Raisonnement par Contraintes

Sylvain Piechowiak

UMR CNRS 8530

UNIVERSITÉ DE VALENCIENNES ET DU HAINAUT CAMBRÉSIS
Le Mont Houy 59313 VALENCIENNES CÉDEX 9 Tél 33 (0)3 27 51 14 38 fax 33 (0)3 27 51 13 16

Objet de cette présentation

- **P**roblèmes de **S**atisfaction de **C**ontraintes

CSP

- Langages de **P**rogrammation (*souvent **L**ogique*) **p**ar **C**ontraintes

PLPC

« Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it. »

Eugene C. Freuder, CONSTRAINTS, Avril 1997

Idées générales

«Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.»

Eugene C. Freuder, CONSTRAINTS, Avril 1997

Se donner les moyens de privilégier la modélisation d'un problème (***quoi***) plutôt que la résolution du problème (***comment***).

Séparer clairement le modèle de la méthode de résolution

(même s'il est important de connaître les méthodes pour bien modéliser)

Décrire simplement les problèmes : utilisation de variables et de relations entre ces variables (***contraintes***). La description du problème ne préjuge en rien de la méthode de résolution.

On se trouve à la frontière de l'IA (Intelligence Artificielle) et de la RO (Recherche Opérationnelle)

Lectures conseillées

Peu de livres sur ce sujet mais des publications...

- F. Fages — *Programmation Logique par Contraintes,* Ellipses, 1996
- E. Tsang — *Foundations of Constraint Satisfaction*, Academic Press, University of Essex, 1995
- K. Marriott, P. J. Stuckey — *Programming with constraints – An introduction*, MIT Press, 1998
- K. Apt, M.G. Wallace — *Constraint Logic Programming using Eclipse*, Cambridge University Press, 2007
- T. Frühwirth, S. Abdennadher — *Essential of constraint programming*, Springer-Verlag, 2003
- Les revues — *Constraints* (Kluwer Academic Publishers)
  Artificial Intelligence (Elsevier)
  Revue d'Intelligence Artificielle (Hermes)
- Les sites internet consacrés aux contraintes
  http://hp1.essex.ac.uk/CSP/
  http://www.cs.unh.edu/ccc/archive

Lectures conseillées

Congrès/Associations

- CP — Constraint Programming
- IJCAI — International Joint Conference on Artificial Intelligence
- AAAI — American Association on Artificial Intelligence
- RFIA — Reconnaissance des formes et IA
- AFPLC — Association Française de la programmation logique par contraintes (**http://www.afplc.org/)**
- ECAI — European Conference on Artificial Intelligence
- PACLP — The Practical Application of Constraint Technologies and Logic Programming

Exemples d'applications

- Génération d'emplois du temps (Timetabling)
- Planification
- Diagnostic (rechercher les causes d'une panne)
- Debugging de programmes (recherche des erreurs dans un programme)
- Conception d'interfaces (création d'interfaces animées), vision, …
- Conception de systèmes distribués / systèmes « multi agents »
- Séquençage de l'AND
- Design de circuits électroniques

Exemples de systèmes/langages

- SKETCHPAD, Sutherland, 1963
- ALICE, J.L. Laurière, 1976
- CONSTRAINTS, G. Sussman, 1977
- THINGLAB, A. Borning
- **GnuPROLOG** (D. Diaz), **ECL^{i}PS^{e}** (projet européen), **BPROLOG** (Zhou), **PROLOG-III & IV** (Colmeraeur)
- ILOG-SOLVER
- CHARME (BULL)
- CLAIRE
- etc.

En général, les langages ou systèmes ne traitent qu'une classe de contraintes souvent liées au type des valeurs manipulées (booléennes, entières, réelles ou intervalles, domaines finis ou infinis, discrets ou continus)

Les CSPs

Un CSP est défini par un triplet (V,D,C) :

- **un ensemble (fini) de variables**

$$V = \{V_1, V_2, \ldots, V_n\}$$

- **un ensemble de domaines (un domaine par variable)**

$$D = \{D_1, D_2, \ldots, D_n\}$$

- **un ensemble fini de contraintes**

$$C = \{C_1, C_2, \ldots, C_m\}$$

Les CSPs

- les variables ne peuvent prendre que les valeurs de leurs domaines

- on dit qu'une variable est **assignée** d'une valeur lorsqu'on lui attribue l'une des valeurs de son domaine

$$V \leftarrow v$$

- on appelle **assignation partielle** toute assignation d'une ou plusieurs variables du CSP

$$V_{i_1} \leftarrow v_{i_1}, V_{i_2} \leftarrow v_{i_2}, \ldots V_{i_n} \leftarrow v_{i_n}$$

- on appelle **assignation totale** toute assignation de toutes les variables du CSP.

$$\left(V_{i_1}, V_{i_2}, \ldots, V_{i_n}\right) \leftarrow \left(v_{i_1}, v_{i_2}, \ldots, v_{i_n}\right)$$

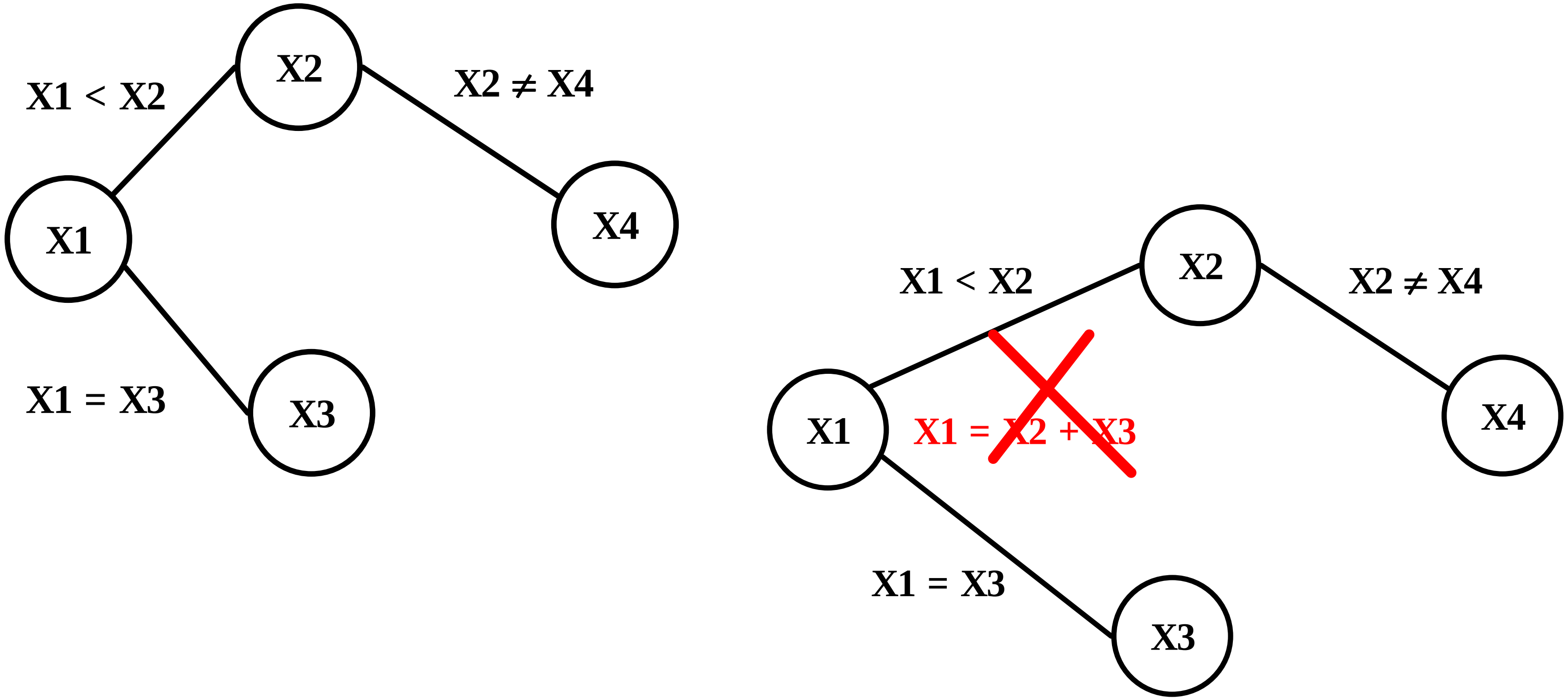
Une assignation est dite **cohérente** si les contraintes sont toutes vérifiées

Les CSPs

Quand les contraintes portent sur n variables au maximum, on parle de **CSP n-aires**.

Cas particulier des **CSP binaires**: ils ont une représentation graphique simple.

Les variables sont représentées par les nœuds et les contraintes par les arcs.



Les CSPs

- Une solution est une assignation totale qui respecte toutes les contraintes.
- Une solution partielle est une assignation partielle qui respecte toutes les contraintes.

Problèmes

- Trouver une solution
- trouver une solution qui satisfasse un critère
- trouver toutes les solutions
- déterminer le nombre de solutions
- déterminer si une assignation totale est une solution
- déterminer si une solution partielle peut faire partie d'une solution totale
- etc.

Les CSPs

Un CSP qui n'a pas de solution est dit **surcontraint**, **incohérent** ou **inconsistent**.

On peut se demander quelles sont les causes de cette incohérence (recherche d'explications)

Dans un premier temps on s'intéresse aux problèmes dans lesquels toutes les contraintes doivent être satisfaites (**contraintes "dures"**).

Mais dans la réalité, il est fréquent de traiter des pbs pour lesquels certaines contraintes (de **préférence, soft constraints**) peuvent être violées. Les contraintes peuvent être *hiérarchisées* en fonction d'un degré de préférence.

Exemple: les 4 reines

Pb: placer 4 reines sur un échiquier de manière qu'aucune reine ne soit attaquée par une autre reine.

On peut modéliser ce problème par :

	V_1	V_2	V_3	V_4
1				
2				
3				
4				

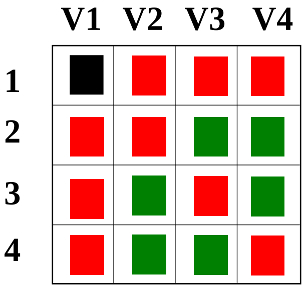
$$V = \{V_1, V_2, V_3, V_4\}$$

$$D = \{D_1, D_2, D_3, D_4\} \quad D_i = \{1, 2, 3, 4\}$$

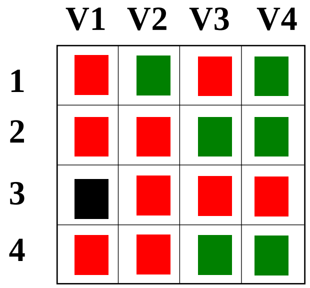
$$C = \{C_{1,2}, C_{1,3}, C_{1,4}, C_{2,3}, C_{2,4}, C_{3,4}\}$$

$C_{i,j}$ *indique la contrainte entre les variables V_i et V_j.*

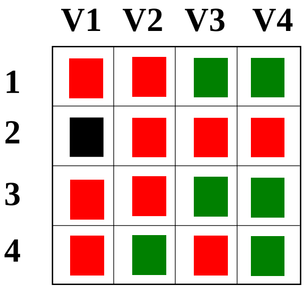
Exemple: les 4 reines



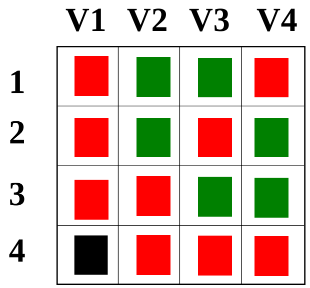
$$C_{1,2} = \{(1,3),(1,4),\ldots\}$$
$$C_{1,3} = \{(1,2),(1,4),\ldots\}$$
$$C_{1,4} = \{(1,2),(1,3),\ldots\}$$



$$C_{1,2} = \{\ldots,(3,1),\ldots\}$$
$$C_{1,3} = \{\ldots,(3,2),(3,4),\ldots\}$$
$$C_{1,4} = \{\ldots,(3,1),(3,2),(3,4),\ldots\}$$



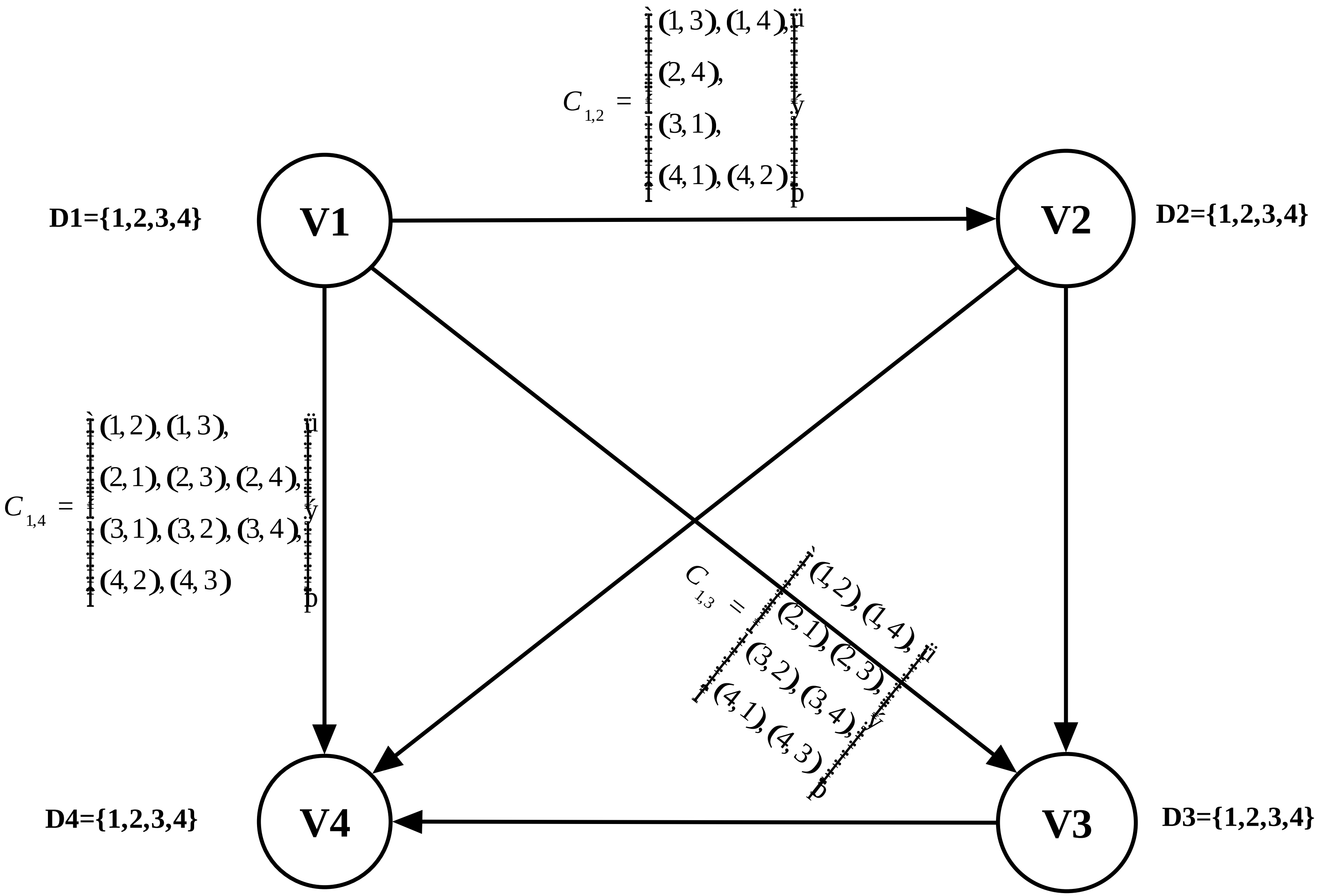
$$C_{1,2} = \{\ldots,(2,4),\ldots\}$$
$$C_{1,3} = \{\ldots,(2,1),(2,3),\ldots\}$$
$$C_{1,4} = \{\ldots,(2,1),(2,3),(2,4),\ldots\}$$



$$C_{1,2} = \{\ldots,(4,1),(4,2)\}$$
$$C_{1,3} = \{\ldots,(4,1),(4,3)\}$$
$$C_{1,4} = \{\ldots,(4,2),(4,3),\ldots\}$$

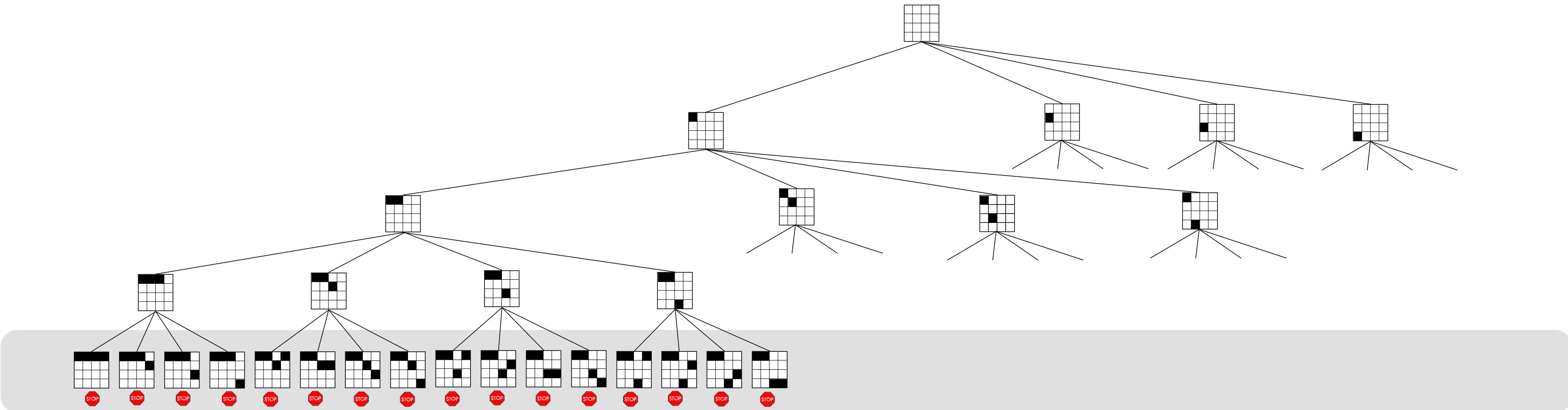
$$C_{1,2} = \{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)\}$$
$$C_{1,3} = \{(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)\}$$
$$C_{1,4} = \{(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,2),(4,3)\}$$

Exemple: les 4 reines



Exemple: les 4 reines

Pb: comment limiter la recherche des solutions dans l'ensemble de toutes les combinaisons possibles (256 cas possibles pour 4 reines) ?



4*4*4*4=256

Exemple: les 4 reines

```
queens(L):-
   length(L,8),
   L::[1..8],
   safe(L),
   labeling(L).
```

```
safe([]).
safe([X|L]):-
   noattack(L,X,1),
   safe(L).

noattack([],_,_).
noattack([Y|L],X,I):-
   diff(X,Y,I),
   I1 is I+1,
   noattack(L,X,I1).

diff(X,Y,I):-
   X#n=Y,
   X#n=Y+I,
   X+I#n=Y.
```

Exemple : le Sudoku

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

1

1

```
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
------+-------+------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
------+-------+------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
```

- $X = \{ x_{ij} \mid 1 \leq i \leq 9, 1 \leq j \leq 9 \}$
- $D_{ij} = \{1,2,\ldots,9\}$
- C:
  - “contraintes de valeur ou de domaine”: $x_{51} = 8$, $x_{81} = 7$, …
  - “contraintes de région”: $\boldsymbol{i,j} \in \{1,2,3\}^2$, $\boldsymbol{i \neq j}$, $x\boldsymbol{i} \neq x\boldsymbol{j}$ **, …**
  - “contraintes de ligne”: i,j $\in \{1..9\}$, $i \neq j$, $x_{1i} \neq x_{1j}$, …
  - “contraintes de colonnes”: i,j $\in \{1..9\}$, $i \neq j$, $x_{i1} \neq x_{j1}$, …

Exemple : le Sudoku

Les variables

Les domaines

Les contraintes

```
:- use_module(library('clp/bounds')).
:- use_module(library('clp/clp_distinct')).

/* Permet de résoudre un sudoku
   Entrée Grid: la grille de sudoku (0 => vide, [1-9] => ajoute une contrainte)
   Sortie Vars: une solution (permet de savoir si il existe plusieurs solutions) */

sudoku(Grid, Vars) :-
  % Definition des variables (grille de sudoku)
  Vars =
   [
    A1,A2,A3,A4,A5,A6,A7,A8,A9,
    B1,B2,B3,B4,B5,B6,B7,B8,B9,
    C1,C2,C3,C4,C5,C6,C7,C8,C9,
    D1,D2,D3,D4,D5,D6,D7,D8,D9,
    E1,E2,E3,E4,E5,E6,E7,E8,E9,
    F1,F2,F3,F4,F5,F6,F7,F8,F9,
    G1,G2,G3,G4,G5,G6,G7,G8,G9,
    H1,H2,H3,H4,H5,H6,H7,H8,H9,
    I1,I2,I3,I4,I5,I6,I7,I8,I9
   ],

  %%%%% REGLES DU JEU %%%%%
  % Les cases prennent des valeurs de 1 a 9
  Vars in 1..9,
  vars_in(Vars, 1, 9),

  % Tous les chiffres d'une ligne sont differents
  all_distinct([A1,A2,A3,A4,A5,A6,A7,A8,A9]),
  all_distinct([B1,B2,B3,B4,B5,B6,B7,B8,B9]),
  all_distinct([C1,C2,C3,C4,C5,C6,C7,C8,C9]),
  all_distinct([D1,D2,D3,D4,D5,D6,D7,D8,D9]),
  all_distinct([E1,E2,E3,E4,E5,E6,E7,E8,E9]),
  all_distinct([F1,F2,F3,F4,F5,F6,F7,F8,F9]),
  all_distinct([G1,G2,G3,G4,G5,G6,G7,G8,G9]),
  all_distinct([H1,H2,H3,H4,H5,H6,H7,H8,H9]),
  all_distinct([I1,I2,I3,I4,I5,I6,I7,I8,I9]),

  % Tous les chiffres d'une colonne sont differents
  all_distinct([A1,B1,C1,D1,E1,F1,G1,H1,I1]),
  all_distinct([A2,B2,C2,D2,E2,F2,G2,H2,I2]),
  all_distinct([A3,B3,C3,D3,E3,F3,G3,H3,I3]),
  all_distinct([A4,B4,C4,D4,E4,F4,G4,H4,I4]),
  all_distinct([A5,B5,C5,D5,E5,F5,G5,H5,I5]),
  all_distinct([A6,B6,C6,D6,E6,F6,G6,H6,I6]),
  all_distinct([A7,B7,C7,D7,E7,F7,G7,H7,I7]),
  all_distinct([A8,B8,C8,D8,E8,F8,G8,H8,I8]),
  all_distinct([A9,B9,C9,D9,E9,F9,G9,H9,I9]),
```

```
  % Tous les chiffres d'un carre sont differents
  all_distinct([A1,A2,A3,B1,B2,B3,C1,C2,C3]),
  all_distinct([A4,A5,A6,B4,B5,B6,C4,C5,C6]),
  all_distinct([A7,A8,A9,B7,B8,B9,C7,C8,C9]),
  all_distinct([D1,D2,D3,E1,E2,E3,F1,F2,F3]),
  all_distinct([D4,D5,D6,E4,E5,E6,F4,F5,F6]),
  all_distinct([D7,D8,D9,E7,E8,E9,F7,F8,F9]),
  all_distinct([G1,G2,G3,H1,H2,H3,I1,I2,I3]),
  all_distinct([G4,G5,G6,H4,H5,H6,I4,I5,I6]),
  all_distinct([G7,G8,G9,H7,H8,H9,I7,I8,I9]),

  %%%%% VALEUR DU PLATEAU %%%%%
  grid(Grid, Vars),

  %%%%% RECHERCHE DE SOLUTIONS %%%%%
  label(Vars),

  printGrid(Vars).

/* Permet d'initaliser la grille de sudoku
   Pour chaque élément différent de 0,
   pose la contrainte Var #= Val  */

grid([],[]) :- !.
grid([0|Q1],[_|Q2]) :-
  !,
  grid(Q1,Q2).

grid([Val|Q1],[Var|Q2]) :-
  !,
  vars_in([Var], [Val]),
  grid(Q1,Q2).

/* Un exemple de résolution */

sudokuExemple(Vars) :-
  sudoku(
   [ 8,0,4,0,0,0,2,0,9,
     0,0,9,0,0,0,1,0,0,
     1,0,0,3,0,2,0,0,7,
     0,5,0,1,0,4,0,8,0,
     0,0,0,0,3,0,0,0,0,
     0,1,0,7,0,9,0,2,0,
     5,0,0,4,0,3,0,0,8,
     0,0,3,0,0,0,4,0,0,
     4,0,6,0,0,0,3,0,1
   ], Vars
  ).
```

Instanciation

SEND+MORE=MONEY

Rechercher une valeur (nombre à 1 seul chiffre) pour chaque lettre de manière que l'addition SEND+MORE=MONEY soit correcte et que les lettres aient des valeurs différentes:

```
   r1 r2 r3 r4
      S E N D
+     M O R E
----------------
=   M O N E Y
```

$D_S=D_E=D_N=D_D=D_M=D_O=D_R=D_Y=\{0,1,2,3,4,5,6,7,8,9\}$

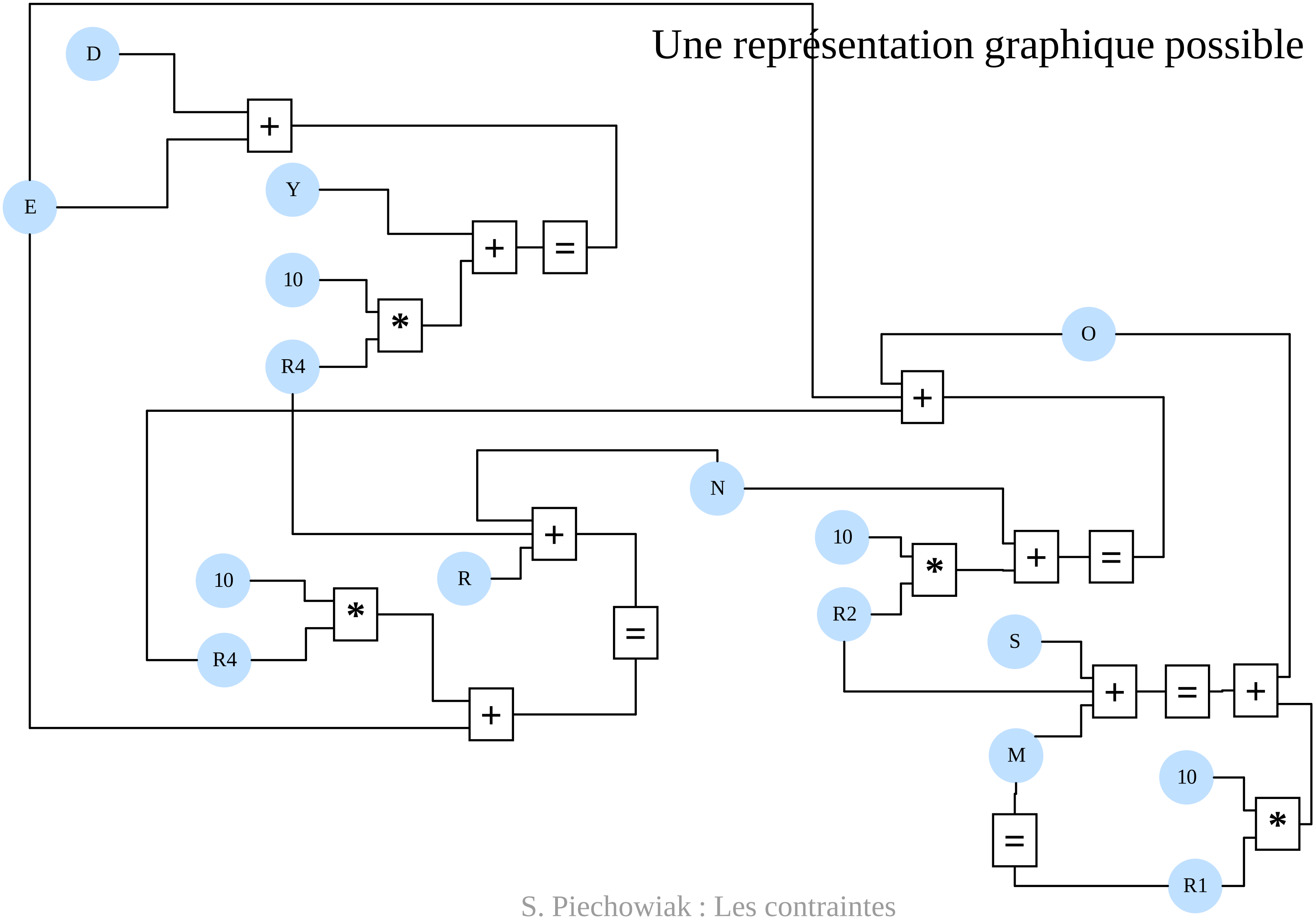
$$
\begin{aligned}
D + E &= Y + 10 \times r_4 \\
r_4 + N + R &= E + 10 \times r_3 \\
r_3 + E + O &= N + 10 \times r_2 \\
r_2 + S + M &= O + 10 \times r_1 \\
r_1 &= M
\end{aligned}
$$

$$\forall a \in \{0,\ldots,9\}, \forall b \in \{0,\ldots,9\}, a+b \in \{0,\ldots,18\}$$

$$\Rightarrow r_i \in \{0,1\}$$

SEND+MORE=MONEY

Une représentation graphique possible ...



