Solving the Traveling Salesman Problem with Genetic Algorithms

Daniel Kearney

April 10, 2018

Contents

0.1 Summary . . . 2

0.2 History of the Genetic Algorithm . . . 2

0.3 The Traveling Salesman Problem . . . 2

0.4 Genetic Algorithm Function . . . 2

0.5 Genetic Algorithm Implementation in Python . . . 2

0.5.1 The Basics . . . 2

0.5.2 Creating the First Generation . . . 4

0.5.3 Selection . . . 4

0.5.4 Crossing Over . . . 5

0.5.5 Mutation . . . 5

0.6 Complexity Analysis . . . 6

0.6.1 Time Complexity . . . 6

0.6.2 Setup . . . 6

0.6.3 Invariant . . . 6

0.6.4 Final Step . . . 7

0.6.5 Synthesis . . . 7

0.7 Results . . . 7

0.7.1 Small Dataset . . . 8

0.7.2 Large Dataset . . . 8

0.7.3 Large Real-world Dataset . . . 8

0.1 Summary

0.2 History of the Genetic Algorithm

0.3 The Traveling Salesman Problem

0.4 Genetic Algorithm Function

The genetic algorithm works by mimicking the mechanism of chromosomes and genes in evolutionary biology. Individuals with various sets of genes compete with one another, reproducing into the next generation, with the whole population becoming fitter and fitter as the fittest solutions survive, and the less fit ones don't.

The genetic algorithm works by initially creating a random 'generation' of solutions to seed the iterative process. Each individual solution – called a 'chromosome' – has a random solution to the problem at hand.

Once the seed generation is created, the loop begins. The invariant is as follows. First, certain individuals are selected into the next generation, with the fittest solutions being favored. Pairs of these parents are 'crossed over', a process where parts of each solution are absorbed into each other. Finally, a bit of randomness is added by 'mutating' some of the solutions with random adjustments.

The loop has no natural termination point. The loop can terminate after a certain number of generations, or when a desired fitness is reached.

The following pseudocode summarizes how the genetic algorithm works.

```
PROCEDURE genetic_algorithm(problem)
  current_generation = generate_random_generation ( problem )
  for G generations:
      next_generation = select( problem, current_generation )
      cross_over ( next_generation )
      mutate ( next_generation )
      current_generation = next_generation
  return fittest ( current_generation )
```

0.5 Genetic Algorithm Implementation in Python

This section describes the implementation of a genetic algorithm in Python.

0.5.1 The Basics

The traveling salesman problem assumes that each city can be reached from any other city; that is, the graph is fully connected. This graph can be very simply represented as a Python list of vertices where the value is each vertex's XY coordinate. Because every vertex connects to every other, we can implicitly assume that each edge (u, v) exists, and that it's value is the simple Euclidean distance between the vertices. As such, the fully connected graph can be represented as a simple list of XY tuples.

```
FullyConnectedGraph = [(0.1313626194361659, 0.9537494015311131
 (0.6931250094496763, 0.042877414320216856
 (0.17355376046059656, 0.41488071009395744
 (0.47339225625323855, 0.9527742641692806
 (0.399394375477357, 0.6665748105617455
 (0.43170810077728095, 0.13591355449019749
 (0.6351099070806486, 0.5983664092055496
 (0.8957003084360835, 0.656752235127581
 (0.7658498920971735, 0.6465721898932276
 (0.8026020533786447, 0.06444102480533265)]
```

Figure 1: A simplified representation of a random fully connected graph with N=10.

We can visualize this fully connected graph in 2D space, and plot each edge.

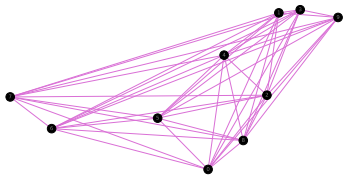


Figure 2: A fully-connected graph with 10 vertices, each placed at random XY coordinates. Each vertex represents a city, and the edge weights are the Euclidean distance between the cities.

Any valid solution to the problem is simply a permutation of the list of vertices. For simplicity, we will number the vertices from 0 to N. For example, for a graph with 10 vertices, any of the following are valid solutions:

```
[2, 3, 4, 8, 9, 6, 1, 5, 7, 0]
[8, 0, 6, 2, 4, 3, 1, 9, 5, 7]
[9, 0, 5, 3, 7, 4, 8, 6, 1, 2]
```

Figure 3: Three valid solutions to the traveling salesman problem with cities numbered 0 to N-1. Each is a permutation of the array 0..10.

This is a visualization of a single solution.

