Constraint Satisfaction Problems

COMP3411/9814: Artificial Intelligence

Constraint Satisfaction Problems

- Assignment problems (e.g. who teaches what class)
- Timetabling problems (e.g. which class is offered when and where?)
- Hardware configuration (e.g. minimise space for circuit layout)
- Transport scheduling (e.g. courier delivery, vehicle routing)
- Factory scheduling (optimise assignment of jobs to machines)
- Gate assignment (assign gates to aircraft to minimise transit)

Closely related to optimisation problems

Lecture Overview

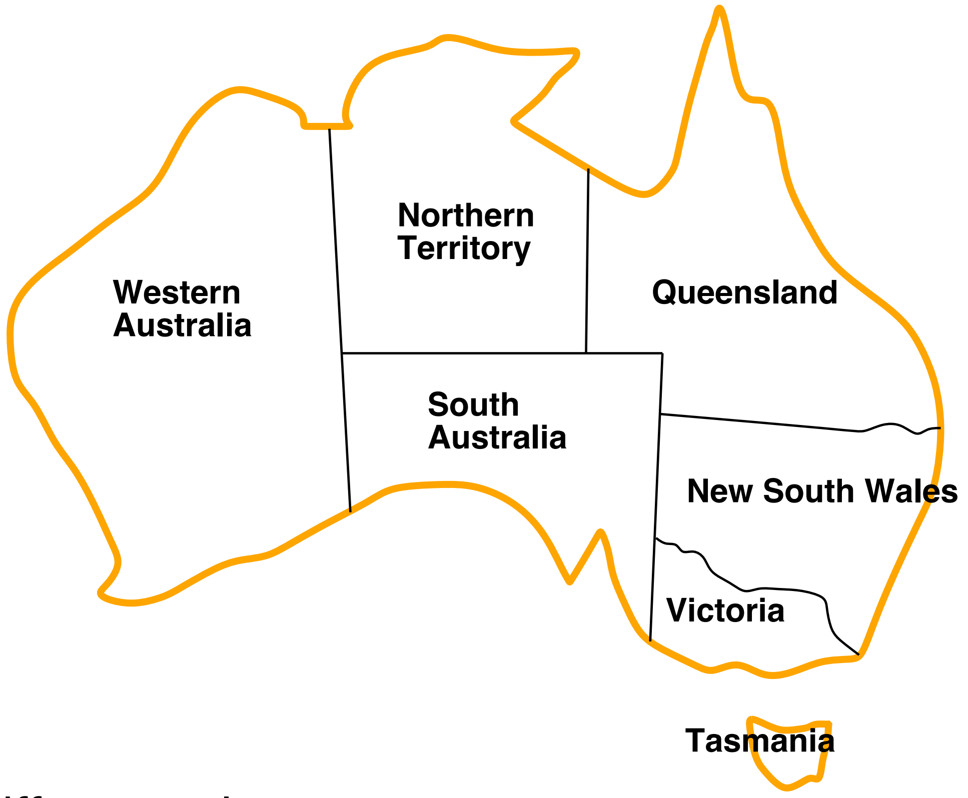
- Constraint Satisfaction Problems (CSPs)
- CSP examples
- Backtracking search and heuristics
- Forward checking and arc consistency
- Variable elimination
- Local search
    - Hill climbing
    - Simulated annealing

Constraint Satisfaction Problems (CSPs)

- Constraint Satisfaction Problems are defined by a set of variables X_i, each with a domain D_i of possible values, and a set of constraints C that specify allowable combinations of values.

- The aim is to find an assignment of the variables X_i from the domains D_i in such a way that none of the constraints C are violated.

Example: Map-Colouring



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

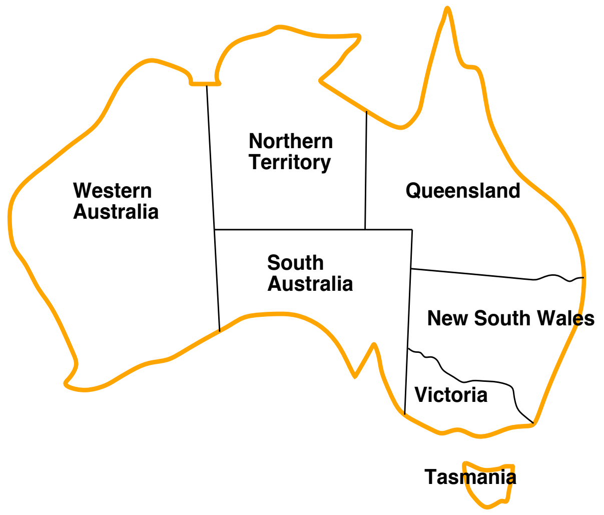
Domains: D_i = {red, green, blue}

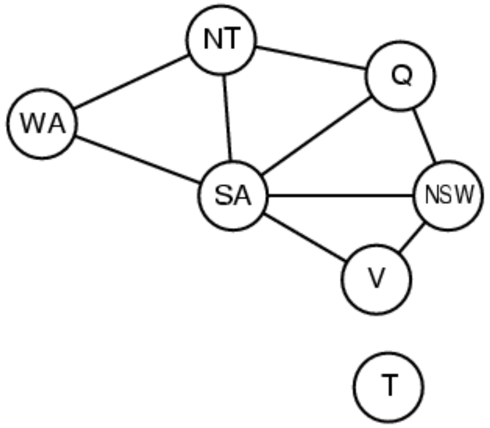
Constraints: adjacent regions must have different colours

e.g. WA≠ NT, etc.

Constraint graph

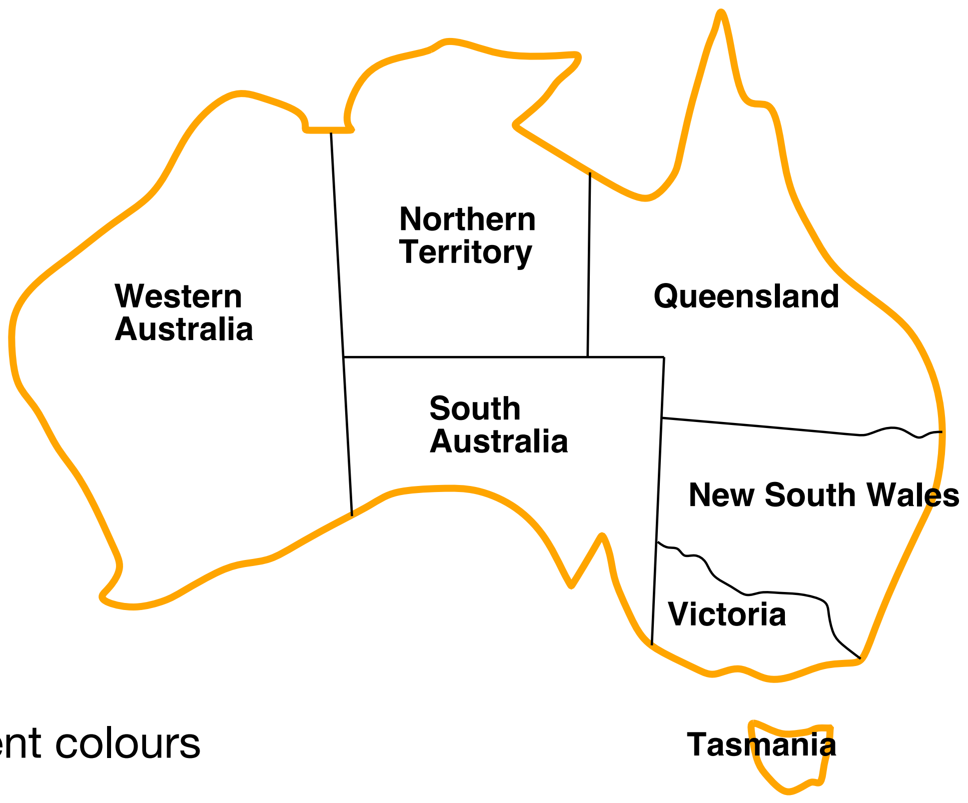
Constraint graph: nodes are variables, arcs are constraints





