Applying Genetic Algorithms to the TSP

Abby Butel-Fry

23 December 2025

Short Abstract

The Traveling Salesperson Problem (TSP) is a combinatorial optimization problem in which a salesperson must visit each city in a set exactly once and return to the starting city, while minimizing the total distance traveled. This paper discusses solving the TSP with the use of genetic algorithms.

Medium Abstract

The Traveling Salesperson Problem (TSP) is a well-known combinatorial optimization problem for which finding guaranteed optimal solutions becomes computationally prohibitive as problem size increases. As a result, heuristic methods such as genetic algorithms (GAs) are often employed to obtain high-quality approximate solutions within reasonable time constraints. In this paper, we investigate the application of genetic algorithms to the TSP and examine how algorithm performance is affected by key parameters, particularly initial population size. We conduct experiments using subsets of varying sizes drawn from the USA13509 dataset in TSPLIB. For each subset, benchmark tour lengths are obtained using Google's OR-Tools TSP. Results from small problem instances demonstrate that larger initial population sizes tend to produce higher-quality solutions and require fewer generations to converge, though each generation is more computationally expensive. These findings highlight important trade-offs between solution quality, convergence speed, and resource requirements when applying genetic algorithms to problems like the TSP.

Introduction

One of the most famous problems in computational complexity theory is the traveling salesperson problem (TSP). The TSP is an example of a routing problem, which is a very common type of real-world optimization problem ("Vehicle Routing", 2025). The goal of the traveling salesperson problem is, given a list of locations for a salesperson to visit, construct the shortest possible route for them which visits every location and which starts and ends at the same point. This means, except for the start/endpoint, the salesperson should visit every location on the list exactly once.

Finding the guaranteed optimal solution for the traveling salesperson problem (especially for large instances), can be extremely computationally expensive. However, if we only need to find a reasonably good (but probably not optimal) solution within a relatively short length of time, we might opt to use a discrete search method called a genetic algorithm. Genetic algorithms (Gas) are a heuristic method that is inspired by natural selection. Candidate solutions are conceptualized as organisms, and the fittest

organisms (i.e., the highest quality solutions) will be the most likely to survive and pass on their advantageous characteristics to subsequent generations.

These interesting algorithms "can be applied to any problem that has these two characteristics: (i) a solution can be expressed as a string, and (ii) a value representing the worth of the string can be calculated" (Chenneck, 2020). The first step of applying a genetic algorithm to a problem is generating an initial population, i.e., a set of random candidate solutions. Next, some subset of these organisms would be selected to breed with one another. When two organisms from the mating pool breed with one another, they will generate a child organism which contains information from both parents. This stage within the genetic algorithm is called the crossover operation, and there are a variety of ways that we can tailor this operation to our problem (Chenneck, 2020). Next, some proportion of the newly generated organisms will undergo a random mutation (i.e., a change to their genome). Finally, we may optionally get rid of some of the less fit parents and/or offspring, then the breeding and subsequent operations are repeated for the desired number of generations or until our stopping condition is satisfied.

Background

The past several decades have seen the development of a wide range of optimization algorithms inspired by natural and biological phenomena, such as neural networks, simulated annealing, and evolutionary computation (Larrañaga et al., 1999). Genetic algorithms, which are an example of evolutionary computation, were developed by John Holland in the 1970s at the University of Michigan. Holland introduced genetic algorithms as "a method for moving from one population of 'chromosomes' (e.g., strings of ones and zeros, or 'bits') to a new population by using a kind of 'natural selection' together with the genetics−inspired operators of crossover, mutation, and inversion" (Mitchell, 1998b).

When applying genetic algorithms to the traveling salesperson problem, there several different ways that we can choose to represent a tour, including the binary, path, adjacency, ordinal, and matrix representations. (Furthermore, there are a wide variety of crossover and mutation operators that have been devised for use with different representations) (Larrañaga et al., 1999). For simplicity, in this paper, we will use a slightly modified version of the path representation, which means we can conceptualize each genome as a sequence of destinations. We will select a specified number of organisms to breed in each generation using roulette wheel selection. This reproduction operator involves assigning each organism "a slice of a circular 'roulette wheel,' the size of the slice being proportional to the individual's fitness," and then spinning the wheel a set number of times (Mitchell, 1998b). This means we are more likely to select fit organisms to reproduce, but if we are unlucky, it is possible that we could select poor quality solutions to breed.

Once we have a set of organisms to breed, we will use the partially-mapped crossover operator which allows offspring to inherit genetic information from both parent organisms while preventing the generation of infeasible solutions (Chenneck, 2020). After performing the crossover operation, some subset of the organisms (i.e., candidate TSP solutions) will undergo a random mutation. The newly generated solutions will replace the solutions from the previous generation (however, if we implement a concept called elitism, we can "[force] the GA to retain some number of the best individuals at each generation") (Mitchell, 1998b). We will continue breeding our organisms until we meet satisfy our stopping criterion (in our model, this happens when we complete a pre-specified number of generations).

