The TSP algorithm we implemented is a simulated annealing algorithm adjusted to work well with the TSP problem. Pseudocode is presented on the last page. Most of the ideas presented here are taken from the Wikipedia article on simulated annealing. We also included an optimization based on the MST concept mentioned in class on April 19.

The basic idea behind simulated annealing is the following:

1. The *current state* considered starts as a version of Prim's algorithm modified so that the head of the current path will add the vertex closest to it and also makes sure to add the final edge from the end of the path back to the beginning.
2. Get a potential *next state* based on the current state
3. Calculate the current *temperature* based on time elapsed so far and the total time limit
4. Calculate the probability that we will move to the next state based on the two states and current temperature
5. Move to the new state with the above probability
6. Repeat steps 2-5 until time is up, keeping track of the best path encountered

Steps 2-4 are implemented in the **get_neighbor**, **get_temperature**, and **get_chance** functions in the pseudocode.

The **get_neighbor** function takes in a current state and generates a new state by doing one of two things:

1. Swap two random consecutive elements of the current state
2. Take a random block of continuous nodes in the current state and reverse them

Option 1 is chosen with some constant probability p. The idea behind this function is to create a new state that has cost close to that of the current state. In either option, only 2 edges change. The reason why there are two different options is to minimize the chance that the algorithm will get stuck in a local minima state.

The **get_temperature** function takes in the amount of time elapsed t_{elapsed} and the total time limit t and returns a temperature between 0 and 1. The formula I used was linear in terms of t_{elapsed}:

$$\text{current temperature} = 1 - \frac{t_{\text{elapsed}}}{t}$$

As t_{elapsed} increases, the temperature decreases. More on this in the **get_chance** description.

The **get_chance** function takes in the current state, potential next state, and temperature, and returns the probability that it will move to the next state. The formula used is

$$\text{chance} = \begin{cases} 1 & \text{if } \Delta\text{cost} < 0 \\ x = \exp\left(-c \cdot \frac{\Delta\text{cost}}{\text{avg of all } \Delta\text{cost values calculated so far}} \cdot \frac{1}{\text{temperature}}\right) & \text{otherwise} \end{cases}$$

where Δcost = (cost of potential new state) − (cost of current state) and c is some constant.

If the new state has lower cost, then the probability is 1. Otherwise, the probability is x where x decreases as temperature decreases and increases as Δcost decreases.

The gritty implementation details and optimizations removed from the pseudocode for clarity. Note that there are several hardcoded constants In particular, $p = 0.37$ and $c = 8$ in the final program. These were chosen by running the program repeatedly on tests from `http://www.math.uwaterloo.ca/tsp/data/index.html` and choosing the values that performed well.

```
simulated_annealing(graph G, time limit t):
    Input: a complete graph G
    Input: the time limit t
    Output: A cycle and its cost
    best_state , best_cost ← null, ∞
    curr_state ← get_starting_tour(nodes)
    while time not up:
        if curr_state costs less than best_state :
            best_state ← curr_state
        new_state ← get_neighbor(curr_state)
        temperature ← get_temperature(time elapsed, t)
        with probability get_chance(curr_state, new_state, temperature):
            curr_state ← next_state
    return best_state and its cost
```

Helper methods:

```
get_starting_tour(nodes):
    tour ← nodes[0]
    node_remaining = {nodes − nodes[0]}
    while nodes_remaining:
        min_distance = ∞
        closest_neighbor = 0
        for neighbor in nodes_remaining:
            if neighbor.distance < min_distance:
                min_distance ← neighbor.distance
                closest_neighbor ← neighbor
        tour ← closest_neighbor
        node_remaining.remove(closest_neighbor)
```

return tour

get_neighbor(curr_state):
with probability 0.37:
flip a contiguous chunk of the curr_state
otherwise:
swap 2 consecutive elements in the curr_state
return the resulting state

get_temperature(time elapsed):
return $1 - (\text{current time elapsed})/t$

get_chance(curr_state, new_state, temperature):
$\Delta\text{cost} \leftarrow$ cost of new_state − cost of curr_state
if $\Delta\text{cost} < 0$:
return 1
else:
return $\exp\left(-8 \cdot \frac{\Delta\text{cost}}{\text{avg of all } \Delta\text{cost values calculated so far}} \cdot \frac{1}{\text{temperature}}\right)$