Constraint Programming

Philippe Morignot
philippe.morignot@vedecom.fr

Table of Contents

General Concepts

- Constraint Programming is a paradigm for solving combinatorial problems.
- Other approaches:
  - Mixed Integer Programming
  - Evolutionary algorithms
  - Search algorithms in a state space (e.g., blind search, heuristic search such as A*).
  - Simulated annealing
  - Taboo algorithm
  - …

General Concepts

- A combinatorial problem is a problem …
  - … which can be modelled as entities …
  - … maintaining relations …
  - … and in which a solution must be found:

    the solution to the problem.
- There can be several solutions.
    - Finding one
    - Finding the best one

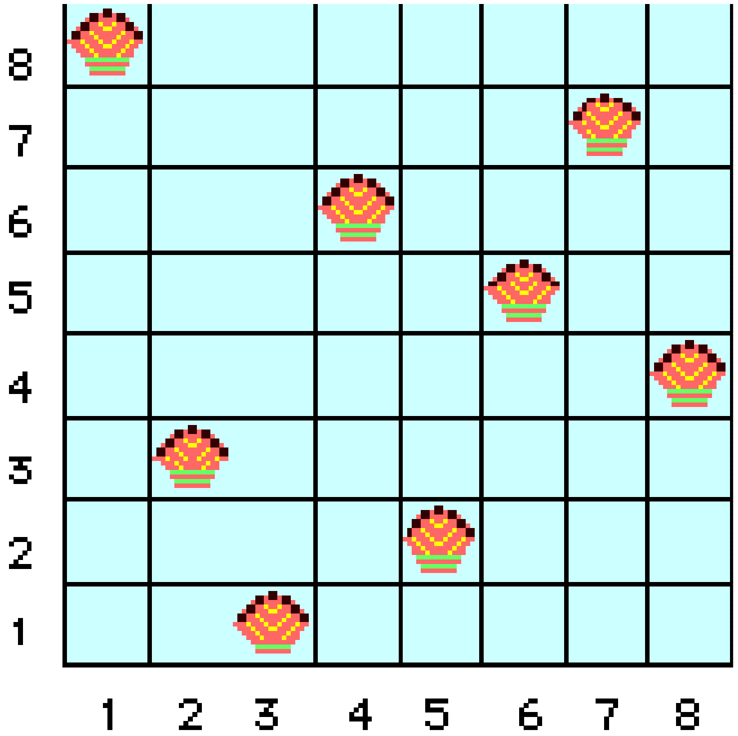
General Concepts

- Example of a combinatorial problem: the sudoku game.

		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	

General Concepts

- Example of a combinatorial problem: the N-queen problem (here, N = 8).



General Concepts

- Example of combinatorial problem: cryptarithmetics.

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

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

General Concepts

- The difficulty: the number of combinations to consider can be gigantic for a real world problem.
  - Example: for the sudoku game, a coarse estimation of the number of combinations is :

    $(8!)^9 \approx 10^{41}$ combinations.
  - For small combinatorial problems, (almost) every algorithm should work …
- Consequence : browsing all these combinations one by one would take a gigantic time, even on the fastest computer.
  - Phenomenon of combinatorial explosion.
  - In the worst case, the number of combinations to consider is an exponential function of the size of one dimension of the data.

General Concepts

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

MUCH TOO LONG !!!!

(except on small problems)

General Concepts

- Idea of constraint programming:
  - Consider the structure of the problem: Decompose the problem as
    - Variables
      - Each variable has a finite domain (variable expressed in extension).
    - Relations among variables (constraints)
      - A constraint must always be satisfied; It reduces the variables' domains.
  - A unique algorithm cleverly uses this model.
    - Heuristics

General Concepts

- Other example of split between model and solver:
  – Fact base / rule base and inference engine in knowledge-based systems.
  – (Mixed Integer) Linear Programming in Operational Research.
- Consequence:
  – A user writes a model and gives heuristics.
  – The Constraint Programming engine searches for a solution.

General Concepts

- Examples of problems « solvable » by CP:
  - Assign companies to interns while following their wishes.
  - Optimally plan air traffic (assign flight corridors to aircrafts, optimize flight crew shifts, …)
  - Schedule tasks, so that the makespan is minimized and the resources consumption minimized.
  - Optimize component location on a mother board.
  - Build a menu both healthy and tasty in a restaurant.
  - …

General Concepts

- Constraint Programming has been proposed by Jean-Louis Laurière in 1976 in his HdR.
- Seminal Publication:
  - Jean-Louis Laurière, A Language and a Program for Stating and Solving Combinatorial Problems. Artif. Intell. 10(1): 29-127 (1978).
- Packages : CPLEX CP Optimizer from IBM, CHOCO from EMN, CHIP from COSYTEC, SICSTUS PROLOG, MINIZINC+ solvers, etc.
- French Association for Constraint Programming: http://afpc.greyc.fr/web/

Model

- Discrete variables with finite domain:
  - For *i* from 1 to n, a variable V_i
  - For *j* from 1 to n, a domain $D_j = \{ v_1, v_2, \ldots, v_{f(j)} \}$.
  - For all *i*, $V_i \in D_i$
- Constraints on these variables:
  - For *k* from 1 to m, $C_k = (X_k, R_k)$ with :
    - $X_k = \{ V_{i1}, V_{i2}, \ldots, V_{ik} \}$ // The variables of C_k
    - $R_k \subset D_{i1} \times D_{i2} \times \ldots \times D_{ik}$ // The possible values of these // variables, together compatible // with C_k

Vocabulary

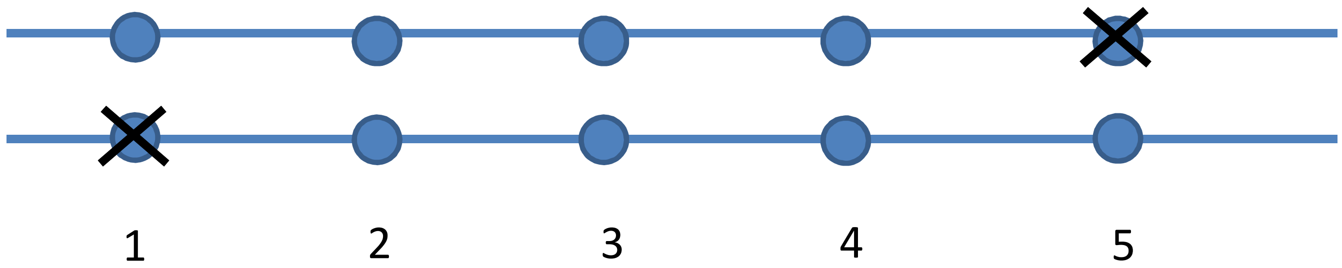
- A CSP might be:
  - Under constrained: too few constraints.
  - Over constrained: too many constraints.
- Given a CSP, one can …:
  - Search for a solution
  - Search for all solutions
  - Search for an optimal solution given some cost function
  - Proove that there is no solution.

Constraints

- A constraint can be expressed:
  - In extension: give the sets of possible values of variables
  - Arithmetically: <, ⩽, >, ⩾, =, ≠, +, -, /, *, …
  - Logically: ⇔, =>, <=, OU, ET, NON, …
  - Globally: AllDifferent(x_1, x_2, …, x_n), Geost(f_1, f_2, …, f_n), ...
- A constraint can be:
  - Hard: a constraint must always be satisfied.
  - Soft: a constraint is sometimes satisfied and sometimes violated, given a criterion
- A contrainte can be:
  - Unary. Example : $x \in [1, 5]$
  - Binairy. Example : $x < y$
  - N-ary. Example : AllDifferent(V_1, V_2, …, V_n)

Constraints

- Example of hard constraints:
  - $x \in [1, 5]$; $y \in [1, 5]$; $x < y$
  - x :
  - y :



- Example of soft constraints:
  - In a scheduling problem,

    $Y = \#(t_i < \text{deadline}_i)$ and maximize Y

Constraints
Global constraints

- **AllDifferent**(V_1, V_2, …, V_n)
  - The variables V_i must be all different.
  - Logically equivalent to:

$$V_1 \neq V_2 \wedge V_1 \neq V_3 \wedge \ldots \wedge V_1 \neq V_n \wedge$$
$$V_2 \neq V_3 \wedge \ldots \wedge V_2 \neq V_n \wedge$$
$$\ldots \wedge$$
$$V_{n-1} \neq V_n$$

- <u>Property:</u> if there are **m** variables in AllDiff, and **n** distinct values together possible, and if **m** > **n**, then the constraint cannot be satisfied.

Example of model

- A model for the sudoku game:
  - A variable is an empty cell in the grid.
  - A domain is the set of integers from 1 to 9
    - If the cell already includes a number, it appears as a constant in the constraint.
  - Constraints:
    - The variables of a small grid are all different and different from constants.
    - The variables of a row are all different.
    - The variables of a column are all different.

