

Chapter 5: General search strategies: Look-ahead

**ICS 275
Spring 2014**

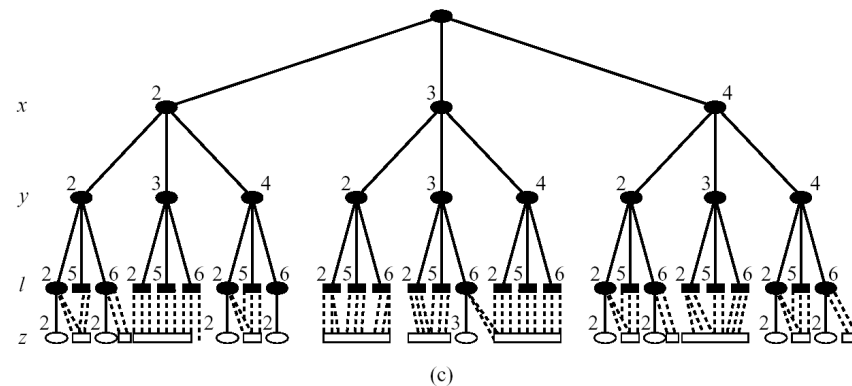
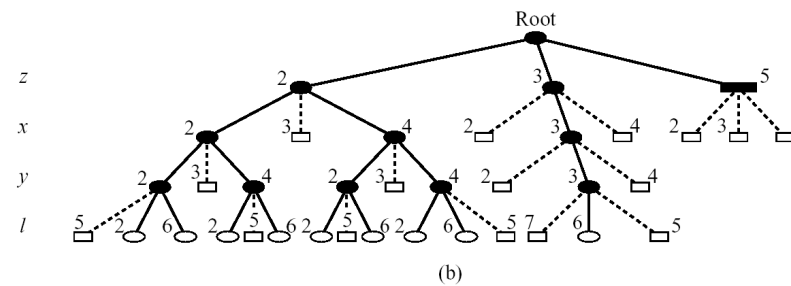
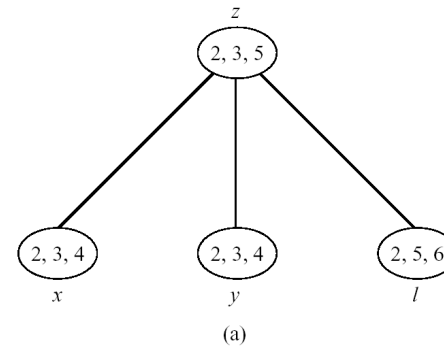
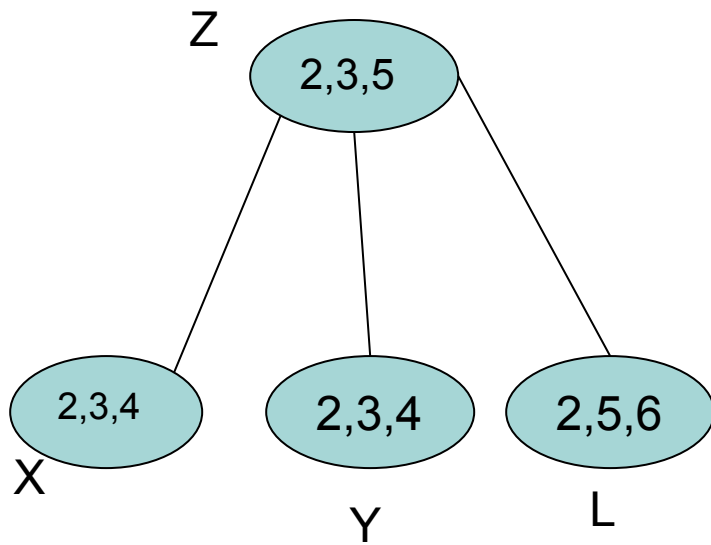
What if the Constraint network is not backtrack-free?

- Backtrack-free in general is too costly so what to do?
- Search?
- What is the search space?
- How to search it? Breadth-first? Depth-first?

The search space for a CN

- A tree of all partial solutions
- A partial solution: (a_1, \dots, a_j) satisfying all relevant constraints
- The size of the underlying search space depends on:
 - Variable ordering
 - Level of consistency possessed by the problem

Search space and the effect of ordering



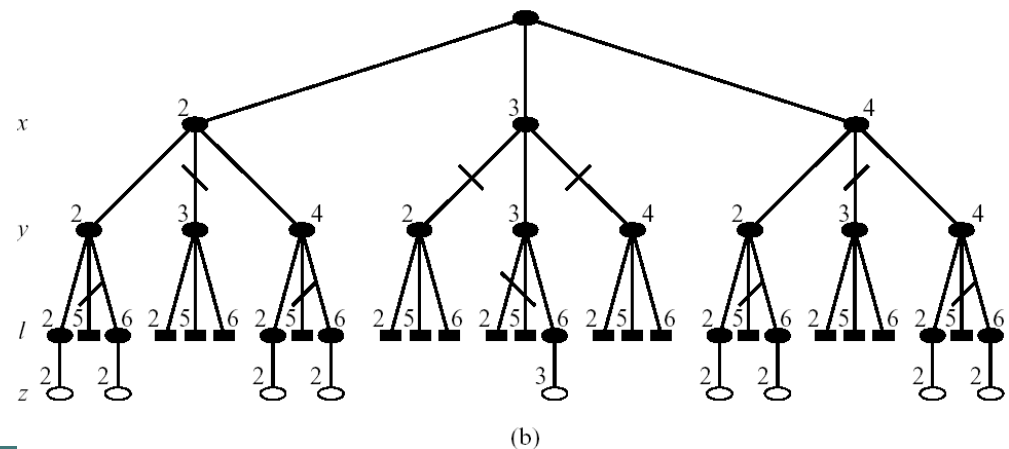
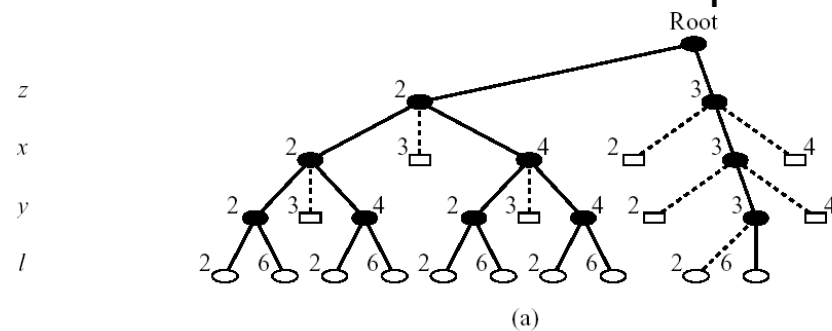
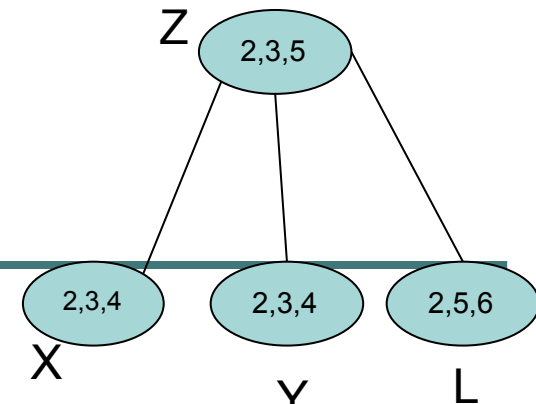
Spring 2014

Dependency on consistency level

- After arc-consistency $z=5$ and $l=5$ are removed

- After path-consistency

- R'_{zx}
- R'_{zy}
- R'_{zl}
- R'_{xy}
- R'_{xl}
- R'_{yl}



The effect of higher consistency on search

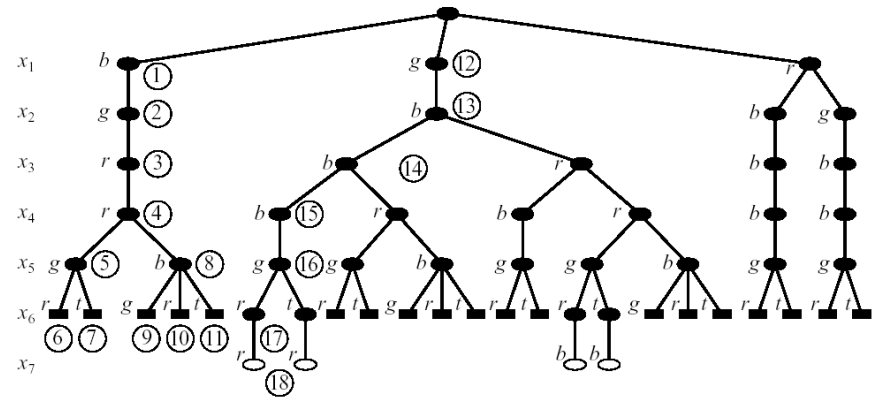
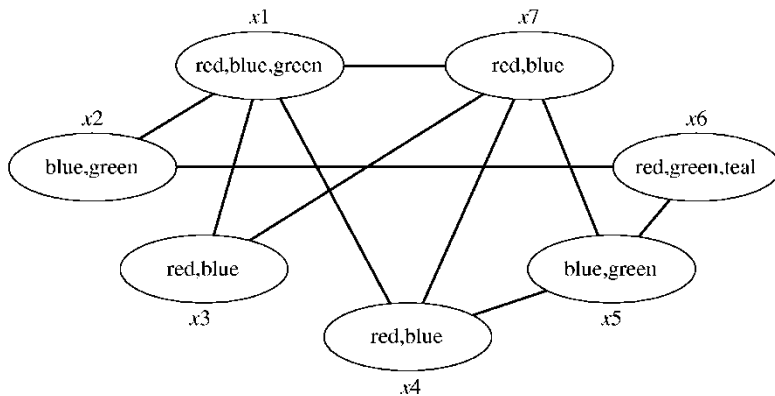
Theorem 5.1.3 *Let \mathcal{R}' be a tighter network than \mathcal{R} , where both represent the same set of solutions. For any ordering d , any path appearing in the search graph derived from \mathcal{R}' also appears in the search graph derived from \mathcal{R} . \square*

