Lecture 6: Constraint Satisfaction Problems (CSPs)

CS 580 (001) - Spring 2018

Amarda Shehu

Department of Computer Science
George Mason University, Fairfax, VA, USA

February 28, 2018

Constraint Satisfaction Problems (CSPs)

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

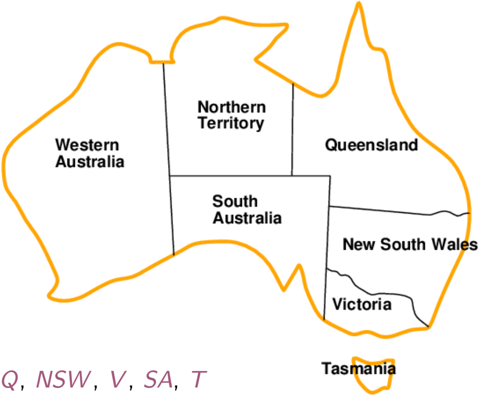
CSP:
state is defined by variables X_i with values from domain D_i
goal test is a set of constraints specifying

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

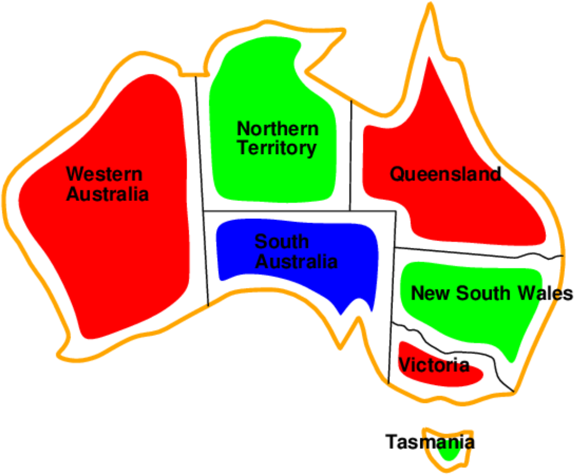


Variables WA, NT, Q, NSW, V, SA, T

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

Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Example: Map-Coloring Continued

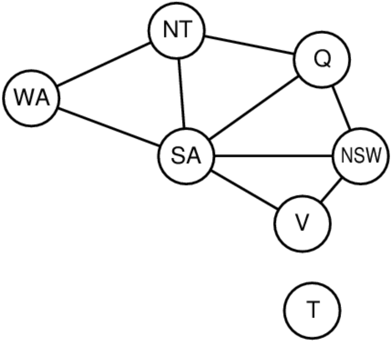


Solutions are assignments satisfying all constraints, e.g.,

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

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 $d \implies O(d^n)$ 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 polynomial time by linear programming (LP)

Varieties of Constraints

Unary constraints involve a single variable
e.g., $SA \neq green$

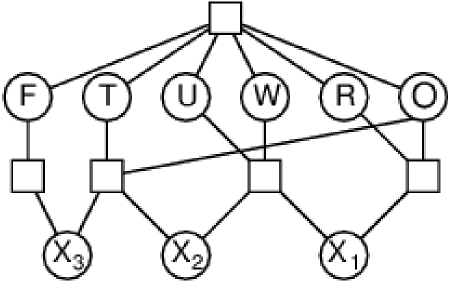
Binary constraints involve pairs of variables
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables
e.g., cryptarithmetic column constraints

Strong vs. soft constraints

Preferences (soft constraints)
e.g., *red* is better than *green*
often representable by a cost for each variable assignment
$\rightarrow$ constrained optimization problems

```
    T W O
+   T W O
---------
  F O U R
```



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

$alldiff(F, T, U, W, R, O)$

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

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

Real-world problems almost always 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, $\emptyset$
- ♢ **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!!!!