Example of model

- A 1st model for the N-queen problem (here, N = 8) :
  - A pair of variables (***x_i***, ***y_i***) per queen ***i***. The queen ***i*** is located on colum ***x_i*** and on row ***y_i***
  - The domain of ***x_i*** is [1, 8]
  - The domain de ***y_i*** is [1, 8]
  - Constraints:
    - $x_i \neq x_j$ // Different columns
    - $y_i \neq y_j$ // Different rows
    - $x_i + y_i \neq x_j + y_j$ // Different 1st diagonal
    - $x_i – y_i \neq x_j – y_j$ // Different 2nd diagonal

Example of model

- A 2^{nd} model for the N-queen problem (here, N = 8) :
  - The variable $\boldsymbol{x_i}$ is the row of the ***i***-th column on which the i-th queen is located.
  - The domain of $\boldsymbol{x_i}$ is [1, 8]
  - Constraints:
    - The constraints on columns are satisfied by construction.
    - $x_i \neq x_j$ // Different rows
    - $x_i + i \neq x_j + j$ // Different 1^{sr} diagonals
    - $x_i - i \neq x_j - j$ // Different 2^{st} diagonals

Example of model

- A 3rd model for the N-queen problem (here, N = 8) :
  - The cells of the grid are numbered from 1 to 64.
  - The variable $\boldsymbol{x_i}$ is the index of the cell in this numbering at which queen i is located.
  - Constraints:
    - $x_i / 8 \neq x_j / 8$ // Different rows
    - $x_i \% 8 \neq x_j \% 8$ // Different columns
    - Constraints on the 1^{e} diagonal
    - Constraints on the 2^{e} diagonal

Example of model

- <u>Cryptarithmetics: SEND + MORE = MONEY</u>
- The model:
  – Variables : S, M ∈ [1, 9] ; E, N, D, O, R, N, Y ∈ [0, 9]
  – Constraints :
    - D + E = Y + 10 * R1
    - N + R + R1 = E + 10 * R2
    - E + O + R2 = N + 10 * R3
    - S + M + R3 = O + 10 * M
    - Secondary variables : R1, R2, R3 ∈ [0, 1]

Vocabulary

- An **assignment** is the fact of assigning a variable to a value of its domain.
  - Variable V_i is assigned to its value v_{ij} : $D_i = \{ v_{ij} \}$
- An **assignment** of variables to values is :

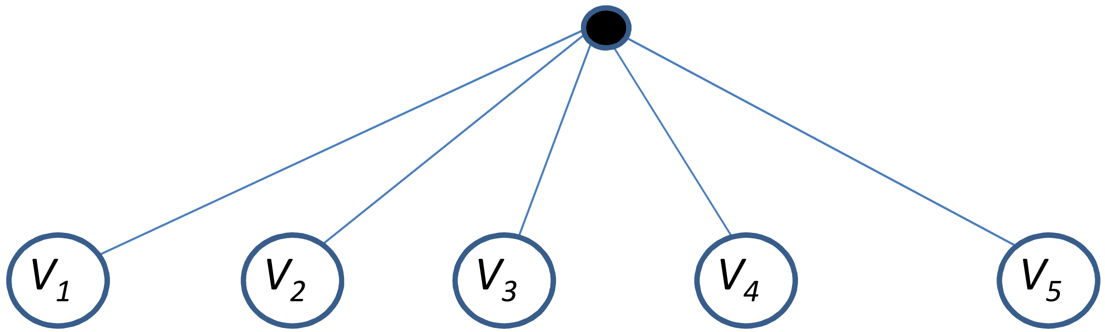
  $A = \{(V_{i1}, v_{i1}), (V_{i2}, v_{i2}), \dots, (V_{ik}, v_{ik})\}$
- An assignment can be :
  - **total** : every variable has a value (k = n)
  - **partial** : some variables have a value, but not all (k < n).
- An assignment A is **consistant** iff it does not violate any constraint C_k.
- A **solution** to a CSP (Constraint Satisfaction Problem) is a total consistant assignment of variables.
- Some CSPs require to maximize an objective function f.

Algorithms

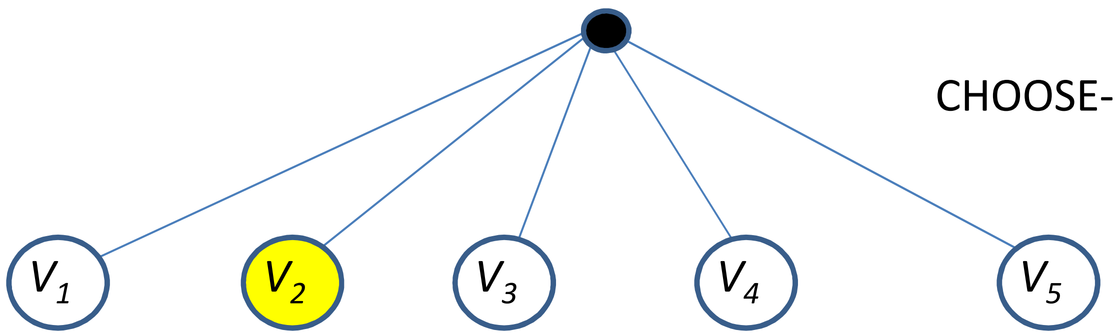
- Commutativity :
  - A problem is commutative iff the order of application of actions has no effect on the result.
- A CSP is commutative : when values are assigned to variables, the same partial assignment is reached whatever the order is.
- Consequence: we can assign variables one after the other, as needed.

Algorithms : backtrack (1 / 14)

Algorithms : backtrack (2 / 14)

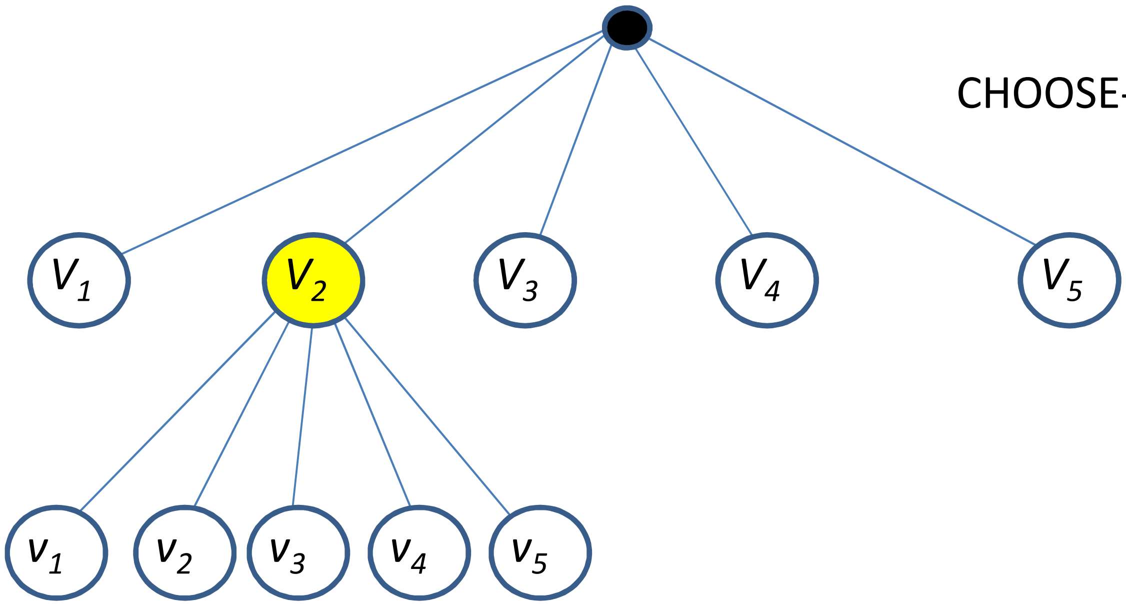


Algorithms : backtrack (3 / 14)



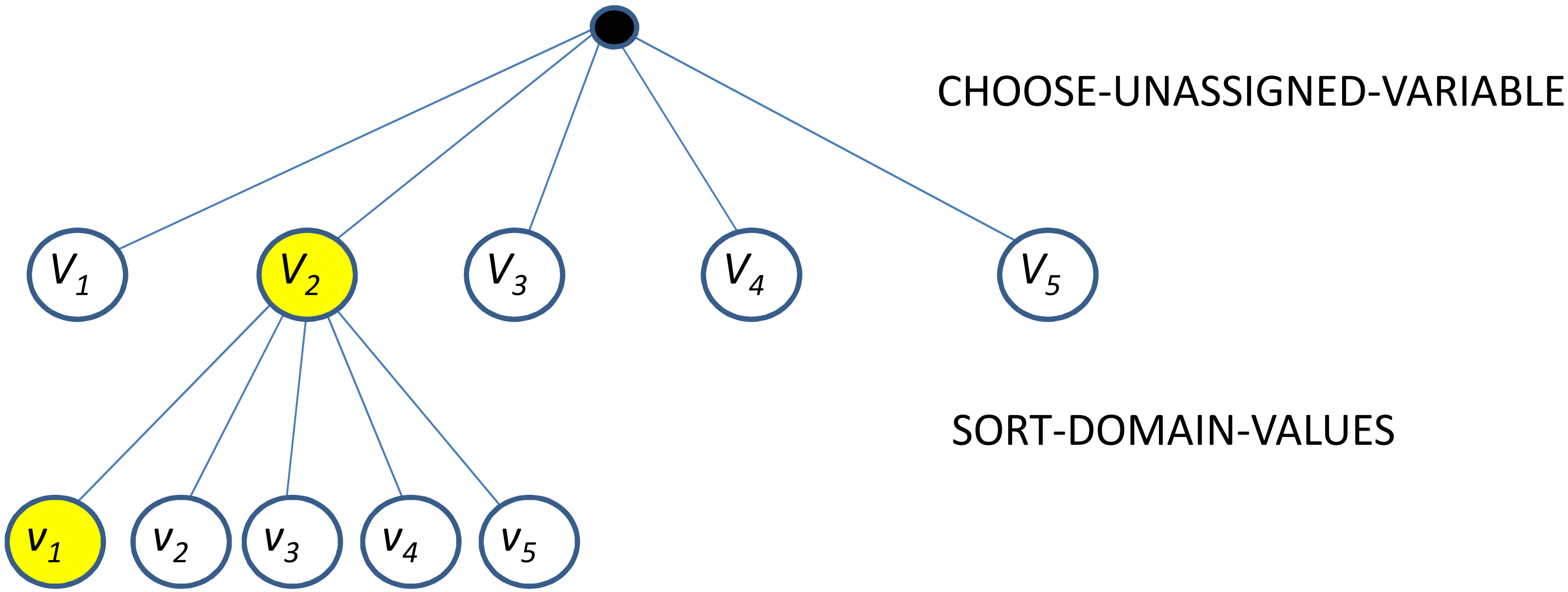
CHOOSE-UNASSIGNED-VARIABLE

Algorithms : backtrack (4 / 14)

