La programmation par contraintes

Philippe Morignot
pmorignot@yahoo.fr

Plan de la séance

- Généralités
- Modèle et modélisation
- Algorithmes
- Filtrage avant
- Retour-arrière
- Consistance
- Structures
- Conclusion

Généralités

- La programmation par contraintes est un paradigme pour résoudre des problèmes combinatoires.
- Autres approches :
  - Programmation linéaire en nombres entiers
  - Algorithmes génétiques
  - Algorithmes de recherche dans un espace d'états (par ex., algorithmes aveugles, algorithmes heuristiques comme A*).
  - Algorithme du recuit simulé
  - Algorithme tabou
  - …

Généralités

- Un problème combinatoire est un problème ..
  - … qui peut se formuler sous forme d'entités …
  - … entretenant des relations
  - … et dont il faut trouver une combinaison :

    la solution au problème posé.
- Il peut y avoir plusieurs solutions.
    - En trouver une
    - En trouver la meilleure

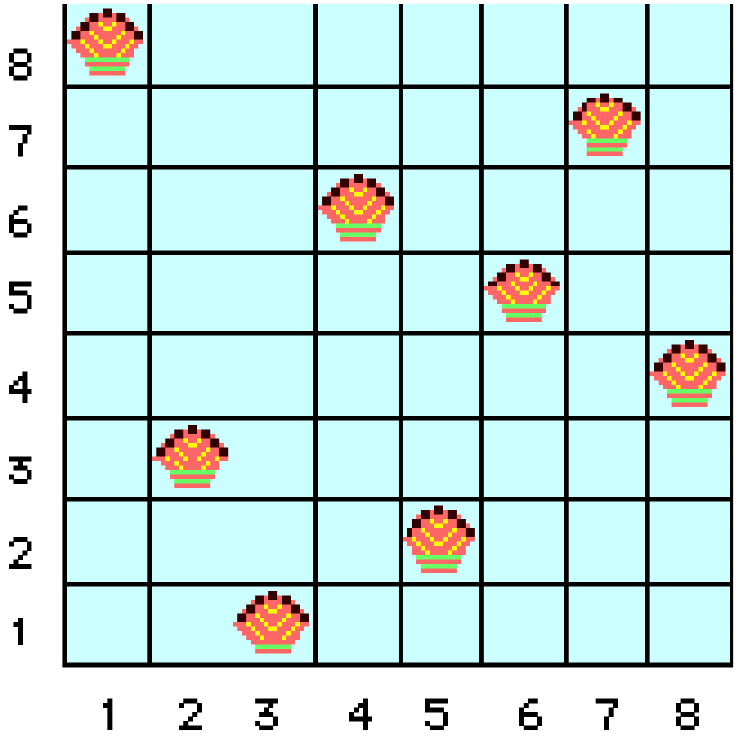
Généralités

- Exemple de problème combinatoire : le jeu du sudoku.

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

Généralités

- Exemple de problème combinatoire : les N reines (ici, N = 8).



Généralités

- Exemple de problème combinatoire : la cryptarithmétique.

```
  SEND
+ MORE
--------------
 MONEY
```

```
        UN
      DEUX
+  DEUX
+  DEUX
+  DEUX
------------
     NEUF
```

Généralités

- La difficulté : le nombre de combinaisons à envisager peut être gigantesque pour un problème de taille réelle.
  - Exemple : pour le jeu de sudoku, une évaluation grossière du nombre de possibilités est :

    $(8!)^9 \approx 10^{41}$ possibilités.
  - Pour des petits problèmes combinatoires, (presque) tout algorithme marche …
- Conséquence : parcourir toutes ces combinaisons une à une prendrait un temps démesuré, même sur le plus rapide des ordinateurs.
  - Phénomène d'explosion combinatoire.
  - Dans le pire des cas, le nombre de combinaisons à envisager est une fonction exponentielle de la taille d'une dimension des données.

Généralités

```
while(…) {
  combinaison = nextCombinaison(existant);
  if (qualite(combinaison) > bestQualite) {
     bestQualite = qualite(combinaison);
     bestCombinaison = combinaison;
  }
}
```

BEAUCOUP TROP LONG !!!!
(sauf sur des petits problèmes)

Généralités

- Idée de la programmation par contraintes :
  - Prendre en compte la structure du problème : décomposer le problème en
    - Variables
      - Chaque variable possède un domaine fini (variable en extension).
    - Relations entre variables (contraintes)
      - Une contrainte doit toujours être vérifiée et réduit les domaines.
  - Un algorithme unique qui utilise intelligemment ce modèle.
    - Heuristiques

Généralités

- Exemples de problèmes abordés par la programmation par contraintes :
  - Affecter des stages à des étudiants en respectant leurs souhaits
  - Planifier le trafic aérien de manière optimale (affectation des couloirs de vol à des avions, optimisation des rotations d'équipage, …)
  - Ordonnancer des tâches pour qu'elles finissent au plus tôt en consommant peu de ressources
  - Optimiser le placement des composants dans un circuit électronique
  - Etablir un menu à la fois équilibré et appétissant
  - …

Généralités

- La programmation par contraintes a été proposée par Jean-Louis Laurière en 1976 dans son doctorat d'état.
- Publication :
  - Jean-Louis Laurière: A Language and a Program for Stating and Solving Combinatorial Problems. Artif. Intell. 10(1): 29-127 (1978).
- Packages : OPL Studio d'IBM, CHOCO de EMN, CHIP de COSYTEC, SICSTUS PROLOG, etc.
- Association Française de Programmation par Contraintes : http://afpc.greyc.fr/web/

Modèle

- Variables discrètes avec domaine fini :
  - Pour i de 1 à n, une variable V_i
  - Pour j de 1 à n, un domaine $D_j = \{ v_1, v_2, \ldots, v_{f(j)} \}$.
  - Pour tout i, $V_i \in D_i$
- Contraintes sur ces variables :
  - Pour k de 1 à m, $C_k = (X_k, R_k)$ avec :
    - $X_k = \{ V_{i1}, V_{i2}, \ldots, V_{ik} \}$ // Les variables de la contrainte C_k
    - $R_k \subset D_{i1} \times D_{i2} \times \ldots \times D_{ik}$ // Les valeurs possibles de ces // variables, compatibles // ensemble avec la contrainte C_k

Vocabulaire

- On appelle affectation le fait d'associer une variable à une valeur de son domaine
  - La variable V_i est affectée à la valeur v_{ij} : $D_i = \{ v_{ij} \}$
- Une assignation de variables à des valeurs est :

  $A = \{(V_{i1}, v_{i1}), (V_{i2}, v_{i2}), \ldots, (V_{ik}, v_{ik})\}$
- Une assignation peut être :
  - totale : toutes les variables ont une valeur (k = n)
  - partielle : certaines variables ont une valeur, mais pas les autres (k < n).
- Une assignation *A* est consistante / cohérente ssi elle ne viole aucune contrainte C_k.
- Une solution d'un CSP (Constraint Satisfaction Problem) est une assignation totale consistante.
- Certains CSP demandent aussi de maximiser une fonction objectif *f*.

