Chapter 3: CSP Resolution

CONSTRAINT PROGRAMMING COURSE

M1 IDO

Introduction

A **Constraint Satisfaction Problem (CSP)** is formally defined as a triple $P = (X, D, C)$, where $X = \{X_1, X_2, \ldots, X_n\}$ is a finite set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of associated domains, and C is a set of constraints.

A **solution** is a complete assignment of values to all variables such that every constraint is satisfied. Throughout this chapter, we assume that all variable domains are finite.

CSP resolution algorithms are broadly categorised into two families:

- Complete algorithms : guaranteed to find a solution if one exists, or prove unsatisfiability.
- Incomplete algorithms : may find solutions faster but offer no completeness guarantee.

We additionally note that specialised algorithms exist for numerical CSPs defined over the reals (linear and non-linear programs) as well as for integer linear programs.

This chapter focuses on generic algorithms applicable to any finite-domain CSP.

Taxonomy of Resolution Algorithms

Category	Algorithm	Key Idea	Completeness
Systematic Search	Generate and Test (GET)	Enumerate all assignments, then check	Complete
Systematic Search	Simple Backtracking (BT)	Extend partial assignments, backtrack on failure	Complete
Filtering (Consistency)	Arc Consistency AC1 / AC3	Prune domains before search	Partial (pre-processing)
Constraint Propagation	Forward Checking (FC)	Prune during search (1 step ahead)	Complete
Constraint Propagation	Look-Ahead (LH)	Prune during search (2 steps ahead)	Complete
Variable/Value Ordering	Heuristics	Guide the search order	Orthogonal to completeness

Generate and Test (GET)
Principle

The **Generate and Test** algorithm constitutes the most naive complete resolution method. It operates in two distinct phases:

- **Generation phase**: enumerate a complete assignment of values to all variables by iterating over the Cartesian product of all domains.
- **Test phase**: once a complete assignment has been generated, evaluate all constraints to determine whether the assignment is a solution.

Because constraint evaluation is deferred until a complete assignment exists, inconsistencies may only be discovered at the deepest level of the search tree. This leads to significant redundancy: partial assignments that are already inconsistent continue to be extended before the inconsistency is finally detected.

Generate and Test (GET) Algorithm

function GET(A, (X, D, C)) : Boolean
begin
  if all variables in X are assigned then
    if A is consistent then return TRUE else   return FALSE endif
  else
    choose an unassigned variable X_i from X
    for each value V_i in D_i do
      if GET(A ∪ {(X_i, V_i)}, (X, D, C)) = TRUE then return TRUE endif
    endfor
    return FALSE
  endif
end

Generate and Test (GET)
Example

Consider the following CSP:

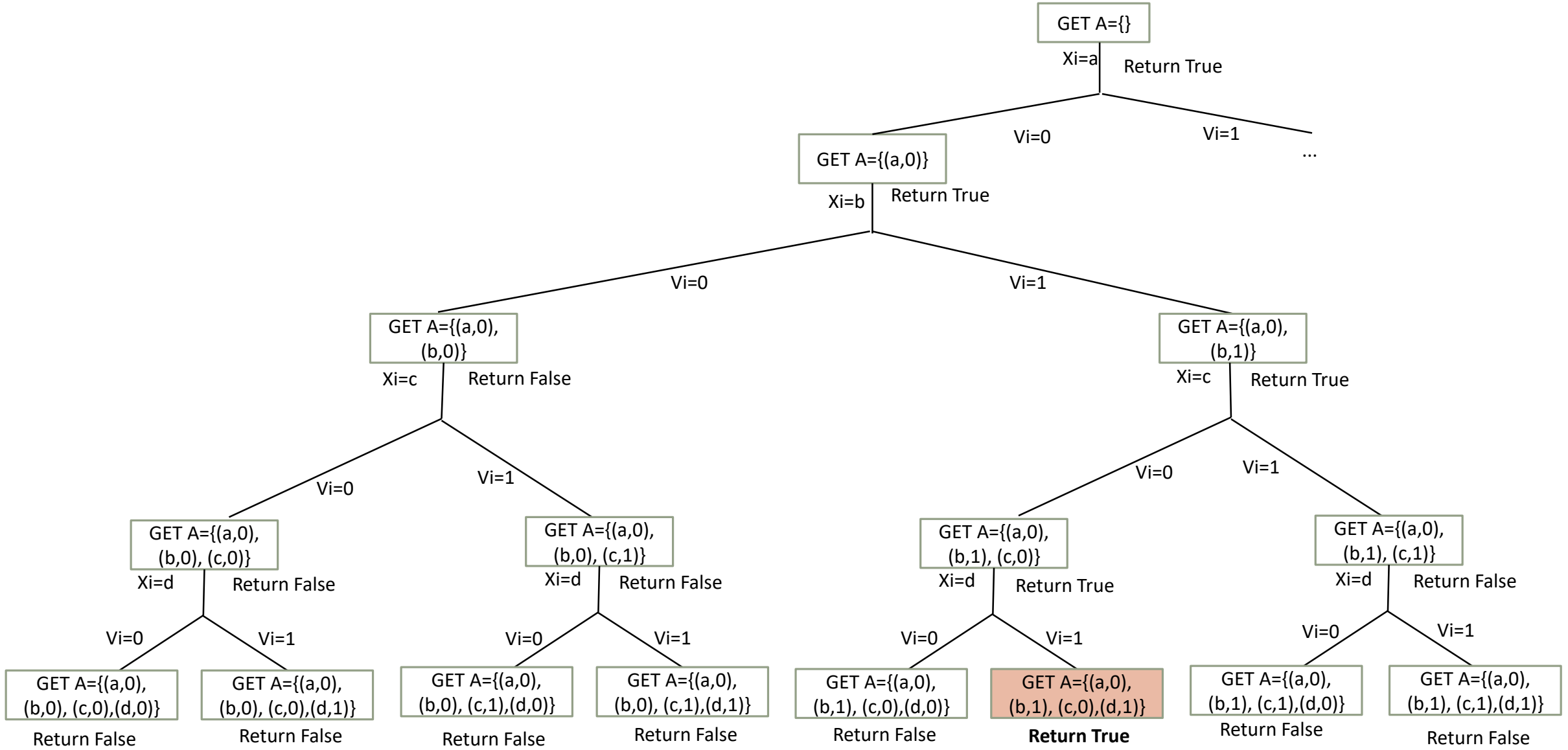
X = {a, b, c, d}

D = {D1, D2, D3, D4} where D1 = D2 = D3 = D4 = {0, 1}

C = {a ≠ b, c ≠ d, a + c < b}

Find all the solutions of the CSP using the algorithm GET.