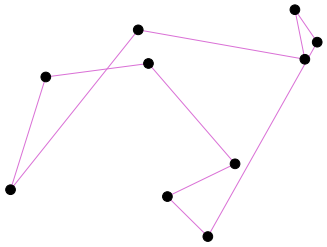


Figure 4: An instance of a possible solution to the traveling salesman problem with N=10.

To compute the fitness of a solution, the total distance of the journey is measured, by walking through

the path one edge at a time, eventually looping back to the starting point.

```
def compute_fitness(self):
  total_dist = 0
  # Compute the fitness, edge by edge
  for i in range(len(self)):
    edge_start = self.get(i)
    edge_end = self.get((i+1)%len(self))
    total_dist += self.graph.dist_between(edge_start, edge_end)
  self.fitness = total_dist
  return self.fitness
```

0.5.2 Creating the First Generation

The first step in a genetic algorithm is to create a first chromosome, using random solutions. This is straightforward, because a random solution is simply a permutation of the list 0..N. Here, we create a range of numbers sorted from 0 to N, and use the random module to randomly shuffle it. This process is repeated for as many solutions are desired in each chromosome.

```
# Create random solutions to seed the algorithm.
def generate_random_chromosomes(self):
  for i in range(self.num_chromosomes):
    arr = range(self.graph.num_nodes)  # Begin with the solution 0, .. N-1
    random.shuffle(arr)          # Shake it up to make a random solution
    self.current_generation.append(Solution(arr, self.graph))
```

0.5.3 Selection

The goal of selection is to select a group of individuals from the current generation to begin the process of creating the next generation. The process aims to predominantly select the fittest individuals, but allow for less fit individuals to also survive on occasion, to preserve diversity. A good approach for this is a 'tournament' selection. In tournament selection, a random batch of four individuals are chosen from the current generation. The fittest is then placed into the next generation. This is repeated until the next generation is full. Using a small selection group – in this case, only four – ensures diversity while favoring the top quartile of fittest solutions.

```
  # Use a "tournament select" to choose winning parents.
  # Repeatedly pick 4 random elements and put the
  # best one into the next into the next generation.
  def select_next_generation(self):
    self.next_generation = []

    # Compute all of the fitnesses so we can run tournaments.
    self.compute_all_current_gen_fitnesses()

    # Fill the next generation with tournament winners
    while len(self.next_generation) < len(self.current_generation
      # Choose four random solutions
      sample = random.sample(self.current_generation, 4)
      # Pick the winner (lower fitness value is best)
      fittest = min(sample, key=lambda x: x.fitness)
      self.next_generation.append(fittest)
```

0.5.4 Crossing Over

Crossing first groups each of the newly selected next generation into pairs, which I refer to as 'mom' and 'dad'. In crossing over, the algorithm merges a bit of each solution into the other.

We do this by first starting with mom. We take a random chunk of mom's solution and insert it into a new array at the same location. We then insert dad's solution into the new array. This is complex because we aim to retain dad's solution order as much as possible, without duplicating any of the vertices (which is invalid). The procedure cross_merge_array below does this by inserting dad's solution into the new array, element by element, unless that element was already inserted as part of crossing-over. This procedure is repeated for both parents, and then repeated for every pair of parents.

```
# Uses ordered crossover to cross two arrays.
def cross_merge_arr(self, parent1, parent2):

  # First choose two random indices in parent1 to select the crossover section.
  crossover_points = sorted(random.sample(parent1, 2))

  # Extract that section of parent1 and store it in a hashset for quick lookups
  crossover = parent1[crossover_points[0]:crossover_points[1]]
  crossover_set = set(crossover)

  # Now push the crossover into an empty output array at the same indices
  out_arr = [None] * self.graph.num_nodes
  out_arr[crossover_points[0]:crossover_points[1]] = crossover

  # Fill in the rest of output array from parent2, maintaining parent2's order.
  # Iterate over parent2's nodes.
  # Attempt to insert each node into the output array.
  # If the slot is empty, AND the node isn't in the crossover set, insert into output array.
  # Otherwise, try to place it in the next slot.
  placement_index = 0
  for item in parent2:
    while placement_index < self.graph.num_nodes and out_arr[placement_index] != None:
      placement_index += 1
    if item not in crossover_set:
      out_arr[placement_index] = item
      placement_index += 1
  return out_arr
```