Problem Formulation

We plan to analyze the behavior of genetic algorithms when we vary parameters like the tour length/number of nodes, the initial population sizes, and the number of solutions carried over between one generation and the next. We have selected a dataset called USA 13509 from TSPLIB that consists of all 13,509 cities in the continental United States which had a population of at least 500 when the dataset was originally compiled. We have selected subsets of varying sizes from this dataset, computed distance matrices, and used Google's OR-Tools TSP solver on each subset to obtain benchmark tour lengths which we can use to evaluate the quality of the solutions we get from the genetic algorithms.

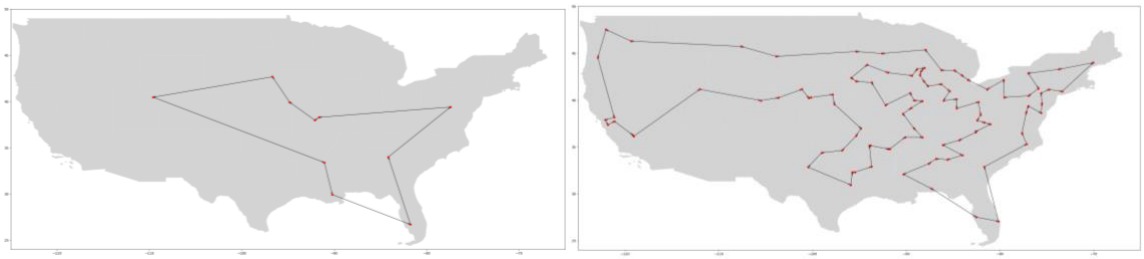


Figure 1: Map showing best guess obtained from Google's OR-Tools TSP solver for tours containing 10 and 100 nodes

Results

First, we will discuss what we found when we varied the size of our initial population when we applied genetic algorithms to the tour containing ten nodes. We kept the mutation probability constant (at 1/10), and we set elitism equal to 1 so that we carried the single best solution from a given generation into the next generation. We can see from the data in Table 1 (as well as the plots in Figure 2) that genetic algorithms which had larger initial populations tend to produce higher quality solutions (on average). Additionally, when initial population sizes were larger, it seems to take fewer generations (on average) for the genetic algorithms to reach their best solution.

Initial Pop. Size	Avg. Final Distance (mi)	Avg. # of Generations to Reach Final Distance	% Error (Compared to Best Guess = 4883 mi)
20	~5,573	22.6	14.1%
200	~5,098	22.4	4.40%
2,000	~4,939	18.8	1.15%
20,000	~4,883	10.4	0%

Table 1: Average best solution (and # of generations to reach best solution) for different initial population sizes

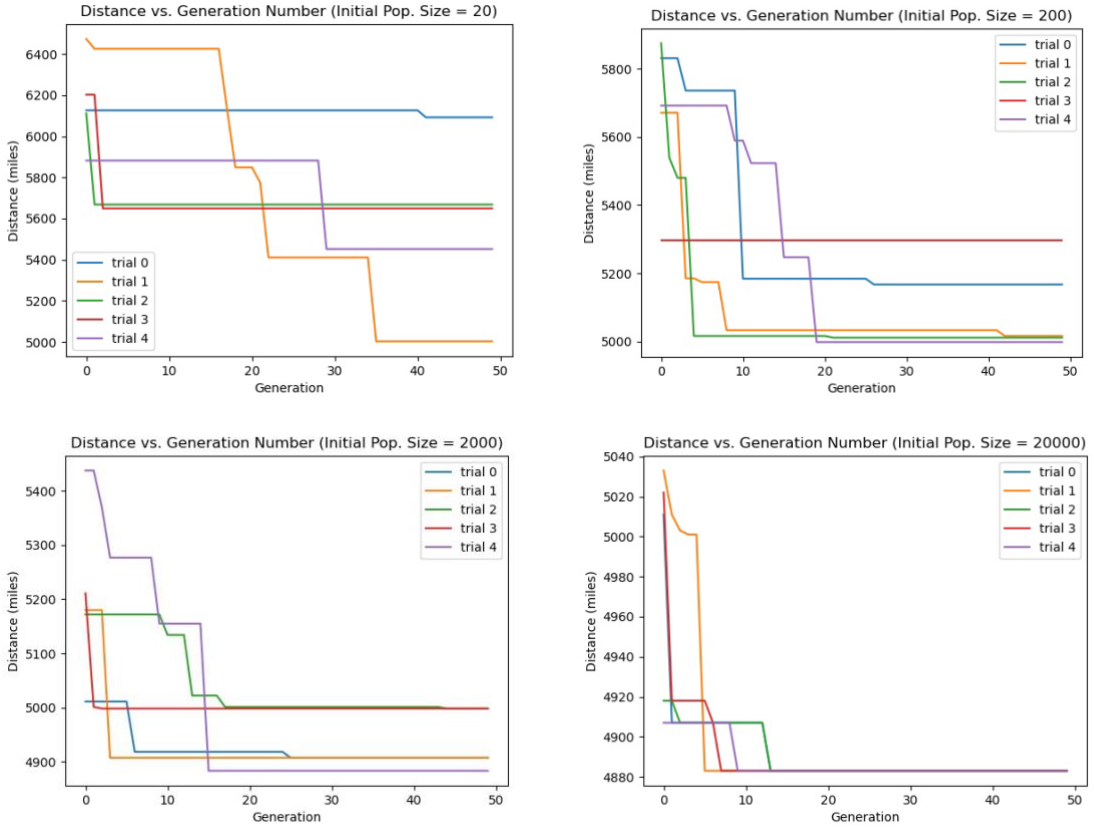


Figure 2: Distance vs. Generation plots for a variety of initial population sizes

