Originality Statement: I confirm that all work submitted as part of this document is my own. I used sources such as GeeksForGeeks, Python documentation, and ChatGPT for things like zip(), sort(), plotting and other Python nitpicky things

Algorithms Used

Simulated Annealing

Start with a random tour and initial temp of 1

Start loop

Swap two cities randomly

Compute the cost of the old tour and the new tour

If new tour is better, accept it

Else accept new tour with probability based on exponential decay function which I defined as

```
def exp_decay(time, init_temp=100, min_temp=0.001, exp_const=0.005):

    Return max(init_temp * math.exp(-exp_const * time), min_temp)
```

Repeat for 1000 iterations

Evolutionary Algorithm

Generate k initial tours, where k is 20

Start Loop

Generate k successor tours with mutation (defined as swapping two random cities)

Pick the k-best tours from the initial and successor tours by least cost is better

The k-best are now the initial tours

Repeat for 1000 iterations

Population-based Search

Generate k initial tours, where k is 20

Start Loop

Generate k successor tours with mutation (defined as swapping two random cities)

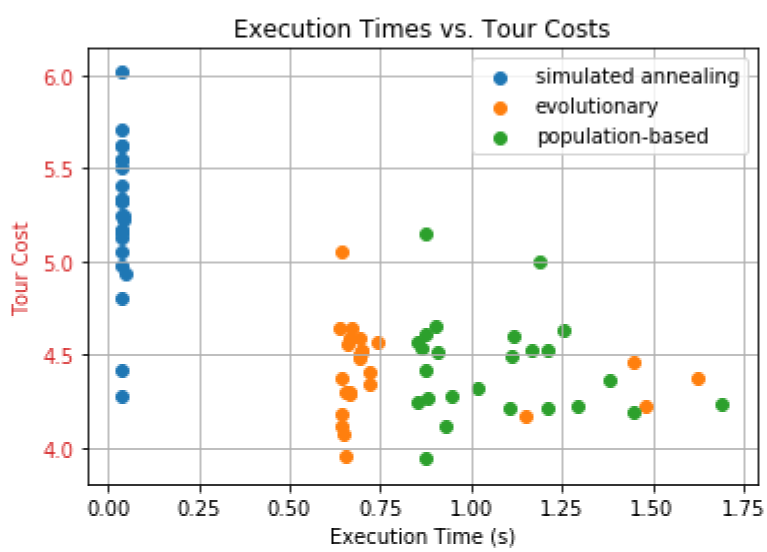
Pick the k-best (beam width) tours from the initial and successor tours by least cost is better

The k-best are now the initial tours

Repeat for 1000 iterations

Experimental Methodology

Using time.time, I recorded the amount of time it took to complete each algorithm for 1000 iterations and recorded the best cost it came up with. I repeated this for 25 runs and plotted the results as below. All parameters used are listed in the algorithm description



Discussion questions:

How many “solutions” did your algorithms generate during their searches?
Simulated annealing only generates one new solution each iteration so in total 1000.

My implementation of evolutionary algorithm started with 20, and generated 20 new solutions each iteration which gives a total of 20 + 20*1000 so 20020.

My implementation of beam search started with 20 and generated 20 new solutions each iteration so also a total of 20020.

How many solutions are there for the TSP with 25 cities?
25! ≈ 15,511,210,043,330,985,984,000
15 quintillion if we want to put it into words

What percentage of all solutions did your algorithms search through?
Simulated annealing: 6.446950284384474e-23
Evolutionary+Beam: 1.2906794469337715e-21
Safe to say, a very *miniscule* amount.

What are some of the benefits and difficulties of each of your search algorithms?
Simulated annealing is always faster than the other algorithms but since it just picks the better solution of two (mostly), it can fall into some bad solutions and it’s not that consistent.

The evolutionary algorithm and beam search for population-based search are virtually identical in the way that I implemented them, but based on the specific perturbation or mutation done, this would affect it. If a state were defined as a robot location on its path to a goal, evolutionary would converge slower, but it may find a better solution.

