Evolution of a Path Generator for a Round-Trip Symmetric Traveling Salesperson Problem Using Genetic Programming

Bretton Swope
Stanford Mechanical Engineering Department
Stanford University
Stanford, California 94305
bswope@stanford.edu

ABSTRACT

This paper describes the application of genetic programming to solve a 10 city example of the Round-Trip Symmetric Traveling Salesperson Problem (TSP). The genetic programming technique employed proves to be an excellent means of obtaining the solution for the specified problem. Furthermore, the results indicate that the solution obtained is case independent.

1. Introduction

The Round-Trip Traveling Salesman Problem (TSP) has been studied for decades in the computer science, mathematics, and artificial intelligence fields. The problem description is simple enough – you are a traveling salesperson who must visit a certain number of cities (assumedly by car) while traveling the least amount of distance. One stipulation about the journey is that you must finish in the city where you started, hence making it a round trip. While this may seem easy at first, most solutions obtained by just looking on a map are sub-optimal. In order to obtain the best travel path, one must employ an algorithm that determines the order of cities to visit that results in the lowest path distance.

The solution to TSP problems continues to appeal to scientists and engineers due to the myriad applications to discrete optimization problems. For example, Tsai et al. have used a TSP approach to solve the optimal gene order in a DNA-microarray (Tsai 2002). Also, the optimization of the order of holes drilled in printed circuit boards discussed by Applegate can result in reduced manufacturing time of products (Applegate 1998).

Many types of algorithms or heuristics have been used to obtain optimal solutions to problems involving up to 15,112 cities. It is the goal of this work to evolve a genetic program that can optimize a path for the simple case of up to 10 cities from a map of the United States. The cities listed along with their distance matrix are shown in Table 1. A map of the cities to be visited is shown in Figure 1.

Table 1 City Distance Matrix

		1	2	3	4	5	6	7	8	9	10
		Atl	Bos	Bir	Buf	Chi	Cle	Det	Ind	Jac	Mem
1	Atlanta	0	921	153	690	579	536	575	415	308	327
2	Boston	921	0	1048	396	848	554	628	805	1029	1128
3	Birmingham	153	1048	0	782	579	611	629	431	387	202
4	Buffalo	690	396	782	0	455	185	237	440	904	806
5	Chicago	579	848	579	455	0	303	220	167	881	489
6	Cleveland	536	554	611	185	303	0	98	257	784	622
7	Detroit	575	628	629	237	220	98	0	225	844	608
8	Indiana	415	805	431	440	167	257	225	0	715	384
9	Jacksonville	308	1029	387	904	881	784	844	715	0	588
10	Memphis	327	1128	202	806	489	622	608	384	588	0

Figure 1: Map of American Cities Used in Traveling Salesperson Problem.

As you can see above, the optimal city path seems relatively simple to figure out for a path of 10 cities. However, one should note that with 10 cities there are 10! or 3.62 million possible paths to take given an initial location.

2. TSP Heuristics

The widespread study of the TSP has created far too many heuristics to allow to an exhaustive list to be displayed in this paper, however there are a few simple techniques that are easy to understand and are worthy to note here.

Nearest Neighbor Heuristic - The nearest neighbor is an easily found greedy algorithm that simply evaluates the distances from the current city to all cities that have not been visited and chooses to visit the one which is closest to the current city. For example, if one were to start at city 1 from the map above, it is easy to read from the table above that the next city would be city 3. In fact, the specific city list used in the city matrix is solvable using the nearest neighbor heuristic, regardless of starting point. However, it is also easy to understand that the algorithm would be rendered sub-optimal for most trials if a city such as Stanford, CA were added to the list. The TSP would visit each closest city relative to their current city, ignoring Stanford because it is far outside of the simple city loop presented. Then, at the end of the trip the salesman would only be left with Stanford to visit before returning to the starting city. If the home city were city 2, then the salesman would be traveling far out of his way to return home.

Two-Optimum Exchanger - A 2-optimum exchanger simply takes a path that is already established and looks to see if switching two nodes will shorten the overall distance of the path. Once there are no remaining switches to be made, the optimal solution has been found.

