Outline (from last lecture)

- Converting a CSP iThe search tree for CSPs, Variable ordering and consistency level
- Look-ahead for value selection:
    - Forward checking,
    - Full-arc-consistency,
    - partial look-ahead,
    - maintaining arc-consistency
- Dynamic Variable ordering (DVO, DVFC)
- Search for Satisfiability
- Converting a CSP into a SAT problem

Look-ahead for sat: DPLL

(Davis-Putnam, Logeman and Laveland, 1962)

DPLL(φ)
Input: A cnf theory φ
Output: A decision of whether φ is satisfiable.
1. Unit_propagate(φ);
2. If the empty clause is generated, return(*false*);
3. Else, if all variables are assigned, return(*true*);
4. Else
5. $\quad$ Q = some unassigned variable;
6. $\quad$ return(**DPLL**($\varphi \wedge Q$) $\vee$
$\quad\quad\quad$ **DPLL**($\varphi \wedge \neg Q$))

Figure 5.13: The DPLL Procedure

On Unit Resolution

To incorporate unit resolution into our satisfiability algorithms, we will introduce a function UNIT-RESOLUTION, which applies to a CNF Δ and returns two results:

- $\mathbf{I}$: a set of literals that were either present as unit clauses in Δ, or were derived from Δ by unit resolution.
- Γ: a new CNF which results from conditioning Δ on literals $\mathbf{I}$.

For example, if the CNF

$$\Delta = \{\, \{\neg A, \neg B\},\, \{B, C\},\, \{\neg C, D\},\, \{A\} \,\},$$

then $\mathbf{I} = \{A, \neg B, C, D\}$ and $\Gamma = \{\}$. Moreover, if

$$\Delta = \{\, \{\neg A, \neg B\},\, \{B, C\},\, \{\neg C, D\},\, \{C\} \,\},$$

then $\mathbf{I} = \{C, D\}$ and $\Gamma = \{\, \{\neg A, \neg B\} \,\}$. Unit resolution is a very important component of search-based SAT solving algorithms. Part 1, Chapter 4 discusses in details the modern implementation of unit resolution employed by many SAT solvers of this type.

Chronological Backtracking

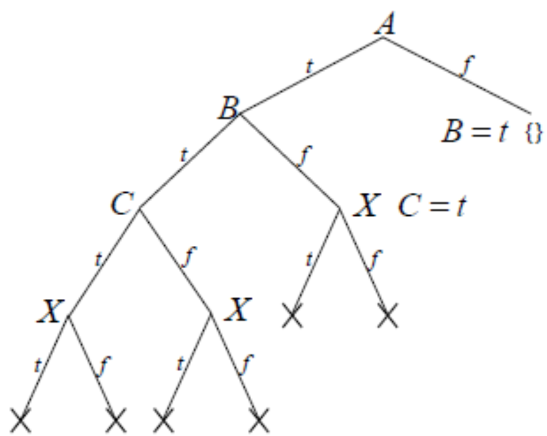


Figure 3.6. A termination tree. Assignments shown next to nodes are derived using unit resolution.

To consider a concrete example, let us look at how standard DPLL behaves on the following CNF, assuming a variable ordering of A, B, C, X, Y, Z:

$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array} \tag{3.1}$$

Reduction from CSP to SAT

Example: CSP into SAT

Notation: variable-value pair = **vvp**

- vvp $\rightarrow$ term
  - V_1 = {a, b, c, d} yields $x_1 = (V_1, a)$, $x_2 = (V_1, b)$, $x_3 = (V_1, c)$, $x_4 = (V_1, d)$,
  - V_2 = {a, b, c} yields $x_5 = (V_2, a)$, $x_6 = (V_2, b)$, $x_7 = (V_2, c)$.
- The vvp's of a variable $\rightarrow$ disjunction of terms
  - V_1 = {a, b, c, d} yields
- (How do we express: "At most one VVP per variable "

Reduction from CSP to SAT

Example: CSP into SAT

Notation: variable-value pair = **vvp**

- vvp $\rightarrow$ term
  - V_1 = {a, b, c, d} yields $x_1 = (V_1, a)$, $x_2 = (V_1, b)$, $x_3 = (V_1, c)$, $x_4 = (V_1, d)$,
  - V_2 = {a, b, c} yields $x_5 = (V_2, a)$, $x_6 = (V_2, b)$, $x_7 = (V_2, c)$.
- The vvp's of a variable $\rightarrow$ disjunction of terms
  - V_1 = {a, b, c, d} yields
- (How do we express: "At most one VVP per variable "

CSP into SAT (cont.)

Constraint:

1. Way 1: Each inconsistent tuple $\rightarrow$ one disjunctive clause
   - For example: how many?

2. Way 2:
   a) Consistent tuple $\rightarrow$ conjunction of terms
   b) Each constraint $\rightarrow$ disjunction of these conjunctions

$\rightarrow$ transform into conjunctive normal form (CNF)

Question: find a truth assignment of the Boolean variables such that the sentence is satisfied

Outline

- The search tree for CSPs, Variable ordering and consistency level
- Look-ahead for value selection:
  - Forward checking,
  - Full-arc-consistency,
  - partial look-ahead,
  - maintaining arc-consistency
- Dynamic Variable ordering (DVO, DVFC)
- Search for Satisfiability
- Converting a CSP into a SAT problem

CompSci 275, CONSTRAINT Networks

Rina Dechter, Fall 2022

General Search: Look-back scuemes
Chapter 6

Outline

- Look-back strategies
- Backjumping: Gaschnig, Graph-based, Conflict-directed
- Learning no-goods, constraint recording.
- Look-back for Satisfiability, integration and Empirical evaluation
- Counting, good caching

Look-back: Backjumping and Learning

- Backjumping:
  - In deadends, go back to the most recent culprit.
- Learning:
  - constraint-recording:
    - no-good recording.
    - good-recording

Backjumping

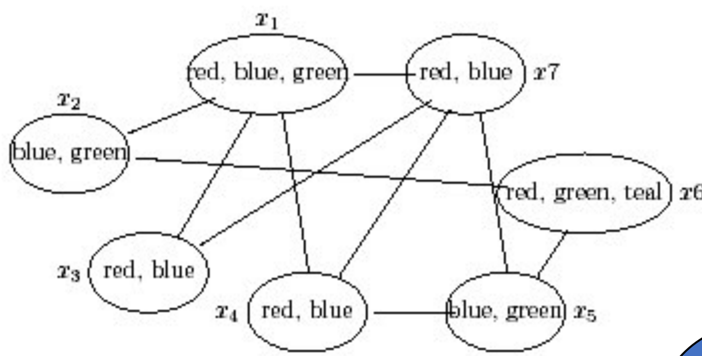


Figure 6.1: A modified coloring problem.

- (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,-) conflict-set of x7
- (r,-,b,-,-,-,-) **minimal conflict-set of x7**
- **Leaf deadend**: (r,b,b,b,g,r)
- Every conflict-set is a **no-good**

Backjumping

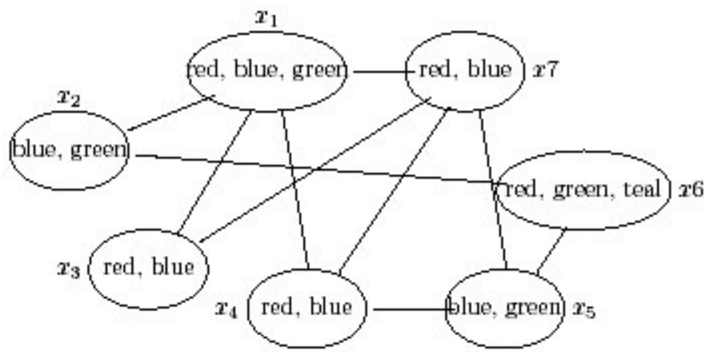


Figure 6.1: A modified coloring problem.

- (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,-) conflict-set of x7
- (r,-,b,-,-,-,-) **minimal conflict-set of x7**
- **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

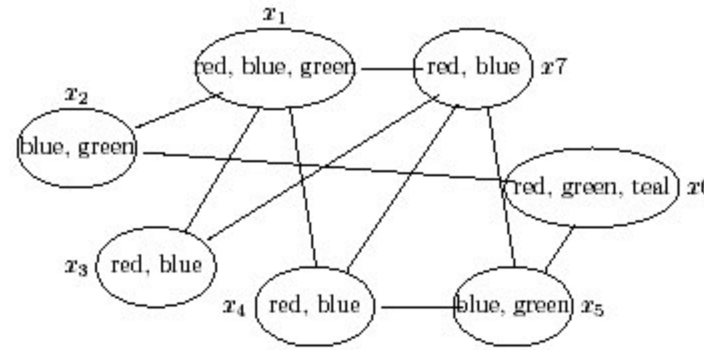
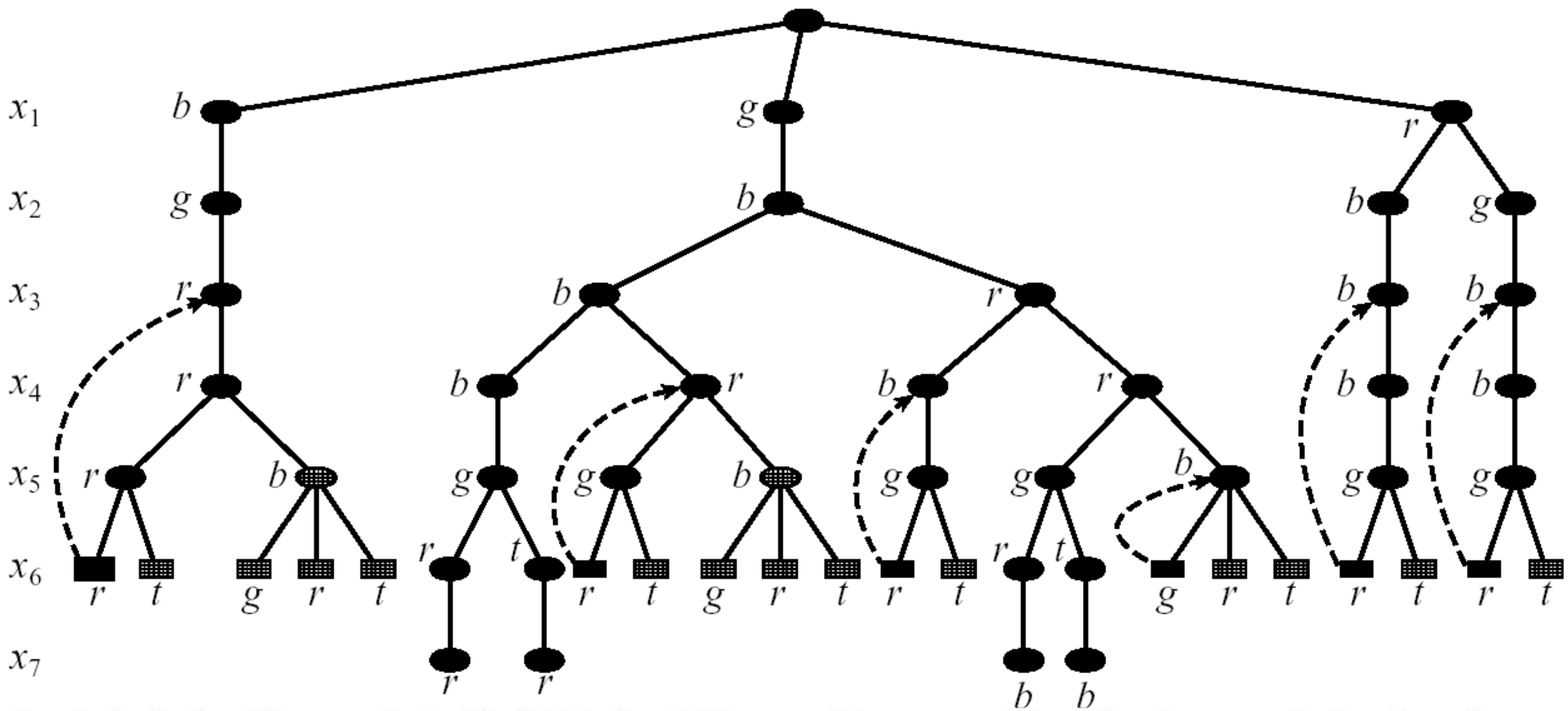


