Cours Master 2

Contraintes

Organisation

- Intervenants :
  - Christian Bessiere (bessiere@lirmm.fr)
  - Clément Carbonnel
  - Michael Sioutis (responsable)

Contenu

- Bases (12h) par Bessiere
- Classes polynomiales (6h) par Carbonnel
- Raisonnement temporel et spatial (6h) par Sioutis

+ TDs par Carbonnel et Sioutis

Success stories

★ **Scientific fields**

- chemistry, biology, robotics, telecoms, software verification, production line, games

★ **Categories of problems**

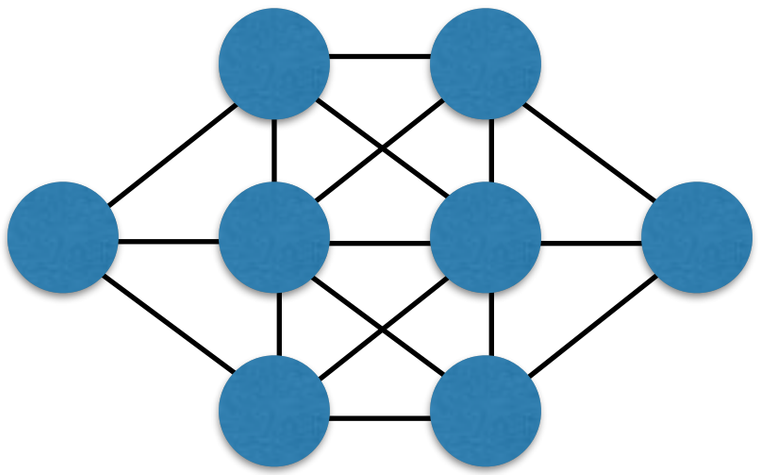
- resource allocation, time tabling, scheduling, configuration, planning

★ **Examples**

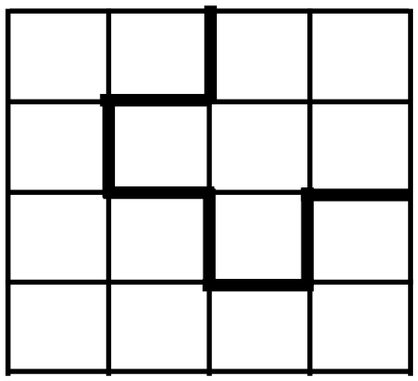
- sport league scheduling, nurse/hostess rostering, gates to planes/trains, excavators in mines, car assembly line, energy (data centers, power plants), port management …

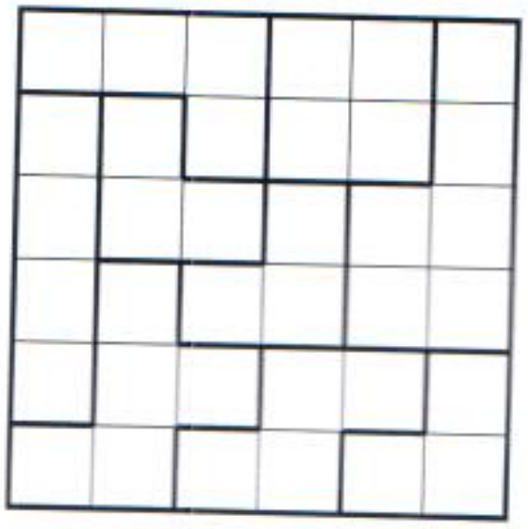
Games…

- Mettre les nombres de 1 à 8 sur les 8 sommets du graphe de sorte que deux nombres consécutifs ne sont pas voisins



- Mettre les nombres de 1 à *n* sur chaque ligne et chaque colonne de sorte que chaque région ait la même somme

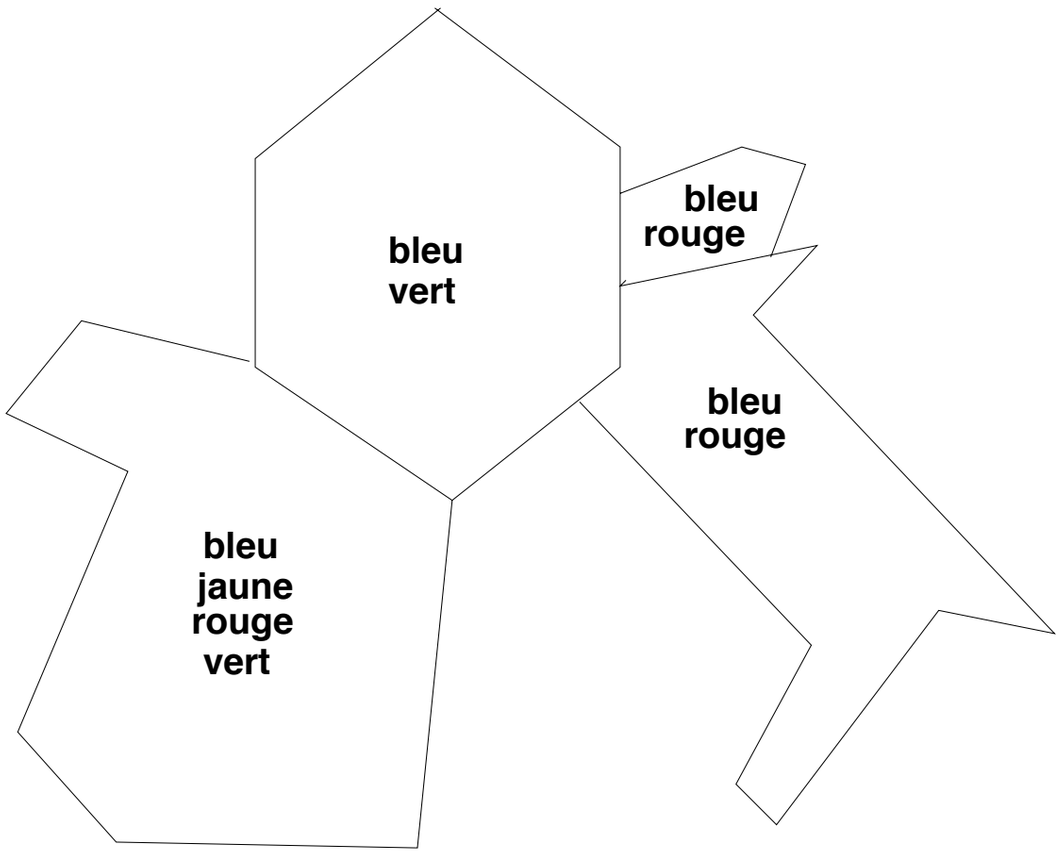




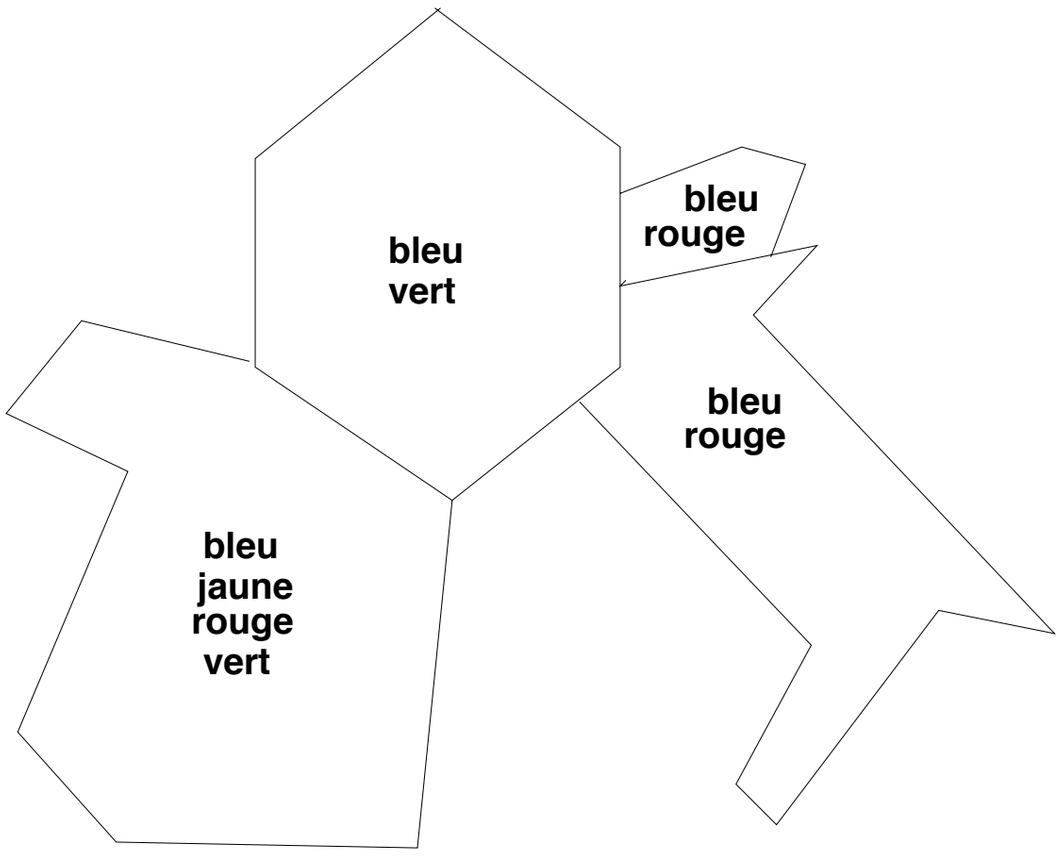
Running example

Running example

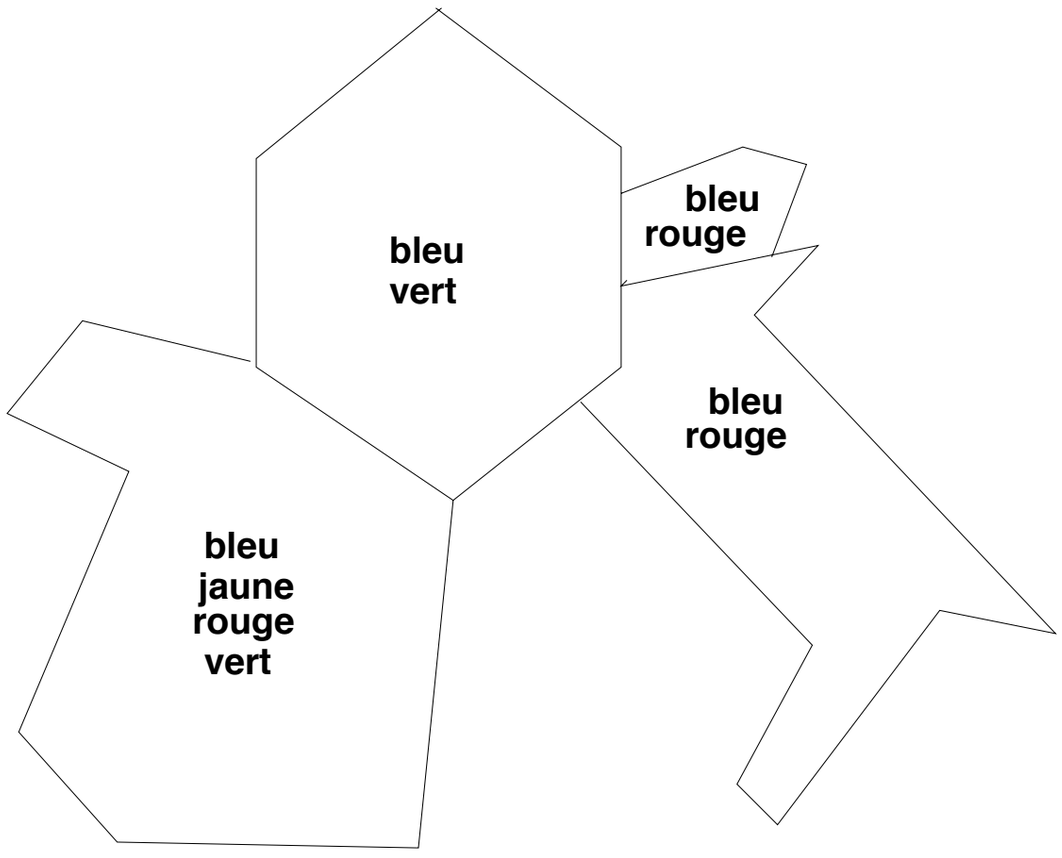
Graph coloring



Constraint network



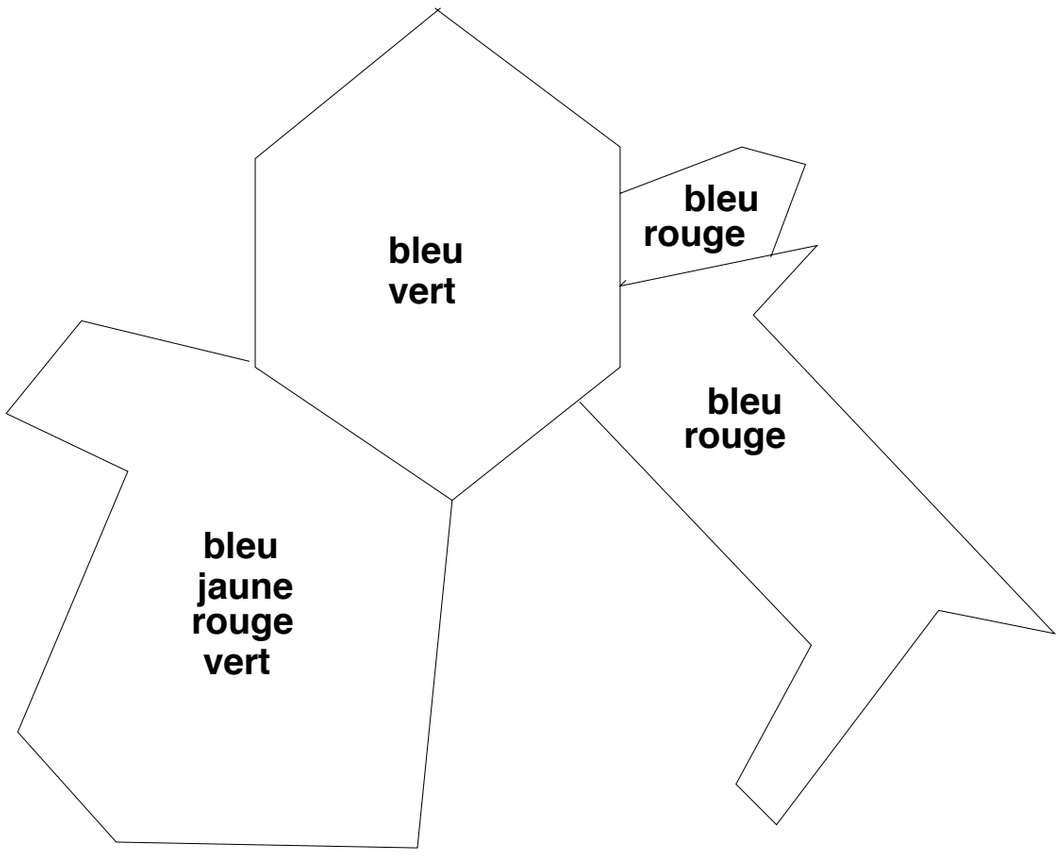
Constraint network



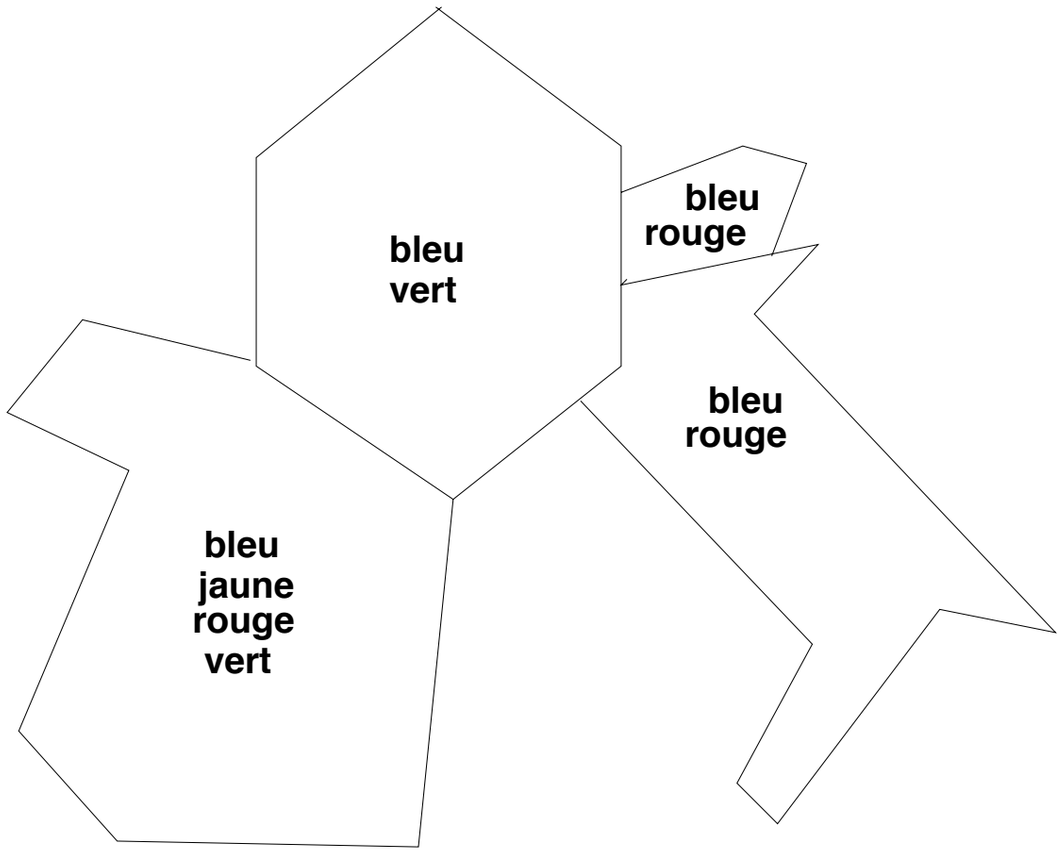
	components	on the example
for each variable	variables	France, Spain, etc
	domain of values	bleu, jaune, etc
on subsets of variables	constraints	not the same color

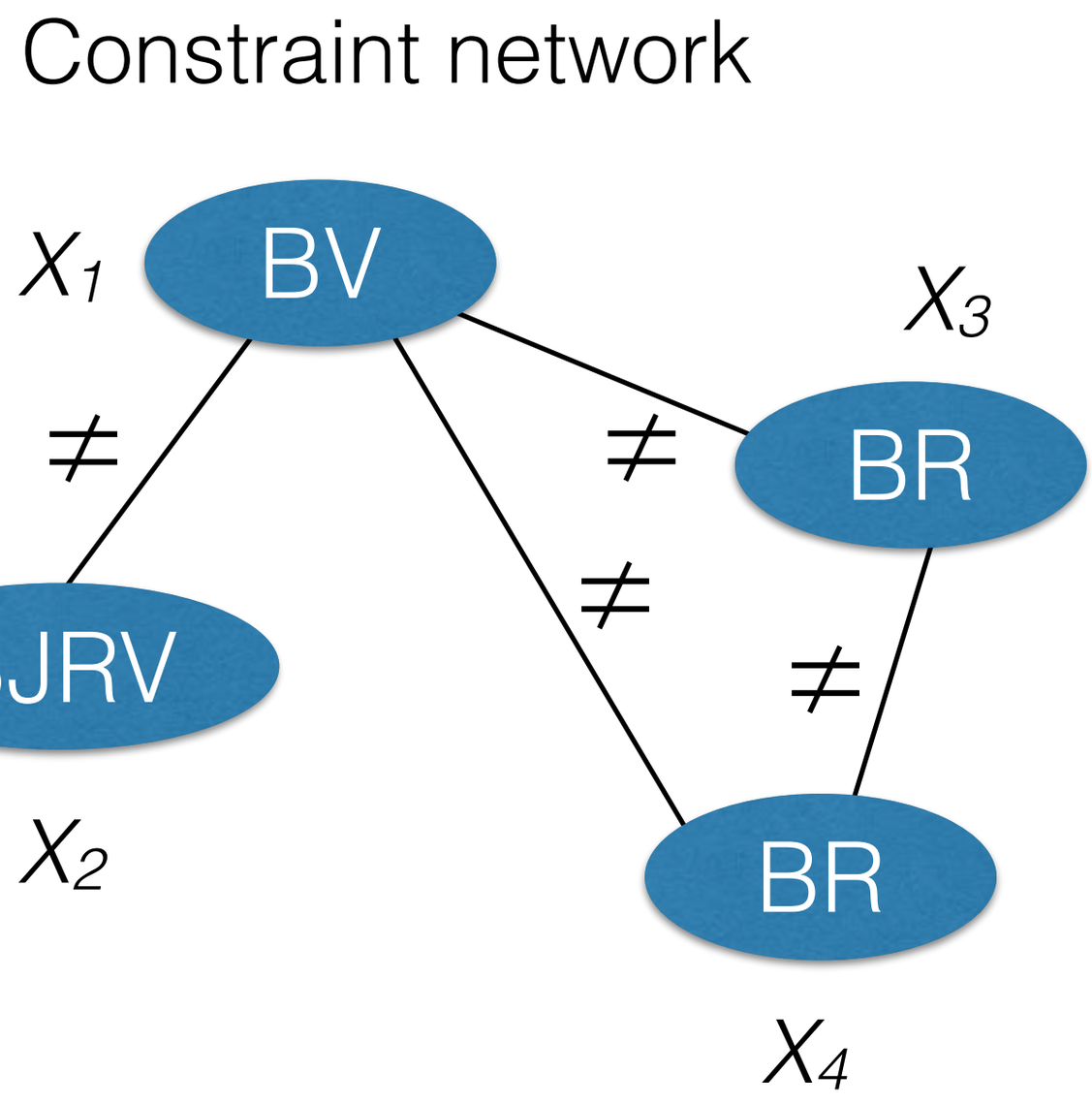
Constraint network

Constraint network

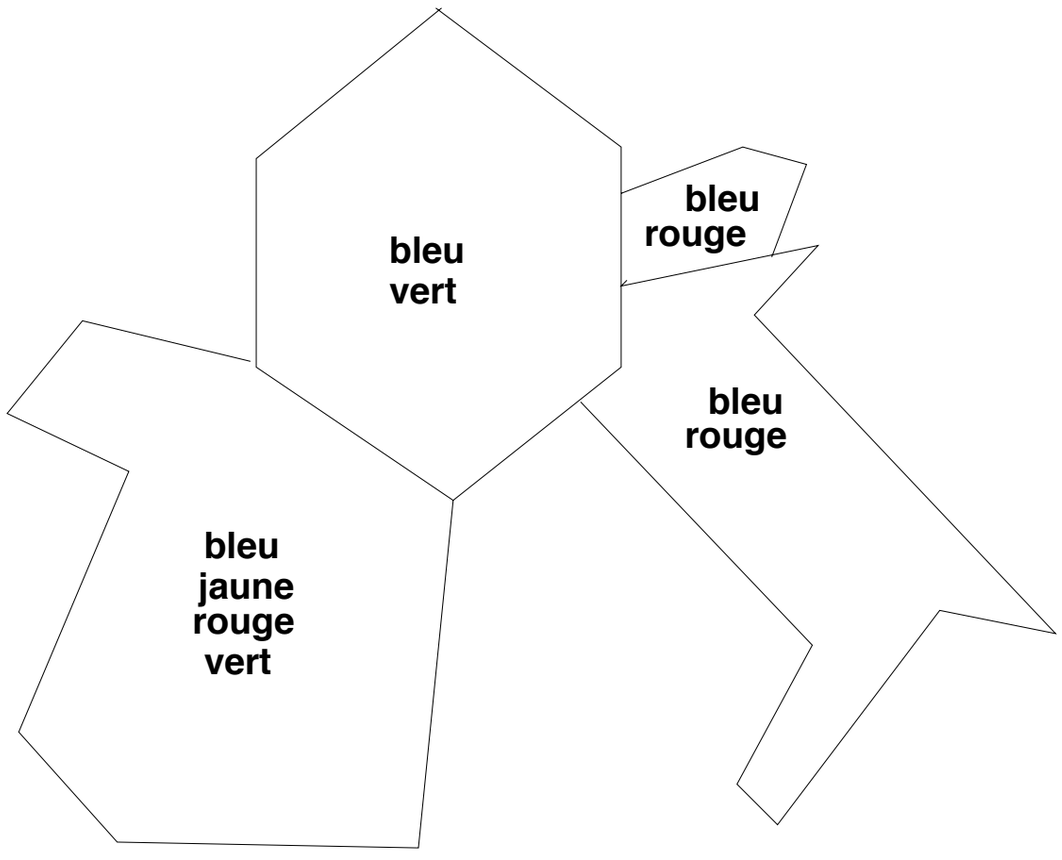


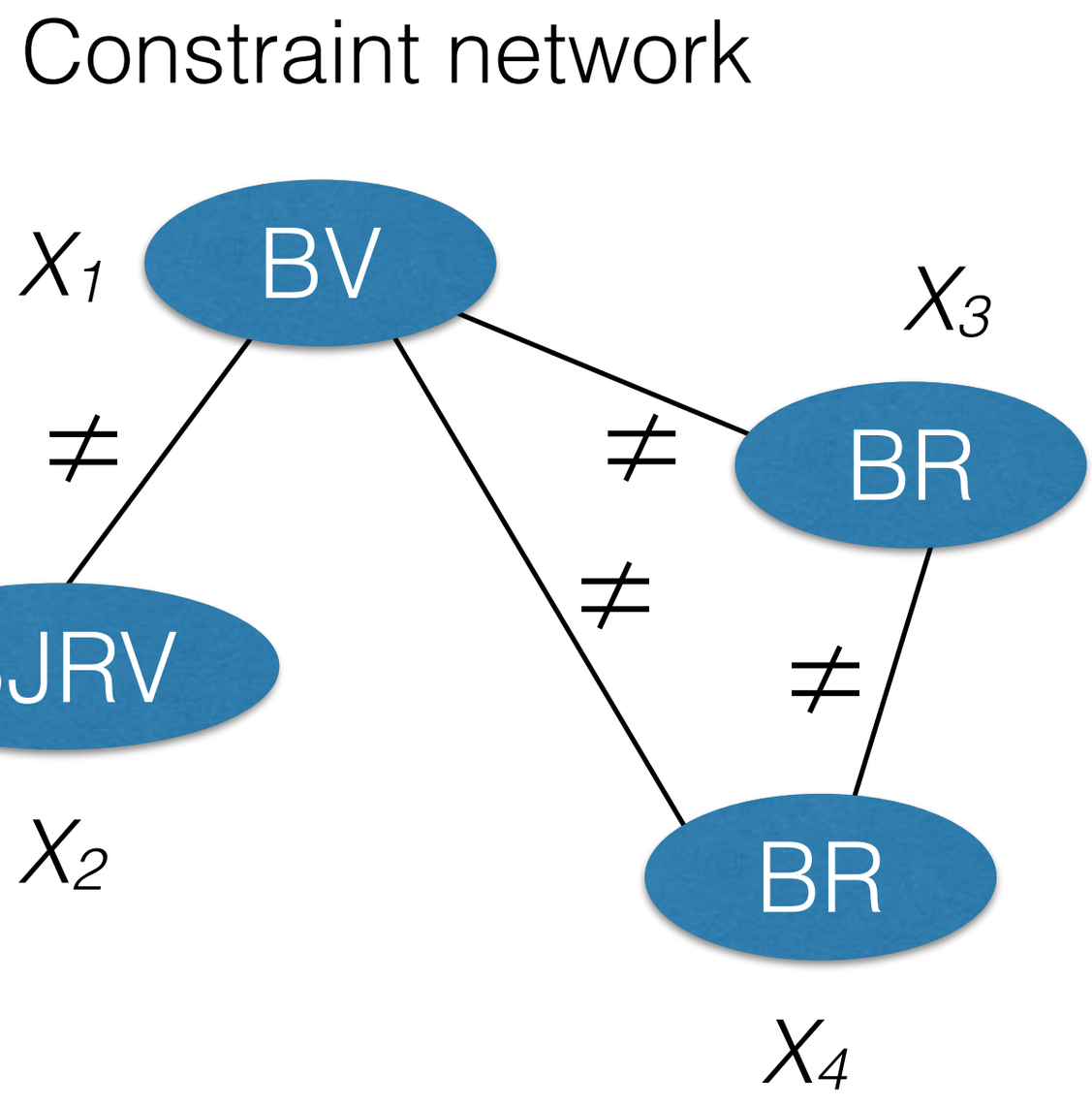
Constraint network





Constraint network

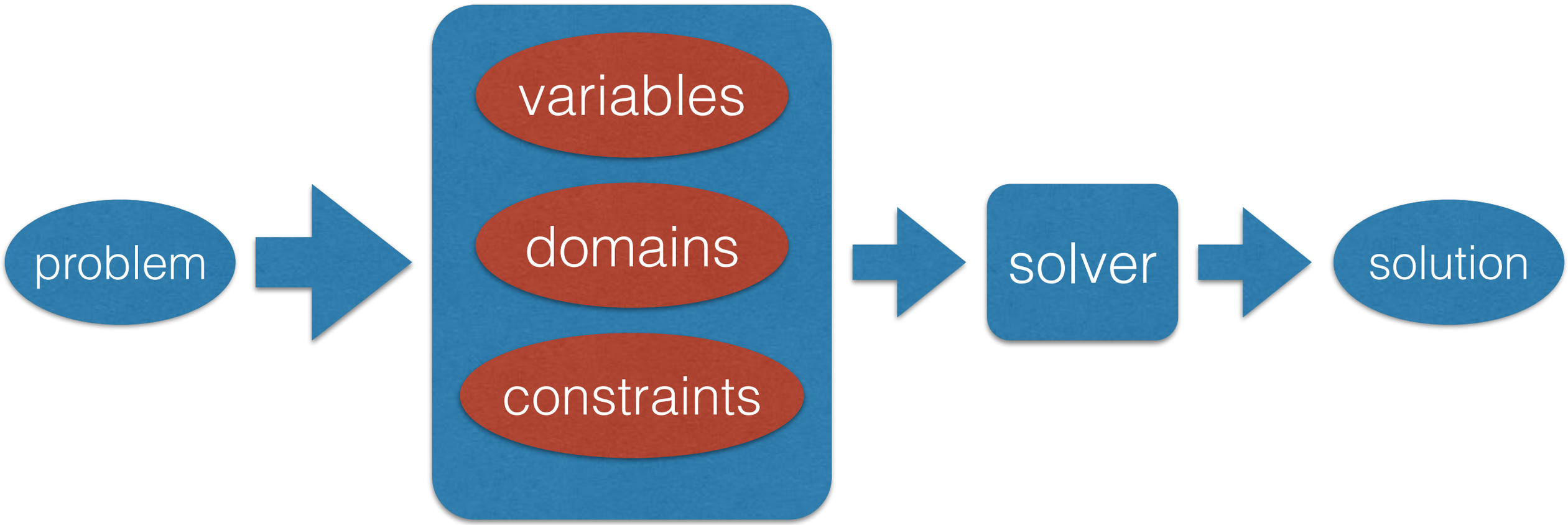




Solution : (X_1,V); (X_2,B); (X_3,B); (X_4,R)

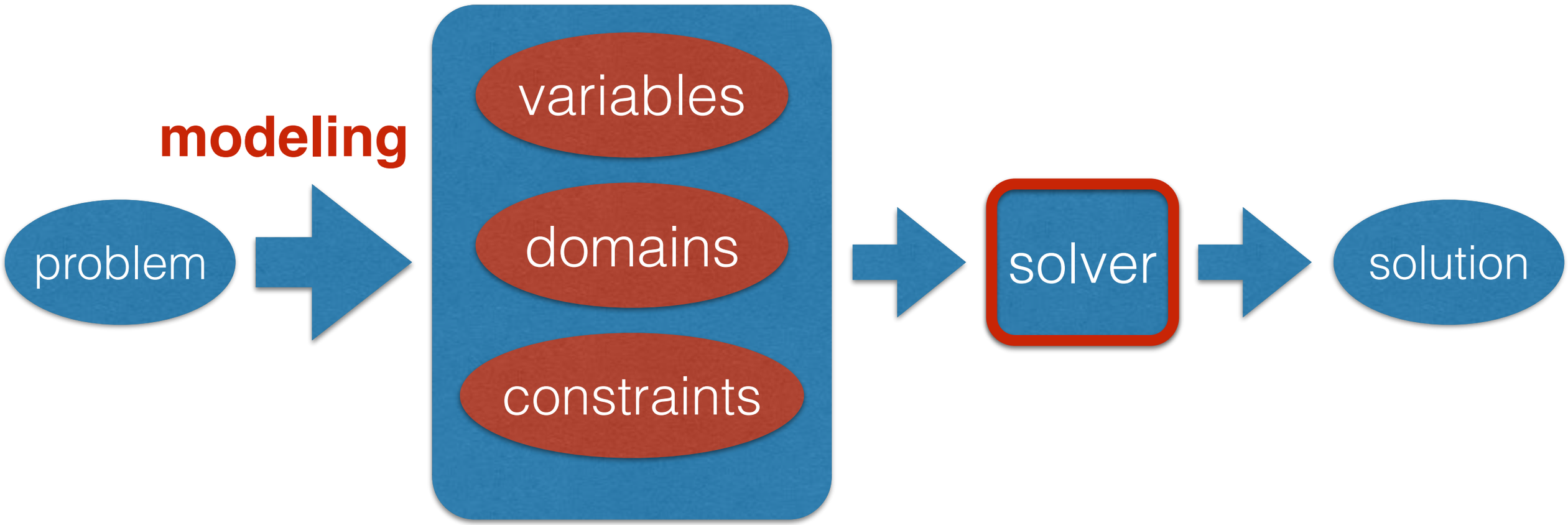
France = vert, Espagne = bleu, Suisse = bleu, Italie = rouge

Formalisme de résolution de problèmes



constraint network

Formalisme de résolution de problèmes



constraint network

Basic definitions

Constraint network

Definition 1 (Constraint network) *A* constraint network *(or* network*) is composed of :*

- *a set of variables* $X = \{X_1, \ldots, X_n\}$
- *a domain on* X*, that is, a set* $D = \{D(X_1), \ldots, D(X_n)\}$*, where* $D(X_i) \subset \mathbb{Z}$ *is the* finite *set of values that variable* X_i *can take (its* domain*), and*
- *a set of constraints* $C = \{c_1, \ldots, c_e\}$

Constraint

Definition 2 (Constraint) *A constraint c is a Boolean function involving a sequence of variables $X(c) = (X_{i_1}, \ldots, X_{i_q})$ called its* scheme. *The function is defined on $\mathbb{Z}^q$. A combination of values (or* tuple*) $\tau \in \mathbb{Z}^q$* satisfies *c if $c(\tau) = 1$ (also noted $\tau \in c$). If $c(\tau) = 0$ (or $\tau \notin c$), τ* violates *c.*

Constraint

Definition 2 (Constraint) *A constraint c is a Boolean function involving a sequence of variables $X(c) = (X_{i_1}, \ldots, X_{i_q})$ called its* scheme. *The function is defined on $\mathbb{Z}^q$. A combination of values (or* tuple*) $\tau \in \mathbb{Z}^q$* satisfies *c if $c(\tau) = 1$ (also noted $\tau \in c$). If $c(\tau) = 0$ (or $\tau \notin c$), τ* violates *c.*

- Notations:
  - c on $X(c)=(X_1,\ldots,X_k)$
  - $c(X_1,\ldots,X_k)$
  - $c(X_i,X_j) \longrightarrow c_{ij}$

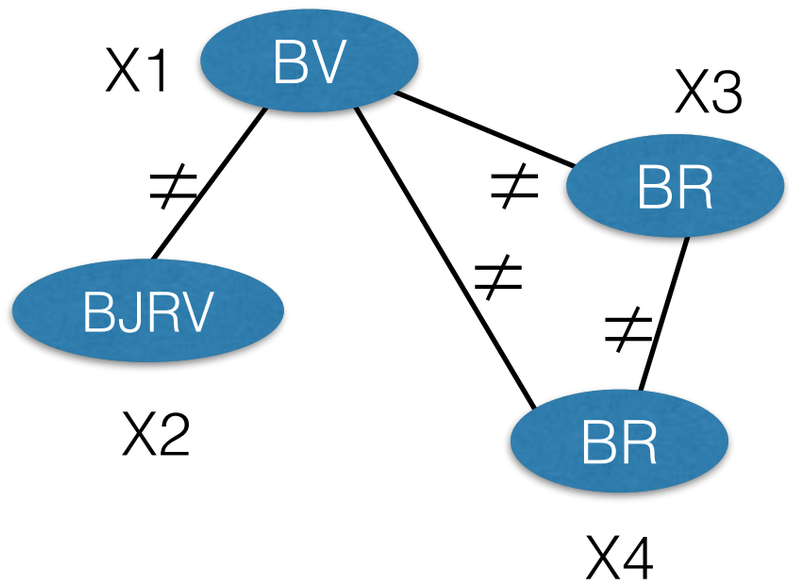
Constraint

Definition 2 (Constraint) *A constraint c is a Boolean function involving a sequence of variables $X(c) = (X_{i_1}, \ldots, X_{i_q})$ called its* scheme. *The function is defined on $\mathbb{Z}^q$. A combination of values (or* tuple*) $\tau \in \mathbb{Z}^q$* satisfies *c if $c(\tau) = 1$ (also noted $\tau \in c$). If $c(\tau) = 0$ (or $\tau \notin c$), τ* violates *c.*

- Notations:
  - c on $X(c)=(X_1,\ldots,X_k)$
  - $c(X_1,\ldots,X_k)$
  - $c(X_i,X_j)$ —> c_{ij}
- Examples:
  - *alldifferent*$(X_1,\ldots,X_k)$
  - $X_1 + X_2 = 5$
  - $c(X_i,X_j) = \{(1,2),(2,1),(3,3),(4,4)\}$

Instantiation

Definition 3 (Instantiation) *Given a network* $N = (X, D, C)$,

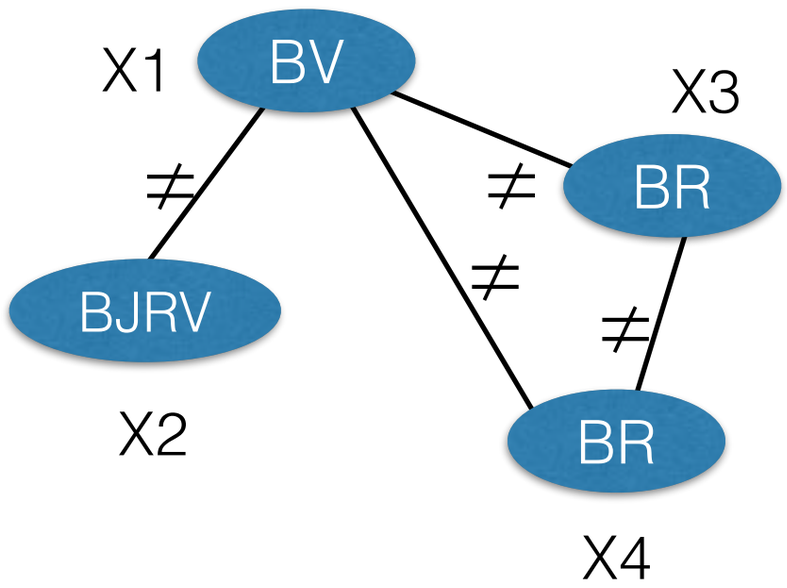


Instantiation

Definition 3 (Instantiation) *Given a network* $N = (X, D, C)$,

— *an* instantiation I *on a set* $Y = \{X_1, \ldots, X_k\} \subseteq X$ *of variables is an assignment of values* $v_1, \ldots v_k$ *to* $X_1, \ldots, X_k$

(X_1,R), (X_2,J), (X_3,B)



Instantiation

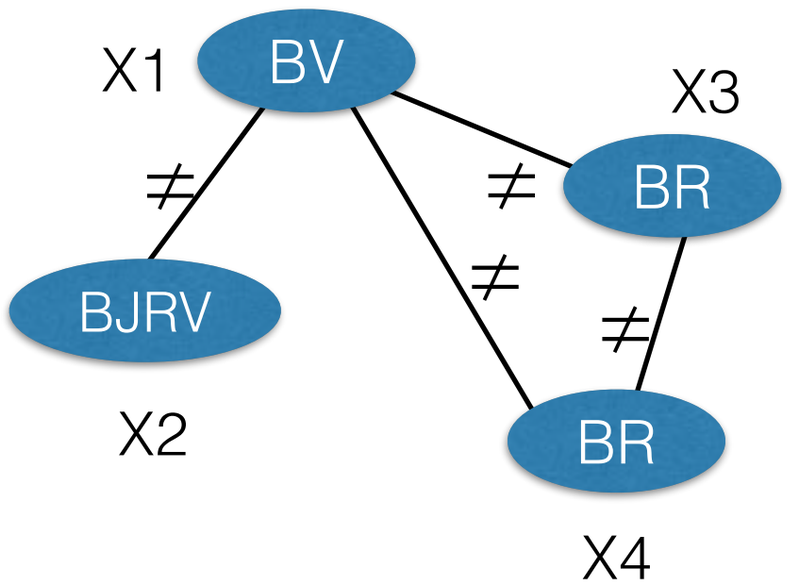
Definition 3 (Instantiation) *Given a network* $N = (X, D, C)$,

— *an* instantiation I *on a set* $Y = \{X_1, \ldots, X_k\} \subseteq X$ *of variables is an assignment of values* $v_1, \ldots v_k$ *to* $X_1, \ldots, X_k$

— *an instantiation* I *on* Y *is* valid *iff* $I \subseteq D^Y$ $(I[X_i] \in D(X_i), \forall X_i \in Y)$

(X_1,R), (X_2,J), (X_3,B)

(X_1,B), (X_2,J), (X_3,B)



Instantiation

Definition 3 (Instantiation) *Given a network* $N = (X, D, C)$,

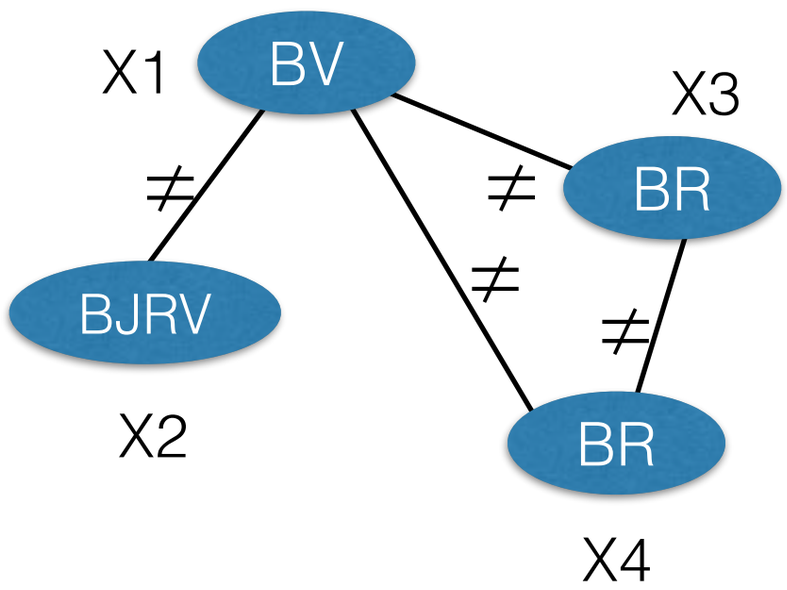
— *an* instantiation I *on a set* $Y = \{X_1, \ldots, X_k\} \subseteq X$ *of variables is an assignment of values* $v_1, \ldots v_k$ *to* $X_1, \ldots, X_k$

— *an instantiation* I *on* Y *is* valid *iff* $I \subseteq D^Y$ $(I[X_i] \in D(X_i), \forall X_i \in Y)$

— *an instantiation* I *on* Y violates c *iff* $X(c) \subseteq Y$ *and* $I[X(c)] \notin c$

(X_1,R), (X_2,J), (X_3,B)

(X_1,B), (X_2,J), (X_3,B)



Instantiation

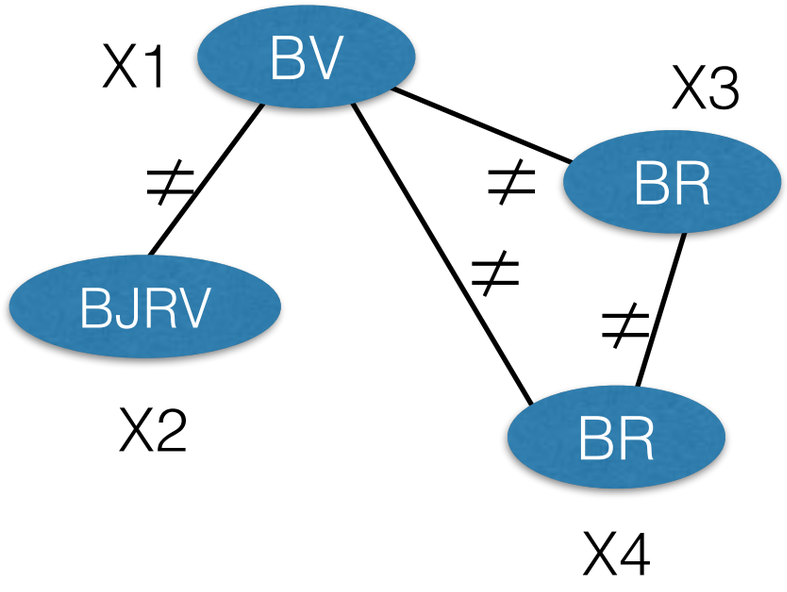
Definition 3 (Instantiation) *Given a network* $N = (X, D, C)$,

- *an* instantiation I *on a set* $Y = \{X_1, \ldots, X_k\} \subseteq X$ *of variables is an assignment of values* $v_1, \ldots v_k$ *to* $X_1, \ldots, X_k$
- *an instantiation* I *on* Y *is* valid *iff* $I \subseteq D^Y$ $(I[X_i] \in D(X_i), \forall X_i \in Y)$
- *an instantiation* I *on* Y violates c *iff* $X(c) \subseteq Y$ *and* $I[X(c)] \notin c$
- *an instantiation* I *on* Y *is* locally consistent *iff it is valid and does not violate any constraint*

(X_1,R), (X_2,J), (X_3,B)

(X_1,B), (X_2,J), (X_3,B)

(X_1,B), (X_2,J), (X_3,R)



Instantiation

Definition 3 (Instantiation) *Given a network* $N = (X, D, C)$,

— *an* instantiation I *on a set* $Y = \{X_1, \ldots, X_k\} \subseteq X$ *of variables is an assignment of values* $v_1, \ldots v_k$ *to* $X_1, \ldots, X_k$

— *an instantiation* I *on* Y *is* valid *iff* $I \subseteq D^Y$ $(I[X_i] \in D(X_i), \forall X_i \in Y)$

— *an instantiation* I *on* Y violates c *iff* $X(c) \subseteq Y$ *and* $I[X(c)] \notin c$

— *an instantiation* I *on* Y *is* locally consistent *iff it is valid and does not violate any constraint*

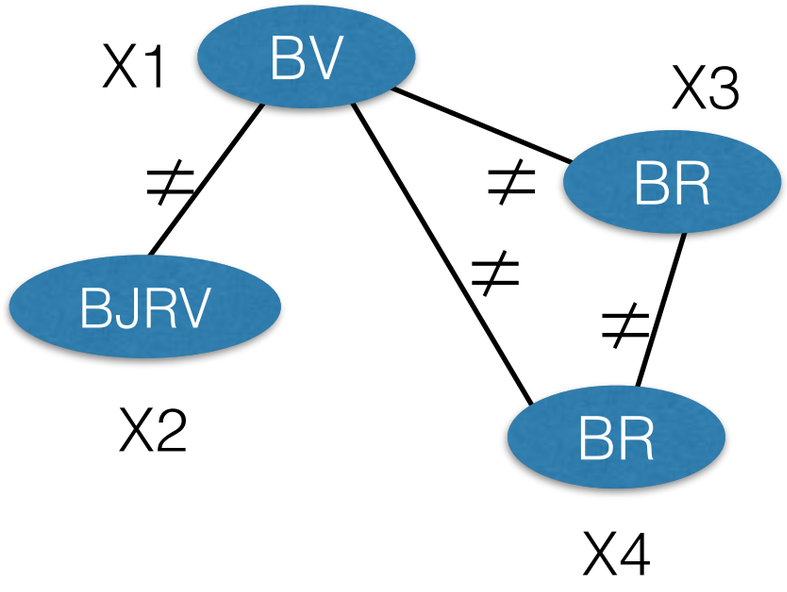
— *a* solution *is an instantiation* I *on* X *which is locally consistent.*

(X_1,*R*), (X_2,*J*), (X_3,B)

(X_1,*B*), (X_2,*J*), (X_3,B)

(X_1,*B*), (X_2,*J*), (X_3,R)

(X_1,*V*), (X_2,*J*), (X_3,B), (X_4,*R*)



Constraint Satisfaction Problem (CSP)

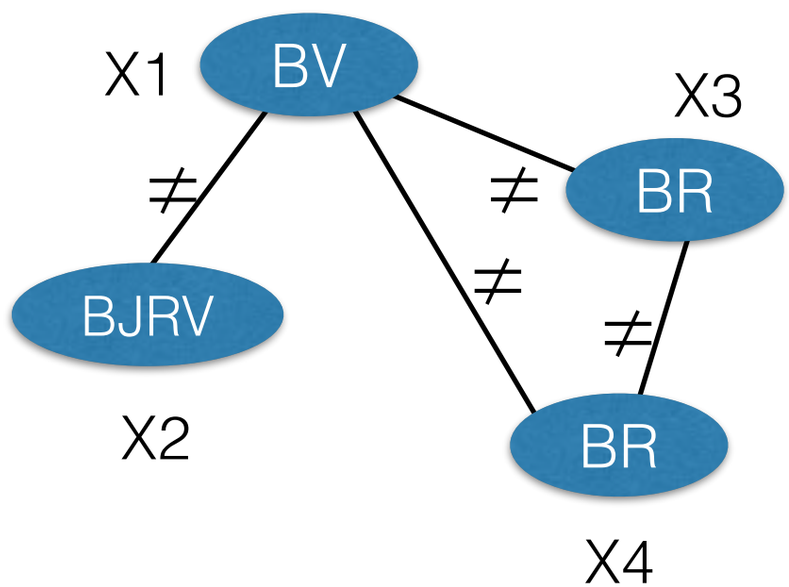
- Instance: *N = (X,D,C)*
- Question: Does there exist a solution to *N?*
- NP-complete

  ➔ search algorithm

Backtracking (BT)

function BT(N : problem ; 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**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end

A call to BT(*N*,∅) decides CSP

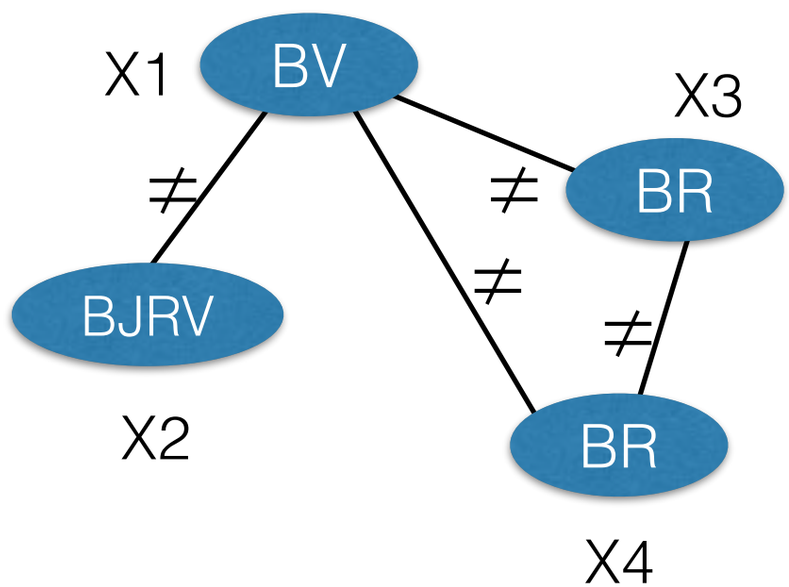


Backtracking (BT)

function BT(N : problem ; 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**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end

termination

A call to BT(*N*,∅) decides CSP



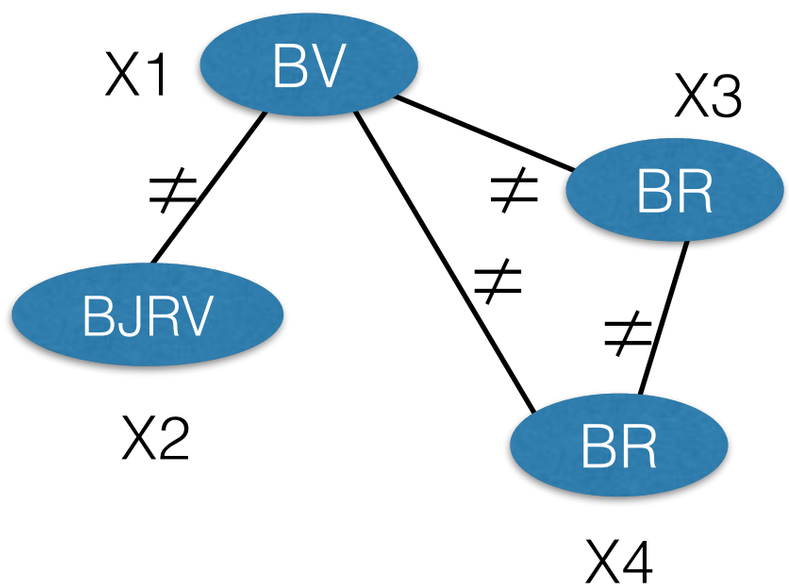
Backtracking (BT)

function BT(N : problem ; 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**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end

termination

branching

A call to BT(*N*,∅) decides CSP



Backtracking (BT)

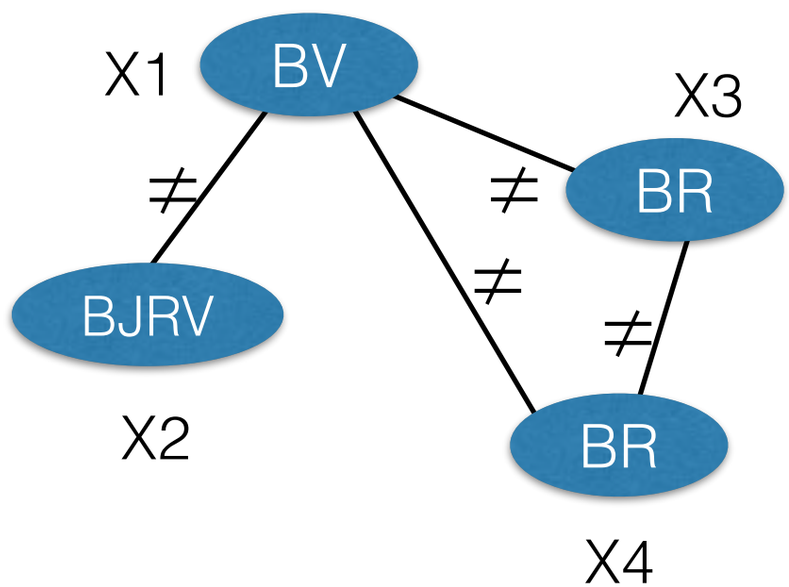
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end

termination

branching

recursive call

A call to BT(*N*,∅) decides CSP



BT on running example

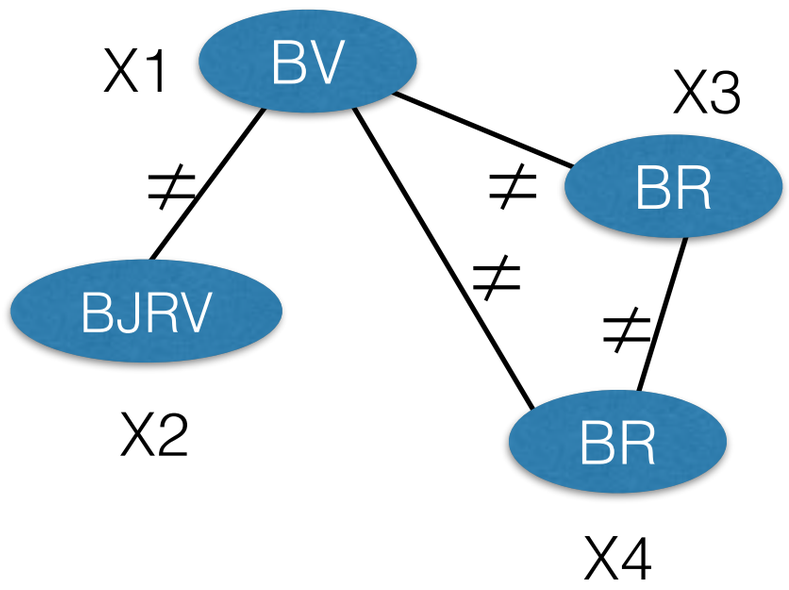
X1

X2

X3

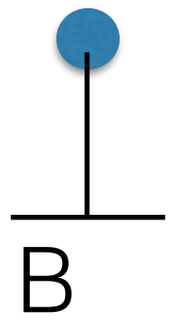
X4

```
function BT(N : problem ; 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
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```



BT on running example

X1

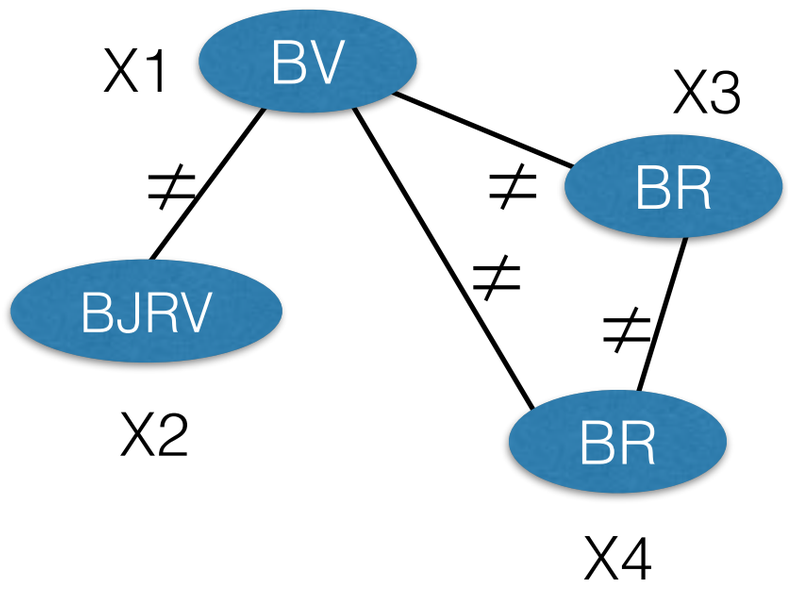



```
function BT(N : problem ; I : instantiation) :Boolean
begin
    if |I| = n then return true
    choose a variable X_i not in I
    for each v_i ∈ D(X_i) do
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```

X2

X3

X4



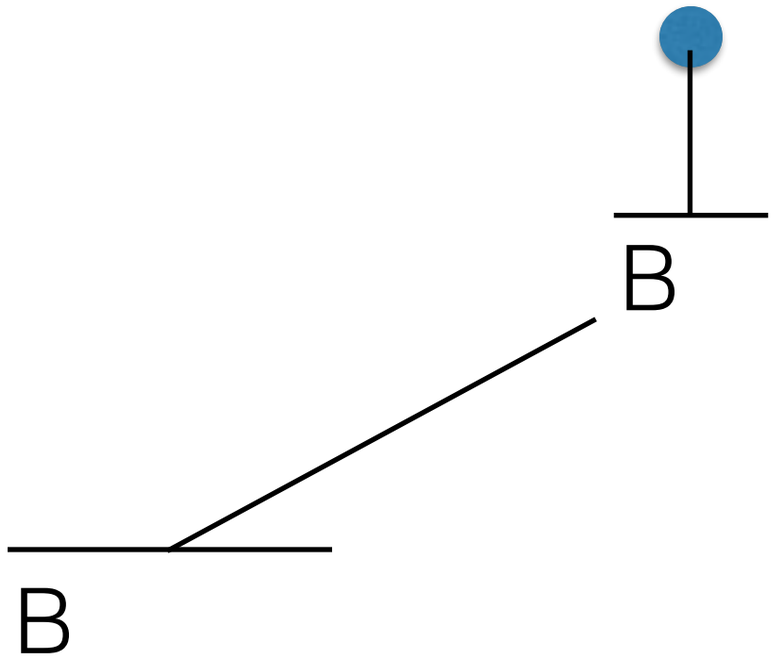
BT on running example

X1

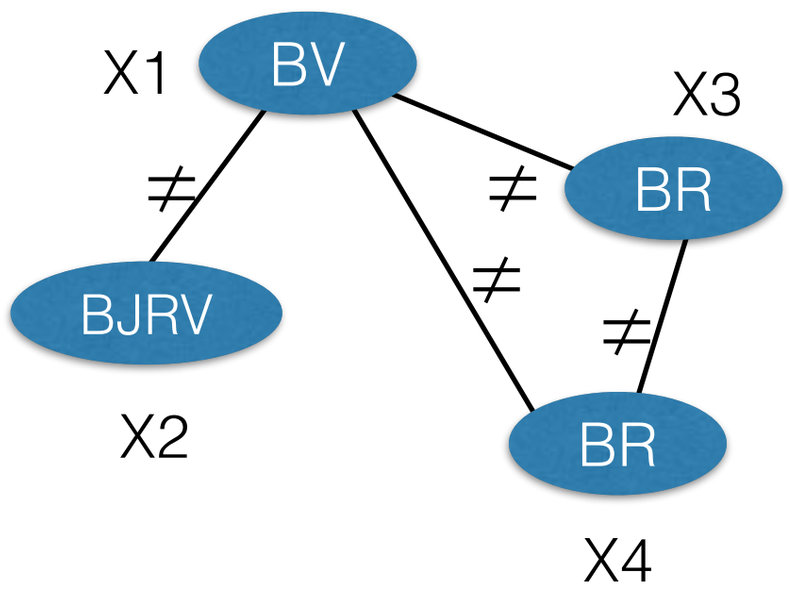
X2

X3

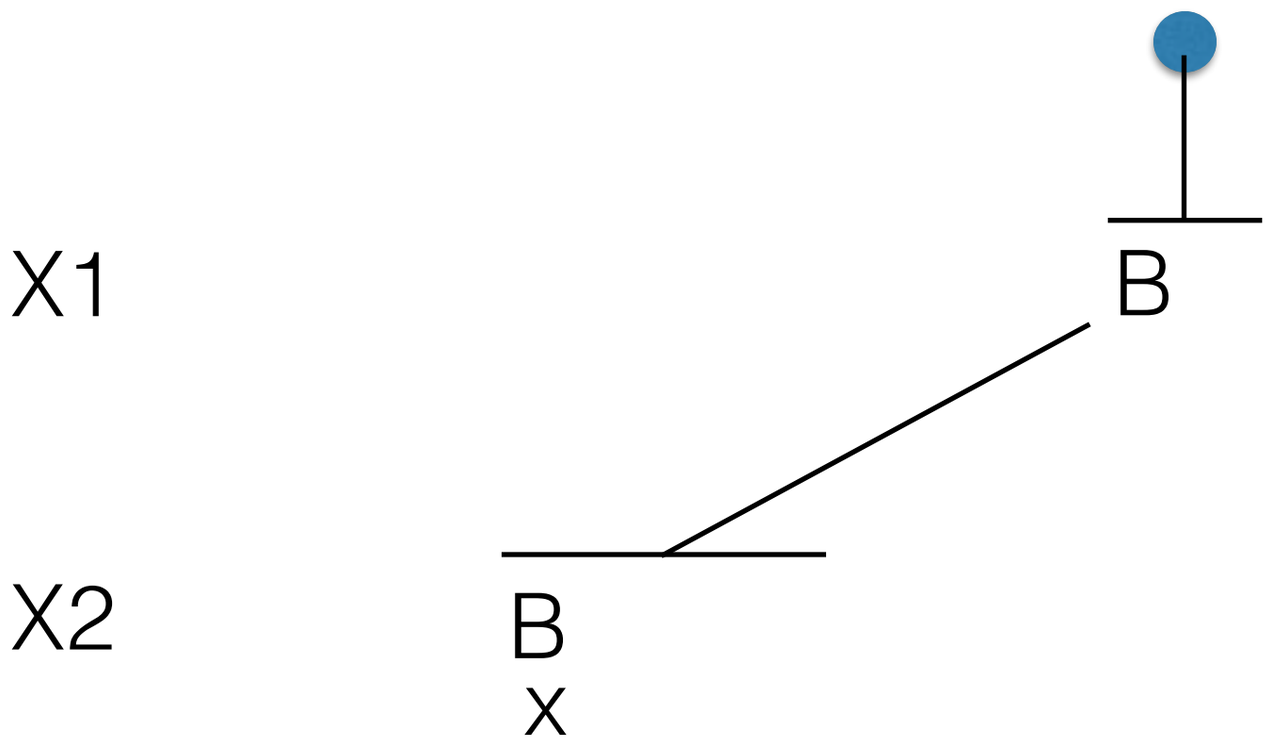
X4



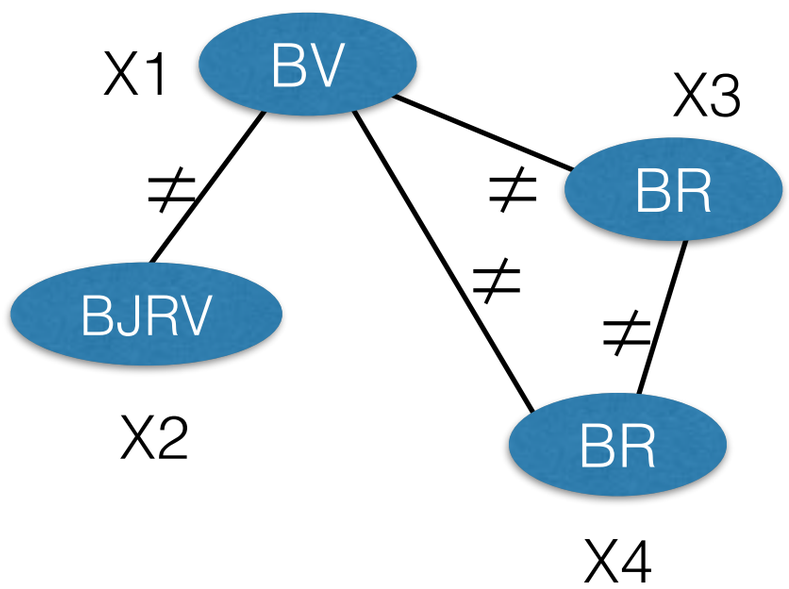
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



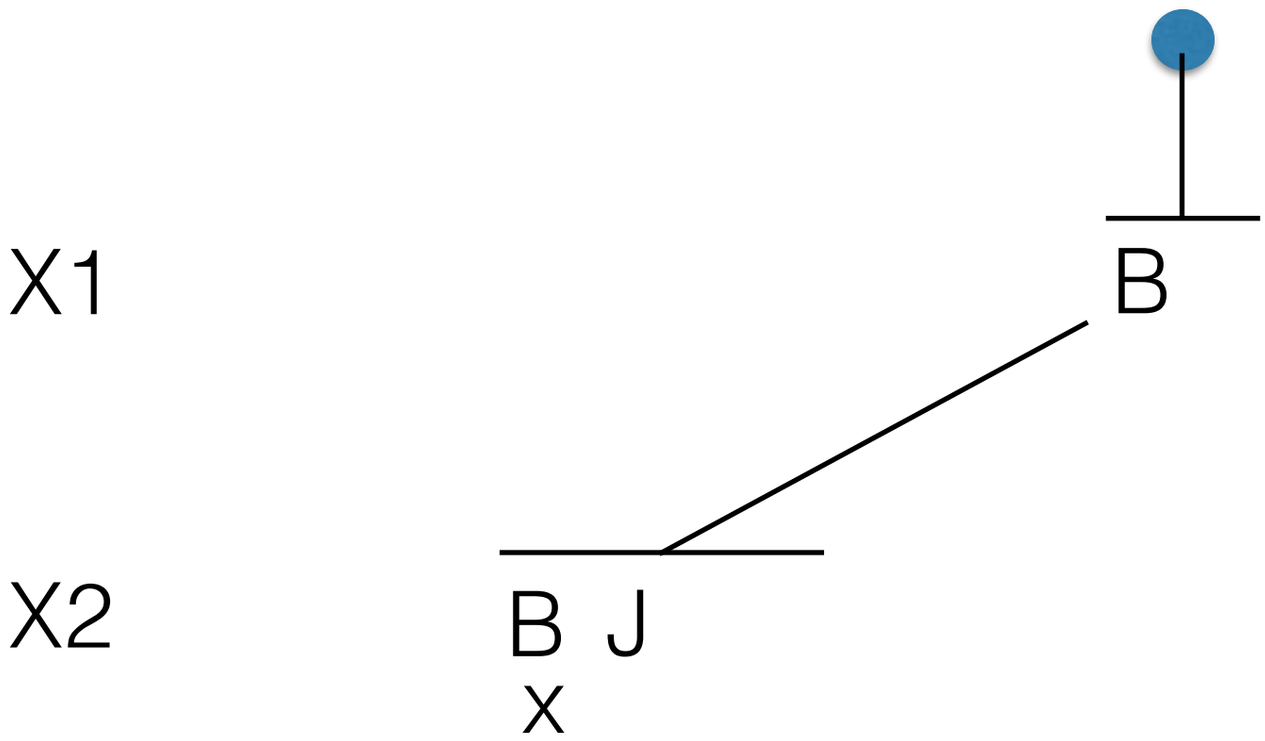
BT on running example



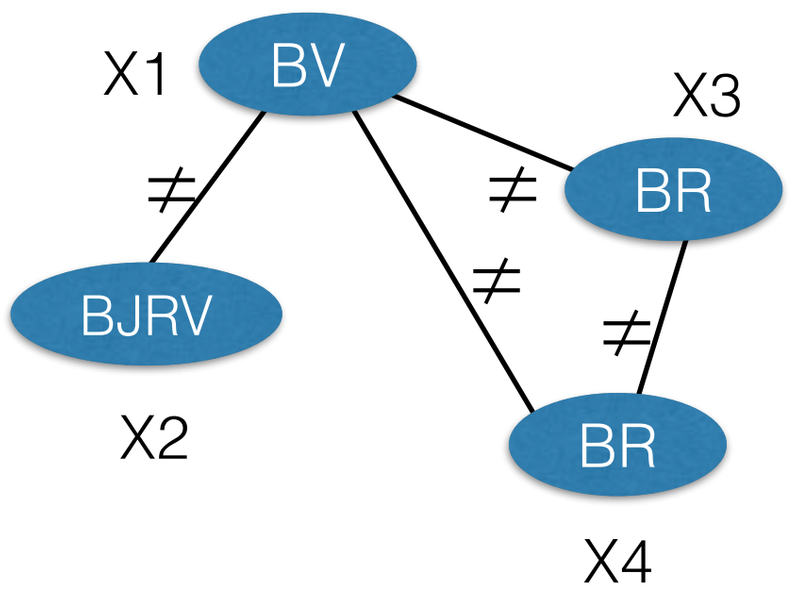
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



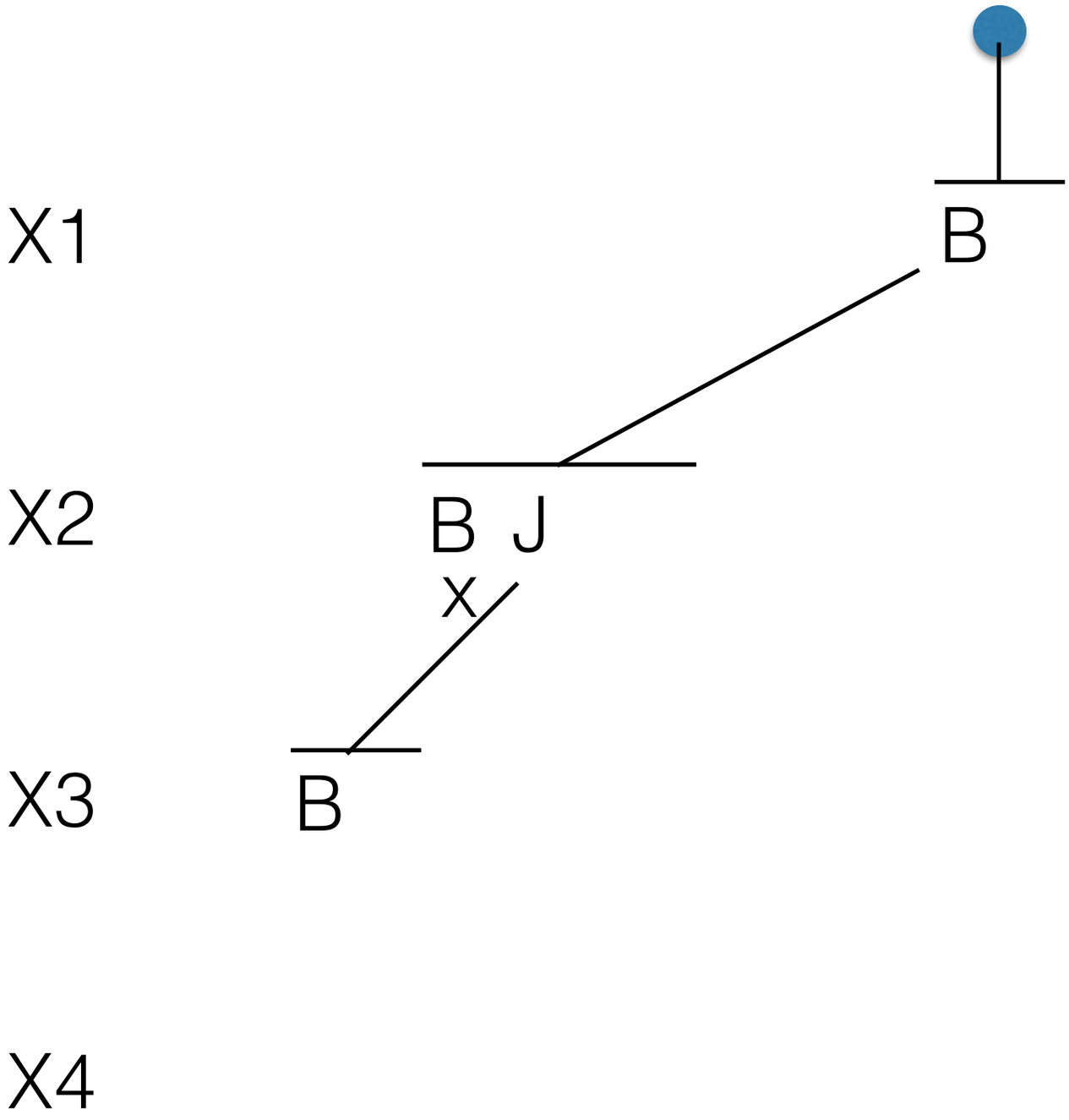
BT on running example



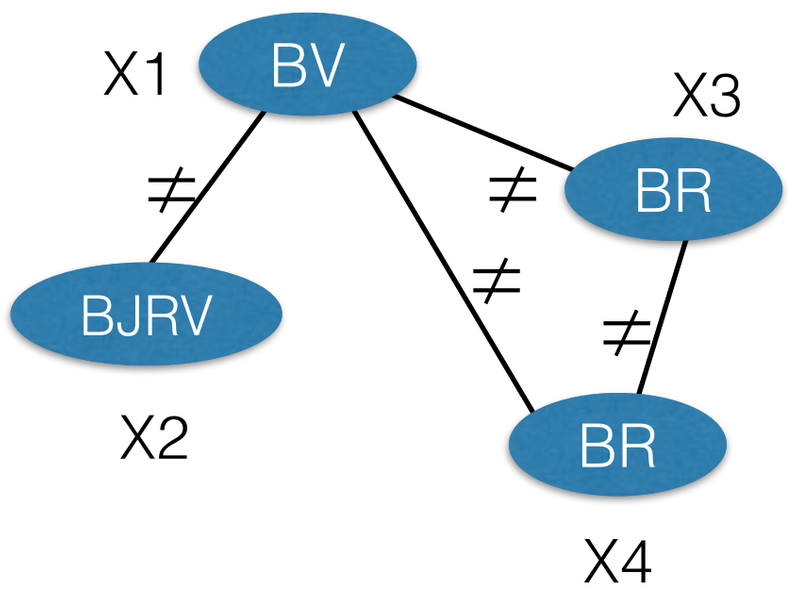
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