SEND+MORE=MONEY

Exemple de programme PLPC

```
sendmoremoney(Vars) :-

   Vars =[S,E,N,R,D,M,O,Y],
   Vars in 0..9,

   all_different([S,E,N,D,M,O,R,Y]),
   S #\= 0,
   M #\= 0,
                 1000*S  +  100*E +  10*N  +  D
              +  1000*M  +  100*O +  10*R  +  E
     =  10000*M + 1000*O  +  100*N +  10*E  +  Y,

   labeling(Vars).
```

définition des variables et de leurs domaines

définition des contraintes à vérifier

recherche des solutions

Factorielle

Exemple de programme PLPC

```
:- use_module(library(clpfd)).

factorielle(N, 1).
factorielle(N, F) :-
   N #> 0,
   N1 #= N - 1,
   F #= N * F1,
   factorielle(N1, F1).
```

Quelle est la factorielle de 5 ?
?factorielle(5,F).
F=120

Quelle est le nombre N dont la factorielle est 120 ?
?factorielle(N,120).
N=5

Traitement des CSPs

La recherche d'une solution ou de toutes les solutions se fait par énumération. On dit souvent par backtracking. On distingue :

- l'étape de l'assignation d'une valeur (à choisir) à une variable (à choisir)
- l'étape du test de cohérence de l'assignation (partielle) courante
- l'étape (éventuelle) de remise en cause (*backtracking*) d'un choix précédent, en cas de situation d'échec.

Dans la communauté CSP on classe les algorithmes en 2 groupes :

- le groupe des algorithmes de type ***look-back ou rétrospectifs :*** il s'agit des algorithmes qui effectuent un travail après l'assignation d'une variable

- le groupe des algorithmes de type ***look-ahead ou prospectifs :*** il s'agit des algorithmes qui effectuent un travail de préparation avant l'assignation d'une variable

Le backtracking

- BT : backtracking chronologique
- BJ : backjumping

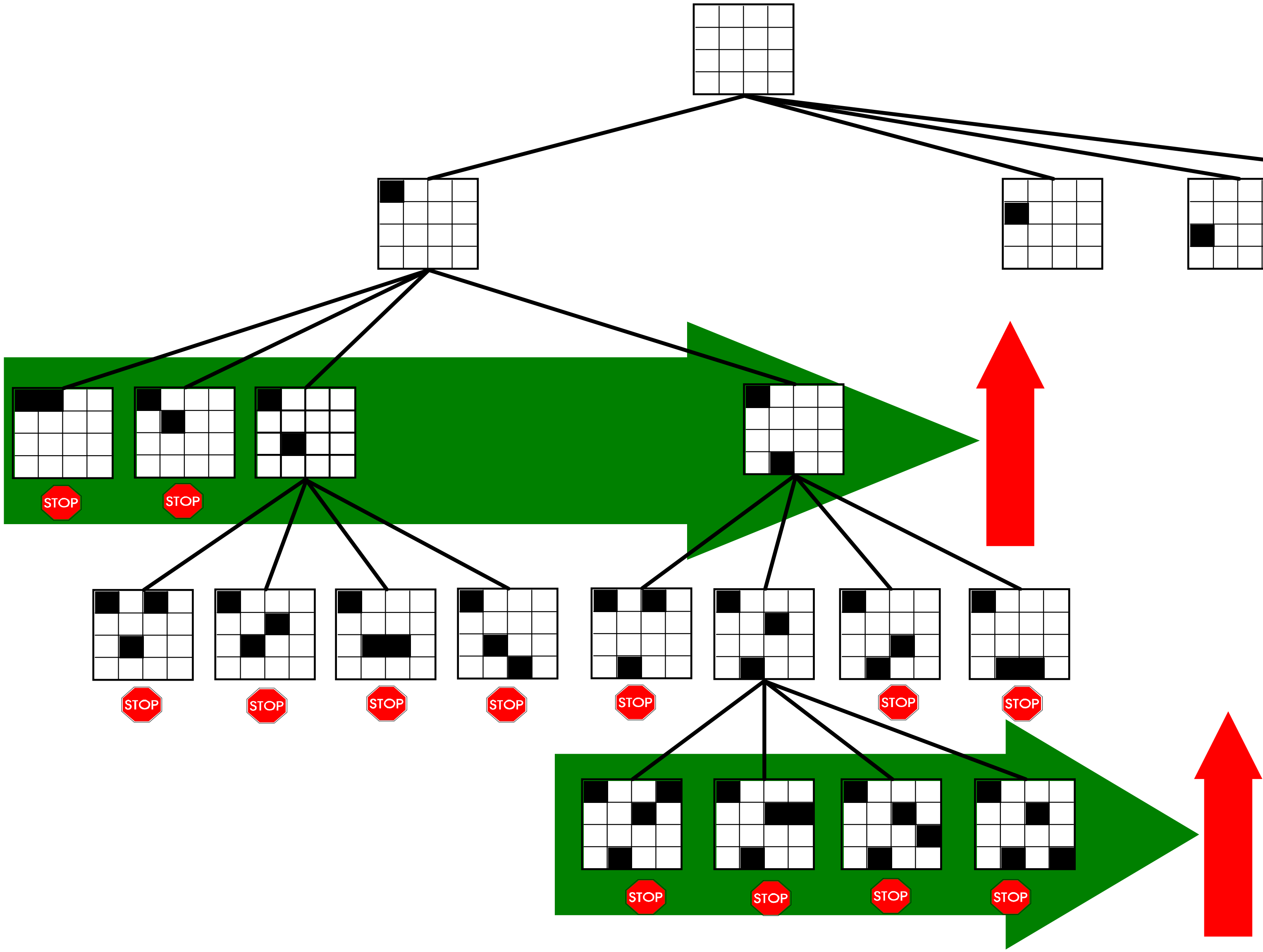
Le backtracking chronologique : BT

Objectif :

- Instancier progressivement les variables. Les valeurs et les variables sont ordonnées arbitrairement et de manière statique (l'ordre ne change pas en cours de traitement).

- Faire un retour arrière (*backtracking*) dès qu'on a essayé toutes les valeurs du domaine d'une variable et qu'à chaque fois on a été en présence d'une incohérence (au moins une contrainte est violée).

Le backtracking chronologique : BT



Le backtracking chronologique : BT

procédure BT

entrée : P = $(V=\{V_1,\ldots,V_n\}, D=\{D_1,\ldots,D_n\}, C=\{C_1,\ldots,C_m\})$

sortie : une solution ou *UNSAT* qui indique l'absence de solution

début

  i ← 1

  $D_i^* \leftarrow D_i$

  tantQue $1 \le i \le n$ **faire**

    soit {*x une valeur prise dans* D_i^* *telle que l'assignation courante soit cohérente*}

    ok ← false

    tantQue ¬ OK et $D_i^* \ne \emptyset$ faire

      soit $x \in D_i^*$: $D_i^* \leftarrow D_i^* - \{x\}$

      si {*l'assignation courante est cohérente*} **alors** OK ← true **finSi**

    finTantQue

    si ¬ OK {*x n'existe pas*}

      alors {*backtracking*}   i ← i – 1

      sinon i ← i + 1 ; $D_i^* \leftarrow D_i$

    finSi

  finTantQue

  si i = 0 **alors** renvoyer(*UNSAT*) **sinon** renvoyer(l'assignation courante) **finSi**

finProcédure

Le backtracking chronologique : BT

Forces & Faiblesses :

- Très facile à programmer !

- Souffre du phénomène de *trashing :* BT refait souvent plusieurs fois la même chose car lors d'un *backtrack*, BT oublie ce qui vient d'être fait.

- Le *backtrack* se fait systématiquement sur la dernière variable assignée. Mais ce n'est pas forcément elle qui provoque l'incohérence.

Le backtracking

Lorsqu'on fait un backtrack sur la variable X_{i-1}, on ne se préoccupe pas (à tord) de savoir s'il y a une contrainte entre X_i et X_{i-1}. Dans le backjumping, on prend en compte cette remarque.

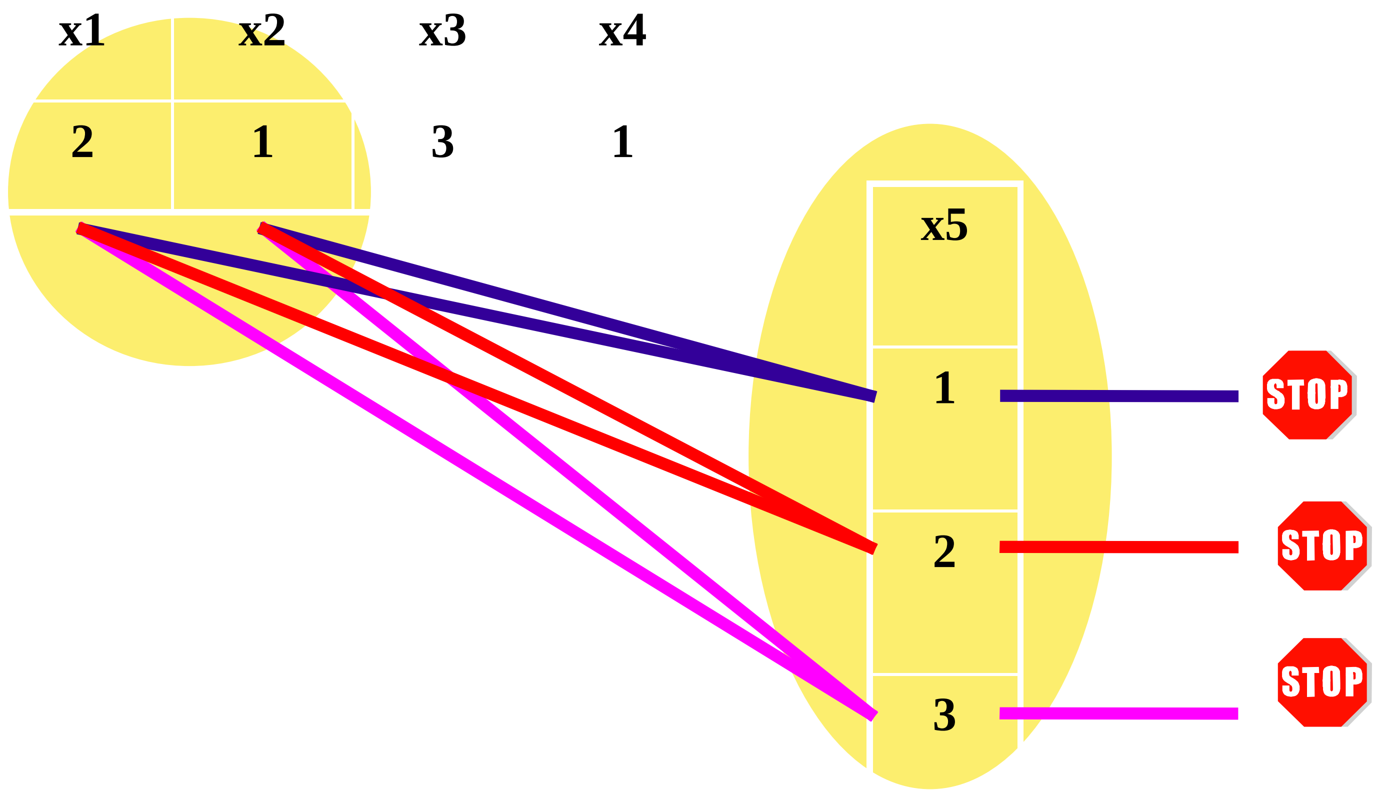
Pour celà, on utilise la notion de *conflit*.

Soit $\vec{V_i} = (V_1 = x_1, ..., V_i = x_i)$ une assignation cohérente des variables $V_1, ...,$ V_i, et soit X une variable qui n'est pas encore assignée.

Si pour chaque valeur v_X prise dans D_X, l'assignation ($X=v_X$) est incohérente avec $\vec{V_i}$ on dit que $\vec{V_i}$ est un *ensemble conflit* de X ou que $\vec{V_i}$ est en conflit avec X.

Le backjumping

idée



Supposons qu'en prenant : $x_1x_2x_5$ = **2 1 1**, x_1 x_2x_5 = **2 1 2** ou x_1 x_2 x_5 = **2 1 3** on aboutisse toujours à un échec. Dans ce cas on peut immédiatement conclure que x_1x_2 = **2 1** ne fera partie d'aucune solution. Il est donc inutile de remettre en cause les valeurs de x_3 ou x_4 tant que la valeur de x_1 vaut **2** et celle de x_2 vaut **1**

Le backjumping : définitions

Soit $\vec{a} = (a_1, \ldots, a_i)$ une instanciation consistante et soit x une variable non instanciée. Si aucune valeur de dom(x) n'est consistante avec $\vec{a}$, alors a est un ***ensemble conflit*** pour x (on dit que a est en conflit avec x).

Toute instanciation partielle $\vec{a}$ qui n'apparaît dans aucune solution est appelée ***no-good*** . Un no-good est dit ***minimal*** si aucun de ses sous uplets n'est un no-good.

Soit un ordre sur les variables: d = $(x_1, x_2, \ldots, x_n)$.
Un tuple $\vec{a}_i = (a_1, \ldots, a_i)$ consistant mais en conflit avec x_{i+1} est appelé état « ***i-leaf dead-ends*** » *(échec au-delà du niveau i)*

Le backjumping : définitions

Soit $\overrightarrow{a_i} = (a_1, ..., a_i)$ un état i-leaf dead-ends. On dit que x_j, $j \leq i$ est ***sauf*** si l'instanciation partielle $\overrightarrow{a_j} = (a_1, ..., a_j)$ est un no-good (qui ne peut être étendu à une solution).

Soit $\overrightarrow{a_i} = (a_1, ..., a_i)$ un état i-leaf dead-ends. ***L'indice b coupable*** relatif à est défini par $b = \min\{ j \leq i \,/\, \overrightarrow{a_j} \text{ est en conflit avec } x_{i+1} \}$
et on désigne x_b comme la variable couple de $\overrightarrow{a_i}$

Par définition $\overrightarrow{x_b}$ est un conflit minimal.

x_b est à la fois :

- sauf (car $\overrightarrow{x_b}$ ne peut être étendu à une solution)
- optimal (si on fait un backtrack plus long, on risque de perdre des solutions)

Si b est trop petit (le saut est trop grand) on perd des solutions.

Si b est trop grand on n'est plus optimal (moins efficace)

Le backjumping : BJ (Gaschnig)

J. Gaschnig, Performance measurement and analysis of search algorithms,
Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979

L'algorithme de Gaschnig utilise une technique de marquage : pour chaque variable, on maintient un pointeur vers le dernier prédécesseur trouvé comme incompatible avec toutes les valeurs de la variable.

Le backjumping : BJ (Gaschnig)

procédure Gaschnig-BJ **entrée** : P = $(V=\{V_1,\ldots,V_n\}, D=\{D_1,\ldots,D_n\}, C=\{C_1,\ldots,C_m\})$
début **sortie** : une solution ou *UNSAT* qui indique l'absence de solution

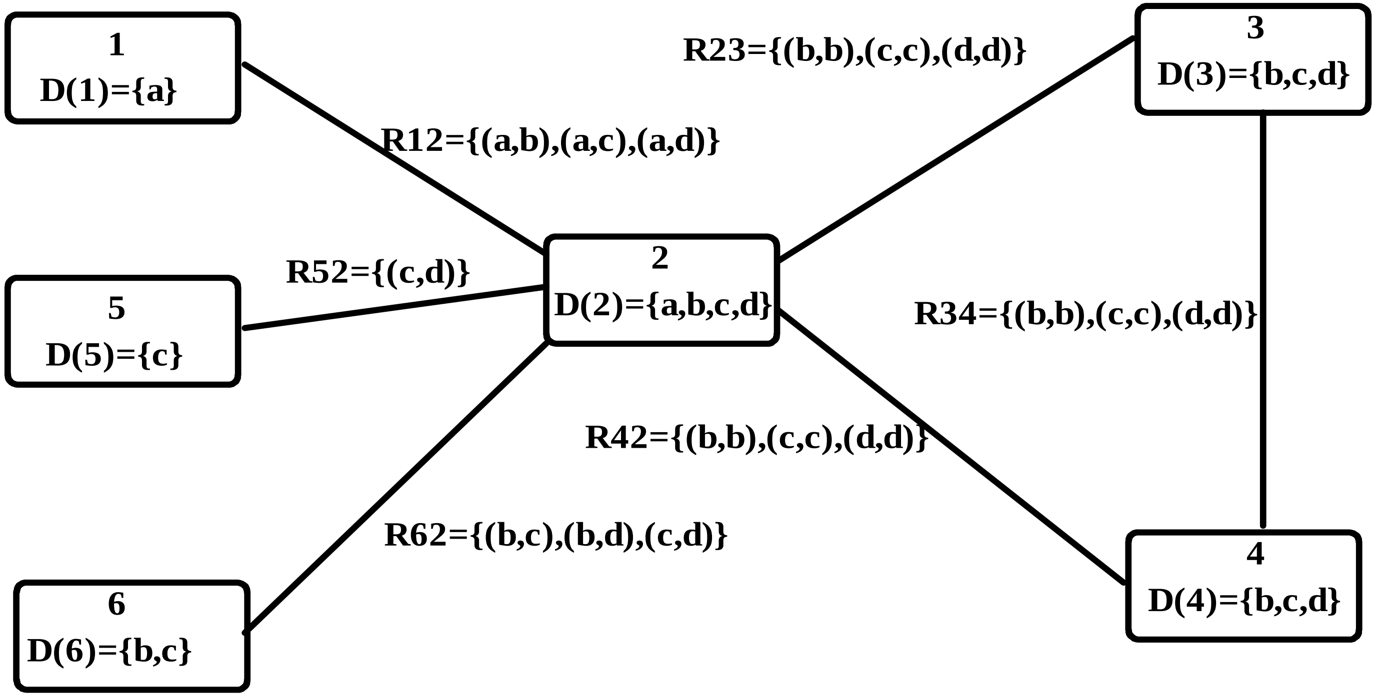
```
i ← 1
D_i* ← D_i
coupable_i ← 0 { coupable_i est un « pointeur » sur le coupable}
tantQue 1 ≤ i ≤ n faire
    soit {x une valeur prise dans D_i* telle que l'assignation courante soit cohérente}
    OK ← false
    tantQue ¬OK ∧ (D_i* ≠ {}) faire
        soit x ∈ D_i* ; D_i* ← D_i* − {x}
        consistant ← true
        k ← 1
        tantQue (k > i) ∧ consistant faire
            si k > coupable_i alors coupable_i ← k finSi
            si {l'assignation de V_1,…,V_k est en conflit avec (V_i=x)} alors consistant := false sinon k := k + 1 finSi
        finTantQue
        si consistant alors OK ← true finSi
    finTantQue
    si ¬OK {x n'existe pas}
        alors {backjumping} i ← coupable_i
        sinon i ← i + 1; D_i* ← D_i ; coupable_i ← 0
    finSi
    si i = 0  alors renvoyer(UNSAT) sinon renvoyer(l'assignation courante) finSi
finTantQue
finProcédure
```

Le backjumping : Graph Based BJ (GBJ)

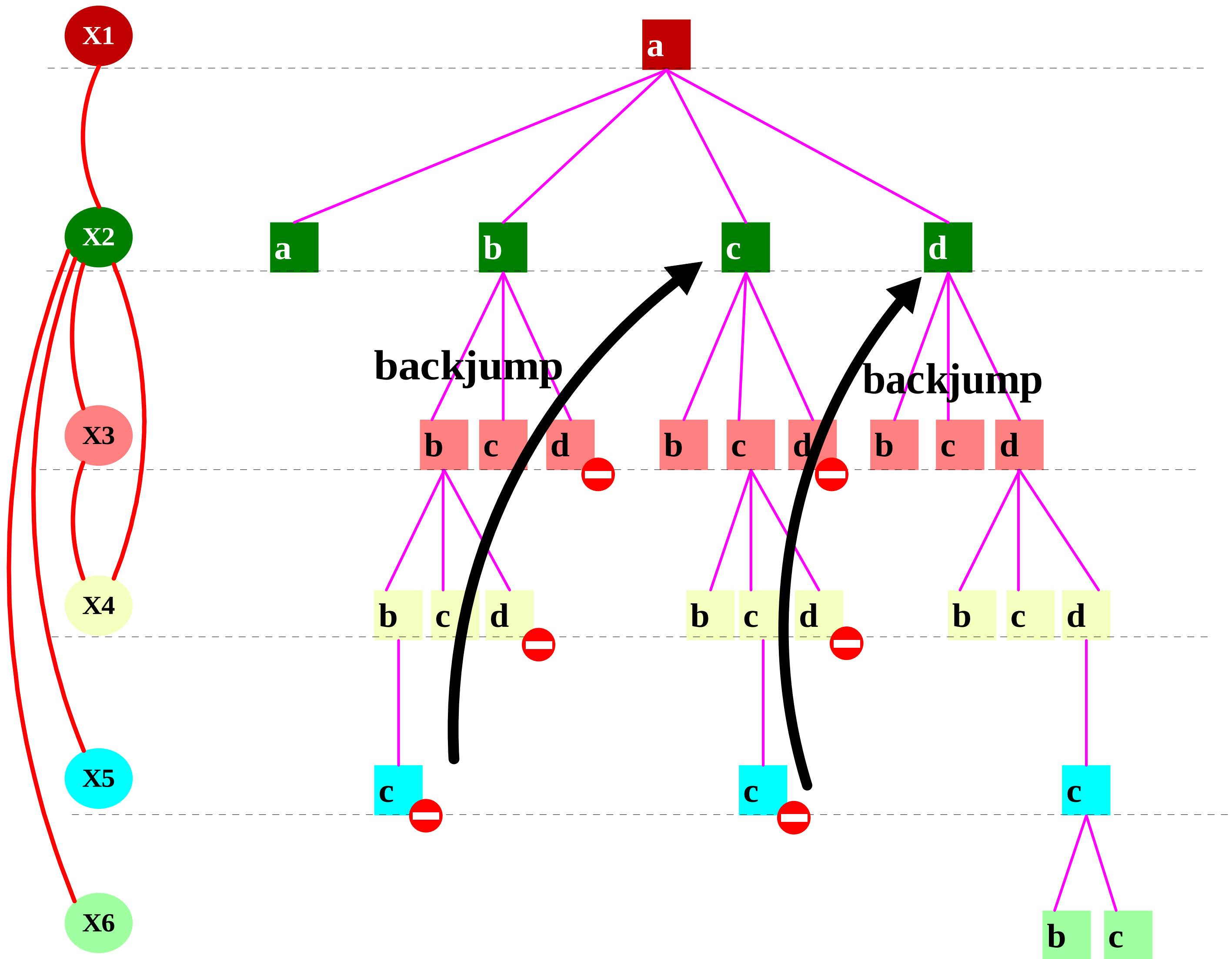
Quand on ne peut pas assigner de valeur à la variable X sans faire apparaître d'incohérence, le backjump se fait vers la variable la plus "récente" Y "connectée" à X. Si Y n'a plus de valeur, le backjump se fait sur Z, variable connectée à X ou Y et ainsi de suite ...

Il s'agit donc d'une méthode qui **exploite la structure du graphe** associé au CSP (le réseau de contraintes).

Le backjumping : exemple



Le backjumping : exemple



Le backjumping : Graph Based BJ (GBJ)

procédure G-BJ

entrée : P = $(V=\{V_1,\ldots,V_n\}, D=\{D_1,\ldots,D_n\}, C=\{C_1,\ldots,C_m\})$

sortie : une solution ou *UNSAT* qui indique l'absence de solution

début

calculer *anc*(V_i) pour chaque variable Vi

anc(V_i) représente l'ensemble des parents de V_i dans le graphe (réseau de contraintes).

$i \leftarrow 1$

$D_i^* \leftarrow D_i$

$I_i \leftarrow$ anc(V_i)

tantQue $1 \leq i \leq n$ **faire**

soit {*x une valeur prise dans* D_i^* *telle que l'assignation courante soit cohérente*}

OK ← false

tantQue ¬OK ∧ ($D_i^* \neq \{\}$) faire

soit $x \in D_i^*$; $D_i^* \leftarrow D_i^* - \{x\}$

si {*l'assignation* (V_i=*x*) *est cohérente avec l'assignation courante de* $V_1,\ldots,V_{i-1}$ } **alors** OK ← true **finSi**

finTantQue

si ¬OK {*x n'existe pas*}

alors {*backjumping*}

iprev ← i ; i ← le plus haut dans I_i ; $I_i \leftarrow I_i \cup iprev - \{V_i\}$

sinon $i \leftarrow i + 1$; $D_i^* \leftarrow D_i$; $I_i \leftarrow$ anc(V_i)

finSi

si i = 0 **alors** renvoyer(*UNSAT*) **sinon** renvoyer(l'assignation courante) **finSi**

finTantQue

finProcédure

Le backjumping : Conflict Directed BJ (CBJ)

P. Prosser, Forward checking with backmarking,
Technical Report AISL-48-93, University of Strathclyde, 1993

Au lieu d’exploiter les informations relatives à la structure du réseau de contraintes, Prosser propose d’exploiter des informations accumulées en cours de recherche.

On associe un ensemble *jumpback* à chaque variable.

Le backjumping : Conflict Directed BJ (CBJ)

entrée : P = ($V = \{V_1, \dots, V_n\}$, $D = \{D_1, \dots, D_n\}$, $C = \{C_1, \dots, C_m\}$)
sortie : une solution ou *UNSAT* qui indique l'absence de solution

procédure CBJ
début
  $i \leftarrow 1$
  $D_i^* \leftarrow D_i$
  $J_i \leftarrow \varnothing$ {ensemble de conflit}
  tantQue $1 \leq i \leq n$ **faire**
    soit {*x une valeur prise dans* D_i^* *telle que l'assignation courante soit cohérente*}
    OK ← false
    tantQue ¬OK ∧ ($D_i^* \neq \varnothing$) faire
      soit $x \in D_i^*$; $D_i^* \leftarrow D_i^* - \{x\}$
      k ← 1 ; *consistent* ← true
      tantQue (k<i) ∧ consistant **faire**
        si {*l'assignation (*V_i*=x) est en conflit avec l'assignation de* $V_1, \dots, V_k$ }
          alors $J_i \leftarrow J_i \cup \{ V_i \}$; consistent ← false **sinon** k ← k+1 **finSi**
      finTantQue
      si consistant **alors** OK ← true **finSi**
    finTantQue
    si ¬OK {*x n'existe pas*}
      alors {*backjumping*}
        iprev ← I ; i ← l'index le plus haut dans J_i ; $J_i \leftarrow J_i \cup iprev - \{V_i\}$
      sinon $i \leftarrow i + 1$; $D_i^* \leftarrow D_i$; $J_i \leftarrow \varnothing$
    finSi
    si i = 0 **alors** renvoyer(*UNSAT*) **sinon** renvoyer(l'assignation courante) **finSi**
  finTantQue
finProcédure

Autres algorithmes retrospectifs

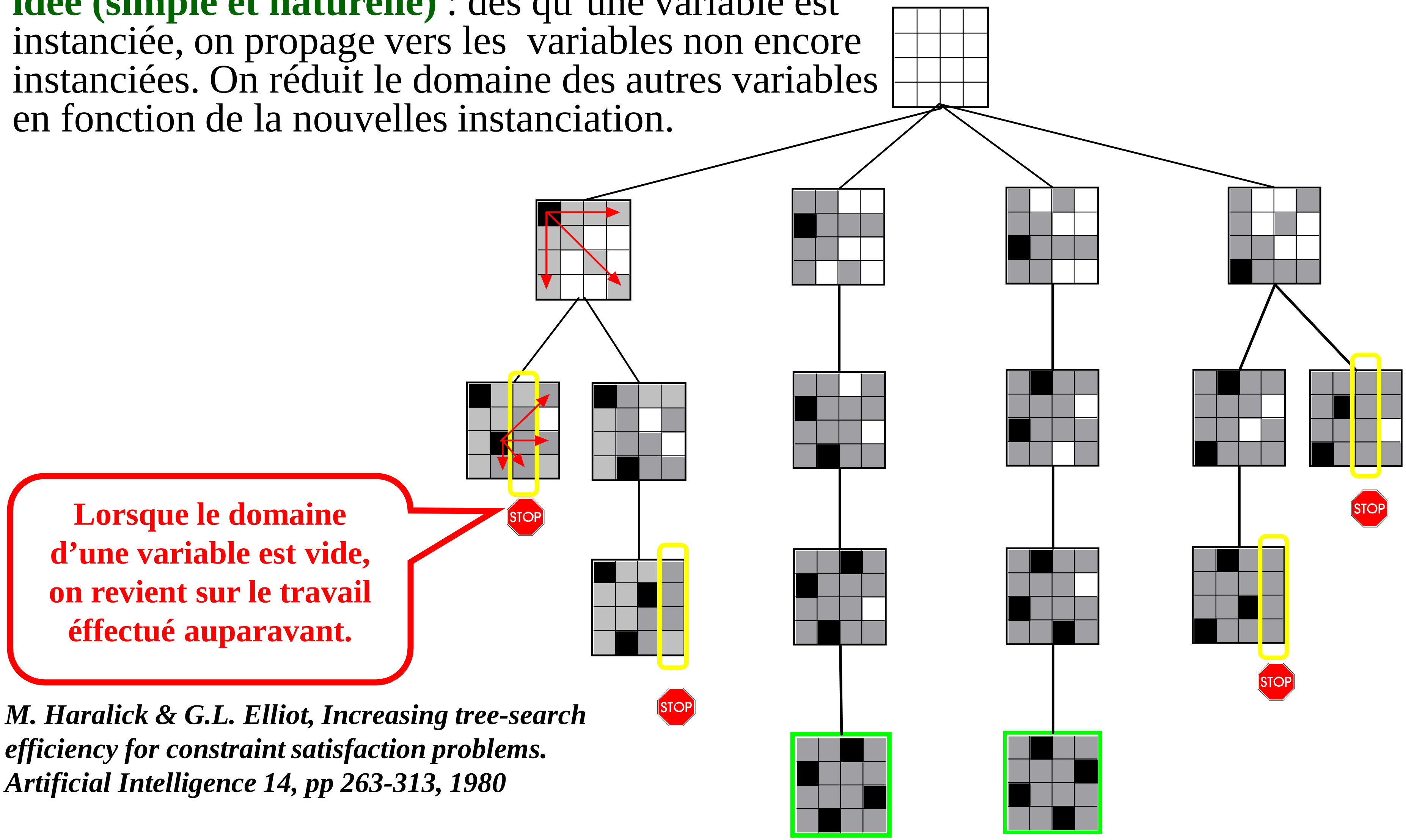
Le **backchecking** consiste à se rappeler que la valeur vi choisie pour la variable Xi en cours d'instanciation est incompatible avec la valeur vj pour une variable Xj précédemment instanciée. Tant que Xj aura la valeur vj, vi ne sera plus considérée pour Xi.