Why do these algorithms not find an optimal solution every time?
In early iterations it’s only looked or looking at a few solutions, so an optimal solution would be found by random luck. In simulated annealing, it only ever compares two solutions so it might throw away a solution that might have been made better with one change and pick one that seemed better at the time. This can lead it to bad solutions early on, but this is fixed with more iterations. Likewise the other two algorithms look at multiple solutions so they may find an optimal path faster. There are fun cost x iteration graphs in the code attached that show this in more detail.

In [195…

```
import math
import copy
import random
import time

import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform
```

Import file and compute distance matrix

In [196…

```
city_points = np.loadtxt(open("hw2.csv", "rb"), delimiter=",")
dist_matrix = squareform(pdist(city_points, 'euclidean'))
```

Helpers

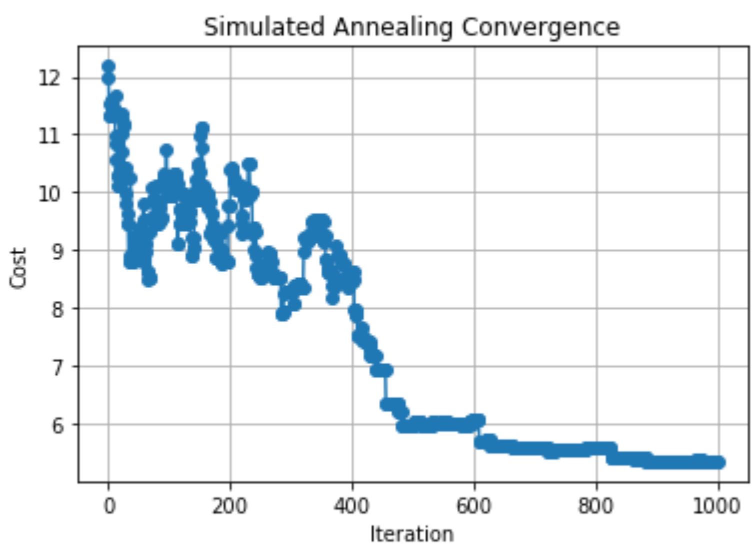
In [197…

```
# create a random solution
def random_tour(num_cities):
    numbers = list(range(1, num_cities))

    # Shuffling the list to get a random order
    random.shuffle(numbers)

    return numbers

def compute_tour_cost(cities):
    total_cost = 0
    for i in range(0, len(cities)-1):
        curr_cost = dist_matrix[cities[i]][cities[i+1]]
        total_cost += curr_cost
    total_cost += dist_matrix[cities[0]][cities[-1]]
    return total_cost

def exp_decay(time, init_temp=100, min_temp=0.001, exp_const=0.005):
    return max(init_temp * math.exp(-exp_const * time), min_temp)

def swap_cities(tour):
    tour_copy = copy.deepcopy(tour)
    rand_city_i1 = np.random.randint(0, len(tour))
    rand_city_i2 = np.random.randint(0, len(tour))

    while rand_city_i1 == rand_city_i2:
        rand_city_i2 = np.random.randint(0, len(tour))

    rand_city_1 = tour_copy[rand_city_i1]
    tour_copy[rand_city_i1] = tour_copy[rand_city_i2]
    tour_copy[rand_city_i2] = rand_city_1

    return tour_copy
```

Simulated Annealing

In [198…
```
def simulated_annealing(init_temp, max_iter=1000):
    curr_tour = random_tour(25)
    costs = np.zeros(max_iter)
    for i in range(0, max_iter):
        new_tour = swap_cities(curr_tour)

        old_cost = compute_tour_cost(curr_tour)
        new_cost = compute_tour_cost(new_tour)

        curr_temp = exp_decay(i, init_temp=init_temp)
        curr_cost = old_cost
        cost_diff = new_cost - old_cost


        if cost_diff < 0 or np.random.rand() < np.exp(-cost_diff / curr_temp
            curr_tour = new_tour
            curr_cost = new_cost

        costs[i] = curr_cost
    return curr_tour, costs
```

