Introduction to Artificial Intelligence

Constraint Satisfaction Problems

Sections 3.7 and 4.4, Exercise 6.15, Weld paper on AI planning

Dieter Fox

Outline

◇ CSP examples

◇ General search applied to CSPs

◇ Backtracking

◇ Forward checking

◇ Heuristics for CSPs

◇ Planning as satisfiability

Constraint satisfaction problems (CSPs)

Standard search problem:
  state is a "black box"—any old data structure
    that supports goal test, eval, successor

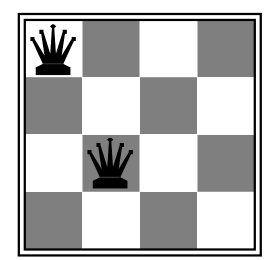
CSP:
  state is defined by *variables* V_i with *values* from *domain* D_i

  goal test is a set of *constraints* specifying
    allowable combinations of values for subsets of variables

Allows useful *general-purpose* algorithms with more power than standard search algorithms

Example: 4-Queens as a CSP

Assume one queen in each column. Which row does each one go in?



$Q_1 = 1 \quad Q_2 = 3$

Variables Q_1, Q_2, Q_3, Q_4

Domains $D_i = \{1, 2, 3, 4\}$

Constraints
  $Q_i \neq Q_j$ (cannot be in same row)
  $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

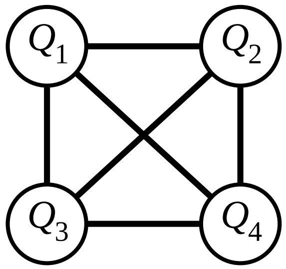
Translate each constraint into set of allowable values for its variables

E.g., values for (Q_1, Q_2) are $(1,3)$ $(1,4)$ $(2,4)$ $(3,1)$ $(4,1)$ $(4,2)$

Constraint graph

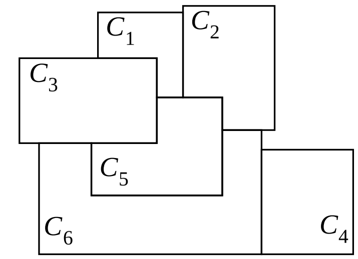
Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



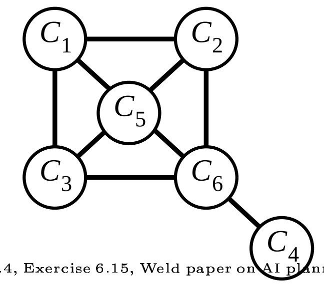
Example: Map coloring

Color a map so that no adjacant countries have the same color



Variables
Countries C_i
Domains
$\{Red, Blue, Green\}$
Constraints
$C_1 \neq C_2$, $C_1 \neq C_5$, etc.

Constraint graph:



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

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

Applying standard search

Let's start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

Initial state: all variables unassigned

Operators: assign a value to an unassigned variable

Goal test: all variables assigned, no constraints violated

Notice that this is the same for all CSPs!

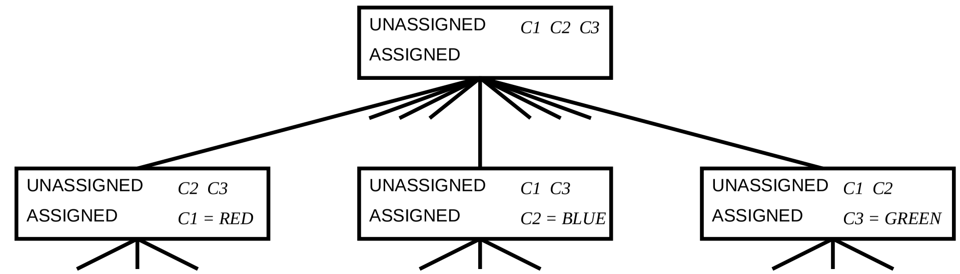
Implementation

CSP state keeps track of which variables have values so far
Each variable has a domain and a current value

datatype CSP-STATE
  components: UNASSIGNED, a list of variables not yet assigned
    ASSIGNED, a list of variables that have values
datatype CSP-VAR
  components: NAME, for i/o purposes
    DOMAIN, a list of possible values
    VALUE, current value (if any)

Constraints can be represented
  explicitly as sets of allowable values, or
  implicitly by a function that tests for satisfaction of the constraint

Standard search applied to map-coloring



Complexity of the dumb approach

Max. depth of space $m =$??

