

INF8175 - Intelligence artificielle

Méthodes et algorithmes

Module 4: Programmation par contraintes



POLYTECHNIQUE
MONTRÉAL

Quentin Cappart

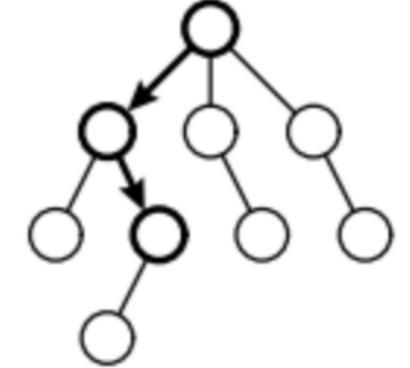
Contenu du cours

Raisonnement par recherche (essais-erreurs avec de l'intuition)

Module 1: Stratégies de recherche

Module 2: Recherche en présence d'adversaires

Module 3: Recherche locale



Raisonnement logique

Module 4: Programmation par contraintes

Module 5: Agents logiques



Stench		Breeze	PIT
Agent	Breeze	Stench Gold	PIT
Stench		Breeze	
Agent	Breeze	PIT	Breeze

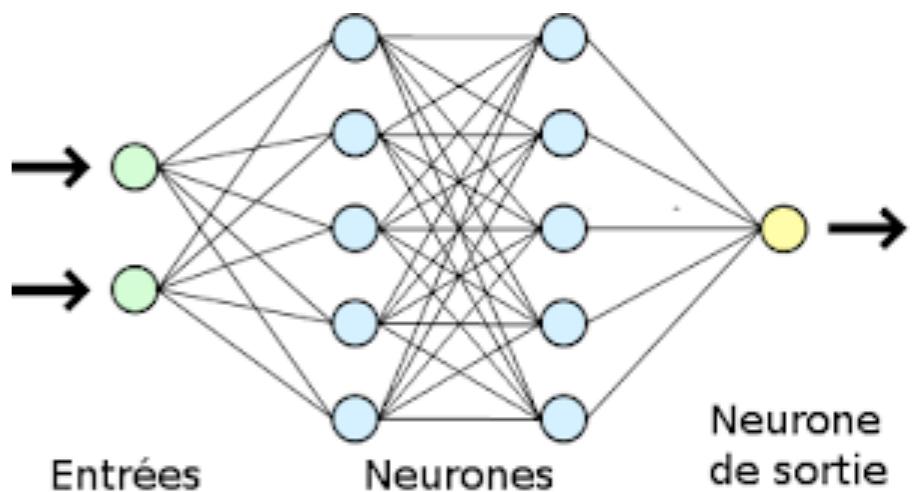
Raisonnement par apprentissage

Module 6: Apprentissage supervisé

Module 7: Réseaux de neurones et apprentissage profond

Module 8: Apprentissage non-supervisé

Module 9: Apprentissage par renforcement



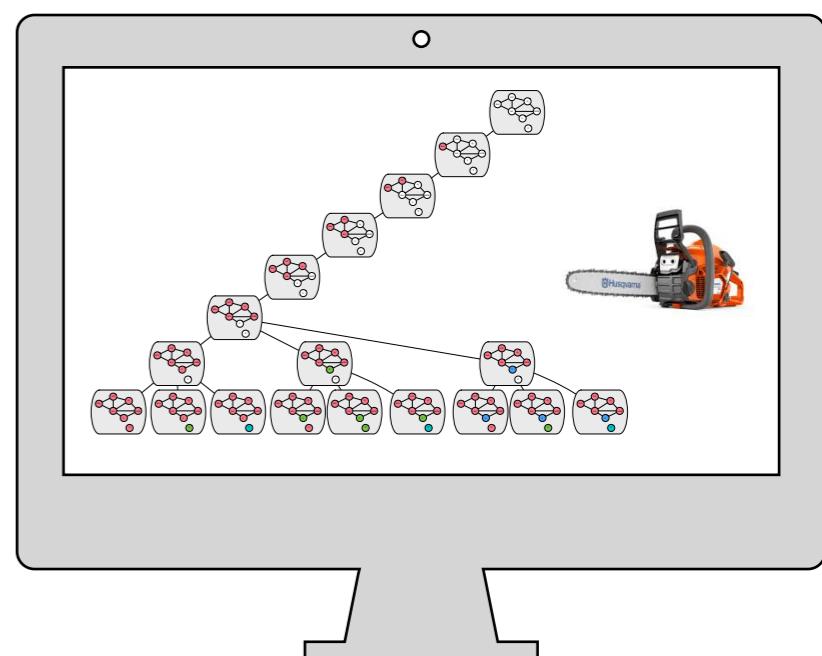
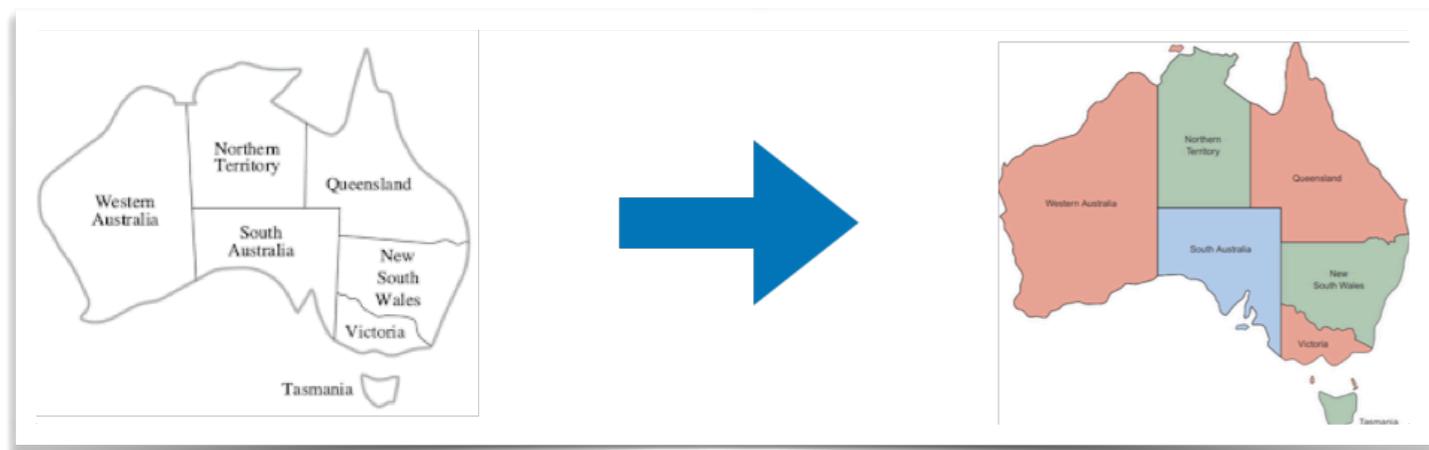
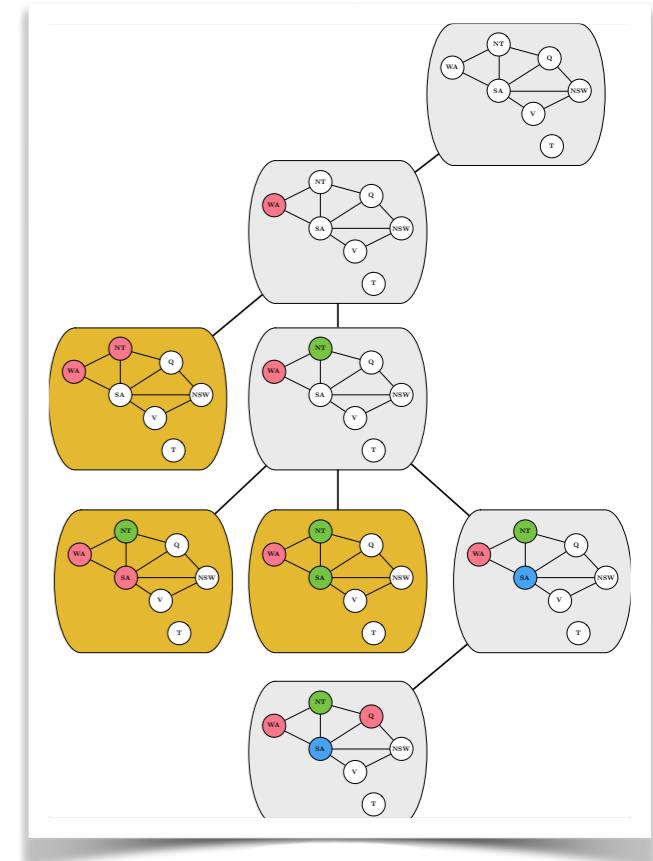
Considérations pratiques et sociétales

Module 10: Utilisation en industrie, éthique, et philosophie

Table des matières

Programmation par contraintes

1. Limitations et difficultés de la recherche locale
2. Concepts et principes fondamentaux de la programmation par contraintes
3. Formalisation des problèmes combinatoires
4. Modélisation en programmation par contraintes
5. Recherche avec retours-arrières (*backtracking search*)
6. Propagation de contraintes (*arc consistency*)
7. Algorithme du point fixe
8. Fonctionnement d'un solveur de programmation par contraintes



Retour sur le module précédent

Problèmes combinatoires (CSP ou COP)

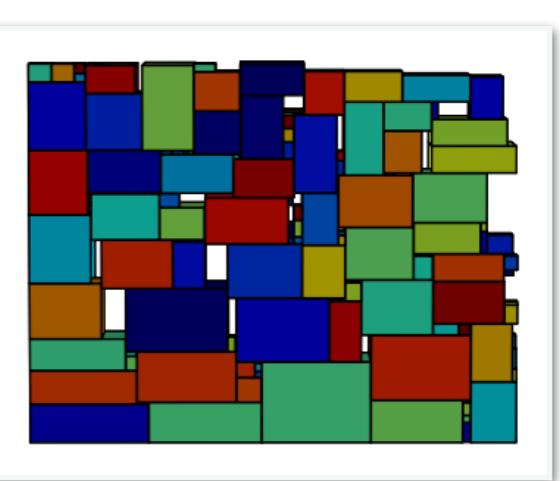
Objectif: Trouver une solution faisable (ou optimale) parmi un ensemble fini de solutions

Nature d'une solution: un état, qui nous est inconnu (et non une séquence d'action)

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		

Investissement	Coût (\$)	Revenu espéré dans 10 ans (\$)
A	200	1 000
B	200	1 000
C	200	1 000
D	500	10 000
E	500	10 000
F	800	13 000
G	300	7 000

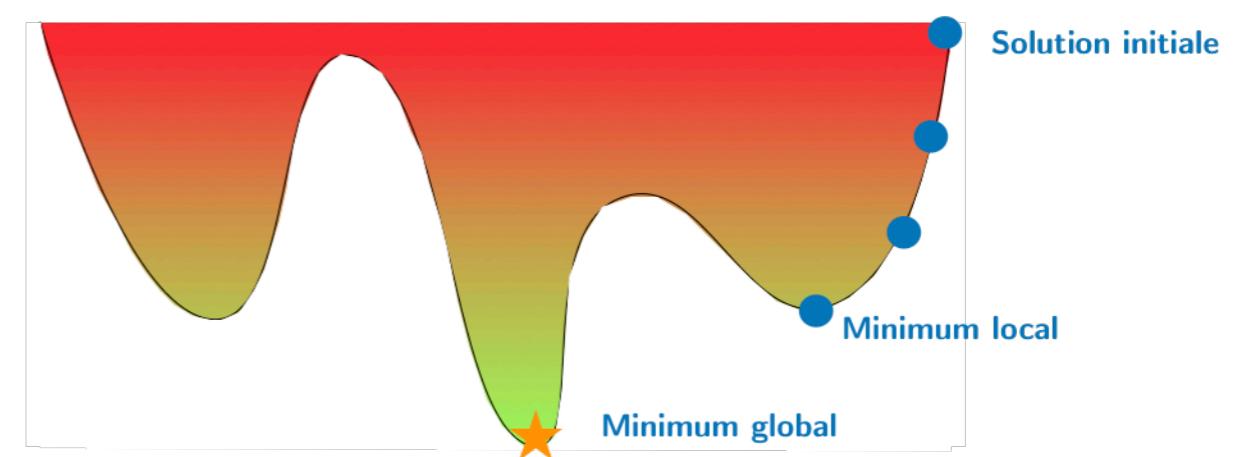
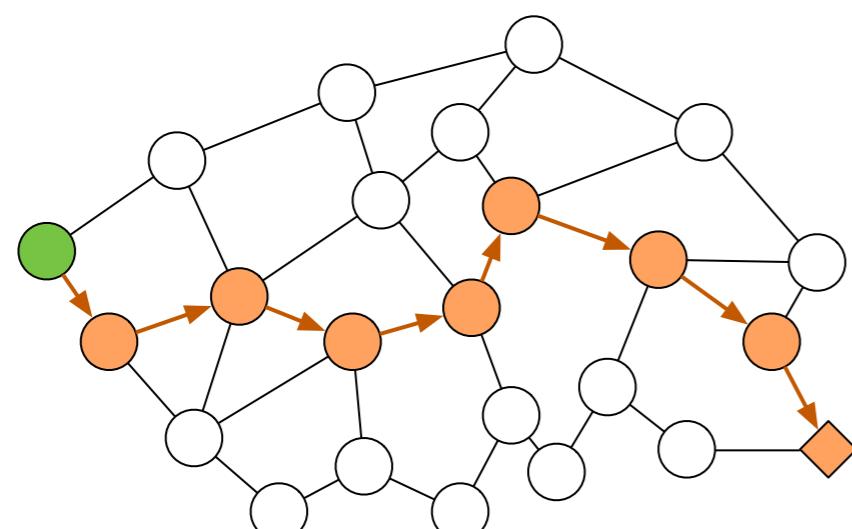
Budget maximal de 1000 \$



Recherche locale

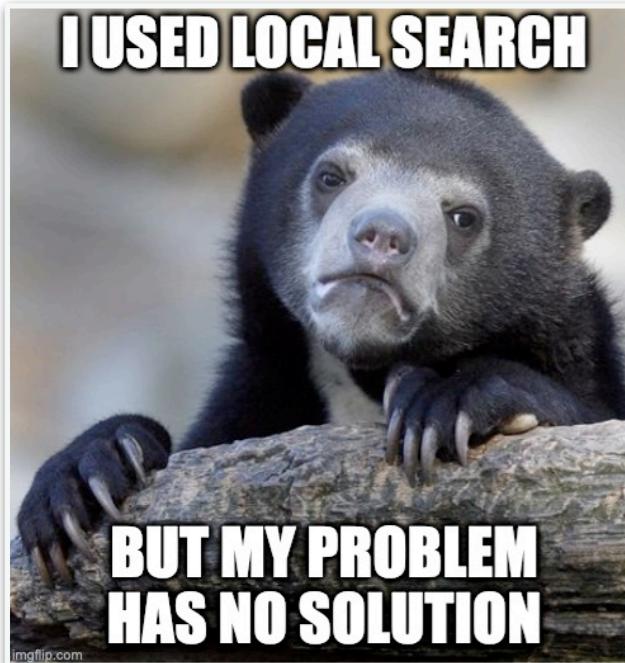
Principe: amélioration itérative d'une solution en sélectionne une nouvelle solution dans un voisinage

Améliorations indispensables: utilisation de mécanismes variés pour s'échapper des minima locaux



Limitations de la recherche locale

Limitation 1: absence de garanties



Propriété: l'espace de recherche n'est pas entièrement exploré

Conséquence: impossibilité de prouver qu'un problème est infaisable

Conséquence: impossibilité de prouver qu'une solution trouvée est optimale

Malgré son potentiel, on n'a aucune garantie sur nos résultats

On se retrouve bloqué si notre problème n'a aucune solution

Limitation 2: difficulté d'utilisation



Besoin: requiert l'implémentation d'un algorithme de recherche

Implication: choix de conception non triviaux (voisinage, météuristiche, etc.)

Implication: efficacité de l'implémentation (structures de données, langage, etc.)

Conséquence: souvent hors de portée des non-experts

Or, ces problèmes se produisent dans énormément de contextes différents !

