*Algorithms for Reasoning with graphical models*

# Slides Set 8:
# Search for Constraint Satisfaction

*Rina Dechter*

(***Dechter2 chapters 5-6, Dechter1 chapter 6***)

# Sudoku – Approximation: Constraint Propagation

- ***Constraint***
- ***Propagation***

- ***Inference***

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 2 | 4 | | 6 | | | |
| 8 | 6 | 5 | 1 | | | 2 | | |
| | 1 | | | | 8 | 6 | | 9 |
| 9 | | | | 4 | | 8 | 6 | |
| | 4 | 7 | | | | 1 | 9 | |
| | 5 | 8 | | 6 | | | | 3 |
| 4 | | 6 | 9 | | | | 7 | 2,3 4,6 2 |
| | | 9 | | | 4 | 5 | 8 | 1 |
| | | | 3 | | 2 | 9 | | |

- ***Variables**: empty slots*
- ***Domains** = {1,2,3,4,5,6,7,8,9}*
- ***Constraints**:*
  - *27 all-different*

***Each row, column and major block must be alldifferent***

***"Well posed" if it has unique solution: 27 constraints***

# 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

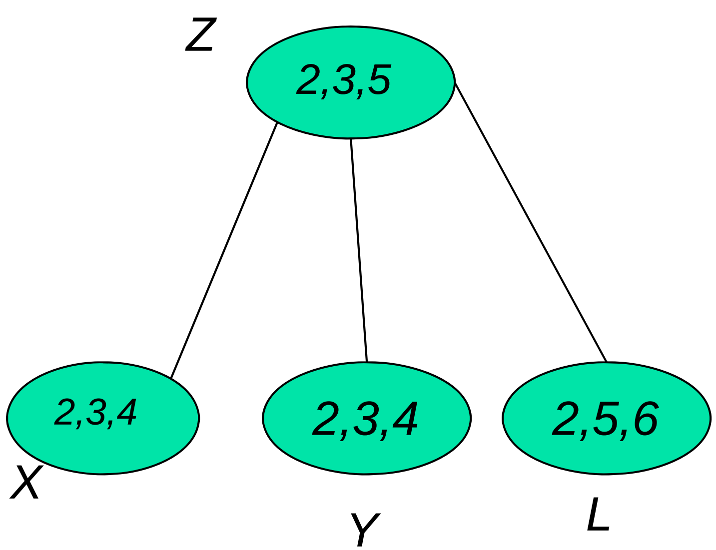
# What if the CN is Not Backtrack-free?

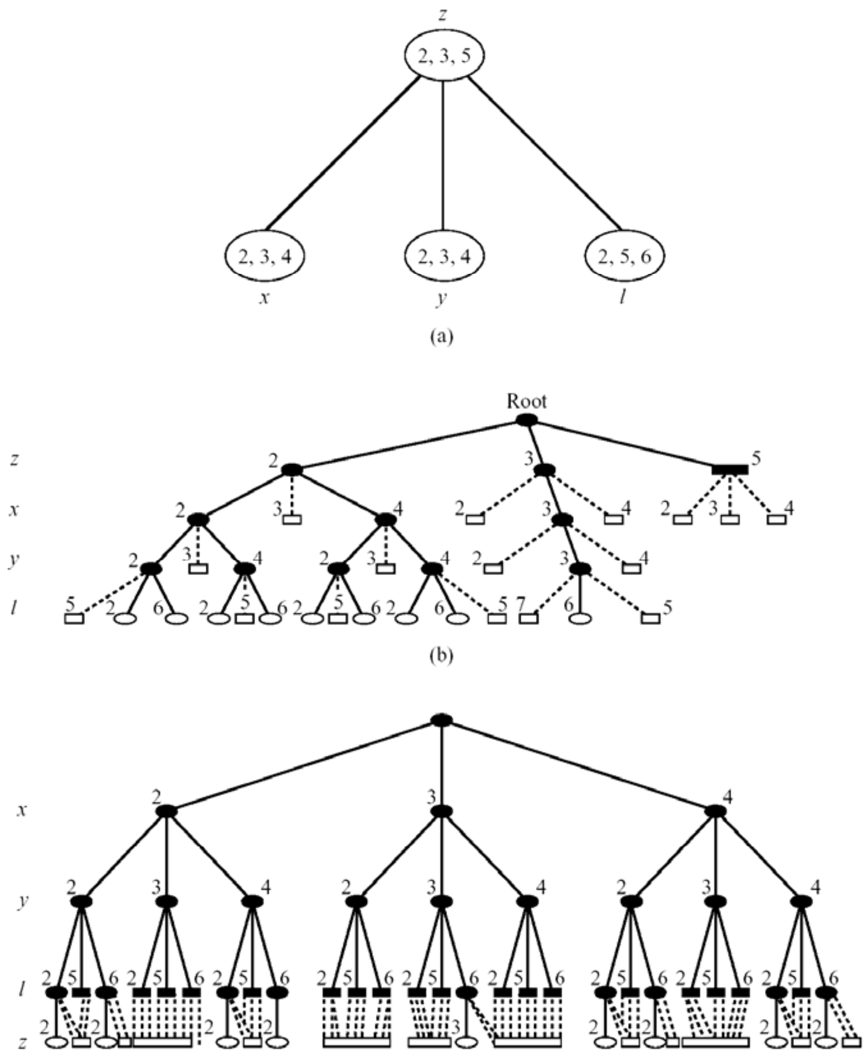
- Backtrack-free in general is too costly, so what to do?
- Search?
- What is the search space?
- How to search it? Breadth-first? Depth-first?

# The Search Space for a CN

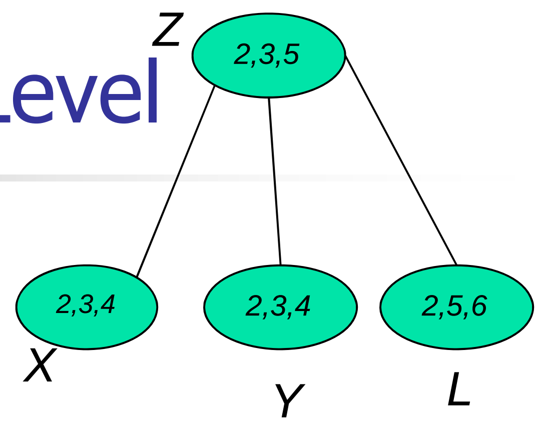
- A tree of all partial solutions
- A partial solution: $(a_1, \ldots, a_j)$ satisfying all relevant constraints
- The size of the underlying search space depends on:
  - Variable ordering
  - Level of consistency possessed by the problem

# The Effect of Variable Ordering

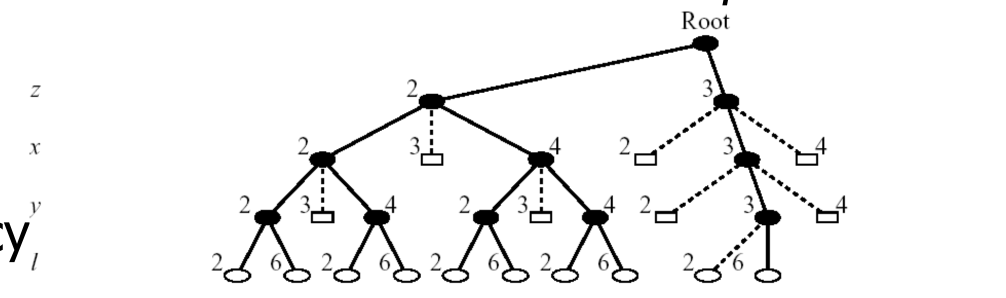


# The Effect of Consistency Level

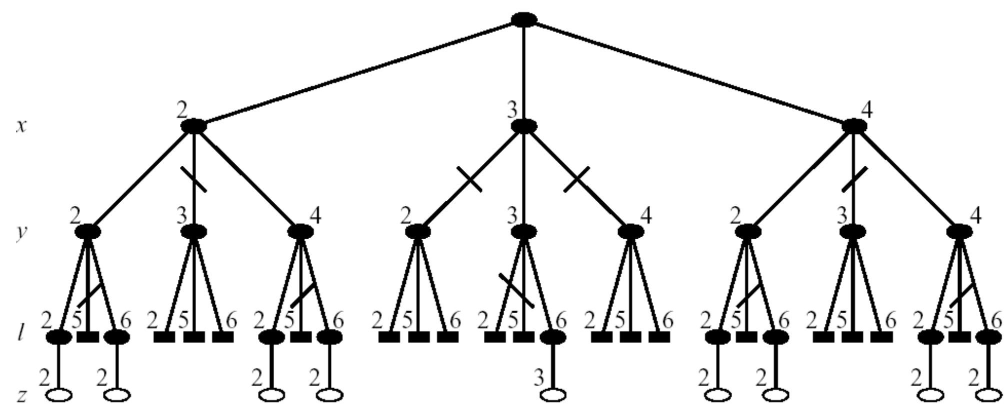



- After arc-consistency z=5 and l=5 are removed



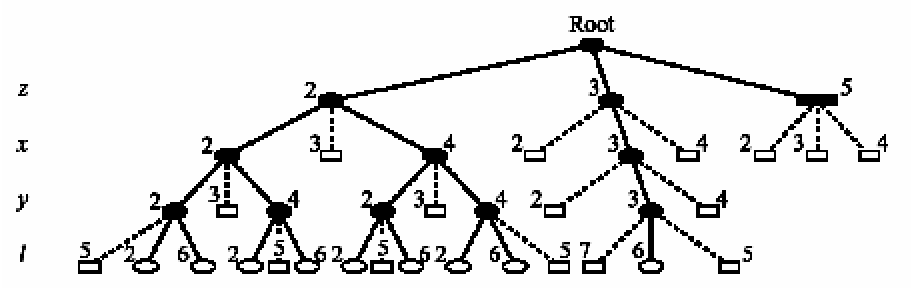
(a)

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

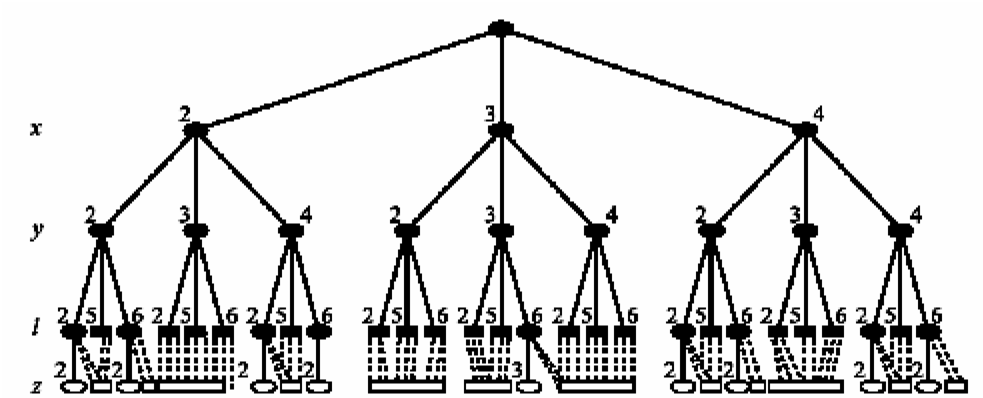


(b)

# The Effect of Variable Ordering

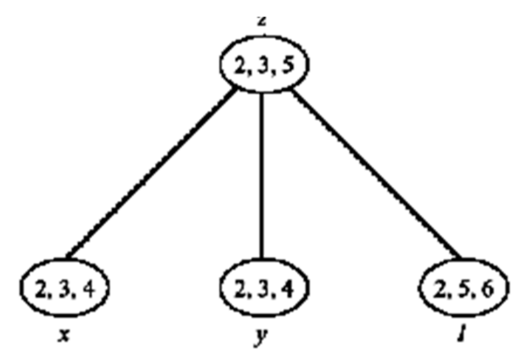


(b)



(c)

*z divides x, y and t*



(a)

# Sudoku –
# Search in Sudoku. Variable ordering?
# Constraint propagation?

- ***Constraint***
- ***Propagation***

- ***Inference***

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 2 | 4 | | 6 | | | |
| 8 | 6 | 5 | 1 | | | 2 | | |
| | 1 | | | | 8 | 6 | | 9 |
| 9 | | | | 4 | | 8 | 6 | |
| | 4 | 7 | | | | 1 | 9 | |
| | 5 | 8 | | 6 | | | | 3 |
| 4 | | 6 | 9 | | | | 7 | 2 ~~3~~ ~~4~~ ~~6~~ **2** |
| | | 9 | | | 4 | 5 | 8 | 1 |
| | | | 3 | | 2 | 9 | | |

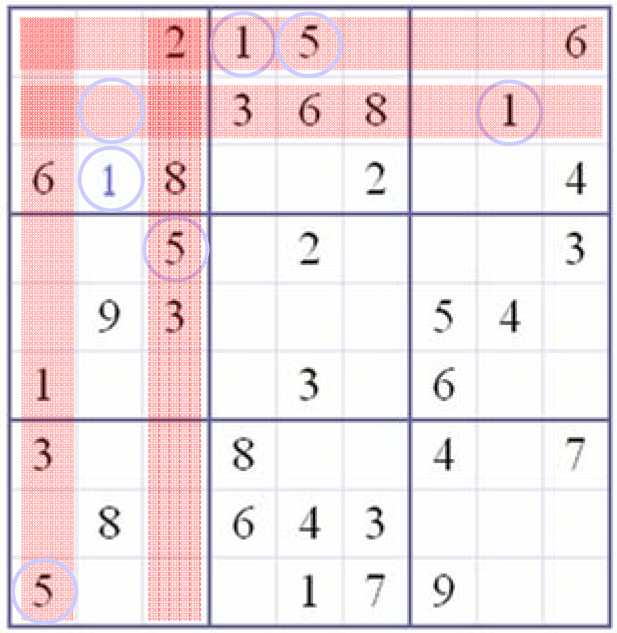
- ***Variables*: empty slots**
- ***Domains* = {1,2,3,4,5,6,7,8,9}**
- ***Constraints*:**
  - ***27 all-different***

***Each row, column and major block must be alldifferent***

***"Well posed" if it has unique solution: 27 constraints***

# Sudoku

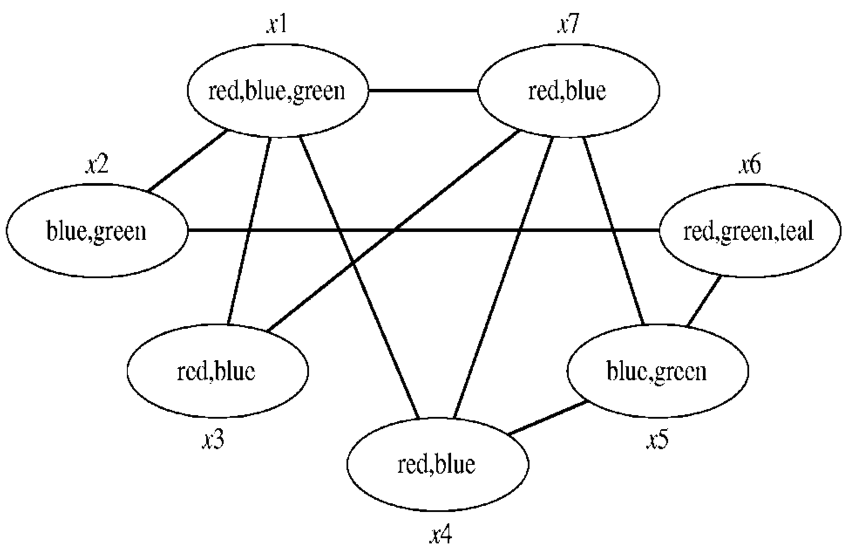
*Alternative formulations:*
*Variables?*
*Domains?*
*Constraints?*

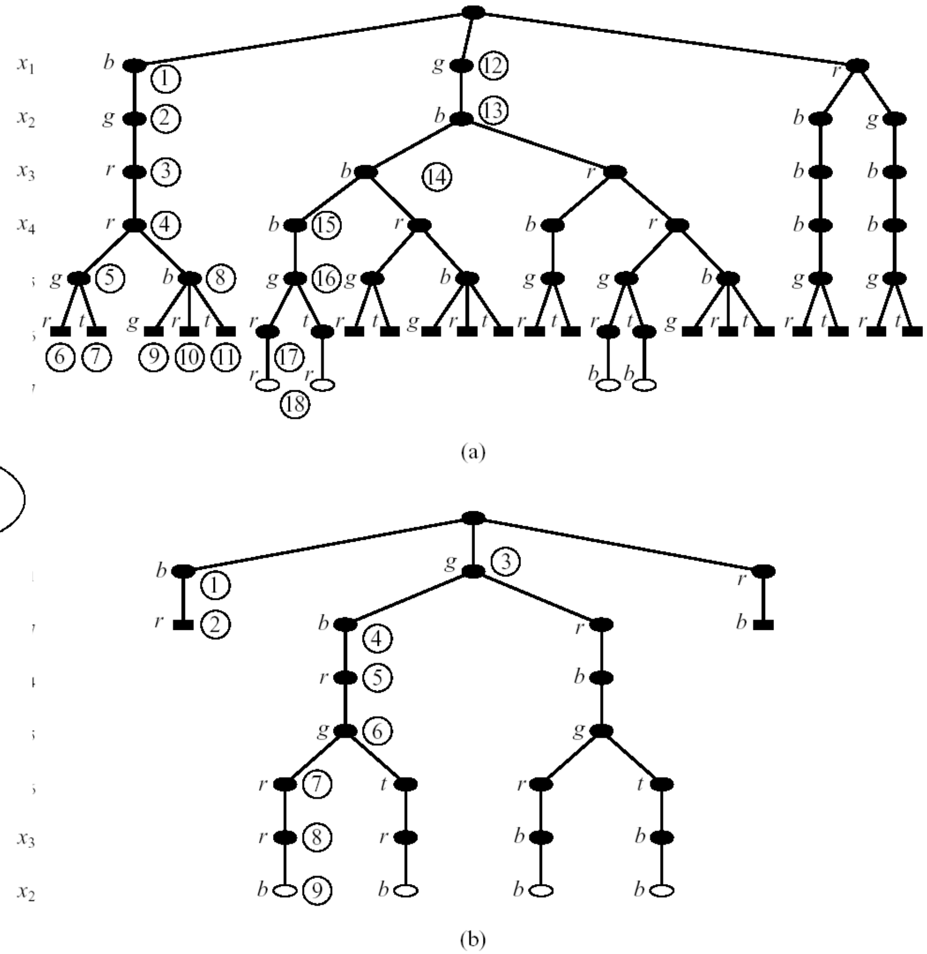



***Each row, column and major block must be alldifferent***

***"Well posed" if it has unique solution***

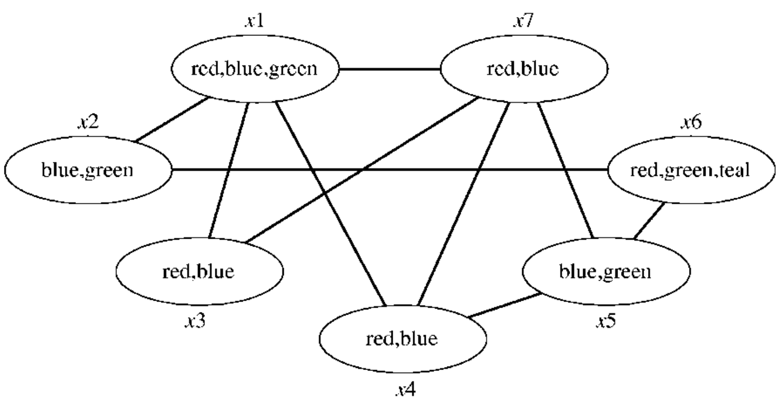
# Backtracking Search for a Solution

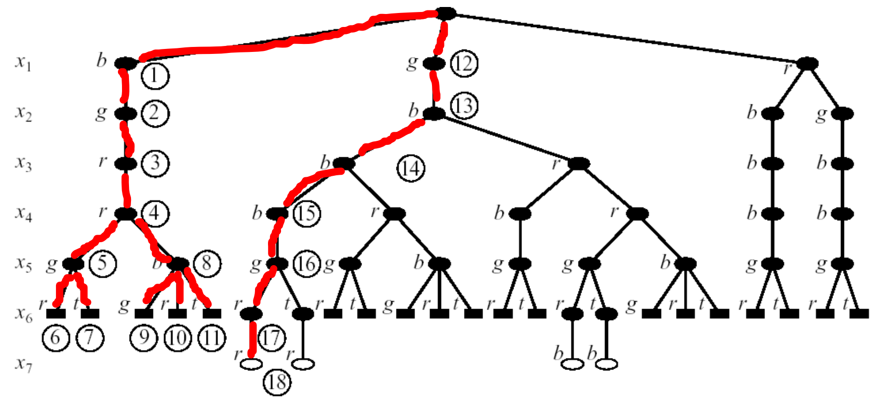


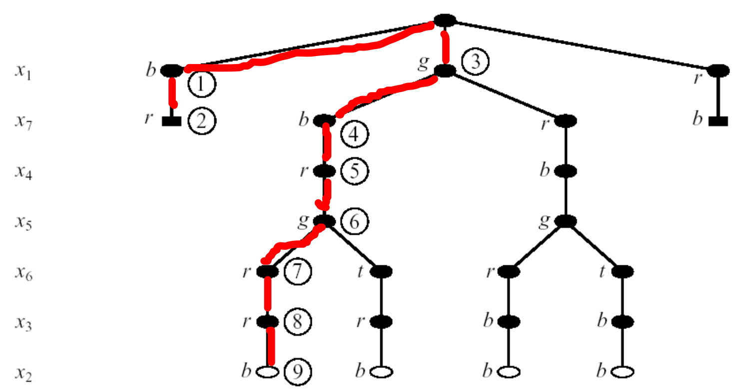
*Second ordering = (1,7,4,5,6,3,2)*

# Backtracking Search for a Solution



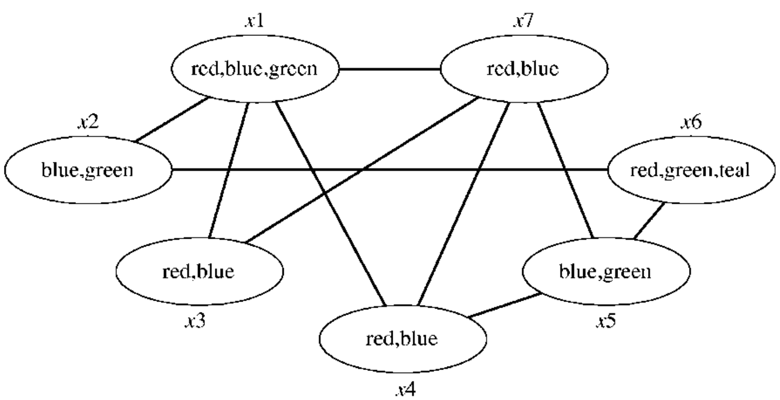



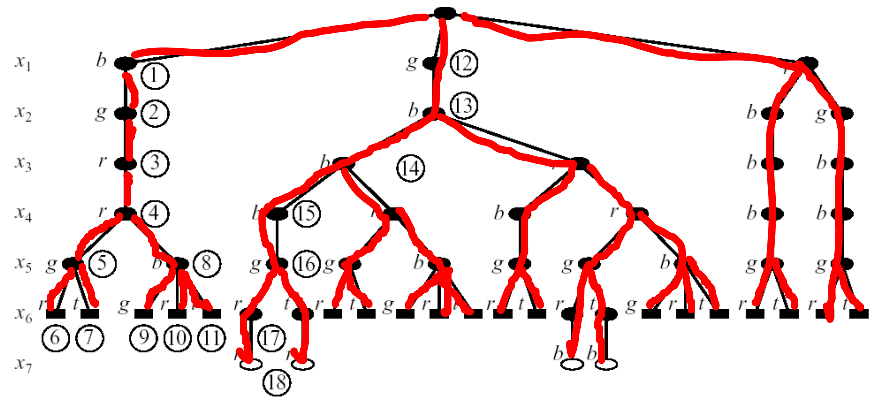
(a)



