Constraint satisfaction search
AND/OR search

COMPSCI 276, Spring 2018

Class 5: Rina Dechter

*(Reading: **Constraint book chapters 5,6 Dechter2 chapter 6**)*

Outline: Search in CSPs

- Improving search by bounded-inference (constraint propagation) in looking ahead
- Improving search by looking-back
- The alternative AND/OR search space

Outline: Search in CSPs

- Improving search by bounded-inference (constraint propagation) in looking ahead
- Improving search by looking-back
- The alternative AND/OR search space

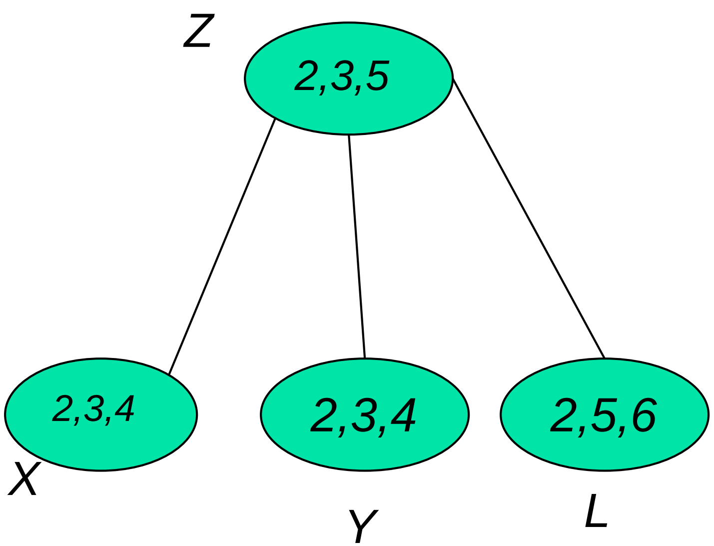
What if the constraint network is not backtrack-free?

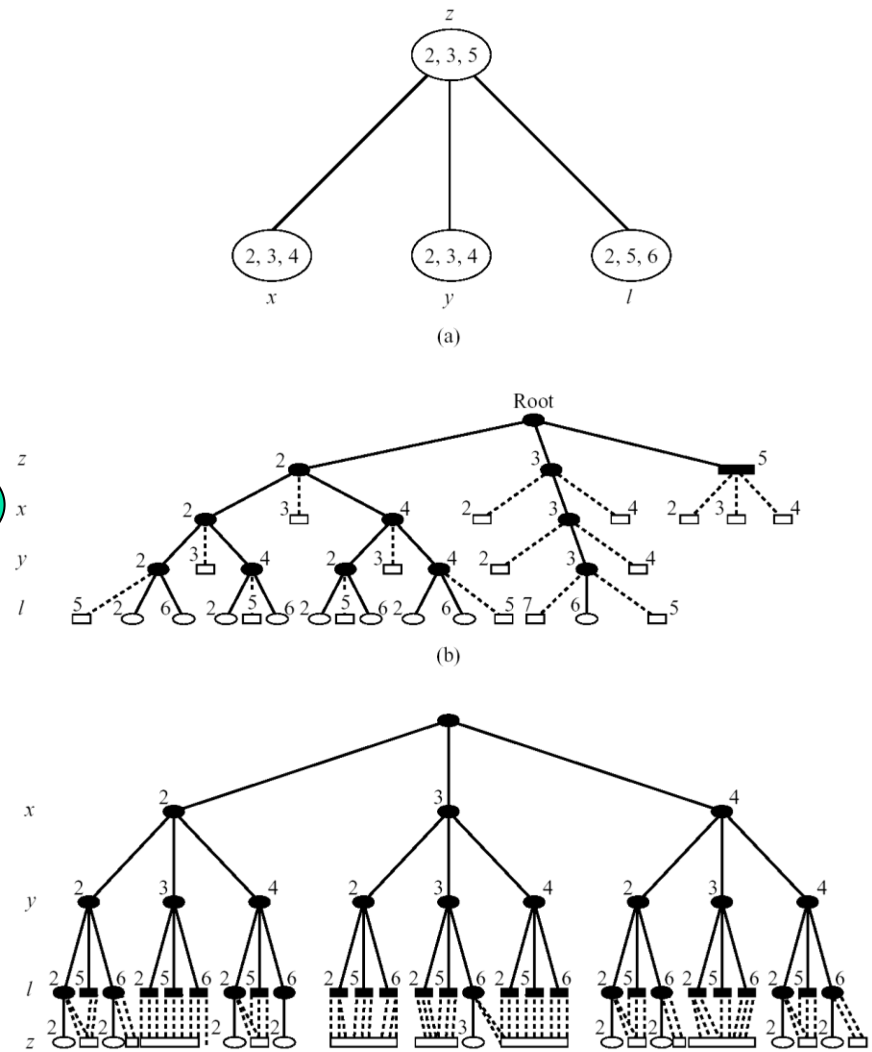
- Backtrack-free in general is too costly, so what to do?
- Search?
- What is the search space?
- How to search it? Breadth-first? Depth-first?

The search space for a CN

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

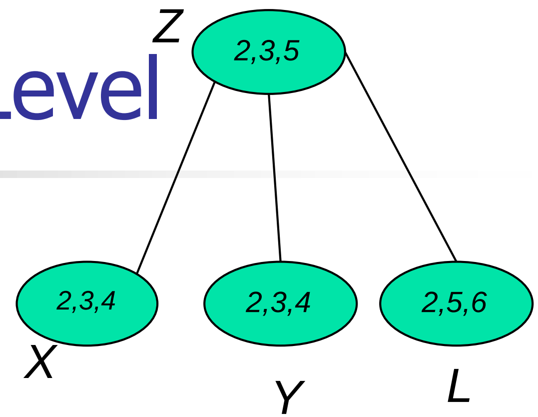
The Effect of Variable Ordering





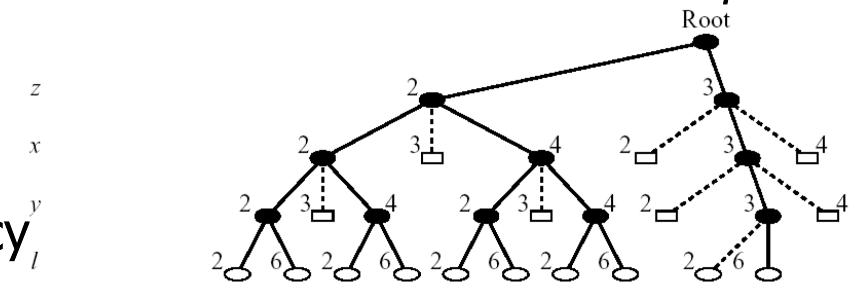
(a) (b) (c)

The Effect of Consistency Level

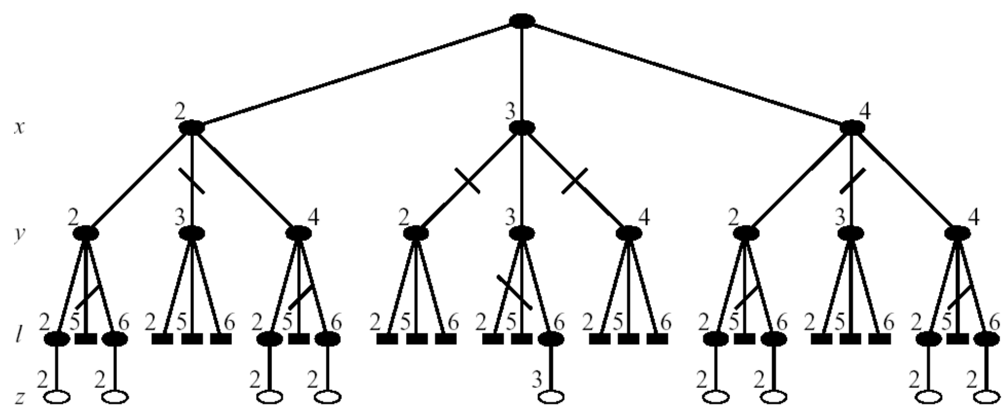


- After arc-consistency z=5 and l=5 are removed

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

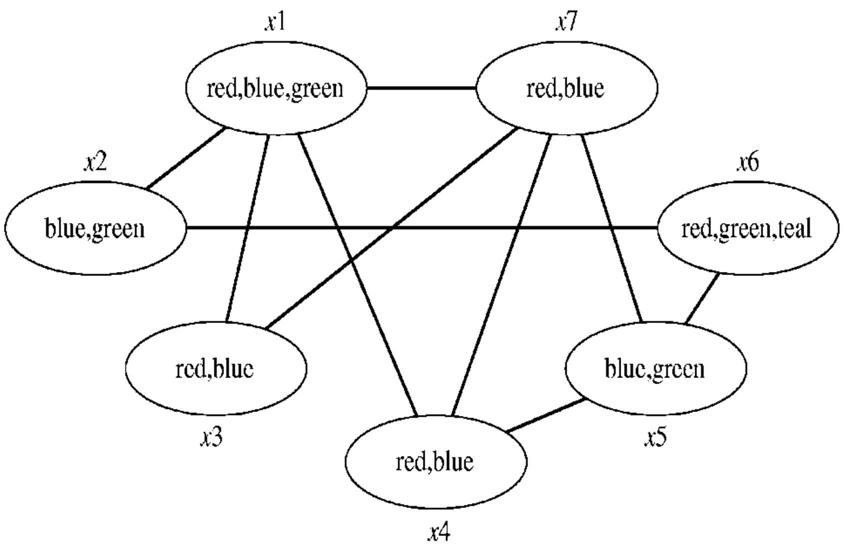


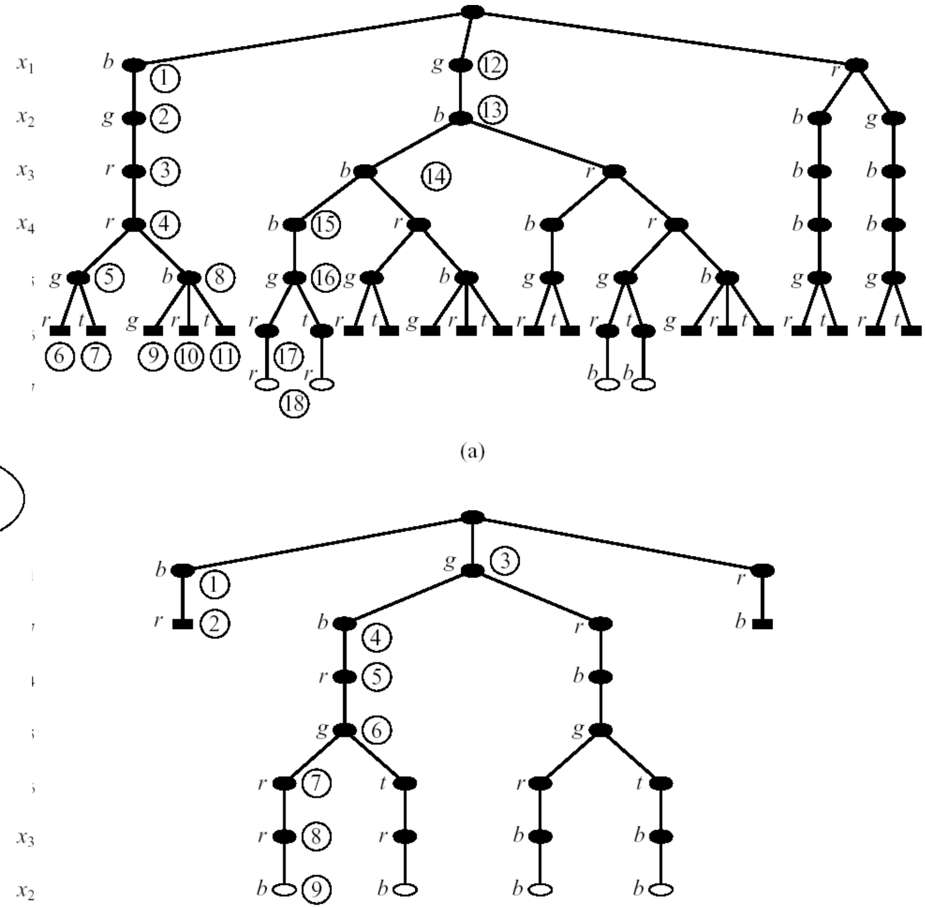
(a)



(b)

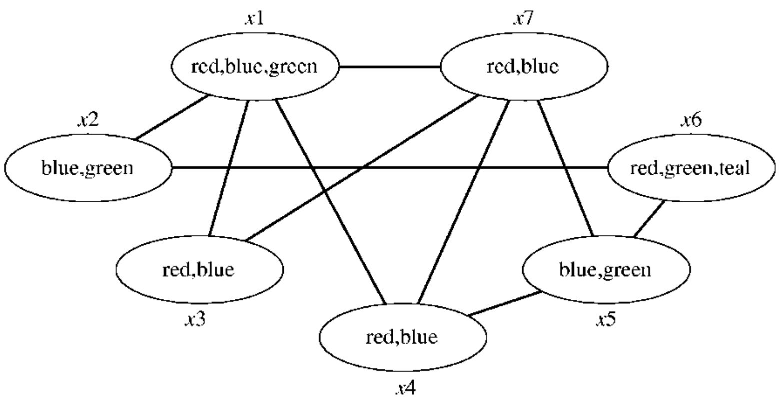
Backtracking Search for a Solution

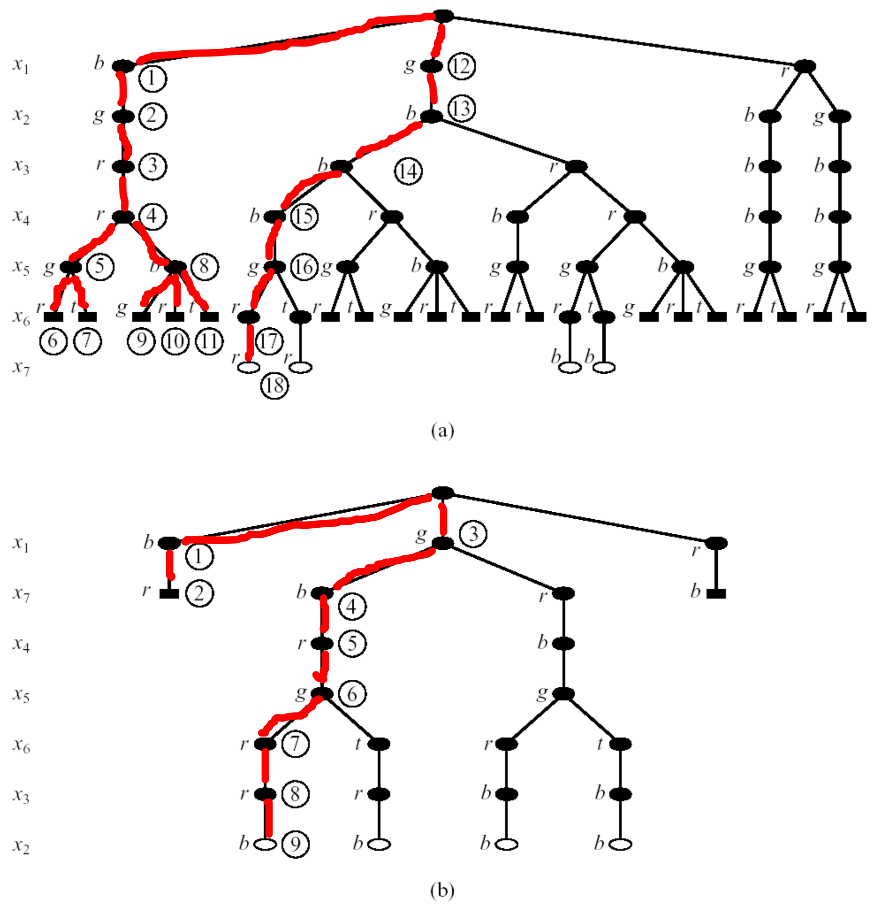




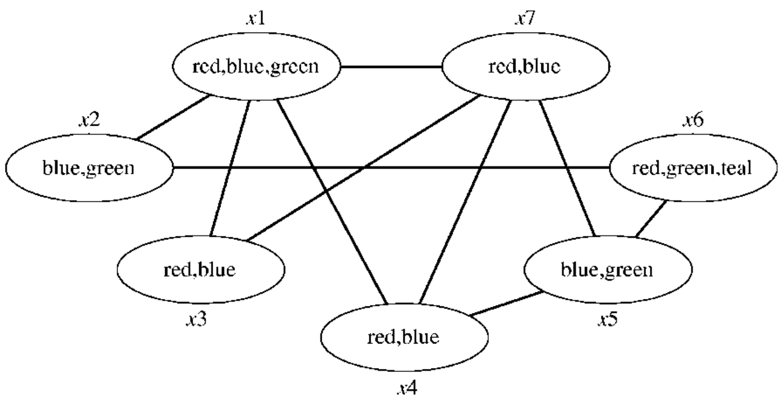
(b)

