A Web-Based Evolutionary Algorithm Demonstration using the Traveling Salesman Problem

Richard E. Mowe
Department of Statistics
St. Cloud State University
mowe@stcloudstate.edu

Bryant A. Julstrom
Department of Computer Science
St. Cloud State University
julstrom@eeyore.stcloudstate.edu

Abstract

An evolutionary algorithm applies operators inspired by biological selection and reproduction to search large solution spaces. In the well-known traveling salesman problem (TSP), we are given a collection of cities and the distances between them, and we seek the shortest tour that visits each city exactly once and returns again to the start city. In an earlier presentation to this symposium, we presented a program written in Visual Basic that demonstrated an example of a genetic algorithm—one flavor of EA—for the TSP. The present project implements the genetic algorithm in Java. The current implementation is more flexible than the old, and can be made available via the World Wide Web to any computer with a Java 2-enabled browser.

1. Introduction

An evolutionary algorithm (EA) applies operators inspired by biological selection and reproduction to search large solution spaces. In an EA, data structures, usually strings, represent candidate solutions to the target problem. The algorithm maintains a population of these structures, which are called chromosomes. Each chromosome has a numerical fitness that indicates the quality of the solution it represents.

The algorithm selects chromosomes from its population to survive or reproduce; chromosomes of higher fitness—that is, that represent better solutions—are more likely to be selected. It applies operators inspired by genetic recombination and mutation to generate new chromosomes that represent new candidate solutions. Crossover combines the genetic material of two parents, while mutation randomly modifies a single parent chromosome. After enough offspring have been generated, they replace some or all of their parents, and the process continues. As generations of chromosomes succeed each other, the solutions that they represent improve. The algorithm halts after a preset number of generations or when it identifies a solution of adequate quality.

In the well-known Traveling Salesman Problem (TSP), we are given a collection of n cities and the distances between each pair of them, and we seek a tour that visits each city exactly once and returns to the start city in the shortest possible distance. Formally, the TSP seeks a Hamiltonian tour of minimum total weight in a complete weighted graph on n vertices. Though simple to state, the TSP is computationally difficult; it is a standard example of an NP-hard problem (Garey and Johnson, 1979, pp.56-60). Thus it is unlikely that any polynomial-time algorithm exists for its exact solution, and we turn to heuristics. Among those heuristics are evolutionary algorithms, which have regularly been applied to the TSP.

Evolutionary algorithms come in several flavors, depending on how chromosomes encode candidate solutions, whether chromosomes are selected to survive or to reproduce, the operators that generate offspring from parents, and the problems to which they are usually applied. The kind of EA we exhibit here—the genetic algorithm (GA)—is characterized by chromosomes that are strings of symbols, selection that chooses chromosomes to be parents in reproduction, crossover and mutation operators that exchange and modify symbols in parent chromosomes, and application to problems of function optimization and combinatorial optimization. The TSP is an example of the latter; it seeks an optimal ordering of its cities.

In an earlier paper (Julstrom and Mowe, 1996, pp. 330-337), we described a Visual Basic program that demonstrated a genetic algorithm for the traveling salesman problem. The present project implements the demonstration in Java. The new implementation is platform-independent, more flexible than the old, and can be made available through the World Wide Web to any computer with a Java 2-enabled browser.

The following sections of the paper describe: the details of the genetic algorithm that the demonstration implements; the earlier Visual Basic program; the Java GA program; implementation of the crossover and mutation operators; the operation of the program; and its incarnation as an application and as an applet.

2. The Genetic Algorithm

Though some of the first researchers to write genetic algorithms for the traveling salesman problem commented on the dearth of evolutionary interest in combinatorial problems like the TSP (Greffenstette, et al., 1985), subsequent investigations have more than filled this gap. Many researchers have written GAs for the TSP, and these algorithms have used a wide variety of program structures, codings of candidate tours, crossover and mutation operators, and hybridization with other heuristics. Michalewicz (1996, Ch.10) provides a good overview of these techniques.

Since it is our purpose to present a demonstration of evolutionary computation via the TSP rather than to construct a state-of-the-art GA for the problem, the design choices our algorithm implements are conventional and conservative. First, the GA's chromosomes represent candidate tours in the obvious way: as permutations of the cities. A chromosome's tour visits the cities in listed order and closes the tour by returning to the first city. Its fitness is the length of this tour.

Second, the program's structure is straightforward. After initializing its population with random tours, it runs through a fixed number of generations. In each generation, it builds a population of offspring by applying either crossover to two parents or mutation to one. Parents are selected by choosing two chromosomes from the population at random, then selecting the one that represents the shorter tour, a mechanism called a 2-tournament. The algorithm also preserves the one best chromosome from the current generation into the next; this is called 1-elitism. Figure 1 summarizes the algorithm's structure.

