Representation and Reasoning for Intelligent Systems

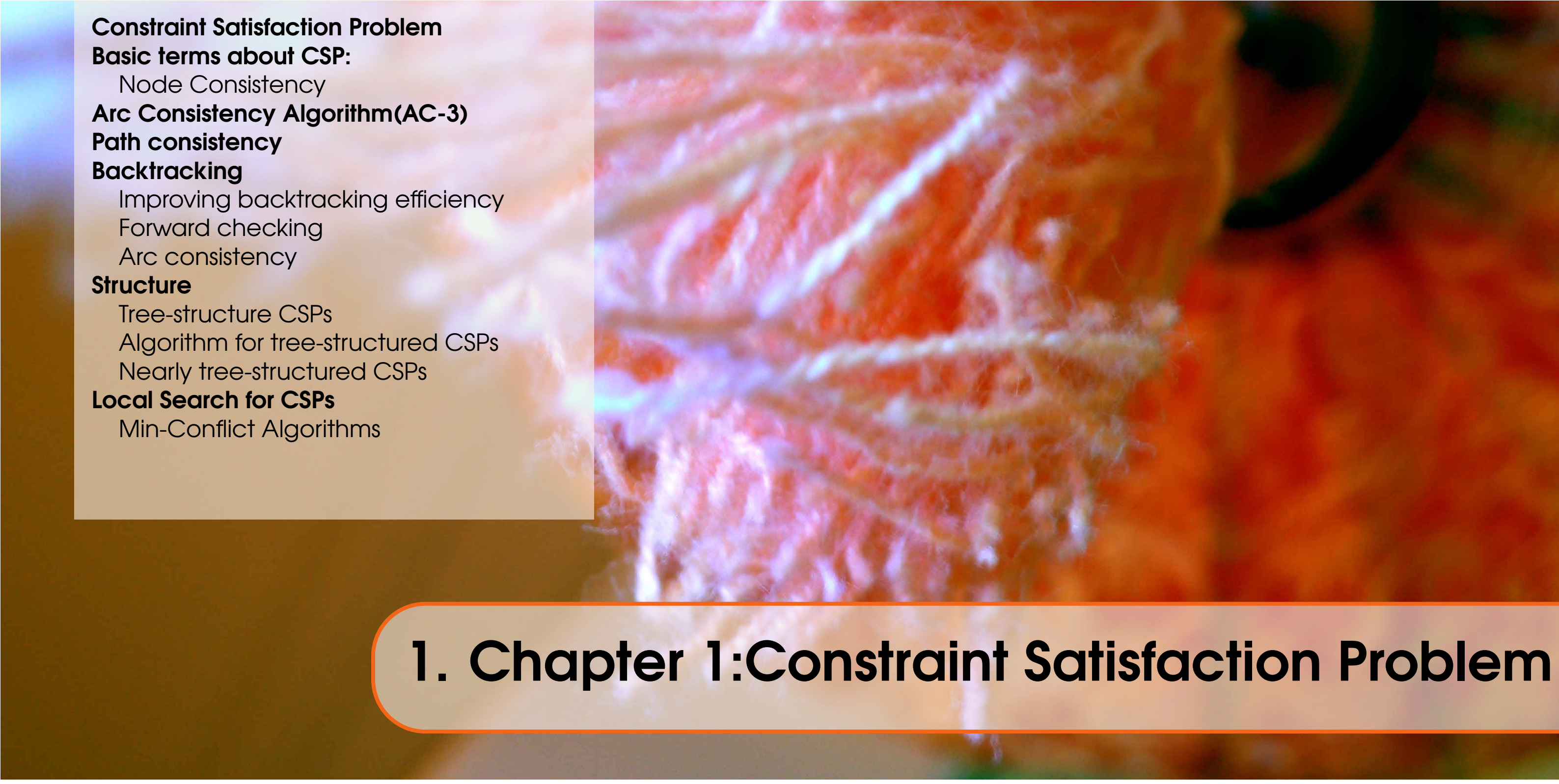
Stefan Klaus

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. $StartJob_1 + 5 \leq StarJob_3$

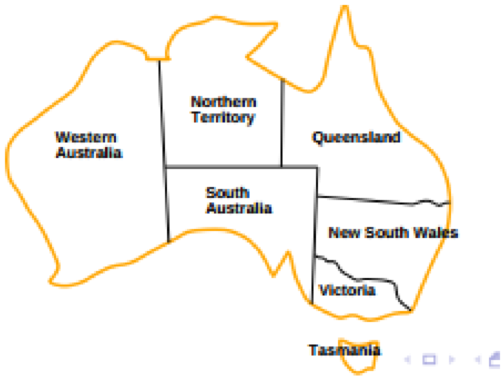
Continuous variables

- 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.