Binary CSP: each constraint relates two variables

Example: Map-Colouring



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

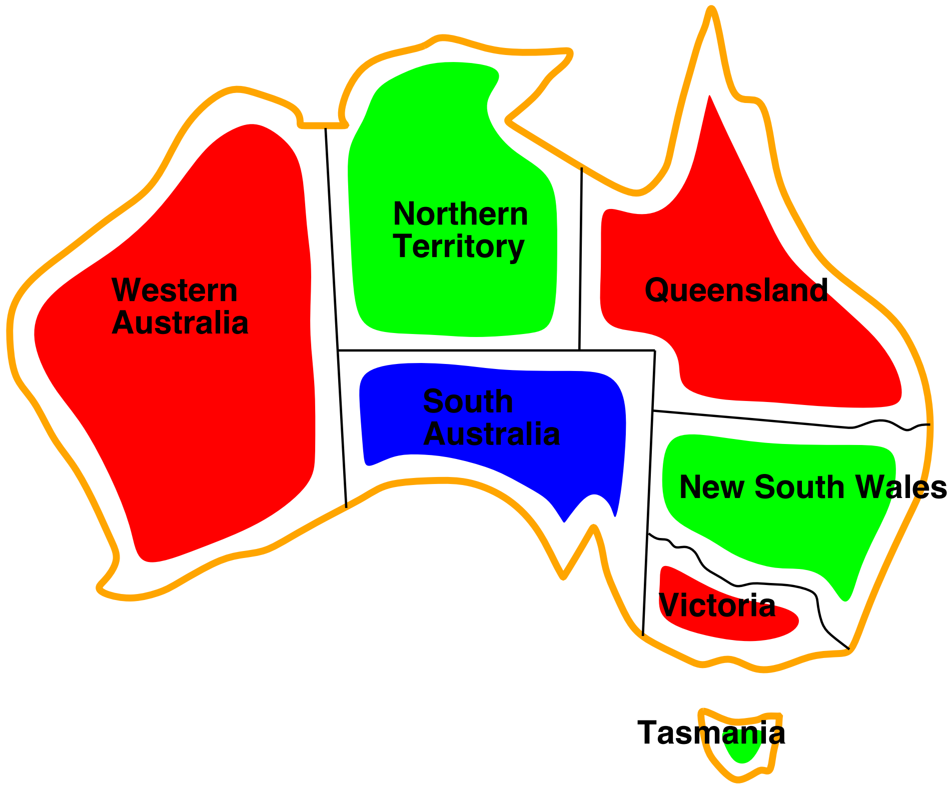
Domains: D_i = {red, green, blue}

Constraints: adjacent regions must have different colours

e.g. WA$\neq$ NT, etc.

or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)}

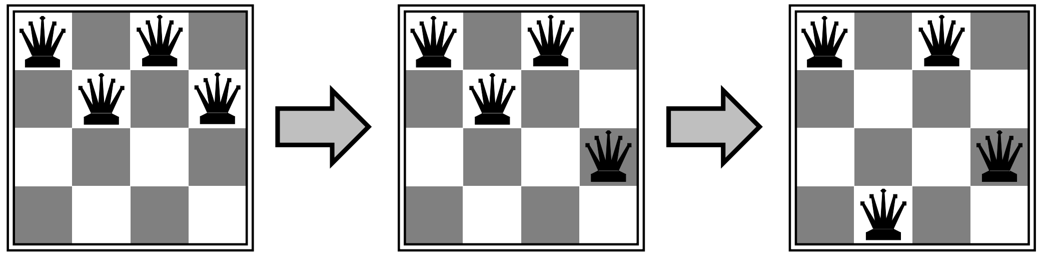
Example: Map-Colouring



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

n-Queens Puzzle as a CSP

Assume one queen in each column. Domains are possible positions of queen in a column. Assignment is when each domain has one element. Which row does each one go in?



Variables: $Q1$, $Q2$, $Q3$, $Q4$

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)

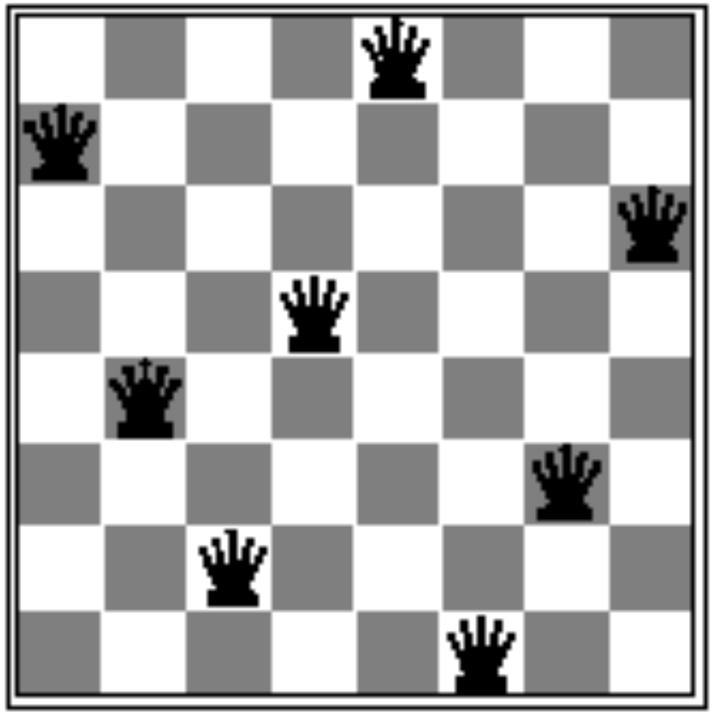
{1, 2, 1, 3}

Violates constraints because

$Q_1 = Q_3$ **and**

$|Q_1 - Q_2| = |i - j| = |1 - 2| = 1$

Example: n-Queens Puzzle



Put n queens on an n-by-n chess board so that no two queens are attacking each other.

Example: Cryptarithmetic

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

Variables:
D E M N O R S Y

Domains:

{0,1,2,3,4,5,6,7,8,9}

Constraints:
$M \neq 0$, $S \neq 0$ (unary constraints)
$Y = D+E$ or $\boxed{Y = D+E-10}$, etc.
$D \neq E$, $D \neq M$, $D \neq N$, etc.

if there is a carry over, have to add to next column

Example: Cryptarithmetic

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

Variables: F T U W R O C_1 C_2 C_3

Domains: {0,1,2,3,4,5,6,7,8,9}

Constraints: AllDifferent(F, T , U , W , R , O)

$O + O = R + 10 \cdot C_1$

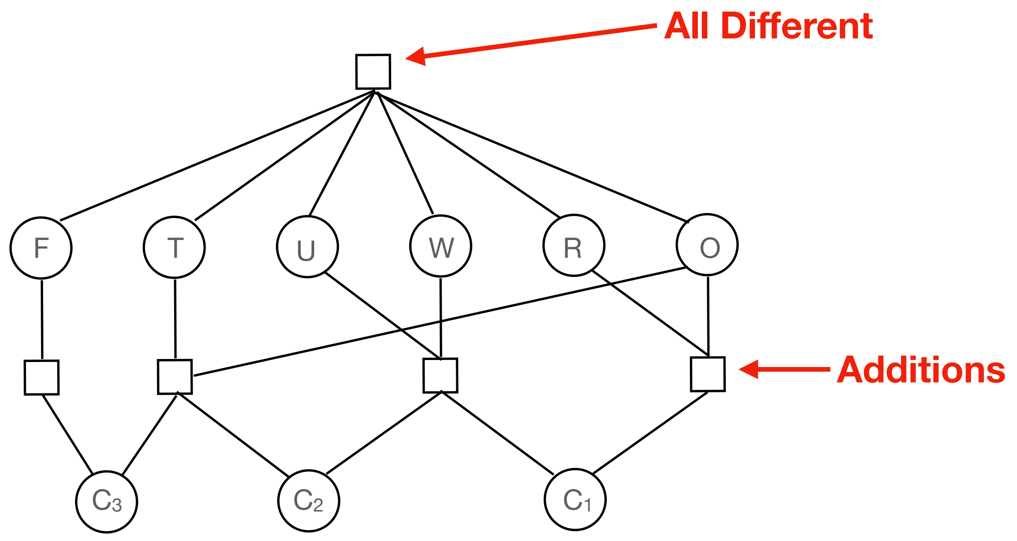
$C_1 + W + W = U + 10 \cdot C_2$

$C_2 + T + T = O + 10 \cdot C_3$

$C_3 = F$

Cryptarithmetic with Auxiliary Variables

$$\begin{array}{ccccc} & & T & W & O \\ + & & T & W & O \\ \hline & F & O & U & R \end{array}$$



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: AllDifferent(F, T , U , W , R , O)