GET exhaustively enumerates all $2^4 = 16$ complete assignments. At each leaf of the search tree, all three constraints are evaluated. The search tree is explored in a left-to-right, depth-first manner: a is instantiated first, then b, then c, and finally d.


GET A={}
Xi=a
Return True
Vi=0
Vi=1
...
GET A={(a,0)}
Xi=b
Return True
Vi=0
Vi=1
GET A={(a,0), (b,0)}
Xi=c
Return False
GET A={(a,0), (b,1)}
Xi=c
Return True
Vi=0
Vi=1
Vi=0
Vi=1
GET A={(a,0), (b,0), (c,0)}
Xi=d
Return False
GET A={(a,0), (b,0), (c,1)}
Xi=d
Return False
GET A={(a,0), (b,1), (c,0)}
Xi=d
Return True
GET A={(a,0), (b,1), (c,1)}
Xi=d
Return False
Vi=0
Vi=1
Vi=0
Vi=1
Vi=0
Vi=1
Vi=0
Vi=1
GET A={(a,0), (b,0), (c,0),(d,0)}
Return False
GET A={(a,0), (b,0), (c,0),(d,1)}
Return False
GET A={(a,0), (b,0), (c,1),(d,0)}
Return False
GET A={(a,0), (b,0), (c,1),(d,1)}
Return False
GET A={(a,0), (b,1), (c,0),(d,0)}
Return False
GET A={(a,0), (b,1), (c,0),(d,1)}
Return True
GET A={(a,0), (b,1), (c,1),(d,0)}
Return False
GET A={(a,0), (b,1), (c,1),(d,1)}
Return False

Generate and Test (GET)
Limitations

The GET algorithm exhibits the following critical drawbacks:

- The search space corresponds to the full set of complete assignments (of size $|D|^n$), which grows exponentially with the number of variables.
- Inconsistency is only detected at the deepest level of the search tree, which is known as late conflict detection.
- Consistent partial assignments and inconsistent partial assignments are treated identically during generation — no early pruning occurs.

Simple Backtracking (BT)
Principle

Simple Backtracking (BT), also referred to as the **Systematic Backtracking Algorithm**, improves upon GET by checking constraint consistency **incrementally** as each new variable is instantiated.

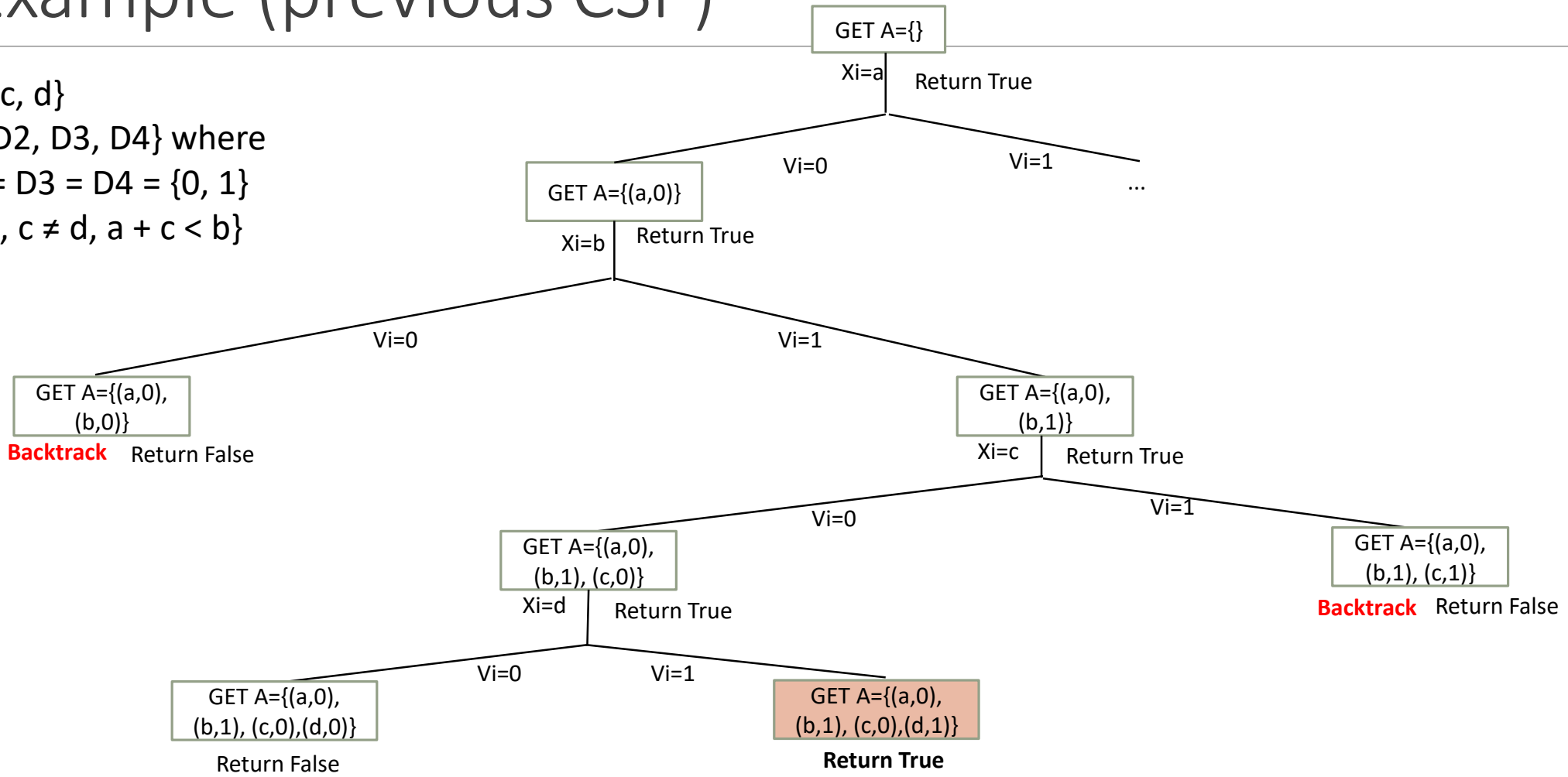
The algorithm maintains a **partial consistent assignment** and attempts to extend it one variable at a time. If the extended assignment becomes inconsistent, the most recently assigned variable is reset and the next available value is tried. If all values for a variable have been exhausted, the algorithm backtracks to the preceding variable.