Cost of node's expansion

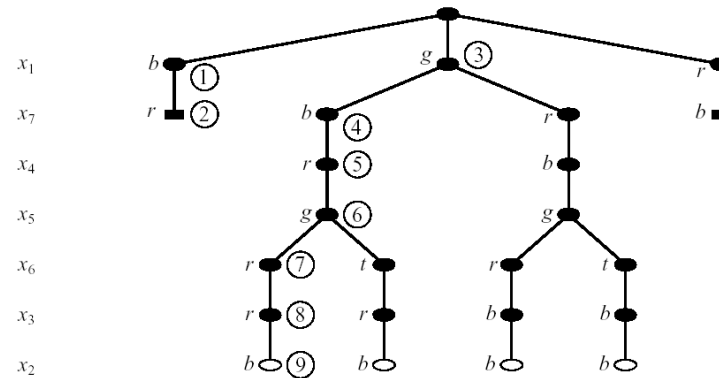
- Number of consistency checks for toy problem:
 - For d_1 : 19 for R , 43 for R'
 - For d_2 : 91 on R and 56 on R'
- Reminder:

Definition 5.1.5 (backtrack-free network) *A network R is said to be backtrack-free along ordering d if every leaf node in the corresponding search graph is a solution.*

Backtracking Search for a Solution

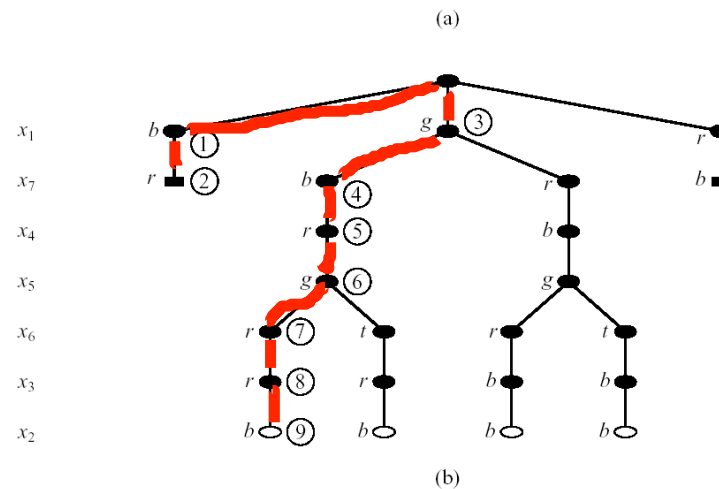
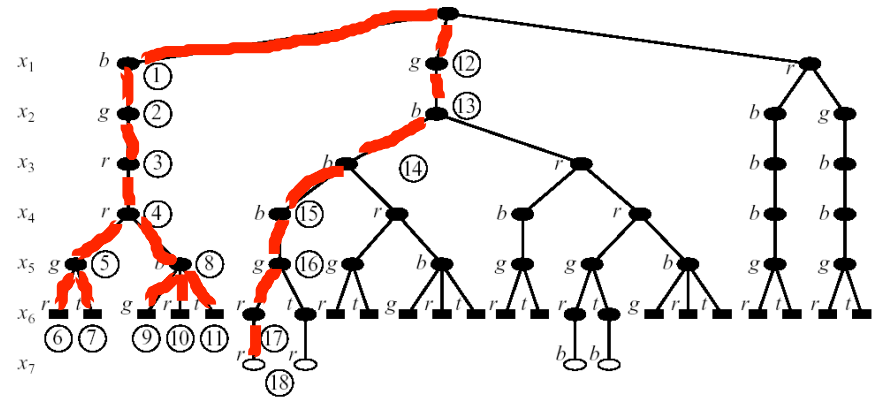
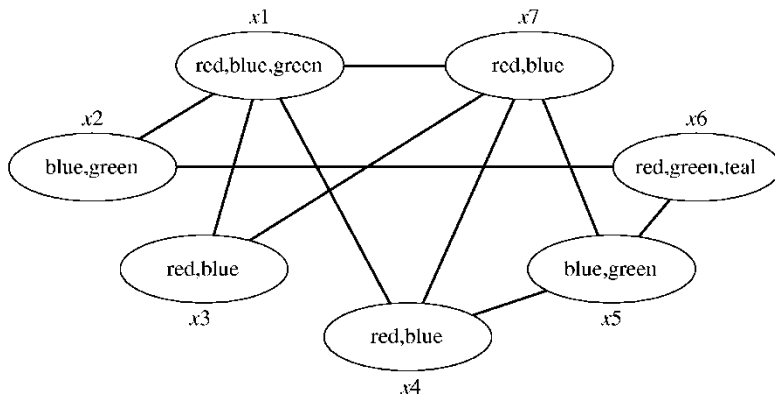


(a)

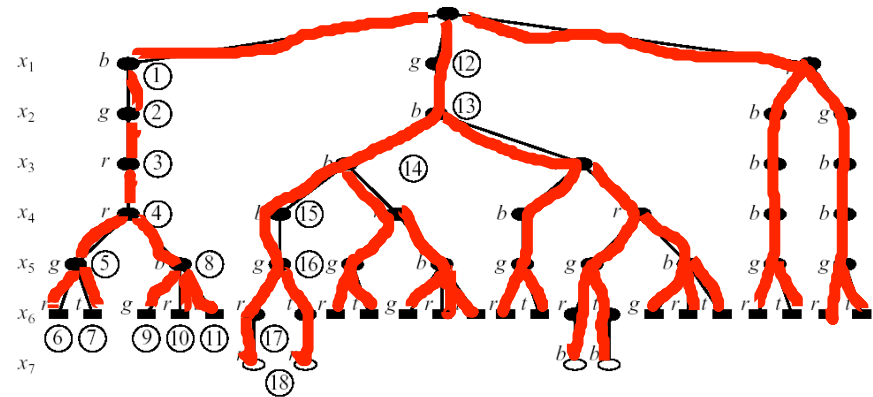
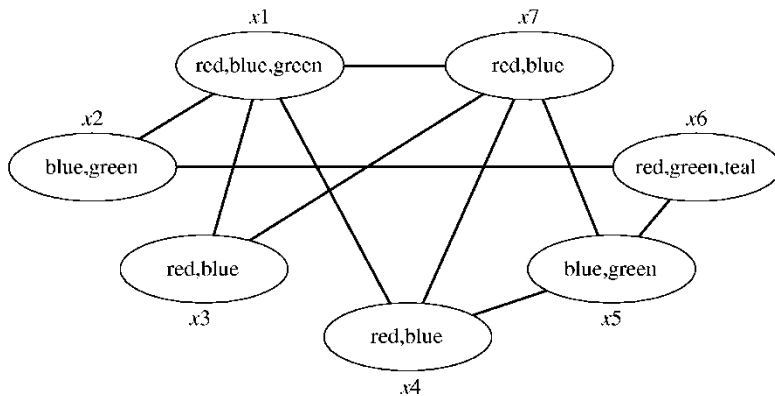


(b)

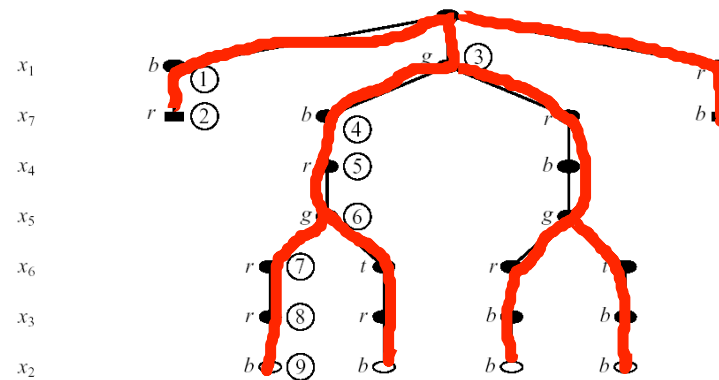
Backtracking Search for a single Solution



Backtracking Search for *All* Solutions

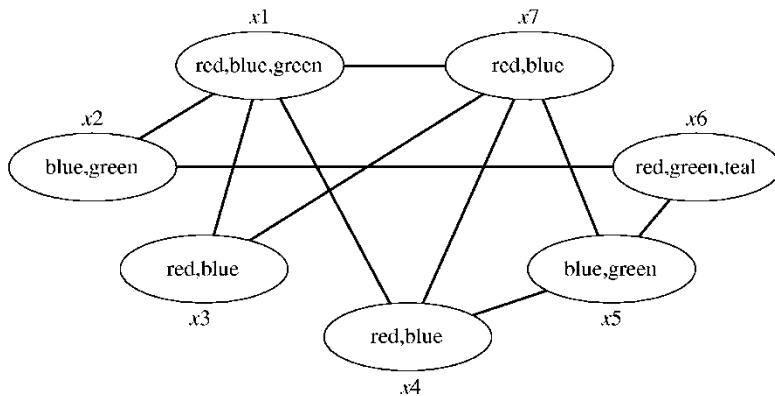


(a)

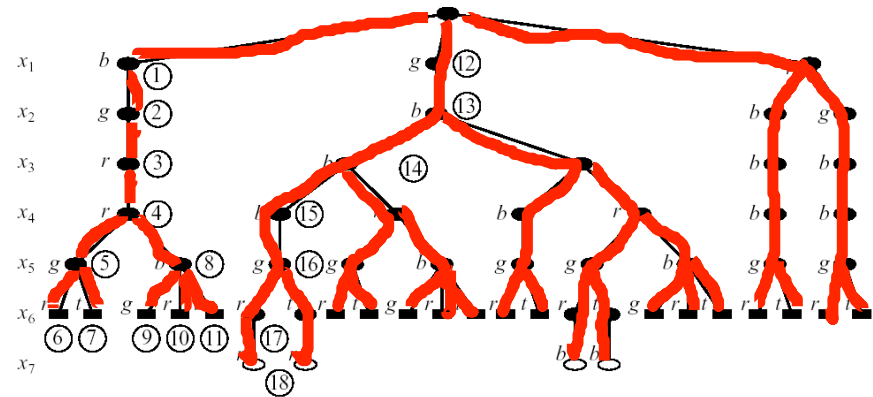


(b)