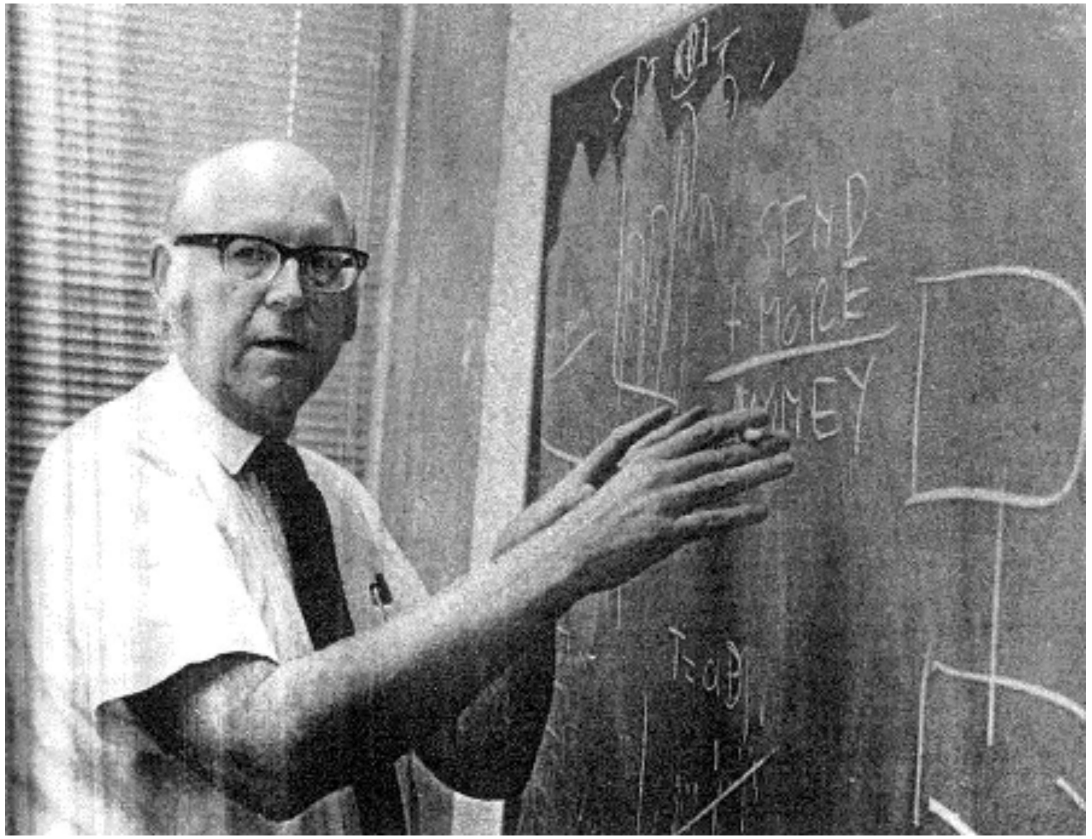
$O + O = R + 10 \cdot C_1$

$C_1 + W + W = U + 10 \cdot C_2$

$C_2 + T + T = O + 10 \cdot C_3$

$C_3 = F$

Cryptarithmetic with Allen Newell



Book: Intended Rational Behavior

CSP Application - Factory scheduling

- An agent has to schedule a set of activities for a manufacturing process, involving casting, milling, drilling, and bolting.
- Each activity has a set of possible times at which it may start.
- The agent has to satisfy various constraints arising from prerequisite requirements and resource use limitations.
- For each activity there is a variable that represents the time that it starts:
  - B – start of bolding
  - D – start of drilling
  - C – start of casting

CSP Application - Factory scheduling

Constraints on the possible dates for three activities:

Variables: $A,\ B, C$ - variables that represent the date of each activity

Domain of each variable is: $\{1, 2, 3, 4\}$

Constraint: $(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3)$

> A starts on or before the same date as B and it cannot be that A and B start on the same date and C starts on or before day 3.

CSP Application - Factory scheduling

Constraint on the possible dates for three activities.

Variables: $A,\ B, C$ - variables that represent the date of each activity

Domain of each variable is: $\{1, 2, 3, 4\}$

Constraint:

$$(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3)$$

A starts on or before the same date as B and it cannot be that A and B start on the same date and C starts on or before day 3.

Constraint defines its **extension**, e.g. table specifying the legal assignments:

A	B	C
2	2	4
1	1	4
1	2	3
1	2	4

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.)
    - Job shop scheduling, variables are start/end days for each job
    - Need a constraint language, e.g. $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$
    - Linear constraints solvable, nonlinear undecidable
- Continuous variables
  - e.g. start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by *linear programming* methods

Types of constraints

- Unary constraints involve a single variable

  $M \neq 0$

- Binary constraints involve pairs of variables

  $SA \neq WA$

- Higher-order constraints involve 3 or more variables

  $Y = D + E$ or $Y = D + E - 10$

- Inequality constraints on continuous variables

  $EndJob1 + 5 \leq StartJob3$

- Soft constraints (preferences)

  11am lecture is better than 8am lecture!

Path Search vs Constraint Satisfaction

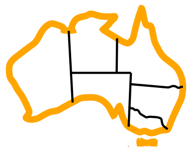
Difference between path search problems and CSPs

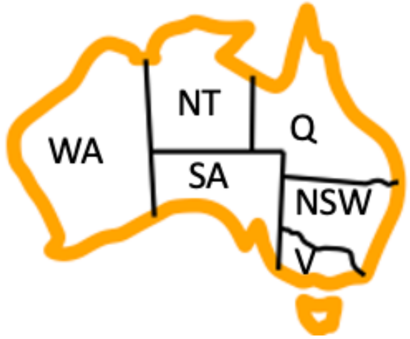
- Path Search Problems (e.g. Delivery Robot)
    - Knowing the final state is easy
    - Difficult part is how to get there
- Constraint Satisfaction Problems (e.g. n-Queens)
    - Difficult part is knowing the final state
    - How to get there is easy

Backtracking Search

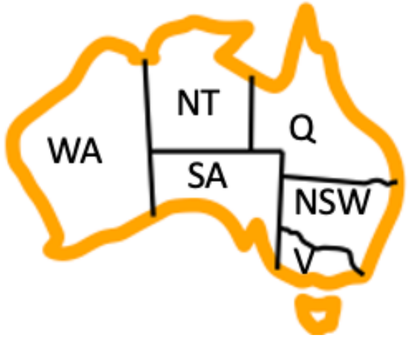
- CSPs can be solved by assigning values to variables one by one, in different combinations.
- Whenever a constraint is violated, go back to most recently assigned variable and assign it a new value.
- Can use Depth First Search, where states are defined by the values assigned so far:
  - Initial state: empty assignment.
  - Successor function:
    - assign a value to an unassigned variable that does not conflict with previously assigned values of other variables.
    - If no legal values remain, the successor function fails
  - Goal test: all variables have been assigned a value, and no constraints have been violated.

