Design and Analysis of Algorithms CSCI 5454
Lecture 13 7 March 2011
Prof. Aaron Clauset

1 Optimization

The field of *optimization* is large and vastly important, with a deep history in computer science (among other places). Generally, an optimization problem is defined by having a score function $f : S \to T$ where S is some set of possible solutions to the problem and T is at least a partially ordered set of "scores," although T is more conventionally a total ordering like the reals $\mathbb{R}$ or the integers $\mathbb{Z}$. The goal of optimization is to find the globally best solution. Optimization algorithms are methods for sorting through S to find that best solution. There are several very general approaches to optimization, which differ mainly in (i) how they eliminate subsets of S from further exploration, (ii) how quickly they converge on a solution and (iii) how strong a guarantee they make on the quality of that solution.

In some cases, f has a "nice" structure, which can be exploited to produce a fast algorithm for optimization, e.g., if f is real-valued and differentiable then Newton's method is a good way to find its roots. And, recall that the graph data structure we used in the max-flow/min-cut problem allowed us to apply a greedy algorithm to find the best max-flow solution; in contrast, a direct greedy approach (pushing greater flow through the existing structure) can lead to a suboptimal solution. The max-flow data structure exploits the nice properties of the max-flow f and makes a greedy approach optimal. In other cases, even if f has "nice" structure, evaluating $f(s)$ for $s \in S$ is a very expensive operation, e.g., in running a global climate simulation. Now, in addition to wanting a good solution, we want to find it while minimizing the number of function evaluations.

1.1 Exploration vs. Exploitation

Much of the qualitative behavior of optimization algorithms is well described by considering how they trade off "exploration" versus "exploitation." Exhaustive-search algorithms, which evaluate f over the the entire domain S and then pick the globally best solution, can be seen as maximally explorative, since they make no attempt to exploit any large-scale structure in f that would allow them to find the global optimum more quickly. In contrast, "hill-climbing" or "gradient ascent" algorithms, which initialize at some point $s \in S$ and then move greedily toward the nearest local optima, are maximally exploitative, since they make no attempt to explore alternative solutions away from its current trajectory. Other techniques, including genetic algorithms, forward-backward algorithms, simulated annealing, the Nelder-Mead algorithm, etc. all strike some middle-ground between these two extremes. Generally speaking, algorithms that both explore and exploit, to some degree, can be applied to a wide variety of problems and expected to return reasonably good results.

1.2 Simulated annealing

One quite general optimization algorithm is *simulated annealing*, which we briefly considered back in Lecture 5 and which connects with Prof. Sankaranarayanan's lecture on MCMC.[1] The name comes from metallurgy—the "goal" in smithing is to produce a piece of metal (a sword, plow shears, a helmet, etc.) whose atomic configuration is very close to its minimum-energy structure; to get there, a smithy heats up the metal and then very slowly cools it, allowing the internal grain structures to align, which reduces the internal energy structure.

The simulated version of this process works like this. In addition to assuming some f for our problem space, we require a function `neighbor(s)` that takes as input some solution s and returns a "neighboring" solution[2] s', a function `accept(f(s),f(s'),t)` that takes as input the scores of two states and a parameter t (the "temperature") and makes a probabilistic decision about whether to accept the proposed move, and a function `updateTemperature()` that takes a real-valued parameter t and returns another real-valued $t' < t$. The monotonic structure of `updateTemperature()` is called the "cooling" or "annealing" schedule. Crucially, for simulated annealing to work, we must require two behaviors, for all values of t, on the `accept` function:

$$\Pr(\text{accept} \mid f(s') > f(s)) > \Pr(\text{accept} \mid f(s') < f(s))$$
$$\Pr(\text{accept} \mid f(s') < f(s), t') < \Pr(\text{accept} \mid f(s') < f(s), t) \text{ for } t' < t$$

that is, `accept` tends to accept proposed states with better scores, and the probability of accepting a "bad" move decreases with time.

One of the most common choices for the `accept` function is the one used in the Metropolis algorithm for Markov chain Monte Carlo, which satisfies the *detailed balance* criterion for MCMC and maximizes acceptance probability, where

$$\Pr(\text{accept} \mid f(s') > f(s)) = 1$$
$$\Pr(\text{accept} \mid f(s') < f(s), t) = \mathrm{e}^{-|f(s')-f(s)|/t}$$

that is, the algorithm always accepts good moves, and it accepts a bad move with probability that decreases exponentially with the difference in scores, normalized by the temperature t.

