Simulated Annealing for Traveling Salesman Problem

David Bookstaber

à This report explains the details of an implementation of a Simulated Annealing (SA) algorithm to find optimal solutions to the Traveling Salesman Problem (TSP). For comparison, a Genetic Algorithm (GA) was applied to the same problem.

Ÿ Why the TSP?

The TSP is a member of a class of problems known as Nondeterministic Polynomial-Time Complete (NPC). A large number of important and practical problems are proven members of this class and are therefore analogous to the TSP, having solutions that are transformable to and from solutions to the TSP in polynomial-time. Methods applicable to solution of one member of this class are therefore of interest to the whole class. It is assumed that NPC problems can only be solved in exponential-time, so that typical orders of magnitude are often intractable to solution on current computers. In practice, suboptimal solutions that are still "really good" are often sufficient, and in these cases monte carlo methods like SA and GA's have proven successful in generating useful solutions despite the size of the problems.

The TSP consists of a set of "cities," or points, with the object of finding the shortest path connecting them under a given metric. The difficulty of this problem is obvious when one realizes that an n-city set has a search space of (n-1)!. As with many such problems, there are specialized heuristic algorithms that have been applied to the TSP with excellent results. However, no attempt will be made here to use our specific knowledge of the problem to improve the algorithms—the present interest is only in using the TSP as a characteristic "hard" problem in hopes that the results may be similar for other problems that are not as well understood or studied, but that nonetheless need solution.

Ÿ The SA implementation requires:

(1) State space with fitness measure
(2) Graph on the state space
(3) Temperature schedule

State space: For each case of the TSP we are given a set of cities. So the search space is most naturally divided into states consisting of unique orderings of those cities. The "fitness" (or "cost") of each state is the length of a path through the cities in that order. The goal is to find a state with the smallest possible fitness.

Graph: The topology of the graph is critical to the ability of the algorithm to find optimal solutions. Intuitively it can be said that we want neither a one-dimensional graph nor a totally

connected graph. The former restricts the algorithm too much, while the latter pretty much amounts to random sampling. This intuition was verified empirically on the following construction:

The most straightforward graph is arrived at by considering interchanges of atomic units (at least one city in length) in the sequence of each state. That is, an atom is selected at random from the sequence of a state and interchanged with another atom selected at random from those within a given range to arrive at a new state. For simplicity, in this implementation the atoms were chosen to be one city in length, so that each iteration one of the n cities was chosen at random and switched with another city chosen at random from within a fixed range in the sequence. (The resulting graph was thus lower in dimension than other published implementations that allow for atoms of varying length. It seems likely that allowing for longer atoms would improve the ability of the SA to break out of local optima, but this is not considered further here.)

Temperature: The key to SA's ability to find global optimums, and what differentiates it from simple hill-climbing algorithms, is that at each iteration it has a (decreasing) probability of moving to less fit states. It always moves to more fit states but, presented with the prospect of moving to a less fit state, it will do so with probability equal to Exp[-(dFitness)/T], where T is temperature. Theoretical analysis of SA indicates an optimal (maximum) cooling rate that ensures ergodicity and thus a positive probability of escaping from local optimums and eventually finding the global optimum. This optimal temperature schedule depends only on two measures of a given problem's graph ("r" and "L"). However, its applicability to implementation was rejected due to: (a) Irrelevance—The Main SA Theorem, though theoretically encouraging, is not really applicable to implementations because it assumes we will let the chain run a lot longer than we can (infinitely long, in fact). In practice, we can't really hope to sample from the complete set of global optima, but only to get an optimum that is not very local. (b) Difficulty—In most problems we don't know what the topology of the state space is and so can't estimate the needed parameters of the graph in order to determine the optimal cooling schedule. In fact, if we could, it would probably also be possible to devise a more specific and efficient algorithm for the problem than SA.

Given that computing time is the primary constraint in implementation, I devised the following universally applicable temperature scheduling methodology:
The first 1%, say, of the iterations of a run are used to "melt" the space—essentially sampling the graph at random to determine the minimum temperature at which all paths would be equally likely. From this an exponential temperature schedule is calculated to take it from a fraction like .5 of this melting point to a relatively cool temperature like .01 of the melting point at the end of the run. (Empirically, these are generally good parameters.)

Ÿ Mathematica Prototype:

'n' is reserved throughout the notebook for number of cities.
'k' determines the graph dimension--it is the number points any selected point can be exchanged with.
(Each iteration, one of the n points is selected at random to be exchanged with a random one of its k forward neighbors.)
'Solution' contains the permutation of the city sequence at each iteration of the algorithm.
'Optimum' is used to store the most efficient sequence found by the algorithm.

