

Chapitre 11 - Programmation par contraintes

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



Problème de satisfiabilité booléenne (SAT)

SAT (*Boolean satisfiability problem*) : problème fondamental en informatique

Soit une **formule logique** $\phi(X, Y, \dots)$ contenant des variables $X, Y, \dots$. La formule ϕ est **satisfiable** si il existe des valeurs pour $X, Y, \dots$ qui la rendent vraie.

Problème :

existe-t-il une assignation de valeurs pour $X, Y, \dots$ qui rend la formule vraie ?

SAT : exemple

Trouver une affectation de X et Y pour la formule suivante :

$$\phi(X, Y) = X \vee Y$$

Cela peut se faire avec une table de vérité:

X	Y	ϕ
0	0	0
0	1	1
1	0	1
1	1	1

$\rightarrow \phi(X, Y)$ est satisfaisable (*sat*).

SAT : exemple

Autre exemple :

$$\phi(X, Y) = (X \wedge Y) \wedge (Y \wedge \neg X)$$

X	Y	$X \wedge Y$	$Y \wedge \neg X$	$\phi(X, Y)$
0	0	0	0	0
0	1	0	1	0
1	0	0	0	0
1	1	1	0	0

C'est une formule insatisfaisable (*unsat*).

Résoudre SAT

C'est un problème **difficile** : **NP-complet**

- Exponentiel dans le pire des cas (conjecture)
- Peut-être résolu relativement efficacement dans de nombreux cas avec des **solveurs SAT**

Généralement, pour montrer qu'un problème est *NP-complet*, on le réduit à un problème de satisfaction.

Solveurs SAT

De nombreux solveurs très efficaces existent :

- Z3
- CVC5
- MiniSat
- CryptoMiniSat
- Lingeling
- ...

SAT : avec CLP(B)

CLP(B) = *constraint logic programming over boolean variables*

Disponible en Prolog :

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

(Important : ne pas oublier le `:-` depuis un fichier.)

Expressions booléennes

Expressions CLP(B) :

- 0 : faux
- 1 : vrai
- variable : valeur inconnue
- $\sim E$: négation de E
- $E1 + E2$: disjonction (*ou*)
- $E1 * E2$: conjonction (*et*)
- $E1 \# E2$: ou exclusif
- $X \wedge E$: quantification existentielle ($\exists x E$)
- $E1 ::= E2$: égalité
- $E1 =\backslash= E2$: différence
- $E1 =< E2$: implication ($\rightarrow$)

Utiliser CLP(B) pour résoudre SAT

$$\phi(X, Y) = X \vee Y$$

?- sat(X + Y).

sat(X =\= X * Y # Y).

À cause des priorités des opérateurs, la réponse doit se lire

sat(X =\= ((X * Y) # Y)).

Donc la formule est satisfaisable.

On obtient aussi une contrainte sur X et Y pour satisfaire la formule, qui est

$$X \neq ((X \wedge Y) \oplus Y).$$

Utiliser CLP(B) pour résoudre SAT

$$\phi(X, Y) = X \vee Y$$

```
?- sat(X + Y), labeling([X, Y]).
```

```
X = 0, Y = 1 ;
```

```
X = 1, Y = 0 ;
```

```
X = 1, Y = 1 ;
```

La relation `labeling/1` permet d'associer des valeurs aux variables.

Utiliser CLP(B) pour résoudre SAT

$$\phi(X, Y) = (X \wedge Y) \wedge (Y \wedge \neg X)$$

`?- sat((X * Y) * (Y * (~X))).`

false.

La formule n'est pas satisfaisable.

X	Y	$X \wedge Y$	$Y \wedge \neg X$	$\phi(X, Y)$
0	0	0	0	0
0	1	0	1	0
1	0	0	0	0
1	1	1	0	0

Utiliser CLP(B) pour résoudre SAT

$$\phi(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z)$$

?- sat($X * Y \# X * Z$).