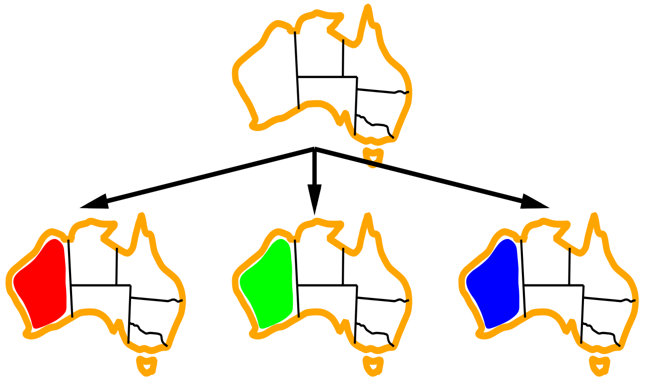
Backtracking example



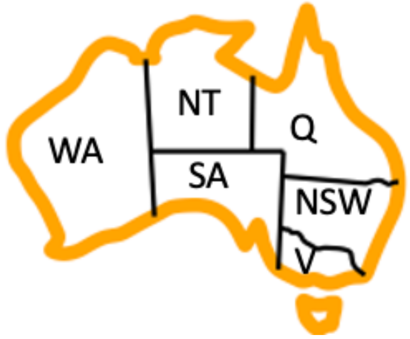


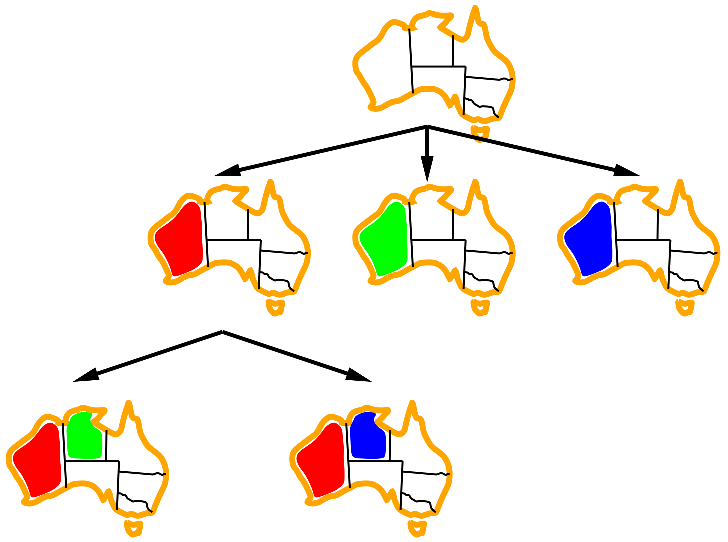
Backtracking example



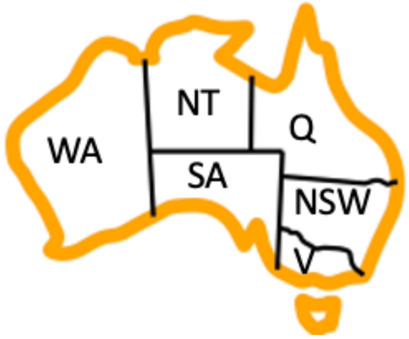


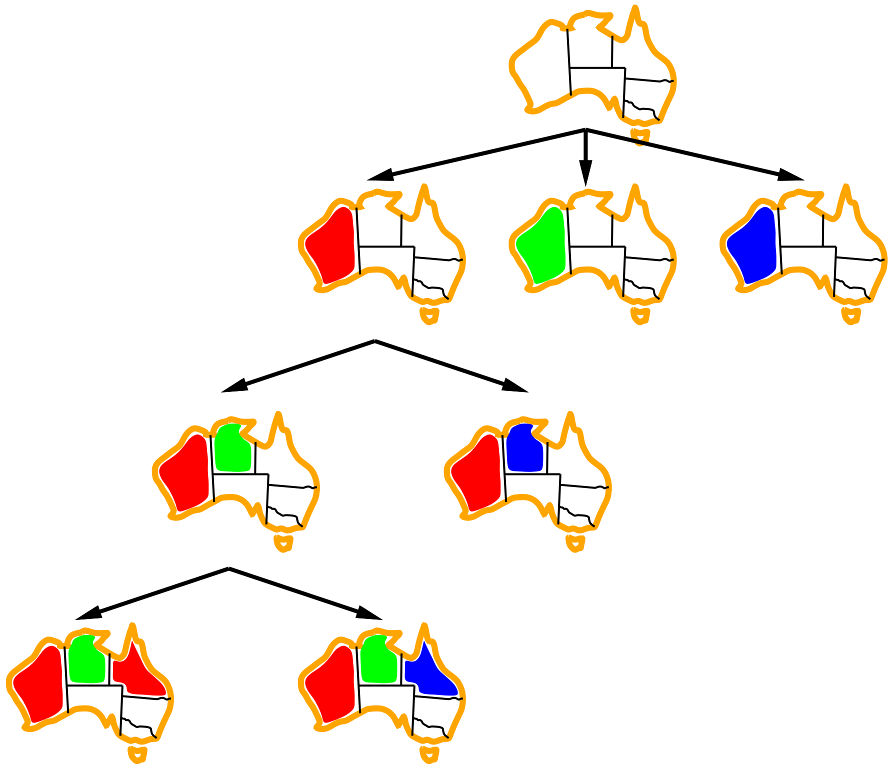
Backtracking example





Backtracking example





Backtracking Search Properties

- If there are n variables, every solution will occur at exactly depth n.
- Variable assignments are commutative

  [WA = red then NT = green] same as [NT = green then WA = red]

- Backtracking search can solve n-Queens for n $\approx$ 25

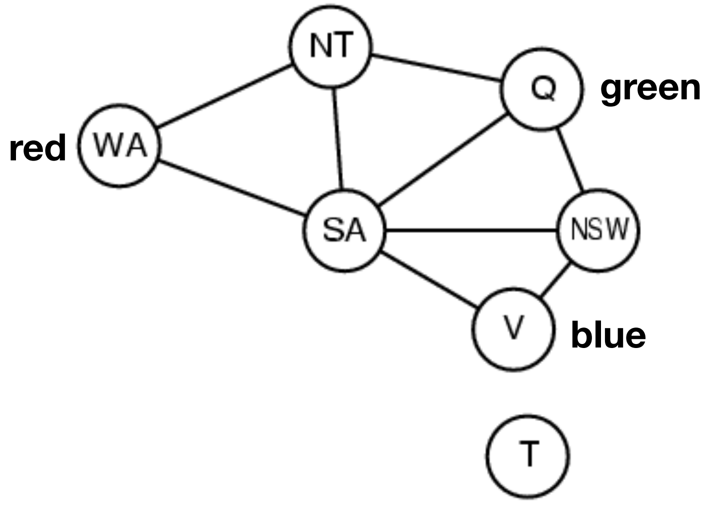
Programming Constraints

```
variables([wa=_, nt=_, q=_, nsw=_, v=_, sa=_, t=_]).

domain(red).
domain(green).
domain(blue).

connected(wa, nt).
connected(wa, sa).
connected(nt, q).
connected(nt, sa).
connected(sa, q).
connected(sa, nsw).
connected(sa, v).
connected(q, nsw).
connected(v, nsw).

adjacent(A, B) :- connected(A, B).
adjacent(A, B) :- connected(B, A).
```



Programming Constraints

```
solve(V) :-
    variables(V),
    assign_all(V).

assign_all([]).
assign_all([State|OtherStates]):-
    assign_all(OtherStates),
    assign_variable(State, OtherStates).

assign_variable(Var = Colour, OtherStates) :-
    domain(Colour),
    constraint(Var = Colour, OtherStates).

constraint(S1 = C, OtherStates) :-
    \+ (adjacent(S1, S2), member(S2 = C, OtherStates)).
```

Prolog's *not* operator

All Solutions by Backtracking

WA	NT	Q	NSW	V	SA	T
green	blue	green	blue	green	red	red
blue	green	blue	green	blue	red	red
red	blue	red	blue	red	green	red
blue	red	blue	red	blue	green	red
red	green	red	green	red	blue	red
green	red	green	red	green	blue	red
green	blue	green	blue	green	red	green
blue	green	blue	green	blue	red	green
red	blue	red	blue	red	green	green
blue	red	blue	red	blue	green	green
red	green	red	green	red	blue	green
green	red	green	red	green	blue	green
green	blue	green	blue	green	red	blue
blue	green	blue	green	blue	red	blue
red	blue	red	blue	red	green	blue
blue	red	blue	red	blue	green	blue
red	green	red	green	red	blue	blue
green	red	green	red	green	blue	blue

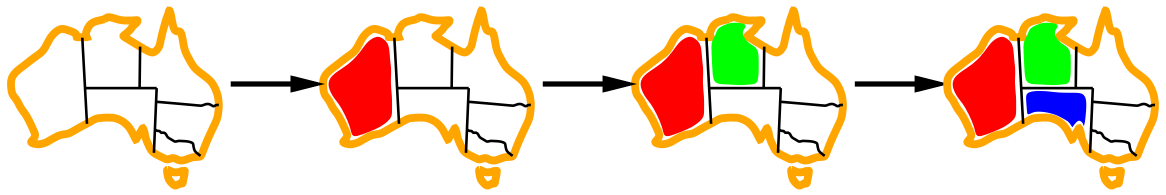
Improvements to Backtracking Search

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

Minimum Remaining Values

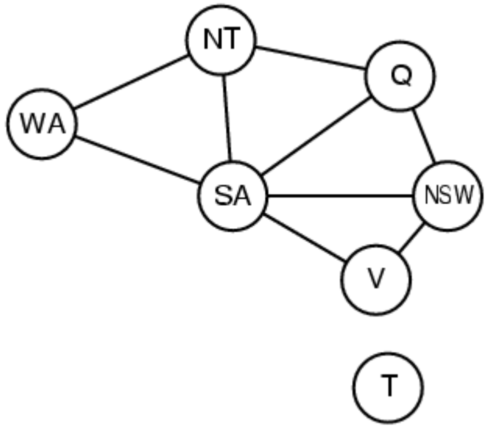
- Minimum Remaining Values (MRV)
  - choose the variable with the fewest legal remaining values
  - Most constrained variable