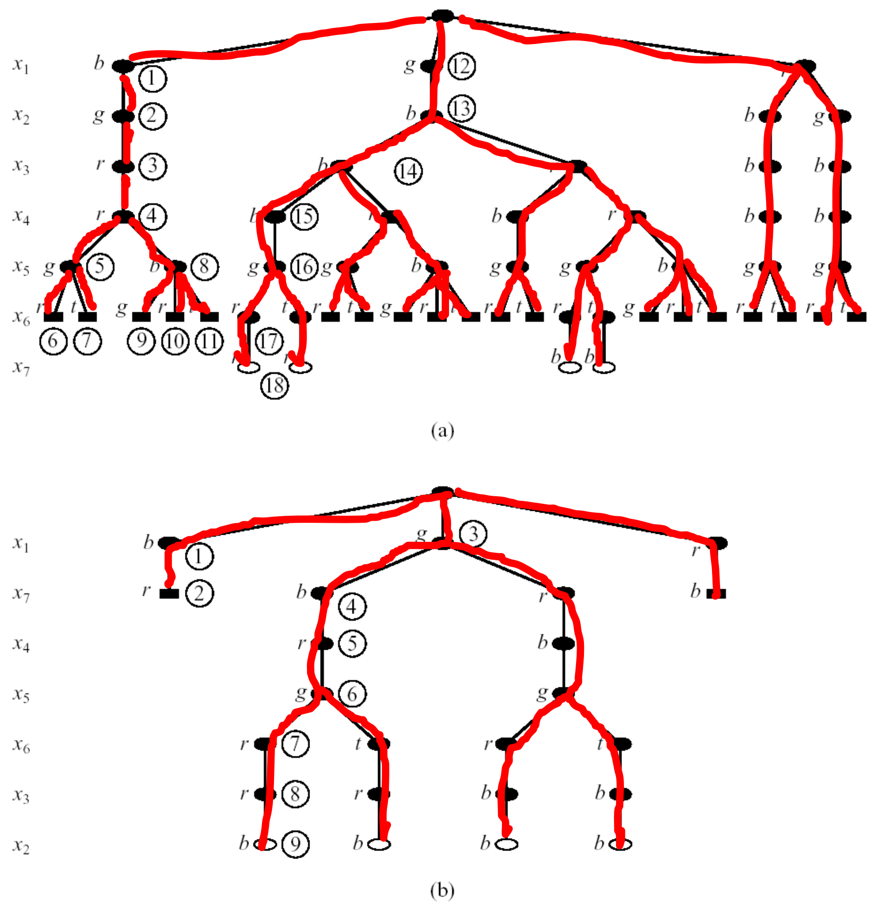
Backtracking Search for a Solution



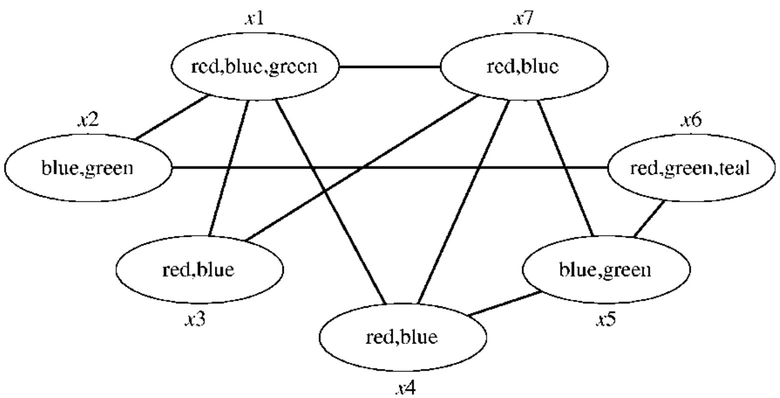


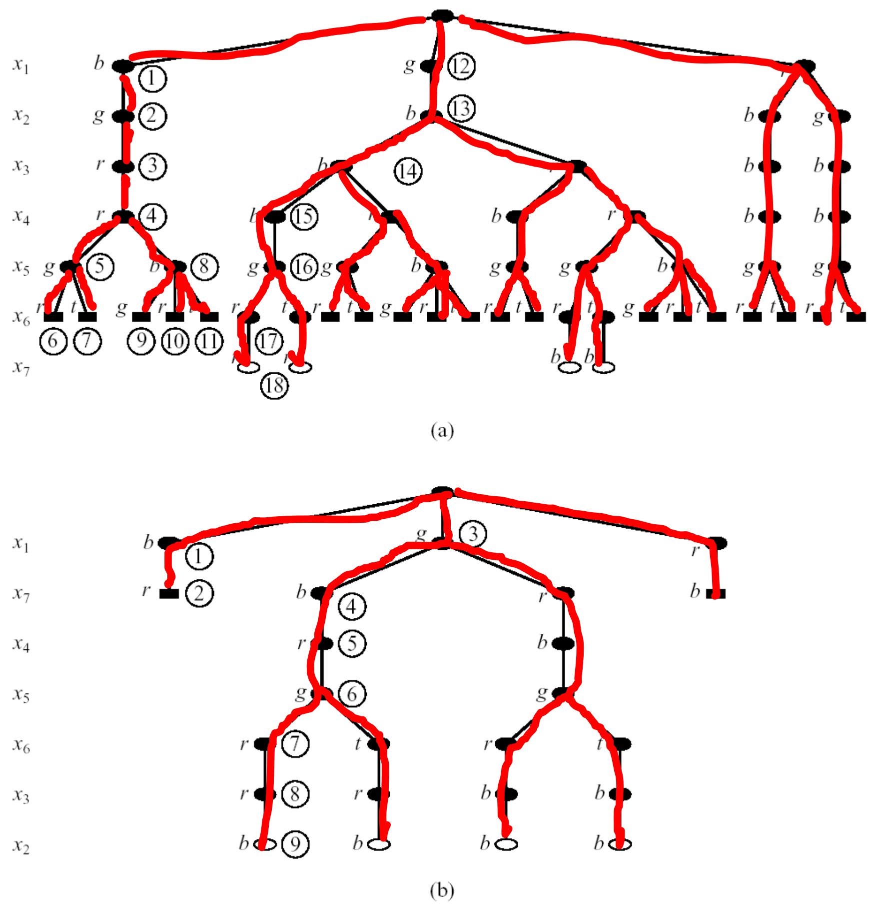
Backtracking Search for All Solutions





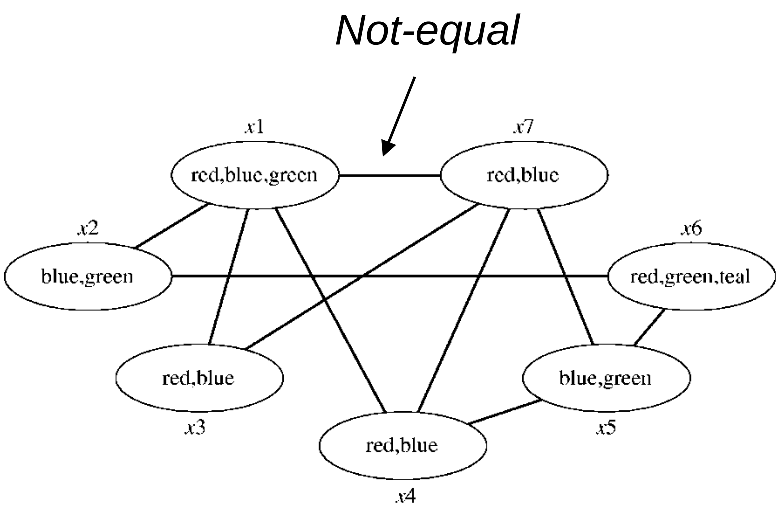
Backtracking search for *all* solutions

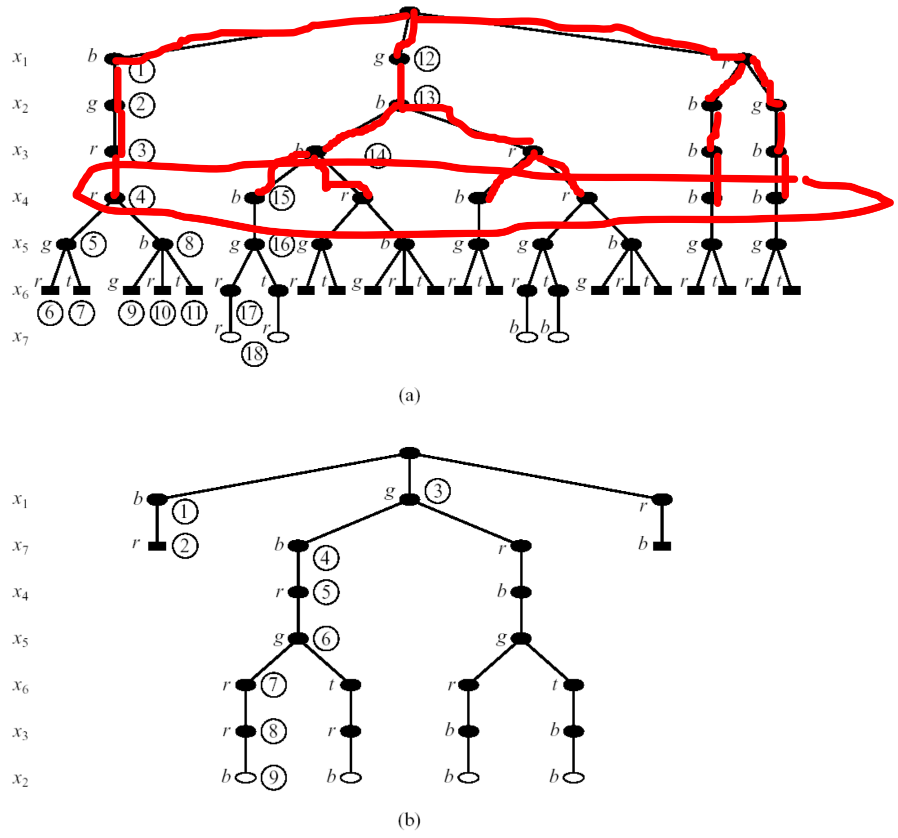




For all tasks
Time: O(exp(n))
Space: linear

Traversing Breadth-First (BFS)?





BFS space is exp(n) while no Time gain → use DFS

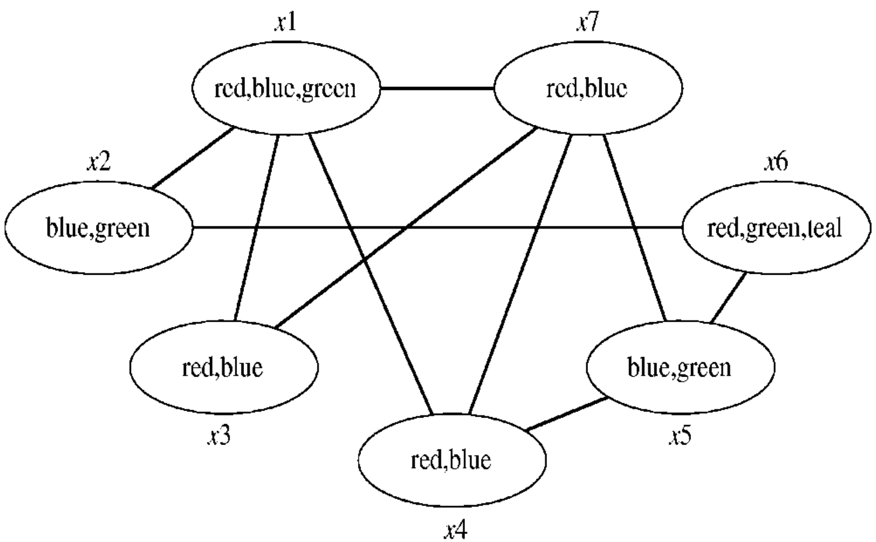
Improving backtracking

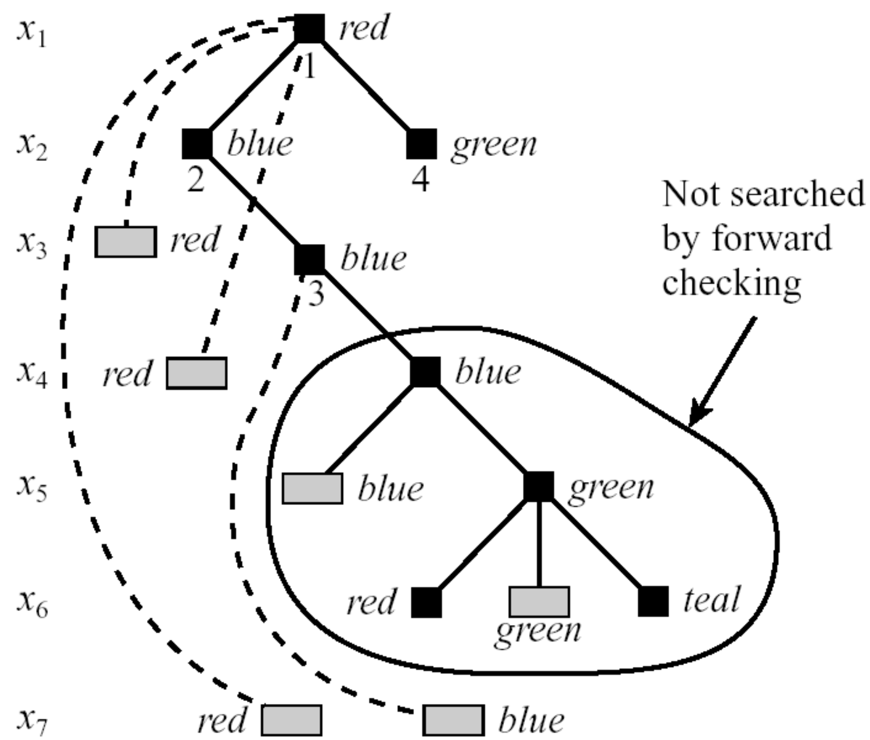
- Before search: (reducing the search space)
  - Arc-consistency, path-consistency
  - Variable ordering (fixed)
- During search:
  - Look-ahead schemes:
    - value ordering,
    - variable ordering (if not fixed)
  - Look-back schemes:
    - Backjump
    - Constraint recording
    - Dependency-directed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

Look-Ahead: Value Orderings

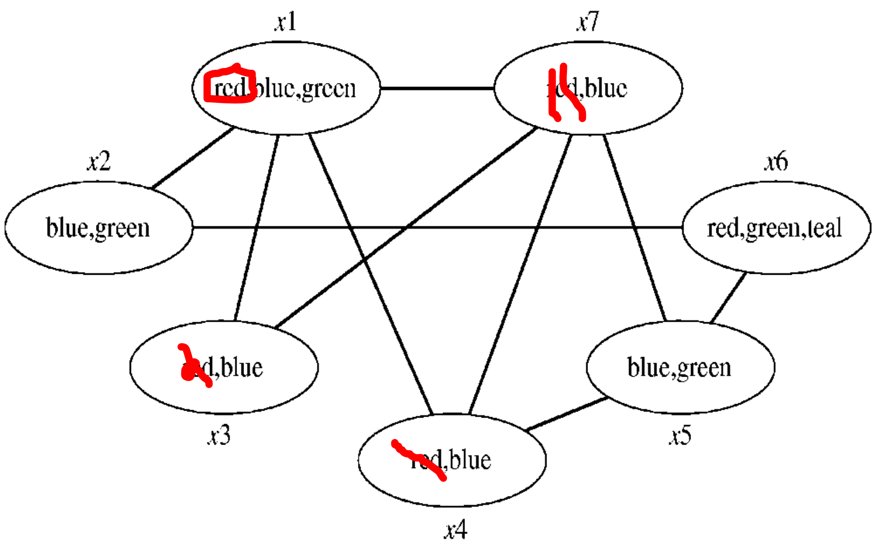
- Intuition:
  - Choose value least likely to yield a dead-end
  - Approach: apply constraint propagation at each node in the search tree
- Forward-checking
  - (check each unassigned variable separately
- Maintaining arc-consistency (MAC)
  - (apply full arc-consistency)
- Full look-ahead
  - One pass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