Conclusions

We were unable to complete all the experiments that we planned because of how long it took to use genetic algorithms when there were over 1,000 locations within a tour. We also did not get a chance to experiment with varying the elitism values for our genetic algorithms. Despite these unfortunate omissions, we did observe some of the effects that varying the initial population size has on the solutions that genetic algorithms generate. As the population size increased, it took fewer generations (on average) for the genetic algorithms to attain their best solution, and these solutions were higher quality than the solutions we obtained when we had smaller populations. This seems like an important trade-off to consider when working with genetic algorithms. Working with larger populations requires more memory and runtime overall, and while it takes fewer generations to find the best solution, each generation takes longer to run.

Bibliography

Chenneck, J. W. (2020, December 7). Chapter 14: Heuristics for Discrete Search: Genetic Algorithms and Simulated Annealing. In *Practical Optimization: a Gentle Introduction*. essay. Retrieved from https://drive.google.com/file/d/1hHiNpSFt2Kb5pjbMRXSdEpxXXTDEYIvR/view.

Google. (2025, January 22). *Vehicle Routing*. Vehicle Routing | OR-Tools | Google for Developers. https://developers.google.com/optimization/routing

Larrañaga, P., Kuijpers, C., Murga, R. *et al.* Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review* **13**, 129–170 (1999). https://doi.org/10.1023/A:1006529012972

Mitchell, M. (1998b). *An Introduction to Genetic Algorithms*. The MIT Press.

Appendix A: Proposal

Applying Genetic Algorithms to the Traveling Salesperson Problem:
Project Proposal

Abby Butel-Fry

Introduction

One of the most famous problems in computational complexity theory is the traveling salesperson problem (TSP). The TSP is an example of a routing problem, which is a very common type of real-world optimization problem ("Vehicle Routing", 2025). The goal of the traveling salesperson problem is, given a list of locations for a salesperson to visit, construct the shortest possible route for them which visits every location and which starts and ends at the same point. This means, except for the start/endpoint, the salesperson should visit every location on the list exactly once.

Finding the guaranteed optimal solution for the traveling salesperson problem (especially for large instances), can be extremely computationally expensive. However, if we only need to find a reasonably good (but probably not optimal) solution within a relatively short length of time, we might opt to use a discrete search method called a genetic algorithm. Genetic algorithms are a heuristic method that is inspired by natural selection. Candidate solutions are conceptualized as organisms, and the fittest organisms (i.e., the highest quality solutions) will be the most likely to survive and pass on their advantageous characteristics to subsequent generations.

These interesting algorithms "can be applied to any problem that has these two characteristics: (i) a solution can be expressed as a string, and (ii) a value representing the worth of the string can be calculated" (Chenneck, 2020). The first step of applying a genetic algorithm to the TSP is generating an initial population, i.e., a set of random lists which include every destination (other than the start/endpoint) exactly once. Next, some subset of these organisms would be selected to breed with one another. When two organisms from the mating pool breed with one another, they will generate a child organism which contains

information from both parents, and which is a feasible solution to the given instance of the TSP. This stage within the genetic algorithm is called the crossover operation, and one approach that is suited to problems like the TSP is the partially matched crossover operation (Chenneck, 2020). Next, some proportion of the newly generated organisms will undergo a random mutation (i.e., a change to their genome). Finally, we may optionally get rid of some of the less fit parents and/or offspring, then the breeding and subsequent operations are repeated for the desired number of generations.