Insertion Heuristic - Another intuitive technique known as the insertion heuristic begins with a three city tour that has the greatest path distance. Then, cities are added to the path based on the amount of distance they add to the total path. There are two basic types of path extending rationales:

- Minimum increase in distance – fueled by pure greed, this method adds the city which increases the total path distance the least.
- Maximum increase in distance – implemented to obtain the overall layout of the path early on in the search process, this rationale adds the city which increases the total path distance the most.

3. Genetic Programming

Genetic programming as described by Koza (1992), is a method of evolving a fit computer program using the Darwinian principle of natural selection and genetic operations. During a genetic programming run, hundreds of thousands of programs are created and run in order to evaluate each program's performance. Those with a higher value of fitness are selected to reproduce and are 'bred' using different means of crossover and mutation. The type of crossover may vary, but the result must consistently produce individuals that are functioning programs.

The programming language that genetic programming either employs directly or emulates is the simple LISP programming language. LISP expressions can be viewed as easily executable tree structures. These trees are constructed from nodes that are either defined as function nodes or terminal nodes. Function nodes perform a desired function on their children and return the result to their parent. Terminal nodes may only be children and thus are at the ends of the tree structure. A sample program tree and LISP expression are shown in Figure 2 below. In the expression and tree, the function node `F1` receives two inputs (one from each child) and returns some function of those inputs to its parent. The function node `F2` receives three inputs from and returns one input to its parent. The terminals `X` and `Y` have no children and simply provide their value to their parent.

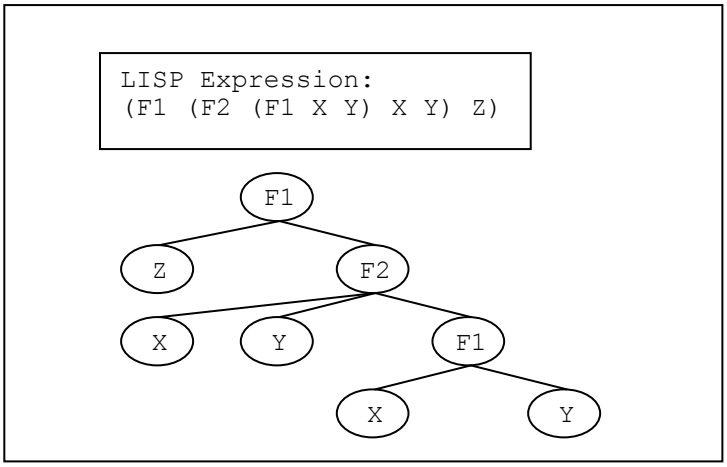

Figure 2: Example LISP Expression and Program Tree

It is typical in genetic programming to minimize the amount of human knowledge or into the genetic program's function set. One seeks to increase the amount of automated operation in relation to the amount of presupplied information by the human user. This ratio is known as the AI ratio.

4. Programming Setup

In order to solve the Traveling Salesperson Problem presented in this paper, a software package entitled ECJ was chosen to provide the genetic programming framework. This software, written in the Java programming language by Sean Luke, provides an easily scalable and robust package for quickly solving many GP and GA problems.

The primary operations used to modify the program trees were crossover, reproduction and mutation. All primary operations used tournament selection with a tournament size of 7. The percentages of crossover and reproduction for the runs were .9 and .1 respectively. The number of trees allowed in the evolved program was set to one.

The runs were performed on an Intel Pentium IV processor running at 2,000 Mhz. Version 1.4.2_02 of the Java virtual machine was used to run the compiled GP code. Further details about the various genetic programming parameters are given in the results and discussion section of the document as appropriate.

5. Results and Discussion

5.1. Initial Approach - Genetically Programmed Traveling Salesperson

For the first attempt at solving the traveling salesperson problem, a simple case of 5 cities was examined and the function set was chosen so that a solution could easily be realized. In order to provide closure to the function set an approach to the problem was found that allows the program to return a path at the end of its evaluation. This path is

then tested against the optimum distance path for the cities found in the return path and a penalty is given based on the distance traveled relative to the optimum distance traveled. Table 2 shows the genetic programming tableau for this initial approach.

