Algorithmes pour les jeux

I) Résolution de jeux à 1 joueur

II) Jeux à 2 joueurs et IA

a) Algorithme minimax

b) Améliorations

I

On s'intéresse ici à la résolution de jeux et casse-tête à un joueur.

Un exemple classique : Sudoku

On a une manière brutale et naïve de résoudre un Sudoku

Ex :

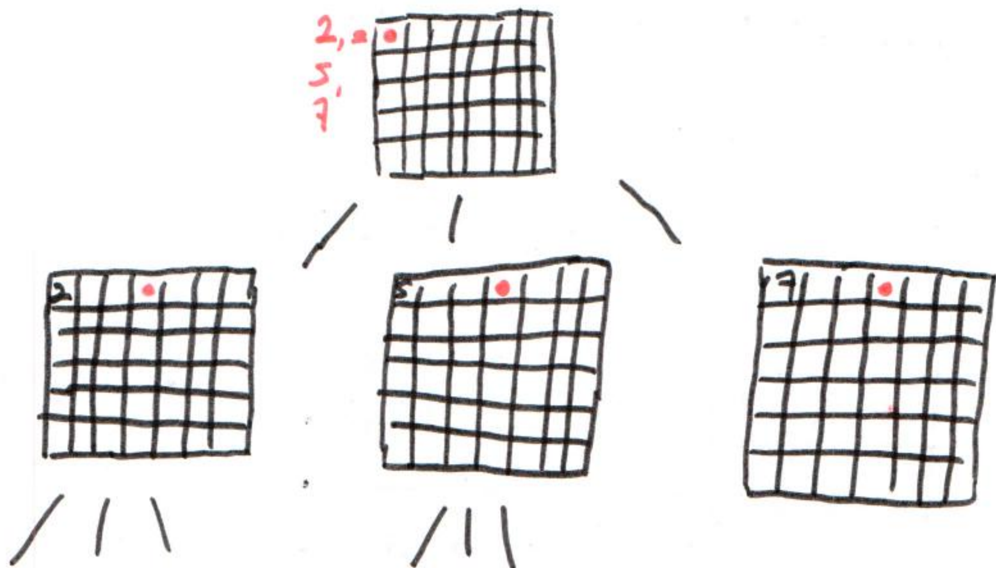
●	4	8		9			3	6
		1	7			2		
3			6		2	4	1	
	9				7	8		
6	7		8		9		4	3
		5	3				7	
	3	6	9		5			4
		4			8	3		
7	1			4		6	5	

- On explore la grille ligne par ligne, jusqu'à trouver la première case non vide : ●
- On cherche quelles sont les valeurs autorisées dans cette case suivant les règles du sudoku : 2, 5, 7.

- On va tester la valeur 2, puis recommencer le procédé en allant regarder la case suivante.

- Si on peut parcourir toute la grille et que pour chaque case vide, il existe une valeur autorisée ⇝ le résultat de l'algorithme fournit une résolution de la grille
- Si pour une certaine case vide, il n'existe plus de valeurs autorisées, il y a une erreur → on revient alors au choix précédent et on modifie la case par la valeur autorisée suivante.

C'est un algorithme de type " backtracking" (retour en arrière)

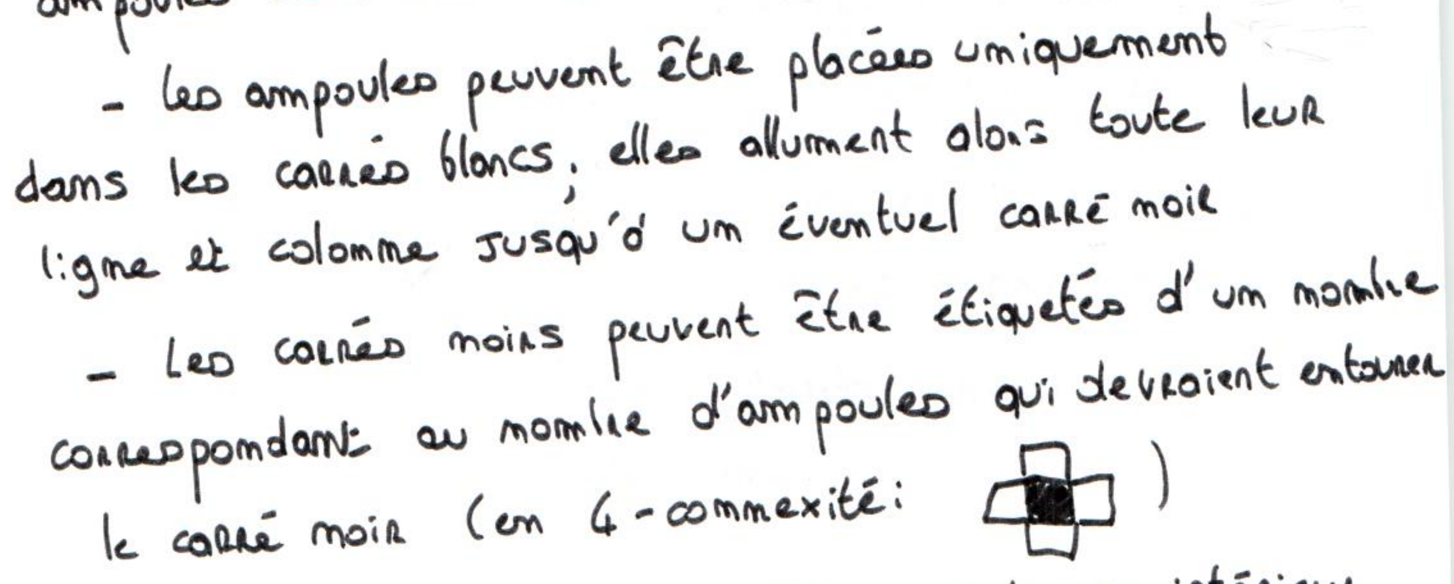


On parcourt ainsi toutes les possibilités de remplir la grille ⇝ C'est en fait un parcours en profondeur d'un arbre, dans lequel on cherche la branche qui mène à une solution

Dans un Sudoku moyen : ~50 cases vides, et en moyenne 3 possibilités par case : 3^{50} grilles à explorer, c'est beaucoup! La résolution d'un Sudoku "difficile" peut prendre jusqu'à quelques minutes

Un autre exemple : le jeu de light up

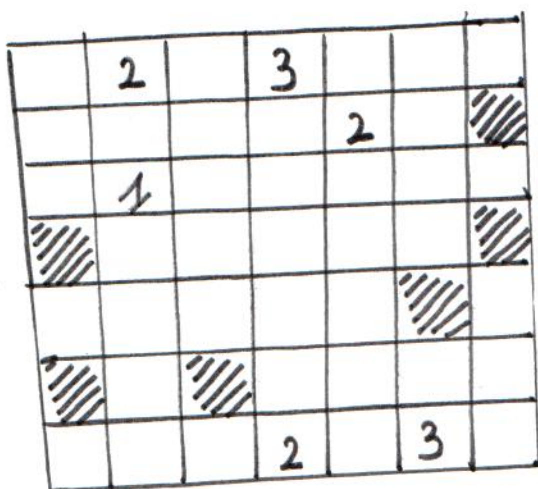
Light up est un puzzle consistant à illuminer une grille formée de carrés noirs et blancs en ajoutant des ampoules selon les règles suivantes :

