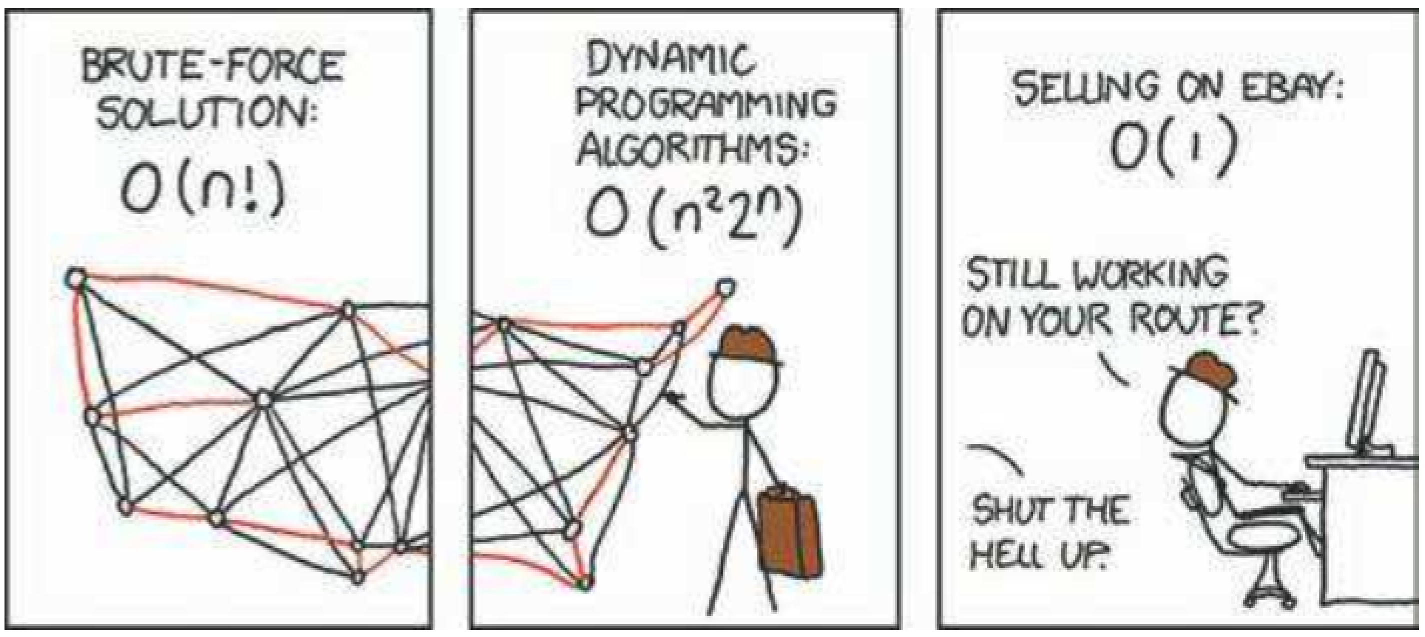
TSP specific heuristics

: Compiled by Anmol Agarwal

Description

The travelling salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"



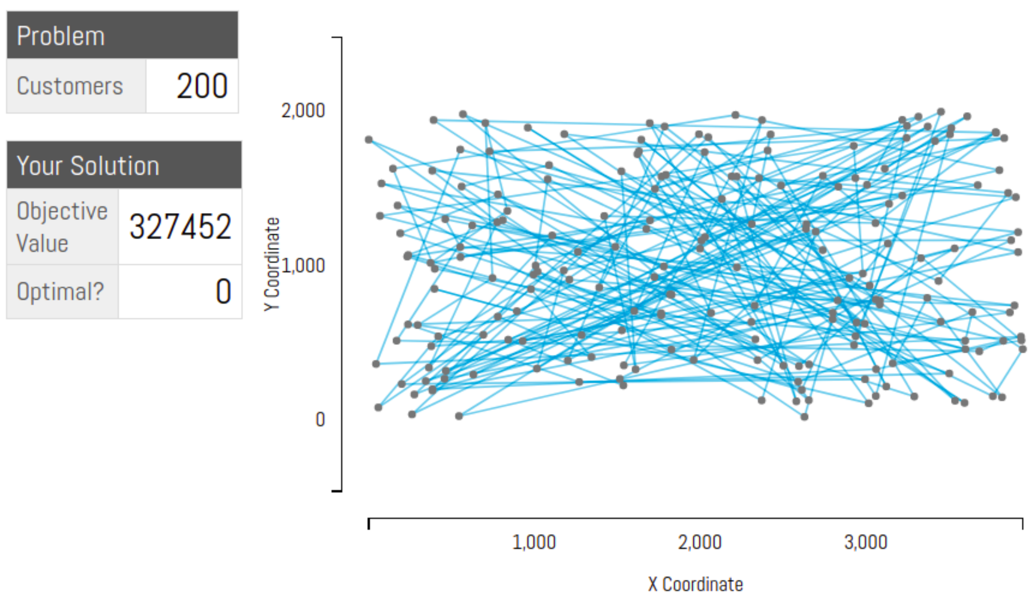
If someone hands us a solution, then we can check easily that it does lead to the value it claims and is it a valid ordering. This property makes the TSP a member of the class known as NP, consisting of all problems for which we can check the correctness of a solution in polynomial time. The pair of letters stands for non-deterministic polynomial. The unusual name aside, this is a natural class of problems: when we make a computational request, we ought to be able to check that the result meets our specifications.

Timeline of our journey while doing the TSP assignment on Coursera

We did this assignment primarily over a span of 2-3 days while experimenting with new heuristics over the course of a week.

