CST401 ARTIFICIAL INTELLIGENCE

MODULE 3

Course Outcomes: After the completion of the course the student will be able to

CO#	CO
CO1	Explain the fundamental concepts of intelligent systems and their architecture. **(Cognitive Knowledge Level: Understanding)**
CO2	Illustrate uninformed and informed search techniques for problem solving in intelligent systems. **(Cognitive Knowledge Level: Understanding)**
CO3	Solve Constraint Satisfaction Problems using search techniques. **(Cognitive Knowledge Level: Apply)**
CO4	Represent AI domain knowledge using logic systems and use inference techniques for reasoning in intelligent systems. **(Cognitive Knowledge Level: Apply)**
CO5	Illustrate different types of learning techniques used in intelligent systems **(Cognitive Knowledge Level: Understand)**

Mapping of course outcomes with program outcomes

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	✓											
CO2	✓	✓										✓
CO3	✓	✓	✓	✓								✓
CO4	✓	✓	✓	✓								✓
CO5	✓	✓			✓							✓

Abstract POs defined by National Board of Accreditation			
PO#	Broad PO	PO#	Broad PO
PO1	Engineering Knowledge	PO7	Environment and Sustainability
PO2	Problem Analysis	PO8	Ethics
PO3	Design/Development of solutions	PO9	Individual and team work
PO4	Conduct investigations of complex problems	PO10	Communication
PO5	Modern tool usage	PO11	Project Management and Finance
PO6	The Engineer and Society	PO12	Life long learning

****

SYLLABUS- MODULE 3 (Search in Complex environments)

Adversarial search - Games, Optimal decisions in games, The Minimax algorithm, Alpha-Beta pruning. Constraint Satisfaction Problems – Defining CSP, Constraint Propagation- inference in CSPs, Backtracking search for CSPs, Structure of CSP problems.

****

ADVERSARIAL SEARCH

In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.

GAMES

Games are competitive environments, in which the agent's goals are in conflict. Games are easy to formalize. Games can be a good model of real-world competitive or cooperative activities. E.g., Military confrontations, negotiation, auctions, etc.

	Deterministic	Stochastic
Perfect information (fully observable)	Chess, checkers, go	Backgammon, monopoly
Imperfect information (partially observable)	Battleships	Scrabble, poker, bridge

Alternating two-player zero-sum games

Here Players take turns. Each game outcome or terminal state has a utility for each player (e.g., 1 for win, 0 for loss). The sum of both players' utilities is a constant.

Games vs. single-agent search

In game we don't know how the opponent will act. The solution is not a fixed sequence of actions from start state to goal state, but a strategy or policy (a mapping from state to best move in that state).

Efficiency is critical to playing well.

- The time to make a move is limited.
- The branching factor, search depth, and number of terminal configurations are huge
- In chess, branching factor ≈ 35 and depth ≈ 100, giving a search tree of nodes
- Number of atoms in the observable universe ≈ 1080
- This rules out searching all the way to the end of the game

PRUNING

We need an algorithm to find optimal move and choosing a good move when time is limited. **Pruning** allows us to ignore portions of the search tree that make no difference to the final

choice, and heuristic **evaluation functions** allow us to approximate the true utility of a state without doing a complete search

MIN- MAX

MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following elements:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values $+1, 0$, or $\frac{1}{2}$. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to $+192$. A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either $0 + 1, 1 + 0$ or $\frac{1}{2} + \frac{1}{2}$. "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of $\frac{1}{2}$.

Game tree

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game a tree where the nodes are game states and the edges are moves. From the initial state, MAX has nine possible moves.

Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN.

> ***search tree*** *is a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.*

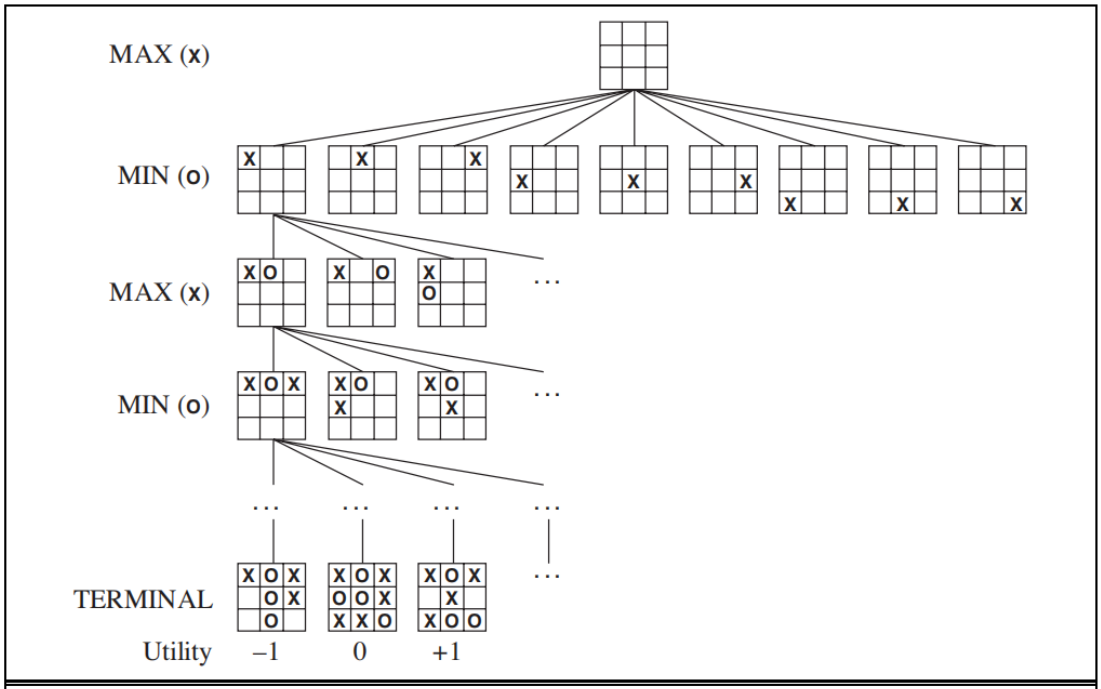


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

OPTIMAL DECISIONS IN GAMES

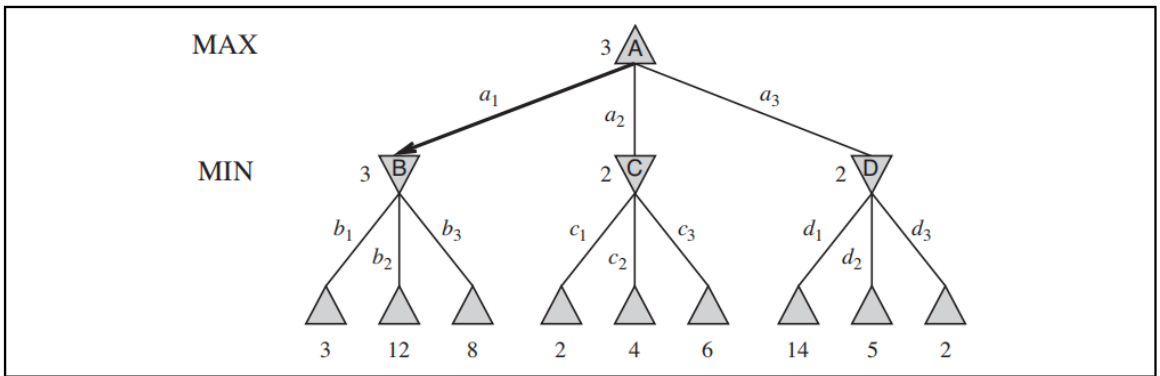


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Minimax Value and Minimax Decision

The minimax value of a node is of being useful in the corresponding state, *assuming that both players play optimally* from there to the end of the game. The minimax value of a terminal state is just its utility(the state of being useful).

MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. The terminal nodes on the bottom level get their utility values from the game's UTILITY

function. Minimax decision will be optimal choice for MAX at root that leads to the state with the highest minimax value

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

The minimax algorithm

Minimax algorithm computes the minimax decision from the current state. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. Minimax algorithm performs a complete depth-first exploration of the game tree

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

maximum depth of the tree is m and there are b legal moves at each point,

time complexity →O(b m).

space complexity → O(bm), for an algorithm that generates all actions at once,

or O(m), for an algorithm that generates actions one at a time

Optimal decisions in multiplayer games

In multiplayer games we replace the single value for each node with a *vector* of values. In a three-player game with players A, B, and C, a vector vA, vB, vC is associated with each node

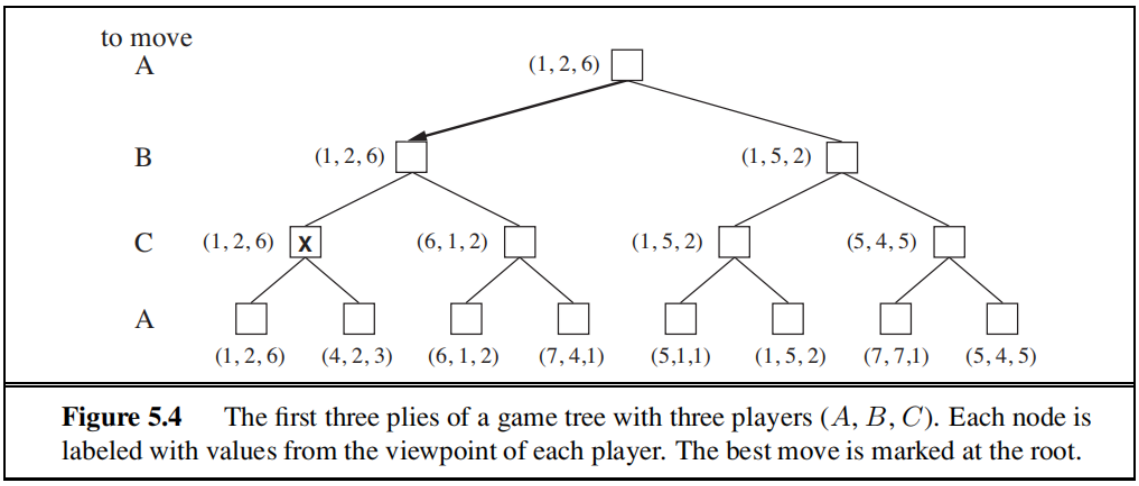


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n

ALPHA BETA PRUNING

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. To avoid this, it is possible to compute the correct minimax decision without looking at every node in the game tree by **pruning.** In alpha beta pruning algorithm it prunes away branches that cannot possibly influence the final decision. Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves

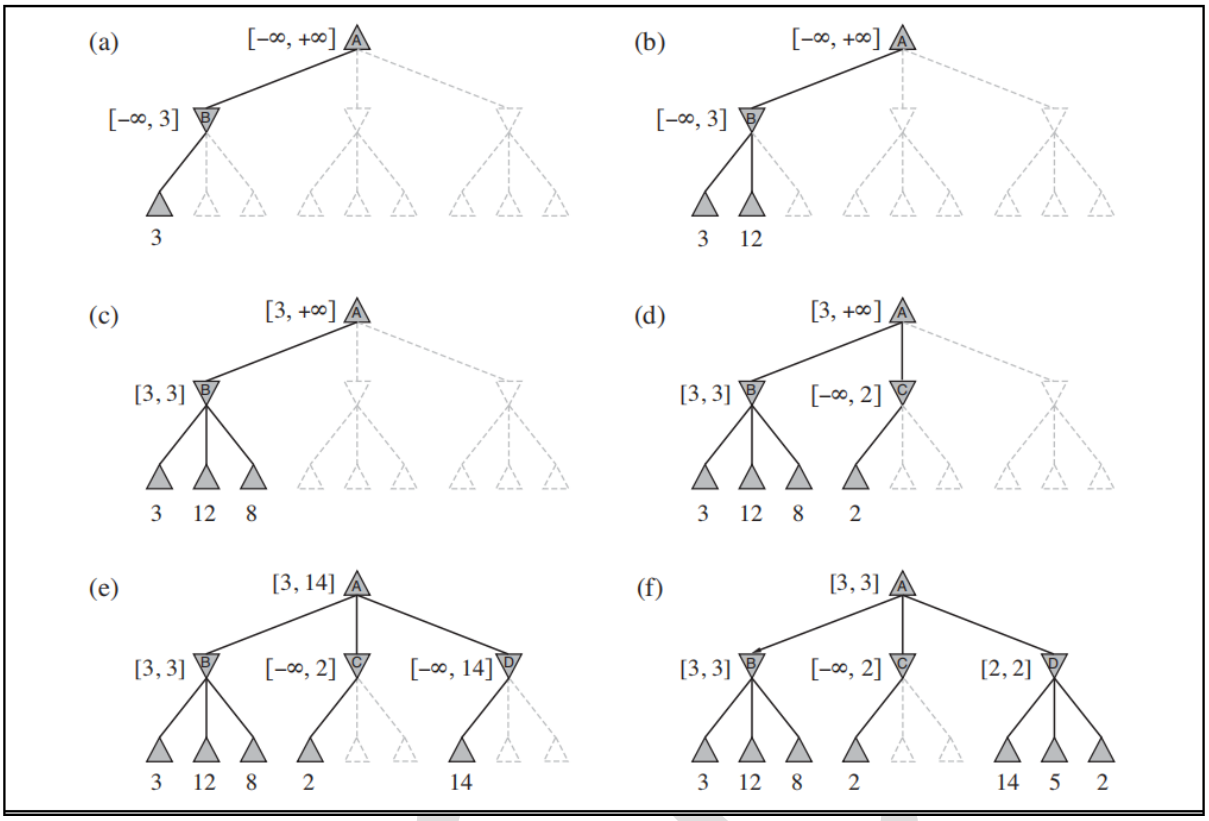
If Player has a better choice m either at the parent node of n or at any choice point further up, then n *will never be reached in actual play.* So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively

The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.



Let the two unevaluated successors of node C have values x and y Then the value of the root node is given by

$$\begin{aligned}\text{MINIMAX}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\ &= \max(3, \min(2,x,y), 2) \\ &= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\ &= 3.\end{aligned}$$

In other words, the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.

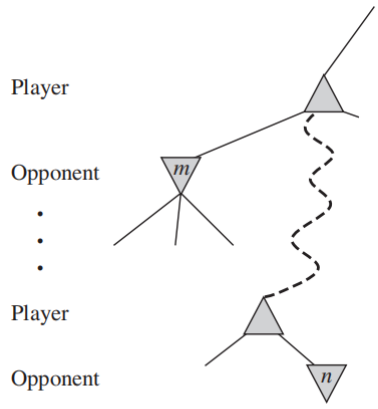


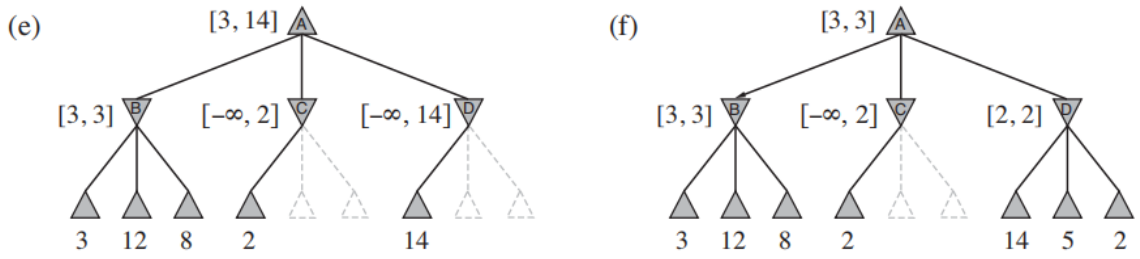
Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Move ordering

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. Eg, here we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of D had been generated first, we would have been able to prune the other two. It might be worthwhile to try to examine first the successors that are likely to be best



Constraint Satisfaction Problem

CSP problem is solved when each variable has a value that satisfies all the constraints on the variable

Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components, X, D, and C:

- X is a set of variables, {X1,...,Xn}.
- D is a set of domains, {D1,...,Dn}, one for each variable.
- C is a set of constraints that specify allowable combinations of values.

