General search strategies: Look-ahead

Chapter 5

The search space

- A tree of all partial solutions
- A partial solution: (a1,…,aj) satisfying all relevant constraints
- The size of the underlying search space depends on:
  - Variable ordering
  - Level of consistency posesed by the problem

Search space and the effect of ordering

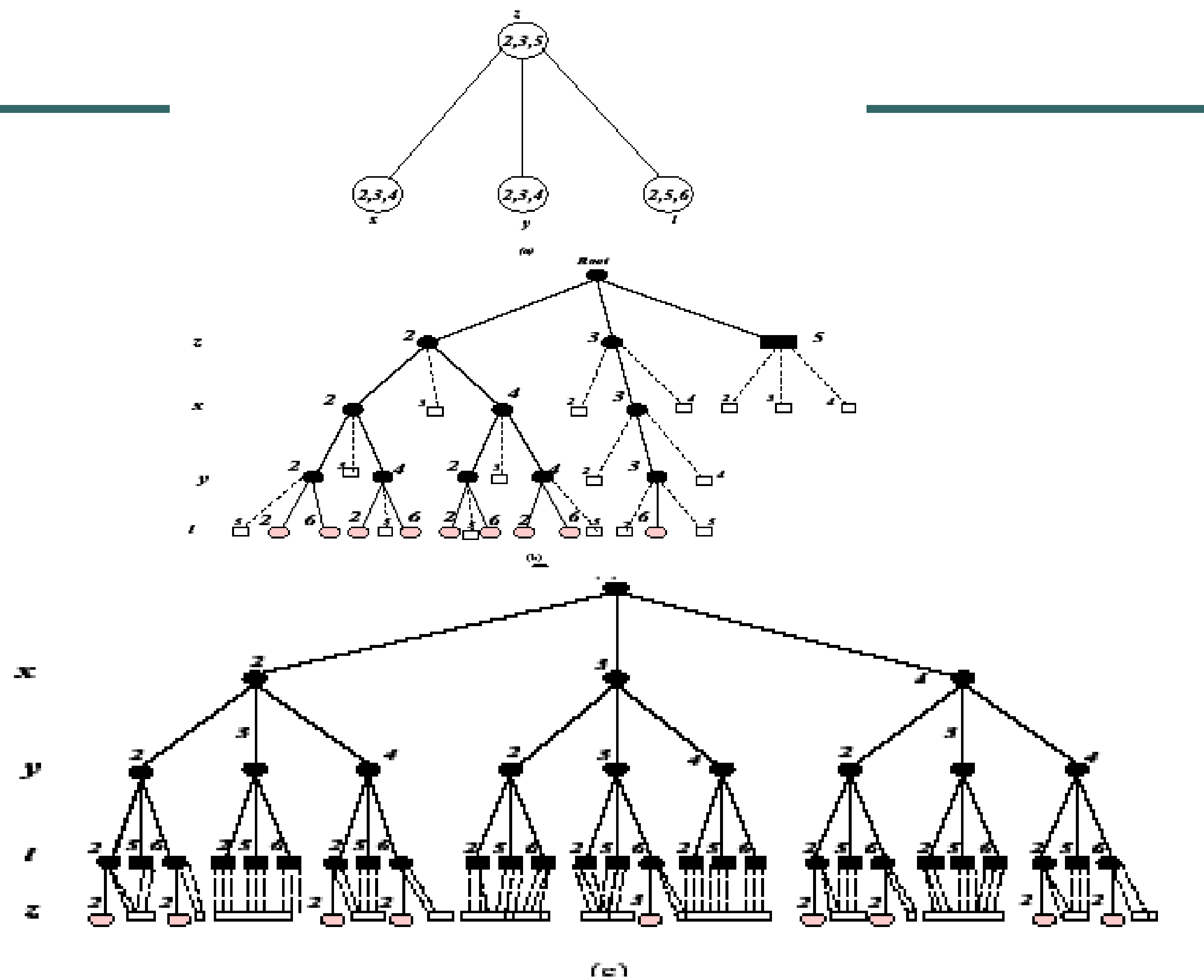
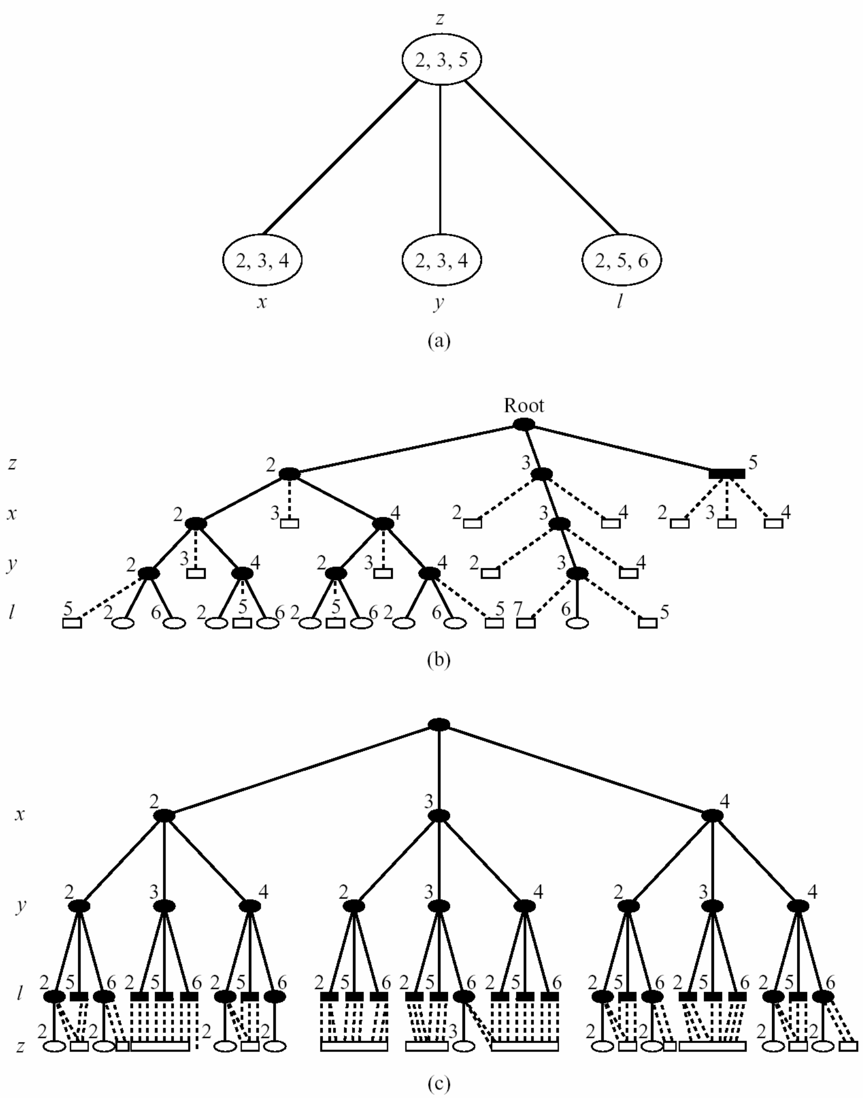