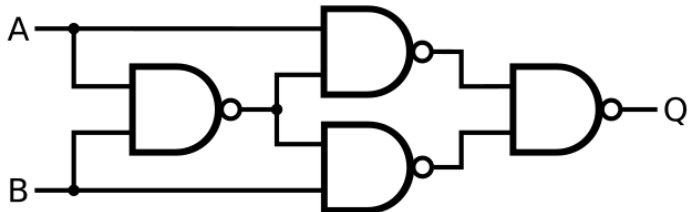
$X = 1$,

sat($Y \neq Z$).

La formule est satisfaisable.

X	Y	Z	$\phi(X, Y, Z)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Exemple : porte logique xor



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

```
nand_gate(X, Y, Z) :- sat(Z == ~(X*Y)).
```

```
xor(A, B, Q) :-
```

```
    nand_gate(A, B, T1),
```

```
    nand_gate(A, T1, T2),
```

```
    nand_gate(B, T1, T3),
```

```
    nand_gate(T2, T3, Q).
```

Exemple : porte logique xor

Est-ce que ce circuit est correct ?

```
?- xor(A, B, Q).  
sat(A==B#Q).
```

Et on sait que si $\text{xor}(B, Q) = A$, alors :

$$\begin{aligned}\text{xor}(A, B) &= \text{xor}(\text{xor}(B, Q), B) \\ &= Q\end{aligned}$$

Ou, on peut raffiner notre requête :

```
?- xor(A, B, Q), sat(Q == A#B).  
sat(A==B#Q)
```

Casse-tête logique : *knights and knaves*

Sur une île étrange, il y a deux types d'habitants :

- Les **chevaliers** (*knights*) disent toujours la vérité
- Les **fourbes** (*knaves*) mentent systématiquement

On doit déduire certaines informations

Knights and Knaves 1

Soit deux habitants **A** et **B**. **A** dit :

A dit « Je suis fourbe ou **B** est un chevalier ».

- Si **A** dit vrai, **A** est un chevalier (car il dit vrai) et **B** aussi
- Si **A** ment, **A** est fourbe, mais il a dit la vérité : contradiction

A et **B** sont donc des chevaliers.

Knights and Knaves 1

- Convention : $X = 1$ ssi X est un chevalier.
- Conséquence : comme les chevaliers disent vrai et les fourbes mentent, dès que X dit une phrase P , la formule $X \Leftrightarrow P$ est vraie.
- Autre conséquence : comme un fourbe ment toujours, le fait que X dise “je suis fourbe” est équivalent au fait que $\neg X$ est vraie.

A dit « Je suis fourbe ou **B** est un chevalier ».

Se traduit en

$$A \Leftrightarrow \neg A \vee B$$

?- `sat(A ::= (~A + B))`.

`A = B, B = 1`.

Knights and Knaves 2

A dit : « Je suis fourbe, mais **B** ne l'est pas »

```
?- sat(A == (~A * B)).
```

```
A = B, B = 0.
```

Knights and Knaves 3 : cardinalité

On peut utiliser des **contraintes de cardinalité** quand un nombre donné de propositions doivent être vraies.

A dit : « Au moins un de nous deux est fourbe »

```
?- sat(A == card([1, 2], [~A, ~B])).
```

```
A = 1,
```

```
B = 0.
```

À lire : « entre 1 et 2 des propositions dans la liste doivent être vraies »

```
?- sat(card([1], [X, Y])).
```

```
sat(X \= Y).
```

Knights and Knaves 4 : conjonction

Il y a maintenant trois habitants **A**, **B** et **C**.

