Constraint Satisfaction Problem (CSP)

Artificial Intelligence

Lecture note

Authors: Alireza Ilami, Ghazal Shenavar, Nima Salem

Dr.Mohammad Hossein Rohban
Department of Computer Engineering
Sharif University of Technology
Spring 2021

Introduction

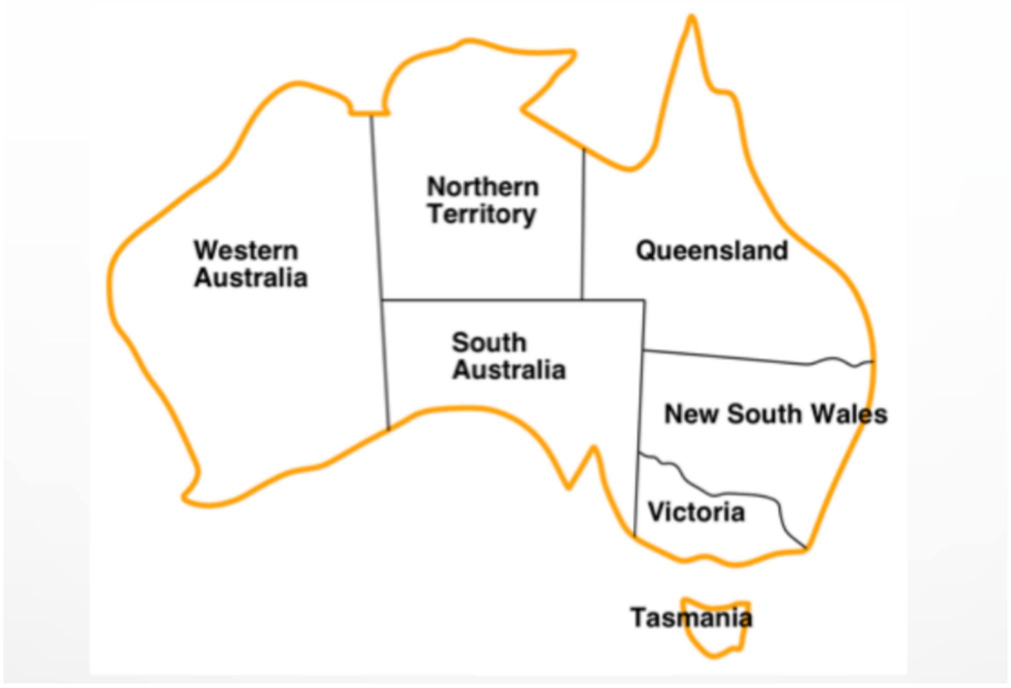
Constraint Satisfaction Problems are a special format of search algorithm problems. They seek to find a search algorithm which is more useful in particular cases with some new conditions. These conditions and differences to the main problem are called Constraints.

In CSP, we define a state by some variables X_i with values from domain D_i.

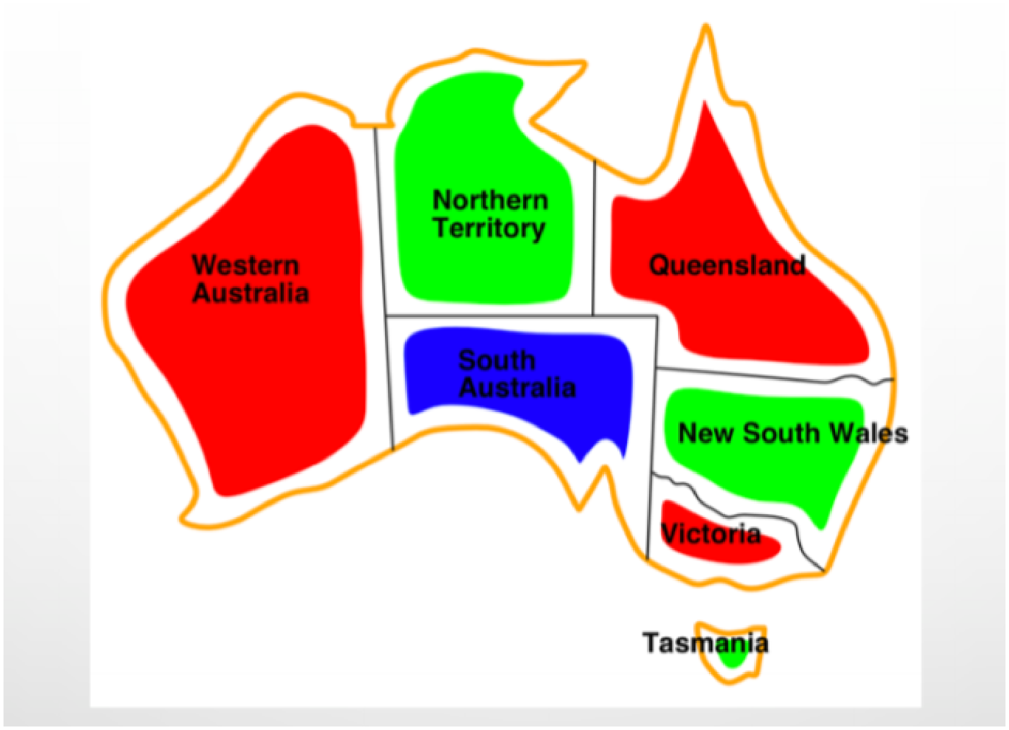
Goal test is a set of constraints specifying allowable combinations of values for subsets of variables.

Goal test can be a function that gets all variables and returns a boolean value that says whether this is an acceptable state or not.

There are various examples of CSPs.
One of the most famous examples is Map Coloring.
Map Coloring:

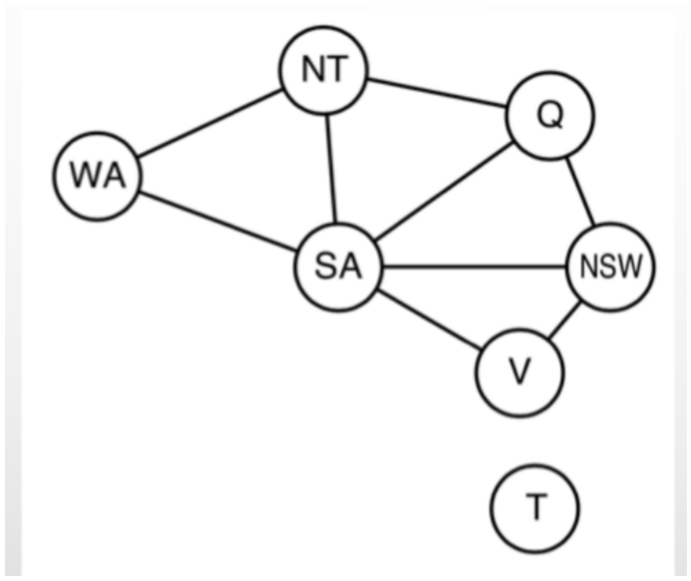


We want to color all Australia regions. The constraint is that adjacent regions must have different colors.
Variables: WA, NT, Q, NSW, V , SA, T
Domains Di = {red, green, blue}
Solutions are assignments satisfying all constraints:
{WA=red, NT =green, Q=red, NSW =green, V =red, SA=blue, T =green}



