PLAN

- Deux études de cas sur l'optimisation de programmes logiques:
  - les carrés latins.
  - le jeu des 16 allumettes.
- L'algorithme *Minimax*.
- Prédicats qui modifient dynamiquement un programme logique.

PLAN

- Deux études de cas sur l'optimisation:
  - les carrés latins.
  - le jeu des 16 allumettes.
- L'algorithme *Minimax*.

Les carrés latins

On veut définir le prédicat

```
/* carre_latin(+Taille,-Liste) */
```

pour trouver les carrés latins de taille donnée. Par exemple

```
carre_latin(3,X).
```

```
X= [8,3,4,1,5,9,6,7,2]
```

8	3	4
1	5	9
6	7	2

Première solution

On utilise un algorithme “générer et tester”, non optimisé:

```
carre_latin(N,L1):-
nombre_latin(N,M),      /* par ex pour N=3 on a  M=15          */
genere(N,L),            /* L contient les entiers de 1 a N*N */
perm(L,L1),             /*  L1 contient une permutation de L */
                        /*  explosion combinatoire:           */
                        /*  (N*N)! possibilites pour L1         */
test_lignes(L1,N,M),    /*  on verifie que L1 respecte        */
                        /*  la condition sur les lignes       */
test_colonnes(L1,N,M). /*  et sur les colonnes                */
```

Explosion combinatoire

Pour les carrés latins de taille 3, cet algorithme vérifie 9!= 362880 configurations.

Pour les carrés latins de taille 4, 16!, environ $2 * 10^{13}$, configurations.

L'énorme majorité de ces configurations ne sont pas des carrés latins, par exemple à cause du fait que la condition sur les lignes n'est pas réspecté:

par exemple, pour $N = 3$, les configurations de la forme:

`[9,8,....]`.

Optimisation: le principe

On peut éviter de générer de telles configurations: on génère des configurations qui, *par construction*, vérifient la condition sur les lignes:

```
ligne_possible(+Taille,        /*  3                      */
               +NombreLatin,   /*  15                     */
               +Configuration, /*  [1,2,3,4,5,6,7,8,9]    */
               -Ligne,         /*  [1,5,9]                */
               -Reste)         /*  [2,3,4,6,7,8]          */
```

Exemples d'utilisation de `ligne_possible`:

```
 ?- ligne_possible(3,15,[1,2,3,4,5,6,7,8,9],L,R).
L = [1,5,9],
R = [2,3,4,6,7,8] ? ;
L = [1,6,8],
R = [2,3,4,5,7,9];
.........

?- ligne_possible(3,15,[2,3,4,6,7,8],L,R).
L = [2,6,7],
R = [3,4,8] ? ;
L = [3,4,8],
R = [2,6,7] ? ;
no
```

Optimisation: l'implantation (1)

`ligne_possible` défini par récurrence sur `Taille`:

```
ligne_possible(1,M,L,[M],Q):-member(M,L),
                             enlever(M,L,Q).


ligne_possible(N,M,[X|L],[X|F],R):-N>1,
                                   X<M,
                                   P is M-X,
                                   Q is N-1,
                                   ligne_possible(Q,P,L,F,R).


ligne_possible(N,M,[X|L],F,[X|R]):-N>1,
                                   ligne_possible(N,M,L,F,R).
```

Optimisation: le principe (2)

A l'aide de `ligne_possible`, on définit le prédicat qui génére des configurations qui vérifient la condition sur les lignes:

```
choix_ligne(+Configuration,     /*  [1,2,3,4,5,6,7,8,9] */
            +Taille,            /*   3                  */
            -Config_resultat)   /*  [1,5,9,2,6,7,3,4,8] */
```

Exemple d'utilisation de `choix_lignes`:

```
 ?- choix_ligne([1,2,3,4,5,6,7,8,9],3,R).
R = [1,5,9,2,6,7,3,4,8] ? ;
R = [5,1,9,2,6,7,3,4,8] ? ;
R = [5,9,1,2,6,7,3,4,8] ?
.......

?- setof(U,choix_ligne([1,2,3,4,5,6,7,8,9],3,U),L),length(L,N).

......

N = 2592
```

2592, au lieu de 9!=362880 configurations.

Optimisation: l'implantation (2)

`choix_ligne` défini par récurrence sur la longueur de `L`:

```
choix_ligne(L,N,L1):-length(L,N),
                     perm(L,L1).


choix_ligne(L,N,L1):-nombre_latin(N,V),
                    ligne_possible(N,V,L,F,Q),
                     choix_ligne(Q,N,T,V),
                     perm(F,F1),
                     concat(F1,T,L1).
```

Optimisation: l'implantation (4)

Voici le prédicat optimisé

`carre_latin(+Taille,-CarreLatin)` :

```
carre_latin(N,R):-    nombre_latin(N,M),
                      genere(N,L),
                      choix_ligne(L,N,R), /* R verifie la */
                                          /* cond. sur les lignes*/
                      test_colonnes(R,N,M).
```

Tests

benchmarks *(sur nivose)*:

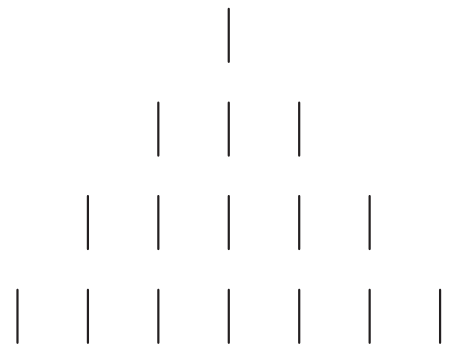
but	version non optimisé	version optimisé
`setof(X,carre_latin(3,X),L), length(L,N).`	15 sec. env.	8 sec.env
`carre_latin(4,X)`	?	5 sec env.

PLAN

- Deux études de cas sur l'optimisation:
  - les carrés latins.
  - le jeu des 16 allumettes.
- L'algorithme *Minimax*.

