Résolution d'un jeu de morpion

vincent.thomas@loria.fr

June 26, 2013

1 Création du jeu de Morpion

Un jeu de morpion sera représenté par une liste contenant des caractères. Chaque position dans la liste désignera une case du plateau de jeu (cf figure 1(a)).

Les caractères utilisés seront '.' pour une case vide, 'X' pour une case occupée par le joueur 1 et 'O' pour une case occupée par le joueur 2.

Par exemple la liste ['.','X','.','O','.','.','.','.','X'] désigne le jeu présenté figure 1(b).

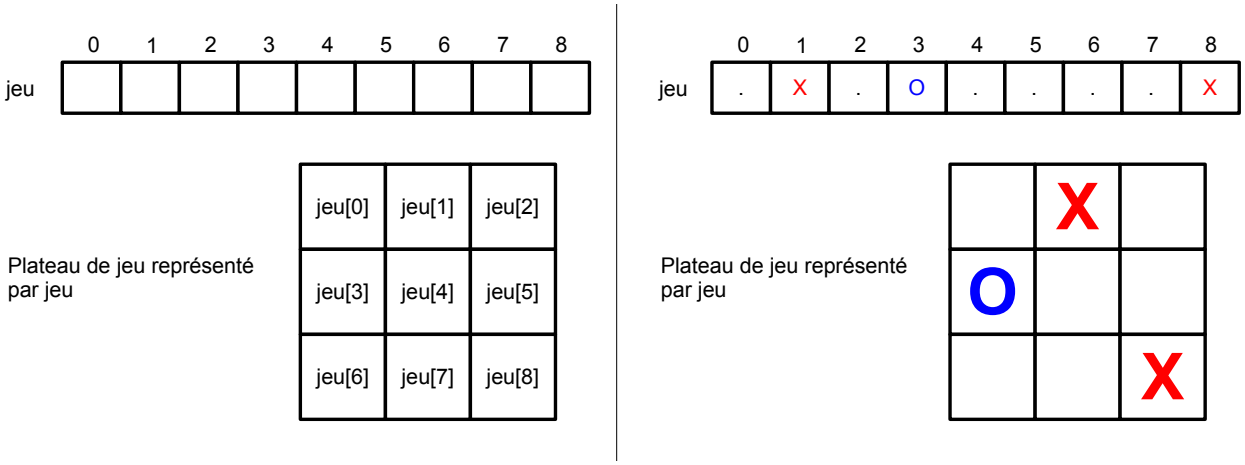


Figure 1: Représentation d'un plateau de jeu (a) plateau vide (b) plateau avec des pièces

1.1 Fonction nouveau jeu

Ecrire une fonction `nouveau()` qui retourne un jeu vide, c'est à dire une liste de 8 cases contenant '.' dans chaque case.

Par exemple

```
>>> coup=nouveauJeu()
>>> coup
('.', '.', '.', '.', '.', '.', '.', '.', '.')
>>> affiche(coup)
. . .
. . .
```

```
. . .
```

1.2 Fonction affiche

On vous donne la fonction affiche() capable d'afficher un jeu passé en paramètre.

```
#permet d'afficher un jeu
def affiche(jeu):
    print(jeu[0],jeu[1],jeu[2])
    print(jeu[3],jeu[4],jeu[5])
    print(jeu[6],jeu[7],jeu[8])

print("affiche jeu vide")
affiche(nouveauJeu())
```

1.3 Détection joueur

Ecrire une fonction `joueur(jeu)` qui permet de savoir qui doit jouer à partir d'un jeu passé en paramètre. Il suffit simplement de compter les 'X' et 'O' et de retourner 'X' ou 'O' en fonction de ces nombres.

Comme le joueur 'X' commence, si le nombre de 'X' est égal au nombre de 'O' c'est à lui de jouer. Si par contre, le nombre de 'X' est supérieur de 1 au nombre de 'O', c'est que c'est au second joueur de placer un pion.

