Genetic Algorithm and The Dynamic Connectivity Problem

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

In

Computer Science & Engineering

Under the Supervision of

DR. PARDEEP KUMAR

By

SANJAY SINGH, 111332

to

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled " Genetic Algorithm and the Dynamic Connectivity Problem ", submitted by Sanjay Singh in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date: 17/12/2014 **DR. PARDEEP KUMAR**

Assistance Professor (Senior Grade)

Acknowledgment

I have taken efforts in this project. However, it was not possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

I would also like to thank my advisor **Dr. Pardeep Kumar** for guiding me through the entire project. I would also like to express my gratitude toward faculty members of **Jaypee University of Information Technology** for their kind cooperation and support.

Without all of you, my project wouldn’t have such great values.

Abstract

Genetic Algorithm and The Dynamic Connectivity Problem

Genetic algorithms are an evolutionary technique that uses crossover and mutation operators to solve optimization problems using a survival of the fittest idea. They have been used successfully in a variety of different problems, including the traveling salesman problem.

In the traveling salesman problem we wish to find a tour of all nodes in a weighted graph so that the total weight is minimized. The traveling salesman problem is NP-hard but has many real world applications so a good solution would be useful.

Many different crossover and mutation operators have been devised for the traveling salesman problem and each give different results. We compare these results and find that operators that use heuristic information or a matrix representation of the graph give the best results.

CHAPTER 1

Introduction

Genetic algorithms are a relatively new optimization technique which can be applied to various problems, including those that are NP-hard. The technique does not ensure an optimal solution, however it usually gives good approximations in a reasonable amount of time. This, therefore, would be a good algorithm to try on the traveling salesman problem, one of the most famous NP-hard problems.

Genetic algorithms are loosely based on natural evolution and use a "survival of the fittest" technique, where the best solutions survive and are varied until we get a good result. We will explain genetic algorithms in detail, including the var- ious methods of encoding, crossover, mutation and evaluation in chapter 2. This will also include the operations used for the traveling salesman problem.

In chapter 3 we will explore the traveling salesman problem, what it is, real world applications, different variations of the problem and other algorithms and methods that have been tried.

Finally, in chapter 4 we will compare and contrast the different applications of genetic algorithms to the traveling salesman problem. In particular we will com- pare, where possible, their results for problems of specific sizes.

Genetic Algorithms

2.1 Introduction

In the field of artificial intelligence, a genetic algorithm (GA) is a search heuristic that mimics the process of natural selection. This heuristic (also sometimes called a metaheuristic) is routinely used to generate useful solutions to optimization and search problems.[1] Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such asinheritance, mutation, selection, and crossover.

Genetic algorithms find application in bioinformatics, phylogenetics, computational science, engineering, economics, chemistry, manufacturing,mathematics, physics, pharmacome trics and other fields.

The genetic algorithm process consists of the following steps:

- Encoding
- Evaluation
- Crossover
- Mutation
- Decoding

A suitable encoding is found for the solution to our problem so that each possible solution has a unique encoding and the encoding is some form of a string. The initial population is then selected, usually at random though alternative techniques using heuristics have also been proposed. The fitness of each individual in the population is then computed; that is, how well the individual fits the problem and whether it is near the optimum compared to the other individuals in the population. This fitness is used to find the individual's probability of crossover. If an individual has a high probability (which indicates that it is significantly closer to the optimum than the rest of its generation) then it is more likely to be chosen to crossover. Crossover is where the two individuals are recombined to create new individuals which are copied into the new generation. Next mutation occurs. Some individuals are chosen randomly to be mutated and then a mutation point is randomly chosen. The character in the corresponding position of the string is changed. Once this is done, a new generation has been formed and the process is repeated until some stopping criteria has been reached. At this point the individual which is closest to the optimum is decoded and the process is complete.

2.2 Basic Explanation

In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties
(its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.

The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitnessof every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

A standard representation of each candidate solution is as an array of bits.[2] Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations. Variable length representations may also be used, but crossover implementation is more complex in this case. Tree-like representations are explored in genetic programming and graph-form representations are explored in evolutionary programming; a mix of both linear chromosomes and trees is explored in gene expression programming.

Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions and then to improve it through repetitive application of the mutation, crossover, inversion and selection operators.