Le jeu des 16 allumettes

- Un jeu à deux joueurs, à information parfaite.
- Position initiale:



- Règle du jeu: à son tour, chaque joueur retire un certain nombre (positif) d'allumettes d'une ligne. Le joueur qui doit retirer la dernière allumette a perdu.

Le jeu des 16 allumettes: représentation

Une position du jeu est représentée par une liste de 4 entiers, la position initiale étant `[1,3,5,7]`.

Le prédicat `move(+Pos1,-Pos2)`, 1ère tentative:

```
move([X,Y,Z,T],[X,Y,Z,U]):-0=<U,U<T.

move([X,Y,Z,T],[X,Y,U,T]):-0=<U,U<Z.

move([X,Y,Z,T],[X,U,Z,T]):-0=<U,U<Y.

move([1,Y,Z,T],[0,Y,Z,T]).
```

Une requête:

```
move([0,0,1,1], L).

[ INSTANTIATION ERROR- =</2 at user:move/2 (clause 1):
expected bound value ]
```

Il faut que `move(P,Q)` *produise* `Q`.

Le prédicat `move/2`

```
between(X,Y,X):-X<Y.

between(X,Y,Z):-X<Y, T is X+1,between(T,Y,Z).


move([X,Y,Z,T],[X,Y,Z,U]):-between(0,T,U).

move([X,Y,Z,T],[X,Y,U,T]):-between(0,Z,U).

move([X,Y,Z,T],[X,U,Z,T]):-between(0,Y,U).

move([1,Y,Z,T],[0,Y,Z,T]).
```

Le prédicat `move/2`

```
   ?- move([0,0,2,2], L).
L = [0,0,2,0] ? ;
L = [0,0,2,1] ? ;
L = [0,0,0,2] ? ;
L = [0,0,1,2] ? ;
no
```

Le prédicat `gagne/1`, 1ère tentative:

Une position est gagnante si le joueur a une stratégie gagnante à partir de cette position.

```
gagne([0,0,0,0]).
```

```
gagne(Pos):-move(Pos,Pos1),move(Pos1,Pos2),gagne(Pos2).
```

la position `[0,0,2,2]` est perdante,pourtant...

```
?-  gagne([0,0,2,2]).
  (1) call:gagne([0,0,2,2]) ?
  (2) call:move([0,0,2,2],_252) ?
  (3) call:between(0,2,_317) ?
  (3) exit:between(0,2,0) ?
  (2) exit:move([0,0,2,2],[0,0,2,0]) ?
  (4) call:move([0,0,2,0],_461) ?
  (5) call:between(0,0,_526) ?
  (5) fail:between(0,0,_526) ?
  (4) redo:move([0,0,2,0],_461) ?
  (6) call:between(0,2,_524) ?
  (6) exit:between(0,2,0) ?
```

```
   (4) exit:move([0,0,2,0],[0,0,0,0]) ?

   (7) call:gagne([0,0,0,0]) ?

   (7) exit:gagne([0,0,0,0]) ?

   (1) exit:gagne([0,0,2,2]) ?

yes
```

La clause

```
gagne(Pos):-move(Pos,Pos1),move(Pos1,Pos2),gagne(Pos2).
```

contient une erreur logique:

Une position est gagnante s'il existe un coup à jouer tel que *pour tout* coup de l'adversaire la position reste gagnante.

Dans la clause ci-dessus un position est gagnante s'il existe un coup à jouer tel que *il existe* un coup de l'adversaire qui mène à une position gagnante.

Première solution (1)

Terminologie: une position de jeu `Q` est un successeur de la position `P` si `move(P,Q)` réussit.

Le principe: on définit par récursion mutuelle les prédicats `gagne/1` et `perd/1`

- la position `[0,0,0,0]` gagne.
- une position gagne si elle a un successeur qui perd.
- une position perd si tous ses successeurs gagnent.

(confronter, par exemple, avec les problèmes d'echecs du type "blanc joue et gagne en n coups")

Première solution (2)

Le prédicat `gagne(Pos)`, version 1:

```
gagne([0,0,0,0]).

gagne(Pos):-move(Pos,Pos1),perd(Pos1).

perd(Pos):-setof(X,move(Pos,X),R),gagnent(R).

gagnent([]).

gagnent([X|L]):-gagne(X),gagnent(L).
```

Première solution (3)

Ce programme est très inefficace. En particulier, si

- `perd(P)` est lancé.
- la liste des successeurs de `P` est `[P_1,...,P_n]`.
- le but `gagne(P_i)` réussit pour $i < j$.
- le but `gagne(P_j)` echoue.

on peut conclure que `perd(P)` echoue.

Mais cette version du programme, avant de déclarer que `perd(P)` echoue, cherche inutilement toutes les solutions alternative de `gagne(P_1)` ,...,`gagne(P_(j-1))` (conséquence du backtracking).

Utilisation de la coupure (1)

Cette source d'inefficacité peut être éliminée avec `!`: il suffit de modifier dans le programme précedent la clause

```
gagne(Pos):-move(Pos,Pos1),perd(Pos1).
```

par

```
gagne(Pos):-move(Pos,Pos1),perd(Pos1),!.
```

cette modification peut se lire: une fois qu'on a trouvé un successeur de `Pos` qui perd, il est inutile d'en chercher d'autres

(comme aux echecs: si on trouve un coup qui mène à echec et mat, on en cherche pas d'autres...)

Utilisation de la coupure (2)

Le prédicat `gagne(Pos)`, version 2:

```
gagne([0,0,0,0]).
gagne(Pos):-move(Pos,Pos1),perd(Pos1),!.
perd(Pos):-setof(X,move(Pos,X),R),gagnent(R).
gagnent([]).
gagnent([X|L]):-gagne(X),gagnent(L).
```

cette version reste relativement inefficace: supposons que

- `perd(P)` est lancé.
- la liste des successeurs de `P` est `[P_1,...,P_n]`.
- le but `gagne(P_1)` echoue

globalement, `perd(P)` echoue, et on a construit inutilement les successueurs `[P_2,...,P_n]` de `P`.

