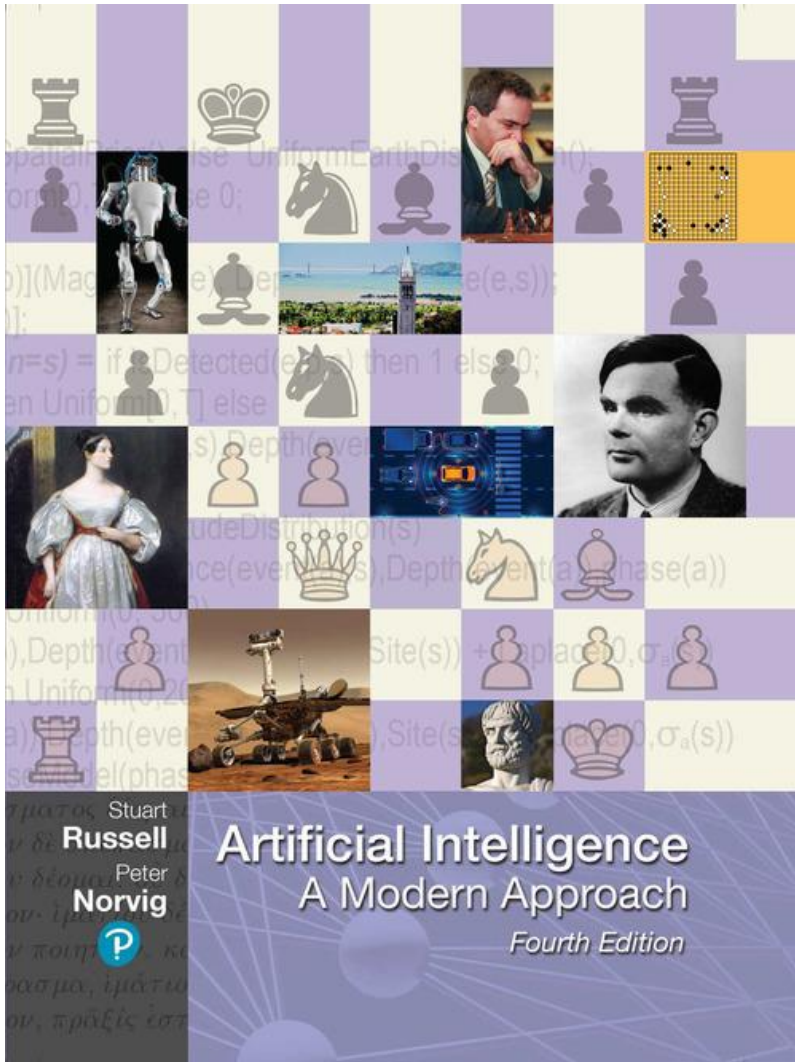


Artificial Intelligence Fundamentals

2023-2024



“Some people call this artificial intelligence, but the reality is this technology will enhance us. So instead of artificial intelligence, I think we'll augment our intelligence.”

—Ginni Rometty

AIMA Chapter 6

Constraint Satisfaction Problems

Outline

- ◆ Defining *Constraint Satisfaction Problems* (CSP)
- ◆ CSP examples
- ◆ Backtracking search for CSPs
- ◆ Local search for CSPs
- ◆ Problem structure and problem decomposition

Defining Constraint Satisfaction Problems

Problems can be solved by searching the state space: a graph where the nodes are states and the edges between them are actions.

We saw that **domain-specific heuristics** could estimate the cost of reaching the goal from a given state, but that from the point of view of the search algorithm, each state is **atomic**, or indivisible— a black box with no internal structure.

In this lecture **we break open the black box** by using a factored representation for each state: a set of variables, each of which has a value.

A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or **CSP**.

*CSP search algorithms take advantage of the structure of states and use **general rather than domain-specific heuristics** to enable the solution of complex problems.*

Defining Constraint Satisfaction Problems

A **constraint satisfaction problem (CSP)** consists of three components, X , D , and C :

- X is a set of variables, $\{X_1, \dots, X_n\}$.
- D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable
- C is a set of constraints that specify allowable combination of values (-> *domain knowledge at play here!*)

CSPs deal with assignments of values to variables.

- A **complete assignment** is one in which every variable is assigned a value, and a solution to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that leaves some variables unassigned.
- **Partial solution** is a partial assignment that is consistent

Constraint satisfaction problems (CSPs)

Standard search problem:

state is a “black box”—any old data structure that supports goal test, eval, successor

CSP:

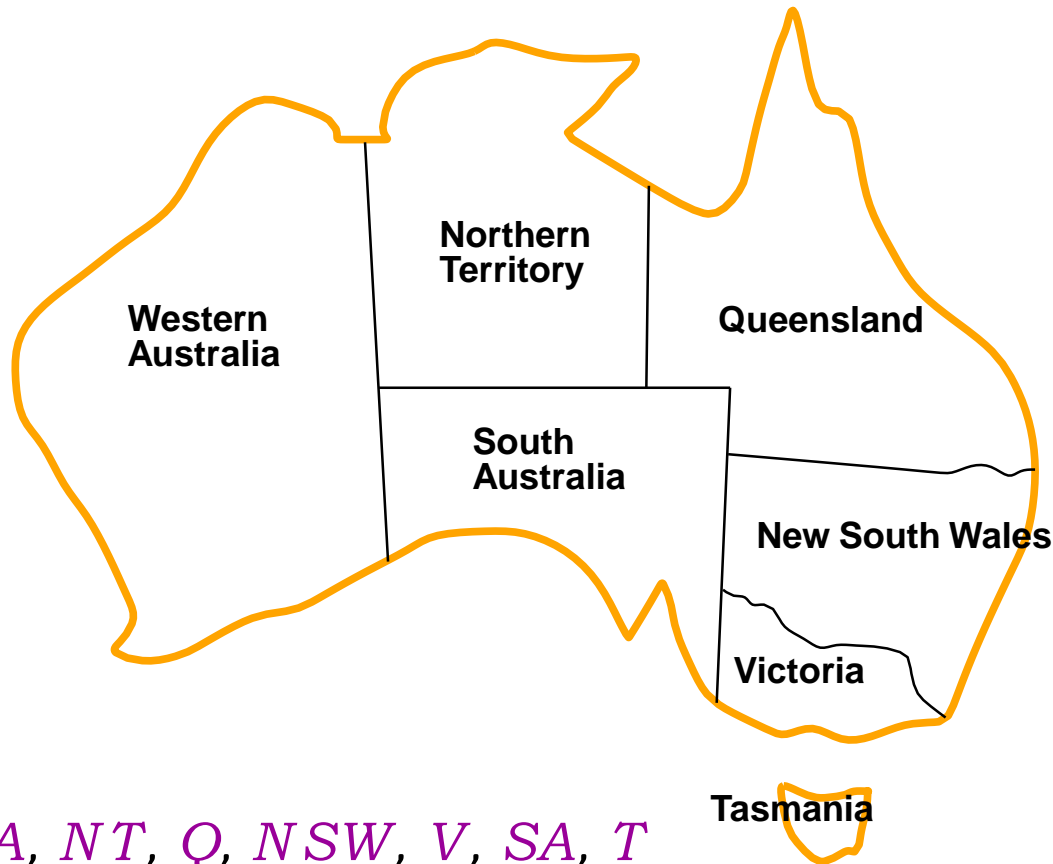
state is defined by **variables** X_i with **values** from **domain** D_i