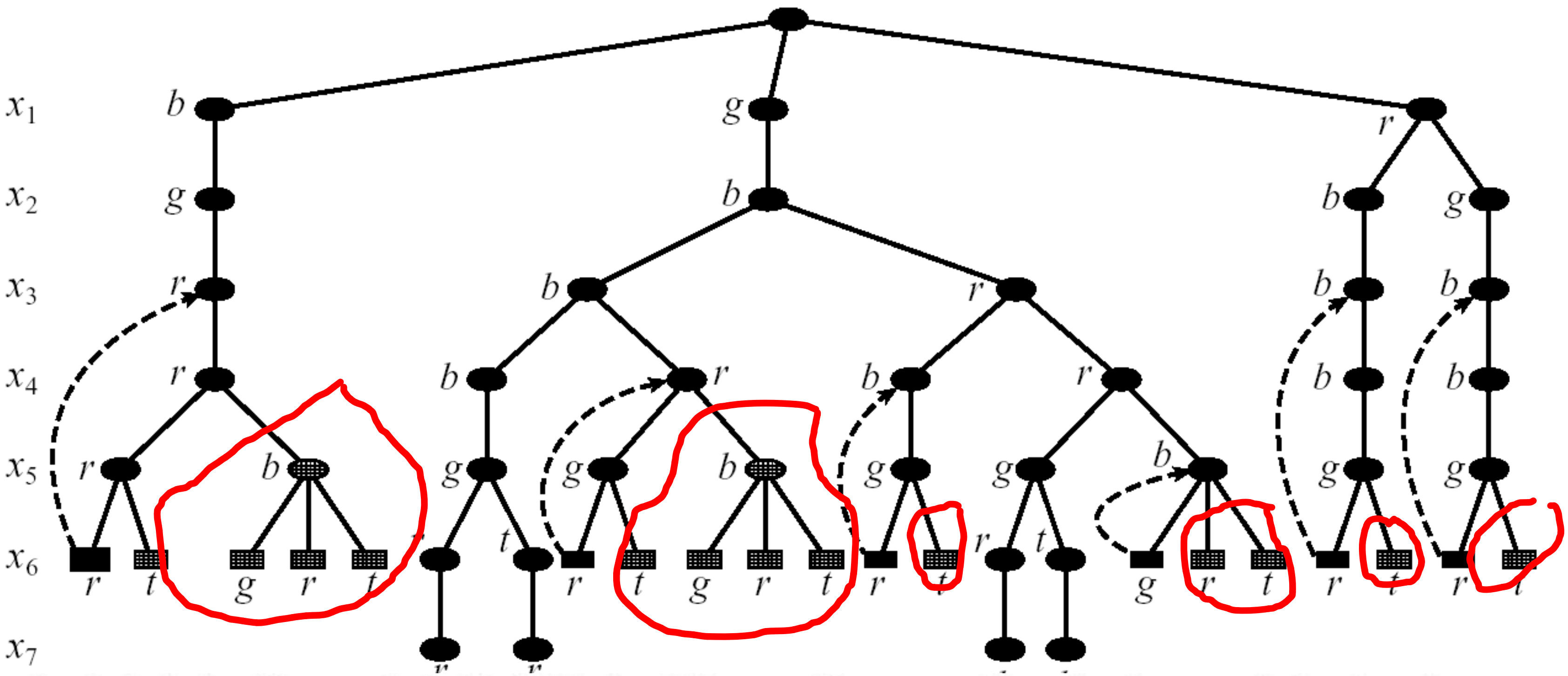
Figure 6.1: A modified coloring problem.



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. □

Flavor of Gaschnig's jumps only at leaf-dead-ends

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

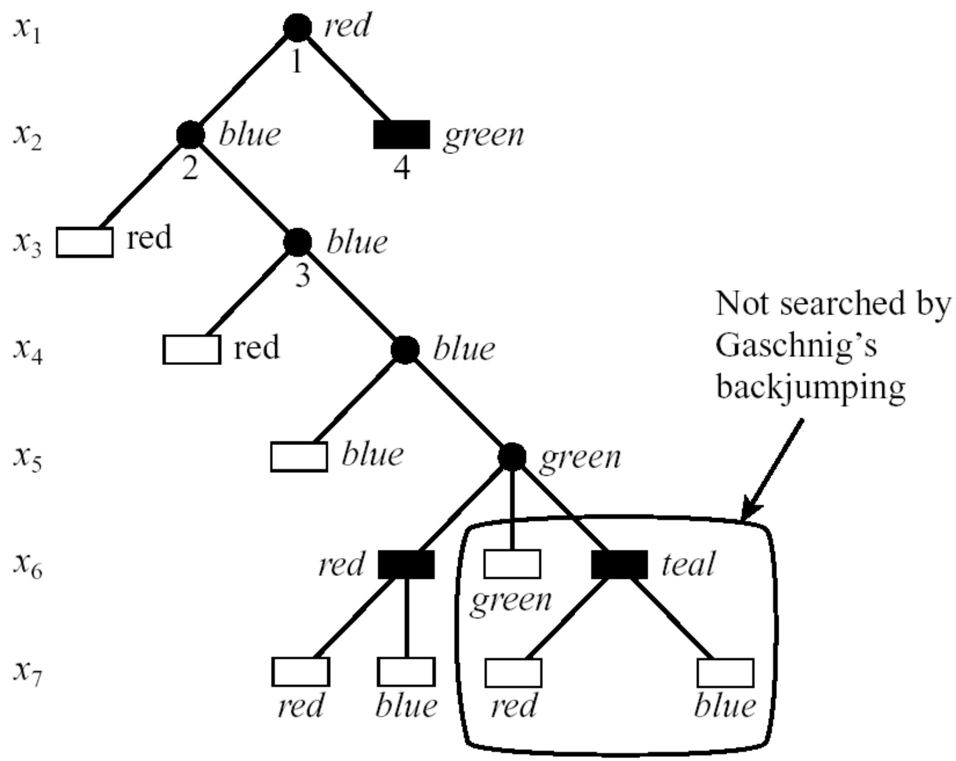


Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to $(\langle x_1, \textit{green}\rangle, \langle x_2, \textit{blue}\rangle, \langle x_3, \textit{red}\rangle, \langle x_4, \textit{blue}\rangle)$, because this is the only case where another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

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



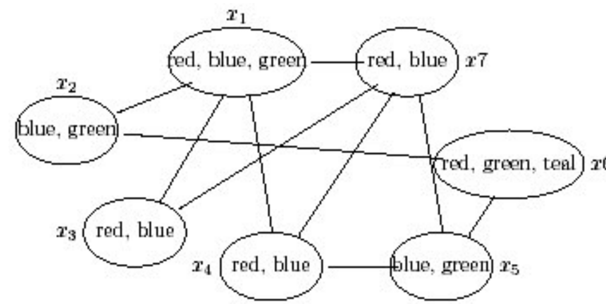


Figure 6.1: A modified coloring problem.

- Conflict set
- Leaf deadend
- Nogood
- Safe jump

Conflict-set analysis

Definition 6.1.1 (conflict set) *Let $\bar{a} = (a_{i_1}, ..., a_{i_k})$ 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.*

Definition 6.1.2 (leaf dead-end) *Let $\vec{a}_i = (a_1, ..., a_i)$ be a consistent tuple. If $\vec{a}_i$ is in conflict with x_{i+1}, it is called a* leaf dead-end.

Definition 6.1.3 (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.*

Definition 6.1.5 (safe jump) *Let $\vec{a_i} = (a_1, ..., a_i)$ be a leaf dead-end state. We say that x_j, where $j \leq i$, is* safe *if the partial instantiation $\vec{a_j} = (a_1, ..., a_j)$ is a no-good, namely, it cannot be extended to a solution.*

Gaschnig's backjumping:
Culprit variable

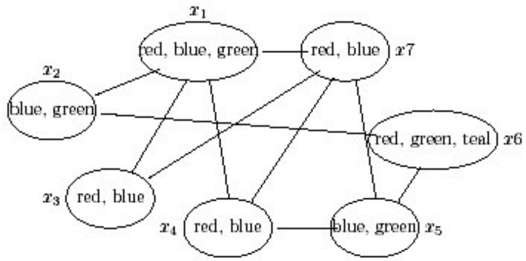
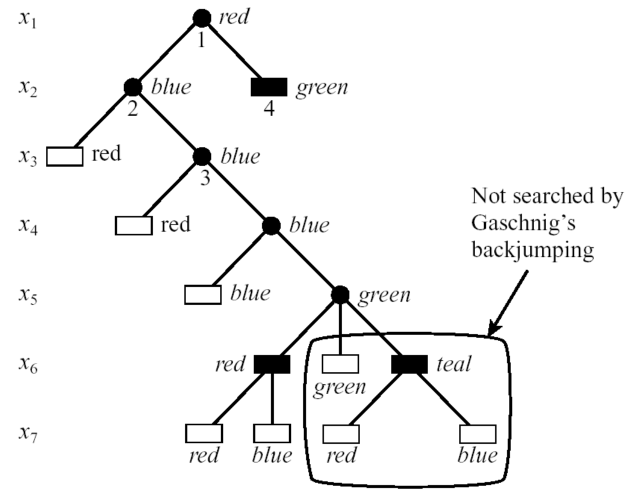


Figure 6.1: A modified coloring problem.



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 $b = \min\{j \leq i | \ \vec{a_j}$ 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.

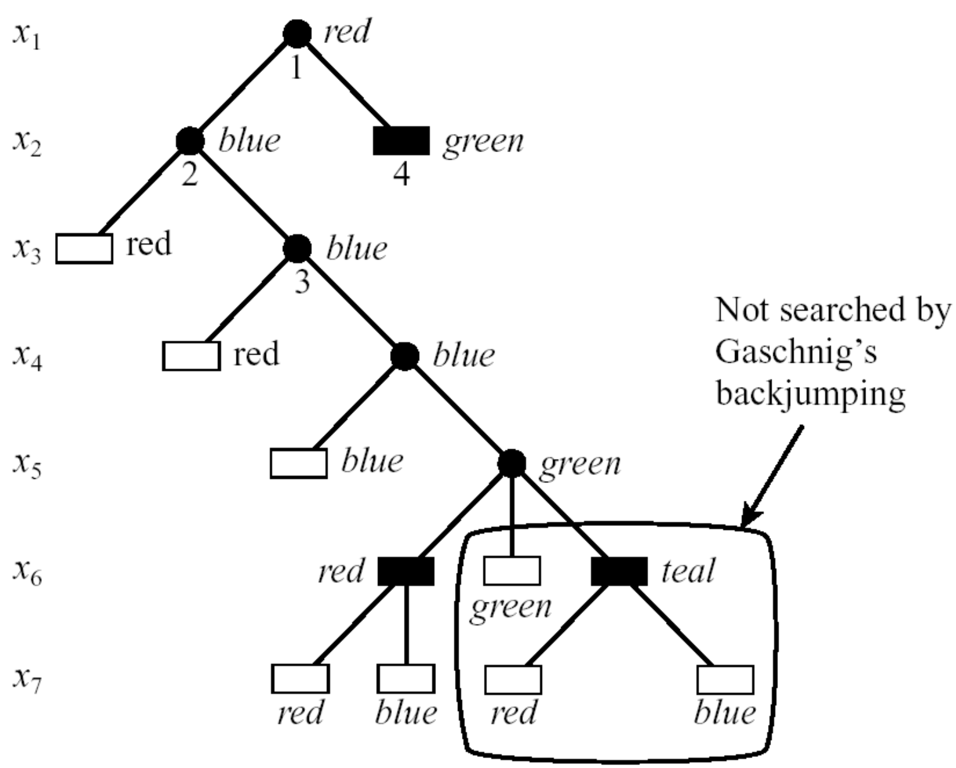
Gaschnig's backjumping

```
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
```

Figure 6.3: Gaschnig's backjumping algorithm.