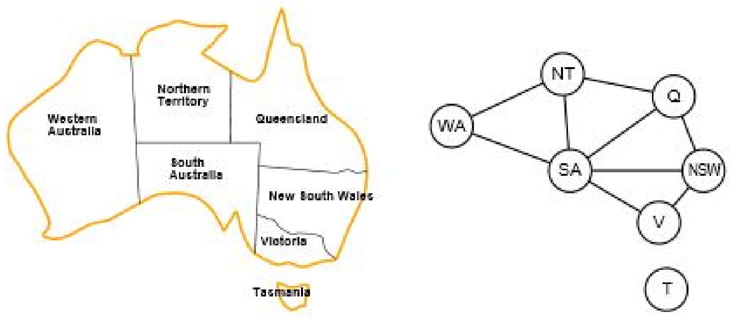
1.3 Arc Consistency Algorithm(AC-3)

```
Input:
  A set of variables X
  A set of domains D(x) for each variable x in X. D(x) contains vx0, vx1...
      vxn, the possible values of x
  A set of unary constraints R1(x) on variable x that must be satisfied
  A set of binary constraints R2(x, y) on variables x and y that must be
      satisfied

Output:
  Arc consistent domains for each variable.

function ac3 (X, D, R1, R2)
// Initial domains are made consistent with unary constraints.
    for each x in X
        D(x) := { x in D(x) | R1(x) }
    // 'worklist' contains all arcs we wish to prove consistent or not.
    worklist := { (x, y) | there exists a relation R2(x, y) or a relation
        R2(y, x) }

    do
        select any arc (x, y) from worklist
        worklist := worklist - (x, y)
        if arc-reduce (x, y)
            if D(x) is empty
                return failure
            else
                worklist := worklist + { (z, x) | z != y and there exists a
                    relation R2(x, z) or a relation R2(z, x) }
    while worklist not empty

function arc-reduce (x, y)
    bool change = false
    for each vx in D(x)
        find a value vy in D(y) such that vx and vy satisfy the constraint
            R2(x, y)
        if there is no such vy {
            D(x) := D(x) - vx
            change := true
        }
    return change
```

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

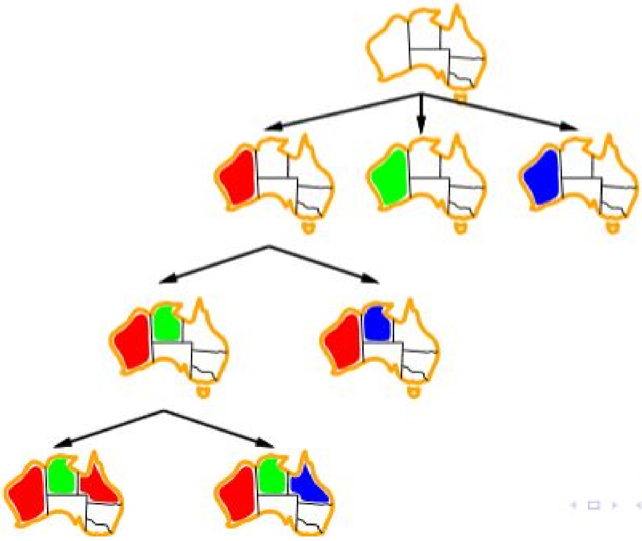
Constraint propagation

- 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

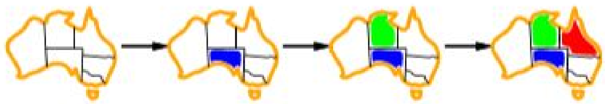
General-purpose methods can give huge gains in speed:

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?

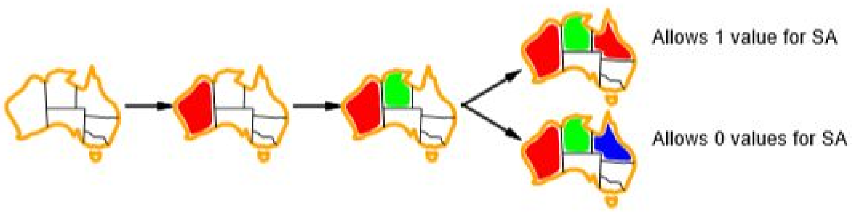
Minimum remaining values
Minimum remaining values (MRV): choose the variable with the fewest legal values.

Degree heuristic

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



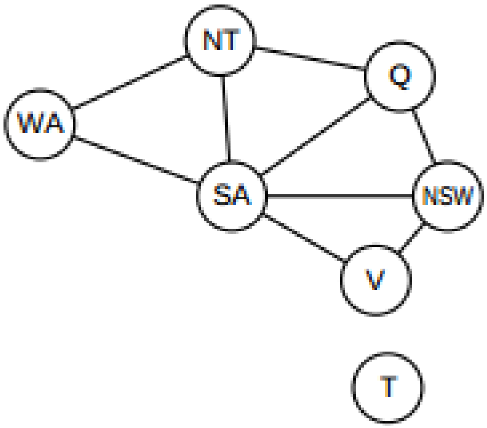
Least constraining value
Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables.