2.2.1 Encoding

The encoding process is often the most difficult aspect of solving a problem using genetic algorithms. When applying them to a specific problem it is often hard to find an appropriate representation of the solution that will be easy to use in the crossover process. Remember that we need to encode many possible solutions to create a population. The traditional way to represent a solution is with a string of zeros and ones. However genetic algorithms are not restricted to this encoding, as we will see in section 2.4. For now we will use a binary string representation.

Consider the problem defined above. Our possible solutions are obviously just numbers, so our representation is simply the binary form of each number. For instance, the binary representations of 12 and 7 are 1100 and 0111 respectively. Note that we added a zero to the beginning of the string 0111 even though it has no real meaning. We did this so that all the numbers in the set $\{0, \ldots, 15\}$ have the same length. These strings are called chromosomes and each element (or bit) of the string is called a gene.

We now randomly generate many chromosomes and together they are called the population.

2.2.2 Evaluation

The evaluation function plays an important role in genetic algorithms. We use the evaluation function to decide how 'good' a chromosome is. The evaluation function usually comes straight from the problem. In our case the evaluation function would simply be the function $f = -2x^2 + 4x - 5$, and because we are trying to maximize the function, the larger the value for f, the better. So, in our case, we would evaluate the function with the two values 7 and 12.

$$f(7) = -71$$

$$f(12) = -241$$

Obviously 7 is a better solution than 12, and would therefore have a higher fitness. This fitness is then used to decide the probability that a particular chromosome would be chosen to contribute to the next generation. We would normalize the scores that we found and then create a cumulative probability distribution. This is then used in the crossover process.

The stopping criteria is used in the evaluation process to determine whether or not the current generation and the best solution found so far are close to the global optimum. Various stopping criteria can be used, and usually more than one is employed to account for different possibilities during the running of the program: the optimal solution is found, the optimal solution is not found, a local optimum is found, etc. The standard stopping criteria that is used stops the procedure after a given number of iterations. This is so that if we do not find a local optimum or a global optimum and do not converge to any one point, the procedure will still stop at some given time. Another stopping criteria is to stop after the "best" solution has not changed over a specified number of iterations. This will usually happen when we have found an optimum - either local or global - or a point near the optimum. Another stopping criteria is when the average fitness of the generation is the same or close to the fitness of the 'best' solution.

2.2.3 Crossover

Crossover can be a fairly straightforward procedure. In our example, which uses the simplest case of crossover, we randomly choose two chromosomes to crossover, randomly pick a crossover point, and then switch all genes after that point. For example, using our chromosomes

$$\begin{aligned} v_1 &= 0111 \\ v_2 &= 1100 \end{aligned}$$

we could randomly choose the crossover point after the second gene

$$\begin{aligned} v_1 &= 01 \mid 11 \\ v_2 &= 11 \mid 00 \end{aligned}$$

. Switching the genes after the crossover point would give

$$v_1'' = 0100 = 4$$
$$v_2'' = 1111 = 15$$

We now have two new chromosomes which would be moved into the next population, called the next generation.

Not every chromosome is used in crossover. The evaluation function gives each chromosome a ’score’ which is used to decide that chromosome’s probability of crossover. The chromosomes are chosen to crossover randomly and the chromosomes with the highest scores are more likely to be chosen. We use the cumulative distribution created in the evaluation stage to choose the chromosomes. We generate a random number between zero and one and then choose which chromosome this corresponds to in our distribution. We do this again to get a pair, then the crossover is performed and both new chromosomes are moved into the new generation. This will hopefully mean that the next generation will be better than the last - because only the best chromosomes from the previous generation were used to create this generation. Crossover continues until the new generation is full.

A population of individualsare is maintained within search space for a GA, each representing a possible solution to a given problem. Each individual is coded as a finite length vector of components, or variables, in terms of some alphabet, usually the binary alphabet {0,1}. To continue the genetic analogy these individuals are likened to chromosomes and the variables are analogous to genes. Thus a chromosome (solution) is composed of several genes (variables). A fitness score is assigned to each solution representing the abilities of an individual to `compete'. The individual with the optimal (or generally near optimal) fitness score is sought. The GA aims to use selective `breeding' of the solutions to produce `offspring' better than the parents by combining information from the chromosomes.