Figure 5.1: (a) A constraint graph, (b) its search space along ordering $d_1 = (z, x, y, l)$, and (c) its search space along ordering $d_2 = (x, y, l, z)$. Hollow nodes and bars in the search space graphs represent illegal states that may be considered, but will be rejected. Numbers next to the nodes represent value assignments.

Search space and the effect of ordering



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

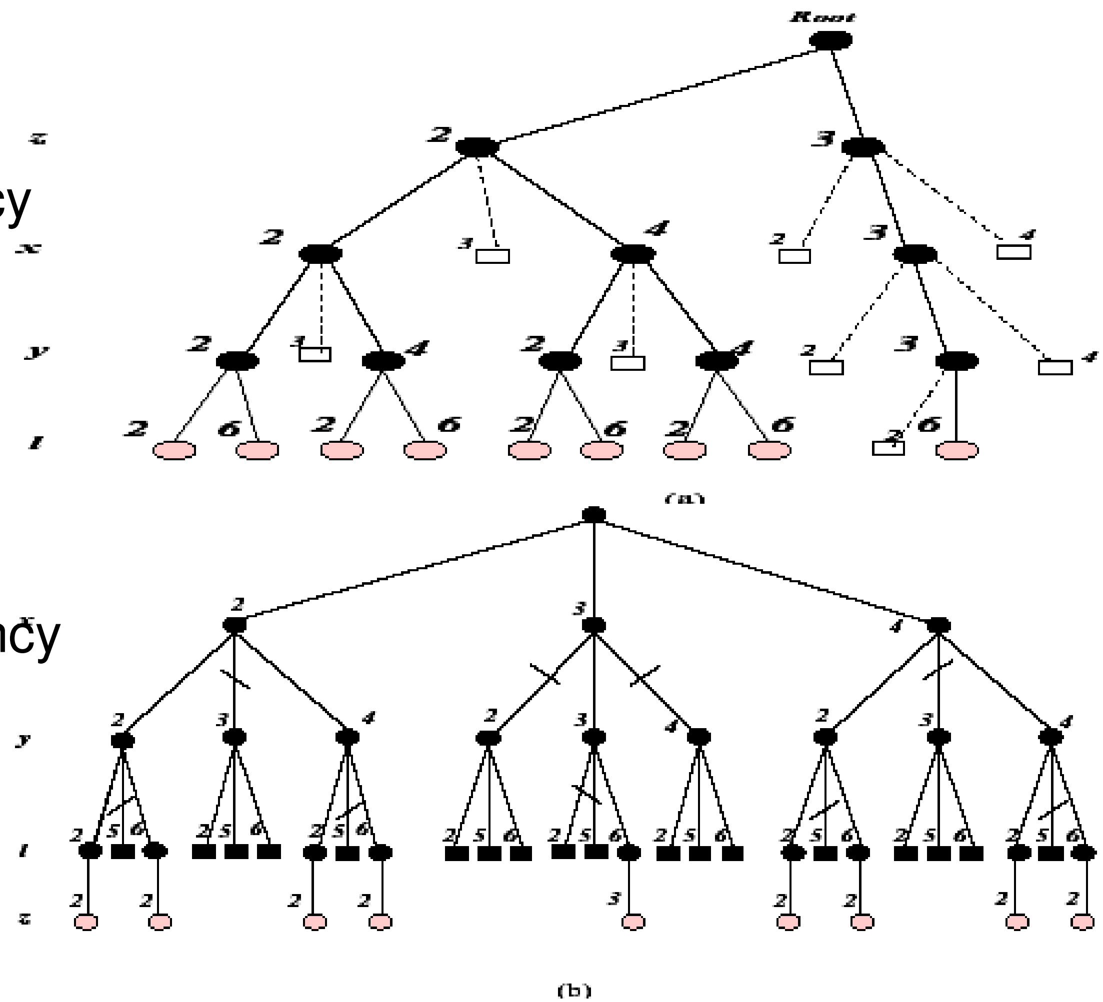
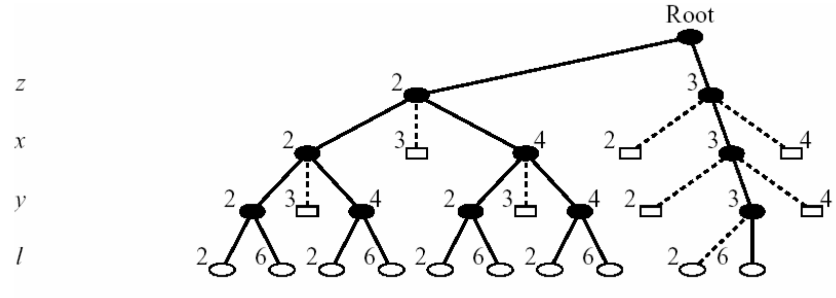


