Homework 1: Traveling Salesman Problem

Rebecca Roskill
rcr2144

MECS 4510
Prof. Hod Lipson

Submitted: Tuesday, October 4th, 2021
Grace hours used: 0
Grace hours remaining: 96

1 Methods

1.1 Datasets: TSP 1 and TSP 2

1.2 Random Search

1.3 Random Mutation Hill Climber

1.4 Genetic Algorithm

1.1 Datasets: TSP 1 and TSP 2

One dataset was provided for the assignment, and one dataset was produced for testing purposes:

TSP 1 The dataset provided for the assignment consisted of 1000 points (x, y) that fall within x=(0, 1) and y=(0, 1). The points are clustered in four square formations.

TSP 2 The dataset produced for testing consisted of 100 points (x, y) that fall within x=[-1, 1]. and y=[-1, 1]. The points are uniformly distributed along the path of a circle with radius 1, centered at the origin.

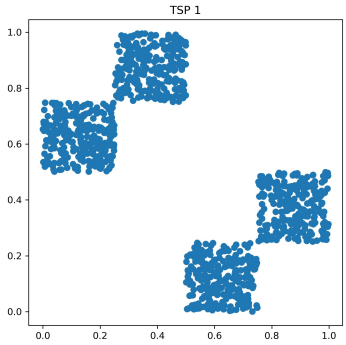

Figure 1.1a: TSP 1 dataset.

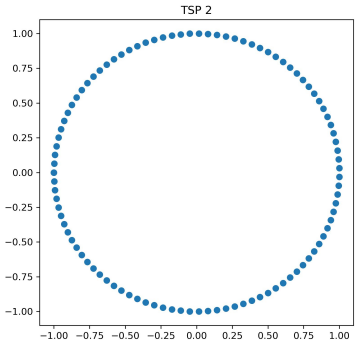

Figure 1.1b: TSP 2 dataset.

1.2 Random Search

Each trial run by *run_random_search* repeats the following process, maintaining the best path found (according to either longest or shortest path, depending on the experiment), as well as the current path as a measure of the learning curve, or "fitness":

1. The *uniform* function implemented in the Python *random* library was used to randomly initialize a priority weight [0, 1] for each point in the dataset.
2. The *assess_input* function was used to order the points by priority, simulating the process of visiting each location, and calculate the euclidean distance between each sequential pair of points.

1.3 Random Mutation Hill Climber

The *uniform* function implemented in the Python *random* library was used to randomly initialize a priority weight [0, 1] for each point in the dataset. Then, each trial run by *run_rmhc_search* repeats the following process, maintaining the best path found (according to either longest or shortest path, depending on the experiment), as well as the current path as a measure of the learning curve, or "fitness":

1. Using *get_random_neighbor,* one "neighbor" of the current ordering is found by swapping two points.
2. The *assess_input* function was used to order the neighbor's points by priority and calculate the euclidean distance between each sequential pair of points.
3. If the neighbor's path outperforms the current "best" path, the neighbor becomes the starting point for the next iteration. Otherwise, we keep the current ordering.

1.4 Genetic Algorithm

The *uniform* function implemented in the Python *random* library was used to randomly initialize a population of *n_processes* inputs, consisting of priority weights [0, 1] for each point in the dataset.

An optional step was performed in some versions of the algorithm (see *1.4.1*, *1.4.2*, *1.4.3*) to order the genes on the chromosome according to approximate regions. These regions were determined using the *KMeans* clustering implementation in the Scikit *sklearn* Python library. k=[1, 9] were tested sequentially until the improvement yielded by incrementing k was <10%.

Then, each trial run by *run_ga_search* repeats the following process, maintaining the best path found (according to either longest or shortest path, depending on the experiment), as well as the current path as a measure of the learning curve, or "fitness":

1. The *assess_input* function was used to order each input in the population's points by priority and calculate the euclidean distance between each sequential pair of points. The population was then sorted by this path distance, and the top-scoring p fraction was selected to reproduce.

…

1.4 Genetic Algorithm (cont.)

2. Using *reproduce,* 80% of the population undergoes mutation and 20% of the population undergoes recombination. This stage repeats to multiply the parents by $1/p$, thus maintaining the same population size for the offspring.

   Mutation – Five types of mutation were implemented:
   i. Random swap: The priority values of two random genes are swapped.
   ii. Random flip: The priority, x, of one random gene is flipped to $x–1$.
   iii. Intra-region swap: The priority values of two random genes within a single region are swapped.
   iv. Cross-region swap: The priority values of two random genes in different regions are swapped.
   v. Hybrid: 50% of the population undergoes "intra-region swap" mutation, while 50% undergoes "cross-region swap" mutation

   Recombination – Four types of recombination were implemented:
   i. Single-point: The first parent and the second parent undergo crossover at a single random point.
   ii. Intra-region: The child is the first parent, with one region replaced a single-point crossover between the first parent and the second parent in that region.
   iii. Cross regions: The child is the first parent, with one region replaced by the second parent's values in that region.
   iv. Hybrid: 50% of the population undergoes "intra-region" recombination, while 50% undergoes "cross regions" recombination.
3. The offspring become the future generation for the next iteration.

1.4.1 Genetic Algorithm: No linkage

- The chromosome is not ordered in a randomized fashion.
- Mutations are random swaps anywhere on the chromosome
- Recombinations are a single-point crossover at a random point, anywhere on the chromosome.

1.4.2 Genetic Algorithm: Reverse linkage → hybrid linkage → tight linkage

- The chromosome is ordered according to the region numbers yielded by k-means clustering. Genes within these regions are ordered in a randomized fashion.
- In the first stage, "reverse linkage," mutations are swaps between regions and crossovers are two-point: they return a copy of the first parent, with one entire region replaced by the second parent's.
- In the second stage, "hybrid linkage," 50% of the population undergoes reverse linkage reproduction, while 50% undergoes tight linkage reproduction.
- In the third and final stage, "tight linkage," mutations are swaps within regions and crossovers are are two-point: they return a copy of the first parent, with part of one region replaced by the second parent's.

