Solving the Traveling Salesman (TSP) Problem with a Genetic Algorithm (GA)

Adam D. Byerly
Bradley University, abyerly@mail.bradley.edu

Abstract – **The Traveling Salesman Problem is one of the most studied problems in the class of NP Hard problems, having no deterministic method of solving below superpolynomial time. Genetic Algorithms are a class of meteheuristics for solving NP class problems non-deterministically but in a reasonable and useful amount of time. This paper discusses the development and execution of a genetic algorithm applied to the Traveling Salesman Problem.**

The Traveling Salesman Problem

The Traveling Salesman Problem is one of the most studied problems in the class of NP Hard problems because it is a good analog for a broad group of real world problems including planning, logistics, microchip manufacturing, and DNA sequencing [1].

The TSP, informally articulated, is finding a path (a modified Hamiltonian Circuit) through each city (and returning to the starting city) in a "sales route" such that the path traveled is the shortest possible route for the salesman. To understand the motivation for the problem more concretely, we can visualize a salesman having to make sales calls in San Diego, Los Angeles, New York City, and Albany. The salesman certainly wants to visit both cities in California before or after visiting both cities in New York, and certainly wants to avoid traveling from New York to California to New York and back to California.

Genetic Algorithms

Genetic Algorithms are a class of algorithms subsumed under the broader category of biological algorithms. Biological algorithms in general seek to model biological processes as a means of finding a solution to a problem. GAs specifically seek to do this by modeling natural selection. Natural selection can be viewed as a search process whereby nature is "searching" for the most fit individuals across a large population and across many generations [2].

Terminology. GAs, in modeling how traits are encoded and passed on to successive generations, tend to use terminology consistent with their biological counterparts. **Chromosomes** are encoded candidate solutions to the problem trying to be solved. In nature, chromosomes are either haploid (single) or, more commonly in sexual species diplod (dual). Further there are often multiple of the single or double chromosomes. For example, humans have 23 pairs of chromosomes and thus have multiple diploid chromosomes. In GAs, typically we use a single haploid chromosome to represent the entire individual. In the case of the TSP, a chromosome (and thus an individual) is going to be a valid permutation of the cities visited by the salesman. **Genes** are individual portions of the chromosome that represent either a trait or set of traits. In the case of the TSP, a gene is a city. Finally, a **population** is a collection of individuals in the environment. In the TSP, the population is a collection of chromosomes, and thus possible routes the salesman could take [3].

Genetic Operators

Whereas chromosomes, genes, and populations are the model portion of a GA, genetic operators are the algorithmic portion. There are three common genetic operators used in most GAs: selection, crossover, and mutation.

Selection. Selection is the operator responsible for deciding what chromosomes are going to survive and/or reproduce into the subsequent generation. There is a large variety of selection methods in the published literature, including the original "Roulette Wheel" method created by John Holland [4] and used to give greater chances of reproducing to the most fit. Also in this list is Sigma Scaling, Elitism, Boltzmann Selection, Rank Selection, Tournament Selection and Steady-State Selection [5]. The method chosen in my implementation was a combination of Elitism and Rank Selection. During the selection operation, the chromosomes are ranked, and a parameterized percentage of those chromosomes are allowed to survive and reproduce into the next generation.

Crossover. Crossover is the operator responsible for creating new chromosomes from those chromosomes selected to reproduce. As is the case with the other genetic operators, there are numerous methods documented in GA literature including the most common methods of simple single and two point crossover. These simple crossover methods are aimed at bit string encoded chromosomes and are not suitable for permutation encoded chromosomes like are used in the TSP. This is because the permutation encoding requires one and only one instance of every gene to be present in the chromosome and simple crossover will create both duplicate genes and missing genes. Historically, the two most common operations that addressed this problem were cycle crossover (CX) and partially matched crossover (PMX). However, a more robust (though more complex) operation called the Edge Recombination Operator (ERO) has been discovered. [6]

The ERO algorithm starts by enumerating the two (or potentially more) parent chromosomes and creating an adjacency matrix (referred to in the following algorithm as a list of neighbor lists) that for each unique gene, stores those genes that are adjacent in the parent chromosomes in a vector (uniquely). Then, the following algorithm is performed to create a new child chromosome [7]:

```
Let K be the empty list
Let N be the first node of a random parent.

While Length(K) < Length(Parent):
    K := K, N (append N to K)
    Remove N from all neighbor lists

    If N's neighbor list is non-empty
       then let N* be the neighbor of N with
       the fewest neighbors in its list (or a
       random one, should there be multiple)
    else let N* be a randomly chosen node that
       is not in K

    N := N*
```