Apprentissage en cours de recherche

Dans les algorithmes précédents, on cherchait à maintenir un point de retour utilisé lors des backtracking. Ici on souhaite gérer (maintenir et exploiter) des informations lors de la recherche : on parle d'algorithmes avec *apprentissage*.

Forward-Checking

idée (simple et naturelle) : dès qu'une variable est instanciée, on propage vers les variables non encore instanciées. On réduit le domaine des autres variables en fonction de la nouvelles instanciation.



M. Haralick & G.L. Elliot, Increasing tree-search efficiency for constraint satisfaction problems. Artificial Intelligence 14, pp 263-313, 1980

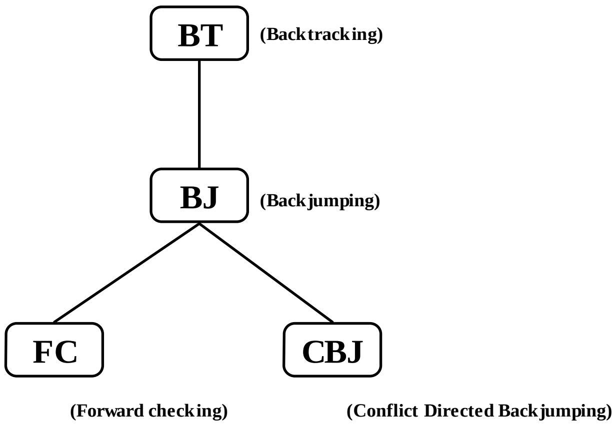
Forward-Checking

procédure FC
début
$D_i^* \leftarrow D_i$ **pour** $1 \leq i \leq n$
$i \leftarrow 1$
tantQue $1 \leq i \leq n$ faire
OK $\leftarrow$ *false*
tantQue $\neg$OK $\wedge$ ($D_i^* \neq \varnothing$) **faire**
soit $x \in D_i^*$; $D_i^* \leftarrow D_i^* - \{x\}$
domaineVide $\leftarrow$ *false*
pour tout k, $i < k \leq n$ **faire**
REVISE(k,i)
si $D_k^* = \varnothing$ **alors** domaineVide $\leftarrow$ *true* **finSi**
fait
si domaineVide
alors restaurer chaque D_k^* , $i < k \leq n$ tel qu'il était avant le choix de la valeur x
sinon OK $\leftarrow$ *true*
finSi
finTantQue
si $\neg$OK {*x n'existe pas*}
alors {*backtrack*}
restaurer chaque D^*, tel qu'il était avant l'instanciation de V_i
$i \leftarrow i$-1
sinon {*step forward*}
$i \leftarrow i$+1
finSi
finTantQue
finProcédure

procédure REVISE(m,n)
début
pour chaque valeur $a \in D_m^{'}$ faire
s'il $\forall b \in D_n^{'}\ \ V_1 = x_1, \ldots, V_i = x_i, V_m = a, V_n = b)$ est inconsistant
alors $D_m^{'} \leftarrow D_m^{'} - \{a\}$
finSI
fait

Elements de compléxité

nombre de noeuds visités



Grzegorz Kondrak, A theorical evaluation of selected backtracking algorithms, Dept of Computing Science, Edmonton, Alberta, 1994

Le filtrage

Objectif :

- Limiter la taille de l'espace de recherche des solutions.
- Supprimer les domaines les valeurs des variables qui ne sont dans aucune solution.
- On dit qu'on ***renforce la consistance*** ou qu'on filtre.

Il y a plusieurs niveaux de consistances. Ex: consistance locale, consistance de chemin

Plus la consistance est forte, plus l'espace de recherche est réduit mais plus il faut des traitements lourds et coûteux.

Pb: Peut-on trouver les solutions uniquement par filtrage ?

Algorithmes de filtrage : Node consistency

(Filtrage des nœuds)

Il s'agit d'un filtrage trivial.

On ne traite que les contraintes d'arité 1 (qui ne portent que sur une seule variable) : on parle également de contrainte sur les domaines.

procédure NC(i: une variable du CSP);
début
    $Di \leftarrow Di \cap \{x \mid Ri(x)\}$
fin

procédure NC(CSP : un CSP)
début
    pour tout i $\in$ CSP **faire** NC(i) **fait**
fin

A.K. Mackworth ; Consistency in networks of relations. Artificial Intelligence 8, p 99-118, 1977

Algorithmes de filtrage d'Arc (AC)

Ces algorithmes sont également qualifiés de *propagation de contraintes* et on dit qu'ils font du *filtrage local.*

Ils sont généralement simples à mettre en œuvre et fournissent un bon compromis entre réduction de l'espace de recherche et temps de calculs.

Historiquement l'idée est née dans le domaine de la vision (reconnaissance de forme) Waltz, 1975.

Algorithmes de filtrage : AC1

AC1 est trivial : on traite de manière cyclique tous les arcs du réseau. Dès qu'il est possible de réduire un domaine grâce à un arc on réitère le processus.

procédure AC1(G : un réseau);
début
  Q ← {(i, j) / arc(i,j) ∈ G ∧ i ≠ j }
  répéter
    CHANGE ← faux
    pour chaque (i,j) ∈ Q **faire**
      CHANGE ← (REVISE((i,j)) ∨ CHANGE))
    fait
  jusqu'à ¬ CHANGE
fin

fonction REVISE((i,j) un arc du réseau G):booléen;
début
  DELETE ← faux;
  pour chaque vi ∈ Di **faire**
    si $\{v_j \in D_j \ / \ R_{ij}(v_i,v_j)\} = \{\}$
      alors retirer x de D_i
        DELETE ← vrai
    fin_si
  fait
  renvoyer(DELETE)
fin

AC1 travaille en aveugle: même si le domaine d'une variable reste inchangé dans une itération, lors de l'itération suivante toutes les contraintes C(X,Y) seront revues alors que c'est inutile !

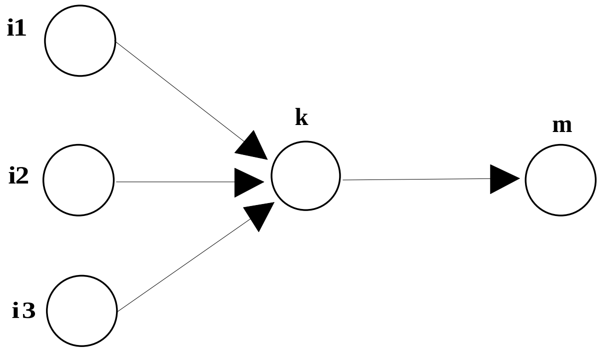
Algorithmes de filtrage : AC2

AC2 est une petite amélioration de AC1 qui utilise une queue de propagation.

procédure AC2(G : un réseau);
début
  pour i **de** 1 **à** n **faire**
  NC(i)
  Q1 $\leftarrow$ {(i,j) / (i,j) $\in$ arcs(G), j < i }
  Q2 $\leftarrow \varnothing$
  tant que Q1 $\neq \varnothing$ **faire**
    dépiler((k,m)) de Q1
    si REVISE((k,m))
      alors Q2 $\leftarrow$ Q2 $\cup$ {(p,k) / ((p,k) $\in$ arcs(G)) $\wedge$ (p $\leq$ i) $\wedge$ (p $\neq$ m)}
    fin_si
  fait
  Q1 $\leftarrow$ Q2
  Q2 $\leftarrow \varnothing$
  fait
fin

Algorithmes de filtrage : AC3

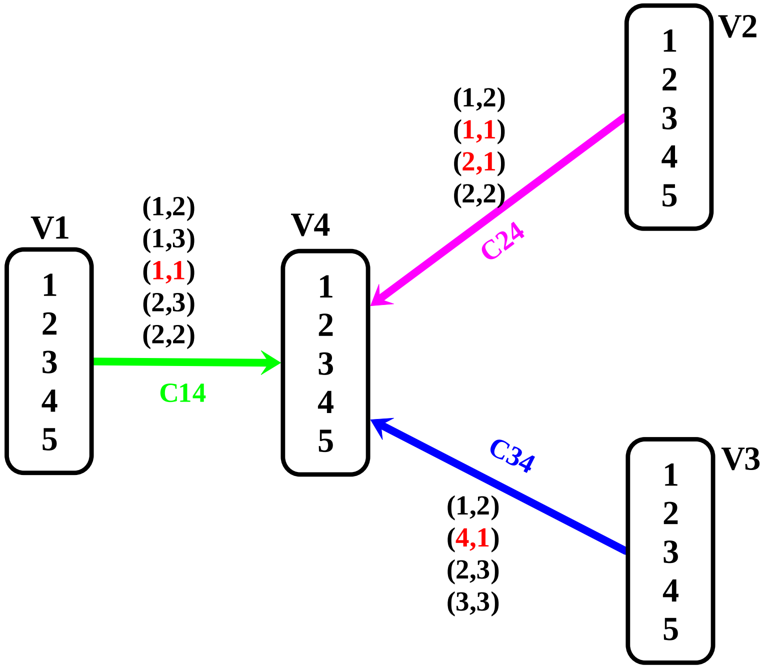
AC3 : on considère chaque arc (k,m). Si l'on modifie le domaine de la variable k, alors il y a des chances pour que l'on soit amené à modifier le domaine de ses noeuds prédécesseurs i_1, i_2, ..., i_p



procédure AC3(G : un réseau);
début
  pour i **de** 1 **à** n **faire** NC(i) **fait**;
  Q ← {(i,j) / (i,j) ∈ arcs(G), j ≠ i }
  tant que Q ≠ {} **faire**
    sélectionner et retirer un arc (k,m) de Q;
    si REVISE((k,m))
      alors Q ← Q ∪ {(i,k) / (i,k) ∈ arcs(G), i ≠ k ∧ i ≠ m }
    finSi
  fait
fin

Algorithmes de filtrage : AC4

Notion de support : Le ***support*** d'une valeur a pour une variable V_i par rapport à la contrainte C_{ij} est l'ensemble des valeurs b des autres variables V_j telles que $(a,b) \in C_{ij}(V_i,V_j)$.



la valeur 1 de V4 est "supportée" par :

- la valeur 1 de V1,
- les valeurs 1 et 2 de V2
- la valeur 4 de V3

la valeur 5 de V4 n'est pas supportée.

la valeur 3 de V4 est "supportée" par :

- la valeur 1 de V1,
- les valeurs 2 et 3 de V3

Rem: si b de V_j est un support de a pour V_i par rapport à la contrainte C_{ij}, alors, a de V_i est aussi un support de b pour V_i par rapport à la contrainte C_{ij},

R. Mohr, T. Henderson, Arc and path consistency revisited.
Artificial Intelligence 28 pp 225-233, 1986

C.C. Han, C.H. Lee, Comments on Mohr an Henderson's path consitency algorithm.
Artificial Intelligence 36 pp 125-133, 1988

Algorithmes de filtrage : AC4

Idée : dès qu'une valeur n'a plus de support, elle peut être supprimée du domaine auquel elle appartient ce qui entraîne une révision des supports des valeurs qu'elle supporte.

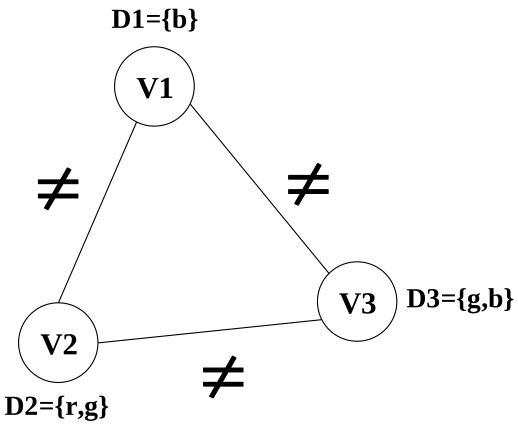
On construit l'ensemble des paires **[(i,j),b]** qui représentent les relations **R_{ij}(b,?)**. A chaque paire on associe le nombre de valeurs v telles que **R_{ij}(b,v)** est vérifiée. A chaque valeur c du nœud j, on associe l'ensemble :

$$S_{j,c} = \left\{ (i,b) / \mathrm{R}_{ij}(b,c) \text{ la valeur } c \text{ du noeud } j \text{ supporte la valeur } b \text{ du noeud } i \right\}$$

Dès que la valeur c est supprimée du nœud j, on décrémente la valeur du compteur associé à la paire [(i,j),b]. Si ce compteur est nul cela signifie que la valeur b du nœud i n'a plus de support et elle doit, à son tour être supprimée du nœud i

Algorithmes de filtrage : AC4

Exemple



$$S_{V_3,g} = \{(V_1,b)(V_2,r)\}$$

Algorithmes de filtrage : AC4

```
procédure AC4(G : un réseau);
début
initAC4(G)
{ phase de propagation }
tant que List ≠ {} faire
   retirer (j,c) de List
   pour chaque (i,b) ∈ Sjc faire
      compteur[(i,j),b] ← compteur[(i,j),b] - 1
      si (compteur[(i,j),b] = 0) ∧ (M[i,b] = 0)
         alors   ajouter (i,b) dans List
                 M[i,b] ← 1
                 Di ← Di - {b}
      finSi
   fait
fait
fin
```

$M[i,b] = 0 \Leftrightarrow b \notin dom(V_i)$

```
procédure initAC4(G : un réseau);
début
{ mise en place de la structure de données }
initialiser M avec 0
Sib ← {}
   pour chaque (i,j) ∈ arcs(G) faire
      pour chaque valeur consistante b de Di faire
         total ← 0
         pour chaque valeur consistante c de Dj faire
            si Rij(b,c)
               alors   total ← total + 1
                       Sjc ← Sjc ∪ {(i,b)}
            finSi
         fait
         si total = 0
            alors   M[i,b] ← 1
                    Di ← Di - {b}
            sinon   compteur[(i,j),b] ← total
         finSi
      fait
   fait
List ← {(i,b) / M[i,b] = 1}
fin
```

Algorithmes de filtrage : AC6

AC6 se base également sur la notion de support mais au lieu de gérer tous les supports de chaque valeur, ici on se limite à gérer **l'existance d'au moins un support pour chaque valeur**. Une valeur a de la variable i est dite **viable** si pour chaque contrainte C_{ij} la variable j possède une valeur b qui supporte a. Seules les valeurs viables doivent être conservées, les autres peuvent être supprimées.

$$\forall a \in i, a \text{ viable } \Rightarrow \forall C_{i,j}, \exists b \in \mathrm{D}_j / \mathrm{R}_{i,j}(a,b)$$

De ce fait, AC6 est une version allégée de AC4: elle est plus efficace notamment en place mémoire).

C. Bessière, Arc-consistency and arc-consistancy again. Artificial Intelligence 65 pp 179-190, 1994

Algorithmes de filtrage : éléments de complexité

Eléments caractéristiques d'un CSP :

- taille des domaines (*a* = taille du plus grand domaine)
- nombre de variables (*n*)
- nombre de contraintes (*e*)

algo.	meilleur	moyen	pire	mémoire	observations
AC1			$O(a^3ne)$	$O(e+na)$	
AC2					
AC3	$O(a^2e)$		$O(a^3e)$	$O(e+na)$	
AC4			$O(a^2e)$	$O(a^2e)$	dans [Wallace, 93], l'auteur montre que le pire des cas est une configuration particulière et rare, ce qui fait qu'en général AC3 est meilleur que AC4.
AC6			$O(a^2e)$	$O(ae)$	

Algorithmes de filtrage : Consistances plus fortes

Peut-on filtrer « plus fortement » ?

OUI ! au lieu de travailler uniquement avec des relations entre 2 variables on peut travailler avec des relations sur 3 variables ou plus.

Déf.

- Un CSP est ***k-consistant*** si toute affectation de k-1 variables peut être étendu en une affectation de k variables
- Un CSP est ***fortement k-consistant*** si $\forall j \leq k$, il est j-consistant
- Un CSP est ***chemin-consistant*** s'il est fortement 3-consistant
- Un CSP est ***(i,j)-consistant*** si toute affectation de i variables peut être étendue en une affectation de i+j variables

JC Régin
CAC: Un algorithme d'arc consistance configurable, générique et adaptatif
JNPC'2004

Et pour les contraintes n-aires, n > 2 ?

On définit la notion d'Hyper-consistance d'arc

une contrainte C sur les variables $x_1, \ldots, x_n$ de domaines : $D_1, \ldots, D_n$

est dite ***hyper-consistante d'arc par rapport*** à : $x_{i \in \{1,\ldots,n\}}$

ssi : $\forall a \in D_i,$

$$\exists b_1 \in D_1, \ldots, \exists b_{i-1} \in D_{i-1}, \exists b_{i+1} \in D_{i+1}, \ldots, \exists b_n \in D_n$$

tel que : $C(b_1, \ldots, b_{i-1}, a, b_{i+1}, \ldots, b_n)$

C est dite ***hyper-consistante d'arc*** ssi elle est hyper-consistante d'arc par rapport à tout $x_{i \in \{1,\ldots,n\}}$

Au lieu d'***hyper-consistante d'arc*** on parle également de consistance d'arc généralisé (**GAC**)

Et pour les contraintes n-aires, n > 2 ?

Autre présentation

Une valeur x d'une variable V est dite ***GAC dans une contrainte*** C si:
- ou bien V n'est pas directement concernée par C : $V \notin var(C)$
- ou bien V est concernée directement par C, $V \in var(C)$, et il existe une assignation de toutes les autres variables de C telle que $(V \leftarrow x)$ et ces autres assignations sont consistantes.

Une valeur x d'une variable V est dite **GAC** si elle est GAC pour chaque contrainte.

Un **CSP est GAC** si toutes ses valeurs (valeurs de ses variables) sont toutes GAC.

Différences entre réseaux

Réseau de contraintes complet : un réseau de contraintes est dit complet si chacune de ses variables est connectée à toutes les autres variables (il y a une contrainte entre chaque paire de variables).

Tous les réseaux de contraintes ne sont pas complets et il est important dans ces cas de mesurer la densité du réseau.

Densité : La densité d'un réseau est le ratio entre le nombre de ses arcs et le nombre des arcs du réseau comportant les mêmes variables mais complet.

Difficulté d'un problème – Tightness : La difficulté d'un problème de satisfaction de contraintes avec les variables $X_1 \ldots X_n$, est le ratio entre le nombre de ses solutions et le cardinal du produit cartésien : $D_1 \times D_2 \times \ldots \times D_n$

Exemple:

Le problème des N reines est complet (densité = 1/1=1)

Le problème des 4 reines possède 2 solutions

Sa difficulté vaut : $2/(4 \times 4 \times 4 \times 4) = 1/128 = 0{,}0078125$

Le problème des 8 reines possède 92 solutions

Sa difficulté vaut : $92/(8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8) = 92/16777216 = 0{,}00000549$

Différences entre réseaux

Il est important de faire la distinction entre la difficulté d'un probème et la difficulté à résoudre ce problème.

On peut avoir :

- des problèmes difficiles mais faciles à résoudre
- Des problèmes faciles mais difficiles à résoudre

Différences entre réseaux

- La difficulté à résoudre un problème de satisfaction de contraintes dépend à la fois de la densité du graphe et de sa difficulté.
- Pour tester les algorithmes de résolution des CSP il est habituel d'utiliser des générateurs de problèmes aléatoires. Les paramètres de ces générateurs sont non seulement le nombre d'arcs mais également la densité du graphe et la difficulté des contraintes.
- Les études dans ce domaine de la générations de CSP montrent que les CSP présentent souvent une phase de transition qui sépare les problèmes qui sont trivialement satisfaits de ceux qui sont trivialement insatisfaits.

Extensions des CSP

- **les CSP traitent des classes de problèmes (booléens, réels, domaines finis, etc.)**
- **comment traiter judicieusement l'ajout / le retrait de contraintes : les CSP dynamiques**

Extensions des CSP : traitement des intervalles de valeurs

Pb des domaines finis:

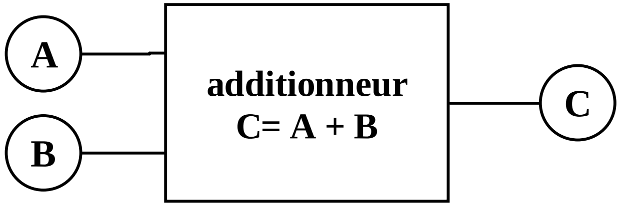
- **Les valeurs sont traités de manière individuelle notamment lors des filtrages.**
- **Ils ne permettent pas de manipuler les imprécisions.**

$C-B = [80,100]-[50,90]=[80-90,100-50]=[10,50]$

$A \leftarrow A \cap C-B = [30,70] \cap [10,50] =$ **[30,50]**

$\{30,31,\ldots,70\} = [30,70]$

$\{50,51,\ldots,90\} = [50,90]$



$\{1,2,\ldots,100\}=[1,100]$

$A+B = [30,70]+[50,90]=[80,160]$

$C \leftarrow C \cap A+B = [1,100] \cap [80,160] =$ **[80,100]**

$C-A = [80,100]-[30,50]=[80-50,100-30]=[30,70]$

$B \leftarrow B \cap C-A = [50,90] \cap [30,70] =$ **[50,70]**

On a réduit les domaines des 3 variables en 3 étapes seulement !

Extensions des CSP : traitement des intervalles de valeurs

+

$$(+\infty) + x = x + (+\infty) = +\infty, \forall x \in \{R+, N+\} \cup \{+\infty\}$$

$$(-\infty) + x = x + (-\infty) = -\infty, \forall x \in \{R-, N+\} \cup \{-\infty\}$$

$(+\infty) + (-\infty)$ et $(-\infty) + (+\infty)$ ne sont pas définis

---

-

$$(+\infty) - x = (+\infty), \forall x \in \{R-, N-\} \cup \{-\infty\}$$

$$x - (+\infty) = (-\infty), \forall x \in \{R-, N-\} \cup \{-\infty\}$$

$$(-\infty) - x = (-\infty) =, \forall x \in \{R+, N+\} \cup \{+\infty\}$$

$$x - (-\infty) = (+\infty), \forall x \in \{R+, N+\} \cup +\{-\infty\}$$

$(+\infty) - (-\infty)$ et $(-\infty) - (+\infty)$ ne sont pas définis

---

×

$$(+\infty) \times x = x \times (+\infty) = signe(x) * (+\infty), \forall x \in \{\square, \square\} \cup \{-\infty, +\infty\}, |x| \geq 1$$

$$(-\infty) \times x = x \times (-\infty) = -signe(x) * (+\infty), \forall x \in \{\square, \square\} \cup \{-\infty, +\infty\}, |x| \geq 1$$

Extensions des CSP : traitement des intervalles de valeurs

On souhaite pouvoir manipuler des intervalles de valeurs dans des expressions arithmétiques

$$[5,+\infty[\,+\,[50,100] = [55,+\infty[$$

Il faut redéfinir une ***arithmétique sur les intervalles [Moore 66]***

Par exemple, si on travaille sur $\mathbb{R}$:

$$[A1,B1]+[A2,B2]=[A1+A2,B1+B2]$$

$$[A1,B1]-[A2,B2]=[A1-B2,B1-A2]$$

$$[A1,B1]\times[A2,B2]=[\min\{A1\times A2, A1\times B2, B1\times A2, B1\times B2\}, \max\{A1\times A2, A1\times B2, B1\times A2, B1\times B2\}]$$

$$1\div[B1,B2] = [1/B2,1/B1], \text{ si } 0\notin [B1,B2]$$

Extensions des CSP : traitement des intervalles de valeurs

Si on travaille sur si on travaille sur R $\cup$\{-∞ , +∞\}

[A1, +∞[+ [A2,B2] = [A1+A2, +∞[

]-∞, A2] + [A2,B2] =]-∞, A2+B2]

]-∞, +∞[+ [A2,B2] =]-∞, +∞[

L'arithmétique peut être étendue aux fonctions classiques :

$$x^n \quad \sqrt{x} \quad e^x \quad \log(x) \quad \sin(x) \quad \cos(x) \quad \tan(x)$$

en travaillant sur les intervalles sur lesquels les fonctions sont monotones, dans ce cas on peut être amené à traiter des *ensembles d'intervalles*

Pb1 : multiplication des intervalles
Pb2 : on perd des propriétés de l'arithmétique classique

Extensions des CSP : traitement des intervalles de valeurs

Pb1 : multiplication des intervalles
Pb2 : on perd des propriétés de l'arithmétique classique

$0 + A = [0,0] + [a,b] = [0+a,0+b] = [a,b] = A$
$1 \times A = [1,1] \times [a,b] = [\min\{a\times1, b\times1, a\times1, b\times1\}=a, \max\{a\times1, b\times1, a\times1, b\times1\}=b]= [a,b] = A$
$A + A = [a,b] + [a,b] = [a+a,b+b] = [2a,2b] = 2A$

$A - A = [a,b] - [a,b] = [a-b,b-a] \neq [0,0] = 0$

$$\begin{aligned} A \times (B+C) &= [a1,a2] \times ([b1,b2]+[c1,c2]) \\ &= [a1,a2] \times [b1+c1,b2+c2] \\ &= [\min\{a1\times(b1+c1),a1\times(b2+c2), a2\times(b1+c1),a2\times(b2+c2)\}, \\ &\quad \max\{a1\times(b1+c1),a1\times(b2+c2),a2\times(b1+c1),a2\times(b2+c2)\}] \end{aligned}$$

$$\begin{aligned} (A\times B)+(A\times C) &= ([a1,a2]\times[b1,b2])+([a1,a2]\times[c1,c2]) \\ &= [\min\{a1\times b1,a1\times b2, a2\times b1,a2\times b2\},\max \{a1\times b1,a1\times b2,a2\times b1,a2\times b2\}] \\ &\quad +[\min\{a1\times c1,a1\times c2, a2\times c1,a2\times c2\}, \max \{a1\times c1,a1\times c2,a2\times c1,a2\times c2\}] \\ &= [\min\{a1\times b1,a1\times b2, a2\times b1,a2\times b2\}+\max\{a1\times c1,a1\times c2, a2\times c1,a2\times c2\}, \\ &\quad \max \{a1\times b1,a1\times b2,a2\times b1,a2\times b2\}+\max\{a1\times c1,a1\times c2, a2\times c1,a2\times c2\}] \end{aligned}$$

Tout ce qu'on peut dire ici : $A \times (B+C) \subseteq (A\times B)+(A\times C)$ on parle de sous-distributivité de la multiplication % à l'addition

Extensions des CSP : traitement des intervalles de valeurs

A chaque relation, est associé un ensemble de règles de maintien de la cohérence, ces règles vérifient les 4 propriétés suivantes :

contractantes : après application des règles de maintenance, l'intervalle de chaque variable est inclus ou égal à l'intervalle précédent.

correct : après application des règles de maintenance, toute valeur d'une instanciation satisfaisant la contrainte est toujours dans l'intervalle (i.e. on ne perd pas de solution).

monotone : l'inclusion des intervalles est préservée par l'application des règles.

idempotente : une seule application des règles de maintenance suffit à calculer les nouveaux intervalles des variables. Une autre application ne modifie plus ces intervalles.