Constraint Graph

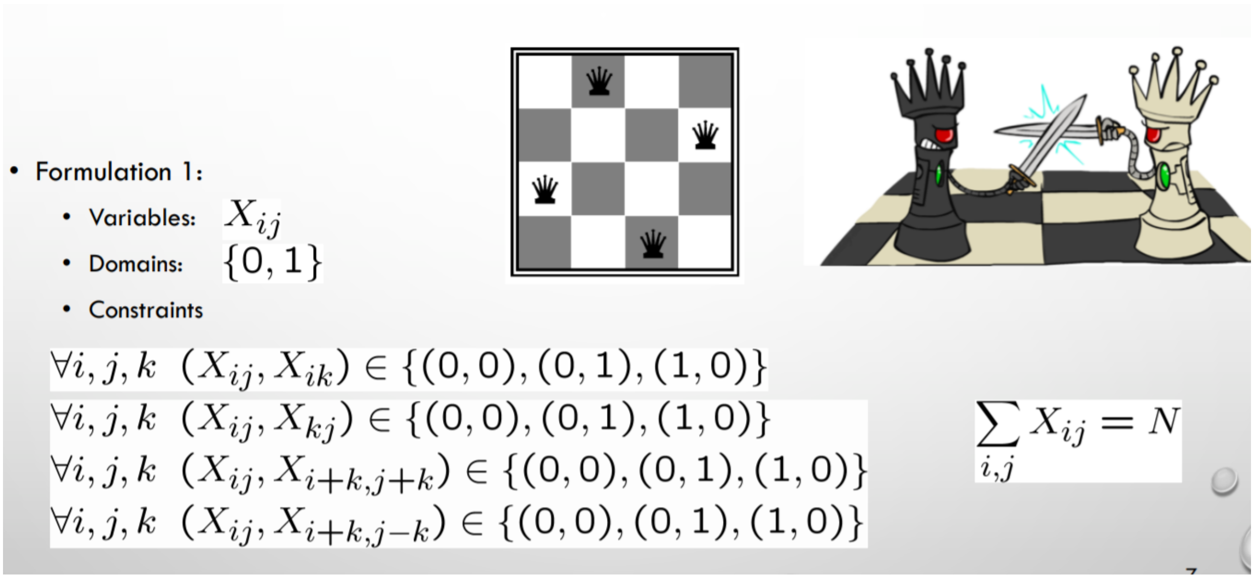
Like the Map Coloring problem, we usually try to define a constraint for the relation between two variables.
These CSPs can be shown by a binary Constraint graph; in which nodes are the variables and arcs are the constraints.
We also have other Constraint Graph types for more complicated problems.
For Map Coloring example, the Constraint Graph is:



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

CSP Examples

N-Queens example:
We want to put n queens in a n * n chess board in a position that none of them threaten each other.



- Formulation 1:
  - Variables: X_{ij}
  - Domains: $\{0, 1\}$
  - Constraints

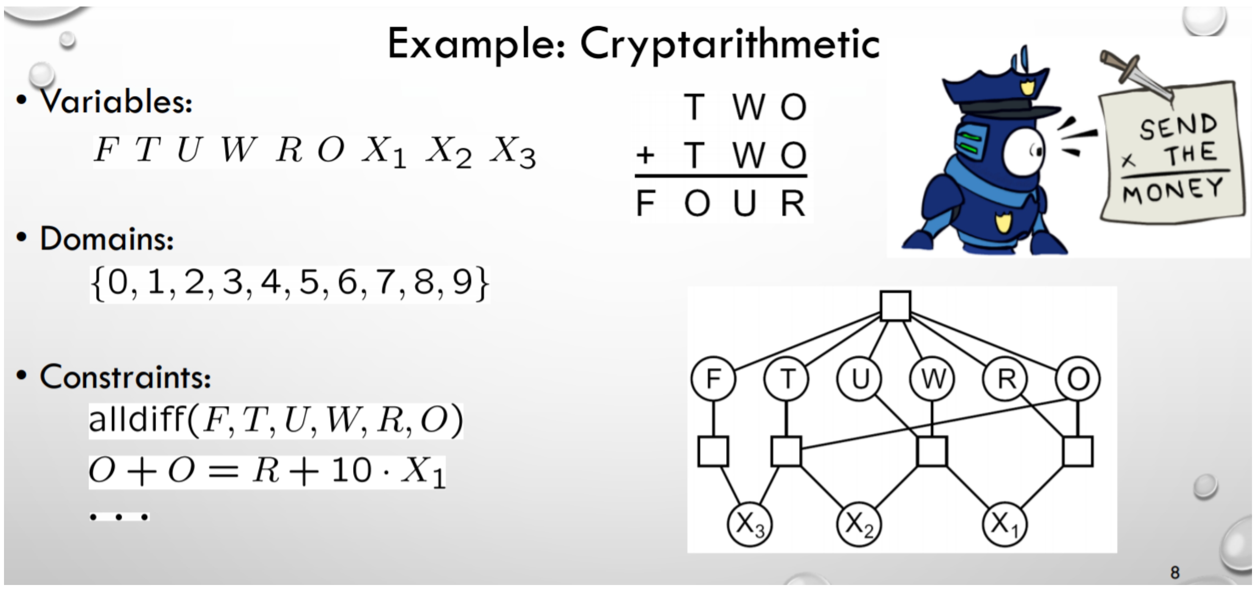
$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

$$\sum_{i,j} X_{ij} = N$$

There must be exactly one queen in each row and each column.

Another famous example is the Cryptarithmetic Problem. We assign each letter to a number between 0 to 9 in order to do some arithmetic operations with letters.

Example:



Example: Cryptarithmetic

- Variables:
  $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
- Domains:
  $\{0,1,2,3,4,5,6,7,8,9\}$
- Constraints:
  $\text{alldiff}(F,T,U,W,R,O)$
  $O+O=R+10\cdot X_1$
  $\cdots$

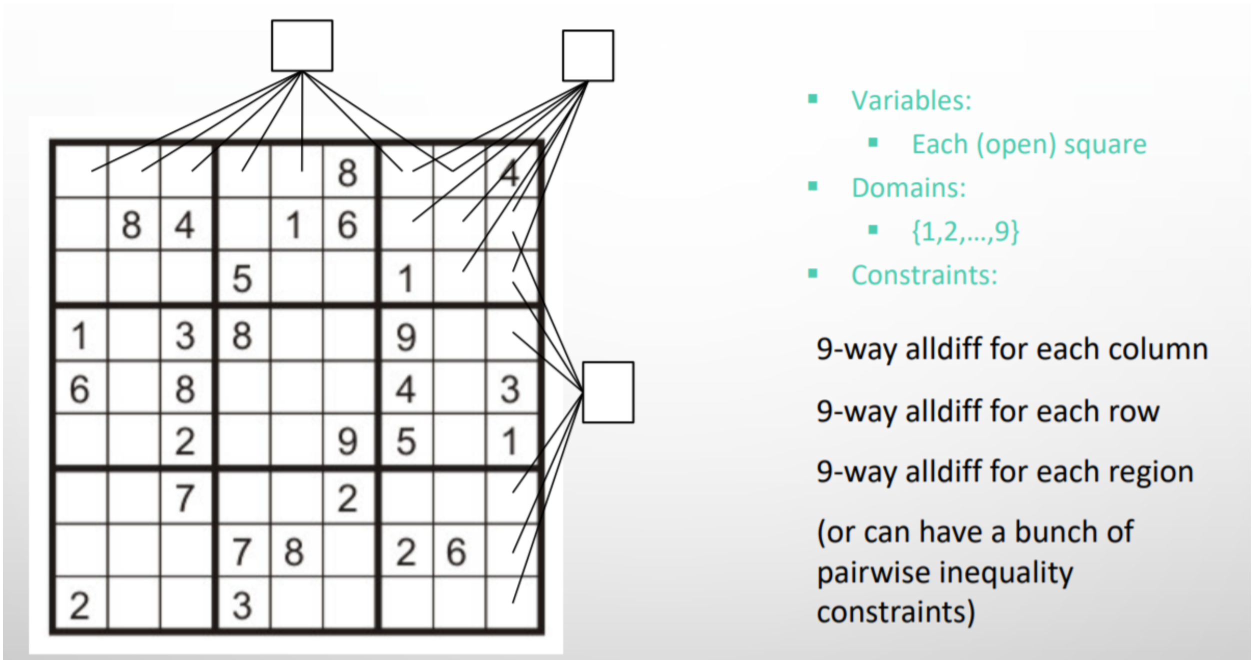
```
    T W O
  + T W O
  -------
  F O U R
```

