Representation and Reasoning for Intelligent Systems

Stefan Klaus

NOT PUBLISHED

BOOK-WEBSITE.COM

First printing, March 2013

Contents

1. Chapter 1:Constraint Satisfaction Problem

1.1 Constraint Satisfaction Problem

Constraint satisfaction problems(CSP) are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. One example of a CSP is the game “Sudoku”

1.2 Basic terms about CSP:

- an assignment is to assign values to some or all variables
- an assignment that does not violate any constraints is called a consistent assignment. With values from domain D_i
- A complete assignment is one in which each variable is assigned
- a partial assignment is one that assigns value to some of the variables

Constraint graph

Binary CSP: each constraint relates at most two variables, e.g: WASA constraint graph: nodes are variables(e.g. region WA), arcs show constraints(e.g. WASA)

Varieties of Variables

Discrete variables

- Finite Domains
- boolean CSPs, include: boolean satisfiability(NP - complete)
- Sudoku
- Infinite Domains(Integers, Strings, etc.)
- job scheduling, variables are start/end days for each job, need a constraint language, e.g. $StartJob_1 + 5 \leq StarJob_3$

Continuous variables

- start /end times for Hubble Telescope observations
- Unary constraints involve a single variable
    - $SA \neq Green$

- binary constraints involve pairs of variables
  - $SA \neq WA$
- Higher-order constraints involve 3 or more variables
  - cryptarithmetic column constraints
- Preferences(soft) constraints
  - red is better than green
  - CSPs with preference are often with optimization search algorithms constraints optimization problems

1.2.1 Node Consistency

If a node is node-consistent if all the value's domain satisfy the variable's unary constraints.

Example:
D = Red, Green, Blue
Variable X
X dislikes green, then X starts with D = Red, Green, Blue, and becomes node consistent after eliminating Green. X is node consistent with the reduced domain, D=Red, Blue.

Arc Consistency
X_i is arc-consistent with respect to another variables X_j if for every value in X_i's current domain D_i there is one value in X_i's domain D_j that satisfies the binary constraint on the arc (X_i, X_j)

Example:
Given two variables X_i , X_j with values in 0,1,2,...9 and constraints (0,0),(1,1),(2,4),(3,9).
To make X_i arc-consistent with respect to X_j, we reduce X_i's domain to 0,1,2,3.
To make X_j arc-consistent with respect to X_i, we reduce X_j's domain to 0,1,4,9.

1.3 Arc Consistency Algorithm(AC-3)

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

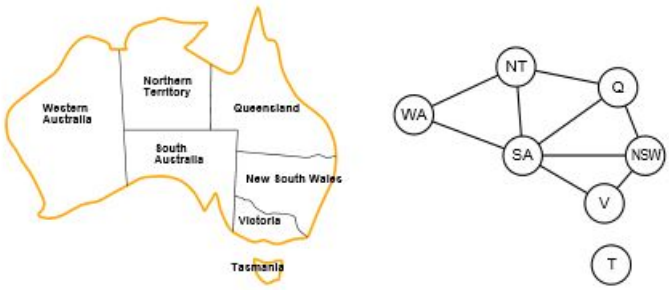
  while $queue$ is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to $queue$
  return $true$

---

function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

1. initially let a queue contain all arcs
2. remove an arc (X_i, X_j) from the queue and make the variable X_i arc-consistent to X_j
   (a) IFX_i domain D_i is unchanged, then check the next arc in the queue
   (b) IF X_i 's domain D_i is revised(smaller), then add all arcs (X_k, X_i)in the queue
   (c) If X_i 's domain D_i is empty, then CSP no solution
3. Keep checking all arcs in the queue until the queue is empty

1.4 Path consistency



A two variable set X_i, X_j is path consistency with respect to a third variable X_m if for every assignment $X_i = a, X_j = b$ consistent with the constraints onX_i, X_j, where is an assignment to X_m ,that satisfies the constraints onX_i, X_m and X_m, X_j