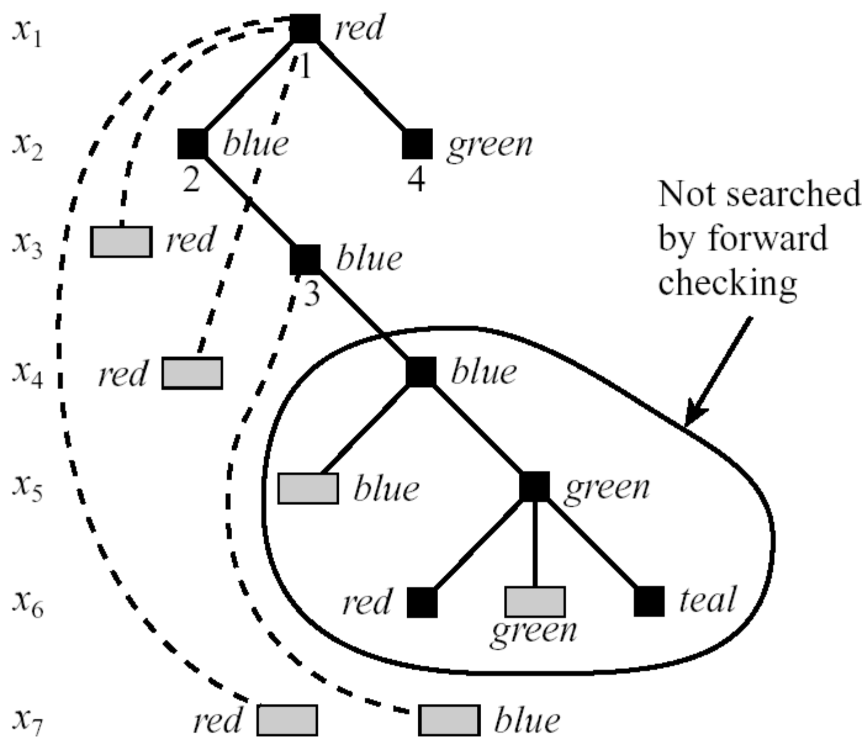
Forward-Checking for Value Ordering





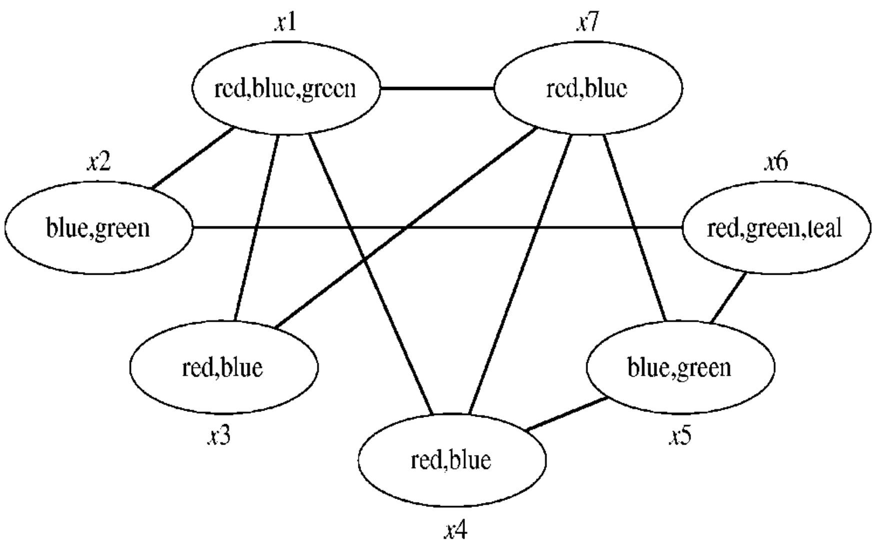
Forward-Checking for Value Ordering

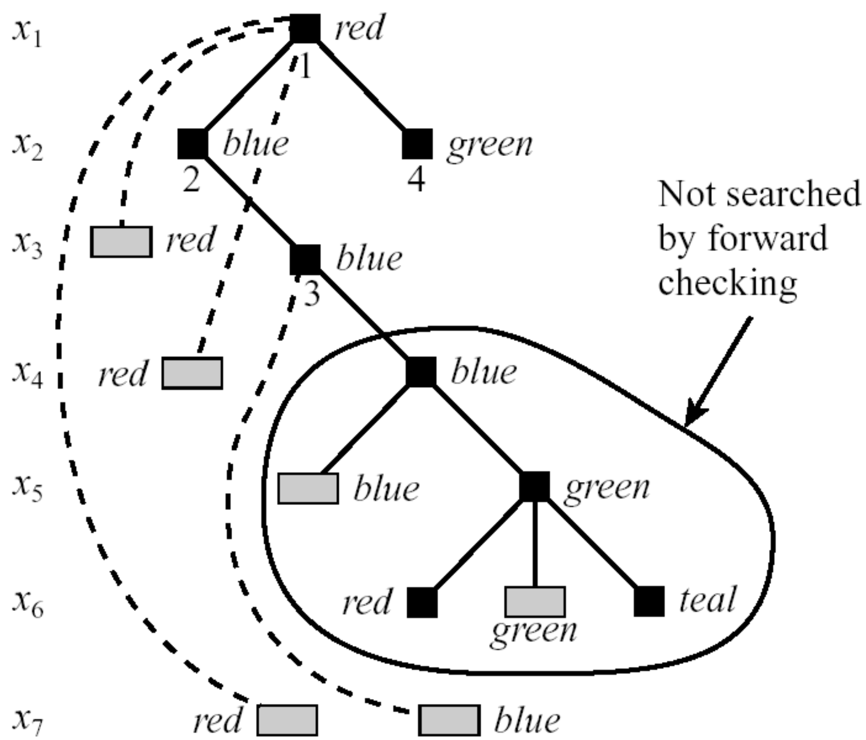




FW overhead: $O(ek^2)$

Forward-Checking, Variable Ordering

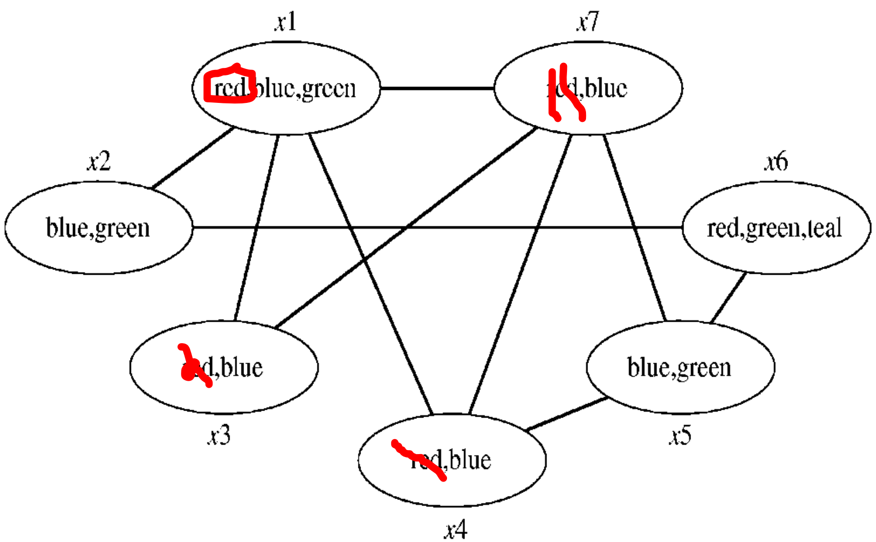


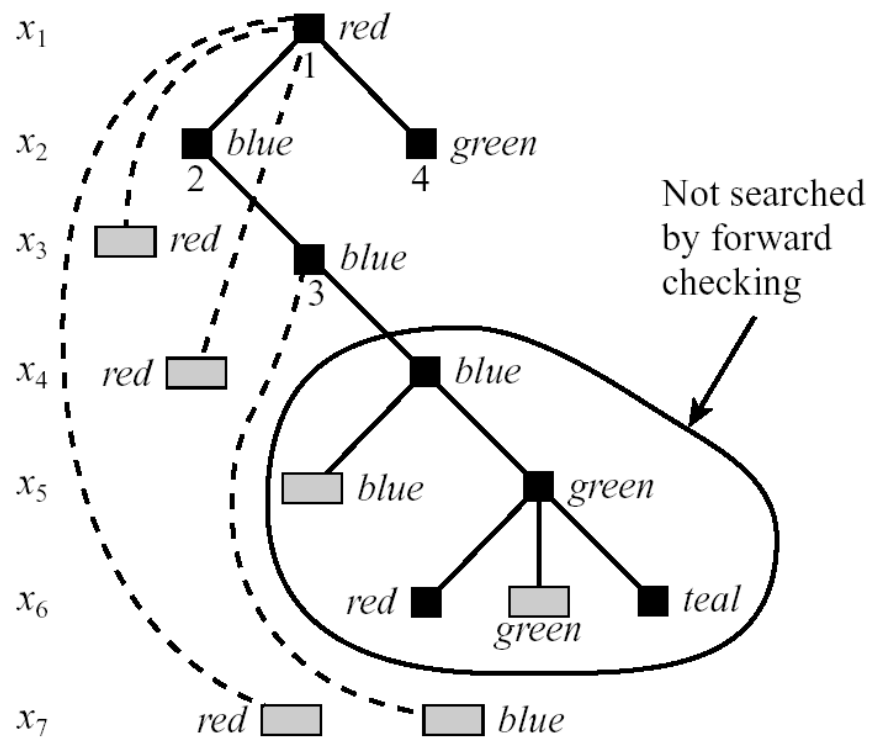


FW overhead: $O(ek^2)$

Forward-Checking, Variable Ordering

After X1 = red choose X3 and not X2

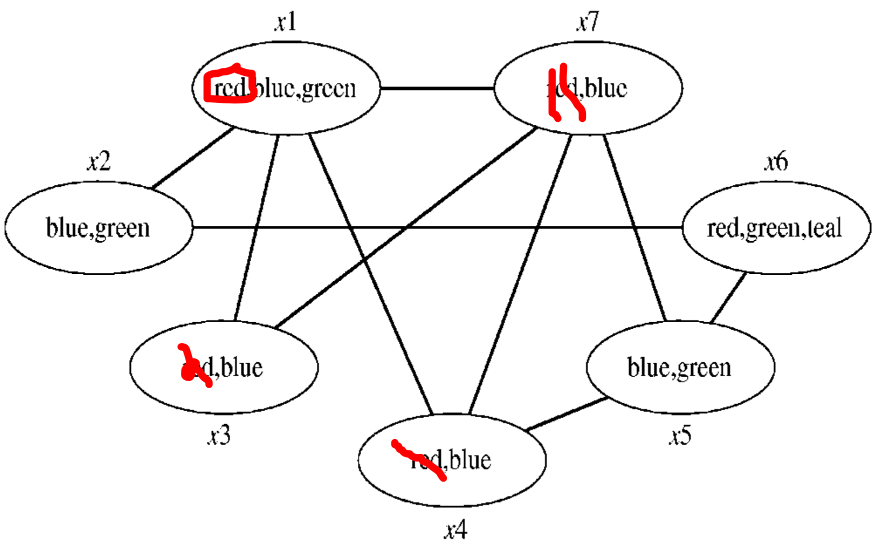


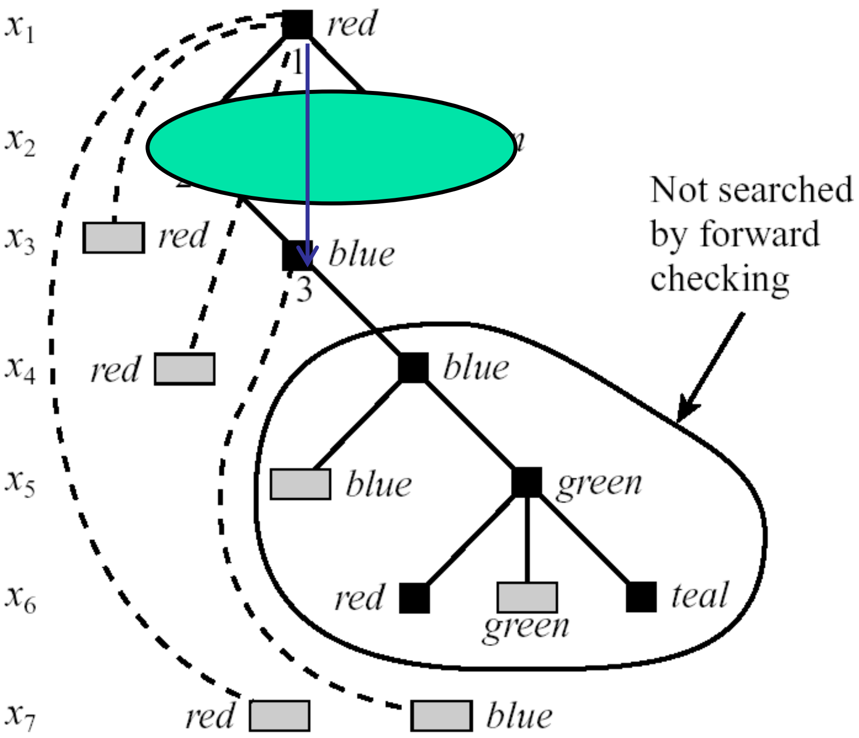


FW overhead: $O(ek^2)$

Forward-Checking, Variable Ordering

After X1 = red choose X3 and not X2

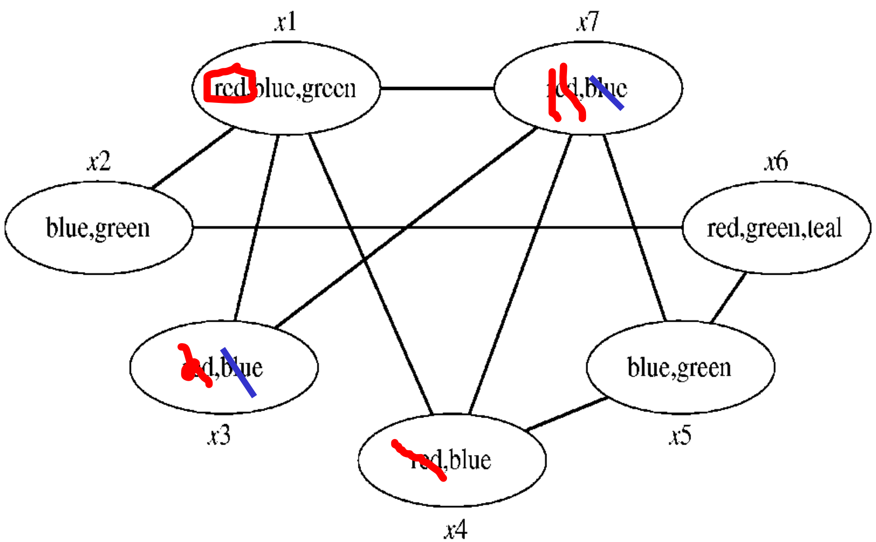


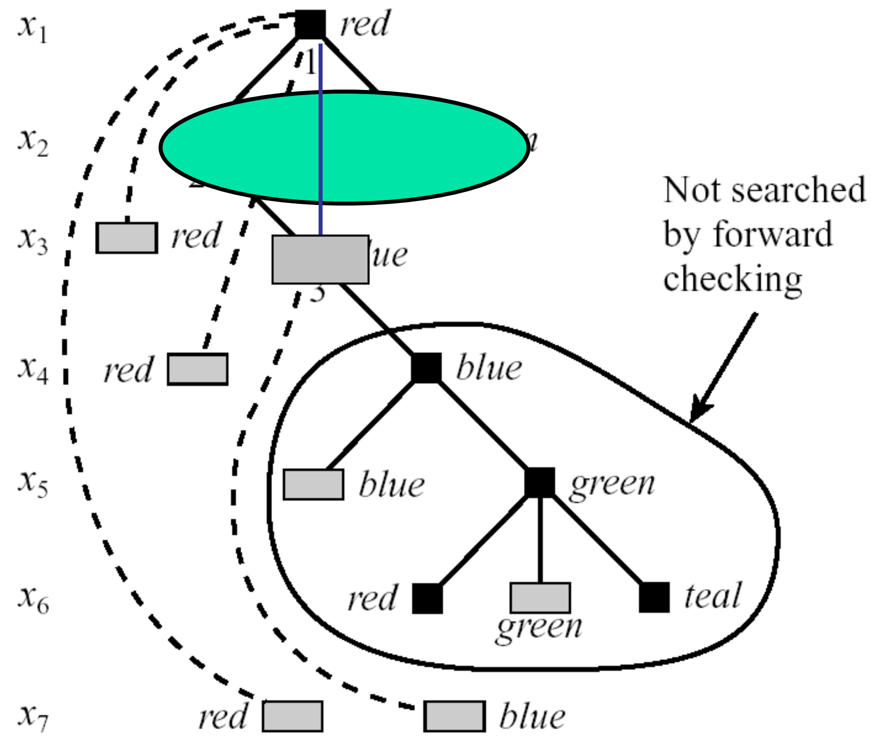


FW overhead: $O(ek^2)$

Forward-Checking, Variable Ordering

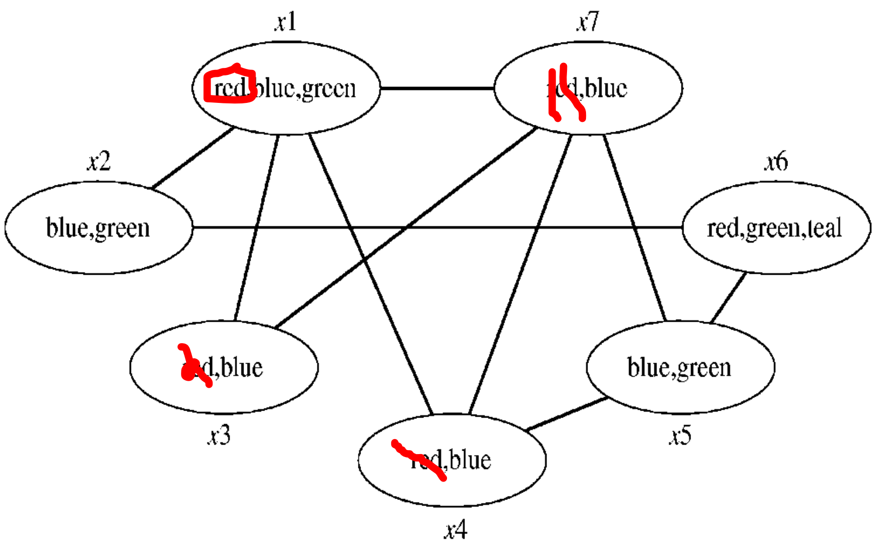
After X1 = red choose X3 and not X2

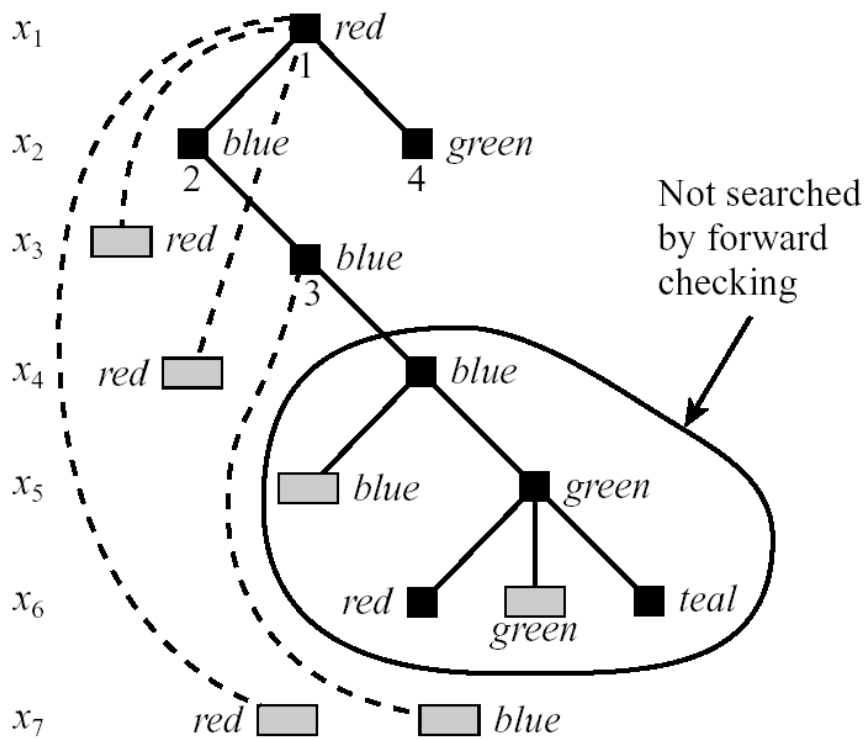




FW overhead: $O(ek^2)$

Arc-consistency for Value Ordering



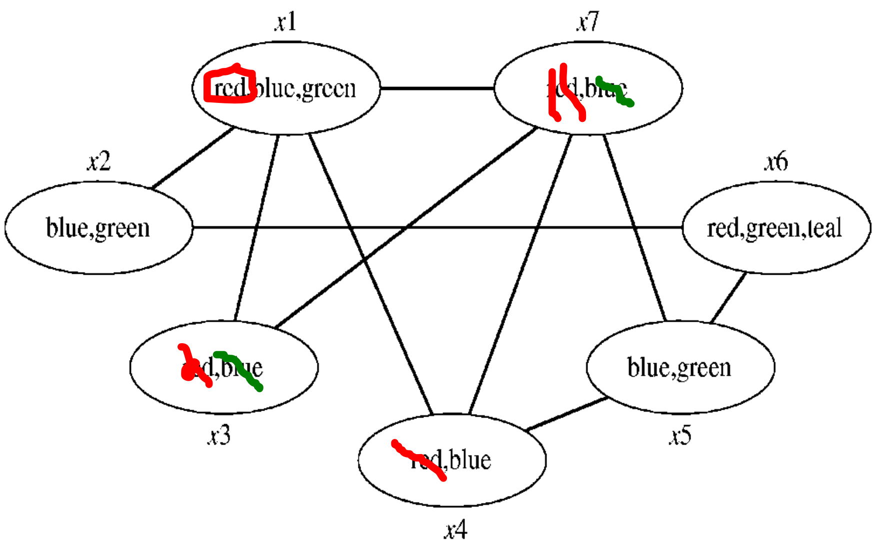


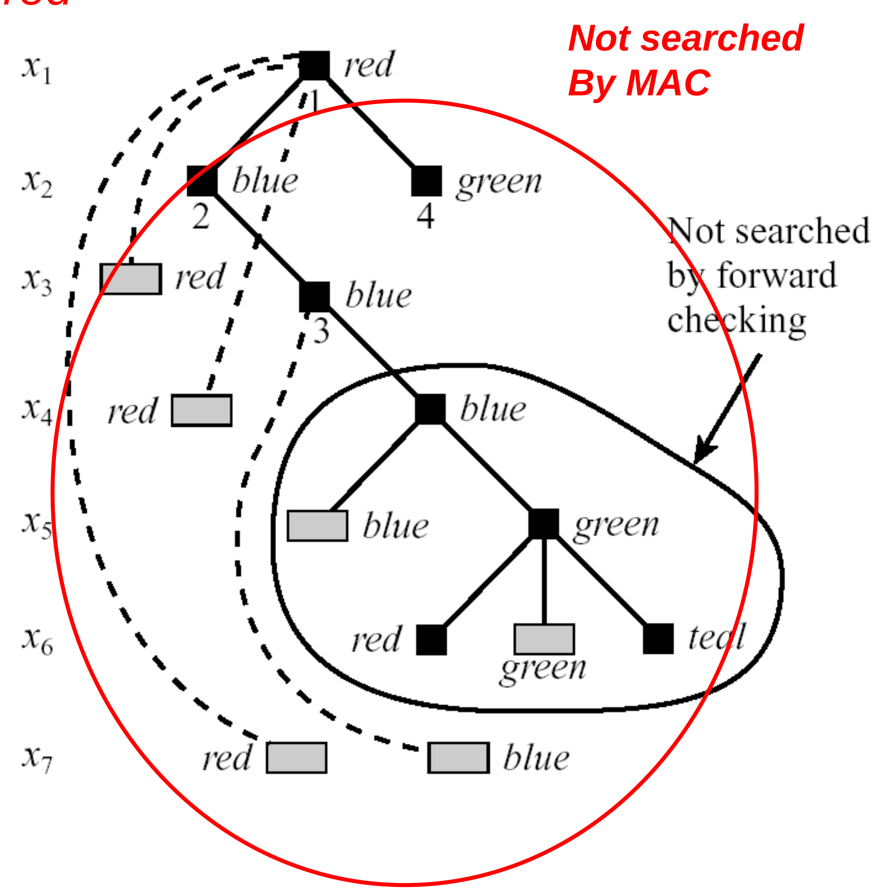
FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$