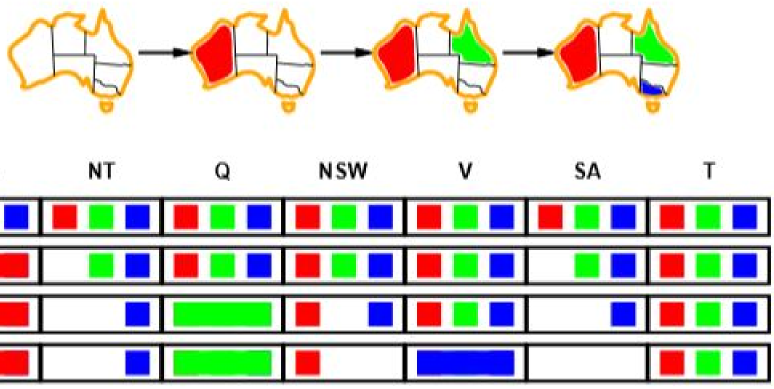


Combining these heuristics makes 1000 queens feasible.

1.5.2 Forward checking

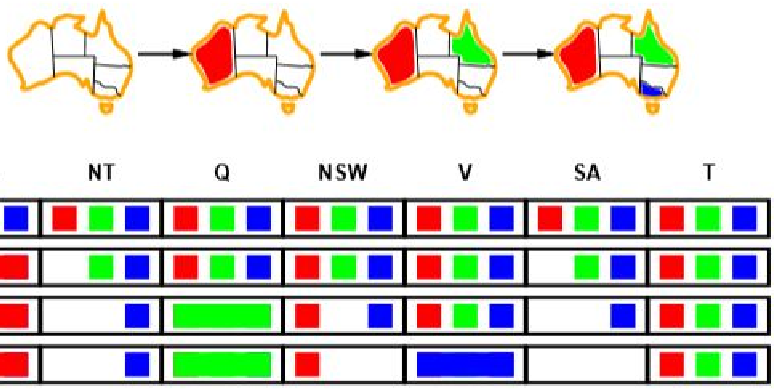
Idea: Keep track if remaining legal values for unassigned variables.
Terminates search when any variable has no legal values.





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!

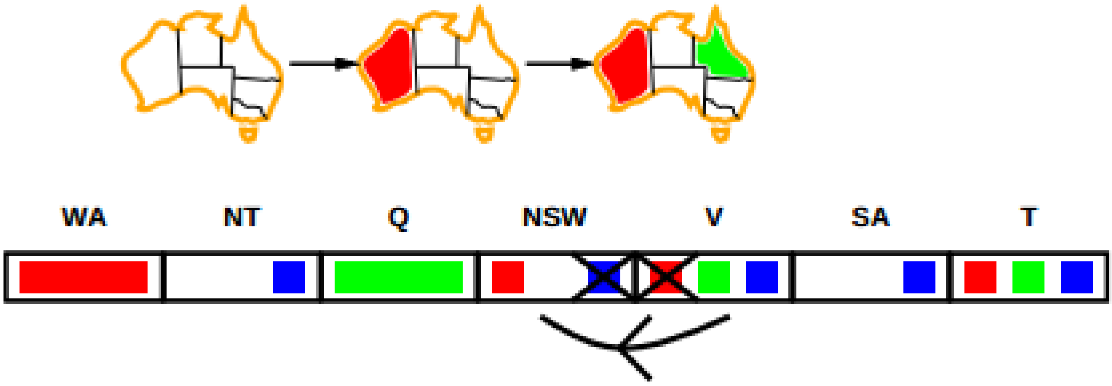
Constraint propagation repeatedly enforces constraints locally.

1.5.3 Arc consistency

Simplest form of propagation makes each arc consistent.

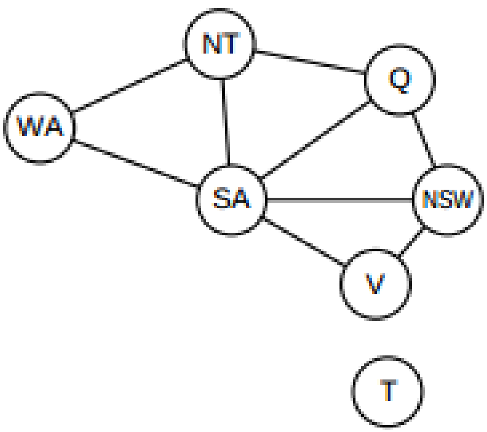
$X \to Y$ is consistent if for every value x of X there is some allowed y

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.