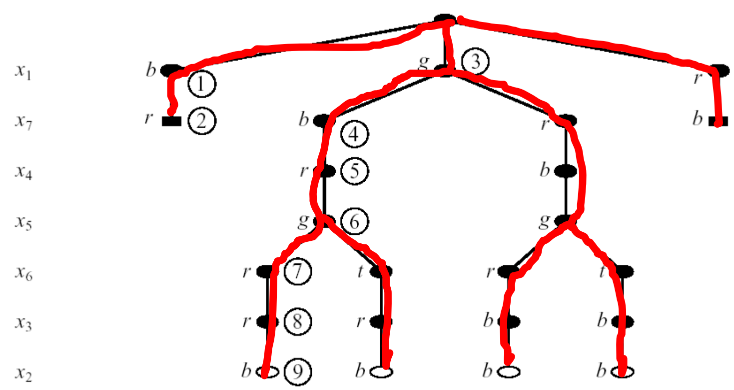
(b)

# Backtracking Search for All Solutions



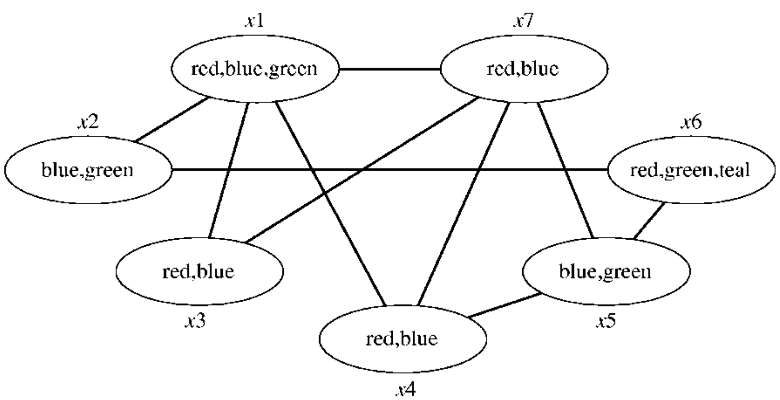



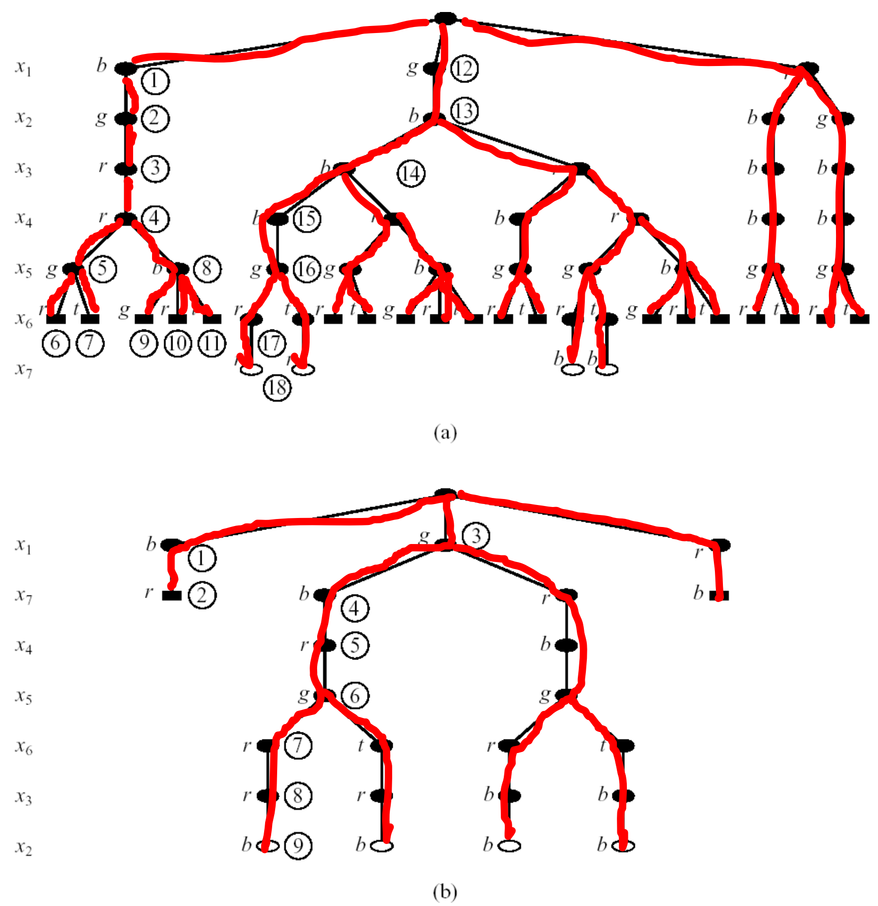
(a)



(b)

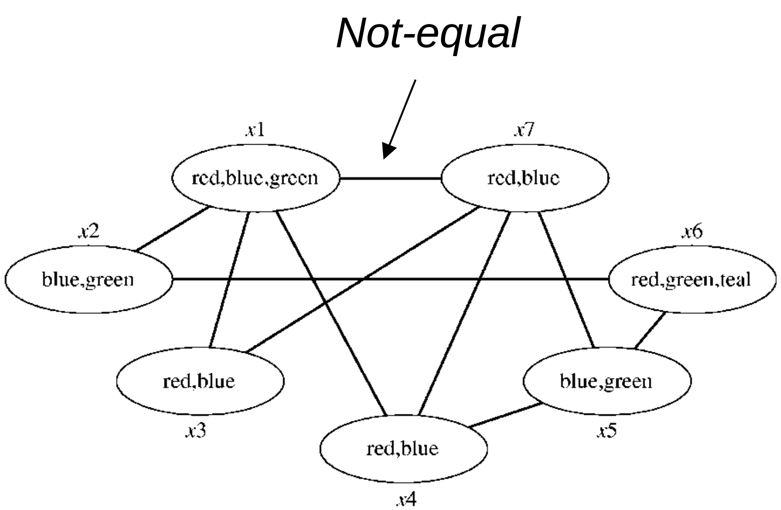
# Backtracking search for *all* solutions

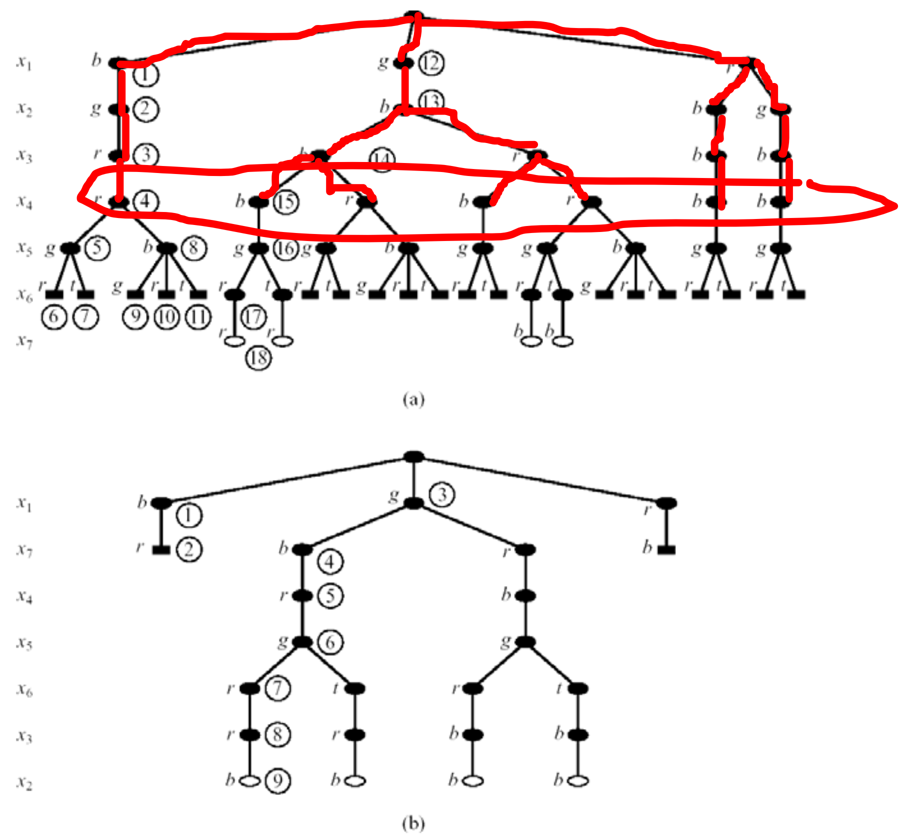



***For all tasks***
***Time: $O(k^n)$***
***Space: linear***
***n= number of variables***
***K = max domain size***

# Traversing Breadth-First (BFS)?



***BFS memory is O($k^n$)***
***while no Time gain → use DFS***

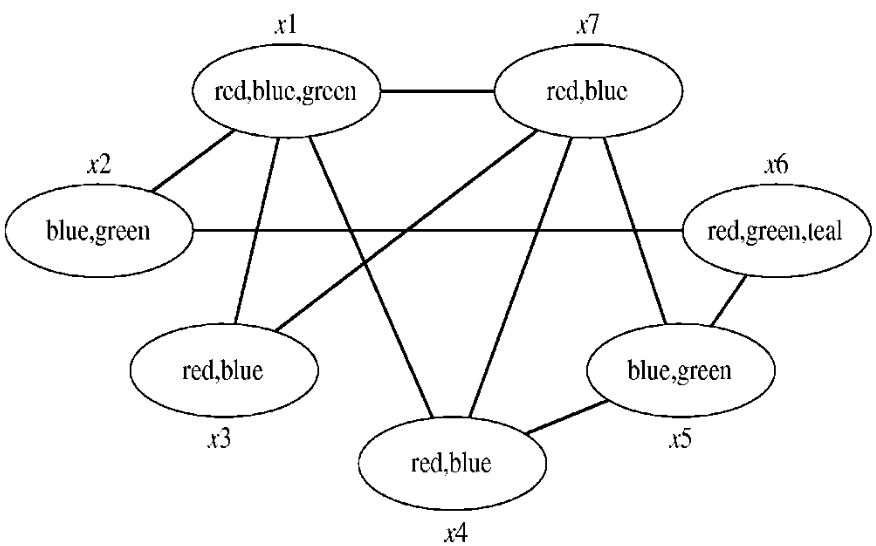
# Improving Backtracking

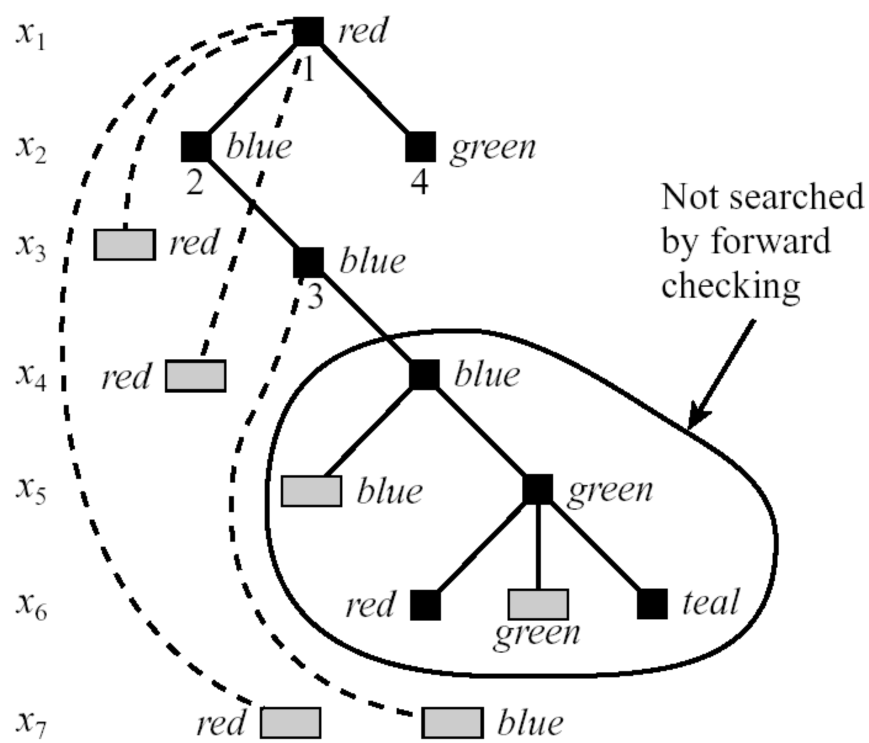
- Before search: (reducing the search space)
  - Arc-consistency, path-consistency
  - Variable ordering (fixed)
- During search:
  - Look-ahead schemes:
    - value ordering,
    - variable ordering (if not fixed)
  - Look-back schemes:
    - Backjump
    - Constraint recording or learning
    - Dependency-directed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

# Look-Ahead: Value Orderings

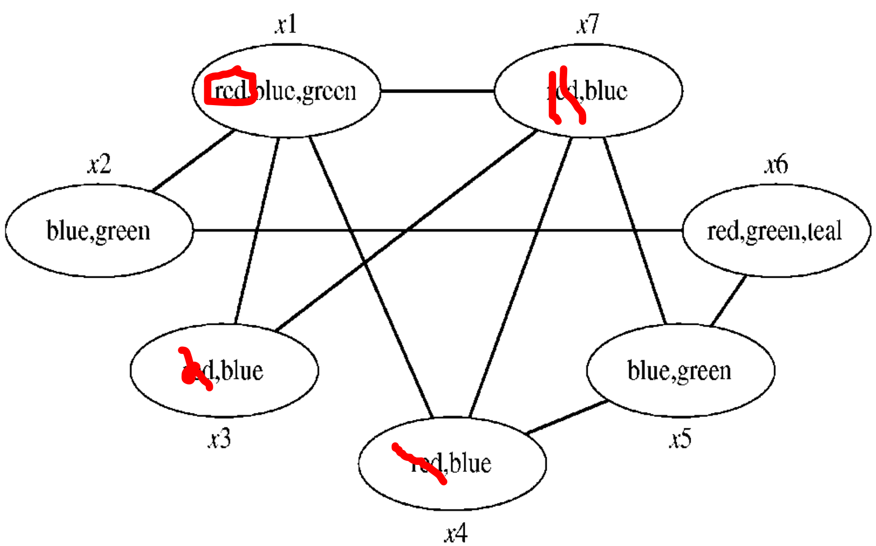
- Intuition:
  - Choose value least likely to yield a dead-end
  - Approach: apply constraint propagation at each node in the search tree
- Forward-checking
  - (check each unassigned variable separately
- Maintaining arc-consistency (MAC)
  - (apply full arc-consistency)
- Full look-ahead
  - One pass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