Example of Gaschnig's backjump



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. □

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

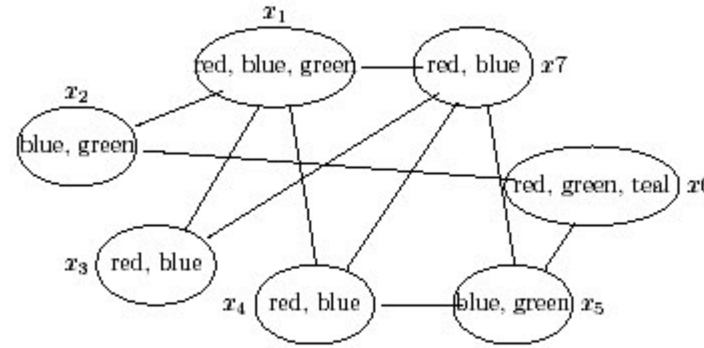
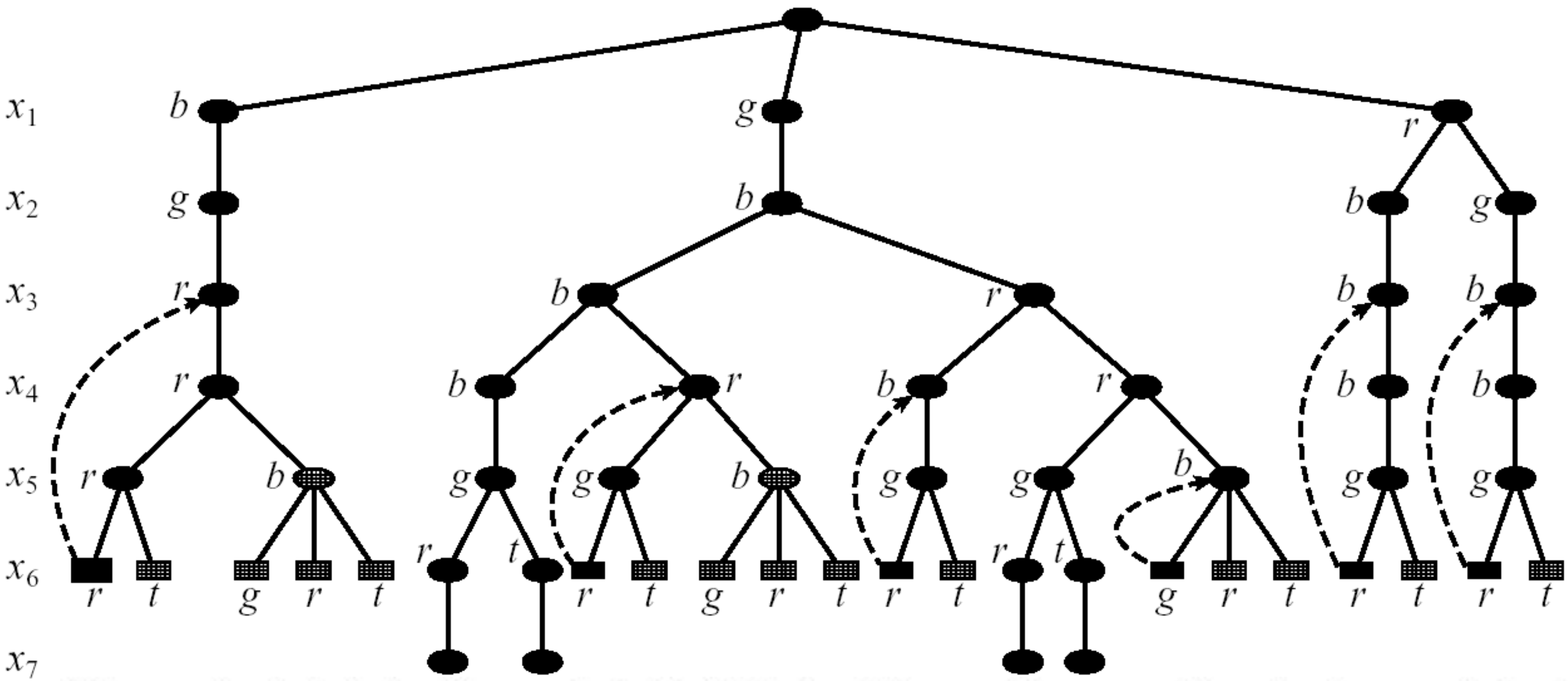


Figure 6.1: A modified coloring problem.



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. □

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

Graph-based backjumping scenarios
Internal deadend at X4

- Scenario 1, deadend at x4:
- Scenario 2: deadend at x5:
- Scenario 3: deadend at x7:
- Scenario 4: deadend at x6:

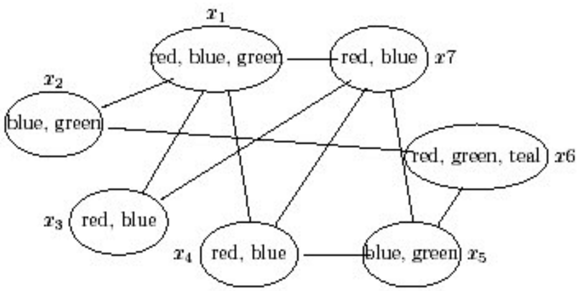
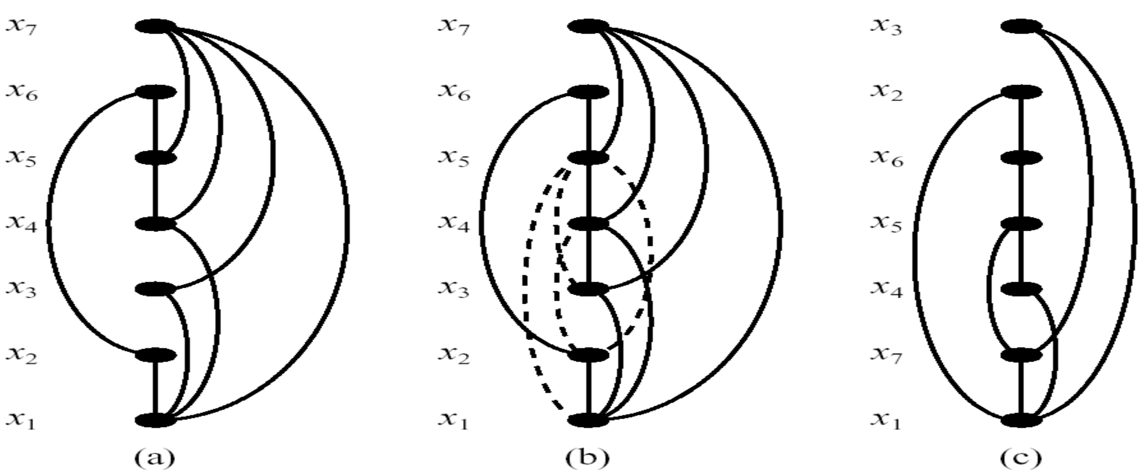


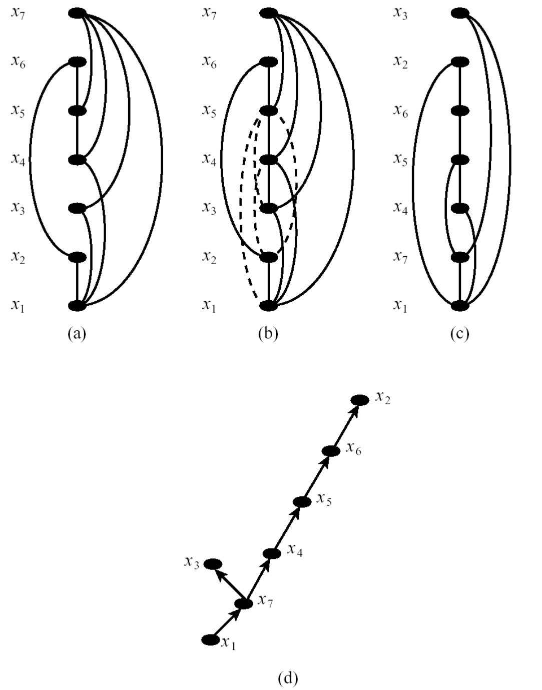
Figure 6.1: A modified coloring problem.



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



- 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* $\mathbf{d}$*, 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* $\vec{a_i} = (a_1, ..., a_i)$ *is a leaf dead-end, we equate* $anc(\vec{a_i})$ *with* $anc(x_{i+1})$*, 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 parent set in the induced-ordered graph

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

What are the relevant deadends?

What is the induced-parent set.

- Scenario 1, deadend at x4:
- Scenario 2: deadend at x5:
- Scenario 3: deadend at x7:
- Scenario 4: deadend at x6:

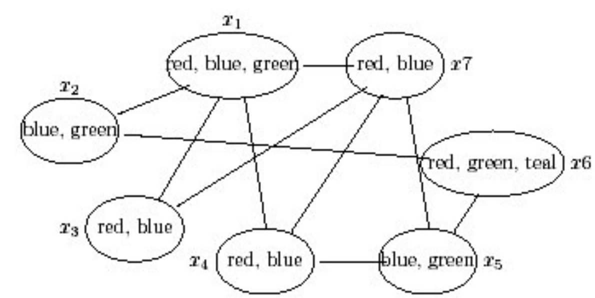
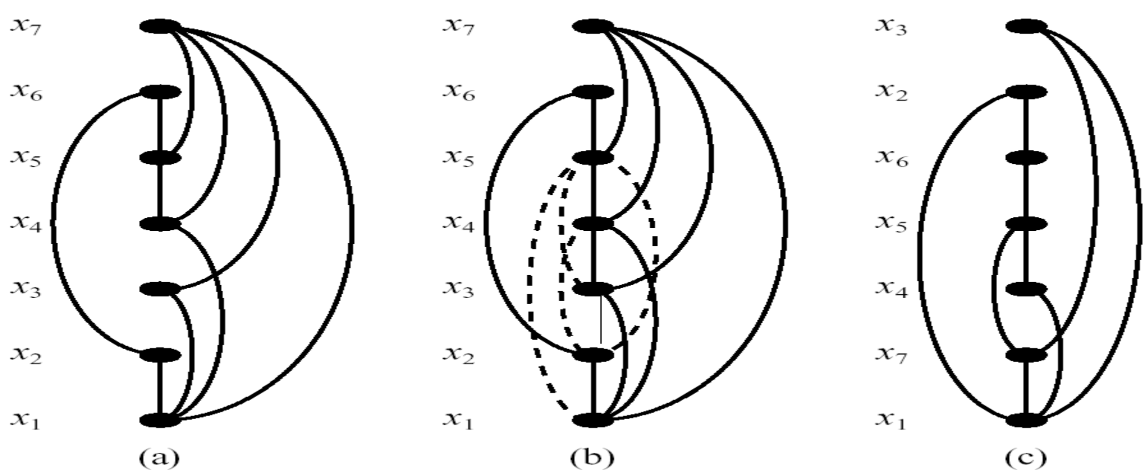
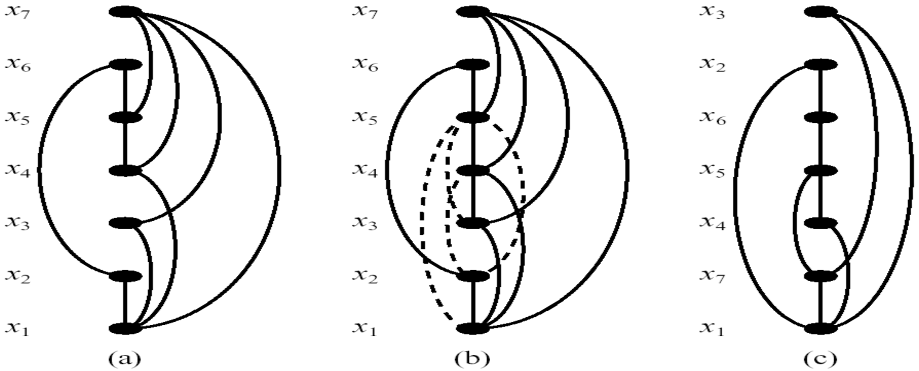


Figure 6.1: A modified coloring problem.



