General Search Strategies Look Back

Chapter 6 @ Constraint Processing by Rina Dechter

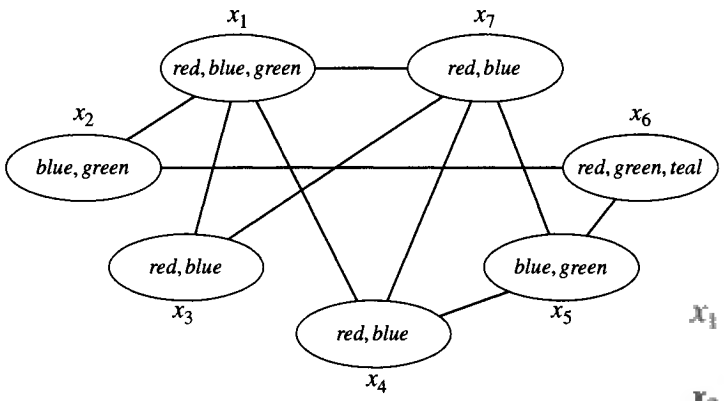
Motivation

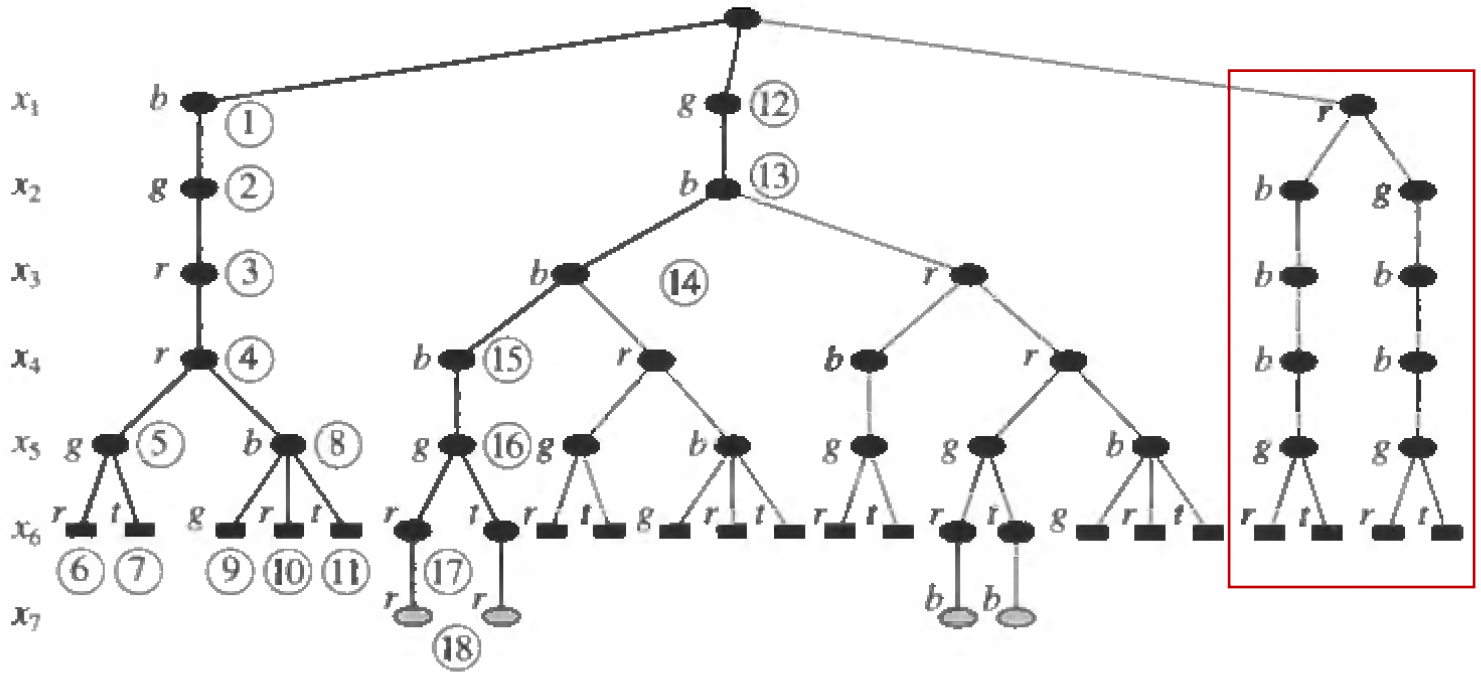
- Improve backtracking backward phase
- Goal: counteract backtracking's propensity for thrashing
  - I.e. repeatedly rediscovering the same inconsistencies and partial successes
- Look-back methods: backjumping and learning
- Backjumping
  - Jump more than one step backwards when encountering a dead-end
- Learning (constraint recording or no-good learning)
  - Record the reasons for the dead-end in the form of new constraints
  - So that the same conflicts will not arise again later in the search

Outline

- Conflict sets
- Backjumping styles
  - Gaschnig's backjumping
  - Graph-based backjumping
  - Conflict-directed backjumping
- Learning algorithms
  - Graph-based learning

Conflict sets





- Goal: identify a culprit variable responsible for a dead-end (▬)

Conflict sets: definitions (I)

Assume fixed variable ordering d = (x_1, … x_n)

(dead-end)

A dead-end state at level i indicates that a current partial instantiation $\vec{a}_i = (a_1, \ldots, a_i)$ conflicts with every possible value of x_{i+1}. $(a_1, \ldots, a_i)$ is called a *dead-end state*, and x_{i+1} is called a *dead-end variable*. That is, backtracking generated the consistent tuple $\vec{a}_i = (a_1, \ldots, a_i)$ and tried to extend it to the next variable, x_{i+1}, but failed; no value of x_{i+1} was consistent with all the values in $\vec{a}_i$. •

Dead-end applies to states (a_1,…a_i) and variables (x_{i+1})

A sub-tuple (a_1,…,a_{i-1}) may also be in conflict with x_{i+1}!

Algorithm should jump back to most recent variable not conflicting with dead-end!