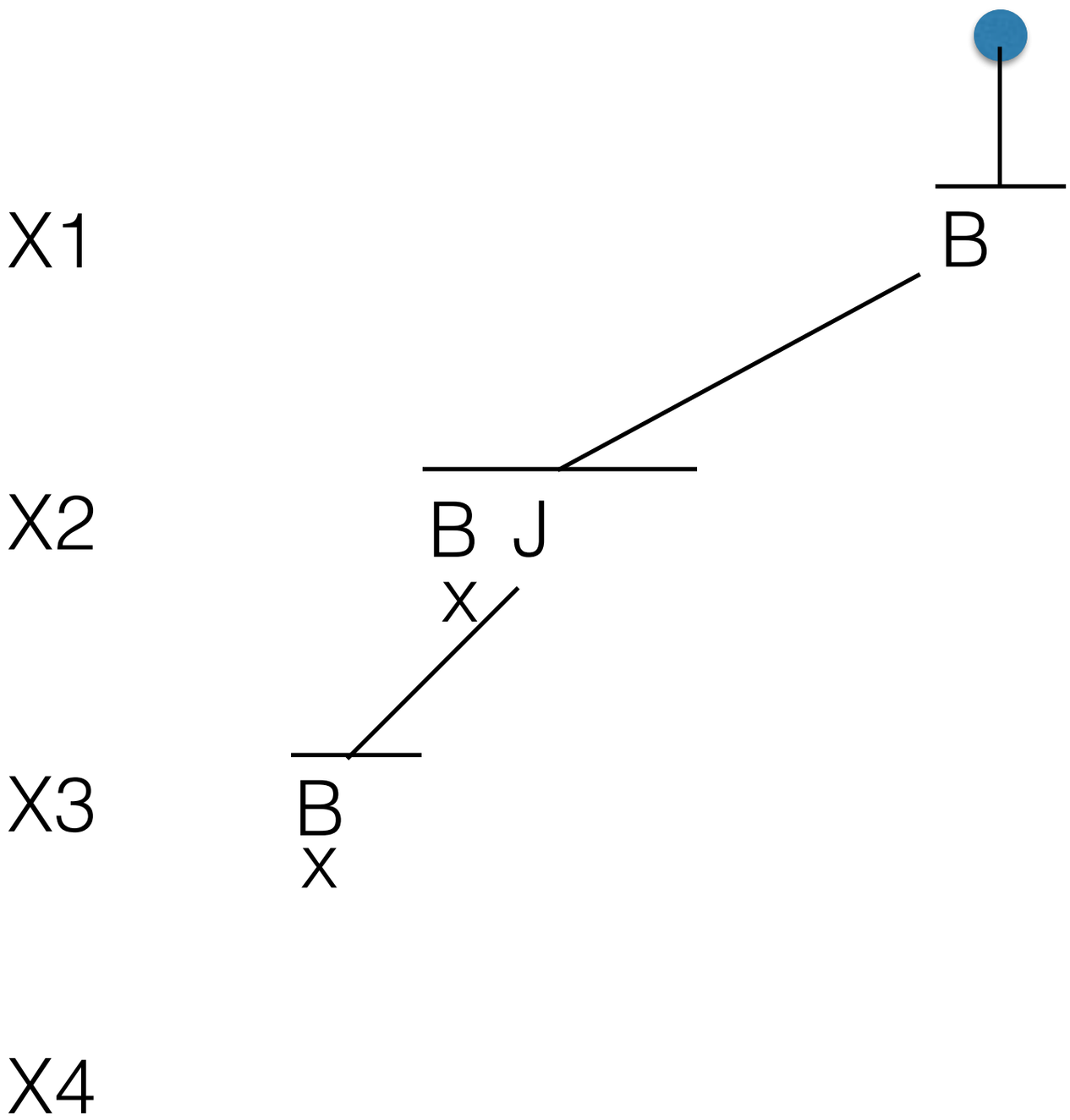
BT on running example



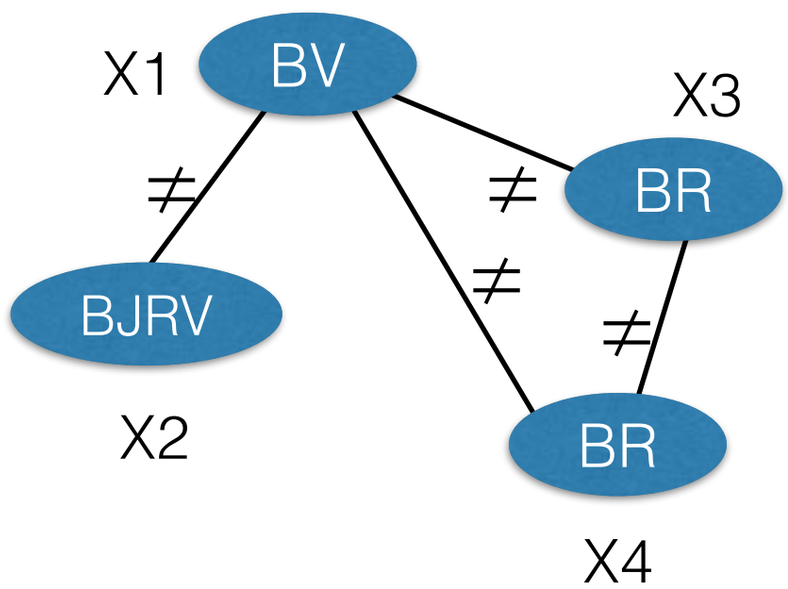
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



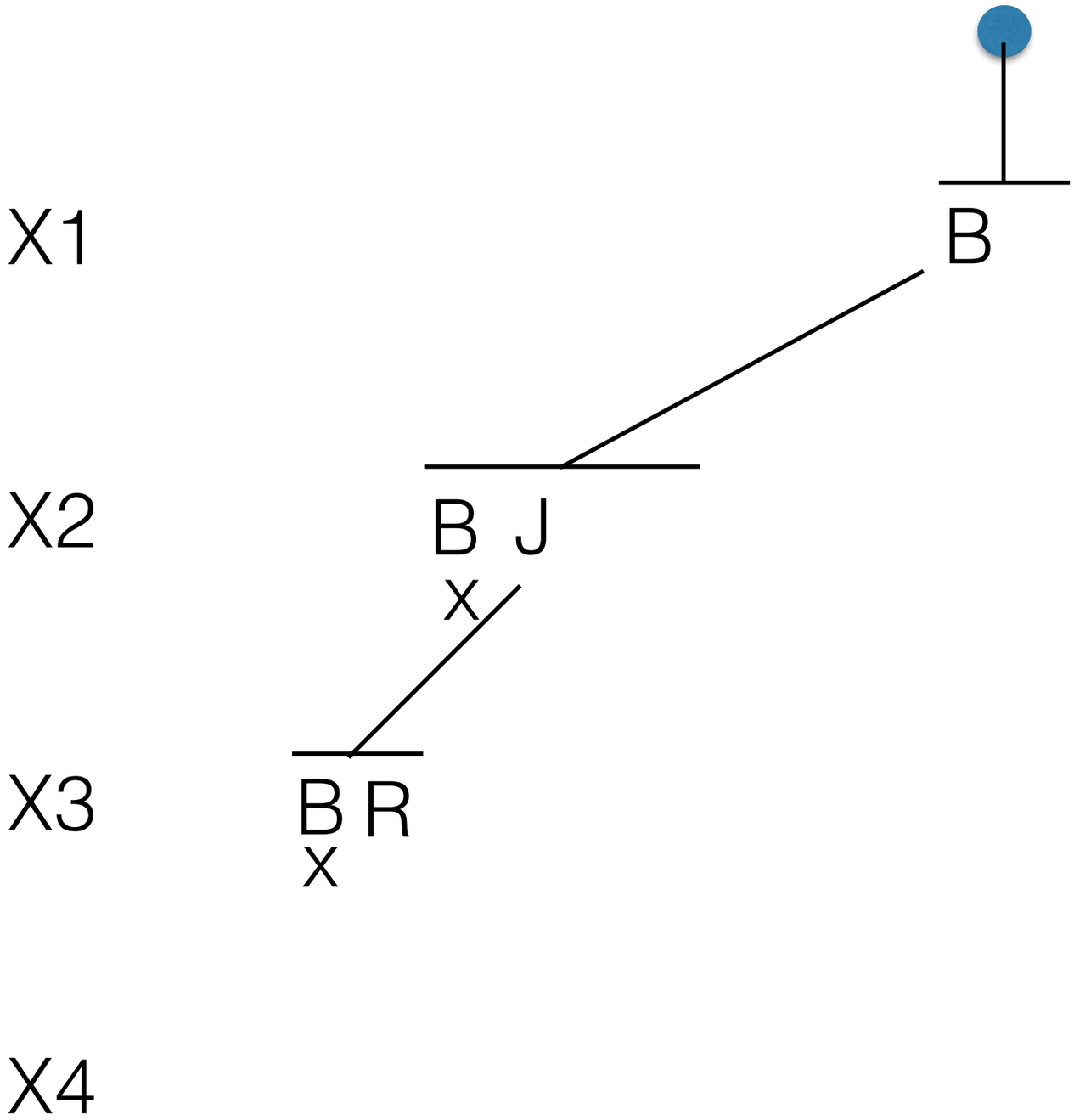
BT on running example



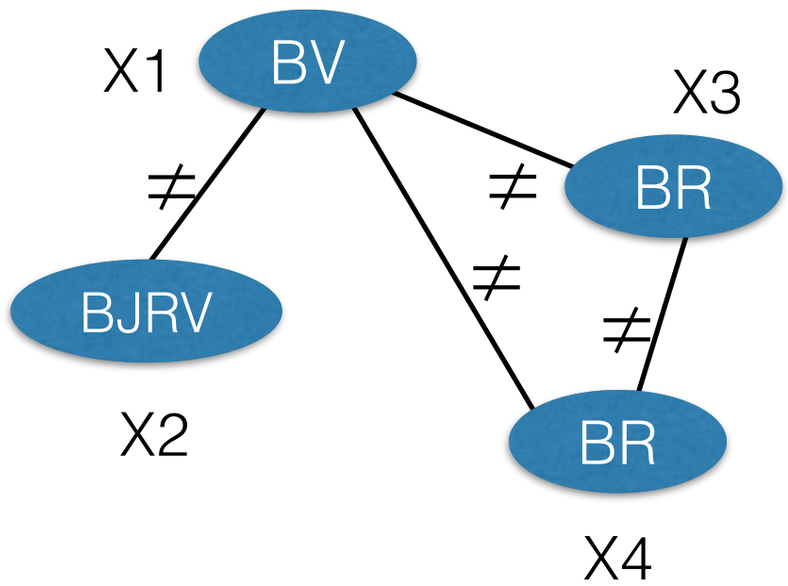

```
function BT(N : problem ; I : instantiation) :Boolean
begin
    if |I| = n then return true
    choose a variable X_i not in I
    for each v_i ∈ D(X_i) do
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```



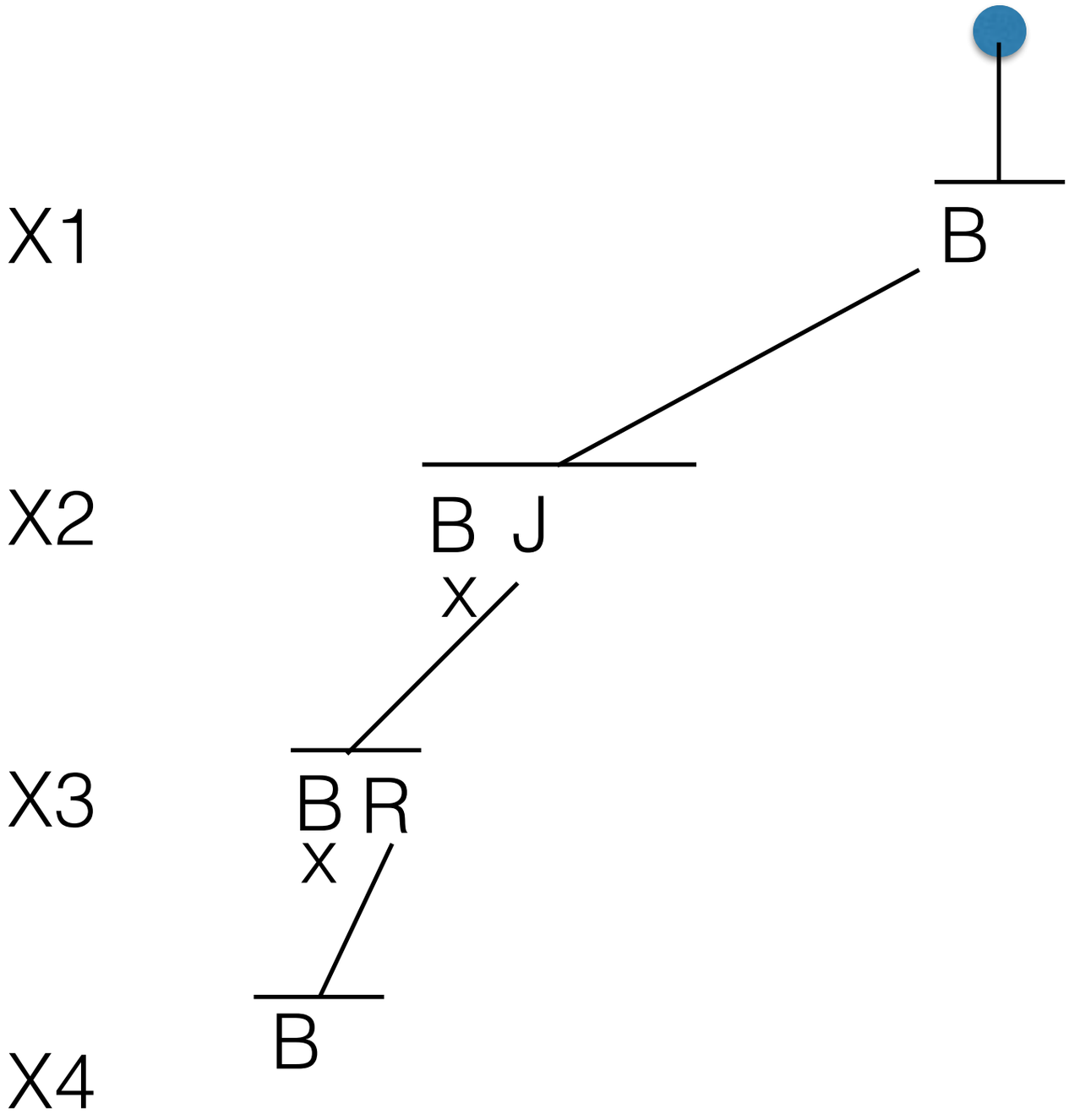
BT on running example



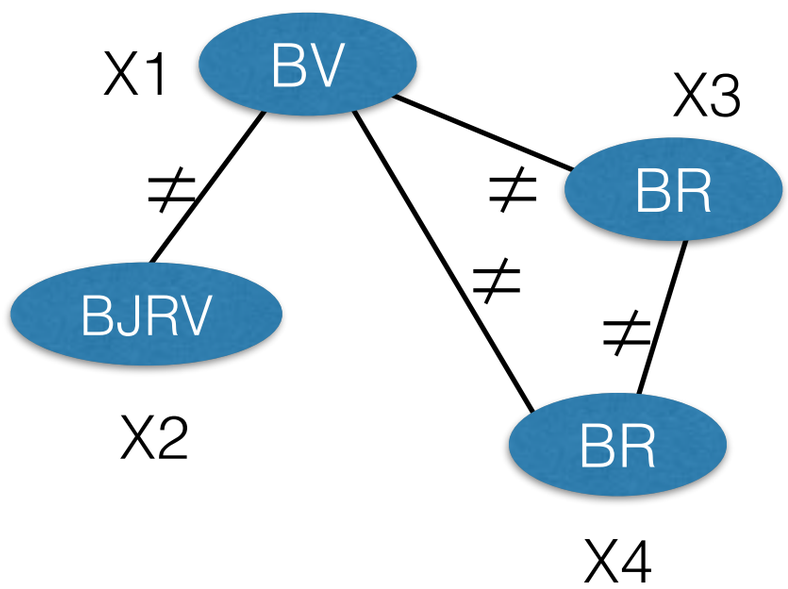

```
function BT(N : problem ; I : instantiation) :Boolean
begin
    if |I| = n then return true
    choose a variable X_i not in I
    for each v_i ∈ D(X_i) do
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```



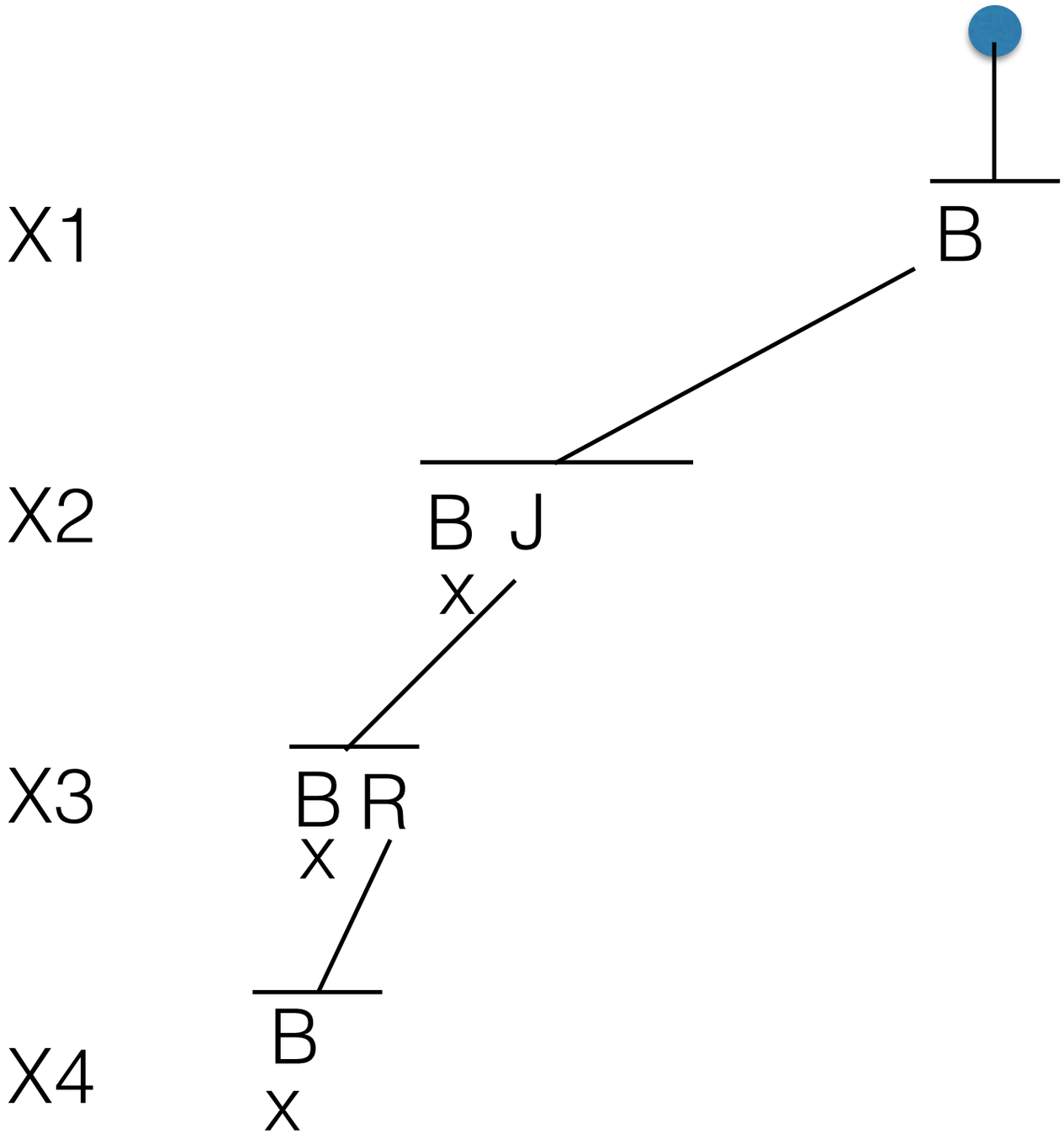
BT on running example



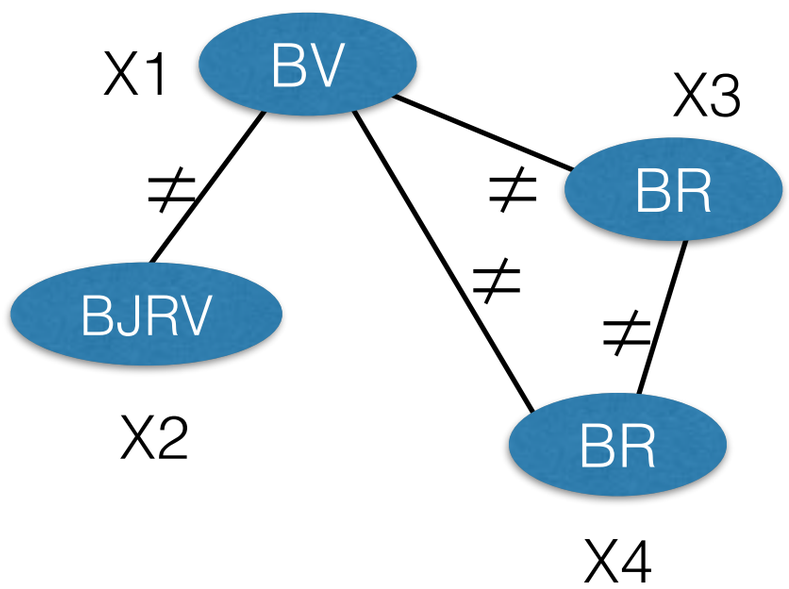

```
function BT(N : problem ; I : instantiation) :Boolean
begin
    if |I| = n then return true
    choose a variable X_i not in I
    for each v_i ∈ D(X_i) do
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```



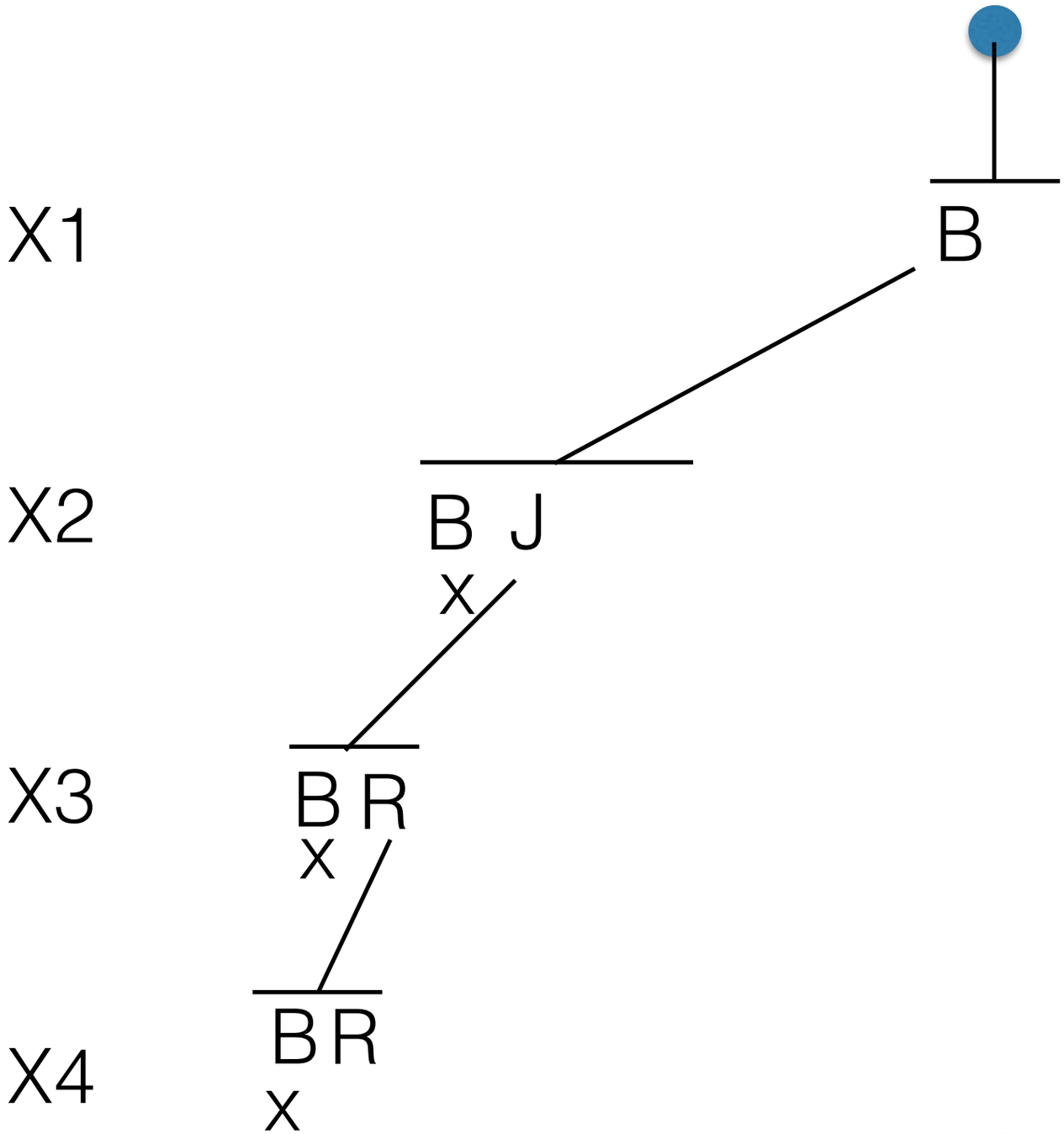
BT on running example



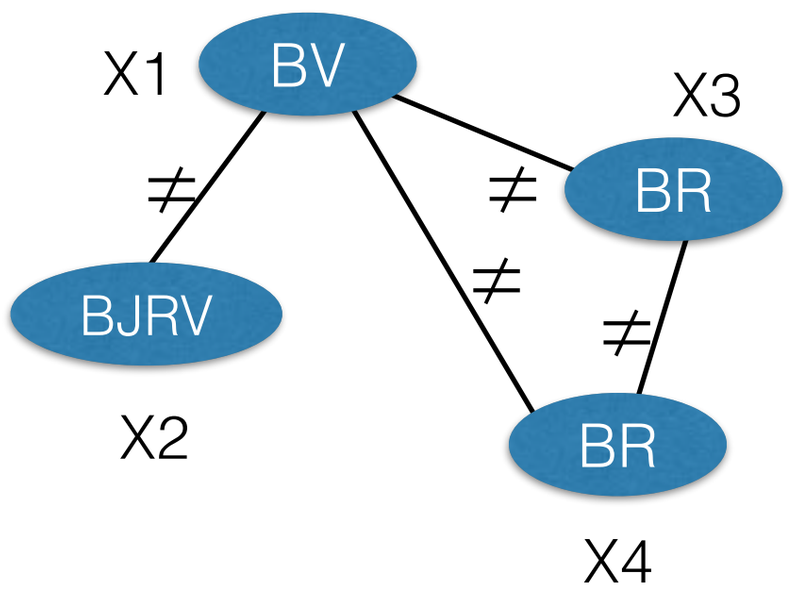
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



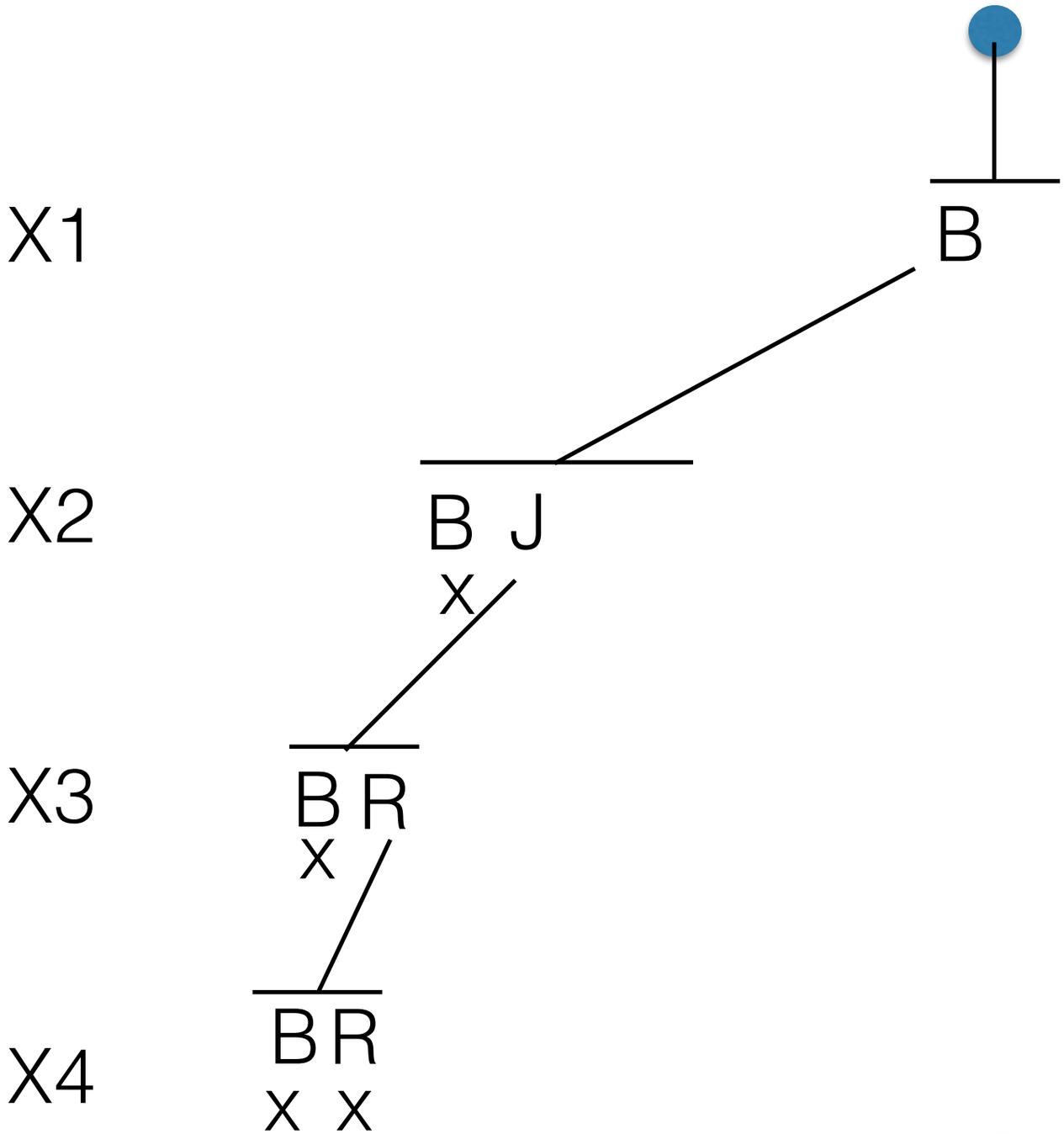
BT on running example



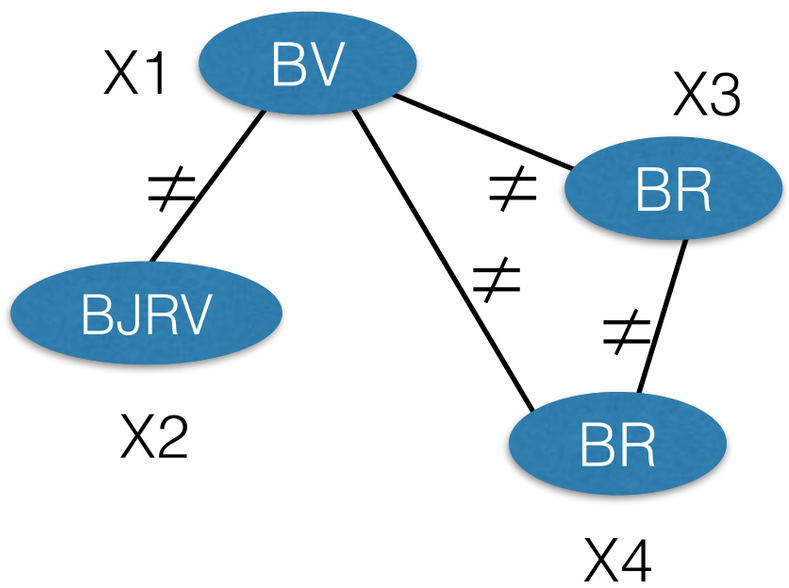
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



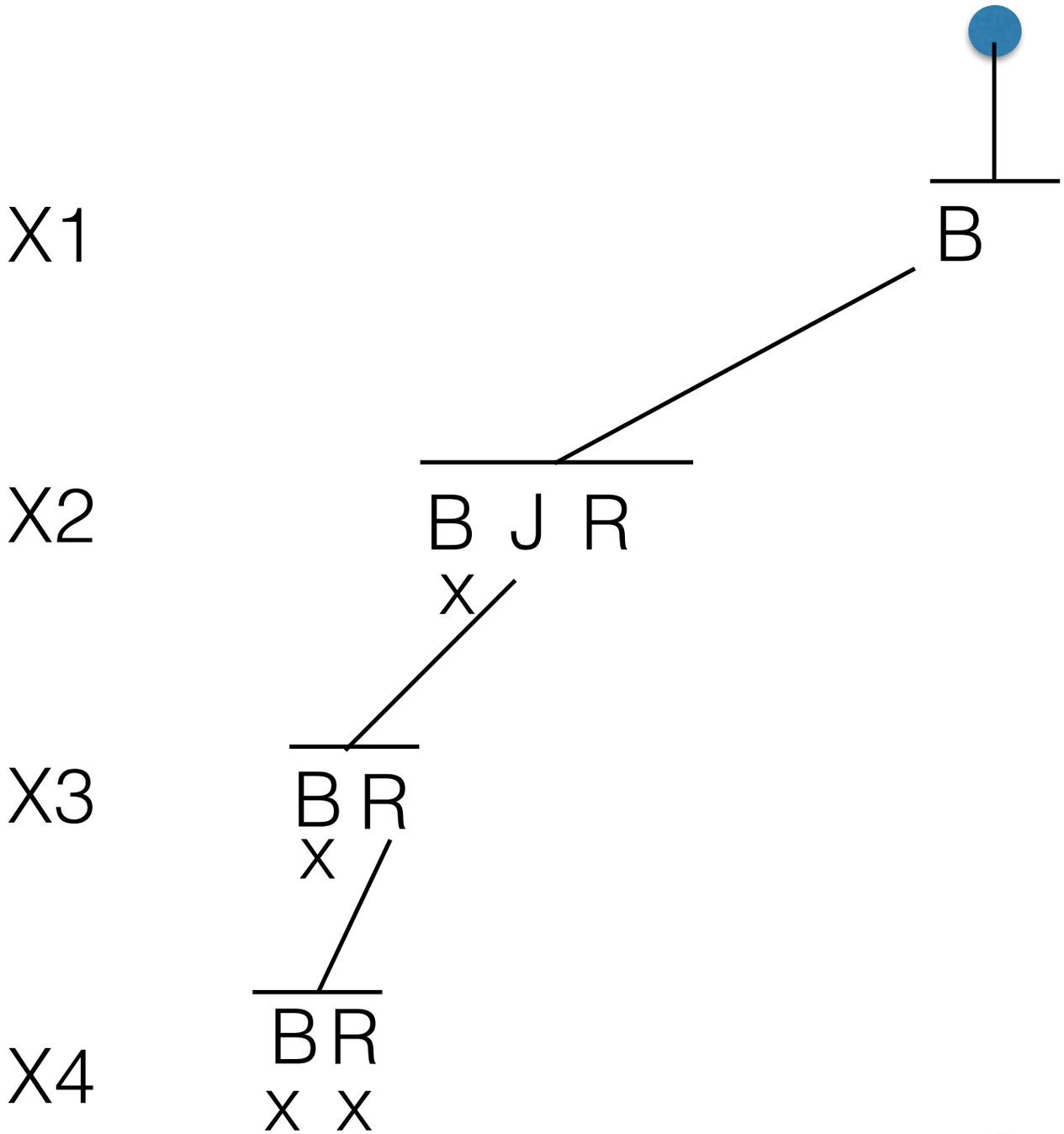
BT on running example



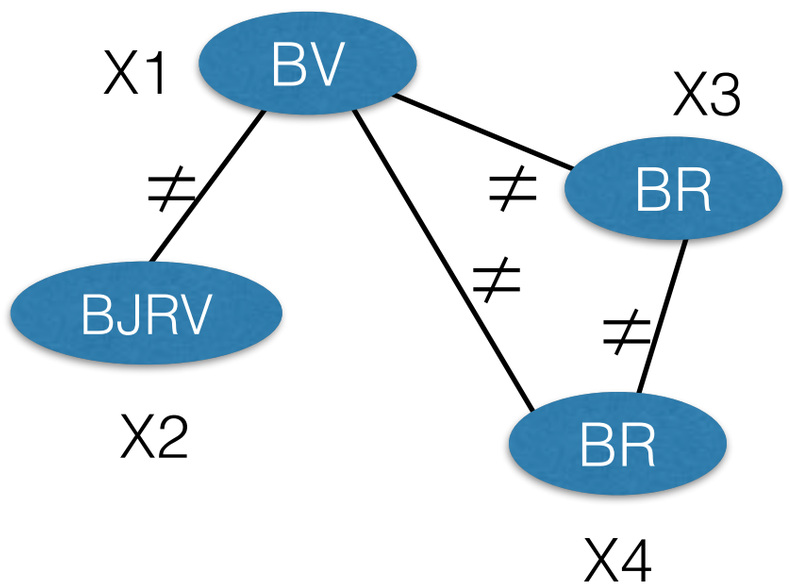
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



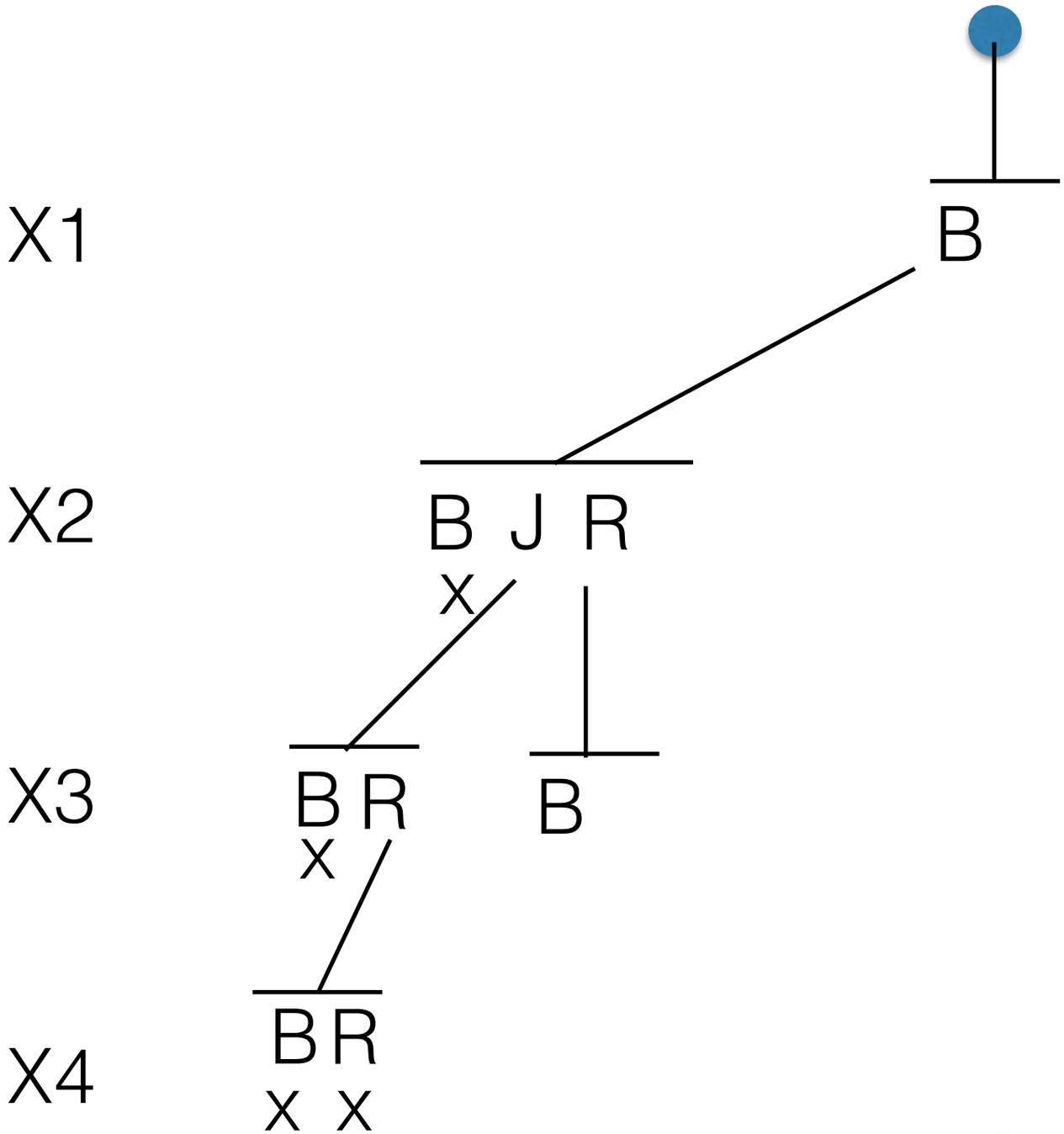
BT on running example



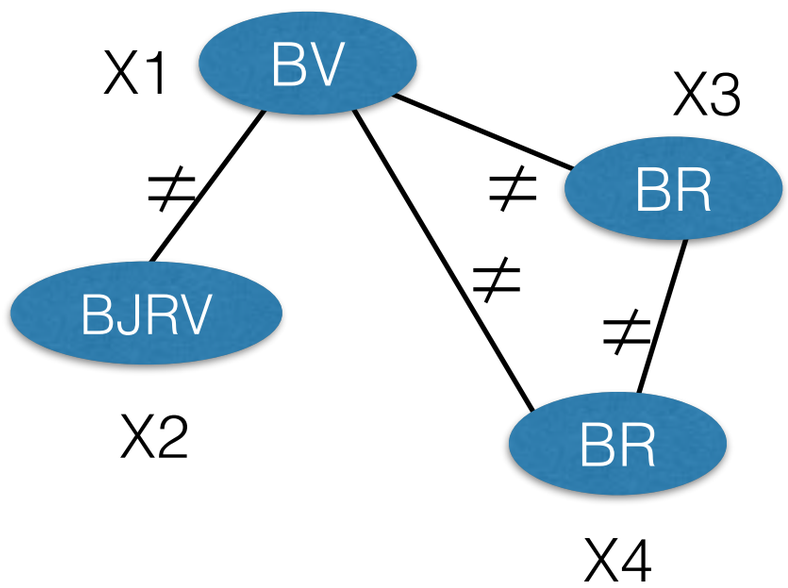
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



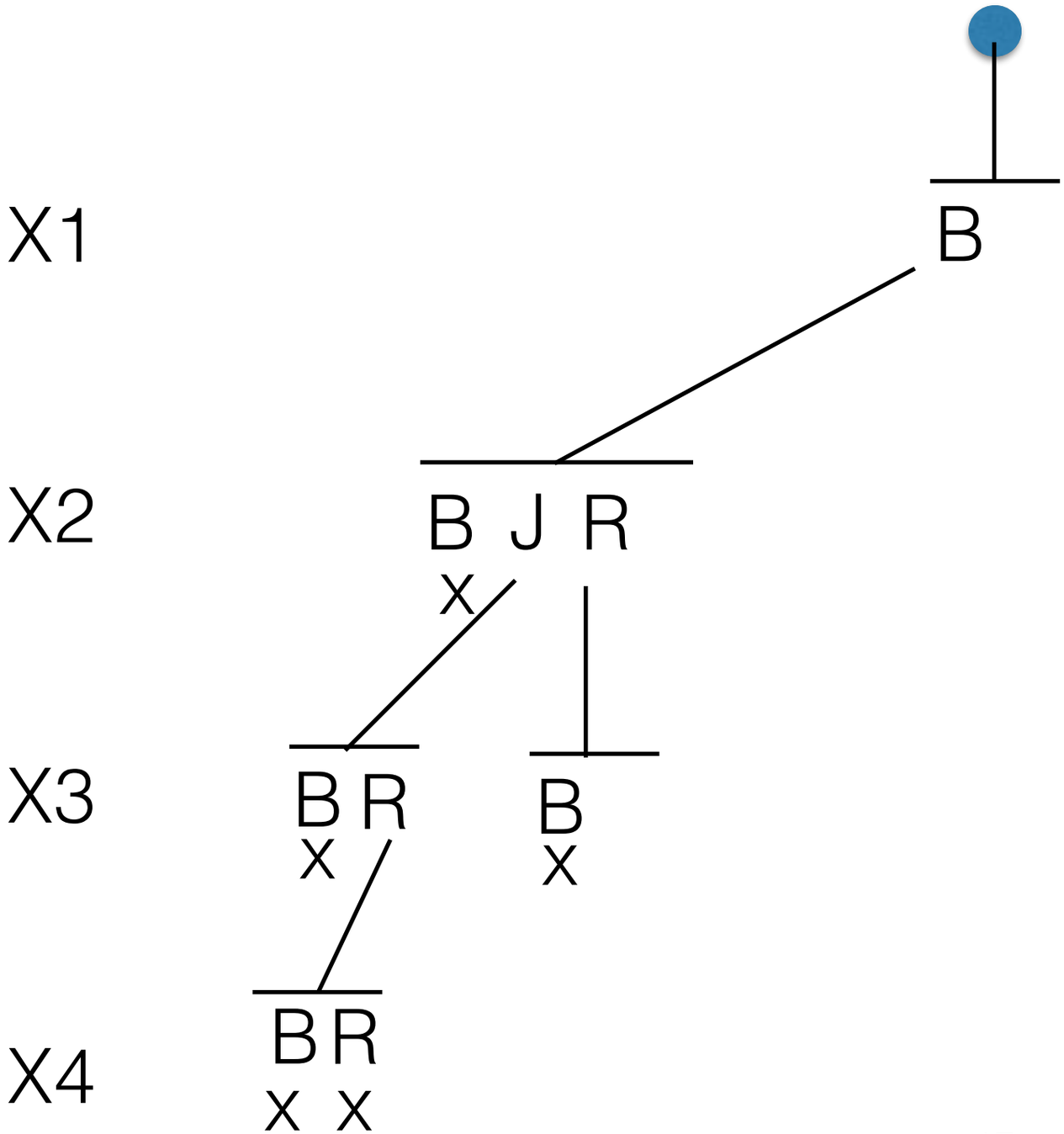
BT on running example



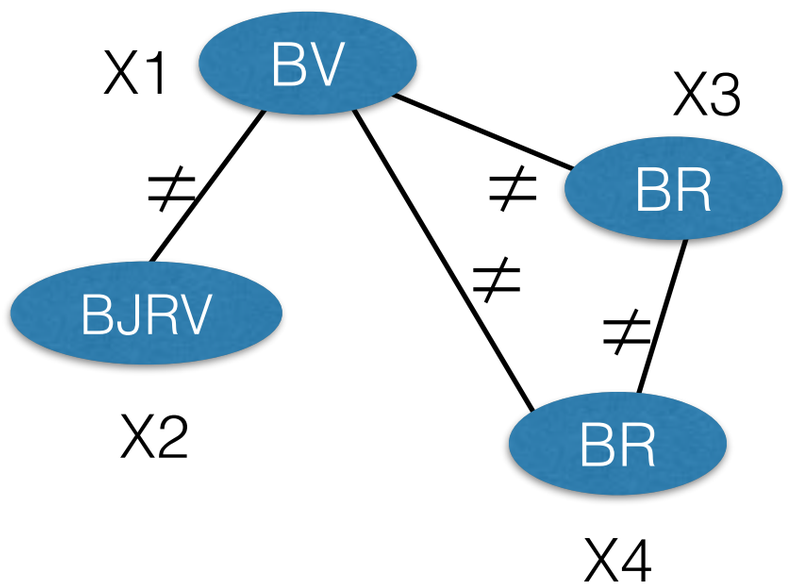
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



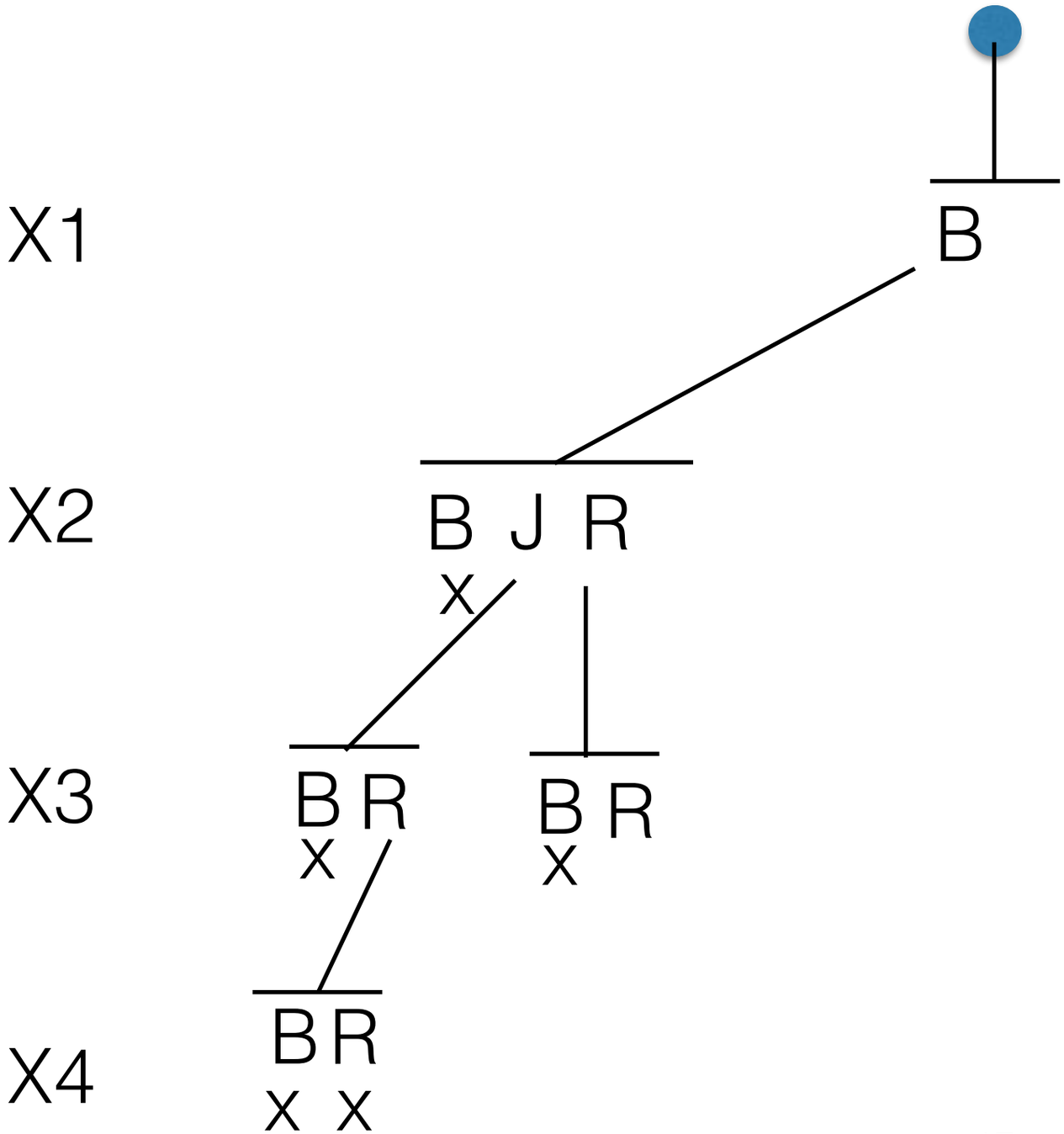
BT on running example



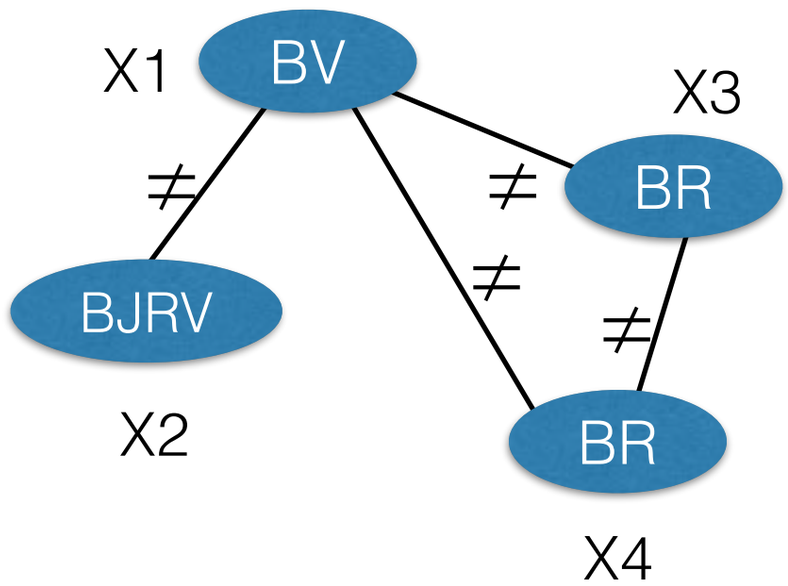
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT$(N, I \cup \{X_i, v_i\})$ **then** return **true**
  return **false**
end



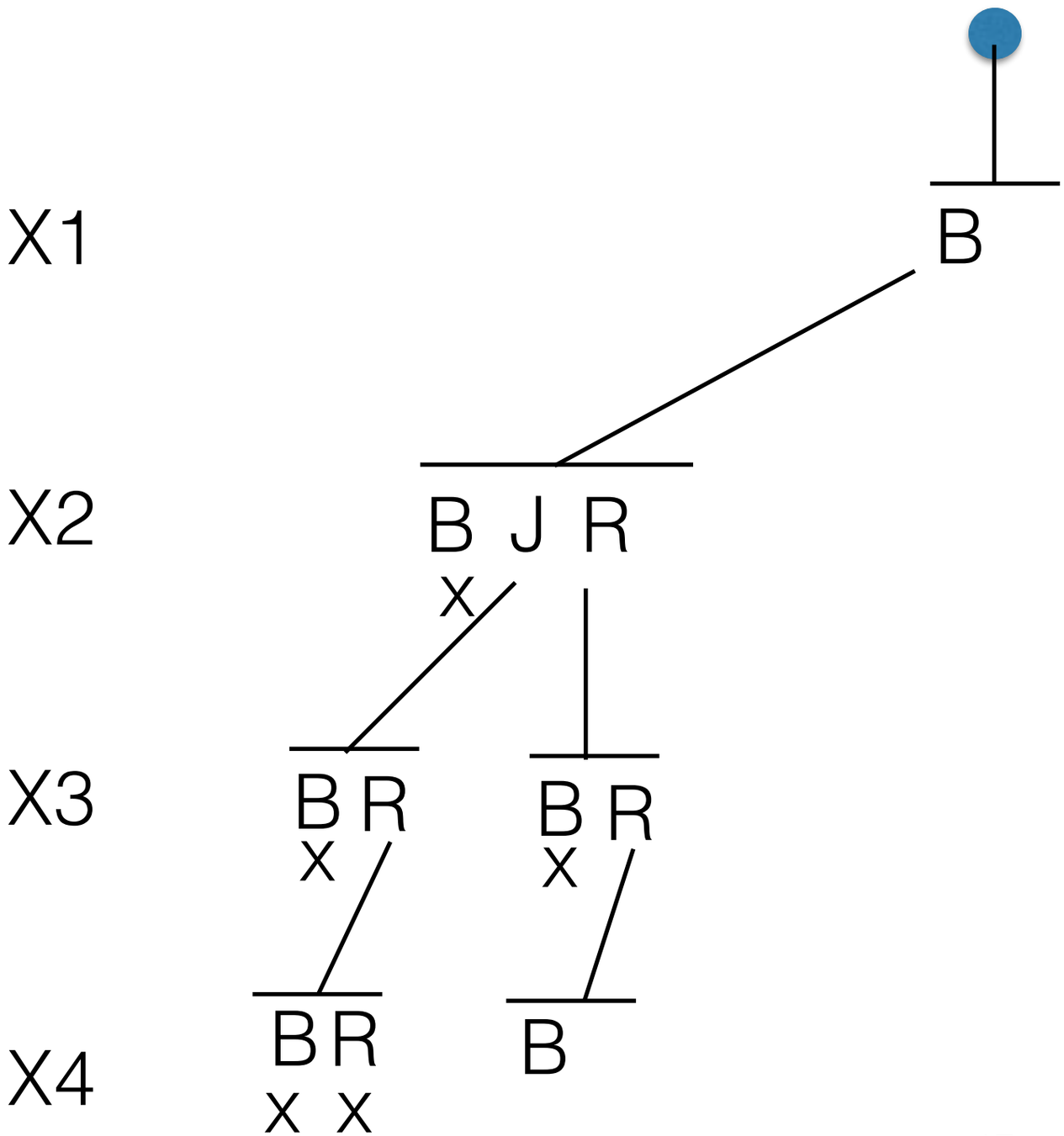
BT on running example



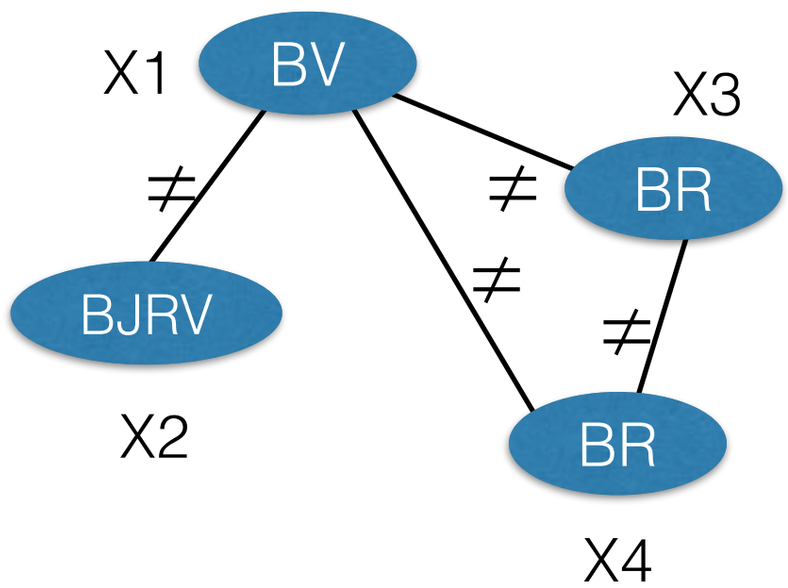

```
function BT(N : problem ; I : instantiation) :Boolean
begin
    if |I| = n then return true
    choose a variable X_i not in I
    for each v_i ∈ D(X_i) do
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```



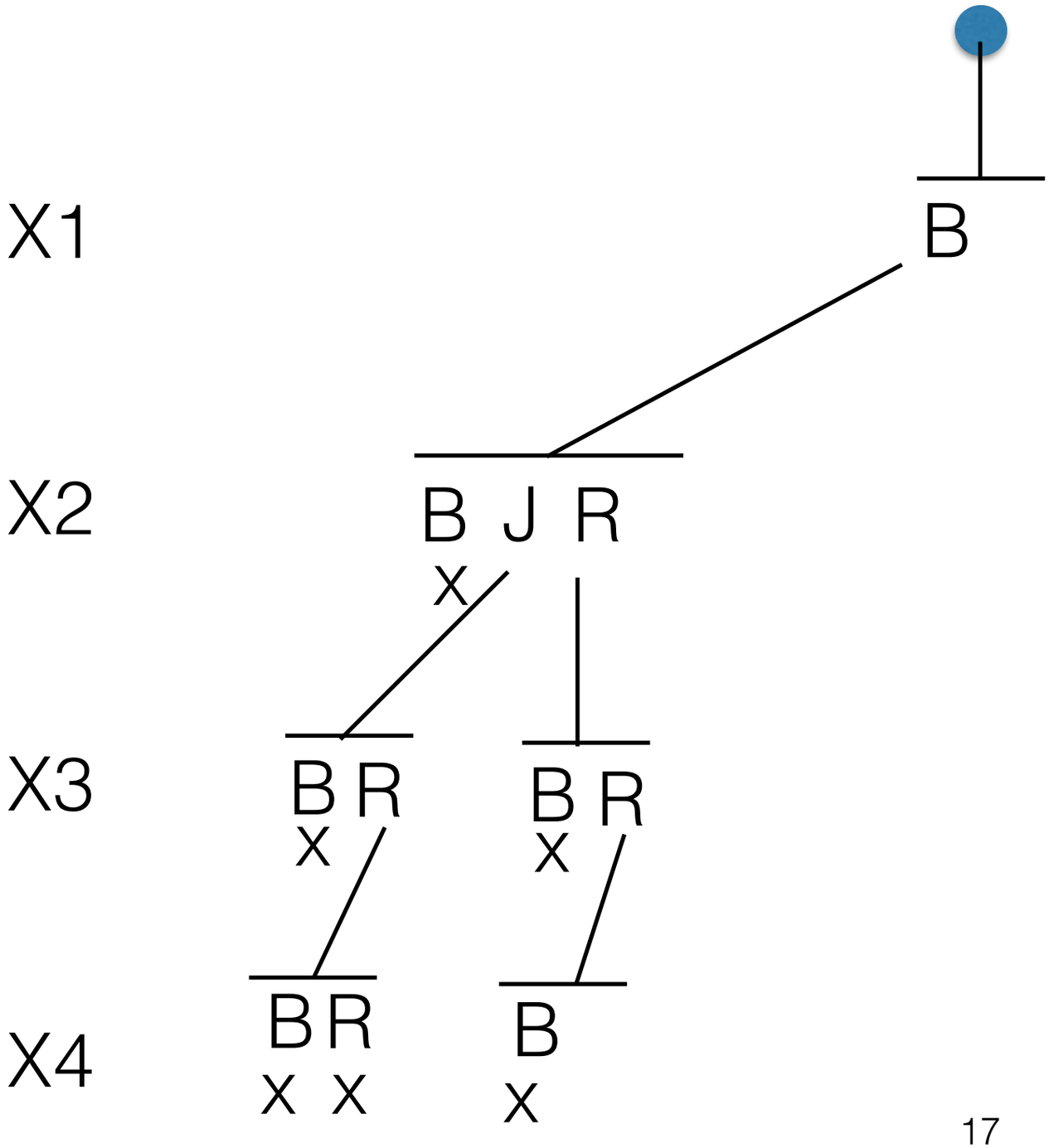
BT on running example



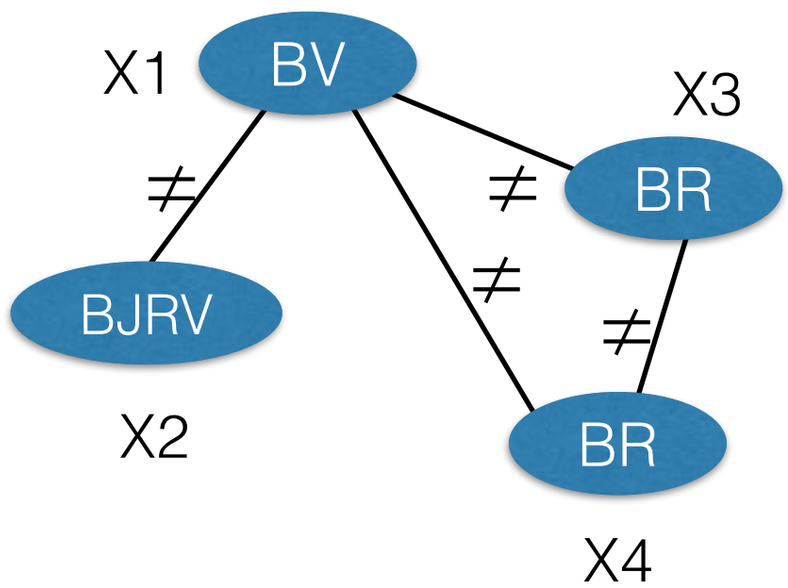
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



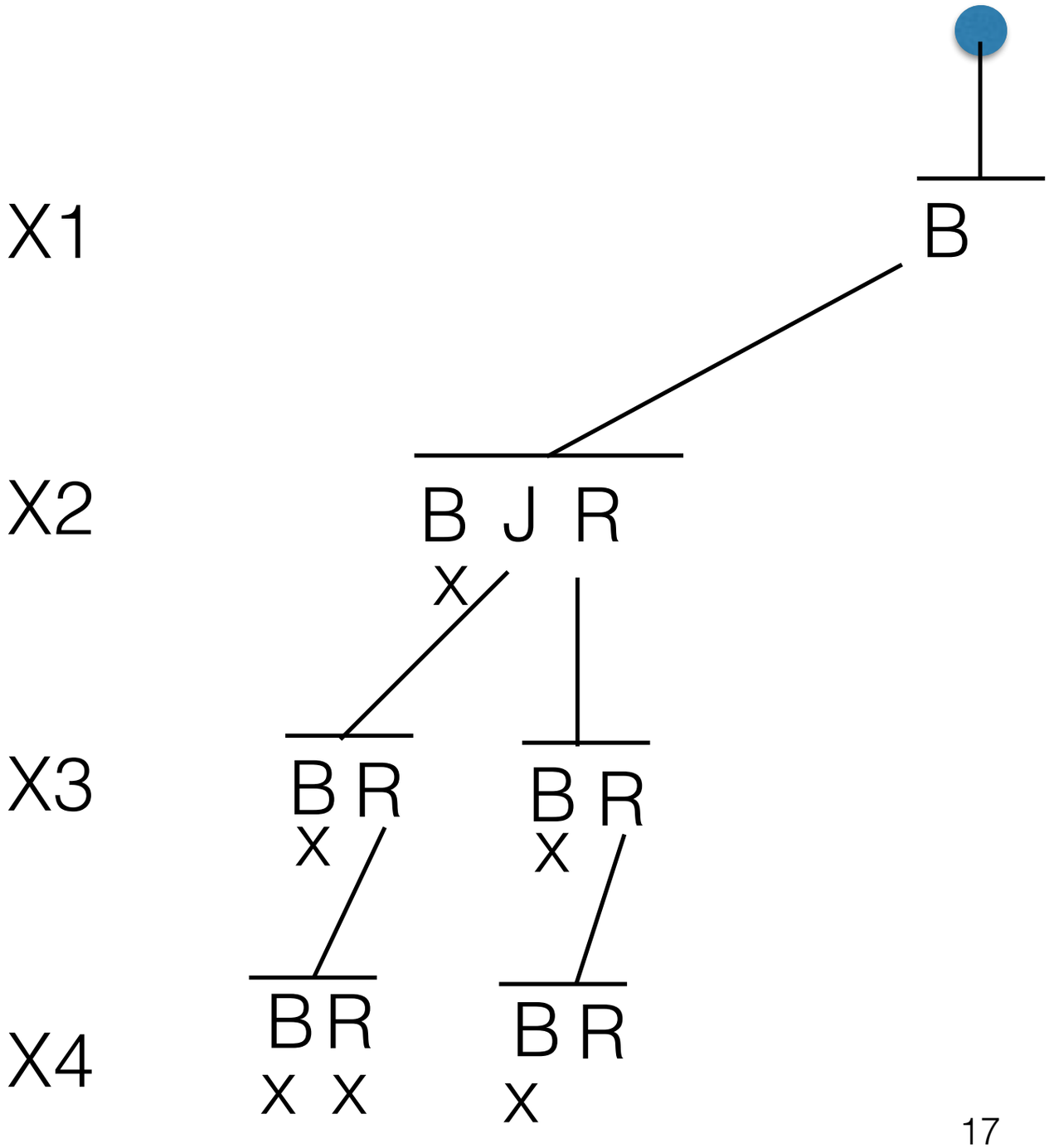
BT on running example



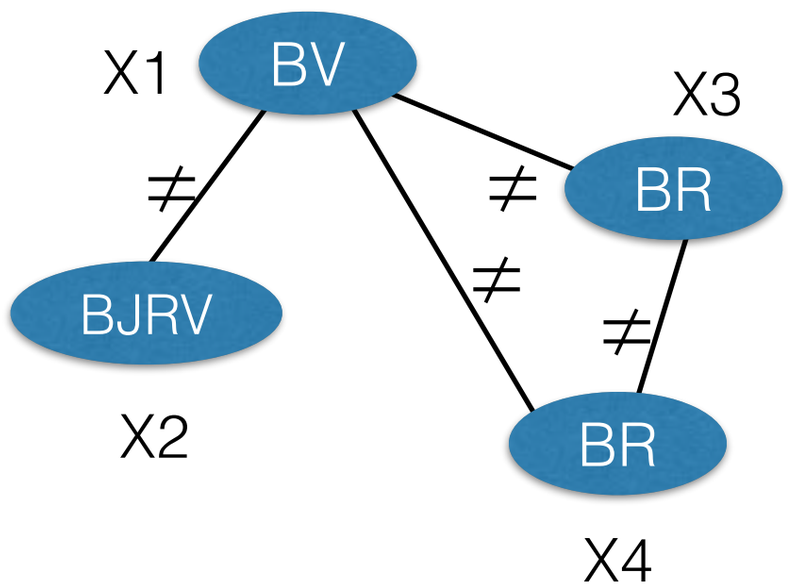
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