1.4.3 Genetic Algorithm: Reverse linkage → hybrid linkage

- The chromosome is ordered as in *1.4.2*.
- In the first stage, "reverse linkage," mutations are swaps between regions and crossovers are two-point: they return a copy of the first parent, with one entire region replaced by the second parent's.
- In the second stage, "hybrid linkage," 50% of the population undergoes reverse linkage reproduction, while 50% undergoes tight linkage reproduction (described above in *1.4.2*).

2 Results

2.1 Performance across algorithms by dataset

2.2 Shortest and longest paths found

2.3 Search animations

2.1 Performance across algorithms by dataset

2.1.1 TSP 1

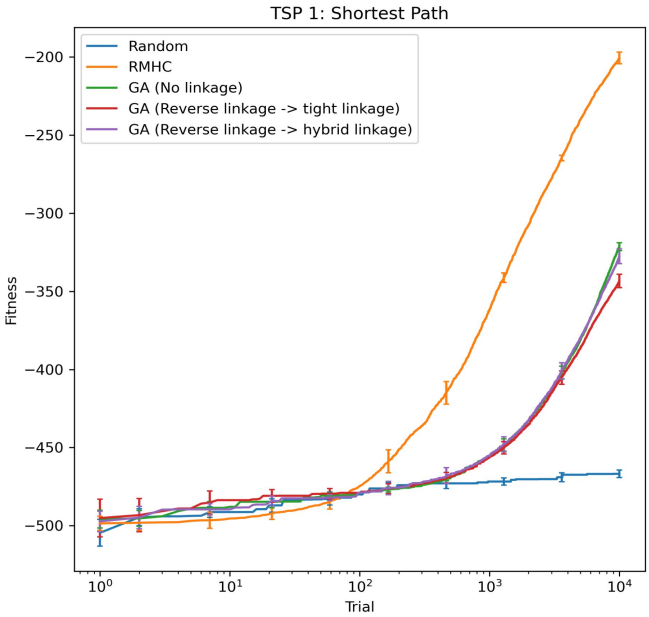

Figure 2.1.1a: Fitness performance across algorithms, searching for the shortest path in TSP 1.

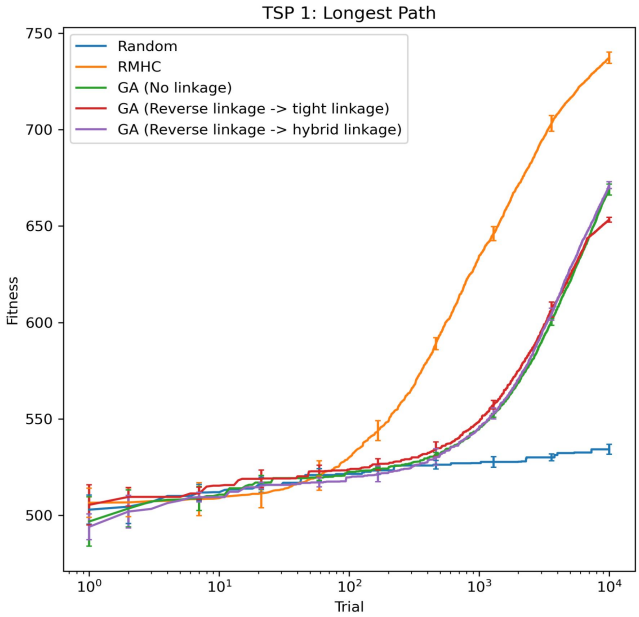

Figure 2.1.1b: Fitness performance across algorithms, searching for the longest path in TSP 1.

2.1.2 TSP 2

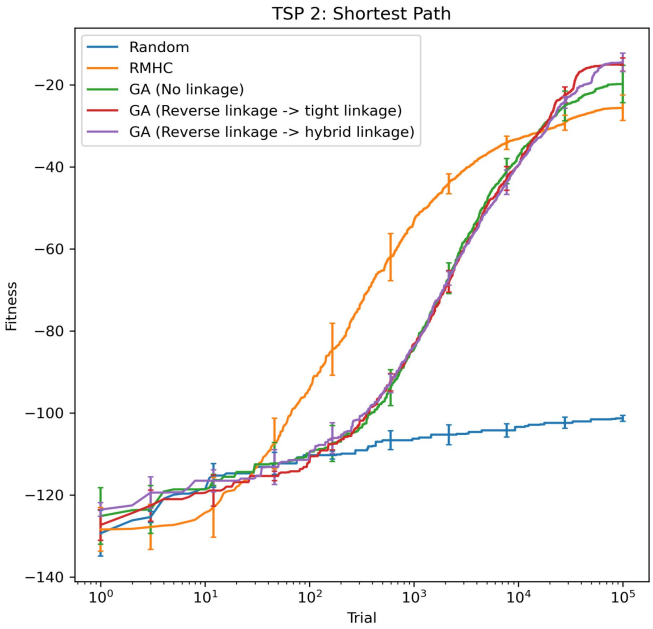

Figure 2.1.2a: Fitness performance across algorithms, searching for the shortest path in TSP 2.

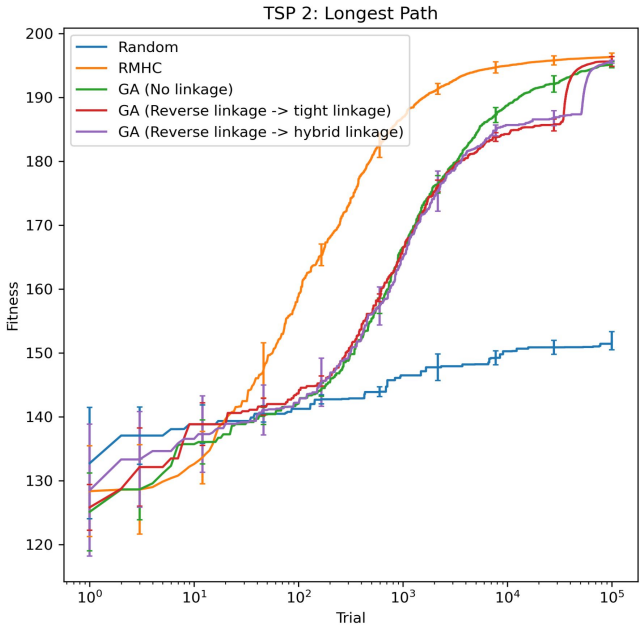

Figure 2.1.2b: Fitness performance across algorithms, searching for the longest path in TSP 2.