Each domain Di consists of a set of allowable values, {v1,...,vk} for variable Xi. Each constraint Ci consists of a pair scope, rel , where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

A relation can be represented as an explicit list of all tuples of values that satisfy the constraint. An abstract relation that supports two operations:

- testing if a tuple is a member of the relation and
- enumerating the members of the relation.

For example, if X1 and X2 both have the domain {A,B}, then the constraint saying the two variables must have different values can be written as

$\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ or as $\langle (X_1, X_2), X_1 \neq X_2 \rangle$

Solving a CSP

To solve a CSP we need to define a state space and the notion of a solution. Each state in a CSP is defined by an **assignment** of values to some or all of the variables, {Xi = vi, Xj = vj ,...}. An assignment that does not violate any constraints is called a **consistent** or legal assignment.

- A **complete assignment** is one in which every variable is assigned, and
- A **solution** to a CSP is a consistent, complete assignment
- A **partial assignment** is one that assigns values to only some of the variables.

Example problem: Map coloring

We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.

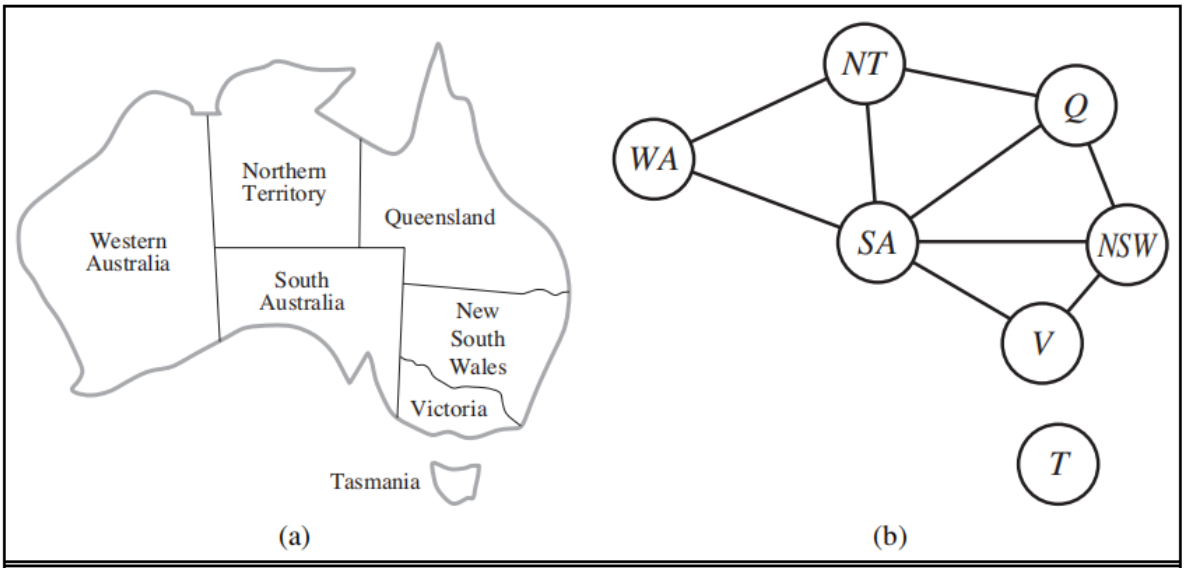


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

X = {WA, NT, Q, NSW ,V, SA, T}

The domain of each variable is the set Di = {red, green, blue}. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

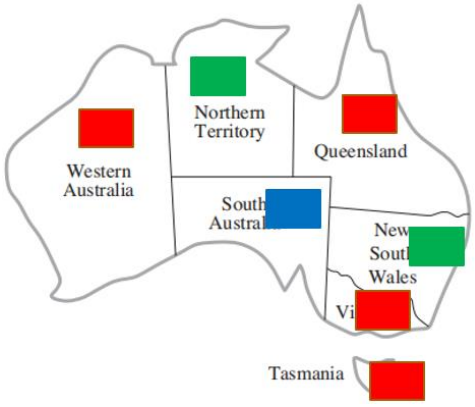
$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}.$$

There are many possible solutions to this problem, such as

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$



Why formulate a problem as a CSP

CSPs yield a natural representation for a wide variety of problems: it is easier to solve a problem using CSP problem solver than to design a custom solution using another search technique. CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space.

E.g., once we have chosen {SA = blue} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue.

Without taking advantage of constraint propagation, a search procedure would have to consider $3^5 = 243$ assignments for the five neighboring variables; with constraint propagation we never have to consider blue as a value, so we have only $2^5 = 32$ assignments to look at, a reduction of 87%.

Once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment. Furthermore, we can see *why* the assignment is not a solution we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

Example problem: Job-shop scheduling

Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another

- a wheel must be installed before the hubcap is put on and
- that only so many tasks can go on at once.
- a task takes a certain amount of time to complete

Precedence constraints

Whenever a task T1 must occur before task T2, and task T1 takes duration d1 to complete, we add an arithmetic constraint of the form

$$T1 + d1 \leq T2$$

A small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affifix hubcaps, and inspect the fifinal assembly. We can represent the tasks with 15 variables:

X = {AxleF , AxleB,Wheel RF ,Wheel LF ,WheelRB,Wheel LB, NutsRF , NutsLF , NutsRB, NutsLB, CapRF , CapLF , CapRB, CapLB,Inspect} .

The value of each variable is the time that the task starts.

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF};$$
$$Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB}\ .$$

Next we say that, for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$Wheel_{RF} + 1 \leq Nuts_{RF}; \quad Nuts_{RF} + 2 \leq Cap_{RF};$$
$$Wheel_{LF} + 1 \leq Nuts_{LF}; \quad Nuts_{LF} + 2 \leq Cap_{LF};$$
$$Wheel_{RB} + 1 \leq Nuts_{RB}; \quad Nuts_{RB} + 2 \leq Cap_{RB};$$
$$Wheel_{LB} + 1 \leq Nuts_{LB}; \quad Nuts_{LB} + 2 \leq Cap_{LB}\ .$$

DISJUNCTIVE CONSTRAINT

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that AxleF and AxleB must not overlap in time; either one comes first or the other does:

(AxleF + 10 ≤ AxleB) **or**

(AxleB + 10 ≤ AxleF)

For every variable except Inspect we add a constraint of the form X + dX ≤ Inspect. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables: Di = {1, 2, 3,..., 27}. This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables.

