Minimax

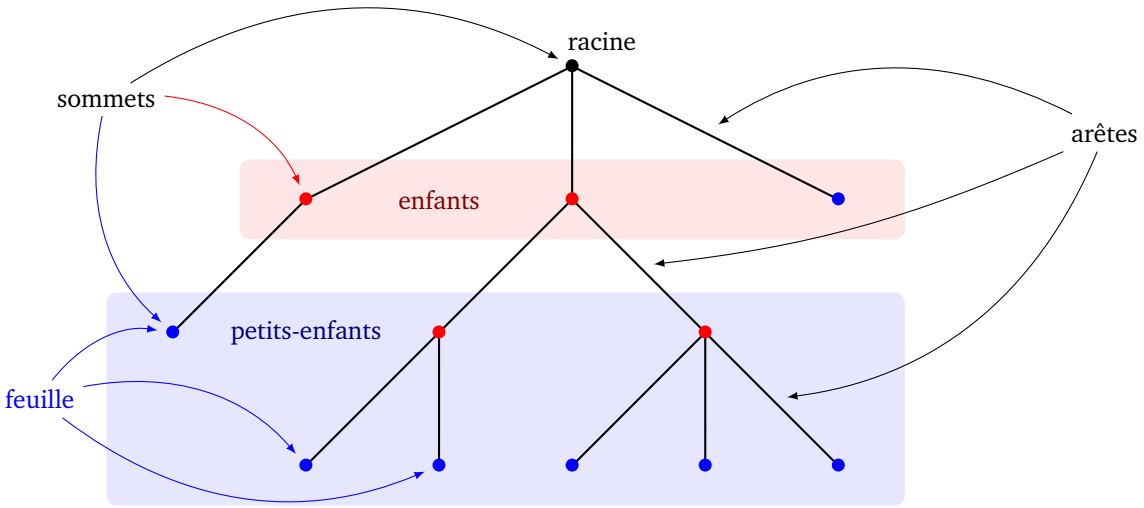
L'algorithme minimax permet de choisir le meilleur coup à jouer en anticipant les mouvements de l'adversaire.

1. Arbres

1.1. Vocabulaire

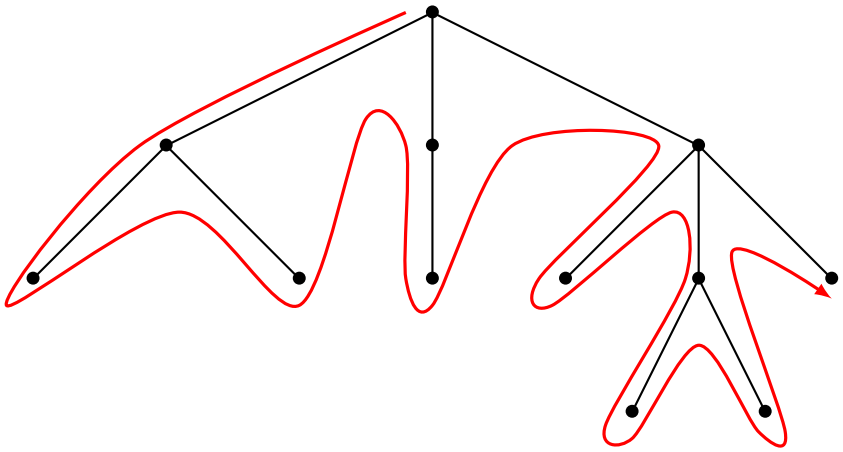
Voici le vocabulaire de base sur les arbres.

- Un ***arbre*** est un graphe sans cycle.
- Comme un graphe, un arbre est donc formé par des ***sommets*** (aussi appelés ***nœuds***) reliés par des ***arêtes***.
- La ***racine*** est un sommet privilégié qui sert d'origine. Cela définit une orientation naturelle sur l'arbre issue de la racine.
- Les sommets reliés à la racine sont les ***enfants*** ; les sommets reliés aux enfants (autres que la racine) sont les ***petits-enfants***…
- Plus généralement les ***enfants d'un sommet*** s sont les sommets s' reliés à s avec s' plus éloigné de la racine que s.
- Une ***feuille*** (ou ***sommet terminal***) est un sommet sans enfant.
- Une ***donnée*** sera généralement associée à chaque sommet (un nombre, une configuration…).



1.2. Parcours en profondeur

Nous souhaitons parcourir tous les sommets d'un arbre. Le ***parcours en profondeur*** permet d'atteindre tous les sommets en partant de la racine puis en balayant l'arbre de haut (la racine) en bas (les feuilles) et de la gauche vers la droite.



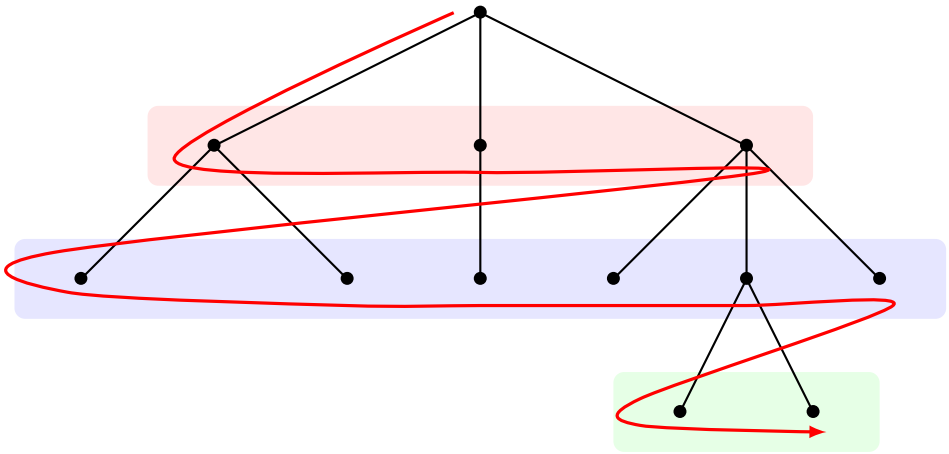
L'algorithme est assez court mais comme c'est un algorithme récursif c'est toujours un peu délicat à comprendre. L'algorithme est donné sous la forme d'une fonction récursive `parcours_profondeur(sommet)`. Pour parcourir tout l'arbre on l'appelle par `parcours_profondeur(racine)`. Dans cet algorithme, on peut effectuer une action sur le sommet à l'aide d'une fonction `traiter(sommet)` à personnaliser : par exemple on affiche la donnée attachée à ce sommet.

> **Algorithme.**
> Fonction : `parcours_profondeur(sommet)`
> Entrée : un sommet d'un arbre.
> Action : traite tous les descendants de ce sommet par un parcours en profondeur.
> - `traiter(sommet)`
> - pour chaque `'enfant'` de `sommet` :
>   — `parcours_profondeur(enfant)`

L'idée est la suivante : on part de la racine, on effectue l'action voulue sur ce sommet par une fonction `traiter(sommet)`, puis on considère l'ensemble des enfants de la racine. On considère le premier enfant (celui le plus à gauche), l'appel récursif fait que l'on traite ce sommet, puis on considère ses enfants de la gauche vers la droite (c'est-à-dire les petits-enfants de la racine) et on descend ainsi dans l'arbre. Une fois qu'on a traité tous les descendants du premier enfant, on passe au second enfant de la racine... Comme l'algorithme est récursif il est important de bien comprendre la condition d'arrêt : ici tout simplement si un sommet n'a pas d'enfant (c'est une feuille) alors on effectue juste l'action sur ce sommet (sans appel récursif) car la boucle « pour chaque `'enfant'` » est vide.
Le mieux est de programmer soi-même cette fonction pour en comprendre vraiment son mécanisme, voir la section « Avec Python » un peu plus loin.

