Constraint Satisfaction Problems II

Table of Contents

[K-Consistency 2](#_Toc138878493)

[Exploiting the Problem Structure 3](#_Toc138878494)

[Independent Subproblems 3](#_Toc138878495)

[Tree Structures 4](#_Toc138878496)

[Nearly Tree Structures 8](#_Toc138878497)

[Iterative Improvement 8](#_Toc138878498)

[Local Search 10](#_Toc138878499)

[Simulated Annealing 11](#_Toc138878500)

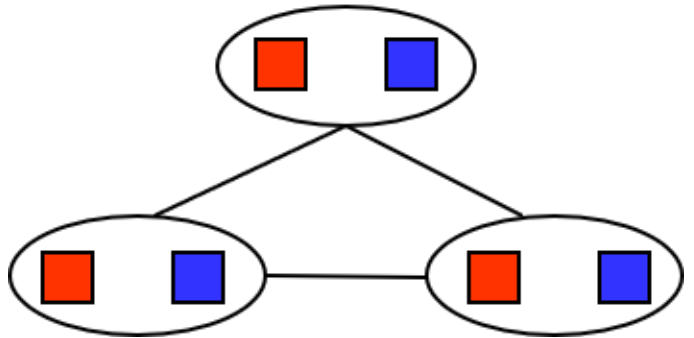
[Genetic Algorithms 12](#_Toc138878501)

To improve upon CSPs, we can exploit several properties of CSPs. We will now be exploring these exploits.

## K-Consistency

Before we can jump into the exploits, we need to understand what **K-Consistency** is. **Consistency** can be of several types. We can have a 1-consistency, where a constraint involves a single node, such as specifying that a state must have a specific colour in the map colouring problem. We can also have 2-consistency, which involves 2 nodes. This is just the **arc consistency** that we already explored.

**K-Consistency** involves nodes. This makes checking for consistencies between groups of nodes more difficult. If we have nodes in total, the time complexity becomes . However, this also ensures the backtracking algorithm makes fewer mistakes. For example, we saw that an arc consistency cannot detect a failure in the situation below, but a 3-consisntency can.

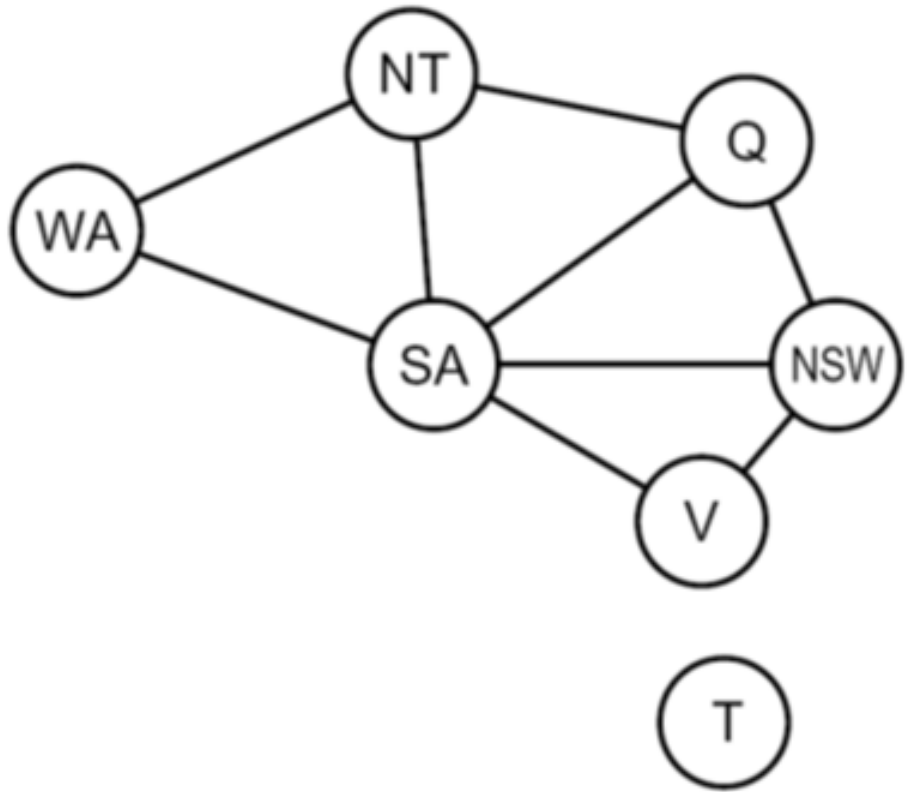


We also have **Strong K-Consistency**, which means that a group is K-consistent, (K-1)-consistent, (K-2)-consistent and so one. If the value of is , Strong K-Consistency can ensure that we solve the problem with **no backtracking**. This is proven in the steps below:

* Choose any assignment to any variable.
* For the next variable, there must be some valid value left to assign due to 2-consistency.
* For the next variable, there must be some valid value left to assign due to 3-consistency.
* …