Arc-Consistency for Value Ordering

Arc-consistency prunes x1=red
Prunes the whole tree





FW overhead: $O(ek^2)$

MAC overhead: $O(ek^3)$

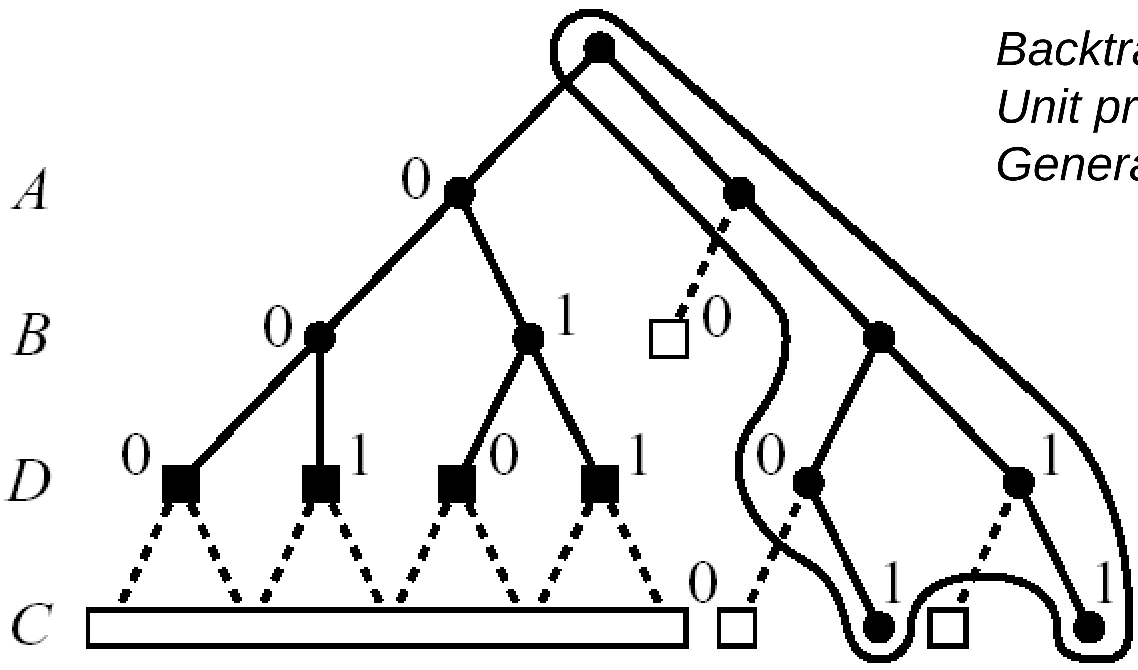
Constraint Programming

- Constraint solving embedded in programming languages
- Allows flexible modeling with algorithms
- Logic programs + forward checking
- Eclipse, ILog, OPL,minizinc
- Using only look-ahead schemes
- Numberjeck (in Python)

Branching-Ahead for SAT: DLL

example: (~AVB)(~CVA)(AVBVD)(C)

(Davis, Logeman and Laveland, 1962)



Backtracking look-ahead with Unit propagation= Generalized arc-consistency

Only enclosed area will be explored with unit-propagation

Outline: Search in CSPs

- Improving search by bounded-inference in branching ahead
- Improving search by looking-back
- The alternative AND/OR search space

Look-back: Backjumping / Learning

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

Look-Back: Backjumping

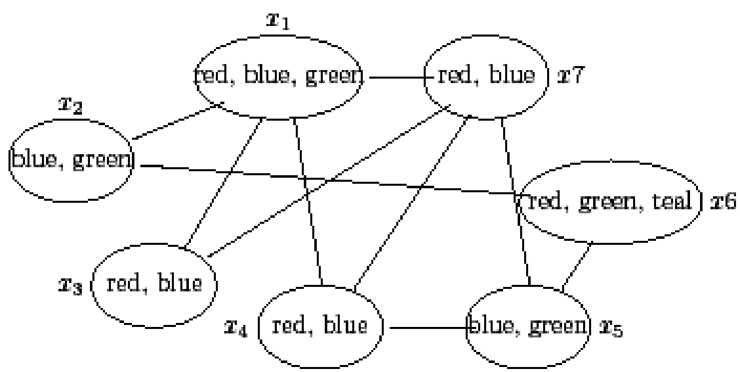
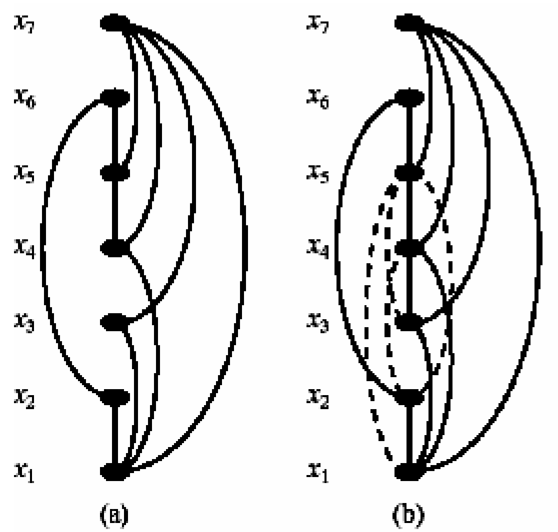


Figure 6.1: A modified coloring problem.

