Simulated Annealing applied to the Traveling Salesman Problem

Eric Miller

INTRODUCTION

The traveling salesman problem (abbreviated TSP) presents the task of finding the most efficient route (or tour) through a set of given locations. Each location should be passed through only once, and the route should loop so that it ends at the starting location. This paper will focus on the symmetric TSP, in which the distance between any two locations is the same in both directions.

The TSP was introduced in 1832 in a German business manual written for the “traveling salesman.” The manual explains that a traveling salesman should take an efficient tour through cities in Germany in an attempt to save time and visit as many cities as possible. The manual did not use any mathematical techniques to solve the problem, but suggested tours throughout 45 German cities that could potentially save the most time.

Though historical details are somewhat hazy on the matter, it is thought that mathematical study of the TSP was likely begun in 1930 by a mathematician named K. Menger (it should be noted that two mathematicians, Euler and Hamilton, were both influential in laying the groundwork for study of the TSP). One of Menger’s conclusions was that the technique of simply always choosing the next closest city to determine the best tour (Nearest Neighbor Algorithm) is not a consistently effective solution to the problem. The TSP gained notable mathematical interest from then on and research on how to improve solutions to the problem is still done today.

So what is so difficult about solving the TSP? Theoretically, one could use what is referred to as “brute-force search” and consider each possible tour through the given set of cities to determine which tour is the shortest. This is easy enough for small sets, but it doesn’t take very many cities in a given set for that kind of computation to become extremely expensive. Consider the small set of points $\{A, B, C, D, E\}$. The number of possible tours beginning and ending with point A and traversing every other point exactly once is equal to the number of permutations of the set $\{B, C, D, E\}$, which is $4! = 24$. Furthermore, as far as the symmetric TSP is concerned, the tour A, B, C, D, E, A is equivalent to the tour A, E, D, C, B, A since both take the same distance and one is easily deduced from the other. Thus, we can generalize the number of possible tours to consider for any set of size n to be $\frac{(n-1)!}{2}$. Unfortunately, the 2 in the denominator does little to limit how fast this expression grows as n increases. Obviously, if one wants to solve the TSP for any large n, an alternative method to brute-force search most be utilized.

The TSP is a Non-deterministic Polynomial-time--hard problem (abbreviated NP-hard). "Polynomial-time," refers to algorithms in which the running time remains proportional to n^k, where n is the size of the input and k is some constant. Keeping this definition in mind, a NP problem essentially refers to problems for which proposed solutions can be identified as valid or invalid, and furthermore, valid solutions can be tested in polynomial time. A NP-complete problem has these same properties, with a further stipulation that any NP problem can be reduced to the NP-complete problem in polynomial time. Finally, a NP-hard problem has the property that a NP-complete problem can be reduced to the NP-hard problem in polynomial time. The goal is to find an exact, generalizable solution to the TSP (or other NP-hard problems) in polynomial-time. All problems in the NP-hard class are related in such a way that solving one problem with an algorithm would essentially solve all NP problems. As one can gather from the term, NP-hard problems are hard to solve.

METHODS

It is possible that an algorithm does not exist in polynomial-time that can be generalized for any instance of the TSP. However, many extremely useful algorithms have been developed that can provide a close approximation to the true solution of particular occurrences of the TSP in relatively short amount of time. These algorithms employ techniques referred to as *heuristics*, which are used to quickly obtain a near-optimal solution that is considered close enough to the global optimum solution to be useful. This paper will discuss one of these heuristic methods; namely, simulated annealing.

Simulated annealing was a method introduced in 1983 by Scott Kirkpatrick, Daniel Gelatt and Mario Vecchi. To understand the motivation for applying simulated annealing to the TSP, it is useful to first look at another method of optimization known as *greedy search*. Greedy search is a simple heuristic that breaks down large problems (like the TSP) into smaller yes-or-no choices that are always determined by whatever is immediately most beneficial. Consider a landscape with a large number of hills and valleys, and suppose that there is an objective to locate the highest point on this landscape in a timely manner. One way to do this would be to arbitrarily pick a spot to start within the cluster of hills, and then only take steps in directions that result in a highest gain in altitude. When there are no more upward steps available, the process is completed and the current hill is declared the highest. This is known as a *hill-climbing*; a type of greedy search. The tallest hill in this scenario is analogous to the global optimum solution tour of a particular TSP, and each location on the landscape corresponds to a specific tour. Taking a step is equivalent to slightly modifying the tour. Unfortunately, one of the major drawbacks to this technique is that it is very likely to have only led to a local maximum, and, returning to the landscape analogy, there is no indication that the height of the hill chosen is even close to the height of the tallest hill. One solution would be to just repeat this method many times, but doing so is costly in regards to time.

A better solution is to occasionally take steps to a lower altitude, thus allowing the search for a higher hill in areas that wouldn't have been accessible otherwise. This is one of the key concepts of simulated annealing. With simulated annealing, a random tour is chosen and its permutation is slightly modified by switching the order of as few as two points to obtain a new tour. If this new tour's distance is shorter than the original tour, it becomes the new tour of interest. Else, if this new tour has a greater distance than the original tour, there exists some probability that this new tour is adopted anyways. At each step along the way, this probability decreases to 0 until a final solution is settled upon.