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

Example: Map-Coloring



Variables WA, NT, Q, NSW, V, SA, T

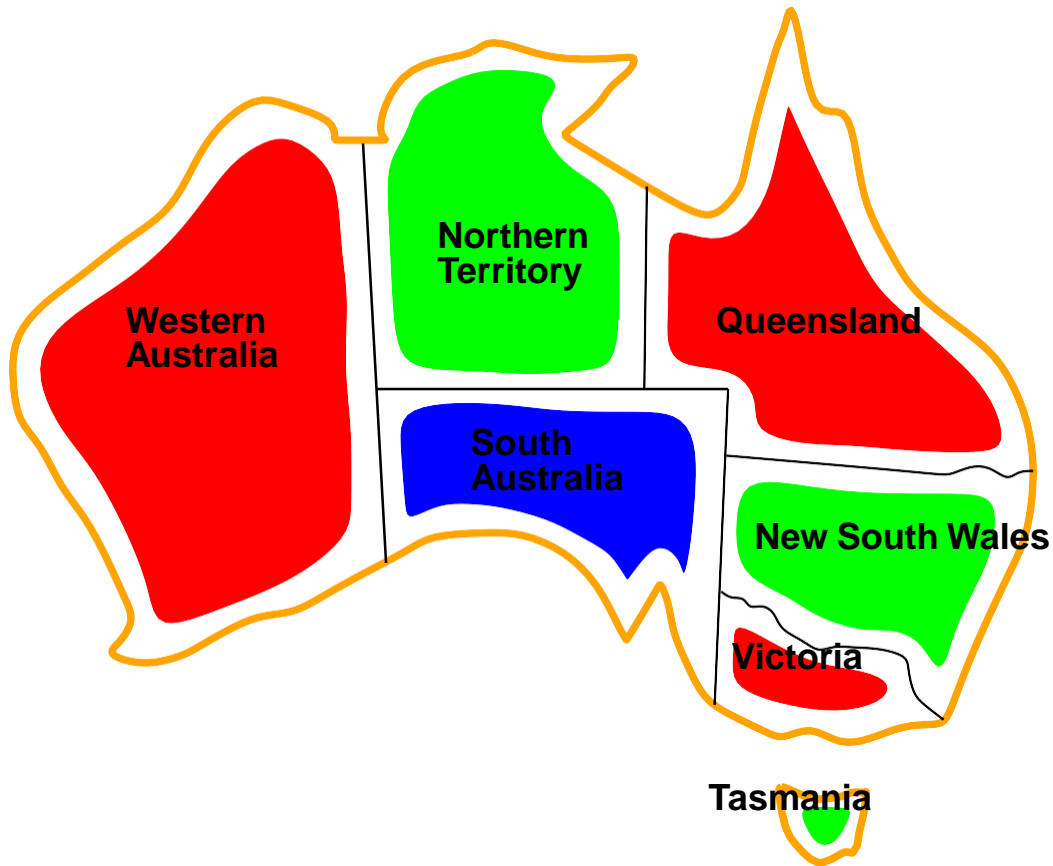
Domains $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

Example: Map-Coloring contd.



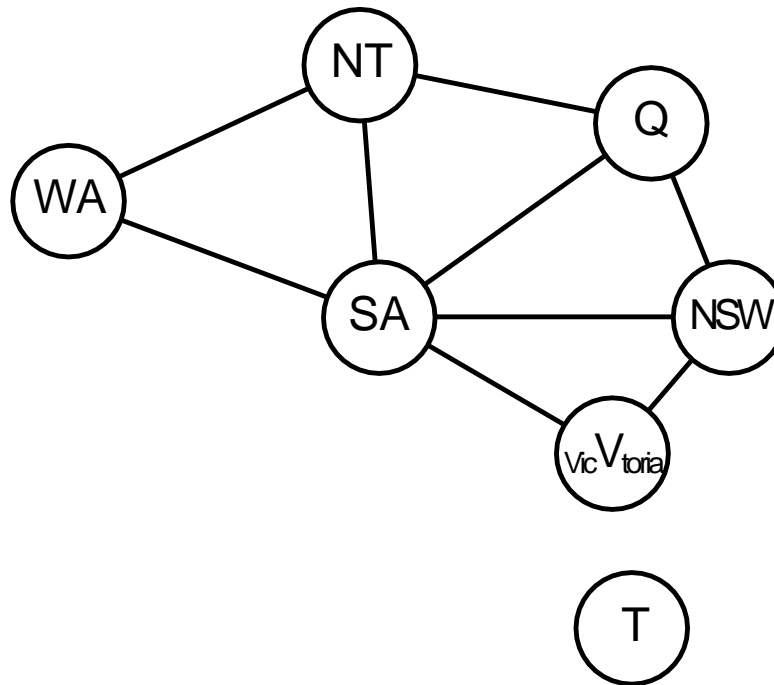
Solutions are assignments satisfying all constraints, e.g.,

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

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!

Varieties of CSPs

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ 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 \rightarrow solvable, **nonlinear** \rightarrow undecidable

Continuous variables

- ◆ e.g., start/end times for Hubble Telescope observations
- ◆ linear constraints solvable in poly time by Linear Programming (LP) methods

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., cryptarithmic column constraints

Preferences (soft constraints), e.g., *red* is better than *green*

often representable by a cost for each variable assignment

→ **constrained optimization problems**

Example: Cryptarithmic

Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeros are allowed.

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

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

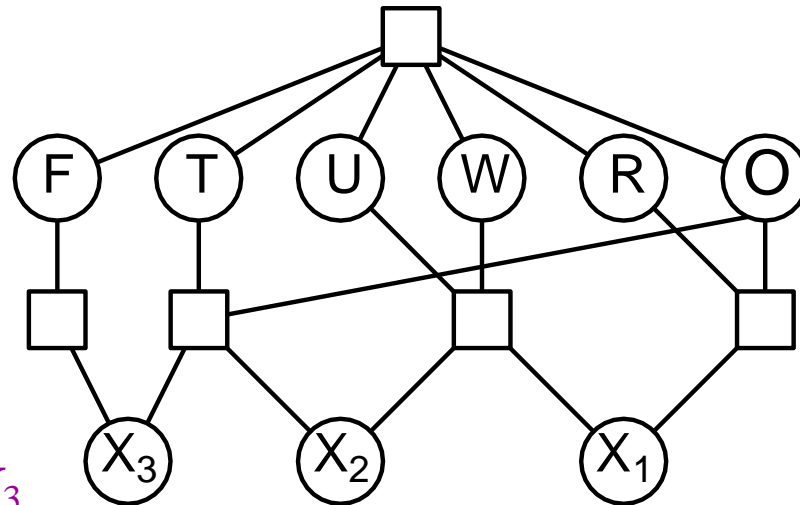
$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1,$

$X_1 + W + W = U + 10 \cdot X_2$

$X_2 + T + T = O + 10 \cdot X_3$

$X_3 = F$



The **constraint hypergraph** for the cryptarithmic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables X_1 , X_2 , and X_3 represent the carry digits for the three columns from right to left.

