(Complete) Search Algorithms

Speed-up Backtracking

- look-back (reasoning on failure)
- look-ahead (constraint propagation)
- variable ordering
- value ordering

Look back
(reasoning on failure)

- Foncer sur les échecs (BT-like), réfléchir ensuite

  C’est à dire :

- Tirer parti des informations implicites contenues dans un échec pour :

  1. sélectionner un meilleur point de retour arrière

  2. ne pas retomber sur la même raison d’échec

Backjumping

- Quand domaine vide : remonter à la variable la plus basse « impliquée dans l'échec »

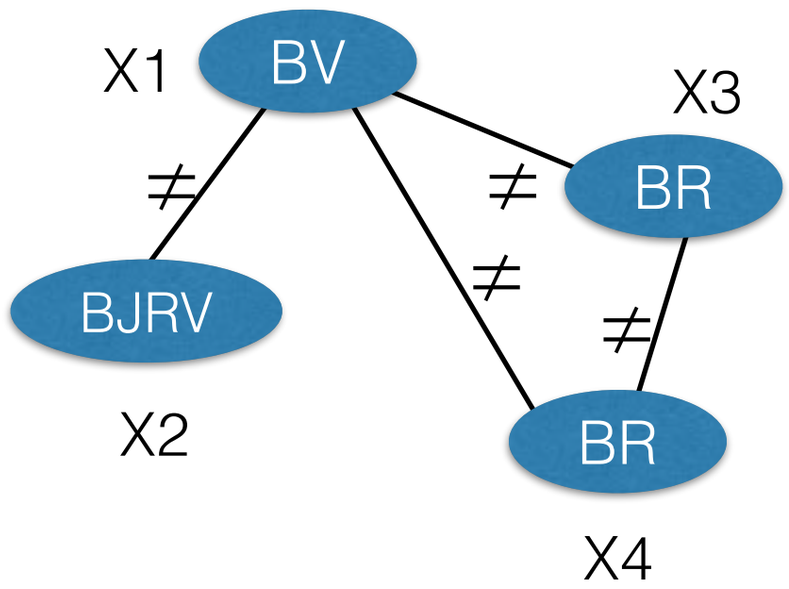
CBJ on running example

X1

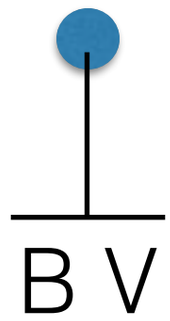
X2

X3

X4



CBJ on running example

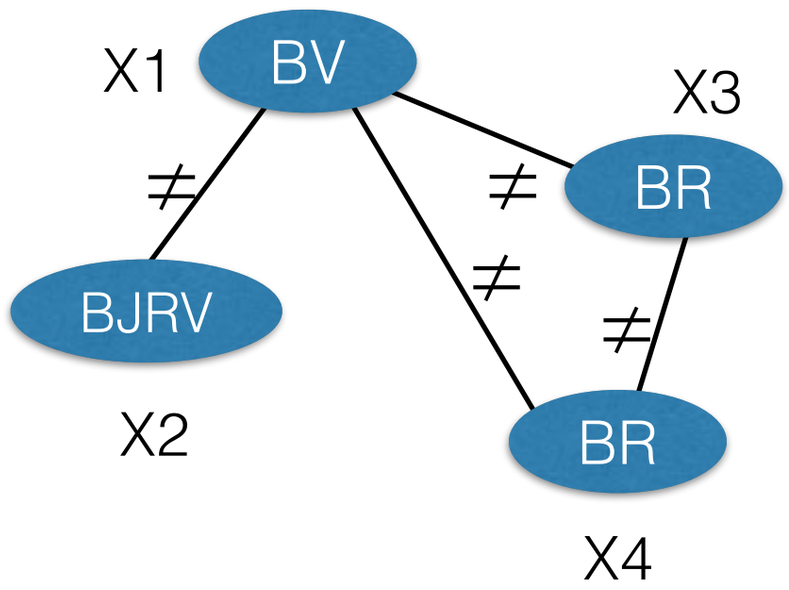


X1

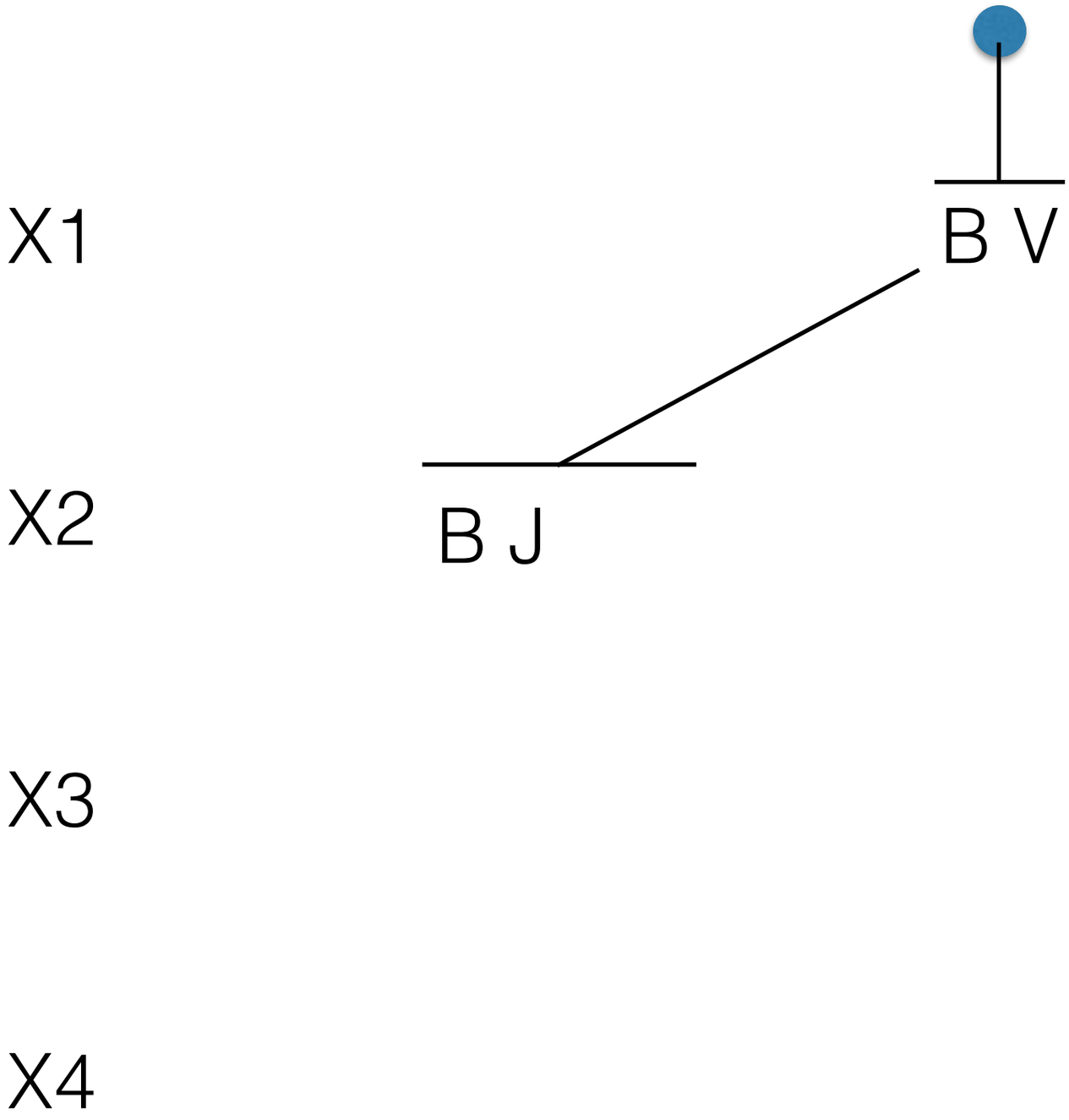
X2

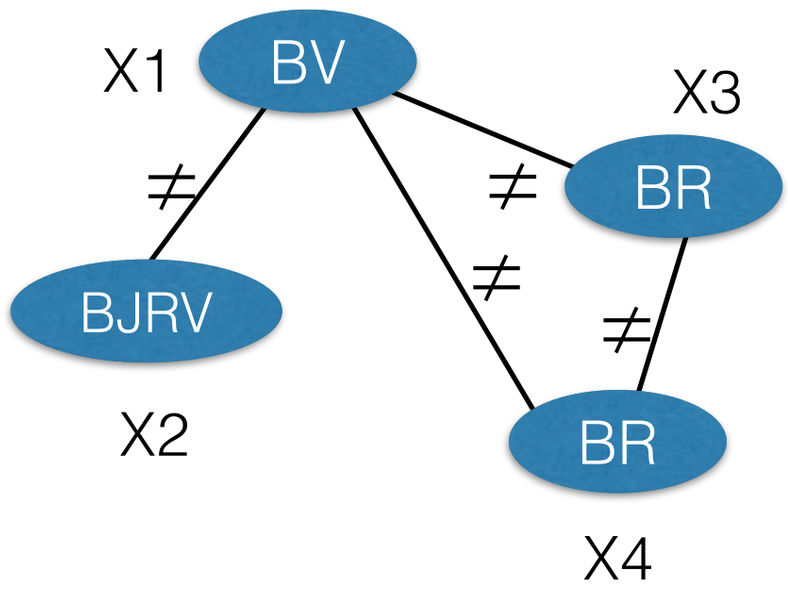
X3

X4

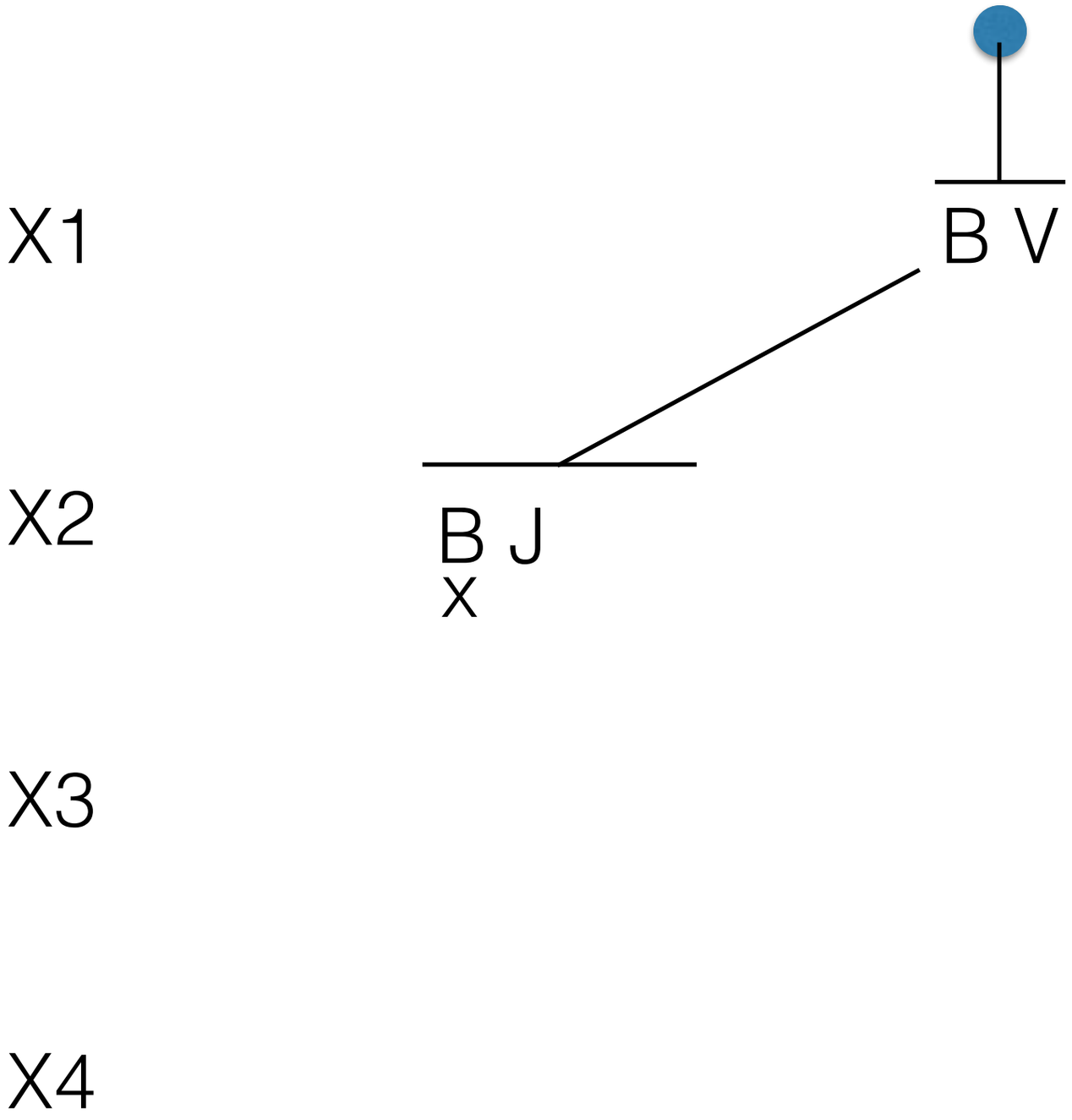


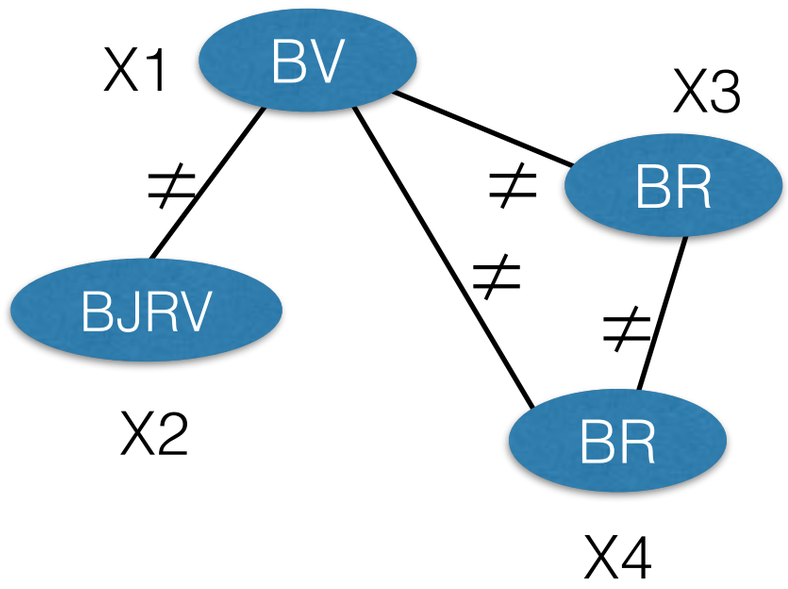
CBJ on running example



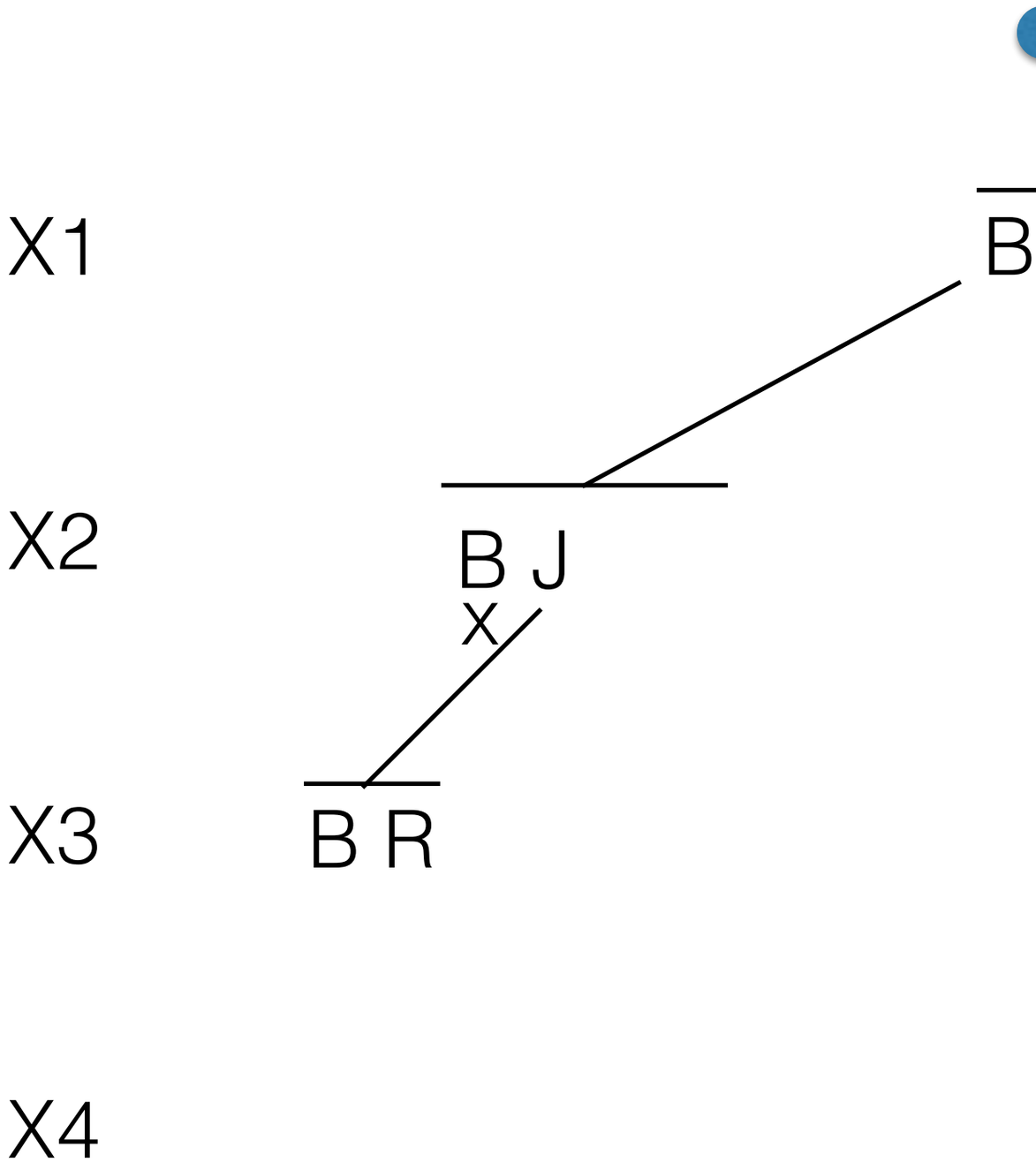


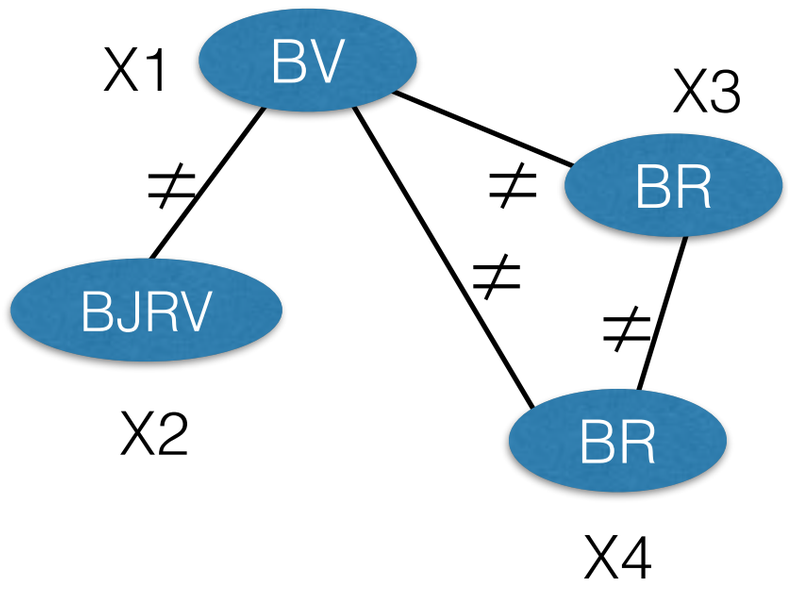
CBJ on running example



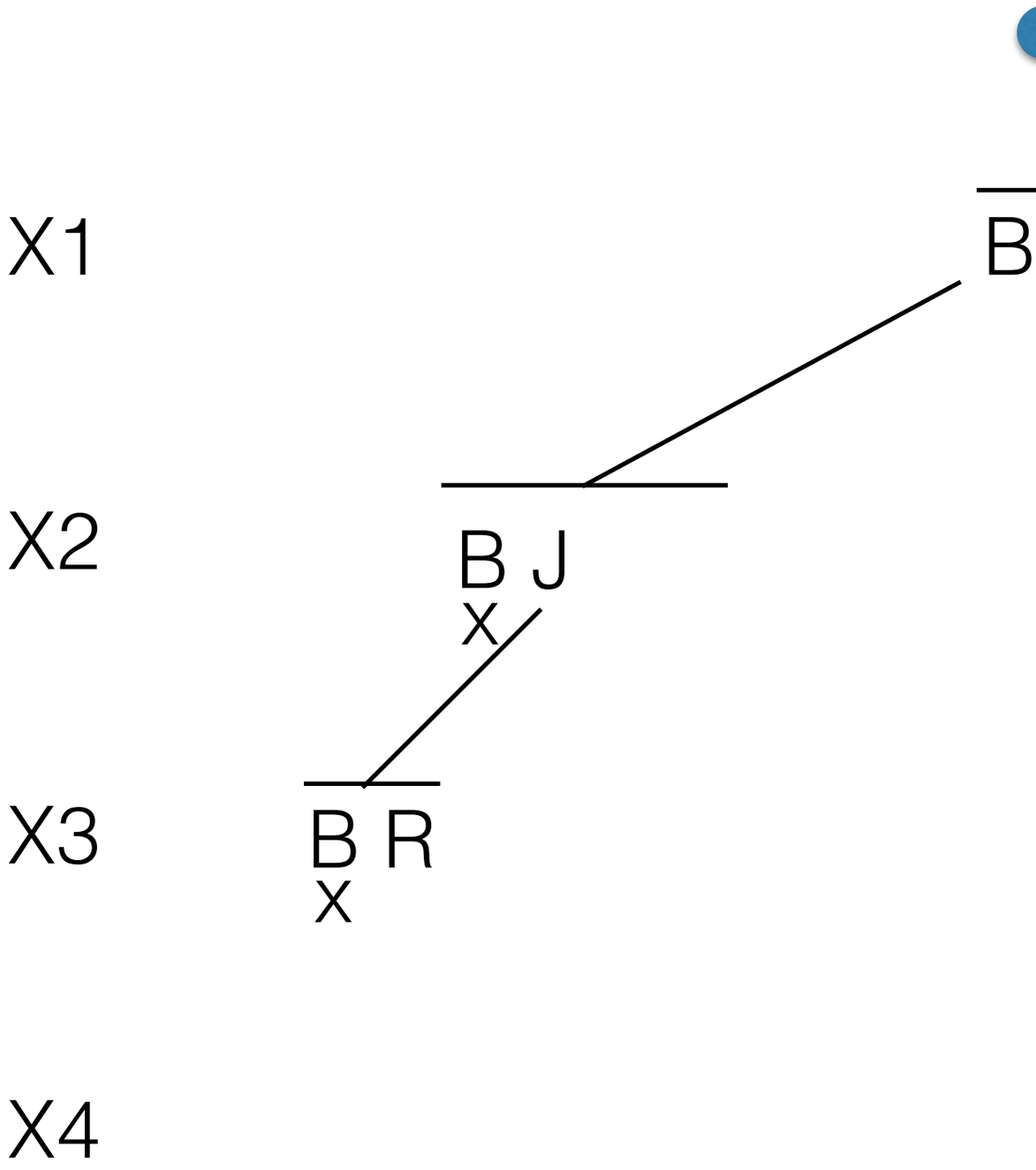


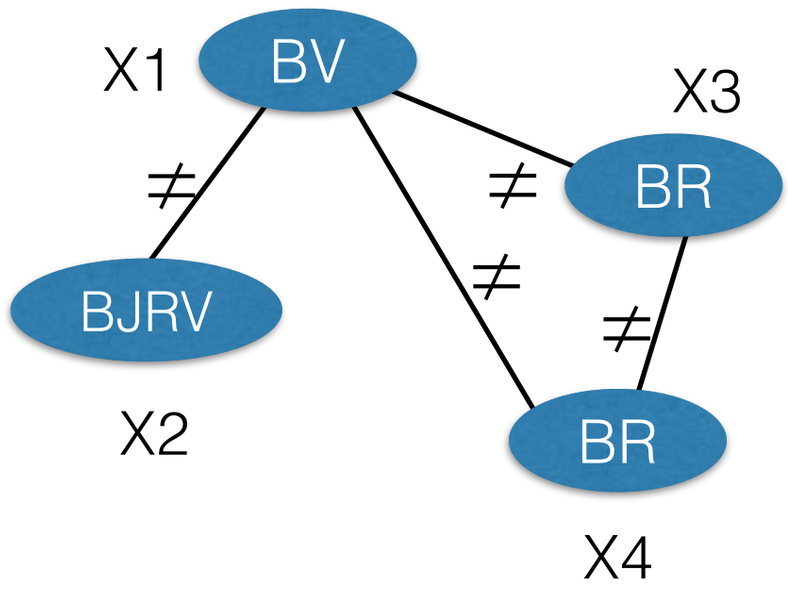
CBJ on running example



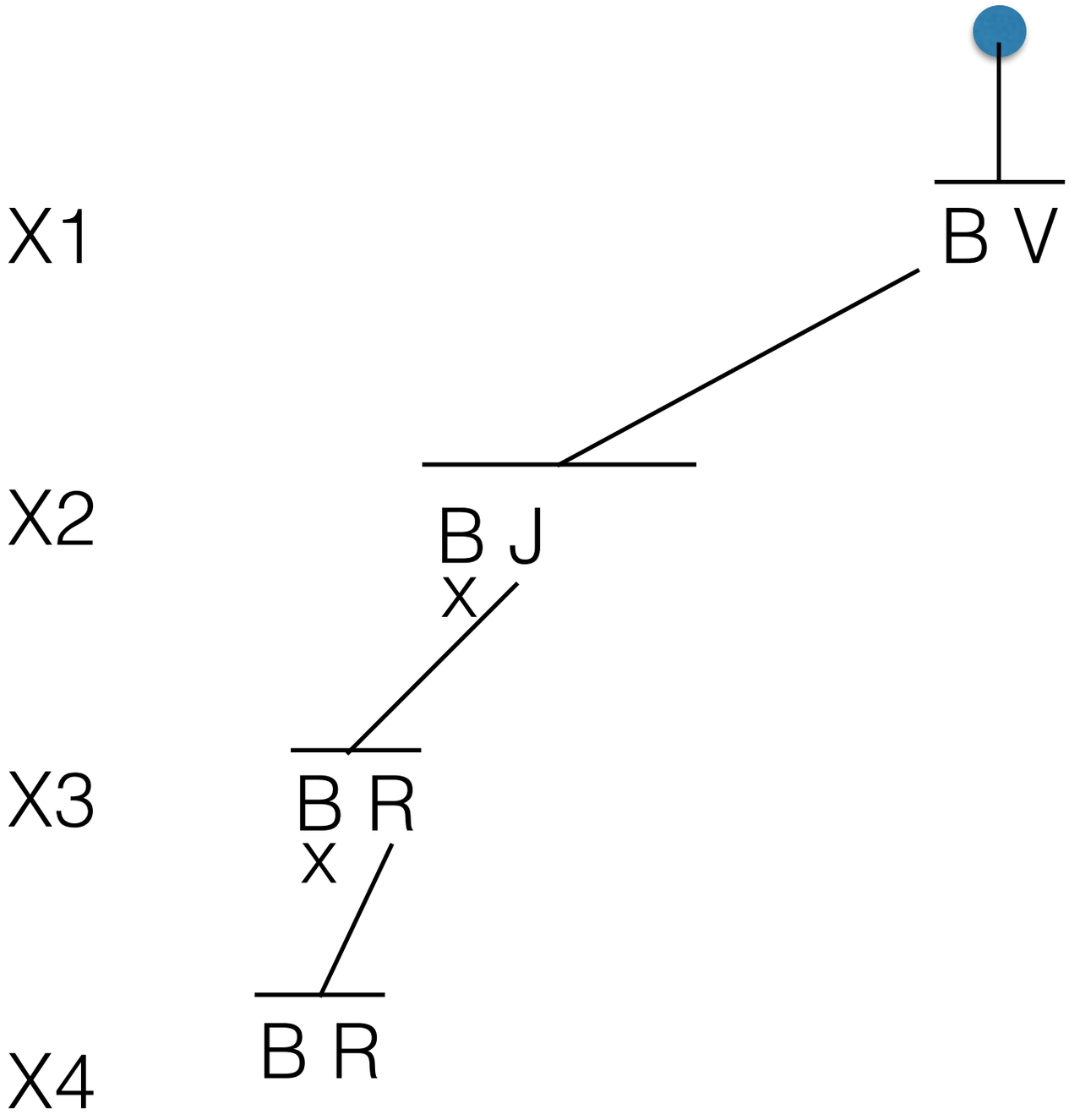


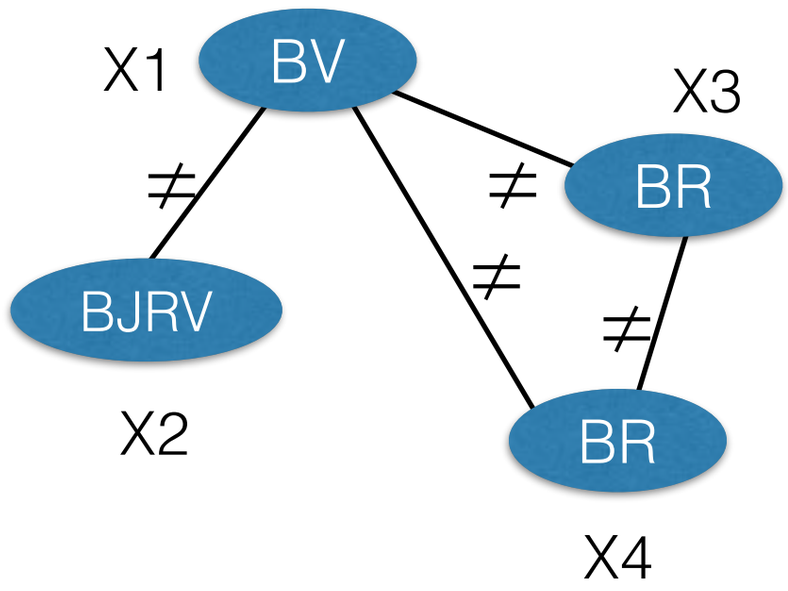
CBJ on running example



