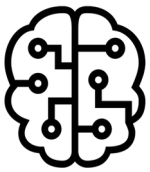
L'intelligence d'un morpion



Depuis plusieurs années les grandes firmes de l'internet (GAFA) investissent énormément d'argent dans les recherches portant sur l'intelligence artificielle. Des chercheurs français comme Yann Le Cun (très connu pour ces publications sur les neurones profonds) se sont particulièrement illustrés par leurs recherches et sont devenus très sollicités par les médias pour vulgariser leurs travaux auprès du grand public. Mais comment expliquer simplement la modélisation mathématique d'un concept aussi abstrait que l'intelligence ?

Ce terme d'intelligence artificiel est devenu finalement un four-tout très souvent utilisé à défaut. En fait il s'agit de la faculté de reproduire un " raisonnement " par des moyens informatiques. C'est l'ordinateur qui "pense"... pour reconnaître, s'adapter à des situations... comme le ferait un être humain.

A l'heure actuelle cette IA peut être très performante mais est limitée à une analyse simple des choses (reconnaître un caractère écrit, reconnaître la parole...) et est encore très loin de pouvoir rivaliser avec l'intelligence d'un enfant.

> *C'est logique ! Où il y a de l'intelligence artificielle, il a forcément de la stupidité artificielle.*

1. Technique de création d'intelligence artificielle

Cette faculté de décision peut être obtenu par deux techniques distinctes :

➔ **l'IA descendante (méthode experte)** : imitation fidèle d'un comportement observé et qui est reproduit à l'identique à l'aide d'un programme informatique. Cette technique a beaucoup évolué dans les années 1980, elle est très performante dans son domaine mais reste confiné dans celui-ci, sans possibilité d'évoluer. Cette méthode est capable à partir de données d'entrée et par application de règles apprises, de déduire une conclusion logique. Elle nécessite que tous les champs de connaissance soient décrits de manière exhaustive, faute de quoi on s'expose à des conclusions erronées.

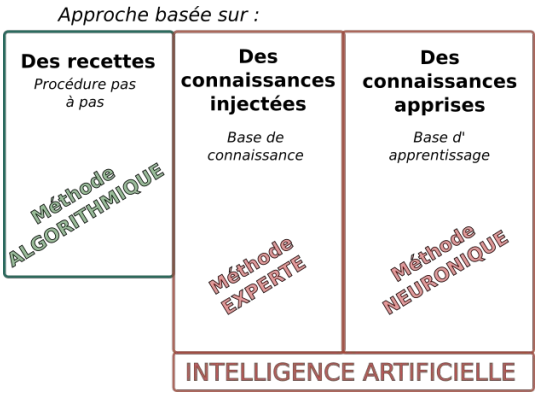


Figure 1: Méthodes de création d'IA

➔ **l'IA ascendante (méthode neuronique)** : cette fois le comportement humain est mimé par suite d'apprentissage et accumulation de connaissances de plus en plus complexes. Les algorithmes évoluent et échappent progressivement au cadre fixé au départ par leurs auteurs. Ce domaine est en pleine évolution depuis les années 2000 et bénéficie de méthodes d'apprentissage de plus en plus performantes (deep learning, machine learning, neurones profonds...)

2.L'intelligence artificielle et les jeux

L'application aux jeux a toujours été un domaine actif de l'Intelligence Artificielle. On s'intéresse dans cette activité aux jeux asynchrones opposant deux joueurs jouant chacun leurs tours. Ces jeux à deux joueurs sont tous des *jeux à information complète et à somme nulle*. Cela signifie que

- *jeu à information complète* : les deux joueurs ont à tout moment toute l'information sur l'état du jeu. Il n'y a par exemple pas de carte dans la main d'un joueur que l'autre ne connaît pas. Il en résulte que chaque joueur connaît toutes les informations à disposition de son adversaire au moment où celui-ci doit jouer.
- *jeu à somme nulle* : les gains réalisés par un joueur sont des pertes pour l'autre joueur. Il y a donc forcément un gagnant et un perdant (ou un match nul). Mais il n'y a pas un joueur qui gagne beaucoup et l'autre moins. Il en résulte que les intérêts des deux joueurs sont opposés. Les meilleurs coups de jeu pour l'un sont les pires pour l'autre.