Conflict sets: definitions (II)

(conflict set)

Let $\bar{a} = (a_{i_1}, \ldots, a_{i_k})$ be a consistent instantiation of an arbitrary subset of variables, and let x be a variable not yet instantiated. If there is no value b in the domain of x such that $(\bar{a}, x = b)$ is consistent, we say that $\bar{a}$ is a *conflict set* of x, or that $\bar{a}$ conflicts with variable x. If, in addition, $\bar{a}$ does not contain a subtuple that is in conflict with x, $\bar{a}$ is called a *minimal* conflict set of x. •

Note the instantiation refers to an arbitrary subset of variables

A minimal conflict set cannot be further reduced

Conflict sets: definitions (II)

(leaf dead-end)

Let $\vec{a}_i = (a_1, \ldots, a_i)$ be a consistent tuple. If $\vec{a}_i$ is in conflict with x_{i+1}, it is called a *leaf dead-end* and x_{i+1} is a leaf dead-end variable. •

(no-good)

Given a network $\mathcal{R} = (X, D, C)$, any partial instantiation $\bar{a}$ that does not appear in any solution of $\mathcal{R}$ is called a *no-good*. *Minimal* no-goods have no no-good subtuples. •

Non leaf dead-ends are called internal dead-ends

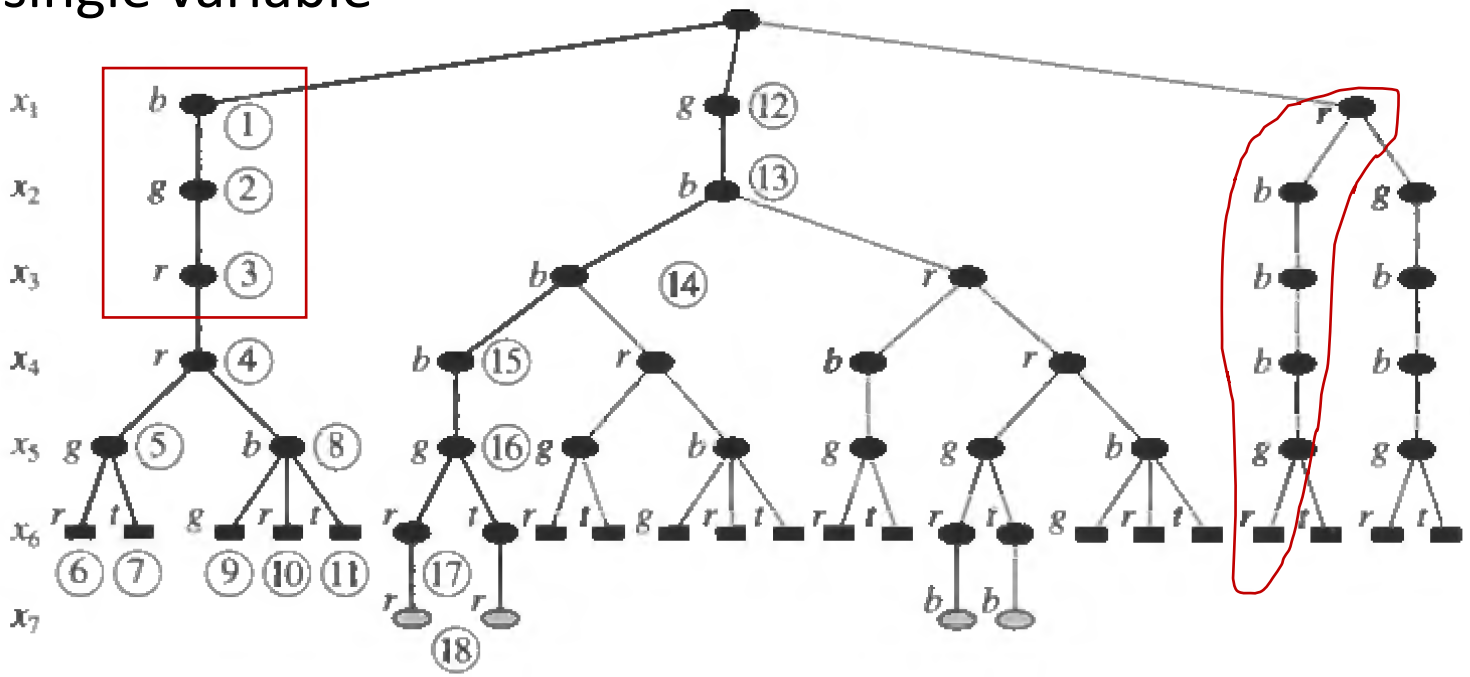
A conflict set is clearly a no-good

But some no-goods are not are not conflict sets of any single variable

I.e. they may conflict with two or more variables

Conflict sets: example

- Assignment (x_1=r,x_2=b,x_3=b,x_4=b,x_5=g,x_6=r) is conflict set relative to x_7
  - x_7 is a leaf dead-end
- (x_1=b,x_2=g,x_3=r) is a no-good
  - that is not conflict-set to a single variable



Safe jump

(safe jump)

Let $\vec{a}_i = (a_1, \ldots, a_i)$ be a leaf dead-end state. We say that x_j, where $j \leq i$, is *safe* (relative to $\vec{a}_i$) if the partial instantiation $\vec{a}_j = (a_1, \ldots, a_j)$ is a no-good, namely, it cannot be extended to a solution. •

Backjumping seeks to jump back as far as possible

Without skipping potential solutions

If x_j value is changed from a_j to another value, we will never explore again any solution that starts with $(a_1,\ldots a_j)$

Backjumping styles

- Gashnig's backjumping
  - Jumps back to the culprit variable
  - Jumps only at leaf dead-ends
- Graph-based backjumping
  - Extracts relevant information from the constraint graph
  - Jumps at internal dead-ends as well
- Conflict-directed backjumping
  - Backjumps at both leaf and internal dead-ends
  - Not restricted to graph information

Gashnig's Backjumping

(culprit variable)

Let $\vec{a}_i = (a_1, \ldots, a_i)$ be a leaf dead-end. The *culprit index* relative to $\vec{a}_i$ is defined by $b = min\{j \leq i | \vec{a}_j$ conflicts with $x_{i+1}\}$. We define the *culprit variable* of $\vec{a}_i$ to be x_b. •

At most *i* subtuples need to be tested for consistency with x_{i+1}

If $\vec{a}_i$ is a leaf dead-end and x_b is its culprit variable, then $\vec{a}_b$ is a safe backjump destination and $\vec{a}_j$, $j < b$, is not.

x_b is both safe and maximal

Gashnig's Backjumping: implementation

- Computation of culprit variable during search
- Each variable x_j mantains a variable $latest_j$
  - Points to the most recently instantiated predecessor found incompatible with any of the variables values
  - The algorithm jumps from a dead-end leaf variable x_{i+1} back to $x_{latest^{i+1}}$
- On subsequent dead-ends, the algorithm goes back to its preceeding variable only

Gashnig's Backjumping: algorithm (I)

procedure GASCHNIG'S-BACKJUMPING

Input: A constraint network $\mathcal{R} = (X, D, C)$.
Output: Either a solution, or a decision that the network is inconsistent.