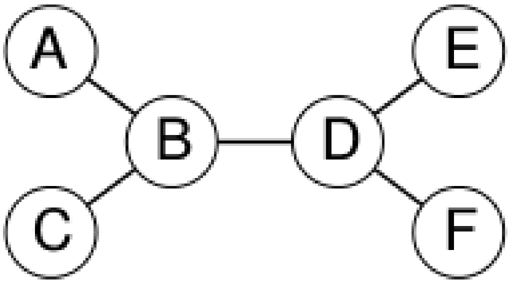
1.6 Structure



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

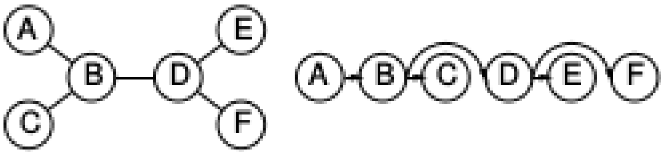
1.6.1 Tree-structure CSPs



- 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

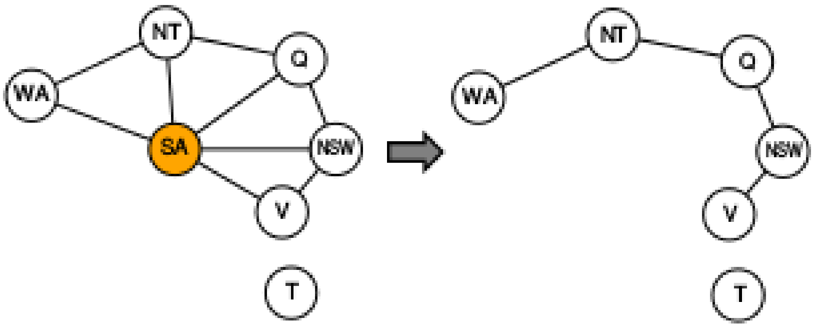
1.6.2 Algorithm for tree-structured CSPs