Nearest neighbour

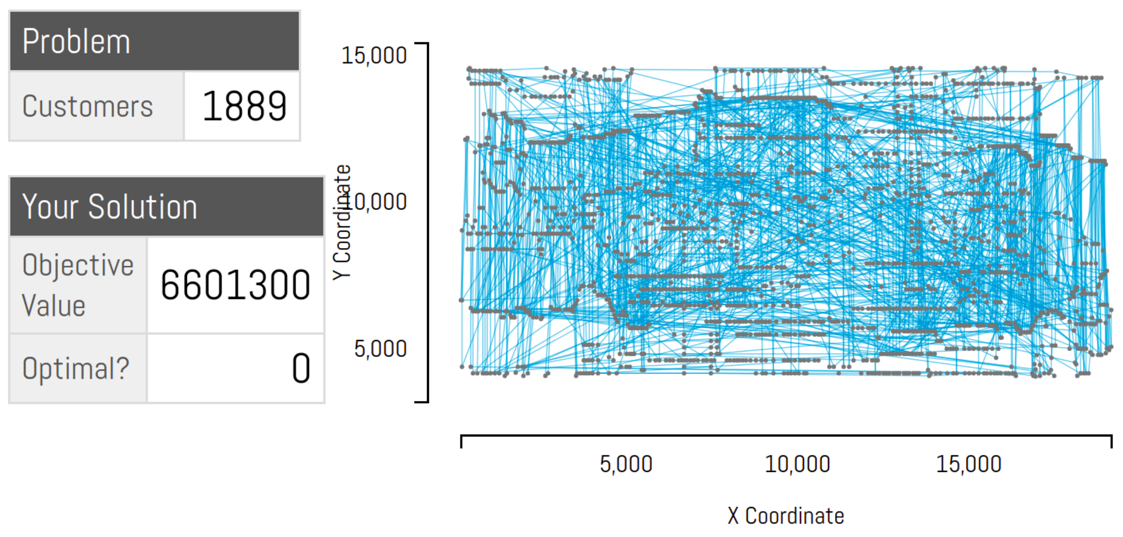
Nearest Neighours (NN) is one of the simplest greedy heuristics which is the first one which pops to mind. Unlike local search heuristics which makes changes on an already formed ordering, this method is constructive in the sense that it starts with an initial city and chooses the next city as the city closest to the current city which hasn't been visited yet. As we can see in the image below, even on one of the smaller test cases, the performance is poor as is evident from the numerous edges crossing each other.



Problem	
Customers	200

Your Solution	
Objective Value	327452
Optimal?	0

Output to nearest Neighbour algorithm on a dataset consisting of 200 cities



Problem	
Customers	1889

Your Solution	
Objective Value	6601300
Optimal?	0

Output to nearest Neighbour algorithm on a dataset consisting of 1889 cities

Using approximation algorithm (2 factor approximation)

Some graph related terms for understanding:

A sequence : $v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n$ where v_i are vertices, e_i are edges, and for all i the edge e_i connects the vertices v_{i-1} and v_i and such a sequence is called a **walk**.

A walk with no repeated edges is called a **tour**.

A walk with no repeated vertices is called a **path.**

A walk or a tour where $v_0 = v_n$ is called **closed.**

A circuit is path that begins and ends at the same vertex.

Euler tour is defined as a way of traversing tree such that each vertex is added to the tour when we visit it (either moving down from parent vertex or returning from child vertex). We start from root and reach back to root after visiting all vertices.

Hamilton Circuit: a circuit that must pass through each vertex (except the first vertex is obviously visited twice) of a graph once and only once.

In other words, Hamiltonian circuit is a circuit that visits every vertex once with no repeats. Being a circuit, it must start and end at the same vertex.

Now that the terminology was clear, the actual approximation can be talked about.

2-factor approximation for Metric TSP:

Input: A **complete graph** $G(V, E)$, where the vertices are the cities

Output desired: A Hamiltonian cycle of minimum cost.

Our graph is K_n and has $n!$ Hamiltonian cycles. This is trivial to verify as there are n choices for the starting vertex, then $(n - 1)$ choices for which vertex to visit next, then $(n - 2)$ choices for which vertex to visit next and so on. Because the graph is complete, there will always be an edge that is capable of taking us to the next projected vertex. After the final vertex, one can easily use the edge that connects back to the starting vertex.

Since G is complete and hence connected, I was able to find a minimum spanning tree for the input graph. I used the **Prim's algorithm** to find the minimum spanning tree as using other standard MST algos like Kruskal would have required me to **store the weights of $\binom{N}{2}$ edges and in cases where N is quite large** $\approx 10^4$, my program would have run out of memory. Also, distance calculation between vertices v_i and v_j was trivially $O(1)$ using distance formula.

Objective is to find the minimum cost Hamiltonian cycle

The key observation is to observe the satisfaction of the **triangle inequality** by the edges. This version of TSP is known as **METRIC TSP**.

So, the following properties are satisfied:

$$d(v_i, v_j) >= 0$$
$$d(v_i, v_j) = d(v_j, v_i)$$
$$d(v_i, v_j) + d(v_j, v_k) >= d(v_i, v_k)$$

The METRIC TSP problem is also **NP-HARD.**

Let the MST obtained be T.

Let the minimum cost Hamiltonian Cycle be H^*. Let e_h be an edge in H^*.

Let T be rooted at some vertex v_0.

Now, we find a Euler tour of T rooted at v_0 by making a DFS traversal and storing the vertex ordering in a list L. We see that each edge in T is repeated twice in L and so the following holds:

$$c(L) = 2 * c(T) \tag{1}$$

Now, we traverse through the list L and retain only the first occurrence of each vertex in this list, and also retain the last vertex in the list, which is v_0. **This list can be interpreted as a cycle C in G, because each vertex occurs exactly once in the list except for the first vertex, which also occurs in the end.**

Now, if L contains the sub-sequence v_i, v_r, v_j where v_i, v_j are being visited for the first time whereas v_r is a repeated vertex, then C contains the sub-sequence v_i, v_j. So, now by triangle inequality, the following holds:

$$d(v_i, v_r) + d(v_r, v_j) >= d(v_i, v_j)$$

and so we can conclude that due to all such sub-sequences of type v_i, v_r, v_j from L and insert v_i, v_j in C instead, we can conclude that :

$$\begin{gathered} c(C) <= c(L) \\ \implies c(C) <= 2 * c(T) \end{gathered} \tag{2}$$

Also, from the minimum cost Hamiltonian cycle H^*, if we remove an edge e_h, then $[H^* - e_h]$ is also a spanning tree of graph G and so,