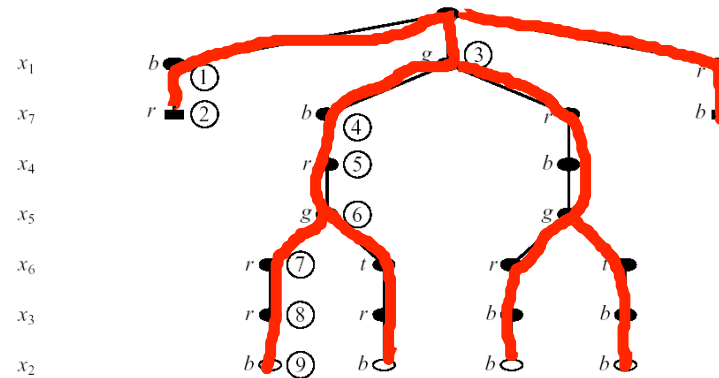
Backtracking Search for *All* Solutions



For all tasks
Time: $O(\exp(n))$
Space: linear

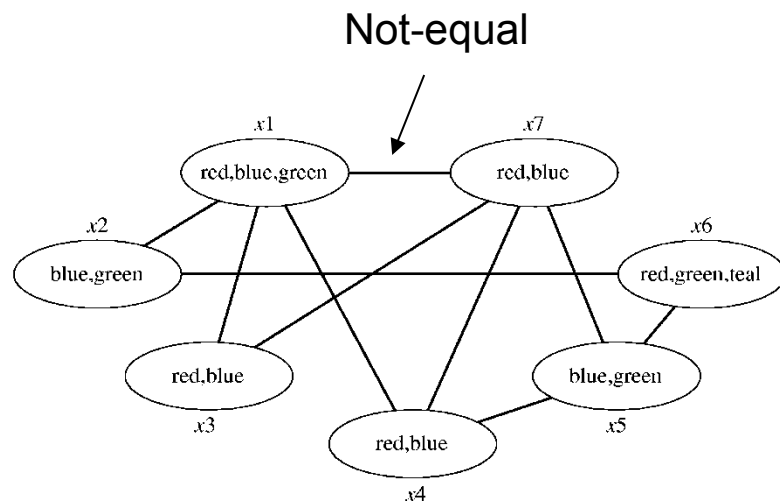


(a)

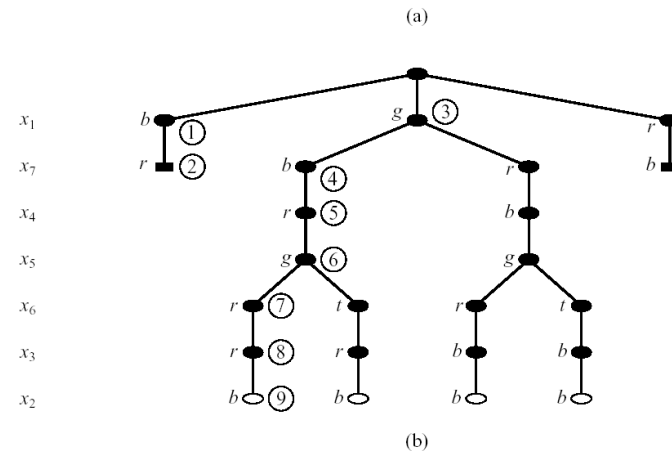
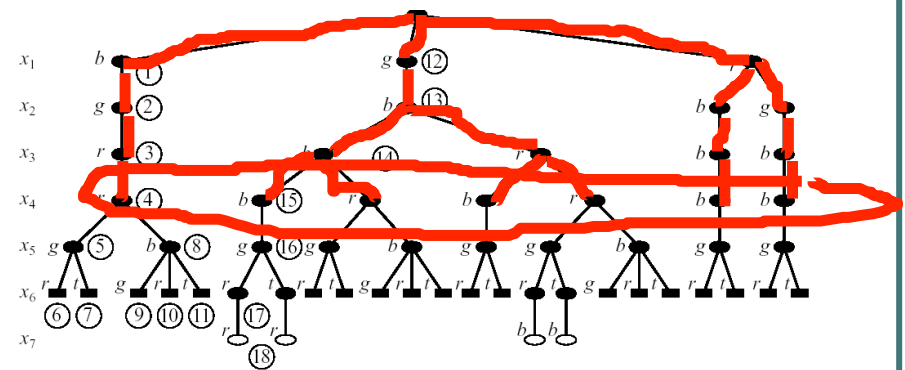


(b)

Traversing Breadth-First (BFS)?



**BFS space is $\exp(n)$ while no
Time gain \rightarrow use DFS**



Backtracking

procedure BACKTRACKING

Input: A constraint network $P = (X, D, C)$.

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

```
 $i \leftarrow 1$                 (initialize variable counter)
 $D'_i \leftarrow D_i$       (copy domain)
while  $1 \leq i \leq n$ 
  instantiate  $x_i \leftarrow \text{SELECTVALUE}$ 
  if  $x_i$  is null          (no value was returned)
     $i \leftarrow i - 1$       (backtrack)
  else
     $i \leftarrow i + 1$       (step forward)
     $D'_i \leftarrow D_i$ 
  end while
if  $i = 0$ 
  return "inconsistent"
else
  return instantiated values of  $\{x_1, \dots, x_n\}$ 
end procedure
```

subprocedure SELECTVALUE (return a value in D'_i consistent with \bar{a}_{i-1})

```
while  $D'_i$  is not empty
  select an arbitrary element  $a \in D'_i$ , and remove  $a$  from  $D'_i$ 
  if CONSISTENT( $\bar{a}_{i-1}, x_i = a$ )
    return  $a$ 
end while
return null                (no consistent value)
end procedure
```

- Complexity of extending a partial solution:
 - Complexity of consistent $O(e \log t)$, t bounds tuples, e constraints
 - Complexity of selectValue $O(e k \log t)$

Figure 5.4: The backtracking algorithm.

Improving backtracking

- Before search: (reducing the search space)
 - Arc-consistency, path-consistency
 - Variable ordering (fixed)
- During search:
 - Look-ahead schemes:
 - value ordering,
 - variable ordering (if not fixed)
 - Look-back schemes:
 - Backjump
 - Constraint recording
 - Dependency-directed backtracking

Look-ahead: value orderings

- Intuition:
 - Choose value least likely to yield a dead-end
 - Approach: apply constraint propagation at each node in the search tree
- Forward-checking
 - (check each unassigned variable separately)
- Maintaining arc-consistency (MAC)
 - (apply full arc-consistency)
- Full look-ahead
 - One pass of arc-consistency (AC-1)
- Partial look-ahead
 - directional-arc-consistency

Generalized look-ahead

procedure GENERALIZED-LOOKAHEAD

Input: A constraint network $P = (X, D, C)$

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

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

$i \leftarrow 1$ (initialize variable counter)

while $1 \leq i \leq n$

 instantiate $x_i \leftarrow \text{SELECTVALUE-XXX}$

if x_i is null (no value was returned)

$i \leftarrow i - 1$ (backtrack)

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

else

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

end while

if $i = 0$

return "inconsistent"

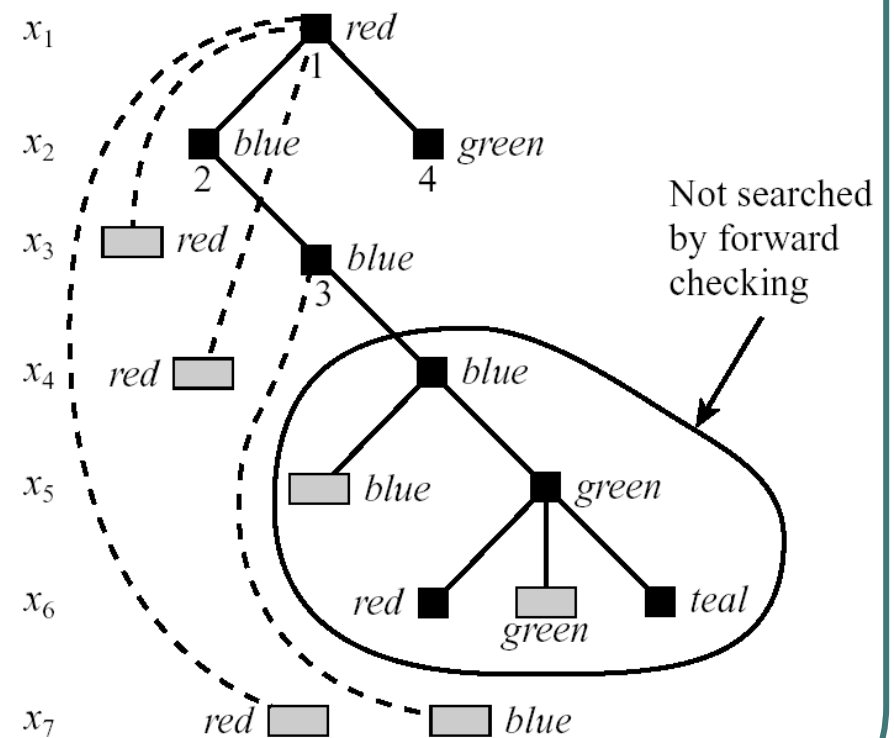
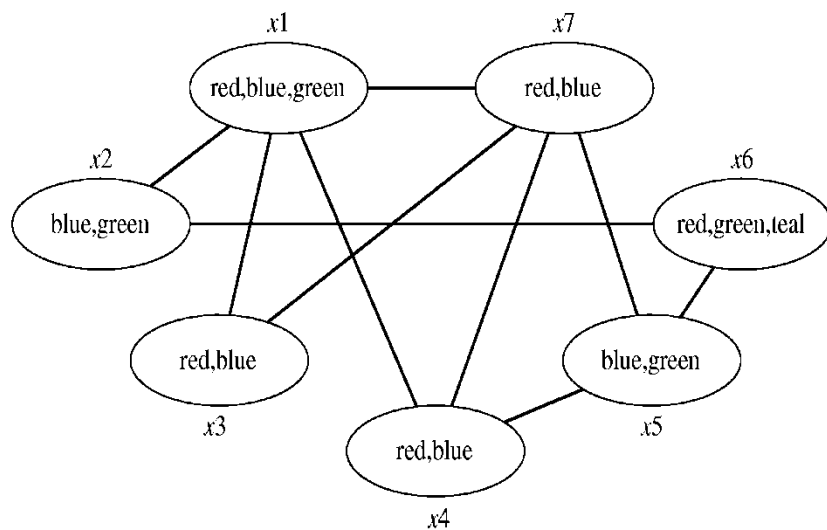
else

