Tout différent

La contrainte *AllDifferent(x1, ..., xn)* demane que les variables *x1, ..., xn* aient toutes des valeurs différentes.

# Notions de support

Support de domaine :

La variable *x* a un support de domaine pour la contrainte *C* si pour toute valeur *u* ∈ *Dx* , il existe une affectation *a* satisfaisant *C* telle que *a[x] = u* et ∀ *y* ∈ portée de *C, a[y] = Dy.*

Support d’intervalle :

La variable x a un support d’intervalle si ∀ *u* ∈ *Dx* , ∃ *a* ∈ *C* , *a[x] = u* et ∀ *y* ∈ portée de *C,* min Dy <= a[y] <= max Dy.

Consistance de borne :

Pour chaque contrainte C, pour chaque variable x ∈ portée de C, chacune des valeurs min Dx et max Dx a un support d’intervalle dans C.

Consistance d’intervalle :

∀ C, ∀ x ∈ portée de C, ∀ u ∈ Dx , x = u a un support d’intervalle dans C.

Consistance de domaine :

∀ C, ∀ x ∈ portée de C, ∀ u ∈ Dx , x = u a un support de domaine dans C.

# Algortihmes pour établir AllDifferent sur n variables

Consistance de domaine : [Régin 94] : O(n2,5)

Consistance de d’intervalle : [Leconte 96] : O(n2)

Consistance de borne : [Puget 98] : O(n log n)

[Mehlhorn, Thiel 00] : O(n+sort(n))

[Lopez-Ortiz 03] : Même complexité, Meilleur en pratique.

Théorème de Hall

∀ *S* ⊆ U(*Dx*), |*S*| >= | { *xi* | *Dxi* ⊆ *S* } |

Intervalle de Hall

Soit I un intervalle. On note *vars*(I) = { xi | Dxi ⊆ I }. I est un intervalle de Hall ssi   
|I| = |*vars*(I)|.

Si I est un intervalle de Hall, pour tout *y* tel que *Dy* ⊈ I, on peut enlever I ∩ *Dy*.

# Détecter les intervalles de Hall

## Algorithme de Pujet

maxi := max(*Dxi*)

mini := min(*Dxi*)

Cki := |{ j<=i | minj >= k }|

Vki := maxi + 1 - k - Cki

Théorème :

Vki = 0 ssi [k, maxi] est un intervalle de Hall.

1. Trier les intervalles selon maxi.
2. Parcourir les variables dans cet ordre et maintenir les compteurs   
   Cki = |{ j<=i | minj >= k }|.
3. Si Cki = maxi + 1 - k, alors [k, maxi] est un intervalle de Hall.

Remarques :

1. On peut se restreindre à des compteurs de la forme Cki où k = minj.
2. Les compteurs sont maintenus par des arbres binaires de recherche stockant les indices j, ordonnés par minj.

## Domination

Lemme :

On supposes les i triés mar maxi croissants.

Si k < k’ et Vki <= Vk’i , alors : Vki+1 <= Vk’i+1

Algorithme de Lopez-Ortiz

Problèmes de satisfaction de contraintes à domaine binaire

Ici, le domaine est : *D =* {Vrai, Faux}.

Litéral :

Variable ou négation d’une variable

Clause :

Ensemble de litéraux, que l’on évalue à vrai ssi au moins un de ces litéraux est évalué à vrai.

Formule :

Ensemble de clauses, que l’on évalue à vrai ssi toutes les clauses s’évalues à vrai.

→ Clause = Litéral ∨ Litéral ∨ Litéral ...

→ Formule = Clause ∧ Clause ∧ Clause ...

# Familles importantes de CSP booléens

*k*-SAT :

Chaque clause est composée d’exactement *k* litéraux.

HornSAT :

Dans toutes les clauses, au plus un litéral est positif.

HornSAT renommable :

S’il existe un ensemble de variable *R* tel que l’on obtient un HornSAT si l’on remplace tous les *x* ∈ *R* par ⅂*x*.

XORSAT :

Toutes les clauses sont de la forme : *z1* ⨁ *z2* ⨁ ... (les *zi* étant des litéraux), s’évaluant à vrai ssi le nombre de litéraux évalués à vrai est impair.

# Un algorithme de complexité linéaire pour résoudre une formule 2-SAT

NB : [ ⅂*a* ∨ *b* ] ⟺ [ *a* ⟹ *b* ] ⟺ [ ⅂*b* ⟹ ⅂*a* ]

À une formule, on associe le graphe orienté dont sommets sont les litéraux et les arrêtes sont données par les implications traduisant les clauses (deux par clause).