Mutation. Mutation is the operator responsible for injecting diversity into the gene pool. It is analogous to mutation in biological evolution. Without occasionally randomly mutating some of the genes the algorithm runs into danger of getting "stuck" in a local optimum. If too much mutation is used, the algorithm essentially reverts to a primitive random search. The classic implementation of mutation is flipping bits in a bit string encoded chromosome. In my algorithm, since we are using permutation encoding, the mutation operator takes the form of swapping the position of a random pair of genes in the chromosome.

GENETIC PARAMETERS

Depending on the forms taken by the genetic operators, a variety of parameters emerge. Given my choices for selection, crossover, and mutation the following parameters emerged.

Survival Rate. This real valued parameter in the range of 0-1 represents the percentage of a generation's population that will survive to the next generation and reproduce. To make comparison between chromosome sizes meaningful, all experiments performed used a survival rate of 0.5.

Mutation Rates. These two real valued parameters both in the range of 0-1 represent the probability that a chromosome will be selected for mutation and the probability that once selected any gene on the chromosome will mutate. To make comparison between chromosome sizes meaningful, all experiments performed used 0.01 and 0.01 for these parameters.

Population Size. This non-zero natural number represents the number of chromosomes present in the initial and subsequent generations. To make comparison between chromosome sizes meaningful, all experiments performed used a population size of 1000.

Stopping Condition. This parameter can take several forms. In our algorithm it took the simple form of a number of generations. To make comparison between chromosome sizes meaningful, all experiments performed used a stopping condition of 5000 generations.

THE CODE

The code was written in C++11. The only third party code involved was tinyxml2, used to parse the input data set files downloaded from TSPLIB.

Some of the utility code, though authored by me, was pre-existent to this project. Namely additional code used to parse XML, create and manage threads and thread pools, code for logging and other similar miscellaneous functions.

The rest of the code, including all code related to GAs and the TSP was written in its first form for this project.

DATA SETS

Gerhard Reinelt of Universität Heidelberg has compiled a large collection of TSP data sets he calls TSPLIB [8]. All of the data sets have a deterministically computed optimum available. Allowing us to determine the differences between the solutions arrived at by the GA and the actual optimum and thus grants us the ability to compute an error measurement.

For our experiments, we chose 5 datasets of increasing city count: burma14 (14 cities), bays29 (29 cities), eil51 (51 cities), pr76 (76 cities), and kroA100 (100 cities). Note that these data set names are reproduced exactly as they were given on the TSPLIB website.

EXPERIMENTAL RESULTS

We ran five different experiments with the same genetic parameters but each with an increasingly larger gene size. The actual values for the edge costs were not normalized among the data sets for two reasons. One, we wanted to run on data exactly as it came from TSPLIB to make replicating the results or performing additional experiments easier and more meaningful. Additionally, the real measure of success is a comparison between the actual optimum and the value the GA converged on. As this is a percentage, it can be compared across data sets using arbitrary proportionality of edge costs.

Data Set	Gene Size	Optimum
burma14	14	3323
bays29	29	2020
eil51	51	426
pr76	76	108159
kroA100	100	21282

Figure 1: Gene Size and Optimums.

Data Set	Converged At	After Generations	Error
burma14	3323	20	0.000%
bays29	2020	77	0.000%
eil51	431	176	1.102%
pr76	109562	1334	1.280%
kroA100	21812	4661	2.431%

Figure 2: Convergence Results and Error.

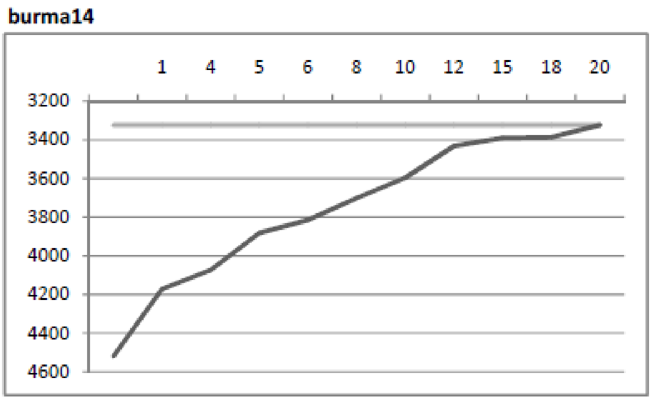

Figure 3: Convergence of burma14 after 20 generations.