Par exemple

```
>>> jeu=nouveauJeu()
>>> joueur(jeu)
'X'
>>> jeu=('.', 'X', '.', '.', '.', '.', '.', '.', '.')
>>> joueur(jeu)
'O'
```

1.4 Coups possible

Ecrire une fonction `coupPossibles(jeu)` qui retourne les coups possibles à partir d'un jeu passé en paramètre. Il s'agit de retourner la liste des indices des cases contenant le caractère '.'.

Par exemple

```
>>> jeu=nouveauJeu()
>>> coupPossible(jeu)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> jeu=('.', 'X', '.', '.', 'X', '.', '.', 'O', '.')
>>> coupPossible(jeu)
[0, 2, 3, 5, 6, 8]
```

1.5 Gagné/Perdu

La fonction suivante `gagner(jeu)` permet de savoir qui a gagné ou perdu. Cette fonction est écrite rapidement (on pourrait avoir une fonction plus générique en utilisant des boucles - par exemple pour le puissance 4 - mais ce n'est pas l'objectif de cet atelier).

Cette fonction retourne

- 1 si le joueur 1 'X' a gagné
- -1 si le joueur 2 'O' a gagné
- 0 si aucun joueur n'a gagné (qu'il reste des coups possibles ou non)

```
#permet de savoir si un joueur remporte la partie
def gagner(jeu):
    res='.'
    if (jeu[0]!='.' and jeu[0]==jeu[3]==jeu[6]):
        res=jeu[0]
    if (jeu[1]!='.' and jeu[1]==jeu[4]==jeu[7]):
        res=jeu[1]
    if (jeu[2]!='.' and jeu[2]==jeu[5]==jeu[8]):
        res=jeu[2]
    if (jeu[0]!='.' and jeu[0]==jeu[1]==jeu[2]):
        res=jeu[0]
    if (jeu[4]!='.' and jeu[4]==jeu[5]==jeu[3]):
        res=jeu[3]
    if (jeu[7]!='.' and jeu[7]==jeu[8]==jeu[6]):
        res=jeu[6]
    if (jeu[0]!='.' and jeu[0]==jeu[4]==jeu[8]):
        res=jeu[0]
    if (jeu[2]!='.' and jeu[2]==jeu[4]==jeu[6]):
        res=jeu[2]
    if (res=='O'):
        return(-1)
    else:
        if res=='X':
            return(1)
    return(0)
```

1.6 fonction Jouer

Ecrire la fonction `jouer(jeu,coup)` qui permet de jouer un coup et qui retourne le nouveau jeu.

Cette fonction prend en paramètre un jeu et un numéro de coup. Elle appelle la fonction joueur pour savoir quel est le joueur qui a joué et elle ajoute la bonne pièce à l'endroit joué sur **une copie** du jeu.

Par exemple

```
>>> jeu=nouveauJeu()
>>> jeu2=jouer(jeu,1)
>>> jeu
('.', '.', '.', '.', '.', '.', '.', '.', '.')
>>> jeu2
('.', 'X', '.', '.', '.', '.', '.', '.', '.')
>>> jeu2=jouer(jeu2,5)
>>> jeu2
('.', 'X', '.', '.', '.', 'O', '.', '.', '.')
>>> jeu2=jouer(jeu2,0)
>>> jeu2
('X', 'X', '.', '.', '.', 'O', '.', '.', '.')
```

1.7 Bilan

Grâce à ces fonctions, nous avons défini le jeu de morpion.

- `nouveauJeu` permet de créer une situation de jeu initiale
- `coupPossible` encode les regles du jeu qui donnent les coups possibles associés à une situation donnée
- `jouer` encode les règles du jeu qui traduisent l'évolution du jeu
- `gagner` encode les règles de victoire

Ce qui suit peut s'adapter à n'importe quel jeu sequentiel à deux joueurs (à condition de redéfinir ces fonctions). Cependant, l'approche qui va suivre peut demander trop de mémoire ou de temps pour aboutir (exemple jeu d'échec)

2 Estimation du nombre de situations d'un jeu de morpion

Une première étape consiste à avoir un ordre de grandeur du nombre de situations possibles. Calculer le nombre exact n'est a priori pas trivial.