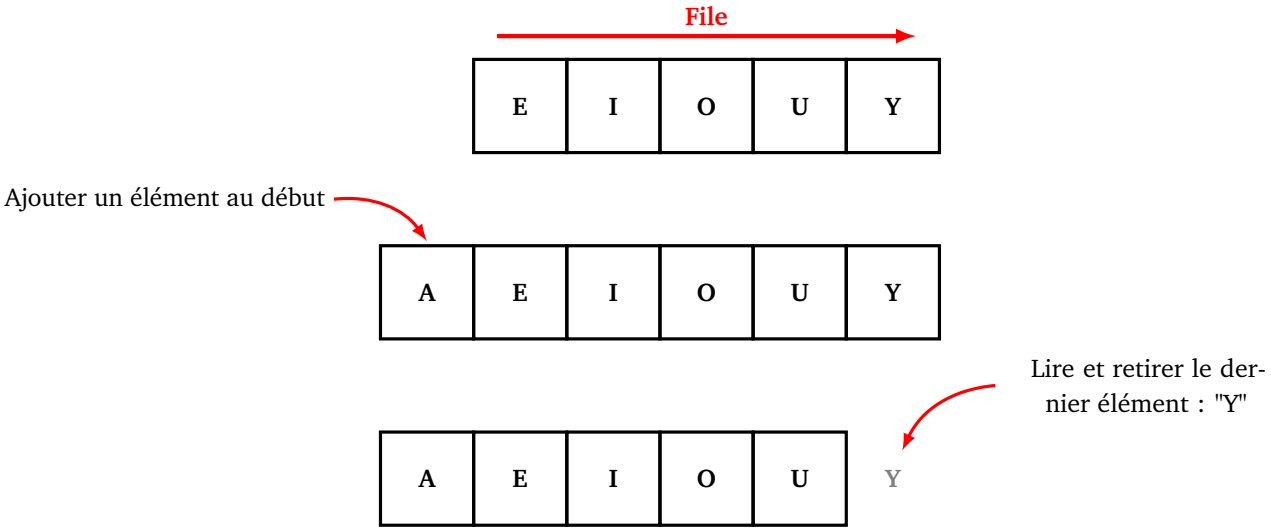
1.3. Parcours en largeur

C'est la façon qui semble humainement naturelle de parcourir un arbre mais un peu plus difficile pour un ordinateur. Le but est de partir de la racine, puis de traiter tous les enfants (de gauche à droite), puis l'ensemble de tous les petits-enfants, puis l'ensemble des petits-petits-enfants...



Avant d'étudier l'algorithme il faut comprendre la notion de « file ». Une ***file*** (*queue*) est une suite de données à laquelle on peut ajouter des éléments d'un côté et en retirer de l'autre. C'est comme une file d'attente : le premier arrivé sera le premier servi (*fifo* : *first in, first out*). Plus précisément voici les trois opérations de base :

- `file_vide()` : est-ce que la file est vide ?
- `ajouter_file(element)` : ajoute un élément à la gauche de la file,
- `retirer_file()` : renvoie l'élément le plus à droite de la file, et le retire de la file.



Il ne faut pas confondre une file avec une « pile » (*stack*) où le dernier élément ajouté à la pile est le premier à être retiré (*filo* : *first in, last out*).
Voici l'algorithme de parcours en largeur. Dans cet algorithme la file sert de stockage pour mémoriser les enfants d'un sommet, car il faut auparavant passer aux frères et sœurs de ce sommet.

> **Algorithme.**
> Fonction : `parcours_largeur(racine)`
> Entrée : un sommet (la racine de l'arbre).
> Action : traite tous les sommets par un parcours en largeur.
> - Partir d'une file qui contient uniquement le sommet `racine`.
> - Tant que la file n'est pas vide :
>
> — `sommet = retirer_file()`

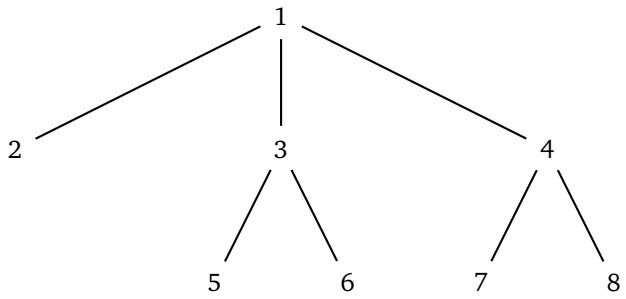
> — `traiter(sommet)`
> — pour chaque `'enfant'` de `sommet` :
> — `ajouter_file(enfant)`

1.4. Avec Python

On commence par définir une classe d'objet `Noeud` qui correspond à un sommet avec une valeur et une liste d'enfants (qui sont eux-mêmes des `Noeud`).

```
class Noeud:
    def __init__(self, val):
        self.enfants = []
        self.val = val
```

Voici comment initialiser l'arbre suivant :




```
racine = Noeud(1)
enfant1 = Noeud(2)
enfant2 = Noeud(3)
enfant3 = Noeud(4)
racine.enfants = [enfant1, enfant2, enfant3]
petitenfant1 = Noeud(5)
petitenfant2 = Noeud(6)
enfant2.enfants = [petitenfant1, petitenfant2]
petitenfant3 = Noeud(7)
petitenfant4 = Noeud(8)
enfant3.enfants = [petitenfant3, petitenfant4]
```

Voici la fonction récursive qui permet un parcours d'arbre en profondeur (ici elle affiche juste la valeur associée à chaque sommet) :

```
def parcours_arbre_profondeur(noeud):
    print(noeud.val)
    for enfant in noeud.enfants:
        parcours_arbre_profondeur(enfant)
```

Sur l'arbre de notre exemple, cela affiche la valeur de chaque sommet :

```
1 2 3 5 6 4 7 8
```

Voici la fonction qui permet un parcours d'arbre en largeur :

```
def parcours_arbre_largeur(noeud):
    file = [noeud]
```

```
while file != []:
    noeud = file.pop()
    print(noeud.val)
    for enfant in noeud.enfants:
        file = [enfant] + file
```

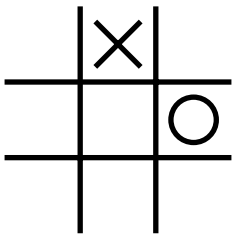
Rappelons que la commande `file.pop()` correspond à `retirer_file()` et renvoie le dernier élément de la liste et le retire de cette liste.
Toujours pour le même exemple, voici maintenant l'ordre d'affichage (racine, puis enfants, puis petits-enfants) :

1 2 3 4 5 6 7 8

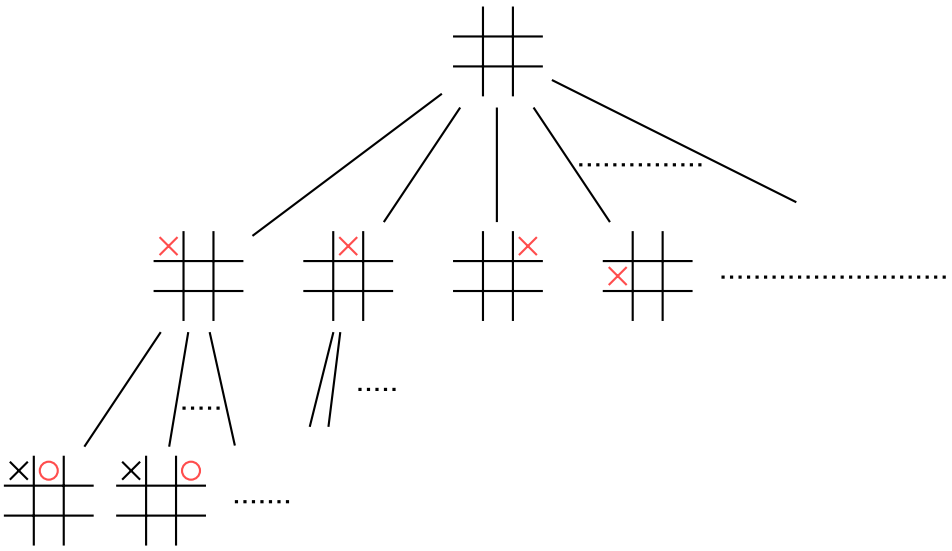
2. Algorithme minimax

2.1. Jeux

Morpion.
Voyons comment les différentes successions de coups possibles d'une partie peuvent être codées sous la forme d'un arbre. Prenons l'exemple du jeu de morpion : sur une grille de 9 cases, il s'agit pour un joueur d'aligner 3 croix et pour son adversaire d'aligner 3 ronds.



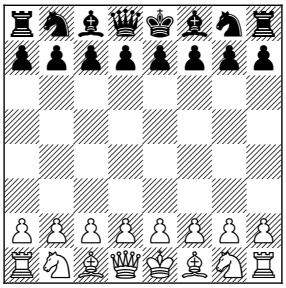
La racine de l'arbre correspond à la position de départ (grille vide), la racine a 9 enfants qui correspondent aux 9 possibilités du premier joueur pour placer sa première croix. Chacun de ces enfants a lui-même 8 enfants qui correspondent aux 8 possibilités (les 8 cases vides restantes) pour le second joueur de placer son premier rond...



