ℙROBABILISTIC 𝕋HINKING

Aryaman Singhal, Kumar Saurav, Lavesh Mangal
8-Paracetamol

April 2023

1 Why randomized optimization?

A lot of computer science problems, especially in machine learning, boil down to the *optimization* of some function f.

There are several deterministic techniques for such optimizations, such as:

1. **Explicitly finding optima**: For example, $f : \mathbb{R}^n \to \mathbb{R}$, where f is differentiable (f has continuous partial derivatives is a sufficient condition) can be optimized by setting $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right) = 0$. This criterion is called first-order-optimality.

2. **Gradient Descent:** When the gradient can be calculated explicitly but the first-order-optimality condition cannot be solved analytically, we can use the technique of gradient descent. You might recall from your JEE days or PHY103A, that the maximum decrease in the potential V is along the electric field $\mathbf{E} = -\nabla V$. Gradient descent for a function $f : \mathbb{R}^n \to \mathbb{R}$ uses tiny steps in the direction of $-\nabla f$ to approximate the minima of f.

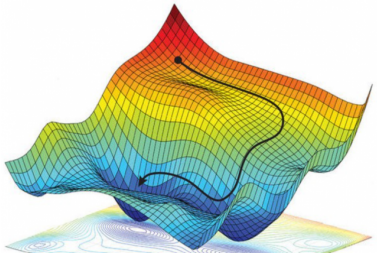

Figure 1: How gradient descent works

3. **Branch and Bound:** This is more commonly used for discrete combinatorial optimization. Examples of this technique include backtracking, integer linear programming etc. An easy example to think of is Sudoku solving. Deterministic Sudoku solvers work by constructing trees of potential solutions exploring them depth-first and quitting a branch as soon as a contradiction is reached.

Given that there are already all of these good techniques for optimization, *why* should we use a randomized search?

Well, each of the above techniques suffers from some drawbacks. The first two techniques suffer from the fact that they are best applied only to *convex functions* (there is a theorem that states that for a convex function, ***a*** minima is ***the*** minima). Gradient descent in particular is prone to getting stuck at local minima ($\nabla f = 0$ here).

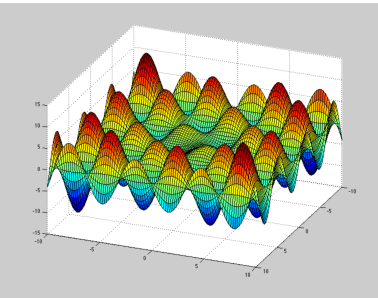

Figure 2: Non-convex function with several local optima

The last technique suffers from the problem of spurious hits. There may be branches that require nearly complete evaluation to be discarded, whence the advantage derived from bounding search is lost.

Now that we have hopefully convinced you of the incompleteness of this set of tools, we introduce you to the technique of **Simulated Annealing**.

2 Simulated Annealing - Sudoku Solver

Your mission, should you choose to accept it, is going to be to write a randomized Sudoku solver. The simulated annealing technique that you will use is described below.

Suppose that we want to find the global minimum value of a *cost function* $f : \Omega \to \mathbb{R}$ where Ω is some discrete set. The set Ω is understood to have an underlying graph structure to it i.e. for each state, there are some states that are considered to be *adjacent* to it. It will be possible to move from a given state to its neighbour.

In order to do simulated annealing we define:

1. **Neighbourhood operator:** The neighbourhood operator is a random variable $\mathcal{N} : \Omega \to \Omega$, which takes in a given state ω and outputs one of its neighbours.

2. **Acceptance Probability:** If we were doing gradient descent(in combinatorial contexts it is sometimes called hill-climbing), we would move from one state ω to another state ω' if the new state had a lower value of f i.e. that $f(\omega') < f(\omega)$. However, in simulated annealing, we allow movement to a *worse* state with some probability. This prevents us from getting stuck in local minima. The acceptance probability function that you will be using is the following:

$$\mathbb{P}(\omega \to \omega' \text{ accepted}) = \exp\left(-\frac{\delta}{T}\right)$$

where $\delta = f(\omega') - f(\omega)$.

3. **Cooling Schedule:** In order to converge to a global minima, we slowly decrease the temperature T.

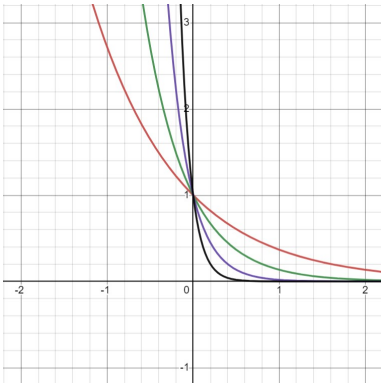

Figure 3: Acceptance probabilities for $T = 1, 0.5, 0.25, 0.125$ (red, green, purple, and black respectively)

From the graphs, you may observe that this leads to a decrease in the probability that a solution with high positive δ is accepted. Thus, this discourages the annealing algorithm from jumping away from the optima as time progresses. The cooling schedule is defined by

$$T_{i+1} = \alpha T_i, \qquad 0 < \alpha < 1$$

4. **Reheating:** If no change has been observed for some number of iterations, say 20, we scrap this run, choose a new initial solution and reset the temperature. This process is called *reheating*.

We describe the specific application to Sudoku solving below. For a given Sudoku puzzle we define the following state space:

$$\Omega = \{\text{all possible filled up boards satisfying each } 3 \times 3 \text{ block }\}$$

Figure 4: Sample constraints

We choose an initial solution as follows. For each 3×3 block, we fill the blank squares with some random permutation of the available numbers.

Let $R_i(\omega)$ denote the set of all numbers in the i-th row and let $C_i(\omega)$ be defined correspondingly for columns. We define the cost function f as:

$$f(\omega) = \sum_{i=1}^{9}\sum_{j=1}^{9} \mathbf{1}_{i \notin R_j(\omega)} + \sum_{i=1}^{9}\sum_{j=1}^{9} \mathbf{1}_{i \notin C_j(\omega)}$$

That is, for each row and column we count how many numbers from 1-9 are not present in that row or column. The sum of all these values is the cost of ω.

The neighbours of a solution are defined by all the filled boards that can be obtained by permuting two numbers within the same 3×3 block.

The initial temperature may be initialized to the standard deviation of the cost δ over all the possible neighbours of the initial solution(you can compute this by computing some number of random moves and calculating the sample standard deviation).

α may be taken to be 0.99.

You should run the algorithm for as many iterations as is computationally feasible for you(typically it shouldn't take more than $\approx 1 - 2s$).

Note: The mentors recommend C++ for this particular assignment. Further, please make sure to use a good PRNG for random number generation. **mt19937** generator is a superior choice to the C style **rand()**.