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**

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, $\{ \}$
 - ◆ **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment.
⇒ 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
⇒ use depth-first search
 - 3) Path is irrelevant, so can also use *complete-state formulation*
 - 4) $b = (n - 1)d$ at depth n , hence $n! d^n$ leaves!!!! 😞

Backtracking search

Variable assignments are **commutative**, i.e.,

$[WA = \text{red} \text{ then } NT = \text{green}]$ same as $[NT = \text{green} \text{ then } WA = \text{red}]$

Only need to consider assignments to a **single variable** at each node

$\Rightarrow b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments is called **backtracking** search ->

It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to extend each one into a solution via a recursive call. If the call succeeds, the solution is returned, and if it fails, the assignment is restored to the previous state, and we try the next value. If no value works then we return failure.

Backtracking search is the basic uninformed algorithm for CSPs

- Can solve n -queens for $n \approx 25$

Backtracking search

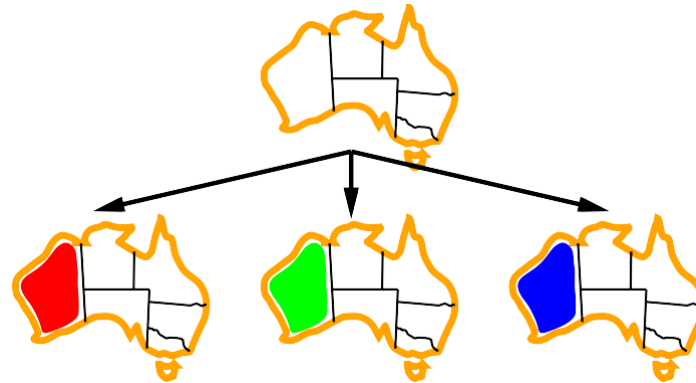
```
function Backtracking-Search(csp) returns solution/failure
  return Recursive-Backtracking({ }, csp)

function Recursive-Backtracking(assignment, csp) returns soln/failure
  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
```

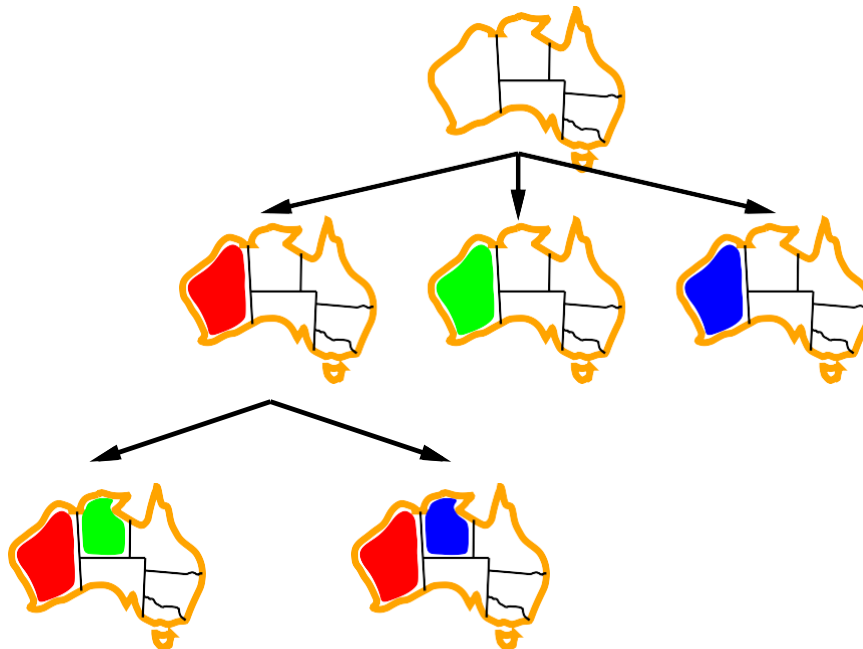
Backtracking example



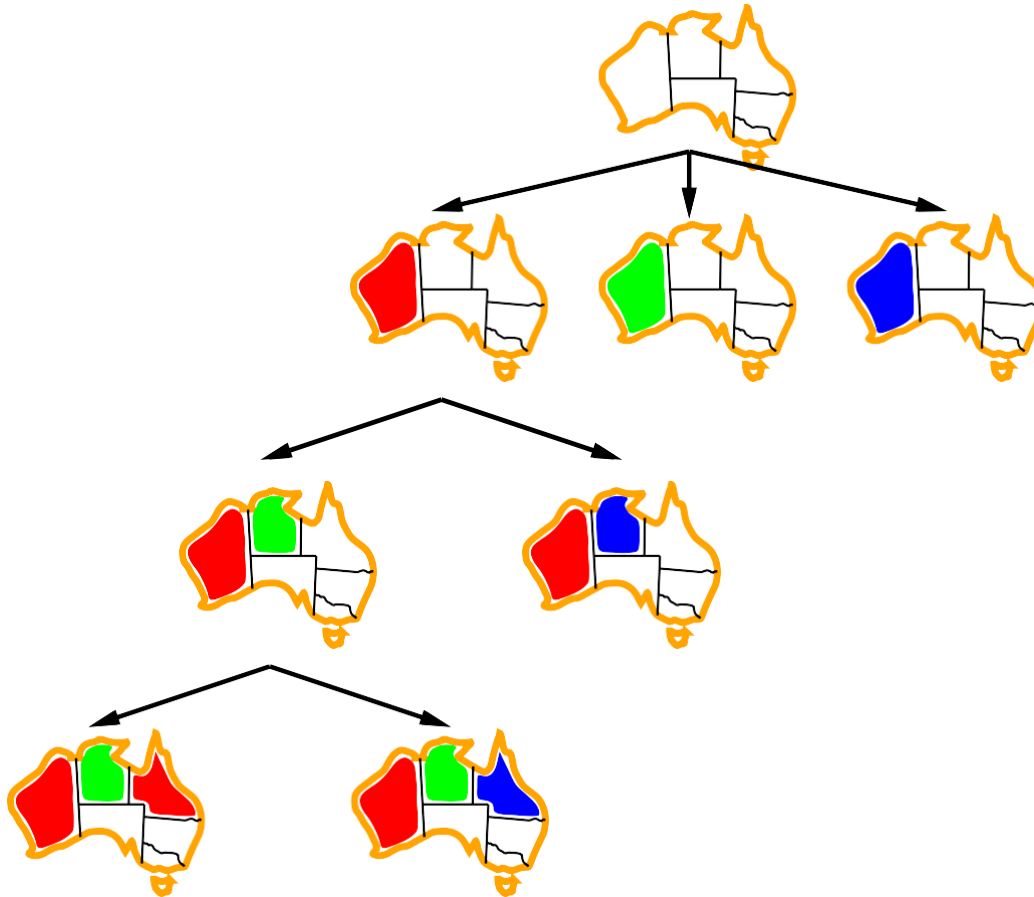
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