2.2 Shortest and longest paths found by algorithm

2.2.1 Random Search

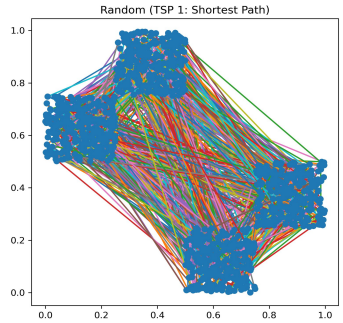

Figure 2.2.1a (left): Shortest path found by Random Search algorithm on TSP 1 dataset in 10^4 iterations; Path length = 465.26

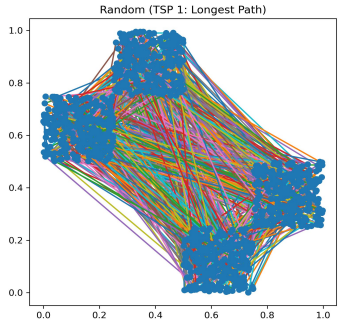

Figure 2.2.1b (left): Longest path found by Random Search algorithm on TSP 1 dataset in 10^4 iterations; Path length = 521.58

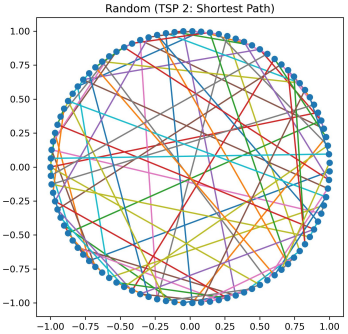

Figure 2.2.1c (left): Shortest path found by Random Search algorithm on TSP 2 dataset in 10^5 iterations; Path length = 99.17

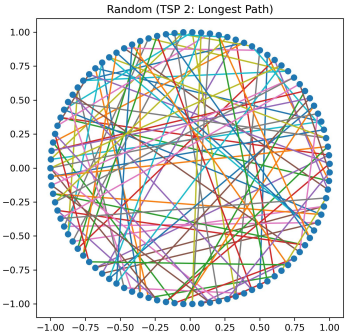

Figure 2.2.1d (left): Longest path found by Random Search algorithm on TSP 2 dataset in 10^5 iterations; Path length = 153.79

2.2.2 RMHC Search

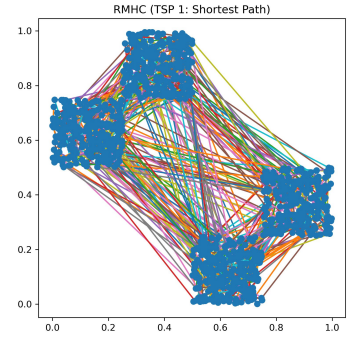

Figure 2.2.2a (left): Shortest path found by RMHC Search algorithm on TSP 1 dataset in 10^4 iterations; Path length = 194.92

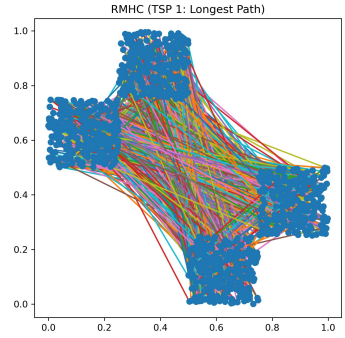

Figure 2.2.2b (left): Longest path found by RMHC Search algorithm on TSP 1 dataset in 10^4 iterations; Path length = 741.24

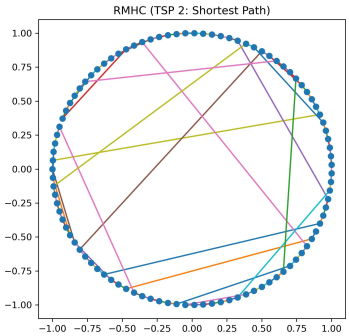

Figure 2.2.2c (left): Shortest path found by RMHC Search algorithm on TSP 2 dataset in 10^5 iterations; Path length = 26.96

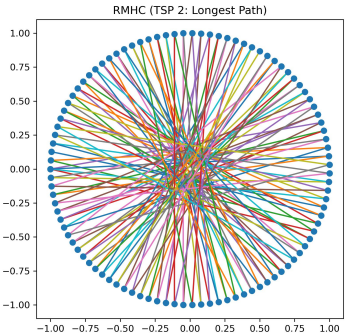

Figure 2.2.2d (left): Longest path found by RMHC Search algorithm on TSP 2 dataset in 10^5 iterations; Path length = 197.04

2.2.3 GA (No linkage) Search

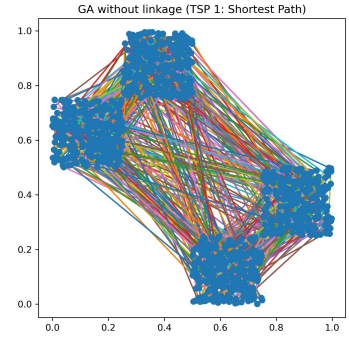

Figure 2.2.3a (left): Shortest path found by RMHC Search algorithm on TSP 1 dataset in 10^4 iterations; Path length = 322.94

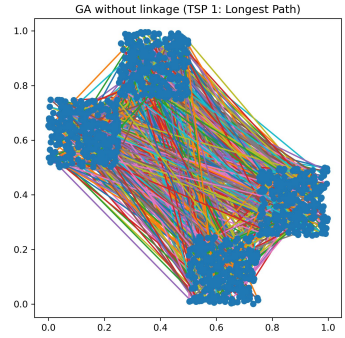

Figure 2.2.3b (left): Longest path found by GA (No linkage) Search algorithm on TSP 1 dataset in 10^4 iterations; Path length = 670.83