Simulated annealing is named after a heat treatment process used on metals. During the annealing process, a metal is heated enough to allow its molecular structure to be altered. The temperature is then steadily lowered, which subsequently lowers the energy of the atoms and restricts their rearrangement until the metal's structure finally becomes set. This process minimizes the number of defects in the structure of the material. Similarly, *simulated* annealing slowly and increasingly restricts the freedom of the solution search until it only approves moves toward superior tours. The probability of accepting an inferior tour is defined by an *acceptance probability function*. This function is dependent on both the change in tour distance, as well as a time-dependent decreasing parameter appropriately referred to as the Temperature. The way that the Temperature changes is referred to as the Cooling Schedule. There is no single way to define the probability acceptance function or the Cooling Schedule in such a way that will work well for any problem, so this paper will simply utilize a commonly used acceptance probability function (known as Kirkpatrick's model) and then explore the effects of changing the Cooling Schedule.

The Algorithm

Definitions**:**

State (s): a particular tour through the set of given cities or points

Neighbor State (s'): state obtained by randomly switching the order of two cities

Cost Function (C): determines the total cost (Euclidian distance) of a state

Relative Change in Cost (δ): the relative change in cost c between s and s'

Cooling Constant (β): the rate at which the Temperature is lowered each time a new solution is found

Acceptance Probability Function (P): determines the probability of moving to a more costly state

n = number of cities or points

T_0 = Initial Temperature

T_k = the Temperature at the k^{th} instance of accepting a new solution state

$T_{k+1} = \beta T_k$, where β is some constant between 0 and 1

$$P(\delta, T_k) = \begin{cases} e^{\left(\frac{-\delta}{T_k}\right)}, & \delta > 0 \\ 1, & \delta \leq 0 \end{cases} \quad for\ T_k > 0.$$

Note that for $\delta > 0$, for any given T, P is greater for smaller values of δ. In other words, a state s' that is only slightly more costly than s is more likely to be accepted than a state s' that is much more costly than s. Also, as T decreases, P also decreases. In mathematical terms, $\lim_{T \to 0^+} e^{\left(\frac{-\delta}{T_k}\right)} = 0, \text{ for } \delta > 0.$

Pseudocode:

```
1) Choose a random state $s$ and define $T_0$ and $\beta$
2) Create a new state $s'$ by randomly swapping two cities in $s$
3) Compute $\delta = \frac{C(s') - C(s)}{C(s)}$
    a. If $\delta \leq 0$, then $s = s'$
    b. If $\delta > 0$, then assign $s = s'$ with probability $P(\delta, T_k)$
        i. Compute $T_{k+1} = \beta T_k$ and increment $k$
4) Repeat steps 2 and 3 keeping track of the best solution until stopping conditions are met
```

Stopping Conditions:

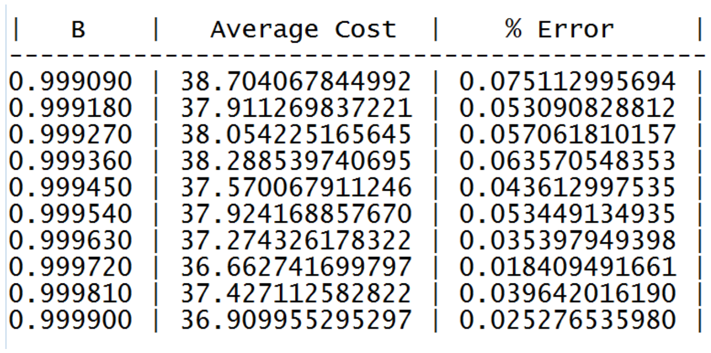
The stopping conditions are quite important in simulated annealing. If the algorithm is stopped too soon, the approximation won't be as close to the global optimum, and if it isn't stopped soon enough, wasted time and calculations are spent with little to no benefit. For the stopping conditions in my function, I will specify a goal cost and a minimum temperature. If either of those values is reached, the search will stop. I will also periodically check to ensure that the cost of the most optimal state found so far is changing. If it doesn't change within a particular period of iterations, the search will be stopped to limit unproductive work.

RESULTS

To test my simulated annealing function, I use problems that have a known global solution so that I can compare my results. For each problem's stopping conditions, I set the global optimum as the goal, $10e-6$ as the minimum temperature, and a solution change check after every 50,000 iterations. These stopping conditions seemed to result in a good medium between time and accuracy (It should be noted that better results can certainly be obtained if time and computational cost is of little importance). Given these stopping conditions, optimal values for T_0 and a corresponding interval for β are found through guess and check. The corresponding solution is analyzed for correctness.

Problem 1: 6x6 grid of 36 points, each placed one unit apart, horizontally and vertically. The optimal tour through these points is 36 units in length.