## Exploiting the Problem Structure

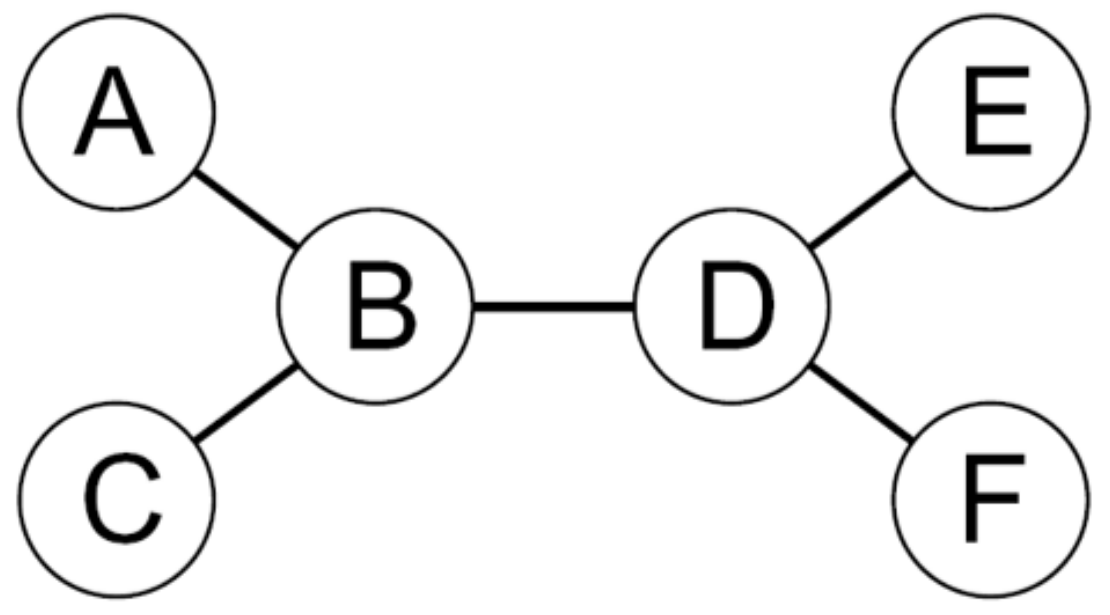
### Independent Subproblems



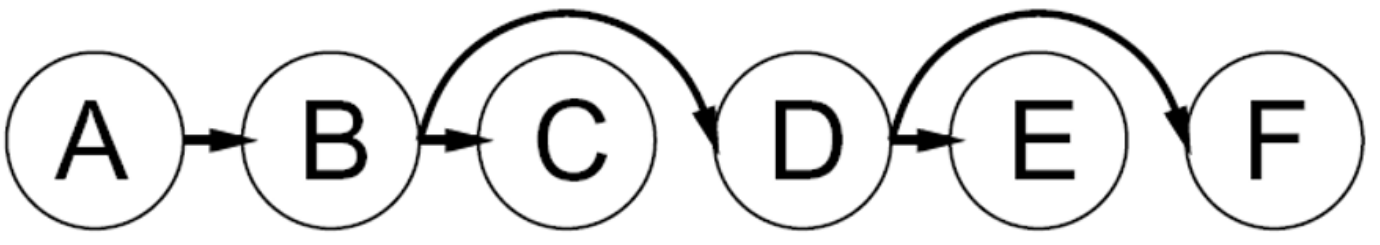
An extreme situation is when we have **independent subproblems**, such as the one shown above. Here, the state Tasmania is separate from the rest of the graph. Having sections in our graph that are not connected to each other makes solving the graph easier because we need to solve two smaller problems instead of one large problem. The DFS algorithm can be used to detect such separations.

If we nave nodes in total and we have groups of nodes each, the total time complexity becomes , compared to if we do not use this exploit. For , and , considering that we have a machine that can check 10 million nodes per second, an problem will require 4 billion years to solve, whereas an problem will require 0.4 seconds to solve.

### Tree Structures

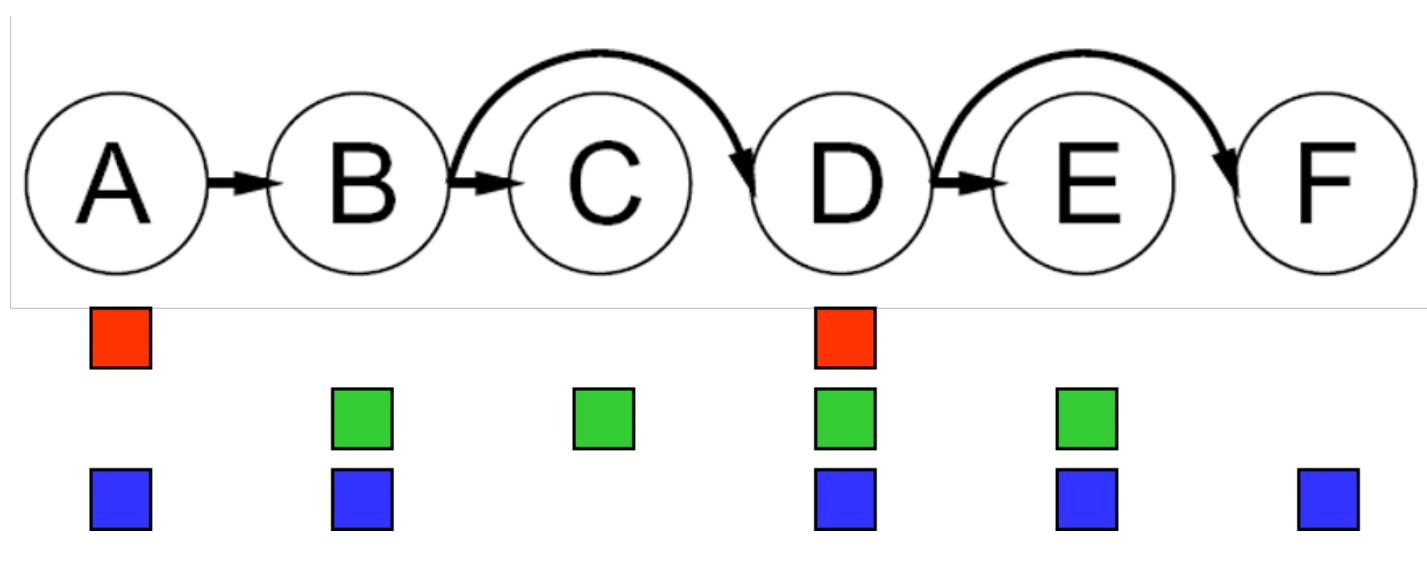


Another relatively rare case is when we have a group with no loops, such as the one shown above. Since there is a single arc per node, the problem can be solved with a time complexity of . Such graphs can be visualized as **trees** by choosing any node as the root. Once a node is chosen, a **topological search** gives the order.