Example:
Can we color the Australia map with two colors?
Make the set WA, SA path consistent with respect to NT?
Assignments: WA = blue, SA = red or WA =blue,SA=red

But no assignment exists for NT

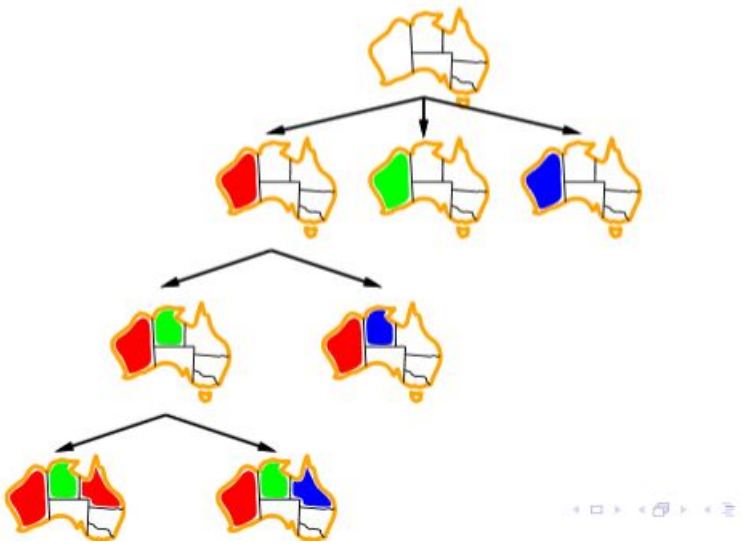
Constraint propagation

- constraint propagation is a specific type of inference
  - use 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
- node consistency
- arc consistency
- path consistency
- k-consistency: k variables involved
- Global consistency

Limits

- Indeed AC-3 works for the easiest Sudoku puzzles
- slightly harder ones can be solved by PC-2, but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle
- To solve the hardest puzzles and to make efficient progress, we will have to be more clever

1.5 Backtracking



1. Select an unassigned value
2. assign values
3. Depth-first search
4. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value

Backtracking search is a depth-first search for CSPs

- choose values for one variable at a time
- backtrack when a variables has no legal left to assign

Backtracking search is the basic uninformed algorithm for CSP. Can solve n-queens for $n \approx 25$.

```
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
```

1.5.1 Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

1. which variable should be assigned next?
2. in what order should values be tried‘?
3. can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Minimum remaining values

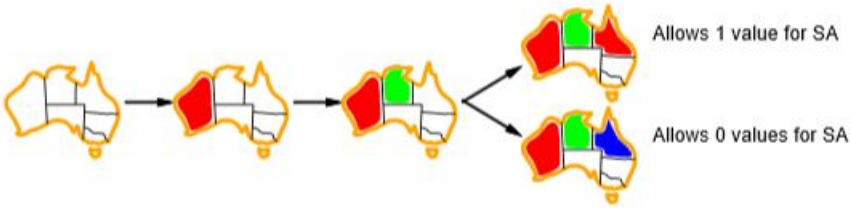
Minimum remaining values (MRV): choose the variable with the fewest legal values.

Degree heuristic

- Tie-breaker among MRV variables
- Degree heuristic: choose the variable with the most constraints on remaining variables

Least constraining value

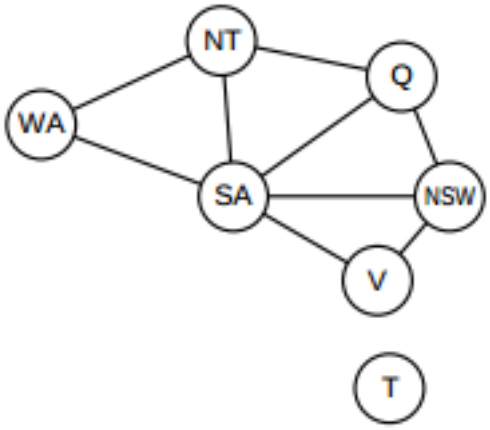
Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables.

