Backtracking search: look-back

ICS 275
Winter 2016

Look-back:
Backjumping / Learning

- Backjumping:
  - In deadends, go back to the most recent culprit.

- Learning:
  - constraint-recording, no-good recording.
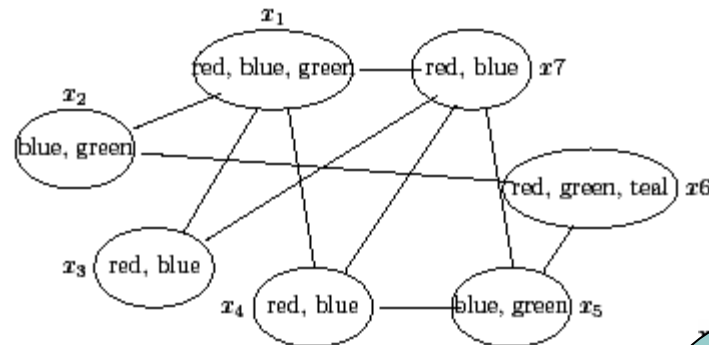 - good-recording

Backjumping

Figure 6.1: **A** modified coloring problem.

- (X1=r,x2=b,x3=b,x4=b,x5=g,x6=r,x7={r,b})
- (r,b,b,b,g,r) **conflict set** of x7
- (r,-,b,b,g,-) c.s. of x7
- (r,-,b,-,-,-,-) **minimal conflict-set**
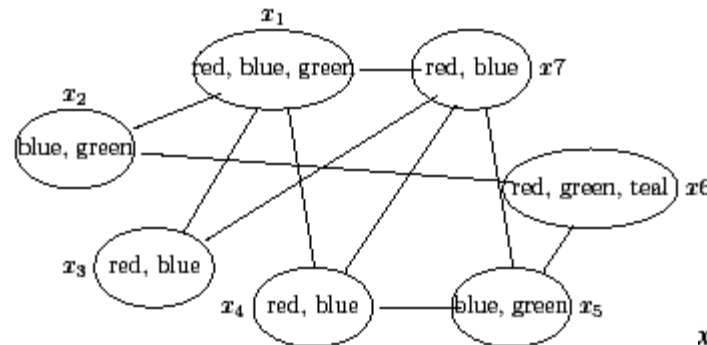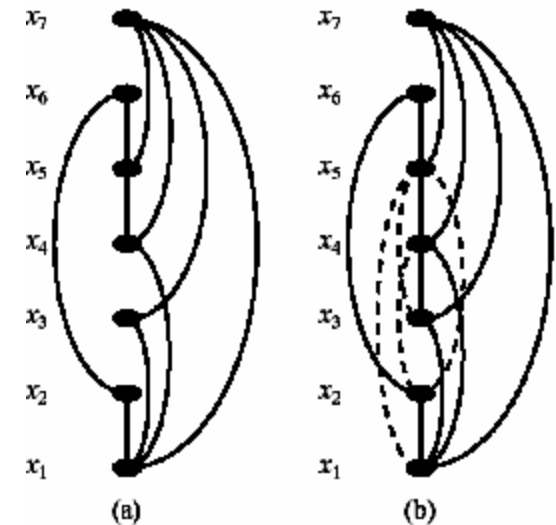- **Leaf deadend**: (r,b,b,b,g,r)
- Every conflict-set is a **no-good**

Backjumping

Figure 6.1: A modified coloring problem.

- (X1=r,x2=b,x3=b,x4=b,x5=g,x6=r,x7={r,b})
- (r,b,b,b,g,r) **conflict set** of x7
- (r,-,b,b,g,-) c.s. of x7
- (r,-,b,-,-,-,-) **minimal conflict-set**
- **Leaf deadend**: (r,b,b,b,g,r)
- Every conflict-set is a **no-good**

Flavor of Gaschnig's jumps only at leaf-dead-ends
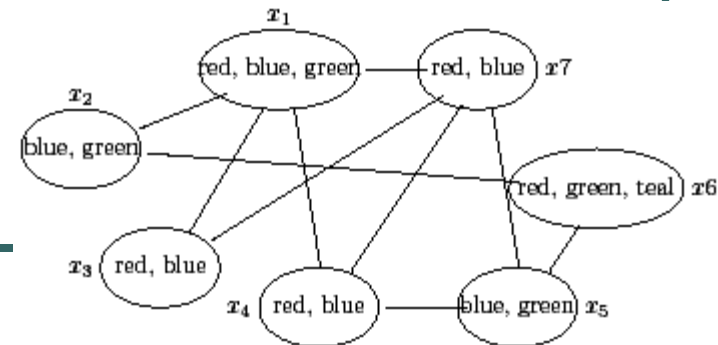
Internal dead-ends: dead-ends that are non-leaf
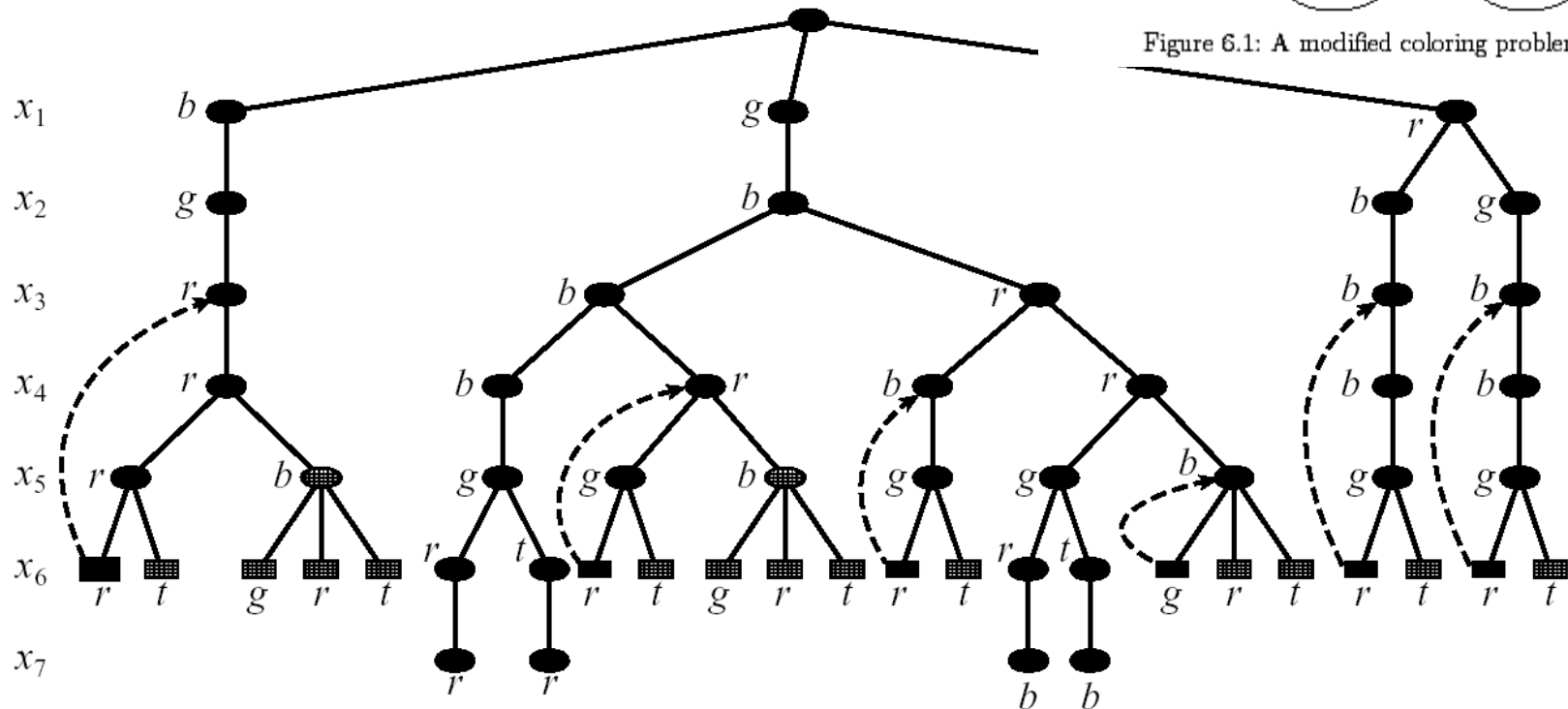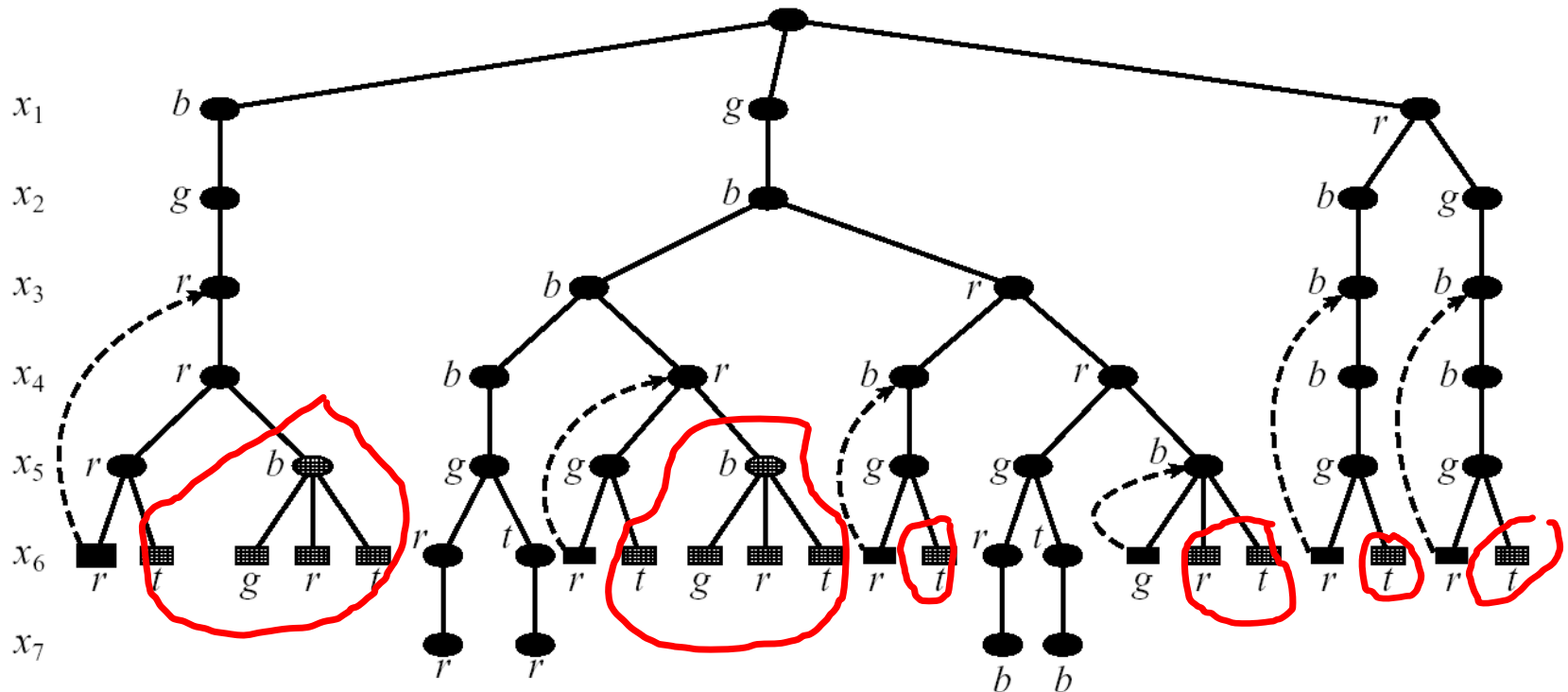

Figure 6.1: A modified coloring problem.

Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to $(\langle x_1, green \rangle, \langle x_2, blue \rangle, \langle x_3, red \rangle, \langle x_4, blue \rangle)$, because this is the only case where another value exists in the domain of the culprit variable. □

Gaschnig jumps only at leaf-dead-ends

Internal dead-ends: dead-ends that are non-leaf

Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to $(\langle x_1, green \rangle, \langle x_2, blue \rangle, \langle x_3, red \rangle, \langle x_4, blue \rangle)$, because this is the only case where another value exists in the domain of the culprit variable. $\square$

Backjumping styles

- Jump at leaf only (Gaschnig 1977)
  - Context-based

- Graph-based (Dechter, 1990)
  - Jumps at leaf and internal dead-ends, graph information

- Conflict-directed (Prosser 1993)
  - Context-based, jumps at leaf and internal dead-ends

Conflict Analysis
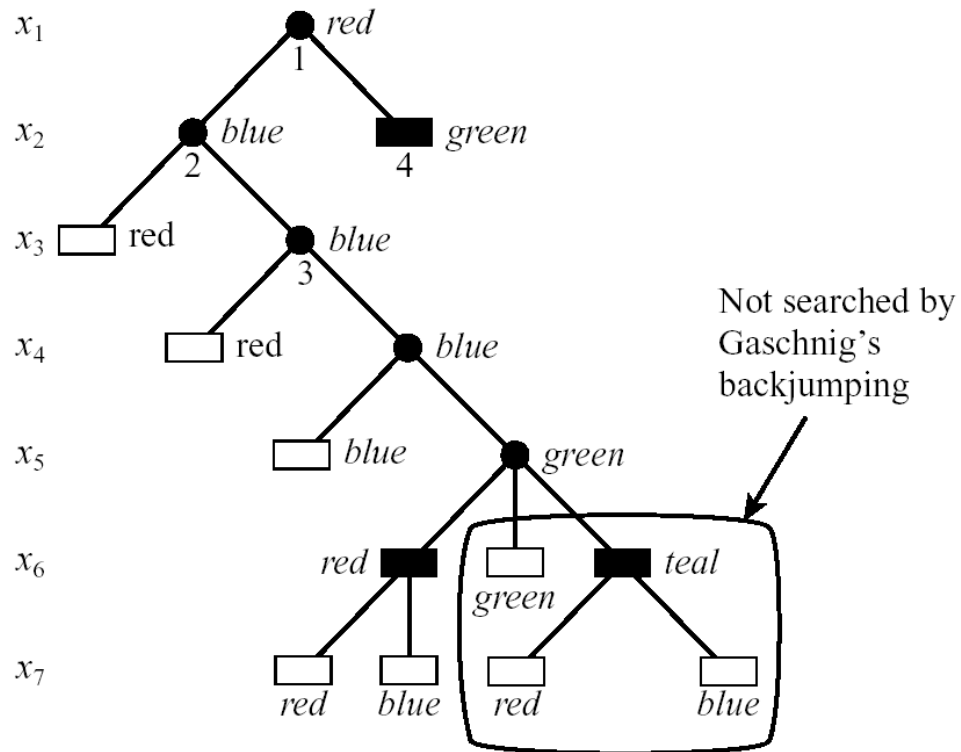
- Conflict set
- Leaf deadend
- Nogood
- Safe jump

Gaschnig's backjumping:
Culprit variable
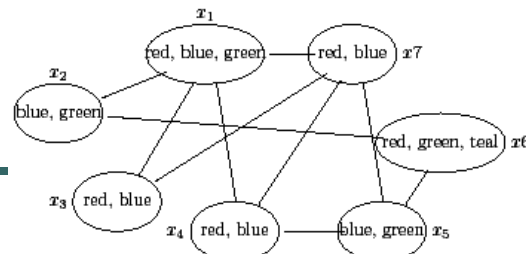
Figure 6.1: A modified coloring problem.
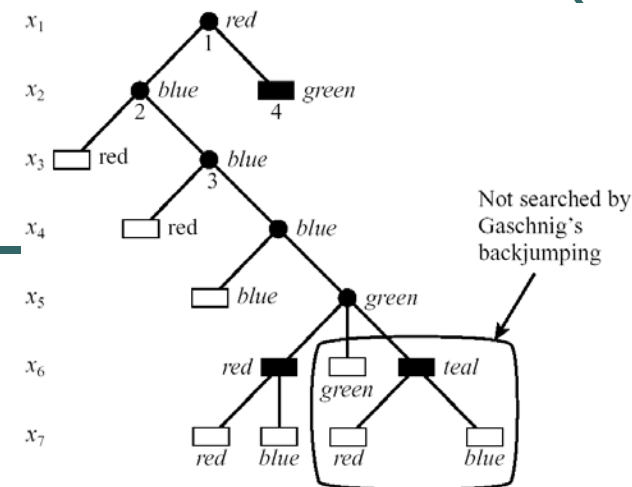
Definition 6.2.1 (culprit variable) Let $\vec{a}_i = (a_1, ..., a_i)$ be a leaf dead-end. The culprit index relative to $\vec{a}_i$ is defined by $b = \min\{j \leq i \mid \vec{a}_j \ conflicts \ with \ x_{i+1}\}$. We define the culprit variable of $\vec{a}_i$ to be x_b.

- If a_i is a leaf deadend and x_b its culprit variable, then a_b is a safe backjump destination and a_j , j<b is not.
- The culprit of x_7 (r,b,b,b,g,r) is (r,b,b) → x_3

Gaschnig's backjumping implementation [1979]

- Gaschnig uses a marking technique to compute culprit.

- Each variable x_j maintains a pointer ($latest_j$) to the latest ancestor incompatible with any of its values.

- While forward generating a_i, keep array $latest_i$, 1<=j<=n, of pointers to the last value conflicted with some value of x_j. The algorithm jumps from a leaf-dead-end x_{i+1} back to $latest_{i+1}$ which is its culprit.

Gaschnig's backjumping

```
procedure GASCHNIG'S-BACKJUMPING
Input: A constraint network 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 ← SELECTVALUE-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

procedure SELECTVALUE-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(a⃗_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
```
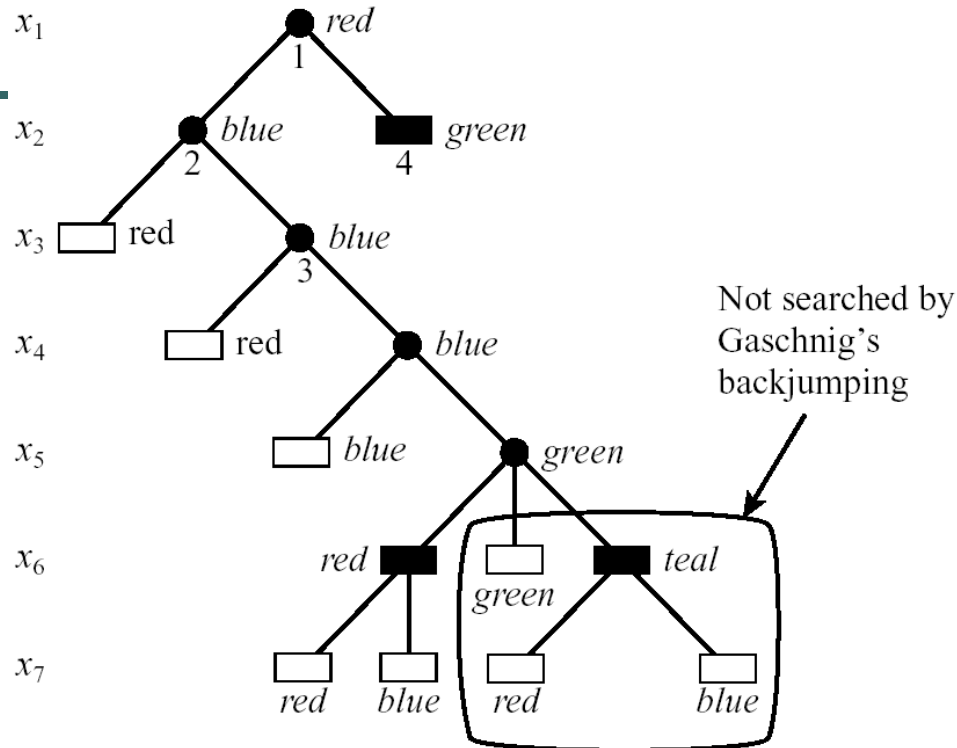
Winter 2016

Figure 6.3: Gaschnig's backjumping algorithm.

Example of Gaschnig's backjump

x_1 • red
    1

x_2 • blue   ■ green
    2   4

x_3 □ red   • blue
    3

x_4 □ red   • blue

Not searched by Gaschnig's backjumping

x_5 □ blue   • green

x_6 red ■   □ green   ■ teal

x_7 □ red   □ blue   □ red   □ blue

Example 6.2.3 Consider the problem in Figure 6.1 and the order d_1. At the dead-end for x_7 that results from the partial instantiation ($< x_1, red >, < x_2, blue >, < x_3, blue >, < x_4, blue >, < x_5, green >, < x_6, red >$), $latest_7 = 3$, because $x_7 = red$ was ruled out by $< x_1, red >$, $x_7 = blue$ was ruled out by $< x_3, blue >$, and no later variable had to be examined. On returning to x_3, the algorithm finds no further values to try ($D'_3 = \emptyset$). Since $latest_3 = 2$, the next variable examined will be x_2. Thus we see the algorithm's ability to backjump at leaf dead-ends. On subsequent dead-ends, as in x_3, it goes back to its preceding variable only. An example of the algorithm's practice of pruning the search space is given in Figure 6.2.