Les feuilles de l'arbre (les sommets sans enfants) correspondent soit à une configuration où l'un des joueurs a gagné, soit à une grille remplie sans vainqueur.

Dans une partie il y a au plus 9 coups. Le nombre total de parties est donc inférieur à $9! = 362\,880$ (9 choix pour le premier coup, 8 pour le second, 7 pour le troisième...). Si on tenait compte des symétries, alors il y aurait beaucoup moins de configurations possibles. Ainsi le morpion est un jeu aux possibilités plutôt limitées (mais vous le saviez déjà).

Échecs.
L'arbre des configurations pour décrire toutes les parties d'échecs possibles est immense ! Chaque joueur a 16 pièces, et il y a 64 cases, donc (en théorie) il y a jusqu'à $16 \times 64 = 1024$ possibilités, donc chaque sommet pourrait avoir plus de mille enfants. Dans la pratique on estime qu'un joueur a en moyenne 30 coups possibles et qu'une partie dure 80 coups. Cela fait un total de $30 \times 30 \times \cdots = 30^{80} \simeq 10^{120}$ parties possibles. Ce nombre, 10^{120}, s'appelle le nombre de Shannon, il est gigantesque, pensez qu'il y a « seulement » 10^{80} atomes dans l'univers ! Ainsi il est impossible de calculer toutes les parties d'échecs possibles.

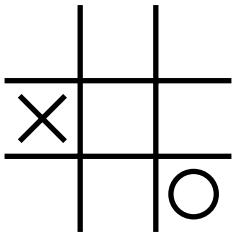


Ce que l'on peut faire avec un ordinateur, c'est chercher toutes les possibilités pour le prochain coup (disons 30 choix), puis pour chacun de ces coups chercher toutes les possibilités de l'adversaire (30 coups pour chacune des 30 possibilités, soit $30 \times 30 = 900$). Si on veut anticiper sur une profondeur de n coups, il faut calculer 30^n possibilités. Par exemple pour $n = 5$ cela fait environ 25 millions de parties à calculer ce qui est accessible. Par contre, pour $n = 7$ cela fait plus de 10 milliards de possibilités à calculer, ce qui n'est plus raisonnable.
Nous allons faire un double travail :

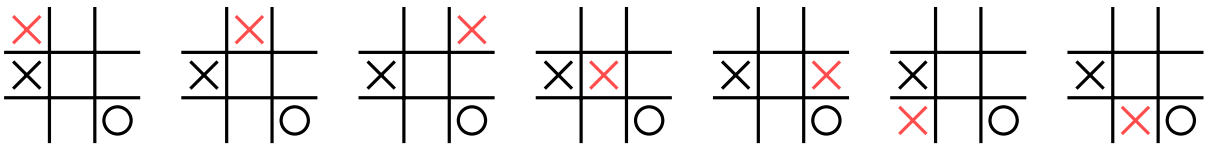
- comment, parmi toutes ces possibilités, sélectionner la suite des coups les plus avantageux (tout en déjouant les stratégies de l'adversaire) ?
- ne pourrait-on pas accélérer le processus en écartant les configurations les plus défavorables ?

2.2. Fonction d'évaluation

Morpion.
Pour pouvoir décider du meilleur coup il faut pouvoir évaluer la qualité de chaque configuration de jeu. Reprenons l'exemple du morpion, je suis dans la situation suivante :

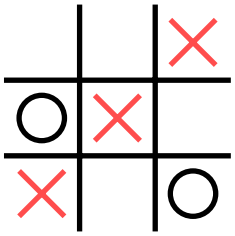


C'est à mon tour de placer une croix, j'ai donc 7 possibilités :

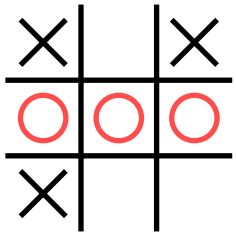


$f(C) = +10$ $f(C) = +2$ $f(C) = +5$ $f(C) = +15$ $f(C) = +10$ $f(C) = +10$ $f(C) = +2$

On va attribuer (ici un peu arbitrairement) une note $f(C)$ à chacune de ces configurations, une valeur haute désignant une configuration qui m'est favorable et me place en position de gagner la partie. Je veux éviter les valeurs basses (généralement négatives) qui sont des configurations favorables à mon adversaire.
Ainsi une ***fonction d'évaluation*** est une fonction f qui à chaque configuration de jeu associe un nombre réel. Plus ce nombre est grand, plus la configuration m'est favorable. Pour une configuration dans laquelle la partie est terminée et que j'ai gagné on peut attribuer la valeur $+\infty$ (ou bien un réel très grand). De même $-\infty$ (ou un réel très négatif) désigne une configuration où c'est mon adversaire qui a gagné.



$f(C) = +\infty$



$f(C) = -\infty$

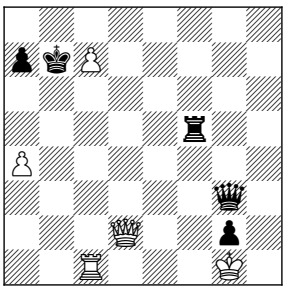
Échecs.
Créer une fonction d'évaluation pertinente est difficile et nécessite une bonne connaissance du jeu. Par exemple aux échecs on attribue souvent une valeur spécifique à chaque pièce comme ceci :

♔	♕	♖	♗	♘	♙
roi	reine	tour	fou	cavalier	pion
∞	10	5	3	3	1

La valeur du roi est infinie car cette pièce ne peut pas être capturée, elle est donc exclue de la fonction d'évaluation. Une fonction d'évaluation basique d'une configuration C est donc :

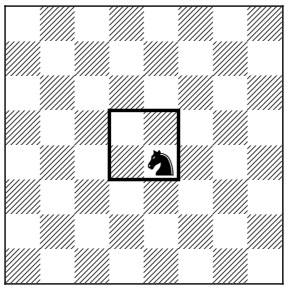
$$f(C) = \text{somme des valeurs des pièces blanches} - \text{somme des valeurs des pièces noires}$$

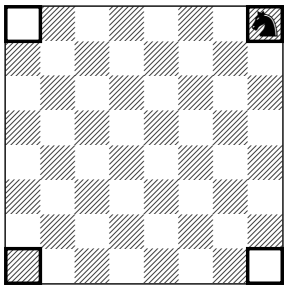
Par exemple, pour la configuration suivante la valeur des pièces blanches est $10+5+1+1=17$, la valeur des pièces noires est aussi 17, la fonction d'évaluation de cette configuration est donc $f(C) = 17-17 = 0$. Ce qui signifie que la situation est équilibrée. (D'ailleurs le prochain joueur qui joue gagne.)



$f(C) = 0$

Mais pour être plus pertinent, il faudrait aussi tenir compte de la position des pièces : par exemple un cavalier sur l'une des 4 cases centrales est dangereux, alors que placé dans un coin il ne sert à rien.



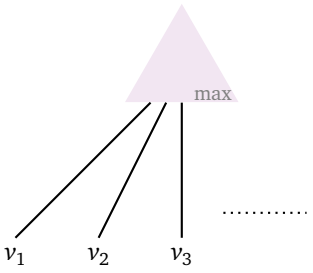


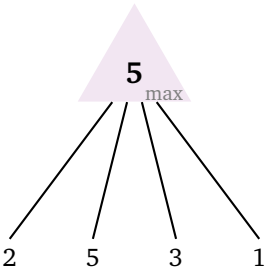
Noter que la fonction d'évaluation est commune aux deux joueurs, sauf que j'ai intérêt à trouver une configuration avec la valeur la plus grande possible alors que mon adversaire veut une configuration avec la valeur la plus petite possible.

Maximum et minimum.
Si c'est à mon tour de jouer et si je ne veux (ou peux) que prévoir un coup à l'avance, alors :

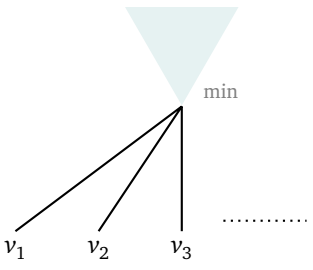
- pour chaque coup numéro i possible, je calcule la configuration C_i obtenue,
- je lui attribue une valeur v_i, en calculant $v_i = f(C_i)$ par la fonction d'évaluation,
- je décide de jouer le coup qui réalise la plus grande valeur des v_i, c'est-à-dire je choisis $i_{\max}$ tel que $v_{i_{\max}} = \max_i(v_i)$.