Par exemple, pour la formule (*a*∨*b*)∧(⅂*a*∨*c*), on convertit cette dernière en [(⅂*a*⟹*b*)∧(⅂*b*∧*a*)] ∧ [(*a*⟹*c*)∧(⅂*c*⟹⅂*a*)]. La graphe correspondant a pour sommets {*a*,*b*,*c*,⅂*a*,⅂*b*,⅂*c* }, et pour arrêtes {(⅂*a*,*b*), (⅂*b*,*a*), (*a*,*c*), (⅂*c*,⅂*a*)}.

Lemme

La formule est satisfiable ssi il n’existe pas de variable *x* telle que *x* et ⅂*x* soient dans une même composante connexe.

# Un algorithme pour HornSAT

On est en présence de 3 types de clauses :

1. Un seul litéral, qui est positif.
2. Plusieurs litéraux négatif dont exactement un est positif.
3. Plusieurs litéraux, tous négatifs.

Tant qu’il existe une clause *x* de la forme 1 :

Poser : *x* = vrai.

Enlever toutes les clauses *C* telles que *x* ∈ *C*.

Enlever ⅂*x* de toutes les clauses.

Poser faux à toutes les autres variables.

# Algorithme de Daris-Putnam

**DavisPutman(F) :**

F ⟵ Simplifier(F)

Si F ne contient aucune variable alors retourner la valeur de F.

X ⟵ une valeur de F.

Si DavisPutman(F|x=0) alors retourner Vrai,

Sinon retourner DavisPutman(F|x=1).

## Règles de simplification

**Simplifier(F) :**

1. Si un litéral apparait plusieurs fois dans une clause, alors n’en garder qu’un seul : *x ∨ x ∨ C → C*
2. Si une clause contient une variable et sa négation, alors la remplacer par Vrai : *x ∨ ⅂x ∨ C → C*
3. Si les litéraux d’une clause C1 sont inclus dans ceux d’une clause C2, alors supprimer C2 : *C1 ⊆ C2 ⟹ C2 → ⨂*
4. Si une variable est de la forme x, remplacer x par Vrai et ⅂x par Faux partout dans F : *C = x ⟹ x = Vrai & ⅂x = Faux*
5. On retire Faux de toute clause non restreinte à Faux :

Faux ∨ C → C

1. Si une clause contient Vrai, on la supprime :

Vrai ∨ C → *⨂*

## Règle supplémentaire : Résolution

1. Si une variable x apparait p fois positivement et q fois négativement, et si F = [x ∨ C1] ∧ ... ∧ [x ∨ Cp] ∧ [⅂x ∨ D1] ∧ ... ∧ [⅂x ∨ Dq] ∧ U :  
   on remplace F par :

F = [C1 ∨ D1] ∧ ... ∧ [C1 ∨ Dp] ∧ ... ∧ [Cp ∨ D1] ∧ ... ∧ [Cp ∨ Dp] ∧ U

(On ne fera cela que lorsque p.q ≤ p+q (min(p,q) = 1 ou max(p,q) = 2).

# Un algorithme pour 3SAT

## Alogrithme de Monien-Speckenmeyer-3

Cf. Poly.

## Complexité du problème de 3-colorabilité

On décrit ainsi le problème de 3-colorabilité : l’ensemble des couleurs est {1,2,3}, l’ensemble des pays est V (noeuds du graphe), les pays sont adjacents ssi (u,v) ∈ E (arrêtes du graphe).

On utilise les variables binaires *Xu,i* valent *vrai* ssi le pays *u* est colorié avec la couleur *i*.

Les contraintes sont :

* ∀ u ∈ V, ∀ i ∈ {1,2,3}, ⅂Xu,i ∨ ⅂Xu,j
* ∀ u ∈ V, Xu,1 ∨ ⅂Xu,2 ∨ ⅂Xu,3
* ∀ (u,v) ∈ E, ∀ i ∈ {1,2,3}, ⅂Xu,i ∨ ⅂Xv,i

On veut établir la complexité de la résolution de ce problème avec utilisation de l’algorithmme Monien-Speckenmeyer-3.

Sur cette formule 3SAT :

1. Soit l’algorithme choisit une clause avec au plus 2 litéraux,
2. Soit la formule est trivialement satisfiable ou non satisfiable.

Le cas 2 donne un O(n). Le cas 1 donne comme majoration de la complexité :   
T(n) ≤ T(n-1) + T(n-2).

On obtient : T(n) ≤ F(n), avec F la suite de Fibonacci. Or : F(n) ~ φn , avec φ le nombre d’or : φ = (1+sqrt(5))/2 ≈ 1,618 , racine de X²-X-1.