Constraint Satisfaction Problems

CHAPTER 5

HASSAN KHOSRAVI

SPRING2011

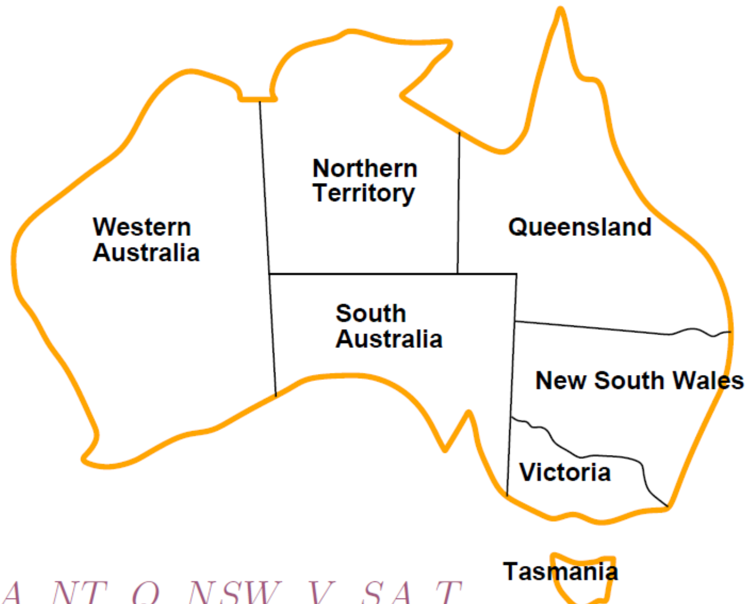
Outline

- ♦ CSP examples
- ♦ Backtracking search for CSPs
- ♦ Problem structure and problem decomposition
- ♦ Local search for CSPs

Constraint satisfaction problems (CSPs)

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

Example: Map-Coloring

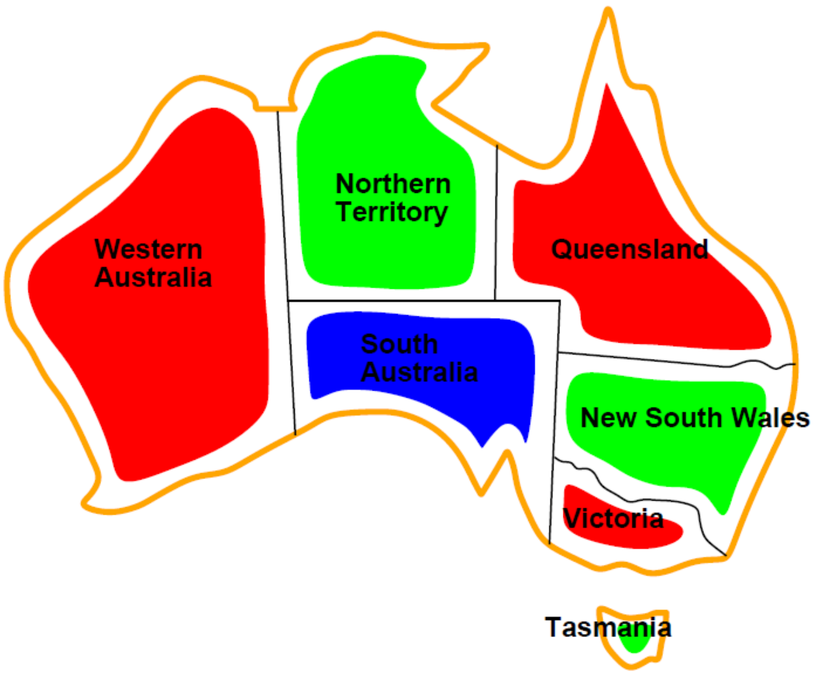


Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

CSPs (continued)

- An assignment is *complete* when every variable is mentioned.
- A *solution* to a CSP is a complete assignment that satisfies all constraints.
- Some CSPs require a solution that maximizes an *objective function*.
- Examples of Applications:
  - Airline schedules
  - Cryptography
  - Computer vision -> image interpretation
  - Scheduling your MS or PhD thesis exam ☺

Example: Map-Coloring contd.

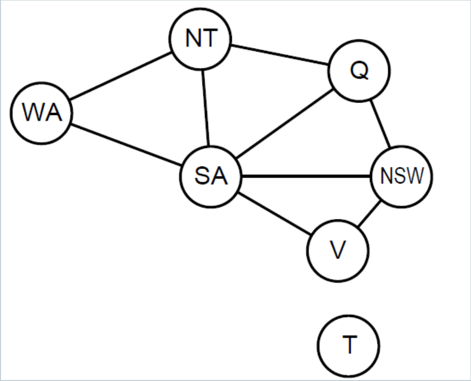


Solutions are assignments satisfying all constraints, e.g.,

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

Constraint graph

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



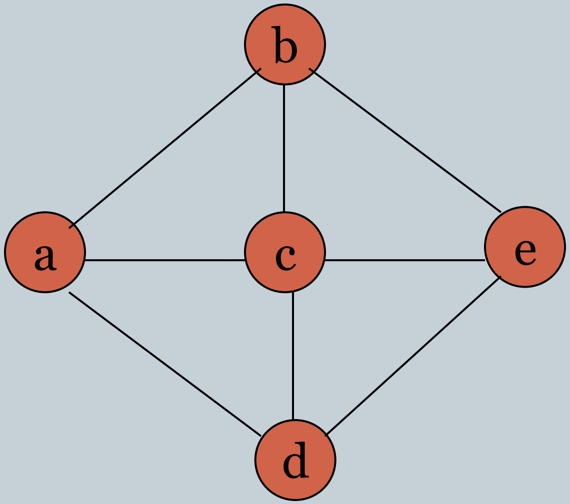
- General-purpose CSP algorithms use the graph structure
- to speed up search. E.g., Tasmania is an independent subproblem!

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA 6= green
- Binary constraints involve pairs of variables,
  - e.g., SA <> WA
- Higher-order constraints involve 3 or more variables
- Preferences (soft constraints), e.g., red is better than green

  often representable by a cost for each variable assignment
- → constrained optimization problems

Problem



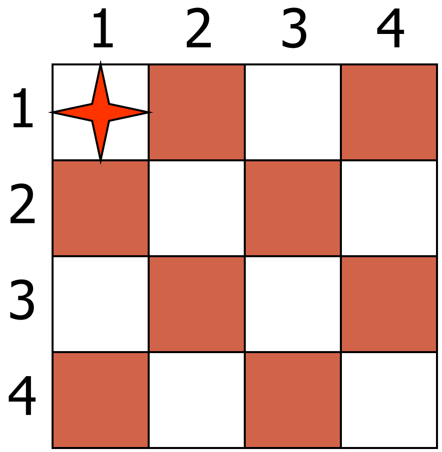
Consider the constraint graph on the right.

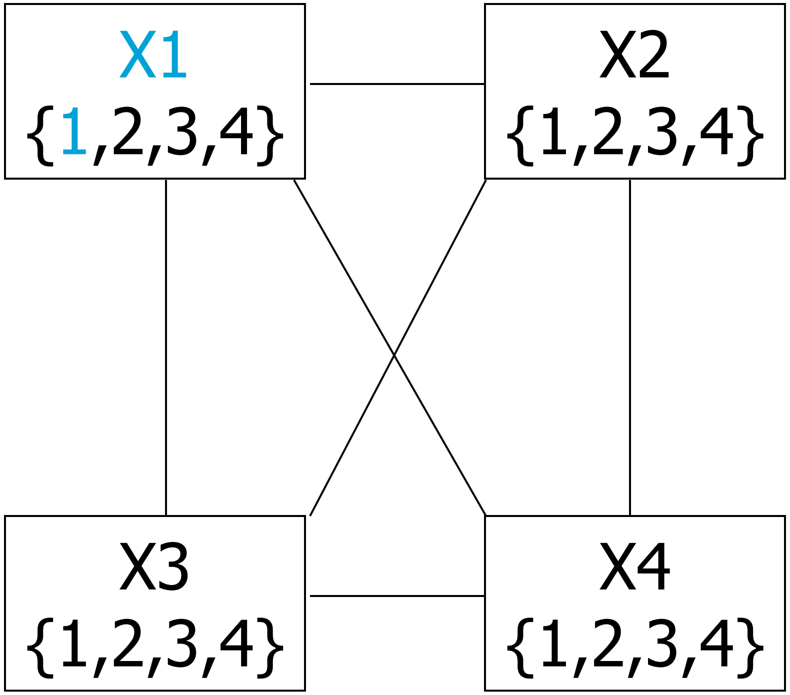
The domain for every variable is [1,2,3,4].

There are 2 unary constraints:
- variable “a” cannot take values 3 and 4.
- variable “b” cannot take value 4.

There are 8 binary constraints stating that variables connected by an edge cannot have the same value.

Example: 4-Queens Problem





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 variablethat does not conflict with current assignment.
    ⇒ fail if no legal assignments (not fixable!)
  - ♦ Goal test: the current assignment is complete

- This is the same for all CSPs!

Standard search formulation (incremental)

- Can we use breadth first search?
  - Branching factor at top level?
    - *nd* any of the d values can be assigned to any variable
  - Next level?
    - *(n-1)d*
  - *We generate $n!.d^n$ leaves even though there are d^n complete assignments. Why?*
  - *Commutatively*
    - *If the order of applications on any given set of actions has no effect on the outcome.*

Backtracking search

- Variable assignments are commutative, i.e.,
  - [WA=red then NT =green] same as [NT =green thenWA=red]
- Only need to consider assignments to a single variable at each node
  - ⇒b=d and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called backtracking search
- Is this uninformed or informed?
  - Backtracking search is the basic uninformed algorithm for CSPs

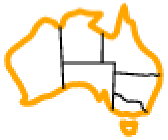
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