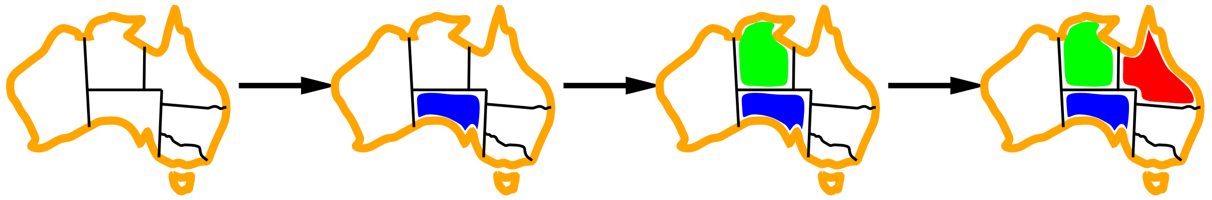
WA
NT
SA
Q
NSW
V



Degree Heuristic

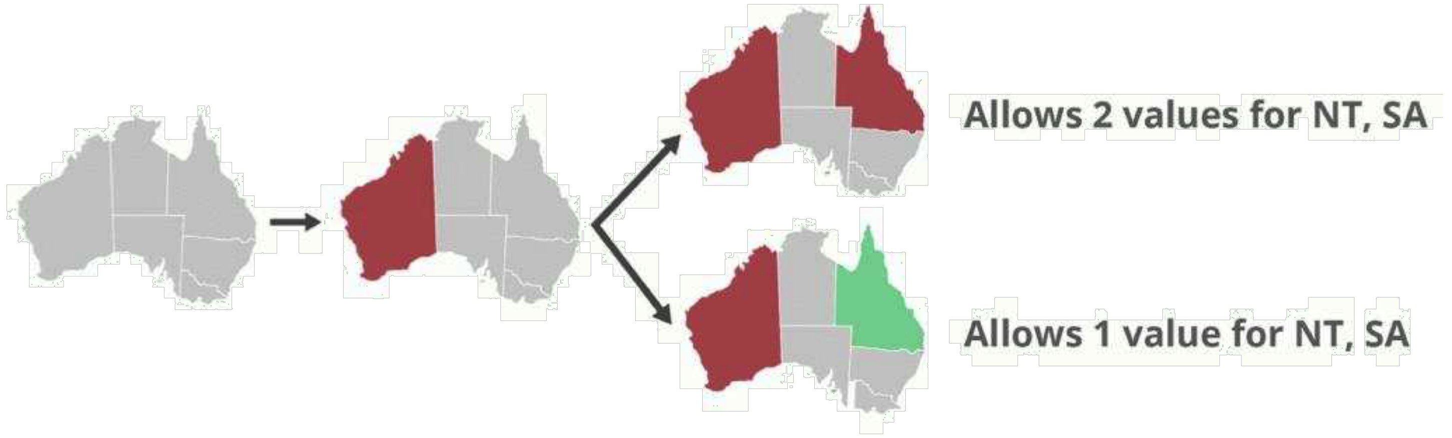
- Tie-breaker among MRV variables
- Degree heuristic:
  - choose the variable with the most constraints on variables (i.e. most edges in graph)
  - If same degree, choose any one

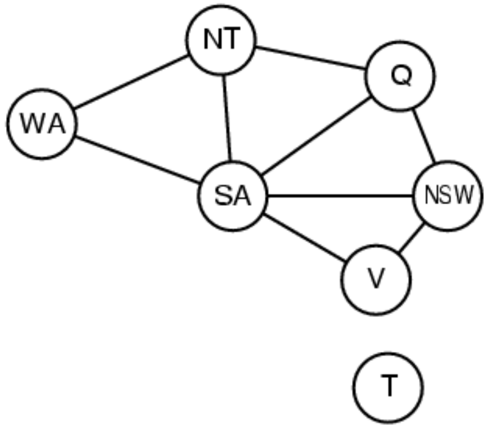




Least Constraining Value

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



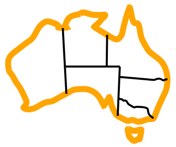


- More generally, 3 allowed values would be better than 2, etc.
  Combining these heuristics makes 1000 queens feasible.

Forward Checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
  - prune off that part of the search tree, and backtrack

WA
NT
Q
SA
NSW
V



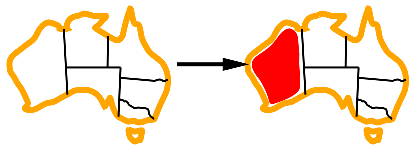
WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦

Initially, all values are available.

Forward Checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
  - prune off that part of the search tree, and backtrack

WA
NT
Q
SA
NSW
V

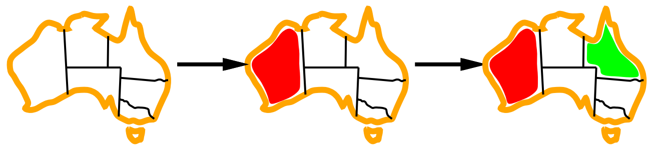


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦

Forward Checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
  - prune off that part of the search tree, and backtrack

WA
NT
Q
SA
NSW
V

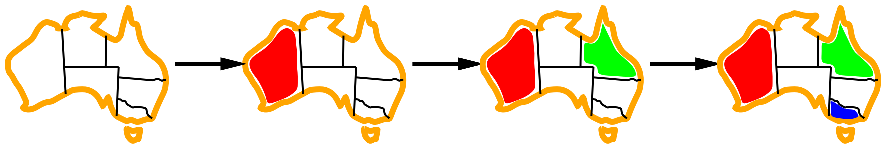


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

Forward Checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
  - prune off that part of the search tree, and backtrack

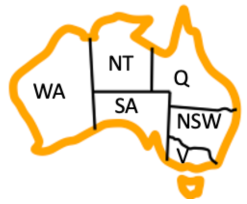
WA
NT
Q
SA
NSW
V

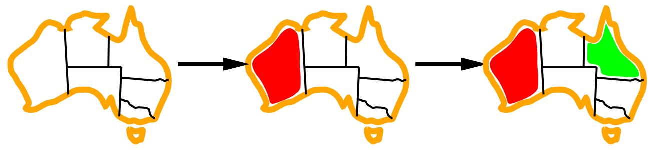


WA	NT	Q	NSW	V	SA	T	
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦	
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦	
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦	Fail