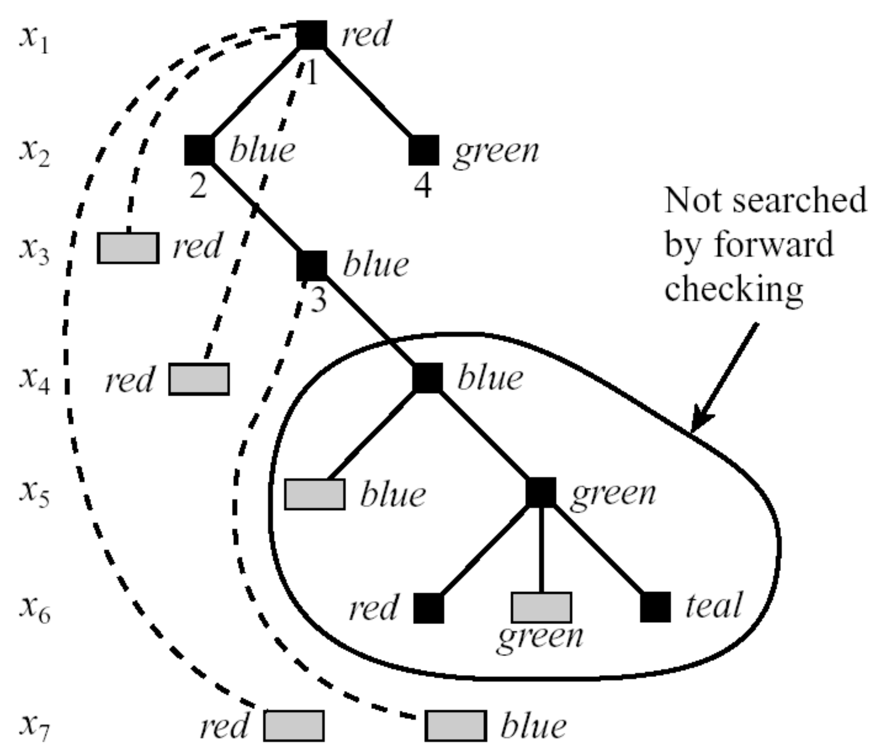
# Forward-Checking for Value Ordering



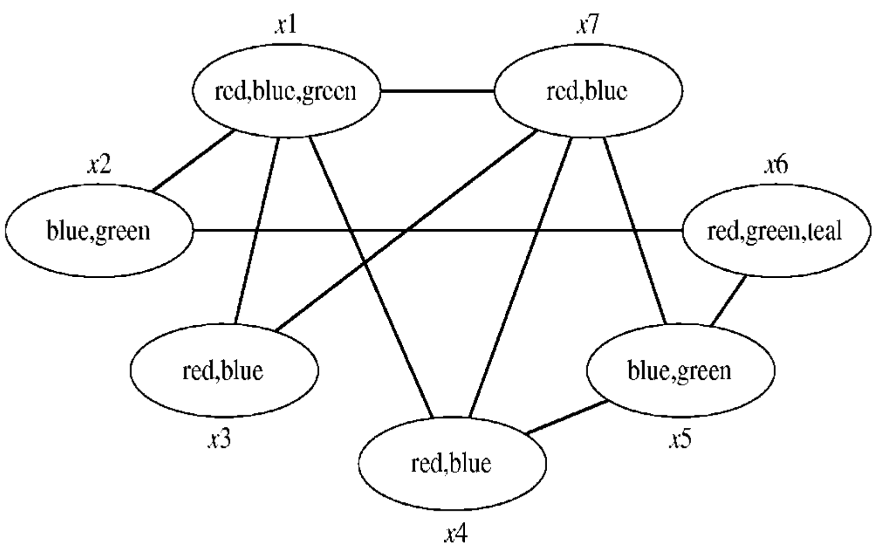
# Forward-Checking for Value Ordering

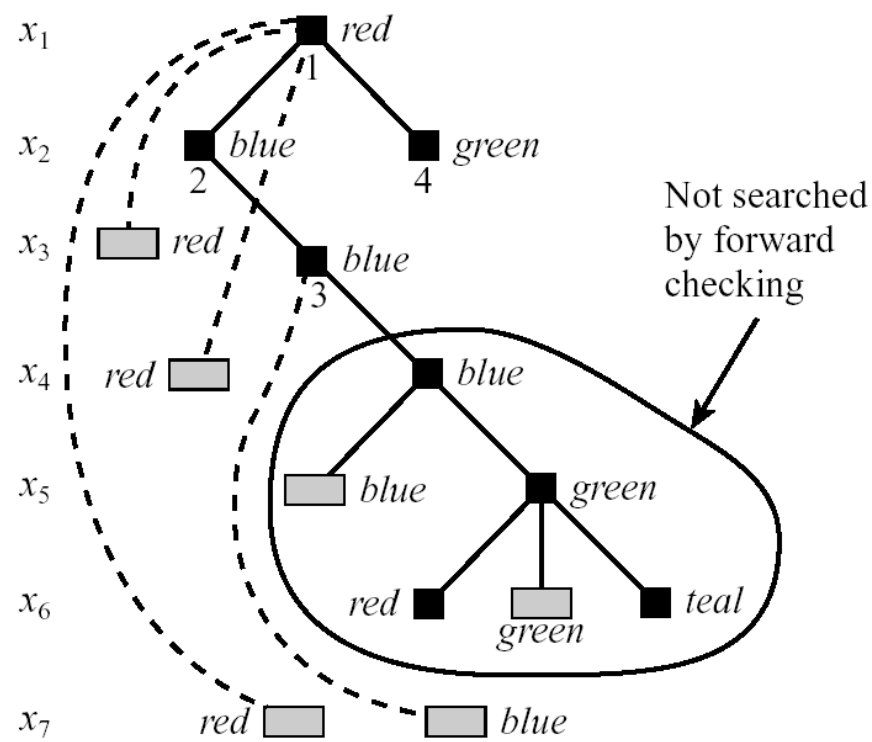



***FW overhead:*** $O(ek^2)$

# Forward-Checking, Variable Ordering

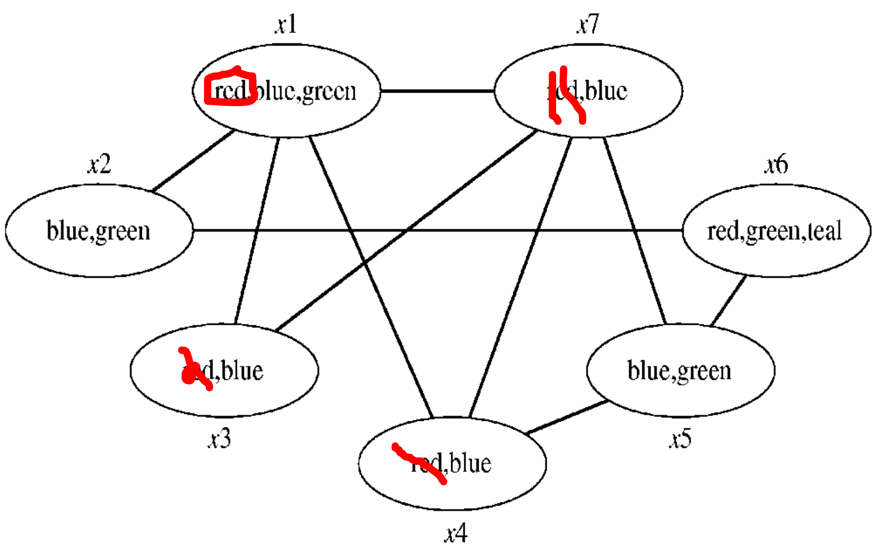


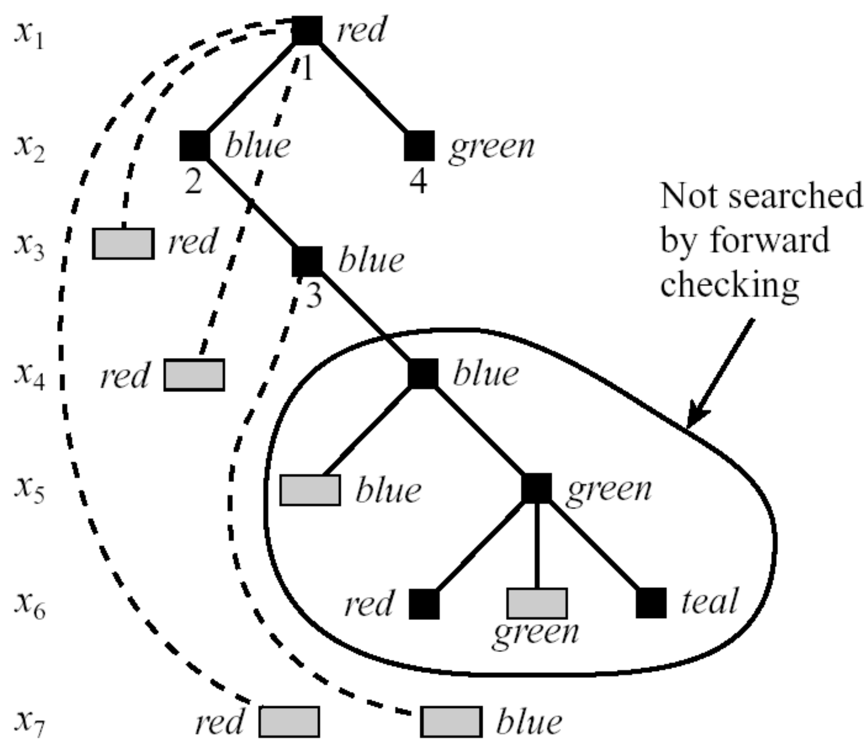




***FW overhead:*** $O(ek^2)$

# Forward-Checking, Variable Ordering

***After X1 = red choose X3 and not X2***

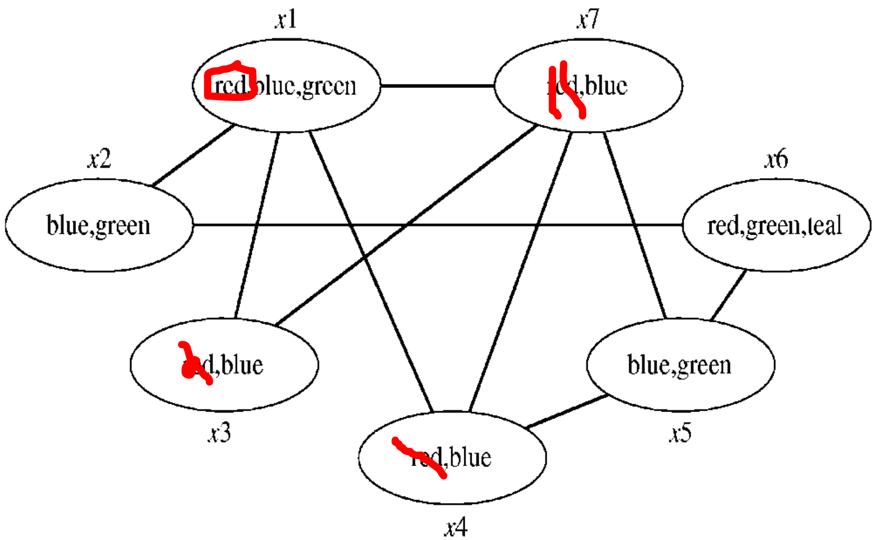


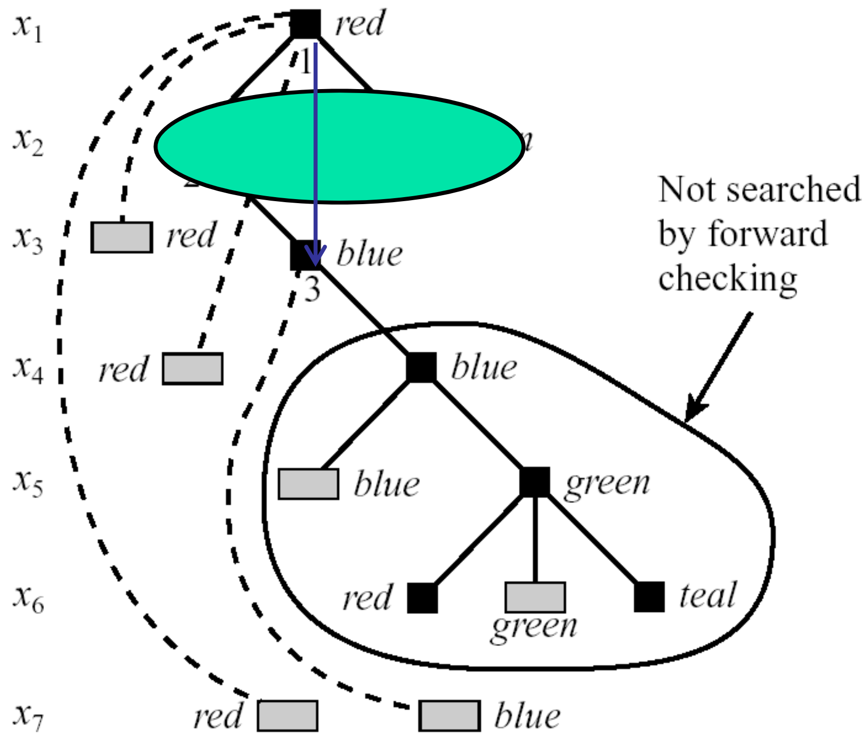




***FW overhead:*** $O(ek^2)$

# Forward-Checking, Variable Ordering

***After X1 = red choose X3 and not X2***

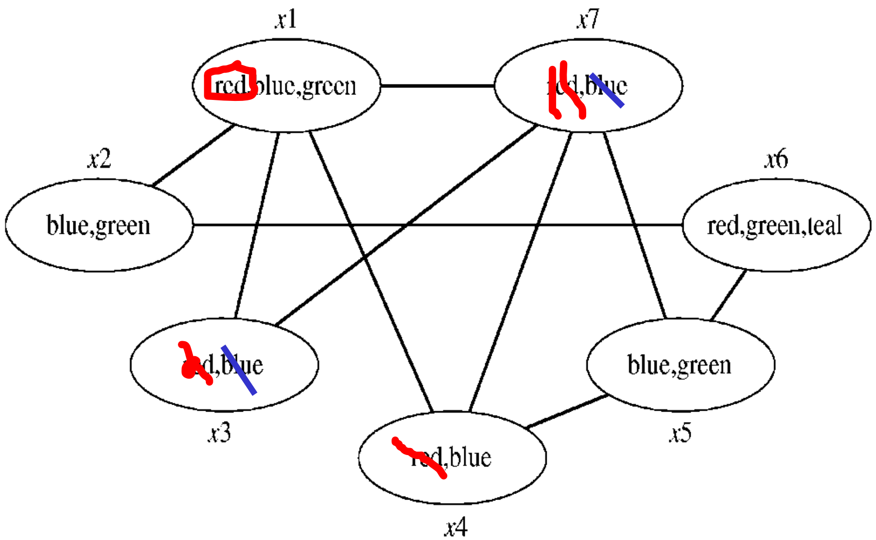


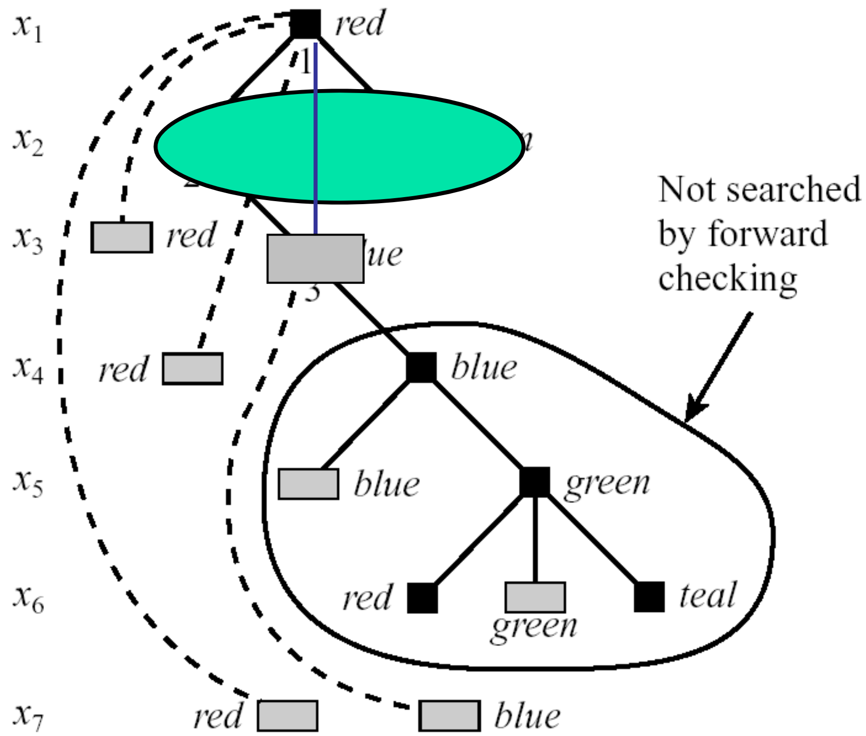




***FW overhead:*** $O(ek^2)$

# Forward-Checking, Variable Ordering

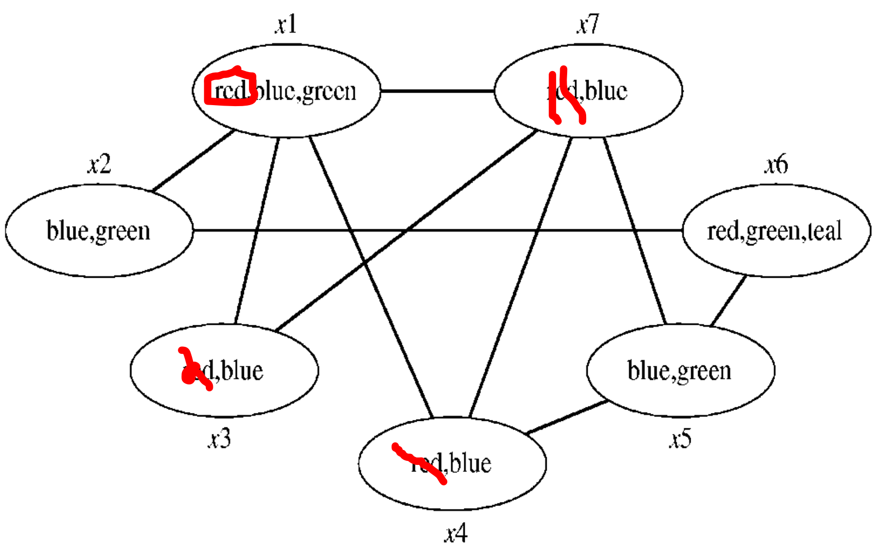
***After X1 = red choose X3 and not X2***

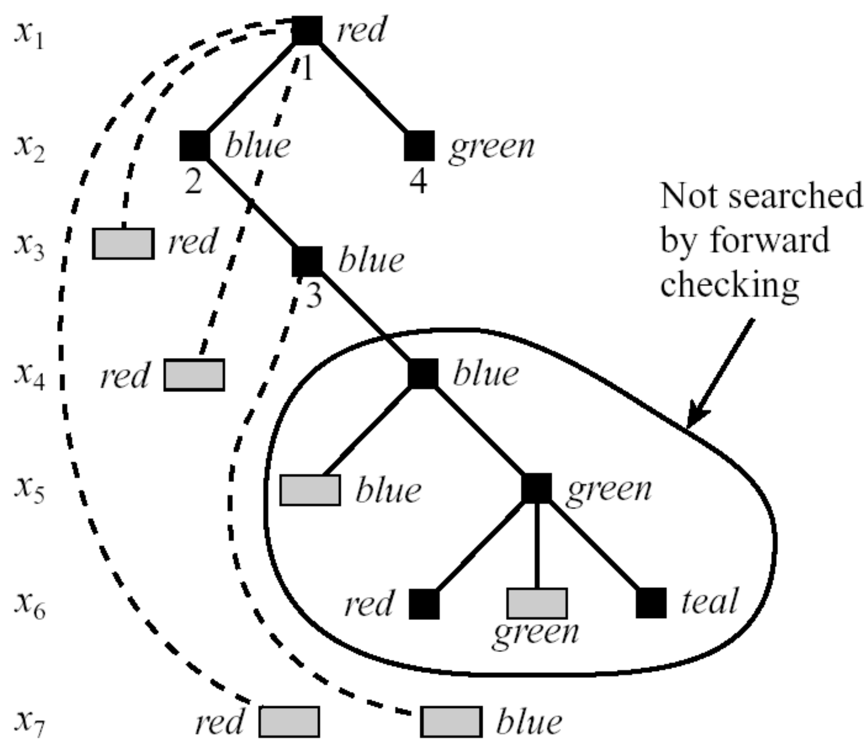



***FW overhead:*** $O(ek^2)$

# Arc-consistency for Value Ordering



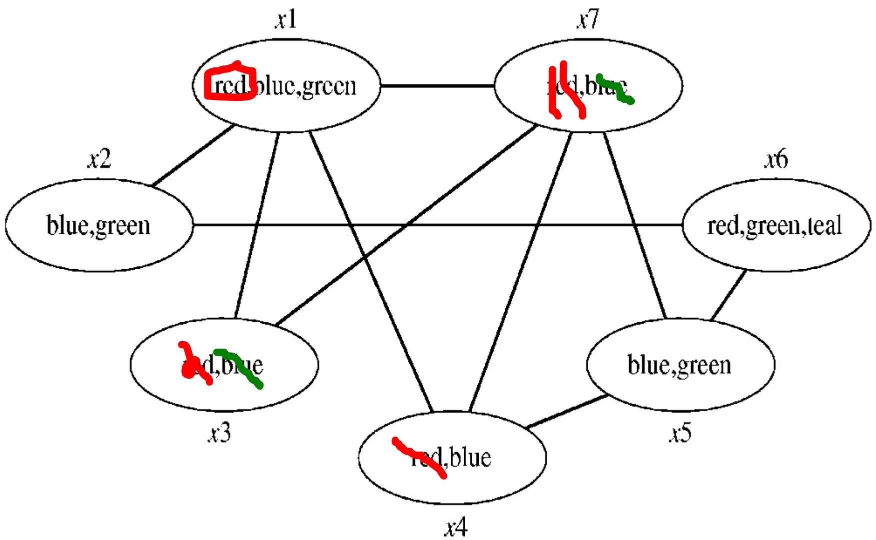



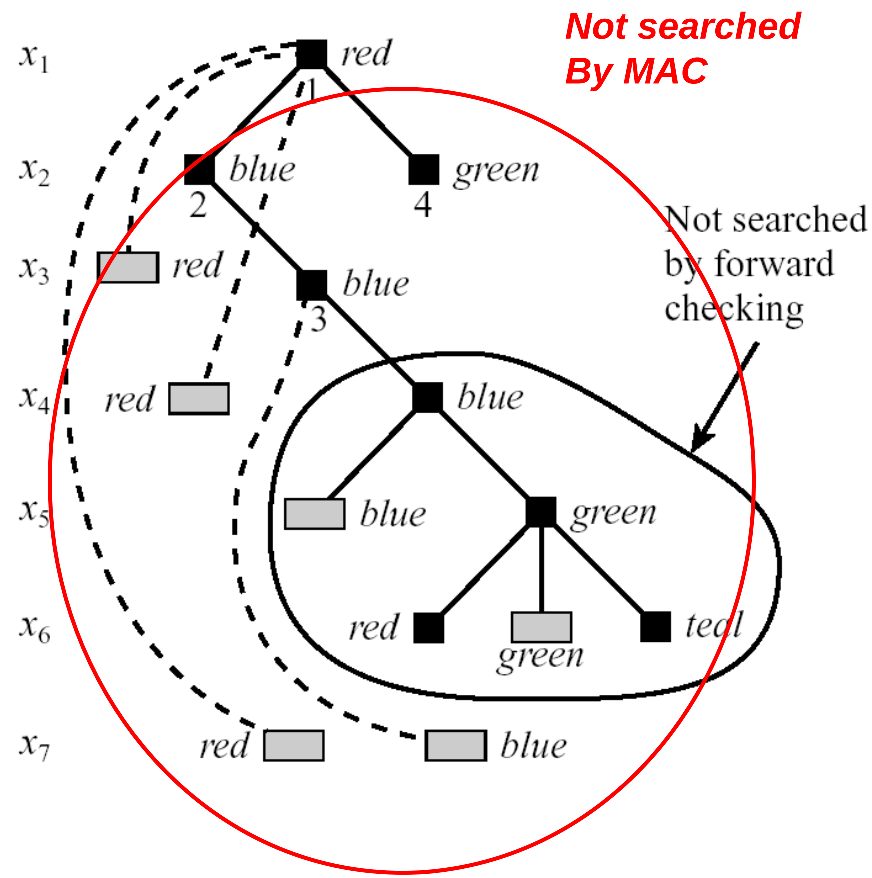

***FW overhead:*** $O(ek^2)$

***MAC overhead:*** $O(ek^3)$

# Arc-Consistency for Value Ordering

*Arc-consistency prunes x1=red*
*Prunes the whole tree*



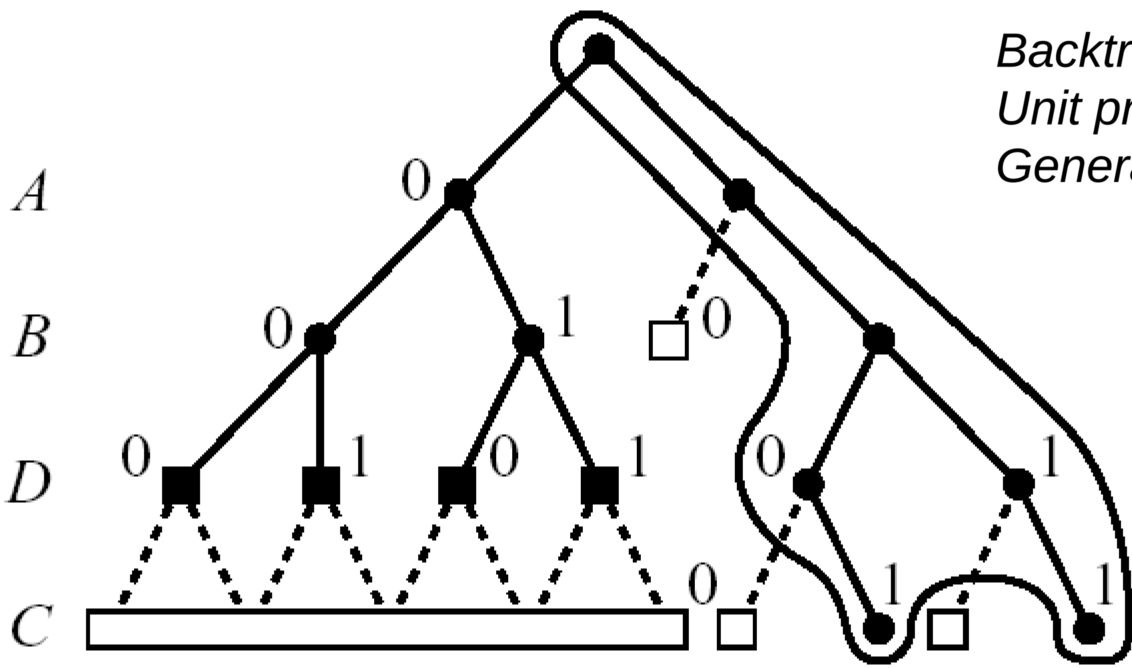

***FW overhead:*** $O(ek^2)$

***MAC overhead:*** $O(ek^3)$

# 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*

# 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  (is that true?)
- Numberjeck (in Python)

# 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 learning, Deep-learning, shallow learning
  - good-recording
  - Clause learning

# Look-Back: Backjumping

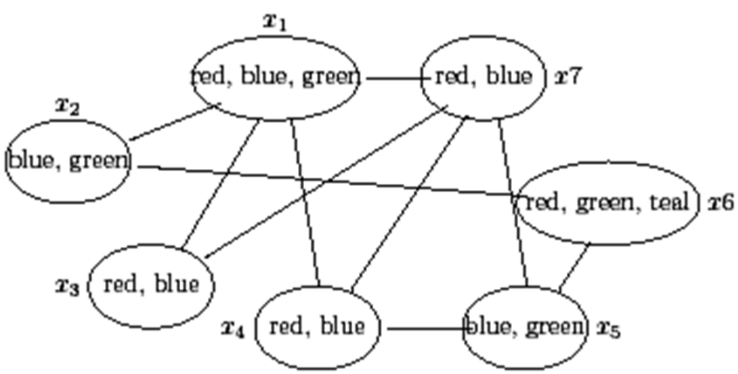
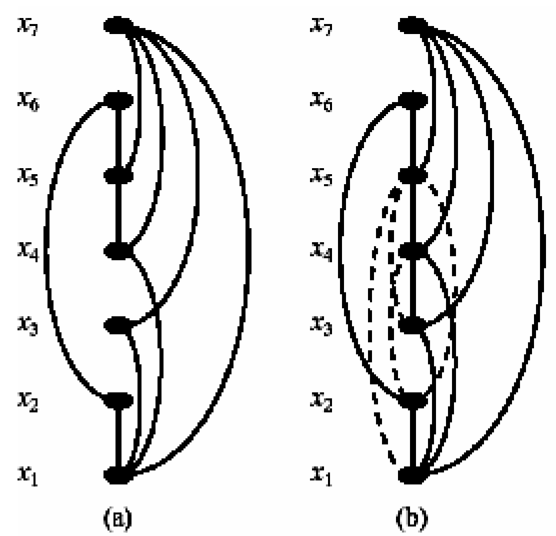



Figure 6.1: A modified coloring problem.