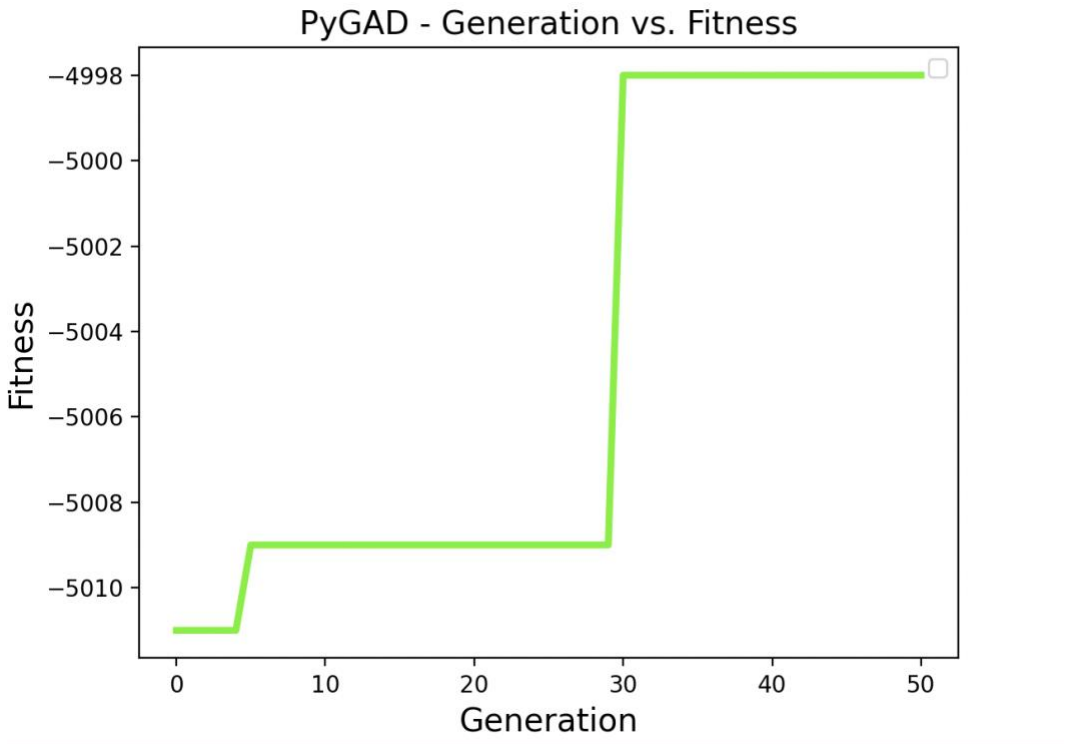
Proposed Work

We will use the USA13509 dataset found on TSPLIB, and we will randomly select subsets of coordinates of different sizes (e.g., 10, 100, 1000, 10000, and 13509). Next, we will use Google's OR-Tools solver to generate a reasonably good guess for each of these instances of the traveling salesperson problem. The solutions generated by this solver can be compared with the results of the genetic algorithms to assess the quality of the results. Additionally, we could opt to include these solutions in the initial populations in an effort to reduce the number of generations needed to generate the fittest possible offspring.

Our overall goal will be to examine how the value of the shortest tour seen so far changes with each generation within a genetic algorithm. In addition to varying the number of distinct nodes in each TSP instance, we will examine the effects of varying the initial population sizes and the number of solutions carried over from one generation to the next.

Sketches of Graphs

Map depicting an example solution for a TSP with ten random nodes selected from USA 13509 dataset



Fitness = -1 x distance in miles

Relevant Journals & Trade Publications for Possible Publication

IAENG International Journal of Applied Mathematics

IEEE Transactions on Evolutionary Computation

International Journal of Computer Science Issues

International Journal of Computational and Experimental Science and Engineering

Resources/References

Chenneck, J. W. (2020, December 7). Chapter 14: Heuristics for Discrete Search: Genetic Algorithms and Simulated Annealing. In *Practical Optimization: a Gentle Introduction*. essay. Retrieved from https://drive.google.com/file/d/1hHiNpSFt2Kb5pjbMRXSdEpxXXTDEYIvR/view.

Google. (2025, January 22). *Vehicle Routing*. Vehicle Routing | OR-Tools | Google for Developers. https://developers.google.com/optimization/routing

Professor Comments:

"In your proposal you said:
In addition to varying the
number of distinct nodes in each TSP instance, we will examine the effects of varying the
initial population sizes, mutation rates, and the number of solutions carried over from one
generation to the next.

I agree that each of those things is interesting, but you don't have to do all of them to get a good grade. Let's say pick 2 of them.
I'll suggest initial population size and number of solutions carried over.
It's good to keep in mind that there might be a tradeoff between time spent on each generation, and amount of improvement per generation. I hesitate to say "time how long each generation takes" because timing runtimes on a computer gets pretty finicky."

Appendix B: Guesses Obtained from Google's OR-Tools

The table below contains the results generated by Google's OR-Tools solver when applied to the same TSP instances which we analyzed using genetic algorithms. The first solution strategy was PATH_CHEAPEST_ARC, and the solver was subsequently run on each instance of the problem using the GUIDED_LOCAL_SEARCH metaheuristic. These solutions were initially generated so that they could be included in the initial populations for the genetic algorithms; however, in the first few instances of the problem, the GAs were not able to improve on this solution. This caused the best solution across every generation to stay the same, which was not the desired behavior. Consequently, these solutions are included primarily so that they can be compared to the results that we get from the genetic algorithms to gauge their overall quality.

# Unique Nodes	**PATH_CHEAPEST_ARC (300 second time limit)**	**GUIDED_LOCAL_SEARCH (300 second time limit)**	**Best Guess**
10	4907	4883	4883
100	12811	12683	12683
1000	35999	36169	35999
10000	123457	127770	123457
13509	149022	147201	147201