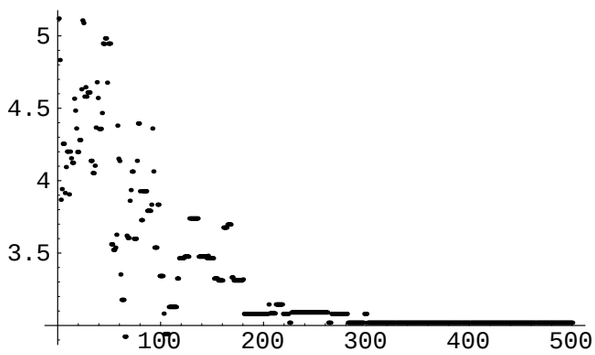
```
n = 10;
k = 2;
Solution = Table[i,{i,1,n+1}];
Optimum = Solution;
```

```
Cities = Table[{Random[Real,1,3], Random[Real,1,3]},{i,1,n}];
```

```
Distance[x_, y_] := Sqrt[(x[[1]]-y[[1]])^2+(x[[2]]-y[[2]])^2];


Fitness[list_] := Distance[Cities[[list[[1]]]],Cities[[list[[n]]]]] +
  Sum[Distance[Cities[[list[[i]]]],Cities[[list[[i-1]]]]],{i,2,n}];


Iterate[temp_,Sol_] := Block[
  {pt = Random[Integer,{1,n}],sh = Random[Integer,{1,k}],nSol = Sol,i},
  i = nSol[[pt]]; nSol[[pt]] = nSol[[Mod[pt+sh,n+1]+1]];
  nSol[[Mod[pt+sh,n+1]+1]] = i; nSol[[n+1]] = Fitness[nSol];
  If[nSol[[n+1]] < Sol[[n+1]],
    {Solution = nSol;If[nSol[[n+1]] < Optimum[[n+1]],Optimum = nSol]},
    If[Random[] < Exp[-(nSol[[n+1]] - Sol[[n+1]])/temp],Solution = nSol]];
  Solution[[n+1]]]


SA[its_] := Block[{},
  m = Ceiling[.01*its];fitM = Table[Iterate[10^10,Solution],{i,0,m}];
  range = Max[fitM] - Min[fitM];decT = .001^(1/its);
  ListPlot[Table[Iterate[range*decT^i,Solution],{i,1,its}]]]
```

Here is a 500-iteration run on a simple case of 10 cities. The first graph shows the fitness at each iteration. In the beginning the temperature is very high and the behavior is more random. Note that a near-optimum is found early on and that as the temperature is dropped toward 0.1% of the melting point, the algorithm settles on a local optimum.

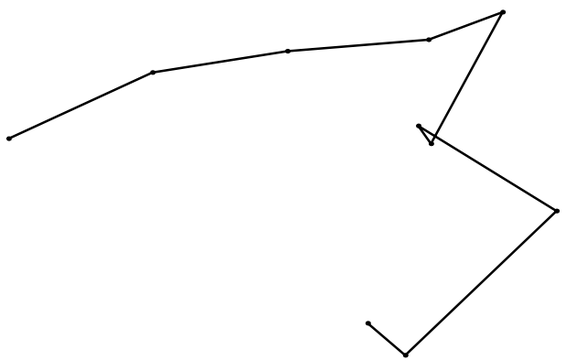
```
SA[500]
```




```
{Solution, Optimum}
```

```
{{6, 3, 1, 9, 2, 10, 5, 8, 4, 7, 2.91345},
 {9, 2, 3, 4, 1, 6, 7, 8, 5, 10, 2.66425}}
```

```
Show[Graphics[Line[Table[Cities[[Optimum[[i]]]],{i,1,n}]]],ListPlot[Cities]]
```



[The distance from the start to the endpoint is included in the fitness calculation but the solution graph is left unconnected for clarity.]