3.Modélisation du jeu du morpion

But : Programmer un jeu de morpion joueur contre joueur en mode console

Il existe un algorithme qui permet de faire jouer un programme à des jeux asynchrones, il s'agit de l'algorithme du **min-max**. Lors de cette activité nous appliquerons cet algorithme au jeu tic-tac-toe connu aussi sous le nom du jeu du morpion

Représentation du plateau

Le jeu du morpion sera programmé dans une classe Morpion dont le code incomplet est fourni sur Moodle dans le fichier Morpion.py

Le plateau de jeu est stocké dans l'attribut de classe __plateau et est représenté par une liste de 9 caractères. Une croix présente sur une case sera repérée par le caractère 'x' dans la position de la liste correspondante. Le rond sera codé par un 'o' et une case vide par '.' . L'exemple suivant illustre le codage d'une situation particulière du plateau

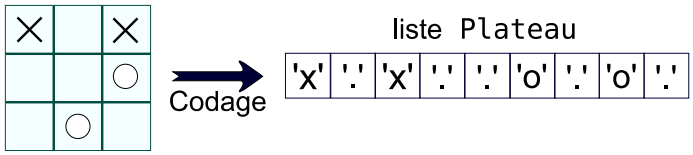


Figure 2: Codage du plateau

Définition de la classe Morpion

Q1. **Compléter** les méthodes incomplètes de la classe Morpion en respectant leur spécification. **Tester** chaque méthode avec les jeux de test présents dans les spécifications.

Q2. **Programmer** des levées d'exceptions lorsque le joueur choisit une case indéfinie du plateau ou si le joueur joue sur une case déjà jouée.

Morpion Joueur Vs Joueur

Q3. **Coder** un jeu de morpion de deux joueurs respectant le cahier des charges suivant :

Cahier des charges

- ✗ *L'affichage du jeu sera réalisé sur la console python*
- ✗ *Le premier joueur du jeu est obligatoirement X*
- ✗ *Une invite demande alternativement à chaque joueur de jouer.*
- ✗ *L'état du plateau est affiché à chaque fois qu'un joueur a validé un coup*
- ✗ *Lorsqu'un coup est joué, l'ordinateur vérifie que ce coup est possible. Dans le cas contraire un message d'avertissement s'affiche sur la console*
- ✗ *Le programme doit tester à chaque fois l'état du jeu pour détecter la fin de partie. Dès que la partie est terminée un message doit indiquer le gagnant ou que la partie est nulle.*

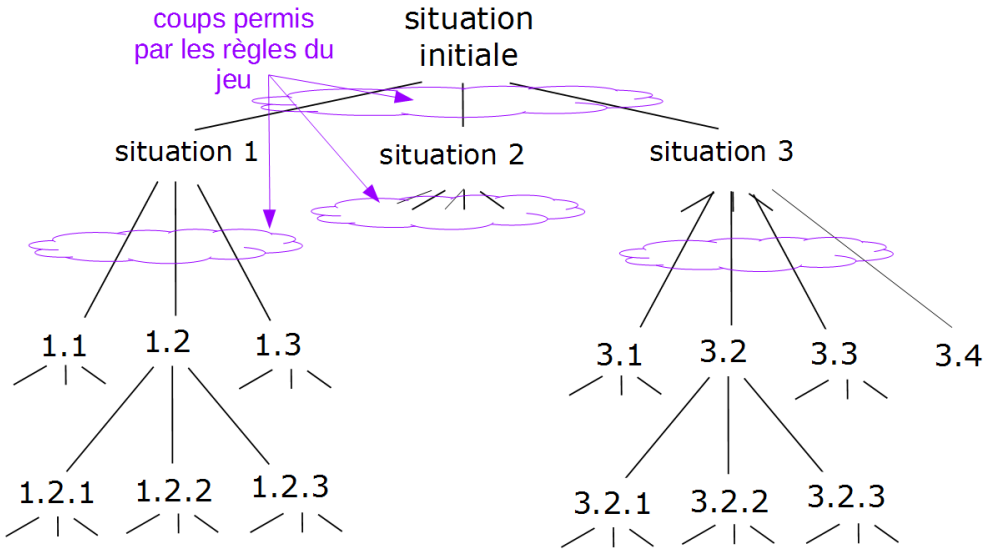
4.Intelligence artificielle pour jouer au morpion

Nous avons vu que dans les jeux asynchrones opposant deux joueurs, on part d'une *situation courante* et les règles du jeu déterminent la liste des nouvelles situations de jeu atteignables à partir de celle-ci. Chacune de ces situations, si elle est choisie par le joueur, devient la prochaine *situation courante* à partir de laquelle l'autre joueur devra à son tour choisir la prochaine situation.