Winter 2016   □

Properties

- Gaschnig's backjumping implements only safe and maximal backjumps in leaf-deadends.

Gaschnig jumps only at leaf-dead-ends
Internal dead-ends: dead-ends that are non-leaf

Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to $(\langle x_1, green \rangle, \langle x_2, blue \rangle, \langle x_3, red \rangle, \langle x_4, blue \rangle)$, because this is the only case where another value exists in the domain of the culprit variable. □
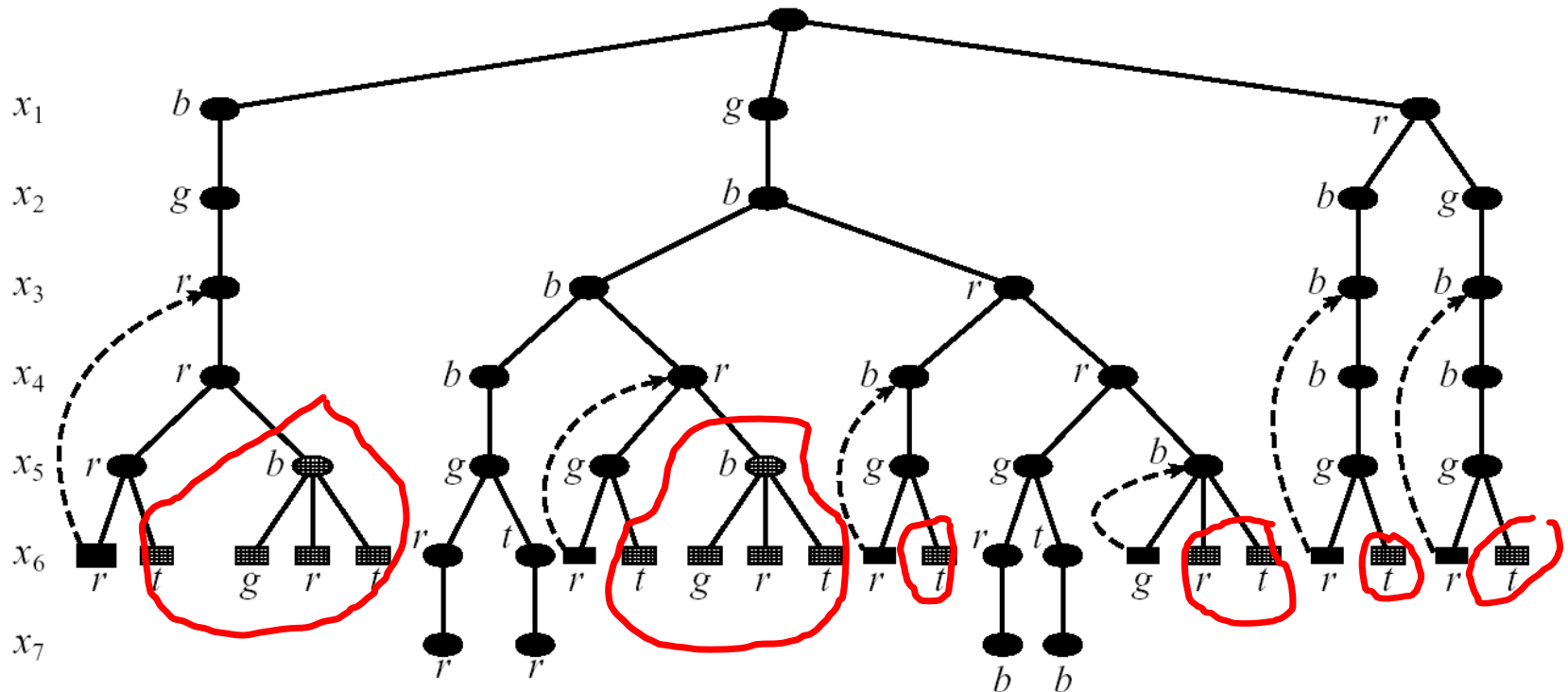
Gaschnig jumps only at leaf-dead-ends
Internal dead-ends: dead-ends that are non-leaf

Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to $(\langle x_1, green \rangle, \langle x_2, blue \rangle, \langle x_3, red \rangle, \langle x_4, blue \rangle)$, because this is the only case where another value exists in the domain of the culprit variable. □

Graph-based backjumping scenarios Internal deadend at X4

Figure 6.1: **A** modified coloring problem.

- Scenario 1, deadend at x4:

- Scenario 2: deadend at x5:

- Scenario 3: deadend at x7:

- Scenario 4: deadend at x6:
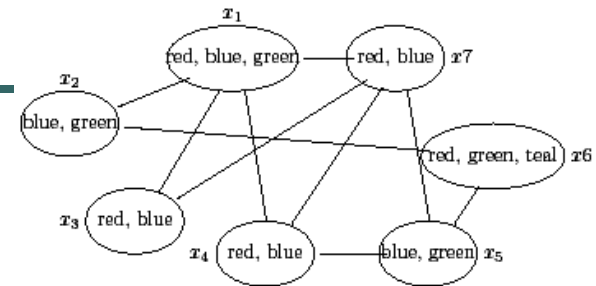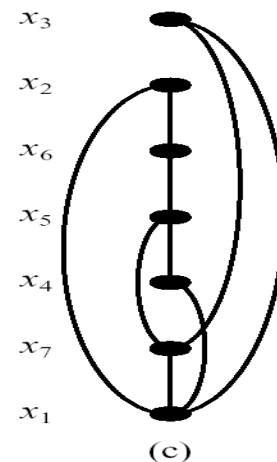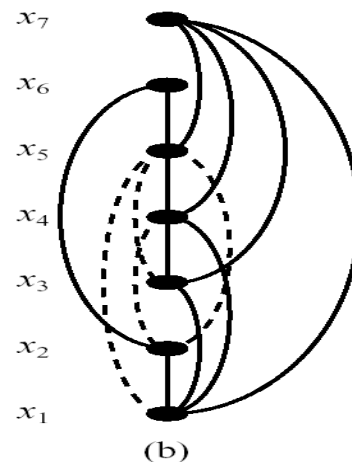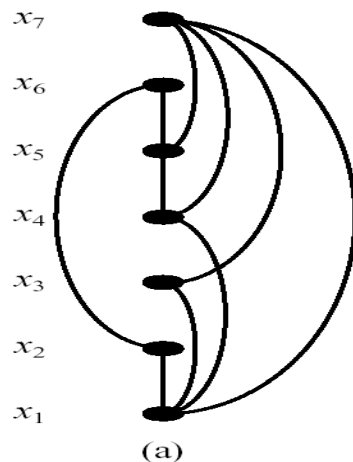
Graph-based backjumping

- Uses only graph information to find culprit
- Jumps both at leaf and at internal dead-ends
- Whenever a deadend occurs at x, it jumps to the most recent variable y connected to x in the graph. If y is an internal deadend it jumps back further to the most recent variable connected to x or y.
- The analysis of conflict is approximated by the graph.
- Graph-based algorithm provide graph-theoretic bounds.

Ancestors and parents

- $\text{anc}(x_7) = \{x_5 x_3 x_4 x_1\}$
- $p(x_7) = 5$
- $p(r,b,b,b,g,r) = x_5$
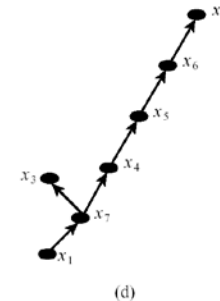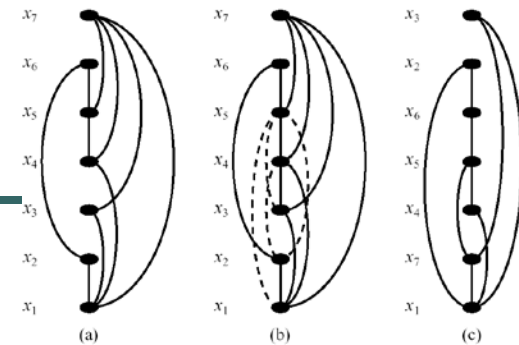
Definition 6.3.2 (ancestors, parent) *Given a constraint graph and an ordering of the nodes d, the* ancestor set *of variable x, denoted* $\text{anc}(x)$, *is the subset of the variables that precede and are connected to x. The* parent *of x, denoted* $p(x)$, *is the most recent (or latest) variable in* $\text{anc}(x)$. *If* $\vec{a_i} = (a_1, ..., a_i)$ *is a leaf dead-end, we equate* $\text{anc}(\vec{a_i})$ *with* $\text{anc}(x_{i+1})$, *and* $p(\vec{a_i})$ *with* $p(x_{i+1})$.

Internal deadends analysis

Definition 6.3.5 (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.*

Definition 6.3.6 (relevant dead-ends) *The relevant dead-ends of x_i's session are defined recursively as follows. The relevant dead-ends of a leaf dead-end x_i, denoted $r(x_i)$, is x_i. If x_i is variable to which the algorithm retracted from x_j, then the relevant-dead-*

The induced-parents of a variable X along an ordering, approximates its induced parent set

induced ancestor set of x_i relative to Y, $I_i(Y)$, is the union of all Y's ancestors, restricted to variables that precede x_i. Formally, $I_i(Y) = anc(Y) \cap \{x_1, ..., x_{i-1}\}$. The induced parent of x_i relative to Y, $P_i(Y)$, is the latest variable in $I_i(Y)$. We call $P_i(Y)$ the graph-based culpribt of x_i.

Graph-based backjumping scenarios Internal deadend at X4

Figure 6.1: **A** modified coloring problem.