Le nombre de situations peut s'estimer de nombreuses manières

- On pourrait considérer que chaque case possède une des trois valeurs '.', 'X' ou 'O', ce qui donne 3^9 situations possibles, mais certaines situations ne sont pas atteignables (comme les joueurs jouent successivement, il ne peut pas y avoir de plateau avec deux X et sans O)
- On pourrait raffiner en disant que comme les joueurs jouent successivement, le nombre de parties complètes possibles est de 9! (9 choix pour le premier coup, 8 pour le second, ...). Le nombre situations serait donc 9! + 9!/1 + 9!/2! + 9!/3! + ... (la sommes du nombre de situations obtenues après n coups) mais ce serait aussi surévaluer le nombre de situations car (1) certaines positions obtenues sont les mêmes (la suite de coups 1,2,3 et la suite 3,2,1 conduisent aux mêmes situations) et car (2) certaines situations ne sont jamais atteintes (quand un joueur a gagné, la partie s'arrête)

- De plus, il est aussi possible de prendre en compte les symétries pour réduire le nombre de situations stratégiquement différentes (au premier coup, il n'y a que trois situations réellement différentes: jouer dans un coin, jouer au centre d'une ligne ou jouer au milieu du plateau)

3 Algorithme min-max

Maintenant que nous avons défini un jeu, nous allons utiliser les fonctions précédentes pour essayer de le résoudre.

3.1 Principe

L'algorithme min-max utilise la récursivité pour estimer la valeur d'une situation. Le principe est le suivant:

- Le joueur dont c'est le tour (joueur1) va choisir le coup qui maximise son gain c'est à dire la valeur de sa situation future (puisqu'une valeur de 1 correspondant à sa victoire et de -1 à la victoire du joueur 2), pour cela, il va devoir estimer toutes les situations auxquelles il peut arriver en 1 coup.
- Pour estimer la valeur d'une situation atteignable en 1 coup, le joueur1 va considérer que son adversaire le joueur2 va choisir ensuite le coup qui améliore au mieux son gain et donc qui minimise le gain du joueur1.
- or pour évaluer ce minimum, le joueur2 va considérer que le joueur1 cherchera ensuite à maximiser son gain ...
- ...
- et ce jusqu'à atteindre une situation finale dont la valeur est donnée par la fonction gagner.

L'ensemble de l'arbre est développé jusqu'aux feuilles dont on sait calculer la valeur (situation gagnante, perdante ou égalité sans coup possible), puis l'évaluation permet de faire remonter ces valeurs en opérant des min et des max successifs.

3.2 Résolution MinMax

Résoudre selon l'algorithme minmax consiste à définir deux fonctions qui vont s'appeler mutuellement

- une fonction `valMax(jeu)` qui va maximiser les valeurs `valMin(jeuApres1Coup)` sur tous les coups possibles. Cette fonction est appelée lorsque le joueur 'X' veut choisir le meilleur coup (en maximisant le gain sachant que le joueur suivant va le minimiser).
- une fonction `valMin(jeu)` qui va minimiser les valeurs `valMax(jeuApres1Coup)` sur tous les coups possibles. Cette fonction est appelée lorsque le joueur 'O' veut choisir le meilleur coup (en minimisant le gain du joueur 1 qui cherchera ensuite à le maximiser).
- Lorsqu'elles sont appelées sur une situation finale, ces fonctions retournent le résultat directement et ne font pas le calcul (états terminaux).