Table 2 Genetic Programming Tableau for Initial GPTSP

Objective:	To find a program that returns the optimum path for a 5 city TSP.
Terminal set:	`City 1, City 2, City 3, City 4, City 5.`
Function set:	`CombinePaths, OptimizePath`
Fitness cases:	The optimal path containing cities 1 through 5.
Raw fitness:	The diference between the distance of the returned path and the distance of the optimal path plus a penalty if a path does not contain all the cities.
Standardized fitness:	Same as Raw fitness.
Hits:	The number of cities contained in the path generated by the program.
Parameters:	Population sizes M = 5,10,20,25,30,50
Wrapper:	none
Success predicate:	An S-expression representing a program that generates a path containing all the required cities and whose length is equal to the optimal path.

The Terminals and Functions found in the Table 2 are defined as follows:

`City 1 through City 5` – These terminal nodes are simply paths containing one city corresponding to their name.

`CombinePaths` – Combines the paths of its two children and returns the combined path to the parent

`OptimizePath` – evaluates its only child's path, optimizes the path, and returns the optimized path to the parent.

As Table 2 indicates, the function set only consisted of two functions that allowed for combining paths and optimizing paths. The terminal set consisted of only five of the ten total cities available. A fitness function was defined as the following:

$$\text{Fitness} = (5\text{-}N_C) + (O_{PD} - I_{PD}) \qquad F(1)$$

Where N_C is the number of cities visited, O_{PD} is the optimal path distance, and I_{PD} is the Individual's path distance.

Using these parameters, solutions quickly emerged during generation zero. This is an expected result because the search space of possible combinations of five cities is not very large and a simple random combining of the five paths will result in an ideal individual rather quickly. In fact, the problem is almost analogus to a pre-GP genetic algorithm in which the chromosome simply indicates how to use the function set. One interesting but unsurprising bit of knowledge that was gleaned from the initial run was that if the population size was reduced to 5, the solution would not converge after 51 generations. While this is only 255 individuals, the program never had more than four out of five hits. In fact, the program never evolved to use city number 2. This is because the first generation did not randomly create any individuals with a city 2 terminal in the initial population of five and hence there was no city 2 to reproduce or crossover into future generations.

5.2. Second Approach - Genetically Programmed Traveling Salesperson

In order to create a more advanced program and increase the AI ratio of the approach, a few modifications were made to the initial approach. First, the rest of the cities were added to the matrix so that the number of possible combinations prevented random luck from generating an optimal individual in the beginning of the run. Secondly, a new function was added to replace the `OptimizePath` function that would force the evolution of more sophisticated programs that were not simply assembling all of the cities and then calling on the `OptimizePath` function at the top of the tree. The tableau for this approach is shown in Table 3 below.

Table 3 Genetic Programming Tableau for Second Approach

Objective:	To find a program that returns the optimum path for a 10 city TSP.
Terminal set:	`City 1, City 2, City 3, City 4, City 5, City 6, City 7, City 8, City 9, City 10`
Function set:	`CombinePaths, OptimizePath, 23SwapAndShift`
Fitness cases:	The optimal path containing cities 1 through 10.
Raw fitness:	The diference between the distance of the returned path and the distance of the optimal path plus a weighted penalty if a path does not contain all the cities plus a weighted penalty based on the number of nodes in the individual's tree.
Standardized fitness:	Same as Raw fitness.
Hits:	The number of cities contained in the path generated by the program.
Parameters:	Population sizes: M = 100,200,300,500,1000 Generations: G = 51, 100, 200, 500
Wrapper:	none
Success predicate:	An S-expression representing a program that generates a path containing all the required cities and whose length is equal to the optimal path.

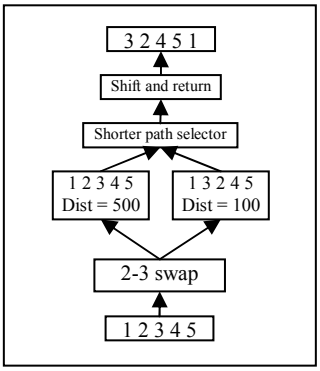

Figure 3: TwoThreeSwapAndShift

The new function added was the `TwoThreeSwapAndShift` function. This function is meant to behave similarly to the 2-optimum improvement heuristic. This function evaluates its only child and if its path contains more than 3 cities it swaps the 2nd and 3rd cities. If this new path is better than the old one, it shifts the order of the path one city to the left and returns the path to its parent. If the swapped path is not shorter than the original path, it simply shifts the path one city to the left and returns the individual to its parent. A diagram of the `TwoThreeSwapAndShift` function is shown in Figure 3. Note there is no direct optimization present in the function, just a simple lookup command that determines which path is shorter.