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

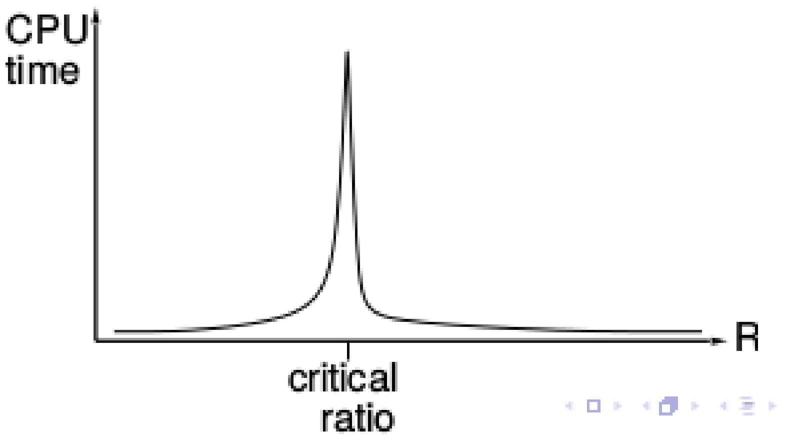
1.7 Local Search for CSPs

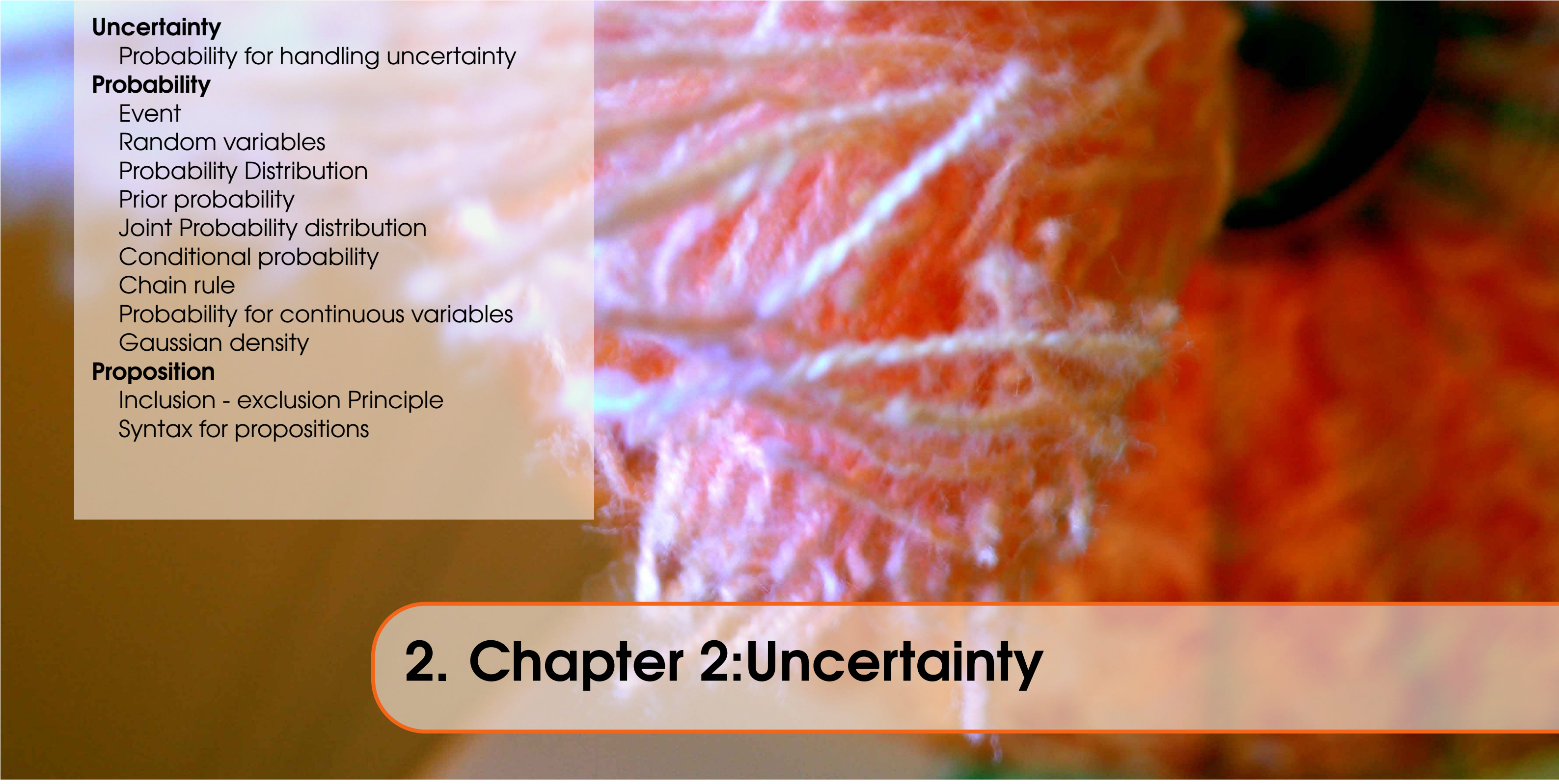
- Local search, e.g. hill-climbing, simulated annealing for CSP
  - typically start with a "complete" state, i.e. all variables assigned to values, but may violated constraints
  - then search changes the value of one variable at each time for violated constraints
- Variable selection: randomly select 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

```
algorithm MIN-CONFLICTS
    input: csp, a constraint satisfaction problem
         max_steps,the number of steps allowed before giving up
         current_state, an initial assignment of values for the variables in
            the csp
    output: a solution set of values for the variable or failure
  for i=1 to max_steps do
     if current_state is a solution of csp then return current_state
     var <-- a randomly chosen variable from the set of conflicted variables
        CONFLICTED[csp]
     value <-- the value v for var that minimizes
        CONFLICTS(var,v,current,csp)
     set var = value in current_state
  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 $R = \frac{number of constraints}{number of variables}$





2. Chapter 2:Uncertainty

2.1 Uncertainty

A purely logical approach may not work very well for statements which include multiple uncertain factors(e.g. Road accident on your way to the airport? Road work? Flooding? Nazi Zombies?
A purely logical approach would state: "A_8 will get me there on time."
Considering uncertain factors this would change to:"If there's no accident on the bridge, and it does not rain, and my tires remain intact etc, THEN A_8 will get me there on time"

Reasons for that:

- Failure to enumerate exceptions, qualifications etc.
- No complete theory for the domain
- Lack of relevant facts, initial conditions and so on

2.1.1 Probability for handling uncertainty

Probabilistic provides a way of summarizing the uncertainty.
An example for that would be: conditional probabilities can be used to represent:

- Given the available evidence, A_8 will get me there on time with probability 0.8
- Given the available evidence, $A_1 0$ will get me there on time with probability 0.9
- Given the available evidence, $A_1 2$ will get me there on time with probability 0.99

Condition	**Result**	**Probability**
Given the available evidence	A_8	0.9
Given the available evidence	$A_1 0$	0.99
Given the available evidence	$A_1 2$	0.999

Table 2.1: Table representation of the different plans

Probability theory is a main tool for dealing with degrees of belief.

Making decisions