Graph-based backjumping scenarios
Internal deadend at X4

Example 6.3.9 Consider again the ordered graph in Figure 6.6a, and let x_4 be a dead-end variable. If x_4 is a leaf dead-end, then $Y = \{x_4\}$, and x_1 is the sole member in its induced ancestor set $I_4(Y)$. The algorithm may jump safely to x_1. If x_4 is an internal dead-end with $Y = \{x_4, x_5, x_6\}$, the induced ancestor set of x_4 is $I_4(\{x_4, x_5, x_6\}) = \{x_1, x_2\}$, and the algorithm can safely jump to x_2. However, if $Y = \{x_4, x_5, x_7\}$, the corresponding induced parent set $I_4(\{x_4, x_5, x_7\}) = \{x_1, x_3\}$, and upon encountering a dead-end at x_4, the algorithm should retract to x_3. If x_3 is also an internal dead-end the algorithm retracts to x_1 since $I_3(\{x_3, x_4, x_5, x_7\}) = \{x_1\}$. If, however, $Y = \{x_4, x_5, x_6, x_7\}$, when a dead-end at x_4 is encountered (we could have a dead-end at x_7, jump back to x_5, go forward and jump back again at x_6, and yet again at x_5), then $I_4(\{x_4, x_5, x_6, x_7\}) = \{x_1, x_2, x_3\}$. The algorithm then retracts to x_3, and if it is a dead-end it will retract further to x_2, since $I_3(\{x_3, x_4, x_5, x_6, x_7\}) = \{x_1, x_2\}$. □



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(a⃗_{i−1}, x_i = a)
         return a
   end while
   return null                 (no consistent value)
end procedure
```

Figure 6.5: The graph-based backjumping algorithm.

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

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 (**emc(x_{i+1}) **)** of a deadend.
  - Define the **jumpback set** of a deadend

Example of conflict-directed backjumping

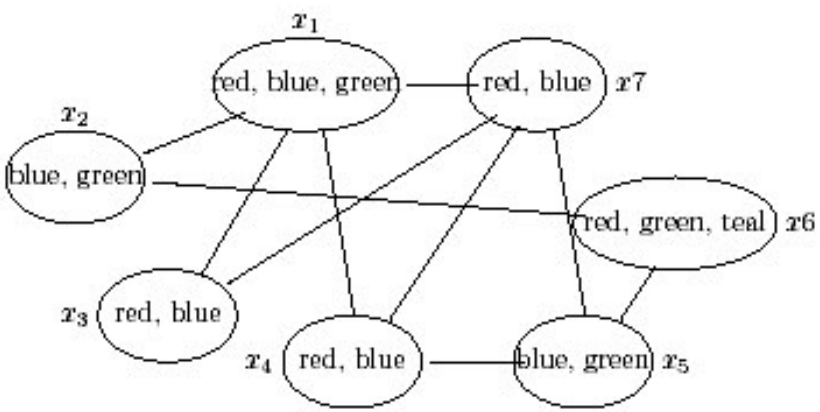


Figure 6.1: A modified coloring problem.

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. □

Properties of conflic-directed backjumping

- Given a dead-end , , the latest variable in its jumpback set 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-end 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 ← 1                            (initialize variable counter)
    D'_i ← D_i                       (copy domain)
    J_i ← ∅                          (initialize conflict set)
    while 1 ≤ i ≤ n
       instantiate x_i ← SELECTVALUE-CBJ
       if x_i is null                (no value was returned)
          iprev ← i
          i ← index of last variable in J_i    (backjump)
          J_i ← J_i ∪ J_iprev − {x_i}(merge conflict sets)
       else
          i ← i + 1                  (step forward)
          D'_i ← D_i                 (reset mutable domain)
          J_i ← ∅                    (reset conflict set)
    end while
    if i = 0
       return "inconsistent"
    else
       return instantiated values of {x_1, . . . , x_n}
end procedure

subprocedure SELECTVALUE-CBJ

    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 CONSISTENT(a⃗_k, x_i = a)
             k ← 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 ← false
       end while
       if consistent
          return a
    end while
    return null                      (no consistent value)
end procedure
```

Figure 6.7: The conflict-directed backjumping algorithm.

Graph-Based backjumping on dFS orderings

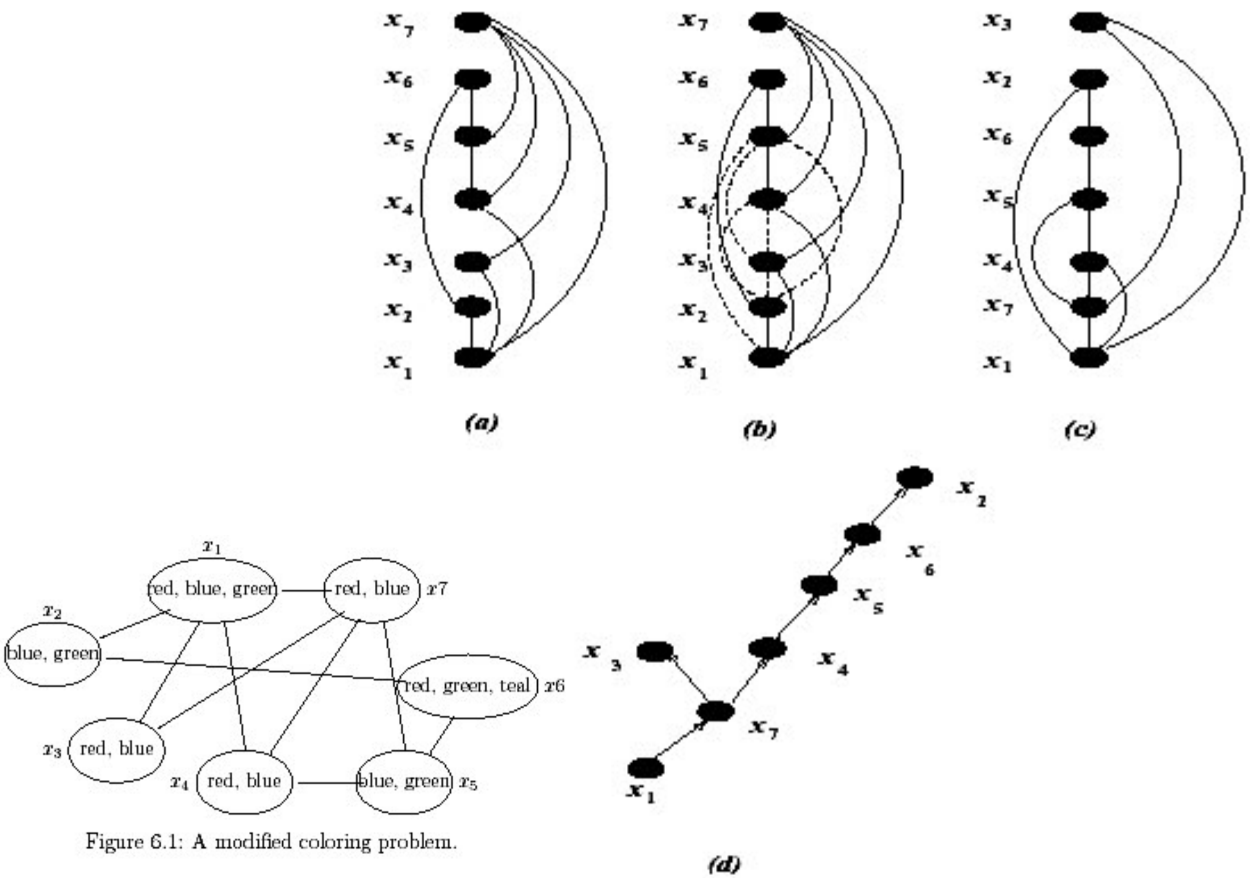
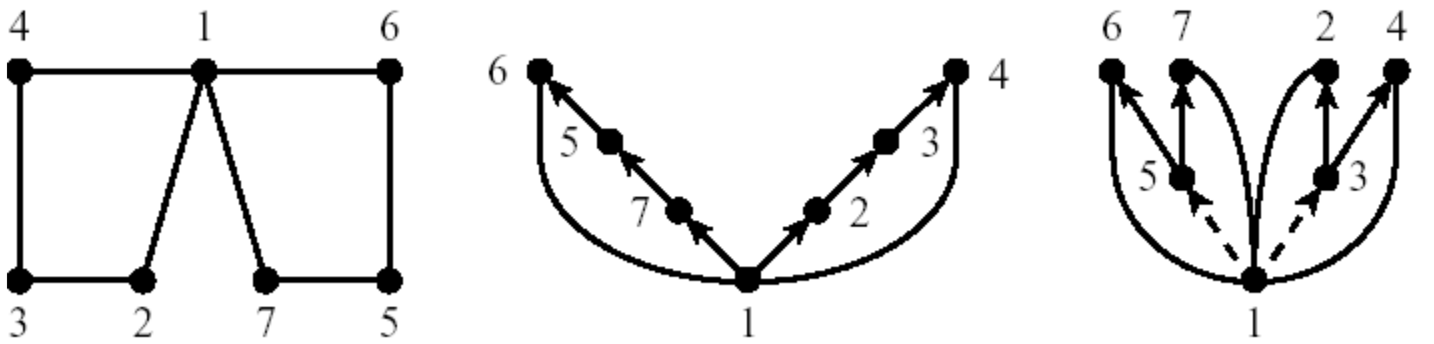


Figure 6.1: A modified coloring problem.

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.

Graph-based backjumping on DFS ordering

Rule: Go back to parent. No need to maintain a parent set



(a) (b) (c)

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.*

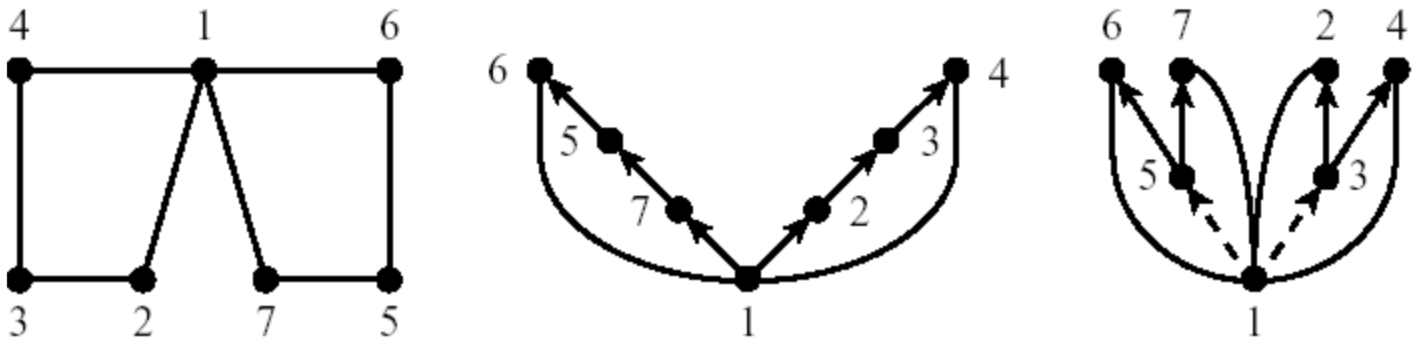
Spanning-tree of a graph;
DFS spanning trees, Pseudo-tree
Pseudo-tree is a spanning tree that does not allow arcs across branches.

Complexity of 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:

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.*

Complexity of backjumping uses pseudo-tree analysis



(a) (b) (c)

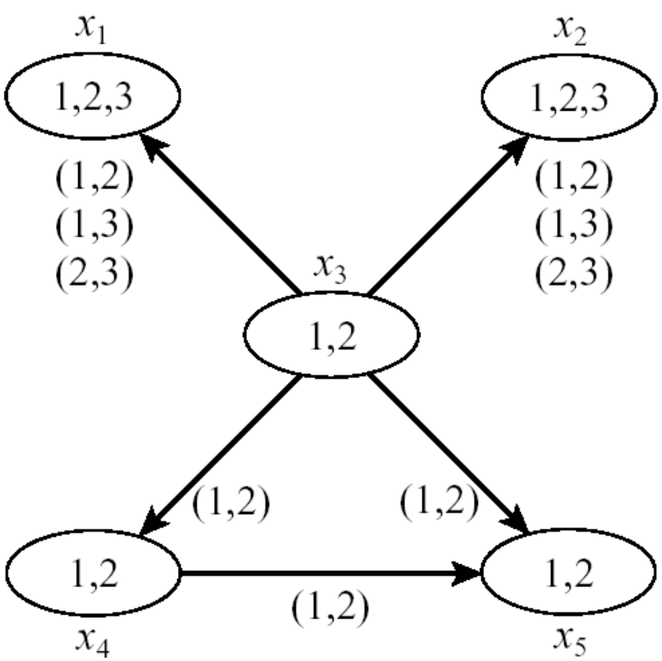
Simple: always jump back to parent in pseudo tree
Complexity for csp: exp(tree-depth)
Complexity for csp: exp(w*log n)

Outline

- Look-back strategies
- Backjumping: Gaschnig, Graph-based, Conflict-directed
- Learning no-goods, constraint recording.
  - Shallow and deep learning, graph-based learning
- Look-back for Satisfiability, integration and Empirical evaluation
- Counting, good caching

Look-back: No-good Learning, Constraint recording

Learning means recording conflict sets used as constraints to prune future search space.