- Scenario 1, deadend at x4:

- Scenario 2: deadend at x5:

- Scenario 3: deadend at x7:

- Scenario 4: deadend at x6:

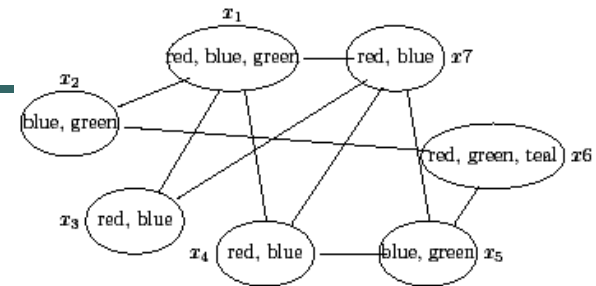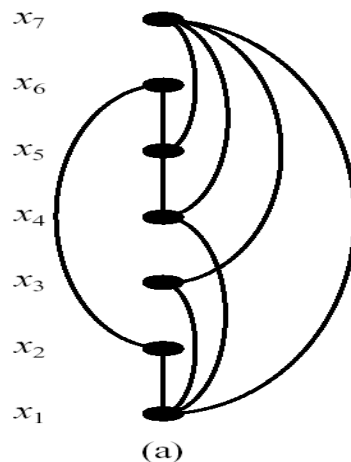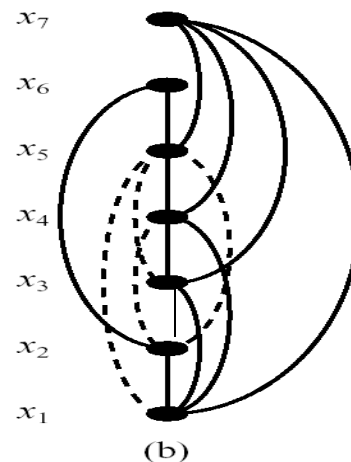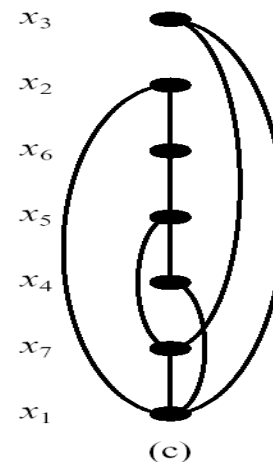# Graph-based backjumping algorithm, but we need to jump at internal deadends too

```
procedure GRAPH-BASED-BACKJUMPING
Input: A constraint network R = (X, D, C)
Output: Either a solution, or a decision that the network is inconsistent.

    compute anc(x_i) for each x_i (see Definition 6.3.2 in text)
    i ← 1                        (initialize variable counter)
    D'_i ← D_i                   (copy domain)
    I_i ← anc(x_i)               (copy of anc() that can change)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE
        if x_i is null           (no value was returned)
            iprev ← i
            i ←latest index in I_i    (backjump)
            I_i ← I_i ∪ I_iprev − {x_i}
        else
            i ← i + 1
            D'_i ← D_i
            I_i ← anc(x_i)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE          (same as BACKTRACKING's)
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        if CONSISTENT(ā_{i-1}, x_i = a)
            return a
    end while
    return null                (no consistent value)
end procedure
```

When not all variables
In the session above
X_i are relevant deadends?
read example 6.6

Figure 6.5: The graph-based backjumping algorithm.

Properties of graph-based backjumping

- Algorithm graph-based backjumping jumps back at any deadend variable as far as graph-based information allows.

- For each variable, the algorithm maintains the induced-ancestor set l_i relative the relevant dead-ends in its current session.

- The size of the induced ancestor set is at most w*(d).

Conflict-directed backjumping
(Prosser 1990)

- Extend Gaschnig's backjump to internal dead-ends.
- Exploits information gathered during search.
- For each variable the algorithm maintains an induced **jumpback set**, and jumps to most recent one.
- **Use the following concepts:**
  - An ordering over variables induced a strict ordering between constraints: $R_1 < R_2 < \ldots R_t$
  - Use **earliest minimal consflict-set** ($\text{emc}(x_{i+1})$) of a deadend.
  - Define the **jumpback set** of a deadend

Example of conflict-directed backjumping
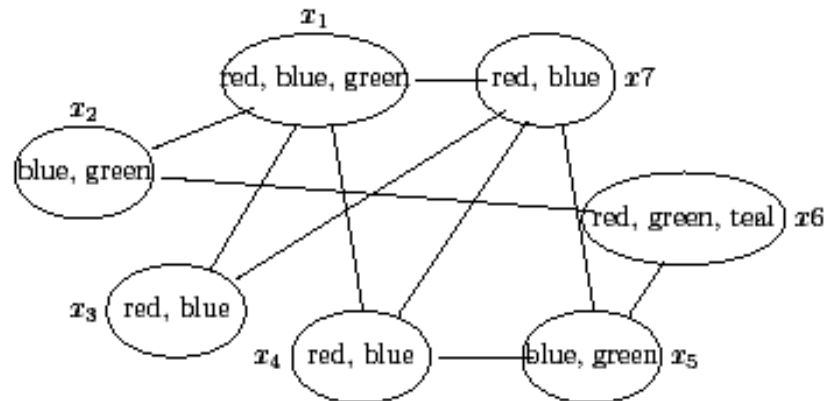
Figure 6.1: **A** modified coloring problem.

Example 6.4.5 Consider the problem of Figure 6.1 using ordering $d_1 = (x_1, \ldots, x_7)$. Given the dead-end at x_7 and the assignment $\vec{a_6} = (blue, green, red, red, blue, red)$, the emc set is $(< x_1, blue >, < x_3, red >)$, since it accounts for eliminating all the values of x_7. Therefore, algorithm conflict-directed backjumping jumps to x_3. Since x_3 is an internal dead-end whose own $var - emc$ set is $\{x_1\}$, the jumpback set of x_3 includes just x_1, and the algorithm jumps again, this time back to x_1. $\square$

Properties

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

- This is the culprit.

- Algorithm conflict-directed backtracking jumps back to the latest variable in the dead-ends's jumpback set, and is therefore safe and maximal.

Conflict-directed backjumping

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

$i \leftarrow 1$ (initialize variable counter)
$D'_i \leftarrow D_i$ (copy domain)
$J_i \leftarrow \emptyset$ (initialize conflict set)
while $1 \leq i \leq n$
    instantiate $x_i \leftarrow$ SELECTVALUE-CBJ
    if x_i is null   (no value was returned)
        $iprev \leftarrow i$
        $i \leftarrow$ index of last variable in J_i (backjump)
        $J_i \leftarrow J_i \cup J_{iprev} - \{x_i\}$(merge conflict sets)
    else
        $i \leftarrow i + 1$ (step forward)
        $D'_i \leftarrow D_i$ (reset mutable domain)
        $J_i \leftarrow \emptyset$ (reset conflict set)
end while
if $i = 0$
    return "inconsistent"
else
    return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure

subprocedure SELECTVALUE-CBJ

    while D'_i is not empty
        select an arbitrary element $a \in D'_i$, and remove a from D'_i
        $consistent \leftarrow true$
        $k \leftarrow 1$
        while $k < i$ and $consistent$
           if CONSISTENT($\vec{a}_k, x_i = a$)
             $k \leftarrow k + 1$
          else
             let R_S be the earliest constraint causing the conflict
             add the variables in R_S's scope S, but not x_i, to J_i
             $consistent \leftarrow false$
        end while
        if $consistent$
           return a
    end while
    return null   (no consistent value)
end procedure

Winter 2016

Figure 6.7: The conflict-directed backjumping algorithm.

Graph-Based Backjumping on DFS Orderings

Example 6.5.1 Consider, once again, the CSP in Figure 6.1. A *DFS* ordering $d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3)$ and its corresponding *DFS* spanning tree are given in Figure 6.6c,d. If a dead-end occurs at node x_3, the algorithm retreats to its *DFS* parent, which is x_7. □
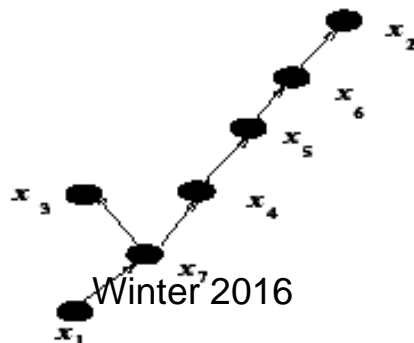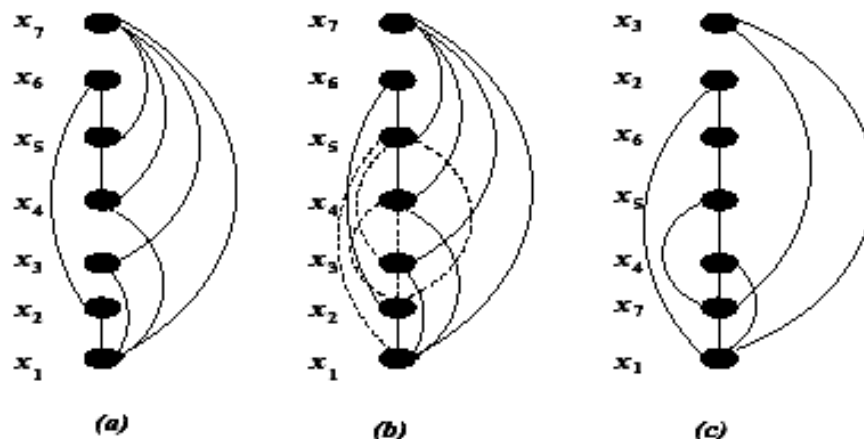
Graph-based Backjumping on DFS ordering

- Example: $d = x_1, x_2, x_3, x_4, x_5, x_6, x_7$
- Constraints: (6,7)(5,2)(2,3)(5,7)(2,7)(2,1)(2,3)(1,4)3,4)
- Rule: go back to parent. No need to maintain parent set
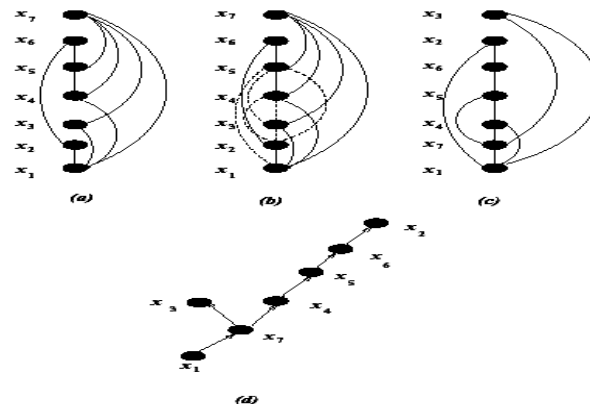
Figure 6.6: Several ordered constraint graphs of the problem in Figure 6.1: (a) along ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, (b) the induced graph along d_1, (c) along ordering $d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3)$, and (d) a DFS spanning tree along ordering d_2.