Vocabulaire

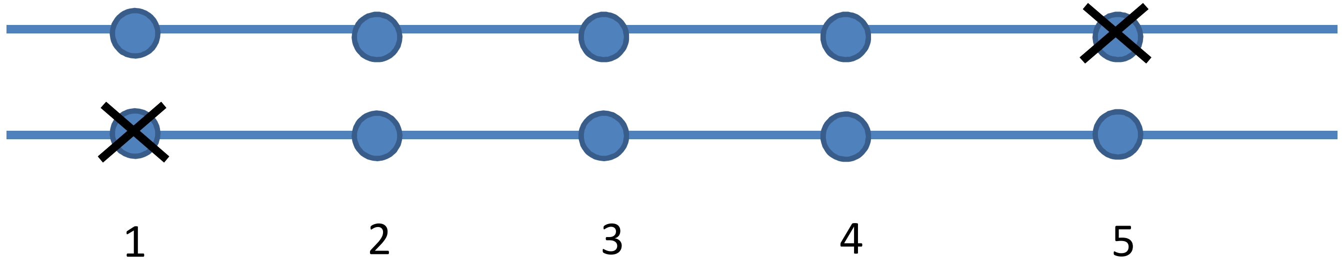
- Un CSP peut être :
  - Sous contraint : trop peu de contraintes.
  - Sur contraint : trop de contraintes.
- Etant donné un CSP, on peut :
  - Rechercher une solution
  - Rechercher toutes les solutions
  - Rechercher une solution optimale vis-à-vis d'une fonction objectif
  - Prouver l'absence de solution.

Les contraintes

- Une contrainte peut être exprimée :
  - En extension : fournir les ensembles de valeurs possibles des variables
  - Arithmétiquement : <, ⩽, >, ⩾, =, ≠, +, -, /, *, …
  - Logiquement : ⇔, =>, <=, OU, ET, NON, …
  - Globalement : AllDifferent(x_1, x_2, …, x_n), Geost(f_1, f_2, …, f_n), ...
- Une contrainte peut être :
  - Dure : elle doit toujours être vérifiée
  - Molle : elle est parfois vérifiée parfois violée, mais selon un critère
- Une contrainte peut être :
  - Unaire. Exemple : $x \in [1, 5]$
  - Binaire. Exemple : $x < y$
  - N-aire. Exemple : AllDifferent(V_1, V_2, …, V_n)

Les contraintes

- Exemple de contraintes dures :
  - $x \in [1, 5]$; $y \in [1, 5]$; $x < y$
  - x :
  - y :



- Exemple de contraintes molles :
  - Dans un problème d'ordonnancement,

    $Y = \#(t_i < \text{deadline}_i)$ et maximiser Y

Contraintes globales

- AllDifferent($\boldsymbol{V_1}$, $\boldsymbol{V_2}$, …, $\boldsymbol{V_n}$)
  - Toutes les variables $\boldsymbol{V_i}$ doivent être distinctes
  - Logiquement équivalent à :

$$\begin{aligned} &\boldsymbol{V_1 \neq V_2 \wedge V_1 \neq V_3 \wedge \ldots \wedge V_1 \neq V_n} \wedge \\ &\qquad \boldsymbol{V_2 \neq V_3 \wedge \ldots \wedge V_2 \neq V_n} \wedge \\ &\qquad\qquad \ldots \wedge \\ &\qquad\qquad\qquad \boldsymbol{V_{n-1} \neq V_n} \end{aligned}$$

- <u>Propriété :</u> s'il y a **m** variables dans AllDiff, et **n** valeurs distinctes possibles ensemble, et que **m** > **n**, alors la contrainte ne peut pas être satisfaite.

Exemples de modélisation

- Un modèle pour le sudoku :
  - Une variable est une case vide d’une grille
  - Un domaine est l’ensemble des nombres entiers de 1 à 9
    - Si la case possède déjà un nombre, elle apparait comme une constante dans les contraintes
  - Contraintes :
    - Toutes les variables d’une petite grille sont différentes et différents des constantes
    - Toutes les variables d’une ligne sont différentes
    - Toutes les variables d’une colonne sont différentes

Exemples de modélisation

- Les N-reines (ici, N = 8) :
  - Une paire de variables ($\boldsymbol{x_i}$, $\boldsymbol{y_i}$) par reine ***i***. La reine ***i*** est sur la colonne $\boldsymbol{x_i}$ et la ligne $\boldsymbol{y_i}$
  - Le domaine de $\boldsymbol{x_i}$: [1, 8]
  - Le domaine de $\boldsymbol{y_i}$: [1, 8]
  - Contraintes :
    - $x_i \neq x_j$ // Colonnes différentes
    - $y_i \neq y_j$ // Lignes différentes
    - $x_i + y_i \neq x_j + y_j$ // 1e diagonale différente
    - $x_i - y_i \neq x_j - y_j$ // 2e diagonale différente

Exemple de modélisation

- Les N-reines (ici, N = 8) :
  - La variable $\boldsymbol{x_i}$ est la ligne de la ***i***-eme colonne sur laquelle se trouve cette reine.
  - Le domaine de $\boldsymbol{x_i}$ est [1, 8]
  - Les contraintes :
    - Les contraintes sur les colonnes sont vérifiées par construction
    - $x_i \neq x_j$ // lignes différentes
    - $x_i + \mathrm{i} \neq x_j + \mathrm{j}$ // 1^e diagonales différentes
    - $x_i - \mathrm{i} \neq x_j - \mathrm{j}$ // 2^e diagonales différentes

Exemple de modélisation

- Les N-reines (ici, N = 8) :
  - Les cases de l'échiquier sont numérotées de 1 à 64.
  - La variable $\boldsymbol{x_i}$ est le numéro de la case de la reine *i*.
  - Contraintes :
    - x_i / 8 ≠ x_j / 8 // Lignes différentes
    - x_i % 8 ≠ x_j % 8 // Colonnes différentes
    - Contraintes sur la 1e diagonale
    - Contraintes sur la 2e diagonale

Exemple de modélisation

- Cryptarithmétique : SEND + MORE = MONEY
- Le modèle :
  - Variables : S, M $\in$ [1, 9] ; E, N, D, O, R, N, Y $\in$ [0, 9]
  - Contraintes :
    - $D + E = Y + 10 * R1$
    - $N + R + R1 = E + 10 * R2$
    - $E + O + R2 = N + 10 * R3$
    - $S + M + R3 = O + 10 * M$
    - Variables auxiliaires : R1, R2, R3 $\in$ [0, 1]

Algorithmes

- Commutativité :
  - Un problème est commutatif ssi l'ordre d'application des actions n'a pas d'effet sur le résultat.
- Un CSP est commutatif : quand on assigne des valeurs à des variables, on atteint la même assignation partielle, quel que soit l'ordre.
- Conséquence : on peut assigner les variables les unes après les autres.

