Artificial Intelligence

Constraint Satisfaction Problems

Dr. Bilgin Avenoğlu

Constraint Satisfaction Problems

- Up to now, we explored the idea that problems can be solved by searching the state space
- Each state is atomic, or indivisible— a black box with no internal structure.

How the Components of Agent Programs Work?

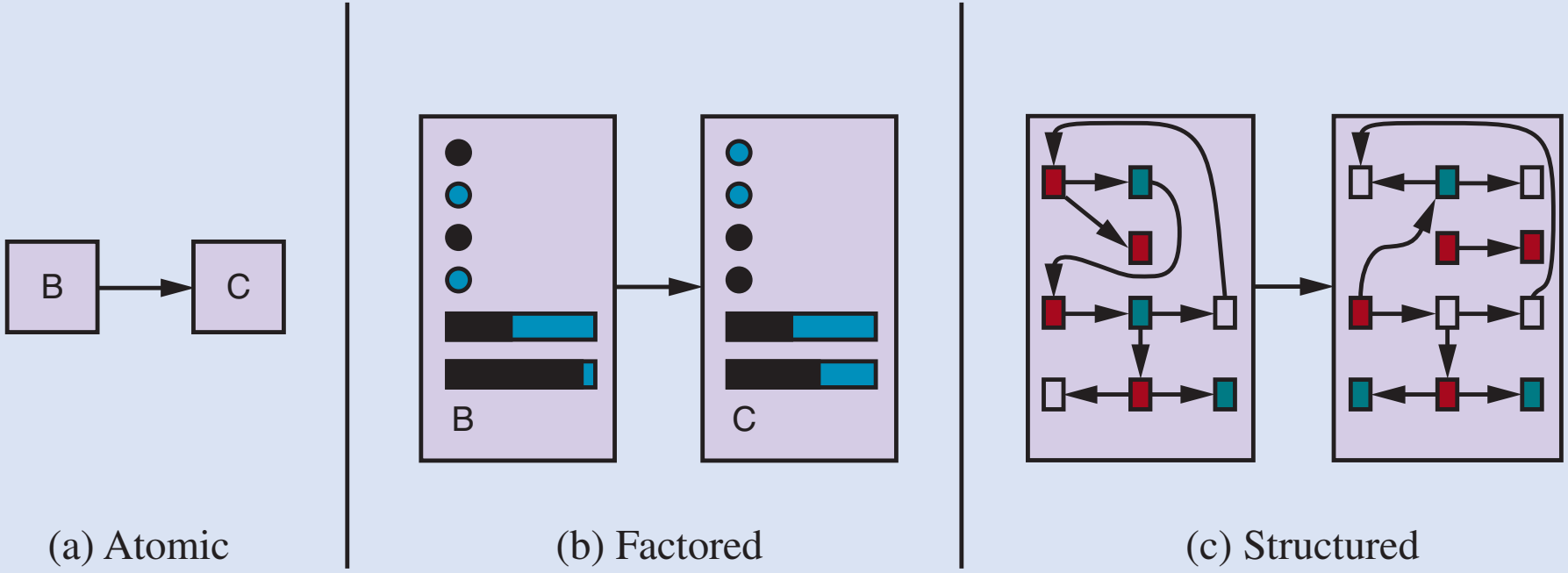


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

Constraint Satisfaction Problems

- Now, 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 - CSP.

Constraint Satisfaction Problems

- CSP search algorithms use general rather than domain-specific heuristics to enable the solution of complex problems.
  - The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.
- CSPs have the additional advantage that the actions and transition model can be deduced from the problem description.

Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components, $\mathcal{X}$, $\mathcal{D}$, and $\mathcal{C}$:
$\mathcal{X}$ is a set of variables, $\{X_1, \ldots, X_n\}$.
$\mathcal{D}$ is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.
$\mathcal{C}$ is a set of constraints that specify allowable combinations of values.

- if X_1 and X_2 both have the domain *{1,2,3}*, then the constraint saying that X_1 must be greater than X_2 can be written as
  - $\langle(X_1, X_2),\{(3,1),(3,2),(2,1)\}\rangle$ or as $\langle(X_1, X_2), X_1 > X_2\rangle$.

Defining Constraint Satisfaction Problems

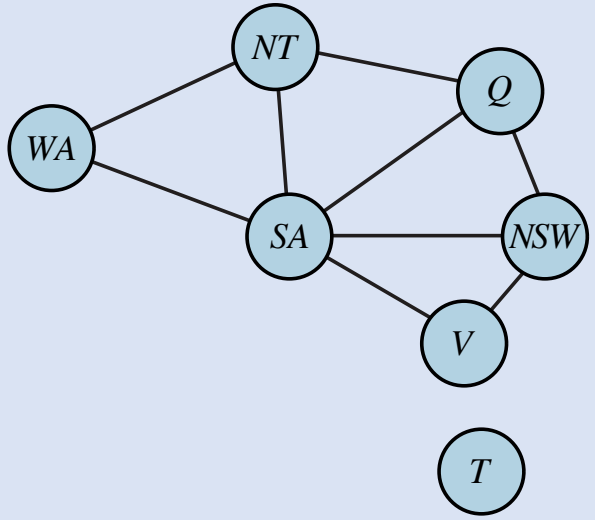
- CSPs deal with assignments of values to variables,
  - $\{X_i = v_i , X_j = v_j , \ldots\}$.
- An assignment that does not violate any constraints is called a consistent or legal assignment.
- A complete assignment is one in which every variable is assigned a value,
- A solution to a CSP is a consistent, complete assignment.
- A partial assignment is one that leaves some variables unassigned, and a partial solution is a partial assignment that is consistent.

Example problem: Map coloring

- Coloring each region either red, green, or blue
  - no two neighboring regions have the same color.
  - *X ={WA,NT,Q,NSW,V,SA,T}.*
- The domain of every variable is the set $D_i = \{red, green, blue\}$.
- The constraints require neighboring regions to have distinct colors.
  - $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$.
- SA ≠ WA is a shortcut for ⟨(SA,WA),SA ≠ WA⟩, where SA ≠ WA can be fully enumerated in turn as
  - {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)} .
- There are many possible solutions to this problem, such as
  - {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=red }.

Constraint graph

- The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



Why formulate a problem as a CSP?

- It is often easy to formulate a problem as a CSP
- CSP solvers fast and efficient.
- A CSP solver can quickly prune large swathes of the search space that an atomic state-space searcher cannot.
- In atomic state-space search we can only ask: is this specific state a goal? No? What about this one?
  - With CSPs, once we find out that a partial assignment violates a constraint, we can immediately discard further refinements of the partial assignment.

