AI #7

Outline

Outline

The topic for today is about the constraint satisfaction problems and their solutions. The famous SAT (Boolean satisfiability) problem belongs to this family, and so do many puzzles, e.g. from collections of Simon Tatham.

At first we see some typical problems. Next we examine the most traditional solving method of CSPs: the backtracking. After we deal with the structure of the problem, which could speed up the solving of the problem. Finally we present how we can solve problems with local search, and how effective this method is.

AI #7

Constraint satisfaction problems (CSPs)

Constraint satisfaction problems (CSPs)

- **Standard search problem:** *state* is a "black box"—any old data structure that supports goal test, eval, successor
- **CSP:**
  - *state* is defined by *variables* X_i with *values* from *domain* D_i
  - *goal test* is a set of *constraints* specifying allowable combinations of values for subsets of variables

Simple example of a *formal representation language*
Allows useful *general-purpose* algorithms with more power than standard search algorithms

At a usual search problem we are not interested in the structure of the state. The search methods that we are interested in have an initial state, and it is clear which states succeed which. This enables us to write very general functions which can solve almost every search problem, but a general program cannot take into consideration the specialities of specific problems.
At CSP we have several variables which describe the state. The values of these variables are from given domains. We have one or more constraints about the variables and their values. We say that a state is a goal state, if the values of the variables satisfy all the constraints.

We can treat CSP as a very simple language which describes problems formally. The fact that we can access specific parts of states enables us to use the specialities of a problem, and restrict the search on the state-space in order to get the solution earlier.

AI #7

Example: Map-Coloring

Example: Map-Coloring

- *Variables*: WA, NT, Q, NSW, V, SA, T
- *Domains*: $D_i = \{red, green, blue\}$
- *Constraints*: adjacent regions must have different colors,
  - e.g., $WA \neq NT$ (if the language allows this), or
  - $(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

One of the most typical example of CSP is the map coloring (`https://en.wikipedia.org/wiki/Graph_coloring`). Here we have countries/regions and we need to colour them in such a way, that the adjacent countries need to be of different colours. By the four colour theorem we can do this for any planar (2D maps), but it is an NP-hard problem to decide, whether a given planar map can be coloured with just three colors. Here we apply the latter problem to Australia, where there are seven regions. The variables are named by their abbreviations. We have the same set of colours for every region, so all the domains are the same. Finally the constraints need to express the rule of map colouring. For this we need to list all the adjacent pairs, and give a constraint so that their values are different. Can you solve it alone?

AI #7

Example: Map-Coloring contd.

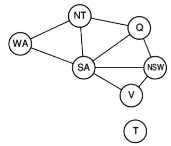
Example: Map-Coloring contd.

- *Solutions* are assignments satisfying all constraints, e.g., $\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

This is the solution: all the regions are coloured, and the adjacent regions are of different colours. We can give this solution not only as a picture, but as a list of assignments.

Constraint graph

- *Binary CSP*: each constraint relates at most two variables
- *Constraint graph*: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

For this map colouring problem all the constraints are binary ones, they contain two variables. In such cases we can draw a graph, where our variables will be the nodes of the graph, and each constraint means one edge of the graph. Now we can take the connected subgraphs, and can solve the subproblems in parallel.

Varieties of CSPs

- Discrete variables
  - finite domains; size $d \implies O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - infinite domains (integers, strings, etc.)
    - e.g., job scheduling, variables are start/end days for each job
    - need a *constraint language*, e.g., $StartJob_1 + 5 \leq StartJob_3$
    - *linear* constraints solvable, *nonlinear* undecidable
- Continuous variables
  - e.g., start/end times for Hubble Telescope observations
  - linear constraints solvable in poly time by LP methods

We have finitely many variables, this is common at CSPs. But the size/type of the domain can vary. For puzzles (like map colouring) we have finite domains only. If the size of the biggest domain is d, then the problem has exponential complexity: $O(d^n)$.

In some cases we have discrete variables, but the domain is infinite. When building a house, the bricklayer must work earlier than the room-painter, and the walls must dry several days before painting. We can uses variables for the start and end dates, and add constraints to the technological limits. This take us to the operational research (`https://en.wikipedia.org/wiki/Operations_research`). If the constraints are linear then we can get a solution, otherwise we cannot solve all the problems, just some specific ones.