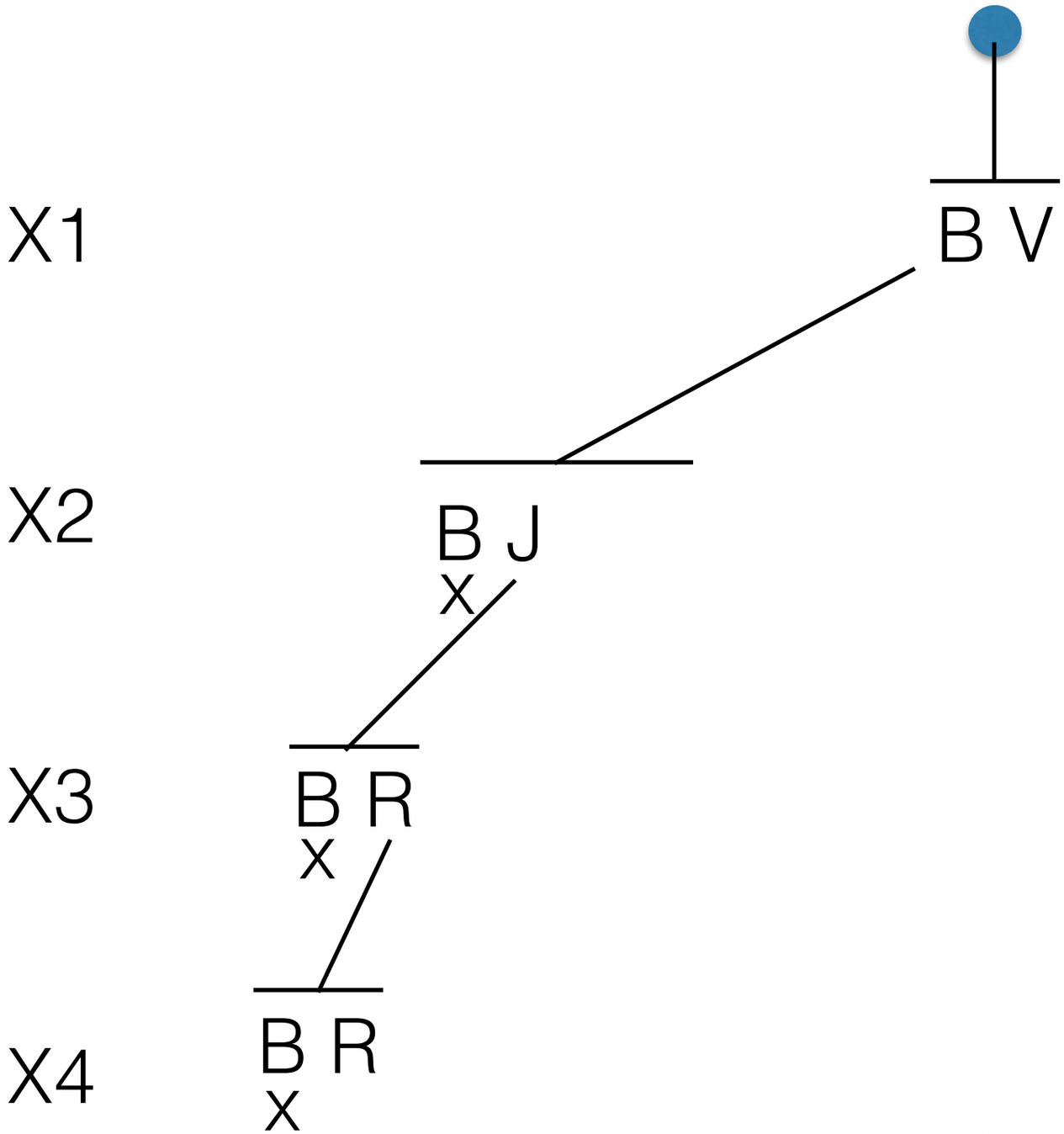


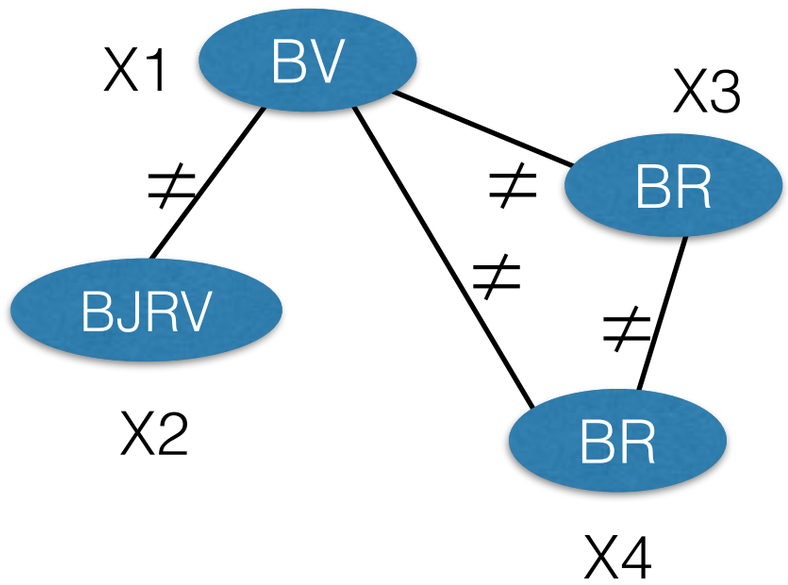
CBJ on running example



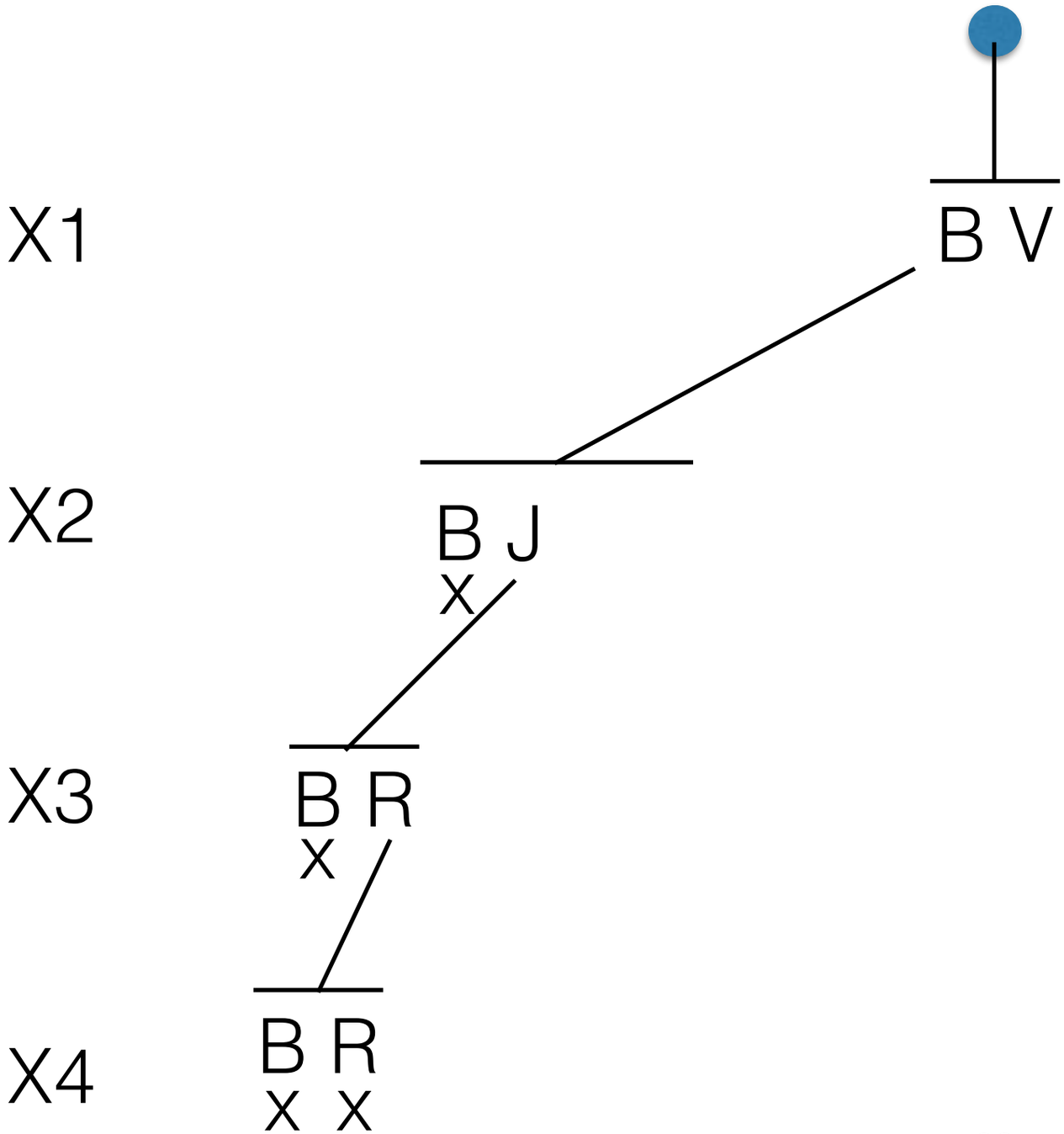


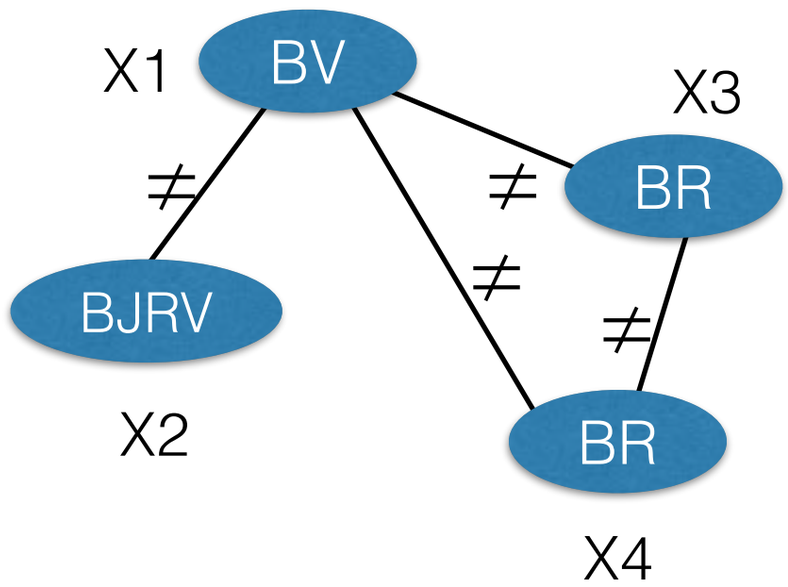
CBJ on running example



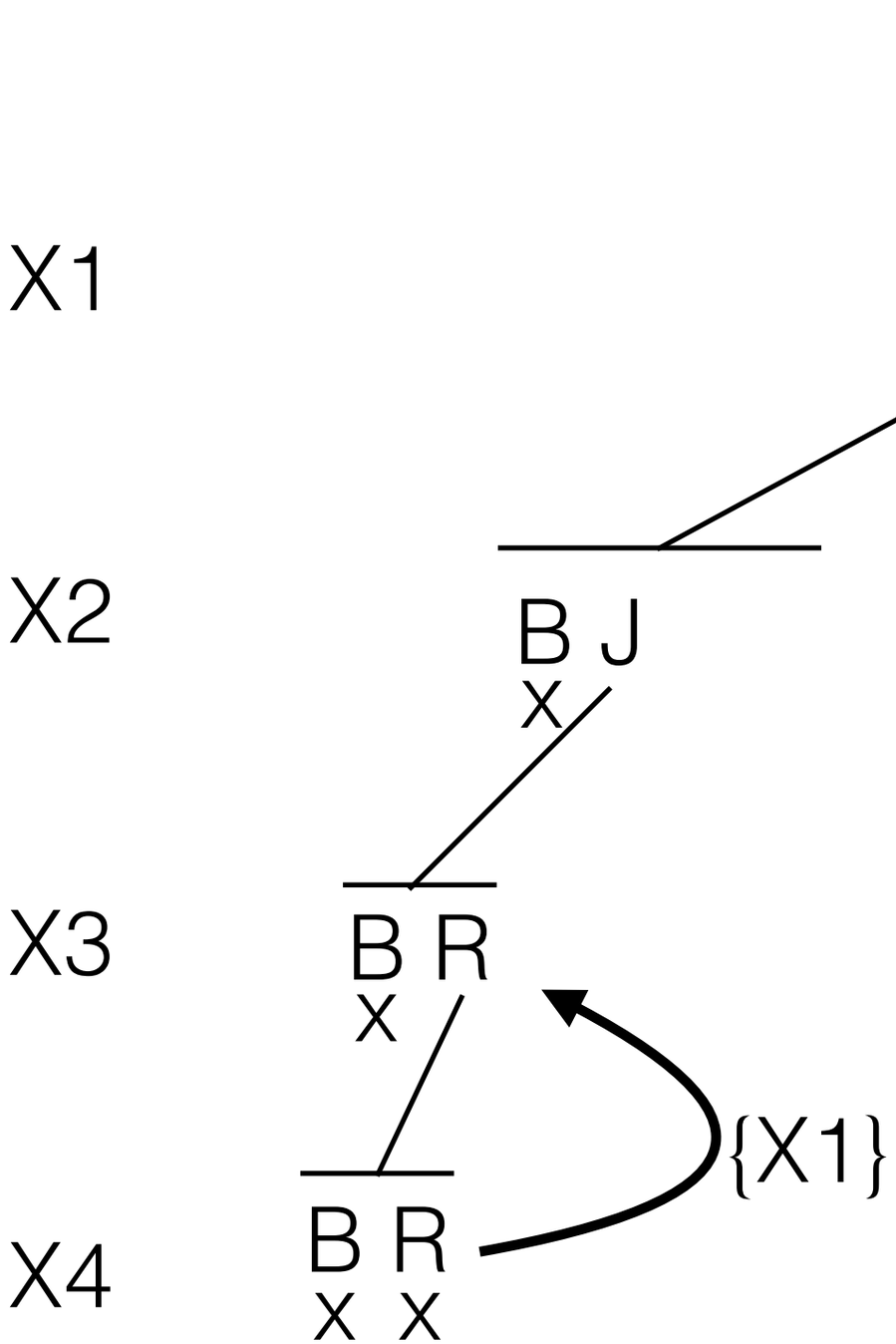


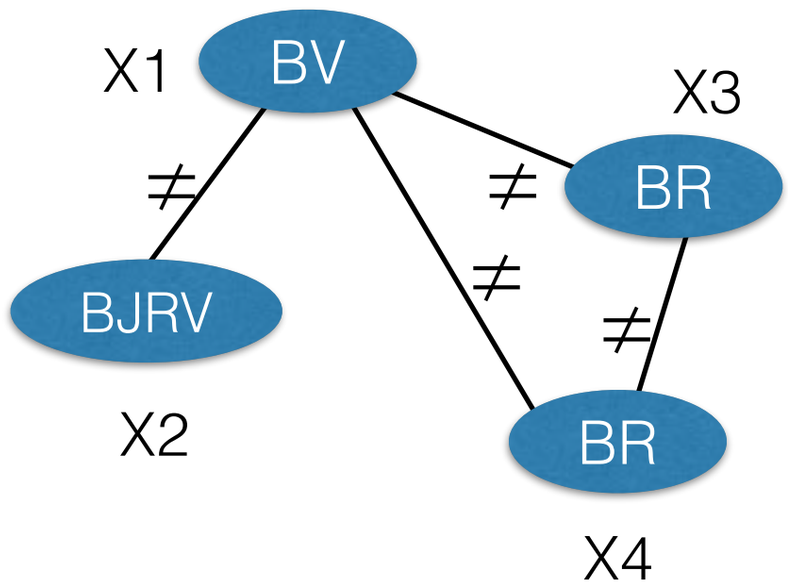
CBJ on running example



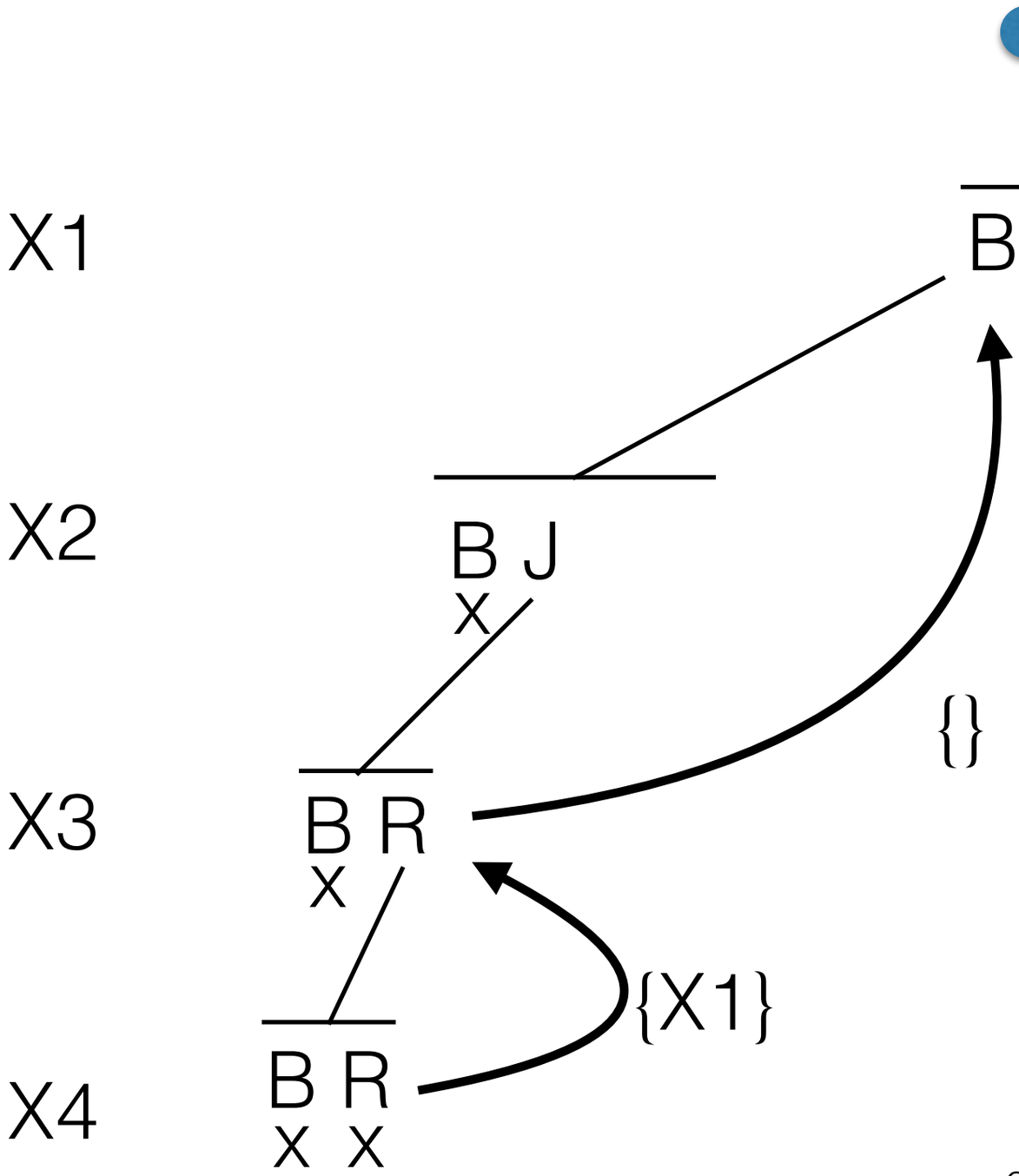


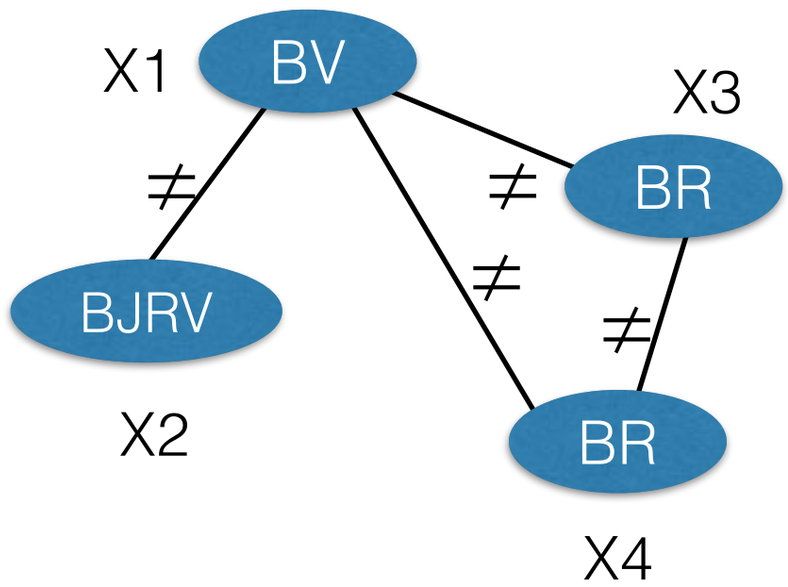
CBJ on running example



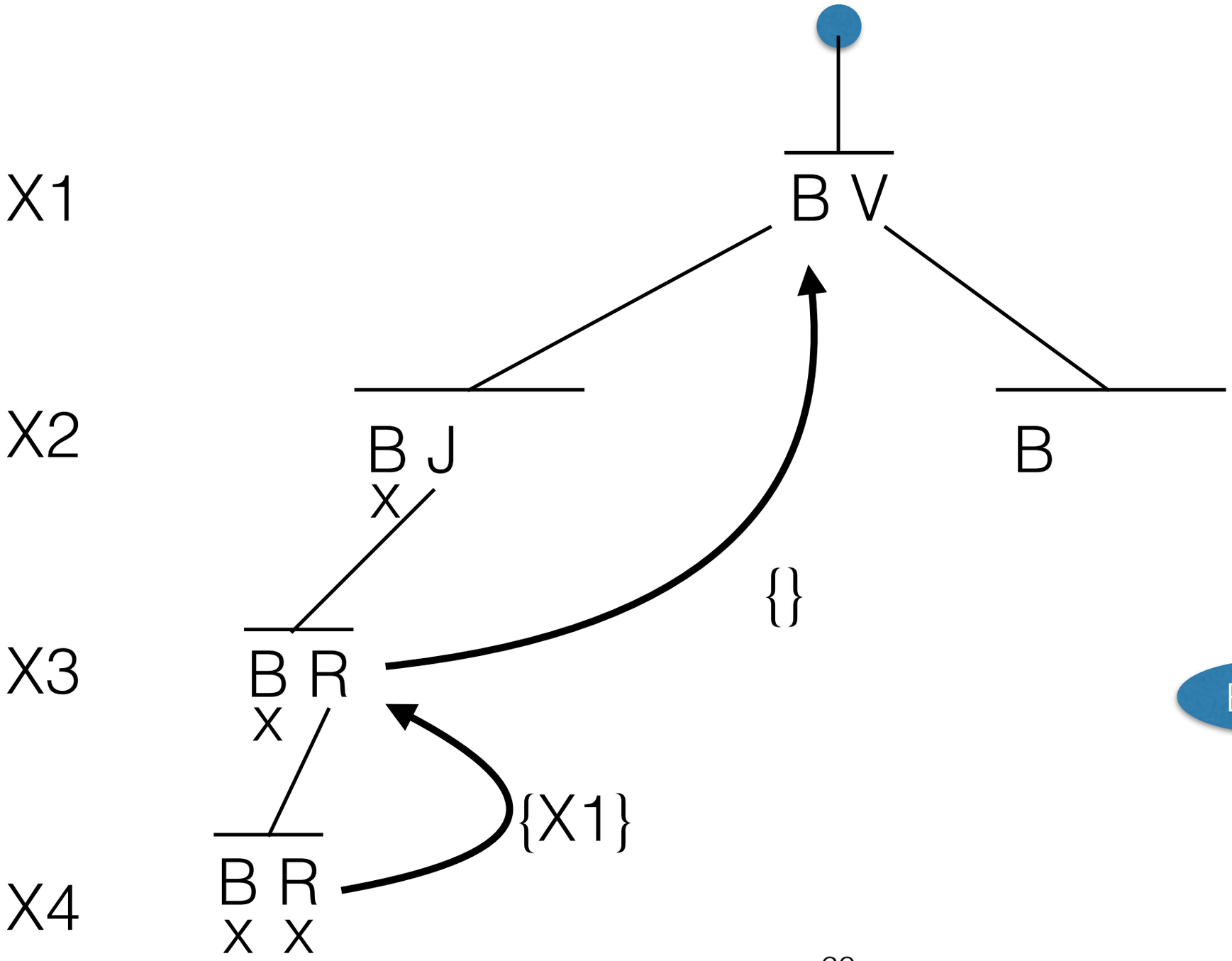


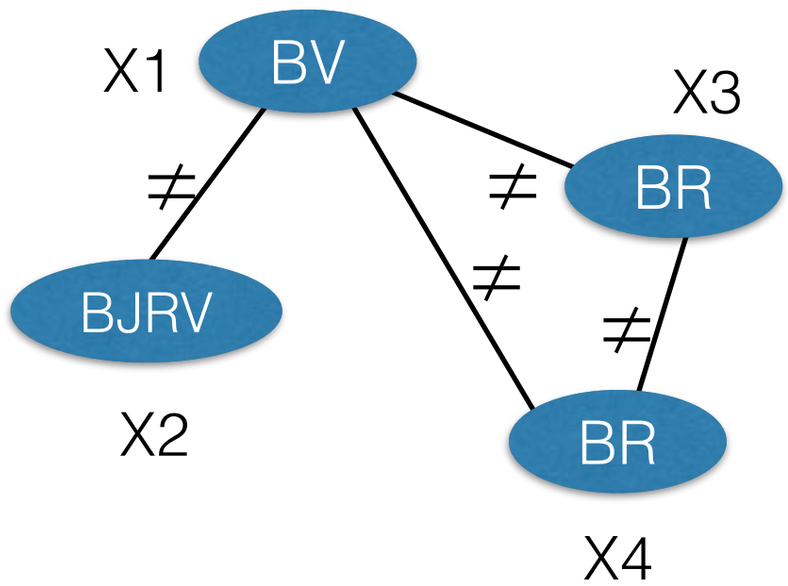
CBJ on running example



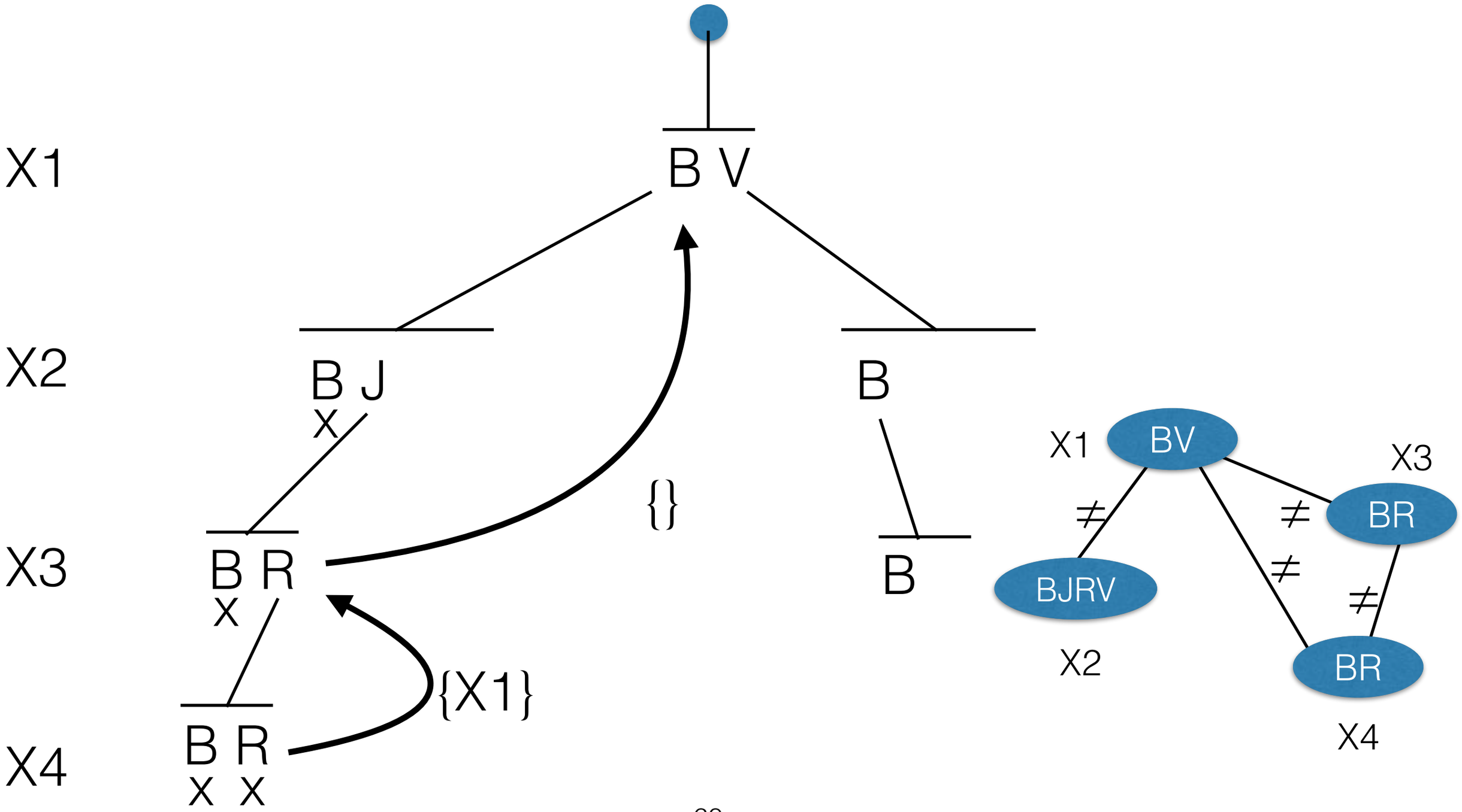


CBJ on running example

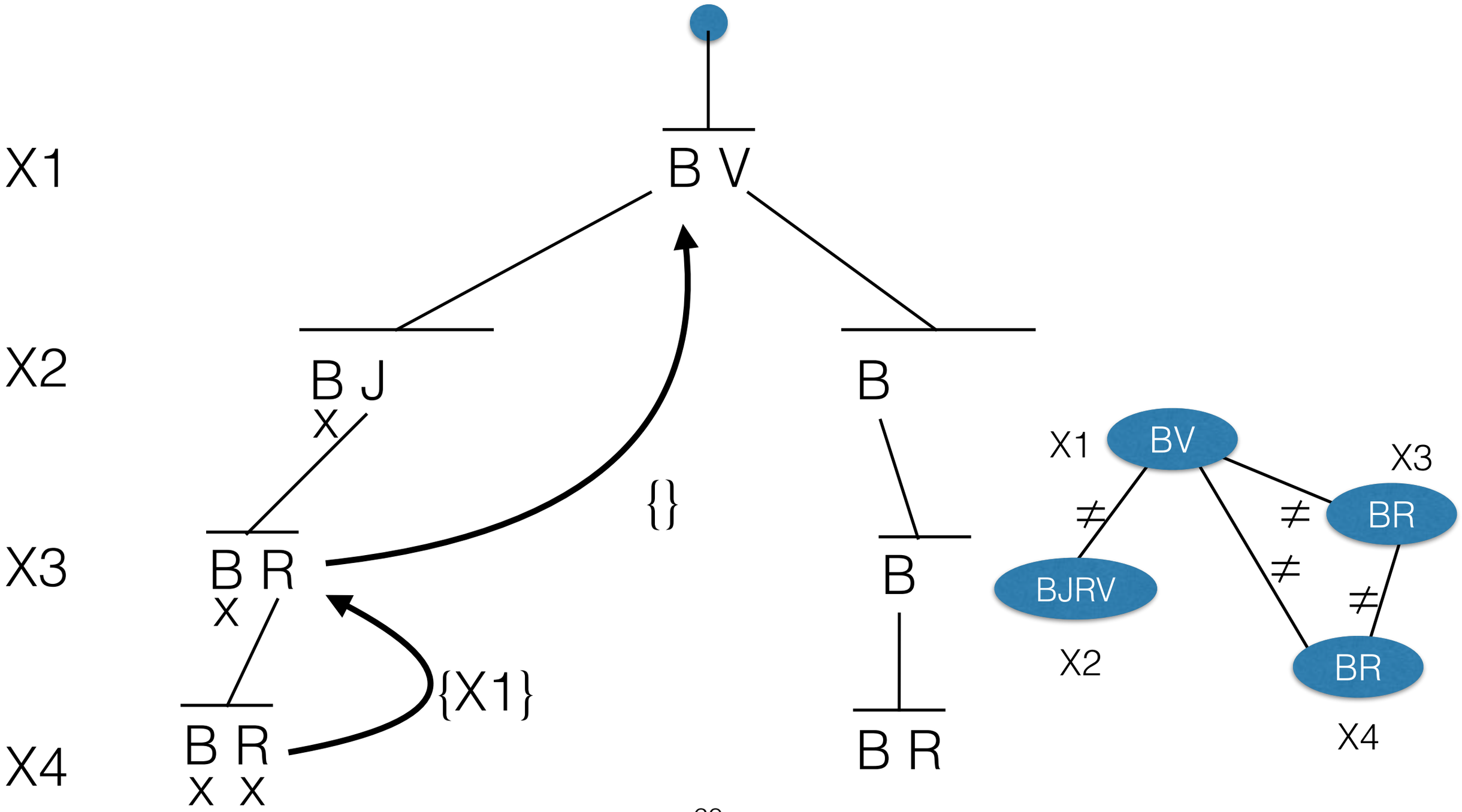




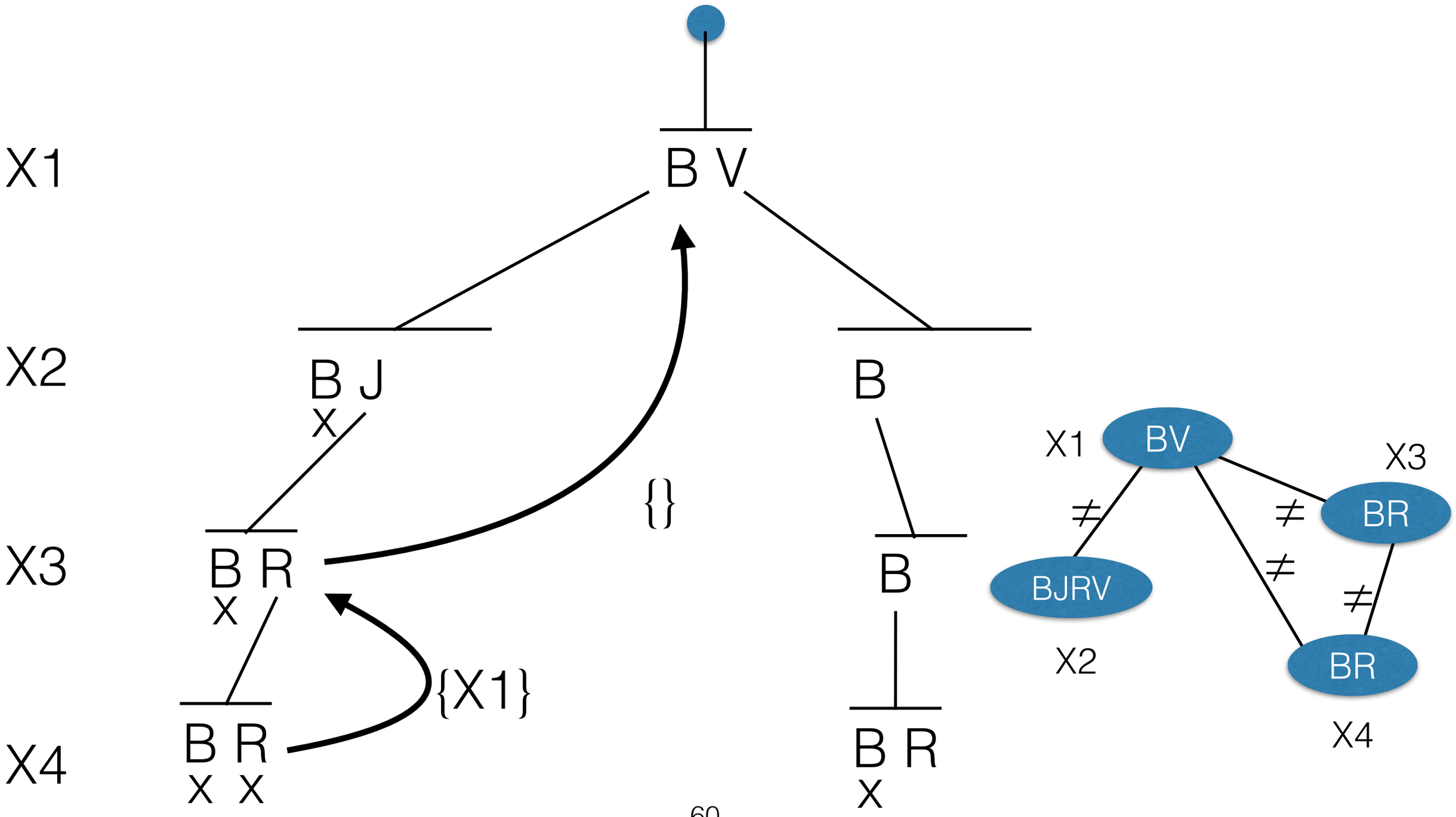
CBJ on running example



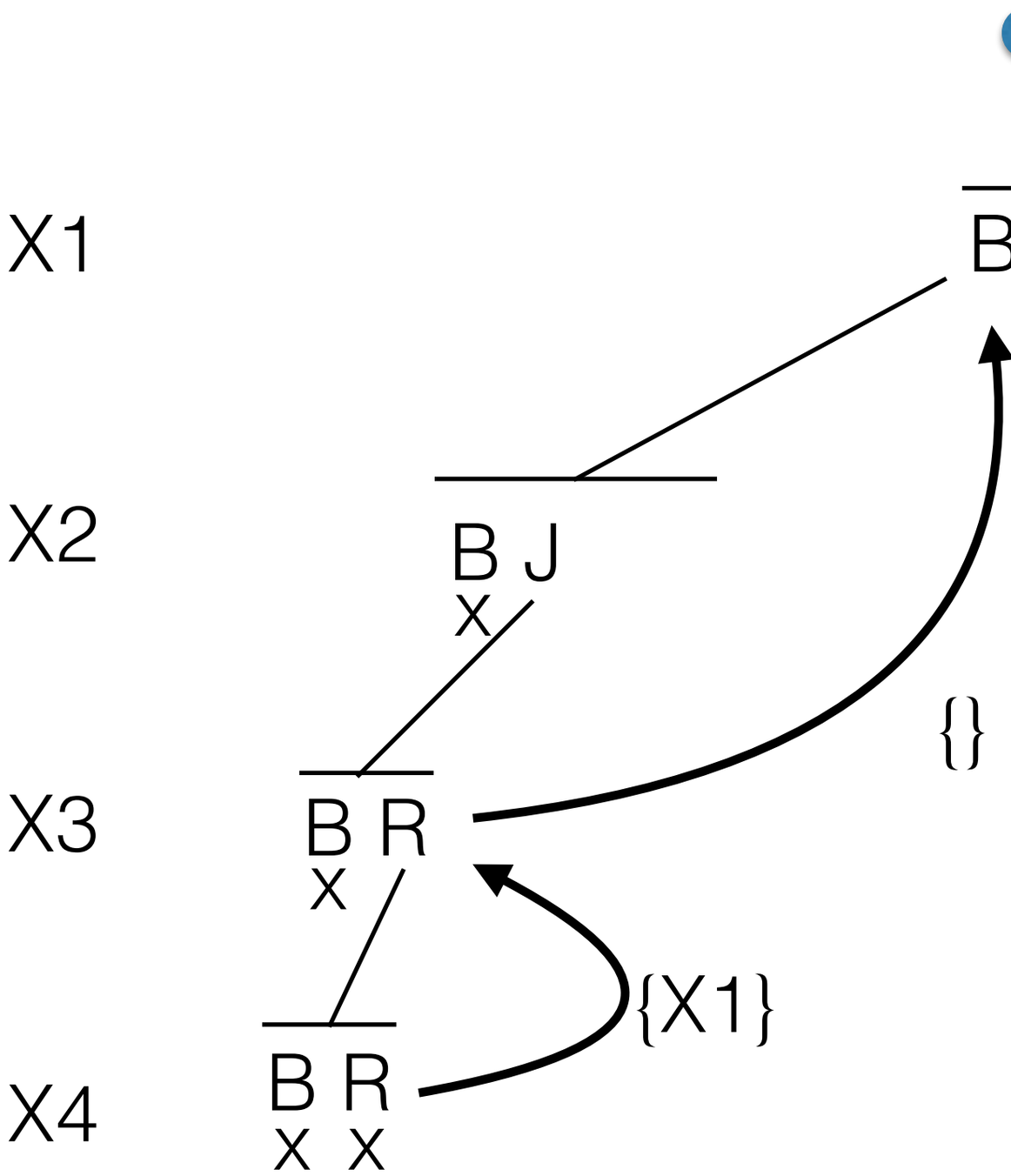
CBJ on running example

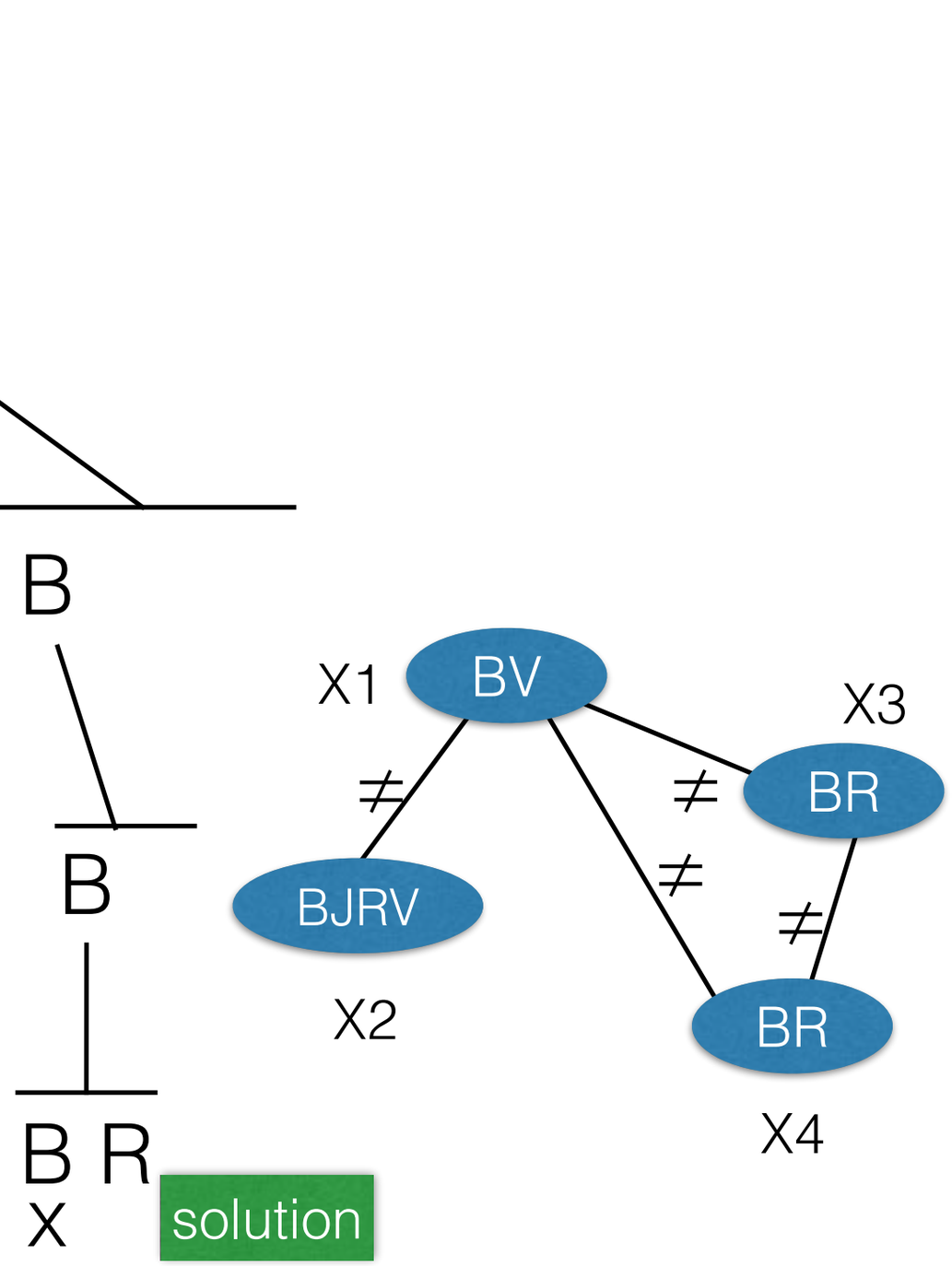


CBJ on running example



CBJ on running example





Nogood learning

- A chaque domaine vide, on mémorise une raison de l'échec (nogood), par exemple : [$X_i \neq 2$]v[$X_j \neq 3$]

- Approche proposée dans les 80s sur les CSPs
  - lourd, peu efficace

- Récupéré par SAT en 1997 puis gros succès à partir de 2001 (1UIP et restart) —> solveurs CDCL

- Revenu dans les CSPs depuis 2007 (*lazy clause generation*). Nogoods plus expressifs : [$X_i \leq 6$]v[$X_j \neq 3$].