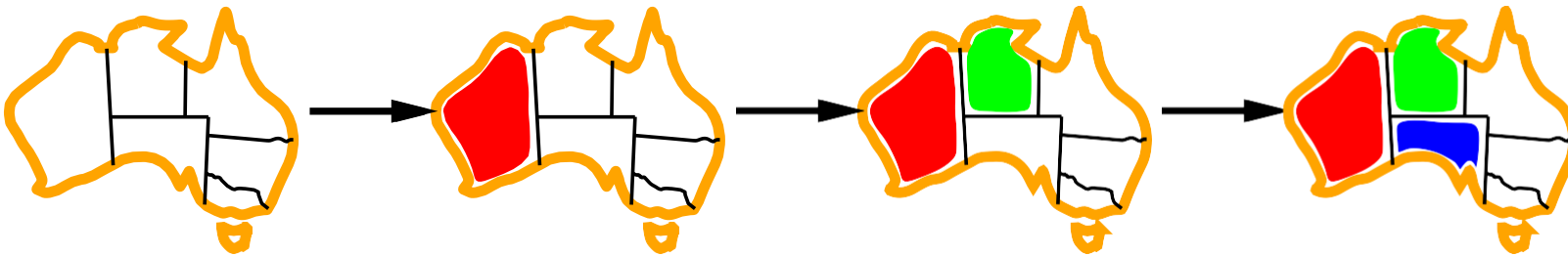
General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its 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**



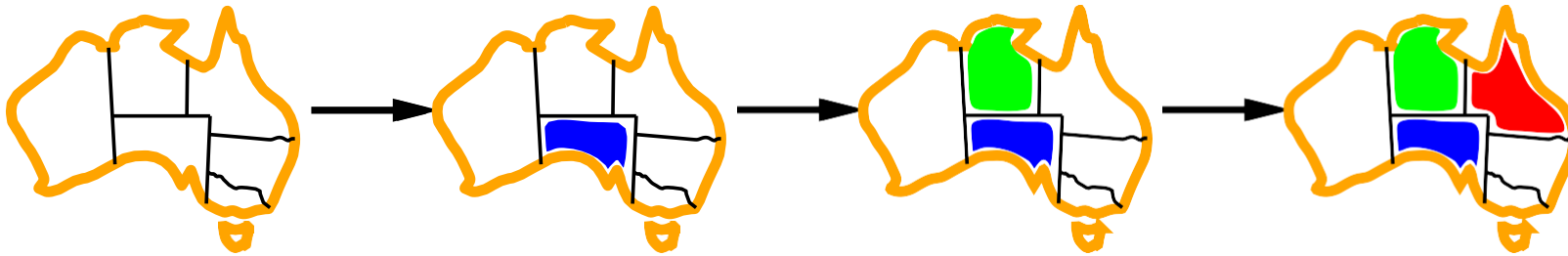
It also has been called the “**most constrained variable**” or “**fail-first**” heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.

Degree heuristic

Tie-breaker among MRV variables

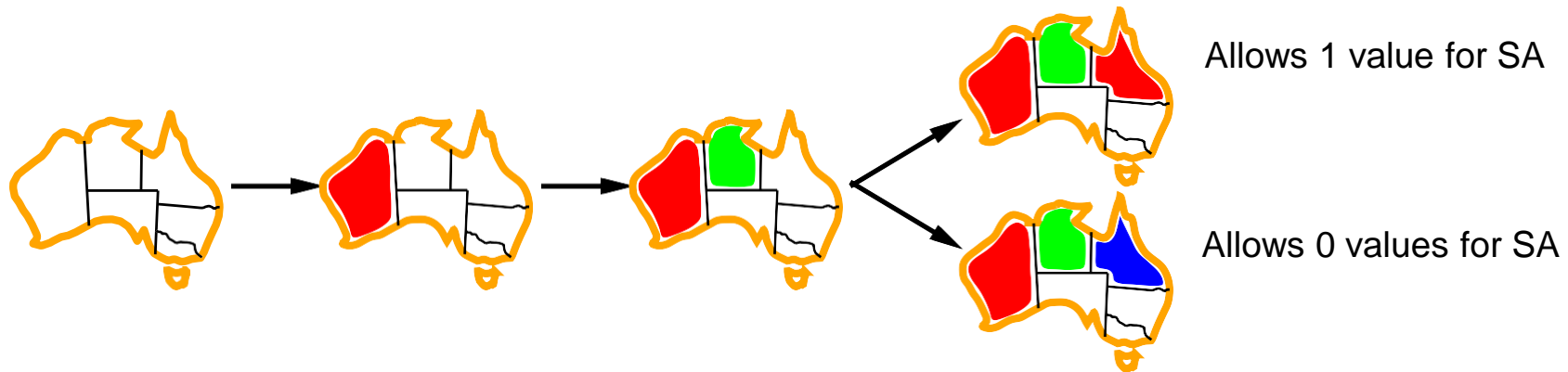
Degree heuristic:

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.

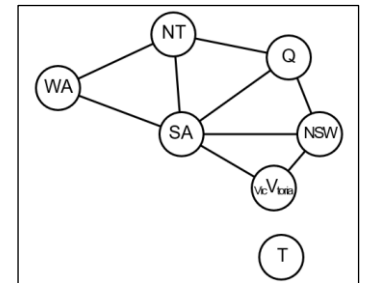
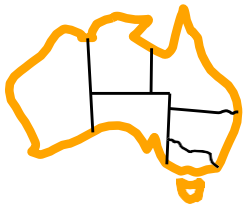
Why should variable selection be fail-first, but value selection be fail-last?

Every variable has to be assigned eventually -> **by choosing the ones that are likely to fail first, we will on average have fewer successful assignments to backtrack over.**

For value ordering, the trick is that we only need one solution -> **it makes sense to look for the most likely values first.**

Forward checking

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



WA

NT

Q

NSW

V

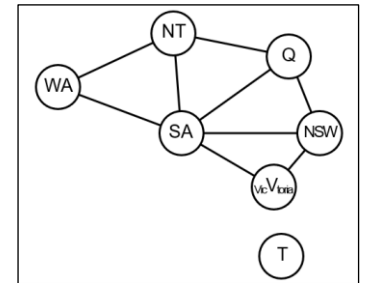
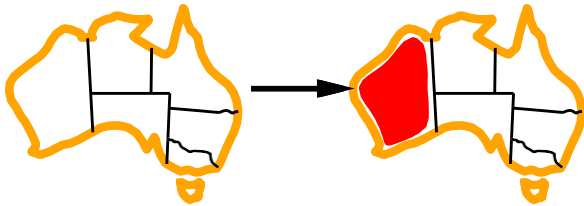
SA

T



Forward checking

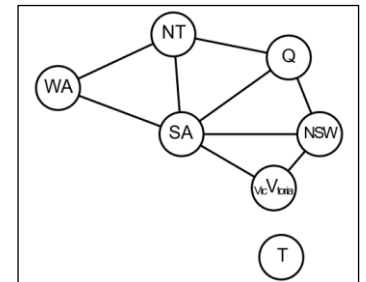
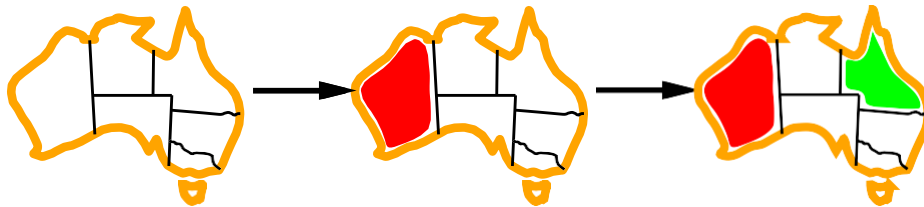
Idea: Keep track of remaining legal values for unassigned variables
 Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div></div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div></div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>