En fait, `perd(P)` peut être défini comme la négation de `gagne(P)`.

Utilisation de la négation

Le prédicat `gagne(Pos)`, version 3:

```
gagne([0,0,0,0]).
gagne(Pos):-move(Pos,Pos1),perd(Pos1).
perd(Pos):-not(gagne(Pos)).
```

L'utilisation explicite de la coupure dans la clause de `gagne` devient inutile, à cause de la coupure (implicite) de `not(gagne(Pos))`.

En réalité, une fois le programme mis sous cette forme, on peut éliminer le prédicat `perd`:

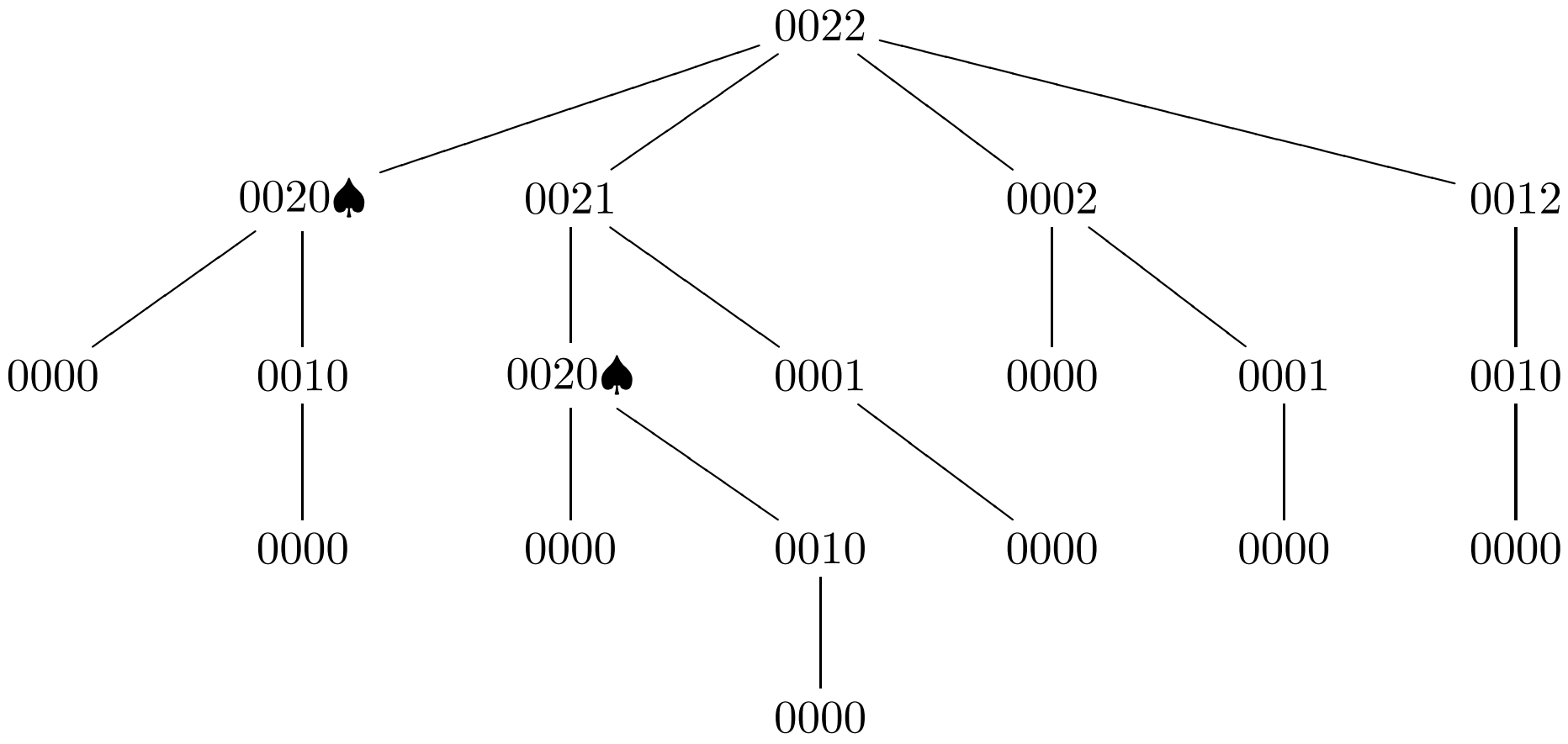
Le prédicat `gagne(Pos)`, version 3bis:

```
gagne([0,0,0,0]).
gagne(Pos):-move(Pos,Pos1),not(gagne(Pos1)).
```

On peut maintenant s'attaquer à une (dernière) source d'inefficacité de la version 3bis: les buts repliqués.

Buts repliqués

L'arbre de recherche (simplifié) de `gagne([0,0,2,2]).`



L' "apprentissage"

On peut éviter de recalculer plusiers fois le même but de la forme `gagne(Pos).` (comme `gagne([0,0,2,0])` dans l'exemple) si, à chaque fois qu'un but `gagne(Pos)` réussit on rajoute l'assertion `gagne(Pos)` à la base de donnée, avec `asserta`.

