Travelling Salesman Problem Solved Using Simulated Annealing

O. Abdurazakov[1]

[1] ***NC State*** *University, Department of Physics, Raleigh, NC 27695*

Here, we study the traveling salesman problem (TSP) in terms of the simulated annealing method (SA) and compare obtained results with those from the brute force method. For a small number cities both method yield similar results. We also obtain probable shortest distances for the larger number of cities on a lattice using the SA method. The SA method proves to be more efficient and practical for a large number of cities due to its simplicity and computationally feasibility.

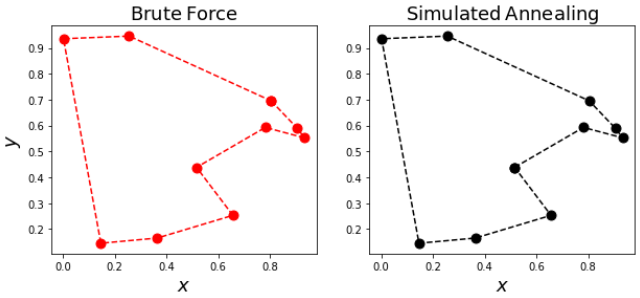

FIG. 1. Comparison of the shortest paths obtained from the brute force method (left panel) and the simulated annealing method (right panel) for seven cities.

I. INTRODUCTION

The traveling salesman problem (TSP) is a well-know optimization problem, which has been studied for many decades. It is about finding the shortest possible path a salesman can take to span a given number of destinations and return back. It is considered to a be NP-hard problem, and many exact and heuristic method have been developed to tackle it. In this project, we study TSP in terms of the simulated annealing method by generating random points on a square lattice of size thirty and more points. The method is inspired by the procedure of annealing metals to obtain desired crystal structure. After being heated, the temperature of an alloy or a metal is slowly decreased during which the system is allowed to explore many possible configurations so that, eventually at low enough temperatures, the crystal structure settles into the lowest possible energy state free of dislocation and imperfections. In a similar fashion, SA algorithm allows the system to explore a broader landscape in a given phase space so that the system does not get trapped in local extrema in the search of global extrema. In our work, we explore various number of cities or points in terms of SA and compare the results to those obtained by brute force.

II. METHODS

We use one of the simplest simulated annealing algorithms to study TSP in our work. We have a number of cities on a square lattice whose coordinates generated randomly by the uniform random generator. Our task is to minimize the total distance or the cost Δduring the travel which is the sum of distances between adjacent cities $d_{i,j}$ along a path. Therefore, $\Delta = \sum_{i,j} d_{i,j}$ where $i \neq j$. Also, we assign a control parameter to the system–an defective temperature T. Initially, an arbitrary configuration is chosen, which is not necessarily the shortest one. We let the system explore other possible configurations by flipping the indices of two randomly chosen cities and accept this move if the cost is lowered or with the probability of $P = \exp(-\Delta/T)$ otherwise. In this way, some 'unfavorable' configurations are also allowed which might 'guide' the system toward the global minimum. The the procedure is repeated at lower temperatures. As it is done in metal annealing, the effective temperature is slowly lowered. Eventually, the system settles into a reasonable optimal configuration in terms of the cost, which is the the total distance along a path.

Initially, we study the problem exactly by calculating the total distance of all possible configurations for seven cities, so there are 7! number of possibilities and 6!/2 possibilities when the over counting is considered. We can see that the number of configurations grow very fast, so the brute force method is no longer a viable route. Then we look for the shortest path among them. We compare the results for seven cities obtained using both methods.

III. RESULTS

The results are obtained for seven randomly generated points (cities) on a square lattice by brute force and SA. We show the configurations with the shortest paths in Fig. 1. We can see that both methods give the same configuration of cities. The shortest distance is approximately 3.13 in both cases. However, note that we need to permute over 7040 different configurations by brute force.

Now we can explore more number of cities on our lattice. Figure 2 displays the obtained optimal configuration for 30, 50, and 100 cities. The corresponding paths length of these optimum configurations are approximately 4.32 ,6.21,and 11.76 respectively. It would be computationally impossible to treat these number of cities by brute force. To get more insight into how the

