Augmenting Genetic Algorithms with Memory to Solve Traveling Salesman Problems

Sushil J. Louis **Gong Li**
Dept. of Computer Science
University of Nevada,
Reno, NV 89557
sushil@cs.unr.edu
li_g@cs.unr.edu

Abstract

This paper explores the feasibility of augmenting genetic algorithms with a long term memory. During a genetic algorithm run, we periodically store individuals in a database. When confronted with a new problem, instead of starting from scratch, we inject the solutions to previously solved similar problems (from the database) into the initial population of the genetic algorithm. We evaluate the performance of the genetic algorithm with such a long term memory on a set of benchmark traveling salesman problems. In addition, we compare the performance of these augmented genetic algorithms when trained on traveling salesman problems of *varying* similarity. Preliminary results indicate that we can always get better performance with injection of previous solutions to similar problems.

1 Introduction

Genetic algorithms (GAs) are randomized parallel search algorithms that search from a population of points [6]. We typically randomly initialize the starting population so that a genetic algorithm can proceed from an unbiased sample of the search space. However in many application areas we confront sets of similar problems. It makes little sense to start a problem solving search attempt from scratch with a random initial population when previous search attempts may have yielded useful information about the search space. Instead, seeding a genetic algorithm's initial population with solutions to similar previously solved problems can provide information (a search bias) that, hopefully, increases the efficiency of the search.

Genetic algorithms have been used to solve large TSPs and can get good solutions quickly [3, 6, 7]. The first efforts to find near optimal solutions to TSPs by using GAs were those of Goldberg using Partial Mapped Crossover [4] and Grefenstette using Greedy Crossover [5]. Davis, Smith, Suh and Van Gucht also tried to solve TSPs with various crossover operators [1, 10, 11]. In this paper, we present evidence to show that augmenting genetic algorithms with a memory of solutions to previously solved similar traveling salesman problems results in better performance than running a genetic algorithm with random initialization. Ramsey and Grefenstette come closest to our approach and use case-based initialization to increase a genetic algorithm's performance in an environment that changes with time [8]. Although they report encouraging results on non-stationary functions, their work does not address the effect of similarity on performance that is addressed in this paper. In the next few sections we describe the genetic operators, our methodology, and results.

2 The Traveling Salesman Problem

The traveling salesman problem(TSP) is: given N cities, if a salesman starting from his home city is to visit each city exactly once and then return home, find the order of a tour such that the total distance traveled is minimum. The TSP is a classical NP-complete problem which has extremely large search spaces and is very difficult to solve. People have tried to use both exact and heuristic or probabilistic methods to solve the TSP.

The objective function for the N cities two dimensional Euclidean TSP is the sum of Euclidean distances between every pair of cities in the tour. That is:

$$\text{Fitness} = \sum_{i=1}^{i=N} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Where, x_i, y_i are the coordinates of city i and x_N, y_N equals x_0, y_0. We also make some changes to the encoding, selection, and recombination.

2.1 Recombination

Our sequential path representation (a ordered list of cities to visit) means that traditional crossover and mutation operators are not suitable for TSPs. Instead we use Greedy Crossover [5]. Greedy crossover selects the first city of one parent, compares the cities leaving that city in both parents, and chooses the closer one to extend the tour. If one city has already appeared in the tour, we choose the other city. If both cities have already appeared, we randomly select a non-selected city. Mutation is implemented by swapping two randomly chosen sites.

2.2 Selection

When using traditional roulette wheel selection, the best individual has the highest probability of survival but does not necessarily survive. We use CHC selection to guarantee that the best individual will always survive in the next generation [2]. In CHC selection if the population size is N, we generate N children by using roulette wheel selection, then combine the N parents with the N children, sort these $2N$ individuals according to their fitness value and choose the best N individuals to propagate to the next generation. We found that with CHC selection the population converges quickly compared to roulette wheel selection and the performance is also better. To prevent convergence to a local optimum, when the population has converged we save the best 10% of the individuals and re-initialize the rest of the population randomly. This results in slightly better performance.