We can also have two point crossover. In this case we randomly choose two crossover points and switch the genes between the two points. In our problem we could pick the points after the first gene and after the third gene.

$$v_1 = 0 \mid 11 \mid 1$$
$$v_2 = 1 \mid 10 \mid 0$$

to get

$$v_1'' = 0101 = 5$$
$$v_2'' = 1110 = 14$$

There are many different crossover routines, some of which will be explored later. We often need to change the crossover routine to make sure that we do not finish with an illegal chromosome - that is, an infeasible solution. In this way, crossover is very problem specific.

2.2.4 Mutation

Mutation is used so that we do not get trapped in a local optimum. Due to the randomness of the process we will occasionally have chromosomes near a local optimum but none near the global optimum. Therefore the chromosomes near the local optimum will be chosen to crossover because they will have the better fitness and there will be very little chance of finding the global optimum. So mutation is a completely random way of getting to possible solutions that would otherwise not be found.

Mutation is performed after crossover by randomly choosing a chromosome in the new generation to mutate. We then randomly choose a point to mutate and switch that point. For instance, in our example we had

$$v_1 = 0111$$

If we chose the mutation point to be gene three, v_1 would become

$$v_1'' = 0101$$

We simply changed the 1 in position three to a 0. If there had been a 0 in position three then we would have changed it to a 1. This is extremely easy in our example but we do not always use a string of zeros and ones as our chromosome. Like crossover, mutation is designed specifically for the problem that it is being used on.

Inversion is a different form of mutation [1]. It is sometimes used in appropriate cases and we will investigate some of these later. Here we will explain the inversion operator on our basic example.

The inversion operator consists of randomly choosing two inversion points in the string and then inverting the bits between the two points. For example

$$v_2 = 1100$$

We could choose the two points after gene one and after gene three.

$$v_2 = 1 \mid 10 \mid 0$$

Now, since there are only two genes between our inversion points, we then switch these two genes to give

$$v_2'' = 1010$$

If we had a larger chromosome, say

$$v_3 = 110100101001111$$

we could choose the inversion points after the third point and after the eleventh point.

$$v_3 = 110 \mid 10010100 \mid 1111$$

Now, we start at the ends of the 'cut' region and switch the genes at either end moving in. So we get

$$v_3'' = 110001010011111$$

Essentially we are just reversing (or inverting) the order of the genes in between the two chosen points.

2.3 Prisoner's Dilemma

Another completely different application of genetic algorithms is the prisoners' dilemma given in [7].

The prisoners' dilemma is a game where two prisoners are held in separate cells and cannot communicate. Each is asked to defect and betray the other and must decide whether to do so rather than cooperating with the other prisoner. If one prisoner defects he receives five points and the other receives zero. However, if both prisoners defect they each receive only one point. If both players cooperate they each receive three points. The problem that we wish to solve is to come up with a strategy to play the game successfully. Michalewicz ([7]) has devised an algorithm to do this. He uses the past three plays to decide what to do for the current play. Each history then consists of three combinations of C and D, ie, CC, CD, DC or DD, so there are $4 \times 4 \times 4 = 64$ different histories. Each player has a particular set play for each of these different histories, and together these plays will give a strategy (or chromosome) also of length 64. For instance, if we had the histories

$$\ldots, (CD)(DC)(CC), (DD)(DC)(CD), (CD)(CD)(CD), \ldots$$

then the chromosome could look like

$$\ldots, C, C, D, \ldots$$

meaning that if the history happened to be (CD)(DC)(C C) this player would co-operate on the next turn. If the history was (DD)(DC)(C D) the player would also cooperate on the next turn, but if the history was (CD)(C D)(C D) the player would defect on the next turn.

Now we can find a play for each different history but what happens at the start of the game when there is no history? Each different player (or chromosome) is given an hypothetical history so that it can generate its first play. That is, we add six more genes to the start of the chromosome to act as the 'last' three plays. The player then uses the play that has been assigned to that particular history. So we end up with a string of 70 genes as our chromosome.

The fitness of each chromosome is found by playing against other players. The usual one or two point crossover works for these chromosomes and mutation is also just the usual mutation routine. Notice that the chromosomes are simply binary strings but we have the letters C and D rather than the binary digits 0 and 1.

2.4 Encoding

In this section we will investigate possible ways to encode different problems. In particular, the traveling salesman problem will be examined.

