

Algorithmes de Jeux

Hatem Ghorbel - Stefano Carrino

2020 – 2021



Les algorithmes de jeu

Domaine d'application

- Jeux de réflexion, déterministes
- Deux joueurs
 - jouant alternativement. . .
 - ... et dont (au moins) l'un est l'ordinateur
- Exemples :
 - Échecs
 - Othello
 - Go
 - ...

L'idée de base

- La première idée qui vient à l'esprit est de jouer le coup qui nous amène dans la meilleure position “immédiate”
 - Il faut donc pouvoir évaluer la “qualité” d'un état du jeu
- Mais cette vue à trop court terme ne suffit souvent pas : il faut tenir compte de la réaction de l'adversaire, si possible plusieurs coups à l'avance !
 - On va donc considérer un *arbre* des coups possibles

Les ingrédients

Structure de données : un arbre

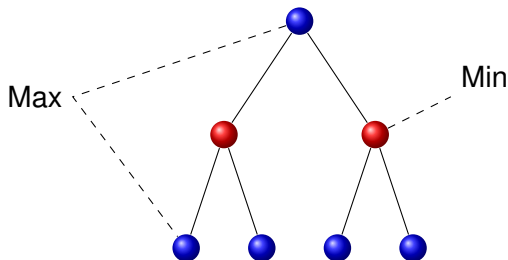
- Les noeuds sont les *états* du jeux
- Les arcs sont les *coups* possibles

Fonction d'évaluation

- Pour chaque état, renvoie une évaluation de la situation
 - Du point de vue de l'ordinateur : plus l'évaluation est élevée, plus la situation est bonne pour l'ordinateur
- Il s'agit d'une évaluation instantanée, ne tenant compte ni de l'histoire ni du futur
- Contient une bonne partie de "l'intelligence" !

Le principe

- On part de l'idée que chaque **joueur** va tenter, à terme, de maximiser sa fonction d'évaluation
- Du point de vue de l'ordinateur, cela va donc dire qu'on va tenter de maximiser un "étage" sur deux, et minimiser l'autre étage



TreeNode

```
class TreeNode:
    def eval(self):
        """Evaluation function"""
        [...]
    def final(self):
        """Is the game over? """
        [...]
    def ops(self):
        """Returns applicable operators"""
        [...]
    def apply(self, op):
        """Applies the operator op on this state
        and returns the resulting TreeNode"""
        [...]
```

La fonction max

```
def max(root, depth):  
    if (depth == 0 or root.final()):  
        return root.eval(), None  
    maxVal = -inf  
    maxOp = None  
    for op in root.ops():  
        new = root.apply(op)  
        val, dummy = min(new, depth-1)  
        if val > maxVal:  
            maxVal = val  
            maxOp = op  
    return maxVal, maxOp
```

La fonction min

```
def min(root,depth):  
    if (depth == 0 or root.final()):  
        return root.eval(), None  
    minVal = inf  
    minOp = None  
    for op in root.ops():  
        new = root.apply(op)  
        val, dummy = max(new,depth-1)  
        if val < minVal:  
            minVal = val  
            minOp = op  
    return minVal, minOp
```


Minimax

```
def minimax(root, depth):  
    val, op = max(root, depth)  
    return val, op
```

Améliorations

- Remplacer la limitation de profondeur par
 - Une mesure de “quiétude” : s’arrêter lorsque la situation paraît “stable”
 - permet d’éviter les problèmes d’horizon
 - délicat à implémenter !
 - Une limitation en temps
 - Si la limitation est stricte, difficulté de répartir le temps passé sur chaque branche. . .
- Ne pas recalculer tout l’arbre à chaque coup, mais étendre le calcul déjà effectué !

Éviter la duplication de code...

• Différences entre min et max :

```
def max(root, depth):
    if (depth == 0 or root.final()):
        return root.eval(), None
    maxVal = -inf
    maxOp = None
    for op in root.ops():
        new = root.apply(op)
        val, dummy = min(new, depth-1)
        if val > maxVal:
            maxVal = val
            maxOp = op
    return maxVal, maxOp
```

```
→ ← def min(root, depth):
    if (depth == 0 or root.final()):
        return root.eval(), None
    ← ← minVal = inf
        minOp = None
        for op in root.ops():
            new = root.apply(op)
            ← ← val, dummy = max(new, depth-1)
                if val < minVal:
                    minVal = val
                    minOp = op
    return minVal, minOp
```

Éviter la duplication de code...

- Différences entre `min` et `max` :

<pre>def max(root,depth): if (depth == 0 or root.final()): return root.eval(), None maxVal = -inf maxOp = None for op in root.ops(): new = root.apply(op) val, dummy = min(new, depth-1) if val > maxVal: maxVal = val maxOp = op return maxVal, maxOp</pre>	<pre>def min(root,depth): if (depth == 0 or root.final()): return root.eval(), None minVal = inf minOp = None for op in root.ops(): new = root.apply(op) val, dummy = max(new, depth-1) if val < minVal: minVal = val minOp = op return minVal, minOp</pre>
---	--

- Comme $a < b \Leftrightarrow -a > -b$, on peut rassembler les fonctions `min`, `max` et `minimax` en une seule.

