition de la valeurs des cases

3. Condition d'arrêt

Il ne nous manque plus qu'à déterminer une condition d'arrêt de l'exploration des états. Autrement dit, un critère à partir duquel on se contente de renvoyer l'estimation de notre heuristique pour évaluer un état s.

Chaque joueur ayant un budget temps de 15min pour au maximum 25 coups l'approche naïve sera donc de diviser ce budget temps au coup, c'est à dire 36 secondes par coup. Ainsi au bout de 30 secondes notre algorithme minimax arrêtera de descendre dans l'arbre des états et commencera à renvoyer les valeurs. Une marge de 6 secondes est prise afin de compter le temps de remonter à travers tous les appels récursifs.

Nous voici donc avec un algorithme de base que l'on appellera V0 dans toute la suite du rapport.

Première améliorations :

1. Alpha Beta

Une façon simple d'améliorer considérablement notre algorithme minimax est sa variante alpha beta. Pour se faire une simple variation du code est suffisante pour éviter de descendre dans des branches stérile de l'arbre. Voici le pseudo code implémenté :

```
maxValue(s, α, β) :
  if isTerminal(s) :
    return ⟨utility(s, maxPlayer), ⊥⟩
  v* = −∞
  m* = ⊥
  for each a ∈ actions(s)
    s' = transition(s, a)
    ⟨v, _⟩ = minValue(s', α, β)
    if v > v* :
      v* = v
      m* = a
      α = max(α, v*)
    if v* ≥ β : return ⟨v*, m*⟩
  return ⟨v*, m*⟩
```

```
alphaBetaSearch(s0) :        α   β
  ⟨v, m⟩ = maxValue(s0, −∞, +∞)
  return ⟨v, m⟩
```

```
minValue(s, α, β) :
  if isTerminal(s) :
    return ⟨utility(s, maxPlayer), ⊥⟩
  v* = ∞
  m* = ⊥
  for each a ∈ actions(s)
    s' = transition(s, a)
    ⟨v, _⟩ = maxValue(s', α, β)
    if v < v* :
      v* = v
      m* = a
      β = min(β, v*)
    if v* ≤ α : return ⟨v*, m*⟩
  return ⟨v*, m*⟩
```

Fig. 4 : Pseudo code de l'alpha beta

2. Profondeur variable

Une seconde façon plus subtile d'améliorer les performances de notre agent est de se dire que les coups de début et de fin de partie sont beaucoup plus évidents et moins complexes que ceux en milieu de partie. Ainsi on donnera progressivement plus de ressources à notre agent au fil de la partie avant de les diminuer à nouveau jusqu'à la fin.

Pour se faire on utilisera une table de profondeur, indiquant à notre agent jusqu'à quelle profondeur de recherche se rendre selon l'avancement de la partie. La table ed profondeur implémentée est la suivante :

T = [1,2,2,2,3,3,3,3,3,3,4,4,4,4,4,3,3,3,3,3,2,2,2,1]

Ces premières améliorations seront désignées sous le nom d'agent V1 jusqu'à la fin du rapport.

Seconde améliorations :

Après s'être occupé de l'architecture même de l'agent dans la partie précédente, celle-ci s'attellera à améliorer au mieux notre heuristique h(s). Pour se faire nous allons combiner linéairement les heuristiques suivantes :

1. Le score

Notée h1(s) c'est tout simplement la différence de score entre le joueur noir et blanc.
En effet, il est avantageux d'avoir un score plus élevé que son adversaire sachant que le premier à 6 reporte la partie.

2. La distance au centre des billes

Notée h2(s) c'est la différence de valeurs des billes noires et blanches selon les poids des grilles de la fig. 3 récompensant les billes proches du centre.
En effet, les billes aux centre sont plus à même de ne pas se faire éjecter facilement du terrain.

3. La distance entre chaque billes

Notée h3(s) c'est la somme de la distance entre chaque paire de bille blanche moins la somme de la distance entre chaque paire de billes noires. La distance entre 2 billes est le nombre de coups qu'il faut pour emmener l'une sur la case de l'autre.
En effet, pour limiter les occasions de notre adversaire à pousser l'une de nos billes, il est préférable d'avoir des billes groupées au maximum entre elles.

4. Le bord du terrain

Notée h4(s) c'est la somme des billes blanches étant sur la couronne extérieur du plateau moins la somme des billes noires étant sur la couronne extérieur du plateau.
En effet, le bord du terrain est une zone dangereuse par définition car l'on peut y perdre une bille en un seul mouvement de l'adversaire.

5. Le centre du terrain

Notée h5(s) c'est la somme des billes noires étant située à une case ou moins du centre du plateau moins la somme des billes blanches étant située à une case ou moins du centre du plateau.
En effet, comme dit plus tôt avoir des billes occupant le centre du terrain est une bonne stratégie défensive, on compte ici des "points" bonus pour les billes dans la première couronne entourant le centre.

6. Attribution des poids

L'heuristique finale sera donc de la forme suivante :

$$h(s) = a*h1(s) + b*h2(s) + c*h3(s) + d*h4(s) + e*h5(s)$$

Il faut donc attribuer des poids à chacunes de ces heuristiques (i.e. valeurs de a, b, c, d et e). Pour se faire, on adopte une recherche locale comme suit : on adopte des poids arbitraires entre 0 et 1, et à chaque itération on fait affronter cet agent contre une version avec des poids légèrement modifiés aléatoirement. On garde le meilleur et on recommence jusqu'à un critères limites (nombres d'itérations)

Ces secondes améliorations seront désignées sous le nom d'agent V2 jusqu'à la fin du rapport.

Resultats :

Les pourcentages de victoires sur 20 matchs entre nos différents agents ont été reporté ci-dessous :

Agent V0	Agent V1	Agent V2
Classique	40%	0%
Alien	55%	0%

Agent V1	Agent V0	Agent V2
Classique	60%	15%
Alien	45%	20%

Agent V2	Agent V0	Agent V1
Classique	100%	85%
Alien	100%	80%

Fig. 5 : Pourcentages de victoires inter-agents

On observe bien une amélioration significative de nos agents au fur et à mesure du projet.

Discussion :

De nombreuses améliorations sont encore possible, on peut en lister quelques unes :

- La profondeur de recherche peut être mieux adaptée au temps restant que l'on dispose. Actuellement notre Agent V2 dispose d'un temps encore non négligeable à la fin d'une partie.
- On peut se prémunir de l'effet d'horizon dans la recherche en creusant plus loin lorsqu'on rencontre un état ambiguë (p.e. Possibilité de perdre une bille au prochain coup)
- On peut optimiser l'alpha beta en implémentant un tri des actions possible à chaque état en les classant des plus prometteuses au moins prometteuses
- L'alpha bêta peut également être optimisée en ajoutant des tables de hachage afin de ne pas faire de travail redondant.
- On aurait pu parcourir l'arbre plus profondément en abandonnant les branches qui semblent être peu prometteuse
- On peut rajouter et/ou affiner des heuristiques élémentaire
- On peut considérablement améliorer l'attribution de nos poids, en effet à cause du temps d'exécution et du manque d'automatisation notre recherche locale des meilleurs poids n'a été lancée que sur un seul départ aléatoire avec 10 itérations améliorantes. Nous sommes donc vraisemblablement très loin des poids optimaux.

References :

- L'intégralité de l'excellent court de Cappart Quentin : INF8175 - Intelligence artificielle : méthodes et algorithmes
- La documentation de la bibliothèque SeaHorse : https://corail-research.github.io/seahorse/