Extensions des CSP : traitement des intervalles de valeurs

Conséquences: la mise en forme d'une expression va influencer la « qualité » du filtrage !

Exemple: on considère le polynôme : $P(X)=X^4-10X^3+35X^2-50X+24$

avec $X=[0,4]$

on a : $P(0)=24$, $P(1)=0$, $P(2) = 0$, $P(3) = 0$, $P(4) = 0$

Évaluation directe:

$$P([0,4]) = [0,4]^4-10\times[0,4]^3+35\times[0,4]^2-50\times[0,4]+24$$
$$=[0,256]-10\times[0,64]+35\times[0,16]-50\times[0,4]+24$$
$$=[0,256]-[0,640]+[0,560]-[0,200]+[24,24]$$
$$=[0,256]+[-640,0]+[0,560]+[-200,0]+[24,24]$$
$$=[-816,840]$$

Extensions des CSP : traitement des intervalles de valeurs

Schéma de Hörner:

$$P(X) = X^4 - 10X^3 + 35X^2 - 50X + 24 = 24 + X(-50+X(35+X(-10+X)))$$

$$P([0,4])=[-256,384]$$

P([0,4]) = 24+[0,4]×(-50+[0,4]×(35+[0,4]×(-10+[0,4])))
=[24,24]+[0,4]×([-50,-50]+[0,4]×([35,35]+[0,4]×([-10,-10]+[0,4])))
=[24,24]+[0,4]×([-50,-50]+[0,4]×([35,35]+[0,4]×[-10,-6]))
=[24,24]+[0,4]×([-50,-50]+[0,4]×([35,35]+[-40,0]))
=[24,24]+[0,4]×([-50,-50]+[0,4]×[-5,35])
=[24,24]+[0,4]×([-50,-50]+[-20,140])
=[24,24]+[0,4]×[-70,90]
=[24,24]+[-280,360]
=[-256,384]

Expansion des racines:

$$P(X) = X^4 - 10X^3 + 35X^2 - 50X + 24 = (X-1)\times(X-2)\times(X-3)\times(X-4)$$

$$P([0,4])=[-24,72]$$

P([0,4]) = ([0,4]-1)×([0,4]-2)×([0,4]-3)×([0,4]-4)
=([0,4]-[1,1])×([0,4]-[2,2])×([0,4]-[3,3])×([0,4]-[4,4])
=[-1,3]×[-2,2]×[-3,1]×[-4,0]
=[-6,6]×[-4,12]
=[-24,72]

Extensions des CSP : traitement des intervalles de valeurs

Forme centrée de Moore:

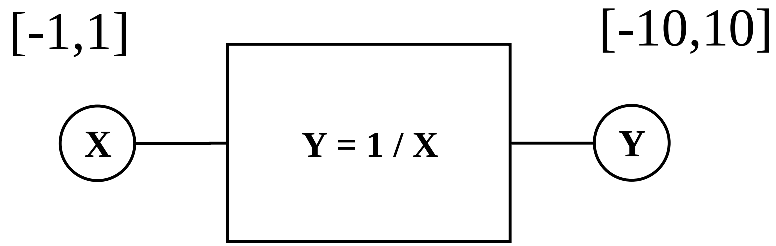
Pour une fonction $f(x_1,\ldots,x_n)$, sa forme centrée est :
$Fc(Y_1,\ldots,Y_n)=f(m_1,\ldots,m_n)+G(X_1-m_1,\ldots,X_n-m_n)$
m_i est le centre de l'intervalle X_i et $Y_i= X_i-m_i$
L'idée est de « décaler » les intervalles en argument de la fonction pour qu'ils chevauchent la valeur 0 ce qui « minimise » les valeurs aux limites et permet d'obtenir un encadrement plus restreint.

$P(X) = X^4-10X^3+35X^2-50X+24$
$P(Y) = Y^4-2Y^3-Y^2+2Y+P(2)$ avec m=2 et Y=[-2,2]

d'où l'on tire P([0,4])=[-24,36]

$P([-2,2]) = [-2,2]^4-2\times[-2,2]^3-[-2,2]^2+2\times[-2,2]+0 =$
$=[-16,16]-2\times[-8,8]-[-4,4]+2\times[-2,2]$
$=[-16,16]+[-16,16]+[-4,4]+[-4,4]$
$=[-40,40]$

Extensions des CSP : traitement des intervalles de valeurs



$$( ]-\infty,-1] \cup [1,+\infty[ ) \cap [-10,10] = [-10,-1] \cup [1,10]$$

$$\frac{1}{[-10,-1] \cup [1,10]} \cap [-1,1] = [-1,-\frac{1}{10}] \cup [\frac{1}{10},1]$$

On a fait apparaître des « trous » dans les intervalles.
Sachant que $[-10,-1] \cup [1,10] \subseteq [-10,10]$
Il n'est pas faux de dire que les 2 CSP ont les mêmes solutions.
Cependant, si on réduit le domaine de Y à $[-10,-1] \cup [1,10]$, le filtrage effectué est plus « fort ». En terme de propagation on doit traiter des unions d'intervalles au lieu d'intervalles.

Extensions des CSP : traitement des intervalles de valeurs

On appelle :

- F : tout sous-ensemble de P $\cup$ {$-\infty$,$+\infty$} (ensemble des flottants)
- F-intervalle : tout intervalle [a,b] où {a,b} $\subseteq$ F
- I(F) : l'ensemble des F-intervalles
- U(F) : { D $\subseteq$ P / $\exists I_1 \in$ I(F),…, $\exists I_n \in$ I(F) : D = $I_1 \cup \ldots \cup I_n$ }

L'approximation (*approx*) d'une relation r est le plus petit élément (au sens de l'inclusion ensembliste) de U(F) contenant r. On approxime chaque n-uplets de réels qui satisfont r par une union d'intervalles.

Un ICSP est ***intervalle-consistant*** ssi pour toute variable X_i de domaine D_i et pour toute contrainte $C(X_1,\ldots,X_i,\ldots,X_n)$

$$D_i = approx\left(D_i \cap \left\{ \begin{array}{l} a_i \in \square \ / \exists a_1 \in D_1, \ldots, \exists a_{i-1} \in D_{i-1}, \exists a_{i+1} \in D_{i+1}, \ldots, \exists a_n \in D_n \\ \text{avec } C(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) \text{ satisfaite} \end{array} \right\}\right)$$

Extensions des CSP : traitement des intervalles de valeurs

Un ICSP est ***enveloppe-consistant*** ssi pour toute variable X_i de domaine D_i et pour toute contrainte $C(X_1,\ldots,X_i,\ldots,X_n)$

$$D_i = envelop(D_i \cap \left\{ \begin{array}{l} a_i \in \mathbb{R} \;/\; \exists a_1 \in D_1,\ldots,\exists a_{i-1} \in D_{i-1}, \exists a_{i+1} \in D_{i+1},\ldots,\exists a_n \in D_n \\ \text{avec } C(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n) \text{ satisfaite} \end{array} \right\})$$

L'enveloppe (notée envelop) d'une relation r est le plus petit F-intervalle contenant r.

Un ICSP est boîte-consistant ssi pour toute variable X_i de domaine D_i et pour toute contrainte $C(X_1,\ldots,X_i,\ldots,X_n)$

$$D_i = envelop(D_i \cap \left\{ \begin{array}{l} a_i \in \mathbb{R} \;/\; \text{l'extension de la contrainte C} \\ \text{aux intervalles } (D_1,\ldots,D_{i-1},[a_i,a_i],D_{i+1},\ldots,D_n) \\ \text{est non vide} \end{array} \right\})$$

Extensions des CSP : traitement des intervalles de valeurs

Un ICSP est ***boite-consistant*** ssi pour toute variable X_i de domaine D_i et pour toute contrainte $C(X_1,\ldots,X_i,\ldots,X_n)$

$$D_i = \mathit{envelop}\left(D_i \cap \left\{ \begin{array}{l} a_i \in \mathbb{R} \ / \ \text{l'extension de la contrainte C} \\ \text{aux intervalles } \left(D_1,\ldots,D_{i-1},[a_i,a_i],D_{i+1},\ldots,D_n\right) \\ \text{est non vide} \end{array} \right\}\right)$$

C’est la consistance qui est généralement traitée dans les solveurs (ILOG-SOLVER, CLP(BNR), etc.

Extensions des CSP : traitement des intervalles de valeurs

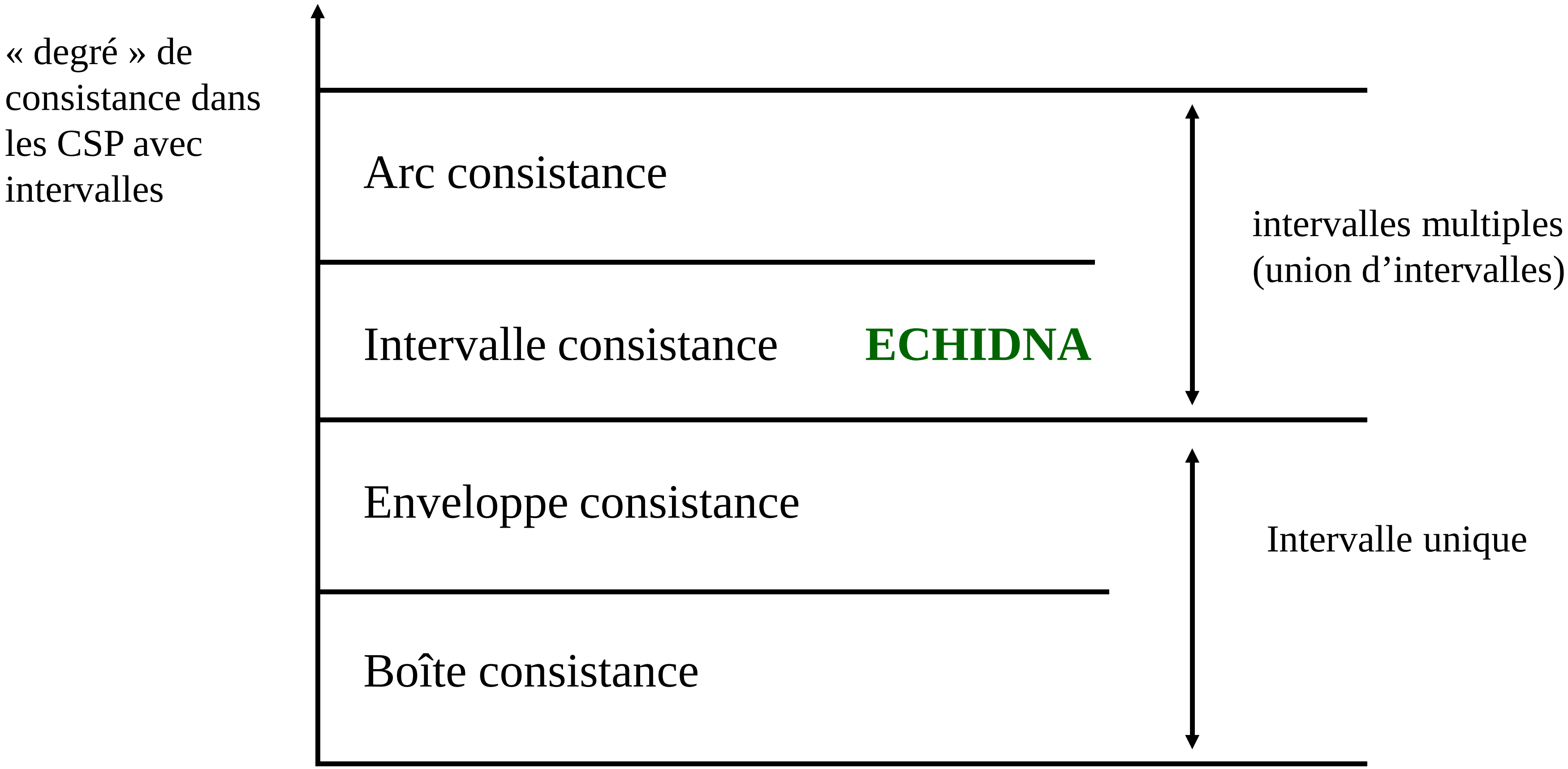
Pour les systèmes ne conservant qu'un intervalle unique on définit l'arc consistance aux bornes ou arc-B-consistance [Lhomme, 93].

Un ICSP (X,D,C) est arc-B-consistant ssi:
$\forall x \in X$ telle que $D_x=[a,b]$
$\forall x\ C(X_1,\ldots,X_i,\ldots,X_n) \in C$
on a:

$$\exists v_1,\ldots,v_k \in D_1 \otimes \ldots \otimes D_k \,/\, C(a,v_1,\ldots,v_k) \text{ satisfaite}$$
$$\exists v_1,\ldots,v_k \in D_1 \otimes \ldots \otimes D_k \,/\, C(b,v_1,\ldots,v_k) \text{ satisfaite}$$

Le CSP (X={x,y} ; D={dom(x}=[-2,2], dom(y)=[1,4]} ; C={y=x^2})
est arc-B-consistant mais il n'est pas arc-consistant (sur les entiers)
car 0 dans dom(x) n'est supporté par aucune valeur de dom(y)

Extensions des CSP : traitement des intervalles de valeurs



Les CSP dynamiques

Problématique

Il n'est pas rare qu'un CSP, qui représente le modèle d'un problème doive être modifié pour :

- corriger des contraintes,
- ajouter de contraintes oubliées
- ajouter des contraintes redondantes pour *améliorer la propagation locale*,
- retirer des contraintes,
- modifier des domaines,
- etc.

Si le CSP n'a pas de solution, l'utilisateur peut-il se satisfaire d'une réponse telle que **`"no-solution"`**. Ne peut on pas déterminer des contraintes à "***relaxer***" pour que le CSP ait des solution s?

Objectif

Proposer des moyens d'ajouter, retirer ou modifier un CSP sans remettre en cause tout le travail effectué.

Les CSP dynamiques

Définition Un CSP dynamique **P** (noté DnCSP) est une suite de CSP (statiques) $P_0, ..., P_i, P_{i+1}, ...$ tels que chaque P_i diffère de P_{i-1} par l'**ajout** ou le **retrait** d'une contrainte

On note $P_i=(X,dom,C(i))$ où $C(i)$ représente l'ensemble des contraintes présentes dans le CSP P_i. Au départ, on a : $P_0 = (X, dom, \{\})$

Rem1: lors de l'ajout d'une contrainte à un CSP, l'ensemble des solutions est **au plus conservé** ; généralement il est réduit. On parle de *restriction*.

Conséquence: si on désigne par $\Phi(P_i)$ le CSP obtenu après l'application d'un filtrage Φ sur le CSP P_i, les CSP $\Phi(P_i)$ et P_i ont les mêmes solutions. De même, les CSP $\Phi(P_{i+1})$ et P_{i+1} ont les mêmes solutions (puisque le filtrage préserve les solutions). Si $C(i+1) = C(i)\cup\{\alpha\}$on a $sol(P_{i+1}) \subseteq sol(P_i)$

Les CSP dynamiques

Rem2: Si $C(i+1) = C(i) \cup \{\alpha\}$

alors : $\Phi(P_{i+1}) = \Phi(P_i \cup \{\alpha\}) = \Phi(\Phi(P_i) \cup \{\alpha\})$

Conséquence: lors de l'ajout d'une contrainte à un CSP il est inutile de "refaire tout le travail de filtrage" on peut partir du filtrage réalisé juste avant l'ajout. Ceci se fait très facilement

Par contre pour le retrait d'une contrainte α, il faut restituer aux différents domaines des variables toutes les valeurs qui avaient été supprimées lors de l'ajout de α.

Le moyen le plus trivial consisterait à repartir au point de départ mais il s'agit d'une méthode très inefficace.

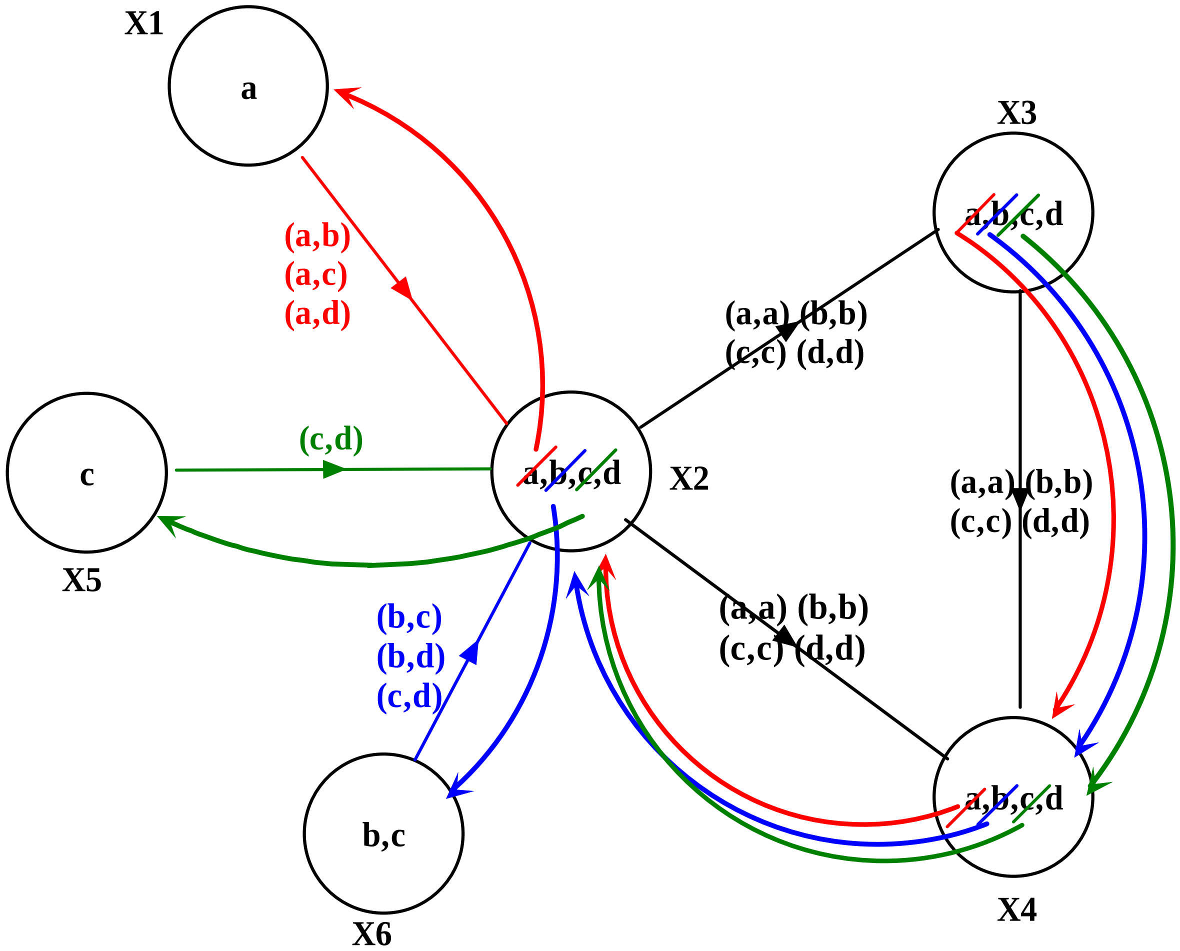
Le mieux consiste à mémoriser des informations qui permettent de retrouver facilement ces valeurs préalablement supprimées. C'est ce que font les algorithmes de type DnACX.

Les CSP dynamiques : DnAC4

Idée : maintenir la consistance d'arc selon AC4 à chaque ajout / retrait de contrainte

chaque valeur possède un **support** par rapport aux contraintes (qq chose qui justifie le maintient d'une valeur dans un domaine)