Constraint Propagation

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





WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

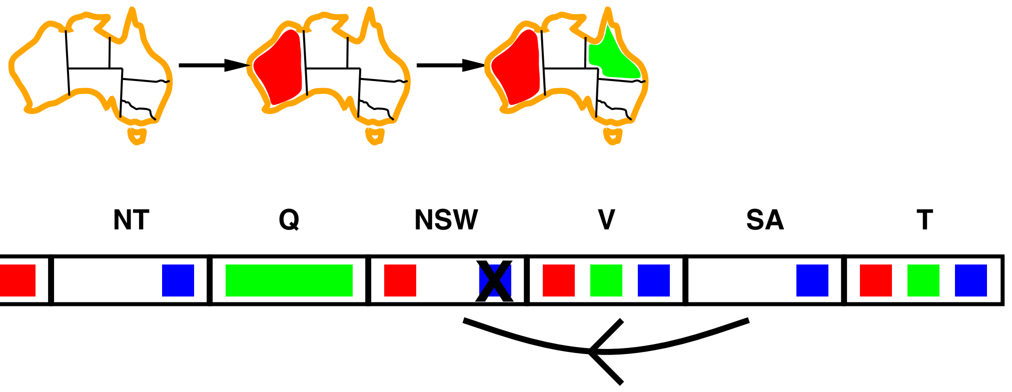
NT and SA cannot both be blue

Constraint propagation repeatedly enforces constraints locally

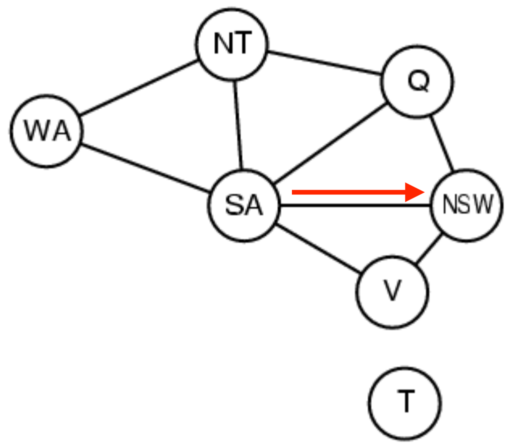
Arc Consistency

$X \rightarrow Y$ is consistent if

for every value x of X there is some allowed y



Only possible value fo SA is blue, so NSW can't blue



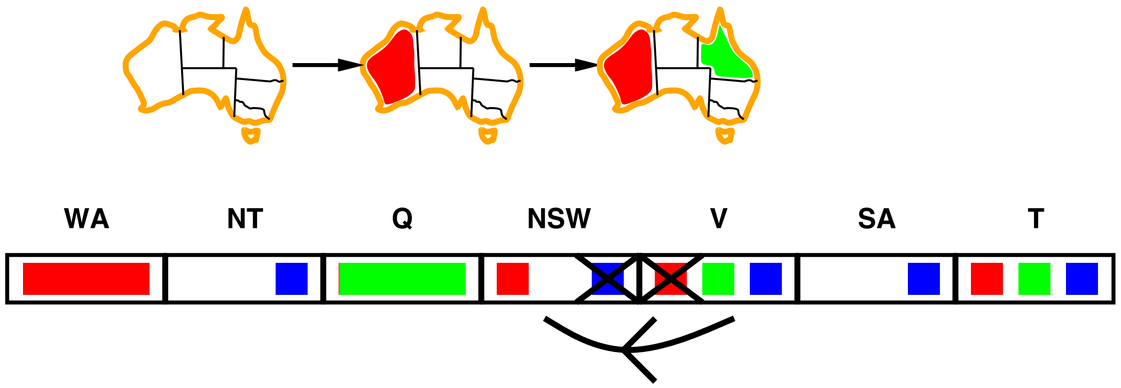
Propagate arc consistency on the graph

Arc Consistency

$X \rightarrow Y$ is consistent if

for every value x of X there is some allowed y

WA
NT
Q
SA
NSW
V



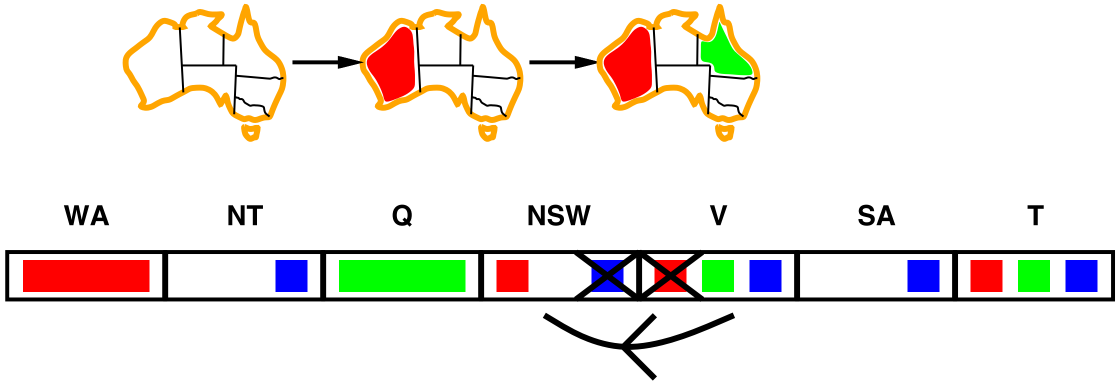
- If X loses a value, neighbours of X need to be rechecked.
- Since *NSW* can only be red now, *V* cannot be red

Arc Consistency

$X \rightarrow Y$ is consistent if

for every value x of X there is some allowed y

WA
NT
Q
SA
NSW
V

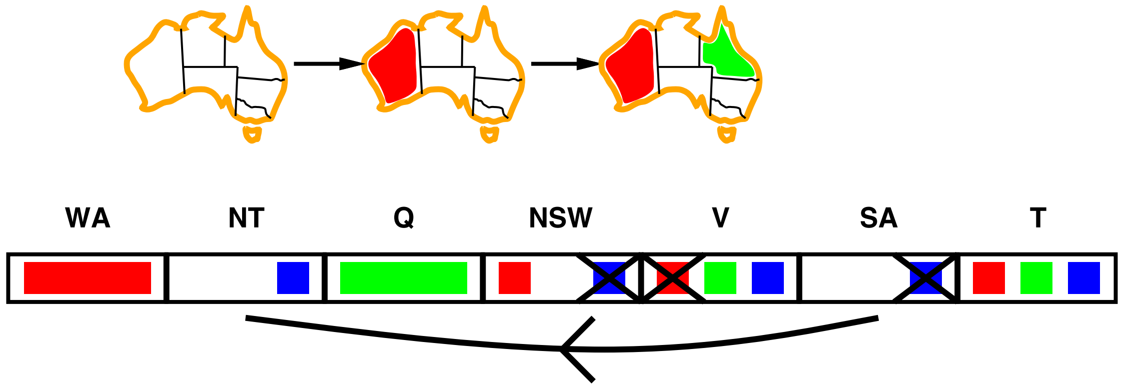


- If X loses a value, neighbours of X need to be rechecked.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor after each assignment

Arc Consistency

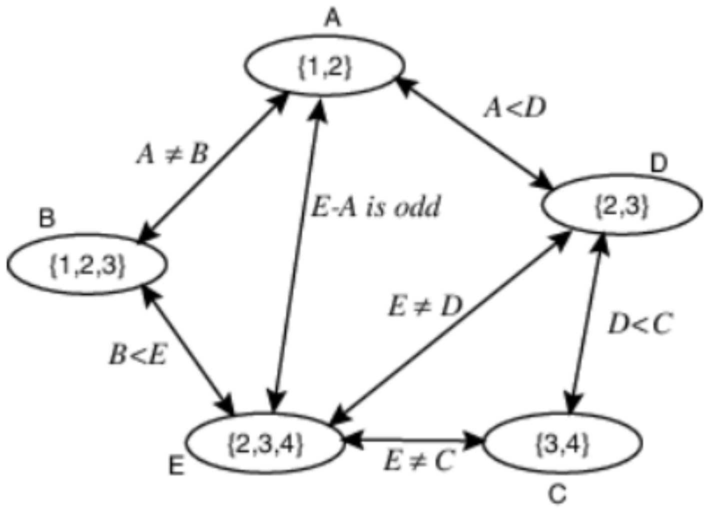
$X \rightarrow Y$ is consistent if

for every value x of X there is some allowed y



- Arc consistency detects failure earlier than forward checking.
- For some problems, it can speed up search enormously.
- For others, it may slow the search due to computational overheads.

Variable Elimination

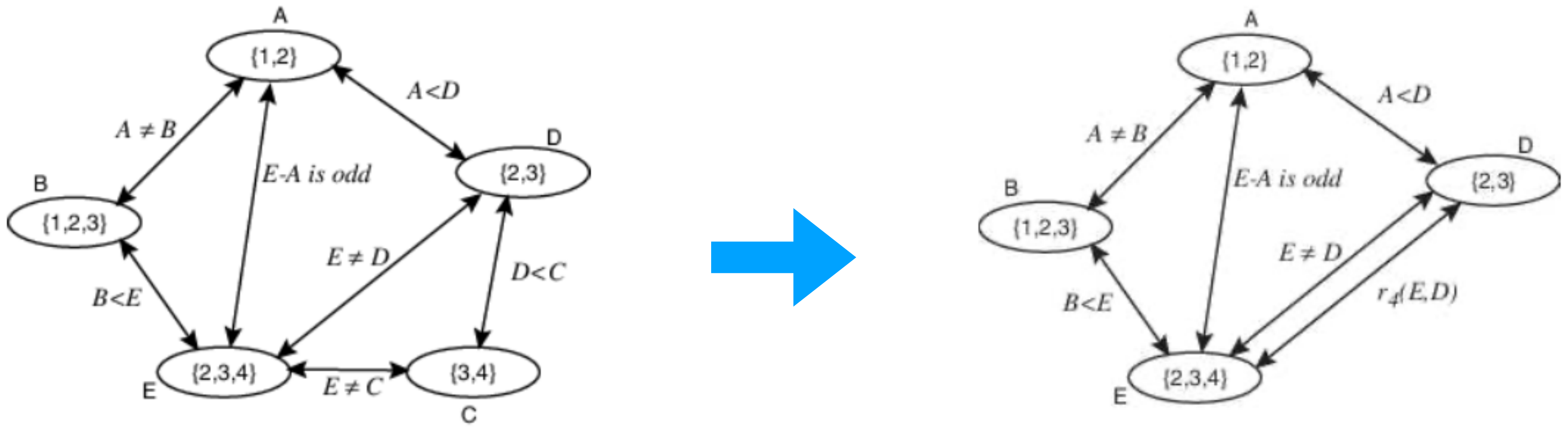


Variables: A, B, C, D, E

Domains: A = {1, 2}, B = {1, 2, 3}, C = {3, 4}, D = {2, 3}, E = {2, 3, 4}

Constraints: A ≠ B, E ≠ C, E ≠ D, A < D, B < E, D < C, E-A is odd

Variable Elimination

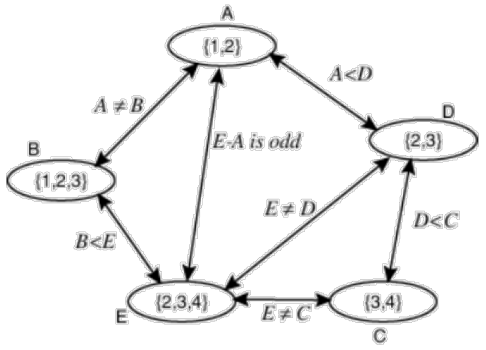


- Eliminates variable, one by one
- Replace them with constraints on adjacent variables

Variable Elimination Example

1. Select a variable X
2. Enumerate constraints

$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

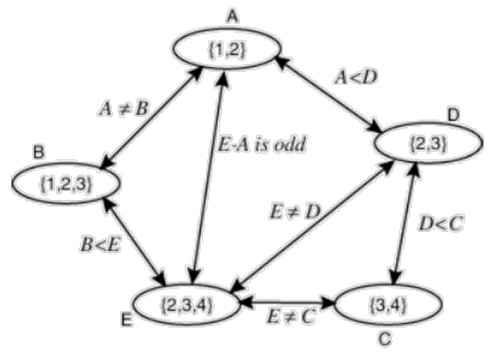


Variable Elimination Example

1. Select a variable X
2. Enumerate constraints

$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

$r_2 : C > D$	C	D
	3	2
	4	2
	4	3



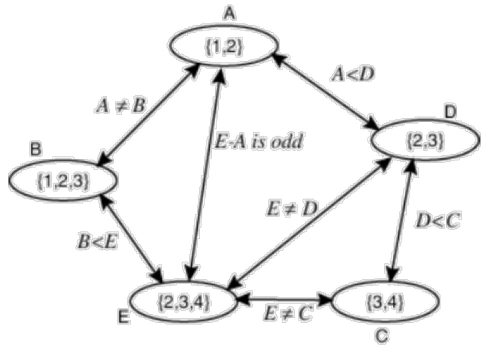
Variable Elimination Example

1. Select a variable X
2. Join the constraints in which X appears

$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

$r_2 : C > D$	C	D
	3	2
	4	2
	4	3

$r_3 : r_1 \bowtie r2$	C	D	E
	3	2	2
	3	2	4
	4	2	2
	4	2	3
	4	3	2
	4	3	3



Variable Elimination Example

1. Select a variable X
2. Join the constraints in which X appears
3. Project join onto its variables other than X (forming *r_4*)