We have already seen the basic way of encoding a problem using a string of zeros and ones, which represent a number in its binary form. We can also use a string of letters, for example ABCDE, or a string of integers, 12345, or just about any string of symbols as long as they can be decoded into something more meaningful.

Imagine we had a problem involving a graph and we needed to encode the adjacency list of the graph. We could create the adjacency matrix, which consists of a one in the i, jth position if there is an arc from node i to node j and a zero otherwise. We could then use the matrix as is or we else could concatenate the rows of the matrix to create one long string of zeros and ones. Notice this time, however, the string is not a binary representation of a number.

This leads us to the first method of encoding a tour of the traveling salesman problem. We do have a graph such as the one described above and we can encode it in the same way, only our matrix will have a one in the i, jth position if there is an arc from node i to node j in the tour and a zero otherwise. For example, the

□ □

represents the tour that goes from city 1 to city 3, city 3 to city 2 and city 2 to city 1. This encoding is known as matrix representation and is given in [3] and [7].

The traveling salesman problem can also be represented by a string of integers in two different ways. The first (given in [9], [3], [2] and [7]) is by the string

$$v = a_1 a_2 \ldots a_n$$

which implies that the tour goes from a_1 to a_2 to a_3, etc and from a_n back to a_1. Notice that the strings $v_1 = 1234$ and $v_2 = 2341$ are equivalent in this representation.

The second way to represent the traveling salesman problem is with cycle notation ([7]), with an integer string

$$v = b_1 b_2 \ldots b_n$$

where the tour goes from city i to city b_i. That is, the string $v_1 = 3421$ means that the tour goes from city 1 to city 3, city 3 to city 2, city 2 to city 4 and city 4 to city 1. Note that not every possible string here represents a legal tour, where a legal tour is a tour that goes to every city exactly once and returns to the first city. It is possible for us to have a string that represents disjoint cycles, for example, $v_2 = 3412$ implies that we go from city 1 to city 3 and back to city 1 and from city 2 to city 4 and back to city 2.

2.5 Crossover

Several crossover methods have been developed for the traveling salesman problem. In this section we describe several of them. We shall compare these methods in chapter 4.

We start by looking at partially matched crossover (PMX) ([4], [1], [2] and [7]). Recall the two-point crossover and assume we were to use this with the integer representation defined for the traveling salesman problem in section 2.4. If we performed a two-point crossover on the chromosomes

$$\begin{aligned} v_1 &= 1234 \mid 567 \mid 8 \\ v_2 &= 8521 \mid 364 \mid 7 \end{aligned}$$

we would get

$$\begin{aligned} v_1' &= 1234 \mid 364 \mid 8 \\ v_2' &= 8521 \mid 567 \mid 7 \end{aligned}$$

which are obviously illegal because v_1' does not visit cities 5 or 7 and visits cities 4 and 3 twice. Similarly v_2' does not visit cities 4 or 3 and visits cities 5 and 7 twice. PMX fixes this problem by noting that we made the swaps $3 \leftrightarrow 5$, $6 \leftrightarrow 6$ and $4 \leftrightarrow 7$ and then repeating these swaps on the genes outside the crossover points, giving us

$$\begin{aligned} v_1'' &= 12573648 \\ v_2'' &= 83215674 \end{aligned}$$

In other words, we made the swaps, $3 \leftrightarrow 5, 6 \leftrightarrow 6, 4 \leftrightarrow 7$ and the other elements stayed the same. v_1'' and v_2'' still consist of parts from both the parents v_1 and v_2 and are now both legal.

This crossover would make more sense when used with the cycle representation, since in this case it would preserve more of the structure from the parents. If, as in our example, we used the first integer representation, the order that the cities were visited would have changed greatly from the parents to the children - only a few of the same edges would have been kept. With cycle notation a lot more of the edges would have been transfered. However, if we use this crossover routine with cycle representation we do not necessarily get a legal tour as a result. We would need to devise a repair routine to create a legal tour from the solution that the crossover gives us, by changing as little as possible in order to keep a similar structure.