Figure 5.2: (a) Search space Example 5.1.1 with ordering d_1 after arc-consistency. (b) Search space for ordering d_2 with reduction effects from enforcing path-consistency marked with slashes.

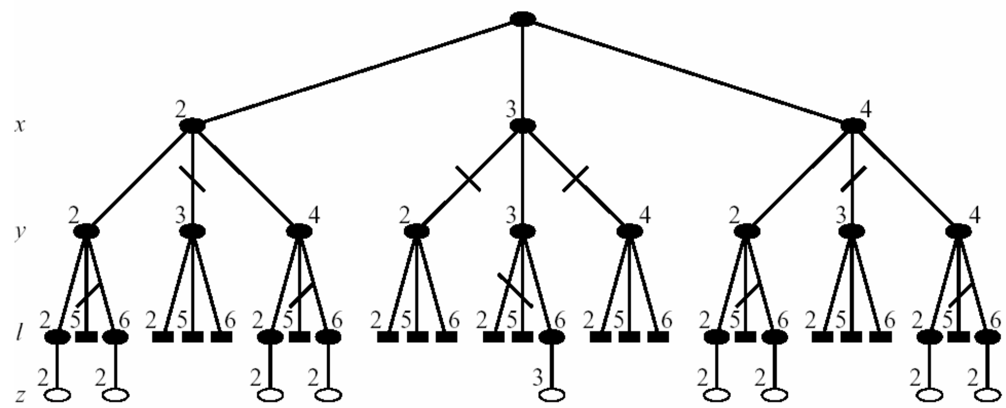
Dependency on consistency level

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



(a)

- After path-consistency
  - R'_zx
  - R'_zy
  - R'_zl
  - R'_xy
  - R'_xl
  - R'_yl



(b)