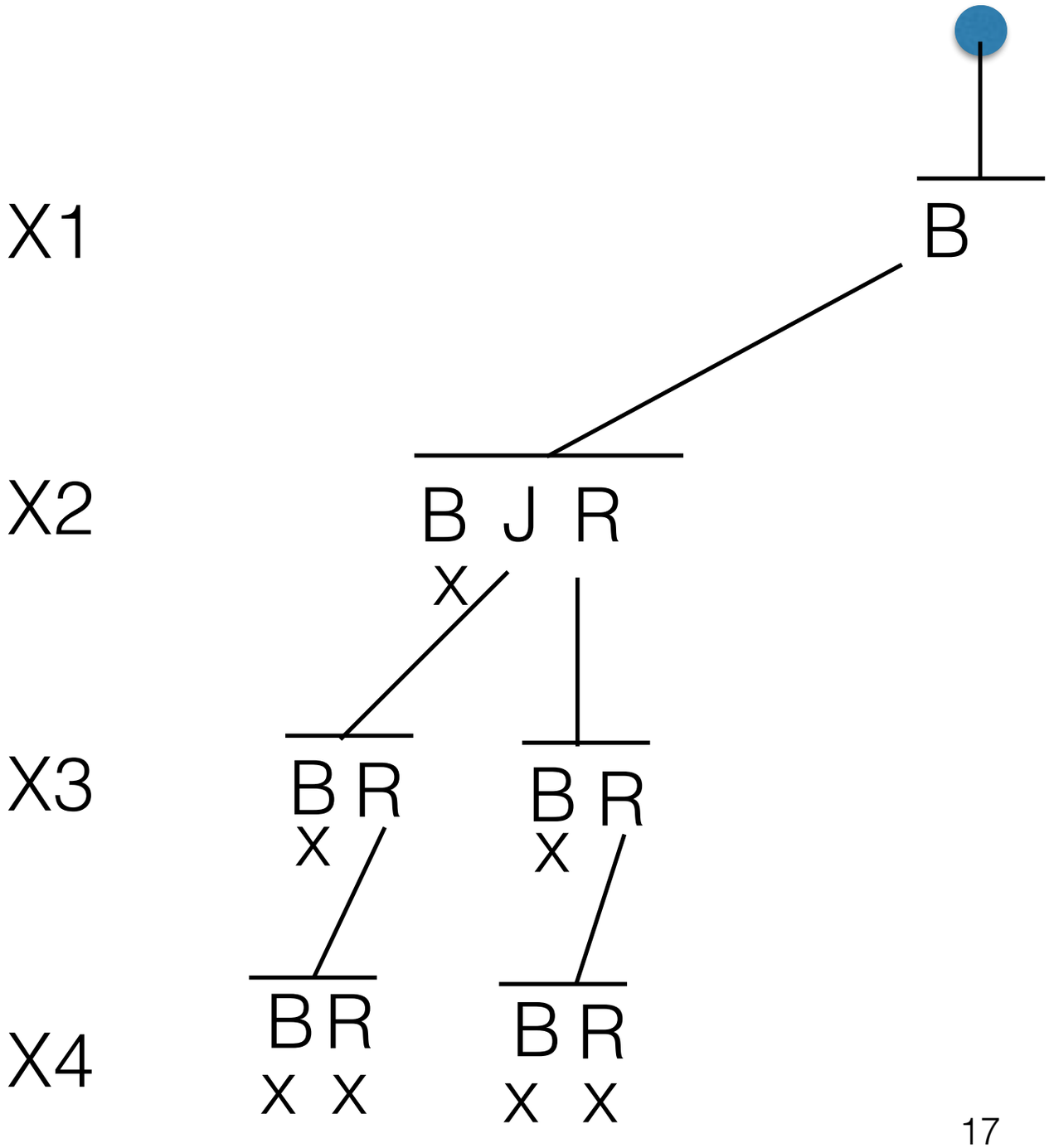
BT on running example



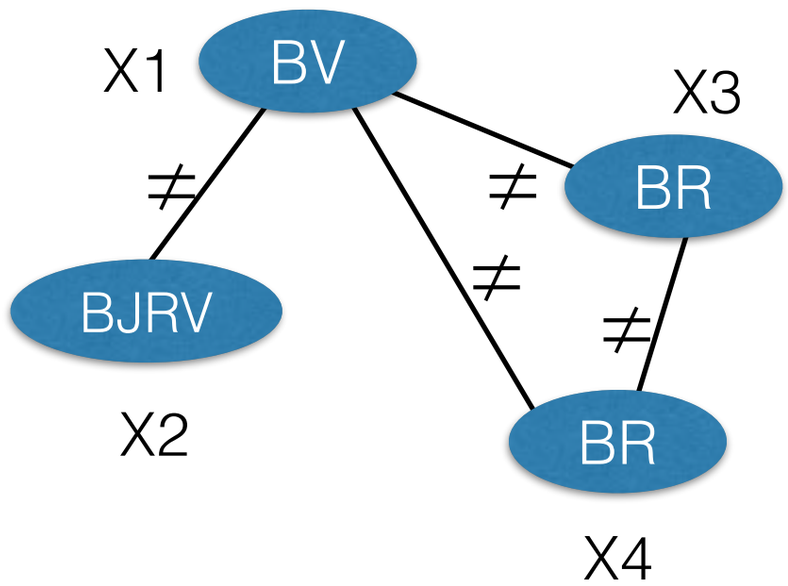
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



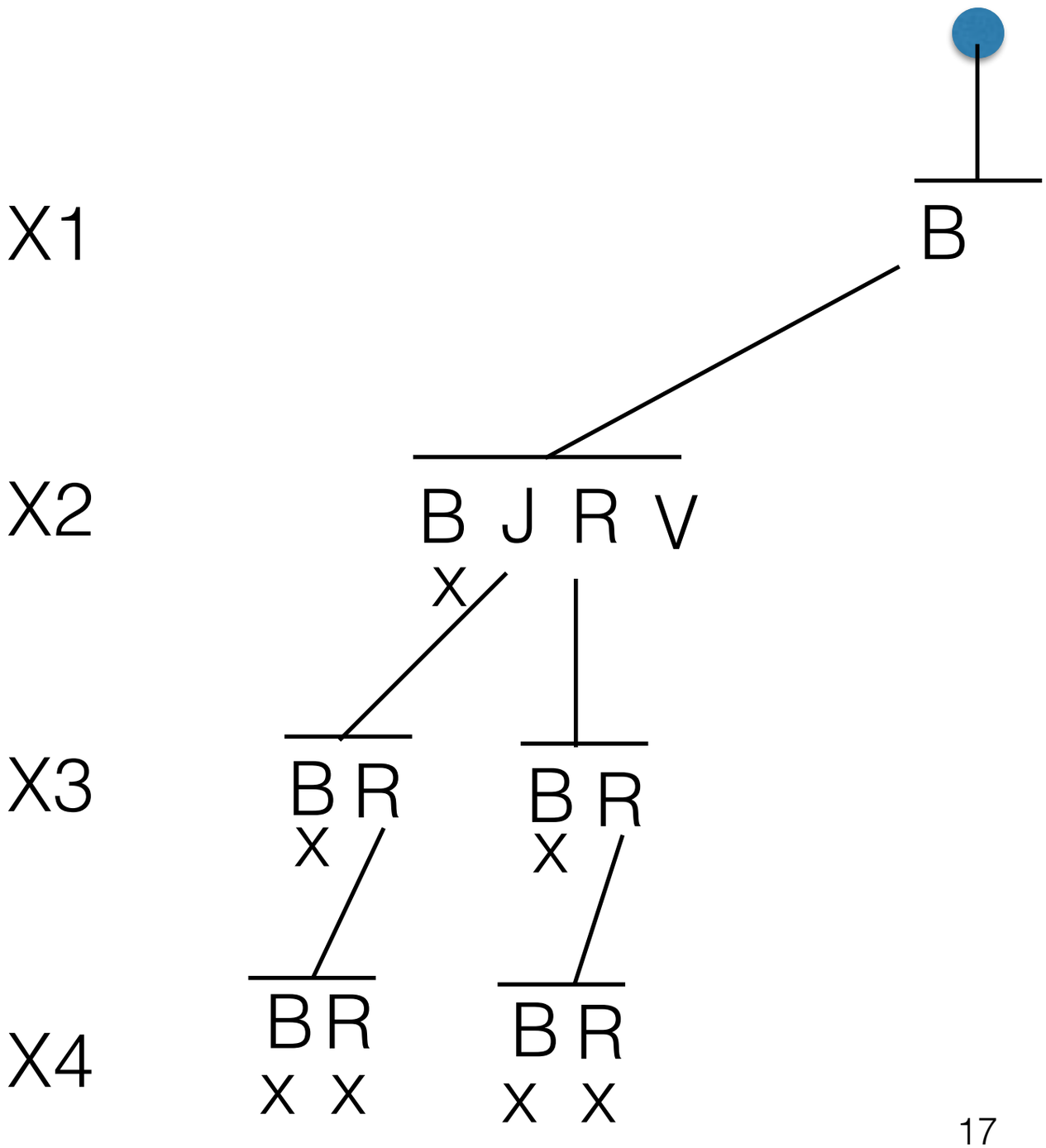
BT on running example



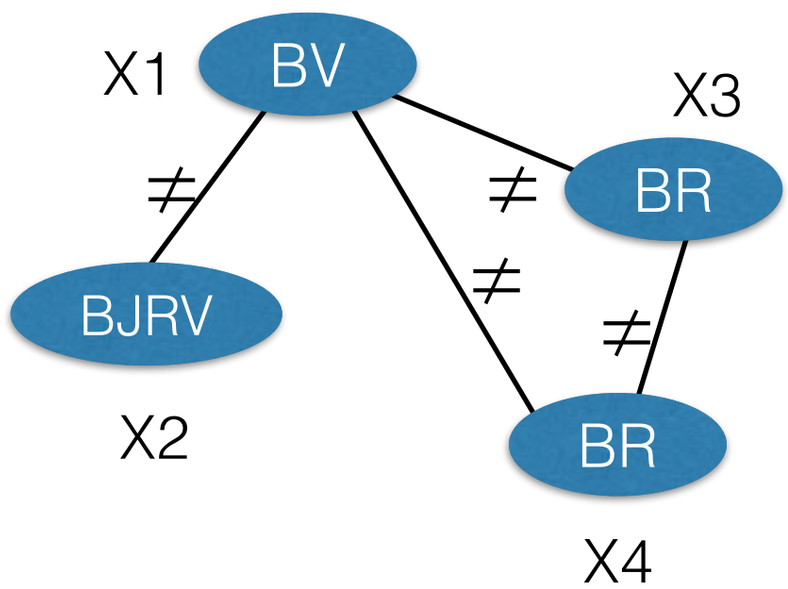
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



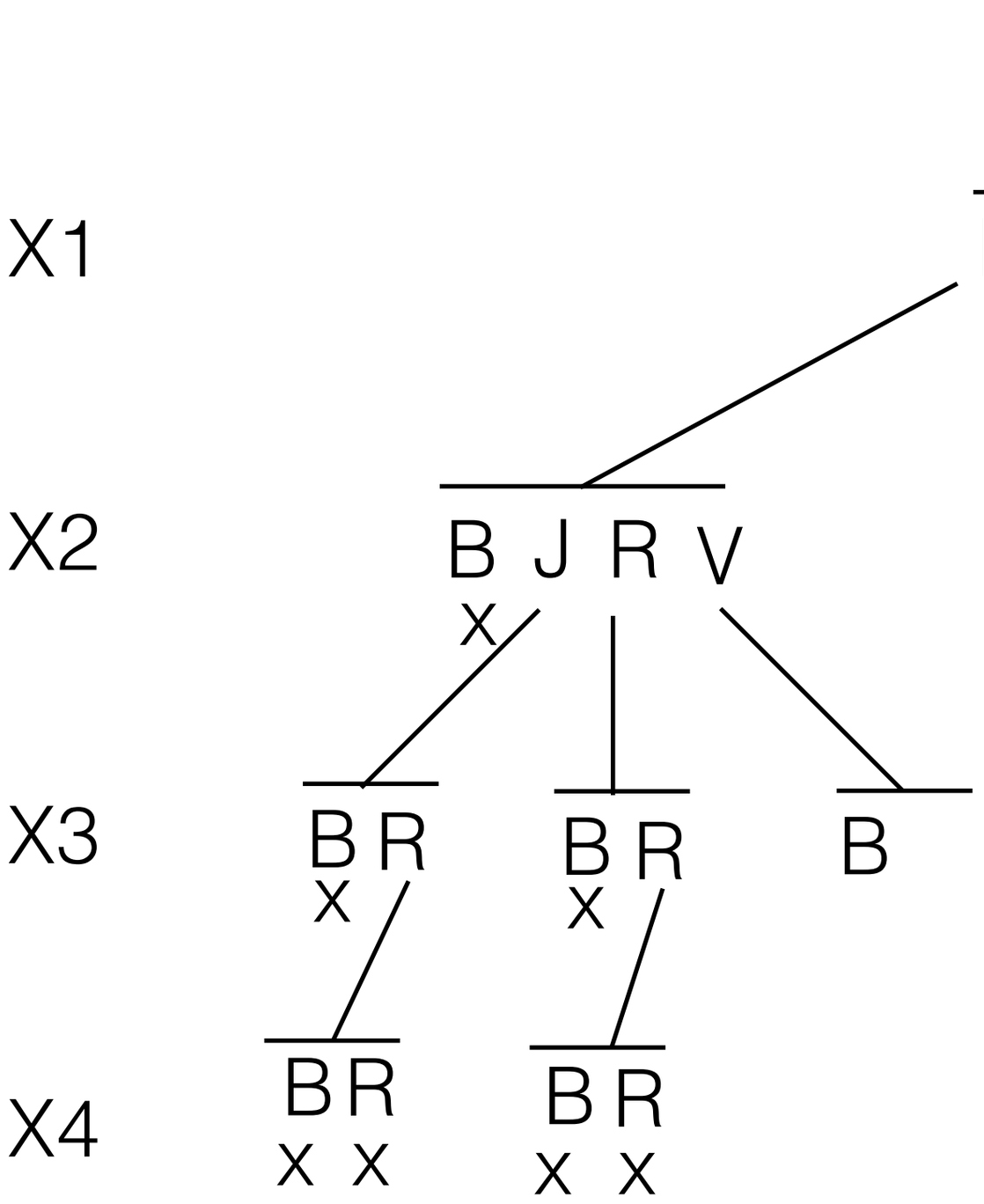
BT on running example



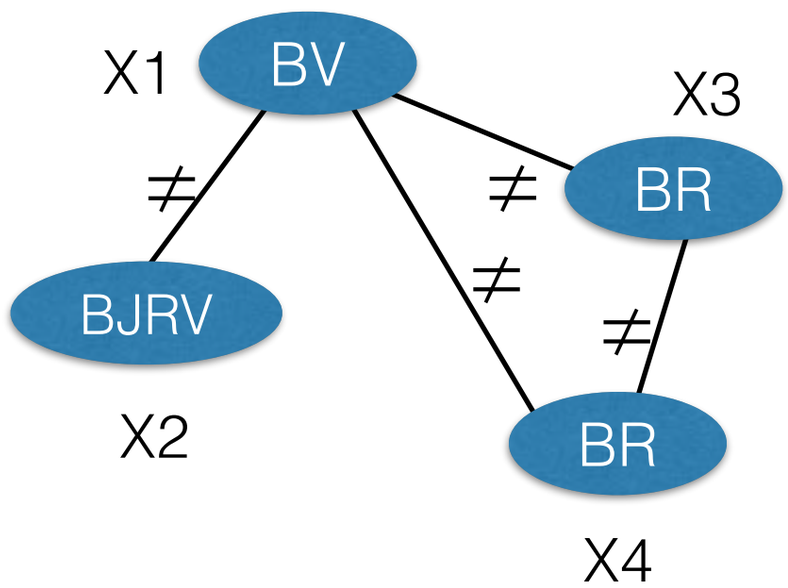
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



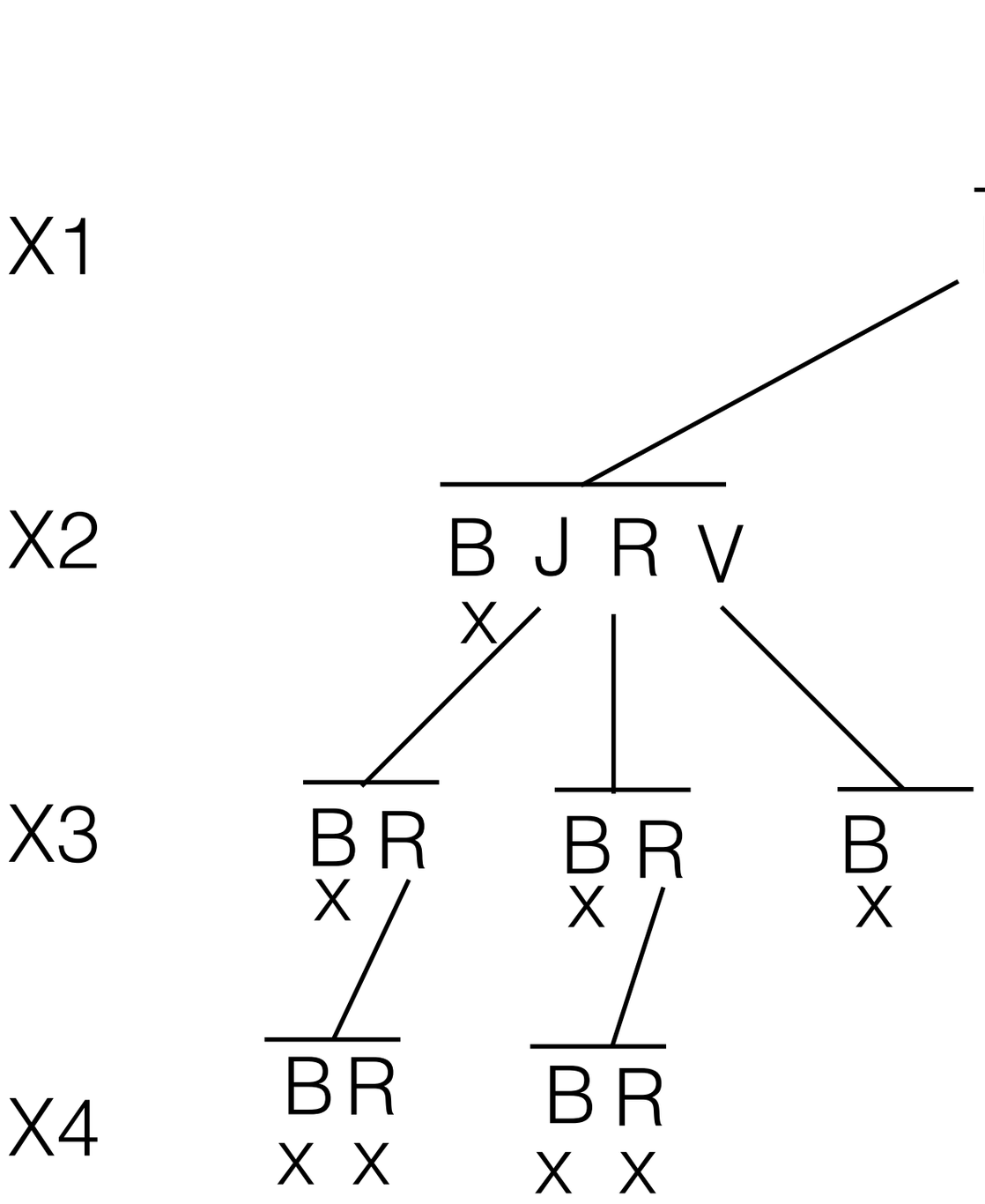
BT on running example



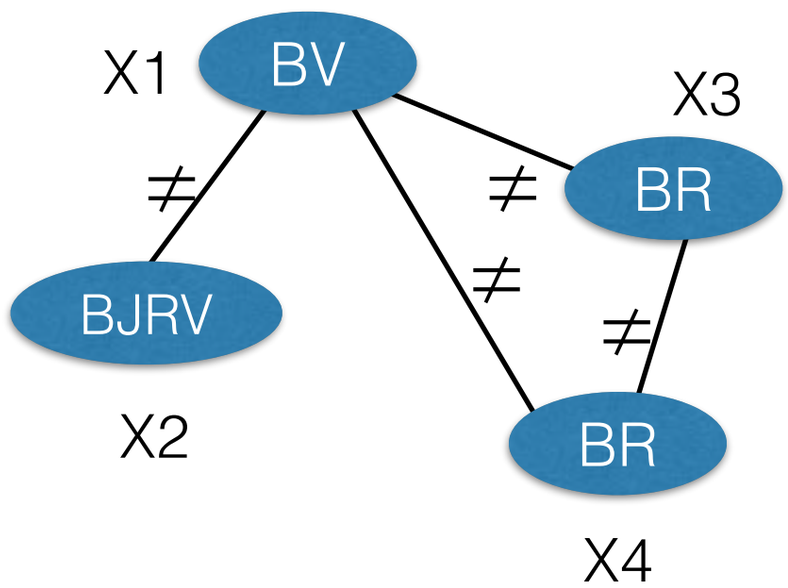
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



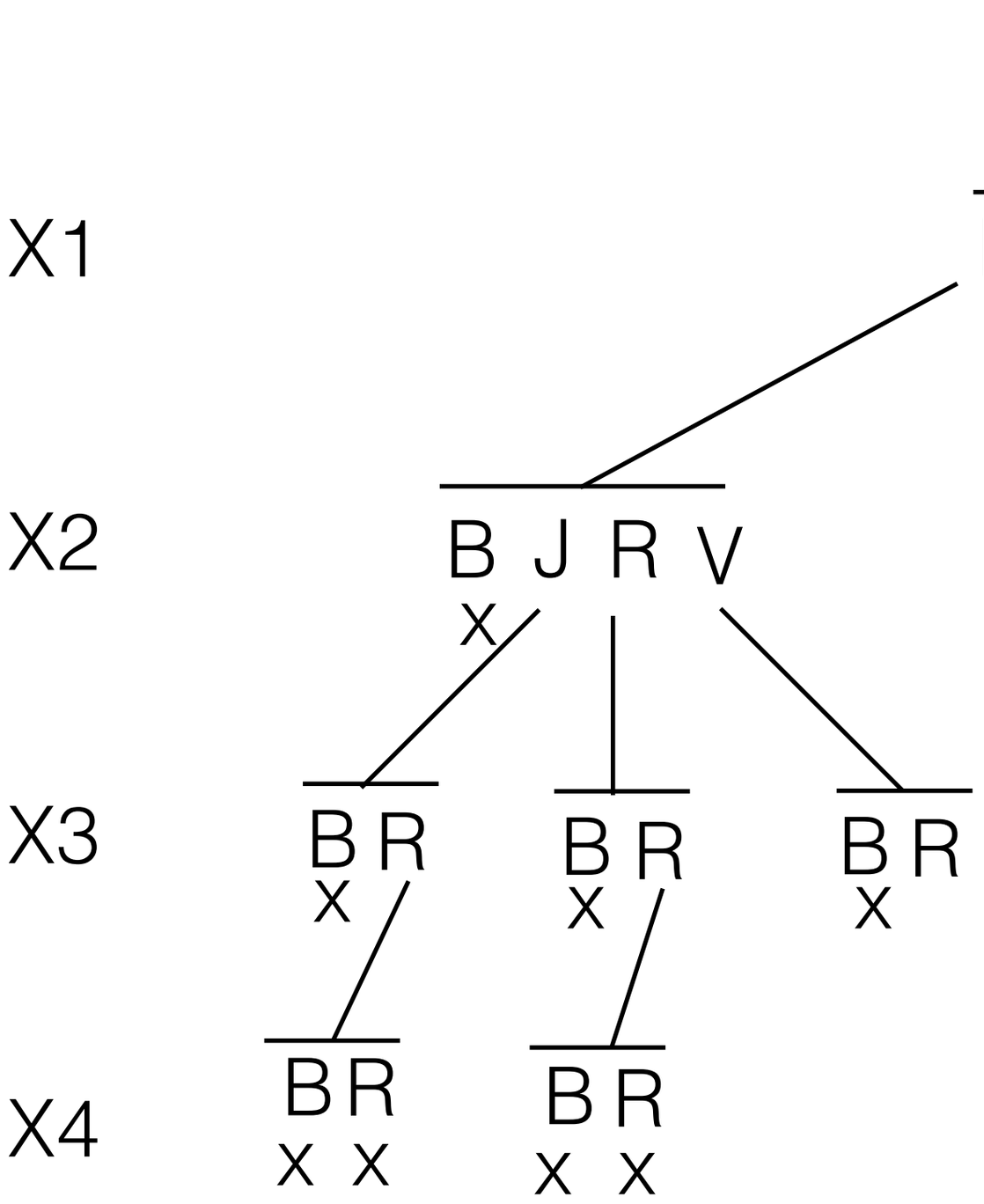
BT on running example



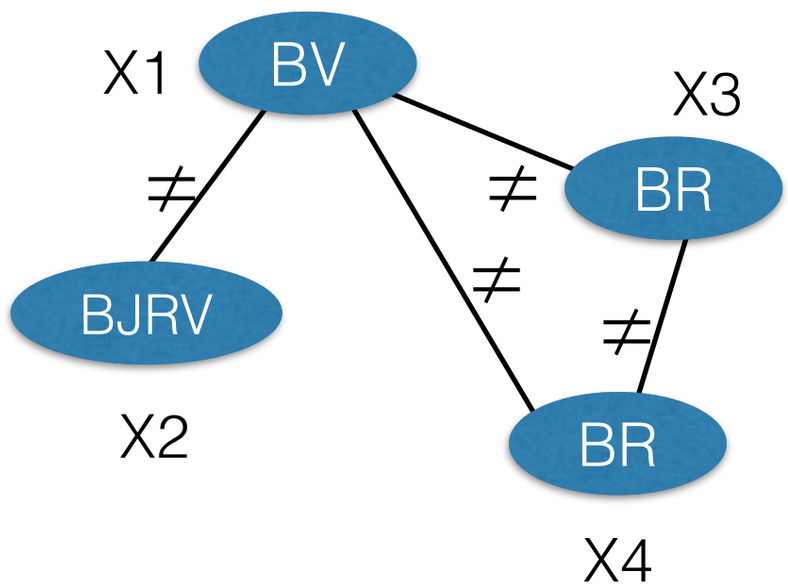
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



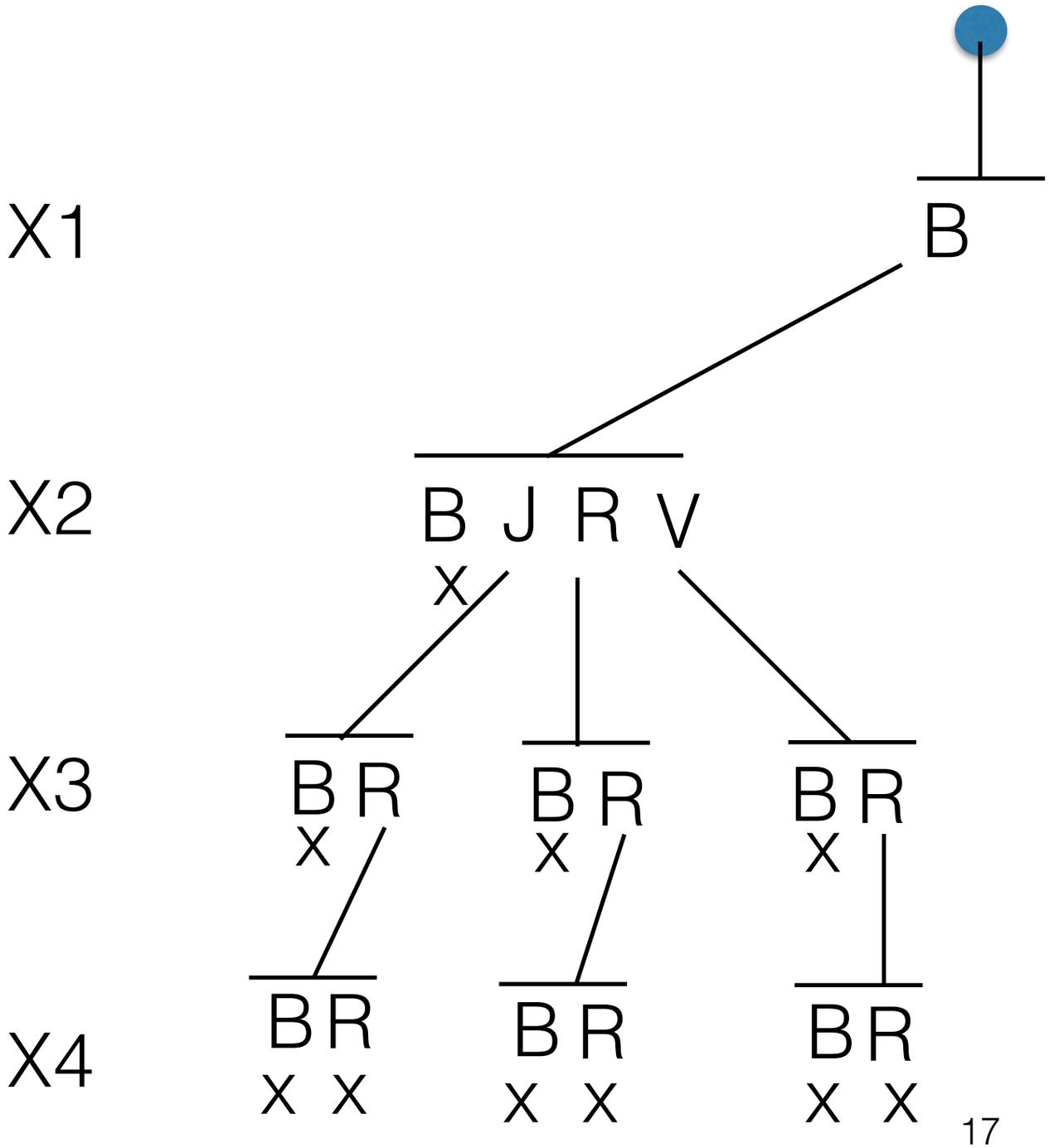
BT on running example



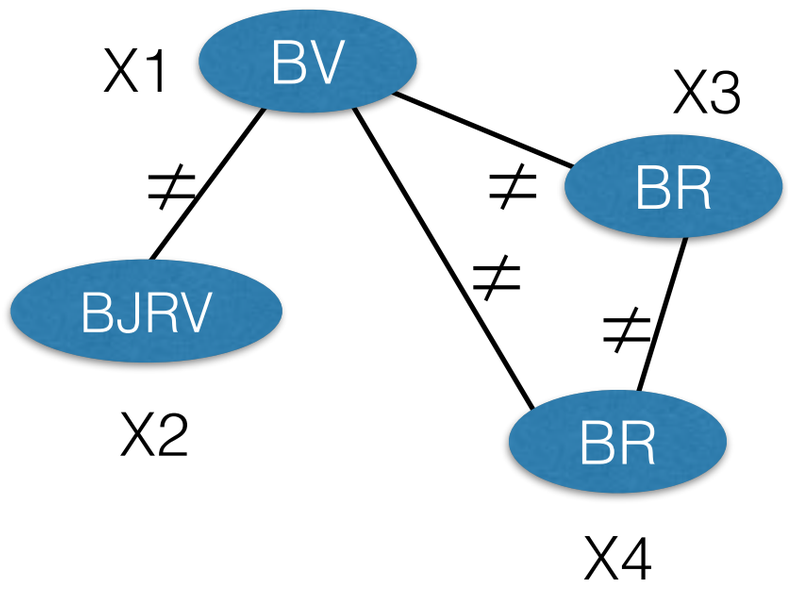
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



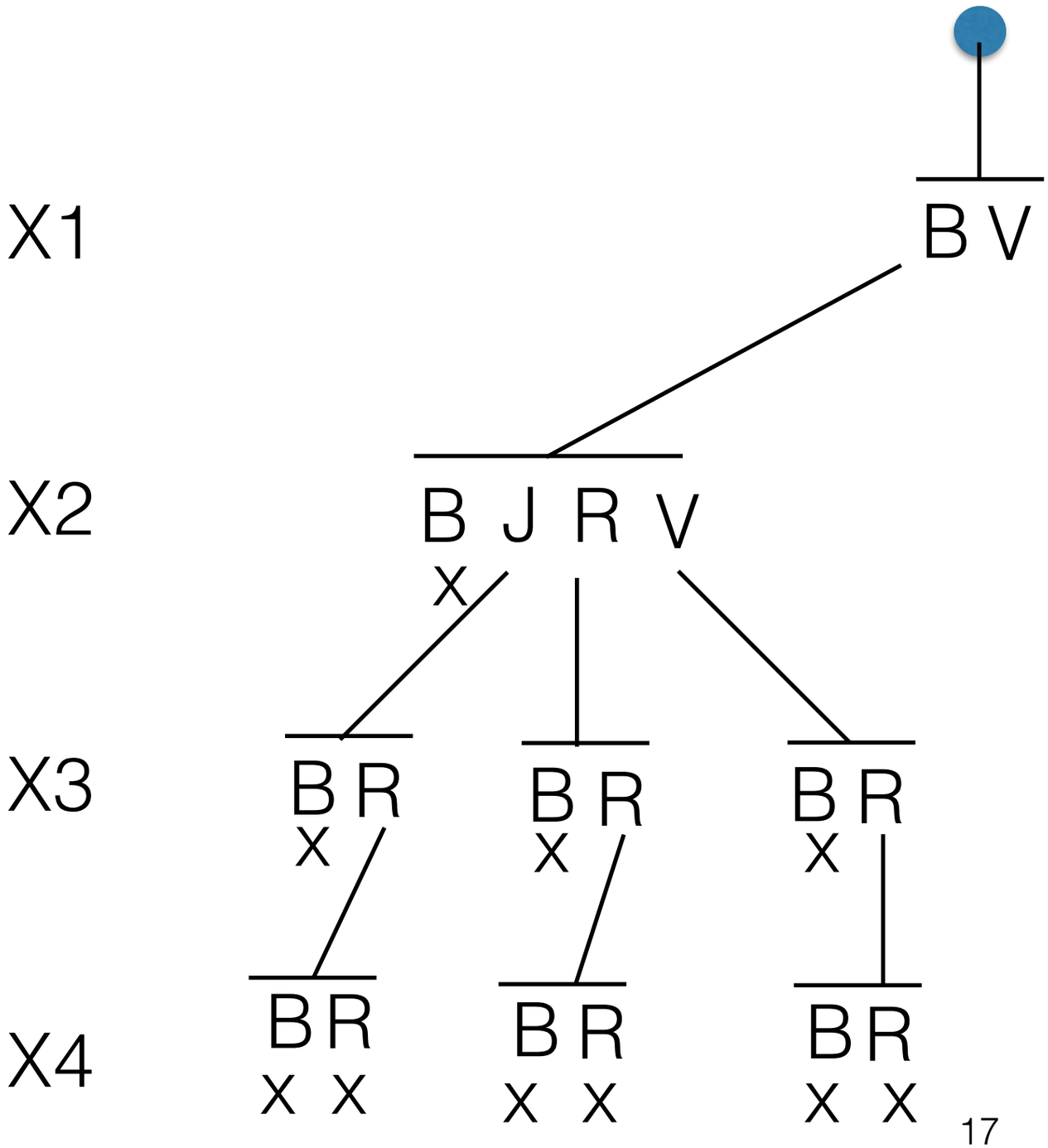
BT on running example



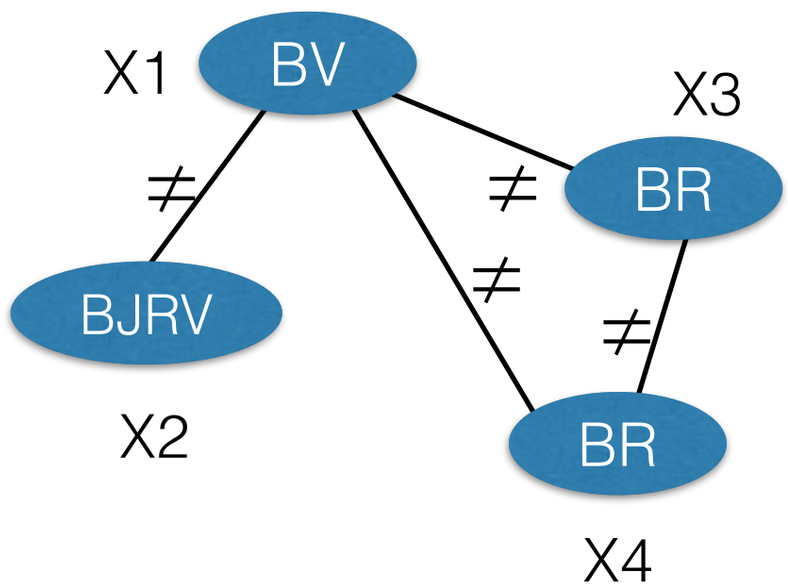

```
function BT(N : problem ; I : instantiation) :Boolean
begin
    if |I| = n then return true
    choose a variable X_i not in I
    for each v_i ∈ D(X_i) do
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```



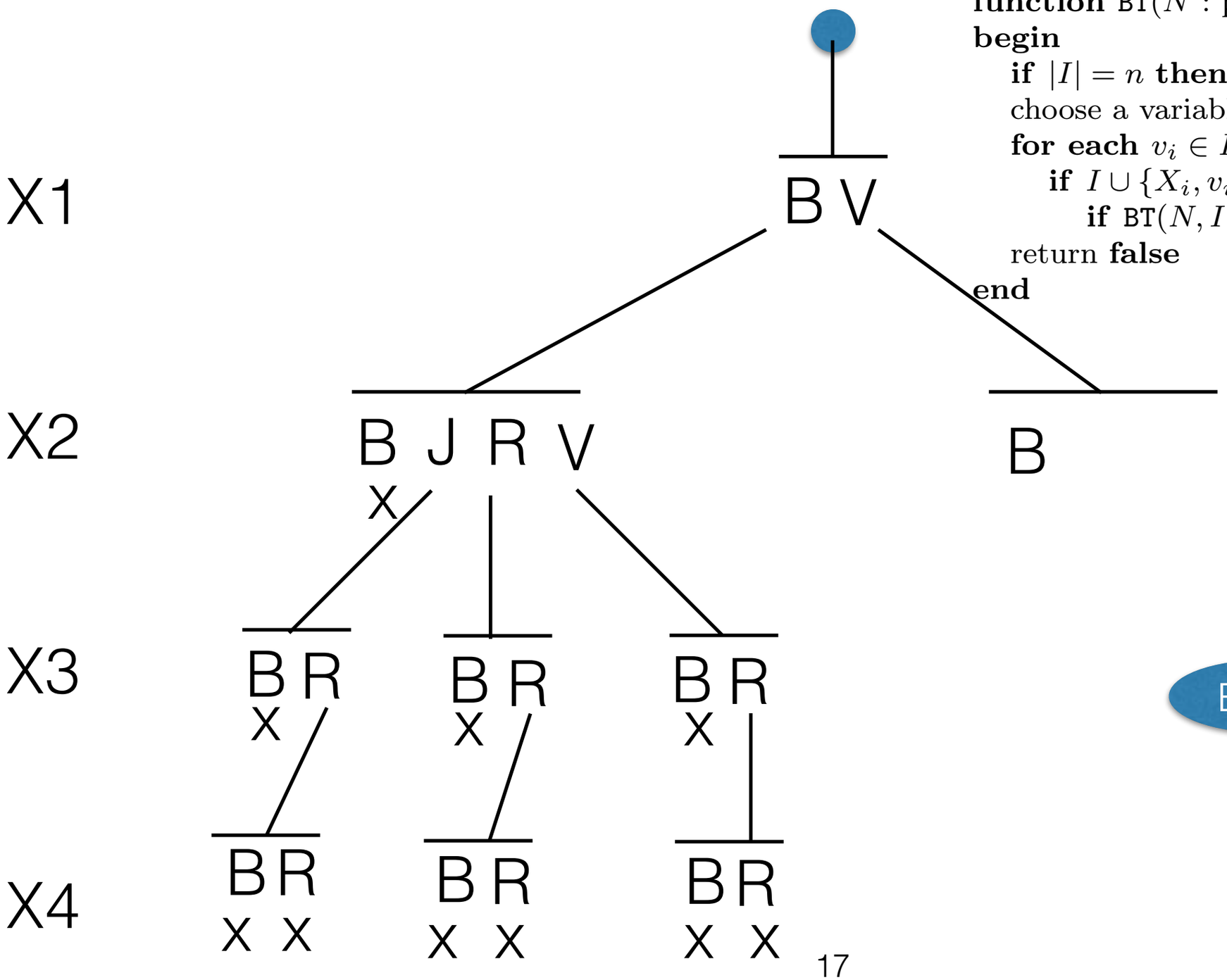
BT on running example



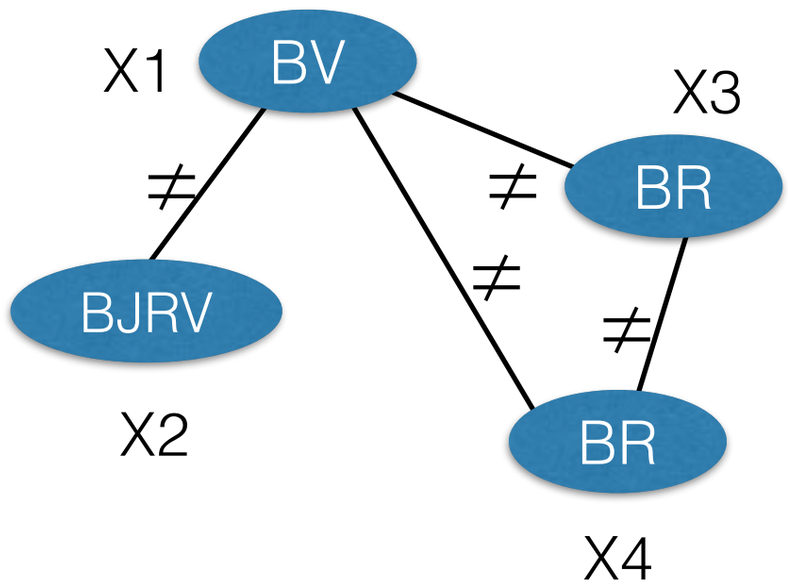
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



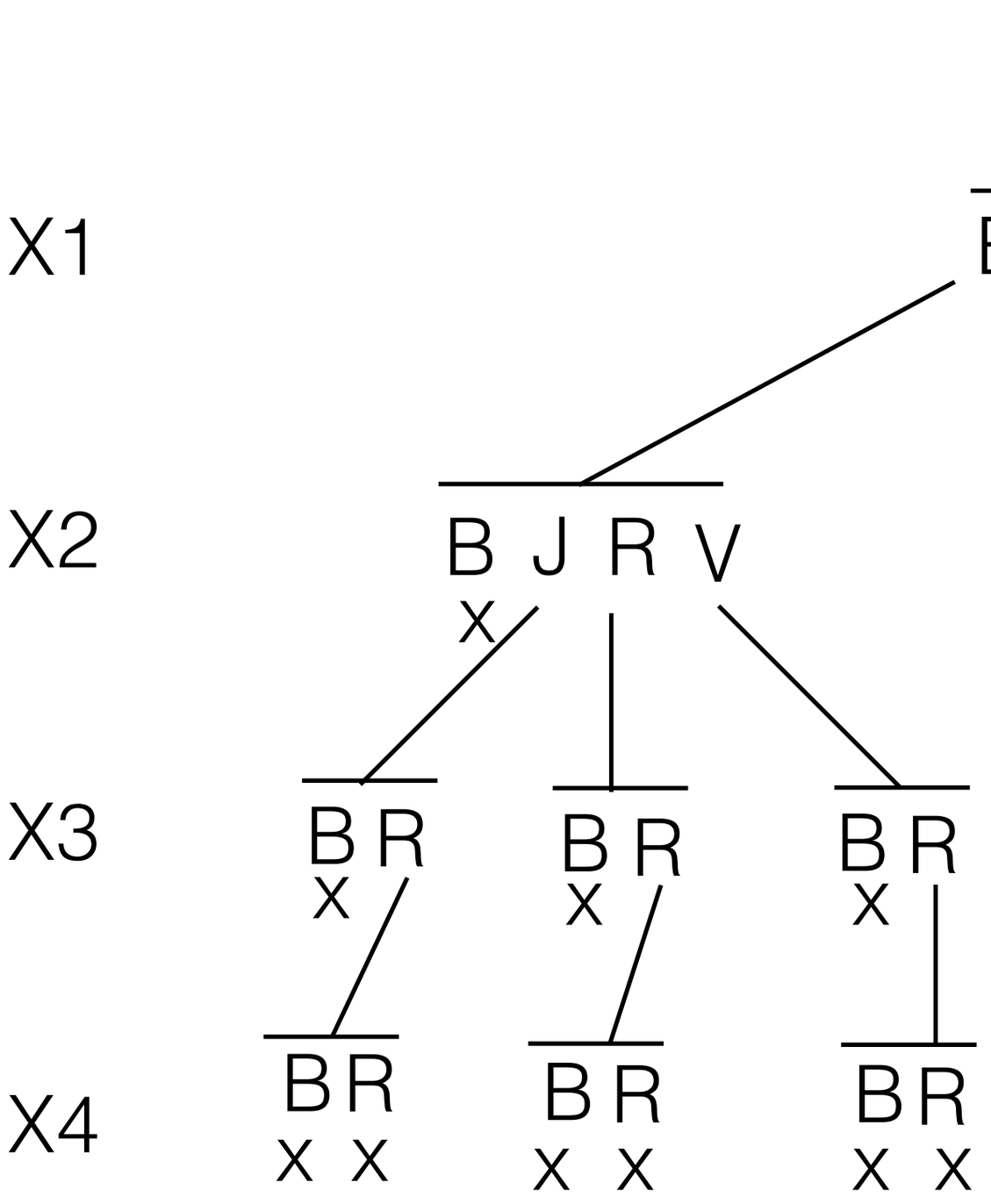
BT on running example



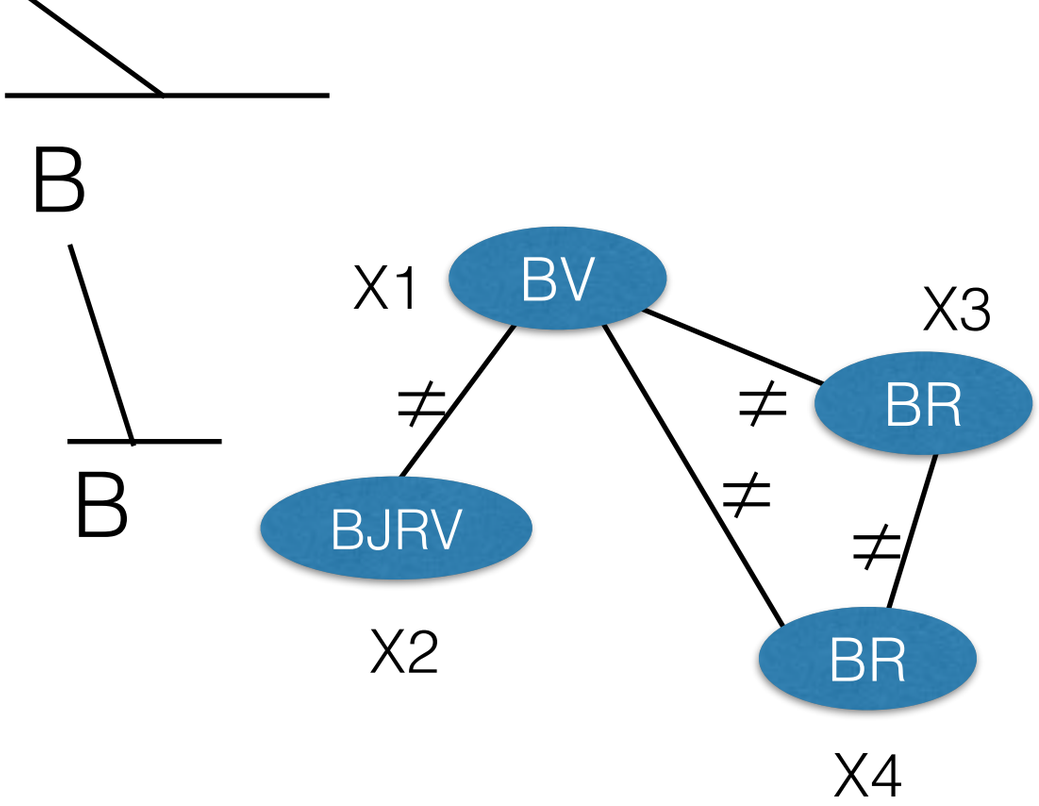
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



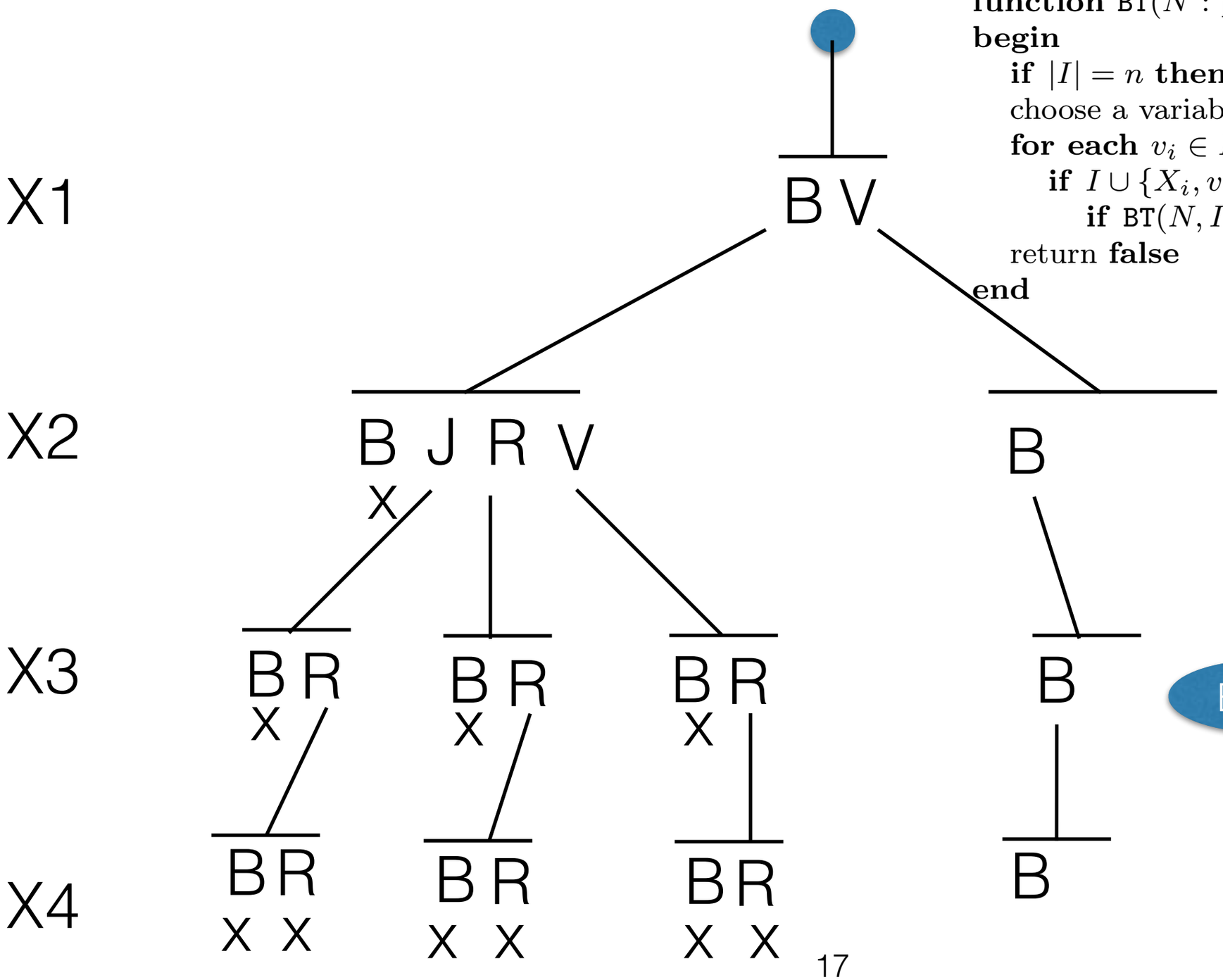
BT on running example



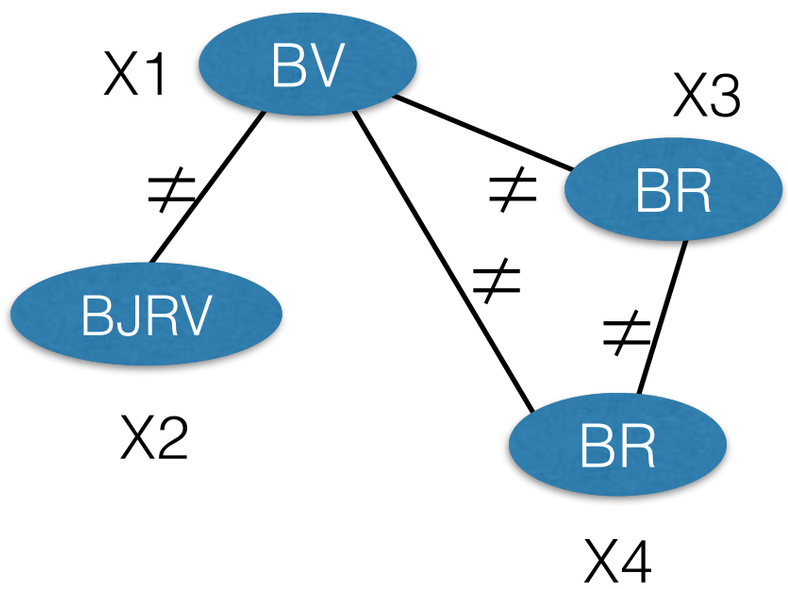
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



BT on running example

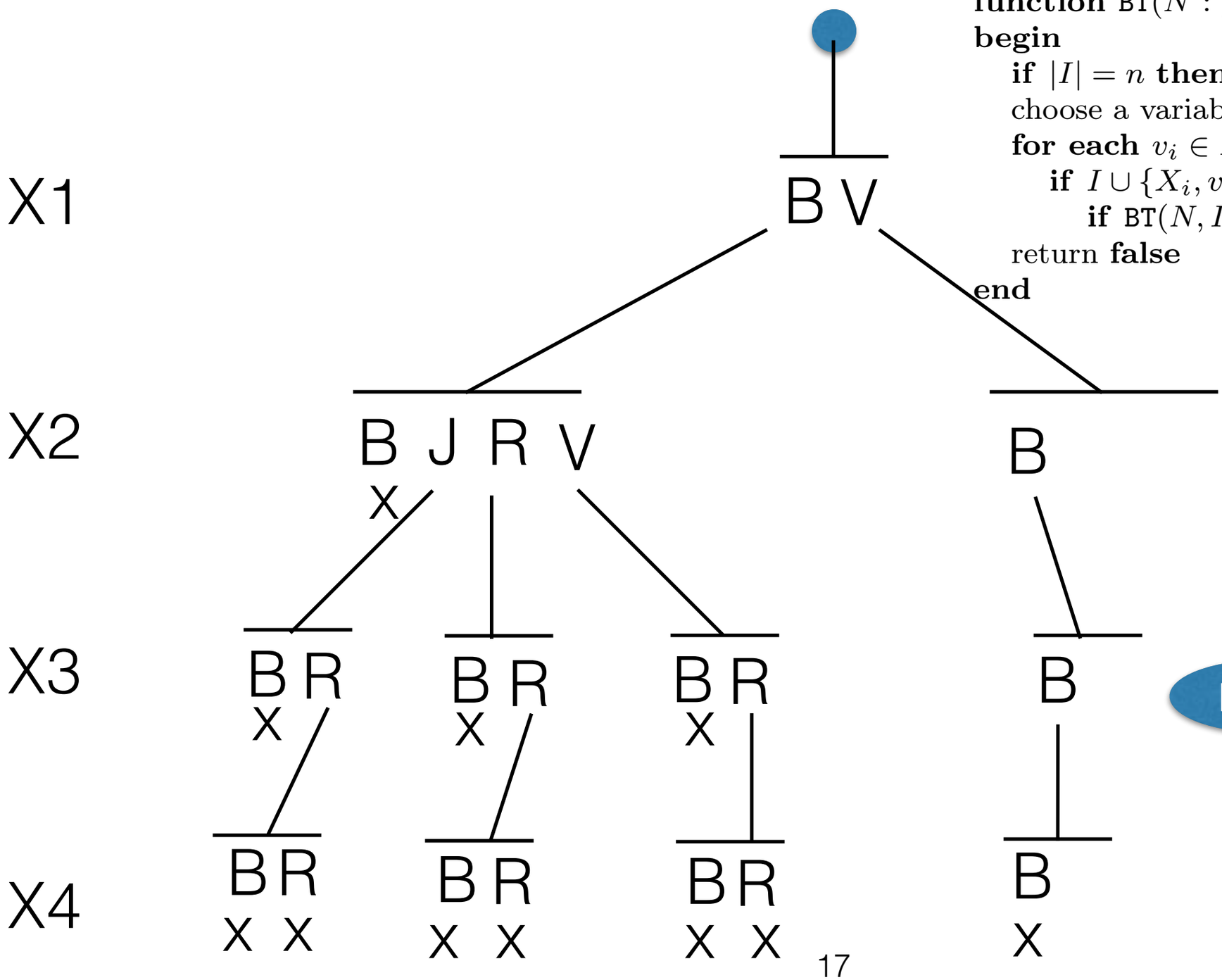


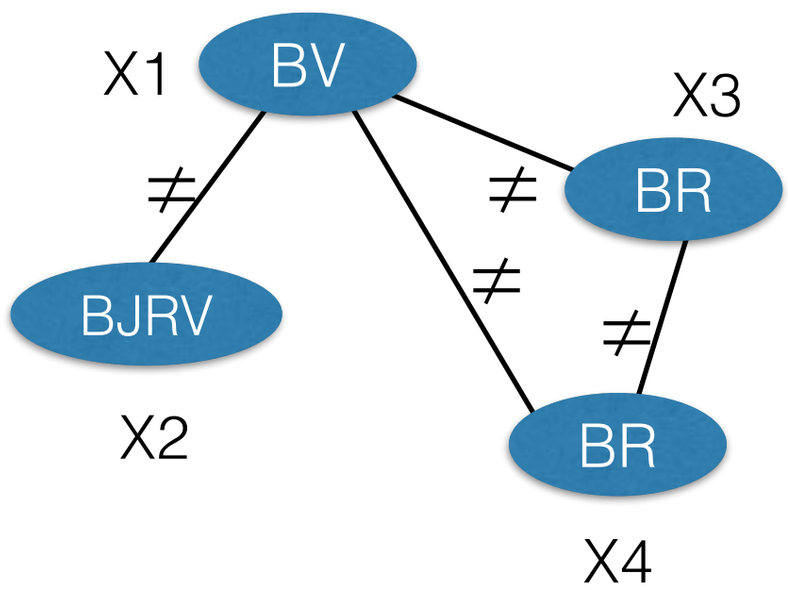
function BT(N : problem ; I : instantiation) :**Boolean**
begin
  if $|I| = n$ **then** return **true**
  choose a variable X_i not in I
  for each $v_i \in D(X_i)$ **do**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



BT on running example

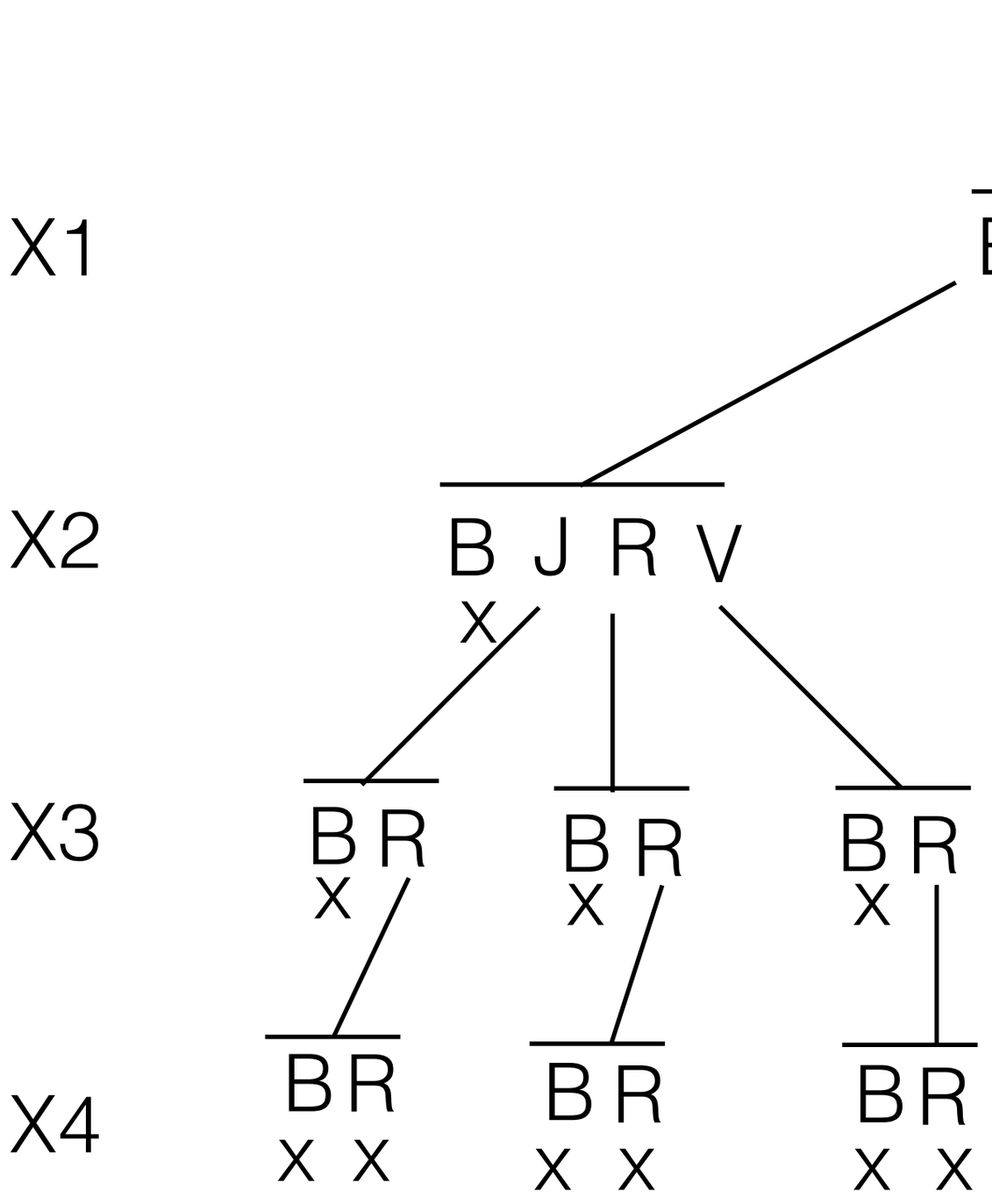
```
function BT(N : problem ; I : instantiation) :Boolean
begin
    if |I| = n then return true
    choose a variable X_i not in I
    for each v_i ∈ D(X_i) do
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```

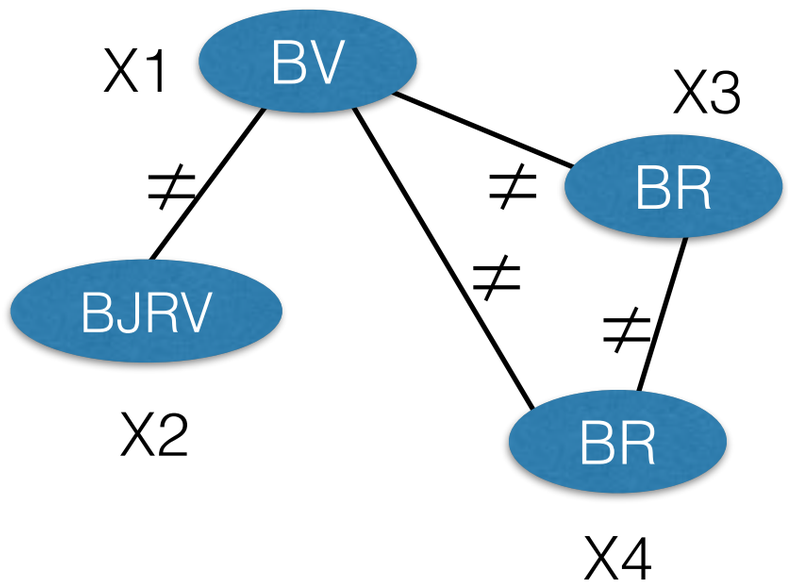




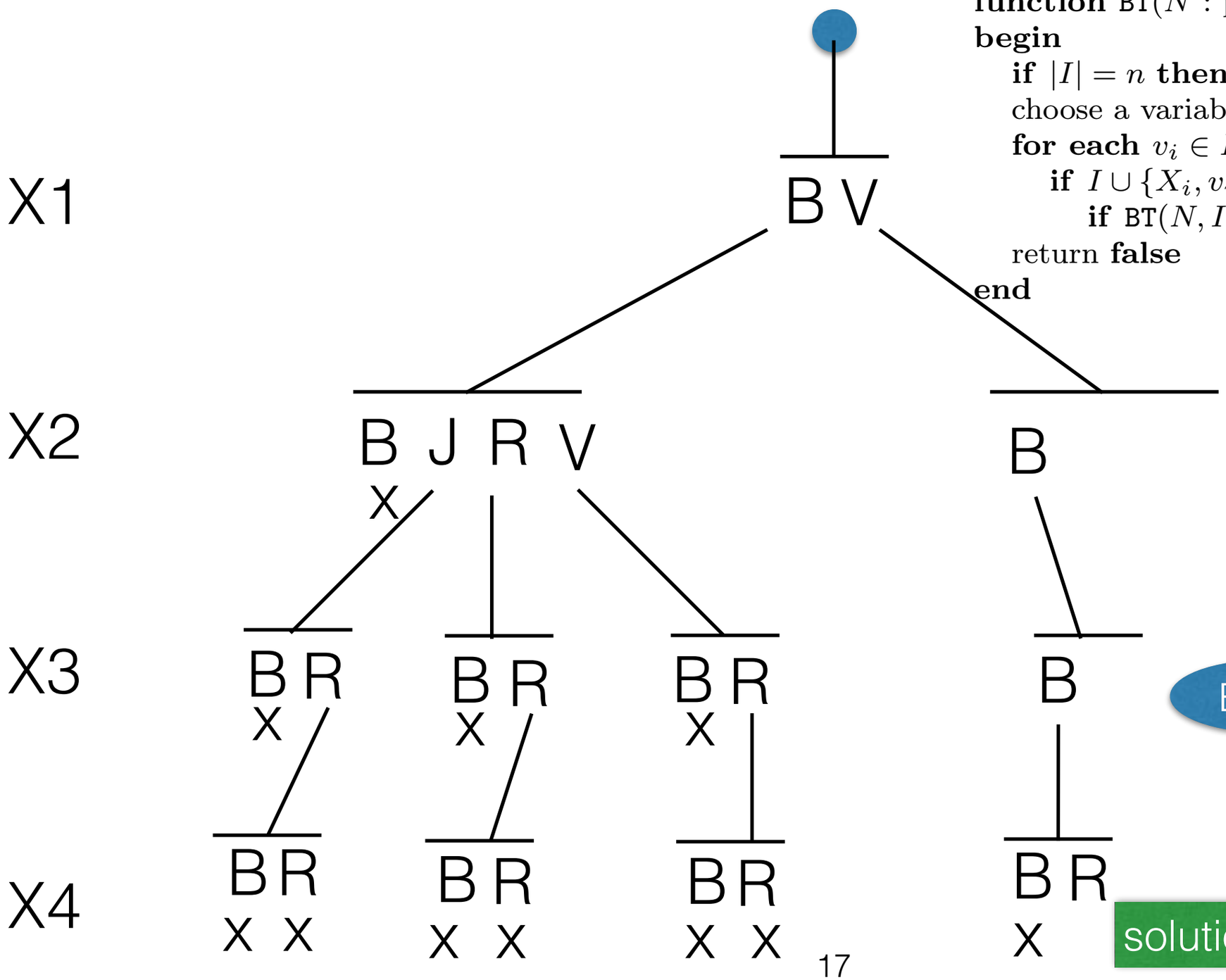
BT on running example

```
function BT(N : problem ; 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
        if I ∪ {X_i, v_i} locally consistent then
            if BT(N, I ∪ {X_i, v_i}) then return true
    return false
end
```

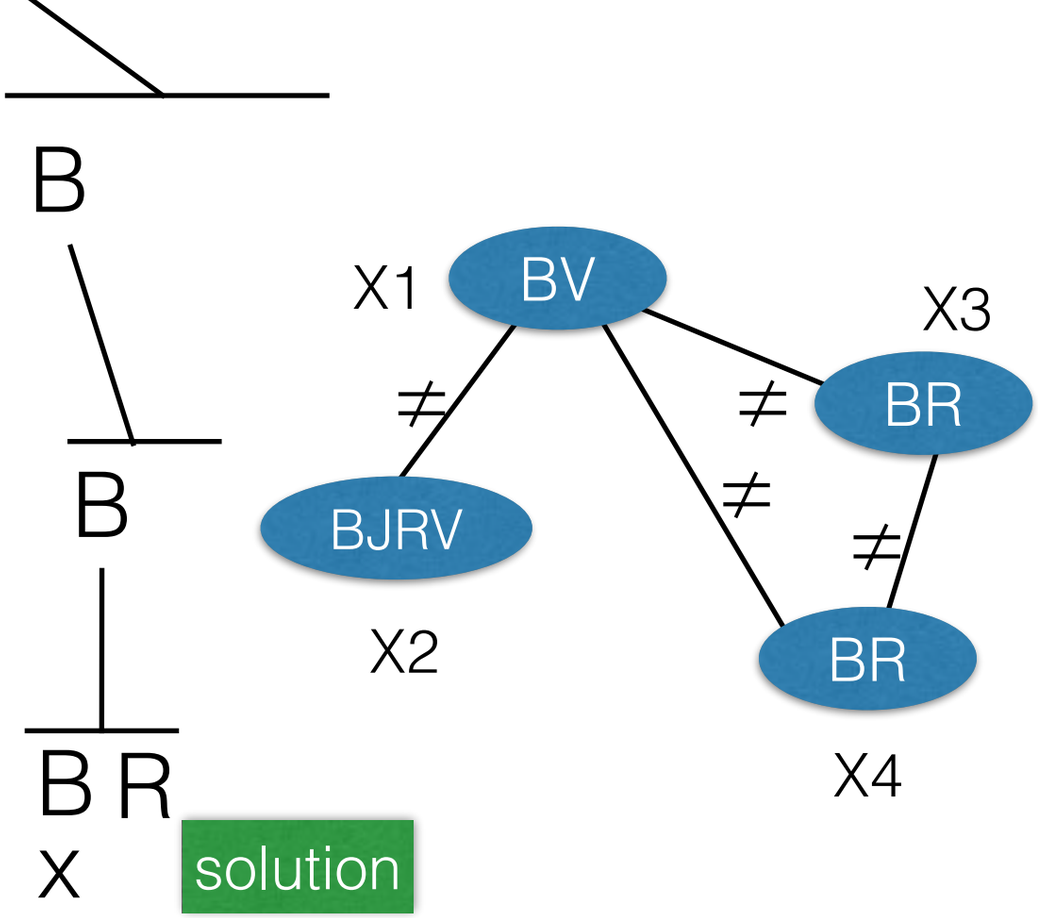




BT on running example



function BT(N : problem ; 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**
    if $I \cup \{X_i, v_i\}$ locally consistent **then**
      if BT($N, I \cup \{X_i, v_i\}$) **then** return **true**
  return **false**
end



Complexity of BT

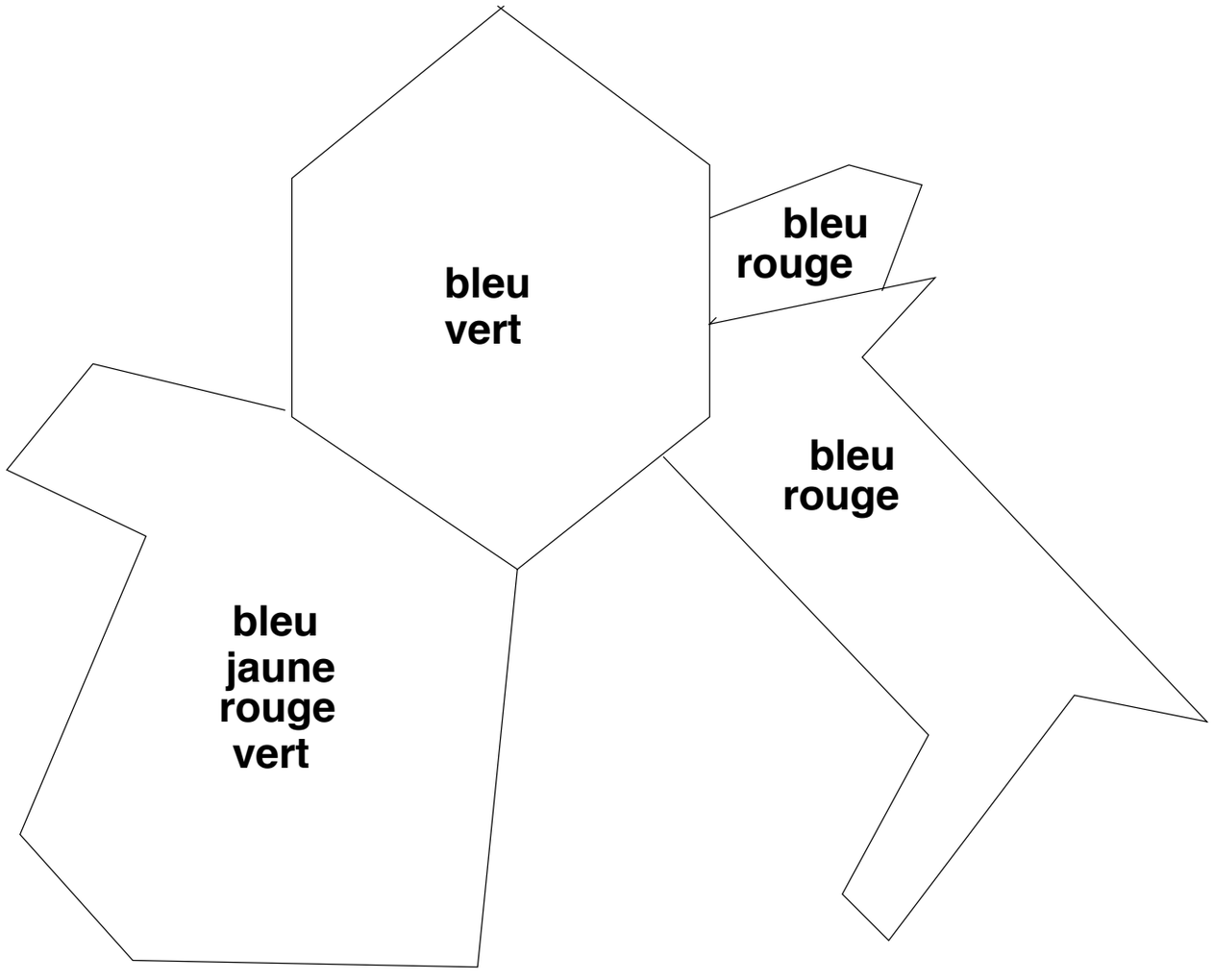
- Space complexity: linear
- Time complexity:

  $O(ed^n)$, where $n = |X|$, $e = |C|$, and $d = max_{1..n}(D(X_i))$

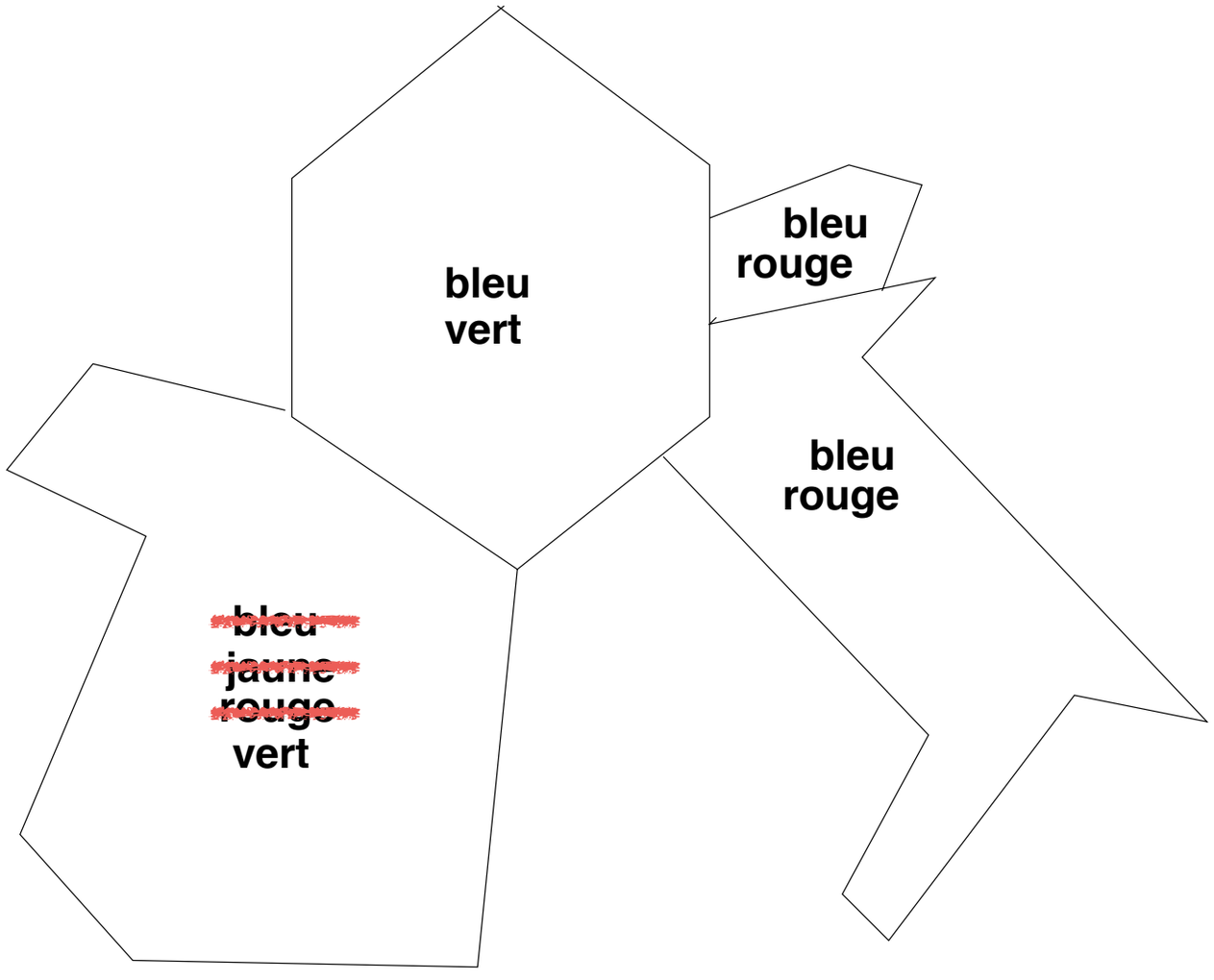
- Repeatedly discovers the same inconsistencies
- Limited to very small problems

Local consistencies (constraint propagation)

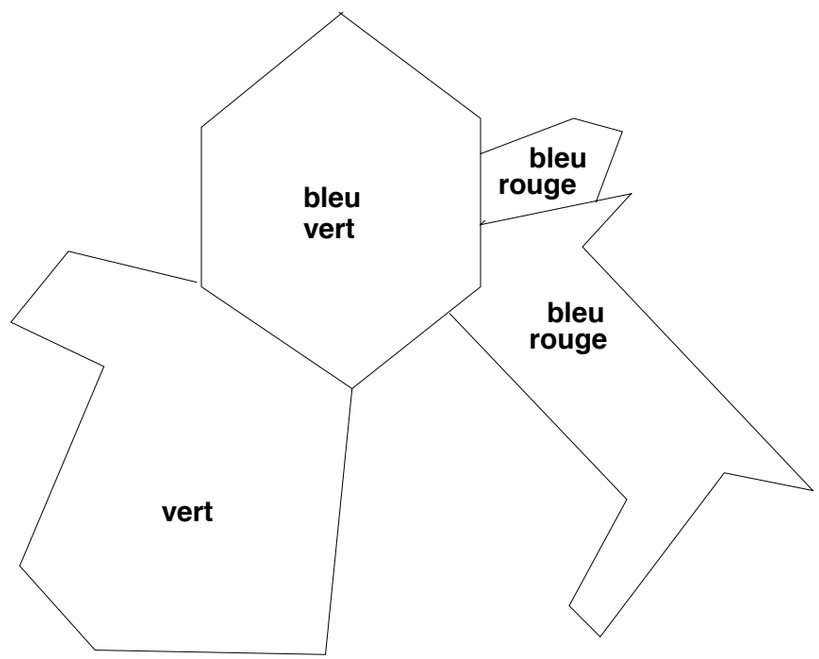
Modified running example



Modified running example



BT



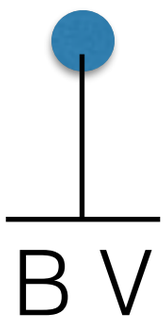
X1

X2

X3

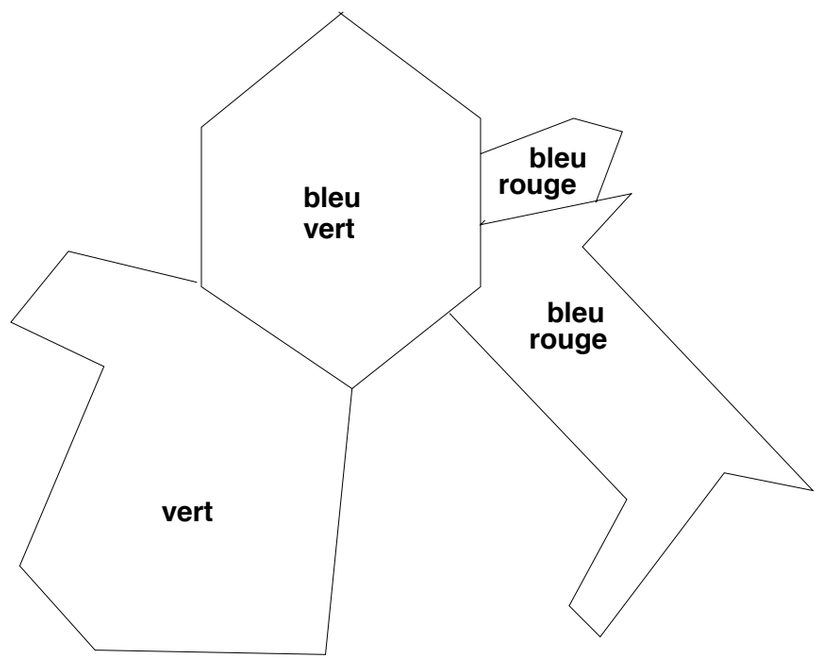
X4

BT



B V

X1

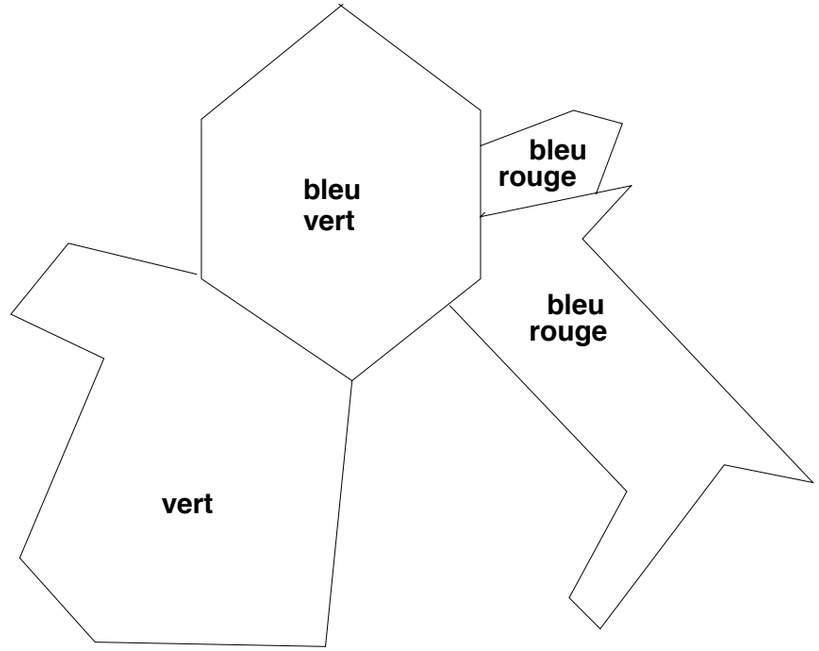


X2

X3

X4

BT



X1

B V

X2

V

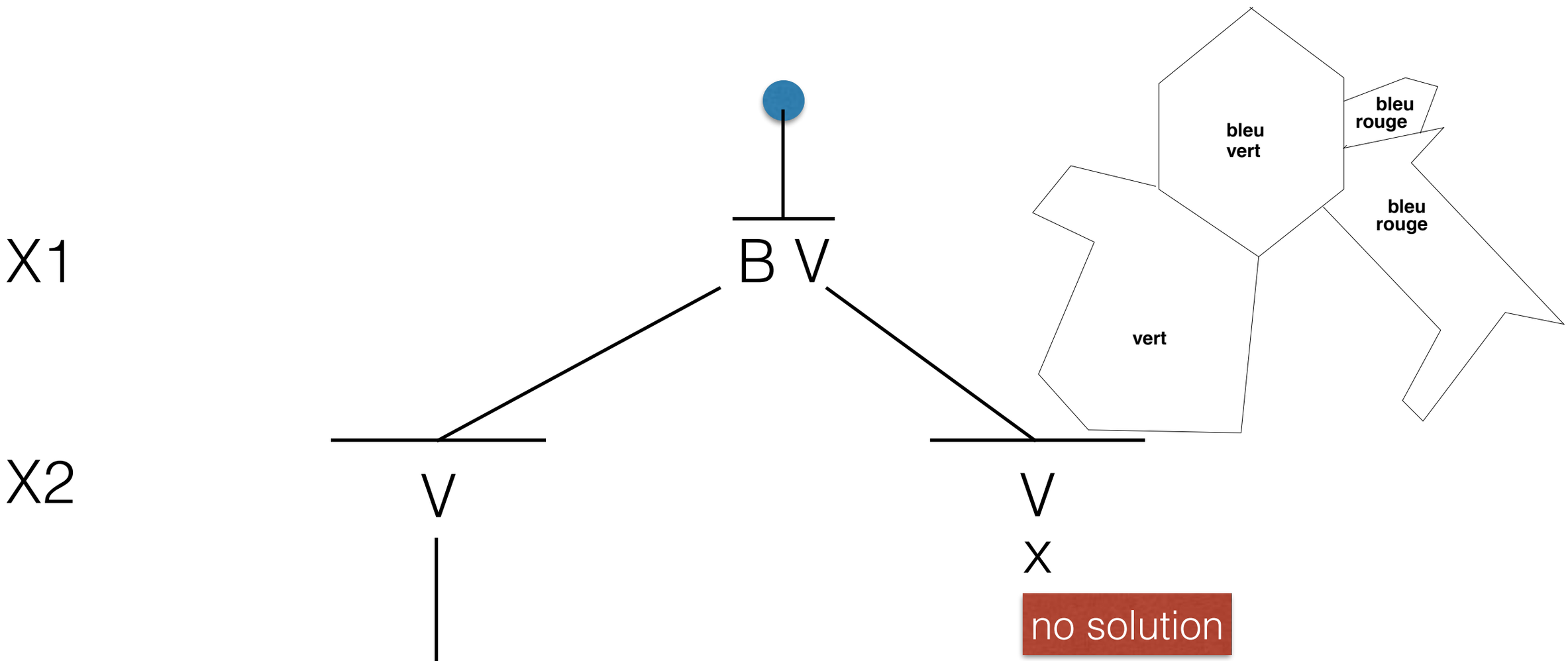
X3

B R
X

X4

B R
X X

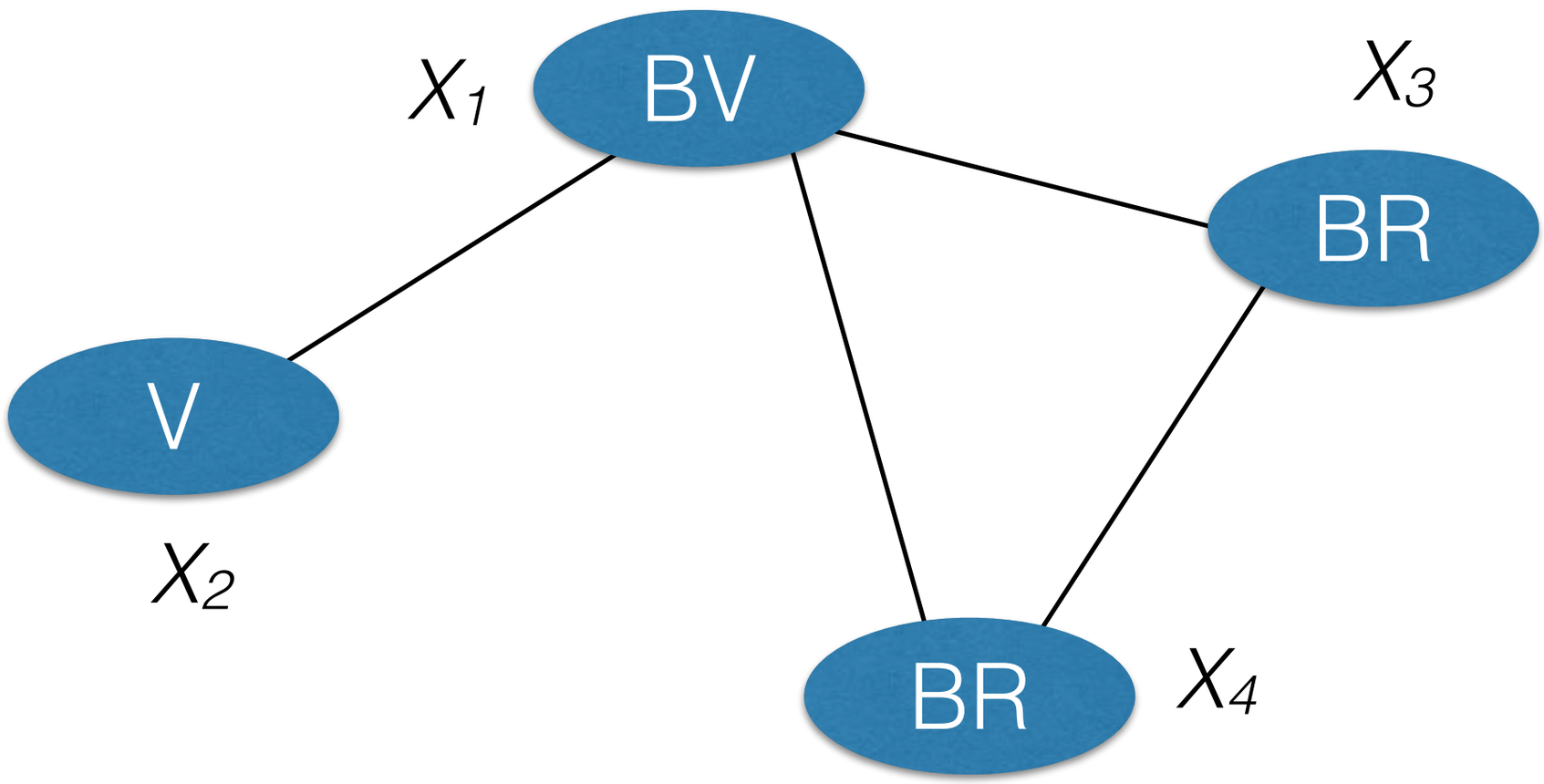
BT


X1
B V
bleu
vert
bleu
rouge
bleu
rouge
vert
X2
V
V
x
no solution
X3
B R
x
X4
B R
x x

Could we detect this inconsistency with a polynomial procedure?

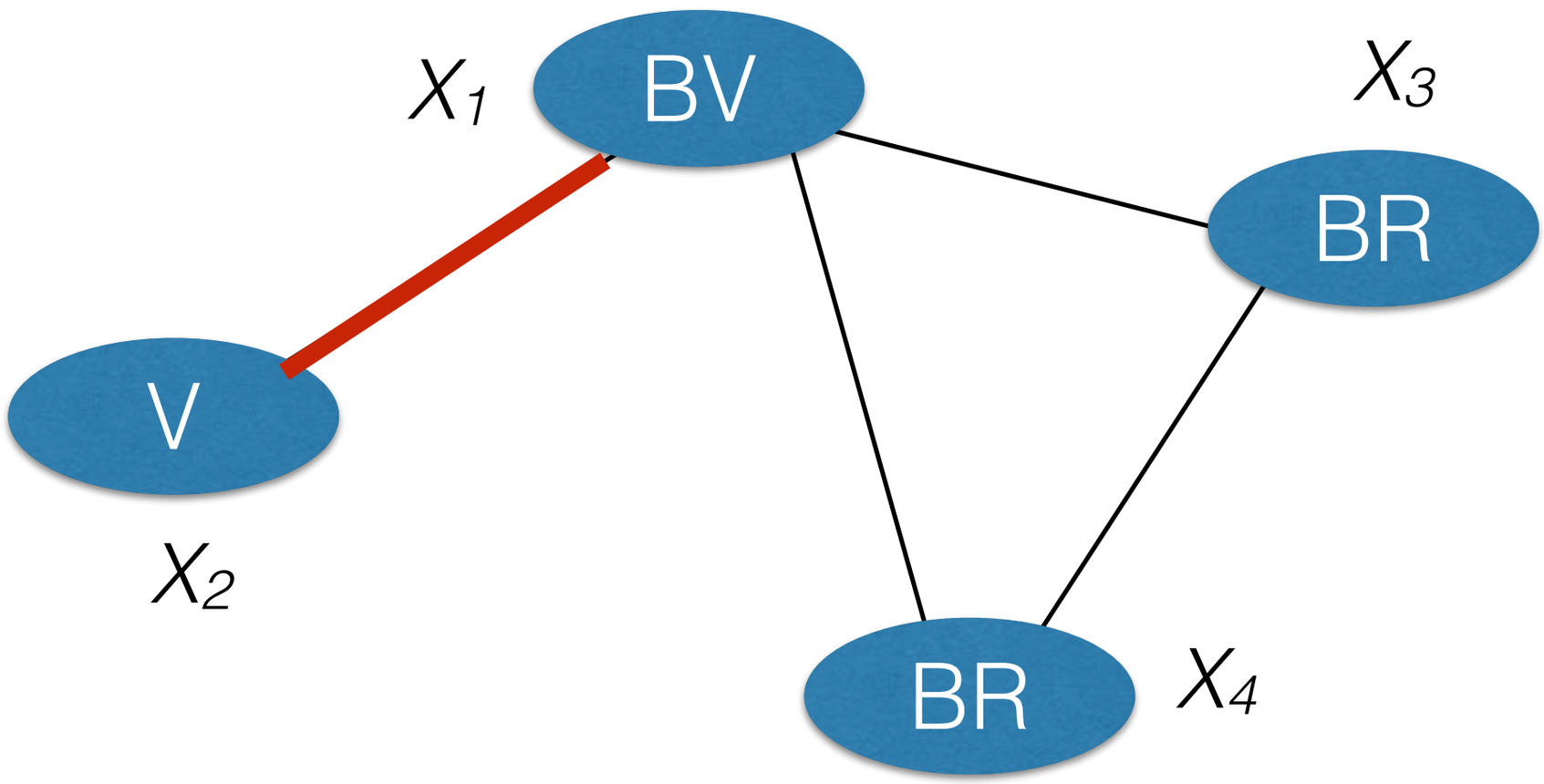
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



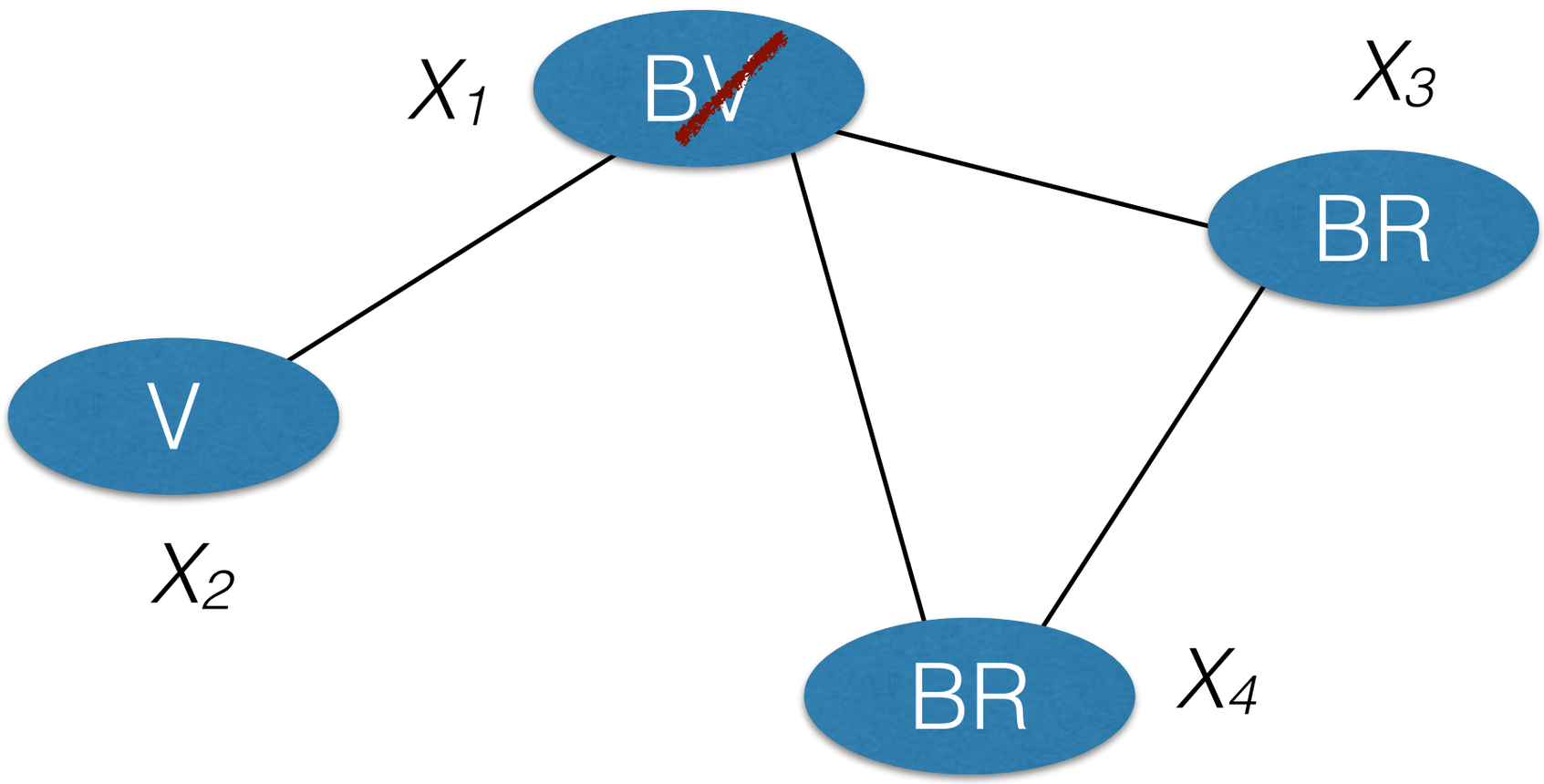
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



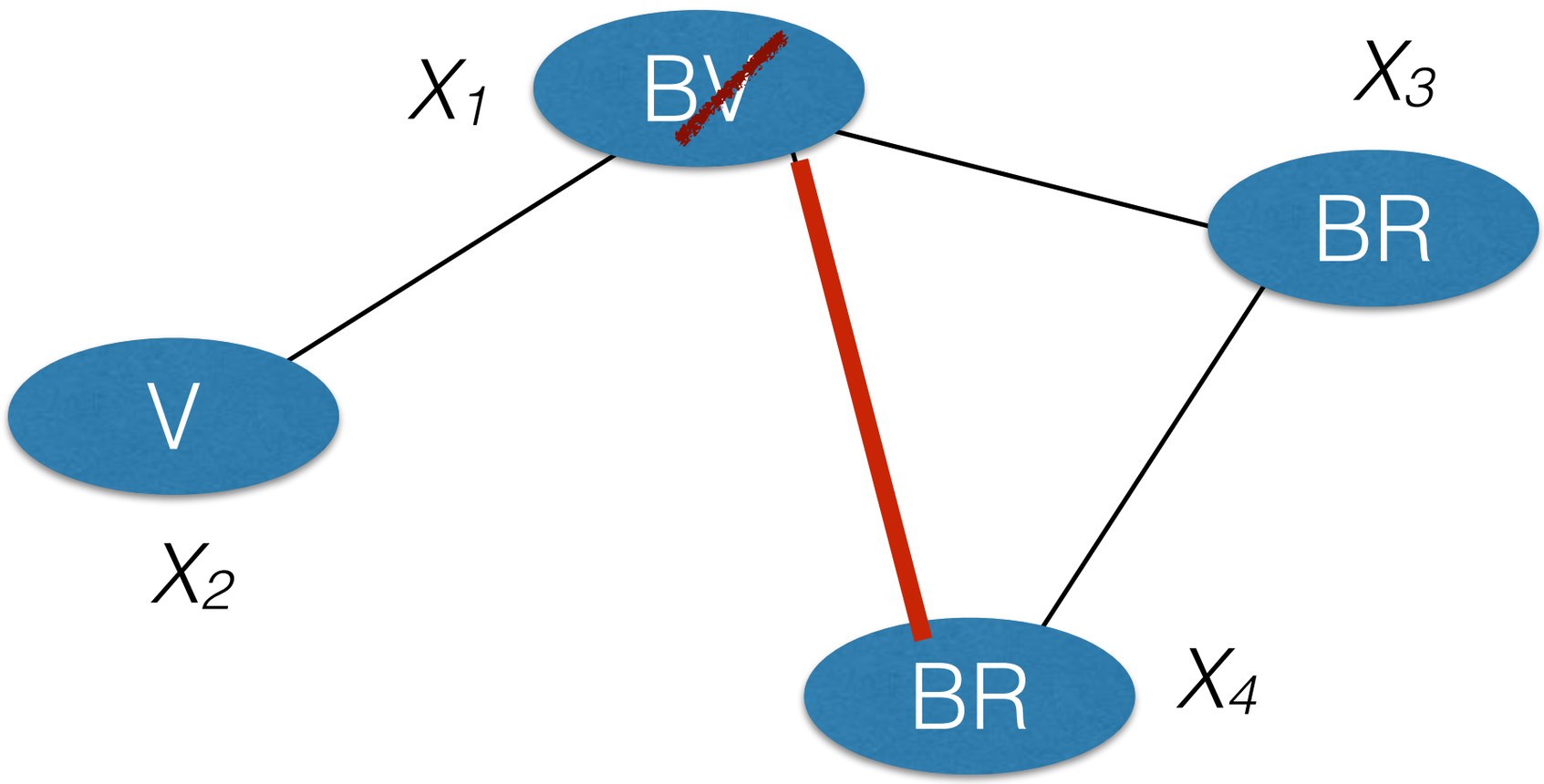
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



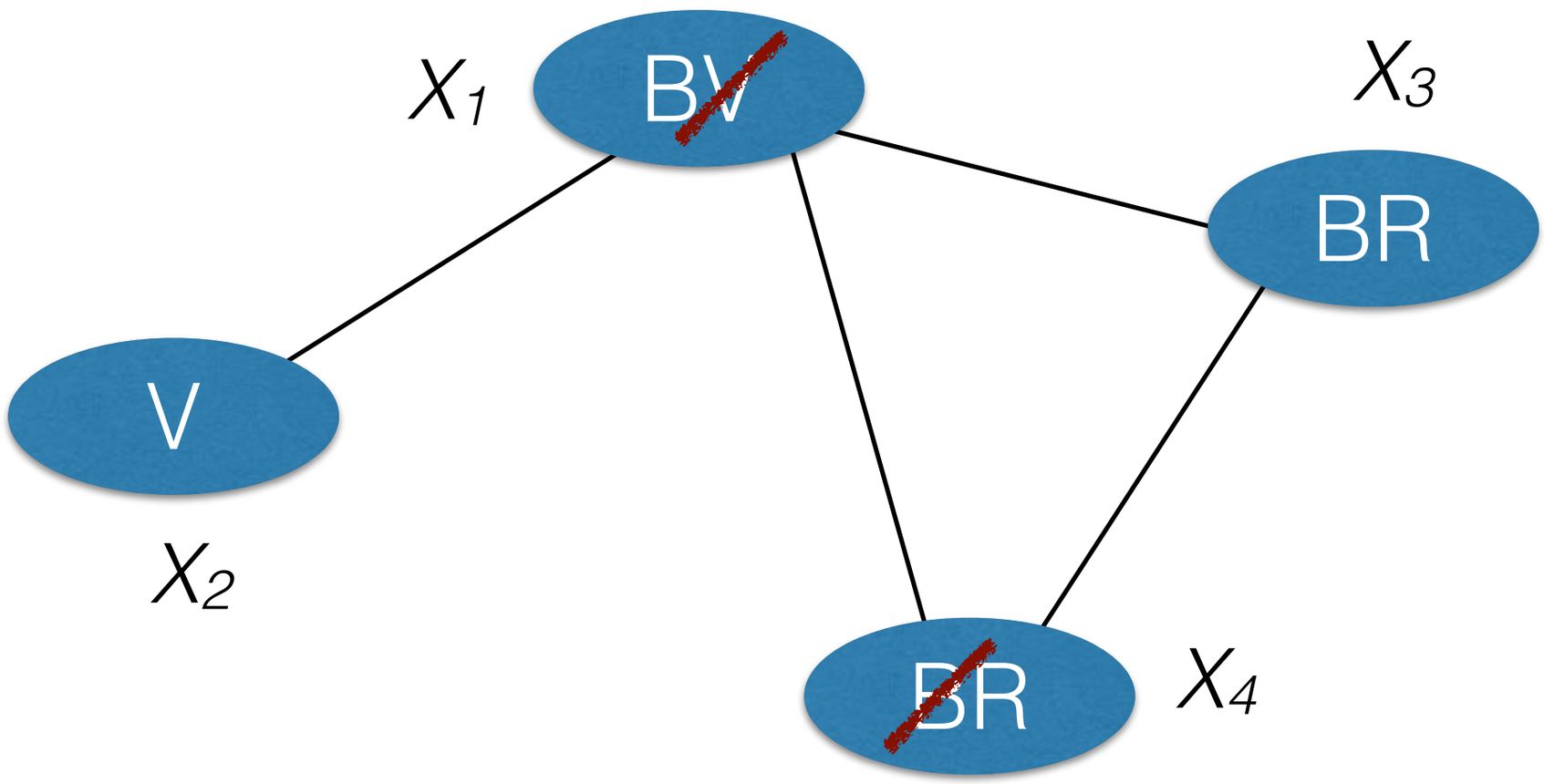
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



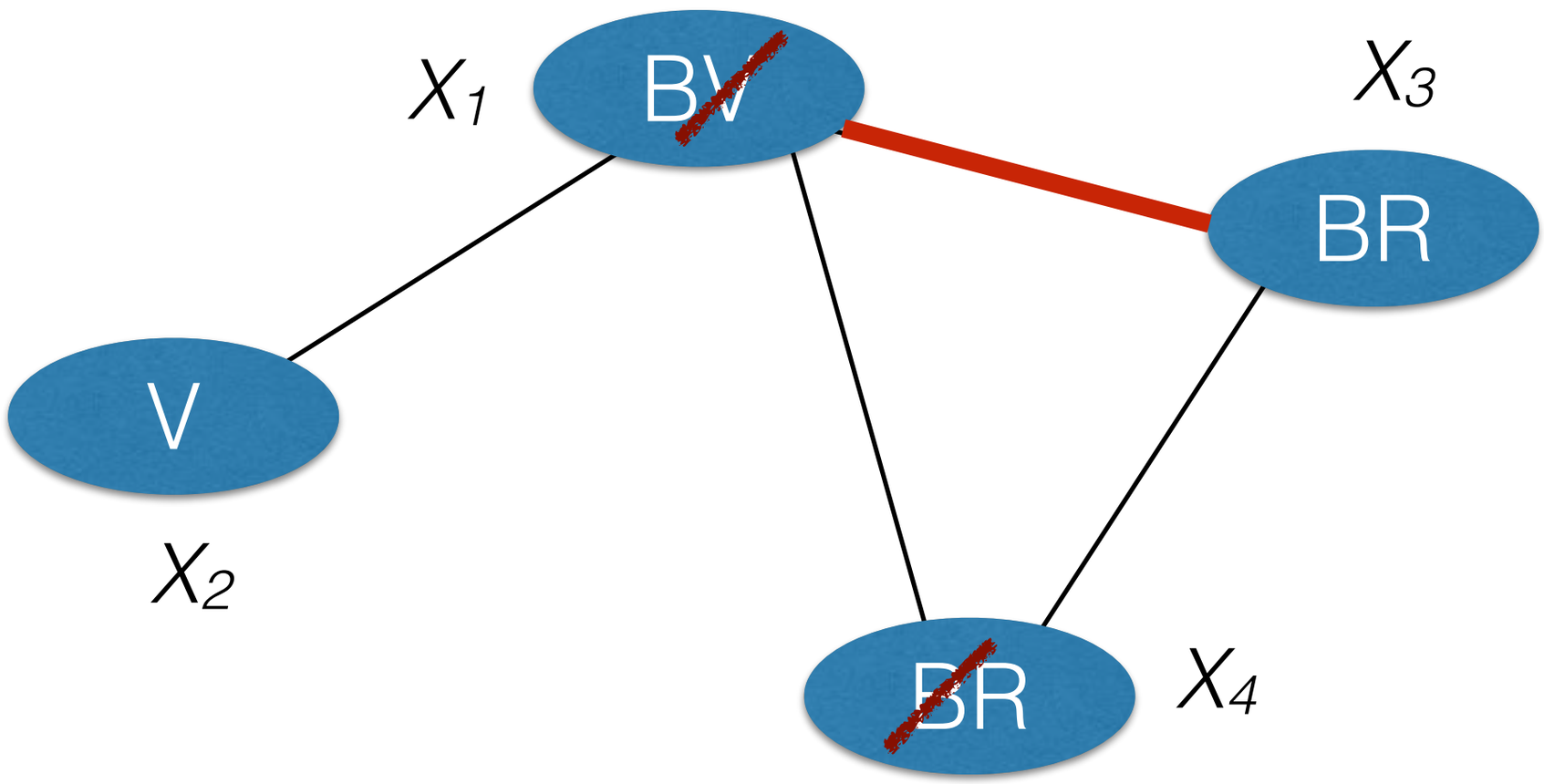
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



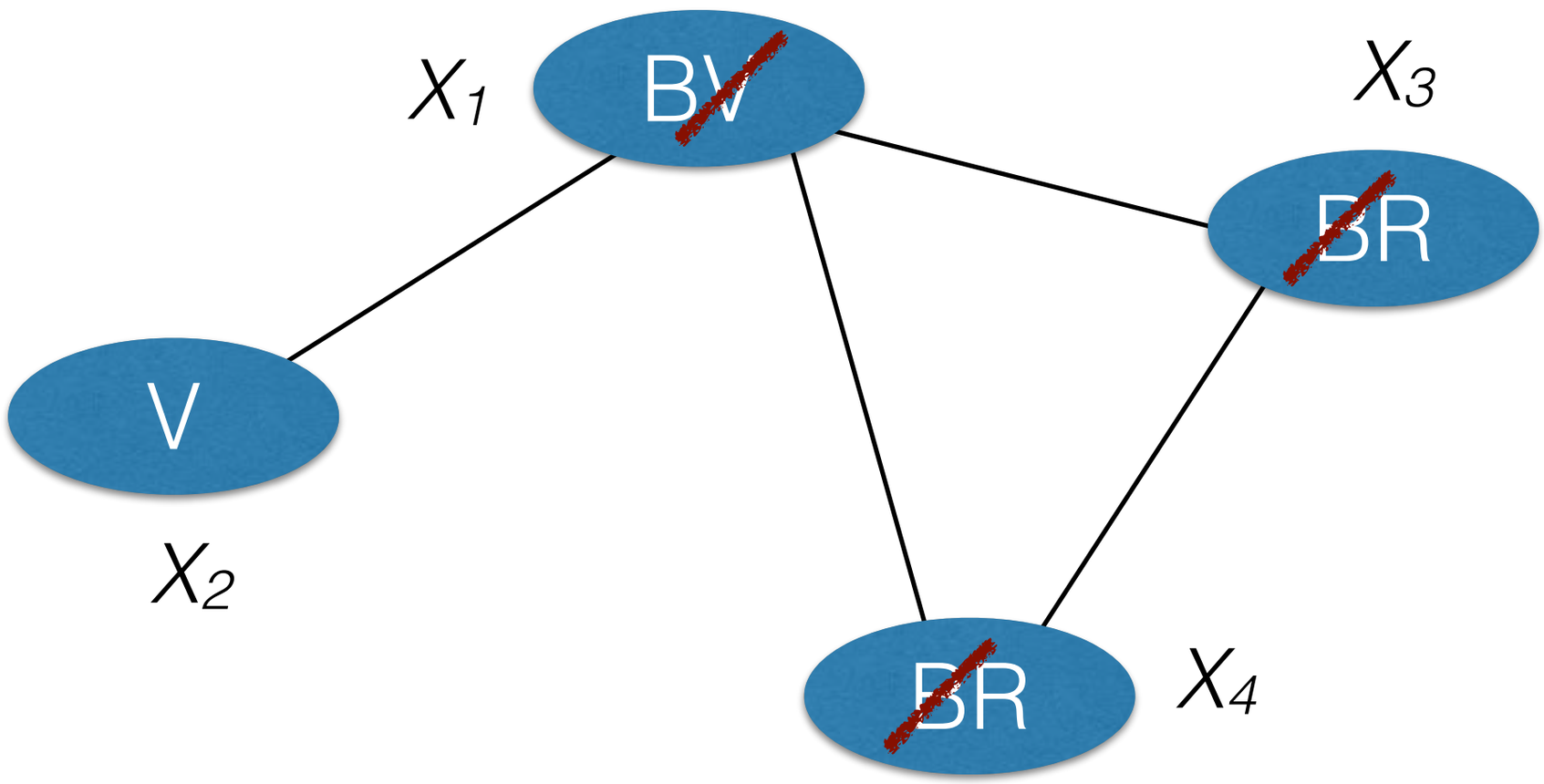
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



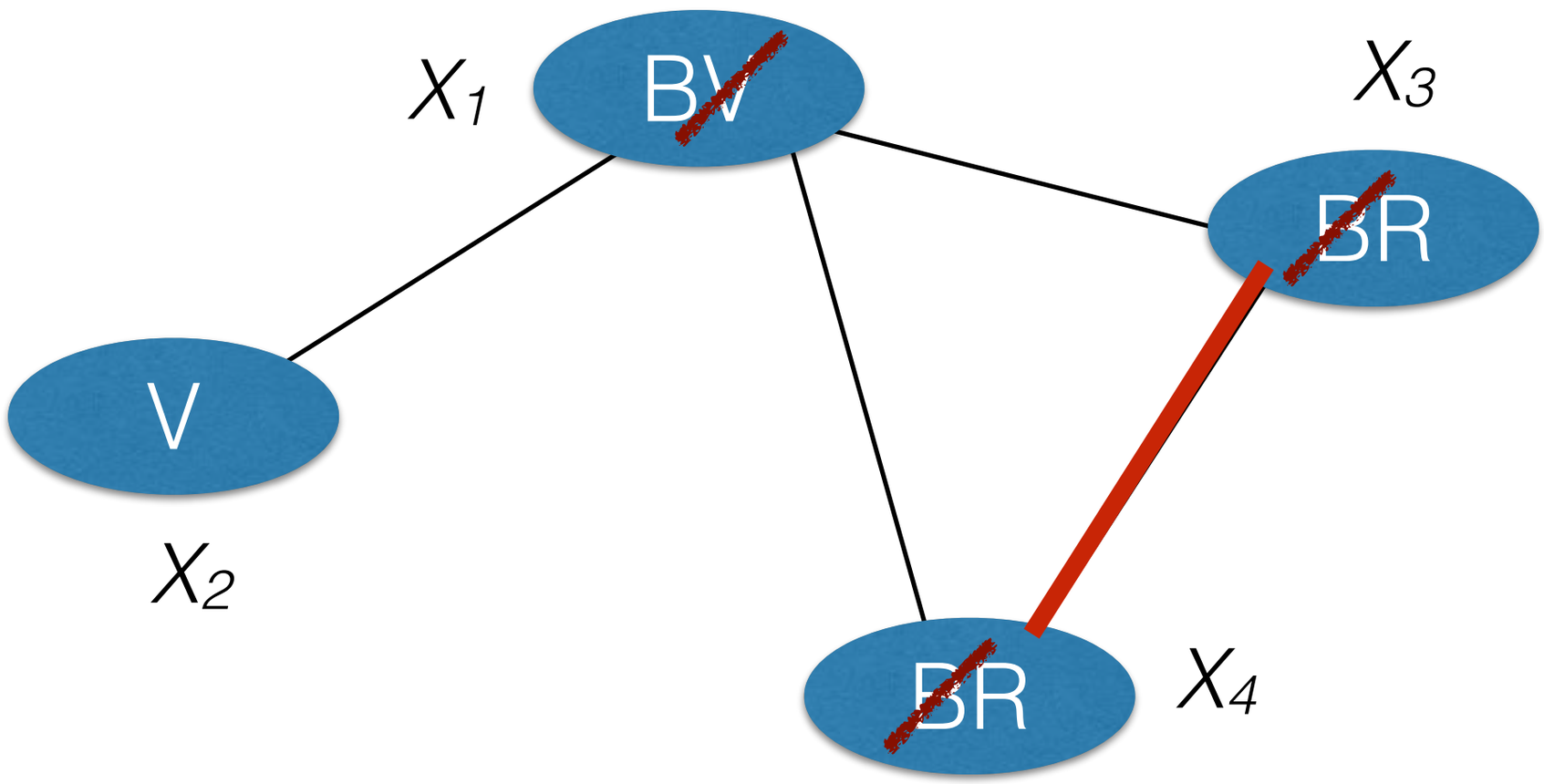
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



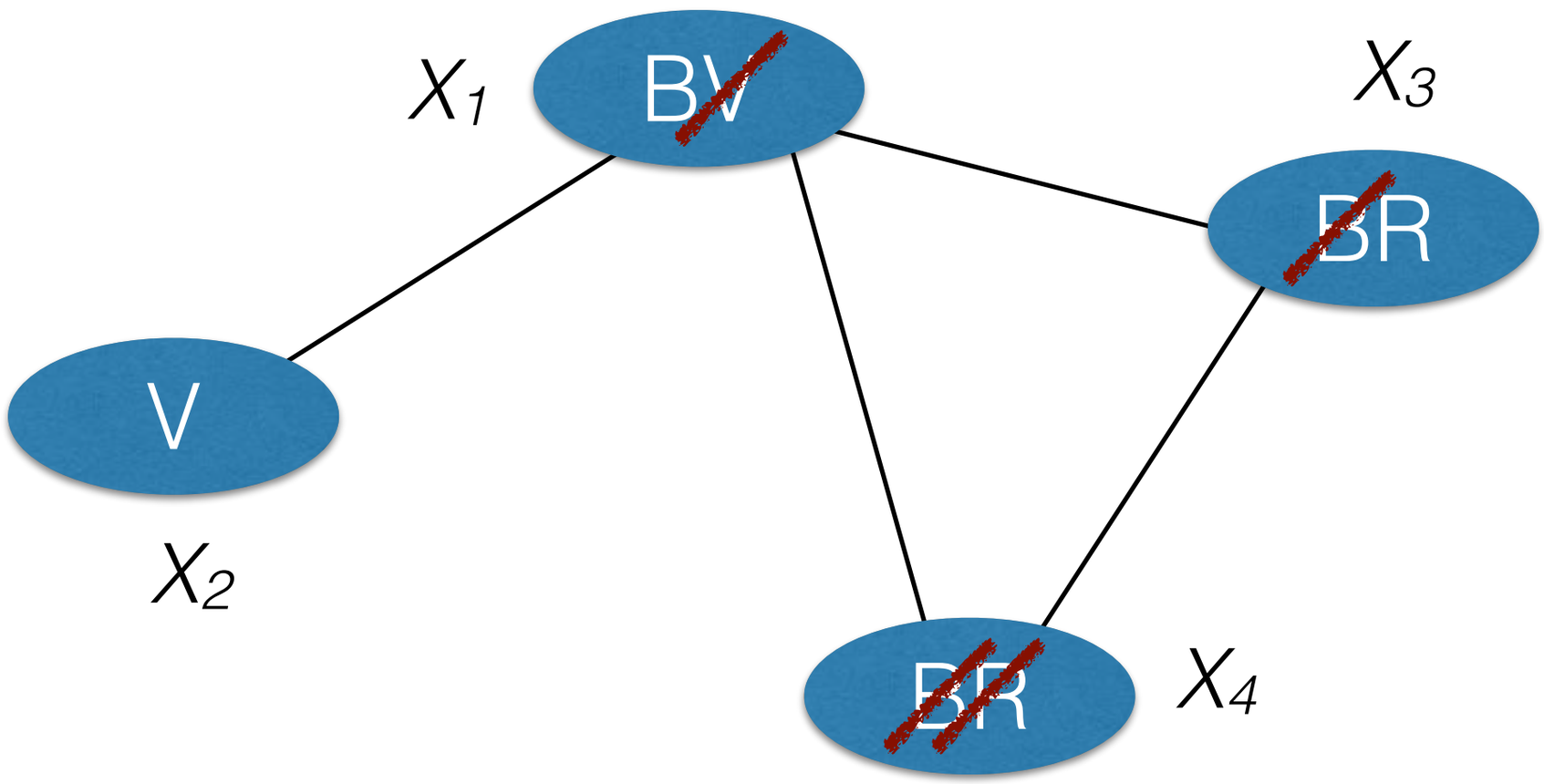
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



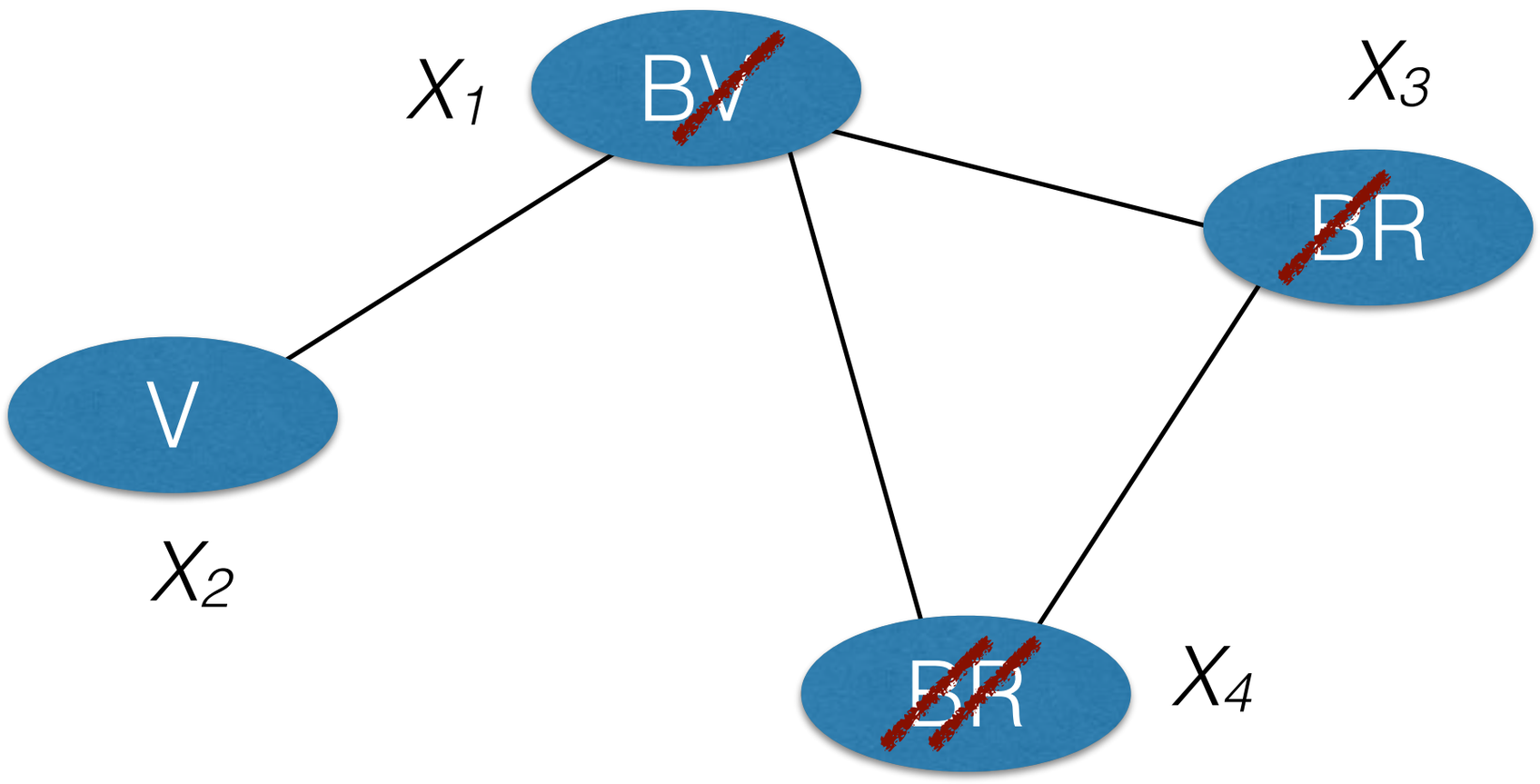
Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



Constraint Propagation

- Eliminate once and for all inconsistencies that BT could reach many times otherwise



empty domain!

Arc consistency

Definition 4 (Arc consistency (AC)) *Let* $N = (X, D, C)$ *a network and* c *a constraint in* C*.*

D: X_1 X_2 X_3

1	1	1
2	2	2
3	3	3
7	7	7
8	8	8

$X_1+X_2=X_3$

1	1	2
1	2	3
2	1	3
1	7	8
7	1	8

Arc consistency

Definition 4 (Arc consistency (AC)) *Let* $N = (X, D, C)$ *a network and* c *a constraint in* C*.*

— *a tuple* $\tau \in c \cap D^{X(c)}$ *is called a* support *on* c

D: X_1 X_2 X_3

1	1	1
2	2	2
3	3	3
7	7	7
8	8	8

$X_1+X_2=X_3$

1	1	2
1	2	3
2	1	3
1	7	8
7	1	8

Arc consistency

Definition 4 (Arc consistency (AC)) *Let* $N = (X, D, C)$ *a network and* c *a constraint in* C*.*

- *a tuple* $\tau \in c \cap D^{X(c)}$ *is called a* support *on* c
- *a value* $v_i \in D(X_i)$ *is* viable *iff* $\forall c \in C, X_i \in X(c), \exists$ *a support* τ *on* c *such that* $\tau[X_i] = v_i$

D: X_1 X_2 X_3

1	1	1
2	2	2
3	3	3
7	7	7
8	8	8

$X_1 + X_2 = X_3$

1	1	2
1	2	3
2	1	3
1	7	8
7	1	8

Arc consistency

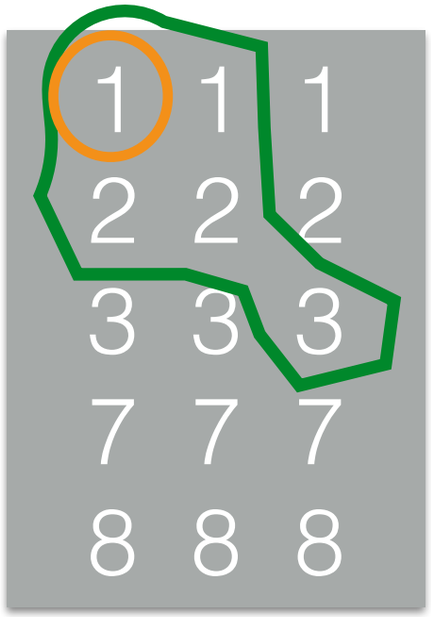
Definition 4 (Arc consistency (AC)) *Let* $N = (X, D, C)$ *a network and* c *a constraint in* C*.*

— *a tuple* $\tau \in c \cap D^{X(c)}$ *is called a* support *on* c

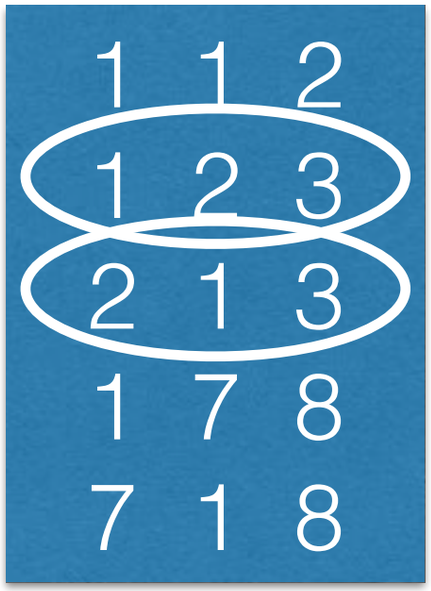
— *a value* $v_i \in D(X_i)$ *is* viable *iff* $\forall c \in C, X_i \in X(c), \exists$ *a support* τ *on* c *such that* $\tau[X_i] = v_i$

— *the domain* D *is* arc consistent *iff* $\forall X_i \in X, \forall v_i \in D(X_i)$, v_i *is viable*

D: X_1 X_2 X_3



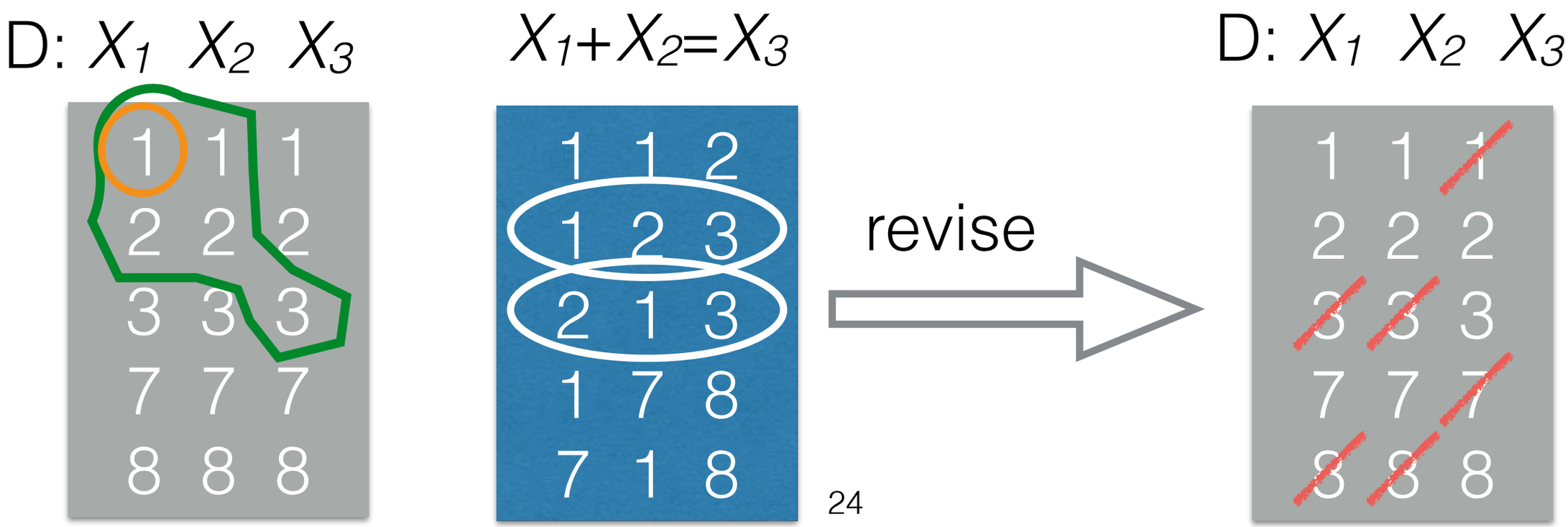
$X_1+X_2=X_3$



Arc consistency

Definition 4 (Arc consistency (AC)) *Let $N = (X, D, C)$ a network and c a constraint in C.*

- *a tuple $\tau \in c \cap D^{X(c)}$ is called a* support *on c*
- *a value $v_i \in D(X_i)$ is* viable *iff $\forall c \in C, X_i \in X(c), \exists$ a support τ on c such that $\tau[X_i] = v_i$*
- *the domain D is* arc consistent *iff $\forall X_i \in X, \forall v_i \in D(X_i)$, v_i is viable*
- *the* AC-closure *of D is the domain D_{AC} which is obtained by iteratively removing non viable values from D until no more exists*



Arc consistency

★ **Important observations**

- D_{AC} is arc consistent
- $sol(X,D,C) = sol(X,D_{AC},C)$
- D_{AC} is the union of all subdomains of D that arc consistent (this is the non-operational definition of AC)

Long history of AC algorithms since 1970:

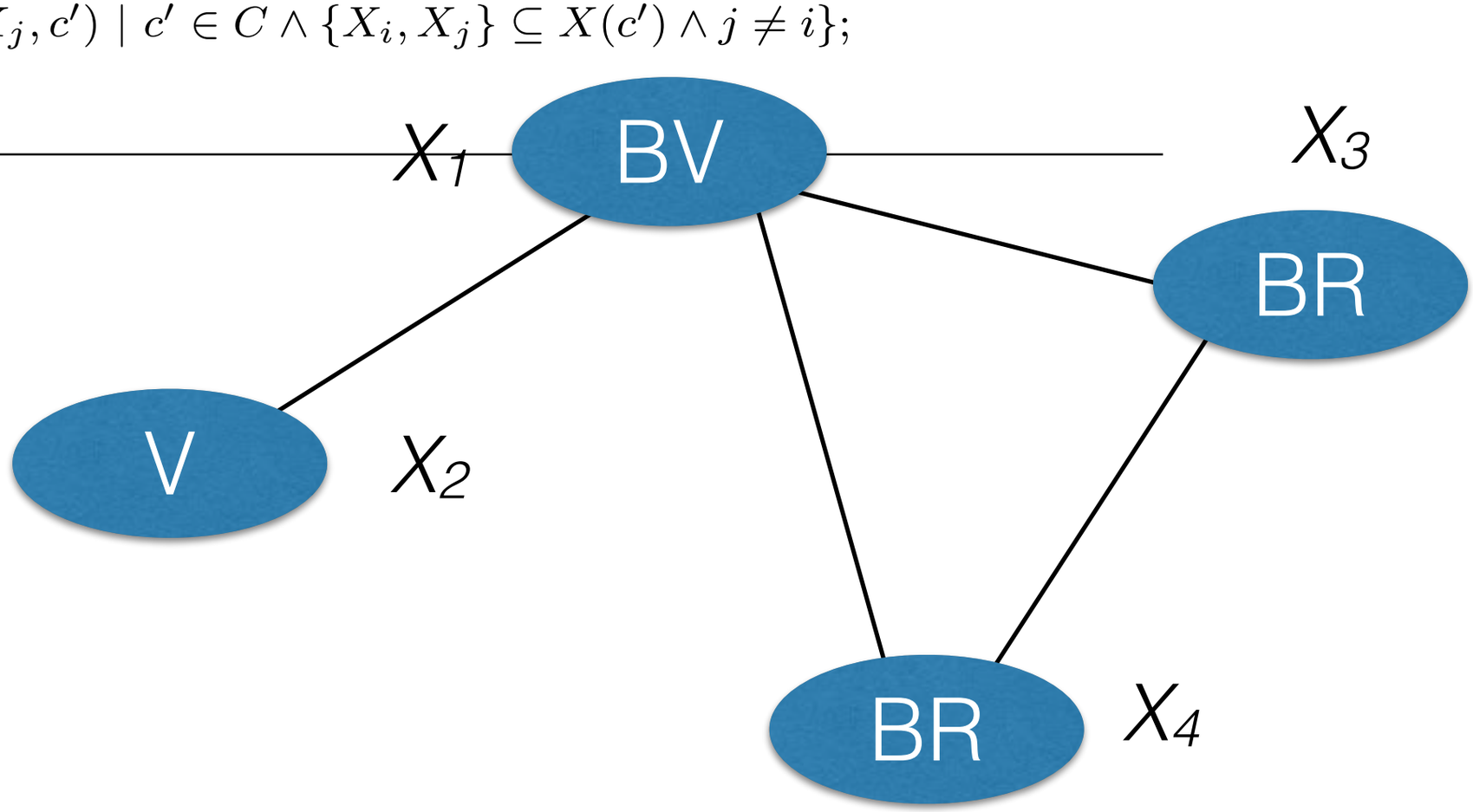
- AC1,…, AC8, AC2001, AC3.2, AC3.3, AC3d…, STR1,…, STR3, etc.

AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
    begin
1       Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2       while Q ≠ ∅ do
3           select and delete (X_i, c) from Q;
4           if revise(X_i, c) then
5               if D(X_i) = ∅ then return false ;
6               else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7       return true ;
    end
```

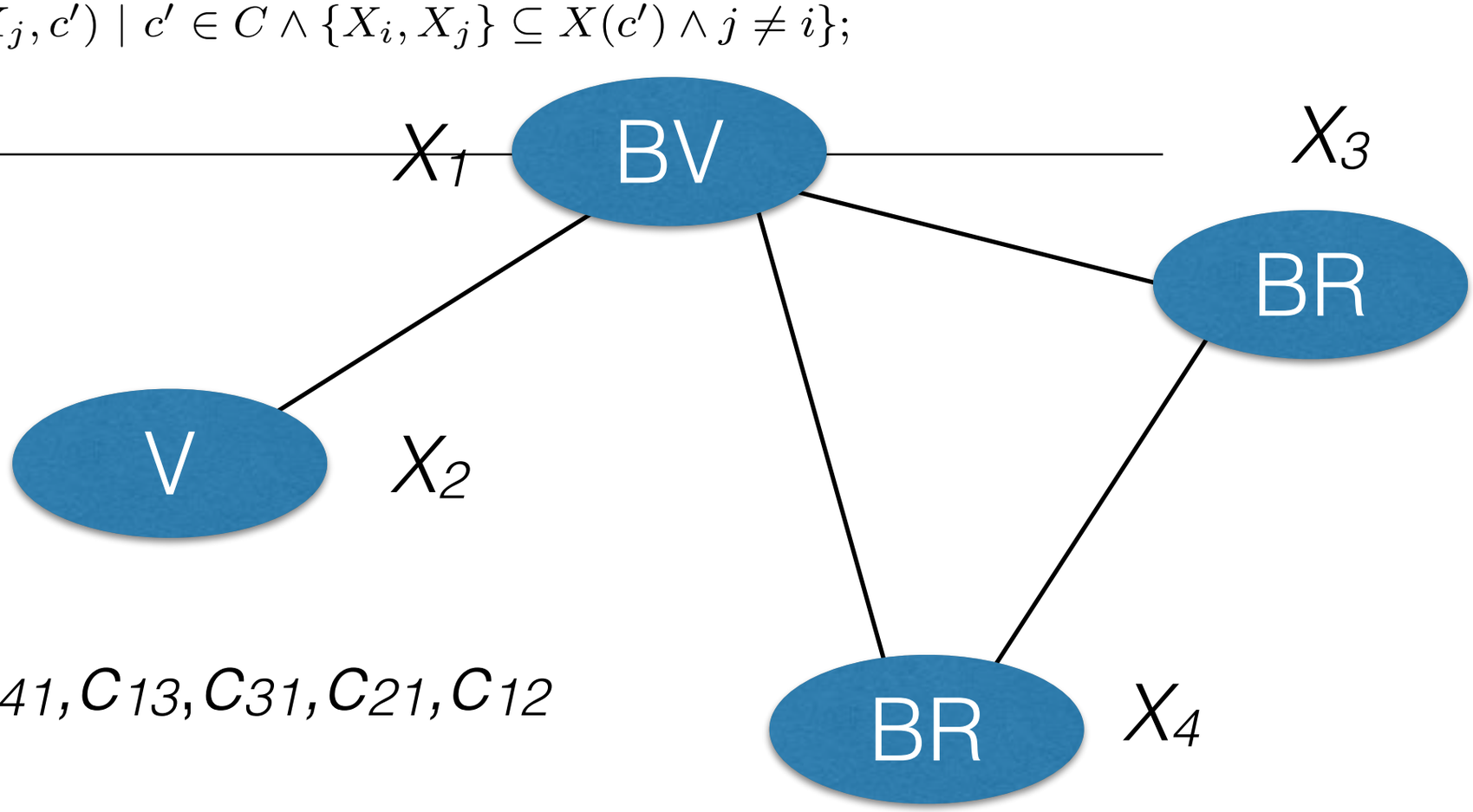
AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



AC3 [Mackworth1977]

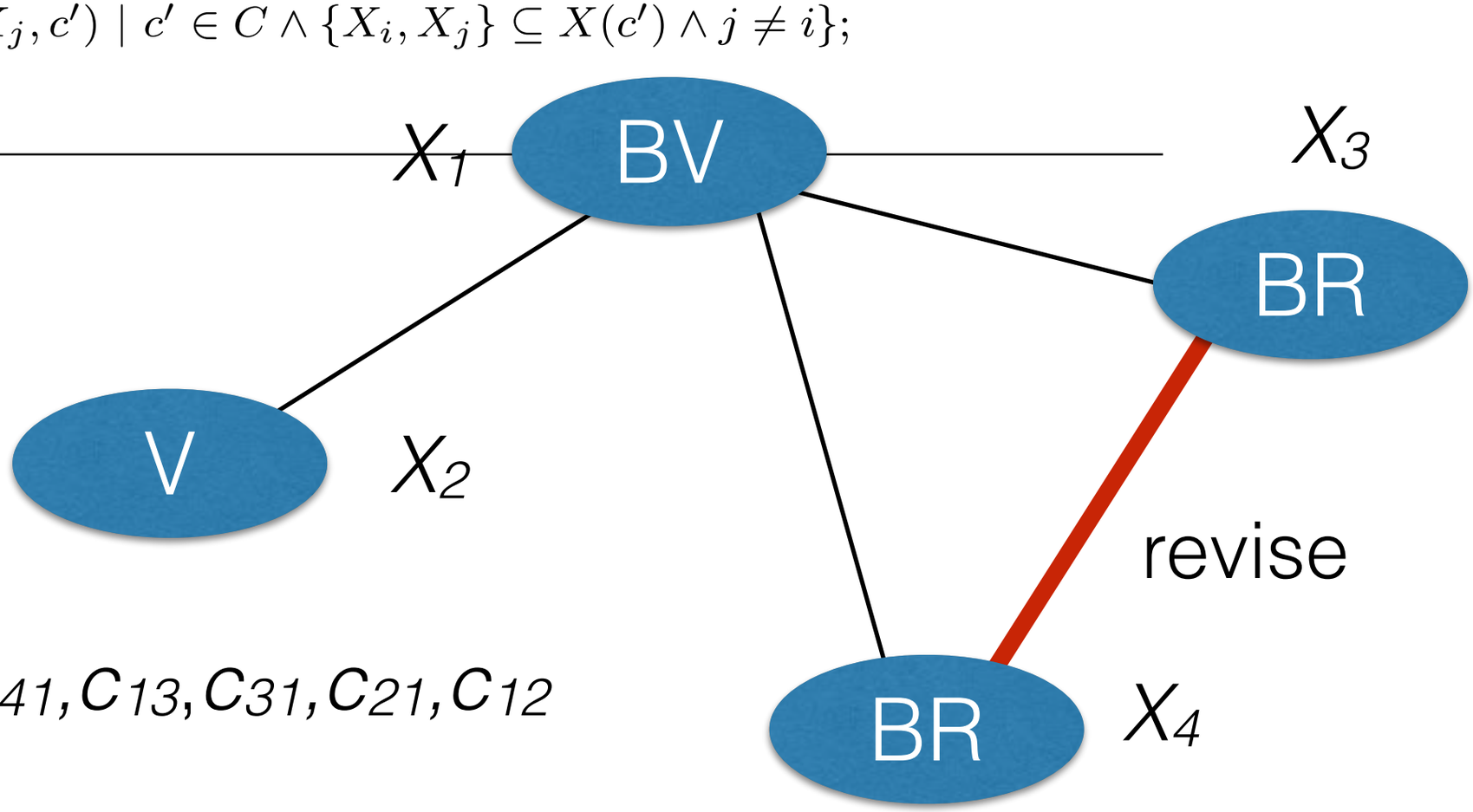
```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {$c_{34}, c_{43}, c_{14}, c_{41}, c_{13}, c_{31}, c_{21}, c_{12}$

AC3 [Mackworth1977]

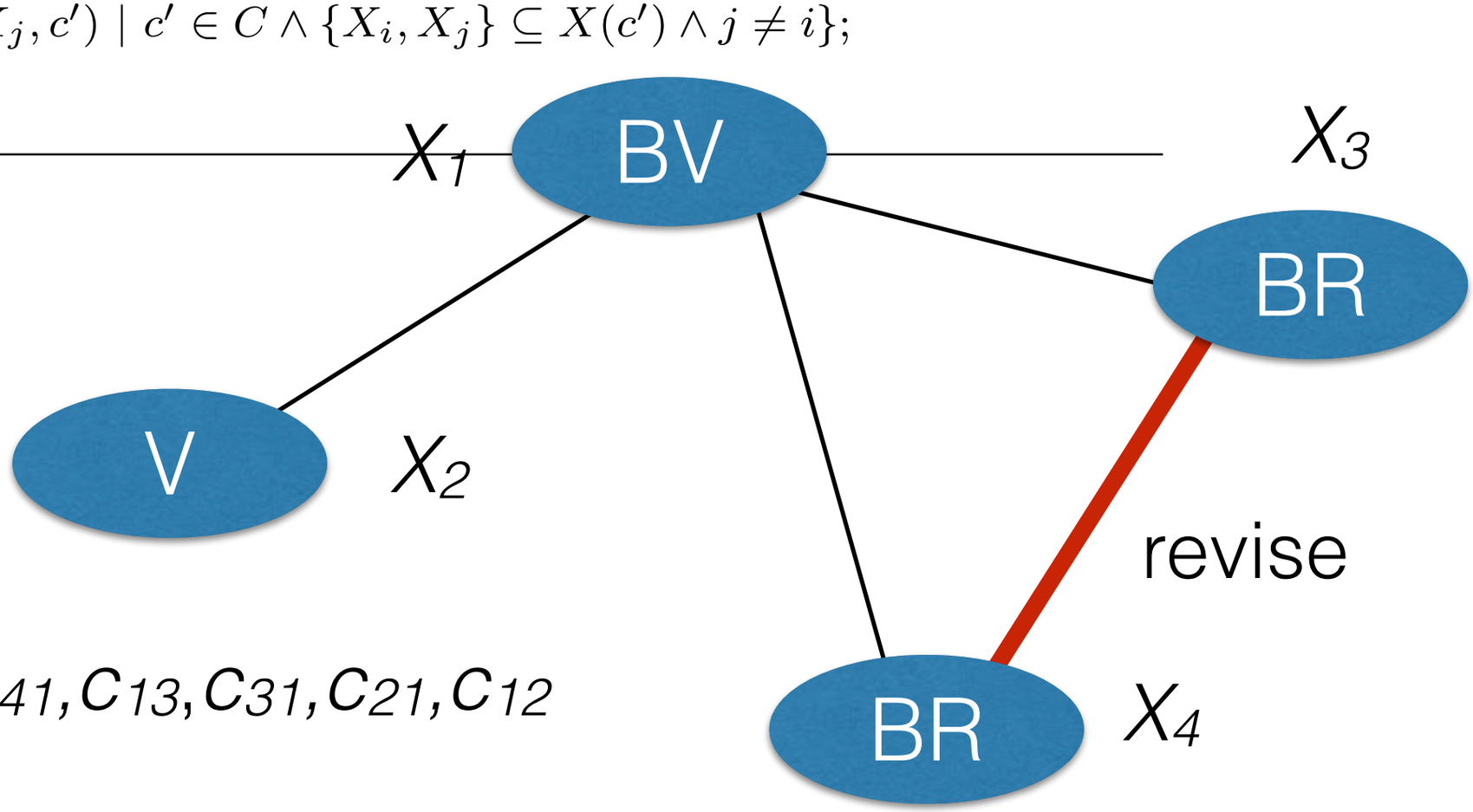
```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {~~c34~~,c43,c14,c41,c13,c31,c21,c12

AC3 [Mackworth1977]

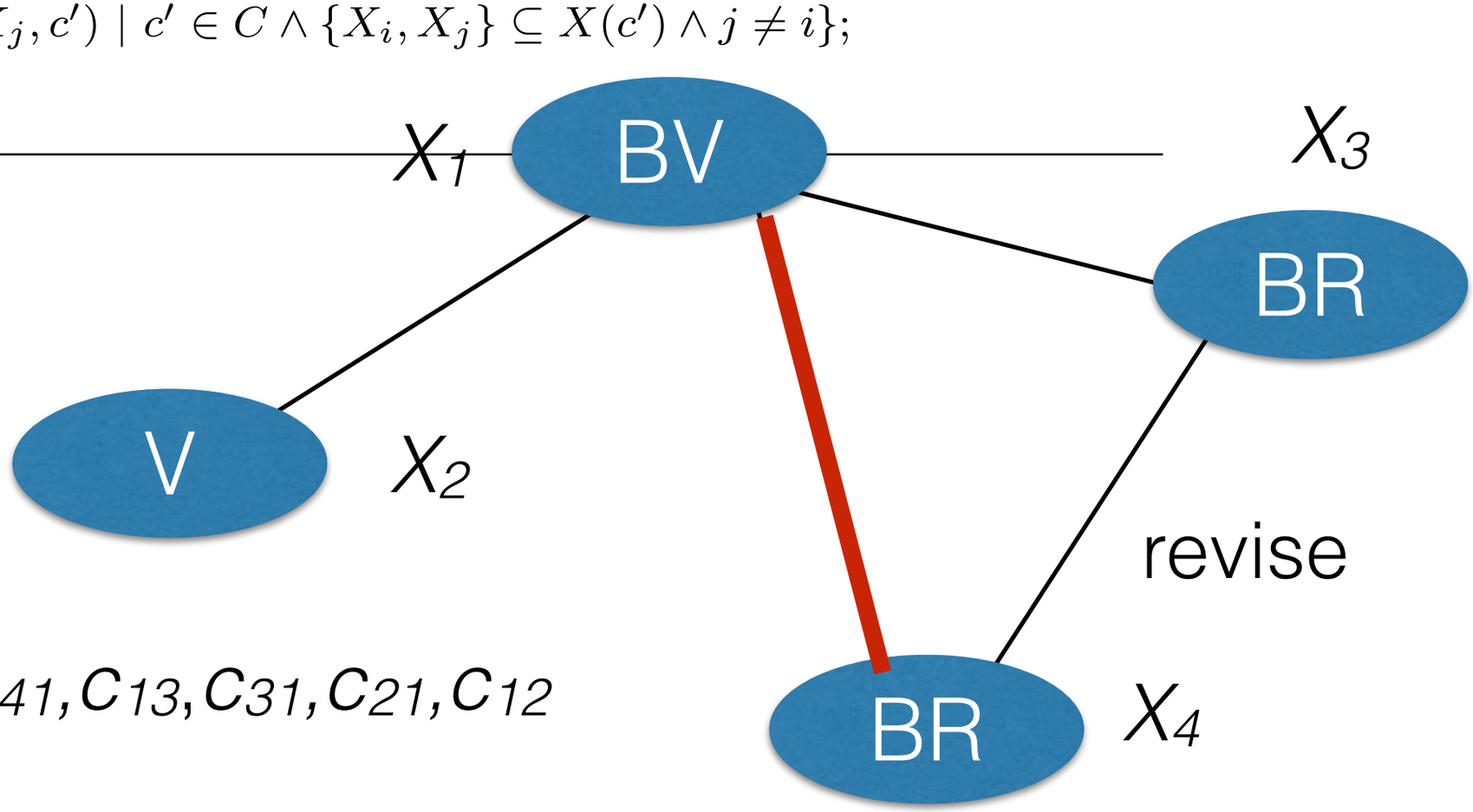
```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {~~c34~~,~~c43~~,c14,c41,c13,c31,c21,c12