Suppose given the following statements:

P(A_8 gets me there on time | Condition = 0.9
P(A_{10} gets me there on time | Condition = 0.99
P(A_{12} gets me there on time | Condition = 0.999
P(A_{24} gets me there on time | Condition = 0.9999

Given that, what actions to choose?

- Depends on preferences, e.g. the length of the wait at the airport

Utility theory is used to represent and infer preferences.
Decision theory = utility theory + probability theory

2.2 Probability

Subjective or Bayesian probability:

- Probabilities relate propositions to one's own state of knowledge, e.g. $P(A_8|noreportedaccidents) = 0.9$
- But might be learned from past experience from past experiences of similar situations
- Probabilities of propositions change with new evidence: e.g. $P(A_8|noreportedaccidents, leaveat5am) = 0.95$

Sample point:
A set *Omega*- the sample space: $\omega \in \Omega$ is a sample point or atomic event, e.g. 6 possible rolls of a dice.
A probability model/space is a sample space with assigning a probability for every $\omega \in \Omega$

- $P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = 1/6$
- Property: $0 \leq P(\omega) \leq 1$ for every $\omega \in \Omega$, $\sum_{\omega \in \Omega} P(\omega) = 1$

2.2.1 Event

An event A is any subset of Ω.

$$P(A) = \sigma_{\omega \in A} P(\omega) \tag{2.1}$$

- An event is 'dice roll is less than 4'
- The probability of the event happening is $P(diceroll < 4) = P(1) + P(2) + P(3) = 1/2$

In AI's language, the sets are decribed by propositions:

- ϕ is "dice roll is less than 4"
- $P(\phi) = \Sigma_{\omega \in \phi} P(\omega)$

2.2.2 Random variables

A random variable $X : \Omega \rightarrow R$ is a function from sample points to some range, e.g., the reals or Boolean.
Let $X(\omega)$ be a Boolean variable to represent whether a dicing result ω is odd, then:

$X(1) = true$
$X(2) = False$

But usually written in short Odd $P(Odd)$

2.2.3 Probability Distribution

For a random variable X taking values from $x_1, \ldots, x_k$ probability distribution $P(X = x_i)$ is the probability of X taking the value of x_i:

$$\begin{aligned} P(odd = true) &= P(1) + P(3) + P(5) \\ &= 1/6 + 1/6 + 1/6 \\ &= 1/2 \\ P(Odd = false) &= P(2) + P(4) + P(6) \\ &= 1/6 + 1/6 + 1/6 \\ &= 1/2 \end{aligned}$$

2.2.4 Prior probability

Prior or unconditional probabilities refer to degrees of belief in propositions in the absence of any other information

- $P(Cavity = true) = 0.1$
- $P(Weather = Sunny) = 0.72$

Probability distribution gives values for all possible probabilities:
$P(Weather) =< 0.72, 0.1, 0.08, 0.1 >$normalized, i.e. sums to 1.