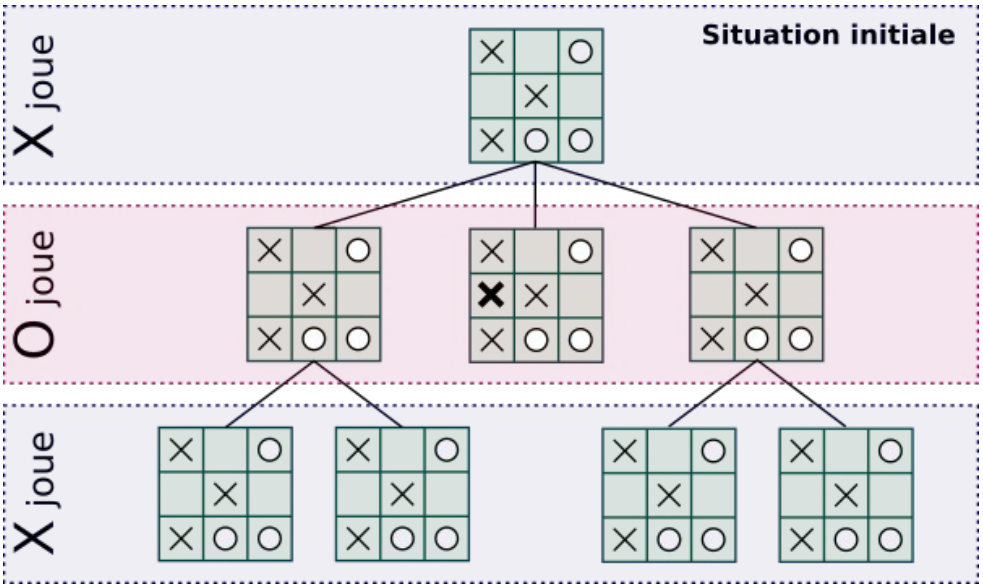
Pour que l'ordinateur puisse jouer, il faut mettre en place une intelligence artificielle capable d'analyser chaque situation afin de choisir un coup gagnant mais aussi contrer les coups de l'adversaire.

L'arbre de jeu

Chaque situation courante propose donc un ensemble d'embranchements que peut choisir le joueur, et cette situation se reproduit à chaque tour de jeu. On peut donc représenter l'ensemble de toutes les parties possibles à partir d'une *situation courante* initiale sous la forme d'un arbre dont les nœuds sont les différentes situations de jeu possibles. Cet arbre est appelé **arbre de jeu**.



➔ A partir de la situation de jeu suivante **compléter** l'arbre de jeu en plaçant les croix/ronds manquants jusqu'à atteindre la fin de partie. **Encadrer** les plateaux en situation gagnante.



Évaluation de la pertinence d'un coup

Le problème qui se pose au joueur qui doit jouer, est de choisir une branche de l'arbre qui l'amène vers la meilleure situation finale possible, c'est-à-dire, si possible une victoire. La question qui se pose à lui est donc *Quel coup jouer maintenant pour gagner plus tard ?*