Variations on the CSP formalism

- Discrete variables
  - Finite domains – Map coloring, scheduling with time limit, 8-queens problem
  - Infinite domains- no deadline job-scheduling problem,
    - Linear constraints solvable, non-linear constraints are undecidable
    - no algorithm exists for solving general **nonlinear constraints** on integer variables
- Continuous variables- eg scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations;
  - **linear programming** problems- where constraints must be linear equalities or inequalities
  - Linear programming problems can be solved in time polynomial in the number of variables

Types of constraints

1. **Unary constraint-** which restricts the value of a single variable
   - Eg, map-coloring problem it could be the case that South Australians won't tolerate the color green; we can express that with the unary constraint

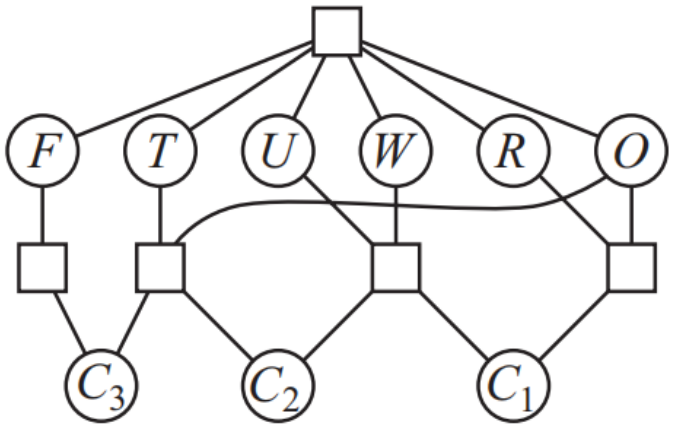
2. **binary constraint-** relates two variables
    - A binary CSP is one with only binary constraints; it can be represented as a constraint graph
3. We can also describe higher-order constraints, such as asserting that the value of Y is between X and Z, with the ternary constraint Between(X, Y, Z).
4. **global constraint-** A constraint involving an arbitrary number of variables
    - Eg, In Sudoku problems all variables in a row or column must satisfy an Alldiffff , which says that all of the variables involved in the constraint must have different values

Real World CSP Problems

- Teaching assignments
- Timetabling
- Hardware configuration (VLSI layout)
- Logistics (transport scheduling)
- Job shop scheduling (Operations research)

Constraint hypergraph

The constraints can be represented in a **constraint hypergraph**. A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints



Constraint optimization problem

Preference constraints: indicating which solutions are preferred. E.g. in university class-scheduling problem Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon.

Preference constraints can often be encoded as costs on individual variable assignments, for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. CSPs with preferences can be solved with

optimization search methods, either path-based or local is called a constraint optimization problem

CONSTRAINT PROPAGATION: INFERENCE IN CSPS

CSPs an algorithm can search or do a specific type of **inference** called **constraint propagation** using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all

Local consistency

If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph

Node consistency

Avariable with the values in the variable's domain satisfy the variable's unary constraints. Eg: variant of the Australia map-coloring problem with South Australians dislike green, the variable SA starts with domain {red, green, blue}, and we can make it node consistent by eliminating green, leaving SA with the reduced domain {red, blue}.

A network is node-consistent if every variable in the network is node-consistent**.** It is always possible to eliminate all the unary constraints in a CSP by running node consistency.

Arc consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. Xi is arc-consistent with respect to another variable Xj. If for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi, Xj). A network is arc-consistent if every variable is arc consistent with every other variable.

For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of digits. The constraint can be written as:

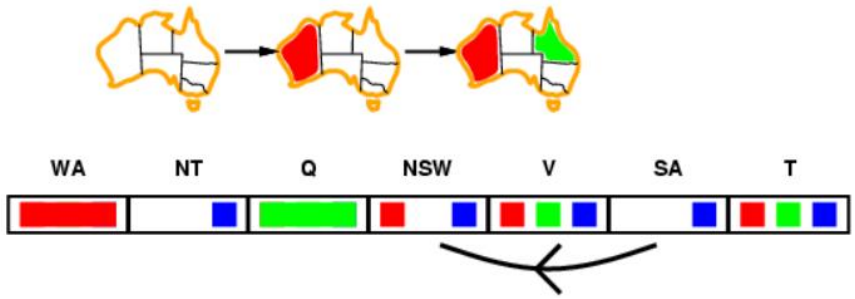
(X, Y), {(0, 0),(1, 1),(2, 4),(3, 9))} .

To make X arc-consistent with respect to Y , we reduce X's domain to {0, 1, 2, 3}. If we also make Y arc-consistent with respect to X, then Y 's domain becomes {0, 1, 4, 9} and the whole CSP is arc-consistent.

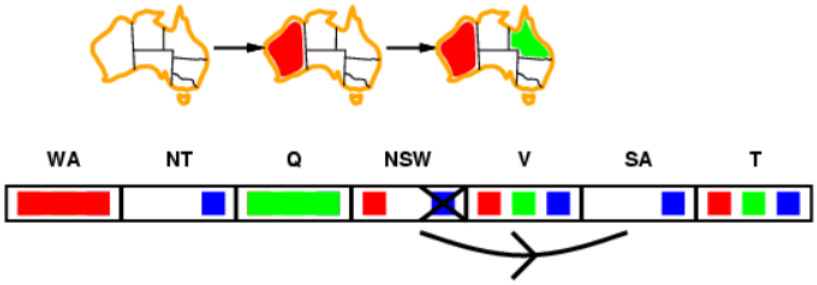
Arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on (SA,WA): {(red, green),(red, blue),(green, red),(green, blue),(blue, red),(blue, green)}

No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

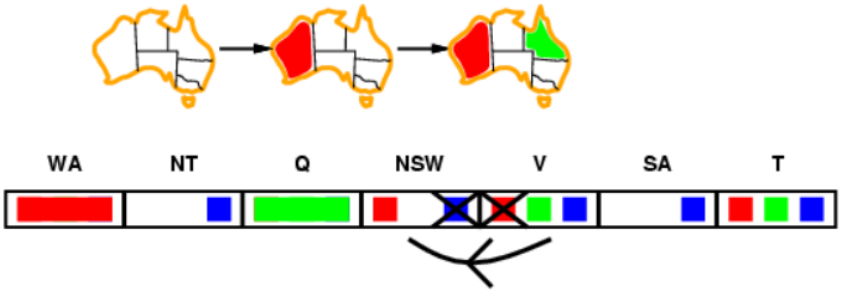
for every value x of X there is some allowed y



for every value x of X there is some allowed y

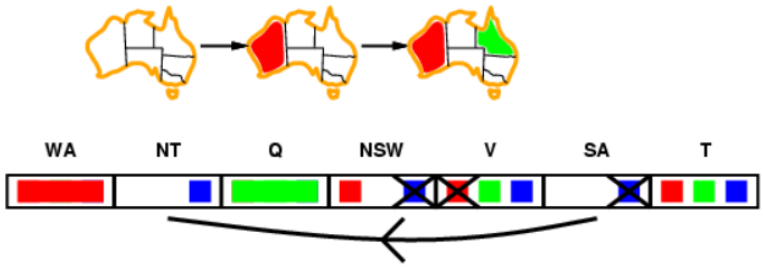


for every value x of X there is some allowed y



- If X loses a value, neighbors of X need to be rechecked

for every value x of X there is some allowed y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

AC-3 Algorithm for Arc Inconsistency

There is a queue of arcs to make every variable arc-consistent

1. Initially, the queue contains all the arcs in the CSP
2. AC-3 then pops off an arbitrary arc (Xi, Xj) from the queue and makes Xi arc-consistent with respect to Xj
3. If Di is unchanged, the algorithm just moves on to the next arc
4. B)Else if this revises Di (makes the domain smaller),
    a. then we add to the queue all arcs (Xk, Xi) where Xk is a neighbor of Xi
        - because the change in Di might enable further reductions in the domains of Dk
        - even if we have previously considered Xk
5. Else if Di is revised down to nothing,
    a. then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure
6. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue
7. At that point, we are left with a CSP that is equivalent to the original CSP
    a. Then they both have the same solutions but
    b. the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains.