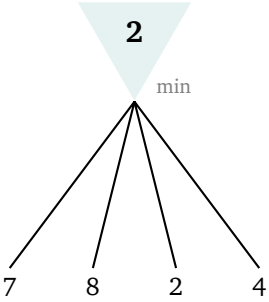
Voici comment on note cette situation en termes d'arbre. La racine sera dessinée sous la forme d'un triangle pointe vers le haut (pour signifier que le joueur cherche à maximiser), les enfants désignent toutes les configurations possibles pour mon prochain coup, ils sont représentés par la valeur v_i issue de la fonction d'évaluation.





Évidemment si c'est à mon adversaire de jouer, il calcule aussi tous ses coups possibles mais cette fois il cherche la configuration ayant la fonction d'évaluation la plus petite possible. Il choisit $i_{\min}$ tel que $v_{i_{\min}} = \min_i(v_i)$. On représente cette fois la racine de l'arbre avec un triangle pointe en bas.





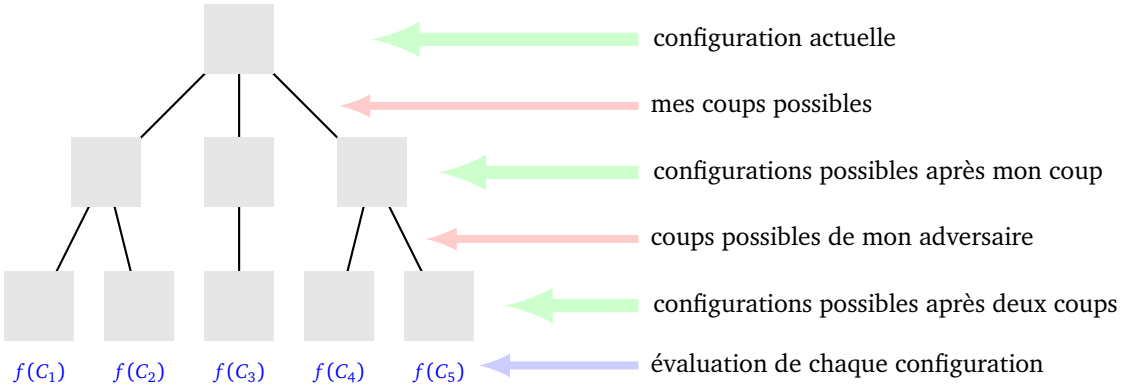
2.3. Algorithme minimax

Avant de détailler l'algorithme donnons les explications générales.

Construction de l'arbre

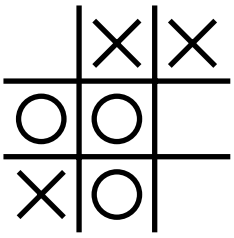
On va décider du meilleur coup à jouer en anticipant sur les n prochains coups. Pour cela on commence par construire l'arbre de toutes les configurations possibles :

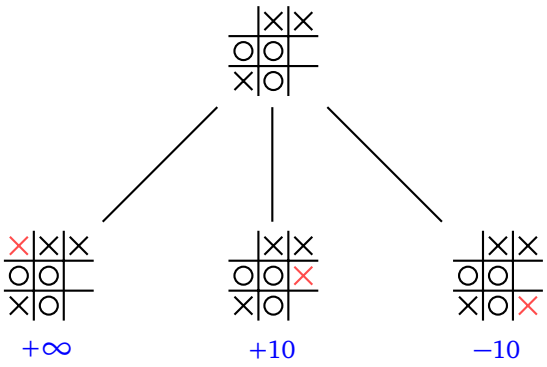
- la racine correspond à la configuration actuelle,
- les enfants correspondent aux coups que je peux jouer ; plus précisément chaque enfant est la configuration obtenue après un des mes coups possibles,
- les petits-enfants correspondent aux configurations après un de mes coups suivi d'un coup de mon adversaire,
- etc.



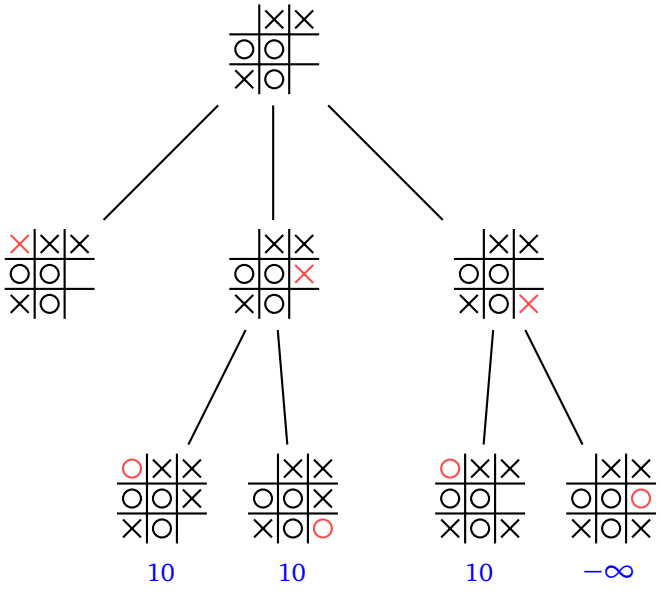
Évaluation sur les feuilles

Les feuilles de notre arbre correspondent aux différentes configurations possibles dans n coups. Nous attribuons une valeur à chacune de ces configurations à l'aide de la fonction d'évaluation. (Nous évaluons seulement les configurations sur les feuilles, il est inutile d'évaluer les configurations intermédiaires.)
Bien sûr, parmi toutes ces configurations j'aimerais arriver à la position ayant la valeur maximale mais mon adversaire fait tout ce qu'il peut pour m'en empêcher.

> **Exemple.**
> Je joue au morpion, voici la situation actuelle.
>
> 
>
> C'est à mon tour de jouer. Voici la liste des coups possibles. Pour chaque nouvelle situation possible, j'attribue une évaluation (ici donnée arbitrairement à titre d'exemple). L'évaluation est haute lorsque la configuration m'est favorable et l'évaluation est basse (et même négative) lorsque la situation est favorable à mon adversaire. L'évaluation $+\infty$ signifie une configuration gagnante (la partie est terminée, j'ai gagné) et $-\infty$ un situation perdante (la partie est terminée, j'ai perdu).



Voici maintenant tous les coups possibles de mon adversaire, ainsi que l'évaluation.



Appliquer l'algorithme minimax

Voici le déroulement de l'algorithme minimax expliqué avec des mots.

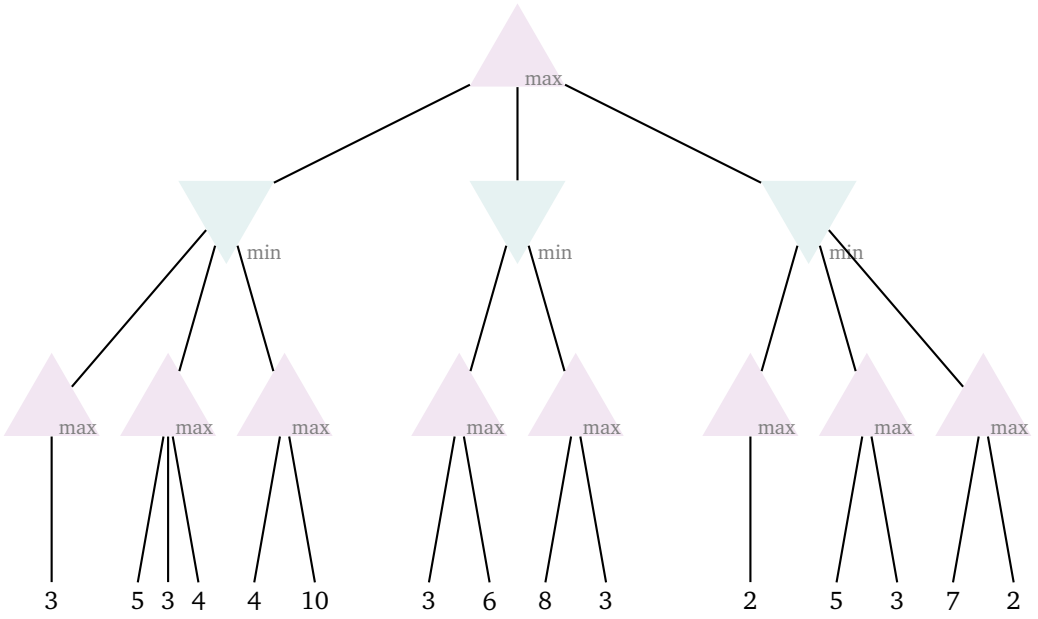
- On part des feuilles (tout en bas de l'arbre), à chaque feuille est associée une valeur obtenue par la fonction d'évaluation. On va remplir l'arbre avec des valeurs, du bas vers le haut.
- Pour chaque sommet de type « max » (avec un triangle vers le haut, qui correspond à mon tour de jouer) la valeur associée est le maximum de la valeur de ses enfants.
- Pour chaque sommet de type « min » (avec un triangle vers le bas, qui correspond à un coup à jouer par l'adversaire) la valeur associée est le minimum de la valeur de ses enfants.
- Partant des feuilles on associe, en remontant du bas vers le haut, une valeur à chaque sommet en terminant par la racine. On mémorise le chemin de la racine vers la feuille qui a permis de réaliser la valeur maximale : cela donne le prochain coup à jouer (ainsi que les coups suivants).