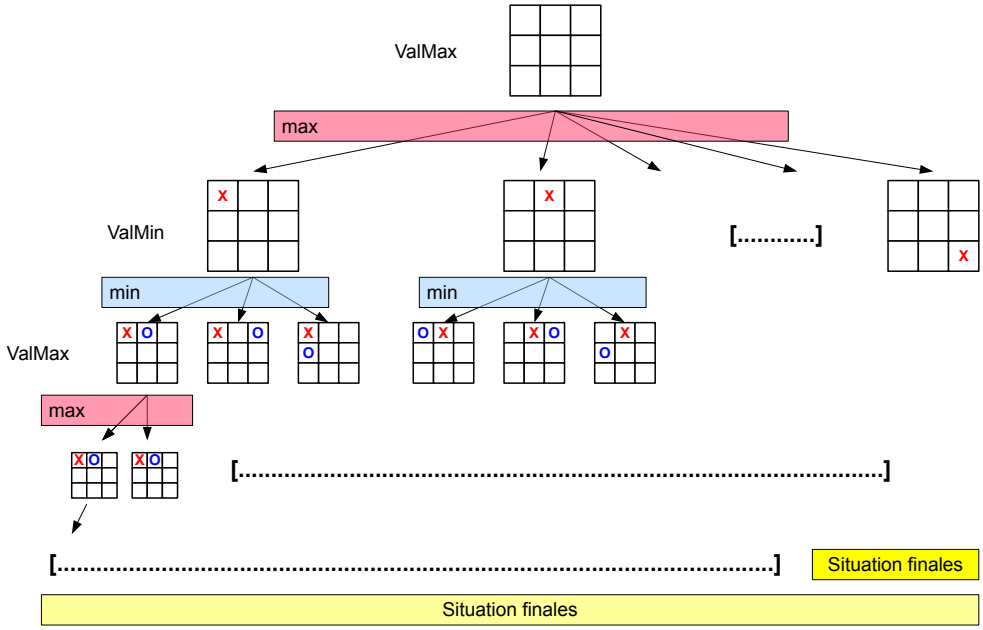


Figure 2: Algorithme MinMax - avec les appels successifs de valMin et valMax, l'algorithme parcourt l'arbre des situations possibles jusqu'aux situations finales qu'on sait évaluer

Ecrire l'algorithme minmax qui calcule la valeur d'une situation initiale. Ajouter en même temps que le calcul, le stockage des différentes valeurs dans un dictionnaire qui sera passé aux fonctions valMin et valMax. Ce dictionnaire contiendra en sortie d'appel les valeurs associées à toutes les situations possibles.

Cet algorithme est simple vu de l'extérieur mais il présente plusieurs inconvénients

1. il masque la manière dont le programme fonctionne puisque c'est l'appel récursif qui fait le parcours des situations possibles
2. il développe tout l'arbre (ce qui peut prendre beaucoup de temps) - dans le cas du morpion, il y a 5478 états différents à considérer
3. il repasse plusieurs fois par les mêmes états sans s'en rendre compte (on peut facilement améliorer les choses en stockant la valeur des états déjà visités)

3.3 Affichage des résultats

Une fois le dictionnaire construit, écrire une fonction qui affiche les résultats. Cette fonction prend en paramètre une situation de jeu, affiche tous les coups possibles et donne pour chaque coup, la valeur de la situation résultante.

Grâce à cet affichage, il est possible de savoir le meilleur coup à jouer (en fonction du joueur, c'est le coup qui maximise ou minimise la valeur de la situation résultante).

3.4 Résolution MinMax - parties plus longues

Quand un des joueurs sait qu'il a perdu, il peut écourter la partie et jouer n'importe quel coup. C'est ce que fait l'algorithme minmax puisque les coups perdants ont tous la même valeur de -1, l'algorithme va donc retourner le premier qu'il rencontre.

Quand l'algorithme joue contre un joueur et qu'il sait qu'il a perdu, il a intérêt à faire durer la partie, car le joueur peut être amené à faire une erreur. Inversement, le programme a intérêt a finir vite une partie, plutôt que de laisser espoir au joueur (ce qui permet au joueur de mieux estimer le coup qui l'a fait perdre).

Plutôt que de remonter la valeur, il est possible de remonter des fonctions valMin et valMax la valeur pondérée par un facteur multiplicatif $\gamma \in]0;1[$. Ainsi, le calcul du maximum va faire qu'on tendra à jouer les coups menant le plus rapidement à la victoire (plus la victoire est éloignée, moins elle semble intéressante, une victoire à n coup est pondérée par le facteur γ^n). De manière inverse, le min va faire qu'on tendra à jouer les coups menant le plus lentement à la défaite (plus la défaite est éloignée, moins elle semblera pénalisant, car une défaite a n coups aura une valeur de $-1 * \gamma^n$).