0.5.5 Mutation

The final step of the genetic algorithm is mutation. Mutation is intended to create a bit of randomness to help the solutions occasionally find brand new paths. The key constraint on mutation is that it must still yield an acceptable solution, that is, a solution where each vertex is visited exactly once. A good way to ensure that is to simply swap two vertices in the solution.

```
# Mutate chromosomes by two swapping random elements at the mutation rate.
# Swapping maintains the solution integrity
# (ensuring no node is visited twice or not visited).
def mutate(self):
  for solution in self.next_generation:
  # Each chromosome has a chance of mutation
    if random.random() < self.mutation_rate:
      index_a = random.randrange(0, self.graph.num_nodes)
      index_b = random.randrange(0, self.graph.num_nodes)
      solution.swap(index_a, index_b)
```

0.6 Complexity Analysis

0.6.1 Time Complexity

The genetic algorithm is interesting because it is a heuristics-based algorithm, that makes no promises to perfectly solve the problem at hand. Largely speaking, the genetic algorithm first requires a setup (creating the first generation), an invariant (selection, crossing over, mutation), and a final step in determining the best chromosome in the final step.

$$T(n) = T(setup) + iterations * T(invariant) + T(finalstep) \tag{1}$$

First, some definitions. These are:

d iterations of the algorithm

N vertices in the graph

c chromosomes

0.6.2 Setup

The setup process requires setting up the first generation of chromosomes. There are c chromosomes, each of which has N elements.

$T(setup) = T(N * c)$

0.6.3 Invariant

The invariant is a bit more complex. The invariant consists of selection, crossing-over, and mutation.

Selection utilizes a tournament select. In my implementation, all of the chromosomes' fiT(N)esses are first computed. Each fiT(N)ess computation requires walking through the entire graph, which is of size N. Then, four chromosomes are selected at random (which takes constant time), and the winner is selected, which also takes constant time since the fiT(N)esses were pre-computed. Therefore:

$T(selection) = T(N * c)$

Crossing over iterates over each chromosome, merging it into its other half. Each merge process takes N iterations, and it is done once per chromosome.

$T(crossing_over) = T(N * c)$

Mutation swaps two elements of each chromosome. Assuming a 100% mutation rate, the swapping process happens c times.

$T(mutation) = T(N)$

0.6.4 Final Step

The final step in the process is to determine the fittest solution. To do so, each solution's fiT(N)ess must be computed, which requires $T(N)$ iterations per chromosomes; then the winner is chosen as the one with the minimum fiT(N)ess value (shortest path) which takes c iterations. Therefore:

$$T(final_step) = T(N * c + c)$$

0.6.5 Synthesis

Putting all of this together:

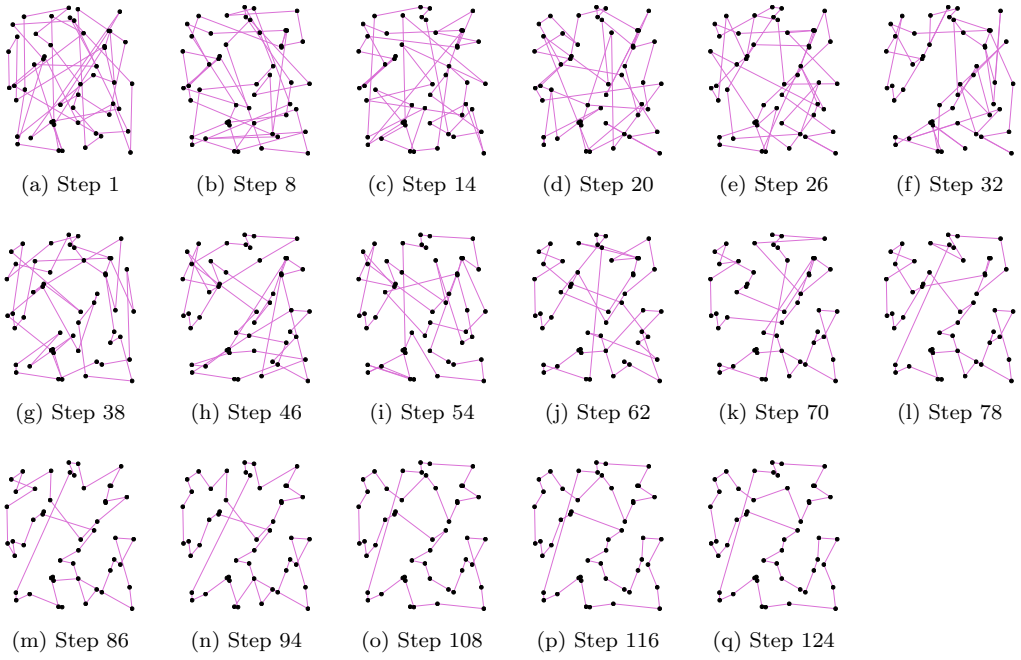
$$\begin{aligned} T(n) &= T(\text{setup}) + \text{iterations} * T(\text{invariant}) + T(\text{final step}) \\ T(n) &= T(N * c) + d * [T(N * c) + T(N * c + c) + T(N)] + T(N * c) \\ T(n) &= \Theta(N * d * c) \end{aligned} \tag{2}$$

Intuitively, this is simple to explain. Fundamentally, the number of chromosomes, the size of the input graph, and the number of iterations all factor into the number of iterations that the algorithm takes.