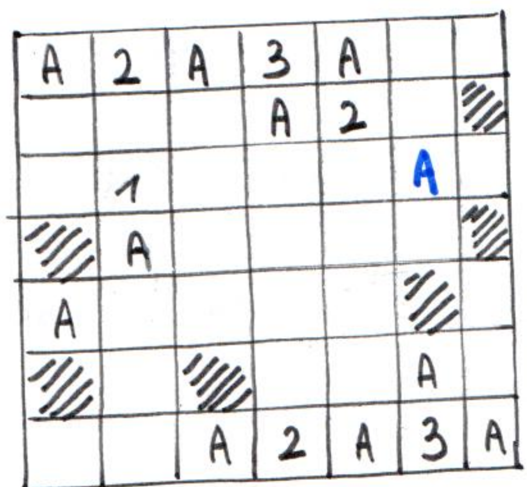


- les ampoules peuvent être placées uniquement dans les carrés blancs ; elles allument alors toute leur ligne et colonne jusqu'à un éventuel carré noir
- Les carrés noirs peuvent être étiquetés d'un nombre correspondant au nombre d'ampoules qui devraient entourer le carré noir (en 4-connexité :)

  Ce nombre d'ampoules doit être exact, pas inférieur ni supérieur. Si un carré noir n'est pas numéroté, le nombre d'ampoules l'entourant peut être quelconque.
- Il ne doit pas y avoir d'ampoule sur un carré blanc déjà éclairé par une autre ampoule.

Ex :





Implémentation d'un algorithme de résolution par backtracking

- On va utiliser une structure de board qui comprendra le nombre de lignes, le nombre de colonnes, et un tableau de caractères

- On va utiliser une structure d'état pour stocker dans un tableau les positions des lampes dans le board.

```
typedef struct state{
        int nb-light;
        int light-pos;
}
```

On devra écrire des fonctions

- print-board (board b) ⇝ affichage du jeu

- add-light qui ajoute une lumière supplémentaire dans le jeu et le vecteur d'états; puis fill-board (board b, state s) qui éclaire le tableau en fonction des lampes présentes

- check (board b, state s) qui vérifie si l'état actuel des lampes respecte les contraintes du jeu ⇝ 0 si non, 1 si oui;

- is-solved (board b, state s) qui détermine si le jeu est résolu (les contraintes sont respectées + tout est éclairé)

solve (board b, state s) qui résout le jeu par backtracking

```
int size = (b.rows) * (b.cols +1);
if (check(b,s) ==0 )
    Return 0;
if (is_solved (b,s) == 1 ){
    print_board(b);
    return 1;
}
for (int k = 1; k<= size; k++) {
    if (b.game[k] == ' ') {
        temp = copy_board(b);
        s= add_light (b,s,k);
        b = fill_board (b,s);
        if (solve(b,s) == 1)
            return 1;
        s.nb_light--;
        b = fill_board(temp,s);
    }
}
return 0
}
```

Vous trouverez dans main.c une fonction qui permet de créer un board de jeu depuis un fichier.txt. Vous pouvez tester votre solver sur les divers niveaux proposés

→ sur un niveau "difficile", la résolution par backtracking peut être coûteuse.

En fait, le temps d'exécution dépend de la position de la solution dans l'arbre du jeu : si il est situé dans les premières branches, la résolution sera rapide ; et lente si c'est une des dernières grilles explorées..

Question Comment améliorer cet algorithme?

① Placer les lumières que l'on peut placer à coup sûr.

Exemples:

	A	
A	4	A
	A	

A	
3	A
A	

		A	
A		2	A
2	A		

N_b = nombre de carrés noirs

$i \in \{1, \dots, N_b\}$

$Need_i$ = nombre dans le carré (dont on a besoin)

$Space_i$ = nombre de carrés blancs pouvant accueillir une ampoule autour de i.

$Have_i$ = nombre d'ampoules qu'on a autour.

SpP_i = {carrés blancs admissibles autour}

Si: $Space_i + Have_i = Need_i$ ⇝ on met une ampoule dans tous les éléments de SpP_i

② Eliminer les cases qui ne pourront pas accueillir d'ampoule.

Ex:

	x	
x	0	x
	x	

x	1	A

etc.

Si: $Have_i = Need_i$ ⇝ toutes les positions restantes de SpP_i peuvent être marquées d'une croix.

Exemple: Si on applique ce pattern ① - ② deux fois:

Le jeu est presque résolu en 2 étapes!

③ Si il y a une case non allumée isolée, on peut y placer une ampoule

N_W = nombre de carrés blancs

$J \in \{1, \dots, N_W\}$

Si $Source_J = 1$, on peut placer une ampoule dans la seule position valable; $Source_J$ correspond au nombre de carrés non éclairés autour de J, incluant J

Exemple:

Il y a d'autres patterns possibles d'amélioration...

Chiu - Chou - Yang - Yen ont publié en 2010 un article " A simple and rapid Lights-up solver " dans lequel ils présent un algorithme efficace du puzzle.

Ils utilisent notamment ①, ② et ③ ainsi que d'autres patterns; ainsi qu'une méthode de parcours arborescente pour résoudre les zones blanches non encore éclairées.

Le fichier lightup.c contient une implémentation de cet algorithme ⇝ vous pouvez ainsi tester et comparer l'efficacité de votre algorithme de backtracking par rapport à celui-ci.

<u>Remarque</u>: En fait, la résolution rapide de gros puzzles de Light Up est un problème qui intéresse encore des chercheurs à l'heure actuelle

⇝ Jeu qui apparaît comme un problème aux " Computer Olympiads " en 2010.

(II) On étudie un jeu à deux joueurs à somme nulle
(somme des gains & pertes égale à 0) où :
- chaque joueur joue dans l'objectif de gagner
- les joueurs jouent chacun leur tour

Exemples: TicTacToe, Dames, Jeu de Go, Échecs, Puissance 4...

- On veut programmer une IA capable de jouer à un de ces jeux (contre une autre IA ou un joueur humain)

Difficultés:
- Il faut une IA capable de s'adapter à une position de jeu changeante
- Il faut une IA capable de jouer en fonction de la stratégie de son adversaire.

Idée (Naïve) • Construire l'arborescence des coups réalisables

Ex:

← Ici, le choix 1 est préférable pour les rouges.