The crossover operator combines two parent tours to build a single offspring tour, and it uses the inter-city distances so as to favor the construction of shorter tours. It chooses a starting city at random, then repeatedly extends the tour to the nearest city that is adjacent to the current one in either parent. If the tour already visits all those cities, the operator appends the nearest unvisited city. Julstrom (1995) called this operator very greedy crossover; it extends earlier greedy crossovers for TSP tours such as those by Greffenstette, et al. (1985) and Jog, Suh, and van Gucht (1989).

The mutation operator reverses a random segment of its one parent tour, thus reversing the order in which the cities in the segment are visited. Every new chromosome is generated by either crossover or mutation, never by both; the choice is made for each new chromosome probabilistically.

```
initialize the population with random tours;
for G generations
{
copy the best chromosome into the next generation;
while the next generation is not full
{
choose an operator: crossover or mutation;
if crossover
{
select two parents in 2-tournaments;
apply crossover to generate an offspring
chromosome;
}
else  // mutation
{
select one parent in a 2-tournament;
mutate it to generate an offspring;
}
evaluate the offspring;
insert the offspring into the next generation;
}
offspring replace parents;
}
```

Figure 1. A sketch of the genetic algorithm for the traveling salesman problem that the demonstration program implements. Evaluating an offspring chromosome means finding the length of the tour that it represents.

3. Visual Basic Prototype

The earlier implementation of this algorithm in Visual Basic featured a graphic user interface (GUI) that displayed tours of the current generation on a 2-dimensional panel. The user could see the first, last, next, previous, and best tours of the current generation and an optimal tour, if one was known. The user could also use buttons to set the length of the GA's run.

The program was valuable as a prototype, but it was limited by its implementation in Visual Basic. First, the program can run only on a computer that runs the Visual Basic application. Second, it runs only on PCs, since Visual Basic is not a cross-platform application. Finally, Visual Basic is object-based rather than object-oriented, which limits the range of data structures available in any program.

The present project uses the Java programming language to implement the genetic algorithm. Java is cross-platform, web-enabled, and provides an implementation that uses rich object-oriented data structures. Further, Java is free and readily available.

4. The Java Application and Applet

This section describes the implementation of the algorithm in Java, beginning with the objects it manipulates and proceeding to its user interface and operation.

Objects

The application manipulates objects in three categories: cities, a graphical user interface (GUI), and the genetic algorithm. The city objects describe the target TSP instance: `XYCities` is an array of `point` objects, each of which specifies the x- and y-coordinates of one city. `Distance` is a two-dimensional array of integers; it holds the rounded Euclidean distances between each pair of cities.

The `MapPanel` and GA objects control the graphical user interface. `MapPanel` is a canvas on which are displayed dots that represent the cities and lines that represent links in a tour of the cities. The GA objects control the `MapPanel` object as well as several buttons and a text area that displays information about generations and tour lengths.

The GA objects are the most complex. An evolutionary algorithm's most important data structure is the population it maintains of chromosomes that encode candidate solutions to its target problem. Thus, the heart of the application is a `population` object. The population is an array of `chromosome` objects, each of which represents a candidate tour. The encoding of tours is straightforward: Integers label the cities of the target TSP instance, and a permutation of those integers represents the tour that visits the cities in the listed order and returns to the start city. The tour's total length is its fitness, which the GA seeks to minimize. A `chromosome` object, then, consists of such an array and an integer to hold its fitness.

The `Model` object contains an array of two `population` objects. One of these represents the current generation; the program repeatedly selects parent chromosomes and applies either crossover or mutation to them to build offspring chromosomes, which it places in the `Model` object's second population object. Associated with the `Model` object are functions that implement the 2-tournament selection scheme and the two genetic operators, described below.

The Interface

The graphical user interface divides the screen into four areas: display buttons, text display, GA buttons, and a tour map. The display buttons specify tours to display from the GA's current generation. The First, Next, Last, and Best buttons allow the user to step through and survey those tours. If an optimal tour for the target TSP instance is known, the Ideal button tells the program to plot it.

The text area displays information about the chromosome whose tour is currently displayed. It shows the number of the chromosome in the current generation and the length of its tour. The text area also shows the probability p that crossover will generate any one new chromosome; the probability that the program will apply mutation is then $(1 - p)$.

The user sets the probability of crossover with the P.Crossover button. The 1 Gen. And X Gen. buttons allow the user to specify the number of generations through which the GA will run. Figure 2 shows an instance of the GUI.