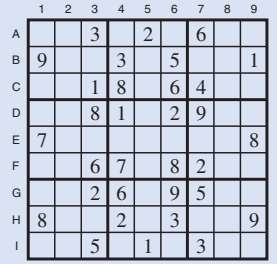
Types of variables

- CSP involves variables that have discrete, finite domains.
  - Map-coloring problem is of this kind.
  - The 8-queens problem can also be viewed as a finite-domain CSP,
    - the variables *Q1 , . . . , Q8* correspond to the queens in columns 1 to 8,
    - the domain of each variable specifies the possible row numbers for the queen in that column, $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$.
  - The constraints say that no two queens can be in the same row or diagonal.

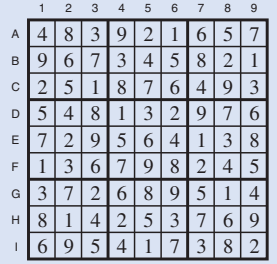
Types of constraints

- A *unary* constraint : restricts the value of a single variable.
  - South Australians won't tolerate the color green; ⟨(SA), SA ≠ green⟩.
- A *binary* constraint relates two variables.
  - SA ≠ NSW is a binary constraint.
- A binary CSP is one with only unary and binary constraints; it can be represented as a constraint graph
- We can also define higher-order constraints.
  - The ternary constraint *Between(X,Y,Z)*,
  - ⟨(X,Y,Z), X < Y < Z or X > Y > Z⟩.

Types of constraints



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		



	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

- A constraint involving an arbitrary number of variables is called a *global constraint*.
- A common global constraints: *Alldiff*
  - All of the variables involved in the constraint must have different values.
  - In Sudoku problems, all variables in a row, column, or 3×3 box must satisfy an *Alldiff* constraint.
- Every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced.
- We can transform any CSP into one with only binary constraints
  - certainly makes the life of the algorithm designer simpler.

Constraint Propagation: Inference in CSPs

- Node consistency
  - A single variable (a node in the CSP graph) is node-consistent if all the values in the variable's domain satisfy the variable's unary constraints.
    - South Australians dislike green, the variable *SA* starts with domain *{red,green,blue}*, and we can make it node consistent by eliminating green, leaving SA with the reduced domain *{red,blue}*.
  - We say that a graph is node-consistent if every variable in the graph is node-consistent.

Arc consistency

- A variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints.
- X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i,X_j).
- A graph is arc-consistent if every variable is arc-consistent with every other variable.
  - For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of decimal digits. We can write this constraint explicitly as
    - ⟨(X,Y),{(0,0),(1,1),(2,4),(3,9)}⟩.
- To make X arc-consistent with respect to Y, we reduce X's domain to $\{0, 1, 2, 3\}$.
- If we also make Y arc-consistent with respect to X , then Y's domain becomes $\{0, 1, 4, 9\}$, and the whole CSP is arc-consistent.

Arc consistency example

- For $x \in [2 \ldots 6]$, $y \in [3 \ldots 7]$ the constraint

$$x < y$$

  is arc-consistent.

- For example
  - if $x = 2$ then there is a solution
  - if $x = 6$ then y must be 7
  - if $y = 3$ then x must be 2.

- For $x \in [2..7]$ and $y \in [3..7]$ the constraint:

$$x < y$$

  is not arc consistent.

- If $x = 7$ (allowed by the domains) then there is no value for y satisfying the constraint.

AC-3 algorithm

- AC-3 maintains a queue of arcs to consider.
- Initially, the queue contains all the arcs in the CSP.
  - Each binary constraint becomes two arcs, one in each direction.
- AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue and makes X_i arc-consistent with respect to X_j.
  - If this leaves D_i unchanged, the algorithm just moves on to the next arc.
  - If this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i.
  - The change in D_i might enable further reductions in D_k, even if we have previously considered X_k.
  - If D_i is revised down to nothing, then we know the whole CSP has no consistent solution.
- Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue.
- We are left with a CSP that is equivalent to the original CSP
  - they both have the same solutions, but the arc-consistent CSP will be faster to search because its variables have smaller domains.
- In some cases, it solves the problem completely (by reducing every domain to size 1) and in others it proves that no solution exists (by reducing some domain to size 0).

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Arc consistency (AC-3)

- Example
  - https://www.boristhebrave.com/2021/08/30/arc-consistency-explained/

- Start with a worklist containing two arcs for each constraint.
- While the worklist is non-empty:
  - Remove an arc from the worklist, say from x to y.
  - For each value in the domain of y:
    - Search for a support of that value on the constraint of the arc
      - If no support is found:
        - Remove the value from the domain of the variable
        - Add arcs from y to any other variables it's constrainted with (excluding x).

Complexity of AC-3

- Assume a CSP with *n* variables, each with domain size at most *d*, and with *c* binary constraints (arcs).
- Each arc (X_k, X_i) can be inserted in the queue only *d* times because X_i has at most *d* values to delete.
- For each value in one domain, *REVISE* method examines all the values of the second domain.
  - So the complexity of the *REVISE* method would be d^2.
- *REVISE* is called at maximum $c + cd$ times.
- The complexity of the whole algorithm is $O((c+cd)d^2)$ which is $O(cd^3)$.

Path Consistency

idea of arc consistency:

- For every assignment to a variable u
  there must be a suitable assignment to every other variable v.
- If not: remove values of u for which
  no suitable "partner" assignment to v exists.

$\rightsquigarrow$ tighter unary constraint on u

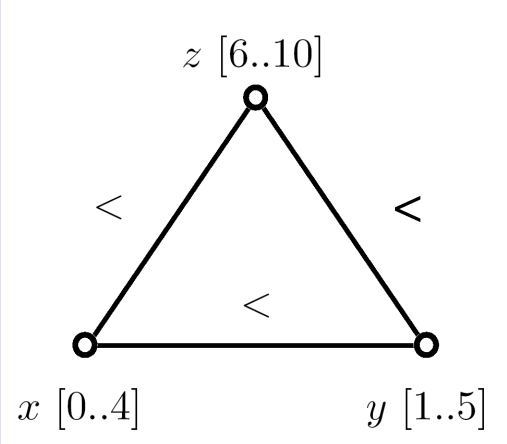
This idea can be extended to three variables (path consistency):

- For every joint assignment to variables u, v
  there must be a suitable assignment to every third variable w.
- If not: remove pairs of values of u and v for which
  no suitable "partner" assignment to w exists.