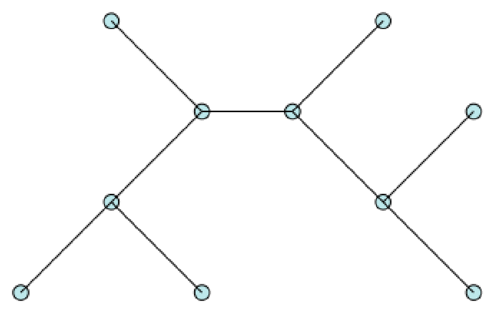
$$c(T) <= c([H^* - e_h]) <= c(H^*) \tag{3}$$

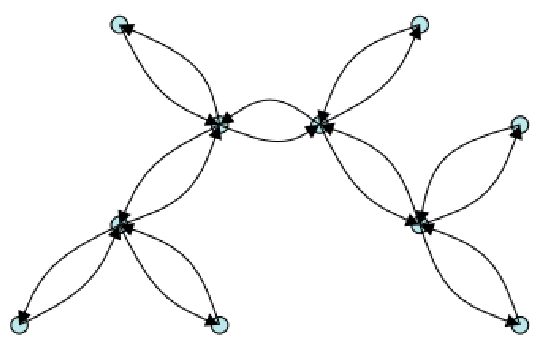
Using (1),(2) and (3), we get the following:

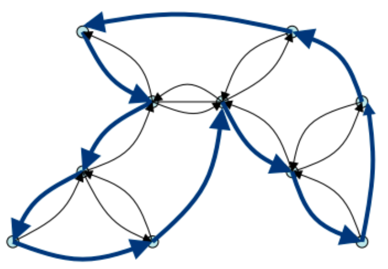
$$c(C) <= 2 * c(H^*)$$

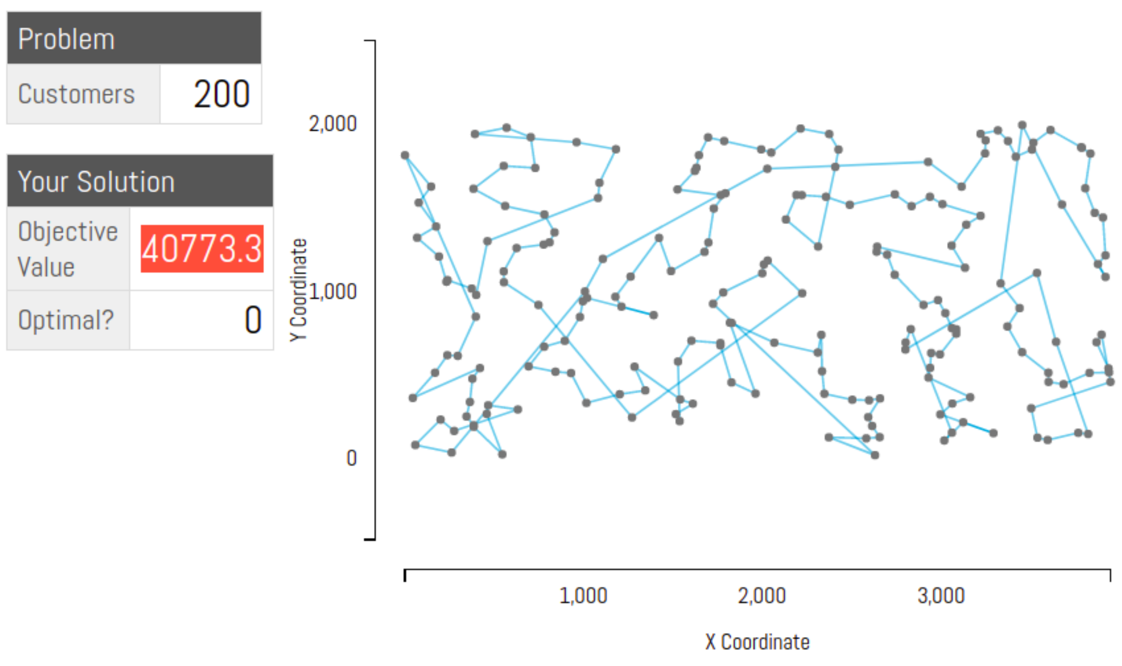
Hence, cycle C is a factor 2 approximation of H^*.

For a visual understanding, the following example can be used:


MST T of a complete graph


Euler Tour L of the graph


Constructed graph C of Euler Tour L



Problem	
Customers	200

Your Solution	
Objective Value	40773.3
Optimal?	0

The improvement of 2-factor approximation over NN (reduced cross edge)

2 opt with approx algorithm

In above applied strategies, one can clearly see in the visualization of the solutions that there are a lot of edges crossing each other. Now, since we were tackling metric TSP, it was quite intuitive that if 2 crossing edges could be removed and a 2 new edges be introduced instead, then as Professor Kanan would say **"we can do better than the current solution".**

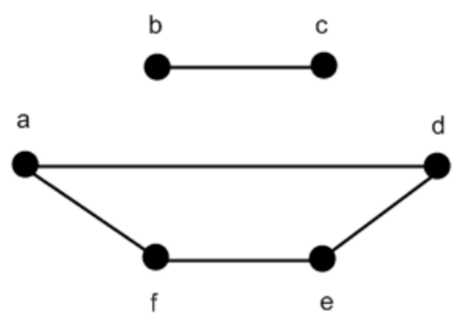
So, 2-opt becomes a great candidate for a local move.

The main idea behind it is to take a route that crosses over itself and reorder it so that it does not.This move deletes two edges, thus breaking the tour into two paths, and then reconnects those paths in the other possible way.

In the below figure, we see edges $e_1 = (b, d)$ and $e_2 = (a, c)$ are crossing each other. It is easy to see that if we remove edges e_1 and e_2 while keeping all remaining edges the same, there are $(\frac{4!}{(2!)(2!)(2!)}) = 3$ ways to connect the quadruple of vertices (a, b, c, d). However, all of them are not legal and progressive at the same time.