In [199…
```
init_temp = 1
iterations = 1000
curr_tour, costs = simulated_annealing(init_temp, iterations)
plt.figure()
plt.plot(range(0, iterations), costs, marker='o')
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Simulated Annealing Convergence")
plt.grid(True)
plt.show()
```



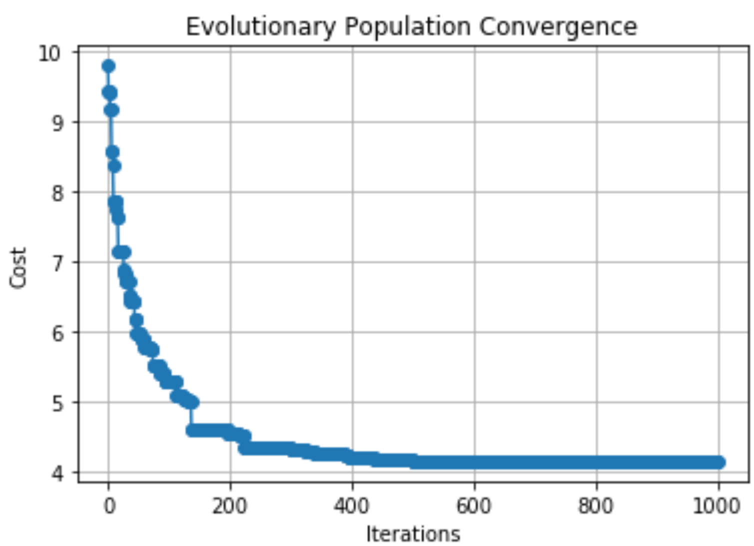
Evolutionary Algorithm

In [200…

```
# mutation in this case is swapping two random cities in the tour
def evolutionary_algorithm(initial_tour_num, evolutions):
    costs = []
    # create k number of initial tours in the initial population
    population = []
    for _ in range(0,initial_tour_num):
        population.append(random_tour(25))

    for _ in range(0, evolutions):
        # generate k successor tours
        successor_tours = []
        for tour in population:
            # mutation is to swap two tours
            successor_tours.append(swap_cities(tour))
        combined_populations = population + successor_tours
        # calculate the costs of all the tours
        all_costs = [compute_tour_cost(tour) for tour in combined_populatior
        # select the k best based on best = lowest cost
        combined_populations = list(zip(combined_populations, all_costs))
        sorted_populations = sorted(combined_populations, key = lambda x: x[
        # remove the k-worst
        new_population = [pair[0] for pair in sorted_populations[:(initial_t
        population = new_population
        costs.append(sorted_populations[0][1])

    return sorted_populations[0][0], costs
```

In [201…

```
population_size = 20
iterations = 1000
curr_tour, costs = evolutionary_algorithm(population_size, iterations)
plt.figure()
plt.plot(range(0, iterations), costs, marker='o')
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.title("Evolutionary Population Convergence")
plt.grid(True)
plt.show()
```