$\rightsquigarrow$ tighter binary constraint on u and v

Example

$\langle x < y, y < z, x < z \,;\, x \in [0..4], y \in [1..5], z \in [6..10] \rangle$ path consistent



$C_{x,y} = \{(a,b) \mid a < b, a \in [0..4], b \in [1..5]\}$

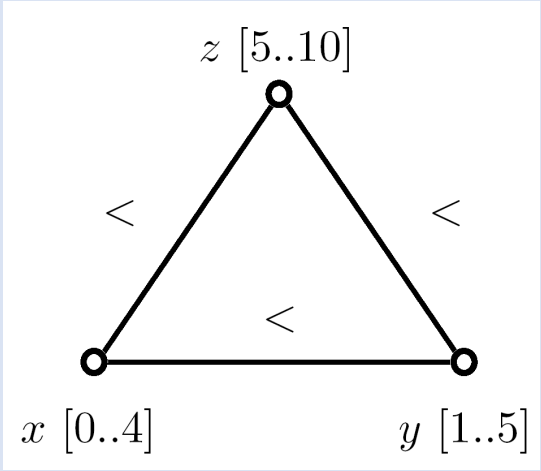
$C_{x,z} = \{(a,c) \mid a < c, a \in [0..4], c \in [6..10]\}$

$C_{y,z} = \{(b,c) \mid b < c, b \in [1..5], c \in [6..10]\}$

➵ the 3 conditions (cf. previous slide) are satisfied

Example

$\langle x < y, y < z, x < z \,;\, x \in [0..4], y \in [1..5], z \in [5..10] \rangle$ not path consistent



$C_{x,z} = \{(a,c) \mid a < c,\ a \in [0..4],\ c \in [5..10]\}$

But for $4 \in [0..4]$ and $5 \in [5..10]$ there is no $y \in [1..5]$ s.t. $4 < y$ and $y < 5$.

PC-6

- The best path-consistency algorithm proposed is PC-6, a natural generalization of AC-6 to path-consistency.
  - Its time complexity is $O(n^3d^3)$
  - Its space complexity is $O(n^3d^2)$,
  - where n is the number of variables and d is the size of domains.

Strong Consistency

Define P as a CSP where X, Y are the variables, domain of both is {1,2,3,4} and conditions in normal form are:

1. node-condition X<4
2. arc-condition X=Y

P is 2-consistent (arc consistent) because for any X value it is possible to find a Y value that fulfills the arc-condition X=Y.

However, P is not 1-consistent (node-consistent) because exist a X value (X=4) that can not fulfill the node condition x<4.

For these reasons, this problem is 2-consistent but not strongly 2-consistent.

Obviously, it is straightforward to convert this example in a strongly 2-consistent problem, just reducing the domain to {1,2,3}.

K-consistency

- A CSP is strongly k-consistent if it is k-consistent and is also
  - (k − 1)-consistent, (k − 2)- consistent, ... all the way down to 1-consistent.
- Suppose we have a CSP with *n* nodes and make it strongly n-consistent.
  - First, we choose a consistent value for X_1.
  - We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on.
  - For each variable X_i, we need only search through the *d* values in the domain to find a value consistent with $X_1,\ldots,X_{i-1}$.
- The total run time is only $O(n^2d)$.

Global Constraints

- A global constraint is one involving an arbitrary number of variables (but not necessarily all variables)
  - *Alldiff* constraint says that all the variables involved must have distinct values
- One simple form of inconsistency detection for *Alldiff* constraints works as follows:
  - if *m* variables are involved in the constraint, and if they have *n* possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

Global Constraints

- Simple algorithm
  - Remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables.
  - Repeat as long as there are singleton variables.
  - If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.
- Can detect the inconsistency in the assignment *{WA=red, NSW=red}*
  - The variables *SA*, *NT*, and *Q* are connected by an *Alldiff* constraint
  - After applying AC-3 with the partial assignment, the domains of SA, NT, and Q are all reduced to {green,blue}.
  - We have three variables and only two colors, so the *Alldiff* constraint is violated.
- A simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc-consistency to an equivalent set of binary constraints.

Sudoku

- A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9.
- The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box.
- A row, column, or box is called a unit.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

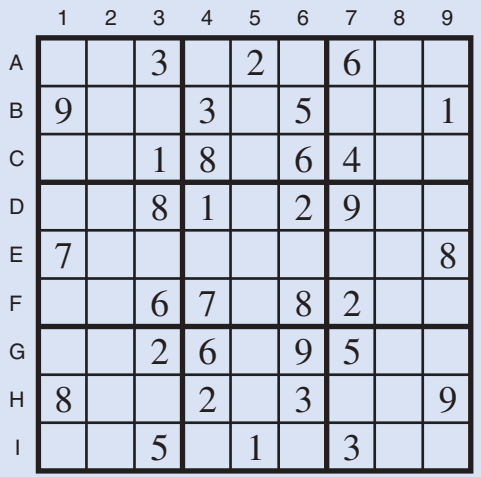
	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Sudoku

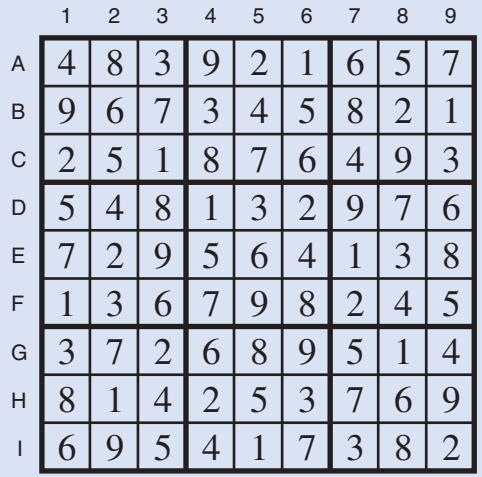
- A Sudoku puzzle can be considered a CSP with 81 variables, one for each square.
- We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row.
- The empty squares have the domain {1,2,3,4,5,6,7,8,9} and the pre-filled squares have a domain consisting of a single value.
- There are 27 different *Alldiff* constraints, one for each unit (row, column, and box of 9 squares):

$$
\begin{aligned}
&Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)\\
&Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)\\
&\dots\\
&Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)\\
&Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)\\
&\dots\\
&Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)\\
&Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)\\
&\dots
\end{aligned}
$$

Sudoku



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		



	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

- *Alldiff* constraints have been expanded into binary constraints (such as A1 ≠ A2) so that we can apply the AC-3 algorithm directly.
  - Consider variable E6,
    - from the constraints in the box, we can remove 1, 2, 7, and 8 from E6's domain.
    - from the constraints in its column, we can eliminate 5, 6, 2, 8, 9, and 3.
    - that leaves E6 with a domain of {4}
  - Consider variable I6
    - applying arc consistency in its column, we eliminate 5, 6, 2, 4 (E6 must be 4), 8, 9, and 3.
    - we eliminate 1 by arc consistency with I5, and we are left with only the value 7 in the domain of I6.
    - now there are 8 known values in column 6, so arc consistency can infer that A6 must be 1.
- Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value

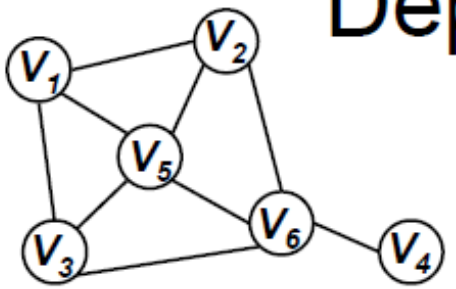
Backtracking Search for CSPs

- Sometimes we can finish the constraint propagation process and still have variables with multiple possible values.
- In that case we have to search for a solution.