SEND
THE
MONEY
F
T
U
W
R
O
X3
X2
X1
8

Our variables and domains are shown. Constraints are arithmetic rules. For example, adding two ‘O’ s must give ‘R’ + 10 x ‘X1’ . ‘X1’ is carried for the next adder.
For this problem, our constraint graph is not binary. Because arcs are not only related to two variables. Therefore, we can define each constraint with a square. And connect all variables which are participating in that constraint.

In this type of constraint graph, there is an arc between a variable and a constraint if and only if the variable is involved in the constraint. So, there is not any arc between two variables or two constraints.

Another example is the Sudoku table.



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Sudoku is a very famous game in which we should put digits 1 - 9 in places so that all numbers of each row, column and box are unique.

Real-World CSPs

There are many CSPs in the real world. We will note some of them:

Assignment problems: Who teaches what class? Constraints are mostly teachers' schedules.
Timetabling problems: What class is offered and when? Some constraints are students' schedules.
There are also some other problems that can be defined as a CSP and be solved, such as Hardware configurations, Transportation scheduling, Circuit layout, factory scheduling, fault diagnosis, etc.

CSP Varieties

All CSPs can be classified into two parts: Discrete variables and Continuous variables.
For discrete variables, problems are divided into two modes. Finite domains and infinite domains.

- Discrete variables
    - Finite domains
        - Size d => $O(d^n)$ complete assignments
        - e.g., Boolean CSPs, including Boolean Satisfiability (NP-Complete)
    - Infinite domains
        - Not discussed in this course
        - Domains can be integers, real numbers or strings
        - e.g., Job Scheduling, variables are start/end dates for each job
        - Need a constraint language. e.g., startjob_1 + 5 <= startjob_3
        - Linear constraints are solvable, nonlinear are undecidable
- Continuous variables
    - Not discussed in this course, discussed in Optimization course
    - e.g., start/end time for Hubble Telescope observations
    - Linear constraints are solvable in poly time by linear programming

Constraint Varieties

- Unary constraints:
    - Involve a single variable
    - SA $\neq$ Green (Map Coloring Problem)
- Binary constraints
    - Involve pairs of variables
    - SA $\neq$ WA (Map Coloring Problem)
- Higher order constraints
    - Involves three or more variables

  - Cryptarithmetic column constraints
- Preferences
  - Soft constraints
  - Red is better than Green
  - Often representable by a cost for each variable assignment
  - this is not CSP, it is Constraint Optimization Problem (COP)

Converting n-ary CSP to binary CSP

If we take constraints as variables:
C1: $x + y <= z$
C2: $x - y = 3$

x, y in {1,2,...,5}

So, C1 is:
{(1, 1, 2), (), ...}

And C2 is:
{(4, 1), …}

Therefore, our nodes are constraints. But what about arcs?
Arc between C1 and C2 check whether mutual x, y, … (variables) take the same value in both constraints.
e.g., (1, 1, 2) in C1 and (4, 1) in C2 can not be correct. Because x is shared and takes two different values.

Standard Search Formulation

We first discuss a solution based on uninformed search ,then we will improve it to an informed search and later on we will further improve it with CSP problems traits in mind.

Let’s look at csp as a search problem and determine what we need:

Initial State: No variable is assigned, the empty assignment : {}

Successor Function: assign a value to an unassigned variable that does not conflict with current assignment.

Failure is recognized in this function if no legal assignment is found.

Success is recognized when all variables have assigned values.

Goal test: The first place that the current assignment is complete.

This formulation is the same for all CSPs.

A good algorithm to use is depth first search because our goal is to assign values to n variables; therefore, every possible solution is in depth n.

As the path is not important in this problem, complete state formulation can be used.

The Number of Leaves: Let's calculate the number of leaves. In the initial state, we can choose any of the n variables to assign a value from d possible values to(d is the size of the domain); therefore, we have nd states in the first layer. From this we can see that the branching factor of the initial state is nd (branching factor is the number of branches that a node in each depth has). With the same principle, every node of the first layer can have (n-1)*d branches. At depth l, every node in depth l has (n - l)d branching factor. With having this in mind let's calculate the number of leaves:

The number of leaves = nd x (n - 1)d x … x (n - l)d x … x d = n!d^n

Backtracking Search

To make the number of leaves less and make the search more efficient, we consider one order of assignment and follow that. This will give us the correct answer because the variable assignments are commutative. For example, [WA=red then NT=green] same as [NT=green then WA=red]

In this way, each node in a special depth only has to consider assignments to one variable; therefore, it's branching factor will reduce to d from (n - l)d. Ultimately the number of leaves will reduce to **d^n**.