Les CSP dynamiques : DnAC4



étape 1 : AJOUT C23

étape 2 : AJOUT C34

étape 3 : AJOUT C24

étape 4 : AJOUT C12

étape 5 : AJOUT C62

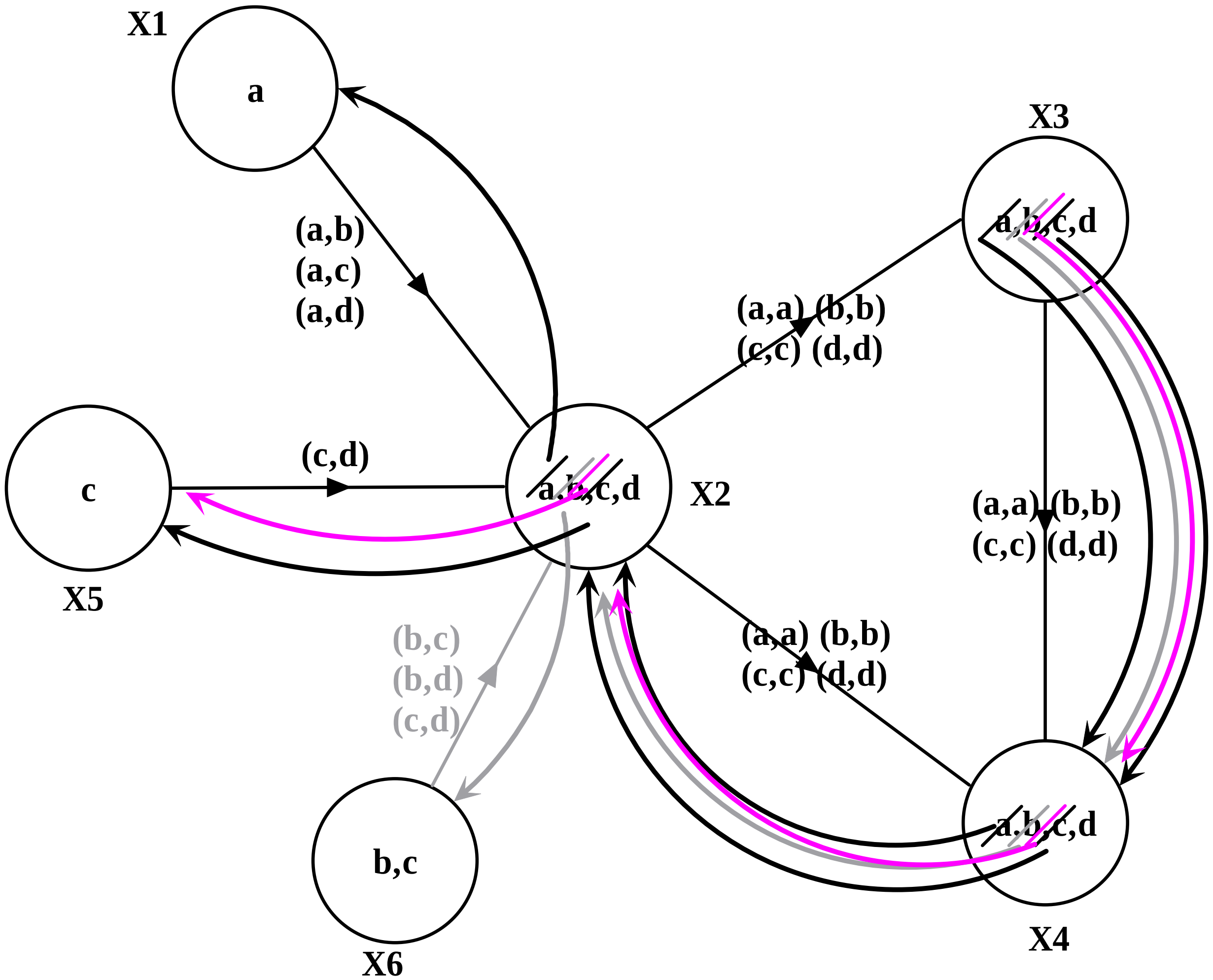
étape 6 : AJOUT C52

Les CSP dynamiques : DnAC4

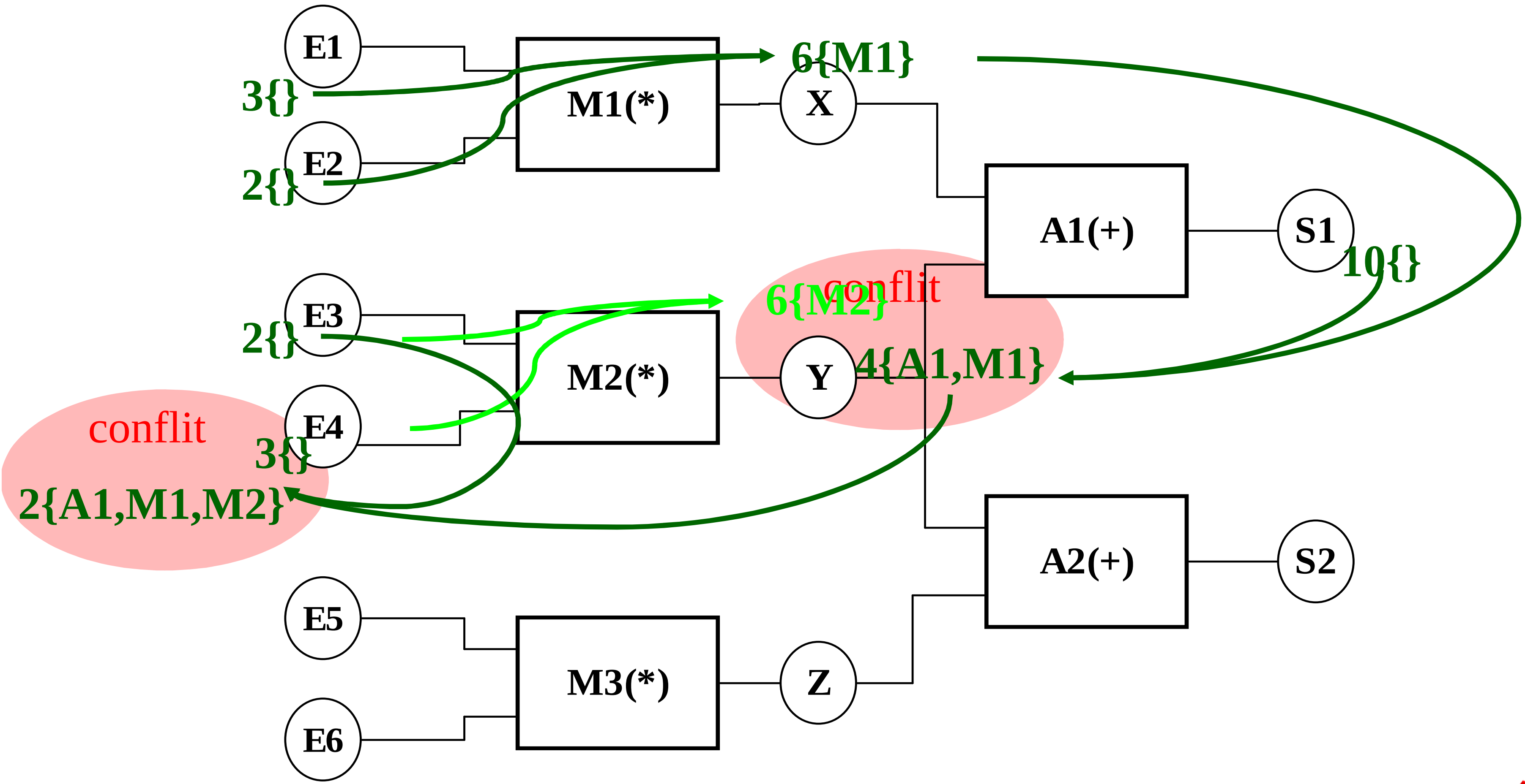
étape 7 : RETRAIT C62

retrait des justifcations

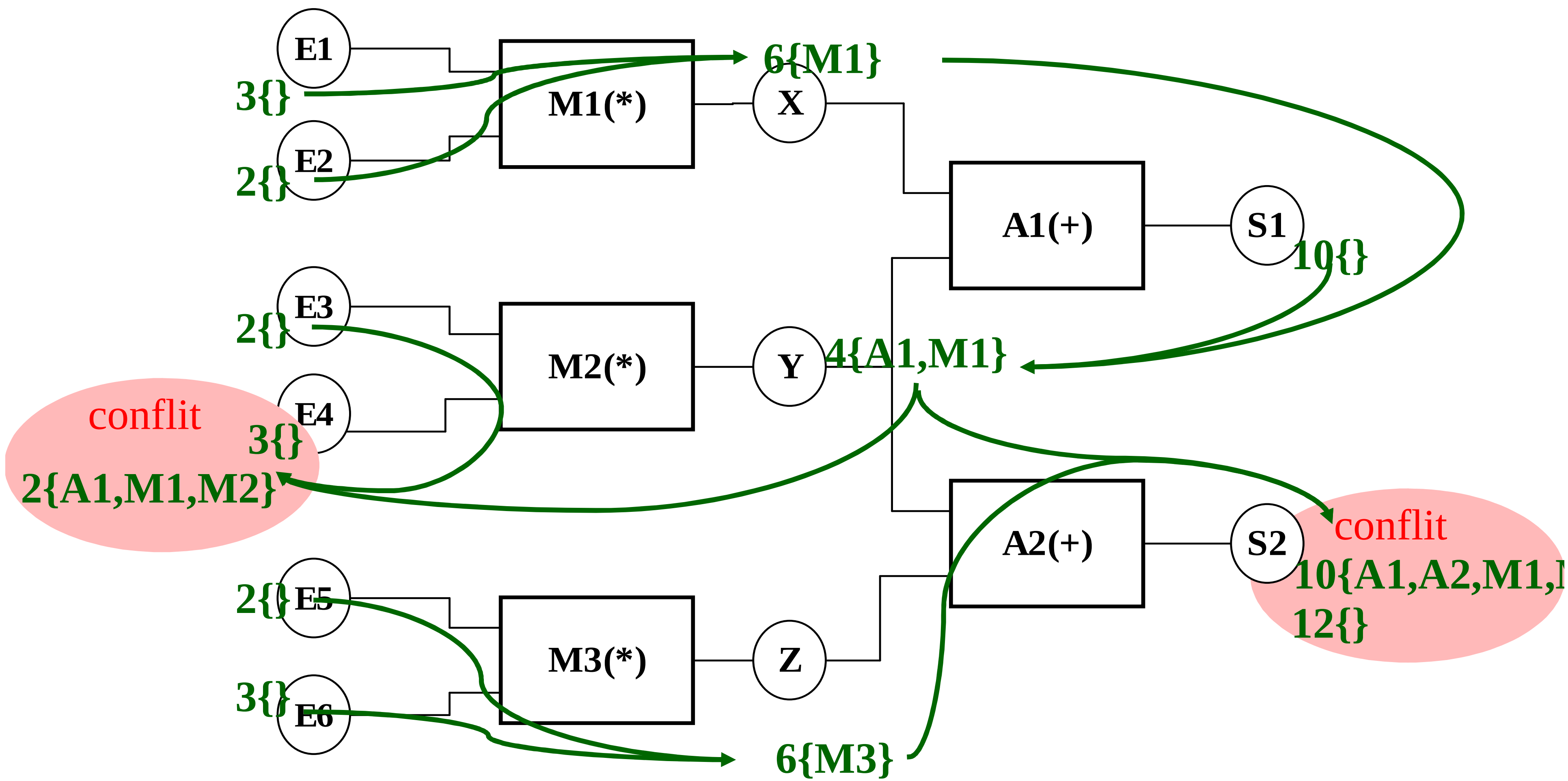
ajout de nouvelles justifications



Propagation

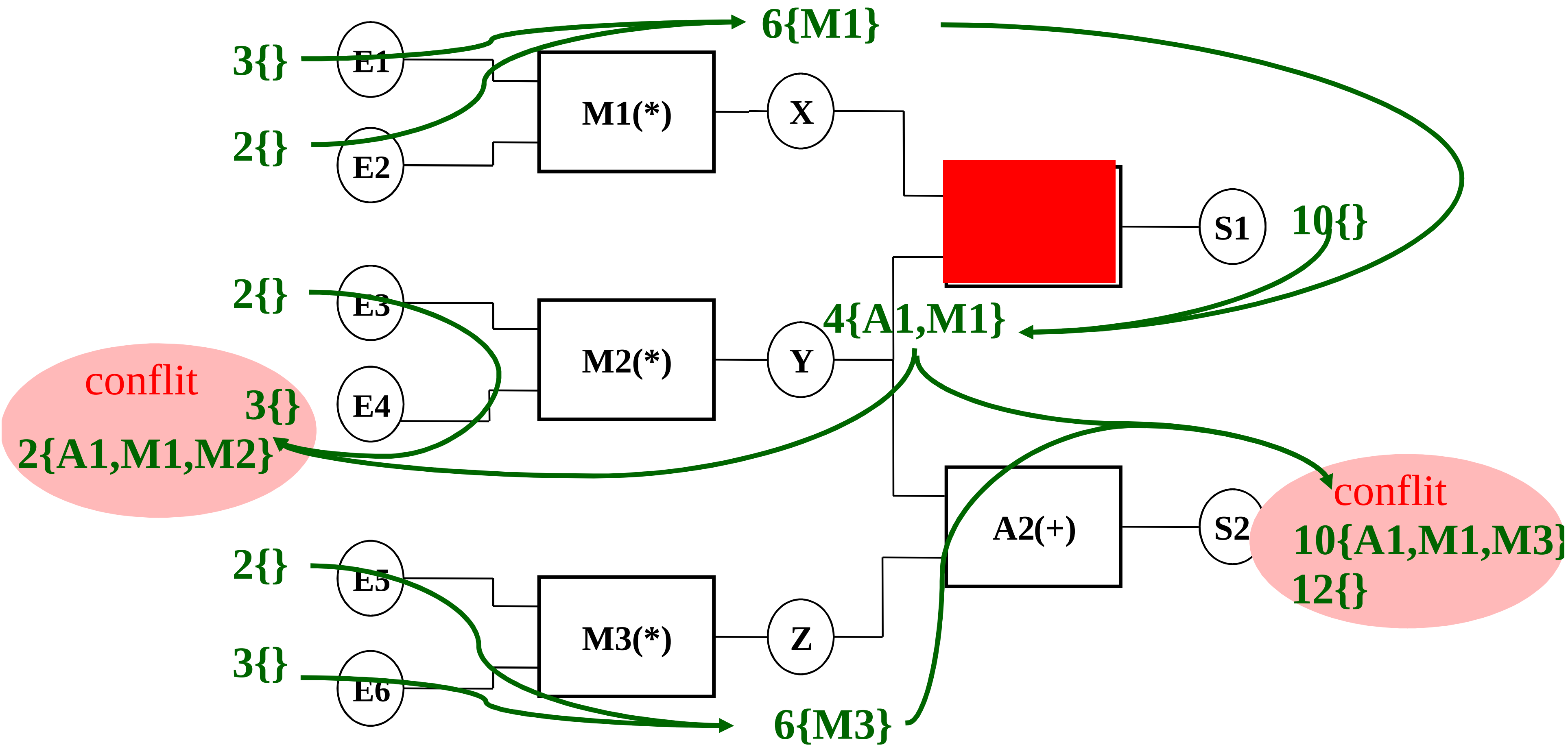


Propagation



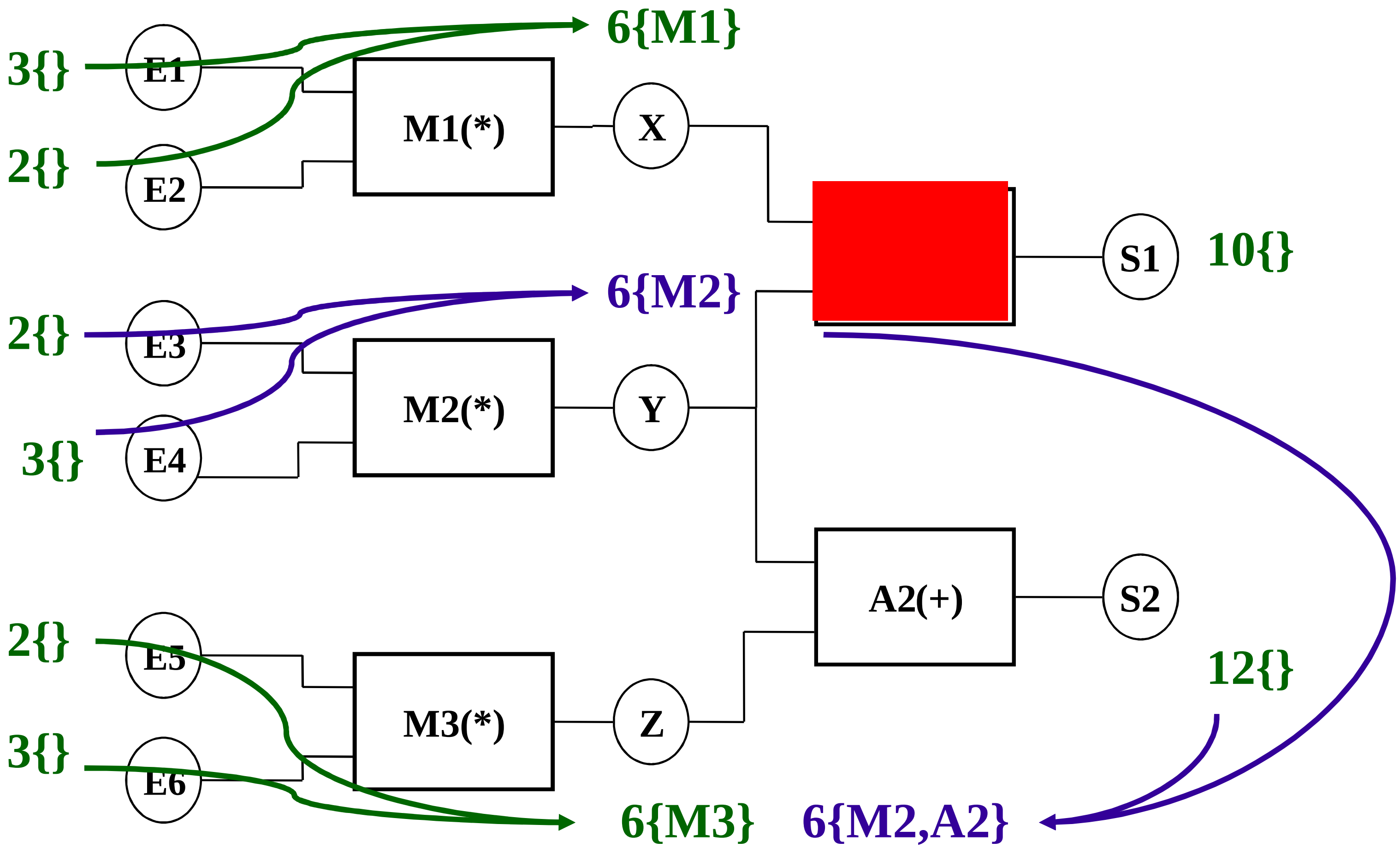
Propagation

Suspension de A1



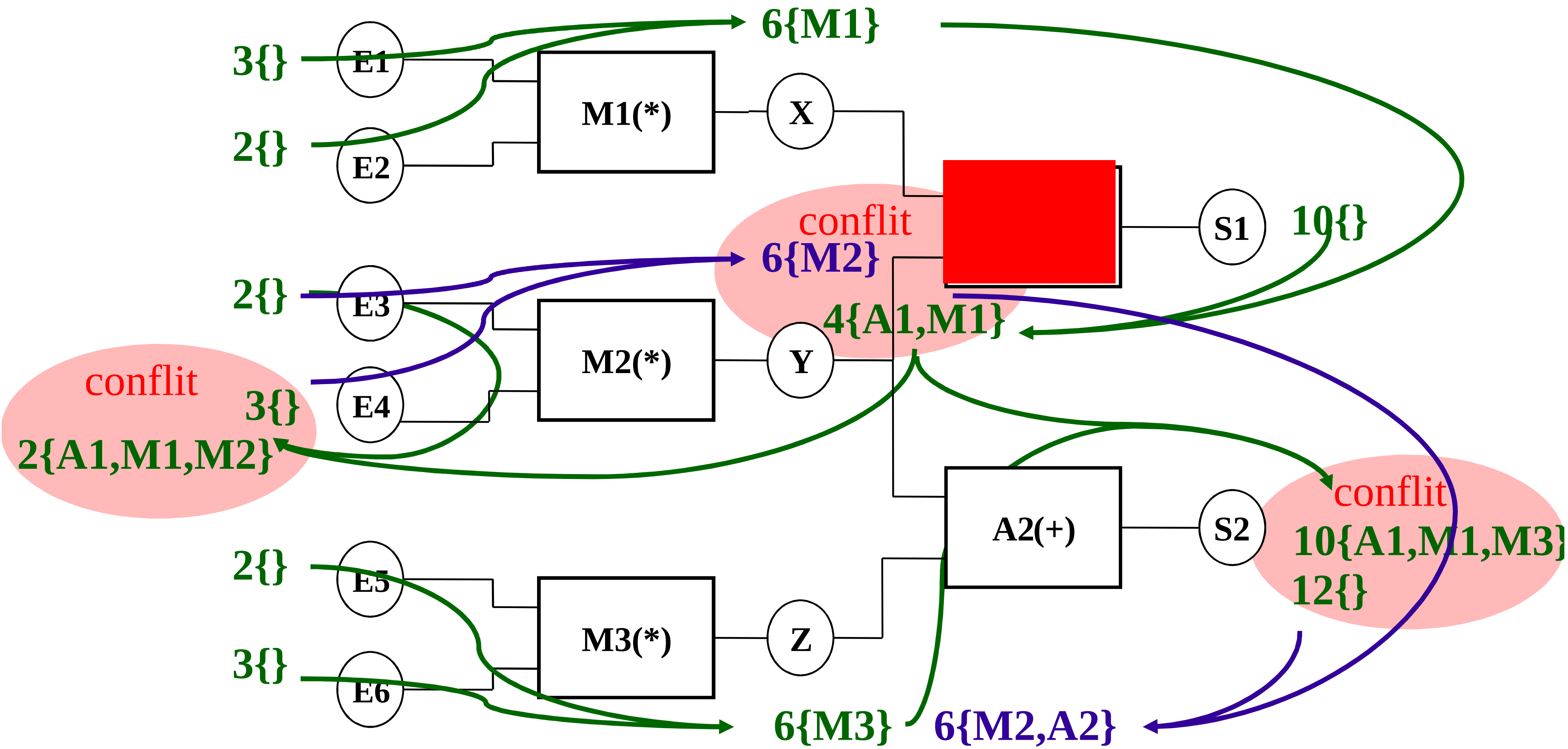
Propagation

Suspension de A1 et nouvelles propagations



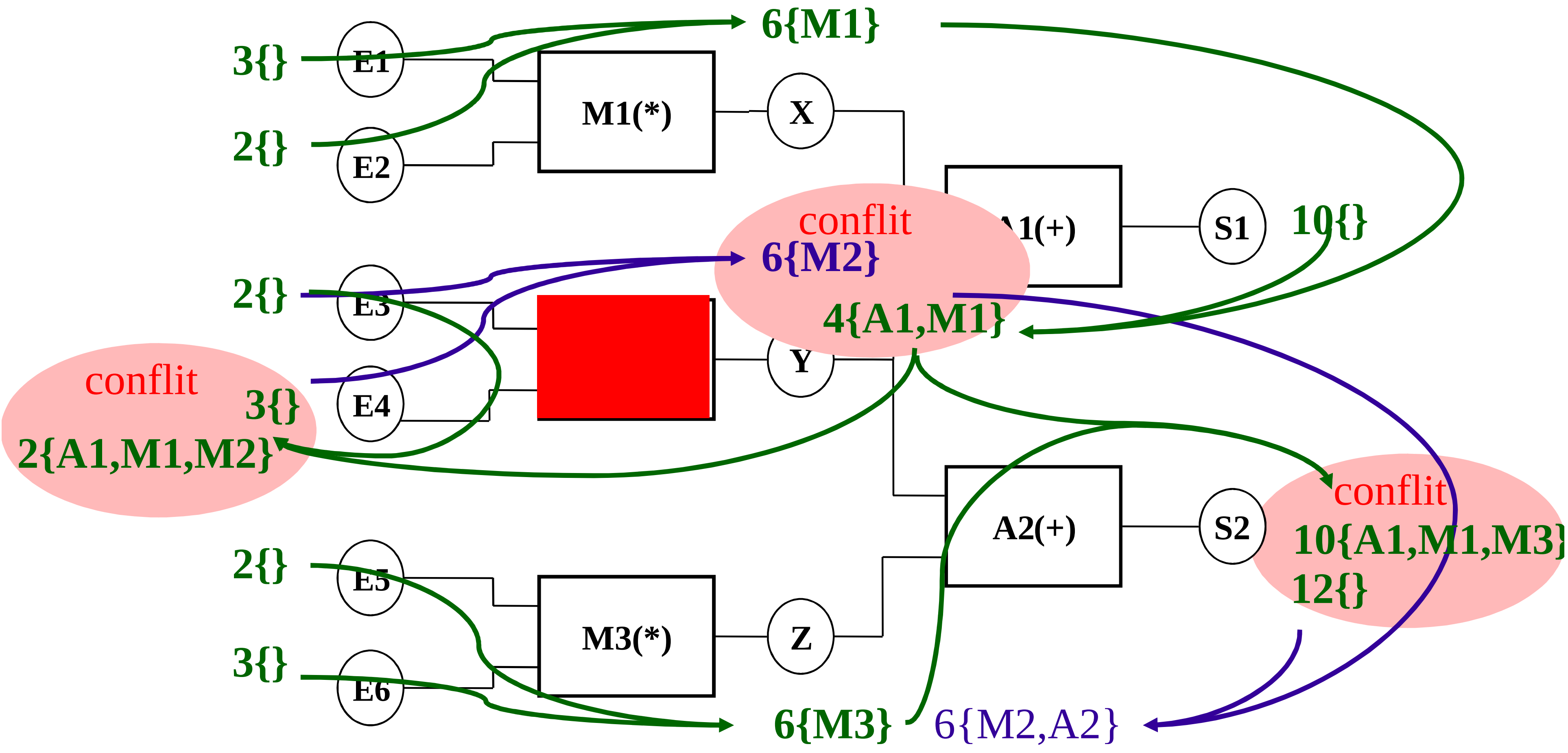
Propagation

Après la suspension de A1 et les nouvelles propagations, on restaure le réseau



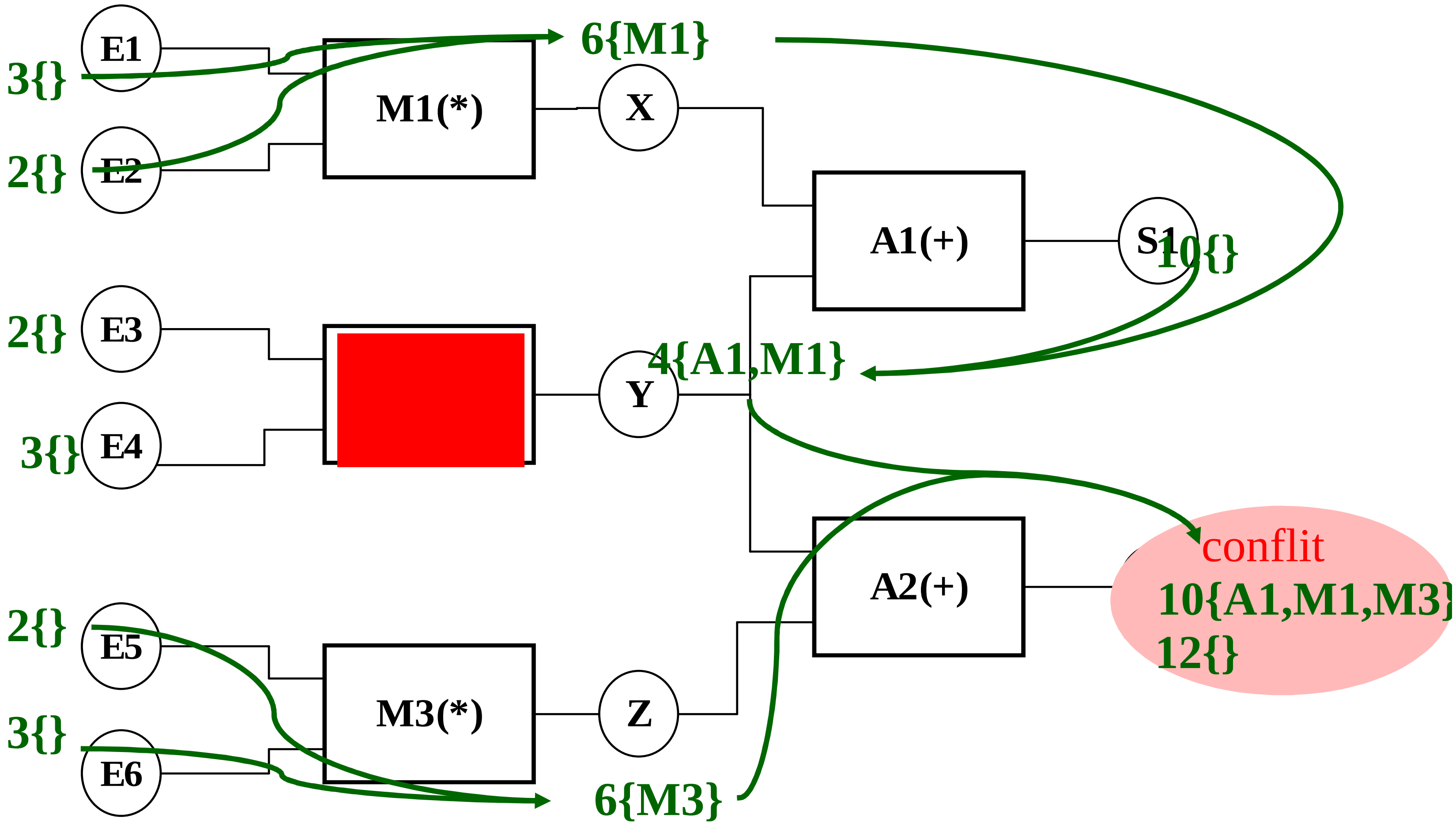
Propagation

Suspension de M2



Propagation

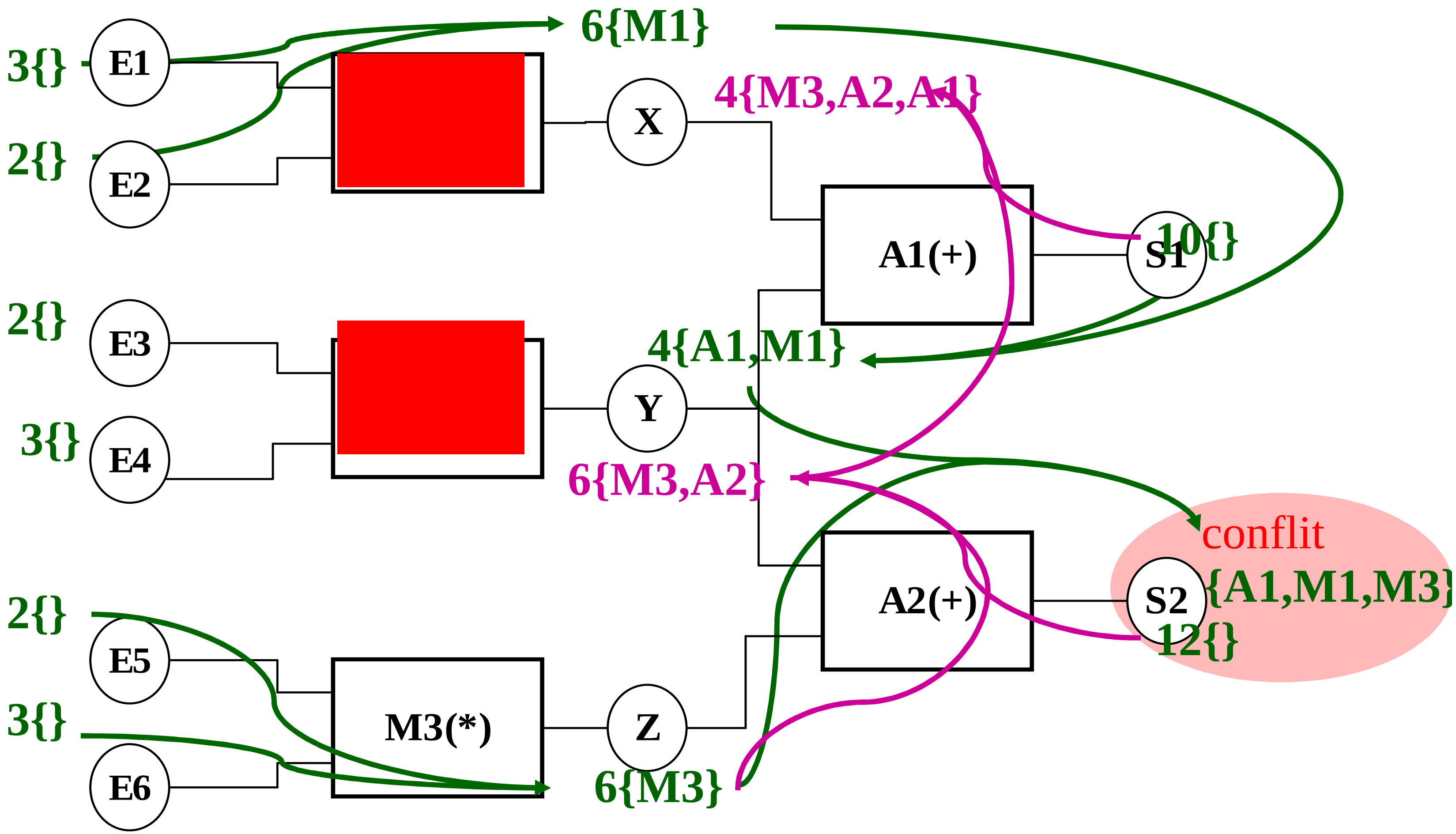
Suspension de M2



On sait que {A1} restaure la consistence: inutile de suspendre A1 ici
On va donc essayer M1 ou M3

Propagation

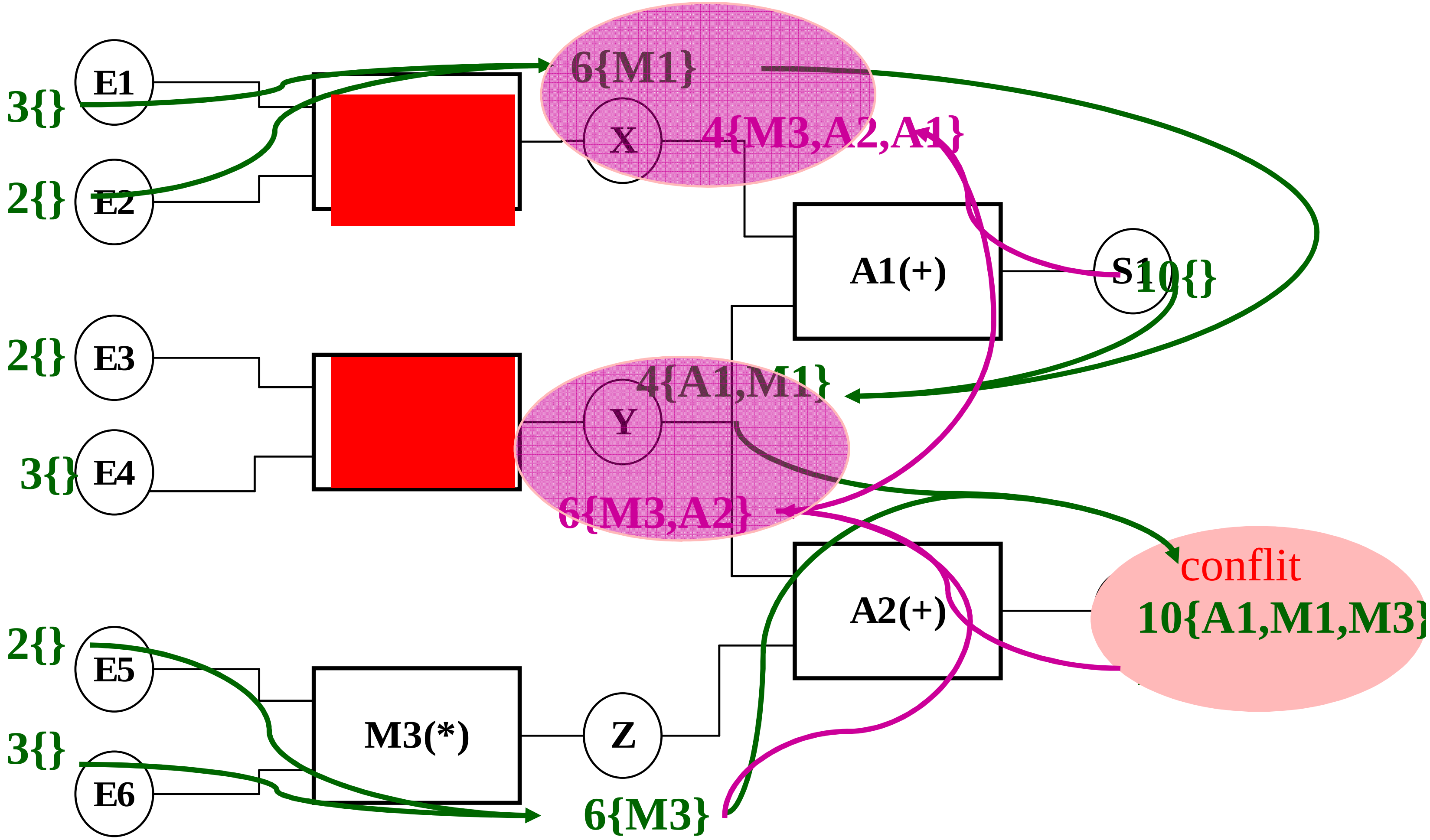
Suspension de M2 et M1



La consistence est rétablie => {M2,M1} est une explication (minimale ?)