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

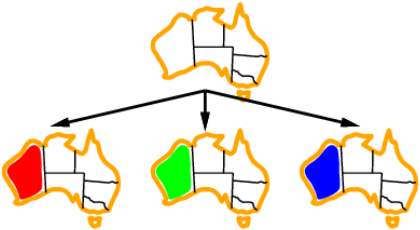
Backtracking search is the basic uninformed algorithm for CSPs

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

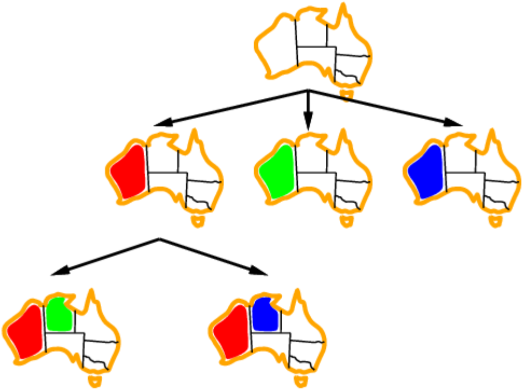
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
                add {var = value} to assignment
                inferences ← INFERENCE(var, assignment, csp)
                if inferences ≠ failure then
                    add inferences to assignment
                    result ← BACKTRACK(assignment, csp)
                    if result ≠ failure then
                        return result
        remove {var = value} and inferences from assignment
    return failure
```

Backtracking Example

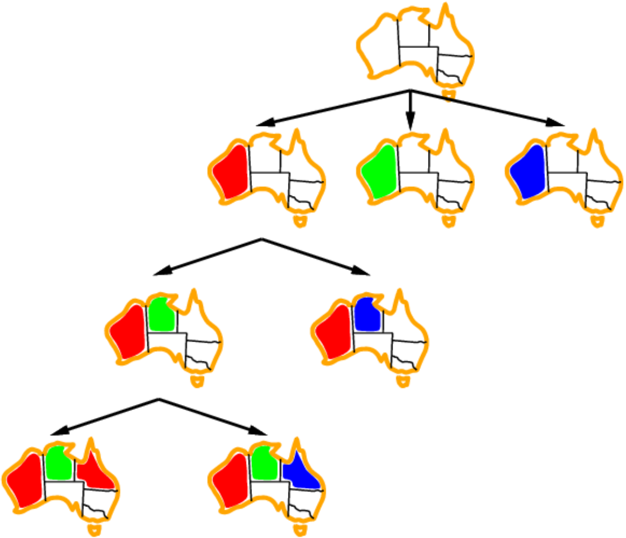
Backtracking Example



Backtracking Example



Backtracking Example



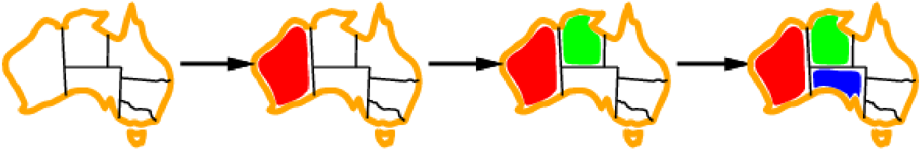
Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next? [SELECT-UNASSIGNED-VARIABLE]
2. In what order should its values be tried? [ORDER-DOMAIN-VALUES]
3. Can we detect inevitable failure early? [INFERENCE]
4. Can we take advantage of problem structure?

Minimum remaining values (MRV) for
var ← SELECT-UNASSIGNED-VAR(csp, assignment):

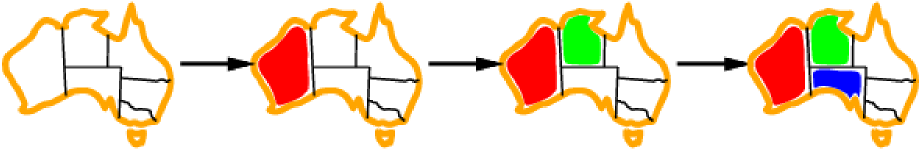
choose the variable with the fewest legal values to prune search tree
also called "most constrained variable" or "fail-first heuristic"



... but MRV heuristic does not help in selecting the first variable

Minimum remaining values (MRV) for
var ← SELECT-UNASSIGNED-VAR(csp, assignment):

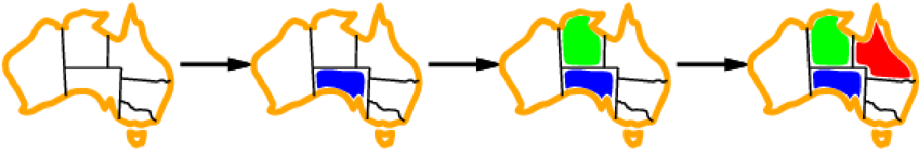
choose the variable with the fewest legal values to prune search tree
also called "most constrained variable" or "fail-first heuristic"



... but MRV heuristic does not help in selecting the first variable

Tie-breaker among MRV variables

Degree heuristic:
choose the variable with the most constraints on remaining variables



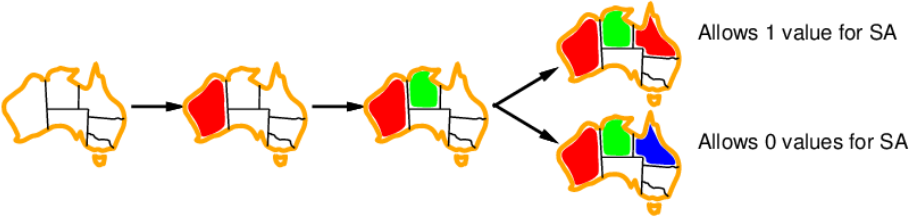
called degree heuristic because can get this information from constraint graph

attempts to reduce branching factor on future choices

Least Constraining Value Heuristic

Least Constraining Value Heuristic for:
var ← ORDER-DOMAIN-VALUES(var, assignment, csp)

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



Goal is to reach one complete assignment fast

Combining above heuristics makes 1000 queens feasible

When all solutions/complete assignments needed, LCV is irrelevant

Inference

Idea: Infer reductions in the domain of variables

