Constraint Satisfaction Problems

Table of Contents

[Examples 3](#_Toc138878542)

[Map Coloring 3](#_Toc138878543)

[N Queens Problem 4](#_Toc138878544)

[Constraint Graphs 6](#_Toc138878545)

[Examples 7](#_Toc138878546)

[Cryptoarithmetic 7](#_Toc138878547)

[Sudoku 8](#_Toc138878548)

[Varieties of CSPs and Constraints 8](#_Toc138878549)

[CSPs in the Real World 9](#_Toc138878550)

[Solving CSPs 10](#_Toc138878551)

[Backtracking Search 11](#_Toc138878552)

[Improving Backtracking 12](#_Toc138878553)

[Filtering 12](#_Toc138878554)

[Ordering 16](#_Toc138878555)

So far, we have been looking into **search problems**. These problems make a few assumptions:

* There is only **one agent**, i.e., only Pac-Man exists, no ghosts.
* Actions are **deterministic**, meaning there is no probability involved.
* The entire state space is **known beforehand**.
* The states are **discrete**, not continuous.

Making these assumptions allow us to make **plans**, which are the sequence of actions we take to reach our goal. The thing we were concentrating on was the path, which could involve different costs, depths or heuristics. Once we decided on the path, we executed the actions in the exact same way.

In **Constraint Satisfaction Problems** (CSPs), the important thing is the goal itself, not the path used to reach the goal. It is a subset of a larger class of problems called **identification problems**, which involved assigning values to variables.

In search problems, we treated the contents of each state as a **black box**, only testing the goal function without concerning ourselves with the state. The successor function could also be anything, since we only cared about the output of the function. In CSPs, which are also a special subset of search problems, we will be using additional information from the states.

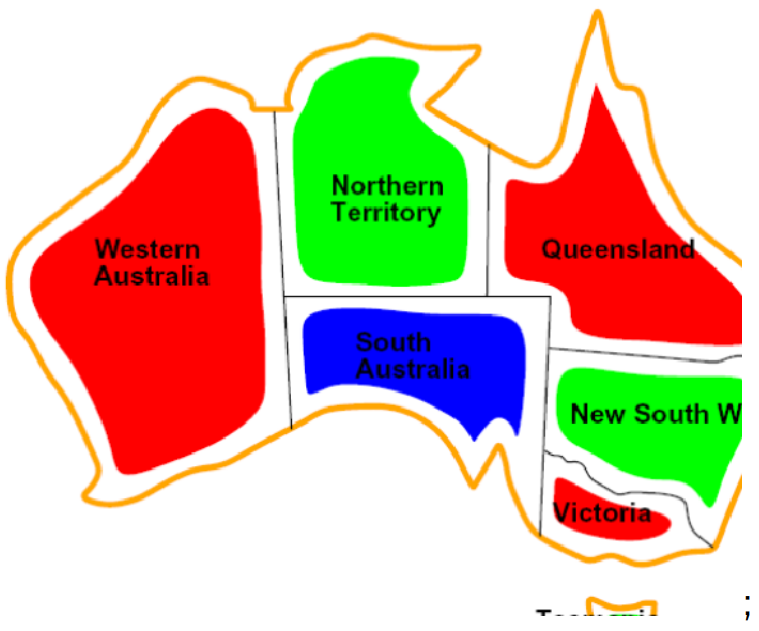
Each state is defined by some **variable** , which can take values from the **domain** , where may or may not depend on . The goal test is a **set of constraints** which specify the allowable values for the variables. The algorithm will assign values to the variables and check if the constraints are being met. This is an improvement over search problems, since we can gain a sense of whether we are heading in the correct direction by checking how many of the constraints are met in the current state, unlike the search problem goal test which was a yes or no answer only.

CSPs are a simple representation of **formal representation languages** and allow us to create useful general-purpose algorithms that are more powerful than standard search algorithms.

## Examples

### Map Coloring

The **Map Colouring** problem specifies that, given a map of different states, we need to assign colours to the states from a specified list such that no adjacent states have the same colour.



We can formulate the problem as follows:

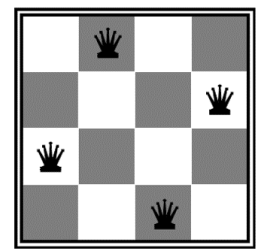
* **Variables** – The different states, , , , , , ,
* **Domain** -
* **Constraints** – Adjacent regions must have different colours.

The constraints can be of two types, implicit and explicit. **Implicit constraints** are defined in terms of only the variables. E.g., is an implicit constraint, since they are adjacent states and cannot have the same colour. Generally, implicit constraints are specified by the problem definition itself. **Explicit constraints** are defined in terms of both the variables and their values. For example, if for some reason we had a constraint that said , this would be an explicit constraint.