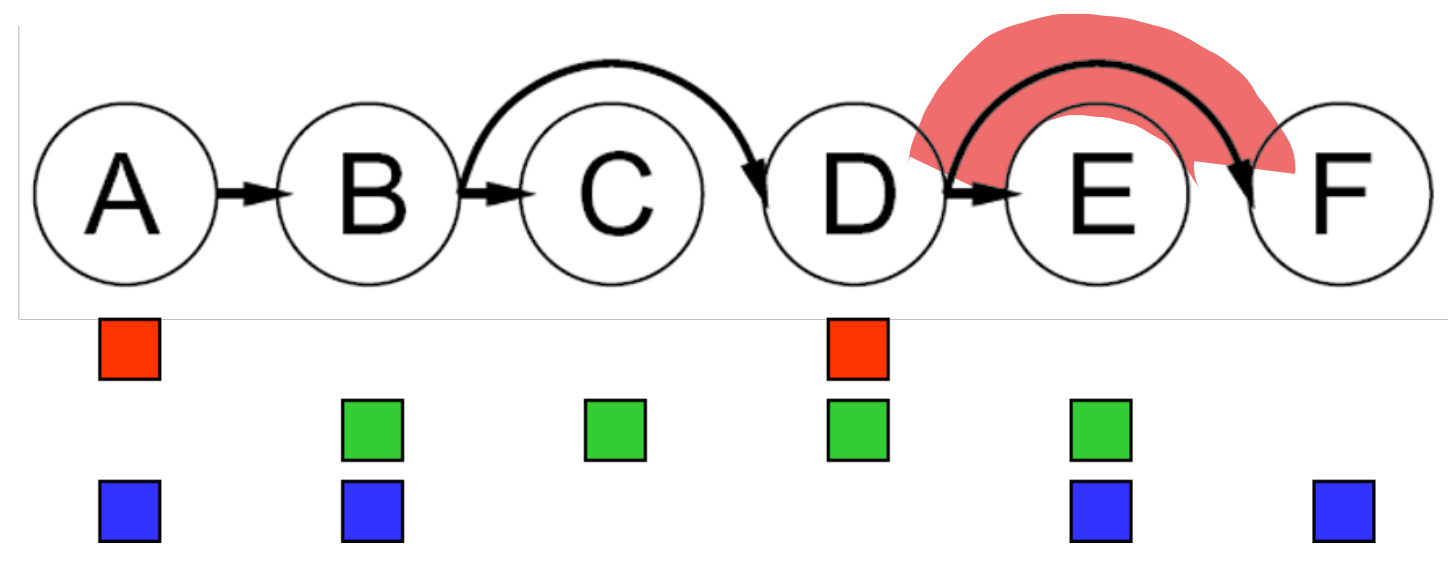


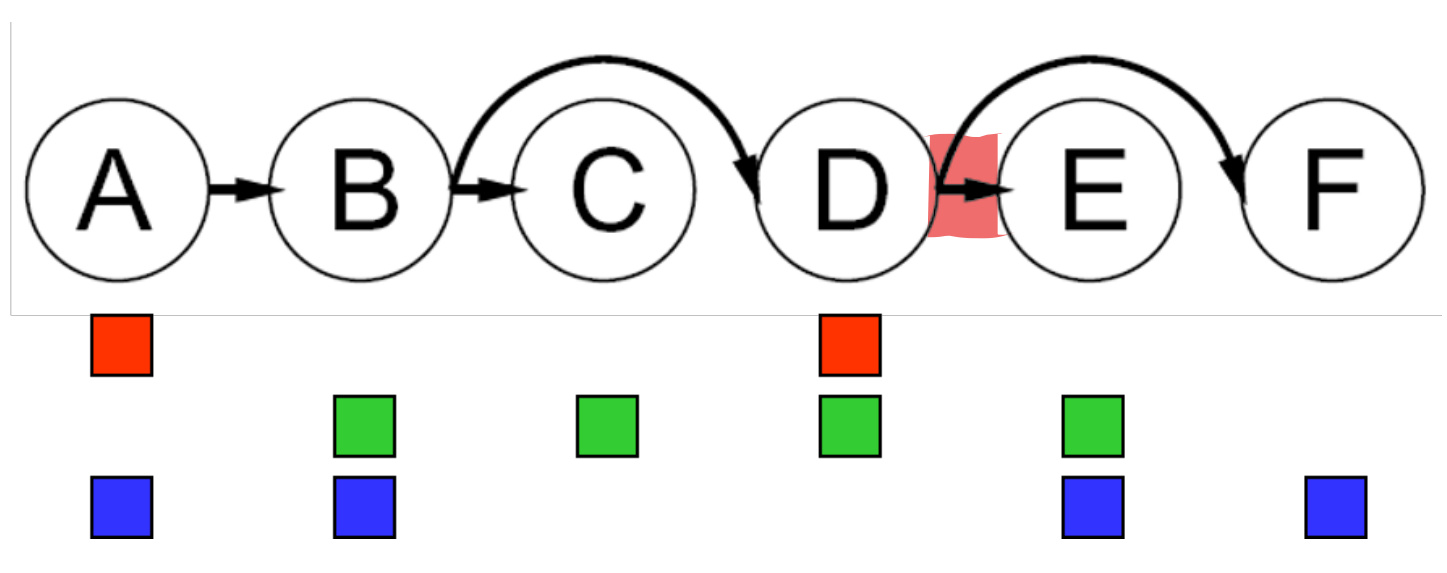
The only reason we are creating a tree shape is so we can give a visual order to things to make it easier to visualize what happens. The actual solution does not require this. To solve the problem, we can now start checking for **arc consistencies** in the backward direction.