AC3 [Mackworth1977]

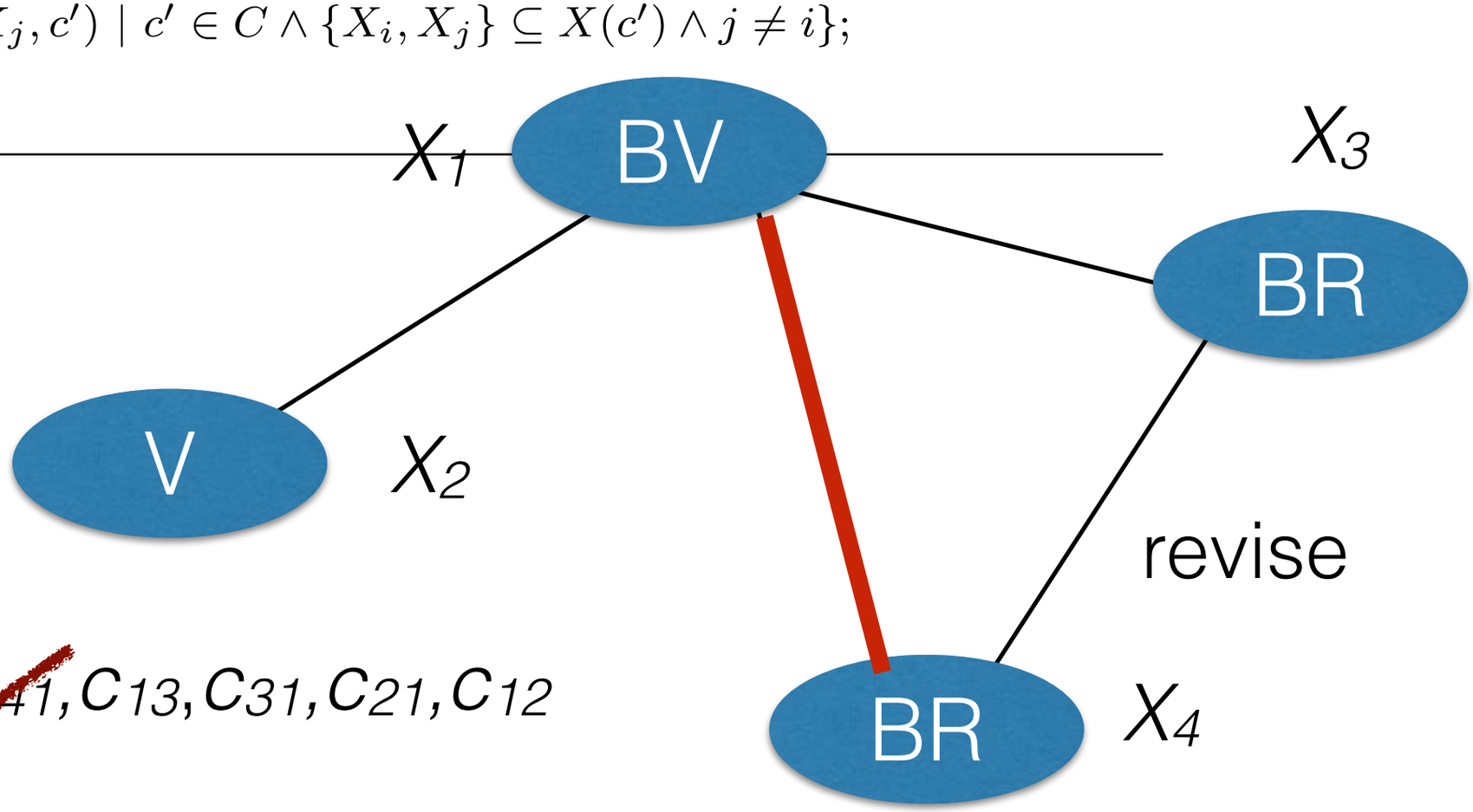
```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {~~c34~~,~~c43~~,~~c14~~,c41,c13,c31,c21,c12

AC3 [Mackworth1977]

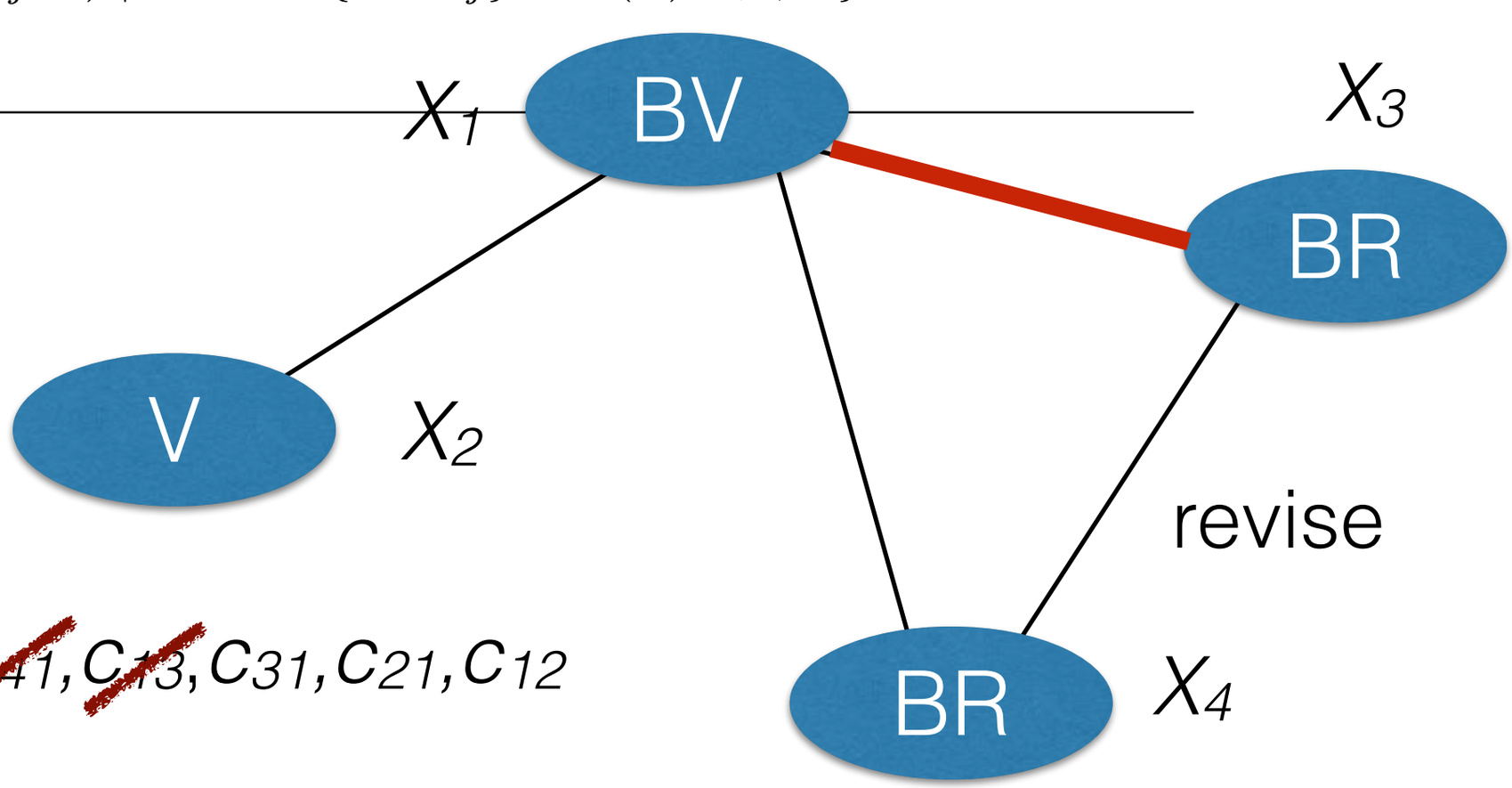
```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {~~c34~~,~~c43~~,~~c14~~,~~c41~~,c13,c31,c21,c12

AC3 [Mackworth1977]

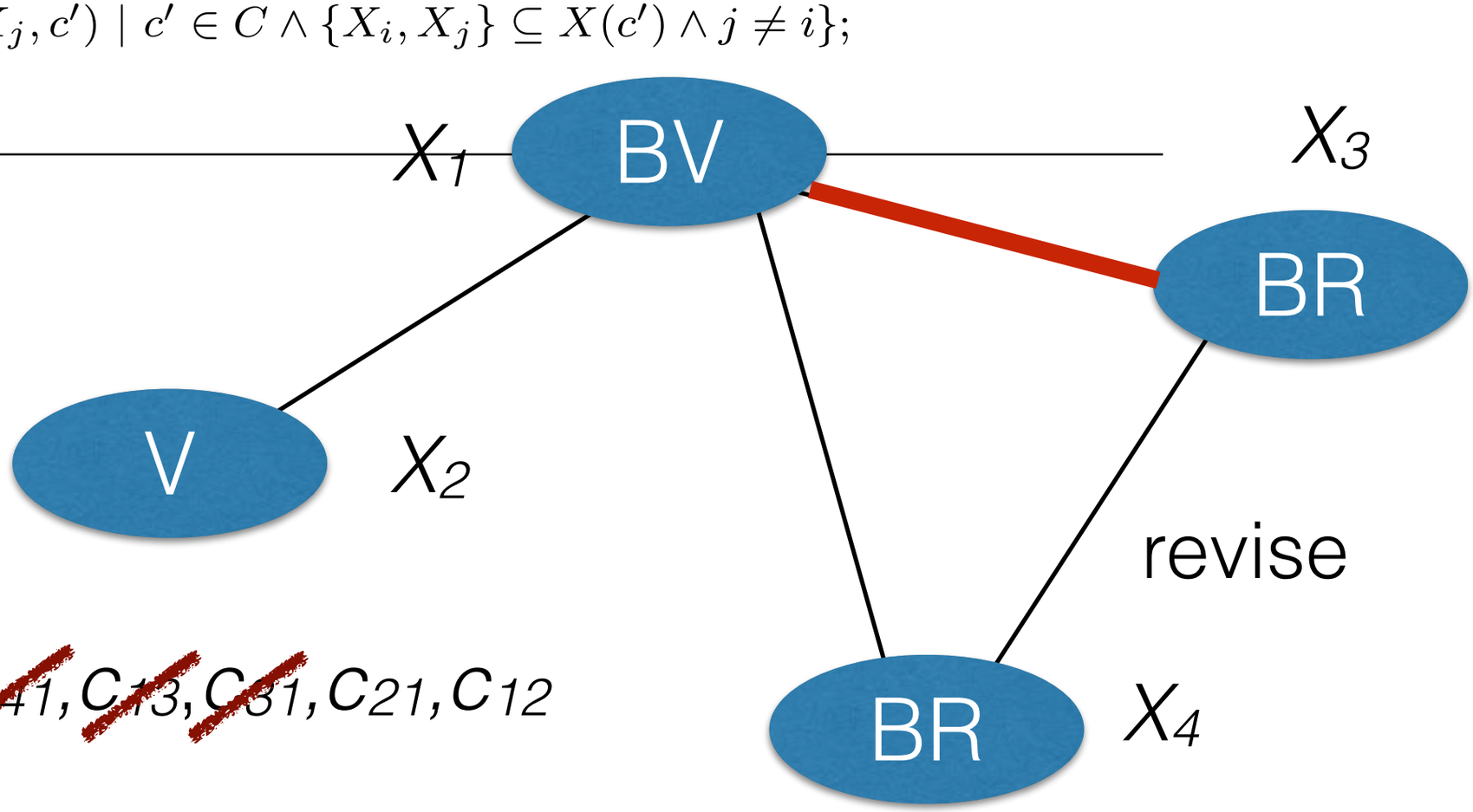
```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {c34,c43,c14,c41,c13,c31,c21,c12

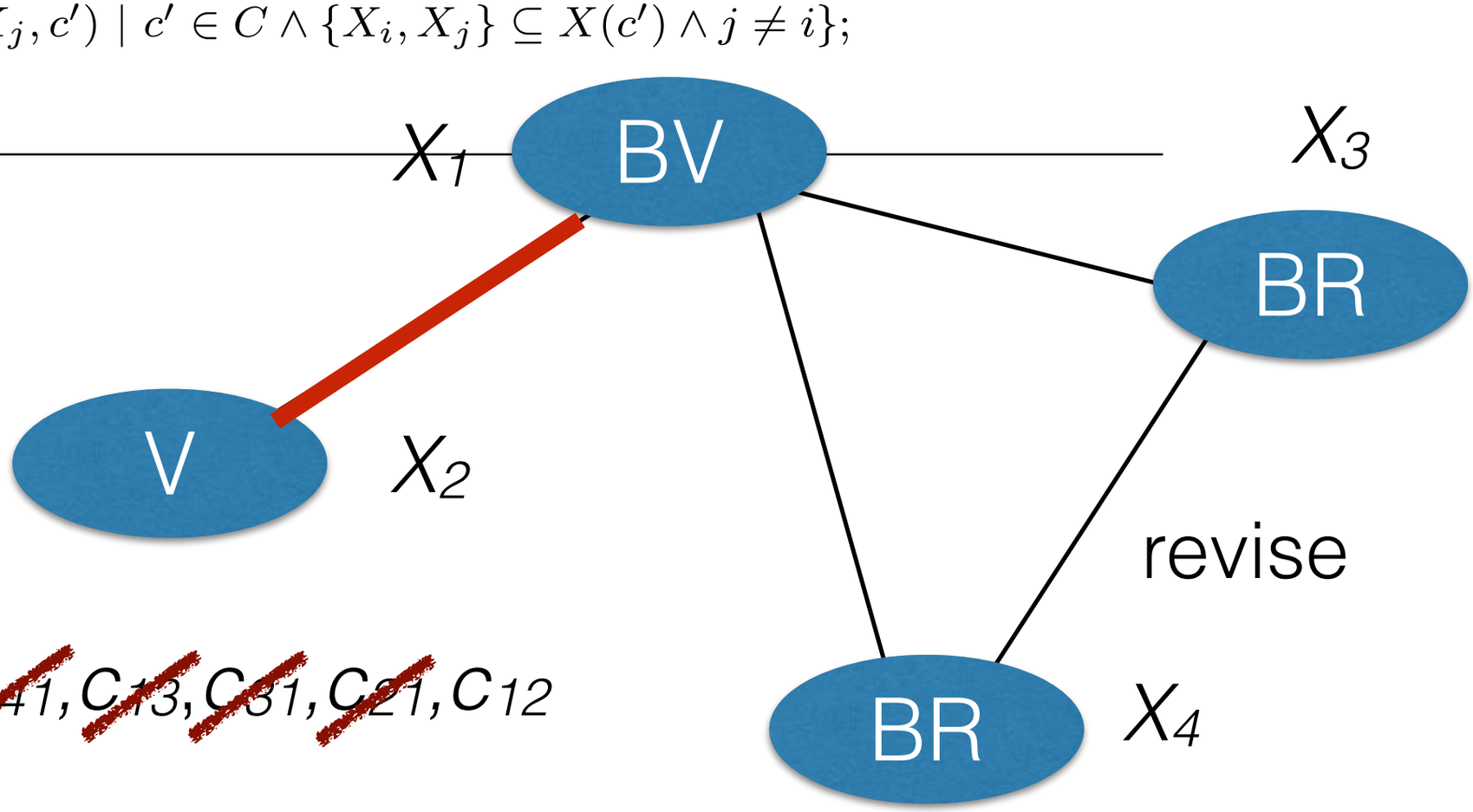
AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



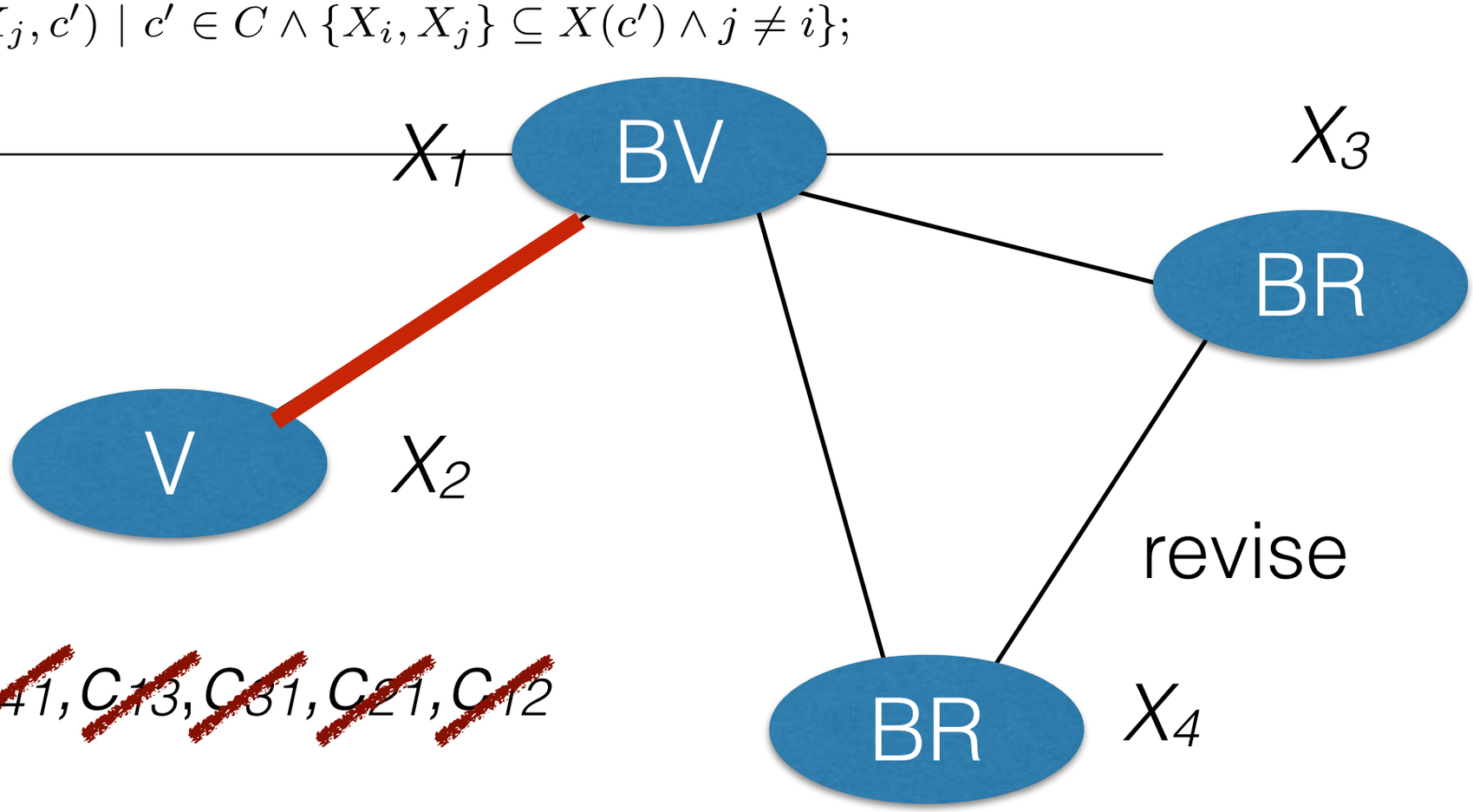
AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



AC3 [Mackworth1977]

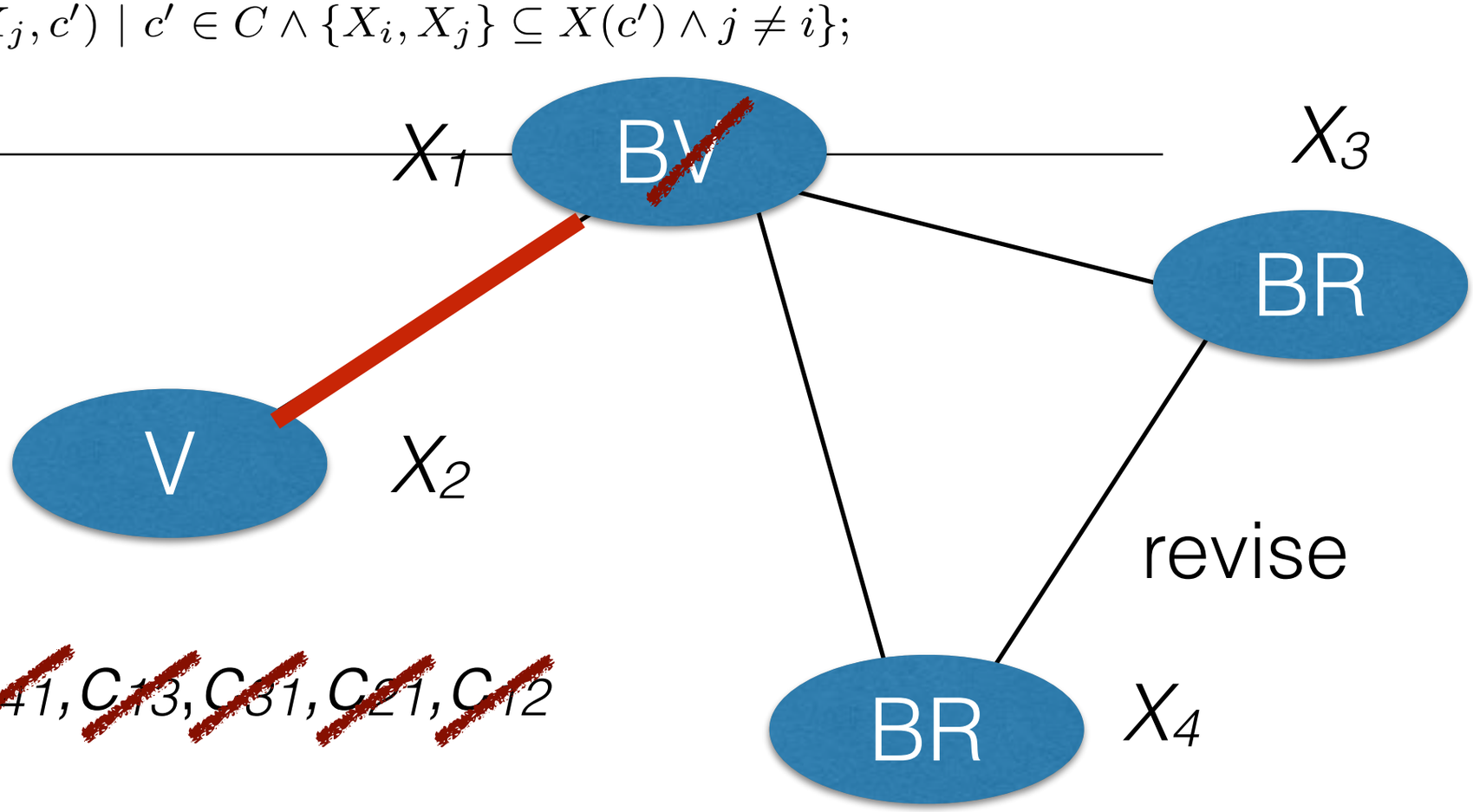
```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {~~c34~~,~~c43~~,~~c14~~,~~c41~~,~~c13~~,~~c31~~,~~c21~~,~~c12~~

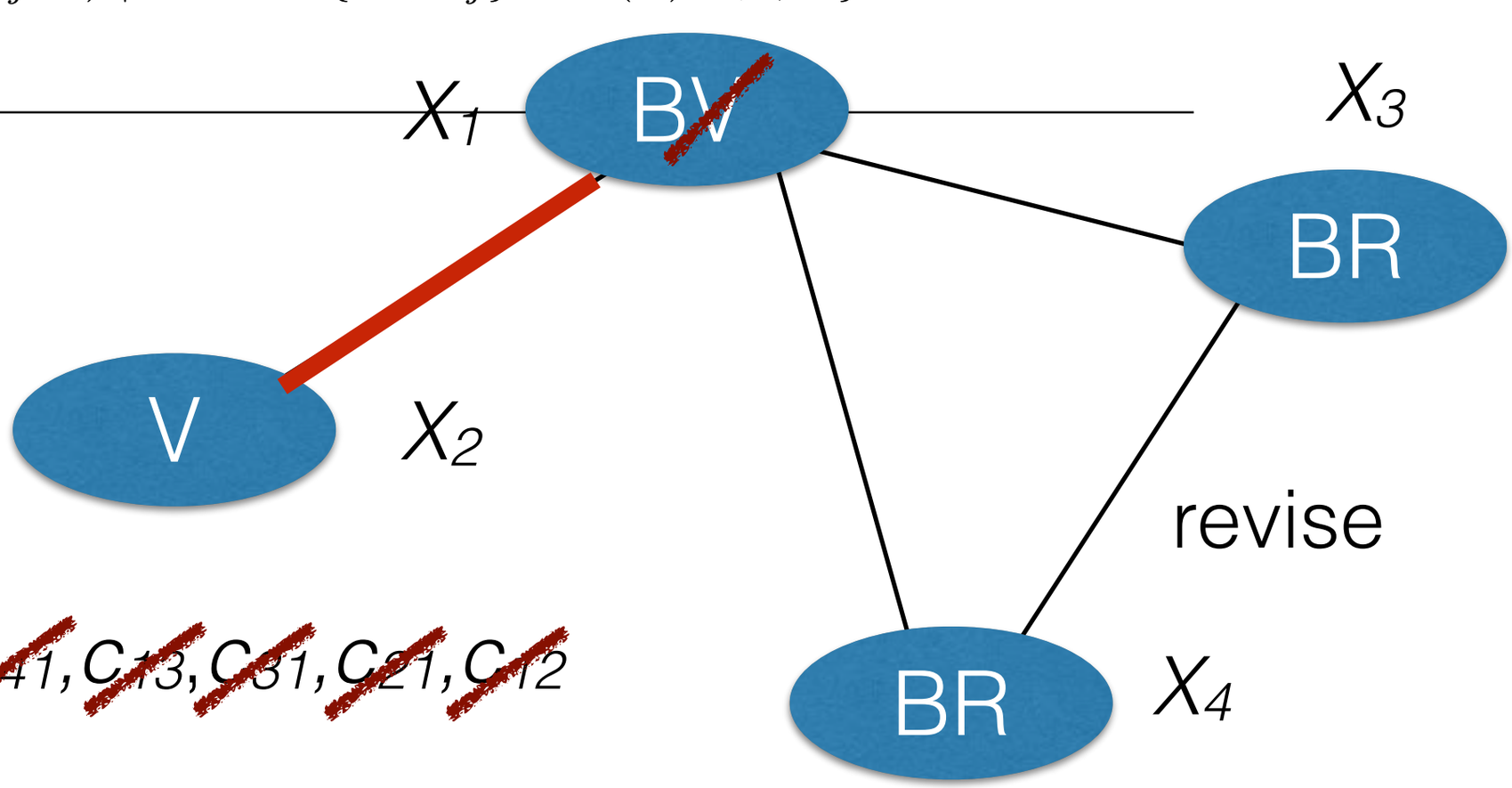
AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```

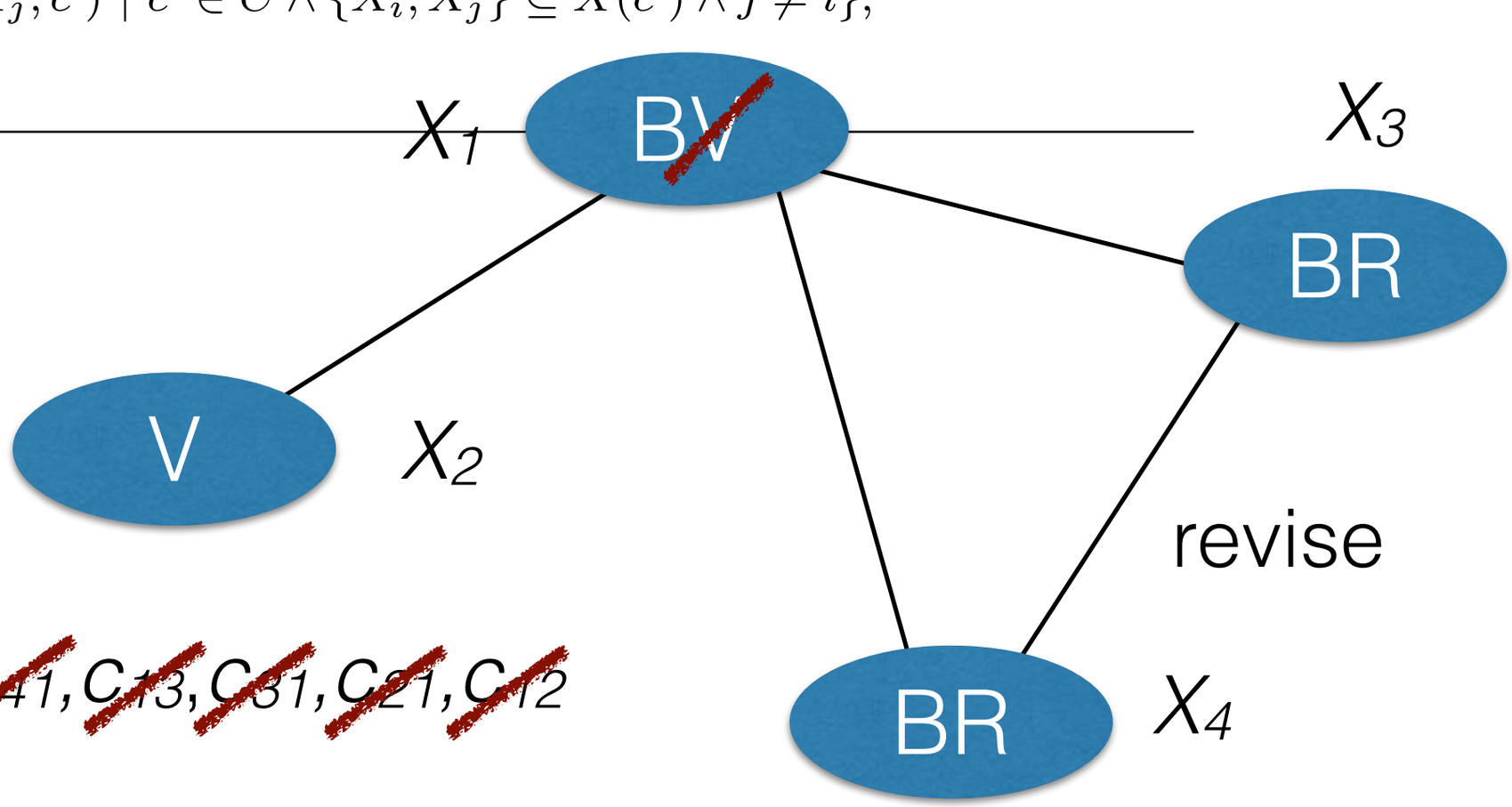


Q= {~~c34~~,~~c43~~,~~c14~~,~~c41~~,~~c13~~,~~c31~~,~~c21~~,~~c12~~

c31,c41

AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```

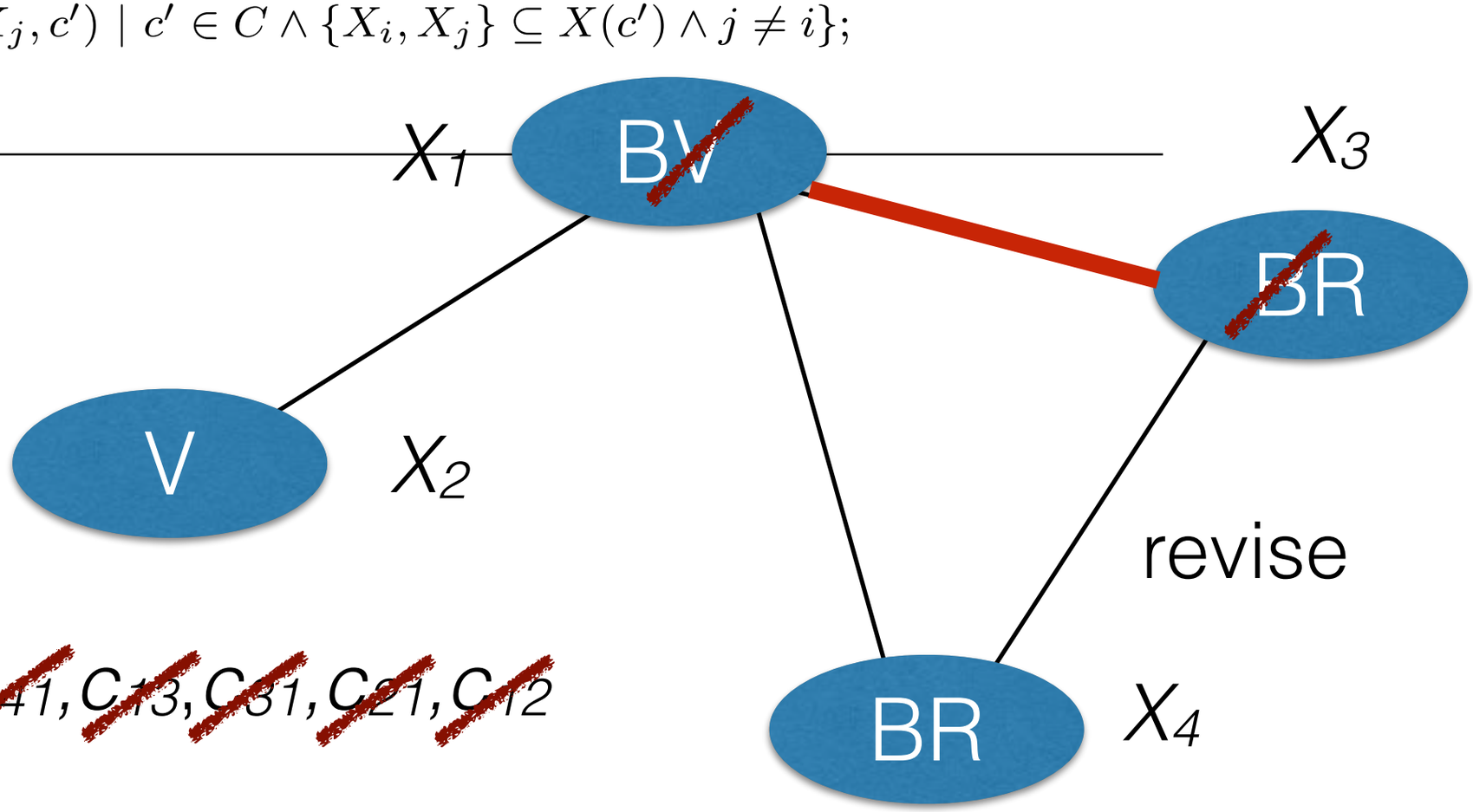


Q= {~~c34~~,~~c43~~,~~c14~~,~~c41~~,~~c13~~,~~c31~~,~~c21~~,~~c12~~

c31,c41

AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```

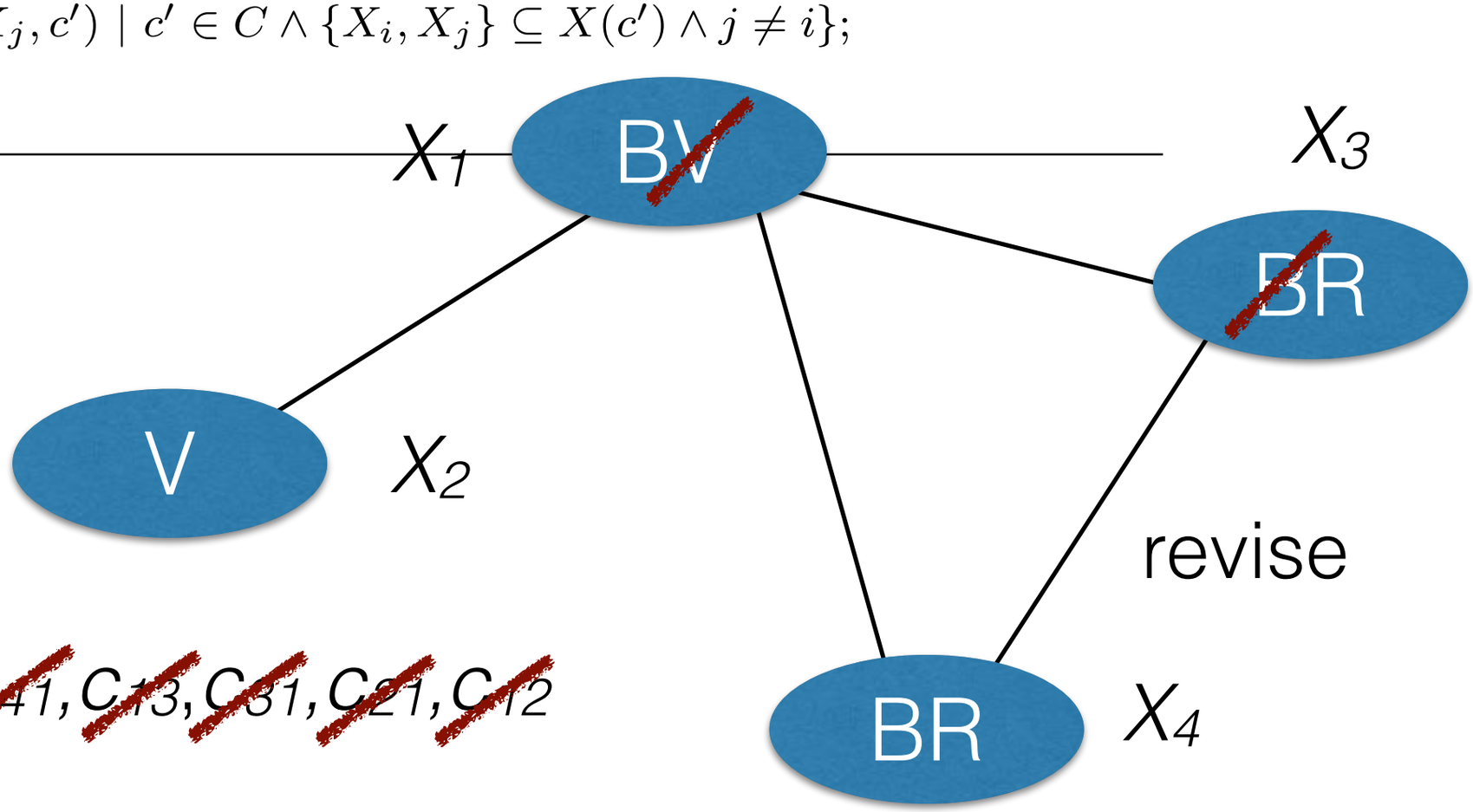


Q= {$c_{34}, c_{43}, c_{14}, c_{41}, c_{13}, c_{31}, c_{21}, c_{12}$

c_{31}, c_{41}

AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```

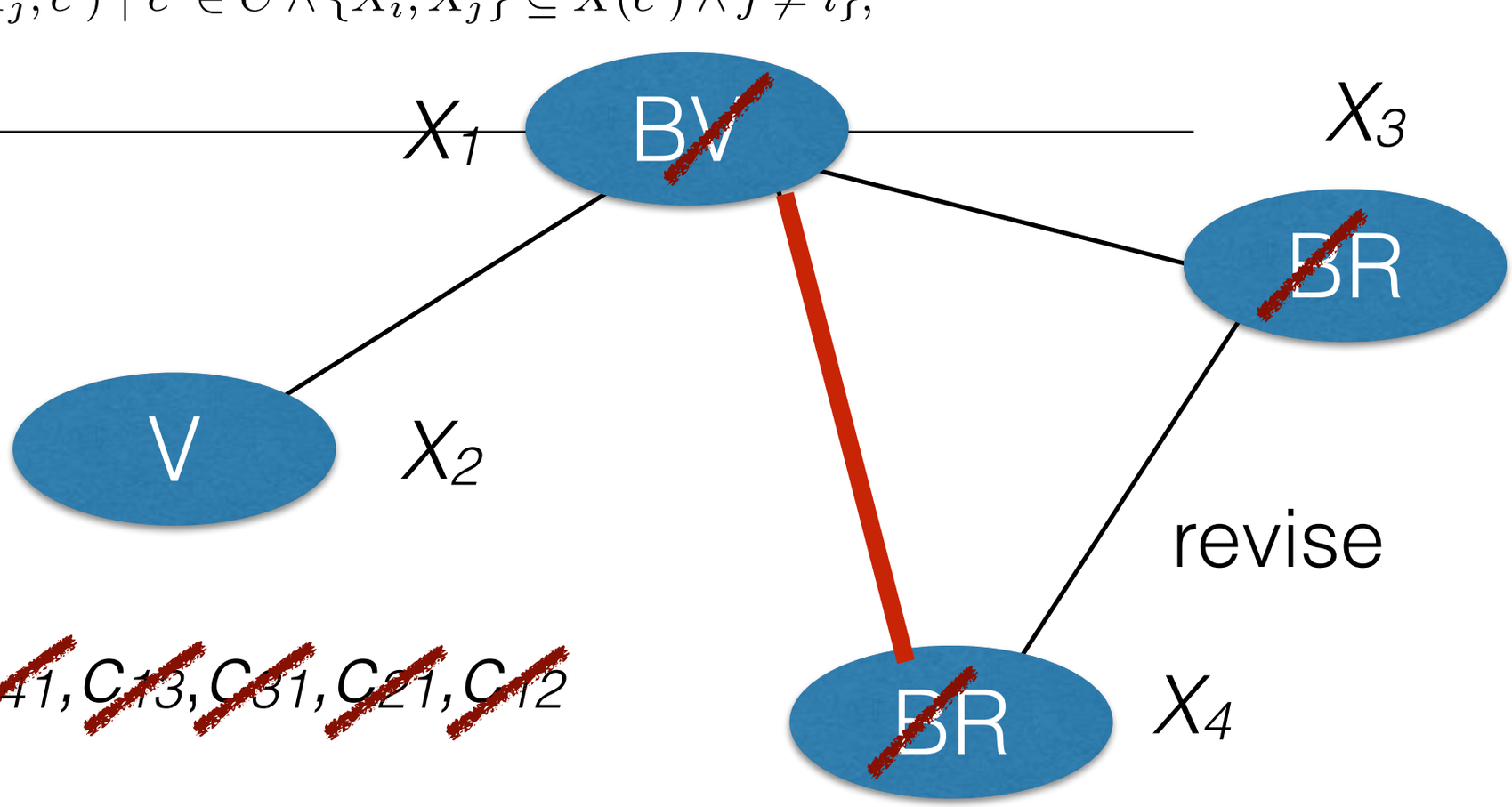


Q= {c34,c43,c14,c41,c13,c31,c21,c12

c31,c41 c43

AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
    begin
1       Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2       while Q ≠ ∅ do
3           select and delete (X_i, c) from Q;
4           if revise(X_i, c) then
5               if D(X_i) = ∅ then return false ;
6               else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7       return true ;
    end
```

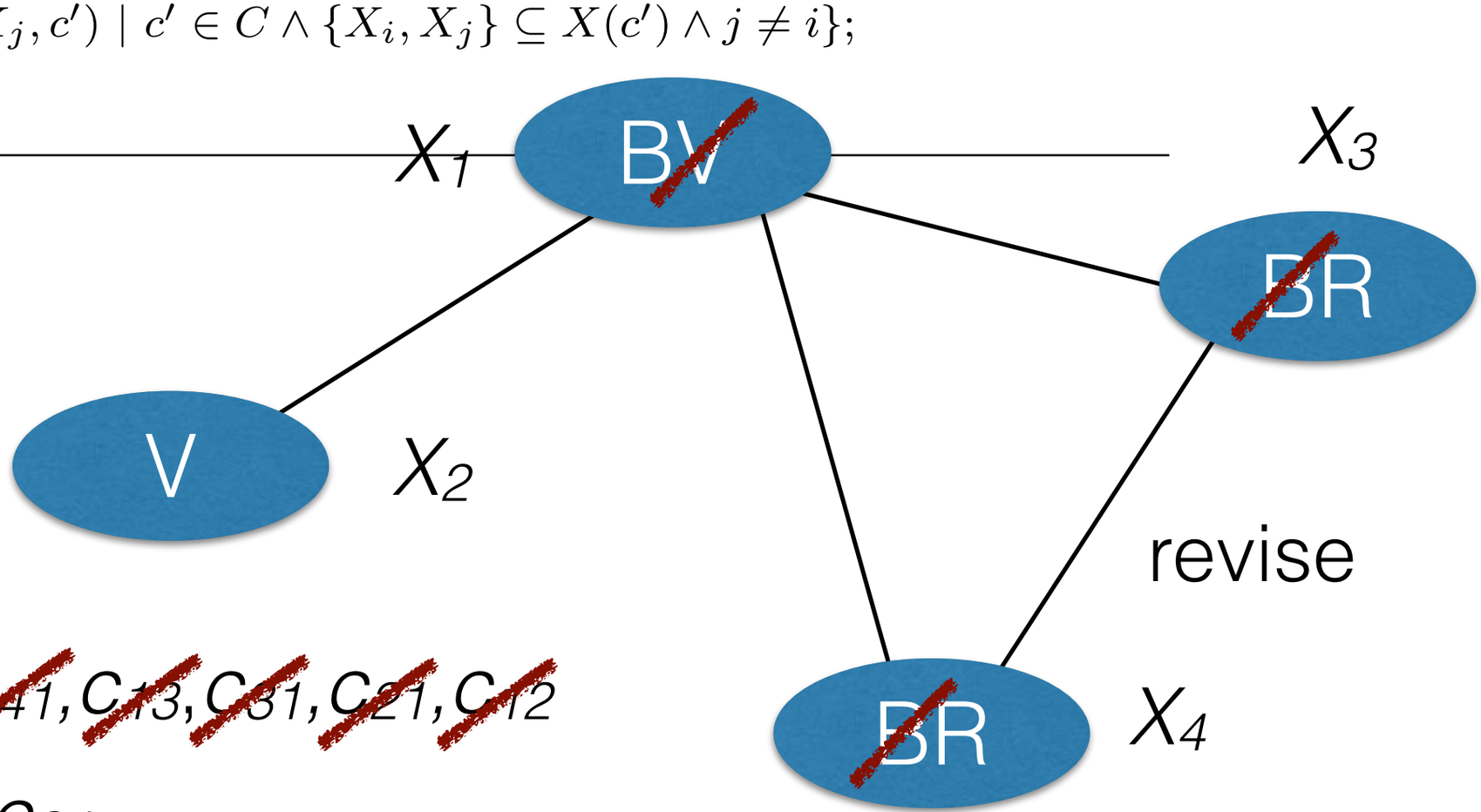


Q= {c34,c43,c14,c41,c13,c31,c21,c12

c31,c41 c43

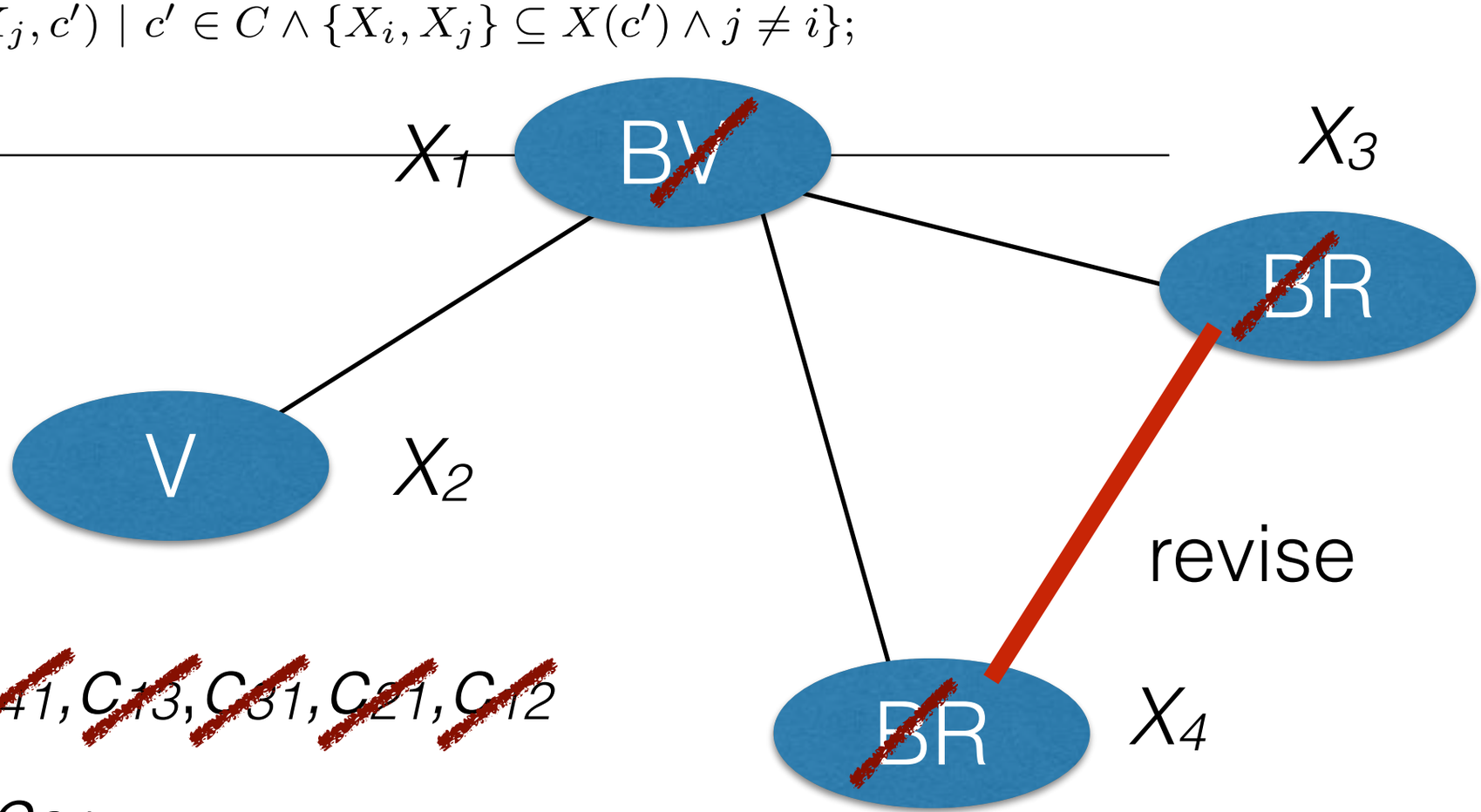
AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```

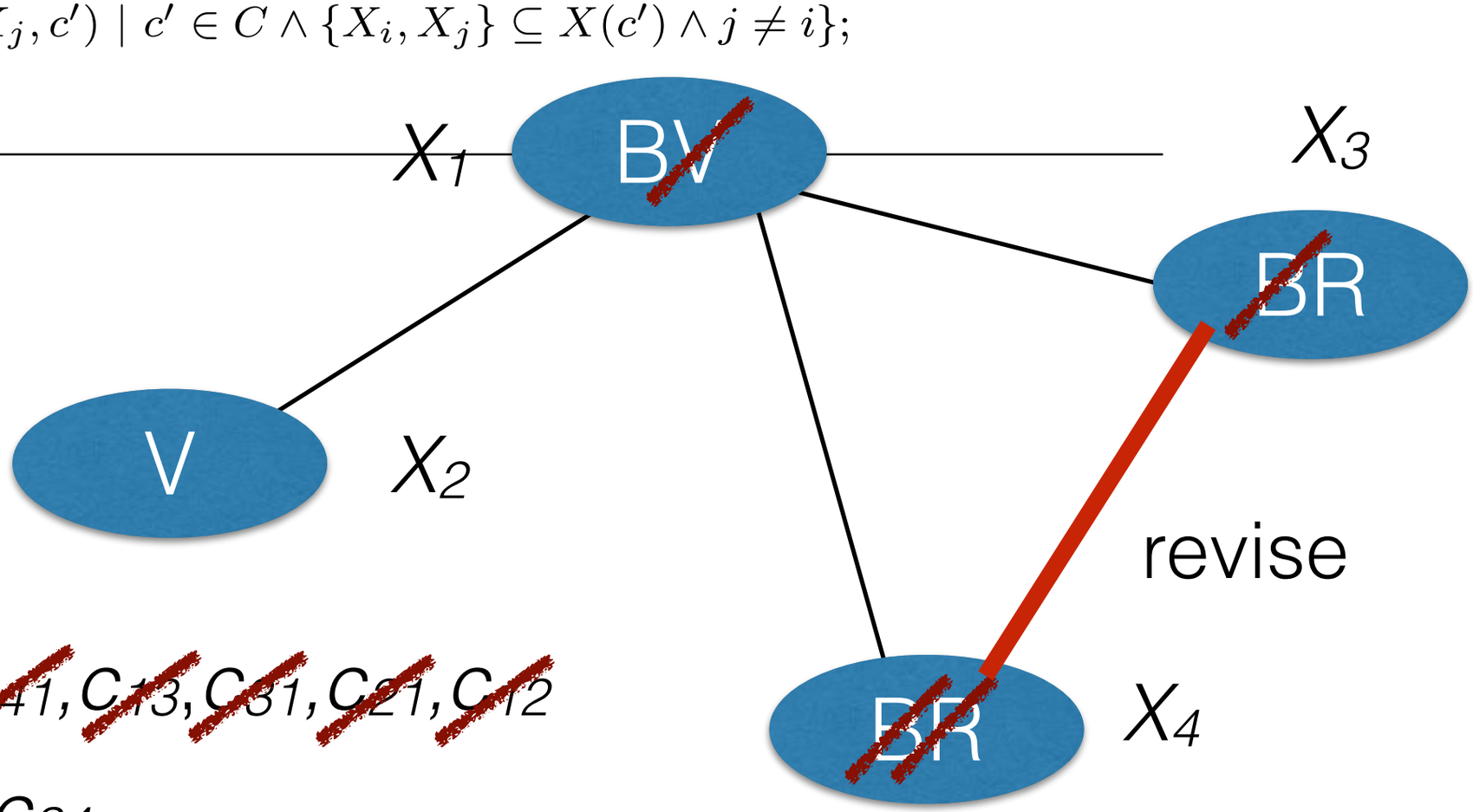


Q= {c34,c43,c14,c41,c13,c31,c21,c12

c31,c41 c43 c34

AC3 [Mackworth1977]

```
function AC3(in X : set) : Boolean
   begin
1      Q ← {(X_i, c) | c ∈ C, X_i ∈ X(c)};
2      while Q ≠ ∅ do
3          select and delete (X_i, c) from Q;
4          if revise(X_i, c) then
5              if D(X_i) = ∅ then return false ;
6              else Q ← Q ∪ {(X_j, c') | c' ∈ C ∧ {X_i, X_j} ⊆ X(c') ∧ j ≠ i};
7      return true ;
   end
```



Q= {c34, c43, c14, c41, c13, c31, c21, c12

c31, c41 c43 c34

empty domain

Revise of AC3

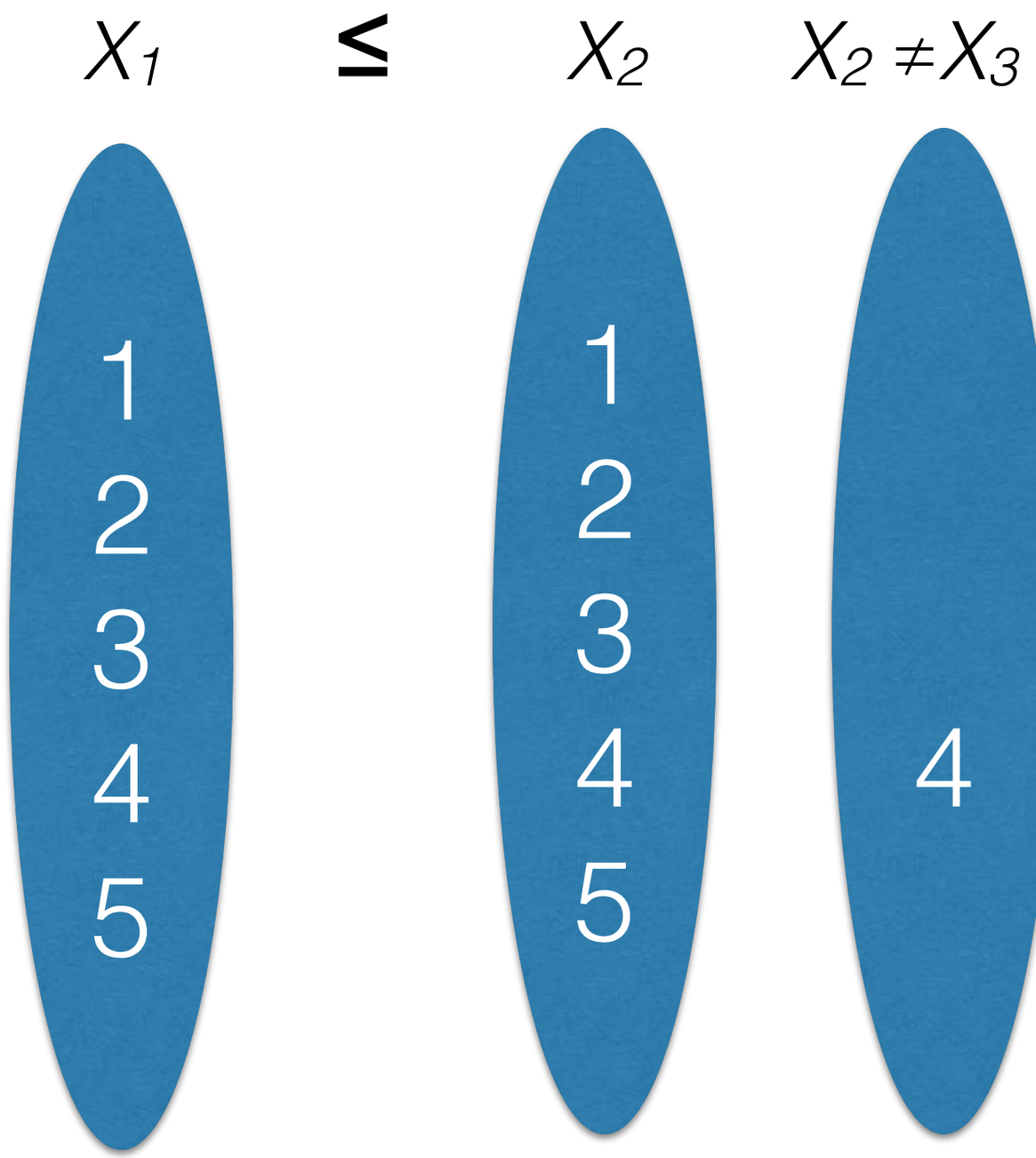
Revise of AC3