Le prédicat `gagne(Pos)`, version 4:

```
gagne([0,0,0,0]).
gagne(Pos):-move(Pos,Pos1),not(gagne(Pos1)),asserta(gagne(Pos)).
```

Tests

Les 4 versions du programme

1. `perd` avec `setof`, sans `!`.
2. `perd` avec `setof`, avec `!`.
3. négation, sans apprentissage.
4. négation, avec apprentissage.

benchmarks *(sur nivose)*:

but	1	2	3	4
`gagne[(1,3,5,7)]`	?	6 sec. env.	3 sec. env.	immédiat
`gagne[(1,0,5,7)]`	17 sec env.	immédiat	immédiat	immédiat

L'importance du prédicat `move`

Si on défini `move` de telle façon qu'on essaye d'abord les "petits" changements de positions (par exemple, on commence par enlever une allumette sur la dernière rangée disponible, puis deux...) les performances se dégradent:

but	1	2	3	4
`gagne[(1,3,5,7)]`	?	30 sec. env.	12 sec. env.	immédiat
`gagne[(1,0,5,7)]`	?	ϵ	immédiat	immédiat

implantation du jeu

Le prédicat `gagne` peut être utilisé pour choisir le prochain coup, à une position donnée:

```
next(Pos,Next):-move(Pos,Next),not(gagne(Next)),!.
```

et, si `Pos` perd, 2 possibilités:

pessimiste (on suppose que l'adversaire joue la stratégie optimale):

```
next(Pos,_):-write('je me rends').
```

optimiste, (on suppose que l'adversaire peut se tromper):

```
next(Pos,Next):-move(Pos,Next),!.
```

Ceci donne le noyau du programme qui permet de jouer contre la machine. Ensuite, il faut implementer l'interface du jeu.

PLAN

- Deux études de cas sur l'optimisation:
  - les carrés latins.
  - le jeu des 16 allumettes.
- L'algorithme *Minimax*.

L'algorithme Minimax (1)

On revient sur le principe de la première solution du jeu des allumettes:

- la position `[0,0,0,0]` gagne.
- une position gagne si elle a un successeur qui perd.
- une position perd si tous ses successeurs gagnent.

Ici, le fait que la position qui est racine d'un arbre de jeu gagne, signifie que **le joueur qui a la main à cette position possède une strategie gagnante**.

On peut internaliser dans l'arbre la dualité joueur/opposant, et dir qu'il existe deux types de noeuds (qui s'alternent strictement sur chaque branche): les noeuds où joueur a la main (traditionellement: noeuds MAX) e les noeuds où l'opposant a la main (traditionellement: noeuds MIN).

L'algorithme Minimax (2)

Dans cette optique, il y a deux type de cas de base:

- 0 allumette, noeud MAX (dont on peut dir que la valeur est 1).
- 0 allumette, noeud MIN (dont on peut dir que la valeur est 0).

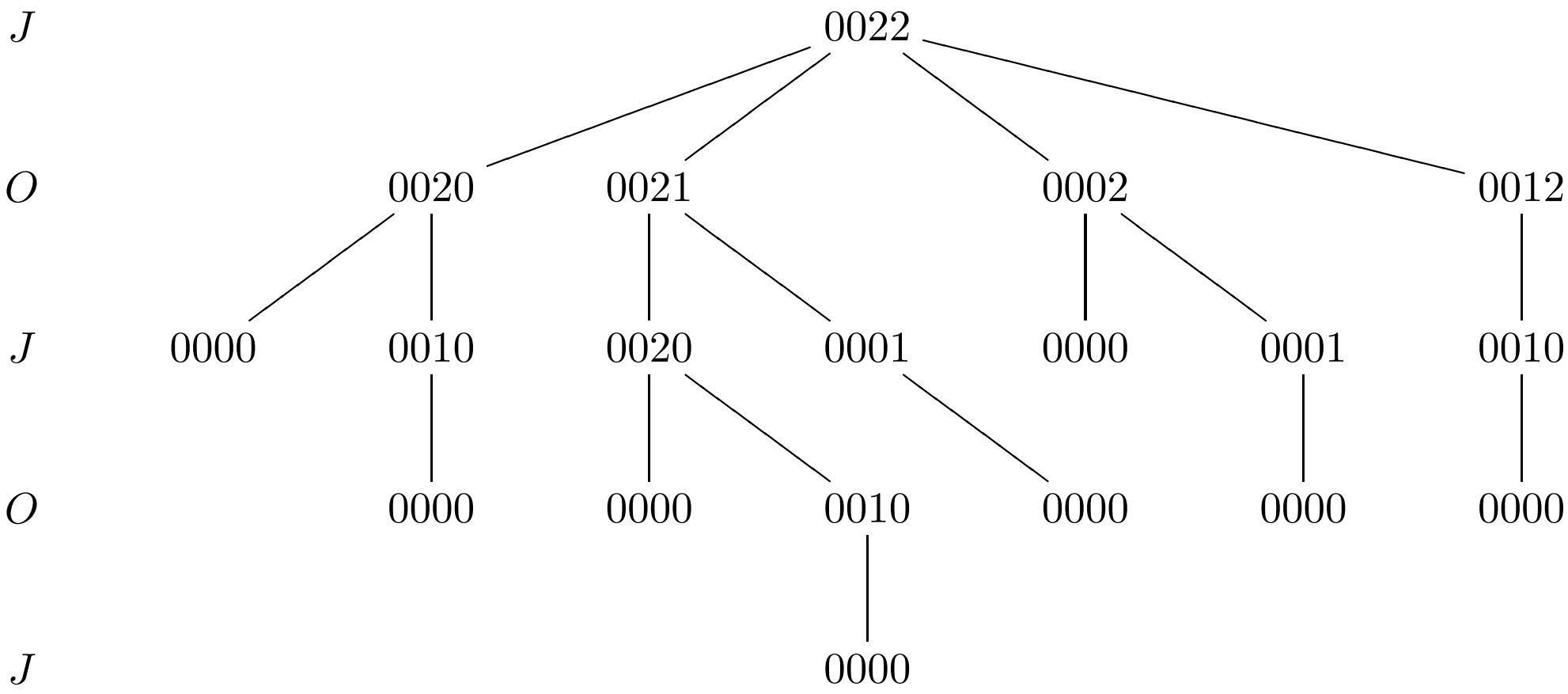
En suite, la valeur d'un noeud qui n'est pas une feuille se calcule:

- La valeur d'un noeud MAX est le max des valeurs de ses fils (le joueur MAX veut gagner, c.à.d. qu'il veut aboutir à une feuille dont la valeur est 1).
- La valeur d'un noeud MIN est le min des valeurs de ses fils (le joueur MIN veut gagner, aussi , c.à.d. qu'il veut aboutir à une feuille dont la valeur est 0).