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



Jumps At Leaf Dead-Ends (Gascnnig-style 1977)

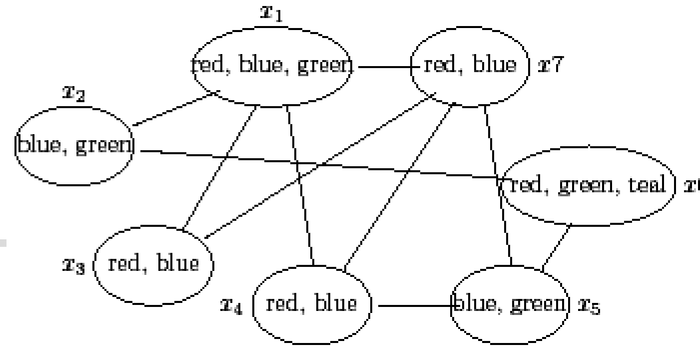
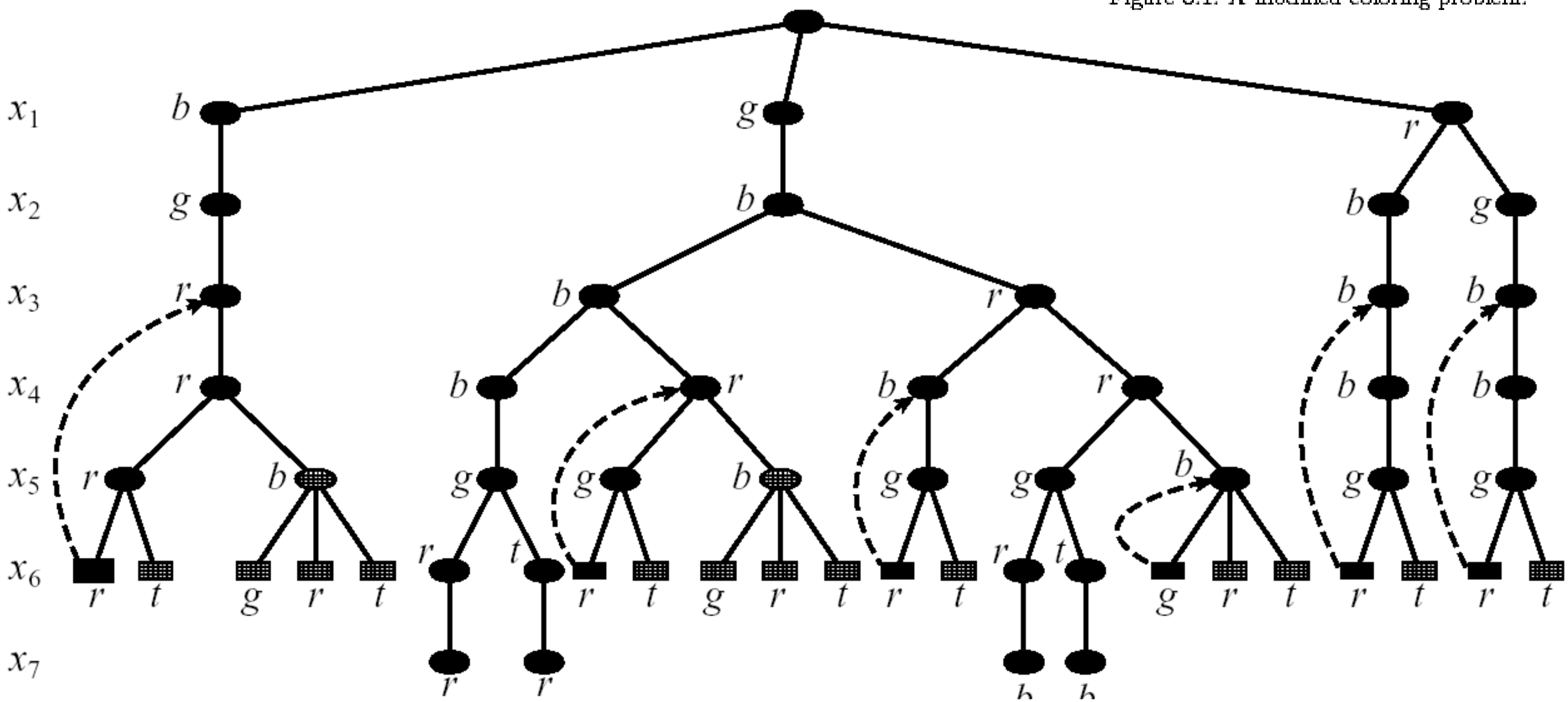


Figure 6.1: A modified coloring problem.



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. □

Jumps at Leaf Dead-End (Gascnnig 1977)

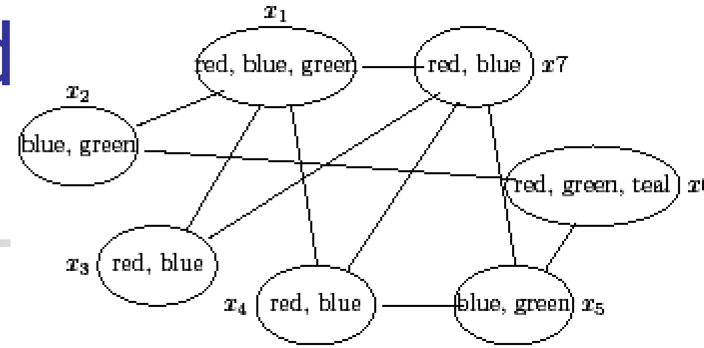
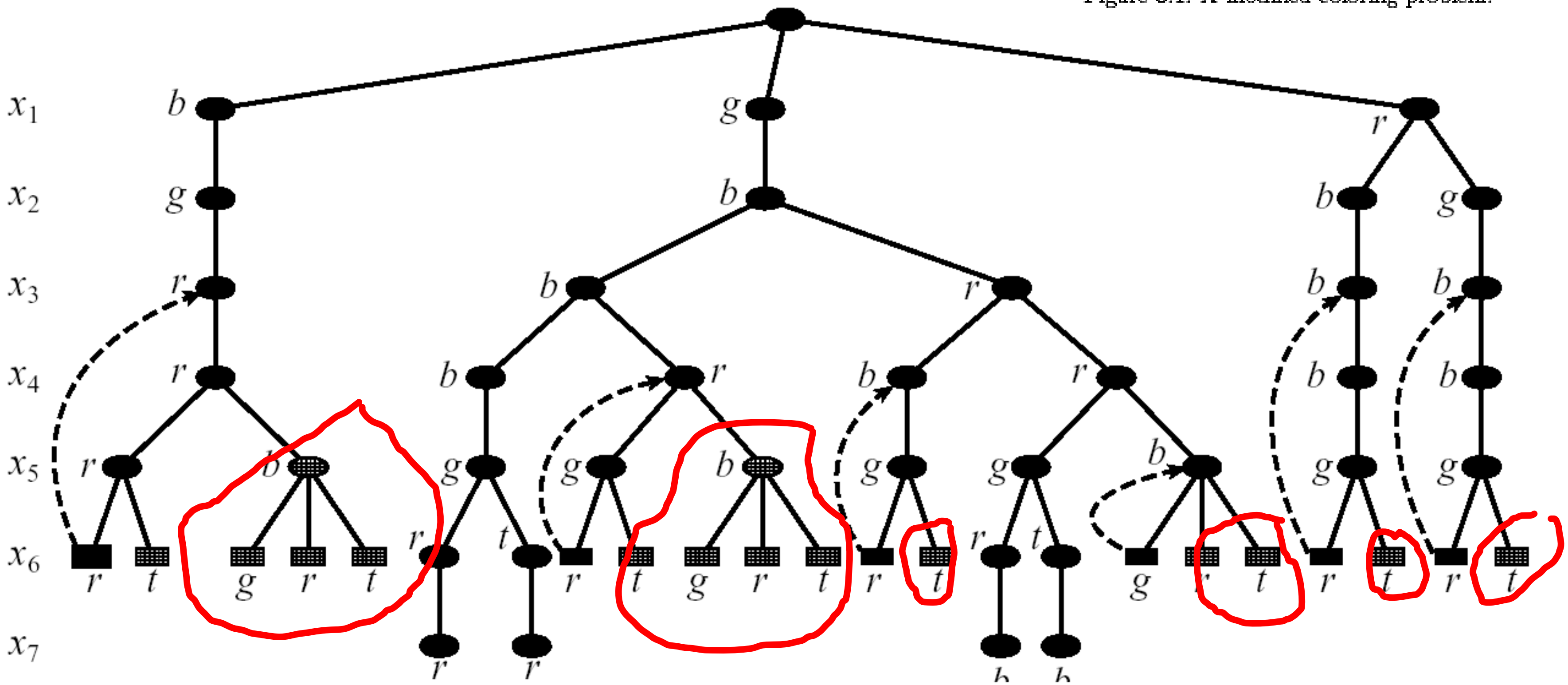


Figure 6.1: A modified coloring problem.



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. □

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:

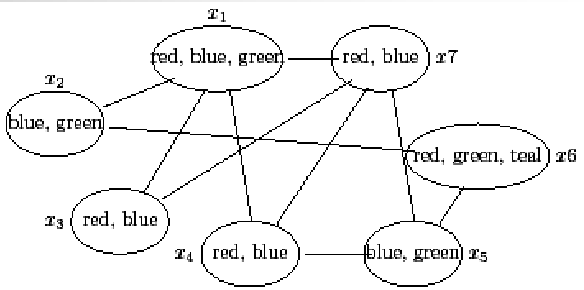
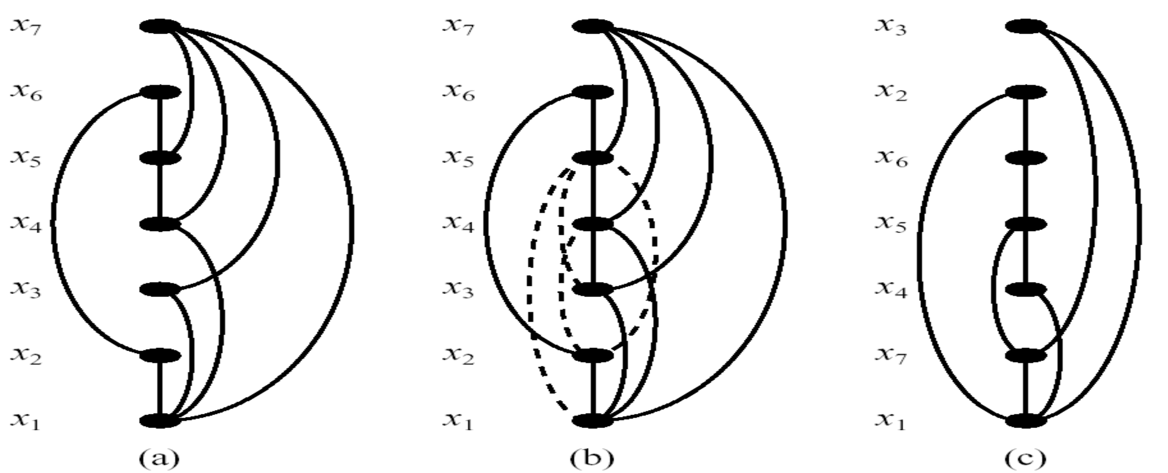


Figure 6.1: A modified coloring problem.

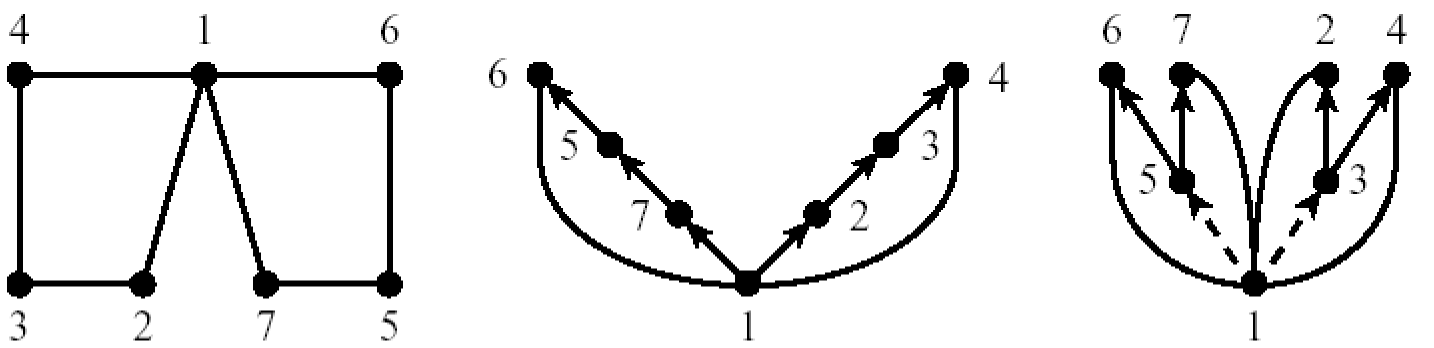


Backjumping styles

- 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

Complexity of Backjumping

Graph-based and conflict-based backjumpint



(a) (b) (c)

Simple: always jump back to parent in pseudo tree

We will see that: complexity: exp(m), m height, base bk

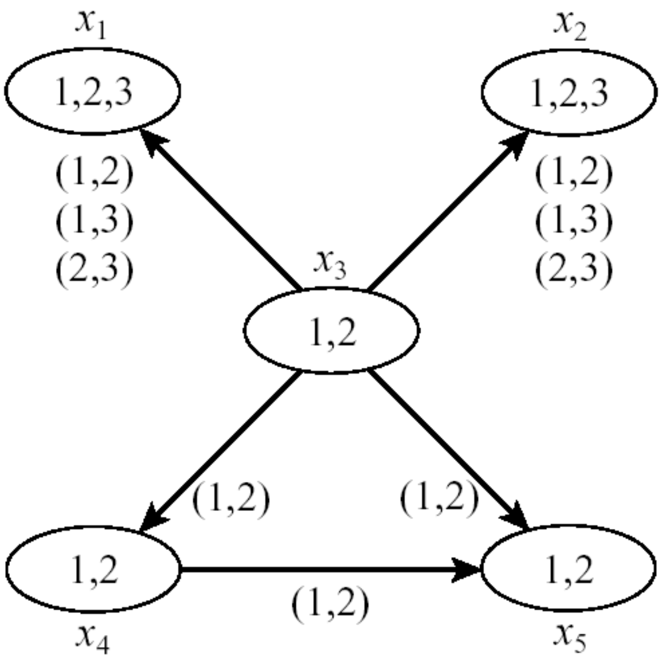
*Complexity for csp: exp(w*log n)*

*From exp(n) to exp(w*logn) while linear space*

(proof details: exercise)

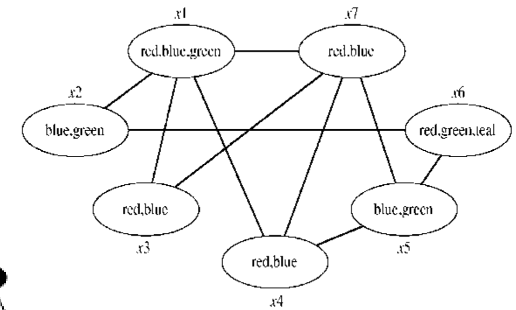
Look-back: No-good Learning

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



- (x1=2,x2=2,x3=1,x4=2) is a dead-end
- Conflicts to record:
  - (x1=2,x2=2,x3=1,x4=2) 4-ary
  - (x3=1,x4=2) binary
  - (x4=2) unary

No-good Learning Example



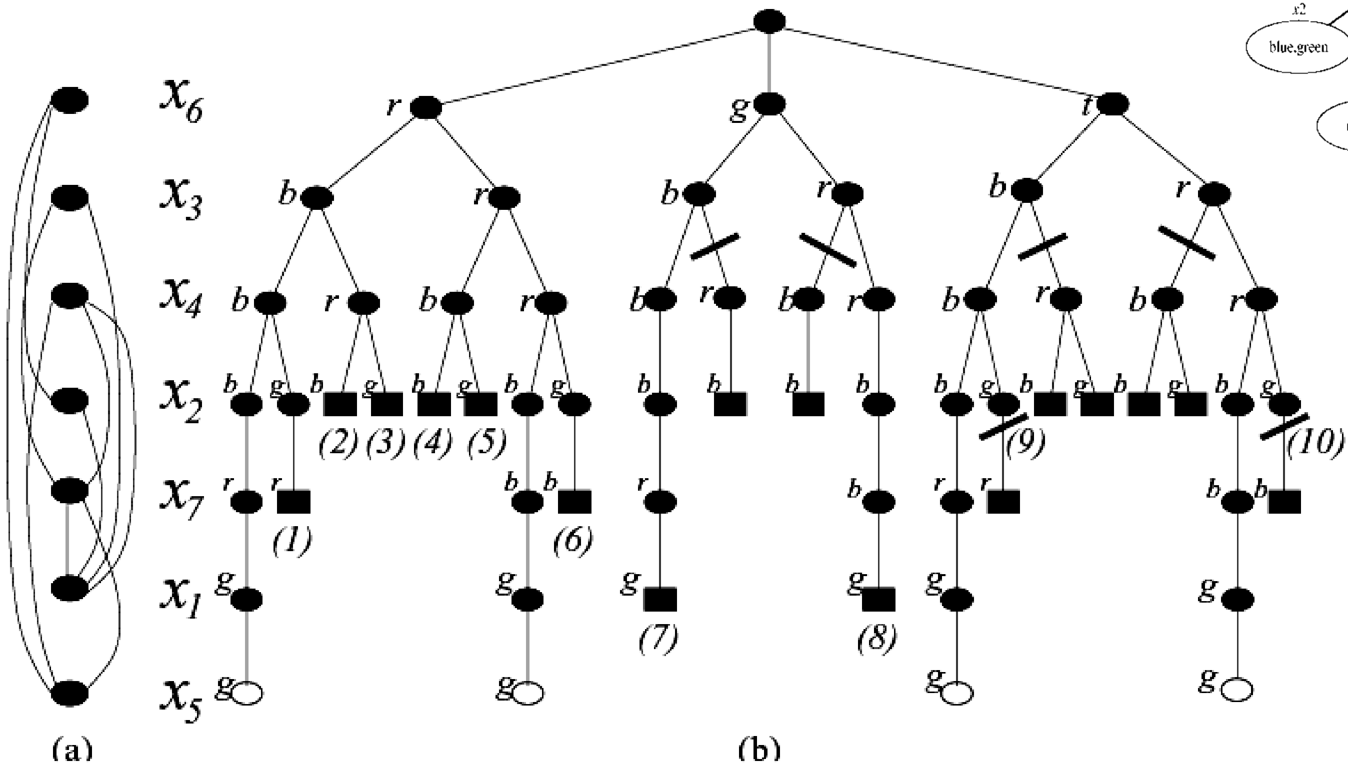


Figure 6.9: The search space explicated by backtracking on the CSP from Figure 6.1, using the variable ordering $(x_6, x_3, x_4, x_2, x_7, x_1, x_5)$ and the value ordering (*blue*, *red*, *green*, *teal*). Part (a) shows the ordered constraint graph, part (b) illustrates the search space. The cut lines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

Deep learning

- Deep learning: recording all and only minimal conflict sets
- Example:
- Although most accurate, overhead can be prohibitive: the number of conflict sets in the worst-case:

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

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

Complexity of backtrack-learning for CSP

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

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

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

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

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

m- depth of tree, e- number of constraints

Moving to New Queries

- Consistency and one solution.
- Counting
- Enumerating

Bucket-elimination for counting

Algorithm elim-count

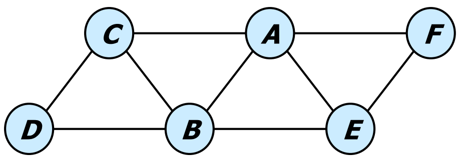
Input: A constraint network $\mathcal{R} = (X, D, C)$, ordering d.