3.5 Extension aux jeux solitaires

Résoudre un jeu solitaire procède de la même démarche, sauf qu'il ne s'agit pas de deux joueur, l'un cherchant à nuire au second et inversement.

Désormais, il suffit simplement d'écrire une fonction valMax qui rappelle une fonction valmax: le joueur essaie de jouer son meilleur coup sachant qu'il essaiera ensuite de jouer son meilleur coup, ...

Cette approche fonctionne bien si le jeu ne présente pas de boucle (par exemple, dans le cas du jeu de solitaire, comme les billes disparaissent à chaque coup, on peut être sur de ne jamais revenir en arrière : le graphe est bien acyclique). Par contre, si le jeu possède une boucle, cette approche ne marche pas puisque si A dépend de B et B dépend de A, calculer A nécessite de calculer B qui nécessite de calculer A et les appels récursifs ne s'arrêtent jamais.

4 Approche par apprentissage

Désormais, le programme ne calcule pas tous les coups possibles en développant tout l'arbre des possibles, mais va évaluer au fur et à mesure de son expérience la valeur des coups qu'il a joués (sachant que certains coups restent inconnus).

Il utilisera ensuite cette évaluation pour estimer les valeurs des situations au fur et à mesure de l'exploration qu'il fera lors des parties.

4.1 Principe de l'approche

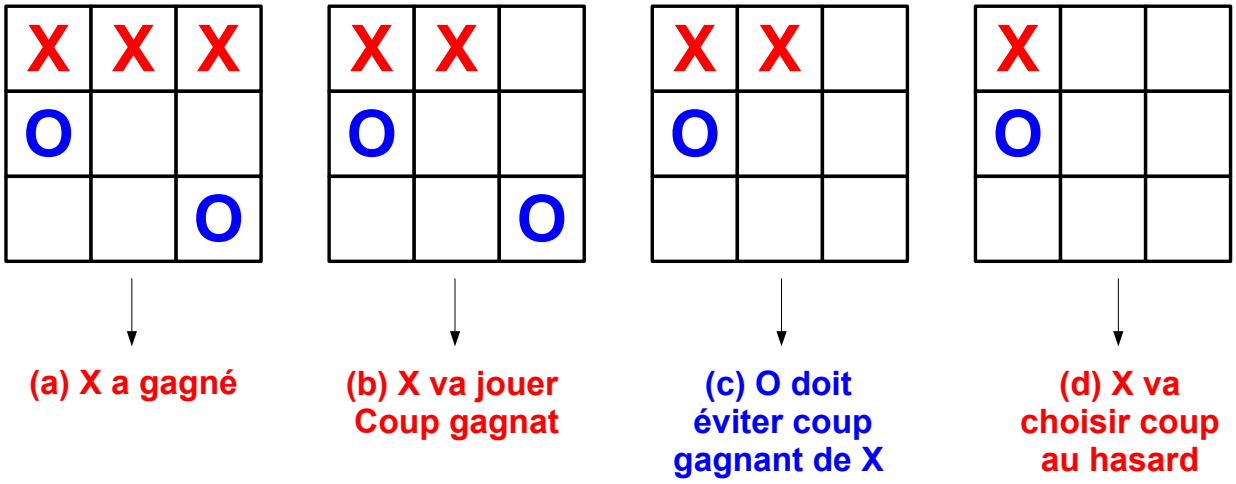
Pour résoudre le jeu, nous allons utiliser un dictionnaire qui associe à chaque situation de jeu l'état de cette situation qui peut être soit gagnante pour le joueur 1, soit gagnante pour le joueur 2 soit conduisant à une égalité (cf figure 4.1.1).

L'objectif va désormais consister à construire ce dictionnaire au fur et à mesure des experiences pour évaluer les situations possibles (jusqu'à la situation initiale).