Algorithmes : le retour arrière

- Algorithme RECHERCHE-RETOUR-ARRIERE(*csp*)

  return RETOUR-ARRIERE-RECURSIF({}, *csp*)

- Algorithme RETOUR-ARRIERE-RECURSIF(*assignation*, *csp*)

  SI *assignation* est totale ALORS return *assignation*

  var <- CHOISIR-VARIABLE-NONAFFECTEE(Variables(*csp*), *assignation*, *csp*)

  POUR TOUT *val* dans ORDONNE-VALEURS-DOMAINE(*var*, *assignation*, *csp*)

  FAIRE

  SI *va*l est consistante avec *assignation* suivant Contraintes(*csp*) ALORS

  ajoute (*var* = *val*) à *assignation*

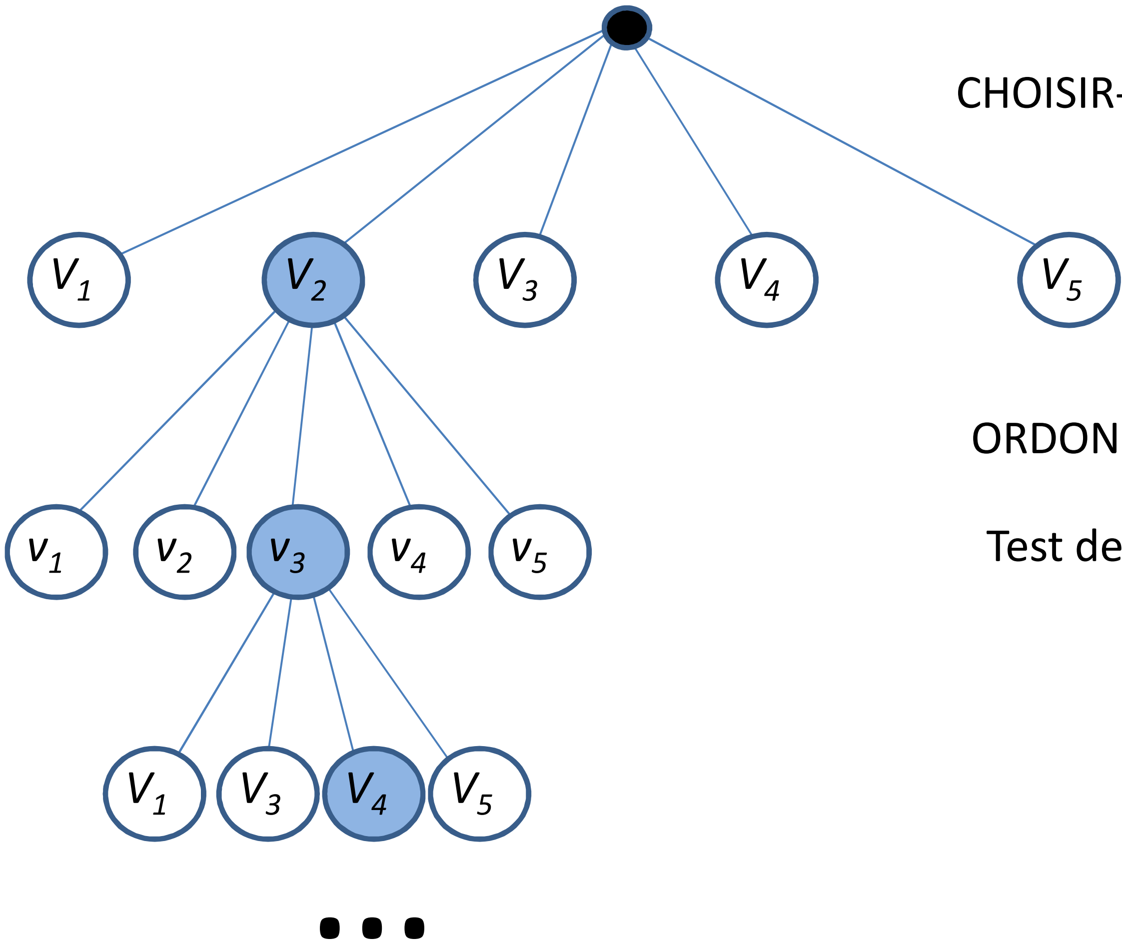
  resultat <- RETOUR-ARRIERE-RECURSIF(*assignation*, *csp*)

  SI *resultat* ≠ ECHEC ALORS return *resultat*

  enleve (*var* = *val*) de *assignation*

  return ECHEC

Algorithmes



CHOISIR-VARIABLE-NONAFFECTEE

ORDONNE-VALEURS-DOMAINE

Test de consistance

Fonctions heuristiques

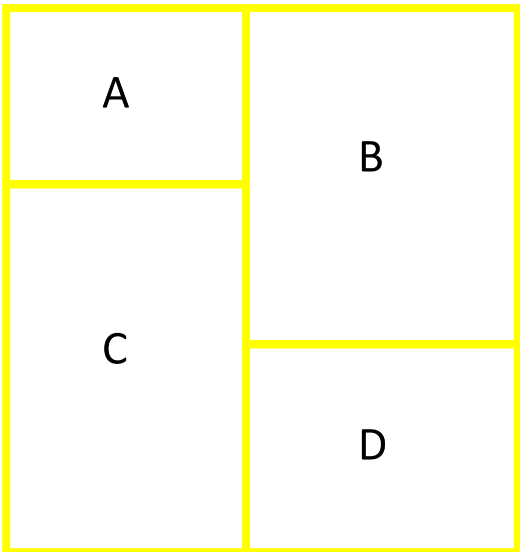
- Une fonction heuristique permet de faire un choix.
  - Exprimée à partir de variables, domaines et contraintes.
  - Ou peut être basée sur le domaine d'application du CSP.
  - Prototype en C++ : int heuristique1(Assignation* assignation, Csp* csp);
- Sur les variables : CHOISIR-VARIABLE-NONAFFECTEE()
  - Statiques / dynamiques.
  - Minimum remaining value / most constrained variable / first-fail : choisir la variable avec le domaine de cardinalité minimale.
  - Choisir la variable reliée au nombre maximal de contraintes.
  - …
- Sur les valeurs : ORDONNE-VALEURS-DOMAINE()
  - Statiques / dynamiques
  - Choisir la valeur qui enlève le moins de valeurs pour les autres variables.
  - …

Filtrage

- Quelle est l'implication pour les autres variables de l'affectation d'une variable à une valeur ?
- FORWARD-CHECKING : à chaque fois qu'une variable $\boldsymbol{V_i}$ est instantiée, considérer les variables $\boldsymbol{V_j}$ connectées à $\boldsymbol{V_i}$ par une contrainte $\boldsymbol{C_k}$, et enlever du domaine de $\boldsymbol{V_j}$ les valeurs qui sont inconsistantes avec $\boldsymbol{C_k}$.

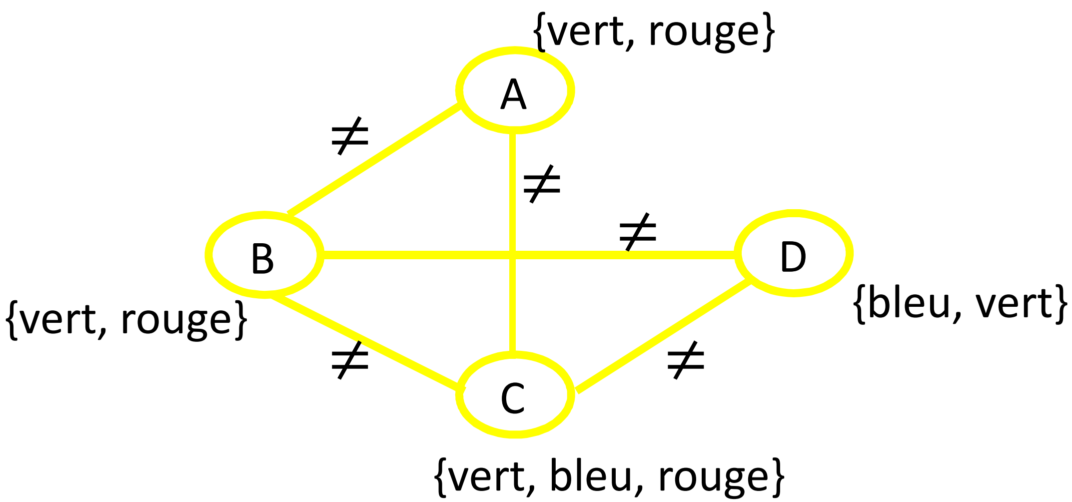
Filtrage par Forward-Checking
Coloration de carte

