Algorithme minimax

ITC PC

M. Charles

02/12/2024

Introduction

Exercice

On rappelle les règles d'un jeu de Nim :

- il y a 21 allumettes ;
- deux joueurs s'affrontent, retirant chacun leur tour entre 1 et 4 allumettes ;
- celui qui prend la dernière allumette a perdu.

Questions :

1. Quel est l'ensemble des positions du jeu ?
2. Quelle·s position·s est/sont perdante·s ?
3. Construire l'ensemble G_1 des positions gagnantes en 1 demi-coup. Pour chacune de ces positions, quel coup permet au joueur de gagner ?
4. Construire les ensembles P_1 et P_2 des positions perdantes en 1 et 2 demi-coups.

Exercice

5. Écrire une fonction Python `coup(n: int) -> int` qui prend en paramètre le nombre d'allumettes restantes et renvoyant un nombre d'allumettes à retirer.

   Si la position est gagnante, `coup` doit renvoyer un (le ?) coup permettant de placer le joueur suivant en position perdante.

   Si la position est perdante, `coup` doit renvoyer 1.

Exercice

5. Écrire une fonction Python `coup(n: int) -> int` qui prend en paramètre le nombre d'allumettes restantes et renvoyant un nombre d'allumettes à retirer.

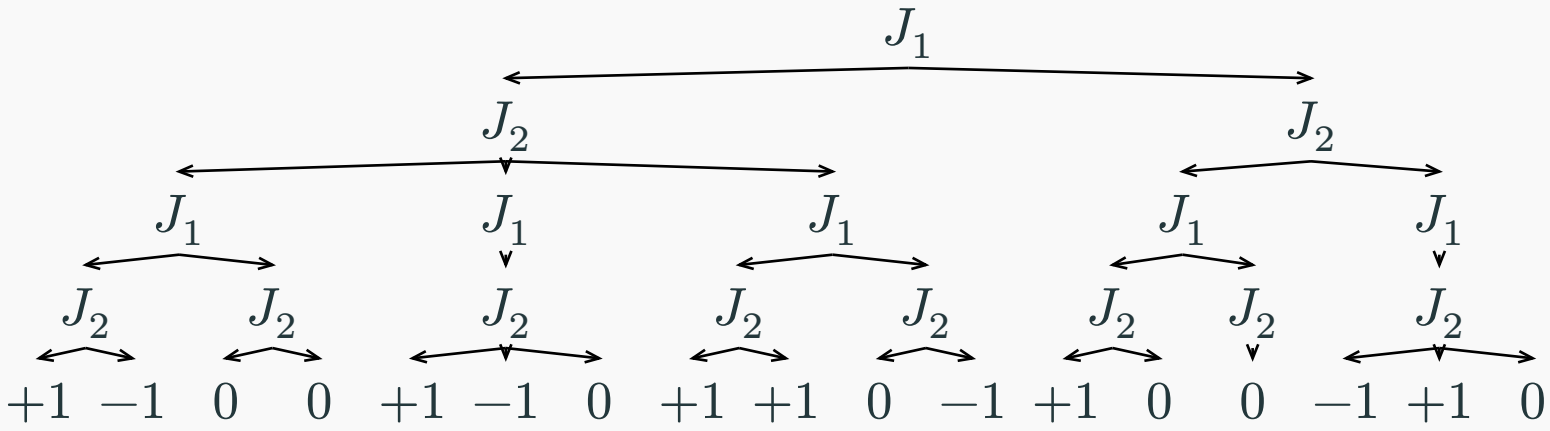
   Si la position est gagnante, `coup` doit renvoyer un (le ?) coup permettant de placer le joueur suivant en position perdante.

   Si la position est perdante, `coup` doit renvoyer 1.

```python
def coup(n: int) -> int:
    if (n-1) % 5 == 0:
        return 1
    else:
        return (n-1) % 5
```

Fonctions d’évaluation

Rappel : construction naïve des attracteurs



$+1$: gagnant pour J_1, 0 : nul, -1 : gagnant pour J_2.

Fonctions d'évaluation

Une **fonction d'évaluation** (statique) est une fonction qui

- prend en argument une position, et qui
- renvoie une valeur « ordonnable » censée donner une idée fiable des chances de gain du joueur ayant le trait.

Construction de fonctions d'évaluation

Cas des échecs :

- Matériel
- Éléments positionnels : sécurité du roi, activité des pièces, structure de pions…
- Moyennes de fonctions d'évaluation plus simples
- Les systèmes utilisés dans la vie réelle sont un peu plus compliqués : https ://github.com/official-stockfish/Stockfish/blob/master/src/evaluate.cpp

Cas du morpion :

- -1, 0 ou $+1$
- $(3\,X_2 + X_1) - (3\,O_2 + O_1)$ [1]

Difficulté : avoir une fonction d'évaluation qui ordonne correctement les positions.

[1] Russell et Norvig, *Artificial intelligence - a modern approach*

Minimax, première version

Idée de départ

On cherche à jouer un coup qui va (de manière équivalente)

- amener l'adversaire dans une position qui soit la pire pour lui.
- **minimiser les dégâts** pouvant être infligés par l'adversaire.
- maximiser notre propre gain, sachant que l'adversaire *peut* chercher à le minimiser.