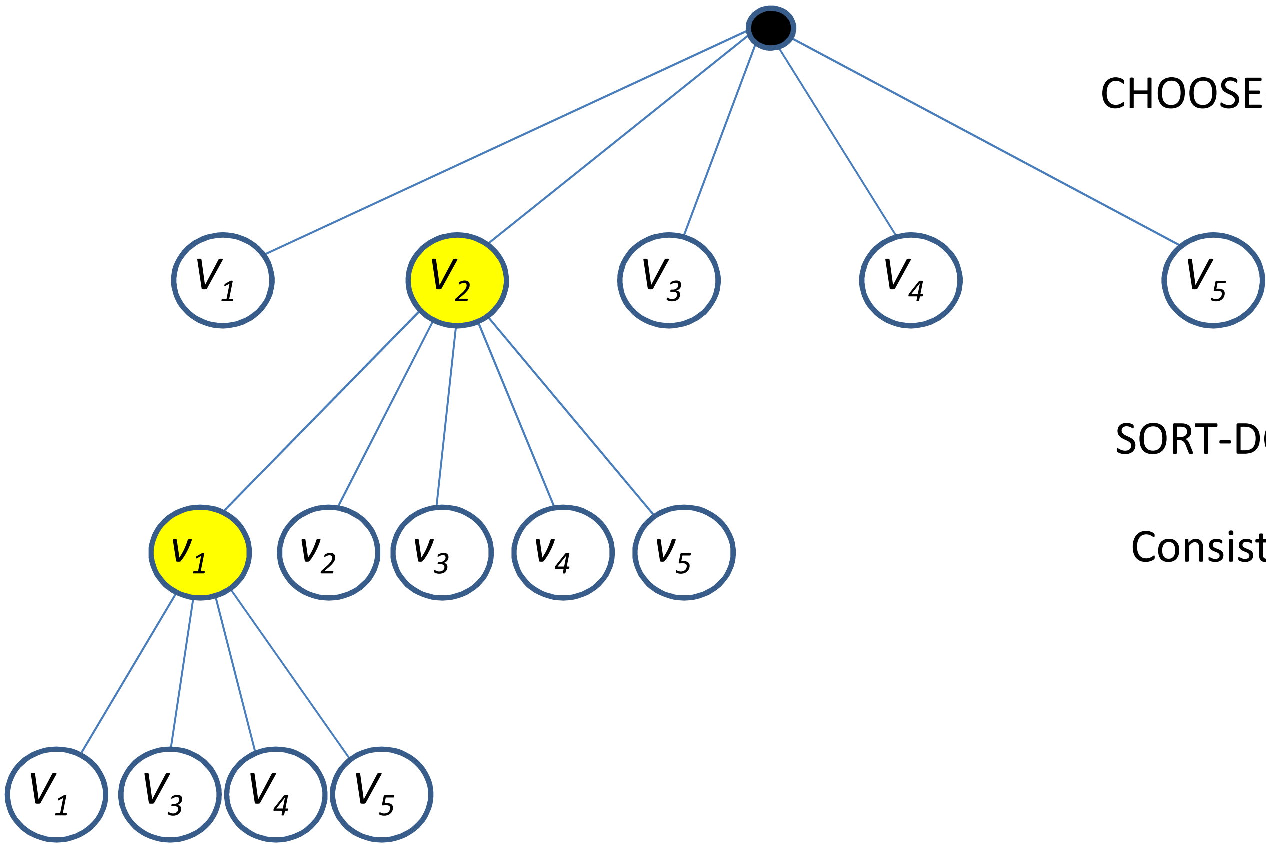


CHOOSE-UNASSIGNED-VARIABLE

Algorithms : backtrack (5 / 14)



Algorithms : backtrack (6 / 14)

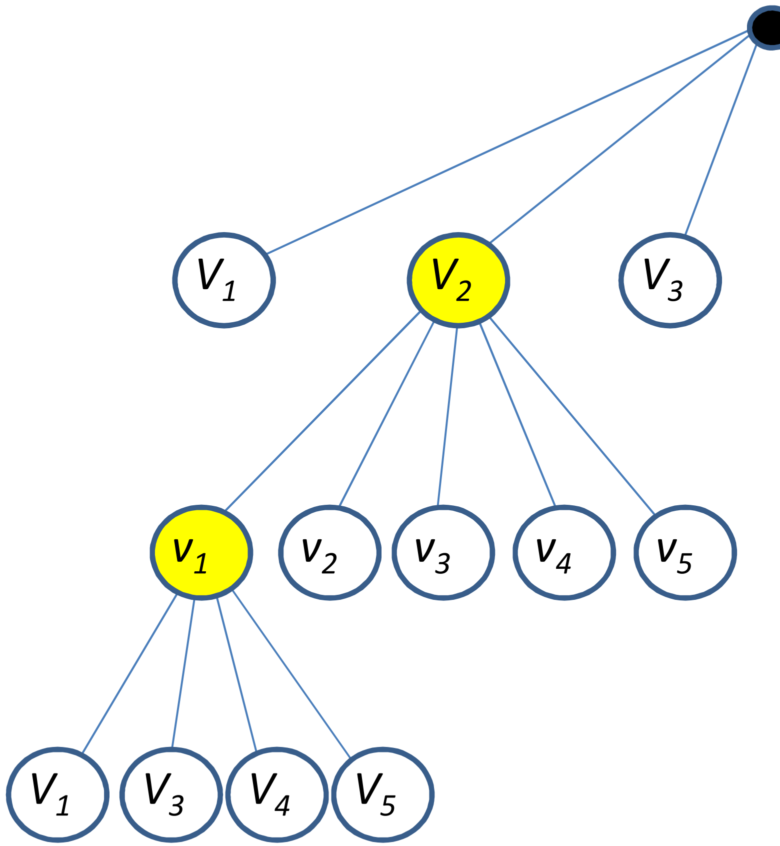


CHOOSE-UNASSIGNED-VARIABLE

SORT-DOMAIN-VALUES

Consistancy checks

Algorithms : backtrack (7 / 14)



CHOOSE-UNASSIGNED-VARIABLE

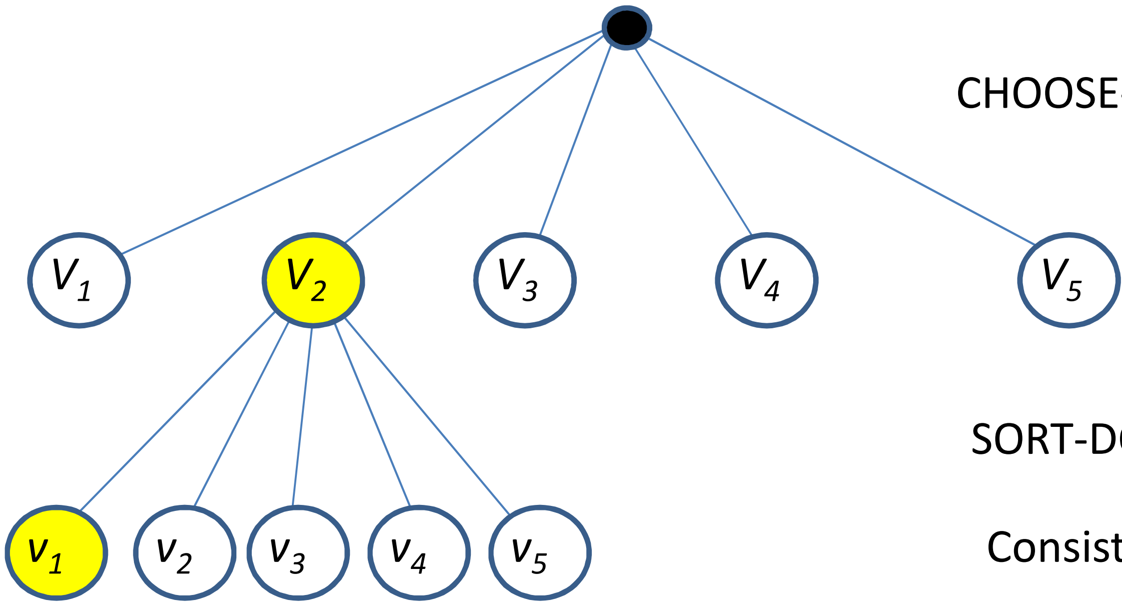
SORT-DOMAIN-VALUES

Consistancy checks

One domain becomes empty !