By adding all ten cities to the terminal set, adding the `TwoThreeSwapAndShift` function, and eliminating the `OptimizePath` function from the function set, much more interesting individuals start to evolve. At first, however, convergence was not possible even with large populations and hundreds of generations. It was discovered that the fitness function was driving the solution away from paths that contained 10 cities. This is easy to understand based on the fitness function F(1) given in the previous section. One should note that the penalty for having 1-9 cities is only a value between 1 and 9 respectively, while the penalty for having a poor path distance is the distance of the path minus the optimal distance for the path. For poorly constructed 10 city paths, the distance penalty would far outweigh the penalty for having a 5 city path. The GP was evolving individuals that had a perfect path but only 8 cities. This problem was solved by using a factor, M, to magnify the penalty for having a path with less than 10 cities. A new fitness function, shown below, was implemented and solutions began to converge once again.

$$\text{Fitness} = M(10-N_C) + (O_{PD} - I_{PD}) + N(C_N) \qquad F(2)$$

The initial value of N was set to zero for reasons explained below. By setting the value of M to 1,000 it was found that the GP quickly adapted and chose to use all 10 cities. This is because even the worst 10 city path only has a distance penalty of a few thousand miles. Individuals with poor path distances and less than ten cities were eliminated very early in the run using this new fitness measure.

The new function set caused the evolved program trees to be considerably more complex than those of the initial problem approach. These new trees would converge rather quickly but contained around one hundred nodes. For example, during one run of 200 individuals, the following correct solution individual emerged during the fourth generation:

```
Final Statistics
Total Individuals Evaluated: 1000
Best Individual of Run:
Fitness: Raw=0.0 Adjusted=1.0 Hits=10

(CombinePaths (CombinePaths (CombinePaths (TwoThreeSwapAndShift (TwoThreeSwapAndShift City10)) (TwoThreeSwapAndShift
(CombinePaths City6 City6))) (CombinePaths (CombinePaths (TwoThreeSwapAndShift (CombinePaths (CombinePaths (CombinePaths
(TwoThreeSwapAndShift (TwoThreeSwapAndShift City1)) (CombinePaths (CombinePaths City4 City7) (TwoThreeSwapAndShift City6)))
(TwoThreeSwapAndShift (TwoThreeSwapAndShift (CombinePaths (TwoThreeSwapAndShift City9) (CombinePaths City5 City1)))))
(CombinePaths (CombinePaths (CombinePaths (CombinePaths City10 City6) (CombinePaths) (TwoThreeSwapAndShift (CombinePaths City6
City4)) (CombinePaths (CombinePaths City7 City3) (CombinePaths City10 City4)))) (CombinePaths (TwoThreeSwapAndShift City3)
(TwoThreeSwapAndShift City6))) (TwoThreeSwapAndShift (CombinePaths City10 City5))))) (CombinePaths (CombinePaths (CombinePaths
```

```
(CombinePaths City9 City8) (CombinePaths City6 City6)) (CombinePaths (CombinePaths City8 City7) (CombinePaths City6 City10)))
(CombinePaths (TwoThreeSwapAndShift (TwoThreeSwapAndShift City6)) (CombinePaths (TwoThreeSwapAndShift City4) (TwoThreeSwapAndShift
    City7))))) (TwoThreeSwapAndShift     (CombinePaths (CombinePaths City3 City9) (CombinePaths City5 City6))))) (CombinePaths
        (TwoThreeSwapAndShift (TwoThreeSwapAndShift (CombinePaths City1 City2))) (CombinePaths (CombinePaths City2 City7)
                                    (TwoThreeSwapAndShift (TwoThreeSwapAndShift City2)))))
```