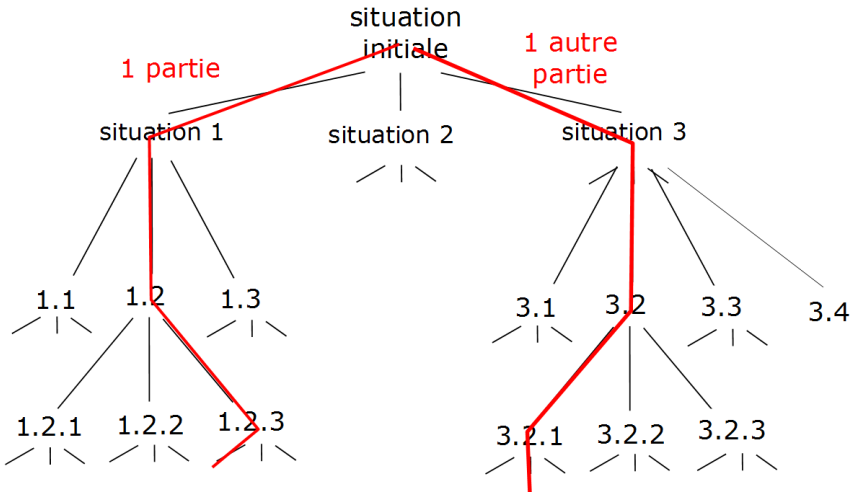


Figure 1: Quel coup jouer maintenant pour gagner plus tard ?

L'objectif est donc la qualité de la situation visée "*dans le futur*" et non pas celle atteinte immédiatement, et cette qualité doit être la meilleure possible. Comment faire ? Et bien comme tout joueur humain le fait, l'algorithme va "*calculer des coups à l'avance*" et prendre sa décision en fonction de ces calculs.

Evaluation des coups

On va noter par **MAX** le joueur qu'on cherche à faire gagner (X) et son adversaire (O) par **MIN**. Les deux joueurs désirent gagner le jeu. On suppose que le joueur MIN joue logiquement et qu'il ne va jamais rater une occasion de gagner. Si pour gagner le joueur MAX essaie de maximiser son score, le joueur MIN désire aussi maximiser son propre score (ou de minimiser le score du joueur MAX). L'algorithme MINIMAX, inventé par Von Neumann, à comme but l'élaboration d'une stratégie optimale pour le joueur MAX. À chaque tour le joueur MAX va choisir le coup qui va maximiser son score, tout en minimisant les bénéfices de l'adversaire. Ces bénéfices sont évalués en termes de la fonction d'évaluation statique utilisée pour apprécier les positions pendant le jeu. Cette fonction d'évaluation statique renvoie une note pondérée pour chaque coup :

- ➔ **+10** si le coup fait gagner *X*
- ➔ **-10** si le coup fait gagner *O*
- ➔ **0** si le coup entraîne une partie nulle
- ➔ Rien (**None**) pour une partie inachevée

Le fichier MorpionIA.py contient une nouvelle classe MorpionIA qui hérite de classe Morpion. Par cet héritage les objets de la classe MorpionIA auront accès à l'ensemble des méthodes/attributs de la classe Morpion.