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

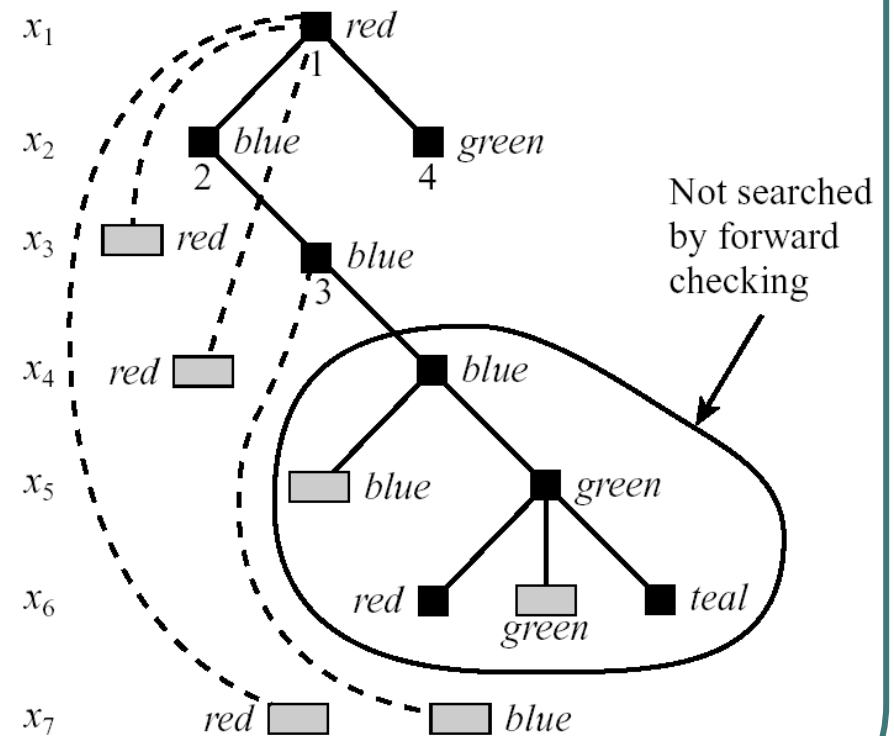
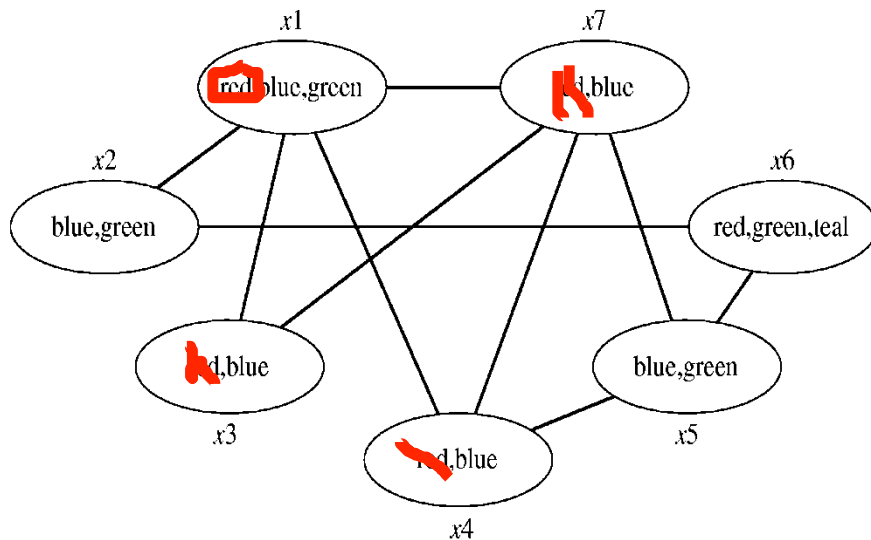
end procedure

Figure 5.7: A common framework for several look-ahead based search algorithms. By replacing SELECTVALUE-XXX with SELECTVALUE-FORWARD-CHECKING, the forward checking algorithm is obtained. Similarly, using SELECTVALUE-ARC-CONSISTENCY yields an algorithm that interweaves arc-consistency and search.

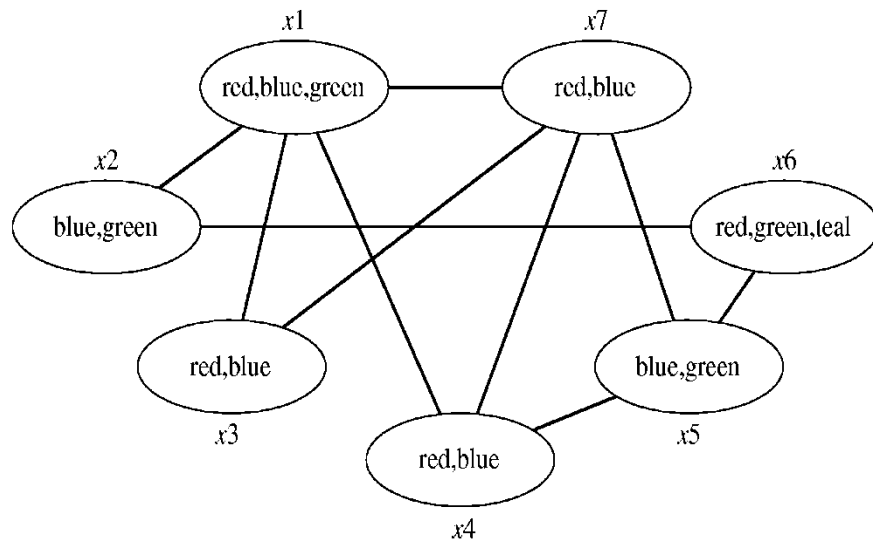
Forward-checking example



Forward-Checking for Value Selection

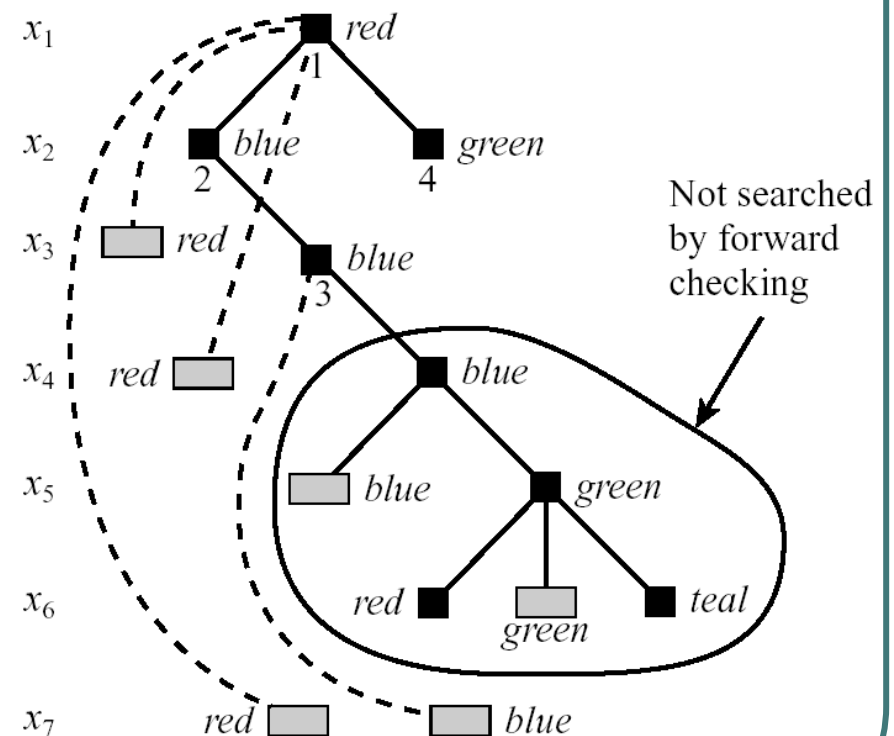


Forward-Checking for Value Ordering

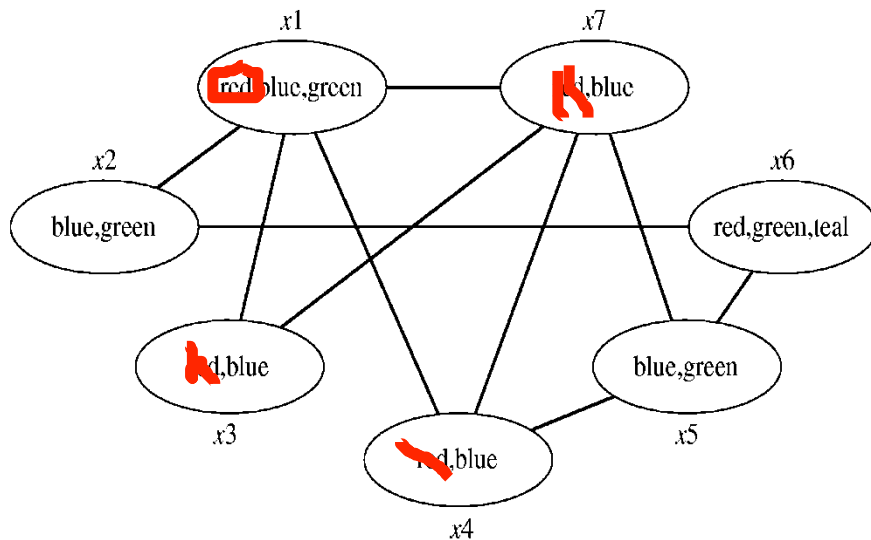


FC overhead: $O(ek^2)$

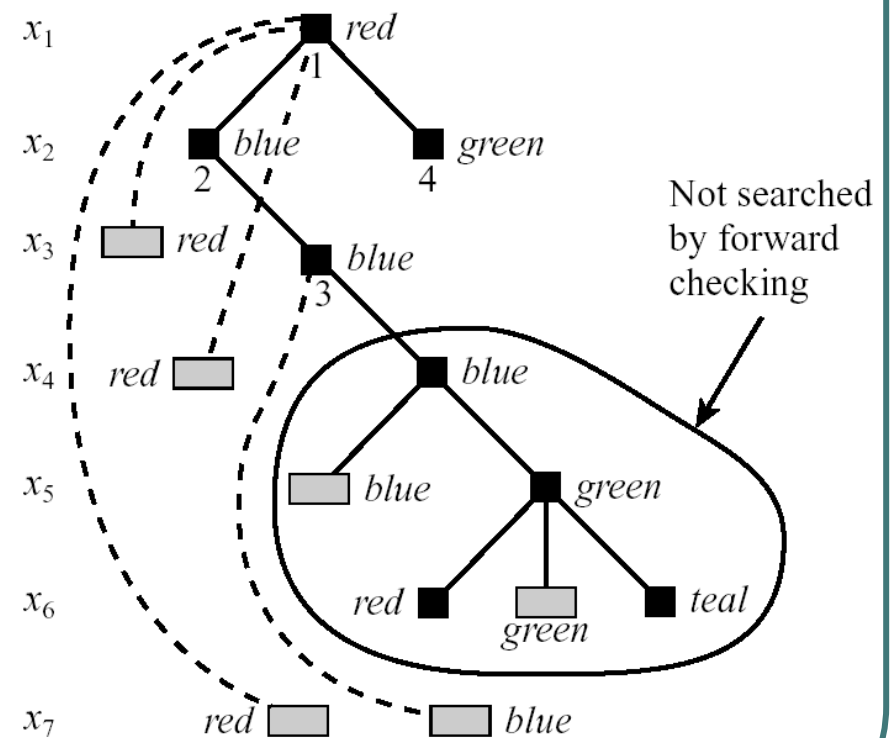
For each value of a future variable e_u
 Tests: $O(k e_u)$, for all future variables $O(ke)$
 For all current domain $O(k^2 e)$