Colorier A,B,C et D (ci-dessous) sans que deux couleurs
ne se touchent, avec les couleurs possibles pour chaque case :



A et B sont verts ou rouges
C est vert, bleu ou rouge
D est bleu ou vert

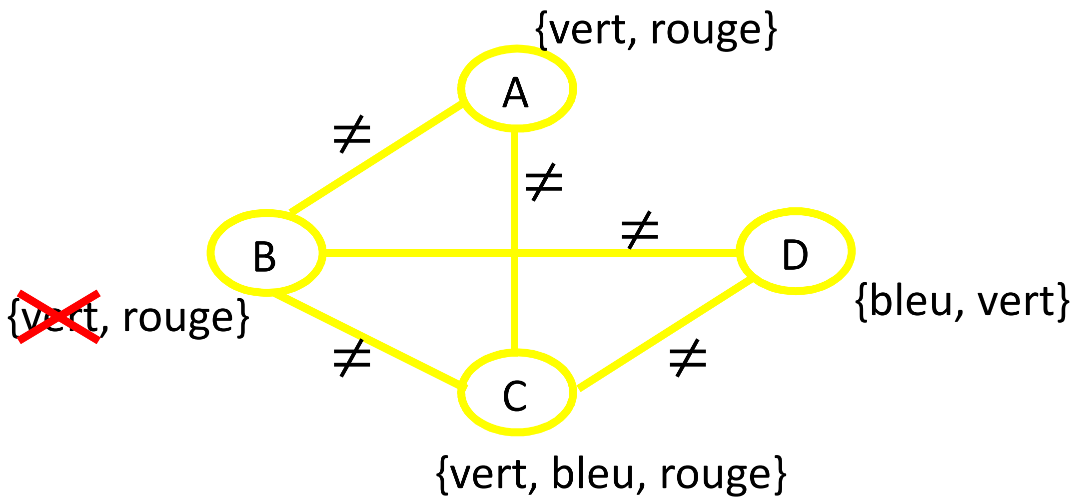
Filtrage par Forward-Checking
Modélisation



- Certains couples de variables sont reliés par une contrainte de différence.
- Le domaine de chaque variable apparaît entre accolades {...}.

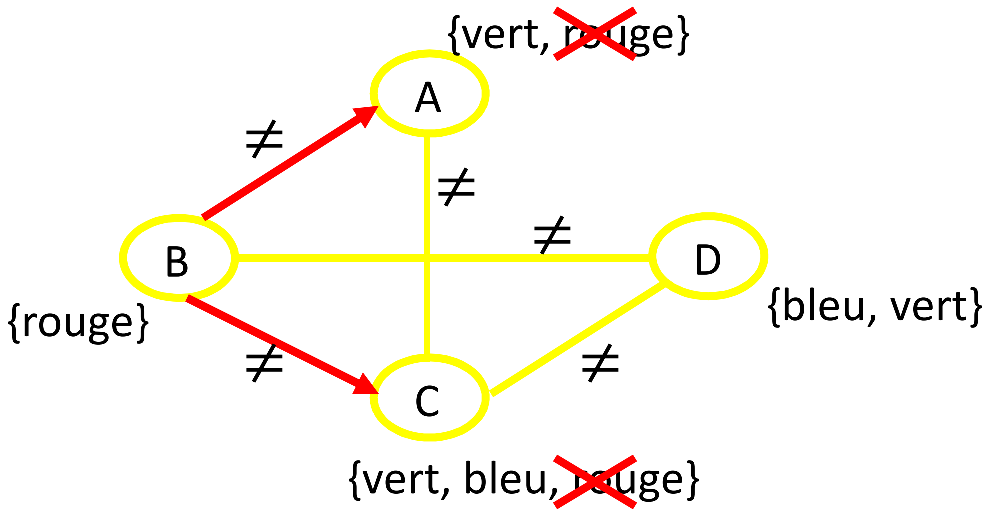
Filtrage par Forward Checking

- Supposons que l'on fixe arbitrairement B à rouge.



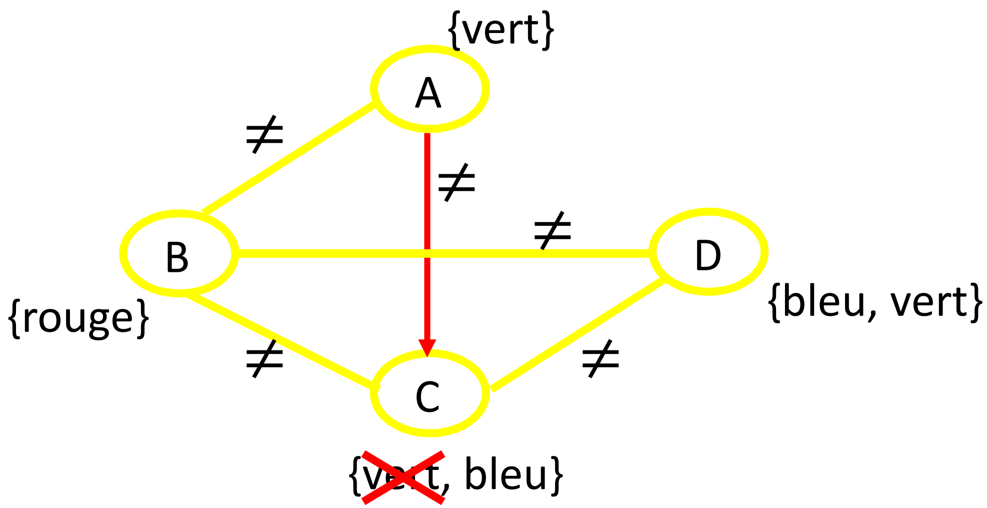
Filtrage par Forward-Checking

- Instantiation de A à vert.



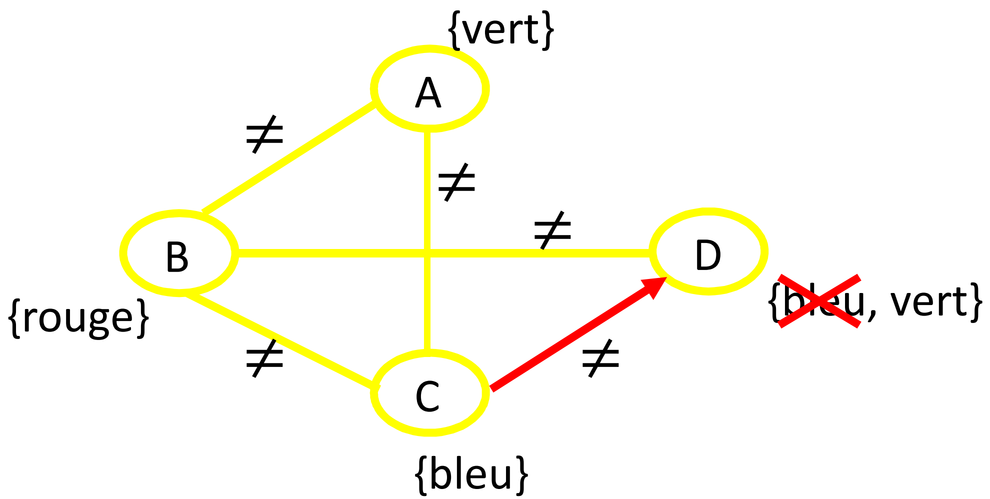
Filtrage par Forward-Checking

- Instantiation de C à bleu.