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
```

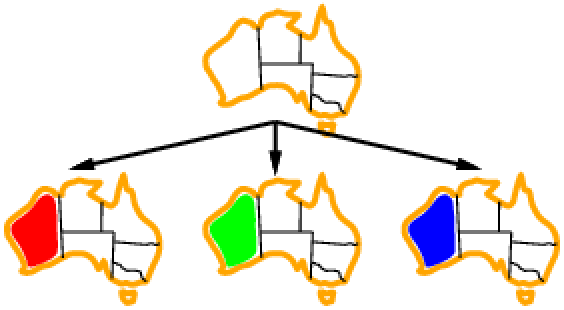
Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
-
  - Which variable should be assigned next?
  -
  - In what order should its values be tried?
  -
  - Can we detect inevitable failure early?
  - Can we take advantage of problem structure?
  -
  -

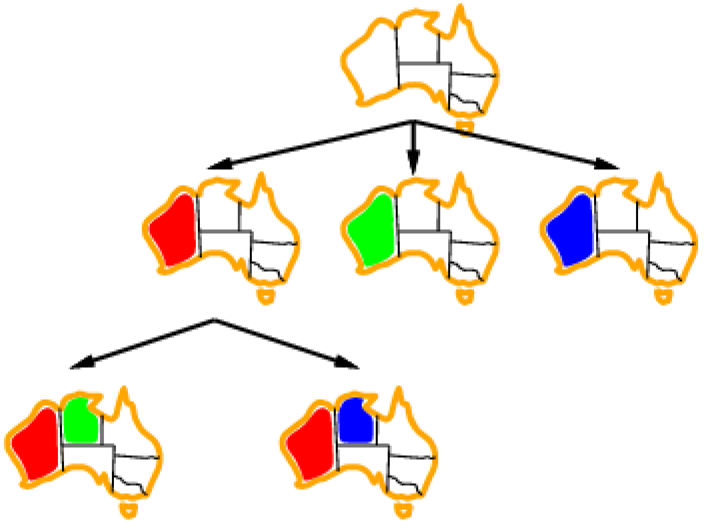
Backtracking example



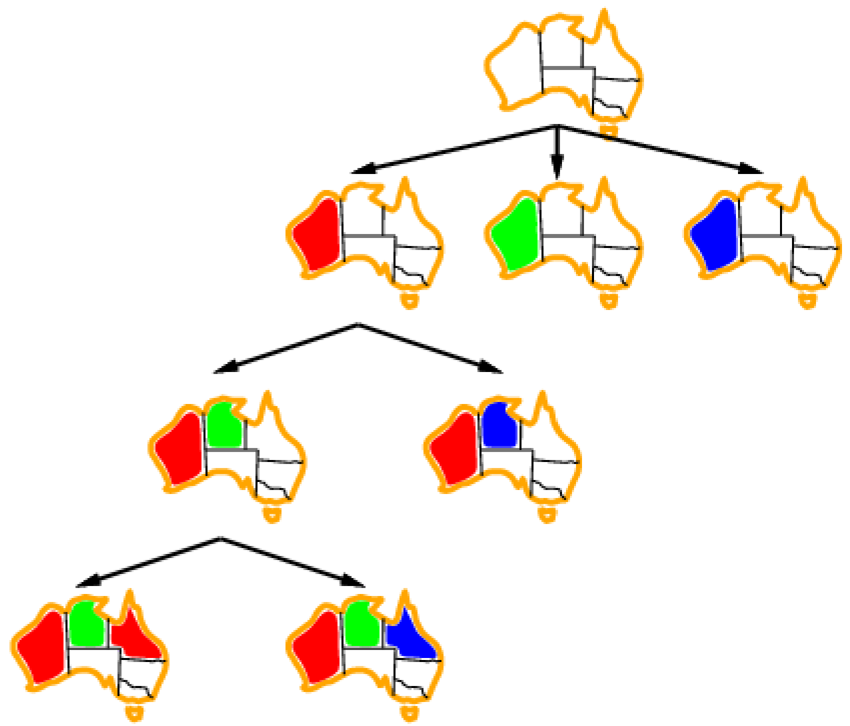
Backtracking example



Backtracking example

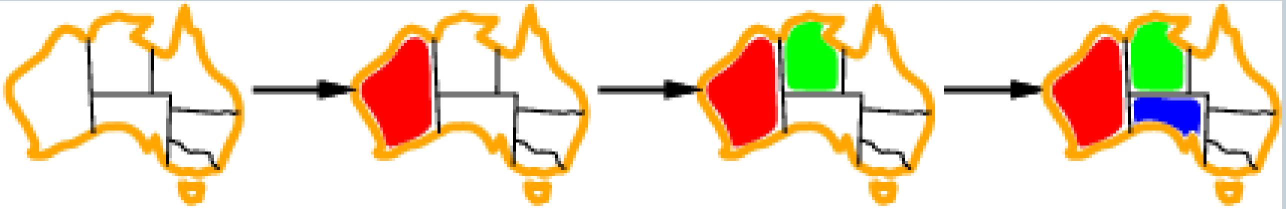


Backtracking example



Most constrained variable

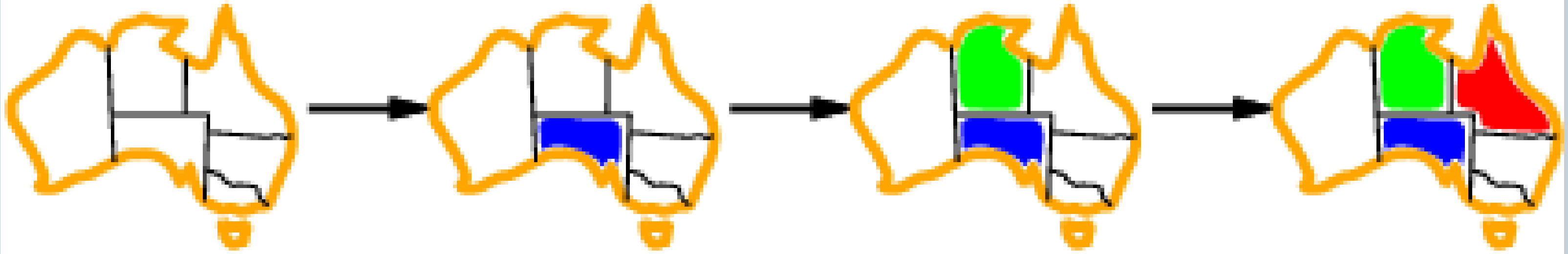
- **Most constrained variable:**
  choose the variable with the fewest legal values
  a.k.a. minimum remaining values (MRV) heuristic



Only picks a variable (Not a value)

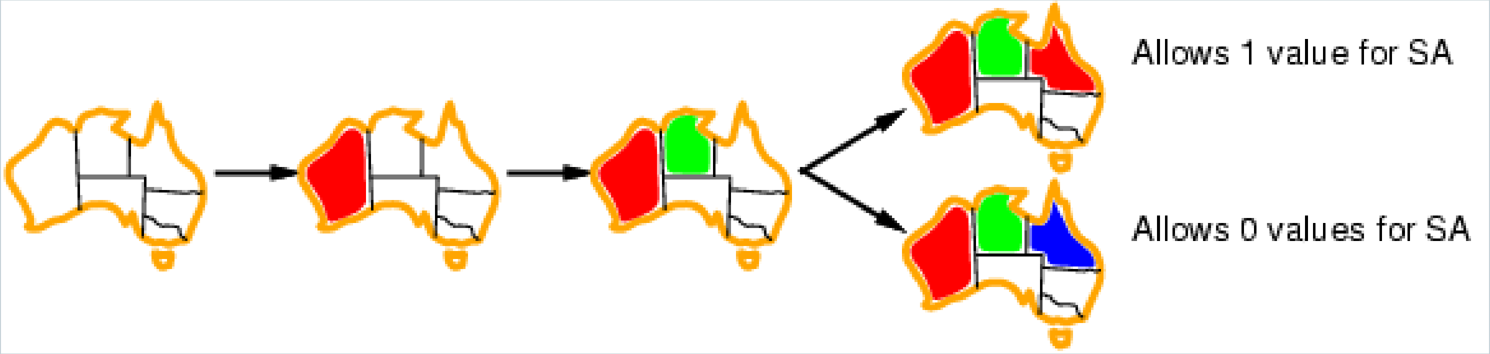
Most constraining variable

- How to choose between the variable with the fewest legal values?



- Tie-breaker among most constrained variables
  - choose the variable with the most constraints on remaining variables

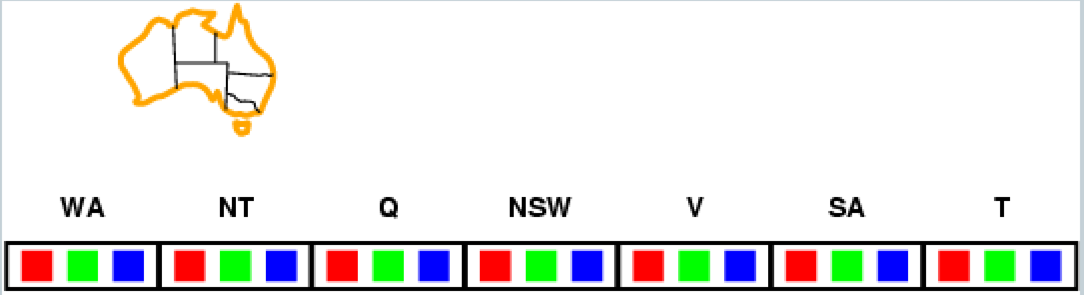
Least constraining value

- 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

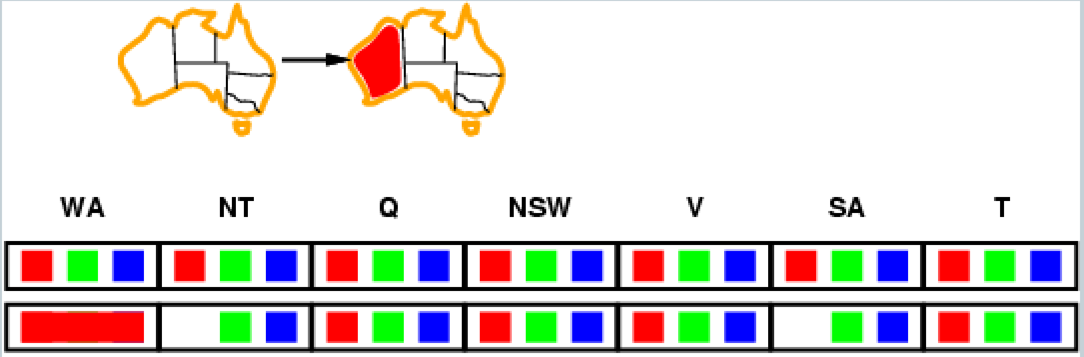
Forward checking

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



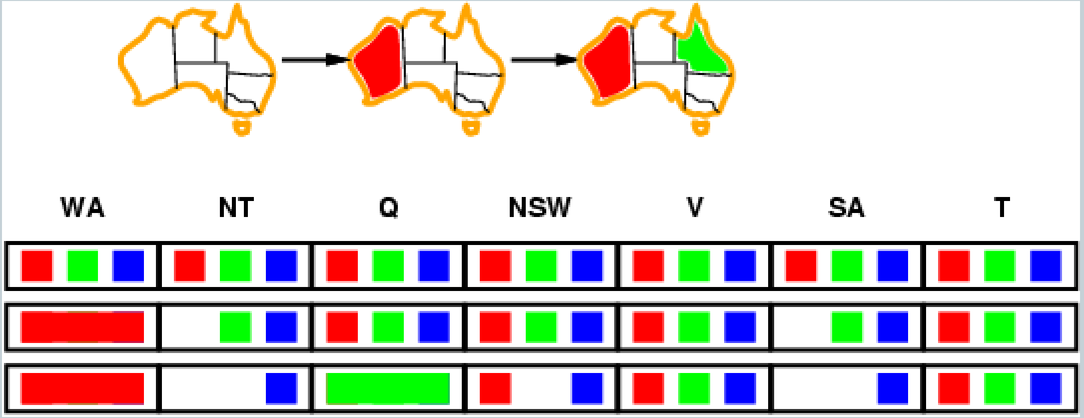
Forward checking

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



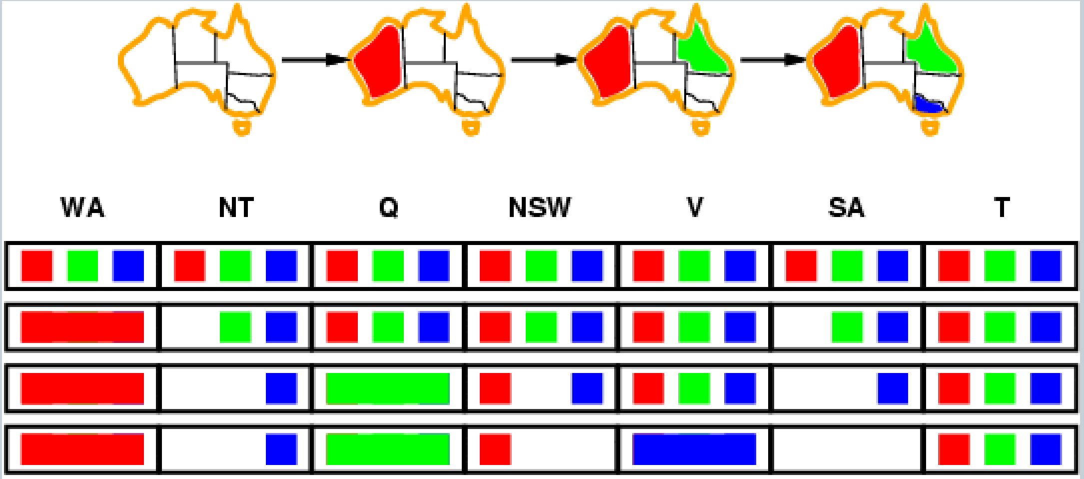
Forward checking

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

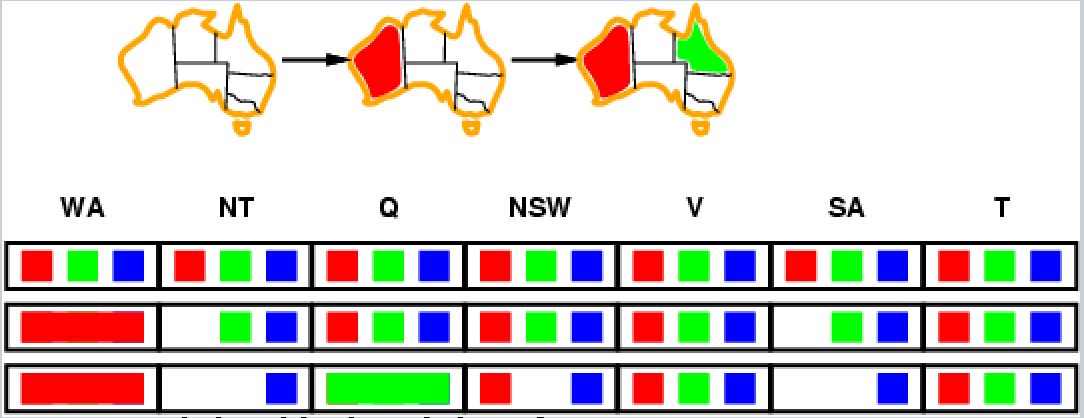


Forward checking

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

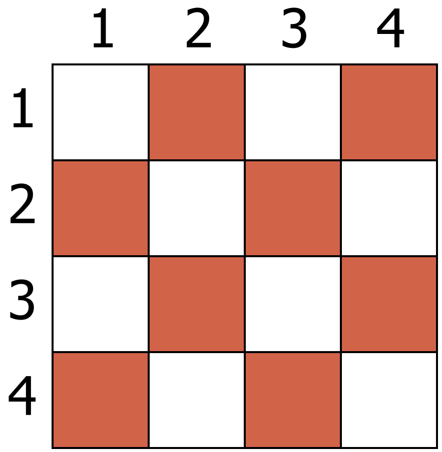


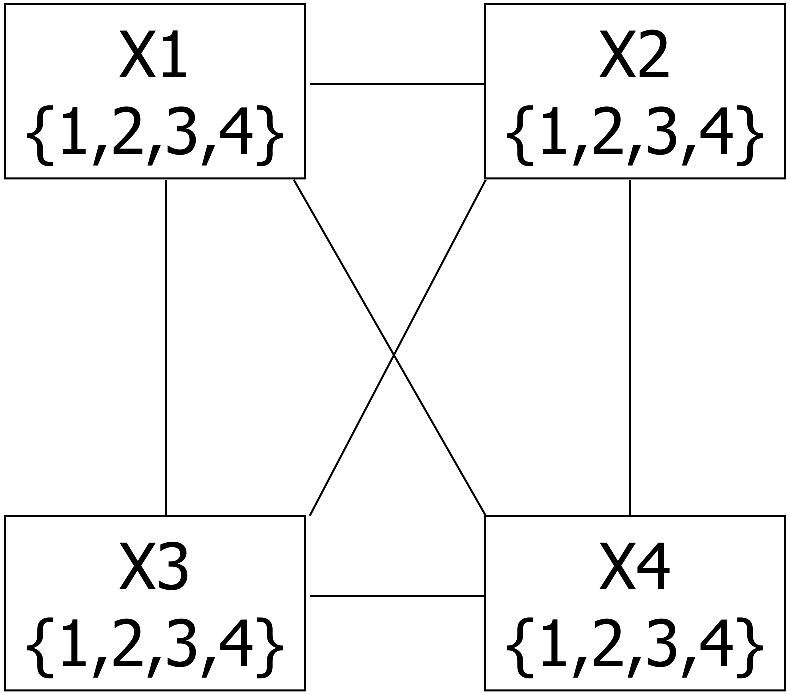
Constraint propagation

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