0.7 Results

0.7.1 50 Vertices

When applied to a random graph with 50 vertices, the genetic algorithm finds an optimized solution after around 100 iterations, with a mutation rate of .3, a crossover rate of .99, and 500 chromosomes.



(a) Step 1 (b) Step 8 (c) Step 14 (d) Step 20 (e) Step 26 (f) Step 32

(g) Step 38 (h) Step 46 (i) Step 54 (j) Step 62 (k) Step 70 (l) Step 78

(m) Step 86 (n) Step 94 (o) Step 108 (p) Step 116 (q) Step 124

Figure 5: The genetic algorithm working on an algorithm with 50 vertices.

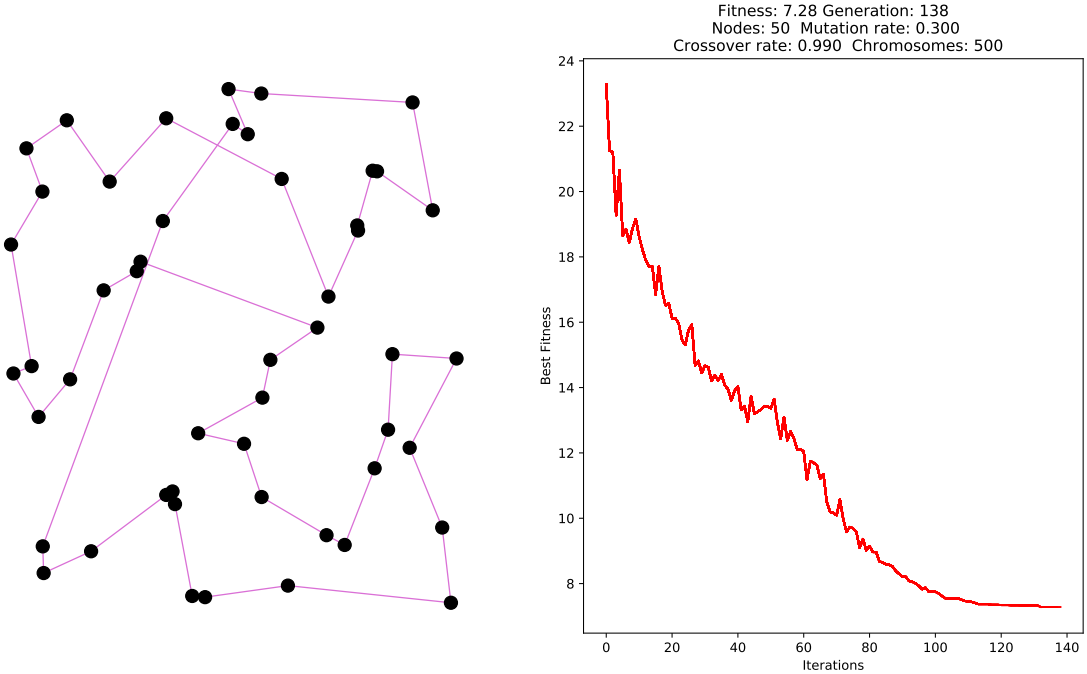


Figure 6: The genetic algorithm applied to a graph of 50 random vertices, with a mutation rate of .3, a crossover rate of .99, and 500 chromosomes.

For this example, the genetic algorithm begins with a random solution with fitness of 23 distance units. After 138 iterations, the fittest solution has 7.28 distance units, a 3x improvements.

0.7.2 100 Verticess

At 100 vertices, the algorithm takes longer to optimize, but is able to optimize from 45 distance units in the first iteration to 10.42 in the 650th generation, an improvement of 4.5x.