BACKTRACK

Algorithms : backtrack (8 / 14)

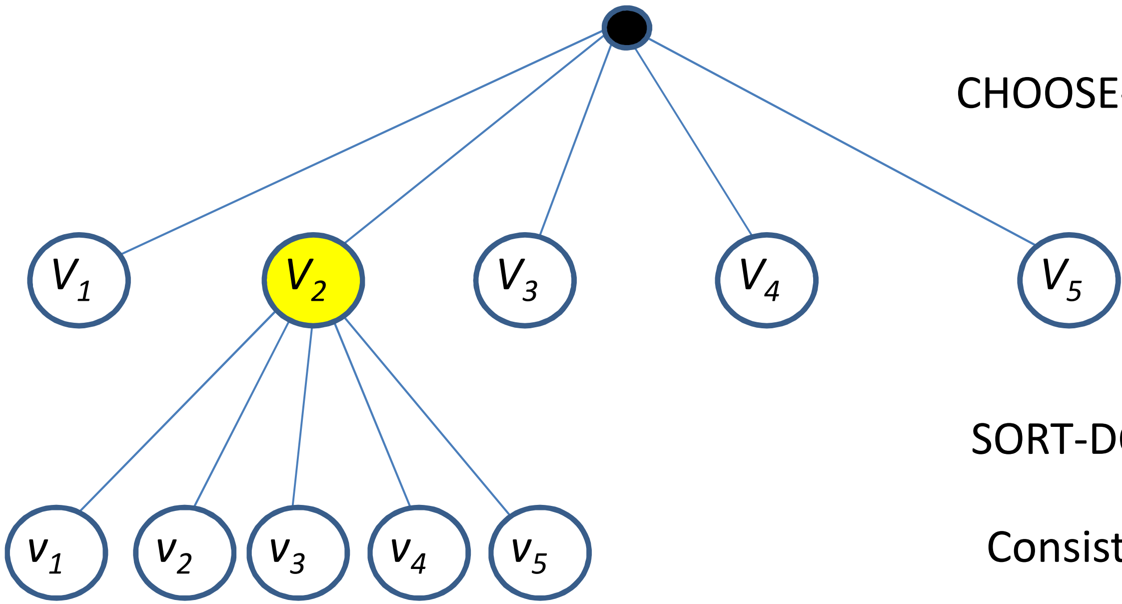


CHOOSE-UNASSIGNED-VARIABLE

SORT-DOMAIN-VALUES

Consistancy checks

Algorithms : backtrack (9 / 14)

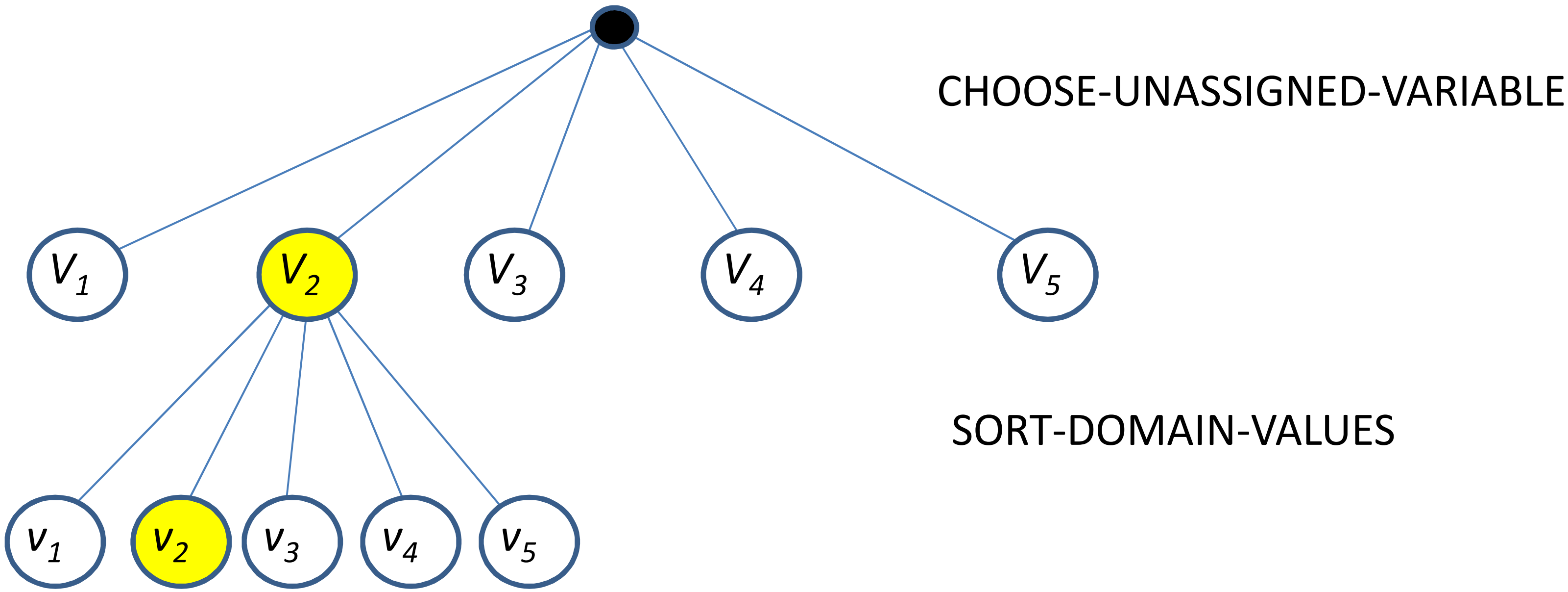


CHOOSE-UNASSIGNED-VARIABLE

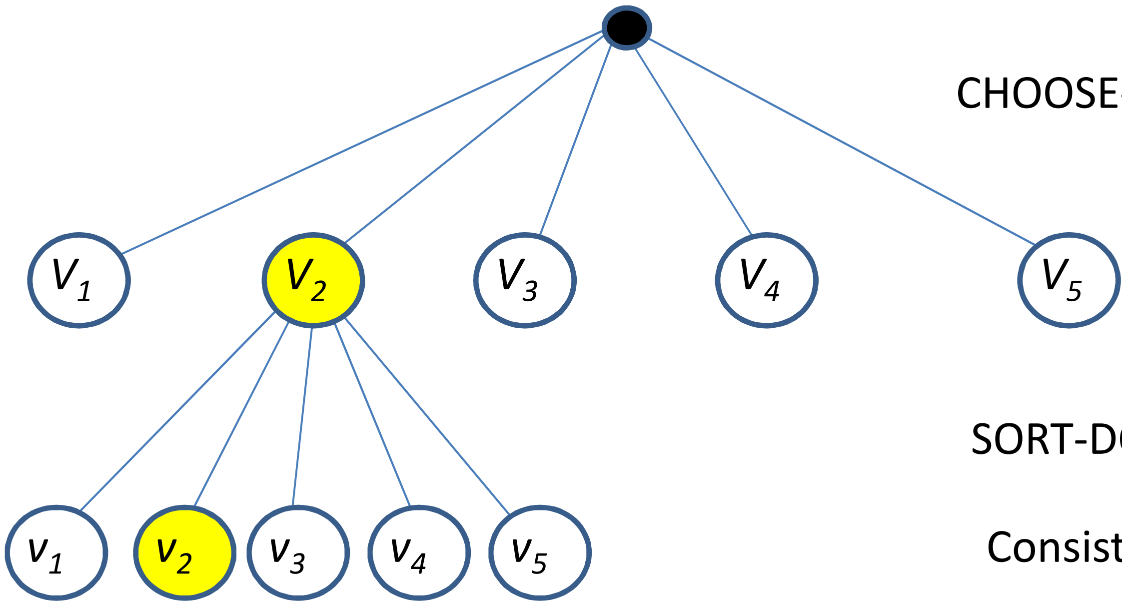
SORT-DOMAIN-VALUES

Consistancy checks

Algorithms : backtrack (10 / 14)



Algorithms : backtrack (11 / 14)