* **Solutions** – These are assignments that satisfy all the constraints. For example, is a possible solution.

## N Queens Problem

The **N Queens Problem** states that we have an board on which we have to place queens (the chess pieces) such that none of them are in a position to be able to attack each other.



Every CSP can be formulated in multiple ways, but some formulations are better than others. We will explore two formulations for this problem to contrast the differences between them.

The first formulation is as follows:

* **Variables** - , the board positions
* **Domain** – {, used to indicate whether the position is occupied or not
* **Constraints** –
  + , meaning two cells in the same row cannot be occupied
  + , meaning two cells in the same column cannot be occupied
  + , meaning two cells in the same diagonal cannot be occupied.
  + , which means the same as the above, just in the other diagonal direction.
  + , meaning the board must have queens on it

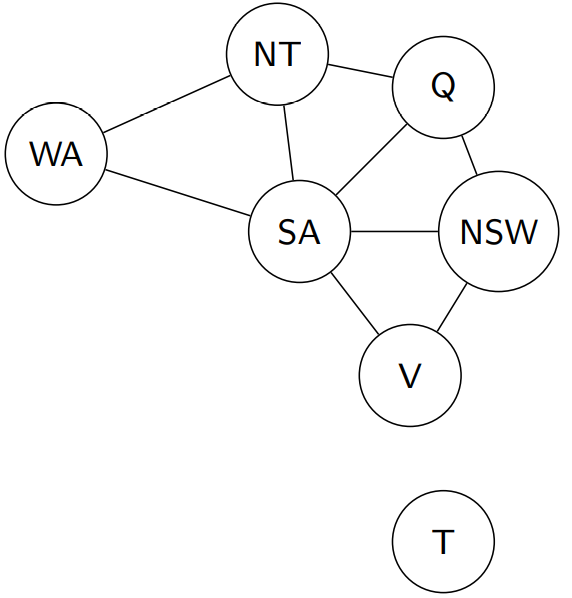
The second formulation is as follows:

* **Variables** – , the row numbers
* **Domain** - , the cell number in a row occupied by a Queen
* **Constraints** –
  + All of the constraints from the previous formulation (implicit constraints)
  + , specify the valid combinations (explicit constraints)

The second formulation is actually better than the first. This is because the first formulation has variables which the second has variables. The more variables we have, the more complex our problem will become.

## Constraint Graphs

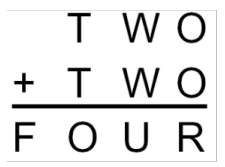
**Constraint Graphs** allow us to visualize the relationship between the variables in a CSP. The different variables are represented by nodes and each constraint is shown as an edge. For example, the graph below has an edge between states that share a border.



General purpose CSP algorithms use constraint graphs to speed up their search. For example, the graph above tells us that is independent, so we can assign any colour to it and be done with it.

## Examples

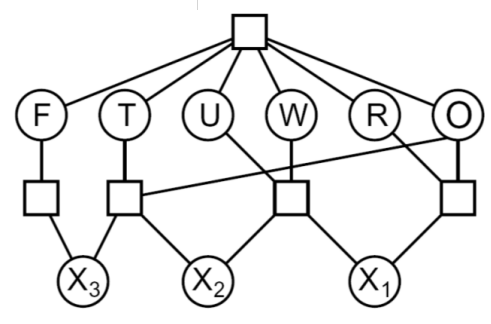
### Cryptoarithmetic



The **Cryptoarithmetic** problem states that alphabets are used in place of digits in a math problem and the correct values for each alphabet must be found so that the math problem is correct.

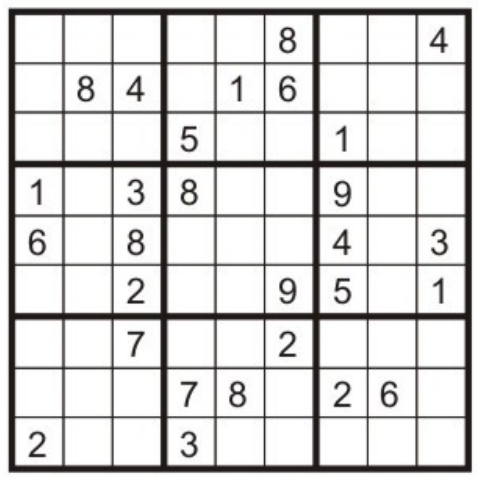
* **Variables** – , , , , , , , , , where the last three variables are for the carry bits of the lower three digit positions.
* **Domains** -
* **Constraints** –
  + All variables values must be different

The constraint graph for this problem looks like this:



Notice the square blocks used in the graph. When a large number of variables are involved in a single constraint, we can use these blocks as a joining point.