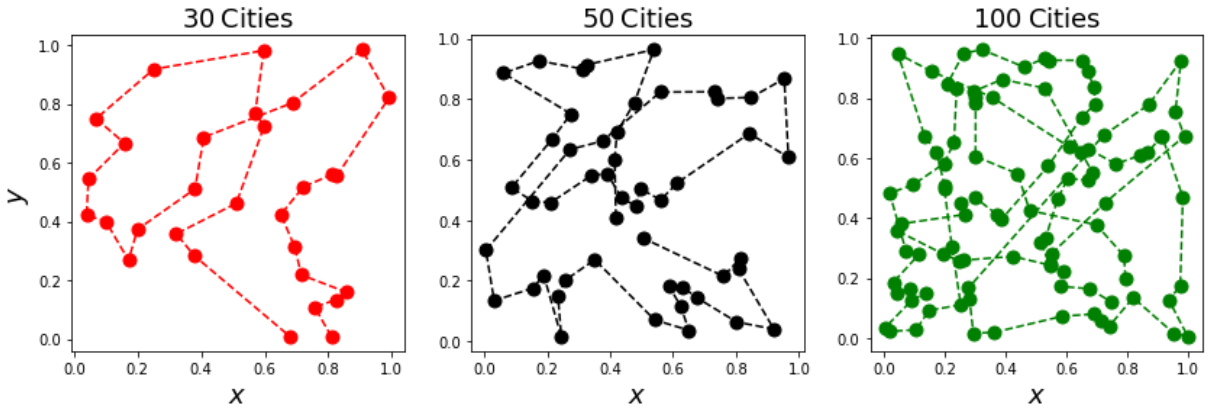

FIG. 2. The shortest paths obtained from the simulated annealing method for thirty, fifty, and hundred cities in a square lattice.

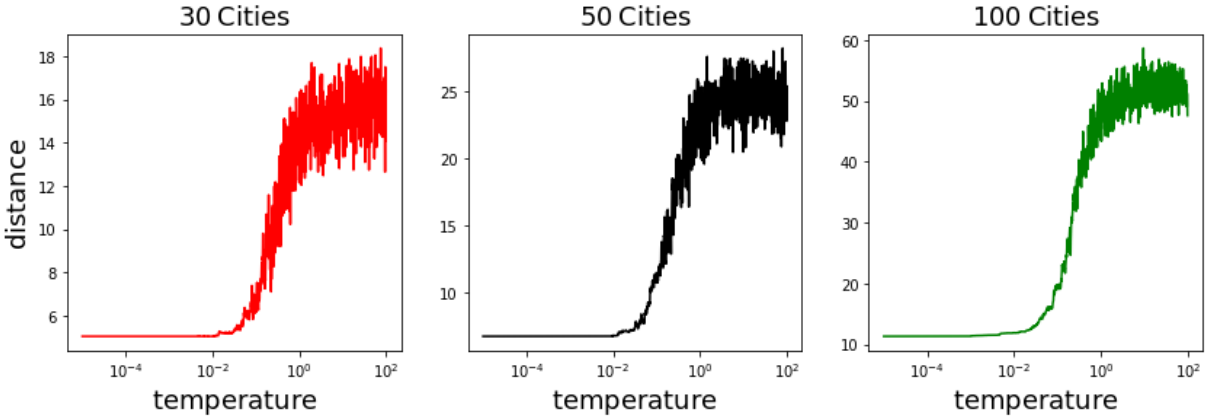

FIG. 3. The length of the shortest paths as functions of an effective temperature using simulated annealing method for thirty, fifty, and hundred cities. Here, the path length and the temperature have the same units (or are taken to be unitless)

these configurations are obtained we plot the the total distance versus the annealing temperature in Fig. 3. We see that at hight temperatures, the cost function (distance) fluctuates showing that the system is able to explore various configurations. However, as the temperature decreased the available phase space is rapidly contracted. Eventually, the system settles possibly into the configuration of shortest path length. The cost function behaves in the same fashion as a function of the effective temperature with respect to the number of cities.

In our algorithm, at each temperature we repeat the flipping of the indices of randomly chosen cities multiple times so that the system can explore more phase space. Although without this performing this procedure we obtain reasonable results, our results improve with increasing the number of these iterations, and it saturates around 50 as shown in Fig. 4. We can observe that the cost function goes down with increasing the number of iterations showing towards the true global optimum configuration. Therefore, we performed SA procedure with at lease 50 iterations at each temperature.

IV. CONCLUSIONS

In this mini-project, we study the traveling salesman problem using the simulated annealing algorithm. Despite its simplicity, it yields a reasonable results for large number of cities in finding the optimal configuration in terms of the distance traveled by the salesman. It is computationally relatively inexpensive method because it does not sample all possible configuration space but only 'thermodynamically' relevant domains. We obtained the possible shortest path for thirty, fifty, and hundred cities randomly generated on a square lattice. For small number of cities, it yields the same results as those obtained by the exact method used in the work. We showed this by simulating the path span over seven cities. It offers better solutions if one allows the system to explore more configurations by increasing the number of iteration at each system temperature.