Program Operation

The program begins by initializing the city, GUI, and GA objects. It initializes one of the `Model` object's two populations with random permutations of the city numbers, thus with representations of random tours on the cities, and its evaluates each by finding the length of its tour. The length of the population's shortest tour is stored in its `Population` object.

To build each subsequent generation (in the other of the `Model`'s `Population` objects), the program first identifies and copies into the next generation the best chromosome, representing the shortest tour, in the current generation. For the remaining chromosomes in the next generation, the program chooses crossover or mutation according to the probability of crossover that P.Crossover specifies, selects one or two parents via 2-tournament selection, and applies the selected operator to the parent or parents. The program consults the `Distance` object to find the length of the tour each offspring chromosome represents; this value is its fitness. When the population of offspring is complete, it replaces its parents, and the process continues through a number of generations specified on the interface.

5. Crossover and Mutation

The interaction between an evolutionary algorithm's coding of candidate solutions and the operators that manipulate that coding is largely responsible for the algorithm's success or failure. The operators the TSP demonstration implements are not the most powerful known, but they do provide good performance on TSP instances of moderate size.

The crossover operator combines two parent chromosomes (that is, tours) to produce one offspring. It chooses a start city for the offspring at random; this city is labeled `current`. It identifies the cities in the parent chromosomes that are adjacent to the current one, and extends the tour by appending the city nearest to the current one that is found in both parents. If there is no such city, it appends the closest city found in either parent. If all the cities adjacent to the current one in the parents have already been included in the offspring tour, the crossover operator appends the nearest city not yet

listed. When the next city has been identified and appended, it becomes `current`, and the process continues until the tour is complete.

The mutation operator reverses a random segment within its one parent chromosome. That is, it generates two distinct random positions within the parent, and reverses the segment of the permutation between them. For example, the chromosome (9 4 3 **7 0 1 2 5** 8 6) might become (9 4 3 **5 2 1 0 7** 8 6). The effect is to reverse the order in which the tour visits some of the cities; it removes two edges from the tour and replaces them with two others.

6. Demonstration

Figures 2, 3, and 4 illustrate a typical run of the GA demonstration application. The TSP instance it addresses contains 30 cities, and is a standard, if small, example in the GA literature (Oliver, Smith, and Holland, 1987). Its shortest tour length is known to be 420.

Figure 2 shows an optimal tour on the 30 cities. Figure 3 shows the best tour in the GA's initial population of random tours. Figure 4 illustrates the GA's progress after 5,000

generations; note the improvement in the population's the best tour. For this run, the GA's population size was set to 50, and the probability that crossover generated any one offspring chromosome was set to 0.7.

7. Application Versus Applet

This program has been implemented as both an application and an applet. In an application, the programmer is responsible for creating an application object and configuring an environment in which the application runs. In an applet, the applet object is created and configured by a browser or the Java component AppletViewer, and HTML code is written to invoke the applet. Running the HTML code invokes the applet.

The differences between the application and applet code are minimal. One difference, however, is significant. Due to security constraints, Java file input and output are restricted in applets. The original application acquired city coordinates by reading them from a file. In an applet, this is not possible. Thus, the web-capable demonstration includes arrays of the 30 cities' x- and y- coordinates, from which it builds the array of `point` objects.

To run the applet, direct your browser to http://intrepid.mcs.stcloudstate.edu/ga/default.htm. Since the application and applet use Java 2 Swing components, a Java 2-compliant browser is required. Netscape 6 works. Internet Explorer does not.

The code is available via ftp://intrepid.mcs.stcloudstate.edu/ga. Download all the files in the directory. Compile the `*.java files`. The application is run from `XYCitiesTest`. The applet is run from `XYCitiesApplet.html`.

8. Conclusion

Active demonstrations contribute powerfully to any exposition or explanation. This paper has described a web-based demonstration of a simple genetic algorithm for the well-known traveling salesman problem. The GA encodes candidate tours as permutations of integers that label cities, selects chromosomes to be parents in 2-tournaments, applies either a greedy crossover or mutation by reversal of a random subtour to generate offspring chromosomes, and is generational with 1-elitism.

The demonstration is implemented as a Java applet, and so is available to anyone with a Java 2-capable browser. It illustrates its progress as it runs the genetic algorithm on a small TSP instance, and the user can set the demonstration's parameters and examine their effect on the GA's progress.