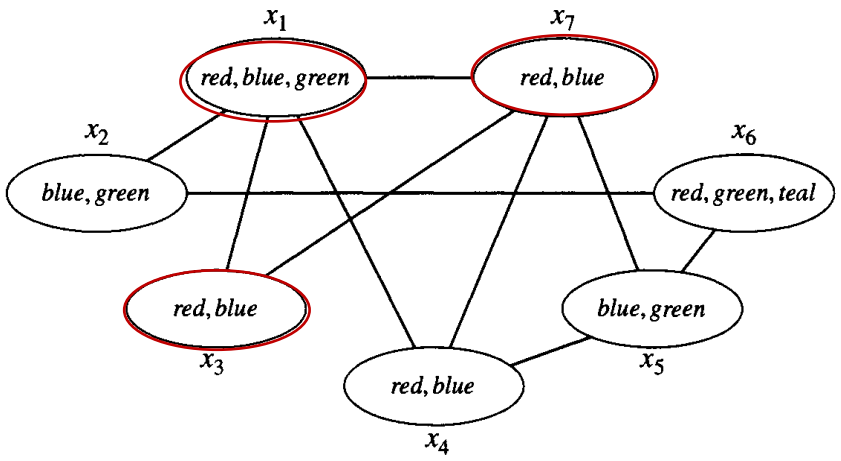
```
i ← 1                                   (initialize variable counter)
D_i' ← D_i                              (copy domain)
latest_i ← 0                            (initialize pointer to culprit)
while 1 ≤ i ≤ n
    instantiate x_i ← SELECT-VALUE-GBJ
    if x_i is null                      (no value was returned)
        i ← latest_i                    (backjump)
    else
        i ← i + 1
        D_i' ← D_i
        latest_i ← 0
end while
if i = 0
    return "inconsistent"
else
    return instantiated values of {x_1, ..., x_n}
end procedure
```

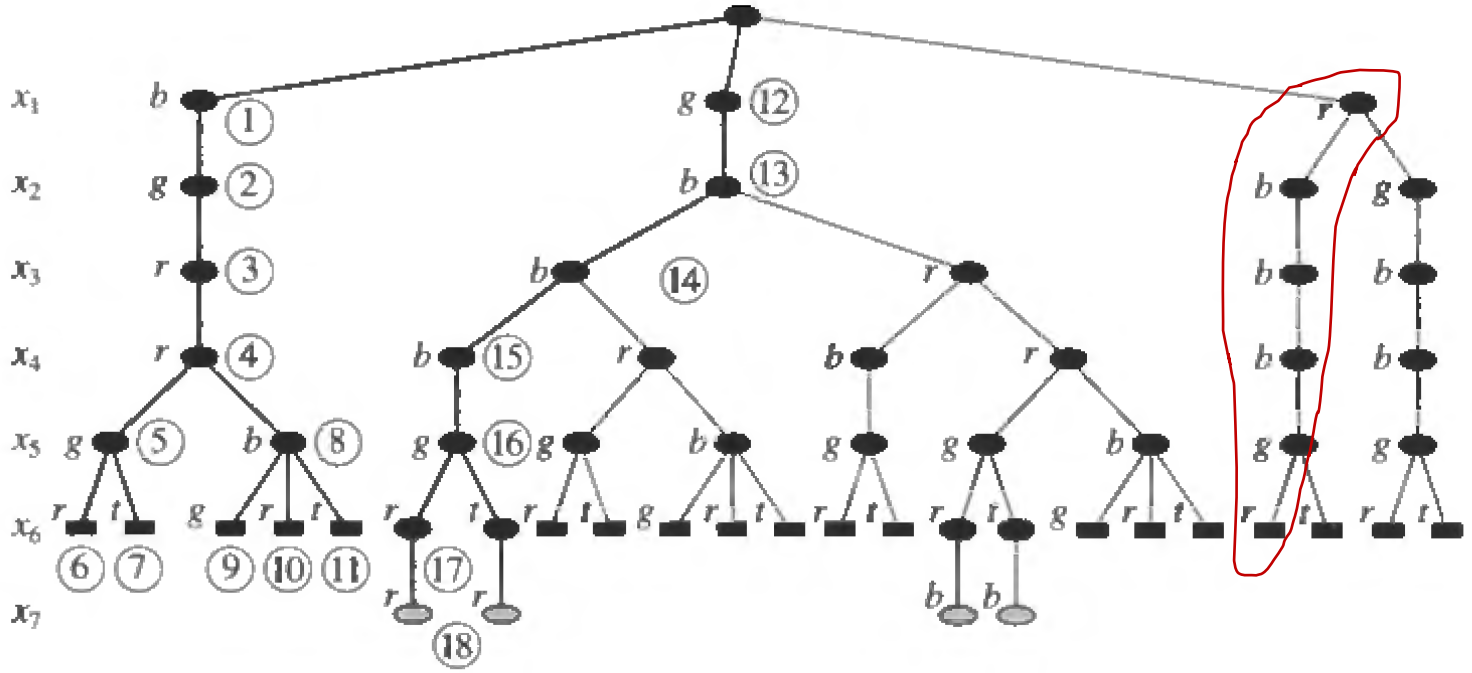
Gashnig's Backjumping: algorithm (II)

```
procedure SELECT-VALUE-GBJ
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        consistent ← true
        k ← 1
        while k < i and consistent
            if k > latest_i
                latest_i ← k
            if not CONSISTENT(ā_k, x_i = a)
                consistent ← false
            else
                k ← k + 1
        end while
        if consistent
            return a
    end while
    return null                          (no consistent value)
end procedure
```