L'implémentation d'une recherche locale efficace n'est pas une tâche aisée

Méthodes de résolution exhaustive intelligente

Retournons à notre idée d'utiliser une approche complète: la recherche exhaustive intelligente

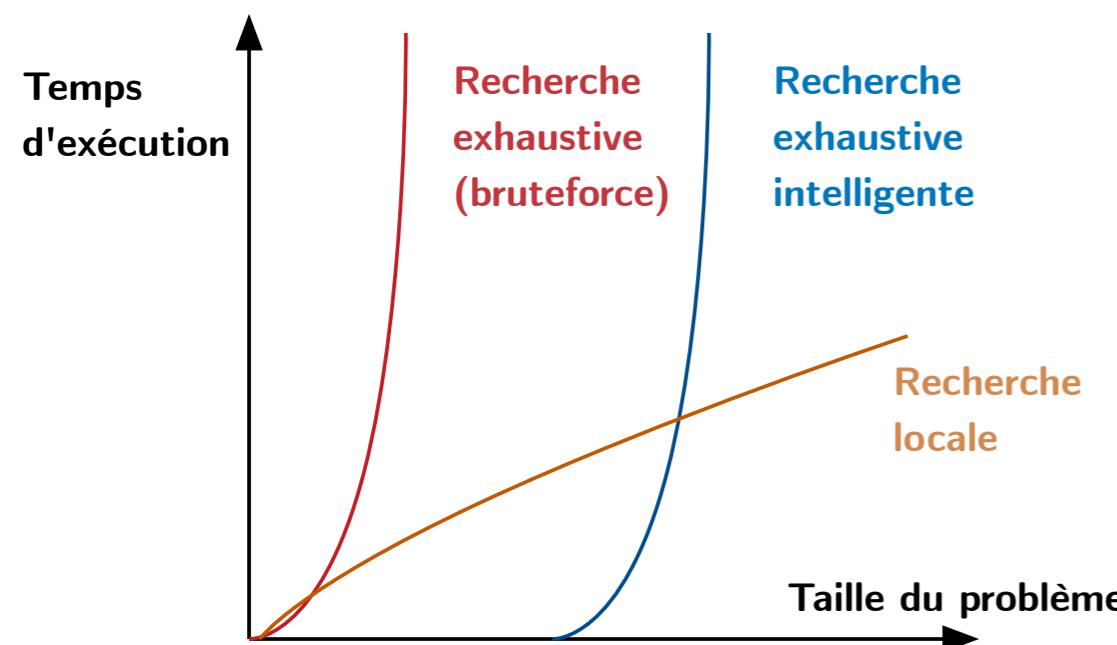
Principe 1: ne pas avoir peur d'un algorithme à complexité exponentielle dans le pire des cas

Principe 2: rajouter des mécanismes afin d'accélérer le processus de résolution

On a ainsi toujours une recherche exhaustive... mais qui intègre une forme d'intelligence

Utilisation d'heuristique: essayer d'abord les solutions les plus prometteuses

Intégration de raisonnements logiques: supprimer les solutions que l'on sait mauvaises à coup sûr



Difficulté: on a toujours une complexité exponentielle

Conséquence: la taille deviendra toujours problématique

Exemples d'algorithmes de résolution de ce type

Programmation en nombres entiers (MAGI - MTH6404)

Résolution SAT (module sur les agents logiques)

Programmation par contraintes (ce module)



Qu'en est-il de la facilité d'utilisation ?

Paradigme de programmation



Paradigme de programmation

Boîte à outils spécifique pour programmer des solutions à des problèmes par l'informatique

Programmation impérative: programmation par algorithmes (séquence d'instructions)

Programmation orientée-objet: utilisation d'objets comme briques logicielles

Programmation fonctionnelle: utilisation de fonctions comme briques logicielles

Philosophie d'un language de programmation

Objectif: rendre facile l'utilisation de certains paradigmes de programmation



C: programmation impérative, procédurale

Java: programmation impérative, orientée-objet, fonctionnelle (Java 8)

Python: programmation impérative, fonctionnelle, orientée-objet

Beaucoup de langages populaires sont impératifs, orientés-objet, et fonctionnels

Programmation déclarative

Objectif: l'utilisateur ne doit pas implémenter une solution à son problème, mais seulement le décrire



HTML: donne le rendu d'une page Web à partir d'une description de contenu

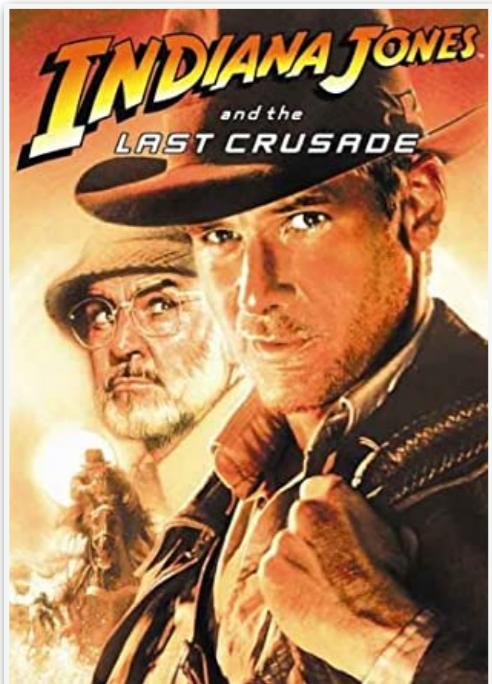
SQL: donne des informations dans une base de données à partir d'une requête

Opposition à la programmation impérative (description et non résolution)

A la poursuite du Saint-Graal



Notre rêve: existe t'il un language déclaratif pour résoudre un problème combinatoire ?



Le Saint-Graal de l'informatique

"Est-il possible qu'un ordinateur résolve un problème seulement sur base de sa description ?"

Opposition forte à une formulation impérative (façon de résoudre le problème)

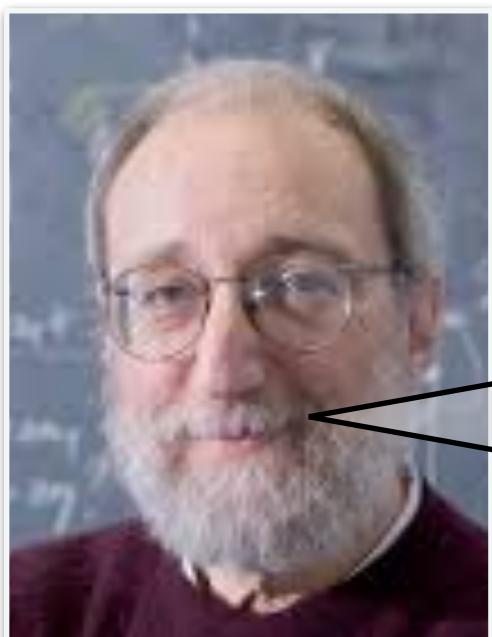
La recherche locale, telle que vue, suit un paradigme impératif

Programmation par contraintes

Paradigme de programmation dédié à la résolution de problèmes combinatoires

Interface déclarative pour l'utilisateur final (description du problème à résoudre)

Résolution interne basée sur de la recherche et des raisonnements logiques



"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"

<https://freuder.wordpress.com/>

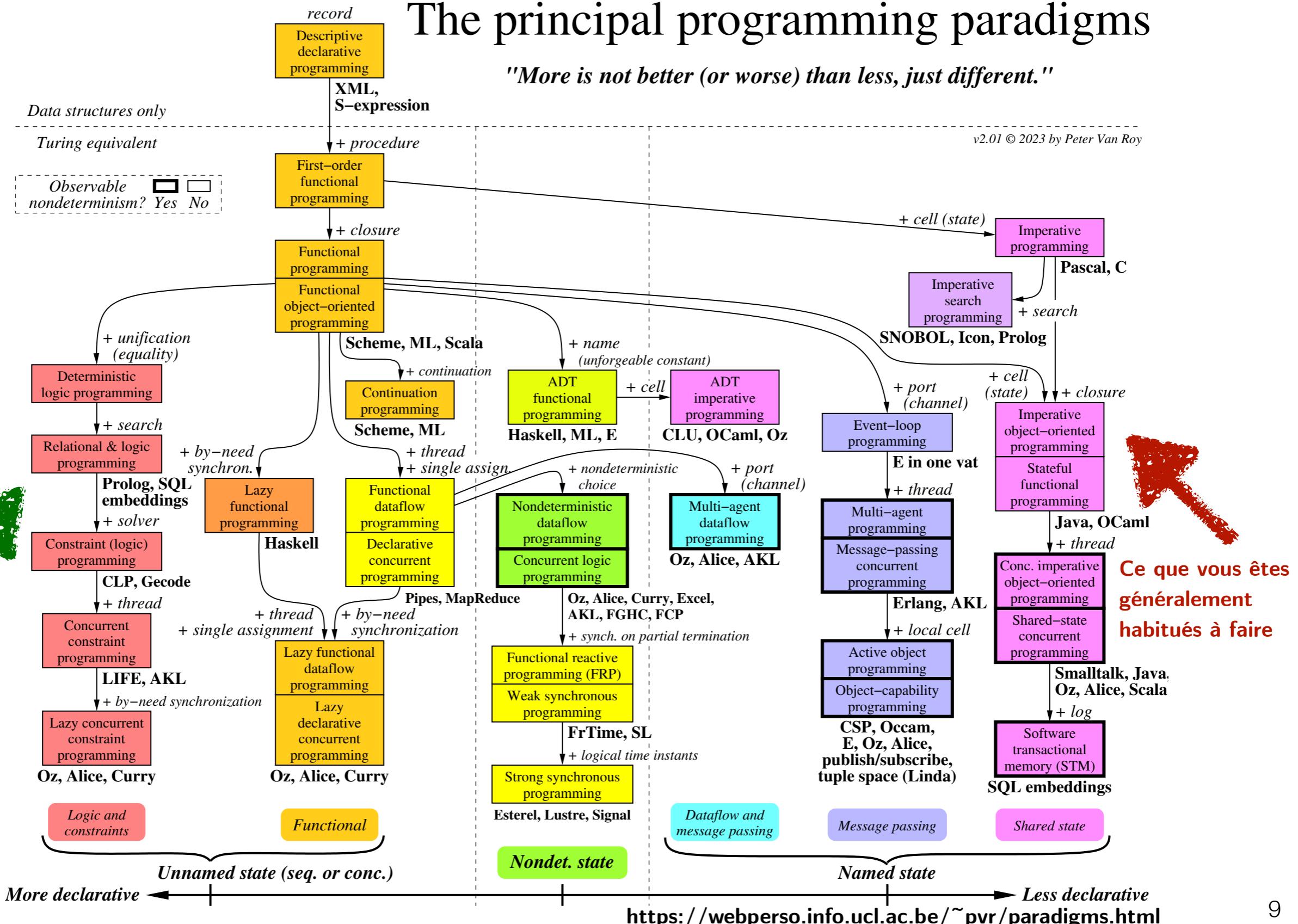
E. Freuder

Principaux paradigmes de programmation

The principal programming paradigms

"More is not better (or worse) than less, just different."

v2.01 © 2023 by Peter Van Roy



Programmation par contraintes (CP)

Programmation par contraintes (*constraint programming - CP*)

Nature: paradigme de programmation pour résoudre des problèmes combinatoires

Objectif: permettre à l'utilisateur de décrire facilement le problème à résoudre via un modèle

Fonctionnement: résolution basée sur de la **recherche** et sur des **raisons logiques**



Programmation par contraintes = **modèle** + **recherche** + **propagation**

Composant 1: construction d'un modèle

Objectif: agit comme une interface entre l'utilisateur et le solveur, pour qu'il puisse décrire son problème

L'utilisateur doit être capable de décrire son problème (responsabilité principale de l'utilisateur)

Composant 2: mécanisme de recherche

Objectif: réaliser une recherche exhaustive intelligente de solutions

N'implique que des faibles interactions avec l'utilisateur (idéalement aucune)

Composant 3: mécanisme de propagation

Objectif: réaliser des raisonnements logiques pour réduire l'espace de recherche

Processus quasi invisible pour l'utilisateur

Partie non déclarative, formant l'algorithme de résolution

Table des matières

Programmation par contraintes



1. Limitations et difficultés de la recherche locale



2. Concepts et principes fondamentaux de la programmation par contraintes

3. Formalisation des problèmes combinatoires

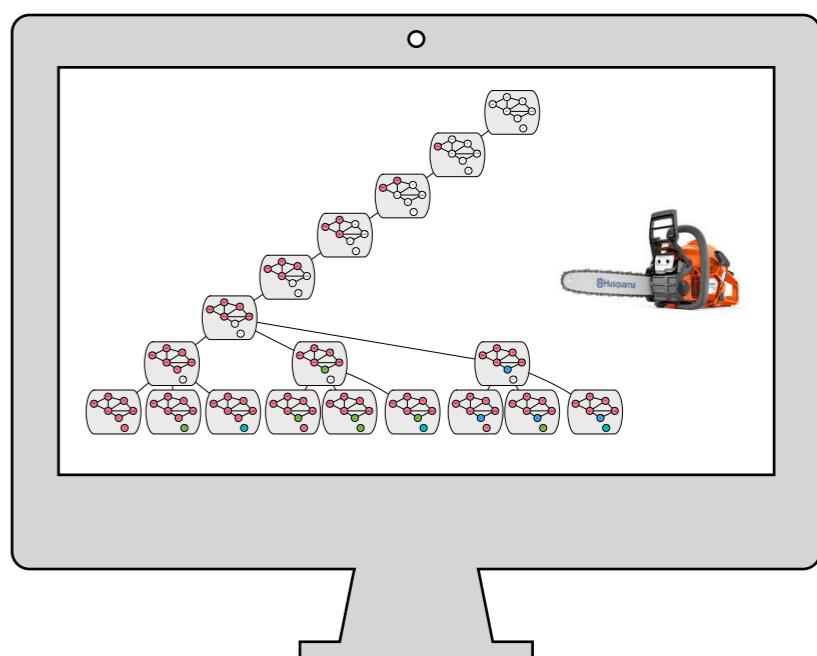
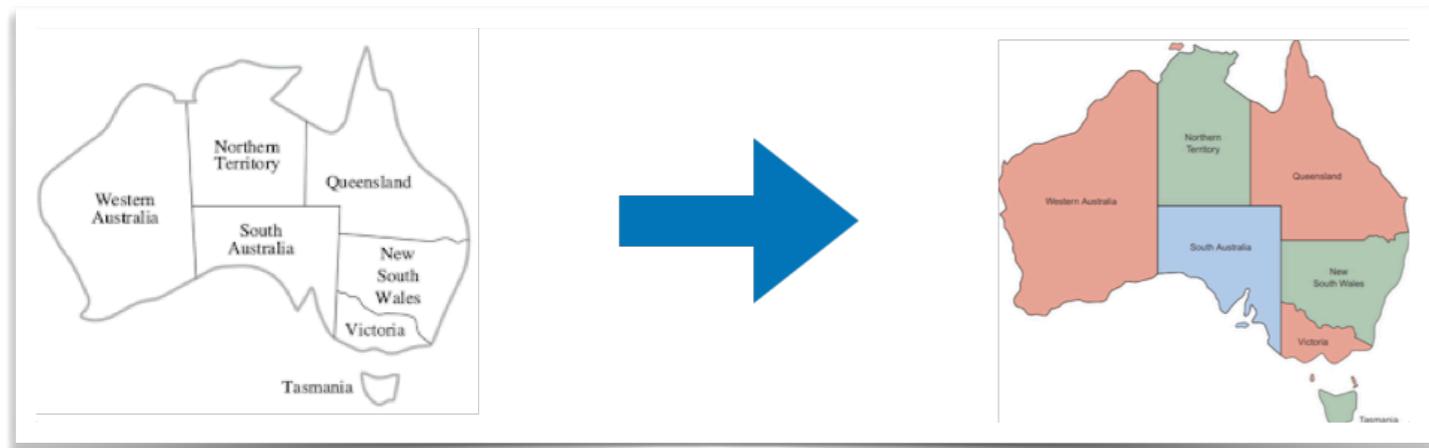
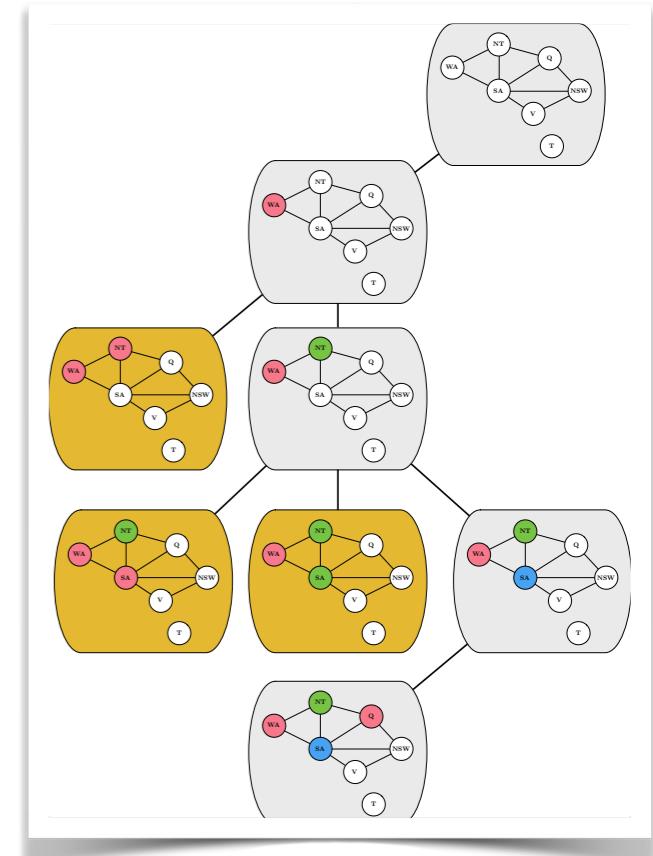
4. Modélisation en programmation par contraintes

5. Recherche avec retours-arrières (*backtracking search*)

6. Propagation de contraintes (*arc consistency*)

7. Algorithme du point fixe

8. Fonctionnement d'un solveur de programmation par contraintes



Modélisation de problèmes combinatoires

Regardons les éléments qui doivent être définis

Quelles sont nos décisions possibles ?

Quels sont nos choix ?

Quels sont les choix qui sont autorisés ?

Eventuellement, que souhaite t-on optimiser ?



Problème de satisfaction combinatoire (CSP)

- Problème défini par un ensemble de variables (X), des domaines discrets pour chaque variable (D), et un ensemble de contrainte (C)

$$\text{CSP} = \langle X, D, C \rangle$$

Résoudre un CSP revient à trouver une assignation de chaque variable qui satisfait toutes les contraintes du problème



Problème d'optimisation combinatoire (COP)

- Problème défini par un ensemble de variables (X), des domaines discrets pour chaque variable (D), un ensemble de contrainte (C), et une fonction objectif (O)

$$\text{COP} = \langle X, D, C, O \rangle$$

Résoudre un COP revient à trouver une assignation de chaque variable qui satisfait toutes les contraintes du problème et qui optimise la fonction objectif

Variables et domaines

Variable de décision

Définition: une décision qui doit être faite

Les choix possibles sont définis par le domaine de la variable

Domaine

Définition: un ensemble de valeurs possibles pour une variable

Domaine entier : $D(x) = \{1, 2, \dots, 9\}$

Domaine binaire : $D(x) = \{0, 1\}$

Domaine décimal : $D(x) = \{1.5, 1.9, 2.4\}$

Domaine de graphes : $D(x) = \{ \text{graphes} \}$

Et plein d'autres possibilités ! (en fait, n'importe quel domaine peut être imaginé)

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		

Quel chiffre mettre dans cette case ?