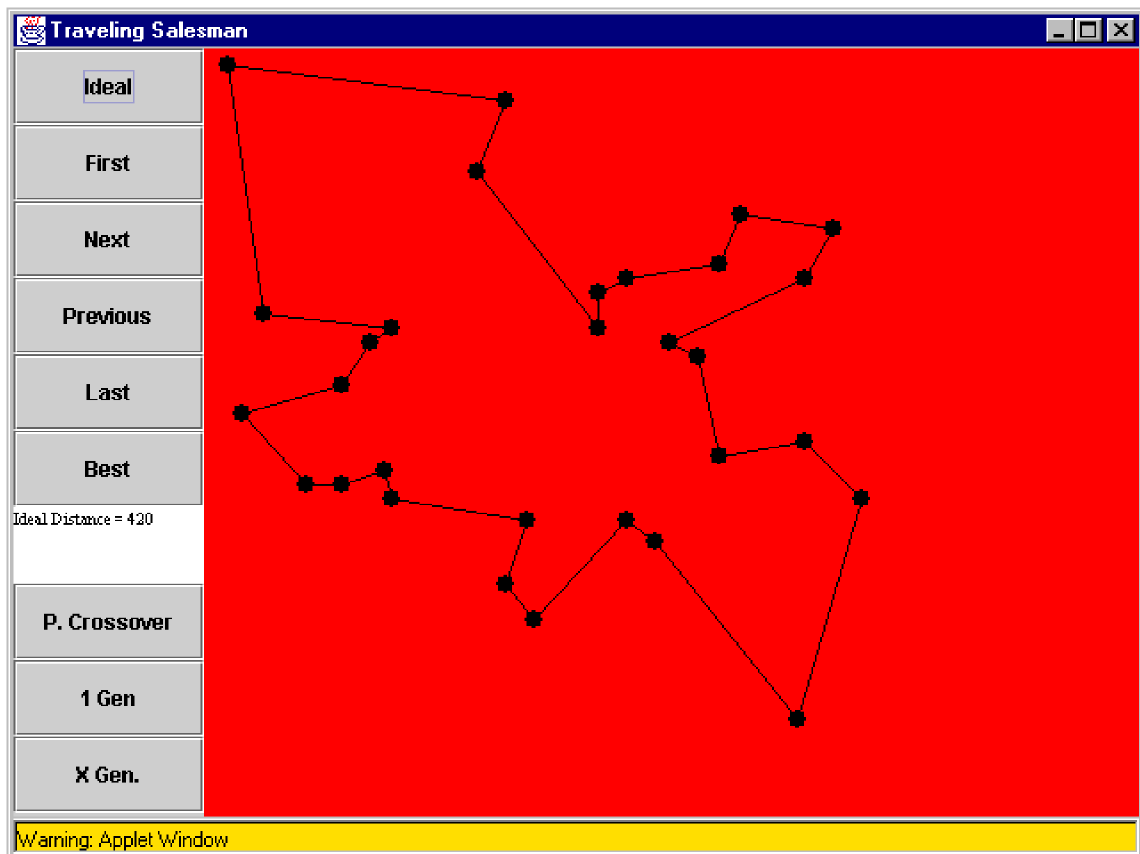

Figure 2. The GUI of the GA demonstration application. The tour shown is known to be optimal on its 30 cities; it has length 420.

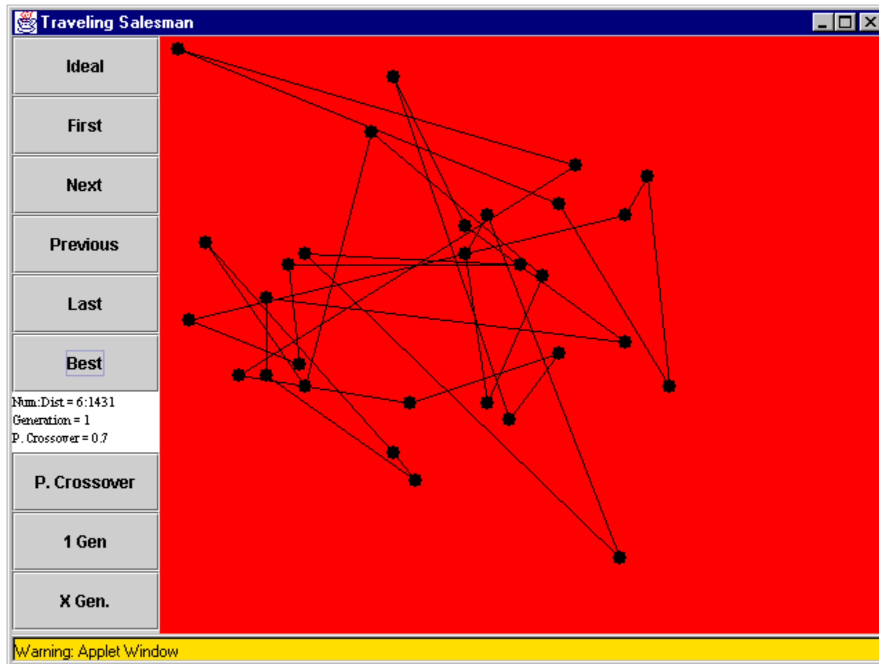

Figure 3. The GA demonstration application's interface at the beginning of a run. It shows the best tour in the initial population of random tours.

