Simulated Annealing

A. Marsh, E. Goeke

December 6, 2018

1 Background and Motivation

Simulated annealing is a general probabilistic optimization method that is useful when finding the global optimum of a function that contains many local optima. Functions best suited to simulated annealing are factorially large, discrete, and have local optima that do not describe or approximate the solution.

Functions which are factorially large cannot be completely explored when searching for global optima due to the computational cost of exploring the entire solution space.

Simulated annealing can operate within a discrete variable space. The challenge posed by a discrete variable space is the difficulty in ascertaining which move will travel in the most "downhill" [1] direction, because each possible "step" is a reconfiguration of the system which is defined by a set of discrete variables. Simulated annealing solves this problem by randomly exploring different permutations of a configuration space by making prescribed "moves" in a Monte Carlo fashion, accepting better configurations, as well as worse configurations with some probability.

Because simulated annealing targets a global optimum, it is desirable for difficult optimization problems such as: reducing interference in complex integrated circuits, single-machine, flow-shop and job-shop scheduling problems [2] and the traveling salesman problem [3], which are all examples of problems modeled by a combinatorial set of variables.

The term annealing comes from a thermodynamics analogy in which a liquid is cooled such that the thermal mobility of molecules is slowly lost and an ordered crystal is formed at an energetic minimum. The key word in this analogy is *slow*. As the liquid cools, the crystal that forms becomes less likely to drastically change its structure, but can still minutely minimize its energy state relative to its larger structural trends.

To search for the optimum, simulated annealing takes a random step and the probability that the step is accepted is drawn from a distribution similar to the Boltzmann distribution (1).

$$P(E) \ \exp\left(-E/kT\right) \tag{1}$$

Where T is the temperature of the system, E is the state energy, and k is the Boltzmann constant. In thermodynamics, this distribution illustrates that there is a non-zero probability that a system at a low temperature could be at a higher energy state, although this probability shrinks as temperature decreases. This phenomenon is important in the context of simulated annealing because there is a probability that the algorithm won't only choose steps which minimize the function. Instead, it will occasionally take steps ”in the wrong direction”, at a frequency which decreases as the function's value decreases, which reduces the likelihood of the algorithm getting stuck in local minima.

In this way simulated annealing differentiates itself from optimization algorithms like gradient descent, which try to find the steepest path to an optimum by using the first derivative of a cost function. Where simulated annealing can be thought of as ”slow” cooling, it is useful to think of gradient descent as ”fast” cooling.

2 Theory

In general the ingredients required for applying an annealing simulation to a system are as follows [1]:

1. A description of possible configurations.
2. A method of randomly perturbing a configuration.
3. A cost function to be minimized, which is analogous to the energy of a configuration.
4. A control parameter, T, and a schedule for updating it.

The control parameter, T is analogous to temperature, and proportionally represents the likelihood of the simulation to accept a perturbation of the system which results in a higher energy configuration.

The cooling schedule is the method for updating T, causing it to decrease as the number of attempted configurations increases. The effect of the cooling schedule is to impose increasingly stringent requirements on the acceptance of a configuration as the number of attempted configurations increases. This is accomplished by decreasing the probability of accepting a change in the configuration that would increase the energy of the system.

2.1 Combinatorial Minimization

Combinatorial minimization is the topic of determining an optimal result from a finite set of possible results, whose discrete configuration space is too large to be completely searched. [4]

Travelling Salesman is an example of a combinatorial minimization problem that is a good case for simulated annealing.

The Travelling Salesman problem [3] describes how to best plan a route visiting many points so as to optimize for some variable, most usually the minimization of distance travelled, but could be other variables, such as the minimization of border crossings.

Setting up the Travelling Salesman problem to be solved by simulated annealing would look like:

1. The description of possible configurations. For a plain Travelling Salesman problem every marked location must be visited at least once. An element of the solution set would be an indexed list of ”moves” from one location to the next. This element represents one possible configuration.