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 $\mathcal{R}$.* $\square$

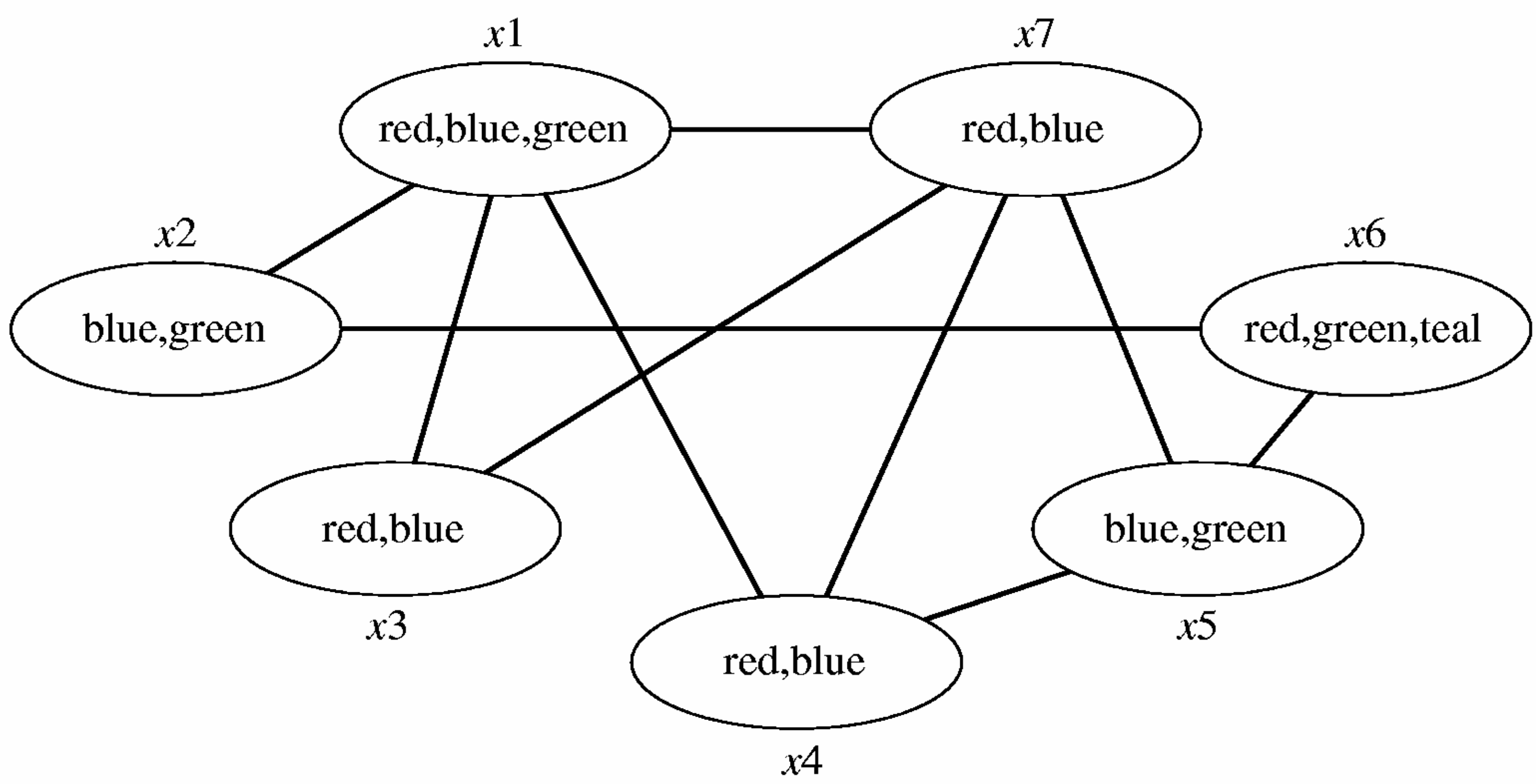
Cost of node's expansion

- Number of consistency checks for toy problem:
  - For d1: 19 for R, 43 for R'
  - For d2: 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.*