Backtracking Search for CSPs

- Consider how a standard depth-limited search could solve CSPs.
- A state would be a partial assignment, and an action would extend the assignment, *NSW = red* or *SA = blue*
- For a CSP with *n* variables of domain size *d*
  - We have a search tree where all the complete assignments are leaf nodes at depth *n*.
- But notice that the branching factor at the top level would be *nd* because any of *d* values can be assigned to any of *n* variables.
- At the next level, the branching factor is *(n − 1)d*, and so on for *n* levels.
- So the tree has $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

Depth First Search



V_1	V_2	V_3	V_4	V_5	V_6
?	?	?	?	?	?

V_1	V_2	V_3	V_4	V_5	V_6
B	?	?	?	?	?

V_1	V_2	V_3	V_4	V_5	V_6
R	?	?	?	?	?

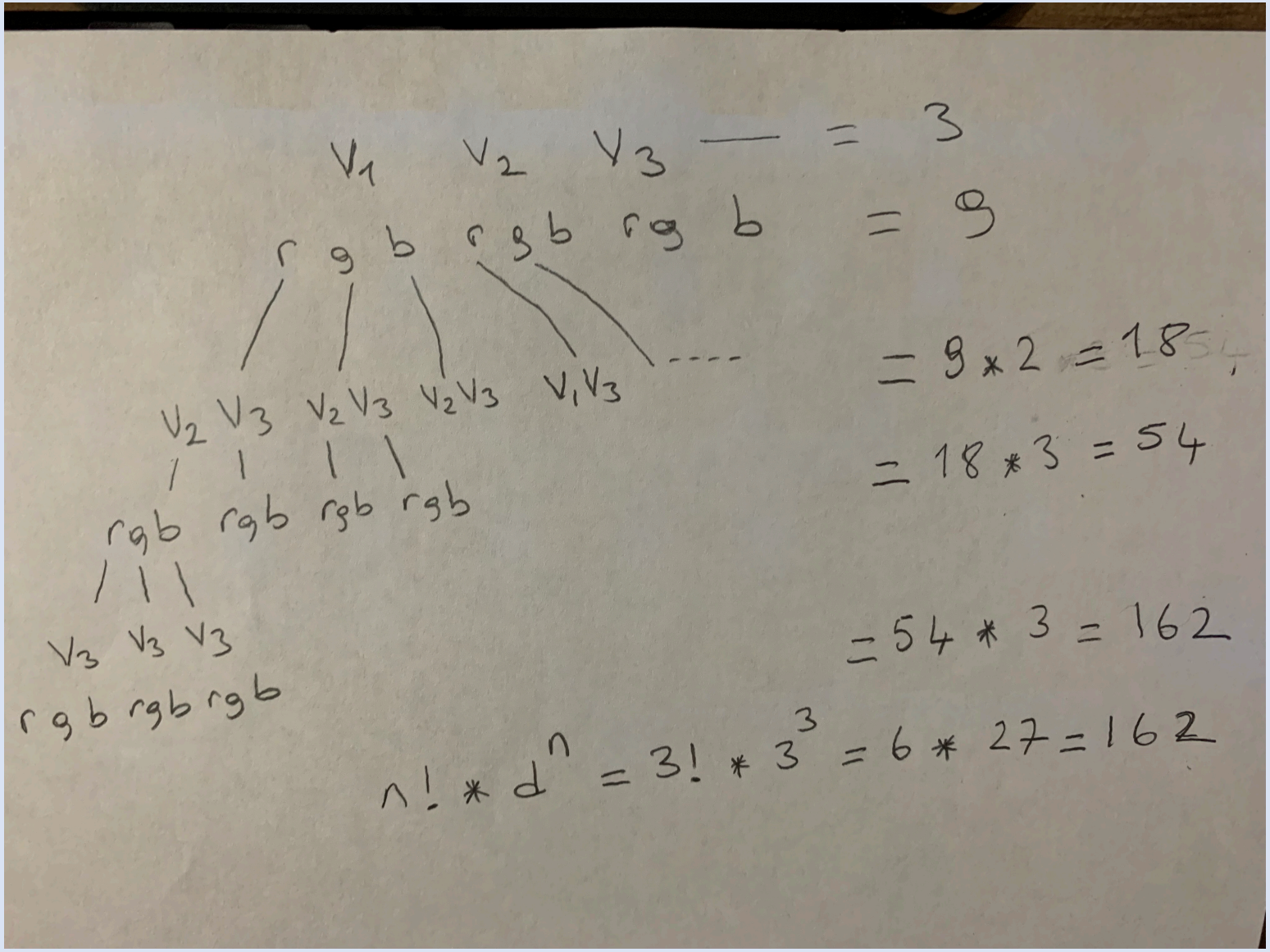
V_1	V_2	V_3	V_4	V_5	V_6
G	?	?	?	?	?

V_1	V_2	V_3	V_4	V_5	V_6
B	B	?	?	?	?

- Recursively:
  - For every possible value in *D:*
    - Set the next unassigned variable in the successor to that value
    - Evaluate the successor of the current state with this variable assignment
    - Stop as soon as a solution is found

9d

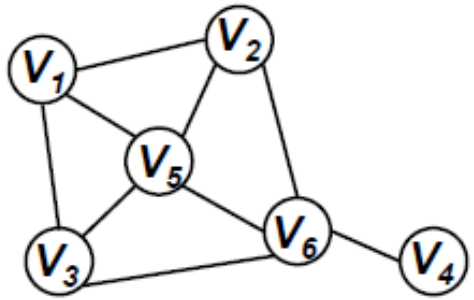
Search tree for CSP



Backtracking Search for CSPs

- We can get back that factor of *n!* by recognizing a crucial property of CSPs: *commutativity*.
- In CSPs, it makes no difference if we first assign NSW = red and then SA = blue, or the other way around.
- Only consider a single variable at each node in the search tree.
- At the root we might make a choice between SA=red, SA=green, and SA=blue, but we would never choose between NSW=red and SA=blue.
- With this restriction, the number of leaves is d^n

Backtracking DFS



V_1	V_2	V_3	V_4	V_5	V_6
?	?	?	?	?	?