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true
```

```
function REVISE( csp, Xi, Xj) returns true iff we revise the domain of Xi
  revised ← false
  for each x in Di do
    if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
      delete x from Di
      revised ← true
  return revised
```

Complexity of AC-3

Assume a CSP with n variables,

- each with domain size at most d, and
- with c binary constraints (arcs).
- Each arc (Xk, Xi) can be inserted in the queue only d times because Xi has at most d values to delete.

Checking consistency of an arc can be done in $O(d^2)$ time, so we get $O(cd^3)$ total worst-case time.

Generalized Arc/Hyperarc Consistent

Arc consistency to handle n-ary rather than just binary constraints. A variable Xi is generalized arc consistent with respect to an n-ary constraint

- if for every value v in the domain of Xi there exists a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and has its Xi component equal to v

For example, if all variables have the domain {0, 1, 2, 3}, then to make the variable X consistent with the constraint X<Y <Z, we would have to eliminate 2 and 3 from the domain of X because the constraint cannot be satisfied when X is 2 or 3.

Arc consistency can go a long way toward reducing the domains of variables, sometimes finding a solution (by reducing every domain to size 1) and sometimes finding that the CSP cannot be solved (by reducing some domain to size 0). But for other networks, arc consistency fails to make enough inferences.

Consider the map-coloring problem on Australia, but with only two colors allowed, red and blue. Arc consistency can do nothing because every variable is already arc consistent: each can be red with blue at the other end of the arc (or vice versa). But clearly there is no solution to

the problem: because Western Australia, Northern Territory and South Australia all touch each other, we need at least three colors for them alone. Arc consistency tightens down the domains using the arcs

Path consistency

Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables. A two-variable set {Xi, Xj} is path-consistent with respect to a third variable Xm if,

- for every assignment {Xi = a, Xj = b} consistent with the constraints on {Xi, Xj},
- there is an assignment to Xm that satisfies the constraints on {Xi, Xm} and {Xm, Xj}.

This is called path consistency because one can think of it as looking at a path from Xi to Xj with Xm in the middle.

Path consistency Fares in coloring the Australia map with two colors

We will make the set {WA, SA} path consistent with respect to NT. We start by enumerating the consistent assignments to the set. In this case, there are only two:

{WA = red, SA = blue} and {WA = blue, SA = red}.

We can see that with both of these assignments NT can be neither red nor blue because it would conflict with either WA or SA. Because there is no valid choice for NT, we eliminate both assignments, and we end up with no valid assignments for {WA, SA}. Therefore, we know that there can be no solution to this problem.

***K*-consistency**

Stronger forms of propagation can be defined with the notion of k-**consistency**. A CSP is k-consistent if, for any set of k − 1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.

- ❑ 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency.
- ❑ 2-consistency is the same as arc consistency.
- ❑ For binary constraint networks, 3-consistency is the same as path consistency.

A CSP is strongly k-consistent if it is k-consistent and is also (k − 1)-consistent, (k − 2)-consistent, ... all the way down to 1-consistent.

Suppose we have a CSP with n nodes and make it strongly n-consistent (i.e., strongly k-consistent for k = n).

We can then solve the problem as follows:

- First, we choose a consistent value for X1.
- We are then guaranteed to be able to choose a value for X2 because the graph is 2-consistent,
- for X3 because it is 3-consistent, and so on.

- For each variable Xi, we need only search through the d values in the domain to find a value consistent with X1,...,Xi−1.
- We are guaranteed to find a solution in time O(n2d).

Any algorithm for establishing n-consistency must take time exponential in n in the worst case. Worse, n-consistency also requires space that is exponential in n. The memory issue is even more severe than the time. In practice, determining the appropriate level of consistency checking is mostly an empirical science. It can be said practitioners commonly compute 2-consistency and less commonly 3-consistency.

Global constraints

Global constraint is one involving an arbitrary number of variables but not necessarily all variables. Eg, in sudoku the Alldiffff constraint says that all the variables involved must have distinct values. One simple form of inconsistency detection for Alldiffff constraints works as follows: if m variables are involved in the constraint, and if they have n possible distinct values altogether, and m>n, then the constraint cannot be satisfied

Simple algorithm for global constraint

If m variables are involved in the constraint, and if they have n possible distinct values altogether, and m>n, then the constraint cannot be satisfied.

1. First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables.
2. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

This method can detect the inconsistency in the assignment {WA = red, NSW = red}. Notice that the variables SA, NT, and Q are effectively connected by an Alldiffff constraint because each pair must have two different colors. After applying AC-3 with the partial assignment, the domain of each variable is reduced to {green, blue}. That is, we have three variables and only two colors, so the Alldiffff constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constraints.

Resource constraint/ Atmost constraint

In a scheduling problem, let P1,...,P4 denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as Atmost(10, P1, P2, P3, P4). We can detect an inconsistency simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain {3, 4, 5, 6}, the Atmost constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain {2, 3, 4, 5, 6}, the values 5 and 6 can be deleted from each domain.

BOUNDS PROPAGATION

For large resource-limited problems it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency-checking methods. E.g., logistical problems involving moving thousands of people in hundreds of vehicles

Instead, domains are represented by upper and lower bounds and are managed by **bounds propagation**. Example in an airline-scheduling problem, let's suppose there are two flights, F1 and F2, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then D1 = [0, 165] and D2 = [0, 385] .

Now suppose we have the additional constraint that the two flights together must carry 420 people: F1 + F2 = 420. Propagating bounds constraints, we reduce the domains to D1 = [35, 165] and D2 = [255, 385] .

We say that a CSP is bounds consistent if for every variable X, and for both the lower- bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y. This kind of bounds propagation is widely used in practical constraint problems.

Sudoku

A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3 × 3 box. A row, column, or box is called a **unit**. Even the hardest Sudoku problems yield to a CSP solver in less than 0.1 second

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row. The empty squares have the domain {1, 2, 3, 4, 5, 6, 7, 8, 9} and the pre-filled squares have a domain consisting of a single value.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

In addition, there are 27 different Alldiff constraints: one for each row, column, and box of 9 squares:

Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)

Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)

…

Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)

Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)

…

Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)

Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)

…

Assume that the Alldiffff constraints have been expanded into binary constraints (such as A1 = A2) so that we can apply the AC-3 algorithm. Consider variable E6 the empty square between the 2 and the 8 in the middle box.

From the constraints in the box, we can remove not only 2 and 8 but also 1 and 7 from E6's domain. From the constraints in its column, we can eliminate 5, 6, 2, 8, 9, and 3. That leaves E6 with a domain of {4}; in other words, we know the answer for E6.

Now consider variable I6 the square in the bottom middle box surrounded by 1, 3, and 3. Applying arc consistency in its column, we eliminate 5, 6, 2, 4 (since we now know E6 must be 4), 8, 9, and 3. We eliminate 1 by arc consistency with I5 , and we are left with only the value 7 in the domain of I6. Now there are 8 known values in column 6, so arc consistency can infer that A6 must be 1. Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value

Sudoku problems are designed to be solved by inference over constraints. But many other CSPs cannot be solved by inference alone; there comes a time when we must search for a solution.

Backtracking Search for CSP

Backtracking search algorithms that work on partial assignments. A standard depth-limited search can be applied. A state would be a partial assignment, and an action would be adding var = value to the assignment. But for a CSP with n variables of domain size d, the branching factor at the top level is nd because any of d values can be assigned to any of n variables.

At the next level, the branching factor is $(n - 1)d$, and so on for n levels. We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments! Inference can be interwoven with search.

Commutativity- crucial property common to all CSPs

CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. We need only consider a single variable at each node in the search tree.

Backtracking search

It is a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, value)
          if inferences ≠ failure then
              add inferences to assignment
              result ← BACKTRACK(assignment, csp)
              if result ≠ failure then
                  return result
      remove {var = value} and inferences from assignment
  return failure
```

BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones

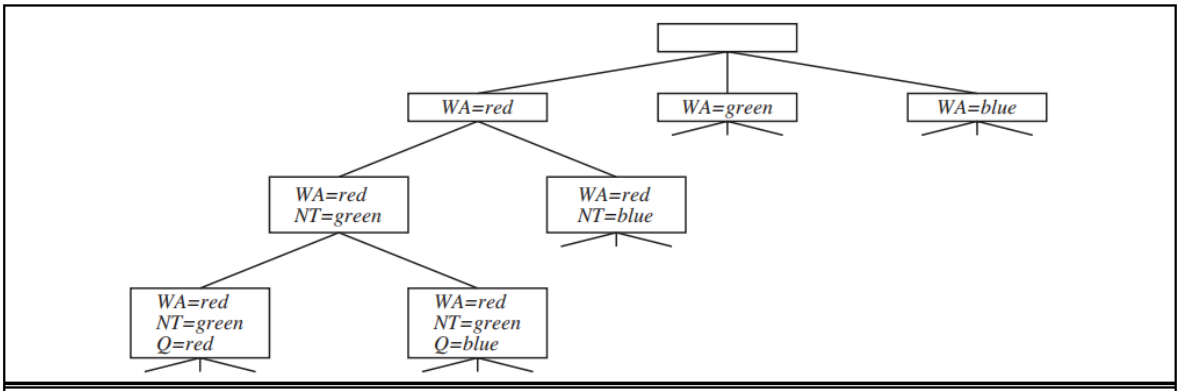


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

To solve CSPs efficiently without domain-specific knowledge, address following questions:

1)function SELECT-UNASSIGNED-VARIABLE: which variable should be assigned next?