Varieties of CSPs

- Discrete variables
  - finite domains; size $d \implies O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - infinite domains (integers, strings, etc.)
    - e.g., job scheduling, variables are start/end days for each job
    - need a *constraint language*, e.g., $StartJob_1 + 5 \leq StartJob_3$
    - *linear* constraints solvable, *nonlinear* undecidable
- Continuous variables
  - e.g., start/end times for Hubble Telescope observations
  - linear constraints solvable in poly time by LP methods

Moreover we have a third option, when our variables are continuous. A famous example for this case is the Hubble Telescope. Initially, it took two weeks for to optimally sort the astronomers' requests for one week. Now a local search solves it almost optimally within 10 minutes. For linear problems/constraints the Simplex method (`https://en.wikipedia.org/wiki/Simplex_algorithm`) is the favoured solving method.

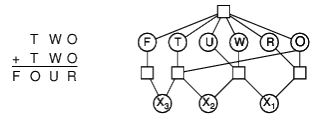
In the following we will work with finite domains.

Varieties of constraints

- *Unary* constraints involve a single variable,
  - e.g., $SA \neq green$
- *Binary* constraints involve pairs of variables,
  - e.g., $SA \neq WA$
- *Higher-order* constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints
- *Preferences* (soft constraints), e.g., *red* is better than *green*
  - often representable by a cost for each variable assignment → constrained optimization problems

We can classify constraints according to different criteria. One such aspect is the number of different variables in one constraint. We can have unary, binary and higher order constraints. The other aspect is whether the constraint need to hold, or can we treat the cases when this constraint does not hold as solution. The former is the hard, the latter is the soft constraint. If the problem contains soft constraints, then usually we add cost to each constraint, and we can calculate the total cost of solutions. This means that we can treat this kind of problems as an optimization problem.

Example: Cryptarithmetic



- *Variables*: $\{F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3\}$
- *Domains*: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- *Constraints*:
  - $ldiff(F, T, U, W, R, O)$
  - $O + O = R + 10 \cdot X_1$, etc.

Maybe you know the SEND+MORE=MONEY problem. This one is a simpler problem. The letters denote numbers, different letter different numbers. What is value of the letters, so that the calculation is correct? We can reformulate it as a diophantine equation $200T + 20W + 2O = 1000F + 100O + 10U + R$, and we interested in its solution. But we can formulate by columns as: $2O = R + 10X_1$, $X_1 + 2W = U + 10X_2$, $X_2 + 2T = O + 10X_3$, $X_3 = F$. Here F, T, U, W, R and O denote numbers, so their domain is $\{0, ..., 9\}$, more precisely we cannot start number with 0, so the domain of T and F is $\{1, ..., 9\}$, and X_is are carries, so the domain here is $\{0, 1\}$. The constraint *different letters denote different numbers* is not described yet. It can be given as pairwise inequalities—as you have seen at the map colouring problem—or using the all-different construction which was given here as `ldiff`.

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floorplanning

Notice that many real-world problems involve real-valued variables

The CSP problems are not artificial ones, we meet them in real life. For example, constructing a timetable for this faculty is a such a problem, at first the departments need to decide who teaches which class (each teachers is knowledgeable in some specific topics, and a teacher has a limited time frame), and if we have all these constraints, the administration needs to give the lectures a place (room) and time. Of course at the same time we cannot have two classes in the same room, and a teacher/student cannot have 2 classes at the same time.

Here you can see other problems. Of course, with real-world problems we usually need to use real-valued variables, which make the problem harder.

Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it
States are defined by the values assigned so far

- *Initial state*: the empty assignment, $\emptyset$
- *Successor function*: assign a value to an unassigned variable that does not conflict with current assignment. $\Longrightarrow$ fail if no legal assignments (not fixable!)
- *Goal test*: the current assignment is complete

1. This is the same for all CSPs!
2. Every solution appears at depth n with n variables $\Longrightarrow$ use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!!