## Sudoku



* **Variables** – Each open square
* **Domain** –
* **Constraints** –
  + Each column must not repeat any digits
  + Each row must not repeat any digits
  + Each region must not repeat any digits
  + Pairwise inequalities (explicit constraints)

## Varieties of CSPs and Constraints

CSPs and their constraints can be of several types.

The **variables** can be either discrete or continuous. **Discrete variables** have either a finite or infinite domain. For a finite domain, there are a fixed number of possible values. For variables with possible values, there can be assignments. For example, Boolean CSPs have discrete variables with a finite domain. For an infinite domain, we have things like job scheduling, where the variables are the start and end times of each job. These problems are solvable for linear constraints, but undecidable for non-linear constraints.

**Continuous variables** are used with things such as observations for the Hubble telescope. These problems are solvable in polynomial time if the constraints are linear.

The **constraints** can be **unary**, which involved a single variable, **binary**, which involve two variables, or **higher-order**, which involve more variables. There is also a special type of constraint called a **soft constraint**, which are things that are preferable but not required to solve the problem. For example, we can prefer for a state to be given a certain colour in the map colouring problem.

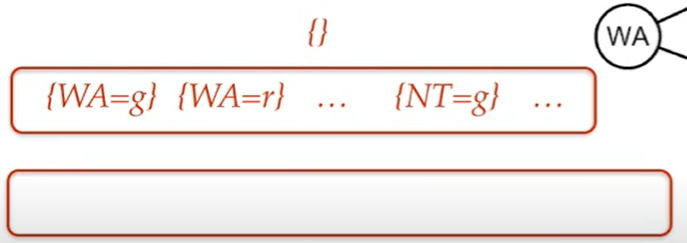
## CSPs in the Real World

* Scheduling problem
* Timetabling problem
* Assignment problem
* Hardware configuration
* Transportation scheduling
* Factory scheduling
* Circuit layout
* Fault diagnosis

## Solving CSPs

To solve CSPs, we will be using traditional search algorithms. This requires converting the CSP problem to a search problem. The **states** will be defined by the values that have been assigned so far, the **initial state** will be the one with no assigned values, the **successor function** will involve assigning a value to an unassigned variable and the **goal test** will involve checking if the current assignments are complete and satisfy all the constraints.

Let’s start by using BFS and DFS. If we apply **BFS** to the map colouring problem, it will assign values to variables layer by layer. The first layer will be empty assignments, the second layer will contain a separate node for each possible variable and value combination, the second layer will be the same minus the node already assigned and so on.



Once the last layer is reached, BFS will check against the constraints. This takes a huge amount of time.

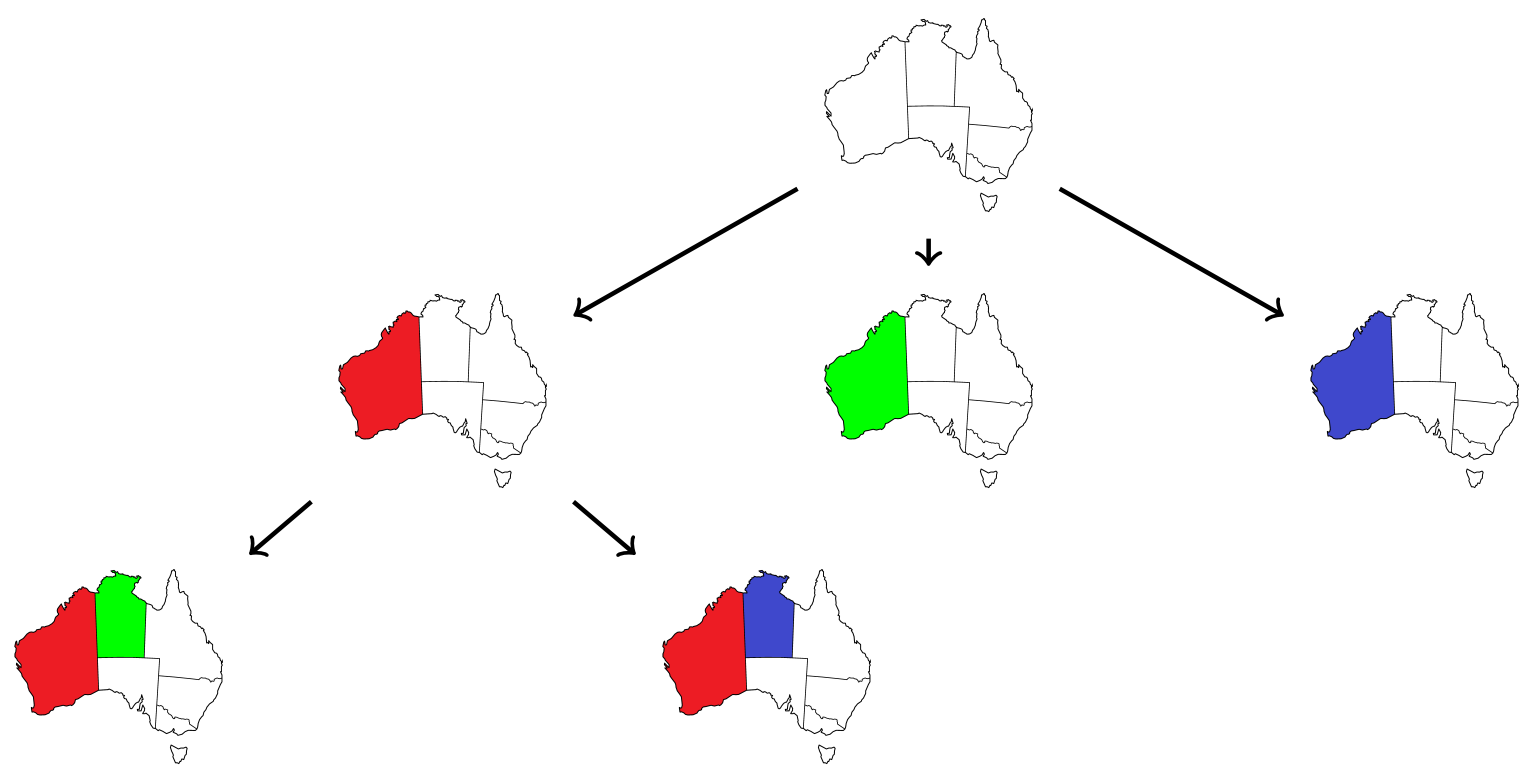
**DFS** on the other hand will explore each branch to its full length instead of going layer by layer. As such, it will get to the point of being able to check against the constraints faster. Overall, this makes DFS better for this problem.