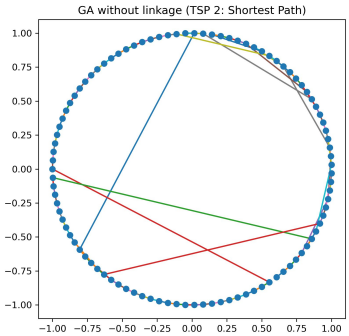

Figure 2.2.3c (left): Shortest path found by GA (No linkage) Search algorithm on TSP 2 dataset in 10^5 iterations; Path length = 18.14

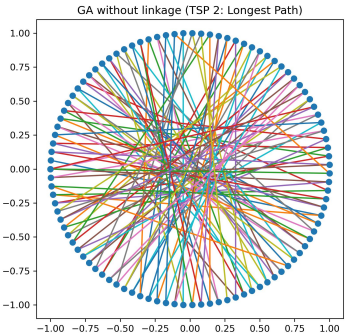

Figure 2.2.3d (left): Longest path found by GA (No linkage) Search algorithm on TSP 2 dataset in 10^5 iterations; Path length = 194.53

2.3 Search animations

RMHC

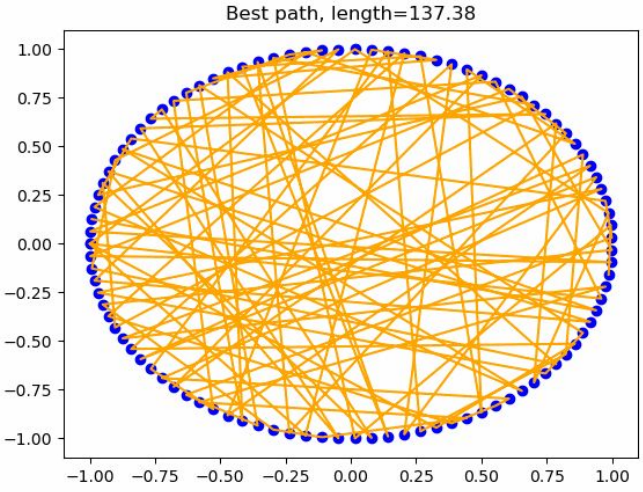

Figure 2.3a: RMHC on sparse circle dataset (10^5 iterations)

GA (No linkage)

Mutation=80% (swap); Recombination=20% (single-point)
No linkage, random ordering

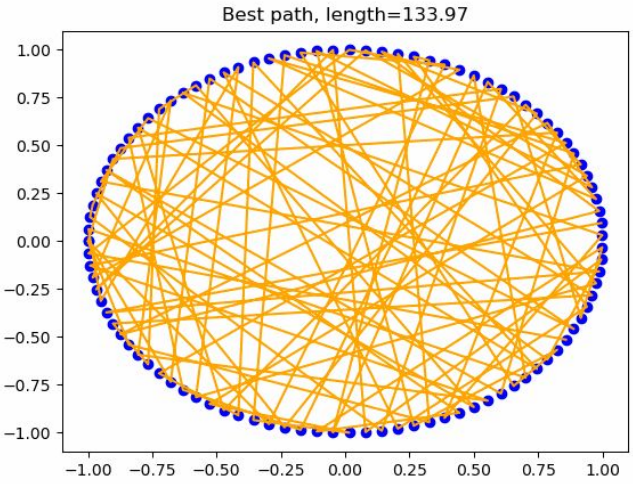

Figure 2.3b: GA (No linkage) on sparse circle dataset (10^5 iterations)

3 Analysis

3.1 Comparison of Random, RMHC, and GA performance

3.2 Effects of mutation-recombination ratio on GA performance

3.3 Effects of linkage on GA performance

3.1 Comparison of Random, RMHC, and GA performance

3.1.1 TSP 1

The shortest-path learning curves achieved by the Random, RMHC, and GA search algorithms demonstrate that RMHC and GA both become favorable to Random within about 10^2 iterations (Figure *2.1.1a*). After this point, Random plateaus to an almost non-improving state, while RMHC and GA maintain super-log improvement through a point around 10^3 iterations. At that point, an inflection point occurs where RMHC begins to slow to sub-log improvement, while GA continues to achieve log or super-log improvement.

This section of my study was significantly hindered by the runtime performance of Python, which I will take into account in future assignments. While interesting differences between the algorithms can be observed within the first 10^4 iterations, the long-term behavior revealed when leveraging a smaller dataset to execute more iterations (see *3.1.2*) cannot be observed. It therefore cannot be concluded from this study that GA will overtake RMHC, although RMHC seems to be plateauing to sub-log improvement in later iterations, while GA maintains approximately log improvement (Figure *2.1.1a*).

3.1.2 TSP 2

Using this smaller dataset to execute more iterations, the long-term behavior of RMHC and GA for shortest-path search can be observed. These results demonstrate the superiority of GA beyond about 10^4 iterations (Figure *2.1.2a*). It is, however, worth noting that GA also seems to plateau shortly after overtaking RMHC.

3.2 Effects of mutation-recombination ratio on GA performance

At constant population (*n_processes*=100) and linkage technique ("No linkage"), the ratio between mutation and recombination in reproduction was varied between 0.2:0.8, 0.5:0.5, 0.8:0.2, and 0.9:0.1. Based on the results (long-term behavior best visualized in Figure *3.2b*), 0.8:0.2 was used in the primary experiment to compare algorithms.