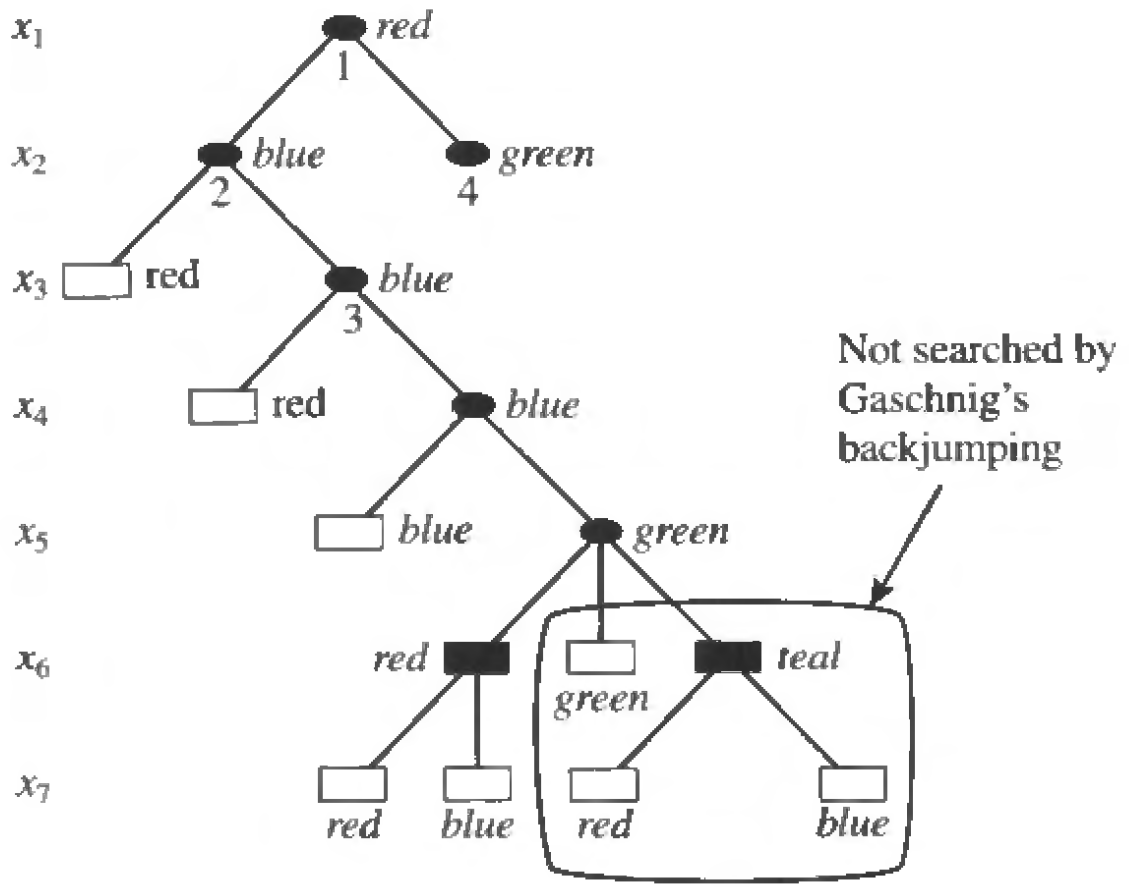
Gashnig's Backjumping: example (I)

- x_7 is leaf dead-end
  - Resulting from assignment (x_1=r,x_2=b,x_3=b,x_4=b,x_5=g,x_6=r)
- *latest*$_7$=3