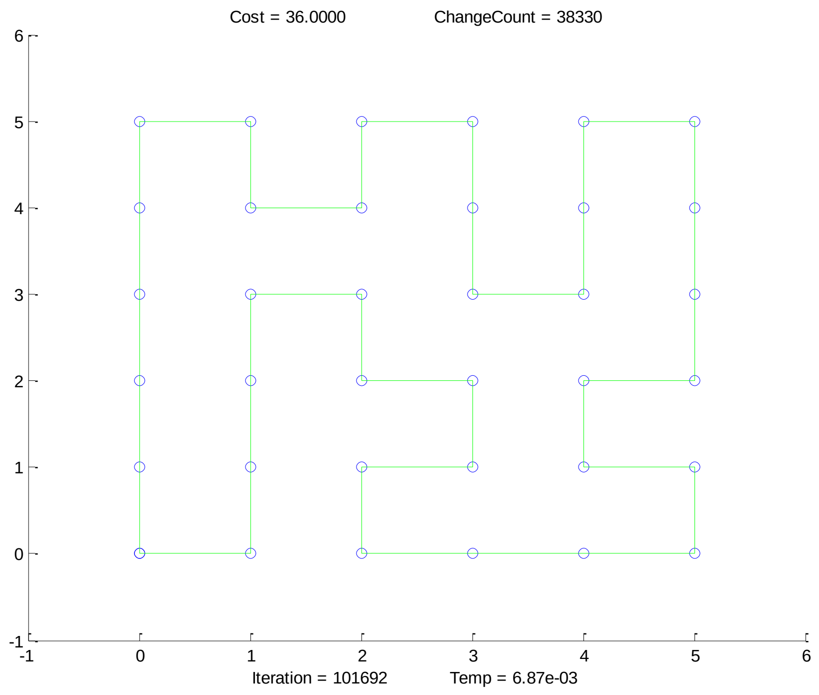
Initial trial runs suggested that good results are obtained when $T_0 = 10$ and $0.999 \leq \beta \leq 0.9999$. Testing 10 values equally spaced values in this interval 5 times each and computing the average gave these results:

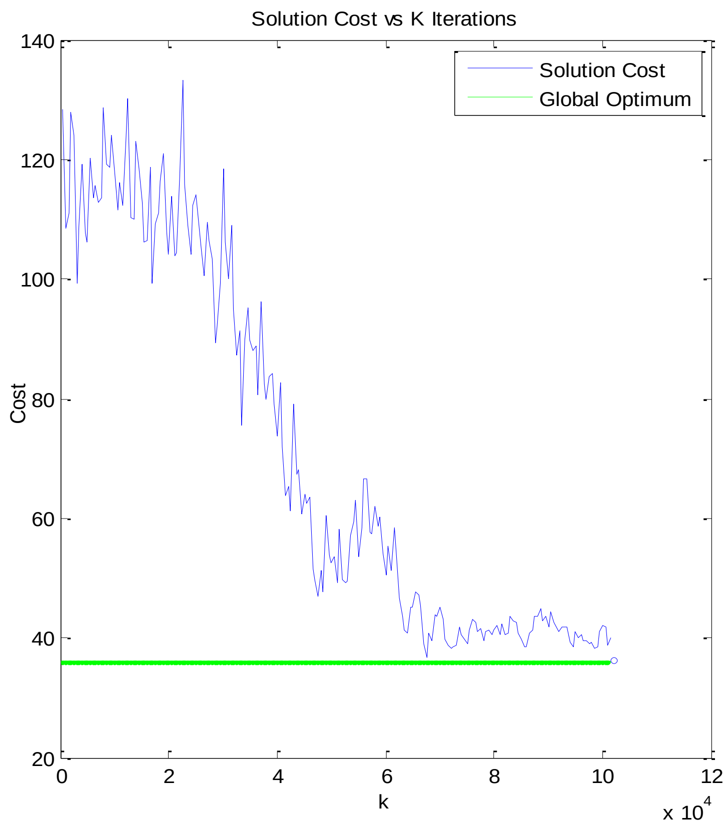


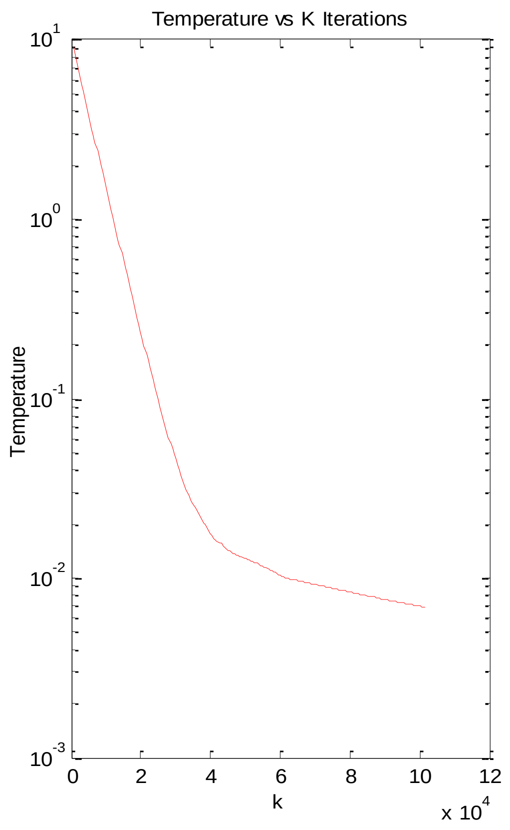
```
|    B     |   Average Cost   |     % Error     |
-------------------------------------------------
0.999090 | 38.704067844992 | 0.075112995694 |
0.999180 | 37.911269837221 | 0.053090828812 |
0.999270 | 38.054225165645 | 0.057061810157 |
0.999360 | 38.288539740695 | 0.063570548353 |
0.999450 | 37.570067911246 | 0.043612997535 |
0.999540 | 37.924168857670 | 0.053449134935 |
0.999630 | 37.274326178322 | 0.035397949398 |
0.999720 | 36.662741699797 | 0.018409491661 |
0.999810 | 37.427112582822 | 0.039642016190 |
0.999900 | 36.909955295297 | 0.025276535980 |
```

There doesn't seem to be a tremendous difference, but we will look at the cooling constant corresponding with the smallest average cost, $\beta = 0.99981$.