Forward-Checking for Value Ordering



FW overhead: : $O(ek^2)$



Forward-checking

```
procedure SELECTVALUE-FORWARD-CHECKING
  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( $\vec{a}_{i-1}, x_i = a, x_k = b$ )
          remove  $b$  from  $D'_k$ 
      end for
      if  $D'_k$  is empty      ( $x_i = a$  leads to a dead-end)
        empty-domain  $\leftarrow$  true
      if empty-domain    (don't select  $a$ )
        reset each  $D'_k, i < k \leq n$  to value before  $a$  was selected
      else
        return  $a$ 
    end while
    return null          (no consistent value)
end procedure
```

Figure 5.8: The SELECTVALUE subprocedure for the forward checking algorithm.

Complexity of selectValue-forward-checking at each node: $O(ek^2)$

Arc-consistency look-ahead

(Gashnig, 1977)

- Applies full arc-consistency on all un-instantiated variables following each value assignment to the current variable.
- Complexity:
 - If optimal arc-consistency is used: $O(ek^3)$
 - What is the complexity overhead when AC-1 is used at each node?

Forward-checking: $O(ek^2)$

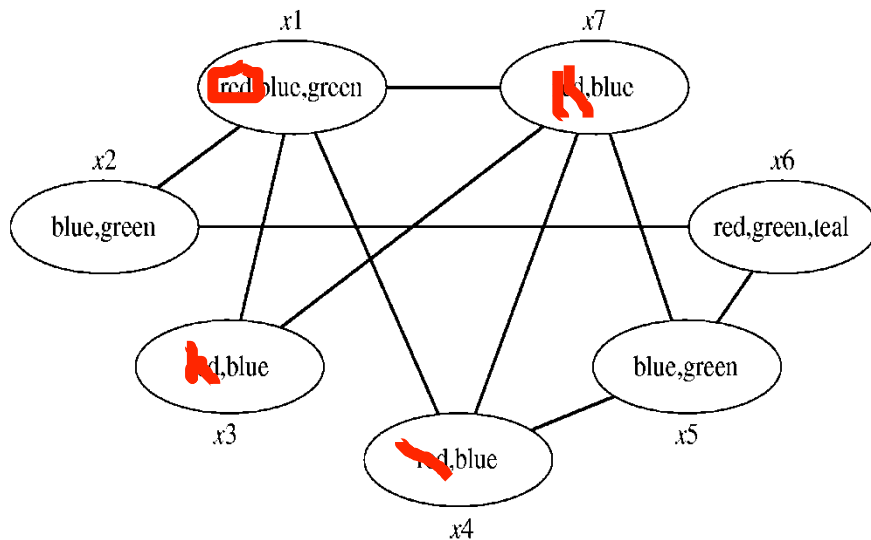
Full arc-consistency look-ahead $O(ek^3)$

With optimal AC:

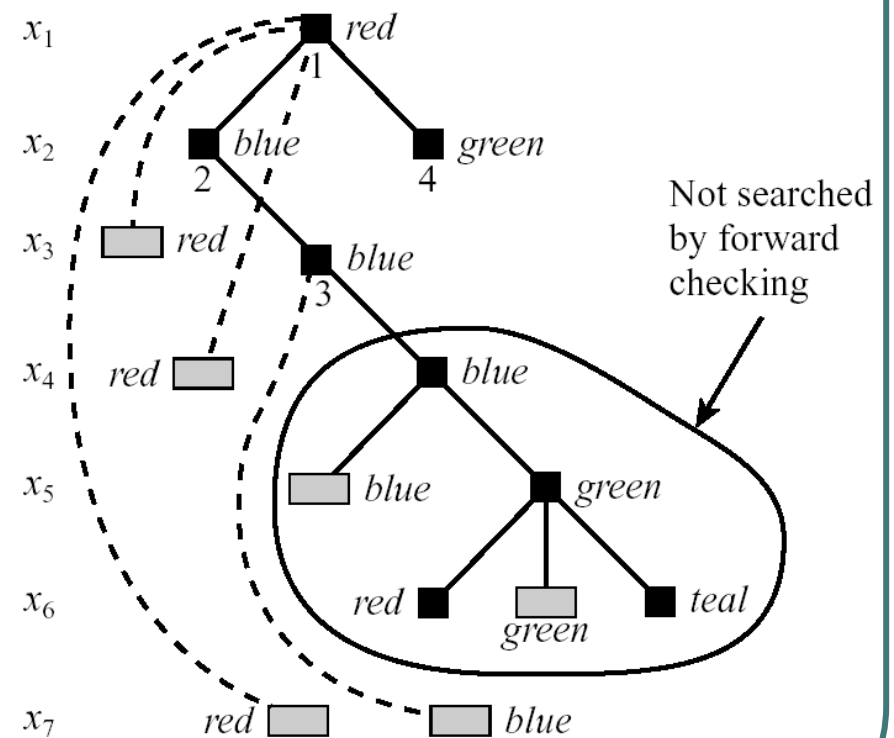
MAC: Maintaining arc-consistency (Sabin and Freuder 1994)

- Perform arc-consistency in a binary search tree: Given a domain $X=\{1,2,3,4\}$ the algorithm assigns $X=1$ (and apply arc-consistency) and if $x=1$ is pruned, it applies arc-consistency to $X=\{2,3,4\}$
- If inconsistency is not discovered, a new variable is selected (not necessarily X)

MAC for Value Ordering

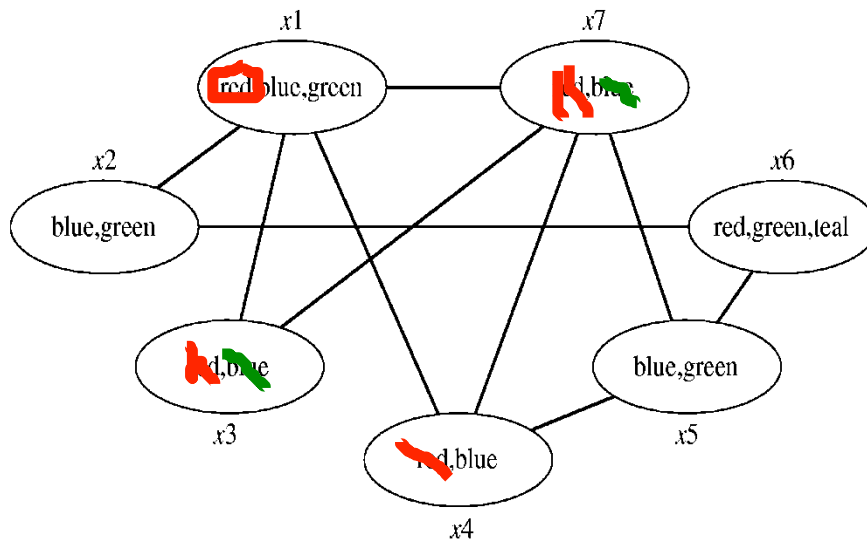
FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$



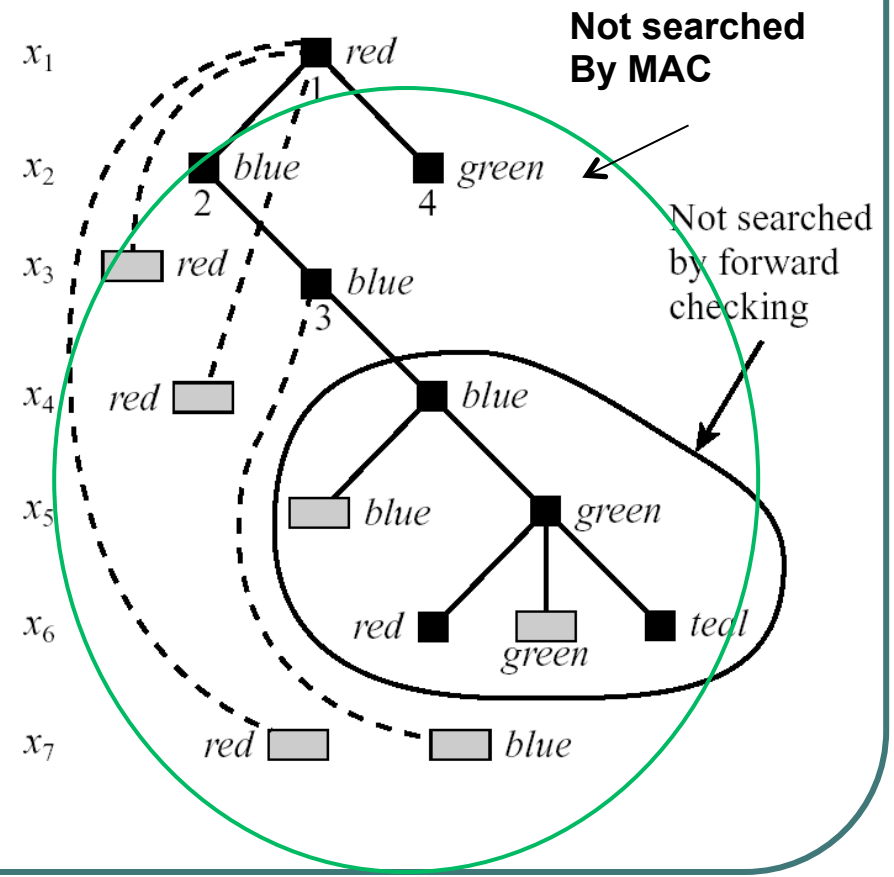
MAC for Value Ordering

Arc-consistency prunes $x_1 = \text{red}$
Prunes the whole tree



FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$



Arc-consistency look-ahead: (a variant: maintaining arc-consistency MAC)

```
subprocedure SELECTVALUE-ARC-CONSISTENCY

  while  $D'_i$  is not empty
    select an arbitrary element  $a \in D'_i$ , and remove  $a$  from  $D'_i$ 
    repeat
      removed-value  $\leftarrow$  false
      for all  $j, i < j \leq n$ 
        for all  $k, i < k \leq n$ 
          for each value  $b$  in  $D'_j$ 
            if there is no value  $c \in D'_k$  such that
               $\text{CONSISTENT}(\vec{a}_{i-1}, x_i = a, x_j = b, x_k = c)$ 
              remove  $b$  from  $D'_j$ 
              removed-value  $\leftarrow$  true
          end for
        end for
      end for
    until removed-value = false
    if any future domain is empty (don't select  $a$ )
      reset each  $D'_j, i < j \leq n$ , to value before  $a$  was selected
    else
      return  $a$ 
    end while
  return null (no consistent value)
end procedure
```

Figure 5.10: The SELECTVALUE subprocedure for arc-consistency, based on the AC-1 algorithm.