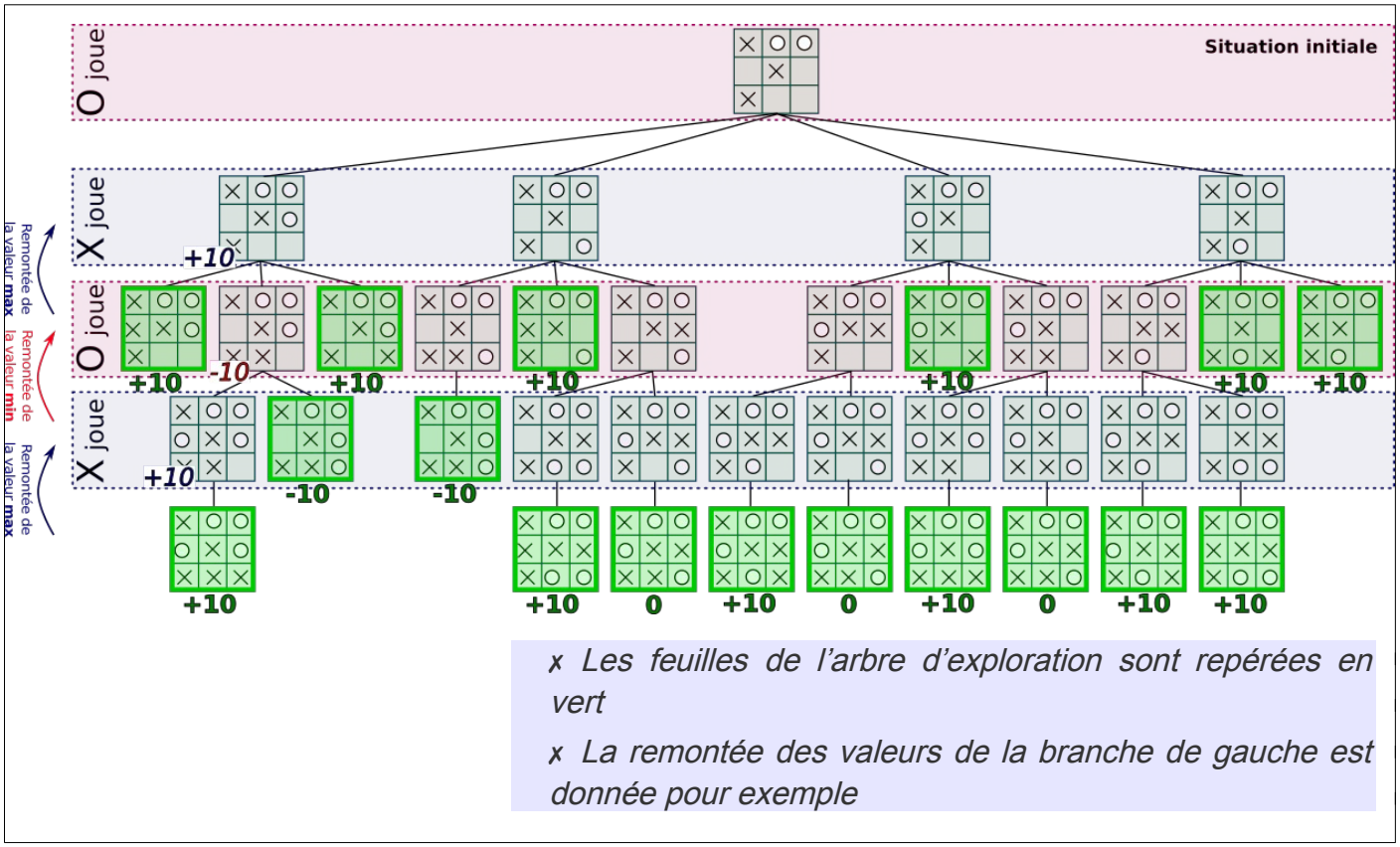
Q4. **Compléter** la méthode evaluationStatique() de la classe MorpionIA en respectant sa spécification. **Tester** cette méthode avec le jeux de test présent dans la spécification.

L'algorithme MINMAX

L'idée de l'algorithme est de développer complètement l'arbre de jeu, de noter chaque feuille (positons extrêmes) avec sa valeur (0, +10 ou -10 selon la situation), puis de faire remonter ces valeurs avec l'hypothèse que chaque joueur choisit le meilleur coup pour lui. Cela signifie en pratique que :

- ✗ Sur les nœuds où MAX (X) joue, on remonte la valeur **maximale** des situations directement suivantes
- ✗ Sur les nœuds où MIN (O) joue, on remonte la valeur **minimale** des situations directement suivantes

Prenons pour exemple la situation :



✗ *Les feuilles de l'arbre d'exploration sont repérées en vert*

✗ *La remontée des valeurs de la branche de gauche est donnée pour exemple*

Q5. Sur cet arbre, **déterminer** la valeur de chaque nœud en effectuant la remontée des valeurs des feuilles.

Q6. **Déterminer** le vainqueur de cette partie en interprétant les valeurs des nœuds situés sous la situation initiale.

L'algorithme Min Max d'évaluation et de parcours d'arbre est le suivant :

```
Fonction MinMax(situation, joueur)
    Pour chaque coup possibles

        si situation finale atteinte
            score=evaluer situation
        sinon
            score=MinMax(situation,adversaire de joueur)
        conserver meilleurScore de score
        Retirer le coup de joueur
    retourner meilleurScore
```

Q7. **Justifier** que cet algorithme est récursif

Q8. **Déterminer** la profondeur maximale de récursivité que peut atteindre cet algorithme lorsqu'il est appliqué au jeu du morpion. **Justifier** que l'utilisation de cet algorithme récursif n'engendrera pas de dépassement mémoire s'il est codé en Python.

