CS 188 Introduction to Artificial Intelligence
Spring 2023 Note 8

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Last updated: January 15, 2023

Ordering

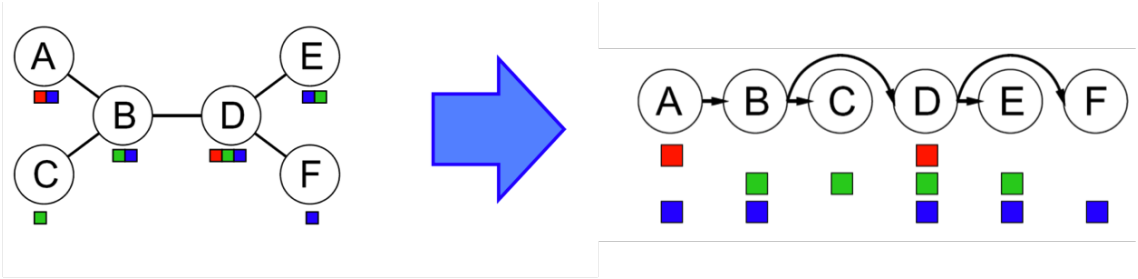
We've delineated that when solving a CSP, we fix some ordering for both the variables and values involved. In practice, it's often much more effective to compute the next variable and corresponding value "on the fly" with two broad principles, **minimum remaining values** and **least constraining value**:

- *Minimum Remaining Values (MRV)* - When selecting which variable to assign next, using an MRV policy chooses whichever unassigned variable has the fewest valid remaining values (the *most constrained variable*). This is intuitive in the sense that the most constrained variable is most likely to run out of possible values and result in backtracking if left unassigned, and so it's best to assign a value to it sooner than later.
- *Least Constraining Value (LCV)* - Similarly, when selecting which value to assign next, a good policy to implement is to select the value that prunes the fewest values from the domains of the remaining unassigned values. Notably, this requires additional computation (e.g. rerunning arc consistency/forward checking or other filtering methods for each value to find the LCV), but can still yield speed gains depending on usage.

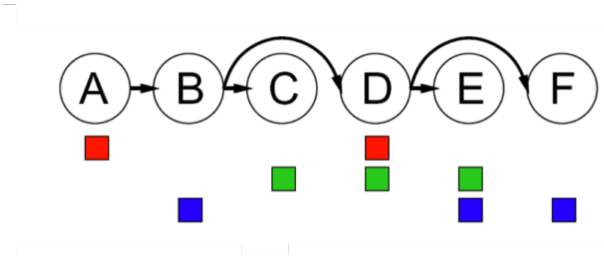
Structure

A final class of improvements to solving constraint satisfaction problems are those that exploit their structure. In particular, if we're trying to solve a **tree-structured CSP** (one that has no loops in its constraint graph), we can reduce the runtime for finding a solution from $O(d^N)$ all the way to $O(nd^2)$, linear in the number of variables. This can be done with the tree-structured CSP algorithm, outlined below:

- First, pick an arbitrary node in the constraint graph for the CSP to serve as the root of the tree (it doesn't matter which one because basic graph theory tells us any node of a tree can serve as a root).
- Convert all undirected edges in the tree to directed edges that point *away* from the root. Then **linearize** (or **topologically sort**) the resulting directed acyclic graph. In simple terms, this just means order the nodes of the graph such that all edges point rightwards. Noting that we select node A to be our root and direct all edges to point away from A, this process results in the following conversion for the CSP presented below:

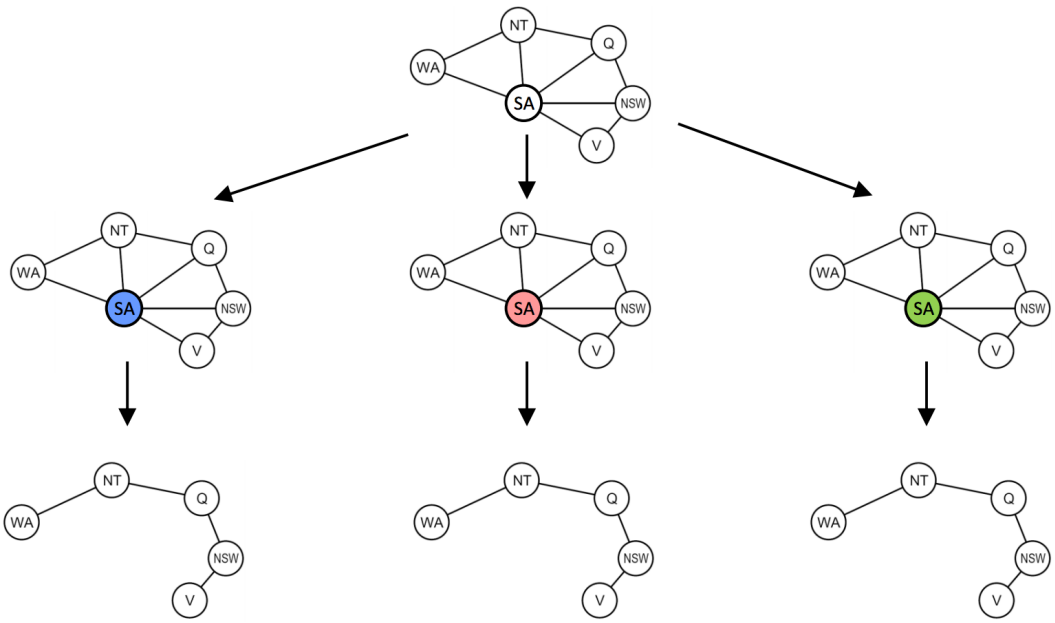


- Perform a **backwards pass** of arc consistency. Iterating from $i = n$ down to $i = 2$, enforce arc consistency for all arcs $Parent(X_i) \longrightarrow X_i$. For the linearized CSP from above, this domain pruning will eliminate a few values, leaving us with the following:



- Finally, perform a **forward assignment**. Starting from X_1 and going to X_n, assign each X_i a value consistent with that of its parent. Because we've enforced arc consistency on all of these arcs, no matter what value we select for any node, we know that its children will each all have at least one consistent value. Hence, this iterative assignment guarantees a correct solution, a fact which can be proven inductively without difficulty.

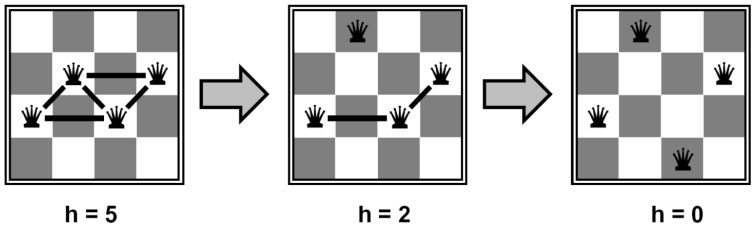
The tree structured algorithm can be extended to CSPs that are reasonably close to being tree-structured with **cutset conditioning**. Cutset conditioning involves first finding the smallest subset of variables in a constraint graph such that their removal results in a tree (such a subset is known as a **cutset** for the graph). For example, in our map coloring example, South Australia (*SA*) is the smallest possible cutset:



Once the smallest cutset is found, we assign all variables in it and prune the domains of all neighboring nodes. What's left is a tree-structured CSP, upon which we can solve with the tree-structured CSP algorithm from above! The initial assignment to a cutset of size c may leave the resulting tree-structured CSP(s) with no valid solution after pruning, so we may still need to backrack up to d^c times. Since removal of the cutset leaves us with a tree-structured CSP with $(n-c)$ variables, we know this can be solved (or determined that no solution exists) in $O((n-c)d^2)$. Hence, the runtime of cutset conditioning on a general CSP is $O(d^c(n-c)d^2)$, very good for small c.