However, both solutions are still slow. This is because they are both checking against the constraints only when all possible variables have been assigned some value. The entire point of a CSP is that we can check against constraints individually and detect invalid assignments way before we assign all the values. These methods are not taking advantage of this.

## Backtracking Search

**Backtracking Search** is an implementation of **DFS** with two additional ideas implemented. The first is that the variables are assigned values one at a time. At each step, we are only dealing with a single variable. The second is that constraints are checked on the go. When assigning a value to a variable, only those values are considered that do not conflict with previous assignments. One issue here is that there are some resources being used to check for conflicts.

For example, in the search tree below, backtracking search checks for two values instead of three in the second layer. This is because the third value (green) is not valid at this point.



function BACKTRACKING-SEARCH(csp) returns a solution, or failure

return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure

if assignment is complete then return assignment

var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)

for each value in ORDER-DOMAIN-VALUE(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

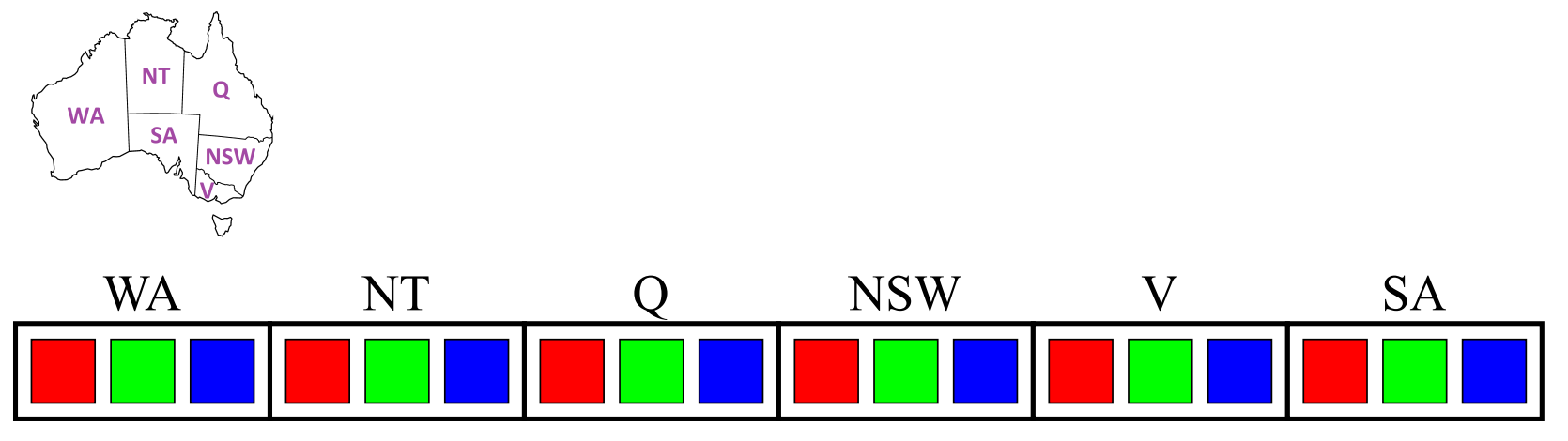
PSEUDOCODE

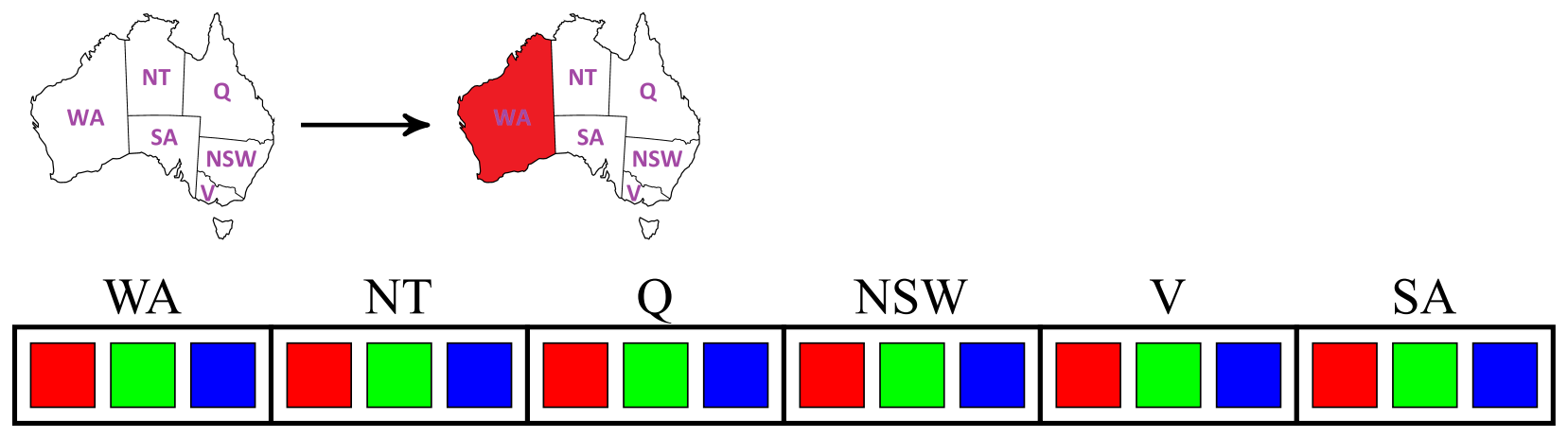
## Improving Backtracking

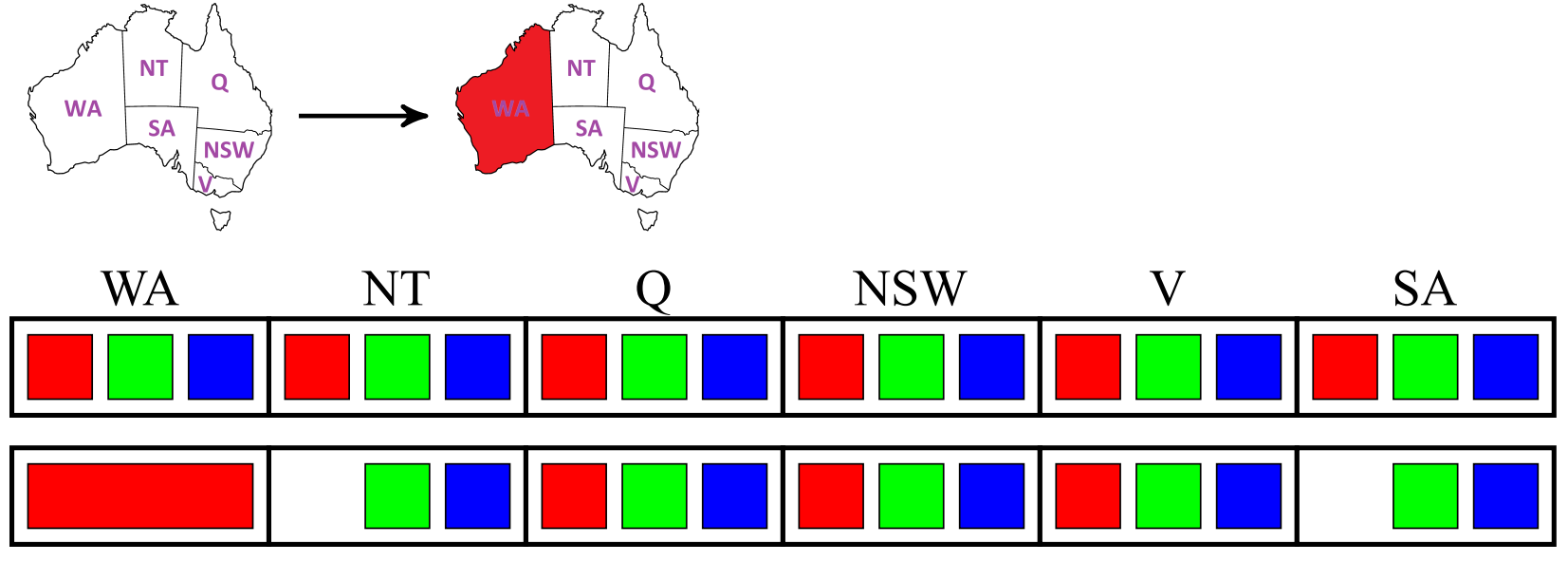
It is possible to improve the backtracking algorithm using three more ideas, filtering, ordering and scheduling. In **filtering**, we try to detect inevitable failures early on, before we even assign values to all the variables. In **ordering**, we adjust which variables we assign in what order and also the order of choosing values to assign. In **scheduling**, we attempt to exploit the problem structure.