Here are plots corresponding to a run with $\beta = 0.99981$ where the algorithm finds the global optimum:

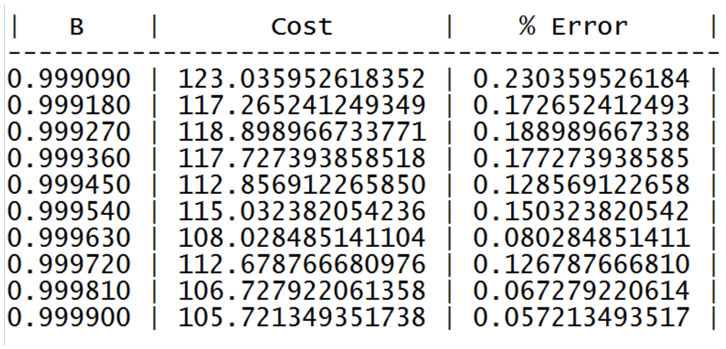

Cost = 36.0000
ChangeCount = 38330
Iteration = 101692
Temp = 6.87e-03


Solution Cost vs K Iterations
Solution Cost
Global Optimum
Cost
k
x 10^4


Temperature vs K Iterations
Temperature
k
x 10^4

Problem 2: 10x10 grid of 100 points, each placed one unit apart, horizontally and vertically. The optimal tour through these points is 100 units in length.

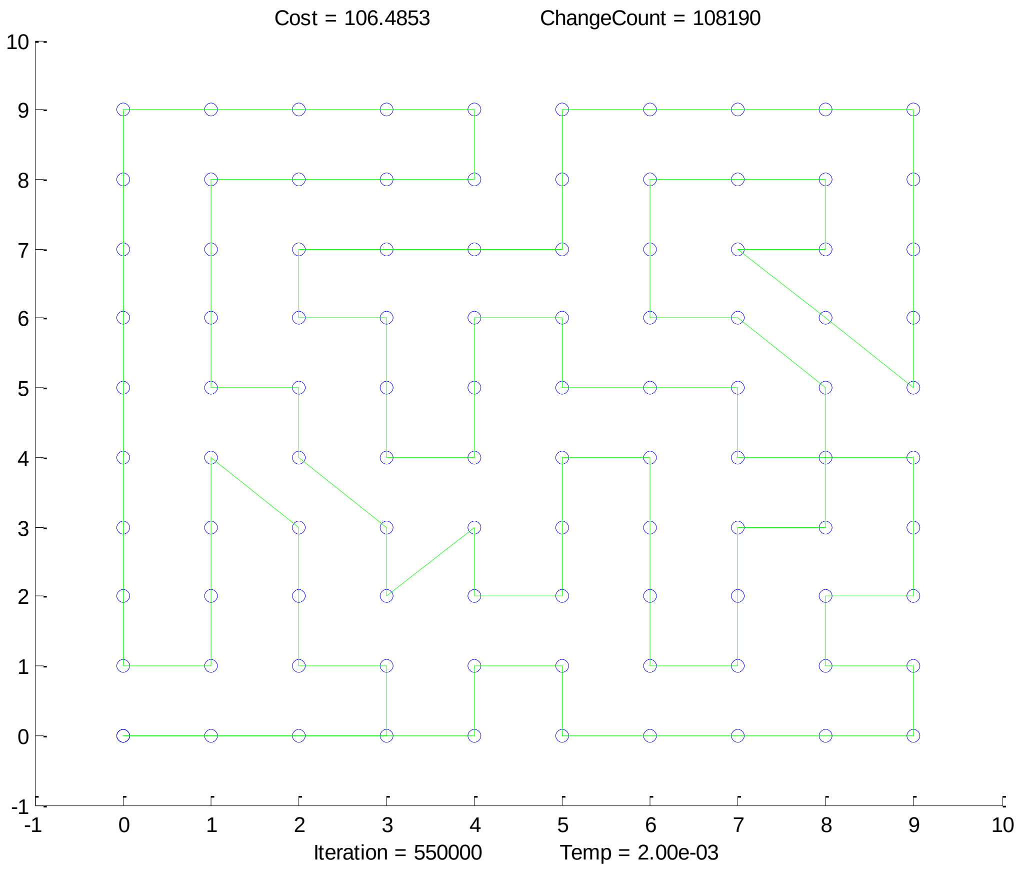
Through guess and check work, a value of $T_0 = 100$ coupled with a similar interval for β as in problem 1 seemed most appropriate for this problem.