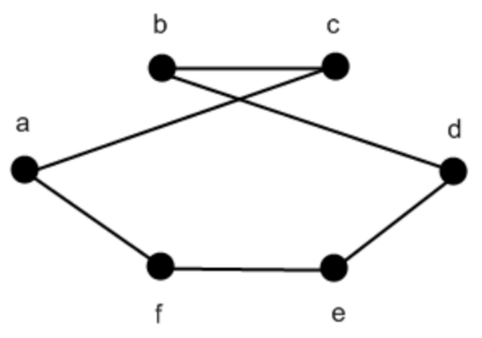
Alternative 1: Introduce edges (b, c) and (a, d)

Clearly, such a move is illegal as the resulting graph is no longer connected.



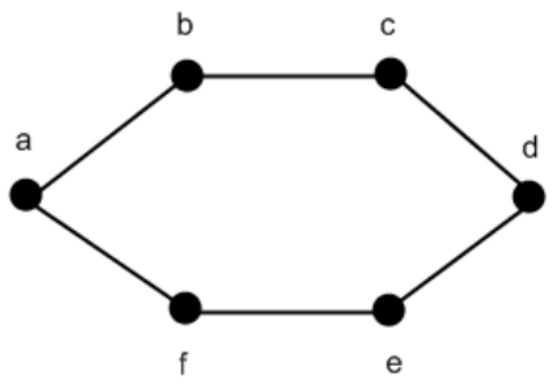
Alternative 2: Introduce edges (a, c) and (b, d)

This is redundant as the original configuration is obtained and hence, we essentially made no progress.



Alternative 3: Introduce edges (b,c) and (a,d)

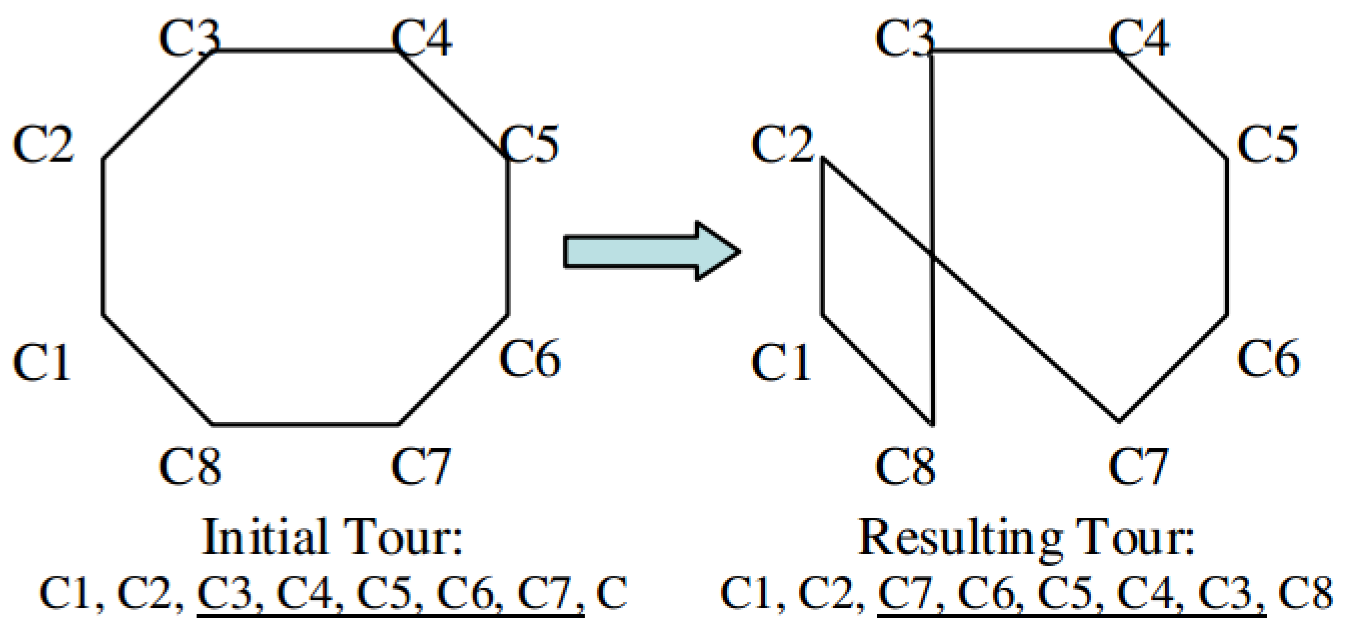
This move seems to remove the 2 edges crossing each other and to introduce 2 new valid edges.



Thus, among all the 3 possible moves, only the 3rd alternative is both progressive and legal.

Implementation wise, I iterate over the initial solutions generated by using the MST 2 factor approximation. For each initial solution, I keep applying 2-opt as a local move as long as there is at least one possible 2-opt move which **leads to an improvement over the current configuration**. Otherwise, the program gets stuck in a local minima and the program starts the next iteration with a new starting initial solution. **If there are several possible 2-opt moves, one can either choose the one which gives the best improvement or the one which is discovered first.**

On observing more closely, a 2-opt move is equivalent to reversing the order of the cities between the two edges and hence, in order to maintain the cyclicity of the tour of directed edges, a segment reversal needs to be performed.



For a general cost function, in order to check whether a tour is 2-opt optimal we check $\binom{V}{2}$ie $O(|V|^2)$ pairs of edges. For each pair, the work required to see if the switch decreases the tour cost can he performed in constant time. Thus, the amount of time required to check a tour for 2-optimality is $O(|V|^2)$.

For implementation specific details, let us assume a fixed orientation of the tour, with each tour edge having a unique representation as a pair (a, b), where a is the immediate predecessor of b in tour sequence. Then each possible 2-Opt move can be viewed as corresponding to a 4-tuple of cities $< t_1, t_2, t_3, t_4 >$, where (t_1, t_2) and (t_4, t_3) are the oriented tour edges deleted and (t_2, t_3) and (t_1, t_4) are the edges that replace them.

SO, if ordering is $(t_1, t_2, t_8, t_9, t_{10}, t_{11}, t_4, t_3)$;

Then new ordering involve reversing sequence starting from t_2 and ending at t_4. So, new ordering will be $(t_1, t_4, t_{11}, t_{10}, t_9, t_8, t_2, t_3)$.