Cycle crossover (CX) ([4], [2] and [7]) works in a very different way. First of all, this crossover can only be used with the first representation we defined, that is, the chromosome $v = 1234$ implies that we go from city 1 to city 2 to city 3 to city 4. This time we do not pick a crossover point at all. We choose the first gene from one of the parents

$$v_1 = 12345678$$

$$v_2 = 85213647$$

say we pick 1 from v_1

$$v_1'' = 1-------$$

We must pick every element from one of the parents and place it in the position it was previously in. Since the first position is occupied by 1, the number 8 from v_2 cannot go there. So we must now pick the 8 from v_1.

$$v_1'' = 1------8$$

This forces us to put the 7 in position 7 and the 4 in position 4, as in v_1.

$$v_1'' = 1--4--78$$

Since the same set of positions is occupied by 1, 4, 7, 8 in v_1 and v_2, we finish by filling in the blank positions with the elements of those positions in v_2. Thus

$$v_1'' = 15243678$$

2 1

$$v''_2 = 82315647$$

This process ensures that each chromosome is legal. Notice that it is possible for us to end up with the offspring being the same as the parents. This is not a problem since it will usually only occur if the parents have high fitnesses, in which case, it could still be a good choice.

Order crossover (OX) ([2] and [7]) is more like PMX in that we choose two crossover points and crossover the genes between the two points. However instead of repairing the chromosome by swapping the repeats of each node also, we simply rearrange the rest of the genes to give a legal tour. With the chromosomes

$$\begin{aligned} v_1 &= 135 \mid 762 \mid 48 \\ v_2 &= 563 \mid 821 \mid 47 \end{aligned}$$

we would start by switching the genes between the two crossover points.

$$\begin{aligned} v''_1 &= - - - \mid 821 \mid - - \\ v''_2 &= - - - \mid 762 \mid - - \end{aligned}$$

We then write down the genes from each parent chromosome starting from the second crossover point.

$$\begin{aligned} v_1 &: \ 48135762 \\ v_2 &: \ 47563821 \end{aligned}$$

then the genes that were between the crossover points are deleted. That is, we would delete 8, 2 and 1 from the v_1 list and 7, 6 and 2 from the v_2 list to give

$$\begin{aligned} v_1 &: \ 43576 \\ v_2 &: \ 45381 \end{aligned}$$

which are then replaced into the child chromosomes, starting at the second crossover point.

$$v_1'' = 57682143$$

$$v_2'' = 38176245$$

Next we consider matrix crossover (MX) ([7] and [3]). For this we have a matrix representation where the element i, $\mathbf{j}$ is 1 if there is an edge from node i to node $\mathbf{j}$ and 0 otherwise. Matrix crossover is the same as one- or two-point crossover. If we have the matrices

we choose the crossover points after the first column and after the second column and crossover the columns to give

We now have multiple 1's in some rows and some rows without any 1's at all. We fix this by moving one of the 1's from the row wth the multiples to a row without any 1's. We choose which 1 to move randomly.

Now notice in A^{00} we have $a \rightarrow a$ and $b \rightarrow c \rightarrow b$. So we have two different cycles. We can fix this by cutting and reconnecting the cycles. Obviously we would cut the edge from a to a and one of the edges between b and c and connect a to b and a to c. When we have a choice as to which nodes we connect (our example was small enough so that we do not have a choice) we choose the ones that exist in one of the parents to try to maintain the structure as much as possible.

Modified order crossover (MOX) ([9]) is similar to order crossover. We randomly choose one crossover point in the parents and as usual, leave the genes before the crossover point as they are. We then reorder the genes after the crossover point in the order that they appear in the second parent chromosome. If we have

$$v_1 = 123 \mid 456$$
$$v_2 = 364 \mid 215$$

we would get

$$v_1' = 123 \mid 645$$
$$v_2' = 364 \mid 125$$

The crossovers explored so far concentrate on the position of the city in the tour whereas it is really the edges that are the most important part of the traveling salesman’s tour, since they define the costs. So what we really want is to deal with edges rather than the positions of each city.