Autrement dit, la valeur d'un noeud MAX est 1 si et seulement si le noeud en question a *au moins un fils* dont la valeur est 1, et la valeur d'un noeud MIN est 1 si *tous ses fils* ont valeur 1.

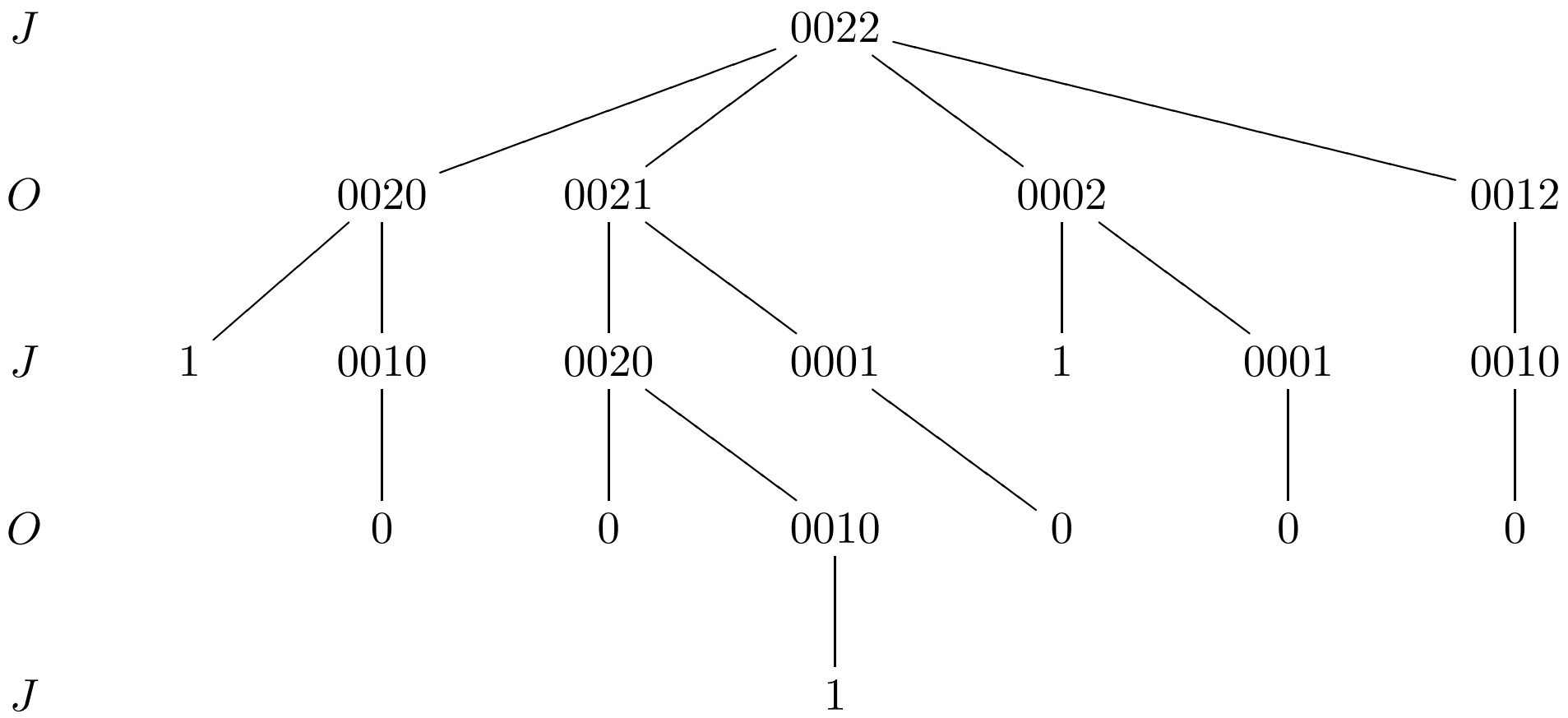
Minimax: un exemple (J=MAX, O=MIN)

L'arbre du jeux des 16 allumettes à partir de `[0,0,2,2]`



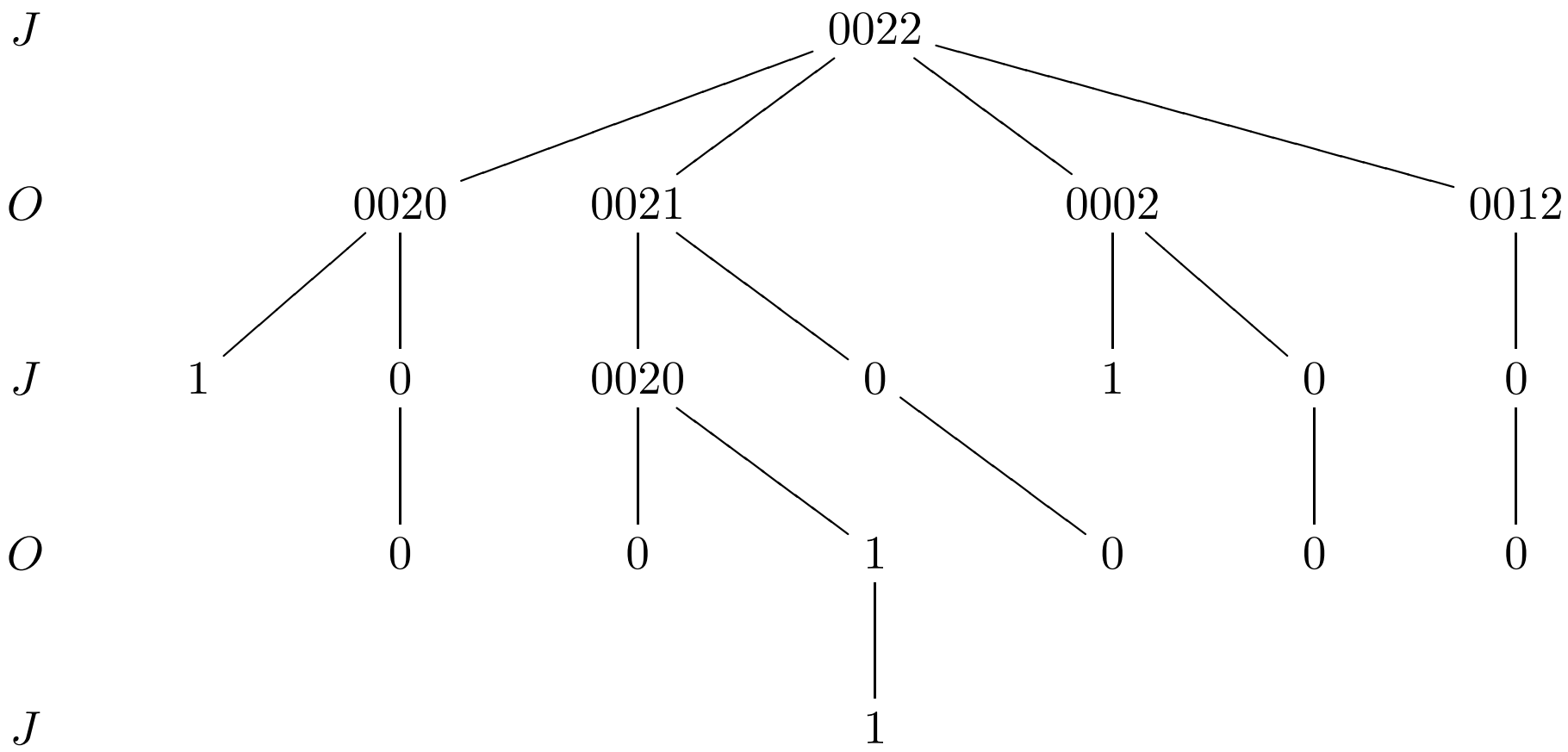
Minimax: un exemple (2)