2)function ORDER-DOMAIN-VALUES: in what order should its values be tried?

3)function INFERENCE: what inferences should be performed at each step in the search?

4)When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

var ← SELECT-UNASSIGNED-VARIABLE(csp)

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, {X1, X2,...}. This static variable ordering seldom results in the most efficient search.

Minimum-remaining-values (MRV) heuristic

The idea of choosing the variable with the fewest "legal" value. A.k.a. "**most constrained variable**" or "fail-first" heuristic, it picks a variable that is most likely to cause a failure soon thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately avoiding pointless searches through other variables.

E.g. After the assignment for WA=red and NT=green, there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q.

Degree heuristic

The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. Degree heuristic attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.

e.g.

- SA is the variable with highest degree 5; the other variables have degree 2 or 3; T has degree 0.
- once SA is chosen, applying the degree heuristic solves the problem without any false steps you can choose any consistent color at each choice point and still arrive at a solution with no backtracking

Least Constraining Value

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. **Least-constraining-value** prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

We have generated the partial assignment with WA = red and NT = green and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. The heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem

Interleaving search and inference

But inference can be even more powerful in the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

Forward checking

This is one of the simplest forms of inference. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X. There is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA*=*red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	(R)	B	(G)	R B	R G B	B	R G B
After *V*=*blue*	(R)	B	(G)	R	(B)		R G B

Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$ is assigned, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$ is assigned, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

There are two important points to notice about this example. First, notice that after WA = red and Q = green are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q.

A second point to notice is that after V = blue, the domain of SA is empty. Hence, forward checking has detected that the partial assignment {WA = red, Q = green, V = blue} is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately

For many problems the search will be more effective if we combine the MRV heuristic with forward checking.

Forward checking only makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent.

MAC (Maintaining Arc Consistency) algorithm

More powerful than forward checking, detect this inconsistency. After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP,

we start with only the arcs(X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately.

Chronological backtracking

The BACKTRACKING-SEARCH algorithm has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited.

Example with a fixed variable ordering Q, NSW , V , T, SA, WA, NT. Suppose we have generated the partial assignment {Q = red, NSW = green, V = blue, T = red}. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly resolve the problem with South Australia

Backtrack to a variable that was responsible for making one of the possible values of the next variable (e.g. SA) impossible. The set (in this case {Q = red, NSW = green, V = blue, }), is called the **conflict set** for SA. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.

Intelligent backtracking: Looking backward

Forward checking can supply the conflict set with no extra work whenever forward checking based on an assignment X = x deletes a value from Y 's domain, it should add X = x to Y 's conflict set.

If the last value is deleted from Y 's domain, then the assignments in the conflict set of Y are added to the conflict set of X. Then, when we get to Y , we know immediately where to backtrack if needed. In fact, every branch pruned by backjumping is also pruned by forward checking. Hence simple backjumping is redundant in a forward-checking search or in a search that uses stronger consistency checking (such as MAC).

Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs

Conflict-directed back jumping

Consider the partial assignment which is proved to be inconsistent: {WA=red, NSW=red}.We try T=red next and then assign NT, Q, V, SA, no assignment can work for these last 4 variables.

Eventually we run out of value to try at NT, but simple backjumping cannot work because NT doesn't have a complete conflict set of preceding variables that caused to fail.

The set {WA, NSW} is a deeper notion of the conflict set for NT, caused NT together with any subsequent variables to have no consistent solution. So the algorithm should backtrack to NSW and skip over T.

A backjumping algorithm that uses conflict sets defined in this way is called conflict-direct backjumping

When a variable's domain becomes empty, the "terminal" failure occurs, that variable has a standard conflict set. Let X_j be the current variable, let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in $conf(X_j)$, and set

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}.$$

The conflict set for an variable means, there is no solution from that variable onward, given the preceding assignment to the conflict set

e.g. assign WA, NSW, T, NT, Q, V, SA.

SA fails, and its conflict set is {WA, NT, Q}. (standard conflict set)

Backjump to Q, its conflict set is

{NT, NSW}∪{WA,NT,Q} - {Q} = {WA, NT, NSW}.

That is, there is no solution from Q onward, given the preceding assignment to {WA, NT, NSW }. Therefore, we Backtrack to NT, its conflict set is

{WA}∪{WA,NT,NSW}-{NT} = {WA, NSW}.

Hence the algorithm backjump to NSW. (over T)

Constraint learning

After backjumping from a contradiction, how to avoid running into the same problem again:

Constraint learning: The idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. We then record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.

Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem.

Consider the state {WA = red, NT = green, Q = blue}

Forward checking can tell us this state is a no-good because there is no valid assignment to SA. In this particular case, recording the no-good would not help, because once we prune this branch from the search tree, we will never encounter this combination again.

But suppose that the search tree in were actually part of a larger search tree that started by first assigning values for V and T. Then it would be worthwhile to record {WA = red, NT = green, Q = blue} as a no-good because we are going to run into the same problem again for each possible set of assignments to V and T

No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.

The structure of problem

The structure of the problem as represented by the constraint graph can be used to find solution quickly. e.g. The problem can be decomposed into 2 **independent subproblems**: Coloring T and coloring the mainland.

Tree: A constraint graph is a tree when any two varyiable are connected by only one path.

Directed arc consistency (DAC): A CSP is defined to be directed arc-consistent under an ordering of variables $X_1, X_2, \ldots, X_n$ if and only if every X_i is arc-consistent with each X_j for j>i. By using DAC, any tree-structured CSP can be solved in time linear in the number of variables.

How to solve a tree-structure CSP:

Pick any variable to be the root of the tree; Choose an ordering of the variable such that each variable appears after its parent in the tree. (**topological sort**). Any tree with n nodes has n-1 arcs, so we can make this graph directed arc-consistent in O(n) steps, each of which must compare up to d possible domain values for 2 variables, for a total time of $O(nd^2)$.

Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value.Since each link from a parent to its child is arc consistent, we won't have to backtrack, and can move linearly through the variables.