1.2.1 The algorithm

The idea then is to initialize the current solution with some given initial state s_0 and then repeatedly consider replacing the current solution s with a "neighboring" solution s', which is accepted

[1] Simulated annealing can be derived directly from the Metropolis-Hastings algorithm used by the Manhattan project for simulating the behavior of nuclear piles and was published by Metropolis in 1953.

[2] The choice of functions here has a large effect on both the time to convergence and on the probability that the algorithm converges on the global optimum.

according to the `accept` function. Finally, for the algorithm to exit, we need to introduce some way to check whether the algorithm has converged, i.e., whether it has effectively stopped exploring alternative solutions. This can be done in a number of ways, and for most applications, the choice of convergence criterion is done by heuristic.

Here's pseudocode for a generic simulating annealer:

```
Generic-SA(f,t0,s0) {
   (t,s) = (t0,s0)              // initial temperature, solution
   while not converged {
      s' = neighbor(s)          //    propose new solution
      if accept(f(s),f(s'),t) { //    accept?
         s = s'                 //       update solution
      }
      t = updateTemperature(t)  // follow annealing schedule
   }
   return state
}
```

Initially, the probability of accepting a "bad" proposal is high and the algorithm's behavior is close to an unbiased random walk on the solution space. Thus, the algorithm is highly explorative. That is, the score of the solution matters very little to the short-term behavior of the optimization algorithm. As the probability decreases, however, the algorithm becomes progressively more exploitative; eventually, when the probability of accepting a "bad" proposal is effectively zero, the algorithm behaves like a hill-climbing algorithm and only changes states if the proposed state is strictly better. The derivative of the annealing schedule controls how quickly the algorithms transitions between exploration and exploitation.

The most common choice of cooling schedule is an exponential function, such that $t' = \alpha \cdot t$ where $0 < \alpha < 1$. When $\alpha \approx 0$, the schedule is extremely aggressive and the algorithm spends very little time exploring new solutions. When $\alpha \approx 1$, the schedule is more gentle and the algorithm explores more of S.

To many theorists, simulated annealing is horribly *ad hoc* because its behavior depends on uncontrolled choices for the annealing schedule, the acceptance function and the neighbor function. Poor choices of these can lead to dismal performance and bad solutions. For instance, if we consider the set of adjacencies on S induced by the neighbor function as a graph G, if G is not connected, i.e., if for at least one pair $u, v \in S$ there is no $u \rightsquigarrow v$, then the choice of initial state s_0 effectively determines the best solution possible, which is the best solution in the component containing s_0.

1.2.2 Hill climbing, aka gradient ascent

The classic optimization algorithm known as "hill climbing" or "gradient ascent" is simply a special case of simulated annealing with the Metropolis-Hastings acceptance function: let $t = 0$. Then, the probability of accepting a bad state is always zero and the algorithm only changes solutions if the proposed solution is better than the current solution.

It is not hard to prove that such a hill climbing algorithm is guaranteed to converge on a *local optimum* of f (do you see how?). More generally, this same convergence guarantee holds for simulated annealing under most cooling schedules. Some theoretical results exist showing that under an infinitely slow cooling schedule, where, if k indexes the number of steps of the algorithm, $\partial t/\partial k \to 0^+$, the algorithm converges on the global optimum. The convergence time, however, can be (infinitely) slow. For some classes of score functions f, these results can be improved; for instance, functions with fairly "smooth" surfaces tend to converge quickly and in some cases the convergence can be proved to be quite fast.

1.2.3 Variations

Variations on the basic simulated annealing algorithm are universally motivated by the desire to reduce the likelihood that the algorithm will get stuck in local optima. For instance, in *simulated tempering*, whose name is also an analogy with metallurgy, the requirement that the cooling schedule be monotonic is relaxed and instead we only require that as the number of algorithm steps $k \to \infty$, the temperature $t \to 0$. This allows us to choose a cooling schedule that periodically raises the temperature parameter t, which shifts the algorithm's behavior back toward exploration, allowing it to escape local optima and their corresponding basins of attraction.

1.2.4 An example

As an example to demonstrate simulated annealing, suppose we want to find the global maximum of the function $f(x) = \sin(x)/x$. This is a function with a fairly rugged surface and a strong global maximum at $x = 0$.

Figure 1 shows the trajectory and results of a simple SA algorithm initialized at $x_0 = 10$. I used the Metropolis-Hastings acceptance function and the neighbor function $x' = x + N(0, 1/25)$ where $N(\mu, \sigma^2)$ is a Normal distribution with mean μ and variance σ^2, and the cooling schedule $t' = 0.99t$. The red shows the surface of the score function, the blue circles show the trajectory of the algorithm and the single green circle shows the final state (my convergence criteria was $t < 10^{-10}$). In this trial, the algorithm did a fairly good job of finding the global maximum, while on others, it would often get stuck in a local maximum. As it turns out, this is a particularly nasty choice of f. Do you see why?