A dit « **B** est fourbe »

B dit « **A** et **C** sont de même type »

?- $\text{sat}(A ::= \sim B), \text{sat}(B ::= (A ::= C)).$

C = 0,

$\text{sat}(A \neq B).$

CLP(FD)

CLP(B) peut être généralisé à tout domaine fini (*finite domain*, *fd*) :

- contraintes arithmétiques
- contraintes d'appartenance à une liste, à un domaine
- contraintes combinatoires

À rajouter au début du fichier :

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

CLP(FD) : contraintes arithmétiques

Contraintes arithmétiques possibles :

- $E1 \# = E2$: égalité
- $E1 \# \setminus = E2$: différence
- $E1 \# \geq E2$: comparaison
- $E1 \# \leq E2$
- $E1 \# > E2$
- $E1 \# < E2$

Expressions arithmétiques

Expressions arithmétiques :

- $-E$
- $E1 + E2$
- $E1 * E2$
- $E1 - E2$
- $E1^{E2}$: exposant
- $\min(E1, E2)$
- $\max(E1, E2)$
- $\text{mod}(E1, E2)$: modulo
- $\text{abs}(E)$: valeur absolue
- $E1 // E2$: division tronquée

Différence avec is

```
?- X #= 1+2.
```

```
X = 3.
```

```
?- 3 #= Y+2.
```

```
Y = 1.
```

Alors que :

```
?- 3 is Y+2.
```

```
ERROR: is/2: Arguments are not sufficiently instantiated
```

```
?- 3 == Y+2.
```

```
ERROR: ==/2: Arguments are not sufficiently instantiated
```

Factorielle

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

On place les contraintes en premier (préférable).

Factorielle

Plus puissant :

```
?- fac(N, 1).
```

```
N = 0 ;
```

```
N = 1 ;
```

```
false.
```

```
?- fac(N, 3).
```

```
false.
```

```
?- fac(N, 720).
```

```
N = 6 .
```

Ce n'est pas possible avec `is` (voir la première façon d'écrire `fac`).

Contrainte in/2

in/2 : contraint une variable a un domaine

```
?- A in 1..3, B in 2..4.
```

```
A in 1..3, B in 2..4.
```

```
?- A in 0..2, B in 0..2, A+B #= B.
```

```
A = 0, B in 0..2.
```

Contrainte label/1

`label([V1, V2, ...])` essaye d'assigner une valeur à chacune des variables.

```
?- A in 0..2, B in 0..2, A+B #= B, label([A,B]).
```

```
A = B, B = 0 ;
```

```
A = 0, B = 1 ;
```

```
A = 0, B = 2.
```

```
?- A in 1..2, B in 2..3, label([A, B]).
```

```
A = 1, B = 2 ;
```

```
A = 1, B = 3 ;
```

```
A = B, B = 2 ;
```

```
A = 2, B = 3.
```

Contrainte ins/2

ins/2 : comme in/2, mais pour chaque élément d'une liste

```
?- [A,B,C] ins 0..1.
```

```
A in 0..1, B in 0..1, C in 0..1.
```

Contrainte all_distinct/1

all_distinct/1 : contraint tous les éléments d'une liste à être distincts

```
?- [X,Y] ins 0..1,  
    all_distinct([X,Y]),  
    label([X,Y]).  
X = 0, Y = 1 ;  
X = 1, Y = 0.
```

Programmation par contraintes

Pour modéliser un problème avec la programmation par contraintes :

- ➊ Établir les contraintes
- ➋ Chercher la solution - Prolog ou la bibliothèque CLP s'en occupe pour nous

Exemple 1 : arithmétique verbale

Chaque lettre désigne un **chiffre différent** (Dudeney, 1924) :

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

Quelle valeur assigner aux lettres pour rendre l'équation vraie ?

Exemple 1 : arithmétique verbale (en Prolog)

```
puzzle(S, E, N, D, M, O, R, Y) :-  
    Vars = [S, E, N, D, M, O, R, Y],  
    Vars ins 0..9,  
    all_distinct(Vars),  
    S * 1000 + E * 100 + N * 10 + D  
+      M * 1000 + O * 100 + R * 10 + E  
#= M * 10000 + O * 1000 + N * 100 + E * 10 + Y,  
M #\= 0,  
S #\= 0.
```

Exemple 1 : arithmétique verbale (résolution)

```
?- puzzle([S, E, N, D, M, O, R, Y]).  
S = 9,  
M = 1,  
O = 0,  
E in 4..7,  
91*E+D+10*R#=90*N+Y,  
all_distinct([9, E, N, D, 1, 0, R, Y]),  
N in 5..8,  
D in 2..8,  
R in 2..8,  
Y in 2..8.
```

Exemple 1 : arithmétique verbale (résolution)

Il faut utiliser `label/1` pour forcer Prolog à nous donner les solutions :

```
?- puzzle([S, E, N, D, M, O, R, Y]), label([S, E, N, D, M, O, R, Y]).
```

```
S = 9,
```

```
E = 5,
```

```
N = 6,
```

```
D = 7,
```

```
M = 1,
```

```
O = 0,
```

```
R = 8,
```

```
Y = 2 .
```

Cette solution est la seule.