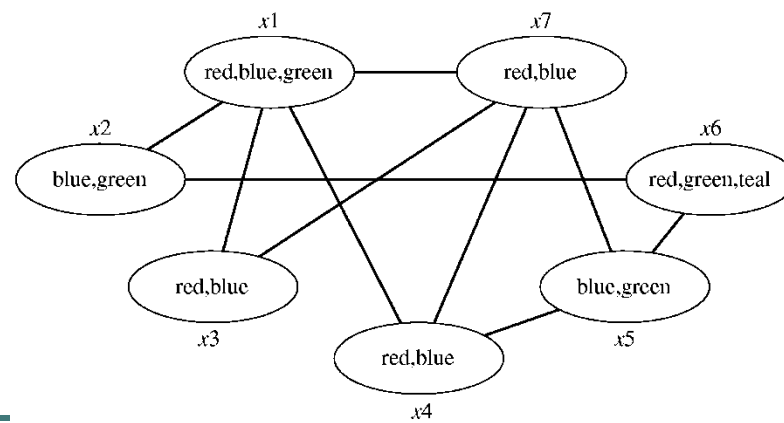
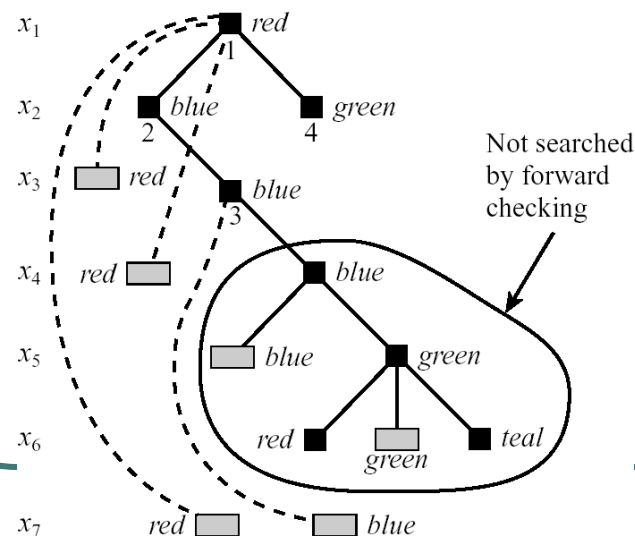
Full and partial look-ahead

- Full looking ahead:
 - Make one pass through future variables (delete, repeat-until)
- Partial look-ahead:
 - Applies (similar-to) directional arc-consistency to future variables.
 - Complexity: also $O(ek^3)$
 - More efficient than MAC

Example of partial look-ahead

Example 5.3.3 Consider the problem in Figure 5.3 using the same ordering of variables and values as in Figure 5.9. Partial-look-ahead starts by considering $x_1 = \text{red}$. Applying directional arc-consistency from x_1 towards x_7 will first shrink the domains of x_3 , x_4 and x_7 , (when processing x_1), as was the case for forward-checking. Later, when directional arc-consistency processes x_4 (with its only value, “blue”) against x_7 (with its only value, “blue”), the domain of x_4 will become empty, and the value “red” for x_1 will be rejected. Likewise, the value $x_1 = \text{blue}$ will be rejected. Therefore, the whole tree in Figure 5.9 will not be visited if either partial-look-ahead or the more extensive look-ahead schemes are used. With this level of look-ahead only the subtree below $x_1 = \text{green}$ will be expanded.

□



Branching-ahead: Dynamic Value Ordering

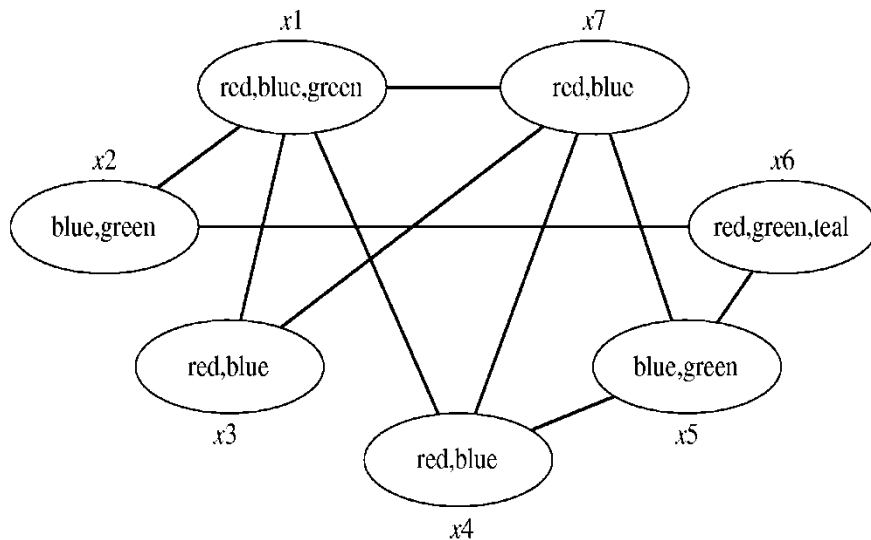
Rank order the promise in non-rejected values

- Rank functions
 - MC (min conflict)
 - MD (min domain)
 - SC (expected solution counts)
- MC results (Frost and Dechter, 1996)
- SC – currently shows good performance using IJGP (Kask, Dechter and Gogate, 2004)

Dynamic Variable Ordering (DVO)

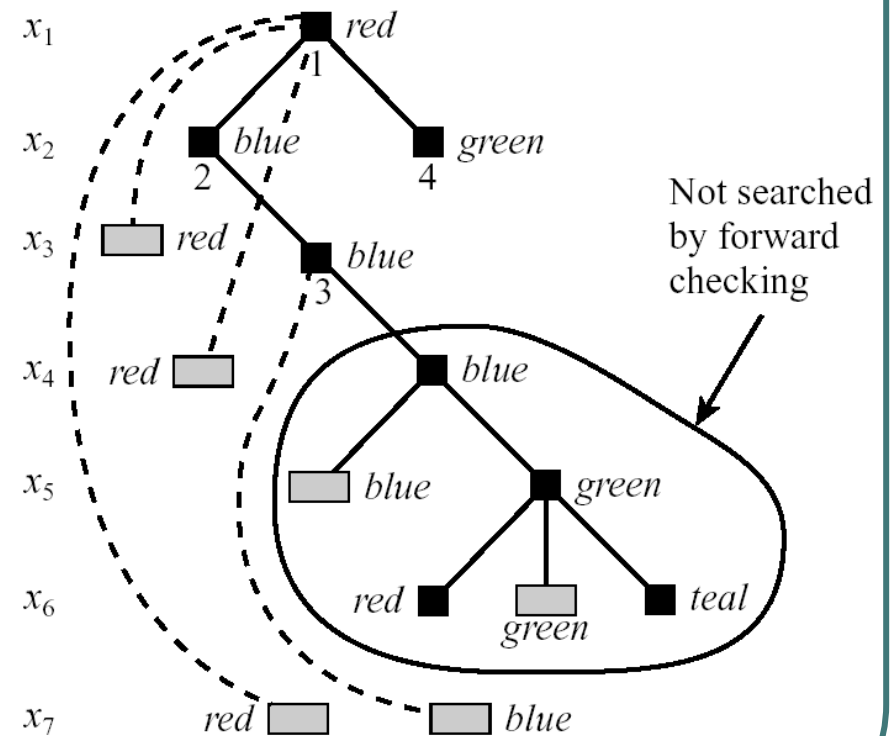
- Following constraint propagation, choose the most constrained variable
- **Intuition:** early discovery of dead-ends
- **Highly effective:** the single most important heuristic to cut down search space
- Most popular with FC
- Dynamic search rearrangement (Bitner and Reingold, 1975) (Purdon, 1983)

Forward-Checking, Variable Ordering



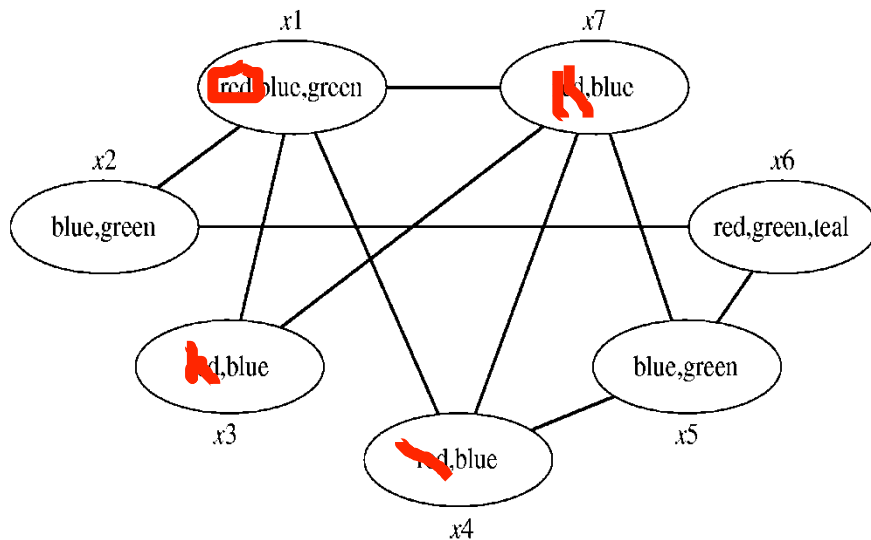
FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$