We have variables (without value) and we want to give value of them to satisfy the constraints. We do this step-by-step, where in each step we assign a value to one variable. This looks like a search problem. For this we need a state space, which is the set of partial assignments of variables. At the starting state there are no assigned variables and the successor function add to a value to some variable if it does not conflict with any constraints. If all variables are assigned (and hence all the constraints are fulfilled), we have a solution, hence it is the goal state.

We can use this method for any CSP (with finite domain). From the construction it is obvious, that all the solutions are at depth level n, where we have n variables. In this case $d = m$ (the deepest level is n), so the depth-limited search is not helpful, so the best choice here is the depth first search.

Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it
States are defined by the values assigned so far

- *Initial state*: the empty assignment, $\emptyset$
- *Successor function*: assign a value to an unassigned variable that does not conflict with current assignment. $\implies$ fail if no legal assignments (not fixable!)
- *Goal test*: the current assignment is complete

1. This is the same for all CSPs!
2. Every solution appears at depth n with n variables $\implies$ use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!!

What is the branching factor at level l? We have $n-l$ unassigned variables, and if the size of the biggest domain is d, then we have $(n-l)d$ possibilities. If we multiply these values for $l = 0, \ldots, n$ then we get the $n!d^n$ which is a huge number even for simple problems. In the case of the cryptarithmetic problem before (TWO+TWO=FOUR) gives 3.63×10^{16}.

Backtracking search

- Variable assignments are *commutative*,
  - i.e., [$WA = red$ then $NT = green$] same as [$NT = green$ then $WA = red$]
- Only need to consider assignments to a single variable at each node
  - $\Longrightarrow$ $b = d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called *backtracking* search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \approx 25$

Fortunately it does not matter in what order our variables get their values, or using the mathematical terminology the variable assignment is commutative. This means that in one step we do not need to take into account all unassigned variables, just select one of them! This decreases the previous complexity to d^n, so the number below decreases to 10^{10}.

Using DFS with single variable assignment is called *backtracking search*.

Without any improvement we can solve the n-queen problem (place n queens on $n \times n$ table without any two in conflict) for 25 queens.

Backtracking search

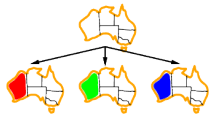
```
function Backtracking-Search(csp): solution/failure
  return Recursive-Backtracking({ }, csp)

function Recursive-Backtracking(assignment,csp): soln/failu
  if assignment is complete then return assignment
  var := Select-Unassigned-Variable(Variables[csp],
            assignment, csp)
  for each value in Order-Domain-Values(var,
            assignment, csp) do
        if value is consistent with assignment
                given Constraints[csp] then
            add {var = value} to assignment
            result := Recursive-Backtracking(assignment,
                        csp)
            if result != failure then return result
            remove {var = value} from assignment
  return failure
```

Let us see its implementation! We need to call a recursive function, where the value of the first argument (the assignments) is empty at first.

If the assignment is complete, we are ready. Otherwise we need to select somehow an unassigned variable, and based on a given order we need to try all the elements of the domain. If this value is consistent (conforms to the constraints), we keep it, we add this assignment to the other assignments, and the function call itself (recursion). If this was successful, we return the solution. Otherwise we reverse the last assignment, and try the next value for the actual variable. If we cannot assign any value to this variable, we need so send back failure to the caller function.

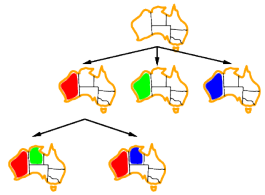
Let us see the map of Australia! At first there are no coloured regions.

Backtracking example



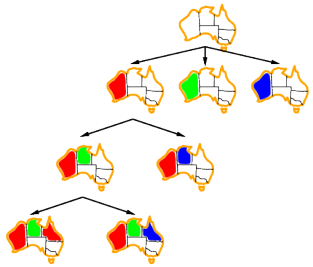
If we first select Western Australia, we can use any of the colours to colour it.

Backtracking example



Let us take the first case, when this region is red. If we next select the Northern Territory, we can only use two colours, the red cannot be used here, as two neighbouring regions cannot be of same colour.