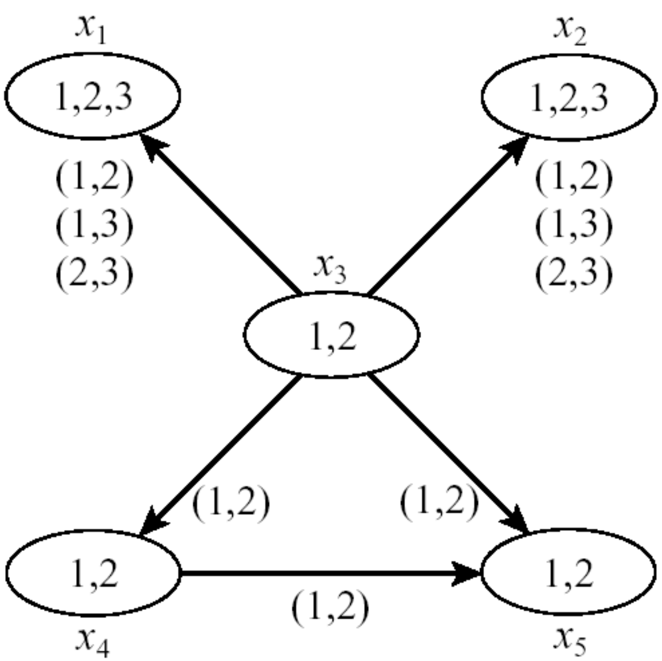
- (x_1=2, x_2=2, x_3=1, x_4=2) is a dead-end
- Conflicts to record:
  - (x_1=2, x_2=2, x_3=1, x_4=2) 4-ary
  - (x_3=1, x_4=2) binary
  - (x_4=2) unary

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.
These 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

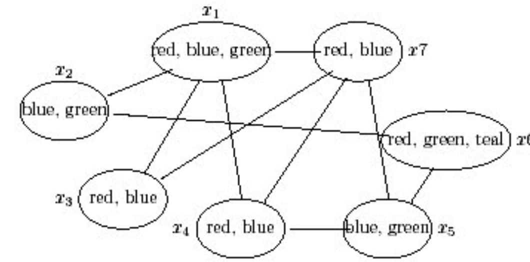
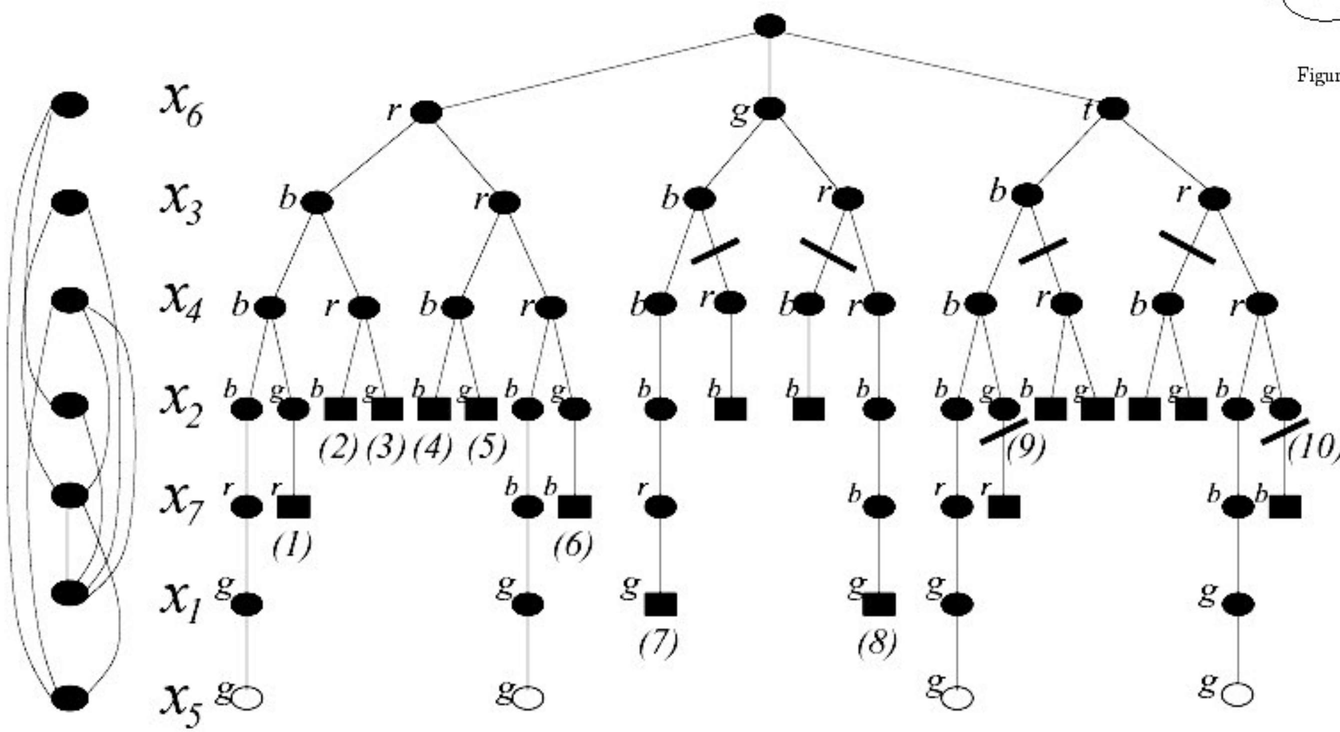


Figure 6.1: A modified coloring problem.



(a) (b)

X3, X4

r r

~~r b~~

~~b r~~

b b

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 ← SELECTVALUE
    if x_i is null                    (no value was returned)
        record a constraint prohibiting a⃗_{i−1}[I_i].
        iprev ← 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:

Deep learning pioneer

Learning example

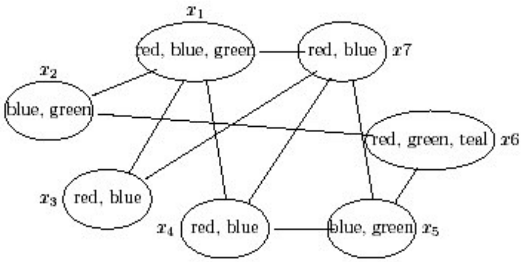
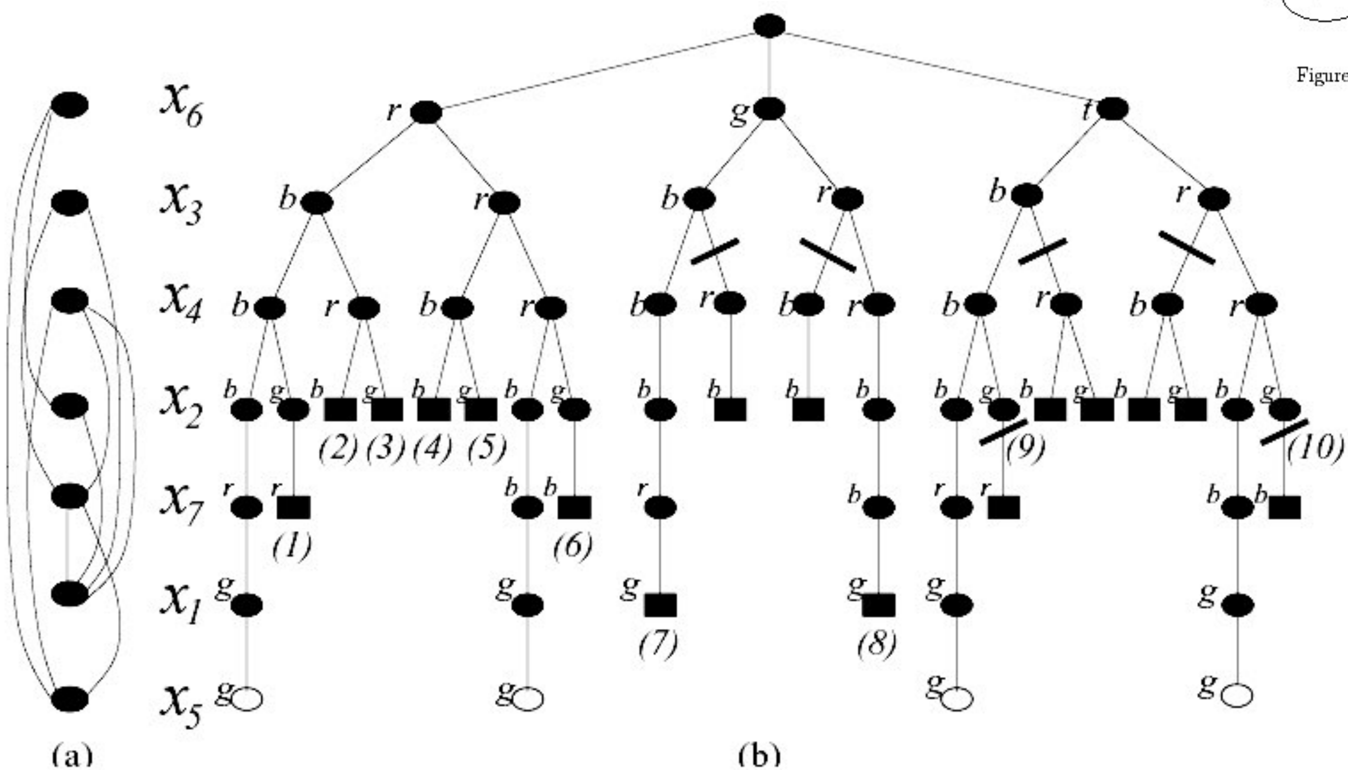


Figure 6.1: A modified coloring problem.



X3, X4

r	r
~~r~~	~~b~~
~~b~~	~~r~~
b	b

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.

Jumpback learning

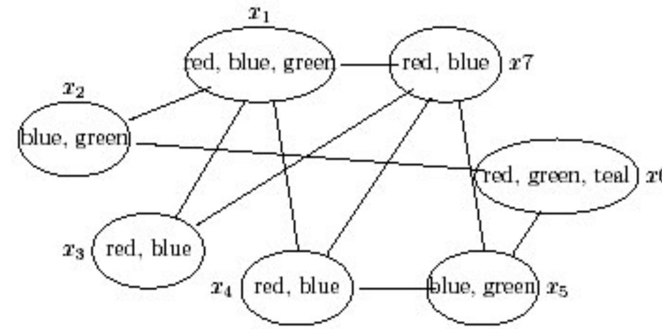


Figure 6.1: A modified coloring problem.

- 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. □

Jumpback set = {x3,x2,x7}

X3,x2,x7		
r	b	r
r	b	b
r	g	r
r	g	b
b	b	r
b	b	b
b	g	b
b	g	r

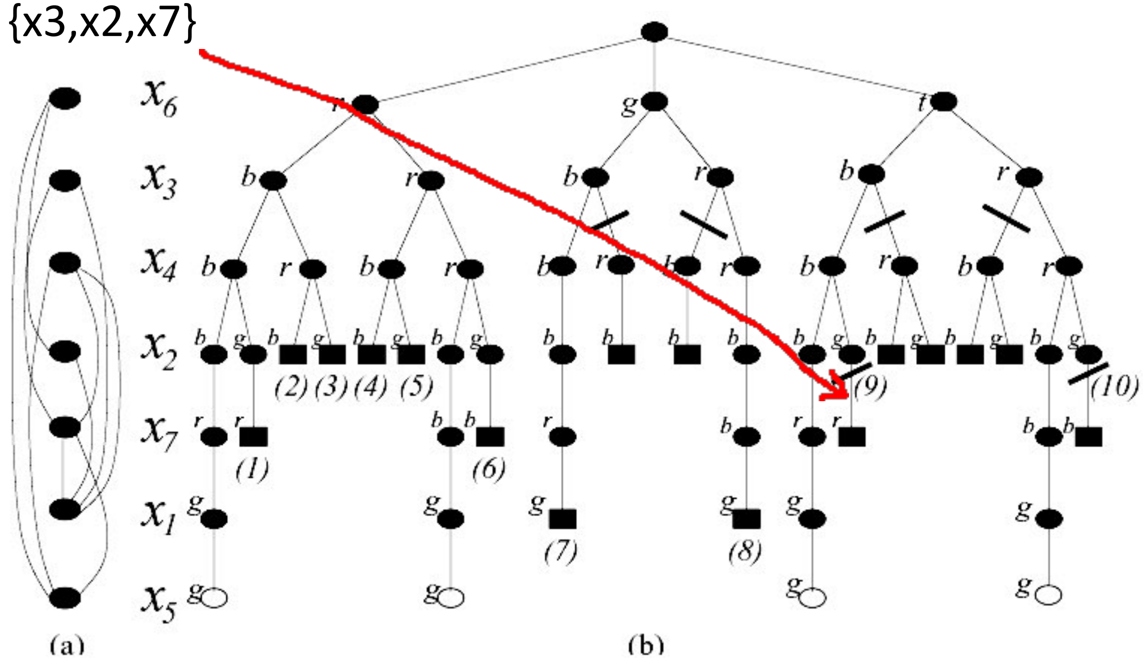


Figure 6.9: The search space explicated by backtracking on the CSP from Figure 6.1.