Theorem 6.5.2 *Given a DFS ordering of the constraint graph, if $f(x)$ denotes the DFS parent of x, then, upon a dead-end at x, $f(x)$ is x's graph-based earliest safe variable for both leaf and internal dead-ends.*

Winter 2016

Complexity of Graph-based Backjumping

- T_i = number of nodes in the AND/OR search space rooted at x_i (level m-i)
- Each assignment of a value to x_i generates subproblems:
  - T_i = k b T_{i-1}
  - T_0 = k
- Solution: $T_m = b^m k^{m+1}$

Theorem 6.5.3 *When graph-based backjumping is performed on a DFS ordering of the constraint graph, the number of nodes visited is bounded by $O((b^m k^{m+1}))$, where b bounds the branching degree of the DFS tree associated with that ordering, m is its depth and k is the domain size. The time complexity (measured by the number of consistency checks) is $O(ek(bk)^m)$, where e is the number of constraints.*

DFS of graph and induced graphs
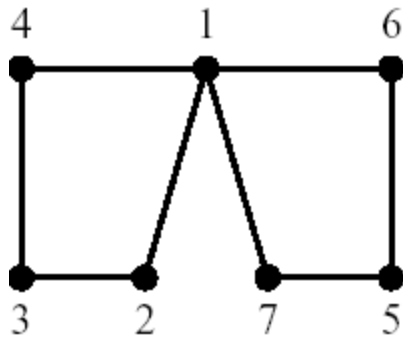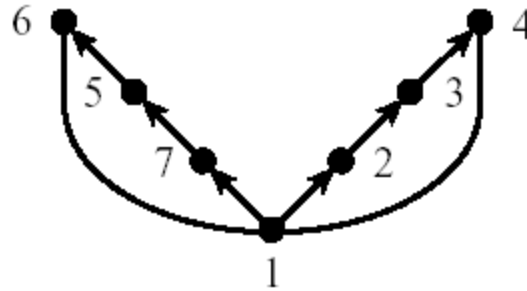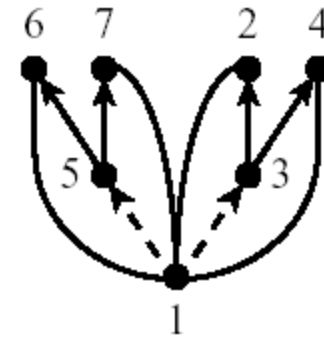
(a) (b) (c)

Spanning-tree of a graph;
DFS spanning trees, Pseudo-tree
Pseudo-tree is a spanning tree that does not allow arcs across branches.

Complexity of backjumping uses pseudo-tree analysis
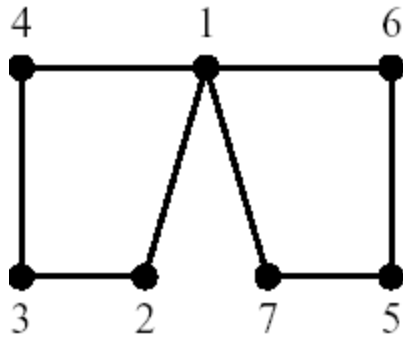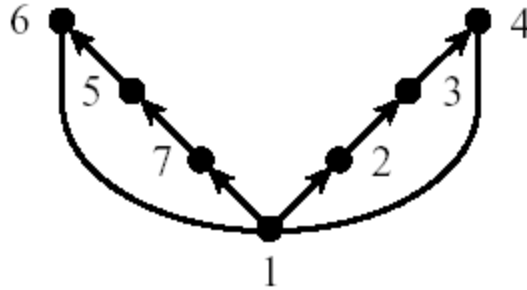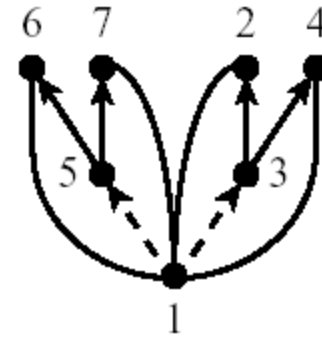
(a)   (b)   (c)

Simple: always jump back to parent in pseudo tree
Complexity for csp: exp(tree-depth)
Complexity for csp: exp(w*log n)

Complexity of backjumping
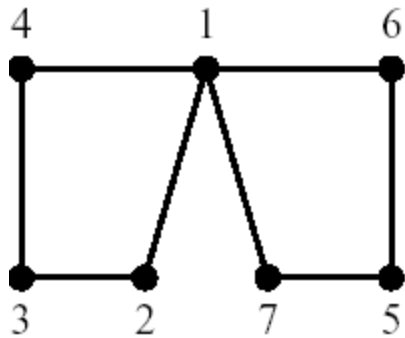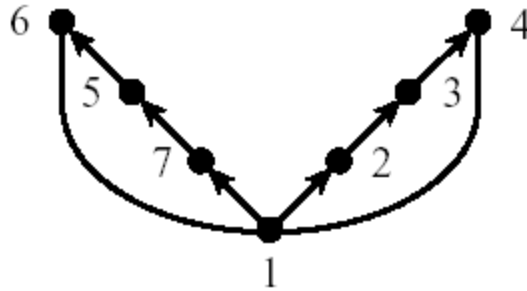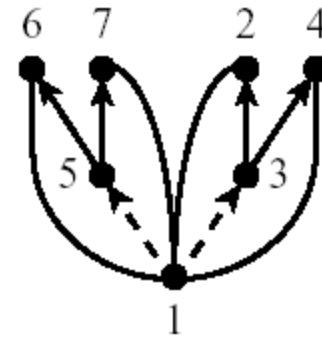
(a)   (b)   (c)

Theorem 6.5.5 *If d is a DFS ordering of (G^*, d_1) for some ordering d_1, having depth m_d^*, then the complexity of graph-based backjumping using ordering d is $O(\exp(m_d^*))$.*

Look-back: No-good Learning

Learning means recording conflict sets used as constraints to prune future search space.
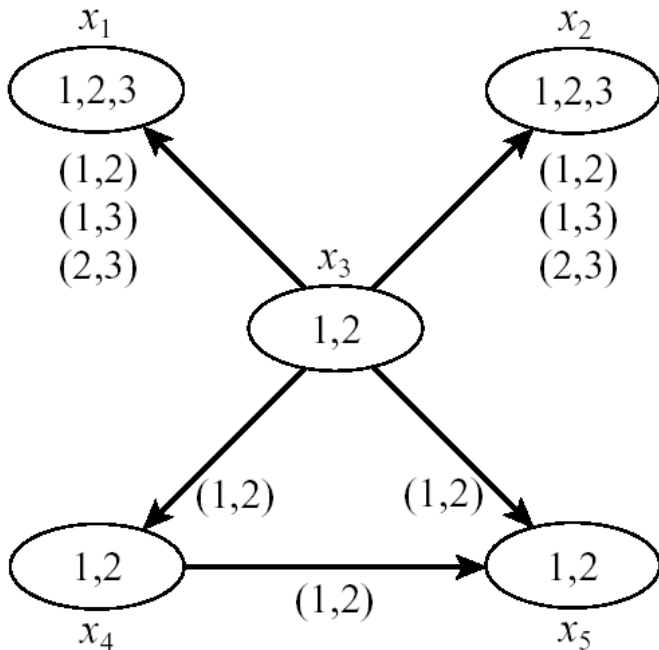
- $(x_1=2, x_2=2, x_3=1, x_4=2)$ is a dead-end

- Conflicts to record:
  - $(x_1=2, x_2=2, x_3=1, x_4=2)$ 4-ary
  - $(x_3=1, x_4=2)$ binary
  - $(x_4=2)$ unary

Learning, constraint recording

- Learning means recording conflict sets
- An opportunity to learn is when deadend is discovered.
- Goal of learning is to not discover the same deadends.
- Try to identify small conflict sets
- Learning prunes the search space.

Nogoods explain deadends

Learning means recording explanations to conflicts
They are implied constraints
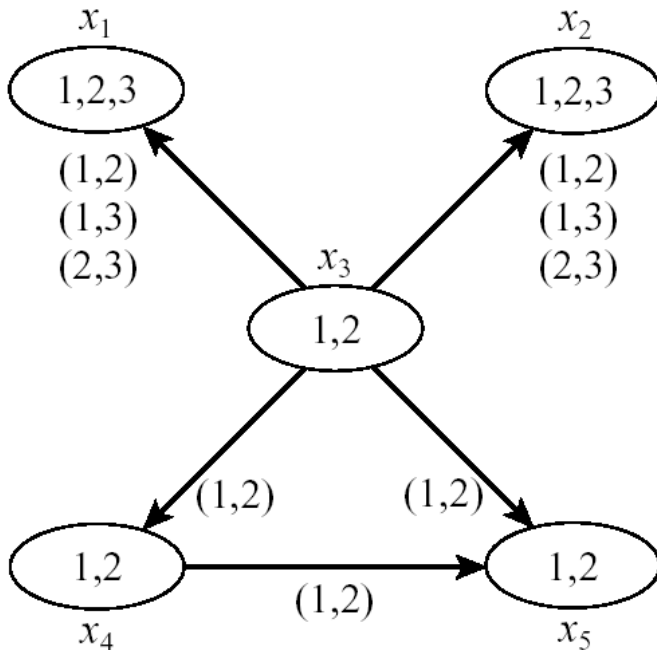
- Conflicts to record are explanations
  - (x1=2,x2=2,x3=1,x4=2) 4-ary
  - (x1=2,x2=2,x3=1,x4=2)→ (x5 ≠1) and
  - (x3=1,x4=2) →   (x5 ≠1)
  - (x4=2)   → (x5 ≠1)