```
function Revise3(in X_i : variable; c_ij : constraint) : Boolean
    begin
1       CHANGE ← false;
2       foreach v_i ∈ D(X_i) do
3           v_j ← first(D(X_j));
4           while (v_j ≠ nil) and ¬c_ij(v_i, v_j) do v_j ← next(v_j, D(X_j));
5           if v_j = nil then
6               remove v_i from D(X_i);
7               CHANGE ← true;
8       return CHANGE ;
    end
```

AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

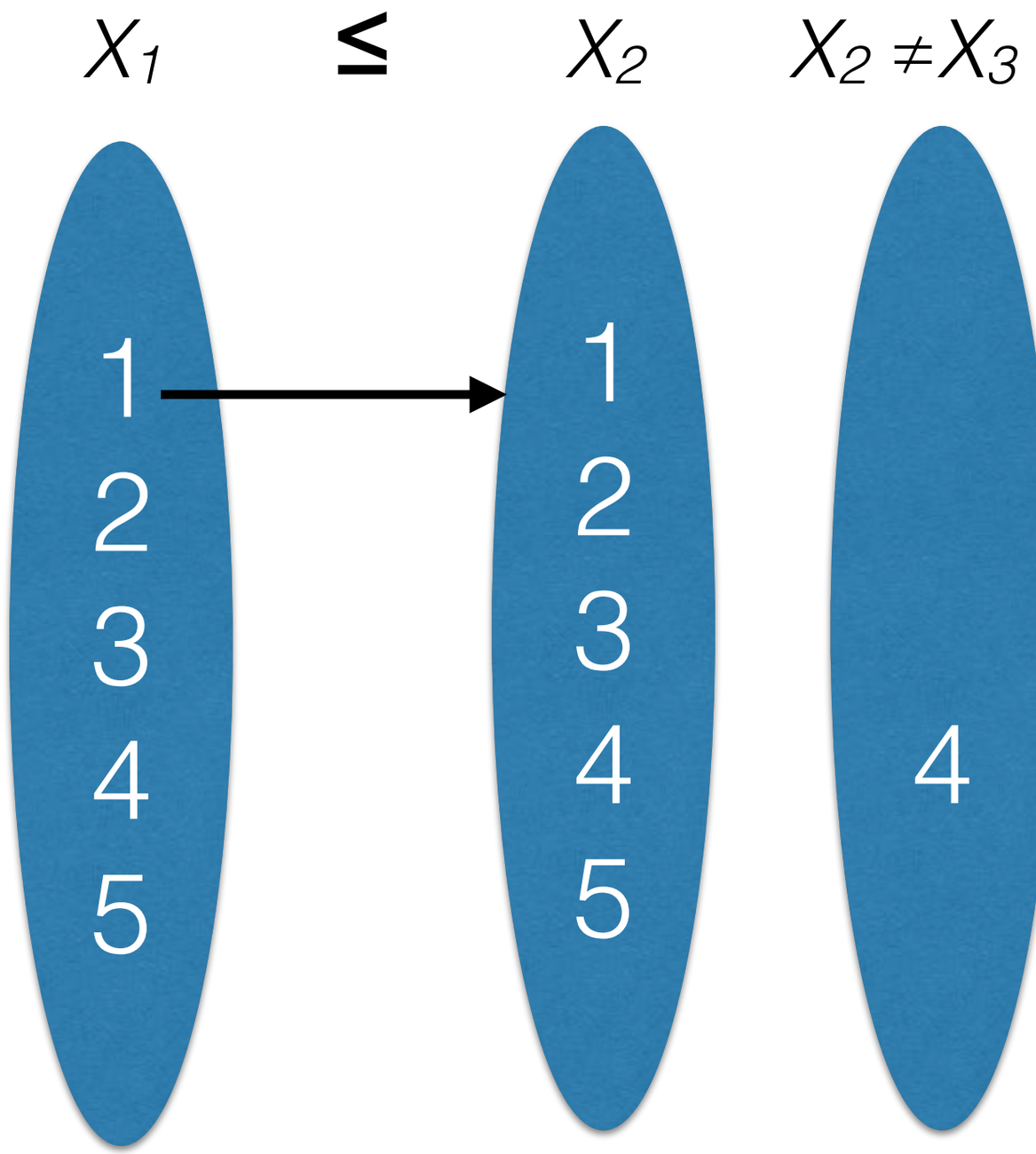
- Function Revise traverses *$D(X_1)$ x $D(X_2)$*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

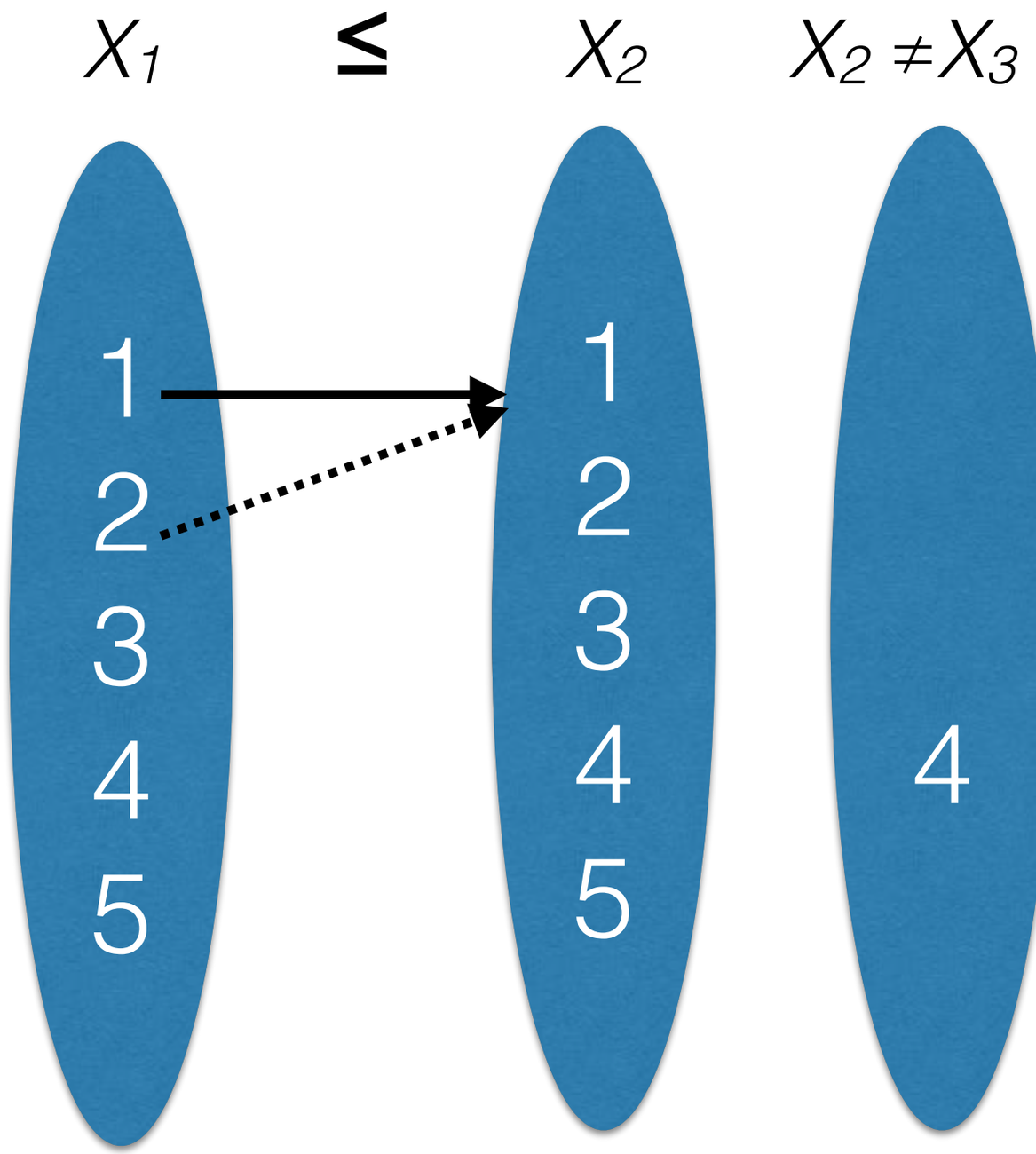
- Function Revise traverses *$D(X_1)$ x $D(X_2)$*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

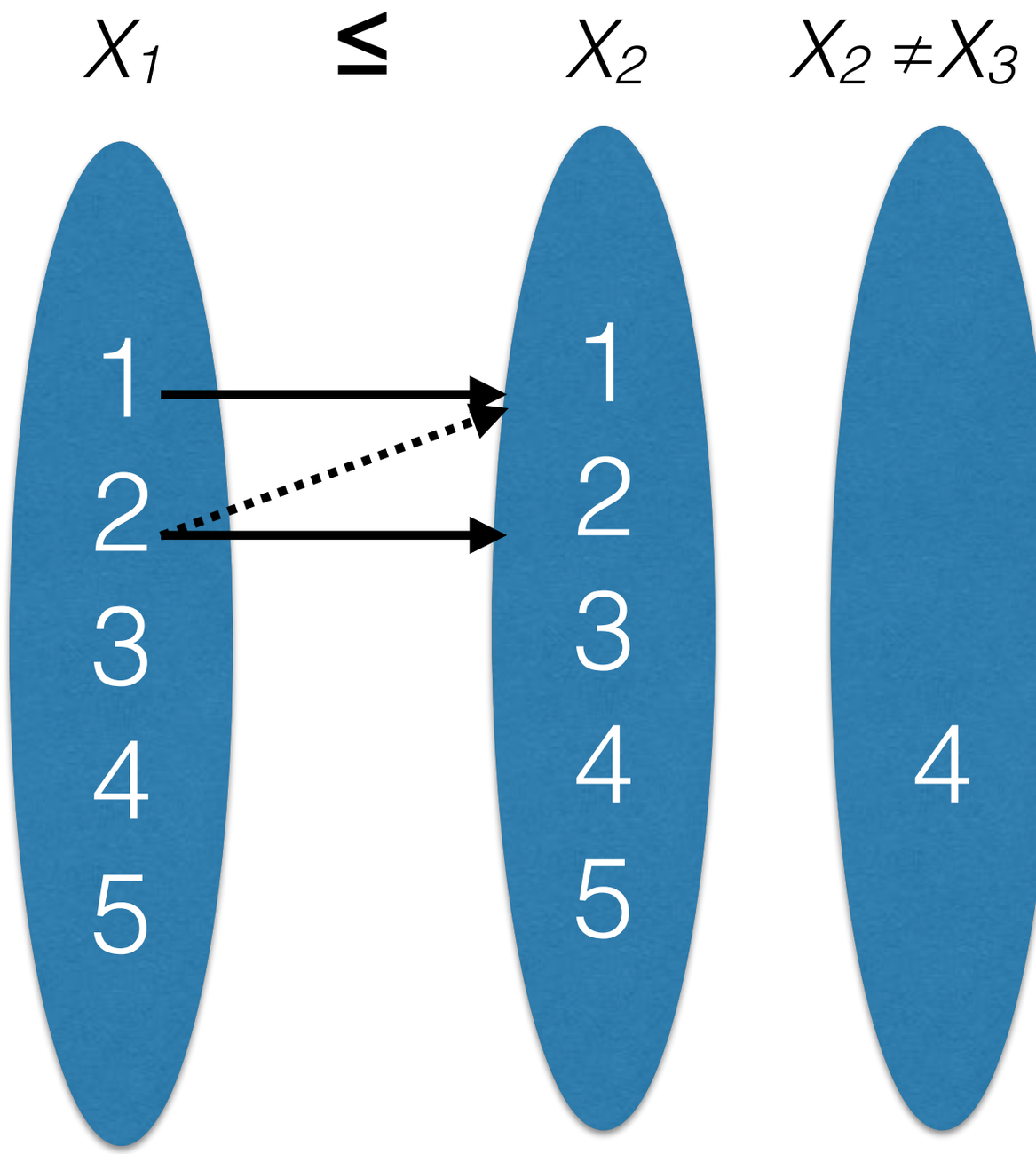
- Function Revise traverses *$D(X_1)$ x $D(X_2)$*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

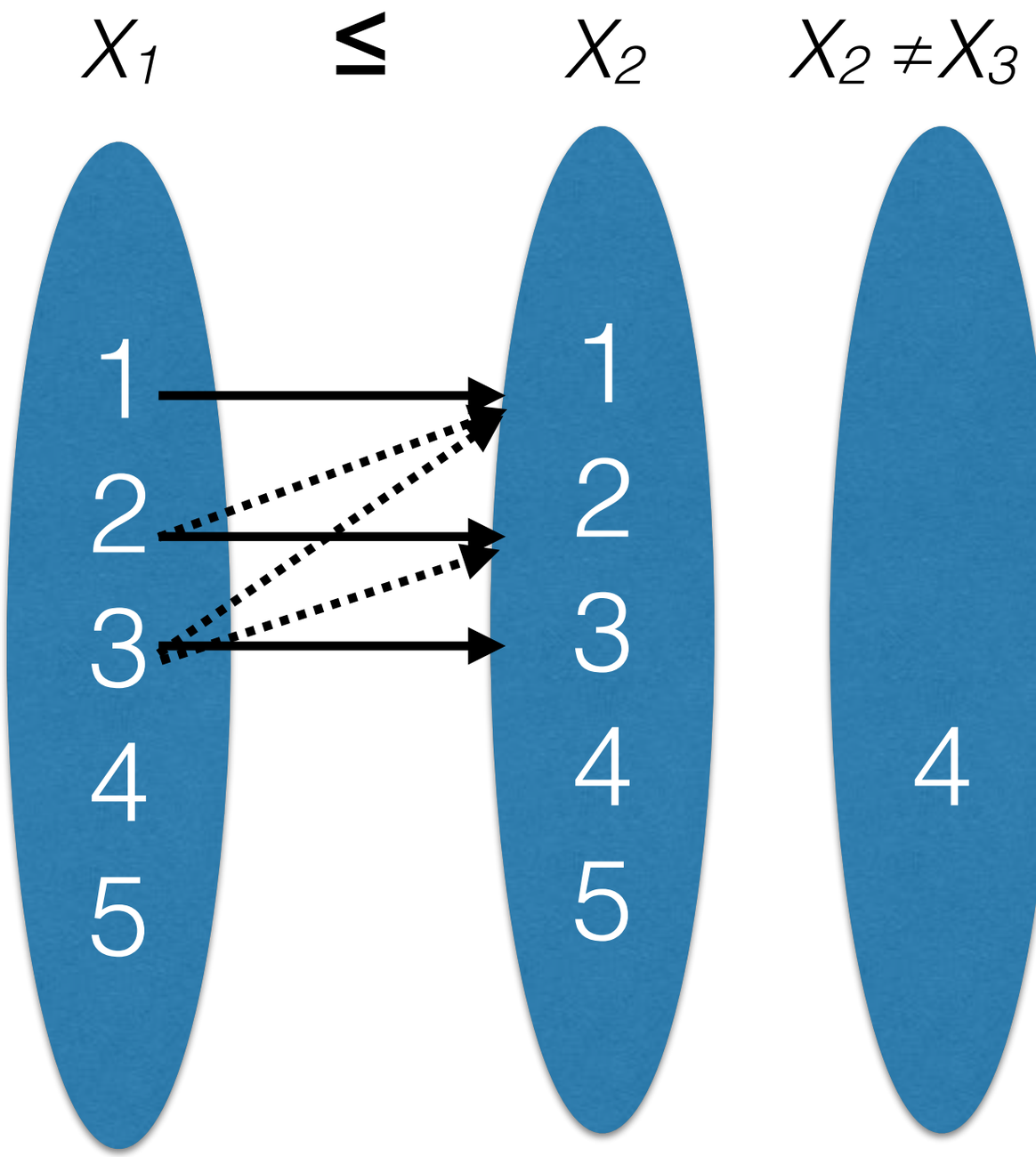
- Function Revise traverses *$D(X_1)$ x $D(X_2)$*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

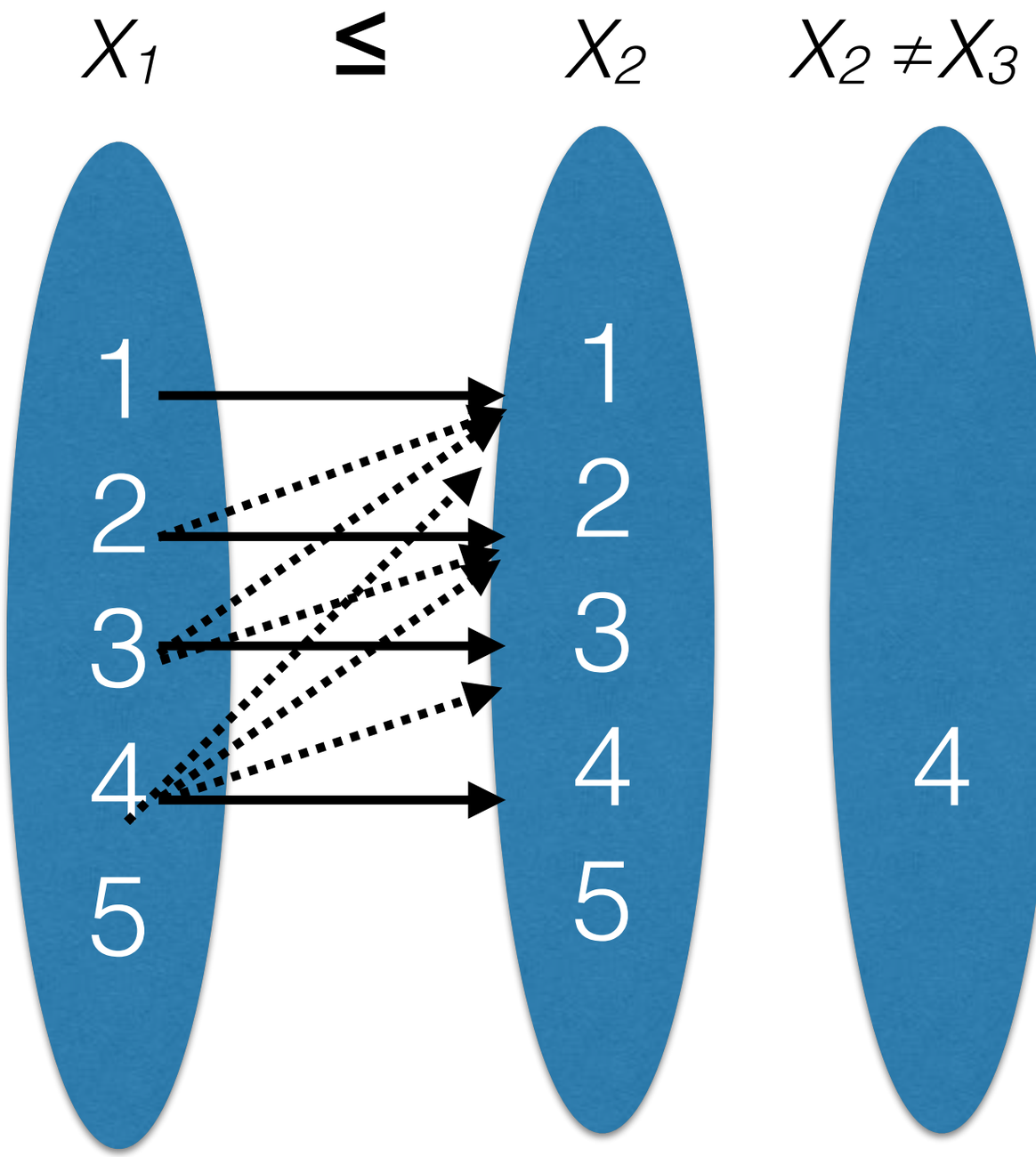
- Function Revise traverses *$D(X_1)$ x $D(X_2)$*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

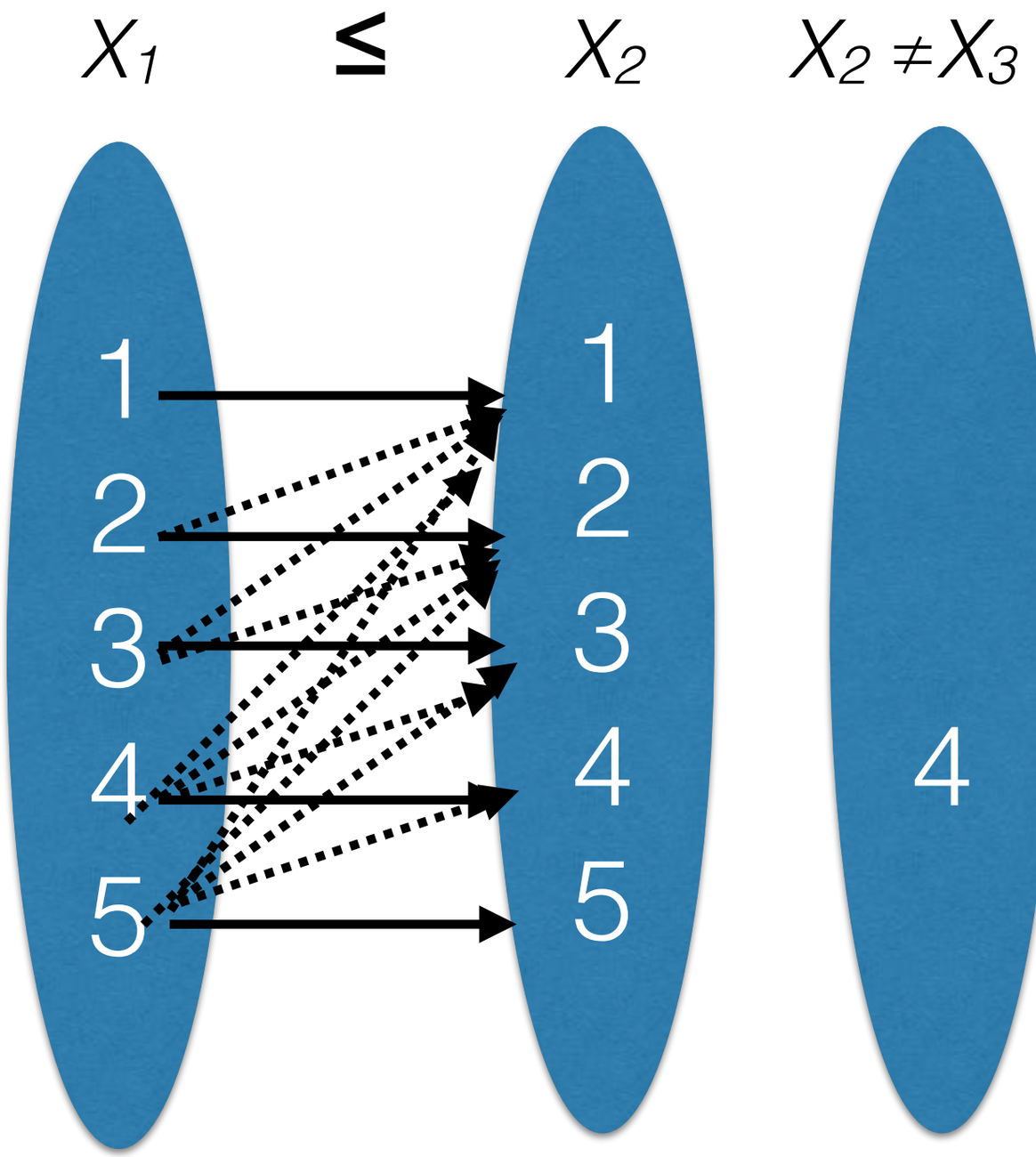
- Function Revise traverses *D(X_1) x D(X_2)*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

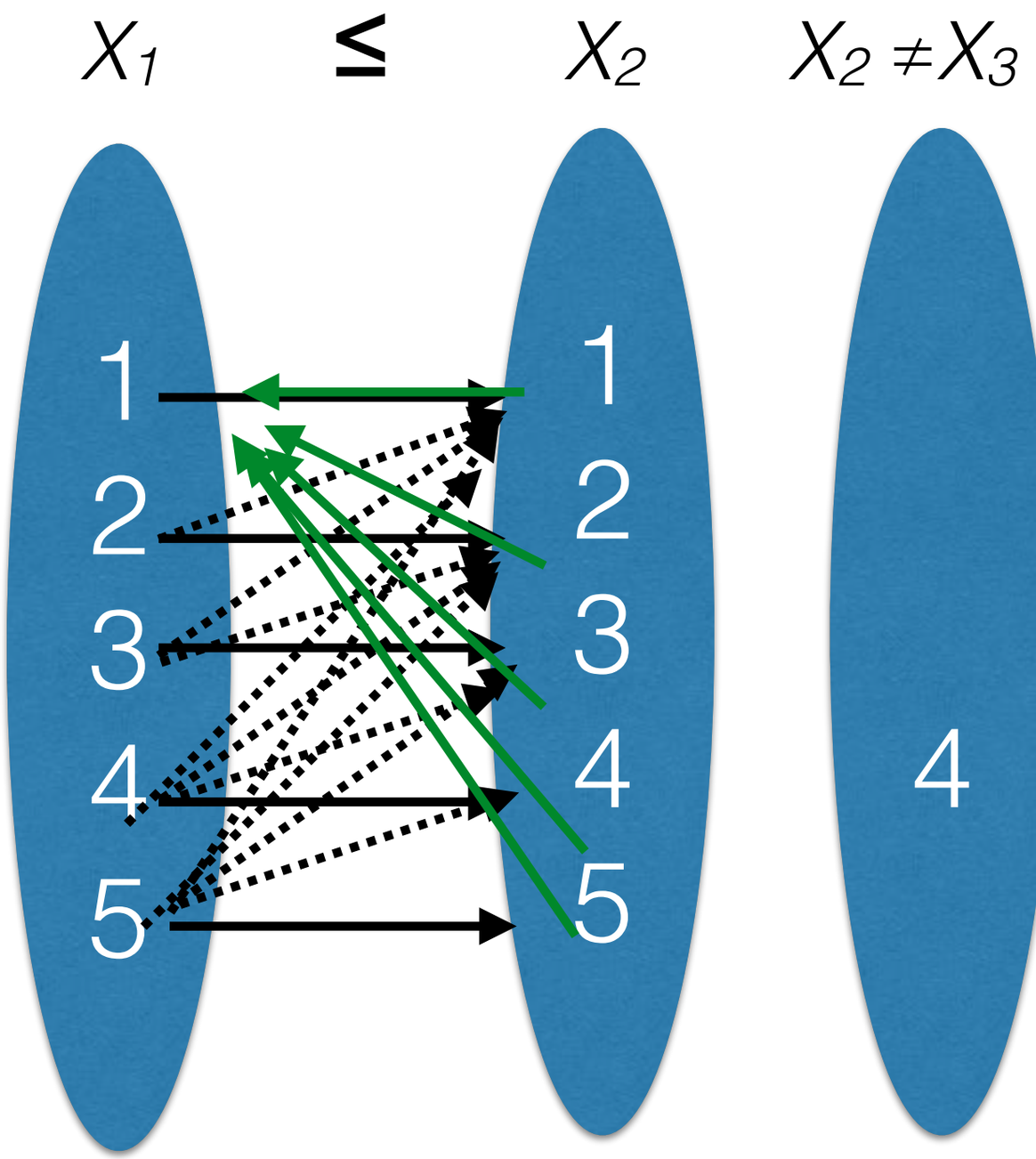
- Function Revise traverses *$D(X_1)$ x $D(X_2)$*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

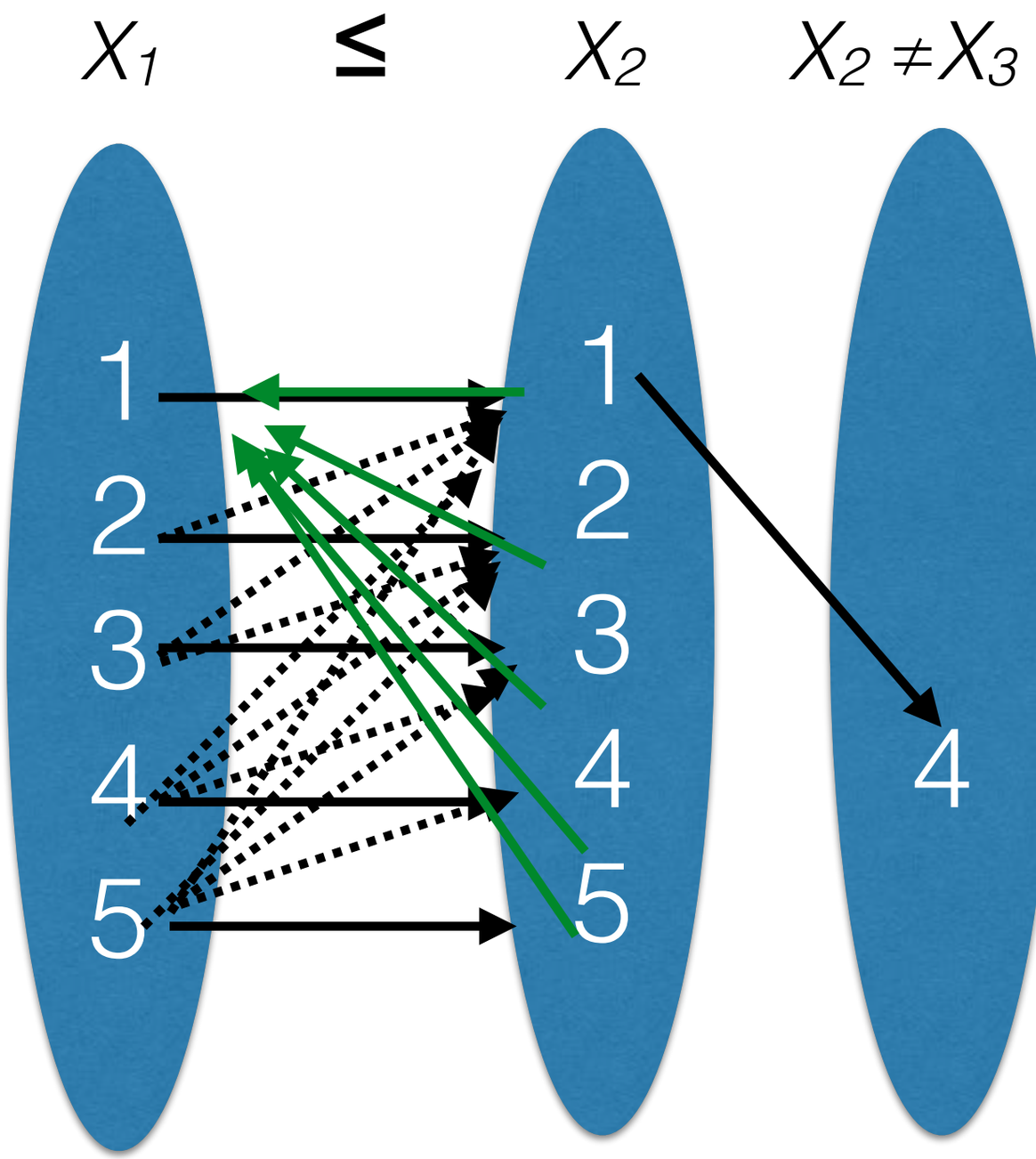
- Function Revise traverses *D(X_1) x D(X_2)*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

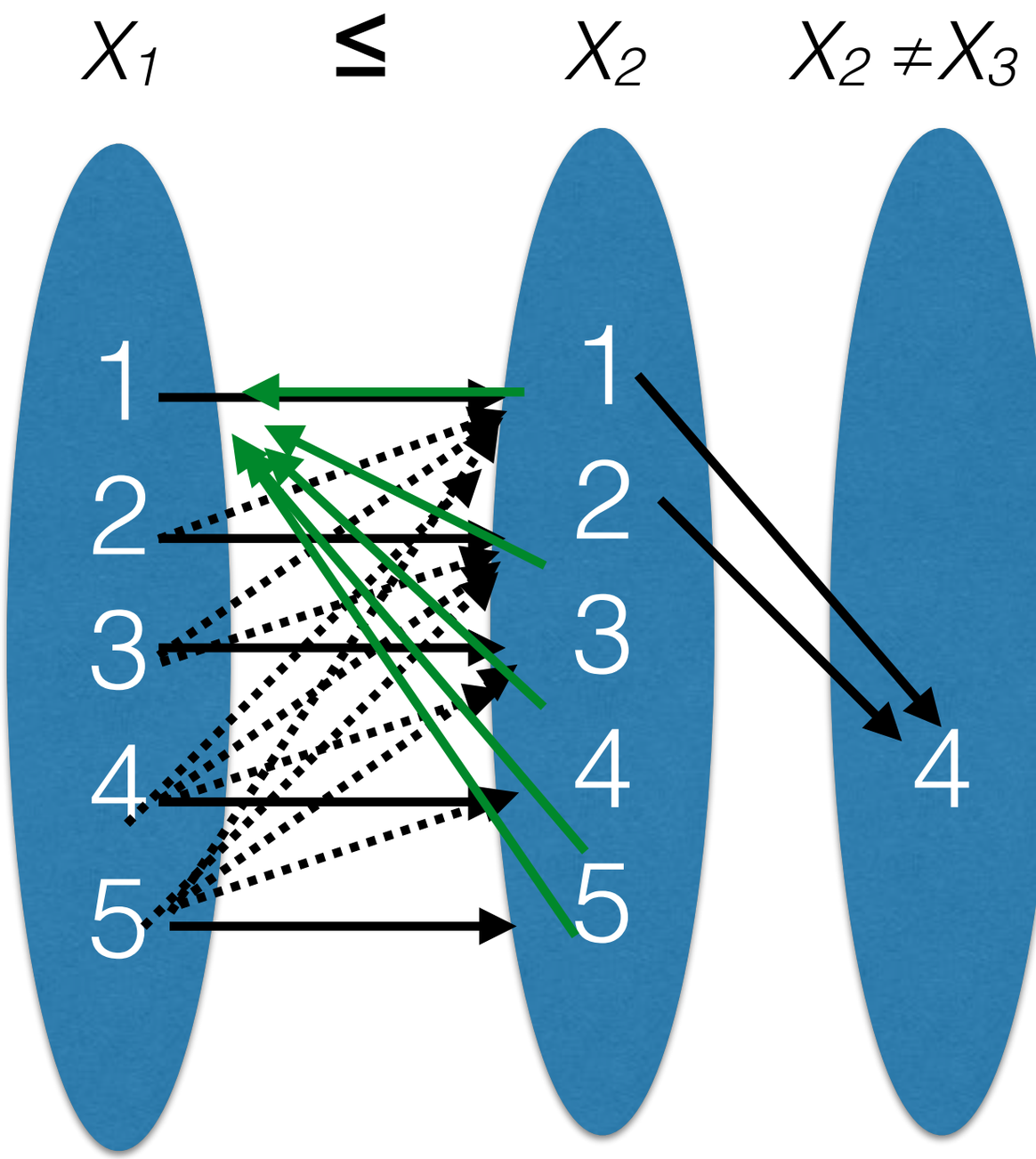
- Function Revise traverses *D(X_1) x D(X_2)*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

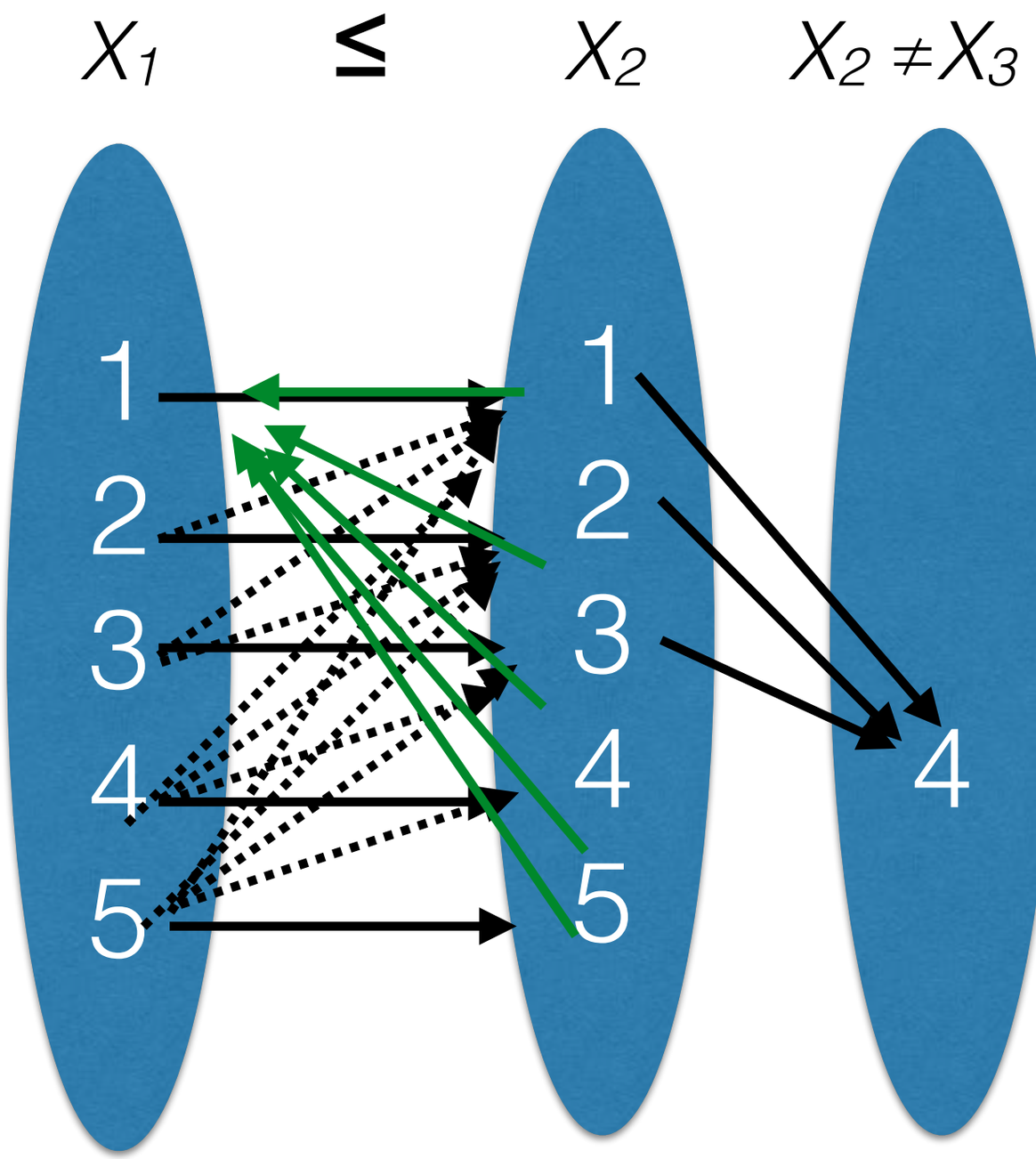
- Function Revise traverses *D(X_1) x D(X_2)*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

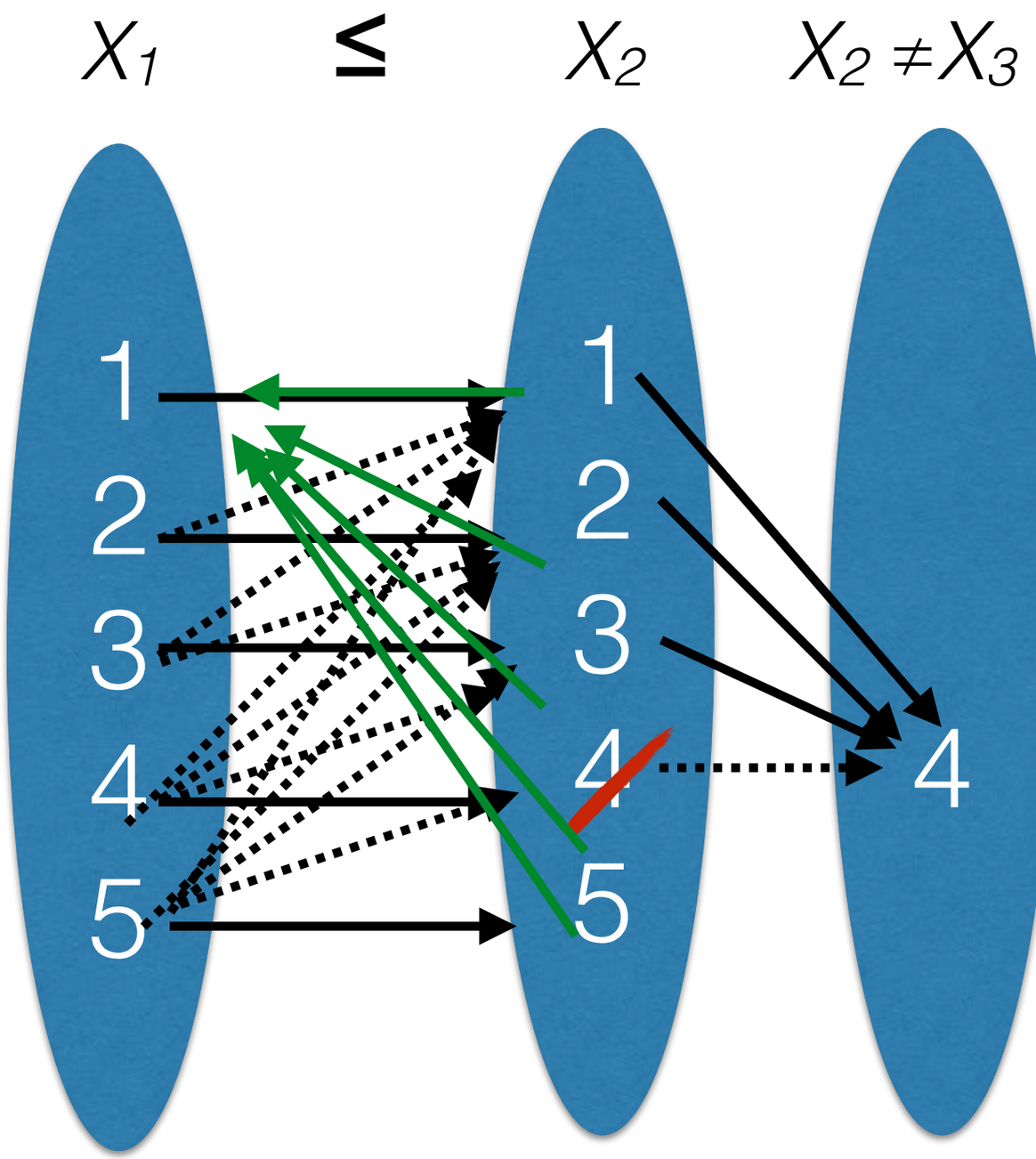
- Function Revise traverses *D(X_1) x D(X_2)*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

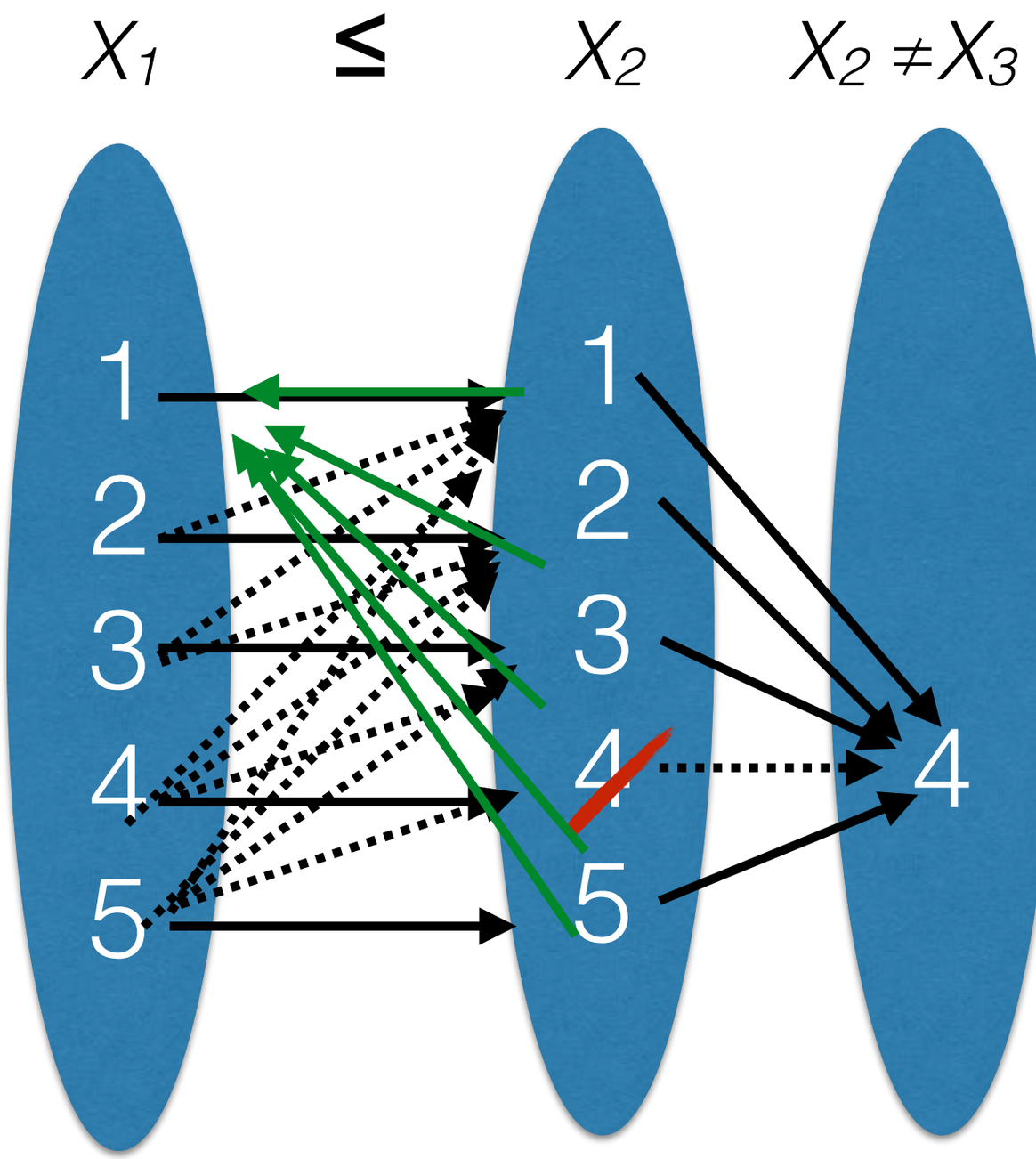
- Function Revise traverses *D(X_1) x D(X_2)*



AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

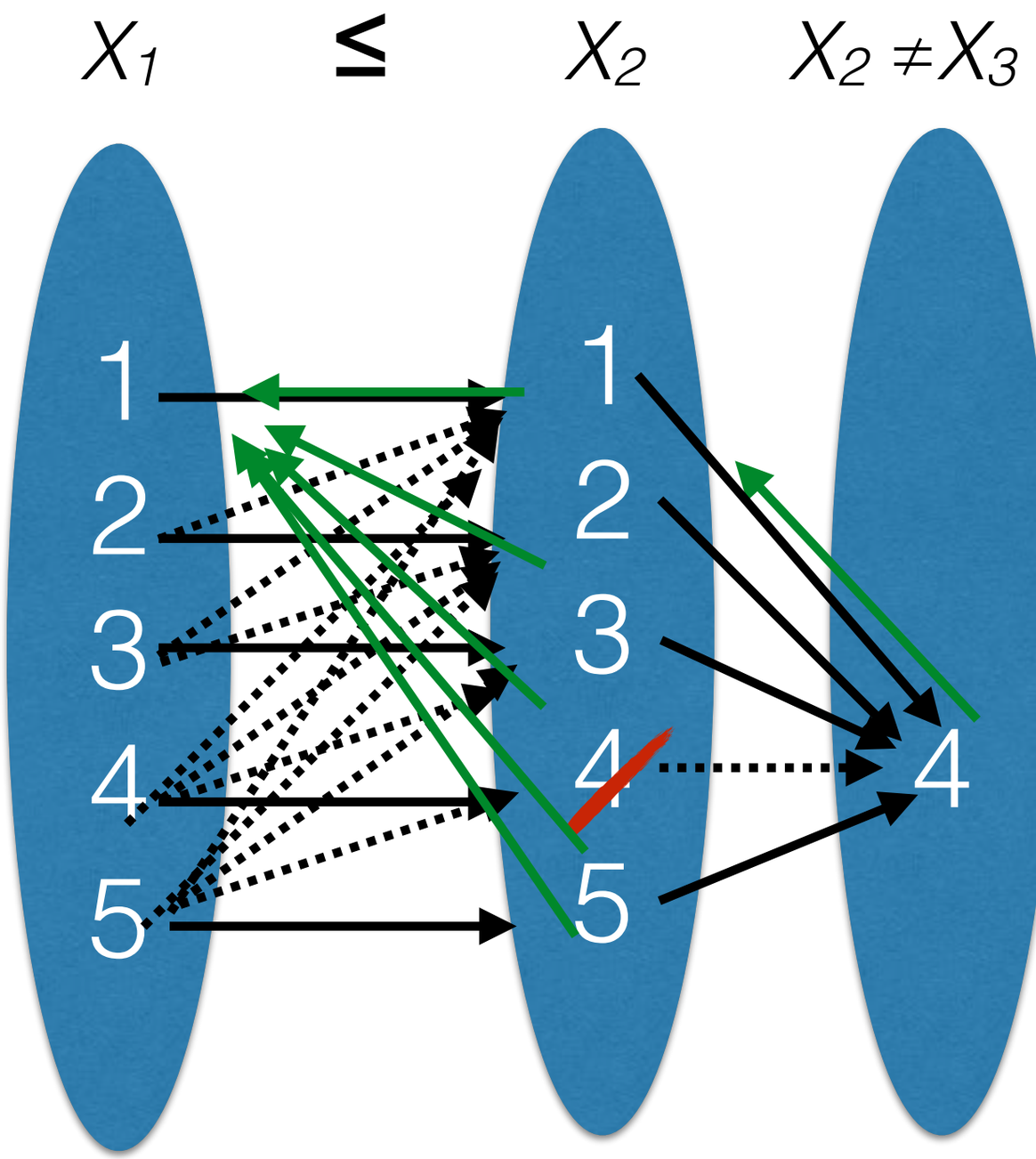
- Function Revise traverses *D(X_1) x D(X_2)*



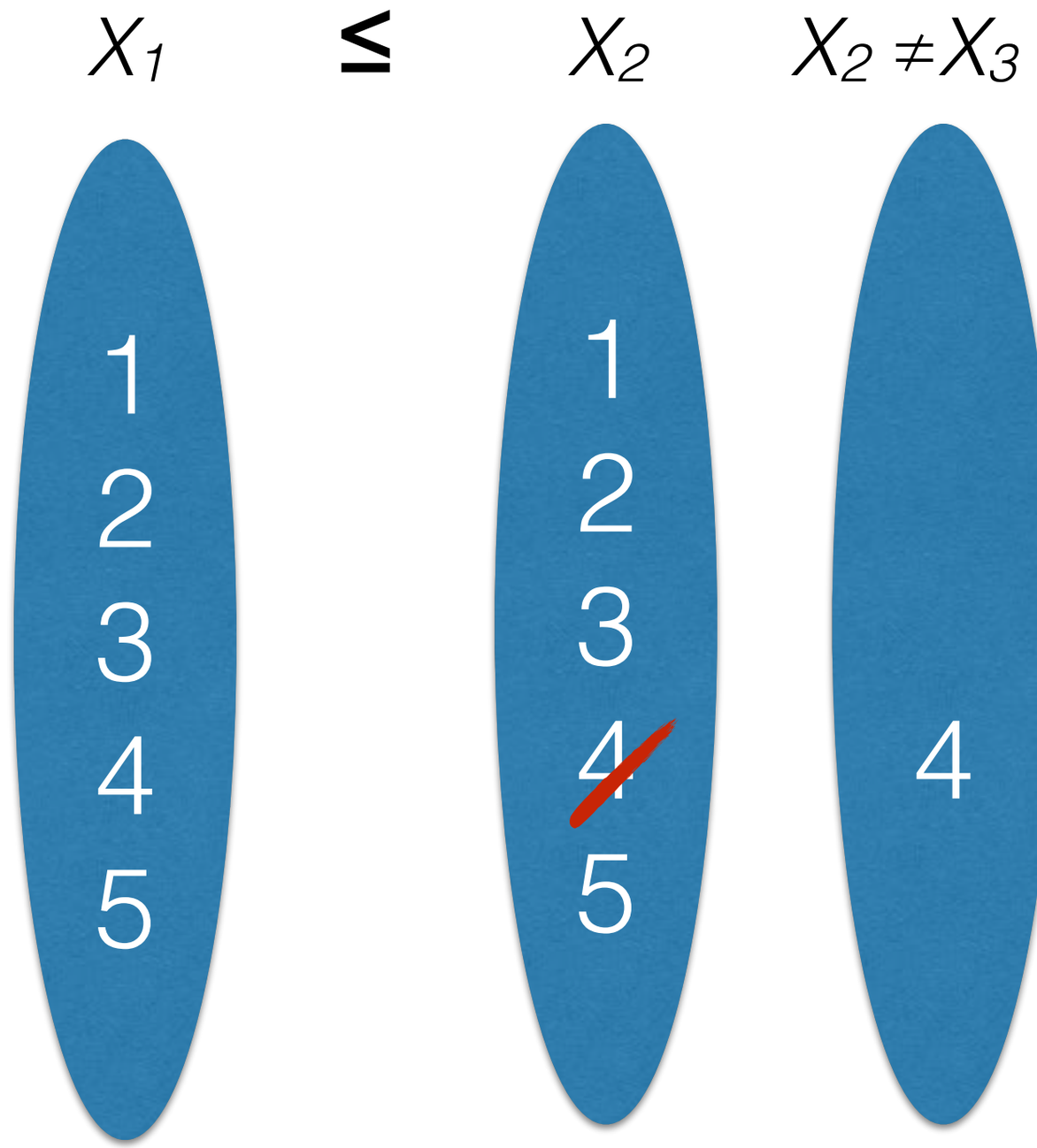
AC3

- Uses the queue Q as we did before: Q={c_{12},c_{21},c_{23},c_{32}}

- Function Revise traverses *D(X_1) x D(X_2)*

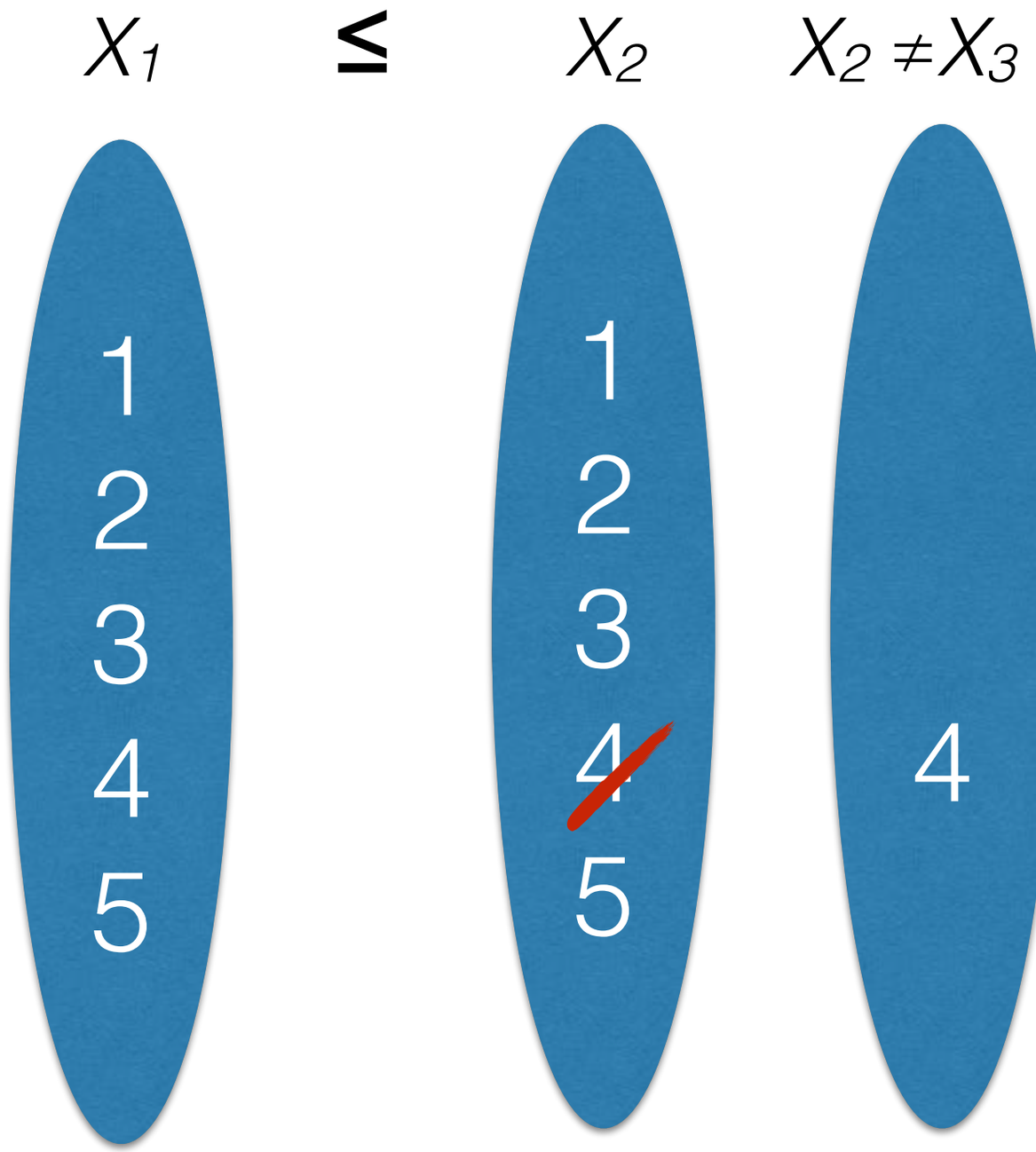


AC3



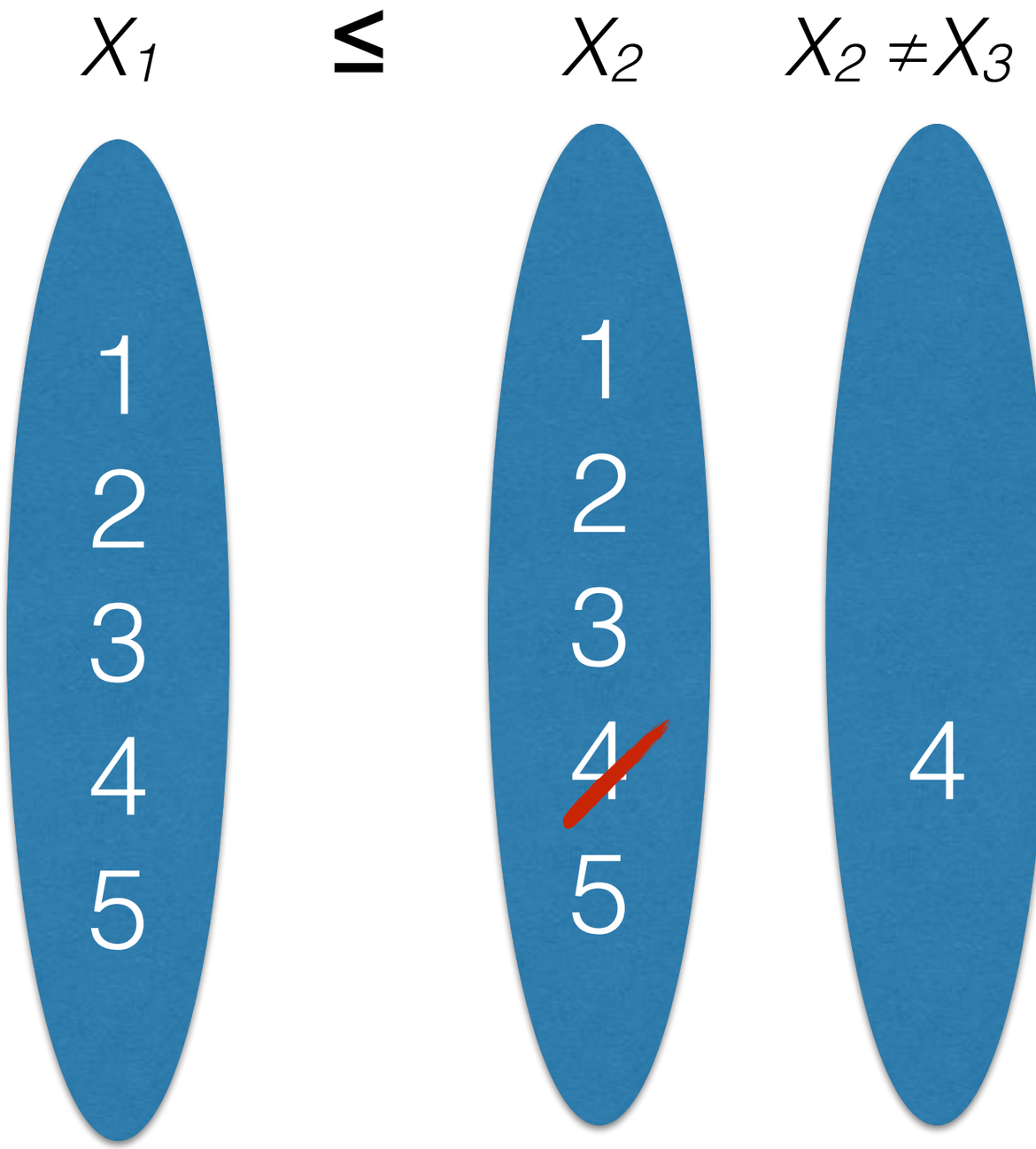
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)



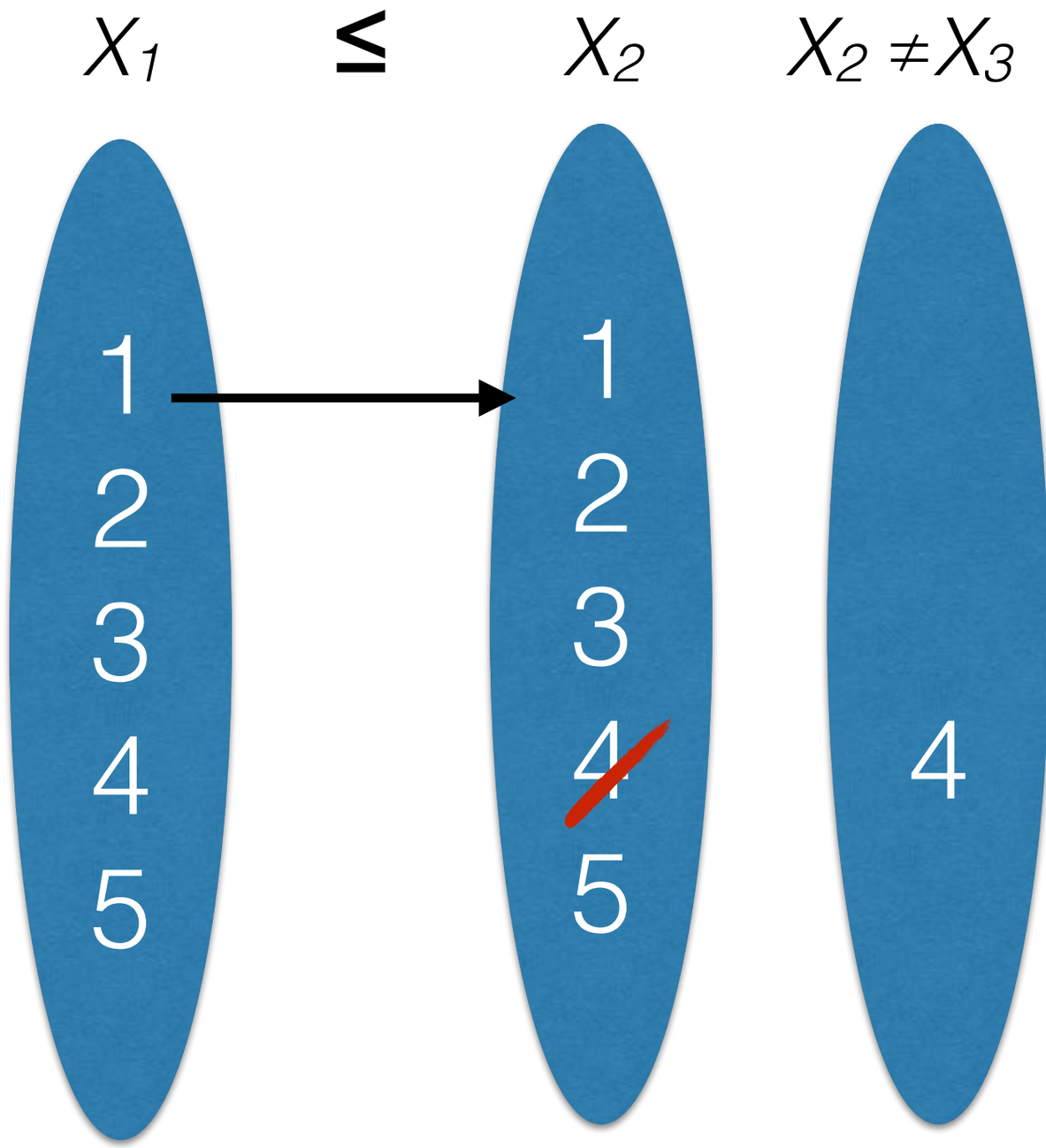
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before



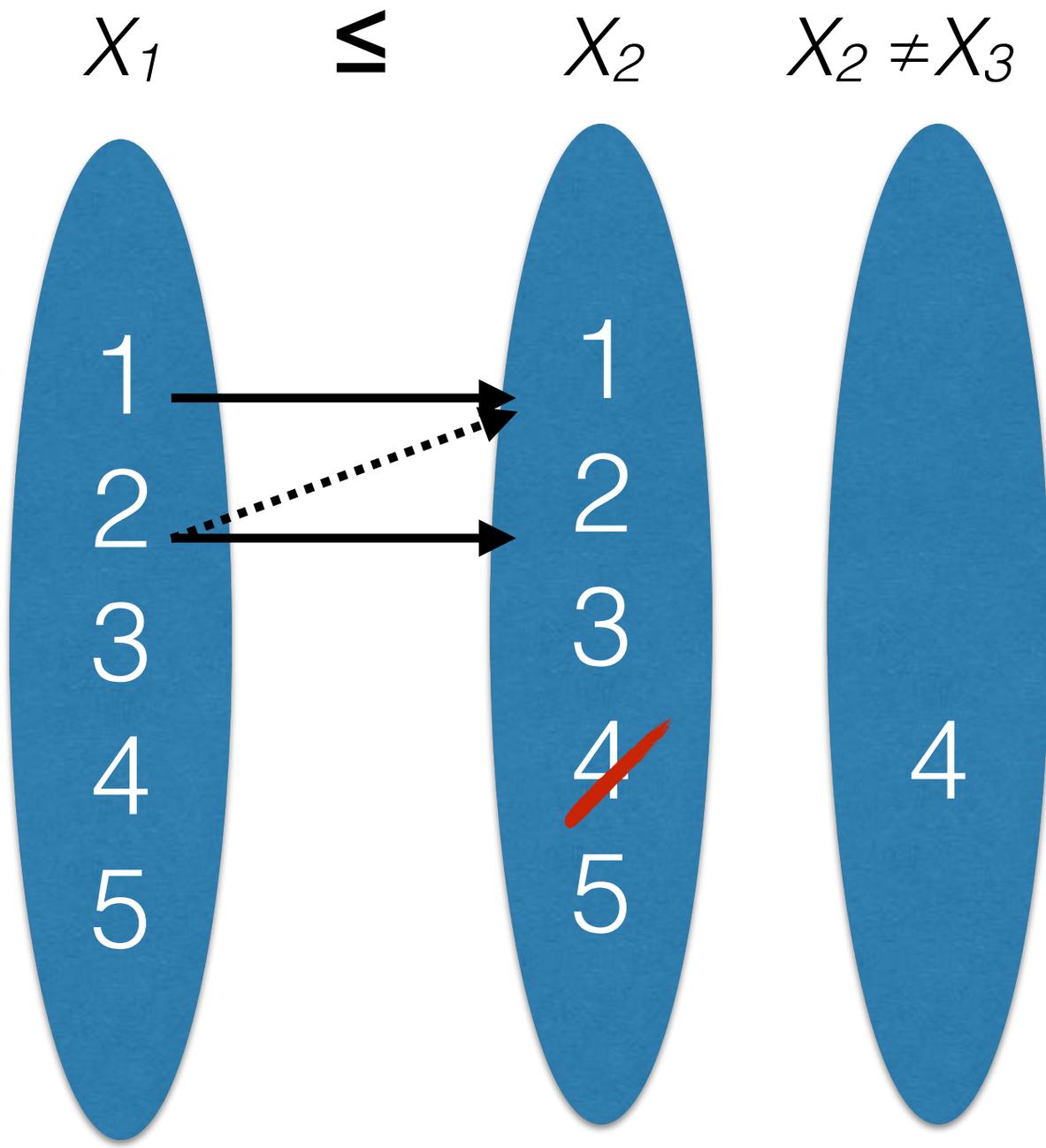
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before



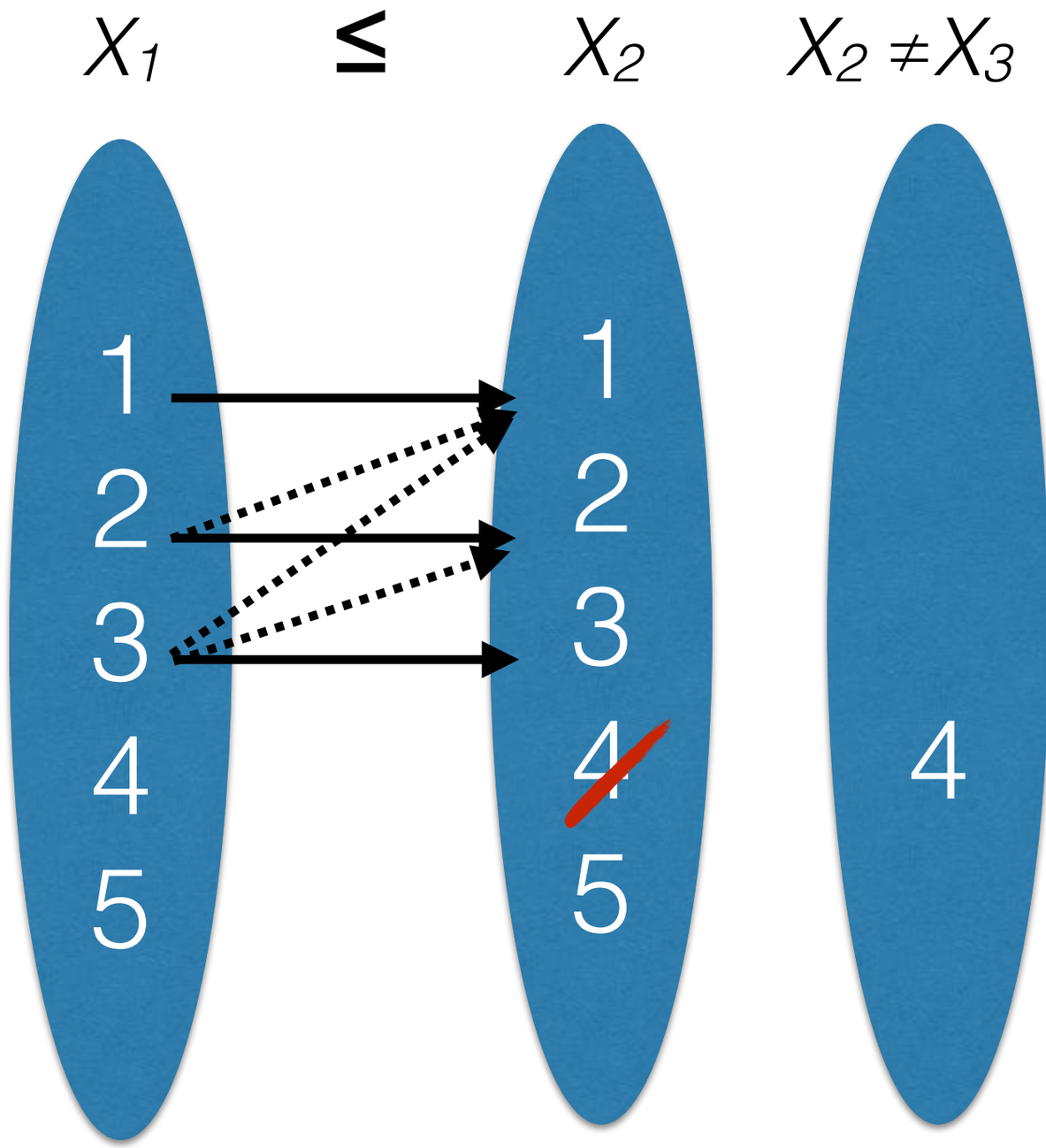
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before



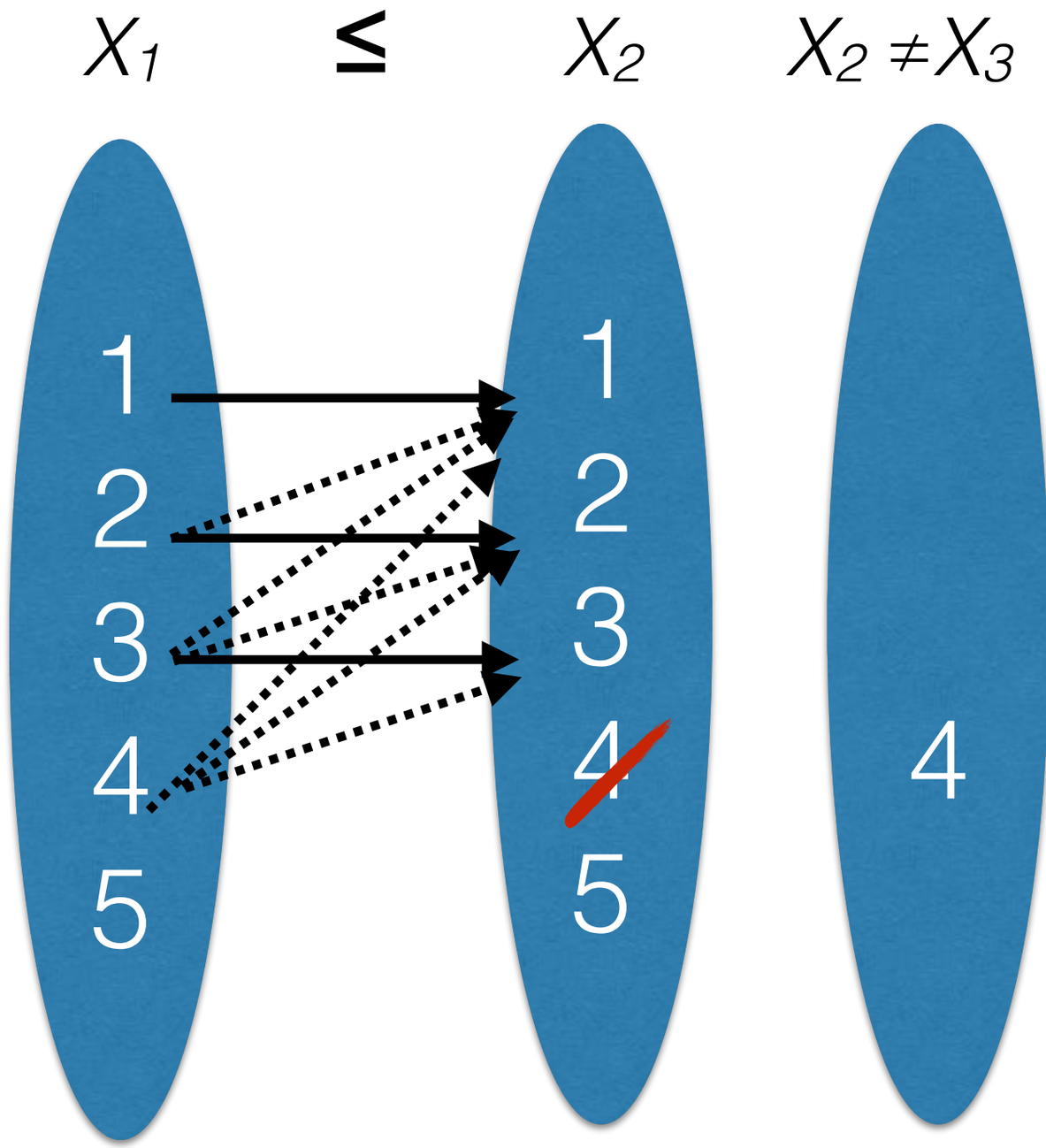
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before



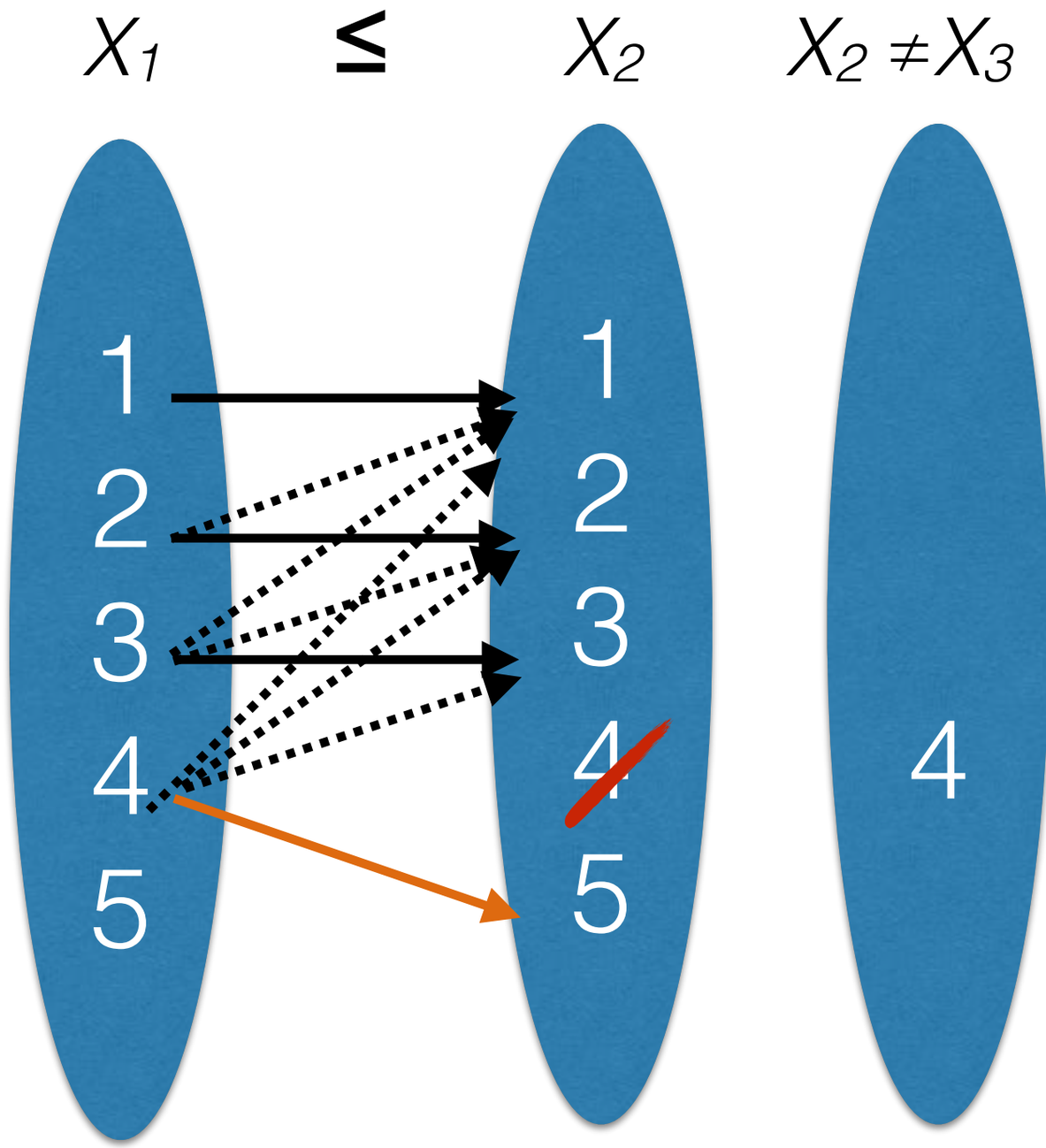
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before



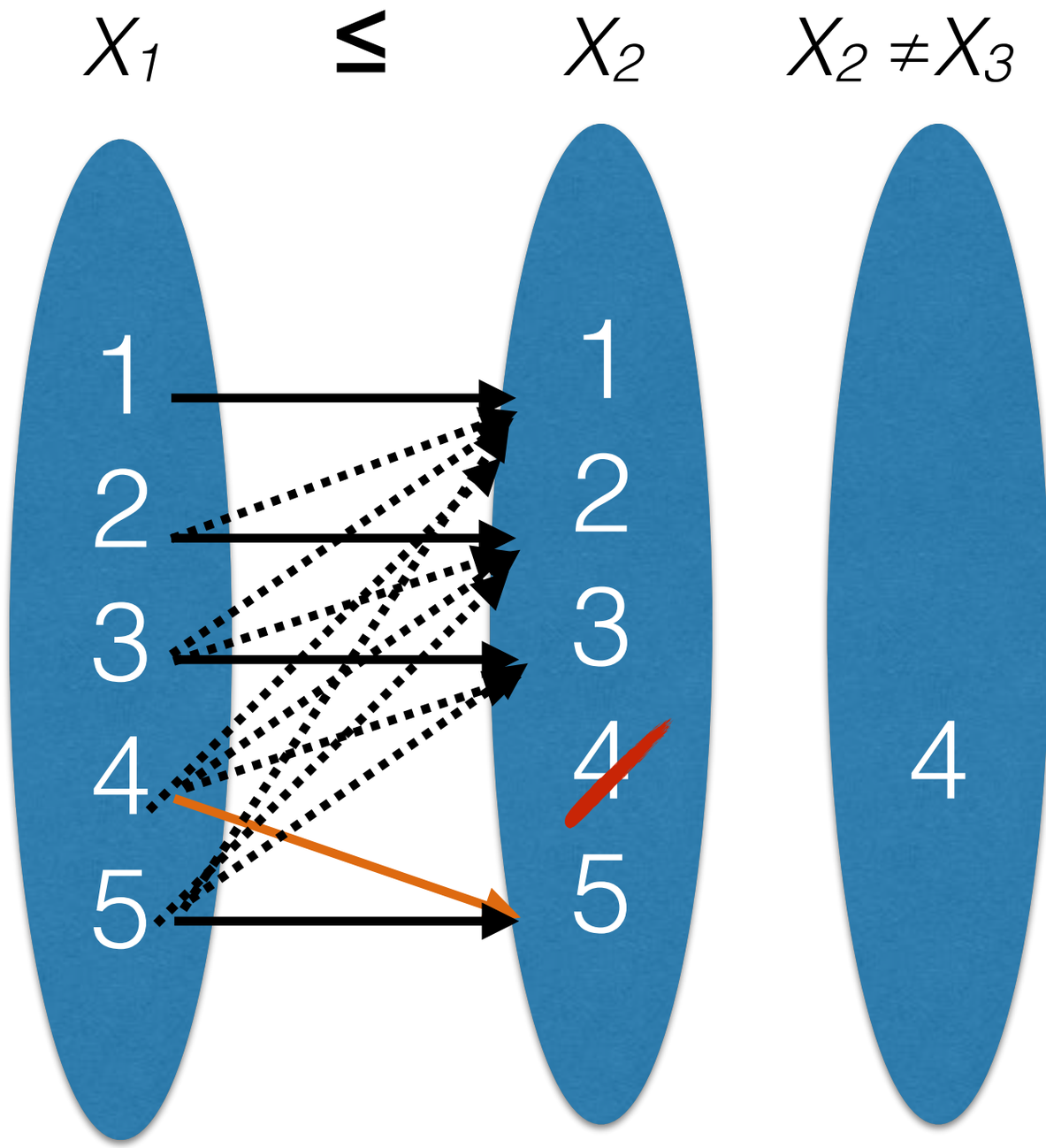
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before



AC3

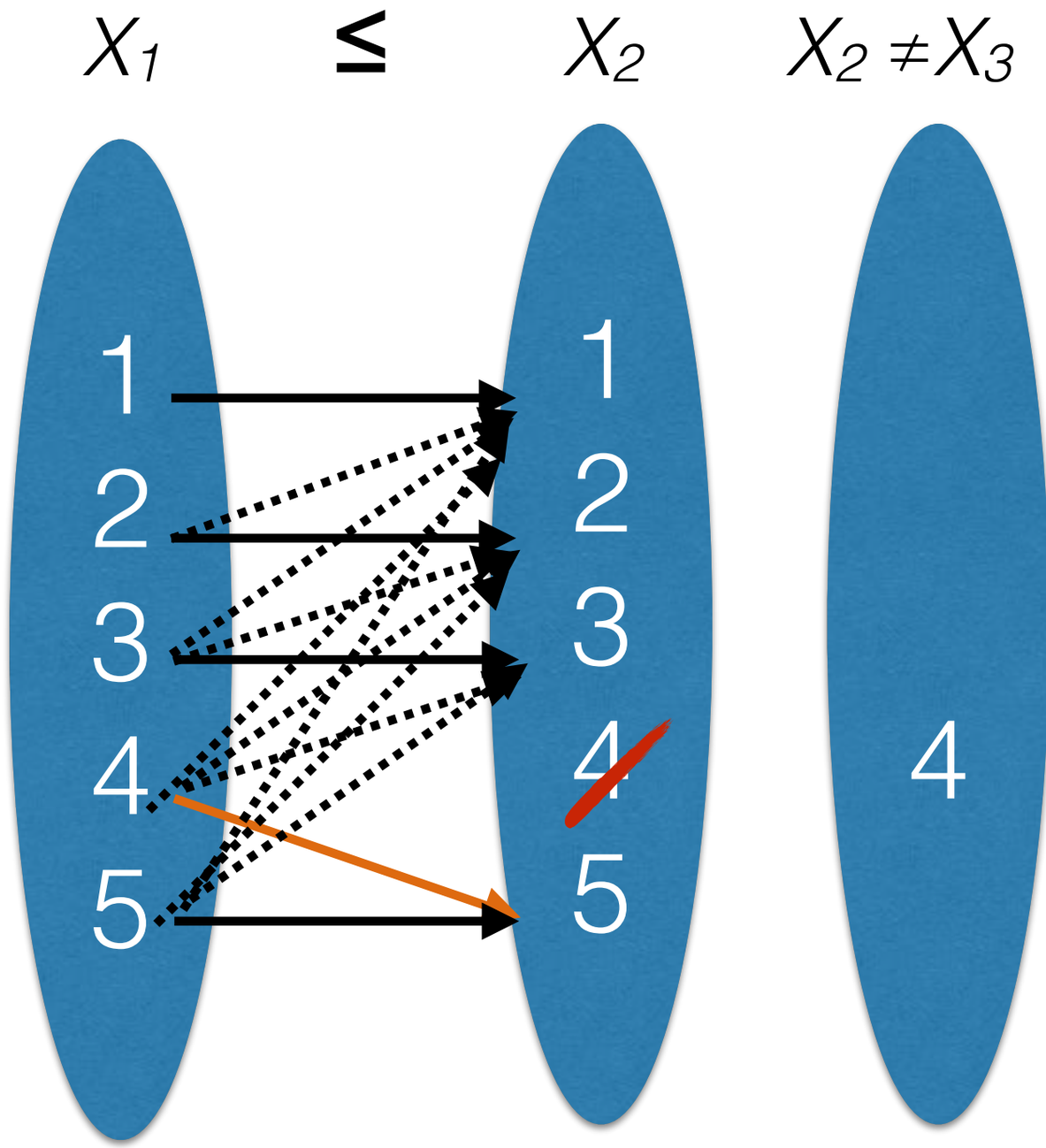
- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before



AC3

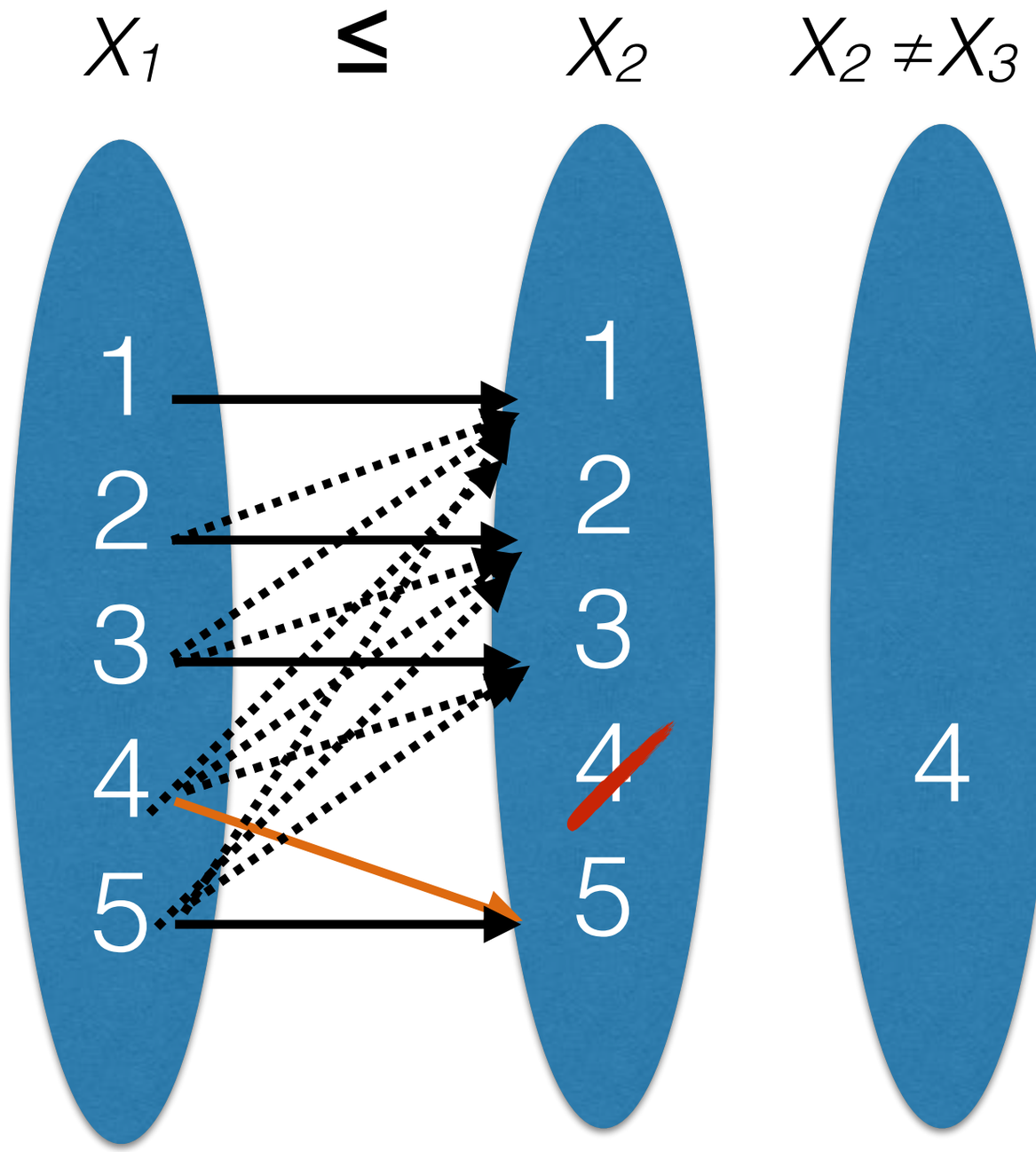
- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before

➡ **AC3 redoes what it already did**



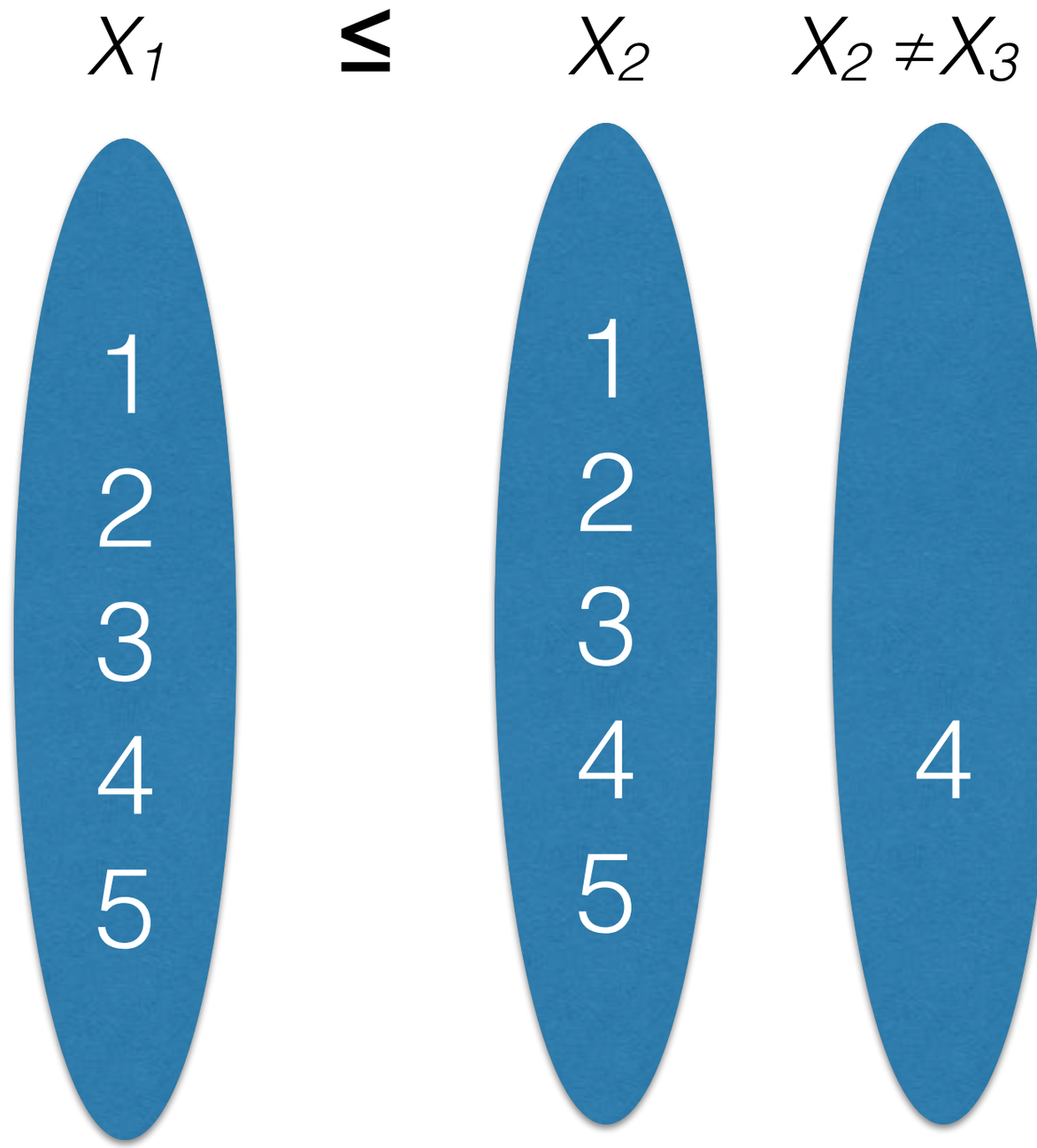
AC3

- We have to put X1 ≤X_2 in Q because of the removal of (X_2,4)
- AC3 forgot what it did before
- ➡ **AC3 redoes what it already did**
- AC3 is not optimal



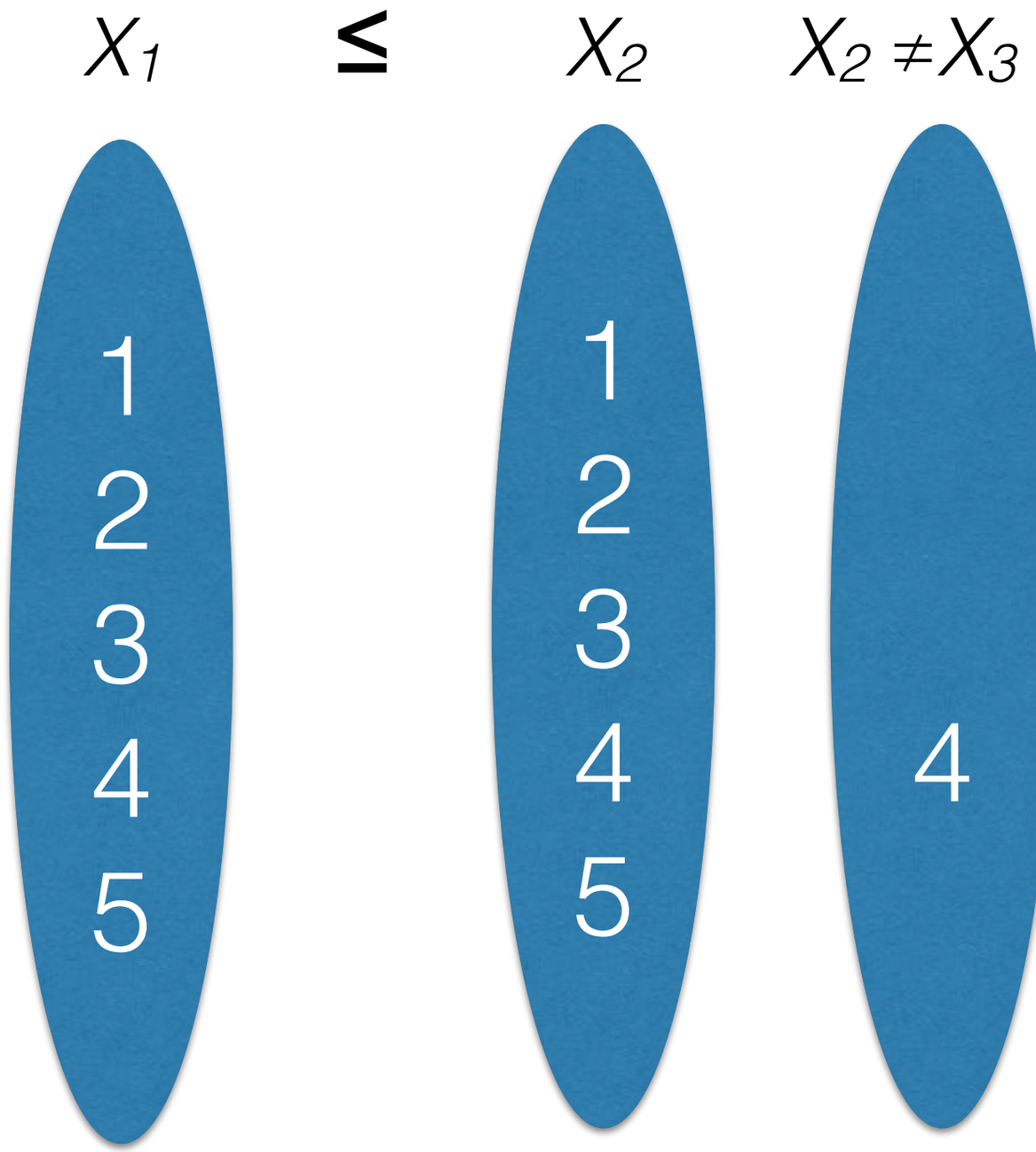
AC2001: an (easy) optimal algorithm

AC2001



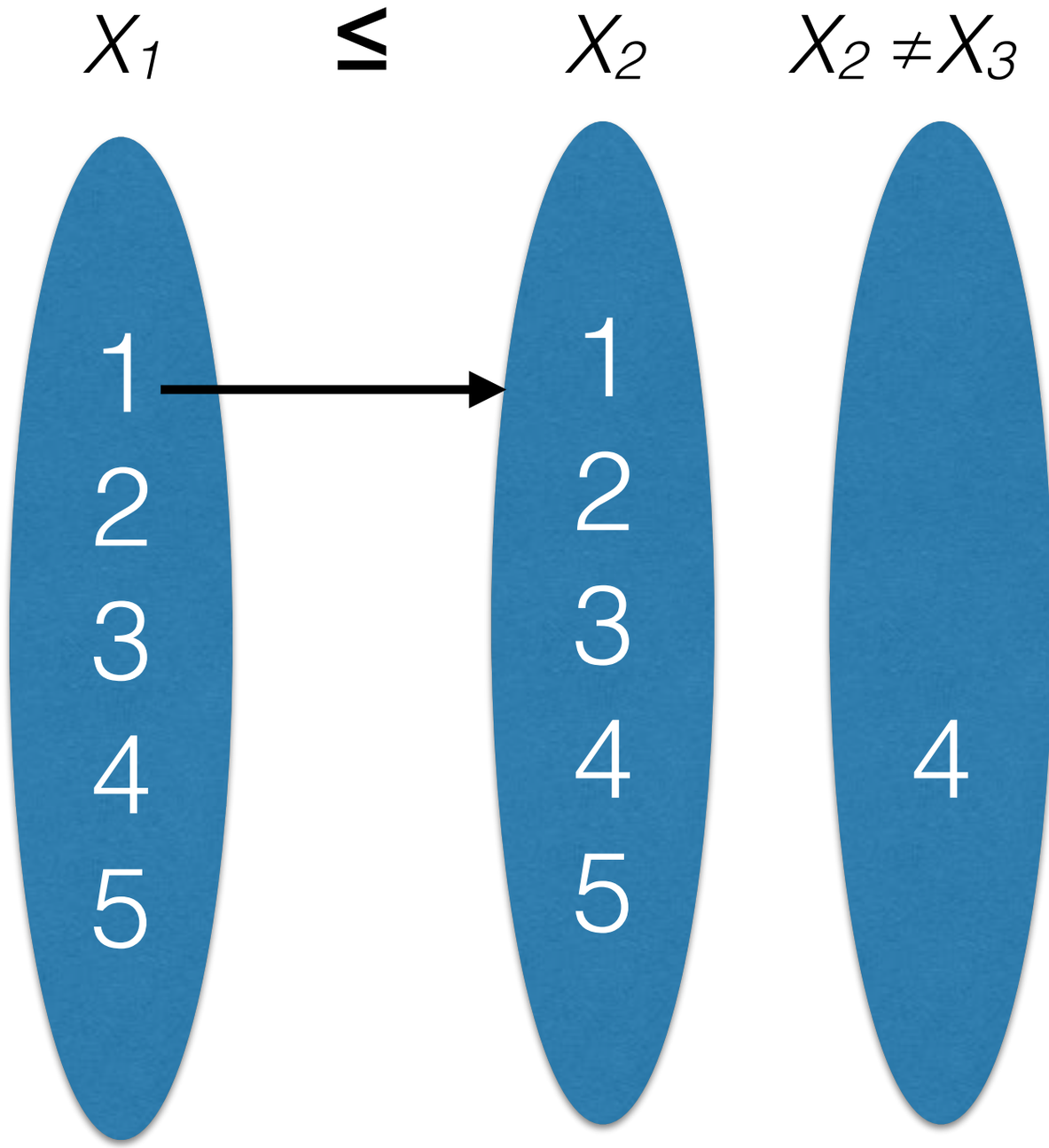
AC2001

- At the first call, AC2001 does like AC3



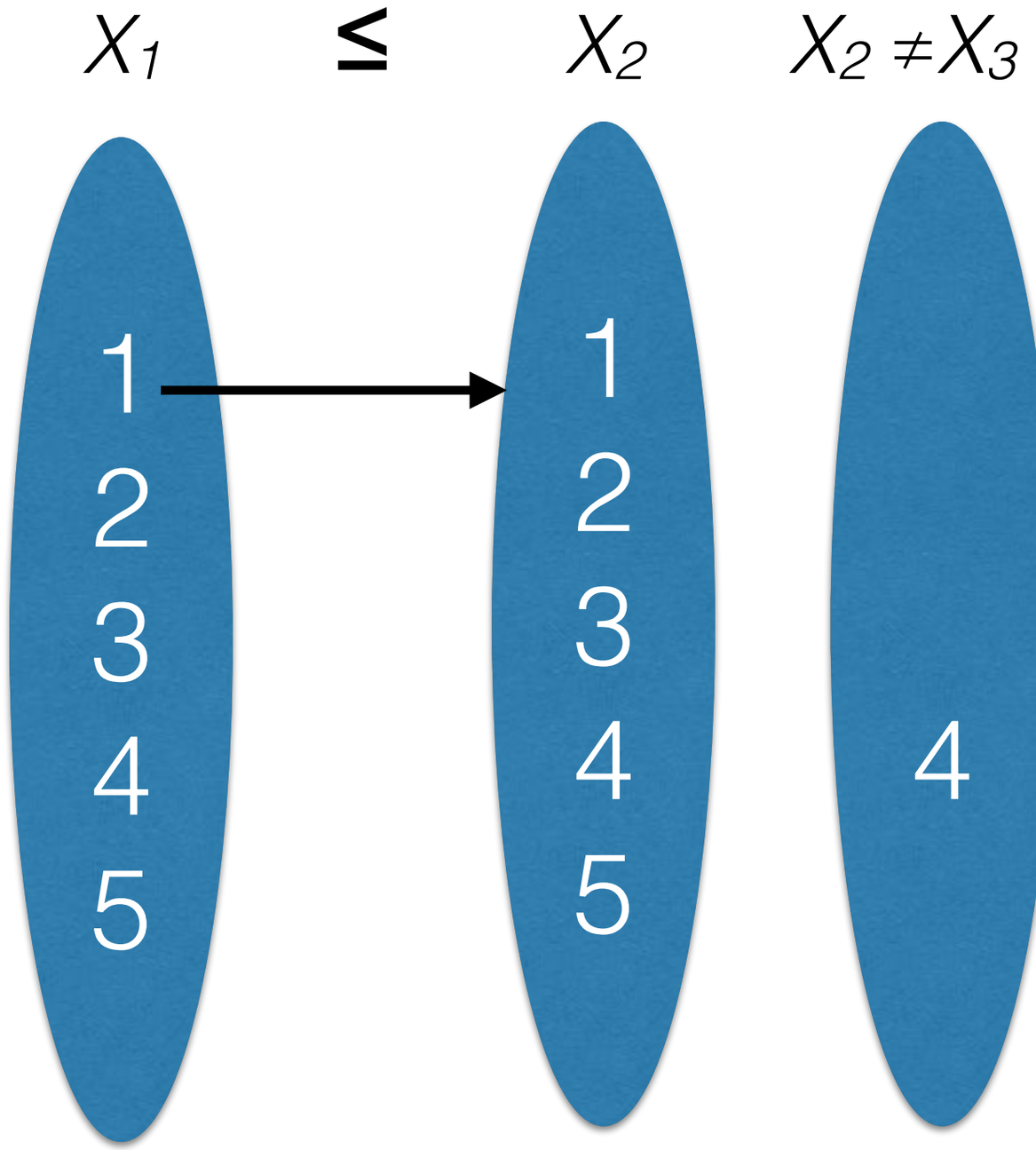
AC2001

- At the first call, AC2001 does like AC3



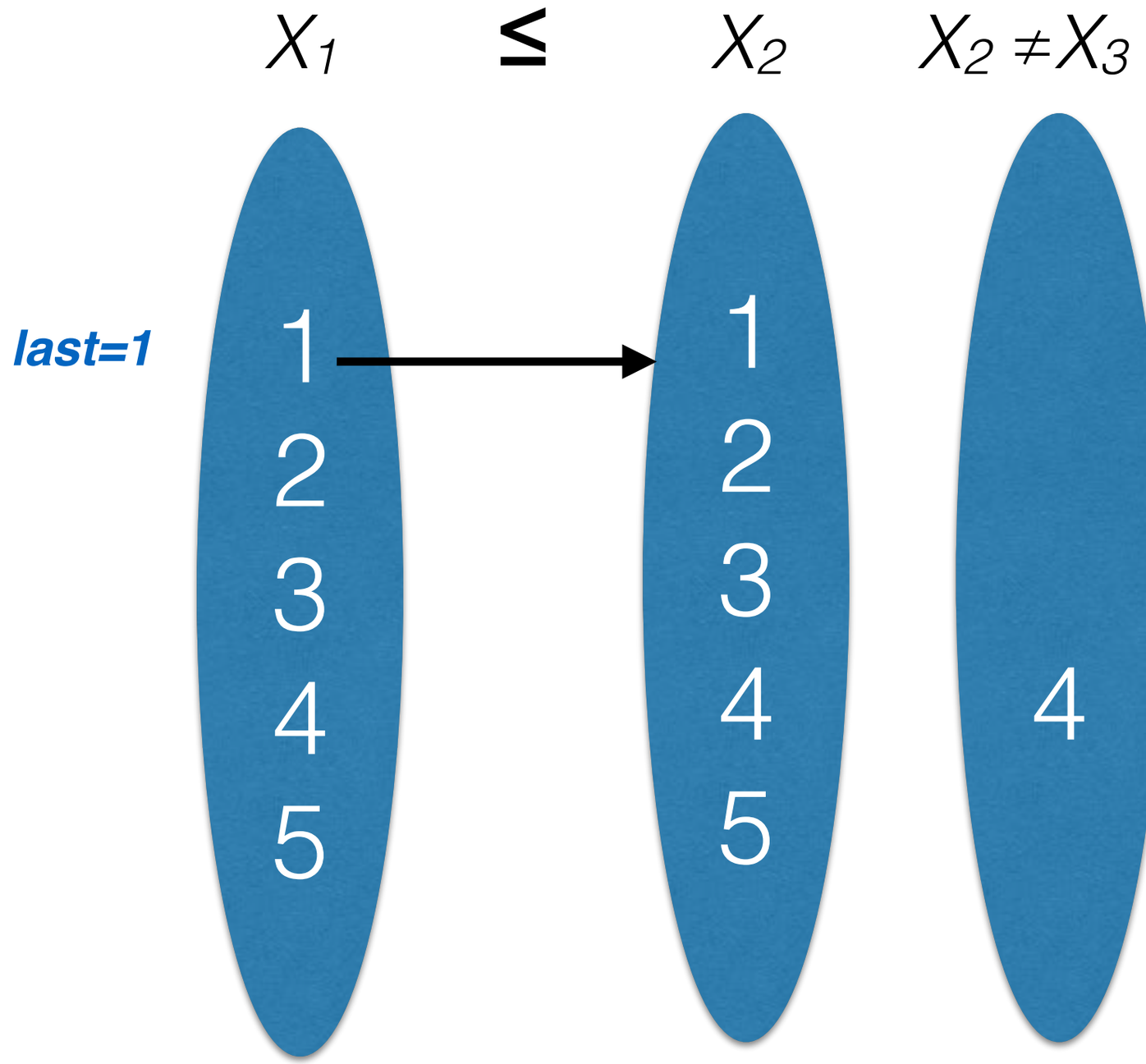
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked



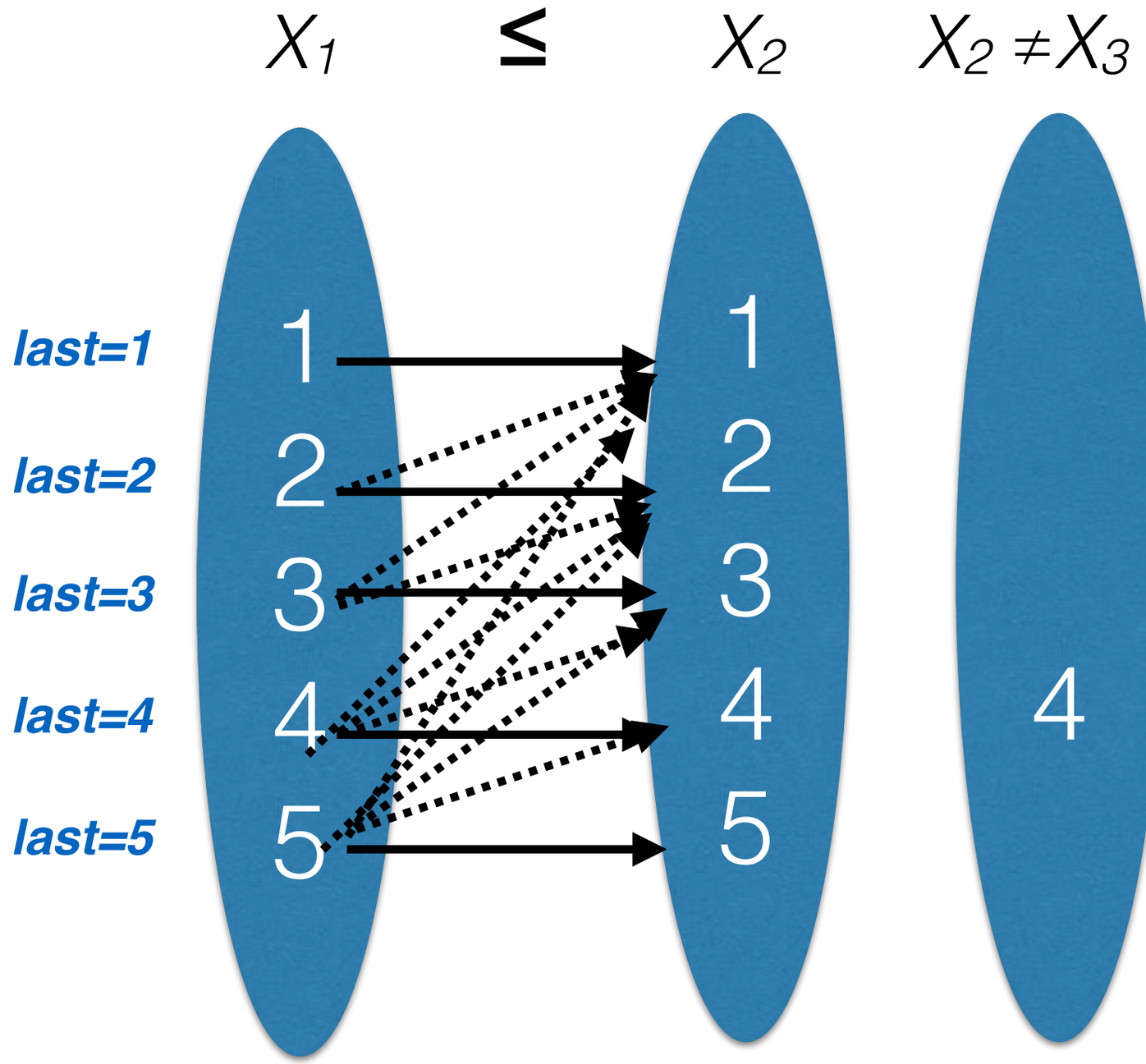
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked



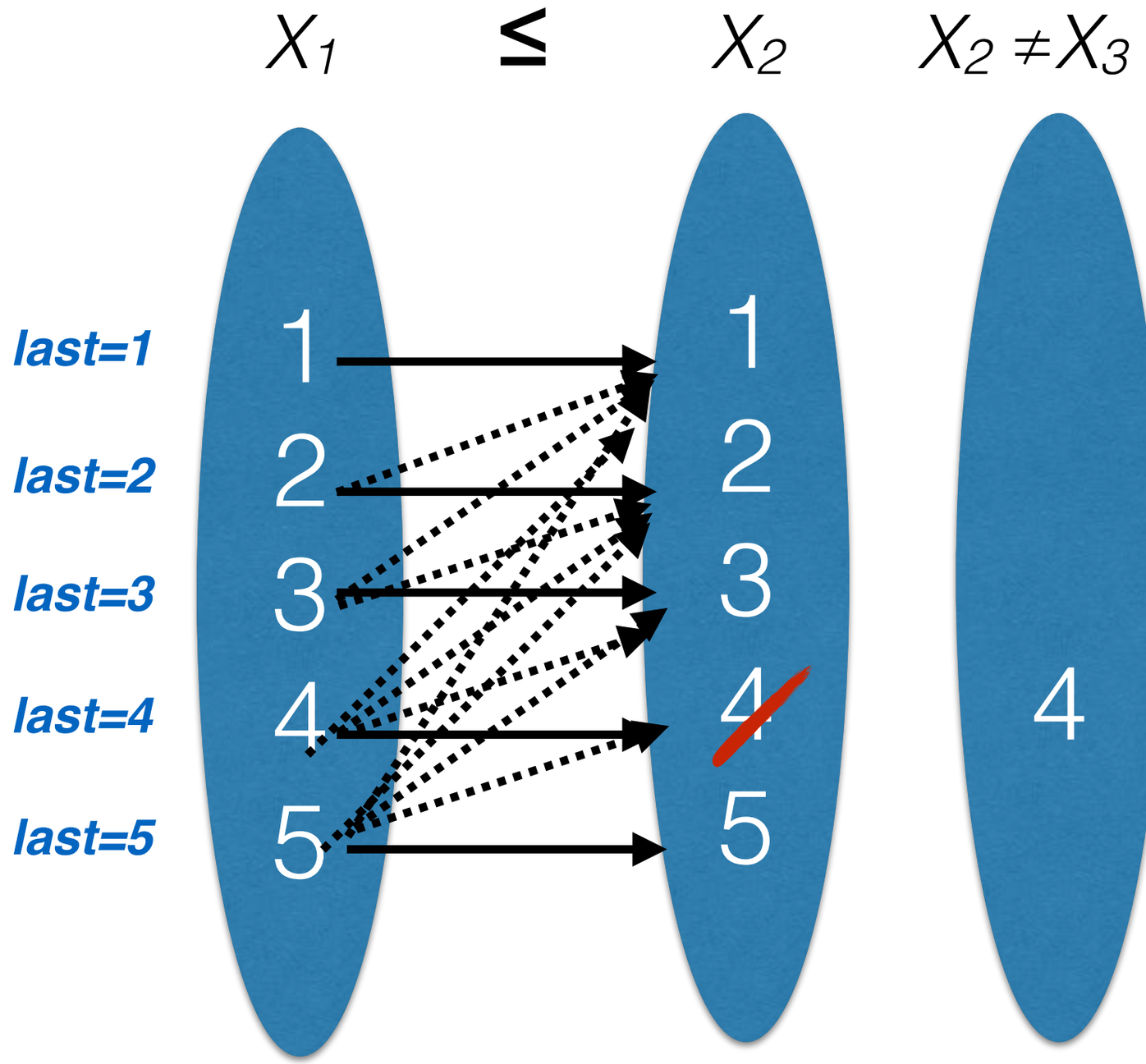
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked



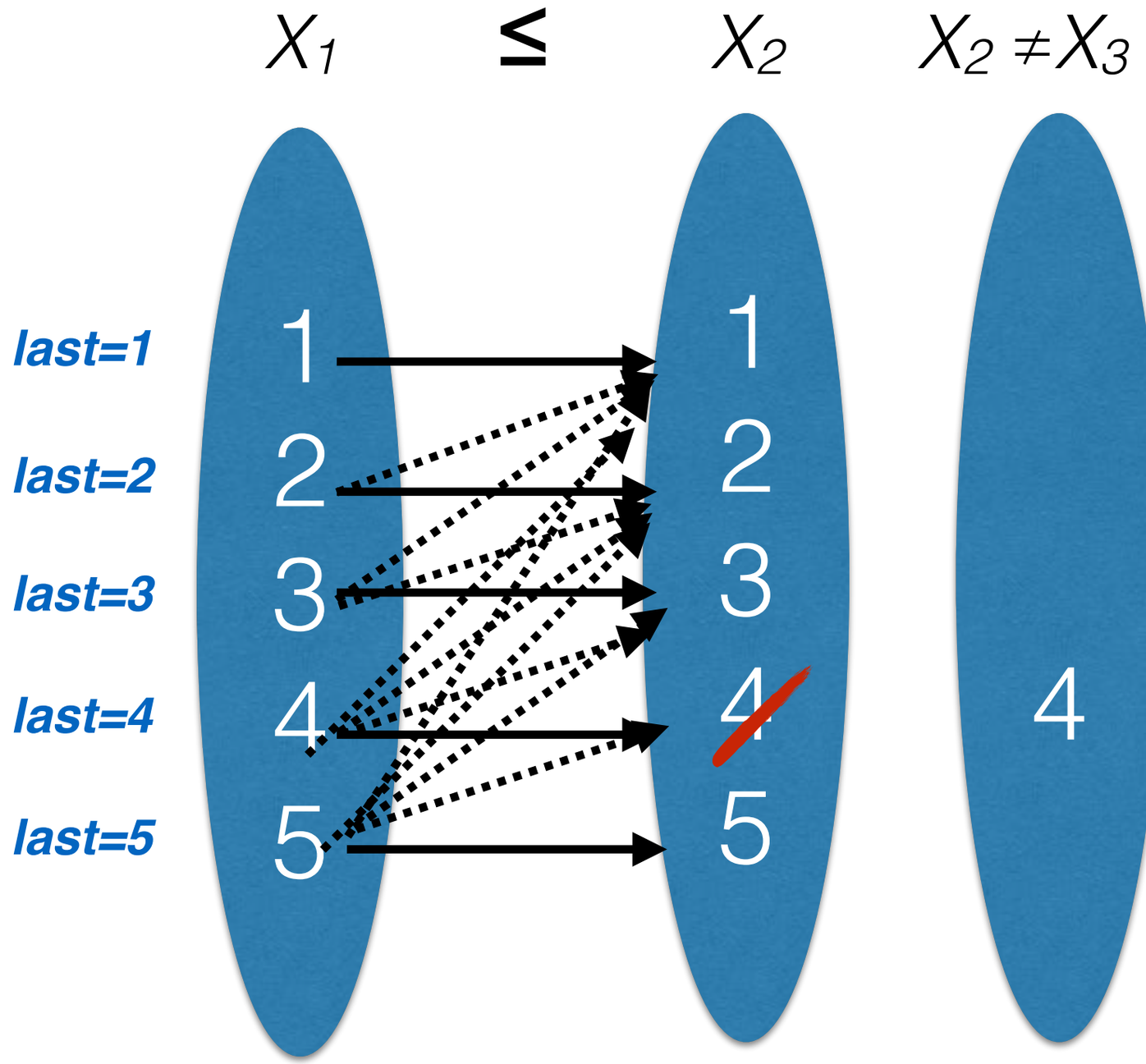
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked



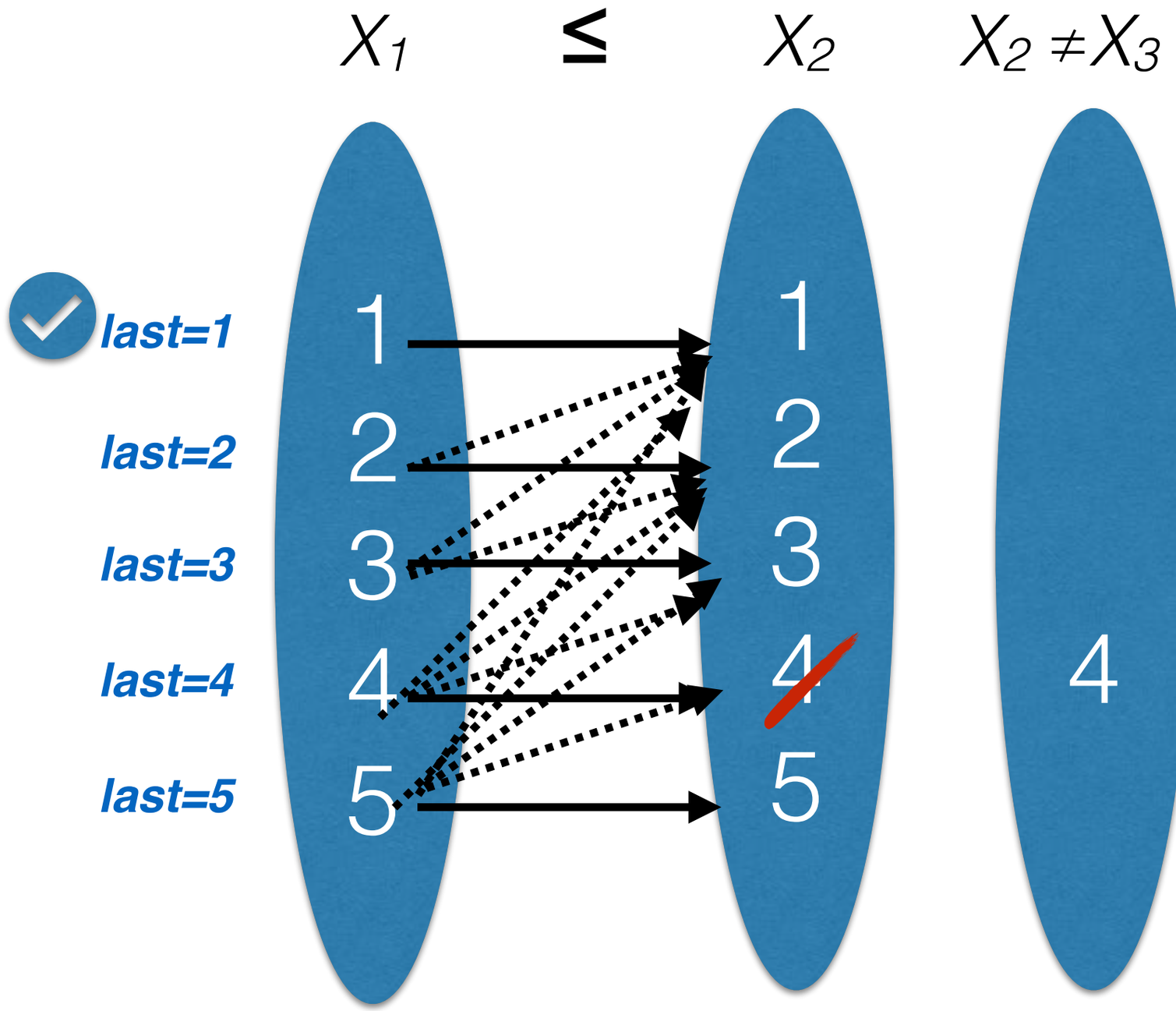
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



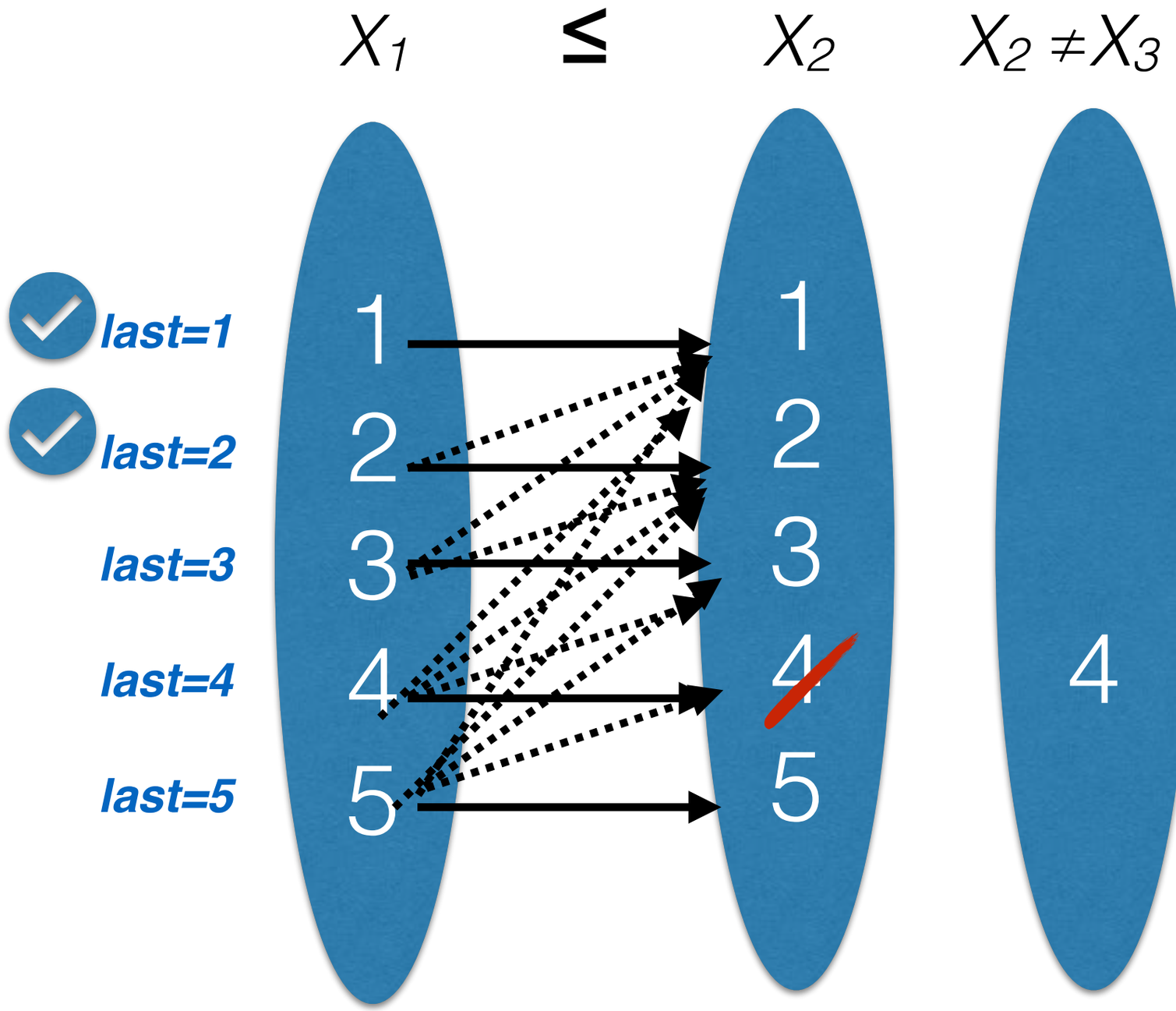
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



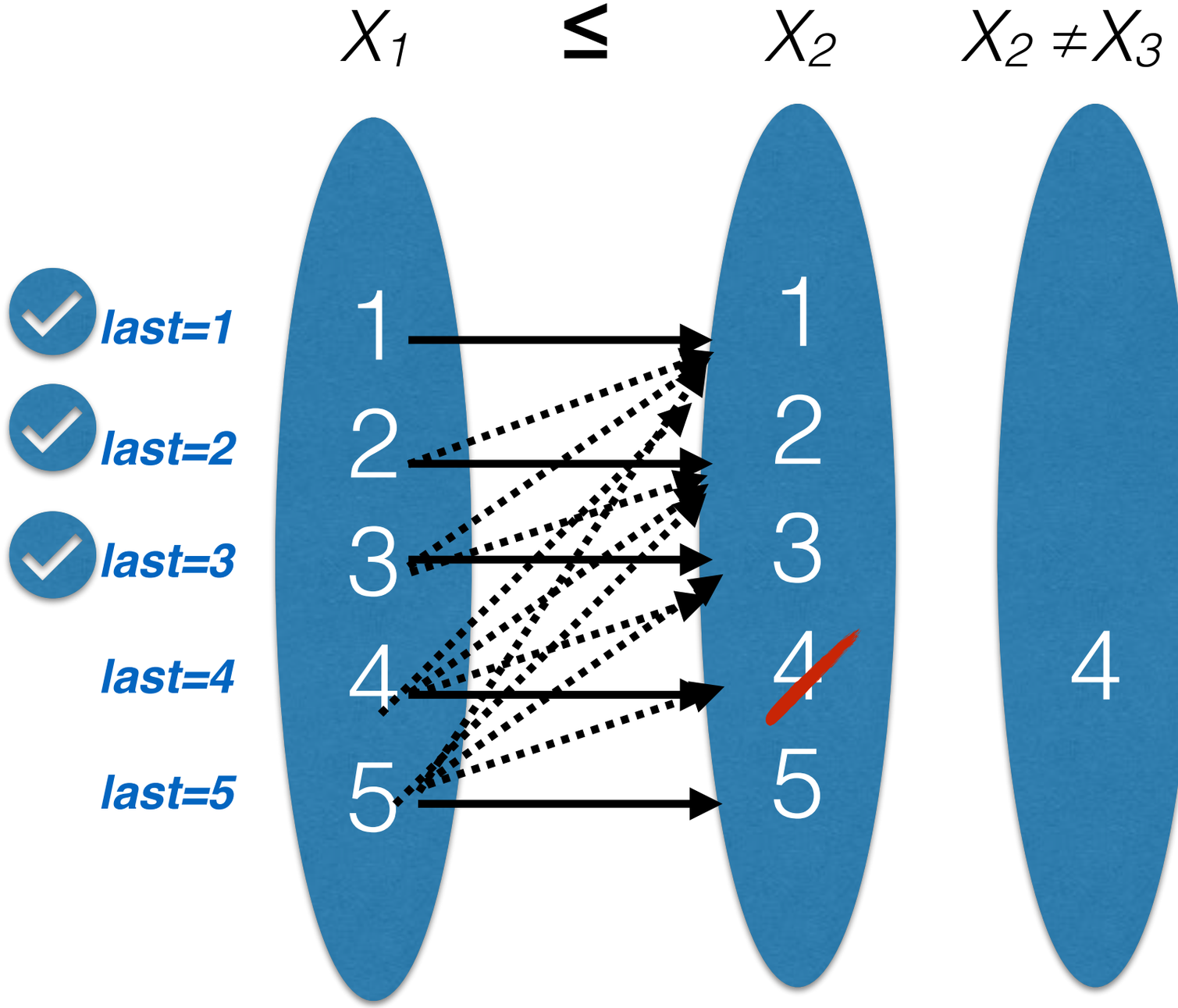
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



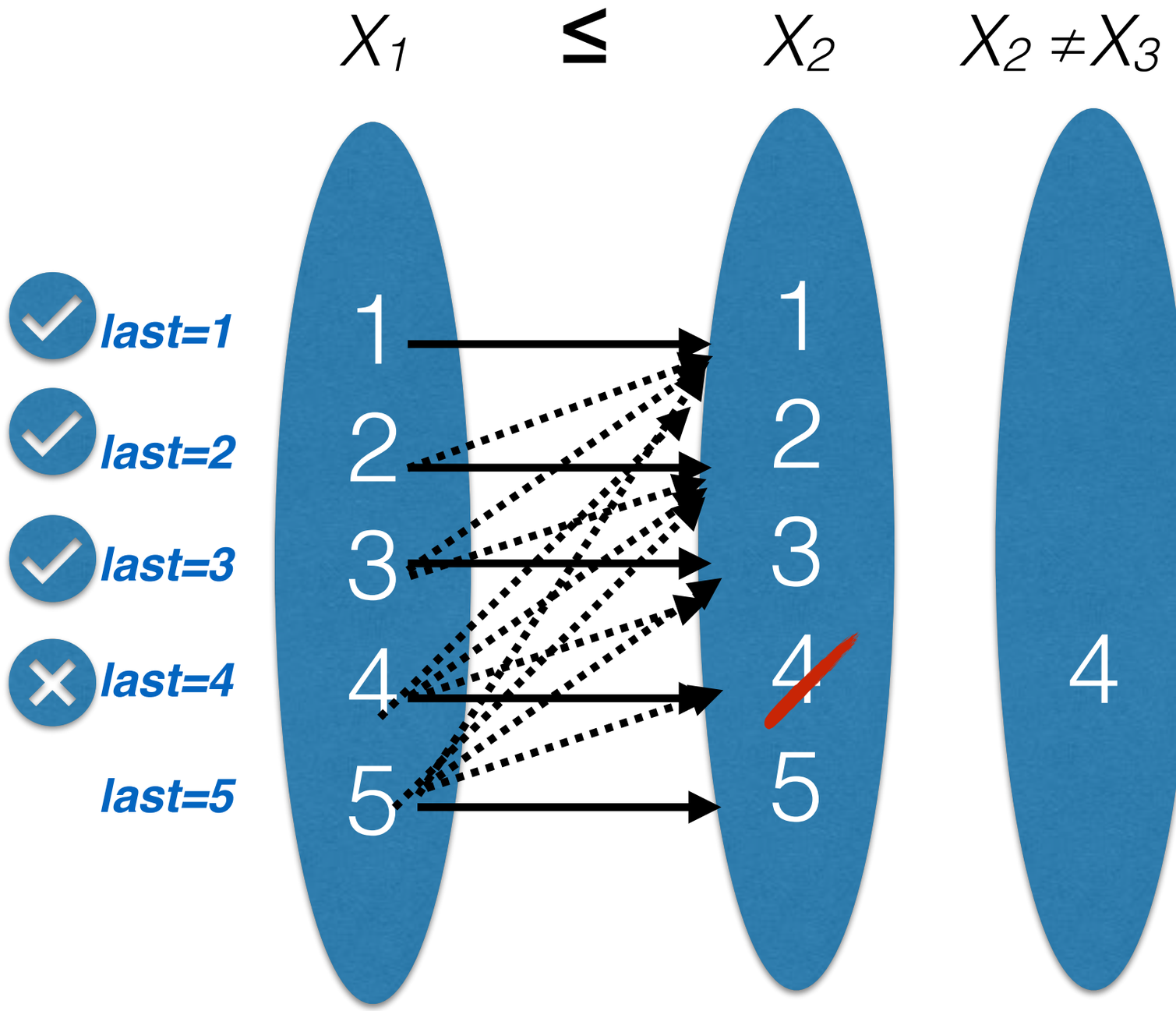
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



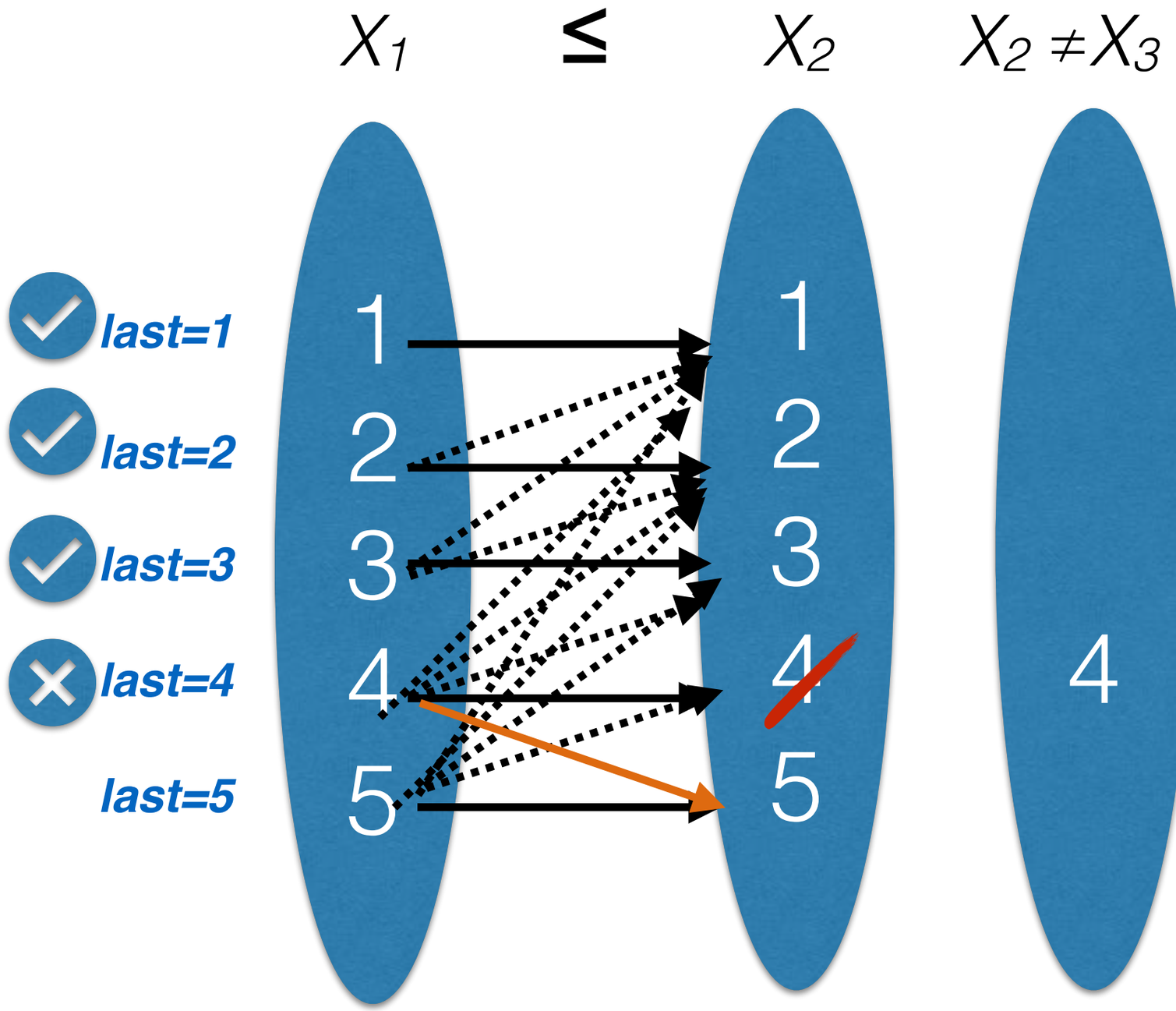
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



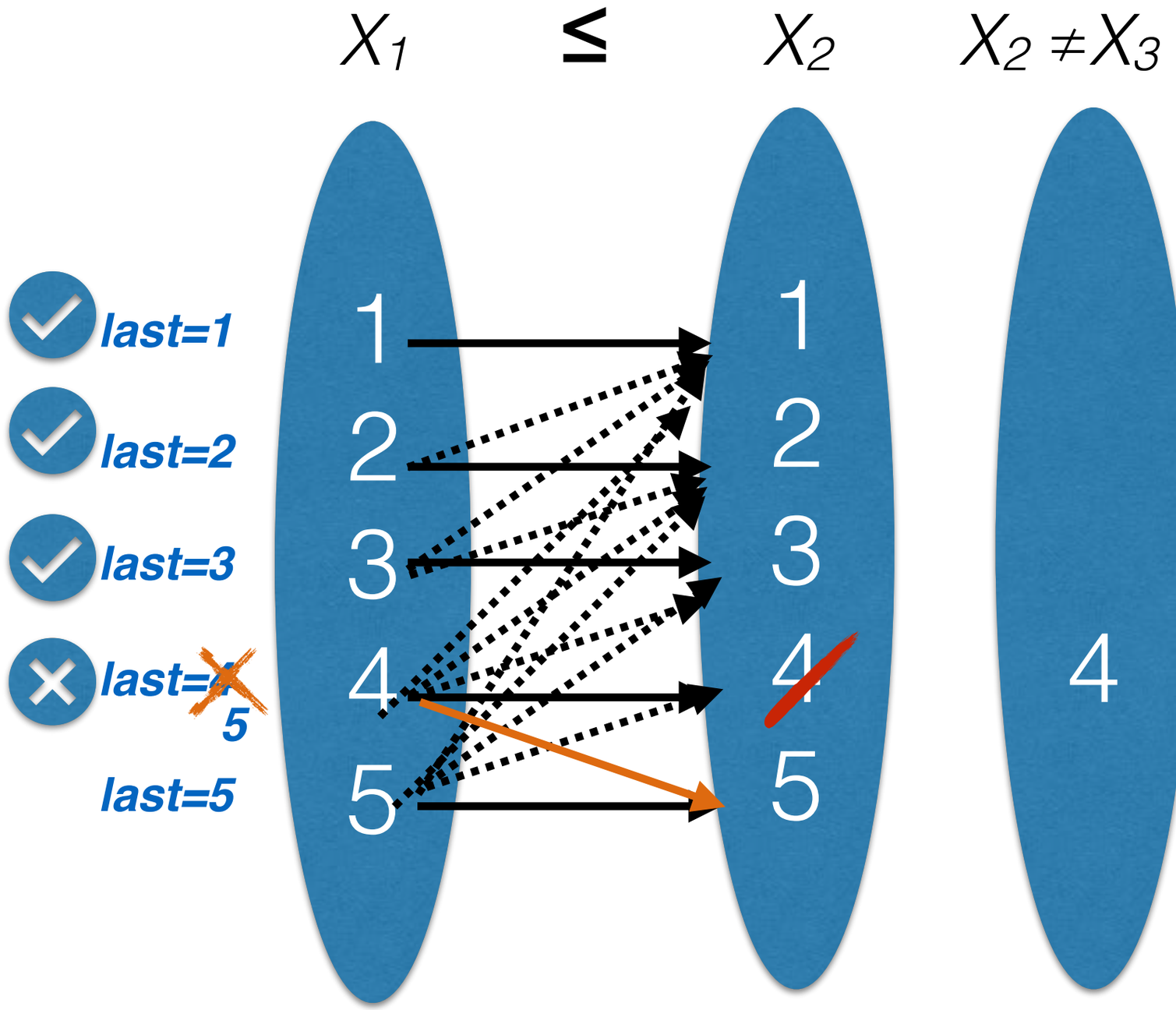
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



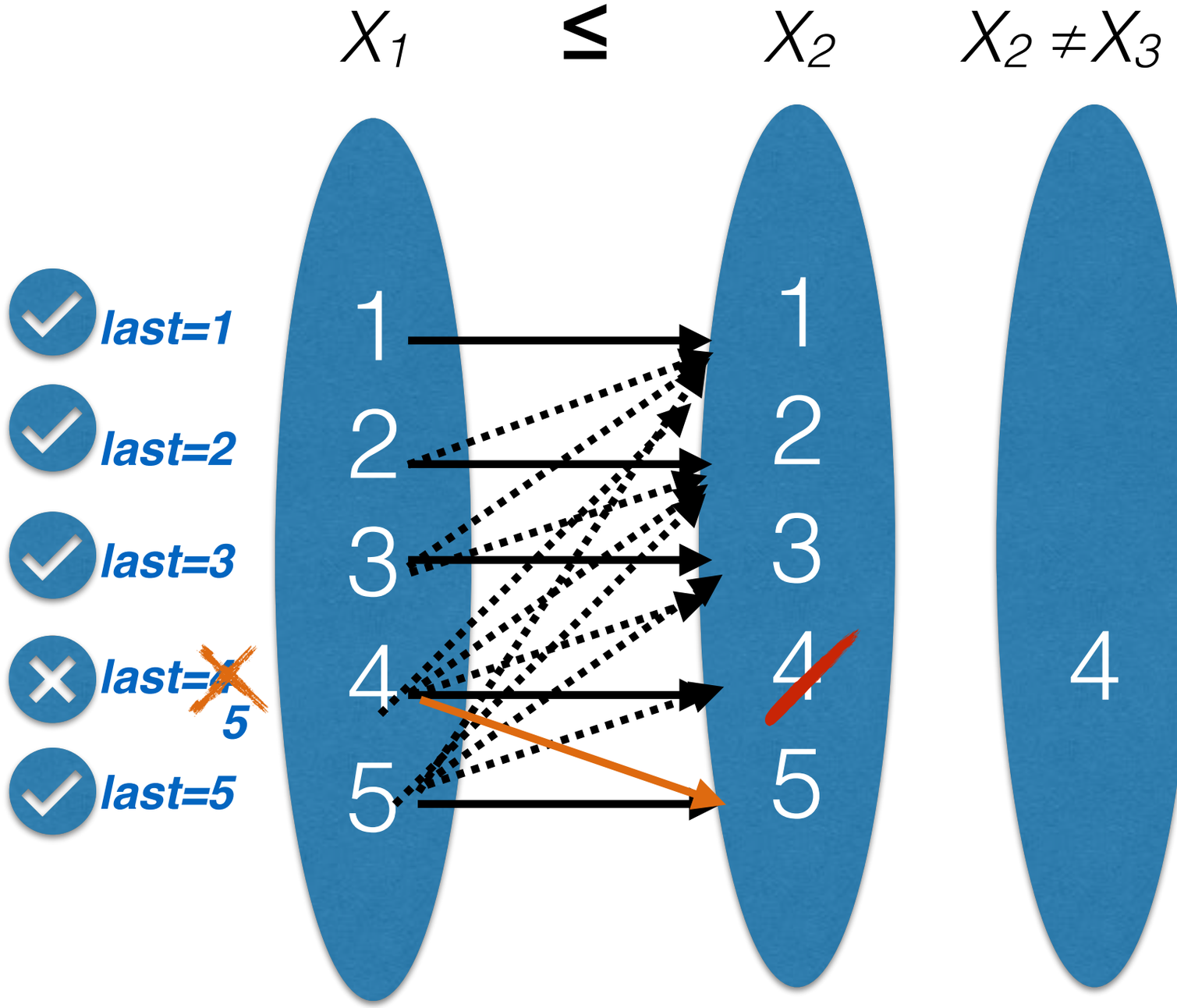
AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



AC2001

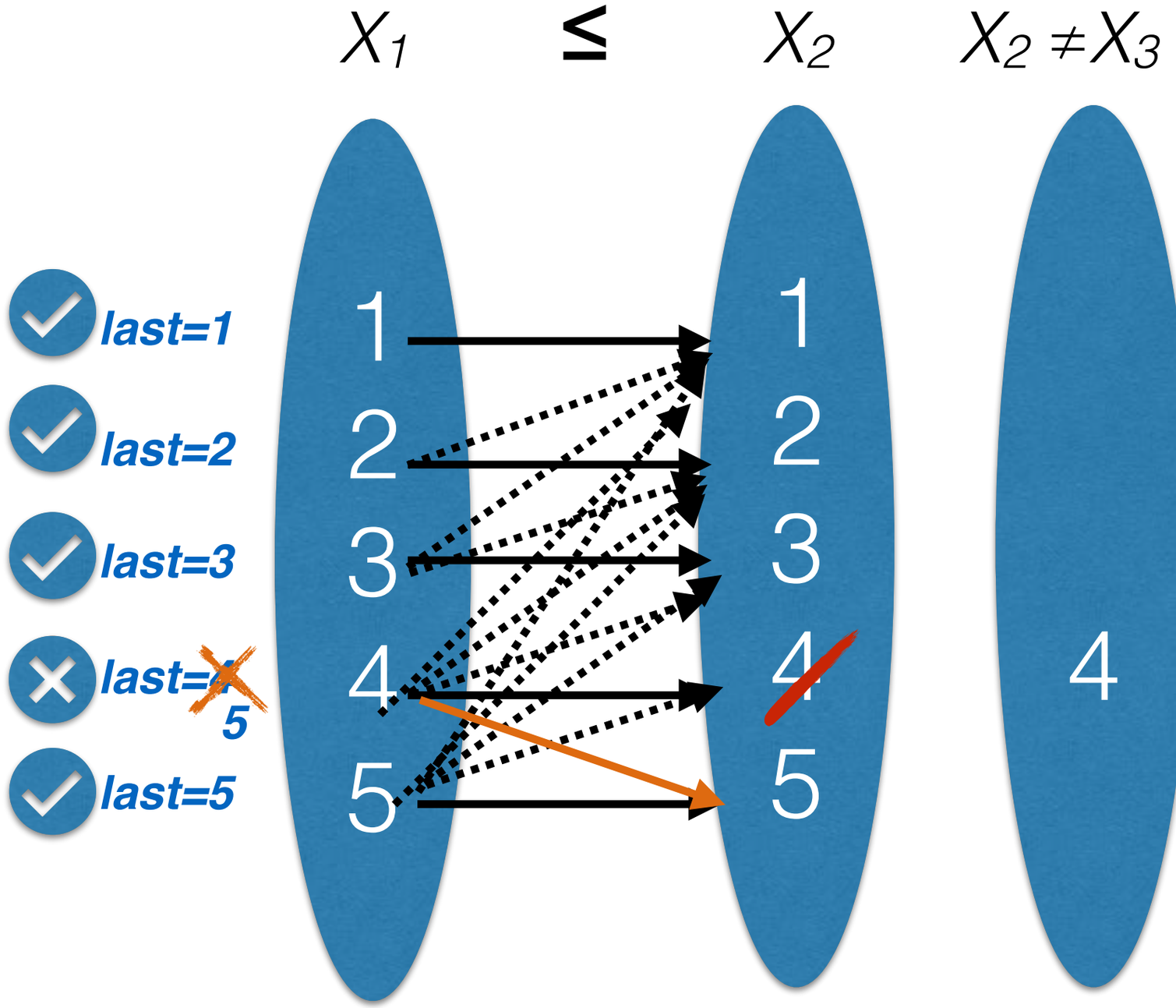
- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last



AC2001

- At the first call, AC2001 does like AC3
- **But** it stores the *last* support checked
- When coming back on this constraint, it uses the Last

➡ **AC2001 only checks unknown pairs**



Revise of AC2001

Revise of AC2001