Exemple 2 : inversion d'un hash non sécuritaire

```
function CheckPassword(password) {  
    hash = 0  
    for (var i = 0, len = password.length; i < len; i++) {  
        hash *= 31  
        hash += password[i].charCodeAt(0);  
    }  
    console.log(hash)  
    if (hash == 3229186608006) {  
        alert('Correct!!')  
        return true;  
    } else {  
        alert('Wrong...!')  
        return false;  
    }  
}
```

Source

Exemple 2 : inversion d'un hash non sécuritaire

On peut implanter le calcul du hash en Prolog :

```
hash([], M, M).  
hash([H|T], M, Res) :-  
    M1 is M * 31 + H,  
    hash(T, M1, Res).
```

Exemple 2 : inversion d'un hash non sécuritaire

Avec CLP(FD) :

```
hash_clp([], M, M).
hash_clp([H|T], M, Res) :-
    M1 #= M * 31 + H,
    hash_clp(T, M1, Res).

solve(Solution) :-
    % Limite la recherche à 15 caractères.
    Length in 0..15,
    length(Codes, Length),
    % Chaque caractère est entre 'a' et 'z'.
    Codes ins 97..122,
    hash_clp(Codes, 0, 3229186608006),
    % Transforme la solution en lettres
    maplist(char_code, Solution, Codes).
```

Exemple 2 : inversion d'un hash non sécuritaire

```
?- solve(L).
```

```
L = [r, e, i, n, d, e, e, r]
```

Exemple 3 : 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

- Chaque case contient un nombre entre 1 et 9
- Il y a 9 lignes
- Il y a 9 colonnes
- Chaque ligne contient des éléments distincts
- Chaque colonne contient des éléments distincts
- Chaque bloc contient des éléments distincts

Prédicats supplémentaires sur les listes

- `append/2` : vrai pour une liste de listes et la concaténation de toutes ses listes (différent de `append/3` que nous avons déjà vu).

```
?- append([[1], [], [], [2, 3], [], [4, 5]], L).
```

```
L = [1, 2, 3, 4, 5].
```

- `same_length/2` : vrai pour deux listes qui ont la même longueur

```
?- same_length([1,2], X).
```

```
X = [_ , _].
```

Prédicats supplémentaires sur les listes

- `maplist/2` : applique un prédicat partiel à chaque élément d'une liste

```
?- maplist(same_length([1, 2]), [[3, 4], [5, 6]]).
```

```
true.
```

- `transpose/2`: calcule la transposée d'une liste de listes

```
?- transpose([[1, 2, 3], [4, 5, 6]], X).
```

```
X = [[1, 4], [2, 5], [3, 6]].
```

Exemple 3 : Sudoku (en Prolog)

```
sudoku(Rows) :-  
    length(Rows, 9),  
    maplist(same_length(Rows), Rows),  
    append(Rows, Vs), Vs ins 1..9,  
    maplist(all_distinct, Rows),  
    transpose(Rows, Columns),  
    maplist(all_distinct, Columns),  
    Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],  
    blocks(As, Bs, Cs), blocks(Ds, Es, Fs), blocks(Gs, Hs, Is).
```

```
blocks([], [], []).  
blocks([N1,N2,N3|Ns1],  
        [N4,N5,N6|Ns2],  
        [N7,N8,N9|Ns3]) :-  
    all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),  
    blocks(Ns1, Ns2, Ns3).
```

Exemple 3 : Sudoku (résolution)

```
?- sudoku([[_,_,_,_,_,_,_,_,_],  
           [_,_,_,_,_,3,_,8,5],  
           [_,_,1,_,2,_,_,_,_],  
           [_,_,_,5,_,7,_,_,_],  
           [_,_,4,_,_,_,_,1,_],  
           [_,9,_,_,_,_,_,_,_],  
           [5,_,_,_,_,_,_,_,7,3],  
           [_,_,2,_,_,1,_,_,_],  
           [_,_,_,_,_,4,_,_,_,9]]).
```

true.