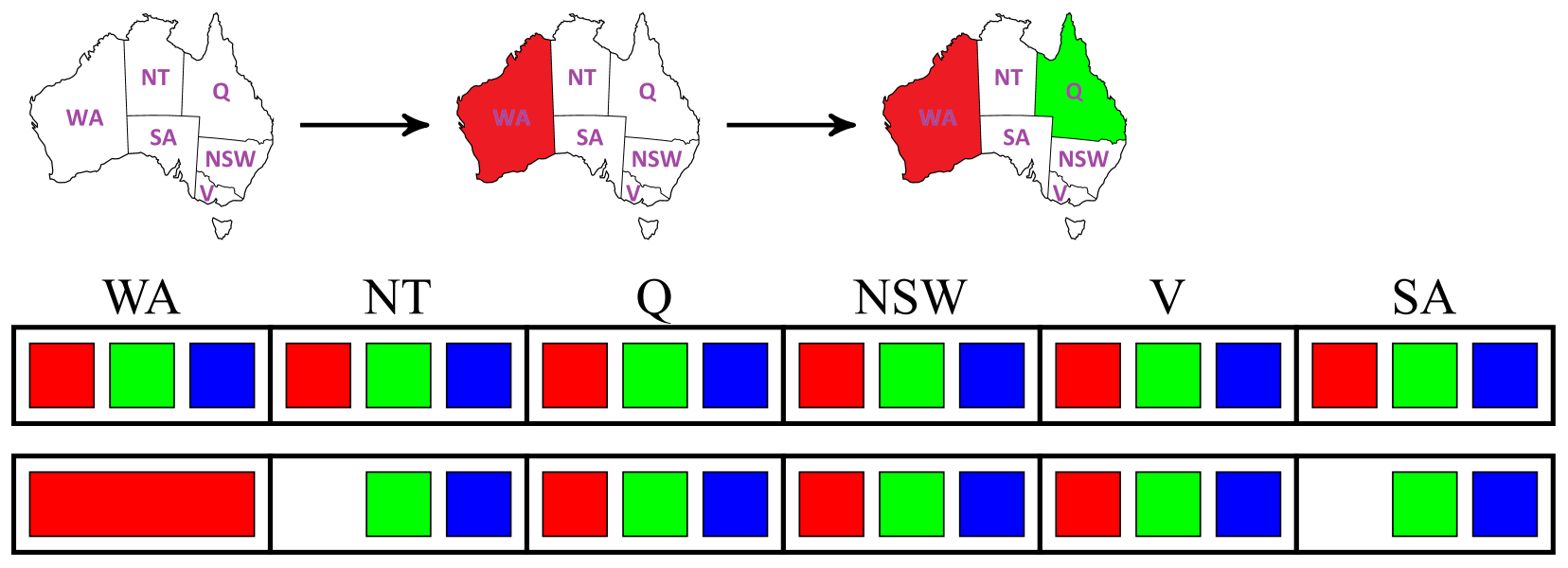
### Filtering

The first filtering mechanism is called **forward checking**. Here, we assign some valid value to one variable, and for every other variable that has a constraint relationship with the current variable, we remove any invalid values from their domains. If a variable has an empty domain, we immediately know that the path we are heading down will not lead to a solution.