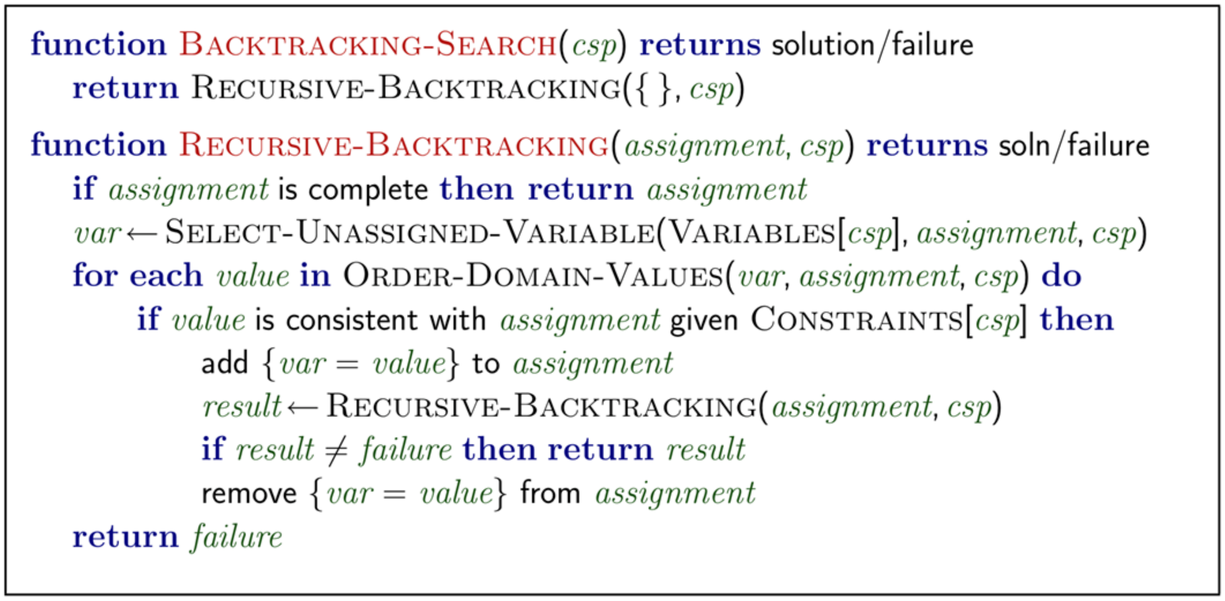
Depth-first search for CSPs with single-variable assignments is called backtracking search. The name backtracking is chosen because if during search, the algorithm fails to find a possible assignment for a variable it will go up to a higher depth to test another value for the variable of that higher depth therefore it backtracks to find a possible answer. For an example, let's assume we are finding values for x, y, z in that order. In

the initial state, no value is assigned. In the first layer we will have all the possible values for x. As we are using depth first search, one value for x would be expanded and its successors in the second layer will have all the values for y. In the next step, we will expand a node from the second layer and make our third layer. Let's assume that z has no legal value; then the algorithm will backtrack to the second layer and expand another node in the y layer.

Backtracking search is the basic uninformed algorithm for CSPs

It can solve the n-queens problem for n ≈ 25

Backtracking pseudo code

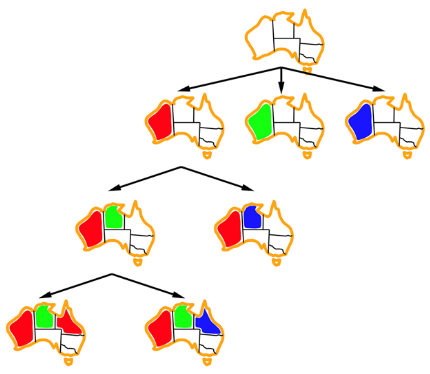


```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

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
```

The algorithm tries to find non-conflicting values for all variables recursively. If it finds an answer, it returns the answer, else if failure is reached for a value it will remove that value from possible answers of that variable and try another value for that variable.

Let's see an example of backtracking: coloring states with no neighbouring states having the same color. The possible colors are blue, green and red:



A bit of extra information: The problems can be divided into 2 categories: decision problems and optimization problems.
Decision problems are problems that have a yes or no answer like: can you color this graph with 3 colors without any of the connected nodes having the same color? Is there a path between these 2 nodes?
Optimization problems are problems in which you are trying to find the best solution from all feasible solutions. For example, the minimum number of colors you need to color a graph without any of the connected nodes having the same color.
Backtracking is used to solve decision problems. However it is important to notice that you can break down optimization problems into a number of decision problems. For example, you can find the minimum number of colors needed to color a graph by starting from a low bound and checking if you can color a graph with that number of colors and everytime you fail, add one more color until you reach success.

Improving Backtracking Efficiency

There are four general ideas for improving backtracking efficiency:

1. Which variable should be assigned next?

2. In what order should its values be tried?

3. Can we detect inevitable failure early?

4. Can we take advantage of problem structure?

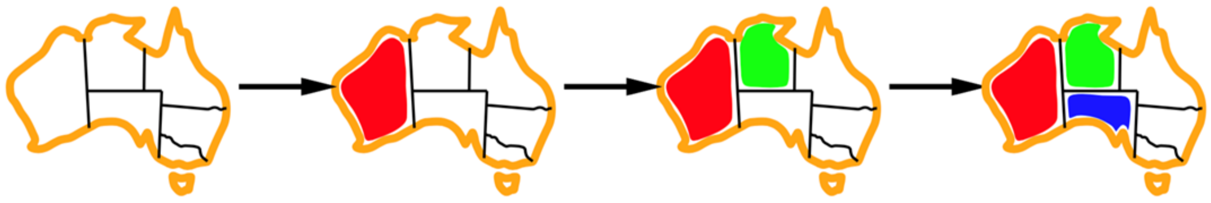
Let's talk about the first question. What order should the variables be assigned in? Should we use a static order? Or should we change the order based on the problem?

One of the possible answers to the above question is choosing **The Minimum Remaining Value (MRV).**

The Minimum Remaining Value (MRV)

MRV is a heuristic that can be used to decide which variable is better to be assigned next. Each time you assign a value to a variable, your problem constraints may limit the possible values the next variable can take. For example, in the below picture we do not want any neighbouring states to have the same color. So after we color the first state, its neighbouring states will have one less possible value while the other states which do not share a border with it may choose any value.

Why do we use this heuristic? Because if we continue our assignment steps in a random order we might reach a point that a variable becomes so limited that it has 0 remaining possible values. For example, In the last step of the picture bellow if we choose to color the north west state before the the state in the south, and color it blue, there will be no possible remaining values for the south state and we will need to backtrack; While if we color the states in the order shown in the picture, the north west state could be colored red in the next step and no backtracking is needed.

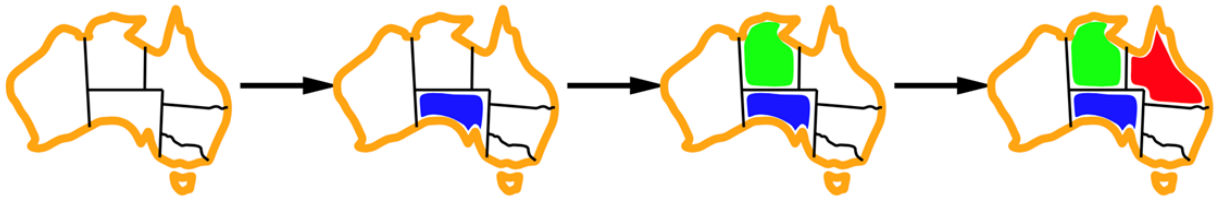


Now, what should we do when we have the same MRV for a number of variables? We need a tie-breaker in this situation.

Degree Heuristic

Degree Heuristic is a tie breaker we can use; this heuristic gives priority to the variable that will impose the most constraints on the remaining variables. For example, in our state coloring example we need to choose the state with the most neighbours.

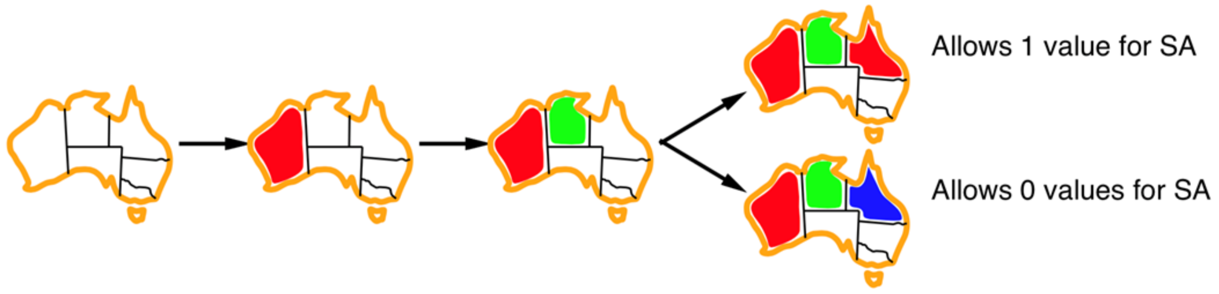
This heuristic will choose the order that can show if a path leads to failure sooner.