Forward checking

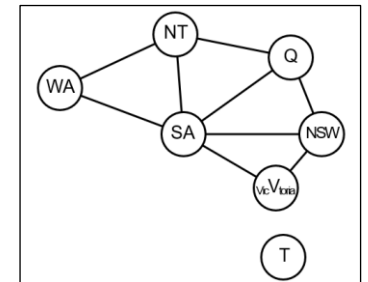
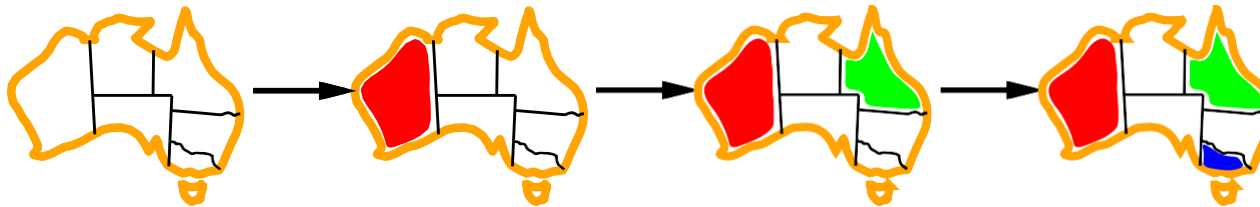
Idea: Keep track of remaining legal values for unassigned variables
 Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Forward checking

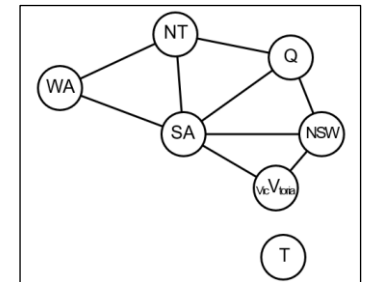
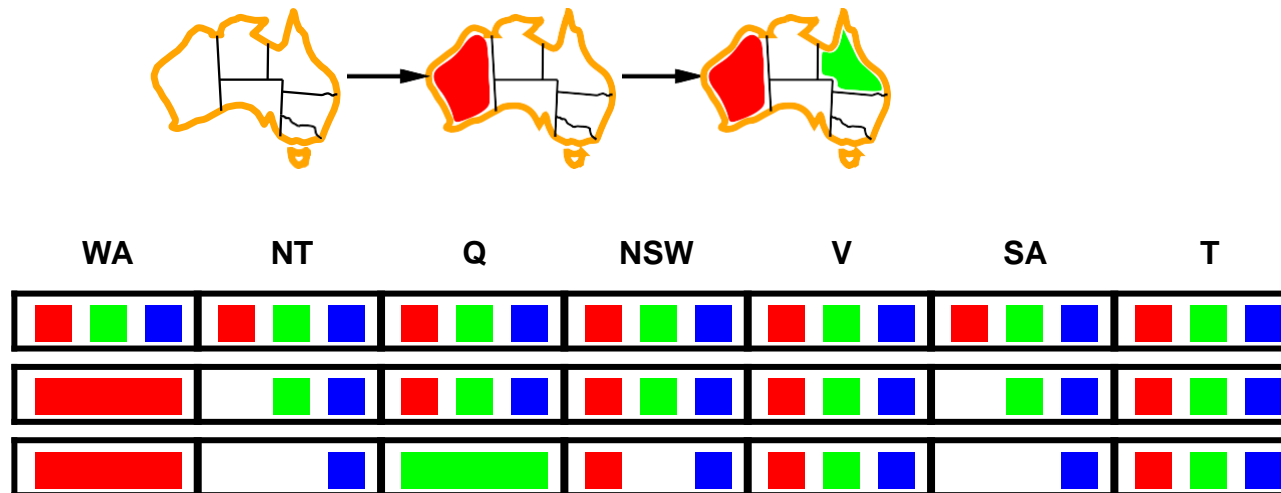
Idea: Keep track of remaining legal values for unassigned variables
 Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Constraint propagation

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



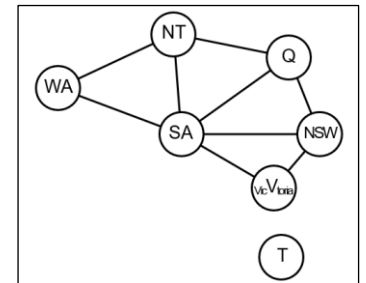
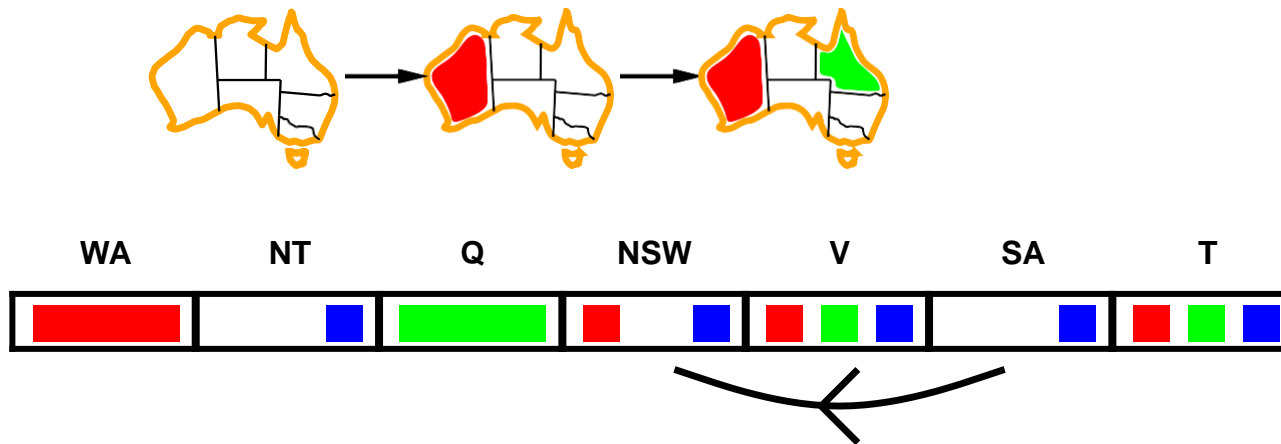
NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc consistency