- Évaluer chaque position en fin de branche

⇝ fonction qui nous permet d'évaluer et de mesurer quel joueur a l'avantage à une position de jeu donnée

Il faut donc un moyen cohérent, dépendant des règles du jeu de définir une fonction

Exemple : Pour les échecs,

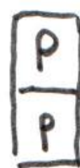
$$
\begin{aligned}
f(p) = {} & 200(N_K - N_{K'}) + 9(N_Q - N_{Q'}) + 5(N_R - N_{R'}) \\
& + 3(N_B + N_N - N_{B'} - N_{N'}) + N_P - N_{P'} \\
& - (D - D' + S - S' + I - I')
\end{aligned}
$$

où $N_\bullet$ désigne le nombre de $\bullet$ chez le joueur blanc (resp. $N_{\bullet'}$ chez le joueur noir) et

K king Q queen R rook B bishop N knight

P pawn

D = nombre de pions doublés



S = nombre de pions bloqués

I = nombre de pions isolés

M = mobilité du joueur

Cette fonction renvoie un nombre flottant : plus il est grand (positif), plus le joueur blanc a l'avantage ; plus il est petit (négatif), plus le joueur noir a l'avantage.

Exemple: Chess Board

	A	B	C	D	E	F	G	H
8							K'	
7						R'		
6					B'			p'
5		p'						
4		B		Q'	B	Q		
3	p						P	
2						p		K
1								

$$f(p) = 200(1-1) + 9(1-1) + 5(0-1) + 3(2+0-1-0) + 3 - 3 - (0-2+0-1+1-1)/2 + (37-46)/10$$
$$\simeq -1,4$$

On peut donc supposer qu'à cette position, le joueur noir a légèrement l'avantage. Cependant, si blanc bouge son fou en H7 (pour mettre en échec le roi), il prend l'avantage en capturant la dame au coup suivant..

Ceci est appelé un effet d'horizon.

- Choisir le chemin permettant d'atteindre la meilleure position finale, et jouer les coups menant à la position souhaitée.

Problème majeur: Complexité spatiale beaucoup trop grande; arbre de jeu immense.

Exemples: Tic Tac Toe : $< 3^9 \sim 20\,000$

Echecs : $\sim (30 \times 30)^{40} \simeq 10^{120}$ ← Ce nombre est appelé nombre de Shannon

(Moyenne de 40 coups par partie, en moyenne 30 choix possibles)

Jeu de Go : $\sim 10^{600}$

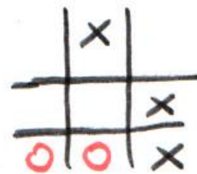
→ impossible à gérer sans amélioration !

Puissance 4 : Au max $3^{42} \sim 10^{20}$ parties possibles puisque chaque case peut recevoir 3 états : Rouge / Jaune / Vide. Meilleure estimation : $1{,}6 \times 10^{13}$, calculée par ordinateur en retirant les configurations impossibles (Jaune / Vide / Jaune) et en éliminant toutes les cellules vides au dessous d'un cas gagnant.

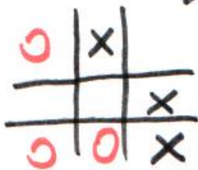
Pour éviter cette explosion combinatoire, on peut se limiter à parcourir l'arbre jusqu'à une certaine profondeur

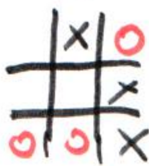
Exemple: TicTacToe avec profondeur 2

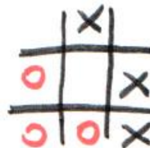
Prof. 0



Prof. 1







Prof. 2

, × gagne

STOP

Principe de résolution :

- on va étudier l'arbre de jeu, et trouver la suite de coups optimale de la position initiale à un état final
  → sélection d'une branche de l'arbre de jeu.

- Estimation d'une position p :
  Soit $\mathcal{P}$ l'ensemble des positions légales

et $\mathcal{P}^*$ l'ensemble des positions légales terminales

Idéalement : fonction $h^* : \mathcal{P}^* \rightarrow \{-\infty, 0, +\infty\}$

($-\infty$: Joueur 2 gagne ; 0 : Match nul ; $+\infty$: Joueur 1 gagne)

qui détermine quel joueur gagne.

Pour obtenir une telle fonction, on utilise une estimation

$h : \mathcal{P} \rightarrow \mathbb{R}$ fonction d'évaluation

Cette fonction devra être d'autant plus fiable que p est proche d'une position terminale

Mais h est seulement une heuristique, et on ne peut pas prévoir les effets d'horizon

→ Il faut donc trouver un bon compromis entre la profondeur de jeu choisie et la fonction d'évaluation

Plus la profondeur est élevée, plus la fonction d'évaluation a des chances d'être efficace en s'approchant des positions terminales ; mais plus la taille en mémoire sera importante.

a) Algorithme mimimax

Hypothèses :

- Les deux adversaires utilisent la même fonction d'évaluation et jouent pour gagner.
- Le Joueur 1 cherche à maximiser son évaluation.
- Le Joueur 2 cherche à mimimiser son évaluation

Joueur 1 = MAX   Joueur 2 = MIN

mimimax (profondeur m, position p, Joueur J)

- Si p est terminale
  Return $h^*(p)$
- Si $m = 0$ ← Om a atteint la profondeur maximale
  Return $h(p)$
- Sinon, soit $p_1, \dots, p_m$ les m positions accessibles depuis p
  - Si $J = MAX$
    Return $\max_{1 \leq i \leq m}$ mimimax$(m-1, p_i, MIN)$
  - Si $J = MIN$
    Return $\min_{1 \leq i \leq m}$ mimimax$(m-1, p_i, MAX)$

Voici um exemple d'appel de mimimax (3, p, MAX)

Inconvénients : (1) Complexité élevée

$\leadsto O(c^m)$ avec c coups légaux en moyenne pour chaque position et profondeur m.

(2) Plus problématique : effets d'horizon ; pour minimax (m, p, J) il se peut que p_{m+1} soit perdante alors que p_m est favorable