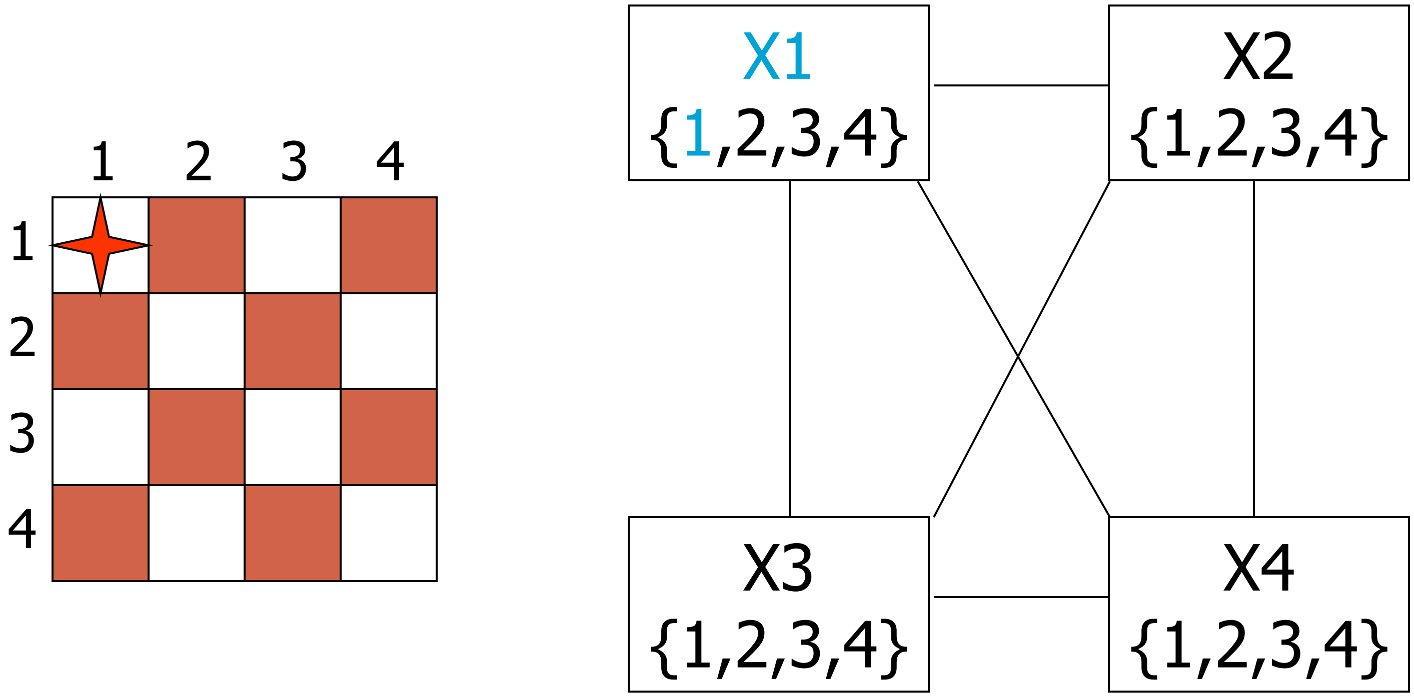
- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally. Has to be faster than searching
-

Example: 4-Queens Problem

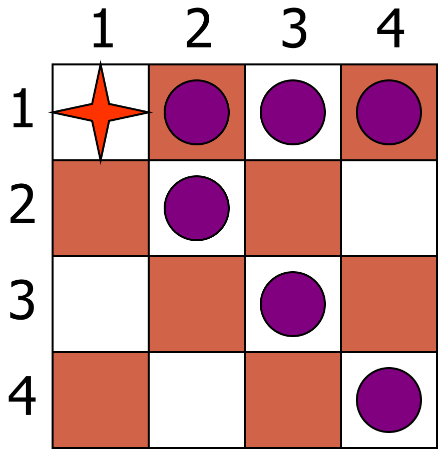


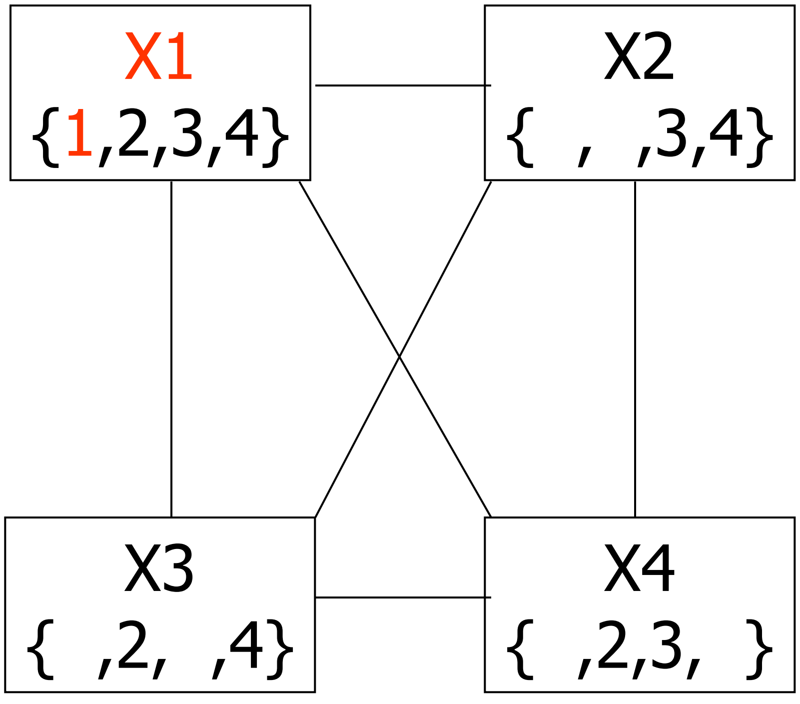


Example: 4-Queens Problem

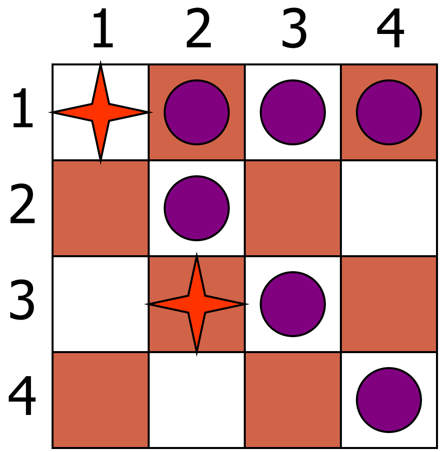


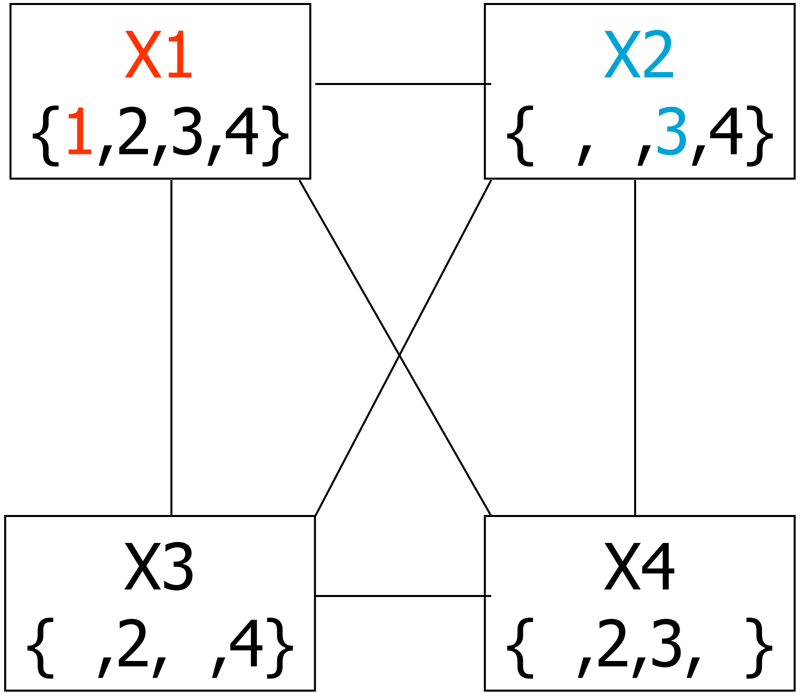
Example: 4-Queens Problem



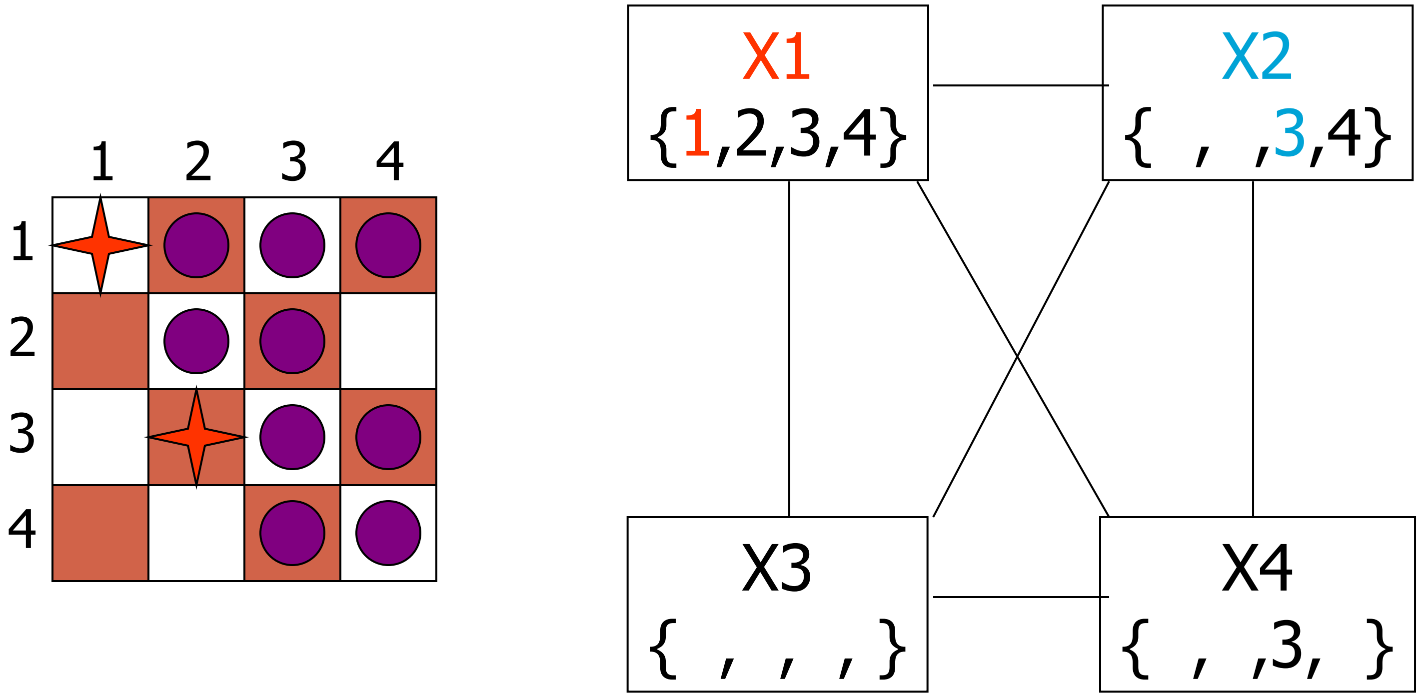


Example: 4-Queens Problem

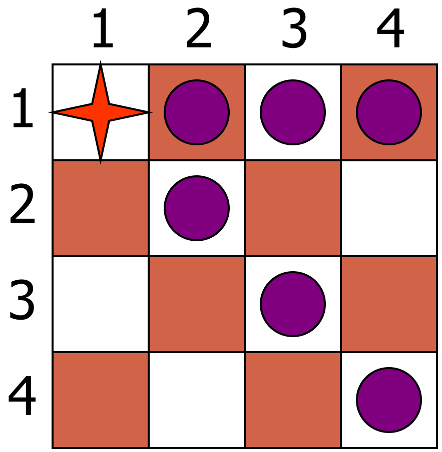


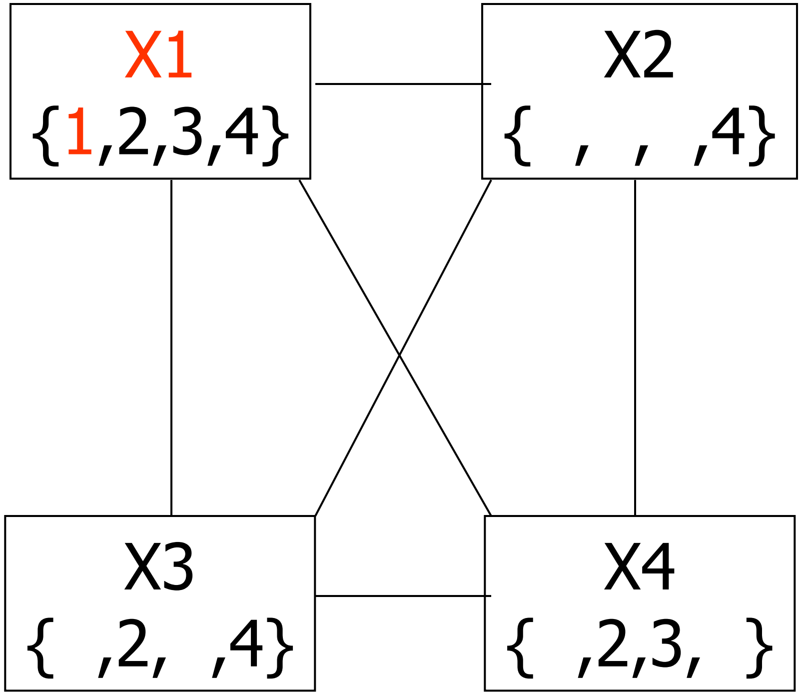


Example: 4-Queens Problem

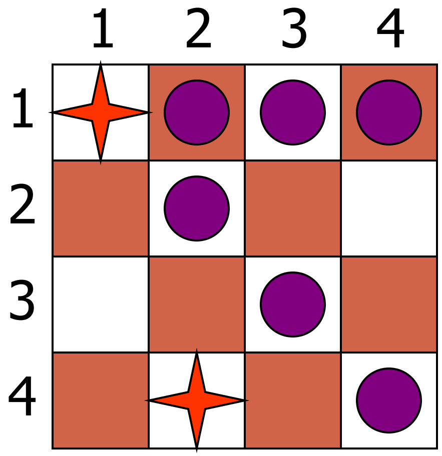


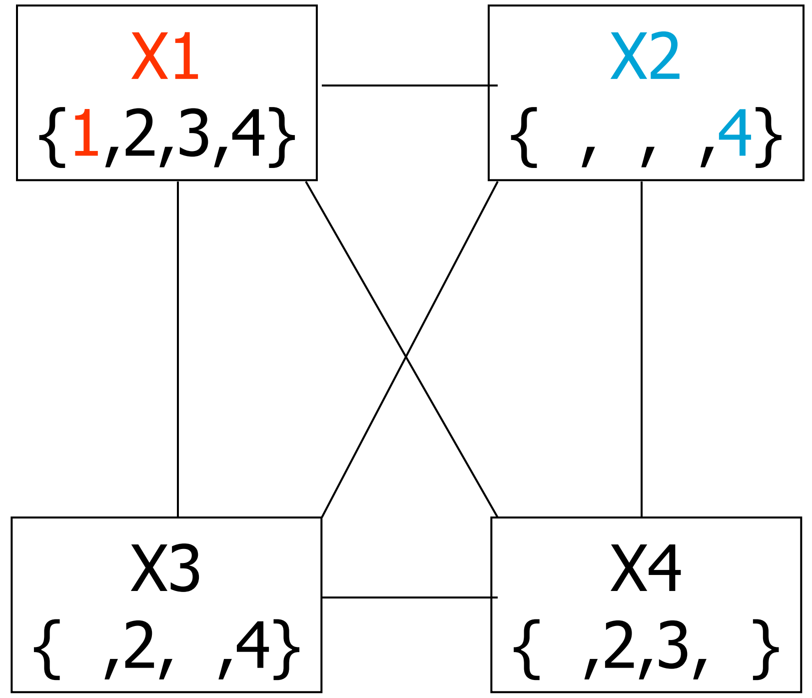
Example: 4-Queens Problem



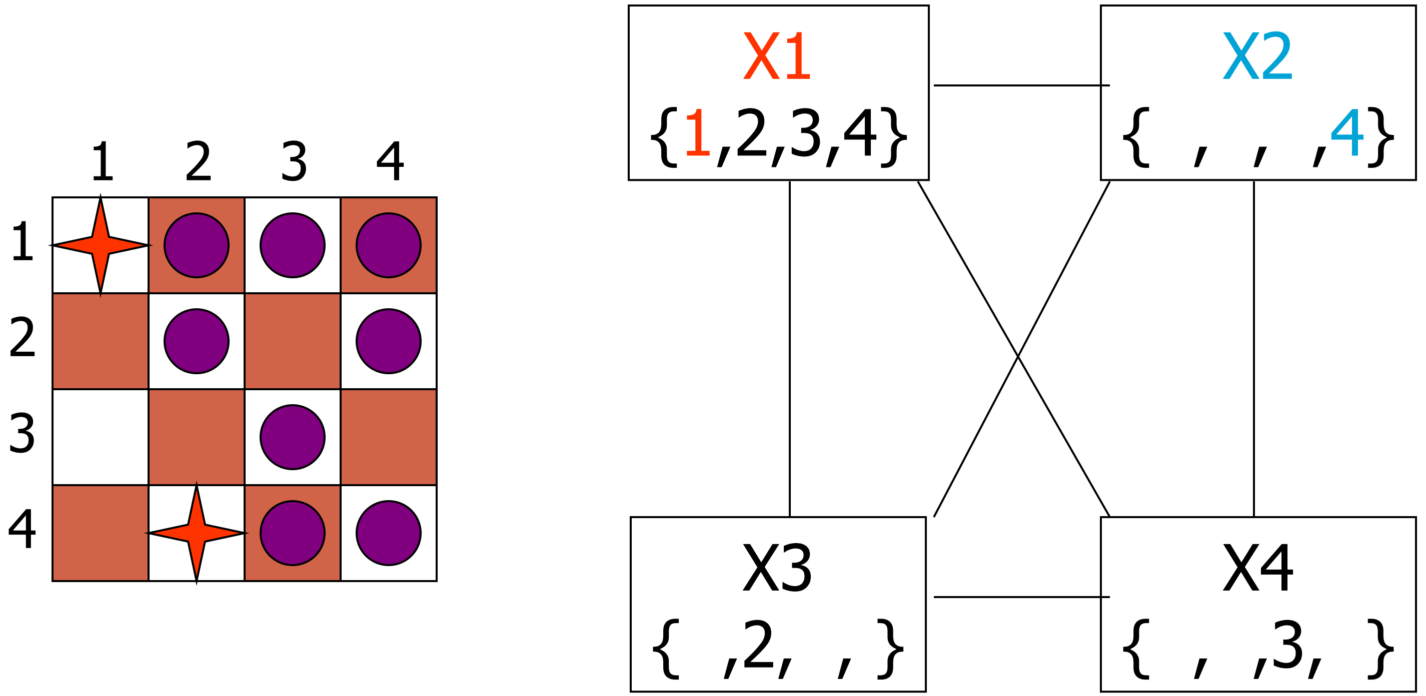


Example: 4-Queens Problem

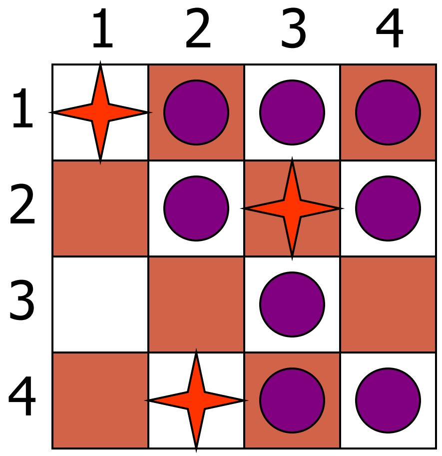


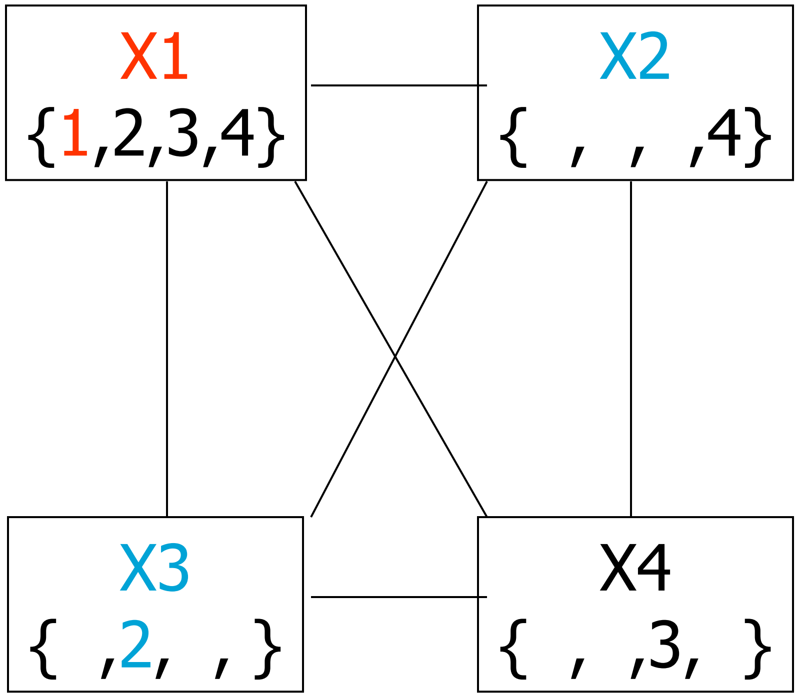


Example: 4-Queens Problem

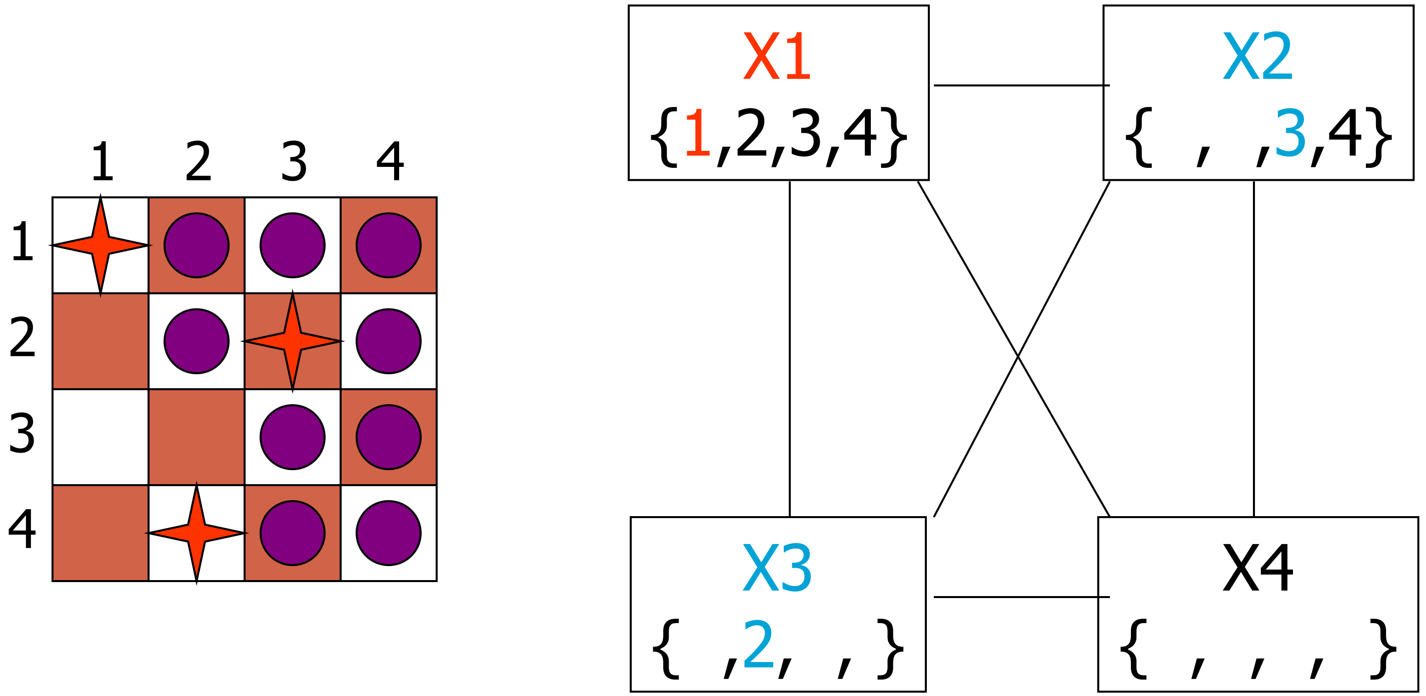


Example: 4-Queens Problem





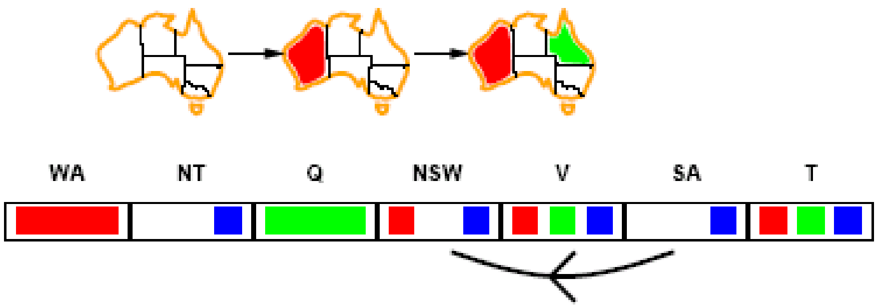
Example: 4-Queens Problem

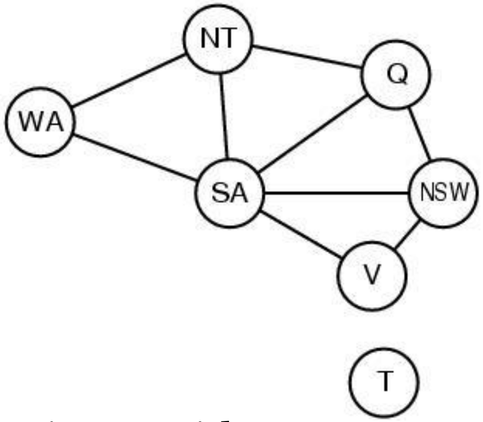


Constraint propagation

- Techniques like CP and FC are in effect eliminating parts of the search space
  - Somewhat complementary to search
- Constraint propagation goes further than FC by repeatedly enforcing constraints locally
  - Needs to be faster than actually searching to be effective
- Arc-consistency (AC) is a systematic procedure for Constraint propagation

Arc consistency





- *An Arc* $X \rightarrow Y$ is consistent if

    for *every* value x of X there is some value y consistent with x

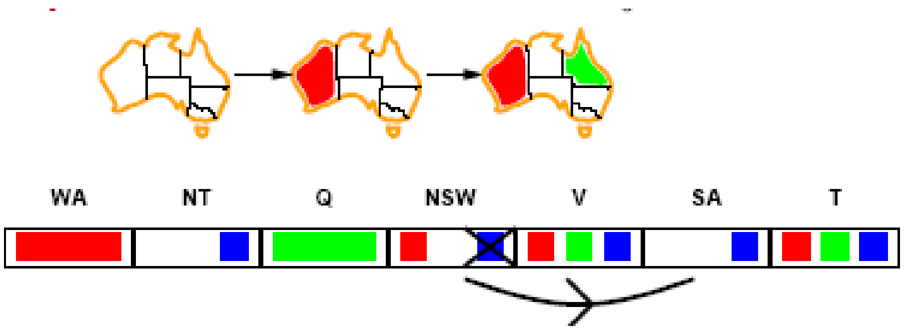
  (note that this is a directed property)

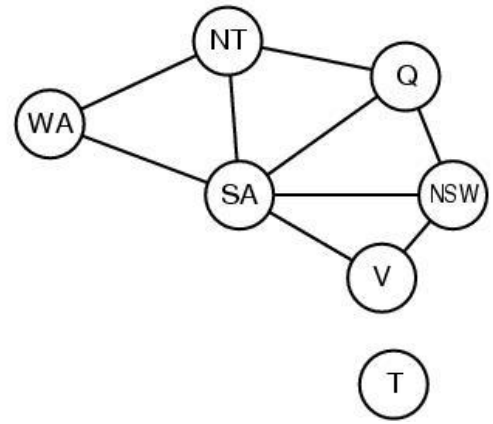
- Consider state of search after WA and Q are assigned:

$SA \rightarrow NSW$ is consistent if