3 Methodology

We construct similar problems from our benchmark TSP [9] problems by modifying the benchmarks in five different ways. Table 1 lists the five kinds of modifications that were used. Since we know the current best solution to the TSP benchmarks, we train the genetic algorithm on the modified problem P_{mod} saving and using the solutions to P_{mod} to help solve the original problem. We compare the performance of the Non-Randomly initialized GA (NRGA) with a Randomly initialized GA (RGA) and the known optimal solution. For all the problems, we use the same crossover and mutation probabilities of 0.85 and 0.05 respectively but use different population sizes and various number of generations for different problems. Table 2 shows the population sizes and the number of generations we use. For each modified problem, we run the RGA 10 times with 10 different random seeds and periodically save the best individual. We inject these chosen individuals (eight total) into the initial population of the NRGA to help solve the original problem.

Problem	Size
same	same
changing one city	same
changing two cities	same
adding one city	one more
deleting one city	one less

Table 1: Different ways of modifying TSPs

No. of cities	Population size	No. of generations
52	200	300
76	250	400
105	300	400
127	300	500

Table 2: Population and generation size for different problems

4 Results and Analysis

4.1 Results

Table 3 shows the average performance across dif-

Problem	Start perc.	End perc.
random	491.64%	111.19%
same	109.58%	104.97%
diff1	112.55%	106.49%
diff2	119.39%	106.05%
add1	109.48%	105.36%
del1	116.49%	107.65%

Table 3: Average distance from optimal for all problems

ferent sized problems for running the NRGA with different degrees of similarity. Column one shows the modified problem, while column two and three show the percentage distance of the first and the last generation tour length from the optimal tour length. From this table, we can see that after injecting individuals and running GAs with previous information, we can always get better results than when running GAs with random initialization on our problems. We provide details on each modified problem below:

1. **Same problem (*same*):** We do not change the problem letting P_{mod} be the original problem and solve the original problem twice. Figure 1 shows the optimal solution, the average performance from running the RGA 10 times and the average performance from running the NRGA 10 times for the 76 cities TSP. As expected, injecting solutions from the same problem can help NRGAs get better performance especially in early generations.[1]

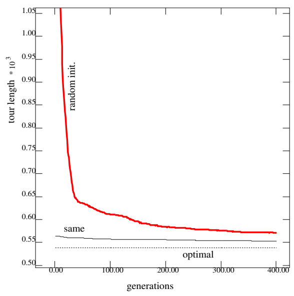

Figure 1: Performance on 76 cities problem with injection from solutions to the same problem

2. **Change one city (*diff1*):** We slightly change the original problem by randomly selecting one city and changing its location by randomly changing its x and y coordinates by 10% to 25%. Note that the problem size remains the same. Now, since P_{mod} and the original problem are different, the performance during early generations is not as good as the performance when injecting solutions from the same problem but the performance is still better than with random initialization.

3. **Change two cities (*diff2*):** We randomly select two cities and change their locations by randomly changing their x and y coordinates by 10% to 25%. The performance graph (not shown here) indicates that the performance is still better than that of an RGA. When changing two cities instead of changing one, the two problems are less similar which leads to the difference in performance between *diff1* and *diff2*. During earlier generations, the performance of *diff1* is always better than that of *diff2*. However, during later generations, this is not always true and *diff2* actually results in better performance on some of our benchmarks.

4. **Add one city (*add1*):** To modify the original problem, we not only change the location of cities, we also change the size of the problems. We do this by adding or deleting one city. When adding one city, we generate a new city and let its coordinates be (randomly) *between* the maximal and minimal coordinates of all cities. We run the RGA to solve the modified problem and save the best individuals. For each of these individuals, we delete the added city so the changed individuals are legal tours for the original problem. Then we inject these changed individuals to solve the original problem. The average performance graphs show that adding one city can help GAs get performance similar to that obtained when injecting solutions from the *same* problem.