Remarques historiques

- Lien avec le **théorème du minimax** :
  - Von Neumann c. 1928
  - Von Neumann-Morgenstern 1944, *The Theory of Games and Economic Behavior*
- Lien avec les **équilibres de Nash** :
  - Nash 1950, *Equilibrium points in n-person games*
  - Nash 1951, *Non-cooperative games*

Point de vue minimax, point de vue négamax

Minimax :

- fonction d'évaluation positive pour les positions gagnantes pour J_1, négative pour les positions gagnantes pour J_2 (ex. : Stockfish)
- J_1 cherche à maximiser cette fonction, J_2 cherche à la minimiser
- dans l'implémentation, il faut faire une disjonction de cas

Négamax :

- fonction d'évaluation positive pour les positions gagnantes pour le joueur ayant le trait
- chaque joueur cherche à maximiser cette fonction
- une position positive pour un joueur est négative pour l'adversaire
- dans l'implémentation, pas de disjonction de cas mais un changement de signe

Pseudocode pour le négamax

```
def évaluation(p) -> int:
    if p.est_finale():
        match p.statut():
            case "gagnée": return +1
            case "nulle":  return 0
            case "perdue": return -1
    else:
        évals_suivantes = []
        for coup in p.coups_légaux():
            e = évaluation(p.après(coup))
            évals_suivantes.append(e)
        return max(-e for e in évals_suivantes)
```

Pseudocode pour le négamax

```
def évaluation(p) -> int:
    if p.est_finale():
        match p.statut():
            case "gagnée": return +1
            case "nulle":  return 0
            case "perdue": return -1
    else:
        évals_suivantes = []
        for coup in p.coups_légaux():
            e = évaluation(p.après(coup))
            évals_suivantes.append(e)
        return max(-e for e in évals_suivantes)
```

Remarques :

- C'est un algorithme de parcours en profondeur !
- Quelle technique pourrait permettre d'optimiser l'algorithme ?

Version optimisée avec mémoïsation

```
def évaluation(p, d={}) -> int:
    if p in d:
        return d[p]
    elif p.est_finale():
        ... # comme précédemment
    else :
        évals_suivantes = []
        for coup in p.coups_légaux():
            e = évaluation(p.après(coup))
            évals_suivantes.append(e)
        d[p] = max(-e for e in évals_suivantes)
        return d[p]
```

Version augmentée avec meilleurs coups

```
def évaluation(p) -> (int, list):
    if p.est_finale():
        ... # comme précédemment
    else:
        suivants = {} # dictionnaire de la forme {coup: éval}
        for coup in p.coups_légaux():
            e, _ = évaluation(p.après(coup))
            suivants[coup] = e
        opt = max(-e for e in suivants.values())
        coups =  [c for c, e in suivants.items()
                    if -e == opt]
        return opt, coups
```

Exercice

Écrire la fonction d'évaluation minimax pour le jeu de Nim de début de cours.

On suppose que la position `p` est donnée par un entier entre 0 et 21. Il faudra également écrire les fonctions `est_finale(p: int) -> bool` et `coups_légaux(p: int) -> list`.

Exercice

Écrire la fonction d'évaluation minimax pour le jeu de Nim de début de cours.

On suppose que la position `p` est donnée par un entier entre 0 et 21. Il faudra également écrire les fonctions `est_finale(p: int) -> bool` et `coups_légaux(p: int) -> list`.

```
def est_finale(p: int) -> bool:
    return p == 0

def coups_légaux(p :int) -> list:
    return list(range(1, min(4,p)))

def evaluation(p) -> (int, list):
    if est_finale(p):
        return 1
    else:
        suivants = {} # dictionnaire de la forme {coup: éval}
        for coup in coups_légaux(p):
            e, _ = evaluation(p - coup)
            suivants[coup] = e
        opt = max(-e for e in suivants.values())
        coups =  [c for c, e in suivants.items() if -e == opt]
        return opt, coups
```

Inconvénient de l'approche précédente ?

Combien de positions différentes pour le morpion ? pour le jeu d'échecs ?

Inconvénient de l'approche précédente ?

Combien de positions différentes pour le morpion ? pour le jeu d'échecs ?

Jeu	Morpion	Échecs	Go
Nombre de coups par position	~ 4	~ 31 à 35	~ 250
Longueur d'une partie	~ 9	~ 70	~ 150
Taille de l'arbre	$\sim 4^9 \sim 250\,\mathrm{k}$	$\sim 31^{70}$	$\sim 250^{150}$

NB : le nombre de coups par position est plutôt appelé *degré*, ou *facteur de branchement*.

Minimax, profondeur bornée

Idée pour pallier le problème

Deux fonctions d'évaluation :

1. une fonction dite **heuristique**, donnant une approximation de la qualité d'une position donnée, de manière statique, sans calculer de coups ;
2. une fonction récursive, qui
   - **calcule** toutes les variantes jusqu'à une **profondeur** d fixée à l'avance,
   - avec le principe du minimax, et
   - se rabat sur l'heuristique pour les positions en profondeur exactement d.

Pseudocode pour le négamax tronqué

```
def heuristique(p) -> int:
    if p.est_finale():
        ... # comme précédemment
    else :
        ... # on se débrouille !

def évaluation(p, d:int) -> int:
    if d == 0:
        return heuristique(p)
    else:
        évals_suivantes = [évaluation(p.après(coup), d-1)
                           for coup in p.coups_légaux()]
        return max(-e for e in évals_suivantes)
```

Limites

- Horizon effect

  Exemple : perte d'une dame retardée par un sacrifice de tour

- Effet de « brouillard de guerre » (*fog of war*)

Exercice

En s'inspirant de la fonction d'évaluation $(3\,X_2 + X_1) - (3\,O_2 + O_1)$ pour le morpion, proposer une fonction heuristique pour le puissance 4.