Look-ahead (constraint propagation)

- Réfléchir d'abord où ne pas descendre, c'est à dire:
- Enlever ***des*** branches qui conduisent à un échec
  - ✤ Instancier X_i
  - ✤ Pour tout $k > i$ supprimer de D(X_k) ***des*** valeurs incompatibles avec X_1.. X_i
  - ✤ Si $D(X_k)=\varnothing$, arrêter cette branche

Forward checking (FC)

function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end

Forward checking (FC)

function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end

FC on running example

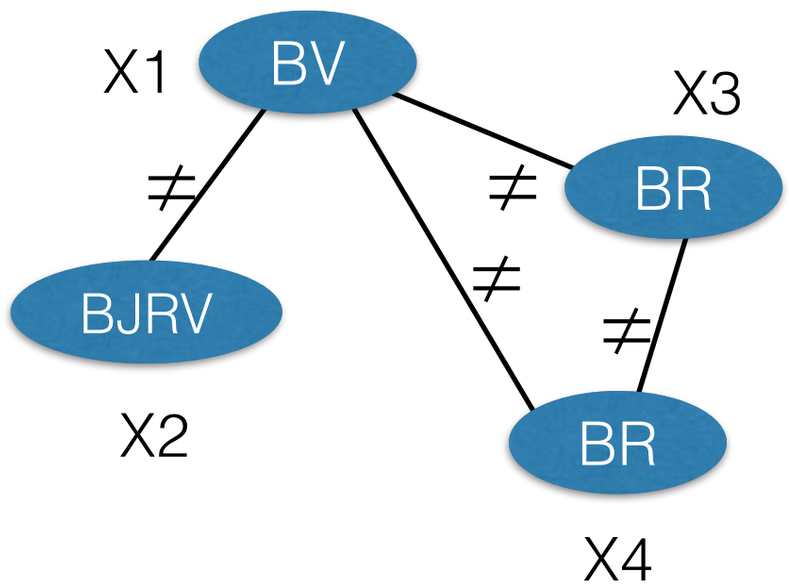
X1

X2

X3

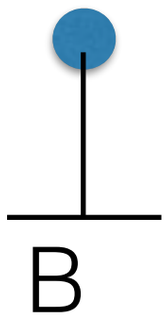
X4

function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



FC on running example

X1

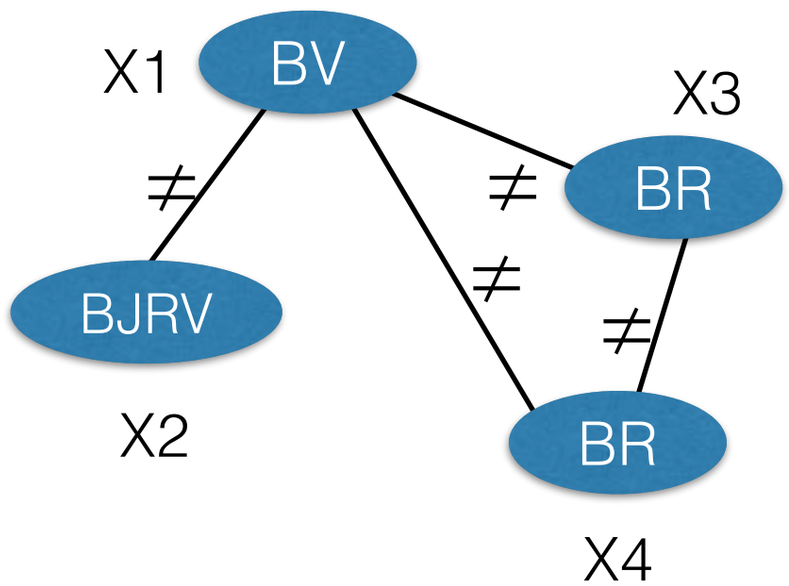


X2

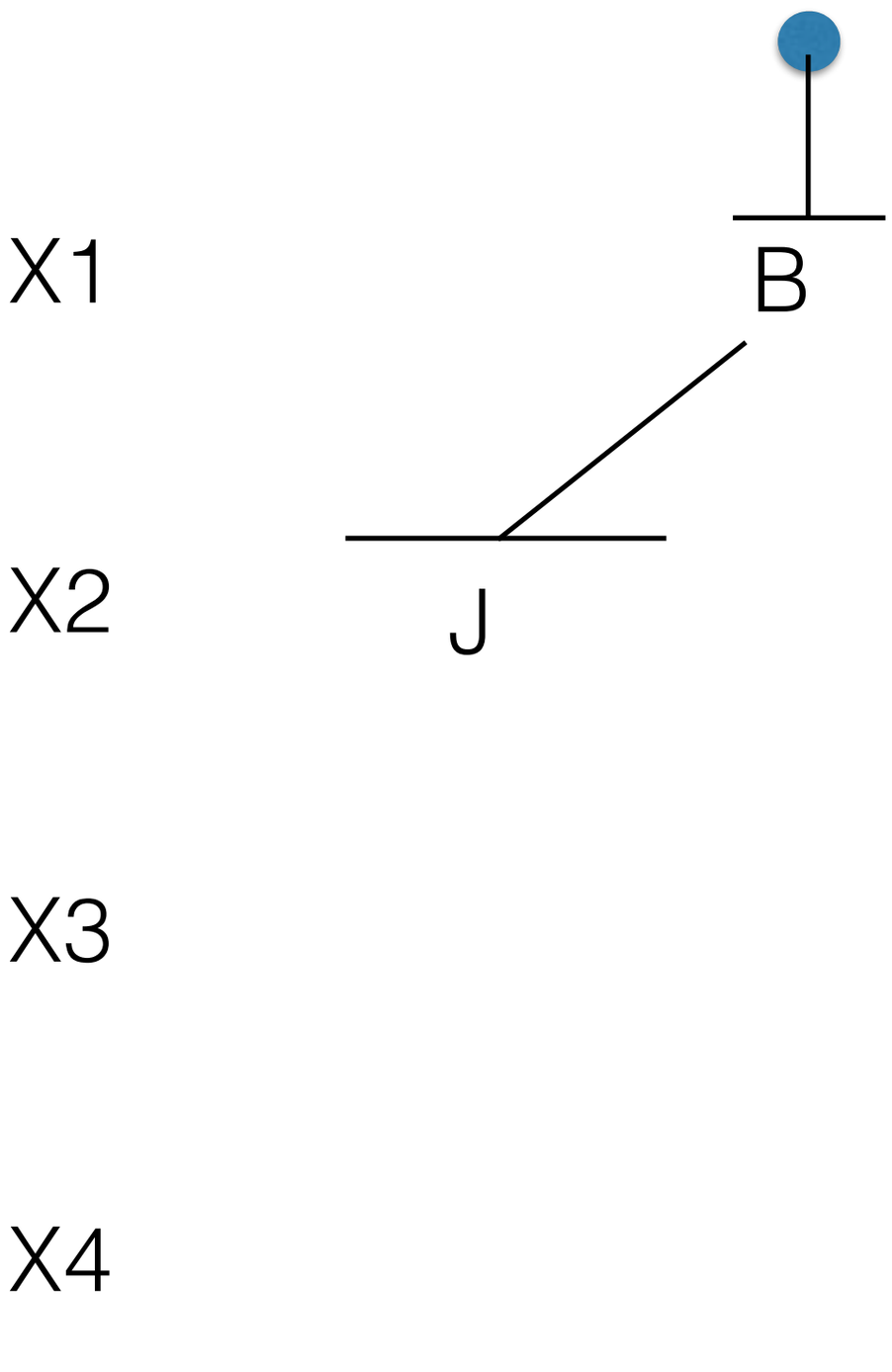
X3

X4

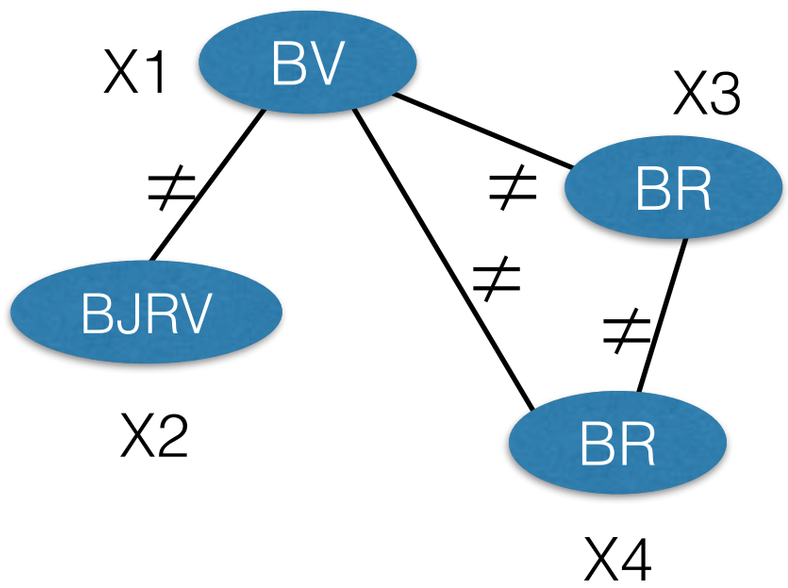
```
function FC(N : network ; I : instantiation) : Boolean
begin
   if |I| = n then return true
   choose a variable x_i not in I
   for each v_i ∈ D(x_i) do
      D(x_i) ← {v_i}
      remove all (x_j, v_j) inconsistent with I ← I ∪ (x_i, v_i)
      if not empty domain then
         if FC(N, I ∪ (x_i, v_i)) then return true
      restore all values pruned because of X_i = v_i
   return false
end
```