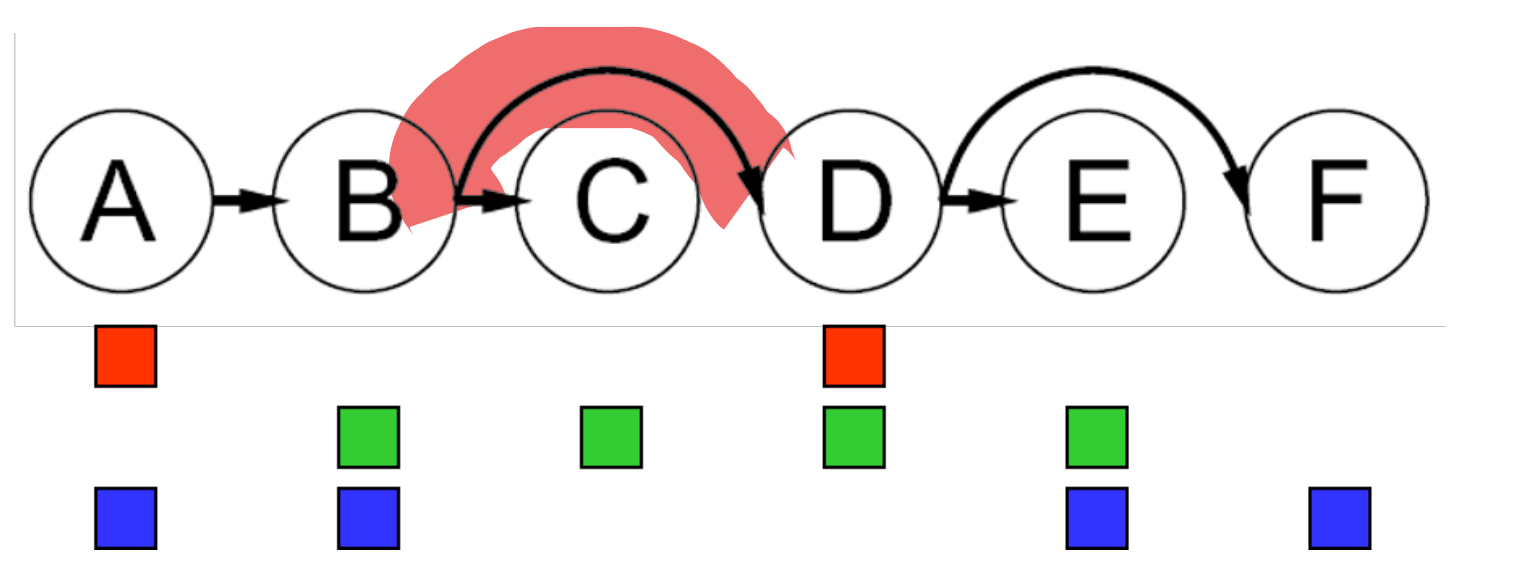
Suppose we start off with a situation like this (for whatever reason):