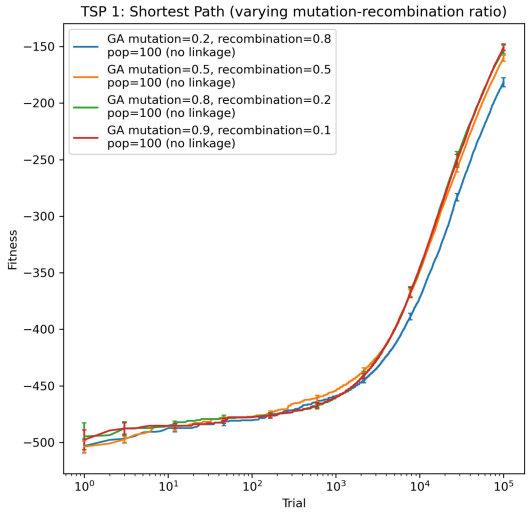

Figure 3.2a: Fitness performance across algorithms, searching for the shortest path in TSP 1.

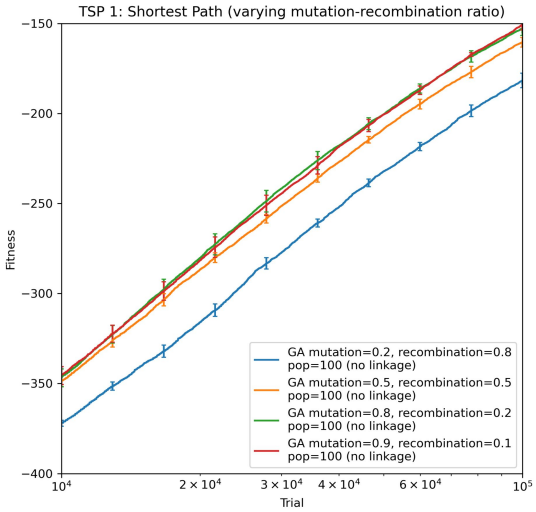

Figure 3.2b: Fitness performance across algorithms, searching for the longest path in TSP 1.

3.2 Effects of linkage on GA performance

The various versions of the GA algorithm (described in *1.4*) experimented with techniques for linking genes on the chromosome and mutating and recombining the population in accordance with these linkages. The results were inconclusive as to the best linkage technique for searching for the shortest path in TSP 1 over 10^4 iterations, as both "No linkage" and "Reverse linkage → hybrid linkage" compete for the best GA in this setting (Figure *2.1.1a*). However, when the techniques were applied to search for the shortest path in TSP 2 over 10^5 iterations, both "Reverse linkage → hybrid linkage" and "Reverse linkage → tight linkage" demonstrated improvements over "No linkage" (Figure *2.1.2a*).

Some intuition for why this might be the case: The algorithms undergo an adequate number of iterations to form similar priorities within regions, by mutating and recombining such that genes can swap regions. Then, when the points are adequately ordered by region, both algorithms increase the opportunity for mutations within the regions themselves, so as to order the path between clustered points more favorably. This intuition is supported by the jumps achieved by the algorithms where they change mutation/combination types (near $3x10^4$ for "Reverse linkage → hybrid linkage" and near $5x10^4$ for "Reverse linkage → tight linkage"). The jumps are much greater in the TSP 2 experiment with more iterations, indicating that the TSP 1 experiment possibly had yet to reach a point where it would have as much to gain from switching mutation/combination types.

4 Appendix

4.1 Produce circular dataset

```
import numpy as np
from matplotlib import pyplot as plt

# Make equidistant circular data
n_points = 1000
r = 1
theta = np.linspace(-np.pi, np.pi, 100)
x = r * np.cos(theta)
y = r * np.sin(theta)
points = list(zip(x,y))

# Plot data
fig = plt.figure(figsize=(5,5))
plt.scatter(x, y)
plt.show()

# Write data to txt file
with open('tsp_circle.txt', 'w') as f:
   f.writelines(['{}, {}\n'.format(round(p[0], 4), round(p[1],
       4)) for p in points])
```

4.2 Search algorithm helper functions

```
from matplotlib import pyplot as plt
import random
import numpy as np
import pandas as pd
from pathos.multiprocessing import ProcessPool
import csv

from sklearn.cluster import KMeans
import matplotlib.cm as cm
random_state = 170


def load_dataset(path):

   # Load dataset from .txt file
   with open(path, 'r') as f:
      lines = f.readlines()
   return [[float(x) for x in line[:-1].split(',')] for line
       in lines]
```

```
def plot_points(points):
   plt.figure(figsize=(10, 10))
   x = [p[0] for p in points]
   y = [p[1] for p in points]
   plt.scatter(x, y)
   plt.show()


def measure_path(path):
   distance = 0
   for i in range(len(path) - 1):
      p1 = path[i]
      p2 = path[i + 1]
      distance += np.sqrt(np.square(p2[1] - p1[1]) +
          np.square(p2[0]
                    - p1[0]))
   return distance


def assess_input(input, points):

   # Create ordering based on the priority weights
   path = [p for (_, p) in sorted(zip(input, points))]

   # Assess fitness based on path length of the ordering
   score = measure_path(path)
   return (score, path)


def get_random_inputs(points):
   inputs = [random.uniform(0, 1) for p in points]
   return inputs

def group_regions(points):
   n_choices = range(1, 10)
   prev_err = float('inf')
   prev_region_nos = None
   prev_n = None

   # Try clustering with a few different n_clusters
   for n in n_choices:
      X = [list(p) for p in points]
      k_means = KMeans(n_clusters=n, random_state=random_state)
      region_nos = k_means.fit_predict(X)

      # Get error, compare to previous n to see if it improves
          by >10%
      err = sum([min(e) for e in k_means.transform(X)])
      if err / prev_err > 0.9:
```

```
            return (prev_n, prev_region_nos)

        # Plot clustering
        plt.figure(figsize=(5, 5))
        plt.scatter([x[0] for x in X], [x[1] for x in X],
            c=[cm.hot(r
                  / n) for r in region_nos])
        plt.title('K-means clustering with n=' + str(n))
        plt.show()

        # Store current values to compare against next for
            improvement
        prev_n = n
        prev_err = err
        prev_region_nos = region_nos

    # Default to highest n if improvement continued >10%
        throughout range
    return (n, region_nos)

def get_initial_population(points, n, order_by_cluster=False,
    initialize_by_cluster=False):
    region_bounds=None

    # Place cities on the gene (list) in accordance with region
    if order_by_cluster:
        points_ordered = []
        (n_regions, region_nos) = group_regions(points)
        region_bounds = []
        last_region_end = 0
        for i in range(n_regions):
            points_in_region = [p for (j, p) in enumerate(points)
                            if region_nos[j] == i]
            points_ordered += points_in_region
            region_bounds += [[last_region_end, last_region_end
                          + len(points_in_region)]]
            last_region_end += len(points_in_region)
    else:
        points_ordered = points

    # Initialize priority weights for each city to random
        values in some range
    # to group regions
    if initialize_by_cluster:
        inputs = []
        for i in range(n):
            input_i = []
            region_priority_span = 1/(len(region_bounds))
            for j, region in enumerate(region_bounds):
                points_in_region =
```

```
                points_ordered[region[0]:region[1]]
            input_i += [random.uniform(j*region_priority_span,
                (j+1) * region_priority_span) for p in
                points_in_region]
        inputs += [input_i]
    # Randomize priority weights for each city
    else:
        inputs = [[random.uniform(0, 1) for p in points_ordered]
            for i in
                range(n)]
    return inputs, points_ordered, region_bounds

def plot_path(path, ax=None):
    if ax == None:
        plt.figure(figsize=(5, 5))
    x = [p[0] for p in path]
    y = [p[1] for p in path]
    plt.scatter(x, y, zorder=2)
    for i in range(1, len(path)):
        (x1, y1) = path[i - 1]
        (x2, y2) = path[i]
        plt.plot([x1, x2], [y1, y2], c='orange', zorder=1)
    if ax == None:
        plt.show()
```