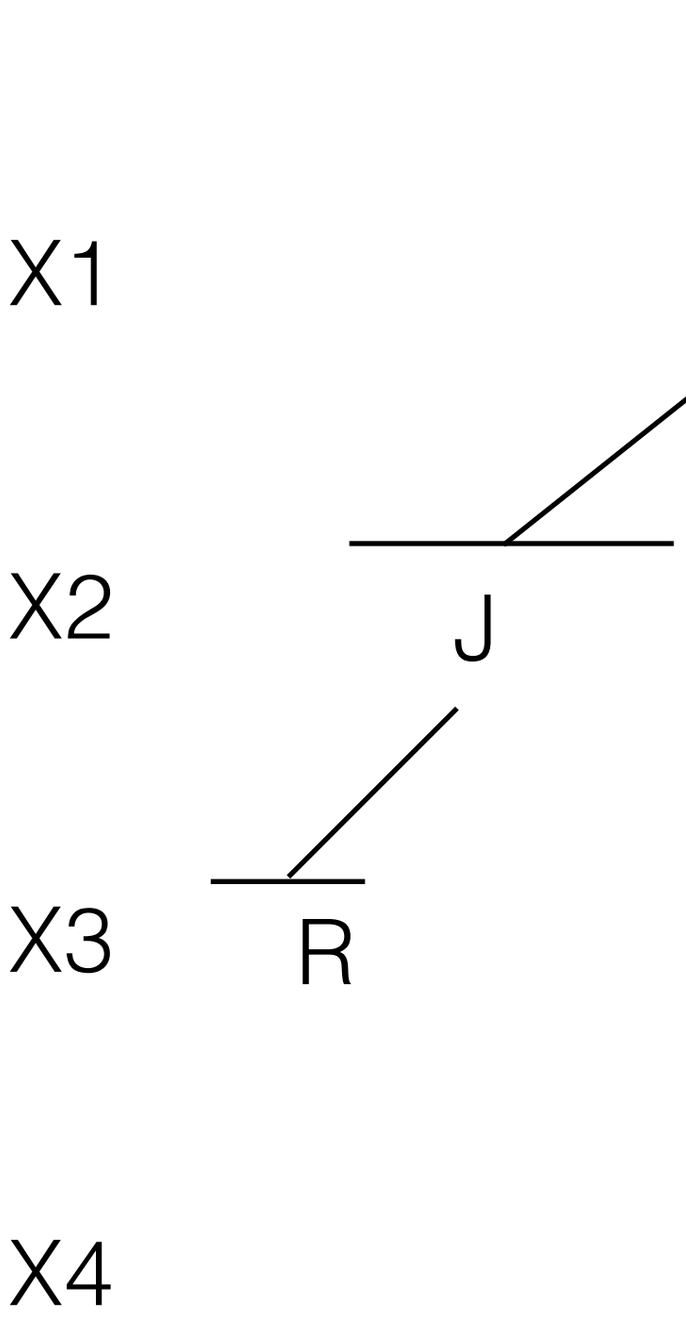
FC on running example



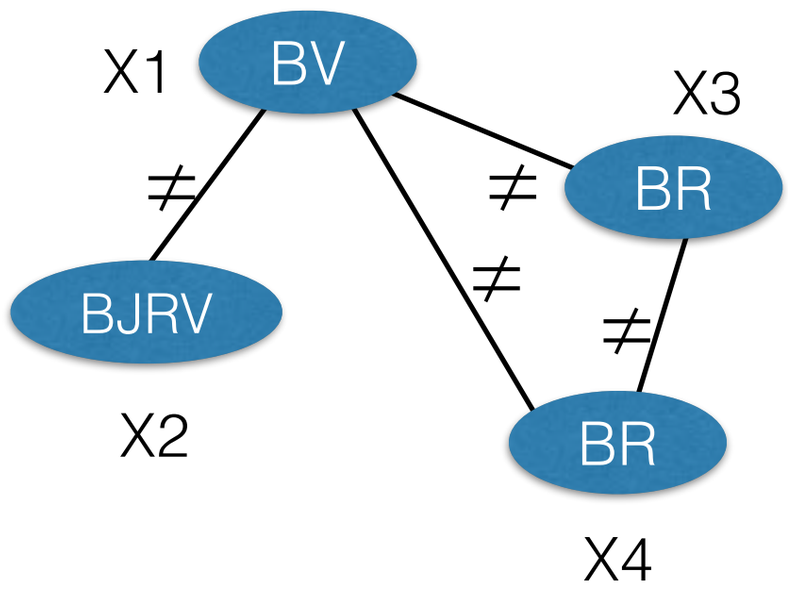
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



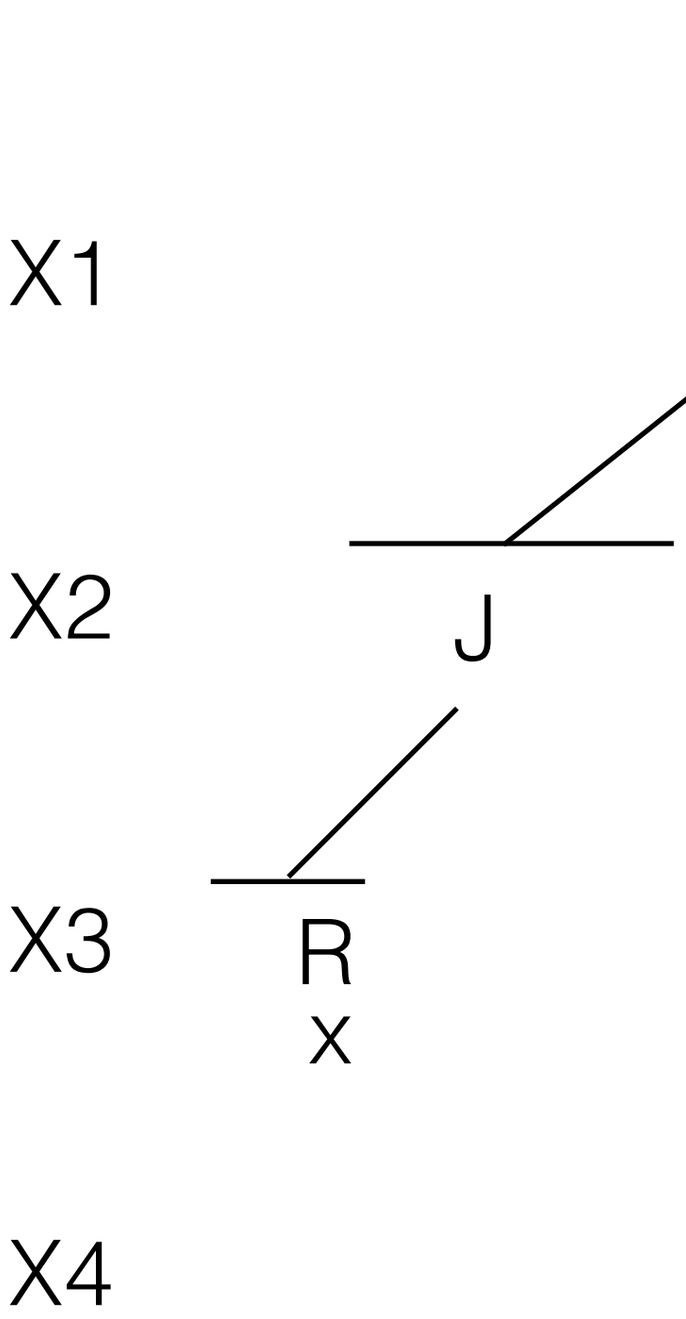
FC on running example



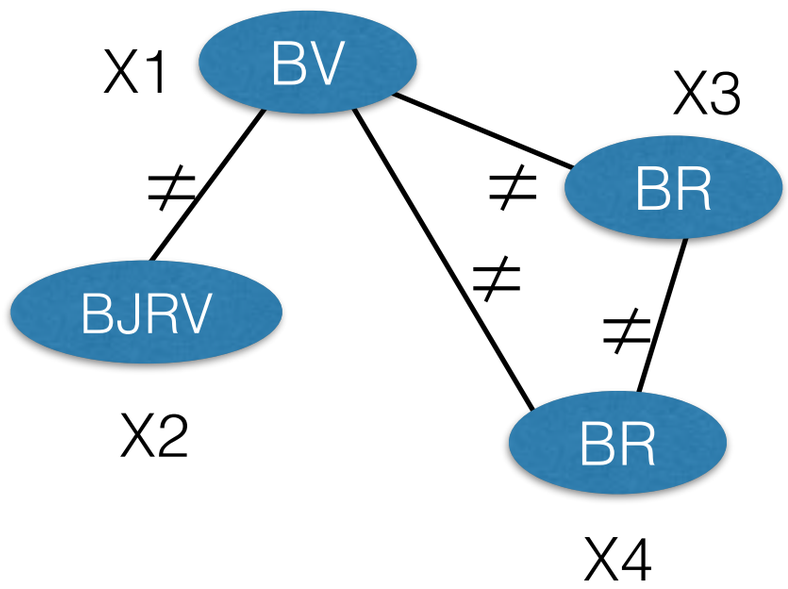
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



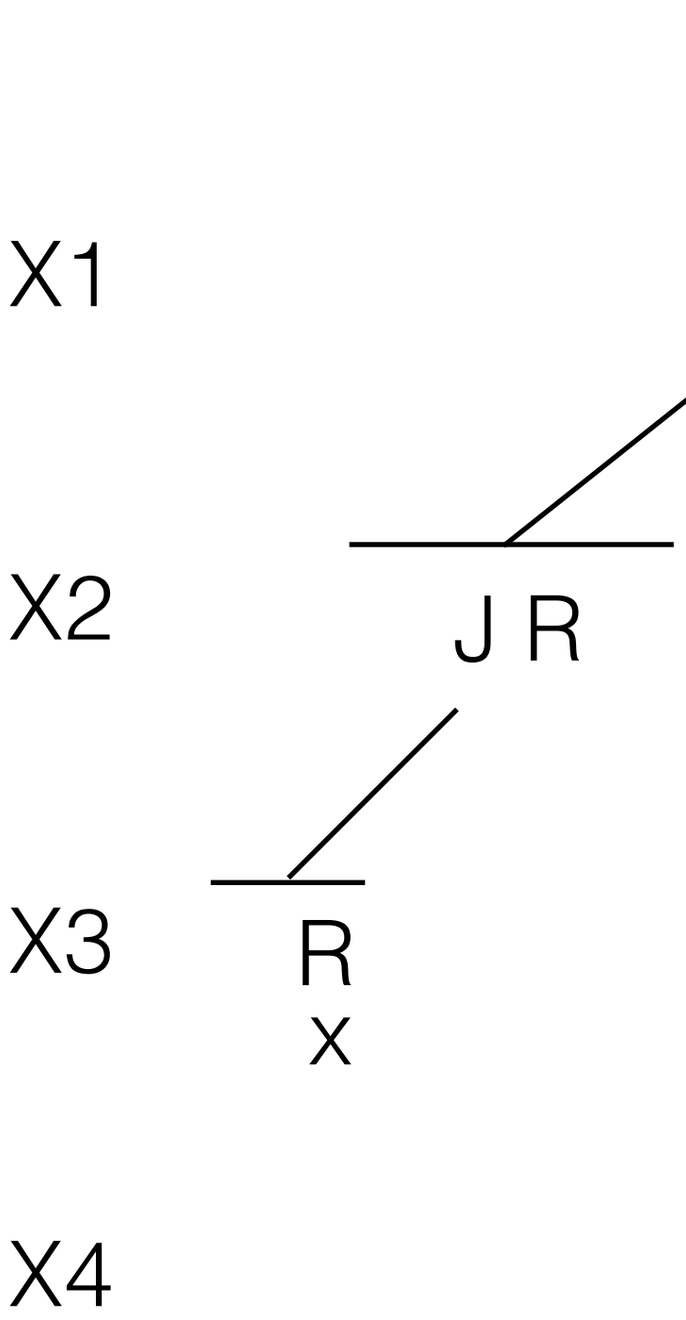
FC on running example



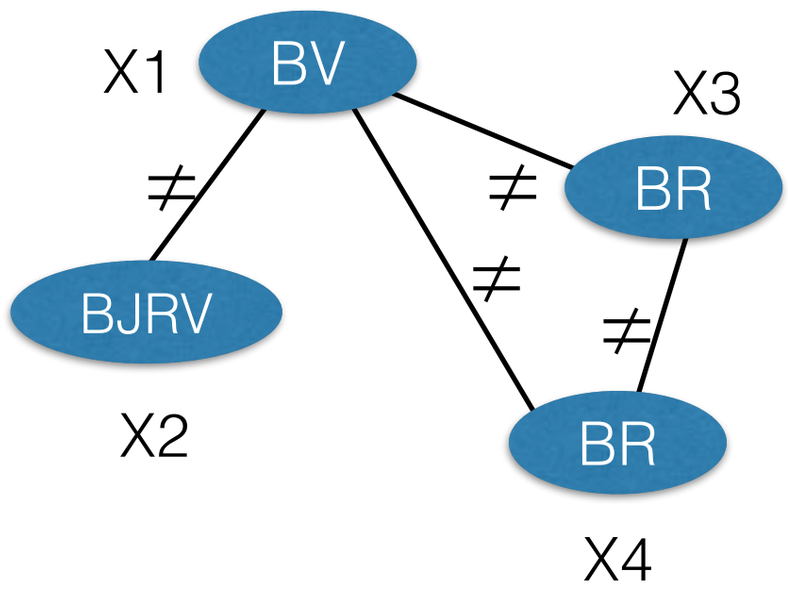
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



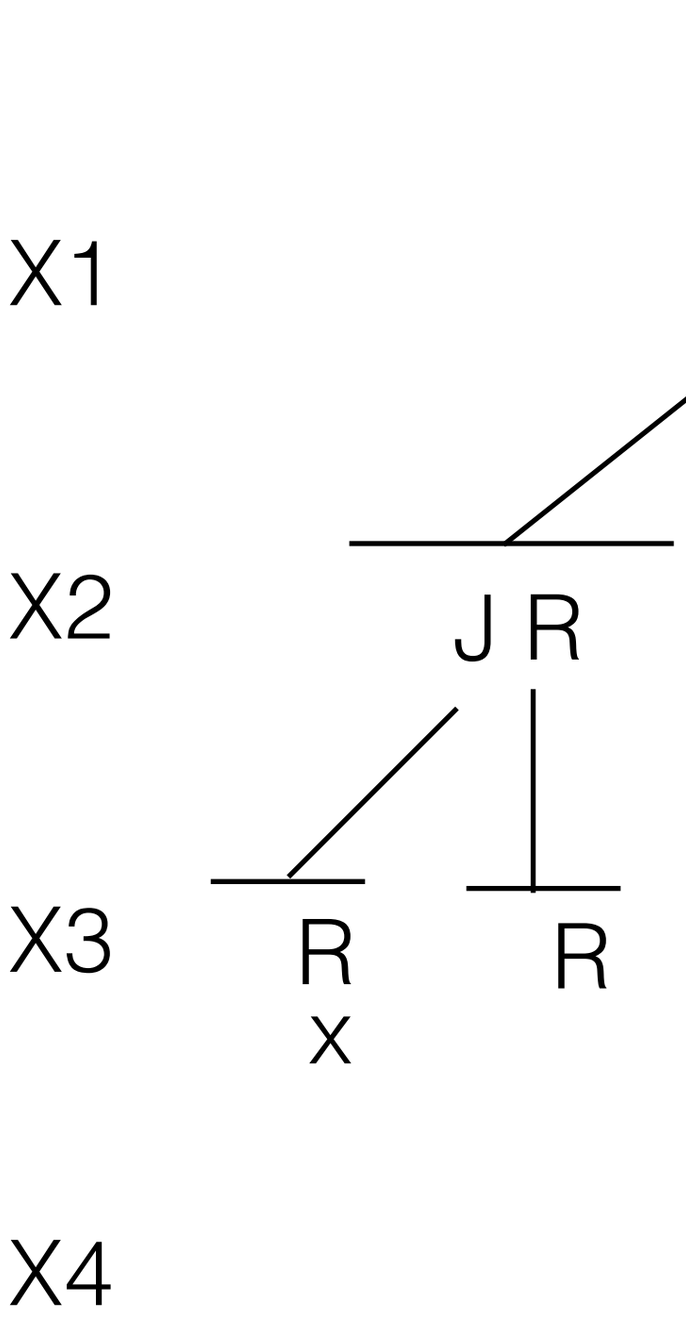
FC on running example



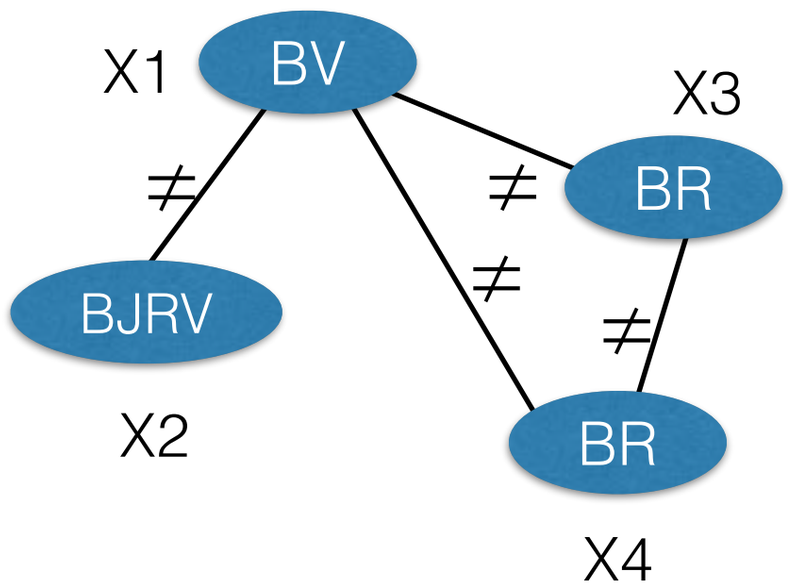
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



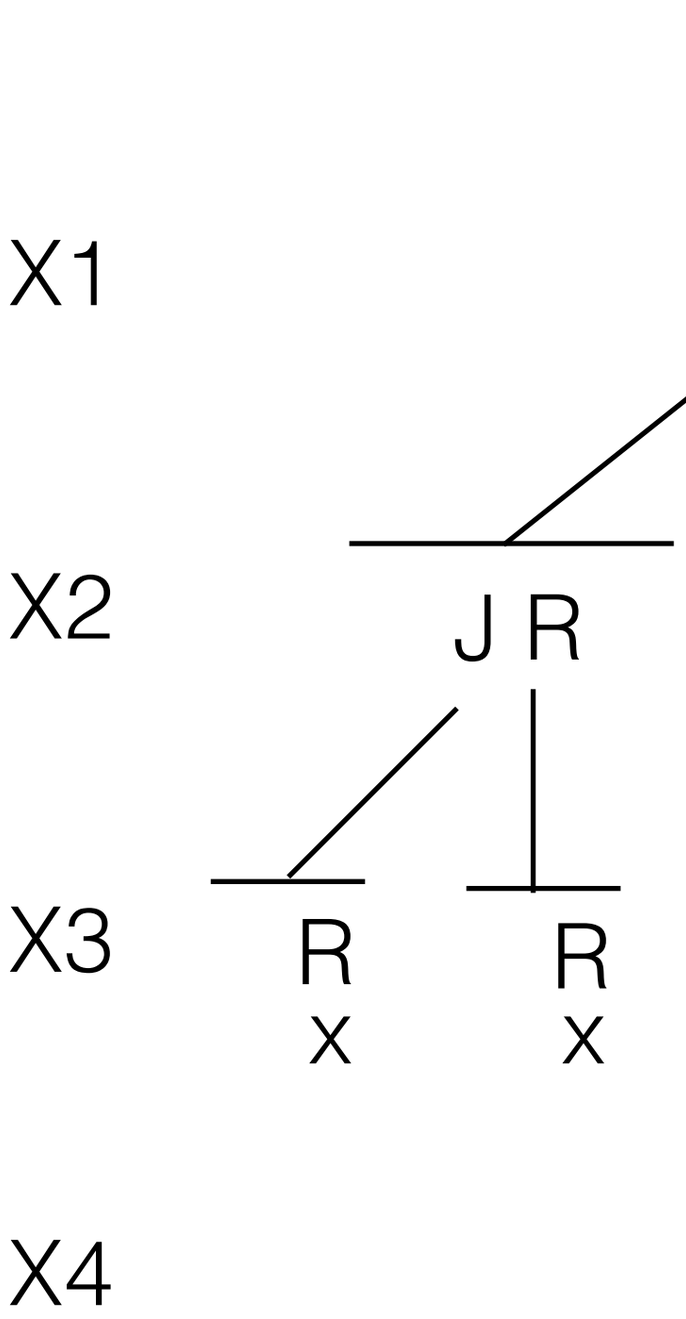
FC on running example



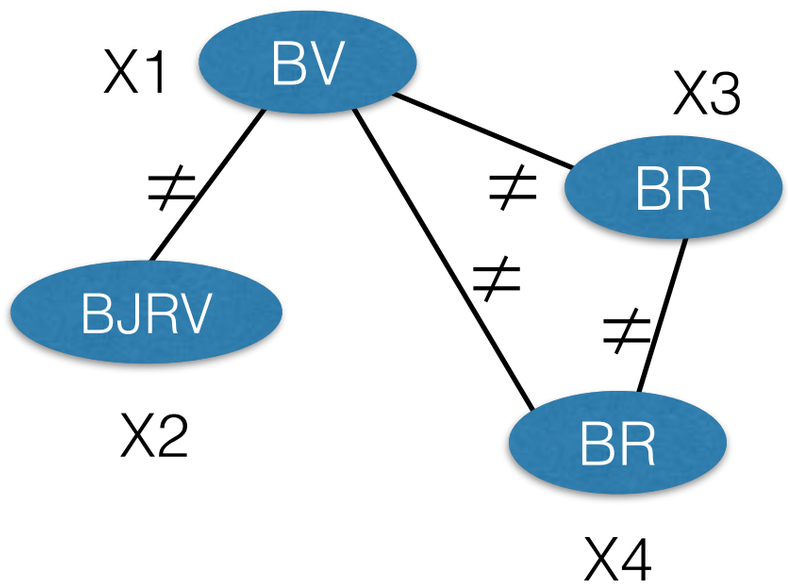
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



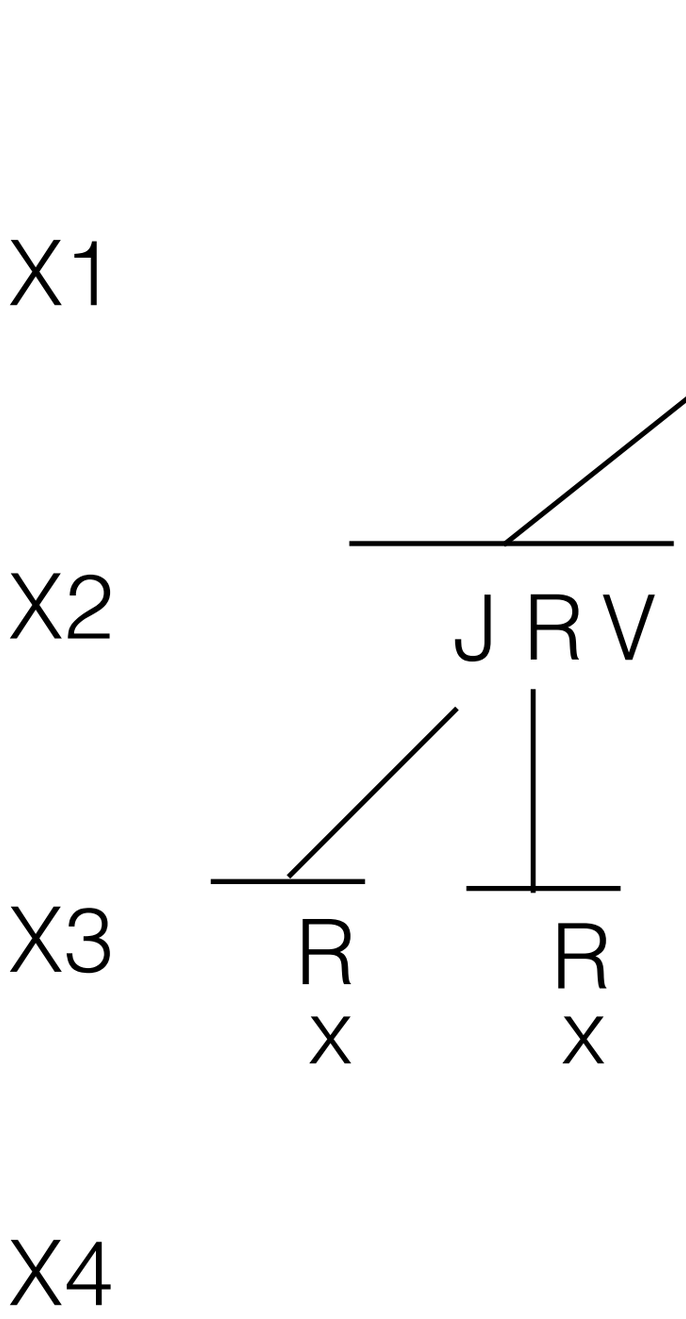
FC on running example



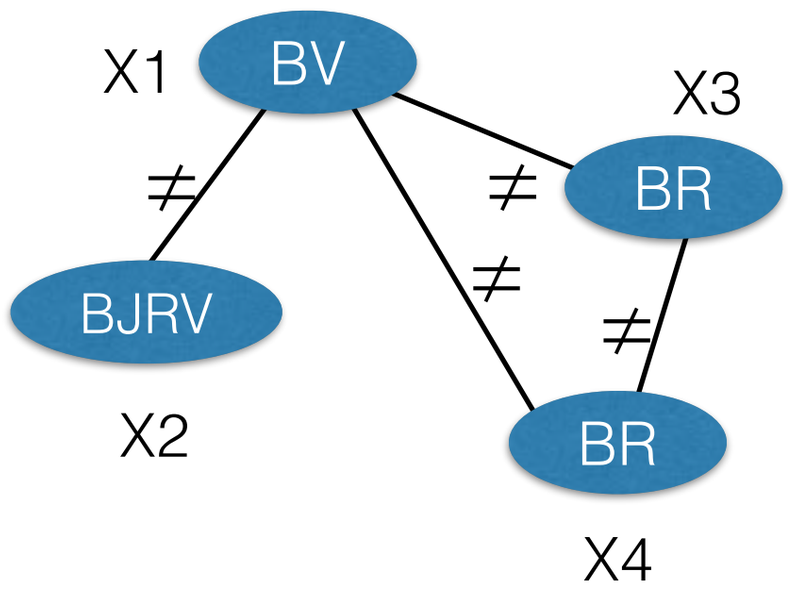
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