SA=*blue* and NSW=*red*

Arc consistency





- $X \rightarrow Y$ is consistent if

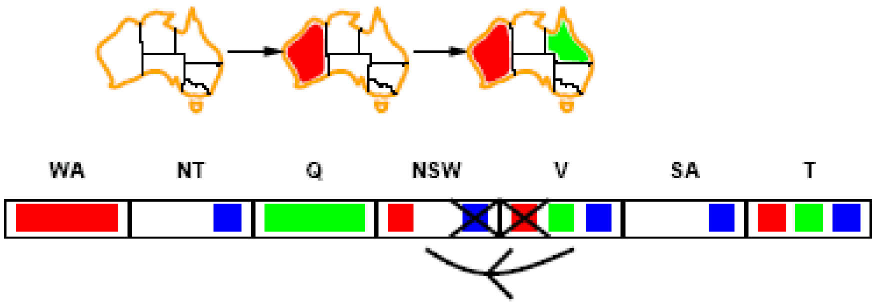
  for *every* value x of X there is some value y consistent with x

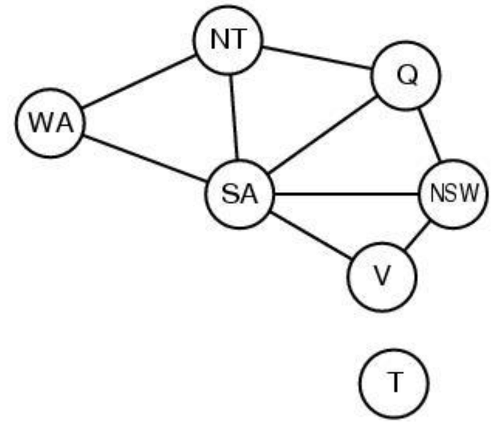
- $NSW \rightarrow SA$ is consistent if

  NSW=red and *SA=blue*

  NSW=blue and SA=???

Arc consistency



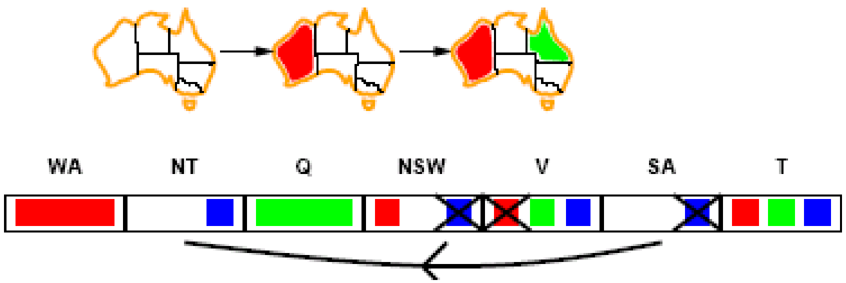


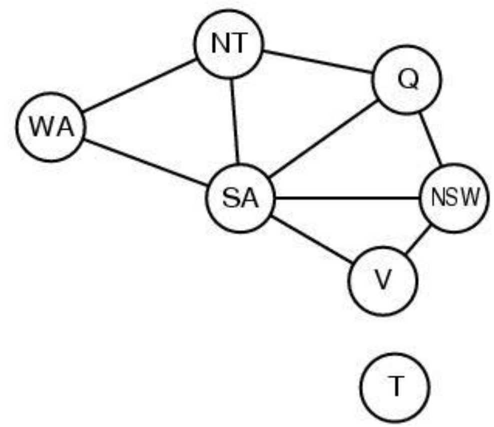
- Can enforce arc-consistency:

  Arc can be made consistent by removing *blue* from *NSW*

- Continue to propagate constraints….
  - Check $V \rightarrow NSW$
  - Not consistent for V = red
  - Remove red from *V*

Arc consistency





- Continue to propagate constraints….
- $SA \rightarrow NT$ is not consistent
  - and cannot be made consistent
- Arc consistency detects failure earlier than FC

Arc consistency checking

- Can be run as a preprocessor or after each assignment
  - Or as preprocessing before search starts
- AC must be run repeatedly until no inconsistency remains
- Trade-off
  - Requires some overhead to do, but generally more effective than direct search
  - In effect it can eliminate large (inconsistent) parts of the state space more effectively than search can
- Need a systematic method for arc-checking
  - If *X* loses a value, neighbors of *X* need to be rechecked:

Arc-consistency as message-passing

- This is a propagation algorithm. It's like sending messages to neighbors on the graph. How do we schedule these messages?

- Every time a domain changes, all incoming messages need to be re-sent. Repeat until convergence → no message will change any domains.

- Since we only remove values from domains when they can never be part of a solution, an empty domain means no solution possible at all → back out of that branch.