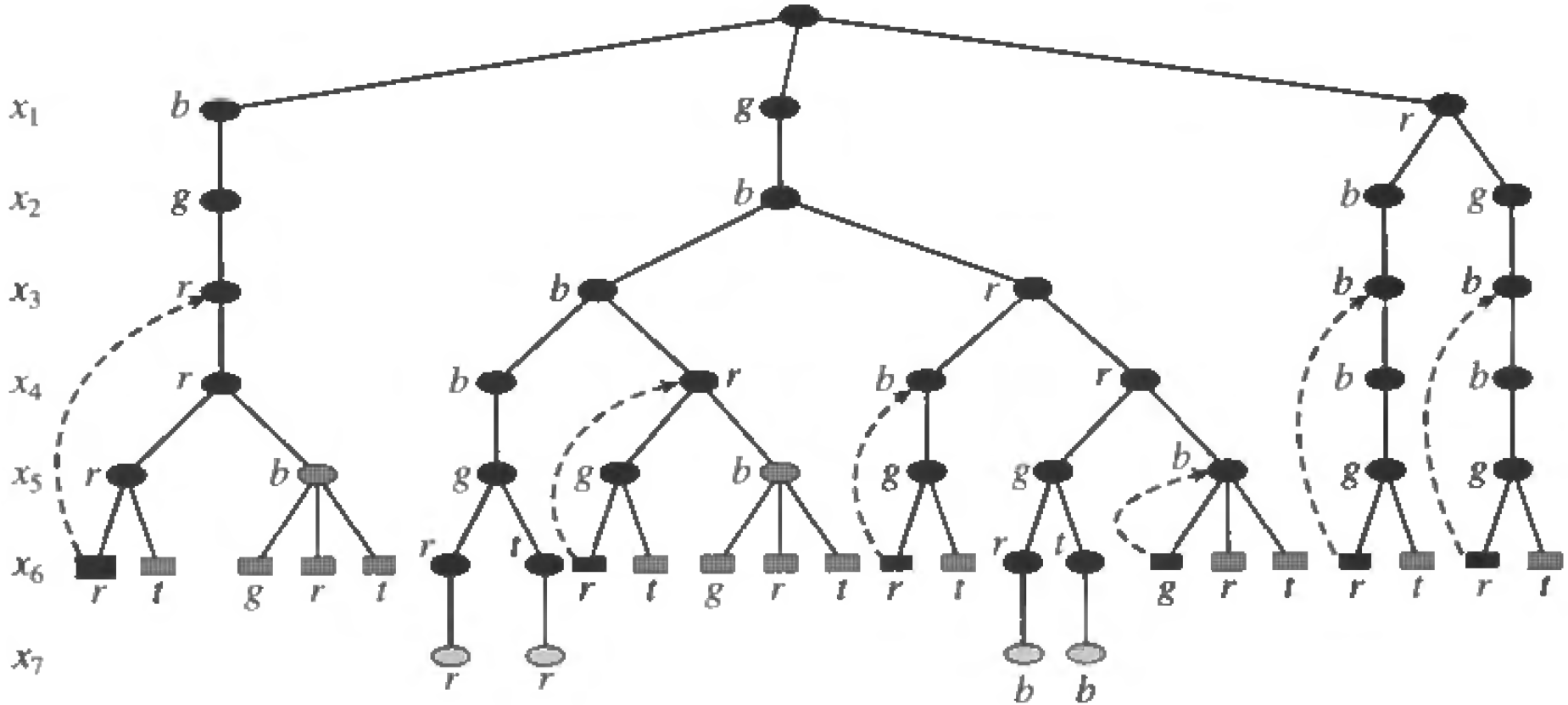




Gashnig's Backjumping: example (II)



Gashnig's Backjumping: example (III)



Graph-based backjumping

- If a backtracking algorithm jumps back to variable x_j from a leaf dead-end, and if x_j has no more candidate values…
  - Then x_j is internal dead-end
- See previous example…
  - All but one are internal dead-ends (exception x_1=g,x_2=b,x_3=r,x_4=b)

You have
3 minutes!

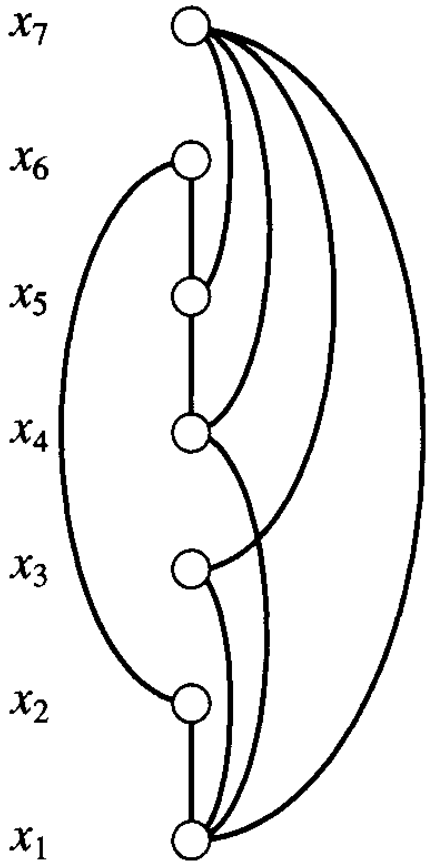
Graph-based backjumping

- Implements jumps at both internal and leaf dead-ends
- Uses pre-compiled information encoded in the graph
  - Avoids computing *latest* at each consistency test
- Uses the subset of earlier variables adjacent to x_{i+1} as an approximation of a minimal conflict set of x_{i+1}

Graph-based backjumping: ancestors, parent

(ancestors, parent)

Given a constraint graph and an ordering of the nodes d, the *ancestor set* of variable x, denoted $anc(x)$, is the subset of the variables that precede and are connected to x. The *parent* of x, denoted $p(x)$, is the most recent (or latest) variable in $anc(x)$. If $\vec{a}_i = (a_1, \ldots, a_i)$ is a dead-end, we equate $anc(\vec{a}_i)$ with $anc(x_{i+1})$, and $p(\vec{a}_i)$ with $p(x_{i+1})$. •



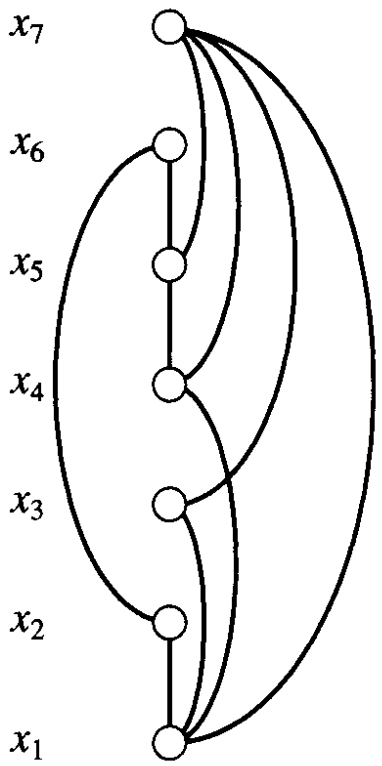
You have 3 minutes!

- anc(x_7)?
- p(x_7)?
- anc(x_7) = {x_1, x_3, x_4, x_5}
- p(x_7) = x_5
- Backjumping to p is safe!

Graph-based backjumping: example

- Assume leaf dead-end at node x_5; return to x_4
- If x_4 has no more values (internal dead-end), safely jump to x_1

- Leaf dead-end at x_7; jump back to x_5
- If x_5 is internal dead-end, go back to x_4
- x_4 is internal dead-end
- May we safely backtrack to x_1?
  - No! Changing the value of x_3 could undo dead-end x_7



Graph-based backjumping: relevant dead-ends

(invisit, session)

We say that backtracking *invisits* x_i if it processes x_i coming from a variable earlier in the ordering. The session of x_i starts upon the invisiting of x_i and ends when retracting to a variable that precedes x_i. At a given state of the search where variable x_i is already instantiated, the *current session* of x_i is the set of variables processed by the algorithm since the most recent *invisit* to x_i. The current session of x_i includes x_i, and therefore the session of a leaf dead-end variable has a single variable. •

(relevant dead-ends)

The relevant dead-ends of x_i's session, denoted $r(x_i)$, are defined recursively as follows. The relevant dead-ends in the session of a leaf dead-end x_i are just x_i. If x_i is a variable to which the algorithm retracted due to a dead-end at x_j, then the relevant dead-ends of $x_i's$ session are the union of its current relevant dead-ends and the ones inherited from x_j, namely, $r(x_i) = r(x_i) \cup r(x_j)$. •