b) Améliorations

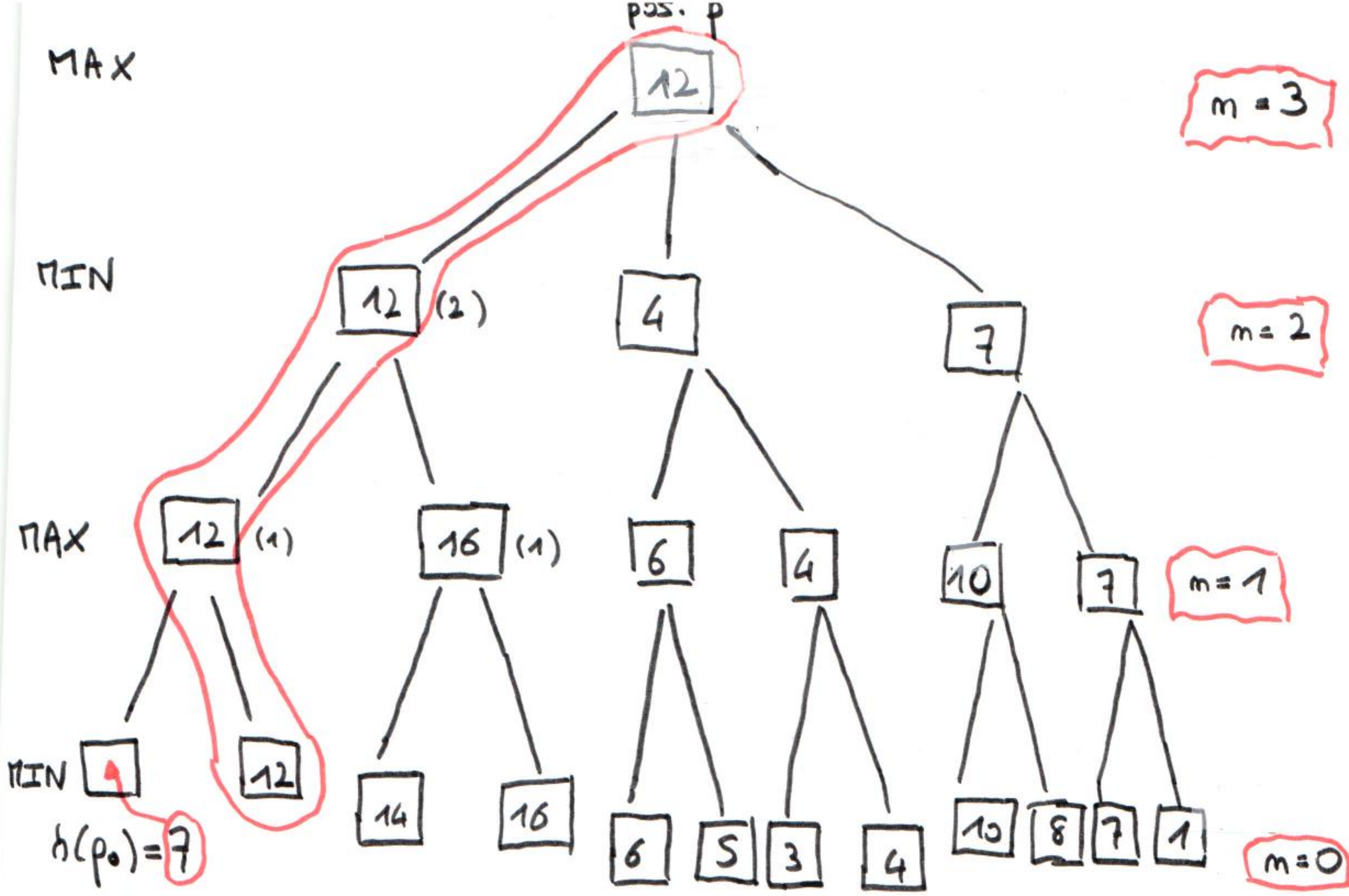
Première idée : on va éviter de parcourir les positions superflues de l'arbre de jeu.

$\leadsto$ élagage de l'arbre de recherche

Cet algorithme s'appelle alpha-beta (α-β) puisque selon le joueur, on va réaliser deux types d'élagage dans l'arbre.

Remarque : Cela n'enlève pas l'effet d'horizon, mais permet d'augmenter la profondeur en réduisant le nombre de positions à explorer.

On donne maintenant deux exemples d'élagage de positions superflues : { une coupure α.
une coupure β

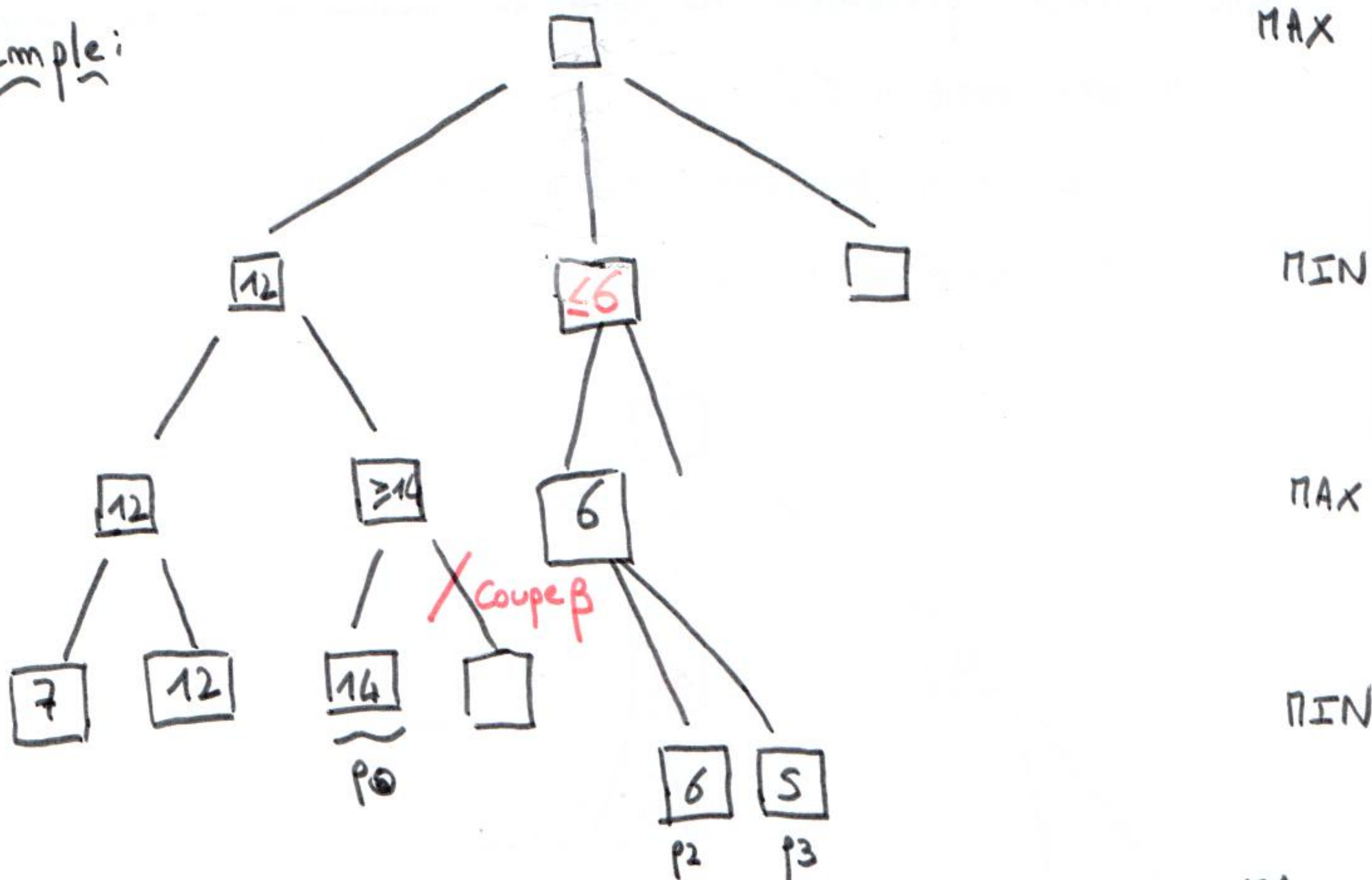


(1) MAX va choisir la valeur maximale renvoyée par l'évaluation de tous les coups possibles de MIN, donc 12

(2) MIN va choisir la valeur minimale renvoyée par l'évaluation de tous les coups possibles de MAX, donc 12

- Em remontant ainsi jusqu'á la position imitiale, om obtient 12, qui est l'évaluation du deuxième coup (dernier étage) ⇝ pour arriver á cette position, MAX devra donc jouer le premier coup et suivre le chemim en rouge.