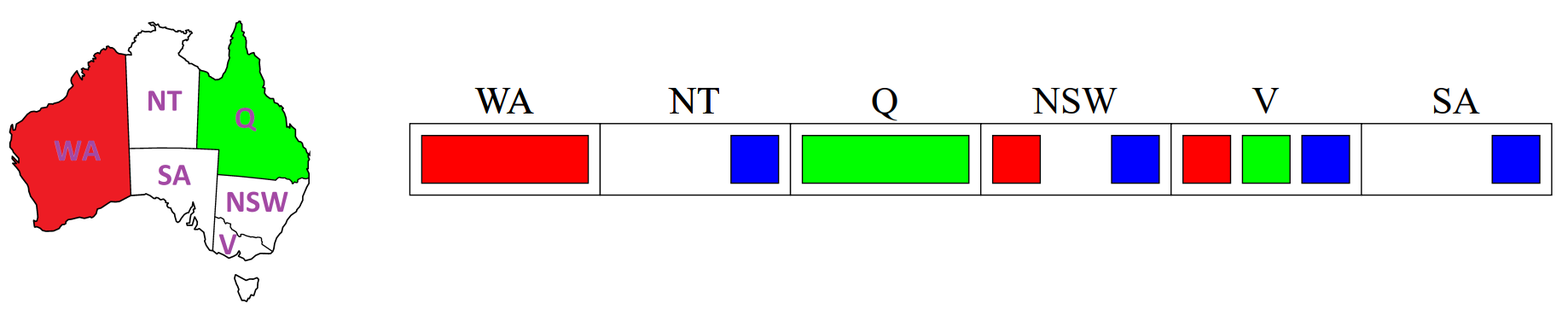


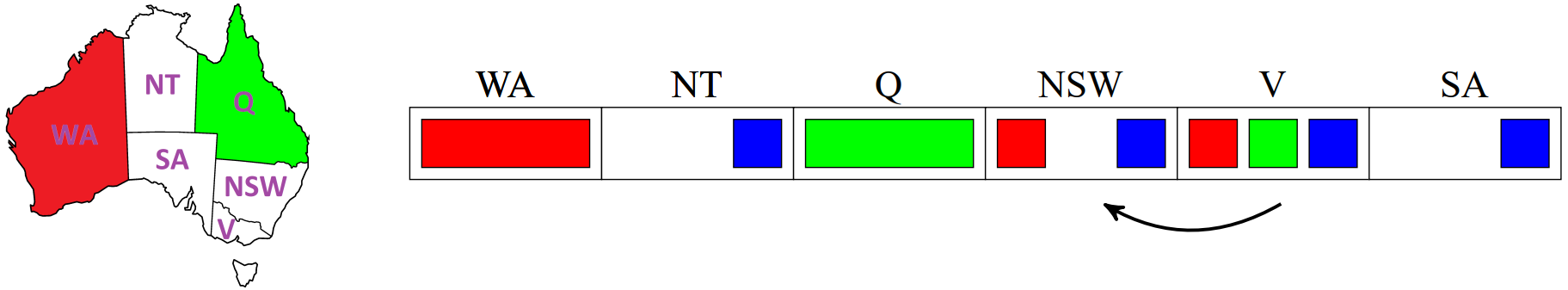


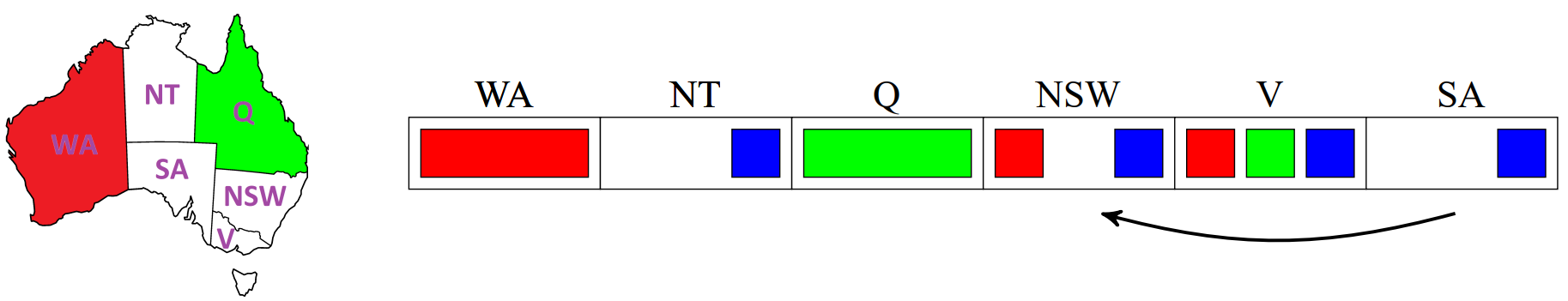
One major issue with forward checking is that it only checks the domains of the immediate neighbours of the variable being assigned a value. However, we can have situations where there are conflicts between other variables due to the new assignments. Analysing these conflicts can allow us to detect failures even earlier.