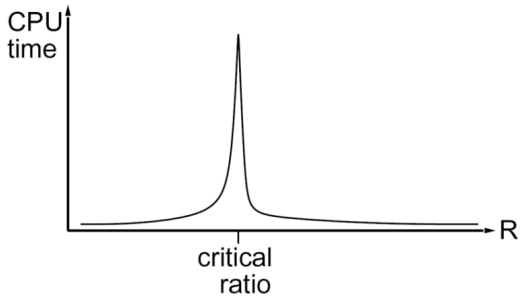
Local Search

As a final topic of interest, backtracking search is not the only algorithm that exists for solving constraint satisfaction problems. Another widely used algorithm is **local search**, for which the idea is childishly simple but remarkably useful. Local search works by iterative improvement - start with some random assignment to values then iteratively select a random conflicted variable and reassign its value to the one that violates the fewest constraints until no more constraint violations exist (a policy known as the **min-conflicts heuristic**). Under such a policy, constraint satisfaction problems like N-queens becomes both very time efficient and space efficient to solve. For example, in following example with 4 queens, we arrive at a solution after only 2 iterations:



In fact, local search appears to run in almost constant time and have a high probability of success not only for N-queens with arbitrarily large N, but also for any randomly generated CSP! However, despite these advantages, local search is both incomplete and suboptimal and so won't necessarily converge to an optimal solution. Additionally, there is a critical ratio around which using local search becomes extremely expensive:

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



The figure above shows the one dimensional plot of an objective function on the state space. For that function we wish to find the state that corresponds to the highest objective value. The basic idea of local

search algorithms is that from each state they locally move towards states that have a higher objective value until a maximum (hopefully the global) is reached.

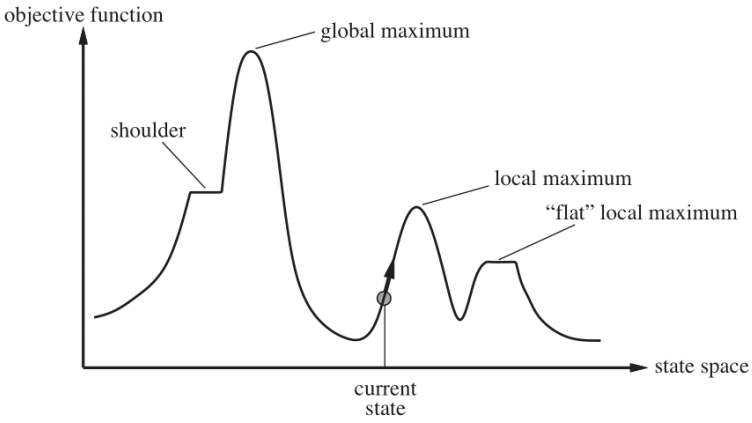


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

We will be covering three such algorithms, **hill-climbing**, **simulated annealing** and **genetic algorithms**. All these algorithms are also used in optimization tasks to either maximize or minimize an objective function.

Hill-Climbing Search

The hill-climbing search algorithm (or **steepest-ascent**) moves from the current state towards a neighboring state that increases the objective value. The algorithm does not maintain a search tree but only the states and the corresponding values of the objective. The "greediness" of hill-climbing makes it vulnerable to being trapped in **local maxima** (see figure 4.1), as locally those points appear as global maxima to the algorithm, and **plateaux** (see figure 4.1). Plateaux can be categorized into "flat" areas at which no direction leads to improvement ("flat local maxima") or flat areas from which progress can be slow ("shoulders"). Variants of hill-climbing, like **stochastic hill-climbing** which selects an action randomly among the uphill moves, have been proposed . This version of hill-climbing has been shown in practice to converge to higher maxima at the cost of more iterations.

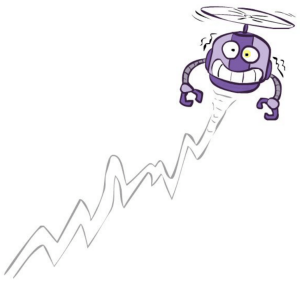
```
function HILL-CLIMBING(problem) returns a state
  current ← make-node(problem.initial-state)
  loop do
      neighbor ← a highest-valued successor of current
      if neighbor.value ≤ current.value then
          return current.state
      current ← neighbor
```

The pseudocode of hill-climbing can be seen above. As the name suggests the algorithm iteratively moves to a state with higher objective value until no such progress is possible. Hill-climbing is incomplete. **Random-Restart hill-climbing** on the other hand, that conducts a number of hill-climbing searches each time from a randomly chosen initial state, is trivially complete as at some point the randomly chosen initial state will coincide with the global maximum.