Exemple :



- Une fois déterminé que $h(p_0) = 14$, on voit que MAX va choisir $\max(14, h(p_1)) \geq 14$, donc la valeur de son parent sera ≥ 14 !

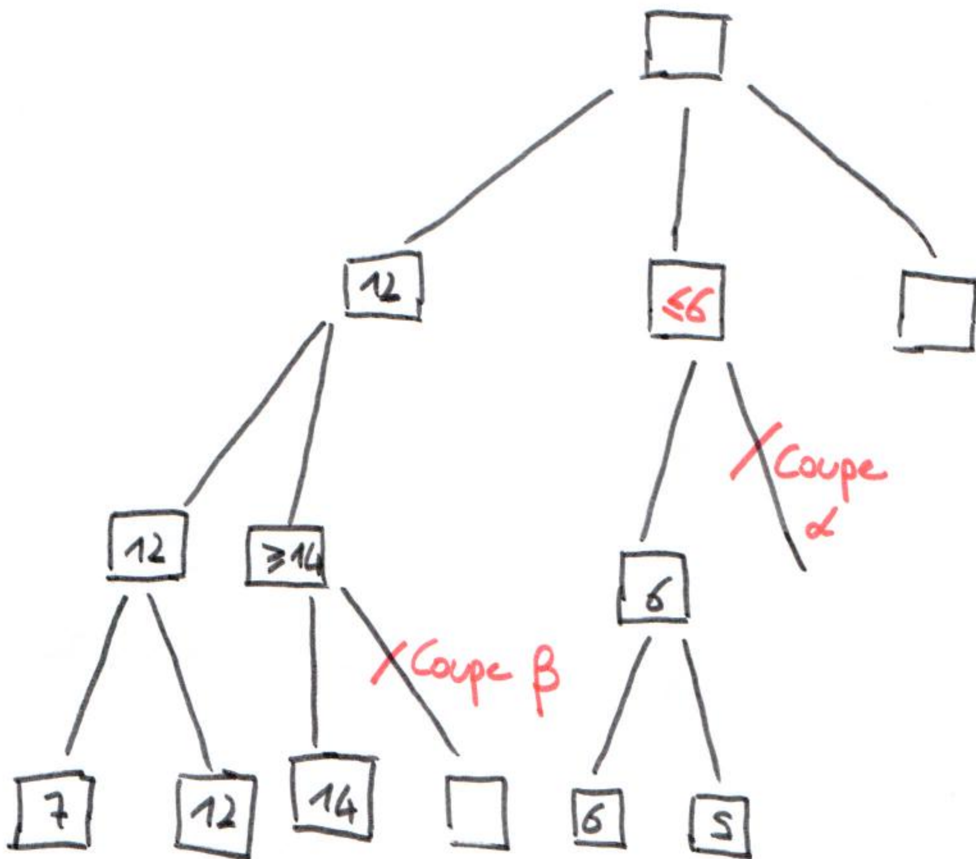
  MIN va donc ensuite choisir la valeur minimale entre 12 et ≥ 14 ⇝ ce sera 12 !

  ... peu importe ce que donne $h(p_1)$: on a donc pas besoin d'explorer la position p_1 et on peut couper la branche qui relie p_1 à son parent ⇝ c'est une coupe β

- Selon le même principe, une fois déterminées $h(p_2)$ et $h(p_3)$, MAX va choisir le coup qui mène à p_3 (donc 6), puis MIN choisira nécessairement une valeur de parent ≤ 6, cf (*).

Ainsi, MAX préfèrera le coup qui mène à 12 plutôt que à une position ≤ 6

↝ on a donc pas besoin d'explorer les autres coups possibles à partir de ≤ 6 : c'est une coupe alpha



Algorithme $\alpha-\beta$ On considère comme précédemment une fonction d'évaluation $h : \mathcal{P} \to \mathbb{R}$ et deux joueurs : MIN et MAX.

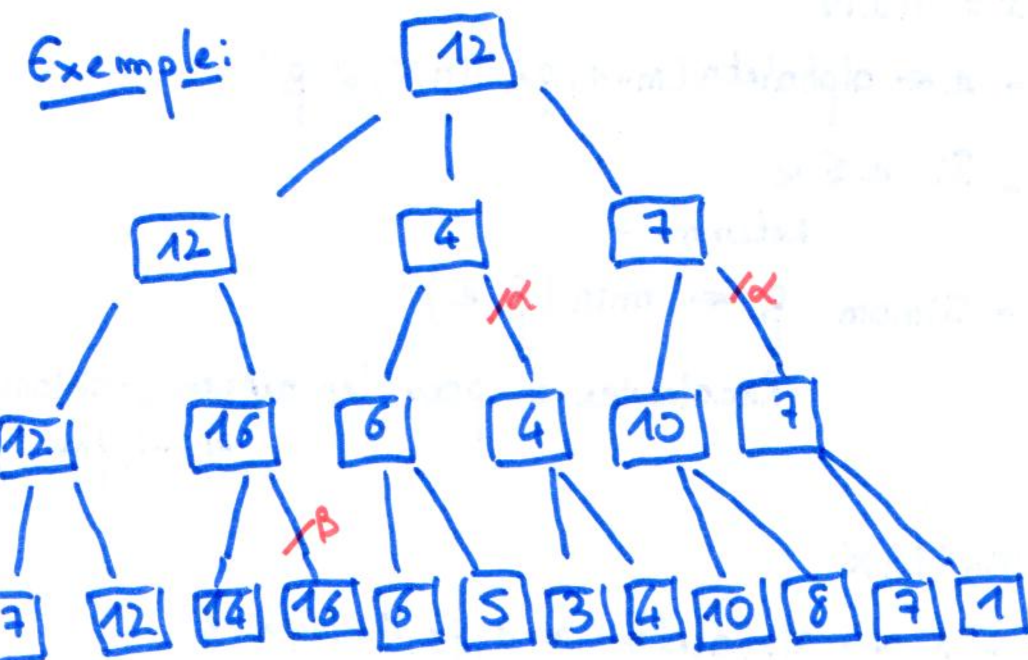
Pour évaluer p à une profondeur n, pour le joueur J, on conserve

- α = meilleure évaluation trouvée pour le joueur MAX dans les branches déjà vues ;
- β = meilleure évaluation trouvée pour le joueur MIN dans les branches déjà vues ;
- au départ, $\alpha = -\infty$ et $\beta = +\infty$.