2. A method of randomly perturbing a configuration. This could be as simple as randomly shuffling the orders of the stops, or a more efficient change, like those proposed by Lin [5]:

    - Reversing the order of a path between two locations.
    - Moving the traversal between two locations to a new random position in the list.

3. A cost function. This could be as simple as the sum of the distance of the paths travelled[1]:

$$E = \sum_{i=1}^{N} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{2}$$

but could also include values that represent the different weights incentivizing or disincentivizing certain behaviors (e.g border crossings):

$$E = \sum_{i=1}^{N} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2 \tag{3}$$

Where μ is a point value assigned to a certain characteristic of the path (the requirement of crossing a border). By assigning a positive μ value to a characteristic, the simulation will attempt to avoid paths that contain that characteristic. The parameter λ is a weight signifying the relative importance of minimizing distance versus minimizing the other characteristic.

4. A control parameter and update schedule. A common control parameter looks like:

$$\text{accept if} : U(0,1) \leq e^{\frac{-\Delta E}{kT}} \tag{4}$$

With T held constant for some integer number of attempted configurations. T is decreased when either a small number of successful attempts, or a larger number of total attempts have occurred. The latter case is to serve as a cut off for computational cost once the result becomes discouragingly hard to improve for a given T.

3 Problem

The problem I solved was one of a traveler (”Homer”) attempting to cross a 2D grid like city diagonally in as little time as possible, starting at position (0,0) and ending at position (N,N), where N is the dimension of the square grid.

The city that Homer travels is represented as a graph with nodes at integer positions (x,x) that act as both support points computationally, and decision points for the traveler, i.e a traffic light. Edges between the nodes represent the simulated paths between decision points.

Path finding along a grid in this manner is a problem with a factorially large solution space- with the 20x20 grid used as a test case presenting $\tilde{1}$37 billion possible paths. The size of this combinatorial solution space makes it ideal for simulated annealing methods to quickly find optimal or close to optimal solutions.

As more elements are added to the problem it becomes more interesting. The elements I introduce into the problem are:

1. Static weighted time costs of traversing an edge from one node to another.

2. A global time of simulation, which is progressed as the traveler makes decisions, and on which aspects of the simulation can be dependent.

3. The property of whether an edge is traversable, which takes an arbitrary function specific to the edge which can depend on the current time of the simulation.

3.1 The Cost Function

The cost function that the simulation will attempt to minimize is the time taken to reach the target node of (N,N). For this simulation each edge traversal has a static weighted time cost. When the traveler makes a 'move' and traverses an edge to reach another node, the current time of the simulation is updated by the time cost of the move, and therefore all aspects of the simulation dependent on time are also updated. This time cost is an abstracted concept that includes the calculation of many predictable and random variables, for example the speed limit along a road, and perhaps some 'traffic congestion coefficient'.

This weight could easily be made into a function of time in the same way the traversability function is. In future simulations including many travelers, it could also become a function of "position of travelers", in order to simulate actual traffic through applied game theory.

3.2 Time Dependence

The time dependent traversability function of edges allows for the easy simulation of traffic cycles, and my simulation uses two such phases. One phase applies to edges parallel to the x axis, and the other applies to the y axis. They are completely out of phase with each other, such that for a completely evenly weighted time cost of edges, an optimal solution could easily be found by the traveler always making moves parallel to the direction of allowed traffic, assuming the speed of the traveler is in phase with traffic.

The introduction of time dependence greatly impacts the size of the solution space, as well as the cost function of solutions. An immediate consequence of time dependence is the solution space grows from factorially large to infinitely large. The characteristic of the simulation for a traveler to have potentially zero moves from their current node-position introduces the necessity of "waiting". In order for the simulation to continue, a traveler must be able to not make a move to another node while still progressing the time of the simulation, as progressing the time can potentially make edges traversable and therefore moves available. Once it is possible for a traveler to wait as a 'move', the traveler may always choose to do so. Moves that advance the time, and therefore possible paths available, and may be repeated infinitely will increase the number of possible paths for any possibly traversable grid to infinity.