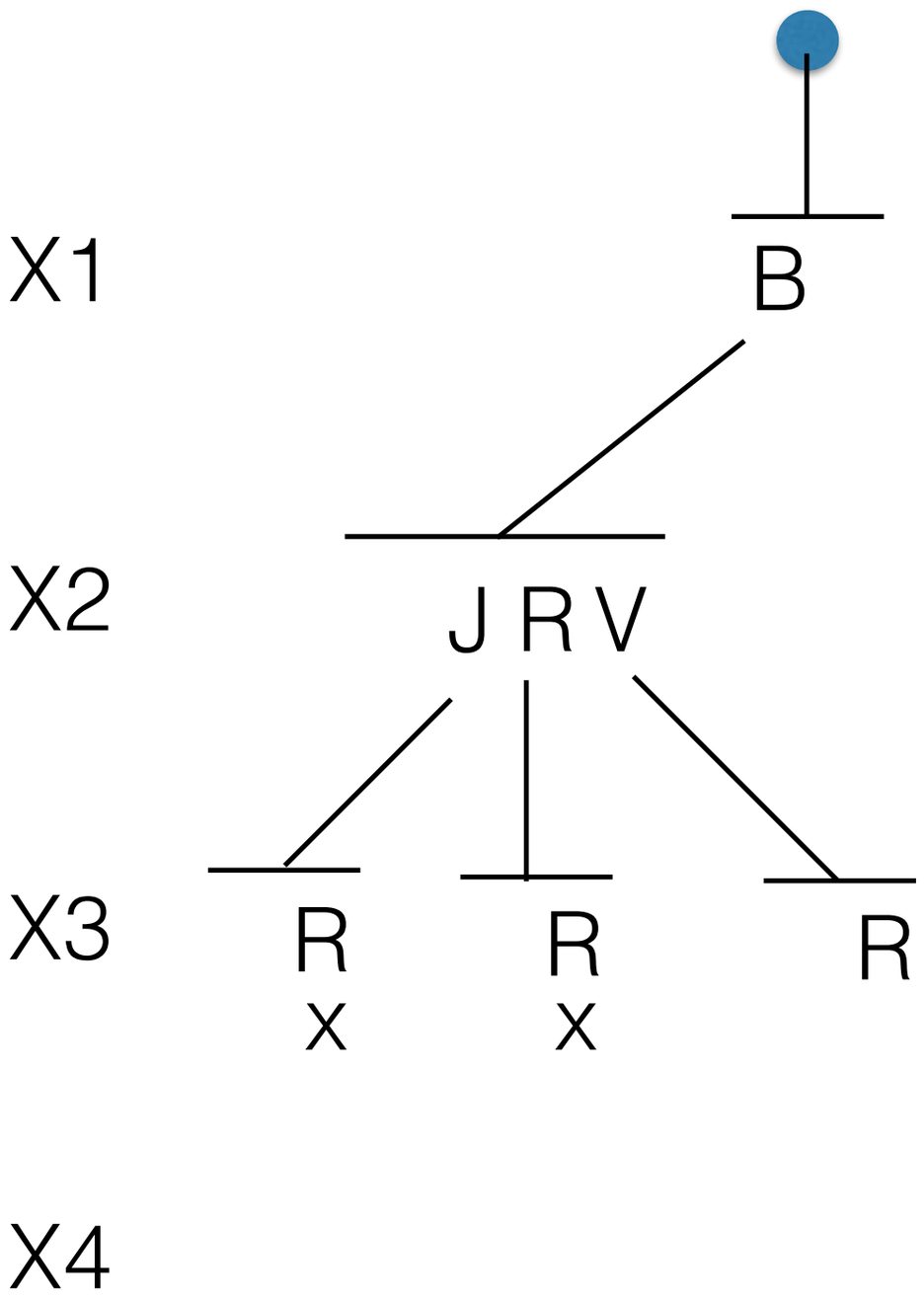
FC on running example



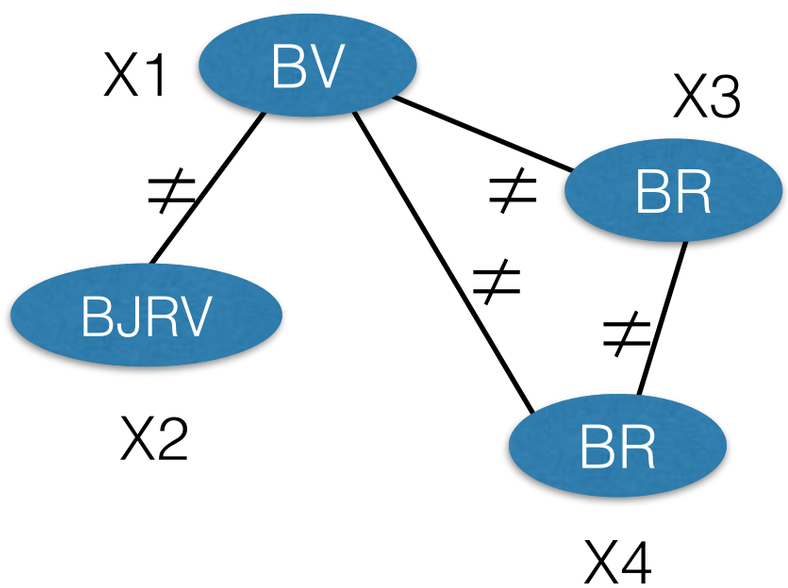
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



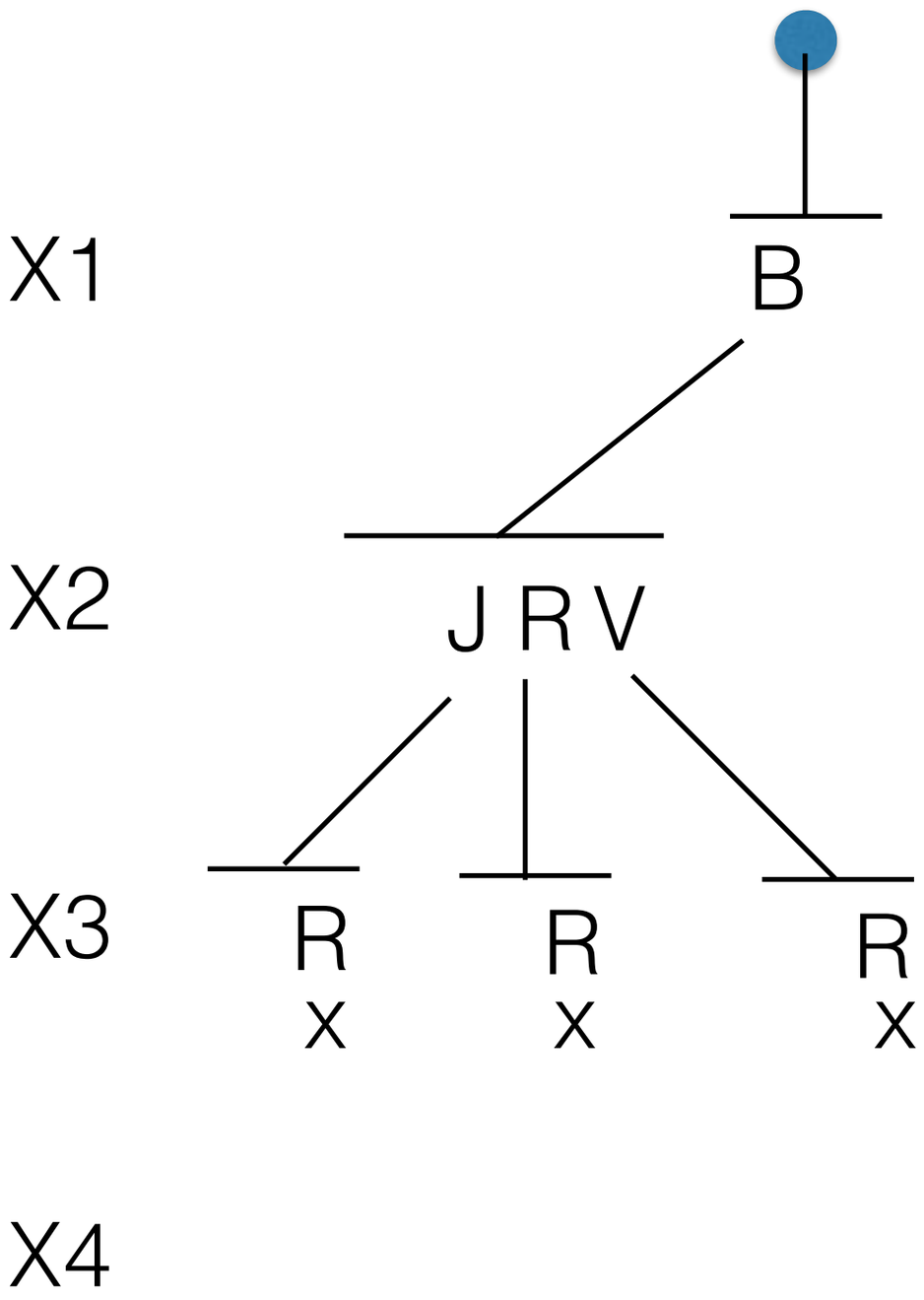
FC on running example



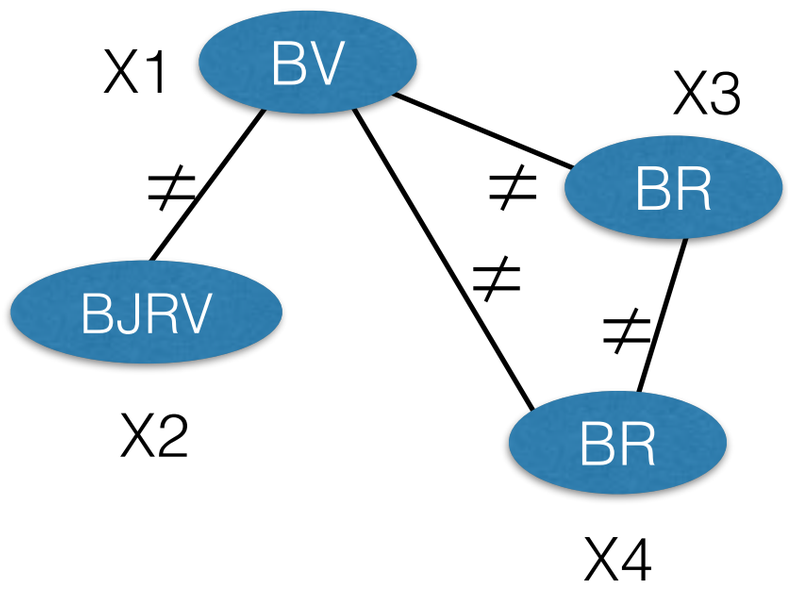
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



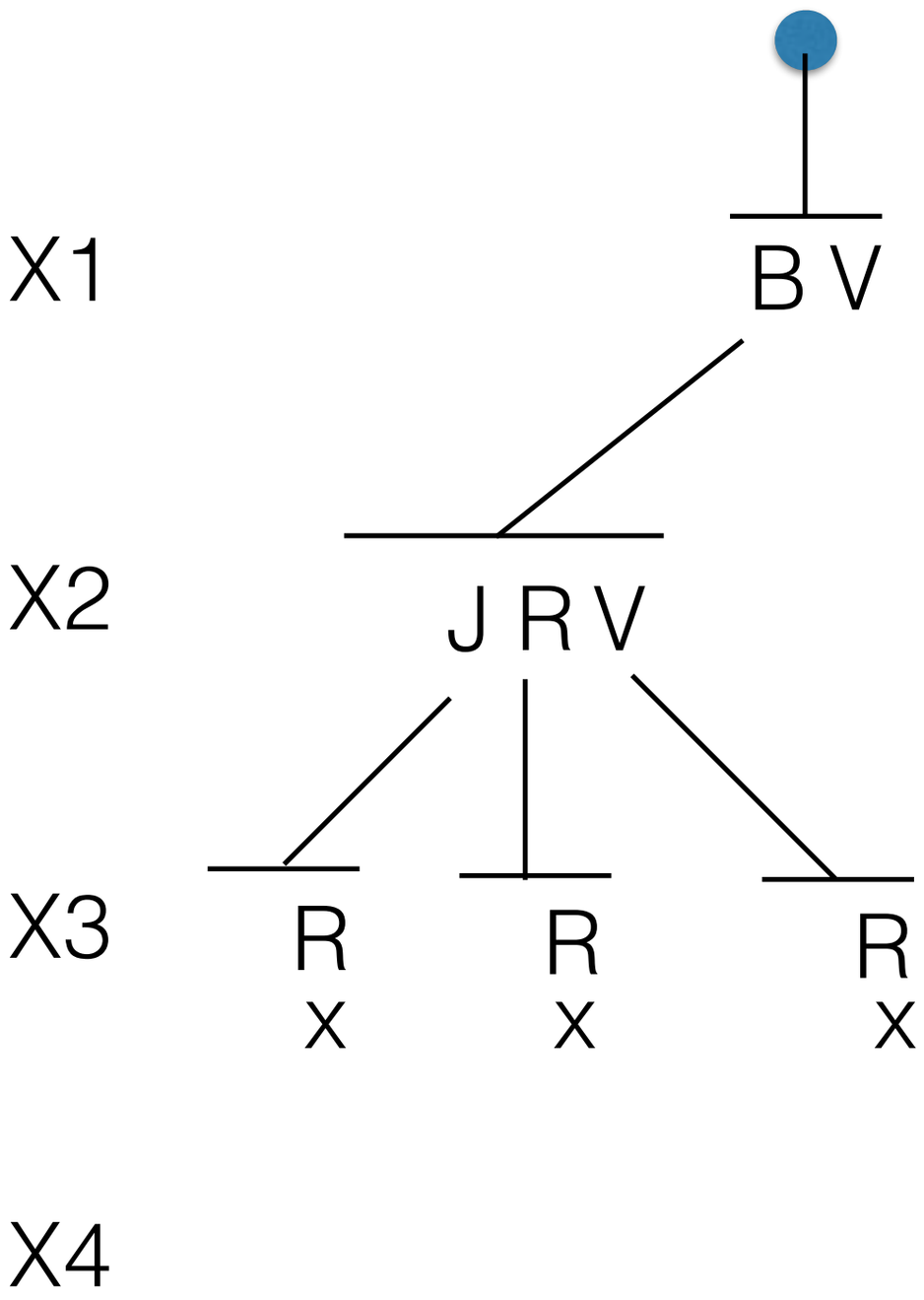
FC on running example



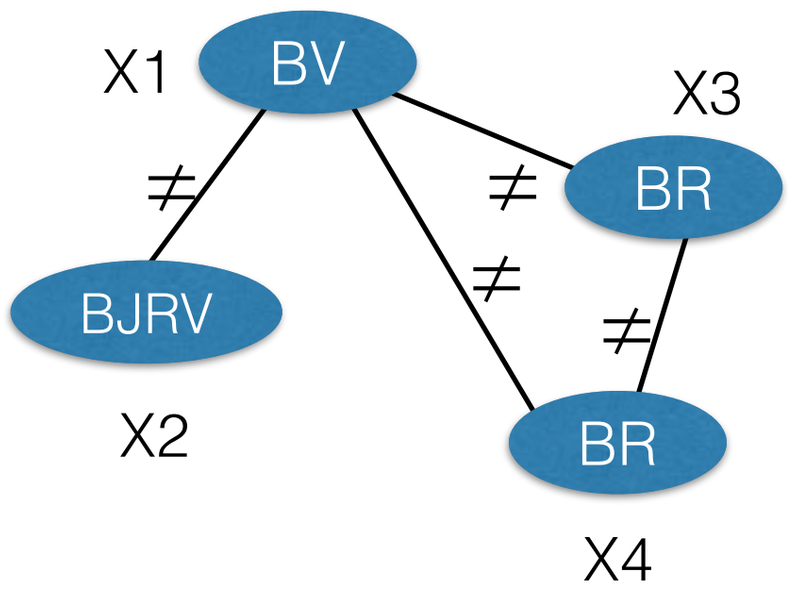
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



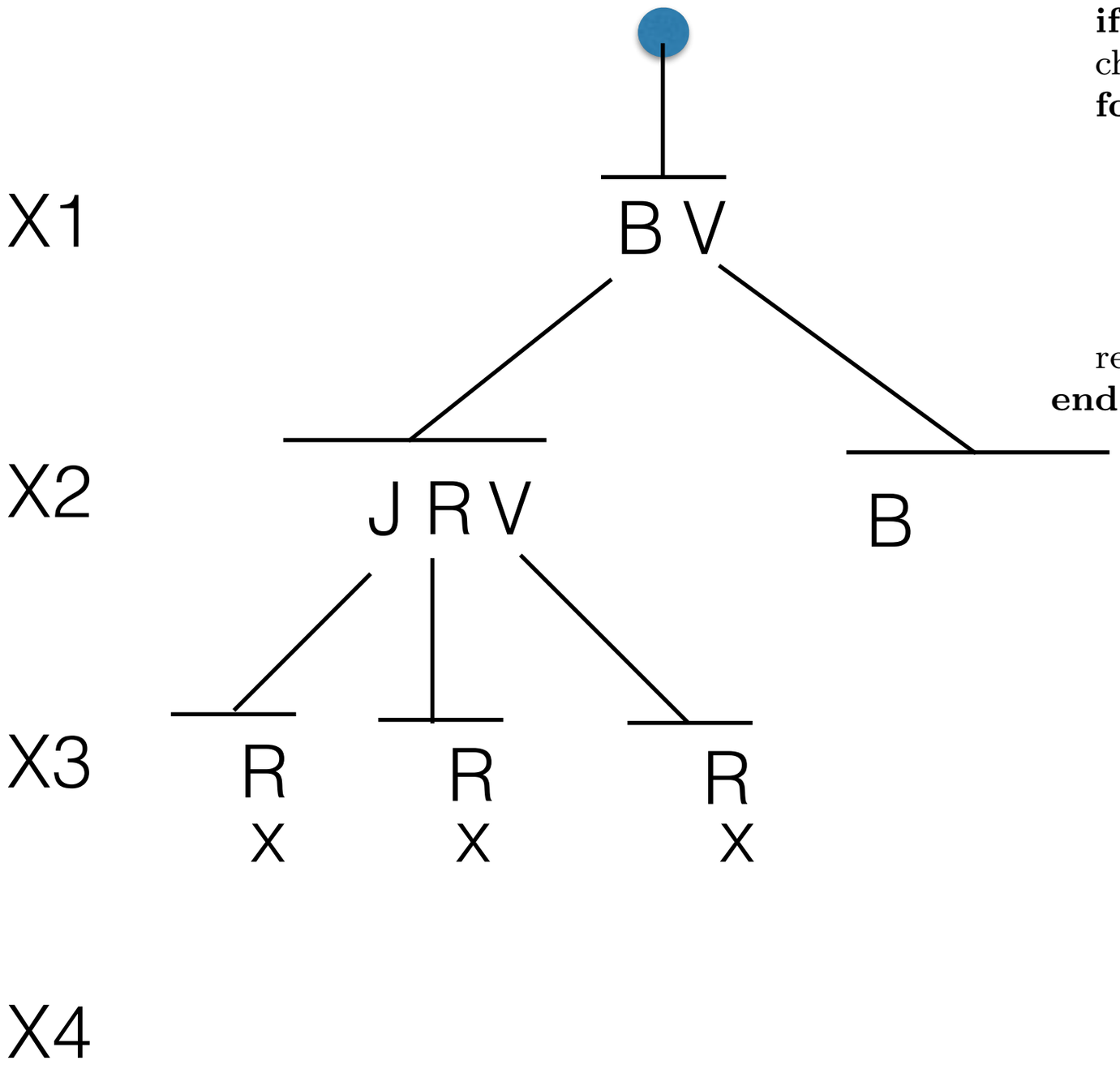
FC on running example



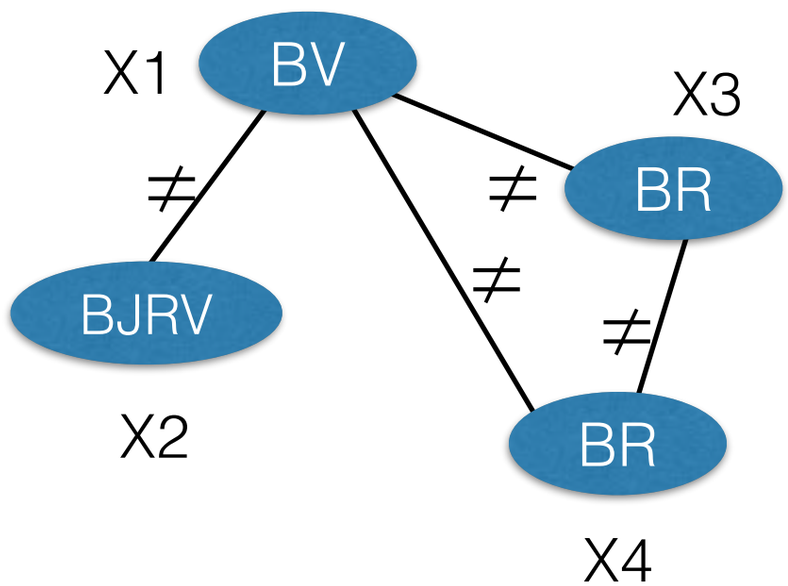
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



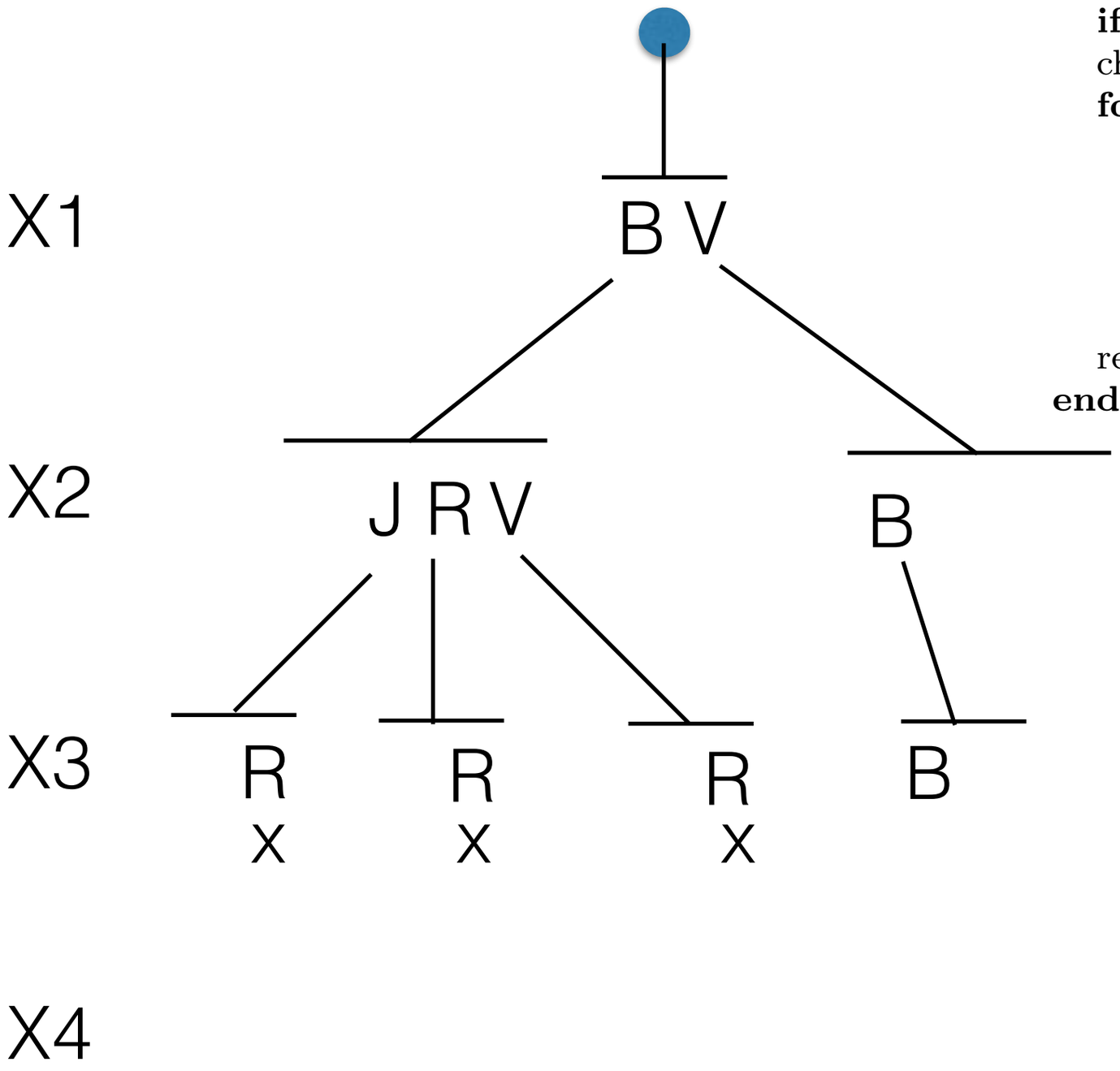
FC on running example



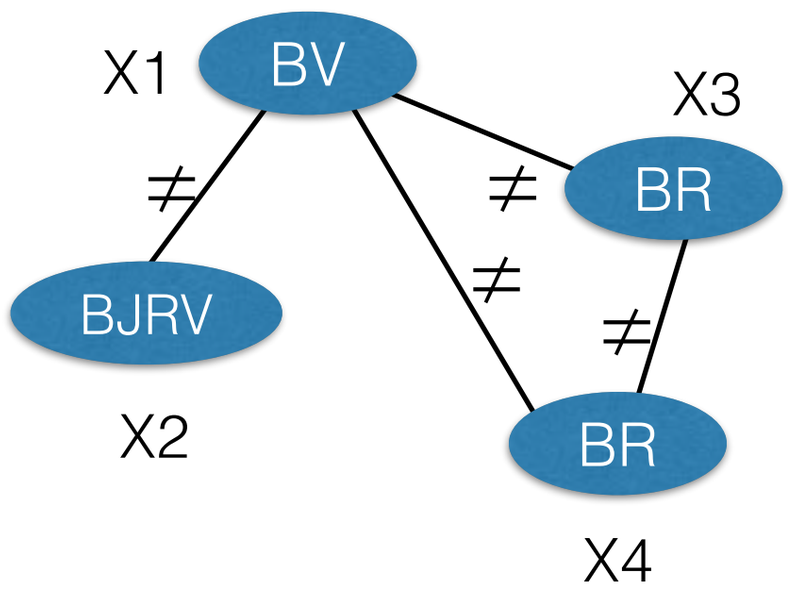
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