Noter qu'en général je ne peux pas espérer obtenir le maximum possible des valeurs aux feuilles, car mon adversaire va tout faire pour m'en empêcher. Pour cet algorithme, je suppose que mon adversaire fait la

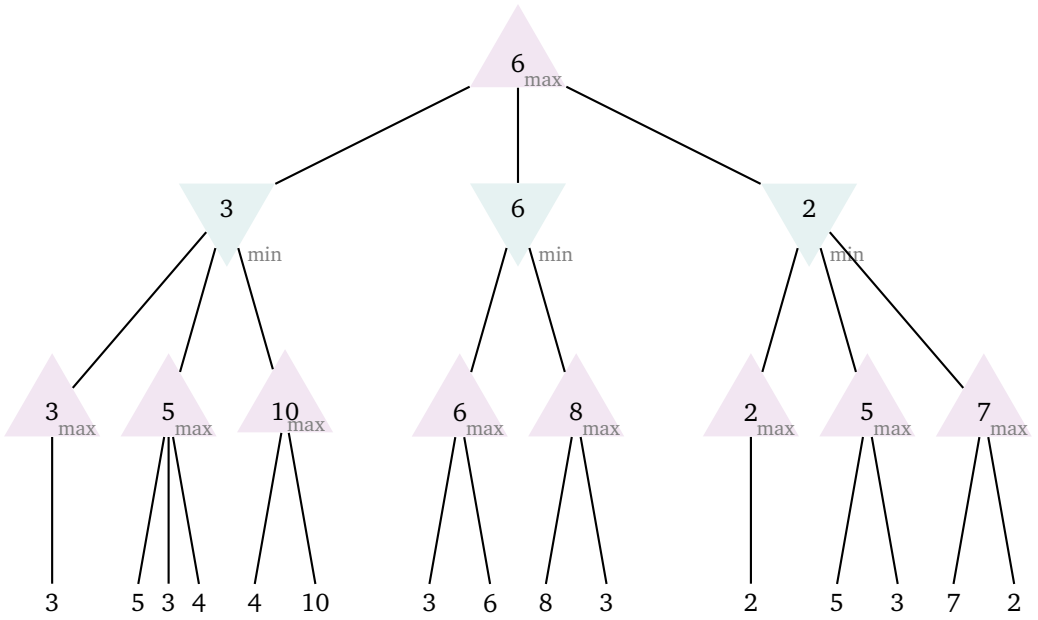
même évaluation que moi, mais sans prévoir plus de coups que moi.

Exemples

Voici l'arbre à remplir, seules les évaluations des positions dans trois coups sont notées.



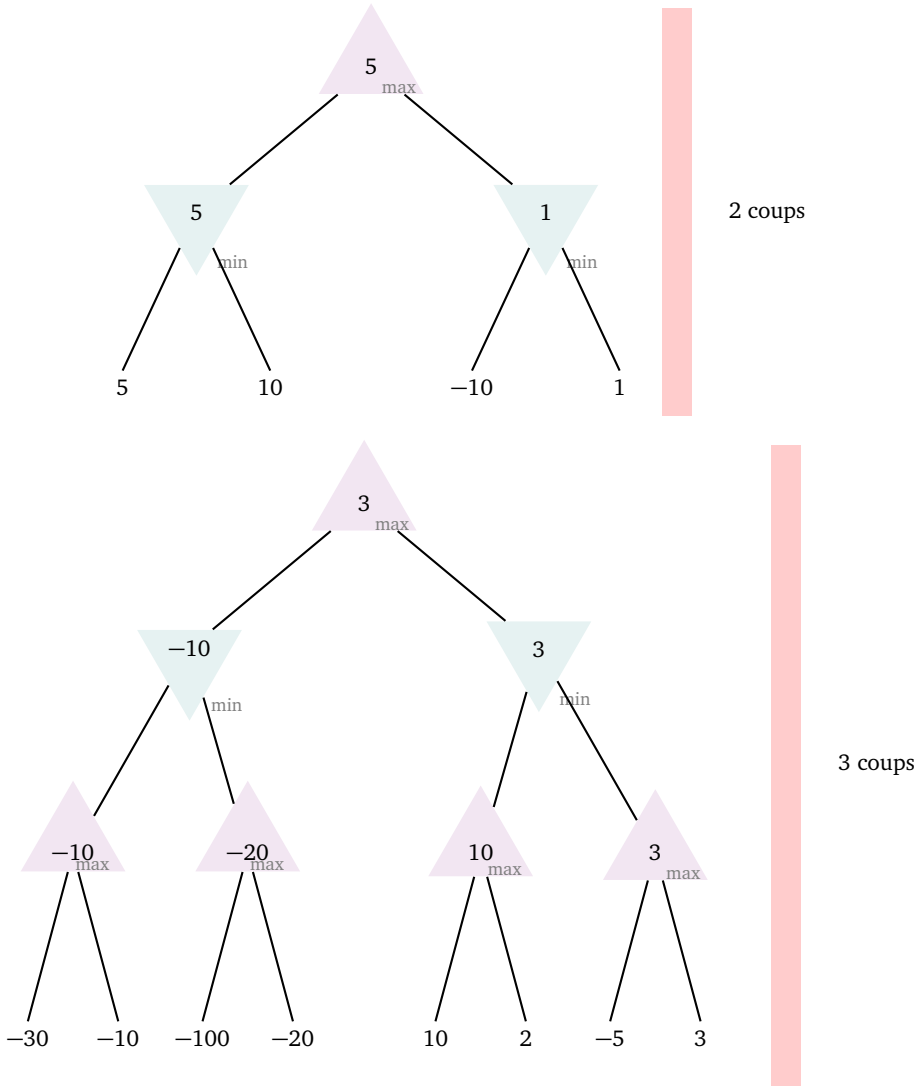
On remplit cet arbre du bas vers le haut, d'abord la ligne des maximums juste au-dessus des feuilles, puis la ligne des minimums et on termine par le maximum tout en haut qui nous donne la meilleure valeur que je puisse espérer.



Ainsi en prévoyant 3 coups à l'avance, je peut atteindre une position de valeur 6. Je ne peux pas espérer atteindre la meilleure position (de valeur 10) sauf si mon adversaire joue mal.

Horizon et sacrifice de la reine. Bien sûr le fait de ne pouvoir prévoir qu'un nombre limité de coups à l'avance ne garantit pas la victoire. Soit l'adversaire anticipe davantage de coups que nous, soit une stratégie qui s'avère bonne à court terme est en fait désastreuse à long terme.

Voici un exemple, dans lequel (arbre du haut) je prévois 2 coups à l'avance et pour lequel je vais choisir la branche de gauche qui semble m'apporter une position favorable (par exemple je prends la reine de mon adversaire), mais si j'avais pu prévoir 3 coups à l'avance (arbre du bas) alors je me serai aperçu que c'était un piège et que la branche de gauche me place dans une position très défavorable.



L'algorithme en détail

Pour l'algorithme minimax nous aurons besoin de deux fonctions :

- la fonction d'évaluation, `evaluation(jeu)`, renvoie une valeur pour une configuration `jeu`. Cette fonction sert à évaluer les configurations sur les feuilles, ce sera notre étape terminale de la fonction récursive.
- Une fonction `joue_un_coup(jeu, coup)` qui renvoie le nouveau jeu correspondant au coup.

Algorithme.
Fonction : `minimax(jeu, profondeur, joueur_maximise)`
Entrée :

- `jeu` : une configuration, c'est-à-dire un sommet d'un arbre.
- `profondeur` : le nombre de coups à anticiper.
- `joueur_maximise` : vrai ou faux, selon que le joueur cherche à maximiser la fonction d'évaluation ou à la minimiser.