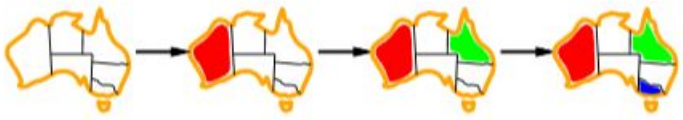


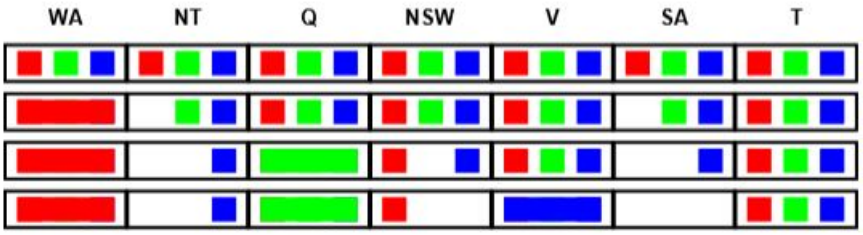
Combining these heuristics makes 1000 queens feasible.

1.5.2 Forward checking

Idea: Keep track if remaining legal values for unassigned variables.
Terminates search when any variable has no legal values.

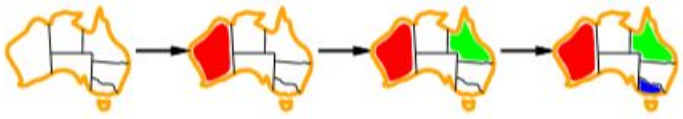


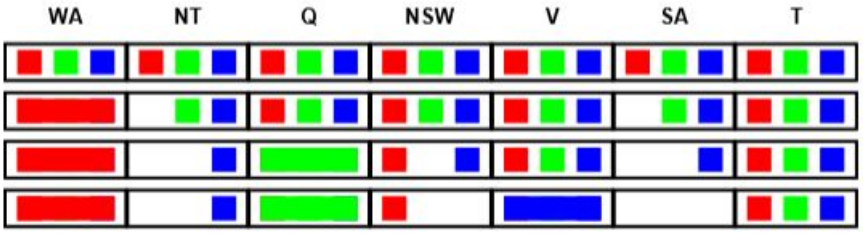




Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:





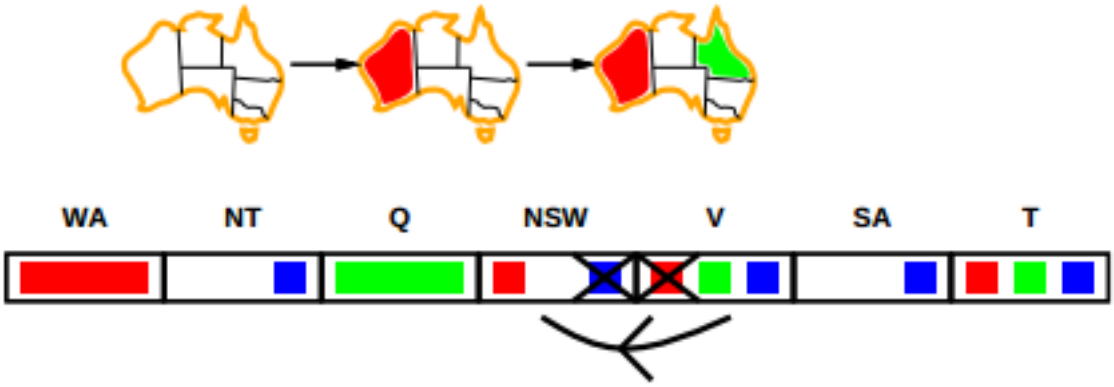
NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally.