- Forward checking is simply sending messages into a variable that just got its value assigned. First step of arc-consistency.

Arc consistency checking

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 we remove a value
  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 between X_i and X_j
      then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

K-consistency

- Arc consistency does not detect all inconsistencies:
  - Partial assignment *{WA=red, NSW=red}* is inconsistent.

- Stronger forms of propagation can be defined using the notion of k-consistency.

- A CSP is k-consistent if for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.
  - E.g. 1-consistency = node-consistency
  - E.g. 2-consistency = arc-consistency
  - E.g. 3-consistency = path-consistency

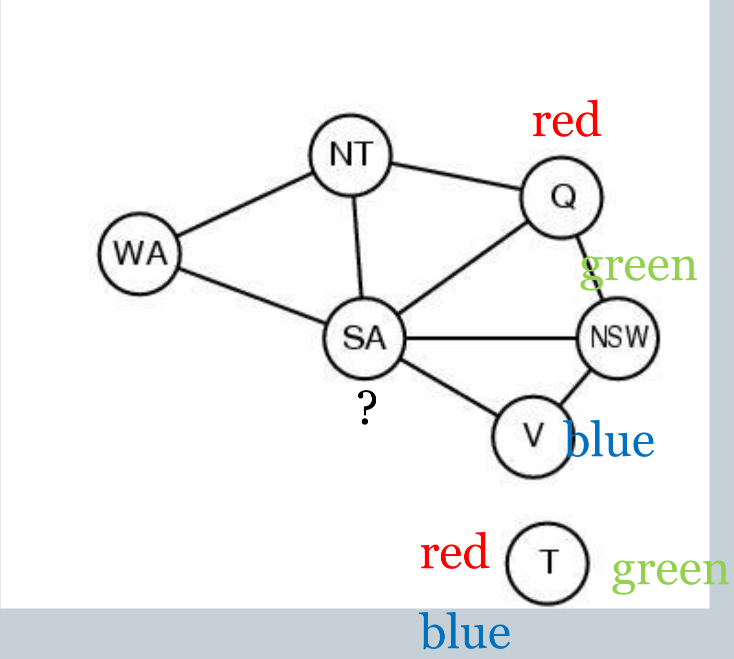
- Strongly k-consistent:
  - k-consistent for all values {k, k-1, …2, 1}

Trade-offs

- Running stronger consistency checks…
  - Takes more time
  - But will reduce branching factor and detect more inconsistent partial assignments
  - No “free lunch”
    - In worst case n-consistency takes exponential time

Back-tracking or back-jumping?

- {Q=red , NSW= green, V= blue, T=red}



Local search for CSPs

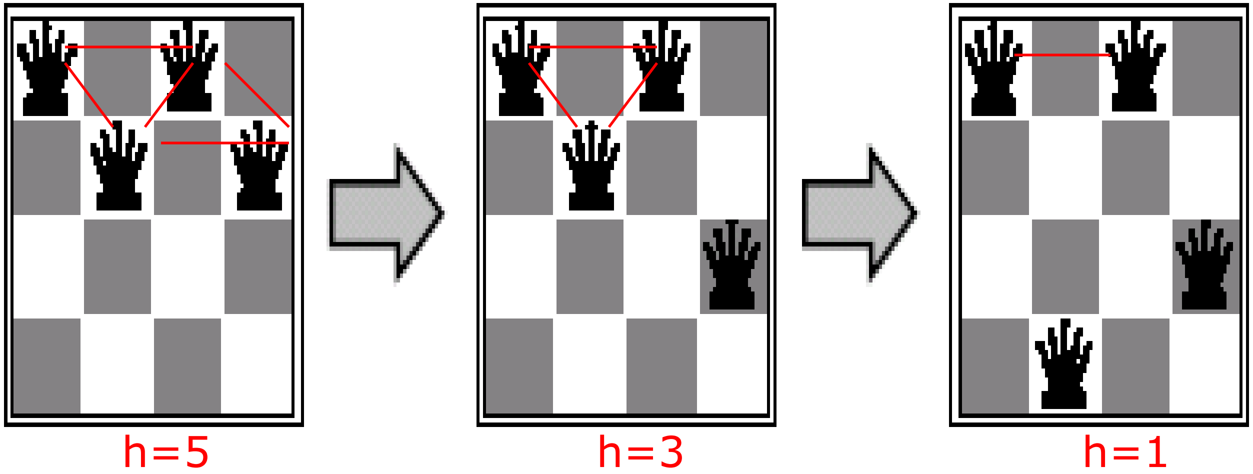
- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values
- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators **reassign** variable values
  - hill-climbing with n-queens is an example
- Variable selection: randomly select any conflicted variable
- Value selection: *min-conflicts heuristic*
  - Select new value that results in a minimum number of conflicts with the other variables

Local search for CSP

function MIN-CONFLICTS(*csp, max_steps*) **return** solution or failure
inputs: *csp*, a constraint satisfaction problem