A graph coloring problem



Backtracking search

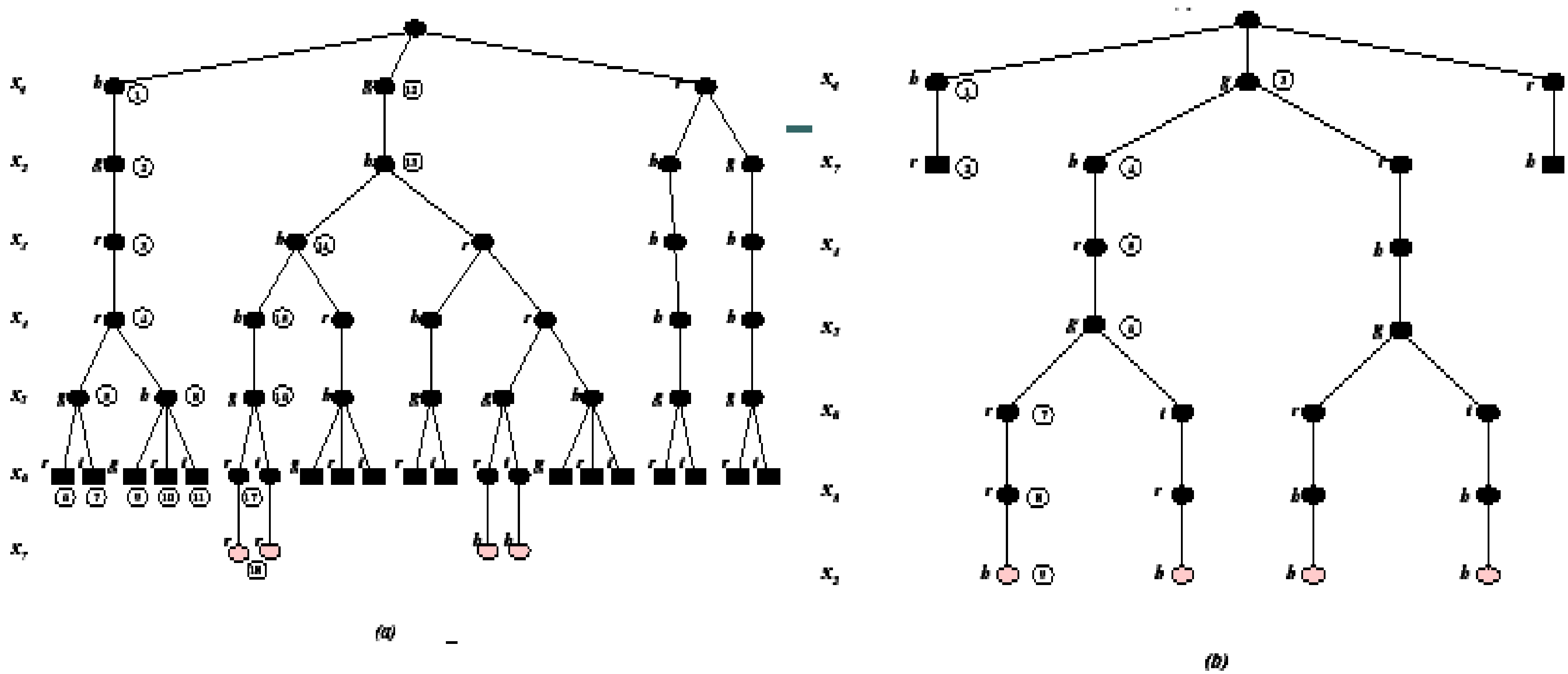
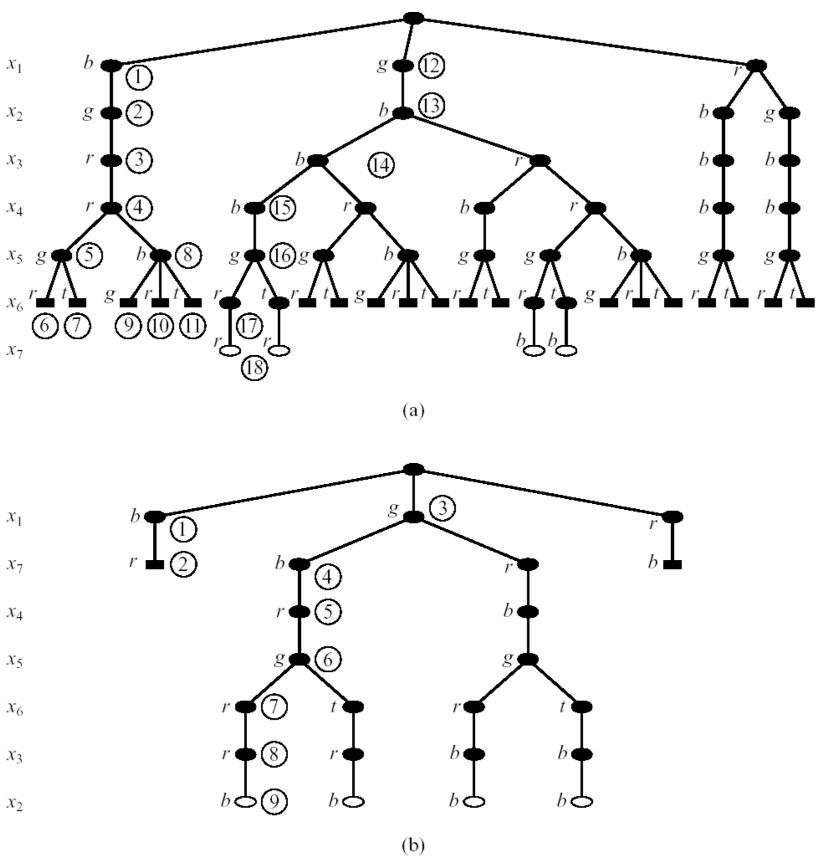


Figure 5.5: Backtracking search for the orderings (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and (b) $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$ on the example instance in Figure 5.3. Intermediate states are indicated by filled ovals, dead-ends by filled rectangles, and solutions by grey ovals. The colors are considered in order (*blue, green, red, teal*), and are denoted by first letters. Bold lines represent the portion of the search space explored by backtracking when stopping after the first solution. Circled numbers indicate the order in which nodes are expanded.