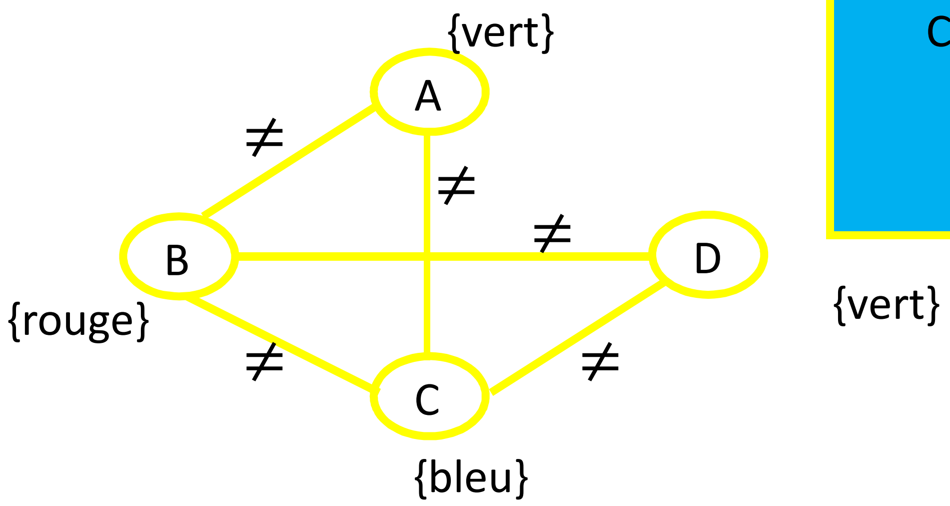
Filtrage par Forward-Checking

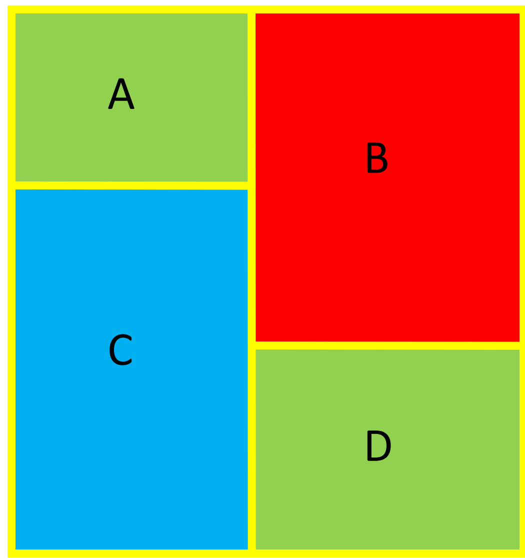
- Instantiation de D à vert.



Filtrage par Forward-Checking

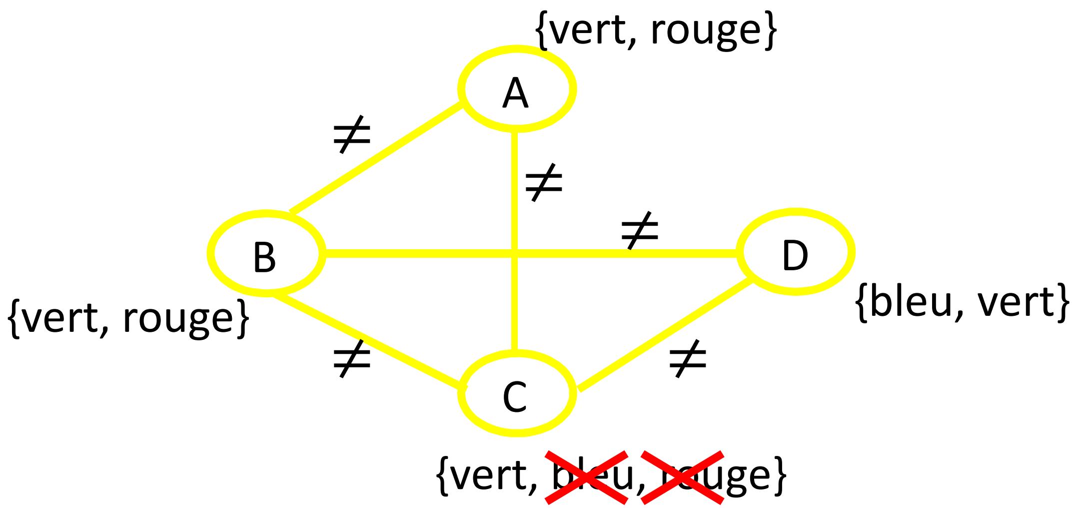
- Solution !





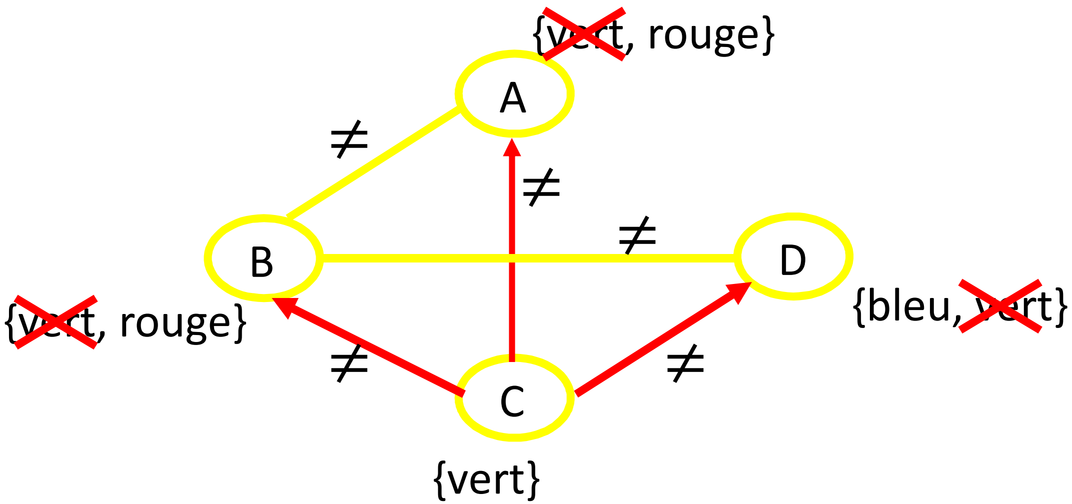
Filtrage par Forward-Checking

- Supposons que l’on fixe arbitrairement C à vert.