max_steps, the number of steps allowed before giving up

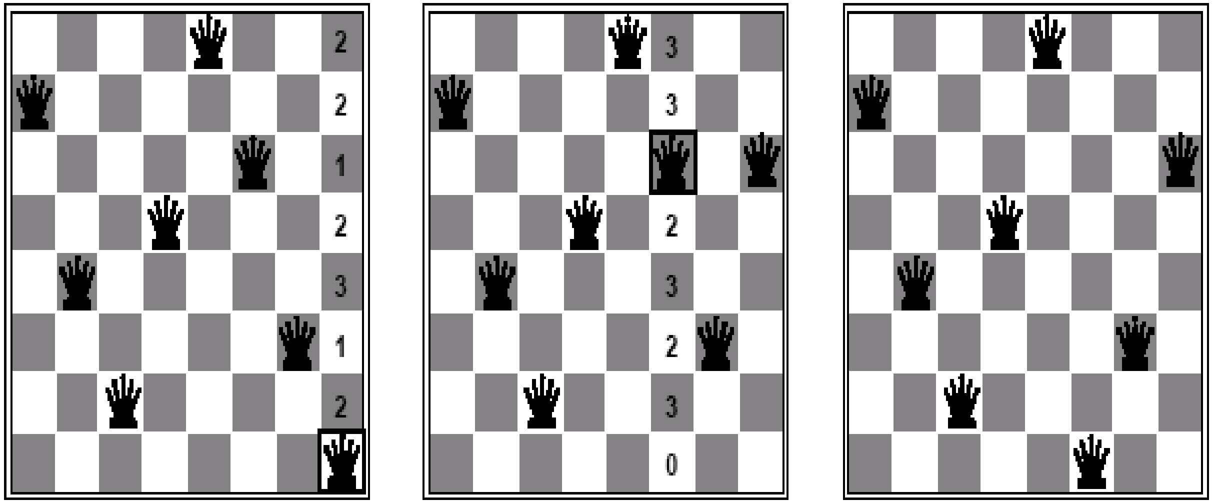
current ← an initial complete assignment for *csp*
for *i* = 1 to *max_steps* **do**
if *current* is a solution for *csp* then return *current*
var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]
value ← the value *v* for *var* that minimize CONFLICTS(*var*,*v*,*current*,*csp*)
set *var* = *value* in *current*
return *failure*

Min-conflicts example 1



Use of min-conflicts heuristic in hill-climbing.

Min-conflicts example 2



- A two-step solution for an 8-queens problem using min-conflicts heuristic
- At each stage a queen is chosen for reassignment in its column
- The algorithm moves the queen to the min-conflict square breaking ties randomly.

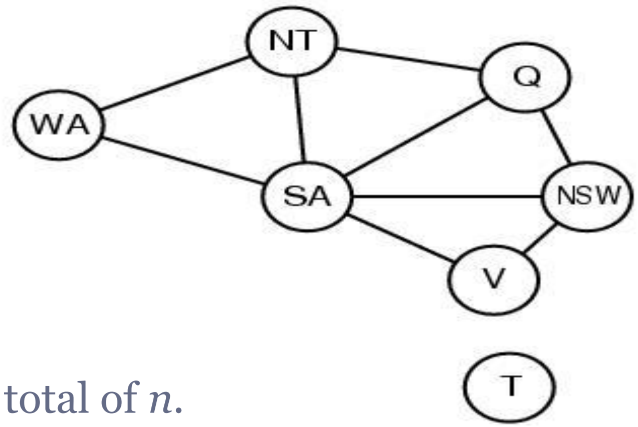
Advantages of local search

- Local search can be particularly useful in an online setting
  - Airline schedule example
    - E.g., mechanical problems require than 1 plane is taken out of service
    - Can locally search for another "close" solution in state-space
    - Much better (and faster) in practice than finding an entirely new schedule

- The runtime of min-conflicts is roughly independent of problem size.
  - Can solve the millions-queen problem in roughly 50 steps.

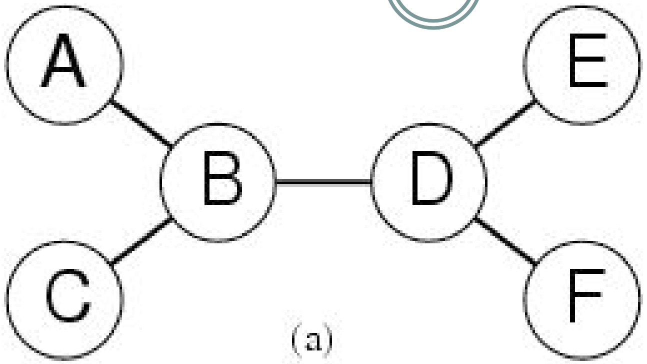
  - Why?
    - n-queens is easy for local search because of the relatively high density of solutions in state-space

Graph structure and problem complexity



- Solving disconnected subproblems
  - Suppose each subproblem has *c* variables out of a total of *n*.
  - Worst case solution cost is $O(n/c\ d^c)$, i.e. linear in *n*
    - Instead of $O(d^n)$, exponential in *n*
- E.g. $n= 80, c= 20, d=2$
  - 2^{80} = 4 billion years at 1 million nodes/sec.
  - $4 * 2^{20}$= .4 second at 1 million nodes/sec