Simple Backtracking (BT) Algorithm

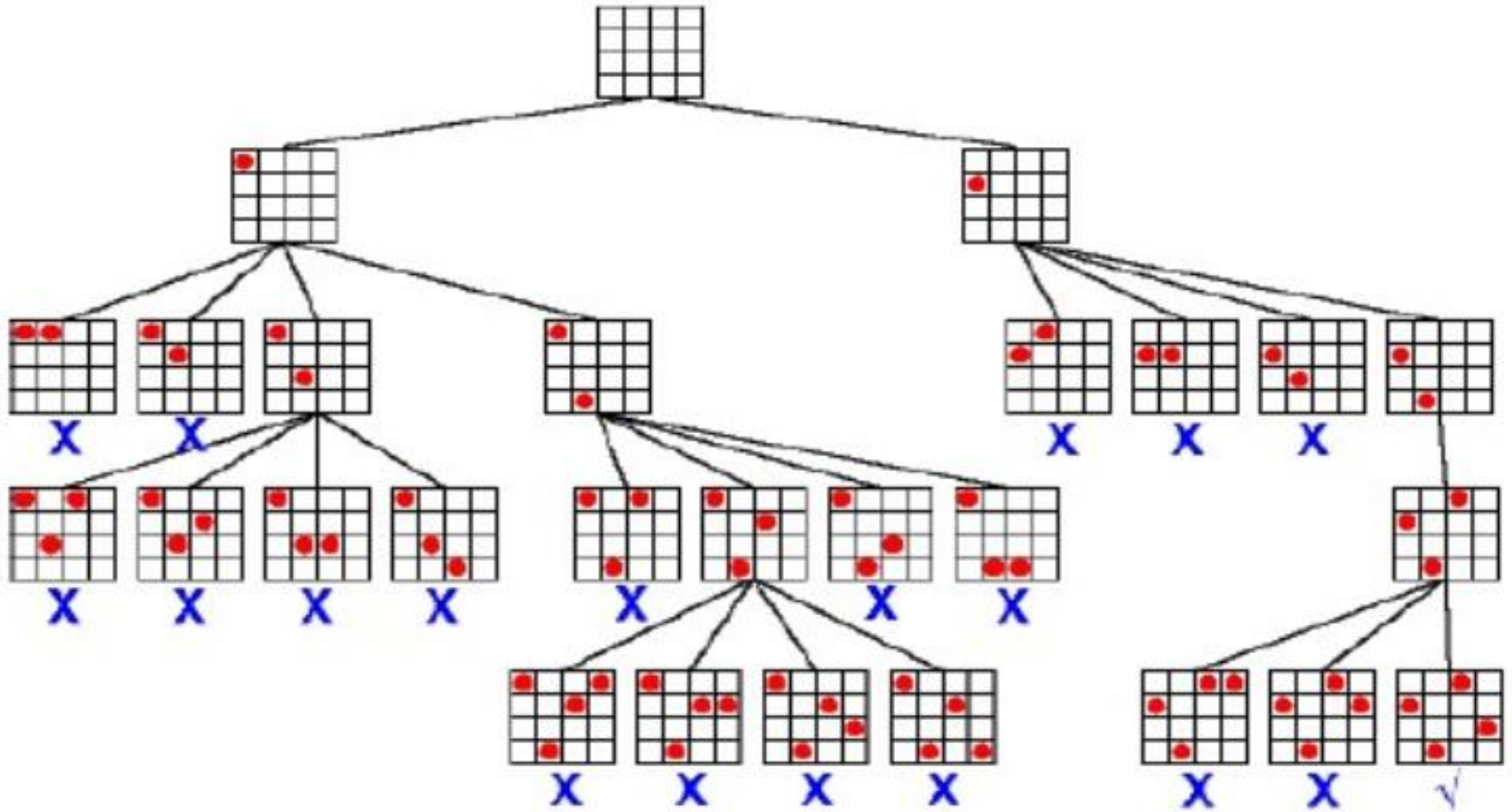
```
function BT(A, (X, D, C)) : Boolean
begin
  if A is not consistent then return FALSE endif
  if all variables in X are assigned then return TRUE else
    choose an unassigned variable Xi from X
    for each value Vi in Di do
      if BT(A ∪ {(Xi, Vi)}, (X, D, C)) = TRUE then return TRUE endif
    endfor
    return FALSE
  endif
end
```

Simple Backtracking (BT)
Example (previous CSP)

X = {a, b, c, d}
D = {D1, D2, D3, D4} where
D1 = D2 = D3 = D4 = {0, 1}
C = {a ≠ b, c ≠ d, a + c < b}



Simple Backtracking (BT)
Application: The 4-Queens Problem



Simple Backtracking (BT)
Limitations

Advantages:

- Significant reduction of the search space compared to GET.
- Detects inconsistencies in partial assignments: no need to complete the assignment before checking constraints.
- Improved both in space complexity and execution time relative to GET.