Depth of solution state $d =$??

Search algorithm to use??

Branching factor $b =$??

Backtracking search

Use depth-first search, but
  1) fix the order of assignment, $\Rightarrow b = |D_i|$
    (can be done in the SUCCESSORS function)
  2) check for constraint violations

The constraint violation check can be implemented in two ways:
  1) modify SUCCESSORS to assign only values that
    are allowed, given the values already assigned
  or 2) check constraints are satisfied before expanding a state

Backtracking search is the basic uninformed algorithm for CSPs

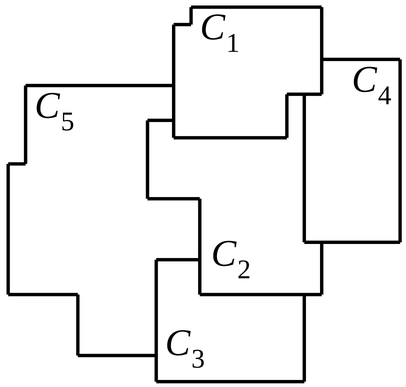
Can solve n-queens for $n \approx 15$

Forward checking

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

Simplified map-coloring example:

	RED	BLUE	GREEN
C_1			
C_2			
C_3			
C_4			
C_5			

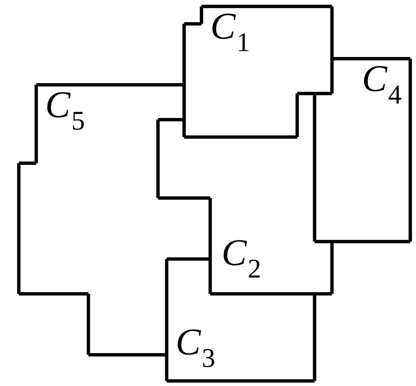


Forward checking

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

Simplified map-coloring example:

	RED	BLUE	GREEN
C_1	$\surd$		
C_2	$\times$		
C_3			
C_4	$\times$		
C_5	$\times$		

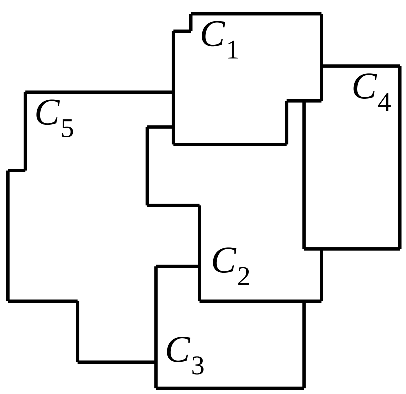


Forward checking

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

Simplified map-coloring example:

	RED	BLUE	GREEN
C_1	$\surd$		
C_2	$\times$	$\surd$	
C_3		$\times$	
C_4	$\times$	$\times$	
C_5	$\times$	$\times$	



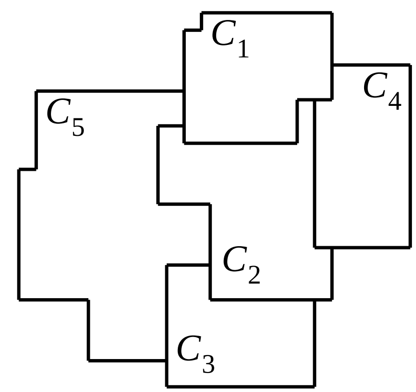
Forward checking

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

Simplified map-coloring example:

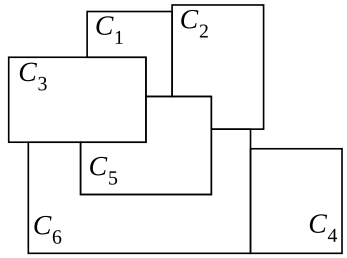
	RED	BLUE	GREEN
C_1	$\surd$		
C_2	$\times$	$\surd$	
C_3		$\times$	$\surd$
C_4	$\times$	$\times$	
C_5	$\times$	$\times$	$\times$