Tree-structured CSPs



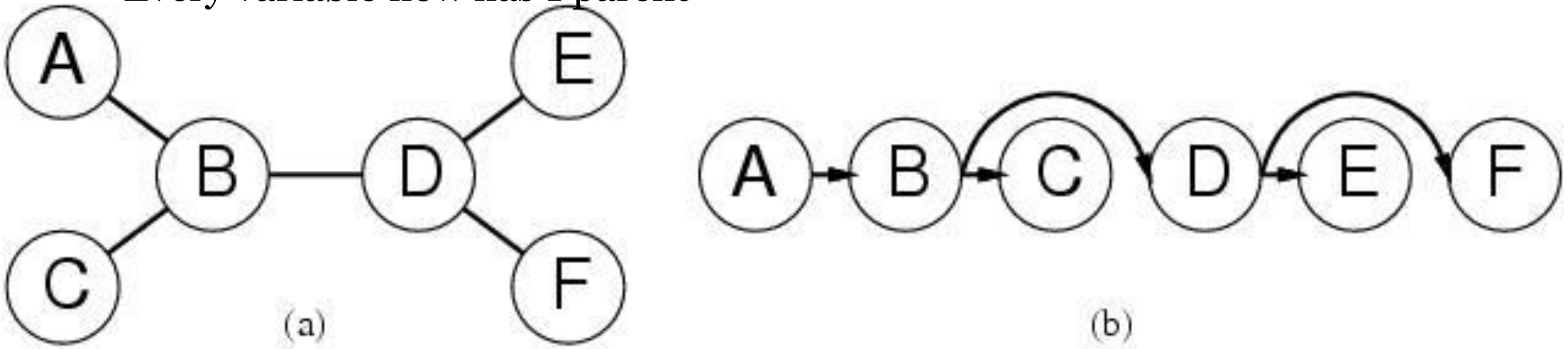
(a)

- Theorem:
  - if a constraint graph has no loops then the CSP can be solved in $O(nd^2)$ time
  - linear in the number of variables!

- Compare difference with general CSP, where worst case is $O(d^n)$

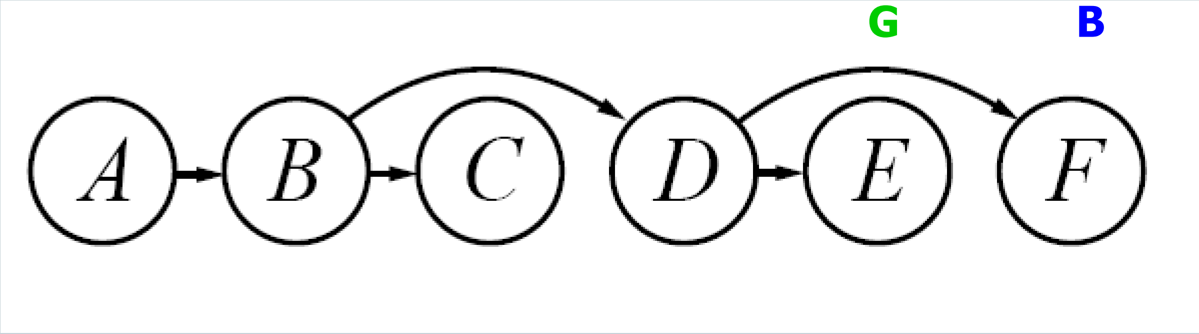
Algorithm for Solving Tree-structured CSPs

- Choose some variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering.
  - Label variables from X_1 to X_n)
  - Every variable now has 1 parent



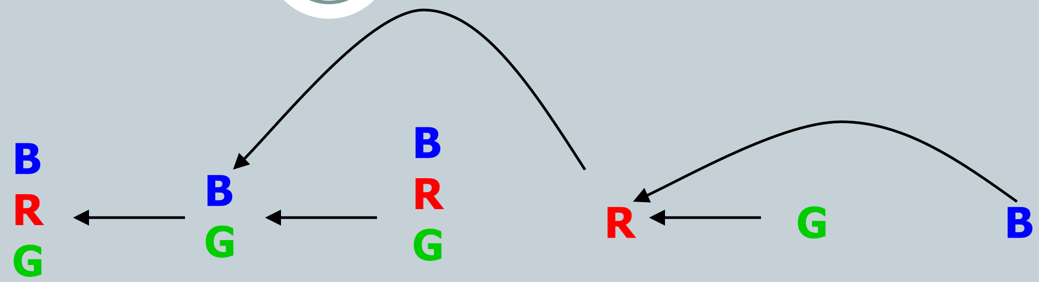
- Backward Pass
  - For j from n down to 2, apply arc consistency to arc [Parent(X_j), X_j)]
  - Remove values from Parent(X_j) if needed
- Forward Pass
  - For j from 1 to n assign X_j consistently with Parent(X_j)

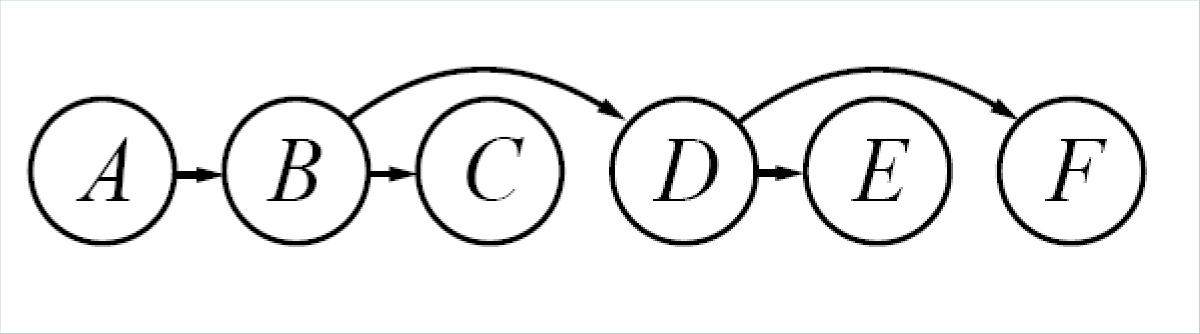
Tree CSP Example



Tree CSP Example

Backward Pass
(constraint
propagation)

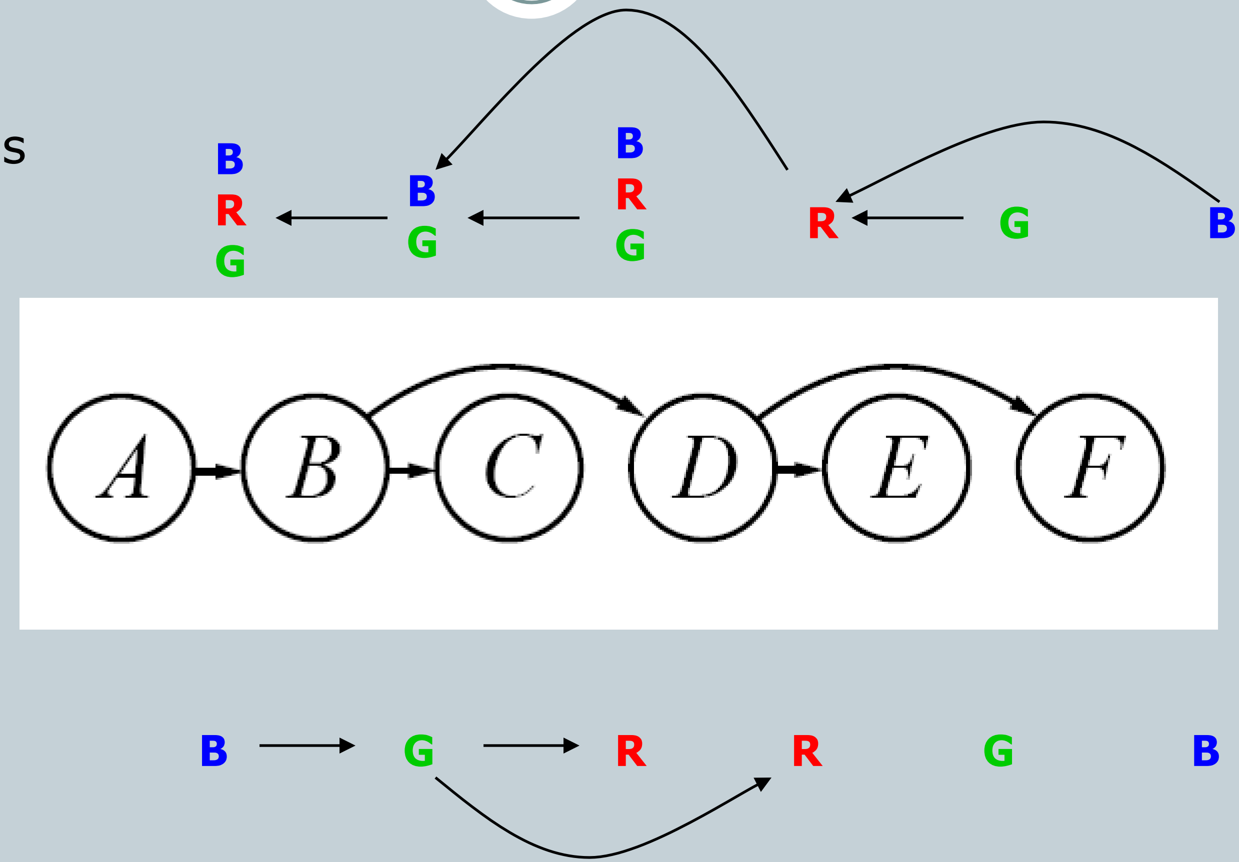




Tree CSP Example

Backward Pass (constraint propagation)

Forward Pass (assignment)



What about non-tree CSPs?

- General idea is to convert the graph to a tree

2 general approaches

1. Assign values to specific variables (Cycle Cutset method)

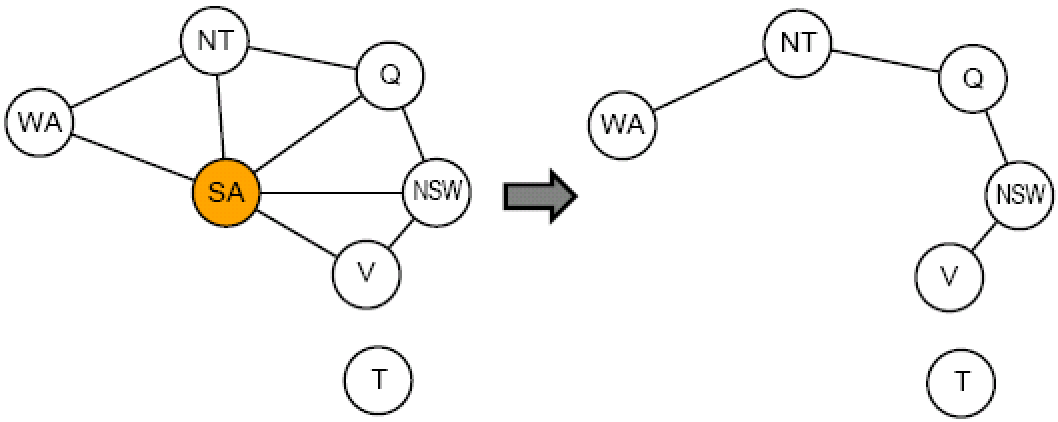
2. Construct a tree-decomposition of the graph
   - Connected subproblems (subgraphs) form a tree structure

Cycle-cutset conditioning

- Choose a subset S of variables from the graph so that graph without S is a tree
  - S = “cycle cutset”
- For each possible consistent assignment for S
  - Remove any inconsistent values from remaining variables that are inconsistent with S
  - Use tree-structured CSP to solve the remaining tree-structure
    - If it has a solution, return it along with S
    - If not, continue to try other assignments for S