Choix dans une grille de Sudoku

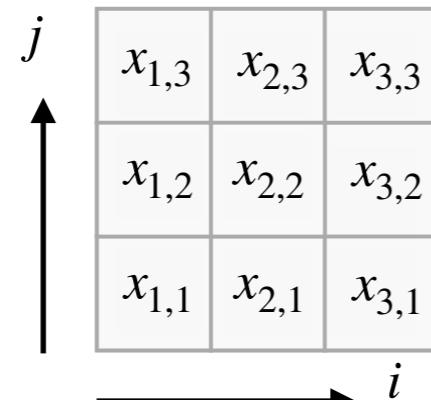
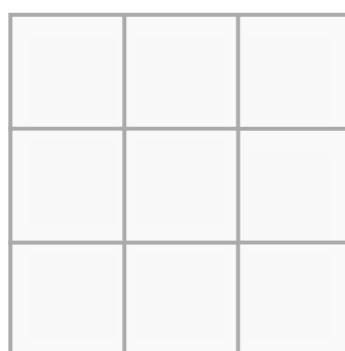
Choix d'un investissement ou non

Choix d'une quantité fractionnaire

Choix entre différentes molécules



Quel serait un bon choix de variables et de domaines pour le carré magique ?



$x_{i,j}$: chiffre dans la case (i, j)

Domaines : $\forall_{i,j} D(x_{i,j}) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Taille du problème : 9 variables, avec chacune 9 valeurs possibles

Recherche exhaustive : $9^9 = 387420489$ solutions à tester

Contraintes

Contraintes

Définition: restriction sur les valeurs possibles que peuvent prendre les variables

Objectif 1: permettre à l'utilisateur d'exprimer aisément un grand nombre de relations (*Saint-Graal*)

Objectif 2: réduire l'espace de recherche par des raisonnements logiques (propagation)

Objectif 3: détecter les solutions (éventuellement partielles) qui sont infaisables

Contraintes arithmétiques

$$3x_1 - x_2 = 2x_2$$

$$x_1 + 3x_2 \leq 5x_2$$

$$x_1 \neq 2x_2 - 3x_3$$

Contraintes logiques

$$\begin{aligned} & x_1 \vee x_2 \\ & ((x_1 = 5) \wedge (x_2 \leq 3)) \implies (x_3 = 2) \end{aligned}$$

Contraintes extensives

x_1	x_2	x_3	x_4
1	2	3	4
1	3	1	2
2	4	3	5
5	3	2	6
7	10	1	2

Seulement certaines configurations de valeurs sont autorisées

Contraintes globales

allDifferent(x_1, x_2, x_3, x_4)

les quatre variables doivent prendre une valeur différente

atMost($n, [x_1, x_2, x_3, x_4], v$)

Au plus n variables peuvent prendre la valeur v

increasing($[x_1, x_2, x_3, x_4]$)

les variables doivent prendre des valeurs croissantes



Force: un très large éventail de contraintes est disponible pour l'utilisateur

Catalogue de contraintes: plus de 400 contraintes différentes répertoriées

Extensibilité: il est également possible de construire de nouvelles contraintes

Une des tâches d'un concepteur de solveur de programmation par contraintes

<https://sofdem.github.io/gccat/>

Exemple du catalogue des contraintes globales

a

abs_value
all_equal_peak
all_min_dist
alldifferent_except_0
alldifferent_same_value
among_low_up
arith
atleast
atmost_nvalue

all_differ_from_at_least_k_pos
all_equal_peak_max
alldifferent
alldifferent_interval
allperm
among_modulo
arith_or
atleast_nvalue
atmost_nvector

all_differ_from_at_most_k_pos
all_equal_valley
alldifferent_between_sets
alldifferent_modulo
among
among_seq
arith_sliding
atleast_nvector

all_differ_from_exactly_k_pos
all_equal_valley_min
alldifferent_consecutive_values
alldifferent_on_intersection
among_diff_0
among_var
assign_and_counts
atmost

all_equal
all_incomparable
alldifferent_cst
alldifferent_partition
among_interval
and
assign_and_nvalues
atmost1

b

balance
balance_path
bin_packing

balance_cycle
balance_tree
bin_packing_capa

c

calendar
change_continuity
circuit_cluster
colored_matrix
common_modulo
cond_lex_greatereq
consecutive_groups_of_ones
counts
cumulative_convex
cutset
cyclic_change

cardinality_atleast
change_pair
circular_change
coloured_cumulativ
common_partition
cond_lex_less
consecutive_values
coveredby_sboxes
cumulative_product
cycle
cyclic_change_joke

5.12. alldifferent

DESCRIPTION LINKS GRAPH AUTOMATON

Origin	[Lauriere78]
Constraint	alldifferent(VARIABLES)
Synonyms	alldiff, alldistinct, distinct, bound_alldifferent, bound_alldiff, bound_distinct, rel.
Argument	VARIABLES collection(var – dvar)
Restriction	required(VARIABLES, var)
Purpose	Enforce all variables of the collection VARIABLES to take distinct values.
Example	(5, 1, 9, 3) The alldifferent constraint holds since all the values 5, 1, 9 and 3 are distinct.

<https://sofdem.github.io/gccat/gccat/Calldifferent.html>

Chaque page contient également d'autres détails sur la contrainte (algorithme, exemple, etc.)

Fonction objectif et modèle du carré magique

Fonction objectif

Définition: expression à maximiser ou minimiser pour un COP

$$\min \sum_{i=1}^n x_i \quad \max \sum_{i=1}^n x_i^2 - x_i^3$$

Modèle du carré magique

Objectif de satisfaction: remplir un carré 3×3 ($n \times n$) avec les nombres de 1 à 9 (n^2)

Contrainte 1: chaque case doit avoir un nombre différent

Contrainte 2: la somme de chaque colonne, rangée, et diagonale doit valoir 15 (T)

satisfy
subject to:

allDifferent($[x_{1,1}, \dots, x_{3,3}]$)

$$\sum_{j=1}^n x_{i,j} = T \quad \forall i \in \{1, \dots, n\}$$

$$\sum_{i=1}^n x_{i,j} = T \quad \forall j \in \{1, \dots, n\}$$

$$\sum_{i=1}^n x_{i,i} = T$$

$$\sum_{i=1}^n x_{i,n-i+1} = T$$

$$x_{i,j} \in \{1, \dots, n^2\} \quad \forall i, j \in \{1, \dots, n\}$$

MagicSquare

Toutes les valeurs sont différentes

Somme sur les colonnes vaut 15

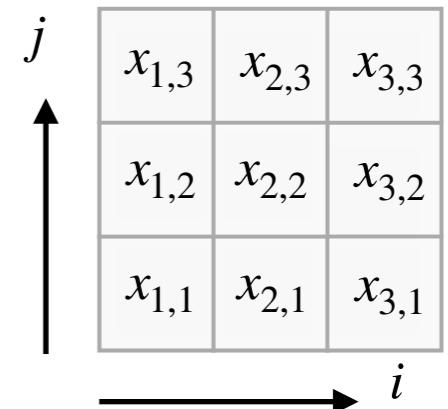
Somme sur les rangées

Somme de la diagonale montante

Somme de la diagonale descendante

On a un carré de cases à remplir

On considère une variable par case



Formulation déclarative

Formulation déclarative d'un problème combinatoire

satisfy

MagicSquare

subject to:

allDifferent($[x_{1,1}, \dots, x_{3,3}]$)

$$\sum_{j=1}^n x_{i,j} = T \quad \forall i \in \{1, \dots, n\}$$

$$\sum_{i=1}^n x_{i,j} = T \quad \forall j \in \{1, \dots, n\}$$

$$\sum_{i=1}^n x_{i,i} = T$$

$$\sum_{i=1}^n x_{i,n-i+1} = T$$

$$x_{i,j} \in \{1, \dots, n^2\} \quad \forall i, j \in \{1, \dots, n\}$$



WHEN YOU FIND

THE HOLY GRAIL

makeameme.org



```
int: n = 3;
int: tot = n * (n^2 + 1) div 2;

array[1..n,1..n] of var 1..n*n: x;

constraint all_different(x);

constraint forall(k in 1..n)(sum(i in 1..n)(x[k,i]) == tot);
constraint forall(k in 1..n)(sum(i in 1..n)(x[i,k]) == tot);
constraint sum(i in 1..n) (x[i,i]) = tot;
constraint sum(i in 1..n) (x[i,n-i+1]) = tot;

solve satisfy;
```

```
Compiling magic_square.mzn
Running magic_square.mzn
total: 15
solution:
6 1 8
7 5 3
2 9 4
-----
Finished in 443msec
```

Minizinc

Principe: language de modélisation pour des problèmes combinatoires

Caractéristiques: open-source, populaire, et compatible avec plusieurs solveurs

Devoir 2: une partie consiste à modéliser et résoudre des problèmes combinatoires avec ce langage

Minizinc



<https://www.minizinc.org>

Minizinc

Nature: langage de modélisation déclarative pour des problèmes combinatoires

Fonction: agir comme une interface avec différent solveurs

Ce n'est donc pas un algorithme de résolution, mais juste une interface

Objectif: permettre à l'utilisateur de se focaliser sur la modélisation

Force: riche catalogue de contraintes (> 100)

Force: compatible avec plusieurs solveurs (Gecode, Chuffed, etc.)

Force: permet de configurer le processus de résolution

Plusieurs ressources sont disponibles sur Moodle pour vous aiguiller dans l'apprentissage



MOOC introductif

Ressource gratuite pour se familiariser avec MiniZinc

De manière plus large, apprend à modéliser et résoudre des CSPs/COPs

Type de pédagogie: story-based learning et résolution de problèmes

<https://www.coursera.org/lean/basic-modeling>

Table des matières

Programmation par contraintes



1. Limitations et difficultés de la recherche locale



2. Concepts et principes fondamentaux de la programmation par contraintes



3. Formalisation des problèmes combinatoires



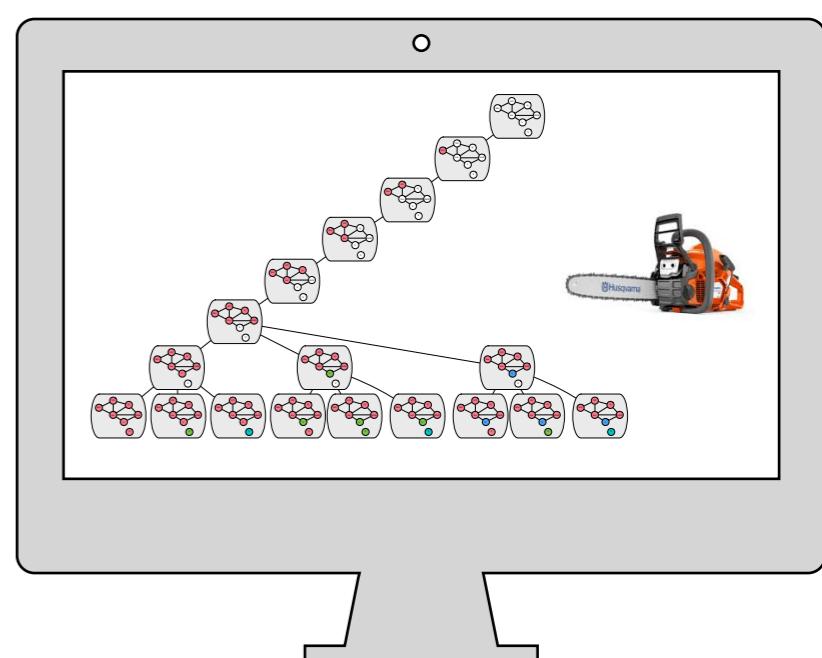
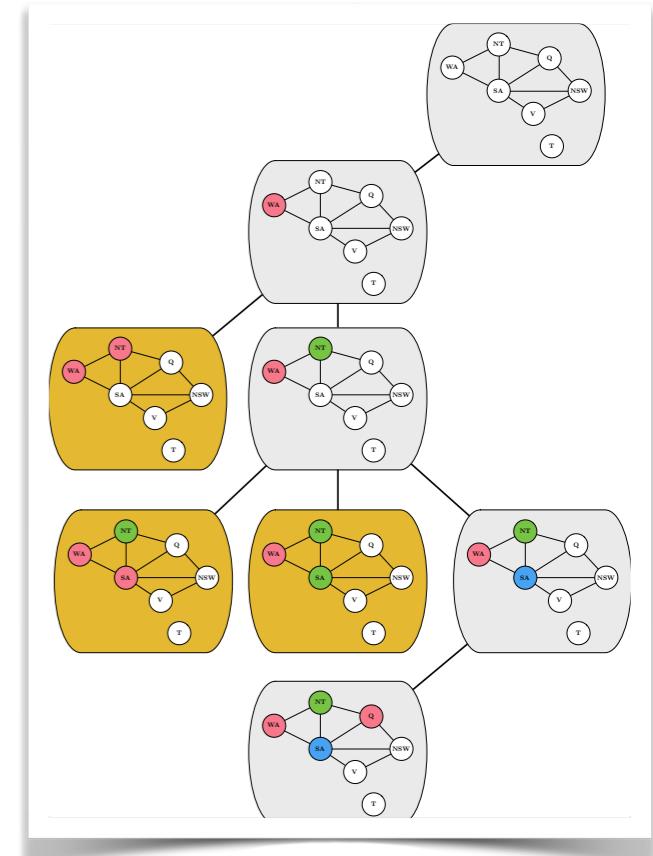
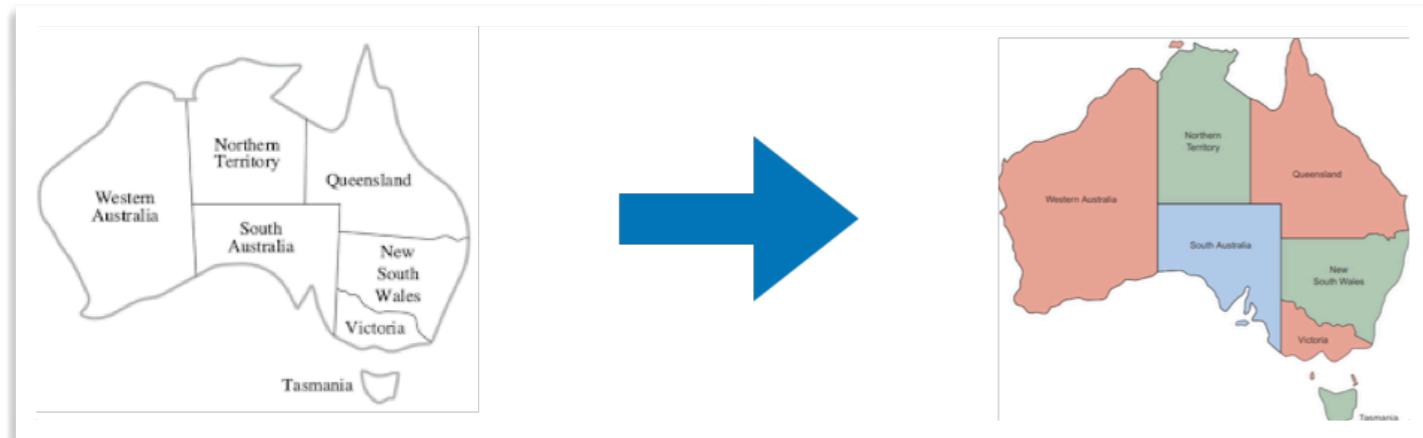
4. Modélisation en programmation par contraintes

5. Recherche avec retours-arrières (*backtracking search*)

6. Propagation de contraintes (*arc consistency*)

7. Algorithme du point fixe

8. Fonctionnement d'un solveur de programmation par contraintes



Algorithme de résolution

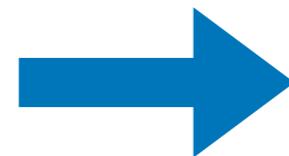
```
int: n = 3;
int: tot = n * (n^2 + 1) div 2;

array[1..n,1..n] of var 1..n*n: x;

constraint all_different(x);

constraint forall(k in 1..n)(sum(i in 1..n)(x[k,i]) == tot);
constraint forall(k in 1..n)(sum(i in 1..n)(x[i,k]) == tot);
constraint sum(i in 1..n) (x[i,i]) = tot;
constraint sum(i in 1..n) (x[i,n-i+1]) = tot;

solve satisfy;
```



```
Compiling magic_square.mzn
Running magic_square.mzn
total: 15
solution:
6 1 8
7 5 3
2 9 4
-----
Finished in 443msec
```



Programmation par contraintes = modèle + recherche + propagation

Solveur de programmation par contraintes

Philosophie: cacher l'algorithme de résolution pour l'utilisateur qui veut résoudre le problème combinatoire

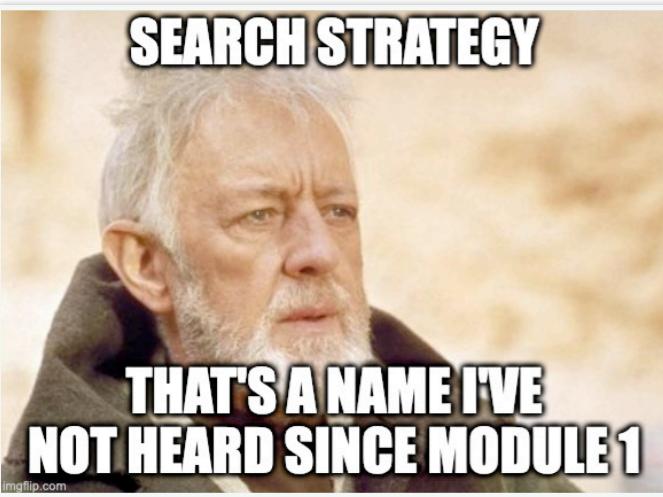
Fonctionnement: Les solveurs CP modernes sont basés à la fois sur de la **recherche**, et de la **propagation**

Conception: par des chercheurs et ingénieurs (informatique, logiciel, ou mathématique)

Regardons comment ces deux aspects sont combinés pour former un solveur

Phase de recherche

SEARCH STRATEGY



Peut-on réutiliser les idées de nos méthodes de recherche du module 1 ?