Induced ancestors

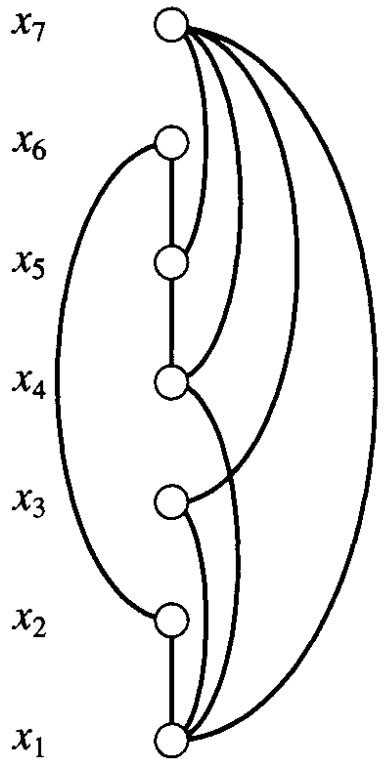
- Consider x_i a dead-end variable
- Y is the set of relevant dead-ends in x_i current *session*
- The induced ancestor set of x_i relative to Y ($I_i(Y)$) is the union of all Y's ancestors restricted to variables that preceed x_i

$$I_i(Y) = \cup_{y \in Y} anc(y) \cap \{x_1, \ldots, x_{i-1}\}$$

- The induced parent of x_i relative to Y is the latest variable in $I_i(Y)$
- $P_i(Y)$ is the graph-based culprit of x_i

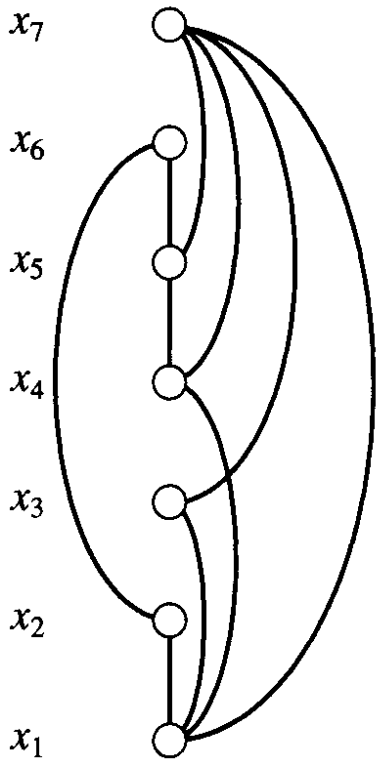
Induced ancestors: example (I)

- Assume x_4 a leaf dead-end variable
- Relevant dead-ends in x_4 session $Y = \{x_4\}$
- Induced ancestor set $I_4(Y) = \{x_1\}$
- Jump safely to x_1



Induced ancestors: example (II)

- Assume x_4 an internal dead-end variable
  - With Y = {x_4, x_5, x_6}, implying $I_4(Y)$ = {x_1, x_2}, jump to x_2
  - With Y = {x_4, x_5, x_7}, implying $I_4(Y)$ = {x_1, x_3}, jump to x_3
  - What if x_3 is also an internal dead-end?
  - $I_3(\{x_3, x_4, x_5, x_7\}) = \{x_1\}$, jump to x_1
  - With Y = {x_4, x_5, x_6, x_7}, $I_4(Y)$ = {x_1, x_2, x_3}, go to x_3
  - If x_3 is also an internal dead-end, then $I_3(\{x_3, x_4, x_5, x_6, x_7\}) = \{x_1, x_2\}$, , and so go to x_2



Conflict-directed backjumping

- Information is gathered during search, rather than being graph-based
  - Depending on constraints being processed
- The algorithm mantains an induced jumpback set
- Need a few definitions…
  - Earlier constraint
  - Earlier minimal conflict set
  - Jumpback set

> Given a dead-end tuple $\vec{a}_i$, the latest variable in its jumpback set J_i is the earliest variable to which it is safe to jump.

Earlier constraint

(earlier constraint)

Given an ordering of the variables in a constraint network, we say that constraint R is *earlier* than constraint Q if the latest variable in *scope*(R) − *scope*(Q) precedes the latest variable in *scope*(Q) − *scope*(R). •

- Consider ordering (x_1,x_2,…)
- Scope of R_1 is {x_3,x_5,x_8,x_9}
- Scope of R_2 is {x_2,x_6,x_8,x_9}
- Hence, R_1 is earlier than R_2
  - Because x_5 preceeds x_6

Earliest minimal conflict set (emc)

(earliest minimal conflict set)

For a network $\mathcal{R} = (X, D, C)$ with an ordering of the variables d, let $\vec{a}_i$ be a tuple whose potential dead-end variable is x_{i+1}. The *earliest minimal conflict set* of $\vec{a}_i$, (or of x_{i+1}) denoted $emc(\vec{a}_i)$, can be generated as follows. Consider the constraints in $C = \{R_1, \ldots, R_c\}$ with scopes $\{S_1, \ldots, S_c\}$, in order as defined in Definition 6.11. For $j = 1$ to c, if there exists $b \in D_{i+1}$ such that R_j is violated by $(\vec{a}_i, x_{i+1} = b)$, but no constraint earlier than R_j is violated by $(\vec{a}_i, x_{i+1} = b)$, then $var\text{-}emc(\vec{a}_i) \leftarrow var\text{-}emc(\vec{a}_i) \cup S_j$. $emc(\vec{a}_i)$ is the subtuple of $\vec{a}_i$ projected over $var\text{-}emc(\vec{a}_i)$. Namely, $emc(\vec{a}_i) = \vec{a}_i[var\text{-}emc(\vec{a}_i)]$. •