Learning example
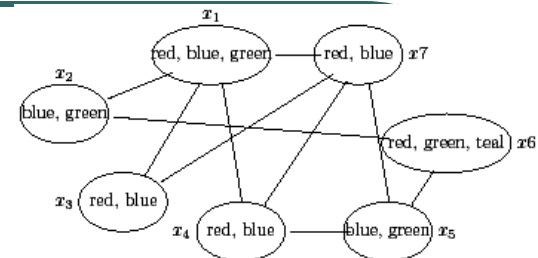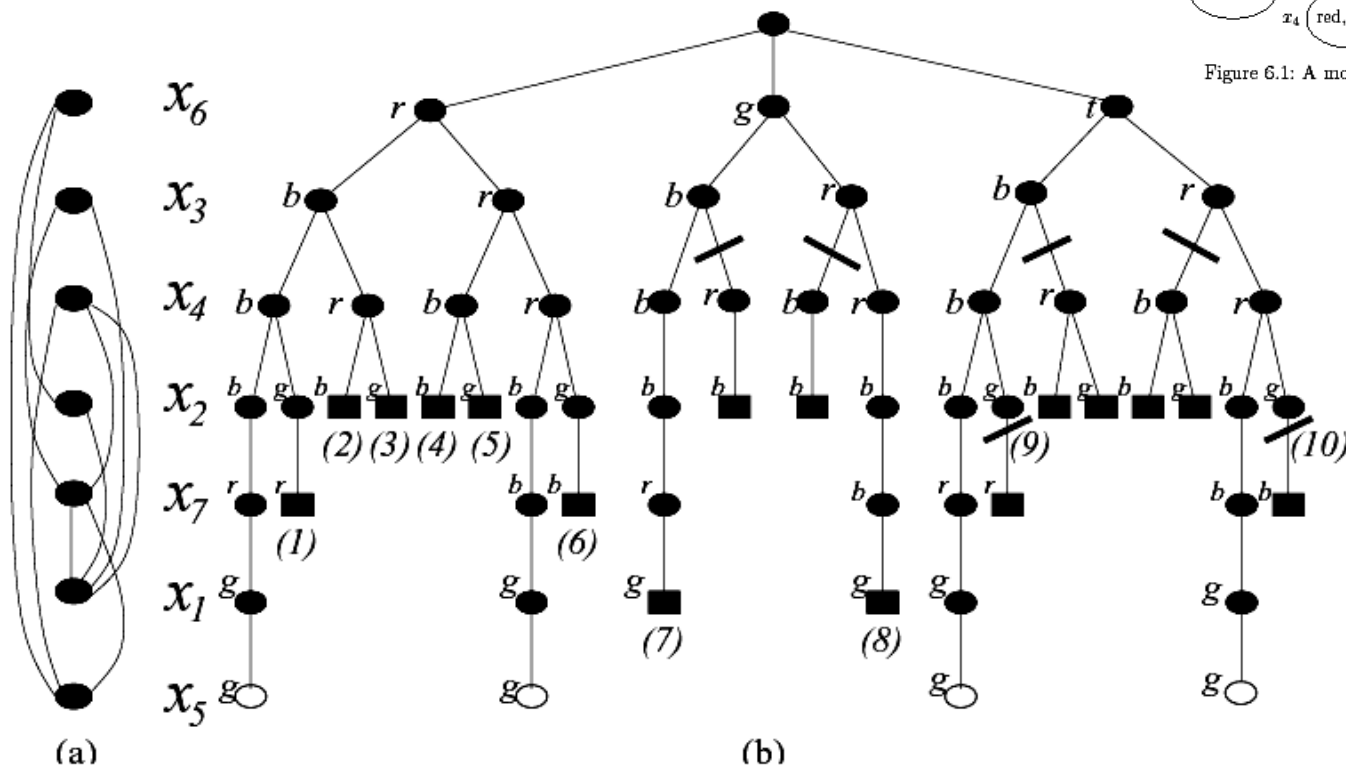
Figure 6.1: **A modified coloring problem.**

Figure 6.9: The search space explicated by backtracking on the CSP from Figure 6.1, using the variable ordering $(x_6, x_3, x_4, x_2, x_7, x_1, x_5)$ and the value ordering (*blue, red, green, teal*). Part (a) shows the ordered constraint graph, part (b) illustrates the search space. The cut lines in (b) indicate branches not explored when graph-based learning is used.

Learning issues

- Learning styles
    - Graph-based or context-based
    - i-bounded, scope-bounded
    - Relevance-based
- Non-systematic randomized learning
- Implies time and space overhead
- Applicable to SAT

Graph-based learning algorithm

procedure GRAPH-BASED-BACKJUMP-LEARNING

    instantiate $x_i \leftarrow$ SELECTVALUE
    if x_i is null   (no value was returned)
        record a constraint prohibiting $\vec{a}_{i-1}[I_i]$.
        $iprev \leftarrow i$
        (algorithm continues as in Fig. 6.5)

Figure 6.10: Graph-based backjumping learning, modifying CBJ

Deep learning

- Deep learning: recording all and only minimal conflict sets

- Example:

- Although most accurate, overhead can be prohibitive: the number of conflict sets in the worst-case:

$$\binom{r}{r/2} = 2^r$$

Jumpback learning

Figure 6.1: A modified coloring problem.

- Record the jumpback assignment

Example _____ For the problem and ordering of Example _____ at the first dead-end, jumpback learning will record the no-good $(x_2 = green, x_3 = blue, x_7 = red)$, since that tuple includes the variables in the jumpback set of x_1. □

---
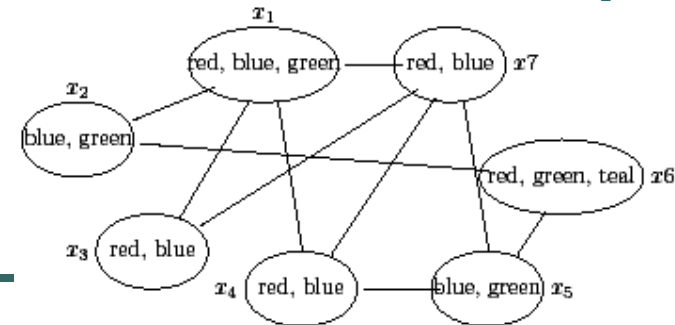
procedure CONFLICT-DIRECTED-BACKJUMP-LEARNING

    instantiate $x_i \leftarrow$ SELECTVALUE-CBJ

    if x_i is null   (no value was returned)

        record a constraint prohibiting $\vec{a}_{i-1}[J_i]$ **and corresponding values**

        $iprev \leftarrow i$

        (algorithm continues as in Fig. 6.7)

Figure 6.11: Conflict-directed bakjump-learning, modifying CBJ

Bounded and relevance-based learning

Bounding the arity of constraints recorded.

- When bound is i: i-ordered graph-based,i-order jumpback or i-order deep learning.

- Overhead complexity of i-bounded learning is time and space exponential in i.

Definition 6.7.3 (i-relevant) *A no-good is i-relevant if it differs from the current partial assignment by at most i variable-value pairs.*

Definition 6.7.4 (i'th order relevance-bounded learning) *An i'th order relevance-bounded learning scheme maintains only those learned no-goods that are i-relevant.*

Complexity of backtrack-learning for CSP

- The complexity of learning along d is time and space exponential in w*(d):

The number of dead-ends is bounded by $O(nk^{w*(d)})$

Number of constraint tests per dead-end are $O(e)$

Space complexity is $O(nk^{w*(d)})$

Time complexity is $O(n^2 \cdot k^{w*(d)+1})$

m- depth of tree, e- number of constraints

Good caching:
Moving from one to all or counting
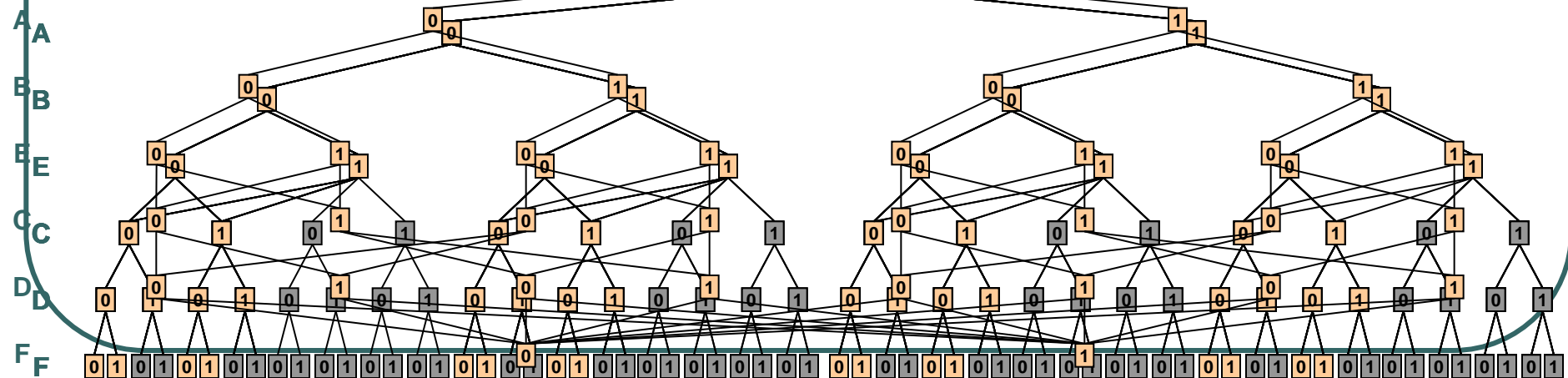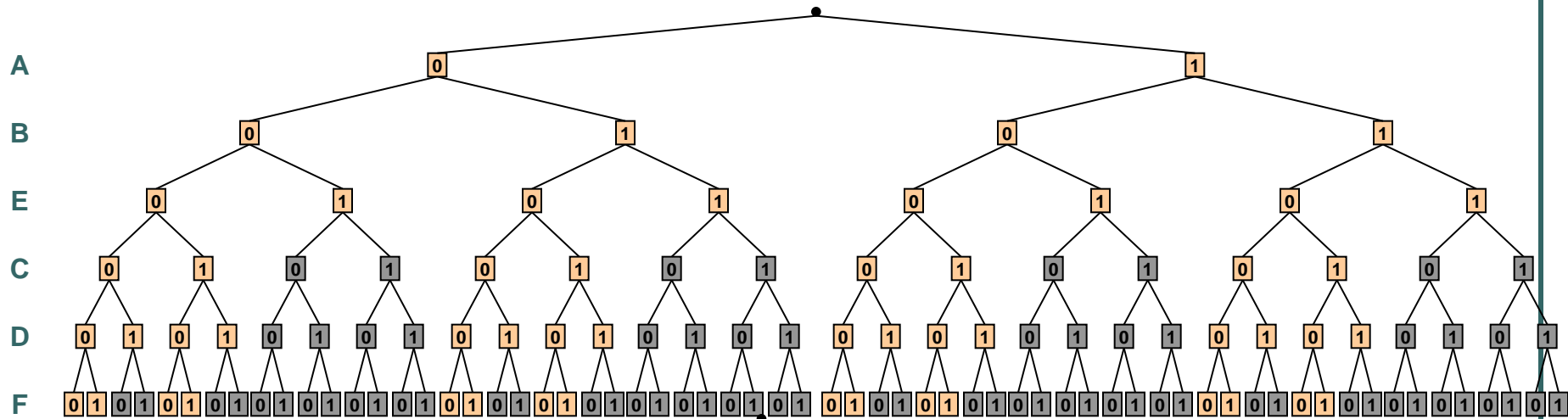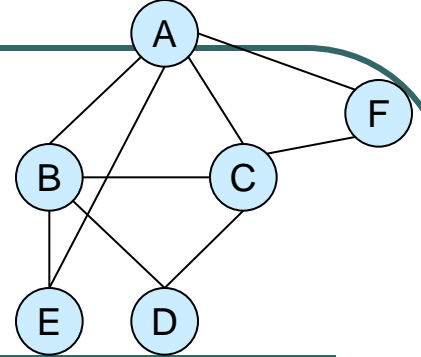
Summary:
Time-space for constraint processing (consistency and counting)

- **Constraint-satisfaction**
  - Search with backjumping
    - Space: linear, Time: $O(nk^{\log n\, w^*})$
  - Search with learning no-goods
    - time and space: $O(nk^{w*})$
  - Variable-elimination
    - time and space: $O(nk^{w*})$
- **Counting, enumeration**
  - Search with backjumping
    - Space: linear, Time: $O(k^n)$
  - Search with no-goods caching only
    - space: O(exp(w*)) Time: O(exp(n))
  - Search with goods and no-goods learning
    - Time and space: O(exp(path-width), O(exp(log n w*))
  - Variable-elimination
    - Time and space: O(exp(w*))

Non-Systematic Randomized Learning

- Do search in a random way with interupts, restarts, unsafe backjumping, **but record conflicts**.
- Guaranteed completeness.