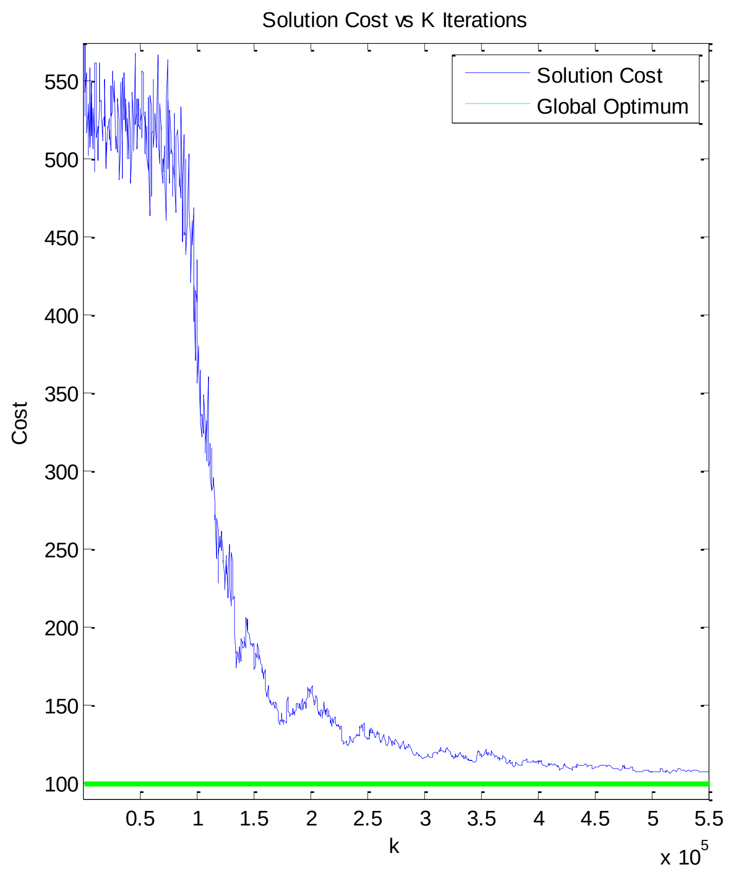


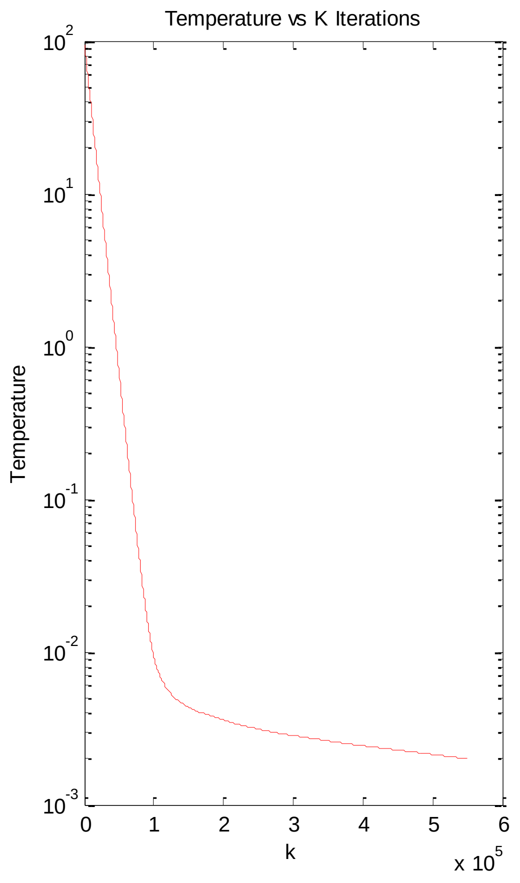
```
|    B     |        Cost        |     % Error     |
--------------------------------------------------
0.999090 | 123.035952618352 | 0.230359526184 |
0.999180 | 117.265241249349 | 0.172652412493 |
0.999270 | 118.898966733771 | 0.188989667338 |
0.999360 | 117.727393858518 | 0.177273938585 |
0.999450 | 112.856912265850 | 0.128569122658 |
0.999540 | 115.032382054236 | 0.150323820542 |
0.999630 | 108.028485141104 | 0.080284851411 |
0.999720 | 112.678766680976 | 0.126787666810 |
0.999810 | 106.727922061358 | 0.067279220614 |
0.999900 | 105.721349351738 | 0.057213493517 |
```

Here it seems that higher values for β are more effective, so we will study the results of $\beta = 0.9999$.

Running these parameters 10 times, here is the best result:





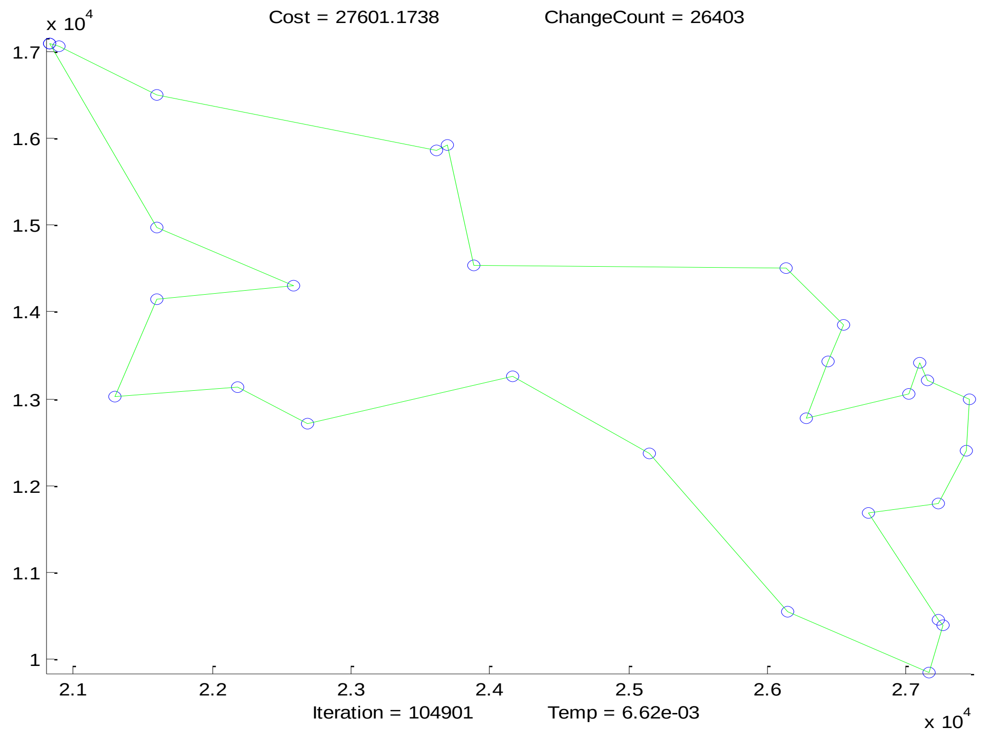


Problem 3:

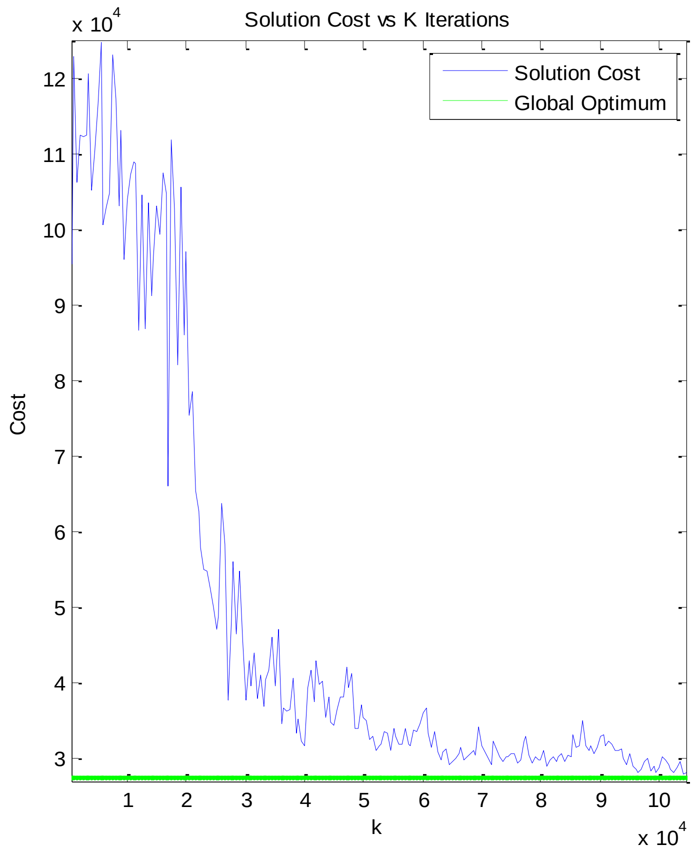
29 cities in Western Sahara. The global optimum is known to have length 27603.

(data obtained from http://www.math.uwaterloo.ca/tsp/world/countries.html)

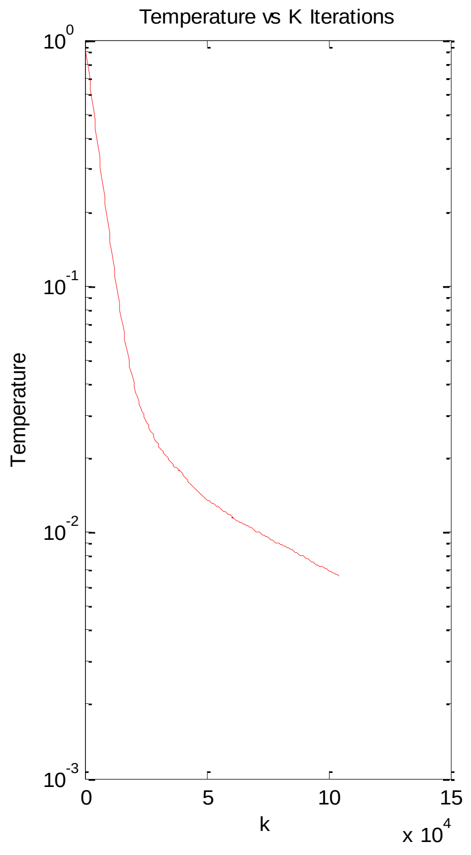
Setting $T_0 = 1$ and $\beta = 0.99981$ (the same values as in problem 1) running the program 10 times I was able to find the following solution:



Cost = 27601.1738
ChangeCount = 26403
x 10^4
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
2.1
2.2
2.3
2.4
2.5
2.6
2.7
Iteration = 104901
Temp = 6.62e-03
x 10^4



Solution Cost vs K Iterations
x 10^4
Solution Cost
Global Optimum
Cost
12
11
10
9
8
7
6
5
4
3
1
2
3
4
5
6
7
8
9
10
k
x 10^4



Temperature vs K Iterations
Temperature
10^0
10^-1
10^-2
10^-3
0
5
10
15
k
x 10^4

DISCUSSION

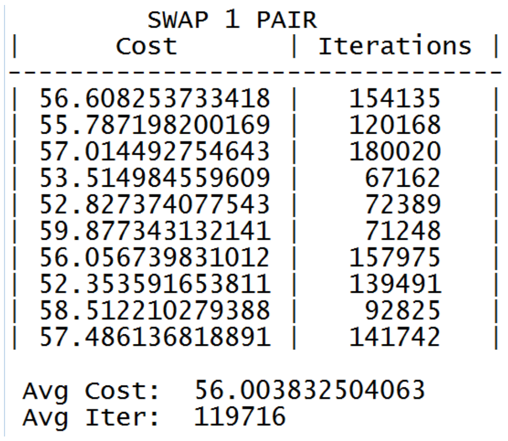
It seems that the key to getting simulated annealing to work lies in the cooling schedule. The most difficult obstacle in my experimentation was the amount of time it took to test different parameters. It quickly became apparent that changing either β and T_0 by just a little bit could produce much different results. If I had more time (or perhaps if I had a faster computer) I would run my function many more times for different values of β and T_0 to find the optimum combination for each problem. I would also like to be able to compare the average time it takes to find the global solution for varying sizes of problem sets.