Also, he move can be evaluated as follows:

Original cost of joining the tuple $< t_1, t_2, t_4, t_3 >$ **is** $dis(t_1, t_2) + dis(t_3, t_4)$.

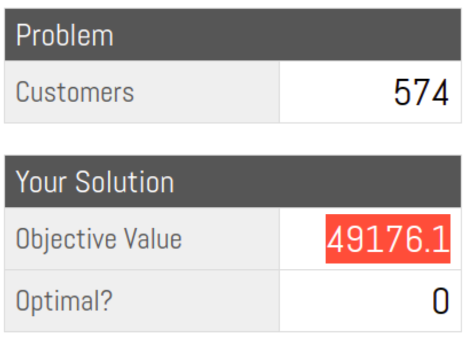
Suspected **cost after 2-opt move for the tuple** $< t_1, t_2, t_4, t_3 >$ **is** $dis(t_1, t_4) + dis(t_2, t_3)$.

So, a 2-opt move will lead to an improvement only if :

$$dis(t_1, t_4) + dis(t_2, t_3) < dis(t_1, t_2) + dis(t_3, t_4)$$

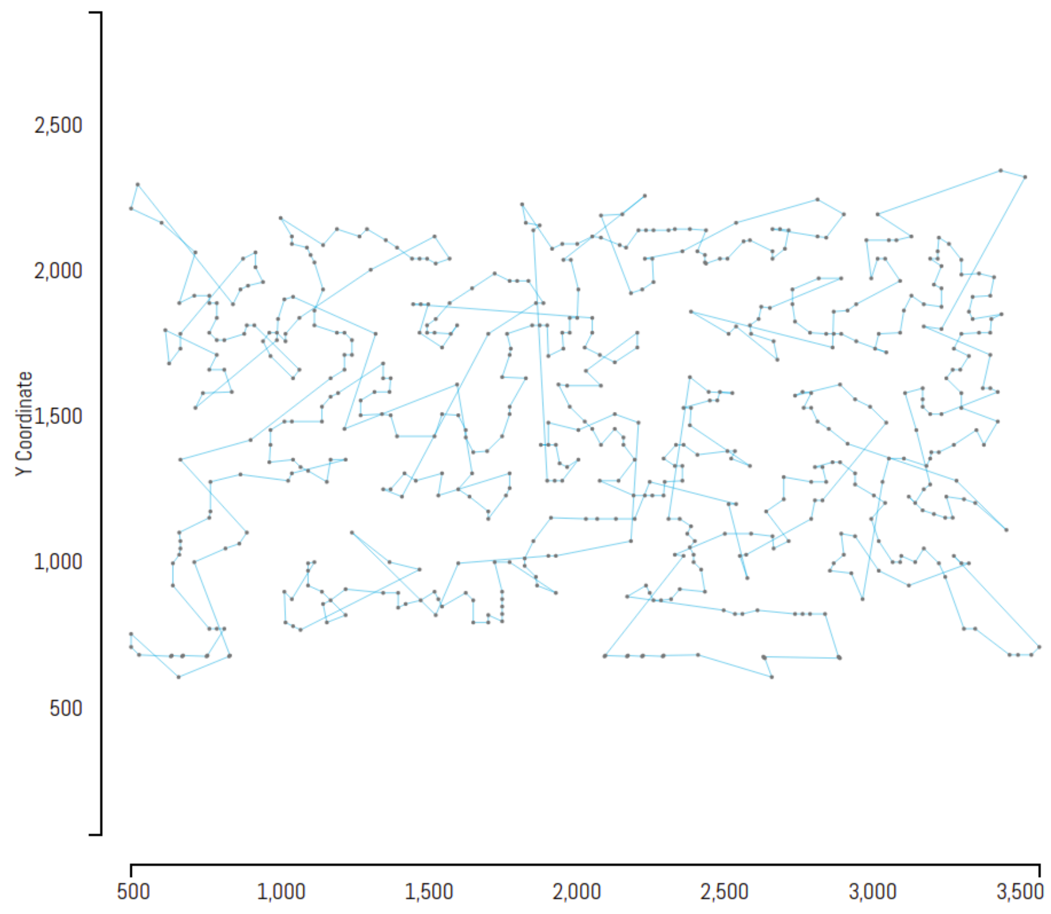
```
bool try_2_opt()
{
    int i, j, t1, t2, t3, t4;
    int pos1 = -1, pos2 = -1;
    double now, poss;
    double diff = -1;
    double max_diff = -1;
    for (t2 = 0; t2 < n; t2++)
    {
        for (t4 = t2 + 1; t4 < n - (t2 == 0); t4++)
        {
            t1 = ((t2 - 1) + n) % n;
            t3 = (t4 + 1) % n;
            now = dist(v[t1], v[t2]) + dist(v[t3], v[t4]);
            poss = dist(v[t1], v[t4]) + dist(v[t2], v[t3]);
            diff = now - poss;
            if (diff > 0)
            {
                if (diff > max_diff)
                {
                    max_diff = diff;
                    pos1 = t2;
                    pos2 = t4;
                }
            }
        }
    }

    if (pos1 == -1 || pos2 == -1)
    {
        return false;
    }
    else
    {
        // debug(pos1);
        // debug(pos2);
        tot -= max_diff;
        modify_vector(pos1, pos2);
        return true;
    }
}
```