Simplest form of propagation makes each arc **consistent**

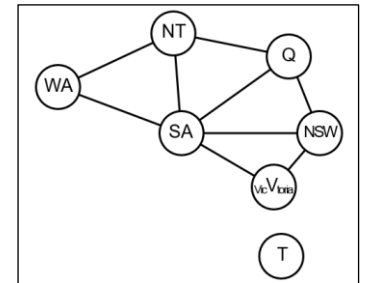
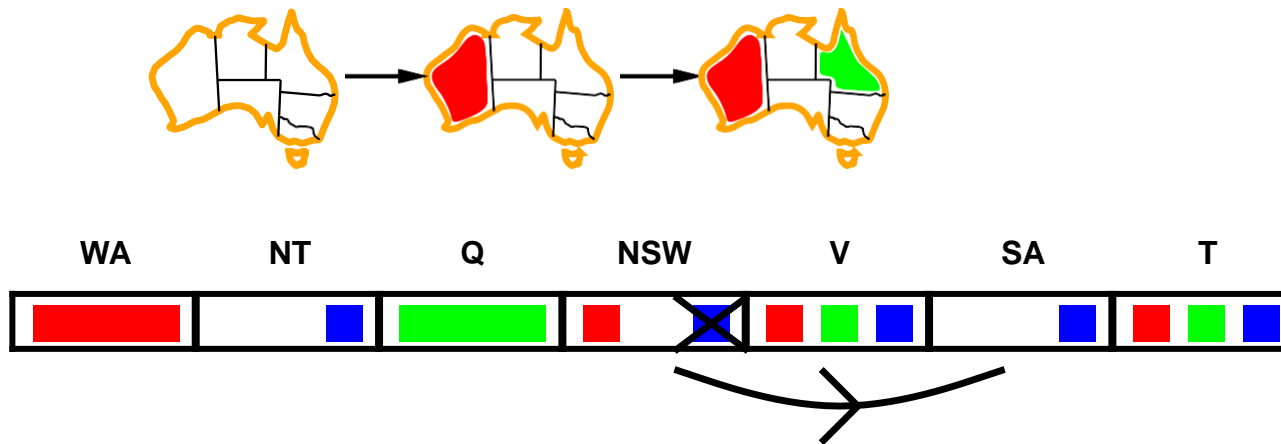
$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed y



Arc consistency

Simplest form of propagation makes each arc **consistent**

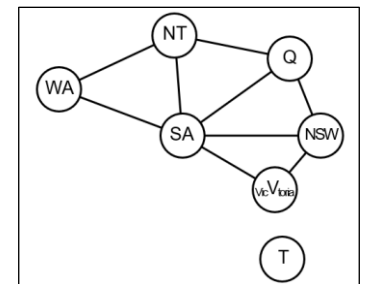
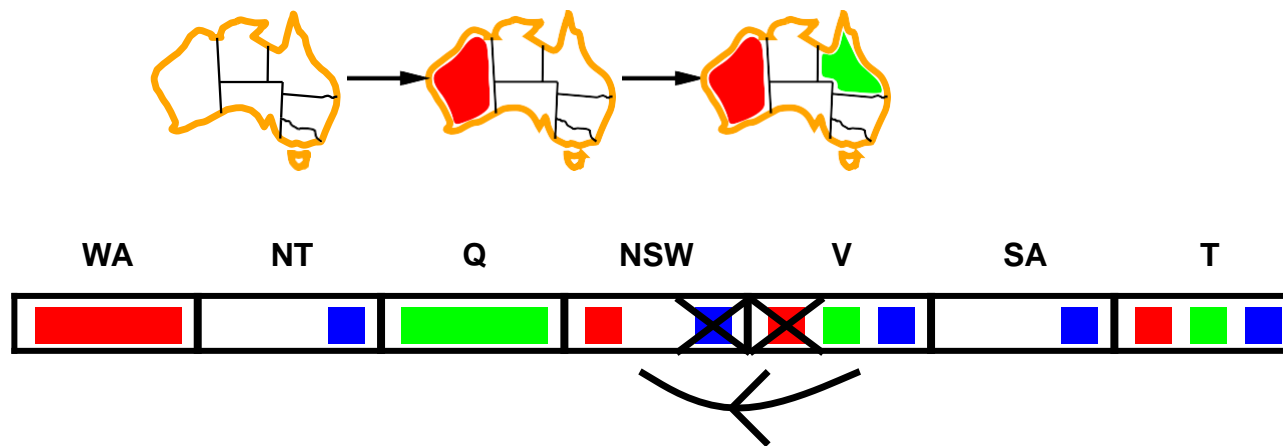
$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed y



Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed y

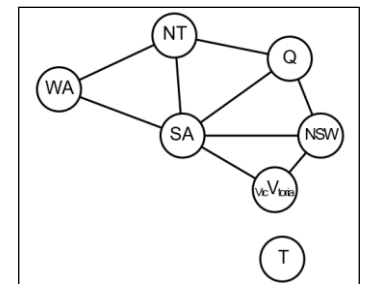
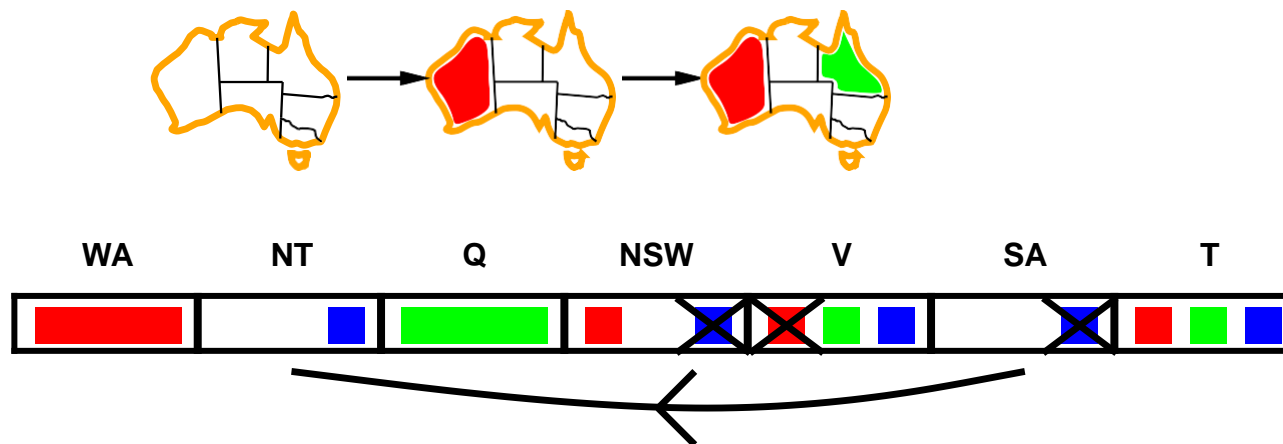


If X loses a value, neighbors of X need to be rechecked

Arc consistency

Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$ 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

We can see that **Maintaining AC (MAC)** is strictly more powerful than **forward checking** because forward checking does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, forward checking does **not recursively propagate constraints when changes are made to the domains of variables**.

Arc consistency algorithm

function **AC-3**(*csp*) **returns** the CSP, possibly with reduced domains

inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \dots, X_n\}$

local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

while *queue* is not empty **do**

$(X_i, X_j) \leftarrow \text{Remove-First}(\textit{queue})$

if **Remove-Inconsistent-Values**(X_i, X_j) **then**

for each X_k **in** **Neighbors**[X_i] **do**

 add (X_k, X_i) to *queue*

function **Remove-Inconsistent-Values**(X_i, X_j) **returns** true iff succeeds

removed \leftarrow false

for each x **in** **Domain**[X_i] **do**

if no value y in **Domain**[X_j] allows (x, y) to satisfy the constraint $X_i \leftrightarrow X_j$

then delete x from **Domain**[X_i]; *removed* \leftarrow true

return *removed*

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

Local Search for CSPs

Local search algorithms can be very effective in solving many CSPs.