- (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)

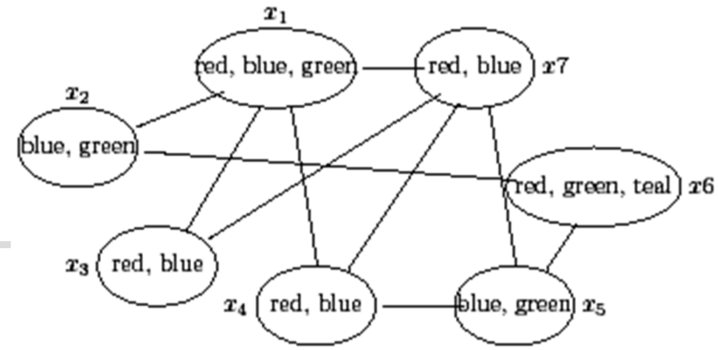
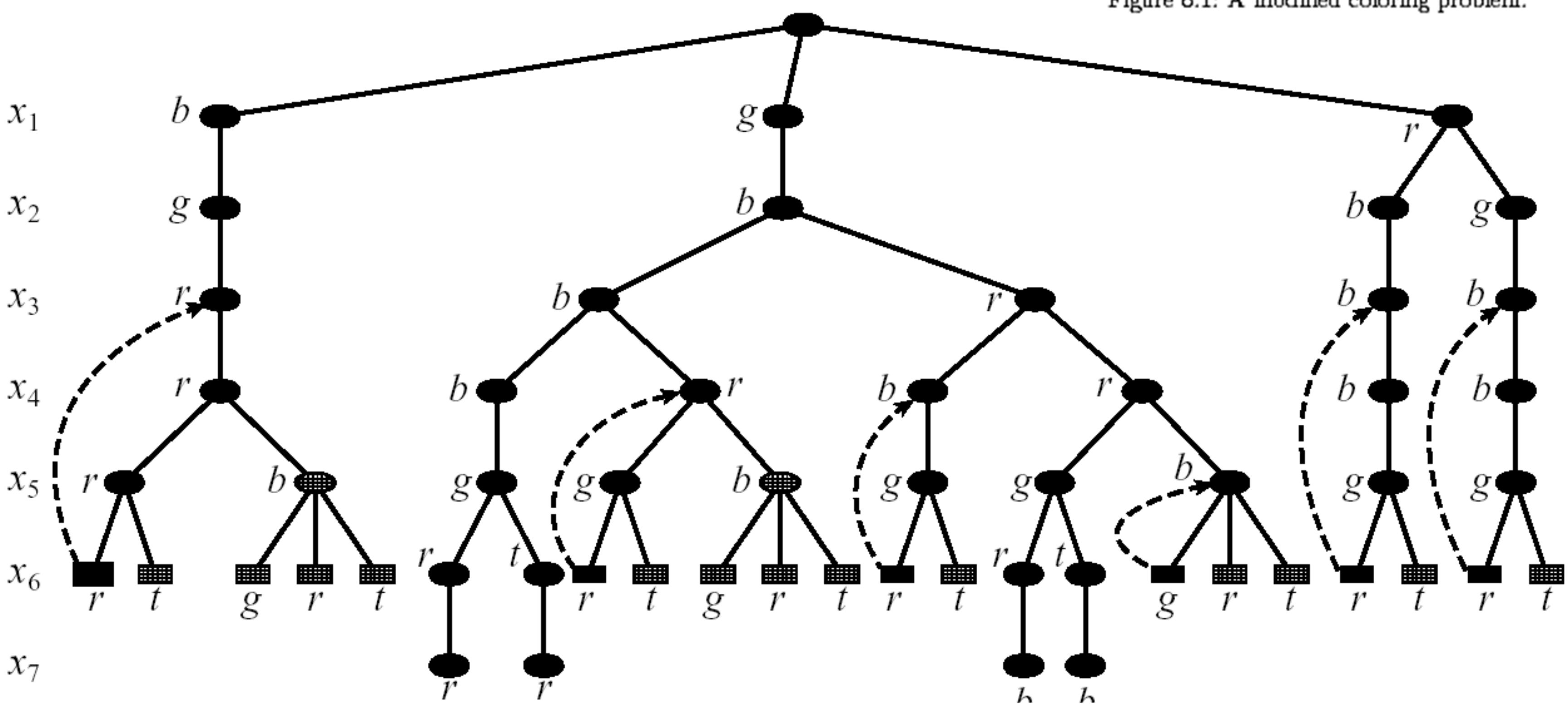



Figure 6.1: A modified coloring problem.




**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)

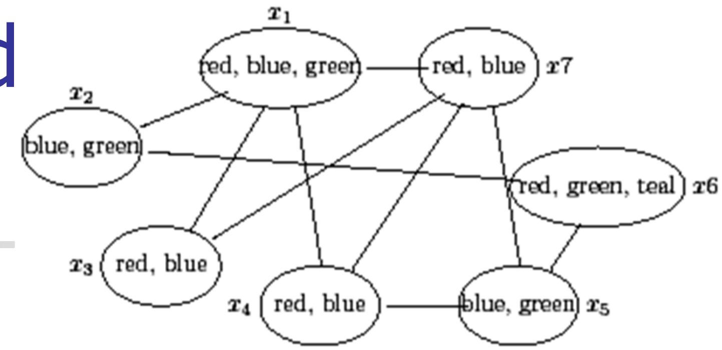
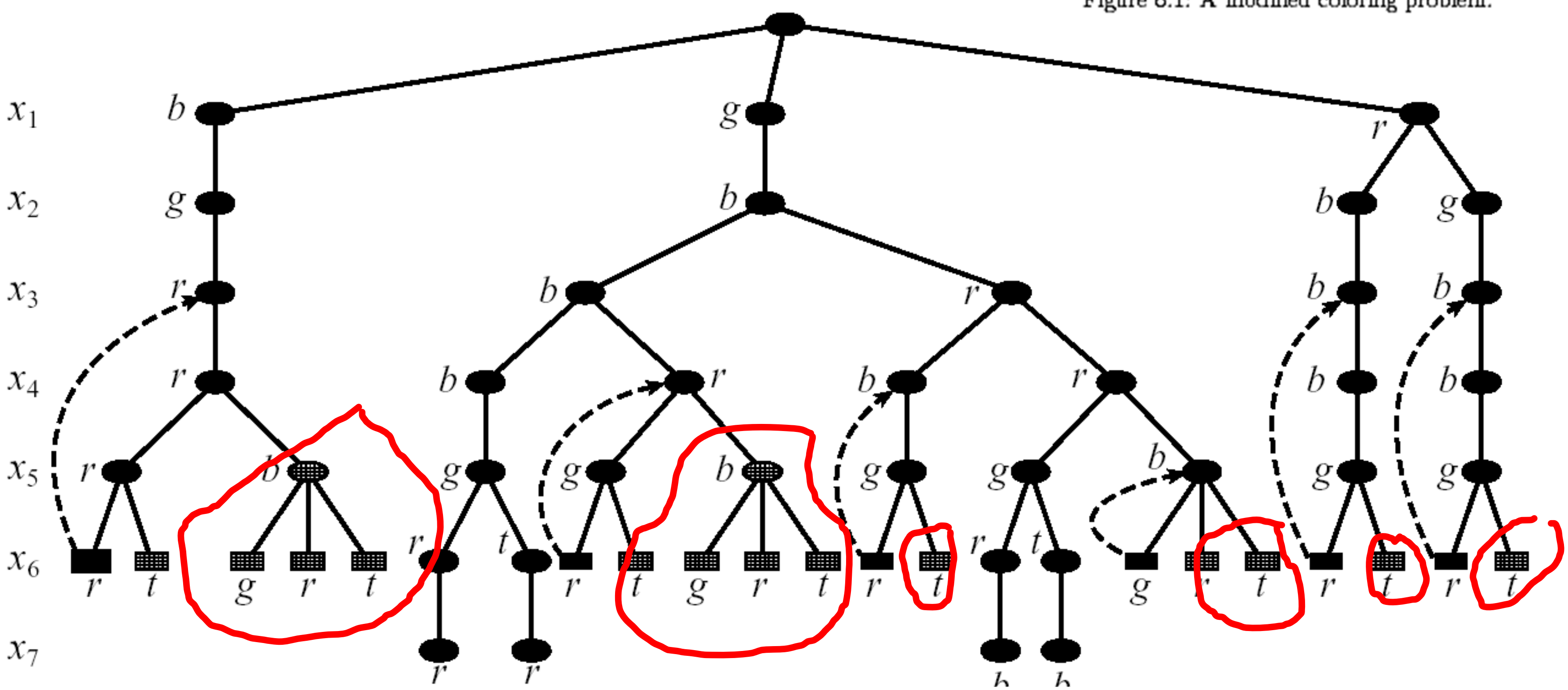



Figure 6.1: A modified coloring problem.




**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:

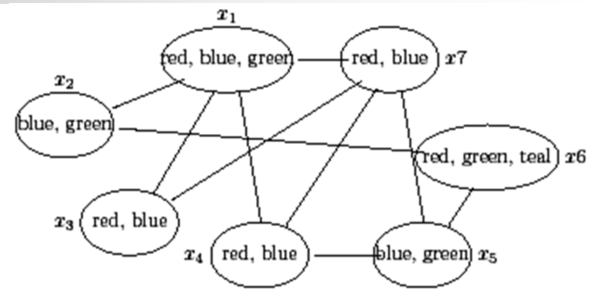
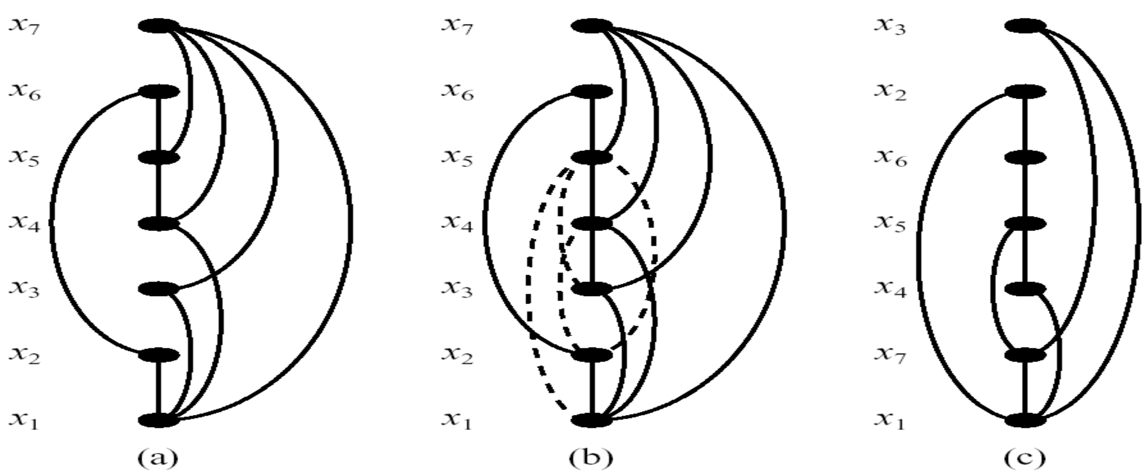


Figure 6.1: A modified coloring problem.



# 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.

# 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).

# Graph-based Backjumping on DFS ordering

- Example: d = $x_1, x_2, x_3, x_4, x_5, x_6, x_7$
- Constraints: (6,7)(5,2)(2,3)(5,7)(2,7)(2,1)(2,3)(1,4)3,4)
- Rule: go back to parent. No need to maintain parent set

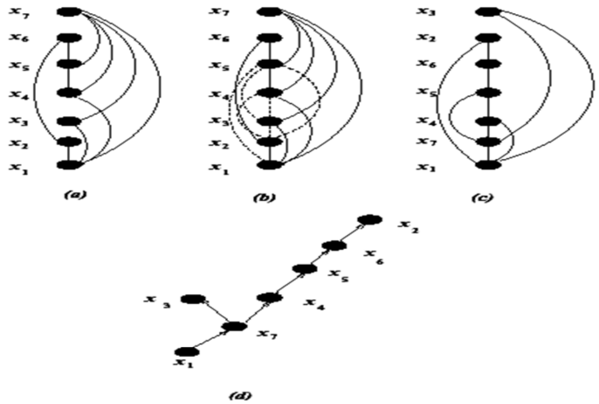


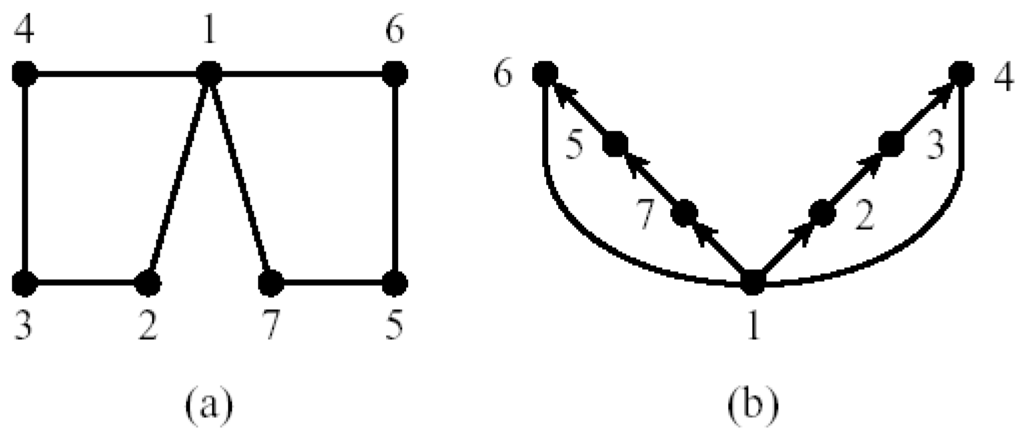
Figure 6.6: Several ordered constraint graphs of the problem in Figure 6.1: (a) along ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, (b) the induced graph along $d_1$, (c) along ordering $d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3)$, and (d) a DFS spanning tree along ordering $d_2$.

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

# Backjumping Styles

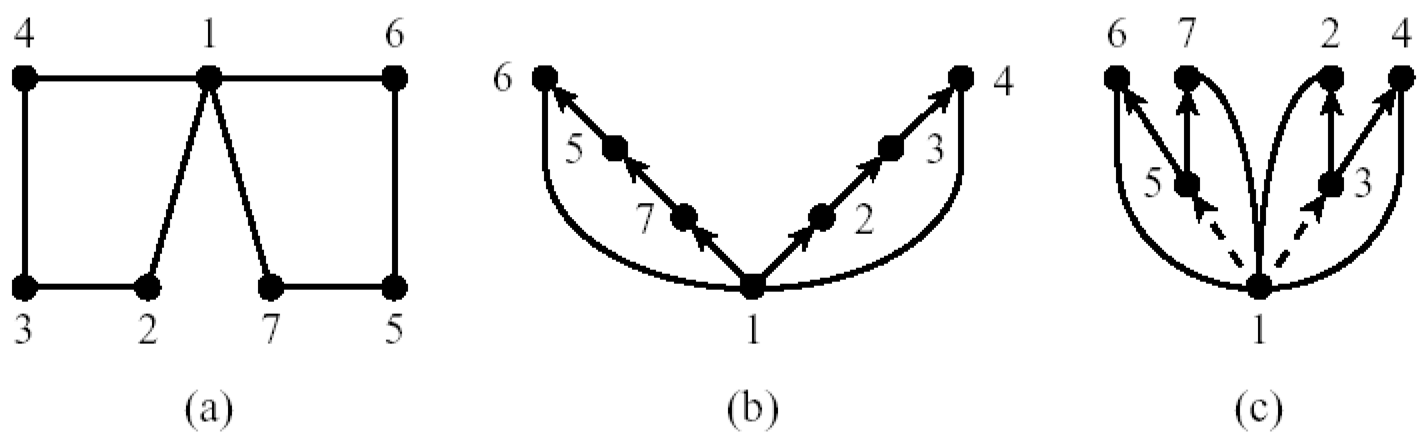
- Jump at leaf only (Gaschnig 1977)
  - Context-based
- Graph-based (Dechter, 1990)
  - Jumps at leaf and internal dead-ends, graph information
- Conflict-directed (Prosser 1993)
  - Context-based, jumps at leaf and internal dead-ends

# DFS of graph and induced graphs




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

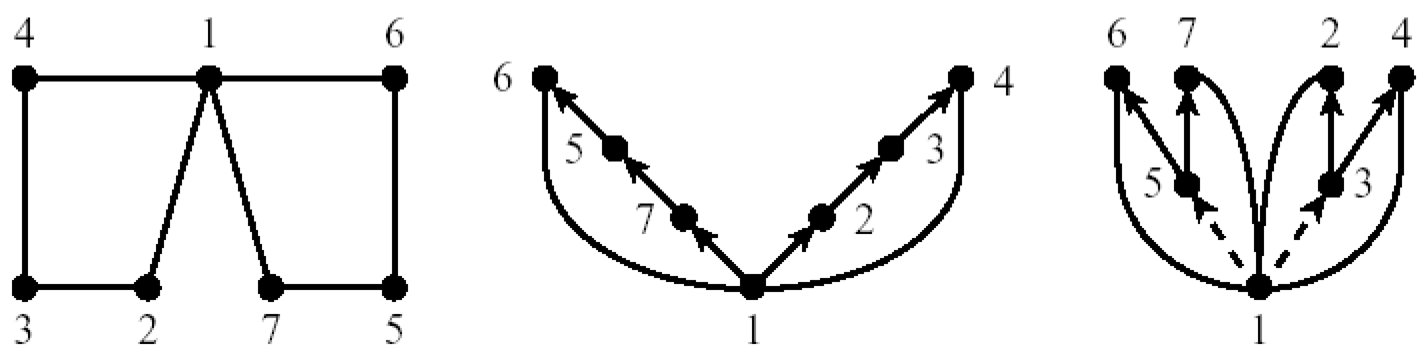
# Complexity of Backjumping Uses Pseudo-Tree Analysis




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

# Complexity of Backjumping

***Graph-based and conflict-based backjumpint***

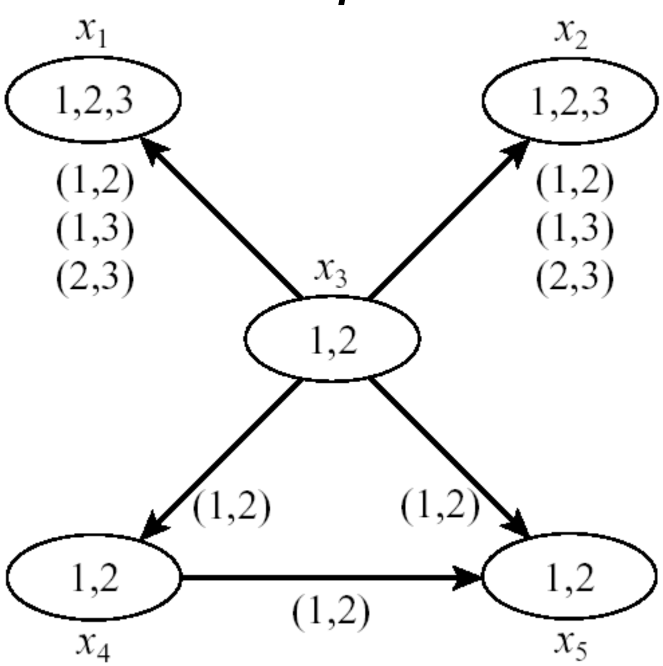



(a) (b) (c)