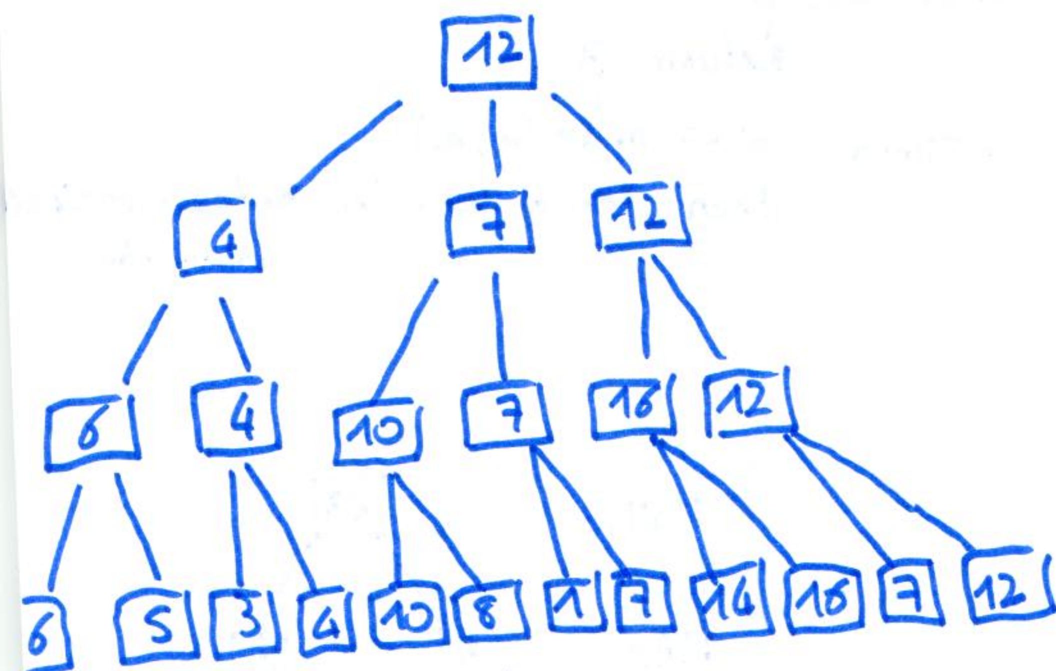
[Knuth '75] : lorsque l'algorithme minimax trouve un coup en parcourant n noeuds, alpha-beta trouve ce même coup en parcourant $\sim 1 + 2\sqrt{n}$ coups si les coups sont bien ordonnés

⇝ l'ordre des coups est important car en fonction de l'ordre, on ne réalise pas le même élagage.

Exemple :



Avec cet ordre, alpha-beta permet d'éviter de parcourir 7 noeuds (sur 22)



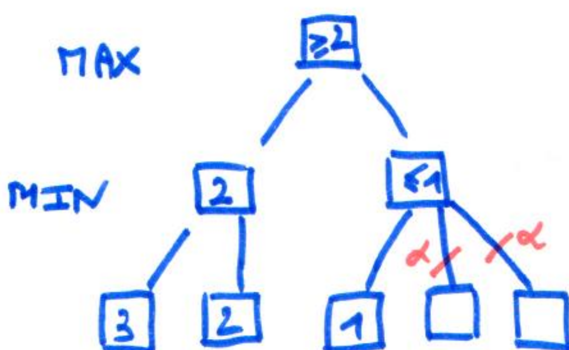
Avec cet ordre, on ne peut élaguer aucune branche.

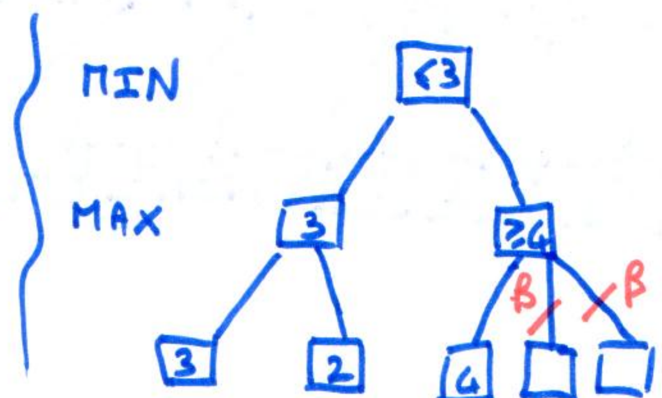
alphabeta (profondeur n, position p, joueur J, α, β)

alpha = -INF; beta = +INF;

- Si p est terminale,
  return $h^*(p)$
- Si $n = 0$
  return $h(p)$
- Sinon :
  - Soient $p_1, \dots, p_m$ les m positions accessibles depuis p
  - Si J = MIN
    - $e \leftarrow$ alphabeta($n-1$, p_1, MAX, α, β)
    - Si $e \leq \alpha$
      return α
    - Sinon $\beta \leftarrow \min(\beta, e)$
      Recalculer e pour les autres positions $p_2, \dots, p_m$
  - Si J = MAX
    - $e \leftarrow$ alphabeta($n-1$, p_1, MIN, α, β)
    - Si $e \geq \beta$
      Return β
    - Sinon $\alpha \leftarrow \max(\alpha, e)$
      Recalculer e pour les autres positions $p_2, \dots, p_m$

Schéma résumé :





Remarque: Il existe une variante de l'algorithme minimax, appelée negamax, plus utilisée actuellement.

Plutôt que de tester si on est à une profondeur paire ou impaire pour savoir si on cherche à maximiser ou minimiser, on peut inverser le signe des évaluations et toujours chercher à maximiser.

On peut toujours procéder à un élagage alpha-beta à partir d'un negamax.

Améliorations de l'Algorithme alpha-beta

1) Autoriser des coupures plus longues: jusqu'à présent, on a réalisé des coupures alpha ou beta en comparant la valeur d'un noeud avec les valeurs des frères de son parent. On peut aussi regarder les parents des parents.

Ex:

10

10 | 20

10 | 5 | 20 | 14

15 | 11 | 5 | 0 | 20 | 21 | 15 | 14

0 | 10 | 11 | 8 | -20 | 5 | -30 | 0 | 20 | 13 | 21 | 20 | 15 | 11 | 16 | 13

α β

Le 20 entouré en rouge ne peut a priori pas donner lieu à un élagage classique.

Mais on peut constater que le frère du parent de son parent a pour valeur 10, et que une valeur ≥ 20 ne sera jamais choisie :

- Soit le min sera obtenue par le frère du 20 à profondeur 3, auquel cas le 20 n'est pas sélectionné.
- Soit le 20 remonte jusqu'à profondeur 1, mais dans ce cas c'est le 10 qui sera choisi

⇝ le 20 ne sera donc jamais choisi, et donc son ~~voi~~ frère non plus (soit il est ≤ 20 et ne remonte pas, soit il est ≥ 20 et il sera encore moins choisi pour la même raison) : on peut donc réaliser une coupure "grand-beta / $\beta+$"

2) Alpha-beta itératif incrémental

L'idée est de commencer par réaliser une recherche à profondeur 1, puis recommencer avec une recherche complète à profondeur 2, puis profondeur 3, etc, jusqu'à ce qu'une solution soit trouvée

<u>Avantage</u>: On trouve toujours la solution la plus courte, puisqu'un noeud n'est jamais engendré tant que <u>tous</u> les noeuds à profondeur inférieure n'ont pas été explorés.