4.3 Random Search

```
def run_random_search(points, n_trials, plot=False):
    print ('Random Search with', n_trials, 'trials')

    # Prep data storage for trials
    trials = range(n_trials + 1)
    best_dist = [float('inf')]
    best_path = None
    worst_dist = 0
    worst_path = None
    learning_curve = [float('inf')]

    for i in range(n_trials):
        if i % (n_trials / 10) == 0:
            print ('Trial', i, 'of', n_trials)
            print ('Best score', round(best_dist[-1], 2))

        # Get a random set of priorities, measure path length
        inputs = get_random_inputs(points)
        (distance, path) = assess_input(inputs, points)

        # Update best distance and learning curve
```

```
            learning_curve += [distance]
            if distance < best_dist[-1]:
                best_dist += [distance]
                best_path = path
            else:
                best_dist += [best_dist[-1]]
            if distance > worst_dist:
                worst_dist = distance
                worst_path = path

        # Plot best and worst path found over trials
        if plot:
            plt.figure(figsize=(10, 5))
            ax = plt.subplot(121)
            plot_path(best_path, ax)
            plt.title('Best path,
                length={}'.format(round(best_dist[-1], 2)))
            plt.subplot(122)
            plot_path(worst_path, ax)
            plt.title('Worst path,
                length={}'.format(round(worst_dist, 2)))
            plt.show()

        # Compile data
        trials_df = pd.DataFrame({'trial': trials,
                                  'best_distance': best_dist,
                                  'learning_curve': learning_curve})
        return (trials_df, best_path)
```

4.4 Random Mutation Hill Climber Search

```
def get_random_neighbor(inputs):
    # Choose two neighbors to randomly swap
    i = random.choice(range(len(inputs)))
    j = random.choice([x for x in range(len(inputs)) if x != i])

    # Produce neighbor and perform swap
    neighbor = inputs.copy()
    neighbor[i] = inputs[j]
    neighbor[j] = inputs[i]
    return neighbor

def run_rmhc_search(points, n_trials, order_by_cluster=False,
    initialize_by_cluster=False, plot=False):
    print ('Hill Climber Random Search with', n_trials,
        'trials')

    # Prep data storage for trials
```

```
trials = range(n_trials + 1)
learning_curve = [float('inf')]
best_dist = [float('inf')]
best_path = None
worst_dist = 0
worst_path = None

# Start with a set of priorities, measure path length
population, points_ordered, region_bounds =
    get_initial_population(points, 1, order_by_cluster,
    initialize_by_cluster)
inputs = population[0]
(distance, path) = assess_input(inputs, points_ordered)
for i in range(n_trials):
   if i % (n_trials / 10) == 0:
      print ('Trial', i, 'of', n_trials)
      print ('Best score', round(best_dist[-1], 2))

   # Randomly swap two priorities (get a neighbor), measure
       new path length
   neighbor = get_random_neighbor(inputs)
   (n_distance, n_path) = assess_input(neighbor,
       points_ordered)

   # If neighbor is an improvement, start next iteration
       from neighbor
   if n_distance < distance:
      inputs = neighbor
      distance = n_distance
      path = n_path

   # Update best distance and learning curve
   learning_curve += [distance]
   if distance < best_dist[-1]:
      best_dist += [distance]
      best_path = path
   else:
      best_dist += [best_dist[-1]]
   if distance > worst_dist:
      worst_dist = distance
      worst_path = path

# Plot best and worst path found over trials
if plot:
   plt.figure(figsize=(10, 5))
   ax = plt.subplot(121)
   plot_path(best_path, ax)
   plt.title('Best path,
       length={}'.format(round(best_dist[-1], 2)))
   plt.subplot(122)
```

```
        plot_path(worst_path, ax)
        plt.title('Worst path,
            length={}'.format(round(worst_dist, 2)))
        plt.show()

    # Compile data
    trials_df = pd.DataFrame({'trial': trials,
                              'best_distance': best_dist,
                              'learning_curve': learning_curve})
    return (trials_df, best_path)
```