While watching the plot throughout the course of some of the solution searches, I noticed that that the search would frequently get stuck in a local minimum even if β was extremely close to 1, particularly for larger problem sets with separate clusters of points. Through my research, I found some cooling schedules that are more adaptive to a variety of problem sets, though they are more complex than what I have implemented. I'd be interested to see how much better these cooling schedules perform than mine.

Problem 3 raised a few questions. For one, my function claims to have found a more optimal solution than what is stated by the source of the data. Unfortunately, I can't figure out where the mistake is. It could be that my cost function is making some slight errors, or perhaps the provided data has a slight error.

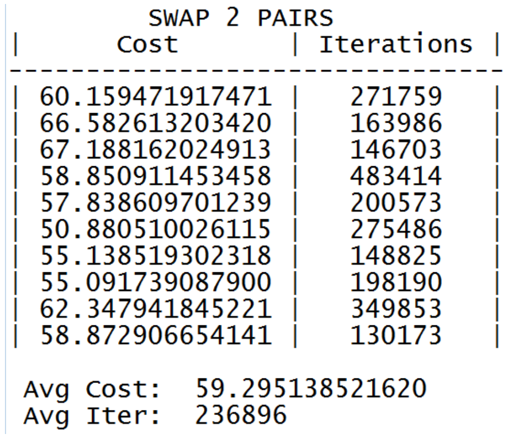
I was particularly interested in finding out if swapping more than one pair of cities could affect the results. Watching the plot as my function searched for a solution, it seemed like being able to switch two cities at once could help break out of local minimums. I generated a new problem set of 40 random points between intervals $x = [0,10]$ and $y = [0,10]$ and modified my function to swap two pairs of cities at once. I also modified the function to output the exact iteration at which the best solution was actually found.

By means of guess and check, $\beta = 0.99981$ and $T_0 = 5$ seemed to produce satisfactory results for both functions. Below is a table comparing the results over 10 test runs of this modified function and the original function.



```
          SWAP 1 PAIR
|       Cost         | Iterations |
-------------------------------
| 56.608253733418 |    154135    |
| 55.787198200169 |    120168    |
| 57.014492754643 |    180020    |
| 53.514984559609 |     67162    |
| 52.827374077543 |     72389    |
| 59.877343132141 |     71248    |
| 56.056739831012 |    157975    |
| 52.353591653811 |    139491    |
| 58.512210279388 |     92825    |
| 57.486136818891 |    141742    |

 Avg Cost:   56.003832504063
 Avg Iter:   119716
```



```
          SWAP 2 PAIRS
|       Cost         | Iterations |
-------------------------------
| 60.159471917471 |    271759    |
| 66.582613203420 |    163986    |
| 67.188162024913 |    146703    |
| 58.850911453458 |    483414    |
| 57.838609701239 |    200573    |
| 50.880510026115 |    275486    |
| 55.138519302318 |    148825    |
| 55.091739087900 |    198190    |
| 62.347941845221 |    349853    |
| 58.872906654141 |    130173    |

 Avg Cost:   59.295138521620
 Avg Iter:   236896
```

Here we see that swapping 2 pairs at once actually produced worse solutions in close to twice the amount of iterations. It's possible that swapping more than one pair of points could help for other problem sets, but it certainly did not help in this case.

In conclusion, it seems that simulated annealing is quite useful, though its effectiveness is extremely reliant on the choosing of an appropriate cooling schedule. Finding such a cooling schedule can be time consuming. A better, more adaptable way of setting the cooling schedule from problem to problem would greatly improve my implementation of the algorithm.

It is quite apparent that simulated annealing requires far less computation than brute-force search and often times is still able to find the global optimum for smaller problem sets like the ones studied in this paper.

REFERENCES

Bookstaber, D. (2014). Simulated Annealing for Traveling Salesman Problem. Retrieved December 1, 2014, from http://www.eit.lth.se/fileadmin/eit/courses/ets061/Material2014/SATSP.pdf

Cook, W. J. (2011). *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton: Princeton University Press.

Cook, W. (n.d.). Traveling Salesman Problem. Retrieved December 7, 2014, from http://www.math.uwaterloo.ca/tsp/index.html

Hong, P.-Y., Lim, Y.-F., Ramli, R., Khalid, R., & International Conference on Mathematical Sciences and Statistics 2013, ICMSS 2013. (November 11, 2013). Simulated annealing with probabilistic analysis for solving traveling salesman problems. *Aip Conference Proceedings, 1557,* 515-519.

Stefanoiu, D., Borne, P., Popescu, D., Filip, F. G., & El, K. A. (2014). *Optimization in Engineering Sciences: Approximate and Metaheuristic Methods*. Hoboken: Wiley.