Propagation

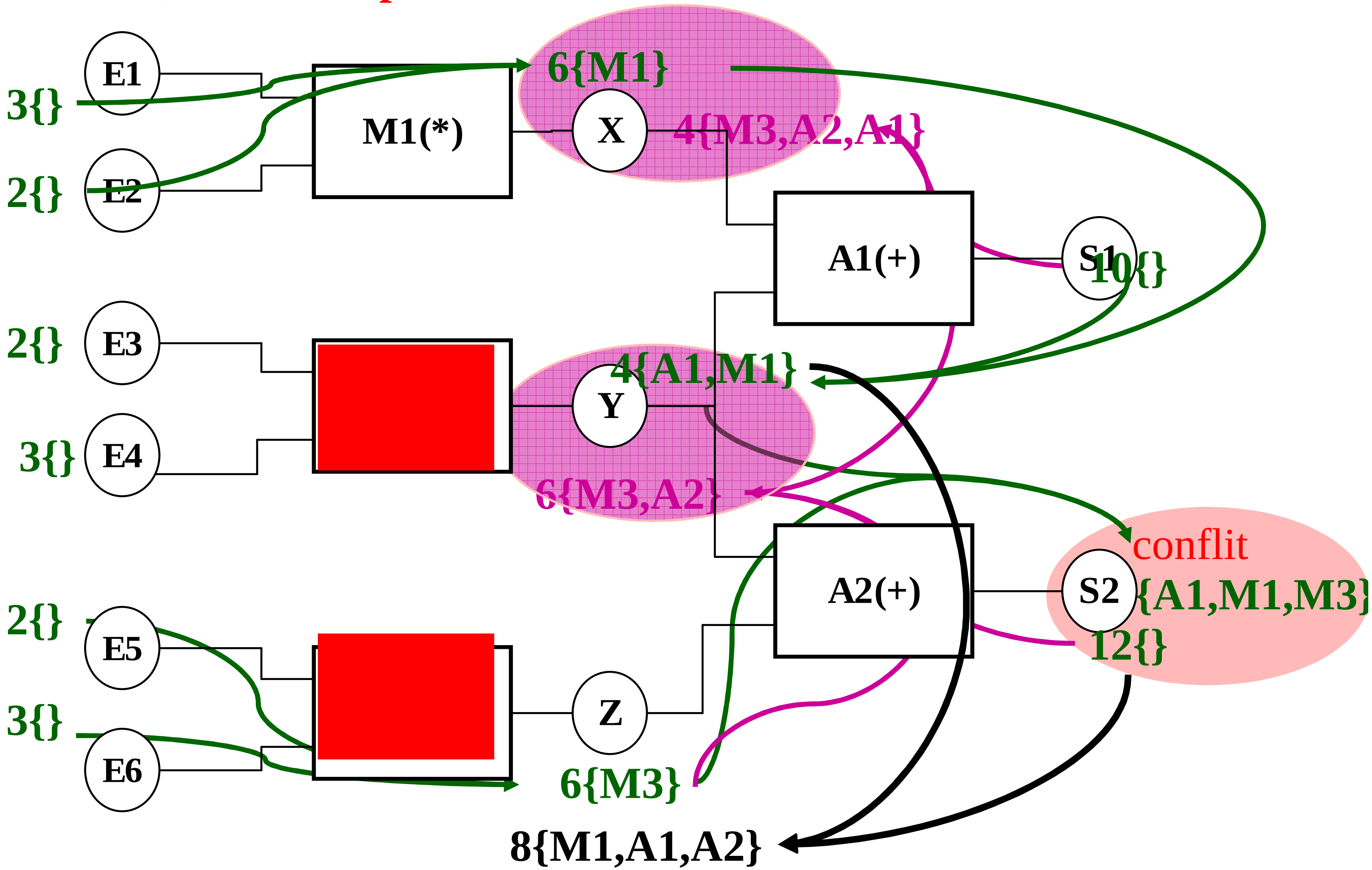
Suspension de M2, M1 est restauré



La consistence est rétablie => {M2,M1} est une explication (minimale ?)

Propagation

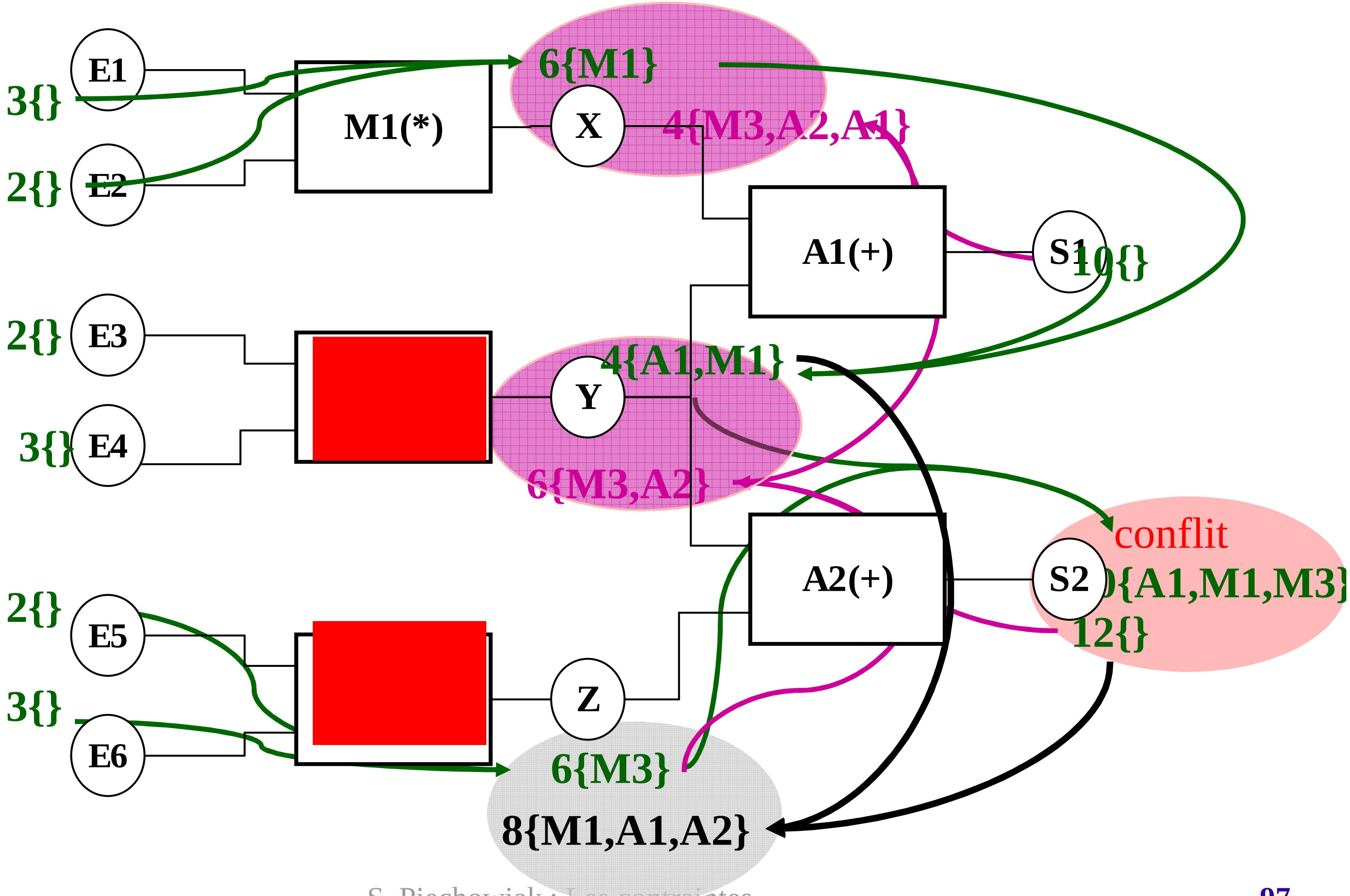
Suspension de M2, M3 est suspendu



La consistence est rétablie => {M2,M3} est une explication (minimale ?)

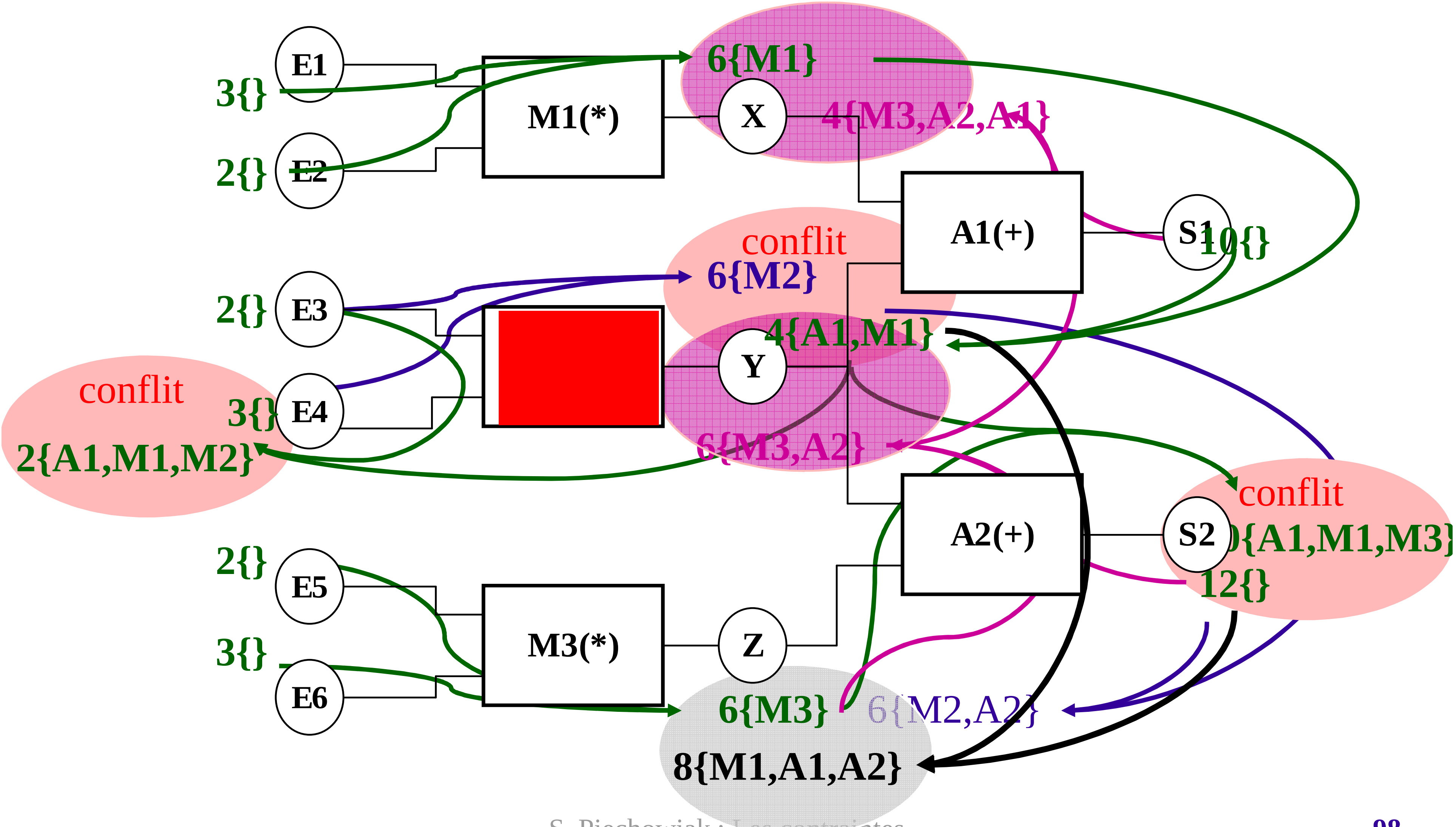
Propagation

Suspension de M2, M3 est restauré



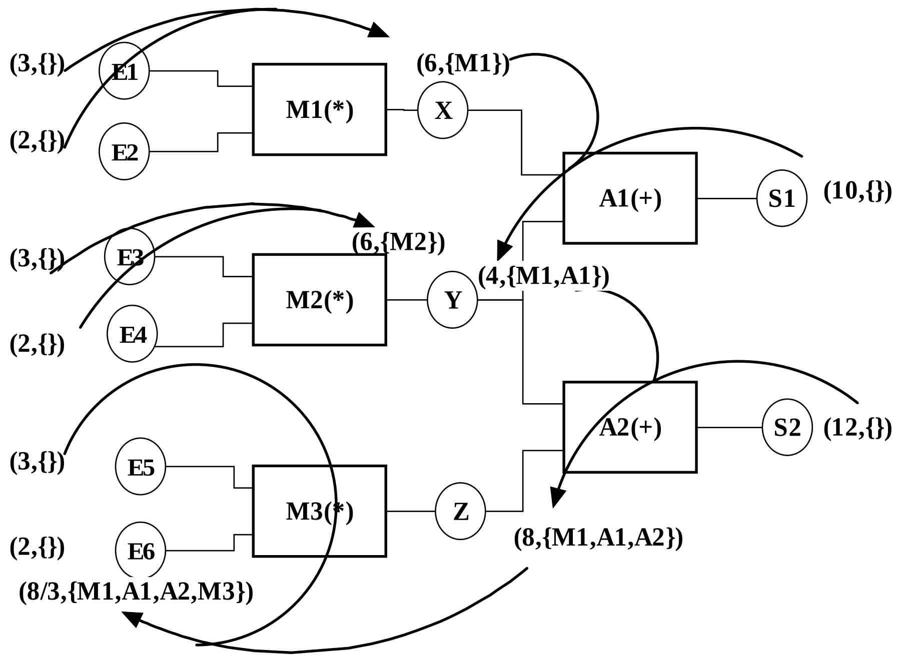
Propagation

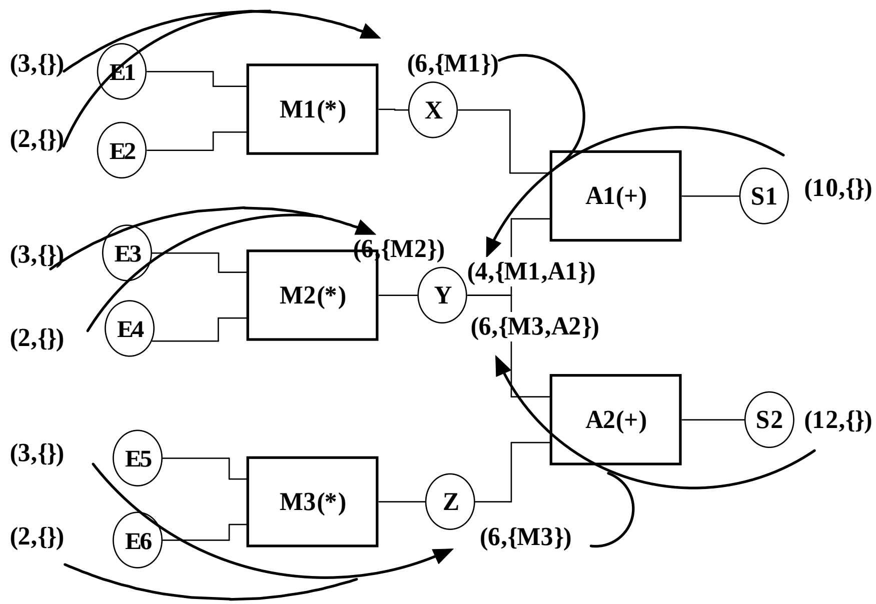
M2 est restauré



CSP : extensions

Minimiser les justifications

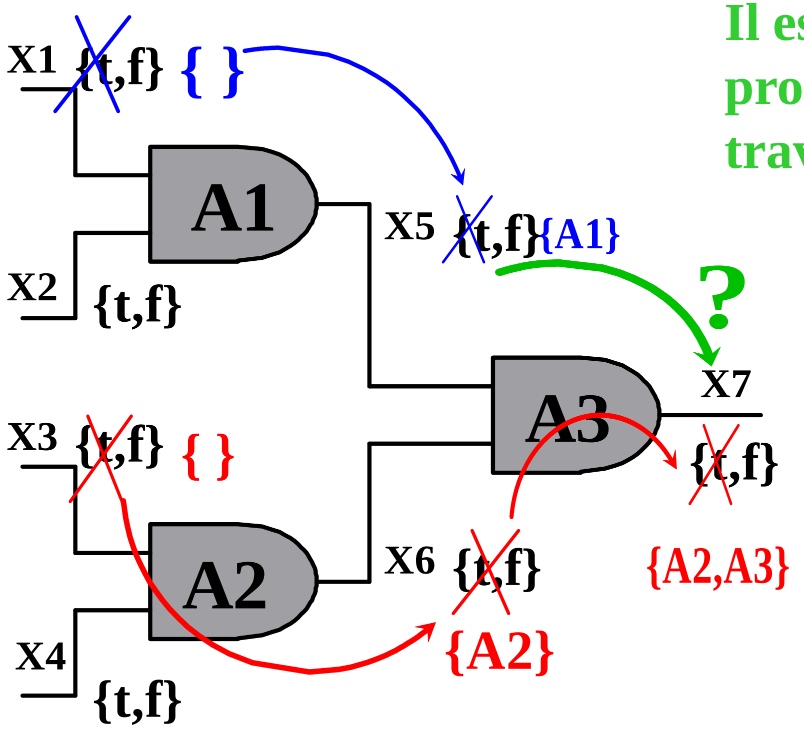




Les CSP dynamiques

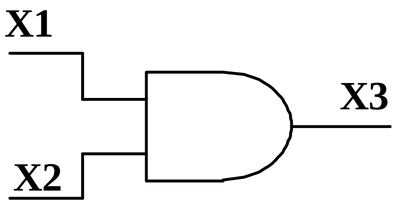
X1 est affectée de la valeur f

X3 est affectée de la valeur f



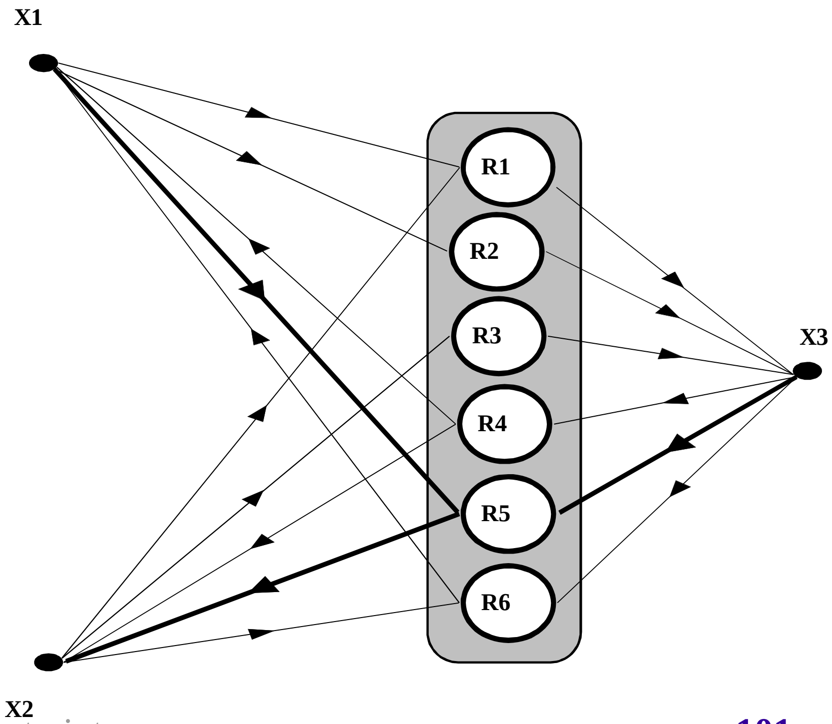
Il est inutile de propager la valeur à travers A3

Representation par Règles

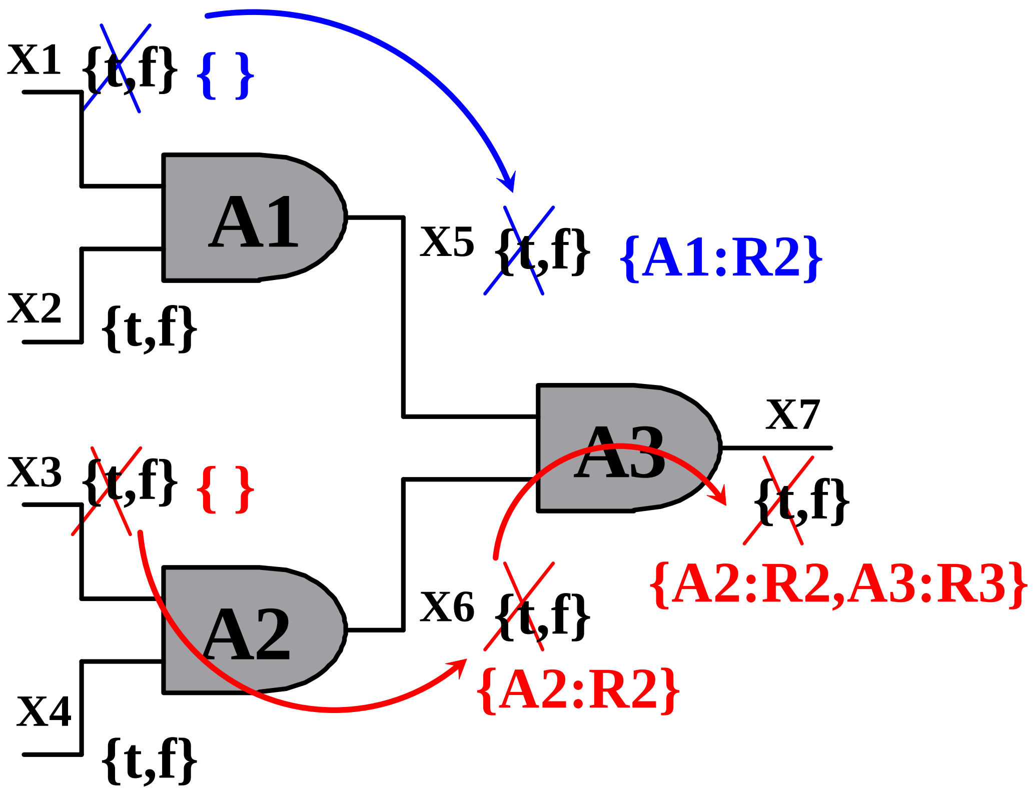


line	X1	X2	X3
1	t	t	t
2	f	t	f
3	t	f	f
4	f	f	f

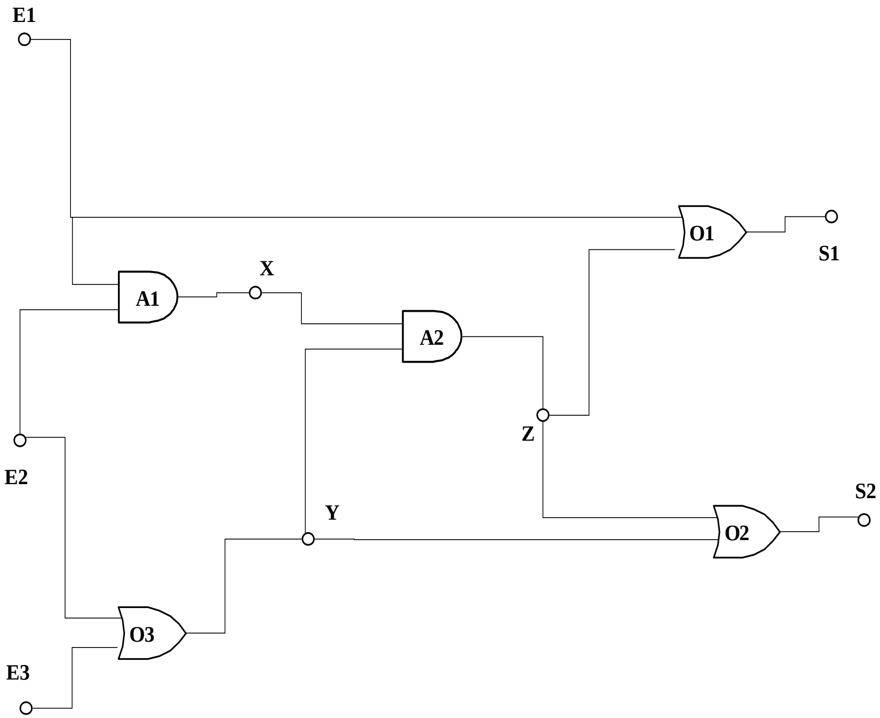
R1 : IF (X_1 = true) and (X_2 = true) THEN X_3 := true
R2 : IF (X_1 = false) THEN X_3 := false
R3 : IF (X_2 = false) THEN X_3 := false
R4 : IF (X_3 = true) THEN (X_1 :=true; X_2 := true)
R5 : IF (X_1 = true) and (X_3 = false) THEN X_2 := false
R6 : IF (X_2 = true) and (X_3 = false) THEN X_1 := false