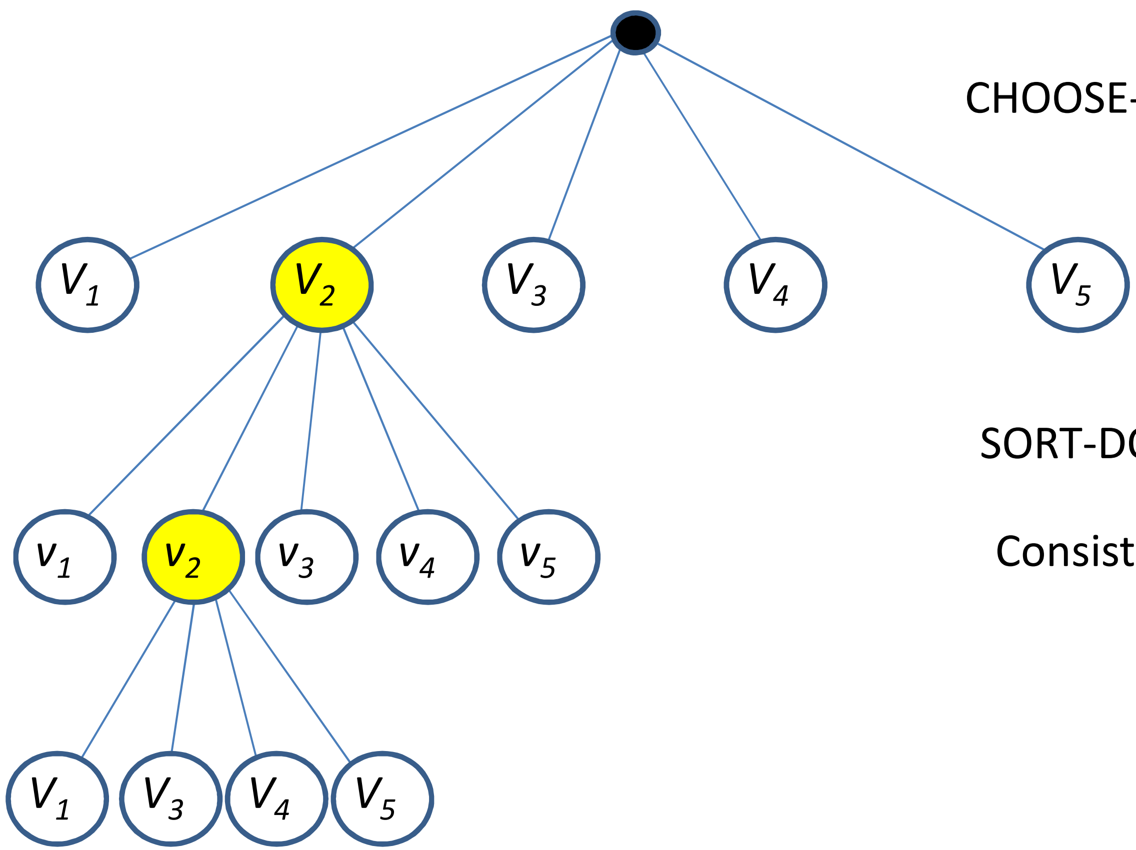


CHOOSE-UNASSIGNED-VARIABLE

SORT-DOMAIN-VALUES

Consistancy checks

Algorithms : backtrack (12 / 14)

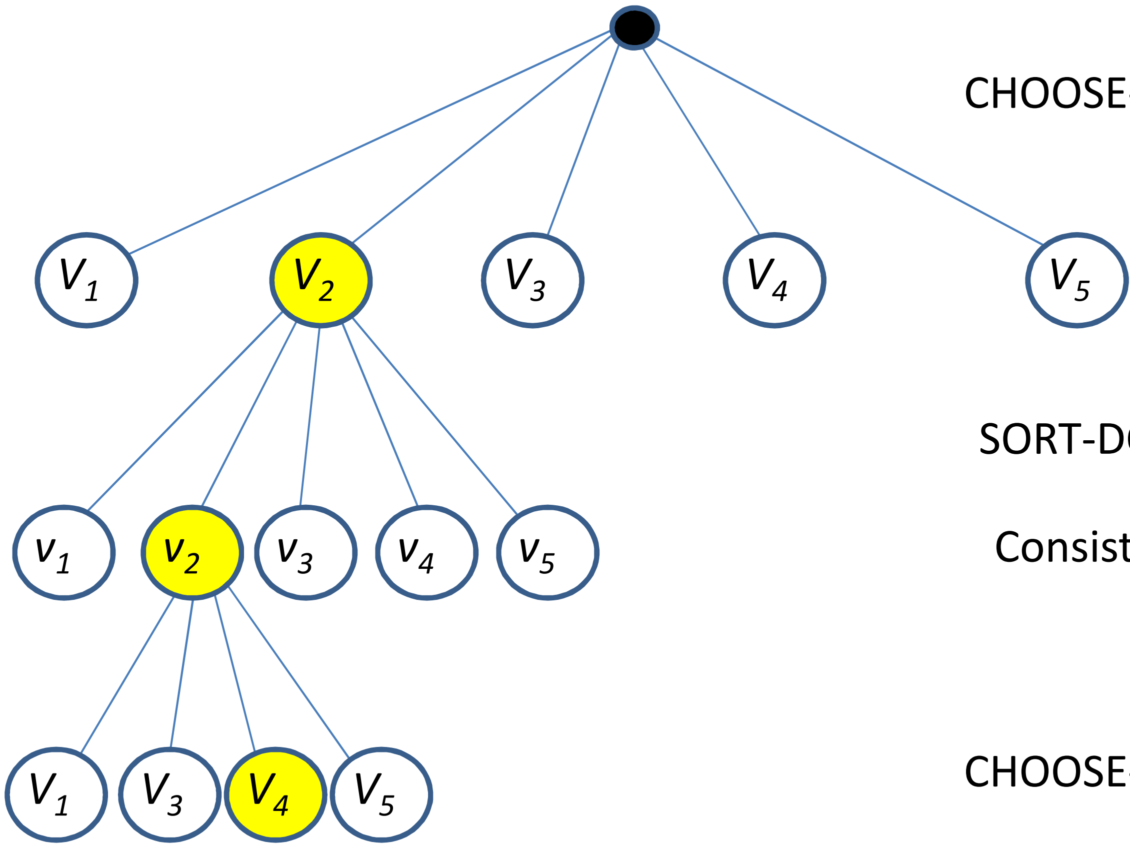


CHOOSE-UNASSIGNED-VARIABLE

SORT-DOMAIN-VALUES

Consistancy checks

Algorithms : backtrack (13 / 14)



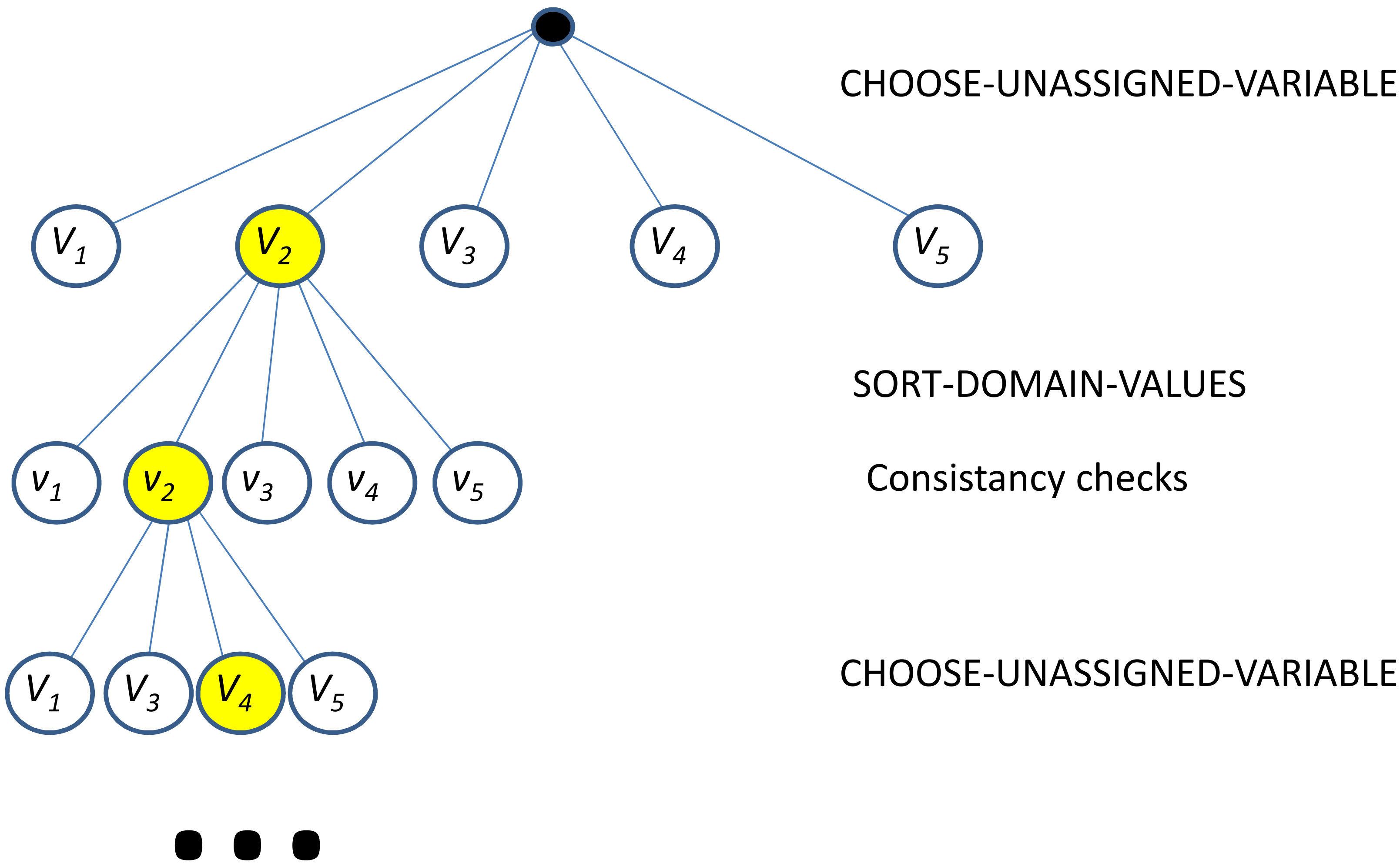
CHOOSE-UNASSIGNED-VARIABLE

SORT-DOMAIN-VALUES

Consistancy checks

CHOOSE-UNASSIGNED-VARIABLE

Algorithms : backtrack (14 / 14)