Programmation par contraintes avec Z3

```
from z3 import *
```

```
def sudoku_solver(board):  
    s_solver = Solver()  
    s_solver.add([And(1 <= box, box <= 9) for box in chain(*boxes)])  
    for i in range(len(boxes)):  
        s_solver.add(Distinct(boxes[i]))  
        s_solver.add(Distinct([row[i] for row in boxes]))  
  
    for rows in [[0,1,2],[3,4,5],[6,7,8]]:  
        for cols in [[0,1,2],[3,4,5],[6,7,8]]:  
            s_solver.add(Distinct([boxes[r][c] for r in rows for c in cols]))  
  
    for i in range(len(board)):  
        for j in range(len(board[0])):  
            if board[i][j] != 0:  
                s_solver.add(boxes[i][j] == board[i][j])  
  
    return s_solver
```

Source

Programmation par contraintes avec Z3

```
import z3
```

```
for length in range(16):
    s = z3.Solver()
    password = [z3.Int(f'password[{i}]') for i in range(length)]
    hash = 0
    for i in range(length):
        s.add(password[i] > 97)
        s.add(password[i] < 127)
        hash *= 31
        hash += password[i]

    s.add(hash == 3229186608006)
    if s.check() == z3.sat:
        model = s.model()
        print(''.join([chr(model.evaluate(password[i]).as_long())
                        for i in range(length)]))
```

Programmation par contraintes : utilité

Les solveurs SAT (et FD) sont utilisés dans beaucoup de domaines :

- vérification formelle de matériel et logiciel
- détection automatique de bugs
- analyse de protocoles cryptographiques
- optimisations dans les compilateurs
- synthèse de circuits électroniques
- planification

Ils offrent des moyens élégants et faciles à déployer pour résoudre ces problèmes.

Prolog peut être également vu comme du **CLP(H)** où H correspond aux *termes de Herbrand*.

Implémentation d'un système de CSP

Définition d'un CSP

Un problème de satisfaction de contrainte (*CSP* pour *constraint satisfaction problem*) est défini par :

- X : un ensemble de variables
- D : un ensemble de domaine, associé à chaque variable
- C : des contraintes sur les variables

Utilité des CSP

Si on a un problème qu'on peut exprimer sous forme de CSP, on peut utiliser un *solveur* existant :

- on a une modélisation déclarative du problème
- on profite d'une implémentation existante
- on a de bonnes performances

Exemple de CSP

Par exemple :

- Variables : X_1 et X_2
- Domaines : $D_1 = D_2 = \mathbb{N}$
 - peut être continu ou discret
 - peut être fini ou infini
- Contraintes : $X_1 \neq X_2$
 - peuvent avoir des formes restreintes (par exemple : contraintes linéaires)
 - ou des formes générales

Il peut exister plusieurs solutions à un problème, par exemple :

- $X_1 = 1, X_2 = 2,$
- $X_1 = 42, X_2 = 37,$
- ...

CSP en OCaml

Exemple de modélisation pour des problèmes avec deux types de contraintes : contraintes unaires et contraintes binaires

```
module CSP = struct  
  type 'a t = {  
    (* Les variables (nommées) *)  
    variables : string list;  
    (* Le domaine initial de chaque variable *)  
    domains : (string * 'a list) list;  
    (* Les contraintes unaires *)  
    unary : (string * ('a -> bool)) list;  
    (* Les contraintes binaires *)  
    binary : ((string * string) * ('a -> 'a -> bool)) list;  
  }  
end
```

Coloration de carte

Un exemple typique : comment colorier une carte de sorte à ce que les régions adjacentes n'ont pas la même couleur ?