L'implémentation incomplète de cet algorithme en Python est la suivante :

```
    def minmax(self,joueur):
        '''Identifie le meilleur coup (algo Min_Max)
        Parametre : joueur (str) : 'x', 'o'
        retour : position du meilleur coup (int)
        '''
        scoreBranches = [] #Liste des scores de chaque branche

        for coup in self.coupsRestants():
            score=self.evaluerCoup(joueur,coup)
            if score==None:
                score,_=self.minmax(                    ))

            scoreBranches.append((score,coup))
            self.jouer('.', coup)  # efface le coup joué

        if joueur=='x' : return max(scoreBranches)
        else : return
```

Q9. **Compléter** l'appel récursif de la méthode minmax() et le retour de fonction (zones cachées). **Tester** chaque méthode avec les jeux de test présents dans les spécifications.

Q10. A partir des bibliothèques Morpion et MorpionIA, **modifier** votre jeu du morpion en y ajoutant ces fonctionnalités :

Cahier des charges supplémentaire

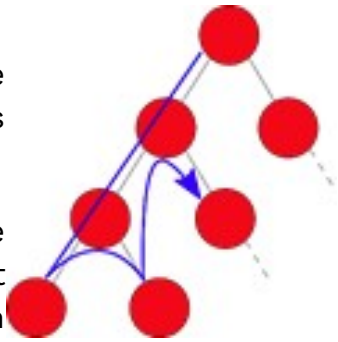
➔ *Le jeu peut se jouer en mode joueur contre joueur ou en mode joueur contre machine*

➔ *une invite au départ demande le mode souhaité*

5.Complexification du jeu

But de cette partie : Augmenter la taille du plateau afin complexifier le jeu et développer les solutions gagnantes de l'IA.

Analyse de l'algorithme d'IA



L'algorithme utilisé pour modéliser l'IA du morpion teste systématiquement l'ensemble des solution du jeu et pondère les coups gagnants.

Cet algorithme entre dans la catégorie des algorithmes de backtracking. Lorsqu'une solution est trouvée, la méthode revient sur les affectations qui ont été faites précédemment (d'où le nom de retour sur trace) afin d'en explorer de nouvelles.

L'algorithme de backtracking est très utilisé pour répondre à des problèmes de recherche et d'optimisation. Le principe de l'algorithme est simple : construire dynamiquement une solution au problème en faisant croître des parties de solution, tant que ces parties sont suffisamment prometteuses.

Q11. **Indiquer** le type de parcours d'arbre effectué par cet algorithme.

La recherche des meilleurs solutions impose à notre algorithme de parcourir l'ensemble des coups possibles. Augmenter la taille du plateau aura pour conséquence d'augmenter le temps de calcul. La question qui se pose est de savoir si notre algorithme se terminera en un temps acceptable lorsque le plateau deviendra grand.

Complexité du problème

Pour rappel, la théorie de la complexité étudie la quantité de ressources (temps, espace mémoire, etc.) dont a besoin un algorithme pour résoudre un problème algorithmique. Dans notre cas la complexité de notre algorithme dépend directement du nombre de cases sur le plateau.

Q12. **Justifier** que cet algorithme est de complexité factorielle **O(n!)**

Q13. **Calculer** le nombre cas que l'algorithme devra étudier si le plateau possède 9 (3x3) puis 16 (4x4) puis 25 (5x5) cases.

On considère le temps nécessaire à la recherche d'une solution à 50 opérations machines.

Q14. **Estimer** le temps de calcul qu'il faudra à un processeur 1GHz pour terminer l'algorithme dans les trois cas précédents (9, 16, 25 cases).

Q15. **Conclure** sur la possibilité d'augmenter la taille du plateau en conservant cet algorithme.

Amélioration de l'algorithme

Une première piste d'amélioration de performance de notre algorithme consiste à ne pas explorer une solution qui l'a déjà été (figure 3). Pour cela, il convient de mémoriser au fur et à mesure, les différents coups avec leur score.

Cette technique, dite technique de **mémoïsation**, réduit le temps de calcul (coût en temps) au détriment de l'occupation mémoire (coût spatial).