Local search algorithms use a **complete-state formulation** where each state assigns a value to every variable, and the search changes the value of one variable at a time.

Min-conflicts heuristic: value that results in the **minimum number of conflicts** with other variables that **brings us closer to a solution**.

- Usually has a series of **plateaus**

Plateau search: allowing sideways moves to another state with the same score.

- can help local search find its way off the plateau.

Constraint weighting aims to concentrate the search on the important constraints

- Each constraint is given a numeric weight, initially all 1.
- weights adjusted by incrementing when it is violated by the current assignment

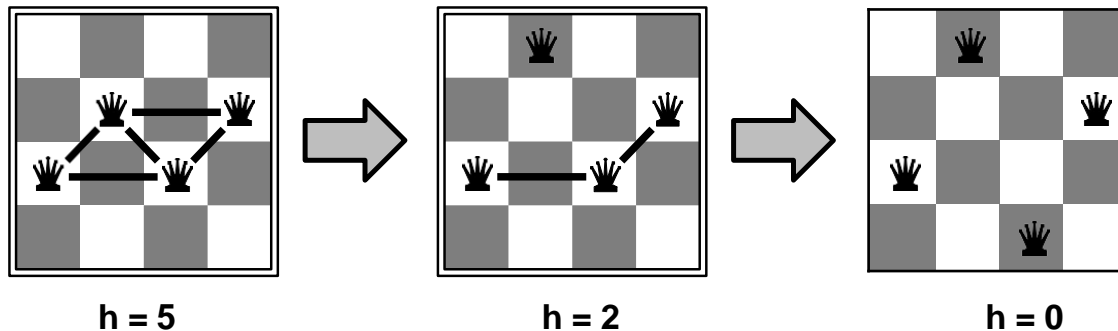
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen within column

Goal test: no attacks

Evaluation: $h(n)$ = number of attacks

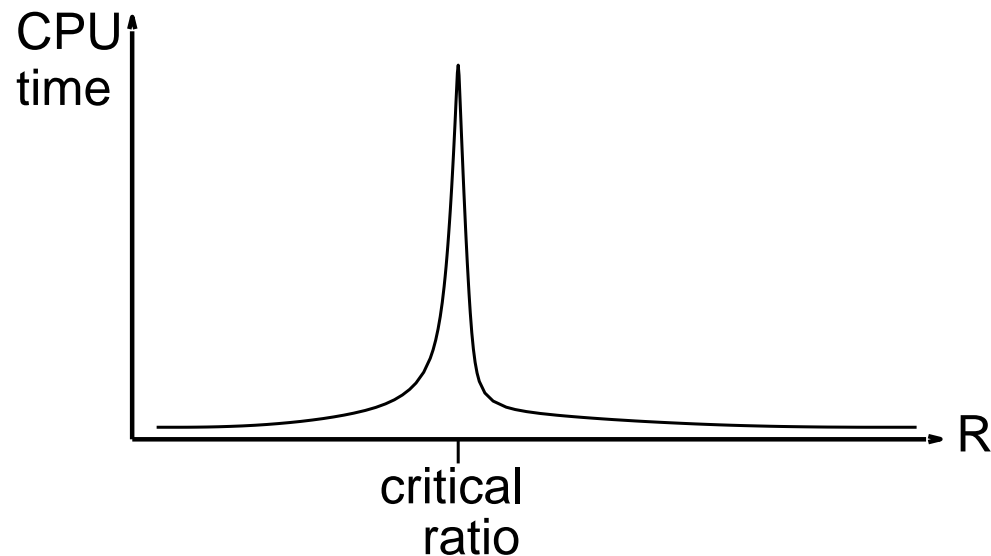


Performance of min-conflicts

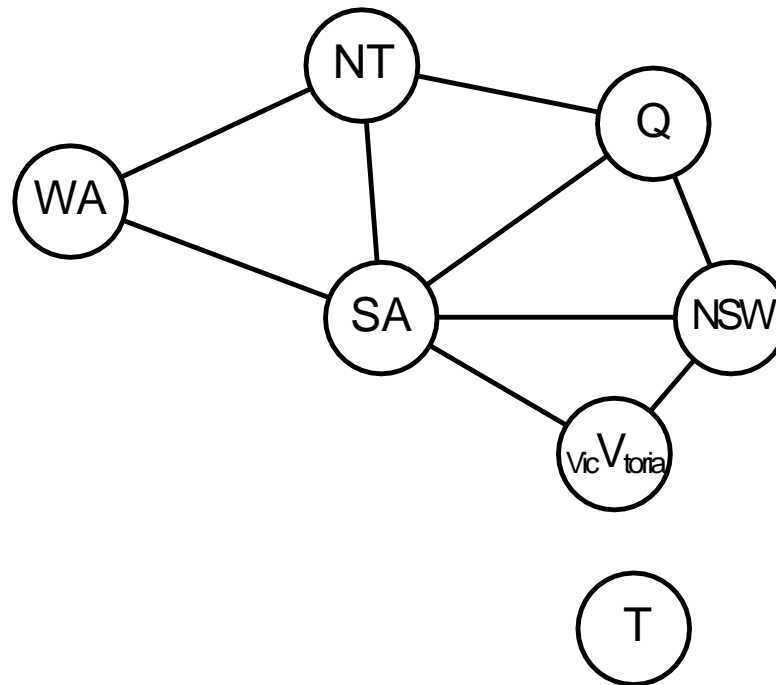
Given random initial state, can solve n -queens in almost constant time for arbitrary n with high probability (e.g., $n = 10,000,000$)

The same appears to be true for any randomly-generated CSP **except** in a narrow range of the ratio

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



Problem structure



Tasmania and mainland are **independent subproblems**

Identifiable as **connected components** of constraint graph

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

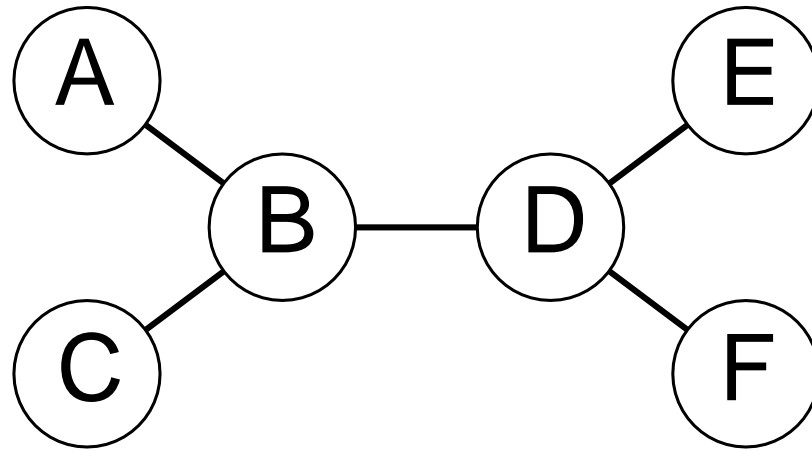
Completely independent subproblems are delicious, then, but rare.

Fortunately, some other graph structures are also easy to solve.