Simulated Annealing Search

The second local search algorithm we will cover is simulated annealing. Simulated annealing aims to combine random walk (randomly moves to nearby states) and hill-climbing to obtain a complete and efficient search algorithm. In simulated annealing we allow moves to states that can decrease the objective. More specifically, the algorithm at each state chooses a random move. If the move leads to higher objective it is always accepted. If on the other hand it leads to smaller objectives then the move is accepted with some probability. This probability is determined by the temperature parameter, which initially is high (more "bad" moves allowed) and gets decreased according to some schedule. If temperature is decreased slowly enough then the simulated annealing algorithm will reach the global maximum with probability approaching 1.

```
function SIMULATED-ANNEALING(problem,schedule) returns a state
current ← problem.initial-state
for t = 1 to ∞ do
    T ←schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.value – current.value
    if ΔE > 0 then current ← next
              else current ← next only with probability e^(ΔE/T)
```



Genetic Algorithms

Finally, we present **genetic algorithms** which are a variant of local beam search and are also extensively used in many optimization tasks. Genetic algorithms begin as beam search with k randomly initialized states called the **population**. States (or **individuals**) are represented as a string over a finite alphabet. To understand the topic better let's revisit the 8 Queens problem presented in class. For the 8 Queens problem we can represent each of the eight individuals with a number that ranges from $1-8$ representing the location of each Queen in the column (column (a) in Fig. 4.6). Each individual is evaluated using an evaluation function (**fitness function**) and they are ranked according to the values of that function. For the 8 Queens problem this is the number of non-attacking pairs of queens.

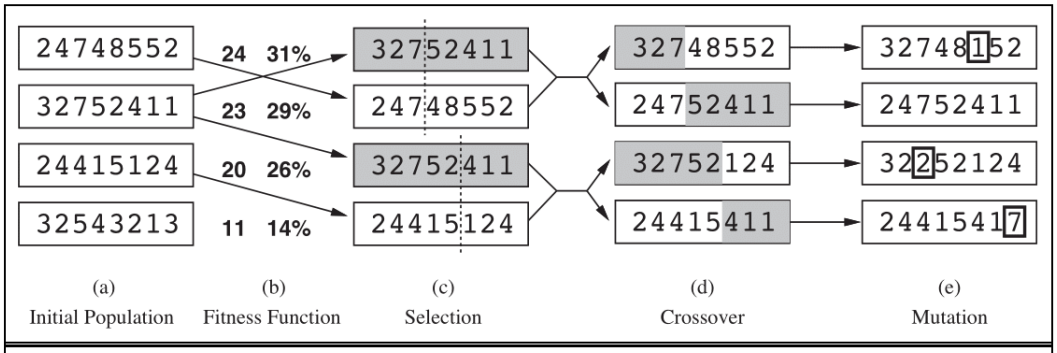


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

The probability of choosing a state to "reproduce" is propositional to the value of that state. We proceed to select pairs of states to reproduce according to these probabilities (column (c) in Fig. 4.6). Offsprings are generated by crossing over the parent strings at the crossover point. That crossover point is chosen randomly for each pair. Finally, each offspring is susceptible to some random mutation with independent probability. The pseudocode of the genetic algorithm can be seen in the following picture.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

Genetic algorithms try to move uphill while exploring the state space and exchanging information between threads. Their main advantage is the use of crossovers since this allows for large blocks of letters, that have evolved and lead to high valuations, to be combined with other such blocks and produce a solution with high total score.

Summary

It's important to remember that constraint satisfaction problems in general do not have an efficient algorithm which solves them in polynomial time with respect to the number of variables involved. However, by using various heuristics, we can often find solutions in an acceptable amount of time:

- *Filtering* - Filtering handles pruning the domains of unassigned variables ahead of time to prevent unnecessary backtracking. The two important filtering techniques we've covered are *forward checking* and *arc consistency*.
- *Ordering* - Ordering handles selection of which variable or value to assign next to make backtracking as unlikely as possible. For variable selection, we learned about a *MRV policy* and for value selection we learned about a *LCV policy*.
- *Structure* - If a CSP is tree-structured or close to tree-structured, we can run the tree-structured CSP algorithm on it to derive a solution in linear time. Similarly, if a CSP is close to tree structured, we can use *cutset conditioning* to transform the CSP into one or more independent tree-structured CSPs and solve each of these separately.