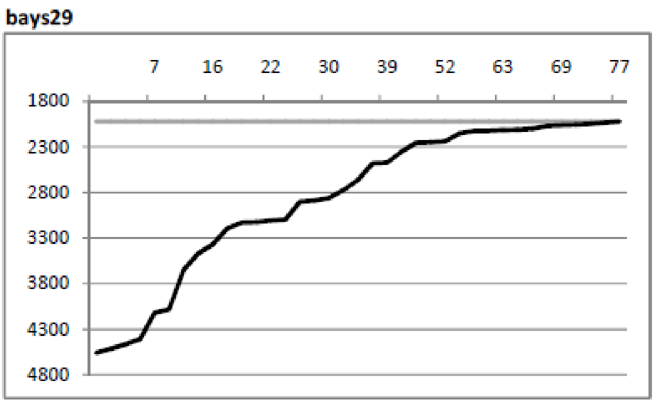

Figure 4: Convergence of bays29 after 77 generations.

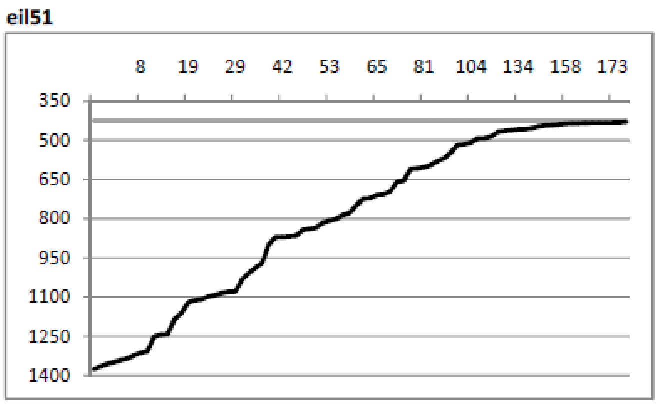

Figure 5: Convergence of eil51 after 176 generations.

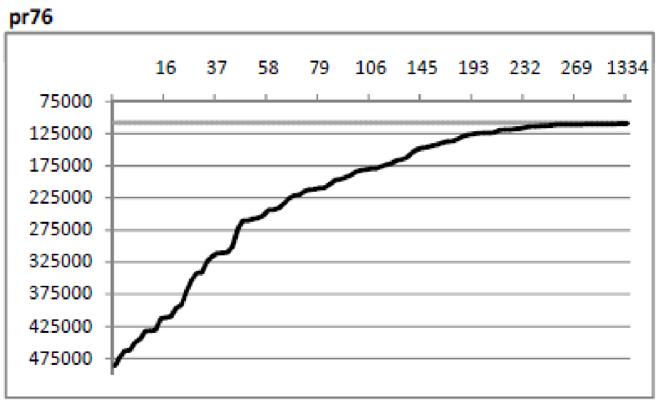

Figure 6: Convergence of pr76 after 1334 generations.

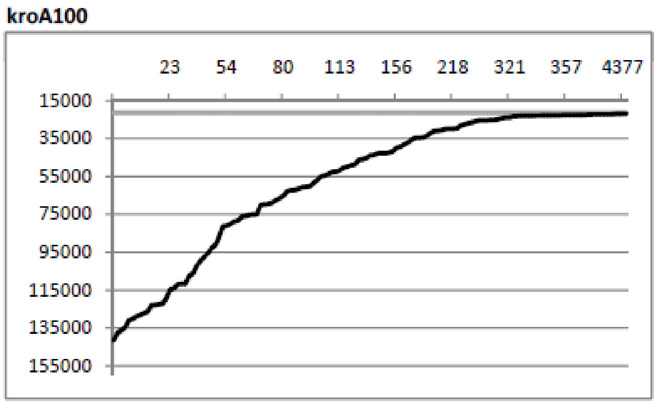

Figure 7: Convergence of kroA100 after 4661 generations.