When: Before and/or during the backtracking search itself

How: Constraint propagation

Algorithms: Forward Checking, AC-3

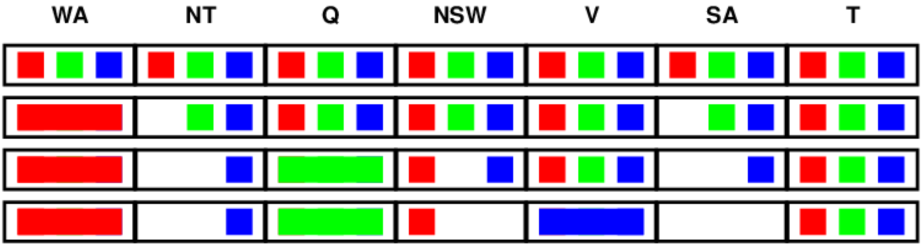
Idea: Keep track of remaining legal values for unassigned variables
Idea: Terminate search when any variable has no legal values

Forward Checking

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

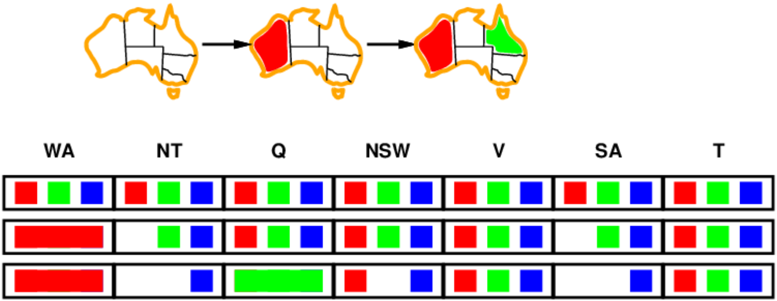
Forward Checking

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



Constraint Propagation

Forward checking propagates information from assigned to unassigned variables:



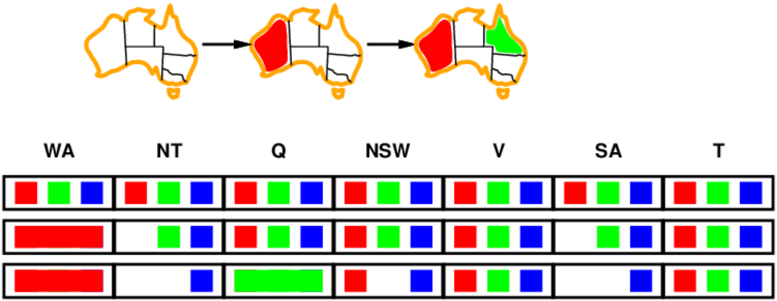
Forward checking establishes **arc consistency**

whenever a var X is assigned, domains of neighbors Y of X in constraint graph are reduced

for each unassigned var Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X

Constraint Propagation

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



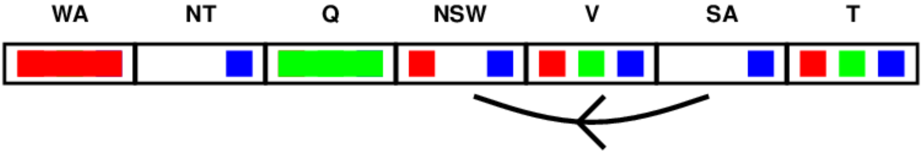
BUT: *NT* and *SA* cannot both be blue!

Constraint propagation repeatedly enforces constraints locally, and does not "chase" arc consistency

When the domain of a neighbor Y of X is reduced, domains of neighbors of Y may also become inconsistent (e.g.: *NT* and *SA*)

Simplest form of constraint propagation makes each arc consistent

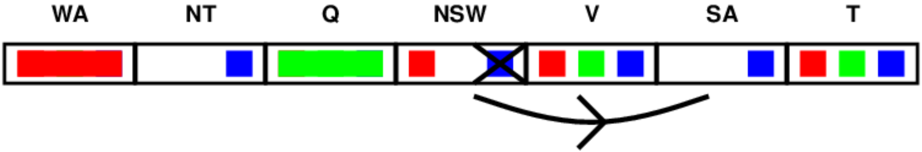
$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed value y of Y



Arc Consistency

Simplest form of propagation makes each arc consistent

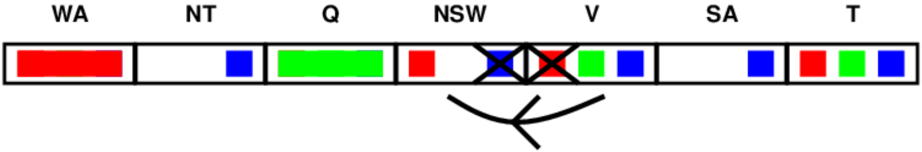
$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed value y of Y



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 value y of Y



If a variable loses a value, its neighbors in the constraint graph need to be rechecked

Maintaining Arc Consistency

More powerful idea than forward checking: If a variable loses a value, its neighbors in the constraint graph need to be rechecked

Recursively propagates constraints when changes are made to domains of variables

This recursive constraint propagation approach detects failure earlier than forward checking

Can be preprocessing or run after each assignment (INFERENCE) in the backtracking search algorithm

Algorithm: Maintaining Arc Consistency (MAC), also known as AC-3

Maintaining Arc Consistency (MAC) Algorithm

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*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*

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?

at most $deg(X_i)$