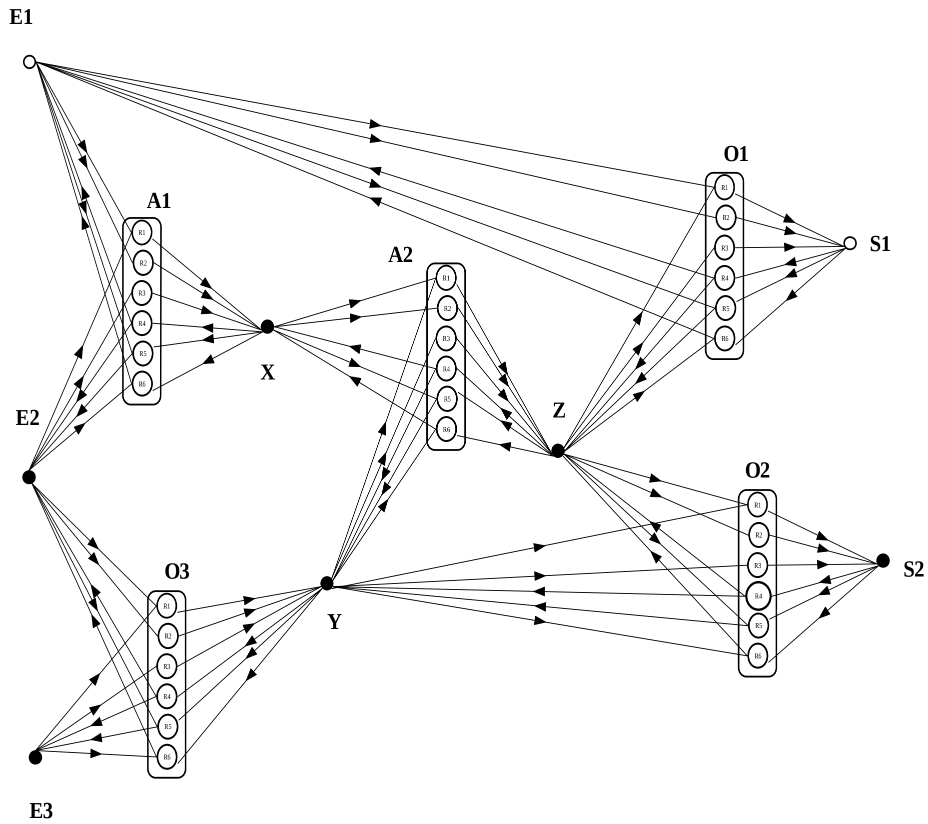


Mémorisation de la trace de propagation



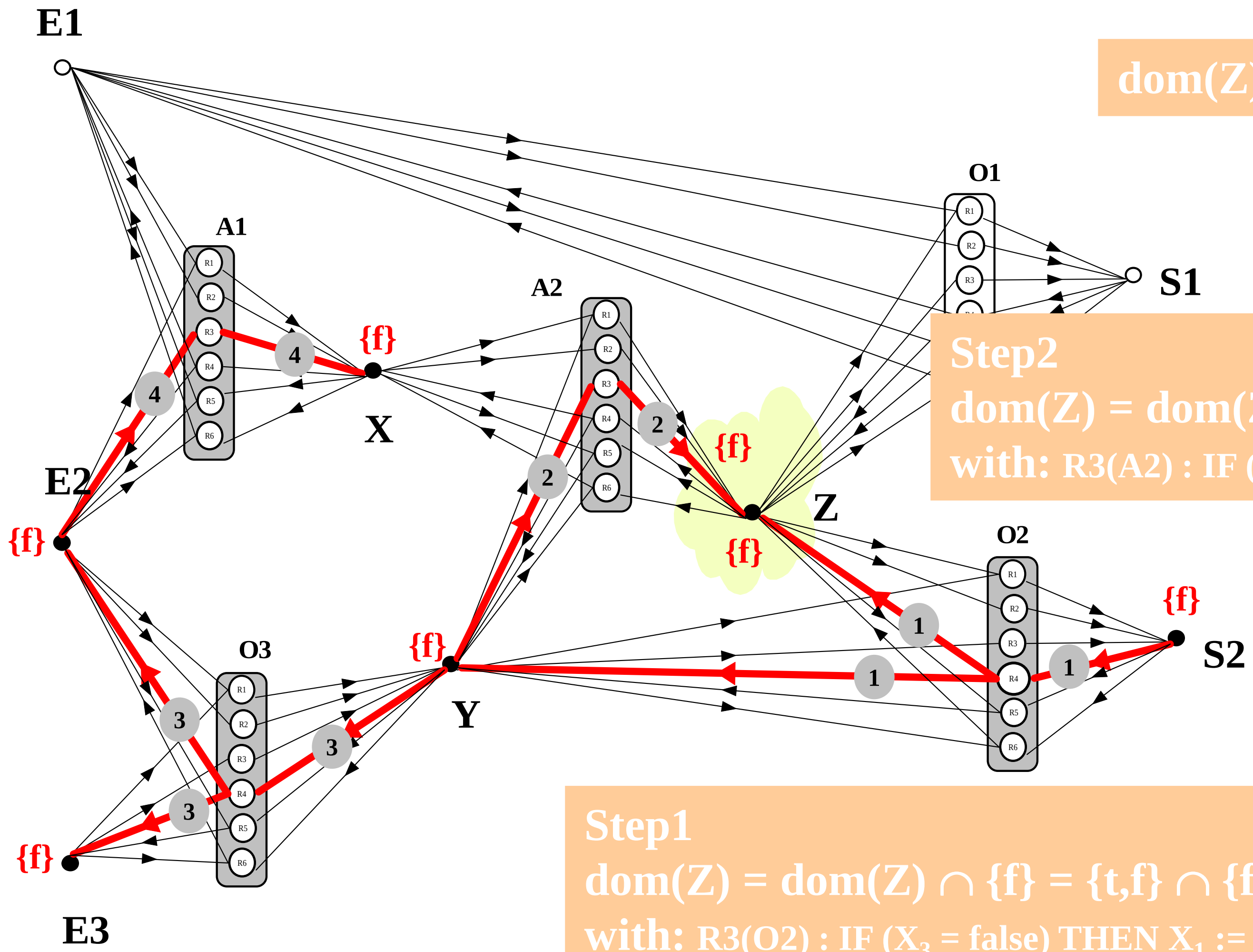
Le réseau de règles de propagation





Le réseau de règles de propagation

dom(Z) = {t,f} (initial domain)

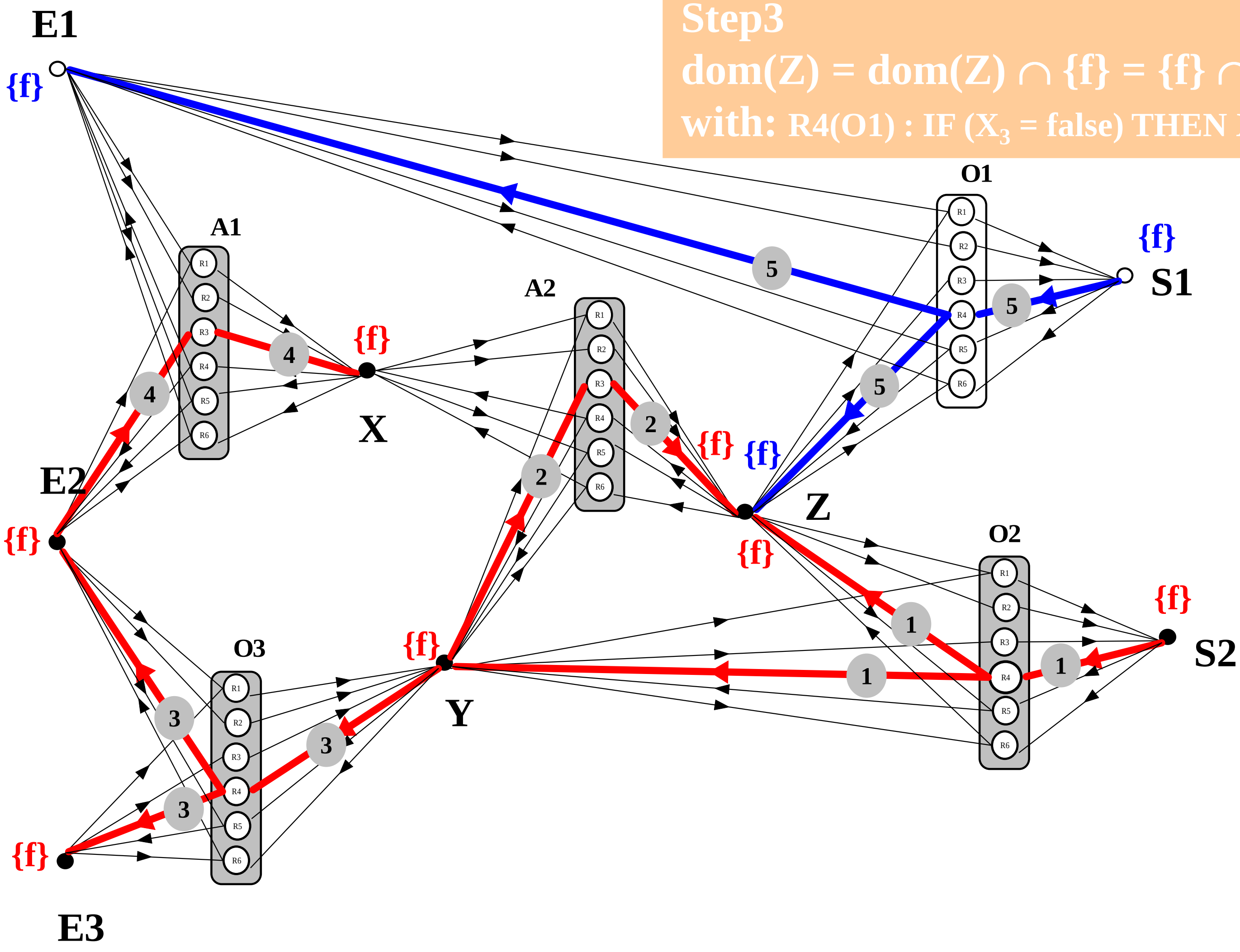


Step2
dom(Z) = dom(Z) $\cap$ {f} = {f} $\cap$ {f} = {f}
with: R3(A2) : IF (X_2 = false) THEN X_3 := false

dom(S2) := {f}

Step1
dom(Z) = dom(Z) $\cap$ {f} = {t,f} $\cap$ {f} = {f}
with: R3(O2) : IF (X_3 = false) THEN X_1 := false; X_2 := false

Le réseau de règles de propagation



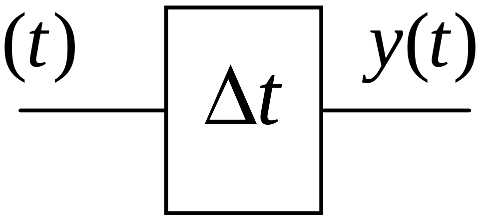
Step3

$dom(Z) = dom(Z) \cap \{f\} = \{f\} \cap \{f\} = \{f\}$

with: R4(O1) : IF (X_3 = false) THEN X_1 := false ; X_2 := false

Réseau de contraintes et prise en compte du temps (séquencement)

$x(t)$



$y(t) = x(t - \Delta t)$

R1: IF x(t-Δt) = v THEN y(t) := v
R2: IF y(t) = v THEN x(t-Δt) := v

pb: il y a des relations

- **entre les valeurs des variables**
- **Entre les dépendances temporelles de ces valeurs**

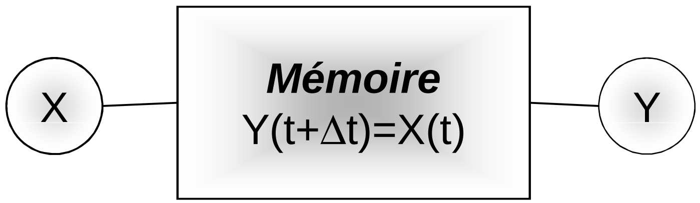
$y(t) = f(x(\varphi(t)))$

Réseau de contraintes sur intervalles de valeurs
et prise en compte du temps (également sous forme d'intervalles)

Traitement du temps :

Les dépendances dépendent des valeurs et du temps

Exemple :



$X(t_1)=v1 \longrightarrow Y(t_1+\Delta t)=v1$

Hypothèse : fonctionnement correct à l'instant t1

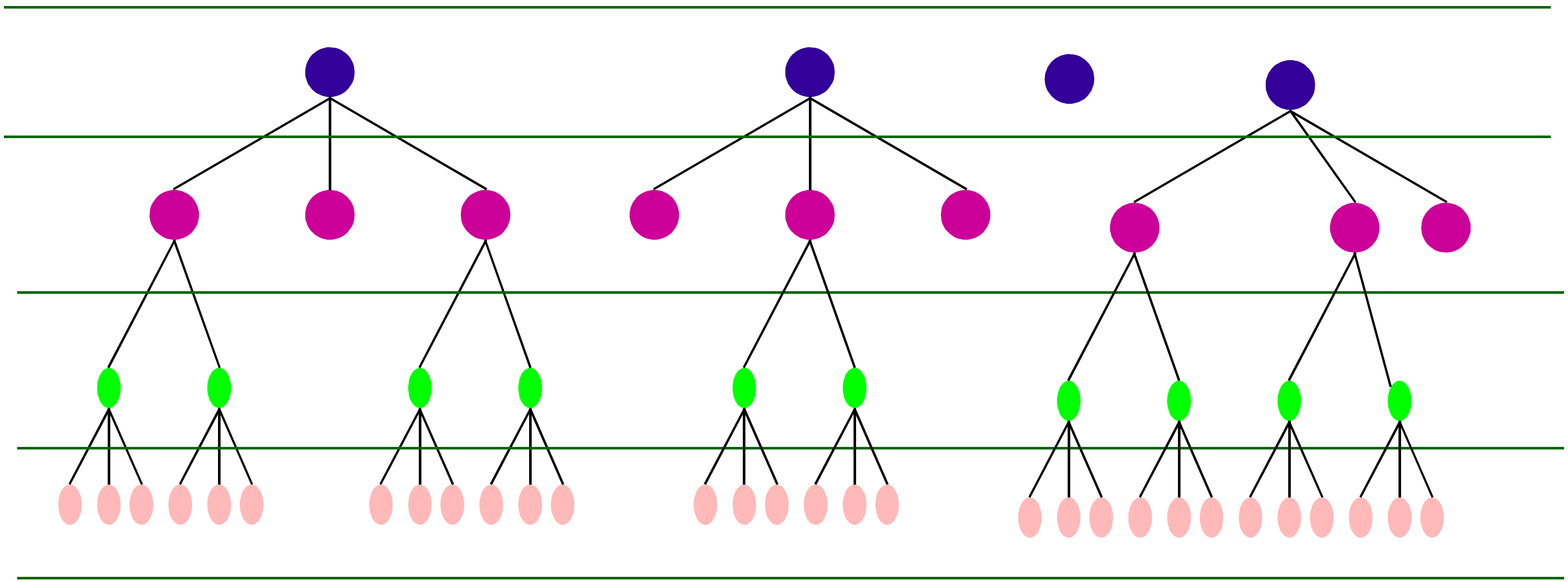
$X([t_1,t_2])=[v1,v2] \longrightarrow Y([t_1,t_2]+\Delta t)=[v1,v2]$

Hypothèse : fonctionnement correct sur l'intervalle de temps $[t_1,t_2]$

Utilisation d'heuristiques

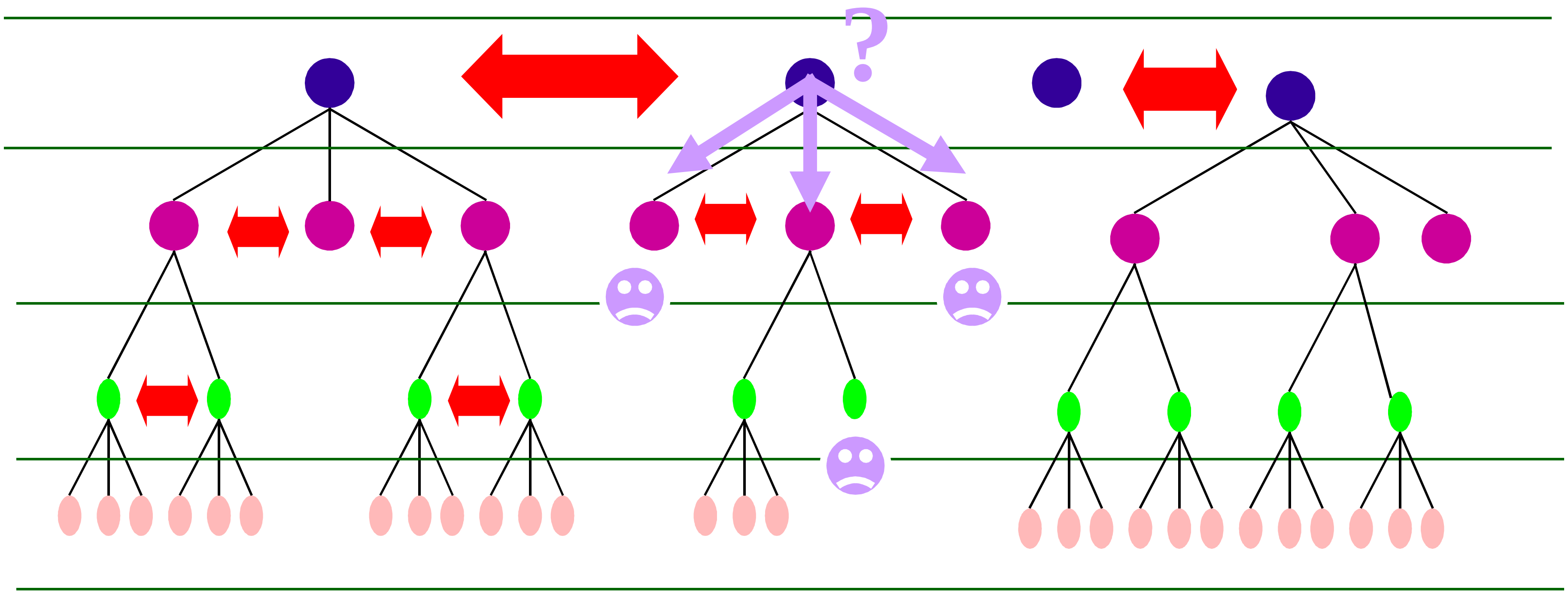
Problématique

La résolution d'un CSP s'apparente à un parcours arborescent d'un espace de recherche.



Quels sont les paramètres sur lesquels on peut agir ?

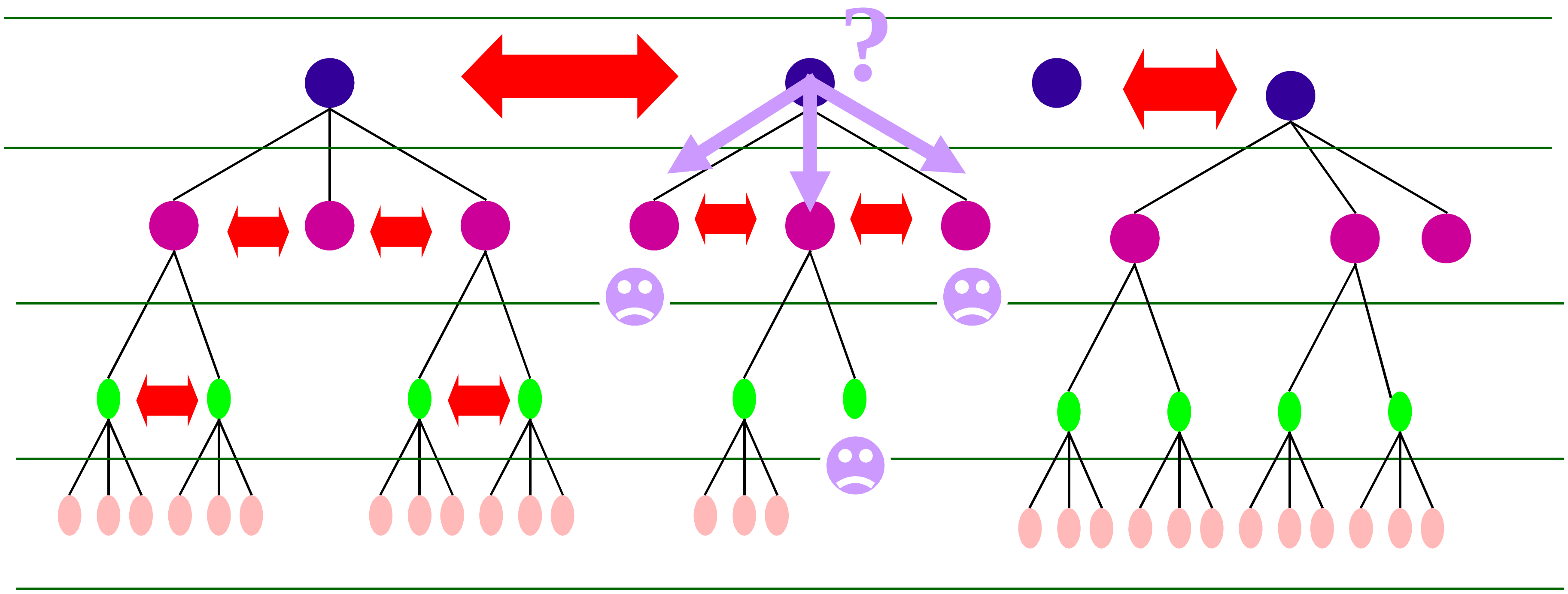
Utilisation d'heuristiques



L'ordre des valeurs dans le domaine de chaque variable

L'ordre des valeurs ne fait que changer l'ordre d'exploration des branches de l'arbre de recherche. Par conséquent, si l'objectif est de trouver toutes les solutions d'un pb cette heuristique n'a que très peu d'intérêt !

Utilisation d'heuristiques

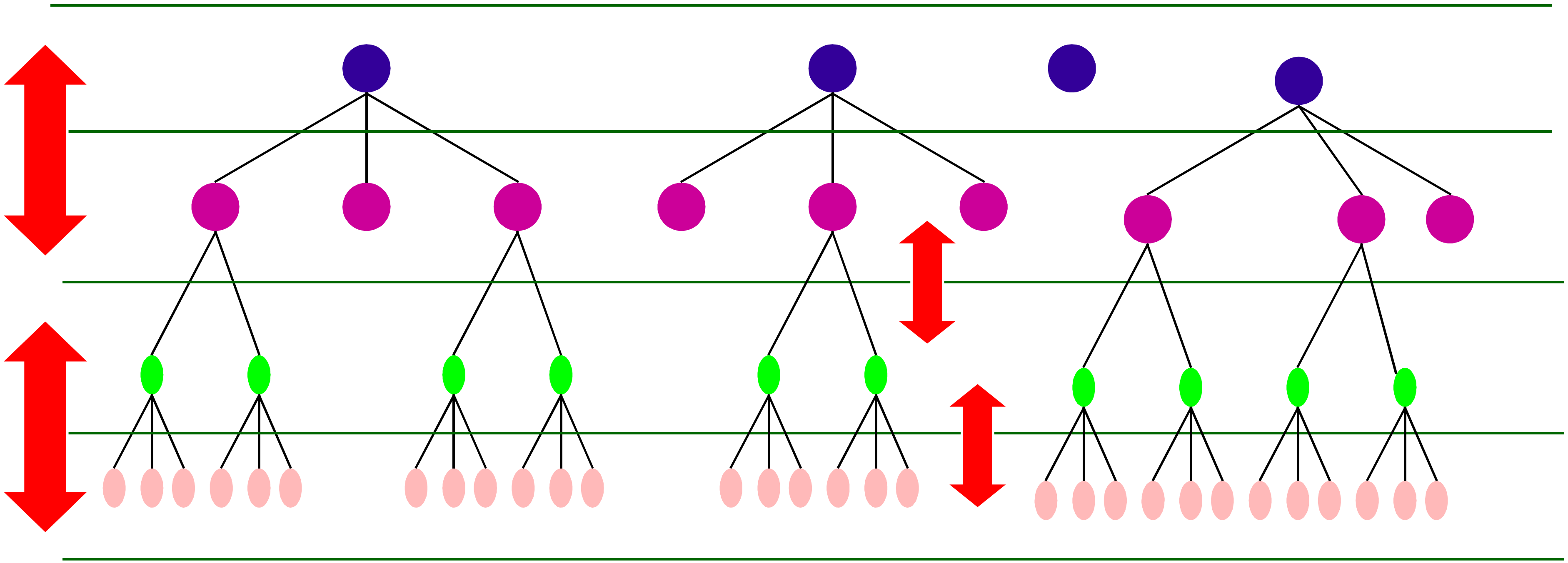


Par exemple en fonction des échecs qui seront atteints

L'heuristique min-conflict favorise les valeurs les plus prometteuses.

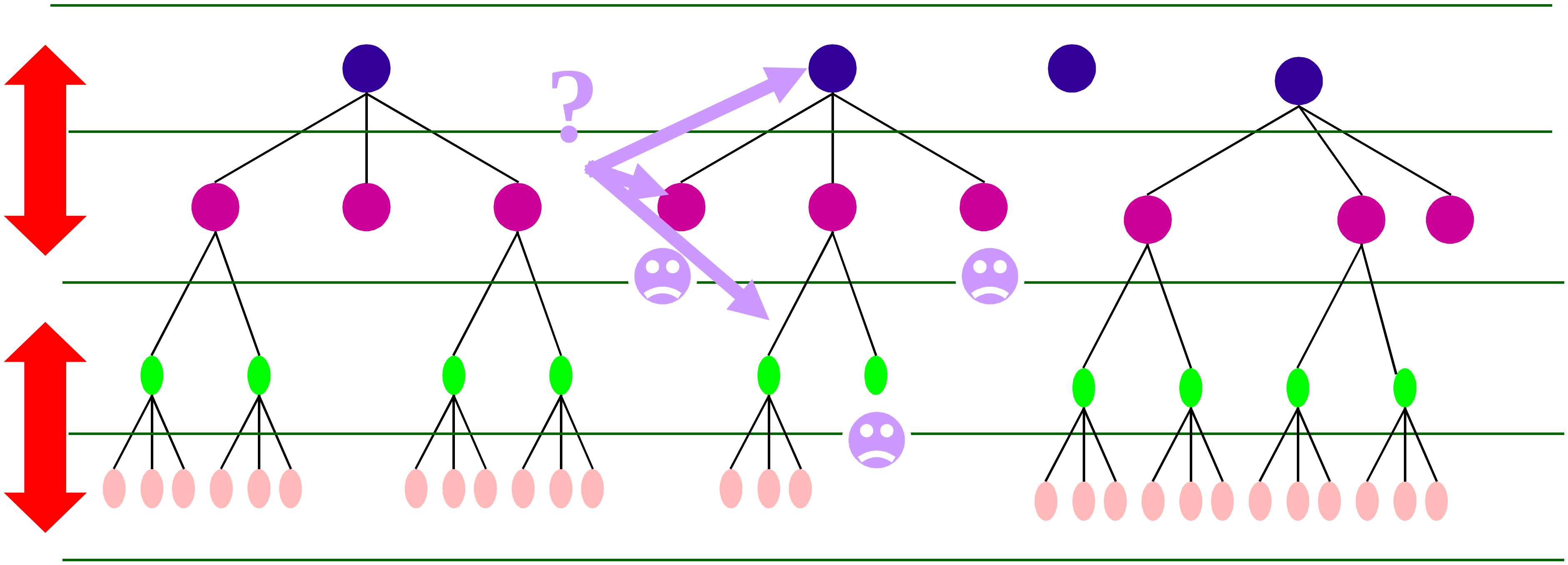
S. Minton at Al.
Minimizing conflitcs: a heuristic repair method for constraint satisfaction and scheduling problems.
Artificial Intelligence 58 pp 161-205, 1992

Utilisation d'heuristiques



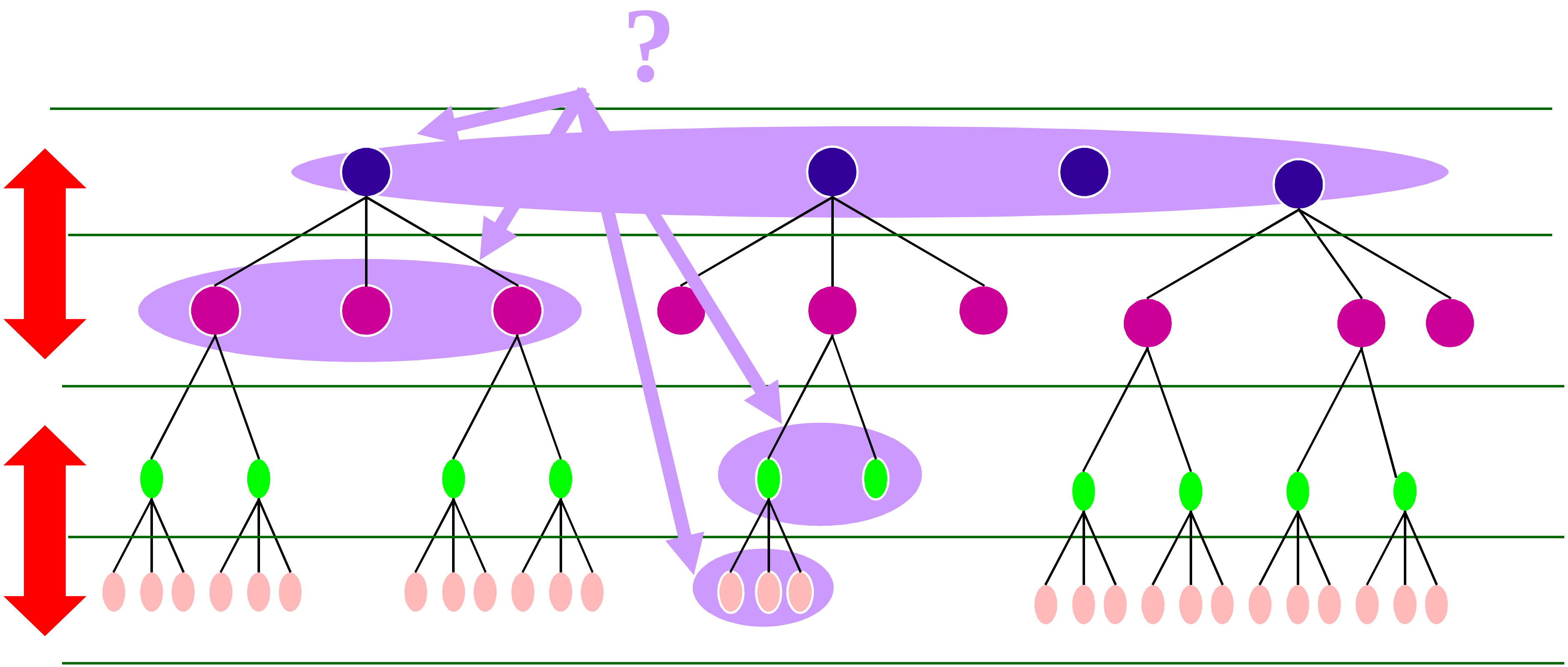
L'ordre des variables

Utilisation d'heuristiques



L'ordre des variables :
en fonction des échecs

Utilisation d’heuristiques



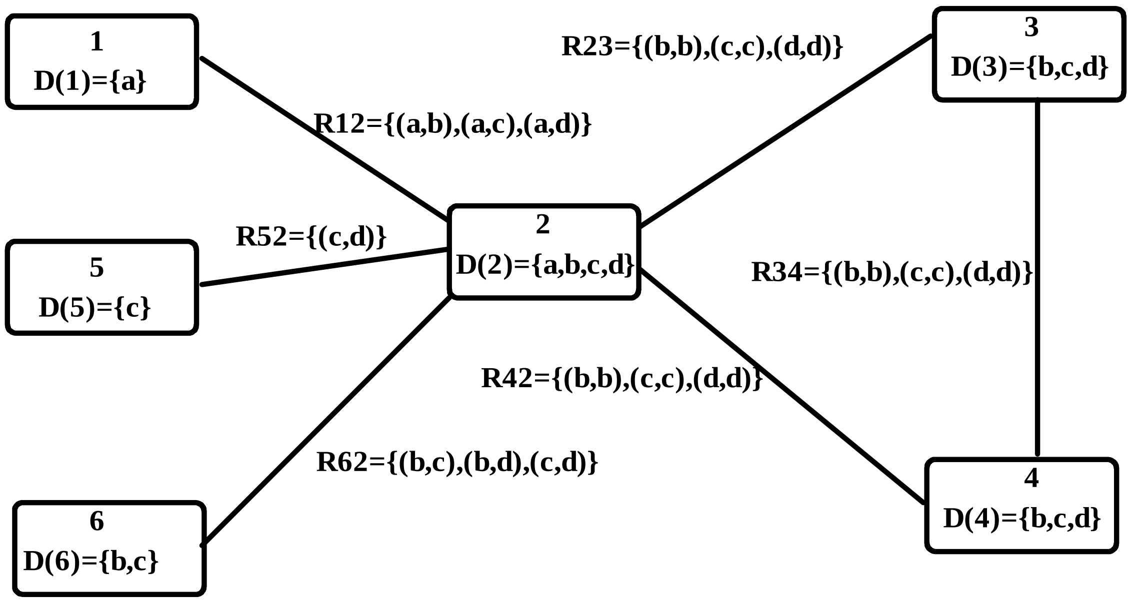
L’ordre des variables :

en fonction de la taille des domaines :

