Jeux combinatoires

- Deux joueurs **J** et **O** qui alternent.
- Un ensemble $\mathcal{P}$ de positions du jeu.
- Deux fonctions de jeu $f_{\mathbf{J}}, f_{\mathbf{O}} : \mathcal{P} \to 2^{\mathcal{P}}$ qui spécifient les coups que les deux joueurs peuvent jouer à partir d'une position donnée pour faire évoluer le jeu (pour un jeu impartial $f_{\mathbf{J}} = f_{\mathbf{O}} = f$).
- $p \in \mathcal{P}$ est une position finale si $f(p) = \emptyset$. Quand le jeu atteint une position finale, il termine; le joueur qui a la main quand le jeu termine perd; l'autre joueur gagne.
- Le jeu termine au bout d'un nombre fini de coups.
- Les ensembles des positions gagnantes G et perdantes P sont définis de manière mutuellement récursive:
  1. Si p est finale, $p \in P$.
  2. S'il existe $p' \in f(p)$ tel que $p' \in P$, alors $p \in G$.
  3. Si pour tout $p' \in f(p)$ $p' \in G$, alors $p \in P$.
- On a: $G \cup P = \mathcal{P}$ et $G \cap P = \emptyset$ (c'est une partition de P).

Le jeu soustractif 1-2-3

- Une position du jeu est un nombre naturel: $\mathcal{P} = \mathbb{N}$.
- Chaque joueur, à son tour, choisit de soustraire 1, 2 ou 3 à la position courante (à condition que la position reste non-négative).
- Le joueur qui ne peut plus jouer (car la position est 0) a perdu.

Les positions Perdantes et Gagnantes:

$$
\begin{array}{ll}
P_0 = \{0\} & G_0 = \{1,2,3\} \\
P_1 = \{0,4\} & G_1 = \{1,2,3,5,6,7\} \\
P_2 = \{0,4,8\} & G_2 = \{1,2,3,5,6,7,9,10,11\} \\
\dots & \dots
\end{array}
$$

L'arbre de jeu à partir de la position $n_0 = 5$ (au tableau):

Implémentation du jeu soustractif

La règle du jeu:

```
move(X,Y):-(X>0, Y is X-1);
           (X>1, Y is X-2);
           (X>2, Y is X-3).
```

Une stratégie gagnante, si on dispose de `gagne/1` qui décide si un position est gagnante:

```
joue(X,Y) :- (move(X,Y),not(gagne(Y)));
              move(X,Y).
```

Le prédicat gagne/1

- Version "intelligente":

```
gagne(X) :- X mod 4 =\= 0.
```

- Version "force brute", sans mémoire:

```
gagne(X) :- move(X,Y), not(gagne(Y)).
```

- Version "force brute", avec mémoire:

```
:- dynamic gagne/1.

gagne(X) :- move(X,Y),
            not(gagne(Y)),
            asserta(gagne(X)).

/* asserta(p(...)) place le fait p(...)
au début de la base de connaissance.
Pour pouvoir utiliser asserta, p doit etre
declare dynamic */
```

Benchmarks pour les trois versions de gagne/1

- intelligente:
```
[eclipse 15]:  gagne1(1000001).
Yes (0.00s cpu)
```
- force brute:
```
[eclipse 9]:  gagne2(35).
Yes (4.43s cpu, solution 1, maybe more) ?  ;
```
- force brute avec mémoire:
```
[eclipse 13]:  gagne3(35).
Yes (0.00s cpu, solution 1, maybe more) ?
[eclipse 14]:  gagne3(1111).
Yes (0.02s cpu, solution 1, maybe more) ?
```

Benchmarks pour les trois versions de gagne/1

mais eclipse ne peut pas “apprendre” à l’infini...

```
[eclipse 17]:  gagne3(1000001).
*** Overflow of the local/control stack!
You can use the "-l kBytes" (LOCALSIZE) option to have a
larger stack.
Peak sizes were:  local stack 64044 kbytes, control stack
67028 kbytes
Abort
```

Le jeu de Nim

- Une position du jeu est une liste de nombres naturels.
- Chaque joueur, à son tour, choisit un élément strictement positif de la liste et le décrémente d'au moins d'une unité.
- Le joueur qui ne peut plus jouer (car tous les éléments de la liste sont 0) a perdu.