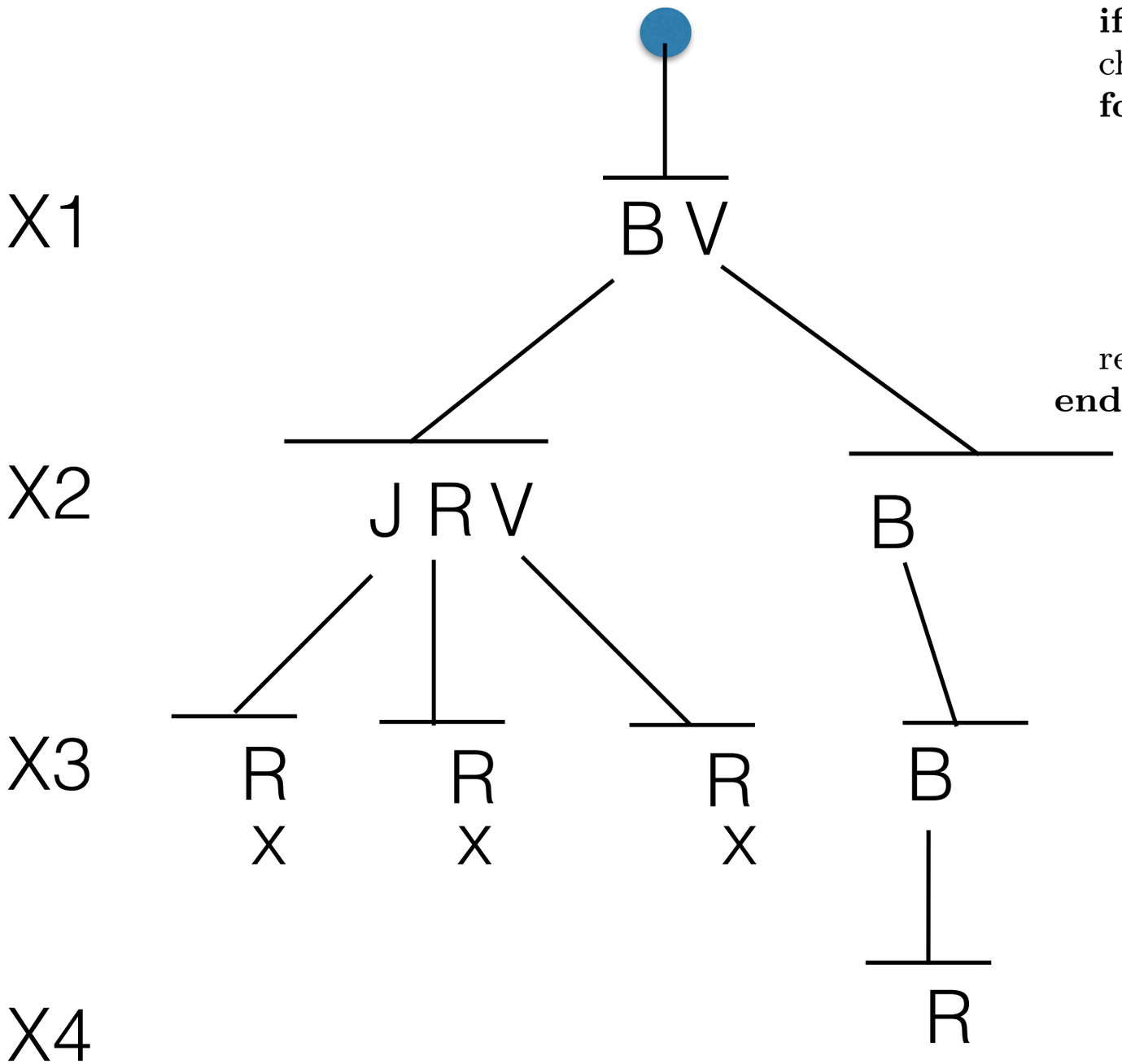
FC on running example



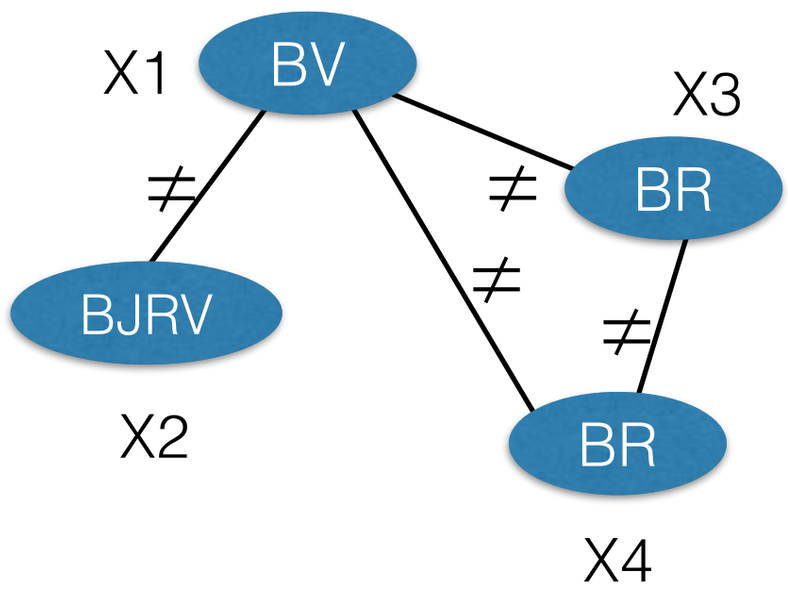
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end



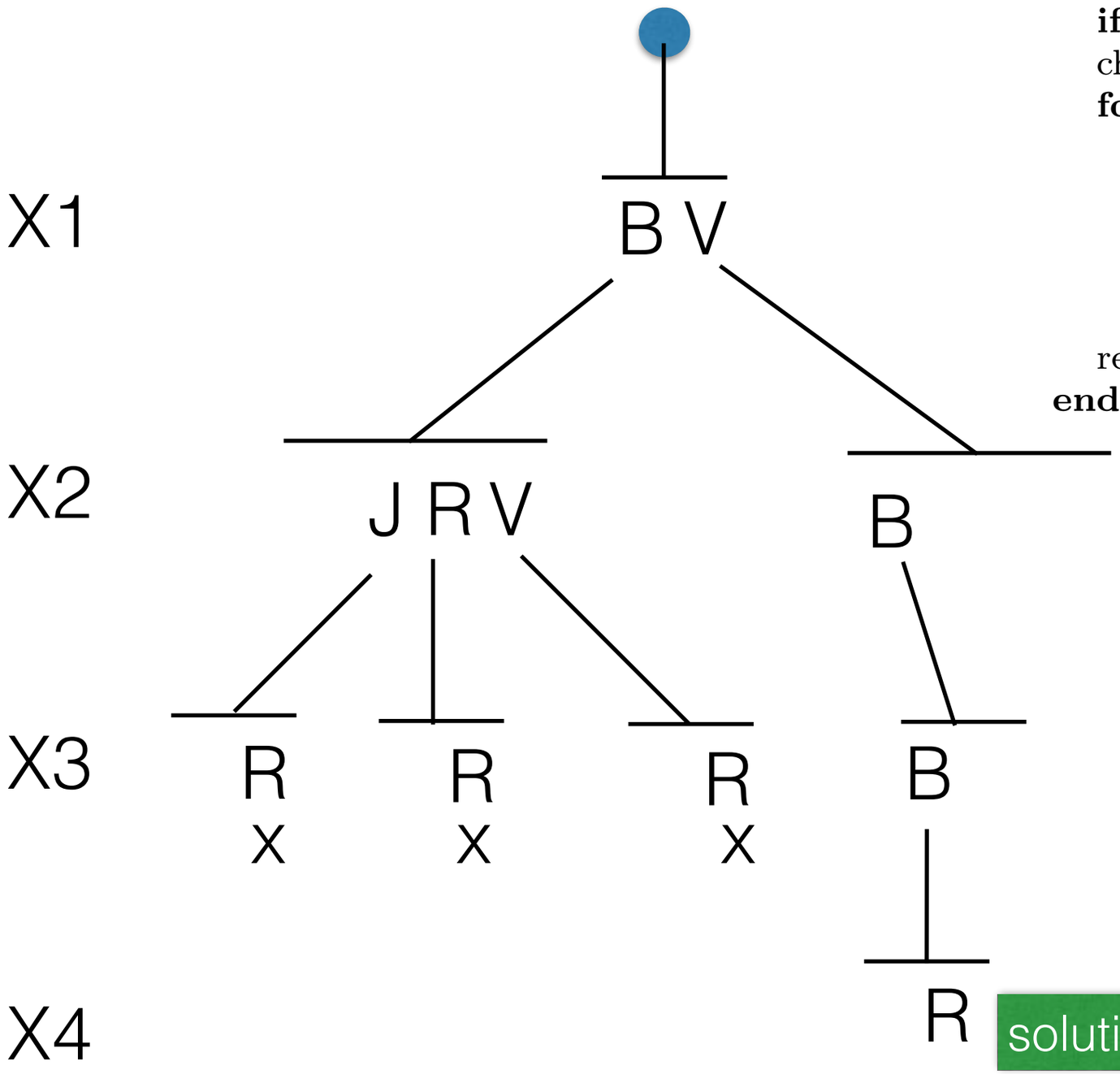
FC on running example



function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end

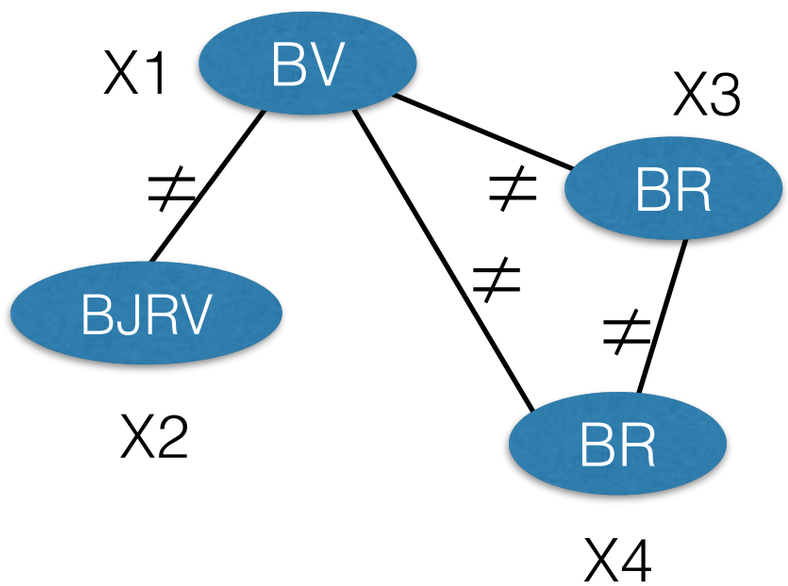


FC on running example

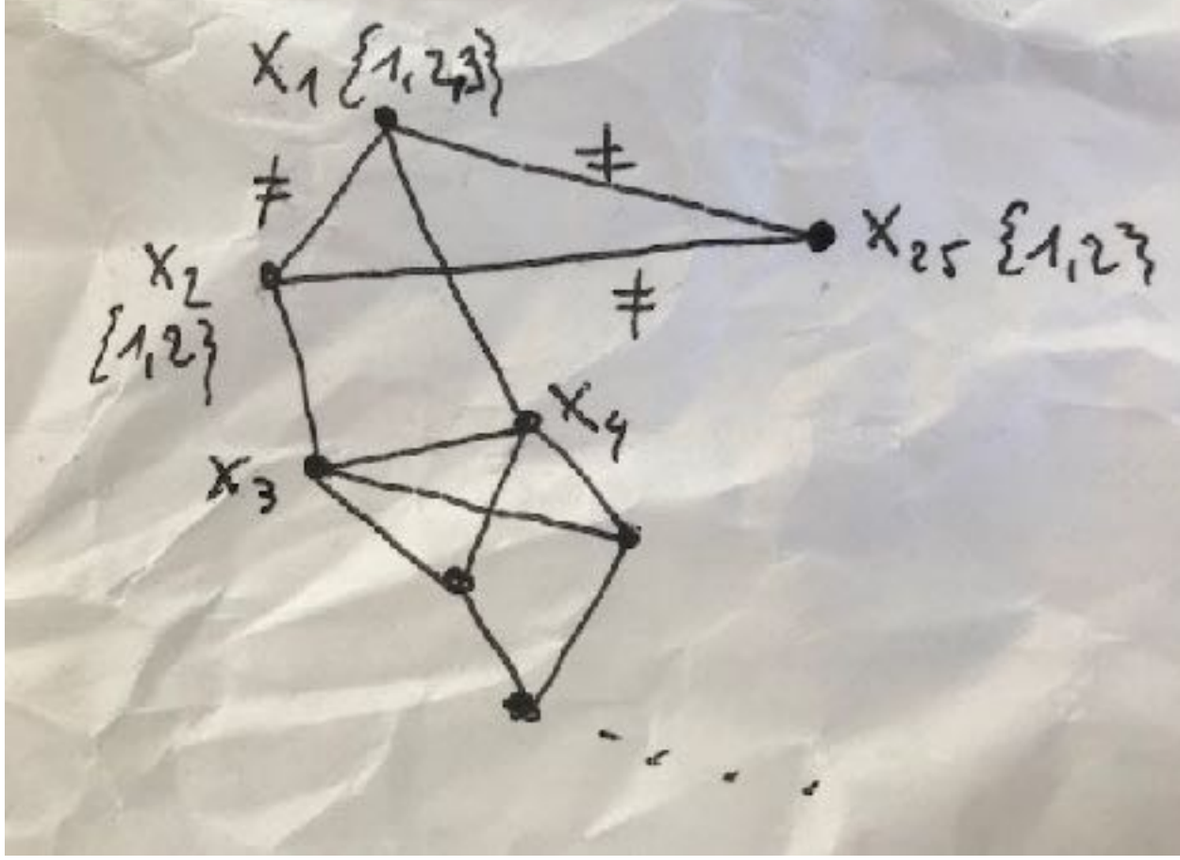


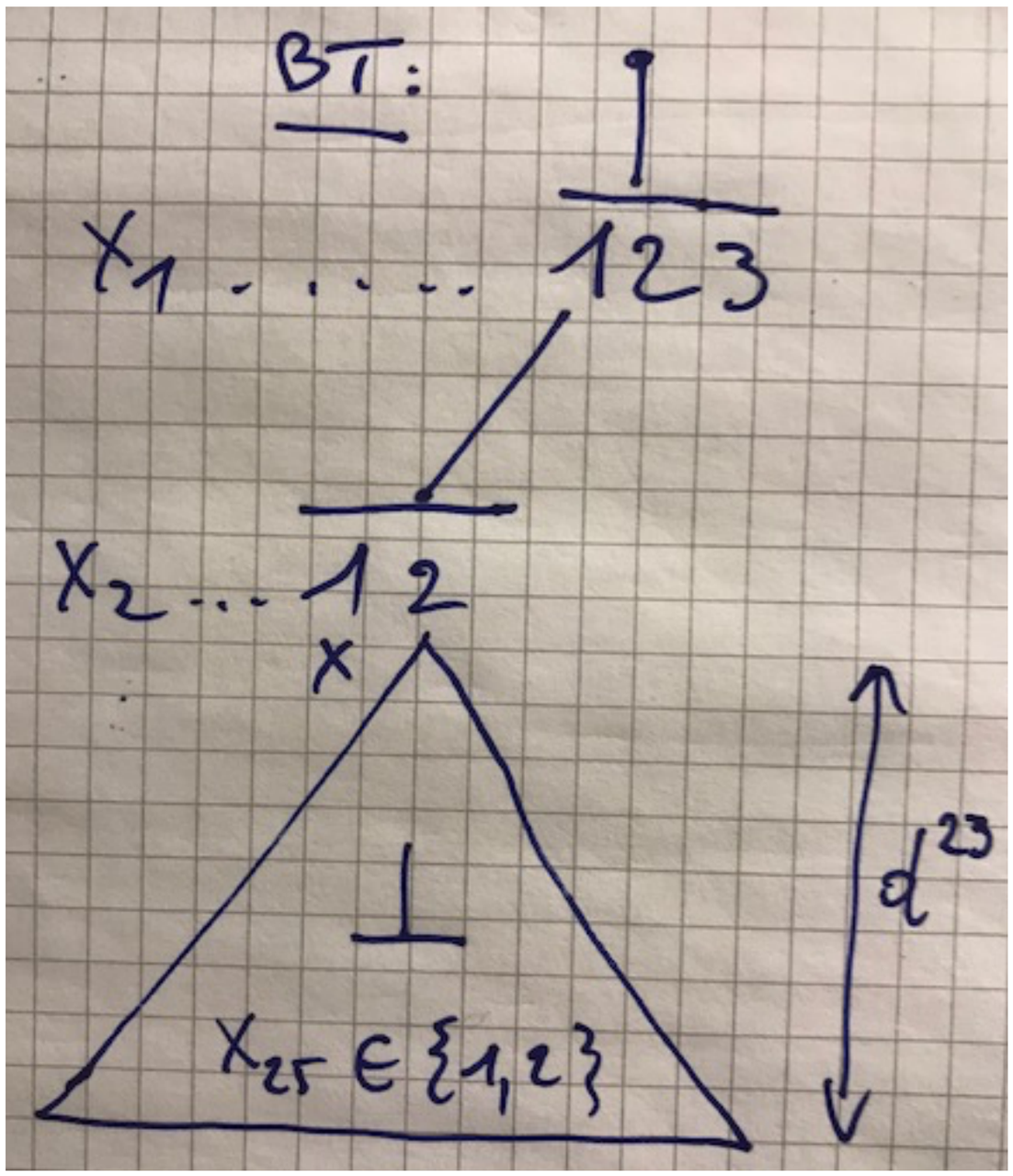
function FC(N : network ; I : instantiation) : **Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable x_i not in I
  for each $v_i \in D(x_i)$ **do**
    $D(x_i) \leftarrow \{v_i\}$
    remove all (x_j, v_j) inconsistent with $I \leftarrow I \cup (x_i, v_i)$
    if not empty domain **then**
      if FC(N, $I \cup (x_i, v_i)$) **then** return **true**
    restore all values pruned because of $X_i = v_i$
  return **false**
end

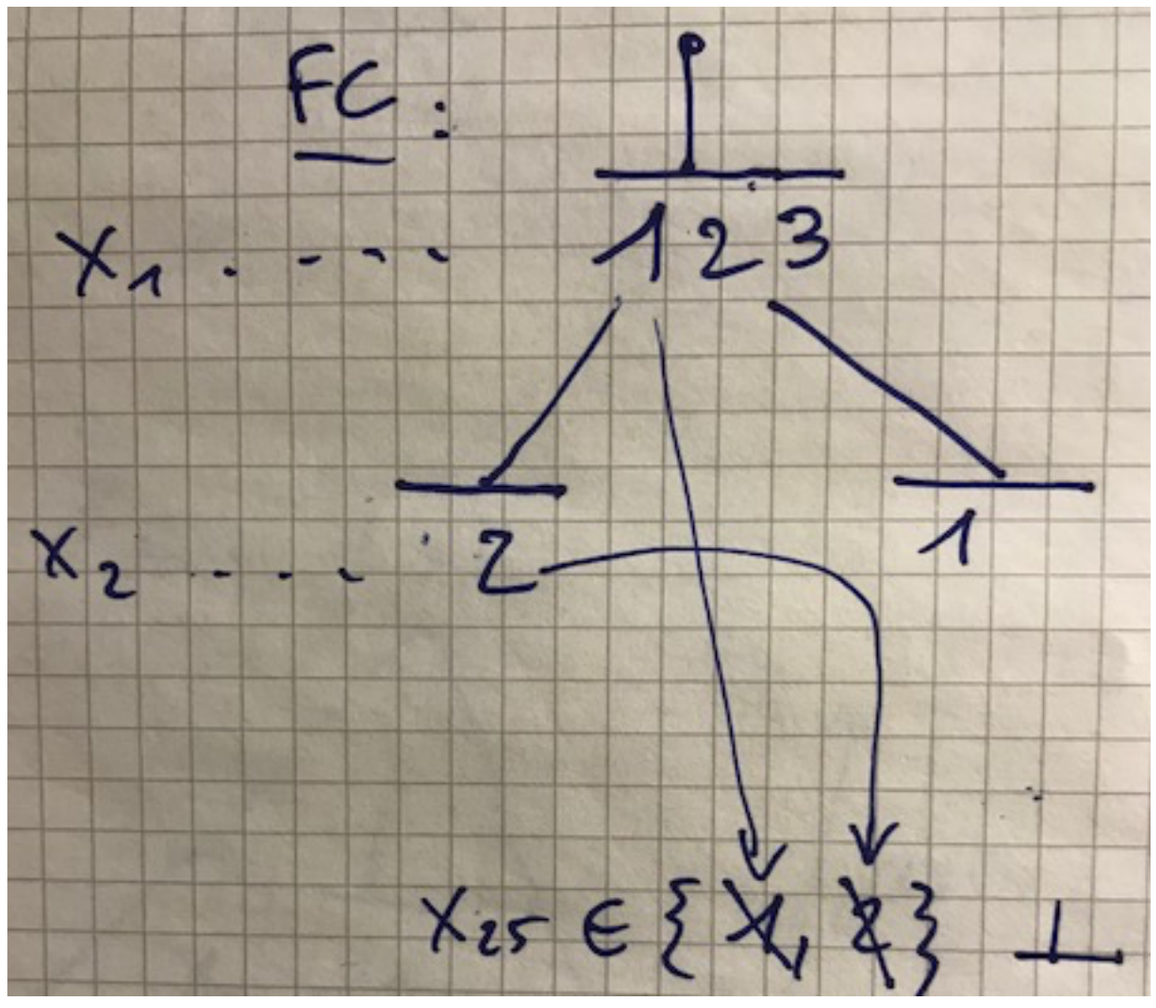
solution



Forward checking







Maintaining Arc Consistency (MAC)

```
Function MAC(N)
    AC(N)
    if empty domain then return 0
    if N fully instantiated then return 1
    select a non singleton variable Xi
    select a value vi ∈ D(Xi)
    return MAC(N+{Xi=vi}) or MAC(N+{Xi≠vi})
```

MAC on running example

X1

X2

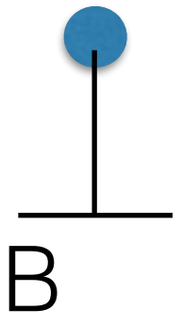
X3

X4

X1	X2	X3	X4
B	B	B	B
	J		
	R	R	R
V	V		

MAC on running example

X1



X2

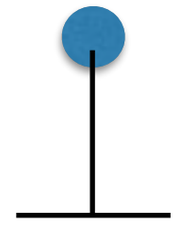
X3

X4

X1	X2	X3	X4
B	B	B	B
	J		
	R	R	R
V	V		

MAC on running example

X1



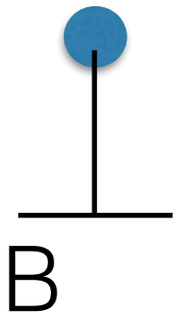
X2

X3

X4

X1	X2	X3	X4
B	B	B	B
	J		
	R	R	R
V	V		

MAC on running example



X1

B

X1	X2	X3	X4
B	~~B~~	~~B~~	~~B~~
	J		
	R	R	R
V	V		

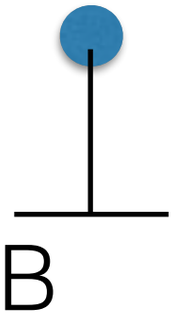
X2

X3

X4

MAC on running example

X1



X2

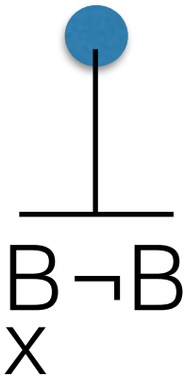
X3

X4

X1	X2	X3	X4
B	B	B	B
	J		
	R	R	R
V	V		

MAC on running example

X1



X1	X2	X3	X4
B	B	B	B
	J		
	R	R	R
V	V		

X2

X3

X4

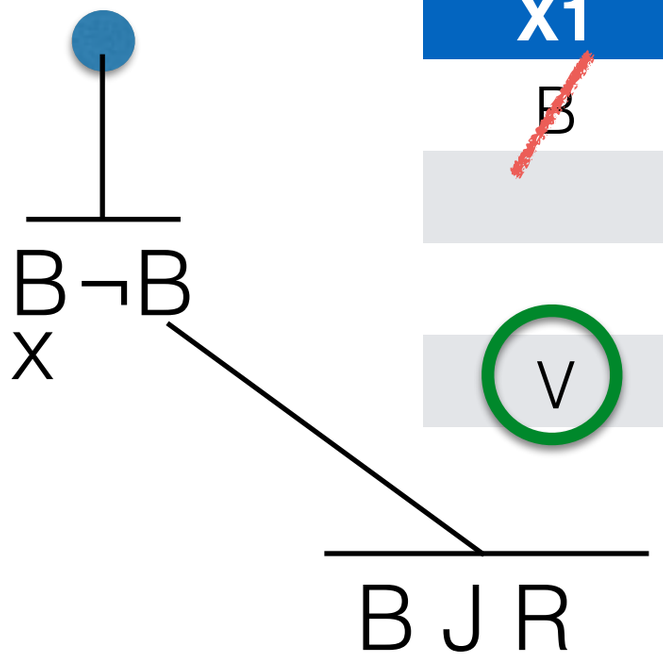
MAC on running example

X1

X2

X3

X4



X1	X2	X3	X4
B	B	B	B
	J		
	R	R	R
V	V		

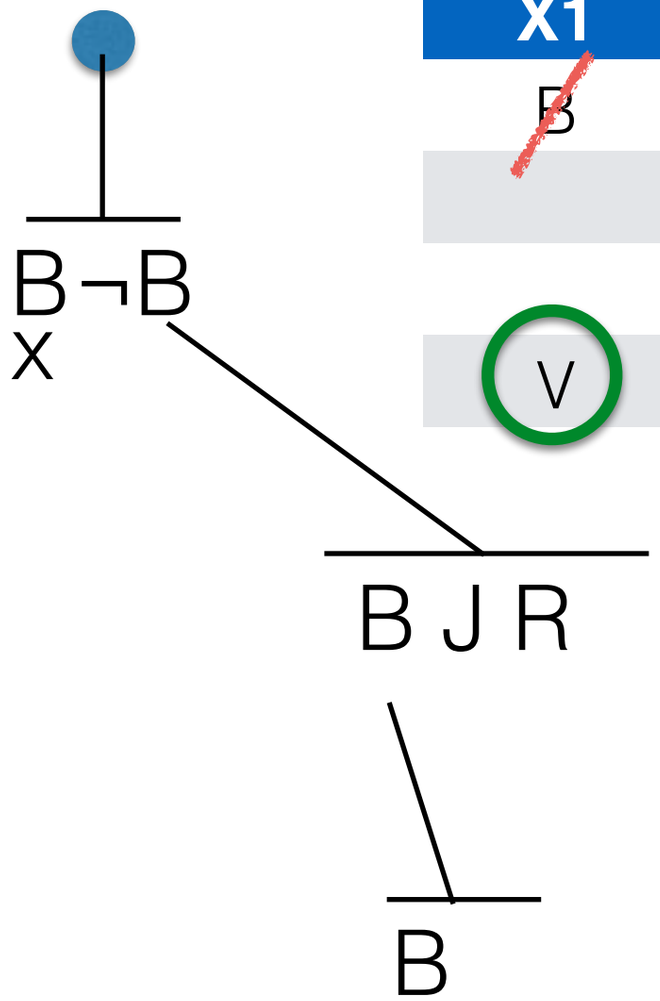
MAC on running example

X1

X2

X3

X4



X1	X2	X3	X4
B	B	B	B
	J		
	R	R	R
V	V		

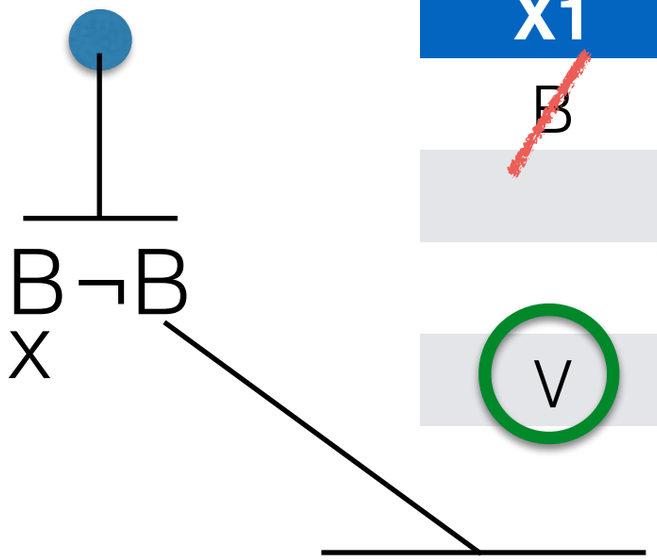
MAC on running example

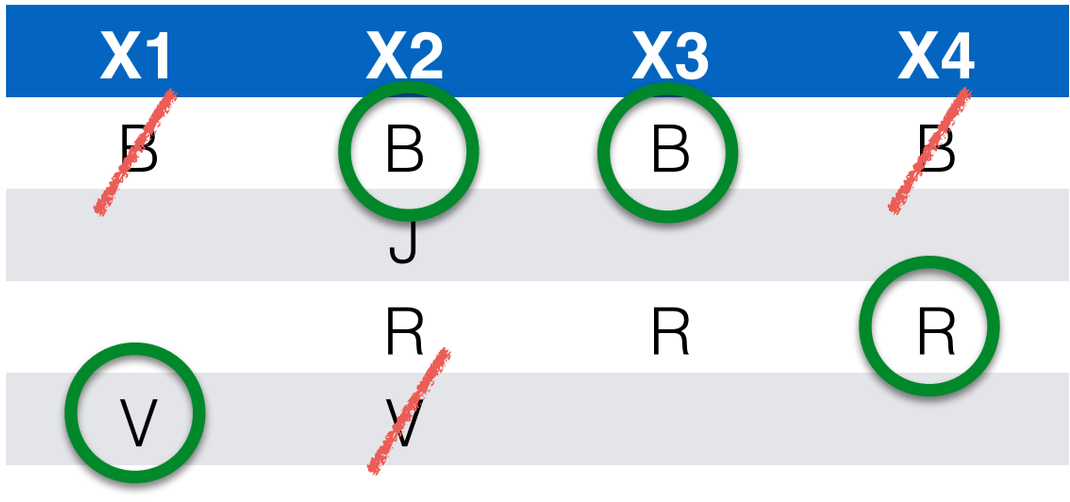
X1

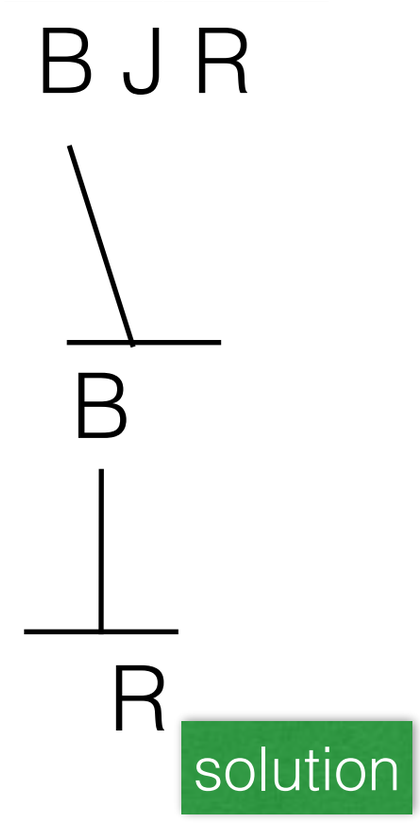
X2

X3

X4







solution

Parenthèse sur les expérimentations

Comment comparer différents algorithmes ?