Appendix C: Google OR-Tools Code

```
"""This file contains code used to compute distance matrices from .tsp
files (with EDGE_WEIGHT_TYPE EUC_2D). It also uses Google's OR-Tools to
solve the TSP.
Source:
https://developers.google.com/optimization/routing/tsp#complete_programs

This code will be used to get a baseline for what reasonable solutions
may look like in order to assess the quality of the solutions that the
genetic algorithm code generates.
"""
from os import getcwd, path
from geopy import distance
import numpy as np

from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp

SEED = 560
rng = np.random.default_rng(seed=SEED)

def read_tsp_file(fname, num_header_rows, num_nodes_file, num_nodes_sample):
    """Read coordinates from .tsp file (downloaded from TSPLIB)

    Parameters
    ----------
    fname : str
        The name of the .tsp file to be parsed.
    num_header_tows : int
        The number of rows at the top of the .tsp file which contain
        text-based metadata.
```

```
    num_nodes_file : int
        The total number of entries (i.e. node names/numbers and
        coordinate pairs) in the .tsp file.
    num_nodes_sample : int
        The number of entries that should be randomly selected from the
        file and returned.

    Returns
    -------
    np.ndarray
        Array where each entry is a tuple containing the name/number,
        latitude, and longitude for a given node.
    """
    coordinates = np.loadtxt(fname,
                             dtype=np.dtype({"names": ["id", "lat", "lon"],
                                             "formats": ["u4", "d", "d"]}),
                             skiprows=num_header_rows,
                             ndmin=1,
                             max_rows=num_nodes_file)

    # scale coordinates to get valid lat/lon values (decimal degrees)
    for index, _ in enumerate(coordinates):
        coordinates[index]["lat"] *= 1e-4
        coordinates[index]["lon"] *= 1e-4

    random_points = rng.choice(coordinates, size=num_nodes_sample,
                               replace=False)
    return np.reshape(random_points, (num_nodes_sample,))


def compute_dist_matrix(coordinate_array):
    """Create array of distances between every pair in coordinate_array
```

```
    In this 2-d array, the entry in the i-th row and j-th column will
    correspond to the distance (in miles) between the i-th and j-th
    nodes in coordinate_array. These distances will be rounded to the
    nearest integer because Google's OR-Tools routing solver expects
    integer data.

    @note
    http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/TSPFAQ.html
    https://stackoverflow.com/questions/49753792/calculating-distances-in-tsplib
    """
    num_nodes = len(coordinate_array)
    distances = np.empty(shape=(num_nodes, num_nodes), dtype=np.uint32)

    for i in range(num_nodes):
        node_i_coords = (coordinate_array[i]["lat"], coordinate_array[i]["lon"])

        # since the distances are symmetric, only need to compute
        # distances on/above the diagonal to populate entire matrix
        for j in range(i, num_nodes):
            node_j_coords = (coordinate_array[j]["lat"], coordinate_array[j]["lon"])

            dist = distance.great_circle(node_i_coords, node_j_coords).miles
            rounded_dist = round(dist)

            distances[i][j] = rounded_dist
            distances[j][i] = rounded_dist

    return distances


def create_data_model(fname):
```

```
    """Stores the data for the problem.
    """
    data = {}
    data["distance_matrix"] = np.load(fname, allow_pickle=True)
    data["num_vehicles"] = 1
    data["depot"] = 0 # index of start/end city
    return data


def print_solution(manager, routing, solution):
    """Prints solution on console."""
    print(f"Objective: {solution.ObjectiveValue()} miles")
    index = routing.Start(0)
    plan_output = "Route for vehicle 0:\n"
    route_distance = 0
    while not routing.IsEnd(index):
        plan_output += f" {manager.IndexToNode(index)} ->"
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.GetArcCostForVehicle(previous_index, index, 0)
    plan_output += f" {manager.IndexToNode(index)}\n"
    plan_output += f"Route distance: {route_distance} miles\n"
    print(plan_output)


def get_routes(solution, routing, manager):
    """Get vehicle routes from a solution and store them in an array."""
    # Get vehicle routes and store them in a two dimensional array whose