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
at most $deg(X_i)$

How many is this over all variables?

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
at most $deg(X_i)$

How many is this over all variables?
sum over all degrees is $O(E)$ of constraint graph

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
  at most $deg(X_i)$

How many is this over all variables?
  sum over all degrees is $O(E)$ of constraint graph
  which is $O(c)$

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
    at most $deg(X_i)$

How many is this over all variables?
    sum over all degrees is $O(E)$ of constraint graph
    which is $O(c)$

How often will the domain of each variable be pruned?

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
at most $deg(X_i)$

How many is this over all variables?
sum over all degrees is $O(E)$ of constraint graph
which is $O(c)$

How often will the domain of each variable be pruned?
cannot be more than the actual size of the domain

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
 at most $deg(X_i)$

How many is this over all variables?
 sum over all degrees is $O(E)$ of constraint graph
 which is $O(c)$

How often will the domain of each variable be pruned?
 cannot be more than the actual size of the domain
 ...so...$O(d)$ times

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
at most $deg(X_i)$

How many is this over all variables?
sum over all degrees is $O(E)$ of constraint graph
which is $O(c)$

How often will the domain of each variable be pruned?
cannot be more than the actual size of the domain
...so...$O(d)$ times

In total, how many arces (X_k, X_i) will be added to the queue over all variables?

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
  at most $deg(X_i)$

How many is this over all variables?
  sum over all degrees is $O(E)$ of constraint graph
  which is $O(c)$

How often will the domain of each variable be pruned?
  cannot be more than the actual size of the domain
  ...so...$O(d)$ times

In total, how many arces (X_k, X_i) will be added to the queue over all variables?
  $O(cd)$

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
 at most $deg(X_i)$

How many is this over all variables?
 sum over all degrees is $O(E)$ of constraint graph
 which is $O(c)$

How often will the domain of each variable be pruned?
 cannot be more than the actual size of the domain
 ...so...$O(d)$ times

In total, how many arces (X_k, X_i) will be added to the queue over all variables?
 $O(cd)$