4.5 Genetic Algorithm Search

```
'''
mutation_type: 'flip':
            'swap': swap two random cities,
            'swap cross-region': swap two cities in different
                regions
               (requires regions param),
            'swap intra-region': swap two cities in the same
                region
               (requires regions param),
            'hybrid': swap two cities in different regions in
                0.5 parents,
                    swap two cities in the same region in 0.5
                        parents,
               (requires regions param)
'''
def mutate(parents, mutation_type='swap', regions=None):
    offspring = []
    if mutation_type == 'flip':
        for parent in parents:
            i = random.choice(range(len(parent)))
            child = parent.copy()
            child[i] = 1-child[i]
            offspring += [child]
    elif mutation_type == 'swap':
        for parent in parents:
            offspring += [get_random_neighbor(parent)]
    elif mutation_type == 'swap cross-region':
        for parent in parents:
            region1 = random.choice(regions)
            region2 = random.choice(regions)
            while region2 == region1:
                region2 = random.choice(regions)
            i = random.choice(range(region1[0], region1[1]))
            j = random.choice(range(region2[0], region2[1]))
            child = parent.copy()
```

```
            child[i] = parent[j]
            child[j] = parent[i]
            offspring += [child]
    elif mutation_type == 'swap intra-region':
        for parent in parents:
            region = random.choice(regions)
            i = random.choice(range(region[0], region[1]))
            j = random.choice(range(region[0], region[1]))
            while i == j:
                j = random.choice(range(region[0], region[1]))
            child = parent.copy()
            child[i] = parent[j]
            child[j] = parent[i]
            offspring += [child]
    elif mutation_type == 'hybrid':
        halfway = int(0.5*len(parents))
        inter = mutate(parents[halfway:], mutation_type='swap
            cross-region', regions=regions)
        offspring += inter
        intra = mutate(parents[:halfway], mutation_type='swap
            intra-region', regions=regions)
        offspring += intra
    return offspring

'''
recombination_type: 'single-point': cross parents at a single,
    random point,
                'cross regions': cross parents only at region
                    boundaries
                    (requires regions param),
                'intra-region': cross parents at a random point
                    in each region
                    (requires regions param),
                'hybrid': cross 0.5 parents only at region
                    boundaries,
                        0.5 at a random point in each region
                    (requires regions param),
'''
def recombine(parents, recombination_type, regions):
    offspring = []
    if len(parents) > 2:
        if recombination_type == 'single-point':
            for i in range(len(parents) - 1):
                parent1 = parents[i]
                parent2 = parents[i+1]
                crossover_point = random.randint(0, len(parent1))
                offspring += [ parent1[:crossover_point] +
                    parent2[crossover_point:] ]
        elif recombination_type == 'cross regions':
            for i in range(len(parents) - 1):
```

```
            parent1 = parents[i]
            parent2 = parents[i+1]
            region = random.choice(regions)
            offspring += [ parent1[:region[0]] +
                parent2[region[0]:region[1]] +
                parent1[region[1]:]]
        elif recombination_type == 'intra-region':
            for i in range(len(parents) - 1):
                parent1 = parents[i]
                parent2 = parents[i+1]
                region = random.choice(regions)
                i = random.choice(range(region[0], region[1]))
                offspring += [ parent1[:i] + parent2[i:region[1]]
                    + parent1[region[1]:]]
        elif recombination_type == 'hybrid':
            halfway = int(0.5*len(parents))
            intra = recombine(parents[:halfway],
                recombination_type='cross regions',
                regions=regions)
            offspring += intra
            inter = recombine(parents[halfway:],
                recombination_type='intra-region', regions=regions)
            offspring += inter
    else:
        offspring = parents
    return offspring


def reproduce(parents, n_offspring, r_mutate=0.5,
    mutation_type='swap',
           recombination_type='single-point', regions=None):
    offspring = []

    parents_copy = parents.copy()
    while len(parents_copy) < n_offspring:
        random.shuffle(parents)
        parents_copy += parents
    parents = parents_copy[:n_offspring]

    input_mutate = parents[:int(r_mutate * len(parents))]
    input_recombine = parents[int(r_mutate * len(parents)):]
    offspring += mutate(input_mutate, mutation_type, regions)
    offspring += recombine(input_recombine, recombination_type,
        regions)

    # Sometimes offspring isn't the right length
    while len(offspring) < n_offspring:
        offspring += mutate([random.choice(parents) for i in
            range(6)], mutation_type, regions)
        offspring += recombine([random.choice(parents) for i in
```

```
            range(6)], recombination_type, regions)

    offspring = offspring[:n_offspring]

    return offspring


def run_ga_search(points, n_trials, n_processes, plot=True,
    r_mutate=0.8,
              order_by_cluster=False,
                  initialize_by_cluster=False,
              mutation_types=['swap'],
                  recombination_types=['single-point']):
    print ('Simple Evolutionary Search with', n_trials, 'trials
        and',
          n_processes, 'processes')
    if not len(recombination_types)==len(mutation_types):
        raise Exception('recombination_types must have the same
            length as mutation_types')

    # Prep data storage for trials
    n_batches = int(n_trials / n_processes)
    trials = np.arange(0, n_trials + 1)
    learning_curve = [float('inf')]
    best_dist = [float('inf')]
    best_path = None
    worst_dist = 0
    worst_path = None

    top_p = 0.2
    top_k = int(top_p * n_processes)
    pool = ProcessPool(nodes=8)

    inputs, points, region_bounds =
        get_initial_population(points, n_processes,
        order_by_cluster, initialize_by_cluster)
    batches_per_reproduction_type = n_batches /
        len(recombination_types)

    # Look for best path while running evolutionary process
    for n in range(n_batches - 1):
        reproduction_type_round =
            int(n/batches_per_reproduction_type)
        recombination_type =
            recombination_types[reproduction_type_round]
        mutation_type = mutation_types[reproduction_type_round]
        print ('Trial', n * n_processes, 'of', n_batches *
            n_processes)
        print ('Best score', best_dist[-1])
        score_path_pairs = [x for x in pool.imap(assess_input,
```

```
            inputs,
                        [points for i in inputs])]
        scores = [score for (score, _) in score_path_pairs]

        # Create new inputs: perform operators
            (crossovers/mutations/segment inversion)
        top_inputs = [i for (_, i) in sorted(zip(scores,
                    inputs))][:top_k]
        for s, p in score_path_pairs:
            if s < best_dist[-1]:
                best_dist += [s]
                best_path = p
            else:
                best_dist += [best_dist[-1]]
            if s > worst_dist:
                worst_dist = s
                worst_path = p
            learning_curve += [s]
        inputs = reproduce(top_inputs, n_processes, r_mutate,
            mutation_type, recombination_type, region_bounds)

    score_path_pairs = [x for x in pool.imap(assess_input,
        inputs,
                    [points for i in inputs])]
    for s, p in score_path_pairs:
        if s < best_dist[-1]:
            best_dist += [s]
            best_path = p
        else:
            best_dist += [best_dist[-1]]
        learning_curve += [s]
        if s > worst_dist:
            worst_dist = s
            worst_path = p

    # Plot best and worst path found over trials
    if plot:
        plt.figure(figsize=(10, 5))
        ax = plt.subplot(121)
        plot_path(best_path, ax)
        plt.title('Best path,
            length={}'.format(round(best_dist[-1], 2)))
        plt.subplot(122)
        plot_path(worst_path, ax)
        plt.title('Worst path,
            length={}'.format(round(worst_dist, 2)))
        plt.show()

    # Compile data
    trials_df = pd.DataFrame({'trial': trials,
```

```
                    'best_distance': best_dist,
                    'learning_curve': learning_curve})
    return (trials_df, best_path)
```