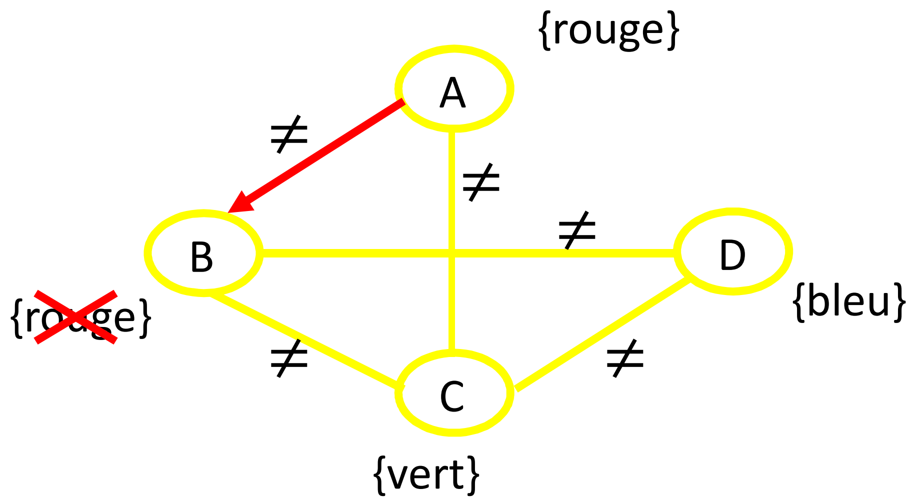
Filtrage par Forward-Checking

- Instantiation de D à bleu, de A et B à rouge.



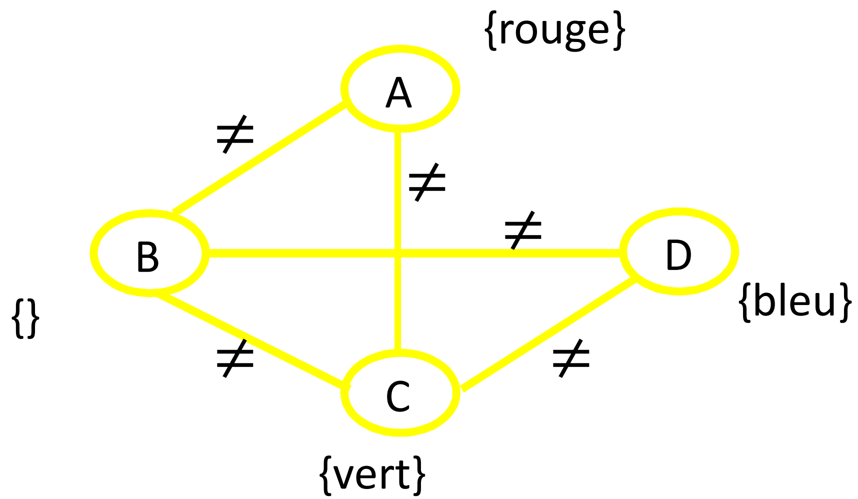
Filtrage par Forward-Checking

- Enlèvement de rouge du domaine de B.



Filtrage par Forward-Checking

- Domaine vide de B : échec !
- Retour-arrière requis dans l’algorithme principal ...



Propagation de contraintes
Consistance d'arc

- Contraintes binaires, avec direction.
- Arc-consistance : l'arc de la variable $\boldsymbol{V_1}$ vers la variable $\boldsymbol{V_2}$ est arc-consistant ssi, pour chaque valeur $\boldsymbol{v_{i1}}$ de $\boldsymbol{V_1}$, il existe au moins une valeur $\boldsymbol{v_{i2}}$ de $\boldsymbol{V_2}$ qui est consistante avec $\boldsymbol{V_1}$.
- Exemple : A { rouge, bleu } <---(≠) ---> B { vert }
- Appliqué de façon répétitive
  - A chaque fois qu'une valeur est enlevée d'un domaine d'une variable (pour enlever une inconsistance d'arc), une nouvelle inconsistance d'arc peut se produire dans les arcs reliés à cette variable.
- Avant une recherche ou après une affectation (Maintaining Arc Consistency)

Propagation de contraintes
***k*-consistance**

- Un CSP est ***k***-consistant ssi, pour tout sous sous-ensemble de (***k*** – 1) variables, et pour toute assignation de ces variables, une valeur consistante peut toujours être assignée à la k-ème variable.
- Exemple :
  - 1-consistance = consistance de nœud
    - Chaque variable prise individuellement est consistante.
  - 2-consistance = consistance d'arc
  - 3-consistance = consistance de chemin

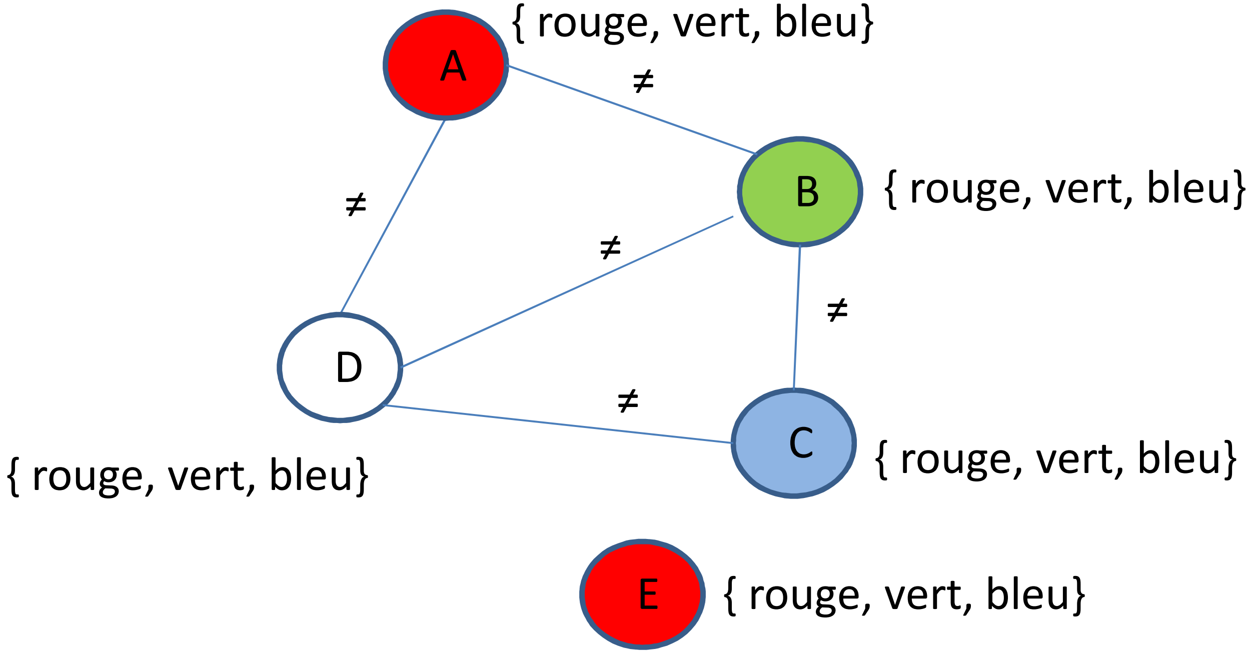
Propagation de contraintes
k-consistance forte

- Un graphe de contraintes est fortement ***k***-consistant ssi il est ***k***-consistant, et (***k*** – 1)-consistant, et (***k*** – 2)-consistant, et …, et ***1***-consistant.
- Si on peut prouver qu’un graphe de contraintes avec ***k*** variables est fortement ***k***-consistant, alors il peut être résolu sans retour arrière.
  - Preuve : Choisir une valeur consistante pour V_1 puisque 1-consistance ; puis on est sûr de trouver une valeur pour V_2, puisque 2-consistance ; … ; puis on est sûr de trouver une valeur pour V_k, puisque ***k***-consistance.
- Un algorithme pour garantir la ***k***-consistance forte prend un temps exponentiel en ***k*** dans le pire des cas.
  - Un algorithme doit il passer beaucoup de temps à établir la consistance et peu à trouver une assignation, ou peu de temps à établir la consistance et beaucoup à trouver une assignation ?
  - Moyen terme entre ***k***-consistance forte et arc-consistance.