While this program may lead to the correct solution, it was wondered whether there may be more parsimonious solutions. In order to determine this, the node counter coefficient in the fitness function , N, was given a value of one. This caused the program to try and minimize the size of the lisp expressions. Using the new fitness function, it was found that much smaller trees could generate an optimal path. Using the new fitness measure, the following 27 node individual was obtained in run V7.9. Its population and generation count were both 500:

```
                                    Final Statistics
                            Total Individuals Evaluated: 250000
                                    Best Individual of Run:
                            Fitness: Raw=27.0 Adjusted=0.035714287 Hits=10

(CombinePaths (CombinePaths (CombinePaths (TwoThreeSwapAndShift (TwoThreeSwapAndShift City9)) (CombinePaths (TwoThreeSwapAndShift
      (TwoThreeSwapAndShift (CombinePaths City6 City7))) (CombinePaths (TwoThreeSwapAndShift City10) (TwoThreeSwapAndShift
  (TwoThreeSwapAndShift City1))))) (CombinePaths (TwoThreeSwapAndShift City4) City5)) (CombinePaths City3 (CombinePaths City8
                                                City2)))
```

It is readily apparent that the second individual has considerably fewer nodes that its predecessor and it still uses all the functions available. One problem with using this method to hone in on a parsimonious solution is that the raw fitness will never obtain a value of zero. The population quickly converges to smaller trees as can be shown by the plot of the best of generation fitness for run V7.9 shown in Figure 4 (Note: The first 3 generations were not included due to their large fitness values).

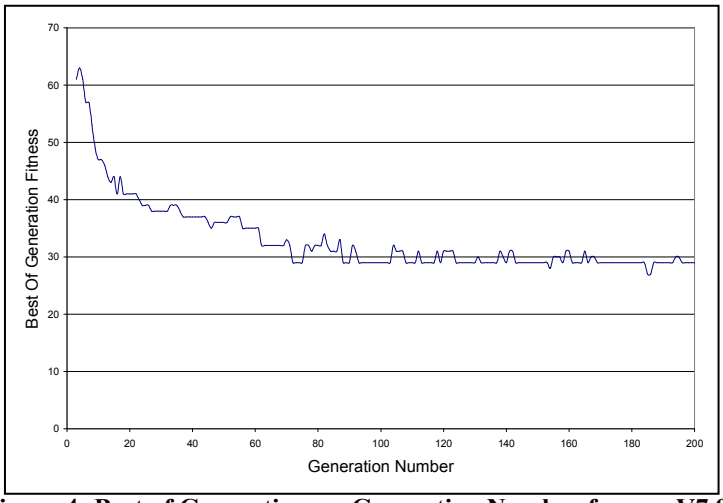

Figure 4: Best of Generation vs. Generation Number for run V7.9

As the figure above indicates, a solution emerged in the 72nd generation which contained only 29 nodes. Also, the first individual with 27 nodes appeared in generation 185. The node count continues to oscillate around 30 for the remainder of the 500 generations of this run.

5.3. Third Approach - Genetically Programmed Traveling Salesperson

Having found a relatively parsimonious solution to the 10 city problem using the `TwoThreeSwapAndShift` function, the function was broken down into two separate functions, one to perform the swapping, known as `TwoThreeSwap` and a separate function known as `ShiftOne` to perform the shifting. A multitude of runs were performed using the same tableau shown in Table 3 with the addition of these two new functions in place of their predecessor the `TwoThreeSwapAndShift` function. This function worked surprisingly well and converged after only 7 generations with a population size of 750 and 8 generations with a population of 200. Table 4 below shows various other runs and their outcomes.

Table 4 Results of Runs using separate Shift and Swap Functions

File Name	Seed	Pop Size	Gen's	Converge	Cities	Value of M	Value of N	Node Count
v8.6	4357	750	500	7	10	1000	0	Na
v8.7	4357	500	50	7	10	1000	0	Na
v8.8	4357	200	50	8	10	1000	0	Na
v8.9	4357	100	50	10	10	1000	0	Na
v8.10	4357	50	50	14	10	1000	0	Na
v8.11	4357	25	50	N	10	1000	0	Na
v8.12	4357	25	500	N	10	1000	0	Na
v8.13	4357	30	500	N	10	1000	0	Na
v8.14	4357	40	500	N	10	1000	0	Na
v8.15	4357	45	500	28	10	1000	0	Na
v8.16	4357	200	50	N	10	1000	1	51
v8.17	4357	200	500	N	10	1000	1	31
v8.18	4357	500	500	N	10	1000	1	34
v8.19	4357	5000	500	N	10	1000	1	19