Output: Augmented output buckets including the intermediate count functions and The number of solutions.

1. **Initialize:** Partition C (0-1 cost functions) into ordered buckets $bucket_1, \ldots, bucket_n$, We denote a function in a bucket N_i, and its scope S_i.)
2. **Backward:** For $p \leftarrow n$ downto 1, do
   Generate the function N^p: $N^p = \sum_{X_p} \prod_{N_i \in bucket_p} N_i$.
   Add N^p to the bucket of the latest variable in $\bigcup_{i=1}^{j} S_i - \{X_p\}$.
3. **Return** the number of solutions, N^1 and the set of output buckets with the original and computed functions.

Figure 13.9: Algorithm *elim-count*

#CSP - Tree DFS Traversal

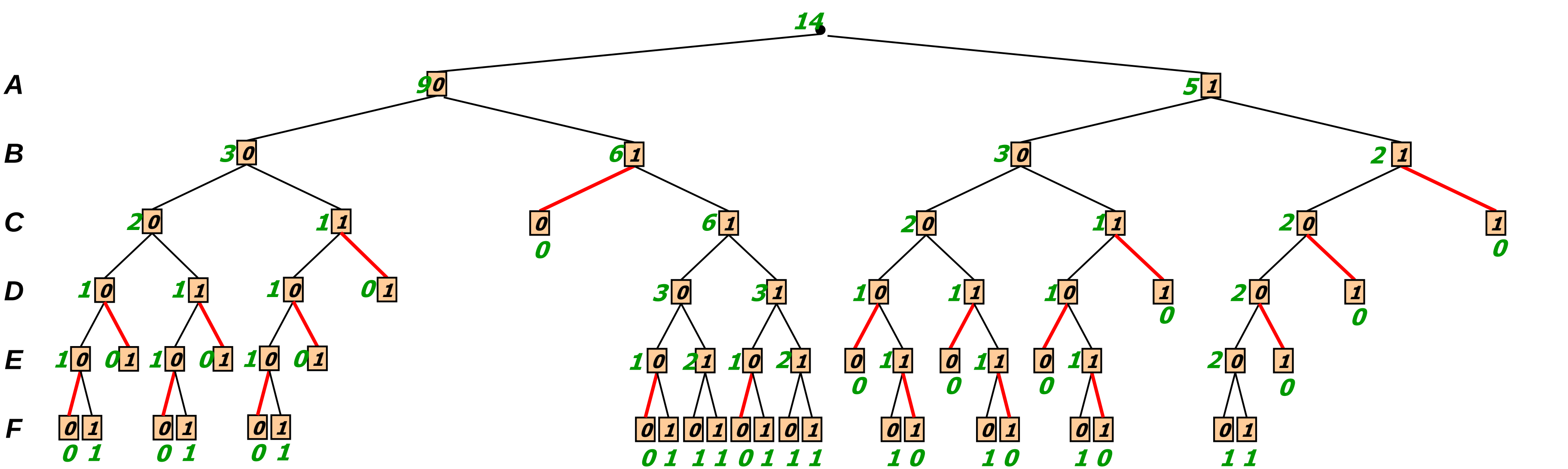


A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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**

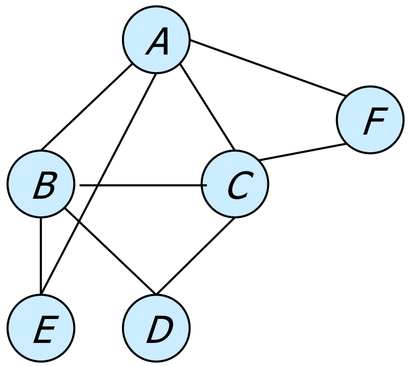


Value of node = number of solutions below it

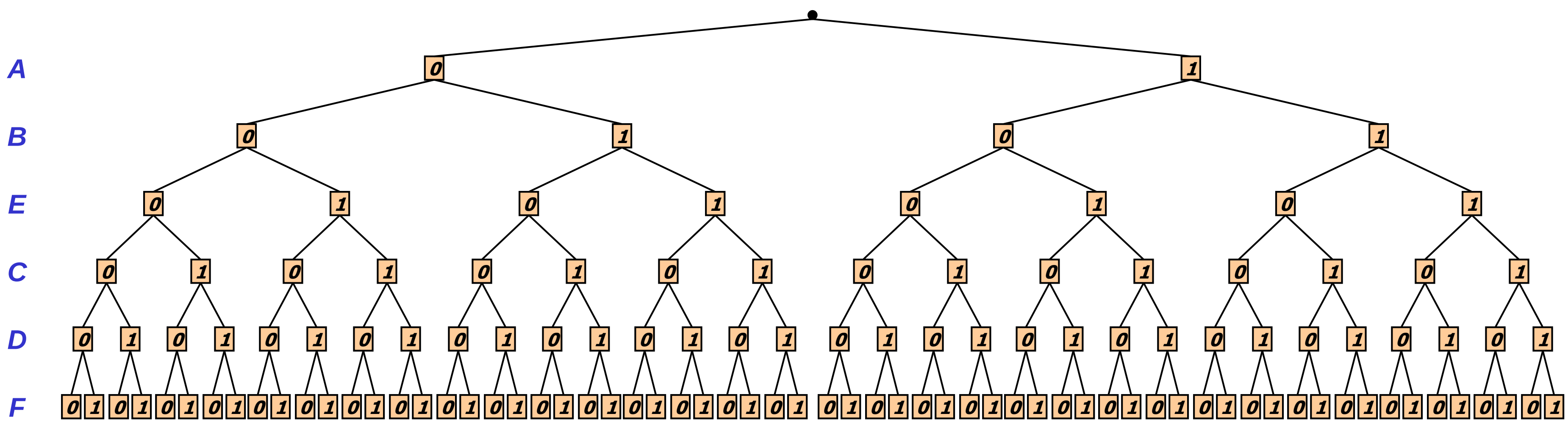
Outline

- Improving search by bounded-inference in branching ahead
- Improving search by looking-back
- The alternative AND/OR search space