Even without the possibility of paths filled with infinite waits, the ability of a traveler to deliberately wait allows them to feasibly explore a much greater number of paths. For example, if a traveler finds themselves at a junction that can be immediately traversed without waiting, by intentionally waiting for the phase of traversable edges to change, an entirely independent sub-path becomes available. This sub-path that is only accessible through waiting can of course have a wildly different time cost from the previous path. This characteristic of waiting will be important for the simulated annealing approach used to find paths of minimum time cost.

4 Solution

4.1 Overview

The solution software will need these elements to apply simulated annealing:

1. A decision making algorithm for the traveler to use that allows them to choose which node to move to when given a list of edges to traverse.
2. A method for introducing deviations between paths.
3. A loss function with which to compare paths.
4. A method for applying simulated annealing to decide which path to iterate on.

The overall algorithm looks like:

```
path1 = []
For I iterations:

    //Generate an initial path:
    While(traveler.current_node !=
traveler.target_node):
        move =
traveler.decide_move(traveler.current_node.get_edges())
        path1.append(move)

    //Generate a second path that deviates from
the first:
    path2 = path1.generate_deviation()

    //Compare the two paths:
    path1 =
simulated_annealing_decision(path1,path2,I)

return path1
```

4.2 Node to Node Decision Making Algorithm

The simulation requires a baseline decision making algorithm that the traveler can use at any node-position in order to decide what their next move should be. At a given node-position, the traveler can see every edge connected to that node, and therefore every possible path available to them. While all connected edges are accessible, their time dependent traversability functions may render them impassable. From this check the traveler will have a filtered list of possible moves to choose between. Each possible move contains information on the edge's time cost, and where the edge leads, and these attributes can inform the traveler's decision on where to move.

Possible algorithms for moves:

- Random Choice. For N available moves, each move has a 1/N chance of being executed.
- Greedy. For N available moves, calculate the velocity of a move towards the target node-position, and choose the move with the greatest positive velocity.

  $$V_m = \frac{\sqrt{(\Delta x_1 - \Delta x_2)^2 + (\Delta y_1 - \Delta y_2)^2}}{\text{time cost}} \tag{5}$$

  Where

  $$\Delta x = x_{target} - x_{current} \tag{6}$$

- Pensive Greedy. Same as greedy, but includes a simulated wait at each node-position to reveal paths that could be available should the traveler choose to wait.
- Combination of the above.

An individual random choice decision is very cheap to calculate, but the simulation will take many more moves to find a possible path. Random decisions do have the benefit of being able to eventually find a possible path in a graph with few possible paths (i.e greed 'traps' that cause deterministic algorithms to fail, many obstacles, out of phase traffic lights etc...).

An individual pensive move decision is often twice as expensive as a regular greedy decision, but will find paths with shorter time costs in roughly the same number of moves as the greedy algorithm.

Because the test case used is a grid without permanent obstacles, and which is always traversable with greedy algorithms, I rule out using random choice algorithms.

I rule out using pensive decision algorithms because the objective of the problem is to use simulated annealing to find shortest paths, and not to design the most effective heuristic for initial path generation. Additionally, the main importance of a deterministic algorithm for decision making is that it reduces the number of possible paths from infinite back to factorially large. Both pensive and regular greedy algorithms accomplish this, but regular greedy is cheaper to calculate per move. Therefore I choose to use a regular greedy algorithm as my node to node decision making algorithm.

Implementing waiting is trivial with this simulation design, as all that needs to be added is a "loop", an edge that connects a node to itself, to every node with a time cost of 1 (in discretized simulation time units). This presents the prospect of waiting as a move with a velocity of 0, which will be chosen over moves with negative velocity, but will always be deferred for moves that get the traveler closer to their target node.