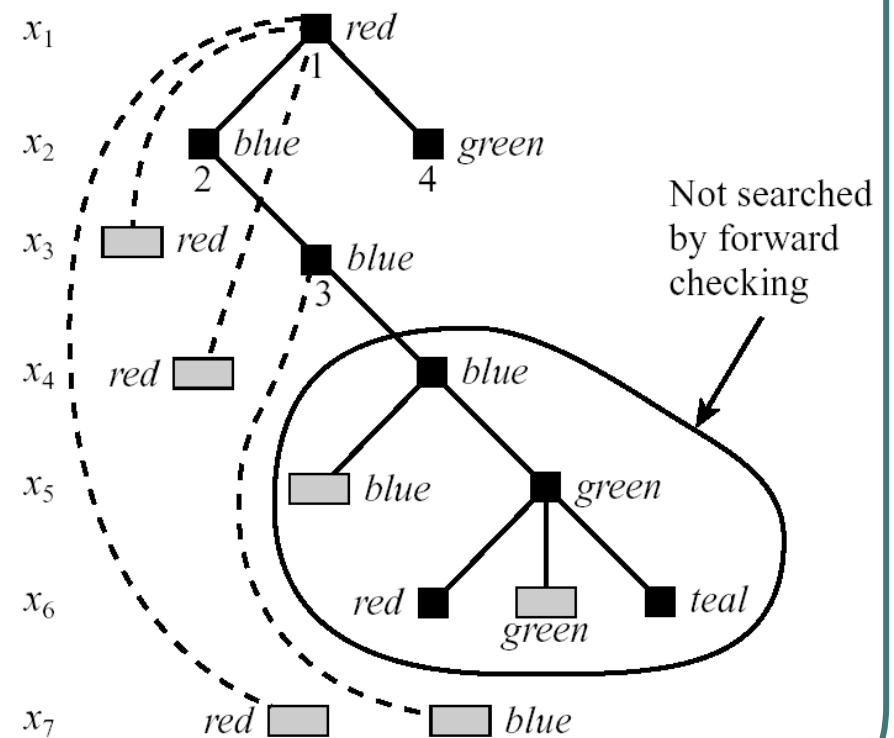
Forward-Checking, Variable Ordering

After $X_1 = \text{red}$ choose X_3 and not X_2



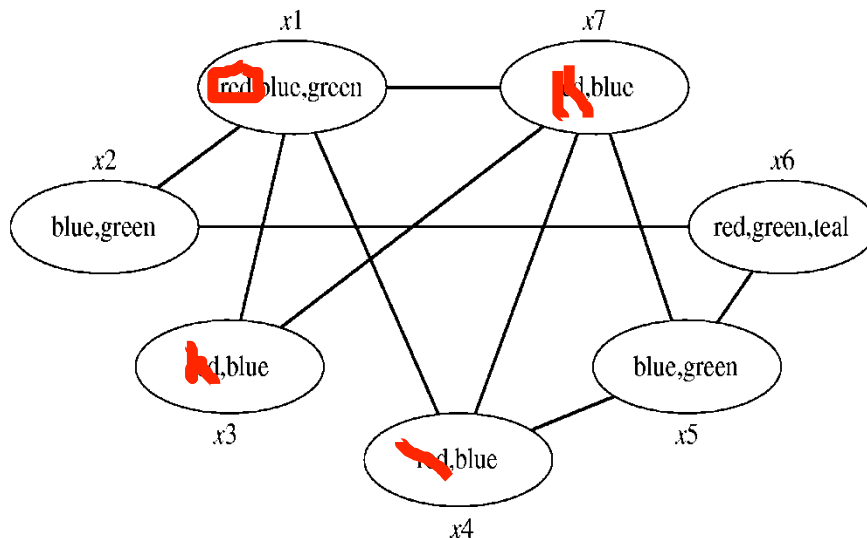
FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$



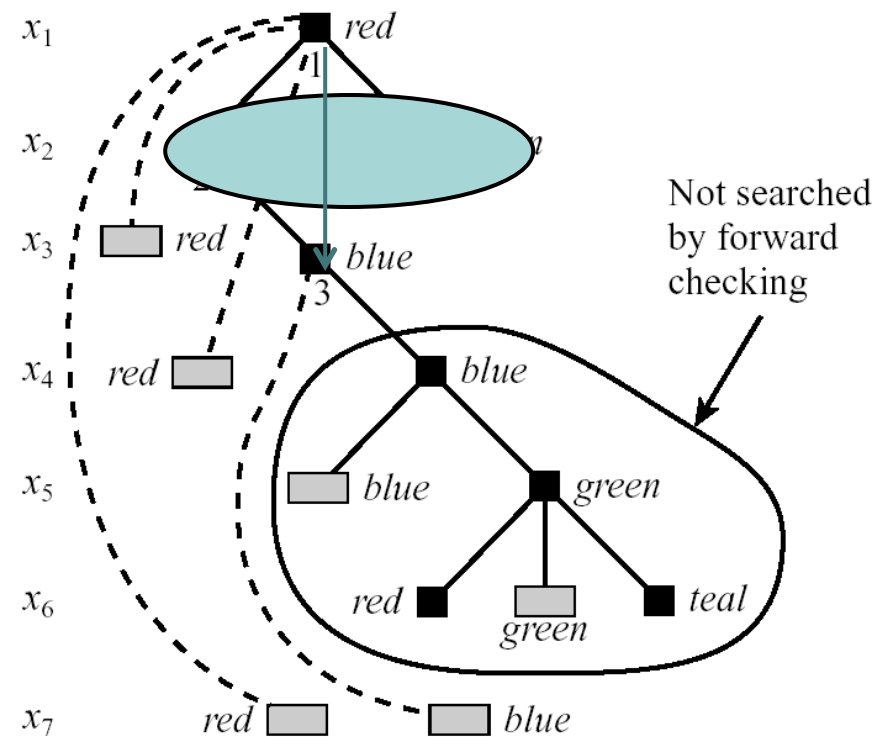
Forward-Checking, Variable Ordering

After $x_1 = \text{red}$ choose x_3 and not x_2



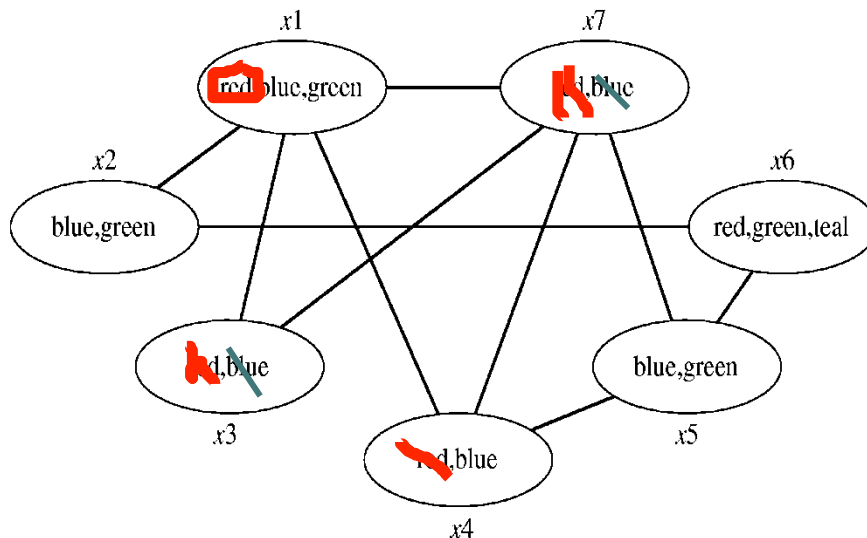
FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$



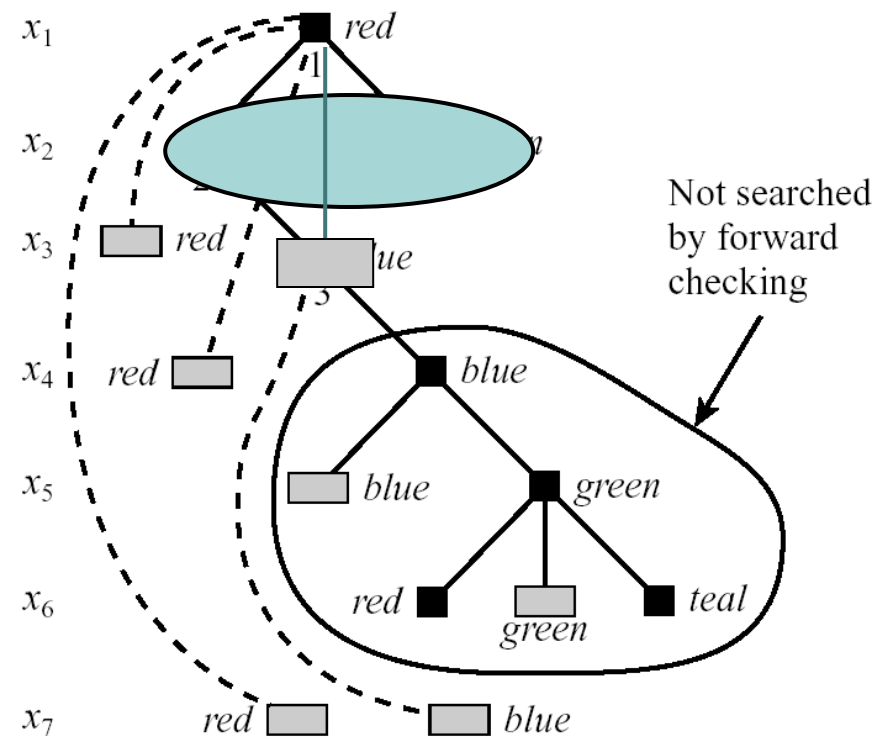
Forward-Checking, Variable Ordering

After $x_1 = \text{red}$ choose x_3 and not x_2

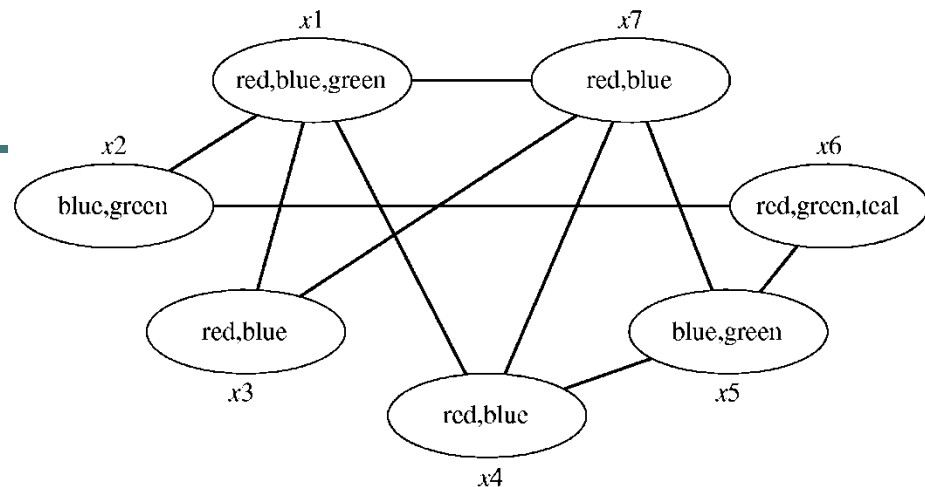


FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$



Example: DVO with forward checking (DVFC)