Limitations:

- No identification of the root cause of a conflict (thrashing): the algorithm may repeat similar failures many times with different variable orderings.
- Redundancy: the same failure configuration may be encountered multiple times during different search paths.
- Conflict detection remains local: the algorithm cannot detect inconsistencies that span more than two levels of the current partial assignment.

,

Filtering Techniques

- Filtering techniques aim to reduce domain sizes by removing values that provably cannot participate in any solution, without performing exhaustive search. This process is known as **constraint propagation** and can be applied either as a **preprocessing step** before search begins or **dynamically** during search.
- Filtering the variable domain by removing locally inconsistent values.
- Filtering can be performed at different levels: node, arc, path

Node Consistency (NC)

Definition: A CSP (X, D, C) is said to be **node-consistent** if, for every variable $X_i \in X$ and every value $v \in D_i$, the singleton assignment $\{(X_i, v)\}$ satisfies all unary constraints on X_i.

Filtering procedure: For each variable X_i, remove from D_i every value v such that $\{(X_i, v)\}$ violates at least one unary constraint. Since unary constraints are evaluated in isolation, this step is inexpensive and is typically performed once before any other filtering.

Arc Consistency (AC)
Definition

A CSP (X, D, C) is **arc-consistent** if

(i) it is node-consistent, and

(ii) for every pair of variables $(X_i, X_j) \in X \times X$ constrained together, and for every value $v_i \in D_i$, there exists at least one value $v_j \in D_j$ such that the partial assignment $\{(X_i, v_i), (X_j, v_j)\}$ satisfies all binary constraints between X_i and X_j. Such a value v_j is called a **support** for v_i on arc (X_i, X_j). Any value lacking a support is removed from the domain.

Filtering procedure: For each variable X_i, remove every value $v_i \in D_i$ for which there exists a variable X_j such that no value $v_j \in D_j$ satisfies the binary constraint between X_i and X_j. Arc consistency is strictly stronger than node consistency.

Arc Consistency (AC)
The REVISE Procedure (Algorithm AC)

The elementary operation underlying all arc consistency algorithms is the REVISE procedure (Algorithm AC)

REVISE iterates over all values V_i in D_i and removes any value that lacks a support in D_j. It returns TRUE if at least one value was deleted, signalling that further propagation may be necessary.

```
Algorithm Arc Consistency: AC
function REVISE((Xi, Xj), (X, D, C)) : Boolean
begin
  DELETED ← FALSE
  for each Vi in Di do
    if no Vj in Dj satisfies the binary constraint between Xi and Xj then
      remove Vi from Di
      DELETED ← TRUE
    end if
  end for
  return DELETE --TRUE iff domain Di was modified
end
```

Algorithm AC1

AC1 is the simplest complete arc consistency algorithm. After an initial node-consistency step, it repeatedly applies REVISE to every arc until no further domain reduction occurs.

AC1 operates by iterating over all arcs in the queue Q in each pass. If any domain is modified during a pass (R = TRUE), the entire queue is re-examined from the beginning. The algorithm terminates when a complete pass produces no domain modification (R = FALSE).

Its main **drawback** is inefficiency: upon detecting a single domain change, the entire set of arcs is revisited, even arcs that are unaffected by the change.

```
function AC1((X, D, C))
begin
  Q ← { (Xi, Xj) | there exists a constraint between Xi and Xj }
  repeat
    R ← FALSE
    for each (Xi, Xj) in Q do
      if REVISE((Xi, Xj), (X, D, C)) then R ← TRUE endif
    end for
  until NOT R
  return (X, D, C)
end
```

Algorithm AC1
Example

Consider the CSP P = (X, D, C) defined as follows:

$X = \{X_1, X_2, X_3, X_4\}$

$D(X_1) = D(X_2) = D(X_3) = D(X_4) = \{1, 2, 3, 4, 5, 6, 7, 8\}$

c_1: X_1 is odd c_2: X_2 is even c_3: X_3 is odd c_4: X_4 is even

c_5: $X_1 < X_2$ c_6: $X_2 < X_4$ c_7: $X_1 < X_4$ c_8: $X_1 < X_3$

c_9: $X_3 < X_4$ c_{10}: $X_2 < X_3$

AC1 Execution

The arc queue is initialised with all ordered pairs of variables linked by a binary constraint:

$Q = \{(X_1,X_2), (X_2,X_1), (X_2,X_4), (X_4,X_2), (X_1,X_4), (X_4,X_1), (X_1,X_3), (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2)\}$

Step 0: Node Consistency

Unary constraints c_1 and c_3 require X_1 and X_3 to be odd; c_2 and c_4 require X_2 and X_4 to be even. Removing parity-violating values yields the following filtered domains:

Variable	Domain after Node Consistency
X_1	{1, 3, 5, 7}
X_2	{2, 4, 6, 8}
X_3	{1, 3, 5, 7}
X_4	{2, 4, 6, 8}

Pass 1 proceeds step by step through the queue:

- **Step 1** (arc (X_1,X_2) and (X_2,X_1)): No modification to $D(X_1)$ or $D(X_2)$ — all values of X_1 admit a larger value in X_2's domain, and vice versa. R remains FALSE.
- **Step 2** (arc (X_2,X_4)): Constraint c_6 requires $X_2 < X_4$. The value $8 \in D(X_2)$ has no support in $D(X_4)$ (no value in {2,4,6,8} is strictly greater than 8). Value 8 is removed. $D(X_2) = \{2,4,6\}$. R ← TRUE.
- **Step 3** (arc (X_4,X_2)): Constraint c_6 requires $X_2 < X_4$. The value $2 \in D(X_4)$ has no support (no value in {2,4,6} is strictly less than 2). Value 2 is removed. $D(X_4) = \{4,6,8\}$. R = TRUE.
- **Step 4** (arcs (X_1,X_4) and (X_4,X_1)): No modification to $D(X_1)$ or $D(X_4)$.
- **Step 5** (arc (X_1,X_3)): Constraint c_8 requires $X_1 < X_3$. The value $7 \in D(X_1)$ has no support in $D(X_3) = \{1,3,5,7\}$ (no value strictly greater than 7). Value 7 removed. $D(X_1) = \{1,3,5\}$.
- **Step 6** (arc (X_3,X_1)): Constraint c_8 requires $X_1 < X_3$. The value $1 \in D(X_3)$ has no support (no value in {1,3,5} is strictly less than 1). Value 1 removed. $D(X_3) = \{3,5,7\}$.
- **Steps 7–8** (arcs (X_3,X_4) and (X_4,X_3)): No modification.
- **Steps 9–10** (arcs (X_2,X_3) and (X_3,X_2)): No modification.