Heuristics (1 / 2)

- A heuristics can make a choice.
  - Expressed in terms of variables, domaines and constraints.
  - Or can be based on exogeneous information (e.g., the application domain of the CSP).
  - Prototype in C++ : int heuristics(Assignment* assignment, Csp* csp);

- **Heuristics on variables :** CHOOSE-UNASSIGNED-VARIABLE()
  - Static / dynamic.
  - **First-fail** : choose the variable with the smallest domain.
  - **Smallest** : choose the variable with the smallest value in its domain.
  - **Constraint**: choose the variable linked to the maximum number of constraints.
  - …

Heuristics (2 / 2)

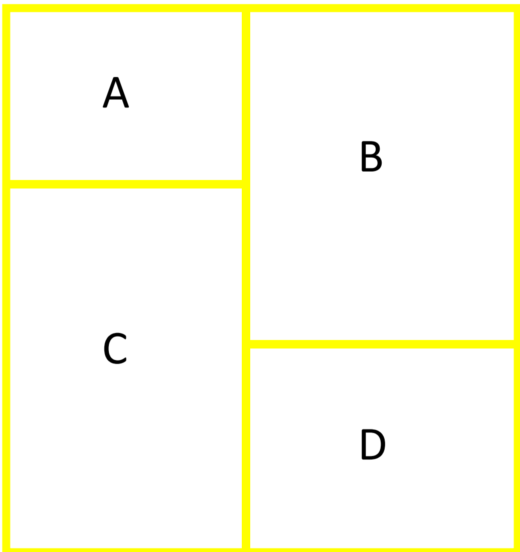
- **<u>Heuristics on values :</u>** SORT-DOMAIN-VALUES()
  - Static / dynamic
  - **Min** : assign a variable to its minimum value
  - **Max** : assign a variable to its maximum value
  - **Median** : assign a variable to its median value; or different from its median value (middle-out)
  - **Split** : constrain the variable to its lower/upper half domain
  - **Regret** : assign the variable to the value which removes the least number of values to other variables.
  - …
- **<u>Search strategy :</u>** additional constraints which orient search.

Filtering

- What does the assigment of a variable imply for other variables ?
- **FORWARD-CHECKING:** each time a variable $\boldsymbol{V_i}$ is instantiated, consider variables $\boldsymbol{V_j}$ connected to $\boldsymbol{V_i}$ by a constraint $\boldsymbol{C_k}$, and remove from the domain of variable $\boldsymbol{V_j}$ the values which are inconsistant with constraint $\boldsymbol{C_k}$.

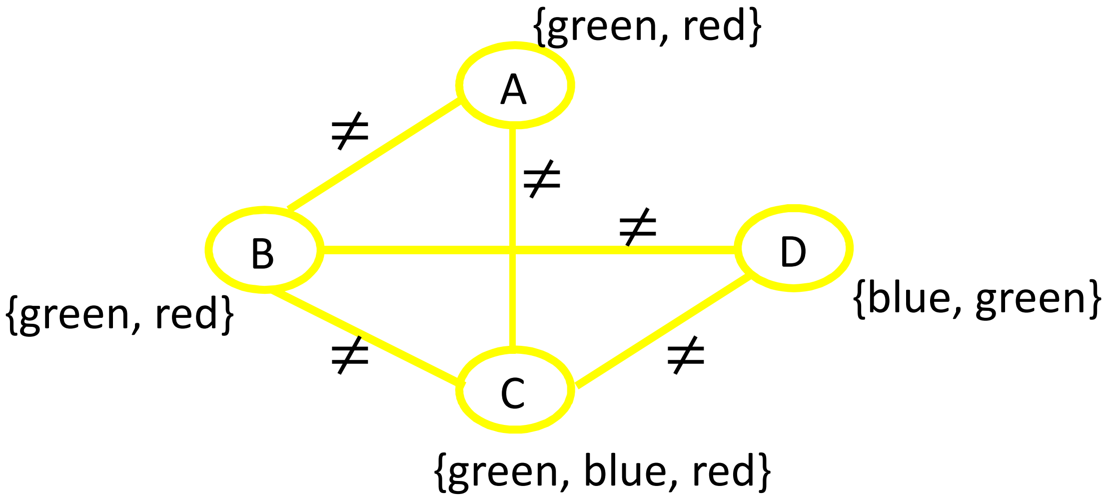
Filtering by Forward-Checking
Graph coloring

Assign a color to A,B,C and D (below) so that no two colors are adjacent, with possible colors for each rectangle:



A and B are green or red
C is green, blue or red
D is blue or green

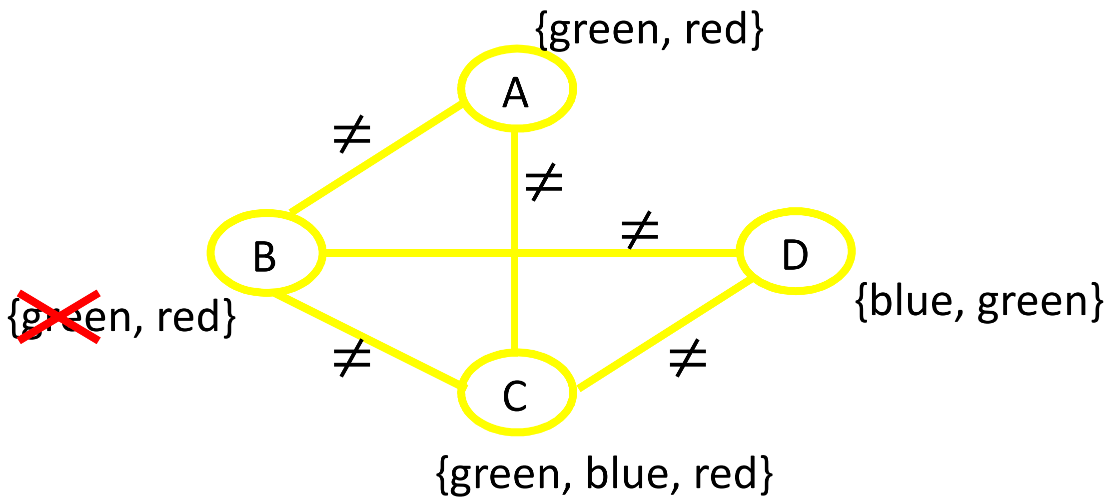
Filtering by Forward-Checking Model



- Some pairs of variables are linked by a constraint of difference
- The domain of each variable is shown in bracket {...}.

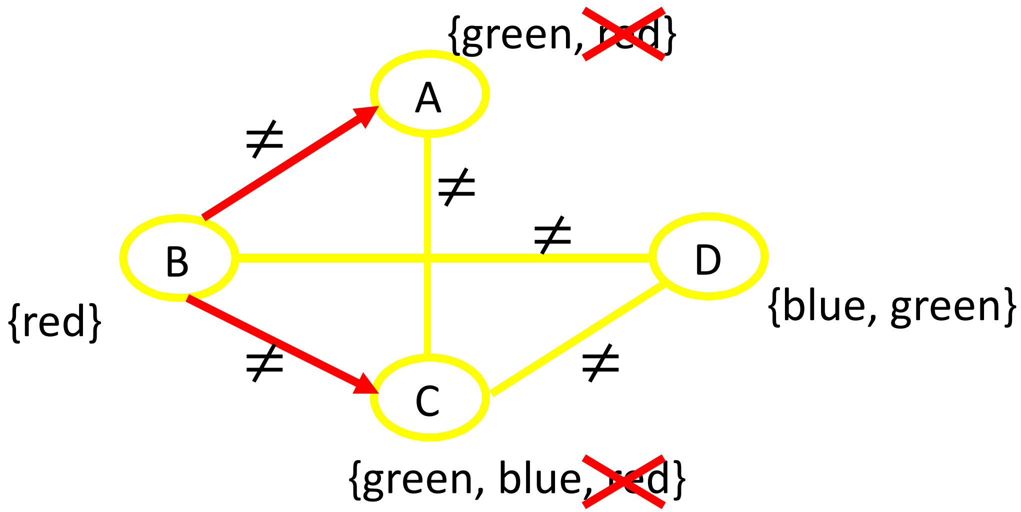
Filtering by Forward-Checking

- Let us assume B is arbitrarily assigned to red.