4.1.1 Evaluation

Plusieurs cas peuvent apparaitre lors de l'évaluation d'une situation

- Si la situation de jeu est une situation finale (un joueur gagnant ou il n'existe plus de coup possible), la valeur associée à cette situation est simplement la valeur de la fonction gagner (situation a)
- Sinon, cette situation n'est pas une situation finale, il va falloir estimer sa valeur en fonction des situations auxquelles cette situation peut conduire:
  - Si il existe un coup gagnant[1] pour le joueur dont c'est le tour alors le joueur jouera ce coup et la situation peut être estimée gagnante pour ce joueur (situation b)
  - S'il n'existe pas de coup gagnant et que tous les coups mènent à une défaite (ou à une égalité), le joueur est dans une impasse et la situation est soit perdante (soit égalité, le moins pire des deux cas possibles)
  - sinon, c'est que certains coups n'ont pas encore été évalués et le joueur a tendance à évaluer ces coups pour estimer sa position. Il va donc avoir tendance à choisir un coup inconnu au hasard et à raisonner sur la situation suivante (situation d). Bien sur, si un coup est perdant pour un joueur il aura tendance à faire un autre coup pour essayer une autre stratégie (situation c)



[1] c'est-à-dire menant à une situation gagnante

4.1.2 Exploration de l'arbre

Ce principe d'évaluation ayant été donné, il faut désormais parcourir l'arbre des coups possibles pour cela, l'algorithme va fonctionner de la manière suivante

- on part d'un nouveau jeu
- tant qu'on n'arrive pas sur une situation connue (gagnante/perdante)
  - le joueur dont c'est le tour
  - regarde ses coups possibles
    - s'il y a un coup gagnant alors la situation est gagnante et on met à jour
    - s'il ne reste que des coups perdants, alors la situation est perdue et on met à jour
    - s'il y a des coups inconnus, alors on choisit un de ses coups
  - le coup est joué et on passe au joueur suivant.

4.2 Résolution par apprentissage

Cette approche consiste à partir du noeud de départ et à générer une trajectoire. Au fur et à mesure que l'on suit la partie, on mettra à jour les situations en fonction de ce que l'on peut dire.

4.2.1 Première étape: trajectoire complète

Faire une fonction `trajectoire(depart, valeur)` qui part du début et qui va jusqu'à une situation finale (gagnante/perdante/égalité ss coup possible) en jouant à chaque fois au hasard.

Quand on arrive à une situation finale, sauvez l'état de la situation dans le dictionnaire `valeur`.

4.2.2 Deuxième étape: prise en compte des fils

calculFils Ecrivez une fonction intermédiaire `calculFils` qui retourne à une situation donnée, l'ensemble des coups possibles partitionnés en

- les coups gagnants
- les coups perdants
- les coups conduisant à une égalité
- les coups inconnus (ceux qui ne sont pas dans le dictionnaire)

On utilisera la table valeur pour le savoir.

Quand on utilisera cette fonction, les coups seront au départ tous quasi inconnus, puis au fur et à mesure de la construction de la fonction, ils prendront des valeurs adéquates.

La fonction `calculFils` retournera le tableau `[gagnant,perdant,egalite,inconnu]`.

On utilisera les fonctions

- `valGagnant(joueur(jeu))` qui donne la valeur d'un coup gagnant en fonction du joueur (1 si 'X' et -1 si 'O')

- `valPerdant(joueur(jeu))` qui donne la valeur d'un coup gagnant en fonction du joueur (-1 si 'X' et 1 si 'O')

```
1
2  #donne la valeur d'un jeu gagnant en fonction du joueur
3  def valGagnant(joueurEnCours):
4      if joueurEnCours=='X' :
5          gagne=1
6      else :
7          gagne=-1
8
9  #donne la valeur d'un jeu perdant en fonction du joueur
10 def valPerdant(joueurEnCours):
11     if joueurEnCours=='X' :
12         gagne=-1
13     else :
14         gagne=1
```

construction valeur Modifier la fonction `trajectoire` pour écrire `trajectoire2(depart, valeur)`. Désormais le jouer regarde les coups suivants avant de jouer

- si l'état courant et gagne ou perdu, il sauve l'état et s'arrête
- s'il a un coup gagnant, il sauve l'état en disant que cet état est gagnant
- s'il y a des coups inconnus, il en choisit 1 au hasard et passe au suivant
- sinon c'est que tous les coups conduisent soit à une défaite, soit à une égalité
  - s'il y a un coup conduisant à une égalité, la valeur est 0
  - sinon tous les coups conduisent à une défaite et la valeur est perdu.
  - enfin, s'il n'y a pas de coup possibles, le jeu s'arrête et la valeur de la situation peut être estimée par `gagner()`

faire un exemple simple

4.2.3 Troisième étape: résolution

Répéter ces trajectoires jusqu'à ce que la situation initiale soit gagnante, perdante ou nulle.