Backtracking example



If we use green for Northern Territory and next select Queensland, the are two ways to colour it, etc.

General-purpose methods can give huge gains in speed:

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?
- Can we take advantage of problem structure?

You may remember: we need to select an unassigned variable, and give an ordering to the values. How can we do this in a clever way?
If in the last line on the previous slide Queensland is coloured blue, we cannot finish the colouring as we cannot use ony of the colours for South Australia. How can we detect such cases when we have hundreds or thousands of variables?

We can use heuristics. There are different heuristics than the ones we used at best-first kind searches. There we evaluated states, based on the distance from a goal state.

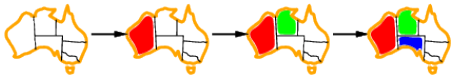
AI #7

Minimum remaining values

Minimum remaining values

Minimum remaining values (MRV):

- choose the variable with the fewest legal values



At MRV heuristics we sort variables based on values suited to the constraints. Let us take the case, when Western Australia is red and Northern Territory is green. For South Australia we can only use the blue, i.e. 1 option. Queensland can be coloured red and blue, i.e. 2 options. New South Wales, Victoria and Tasmania can be coloured at this state in all three ways. So South Australia has the minimal legal values, so this heuristic gives back this variable. This means at this step we will colour it blue.

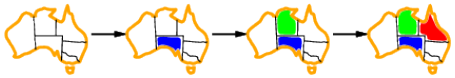
If there is a variable which has no legal values, this heuristic finds it, and hence we know that we need to go back. If some variable has only one legal value, we will use it immediately. So this heuristic help us to find dead ends as soon as possible, to avoid discovering hopeless states.

Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:

- choose the variable with the most constraints on remaining variables



At the construction of the faculty timetable the biggest lectures are fixed first. A small seminar can be moved easily in case of a conflict, but a lecture with many practices/labs has many dependencies to consider. In the case of CSP this observation, experience gives the degree heuristic. We can use it, if the MRV does not select a unique variable.

AI #7

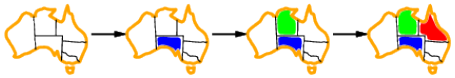
Degree heuristic

Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:

- choose the variable with the most constraints on remaining variables

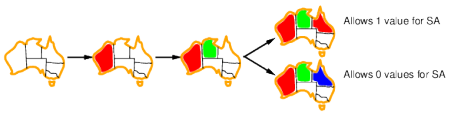


At the coloring problem in the starting state all the regions can have all three colours. South Australia has 5 adjacent uncoloured region, Western Australia has 2, etc. The winner here is South Australia with 5, so we need to colour this region first. The other regions of mainland have two colour options left (red and green), because all are adjacent with South Australia. So the MRV heuristic cannot decide. Victoria and Western Australia have one non-coloured neighbours each, so based on degree heuristic we need to choose between Northern Territory, Queensland and New South Wales. In this case the Northern Territory was selected and coloured to green. The MRV heuristic could select between Western Australia and Queensland, because here we can only use one colour (red). And we can continue this process using these two heuristics.

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

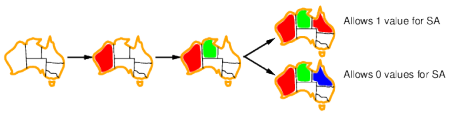
You may remember, if you selected a variable, we need to try all the possibilities (values), but we need to determine a suitable order. At this heuristic we want to lift a minimal barrier for the following steps, to allow for the longest sequence of assignments. At the picture before the branching we have coloured Western Australia and Northern Territory. Let us assume, that the next selected region is Queensland. We have two options as the Northern Territory is green: red and blue.

By coloring Queensland we restrict the possibilities of South Australia and New South Wales. If we colour it to red, the only possibility of South Australia (blue) remains, and New South Wales lost the colour red.

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

If we colour Queensland to blue, South Australia has no remaining colours, and New South Wales lost the colour blue.