1.5.3 Arc consistency

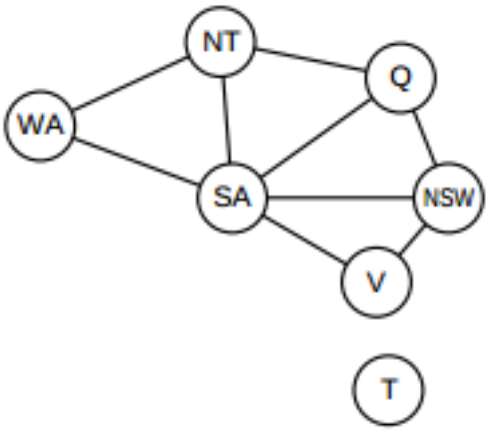
Simplest form of propagation makes each arc consistent.
$X \to Y$ is consistent if for every value x of X there is some allowed y

If X looses a value, neighbours needs to be rechecked.



Arc consistency detects failure earlier than *forward checking*. Can be run as a predecessor or after each assignment.

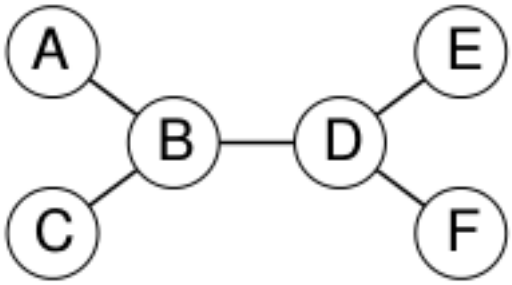
1.6 Structure



Tasmania and mainland Australia are independent subsections which can be identified as connected components of constraint graph.

- Suppose each subproblem has a c variable out of n total
- Worst-case solution cost is $n/c * d^c$ liniar in n
- E.g. $n = 80, d = 2, c = 20$
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $2 * 2^{20} = 0.4$ seconds at 10 million nodes/sec

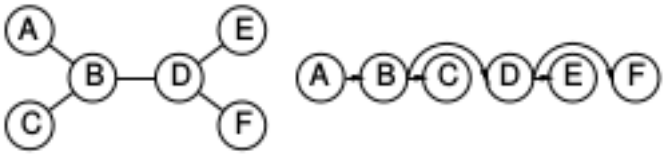
1.6.1 Tree-structure CSPs



- Theorem: if the constraitn graph has no loops, the CSP can be solved in $O(nd^2)$ time
- Compare to general CSPs, where worst-case time is $O(d^n)$
- This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning

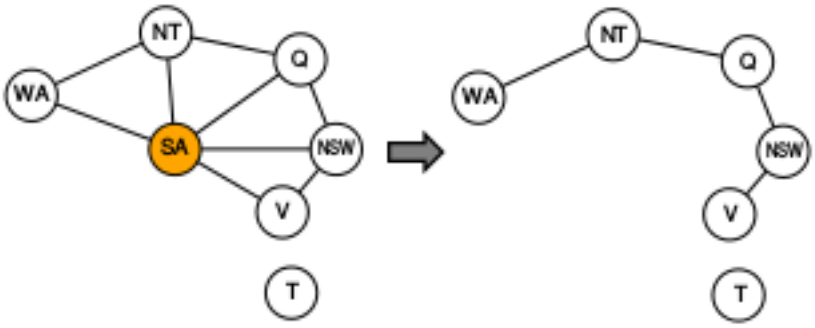
1.6.2 Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering
2. For j from n down to 2, apply $RemoveInconsistent(Parent(X_j), x_j)$
3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$



1.6.3 Nearly tree-structured CSPs

- Conditioning: instantiate a variable, prune its neighbours domains
- Cutset conditioning: instantiate(in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size $c \Rightarrow runtime O(d^c * (n-c)d^2)$, very fast for small c