Solution trouvée rapidement

Solution trouvée à grande profondeur, sans profondeur incrémentale

Complexité : $O(d)$ si l'algo se termine à profondeur d.

A priori, on perd beaucoup de temps dans les itérations précédant la solution. Mais ce travail supplémentaire est en général beaucoup plus petit que la dernière itération

<u>Ex</u>: Si n coups en moyenne par position, à profondeur d on a n^d noeuds.

Le nombre de noeuds à profondeur $d-1$ est de n^{d-1} mais chaque noeud est engendré 2 fois (une fois par le lancement à prof $d-1$, une fois pour le lancement à profondeur d.)

Au final : nombre de noeuds engendrés :

$$n^d + 2n^{d-1} + 3n^{d-2} + \dots + dn = O(n^d)$$

Si n est grand, le premier terme est dominant, et c'est la dernière itération qui prend le plus de temps.

De plus : alpha-beta incrémental permet de

- développer des heuristiques de parcours, en reprenant le parcours de l'arbre à partir de la position qui était jugée la meilleure au lancement à profondeur précédente
- optimiser l'élagage, en implémentant un tri rapide pour placer les meilleurs coups dans l'ordre

3) Approfondissement sélectif, quiescence

- Pour pallier aux inconvénients des effets d'horizon, Deep Blue a utilisé une meta fonction d'évaluation qui n'évalue pas la position mais plutôt le type de la position

→ elle évalue si une position est stable (essentiellement, s'il reste des pièces en prise) qui donneront grosso modo la même évaluation à profondeur d ou d+1.

Une position stable a donc une évaluation en laquelle on peut avoir confiance alors qu'une instable non.

DeepBlue utilise un mécanisme de quiescence pour continuer de développer les positions lasses de l'arbre jusqu'à des positions stables.

Aux échecs, la quiescence est couramment utilisée pour des coups de capture ou des coups de promotion (sauf quand le roi est en échec)

Chaque étape de l'algorithme est constituée de 4 phases:

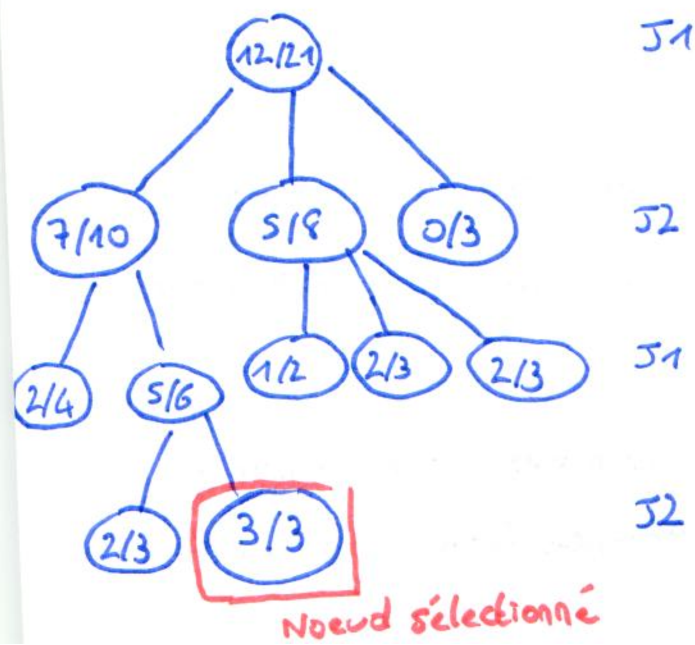
(1) Sélection: Depuis la racine de l'arbre, on sélectionne successivement des enfants jusqu'à atteindre une feuille.

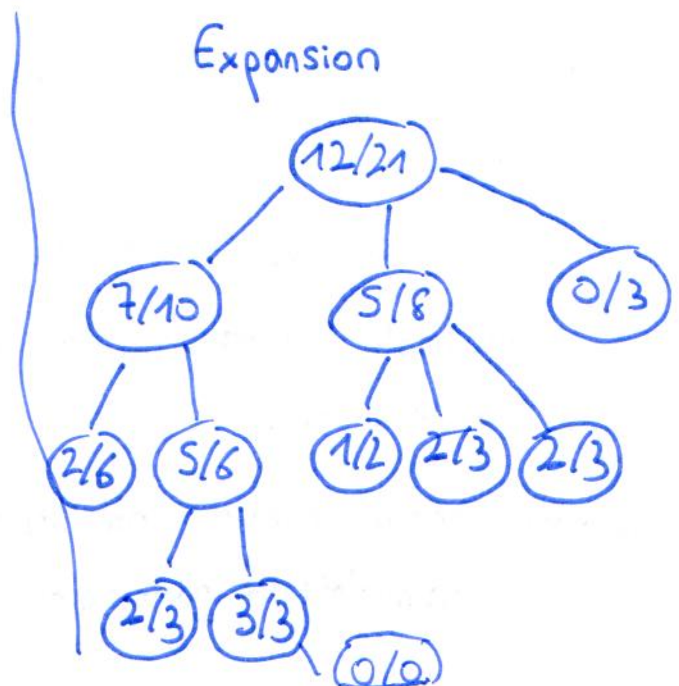
(2) Expansion: Si cette feuille n'est pas une position terminale, créer un enfant (ou plusieurs) en appliquant un mouvement suivant les règles du jeu, puis ajouté cet enfant marqué (0/0)

(3) Simulation: ("Playout" ou "Roll-out" en anglais)
Simuler une exécution d'une partie au hasard depuis cet enfant, jusqu'à atteindre une position de jeu terminale

(4) Back propagation: Utiliser le résultat de la partie au hasard et mettre à jour les informations sur la branche en partant de la feuille vers la racine

Exemple:





- Approfondissement sélectif:

Si un coup semble intéressant, on continue à chercher à une profondeur plus grande que la profondeur habituelle.

Si un coup semble mauvais, on arrête de chercher à une profondeur plus petite que la profondeur habituelle.