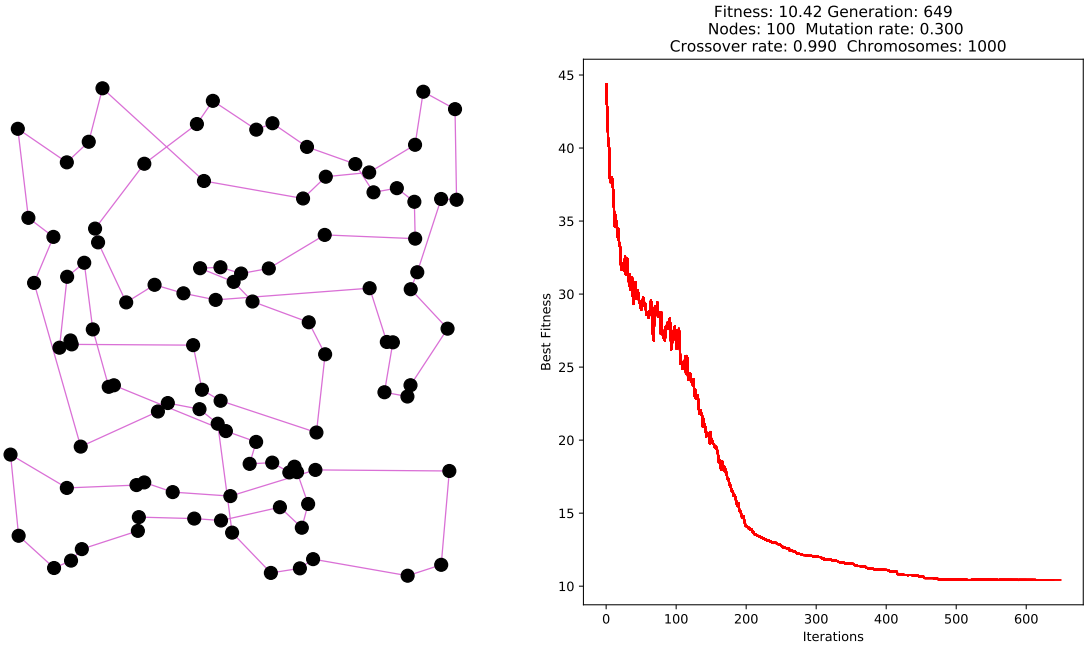


Figure 7: The genetic algorithm applied to a graph of 100 random vertices, with a mutation rate of .3, a crossover rate of .99, and 500 chromosomes.

0.7.3 Real-world Dataset

A great litmus test is a real-world dataset. This dataset is 128 of the largest cities in North America.

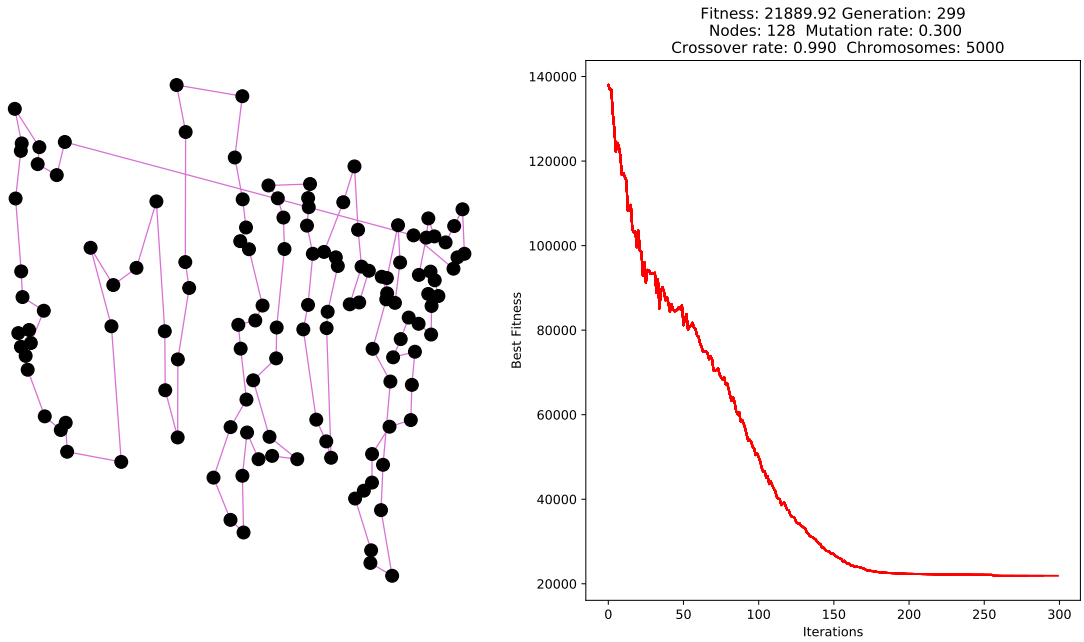


Figure 8: The genetic algorithm applied to an XY projection of 128 major cities in North America.

0.7.4 Small Dataset

0.7.5 Large Dataset

0.7.6 Large Real-world Dataset