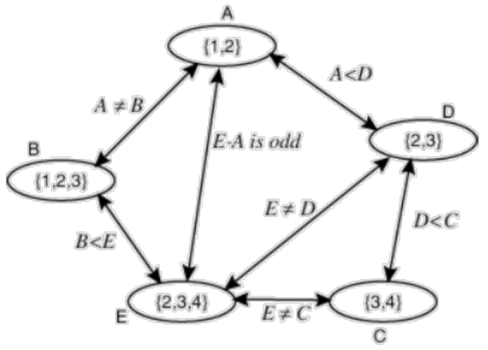
$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

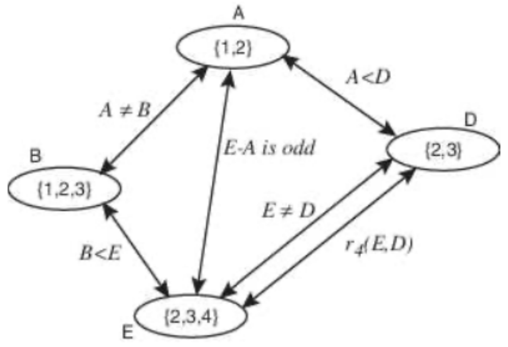
$r_2 : C > D$	C	D
	3	2
	4	2
	4	3

$r_3 : r_1 \bowtie r2$	C	D	E
	3	2	2
	3	2	4
	4	2	2
	4	2	3
	4	3	2
	4	3	3

$r_4 : \pi_{\{D,E\}} r_3$	D	E
	2	2
	2	3
	2	4
	3	2
	3	3

$\hookrightarrow$ new constraint

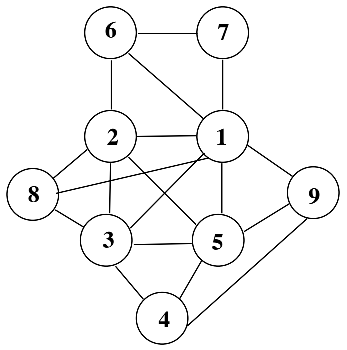




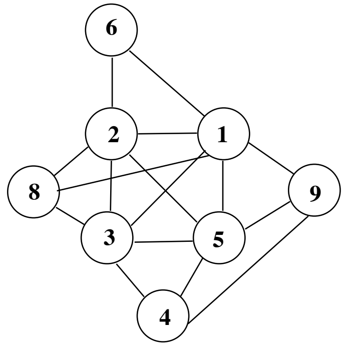
To generate one or all solutions, the algorithm remembers the joined relation *C*, *D*, *E* to construct a solution that involves *C* from a solution to the reduced network.

Variable Elimination

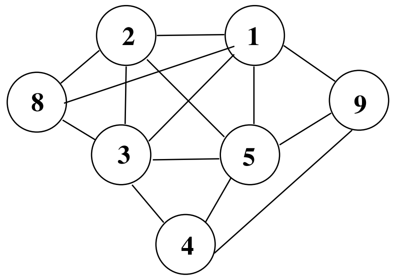
1. Select a variable X
2. Join the constraints in which X appears, forming constraint R1
3. Project R1 onto its variables other than X, forming R2
4. Replace all of the constraints in which X appears by R2
5. Recursively solve the simplified problem, forming R3
6. Return R1 joined with R3



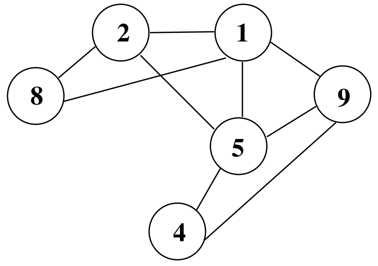
a) Initial constraint graph



b) After eliminating X7



c) After eliminating X6

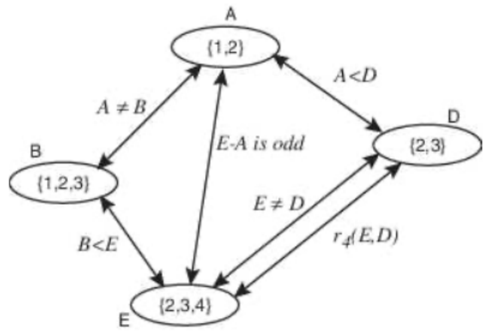


d) After branching in X3

Variable Elimination

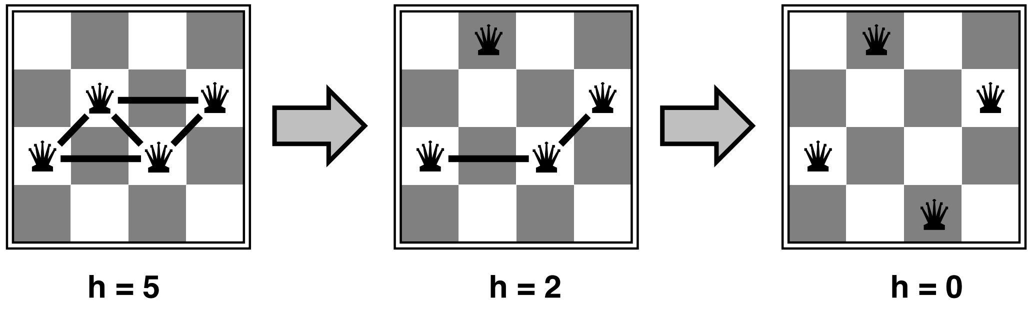
1. If there is only one variable,
   return the join of all the relations in the constraints

2. Otherwise
   - 2.1. Select a variable X
   - 2.2. Join the constraints in which X appears, forming constraint R1
   - 2.3. Project R1 onto its variables other than X, forming R2
   - 2.4. Replace all of the constraints in which X appears by R2
   - 2.5. Recursively solve the simplified problem, forming R3
   - 2.6. Return R1 joined with R3



Local Search

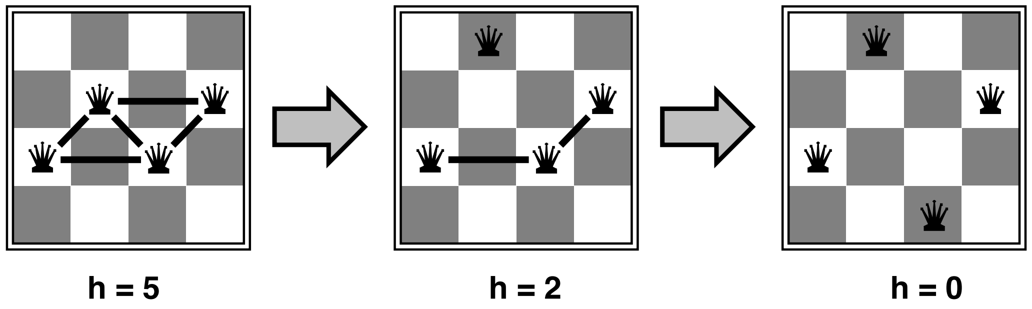
There is another class of algorithms for solving CSP’s, called “Iterative Improvement” or “Local Search”.



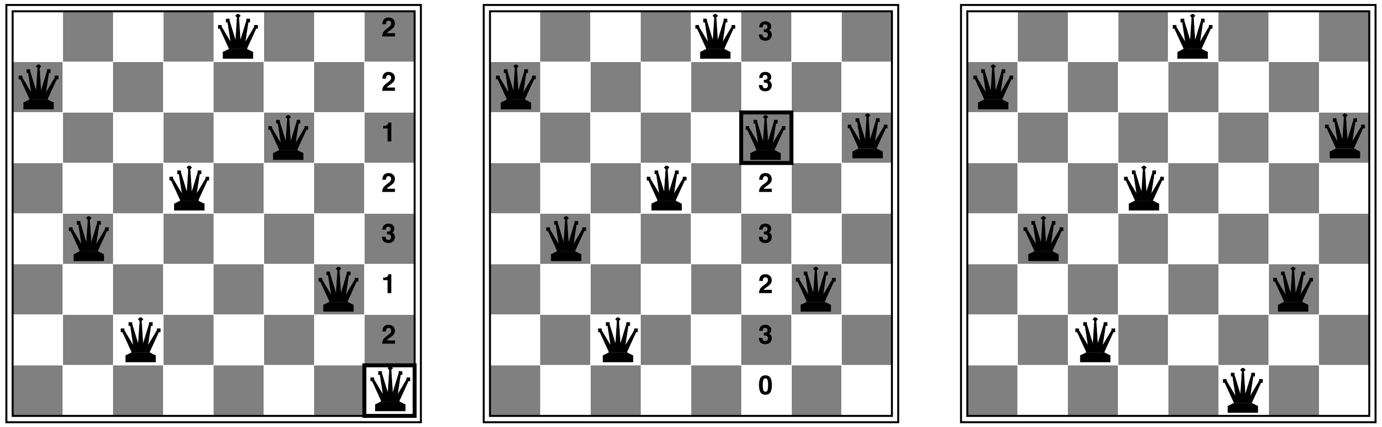
Local Search

There is another class of algorithms for solving CSP's, called "Iterative Improvement" or "Local Search".