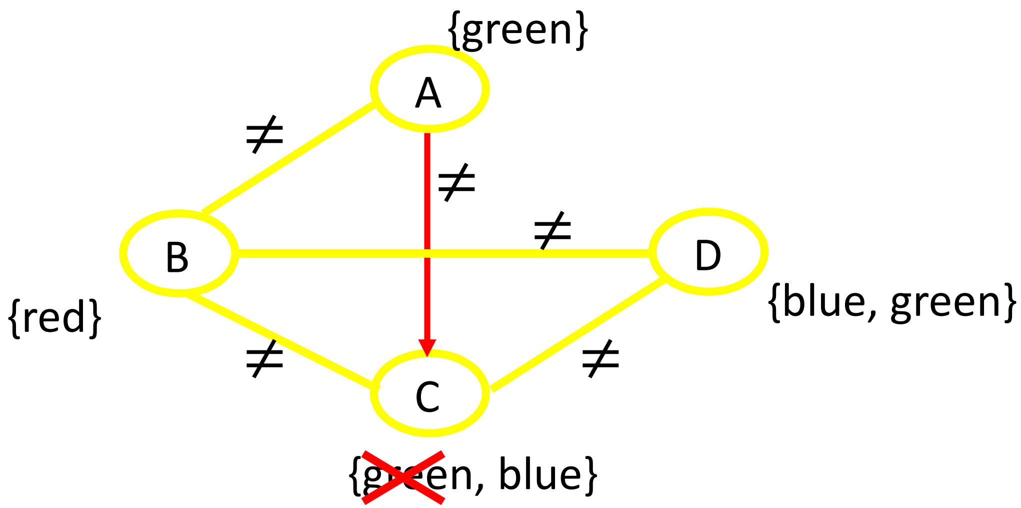
Filtering by Forward-Checking

- Instantiation of A to green.



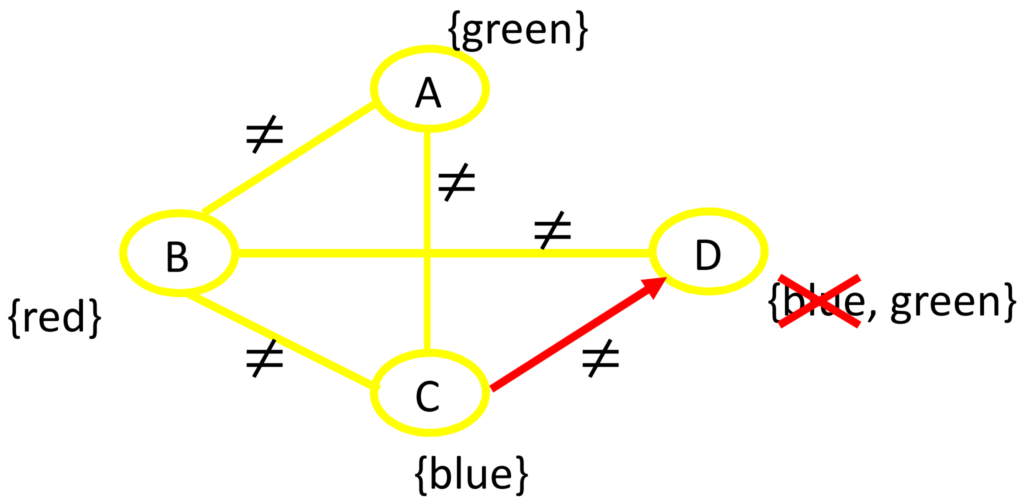
Filtering by Forward-Checking

- Instantiation of C to blue.



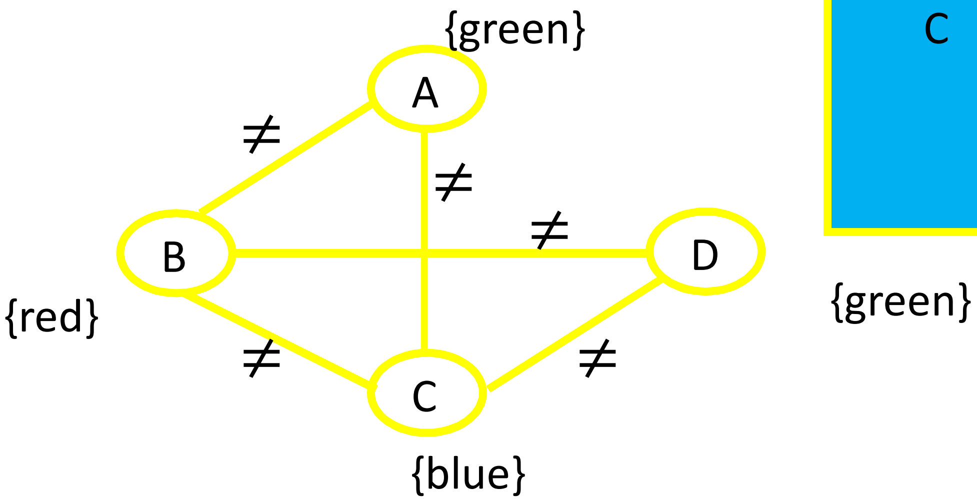
Filtering by Forward-Checking

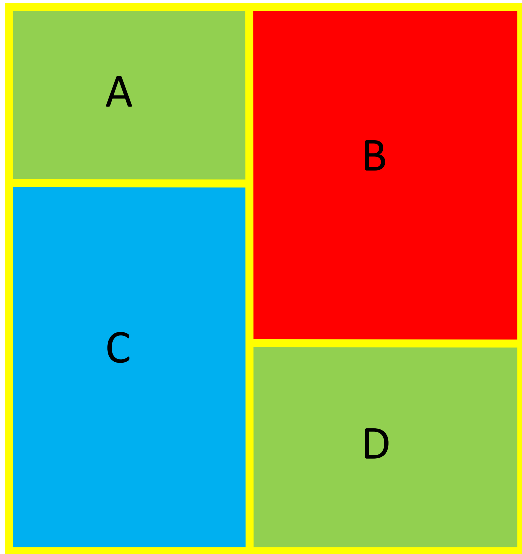
- Instantiation of D to green.



Filtering by Forward-Checking

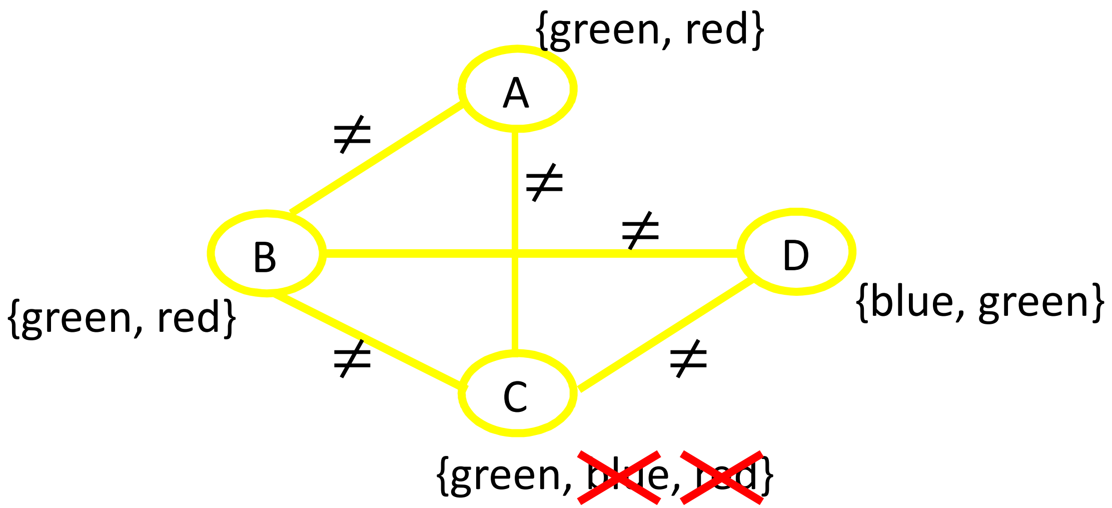
- Solution !





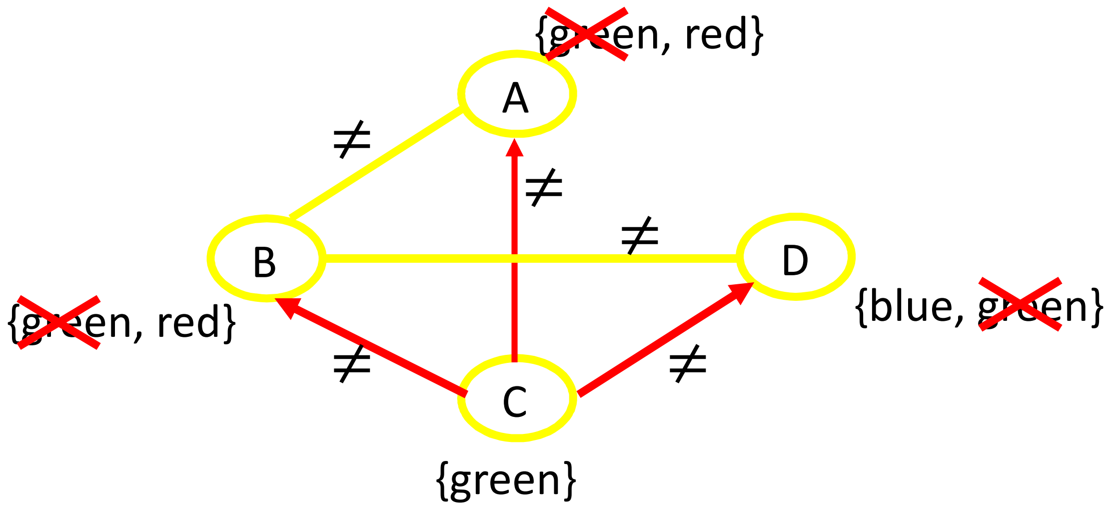
Filtering by Forward-Checking

- Now, let us assume that C is arbitrarily assigned to green.