- Analyse en *O(…)* —> pas adapté à la pratique
- Comparer les performances sur la résolution de problèmes
- Mais quels problèmes ?
    - années 80/90: n-reines (poly!), zebra problem (25 variables), random problems

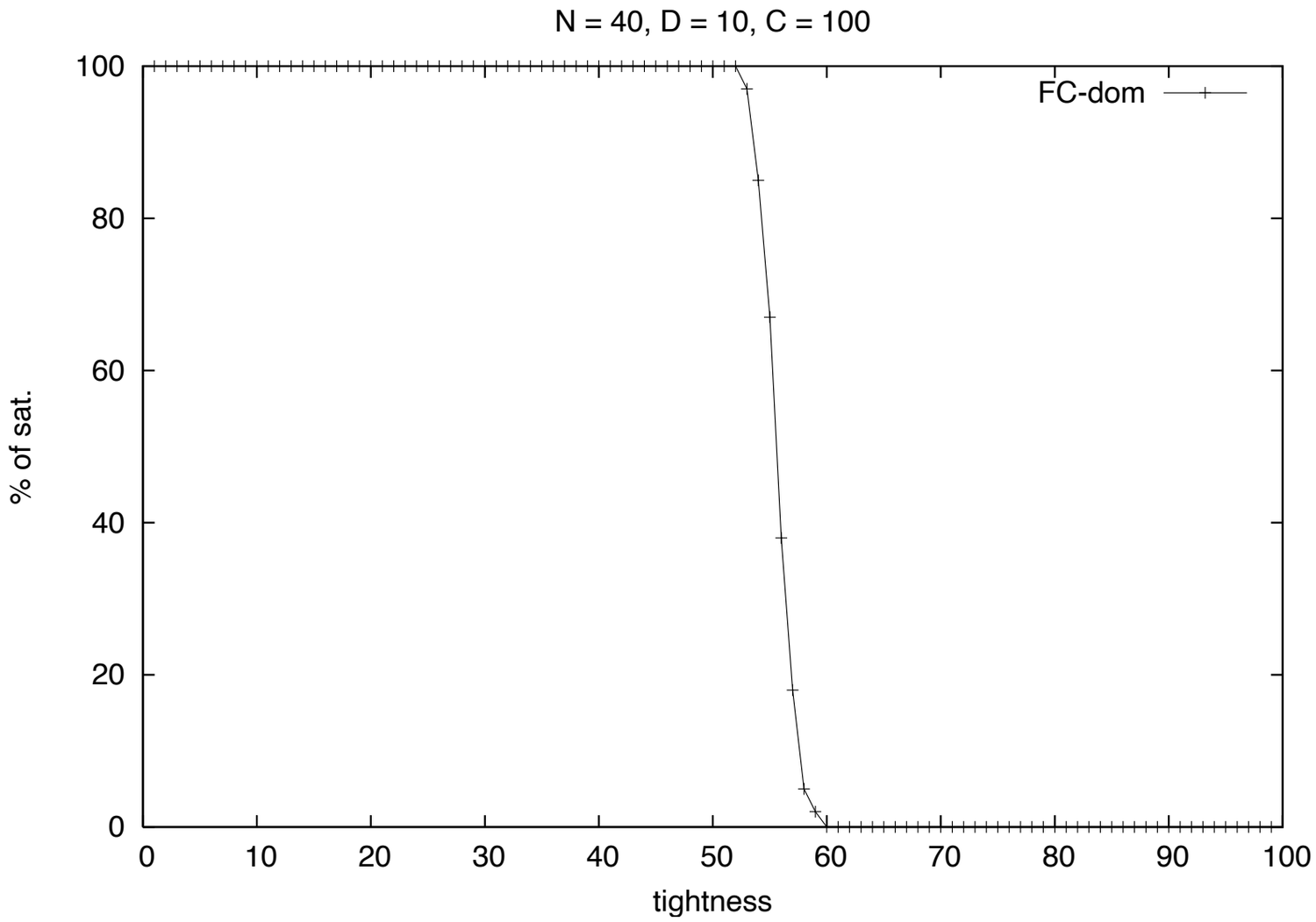
Random problems (binary)

- *n*: number of variables
- *d*: size of domains
- *p1*: proportion of constraints
  - number of constraints = $p1 \times n(n-1)/2$
- *p2*: tightness of constraints
  - number of forbidden tuples = $p2 \times d^2$

Cheeseman, Kanefsky, Taylor's conjecture

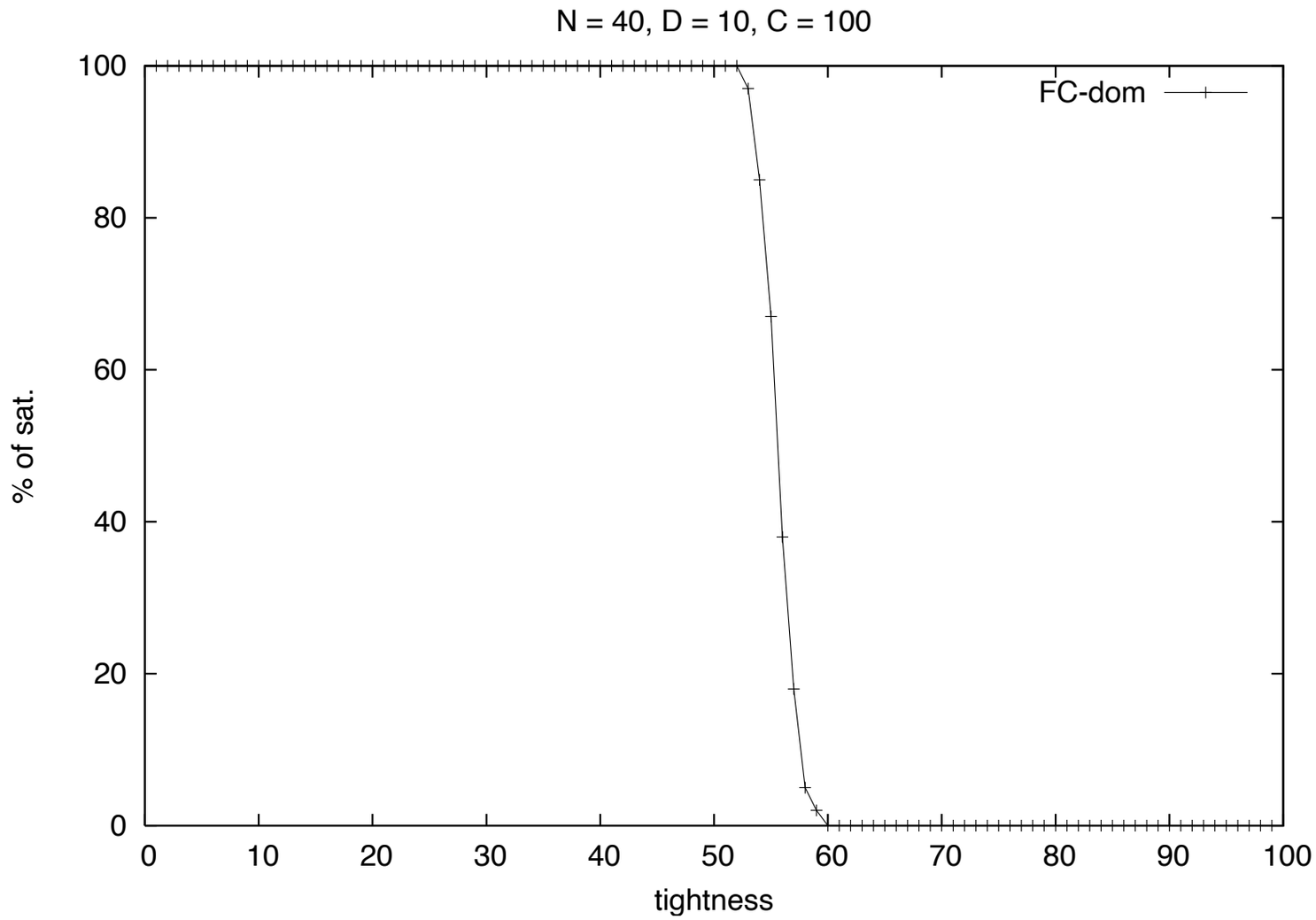
- Take any NP-complete problem
- Characterize it by *k* parameters
  - 3SAT: #atoms, #clauses
  - 3COL: #nodes, #edges
  - 2CSP: #variables, #values, #constraints, #tuples
- Fix *k-1* parameters and vary the free one
- Generate 100 instances for each value of *k*

Phase transition

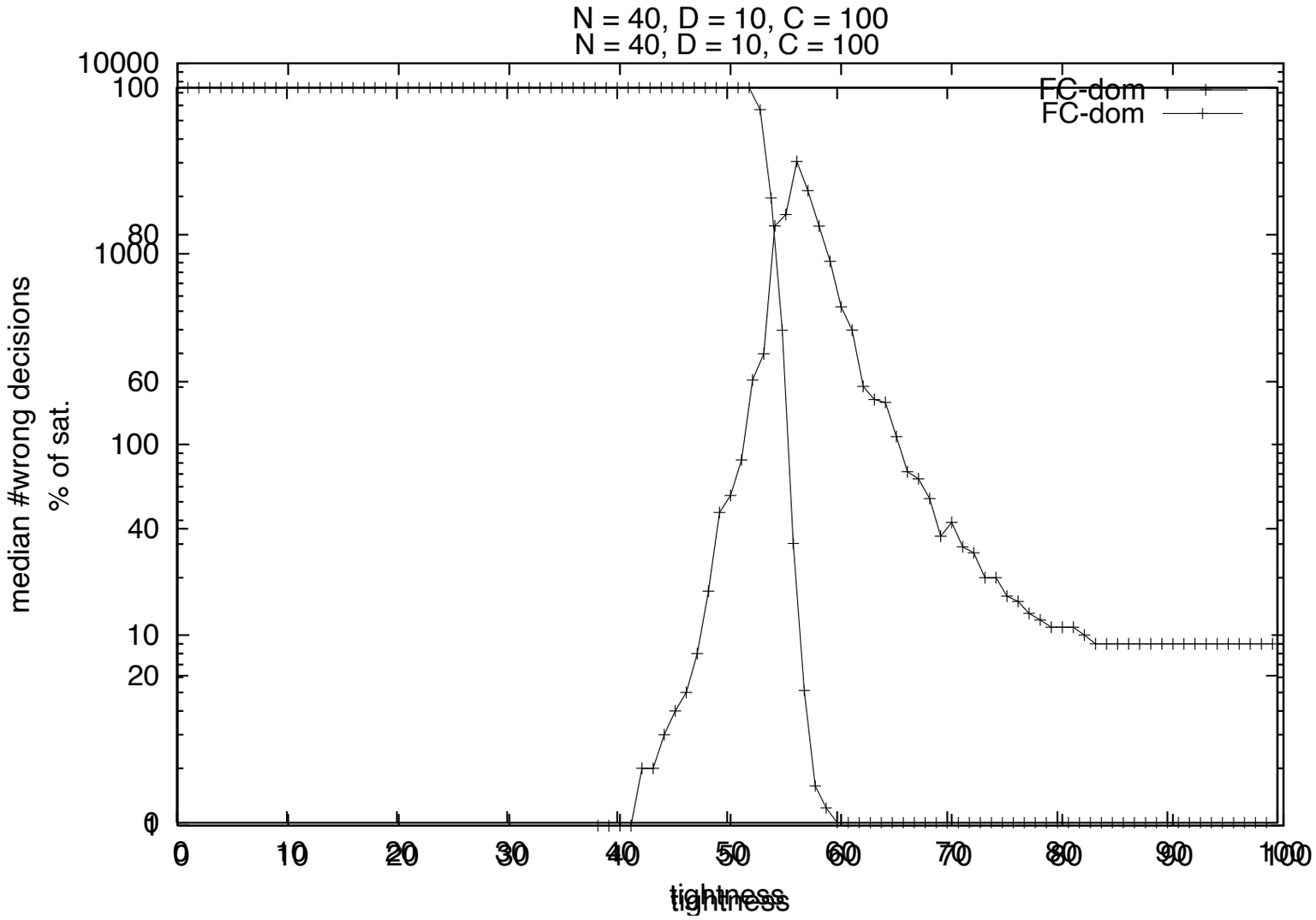


We observe a sudden *phase transition* from ‘all satisfiable’ to ‘all unsatisfiable’ while *k* (in/de)creases

Solving these instances



Solving these instances



We observe a sudden increase of hardness at the phase transition

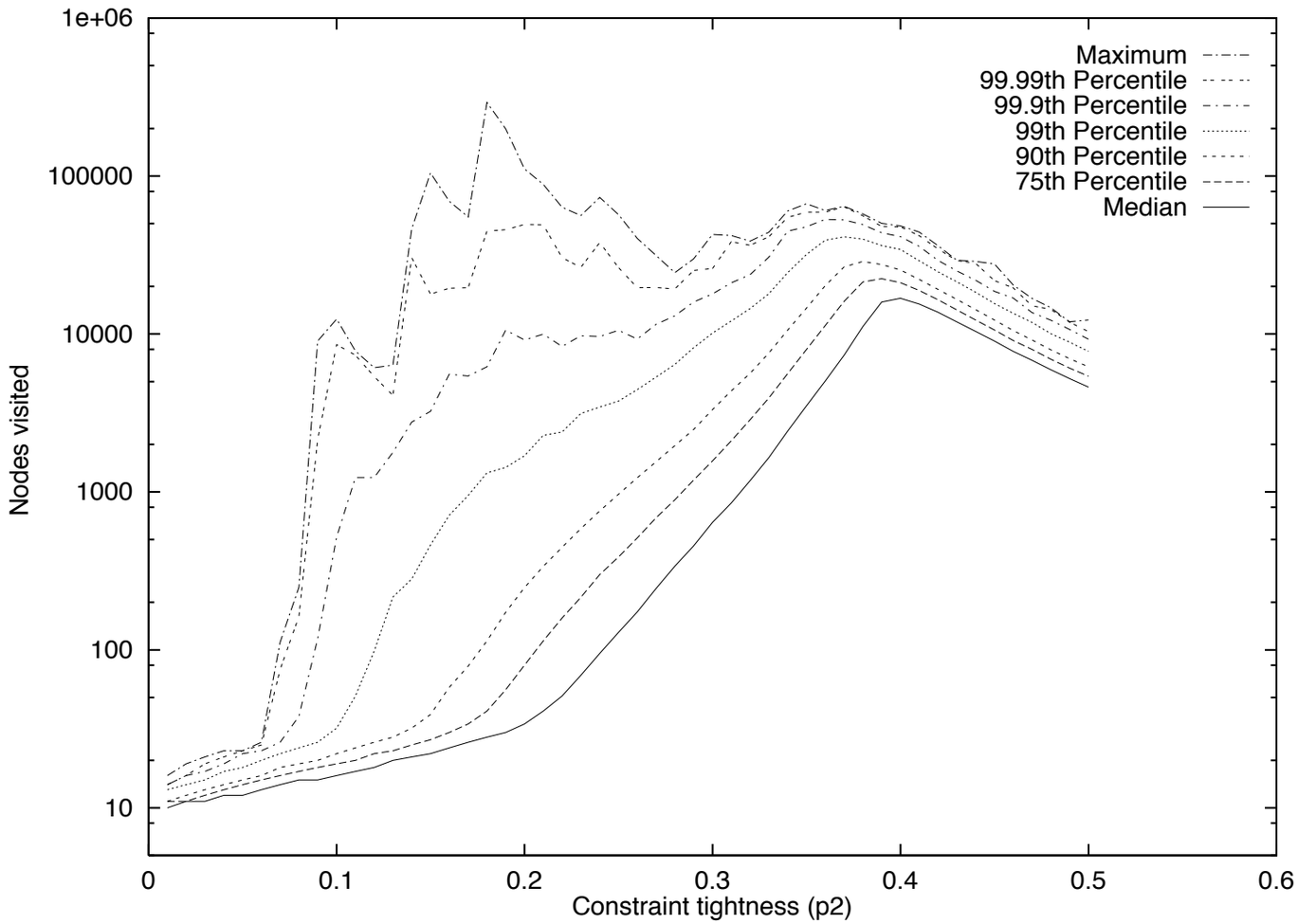
Median versus mean

- When plotting the **mean** we observe a ‘stochastic’ behavior in the easy satisfiable region

- **➜ exceptionally hard problems (ehps)**

  - in the easy sat region

  - orders of magnitude harder than hard instances (in the phase transition)



What are these EHPs?

- They are algorithm-dependent!!
- They are as easy as the other instances in the same class **but** the algorithm made a huge mistake up in the tree

  ➜ restart strategy!

Conclusion on phase transition

- Allowed us to get rid of inefficient algorithms (FC vs MAC)
- Fortunate that real problems led to the same conclusions!

Exploration Heuristics

Variable ordering heuristics

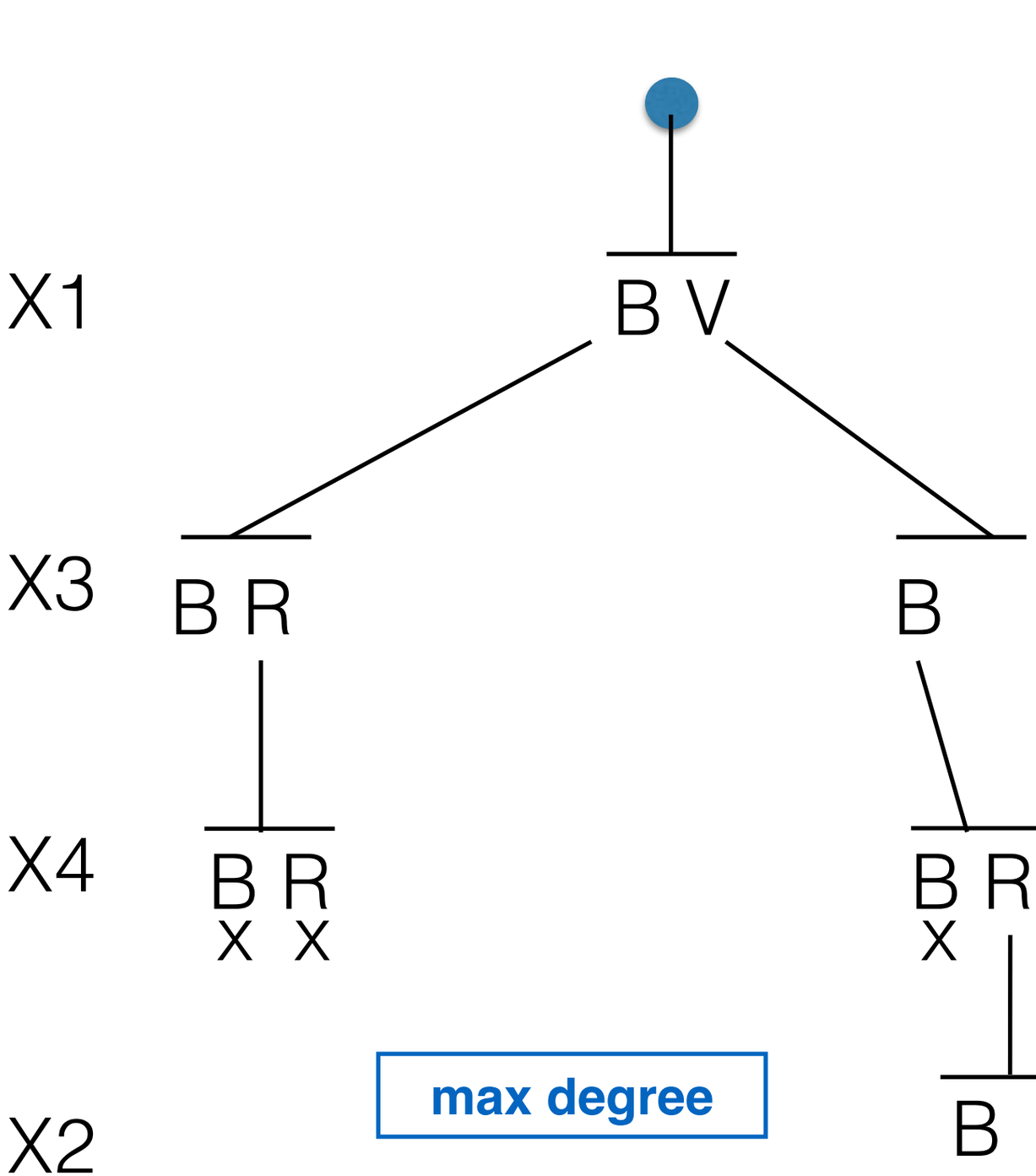
Variable ordering heuristics

- We must instantiate **all** variables
  - ➜ start by the ‘most difficult’ variables
  - ➜ reduce depth of search tree (called *fail first principle*)

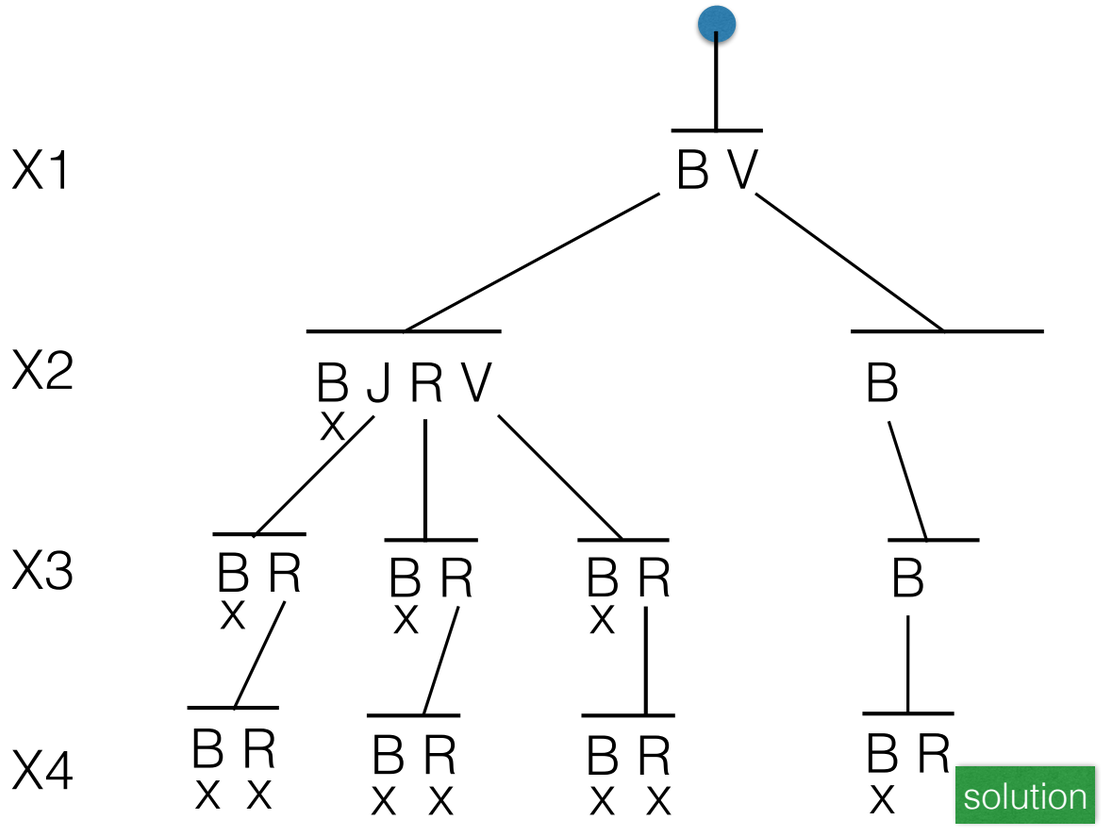
Static variable ordering (SVO)

- **deg**: select the variable with greatest degree

BT-deg



max degree



lexico

solution

Variable ordering heuristics

Variable ordering heuristics

- We must instantiate **all** variables
  - ➜ start by the ‘most difficult’ variables
  - ➜ reduce depth of search tree (called *fail first principle*)