Can solve n-queens up to $n \approx 30$



Heuristics for CSPs

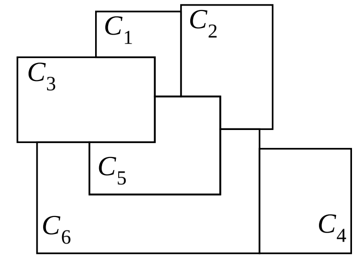
More intelligent decisions on
  which value to choose for each variable
  which variable to assign next



Given $C_1 = Red$, $C_2 = Green$, **choose** $C_3 =$??
  .
Given $C_1 = Red$, $C_2 = Green$, **what next**??
  .

Heuristics for CSPs

More intelligent decisions on
  which value to choose for each variable
  which variable to assign next



Given $C_1 = Red$, $C_2 = Green$, **choose** $C_3 =$??
  $C_3 = Green$: *least-constraining-value*
Given $C_1 = Red$, $C_2 = Green$, **what next**??
  C_5: *most-constrained-variable*

Can solve n-queens for $n \approx 1000$

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

min-conflicts heuristic:
  choose value that violates the fewest constraints
  i.e., hillclimb with $h(n)$ = total number of violated constraints

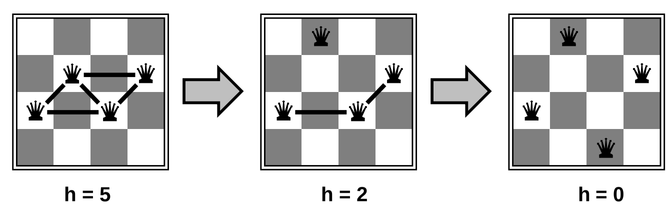
Example: 4-Queens

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

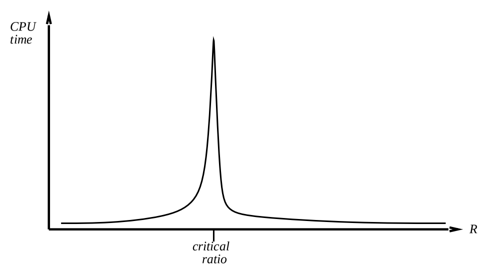


Performance of min-conflicts

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

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Propositional Satisfiability

Determine whether a sentence in **CNF** (conjunctive normal form) is satisfiable

E.g. $(P \vee Q \vee \neg S) \wedge (\neg P \vee R \vee T) \wedge (\neg R \vee T)$

```
function Gsat(sentence, max-restarts, max-climbs) returns a truth assignment or failu
  for i ← 1 to max-restarts do
    A ← A randomly generated truth assignment
    for j ← 1 to max-climbs do
      if A satisfies sentence then return A
      A ← a random choice of one of the best successors of A
    end
  end
  return failure
```

WalkSat: Add randomness

SatPlan: Planning as Satisfiability

```
function SatPlan(initial state, goal, actions, max-length) returns plan or failure

  for i ← 1 to max-length do
    Compile the planning problem (initial state, goal, actions) into CNF
    Try to solve CNF (e.g. using Gsat, WalkSat)
    if satisfying assignment is found then decode and return plan
  end
  return failure
```


SatPlan contd.

Compilation:

$(At(object, loca, i) \wedge (loca \neq locb)) \Rightarrow \neg At(object, locb, i)$

$goal = At(Book, University, n)$

$\forall i \;\; At(A, B, i)$

is instantiated to:

SatPlan contd.

Decoding, plan construction (if satisfying assignment is found):

Check successor states and find actions responsible for transitions

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

Forward checking prevents assignments that guarantee later failure

Variable ordering and value selection heuristics help significantly

CSPs for planning as satisfiability are often more efficient than special purpose planners

BlackBox [Kautz] combines SatPlan with GraphPlan

Based on AIMA Slides ©S. Russell and P. Norvig, 1998 Sections 3.7 and 4.4, Exercise 6.15, Weld paper on AI planning