V_1	V_2	V_3	V_4	V_5	V_6
B	?	?	?	?	?

Order of values:
(**B**,**R**,**G**)

V_1	V_2	V_3	V_4	V_5	V_6
B	**B**	?	?	?	?

Don't even consider that branch because V_2=**B** is inconsistent with the parent state

V_1	V_2	V_3	V_4	V_5	V_6
B	**R**	?	?	?	?

V_1	V_2	V_3	V_4	V_5	V_6
B	**R**	**R**	**B**	?	?

Backtrack to the previous state because no valid assignment can be found for V_6

V_1	V_2	V_3	V_4	V_5	V_6
B	**R**	**R**	**B**	**G**	?

Example – N Queen Problem

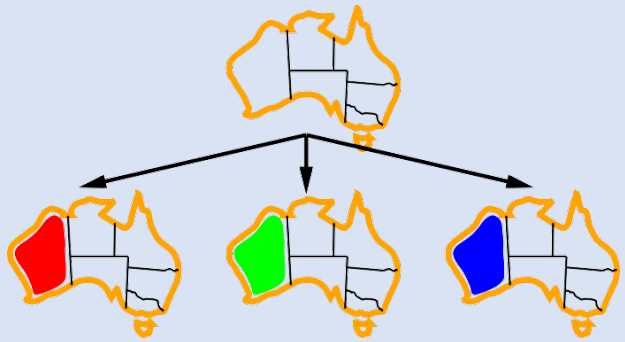
- https://www.youtube.com/watch?v=0DeznFqrgAI

Backtracking Search for CSPs

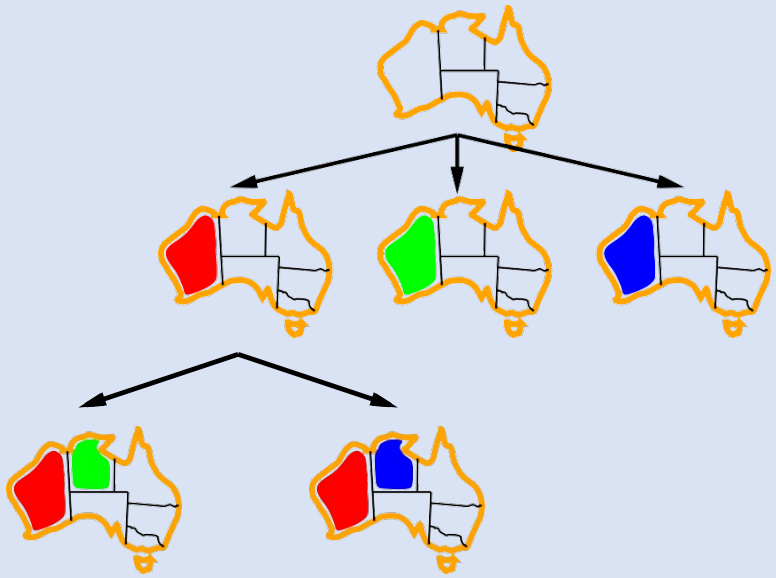
- Depth-first search for CSPs with single-variable assignments is called backtracking search
- Backtracking search is the basic uninformed algorithm for CSPs

Backtracking example

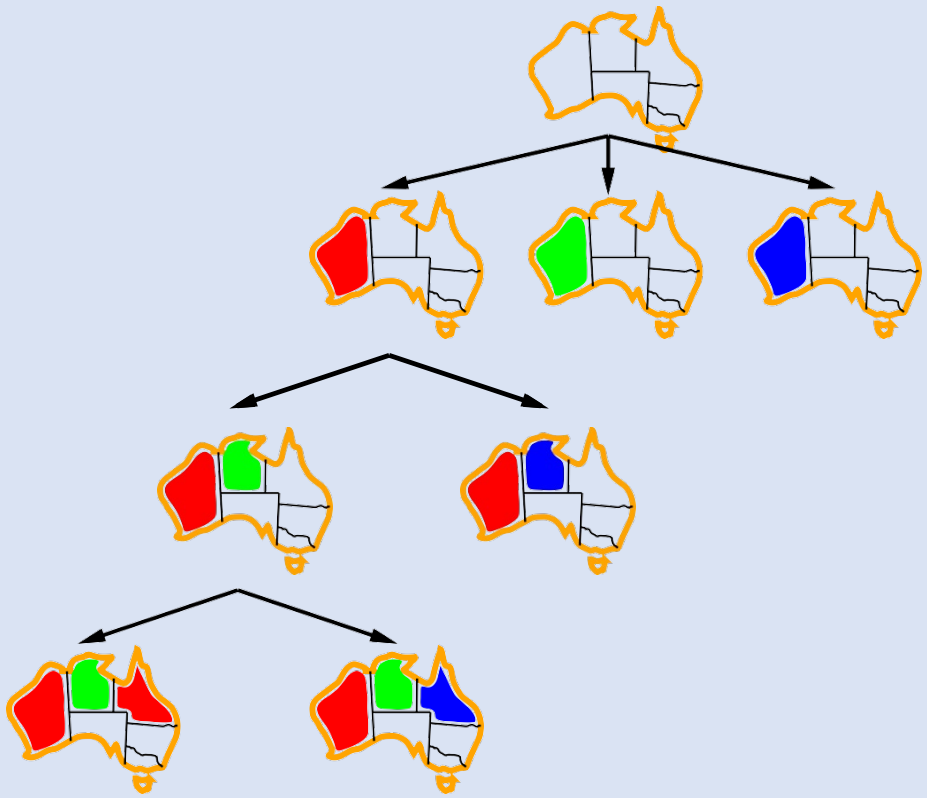
Backtracking example



Backtracking example



Backtracking example



Backtracking algorithm

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

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

Figure 5.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES implement the general-purpose heuristics discussed in Section 5.3.1. The INFERENCE function can optionally impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are retracted and a new value is tried.