    # i,j entry is the jth location visited by vehicle i along its route.
    routes = []
    for route_nbr in range(routing.vehicles()):
        index = routing.Start(route_nbr)
```

```
        route = [manager.IndexToNode(index)]
        while not routing.IsEnd(index):
            index = solution.Value(routing.NextVar(index))
            route.append(manager.IndexToNode(index))
        routes.append(route)
    return routes


def main():
    """Entry point of the program."""
    sample_sizes = [10, 100, 1000, 10000, 13509]
    parent_dir = path.dirname(getcwd())
    tsp_fname = f"{parent_dir}/usa13509.tsp"
    data_dir = f"{parent_dir}/data"
    time_lim_sec = 300
    manager = None

    #*************************************************************************#
    #         LOAD COORDINATES; LOAD OR COMOPUTE DISTANCE MATRICES          #
    #*************************************************************************#

    for size in sample_sizes:
        coord_fname = f"{data_dir}/usa13509_coords_{size}_nodes_{SEED}.npy"
        dist_mat_fname = f"{data_dir}/usa13509_dist_matrix_mi_{size}_nodes_{SEED}.npy"

        # first, check whether a .npy file containing these coordinates exists
        if path.exists(coord_fname) and path.getsize(coord_fname) > 0:
            coords = np.load(coord_fname, allow_pickle=True)
        else: # if it doesn't exist, create it
            coords = read_tsp_file(tsp_fname, num_header_rows=9,
                                   num_nodes_file=13509,
                                   num_nodes_sample=size)
```

```
    np.save(coord_fname, coords)

# next, determine whether a file containing a distance matrix exists
# for this set of coordinates
if path.exists(dist_mat_fname) and path.getsize(dist_mat_fname) > 0:
    dist_matrix = np.load(dist_mat_fname, allow_pickle=True)
else: # if it doesn't exist, create it
    dist_matrix = compute_dist_matrix(coords)
    np.save(dist_mat_fname, dist_matrix)

#*************************************************************#
#         FOR CURRENT SET OF POINTS, RUN THE SOLVER         #
#*************************************************************#
# Instantiate the data problem.
data = create_data_model(dist_mat_fname)

# Create the routing index manager.
manager = pywrapcp.RoutingIndexManager(
len(data["distance_matrix"]), data["num_vehicles"], data["depot"]
)

# Create Routing Model.
routing = pywrapcp.RoutingModel(manager)

def distance_callback(from_index, to_index):
    """Returns the distance between the two nodes."""
    # Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data["distance_matrix"][from_node][to_node]
```

```
    transit_callback_index = routing.RegisterTransitCallback(distance_callback)

    # Define cost of each arc.
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # Setting first solution heuristic.
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    )
    search_parameters.time_limit.seconds = time_lim_sec


    # Solve the problem.
    solution = routing.SolveWithParameters(search_parameters)

    # Print solution on console.
    if solution:
        print_solution(manager, routing, solution)
        first_route = np.array(get_routes(solution, routing, manager)[0])
        first_distance = np.array(solution.ObjectiveValue())
        first_guess = np.empty(2, dtype=object)
        first_guess[0] = first_route.copy()
        first_guess[1] = first_distance.copy()
        guess_fname =
f"{data_dir}/usa13509_ortools_guess_{size}_nodes_{SEED}_PATH_CHEAPEST_ARC.npy"
        np.save(guess_fname, first_guess)


    # Try guided local search strategy.
    search_parameters.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH)
```

```
        search_parameters.time_limit.seconds = time_lim_sec
        #search_parameters.log_search = True

         # Solve the problem.
        solution = routing.SolveWithParameters(search_parameters)

        # Print solution on console.
        if solution:
            print_solution(manager, routing, solution)
            new_route = np.array(get_routes(solution, routing, manager)[0])
            new_distance = np.array(solution.ObjectiveValue())
            new_guess = np.empty(2, dtype=object)
            new_guess[0] = new_route.copy()
            new_guess[1] = new_distance.copy()
            guess_fname =
f"{data_dir}/usa13509_ortools_guess_{size}_nodes_{SEED}_{time_lim_sec}_sec.npy"
            np.save(guess_fname, new_guess)


if __name__ == "__main__":
    main()
```