Sortie : le score maximal que je peux espérer (ou bien le score minimal que mon adversaire peut espérer).

- Si `profondeur = 0` alors renvoyer `evaluation(jeu)`.
- Si `joueur_maximise` est Vrai :
  - `score_max` $= -\infty$
  - Pour chaque `coup` possible :
    - `nouveau_jeu = joue_un_coup(jeu, coup)`

> — `score = minimax(nouveau_jeu, profondeur-1, Faux)`
> — Si `score > score_max`, faire `score_max = score`.
> — Renvoyer `score_max`
> • Si `joueur_maximise` est Faux :
> — `score_min` = $+\infty$
> — Pour chaque coup possible :
> — `nouveau_jeu = joue_un_coup(jeu, coup)`
> — `score = minimax(nouveau_jeu, profondeur-1, Vrai)`
> — Si `score < score_min`, faire `score_min = score`.
> — Renvoyer `score_min`

L'appel initial se fait par `minimax(jeu, profondeur, Vrai)` à partir de la position `jeu` actuelle et du nombre de coups `profondeur` que je veux anticiper. On verra dans l'implémentation concrète comment renvoyer la liste des coups qui permettent de réaliser ce maximum (ou minimum).

2.4. Avec Python

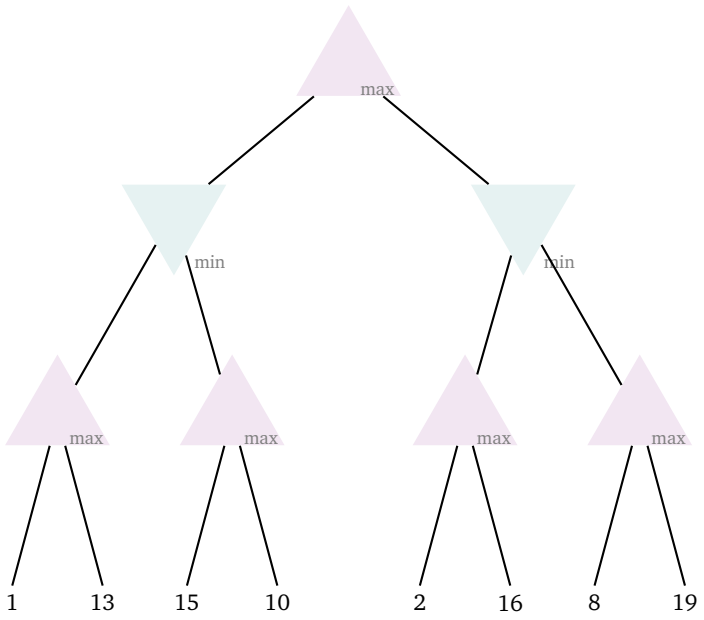
Préliminaires :

- Une fonction `evaluation(jeu)` ; pour nos tests ci-dessous cette fonction renvoie une valeur aléatoire (entre 1 et 20).
- Une fonction `joue_un_coup(jeu, coup)` qui calcule et renvoie la nouvelle configuration du jeu après le coup. Pour nos tests elle renvoie le jeu sans modifications.

```
def minimax(jeu, prof, coups_prec, joueur_maximise):
    M = 3 # Ici M coups possibles à chaque fois
    if prof == 0:              # On est à une feuille
        score = evaluation(jeu)
        print('Coups :',coups_prec, 'score :', score)
        return (score, coups_prec)
    else:
        if joueur_maximise:
            score_max = -math.inf
            for coup in range(M):
                nouv_jeu = joue_un_coup(jeu, coup)
                score, nouv_coups_prec =
                            minimax(nouv_jeu,prof-1,coups_prec+[coup],False)
                if score > score_max:
                    score_max = score
                    coups_max = nouv_coups_prec
            return (score_max, coups_max)
        else:
            score_min = +math.inf
            for coup in range(M):
                nouv_jeu = joue_un_coup(jeu, coup)
                score, nouv_coups_prec =
                             minimax(nouv_jeu,prof-1,coups_prec+[coup],True)
                if score < score_min:
                    score_min = score
                    coups_min = nouv_coups_prec
```

```
            return (score_min, coups_min)
```

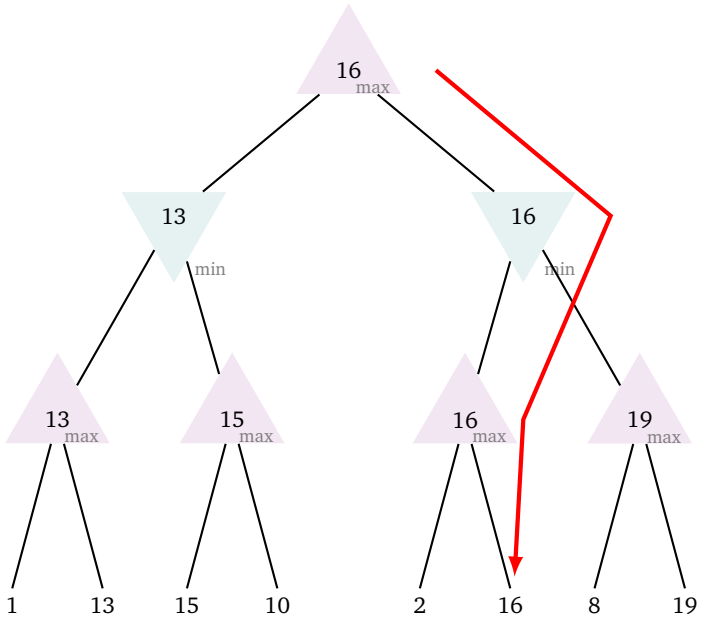
Avec `jeu = None` (car on ne teste pas notre algorithme sur un vrai jeu), l'appel de la fonction est par exemple : `minimax(jeu, 3, [], True)` afin d'anticiper sur les 3 prochains coups. Ici la fonction renvoie le meilleur score attendu (comme l'algorithme vu précédemment) mais aussi la suite des coups qui permet d'atteindre ce maximum.
Voici un exemple correspondant à l'arbre :



Il y a deux enfants à chaque branche ($M = 2$) et le nombre de coups anticipés est 3.

```
Coups : [0, 0, 0] score : 1
Coups : [0, 0, 1] score : 13
Coups : [0, 1, 0] score : 15
Coups : [0, 1, 1] score : 10
Coups : [1, 0, 0] score : 2
Coups : [1, 0, 1] score : 16
Coups : [1, 1, 0] score : 8
Coups : [1, 1, 1] score : 19

Coups à jouer  (16, [1, 0, 1])
```



La valeur « minimax » est bien 16 et s'obtient par le chemin `[1, 0, 1]` c'est-à-dire branche de droite (1), puis branche de gauche (0) puis branche de droite (1 de nouveau).

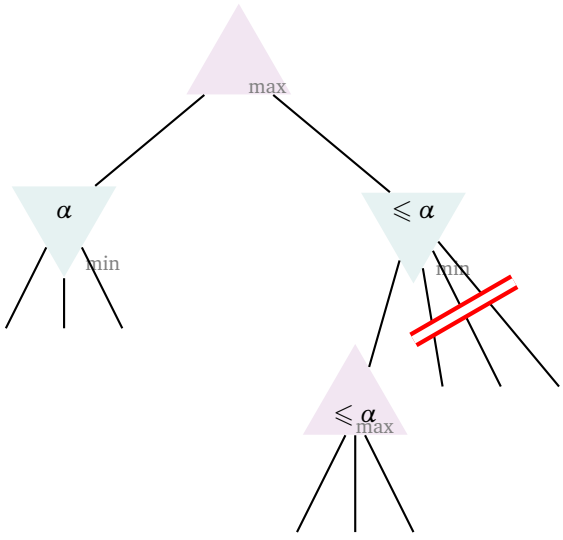
3. Élagage alpha-beta

3.1. Idée

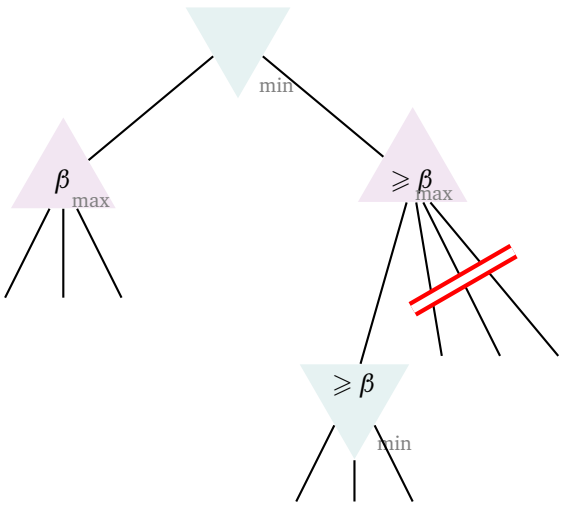
Rien ne sert de prévoir toutes les successions de coups lorsque de toute façon on a déjà trouvé mieux. Cela permet de réduire drastiquement le nombre de coups à évaluer. Si à chaque position il y a n possibilités et que l'on veut prévoir les d prochains coups, alors la complexité est en $O(n^d)$. La méthode dite « élagage alpha-beta » a pour complexité en moyenne $O(n^{\frac{d}{2}})$. Cela signifie qu'au lieu d'avoir à étudier n branches à chaque sommet, il y a seulement $\sqrt{n}$ branches en moyenne à considérer. Par exemple si à chaque fois il y a 36 coups possibles, avec l'élagage alpha-beta il n'y a plus, en moyenne, que 6 branches à étudier.