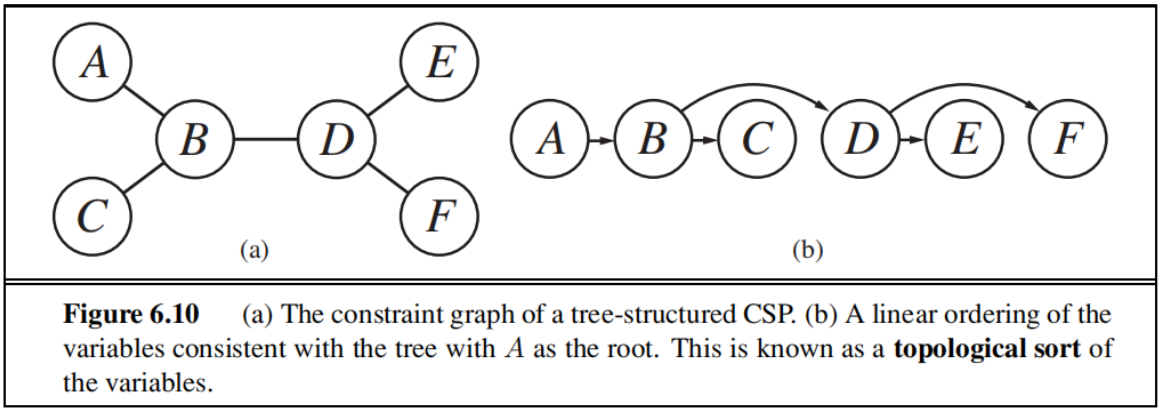


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

There are 2 primary ways to reduce more general constraint graphs to trees:

1. Based on removing nodes;

2. Based on collapsing nodes together

Based on removing nodes;

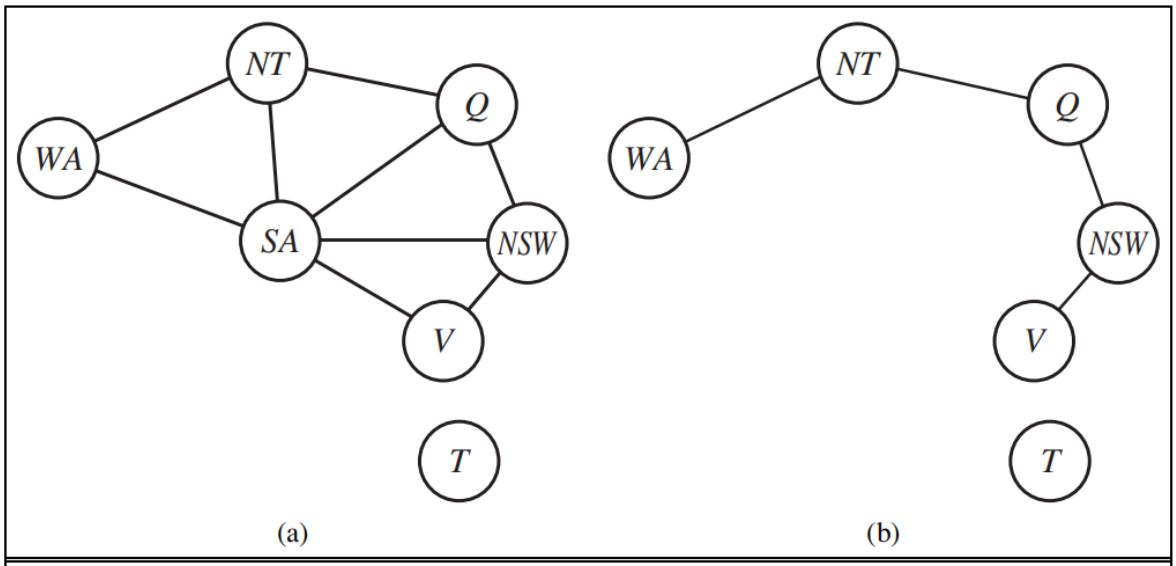


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of *SA*.

Example, We can delete SA from the graph by fixing a value for SA and deleting from the domains of other variables any values that are inconsistent with the value chosen for SA.

The general algorithm:

Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.

For each possible assignment to the variables in S that satisfies all constraints on S,

(a) remove from the domain of the remaining variables any values that are inconsistent with the assignment for S, and

(b) If the remaining CSP has a solution, return it together with the assignment for S.

Time complexity: $O(d^c \cdot (n-c)d^2)$, c is the size of the cycle cut set.

Cutset conditioning: The overall algorithmic approach of efficient approximation algorithms to find the smallest cycle cutset.

Based on collapsing nodes together

Tree decomposition: construct a tree decomposition of the constraint graph into a set of connected subproblems, each subproblem is solved independently, and the resulting solutions are then combined.

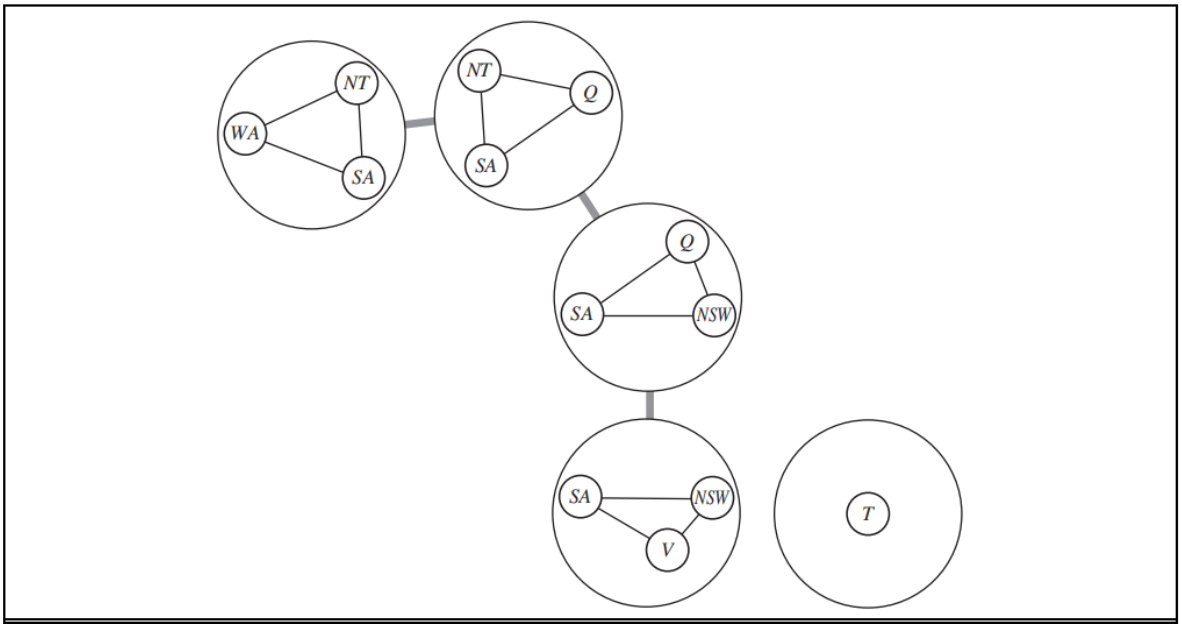


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

A tree decomposition must satisfy 3 requirements:

1. Every variable in the original problem appears in at least one of the subproblems.
2. If 2 variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
3. If a variable appears in 2 subproblems in the tree, it must appear in every subproblem along the path connecting those those subproblems.

We solve each subproblem independently. If any one has no solution, the entire problem has no solution. If we can solve all the subproblems, then construct a global solution as follows: First, view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem. Then, solve the constraints connecting the subproblems using the efficient

algorithm for trees. A given constraint graph admits many tree decomposition; In choosing a decomposition, the aim is to make the subproblems as small as possible.

Tree width

The tree width of a tree decomposition of a graph is one less than the size of the largest subproblems. The tree width of the graph itself is the minimum tree width among all its tree decompositions.

Time complexity: $O(nd^{w+1})$, w is the tree width of the graph.

The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is quite efficient if a small cutset can be found. **Tree decomposition** techniques transform the CSP into a tree of subproblems and are efficient if the **tree width** of constraint graph is small.

The structure in the values of variables

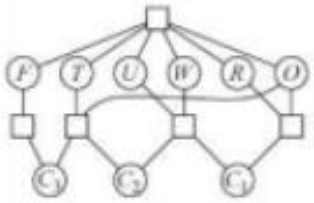
By introducing a **symmetry-breaking constraint**, we can break the **value symmetry** and reduce the search space by a factor of n!. E.g. Consider the map-coloring problems with n colors, for every consistent solution, there is actually a set of n! solutions formed by permuting the color names.(value symmetry). On the Australia map, WA, NT and SA must all have different colors, so there are 3!=6 ways to assign.