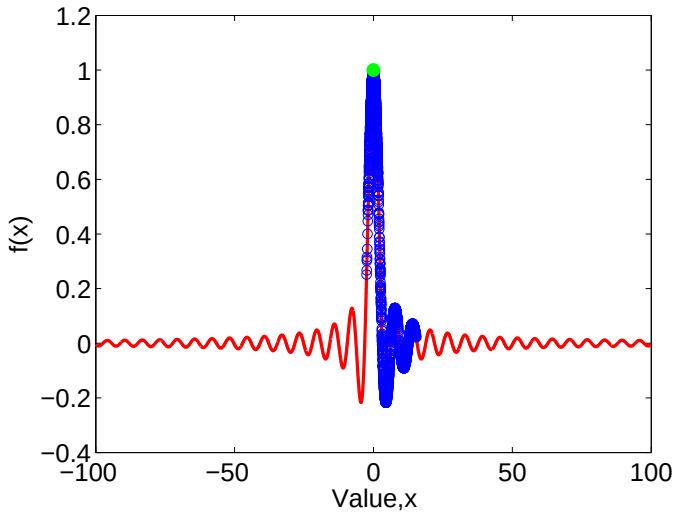

Figure 1: An example of running a simulated annealing algorithm to find the global maximum of the function $f(x) = \sin(x)/x$ with $x_0 = 10$. The green dot shows the final state.

1.3 Genetic algorithms

Genetic algorithms are another popular optimization heuristic, inspired by the ability of natural selection to quickly find fairly good solutions to complex problems.[3]

These algorithms have very similar structure to the simulated annealing algorithm described above, but with a couple of crucial differences. First, instead of the algorithm tracking a single solution, a genetic algorithm tracks a population of solutions G_t. As with simulated annealing, a genetic algorithm proceeds in steps, index here by t; at each step the score of each solution is calculated and then a new population of solutions is created. The creation of the new population is where the analogy with biology enters: in the classic version, the size of the next generation G_{t+1} is the same as G_t, and for each member of G_{t+1}, we select two "parent" solutions s, t from G_t each with probability proportional to its "relative fitness" $f(s)/\sum_{s \in G_t} f(s)$. (Many other variations exist.) The parents are first "crossed over", in which some parts of s and the remaining parts of t are combined to produce a new "offspring" u. Before u is placed in G_{t+1}, each of its parts is "mutated" with some small probability.

The mutation operator thus plays the role of the neighbor function in simulated annealing, defining

[3]Genetic algorithms were originally developed by John Holland in 1975. See Melanie Mitchell's *An Introduction to Genetic Algorithms*, MIT Press 1999.

the "local" neighborhood of solutions. In contrast, the crossover operator is very different: instead of taking a small step in S, crossover causes the new solution u to be some complicated and large jump across the space, at least with respect to the number of "mutation" operates that would be needed to yield an equivalent change in state. Thus, the mutation operator plays the role of encouraging exploitation, by generating minor variations on existing themes, while the crossover operator plays the role of encouraging exploration, by generating very large changes on the existing population.

The structure of the crossover operator has made the behavior of genetic algorithms difficult to analyze mathematically. Most analyses try to bound the overall behavior by letting the crossover rate go to zero. In this case, a genetic algorithm reduces to a kind of population-level simulated annealing at a fixed temperature. The behavior of this algorithm can be shown to be equivalent to those of the *replicator equation*, which have a close analogy with Bayesian learning, i.e., they converge exponentially on good solutions.

1.4 A panoply of methods

There are an enormous number of optimization algorithms available—too many to even attempt to list— and unfortunately this panoply is not very well organized from either a theoretical perspective or from a critical applied perspective. For instance, a good applied taxonomy might group methods by what kind of score functions f would cause them trouble, and a good theoretical taxonomy might group them by the kinds of guarantees they provide.

And yet, simulated annealing and genetic algorithms, along with techniques we haven't discussed like "ant colony optimization" and "particle swarm optimization" are often used in complex practical situations, largely because they seem to work pretty well in practice.[4] As a word of advice, you should be suspicious of all such algorithms: without a theoretical basis for their success, it's difficult to understand under what conditions we should expect them to behave badly, and, when they do behave badly, it's not clear which of the many uncontrolled choices embedded in them are to blame.

2 For Next Time

1. Crowdsource lectures begin!

[4] Plus, they sound sophisticated.