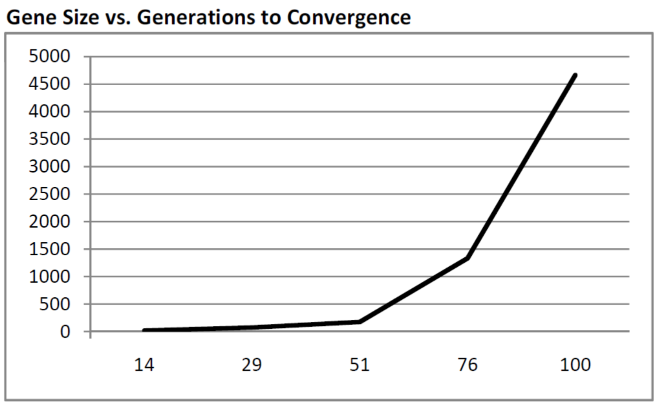

Figure 8: Plot of gene size vs. generations to convergence.

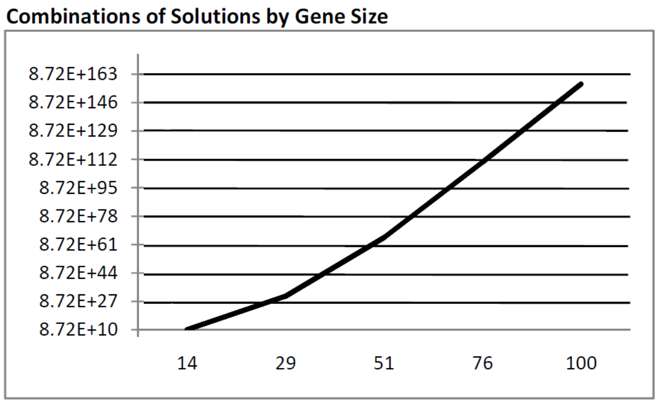

Figure 9: Logarithmic plot of gene size vs. combinations of solutions.

Conclusion

When plotting the data set solution values as the generations proceeded, we find the (roughly) logarthmic plot we would expect in all five cases.

The most interesting result of the experiments is seen when we look at the plot of gene size vs. the number of generations it took to converge to the solution the trial ultimately arrived at (Figure 8). Up to 51 genes we see a very quick convergence relative to the number of combinations of possible solutions. At 76 genes the slope of the line is roughly 1, and at 100 genes the slope of the line goes up to 2 to 3. Given the 5 data sets we used, genetic operators chosen, and parameters used, this would imply that this particular implementation of a GA excels at gene sizes 51 and below and is still quite performant at a gene size of 76.

Another interesting result comes when we examine the logarithmic graph of the combinations of solutions by gene size (Figure 9). The line plotted appears roughly linear (on the logarithmic scale), as would be expected given our choices for gene sizes for the 5 experiments. What is interesting is that we would expect this graph to look similar to the graph in figure 8, and very roughly it does. However, the portion of the graphs for gene sizes 14, 29, and 51 show the slope for generations to convergence being less than the slope for combinations of solutions. Then up to gene size 76, roughly the same size, and beyond, the slope of the generations to convergence being higher than the possible combinations of solutions. This would imply, again, that given the 5 data sets we used, genetic operators chosen, and parameters used, this particular implementation of a GA excels at gene sizes 51 and lower.

References

[1] Applegate, David L. *The Traveling Salesman Problem: A Computational Study*. Princeton: Princeton UP, 2006. Print.

[2-3,5] Mitchell, Melanie. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT, 1998. Print.

[4] Holland, John H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor: U of Michigan, 1975. Print.

[6] Whitley, Darrell; Timothy Starkweather, D'Ann Fuquay. "Scheduling problems and traveling salesman: The genetic edge recombination operator" *Genetic Algorithms: Proceedings of the Third International Conference on Genetic Algorithms, George Mason University, June 4-7, 1989*. San Mateo: Morgan Kaufmann Publ., 1989. 133-40. Print.

[7] Whitley, Darrell, Timothy Starkweather, and Daniel Shaner. "The Travelling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination." *Handbook of Genetic Algorithms*. By Lawrence Davis. New York: Van Nostrand Reinhold, 1991. Print.

[8] "TSPLIB." *TSPLIB*. Universität Heidelberg, n.d. Web. 26 June 2014.