Problem	
Customers	574

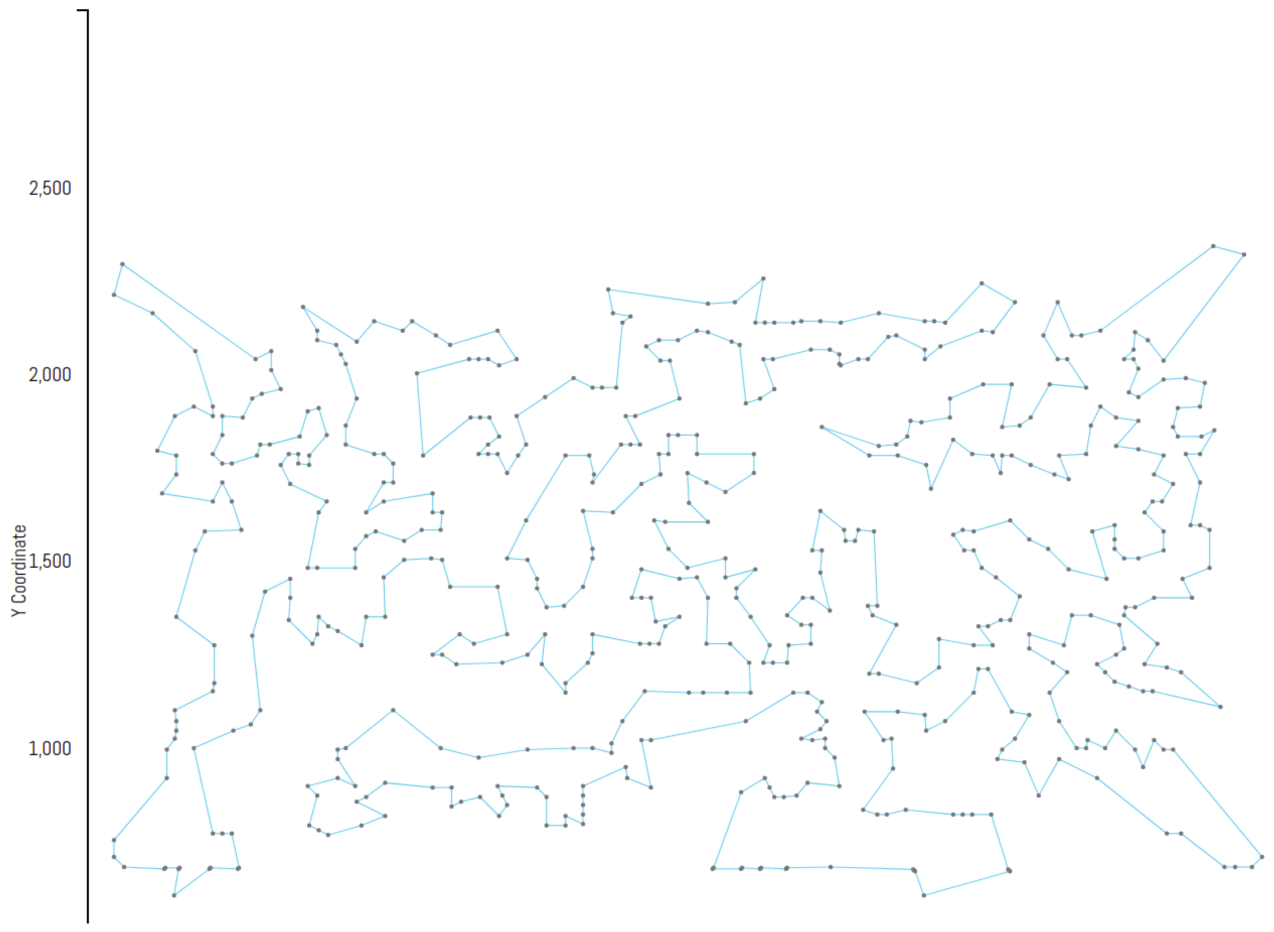
Your Solution	
Objective Value	49176.1
Optimal?	0



Without 2opt heuristic, observe the large number of edges crossing each other

Problem	
Customers	574

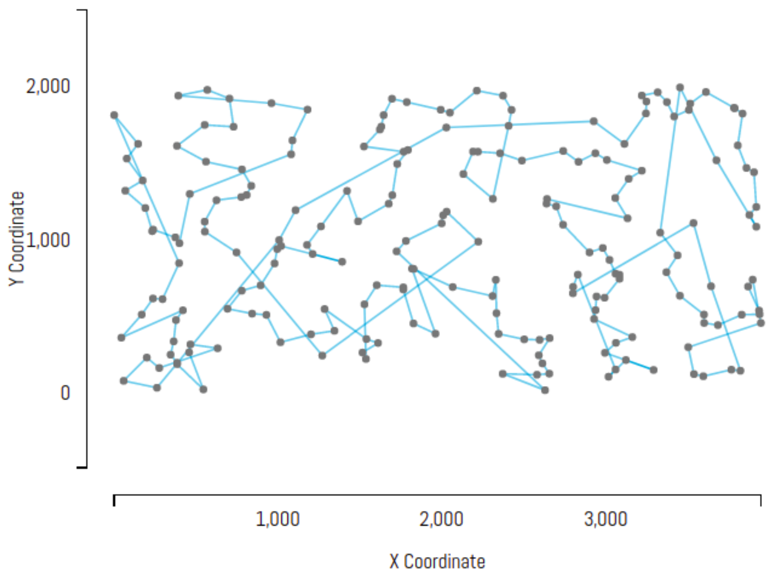
Your Solution	
Objective Value	38695.7
Optimal?	0



2-opt succeeds in removing all edges crossing each other

Problem	
Customers	200

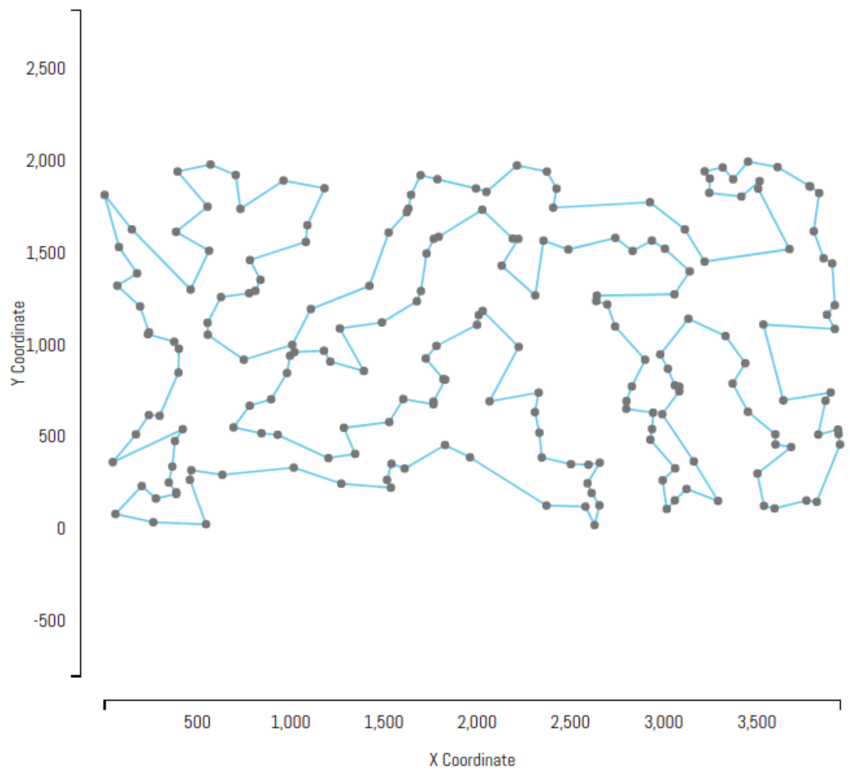
Your Solution	
Objective Value	40773.3
Optimal?	0