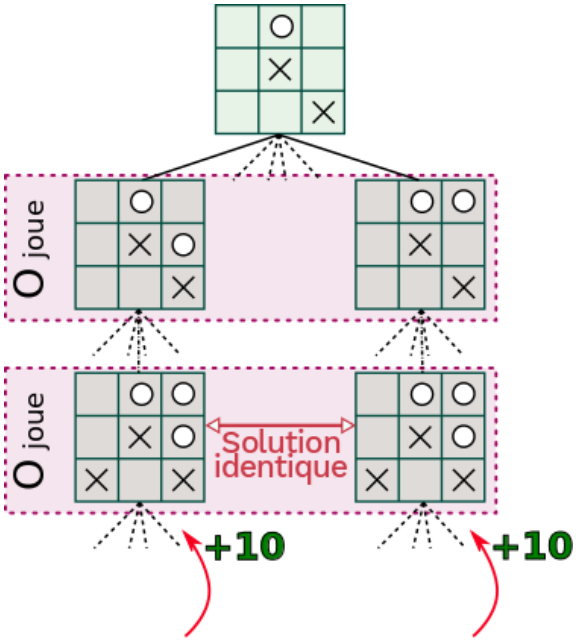


Figure 3: Exemple de situation identique

Implémentation de de la mémoïsation

L'ensemble des solutions / score sera stocké dans un dictionnaire, les états du plateau constituant les clés de ce dictionnaire et les scores seront les valeurs associées.

Les objets listes ne pouvant pas constituer une clé valide (objet non hashable), une chaîne de caractère composée des caractères de la liste séparés par des virgules servira de clé. Par exemple la solution de la figure 3 sera stockée de la manière suivante :

```
memo → { … , '.,o,o,.,x,o,x,.,x' : 10 , … }
```

La concaténation des caractères de la liste séparés par des virgules peut être obtenu avec l'instruction :

```
>>> ','.join(['.','o','o','.','x','o','x','.','x'])
'.,o,o,.,x,o,x,.,x'
```

Le chemin inverse est obtenu avec la méthode split() :

```
>>>'.,o,o,.,x,o,x,.,x'.split(',')
['.','o','o','.','x','o','x','.','x']
```

La méthode minmax_memo() suivante implémente cette mémoïsation.

```
def minmax_memo(self,joueur):
      scoreBranches = []
      for coup in self.coupsRestants():
            score=self.evaluerCoup(joueur,coup)
            if score==None:
                  ##### Test si la cas est dans le dico de memoisation
                  if ','.join(self.getPlateau()) in self.memo :
                        #### Retourne le score sans tester les sous arbres
                        [                                                ]  Q9

                        #### Efface le coup joué
                        self.jouer('.', coup)
                        return score

            score,_=self.minmax_memo(self.adversaire(joueur) )

            scoreBranches.append((score,coup))
            self.jouer('.', coup) # efface le coup joué

      if joueur=='x' :
            score = max(scoreBranches)
            ###### Ajout de la memorisation du score si 'x' joue
            [                                                ]  Q10
            ######
            return score
      else :
            score = min(scoreBranches)
            ###### Ajout de la memorisation du score si 'o' joue
            [                                                ]  Q10
            ######

            return score
```

Q16. **Compléter** la zone repérée *Q9* afin de renvoyer le score si l'état du plateau est dans le dictionnaire self.memo

Q17. **Compléter** la zone repérée *Q10* afin de mémoriser l'état du plateau et le score dans la variable self.memo

Evaluation des performances de la mémoïsation

Sur Moodle, le répertoire performances contient la classe MorpionIA et une bibliothèque d'évaluation des performance des algorithmes de backtracking avec et sans mémoïsation (runtest.py). Les fonctions suivantes sont disponibles :

➔ runtest.compare_performances() : Compare les performances en temps des deux algorithmes pour un certain nombre de coups restants sur un plateau de 9 cases

➔ runtest.performance_IA_memo() : Mesure les performances en temps et espace de l'algorithme de backtracking avec mémoïsation.

Q18. **Ouvrir** et **exécuter** le fichier MorpionIA.py puis **exécuter** les commandes de test

Q19. **Analyser** les performances affichées et conclure en décrivant les avantages / inconvénients de la mémoïsation. **Préciser** les limites des deux algorithmes quant à leur capacité à supporter l'agrandissement du plateau.