Backtracking search



(a)

(b)

Backtracking

```
procedure BACKTRACKING
Input: A constraint network P = (X, D, C).
Output: Either a solution, or notification that the network is inconsistent.

   i ← 1                        (initialize variable counter)
   D'_i ← D_i                   (copy domain)
   while 1 ≤ i ≤ n
      instantiate x_i ← SELECTVALUE
      if x_i is null            (no value was returned)
         i ← i − 1              (backtrack)
      else
         i ← i + 1              (step forward)
         D'_i ← D_i
   end while
   if i = 0
      return "inconsistent"
   else
      return instantiated values of {x_1, ..., x_n}
end procedure

subprocedure SELECTVALUE   (return a value in D'_i consistent with a⃗_{i−1})

   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 5.4: The backtracking algorithm.

- 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)

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 backtacking

Look-ahead: value orderings

- Intuition:
  - Choose value least likely to yield a dead-end
  - Approach: apply 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 ← D_i for 1 ≤ i ≤ n          (copy all domains)
    i ← 1                             (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE-XXX
        if x_i is null                (no value was returned)
            i ← i − 1                 (backtrack)
            reset each D'_k, k > i, to its value before x_i was last instantiated
        else
            i ← i + 1                 (step forward)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., 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 on graph coloring

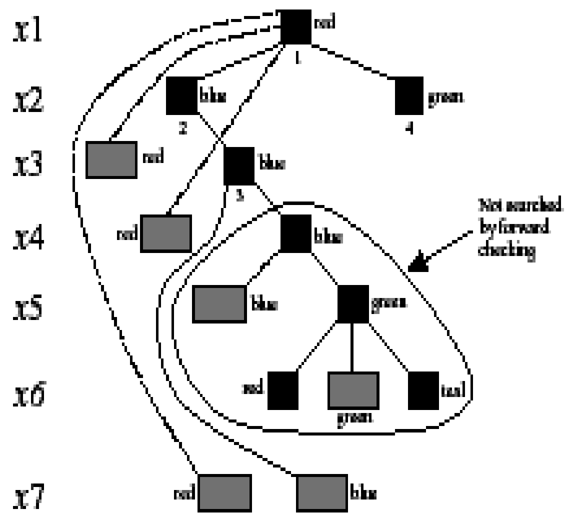
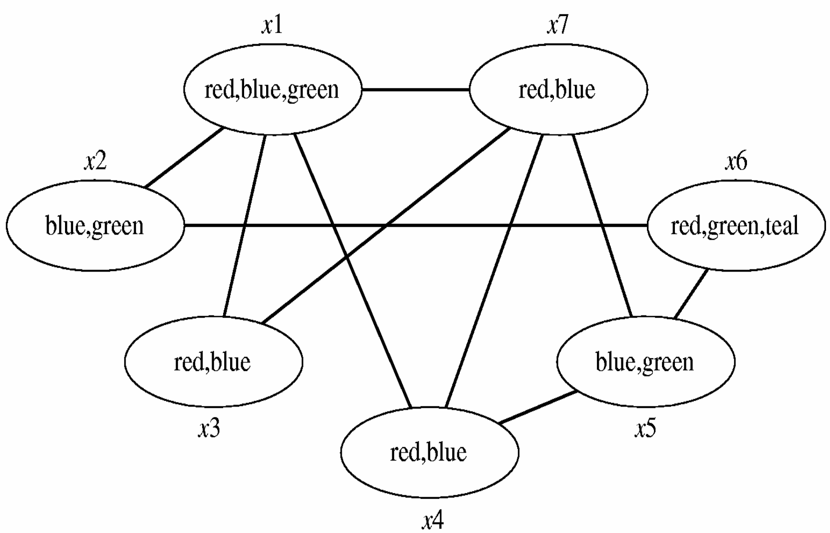
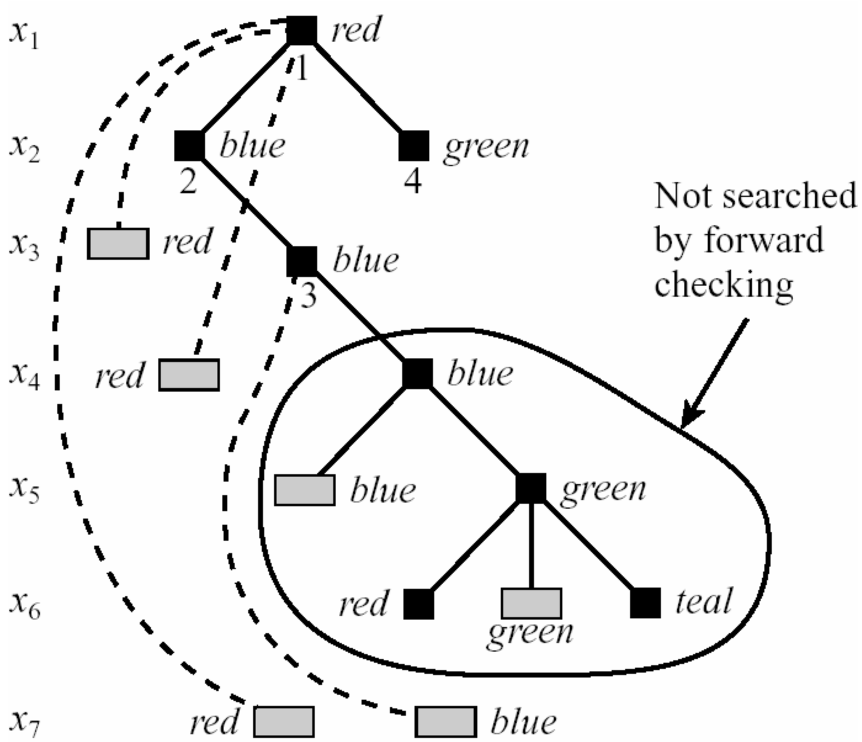


Figure 5.9: Part of the search space explored by forward-checking in the example in Figure 5.3. Only the search space below $x_1 = red$ and $x_2 = blue$ is drawn. Dotted lines connect values with future values that are filtered out.

Example 5.3.2 Consider again the coloring problem in Figure 5.3. In this problem, instantiating $x_1 = red$ reduces the domains of x_3, x_4 and x_7. Instantiating $x_2 = blue$ does not affect any future variable. The domain of x_3 includes only blue , and selecting that value causes the domain of x_7 to be empty, so $x_3 = blue$ is rejected and x_3 is determined to be a deadend. See Figure 5.9. □

Forward-checking example





Forward-checking