Bonne nouvelle: les principes généraux de nos idées peuvent être ré-utilisés

Difficulté: il y a quelques adaptations pour rendre la recherche efficace

Première étape: formaliser la situation comme un problème de recherche

Formalisation du problème de recherche

Principe: considérer chaque assignation partielle de l'ensemble des variables comme états

Etats: chaque combinaison d'assignations variable/valeur

Etat initial: l'assignation vide (aucune variable n'a de valeur assignée)

Etat final: toutes les variables ont une valeur respectant les contraintes

Ensemble des actions: assigner à une variable spécifique une valeur de son domaine

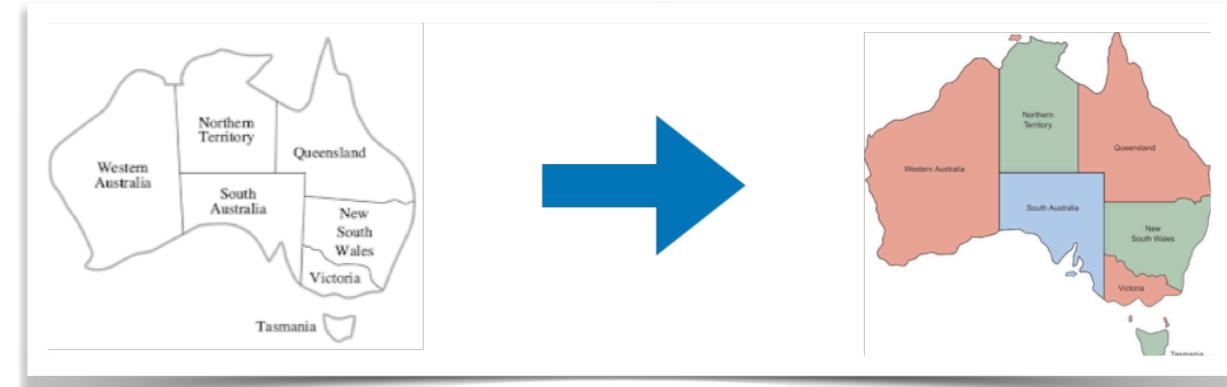
Transition: réaliser une assignation

Coût: l'impact sur la fonction objectif de l'assignation

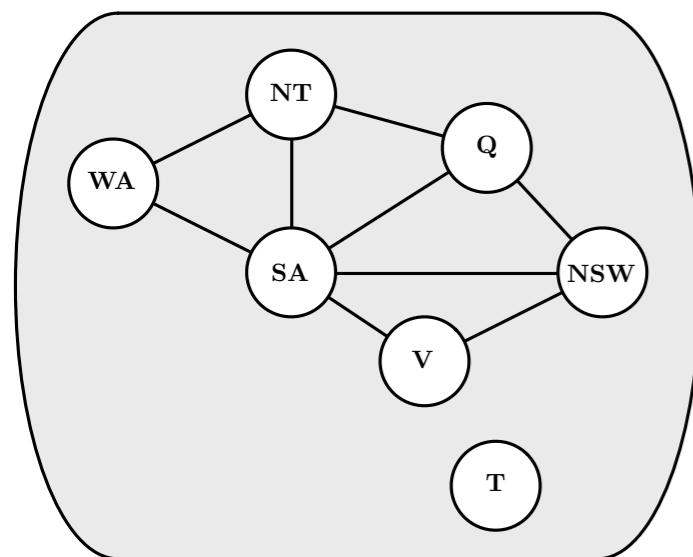
Exemple: coloriage de carte

Objectif: colorier une carte avec 3 couleurs

Contrainte: deux pays frontaliers ne peuvent pas avoir la même couleur



Exemple: coloriage de carte



Modélisation du CSP: besoin de définir nos entités précédentes

Ensemble des régions : $v \in V (\{WA, NT, SA, Q, NSW, V, T\})$

Ensemble des frontières entre deux régions : $(u, v) \in E$

Variables indiquant la couleur d'une région : x_v

Satisfy

Subject to $x_u \neq x_v \quad \forall (u, v) \in E$

$x_v \in \{\text{red, green, blue}\} \quad \forall v \in V$

Taille du problème : $3^7 = 2187$ possibilités



Quelle stratégie de recherche vous semble la plus adaptée ?

Branche de l'arbre: assignation d'une valeur à une variable

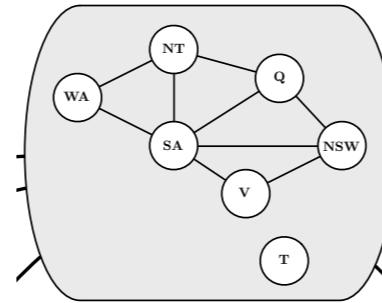
Solution: assignation complète des variables

Ces états correspondent à des noeuds feuilles d'un arbre de recherche

Caractéristique: la profondeur de l'arbre est bornée par le nombre de variables

Une recherche en profondeur (DFS) semble être un bon point de départ

Coloriage de carte: visualisation de l'arbre de recherche



? Combien de noeuds feuilles sont présents avec cette formulation ?

Observation: on a n variables ayant chacune un domaine de d valeurs

Premier niveau: on considère tous les couples variables/valeurs $\rightarrow n \times d$

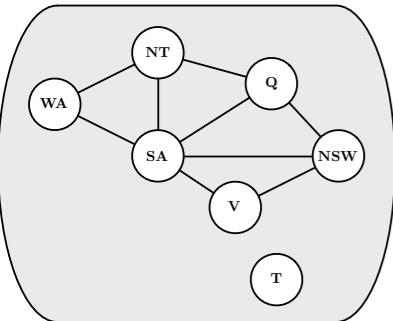
Deuxième niveau: on a une variable de moins à considérer $\rightarrow (n - 1) \times d$ par noeud parent

Assignation complète: lorsque les n variables ont chacune une valeur

Nombre de feuilles: $(n \times d) \times ((n - 1) \times d) \times ((n - 2) \times d) \times \dots \times d = n!d^n$

Réduction de l'espace de recherche

Un problème avec d^n assignations complètes possibles donne un espace de recherche avec $n!d^n$ feuilles



On a n variables avec d valeurs possibles

2187 colorages complets : 11022480 feuilles



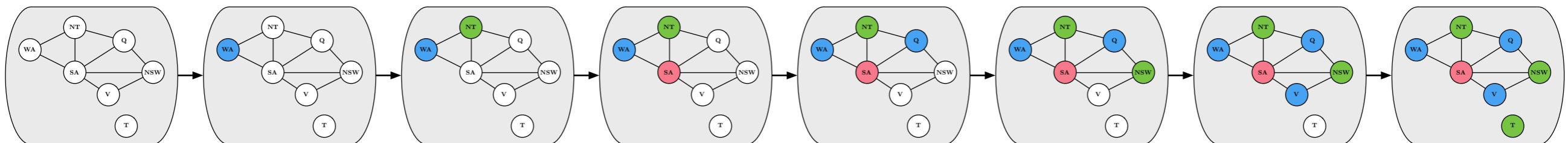
Peut-on réduire l'espace de recherche ?

Bonne nouvelle: on peut exploiter une propriété des problèmes combinatoires



Commutativité des problèmes combinatoires

L'ordre des actions (assignation des variables) n'a pas d'influence sur la solution finale obtenue



$x_{WA} = \text{blue}$

$x_{NT} = \text{green}$

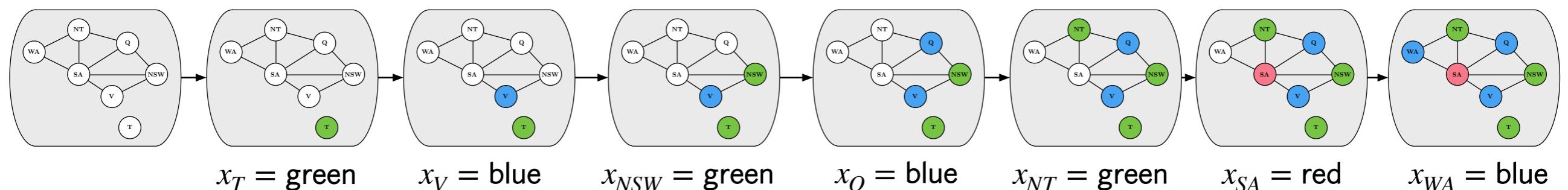
$x_{SA} = \text{red}$

$x_Q = \text{blue}$

$x_{NSW} = \text{green}$

$x_V = \text{blue}$

$x_T = \text{green}$



$x_T = \text{green}$

$x_V = \text{blue}$

$x_{NSW} = \text{green}$

$x_Q = \text{blue}$

$x_{NT} = \text{green}$

$x_{SA} = \text{red}$

$x_{WA} = \text{blue}$

Observation: quelque soit l'ordre des assignations, la solution finale obtenue est la même

Conséquence directe: on ne peut considérer qu'une seule variable par niveau

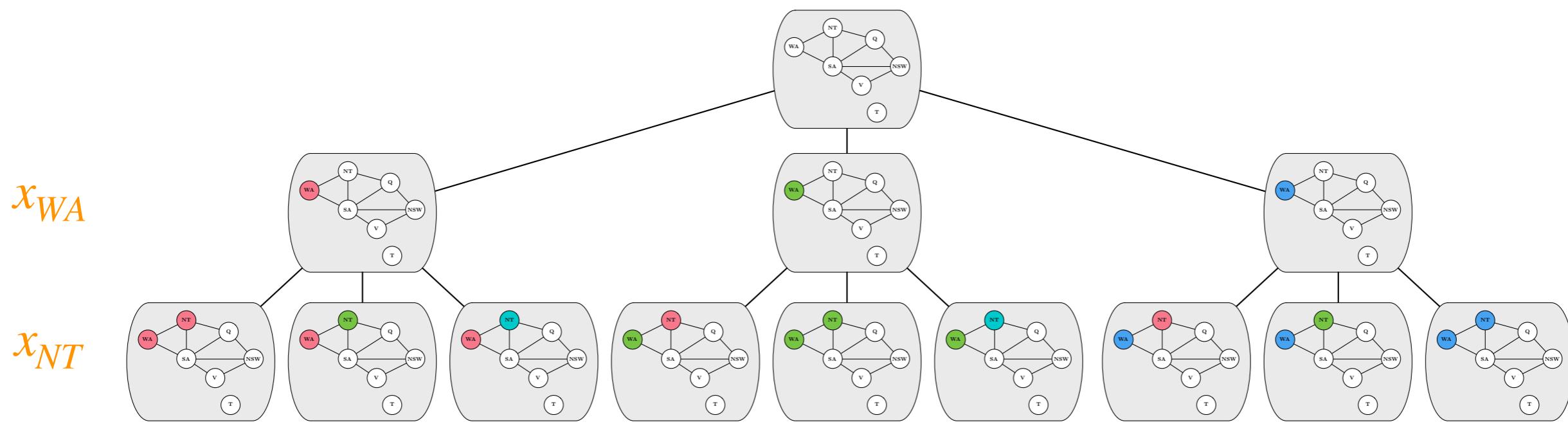
Amélioration 1: considérer une variable par niveau

Formalisation améliorée du problème de recherche

Problème de recherche: le même qu'avant, sauf qu'on ne va considérer qu'une seule variable par niveau

Principe: on va fixer un ordre des variables

Ordre des variables : $x_{WA} \rightarrow x_{NT} \rightarrow x_{SA} \rightarrow x_Q \rightarrow x_{NSW} \rightarrow x_V \rightarrow x_T$



Combien de noeuds feuilles sont présents avec cette formulation ?

Observation: on a n variables et d choix par niveau

Remarque: n donne la profondeur de l'arbre, et d correspond au facteur de branchement

Nombre de feuilles: d^n (identique au nombre de colorages possibles)

Exploration de l'arbre - recherche en profondeur

Notre prochaine étape est d'explorer cet arbre de recherche

Situation actuelle: on a un arbre de recherche où chaque solution potentielle correspond à un noeud feuille

Intuition: on veut tester pouvoir tester rapidement les noeuds feuilles

Idée 1: la recherche en profondeur permet cela, avec un bon contrôle sur la consommation mémoire



Que pensez-vous de cette idée ?

x_{WA}

x_{NT}

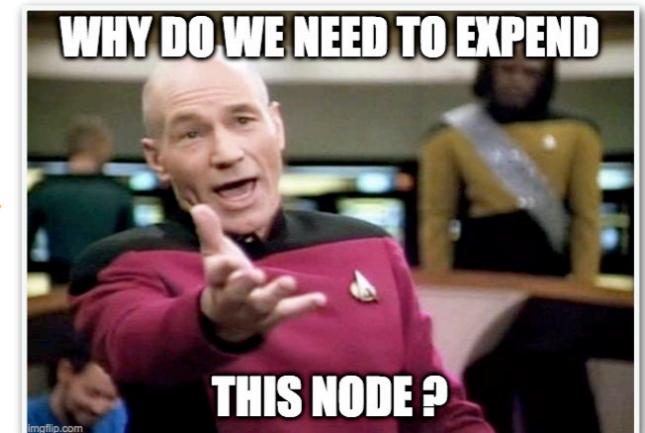
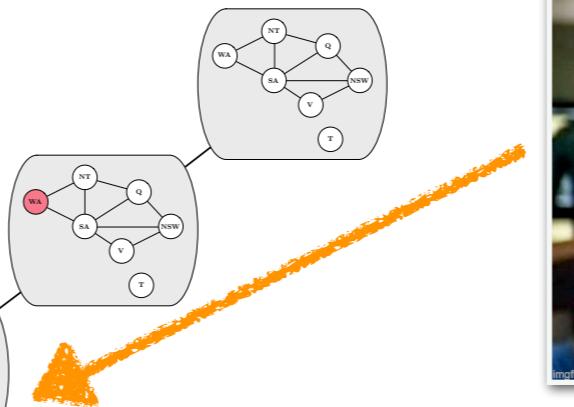
x_{SA}

x_Q

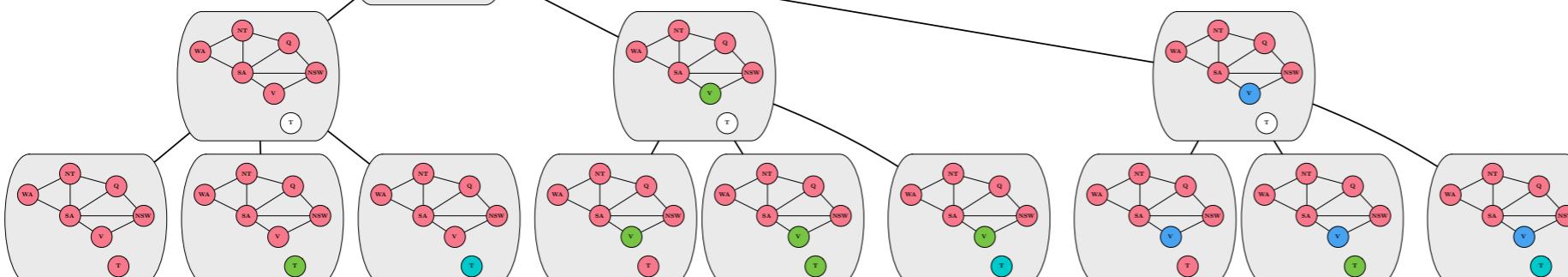
x_{NSW}

x_V

x_T



- (1) Un conflit apparent a été généré tôt dans l'arbre
- (2) Enormément de noeuds seront explorés avant que ce conflit ne soit enlevé
- (3) Or, on sait qu'aucune solution faisable ne sera possible tant qu'il y a un conflit



Amélioration 2: détection des conflits

Notre recherche en profondeur simple est extrêmement sensible aux conflits arrivant tôt dans l'arbre

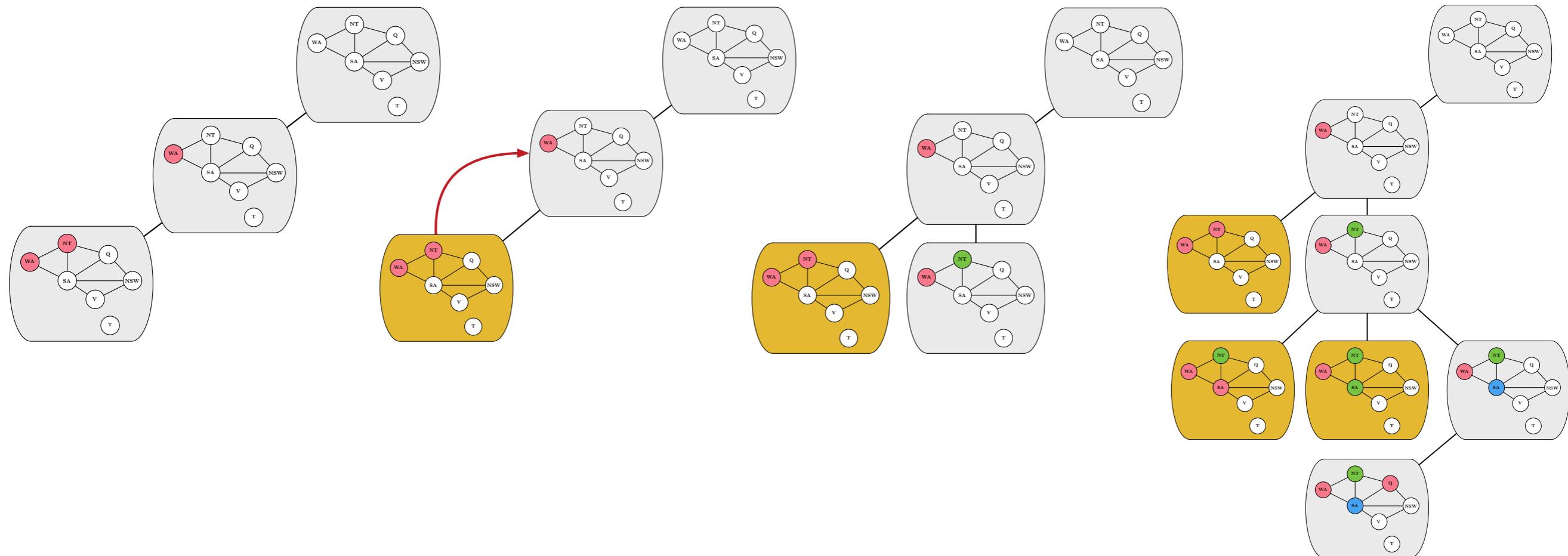
Conséquence: énormément de noeuds seront explorés avant qu'une alternative viable soit testée



Avez-vous une idée pour empêcher ce comportement ?