```
function Revise3(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) do
3          v_j ← first(D(X_j));
4          while (v_j ≠ nil) and ¬c_ij(v_i, v_j) do v_j ← next(v_j, D(X_j));
5          if v_j = nil then
6              remove v_i from D(X_i);
7              CHANGE ← true;
8      return CHANGE ;
   end
```

```
function Revise2001(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) s.t. Last(X_i, v_i, X_j) ∉ D(X_j) do
3          v_j ← next(Last(X_i, v_i, X_j), D(X_j));
4          while (v_j ≠ nil) and c(v_i, v_j) ∉ c_ij do v_j ← next(v_j, D(X_j));
5          if v_j ≠ nil then Last(X_i, v_i, X_j) ← v_j;
6          else
7              remove v_i from D(X_i);
8              CHANGE ← true;
9      return CHANGE ;
   end
```

Revise of AC2001

```
function Revise3(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) do
3          v_j ← first(D(X_j));
4          while (v_j ≠ nil) and ¬c_ij(v_i, v_j) do v_j ← next(v_j, D(X_j));
5          if v_j = nil then
6              remove v_i from D(X_i);
7              CHANGE ← true;
8      return CHANGE ;
   end
```

```
function Revise2001(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) s.t. Last(X_i, v_i, X_j) ∉ D(X_j) do
3          v_j ← next(Last(X_i, v_i, X_j), D(X_j));
4          while (v_j ≠ nil) and c(v_i, v_j) ∉ c_ij do v_j ← next(v_j, D(X_j));
5          if v_j ≠ nil then Last(X_i, v_i, X_j) ← v_j;
6          else
7              remove v_i from D(X_i);
8              CHANGE ← true;
9      return CHANGE ;
   end
```

Revise of AC2001

```
function Revise3(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) do
3          v_j ← first(D(X_j));
4          while (v_j ≠ nil) and ¬c_ij(v_i, v_j) do v_j ← next(v_j, D(X_j));
5          if v_j = nil then
6              remove v_i from D(X_i);
7              CHANGE ← true;
8      return CHANGE ;
   end
```

```
function Revise2001(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) s.t. Last(X_i, v_i, X_j) ∉ D(X_j) do
3          v_j ← next(Last(X_i, v_i, X_j), D(X_j));
4          while (v_j ≠ nil) and c(v_i, v_j) ∉ c_ij do v_j ← next(v_j, D(X_j));
5          if v_j ≠ nil then Last(X_i, v_i, X_j) ← v_j;
6          else
7              remove v_i from D(X_i);
8              CHANGE ← true;
9      return CHANGE ;
   end
```

Revise of AC2001

```
function Revise3(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) do
3          v_j ← first(D(X_j));
4          while (v_j ≠ nil) and ¬c_ij(v_i, v_j) do v_j ← next(v_j, D(X_j));
5          if v_j = nil then
6              remove v_i from D(X_i);
7              CHANGE ← true;
8      return CHANGE ;
   end
```