4.6 Animations

```
from matplotlib.animation import FuncAnimation, PillowWriter

def get_ga_frames(points, n_trials, n_processes, plot=True,
    inverted=False, r_mutate=0.8,
              order_by_cluster=False,
                  initialize_by_cluster=False,
              mutation_types=['swap'],
                  recombination_types=['single-point']):
    print ('Simple Evolutionary Search with', n_trials, 'trials
        and',
          n_processes, 'processes')
    if not len(recombination_types)==len(mutation_types):
        raise Exception('recombination_types must have the same
            length as mutation_types')

    # Prep data storage for trials
    n_batches = int(n_trials / n_processes)
    best_dist = [float('inf')]
    best_paths = []

    top_p = 0.2
    top_k = int(top_p * n_processes)
    pool = ProcessPool(nodes=8)

    inputs, points, region_bounds =
        get_initial_population(points, n_processes,
        order_by_cluster, initialize_by_cluster)
    batches_per_reproduction_type = n_batches /
        len(recombination_types)

    # Look for best path while running evolutionary process
    for n in range(n_batches - 1):
        reproduction_type_round =
            int(n/batches_per_reproduction_type)
        recombination_type =
            recombination_types[reproduction_type_round]
        mutation_type = mutation_types[reproduction_type_round]
        print ('Trial', n * n_processes, 'of', n_batches *
            n_processes)
        print ('Best score', best_dist[-1])
        score_path_pairs = [x for x in pool.imap(assess_input,
            inputs,
```

```
                    [points for i in inputs], [inverted for i
                        in inputs])]
        scores = [score for (score, _) in score_path_pairs]

        # Create new inputs: perform operators
            (crossovers/mutations/segment inversion)
        top_inputs = [i for (_, i) in sorted(zip(scores,
                  inputs))][:top_k]
        for s, p in score_path_pairs:
            if s < best_dist[-1]:
                best_dist += [s]
                best_paths += [(p,s)]
            else:
                best_dist += [best_dist[-1]]

        inputs = reproduce(top_inputs, n_processes, r_mutate,
            mutation_type, recombination_type, region_bounds)

    score_path_pairs = [x for x in pool.imap(assess_input,
        inputs,
                    [points for i in inputs], [inverted for i in
                        inputs])]
    for s, p in score_path_pairs:
        if s < best_dist[-1]:
            best_dist += [s]
            best_paths += [(p,s)]
        else:
            best_dist += [best_dist[-1]]

    return best_paths

def get_rmhc_frames(points, n_trials, order_by_cluster=False,
    initialize_by_cluster=False, plot=True, inverted=False):
    print ('Hill Climber Random Search with', n_trials,
        'trials')

    # Prep data storage for trials
    best_dist = [float('inf')]
    best_paths = []

    # Start with a set of priorities, measure path length
    population, points_ordered, region_bounds =
        get_initial_population(points, 1, order_by_cluster,
        initialize_by_cluster)
    inputs = population[0]
    (distance, path) = assess_input(inputs, points_ordered,
        inverted)
    for i in range(n_trials):
        if i % (n_trials / 10) == 0:
            print ('Trial', i, 'of', n_trials)
```

```
            print ('Best score', round(best_dist[-1], 2))

        # Randomly swap two priorities (get a neighbor), measure
            new path length
        neighbor = get_random_neighbor(inputs)
        (n_distance, n_path) = assess_input(neighbor,
            points_ordered, inverted)

        # If neighbor is an improvement, start next iteration
            from neighbor
        if n_distance < distance:
            inputs = neighbor
            distance = n_distance
            path = n_path

        # Update best distance and learning curve
        if distance < best_dist[-1]:
            best_dist += [distance]
            best_paths += [(path, distance)]

    return best_paths

points = load_dataset('tsp_circle_sparse.txt')
n_trials = 100000
paths = get_ga_frames(points, n_trials, 50)
    #get_rmhc_frames(points, n_trials)

fig, ax = plt.subplots()
x, y = [], []
ln, = plt.plot([], [], 'orange')

def init():
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    return ln,

def update(i):
    x = [p[0] for p in paths[i][0]]
    y = [p[1] for p in paths[i][0]]
    ln.set_data(x, y)
    plt.cla()
    plt.plot(x, y, 'orange')
    plt.scatter(x, y, c='b')

    plt.title('Best path, length={}'.format(round(paths[i][1],
        2)))
    return ln,

anim = FuncAnimation(fig, update, frames=len(paths),
                init_func=init, blit=True)
```

```
writergif = PillowWriter(fps=30)
anim.save('animation.gif', writer=writergif)
plt.show()
```