On évalue les feuilles: 1 pour Joueur , 0 pour Opposant:



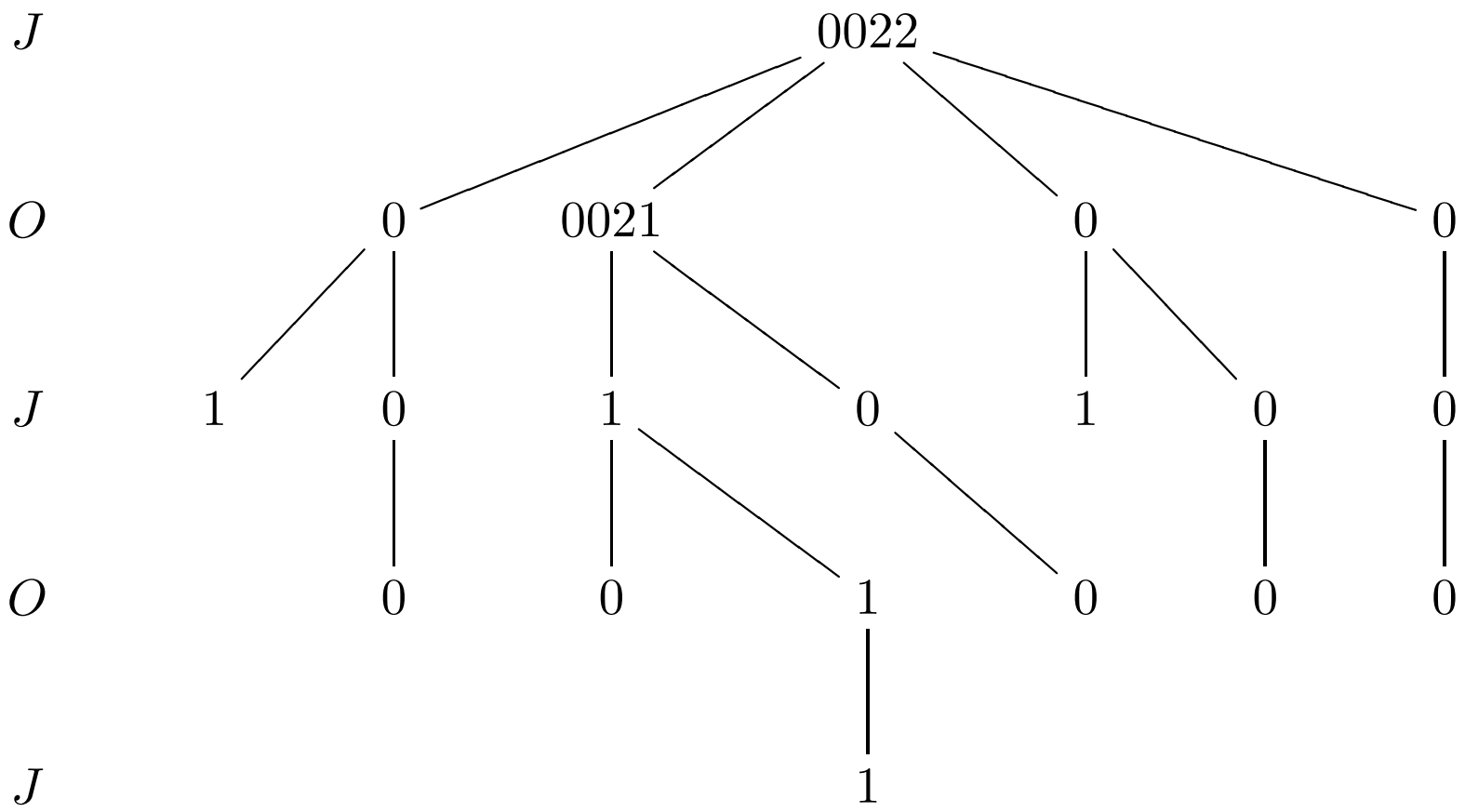
Minimax: un exemple (3)

Propagation des valeurs: max des fils pour J, min pour O:



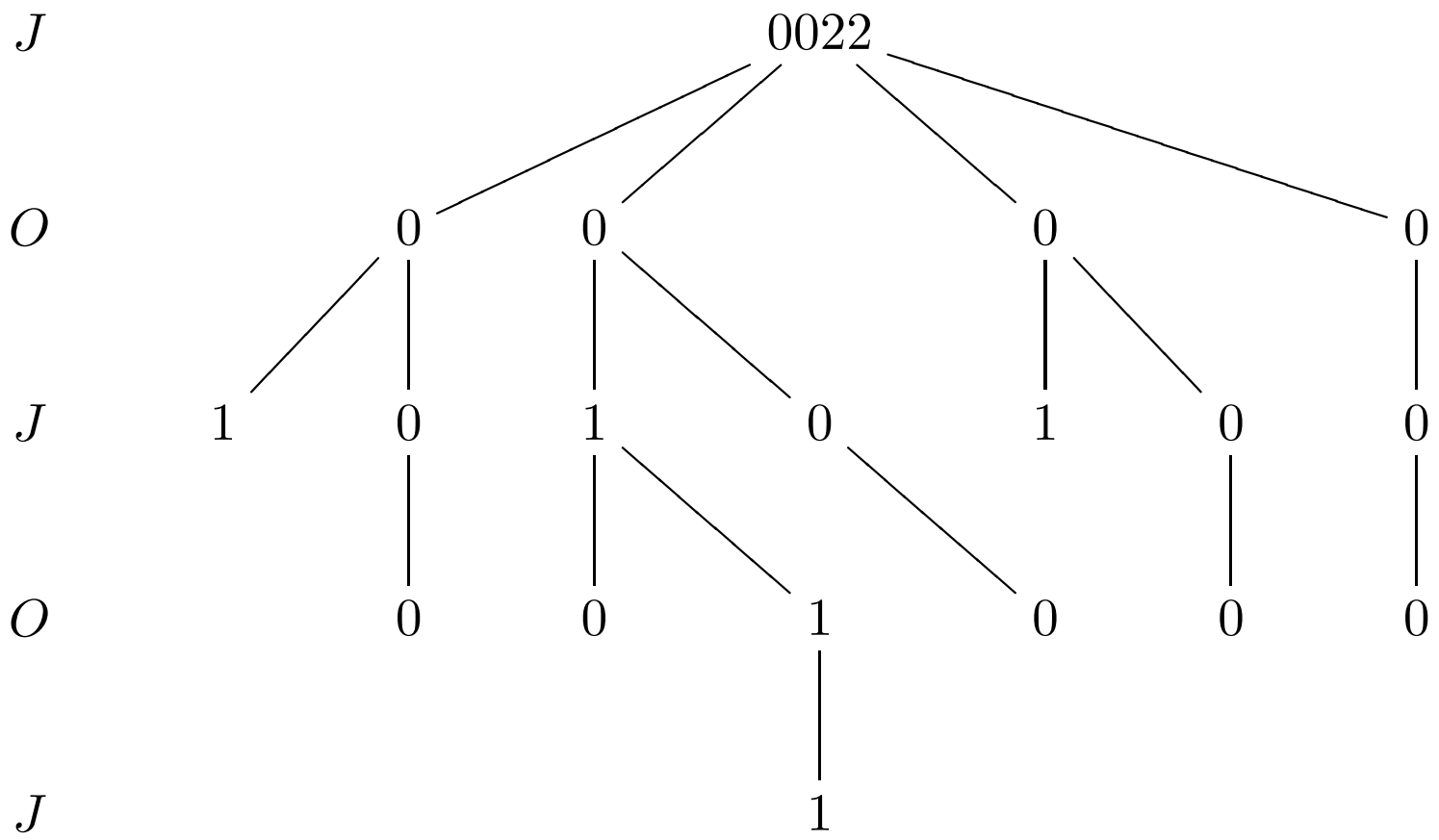
Minimax: un exemple(4)

Propagation des valeurs: max des fils pour J, min pour O:



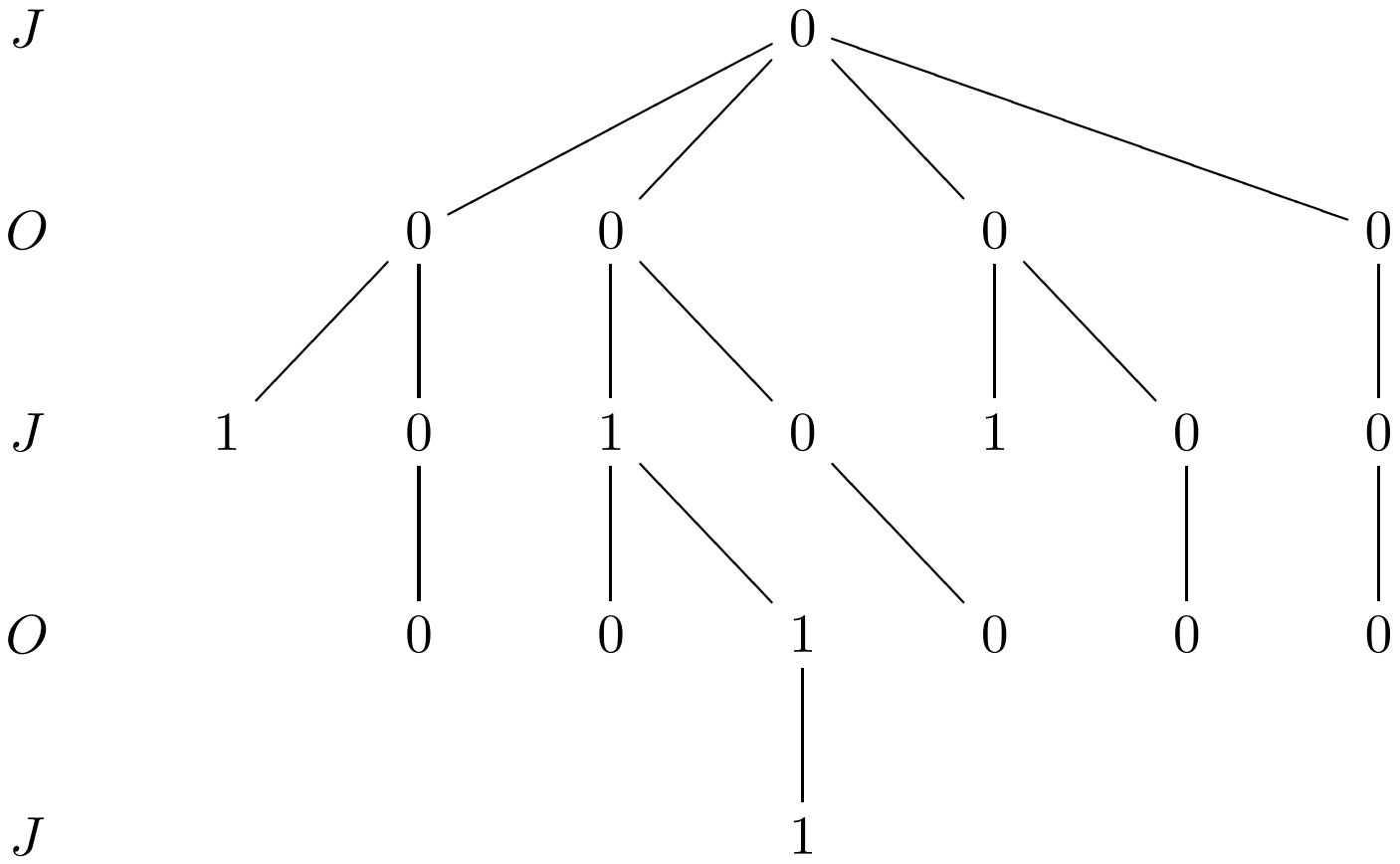
Minimax: un exemple(5)