Dataset of 200 vertices without 2-opt

Problem	
Customers	200

Your Solution	
Objective Value	32173
Optimal?	0



Dataset of 200 vertices with 2-opt

Simulated annealing (along with setting of parameters)

Borrowing from Physics and Biology

The key feature of simulated annealing is that it provides a means to escape local optima by allowing hill-climbing moves (i.e., moves which worsen the objective function value) in hopes of finding a global optimum.

Origin of this strategy

Simulated annealing is inspired from the annealing process in metallurgy. Generally, when a substance goes through the process of annealing, it is first heated until it reaches its fusion point to liquefy it, and then slowly cooled down in a control manner until it solids back. The final properties of this substance depend strongly on the cooling schedule applied; if it cools down quickly the resulting substance will be brittle and will be easily broken due to an imperfect structure, if it cools down slowly the resulting structure will be well-organised and strong.

So, here the metal can be thought to be synonymous to the solution ordering of the problem. Now, if the temperature is reduced quickly, the algorithm won't get enough chances to explore different sections of the solution space and as a result the quality of the solution would be bad. On the other hand, if temperature is decreased patiently, the algorithm will get multiple chances to explore potentially better sections of the solution space. So, the trade-off here is between the time taken and the different sections of the solution state explored. Chances are that more the number of sections explored, better would be the quality of the solution obtained. This trade-off can be monitored by tinkering with the parameters of annealing : mainly the initial temperature and the temperature reduction schedule.

When solving an optimisation problem using simulated annealing the structure of the substance represents a codified solution of the problem, and the temperature is used to determined how and when new solutions are perturbed and accepted. The algorithm is basically a three steps process: make minor changes to the current solution, evaluate the quality of the solution, and accept the solution if it is better than the new one.

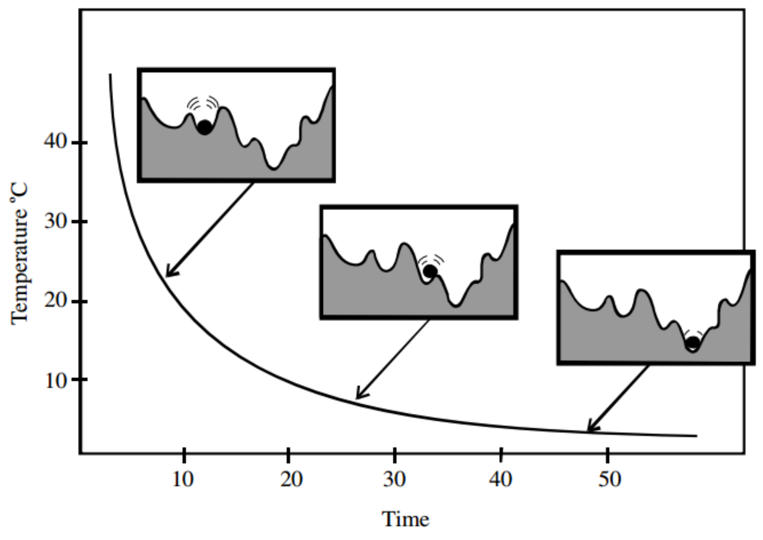
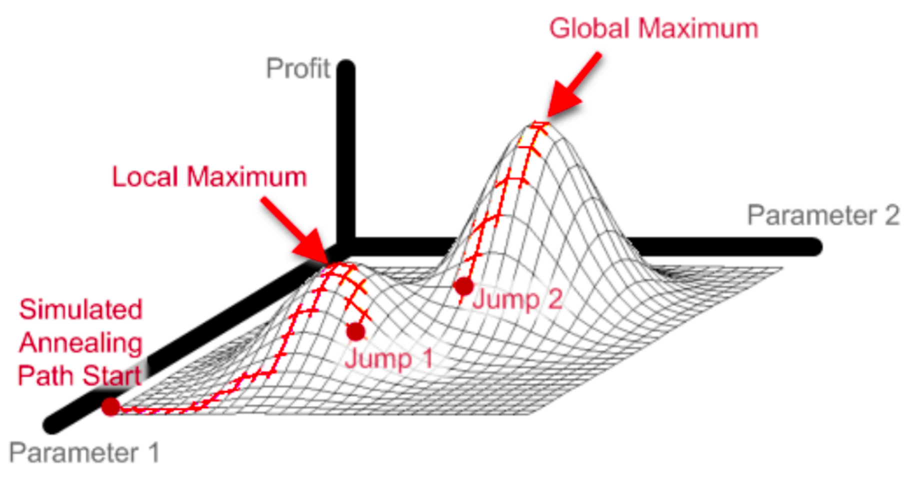


Image taken from paper ' Practical considerations of Simulated Annealing' by Ledesma and Sanchez. Attempt is made to minimize the objective value

The method of simulated annealing can be easily understood by observing the above figure which shows a hermetic box with an internal uneven surface (with peaks and valleys), and a ball resting on this surface. **The objective is to move the ball to a position as close as possible to the bottom of the box.** At the beginning of the process the temperature is high and strong perturbations are applied to the box allowing the ball to easily jump over high peaks in search of the bottom of the box. Because of the high energy applied, it is possible for the ball to go down or up easily. As time goes by, the temperature decreases and the **ball has less energy to jump high peaks**. When the temperature has decreased to the point when the ball is able to jump only very small peaks, the ball should be hopefully very close to the bottom of the box and the process is completed.



