Elevator Python Project Report; Pae Swanson; PHZ3152

In order to make an effective algorithm that incorporated the Monte Carlo method, I created a passenger class initialized with the floor they start on and the floor they end on. This runs under the assumption that the final floors are known by the elevator *before* the passenger gets on; obviously, that is not how elevators work, so this algorithm functions more as a representation of the travelling salesman problem. Now that I've typed that out, I'm wondering if I should have treated the problem like an actual elevator, but it's probably fine. The other two variables for the passenger class are the indices of the overall order of the elevator's path; this way, I could set up a check to ensure that the initial floor of any given passenger comes before their destination, since you can't drop someone off before you pick them up. The algorithm doesn't specifically account for passengers on the same floors or going to/from the same floors, aside from making sure that the initial floor does not equal the destination of a given passenger.

```
class Pass:     # Passenger object because I couldn't think of a better way to keep the 
floors in order lol
    def __init__(self, i, f, ii, fi):
        self.i  = i      # initial floor
        self.f  = f      # final floor
        self.ii = ii     # initial floor global index
        self.fi = fi     # final floor global index
```

Then the code randomly selects one of the N passengers as well as either the initial floor or the final floor to move in the "global" order. It then runs through a check (given which swap is occurring), and makes sure that the new index would not result in an incorrect order; i.e., that the initial floor of a passenger comes before their final floor in the global order after the swap. There might have been an easier way to do this, but I think I've become too accustomed to C++ and object-oriented programming, so I'm lost if something doesn't have a struct or a class. If the proposed change passes the sanity check, then the values at the randomized indices are temporarily swapped. The passengers' global indices are also temporarily updated so that the order can be internally maintained if that passenger is selected again.

The overall distance to be travelled by the elevator is calculated simply by taking the global order and adding every distance between n and n + 1, up to the size of the order (which is two times the number of passengers). Without annealing, if the total distance for a given *proposed* order is smaller than the value for the *current* order, the current order is replaced. This is the same with annealing, though the proposed order will become the new order if the difference is within a certain threshold, even if it is less efficient. In the event that the proposed order is rejected, the temporary values are dropped and revert to their original values. Otherwise, the new order and list of passengers are what is returned. This continues until either a. the while loop has executed a chosen number of times or b. the total distance of the current order has been reduced to or beyond a satisfactory chosen value.

Assuming the sanity tests are passed, the three algorithm options are:

- Completely random (accept change no matter what):

```
passengers[indeces[0]] = pass1
passengers[indeces[1]] = pass2
```

- Without annealing (only accept change if new order is more efficient):

```
if dtemp < dtot:
        passengers[indeces[0]] = pass1
        passengers[indeces[1]] = pass2
```

- With annealing (accept if more efficient OR if the increased distance is within some threshold):

```
if dtemp < dtot or abs(dtemp - dtot) <= ann_threshold:
        passengers[indeces[0]] = pass1
        passengers[indeces[1]] = pass2
```

  - Annealing constant:

```
ann_threshold = C  # Initialized before any computations, C is some number
```

  - Annealing variable:

```
ann_threshold = P * dtot
# Initialized after loop has started, P is some constant percentage and dtot is
the most recently calculated accepted order; the threshold should get smaller as
dtot approaches the solution.
```

I experimented with annealing threshold values and found that:

1) A constant annealing threshold results in end distances that are overall relatively close to each other.
2) A variable threshold (calculated as some percentage of the current distance) has various results, but still tends to be less effective than non-annealing.
3) For both constant and variable:
    a. If the constant/factor is large enough, it behaves like the random algorithm. For the constant threshold, as the constant approaches infinity; for variable, as the factor approaches 1.
    b. If the constant/factor is small enough, it behaves more like the non-annealing algorithm. For both constant and variable threshold, as the constant/factor approaches 0.
4) It can be a coin toss to see if the annealing algorithm comes up with something better than the non-annealing; it's a bit like having a risky choice with a high reward and a safe choice without. Though on average the non-annealing method has better results, annealing can produce a more efficient result than any of the other's solutions, even if it is on *average* less efficient. Introducing the minimum tracker can account for this if the most efficient solution is desired, so one can run multiple randomized trials with different algorithms.

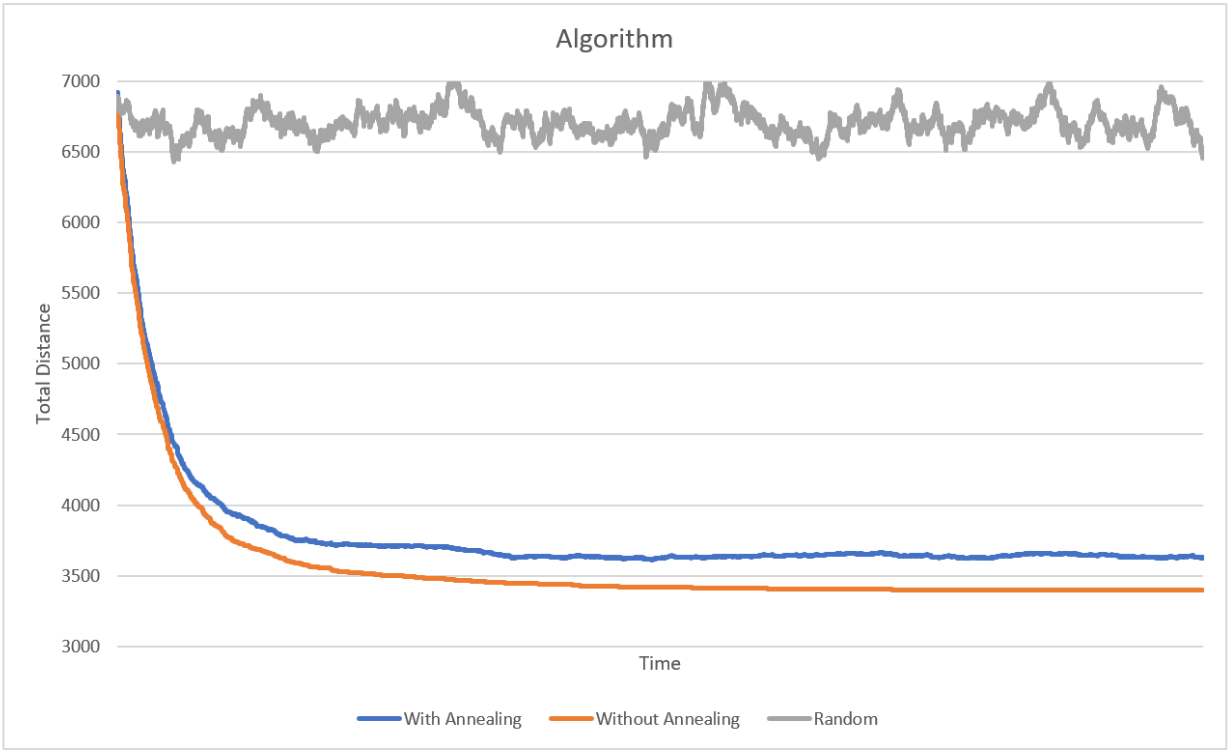

Figure: With the random method that accepts any change to the global order, the variation is about less than 9%, and it comes nowhere near the optimized solution. Without annealing, a favorable solution is achieved with convergence to a minimum distance. With annealing, a favorable solution is achieved with overall convergence to a minimum distance, but this is not as effective of an answer than non-annealing. It also varies greatly depending on the factor and type of annealing used, though it could possibly provide a better solution if the order resulting in the minimum distance is saved.