- *Simple: always jump back to parent in pseudo tree*
- *Complexity for csp: exp(w*log n), exp(m), m= depth*
- *From exp(n) to exp(w*logn) while linear space*
- *(proof details: exercise)*

# Look-back: NoGood Learning

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

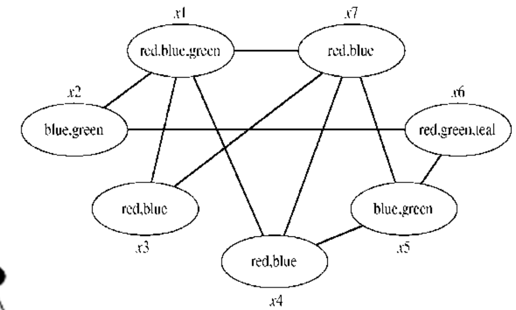



- (x1=2,x2=2,x3=1,x4=2) is a dead-end
- Conflicts to record:
  - (x1=2,x2=2,x3=1,x4=2) 4-ary
  - (x3=1,x4=2) binary
  - (x4=2) unary

# 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.

# No-good Learning Example



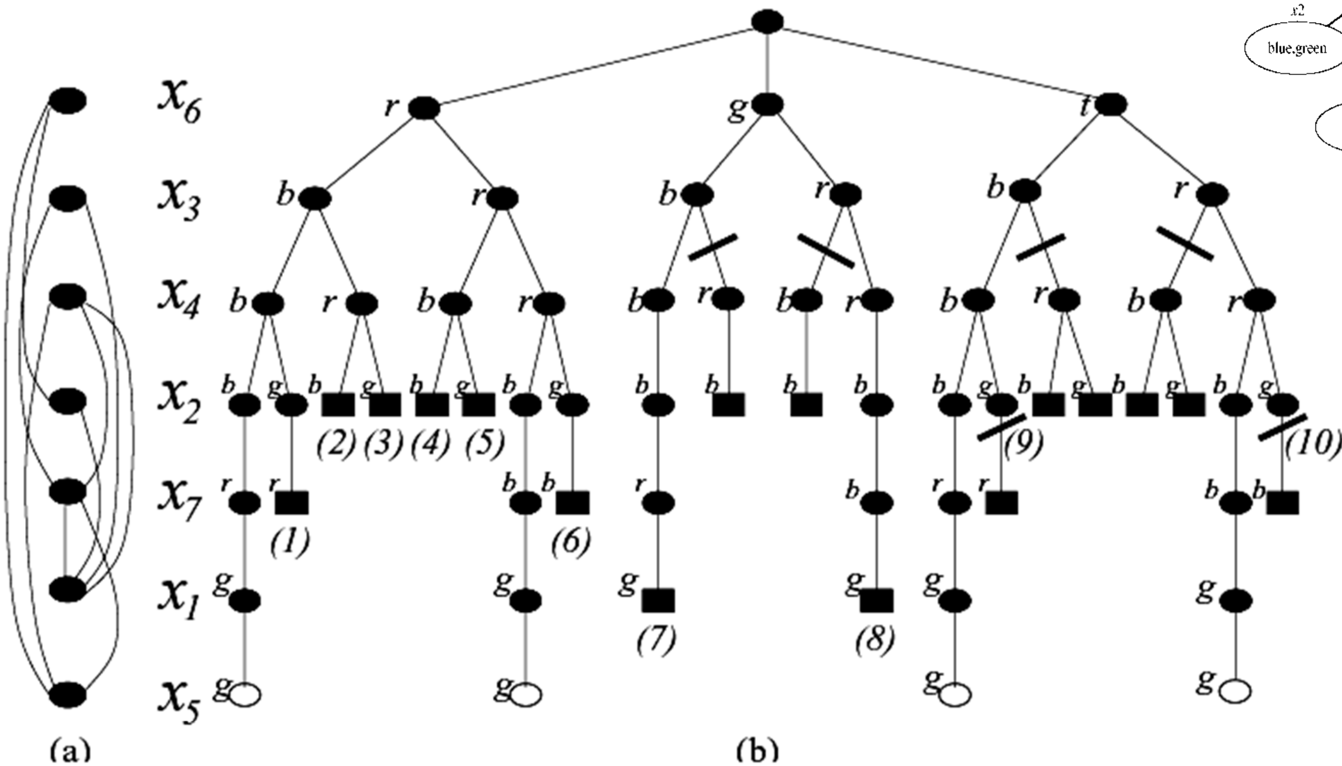




Figure 6.9: The search space explicated by backtracking on the CSP from Figure 6.1, using the variable ordering $(x_6, x_3, x_4, x_2, x_7, x_1, x_5)$ and the value ordering (*blue, red, green, teal*). Part (a) shows the ordered constraint graph, part (b) illustrates the search space. The cut lines in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

# 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: CDCL (Conflict-Directed Clause Learning)

# Deep Learning

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

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

https://medium.com/a-computer-of-ones-own/rina-dechter-deep-learning-pioneer-e7e9ccc96c6e

# 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.*

# Graph-Based Learning Scenarios Internal Deadend at X4, conflicts?

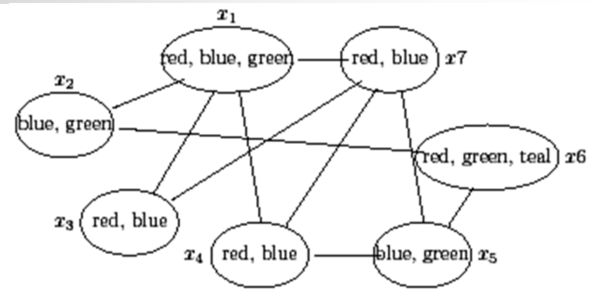
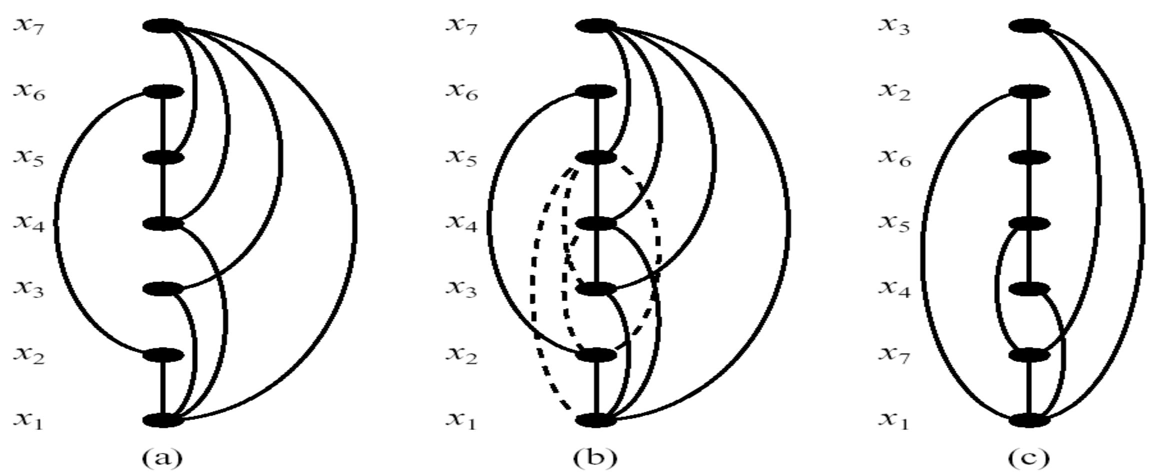



Figure 6.1: A modified coloring problem.

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



# Complexity of Backtrack-Learning for CSP

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

For graph-based learning 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 $O(nk^{w^*(d)})$

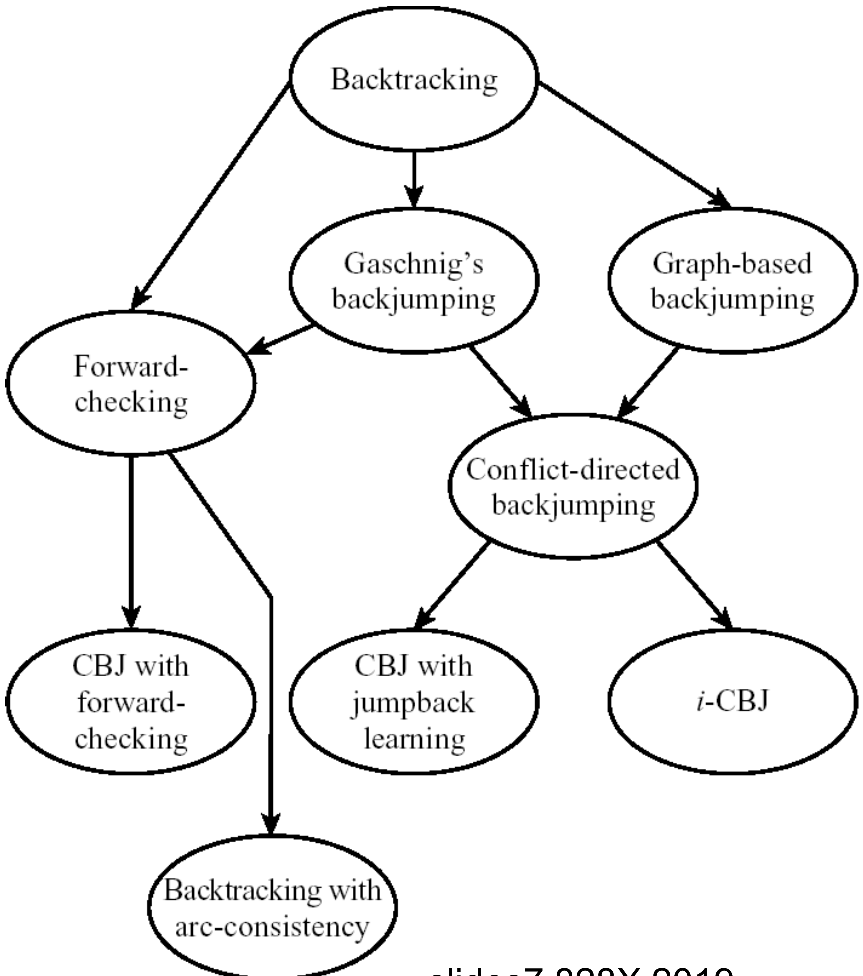
Time complexity is $O(n^2 \cdot k^{w^*(d)+1})$

# Proof of Complexity NG learning

**Theorem 6.7.5** *Let $d$ be an ordering of a constraint graph, and let $w^*(d)$ be its induced width. Any backtracking algorithm using ordering $d$ with graph-based learning has a space complexity of $O(n \cdot (k)^{w^*(d)})$ and a time complexity of $O(n^2 \cdot (2k)^{w^*(d)+1})$, where $n$ is the number of variables and $k$ bounds the domain sizes.*

**Proof:** Graph-based learning has a one-to-one correspondence between dead-ends and conflict sets. Backtracking with graph-based learning along $d$ records conflict sets of size $w^*(d)$ or less, because the dead-end variable will not be connected to more than $w^*(d)$ earlier variables by both original constraints and recorded ones. Therefore the number of dead-ends is bounded by the number of possible no-goods of size $w^*(d)$ or less. Moreover, a dead-end at a particular variable $x$ can occur at most $k^{w^*(d)}$ times after which point constraints are learned excluding all possible assignments of its induced parents. So the total number of dead-ends for backtracking with learning is $O(n \cdot k^{w^*(d)})$, yielding space complexity of $O(n \cdot k^{w^*(d)})$. Since the total number of values considered between successive dead-ends is at most $O(kn)$, the total number of values considered during backtracking with learning is $O(kn \cdot n \cdot k^{w^*(d)}) = O(n^2 \cdot k^{w^*(d)+1})$. Since each value requires testing all constraints defined over the current variable, and at most $w^*(d)$ prior variables, at most $O(2^{w^*(d)})$ constraints are checked per value test, yielding a time complexity bound of $O(n^2(2k)^{w^*(d)+1})$. □

# Relationships between various backtracking algrithms



# Moving to New Queries

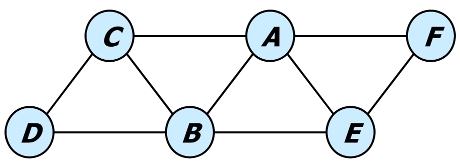
- Consistency and one solution.
- Counting
- Enumerating

# 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

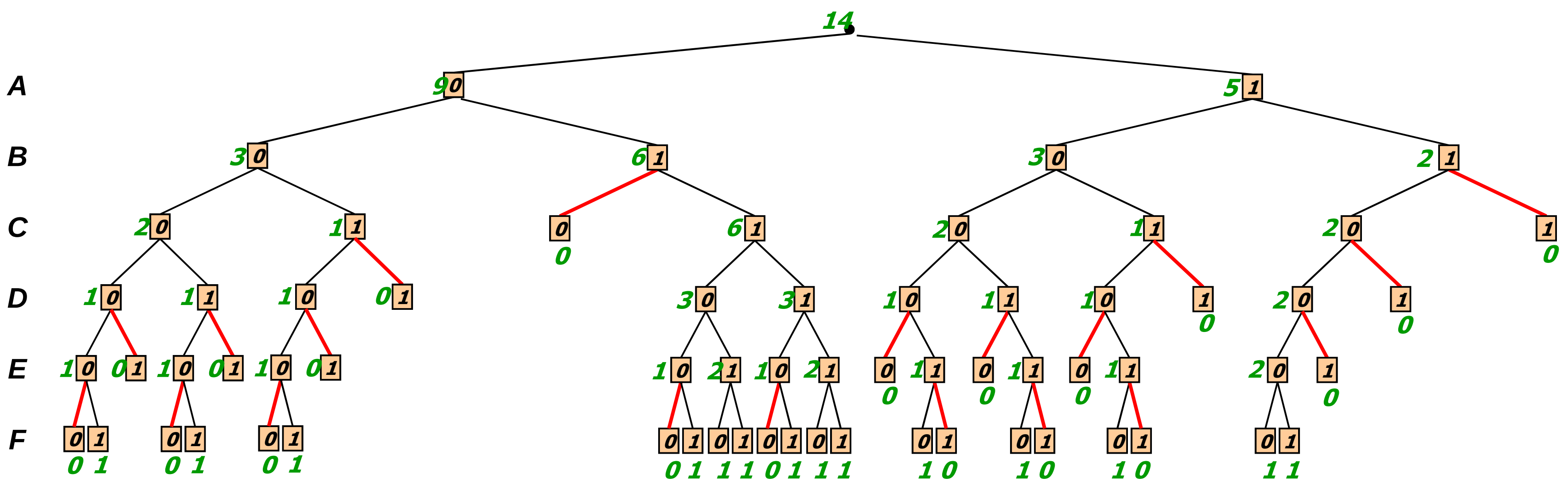



| A | B | C | $R_{ABC}$ |
|---|---|---|---|
| 0 | 0 | 0 | **1** |
| 0 | 0 | 1 | **1** |
| 0 | 1 | 0 | **0** |
| 0 | 1 | 1 | **1** |
| 1 | 0 | 0 | **1** |
| 1 | 0 | 1 | **1** |
| 1 | 1 | 0 | **1** |
| 1 | 1 | 1 | **0** |