Look-back for SAT

- A partial assignment is a set of literals: σ
- A jumpback set if a J-clause:
- Upon a leaf deadend of x resolve two clauses, one enforcing x and one enforcing $\neg x$ relative to the current assignment
- A clause forces x relative to assignment σ if all the literals in the clause are negated in σ.
- Resolving the two clauses we get a nogood.
- If we identify the earliest two clauses we will find the earliest conflict.
- The argument can be extended to internal deadends.

Integration of algorithms

procedure FC-CBJ

Input: A constraint network $\mathcal{R} = (X, D, C)$.

Output: Either a solution, or a decision that the network is inconsistent.

$i \leftarrow 1$ (initialize variable counter)

call SELECTVARIABLE   (determine first variable)

$D'_i \leftarrow D_i$ for $1 \leq i \leq n$ (copy all domains)

$J_i \leftarrow \emptyset$ (initialize conflict set)

while $1 \leq i \leq n$

     instantiate $x_i \leftarrow$ SELECTVALUE-FC-CBJ

     if x_i is null   (no value was returned)

         $iprev \leftarrow i$

         $i \leftarrow$ latest index in J_i (backjump)

         $J_i \leftarrow J_i \cup J_{iprev} - \{x_i\}$

         reset each $D'_k, k > i$, to its value before x_i was last instantiated

     else

         $i \leftarrow i + 1$ (step forward)

         call SELECTVARIABLE   (determine next variable)

         $D'_i \leftarrow D_i$

         $J_i \leftarrow \emptyset$

end while

if $i = 0$

     return "inconsistent"

else

     return instantiated values of $\{x_1, \ldots, x_n\}$

end procedure

Figure 6.12: The main procedure of the FC-CBJ algorithm.

subprocedure SELECTVALUE-FC-CBJ

    while D'_i is not empty
        select an arbitrary element $a \in D'_i$, and remove a from D'_i
        empty-domain $\leftarrow$ *false*
        for all $k, i < k \leq n$
            for all values b in D'_k
                if not CONSISTENT$(\vec{a}_{i-1}, x_i = a, x_k = b)$
                    let R_S be the earliest constraint causing the conflict
                    add the variables in R_S's scope S, but not x_k, to J_k
                    remove b from D'_k
            endfor
            if D'_k is empty   ($x_i = a$ leads to a dead-end)
                empty-domain $\leftarrow$ *true*
        endfor
        if *empty-domain*   (don't select a)
            reset each D'_k and $j_k, i < k \leq n$, to status before a was selected
        else
            return a
    end while
    return null   (no consistent value)
end subprocedure

Figure 6.14: The SelectValue subprocedure for FC-CBJ.

Relationships between various backtracking algrithms
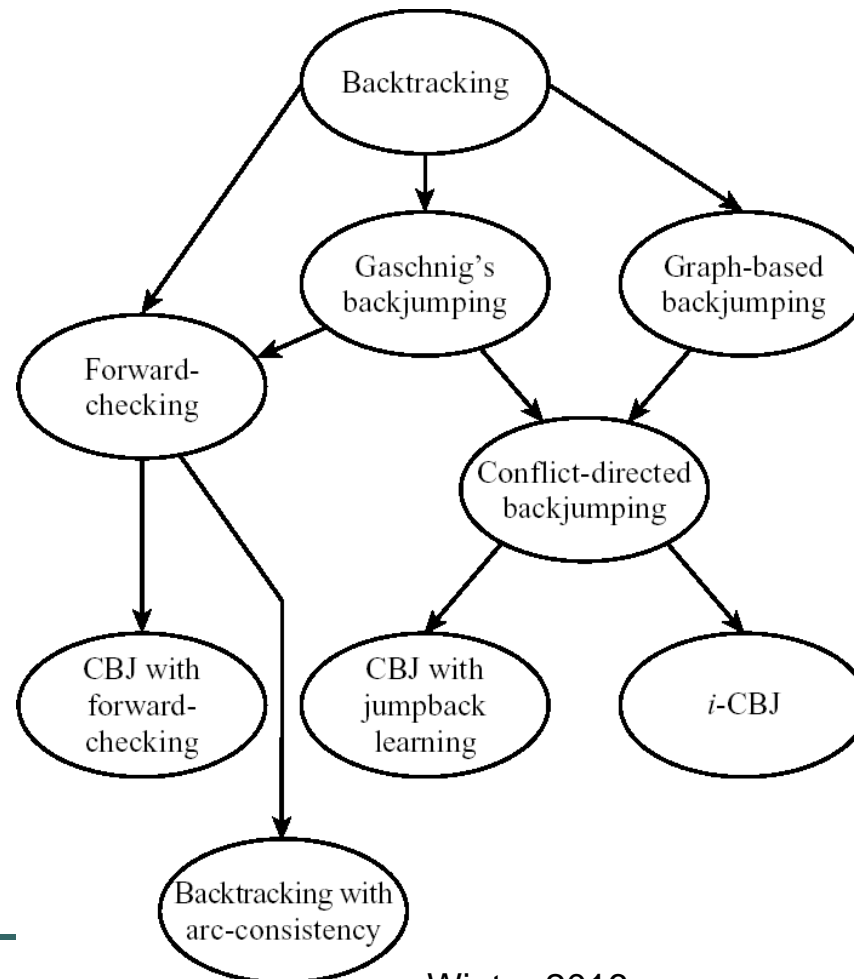
Empirical comparison of algorithms

- Benchmark instances
- Random problems
- Application-based random problems
- Generating fixed length random k-sat (n,m) uniformly at random
- Generating fixed length random CSPs
- (N,K,T,C) also arity, r.

The Phase transition (m/n)
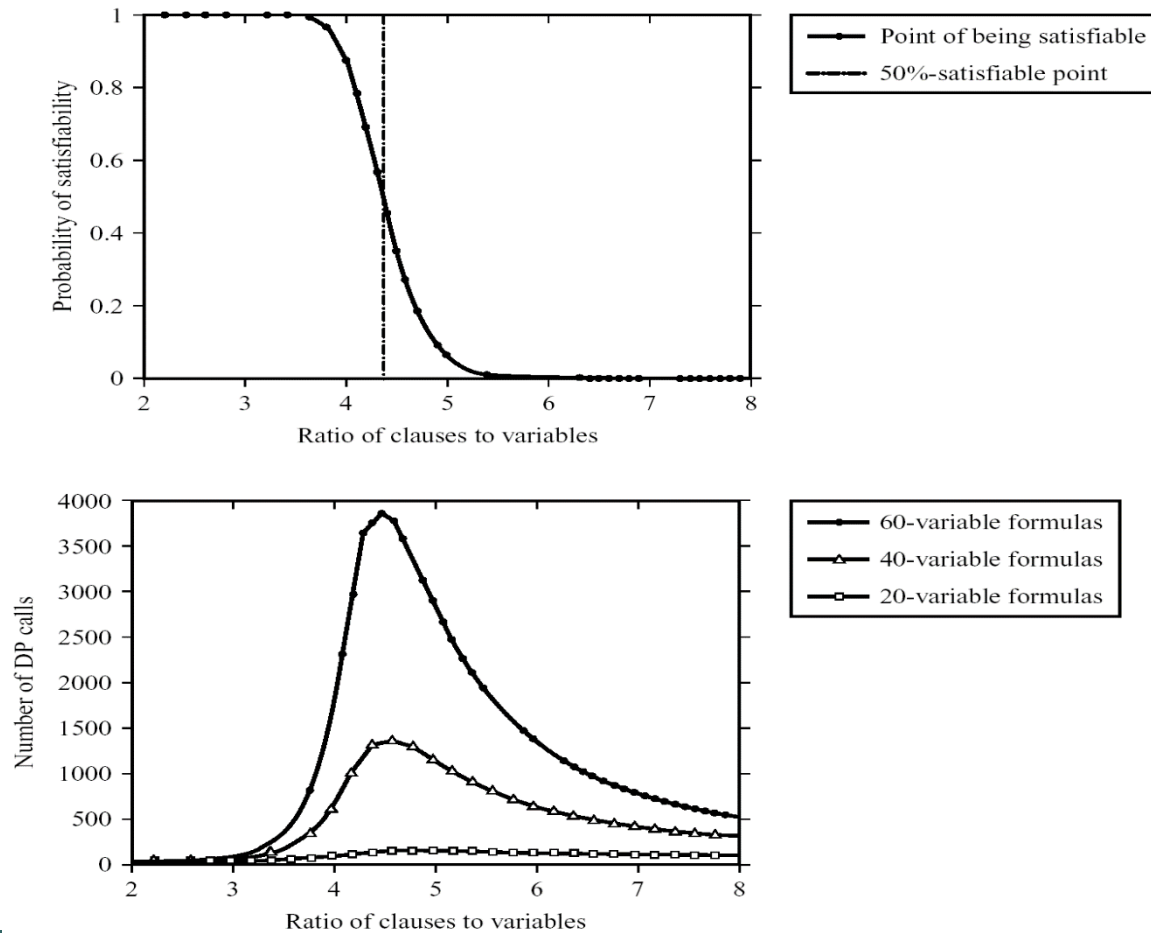
Some empirical evaluation

- Sets 1-3 reports average over 2000 instances of random csps from 50% hardness. Set 1: 200 variables, set 2: 300, Set 3: 350. All had 3 values.:

- Dimacs problems

Algorithm	Set 1		Set 2		Set 3		ssa 038		ssa 158	
FC	207	68.5		–		–	46	14.5	52	20.0
FC+AC	40	55.4	1	0.6	1	0.4	4	3.5	18	8.2
FCr-CBJ	189	69.2	222	119.3	182	140.8	40	12.2	26	10.7
FC-CBJ+LVO	167	73.8	132	86.8	119	111.8	32	11.0	8	4.5
FC-CBJ+LRN	186	63.4	32	15.6	1	0.5	23	5.5	19	8.6
FC-CBJ+LRN+LVO	160	74.0	26	14.0	1	3.8	16	3.8	13	7.1

Figure 6.16: Empirical comparison of six selected CSP algorithms. See text for explanation. In each column of numbers, the first number indicates the number of nodes in the search tree, rounded to the nearest thousand, and final 000 omitted; the second number is CPU seconds.