4.3 Generating Deviations Between Paths

Deviations between paths are generated by following an identical path up until move index d, where d is a random integer [1,z-1), where z is the number of moves used by the previous path. At move index d in the path the decision making algorithm will disallow the choice previously made at that index. This will cause the decision making algorithm to take its second best choice for that move. After that, the traveler will use its default move decision algorithm until it finishes the path.

Because the traversable paths available to the traveler are dependent on time, a single deviation will cause any further step to be completely independent from the same indexed step in another path. With this deviation strategy partially dependent paths may be generated and compared.

A note on the software implementation of these deviations- because the two paths will be identical up to some point, and my decision making algorithm is completely deterministic, it does not make since to recalculate all the moves to recreate the dependent section of the deviated path. Instead, the "state"- an end time, end node, and history of node-positions- is saved to the new path, and the traveler on the deviated path is "teleported" in time and space to continue where they must diverge.

4.4 Loss Function

Because the objective of this problem is to calculate the "shortest" path between two points, I choose to define shortest with regards to the time cost of the path. This choice allows me to use both node positions, and weighted edge time costs as variables that paths are dependent upon. This dependence stems from a likelihood, but not a certainty, that paths that make more positional moves take more time. The interdependence of paths on time and space makes for a problem suited to simulated annealing, which is able to quickly prune factorially large solution spaces.

4.5 Applying Simulated Annealing

With the ability to generate different paths, and the possession of a single metric with which to compare the efficiency of paths, simulated annealing can be used to "slowly cool" the exploration of solution space, always accepting more efficient paths, and with some probability accepting less efficient ones.

The decision making for which path to continue iterating (i.e generating deviations for) is:

$$\text{if } e^{\frac{\text{time cost}_1 - \text{time cost}_2}{T}} : \text{path}_1 = \text{path}_2 \tag{7}$$

5 Results

For the test case of a 20x20 node grid with edges of time cost randomInt(1,10) and two stoplight phases of opposite direction, convergence of to the shortest path was found in ¡1500 iterations, with each iteration being the generation of a deviation upon the previously kept path, whose time costs are then compared and the new path is chosen via simulated annealing. The run time for 100000 iterations was 20-30 seconds. Every time the code has been ran, the simulated annealing algorithm has found a path shorter than the initial one in terms of time cost.

Each time the code (file named "run_sim.py") is ran two .gif files are generated. The first is a procession of images of every path attempted, the second is only images of paths that were kept from one iteration to the next.

6 Conclusion

The aim of this project was to find optimal time dependent paths in complex environments. This has been accomplished for the test case, and moreover, the flexibility of the simulation program that was designed allows for the rapid testing of an arbitrarily shaped graph with edges whose weights are functions of variables that change within the simulation. The manner in which this problem has been solved has also created a machinery than can solve a much broader range of path-finding problems.

Further simulations can be done for:

- Graphs with "greed-traps", obstacles positioned such that generic greedy algorithms become stuck indefinitely.
- Graphs of unusual shape (GOUS),
- Graphs with moving or static obstacles (i.e nodes whose edges are never traversable)

With slight alterations to the code, multiple travelers can be supported on the same graph, which opens the possibility of complex traffic simulations as travelers interact with each other. All that is required is traversability and time cost functions to become a function of traveler positions or densities.

References

[1] William H. Press. *Numerical recipes in Fortran 90: the art of parallel scientific computing: volume 2 of Fortran numerical recipes.* Cambridge Univ. Press, 1996.

[2] C Koulamas, SR Antony, and R Jaen. A survey of simulated annealing applications to operations research problems. *Omega*, 22(1):41 – 56, 1994.

[3] Eric Weisstein. Traveling salesman problem.

[4] A Schrijver. *A Course in Combinatorial Optimization.* 2006.

[5] S Lin. *Bell System Technical Journal*, 44:2245 – 2269, 1965.