Grefenstette(1981, cited in [4]) has devised a crossover routine which picks each node from one of those which is incident to the current node in one of the parents. We do this by creating an edge list for each node. The chromosomes

$$v_1 = 123456$$

$$v_2 = 364215$$

have edge list

node 1 :	256
node 2 :	134
node 3 :	2456
node 4 :	2356
node 5 :	1346
node 6 :	1345

We first choose one of the initial nodes from one of the parents, i.e., 1 or 3 in this example. We choose the one that has the least number of incident nodes, or if they have the same number we randomly choose one. We then consider the nodes incident to node 1 since this is the node we first chose. Again we choose the node with the least number of previously unchosen incident nodes. So we choose node 2. We continue this process of considering nodes which have not previously been selected. If we encounter a situation in which we cannot choose a node that has not previously been selected we randomly choose a previously unselected node. This means that we will get a node which is not incident to our current node in one of the parents, but unfortunately this is unavoidable. So our parent chromosomes could give the offspring

$$v_1' = 124365$$

Notice that we were successful in being able to choose nodes that were incident in one of the parents at all times. We also only get one offspring from this crossover so we need to do twice as many crossovers to create the new generation.

We also have crossover operators that use heuristic information. The heuristic crossover ([4]) chooses a random node to start at and then considers the two edges leaving the current node in the parent chromosomes and picks the shortest edge that does not introduce a cycle. If both edges introduce a cycle we choose a random edge that does not do so.

2.6 Mutation

First we will look at the 2-opt operator ([4]). We randomly select two edges (a, b) and (c, d) from our tour and check if we can connect these four nodes in a different manner that will give us a lower cost. To do this we check if

$$c_{ab} + c_{cd} > c_{ac} + c_{db}$$

If this is the case we replace the edges (a, b) and (c, d) with the edges (a, c) and (d, b). Note that we assume that a, b, c and d appear in that specific order in the tour even if b and c are not connected.

We also have a 3-opt operator ([4]) which looks at three random edges instead of two. If we have edges (a, b), (c, d) and (e, f), we check if

$$c_{ab} + c_{cd} + c_{ef} > c_{ac} + c_{be} + c_{df}$$

If it is we replace (a, b), (c, d) and (e, f) with the edges (a, c), (b, e) and (d, f).

The Or-opt operator ([4]) is similar to the 3-opt. We randomly choose a set of connected nodes and check if this string can be inserted between two other connected nodes to give us a reduced cost. We can calculate this by finding the total cost of the edges being inserted and the total cost of the edges being removed. If the cost of the edges being removed is greater than the cost of those being inserted the switch is made.

Another three mutation operators (given in [7]) are insertion where we randomly select a city and insert it in a random place. Displacement is where we select a subtour and insert it in a random place. We also have reciprocal exchange where we choose two random cities and swap them.

Chapter 3

The Traveling Salesman Problem

3.1 Introduction

The Travelling Salesman Problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

TSP is a special case of the travelling purchaser problem.

In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether the graph has any tour shorter than L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (or perhaps exponentially) with the number of cities.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.[1]

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows may be imposed.

3.2 Applications

The traveling salesman problem has many different real world applications, making it a very popular problem to solve. Here we explain a few of these given in [8] and [6]. For example, some instances of the vehicle routing problem can be modelled as a traveling salesman problem. Here the problem is to find which customers should be served by which vehicles and the minimum number of vehicles needed to serve each customer. There are different variations of this problem including finding the minimum time to serve all customers. We can solve some of these problems as the TSP.

The problem of computer wiring can also be modelled as a TSP. We have several modules each with a number of pins. We need to connect a subset of these pins with wires in such a way that no pin has more than two wires attached to it and the length of the wire is minimized.

An application found by Plate, Lowe and Chandrasekaran (cited in [8]) is overhauling gas turbine engines in aircraft. Nozzle-guide vane assemblies, consisting of nozzle guide vanes fixed to the circumference, are located at each turbine stage to ensure uniform gas flow. The placement of the vanes in order to minimize fuel consumption can be modelled as a symmetric TSP.

The scheduling of jobs on a single machine given the time it takes for each job and the time it takes to prepare the machine for each job is also TSP. We try to minimize the total time to process each job.

A robot must perform many different operations to complete a process. In this application, as opposed to the scheduling of jobs on a machine, we have precedence constraints. This is an example of a problem that cannot be modelled by a TSP but methods used to solve the TSP may be adapted to solve this problem.

3.3 Different Forms of the Problem

There are many different variations of the traveling salesman problem. First we have the shortest Hamiltonian path problem ([6] and [8]). If we have a graph where each edge has a weight and two nodes v_s and v_t are given we must find the shortest Hamiltonian path from v_s to v_t. If we add an edge from v_t to v_s and give it weight $-M$ where M is large and positive, our optimal TSP tour will always include this edge (because it will reduce the cost of the tour) and will therefore solve the Hamiltonian problem.