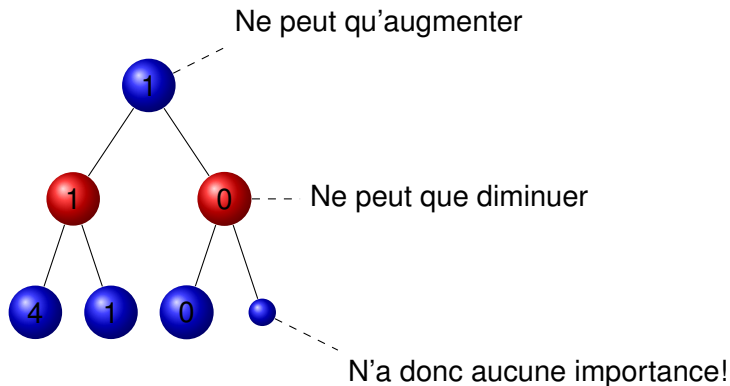
Minimax – version 2

```
def minimax2(root, depth, minOrMax):  
    # minOrMax = 1 : maximize  
    # minOrMax = -1 : minimize  
    if (depth == 0 or root.final()):  
        return root.eval(), None  
    optVal = minOrMax * -inf  
    optOp = None  
    for op in root.ops():  
        new = root.apply(op)  
        val, dummy = minimax2(new, depth-1, -minOrMax)  
        if val * minOrMax > optVal * minOrMax:  
            optVal = val  
            optOp = op  
    return optVal, optOp
```

Sortons la trançonneuse...

- Dans l'algorithme minimax, le nombre de noeuds à considérer augmente très rapidement avec la profondeur
 - Généralement, évolution exponentielle
- Si on arrivait à déterminer que l'exploration de certaines branches de l'arbre est inutile, on pourrait gagner beaucoup de temps...
- C'est exactement le but de l'algorithme Alpha-beta
- On va donc "couper" des branches de l'arbre, mais de manière à ce que l'algorithme retourne toujours la même valeur qu'avant l'optimisation !

Alpha-beta en image




Alpha-beta en texte

- Si la valeur d'un noeud N où l'on minimise devient inférieure à celle de son parent direct, on a pas besoin de considérer les autres enfants de N
- Si la valeur d'un noeud M où l'on maximise devient supérieure à celle de son parent direct, on a pas besoin de considérer les autres enfants de M
- Remarque : la valeur de maximisation est traditionnellement appelée α (alpha) et celle de minimisation β (beta), d'où le nom de l'algorithme.

De minimax à alphabeta

```
def max(root,depth):  
    if (depth == 0 or root.final()):  
        return root.eval(), None  
    maxVal = -inf  
    maxOp = None  
    for op in root.ops():  
        new = root.apply(op)  
        val, dummy = min(new, depth-1)  
        if val > maxVal:  
            maxVal = val  
            maxOp = op  
    return maxVal, maxOp
```



```
def ab_max(root,parentMin,depth):  
    if (depth == 0 or root.final()):  
        return root.eval(), None  
    maxVal = -inf  
    maxOp = None  
    for op in root.ops():  
        new = root.apply(op)  
        val, dummy = ab_min(new, maxVal, depth-1)  
        if val > maxVal:  
            maxVal = val  
            maxOp = op  
        if maxVal > parentMin:  
            break  
    return maxVal, maxOp
```

Alpha-beta en code

```
def ab_max(root,parentMin,depth):  
    if (depth == 0 or root.final()):  
        return root.eval(), None  
    maxVal = -inf  
    maxOp = None  
    for op in root.ops():  
        new = root.apply(op)  
        val, dummy = ab_min(new, maxVal, depth-1)  
        if val > maxVal:  
            maxVal = val  
            maxOp = op  
            if maxVal > parentMin:  
                break  
    return maxVal, maxOp
```

- De même pour `ab_min`, alphabeta.

Alpha-beta en code – version 2

```
def alphabeta2(root, depth, minOrMax, parentValue):  
    # minOrMax = 1 : maximize  
    # minOrMax = -1 : minimize  
    if (depth == 0 or root.final()):  
        return root.eval(), None  
    optVal = minOrMax * -inf  
    optOp = None  
    for op in root.ops():  
        new = root.apply(op)  
        val, dummy = alphabeta2(new, depth-1,  
                                -minOrMax, optVal)  
        if val * minOrMax > optVal * minOrMax:  
            optVal, optOp = val, op  
            if optVal * minOrMax > parentValue * minOrMax:  
                break  
    return optVal, optOp
```

Remarques

- Pour pouvoir couper les enfants d'un noeud N , celui-ci doit déjà avoir une valeur
 - On aura donc déjà exploré au moins une branche en-dessous de N
 - Donc on ne coupe *jamais* une branche de gauche !
- Les coupures dépendent de l'ordre dans lequel on considère les opérateurs
 - Idéalement, on va donc les ordonner de manière à *provoquer* des coupures. . .
 - . . . ce qui n'est pas facile !

Conclusion

- Le principe de minimax/alphabeta est très simple. . .
- . . . Mais pour obtenir un programme qui joue (bien), tout reste à faire !
 - Implémenter toute la logique du jeu de manière efficace (en temps et en mémoire. . .)
 - Trouver une bonne fonction d'évaluation
 - Éventuellement
 - optimiser le nombre de coupures
 - bien gérer la profondeur et le temps
 - bibliothèques d'ouvertures et de fins de partie
 - . . .