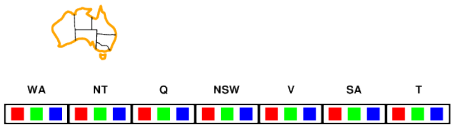
Summing up: if we sum the lost colours, we get 1 for each case, but the colour blue prevent us from solve the problem. With red we can get a solution.

These heuristics are independent, you can use any combination of them. aima-python contains a program, where you can play with these combinations. If you apply all of them the n-queens problem can be solved for 1000 queens.

`https://github.com/aimacode/aima-python/blob/master/csp.ipynb`

Forward checking

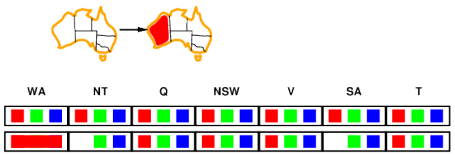
- *Idea*: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



If we assign a value to a variable, then we can take into the account the direct consequences of this step. So let us store the remaining possibilities! At first when there are no assigned variables, all regions have three options.

Forward checking

- *Idea*: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



If Western Australia is coloured red, then its neighbours cannot be red, so we need to delete this option from Northern Territory and South Australia.

AI #7

Forward checking

Forward checking

- *Idea*: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values

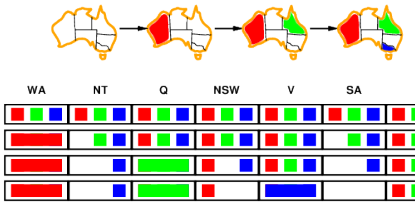
If the second region is Queensland and it is coloured green, then the adjacent regions (Northern Territory, South Australia and New South Wales) cannot be green, so we need to delete this option from them.

AI #7

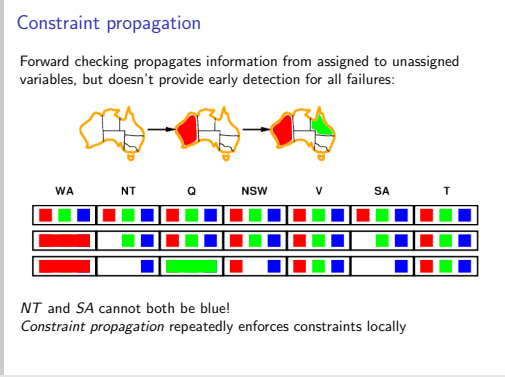
Forward checking

Forward checking

- *Idea*: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



If the third region Victoria is coloured blue, then the adjacent regions (South Australia and New South Wales) cannot be blue, so we delete these options. Therefore there is no options left for South Australia, so we need to go back.

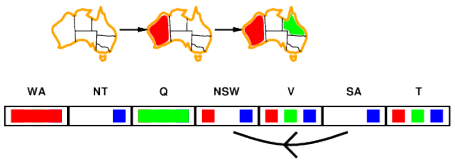


The forward checking only uses the direct consequences (only considers one step). We can construct all the logical consequences of one step. After the second step of the previous example we could realize that two adjacent regions (Northern Territory and South Australia) can both only be blue, which leads to a contradiction.

We construct the consequence is small steps. Our tool for this is the arc consistency.

Arc consistency

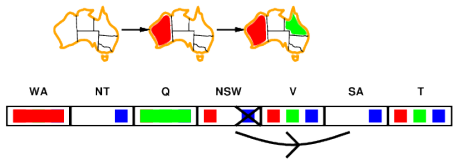
Simplest form of propagation makes each arc *consistent*
$X \to Y$ is consistent iff for **every** value x of X there is **some** allowed y



A non-symmetric arc $X \to Y$ (between variables X and Y) is consistent, if for every value $x \in X$ there is some allowed $y \in Y$. Let denote South Australia by X and New South Wales by Y. X has only one value (blue) and in Y there is a suitable (i.e. different) value, red. So the arc consistency holds in this direction.

Arc consistency

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



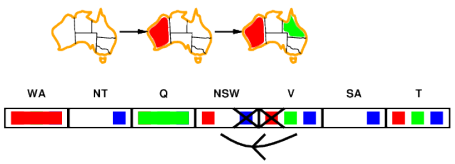
If we take the opposite direction (New South Wales is X and South Australia is Y), then if X is red, the blue in Y is OK. But if X is blue, Y does not contain a suitable value. Hence we cannot colour X (New South Wales) blue, because it cannot give a solution. Therefore we delete this option here.