End of Pass 1: R = TRUE (domains were reduced).

X_1	X_2	X_3	X_4
{1, 3, 5}	{2, 4, 6}	{3, 5, 7}	{4, 6, 8}

The algorithm initiates Pass 2, re-examining all arcs.
Re-intialize R to FALSE.
Q = {(X_1,X_2), (X_2,X_1), (X_2,X_4), (X_4,X_2), (X_1,X_4), (X_4,X_1), (X_1,X_3), (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2)}
In Pass 2, no domain is further modified, so R = FALSE and the algorithm terminates.

Remark: The main inefficiency of AC1 is that a complete second pass over all arcs is required even though the domain modifications were confined to X_1, X_2, X_3, and X_4 in specific arcs.

Final arc-consistent domains (AC1):

X_1	X_2	X_3	X_4
{1, 3, 5}	{2, 4, 6}	{3, 5, 7}	{4, 6, 8}

Algorithm AC3

AC3 is a more efficient arc consistency algorithm that tracks which arcs need re-examination. Only the arcs potentially affected by a domain reduction are re-added to the queue.

The key insight of AC3 is: if REVISE reduces D_i, then only the arcs (X_k, X_i) (where $X_k \neq X_j$) need to be re-added to the queue, because the supports for values in D_k pointing to D_i may no longer exist. Arcs not involving X_i are unaffected and do not need to be re-examined. AC3 is the most widely used arc consistency algorithm in practice due to its favourable time complexity.

```
function AC3((X, D, C))
begin
  Q ← { (Xi, Xj) | there exists a constraint between Xi and Xj }
  while Q ≠ ∅ do
    remove (Xi, Xj) from Q
    if REVISE((Xi, Xj), (X, D, C)) then
      // Di was modified: all arcs pointing INTO Xi (except from Xj)
must be re-checked
      Q ← Q ∪ { (Xk, Xi) | constraint(Xk, Xi) ∧ Xk ≠ Xi ∧ Xk ≠ Xj }
    end if
  end while
  return (X, D, C)
end
```

Algorithm AC3
Example

Consider the CSP P = (X, D, C) defined as follows:

$X = \{X_1, X_2, X_3, X_4\}$

$D(X_1) = D(X_2) = D(X_3) = D(X_4) = \{1, 2, 3, 4, 5, 6, 7, 8\}$

c_1: X_1 is odd c_2: X_2 is even c_3: X_3 is odd c_4: X_4 is even

c_5: $X_1 < X_2$ c_6: $X_2 < X_4$ c_7: $X_1 < X_4$ c_8: $X_1 < X_3$

c_9: $X_3 < X_4$ c_{10}: $X_2 < X_3$

AC3 Execution

AC3 processes the same initial queue but, critically, when REVISE reduces a domain D_i, only the arcs (Xk, X_i) for other variables Xk ≠ Xj are re-added to the queue. Arcs already in the queue are not duplicated. This avoids the unnecessary re-examination performed by AC1.

The arc queue is initialised with all ordered pairs of variables linked by a binary constraint:

$Q = \{(X_1,X_2), (X_2,X_1), (X_2,X_4), (X_4,X_2), (X_1,X_4), (X_4,X_1), (X_1,X_3), (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2)\}$

Step 0: Node Consistency

Unary constraints c_1 and c_3 require X_1 and X_3 to be odd; c_2 and c_4 require X_2 and X_4 to be even.
Removing parity-violating values yields the following filtered domains:

Variable	Domain after Node Consistency
X_1	{1, 3, 5, 7}
X_2	{2, 4, 6, 8}
X_3	{1, 3, 5, 7}
X_4	{2, 4, 6, 8}

Key steps:

- **Steps 1–2** (arcs (X_1,X_2) and (X_2,X_1)): No modification; nothing added to the queue.
  - Q = { (X_2,X_4), (X_4,X_2), (X_1,X_4), (X_4,X_1), (X_1,X_3), (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2)}
- **Step 3** (arc (X_2,X_4)): Value 8 removed from $D(X_2)$. Arcs (X_1,X_4) and (X_3,X_4) are re-added, but only the arc (X_1,X_2) is added back to the queue (since (X_3,X_2) is already present).
  - Q = { (X_4,X_2), (X_1,X_4), (X_4,X_1), (X_1,X_3), (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2), (X_1,X_2)}
- **Step 4** (arc (X_4,X_2)): Value 2 removed from $D(X_4)$. Arcs (X_1,X_4) and (X_3,X_4) are already in the queue.
  - Q = { (X_1,X_4), (X_4,X_1), (X_1,X_3), (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2), (X_1,X_2) }
- **Steps 5-6** (arcs (X_1,X_4) and (X_4,X_1)): No modification; nothing added to the queue.
  - Q = { (X_1,X_3), (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2), (X_1,X_2) }
- **Step 7** (arc (X_1,X_3)): Value 7 removed from $D(X_1)$. Arcs (X_2,X_1) and (X_4,X_1) are added.
  - Q = { (X_3,X_1), (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2), (X_1,X_2), (X_2,X_1), (X_4,X_1) }
- **Step 8** (arc (X_3,X_1)): Value 1 removed from $D(X_3)$. Arcs (X_2,X_3) and (X_4,X_3) are already in the queue.
  - Q = { (X_3,X_4), (X_4,X_3), (X_2,X_3), (X_3,X_2), (X_1,X_2), (X_2,X_1), (X_4,X_1) }
- **Steps 9-10** (arcs (X_3,X_4) and (X_4,X_3)): No modification; nothing added to the queue.
  - Q = { (X_2,X_3), (X_3,X_2), (X_1,X_2), (X_2,X_1), (X_4,X_1) }

