LE GUENNEC Yaakoub
BRAGAGNOLO Chia-ling Angela

Rapport TP2 Hexapawn

Master Informatique - E-Services

Introduction

Le TP Hexapawn a pour but de comparer une version naïve d'un algorithme qui est amené à calculer plusieurs fois une même occurence d'une donnée et une version dynamique qui enregistre ses résultats au fur et à mesure des calculs.

Un configuration de jeu est ici représentée par le plateau de jeu et et le joueur courant.

La première version est appelé version naive car c'est la plus simple à mettre en place mais son défaut est qu'elle calcule plusieurs fois des valeurs pour des mêmes configurations.
La version dynamique possède de la mémoire où elle stocke les configurations et leurs valeurs déjà calculé au fur et à mesure
qu'elle s'appelle récursivement.

Structure du projet

Le projet est divisé en 3 fichiers :

versionNaive.py : Contient la version naïve de l'algorithme de calcule de la valeur de configuration.
versionDynamique.py : Contient la version dynamique de l'algorithme de calcule de la valeur de configuration.
utilitaire.py : Contient les fonctions de gestion de plateau et la logique du jeu.

Choix de structure de données

Pour représenter une configuration de jeu :

configuration : Liste composée du plateau de jeu et du joueur courant [board, player]
plateau de jeu : Liste contenant toutes les lignes du plateau [Line1, Line2, Line3...]
ligne du plateau : Liste des cases
case : Chaine de caractère contenant la représentation d'un pion ou d'une case vide.

Nous avons choisit d'utiliser une liste pour représenter une configuration, un plateau de jeu et une ligne de plateau, car on va beaucoup utiliser les opérations de parcours "dans l'ordre" (là où les listes excellent ; opération en *O(1)*).

Gestion du plateau de jeu

Le plateau de jeu étant une liste de liste il est aisé de se le représenter.

Exemple d'un plateau *3x4*:

```
[  ['p', 'p', ' ', 'p']
   [' ', 'p', ' ', ' ']
   ['P', ' ', 'P', 'P']  ]
```

La seule opération utile sur le plateau est, à partir d'une configuration (plateau et joueur), nous renvoyer toutes les configurations (du plateau) possibles en un coup.

Dans le module utilitaire.py, la fonction *getSuccessors(config)* s'occupe de nous renvoyer la liste de toutes les configurations successeurs de `config`.

Logique de la fonction :

On parcours les lignes du plateau à la recherche des cases où se trouve un pion du joueur courant et une fois qu'on a trouvé une case occupée par un pion du joueur,

on teste si on peut avancer en avant d'un pas (si la case est vide) ou si on peut capturer en diagonal un pion adverse.

Et à chaque déplacement de pion possible, on enregistre la nouvelle configuration à la liste des successeurs à renvoyer.

Selon les règles du jeu, si on ne peut plus se déplacer ou qu'un pion adverse se trouve dans notre ligne, on a perdu.
Le premiers cas est simple il n'y a rien à faire, si on ne trouve pas de coup possible, la liste renvoyée sera vide et on traite cette information plus tard dans notre algorithme.

Ensuite, si un pion se trouve dans la zone adverse, on renvoie une liste vide de successeurs, là aussi il faudra traiter cette information dans notre implémentation de l'algorithme.

Algorithme version naïve

Étapes de la logique de l'algorithme :

1. On récupère une configuration ***[board, player]***
2. On récupère les configurations successeurs de notre configuration courant
3. On vérifie qu'il y a des successeurs:
   Si il y en a pas, ça signifie que ne peut déplacer aucun pion, donc on a perdu; on renvoie donc la **valeur 0.**
   Sinon on continue
4. Pour chaque successeur, récupère et enregistre sa valeur en appelant récursivement la fonction .
5. On discrimine les valeurs (l'ordre est important) : Si 0 présent dans les valeurs des successeurs, ça veut dire que l'autre joueur perdra dans un coup et donc que l'on gagne en un coup, ainsi on renvoie la valeur +1.

Si toutes les valeurs sont positives, ça veut dire que l'on va perdre quoi qu'on fasse dans k coups.
On choisit donc la valeur la plus élevé pour retarder la défaite.
Sinon, on choisit dans les valeurs négatives, la plus petite valeur (en valeur absolue), pour gagner avec le moins de coups.
6. On renvoie la valeur de la configuration.

Implémentation en Python version naïve

Notre implémentation suit les étapes de la logique de l'algorithme, grâce aux opérations sur les liste de Python, le code est assez court.