Observation: il n'y a aucun intérêt d'étendre un état qui contient déjà un conflit

Nouvelle idée: faire un retour en arrière sur le noeud parent dans l'arbre dès qu'un conflit est détecté



Cette stratégie est connue sous le nom de recherche à retours-arrières

Recherche avec retours-arrières (*backtracking search*)



Recherche avec retours-arrières (*backtracking search*)

Recherche en profondeur où on remonte au noeud parent dès qu'un conflit est détecté

Note: un conflit se produit lorsqu'il y a au moins une contrainte du problème qui n'est pas respectée

BacktrackingSearch(P) :

```
 $\langle X, D, C \rangle = P$ 
 $s = \{\}$ 
 $L = \text{LIFO}()$ 
push( $L, s$ )
while  $L \neq \emptyset$  :
     $s = \text{pop}(L)$ 
    if assignmentCompleted( $s$ ) : return  $s$ 
     $x = \text{selectUnassignedVariable}(s, X)$ 
    for each  $v \in D(x)$  :
         $s' = s \cup \{x \leftarrow v\}$ 
        if feasible( $s', C$ ) : push( $L, s'$ )
return no solution
```

Un problème correspond à un CSP

Etat initial: aucune assignation variable/valeur n'est faite

Structure *last-in first-out* (LIFO) similaire à un DFS

Etat final: si l'assignation est complète, on retourne la solution trouvée

Amélioration 1: on sélectionne une variable qui n'est pas encore assignée

On branche sur toutes les valeurs possibles du domaine de la variable choisie

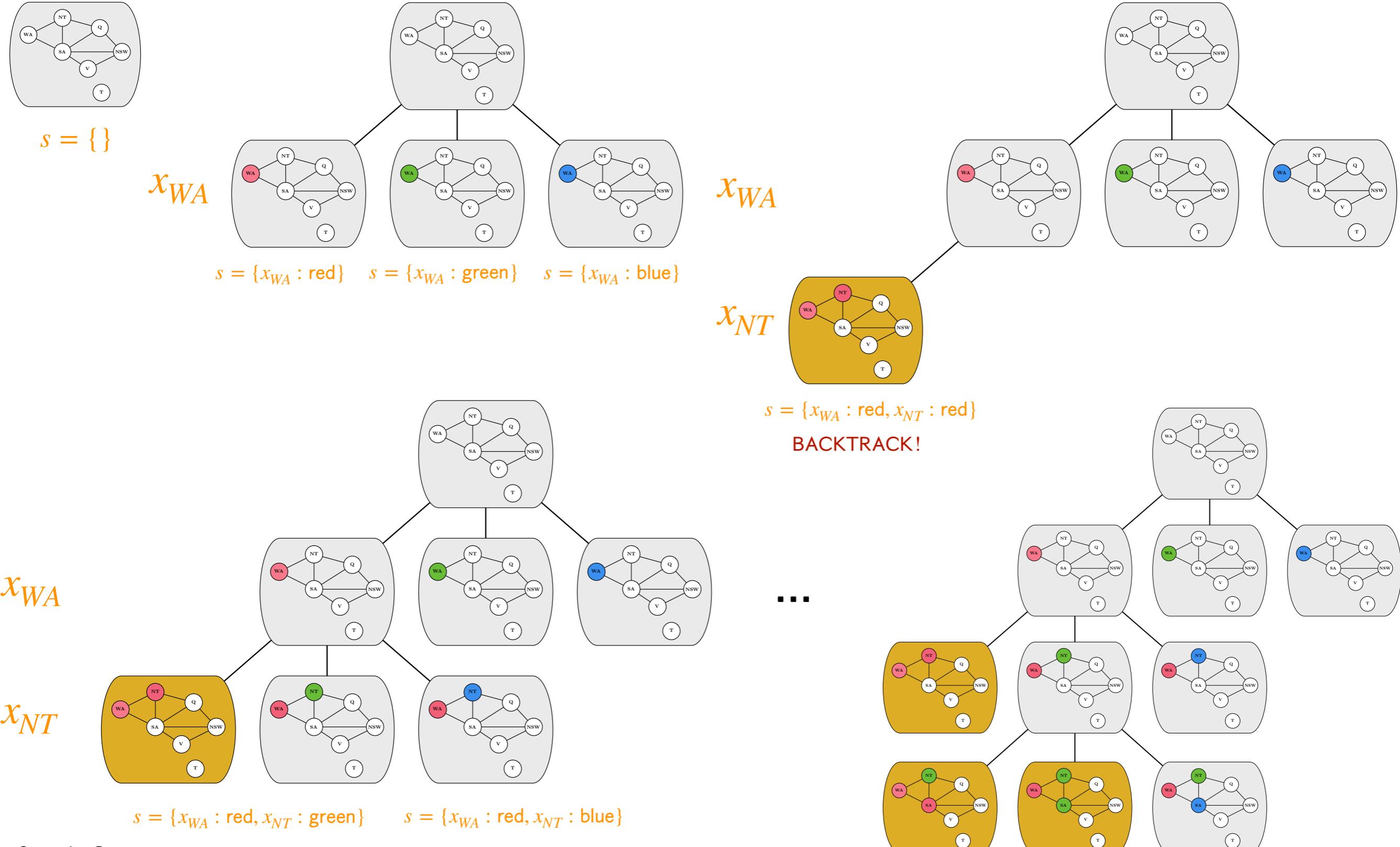
On ajoute l'assignation considérée à la solution partielle actuelle

Amélioration 2: on ne branche QUE si l'assignation est valide (active le backtracking)

Note: cet algorithme est essentiellement un DFS, on intègre juste les 2 améliorations précédentes

Note: pour un COP, on ne peut pas s'arrêter dès qu'on a une solution car on souhaite trouver la meilleure

Recherche avec retours-arrières (*backtracking search*)



Phase de recherche en programmation par contraintes

Programmation par contraintes = modèle + recherche + propagation

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

<https://en.wikipedia.org/wiki/Backtracking>

Point de départ: recherche basée sur une stratégie DFS

Amélioration 1: exploitation de la commutativité pour réduire l'espace de recherche

Amélioration 2: retour en arrière dès qu'un conflit a été détecté

Résultat: recherche à retours-arrières

Aller plus loin

Economie de mémoire: n'étendre qu'un successeur au lieu de tous

$\mathcal{O}(bm) \rightarrow \mathcal{O}(m)$: avec m le nombre de variables et b le nombre de valeurs

Preuve : raisonnement similaire à celui fait pour la complexité spatiale du DFS

Ordre des variables: influence sur l'efficacité de la recherche

Ordre des valeurs: influence sur l'efficacité de la recherche

Amélioration 1: utilisation d'heuristiques pour déterminer de bons ordres

Lecture complémentaire disponible sur Moodle

Amélioration 2: raisonnements logiques pour réduire l'espace de recherche

Table des matières

Programmation par contraintes



1. Limitations et difficultés de la recherche locale



2. Concepts et principes fondamentaux de la programmation par contraintes



3. Formalisation des problèmes combinatoires



4. Modélisation en programmation par contraintes

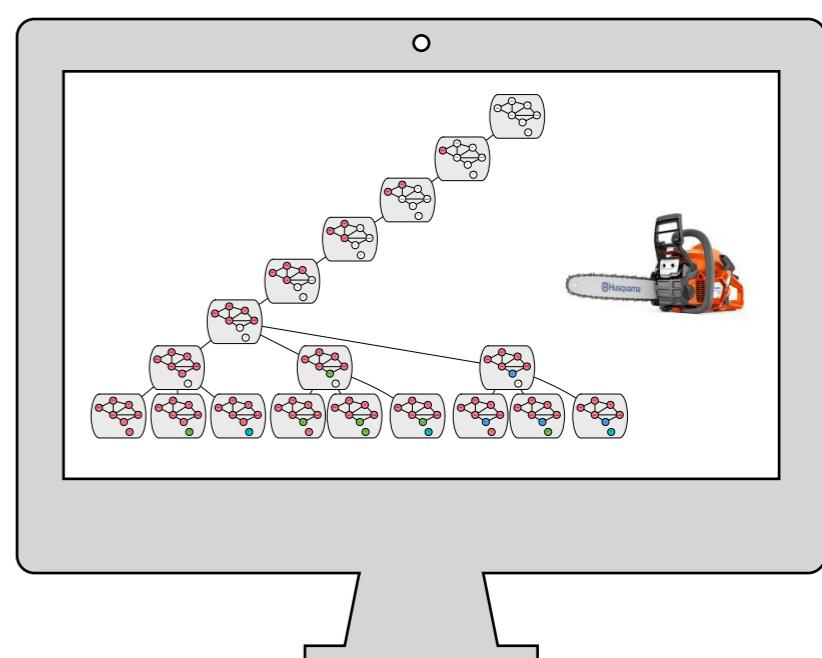
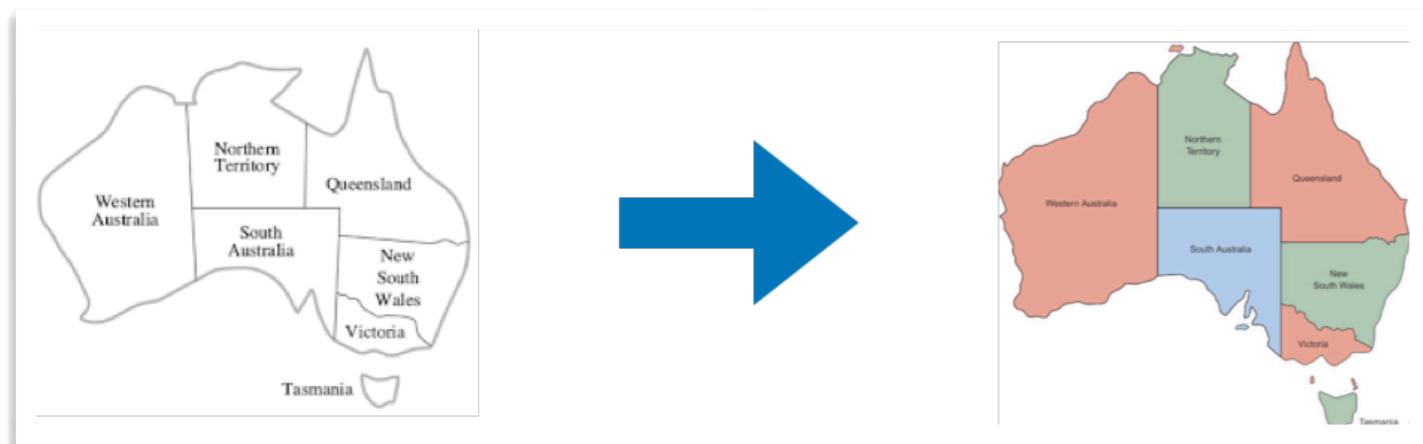
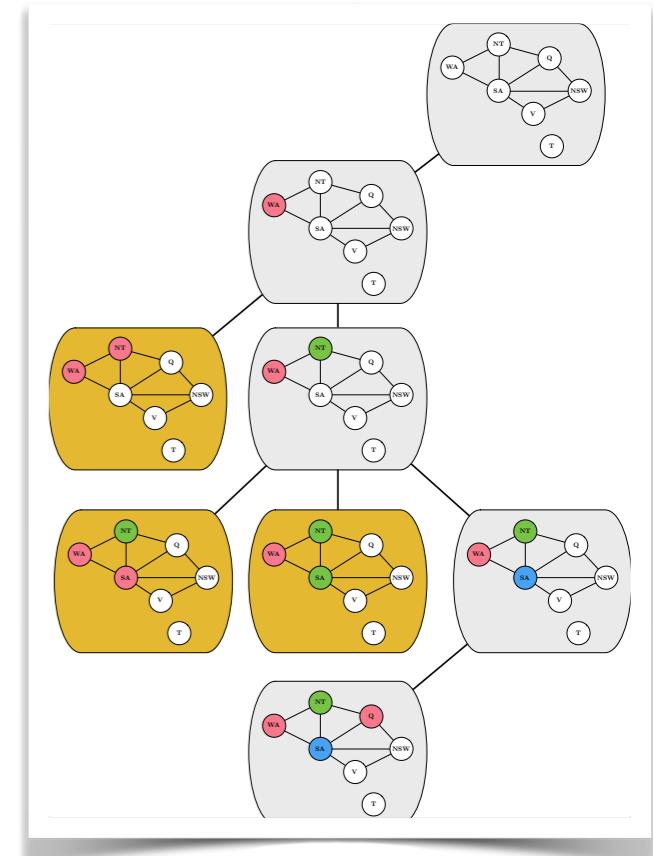


5. Recherche avec retours-arrières (*backtracking search*)

6. Propagation de contraintes (*arc consistency*)

7. Algorithme du point fixe

8. Fonctionnement d'un solveur de programmation par contraintes

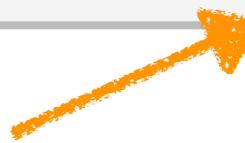


Phase de recherche en programmation par contraintes



Peut-on améliorer notre processus de résolution ?

Programmation par contraintes = modèle + recherche + propagation



Raisonnements logiques

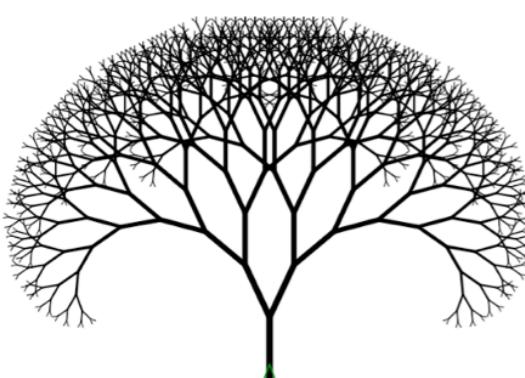
Responsabilité des contraintes

- (1) Fournir des facilités de modélisation pour l'utilisateur (*modeling support*)
- (2) Permettre de détecter des conflits pour un retour-arrière (*feasibility checking*)
- (3) Réduire le domaine des variables en enlevant des valeurs que l'on sait impossible (*propagation*)



Propagation de contraintes (*constraint propagation*)

Réduction de la taille des domaines via des raisonnements logiques appliqués sur les contraintes



Importance: un des mécanismes principaux d'un solveur CP

Impact: permet de réduire drastiquement l'espace de recherche

Mécanisme complémentaire à une stratégie de recherche

Propagation de contraintes: exemple introductif

Satisfy

Subject to $x \geq 3 + 2y$

$$x \in \{1, \dots, 10\}$$

$$y \in \{1, \dots, 10\}$$

Exemple: simple CSP avec une contrainte d'inégalité

Taille du problème : $|D(x)| \times |D(y)| = 100$ combinaisons variables/valeurs



Peut-on réduire la taille des domaines en exploitant la contrainte ?

Principe: analysons quelle est la meilleure situation si on veut satisfaire l'inégalité

(1) Point de vue de la variable x

Situation la moins restrictive: avoir le plus petit y possible (inégalité '*plus grand*')

$$x \geq 3 + 2y \rightarrow x \geq 5, \text{ avec } y = 1$$

Certitude: x ne peut pas avoir une valeur en dessous de 5

(2) Point de vue de la variable y

Situation la moins restrictive: avoir le plus grand x possible (inégalité '*plus petit*')

$$y \leq \frac{x - 3}{2} \rightarrow y \leq 3.5, \text{ avec } x = 10$$

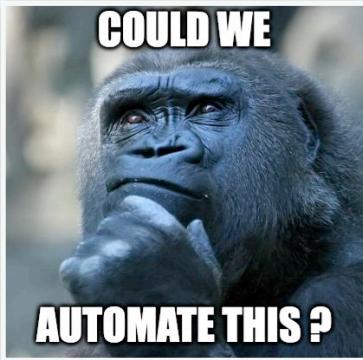
Certitude: y ne peut pas avoir une valeur au dessus de 3.5

Domaine de x après la propagation : $\{5, \dots, 10\}$

Domaine de y après la propagation : $\{1, 2, 3\}$

On n'a plus que 18 combinaisons valides, sans aucune recherche effectuée

Cohérence d'arc



Bonne nouvelle: oui! il existe plusieurs méthodes, ayant chacune leurs forces et faiblesses

Mauvaise nouvelle: cela implique qu'il y a de nouveaux algorithmes à implémenter

Mécanisme central: principe de la **cohérence d'arc** (*arc consistency*)



Cohérence/consistance d'arc (*arc consistency*)

Soit une **contrainte binaire**, qui implique deux variables x et y

La variable x est ***arc-consistent*** avec y , si pour chaque valeur de son domaine, il existe au moins une valeur dans le domaine de y permettant de satisfaire la contrainte

Intuition: chaque valeur de x doit avoir une valeur de y permettant de satisfaire la contrainte

Exemple: simple CSP avec une contrainte binaire quelconque

Satisfy

Subject to $y = x^2$

$$x \in \{0,1,2,3\}$$

$$y \in \{0,1,4,7,9\}$$



Est-ce que x est ***arc-consistent*** avec y selon la contrainte ?