- **grande taille => arbre large**
- **petite taille => arbre haut**

“First-fail principle” = privilégier les variables qui ont le plus de chance de conduire à un échec rapidement (plus vite on arrive à un échec moins il y aura de backtracking)

Utilisation d'heuristiques



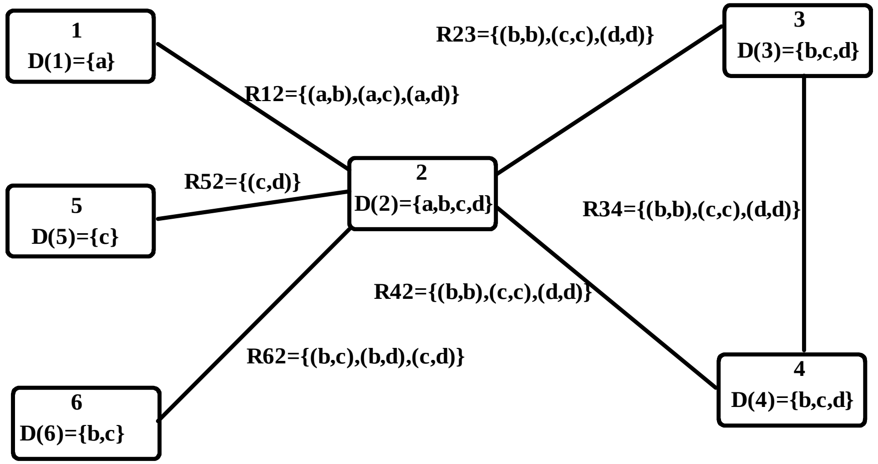
On associe une pondération *critère* à chaque variable

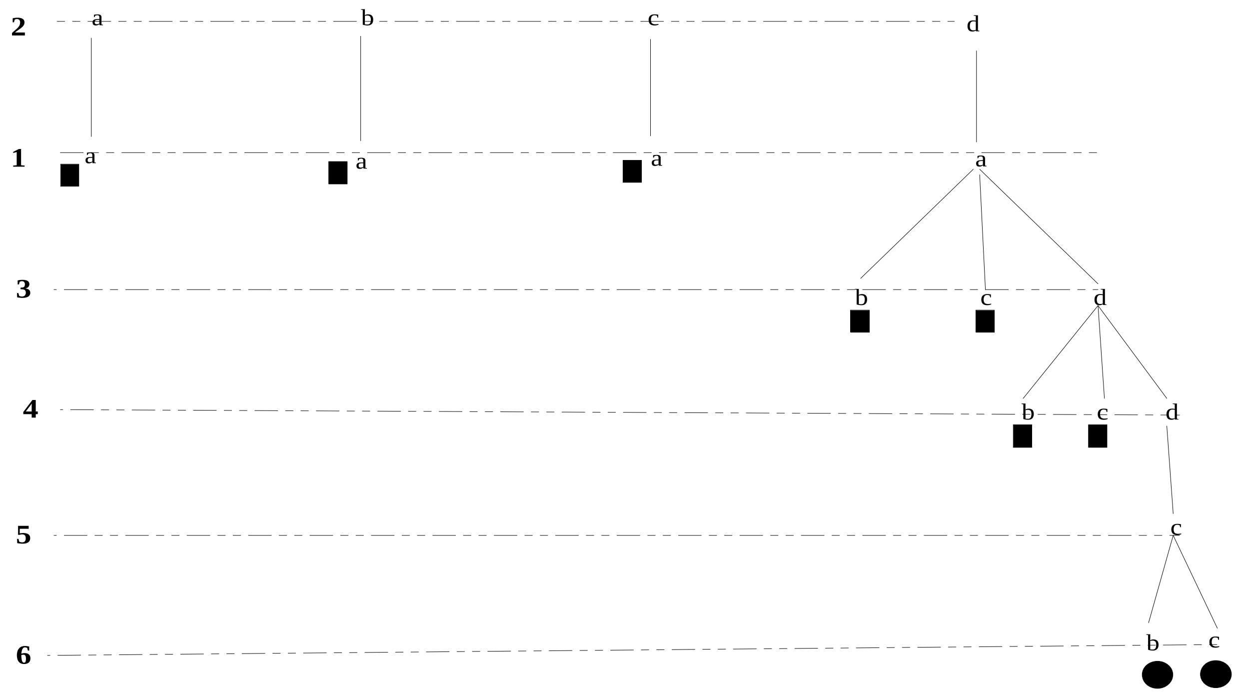
Fonction 1 $$\text{critère(xi)} = \frac{|\text{domaine(xi)}|}{\text{nombre de contraintes contenant xi}}$$

De manière à ordonner l'ensemble des variables :

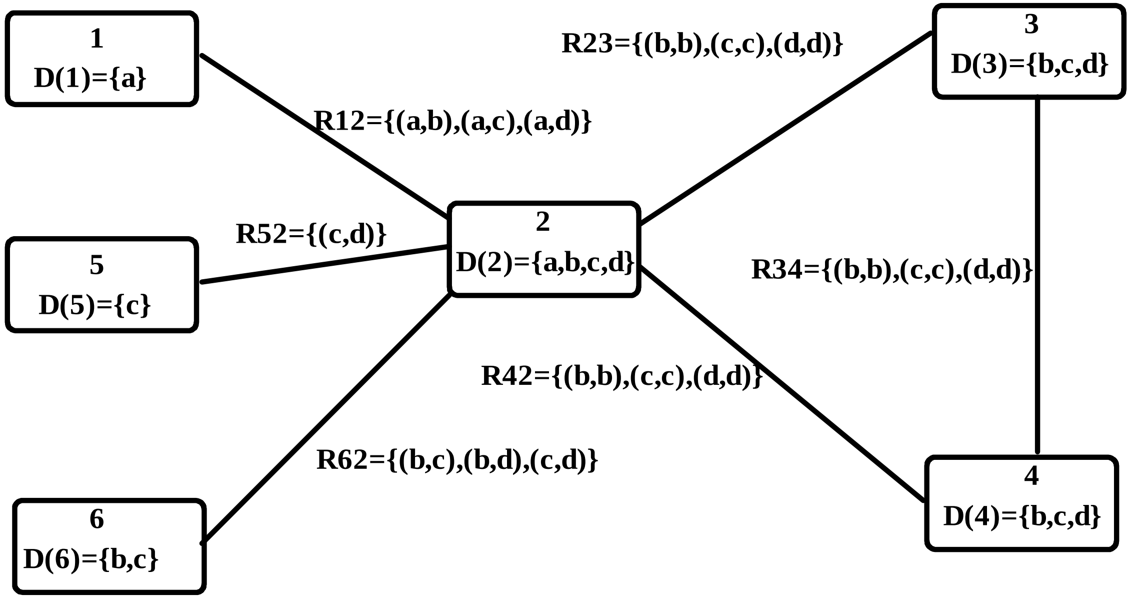
$$xi <_{\alpha} xj \Rightarrow \text{critère}(xi) < \text{critère}(xj)$$

Utilisation d'heuristiques





Utilisation d'heuristiques



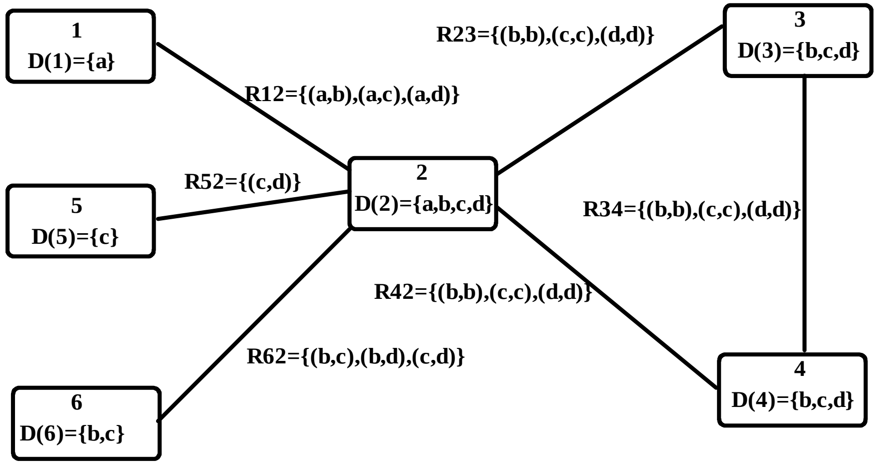
On associe une pondération *poids* à chaque variable

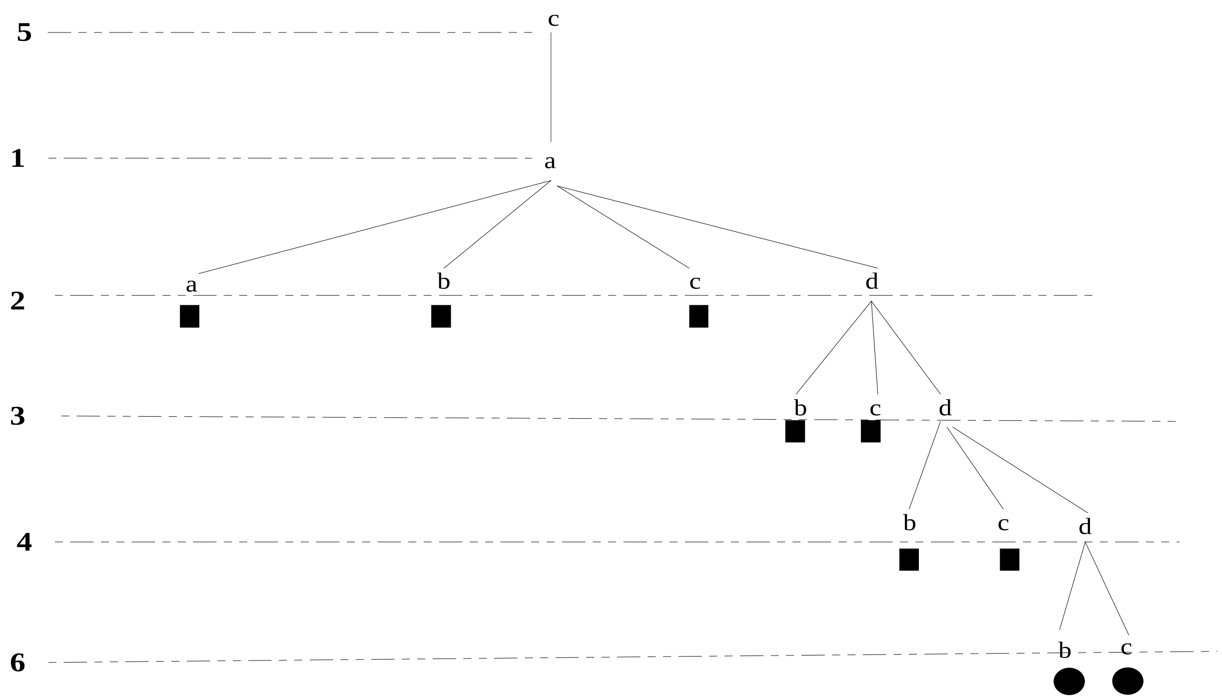
$$\text{poids}(C(xi_1,\ldots,xi_q)) = \frac{|\text{domaine}(xi_1)| \times \ldots \times |\text{domaine}(xi_q)|}{|C|}$$

Fonction 2

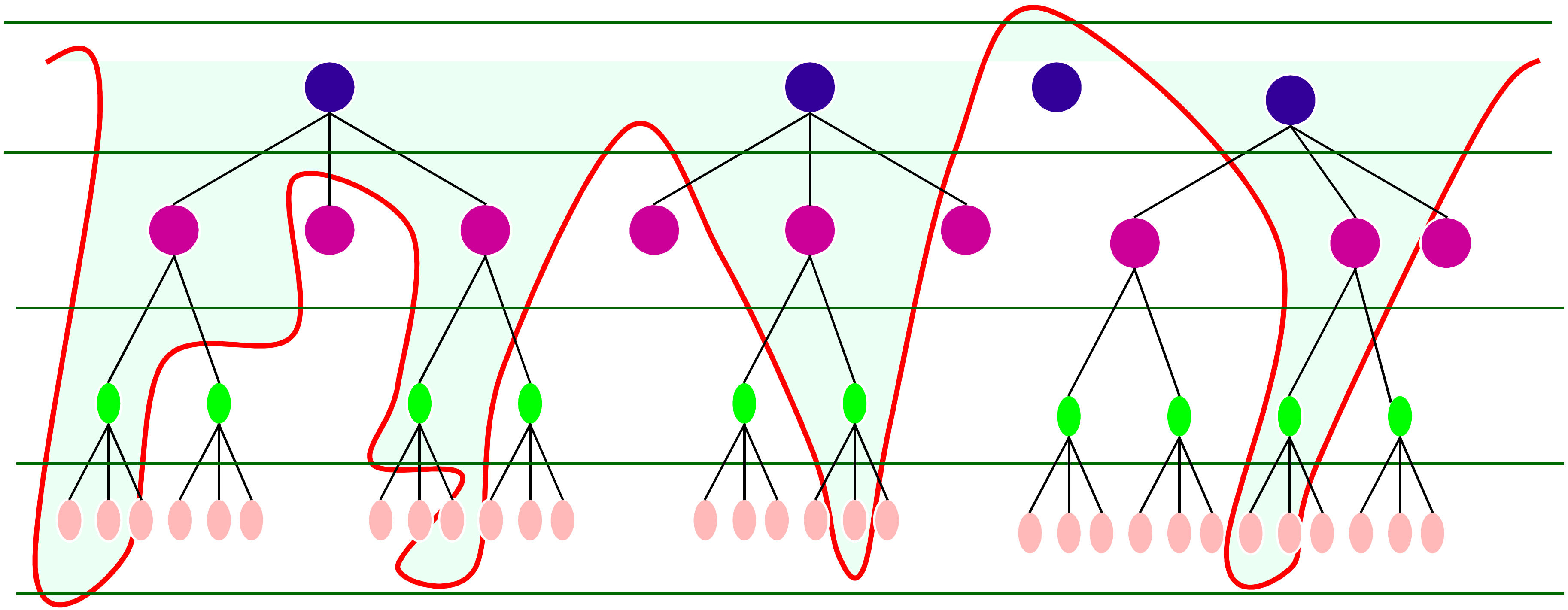
$$\text{critère}(xi) = \frac{|\text{domaine}(xi)|^2}{\sum\limits_{\substack{\text{toutes les contraintes} \\ \text{qui contraingnent xi}}} \text{poids}(C)}$$

Utilisation d'heuristiques





Utilisation d'heuristiques

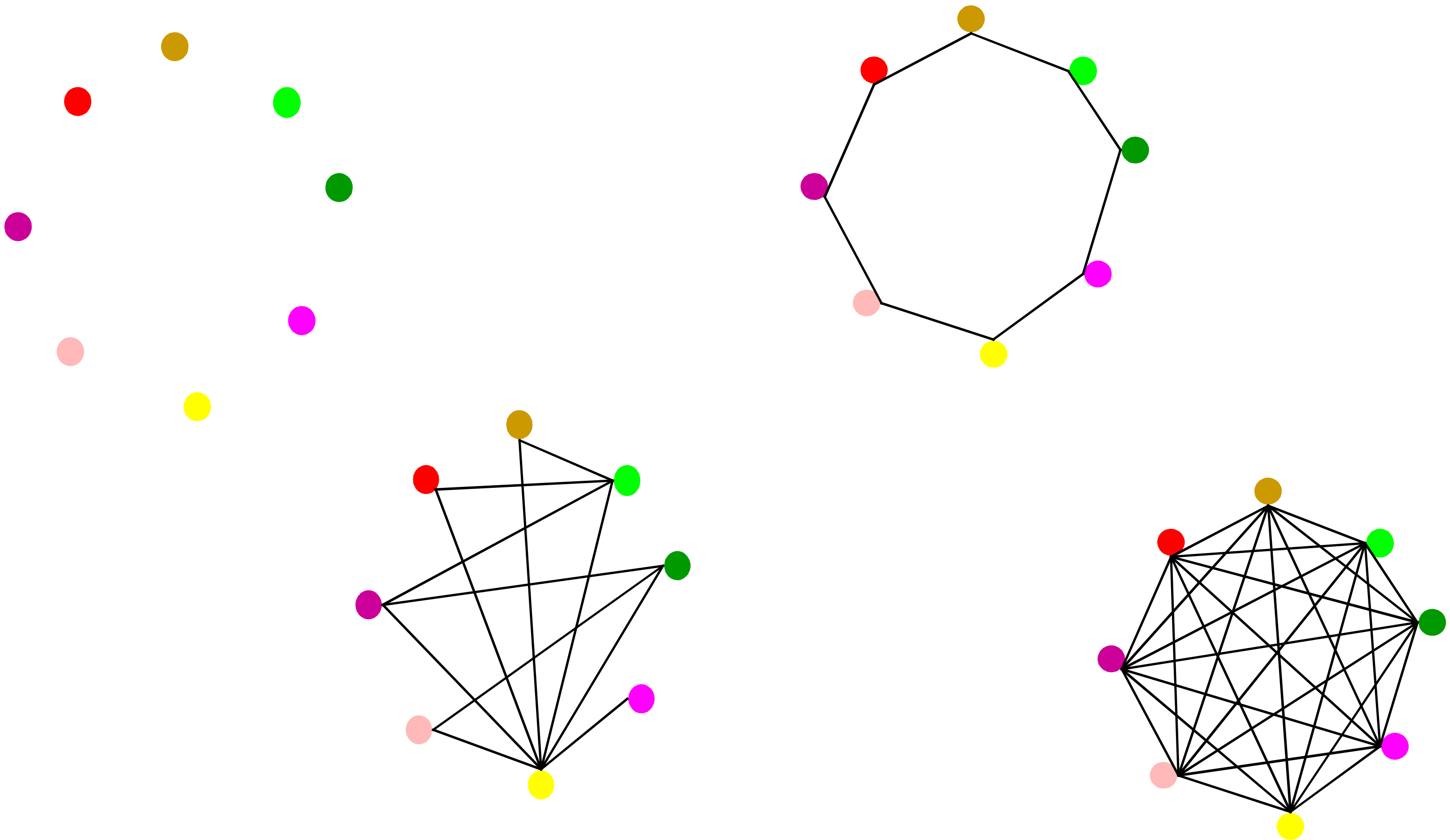


Trouver un moyen de limiter au mieux l'espace de recherche :

- ***a priori*** **(cf filtrage)**
- **pendant la recherche (apprentissage)**

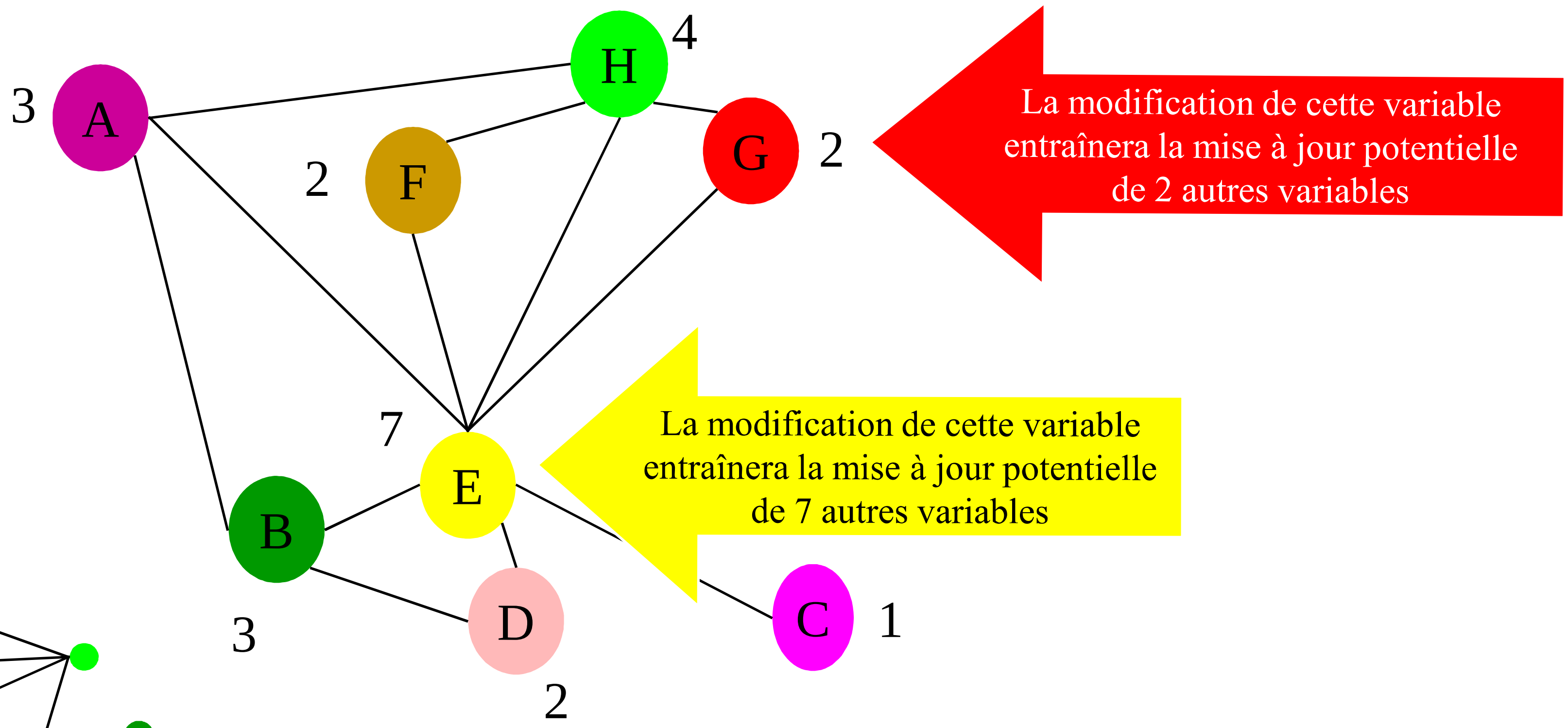
Utilisation d'heuristiques

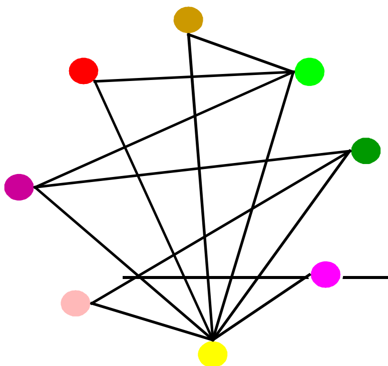
Choisir les variables en fonction de la structure du réseau de contraintes



Utilisation d'heuristiques

Choisir les variables en fonction de la structure du réseau de contraintes



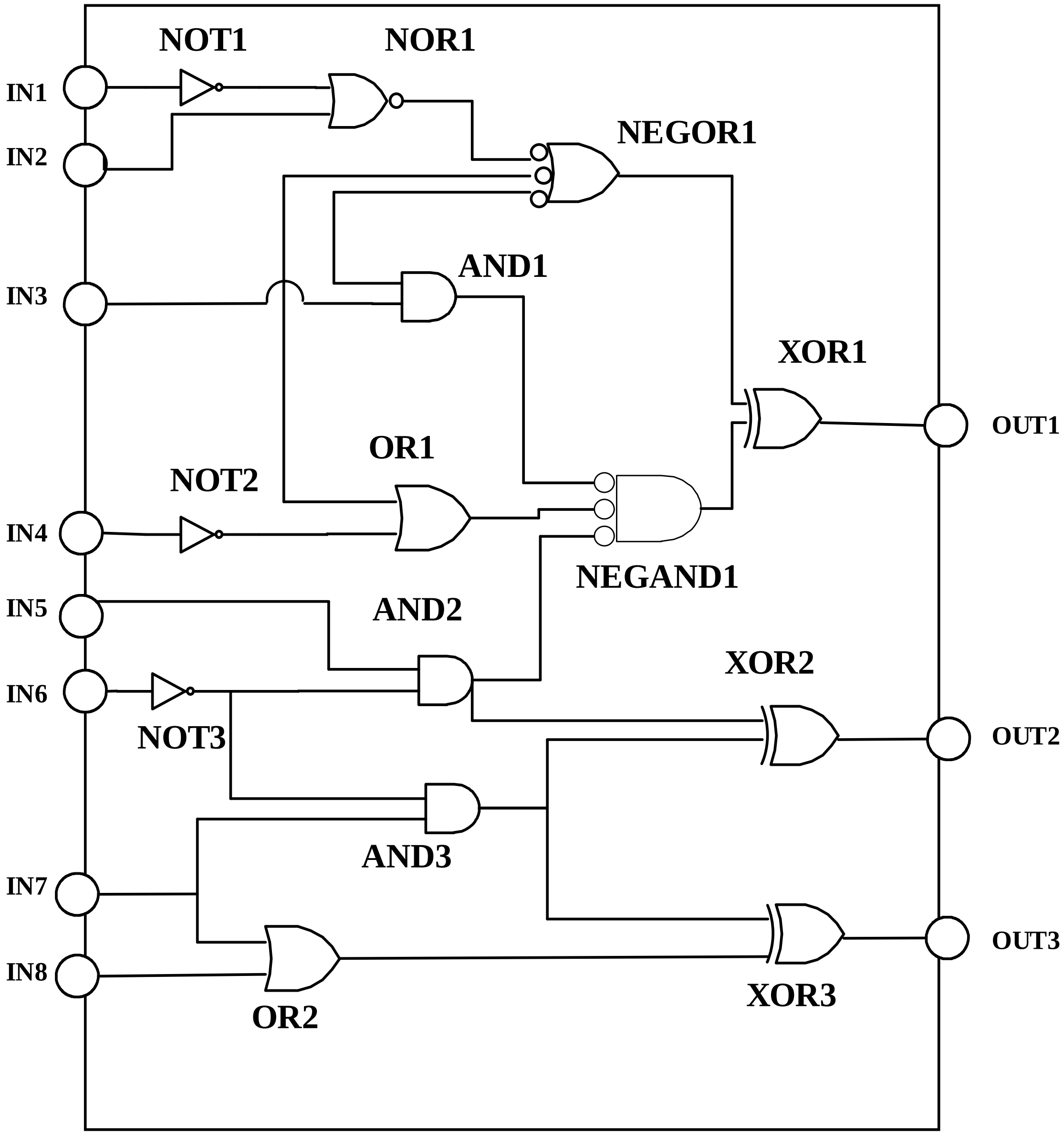


Apprendre pendant la résolution

Look-ahead Value Ordering (LVO)

Dans cette heuristique, on compte le nombre de fois où une valeur de la variable en cours d'instanciation est en conflit avec une valeur d'une variable non instanciée.
La valeur conduisant au plus petit nombre de conflits est choisi en priorité.
Cette heuristique s'utilise avec le FC et ses dérivées.

Les CSP distribués


NOT1
NOR1
IN1
IN2
NEGOR1
AND1
IN3
XOR1
OUT1
OR1
NOT2
IN4
NEGAND1
IN5
AND2
XOR2
IN6
NOT3
OUT2
AND3
IN7
OUT3
IN8
XOR3
OR2