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



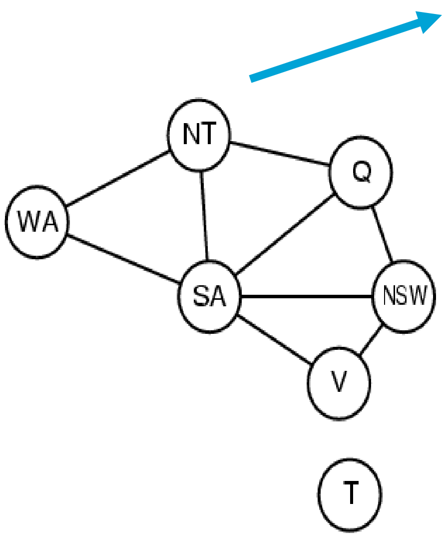
Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

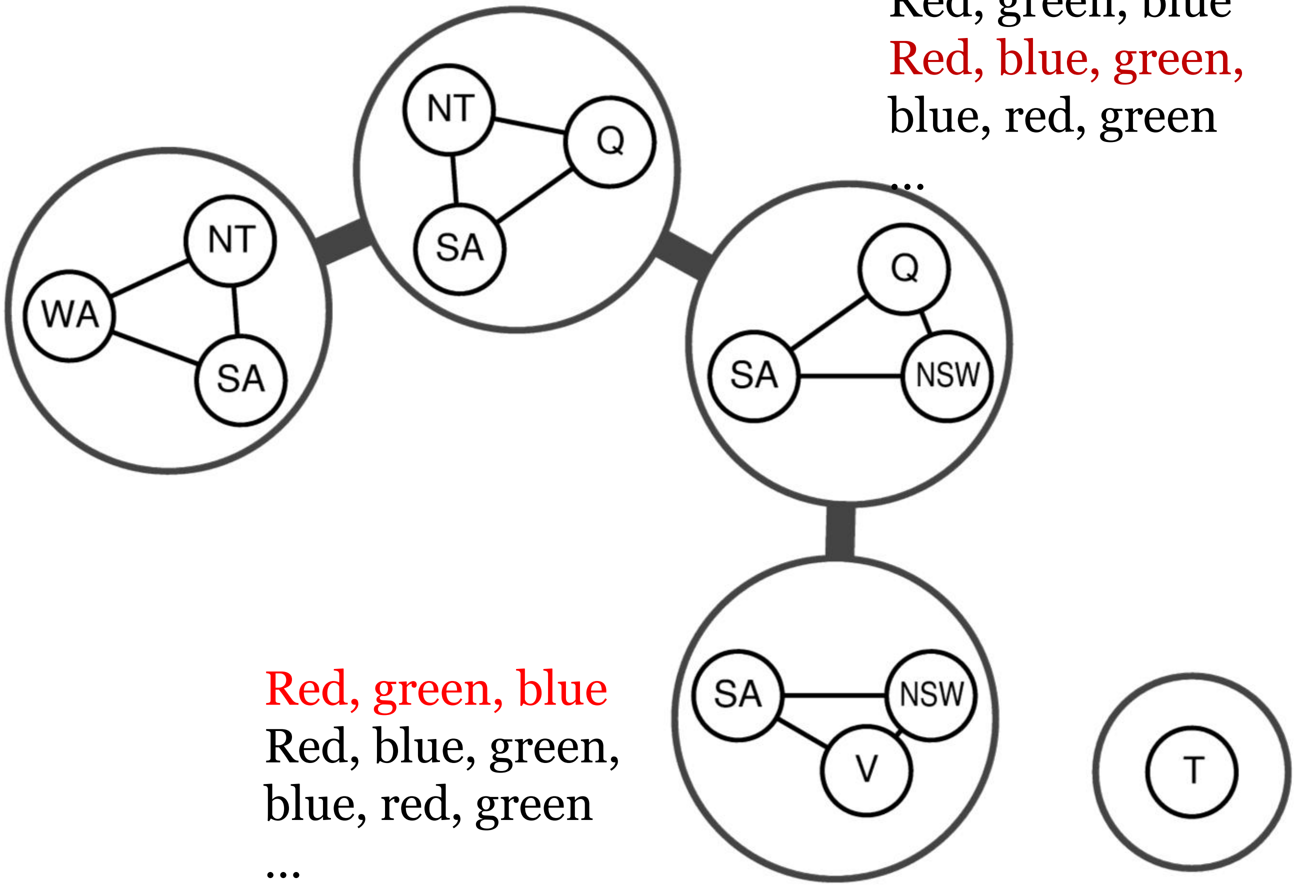
Cutset size c $\Rightarrow$ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c

Finding the optimal cutset

- If c is small, this technique works very well
- However, finding smallest cycle cutset is NP-hard
  - But there are good approximation algorithms

Tree Decompositions





Red, green, blue
Red, blue, green,
blue, red, green
...

Red, green, blue
Red, blue, green,
blue, red, green
...

Rules for a Tree Decomposition

- Every variable appears in at least one of the subproblems
- If two variables are connected in the original problem, they must appear together (with the constraint) in at least one subproblem
- If a variable appears in two subproblems, it must appear in each node on the path.

Tree Decomposition Algorithm

- View each subproblem as a “super-variable”
  - Domain = set of solutions for the subproblem
  - Obtained by running a CSP on each subproblem
  - E.g., 6 solutions for 3 fully connected variables in map problem
- Now use the tree CSP algorithm to solve the constraints connecting the subproblems
  - Declare a subproblem a root node, create tree
  - Backward and forward passes
- Example of “divide and conquer” strategy

Summary

- CSPs
  - 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
- Heuristics
  - Variable ordering and value selection heuristics help significantly
- Constraint propagation does additional work to constrain values and detect inconsistencies
  - Works effectively when combined with heuristics
- Iterative min-conflicts is often effective in practice.
- Graph structure of CSPs determines problem complexity
  - e.g., tree structured CSPs can be solved in linear time.