(Source: Russel et Norvig, AI: A Modern Approach, 2005, Chapitre 6)

Coloration de carte : modélisation

On associe une variable à chaque région :

$$X = \{WA, NT, Q, NSW, V, SA, T\}$$

On choisit les couleurs, par exemple trois couleurs :

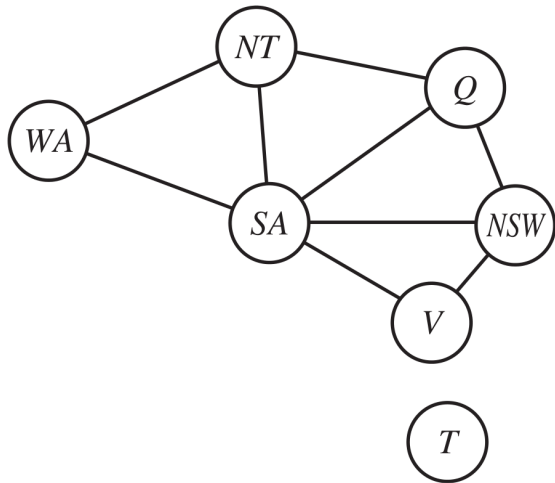
$$D_i = \{\text{rouge}, \text{vert}, \text{bleu}\}$$

On définit les contraintes :

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$$

Graphe de contrainte

On peut voir ce problème sous forme de graphe.



Coloration de carte : en OCaml

```
let map_coloring =  
  let colors = ["red"; "green"; "blue"] in  
  let states = ["WA"; "NT"; "Q"; "NSW"; "V"; "SA"; "T"] in  
  let different x y = ((x, y), fun vx vy -> vx <> vy) in  
  CSP.{  
    variables = states;  
    domains = List.map (fun state -> (state, colors)) states;  
    unary = [];  
    binary = [different "SA" "WA"; different "SA" "NT";  
              different "SA" "Q"; different "SA" "NSW";  
              different "SA" "V"; different "WA" "NT";  
              different "NT" "Q"; different "Q" "NSW";  
              different "NSW" "V"];  
  }
```

Résolution

Comment résoudre le problème ?

Solution naïve : on énumère les solutions et vérifie lesquelles satisfont aux contraintes

Vérification d'une solution

Pour vérifier qu'une solution satisfait au problème, il faut :

- que les valeurs soient dans les domaines
- que les contraintes unaires soient satisfaites
- que les contraintes binaires soient satisfaites

Vérification d'une solution

```
let satisfied_by (problem : 'a t) (values : (string * 'a) list) : bool =
  List.for_all (fun variable ->
    match List.assoc_opt variable values with
    | None -> false (* Variable manquante de la solution *)
    | Some value ->
      let domain = List.assoc variable problem.domains with
      let in_domain = List.mem value domain in
      let unary_sat = match List.assoc_opt variable problem.unary with
        | None -> true (* pas de contrainte, c'est bon ! *)
        | Some c -> c value in
      in_domain && unary_sat) problem.variables &&
  List.for_all (fun ((var1, var2), c) ->
    match List.assoc_opt var1 values, List.assoc_opt var2 values with
    | Some v1, Some v2 -> c v1 v2
    | _ -> false)
  problem.binary
```

Résolution naïve

On peut énumérer les solutions possibles :

```
let rec enumerate (possible_values : (string * 'a list) list)
  : (string * 'a) list list =
  match possible_values with
  | [] -> [[]]
  | (var, values) :: rest ->
    let sub_solutions = enumerate rest in
    List.flatten
      (List.map (fun sub_solution ->
        List.map (fun value ->
          (var, value) :: sub_solution)
          values)
        sub_solutions)
```

Résolution naïve

Une solution nous suffit :

```
let naive_solve (problem : 'a CSP.t)
  : (string * 'a) list option =
  List.find_opt (CSP.satisfied_by problem)
    (enumerate problem.domains)
```

Résolution naïve

Exemple :

```
# naive_solve map_coloring;;  
- : (string * string) list option =  
Some  
  [("WA", "green"); ("NT", "blue"); ("Q", "green"); ("NSW", "blue");  
   ("V", "green"); ("SA", "red"); ("T", "red")]
```

Résolution naïve

Mais :

- ❶ ça ne supporte pas les domaines infinis
- ❷ pour n variables ayant m valeurs possibles chaque, $\mathcal{O}(m^n)$ solutions
- ❸ on énumère tout avant de vérifier

On va résoudre le dernier point.