How long does it take to check consistency of an arc?

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
at most $deg(X_i)$

How many is this over all variables?
sum over all degrees is $O(E)$ of constraint graph
which is $O(c)$

How often will the domain of each variable be pruned?
cannot be more than the actual size of the domain
...so...$O(d)$ times

In total, how many arces (X_k, X_i) will be added to the queue over all variables?
$O(cd)$

How long does it take to check consistency of an arc?
$O(d^2)$

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
at most $deg(X_i)$

How many is this over all variables?
sum over all degrees is $O(E)$ of constraint graph
which is $O(c)$

How often will the domain of each variable be pruned?
cannot be more than the actual size of the domain
...so...$O(d)$ times

In total, how many arces (X_k, X_i) will be added to the queue over all variables?
$O(cd)$

How long does it take to check consistency of an arc?
$O(d^2)$

So, putting it all together: $T(\mathrm{AC}-3) \in \mathrm{O}(\mathrm{cd}^3)$

Time Complexity Arc Consistency Algorithm

Given: c constraints, $\leq d$ values in the domain of each variable X_i

How many (X_k, X_i) arces will be added to the queue when pruning domain of some X_i?
at most $deg(X_i)$

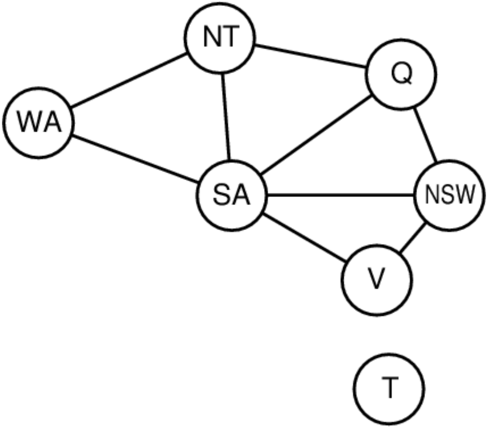
How many is this over all variables?
sum over all degrees is $O(E)$ of constraint graph
which is $O(c)$

How often will the domain of each variable be pruned?
cannot be more than the actual size of the domain
...so...$O(d)$ times

In total, how many arces (X_k, X_i) will be added to the queue over all variables?
$O(cd)$

How long does it take to check consistency of an arc?
$O(d^2)$

So, putting it all together: $T(\mathrm{AC}-3) \in \mathrm{O}(\mathrm{cd}^3)$



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

Suppose each subproblem has c variables out of n total

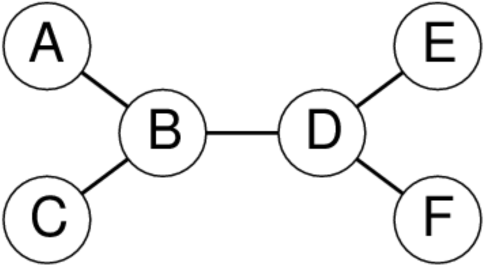
Worst-case solution cost is $n/c \cdot d^c$, **linear** in n

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

Tree-structured CSPs

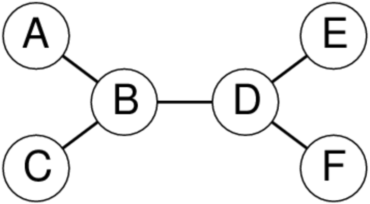


Theorem: if the constraint graph has no cycles (so, it's a tree),
the CSP can be solved in $O(n\,d^2)$ time

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.

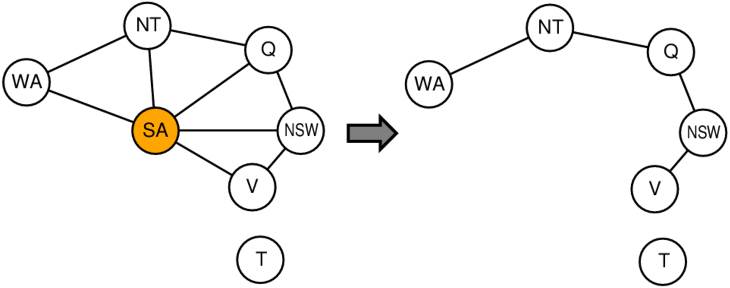
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For j from n down to 2, apply $\textsc{RemoveInconsistent}(Parent(X_j), X_j)$

3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

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