Retour-arrière

- L'algorithme RECHERCHE-RETOUR-ARRIERE() précédent remonte à la dernière variable instantiée.
  - Retour-arrière chronologique.
- Backjumping : revenir sur l'une des variables (par exemple, la plus récente) qui est la cause d'un échec, dans l'arbre de recherche.
- L'ensemble de conflit d'une variable **V** est l'ensemble des variables précédemment instantiées, reliés à **V** par une contrainte.

Retour-arrière



- Supposons l’ordre d’instantiation des variables : A, B, C, E, D.
- A = rouge ; B = vert ; C = bleu ; E = rouge
- Pas valeur pour D, donc, avec un retour arrière chronologique, retour arrière sur E !
- L’ensemble de conflit de D est { A, B, C }, donc l’algorithme de backjumping remonte sur C.

Retour-arrière

- Une implémentation du BACKJUMPING : chaque fois que l'algorithme FORWARD-CHECKING, depuis l'affectation de la variable **X**, enlève une valeur du domaine de la variable **Y**, il ajoute **X** à l'ensemble de conflit de **Y**.
- On peut montrer que : toute branche enlevée par BACKJUMPING dans l'arbre de recherche est aussi enlevée par FORWARD-CHECKING.
- BACKJUMPING n'est pas suffisant …

Retour-arrière

- Backjumping dirigé par les conflits (conflict-directed backjumping) :
  - Lors d'une recherche, supposons que la variable $\boldsymbol{V_1}$ a un domaine vide.
  - Backjumping sur la variable la plus récente de l'ensemble des conflits de $\boldsymbol{V_1}$, , soit $\boldsymbol{V_2}$.
  - L'ensemble de conflit de $\boldsymbol{V_2}$ est changé avec celui de $\boldsymbol{V_1}$:
    - Le nouvel ensemble de conflit de $\boldsymbol{V_2}$ est l'ancien plus celui de $\boldsymbol{V_1}$ (moins $\boldsymbol{V_1}$).

Structure du problème
Domaines indépendants

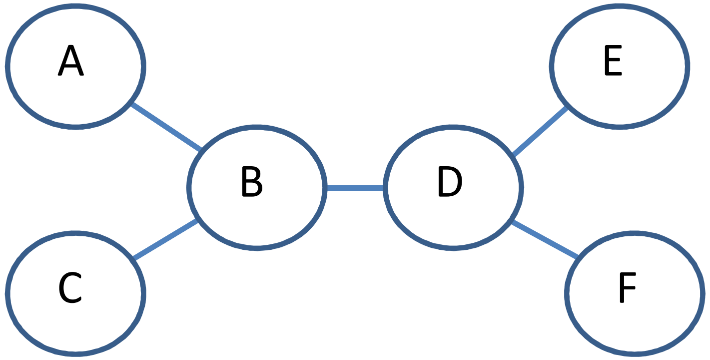
- Indépendance de sous problèmes :
  - Composantes connexes CSP_i d'un CSP : si l'assignation A_i est une solution de CSP_i, $U_i\ A_i$ est une solution de $U_i\ CSP_i$
  - Supposons que chaque CSP_i ait c variables parmi n, chacune ayant un domaine avec d valeurs.
    - n/c sous-problèmes, chacun prenant au plus d^c de travail.
    - <u>Complexité en temps :</u> $O(n/c * d^c)$ soit linéaire en n.
  - A comparer avec la complexité de CSP : $O(d^n)$
    - Exemple : un CSP booléen (d = 2) avec 80 variables (n = 80) et 4 sous-problèmes (c = 20).

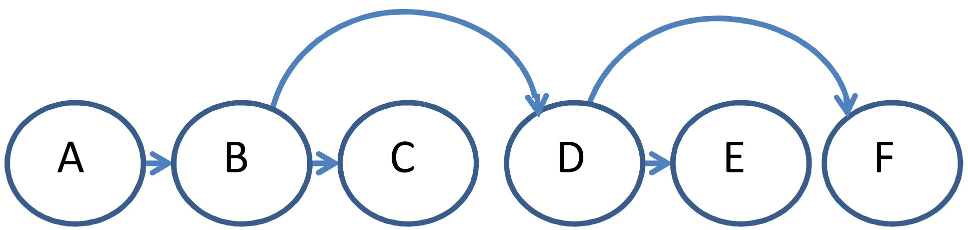
      Pire des cas pour ce CSP indépendant = $10^{6,62}$ (environ 1s)

      Pire des cas pour ce CSP = 10^{24} (920 milliards d'années)

Structure du problème
Un arbre

- L'ensemble des contraintes est un arbre.
  - Pour toute paire de variables (***V****i*, ***V****j*),
    V*i* et ***V****j* sont reliées par au plus un chemin.