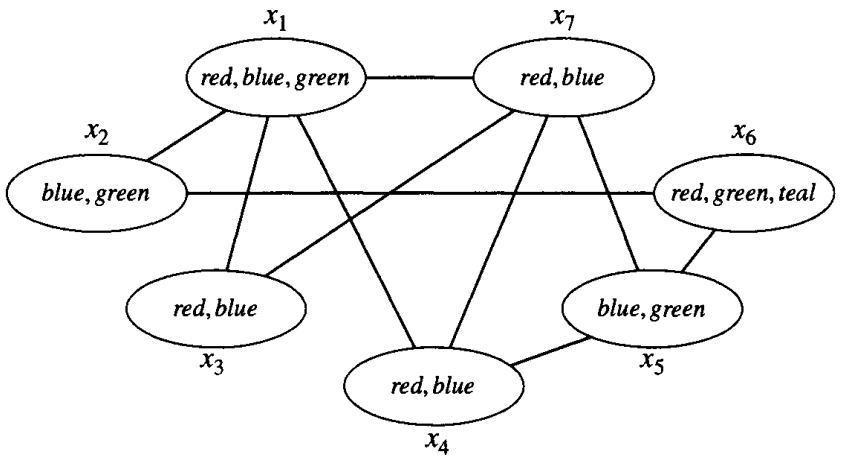
Jumpback set

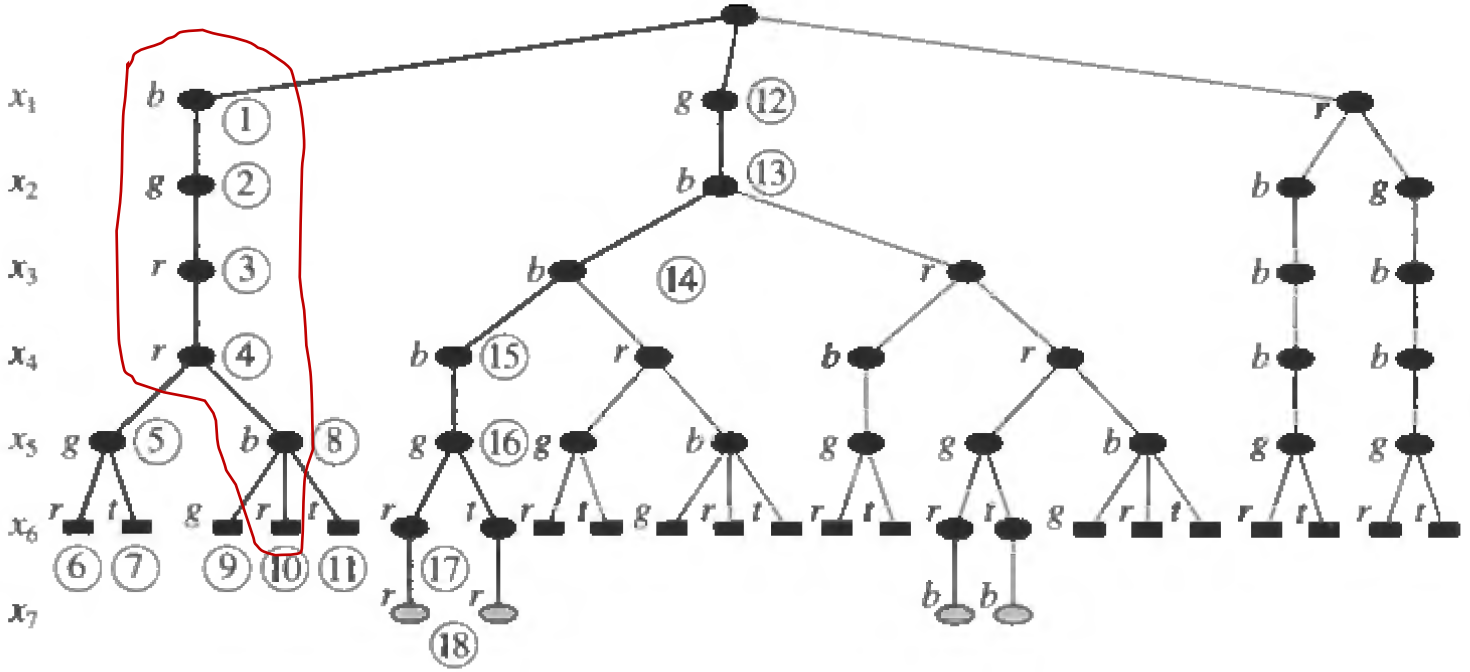
- The jumpback set J_{i+1} of a leaf dead-end x_{i+1} is its var-emc
- The jumpback set of an internal state includes var-emc of all relevant dead-ends in the current session

- Definition of relevant dead-ends the same as in graph-based
  - var-emc plays the same role as ancestors
  - J_i plays the role of induced ancestors

Conflict-directed backjumping: example

- Dead-end x_7 from assignment (b,g,r,r,b,r)
- Emc set is $\{x_1$=b,x_3=r$\}$, jump to x_3
- x_3 is internal dead-end with var-emc=$\{x_1\}$; jump to x_1