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., hill-climber with $h(n)$ = total number of violated constraints

Take-home: Propose a simple EA for 4-queens CSP

Example: 4-Queens as CSP

States:

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

Example: 4-Queens as CSP

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

Operators:

Example: 4-Queens as CSP

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

Operators: move queen in column

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

Operators: move queen in column

Goal test:

Example: 4-Queens as CSP

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

Operators: move queen in column

Goal test: no attacks

Example: 4-Queens as CSP

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

Operators: move queen in column

Goal test: no attacks

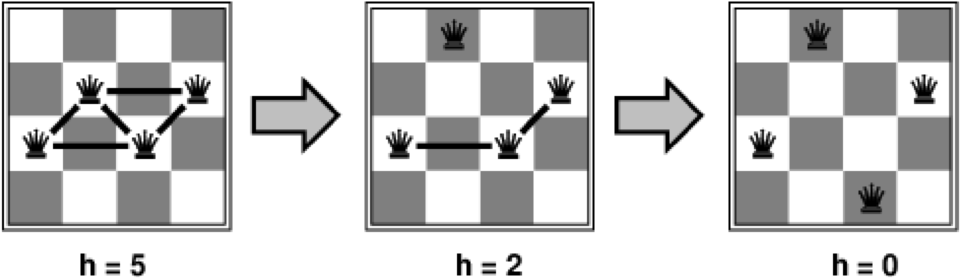
Evaluation:

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



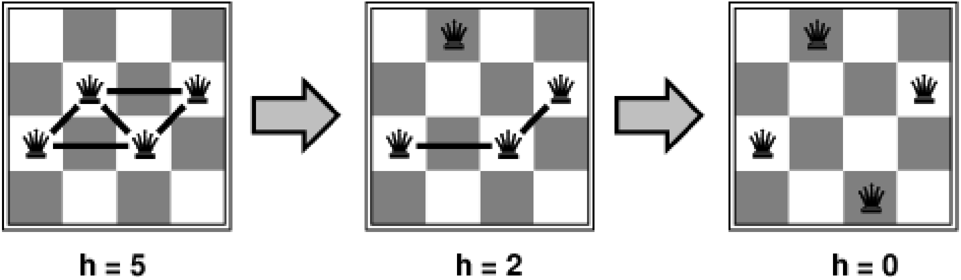
Example: 4-Queens as CSP

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

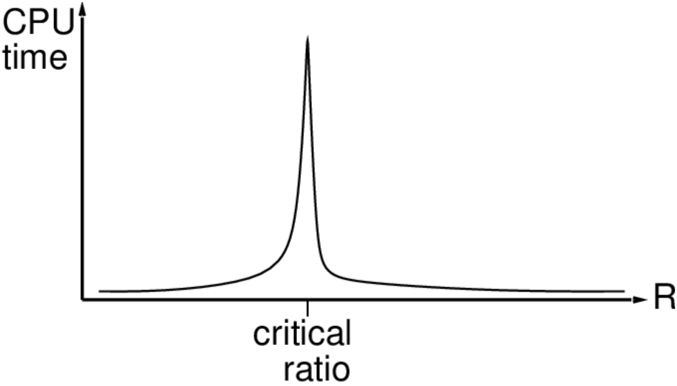


Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 10{,}000{,}000$)

The same appears to be true for any randomly-generated CSP

except in a narrow range of the ratio

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



4-Queens as a CSP

Work through the 4-queens as CSP in greater detail

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

Work through the 4-queens as CSP in greater detail

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

Variables Q_1, Q_2, Q_3, Q_4

4-Queens as a CSP

Work through the 4-queens as CSP in greater detail

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

Variables Q_1, Q_2, Q_3, Q_4

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

4-Queens as a CSP

Work through the 4-queens as CSP in greater detail

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

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)

4-Queens as a CSP

Work through the 4-queens as CSP in greater detail

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

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

Work through the 4-queens as CSP in greater detail

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

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 $(1, 3)$ $(1, 4)$ $(2, 4)$ $(3, 1)$ $(4, 1)$ $(4, 2)$

CSPs are a special kind of search problems:
  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