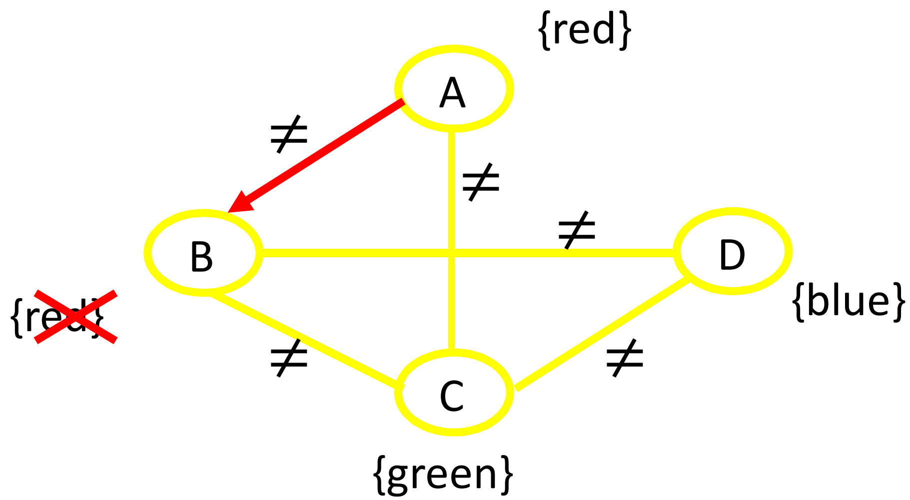
Filtering by Forward-Checking

- Instantiation of D to blue, and A and B to red.



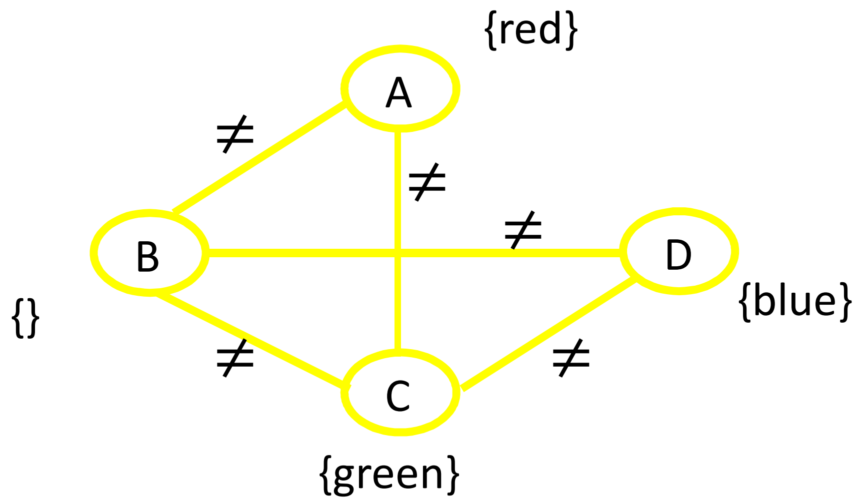
Filtering by Forward-Checking

- Removing red from B’s domain.

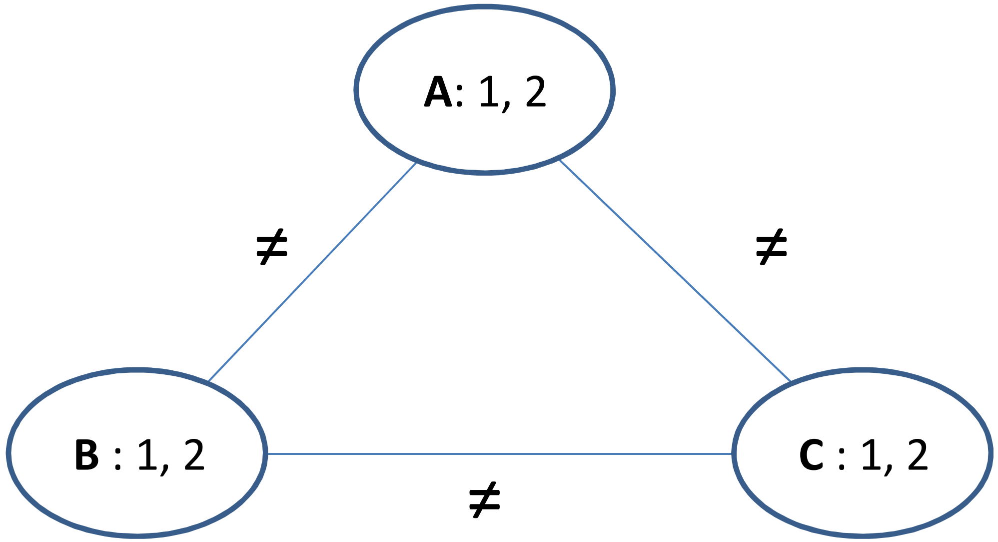


Filtering by Forward-Checking

- B has an empty domain: failure !
- Backtrack is needed in the process…



Arc consistancy

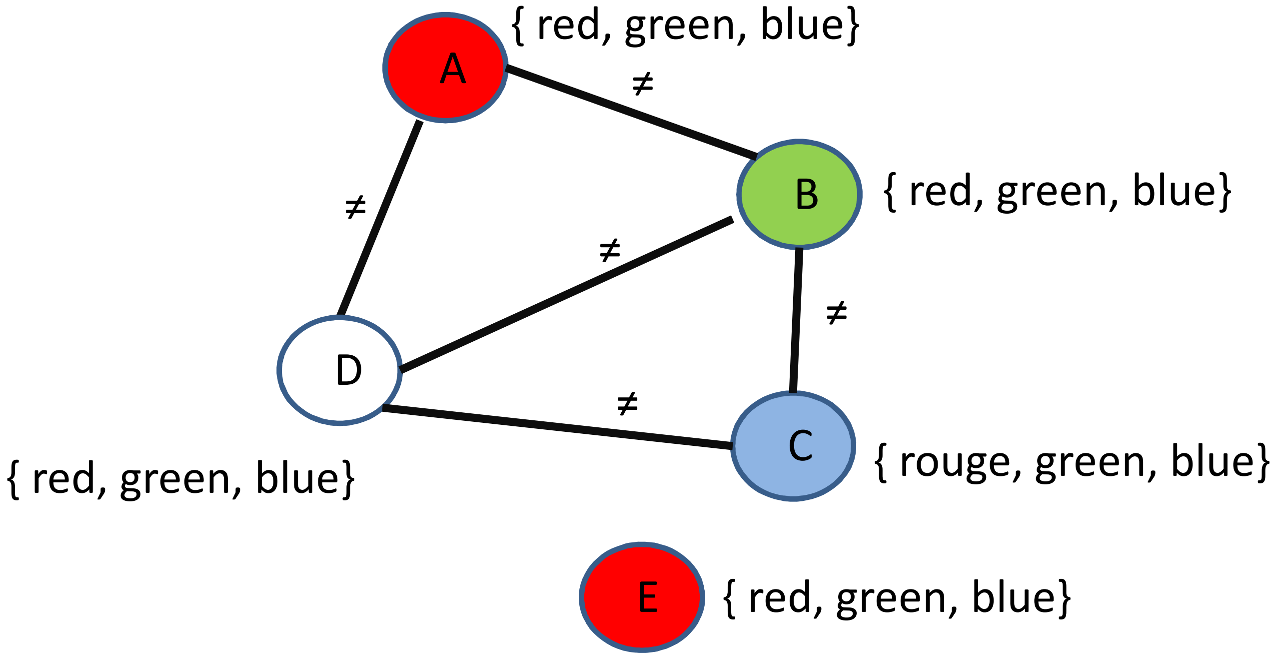


- Arc consistancy is not sufficient for the example above: path consistancy is required.
- K-consistancy, strong k-consistancy.

Back-jumping

- The algorithm Backtrack goes back to the previous choice (e.g., lastly assigned variable).
  - Chronological backtracking.
- Back-jumping: goes back to a variable which caused the failure (e.g., the most recent) in the search tree.
- The conflict set of a variable **V** is the set of previously instantiated variables, linked to **V** by a constraint.

Back-jumping



- Let us assume that the order of instantiation is ***A***, ***B***, ***C***, ***E***, ***D***.
- ***A*** = red ; ***B*** = green ; ***C*** = blue ; ***E*** = red
- No value for ***D***. Chronological backtracking goes back to ***E*** !!!
- The conflict set of ***D*** is { ***A***, ***B***, ***C*** }. Back-jumping goes back to ***C*** and not ***E***.

Conclusion

- Constraint programming is a paradigm for solving combinatorial problems.
- Constraint programming is based on a model (variables, domains, constraints) and an algorithm.
- The algorithm uses forward checking to filter the domains of variables (removing values). Arc consistancy is the more general algorithm for that.
- When an empty domain is detected, goes up in the search tree (chronological backtracking, back jumping) to change a previously made choice (and do better!).