- Iterative Improvement
  - assign all variables randomly in the beginning (thus violating several constraints),
  - change one variable at a time, trying to reduce the number of violations at each step.
  - Greedy Search with h = number of constraints violated

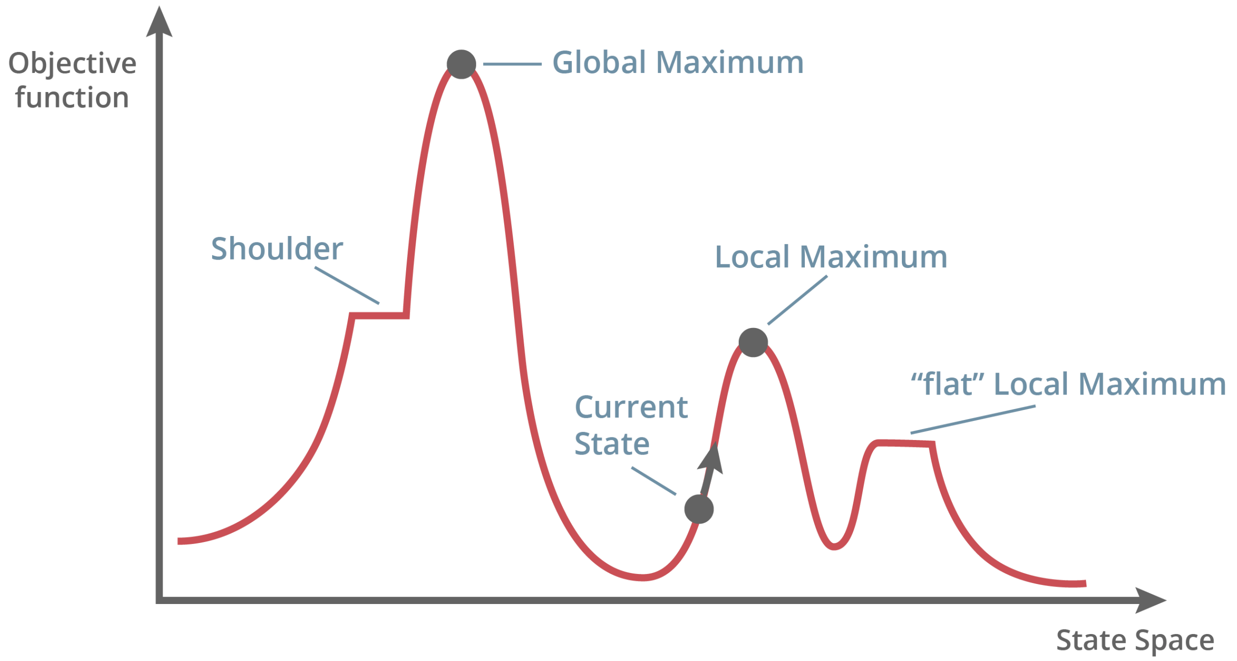


Hill-climbing by min-conflicts



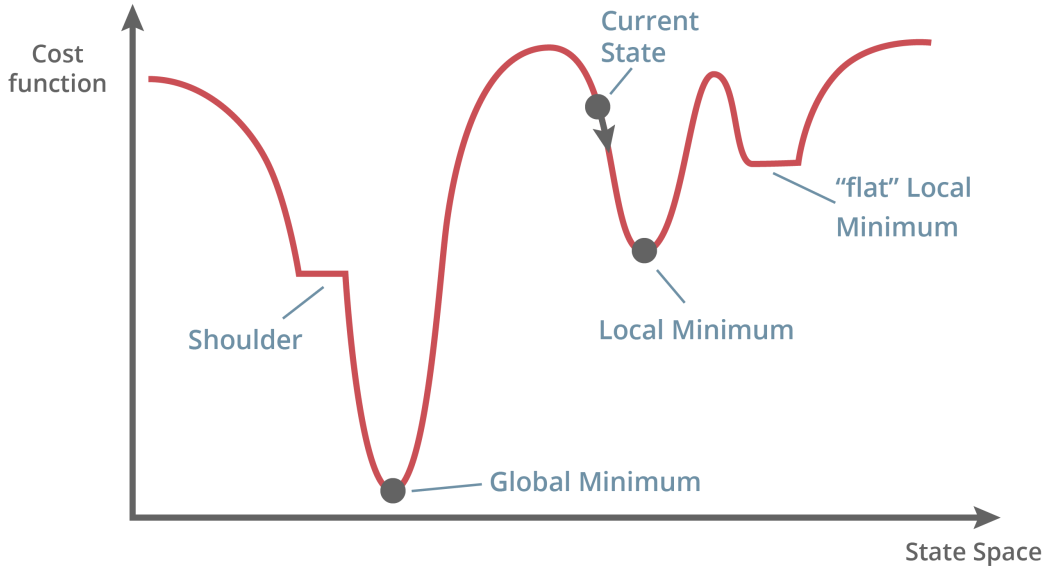
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic
    - choose value that violates the fewest constraints

Flat regions and local optima



- May have to go sideways or backwards to make progress towards the solution
- Exploration vs Exploitation

Inverted View



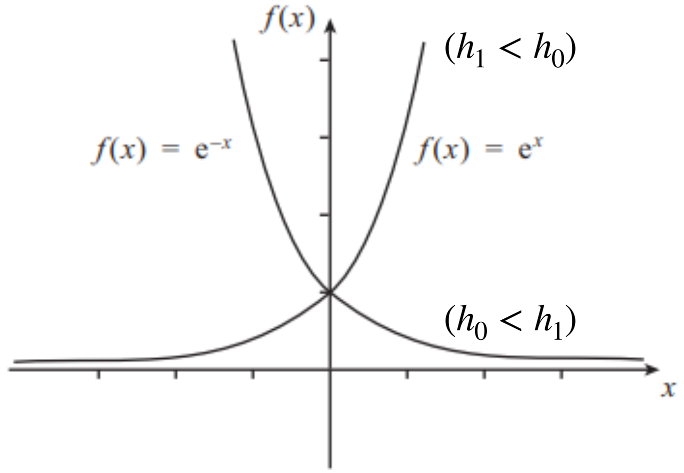
When we are minimising violated constraints, it makes sense to think of starting at the top of a ridge and climbing down into the valleys.

Simulated Annealing

- Stochastic hill climbing based on difference between evaluation of previous state (h_0) and new state (h_1).
  - If $h_1 < h_0$, definitely make the change (smaller is better)
  - Otherwise, make the change with probability

    $$e^{-(h_1 - h_0)/T}$$

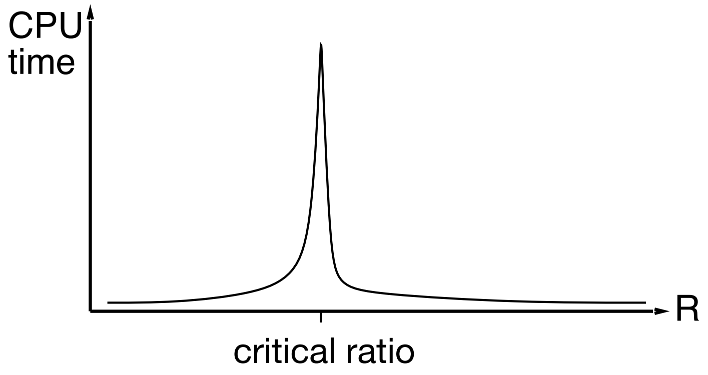
    where T is a "temperature" parameter.
- Reduces to ordinary hill climbing as $T \to 0$
- Becomes totally random search as $T \to \infty$
- Sometimes, we gradually decrease the value of T during the search



Phase Transition in CSP's

- Given random initial state, hill climbing by min-conflicts with random restarts can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000).
- In general, randomly-generated CSP's tend to be easy if there are very few or very many constraints. They become extra hard in a narrow range of the ratio

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



Summary

- CSPs are a special kind of search 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
- Iterative min-conflicts is usually effective in practice
- Simulated Annealing can help to escape from local optima