- Choose a subset S from $VARIABLE[csp]$ such that the constraint graph becomes a tree after removal of S
- For each possible assignment to the variables in S satisfies all constraint on S
  - remove from the domains of the remaining variables any values that are inconsistent with the assignment of S
  - If the remaining CSP has solution, then return it together the assignment to S

1.7 Local Search for CSPs

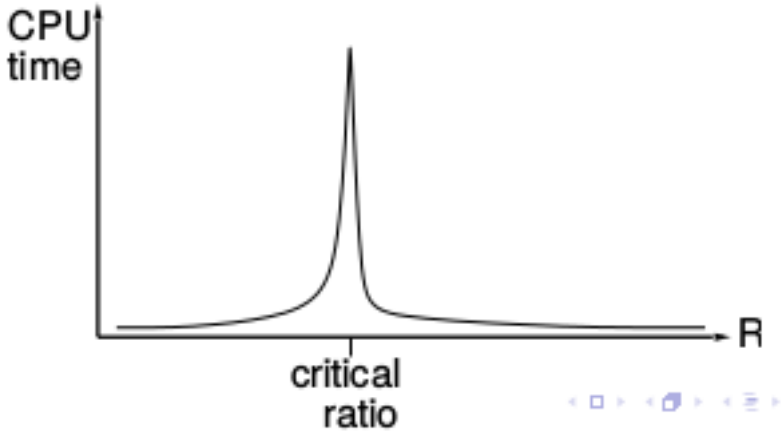
- Local search, e.g. hill-climbing, simulated annealing for CSP
  - typically start with a "complete" state, i.e. all variables assigned to values, but may violated constraints
  - then search changes the value of one variable at each time for violated constraints
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristics:
  - choose value that violates the fewest constraints i.e. hillclimb with $h(n)$ =total number of violated constraints