| 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** |

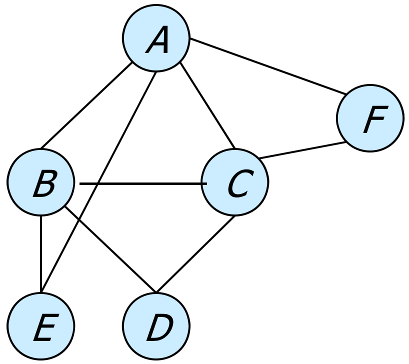



*Value of node = number of solutions below it*

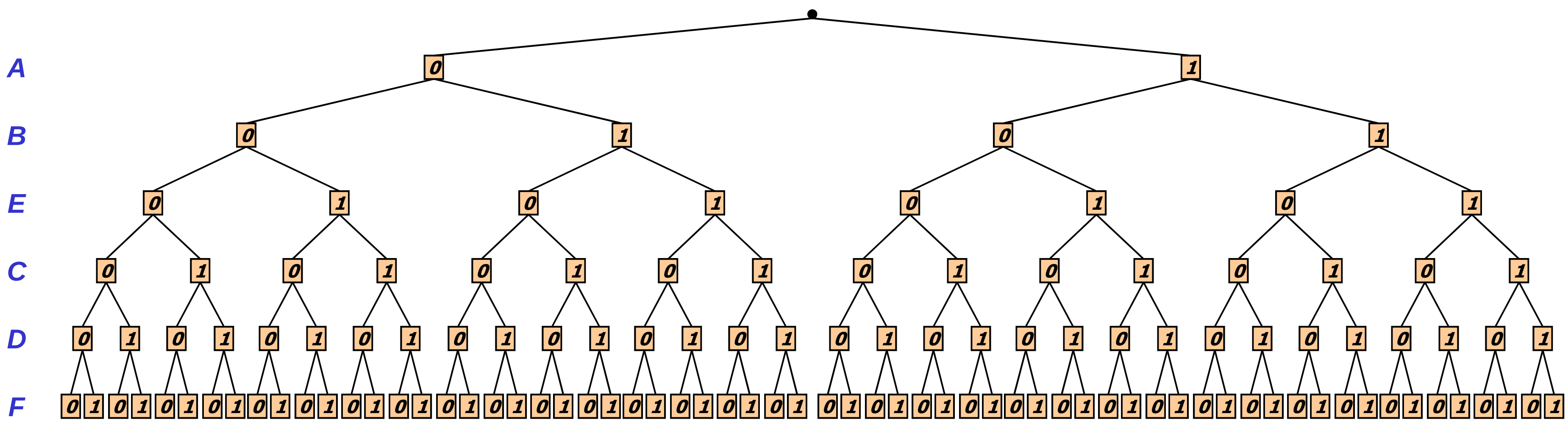
# Outline

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

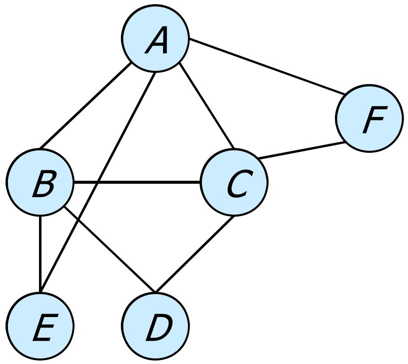
# OR Search Space



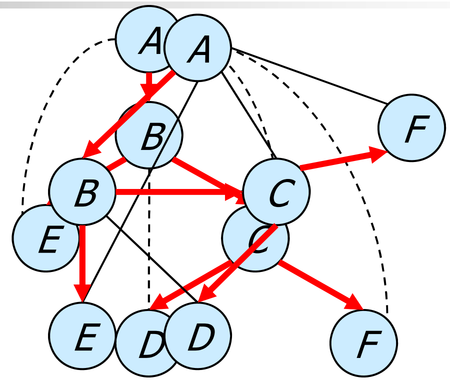

*Ordering: A B E C D F*



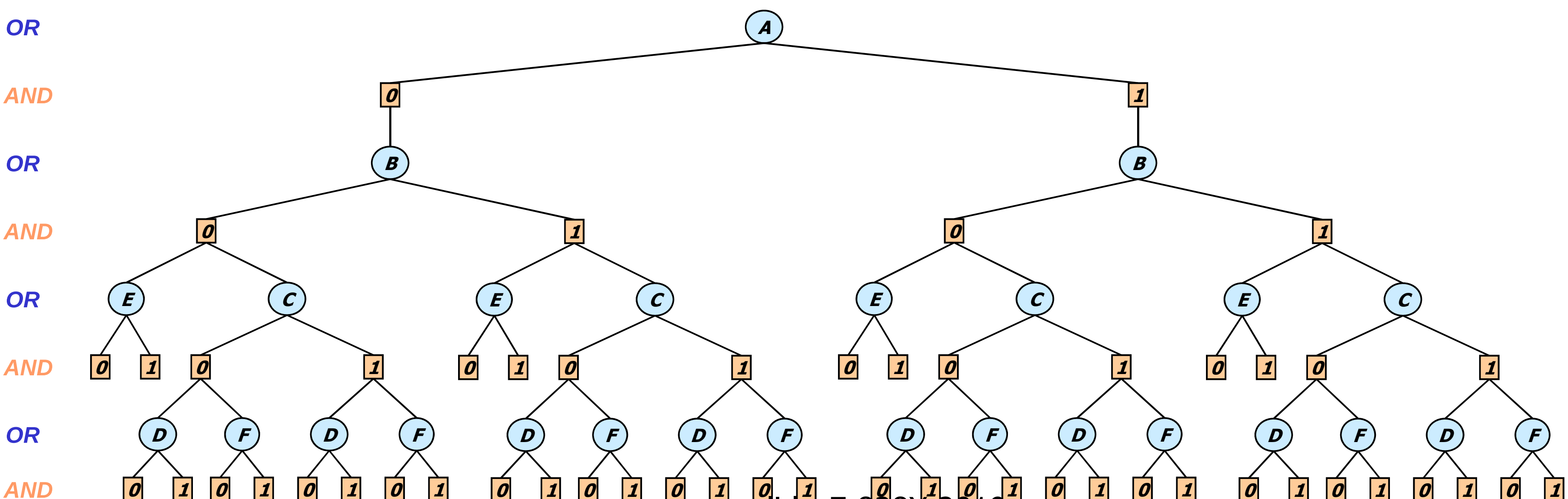
# AND/OR Search Space



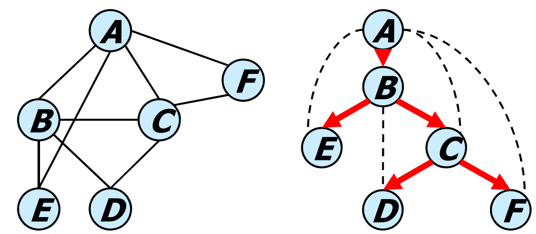

*Primal graph*

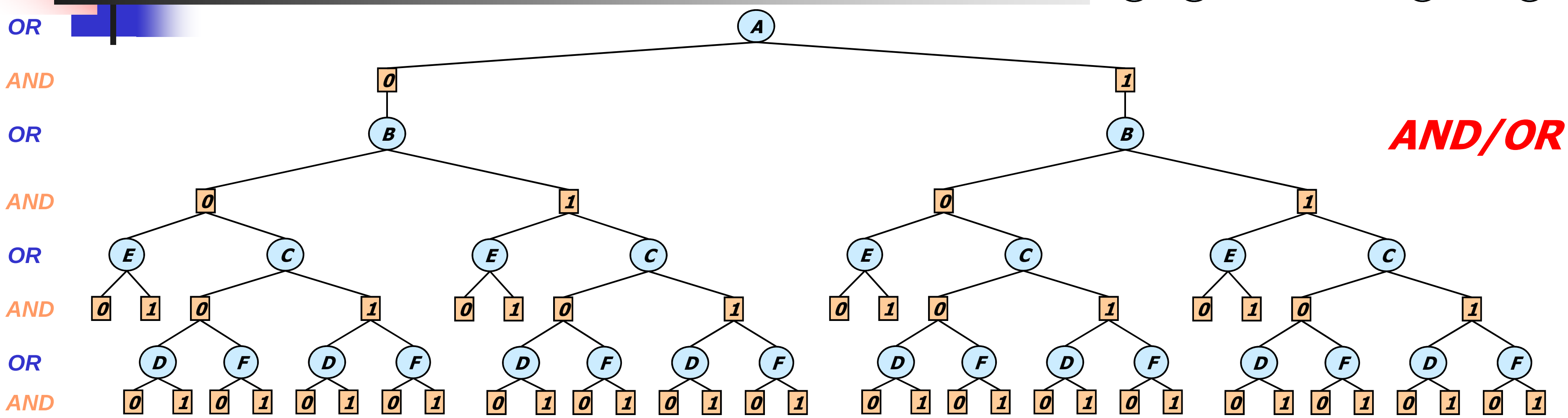



*DFS tree*

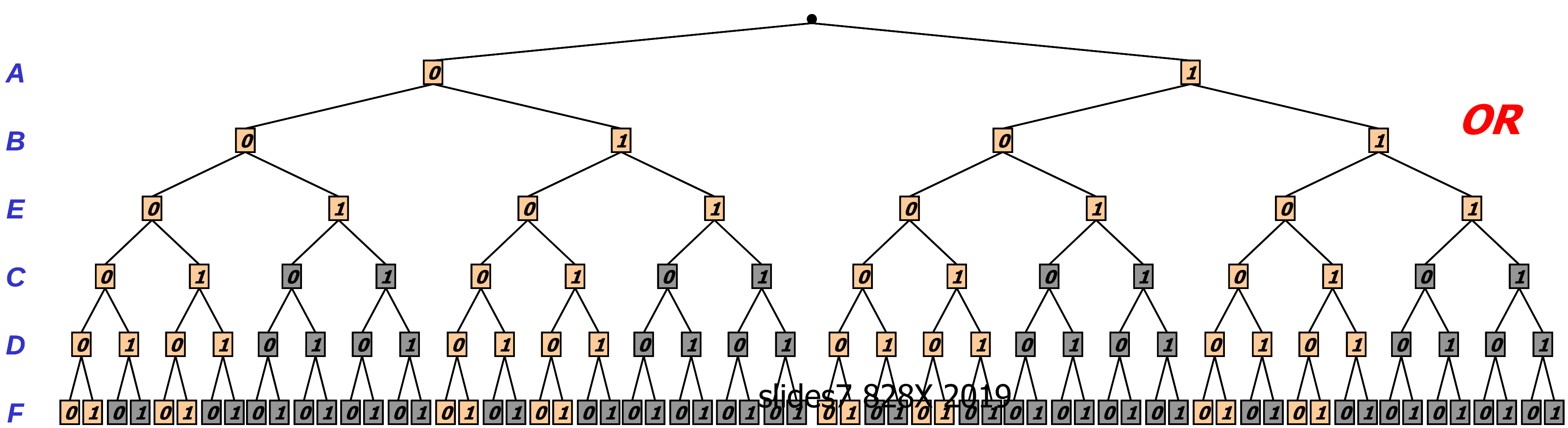


# AND/OR vs. OR



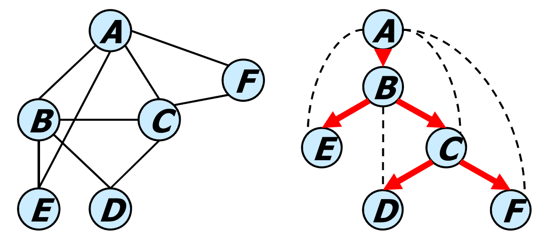



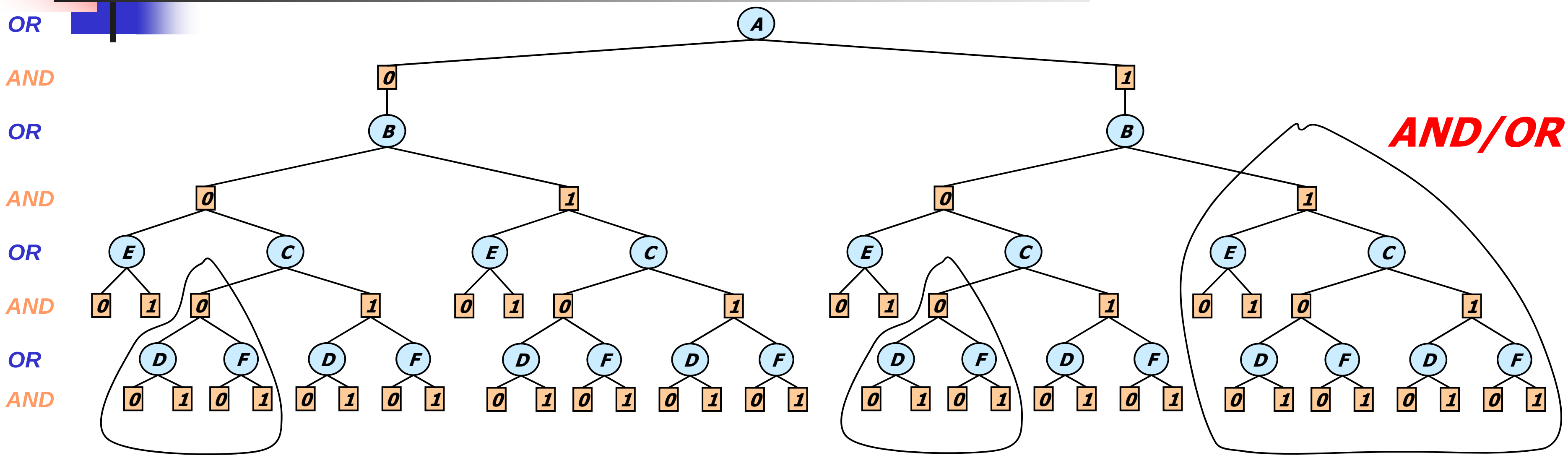

**AND/OR size: exp(4), OR size exp(6)**

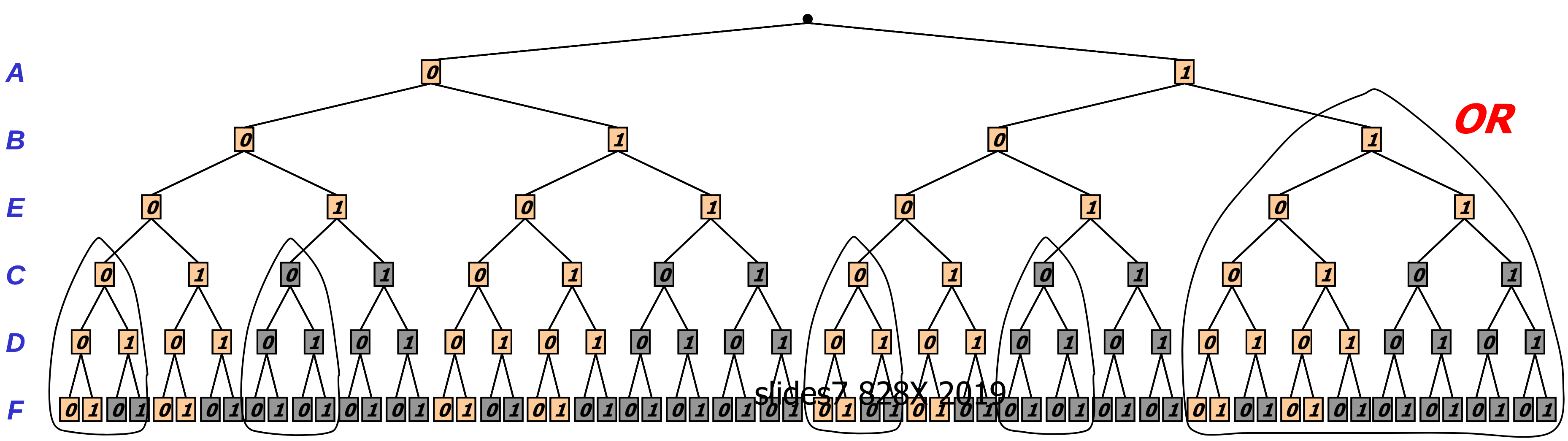


# AND/OR vs. OR

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

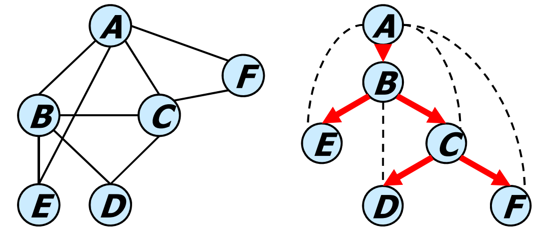


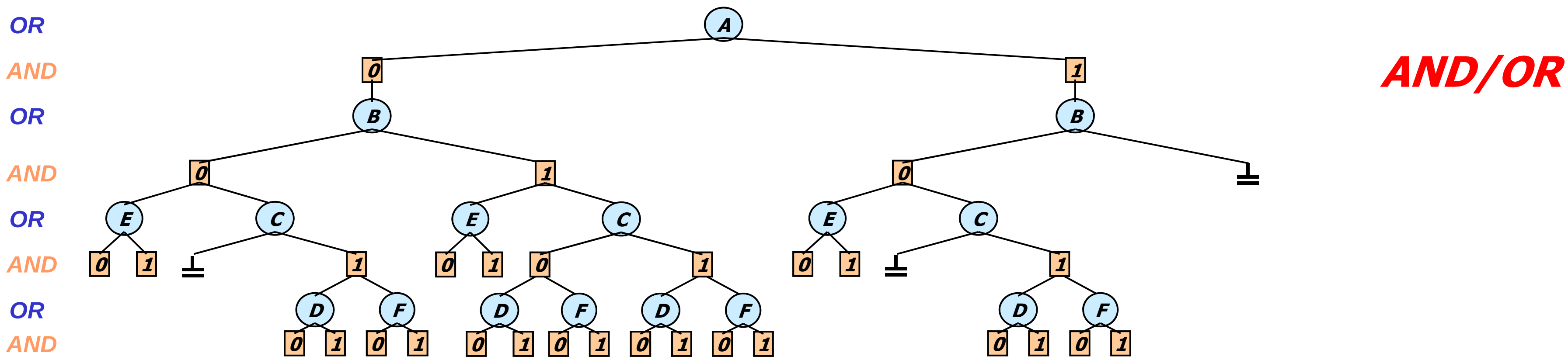



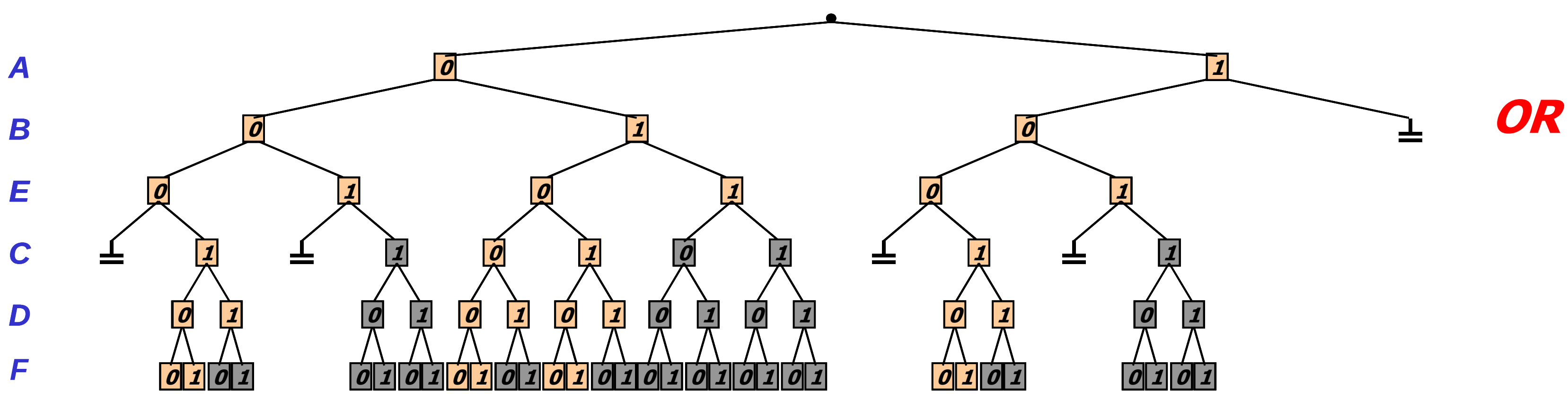



# AND/OR vs. OR

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

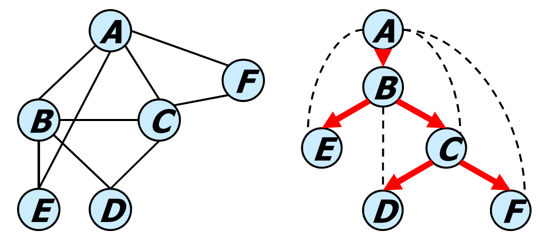


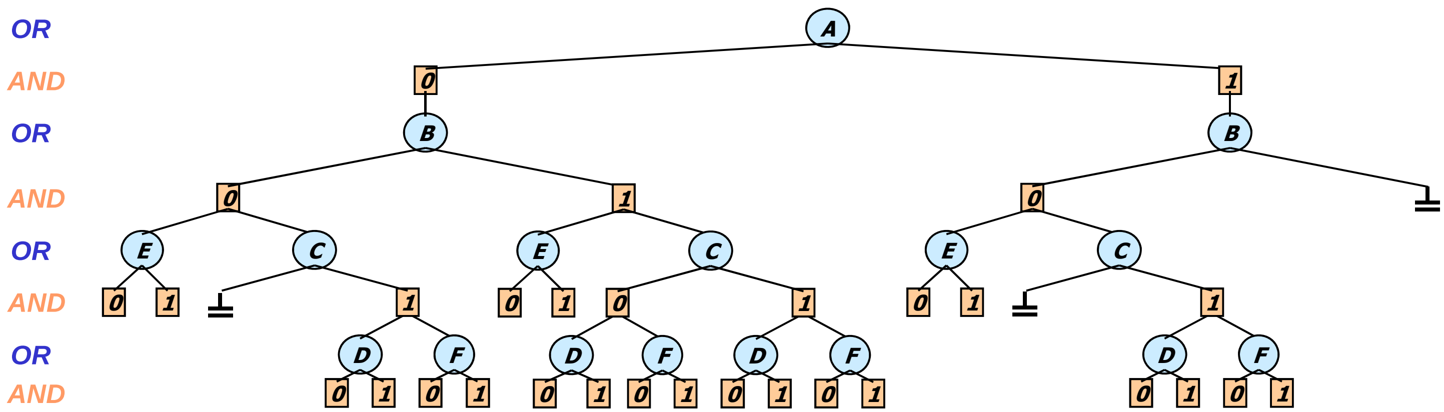



# AND/OR vs. OR

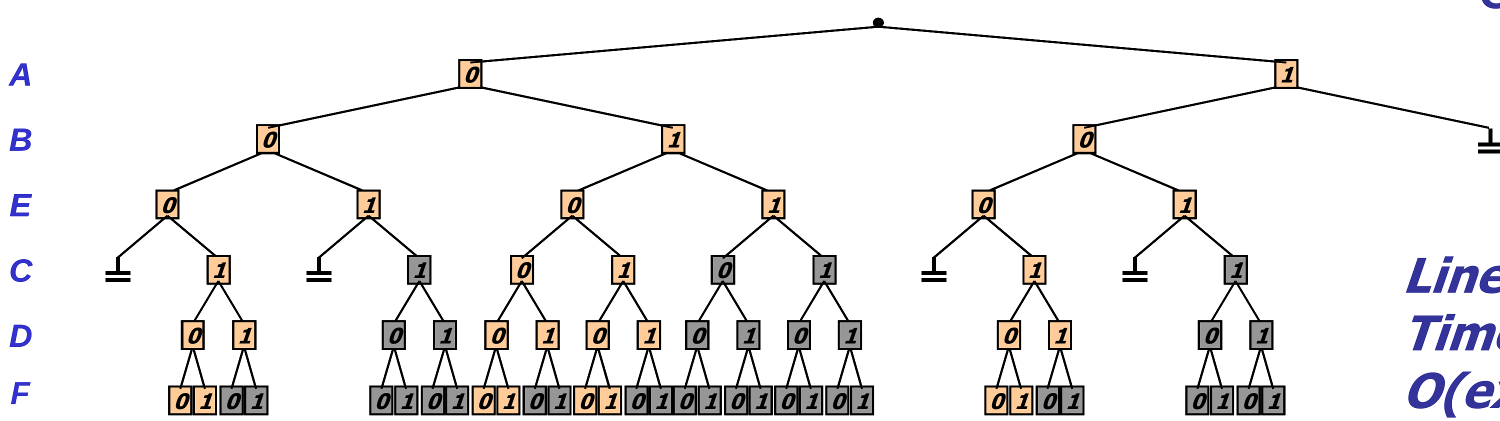
(A=1,B=1)
(B=0,C=0)




**AND/OR**

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




**OR**

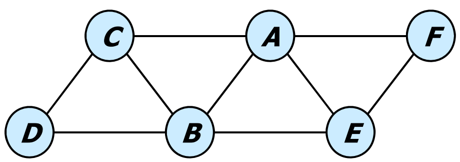
**Linear space,**
**Time:**
**O(exp(n))**

# AND/OR vs. OR Spaces

| width | depth | OR space | | AND/OR space | | |
|---|---|---|---|---|---|---|
| | | Time (sec.) | Nodes | Time (sec.) | AND nodes | OR nodes |
| 5 | 10 | 3.15 | 2,097,150 | 0.03 | **10,494** | 5,247 |
| 4 | 9 | 3.13 | 2,097,150 | 0.01 | **5,102** | 2,551 |
| 5 | 10 | 3.12 | 2,097,150 | 0.03 | **8,926** | 4,463 |
| 4 | 10 | 3.12 | 2,097,150 | 0.02 | **7,806** | 3,903 |
| 5 | 13 | 3.11 | 2,097,150 | 0.10 | **36,510** | 18,255 |

*Random graphs with 20 nodes, 20 edges and 2 values per node*

# #CSP – AND/OR Search Tree

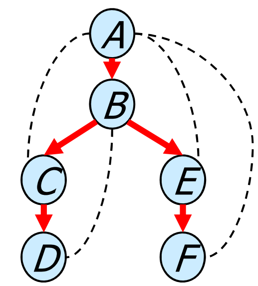


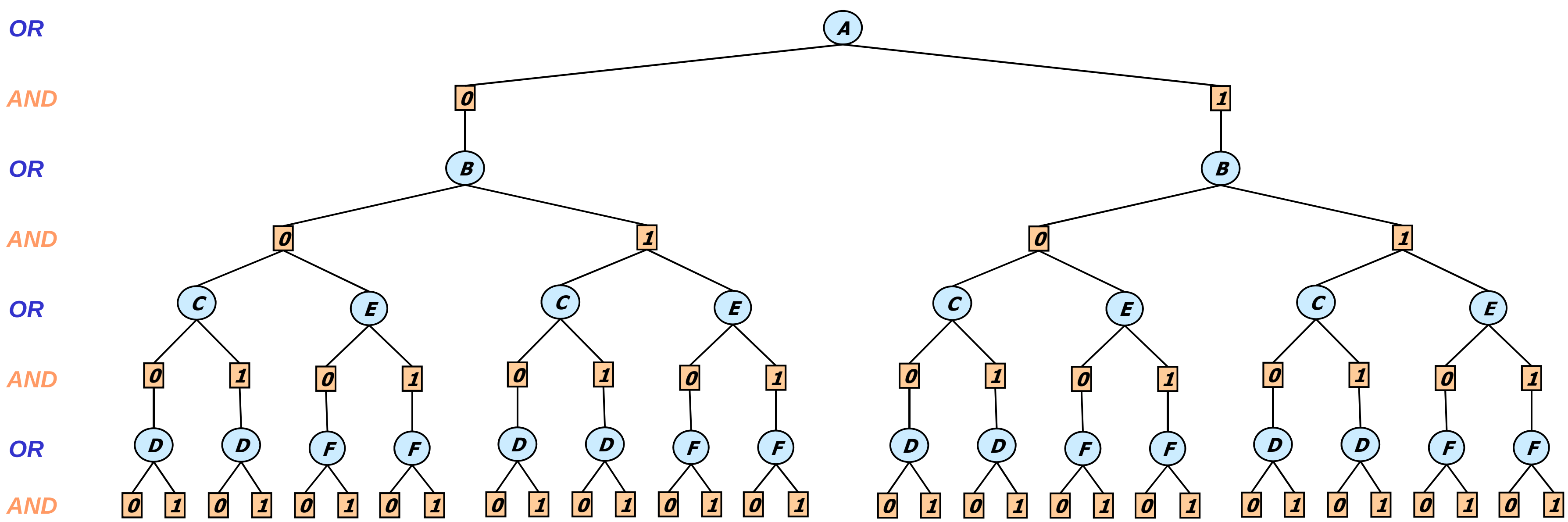

| A | B | C | $R_{ABC}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| 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 |