Le jeu de Nim

Existe-t-il une stratégie intelligente pour le jeu de Nim?

Définition: la somme Nim des entiers naturels $(n_1, \ldots, n_k)$ est le XOR chiffre par chiffre des représentations binaires de $(n_1, \ldots, n_k)$.

Par exemple, la somme Nim de $(1, 3, 5, 7)$ est

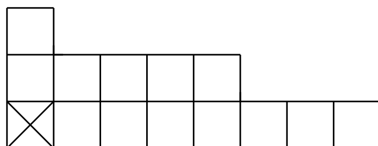
$$\begin{array}{ccc} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ \hline 0 & 0 & 0 \end{array}$$

Proposition: Une position $(n_1, \ldots, n_k)$ est gagnante pour le jeu de Nim si sa somme Nim est strictement positive; elle est perdante si sa somme Nim est nulle.

Il n'est donc pas nécessaire d'utiliser l'algorithme "force brute" pour le jeu de Nim, mais...

Chomp

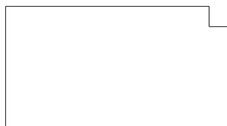
Position initiale: une matrice $n \times m$.
Règle du jeu: le joueur choisit une case $[i, j]$ de la position courante. La case $[i, j]$ et toutes les cases situées en haut et à droite de celle-ci disparaissent alors (elles sont "mangées", d'où le nom du jeu). Il s'agit des cases $[k, l]$, $k \leq i$, $l \geq j$.
P. e., commençant avec une matrice 3×8 et les deux premiers coups $[2, 6]$ (qui mange 6 cases) et $[1, 2]$ (qui mange 4 cases), on atteint la position:



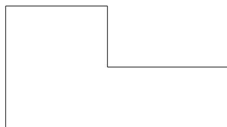
Le joueur qui a la main quand la matrice est réduite à la case $(n, 1)$ (le coin en bas à gauche) a perdu (variante du jeu normale, le dernier qui peut jouer perd).

Proposition: la position intiale du jeu *chomp* $n \times m$ est gagnante, pour tout $n, m \in \mathbb{N}$.

Preuve: supposons que la position initiale soit perdante. **J** joue $(1, m)$, qui mène à la position p_0 ci-dessous:



Vu que p_0 est gagnante par hypothèse, il existe à présent un coup jouable par **O** qui mène à une position perdante. Soit (i, j) ce coup et soit p_1 la position perdante atteinte:



Mais **J** aurait pu jouer (i, j) dès le début, et atteindre p_1, perdante. Donc la position initiale est gagnante. Absurde.

Aucune statégie “intelligente” (c.à.d. peu couteuse) n’est connue pour chomp. Il ne reste que la stratégie “force brute”:

```
/* une position est une liste triée d’entiers;
le prédicat move/2 est un peu plus compliqué
que celui du jeu de Nim */

:- dynamic gagne/1.

gagne(L) :- all_zero(L)./* la position vide est gagnante*/
gagne(L) :- move(L,G), not(gagne(G)), asserta(gagne(L)).

gagne(N,M) :- genere(N,M,L), gagne(L).

/* genere(3,4,L) produit L=[4,4,4] */
```

Benchmarks pour “Brute force Chomp”

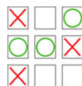
```
[eclipse 14]:  gagne(6,7).
Yes (8.41s cpu, solution 1, maybe more) ?
```

Comment gagner dans le cas 6×7

```
[eclipse 15]:  genere(6,7,L), move(L,L1), not(gagne(L1)).
L = [7, 7, 7, 7, 7, 7]
L1 = [4, 4, 7, 7, 7, 7]
Yes (0.01s cpu, solution 1, maybe more) ?
```

Evaluer une position

Beaucoup de jeu ne sont pas de jeu combinatoires, par exemple parce que certaines positions sont des “match nuls” :



ou parce que des cycles sont possibles:

Dans d’autres cas, il ne s’agit simplement pas d’un jeu à deux joueurs:

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

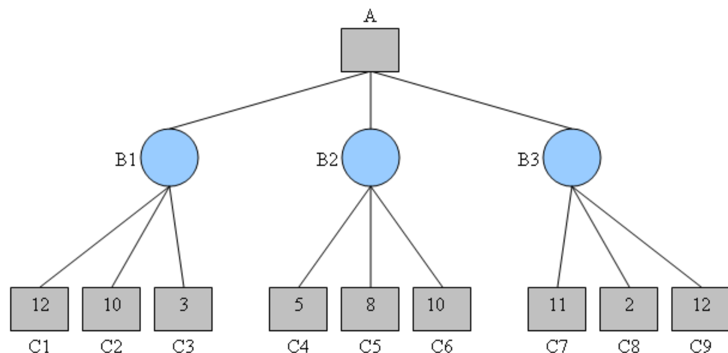
Fonctions heuristiques

Dans ces cas, plutôt que d'explorer l'arbre de jeu jusqu'aux feuilles (ce qui est en général irréaliste), on l'explore jusqu'à une profondeur donnée, et on évalue le positions à la frontière du sous-arbre ainsi engendré grâce à une fonction heuristique $h : \mathcal{P} \to \mathbb{Z}$.
La fonction h évalue une position de jeu, ou plus généralement un état du problème, et donne une approximation de la "distance" de cet état de la solution.
Exemple pour le jeu du taquin: la valeur d'une position est la somme de distances de Manhattan de chaque pièce par rapport à sa destination:
$h(\text{Start State}) = 18$
pour l'exemple du transparent précedent.
Une fois cette frontière évaluée, on fait rémonter les valeurs vers la racine de l'arbre de jeu en utilisant l'algorithme MinMax.

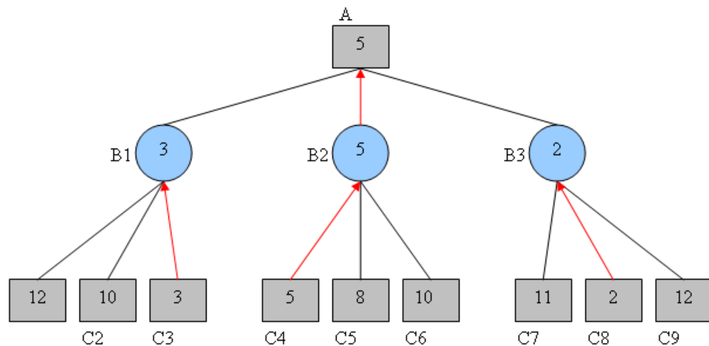
Le principe de l'algorithme MinMax

- **Le cadre:** Deux joueurs **J** (joueur) et **O** (opposant), jouant à tour de rôle, sur un ensemble de positions de jeu P.
- **Les données:**
  1. Une fonction heuristique $h : P \to \mathbb{Z}$, qui évalue les positions du jeu *du point de vue de* **J**: si $h(p_1) < h(p_2)$, alors la position p_2 est meilleur que la position p_1, pour **J**.
  2. La profondeur d'exploration de l'arbre de jeu: un entier a.
- **L'algorithme:**
  1. A partir d'une position de jeu donnée, on construit l'arbre de jeu jusqu'à la profondeur a.
  2. On évalue les positions aux feuilles de cet arbre en utilisant la fonction h.
  3. On fait remonter les valeurs aux noeuds internes: si **O** joue au noeud n, le minimum des valeurs des fils de n remonte vers n . Si c'est **J** qui joue au noeud n, le maximum des valeurs des fils de n remonte au noeud n.

Exemple de MinMax (1)



Exemple de MinMax (2)



Le morpion

$\mathcal{P}$ est l'ensemble des tableaux 3×3 partiellement remplis par les symboles $\circ$ et $\times$ (**J** joue les $\times$). Quelques exemples de positions:

p_0:

p_1:

		$\circ$
	$\times$	
$\times$	$\circ$	

p_2:

$\times$		$\circ$
$\circ$	$\times$	$\times$
$\times$	$\circ$	$\circ$

$$h(p) = \begin{cases} pg(\mathbf{J}) - pg(\mathbf{O}) & \text{si partie ouverte} \\ 0 & \text{si match nul} \\ 100 & \text{si } \mathbf{J} \text{ a gagné} \\ -100 & \text{si } \mathbf{J} \text{ a perdu} \end{cases}$$

$pg(\mathbf{J})$ désigne le nombre de "possibilité de gain" pour **J**.

$$h(p_0) = 0 \qquad h(p_1) = 1 \qquad h(p_2) = 0$$

(pour p_1: la première colonne, la deuxième lignes et la diagonale principale sont les 3 possibilité de gain pour **J**, la première ligne et la troisième colonne sont les 2 possibilité de gain pour **O**.)