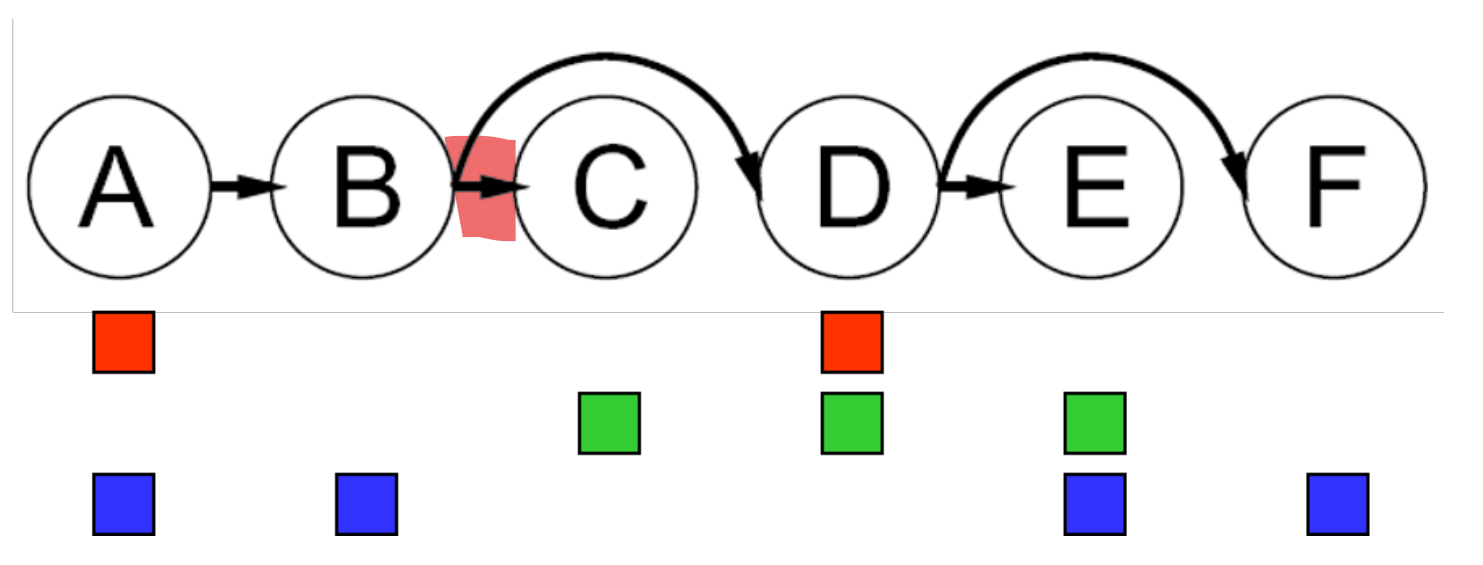


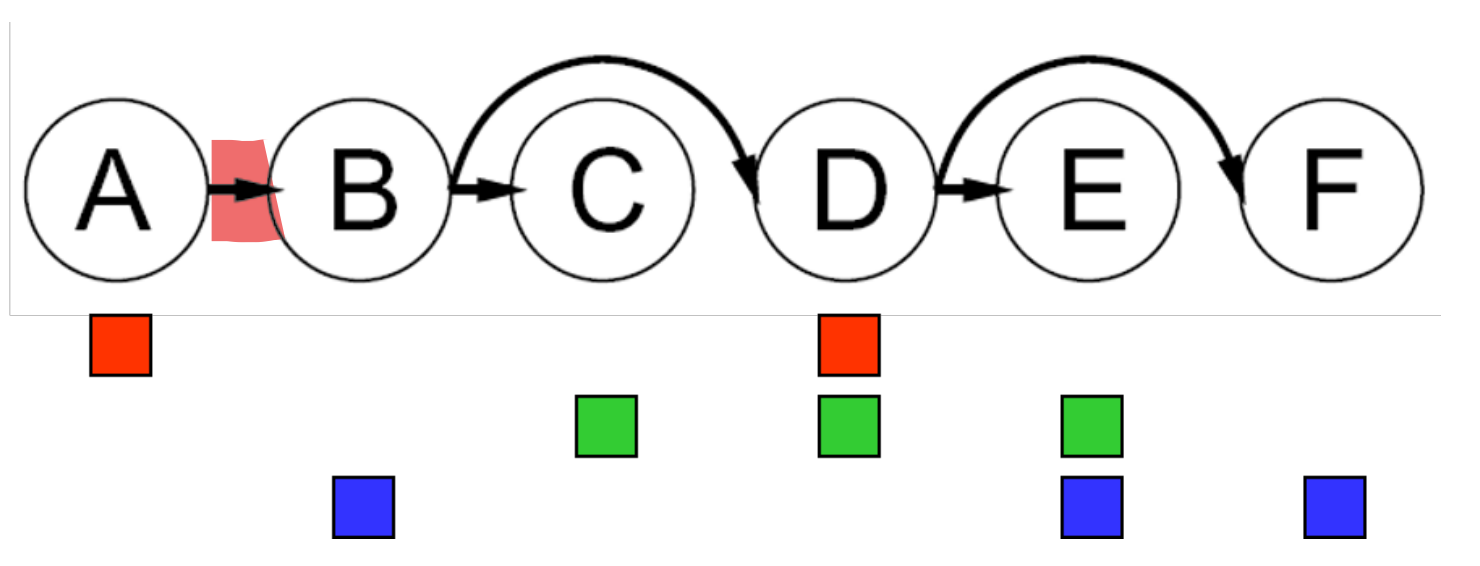
Checking for arc consistencies step by step from the bottom of the tree to the root will be as follows:



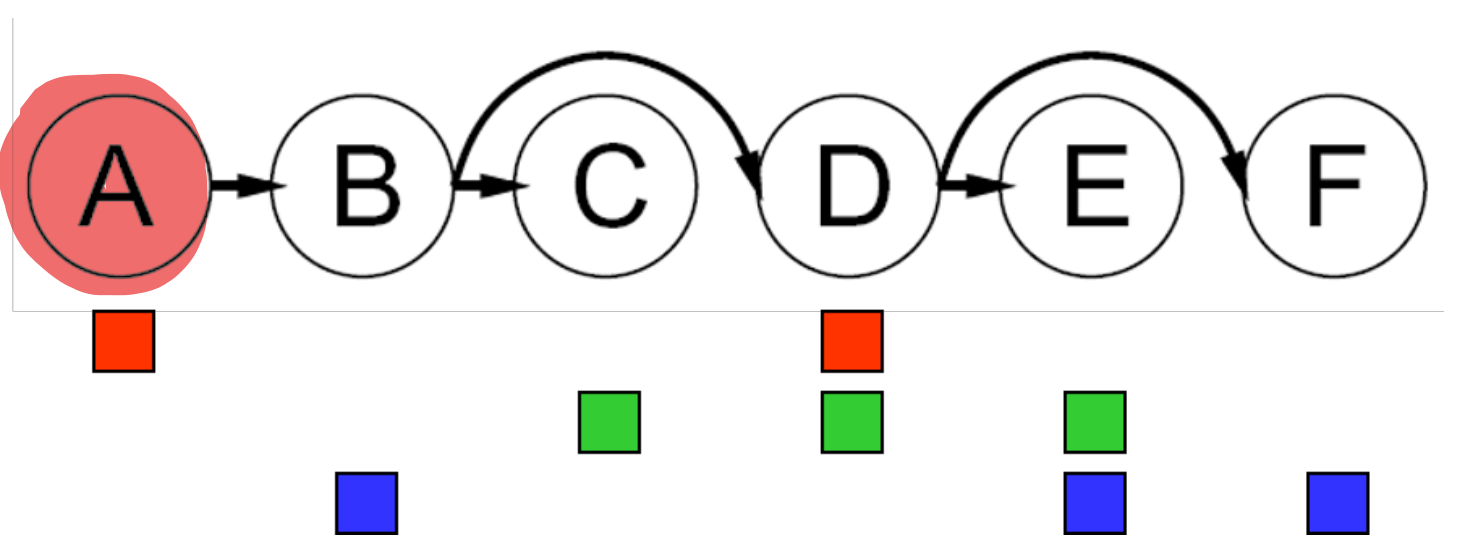


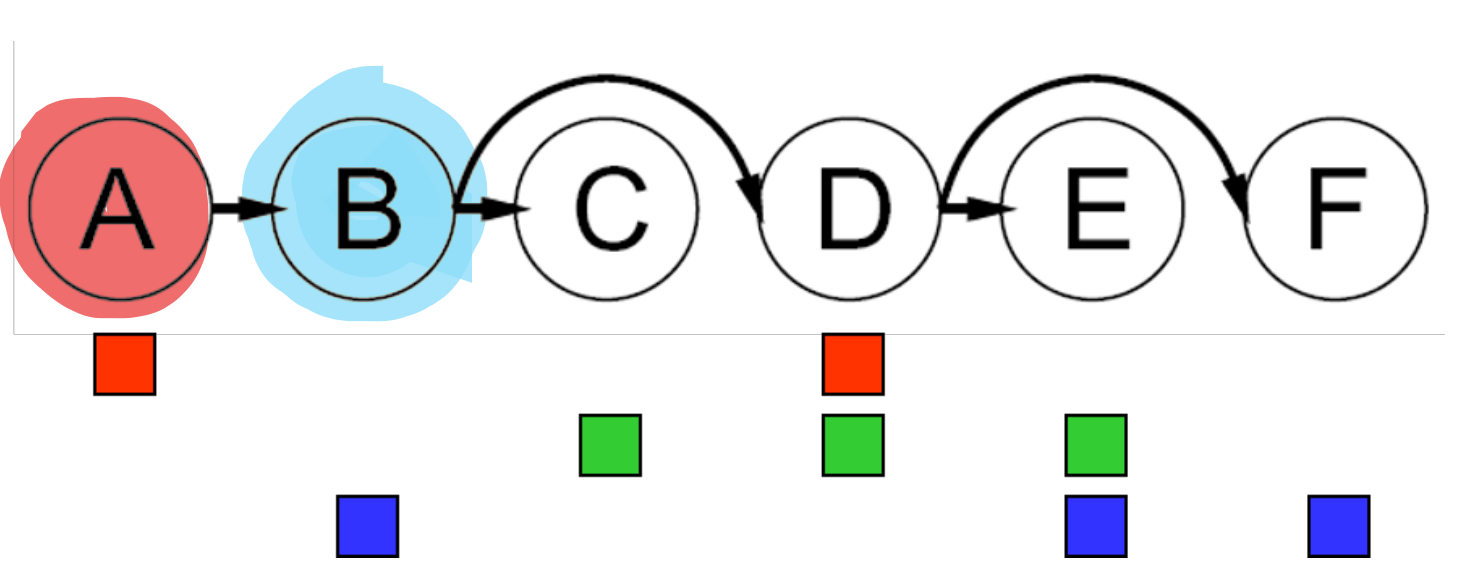


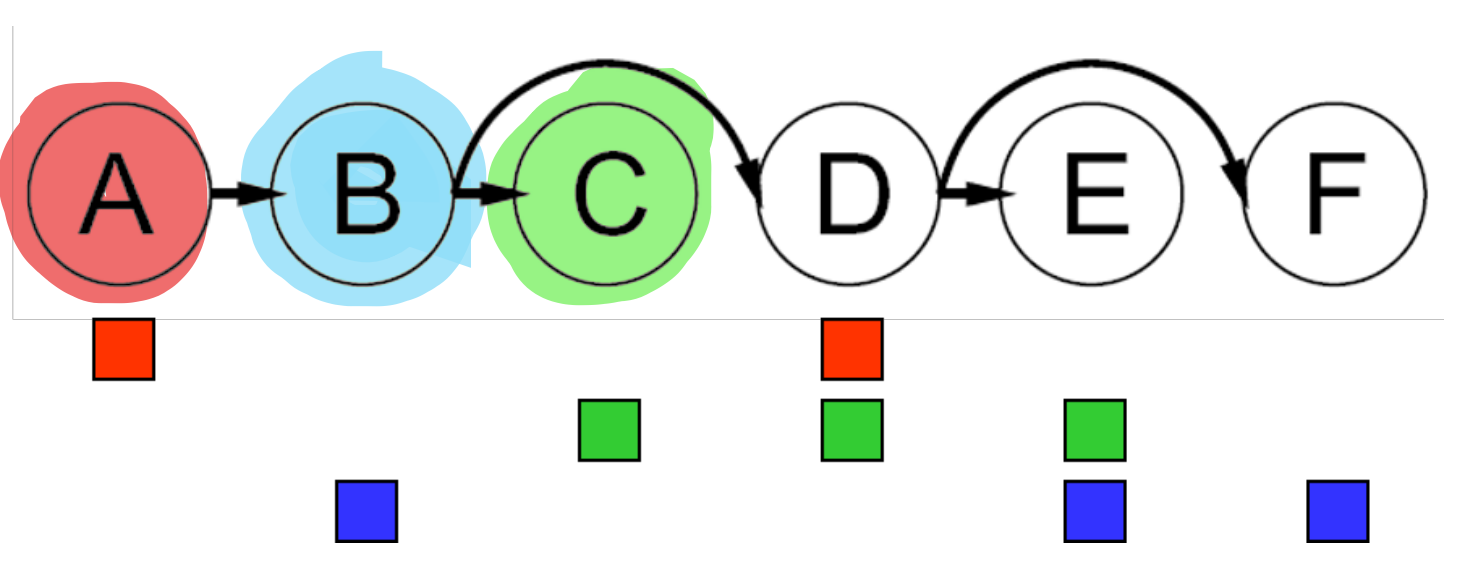