In the case where optimisation function is to be maximized, attempt is made to escape the local minima by making chaotic jumps every now and then

Annealing schedule.

Determining the initial value of T requires experimentation. I first tried to determine the range of values of ΔE that will be encountered from move to move. I observed the values of $\delta(cost)$ for the first few local moves and set the value of T larger than the largest among them. **I hold each new value of T constant for, say, 1000N reconfigurations, or for 100N successful reconfigurations, whichever comes first.**

There are two possible typical cooling schedules in the literature: exponential and linear.

In a typical exponential cooling, the process spends little time at high temperatures, and as the temperature decreases more and more time is spend at each temperature, allowing the algorithm to refine very well the solution found at high temperatures.

On linear cooling, the temperature decreases linearly as the time increases, thus, the algorithm spends the same amount of time at each temperature.

I have used an exponential decrease in temperature with the new_temp = (TFACTR) * current temperature.

After much experimentation, I set TFACTR as 0.9.

How exactly does T govern the acceptance of a particular move ?

If S is the original candidate solution and R its newly tweaked child.

If R is better than S, we'll always replace S with R as usual.

But if R is worse than S, we may still replace S with R with a certain probability $P(T, R, S)$:

$$P(T, R, S) = e^{-\frac{(cost(R) - cost(S)}{T}} \quad where \ T \geq 0.$$

This equation is interesting in the following ways.

- Note that the fraction is negative because R is worse than S. First, if R is much worse than S, the fraction is larger, and so the probability is close to 0. If R is very close to S, the probability is close to 1. Thus if R isn't much worse than S, we'll still select R with a reasonable probability.
- If current prospected neighbour is worse and T is close to 0, the fraction is again a large number, and so the probability for a degrading move is close to 0. So, the probability of making a degrading move towards the end of the journey is practically null.
- If current prospected neighbour is worse and T is high, the probability of making a degrading move is close to 1.

📌 In cases specific to the TSP assignment on Coursera, I found the delta(cost) to be quite closely related to the order of the magnitude of the coordinates. Eg: T will be higher if the coordinates are ~ 10^6 but T is considerably lower when coordinates are ~ 10^2.

```
void perform_annealing()
{
    int ans, nover, num_limit;
    int i, j, k, num_success, nn, idec;
    static int n[7];
    long i_dummy;
    double path, delta_e, t;
    int ncity = num_cities;

    ////////////////////////////////////////////////////////////
    nover = 1000 * ncity; //    Maximum number of paths tried at any temperature.
    num_limit = 10 * ncity;  //    Maximum number of successful path changes before continuing.
    t = 10000;            //initial temperature

    //setting initial length as input path length
    path = tot;

    if (ncity < 100)
    {
        nover = 100 * ncity; //    Maximum number of paths tried at any temperature.
        num_limit = 10 * ncity; //    Maximum number of successful path changes before continuing.
        t = 30;
    }
    else if (ncity > 10000)
    {
        // nover = 1 * ncity; //    Maximum number of paths tried at any temperature.
        // num_limit = 1 * ncity; //    Maximum number of successful path changes before continuing.
        // t = 10;
        return;
    }

    ///////////////////////////////////////////////////////////////////////
    i_dummy = -1;

    //  Try up to 100 temperature steps.
    for (j = 1; j <= 100; j++)
    {
        num_success = 0;
        //nover=100*city_num
        for (k = 1; k <= nover; k++)
        {
            do
            {
                //random choosing of vertices for 2-opt
                n[1] = (int)(ncity * ran3(&i_dummy));       //   Choose beginning of segment
                n[2] = (int)((ncity - 1) * ran3(&i_dummy)); //..and end of segment...
                if (n[2] >= n[1])
                {
                    ++n[2];
                }
                nn = 1 + ((n[1] - n[2] + ncity - 1) % ncity); //   nn is the number of cities  not on the segment;
            } while (nn < 3);

            //cost if we try to make this 2-opt move (essentially a segment reversal)
            delta_e = get_cost(n[1], n[2]);

            // Consult the oracle regarding this move
            ans = oracle_opinion(delta_e, t);

            if (ans)
            {
                ++num_success;
                path += delta_e;
                tot += delta_e;
                best_cost = min(best_cost, tot);

                //Carry out the reversal.
                modify_vector(min(n[1], n[2]), max(n[1], n[2]));
            }

            if (num_success >= num_limit)
            {
                break;
            }
            //Finish early if we have enough successful changes at this temperature
        }
        // printf("\n %s %10.6f %s %12.6f \n", "T =", t, " Path Length =", path);
        // printf("Successful Moves: %6d\n", num_success);
        //Annealing schedule
        t *= TFACTR;
        // If no success, we are done.
        if (num_success == 0)
        {
            return;
        }
    }
}

int oracle_opinion(double delta_e, double t)
{
    static long tmp_rand = 1;
    // debug(delta_e);
    if (delta_e < 0)
    {
        neg++;
    }
    else
    {
        pos++;
    }
    return delta_e < 0.0 || ran3(&tmp_rand) < exp(-delta_e / t);
}
```

Guided fast local search

A thorough mention of this can already be found in the LS NOTES. Check them out for details regarding alpha variation.

```
void penalize_features(double g_val)
{
    int src, dest;
    int max_pos = -1;
    double curr_max = -1;
    for (int i = 0; i < n; i++)
    {
        src = v[i];
        dest = v[(i + 1) % n];
        util[i] = (dist(src, dest)) / (1 + penalty[src][dest]);
        if (util[i] > curr_max)
        {
            max_pos = i;
            curr_max = util[i];
        }
    }

    if (max_pos == -1)
    {
        //cout << "WEIRD STUFF\n";
        exit(0);
    }
    src = v[max_pos];
    dest = v[(max_pos + 1) % n];
    penalty[src][dest]++;
    penalty[dest][src]++;
    lambda = alpha * (local_val) / n_features;
    // debug(lambda);
    // debug(tot);
}
```