5. **Delete one city (*del1*):** We also randomly delete one city, run GAs to solve the modified problem and save the best individuals. For each one of these individuals, we randomly select one site and insert the deleted city, so that the changed individuals become legal tours for the original problem. We then inject these new individuals to solve the original problem. Because we randomly insert the deleted city, the performance during the first few generations is not as good as when injecting solutions from other similar problems but the performance is better than an RGA.

4.2 Analysis

As Table 3 shows, after injecting solutions of previous similar problems, the performance is always better than running the GA with random initialization. This may imply that injecting individuals from previous solved similar problems is a good way to help the GA get better results for TSPs. This may also mean that if most cities of the two TSPs are the same, we can use solutions from one TSP to solve the other one.

[1] We do not show the performance graphs for other TSP benchmarks due to space constraints.

Short, highly fit subsets of strings (building blocks) play an important role in the action of genetic algorithms because they combine to form better strings [3]. We believe edges are the building blocks for traveling salesman problems. After we inject individuals containing good edges from similar problems, the NRGA can combine these good edges and get good performance quickly.

Our results show that injecting individuals from the *same* problem and the *add1* problem can help the NRGA get better performance than when injecting individuals from other problems. In these two cases, we use information from all cities and the length of each edge is still the same. This implies that problems can be similar enough to be useful even if they are of different size. In other words, different sized problem with information from all cities and all edges can help the NRGA get better performance than a same sized problem with some different edges.

5 Conclusions

This paper demonstrates that running a genetic algorithm with injection of individuals from solutions to similar problems can get better performance than running a genetic algorithm with random initialization. We also find that running GAs with information from the original problem or the adding one city problem can get better performance than running GAs with information from other kinds of modified problems. We believe this is because the *same* problem and the *add1* problem contain all cities of the original problem and do not change the length of the edges and are therefore more similar to the original problems.

Although the approach has promise, much work remains to be done. We need to consider the effect of different selection schemes, recombination operators, and niching operators, for genetic search, as well different search algorithms. Deriving more refined estimates on the quality and quantity of individuals to inject will broaden applicability. Individuals need not be injected solely into the initial population. We can keep track of the performance of injected individuals and their progeny and use this information to design and inject individuals in intermediate generations. Finally, the tradeoffs between speed and solution quality needs to be explored in more detail.

6 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 9624130.

References

[1] L. Davis. *Job shop scheduling with Genetic Algorithms.* Lawrence Eribaum Associates, Mahwah, NJ, 1985.

[2] Larry J. Eshelman. *The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination.* Morgan Kauffman, San Mateo, CA, 1990.

[3] D. E. Goldberg. *Genetic Algorithms in Serach, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[4] D.E. Goldberg and R. Lingle. *Alleles, Loci and the traveling salesman problem.* Lawrence Eribaum Associates, Mahwah, NJ, 1985.

[5] J.J Grefenstette, R. Gopal, R. Rosmaita, and D.V. Gucht. *Genetic Algorithms for the traveling salesman problem.* Lawrence Eribaum Associates, Mahwah, NJ, 1985.

[6] J. Holland. *Adaptation In Natural and Artificial Systems.* The University of Michigan Press, Ann Arbour, 1975.

[7] Heinz Muhlenbein. *Evolution in Time and Space – The Parallel Genetic Algorithm.* Morgan Kauffman, San Mateo, CA, 1990.

[8] C. Ramsey and J. Grefensttete. Case-based initialization of genetic algorithms. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 84–91, San Mateo, California, 1993. Morgan Kauffman.

[9] Gerhard Reinelt. *TSPLIB.* University of Heidelberg, http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html, 1996.

[10] D. Smith. *Bin packing with adaptive search.* Lawrence Eribaum Associates, Mahwah, NJ, 1985.

[11] J.Y. Suh and D. Van Gucht. *Incorporating heuristic information into genetic search.* Lawrence Eribaum Associates, Mahwah, NJ, 1985.