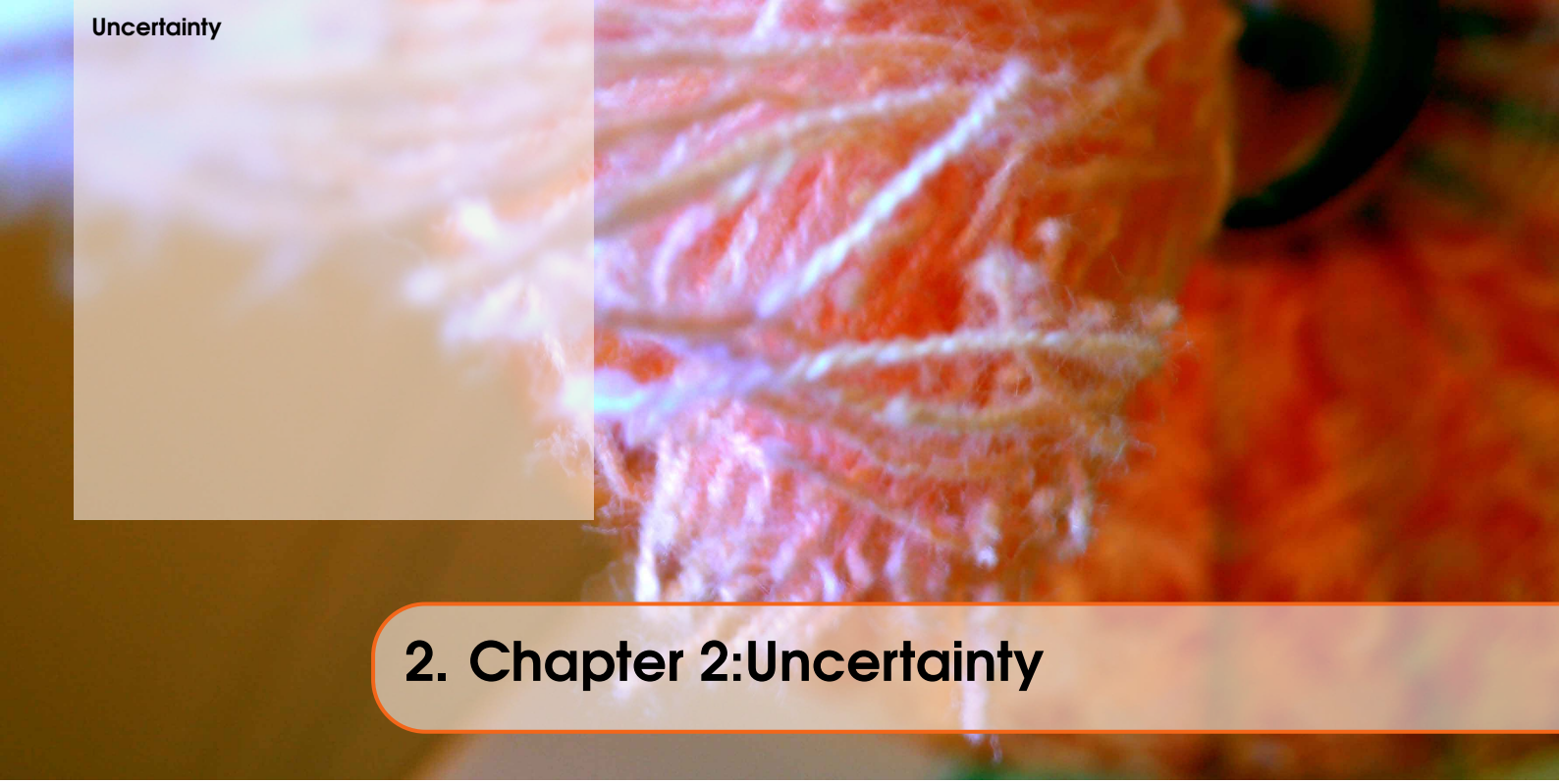
1.7.1 Min-Conflict Algorithms

function MIN-CONFLICTS(csp, max_steps) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen conflicted variable from csp.VARIABLES
    $value \leftarrow$ the value v for var that minimizes CONFLICTS(var, v, $current$, csp)
    set $var = value$ in $current$
  return $failure$

- Given random initial state, can solve *n-queens* in almost constant time for arbitrary n with high probability (e.g. $n = 10000000$)
- The same to be true for any randomly-generated CSP *except* in a narrow range of the ratio $R = \frac{numberofconstraints}{numberofvariables}$





2. Chapter 2:Uncertainty

2.1 Uncertainty

3. Presenting Information

3.1 Table

Treatments	Response 1	Response 2
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

Table 3.1: Table caption

3.2 Figure

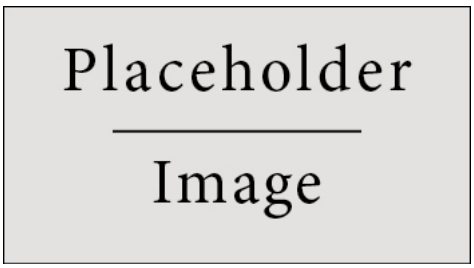


Figure 3.1: Figure caption

4. Chapter 7:Case-Based Reasoning

4.1 Case-Based Reasoning

- Case law: made up of, and from, hundred and thousands of precedent cases
- Principle: cases with similar facts should be treated in similar ways
- In cases where the parties disagree on what the law is: looking to past precedential decisions of relevant courts
- If a similar dispute has been resolved in the past: following the reasoning used in the prior decision
- If the current dispute is fundamentally distinct from all previous cases: creating new
- CBR: analogous to human experts solving a problem through employing their relevant past experience
  - exemplar-based reasoning
  - instance-based reasoning
  - memory-based reasoning
  - case-based reasoning
  - analogy-based reasoning
- if the new problem has some novel aspects, aspects, then the solution to the new problem is added to the case base
- Different from diagnostic fault tree or rule-based system
  - memory-based problem-solving
  - reusing past experiences