3.2. Cas de base

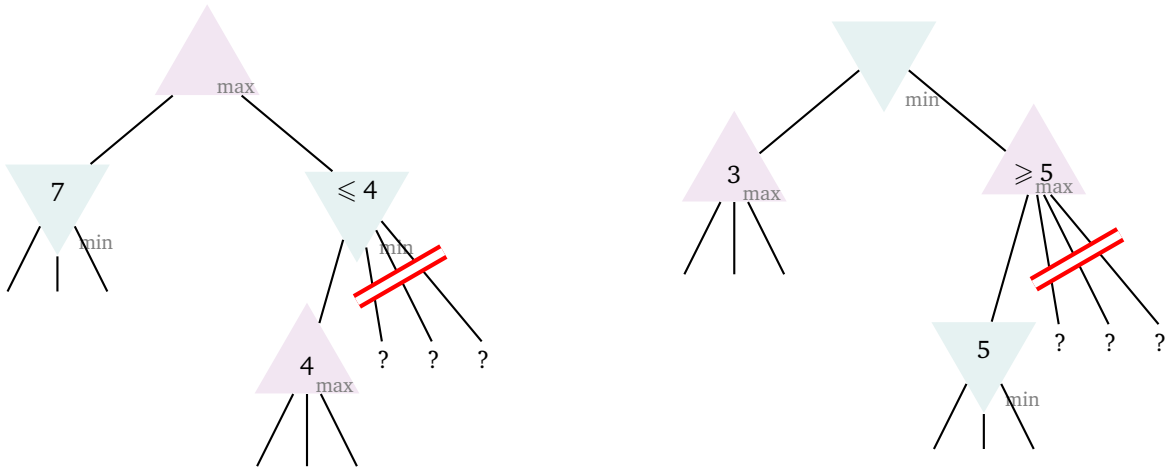
Coupure α



Coupure β

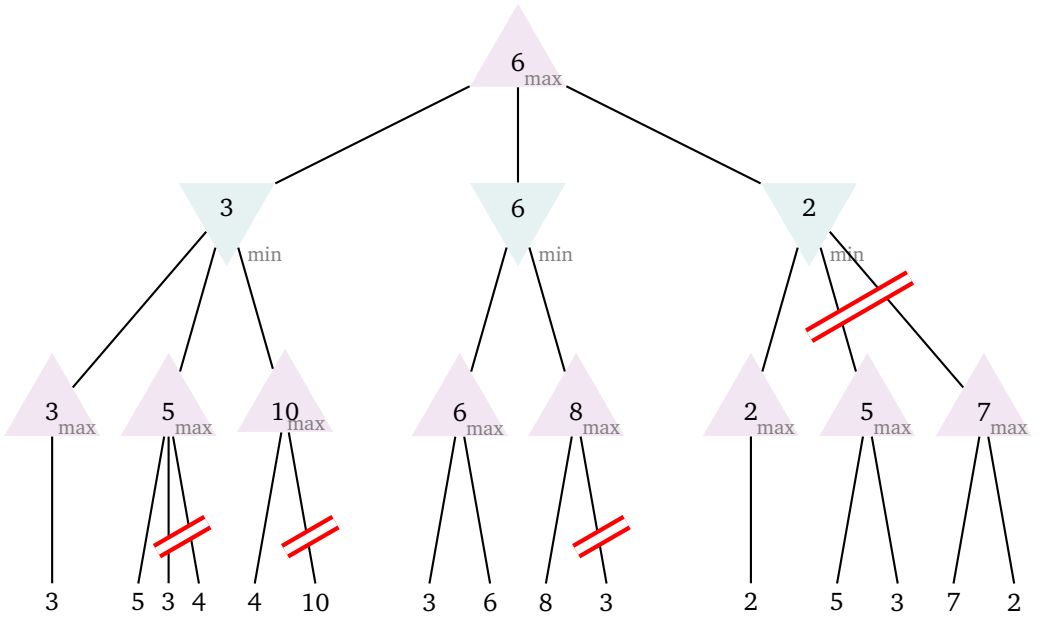


Exemple.
Sur les deux exemples ci-dessous, vous pouvez compléter la valeur finale dans le triangle du haut sans avoir besoin de connaître les valeurs (marquées par des « ? ») issues des branches coupées.

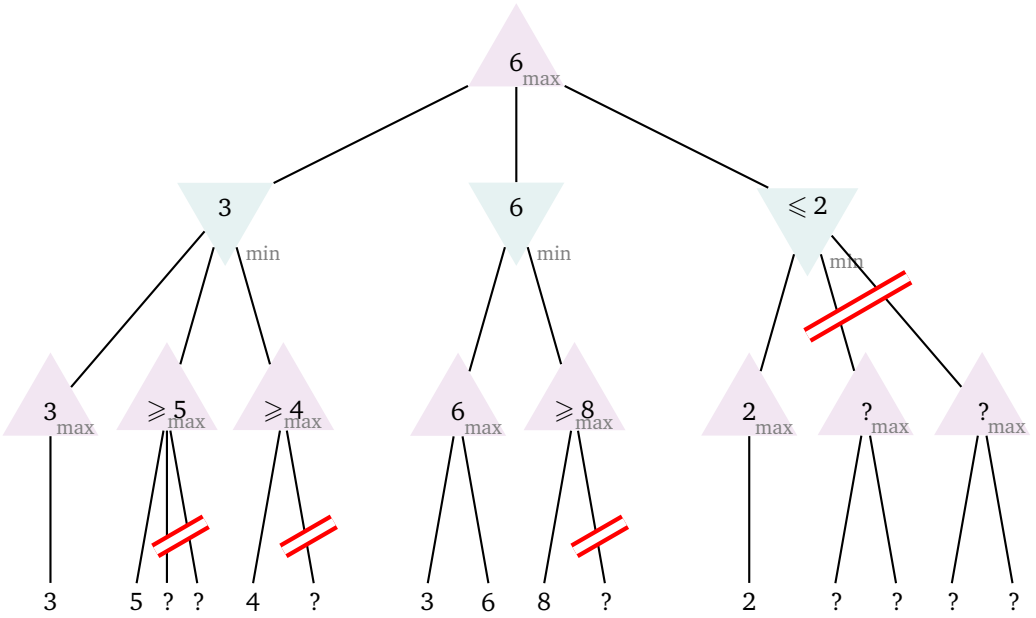


3.3. Exemple

Voici les branches que l'on pourrait élaguer pour l'exemple rencontré précédemment.



Cela signifie que l'on peut éviter 8 évaluations sur les 15. Ces évaluations non faites sont les feuilles marquées par des « ? » ci-dessous. Quelles que soient les valeurs qui remplacent ces « ? » le résultat final de l'algorithme (la valeur dans le triangle supérieur) est le même.



3.4. Algorithme

L'algorithme d'élagage est une adaptation de l'algorithme minimax. La fonction `alphabeta()` renvoie la même valeur que `minimax()` mais avec moins de calculs. α et β sont deux paramètres (on aura toujours $\alpha \leqslant \beta$). Lorsqu'on appelle la fonction la première fois on pose $\alpha = -\infty$ et $\beta = +\infty$ (ou bien des très grands nombres).