- **Steps 9-10** (arcs (X_3,X_4) and (X_4,X_3)): No modification; nothing added to the queue.
  - Q = { (X_2,X_3), (X_3,X_2), (X_1,X_2), (X_2,X_1), (X_4,X_1) }
- **Steps 11-10** (arcs (X_2,X_3) and (X_3,X_2)): No modification; nothing added to the queue.
  - Q = { (X_1,X_2), (X_2,X_1), (X_4,X_1) }
- **Steps 12-13** (arcs (X_1,X_2) and (X_2,X_1)): No modification; nothing added to the queue.
  - Q = { (X_4,X_1) }
- **Step 14** (arc (X_4,X_1)): No modification; nothing added to the queue.
  - Q = { }

The queue eventually becomes empty with no domain having been emptied. The CSP is arc-consistent. AC3 terminates more quickly than AC1 because it does not perform a wasted pass over unaffected arcs.

Final arc-consistent domains (AC3) — identical to AC1 result:

X_1	X_2	X_3	X_4
{1, 3, 5}	{2, 4, 6}	{3, 5, 7}	{4, 6, 8}

Note: **AC3 reaches the same arc-consistent problem as AC1**, but does so without the redundant full-queue pass, making it significantly more efficient for large problems.

Constraint Propagation During Search

The filtering techniques presented previously are applied **offline** (before search). A more powerful approach is to integrate constraint propagation **dynamically** into the search process: each time a variable is assigned, the propagation is triggered to reduce the domains of the remaining unassigned variables.

Two algorithms implement this idea at different levels of consistency strength:

- Forward Checking (FC)
- Look-Ahead (LH)

Forward Checking (FC)
Principle

Forward Checking combines backtracking search with **one-step look-ahead** based on node consistency.

After each assignment of a variable X_i, the algorithm proactively filters the domains of **all as-yet-unassigned variables**: for each unassigned variable X_j, every value $v \in D_j$ inconsistent with the current partial assignment is removed. If any domain becomes empty, a dead end has been reached, and the algorithm immediately backtracks.

This approach detects **dead ends one step early**: rather than discovering the empty domain only when attempting to instantiate X_j, the inconsistency is detected as soon as X_i is assigned. This can eliminate entire subtrees before they are entered

Forward Checking (FC)
Algorithm

```
function FC(A, (X, D, C)) : Boolean
begin
  if all variables in X are assigned then return TRUE endif
  choose an unassigned variable Xi from X
  for each value Vi in Di do
    D' ← D
    for each unassigned variable Xj do
      D'j ← { Vj ∈ Dj | A ∪ {(Xi,Vi),(Xj,Vj)} is consistent }
      if D'j = ∅ then return FALSE -- dead end detected early
      endif
    endfor
    if FC(A ∪ {(Xi, Vi)}, (X, D', C)) = TRUE then return TRUE endif
  endfor
  return FALSE
end
```

Forward Checking (FC)
Example

Consider the following binary CSP:

$X = \{X_1, X_2, X_3, X_4\}$

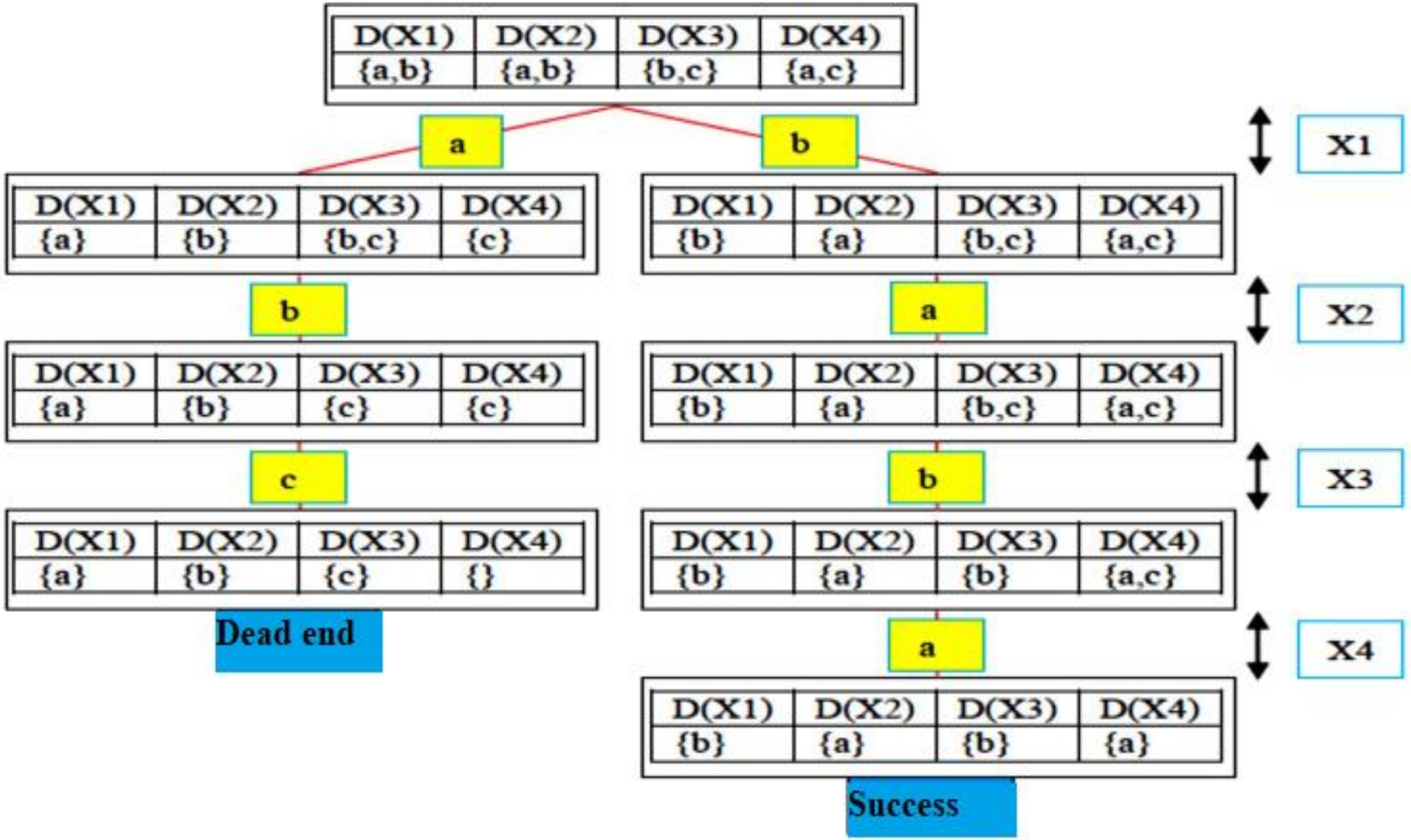
$D(X_1) = D(X_2) = \{a, b\}, \quad D(X_3) = \{b, c\}, \quad D(X_4) = \{a, c\}$