AI #7

Arc consistency

Arc consistency

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

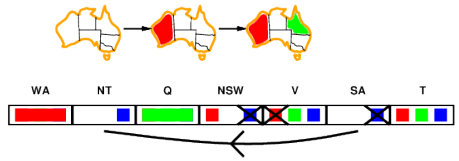


- If X loses a value, neighbors of X need to be rechecked

Changes can spread, so we need check the neighbours. Victoria is adjacent to New South Wales, and as New South Wales can only be red, Victoria needs to be a different colour (we need to delete the red here).

Arc consistency

Simplest form of propagation makes each arc *consistent*
$X \to Y$ is consistent iff 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

Arc consistency could detect the contradiction mentioned before (Northern Territory and South Australia). To preserve the (arc) consistency is not a costly process, so we can apply this after each assignment.

Arc consistency algorithm

```
function AC-3(csp): the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1,X2,...,Xn}
  local var.: queue, a queue of arcs,
                  initially all the arcs in csp

  while queue is not empty do
      (Xi, Xj) := Remove-First(queue)
      if Remove-Inconsistent-Values(Xi, Xj) then
          for each Xk in Neighbors[Xi] do
              add (Xk, Xi) to queue
```

What are the necessary checks we need to do? The literature knows methods from AC-1 to AC-8. The most known/applied is the AC-3. aima-python contains this, its variant and AC-4. This function use a queue (similarly to the BFS), which contains any related pairs of variables (in both direction).

We process this queue, and if it is possible to delete any values from some domain, we need to take the neighbours of the modified domain, and add new arcs to the end of the queue. If the queue becomes empty, we are done.

AI #7

Arc consistency algorithm

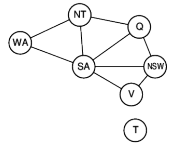
Arc consistency algorithm

```
function Remove-Inconsistent-Values(Xi, Xj):
                          return true iff succeeds
  removed := false
  for each x in Domain[Xi] do
      if no value y in Domain[Xj] allows
               (x, y) to satisfy the constraint Xi <-> Xj
           then delete x from Domain[Xi];  removed := true
  return removed
```

$O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (but detecting *all* is NP-hard)

The `Remove-Inconsistent-Values` function is based on the definition of arc consistency (if some constraint does not hold for x and y, we delete x from its domain. This function contains a double cycle on domains (size d), and one arc can get back into the queue d times. The number of arcs is proportional to n^2. So complexity of the arc consistency is quadratic in size of variables.

Problem structure



- Tasmania and mainland are *independent subproblems*
- Identifiable as *connected components* of constraint graph

Previously we said that if the problem can be broken down into subproblems, then we solve subproblems independently and in parallel. As we describe problems as graphs, the subproblems are subgraphs, more precisely the connected components of the graph of the problem.

Problem structure contd.

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

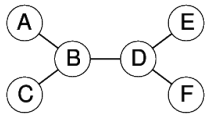
A simple calculation: the original problem has n variables, its subproblems have c, and the biggest domain has d values. The original d^n is reduced to $n/c \times d^c$, which is linear in n. Lets see this with numbers: if we are able to break down the almost unsolvable problem into four similarly sized subproblems, then we can get a solution almost immediately.

AI #7

Tree-structured CSPs

Let see the very special case, when the constraint-graph has no cycles, i.e. it is a tree. In this case the complexity of the solution method is linear in n, while the general method is exponential in n. This special structure can be used in further chapters.

Tree-structured CSPs

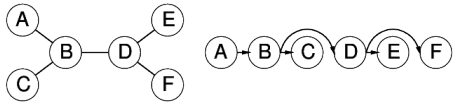


Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

- 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.