```
procedure selectValue-forward-checking
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        empty-domain ← false
        for all k, i < k ≤ n
            for all values b in D'_k
                if not consistent(a⃗_{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 ← true
        if empty-domain                (don't select a)
            reset each D'_k, i < k ≤ 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?

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

- Perform arc-consistency ina binary search tree: Given a domain X={1,2,3,4} the algorithm assig X=1 (and apply arc-consstency) and if x=1 is pruned, it
- Applies arc-consistency to X={2,3,4}
- If no inconsistency a new variable is selected (not necessarily X)

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

```
subprocedure SELECTVALUE-ARC-CONSISTENCY

  while D'_i is not empty
    select an arbitrary element a ∈ D'_i, and remove a from D'_i
    repeat
    removed-value ← false
      for all j, i < j ≤ n
        for all k, i < k ≤ n
          for each value b in D'_j
            if there is no value c ∈ D'_k such that
                CONSISTENT(a⃗_{i-1}, x_i = a, x_j = b, x_k = c)
              remove b from D'_j
              removed-value ← 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 ≤ n, to value before a was selected
    else
      return a
  end while
  return null                  (no consistent value)
end procedure
```

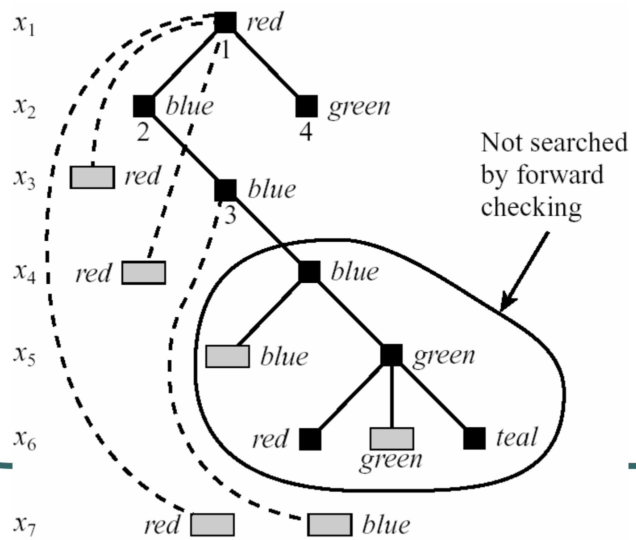
Figure 5.10: The SELECTVALUE subprocedure for arc-consistency, based on the AC-1 algorithm.

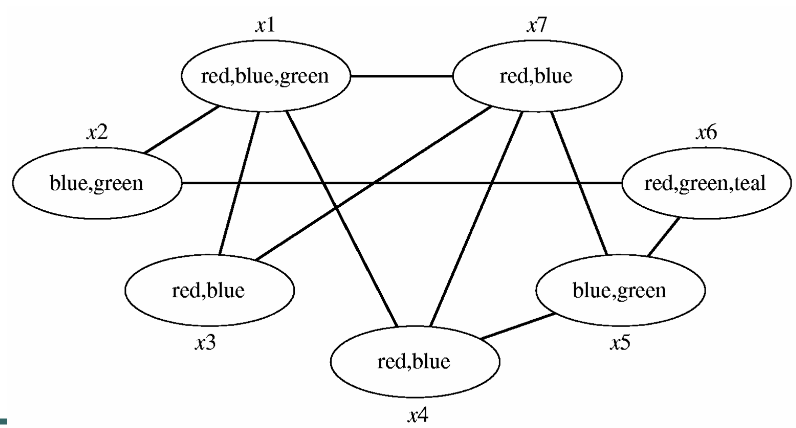
Full and partial look-ahead

- 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 Conside 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 $x_1 = red$. 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 $x_1 = blue$ 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 $x_1 = green$ will be expanded. □





Dynamic value ordering (LVO)

Use constraint propagation to rank order the promise in non-rejected values.