Ÿ C Implementation: (For speed it was necessary to reprogram the algorithm in C, using Mathematica only to display the results. The following code managed to execute about 10,000 iterations per minute on the 100-city problem using a 486-66 processor.)

```
/*              C code for Simulating Annealing on TSP -- David Bookstaber          */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

#define N 100  // Number of cities
#define K 70    // Dimension of state graph (range of interchange)
#define ITS 100000L   // Iterations
#define PMELT 0.7               // Fraction of melting point for starting temperature
#define TARGT 0.01   // Fraction of melting point for ending temperature
#define STAGFACTOR 0.05   // Fraction of runs to allow stagnant before reheating

typedef struct {int list[N]; float fit;} state;
float city[N][2];

float distance(int a, int b) {
        return(pow(pow(city[a][0]-city[b][0],2)+pow(city[a][1]-city[b][1],2),0.5));
}

float fitness(state x) {
        int i;
        float sum = distance(x.list[0],x.list[N-1]);
        for(i = 1;i < N;i++) sum += distance(x.list[i],x.list[i-1]);
        return(sum);
}

state iterate(float temp,state x) {
        int i, pt, sh;
        state y = x;
        pt = rand() % N;
        sh = (pt+(rand() % K)+1) % N;
        y.list[pt] = y.list[pt]^y.list[sh];
        y.list[sh] = y.list[sh]^y.list[pt];
        y.list[pt] = y.list[pt]^y.list[sh];
        y.fit = fitness(y);
        if(y.fit < x.fit) {
                return(y);
        }
        else if((float)rand()/(1.0*RAND_MAX) < exp(-1.0*(y.fit-x.fit)/temp))
                return(y);
        else
                return(x);
}

void main() {
        int i, j, k, n;
        long l, optgen;
        double minf, maxf, range, Dtemp;
        state x, optimum;
        FILE *fp;
        clrscr();
        srand(1);
/* Initialization of city grid and state list */
        for(i = 0,k = 0,n = sqrt(N);i < n;i++) {
                for(j = 0;j < n;j++,k=n*i+j) {
                        city[k][0] = i; city[k][1] = j;
                        x.list[k] = k;
                }
        }
/* Randomization of state list--requires N Log[N] "shuffles" */
        for(i = 0,k = rand()%(N-1)+1;i < N*log(N);i++,k = rand()%(N-1)+1) {
                x.list[0] = x.list[0]^x.list[k];
                x.list[k] = x.list[k]^x.list[0];
                x.list[0] = x.list[0]^x.list[k];
        }
```

```
/* Sample state space with 1% of runs to determine temperature schedule */
        for(i = 0,maxf = 0,minf = pow(10,10),x.fit=fitness(x);i < max(0.01*N,2);i++) {
                x = iterate(pow(10,10),x);
                minf = (x.fit < minf) ? x.fit : minf;
                maxf = (x.fit > maxf) ? x.fit : maxf;
        }
        range = (maxf - minf)*PMELT;
        Dtemp = pow(TARGT,1.0/ITS);
/* Simulate Annealing */
        for(optgen = l = 1,optimum.fit = x.fit;l < ITS;l++) {
                x = iterate(range*pow(Dtemp,l),x);
                if(x.fit < optimum.fit) {
                        optimum = x;
                        optgen = l;
                }
/* Reheat if stagnant */
                if(l-optgen == STAGFACTOR*ITS) Dtemp = pow(Dtemp,.05*l/ITS);
/* Graphics */
                gotoxy(1,1); printf("Iteration: %ld\t",l);
                gotoxy(1,2); printf("Fitness: %f\t\tTemp: %f\t\t",x.fit,range*pow(Dtemp,l));
                gotoxy(1,3); printf("Current Optimum %f found on %ld\t\t",optimum.fit,optgen);
                gotoxy(1,4); printf("Global Optimum is %d",N);
                gotoxy(1,5); printf("Sample Range: %f\tTemp decrement: %f",range,Dtemp);
/* End Graphics */
        }
/* Output Solution */
        fp = fopen("\\DOCS\\SA.OUT","wt");
        fputc('{',fp);
        for(i = 0;i < N;i++)
                fprintf(fp,"%d,",optimum.list[i]);
        fprintf(fp,"%f",optimum.fit);
        fputc('}',fp);
        fclose(fp);
}
```

Ÿ Mathematica commands to Load results from SA.EXE (compiled from C)

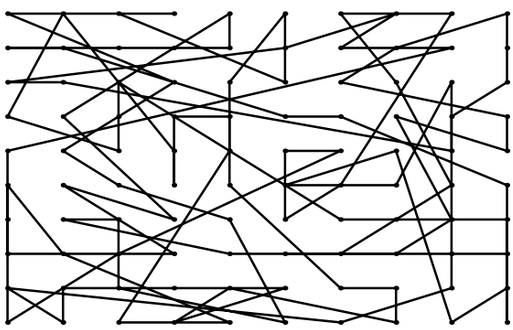
```
infile = OpenRead["\docs\sa.out"];

results = ReadList[infile][[1]];

Close[infile];
```

Ÿ Results

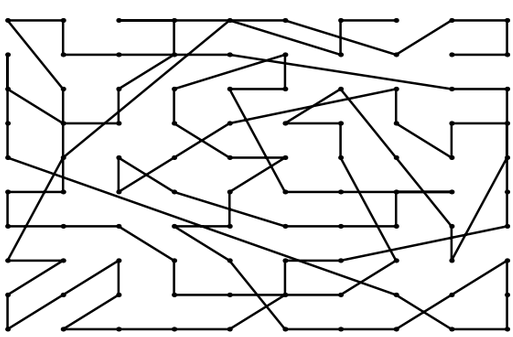
Following are samples of runs on a 100-city TSP. The cities are given in a square grid so that we can tell how close the algorithm came to the global optimum (which in this case we know to have length 100). Though this particular case is easy for us, for the algorithm it is a formidable problem given here only 100,000 or 1,000,000 samples per run in a search space of 99! states (that's about 10^156). Presumably its performance on this problem will be similar to that on problems for which we aren't sure of the global optimum.