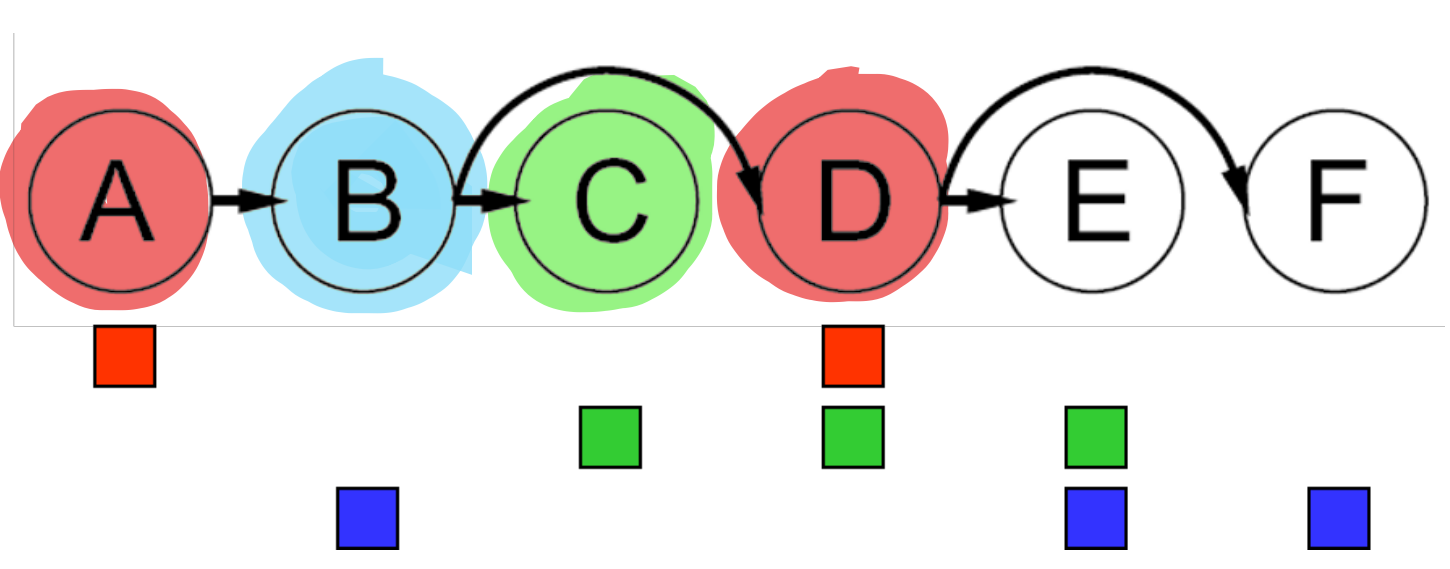


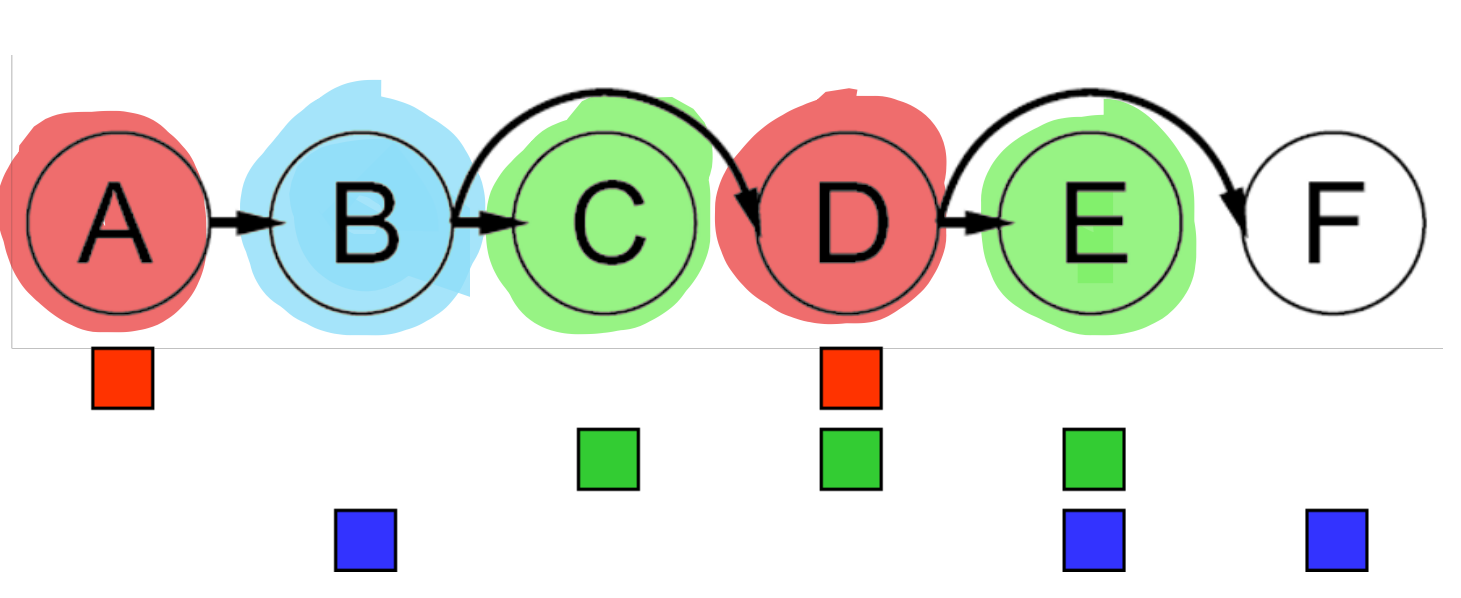
Finally, we can start assigning values, starting from the root:

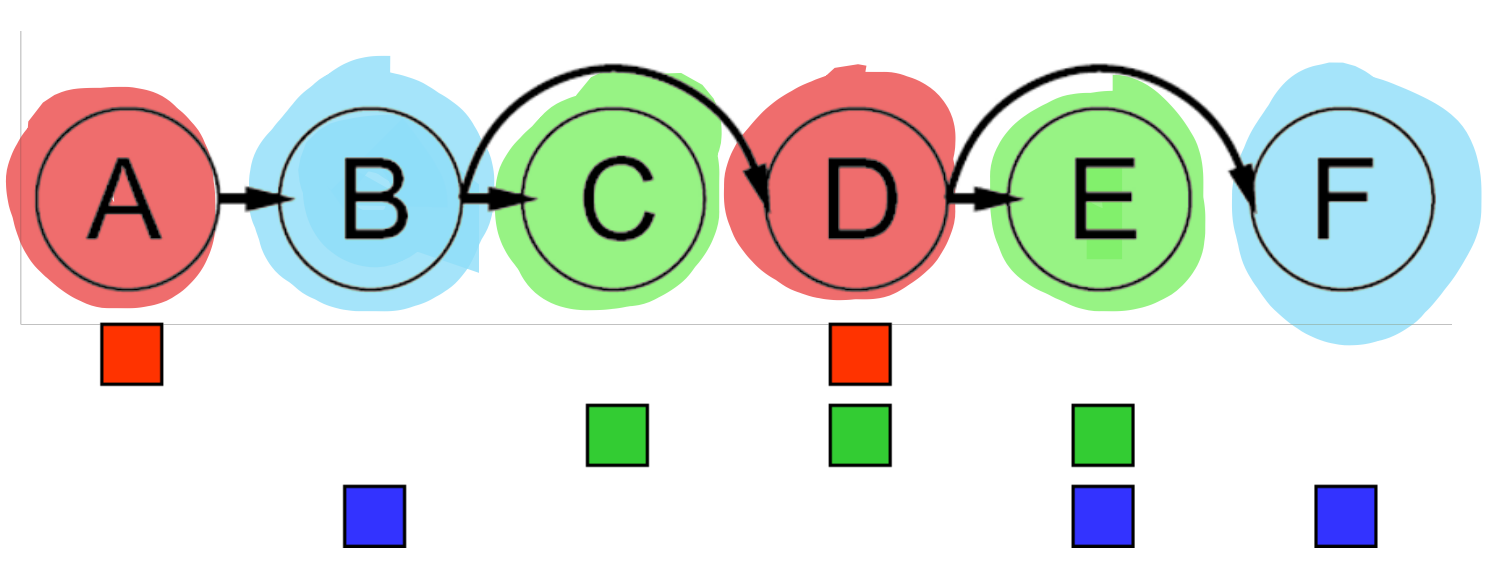










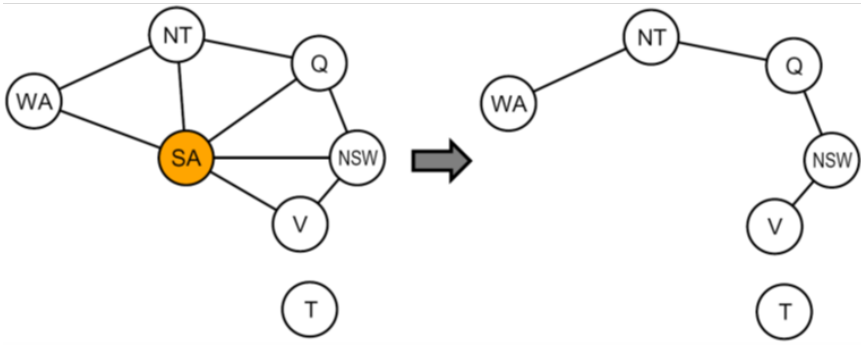


Each parent node is guaranteed to be consistent with all its children. The only time consistency can break between and is if the domain of changes. However, since we ensure consistency of with its children before we check for consistency between and , there is no chance of the domain of changing after this point. Since a parent is guaranteed to be consistent with its children, we can choose any value in the forward assignment and solve the problem **without backtracking**.

We need to go over nodes two times, each time checking potentially values. Thus, the total time complexity is .

Tree structures are unique in that they can be solved without backtracking with the help of arc consistencies. However, can also solve structures that have **loops** in them. For a maximum loop size of , we will then need to use -consistency.

### Nearly Tree Structures



Sometimes we have situations where a normal graph can be made into a tree by assigning a value to just one variable. We of course have to try all possible values for the variable. The tree we end up with is called a **cutset**.

For a cutset size of , the time complexity is . This is very fast for small values of .

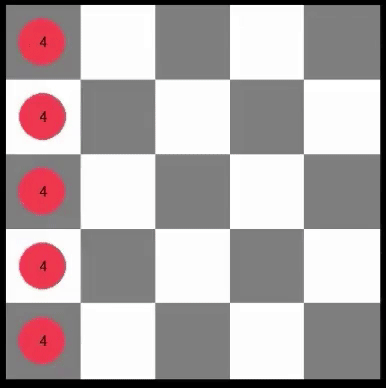
Now the question is, how do we identify that a variable can be assigned a value to create a tree structure. This is an NP problem. :)

## Iterative Improvement

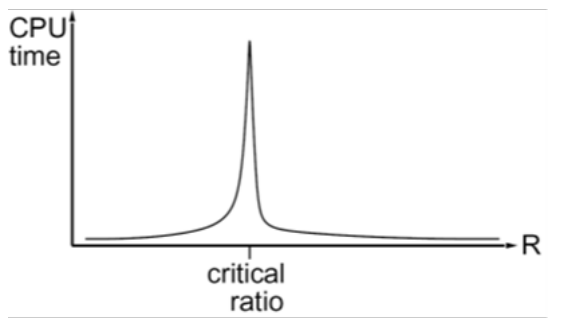
Another way we could solve CSPs is to use **iterative improvement**. Here, we start off with every variable having some value assigned, disregarding any conflicting constraints. We choose a variable that has unsatisfied constraints and **reassign** the value of the variable. Following this mechanism, we can solve the CSP without using a fringe.

To choose the new value for the chosen variable, we can use the **min-conflicts** heuristic, where we choose the value which results in the fewest overall constraint violations.

For example, consider the solution to the 4-Queens problem using iterative improvement.



The flaw in this algorithm is that there is a small possibility that we end up in a situation were we repeatedly put the queens in conflicting position, but the possibility of this continuing to happen is nearly nil. On the other hand, with a pretty high probability, we can solve this problem in **nearly constant time** for any arbitrary value of (e.g. ), so the benefits outweigh the drawbacks practically. The same applies for any randomly generated CSP, except for a very narrow range of values for .



## Local Search

In Tree Search, we used a fringe to keep track of unexplored alternatives to the current path we are exploring. This ensured that the final algorithm was complete. The idea of **local search** is to discard the fringe and do our best to improve a **single option** until it cannot be made any better. Thus, there is no fringe. This process is generally much faster and memory efficient, but it is obviously incomplete and suboptimal.



Consider the **Hill Climbing** problem, which is not technically a CSP but rather a search problem. We start at a random location and keep going up for as long as we can. Every time, we search for neighbours in a narrow radius around us to determine if there are any that are above us. If there is, we go there. If there isn’t, we declare that we have reached the top.

If we repeat the above process a few times, each time starting from a random location, and we get the same maximum answer multiple times, it is very unlikely that we managed to miss the true maximum height. Again, although this is not guaranteed to work, practically, it is very unlikely it will fail.

## Simulated Annealing

The idea of **simulated annealing** is to randomly jump in a random direction with some given probability. It has practically been found that if we start doing this with a high probability and decrease the probability of this happening over time, this mechanism works very well. The idea is that if we are in a local optimum, we will manage to jump out of the area and into one that puts us on the path to the global optimum.

This entire thing sounds like magic, but the reality is that it works. There is a **theoretical guarantee** that for a stationary distribution, will cause us to converge to the optimal state given that is small enough, but we are skipping over the exact details of this.

function SIMULATED-ANNEALING(problem, schedule) returns a solution state

inputs: problem, a problem

schedule, a mapping from time to "temperature"

local variables: current, a node

next, a node

T, a "temperature" controlling prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE[problem])

for t ← 1 to do

T ← schedule[t]

if T = 0 then return current

next ← a randomly selected successor of current

E ← VALUE[next] - VALUE[current]

if E > 0 then current ← next

else current ← next only with probability

The algorithm never actually stops, since we have no way of knowing if we are at the true global optimum. We just decided to stop it at some point. As such, practically, the guarantee does not hold. If we need a lot of downhill steps to escape a local optimum, it is far less likely that we will take them all in a row.

## Genetic Algorithms

The idea behind **genetic algorithms** is to keep the best hypotheses based on some **fitness function**. From these hypotheses, we perform a **crossover**, which involves taking parts of each and combining them.

We can visualize this practically using the N-Queens problem. We can take a few pretty good solutions, none of which are complete solutions, and combine sections of them to reach a perfect solution.