The asymmetric traveling salesman problem ([8]) is when the cost of traveling from city i to city j is not the same as the cost from city j to city i. This can be solved in the same way as the standard TSP if we apply certain edge weights that ensure that there is a Hamiltonian cycle in the graph.

The multisalesmen problem ([8] and [6]) is the same as the standard TSP except that we have more than one salesman. We need to decide where to send each salesman so that every city is visited exactly once and each salesman returns to the original city.

The bottleneck traveling salesman problem ([8] and [6]) is where we want to minimize the largest edge cost in the tour instead of the total cost. That is, we want to minimize the maximum distance the salesman travels between any two adjacent cities.

The time dependent traveling salesman problem ([6]) is the same as the standard traveling salesman problem except we now have time periods. The cost c_{ijt} is the cost of traveling from node i to node j in time period t. We want to minimize

3.4 Methods of Solving the TSP

Homaifar ([3]) states that "one approach which would certainly find the optimal solution of any TSP is the application of exhaustive enumeration and evaluation. The procedure consists of generating all possible tours and evaluating their corresponding tour length. The tour with the smallest length is selected as the best, which is guaranteed to be optimal. If we could identify and evaluate one tour per nanosecond (or one billion tours per second), it would require almost ten million years (number of possible tours $= 3.2 \times 10^{23}$) to evaluate all of the tours in a 25-city TSP."

Obviously we need to find an algorithm that will give us a solution in a shorter amount of time. As we said before, the traveling salesman problem is NP-hard so there is no known algorithm that will solve it in polynomial time. We will probably have to sacrifice optimality in order to get a good answer in a shorter time. Many algorithms have been tried for the traveling salesman problem. We will explore a few of these in this section.

Greedy Algorithms ([8]) are a method of finding a feasible solution to the traveling salesman problem. The algorithm creates a list of all edges in the graph and then orders them from smallest cost to largest cost. It then chooses the edges with smallest cost first, providing they do not create a cycle. The greedy algorithm gives feasible solutions however they are not always good.

The Nearest Neighbor ([8]) algorithm is similar to the greedy algorithm in its simple approach. We arbitrarily choose a starting city and then travel to the city closest to it that does not create cycle. We continue to do this until all cities are in the tour. This algorithm also does not always give good solutions because often the last edge added to the tour (that is, the edge e_{n1} where n is the number of cities) can be quite large.

A minimum spanning tree ([8] and [6]) is a set of $n - 1$ edges (where again n is the number of cities) that connect all cities so that the sum of all the edges used is minimized. Once we have found a minimum spanning tree for our graph we can create a tour by treating the edges in our spanning tree as bidirectional edges. We then start from a city that is only connected to one other city (this is known as a 'leaf' city) and continue following untraversed edges to new cities. If there is no untraversed edge we go back along the previous edge. We continue to do this until we return to the starting city. This will give us an upper bound for the optimal traveling salesman tour. Note, however, that we will visit some cities more than once. We are able to fix this if whenever we need to traverse back to a city we have already been to, we instead go to the next unvisited city. When all cities have been visited we go directly back to the starting city.

Chapter 4

Genetic Algorithms as a Method of Solving the Traveling Salesman Problem

4.1 Introduction

The different forms of encoding, crossover and mutation that we have seen so far can be combined to give various genetic algorithms that can be used to solve the traveling salesman problem. Obviously some crossover routines can only be used with a certain form of encoding so we do not have too many different genetic algorithms to explore. Also, only certain methods have been attempted, so we will only look at these. Finally, we will keep in mind that these programs have been tested on different problems and it will therefore be difficult to compare them to each other.

4.2 Comparison of Methods

First we will note the best known solutions for particular problems given in [3]. For the 25 city problem the best known solution is 1,711, the 30 city problem is 420, the 42 city problem is 699, the 50 city problem is 425, the 75 city problem is 535, the 100 city problem is 627, the 105 city problem is 14,383 and the 318 city problem is 41,345. These problems are standard problems with set edge costs that can be used to test new algorithms.

We will now consider pure genetic algorithms with no heuristic information used.