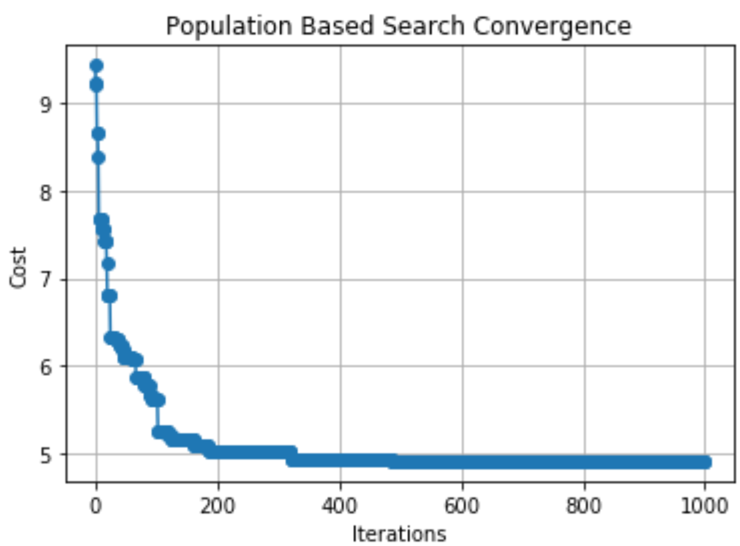
Population-based Search

In [202…

```
def population_based_search(beam_width, iterations):
    # start with k random tours
    tours = []
    for _ in range(0,beam_width):
        tours.append(random_tour(25))
    costs = []
    for _ in range (0, iterations):
        # generate successor tours
        successor_tours = []
        for tour in tours:
            # mutation is to swap two tours
            successor_tours.append(swap_cities(tour))
        combined_tours = tours + successor_tours
        # compute all the costs for the tours
        all_costs = [compute_tour_cost(tour) for tour in combined_tours]
        tours_with_costs = list(zip(combined_tours, all_costs))
        sorted_tours = sorted(tours_with_costs, key = lambda x: x[1])
        # select the top beam width tours
        new_set_tours = [pair[0] for pair in sorted_tours[:(beam_width)]]
        # save them for the next round
        tours = new_set_tours
        costs.append(sorted_tours[0][1])

    return sorted_tours[0][0], costs
```

In [203…

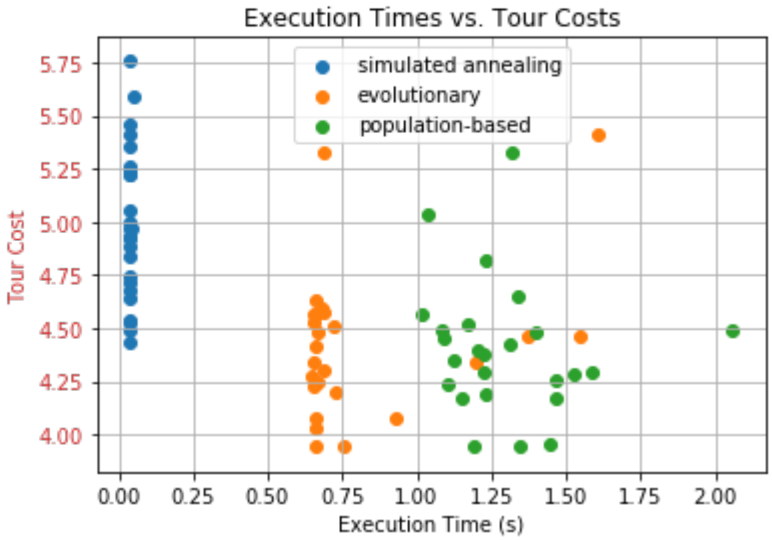
```
beam_width = 10
iterations = 1000
curr_tour, costs = population_based_search(beam_width, iterations)
plt.figure()
plt.plot(range(0, iterations), costs, marker='o')
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.title("Population Based Search Convergence")
plt.grid(True)
plt.show()
```