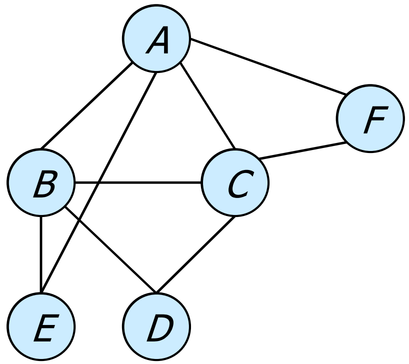
OR Search Space



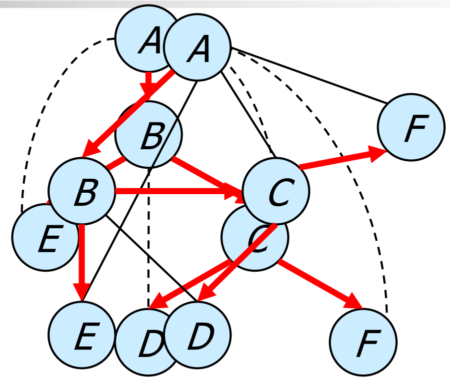
Ordering: A B E C D F



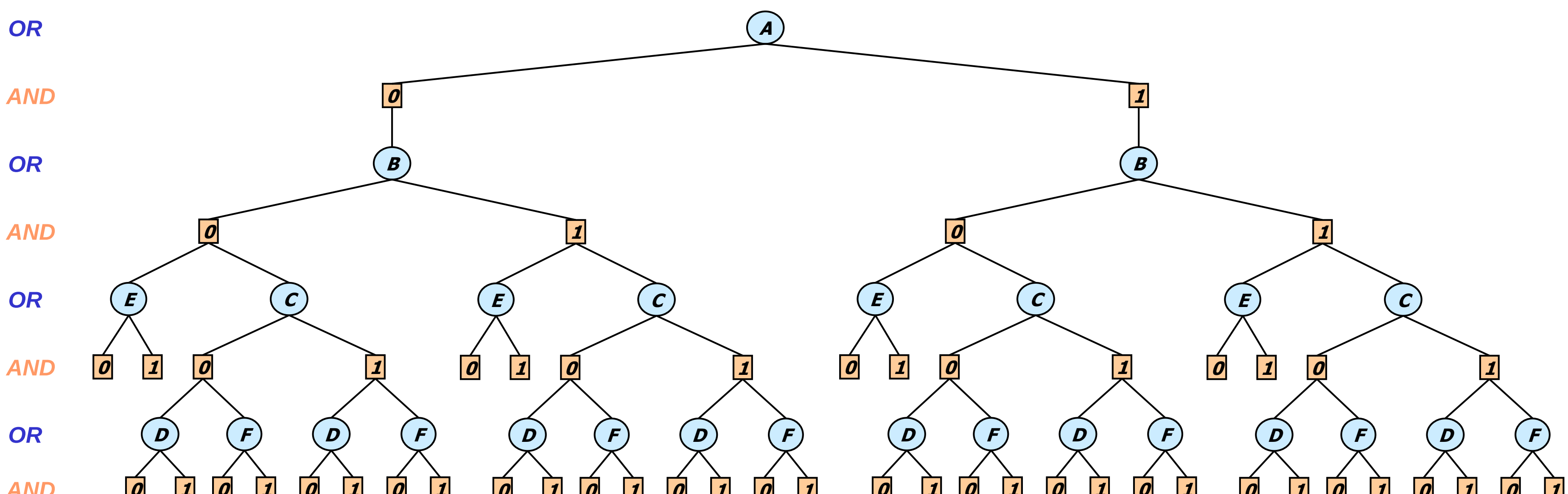
AND/OR Search Space



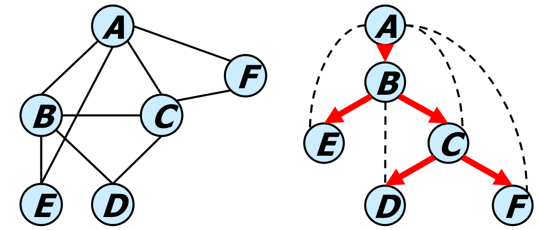
Primal graph

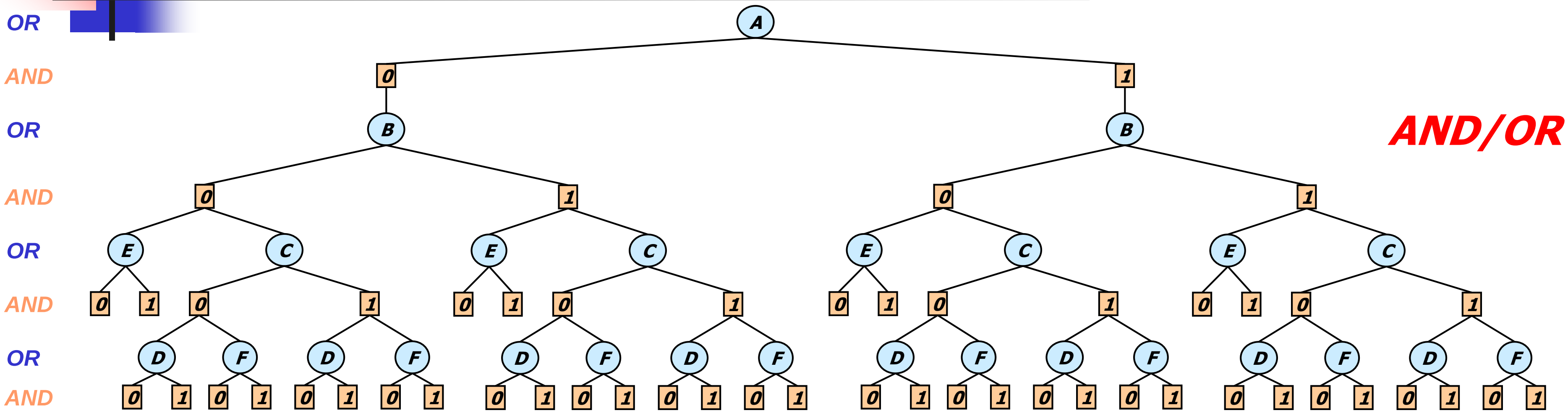


DFS tree

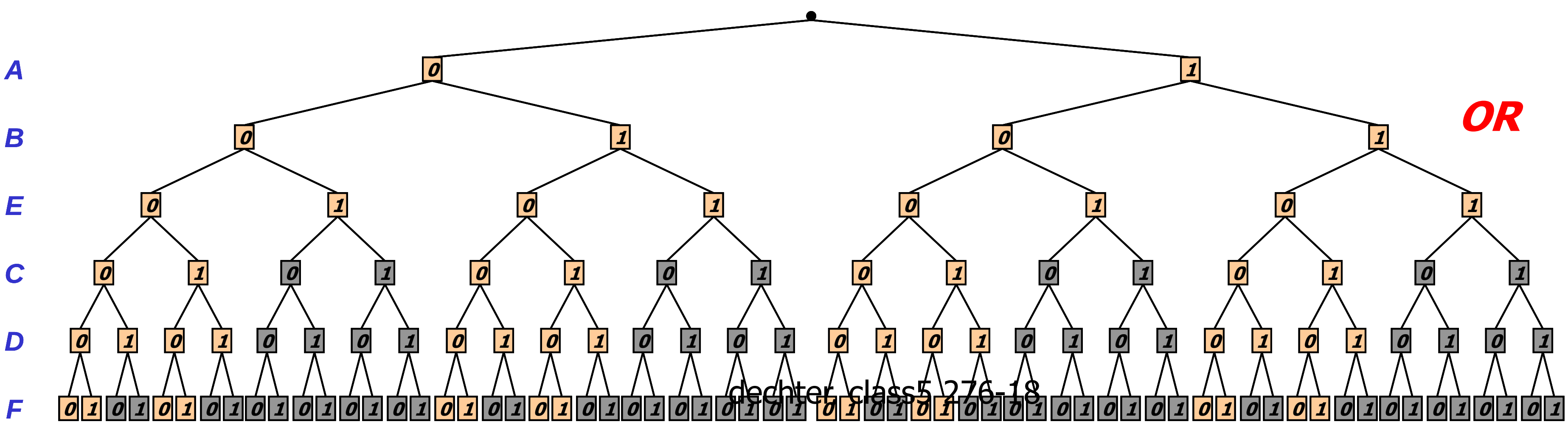


AND/OR vs. OR



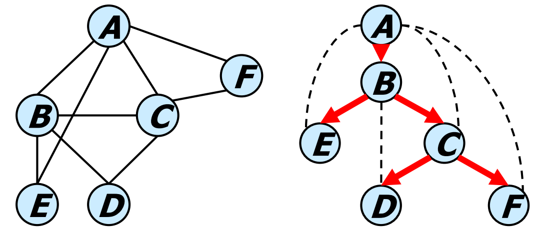


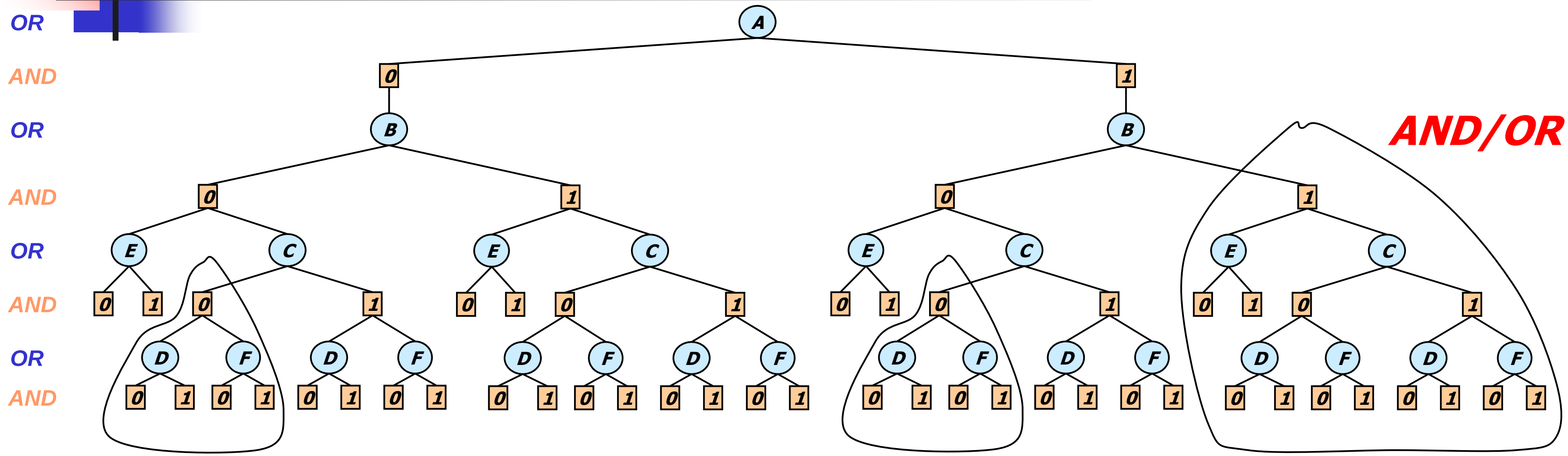
AND/OR size: exp(4), OR size exp(6)

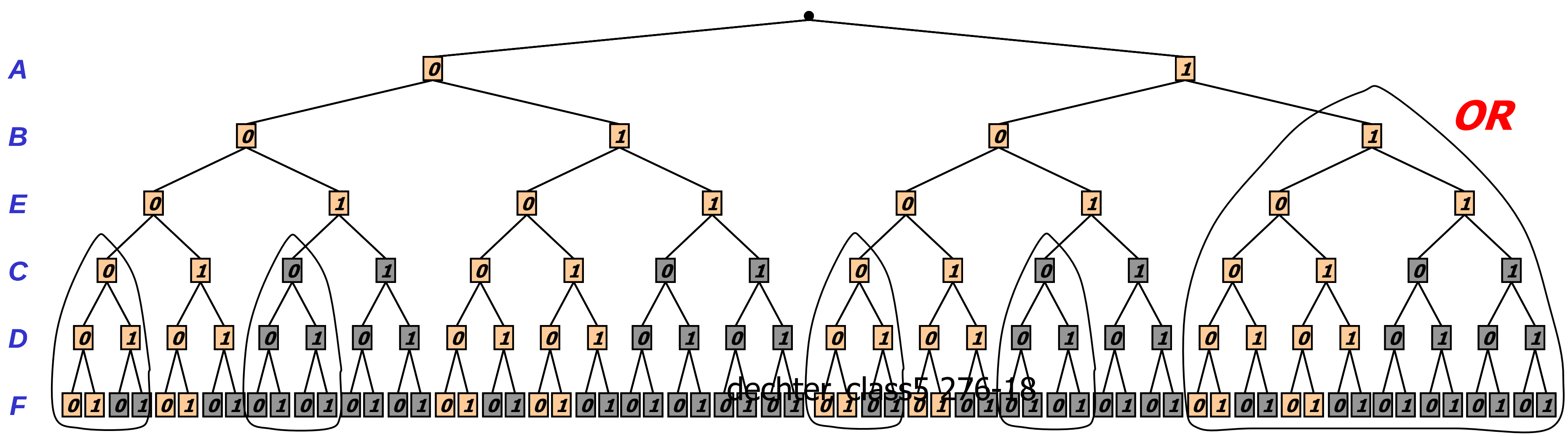


AND/OR vs. OR

No-goods
(A=1,B=1)
(B=0,C=0)

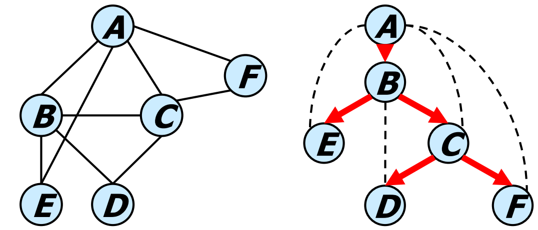


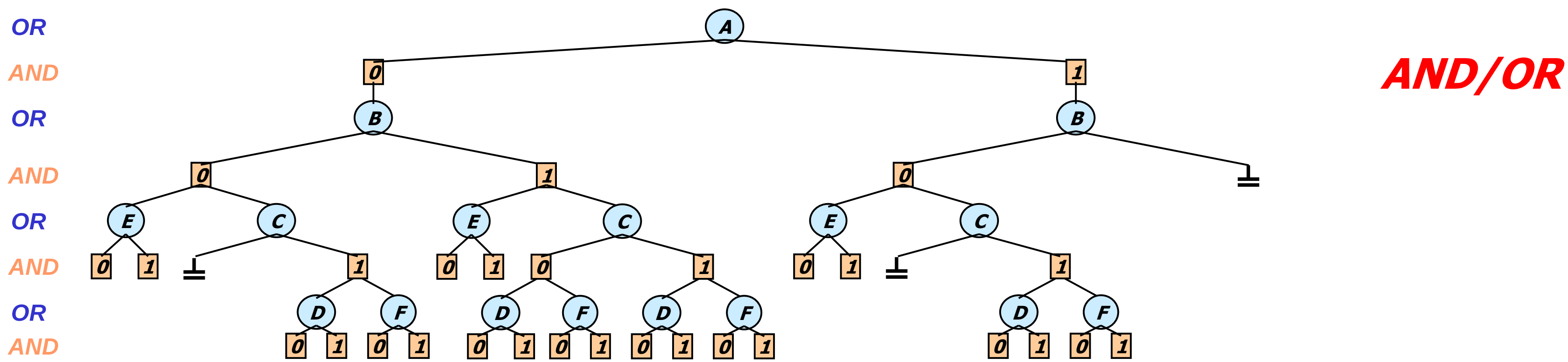


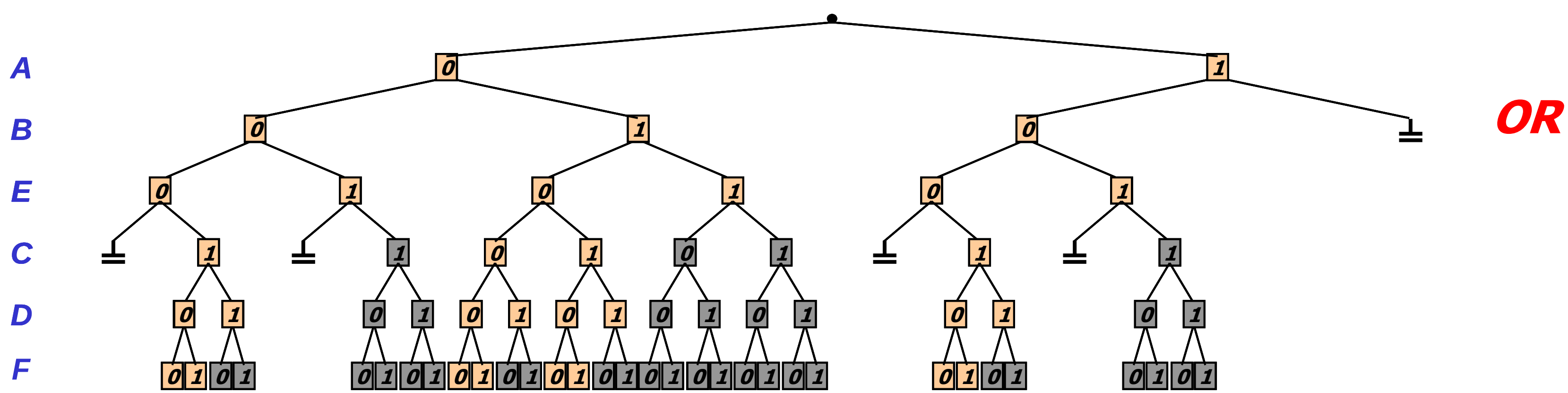


AND/OR vs. OR

(A=1,B=1)
(B=0,C=0)

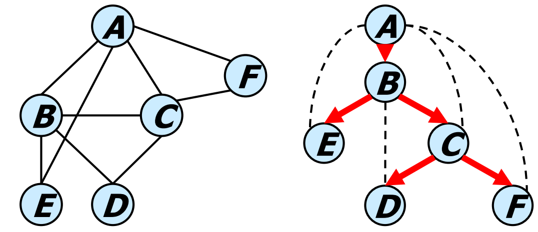


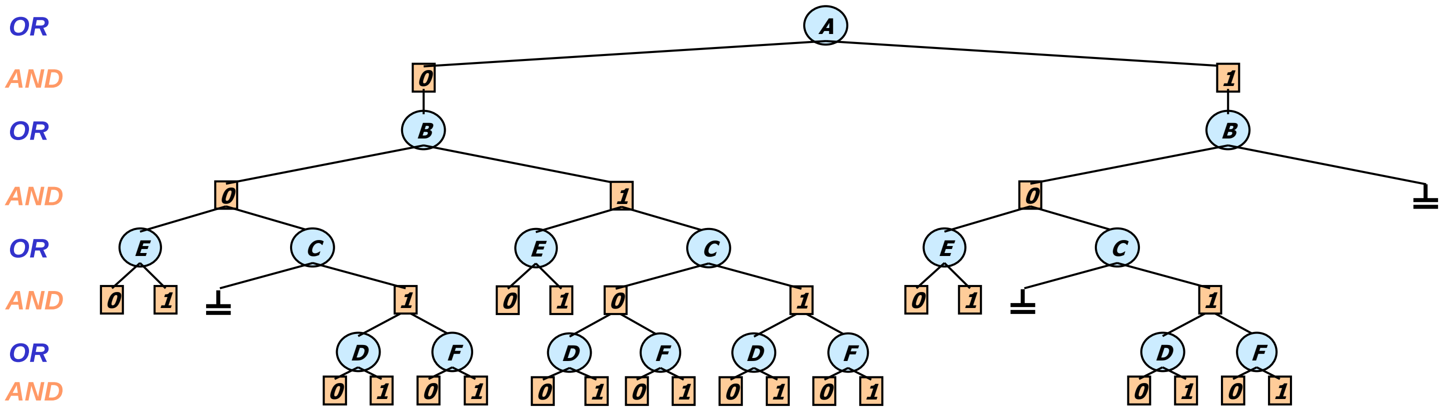




AND/OR vs. OR

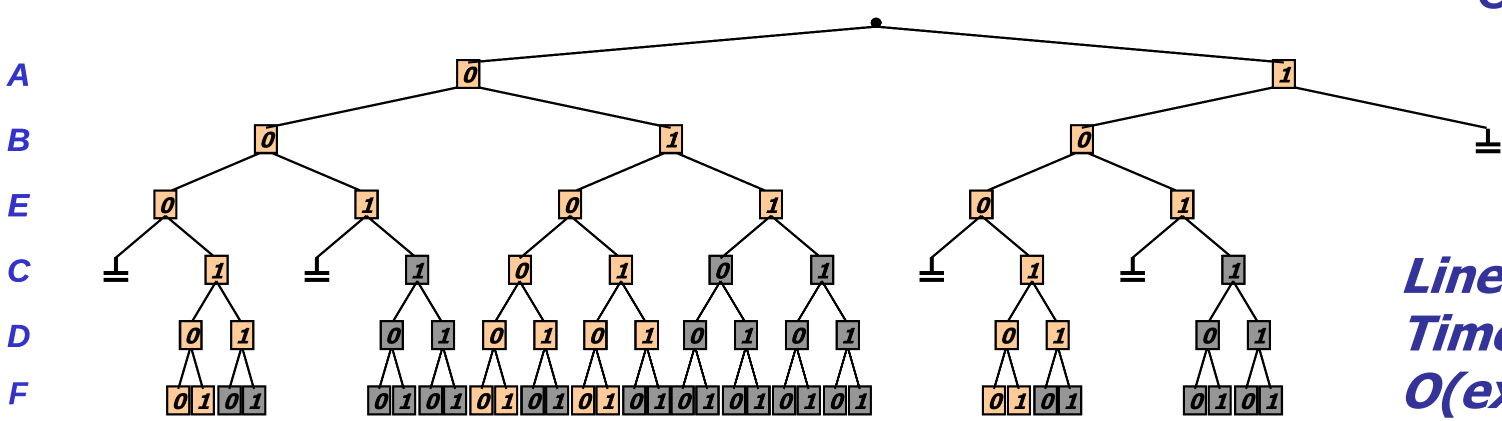
(A=1,B=1)
(B=0,C=0)





AND/OR

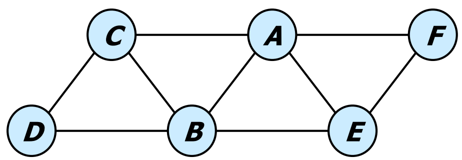
Space: linear
Time:
O(exp(m))
O(w* log n)



OR

Linear space,
Time:
O(exp(n))