For example, a **constraint graph is a tree** when any two variables are connected by only one path.

We will show that *any tree-structured CSP can be solved in time linear in the number of variables.*

Tree-structured CSPs



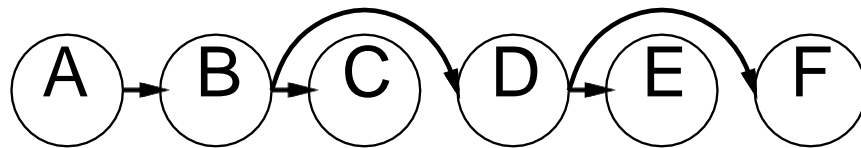
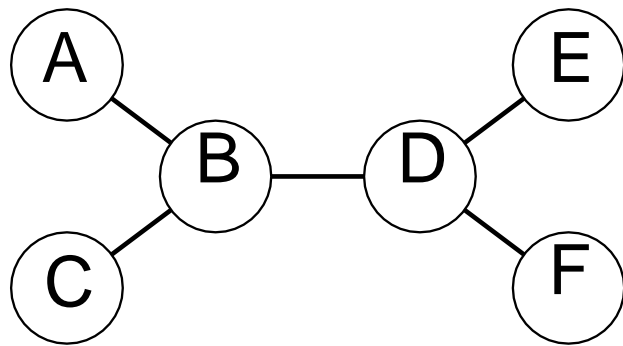
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time

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.

Algorithm for tree-structured CSPs

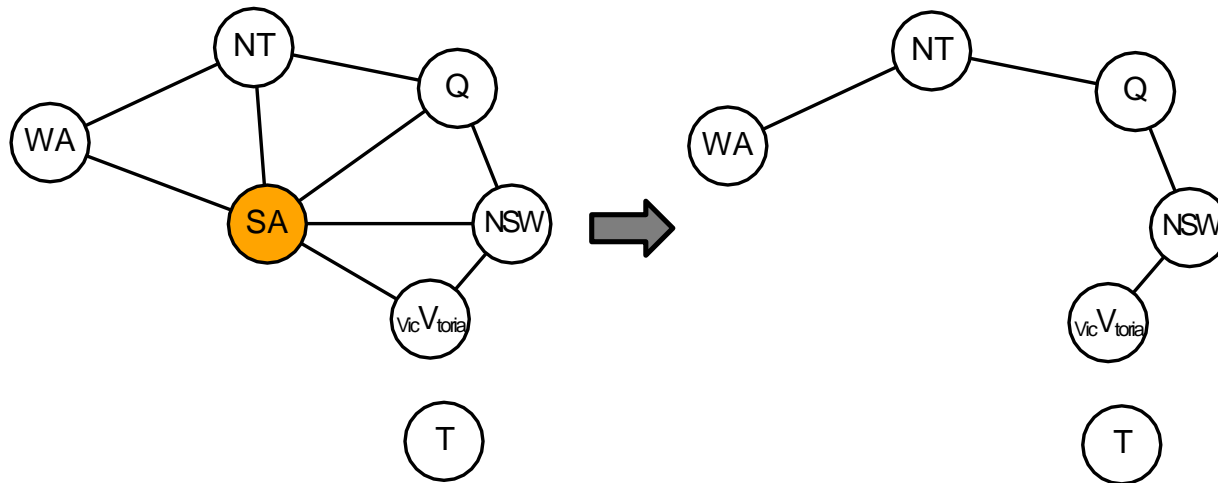
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For j from n down to 2 , apply $\text{RemoveInconsistent}(\text{Parent}(X_j), X_j)$
3. For j from 1 to n , assign X_j consistently with $\text{Parent}(X_j)$

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 $c \Rightarrow$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small c

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

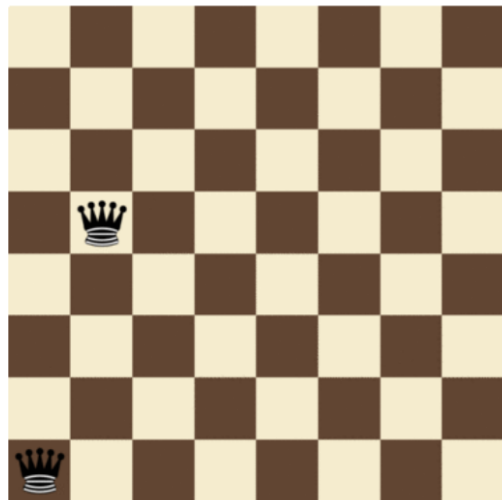
i.e., hillclimb with $h(n)$ = total number of violated constraints

CSPs in Practice

Constraint Satisfaction Problems Lab

Introduction

Constraint Satisfaction is a technique for solving problems by expressing limits on the values of each variable in the solution with mathematical constraints. For example, constraints in [the Sudoku project](#) are enforced implicitly by filtering the legal values for each box, and [the planning project](#) represents constraints as arcs connecting nodes in the planning graph. In this project we will use [SymPy](#), a symbolic math library, to explicitly construct binary constraints and then use Backtracking to solve the N-queens problem (which is a generalization [8-queens problem](#)). Using symbolic constraints makes it easier to visualize and reason about the constraints (especially for debugging), but comes with a performance penalty. See the [SymPy_Intro notebook](#) in the same directory for example code on sympy.




Briefly, the 8-queens problem asks us to place 8 queens on a standard 8x8 chessboard such that none of the queens are in "check" (i.e., no two queens occupy the same row, column, or diagonal). The N-queens problem generalizes the puzzle to to any size square board.

This project consists of three main steps:

- **Step 1:** Implement the NQueensCSP class to develop an efficient encoding of the N-queens problem and explicitly generate the constraints bounding the solution
- **Step 2:** Implement the search functions for recursive backtracking
- **Step 3:** Solve the N-queens problem

«[Constraint Satisfaction Problems Lab](#)» By Trang Nguyen.

CSPs with «python-constraint» v1.4.0

 queens-solver.py

Raw

```
1  #!/usr/bin/env python2
2
3  from constraint import *
4
5  problem = Problem()
6  cols = range(1,9)      # these are the variables
7  rows = range(1,9)      # these are the domains
8  problem.addVariables(cols, rows)      # adding multiple variables at once
9
10 # that each queen has to be in a separate column is
11 # implied through the loop and added constraints
12 for col1 in cols:
13     for col2 in cols:
14         if col1 < col2:
15             problem.addConstraint(lambda row1, row2, col1=col1, col2=col2:
16                 abs(row1-row2) != abs(col1-col2) and      # this is the diagonal check
17                 row1 != row2, (col1, col2))                # this is the horizontal check
18
19 solution = problem.getSolution()
20 print solution
```

Doc + examples: <https://pypi.org/project/python-constraint>

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

Local search using the min-conflicts heuristic has also been applied to constraint satisfaction problems with great success

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

In the next lecture...

- ◆ Game Theory
- ◆ Optimal Decisions in Games
 - minimax decisions
 - α - β pruning
 - Monte Carlo Tree Search (MCTS)
- ◆ Resource limits and approximate evaluation
- ◆ Games of chance
- ◆ Games of imperfect information
- ◆ Limitations of Game Search Algorithms