Variable and value ordering

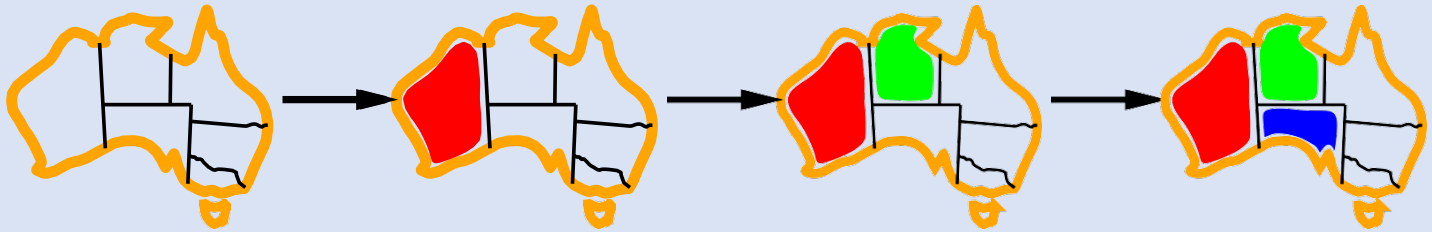
- The backtracking algorithm contains the line

$$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment)\,.$$

- Strategies for SELECT-UNASSIGNED-VARIABLE
  - static ordering: choose the variables in order, {X1,X2,...}.
  - choose randomly.
- Neither strategy is optimal.
  - after the assignments for *WA=red* and *NT=green*, there is only one possible value for *SA*, so it makes sense to assign SA=blue next rather than assigning Q.
  - after SA is assigned, the choices for Q, NSW, and V are all forced.

Minimum remaining values

Minimum remaining values (MRV):
choose the variable with the fewest legal values

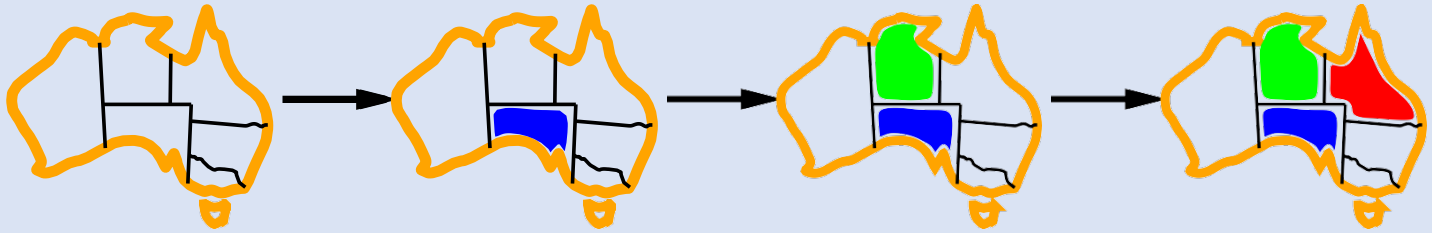


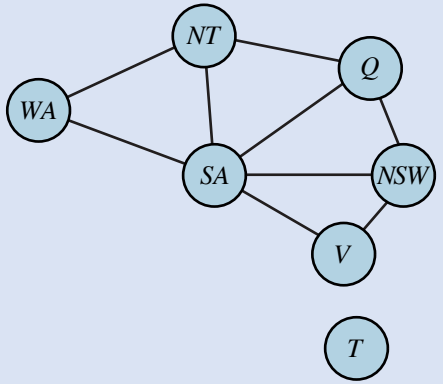
Degree heuristic

- The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors.

Tie-breaker among MRV variables

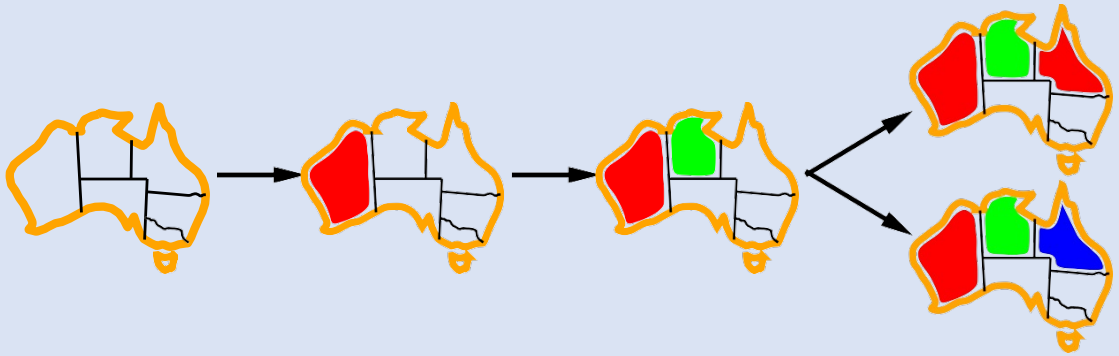
Degree heuristic:
choose the variable with the most constraints on remaining variables





Least constraining value

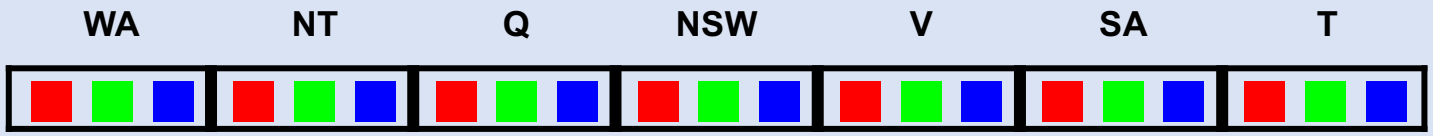
Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



Combining these heuristics makes 1000 queens feasible

INFERENCE - Forward checking

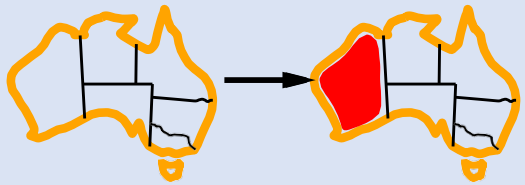
Idea: Keep track of remaining legal values for unassigned variables (arc consistency)
Terminate search when any variable has no legal values

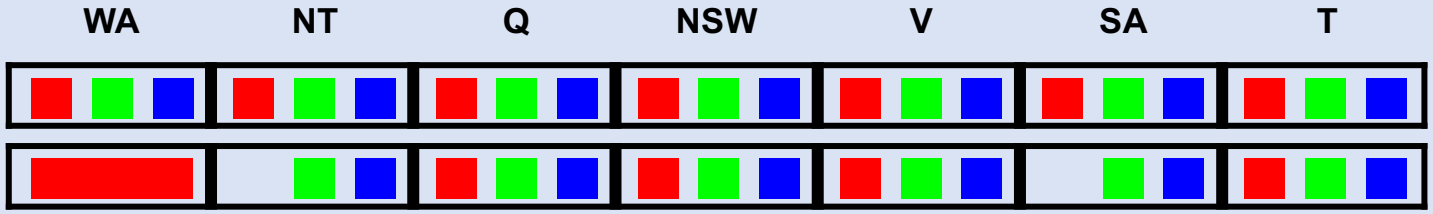


INFERENCE - Forward checking

Idea: Keep track of remaining legal values for unassigned variables (arc consistency)