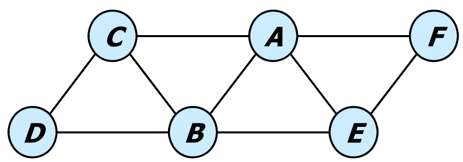
| A | B | E | $R_{ABE}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| 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

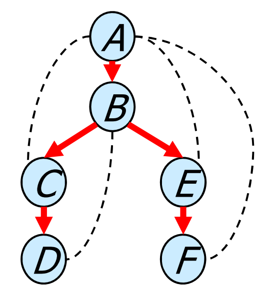


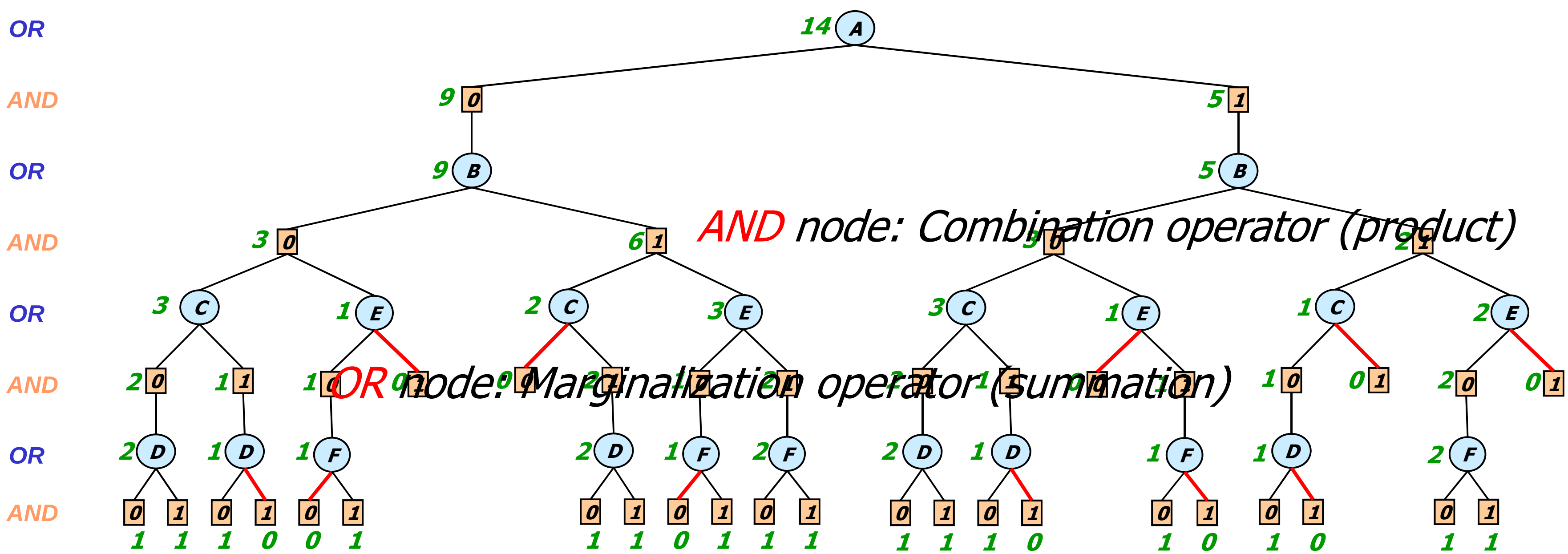

| A | B | C | $R_{ABC}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| 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 |



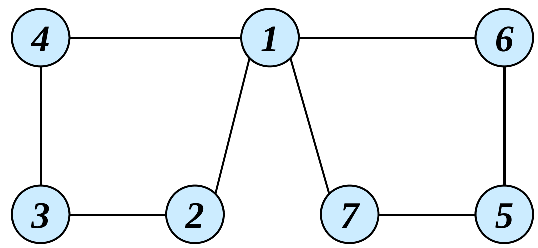




AND node: Combination operator (product)

OR node: Marginalization operator (summation)

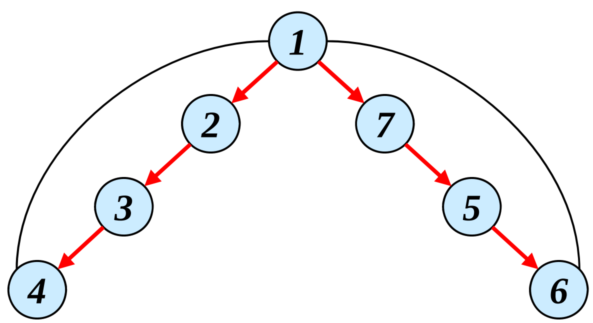
# Pseudo-Trees

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

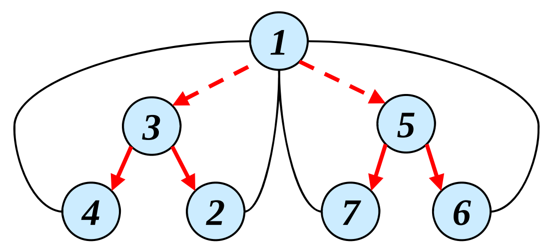



*(a) Graph*

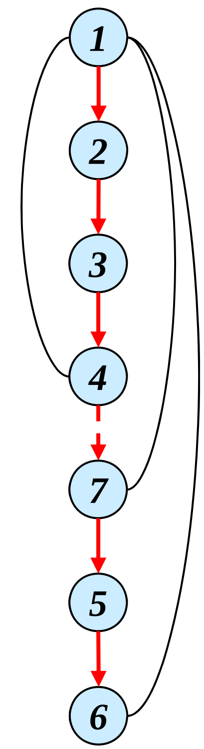
$$h <= w^* \log n$$



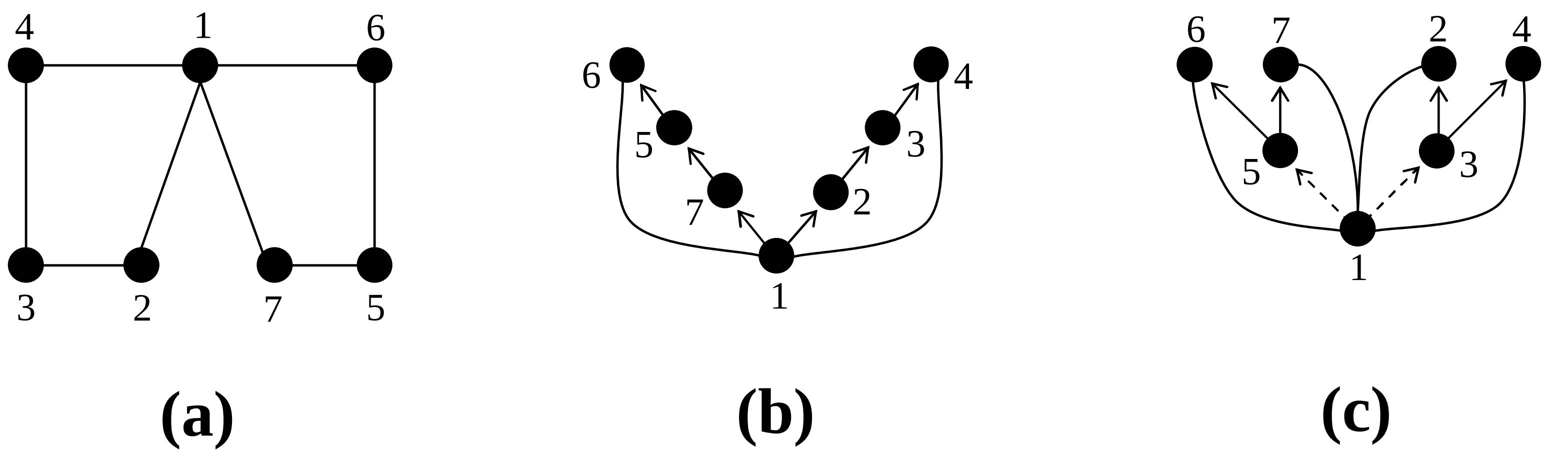

*(b) DFS tree*
*depth=3*




*(c) pseudo- tree*
*depth=2*

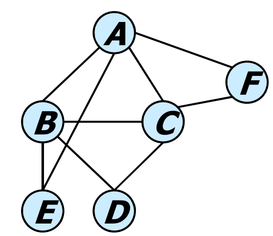



*(d) Chain*
*depth=6*

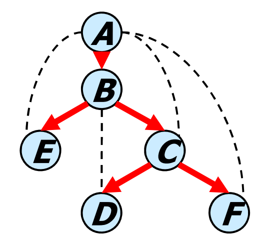

4
1
6
3
2
7
5
(a)
6
5
7
1
2
3
4
(b)
6
7
2
4
5
3
1
(c)

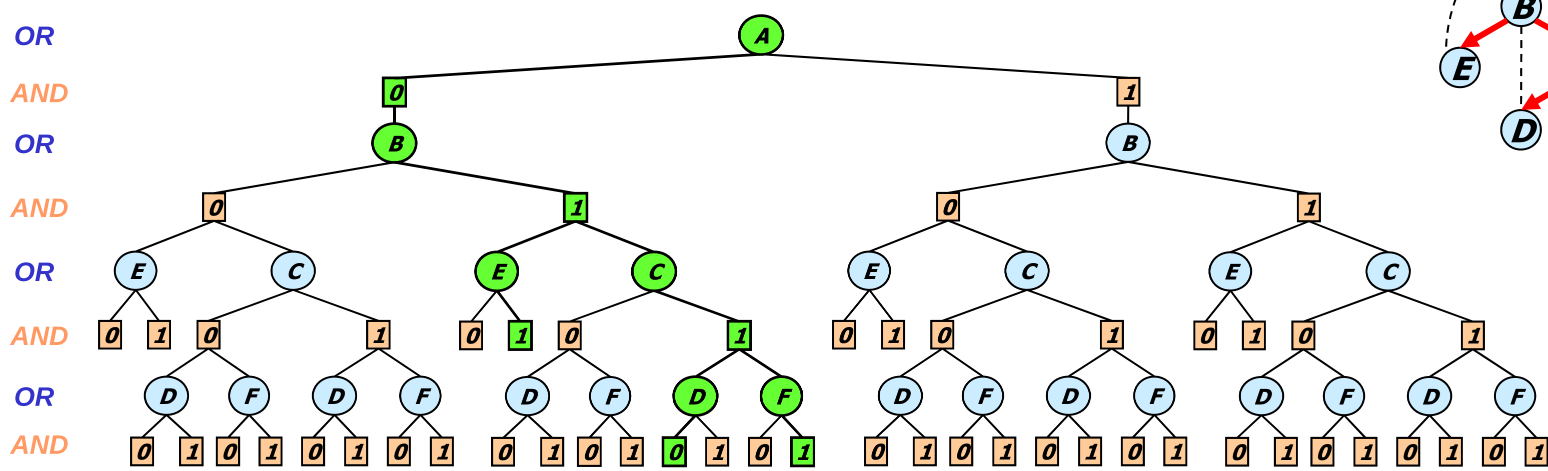
# AND/OR search tree for graphical models

- **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**

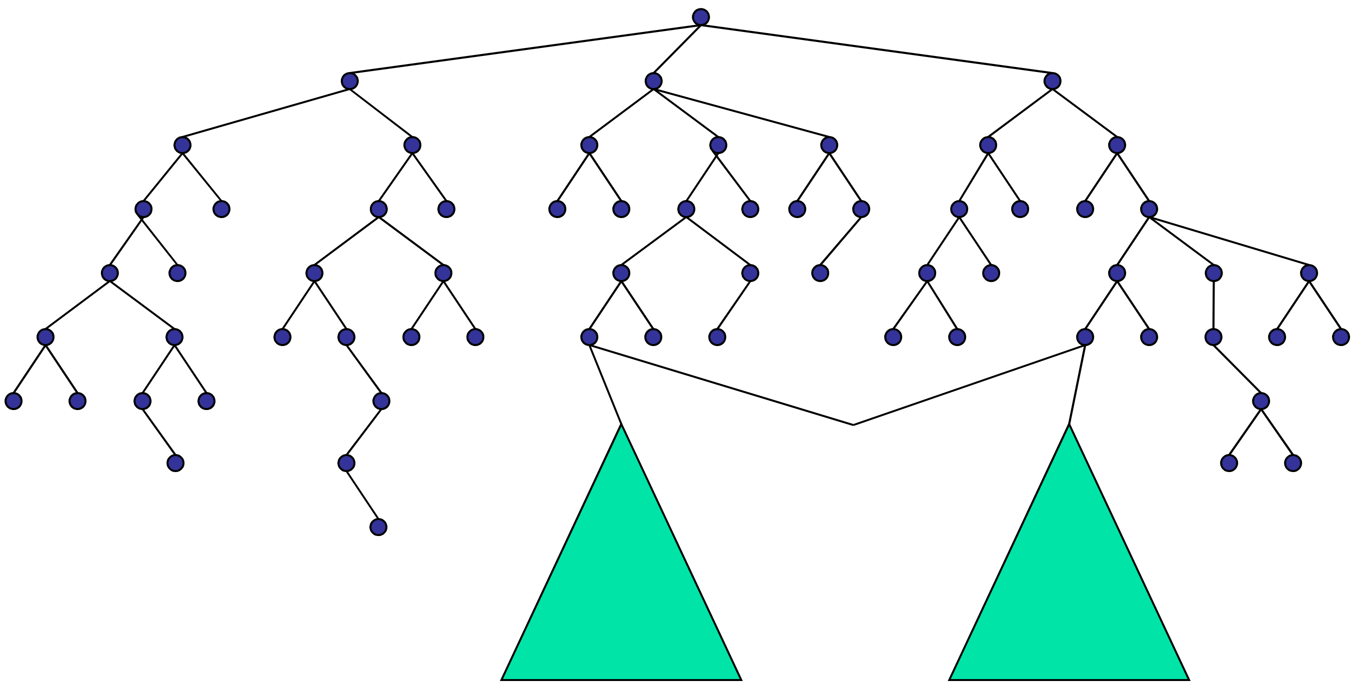


*The end*

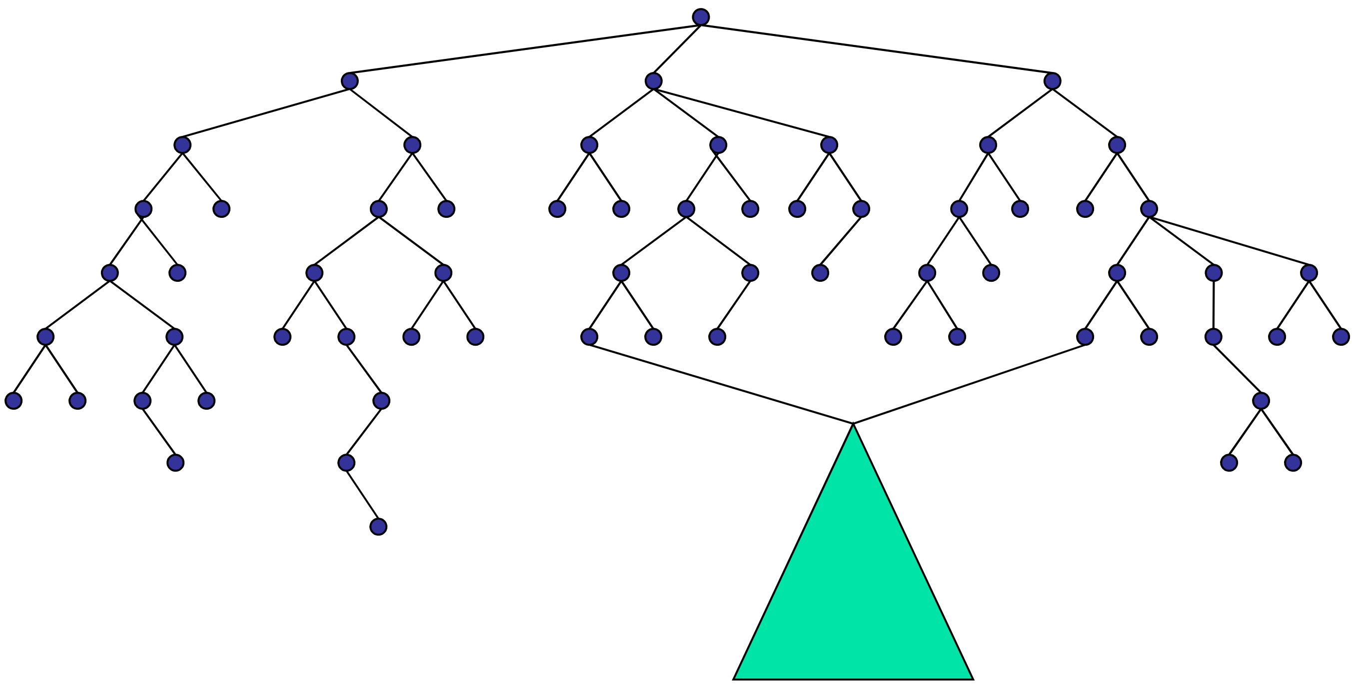
# From Search Trees to Search Graphs

- Any two nodes that root identical subtrees (subgraphs) can be merged



# From Search Trees to Search Graphs

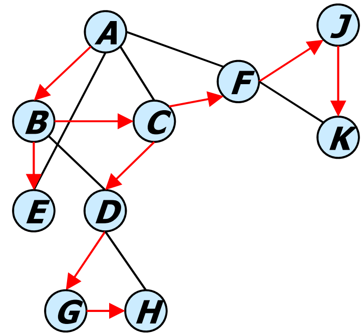
- Any two nodes that root identical subtrees (subgraphs) can be merged

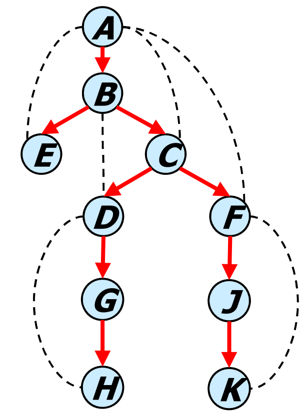


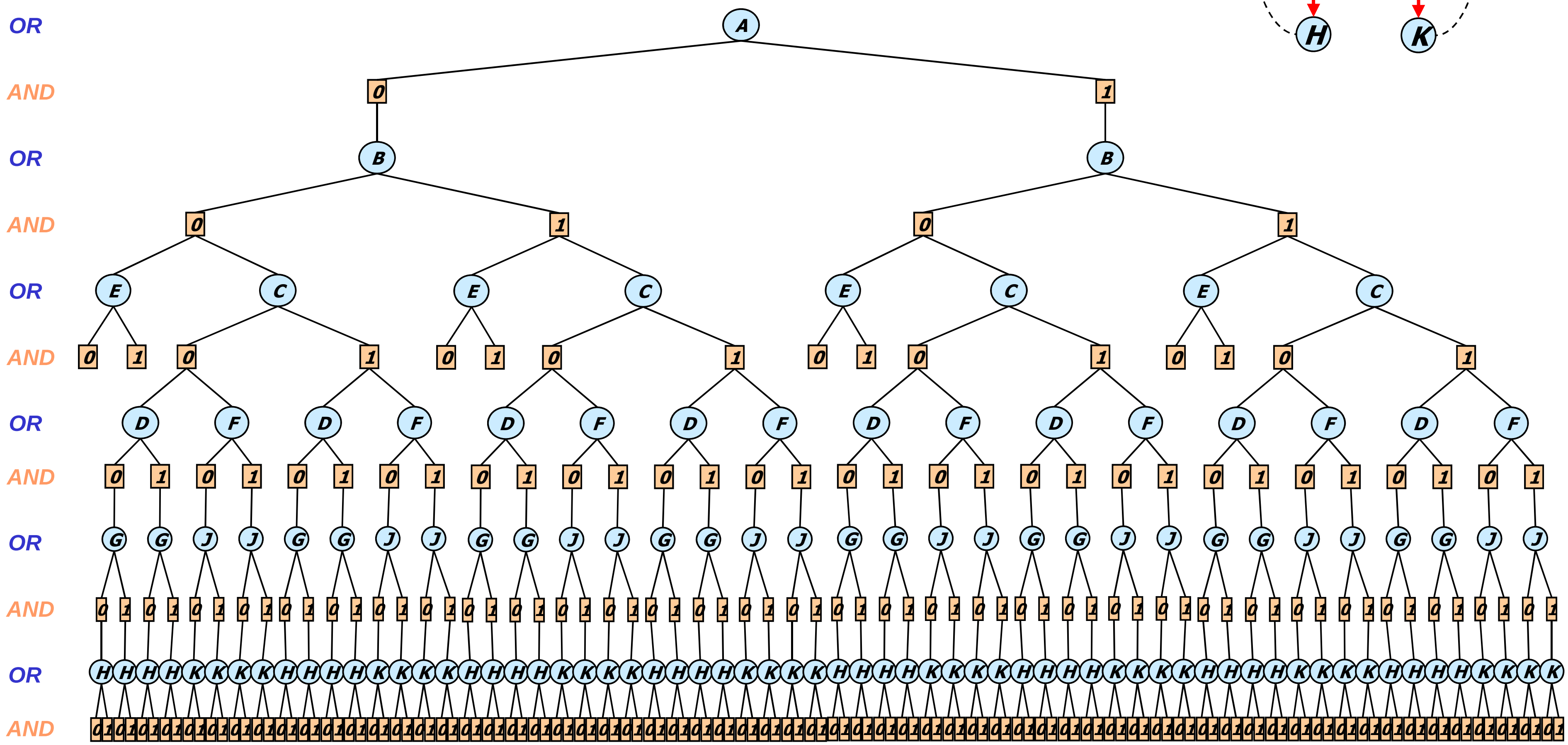
# From Search A/O Trees to Search A/O Graphs

- Any two nodes that root identical subtrees/subgraphs can be merged
- Minimal AND/OR search graph: closure under merge of the AND/OR search tree
  - Inconsistent sub-trees can be pruned too.
  - Some portions can be collapsed or reduced.