Algorithm for tree-structured CSPs

- Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

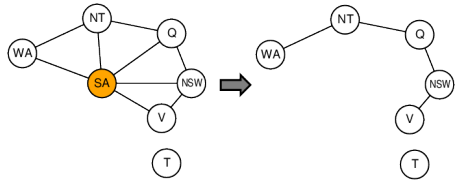


- For j from n down to 2, apply RemoveInconsistent($Parent(X_j), X_j$)
- For j from 1 to n, assign X_j consistently with $Parent(X_j)$

We can start from anywhere, select any variable as the root of the sequence. Use topological ordering (`https://en.wikipedia.org/wiki/Topological_sorting`) for the tree! Next go backwards and delete from the parents domain what is inconsistent with any of its children. Finally assign a value to the variables from the root.

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
Cutset size $c \implies$ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c

What can we do if our graph is not a tree? If it is almost a tree, then there is a solution. *Delete* nodes from the cycles. In case of Australia we have several cycles, and South Australia is part of all of them. If we delete this region, we can get a tree (without branching). How can we delete a node/variable? Assign a value to it, and take the direct consequences (delete the corresponding values from the neighbours domains).
If we needed to delete c variables, we can have c^d assignments of these variables, and for the remaining part the complexity is $(n-c)d^2$. If c is small, this will be small too.

This was a different improvement of the backtracking search. But we have a totally different method to solve CSPs.

Iterative algorithms for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators *reassign* variable values
- Variable selection: randomly select any conflicted variable
- Value selection by *min-conflicts* heuristic:
  - choose value that violates the fewest constraints
  - i.e., hillclimb with $h(n)$ = total number of violated constraints

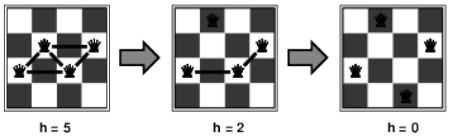
Similarly to search problems, we can use the local search to find the solution. In this case a state will be a **complete** assignment. Here the partial assignments—which were an essential part of the backtracking method—cannot be used.

Any (e.g. a random) assignment will not necessarily be an valid solution, it would probably violate several constraints. But any complete assignment means for us a state, and changing values of some variables give a successor state. We assign heuristic functions based on a number of violated constraints, and we can use the most typical local search methods: hill-climbing, simulated annealing, etc.

The method min-conflicts performs very well for many tasks. Here we have a cycle, which ends if there are no more conflicts. Otherwise in this cycle we need to randomly select a conflicted variable, and choose its best value based on this heuristic.

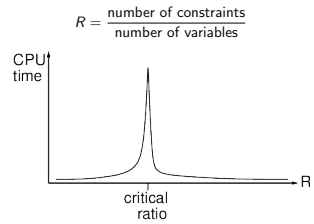
Example: 4-Queens

- *States*: 4 queens in 4 columns ($4^4 = 256$ states)
- *Operators*: move queen in column
- *Goal test*: no attacks
- *Evaluation*: $h(n)$ = number of attacks



We need four variables, the i^{th} variable gives the row of the queen in the i^{th} column (they need to be in different columns and in different rows). So one queen could move up and down only. Our heuristic gives the number of attacks (denoted by lines). At the beginning this is 5. If we move the second queen to the uppermost row, this decreases to 2. If then we move the third queen into the lowermost row, we get a solution.

Performance of min-conflicts
Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 10,000,000$)
The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio



This methods work very well for a large number of queens, and gives a solution in a very short amount of time.

At randomly generated CSPs if we have a small number of constraints (according to the number of variables) we can solve problems easily. If we have many constraints, then the problem becomes unsolvable (over-constrained), which can be detected easily, but not with min-conflicts. We have a ratio, when the solution of the problem is really hard.

AI #7

Summary

Summary

- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by *constraints* on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- The CSP representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice

CSP is a special kind of problem, the problem has an inner structure, which can described with variables, and we have constraints about the values of these variables. The available values of a variable defines a domain for each variable.

The main solving method is backtracking, which is a systematic search (DFS with one variable assignment). We can apply several heuristics to speed up the search. The forward checking considers only one step, while the constraint propagation considers several ones.

We can improve with the analysis of the problem structure: subproblems, (near) tree-like problems. The tree structured CSPs can be solved very fast.

The min-conflicts local search method is also very effective in practice.