Oui: chaque valeur de x a un valeur support dans y

Supports: $\langle x = 0 \rightarrow y = 0 \rangle, \langle x = 1 \rightarrow y = 1 \rangle, \langle x = 2 \rightarrow y = 4 \rangle, \langle x = 3 \rightarrow y = 9 \rangle$



Est-ce que y est ***arc-consistent*** avec x ?

Non: la valeur 7 n'a aucun support dans x

Support manquant: $\langle y = 7 \rightarrow \perp \rangle$

Conclusion: on peut donc retirer sûrement la valeur 7 de y

Propagateur de contraintes



Propagateur de contraintes (*propagator*)

Algorithme effectuant la propagation pour une contrainte (ou un type de contrainte)

En pratique: un propagateur doit être implémenté pour chaque contrainte présente dans un solveur CP

Note: souvent, plusieurs propagateurs sont disponibles par contrainte et diffèrent par deux propriétés

Force de propagation: qualité des raisonnements logiques pour réduire les domaines

Vitesse de propagation: efficacité temporelle des raisonnements logiques

Exemple: regardons un propagateur dédié aux contraintes binaires pour obtenir la cohérence d'arc

PropagationAC(x, y, c) :

for each $v_x \in D(x)$:

if no value v_y from $D(y)$ allows c with v_x to be satisfied :

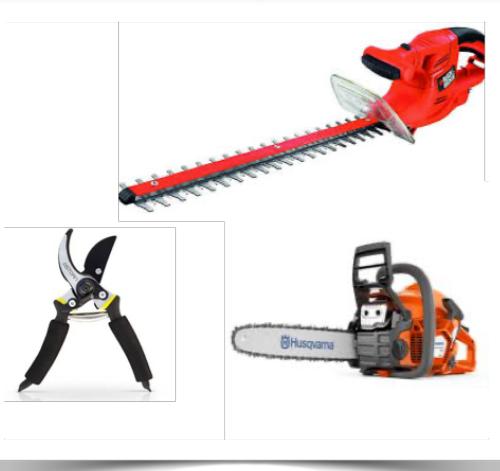
$$D(x) = D(x) \setminus v_x$$

Objectif: propage la contrainte binaire $c(x,y)$ pour la variable x

Boucle: on parcourt toutes les valeurs du domaine de x

Condition: si une valeur de x n'est pas supportée par une valeur de y

Résultat: alors on l'enlève du domaine de x (*pruning*)



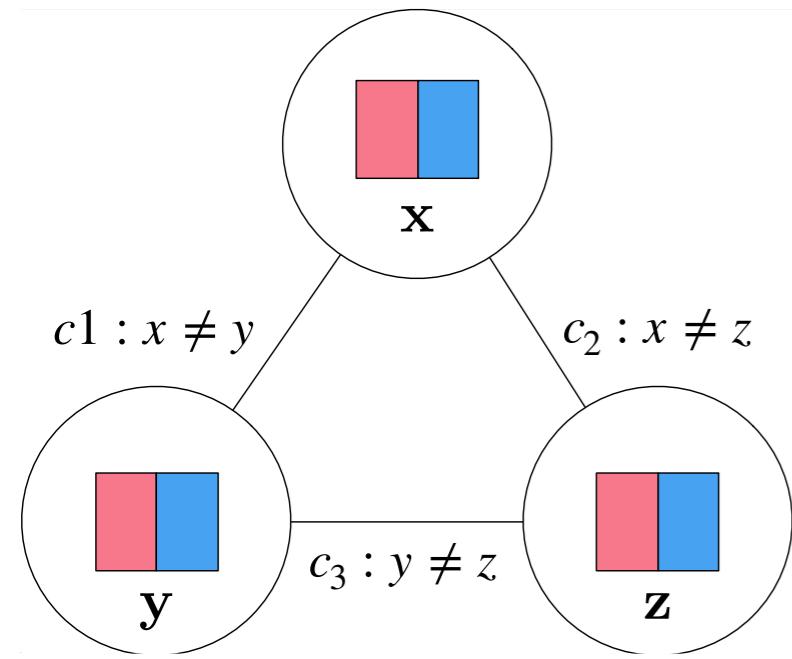
Champ d'application: fonctionne pour chaque contrainte binaire

Complexité temporelle: $\mathcal{O}(d^2)$ (taille des domaines)

En pratique: le propagateur doit être appelé au moins deux fois (pour x et y)

Beaucoup d'améliorations existantes: AC3, AC4, AC5, AC6, AC2001

Limitation de la cohérence d'arc



Est-ce qu'on peut éliminer toutes les configurations infaisables ?

Essai: appliquons la cohérence d'arc sur les trois contraintes

- (1) $c_1 (x \rightarrow y)$: toutes les valeurs de x ont un support dans y
- (2) $c_1 (y \rightarrow x)$: toutes les valeurs de y ont un support dans x
- (3) c_2 et c_3 : résultat similaire

Mauvaise nouvelle: inconsistance non détectée (2 couleurs/3 variables)

Difficulté: la cohérence d'arc ne détecte que les conflits de chaque contrainte prise individuellement

Notre situation: le conflit est dû à la combinaison de plusieurs contraintes

On a besoin de notions de cohérence plus forte pour gérer cette situation (et avoir un propagateur dédié)

Autres formes de cohérence

K-consistence: principe de la cohérence d'arc, mais pour k variables (cohérence plus forte)

Contraintes globales: algorithme de propagation dédié et spécifique à la contrainte (p.e., *alldifferent*)

Obtenir une cohérence plus forte est généralement plus coûteux !

Bound-consistence: cohérence d'arc, mais appliquée sur les bornes des domaines (cohérence moins forte)

Gestion des propagations



Comment gérer les CSPs avec plusieurs contraintes ?

Satisfy

Subject to $x \geq 3 + 2y$ (c_1)

$x \leq 8$ (c_2)

$x \in \{1, \dots, 10\}$

$y \in \{1, \dots, 10\}$

Domaine initial

$x \in \{1, \dots, 10\}$

$y \in \{1, \dots, 10\}$

Après la propagation de c_1

$x \in \{5, \dots, 10\}$

$y \in \{1, 2, 3\}$

Après la propagation de c_2

$x \in \{5, \dots, 8\}$

$y \in \{1, 2, 3\}$

Etat actuel: les deux contraintes ont été propagées une seule fois



Est-il possible de réduire davantage les domaines ?

Oui: le fait d'avoir modifié la variable x via c_2 rend possible de nouvelles propagations via c_1

Modification: La valeur 3 de y n'a aucun support dans x , on peut donc la retirer du domaine

$$y \leq \frac{x - 3}{2} \rightarrow y \leq 2.5, \text{ avec } x = 8$$



$x \in \{5, \dots, 8\}$
 $y \in \{1, 2\}$



Gestion des propagations

A chaque fois qu'une valeur est retirée du domaine d'une variable,
on doit re-propager toutes les contraintes impliquant la variable modifiée

Algorithme du point-fixe

Algorithme du point fixe (*fix-point algorithm*)

Fonction: réalisation concrète de la gestion des propagations dans un solveur CP

Objectif: propager successivement les contraintes jusqu'à ce que les domaines ne peuvent plus être réduits

FixPoint(P) :

$\langle X, D, C \rangle = P$

$Q = \text{FIFO}()$

$\text{push}(Q, C)$

while $Q \neq \emptyset$:

$c = \text{pop}(Q)$

$D' = \text{propagate}(c)$

 if there is an empty domain in D' : return failure

$X' = \{x \in X \mid D'(x) \neq D(x)\}$

$C' = \text{constraintsInvolvedBy}(X')$

$\text{push}(Q, C')$

$D = D'$

return success

On maintient une liste des contraintes à propager

Toutes les contraintes doivent être propagées au moins une fois

Tant qu'il reste des contraintes à propager...

...On prend une contrainte (ordre FIFO - autres choix possibles)

...On la propage et on accède aux domaines modifiés

Si un des domaines est vide, on retourne un échec (pas de solutions)

Sinon, on accède aux variables modifiées par la propagation

On rajoute les contraintes impliquées dans la liste

On met à jour les domaines

Situation où tous les domaines sont non-vides

Le point de conception majeur est l'implémentation des algorithmes de propagation

Algorithme du point fixe: exemple

(1) Situation initiale

$$x \in \{1, \dots, 10\} \quad y \in \{1, \dots, 10\} \quad z \in \{1, \dots, 10\}$$

$$Q = \langle c_1, c_2, c_3 \rangle$$

(2) Propagation de c_1 (consistance d'arc)

$$x \in \{5, \dots, 10\} \quad y \in \{1, 2, 3\} \quad z \in \{1, \dots, 10\}$$

$$Q = \langle c_2, c_3 \rangle$$

(3) Propagation de c_2 (consistance d'arc)

$$x \in \{5, \dots, 8\} \quad y \in \{1, 2, 3\} \quad z \in \{1, \dots, 10\}$$

$$Q = \langle c_3, c_1 \rangle \quad \text{La contrainte } c_1 \text{ est rajoutée dans la queue car elle dépend d'une variable modifiée (x)}$$

(4) Propagation de c_3 (consistance d'arc)

$$x \in \{5, \dots, 8\} \quad y \in \{1, 2, 3\} \quad z \in \{2, 3, 4\}$$

$$Q = \langle c_1 \rangle \quad \text{Aucune contrainte n'est ajoutée car la variable modifiée (z) n'est pas impliquée dans les autres contraintes}$$

(5) Propagation de c_1 (consistance d'arc)

$$x \in \{5, \dots, 8\} \quad y \in \{1, 2\} \quad z \in \{2, 3, 4\}$$

$$Q = \langle c_3 \rangle \quad \text{La contrainte } c_3 \text{ est rajoutée dans la queue car elle dépend d'une variable modifiée (x)}$$

(6) Propagation de c_3 (consistance d'arc)

$$x \in \{5, \dots, 8\} \quad y \in \{1, 2\} \quad z \in \{2, 3\}$$

$$Q = \langle \rangle \quad \text{Aucune contrainte n'est ajoutée car la variable modifiée (z) n'est pas impliquée dans les autres contraintes}$$

(7) La queue est vide: l'algorithme du point fixe s'arrête

Satisfy	
Subject to	$x \geq 3 + 2y (c_1)$
	$x \leq 8 (c_2)$
	$z = y + 1 (c_3)$
	$x \in \{1, \dots, 10\}$
	$y \in \{1, \dots, 10\}$
	$z \in \{1, \dots, 10\}$

Algorithme du point-fixe et propagation en pratique

FixPoint(P) :

$\langle X, D, C \rangle = P$

$Q = \text{FIFO}()$

$\text{push}(Q, C)$

while $Q \neq \emptyset$:

$c = \text{pop}(Q)$

$D' = \text{propagate}(c)$

if there is an empty domain in D' : return failure

$X' = \{x \in X \mid D'(x) \neq D(x)\}$

$C' = \text{constraintsInvolvedBy}(X')$

$\text{push}(Q, C')$

$D = D'$

return success

PropagationAC(x, y, c) :

for each $v_x \in D(x)$:

if no v_y from $D(y)$ allows c with v_x to be satisfied :

$D(x) = D(x) \setminus v_x$

```

@Override
public void fixPoint() {
    try {
        notifyFixPoint();
        while (!propagationQueue.isEmpty()) {
            propagate(propagationQueue.remove());
        }
    } catch (InconsistencyException e) {
        // empty the queue and unset the scheduled status
        while (!propagationQueue.isEmpty())
            propagationQueue.remove().setScheduled(false);
        throw e;
    }
}
  
```

<https://github.com/minicp/minicp/blob/master/src/main/java/minicp/engine/core/MiniCP.java>

```

// dom consistent filtering in the direction from -> to
// every value of to has a supportY in from   X
private void pruneEquals(IntVar from, IntVar to, int[] domVal) {
    // dump the domain of to into domVal
    int nVal = to.fillArray(domVal); Vx
    for (int k = 0; k < nVal; k++)
        if (!from.contains(domVal[k])) to.remove(domVal[k]);
}
  
```

Contrainte: $x = y$

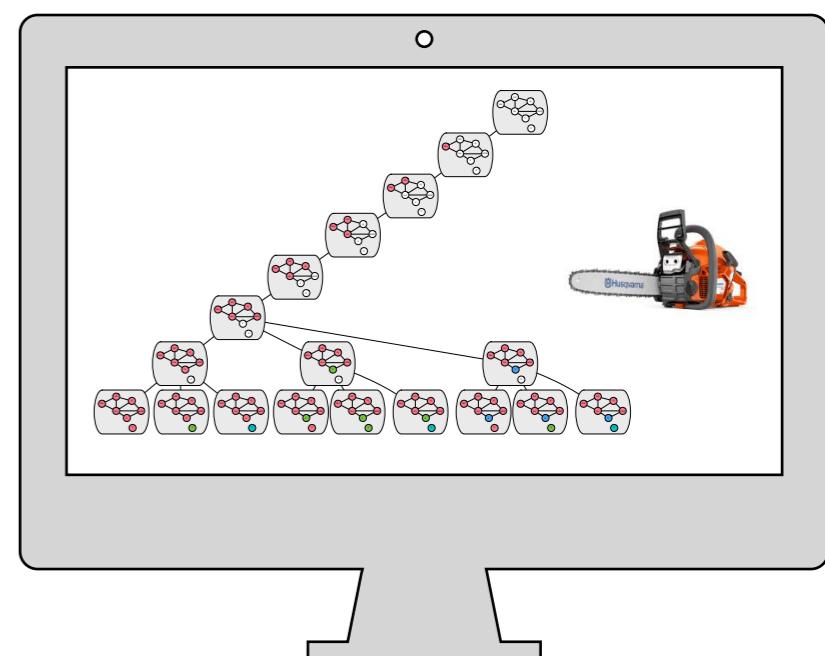
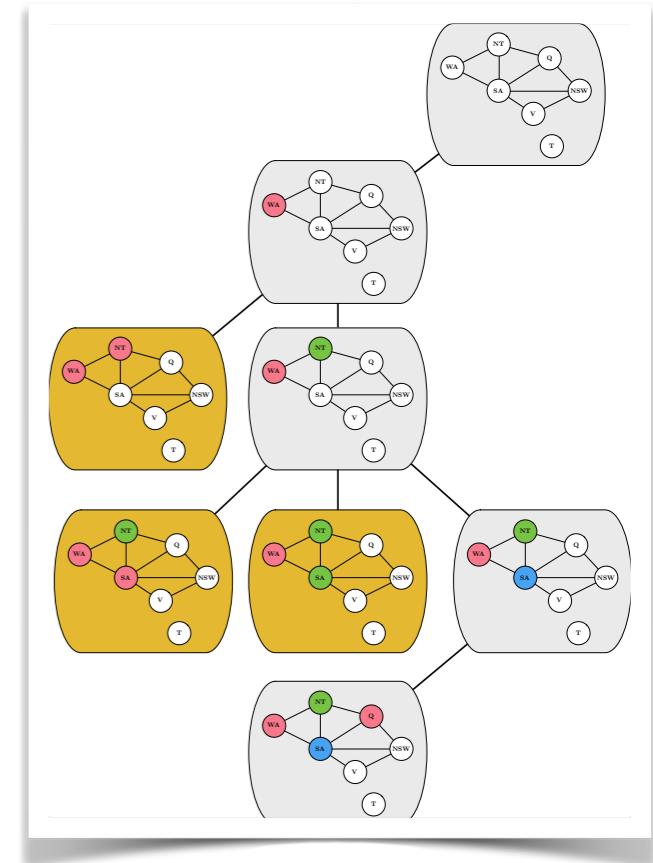
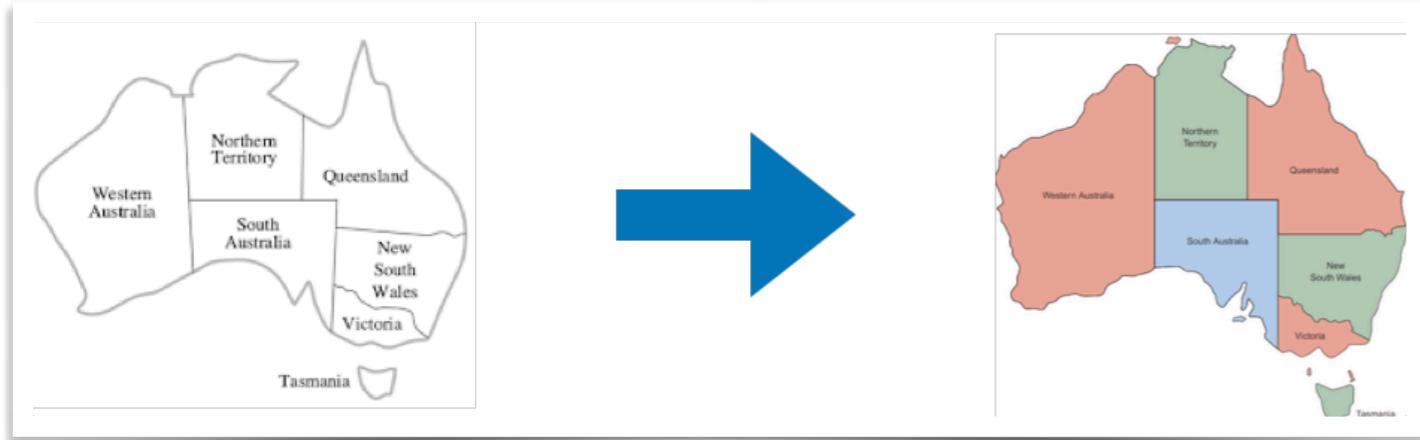
<https://github.com/minicp/minicp/blob/master/src/main/java/minicp/engine/constraints/Equal.java>