Runs with populations less than 40 were unable to converge within 500 generations while those with populations over 45 were able to converge in acceptable number of generations. When the node count was factored into the fitness to obtain parsimony, the lowest node count that used the entire function set occurred during run V8.17 and was 31 nodes for a population of 200 after 500 generations. Although an individual with only 19 nodes was obtained in run V8.19 by increasing the population size to 5000 individuals, the 19 node individual didn't use all of the functions available and was able to simply combine the paths in the correct order thus turning the problem into a genetic algorithm problem.

It might seem surprising at first that the number of nodes seems so small when we take into account tree parsimony. One would expect the GP to evolve the behavior of the `TwoThreeSwapAndShift` function by simply calling a `ShiftOne` function every time the `TwoThreeSwap` function is called. Instead, the GP takes advantage of the fact that when a city is added or a path is combined, a shift takes place automatically and a `ShiftOne` command is not necessary. This type of illogical discontinuity is a well known advantage of genetic programming and is illustrated well by this case.

Figure 5 illustrates the best of generation raw fitness, average individual raw fitness, and 1000 times the number of cities visited by the best of generation for run V8.9. Note how the program realizes the need to visit all the cities by the 5th generation. The increase in cities visited directly corresponds to a decrease in the raw fitness of the best of generation individual due to the heavy weight factor applied to missing any cites. It is also interesting to note that the average raw fitness of the generation is following a similar path as the best of generation raw fitness as expected.

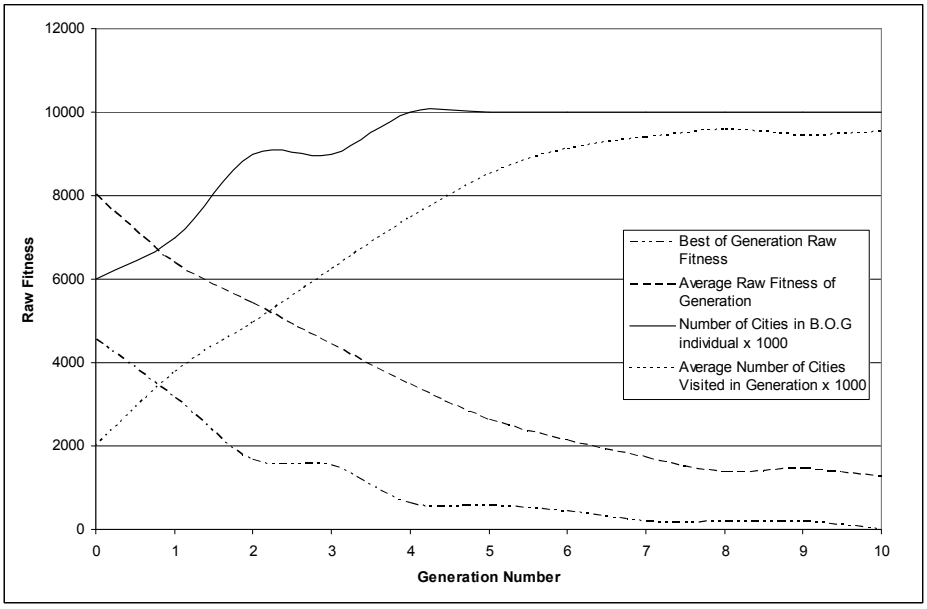

Figure 5: Raw Fitness vs. Generation Number for Run V8.9

5.3. Final Approach - Genetically Programmed Traveling Salesperson

While studying the data from the previous approaches it because apparent that the programs evolved might be case specific because only one city matrix was used in the evaluation function. Even with a 10 city path, The GP may be evolving an individual that can swap, shift and combine its way to an optimum case just by using simple GA type search of all possible combinations of cities. To ensure that this wasn't the case, a final approach was used that added a twist to the fitness function to ensure that the solution was applicable to more than one order of input nodes. The evaluation is as follows:

1. Fist the individual was evaluated using fitness function F(2) with M and N values of 1000 and zero, respectively.
2. Next, anywhere the terminals `city 3` and `city 5` were located in the tree, they were swapped. This new individual had the same function tree, but the input by way of the terminals was different.
3. the fitness of the new individual was evaluated using fitness function F(2) and the result was added to the result of part 1 above. This resulting total was the overall individual fitness.