Experiments

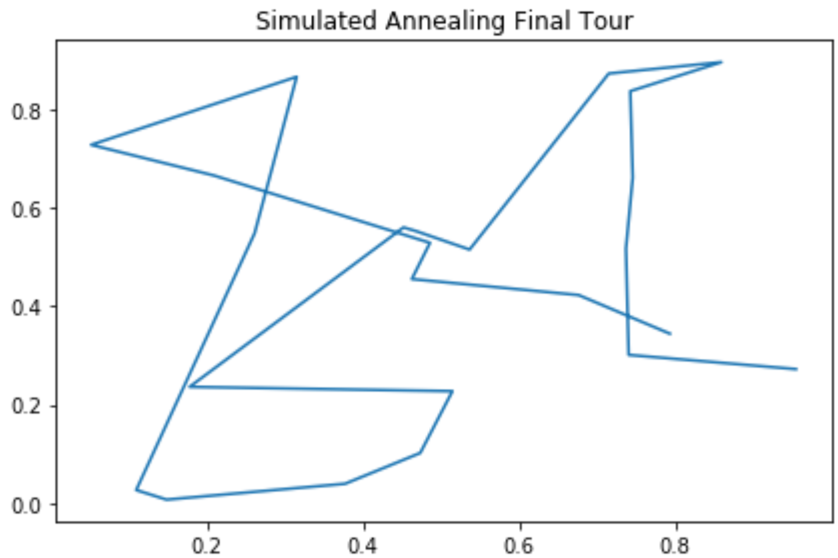
In [204…

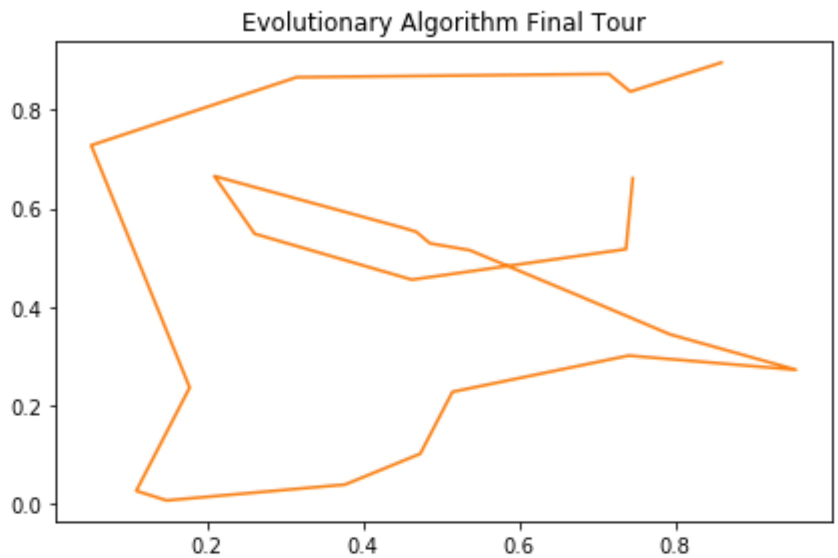
```
runs = 25
init_temp = 1
initial_num_tours = 20
beam_width = 20
iterations = 1000

sa_times = []
sa_cost = []
for _ in range(runs):
    start_time = time.time()  # Record the current time
    sa_tour, costs = simulated_annealing(init_temp, iterations)  # Call your
    end_time = time.time()  # Record the time after the function completes
    execution_time = end_time - start_time
    sa_times.append(execution_time)
    sa_cost.append(costs[-1])

ea_times = []
ea_cost = []
for _ in range(runs):
    start_time = time.time()  # Record the current time
    ea_tour, costs = evolutionary_algorithm(initial_num_tours, iterations)
    end_time = time.time()  # Record the time after the function completes
    execution_time = end_time - start_time
    ea_times.append(execution_time)
    ea_cost.append(costs[-1])

ps_times = []
ps_cost = []
for _ in range(runs):
    start_time = time.time()  # Record the current time
    ps_tour, costs = population_based_search(beam_width, iterations)  # Call
    end_time = time.time()  # Record the time after the function completes
    execution_time = end_time - start_time
    ps_times.append(execution_time)
    ps_cost.append(costs[-1])

# Create a figure and axis
fig, ax = plt.subplots()

# Scatter plot for execution times
ax.set_xlabel('Execution Time (s)')
ax.set_ylabel('Tour Cost', color='tab:red')
ax.scatter(sa_times, sa_cost, label='simulated annealing', color='tab:blue')
ax.scatter(ea_times, ea_cost, label='evolutionary', color='tab:orange')
ax.scatter(ps_times, ps_cost, label='population-based', color='tab:green')
ax.tick_params(axis='y', labelcolor='tab:red')

# Add a legend
ax.legend()

# Set the axis labels
plt.xlabel('Execution Time (s)')
plt.title('Execution Times vs. Tour Costs')
plt.grid()

# Show the scatterplot
plt.show()
```