Exemple d’application de MinMax à profondeur 2, à partir de la position

	○	

J a la main ...et joue sa survie (au tableau)

Implémentation de MinMax (1)

On se donne

- une représentations en Prolog des positions de jeu (typiquement par des listes, ou des listes de listes...).
- un prédicat `move/2` qui implémente la règle du jeu: si `p` représente une position du jeu, `move(p,X)` unifie `X` avec toutes les positions atteignables par un coup à partir de `p` (par des retours en arrière successifs: `move` ne génère pas une liste de positions).
- deux constantes `j` et `o` (les joueurs).
- un prédicat `finale/1`: `final(p)` réussit si `p` représente une position finale, échoue sinon.
- un prédicat `h/2` qui implémente la fonction héuristiqe: `h(p,X)` calcule la valeur de la position `p`, et affecte à `X` cette valeur.
- une profondeur d'exploration de l'arbre de jeu: un entier a.

Implémentation de MinMax (2)

`minmax/4`

- Premier argument $(+)$: le joueur (`j` ou `o`).
- Deuxième argument $(+)$: la profondeur (égale à `a` lors de l'appel initiale, decrémentée ensuite).
- Troisième argument $(+)$: la position de jeu courante.
- Quatrième argument $(-)$: la valeur MinMax de la position courante.

Deux cas de bases:

1. La position courante est finale.
2. La profondeur est 0.

Dans ces deux cas, la valeur MinMax de la position courante est donnée par `h`.

Implémentation de MinMax (3)

Dans le cas général, pour évaluer `minmax(J,A,P,V)`:

- On construit la liste `L` des positions atteignables à partir de `P`, par un appel de `setof(X,move(P,X),L)`.
- On calcule sa valeur MinMax de chaque élément de `L`, par des appels récursifs de `minmax`, en inversant le joueur et en decrémentant la profondeur.
- si le `J` est `j`, on affecte à `V` le maximum des valeurs calculées au point précedent.
- Sinon, on affecte à `V` le minimum de ces valeurs.

Implémentation de MinMax: le code

```
minmax(_,_,P,V) :- final(P),h(P,V). /*position finale*/
minmax(_,0,P,V) :- h(P,V).   /*profondeur maximale*/
minmax(J,A,P,V) :- setof(X,move(P,X),L),
                   A1 is A-1,
                   swap(J,J1), /*swap(o,j),swap(j,o)*/
                   mapminmax(J1,A1,L,LV),
                   (J==j -> maxliste(LV,V); minliste(LV,V)).


mapminmax(_,_,[],[]).
mapminmax(J,A,[P|L],[V|LV]) :- minmax(J,A,P,V),
                               mapminmax(J,A,L,LV).

maxliste([X],X) :- !.
maxliste([X|L],Y) :- maxliste(L,Z), max(X,Z,Y).
/* minliste pareil, avec min au lieu de max. */
```

Stratégie MinMax

On peut utiliser `minmax` pour choisir le coup à jouer, à partir d'une position `p`: On calcule la valeur MinMax de chacune des positions atteignables à partir de `p` et on choisit une position dont la valeur est maximale (si `j` joue) ou minimale (si `o` joue).
Autre possibilité : ajouter un argument `NextPos` au prédicat `minmax` (qui passe donc de 4 à 5 arguments). Quand on calcule le maximum (ou minimum) de la liste des positions atteignables on fait remonter à la fois la valeur et la position.
Voici une partie de Morpion dans laquelle les deux joueurs jouent la stratégie MinMax à profondeur 2:

```
-------   -------   -------   -------   -------   -------   -------   -------   -------
| | | |   | | | |   | | | |   | | |o|   | | |o|   | | |o|   | |x|o|   | |x|o|   |o|x|o|
-------   -------   -------   -------   -------   -------   -------   -------   -------
| |x| |   | |x| |   | |x| |   | |x| |   | |x|x|   |o|x|x|   |o|x|x|   |o|x|x|   |o|x|x|
-------   -------   -------   -------   -------   -------   -------   -------   -------
| | | |   | | |o|   |x| |o|   |x| |o|   |x| |o|   |x| |o|   |x| |o|   |x|o|o|   |x|o|o|
-------   -------   -------   -------   -------   -------   -------   -------   -------
```