1. On récupere la config : *calculateConfigValue_naiveVersion(config)*
2. On récupère les successeurs: *successors = list(getSuccessors(config))*
3. On vérifie qu'on peut jouer : *if len(successors) == 0: return 0*
4. On calcule les valeurs des successeurs:

```
for successor in successors :
        value = calculateConfigValue_naiveVersion(successor)
        values.append( value )
```

5. On discrimine les valeurs des successeurs et on renvoie la meilleure valeur:

```
if 0 in values:
        return 1
elif allPositive(values):
        return -(max(values)+1)
else :
        return -( max(extractNegativeValues(values)) -1 )
```

Algorithme version dynamique

Étapes de la logique de l'algorithme :

1. On récupère une configuration [board, player]

2. On vérifie que l'on a pas déjà calculé et enregistré cette configuration

Si oui, on renvoie la valeur associé à cette configuration

Si non, on continue

3. On récupère les configurations successeurs de notre configuration courant

4. On vérifie qu'il y a des successeurs:

Si il y en a pas, ça signifie que ne peut déplacer aucun pion, donc on a perdu; on renvoie donc la valeur 0

Sinon on continue

5. Pour chaque successeur, récupère et enregistre sa valeur en appelant récursivement la fonction .

6. On discrimine les valeurs :

Si 0 présent dans les valeurs des successeurs, ça veut dire que l'autre joueur perdra dans un coup et donc que l'on gagne dans un coup, ainsi la valeur à retourner sera la valeur +1.

Si toutes les valeurs sont positives, ça veut dire que l'on va perdre quoi qu'on fasse dans k coups.

On retournera donc la valeur la élevé pour retarder la défaite.

Sinon, on renverra dans les valeurs négatives, la plus petite valeur (en valeur absolue), pour gagner avec le moins de coups.

7. On enregistre la configuration avec sa valeur associée.

8. On renvoie la valeur de la configuration.

Implémentation en Python version dynamique

1. On récupère la config : *calculateConfigValue_naiveDynamique(config)*

2. On "hash" la configuration et on vérifie qu'on l'a pas déjà enregistré dans notre dictionnaire.

Si oui, on renvoie bien sa valeur

```
strBoard = ""
        for line in config[0]:
```

```
                strBoard += ''.join(line)
                strPlayer = config[1]
                strConfig = strBoard+strPlayer

            if strConfig in memorizedConfigs:
                return memorizedConfigs[strConfig]
```

3. Sinon on récupère les successeurs: *successors = list(getSuccessors(config))*
4. On vérifie qu'on peut jouer : *if len(successors) == 0: return 0*
5. On calcule les valeurs des successeurs:

```
    for successor in successors :
                value = calculateConfigValue_naiveVersion(successor)
                values.append( value )
```

6. On discrimine les valeurs des successeurs et on renvoie la meilleure valeur:

```
    if 0 in values:
                return 1
            elif allPositive(values):
                return -(max(values)+1)
            else :
                return -(  max(extractNegativeValues(values))  -1 )
```

7. On mémorise sa configuration et sa valeur :

```
    memorizedConfigs[strConfig] = currentConfigValue
```

8. Enfin, on renvoie la valeur de la configuration :

```
    return currentConfigValue
```

Tests Contest

Résultats tests version naïve :

Résultats

Test 1	**Succès** (0.02s)
Test 2	**Succès** (0.02s)
Test 3	**Succès** (2.532s)
Test 4	**Echec** *TIMED_OUT (Time limit exceeded)*
Test 5	**Echec** *TIMED_OUT (Time limit exceeded)*
Test 6	**Echec** *TIMED_OUT (Time limit exceeded)*
Test 7	**Echec** *TIMED_OUT (Time limit exceeded)*
Test 8	**Succès** (0.424s)

Résultats tests version dynamique :

Résultats

Test 1	**Succès** (0.02s)
Test 2	**Succès** (0.02s)
Test 3	**Succès** (0.24s)
Test 4	**Succès** (0.956s)
Test 5	**Succès** (0.636s)
Test 6	**Echec** *TIMED_OUT (Time limit exceeded)*
Test 7	**Echec** *TIMED_OUT (Time limit exceeded)*
Test 8	**Succès** (0.092s)

Interprétation des tests

Pour les deux premiers tests, la différence n' est pas visible, on suppose qu'il n'y a pas assez de calcule (de tours possibles) pour qu'on voit une plus grande efficacité pour la version dynamique.

En revanche, la version dynamique passe les tests 4 et 5, ce que ne fait pas la version naïve (les calculs durent trop longtemps).

On voit l'efficacité de l'implémentation dynamique au test 8, où on a un temps de *0.424 secondes* pour la version naïve et *0.092 secondes* pour la version dynamique.

Ainsi, la version dynamique a été 4 fois plus rapide que la version naïve pour trouver la valeur du test 8.

Pour les deux tests qui ne sont pas passé par *timeout,* on suppose qu'il existe une meilleure représentation des données ou un meilleur traitement des opérations sur le plateau.

Conclusion

Quand on sait qu'on sera amené à calculer plusieurs fois une même occurrence d'une donnée, il est parfois préférable de sauver du temps de calcul par de la mémoire pour améliorer les performances d'un algorithme.