Now let’s talk about the second question: In what order should its values be tried?

Least Constraining Value

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



This heuristic allows us to continue our search for a longer time and avoid dead ends.

At first glance, this might look contradictory to our goal of trying to reach the dead end sooner in the last part; this difference is because of the nature of values and variables.

Each variable should be assigned in our search to find an answer but values can just not be used. So if a variable has no possible values, failure is inevitable but if a value results in failure we can simply use another value.

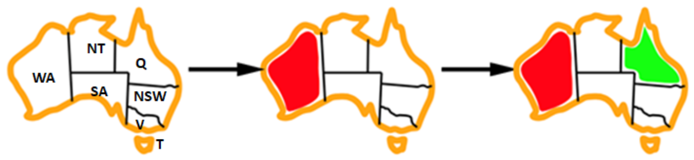
Combining the heuristics that were mentioned can make 1000 queens feasible.

On to the third question: Are we able to detect inevitable failure early? Do we need to go as deep as the leaves to realise an answer results in failure? Or can we distinguish failure sooner and save time?

Forward Checking

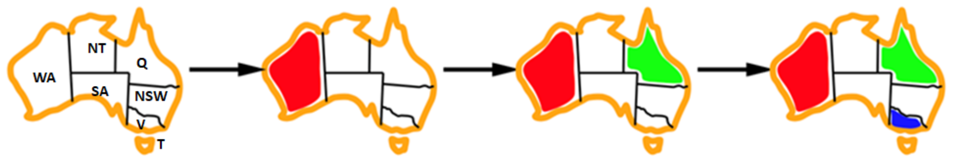
Keep track of remaining legal values for unassigned variables and terminate the search when any variable has no legal value.

For example:



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

Each table is the possible values for states in each step. Before taking any step, all states could have been colored by all colors. After taking two steps, as we can see in the table, all uncolored states still have possible values. We take another step.

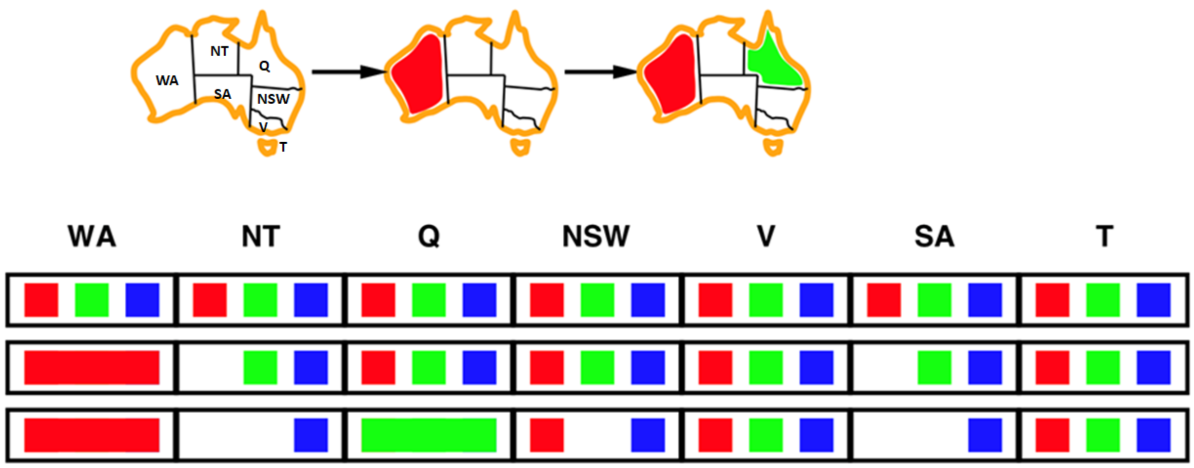


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

Now the middle down state no longer has a possible value. If we use MRV, the state will be chosen as our next step and we will detect failure in one step. If we don't use MRV, we might need to take many more steps to detect failure. However, we can detect failure at this very step that we are in.

Constraint Propagation

We still want faster detection so we use this method. Constraint propagation repeatedly enforces constraints locally. For an example:

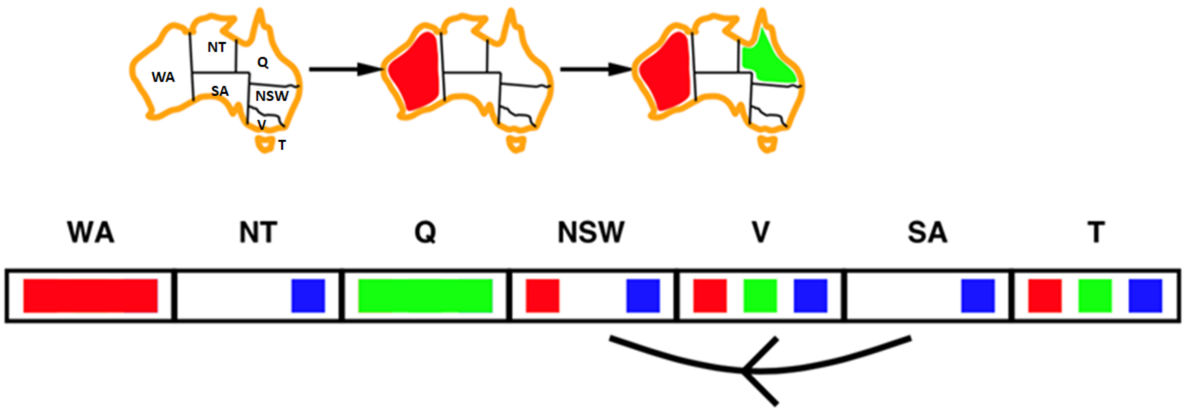


As NT and SA are both neighbours so they can not both be blue, failure can be detected at this state by checking the constraints.