Terminate search when any variable has no legal values

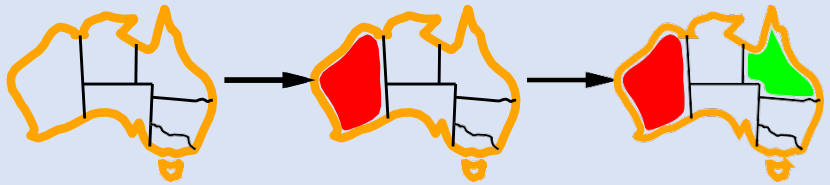




INFERENCE - Forward checking

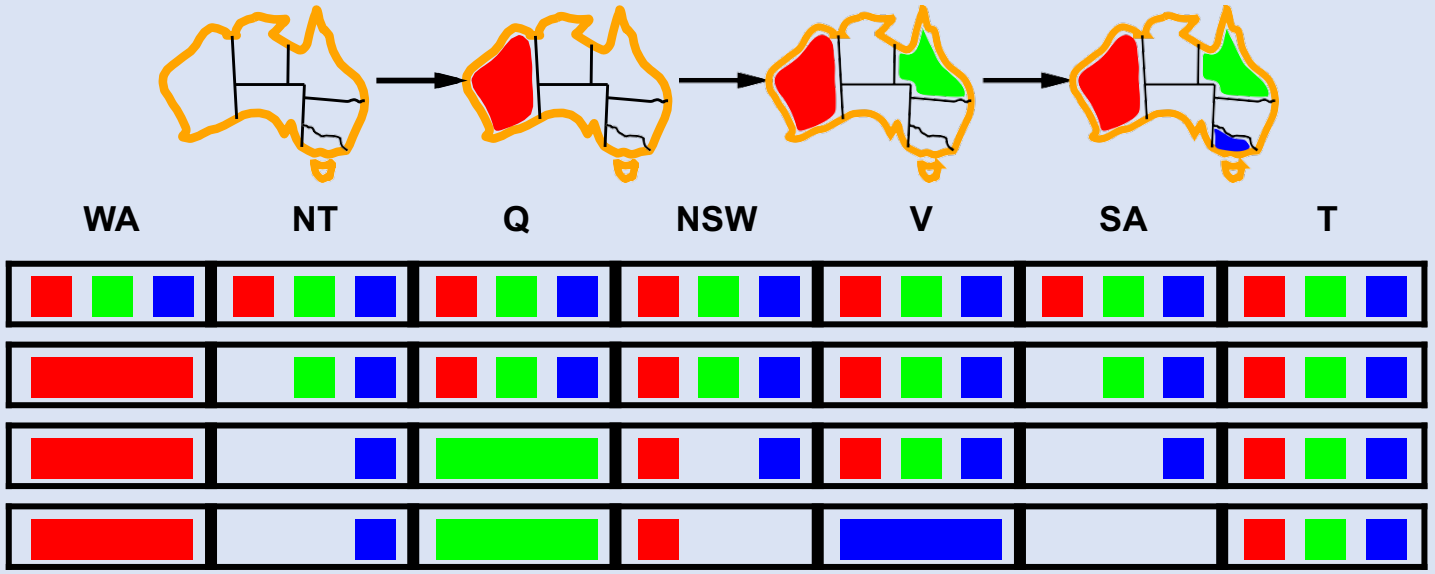
Idea: Keep track of remaining legal values for unassigned variables (arc consistency)

Terminate search when any variable has no legal values



INFERENCE - Forward checking

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



- MAC detects inconsistencies like this. After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i.
- From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately.

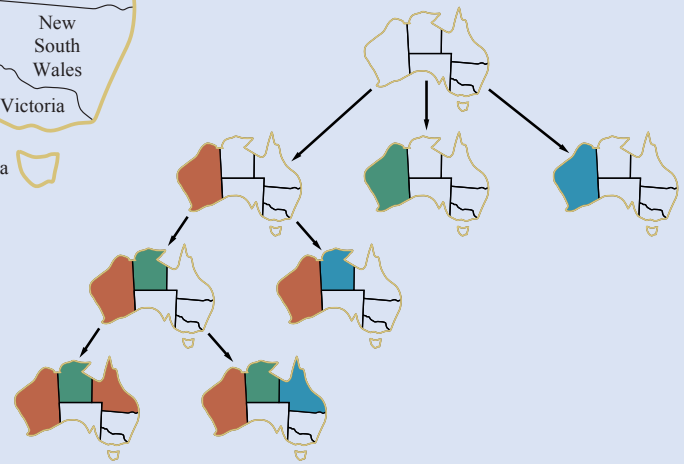
Intelligent backtracking: Looking backward

- Chronological backtracking :
  - what to do when a branch of the search fails: back up to the preceding variable and try a different value for it.
  - the most recent decision point is revisited.
- Consider a fixed variable ordering *Q, NSW , V , T , SA, WA, NT*
  - partial assignment {Q=red, NSW=green, V=blue, T=red}
  - when we try the next variable, *SA*, we see that every value violates a constraint.
  - we back up to *T* and try a new color for Tasmania!
    - Obviously this is silly - recoloring Tasmania cannot possibly help in resolving the problem with South Australia.

Intelligent backtracking: Looking backward

- Backtrack to a variable that might fix the problem
  - a variable that was responsible for making one of the possible values of *SA* impossible.
  - keep track of a set of assignments that are in conflict with some value for *SA*.
    - The conflict set: *{Q=red,NSW=green,V=blue}*.
  - the backjumping method backtracks to the most recent assignment in the conflict set;
  - backjumping would jump over Tasmania and try a new value for *V (Q, NSW , V , T , SA, WA, NT)*.
  - It is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign.
  - If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.

Constraint learning



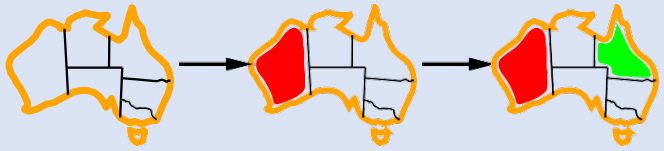
- The idea of finding a minimum set of variables

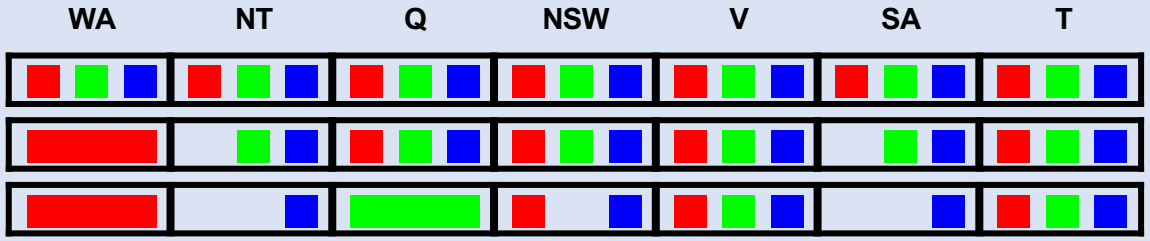
from the conflict set that causes the problem.