Propagation des valeurs: max des fils pour J, min pour O:



Minimax: un exemple(6)

Propagation des valeurs: max des fils pour J, min pour O:



L'algorithme Minimax (3)

Il existe deux familles d'algorithmes minimax:

- à *arbre de jeux*.
- à *arbre de recherche*.

Dans le premier cas la profondeur des arbres explorés n'est pas borné: les feuilles sont des configurations finales, et la fonctions d'évaluation est dans la pluparts des cas triviale (typiquement: 1 pour "MAX gagne", 0 pour "match nul", -1 pour "MIN gagne").

Dans le deuxième cas, la hauteur de l'arbre est bornée, et la fonction d'évaluation peut être compliquée (comment évaluer, par exemple, une position aux echecs?).

Dans les deux cas, la propagation des valeurs vers la racine suit le même principe: on propage le minimum des valeurs des fils sur les noeuds MIN et le maximum sur les noeuds MAX.

Implantation de minimax *à arbre de jeu*

```
/* minimax(Joueur,Position,Valeur) */
/* on suppose finale/1 heurist/2 move/2 */

minimax(J,P,V):-finale(P),heurist(P,V),!.

minimax(max,P,V):- setof(Next,move(P,Next),Fils),
mapminimax(min,Fils,Valeurs), maximum(Valeurs,V).

minimax(min,P,V):- setof(Next,move(P,Next),Fils),
mapminimax(max,Fils,Valeurs), minimum(Valeurs,V).

mapminimax(_,[],[]).
mapminimax(J,[P|L],[V|G]):- minimax(J,P,V), mapminimax(J,L,G).

maximum([X],X).
maximum([X|L],M):-maximum(L,N), max(X,N,M).

minimum([X],X).
minimum([X|L],M):-minimum(L,N), min(X,N,M).
```

```
max(N,M,N):- N>=M,!.
max(N,M,M):- N<M.

min(N,M,N):- N=<M,!.
max(N,M,M):- N>M.
```

Implantation de minimax *à arbre de recherche*

```
/* h(Pos,Val) heuristique */
/* move(Joueur,Pos,NewPos) regle du jeu */
/* pinfinie,minfinie/1 tres grand et tres petit
entiers*/
/* minimax(Joueur,Pos,Valeur,Pos_choisie,Pr) */


minimax(_,Config,Valeur,Config,0) :-
 h(Config,Valeur),!.

minimax(max,Config,Valeur,Decision,H) :-
 H > 0,
setof(X,move(max,Config,X),Positions),
 H2 is H-1,
 minfinie(Z),
 selection_max(Positions,nil,Z,Decision,Valeur,H2).
```

```
minimax(min,Config,Valeur,Decision,H) :-
H > 0,
setof(X,move(min,Config,X),Positions),
H2 is H-1,
pinfinie(Z),
selection_min(Positions,nil,Z,Decision,Valeur,H2).

selection_max([],Pos,Valeur,Pos,Valeur,_).
selection_max([Position|Suivant],Pos,Valeur,MPos,MVal,H) :-
minimax(min,Position,Val,Decision,H),
max(Valeur,Pos,Val,Position,MVal2,MPos2),
selection_max(Suivant,MPos2,MVal2,MPos,MVal,H).

selection_min([],Pos,Valeur,Pos,Valeur,_).
selection_min([Position|Suivant],Pos,Valeur,MPos,MVal,H) :-
minimax(max,Position,Val,Decision,H),
in(Valeur,Pos,Val,Position,MVal2,MPos2),
selection_min(Suivant,MPos2,MVal2,MPos,MVal,H).
```

```
max(V1,P1,V2,P2,V1,P1) :-
 V1 > V2,!.
max(V1,P1,V2,P2,V2,P2).

min(V1,P1,V2,P2,V1,P1) :-
 V1 < V2,!.
min(V1,P1,V2,P2,V2,P2).
```

Quelques prédicats pour modifier dynamiquement un programme logique:

`asserta(+C)`

Ajoute le prédicat (atomique) `C` au debut du programme.

`assertz(+C)`

Ajoute le prédicat (atomique) `C` à la fin du programme.

`abolish(+P,+N)`

Elimine les régles du prédicat `P` d'arité `N`.