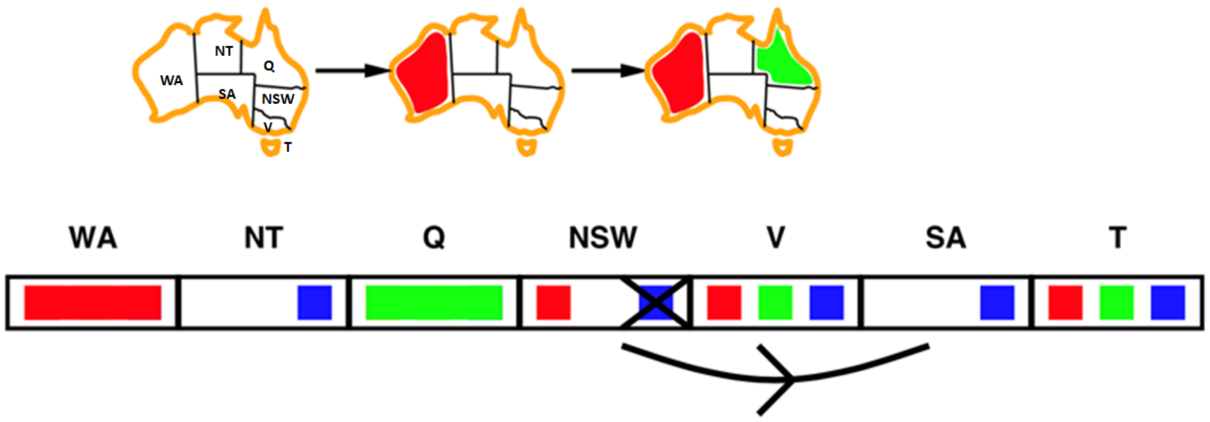
Arc Consistency

Simplest form of propagation is called Arc Consistency. Arc Consistency states that any arc X → Y is consistent iff for ***every*** value x of X there is ***some*** allowed y.

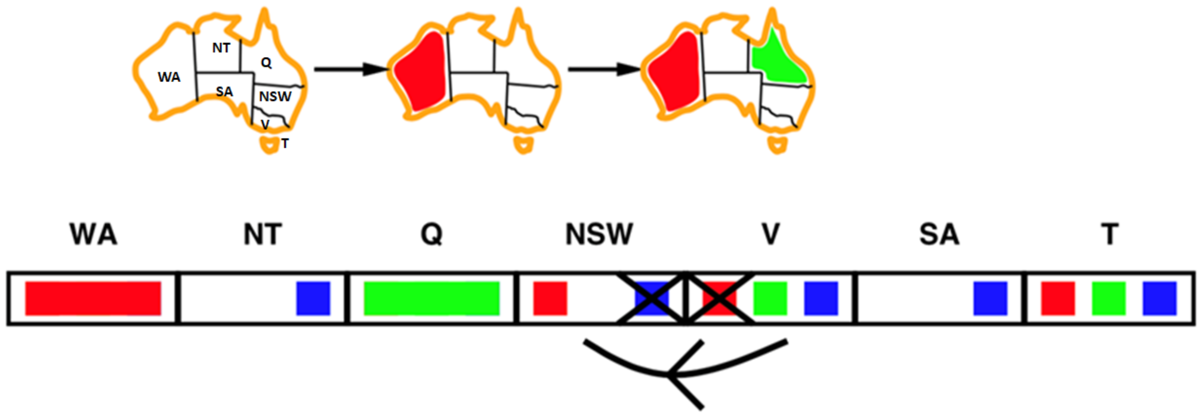
In our example:



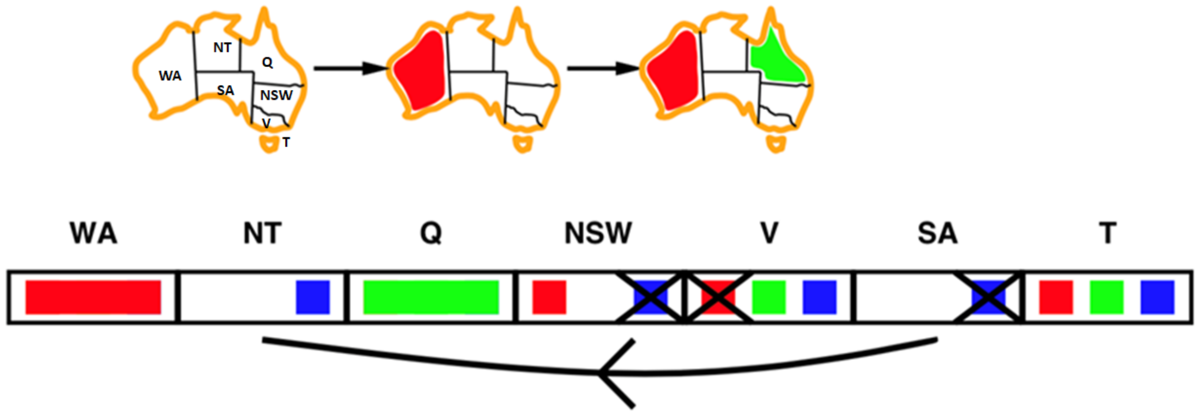
This arc is consistent; because if we color SA blue, we can color NSW red.



This arc can be made consistent by deleting blue from possible NSW values; because if we choose blue as the color of NSW, the SA color will be left with no possible values.



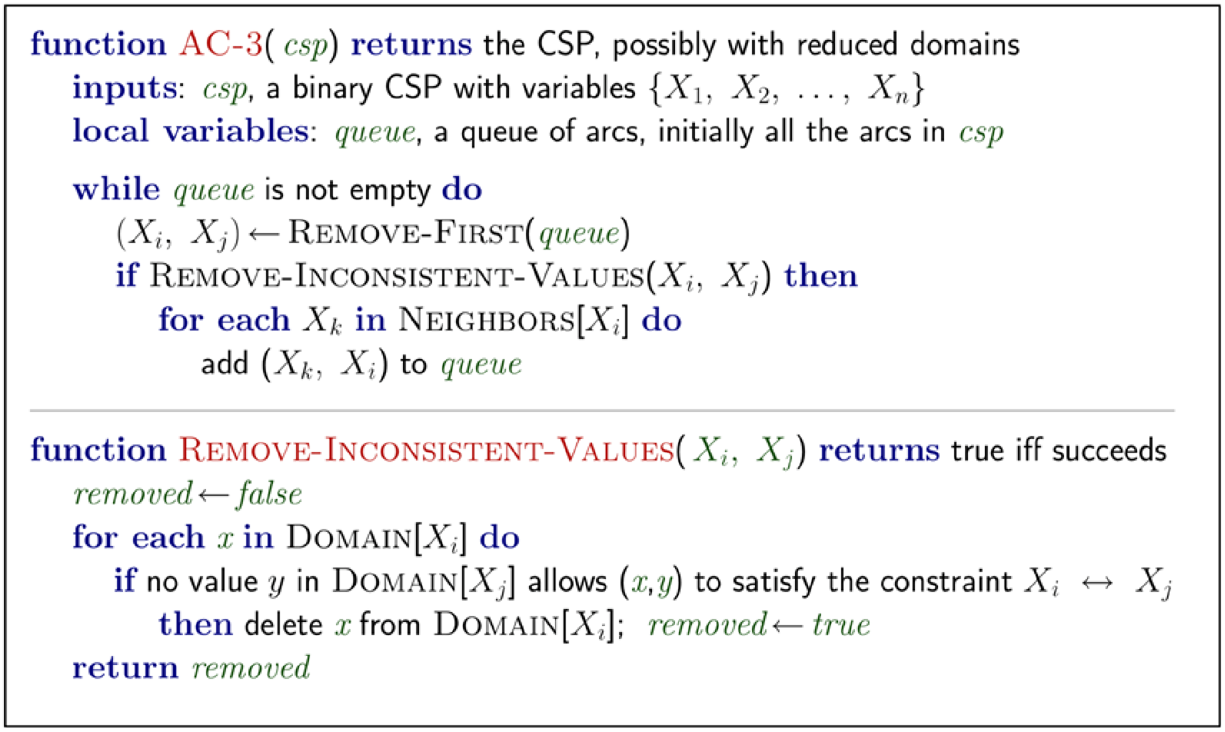
This arc can also be made consistent by deleting red from possible V values.