#CSP – AND/OR Search Tree

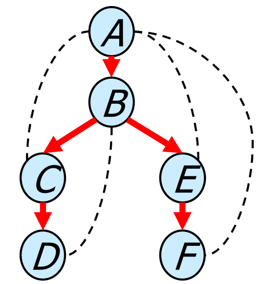


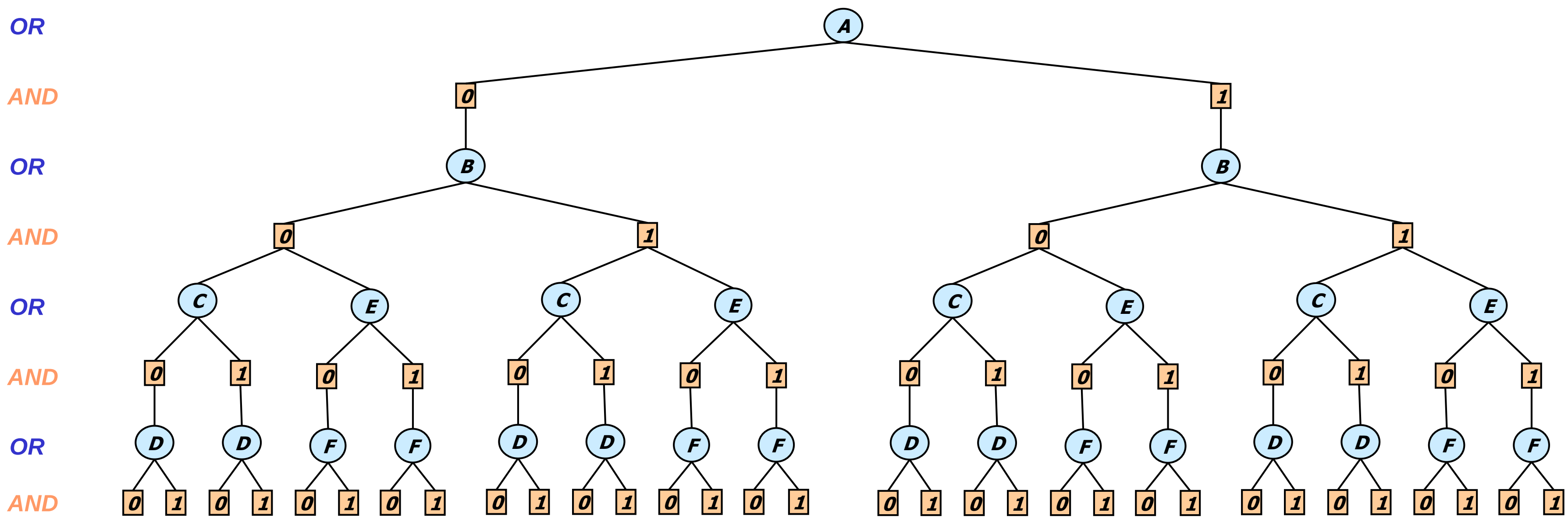
A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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**

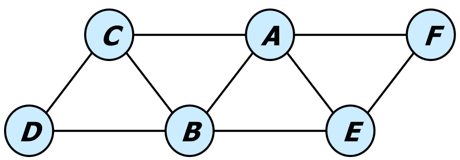
A	B	E	R_{ABE}
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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 – AND/OR Tree DFS

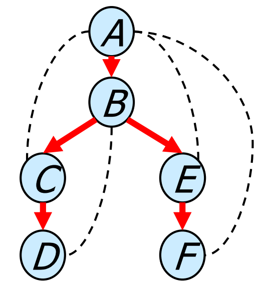


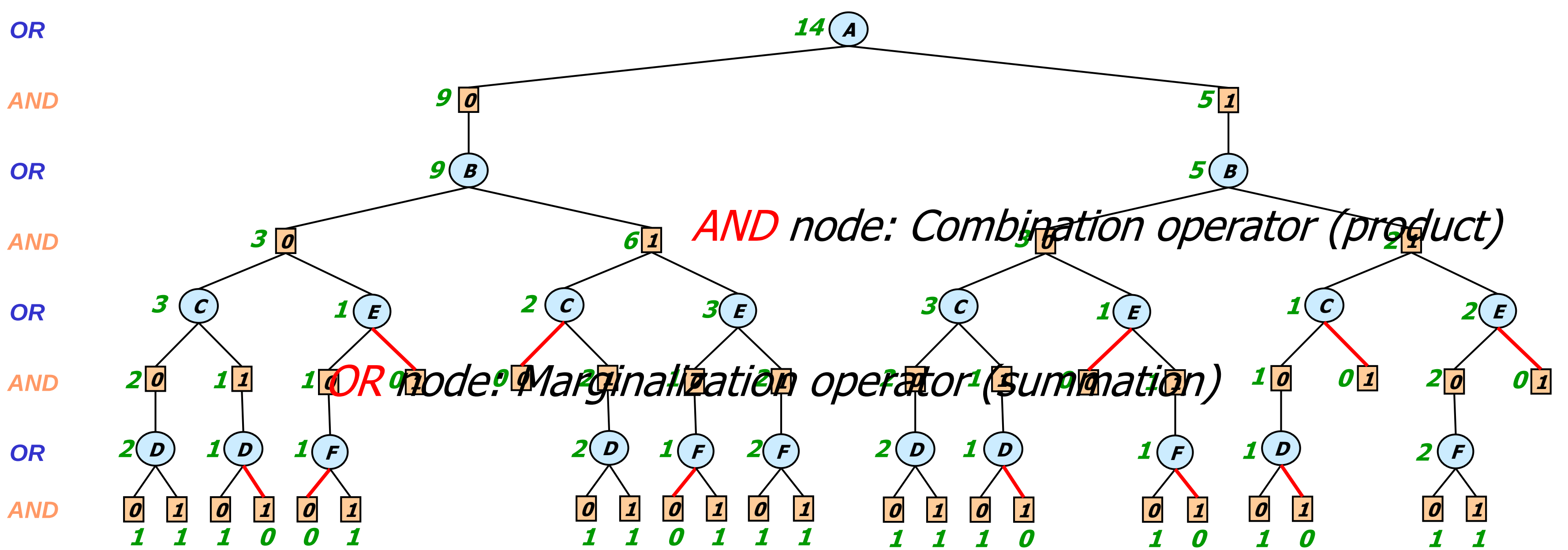
A	B	C	R_{ABC}
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

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**



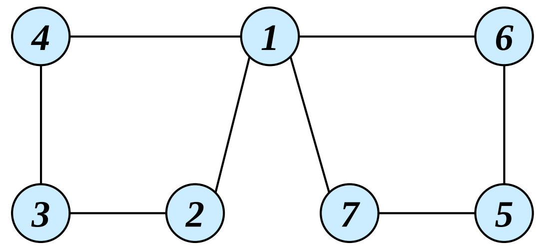


AND node: Combination operator (product)

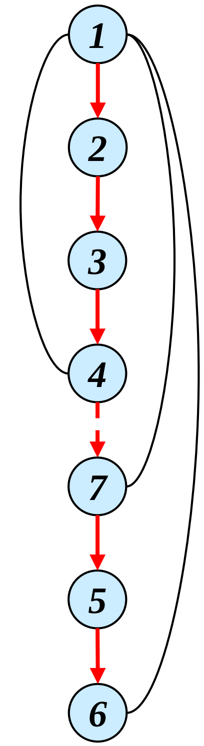
OR node: Marginalization operator (summation)

Pseudo-Trees

(Freuder 85, Bayardo 95, Bodlaender and Gilbert, 91)

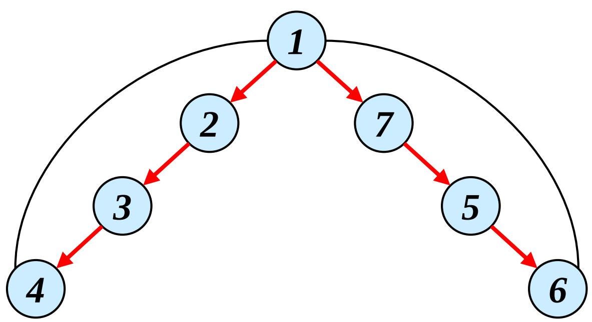


(a) Graph

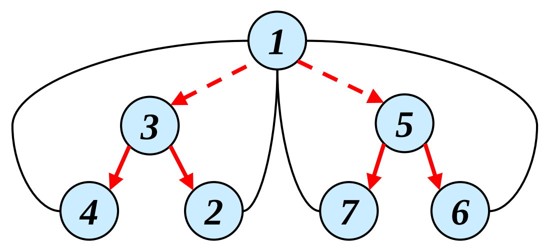


(d) Chain
depth=6

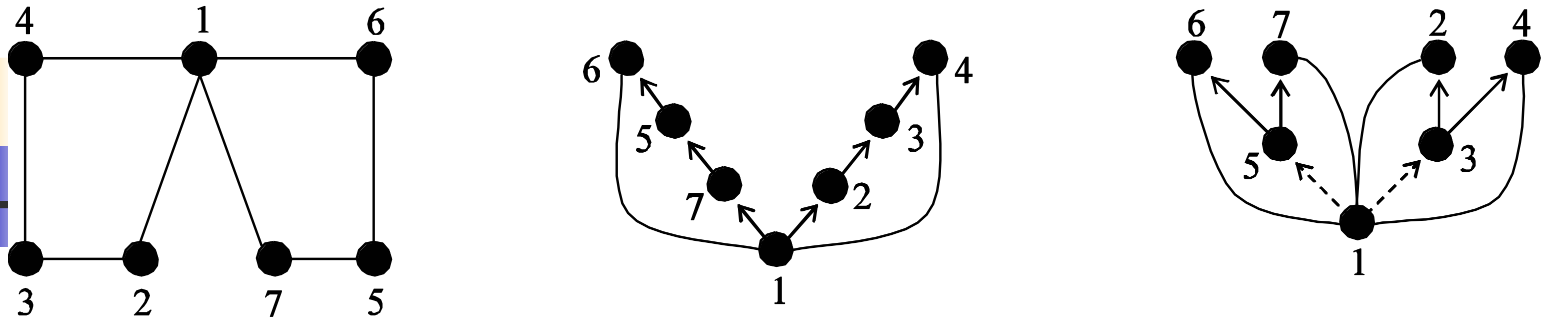
$h <= w^* \log n$



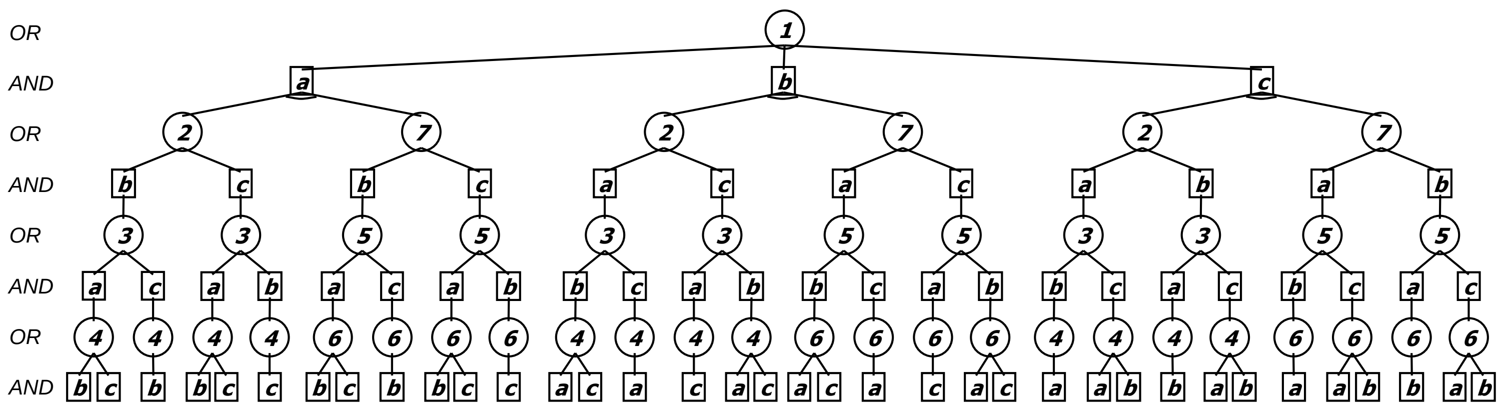
(b) DFS tree
depth=3

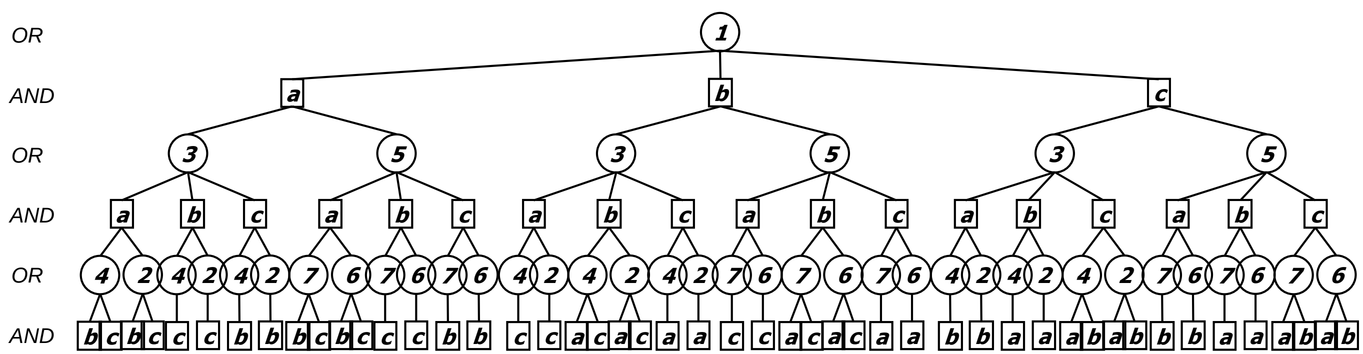


(c) pseudo- tree
depth=2


4 1 6
3 2 7 5
6 5 7 1 2 3 4
6 7 2 4
5 3
1

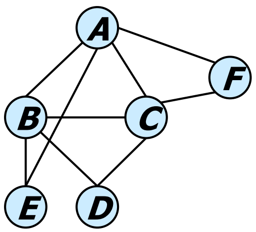
(a) **(b)** **(c)**


OR
AND
OR
AND
OR
AND
OR
AND

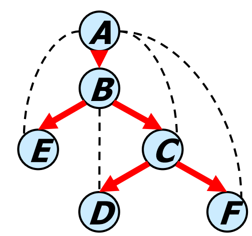

OR
AND
OR
AND
OR
AND

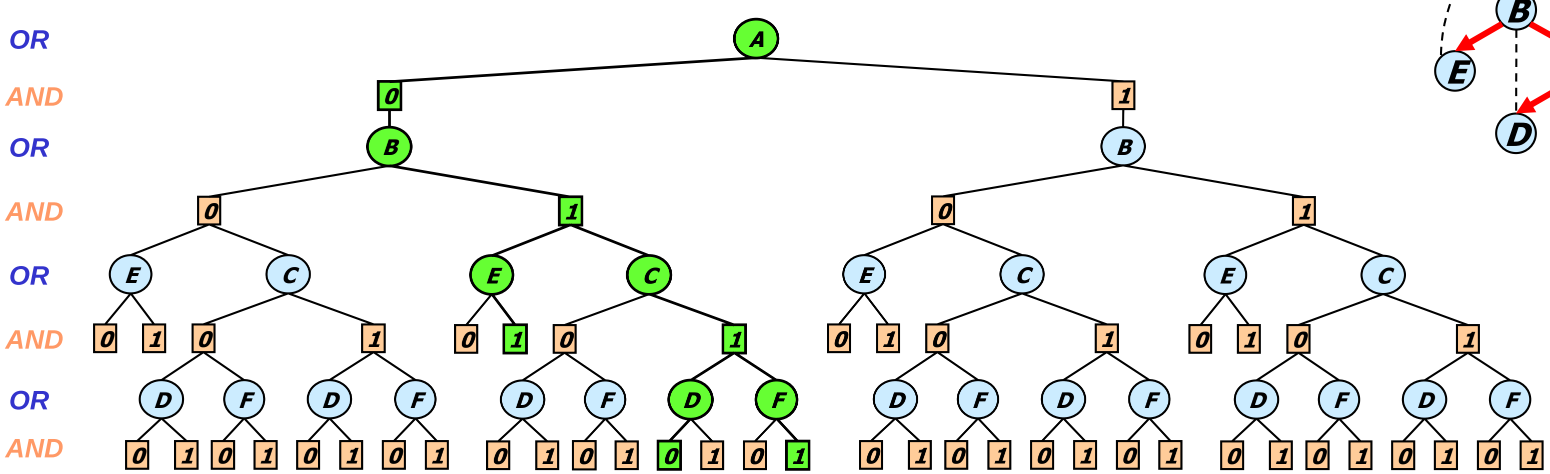
AND/OR search tree for graphical models

- **The AND/OR search tree of R relative to a tree, T, has:**
  - **Alternating levels of: OR nodes (variables) and AND nodes (values)**
- **Successor function:**
  - **The successors of OR nodes X are all its consistent values along its path**
  - **The successors of AND <X,v> are all X child variables in T**



- **A solution is a consistent subtree**
- **Task: compute the value of the root node**





The end