$c_1: X_1 \neq X_2, \quad c_2: X_2 \neq X_3, \quad c_3: X_1 \neq X_4, \quad c_4: X_3 \neq X_4$

- Assign $X_1 = a$. Forward checking: $D(X_2) \leftarrow \{b\}$ (a removed by c_1); $D(X_4) \leftarrow \{c\}$ (a removed by c_3). $D(X_3)$ unchanged.
- Assign $X_2 = b$. Forward checking: $D(X_3) \leftarrow \{c\}$ (b removed by c_2). No empty domain → continue.
- Assign $X_3 = c$. Forward checking: $D(X_4)$ already $\{c\}$; c_4 requires $X_3 \neq X_4 \Rightarrow$ c is removed. $D(X_4) = \emptyset$ → dead end detected. Backtrack.
- Back to $X_1 = a$ branch: no other value for X_3 → backtrack further.
- Try $X_1 = b$. Forward checking: $D(X_2) \leftarrow \{a\}$; $D(X_4) \leftarrow \{a,c\}$ (unchanged since c_3 only excludes b).
- Assign $X_2 = a$. Forward checking: $D(X_3) \leftarrow \{b,c\}$ (unchanged).
- Assign $X_3 = b$. Forward checking: $D(X_4)$: c_4 requires $X_4 \neq b$ → $D(X_4) = \{a,c\}$. Continue.
- Assign $X_4 = a$. All constraints satisfied. Solution: $\{X_1=b, X_2=a, X_3=b, X_4=a\}$.

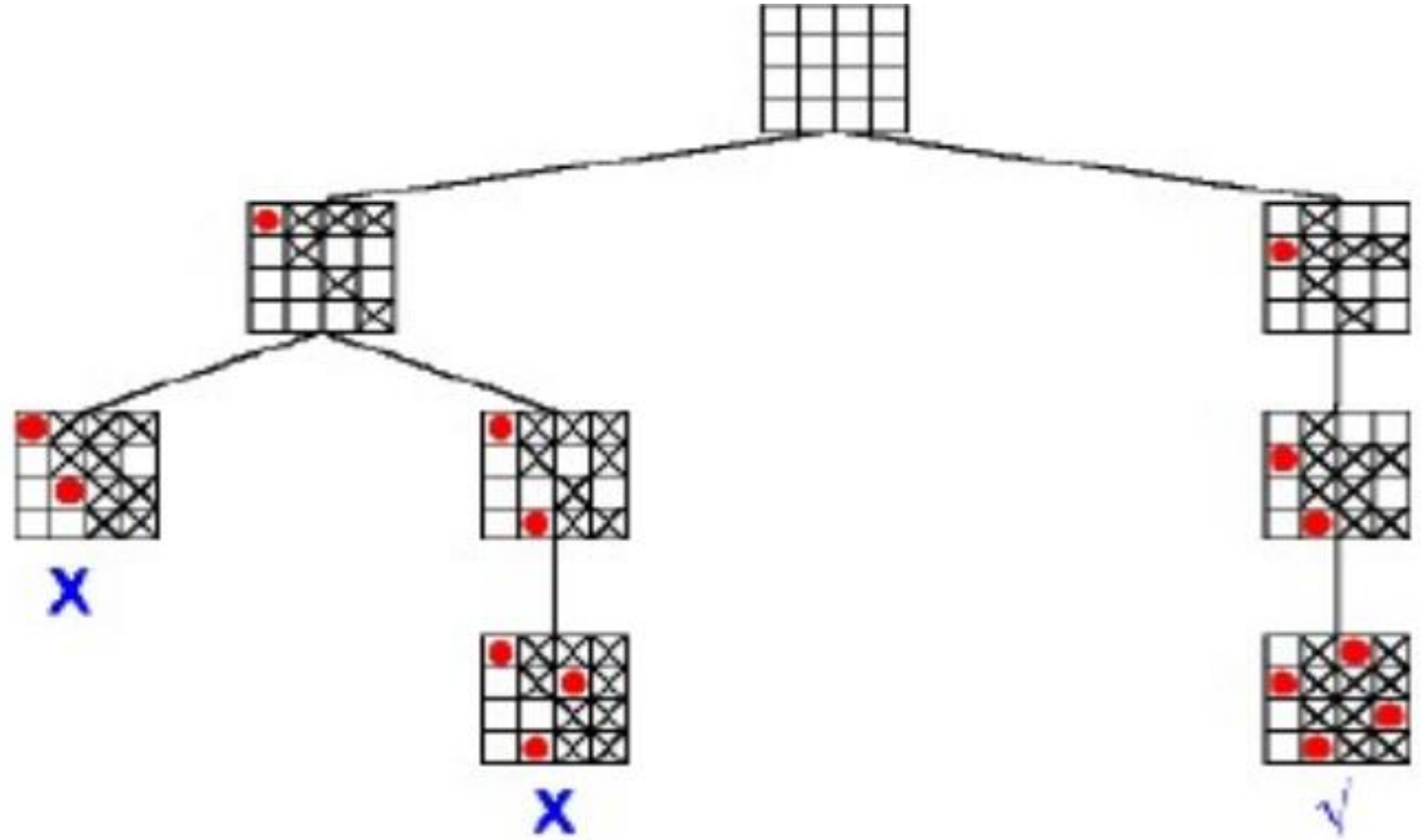
Forward Checking (FC)