Example 5.3.4 Consider again the example in Figure 5.3. Initially, all variables have domain size of 2 or more. DVFC picks x_7 , whose domain size is 2, and the value $\langle x_7, \text{blue} \rangle$. Forward-checking propagation of this choice to each future variable restricts the domains of x_3 , x_4 and x_5 to single values, and reduces the size of x_1 's domain by one. DVFC selects x_3 and assigns it its only possible value, *red*. Subsequently, forward-checking causes variable x_1 to also have a singleton domain. The algorithm chooses x_1 and its only consistent value, *green*. After propagating this choice, we see that x_4 has one value, *red*; it is selected and assigned the value. Then x_2 can be selected and assigned its only consistent value, *blue*. Propagating this assignment does not further shrink any future domain. Next, x_5 can be selected and assigned *green*. The solution is then completed, without dead-ends, by assigning *red* or *teal* to x_6 . \square

Algorithm DVO (DVFC)

```
procedure DVFC
Input: A constraint network  $\mathcal{R} = (X, D, C)$ 
Output: Either a solution, or notification that the network is inconsistent.
   $D'_i \leftarrow D_i$  for  $1 \leq i \leq n$     (copy all domains)
   $i \leftarrow 1$                         (initialize variable counter)
   $s = \min_{i < j \leq n} |D'_j|$  (find future var with smallest domain)
   $x_{i+1} \leftarrow x_s$  (rearrange variables so that  $x_s$  follows  $x_i$ )
  while  $1 \leq i \leq n$ 
    instantiate  $x_i \leftarrow \text{SELECTVALUE-FORWARD-CHECKING}$ 
    if  $x_i$  is null                      (no value was returned)
      reset each  $D'$  set to its value before  $x_i$  was last instantiated
       $i \leftarrow i - 1$                 (backtrack)
    else
      if  $i < n$ 
         $i \leftarrow i + 1$               (step forward to  $x_s$ )
         $s = \min_{i < j \leq n} |D'_j|$  (find future var with smallest domain)
         $x_{i+1} \leftarrow x_s$  (rearrange variables so that  $x_s$  follows  $x_i$ )
         $i \leftarrow i + 1$               (step forward to  $x_s$ )
    end while
  if  $i = 0$ 
    return "inconsistent"
  else
    return instantiated values of  $\{x_1, \dots, x_n\}$ 
end procedure
```

Figure 5.12: The DVFC algorithm. It uses the SELECTVALUE-FORWARD-CHECKING sub-procedure given in Fig. 5.8.

DVO: Dynamic Variable Ordering, More involved heuristics

- *dom*: choose a variable with min domain
- *deg*: choose variable with max degree
- *dom+deg*: *dom* and break ties with max degree
- *dom/deg* (Bessiere and Ragin, 96): choose min *dom/deg*
- *dom/wdeg*: domain divided by weighted degree. Constraints are weighted as they get involved in more conflicts. *wdeg*: sum the weights of all constraints that touch *x*.

Implementing look-aheads

- Cost of node generation should be reduced
- Solution: keep a table of viable domains for each variable and each level in the tree.
- Space complexity $O(n^2k)$
- Node generation = table updating $O(e_d k) \Rightarrow O(ek)$

Branching Strategies (selecting the search space)

(see vanBeek, chapter 4 in Handbook)

- Enumeration branching: the naïve backtracking search choice
- A branching strategy in the search tree: a set of branching constraints $p(b_1, \dots, b_j)$ where b_i is a branching constraint
- Branches are often ordered using a heuristic.
- To ensure completeness, the constraints that are ordered on the branches should be exclusive and exhaustive.
- Most common are unary constraints:
 - Enumeration: $(x=1, x=2, x=3, \dots)$
 - Binary choices: $(x=1, x \neq 1)$
 - Domain splitting: $(x > 3, x < 3)$
- Using domain-specific formulas
 - Scheduling: one job before or after: $(x_1 + d_1 < x_2, x_2 + d_2 < x_1)$
 - Can be simulated by auxiliary variables.
 - Searching the dual problem
 - Formula-based splitting in SAT

Randomization

- Randomized variable selection (for tie breaking rule)
- Randomized value selection (for tie breaking rule)
- Random restarts with increasing time-cutoff
- Capitalizing on huge performance variance
- All modern SAT solvers that are competitive use restarts.

The cycle-cutset effect

- A cycle-cutset is a subset of nodes in an undirected graph whose removal results in a graph with no cycles
- A constraint problem whose graph has a cycle-cutset of size c can be solved by partial look-ahead in time $O((n - c)k^{(c+2)})$

Extension to stronger look-ahead

- Extend to path-consistency or i-consistency or generalized-arc-consistency

Definition 5.3.7 (general arc-consistency) *Given a constraint $C = (R, S)$ and a variable $x \in S$, a value $a \in D_x$ is supported in C if there is a tuple $t \in R$ such that $t[x] = a$. t is then called a support for $\langle x, a \rangle$ in C . C is arc-consistent if for each variable x , in its scope and each of its values, $a \in D_x$, $\langle x, a \rangle$ has a support in C . A CSP is arc-consistent if each of its constraints is arc-consistent.*

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. Q = some unassigned variable;
6. return(DPLL($\varphi \wedge Q$) \vee
 DPLL($\varphi \wedge \neg Q$))

Figure 5.13: The DPLL Procedure

Example of DPLL

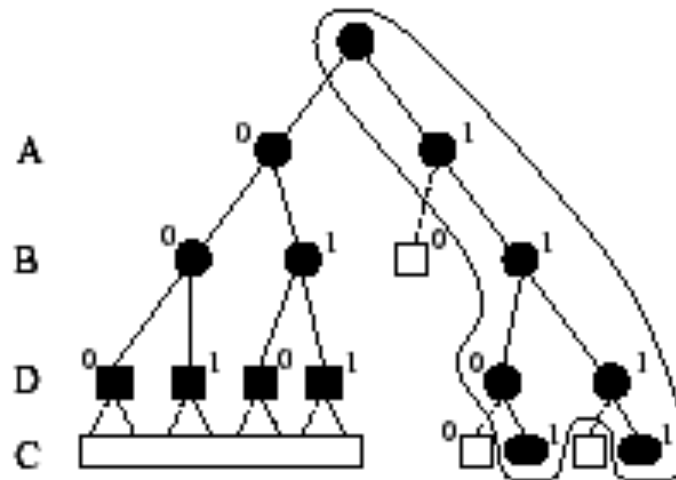


Figure 5.14: A backtracking search tree along the variables A, B, D, C for a cnf theory $\varphi = \{(\neg A \vee B), (\neg C \vee A), (A \vee B \vee D), C\}$. Hollow nodes and bars in the search tree represent illegal states, triangles represent solutions. The enclosed area corresponds to DPLL with unit-propagation.

What is SAT?

Given a sentence:

- **Sentence:** conjunction of clauses
$$(c_1 \vee \neg c_4 \vee c_5 \vee c_6) \wedge (c_2 \vee \neg c_3) \wedge (\neg c_4)$$
- **Clause:** disjunction of literals $(c_2 \vee \neg c_3)$
- **Literal:** a term or its negation $c_1, \neg c_6$
- **Term:** Boolean variable $c_1 = 1 \Leftrightarrow \neg c_1 = 0$

Question: Find an assignment of truth values to the Boolean variables such the sentence is satisfied.

CSP is NP-Complete

- Verifying that an assignment for all variables is a solution
 - Provided constraints can be checked in polynomial time
- Reduction **from 3SAT to CSP**
 - Many such reductions exist in the literature (perhaps 7 of them)

Problem reduction

Example: CSP into SAT *(proves nothing, just an exercise)*

Notation: variable-value pair = vvp

- vvp \rightarrow term
 - $V_1 = \{a, b, c, d\}$ yields $x_1 = (V_1, a)$, $x_2 = (V_1, b)$, $x_3 = (V_1, c)$, $x_4 = (V_1, d)$,
 $x_1 \vee x_2 \vee x_3 \vee x_4$
 - $V_2 = \{a, b, c\}$ 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
- (Optional) At most one VVP per variable

$$\begin{aligned} & (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4) \vee \\ & (\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4) \end{aligned}$$

CSP into SAT (cont.)

Constraint: $C_{V_1V_2} = \{(a, a), (a, b), (b, c), (c, b), (d, a)\}$

1. Way 1: Each inconsistent tuple \rightarrow one disjunctive clause
 - For example: $\neg x_1 \vee \neg x_7$ how many?

2. Way 2:

- a) Consistent tuple \rightarrow conjunction of terms $x_1 \wedge x_5$
- b) Each constraint \rightarrow disjunction of these conjunctions

$$(x_1 \wedge x_5) \vee (x_1 \wedge x_6) \vee (x_2 \wedge x_7) \\ \vee (x_3 \wedge x_6) \vee (x_4 \wedge x_5)$$

\rightarrow transform into conjunctive normal form (CNF)

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