Domains that CBR Works well:

- Broad but shallow domain
  - not a single tree, but a forest of small trees
  - a number of loosely connected problems that must be dealt with
  - need different kinds of expertise
- Experience, rather than theory, is the primary source of knowledge
  - Many past examples of problems that occur
  - rather than having a deep understanding of the domain
- solutions are reusable
  - old solution is useful for a new problem
  - if each problem is different, then there is little to be gained by trying to reuse past

solutions

4.2 Case-Based Reasoning System and 4R Cycle

- Input: new problem
- Output: a solution to the new problem
- case base
  - store cases(experience)

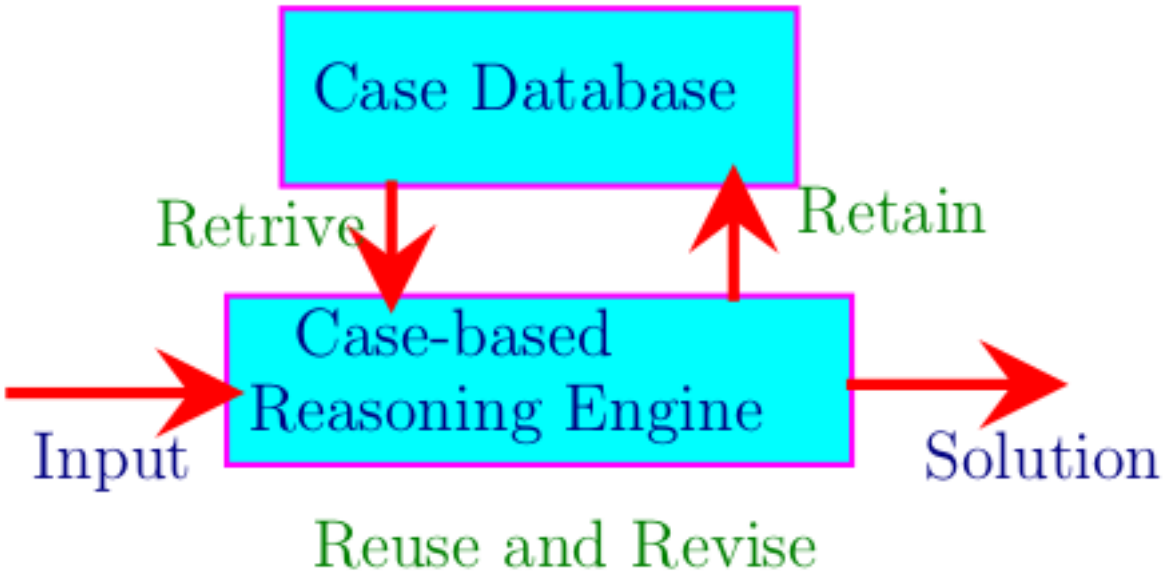


Figure 4.1: The Case-Based Reasoning System

4.2.1 4R Cycle

- Retrieve: relevant cases, match most similar cases, retrieve solutions from theses cases
- Reuse: solutions in stored cases
- Revise: the retrieved solution(s) to reflect differences between new case and retrieved case(s)
- Retain: new cases into database

See figure4.2 for visualisation.

New Problem vs Old Case

- Observations define a new problem
- Compare similarity of each feature
- Not all feature values may be known
- Some features may be more important
- New problem = case without a solution
- Similarity by weighted average