Ex: Chinook, IA pour les dames

- Si il analyse un coup qui perd 3 pions, plutôt que d'analyser à profondeur 10, il va réduire à seulement 5 coups à l'avance; en considérant que le coup a de bonnes chances d'être mauvais

- Si il analyse un coup qui paraît très bon, il augmentera la profondeur à 12 coups à l'avance.

c) Monte-Carlo Tree Search (MCTS)

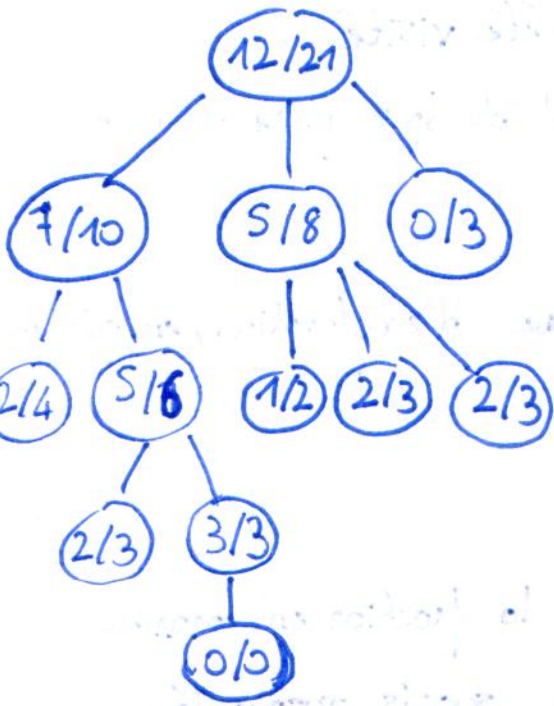
L'algorithme MCTS est un autre algorithme d'IA pour les jeux basé sur une recherche arborescente

MCTS conserve en mémoire un arbre qui correspond aux noeuds déjà explorés de l'arbre de jeu.

⤳ une feuille de cet arbre est soit une position terminale du jeu, soit un noeud dont aucun enfant n'a encore été exploré.

Dans chaque noeud, on stocke deux nombres : le nombre de simulations gagnantes, et le nombre total de simulations.

Simulation



Simulation d'une partie aléatoire

J1

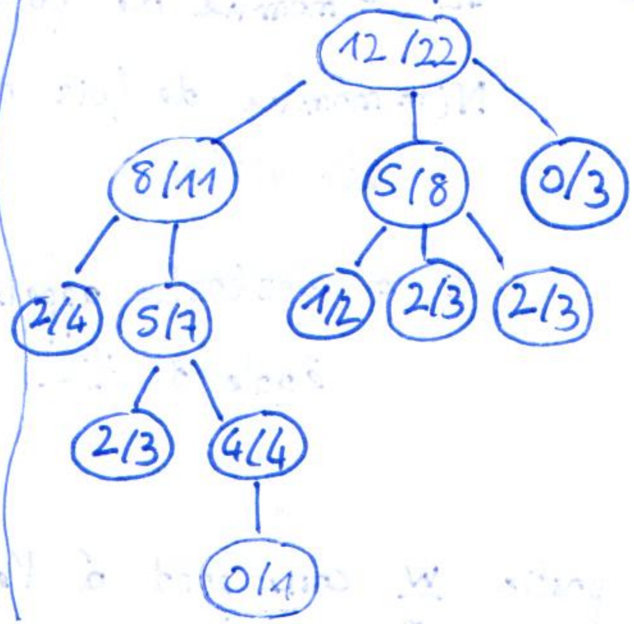
J2

J1

J2

J1

Backpropagation (partie perdante pour J1)



Si la partie est perdante pour J1 : on incrémente le nombre de simulation total uniquement pour les nœuds J1 et on incrémente à la fois le nombre de simulations totales et gagnantes pour J2 ... et inversement si la partie est gagnante pour J1.

- Pour la phase de sélection, il faut choisir un bon compromis entre l'exploitation des choix qui ont l'air prometteurs et l'exploration des nœuds à partir desquels peu de simulations ont été réalisées

↝ Formule UCT (Upper Confidence Bound) introduite par Kocsis et Szepesvari donne un bon compromis

On choisit en chaque nœud de l'arbre le successeur i qui maximise l'expression

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

où w_i = nb de parties gagnées marquées dans i

n_i = nombre de fois où i a été visité

N_i = nombre de fois où le noeud de base, père de i, a été visité

c = constante appelée paramètre d'exploration, en général égale à $\sqrt{2}$.

- La partie $\frac{w}{n}$ correspond à l'exploitation : la fraction est grande pour les successeurs qui ont eu beaucoup de succès jusque là.
- La partie $\sqrt{\frac{\ln N_i}{n_i}}$ correspond à l'exploration : elle est grande pour des successeurs impliqués dans peu de simulations.

Avantages : - pas besoin de fonction d'évaluation en cours de partie, uniquement de savoir jouer aléatoirement selon les règles du jeu et d'évaluer une partie terminée avec une fonction score

- Algorithme dit "anytime" : il est interruptible à tout instant en retournant son meilleur résultat courant. L'interruption peut se faire en nombre d'itérations, ou en temps, ou réalisée par la découverte d'une solution.

```
selection (pos p)
    Si p est terminale
        Return p
    N = nombre de visites du père de p      ← Fonction getN
    Si N == 0
        return p
    M ← {mouvements possibles depuis p}
    {max, best} ← {-1, ∅}
    pour chaque m ∈ M faire
    |   mi = nombre de fois où p a été visitée      ← Fonction getNi
    |   Si mi == 0
    |       Return p
    |   new-eval ← UCT(p, m)
    |   Si new-eval > max
    |       {max, best} ← {new-eval, m}
    p' ← apply-move(p, best)
    Return selection(p');
```

```
expansion (pos p)
    Si p est terminale
        Return p;
    N ← getN (p)
    Si N = = 0
        make_moves (s)
    M ← {mouvements possibles depuis p}
    pour chaque m ∈ M
        Ni ← getNi (p, m)
        Si Ni = = 0
            p' = apply-move (p, m)
            Return p';

playout (pos p)
    while not (p position terminale)
        | M ← next_moves (p)
        | m ← random (M)
        | p ← apply_move (p, m)
    Return score (p);

back propagate (pos p, score)
    if parent(p) = = ∅
        Return;
    add_score (p, score)
    return backpropagate (parent(p), score);
```

Fonction d'évaluation Puissance 4

① Grille de jeu pondérée

$$\begin{bmatrix} 3 & 4 & 5 & 7 & 5 & 4 & 3 \\ 4 & 6 & 8 & 10 & 8 & 6 & 4 \\ 5 & 8 & 11 & 13 & 11 & 8 & 5 \\ 5 & 8 & 11 & 13 & 11 & 8 & 5 \\ 4 & 6 & 8 & 10 & 8 & 6 & 4 \\ 3 & 4 & 5 & 7 & 5 & 4 & 3 \end{bmatrix}$$

score(J1) = somme des poids des cases sur lesquelles J1 a un jeton

② Compter le nombre d'alignements

Pour chaque joueur, on évalue son score de la façon suivante :

+1 par jeton

+5 par 2 jetons alignés

+50 par 3 jetons alignés

+1000 par 4 jetons alignés

$f(p) = \text{Score}(J1) - \text{Score}(J2)$

Inconvénient : cette évaluation tient compte des alignements "bloqués"

Ex:

		2	1	
		1	2	
2	1	1	1	2

donnera un score positif pour J1 avec 2 alignements de 3,

alors qu'un seul alignement de 4 n'est encore possible

<u>Mieux:</u> Ne compter dans ce calcul que les alignements encore envisageables