Table des matières

Programmation par contraintes



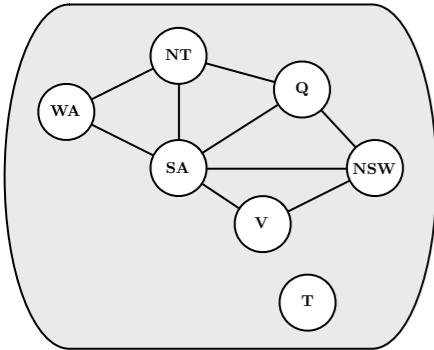
1. Limitations et difficultés de la recherche locale
2. Concepts et principes fondamentaux de la programmation par contraintes
3. Formalisation des problèmes combinatoires
4. Modélisation en programmation par contraintes
5. Recherche avec retours-arrières (*backtracking search*)
6. Propagation de contraintes (*arc consistency*)
7. Algorithme du point fixe
8. Fonctionnement d'un solveur de programmation par contraintes



Processus de résolution en programmation par contraintes

Programmation par contraintes = modèle + recherche + propagation

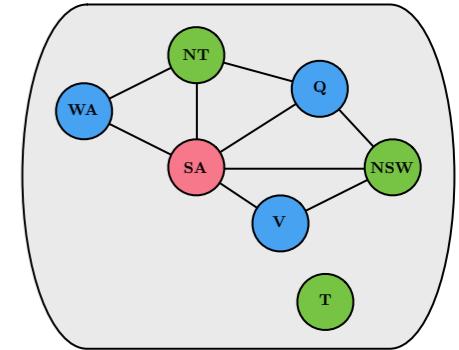
CSP à résoudre



Modélisation mathématique

Satisfy
Subject to $x_u \neq x_v$ $\forall (u, v) \in E$
 $x_v \in \{\text{red, green, blue}\}$ $\forall v \in V$

Solution



```
int: nc = 3;

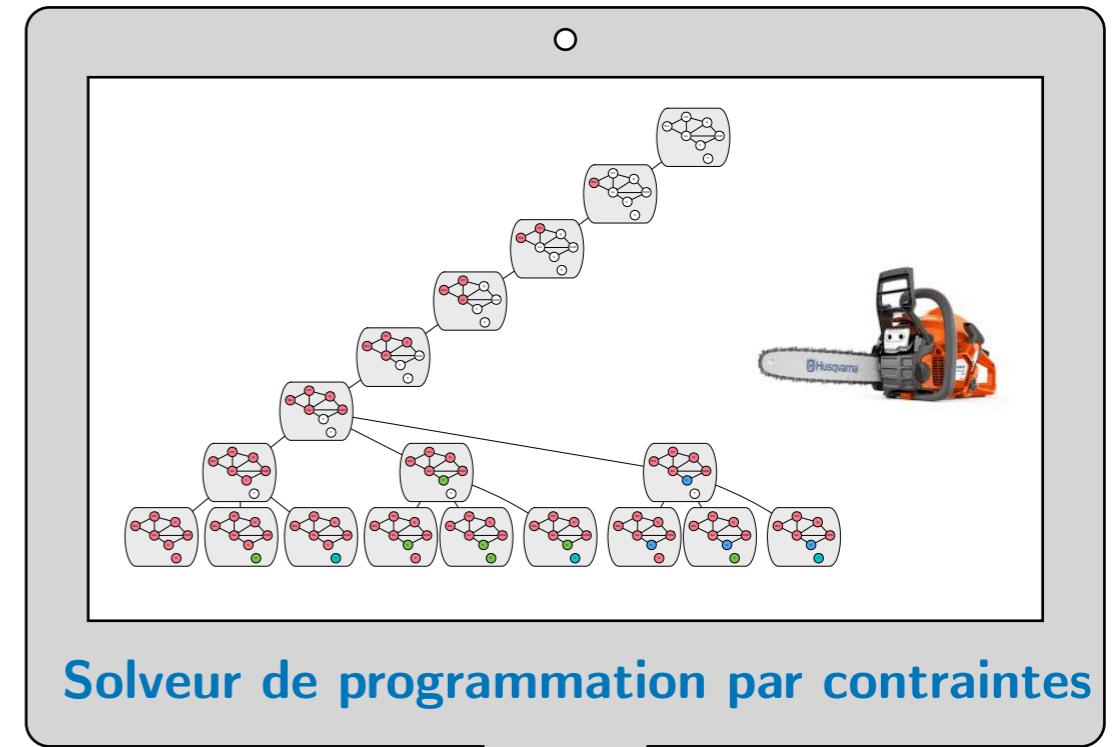
var 1..nc: wa;  var 1..nc: nt;  var 1..nc: sa;  var 1..nc: q;
var 1..nc: nsw; var 1..nc: v;   var 1..nc: t;

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;

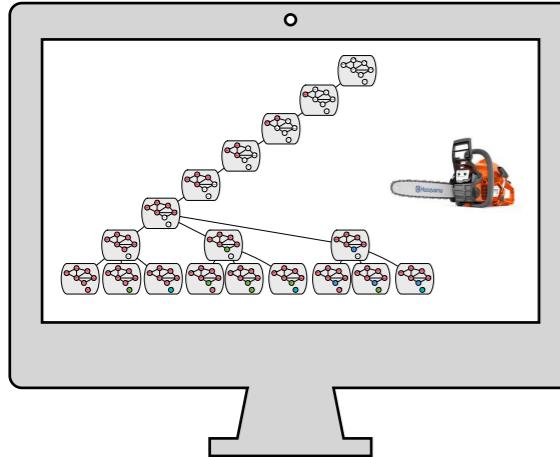
solve satisfy;
```



Modèle Minizinc (input du solveur)



Fonctionnement d'un solveur CP



Recherche: recherche à retours-arrières (*backtracking search*)

Propagation: propagateur propre à chaque contrainte (p.e., *arc-consistency*)

Gestion des propagation: algorithme du point fixe



Comment ces trois mécanismes sont agencés dans un solveur ?

(1) La recherche est lancée sur le problème initial

(2) L'algorithme du point fixe est exécuté à chaque noeud de l'espace de recherche

Ainsi, à chaque fois qu'on assigne une valeur à une variable, on réduit au maximum les domaines

(3) Lorsqu'un domaine est vide, notre configuration actuelle est infaisable et on effectue un retour arrière

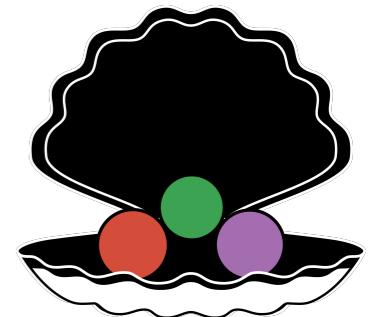
La plupart des solveurs modernes sont basés sur ce mécanisme



Gecode (C++)



OSCAR
OPERATIONAL RESEARCH IN SCALA



SeaPearl.jl

Pseudo-code d'un solver CP

Le principe général est une recherche à retours-arrières qui intègre le point-fixe à chaque noeud de l'arbre

CPSolver(P) :

$\langle X, D, C \rangle = P$

$s = \{\}$

status = FixPoint(s, P)

status is failure : return no solution

$L = \text{LIFO}()$

push(L, s)

while $L \neq \emptyset$:

$s = \text{pop}(L)$

if assignmentCompleted(s) : return s

$x = \text{selectUnassignedVariable}(s, X)$

for each $v \in D(x)$:

$s' = s \cup \{x \leftarrow v\}$

status = FixPoint(s', P)

status is success : push(L, s')

return no solution

Structure générale: recherche à retours-arrières (slides précédents)

Application du point fixe pour réduire les domaines (D change après l'opération)

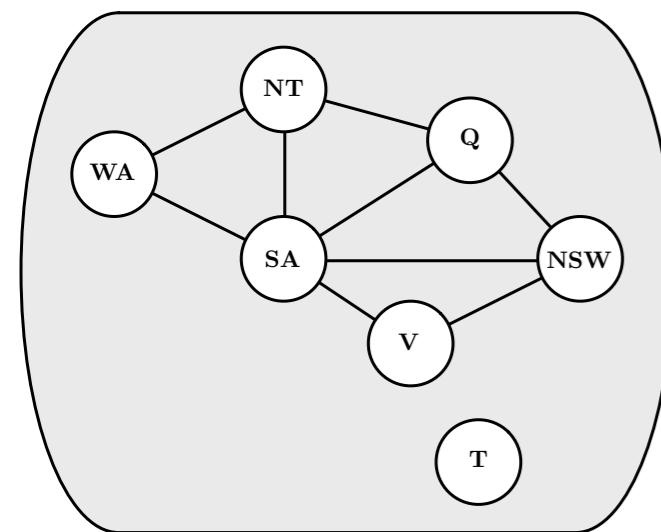
Si un domaine est vide, on sait déjà que le problème est infaisable

Application du point fixe sur un noeud de recherche

Seulement si on a que des domaines non-vides, alors on peut brancher

En pratique: d'autres mécanismes sont nécessaires pour rendre le solveur plus efficace

Recherche + propagation: illustration



Exemple: problème de coloration de graphe

Etat initial: toutes les variables ont les trois couleurs dans leur domaine



Exécution du point fixe: aucune modification des domaines



$x_{WA} = \text{red}$

WA	NT	Q	NSW	V	SA	T
Red						
Green						
Blue						

Point fixe: rouge enlevé de NT et SA

Branchement de la recherche

$x_Q = \text{green}$

WA	NT	Q	NSW	V	SA	T
Red	White	Green	Red	Red	Green	Red
Green	Blue	White	Red	Red	Blue	Red

Point fixe: vert enlevé de NSW, NT et SA

Branchement de la recherche

$x_V = \text{blue}$

Backtrack

$x_V = \text{green}$

WA	NT	Q	NSW	V	SA	T
Red	Green	White	Red	Blue	White	Red
Green	Blue	White	Red	White	Blue	Red

Point fixe: bleu enlevé de NSW et SA

Domaine vide pour SA

Visualisation du processus sur Mizininc

```
int: n = 3;
int: tot = n * (n^2 + 1) div 2;

array[1..n,1..n] of var 1..n*n: x;

constraint all_different(x);

constraint forall(k in 1..n)(sum(i in 1..n)(x[k,i]) == tot);
constraint forall(k in 1..n)(sum(i in 1..n)(x[i,k]) == tot);
constraint sum(i in 1..n) (x[i,i]) = tot;
constraint sum(i in 1..n) (x[i,n-i+1]) = tot;

solve satisfy;
```

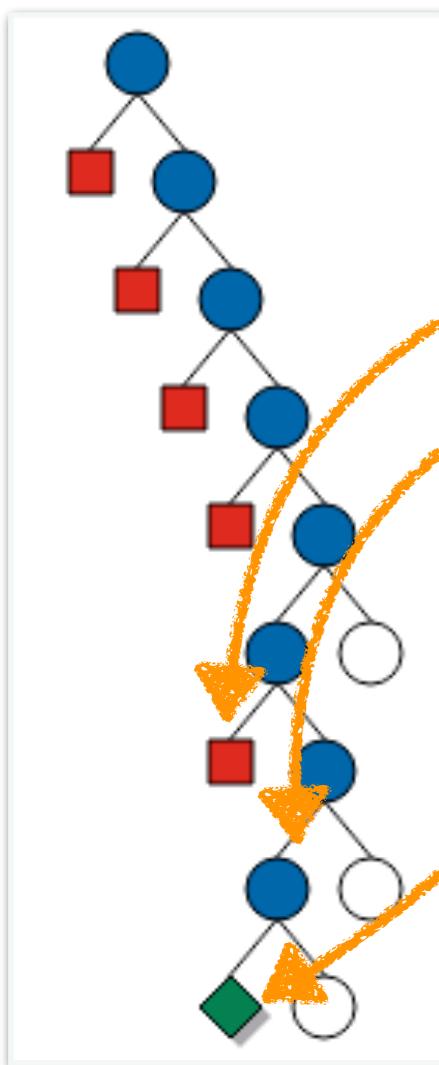
Solver configuration:
Run ▾ Gecode Gist 6.3.0

Gist

- Inspect
- Inspect before fixpoint
- Compare
- Compare before fixpoint
- Node statistics
- Center current node

Next solution

All solutions



Gist Console: Gecode/FlatZinc

Clear Stay on top

```
x = array2d(1..3, 1..3, [5, 9, 1, 1, 5, 9, 9, 1, 5]);
x = array2d(1..3, 1..3, [{4,6}, {7,9}, 2, {1,3}, 5, {7,9}, 8, {1,3}, {4,6}]);
x = array2d(1..3, 1..3, [6, 7, 2, 1, 5, 9, 8, 3, 4]);
```

5	9	1
1	5	9
9	1	5

Solution
infaisable

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

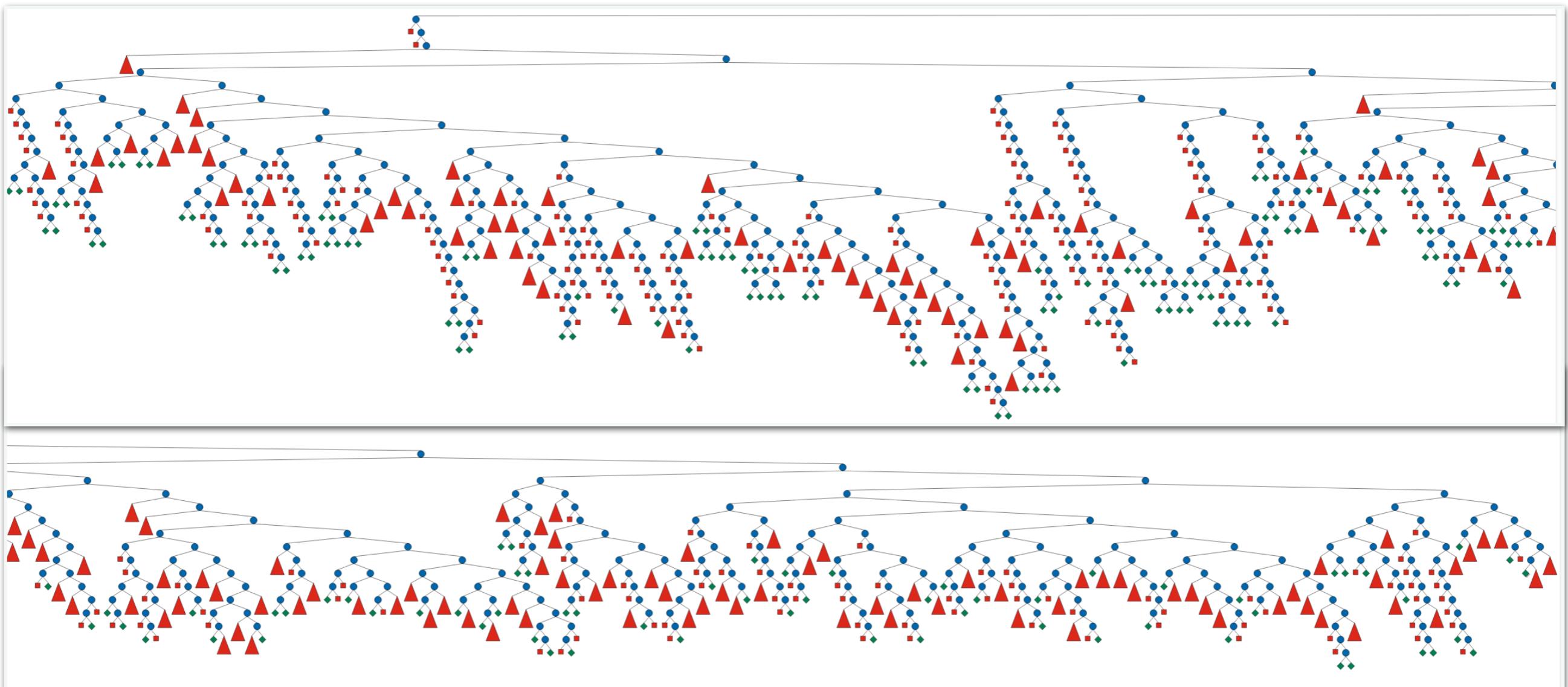
Solution
non-complete

6	7	2
1	5	9
8	3	4

Solution
faisable

En pratique...