Algorithme.
Fonction : `alphabeta(jeu, profondeur, alpha, beta, joueur_maximise)`

- Si `profondeur = 0` alors renvoyer `evaluation(jeu)`.
- Si `joueur_maximise` est Vrai :
  - `score_max` $= -\infty$
  - Pour chaque `coup` possible :
    - `nouveau_jeu = joue_un_coup(jeu, coup)`
    - `score = alphabeta(nouveau_jeu, profondeur-1, alpha, beta, Faux)`
    - Si `score > score_max`, faire `score_max = score`.
    - Si `score` $\geqslant$ `beta`, arrêter ici la boucle « pour ».
    - `alpha = max(alpha, score)`
  - Renvoyer `score_max`
- Si `joueur_maximise` est Faux :
  - `score_min` $= +\infty$
  - Pour chaque `coup` possible :
    - `nouveau_jeu = joue_un_coup(jeu, coup)`
    - `score = alphabeta(nouveau_jeu, profondeur-1, alpha, beta, Vrai)`
    - Si `score < score_min`, faire `score_min = score`.
    - Si `score` $\leqslant$ `alpha`, arrêter ici la boucle « pour ».
    - `beta = min(beta, score)`
  - Renvoyer `score_min`

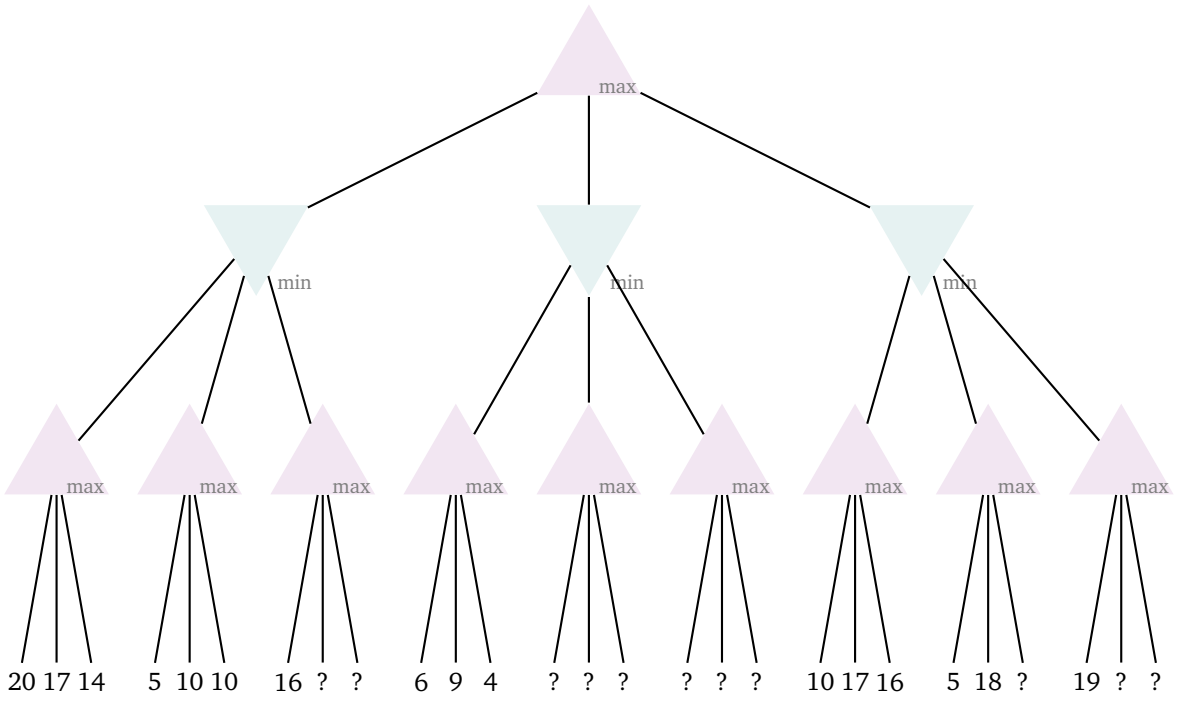
L'élagage α correspond à la première partie (lorsque le joueur chercher à maximiser le score), l'élagage β à la seconde partie (lorsque le joueur chercher à minimiser le score).

3.5. Avec Python

L'implémentation ne pose pas de difficulté particulière, il suffit de modifier le code précédent lorsque le joueur cherche à maximiser (faire une modification similaire pour minimiser) :

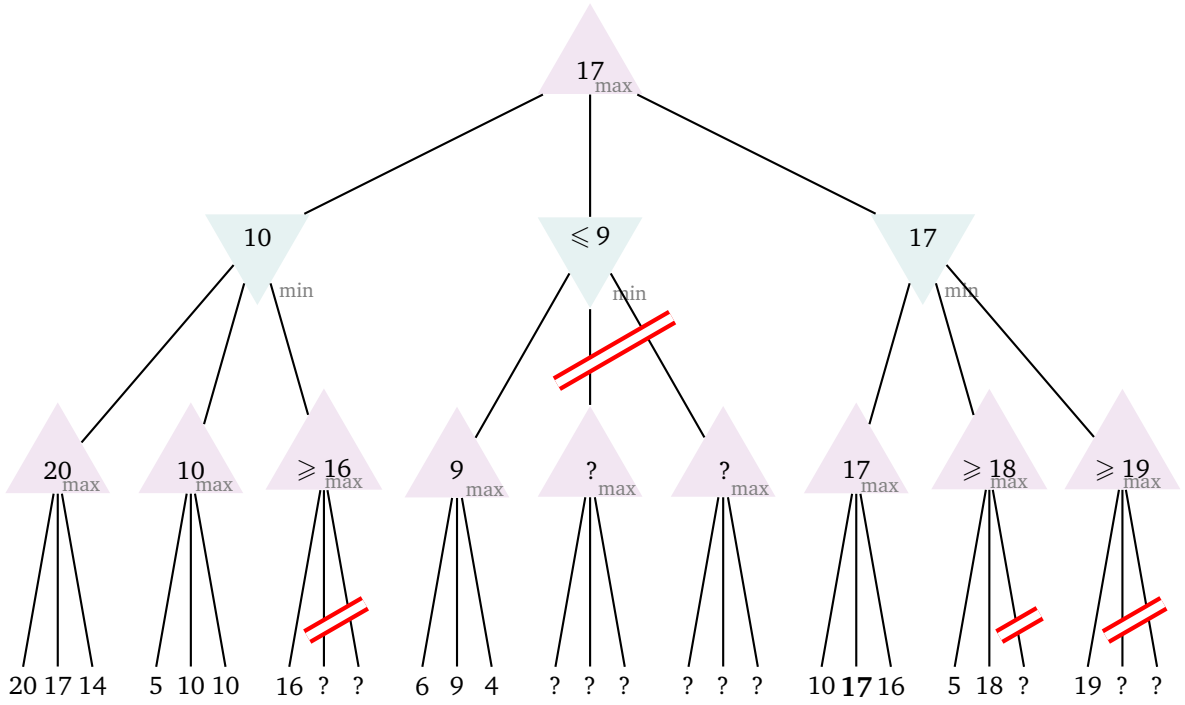
```
if score > score_max:
    score_max = score
    coups_max = nouv_coups_prec
if score >= beta:  # Coupure alpha
    break          # On arrête de chercher les coups suivants
```

Voici un exemple avec une profondeur de 3 et $M = 3$ coups possibles à chaque fois. Sur les $3^3 = 27$ possibilités, l'élagage alpha-beta permet ici d'évaluer seulement 16 configurations.




```
Coups : [0, 0, 0] score : 20
Coups : [0, 0, 1] score : 17
Coups : [0, 0, 2] score : 14
Coups : [0, 1, 0] score : 5
Coups : [0, 1, 1] score : 10
Coups : [0, 1, 2] score : 10
Coups : [0, 2, 0] score : 16
Coups : [1, 0, 0] score : 6
Coups : [1, 0, 1] score : 9
Coups : [1, 0, 2] score : 4
Coups : [2, 0, 0] score : 10
Coups : [2, 0, 1] score : 17
Coups : [2, 0, 2] score : 16
Coups : [2, 1, 0] score : 5
Coups : [2, 1, 1] score : 18
Coups : [2, 2, 0] score : 19

Coups à jouer  (17, [2, 0, 1])
```


17 max
10 min
⩽ 9 min
17 min
20 max
10 max
⩾ 16 max
9 max
? max
? max
17 max
⩾ 18 max
⩾ 19 max
20 17 14
5 10 10
16 ? ?
6 9 4
? ? ?
? ? ?
10 17 16
5 18 ?
19 ? ?