As we are trying to make this arc consistent, we end up with a state with no possible values and failure is detected.

In arc consistency, notice that if X loses a value, neighbors of X need to be rechecked. Arc consistency can be run as a preprocessor or after each step to check for failure.

Arc Consistency can be checked by an algorithm named AC-3.



```
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
```

AC-3 has a queue of arcs that we need to check the constraints on. In every step, it takes the first one and tries to make it consistent. If some values of X_i are deleted all the arcs that end in X_i are added to the queue to be checked later on.

Time order of the AC-3 algorithm is $O(n^2d^3)$ where n is the number of variables and d is the domain size. To calculate this order, we first notice that each arc is added into the queue if the possible values of its ending node is changed; the ending node can have d values so it is changed at most d times. The order of the number of arcs is n^2 and each constraint processing in Remove-Inconsistent-Values takes $O(d^2)$; so the overall order of the algorithm becomes $O(n^2d^3)$.

The order of the AC-3 algorithm can be reduced to $O(n^2d^2)$.

Arc consistency for n-ary CSP

All that was discussed was for binary problems, what do we do with n-ary problems? As stated before, change the n-ary problem to a binary one and check arc consistency on the binary equivalent.

Bound Propagation

If the domain for each variable is large, Bound Propagation is used.

A CSP is bound-consistent if for every constraint involving X and Y, and for both the lower-bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y.

For an example:

Let's say F_1 is an integer in [0, 165] and F_2 is similarly in [0, 385].

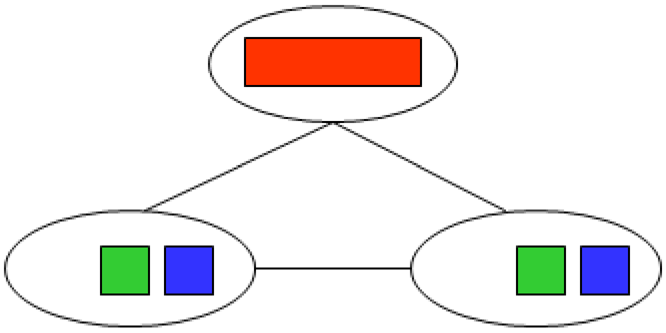
Let's assume we have the constraint $F_1+F_2 = 420$.

To make it consistent, the domains should be changed to [35, 165] and [255, 385].

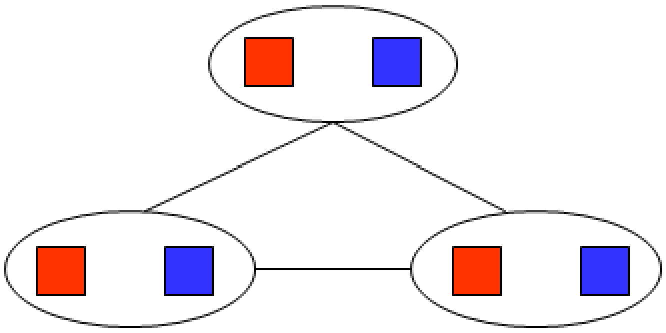
This kind of bounds propagation is widely used in practical constraint problems.

Limitations of Arc Consistency

Let's look at two examples:



In this example, arc consistency doesn't detect failure.



In this example, arc consistency doesn't detect failure as well; but is it really not a failure?

As we can see there are failures that arc consistency can not detect; therefore, after enforcing arc consistency: 1.Can have one solution left. 2.Can have multiple solutions left. 3.Can have no solutions left (and not know it)

The second limitation is arc consistency still runs inside a backtracking search.

To solve these limitations, we look at k-Consistency.

K-Consistency

Stronger form of propagation is 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.

It is good to notice that:

1. 1-consistency (node consistency): each single node's domain has a value which meets that node's unary constraints.
2. 2-consistency (arc consistency): for each pair of nodes, any consistent assignment to one can be extended to the other.
3. 3-consistency is called path consistency.
4. For higher k, our computing becomes more expensive.

Strong k-Consistency

A CSP is strong k-Consistent if it is k-consistent and is also (k − 1)-consistent, (k − 2)-consistent, . . . all the way down to 1-consistent.

Now suppose we have a CSP with n nodes and make it strong n-consistent. We can then solve the problem with performing the following actions:

- Choose any assignment to any variable.
- Choose a new variable.

  By 2-consistency, there is a choice consistent with the first
- Choose a new variable.
- By 3-consistency, there is a choice consistent with the first 2.
- …

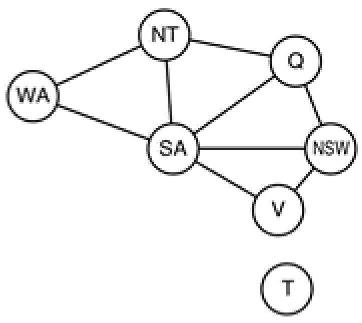
We are guaranteed to find a solution in time $O(n^2 . d)$.

any algorithm for establishing n-consistency must take time exponential in n in the worst case. Worse, n-consistency also requires space that is exponential in n. The memory issue is even more severe than the time.

Problem Structure

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly.

Divide problem into independent subproblems: Looking again at the constraint graph for Australia, we see that Tasmania is not connected to the other lands Intuitively, it is obvious that coloring Tasmania and coloring the others are independent subproblems means that, any solution for the other lands with any solution for Tasmania yields a solution for the whole map. Independence can be ascertained simply by finding connected components of the constraint graph.

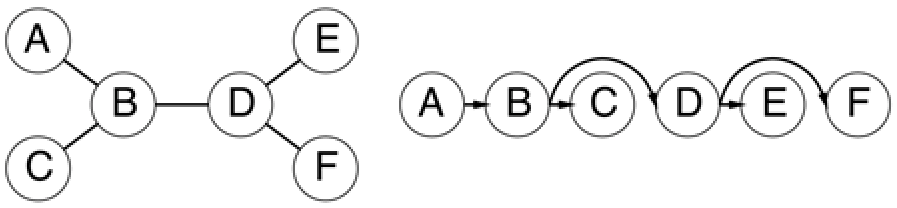


If our constraint graph has k independent connected graph with c variables ($k = \frac{n}{c}$), we can show that the whole problem can be solved in where d is the size of the domain. Which is linear in n; whereas time complexity of the total problem without decomposition is exponential in n.

Tree-structured CSPs

Completely independent subproblems are good, then, but rare. Fortunately, some other graph structures are also easy to solve. For example, a constraint graph is a tree when any two variables are connected by only one path.

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and then choose an ordering of the variables such that each variable appears after its parent in the tree (topological sort).



we can make this graph arc-consistent in $O(n\,.\,d^2)$.