Example



Forward Checking (FC)

4-Queens Example

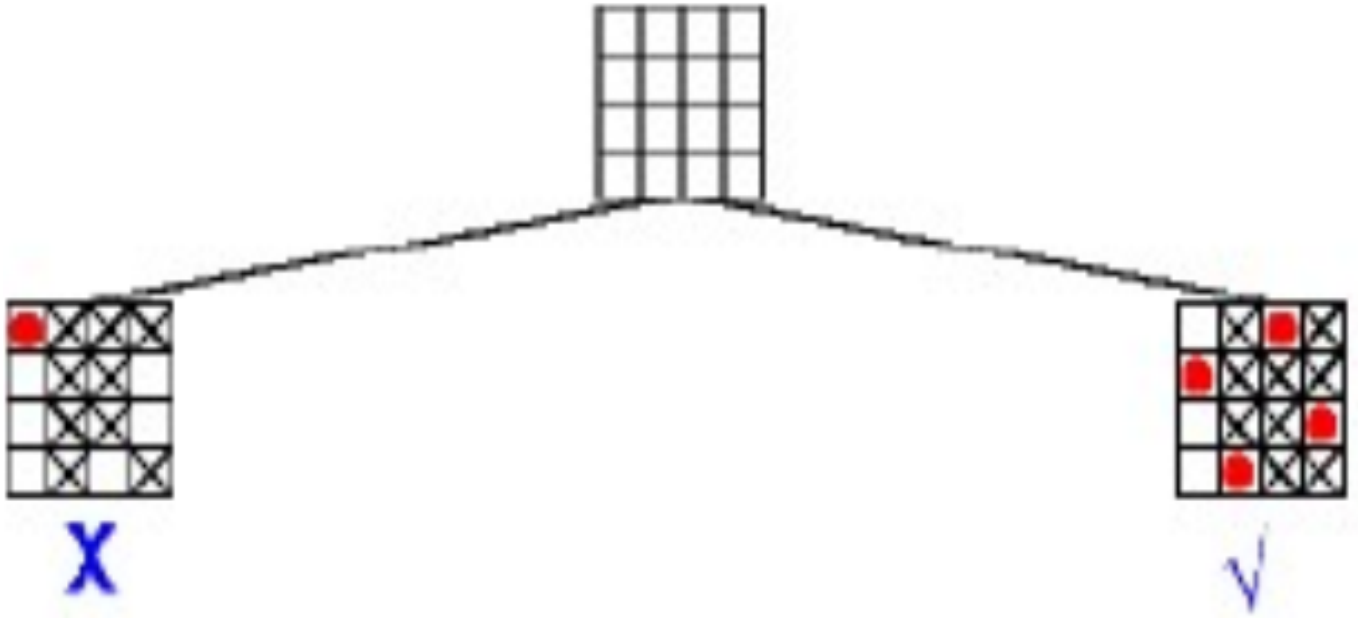


Look-Ahead (LH)
Principle

The Look-Ahead algorithm extends Forward Checking (FC) by applying **arc consistency** between unassigned variables after each instantiation. While FC only checks constraints between the most recently assigned variable X_i and each unassigned variable X_j (one step ahead), LH also propagates constraints between pairs of unassigned variables (X_j, X_k) where $j \neq k$ (two steps ahead). In effect, LH maintains arc consistency among all remaining variables at each node of the search tree.

Formally, after assigning X_i, LH removes from D_j every value v_j for which there exists an unassigned variable X_k such that no value $v_k \in D_k$ is consistent with the combined assignment $A \cup \{(X_i,V_i),(X_j, v_j)\}$. This stronger pruning allows LH to detect dead ends even further in advance than FC, at the cost of greater computation per node.

Look-Ahead (LH)
4-Queens Example



On the 4-Queens problem, Look-Ahead explores only **2 nodes** of the search tree (compared to 3 for FC and up to 27 for simple backtracking), demonstrating the dramatic pruning effect of two-step anticipation.

Heuristics for Variable and Value Ordering

Backtracking search leaves two key decisions unspecified: the **order in which variables are instantiated** and the **order in which values are tried** for each variable.

These choices define the **search strategy** and can dramatically affect the number of nodes explored. Such strategies are collectively referred to as **heuristics**.

Empirical evidence consistently shows that variable ordering heuristics have a **greater impact on performance** than value ordering heuristics.

Heuristics can be either **static** (fixed before search begins) or **dynamic** (recomputed at each search node).

The Fail-First Principle

The fundamental motivation for variable ordering heuristics is the **Fail-First Principle (FFP)**: by attempting to instantiate the most constrained variable first, failures are encountered as early as possible in the search tree, pruning large subtrees before they are explored.

This stands in contrast to a naive ordering that may explore many unfruitful branches before discovering an inevitable failure.

Variable Ordering Heuristics

- **Minimum Remaining Values (MRV) / Smallest Domain First**: assign first the variable whose current domain has the smallest number of remaining values. A variable with a domain of size 1 is effectively already determined; a variable with an empty domain represents an immediate failure.
- **Degree Heuristic:** among variables with equal domain sizes, prefer the variable involved in the largest number of constraints with other unassigned variables. Such a variable has more potential to propagate information forward.
- **Most Constrained Variable**: assign first the variable linked by constraints to the greatest number of already-assigned variables. This maximises immediate constraint evaluation.
- **Least Constraining Value (LCV)** — value ordering: when choosing a value for a variable, prefer the value that eliminates the fewest values from the domains of neighbouring unassigned variables. This maximises the likelihood that future instantiations will succeed.

In practice, the MRV heuristic combined with the degree heuristic (as a tie-breaker) and LCV for value ordering constitutes a highly effective default strategy.