[Admittedly there are in this huge space also a very large number of optimal and near-optimal solutions. However, we can put a (very) rough estimate on the upper bound of the number of such solutions: Suppose we can start at any point on the graph, and that at most we have 3 adjacent points from which to choose so as to follow an optimal path (left, right, and straight, though often one or more of these directions is obscured by the side of the graph or a point already touched). Such a path should be within 10 Sqrt[2] of the optimum of 100 in length (because it may end on the exact opposite side of the graph and have to take the diagonal back to where it began). The upper bound on the number of such solutions is therefore 100*3^98 or about 10^49, but still only 10^(-105) of the search space!]

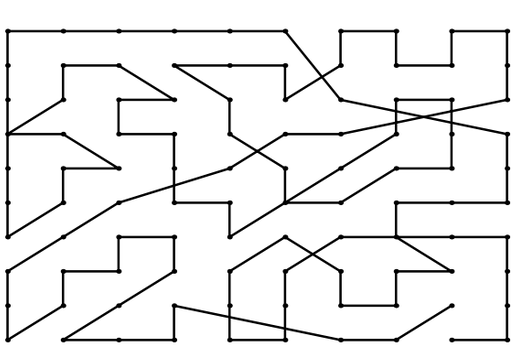
Ÿ **Here is a partial result at 1,000 iterations with a fitness of 251.687:**



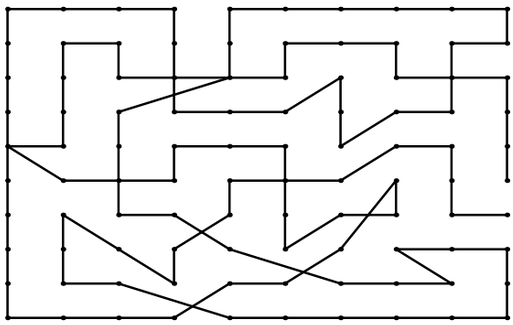
Ÿ **After almost 10,000 iterations it is more settled, but this obviously requires cooling much too quickly given the size of the search space. This example has fitness 156.16:**



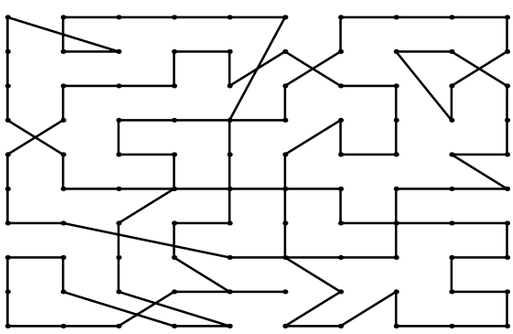
Ÿ **This run consisted of 100,000 iterations but was cooled to 0.1% of the melting point--apparently too quickly because this solution (with length 121.072) was found at iteration 48681.**



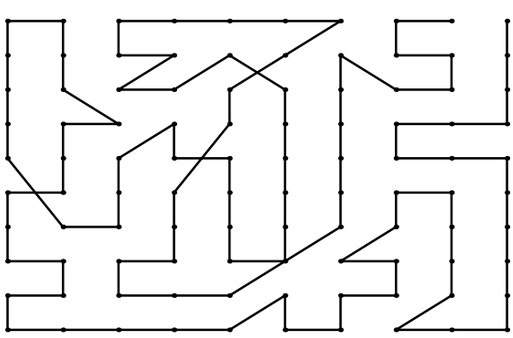
Ÿ **Again with 100,000 iterations, here cooled to only 1% of melting point, this record optimum (115.915) was found at iteration 96874, indicating a good temperature schedule.**



Ÿ **Trying to cool even more slowly to just 10% of the melting point on 100,000 iterations, this run's solution had fitness 122.482 found at iteration 99917, indicating that the algorithm was still settling when it was stopped.**



Ÿ **This run of 1,000,000 iterations cooled to 0.1% of the melting point, significantly outperforming the runs of 100,000. This solution has length 108.857, found at iteration 485,954.**



Ÿ This run of 1,000,000 iterations cooled to 1% of melting point and gave the best solution at iteration 716,175, having length 106.142.