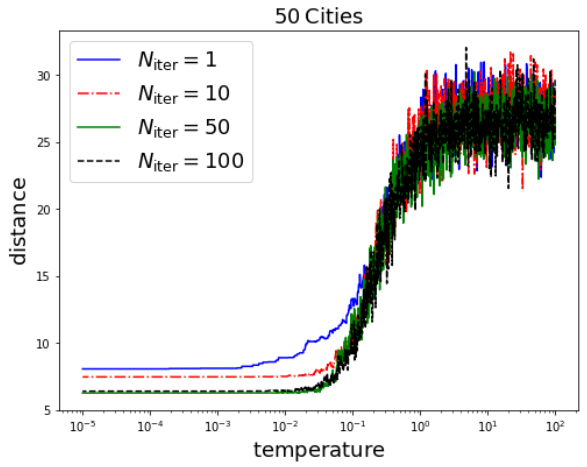

FIG. 4. The path length as a function of an effective temperature for fifty cities plotted for varies number of equilibration steps or iterations.

APPENDIX: CODES

```
#include<iostream>
#include<fstream>
#include<ctime>
#include<cstdlib>
#include<vector>
#include<cmath>
#include<algorithm>
#include<iomanip>

  using namespace std;

const int N = 7; //The number of cities
double random_number(){return (double)rand()/(
    RAND_MAX + 1.0);}


int factorial(int n) {
   if ((n==0)||(n==1))
      return 1;
   else
      return n*factorial(n-1);
}

struct City
{
  double x,y;
  string name;
};

struct Path
{
  double length;
  double index[N];
};

void dist_func(City point1, City point2, double
    &dist){
  dist = sqrt((point1.x-point2.x)*(point1.x-
    point2.x) + (point1.y-point2.y)*(point1.y-
    point2.y));
  }
void swap_cities(Path &path){
  int i = rand()%N;
  int j = rand()%N;
  while(i==j){j = rand()%N;} //Make sure that
    they are not the same
  int temp = path.index[i];
  path.index[i] = path.index[j];
  path.index[j] = temp;
  }

double calc_dist(Path path, vector<City> p){
  double dist;
  double delta = 0.0;
  for(int i = 0; i < N; i++){
    dist_func(p[path.index[i%N]],p[path.index[(i
    +1)%N]],dist);
    delta +=dist;
  }
  return delta;
  }
main(){
  vector<City> p(N);
  srand(time(NULL));
  for(int i = 0; i < N; i++){
    p[i].x = random_number();
    p[i].y = random_number();
  }


  Path path;
  for(int i = 0; i < N; i++){
    path.index[i] = i;
  }

  double cooling = 0.999;
  double T_init = 100;
  double T_final = 0.00001;
  double prob,diff;
  double T = T_init;
  int k = 0;
  Path new_path;
  ofstream outfile("cost.dat");
  while (T > T_final){

    for(int i = 0; i < 50; i++){
      new_path = path;
      swap_cities(new_path);
      diff = calc_dist(new_path,p) - calc_dist(
    path,p);
      prob = exp(-diff/T);
      if (diff<=0){path = new_path;}
      else if (diff>0 && prob>random_number())
      {
        path = new_path;
      }
    }
    T *= cooling;
    k++;
    if (k%10==0){ outfile << T << " " <<
    calc_dist(path,p)<< endl;}
  }

  int m;
  ofstream outfile1("cities_sa.dat");
  for(int i = 0; i < N; i++){
    m =path.index[i];
    outfile1 <<" "<< m <<" "<< p[m].x << "   " <<
     p[m].y << endl;
  }

  int num[N];
```

```
        for(int j =0; j<N; j++){num[j] = j;}
        int q = 0;
        double distance;
        vector<Path> route(factorial(N));
cout<< "The number of paths is "<< factorial(N)<< endl;
        sort(num,num+N);
        do{
                double total_distance = 0;
                for(int j  = 0; j < N; j++){
                                dist_func(p[num[j%N]],p[num[(j+1)%N]],distance);
                                total_distance += distance;
                }
                route[q].length = total_distance;
                for(int i = 0; i < N; i++)
                        route[q].index[i] = num[i];

                q += 1;
        }while(next_permutation(num,num+N));
  int pp = 0;
        double temp = route[0].length;
        for(int k = 1; k < factorial(N); k++){
                if(route[k].length<temp){
      pp = k;
                        temp = route[k].length;
                }
        }

        cout << "The shortest distance is " << setprecision(7) << temp << " k= " << pp << endl;

  ofstream outfile2("cities_bf.dat");
  for(int i = 0; i < N; i++){
    m =route[pp].index[i];
    outfile2 <<" "<< m <<" "<< p[m].x << "  " << p[m].y << endl;
  }

}
```

. /home/omo/advanced_comp_physisc/hw1/ts.cc