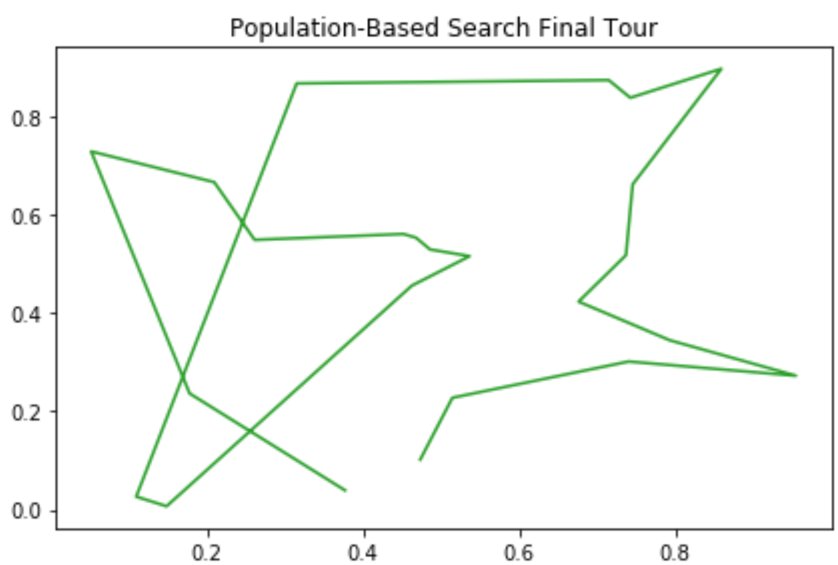


In [205…
```
x_sa = [city_points[i][0] for i in sa_tour]
y_sa = [city_points[i][1] for i in sa_tour]
x_ea = [city_points[i][0] for i in ea_tour]
y_ea = [city_points[i][1] for i in ea_tour]
x_ps = [city_points[i][0] for i in ps_tour]
y_ps = [city_points[i][1] for i in ps_tour]
```

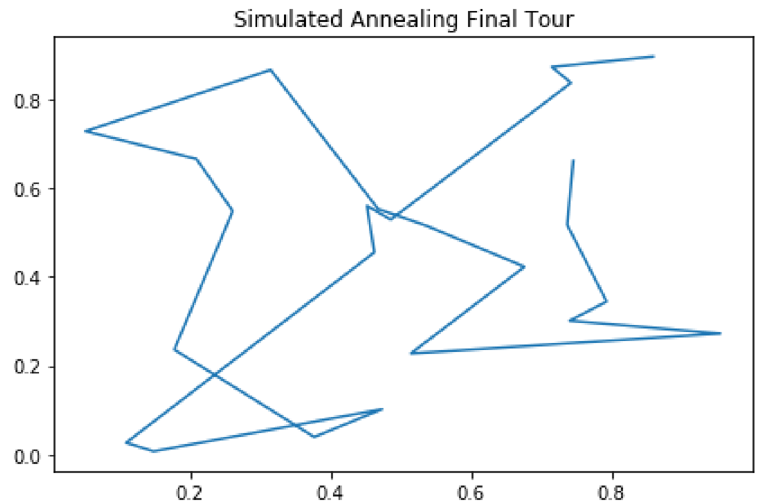
In [206…
```
fig, axs = plt.subplots(3, 1, figsize=(6, 12))

axs[0].plot(x_sa, y_sa, label='Simulated Annealing', color='tab:blue')
axs[1].plot(x_ea, y_ea, label='Evolutionary Algorithm', color='tab:orange')
axs[2].plot(x_ps, y_ps, label='Population-Based Search', color='tab:green')

# Set titles for  subplot
axs[0].set_title('Simulated Annealing Final Tour')
axs[1].set_title('Evolutionary Algorithm Final Tour')
axs[2].set_title('Population-Based Search Final Tour')

plt.tight_layout()

plt.show()
```