Jumpback learning

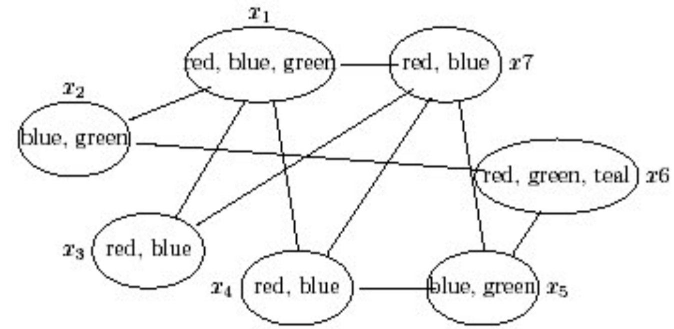


Figure 6.1: A modified coloring problem.

- 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

Number of constraint tests per dead-end are

Space complexity is

Time complexity is

n- depth of tree, e- number of constraints

Non-Systematic randomized learning

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

- Guaranteed completeness.

Outline

- Look-back strategies
- Backjumping: Gaschnig, Graph-based, Conflict-directed
- Learning no-goods, constraint recording.
- Look-back for Satisfiability, integration and Empirical evaluation
- Counting, good caching

Look-back for SAT

- A partial assignment is a set of literals:
- A jumpback set if a J-clause:
- Upon a leaf deadend of *x* resolve two clauses, one enforcing x and one enforcing ¬x relative to the current assignment
- A clause forces x relative to assignment if all the literals in the clause are negated in .
- 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.

phi = {A,B,X}, {~C,~X}

{A,B,~C}

Assignment= (~A,~B,C,F,R → X)

Look-back for SAT

procedure SAT-CBJ-LEARN
Input: A CNF theory φ, assigned variables σ over $x_1, ..., x_{i-1}$, unassigned variables X,
Output: Either a solution, or a decision that the network is inconsistent.

1. $J_i \leftarrow \emptyset$
2. While $1 \leq i \leq n$
3. Select the next variable: $x_i \in X$, $X \leftarrow X - \{x_i\}$
4. instantiate $x_i \leftarrow$ SELECTVALUE-CBJ.
5. If x_i is null (no value returned), then
6. add J_{x_i} to φ (learning)
7. $iprev \leftarrow$ index of last variable in J_i (*backjump*)
8. $J_i \leftarrow resolve(J_i, J_{prev})$ (merge conflict sets)
9. else,
10 $i \leftarrow i + 1$ (go forward)
11. $J_i \leftarrow \emptyset$ (reset conflict set)
12. Endwhile
13. if i = 0 Return "inconsistent"
14. else, return the set of literals σ

end procedure

subprocedure SELECTVALUE-CBJ
1. If CONSISTENT($\sigma \cup x_i$) then return $\sigma \leftarrow \sigma \cup \{x_i\}$
2. If CONSISTENT($\sigma \cup \neg x_i$) then return $\sigma \leftarrow \sigma \cup \{\neg x_i\}$
3. else,
4. determine α and β the two earliest clauses forcing x_i and $\neg x_i$,
5. $J_i \leftarrow resolve(\alpha, \beta)$.
5. Return $x_i \leftarrow$ null (no consistent value)

end procedure

Figure 6.12: Algorithm SAT-CBJ-LEARN

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

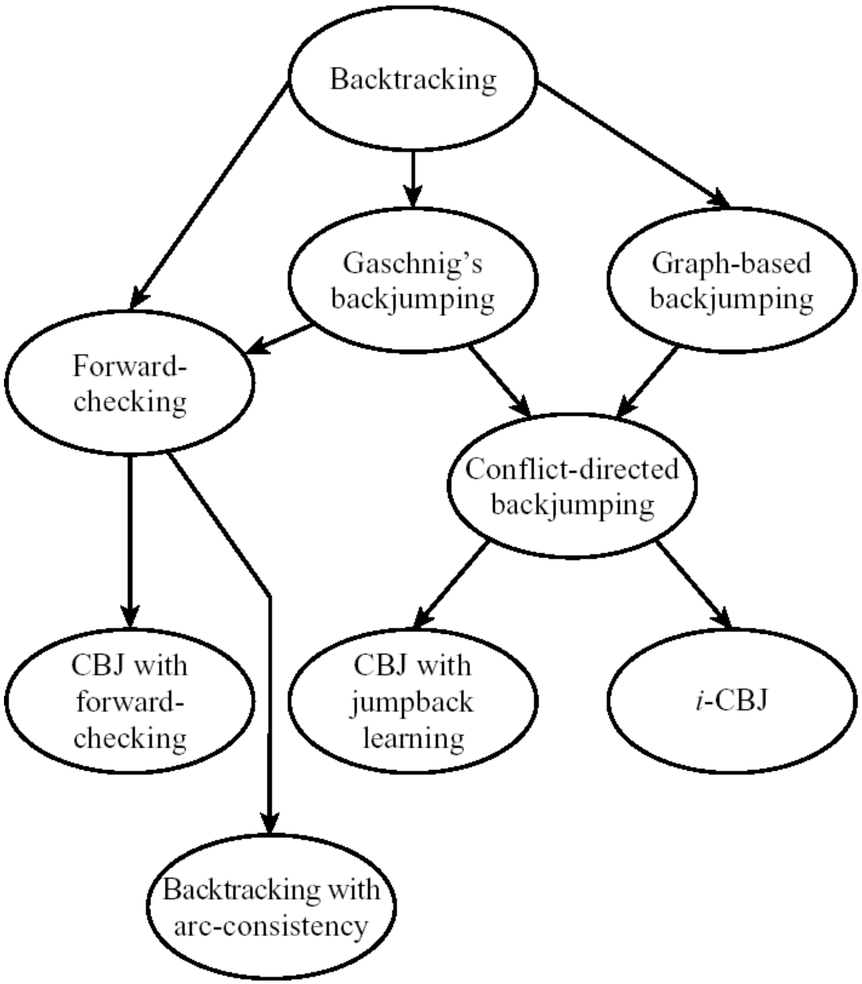
Figure 6.13: The main procedure of the FC-CBJ algorithm

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 algorithms

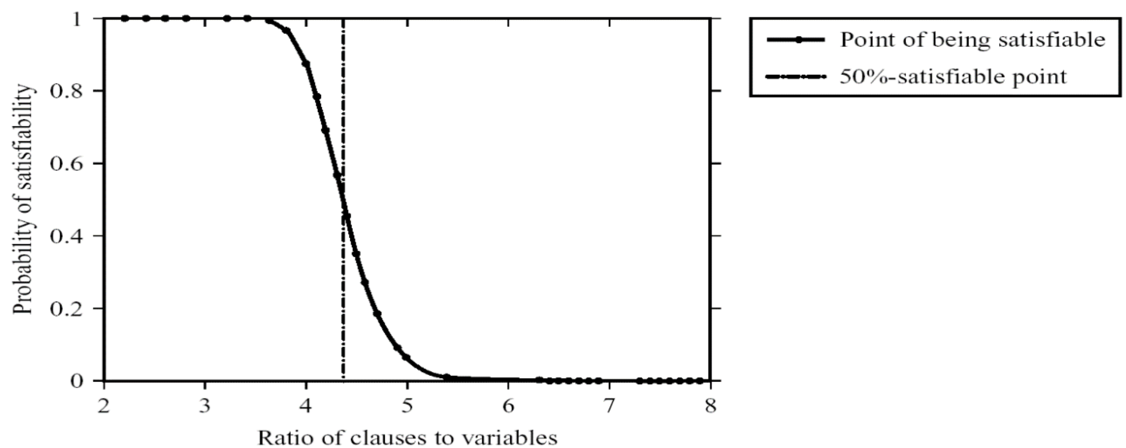


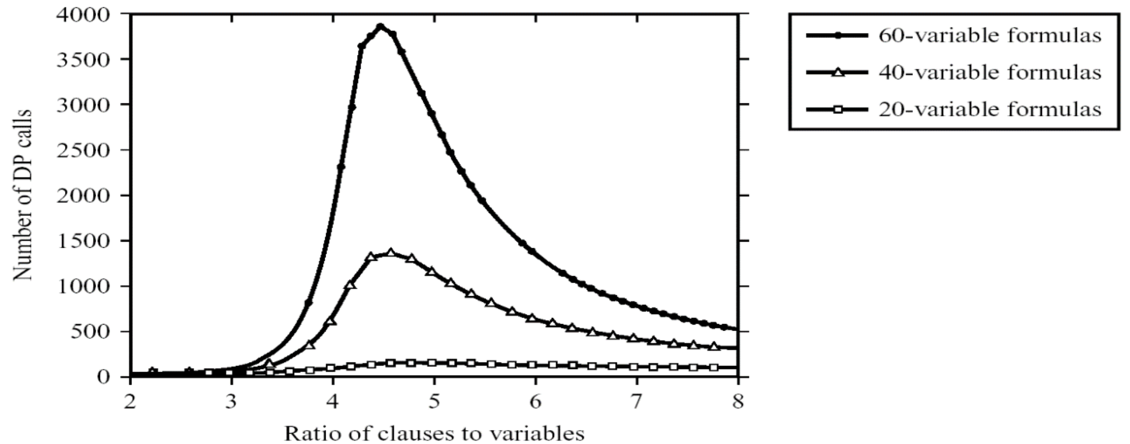
Empirical comparison of algorithms

- 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)

- Fixed length formulas are generated by selecting a fixed number m of clauses uniformly at random of a given length k.
- Small number of clauses yield easy solvable instances. Large number of clauses yield easy unsolvable instances.
- Peak hardness dependant on m/n. for 3-sat, m/n = 4.2
- Random CSPs are generated via (N,k,C,T)=(number of variables, domains, number of binary constraints, T tightness





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. Entries: average number of nodes, average time in sec
- Dimacs problems

Algorithm	Set 1		Set 2		Set 3		ssa 038		ssa 158	
FC	207	[illegible]		-		-	46	14.5	52	20.0
[illegible]	[illegible]	[illegible]	1	[illegible]	1	[illegible]	4	[illegible]	[illegible]	[illegible]
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	2.8	16	[illegible]	[illegible]	[illegible]

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.