CODE

SA.m – Simulated Annealing File

aprobfun.m – Acceptance Probability Function

cost.m – Cost function

switchCities.m – Switches the order of two columns

```
function P = aprobfun(d,T)
% Acceptance probability function - returns probability that a new state
% will be accepted given the relative change in cost and current temperature.
% d = (cost of new state - cost of current state)/cost of current state
% T = current temperature

if d <= 0
    P = 1;
elseif d > 0
    P = exp(-d/T);
end
```

```
function c = cost(s)
% Determines the cost (total distance) of state s using the norm
% function. Input s is a 2xn matrix where columns correspond to the
% coordinates of a state, with the first city repeated at the end.

n = size(s,2);
c = 0;
for i = 1:n-1
    c = c + norm(s(:,i)-s(:,i+1));
end
```

```
function f = switchCities(cities)
% switches order of two random cities for Simulated Annealing
% cities: 2xn matrix containing looped city coordinates

n = length(cities(1,:)) - 1;
p1ind = randi(n-1)+1;
p2ind = randi(n-1)+1;
snew = cities;
snew(:,[p1ind p2ind]) = snew(:,[p2ind p1ind]);
f = snew;
```

```
function [cbest,kSol,k,sbest] = SA(xdata,ydata,T0,B,goal,kCheck,tempMin)
% Runs a simple Simulated Annealing algorithm for solving Traveling
% Salesman Problems.
% INPUT:
% xdata = x coordinates of each city in a horizontal array
% ydata = y coordinates of each city in a horizontal array
% T0 = Initial Temp, B = Cooling Constant
% Goal = Stopping Cost
% kCheck = # of iterations after which to check if new solution was found
% tempMin = Minimum Temperature to reach before stopping
% OUTPUT:
% cbest = cost of the best found solution state
% kSol = iteration at which the best solution was found
% k = total iterations actually performed
% sbest = matrix of coordinates in order of best route
% Example Data:
%xx = [0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 7 7 7 7
7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9 9];
%yy = [0 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 9 8 7 6 5
4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 9 8 7 6
5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1 0];

n = length(xdata); % # of cities
% Define starting points
x1 = xdata(:,1);
y1 = ydata(:,1);
% Remove starting point from data that will be rearranged
xdata(:,1)=[];
ydata(:,1)=[];
% Make a random permutation of the cities, excluding the first
z = randperm(n-1);
xdata = xdata(z);
ydata = ydata(z);
% Add starting city back to beginning and end
s0 = [x1 xdata x1;y1 ydata y1];
sbetter = s0;
cbest = cost(s0);
clastbest = cbest;
stop = false;
changeCount = 0;
count = 0;
k = 0;
T = T0;
while T > tempMin && cbest > goal && ~stop
    % Switch 2 random points
    snew = switchCities(sbetter);

    % Option to switch another pair of points:
    %snew = switchCities(snew);

    cbetter = cost(sbetter);
    d = (cost(snew) - cbetter)/cbetter;
    P = aprobfun(d,T);
    randomfrac = rand(1);
    if randomfrac <= P
```

```
        sbetter = snew;
        T = T * B;
        changeCount = changeCount + 1;
    end
    k = k + 1;
    if cbetter<cbest
        cbest = cbetter;
        sbest = sbetter;
        kSol = k;
    end
    % Check if a new sbest has been found within specified # of iterations
    if mod(k,kCheck)==0
        if clastbest == cbest
            stop = true;
        end
        clastbest = cbest;
    end
    % Real time plotting of solutions
    if mod(k,500)==0
    hold off
    pause(eps)
    scatter(s0(1,:),s0(2,:))
    axis ([min(s0(1,:))-1,max(s0(1,:))+1,min(s0(2,:))-1,max(s0(2,:))+1])
    hold on
    plot(sbetter(1,:),sbetter(2,:),'g')
    str1 = sprintf('Cost = %5.4f                 ChangeCount =
%d',cost(sbetter),changeCount);
    str2 = sprintf('Iteration = %d            Temp = %2.2e',k,T);
    xlabel(str2)
    title(str1)
    end
    %}
    % create arrays for plotting solution cost progress
     if mod(k,500)==0
         count = count + 1;
    ccbest(count) = cbetter;
    kk(count) = k;
    TT(count) = T;
    end
    %}
end
% Plot of solution and temp progress
hold off
figure;
subplot(1,2,1)
plot(kk,ccbest,kk,goal,'g')
axis([-inf, inf, goal - 1, inf]);
title('Solution Cost vs K Iterations')
xlabel('k')
ylabel('Cost')
legend('Solution Cost','Global Optimum')
subplot(1,2,2)
semilogy(kk,TT,'r')
title('Temperature vs K Iterations')
xlabel('k')
ylabel('Temperature')
% Plot of final solution
```

```
figure;
scatter(s0(1,:),s0(2,:))
axis ([min(s0(1,:))-1,max(s0(1,:))+1,min(s0(2,:))-1,max(s0(2,:))+1])
hold on
plot(sbest(1,:),sbest(2,:),'g')
str1 = sprintf('Cost = %5.4f                 ChangeCount =
%d',cost(sbest),changeCount);
str2 = sprintf('Iteration = %d            Temp = %2.2e',k,T);
xlabel(str2)
title(str1)
%}
end
```