When we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value. Since each link from a parent to its child is arc consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child.

```
function TREE-CSP-SOLVER( csp) returns a solution, or failure
   inputs: csp, a CSP with components $X$, $D$, $C$

   $n \leftarrow$ number of variables in $X$
   assignment $\leftarrow$ an empty assignment
   root $\leftarrow$ any variable in $X$
   $X \leftarrow$ TOPOLOGICALSORT($X$, root)
   for $j = n$ down to 2 do
      MAKE-ARC-CONSISTENT(PARENT($X_j$), $X_j$)
      if it cannot be made consistent then return failure
   for $i = 1$ to $n$ do
      assignment[$X_i$] $\leftarrow$ any consistent value from $D_i$
      if there is no consistent value then return failure
   return assignment
```

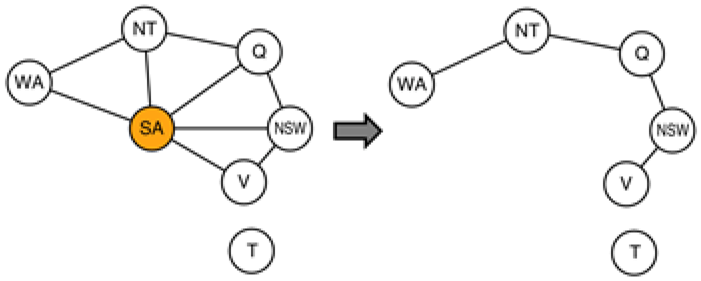
Nearly tree-structured CSPs

Now, we have a good algorithm for tree-structured CSP. Consider if we could reduce other constraint graphs to trees somehow.

One way to do this, is to remove some nodes and their vertices to make the constraint graph a tree.

Consider the constraint graph for Australia, if we delete South Australia, the graph would become a tree. Fortunately, we can do this by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

In the general case, the value chosen for SA could be the wrong one, so we would need to try each possible value.



The general algorithm is as follow:

- Choose a subset of variables such that after removing, the constraint graph becomes a tree. We call this subset a cycle cutset.
- For each assignment of S (current cutset) variables that satisfies all constraint on S:
    1. Remove any values from domains of remaining variables that are inconsistent with values assigned for S.
    2. If the remaining tree-structured CSP has a solution, return it with an assignment for S.

If cutset has size c, total time complexity of algorithm becomes $O(d^c.(n-c)^d)$ where, d^c is for all assignments of cutset and $(n-c)^d$ is for solving the remaining tree.

Tree decomposition

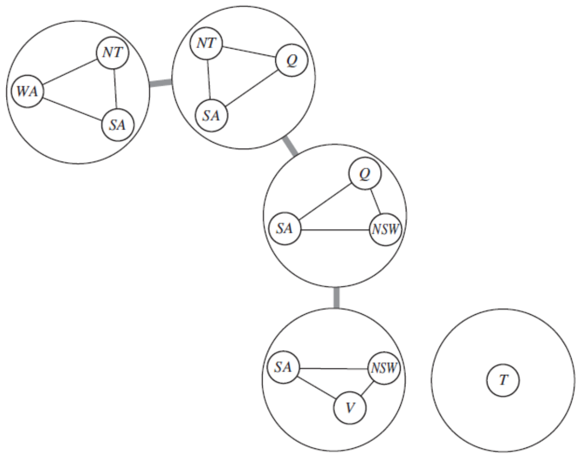
This approach is based on constructing a tree decomposition of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting solutions are then combined.

This approach should follow these requirements:

- Every variable in the main problem should appear at least in one of the subproblems.
- If two variables are connected in a constraint graph, they should appear at least in one of the subproblems.
- If a variable appears in two subproblems, it must appear in every subproblem along the path connecting those subproblems.

The first two requirements ensure that all constraints participate in decomposition and the third part somehow ensures that all same variables in different subproblems must have the same value.

Figure below shows the tree decomposition for the map coloring problem.



We can solve the problem using tree decomposition with following these steps:

- Solve each subproblem independently, if any one has no solution, we know that the main problem has no solution.
- Name each subproblem a mega-variable whose domain is all solutions for solving subproblem

- Solve the constraints connecting the subproblems, using the algorithm for trees given earlier. Constraints between subproblems ensure that all same variables in different subproblems must have the same value.

We can define tree width for tree decomposition of a graph that means one less than the size of the largest subproblem in tree decomposition. Also, tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions. If a graph has tree width w and we are given the corresponding tree decomposition, then the problem can be solved in time.

Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

Iterative algorithms for CSPs

Local search algorithms such Hill-climbing and simulated annealing can be effective in solving CSPs.

For solving CSP in this way we should follow steps below:

1. Assign a value to every variable without considering the constraints.
2. If the current assignment doesn't satisfy all constraints, choose a random variable to change its value.
3. Choose a new value for the selected variable based on a heuristic.
4. Go back to step2 on continue the algorithm.

The common heuristics that are used for this algorithm is to select the value that results in the minimum number of conflicts with other variables. This heuristic is called min-conflicts.

Next figure shows the local search algorithm for CSP.

```
function MIN-CONFLICTS(csp, max_steps) returns a 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 csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
     set var = value in current
  return failure
```

For example, for 4-Queens problem we have:

- Initial state: four length string that shows queen's row position in columns.
- Operators: change queen's row in column.
- Goal test: no attack.
- Heuristic: number of attacks.

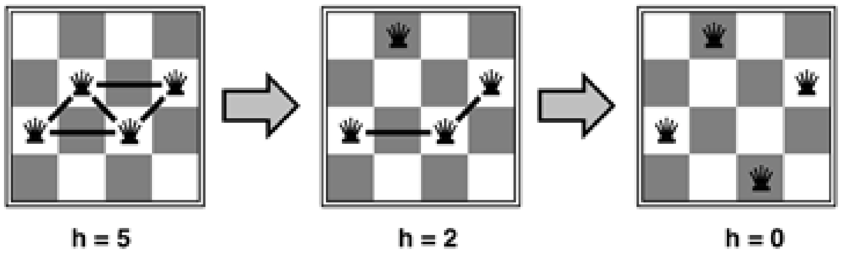
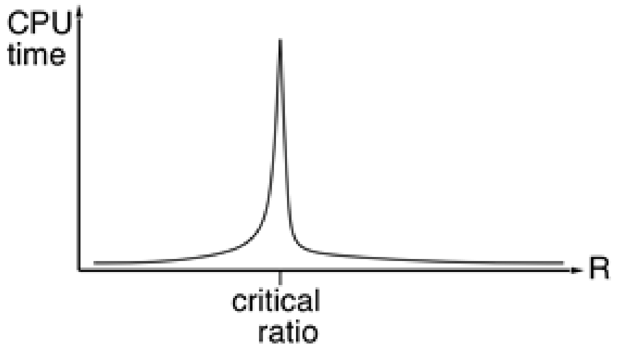


Figure below shows performance of min-conflicts, as it's clear, if in our problem, R is near to *critical ratio,* the algorithm doesn't converge to the goal state.

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



Summery

- CSPs are a special kind of problem:
    - states defined by values of a fixed set of variables.
    - goal test defined by constraints on variable values.
- Backtracking depth-first search with one variable assigned per node.
- Variable ordering and value selection heuristics help significantly.
- Forward checking prevents assignments that guarantee later failure.
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies.
- The CSP representation allows analysis of problem structure.
- Tree-structured CSPs can be solved in linear time.
- Iterative min-conflicts are usually effective in practice.

References

1. Artificial Intelligence Course, Dr.MH Rohban, Department of Computer Engineering, Sharif University of Technology (Spring 2021)
2. Artificial intelligence: A modern approach, 3rd edition (16 Feb. 2010)