Example: look-ahead value ordering (LVO) is based of forward-checking propagation

LVO uses a heuristic measure to transform this information to ranking of the values

Empirical work shows the approach is cost-effective only for large and hard problems.

MC (min-conflict), MD (min-domain) ES (expected solutions). MC was best empirically (Frost and Dechter 1996)

Look-ahead: variable ordering

- Dynamic search rearangement (Bitner and Reingold, 1975)(Purdon,1983):
  - Choose the most constrained variable
  - Intuition: early discovery of dead-ends

DVO

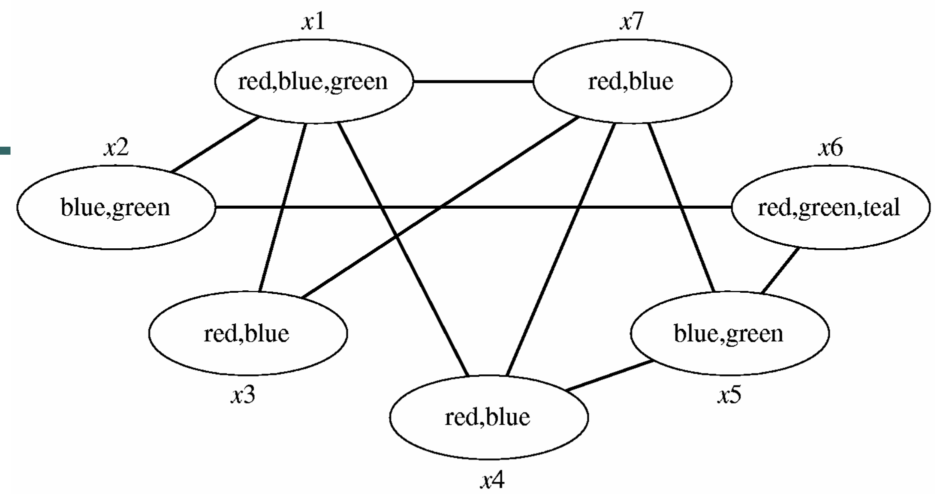
subprocedure SELECTVARIABLE

$m \leftarrow \min_{i \le j \le n} |D'_j|$ (find size of smallest future domain)
select an arbitrary uninstantiated variable x_k such that $|D'_k| = m$
rearrange future variables so that x_k is the ith variable

end subprocedure

Figure 5.11: The subprocedure SELECTVARIABLE, which employs a heuristic based the D' sets to choose the next variable to be instantiated.

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 $< x_7, blue >$. 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. □

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 $x_i \leftarrow$ 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 $\{x_1, \ldots, x_n\}$
end procedure

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

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 $O(e_d k) \Rightarrow O(ek)$

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* $< x, a >$ *in* C. C *is arc-consistent if for each variable* x, *in its scope and each of its values,* $a \in D_x$, $< x, a >$ *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. $\quad Q$ = some unassigned variable;
6. $\quad$ return(**DPLL**($\varphi \wedge Q$) $\vee$
$\qquad\qquad$ **DPLL**($\varphi \wedge \neg Q$))

Figure 5.13: The DPLL Procedure

Example of DPLL

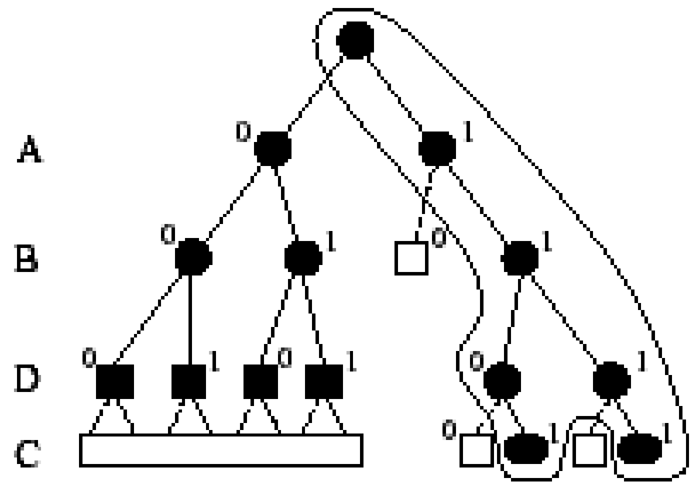


Figure 5.14: A backtracking search tree along the variables A, B, D, C for a cnf theory $\varphi = \{(\neg A \vee B), (\neg C \vee A), (A \vee B \vee D), C\}$. Hollow nodes and bars in the search tree represent illegal states, triangles represent solutions. The enclosed area corresponds to DPLL with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