Learning algorithms

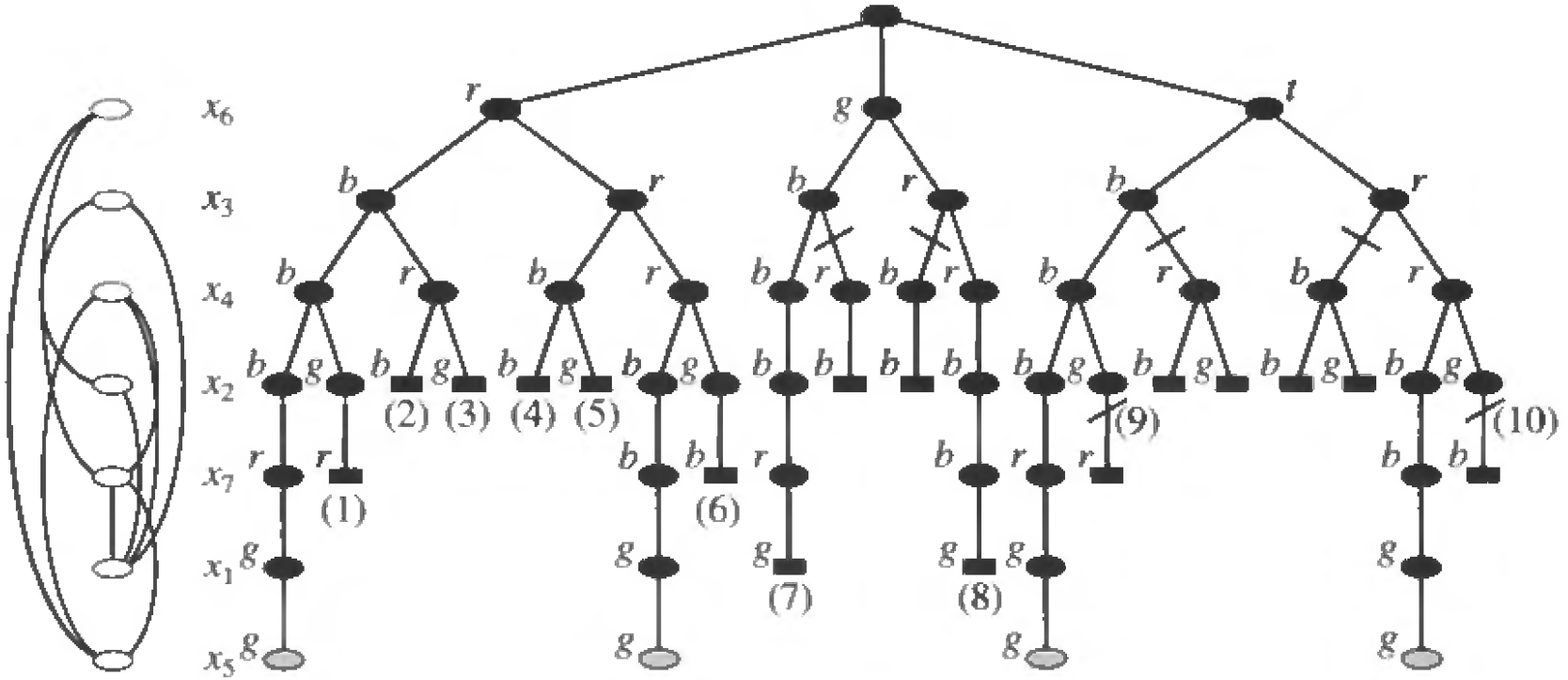
- No-goods are made explicit as new constraints
  - The same no-good may be rediscovered in different paths
- Opportunity to apply constraint recording / learning
  - Whenever a dead-end is found
  - Graph- or constraint-based
- Ideally identify conflict sets as small as possible (minimal)

An example: graph-based learning

- Information on conflicts is derived from the constraint graph
- In leaf dead-ends consider the ancestors
- In internal dead-ends consider the induced ancestor set

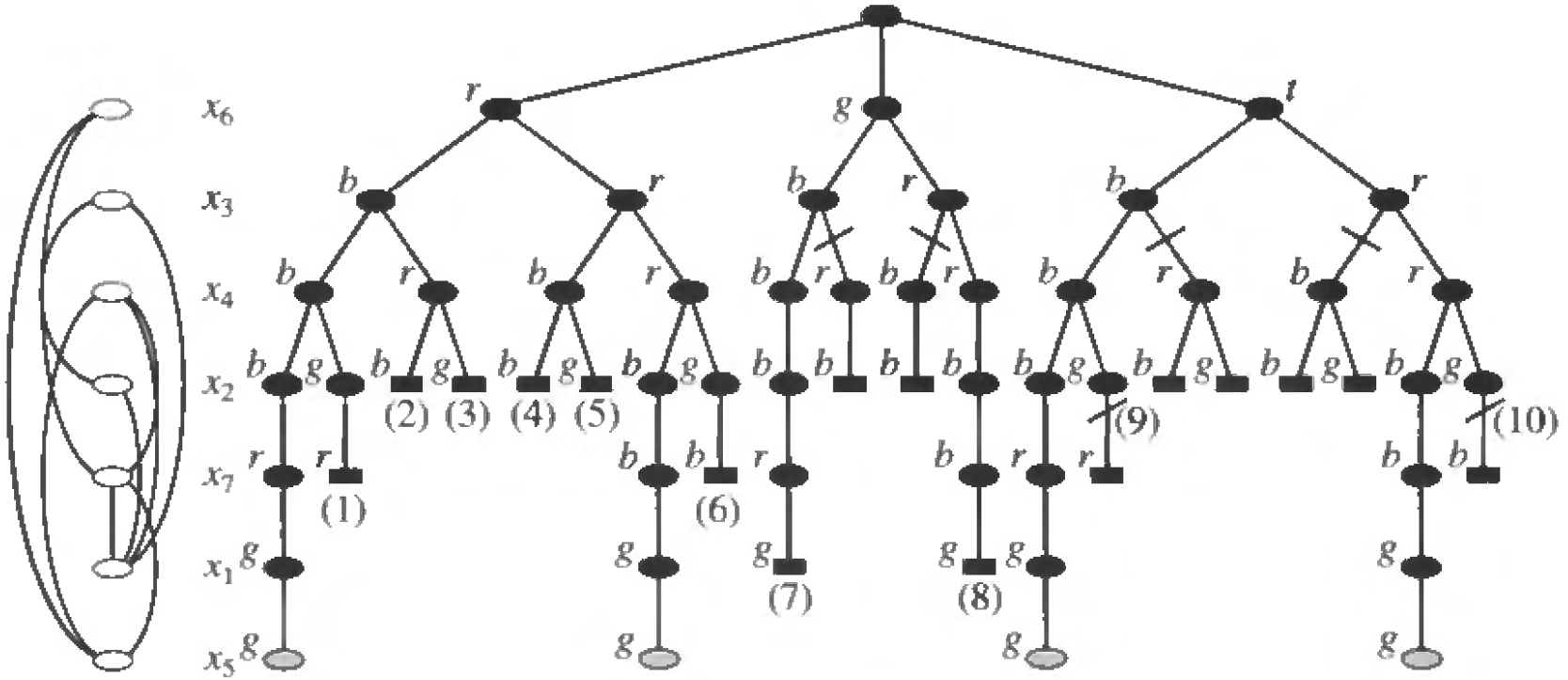
Graph-based learning: example (I)

- Consider dead-end (1)
- anc(x_1)={x_2,x_3,x_4,x_7}
- Record no-good (x_2=g,x_3=b,x_4=b,x_7=r)
  - Appears later in the search and prunes dead-end (9)



Graph-based learning: example (II)

- Dead-ends (2) and (4)
- $anc(x_7)=\{x_3,x_4\}$
- Record no-goods $(x_3=b,x_4=r)$ and $(x_3=r,x_4=b)$
  - Prune dead-ends (3) and (5)
- Homework: analyse other dead-ends



Summary

- Look-back methods: backjumping and learning
- Graph-based and constraint-based approaches


That's all Folks!