  - no-good: set of variables, with their corresponding values.
- Consider the state *{WA=red, NT=green, Q=blue}*
- Forward checking can tell us this state is a no-good because there is no valid assignment to *SA*.
  - suppose that the search tree were actually part of a larger search tree that started by first assigning values for *V* and *T* .
  - it would be worthwhile to record *{WA=red, NT=green, Q=blue}* as a no-good because we are going to run into the same problem again for each possible set of assignments to *V* and *T* .

Constraint propagation

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





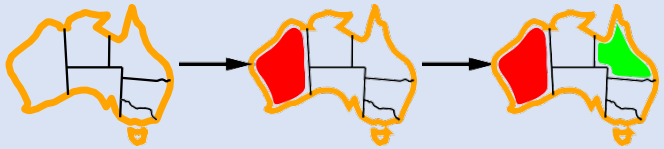
NT and SA cannot both be blue!

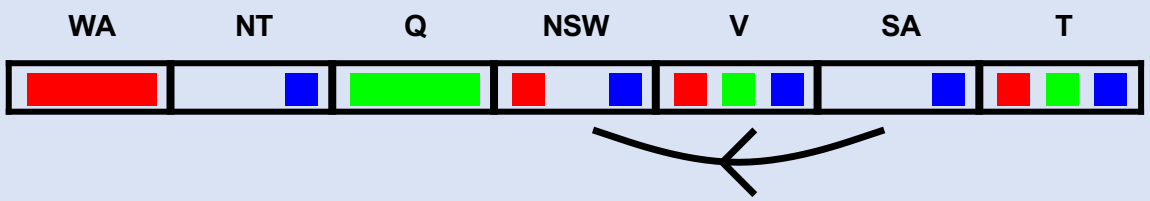
Constraint propagation repeatedly enforces constraints locally

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

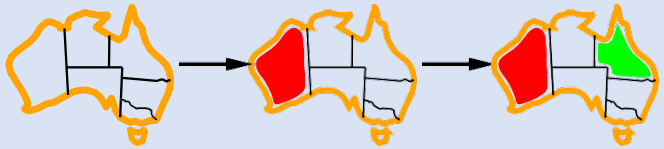


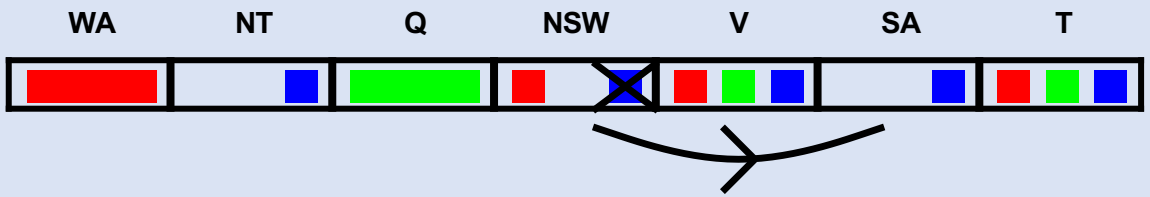


Arc consistency

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

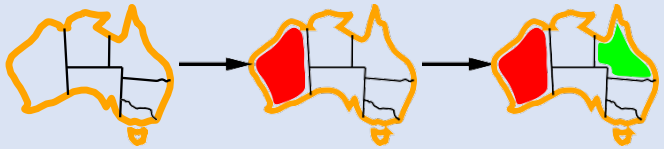


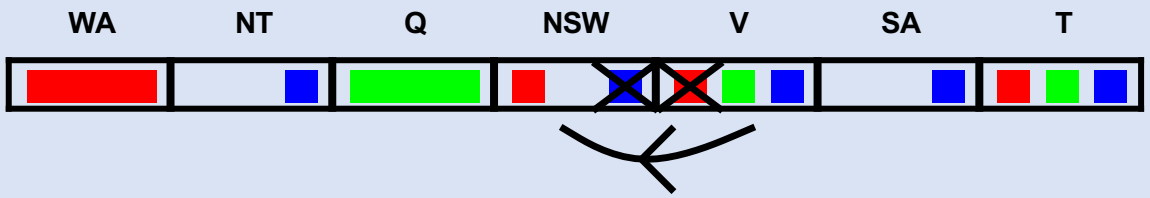


Arc consistency

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



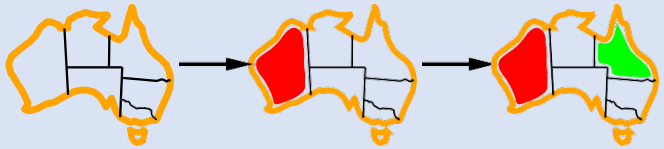


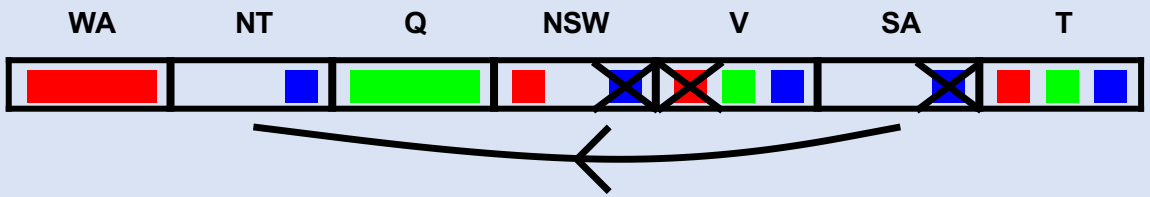
If X loses a value, neighbors of X need to be rechecked

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

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm

```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← Remove-First(queue)
    if Remove-Inconsistent-Values(Xi, Xj) then
      for each Xk in Neighbors[Xi] do
        add (Xk, Xi) to queue

---

function Remove-Inconsistent-Values(Xi, Xj) returns true iff succeeds
  removed ← false
  for each x in Domain[Xi] do
    if no value y in Domain[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
      then delete x from Domain[Xi]; removed ← true
  return removed
```

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

The End!