```
function Revise2001(in X_i : variable ; c_ij : constraint) : Boolean
   begin
1      CHANGE ← false;
2      foreach v_i ∈ D(X_i) s.t. Last(X_i, v_i, X_j) ∉ D(X_j) do
3          v_j ← next(Last(X_i, v_i, X_j), D(X_j));
4          while (v_j ≠ nil) and c(v_i, v_j) ∉ c_ij do v_j ← next(v_j, D(X_j));
5          if v_j ≠ nil then Last(X_i, v_i, X_j) ← v_j;
6          else
7              remove v_i from D(X_i);
8              CHANGE ← true;
9      return CHANGE ;
   end
```

Complexity

- AC3
  - time $O(ed^3)$ on binary, space $O(e)$
- AC2001
  - time $O(ed^2)$ on binary ($O(ed^k)$ for k-ary), **optimal** space $O(ed)$
- STR
  - time $O(T)$, space $O(T)$ (T = taille de la table)

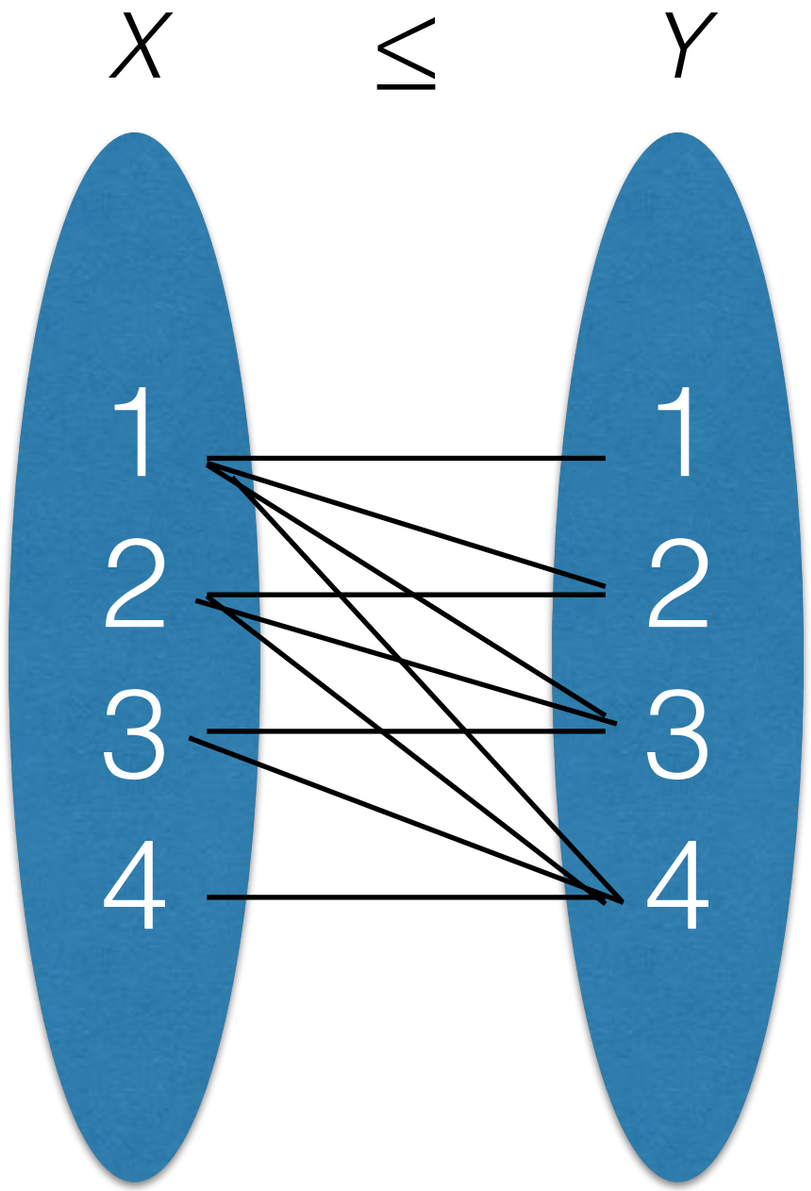
Arc consistency in solvers

- People who designed the first generation of solvers were not CP researchers

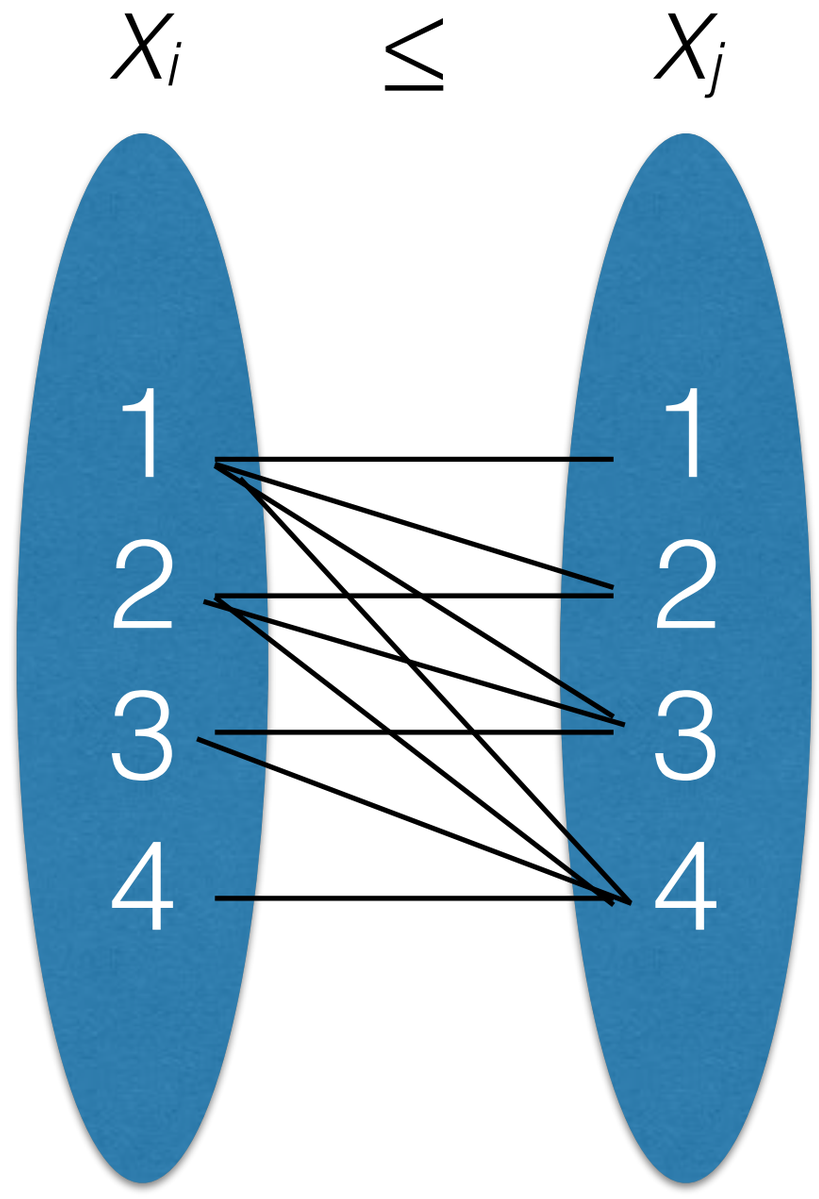
- They wanted efficient propagators for the constraints occurring often —> arithmetic

Removal events

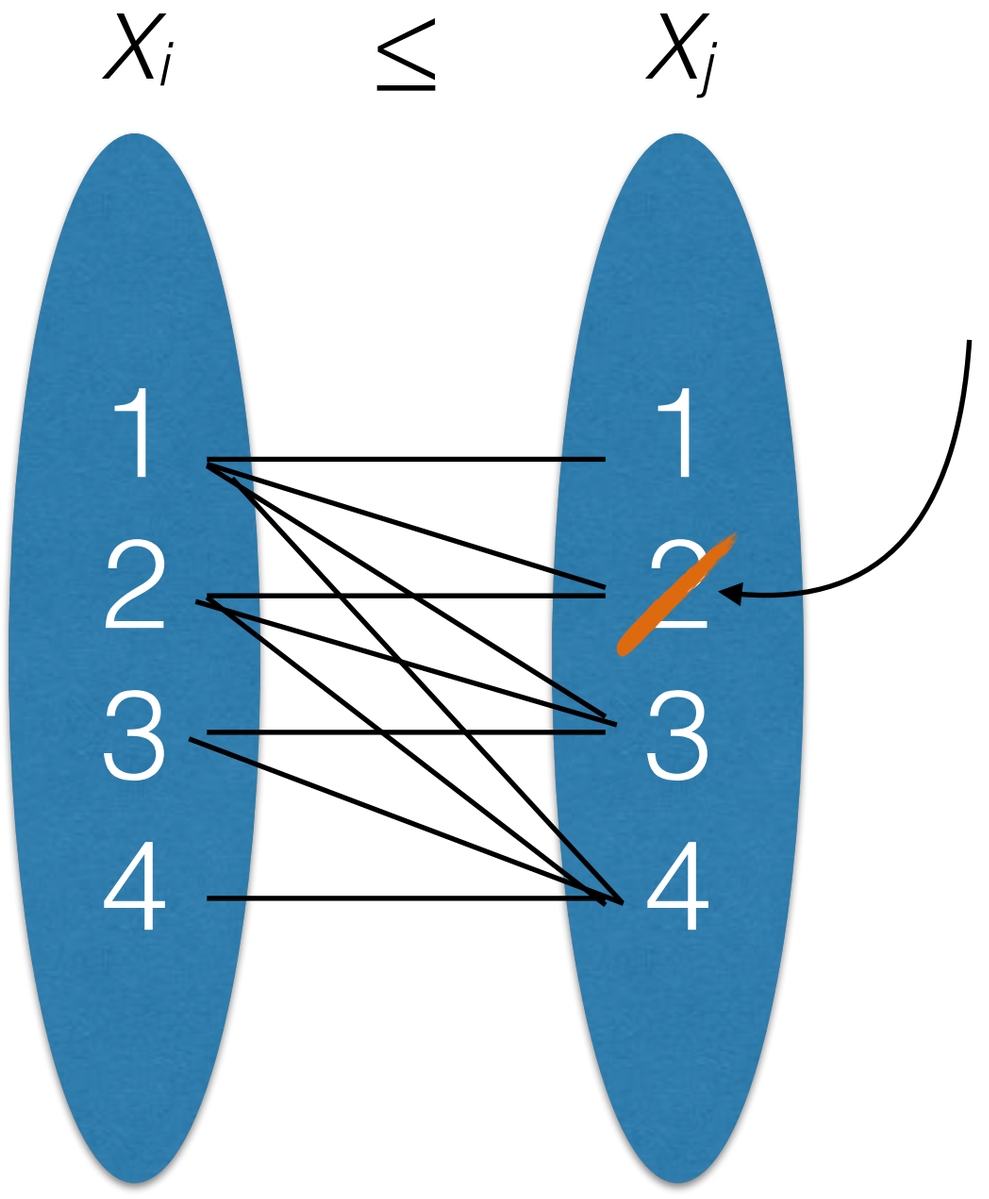
- Each domain reduction is labeled with an ‘event’
  - ***remValue***(X_i): a value is removed from D(X_i)
  - ***incMin***(X_i): the smallest value is removed from D(X_i)
  - ***decMax***(X_i): the greatest value is removed from D(X_i)
  - ***instantiate***(X_i): D(X_i) is reduced to a singleton



Removal events

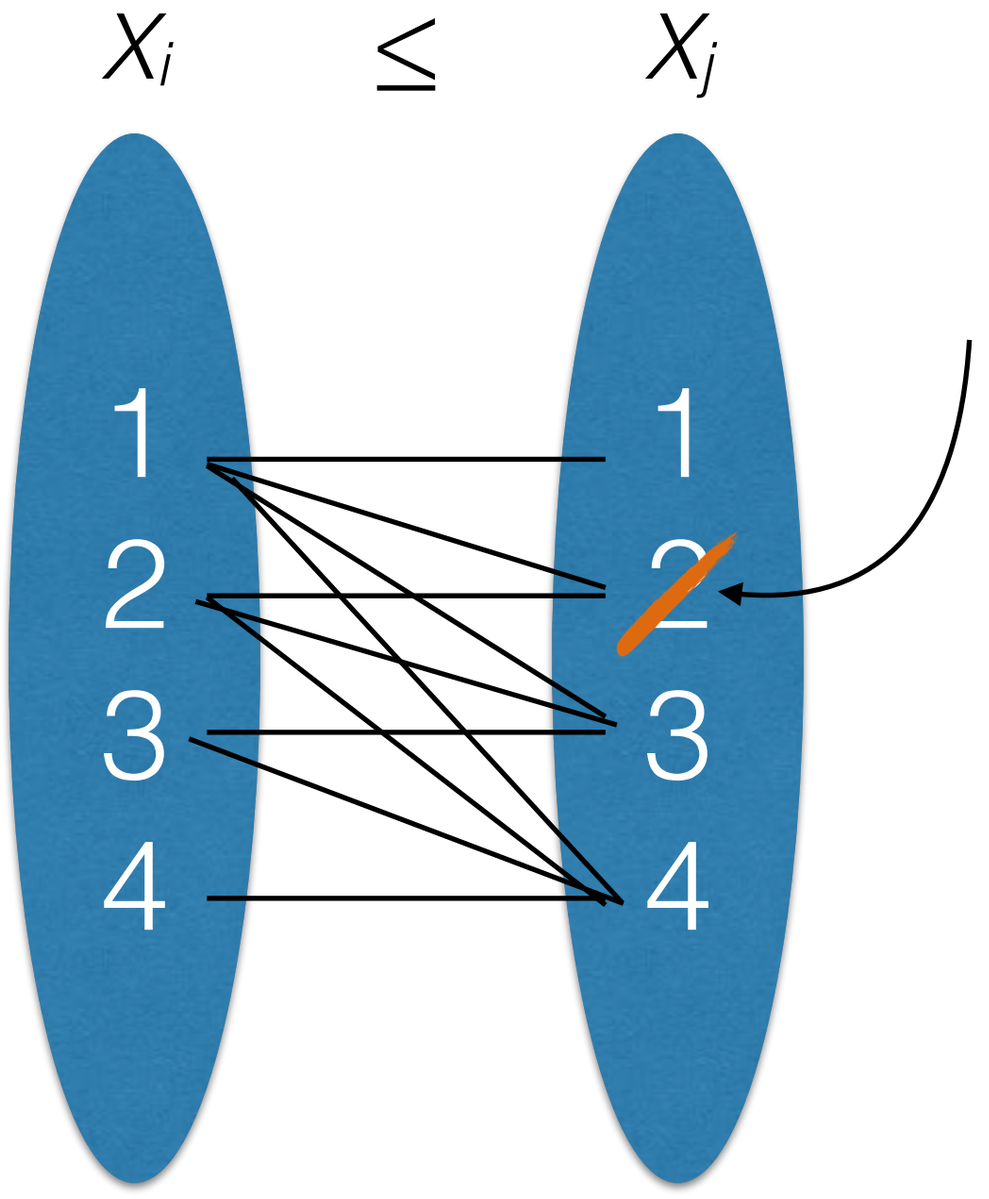


Removal events



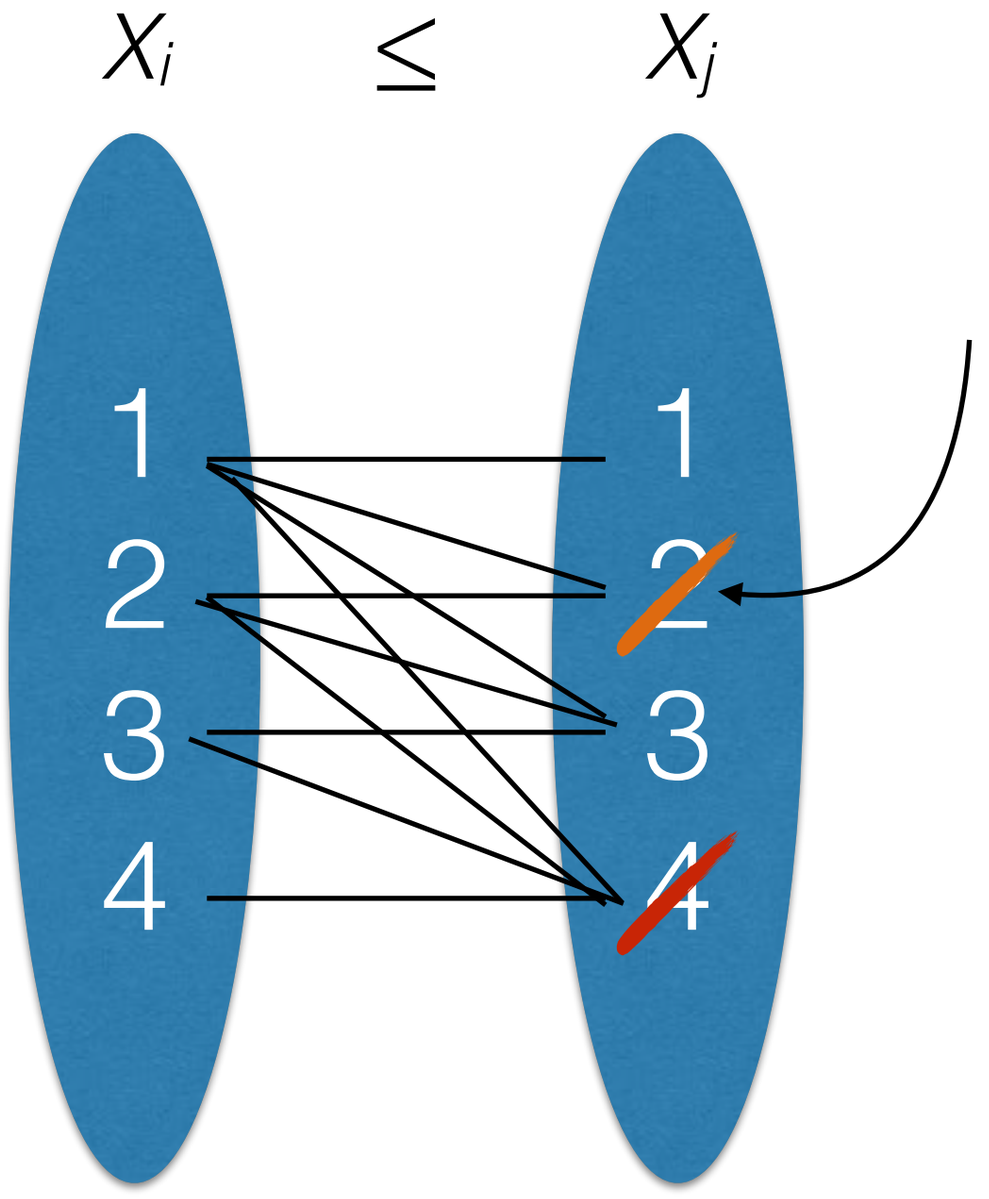
Removal events

- Removing (X_j,*2*) has no impact on X_i



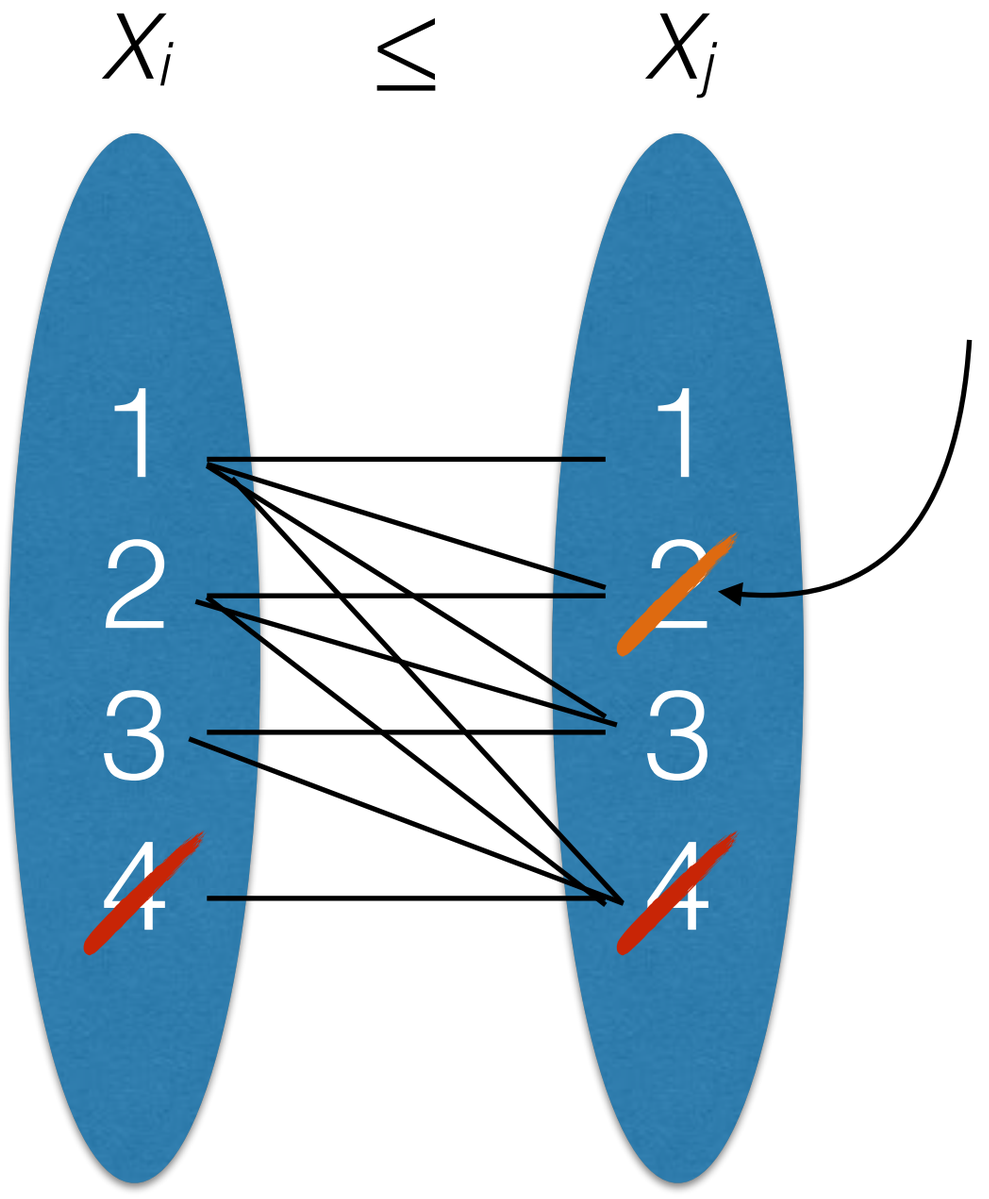
Removal events

- Removing (X_j,*2*) has no impact on X_i



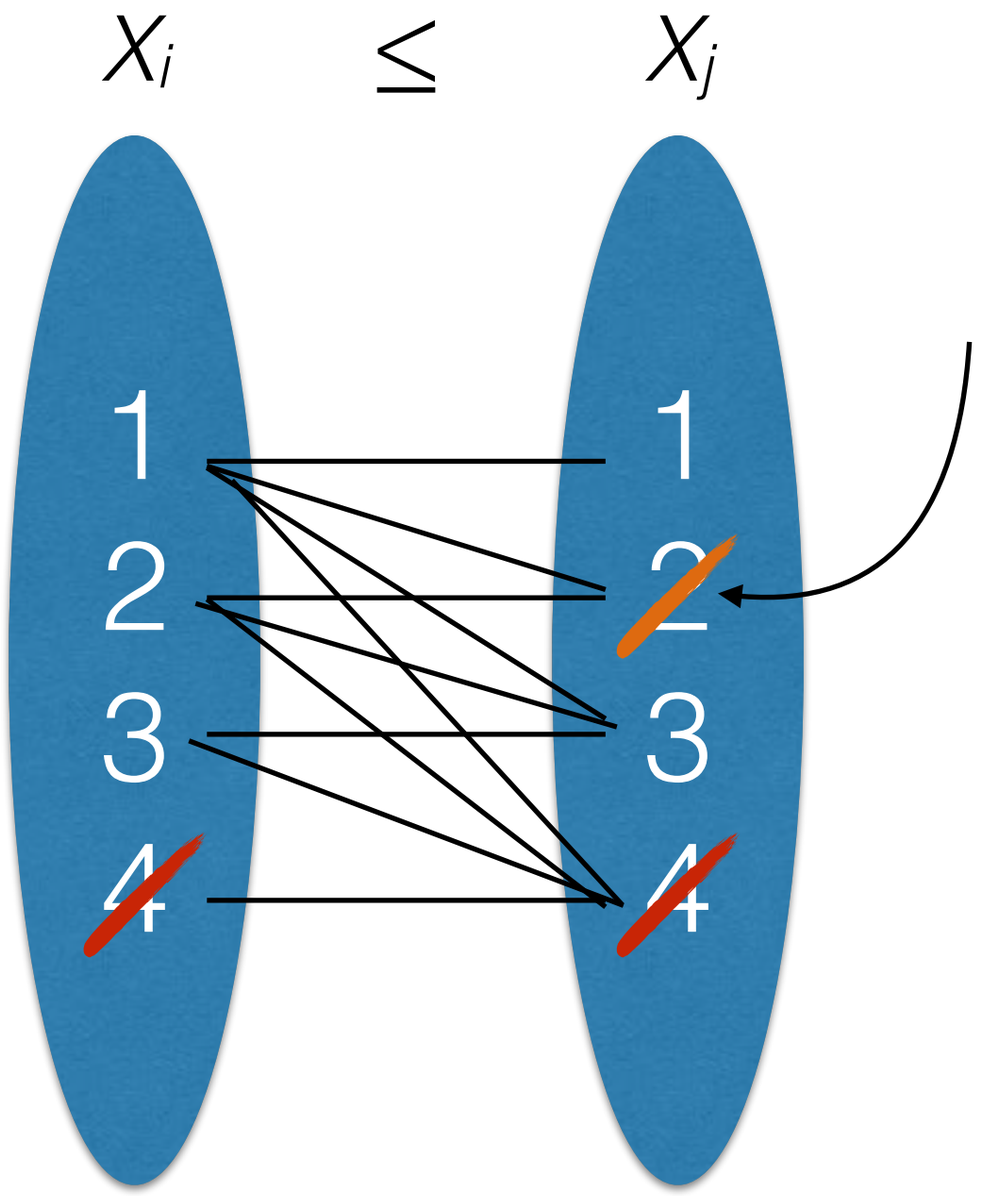
Removal events

- Removing (X_j,*2*) has no impact on X_i



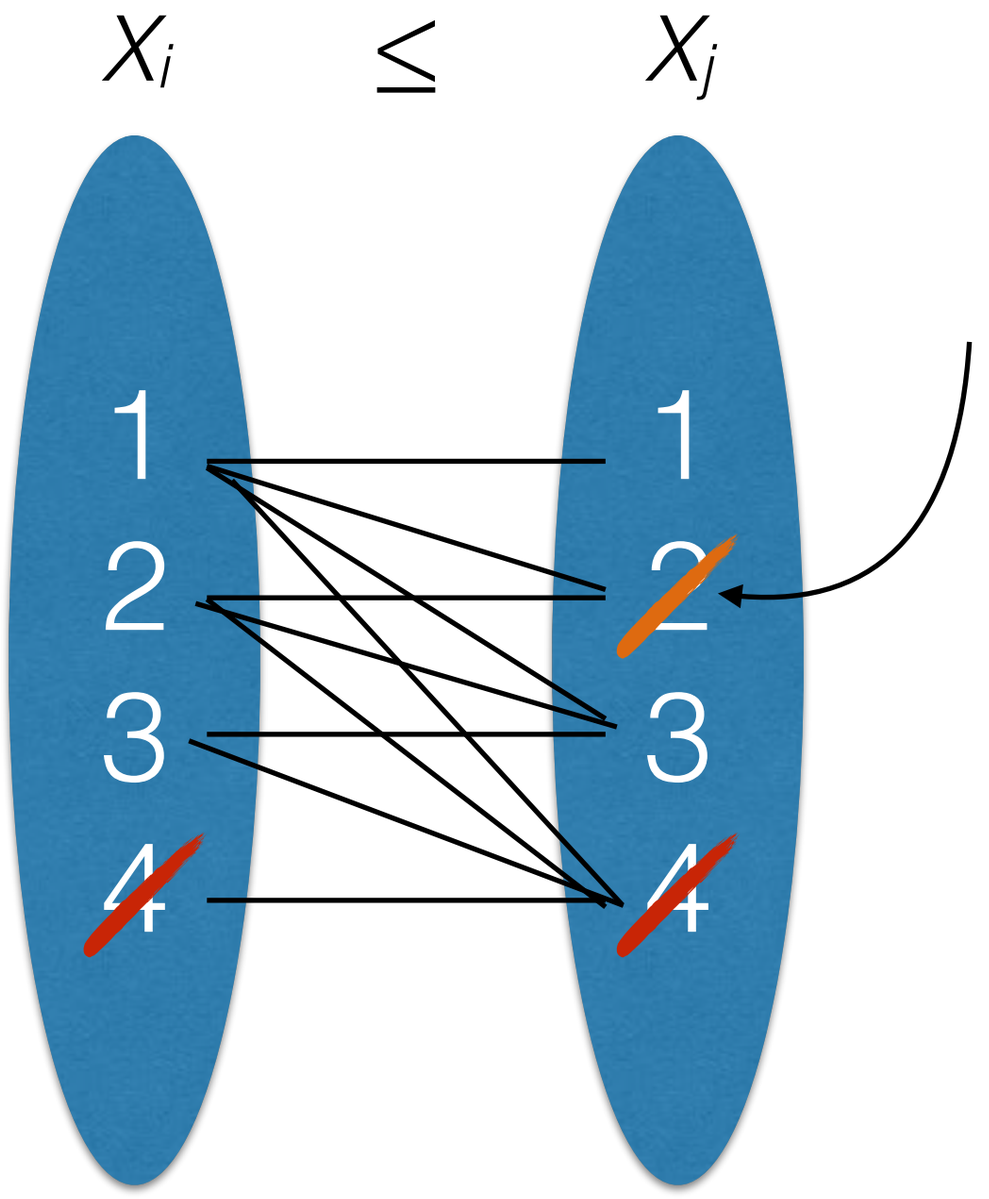
Removal events

- Removing (X_j,*2*) has no impact on X_i
- Removing (X_j,*4*) **has** impact on X_i



Removal events

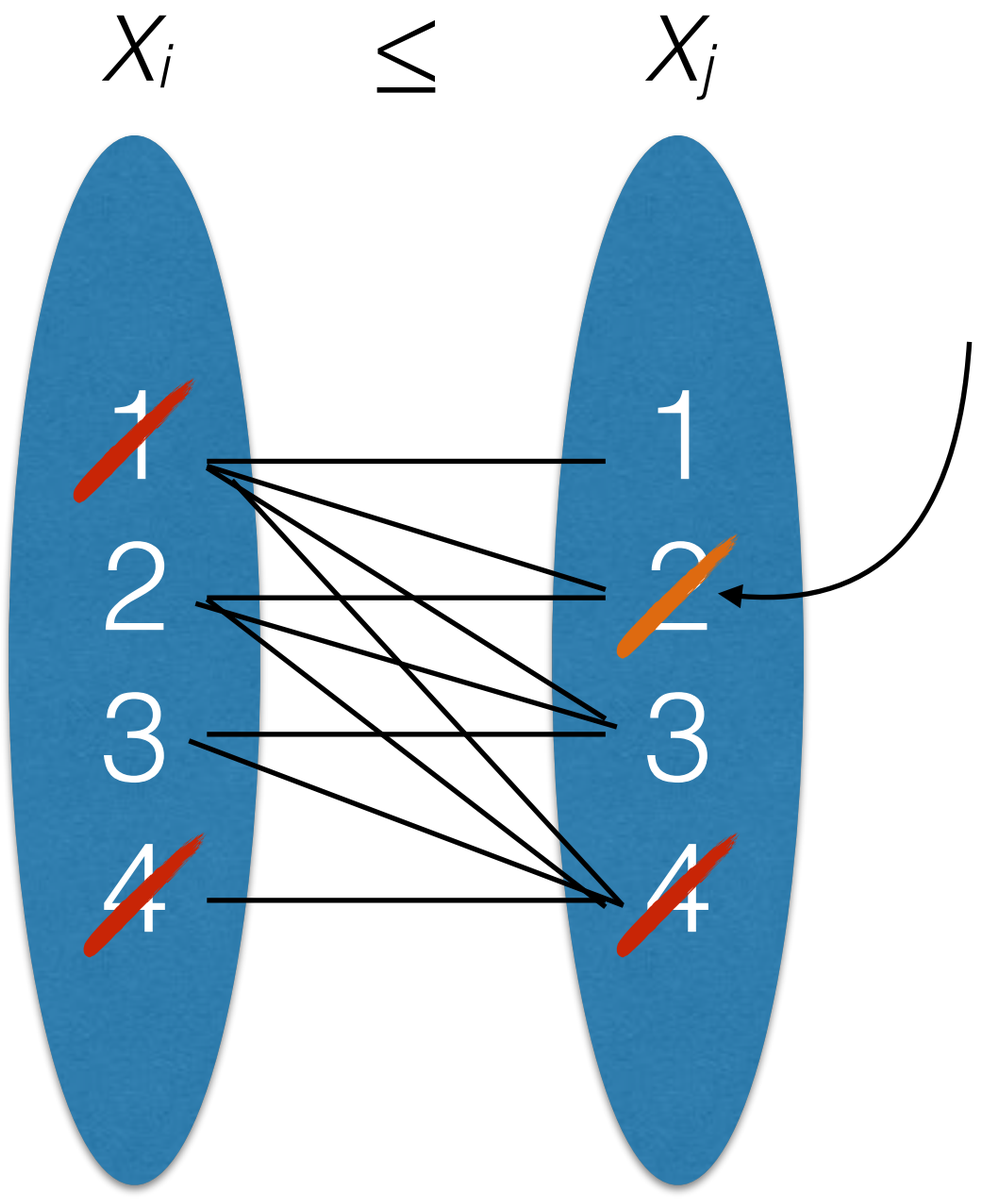
- Removing (X_j,*2*) has no impact on X_i
- Removing (X_j,*4*) **has** impact on X_i



function `Revise`$(X_i, X_i \leq X_j)$:**Boolean**
  begin
    if $decmax(X_j)$ **then**
      remove $v \in D(X_i) \mid v > max(X_j)$
  end

Removal events

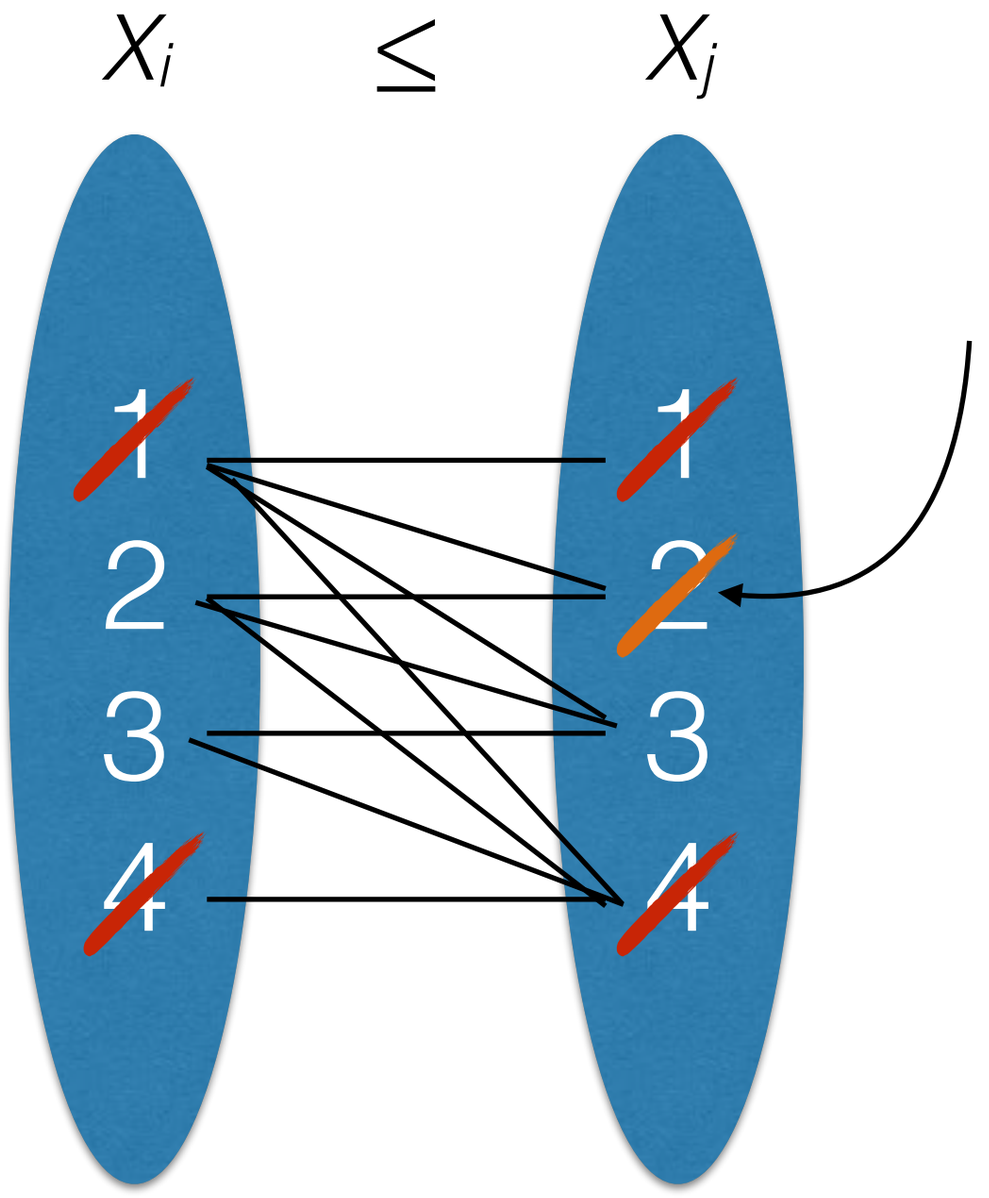
- Removing (X_j,*2*) has no impact on X_i
- Removing (X_j,*4*) **has** impact on X_i



function `Revise`$(X_i, X_i \leq X_j)$:**Boolean**
  begin
    if $decmax(X_j)$ **then**
      remove $v \in D(X_i) \mid v > max(X_j)$
  end

Removal events

- Removing (X_j,*2*) has no impact on X_i
- Removing (X_j,*4*) **has** impact on X_i

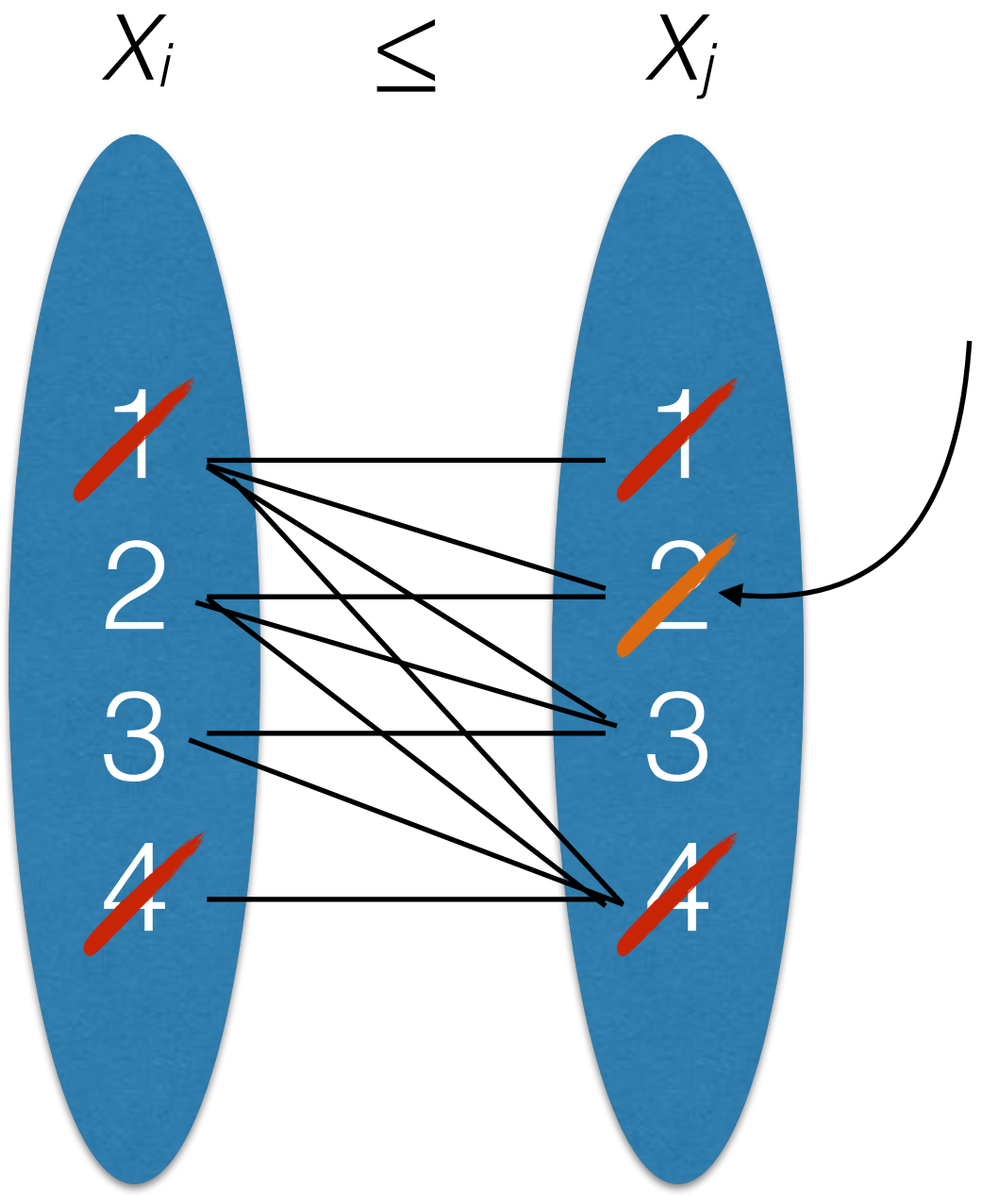


function `Revise`$(X_i, X_i \leq X_j)$:**Boolean**
  begin
    if $decmax(X_j)$ **then**
      remove $v \in D(X_i) \mid v > max(X_j)$
  end

Removal events

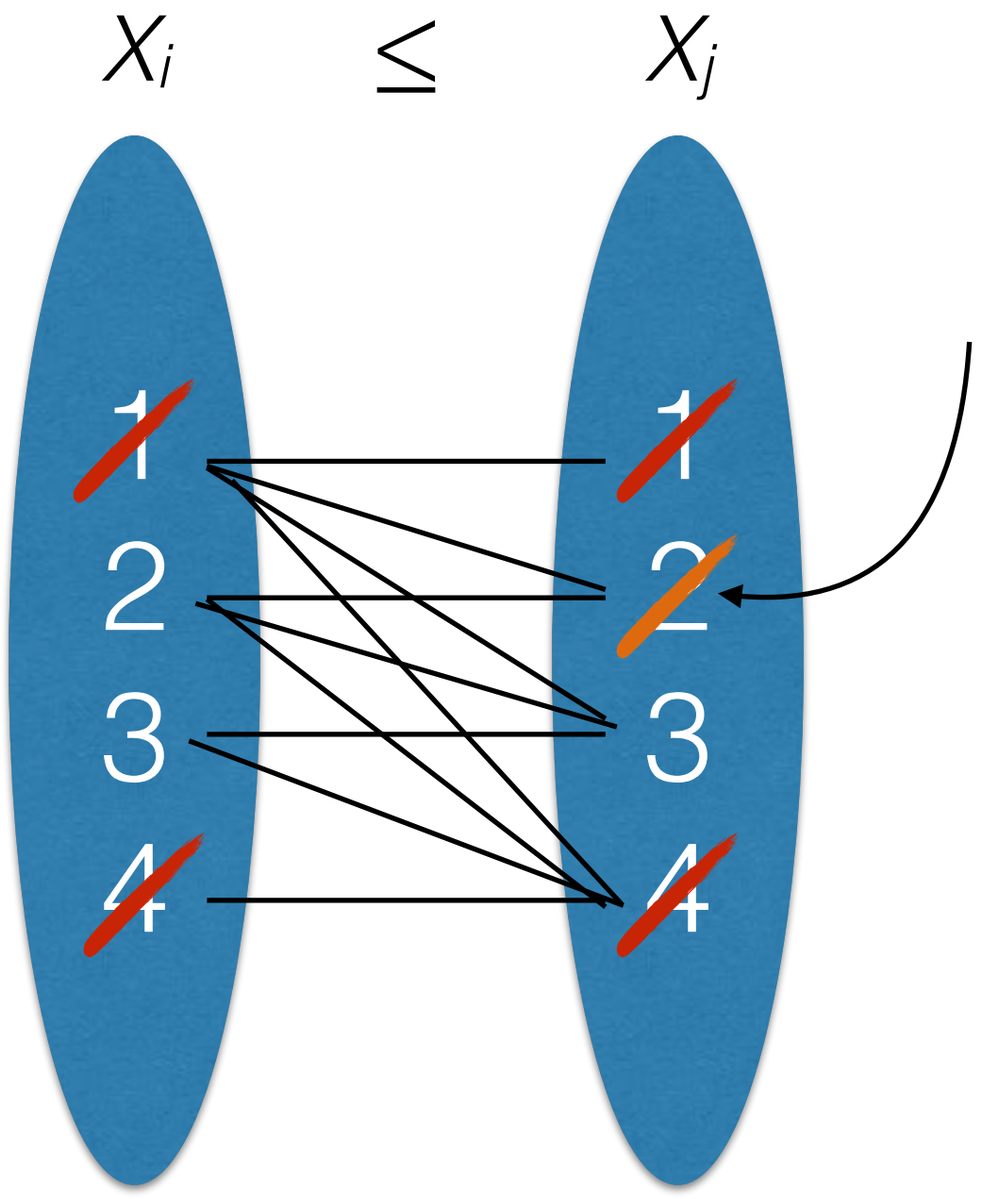
- Removing (X_j,*2*) has no impact on X_i
- Removing (X_j,*4*) **has** impact on X_i
- Removing (X_i,*1*) **has** impact on X_j

function `Revise`$(X_i, X_i \leq X_j)$:**Boolean**
  begin
    if $decmax(X_j)$ **then**
      remove $v \in D(X_i) \mid v > max(X_j)$
  end



Removal events

- Removing (X_j,*2*) has no impact on X_i
- Removing (X_j,*4*) **has** impact on X_i
- Removing (X_i,*1*) **has** impact on X_j

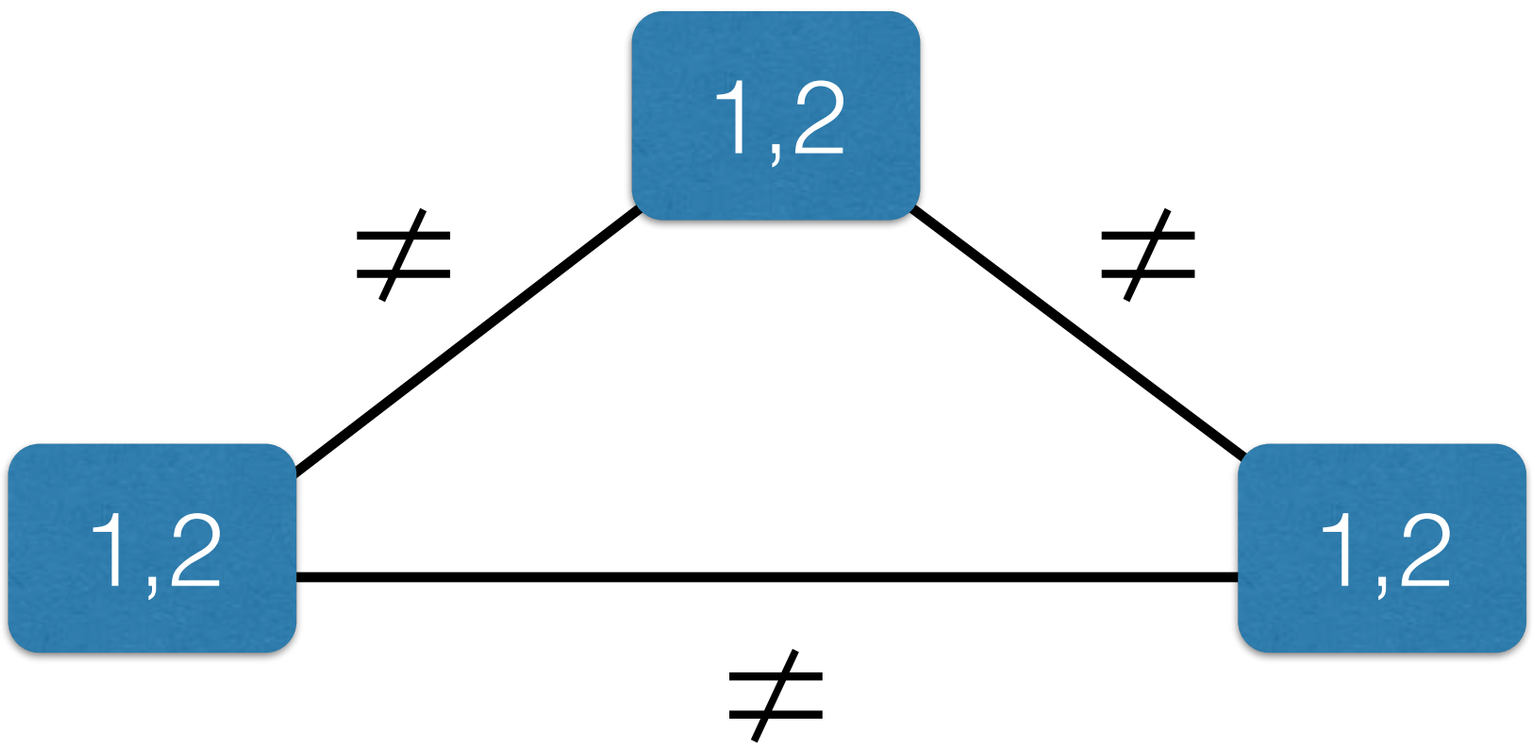


function `Revise`$(X_i, X_i \leq X_j)$:**Boolean**
  begin
    if $decmax(X_j)$ **then**
      remove $v \in D(X_i) \mid v > max(X_j)$
  end

function `Revise`$(X_j, X_i \leq X_j)$:**Boolean**
  begin
    if $incmin(X_i)$ **then**
      remove $v \in D(X_j) \mid v < min(X_i)$
  end

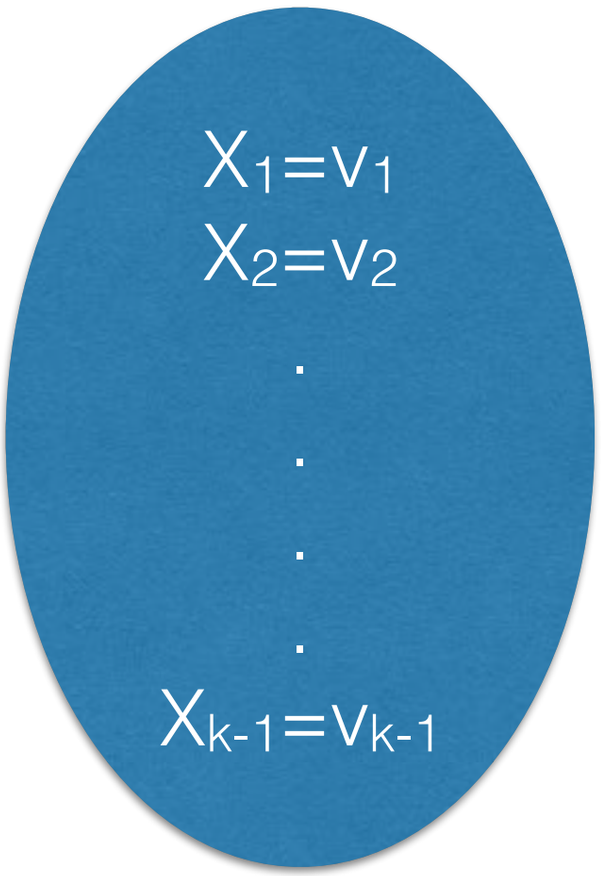
Warning on AC

- A network can be arc consistent while being inconsistent (no solution)

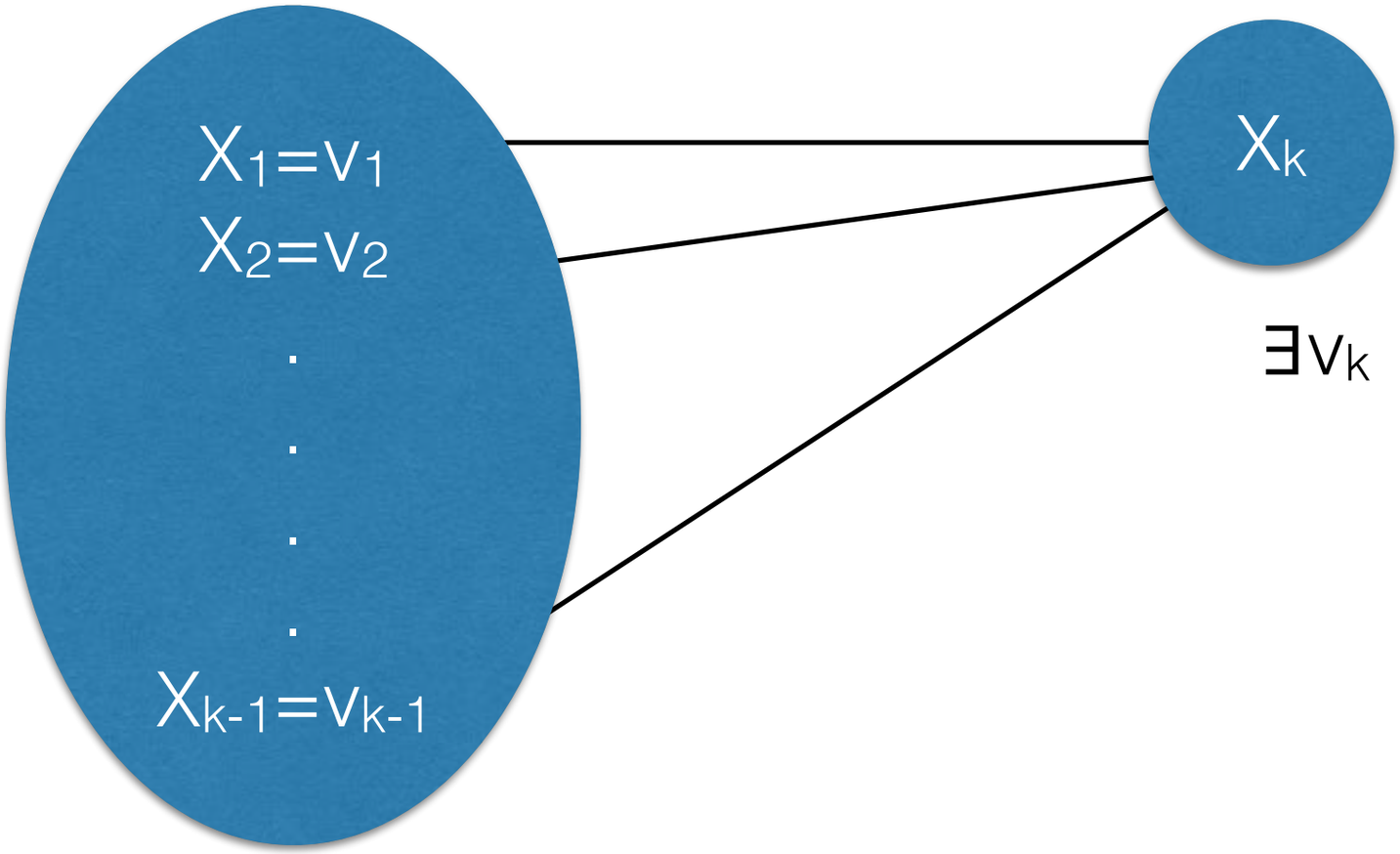


Stronger levels of consistency

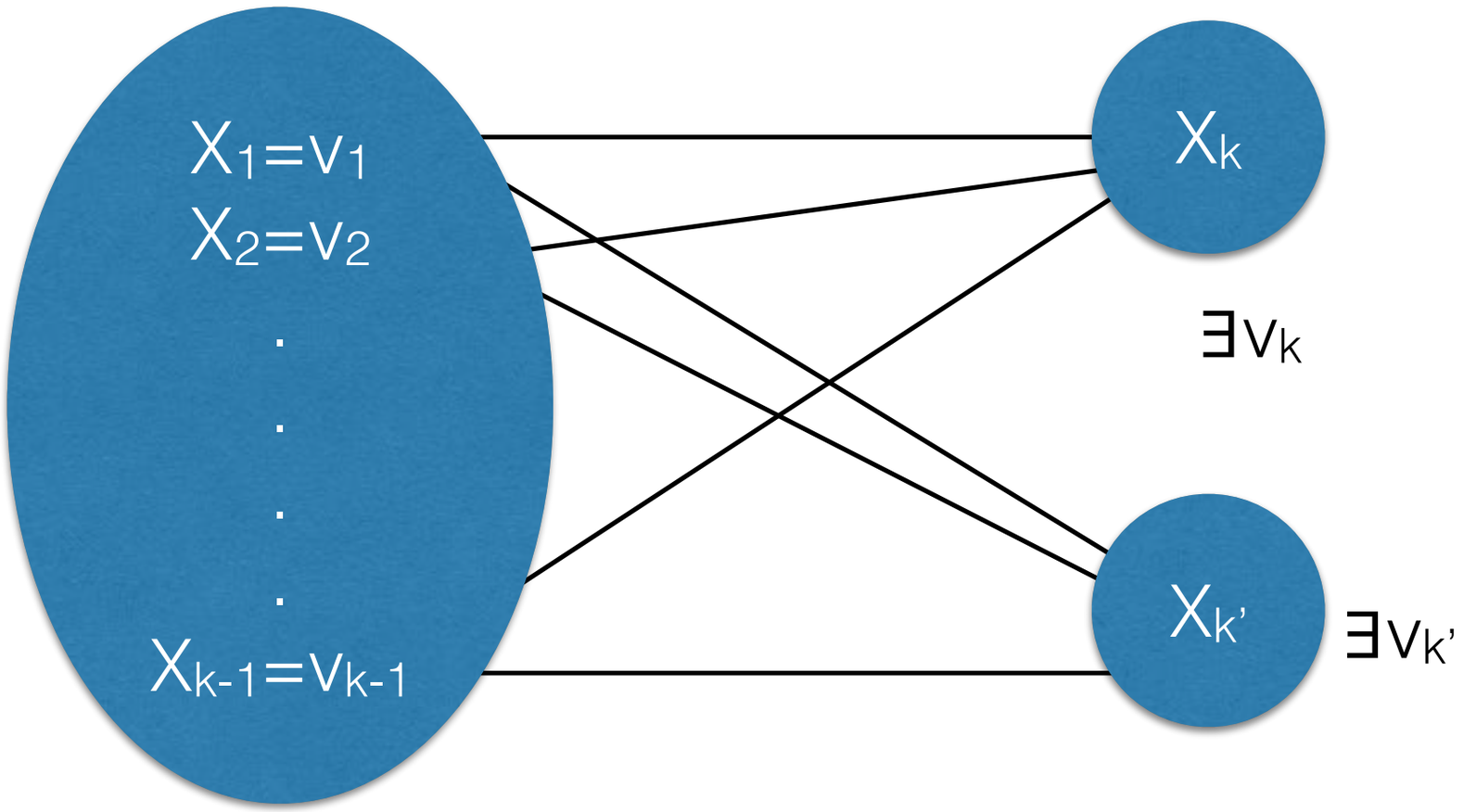
k-consistency



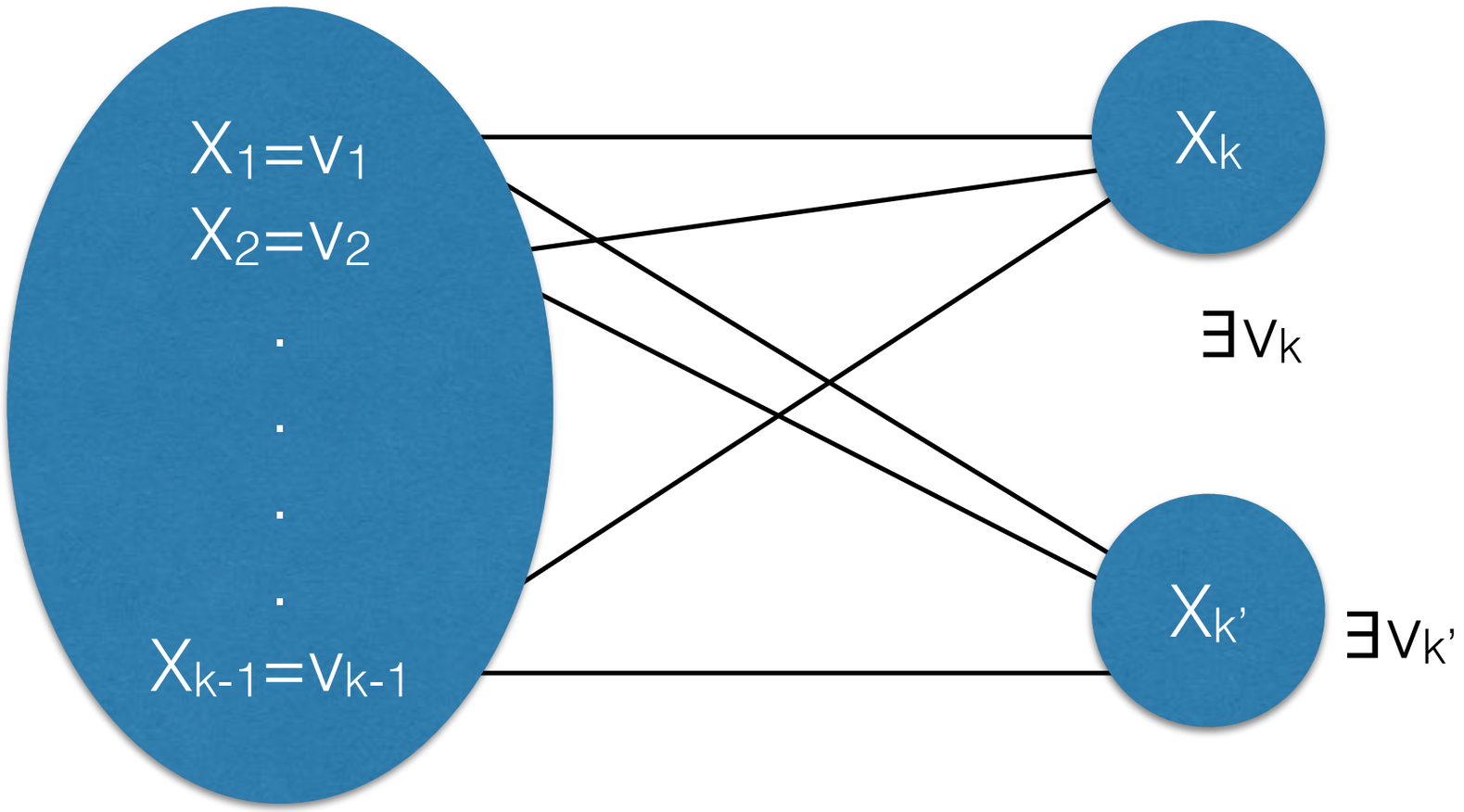
k-consistency



k-consistency

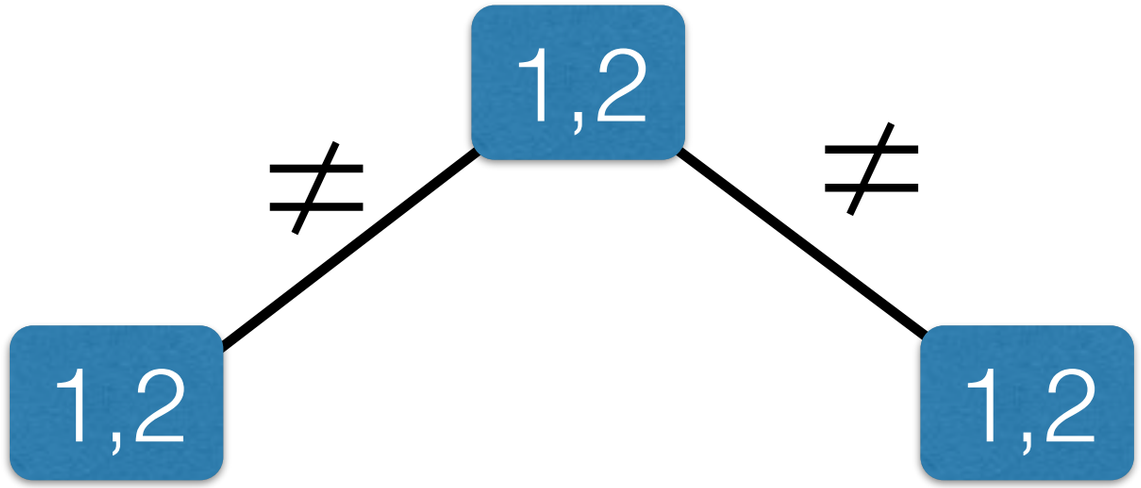


k-consistency

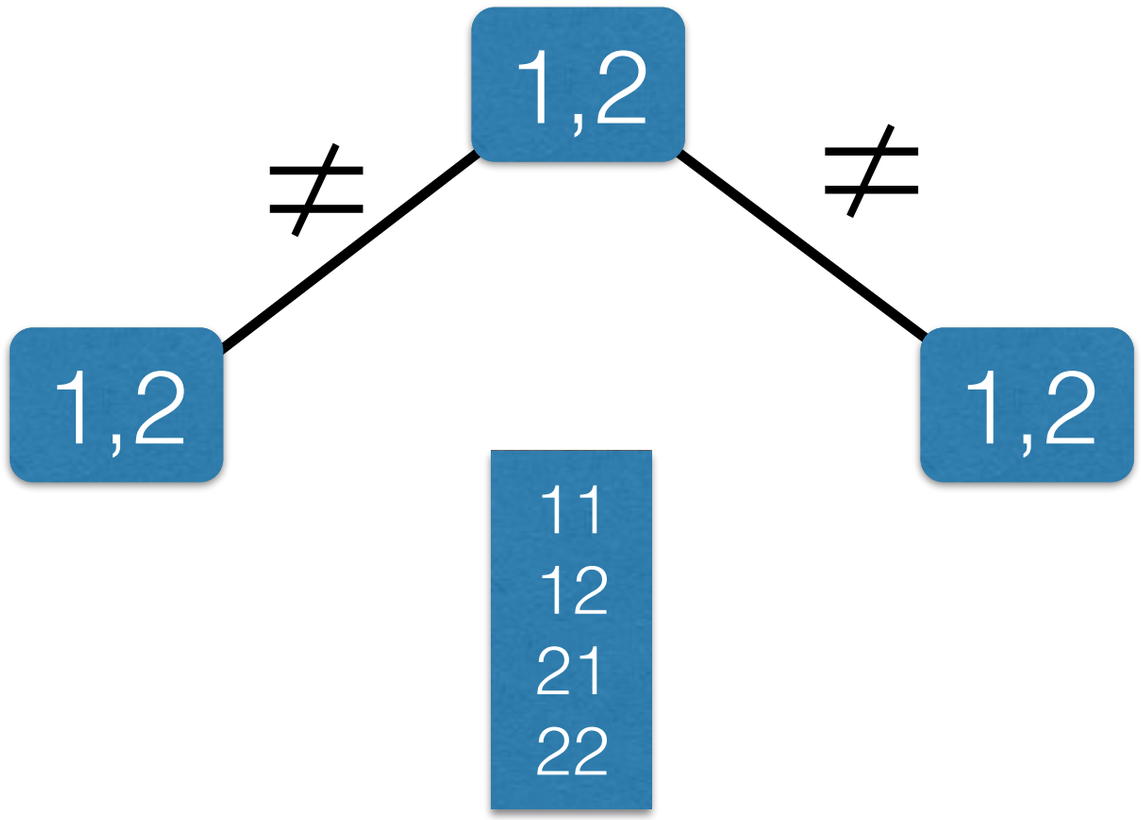


Definition 5 (*k*-consistency) *A network N is* k-consistent *iff* $\forall Y \subseteq X, |Y| = k-1$, $\forall I$ *locally consistent instantiation on* Y, $\forall X_{i_k} \in X \setminus Y$, $\exists v_{i_k} \in D(X_{i_k})$ *such that* $I \cup (X_{i_k}, v_{i_k})$ *is locally consistent.*

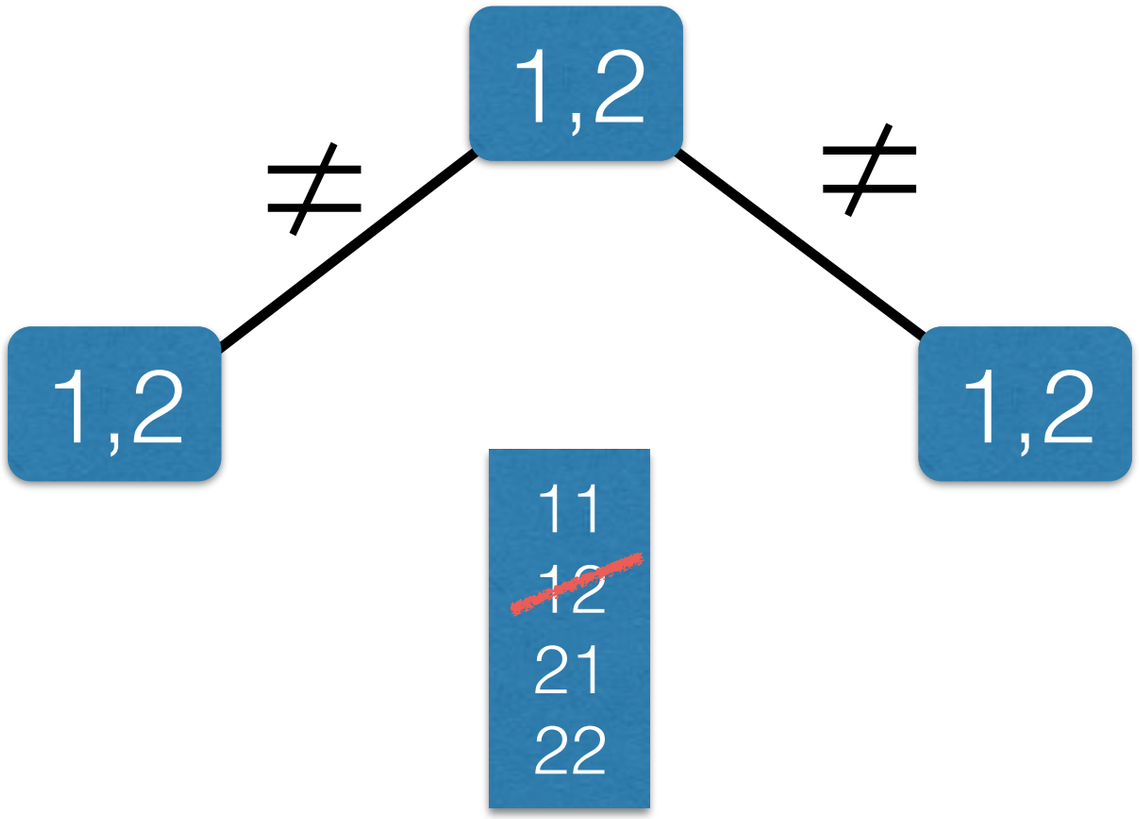
Example: 3-consistency



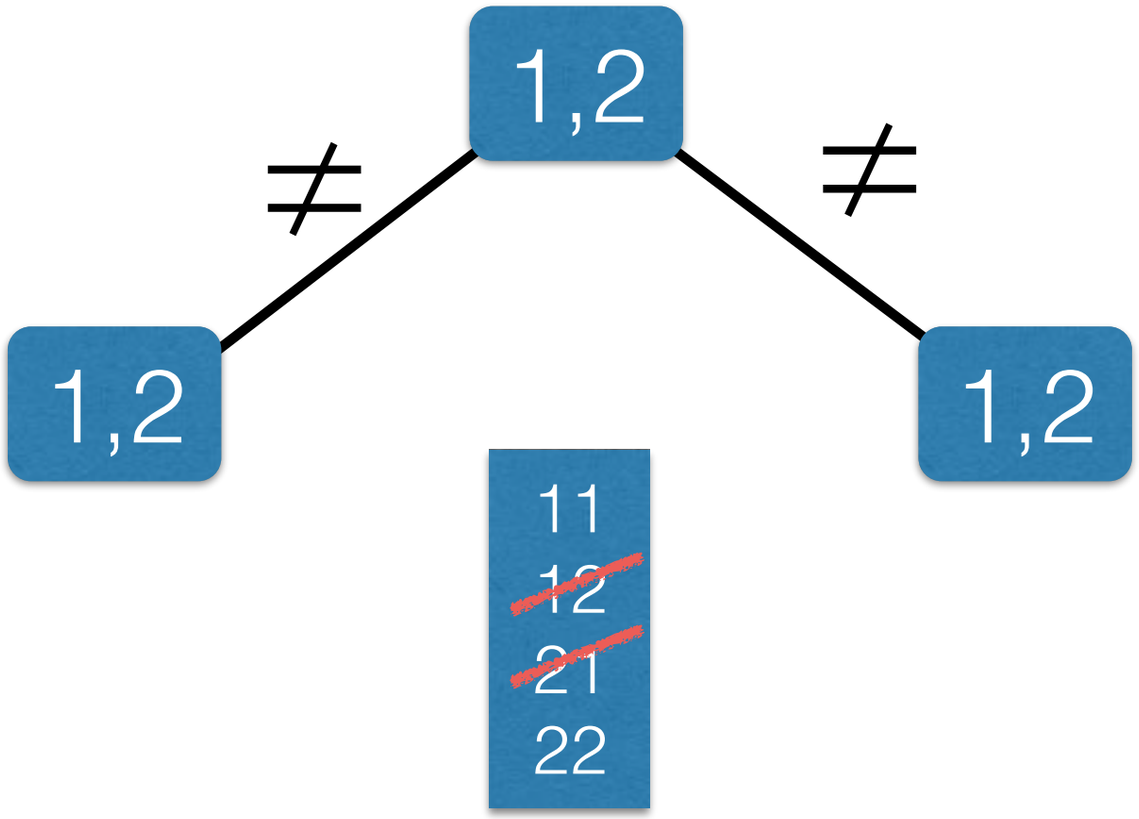
Example: 3-consistency



Example: 3-consistency

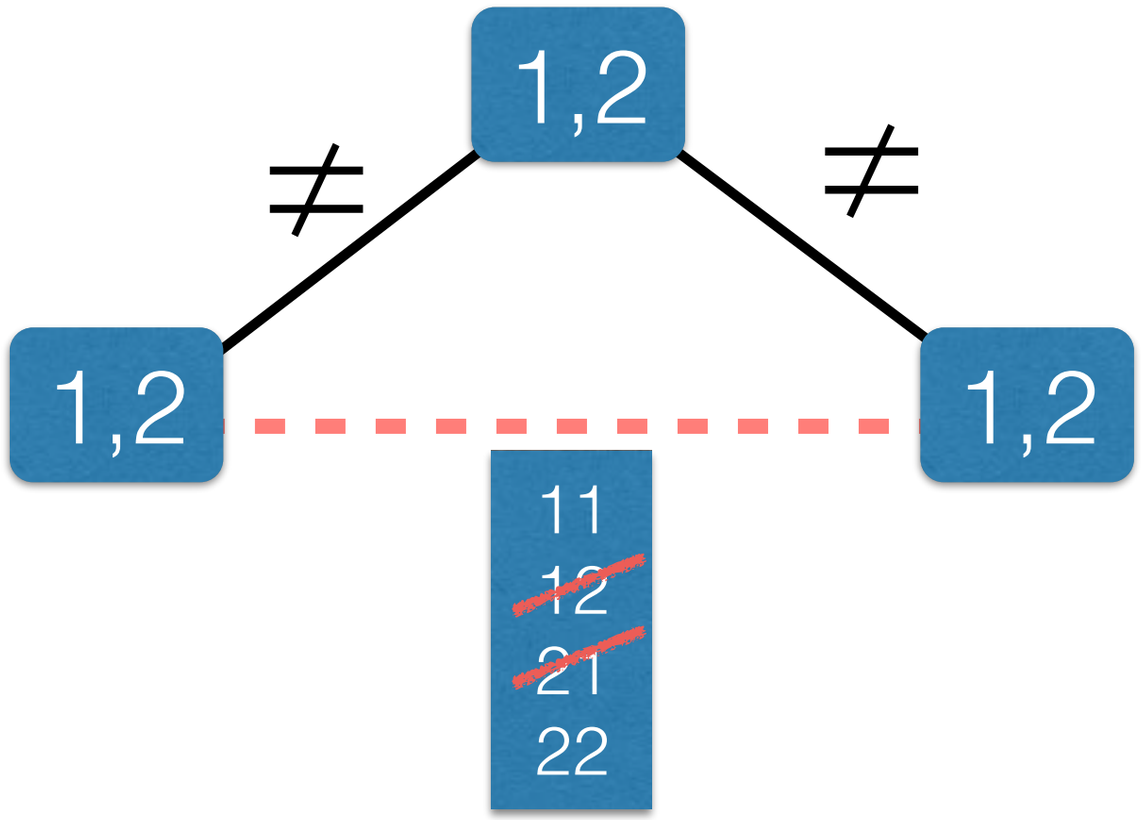


Example: 3-consistency



Example: 3-consistency

- Enforcing *k*-consistency creates new constraints!

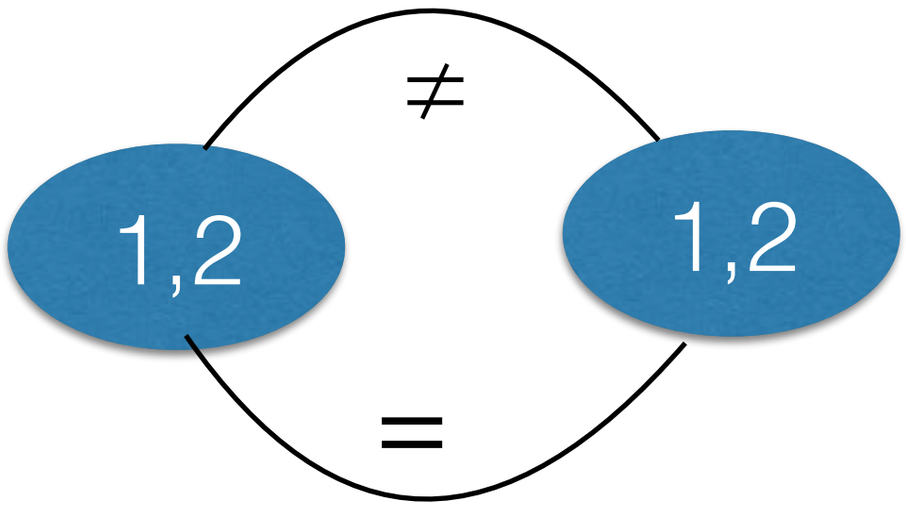


2-consistency

	normalized network	non normalized
binary network	2-c = AC	2-c > AC
non binary	2-c < AC	2-c ≠ AC

Normalized = not two constraints with the same scope

2-c > AC:



2-c < AC:

X+Y=Z

1	1	1
2	2	2
3	3	3

Strong *k*-consistency

Strong k-consistency

Definition 6 (Strong k-consistency) *A network is* strongly k-consistent *iff it is j-consistent for any $j, 1 \leq j \leq k$.*

Strong *k*-consistency

Definition 6 (Strong k-consistency) *A network is* strongly k-consistent *iff it is j-consistent for any $j, 1 \leq j \leq k$.*

- time complexity is in $O(n^{k+1}d^{k+1})$ if space in $O(n^{k-1}d^{k-1})$

Strong *k*-consistency

Definition 6 (Strong k-consistency) *A network is* strongly k-consistent *iff it is j-consistent for any $j, 1 \leq j \leq k$.*

- time complexity is in $O(n^{k+1}d^{k+1})$ if space in $O(n^{k-1}d^{k-1})$
- time complexity is in $O(n^k d^k)$ if space in $O(n^k d^k)$

Global consistency

Global consistency

Definition 7 *An instantiation is said* globally consistent *iff it can be extended to a solution. A network is said globally consistent iff every locally consistent instantiation is also globally consistent.*

Global consistency

Definition 7 *An instantiation is said* globally consistent *iff it can be extended to a solution. A network is said globally consistent iff every locally consistent instantiation is also globally consistent.*

Observation:

A globally consistent network is obviously BT-free

Global consistency

Definition 7 *An instantiation is said* globally consistent *iff it can be extended to a solution. A network is said globally consistent iff every locally consistent instantiation is also globally consistent.*

Observation:

A globally consistent network is obviously BT-free

Proposition 4 *A network is globally consistent iff it is strongly n-consistent.*