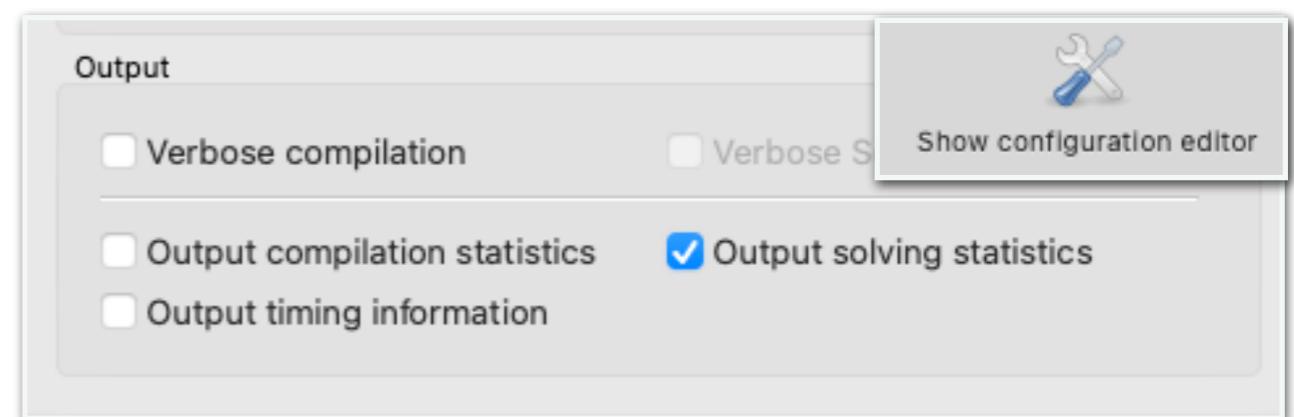
Situation: carré magique de taille 4x4 où on cherche **TOUTES** les solutions



Nombre de solutions faisables: 7040

Nombre de backtracks (failures): 202 754

Nombre de noeuds explorés dans l'arbre: 419 587



Conception de solveurs

Critères fondamentaux

Efficacité: performance des mécanismes internes pour résoudre le problème

Expressivité: le nombre de contraintes disponibles pour l'utilisateur

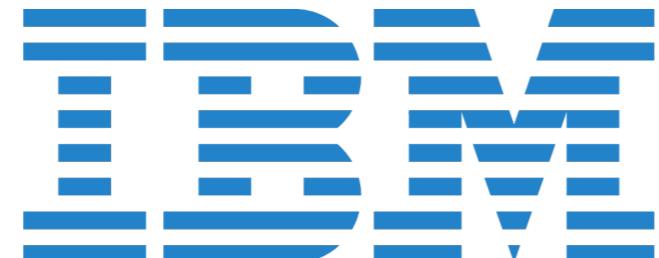
Simplicité d'utilisation: la facilité qu'a l'utilisateur à prendre le solveur en main et à le manipuler

Modularité: la possibilité de modifier et d'améliorer le solveur (à des fins de recherche par exemple)

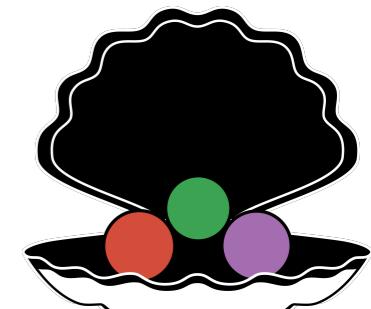
Conception de solveurs



Gecode (C++)



CP Optimizer



SeaPearl.jl

Chaque solveur existant a son propre équilibre entre ces critères



Google OR-Tools

Ils diffèrent par plusieurs choix de conception

Langage de programmation

Algorithme des propagateurs

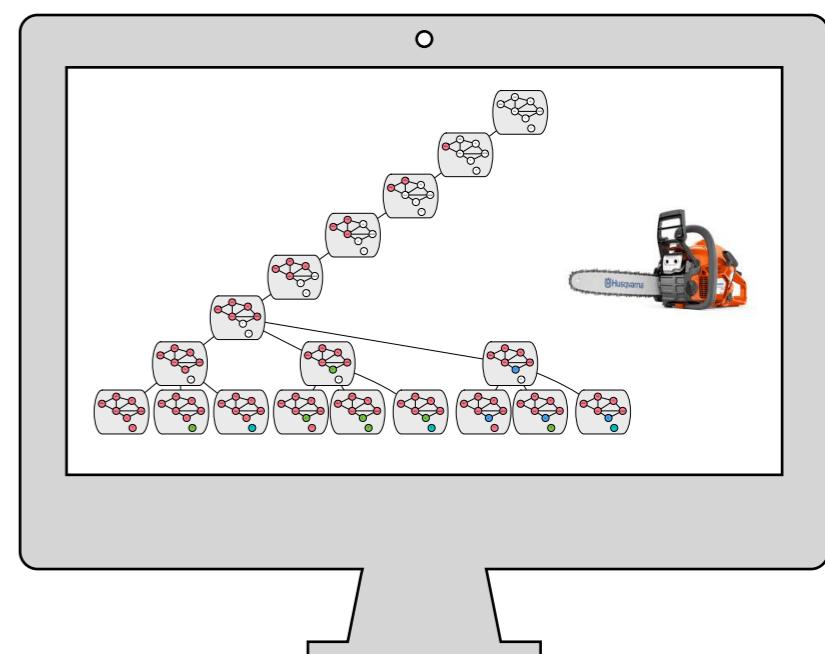
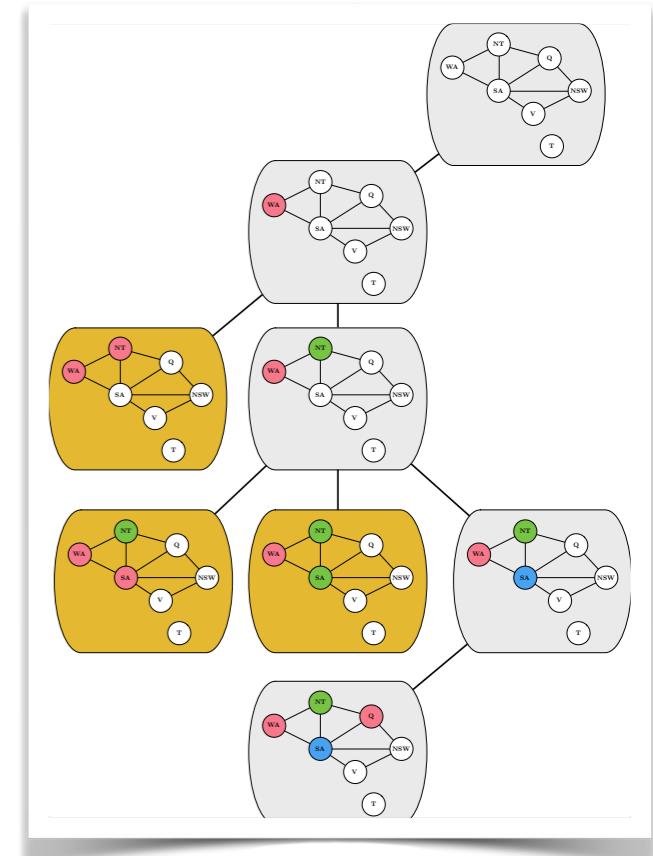
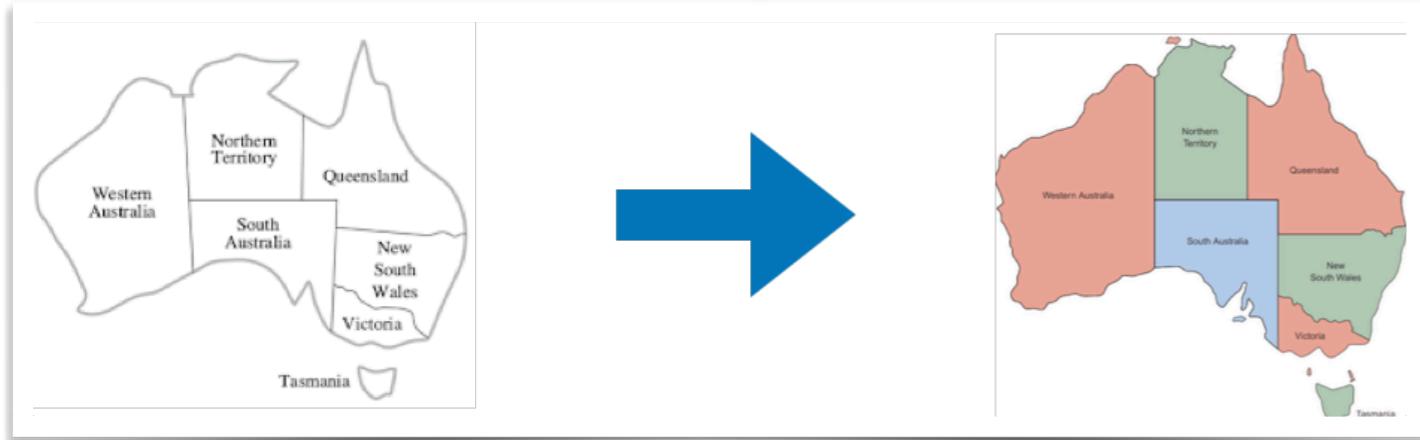
Structure de données

Mécanismes supplémentaires (apprentissage, heuristiques de recherche, etc.)

Table des matières

Programmation par contraintes

- ✓ 1. Limitations et difficultés de la recherche locale
- ✓ 2. Concepts et principes fondamentaux de la programmation par contraintes
- ✓ 3. Formalisation des problèmes combinatoires
- ✓ 4. Modélisation en programmation par contraintes
- ✓ 5. Recherche avec retours-arrières (*backtracking search*)
- ✓ 6. Propagation de contraintes (*arc consistency*)
- ✓ 7. Algorithme du point fixe
- ✓ 8. Fonctionnement d'un solveur de programmation par contraintes



Synthèse des notions vues

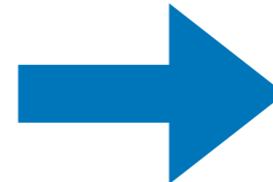
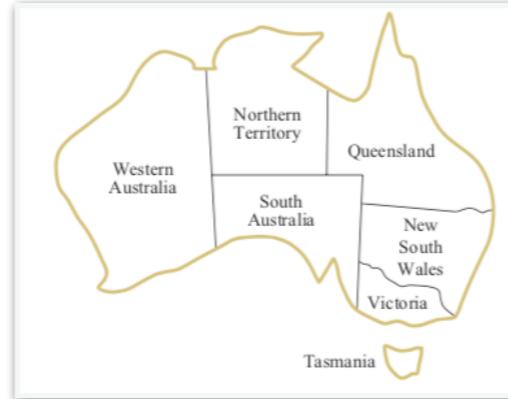
Formalisation des problèmes combinatoires (CSP et COP)

Variables

Domaines

Contraintes

Fonction objectif



Satisfy

Subject to $x_u \neq x_v$

$\forall (u, v) \in E$

$x_v \in \{\text{red, green, blue}\}$

$\forall v \in V$

Programmation par contraintes = modèle + recherche + propagation

```
int: nc = 3;

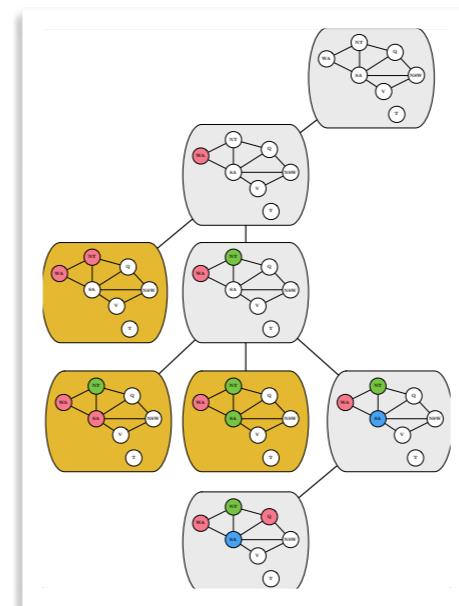
var 1..nc: wa;  var 1..nc: nt;  var 1..nc: sa;  var 1..nc: q;
var 1..nc: nsw; var 1..nc: v;   var 1..nc: t;

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;

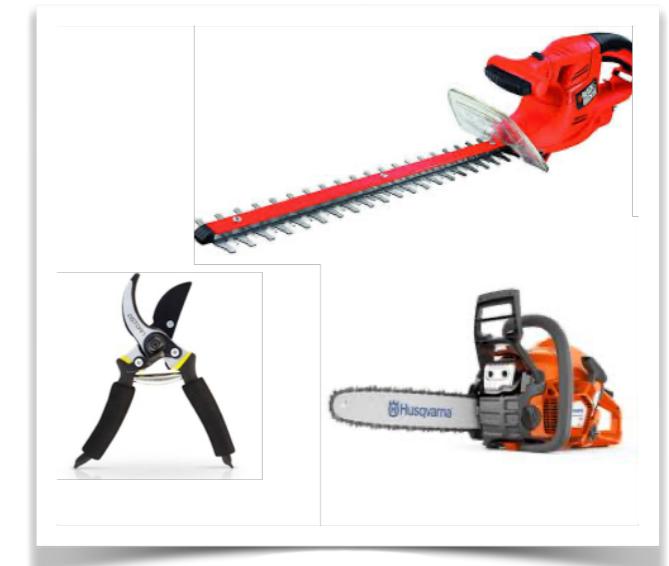
solve satisfy;
```



Modèle du problème

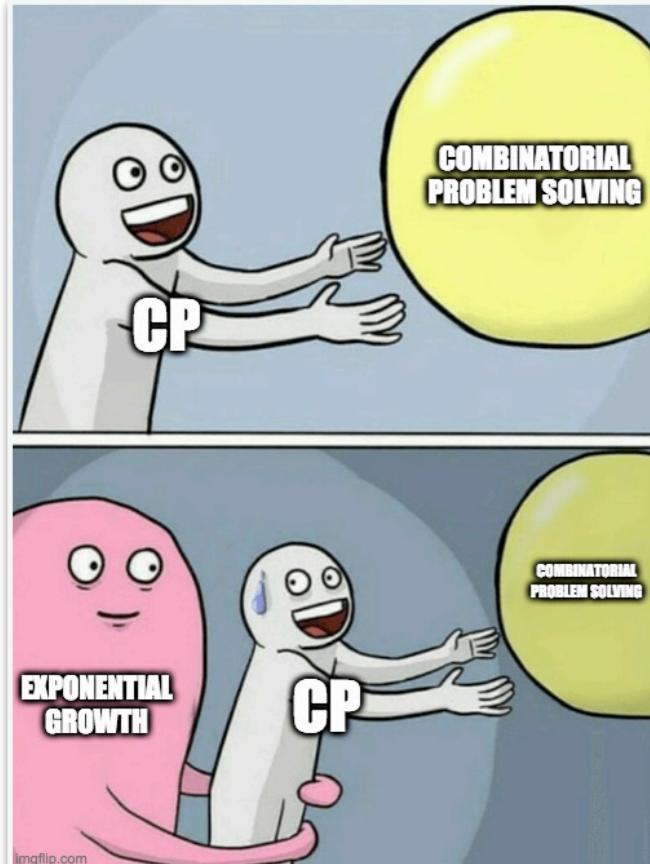


Recherche à
retours arrières



Propagateurs de contraintes
et algorithme point fixe

Aller plus loin en CP



Limitations

Limitation: pas toujours la méthode la plus efficace en pratique

Limitation: croissance exponentielle du nombre de possibilités

Difficultés: performances fort dépendantes à la qualité du modèle construit

Points non abordés

Implémentation d'autres algorithmes de propagation

Elaboration d'heuristiques de recherche

Structures de données utilisées dans un solveur

Astuces pour construire des bons modèles

Combinaison avec de la recherche locale

Aller plus loin...

INF6101 - Programmation par contraintes (Gilles Pesant)

Coursera - <https://www.coursera.org/lean/basic-modeling>

Cours en ligne MiniCP - <http://www.minicp.org/>

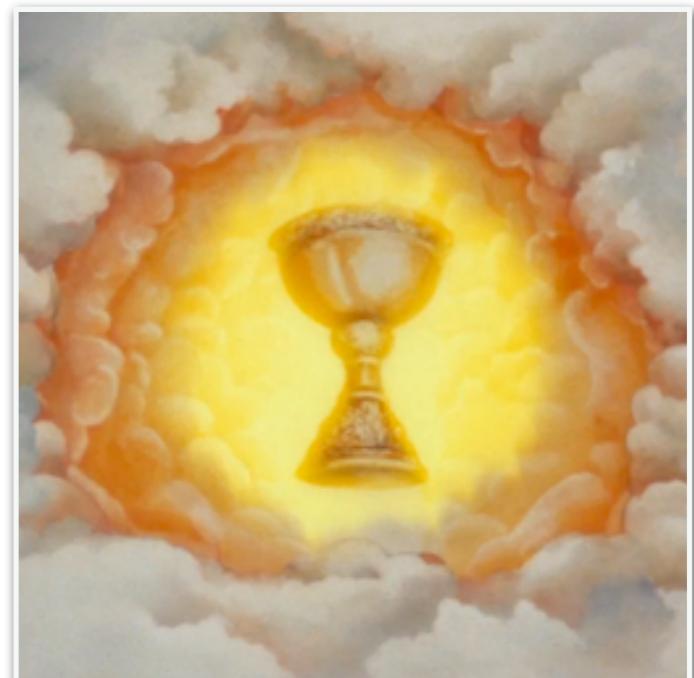
Exemples de questions d'examen

Théorie

1. Expliquer les différences entre la recherche locale et la programmation par contraintes
2. Expliquer ce qu'est la propagation, et savoir l'illustrer sur un exemple
3. Expliquer en quoi la recherche DFS pure n'est pas adaptée pour la résolution de CSP

Pratique

1. Savoir modéliser adéquatement un problème comme un CSP ou un COP
2. Savoir appliquer le backtracking search sur un exemple
3. Savoir appliquer l'algorithme du point fixe sur un exemple





DALLE: *Painting of Monet, a woodchopper with the Holy Grail*