All Solutions and Counting

- For all solutions and counting we will see
  - The additional impact of Good learning
  - BFS makes sense with good learning
  - BFS and DFS time and space exp(path-width)
  - Good-learning doesn't help consistency task

#CSP – OR Search Tree

A	B	C	R_{ABC}
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

B	C	D	R_{BCD}
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

A	B	E	R_{ABE}
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

A	E	F	R_{AEF}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Winter 2016

#CSP – OR Search Tree

A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

B	C	D	R_{BCD}
0	0	0	**1**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**1**

A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

A	E	F	R_{AEF}
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

B	C	D	R_{BCD}
0	0	0	**1**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**1**

A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

A	E	F	R_{AEF}
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

14 solutions

#CSP - Tree DFS Traversal

A	B	C	R_{ABC}
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

B	C	D	R_{BCD}
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

A	B	E	R_{ABE}
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

A	E	F	R_{AEF}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Value of node = number of solutions below it

#CSP - OR Search Tree

A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

B	C	D	R_{BCD}
0	0	0	**1**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**1**

A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

A	E	F	R_{AEF}
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

14 solutions

#CSP - Searching the Graph by Good Caching

A	B	C	R_ABC
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

B	C	D	R_BCD
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

A	B	E	R_ABE
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

A	E	F	R_AEF
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

A   context(A) = [A]

B   context(B) = [AB]

C   context(C) = [ABC]

D   context(D) = [ABD]

E   context(E) = [AE]

F   context(F) = [F]

#CSP - Searching the Graph by Good Caching

A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

B	C	D	R_{BCD}
0	0	0	**1**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**1**

A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

A	E	F	R_{AEF}
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

A context(A) = [A]

B context(B) = [AB]

C context(C) = [ABC]

D context(D) = [ABD]

E context(E) = [AE]

F context(F) = [F]

#CSP - Searching the Graph by Good Caching
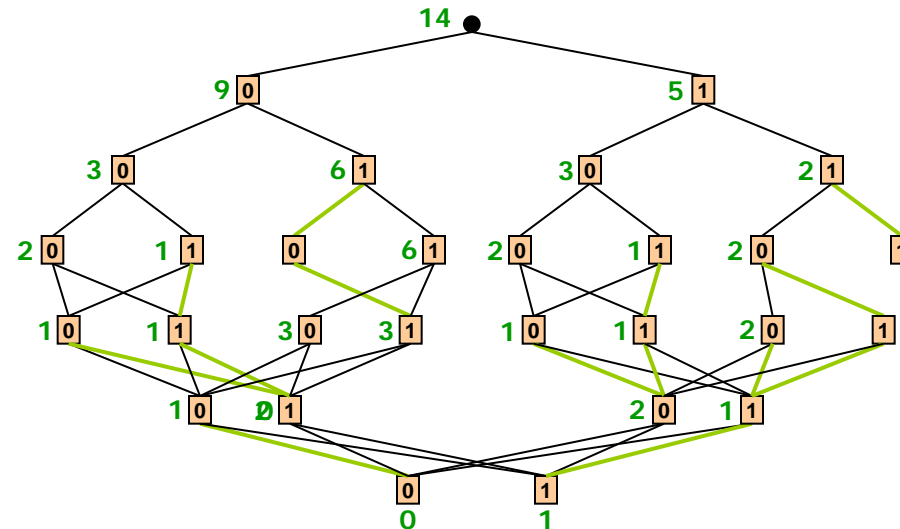
A	B	C	R_{ABC}
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

B	C	D	R_{BCD}
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

A	B	E	R_{ABE}
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

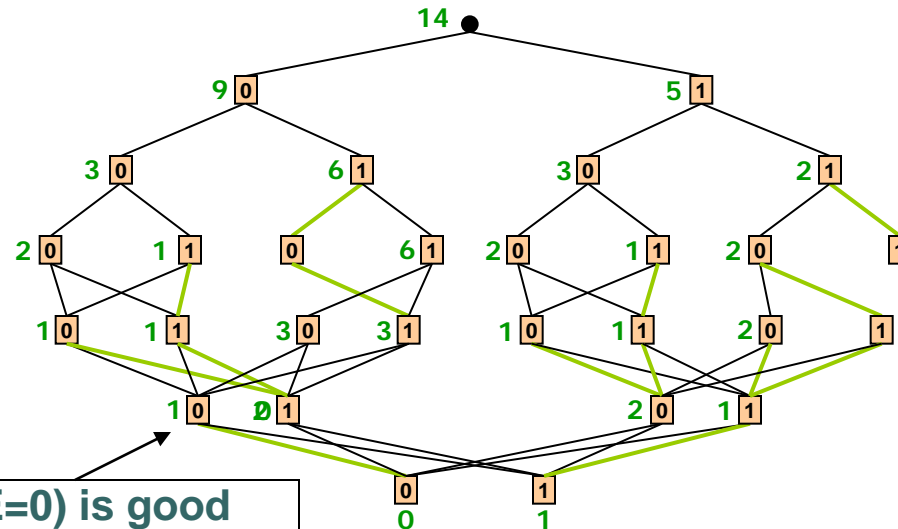| A	E	F	R_{AEF}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

A   context(A) = [A]

B   context(B) = [AB]

C   context(C) = [ABC]
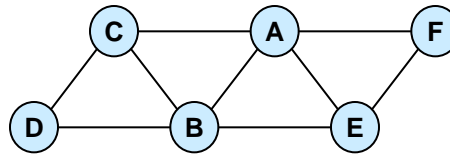
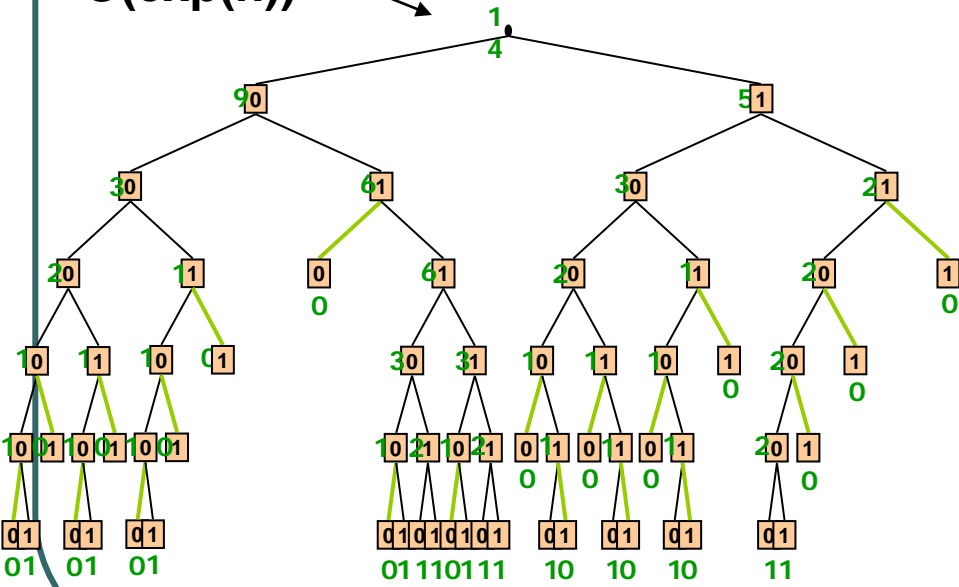D   context(D) = [ABD]

E   context(E) = [AE]

F   context(F) = [F]

(A=0,E=0) is good
V(A=0,E=0)=1
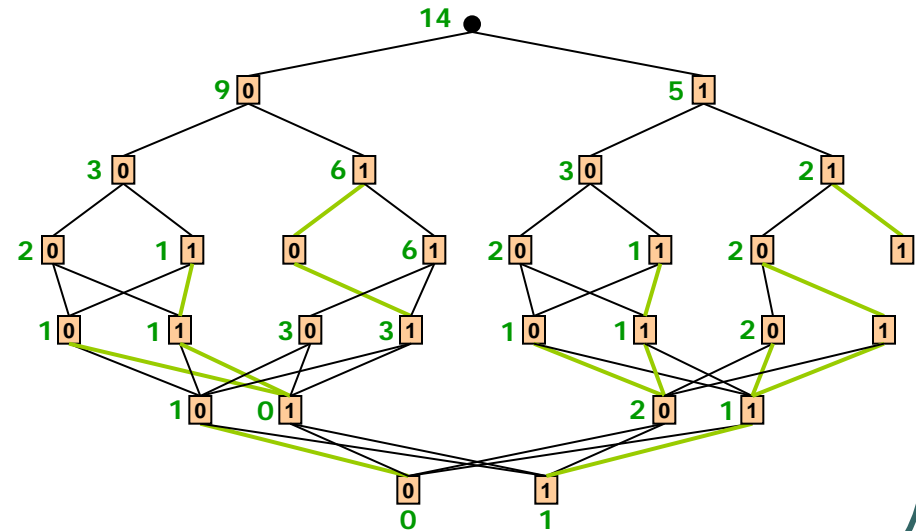
Winter 2016

#CSP - Searching the Graph by Good Caching

No caching:
O(exp(n))

Good-caching:
O(exp(pw))

Summary: Time-Space for Constraint Processing

- Constraint-satisfaction, one solution
  - **Naive backtracking**
    - **Space: O(n),**
    - **Time: O(exp(n))**
  - **Backjumping**
    - **Space: O(n),**
    - **Time: O(exp(log n w*))**
  - **Learning no-goods**
    - **Space: O(exp(w*))**
    - **Time: O(exp(w*))**
  - **Variable-elimination**
    - **Space: O(exp(w*))**
    - **Time: O(exp(w*))**

- Counting, enumeration
  - **Backtracking, backjumping**
    - **Space: O(n),**
    - **Time: O(exp(n))**
  - **Learning no-goods**
    - **space: O(exp(w*))**
    - **Time: O(exp(n))**
  - **Search with goods and no-goods learning**
    - **Space: O(exp(pw*))**
    - **Time: O(exp(pw*)), both, O(exp(w*logn))**
  - **Variable-elimination**
    - **Space: O(exp(w*))**
    - **Time: O(exp(w*))**
  - **BFS is time and space O(exp(pw*))**

Summary: search principles

- DFS is better than BFS search
- Constraint propagation (i.e., bounded inference) prunes search space
- Constraint propagation yields good advise for how to branch and where to go
- Backjumping and no-good learning helps prune search space and revise problem.
- Good learning revise problem but helps only counting, enumeration