The parameters used in this approach remained the same as those used in the third approach except the fitness case was changed to reflect the steps outlined above. Using this new approach, we were able to obtain an individual that converged in the 7th generation of a run of 200 individuals per generation. While this is not proof that we are obtaining the solution for all traveling salesperson problems, it is proof that the program is not suited for merely one case of inputs. In fact, the GP managed to optimize the unaltered individual several times in generation 5 but was only able to optimize the altered individual once. This data proved to be very exciting and indicates that there is potential for evolving a general case solution to a traveling salesman problem.

6. Conclusions

Several different approaches were used to create an evolved genetic program that is able to solve a Round-Trip Symmetric Traveling Salesperson Problem. The first method had a very low AI ratio in that the functions given performed most of the work and the genetic program merely had to ensure that all cities were contained in the path. These solutions converged far to fast and the solution was too trivial allowing the program to evolve the solution in the initial generation of randomly generated individuals. Runs that did not converge were only obtained by substantially limiting the population size.

The second approach to the TSP problem replaced the `OptimzePath` function with an interesting function similar to the 2-optimum improvement heuristic. This method provided a cleaner-handed approach and thus a higher AI ratio. The solutions using the applied function set continued to converge rather quickly. An investigation into parsimonious programs was performed by adding a multiple of the number of nodes in the program trees to the fitness functions. This new fitness measure provided individuals that had approximately one-quarter the number of nodes as those previously discovered in the second approach.

The third approach to the TSP problem further increased the AI ratio by separating the `TwoThreeSwapAndShift` function into two separate functions, `TwoThreeSwap` and `ShiftOne`. This resulted in later convergence of an individual with optimal path configuration. A noteworthy observation occurred in the third approach that involved a well known characteristic of genetically evolved programs. This characteristic was realized as the individuals evolved using a larger function set were able to use a similar amount of nodes.

Finally, in the fourth approach, a more advanced fitness measure was performed that ensured that the evolved program was not case specific. The fitness function used a clever swapping of the `city 2` and `city 5` terminals to achieve this.

The results obtained show that it is possible to evolve a round trip symmetric traveling salesperson problem involving 10 cities. The data gathered also indicates that the solution obtained is proven to be case independent but not matrix independent.

7. Future Work

Future work should be performed to evolve solutions that are not matrix or case specific. This could be done by evaluating an individual against many different paths and summing the individual deviations from the ideal distance for the particular city set. Also, more cities should be added to the city matrix so that the problem becomes increasingly complex. Along the same lines, more distance matrices could be used to perform the fitness evaluation. Perhaps an individual could be measured against totally different sets of cities to ensure that we are not simply optimizing for one set of cities. Also, new functions should be employed that continue to increase the AI ratio of the approach. Such function sets could involve more mathematical operations and thus force the GP to evolve quantitative expressions to determine an optimal path.

Acknowledgments

The author would like to thank Mark Swope for his Java debugging expertise.

Bibliography

Applegate, D. et al. 1998. On the solution of traveling salesman problems *Documenta Mathematica - Extra Volume, International Conference on Mathematics III.* Pages 645-658.

Davis, L. D. et al. 1999. *Evolutionary Algorithms*. New York, NY: Springer.

Huai-Kuang Tsai, et al. 2002. Applying Genetic Algorithms to Finding the Optimal Gene Order in Displaying the Microarray Data. Langdon W., et al. (editors). *Proceedings of the Genetic and Evolutionary Computation Conference, Gecco 2002.* San Francisco, CA: Morgan Kaufman Publishers. Pages 610-617.

Miettinen, K. et al. 1999. Evolutionary Algorithms in Engineering and Computer Science. New York, NY: John Wiley & Sons, LTD.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.