Appendix D: GA Code

```
#*********************************************************************#
# Overarching Question: "How does the obj. function value change with
# iteration number on TSPs of various sizes?"
# - want to record length of best tour seen so far at each iteration,
#   graph that as y vs. x values of iteration number, and try to
#   characterize how it changes
#
# Also, want to vary:
#   - number of cities
#   - population size
#   - number of solutions carried over between generations
#*********************************************************************#


from os import getcwd, path
from sys import maxsize
```

```
import numpy as np
import pandas as pd
import pygad

np.set_printoptions(threshold=maxsize)
SEED = 560
rng = np.random.default_rng(seed=SEED)


def generate_initial_pop(initial_pop_size, num_unique_nodes, guess_fname):
    guess_length = num_unique_nodes + 1
    initial_pop = np.empty((initial_pop_size, guess_length),
                           dtype=np.uint32)

    if guess_fname != None:
        best_guess = np.load(guess_fname, allow_pickle=True)[0]
        initial_pop[0] = np.copy(best_guess)
        next_guess_index = 1
    else:
        next_guess_index = 0



    for guess_index in range(next_guess_index, initial_pop_size):
        nodes = np.arange(num_unique_nodes)
        current_guess = rng.choice(nodes, size=num_unique_nodes, replace=False)
        endpoint = np.copy(current_guess[0])
        current_guess = np.append(current_guess, endpoint)
        initial_pop[guess_index] = np.copy(current_guess)

        #initial_pop[guess_index][0:num_unique_nodes] = np.copy(current_guess)
        #initial_pop[guess_index][num_unique_nodes] = np.copy(endpoint)
        #for gene_index in range(len(best_guess)):
```

```
        #    initial_pop[guess_index][gene_index] = current_guess[gene_index]
    #print(initial_pop)
    return initial_pop


def fitness_function(ga_instance, solution, solution_index):
    """Evaluate objective function, impose constraints

    The objective function that we are trying to minimize is total
    distance travelled. However, the fitness function is required to
    return high values for more optimal solutions. The constraints are
    that the path must start and end at the same node, and every other
    city must be visited exactly once.
    """
    #print(f"fitness function params: solution = {solution}")
    #print(f"fitness function params: solution_index = {solution_index}")
    # ensure starting point and ending point are the same
    #print(solution.shape)
    if solution[0] != solution[-1]:
        """print(f"ERROR")
        print(f"{solution_index}:\t{solution}")"""
        print("INFEASIBLE SOL")
        print(f"{solution_index}:\t{solution}")
        return float("-inf")

    # next, check that all other nodes are visited once
    remaining_nodes = solution[1:-1]
    unique_nodes = np.unique(remaining_nodes)

    if len(remaining_nodes) != len(unique_nodes):
        """print(f"ERROR")
        print(f"{solution_index}:\t{solution}")"""
```

```
        print("INFEASIBLE SOL")
        print(f"{solution_index}:\t{solution}")
        return float("-inf")

    # at this point, solution is feasible; evaluate total (scaled) distance
    distance = 0
    current_index = 0
    next_index = current_index + 1

    while next_index < len(solution):
        current_node = int(solution[current_index])
        next_node = int(solution[next_index])
        distance += dist_matrix[current_node][next_node]
        current_index += 1
        next_index += 1

    # PyGAD maximizes fitness; returning this value ensures that shorter
    # paths yield greater fitness values than longer paths without having
    # to keep track of scaling factors, etc.
    """print(f"{solution_index}:\t{solution}\t\t{distance}")"""
    return -1 * distance


def crossover_func(parents, offspring_size, ga_instance):
#def crossover_func(parents, offspring_size):
    """
    Decided to use partially matched crossover after reading pg. 6-7 in
    chapter 14 of Optimization: a Gentle Introduction. This crossover
    operator will prevent offspring from containing duplicate values
    which violate the constraints of the TSP. Also referenced the
    description here:
```

```
https://en.wikipedia.org/wiki/Crossover_(evolutionary_algorithm)#Partially_mapped_crossover_(PMX)
    """
    offspring = np.empty(offspring_size, dtype=np.uint32)
    num_genes = offspring_size[1]
    for offspring_index in range(offspring_size[0]):
        parent_indices = rng.integers(0, len(parents), 2)
        #@note trying this to make sure that the same element isn't chosen twice
        #@note not sure if I should worry about case where different parents happen to be identical...probably not?
        while parent_indices[0] == parent_indices[1]:
            parent_indices = rng.integers(0, len(parents), 2)
        #print(f"PARENT INDICES: {parent_indices}")
        p0 = parents[parent_indices[0]]
        #print(f"p0:\t{p0}")
        p1 = parents[parent_indices[1]]
        #print(f"p1:\t{p1}")
        tmp_offspring = [-1] * num_genes
        #print(tmp_offspring)

        slice_start = rng.integers(0, num_genes)
        slice_end = rng.integers(slice_start, num_genes)

        #print("slice start index: " + str(slice_start))
        #print("slice end index: " + str(slice_end))

        # copy initial segment from p0
        for i in range(slice_start, slice_end + 1):
            tmp_offspring[i] = p0[i]
```

```
    # ensure that if the start/end is populated, it matches on both ends
    if tmp_offspring[0] != -1:
        tmp_offspring[-1] = tmp_offspring[0]
    elif tmp_offspring[-1] != -1:
        tmp_offspring[0] = tmp_offspring[-1]

    #print(tmp_offspring)

    # copy genes from p1 that which occur in this segment but which are
    # not yet included in the offspring
    for i in range(slice_start, slice_end + 1):
        # this is m in the description of the algorithm on Wikipedia
        curr_p1_gene = p1[i]

        if curr_p1_gene in tmp_offspring:
            #print(str(curr_p1_gene) + " already occurs in offspring")
            continue

        else:
            # this is n in the description of the algorithm on Wikipedia
            corresponding_offspring_gene = tmp_offspring[i]
            indices = np.where(p1==corresponding_offspring_gene)[0]
            index = indices[0]

            # in the offspring, copy m into the position where n is in p1
            # if that position is not already occupied