We can impose an arbitrary ordering constraint NT<SA<WA that requires the 3 values to be in alphabetical order. This constraint ensures that only one of the n! solution is possible: {NT=blue, SA=green, WA=red}. (symmetry-breaking constraint)

Sample Solution: Cryptarithmetic Problem

Example of solving the cryptarithmetic problem on a paper, using the strategy of backtracking with forward checking and the Minimum Remaining Value Heuristic and least-constraining-value heuristic.

```
  T W O
+ T W O
-------
F O U R
```



Cryptarithmetic problems are mathematical puzzles in which the digits are replaced by letters of the alphabets.

1. Formulate the cryptarithmetic problem as a constraint satisfaction problem (CSP) by giving the variables, their domains, and the constraints.

   The variables C_1, C_2 and C_3 represent the carry digits for the three columns.

   Variable: F, T, U, W, R, O
   Domain: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
   Constraints:
   Alldiff(F, T, U, W, R, O) → The variables F, T, U, W, R, O must have different assigned values.
   The C_1, C_2 and C_3 represent the carries and can take the values {0,1}.
   The number don't have any leading zeros. i.e. 0567 (wrong), 567 (correct).
   After replacing letters by their digits, the resulting arithmetic operations must be correct.

   $O + O = R + 10 * C_1$ ---------- (1)
   $C_1 + W + W = U + 10 * C_2$ ---------- (2)
   $C_2 + T + T = O + 10 * C_3$ ---------- (3)
   $F = C_3$ ---------- (4)

2. Solve this CSP problem using the strategy of backtracking with forward checking and the Minimum Remaining Value Heuristic and least-constraining-value heuristic.
   The backtracking search algorithm runs as follows:
   a. Step 1: Consider the constraint that the number don't have any leading zeros. i.e. 0567 (wrong), 567 (correct).
      This implies that the variables F, T can't take the value 0 and C_3 can take only 0 or 1.
      Thus, we set $F = 1$ and consequently $C_3 = 1$
      $F = 1$

      Now, the domain of values for the variables U, W, R, O is {0,2,3....9} and the domain of values for the variable T is {2,3....9}.

15 (a) (a) Solve the following crypt arithmetic problem by hand, using the strategy of backtracking with forward checking and the MRV and least-constraining-value heuristics. **(8)**

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

Variables:{T,W,O,F,U,R,C1,C2,C3}

Domain of {T,W,O,F,U,R}={0,1,2,3,4,5,6,7,8,9}

Domain of {C1,C2,C3}={0,1}

0 1 2 3 4 5 6 7 8 9

F

C3=1 C2 C1

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

1

Using Minimum Remaining Value Heuristic we choose variable with fewest legal values, Here we take C3 whose domain={0,1}

Since F cannot be 0, by forward checking we can eliminate 0 from C3. Since F is a carry eliminate other options too

So C3=1 F=1

T	W	O	F	U	R	C3	C2	C1
~~0~~,1,2,3,4,5,6,7,8,9	~~0~~,1,2,3,4,5,6,7,8,9	~~0~~,1,2,3,4,5,6,7,8,9	~~0~~,1,~~2,3,4,5,6,7,8,9~~	~~0~~,1,2,3,4,5,6,7,8,9	~~0~~,1,2,3,4,5,6,7,8,9	~~0~~,1	0,1	0,1

Now C2 and C1 are the MRV so we choose C2(D={0,1}) both have no issue in Forward checking

0 1 2 3 4 5 6 7 8 9

F

C3=1 C2=0 C1

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

1

Now C2 and C1 are the MRV so we choose X2(D={0,1)) both have no issue in Forward checking. Lets choose C2=0

T	W	O	F	U	R	C3	C2	C1
~~0,1,2,3,~~ 4,5,6,7, 8,9	~~0,1,2,3,~~ 4,5,6,7, 8,9	~~0,1,2,3,~~ 4,5,6,7, 8,9	0 ~~1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ 4,5,6,7, 8,9	~~0,1,2,3,~~ 4,5,6,7, 8,9	~~0,1~~	~~0,1~~	0,1

0 1 2 3 4 5 6 7 8 9

F

C3=1 C2=0 C1=0

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

1

Now C1 has Minimum Remaining Values so we choose C1(D={0,1)) both have no issue in Forward checking. Lets choose C1=0

T	W	O	F	U	R	C3	C2	C1
~~0,1,2,3,~~ 4,5,6,7, 8,9	~~0,1,2,3,~~ 4,5,6,7, 8,9	~~0,1,2,3,~~ 4,5,6,7, 8,9	0 ~~1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ 4,5,6,7, 8,9	~~0,1~~	~~0,1~~	~~0,1~~

0 1 2 3 4 5 6 7 8 9

F

C3=1 C2=0 C1

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \\ 1 & 4 & & 8 \end{array}$$

The variable O must be even number(T+T will always be even) and it should be less than 5 since

O+O=R+10X0

2O=R //if O is more than 5 then R become greater than 10 violating Constraint of C2=0. Lets Arbitarily choose O=4 So make R=8

T	W	O	F	U	R	C3	C2	C1
~~0,1,2,3,~~ 4,5,6,7, ~~8,9~~	~~0,1,2,3,~~ 4,5,6,7, ~~8,9~~	~~0,1,2,3,~~ 4 ~~5,6,7,~~ ~~8,9~~	0 ~~1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ 4,5,6,7, ~~8,9~~	~~0,1,2,3,~~ ~~4,5,6,7,~~ 8 9	~~0,1~~	0 ~~1~~	0 ~~1~~

0 1 2 3 4 5 6 7 8 9

F

C3=1 C2=0 C1

$$\begin{array}{cccc} & T_7 & W & O \\ + & T & W & O \\ \hline F & O & U & R \\ 1 & 4 & & 8 \end{array}$$

Propagating these constraints to T, we get 2T=14 , ie, T=7

T	W	O	F	U	R	C3	C2	C1
~~0,1,2,3,~~ 4,5,6 7 ~~8,9~~	~~0,1,2,3,~~ 4,5,6,7, ~~8,9~~	~~0,1,2,3,~~ 4 ~~5,6,7,~~ ~~8,9~~	0 ~~1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ 4,5,6,7, ~~8,9~~	~~0,1,2,3,~~ ~~4,5,6,7,~~ 8 9	~~0,1~~	0 ~~1~~	0 ~~1~~

0 1 2 3 4 5 6 7 8 9

F

C3=1 C2=0 C1

$$\begin{array}{cccc} & T_7 & W & O \\ + & T & W & O \\ \hline F & O & U & R \\ 1 & 4 & 6 & 8 \end{array}$$

U must be an even no less than 9(2W=U)

Only variable U survive forward checking is 6

T	W	O	F	U	R	C3	C2	C1
~~0,1,2,3,~~ ~~4,5,6~~ 7 ~~8,9~~	~~0,1,2,3,~~ 4,5,6,7, ~~8,9~~	~~0,1,2,3,~~ 4 ~~5,6,7,~~ ~~8,9~~	0 ~~1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ 4,5 ~~6~~ 7, ~~8,9~~	~~0,1,2,3,~~ ~~4,5,6,7,~~ 8 9	~~0,1~~	0 ~~1~~	0 ~~1~~

0 1 2 3 4 5 6 7 8 9

F

Only variable left is W(2W=U) which is left with value 3

C3=1 C2=0 C1

$$\begin{array}{cccc} & T_7 & W_3 & O \\ + & T & W & O \\ \hline F & O & U & R \\ 1 & 4 & 6 & 8 \end{array}$$

T	W	O	F	U	R	C3	C2	C1
~~0,1,2,3,~~ ~~4,5,6,~~7 ~~8,9~~	~~0,1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,~~1,2,3, ~~4,5,6,7,~~ ~~8,9~~	~~0,1,2,3,~~ ~~4,5,~~6,~~7,~~ ~~8,9~~	~~0,1,2,3,~~ ~~4,5,6,7,~~ ~~8,9~~	~~0,~~1	~~0,~~1	~~0,~~1