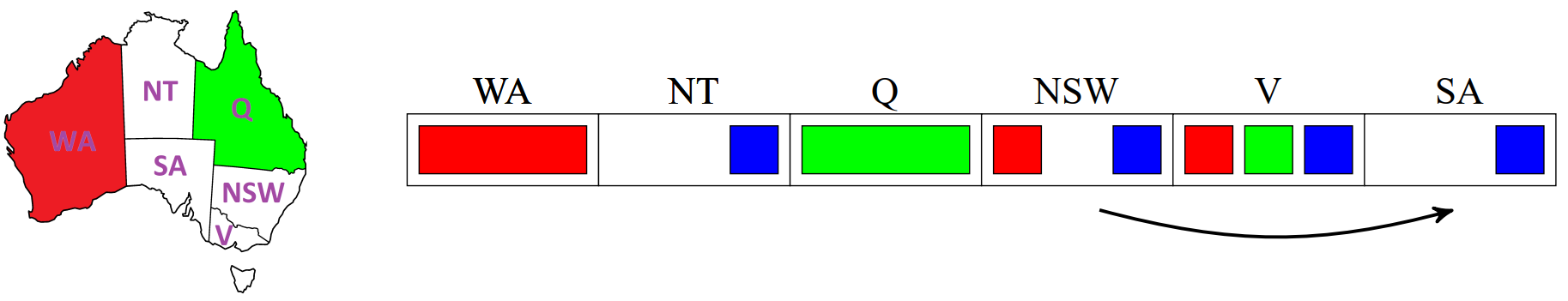
For example, at the point we stopped the algorithm in the images above, the only remaining value in the domains of both NT and SA is ‘blue’. However, these two states are neighbours, so the remaining value breaks a constraint. If we checked for this now, we would be able to know before getting to those variables that the current path will fail. This is called **constraint propagation**.

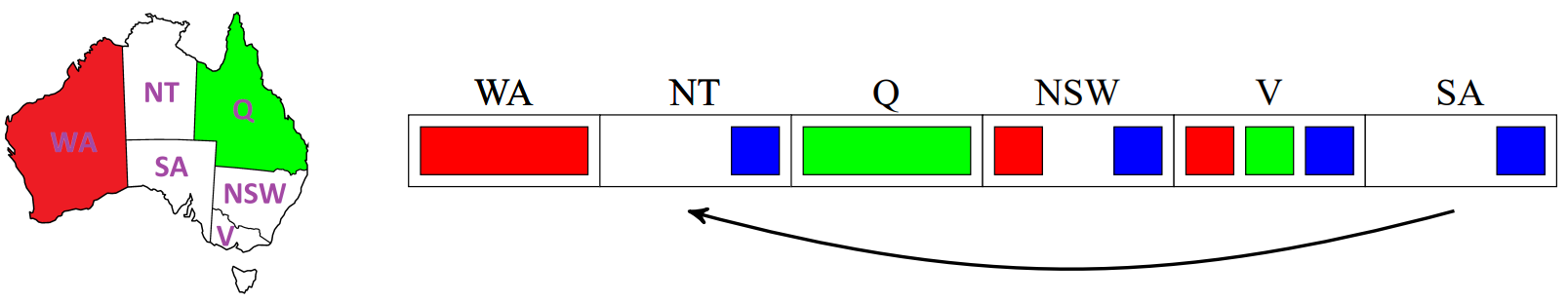
For every pair of variables participating in a constraint, we create an arc from one variable, called the **tail**, to the other variable, called the **head**. For every value in the tail, there must be some value in the head that does not violate the constraint. If there is no such value, the value must be deleted **from the tail**. This process is called checking for **arc consistency**.











At this point, we see that there are no values in NT that can be used if SA uses blue. Thus, we need to **remove blue** from SA. However, this makes the domain of SA empty, which means we cannot find a solution.

Remember that every arc must be rechecked if we remove a value from any variable. This is because that removal could have affected the other constraints.

The downside to this is of course the **overhead** added by this process. Still, for larger problems, the overall algorithm is sped up significantly as a result. We can even run an arc consistency check before we make the first assignment, removing any values that are invalid by default.

Constraint propagation does not guarantee that we will immediately find the solution. We might have one, more or no solutions left and not be aware of the situation. Additionally, we still need backtracking search. This is not a standalone method to solve CSP.

### Ordering

We can order based on the variables or the constraints. For variables, it is a smart choice to pick the variable with the **minimum remaining values**. This allows us to quickly discover if the path fails. If our current series of actions does not lead to a solution, we will fail quickly.

For values, we should choose the **least constraining value**, meaning we pick the value for a variable that cause the least number of values to be removed from other variables. Note that calculating this is resource intensive because we have to rerun filtering. Choosing the least constraining value ensures that the other variables have more options and thus have a higher chance of successfully finding a solution.