Résolution avec backtracking

- On assigne la première valeur possible à la première variable
- On essaie de résoudre le problème avec $n - 1$ variables
- Si on bloque, on revient en arrière

C'est similaire à la façon d'évaluer les programmes Prolog.

Résolution avec backtracking

Cas de base : si on a trouvé toutes les assignments, on a fini

```
let backtrack_search (csp : 'a CSP.t) : (string * 'a) list option =  
  let is_complete (assignment : (string * 'a) list) : bool =  
    List.length (List.map fst assignment) = List.length csp.variables in  
  let select_unassigned assignment = ...  
  let rec aux (assignment : (string * 'a) list) : (string * 'a) list option =  
    if is_complete assignment then  
      Some assignment  
    else  
      ...  
  in  
  aux []
```

Résolution avec backtracking

Cas récursif : toutes les variables ne sont pas assignées

- on sélectionne une assignation
- on vérifie si elle fait sens
- on continue la recherche

Résolution avec backtracking

```
let rec aux (assignment : (string * 'a) list) : (string * 'a) list option =  
  ...  
  else  
    let var = select_unassigned assignment in  
    let possible_values = List.assoc var csp.domains in  
    List.fold_left (fun solution value ->  
      match solution with  
      | Some s -> Some s  
      | None ->  
        let new_assignment = (var, value) :: assignment in  
        if CSP.consistent_with csp new_assignment then  
          aux new_assignment  
        else  
          None) None possible_values in  
aux []
```

Résolution avec backtracking

Il y a beaucoup d'améliorations possibles :

- la sélection des variables peut être plus adéquate
 - sélection naïve : dans l'ordre d'apparence
 - mieux : sélection de celle qui a le moins de valeur possible
 - alternative : celle qui est dans le plus de contraintes
- *forward checking* : utiliser des algorithmes de consistance après chaque sélection pour réduire la recherche
- et beaucoup d'autres

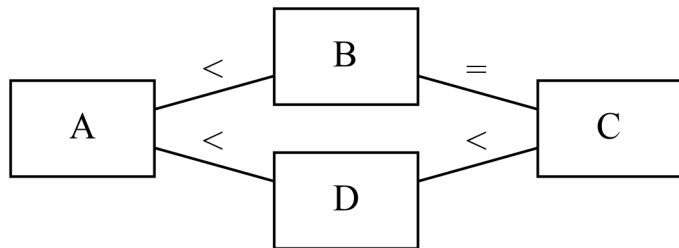
Consistance

Exemple :

$$X = \{A, B, C, D\}$$

$$D_i = \{1, 2, 3\}$$

$$C = \{C > 2, A < B, B = C, D < C, A \leq D\}$$



Consistance de nœud

Exemple :

$$X = \{A, B, C, D\}$$

$$D_i = \{1, 2, 3\}$$

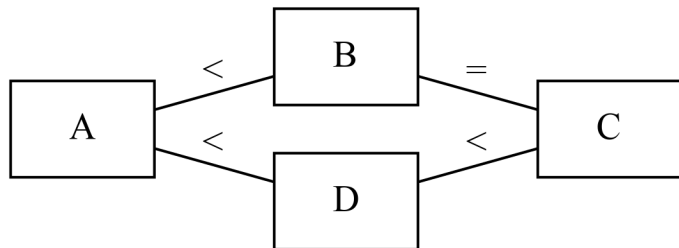
$$C = \{C > 2, A < B, B = C, D < C, A \leq D\}$$