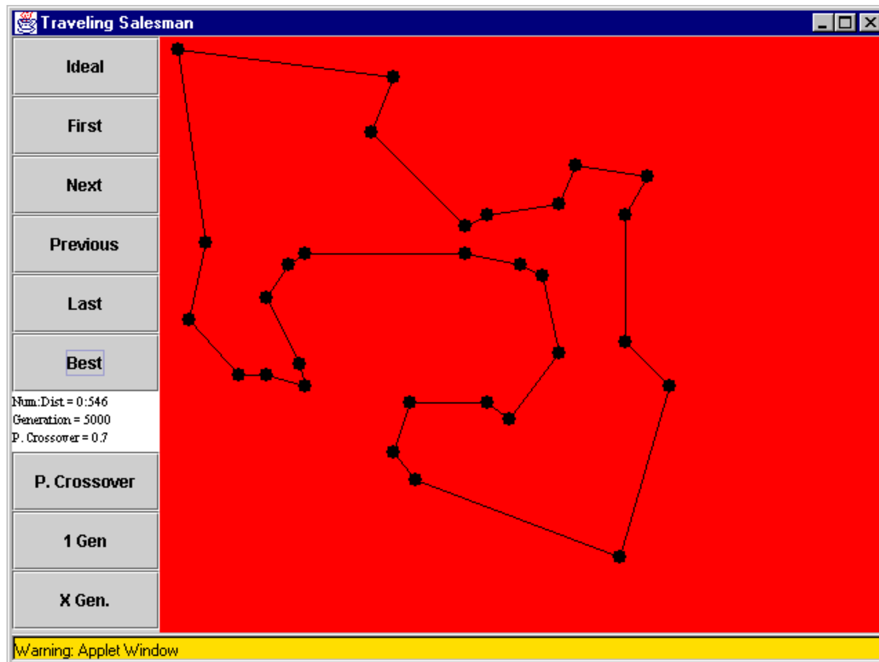

Figure 4. The GA demonstration application's interface after 5,000 generations. Note the improvement in the population's best tour. For this run, the population size was 50 and the probability of crossover, shown in the interface, was 0.7.

References

M.R.Garey and D.S.Johnson (1979*). Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H.Freeman and Company.

J.Greffenstette, R.Gopal, B.Rosmaita, and D.Van Gucht (1985). Genetic algorithms for the Traveling Salesman Problem. In *Proceedings of the First International Conference on Genetic Algorithms and their Applications* (ICGA'85) (J.J.Greffenstette, Ed.). Hillsdale, NJ: Lawrence Erlbaum Associates, pp.160-165.

P.Jog, J.Y.Suh, and D.van Gucht (1989). The effects of population size, heuristic crossover, and local improvement on a genetic algorithm for the traveling salesman problem. *In Proceedings of the Third International Conference on Genetic Algorithms* (ICGA'89) (J.D.Schaffer, Ed.). San Mateo, CA: Morgan Kaufmann, pp.110-115.

B.A.Julstrom (1995). Very greedy crossover in a genetic algorithm for the traveling salesman problem. *In Applied Computing 1995: Proceedings of the 1995 ACM Symposium on Applied Computing* (SAC'95) (K. M. George, Janice H. Carroll, Ed Deaton, Dave Oppenheim, and Jim Hightower, Eds.). New York: ACM Press, pp.324-328.

B.A.Julstrom and R.E.Mowe (1996). A Genetic Algorithm That Illustrates Its Progress on the Traveling Salesman Problem. In *Proceedings of the 29th Annual Small College Computing Symposium*. St. Cloud, MN: SCCS, pp 330-337.

Z.Michalewicz (1996). *Genetic Algorithms + Data Structures = Evolution Programs* (Third edition). Berlin: Springer-Verlag.

I.M.Oliver, D.J.Smith, and J.R.C.Holland (1987). A study of permutation operators on the Traveling Salesman Problem. *In Proceedings of the Second International Conference on Genetic Algorithms* (ICGA'87) (J.J.Greffenstette, Ed.). Hillsdale, NJ: Lawrence Erlbaum Associates, pp.224-230.