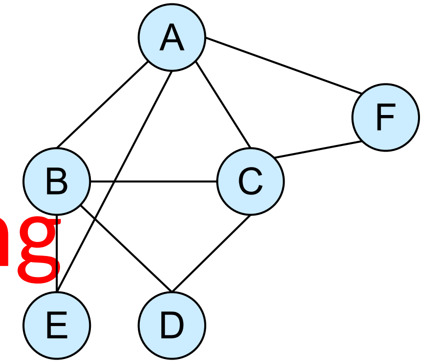
Results Interpretation

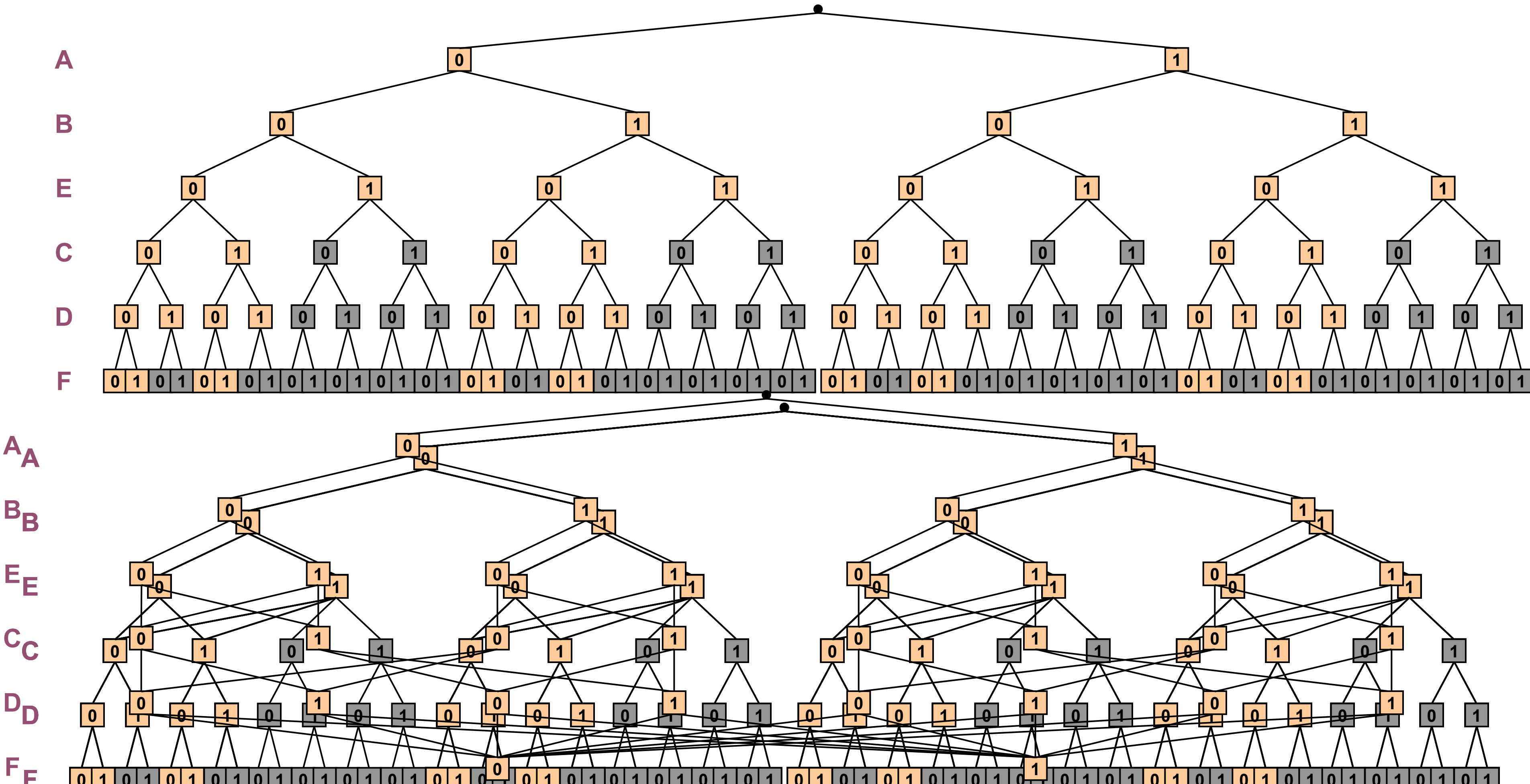
These results show that interleaving an arc-consistency procedure with search was generally quite effective in these studies, as was combining learning and value ordering.
An interesting observation can be made based on the nature of the constraints in each of
the three sets of random problems. The problems with more restrictive, or tighter, constraints, had sparser constraint graphs. With the looser constraints, the difference
in performance among the algorithms was much less than on problems with tighter con-
straints. The arc-consistency enforcing, and constraint-learning procedures were much
more effective on the sparser graphs with tight constraints. These procedures are able to
exploit the local structure in such problems. We also see that FC+AC prune the search
space most effectively.

Outline

- Look-back strategies
- Backjumping: Gaschnig, Graph-based, Conflict-directed
- Learning no-goods, constraint recording.
- Look-back for Satisfiability, integration and Empirical evaluation
- Good caching, counting

Good caching:
Moving from one to all or counting





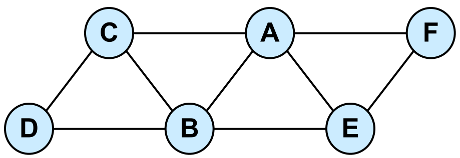
Summary: Time-space for consistency and counting

- **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^*))

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

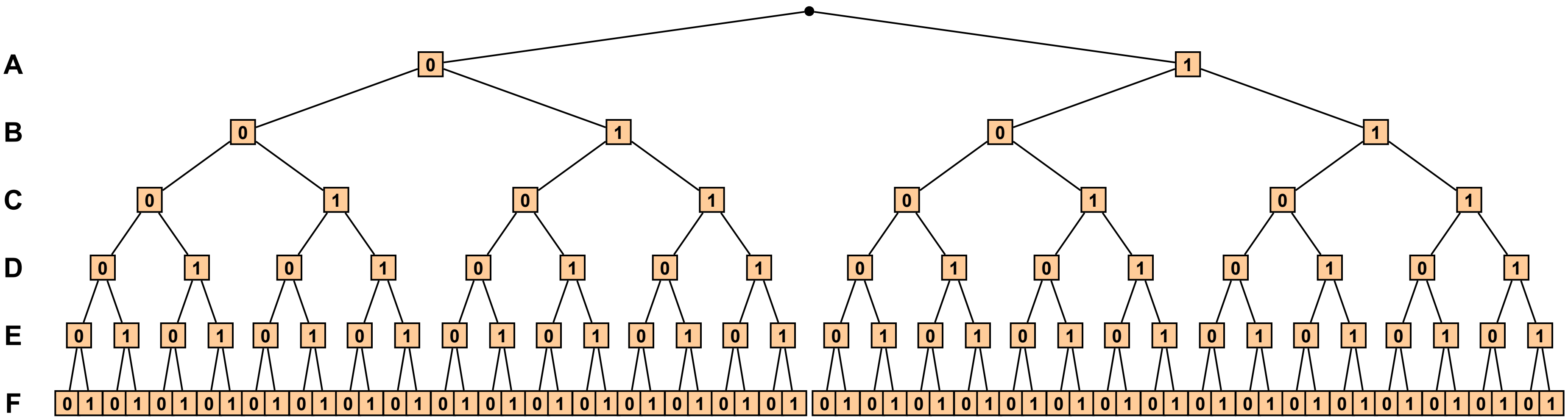


A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

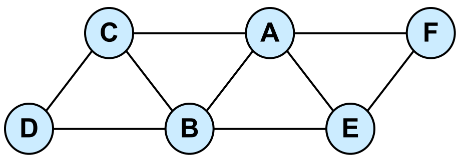
B	C	D	R_{BCD}
0	0	0	**1**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**1**

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**



#CSP – OR Search Tree

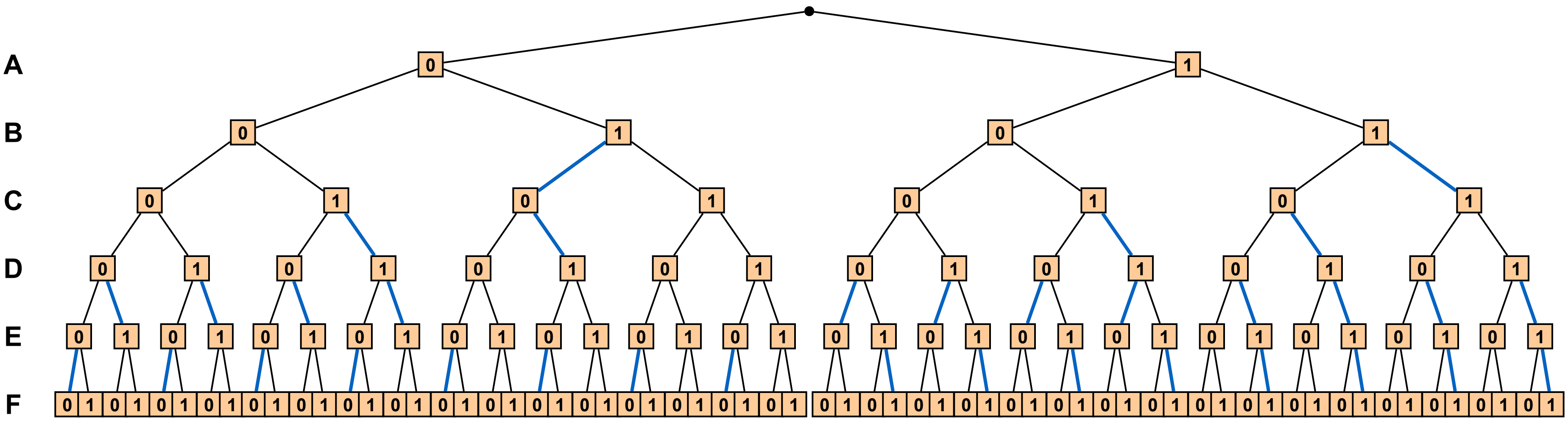


A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

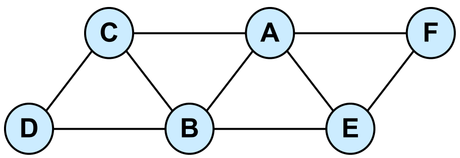
B	C	D	R_{BCD}
0	0	0	**1**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**1**

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**



#CSP - OR Search Tree



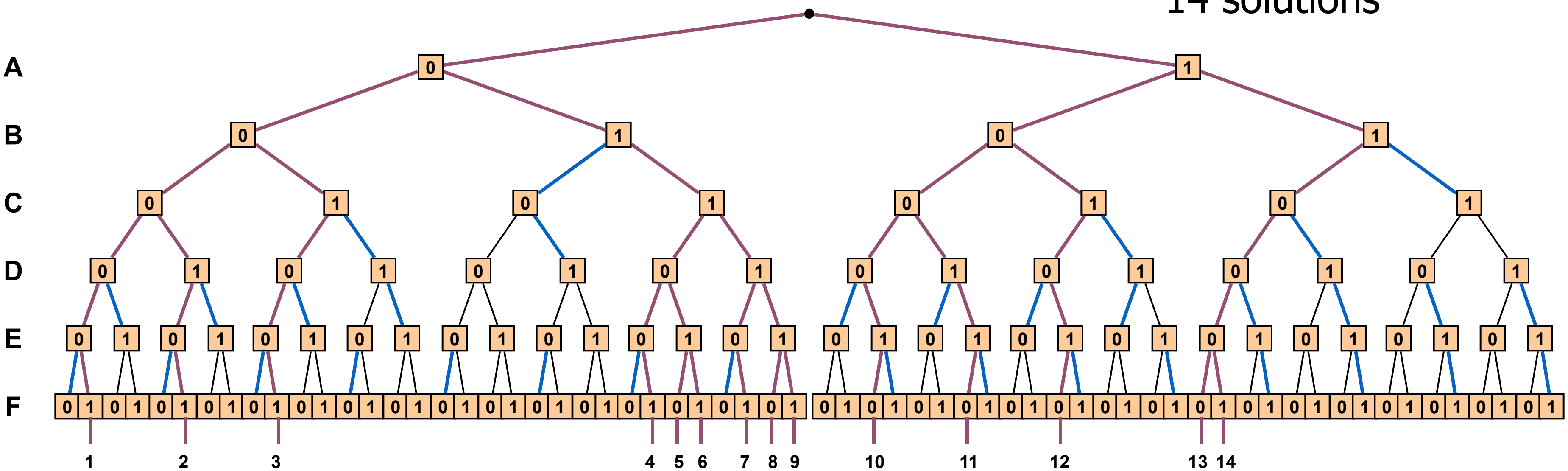
A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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**

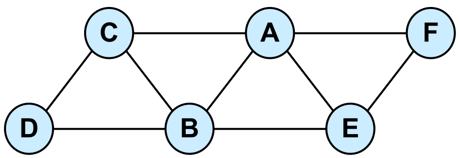
A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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 - Tree DFS Traversal

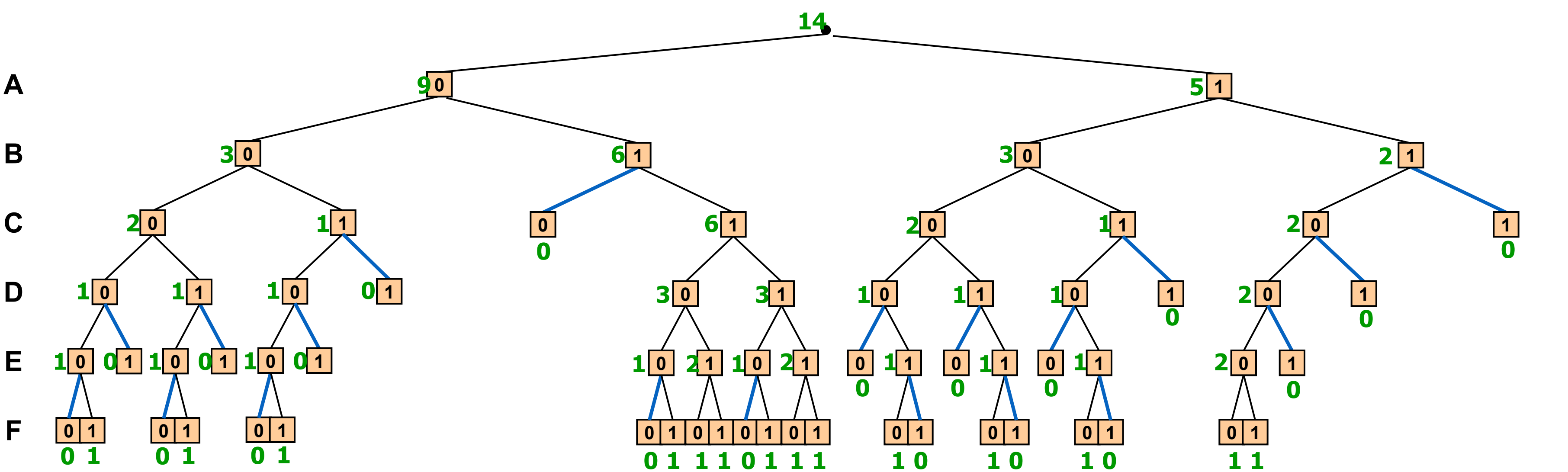


A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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**

