1 Introduction

Simulated annealing provides an approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete (e.g., the traveling salesman problem).

The name and inspiration of the algorithm comes from the process of annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. Similarly, simulated annealing functions by taking some initial greedy solution and slowly transforming the solution into an optimal one by generating neighboring solutions and moving to it based on a temperature-dependent probability that decreases over time (hence the notion of slow cooling). In general, simulated annealing works as follows:

"The temperature progressively decreases from an initial positive value to zero. At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and moves to it according to the temperature-dependent probabilities of selecting better or worse solutions, which during the search respectively remain at 1 (or positive) and decrease towards zero."

2 The Basic Iteration

The goal in simulated annealing is to bring the system from an arbitrary initial state to a state with the minimum possible energy; in our case, energy represents the cost of traversing a path. At every iteration of the algorithm, we do the following:

1. *Given the current state s, generate some neighboring state s'.* New states are produced through conservatively altering a given state. In the traveling salesman problem, each state is defined as a permutation of the notes to be visited.

2. *Probabilistically decide between moving the system to state s' or staying in state s.* The probability of making the transition from current state s to a candidate new state s' is specified by an *acceptance probability function* $P(c, c', T)$, that depends on the costs $c = cost(s)$ and $c' = cost(s')$ of the two states and on a global time-varying parameter T called the *temperature*

3. *Update the temperature T.* Temperature should decrease as a function of the time elapsed so that the annealing schedule can terminate at $T = 0$.

The algorithm starts initially with T set to a high value (or infinity), and then it is decreased at each step following some annealing schedule—which may be specified by the user, but must end with $T = 0$ towards the end of the allotted time budget

3 Pseudocode

The algorithm works as follows:

```
SimulatedAnnealing($G, t$):
    Input: a complete graph $G$
    Input: time limit $t$
    Output: a tour and its cost
    bestState, bestCost = null, $\infty$
    $s, c$ = GetStartingState($G$)
    $T = t$
    while $T > 0$:
        $s', c'$ = GetNeighboringState($s$)
        if GetAcceptanceProbability($c, c', T$) $\geq$ random(0, 1):
            $s = s', c = c'$
            if $c < bestCost$:
                bestState, bestCost = $s, c$
        $T$ = GetTemperature(time elapsed, $t$)
    return bestState, bestCost
```

The **GetStartingState** function takes in a complete graph G and runs a greedy algorithm by choosing the closest neighbor at each step in the function:

```
GetStartingState($G = (V, E)$):
    currNode = $V[0]$
    path = [currNode]
    remainingNodes = $V$ - currNode
    while remainingNodes > 0:
        minDistance = $\infty$
        nextNode = null
        for node in remainingNodes:
            if distance(currNode, node) <minDistance:
                minDistance = distance(currNode, node), nextNode = node
        Append nextNode to path
        remainingNodes = remainingNodes - nextNode
    return path
```

The **GetNeighboringState** function takes in the current state s, which is just a path of nodes, and generates a neighboring state by running the 2-opt algorithm often used in traveling salesman problems. Traditionally, in simulated annealing, the neighbors of a state are the set of permutations produced by reversing the order of any two successive cities. The intuition is that after a few iterations of the simulated annealing algorithm, the current state is expected to have much lower energy than a random state. Therefore, you should modify the current state such that the total cost of s' is likely to be similar to that of the current state. However, we modified the algorithm to perform the 2-opt algorithm instead in order to reduce the chances that the algorithm gets stuck in a local minima state and is unable to improve, as it is constantly generating near identical states that the acceptance probability function rejects.

```
GetNeighboringState($s$):
    Generate random indices $i, k$ such that $0 \leq i < k \leq$ number of nodes in $s$
    neighbor = []
    Take $s[0]$ to $s[i-1]$ and add them in order to neighbor
    Take $s[i]$ to $s[k]$ and add them in reverse order to neighbor
    Take $s[k+1]$ to the end and add them in order to neighbor
    return neighbor
```

The **GetAcceptanceProbability** function takes in the costs of the current and candidate state as well as the current temperature and returns the probability between 0 and 1 (inclusive) that the candidate state should be accepted by the simulated annealing algorithm.
The probability $P(c, c', T)$ is equal to 1 when $c' < c$, so the algorithm always accepts a path with a total lower cost. However, when $c' > c$, the P function is usually chosen so that the probability of moving to the candidate state decreases as the difference between the two costs increases. The temperature T plays a role as well: for a large T, P favors larger energy variations, while when T is small, P favors finer energy variations. We take the simple P function first used by Kirkpatrick, Gelatt Jr., and Vecchi in 1983 when simulated annealing was first proposed as a solution to this problem:

```
GetAcceptanceProbability($c, c', T$):
    diffCost = $c' - c$
    if diffCost < 0:
        return 1
    else:
        return exp($\frac{-diffCost}{T}$)
```

The **GetTemperature** function takes in the time elapsed in the algorithm and the total time allocated for the algorithm to run. T must decrease as the time elapsed increases, so we use the following algorithm:

```
GetTemperature(timeElapsed, $t$):
    return 1 - (timeElapsed)/$t$
```

4 Sources

1. https://en.wikipedia.org/wiki/Simulated_annealing
2. https://en.wikipedia.org/wiki/2-opt