Simulated Annealing

Trying to find good solutions from bad beginnings....

A sample problem, first

- Let's say that you've got a function, and you're trying to find a "best answer" from a set of inputs
  - For instance, trying to solve the Knapsack problem with a maximum weight of 1000 units, and 500 items of varying weights to choose from
  - The "traveling salesman" problem is another example: given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city?
- How do you find the best answer?

Some possible approaches

- You can try combinations of stuff at random
  - "I'm going to try 2000 combinations, and just use the best one"
  - Doesn't guarantee a good solution at all (might be way too light, might go over on all of them, etc.)
- You can "hill climb"
  - Sort the objects based on size (O(n log n)) and then keep putting the "next heaviest thing that will fit" into the knapsack (O(n))
  - Should give you some sort of an approximation, but no guarantee of a perfect fit
  - A decent, but "greedy" method
- You can try all possible combinations
  - Guaranteed to work, but explosive complexity (O(n!))
- ....

Or, you can try something different

Enter simulated annealing (SA)

- Simulated annealing (SA) is one approach that can be used when trying to find a good approximation to a global optimum of a given function in a large search space.
- In other words:
  - If you're looking for a minimum (or maximum) value from a function given a lot of possible inputs, SA is one way of trying to get a good approximation.

Where does it come from?

- Annealing:
  - A process in metallurgy where a metal is first heated up and then put through a controlled cooling process
  - The idea is that this increases the internal crystalline structure of the metal, making it harder/stronger, and decreasing the number of defects
    - The heat causes the atoms to become "unstuck" from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy
    - The slow cooling gives them more chances of finding configurations with lower internal energy than the initial one

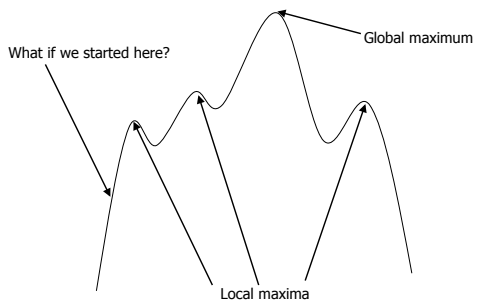
So we simulate this approach

- In simulated annealing:
  - We generate an initial configuration for the inputs to the function
  - We then pick a new configuration at random from "nearby" the current one, and see if it yields a better result
    - If it does, we'll (usually) make this our new "current configuration", and continue on
    - If it *doesn't* give us a better result, we might still make it our current configuration, but we make that choice at random
  - The "heating" and "cooling" processes are what determine whether or not we'll use a poorer solution as a current configuration
    - Early on (while the "temperature" is still high), we are more likely to allow a poorer solution to be used
    - As the algorithm continues (and the "temperature" falls), we become less and less likely to accept a poorer solution.

Some things of note

- Using SA means that we need to be able to:
  - Generate "neighbors" of a given configuration quickly/easily
    - In Knapsack, this might be done by adding/removing a single item from the collection being considered
    - In Traveling Salesman, we might pick two of the cities in the current travel plan and swap the order in which they are visited
  - Calculate the "score" for a given configuration
    - In Knapsack, this might be done by calculating how much more stuff can be put into the sack
    - In Traveling Salesman, this could simply be the length of the current travel plan
    - Traditionally, SA calls this the "energy" of the configuration, and tries to find the "lowest energy" solution

Why should we *ever* try a poorer solution?



Allowing for "downhill" moves saves the us from getting stuck at local maxima, which can happen with a "greedy" method.

The core algorithm

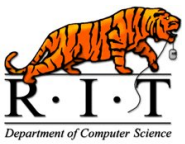
```
// Figure out where we're starting from
curConfig = generateRandomConfig();
curScore = score(curConfig)                // Remember, lower scores are better....

// It's the best that we've seen so far….
bestConfig = curConfig
bestScore = curScore

k = 0 // Energy evaluation count
while k < kmax and curScore > maxAcceptableScore
   newConfig = randomNeighbour(config)     // Pick some neighbour
   newScore = score(newConfig)                    // Compute its energy
   if (newScore < bestScore) {
      // Save the "new best" configuration's data
      bestConfig = newConfig
      bestScore = newScore
   }
   if (random() < calculateProbability(curScore, newScore, ((float)k)/kmax)) {
      // Make it the new configuration
      curConfig = newConfig
      curScore = newScore
   }
   k = k + 1 // One more evaluation done
}

// Return the best solution found
return bestConfig
```

An example of calculateProbability()

```
float calculateProbability( curScore, newScore, temperature ) {
   if ( newScore < curScore ) {
      // lower scores are better, so the new configuration is an improvement!
      return 1.0
   }
   else {
      // If we're worse off with the new config, we're less likely to accept it
      // The "colder" it gets, the less likely we will accept it
      return e^(curScore - newScore)/temperature);
   }
}
```

Some extensions to SA

- "Restarting"
  - Sometimes we might decide that we're just too far off in the weeds (e.g., we've accepted a series of poorer solutions)
  - In this case, we might move back to the best configuration seen so far, and maybe raise the temperature somewhat (to increase randomness again)
- "Tabu Search" (TA)
  - In simple SA, it's possible for the system to get caught in a cycle, where it keeps moving between a group of possible configurations
  - In Tabu Search, we keep a "tabu list" of configurations that we've already seen (e.g., the last 10), and moves to those configurations are not allowed



Any questions?