$$\begin{aligned} P(Weather = sunny) &= 0.72 \\ P(Weather = rain) &= 0.1 \\ P(Weather = cloudy) &= 0.08 \\ P(Weather = snow) &= 0.1 \end{aligned}$$

2.2.5 Joint Probability distribution

Joint Probability distribution for a set of r.v.s gives the probability of every atomic event on those r.v.s(i.e. every sample point)

	Toothache		¬ Toothache	
	catch	¬catch	catch	≠ catch
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

2.2.6 Conditional probability

Conditional or posterior probabilities:
given some information, sometimes called evidence, the probability of an event happening under the evidence.

$$
\begin{aligned}
& P(X_1,\ldots,X_n) \\
= & P(X_1,\ldots,X_{n-1})P(X_n|X_1,\ldots,X_{n-1} \\
= & P(X_1,\ldots,X_{n-2})P(X_{n-1}|X_1,\ldots,X_{n-2})P(X_n|X_1,\ldots,X_{n-1}) \\
= & \ldots \\
= & \prod_{i=1}^{n} P(X_i|X_1,\ldots,X_{i-1})
\end{aligned}
$$

- If a patient is observed to have toothache and no other information is yet available, then the probability of having cavity is 0.8
- P(cavity|tootache) = 0.8

$$Tootache = true \Rightarrow P(cavity|tootache) = 0.8 \tag{2.2}$$

If we know more facts e.g. cavity is also given, then we have:
$P(cavity|tootache, cavity) = 1$ The less specific belief *remains valid* after more evidence arrives, but is not always *useful*. New evidence may be irrelevant, allowing simplification, e.g.:

$$P(cavity|tootache, dice = 1) = P(cavity|tootache) = 0.8 \tag{2.3}$$

This kind of inference, sanctioned by domain knowledge, is crucial.

Theorem 2.2.1 — Definition of conditional probability.

$$P(a|b) = \frac{P(a \cap b)}{P(b)} if P(b) \neq 0 \tag{2.4}$$

When the example used so far is applied to this theorem we will get this:

$$P(cavity|tootache) = \frac{P(cavity, foothache}{P(tootache)} \tag{2.5}$$

If the product rule is applied, it gives an alternative formulation:

$$P(a \cap b) = P(a|b)P(b) = P(b|a)P(a) \tag{2.6}$$

A general version holds for whole distributions, e.g.:

$$P(weather, cavity) = P(weather|cavity)P(cavity) \tag{2.7}$$

2.2.7 Chain rule

Chain rule is derived by successive application of product rule:

2.2.8 Probability for continuous variables

Where X is a real r.V., express distribution as a parametrized function of value, as shown in the next theorem:

$$\begin{aligned} D(x=x) &= U[18.26](x) \\ &= \text{uniform density between [19, 26]} \end{aligned}$$

Theorem 2.2.2 — Probability density function D(x).

$$P(a < X < b) = \int_a^b D(x)dx \tag{2.8}$$

Let X be a random variable a represent a student's age x in the university. Suppose that:

D(X = 20.5) = 0.125 really means:

$$lim_{dx \to 0} P(20.5 \leq X \leq 20.5 + dx) = 0.125 \tag{2.9}$$

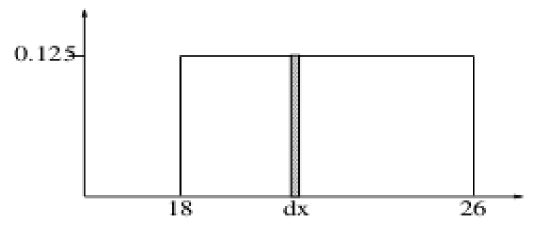