            if tmp_offspring[index] == -1:
                tmp_offspring[index] = curr_p1_gene
                # ensure start/endpoints match
                if index == 0 or index == num_genes - 1:
                    tmp_offspring[(num_genes - 1) - index] = curr_p1_gene
```

```
            else:
                # the place where n is in p1 is occupied in offspring
                new_index = np.where(p1==tmp_offspring[index])[0][0]
                tmp_offspring[new_index] = curr_p1_gene
                # ensure start/endpoints match
                if new_index == 0 or new_index == num_genes - 1:
                    tmp_offspring[(num_genes - 1) - new_index] = curr_p1_gene
        #print(tmp_offspring)
        # fill in remaining genes from p1 which have not appeared in offspring
        unused_p1_genes = np.array([gene for gene in p1 \
                                    if gene not in tmp_offspring])
        #print(unused_p1_genes)
        for gene in unused_p1_genes:
            if -1 in tmp_offspring:
                first_unused_index = tmp_offspring.index(-1)
                tmp_offspring[first_unused_index] = gene
                if first_unused_index == 0:
                    tmp_offspring[-1] = gene
            else:
                break
        #print(tmp_offspring)

        tmp_offspring = np.array(tmp_offspring)
        #for i in range(num_genes):
        #    offspring[offspring_index][i] = tmp_offspring[i]
        #print(f"offspring:\t{tmp_offspring}")
        offspring[offspring_index] = np.ndarray.copy(tmp_offspring)

    return offspring


def mutation_func(offspring, ga_instance):
```

```
"""For the time being, rather than selecting genes to swap, if an
offspring is to undergo mutation, I will select a single (random)
gene in that offspring, remove it from the offspring, and insert it
into a random index of that offspring. After this operation, I will
ensure that the offspring is still a feasible solution.
"""
#print(offspring)
num_genes = len(offspring[0])

for i in range(len(offspring)):
    tmp_offspring = np.ndarray.copy(offspring[i])
    if rng.random() < ga_instance.mutation_probability:
        #print(offspring[i])
        indices = rng.integers(0, num_genes, 2)
        initial_index = indices[0]
        final_index = indices[1]
        affected_gene = np.copy(tmp_offspring[initial_index])
        tmp_offspring = np.delete(tmp_offspring, initial_index)

        if initial_index == 0:
            tmp_offspring[-1] = tmp_offspring[0]
        elif initial_index == num_genes - 1:
            tmp_offspring[0] = tmp_offspring[-1]

        tmp_offspring = np.insert(tmp_offspring, final_index,
                                  affected_gene)

        if final_index == 0:
            tmp_offspring[-1] = tmp_offspring[0]
        elif final_index == num_genes - 1:
            tmp_offspring[0] = tmp_offspring[-1]
```

```
            #print(offspring)
            offspring[i] = np.ndarray.copy(tmp_offspring)
            #print(offspring[i])

    #print(offspring)
    return offspring

def on_gen(ga_instance):
    print(f"Generation : {ga_instance.generations_completed}", flush=True)
    print(f"Fitness of the best solution : {ga_instance.best_solution()[1]}", flush=True)


def main():
    """Entrypoint for program"""
    global dist_matrix
    #global scaling_factor
    #global rng
    #sample_sizes = [10, 100, 1000, 10000, 13509]
    sample_sizes = [10]
    initial_pop_sizes = [20, 200, 2000, 20000]
    num_trials = 5

    parent_dir = path.dirname(getcwd())
    data_dir = f"{parent_dir}/data"
    dataset_name = "usa13509"
    unit = "mi"
    extension = ".npy"
    ortools_solver_time_lim = 300

    for num_nodes in sample_sizes:
        #print(f"SAMPLE SIZE = {num_nodes}")
        dist_matrix_fname = f"{data_dir}/{dataset_name}_dist_matrix_{unit}_"\
```

```
                    f"{num_nodes}_nodes_{SEED}{extension}"

    dist_matrix = np.load(dist_matrix_fname, allow_pickle=True)
    #print(dist_matrix)

    initial_pop_size = 2000 #num_nodes * 100
    num_generations = 50
    # top 10% of organisms will mate
    num_parents_mating = round(0.10 * initial_pop_size)
    parent_selection_type = "rws" # roulette wheel selection
    #keep_parents = -1 # keep default (-1) for now; has no effect if keep_elitism = 1
    # keep n best solutions in the next generation
    keep_elitism = 1 # keep default (1) for now
    mutation_probability = 0.1
    save_best_solutions = False
    save_solutions = False

    guess_fname =
f"{data_dir}/{dataset_name}_ortools_best_guess_{num_nodes}_nodes_{SEED}.npy"
    distance_array = np.empty((num_trials, num_generations), dtype=np.uint32)
    distance_csv =
f"{data_dir}/{dataset_name}_{num_nodes}_nodes_{initial_pop_size}_init_pop_1e-
1_mut_prob.csv"

    for trial_index in range(num_trials):
        #initial_pop = generate_initial_pop(initial_pop_size, num_nodes, guess_fname)
        initial_pop = generate_initial_pop(initial_pop_size, num_nodes, None)
        #print(f"initial_pop shape: {initial_pop.shape}")
        ga_instance = pygad.GA(num_generations=num_generations,
                    num_parents_mating=num_parents_mating,
                    fitness_func=fitness_function,
```

```
                    initial_population=initial_pop,
                    gene_type=np.uint32,
                    parent_selection_type=parent_selection_type,
                    #keep_parents=keep_parents,
                    keep_elitism=keep_elitism,
                    crossover_type=crossover_func,
                    mutation_type=mutation_func,
                    mutation_probability=mutation_probability,
                    on_generation=on_gen,
                    save_best_solutions=save_best_solutions,
                    save_solutions=save_solutions)

        ga_instance.run()

        print(ga_instance.best_solution())
        approx_distance = ga_instance.best_solution()[1] * -1
        #print(f"distance: {approx_distance}")
        #ga_instance.plot_fitness()
        #print(f"generation where best fitness spotted:
{ga_instance.best_solution_generation}")

        for sol_index in range(num_generations):
            distance_array[trial_index][sol_index] = -1 *
ga_instance.best_solutions_fitness[sol_index]
        #print(distance_array[trial_index])

        print(ga_instance.best_solutions_fitness)
    #print(distance_array)
    df = pd.DataFrame(distance_array)
    df.to_csv(distance_csv)
```

```
if __name__ == "__main__":
    main()
```