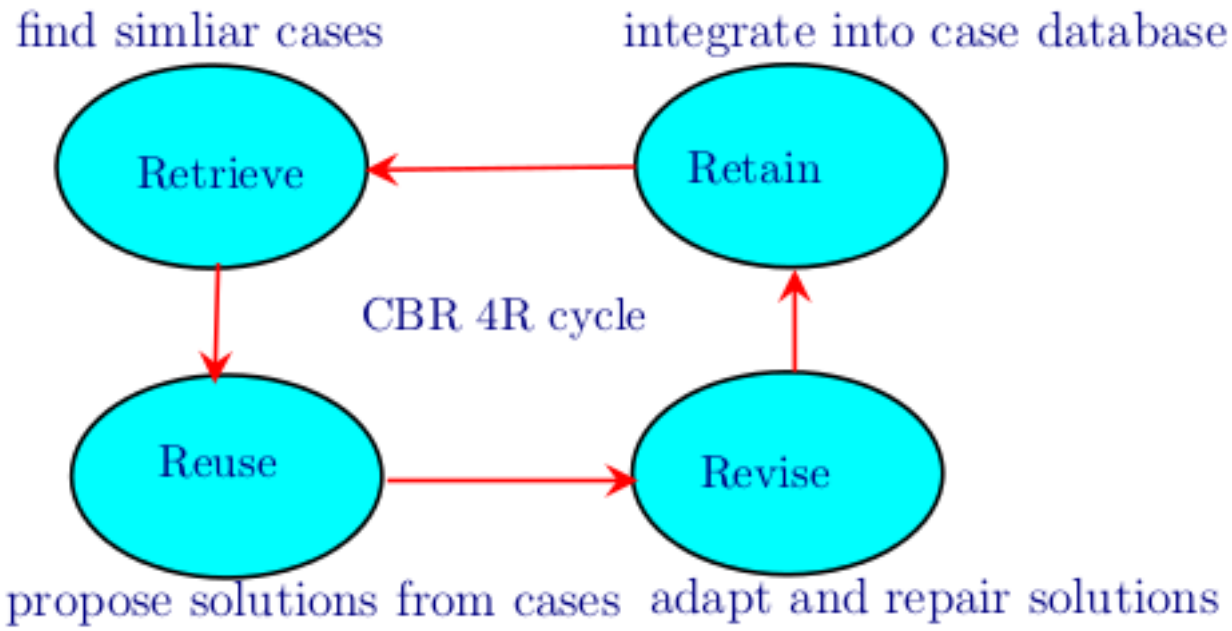


Figure 4.2: The Case-Based 4R cycle

4.3 Design Case-Based Reasoning System

4.3.1 Case Representation

A case in diagnosis represents one diagnostic situation, include two parts:

1. Features
   (a) symptoms
   (b) failure
   (c) feature values
   (d) repair strategies
   (e) test time and cost
2. Solutions
   (a) cause of failure
   (b) replace or repair fault unit

4.3.2 Design Case database

- Dependent on the structure and content of its collection of cases
  - Deciding what to store in a case
  - Finding an appropriate structure for describing case contents
  - Deciding how the case memory should be organized and indexed for effective retrieval and reuse

4.3.3 Retrieve: index

- Retrieve a case from case database
- Select indexes
  - Similar to books in library, index may help search cases in case database

Match Case

- Compare features and their value between the stored case and new problem
- Nearest-neighbour matching algorithm

4.3.4 Retrieval: Ranking

- Possibly more than one case is matched
- Among matched cases, ranking may be used to choose a case to reuse
- If a matched case cannot provide the solution to the problem,. lower rank cases may be taken as the candidate for the problem
- Ranking value will depend on observation time and cost
  - Higher the rank $\longrightarrow$ better solution(cheaper, faster, etc.)
  - Ranking Observation cost, observation time and case frequency need to be considered

4.3.5 Reuse/Revive

- Adapt/repair old solutions
- Different approaches
  - Substitution
  - Parameter adjustment(via specialized heuristics, e.g. Judge)
  - Local search(replacing fruits in a recipe)
  - Special purpose adaptation and repair
  - Model-based

4.3.6 Retain: Store new cases and stop reasoning

- Store all new cases
- Or store selected new cases(based on certain criteria?)
- The reasoning stops until a satisfied solution(from a solved case) is found
- or stop the reasoning procedure by the system

Bibliography

Index