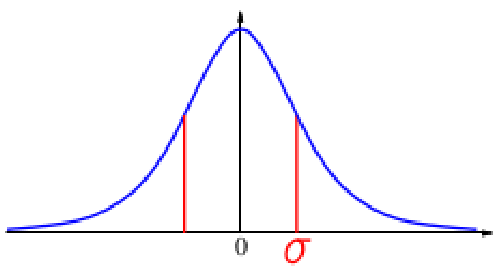
Figure 2.1: Probability density function for equation: 2.9

2.2.9 Gaussian density

Theorem 2.2.3 — Gaussian density function. Is a widely used density function

$$D(X=x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-(x-\mu)^2/2\sigma^2} \tag{2.10}$$

When $\mu = 0$ the functions looks like this:



Probability in $X \in (a,b)$ is calculated by using the *probability density function* shown in equation: 2.9.

2.3 Proposition

A proposition can be seen as an event(a set of sample points), where the following proposition is true:
A is "dice roll is less than 4"

$$P(A) = \sum_{\omega \in A} P(\omega) \tag{2.11}$$

Given Boolean random variables A and B:
- event a = the set of sample points where $A(\omega)$ = true
- event $\neg a$ = the set of sample points where $A(\omega)$ = false
- event $a \cap b$ = points where $A(\omega)$ and $B(\omega)$ = true

Often in AI applications, the sample points are defined by the values of a set of random variables, i.e., the sample space is the Cartesian product of the ranges of the variables.
With Boolean variables, a sample points = propositional logic model.

- e.g. A = true, B = false, or a∩¬b

Proposition = disjunction of atomic events in which it is true.

$$\begin{aligned} (a \cup b) &= (\neg a \cap b) \cup (a \cap \neg b) \cup (a \cap b) \\ \Rightarrow P(a \cup b) &= P(\neg a \cap b) + P(a \cap \neg b) + P(a \cap b) \end{aligned}$$

2.3.1 Inclusion - exclusion Principle

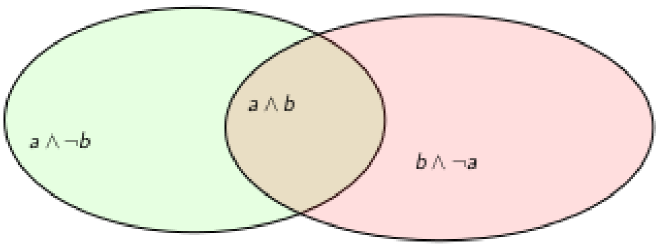


Figure 2.2: Inclusion-Exclusion Principle

The definitions imply that certain logically related events must have related probabilities(see figure 2.3.1).

$$P(a \cup b) = P(a) + P(b) - P(a \cap b) \tag{2.12}$$

2.3.2 Syntax for propositions

Propositional or Boolean random variables:

- e.g. Cavity(do I have a cavity?)
- Cavity = *true* is a proposition, also written cavity

Discrete random variables(finite or infinite):

- e.g., Weather is one of <*sunny, rain, cloudy, snow*>
- Weather = *rain* is a proposition
- Values must be exhaustive and mutually exclusive

Continuous random variables(bounded or unbounded):

- e.g., Temp = 21.6
- also allow, e.g. Temp < 22.0

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

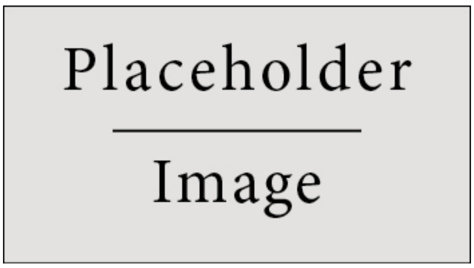


Figure 3.1: Figure caption

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)

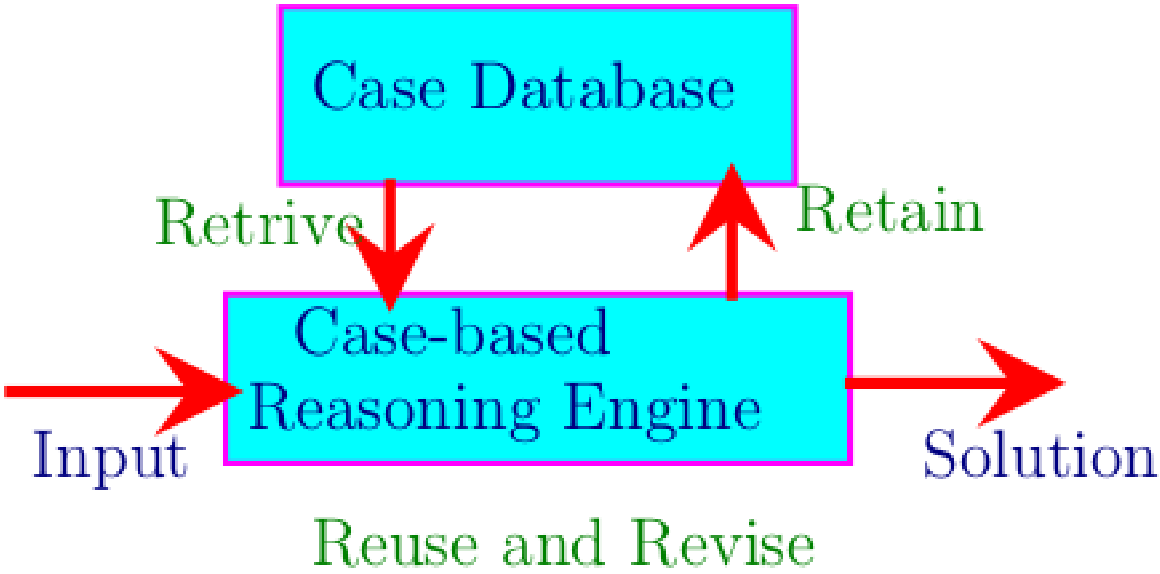


Figure 4.1: The Case-Based Reasoning System

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

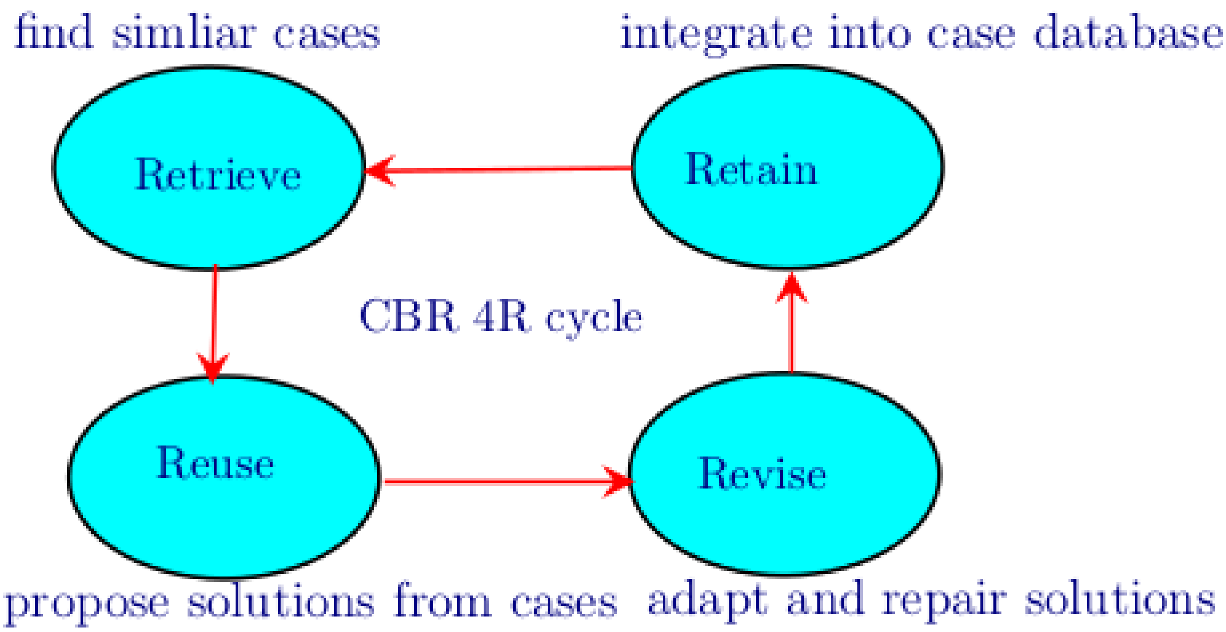


Figure 4.2: The Case-Based 4R cycle

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