Variable ordering heuristics

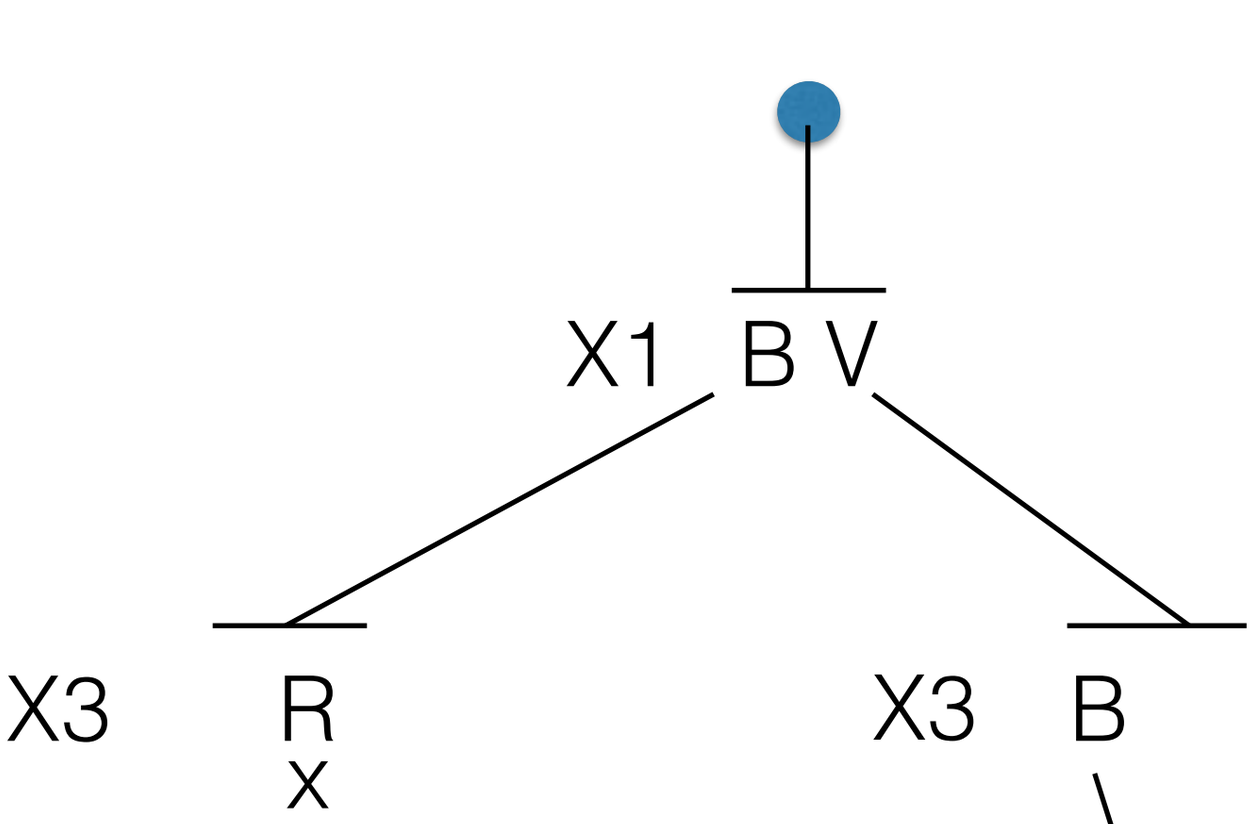
- We must instantiate **all** variables
  - ➜ start by the ‘most difficult’ variables
  - ➜ reduce depth of search tree (called *fail first principle*)

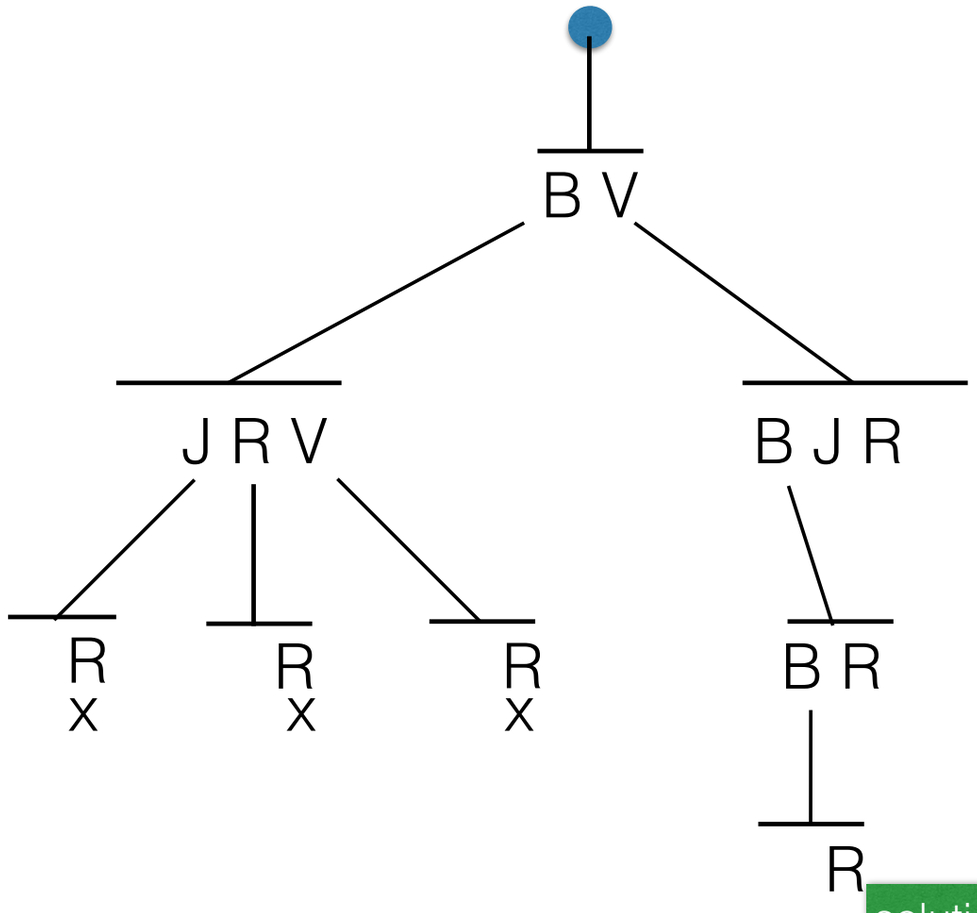
- We may need to try all values of the selected variable
  - ➜ start by small domains
  - ➜ reduce branching factor

Dynamic variable ordering (DVO)

- **dom**: select the variable with smallest domain
- dom does **not** follow the fail first principle
- …but it works well

FC-dom





solution

solution

deg versus dom

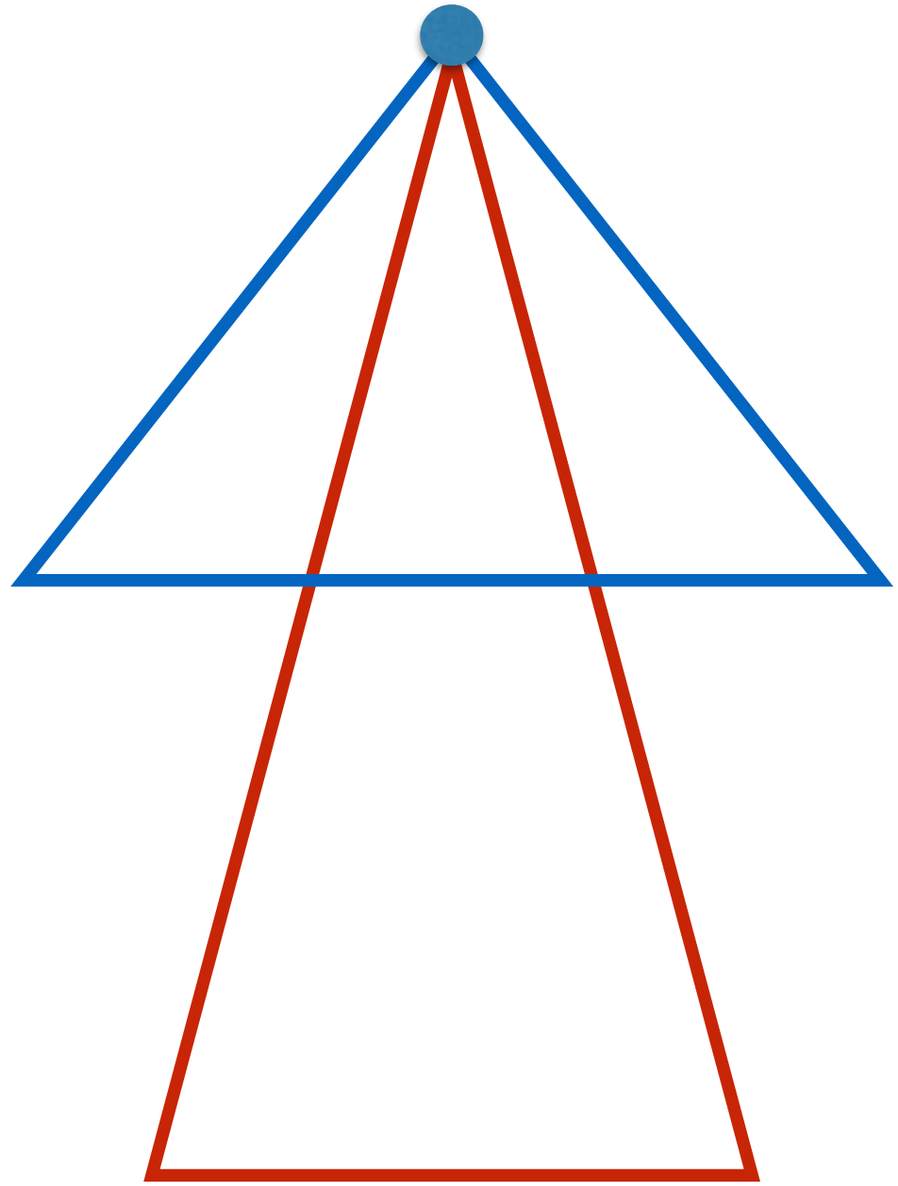
- deg reduces the ***depth*** of the search tree
- dom reduces the ***width*** of the search tree (branching factor)

- dom is good on dense networks, deg on sparse ones

Search tree shape

- difficult variables reduces the ***depth*** of the search tree

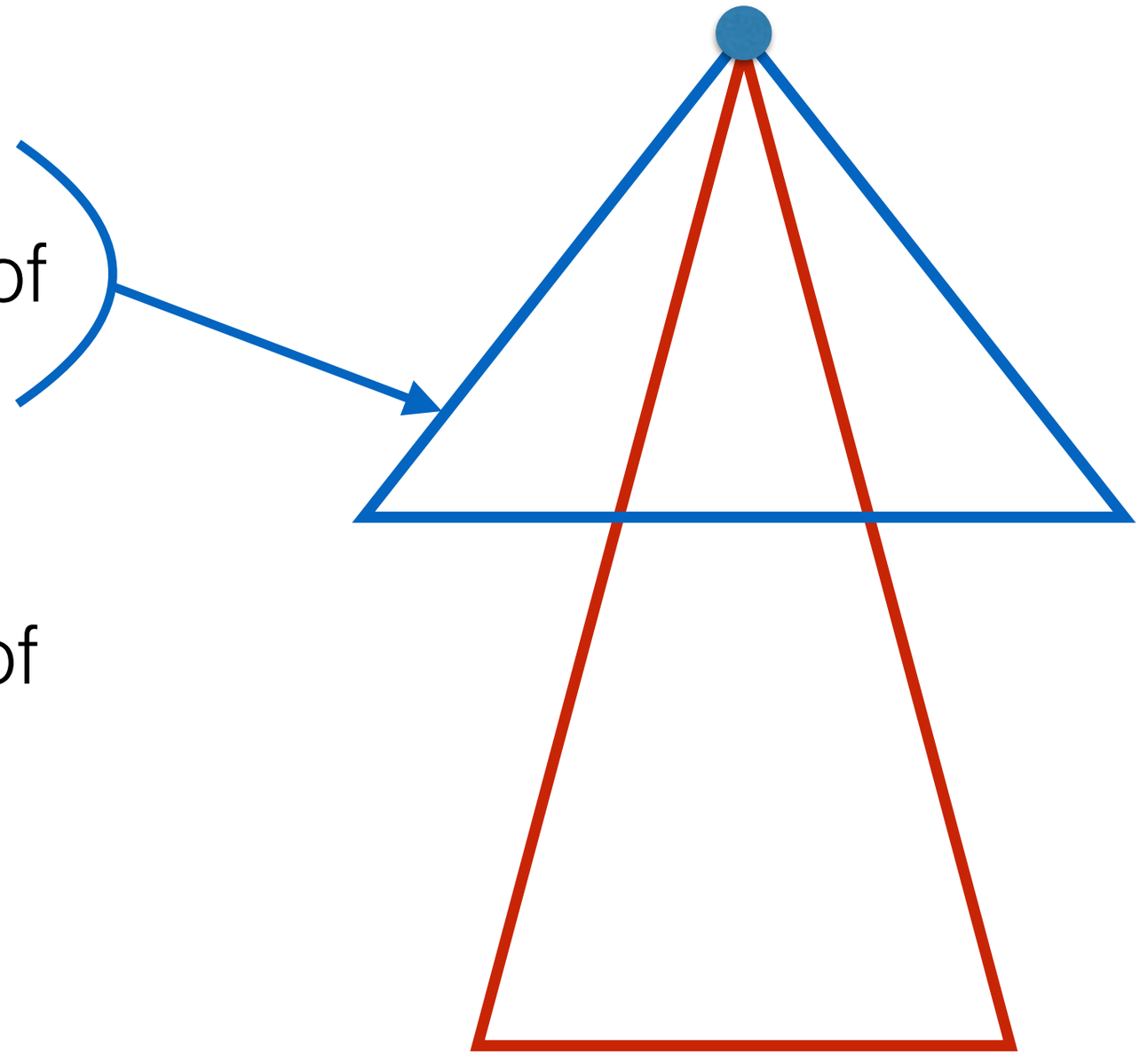
- small domains reduces the ***width*** of the search tree



Search tree shape

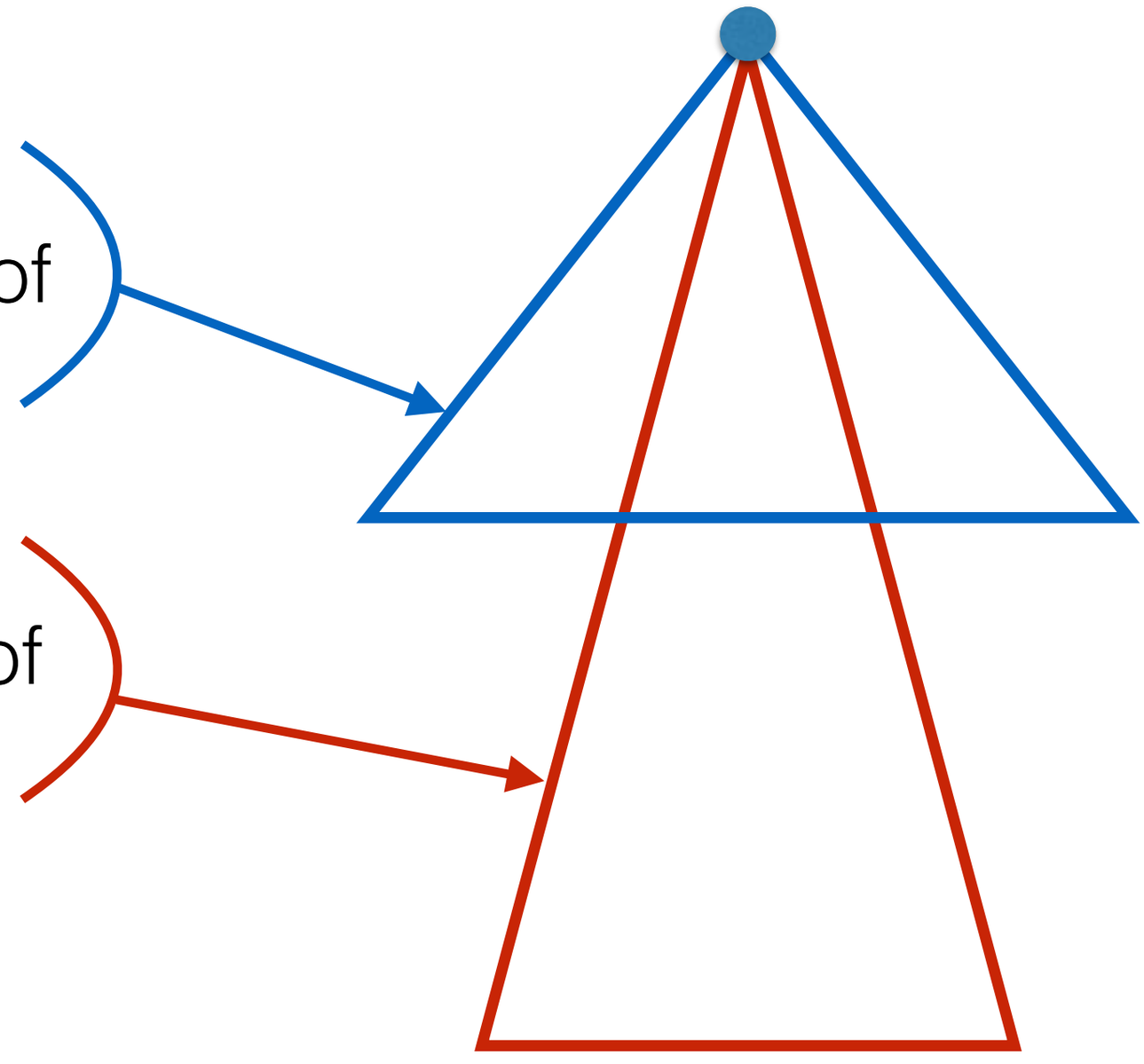
- difficult variables reduces the ***depth*** of the search tree

- small domains reduces the ***width*** of the search tree



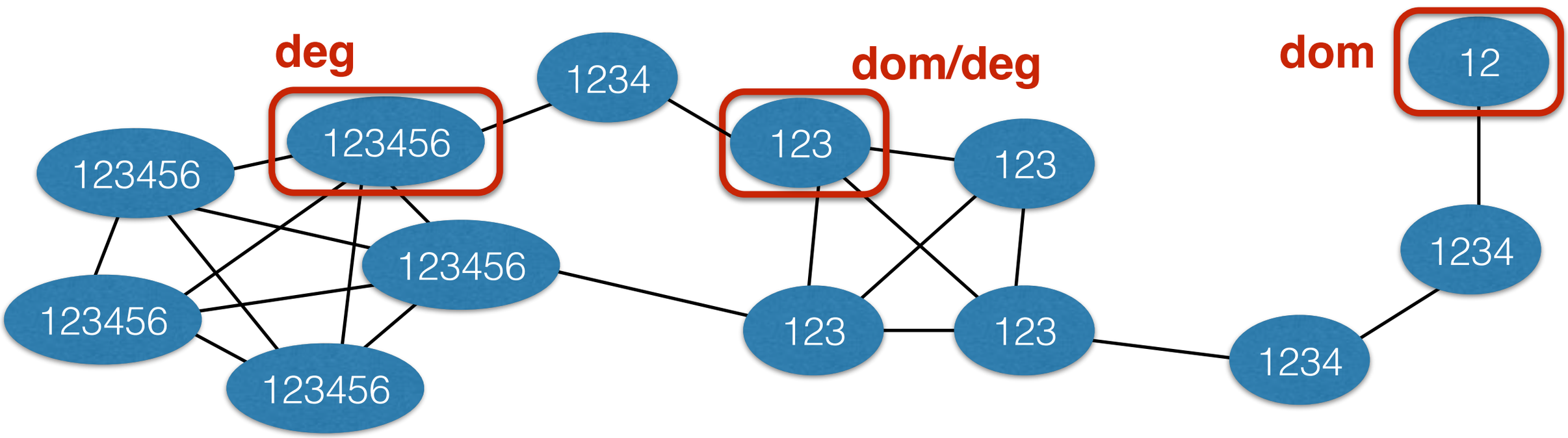
Search tree shape

- difficult variables reduces the ***depth*** of the search tree
- small domains reduces the ***width*** of the search tree



Combining dom and deg

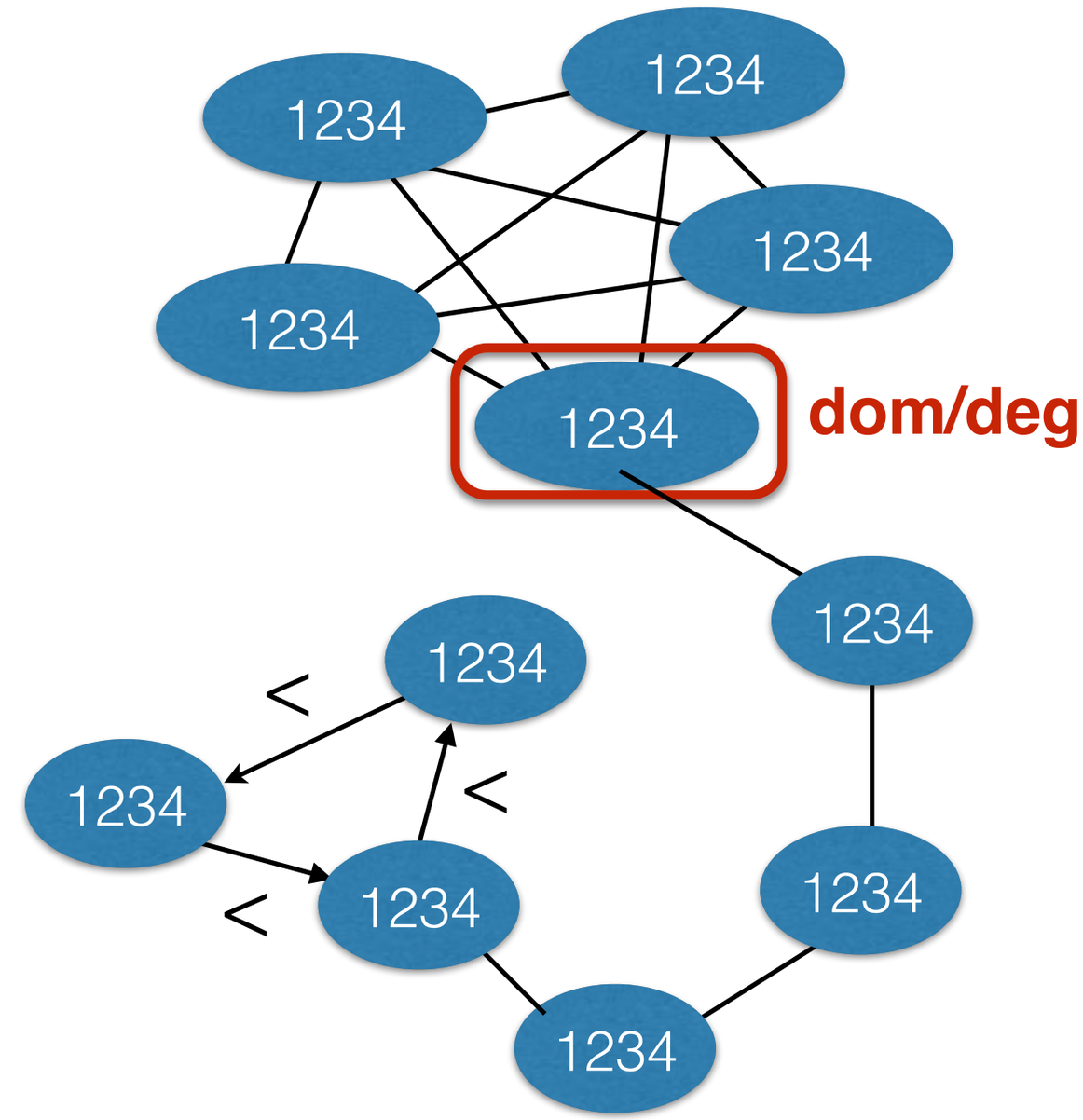
- **dom/deg**: select the variable X_i with smallest ratio: $\frac{|D(X_i)|}{|\Gamma(X_i)|}$



Reasoning with conflicts

- **dom/wdeg**:
  - Each constraint c_j is attached a weight w_j initialized to 1
  - At each fail caused by Revise(c_j), then w_j ++
  - select the variable X_i with smallest ratio

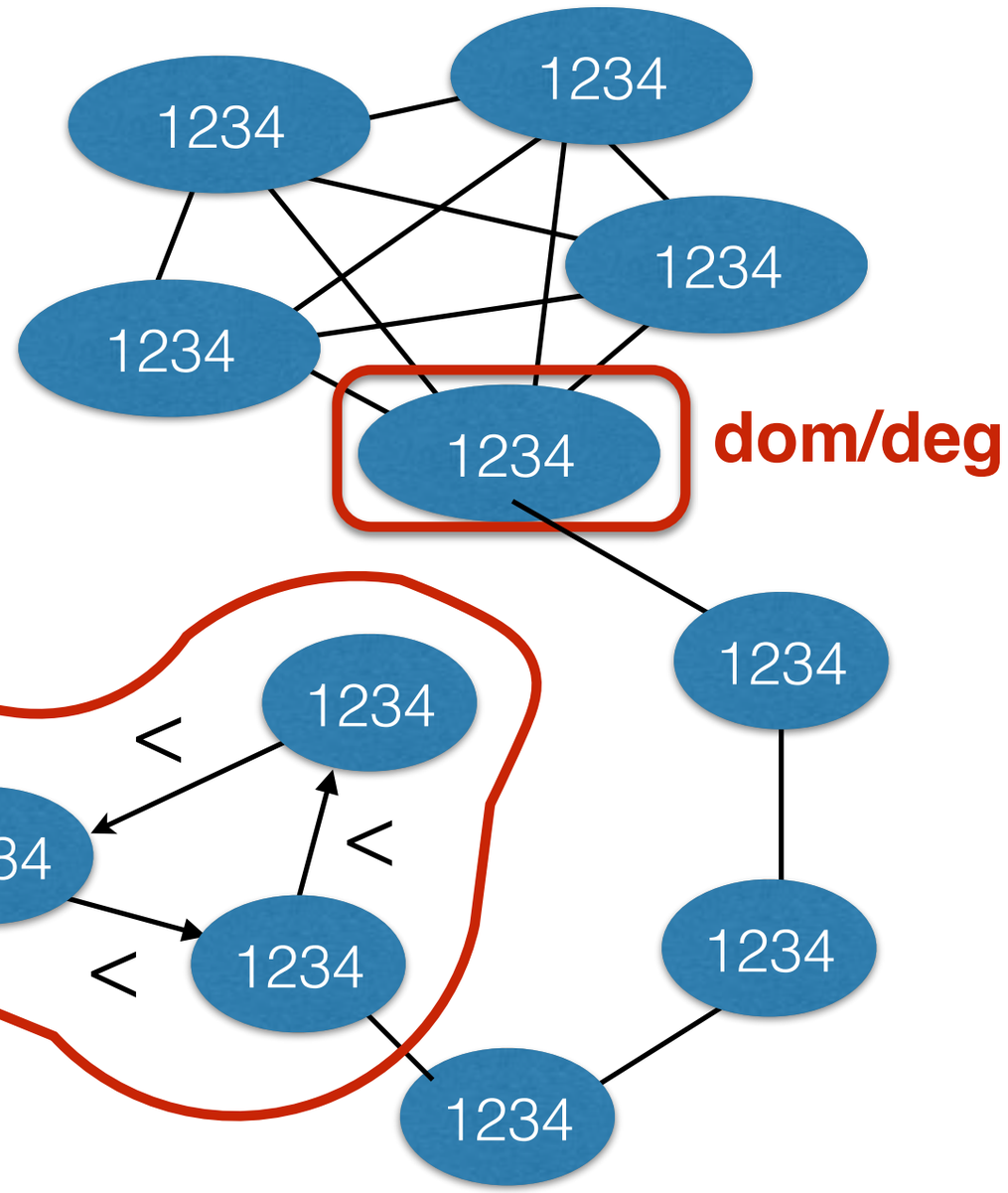
$$\frac{|D(X_i)|}{\sum_{c_j \in \Gamma(X_i)} w_j}$$



Reasoning with conflicts

- **dom/wdeg**:
  - Each constraint c_j is attached a weight w_j initialized to 1
  - At each fail caused by Revise(c_j), then w_j ++
  - select the variable X_i with smallest ratio

$$\frac{|D(X_i)|}{\sum_{c_j \in \Gamma(X_i)} w_j}$$



Value ordering heuristics

- We may not need to try all values of a variable to find a solution
  - ➜ start by the most promising

- Choosing a promising value is expensive (except if expert knowledge)

- On inconsistent problems value ordering is useless (Now, 95% of the time to solve hard problems is spent on inconsistent subtrees)

- ➜ Use only if ***expert knowledge available***

To take home

- Two essential components
  - Constraint propagation to reduce the size of the search tree
  - Variable ordering heuristics to pull failures as high as possible in the search tree

Domain-based
consistencies stronger than AC

Singleton Arc Consistency (SAC)

Definition 20 (Singleton arc consistency) *A network* $N = (X, D, C)$ *is* singleton arc consistent *iff for all* $X_i \in X$*, for all* $v_i \in D(X_i)$*, the network* $N + \{X_i = v_i\}$ *is not arc inconsistent (i.e.,* $AC(N + \{X_i = v_i\}) \neq \emptyset$*).*