Même si le domaine de C contient 3 valeurs, on peut utiliser la contrainte unaire pour réduire son espace de valeur

C'est la **consistance de nœud**

$$C > 2 \wedge C \in \{1, 2, 3\} \Rightarrow C = 3$$

Consistance d'arcs



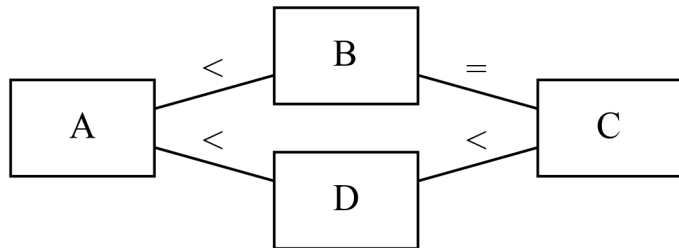
On sait désormais que $C = 3$

Des contraintes binaires, on voit que $D < C$, donc on doit avoir :

$$C = 3 \wedge D < C \wedge D \in \{1, 2, 3\} \Rightarrow D \in \{1, 2\}$$

C'est la **consistance d'arc** entre D et C

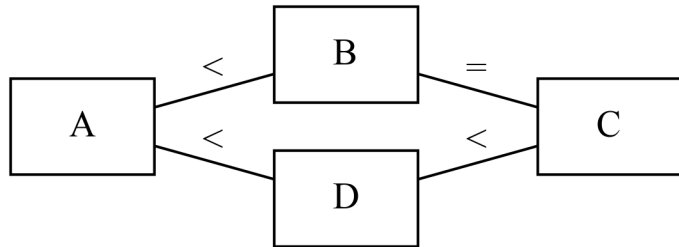
Consistance d'arcs



On peut faire de même avec l'arc A - D, on trouve donc les seules valeurs possibles pour A et D :

$$A \in \{1, 2, 3\} \wedge D \in \{1, 2\} \wedge A < D \Rightarrow A = 1 \wedge D = 2$$

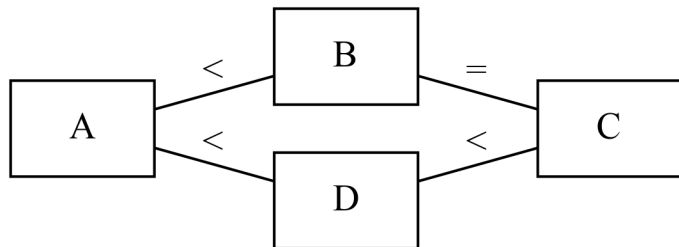
Consistance d'arcs



La consistance d'arc permet également de trouver la valeur de B

$$C = 3 \wedge B = C \Rightarrow B = 3$$

Consistance d'arcs



Donc, on a trouvé :

- $A = 1$
- $B = 3$
- $C = 3$
- $D = 2$

Consistance

On peut utiliser des algorithmes tels que l'algorithme AC3 (*Arc Consistency Algorithm #3*) pour automatiser ce raisonnement.

- ce n'est pas complet : on ne trouvera pas toujours une solution
- mais cela se combine bien avec le backtracking pour accélérer la convergence

On reste dans un problème **NP-complet**, mais en pratique on peut résoudre certains problèmes efficacement.

Pour aller plus loin

Il existe des modèles plus restreints avec des algorithmes spécifiques de résolution, par exemple :

- Problème SAT : résolution généralement basée sur l'algorithme **DPLL**
- **SMT** (*SAT Modulo Theories*) : extension de SAT pour modéliser d'autres domaines que les booléens

Concepts importants

- **Programmation par contrainte** : approche déclarative pour la résolution de problèmes modélisables par contraintes
 - *CSP (constraint satisfaction problem)* : les problèmes à résoudre
 - *CLP (constraint logic programming)* : une approche pour les résoudre
- **Modélisation d'un problème** : représentation du problème sous formes de variables et contraintes
- **Solveur** : outil qui automatise la résolution d'un problème
 - Intègre généralement une forme de **retour en arrière** (*backtracking*)