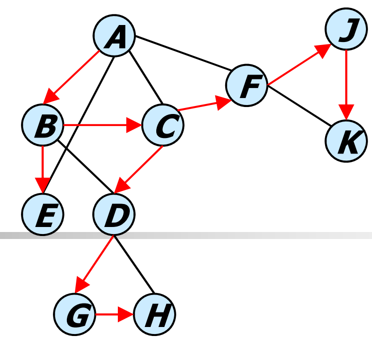
# AND/OR Tree

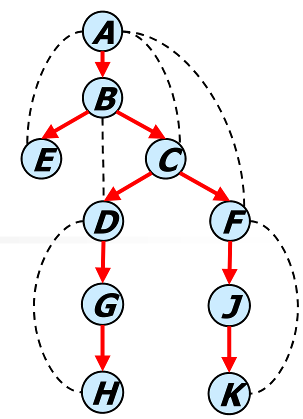

A
J
B
C
F
K
E
D
G
H

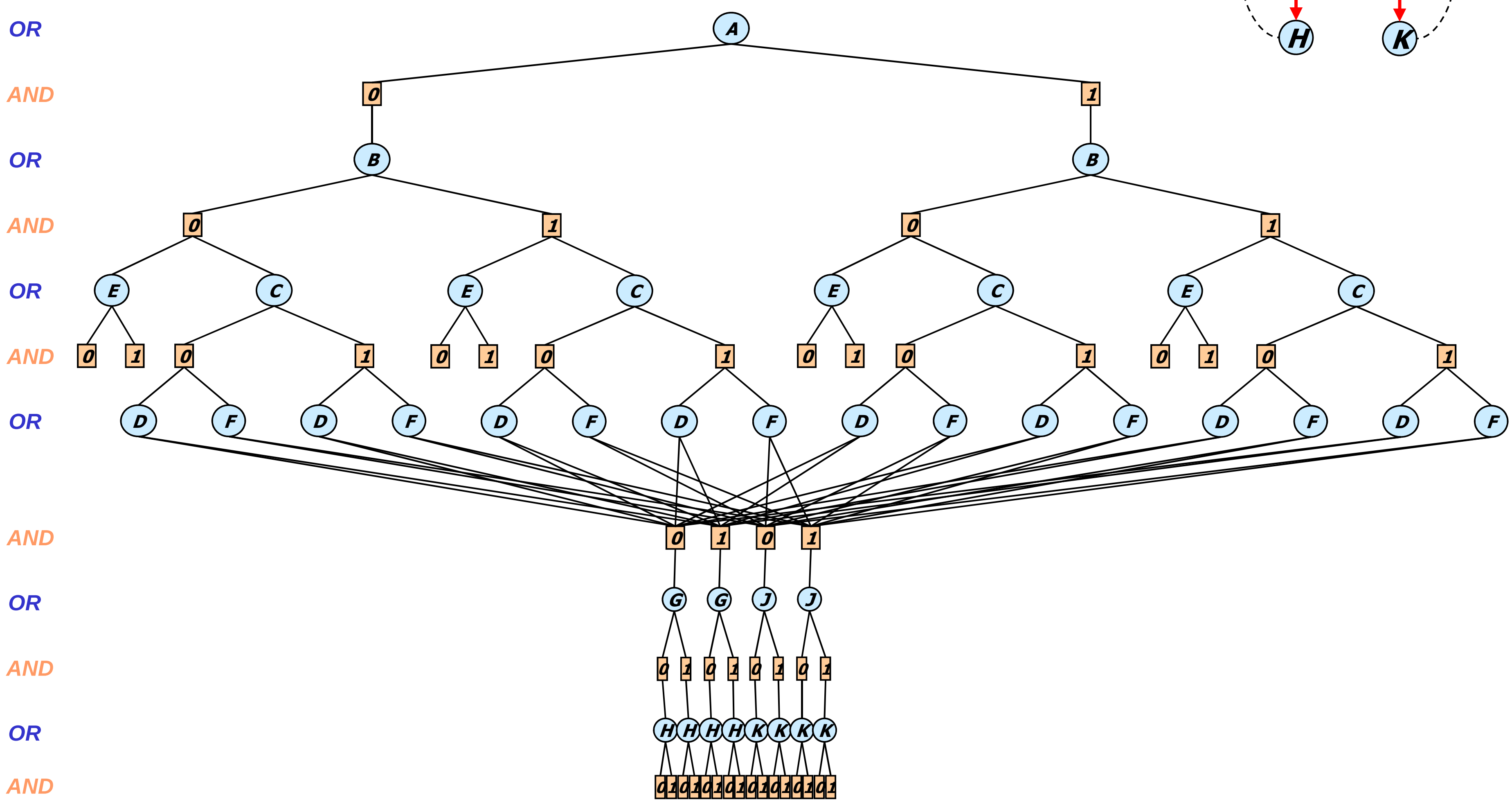

A
B
E
C
D
F
G
J
H
K


OR
AND
OR
AND
OR
AND
OR
AND
OR
AND
OR
AND

# An AND/OR Graph: Caching Goods

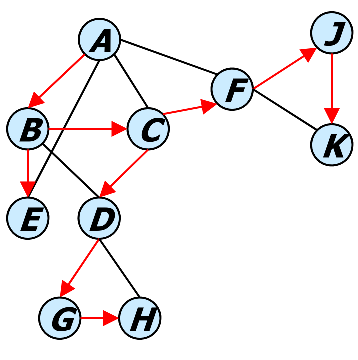


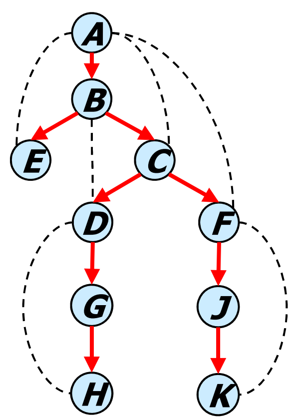



# Context-based Caching

- Caching is possible when context is the same
- context = current variable +
  parents connected to subtree below




*context(B) = {A, B}*

*context(c) = {A,B,C}*

*context(D) = {D}*

*context(F) = {F}*

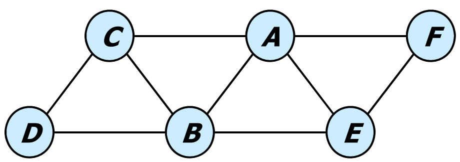
*What is the context size?*
*Induced-width*

# Complexity of AND/OR Graph

- **Theorem:** Traversing the AND/OR search graph is time and space exponential in the induced width/tree-width.

- If applied to the OR graph complexity is time and space exponential in the path-width.

# #CSP – AND/OR Tree DFS

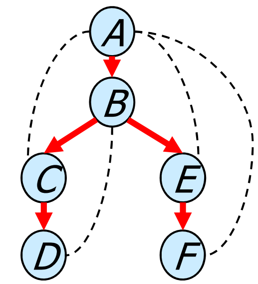


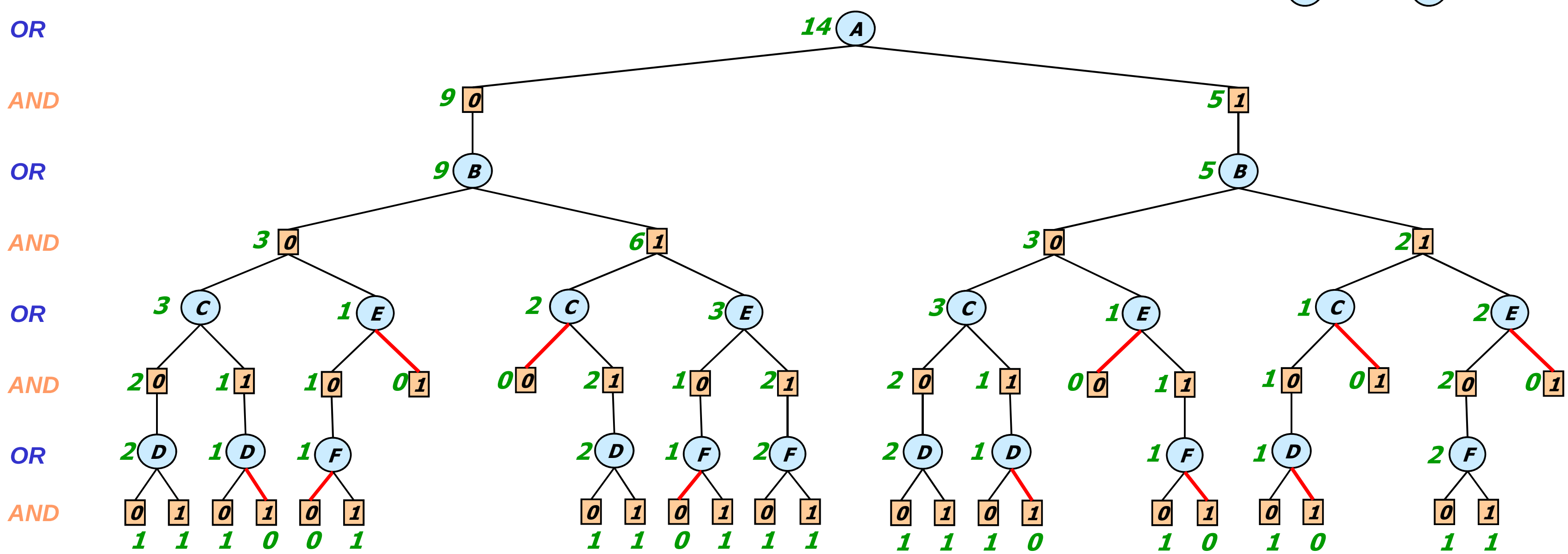

| A | B | C | $R_{ABC}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| 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 |

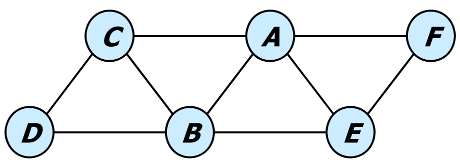
| A | B | E | $R_{ABE}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| 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 |



# #CSP – AND/OR Search Graph (Caching Goods)

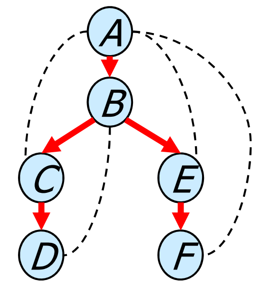


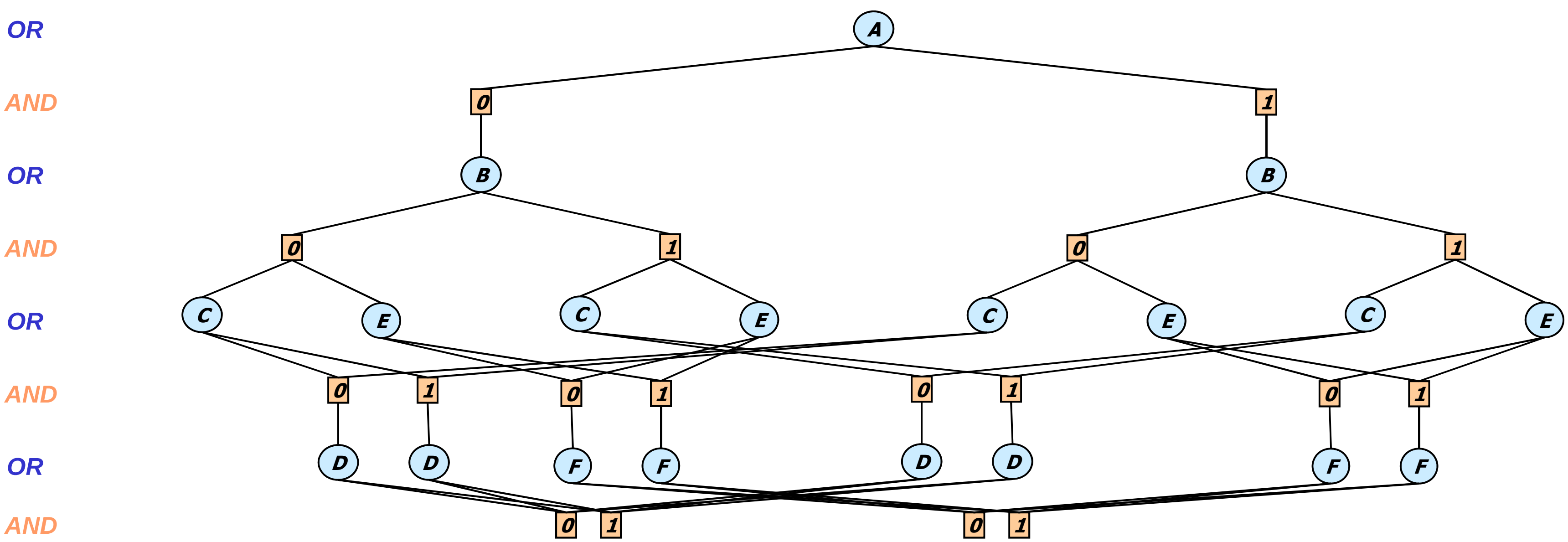

| A | B | C | $R_{ABC}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| 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 |

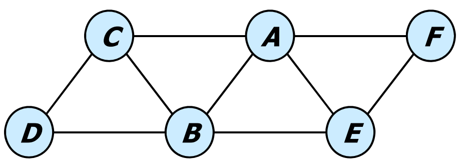
| A | B | E | $R_{ABE}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| 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 Search Graph (Caching Goods)

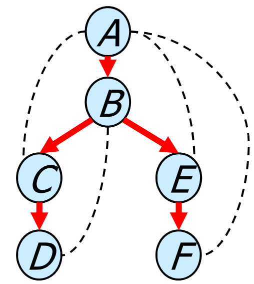



| A | B | C | $R_{ABC}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

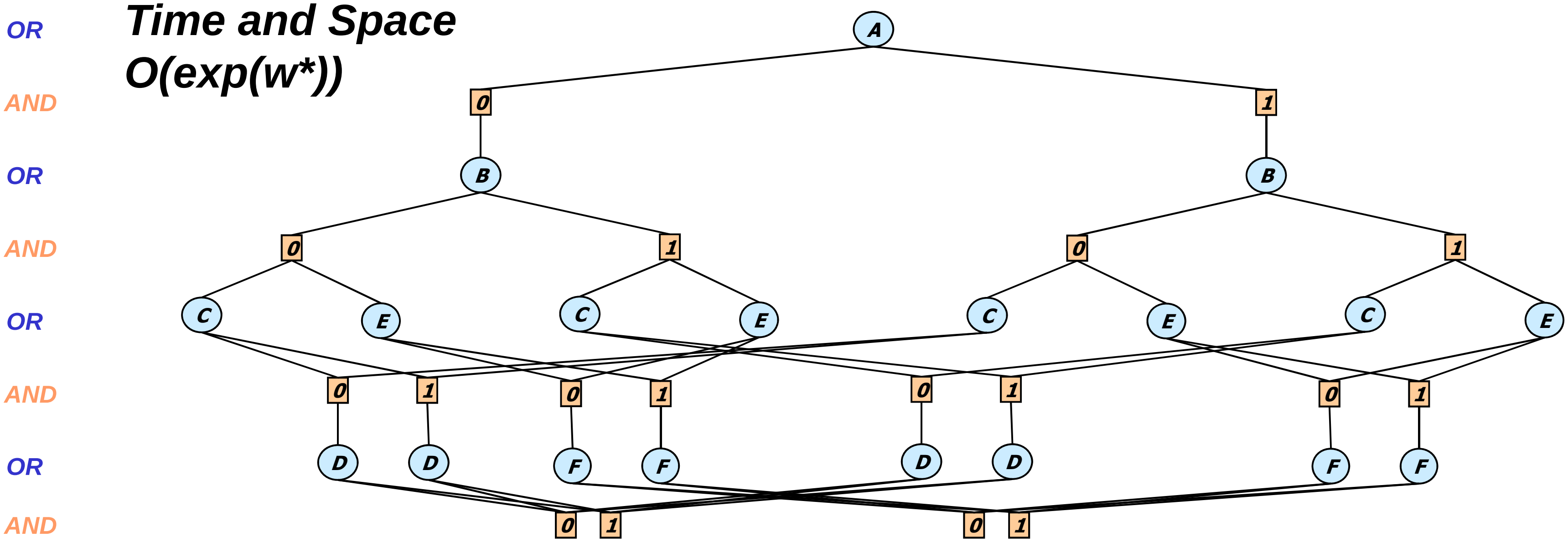
| B | C | D | $R_{BCD}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| 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 |

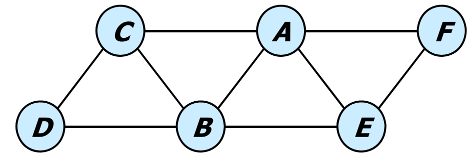
| A | E | F | $R_{AEF}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

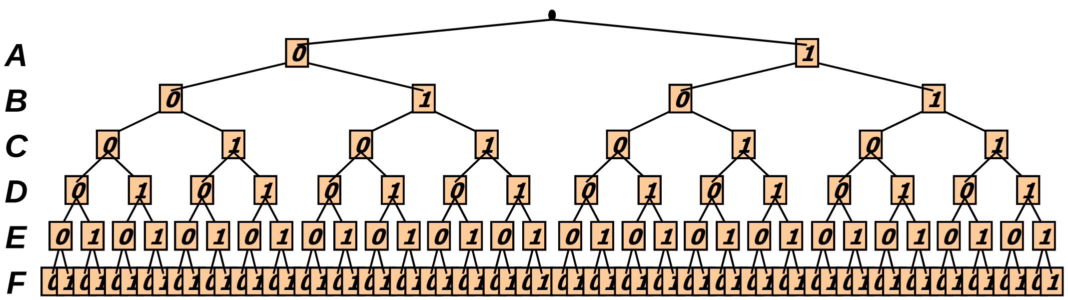



***Time and Space O(exp(w*))***



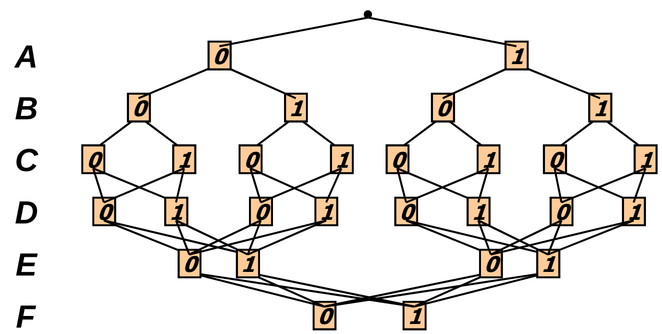
# All Four Search Spaces



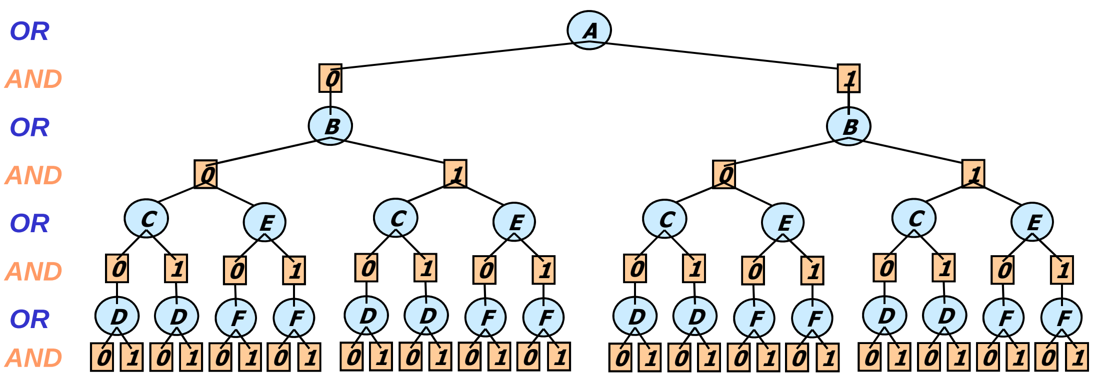

*Full OR search tree*

***126 nodes***



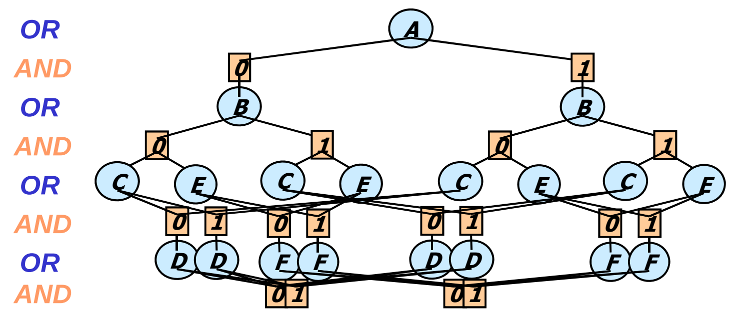

*Context minimal OR search graph*

***28 nodes***




*Full AND/OR search tree*

***54 AND nodes***




*Context minimal AND/OR search graph*

***18 AND nodes***

# AND/OR vs. OR DFS Algorithms

*k = domain size*
*m = tree depth*
*n = # of variables*
*w*= induced width*
*pw*= path width*

- **AND/OR tree**
  - **Space:** $O(n)$
  - **Time:** $O(n\,k^m)$
    $O(n\,k^{w^* \log n})$

  **(Freuder85; Bayardo95; Darwiche01)**

- ***OR tree***
  - ***Space:*** $O(n)$
  - ***Time:*** $O(k^n)$

- **AND/OR graph**
  - **Space:** $O(n\,k^{w^*})$
  - **Time:** $O(n\,k^{w^*})$

- ***OR graph***
  - ***Space:*** $O(n\,k^{pw^*})$
  - ***Time:*** $O(n\,k^{pw^*})$

# Summary: Time-Space for Constraint Processing

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

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