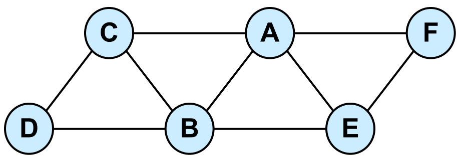
A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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



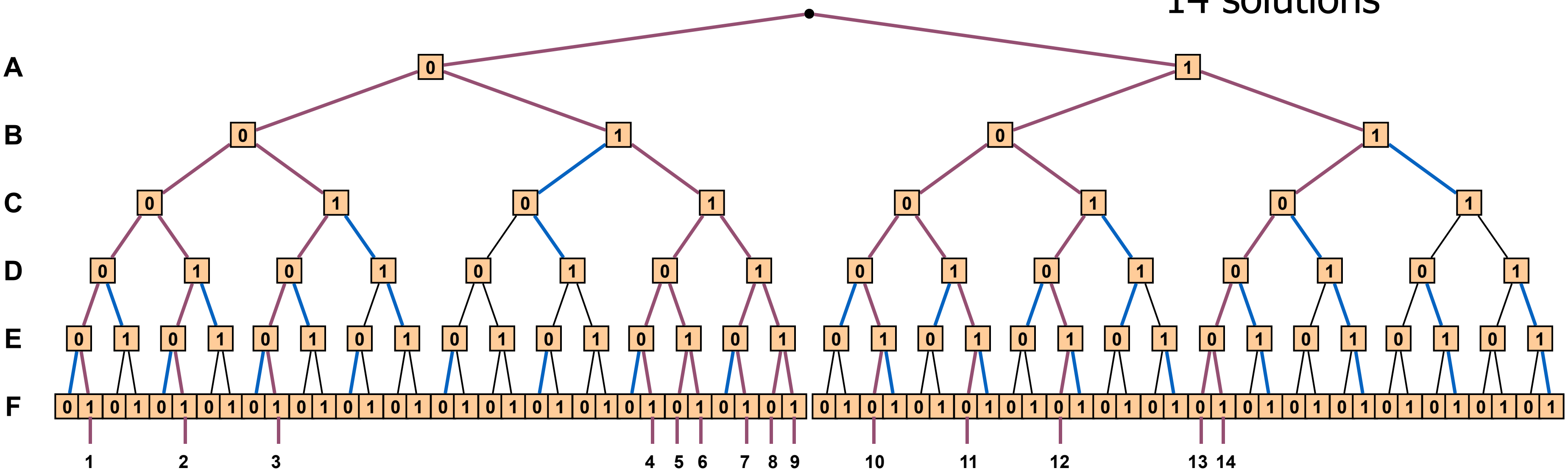
A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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**

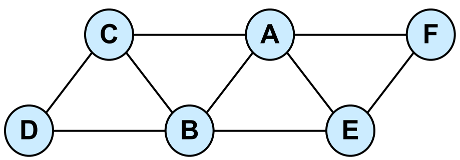
A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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]**

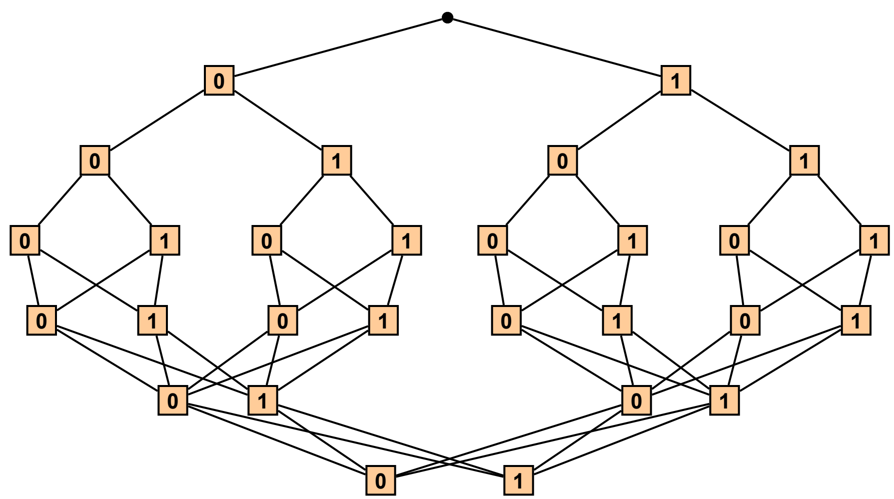
B **context(B) = [AB]**

C **context(C) = [ABC]**

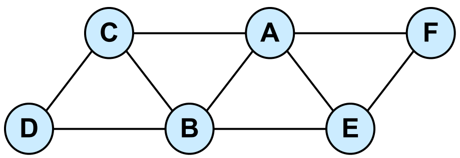
D **context(D) = [ABD]**

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]**

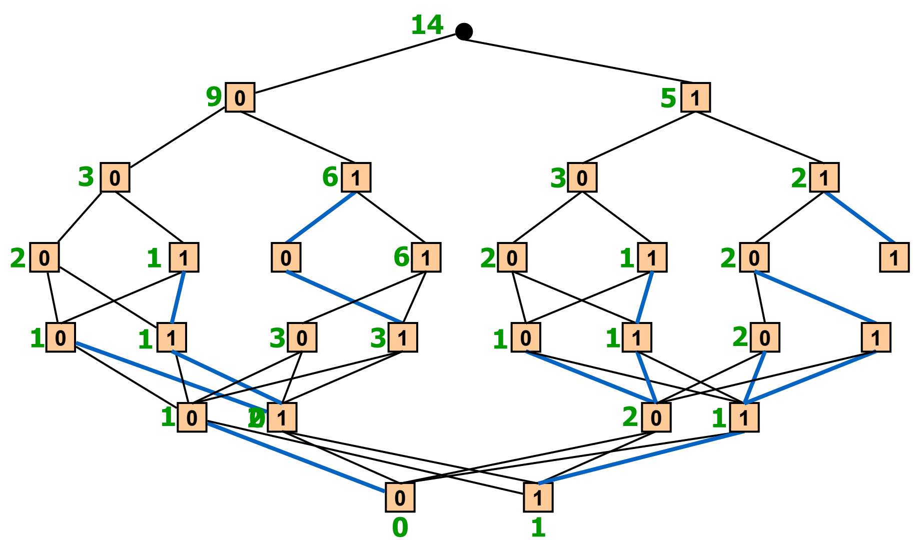
B **context(B) = [AB]**

C **context(C) = [ABC]**

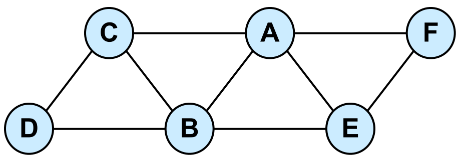
D **context(D) = [ABD]**

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]**

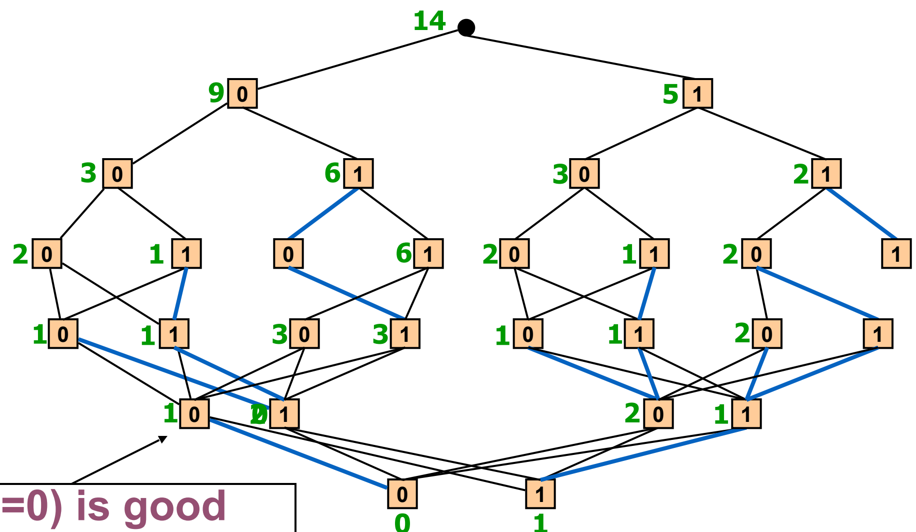
B **context(B) = [AB]**

C **context(C) = [ABC]**

D **context(D) = [ABD]**

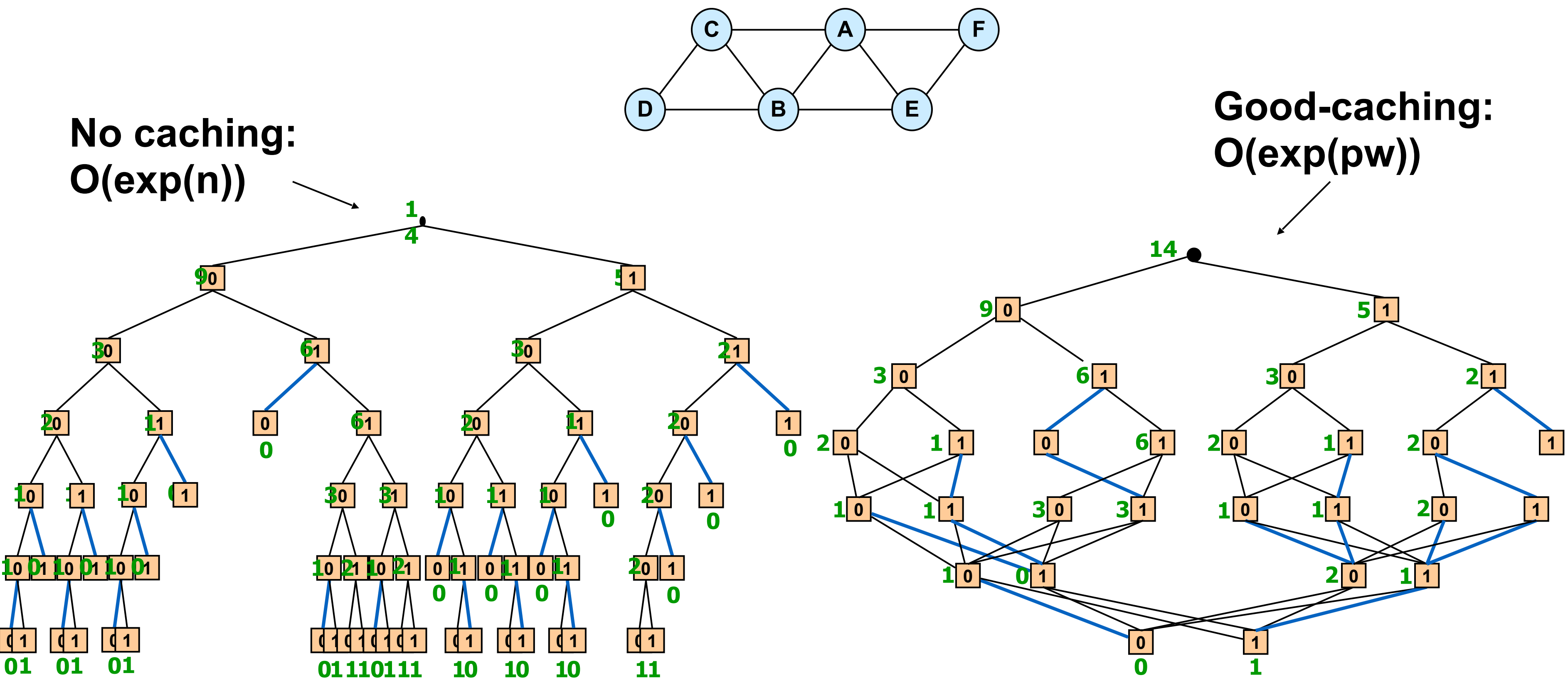
E **context(E) = [AE]**

F **context(F) = [F]**



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

#CSP - Searching the Graph by Good Caching



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

Outline

- Look-back strategies
- Backjumping: Gaschnig, Graph-based, Conflict-directed
- Learning no-goods, constraint recording.
- Look-back for Satisfiability, integration and Empirical evaluation
- Counting, good caching