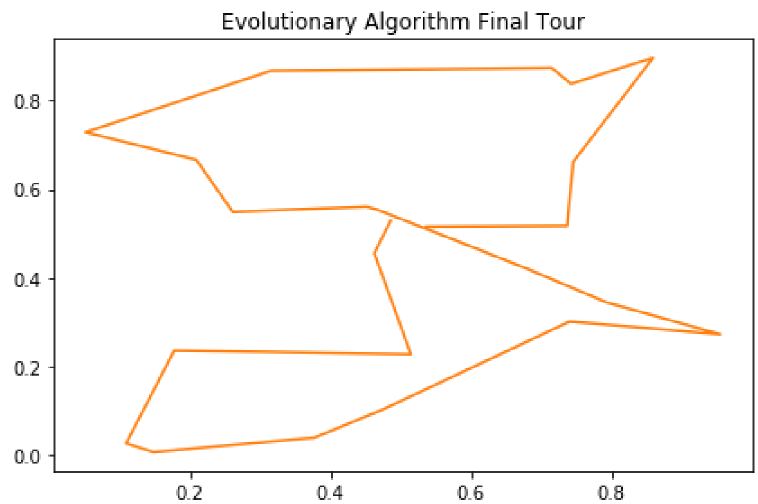

Simulated Annealing Final Tour
0.0
0.2
0.4
0.6
0.8
0.2
0.4
0.6
0.8

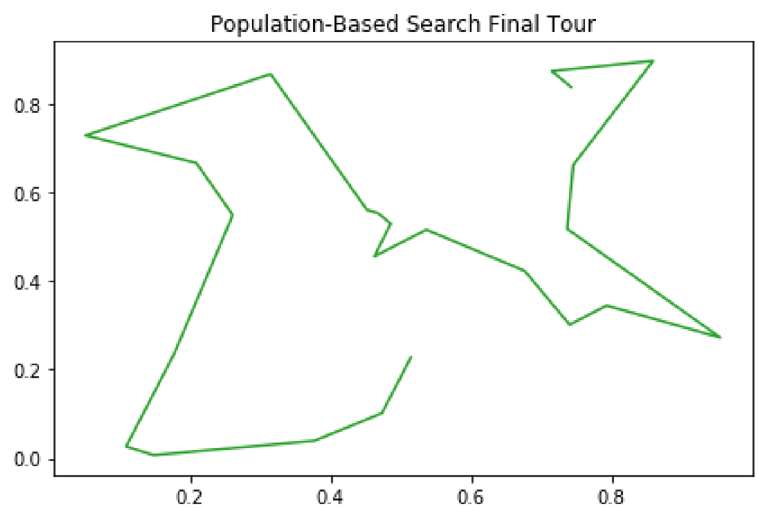

Evolutionary Algorithm Final Tour
0.0
0.2
0.4
0.6
0.8
0.2
0.4
0.6
0.8


Population-Based Search Final Tour
0.0
0.2
0.4
0.6
0.8
0.2
0.4
0.6
0.8

Final Tours


Simulated Annealing Final Tour
0.0
0.2
0.4
0.6
0.8
0.2
0.4
0.6
0.8


Evolutionary Algorithm Final Tour
0.0
0.2
0.4
0.6
0.8
0.2
0.4
0.6
0.8


Population-Based Search Final Tour
0.0
0.2
0.4
0.6
0.8
0.2
0.4
0.6
0.8