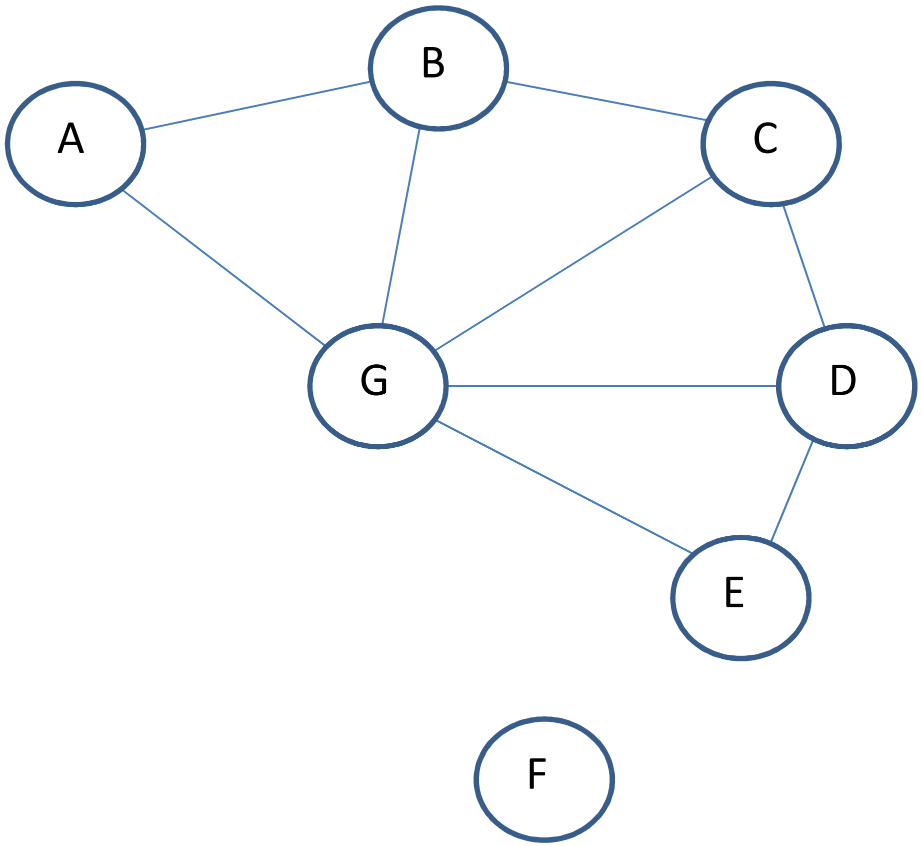


Structure du problème
Un arbre

- Etapes de l'algorithme ARBRE(csp):
  1. Choisir une variable racine, et trier les variables en un ordre compatible avec l'arbre (soit $\boldsymbol{V_1}, \boldsymbol{V_2}, \ldots, \boldsymbol{V_n}$), de façon que le parent de la variable $\boldsymbol{V}_{\mathrm{i}}$ soit avant elle.
  2. POUR ***j*** de *n* à 2, appliquer l'arc-consistance à chaque arc ($\boldsymbol{V_i}$, $\boldsymbol{V_j}$) avec $\boldsymbol{V_i}$ le parent de $\boldsymbol{V_j}$ (en enlevant des valeurs de Domaine($\boldsymbol{V_i}$))
  3. POUR ***j*** de 1 à *n*, assigner n'importe quelle valeur à $\boldsymbol{V_j}$ consistante avec celle de son parent $\boldsymbol{V_i}$.
- <u>Complexité en temps :</u> $O(nd^2)$

  Si un ensemble de contraintes est un arbre, alors il peut être résolu par un algorithme de complexité linéaire en temps.

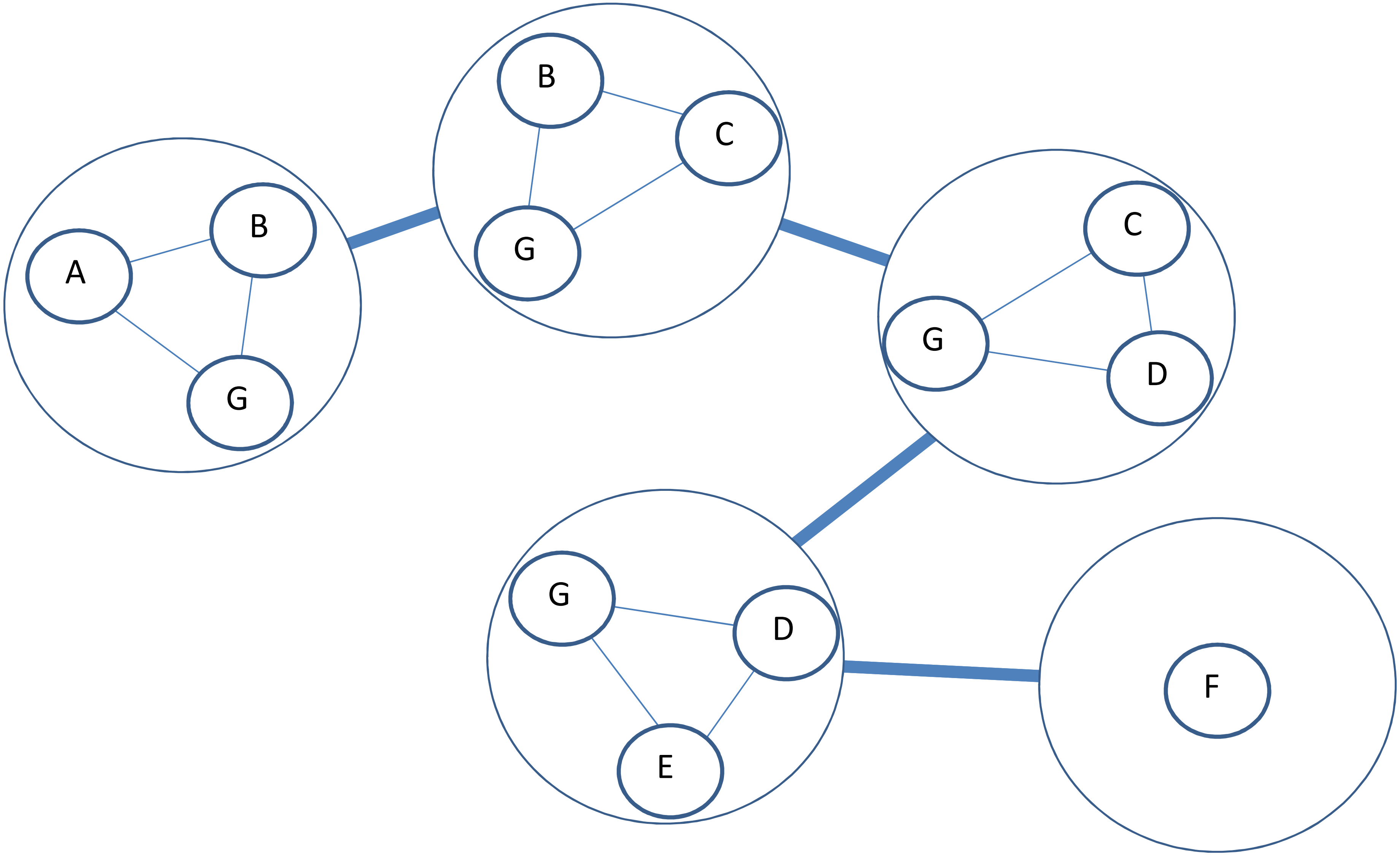
Structure du problème
Se ramener à un arbre : enlever des variables



Structure du problème
Se ramener à un arbre : enlever des variables

- Etapes de l'algorithme ENSEMBLE-COUPES(**csp**)
  1. CHOISIR un sous ensemble ***S*** de *Variables(**csp**)*, tel que la structure des contraintes devient un arbre après enlèvement de ***S***.
  2. POUR TOUTE assignation ***A*** de variables dans ***S*** qui satisfait les contraintes dans ***S***
     a) Enlever du domaine des variables restantes toutes les valeurs inconsistantes avec ***A***
     b) SI le CSP restant à une solution, ALORS la retourner avec ***A***.
- Soit $c = Card(S)$.
  – Complexité en temps : $O(d^c\ (n - c)d^2)$
  – Trouver le plus petit ensemble S est difficile …

Structure du problème
Se ramener à un arbre : décomposition arborescente



Structure du problème
Se ramener à un arbre : décomposition arborescente

- Pré-requis pour une décomposition arborescente :
  - Toute variable du problème apparait dans au moins un sous problème
  - Si V_i et V_j sont reliées par une contrainte, alors V_i et V_j (et la contrainte) apparaissent ensemble dans au moins un sous problème.
  - Si V_i apparait dans 2 sous problèmes de l'arbre, alors V_i apparait dans le chemin entre ces 2 sous problèmes.
- Etapes de l'algorithme :
  - Résoudre chaque sous problème SI pas de solution, ALORS pas de solution pour le problème global.
  - Appliquer ARBRE() sur les macro-variables (sous-problèmes, assignations).
- Complexité :
  - Largeur de l'arbre :

    $w = \min_{\text{décomp.}} (\max_i(\text{taille d'un sous problème } CSP_i \text{ selon } \textit{décomp.}) - 1)$
  - Complexité : $O(nd^{w+1})$
  - Mais trouver une décomposition de largeur minimale est difficile …

Conclusion

- La programmation par contraintes est un paradigme pour résoudre des problèmes combinatoires.
- Décomposition en un modèle (variables, domaines, contraintes) et un algorithme.
- L'algorithme utilise un filtrage avant et un retour-arrière / saut arrière.
- Formes particulières des contraintes : indépendance, arbre, coupes, mega-variables.