5 Lancement du jeu

Dans cette partie, voici deux fonctions qui vont permettre de lancer le en utilisant à chaque fois le dictionnaire construit dans les parties précédentes.

5.1 Jeu automatique

Le jeu tourne automatiquement en parcourant les situations vues, on reste donc dans le cadre des explorations qui ont été faites.

```
#coupMeilleur
def coupMeilleur(jeu,table):
    #cherche le coup égal à la valeur
    coups=coupPossible(jeu)
    valeur=table[jeu]

    #pour faire au hasard
    #on choisit le coup au hasard parmi les meilleurs
    fini=False
    while (not fini):
        #choisit un coup au hasard
        coup=coups[random.randint(0,len(coups)-1)]
        #si la valeur est celle attendue, on return
        njeu=jouer(jeu,coup)
        #si la cle existe
        if (njeu in table):
            if (table[njeu]==valeur):
                return coup

#permet de lancer une partie
def partie(depart,table):
    jeu=depart
    #tant que le jeu n'est pas fini
    while (gagner(jeu)==0) and len(coupPossible(jeu))!=0:
        coup=coupMeilleur(jeu,table)
        print("coup joue", coup)
        jeu=jouer(jeu,coup)
        affiche(jeu)
        print("-----")
```

5.2 Jouer une partie

On propose à un utilisateur de faire un adversaire. Comme l'utilisateur peut choisir un autre coup que les coups calculés (parce qu'il ne joue pas de manière optimale par exemple), il est nécessaire de recalculer une solution si on arrive sur un état non connu.

```
#permet de choisir le coup
def afficheMatrice():
    print ("------")
    print ("0 1 2")
    print ("3 4 5")
    print ("6 7 8")
```

```
7       print ("------")
8
9   #permet de jouerPartie
10  def jouerPartie(depart,table,humain):
11      jeu=depart
12      print("******* bonne partie *****")
13      affiche(jeu)
14      #tant que le jeu n'est pas fini
15      while (gagner(jeu)==0) and len(coupPossible(jeu))!=0:
16          #si c'est au joueur
17          if joueur(jeu)==humain :
18              print("----- a vous -----")
19              coup=-1
20              #demander coup valide
21              coups=coupPossible(jeu)
22              afficheMatrice()
23              while(coup not in coups):
24                  coup=int(input("quel coup jouer ? "))
25              #jouer coup
26              jeu=jouer(jeu,coup)
27              affiche(jeu)
28
29          # si c'est au programme
30          else:
31              print("----- a moi -----")
32              #deux cas se posent
33              # si la situation n'est pas connue, on la calcule
34              #cela arrive si on sort du chemin optimal
35              if (jeu not in table):
36                  table=calcul(jeu)
37              #joue le meilleur coup
38              coup=coupMeilleur(jeu,table)
39              jeu=jouer(jeu,coup)
40              affiche(jeu)
```

6 Extension

L'approche qui a été développée est utilisable dans de nombreux jeux (ex jeu de nim). Il suffit juste de redéfinir les fonctions de la partie 1

- `nouveauJeu()`
- `affiche()`
- `joueur()`
- `coupPossible()`

- `gagner()`
- `jouer()`

Pour conclure, voici quelques exemples de jeu de stratégie abstraits du commerce (en citant le site de référence `http://trictrac.net`)

- diaballik `http://trictrac.net/jeu-de-societe/diaballik/infos`
- Gipf `http://trictrac.net/jeu-de-societe/gipf/infos`
- Mana `http://trictrac.net/jeu-de-societe/mana/infos`
- Kamon `http://trictrac.net/jeu-de-societe/kamon/infos`
- Dvonn `http://trictrac.net/jeu-de-societe/dvonn/infos`