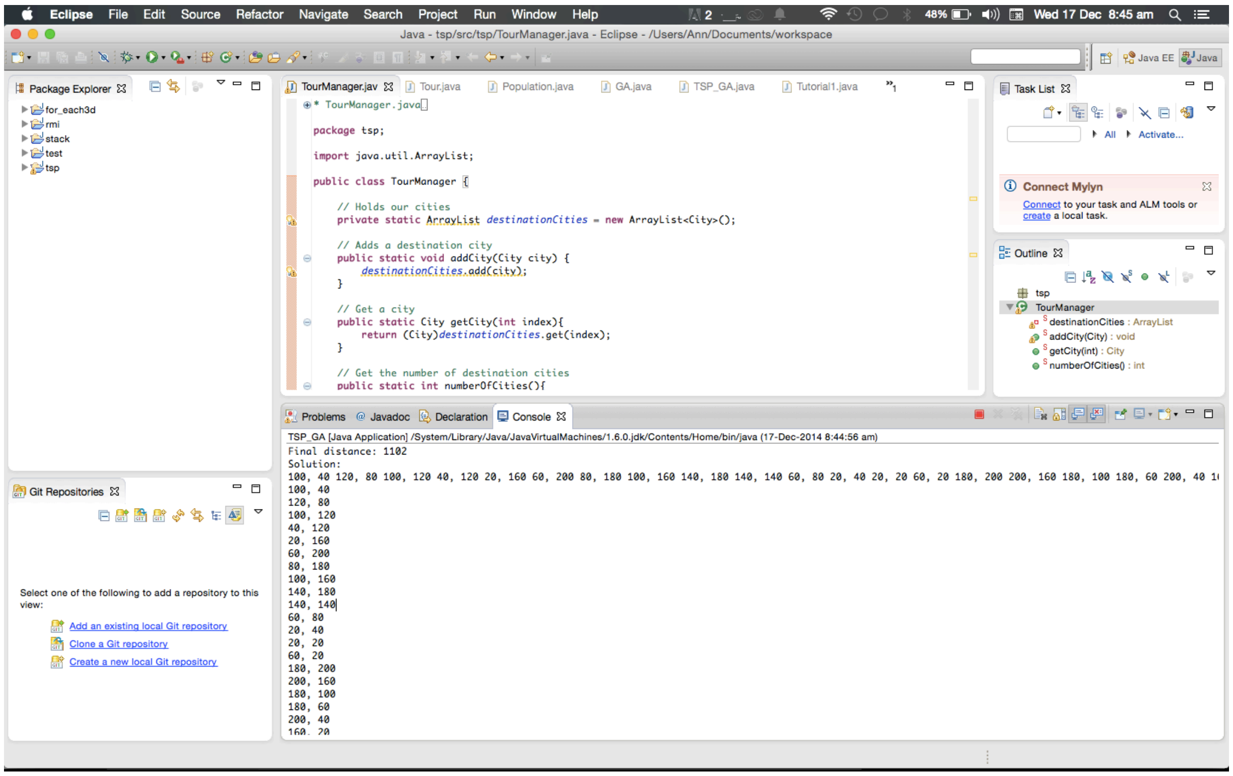
Consider the partially modified crossover (PMX) with the tour notation and no mutation operator. Jog ([4]) found that this algorithm gave a tour who's length was ten percent larger than the known optimum for the 33 city problem. For the 100 city problem, the result was 210 percent larger than the known optimum. Homaifar ([3]) states that the best tour length of this same algorithm is 498 for the 30 city problem.

The algorithm using order crossover gives a better performance, giving a result of length 425 for the 30 city problem, while cycle crossover only gives a tour of length 517 for the same problem. The best known solution for the 30 city problem is 420 so order crossover seems to be the best so far ([3]).

Now we will consider the matrix crossover method ([3]). We will use the two point matrix crossover method as well as inversion. Homaifar found that this method performed well with 30, 50, 75, 100 and 318 city problems giving tours of lengths 420, 426, 535, 629 and 42154 respectively, which are all less then two percent above the best known solution. So it seems that the idea of using edges rather than the position of cities as our variable is promising. This makes sense, as it is the edge which holds the costs and we want to pick which edges to use to connect all cities. Note however, that the matrix representation takes more space to store and also more computation time for the crossover and mutation processes than the integer representation and basic crossovers.

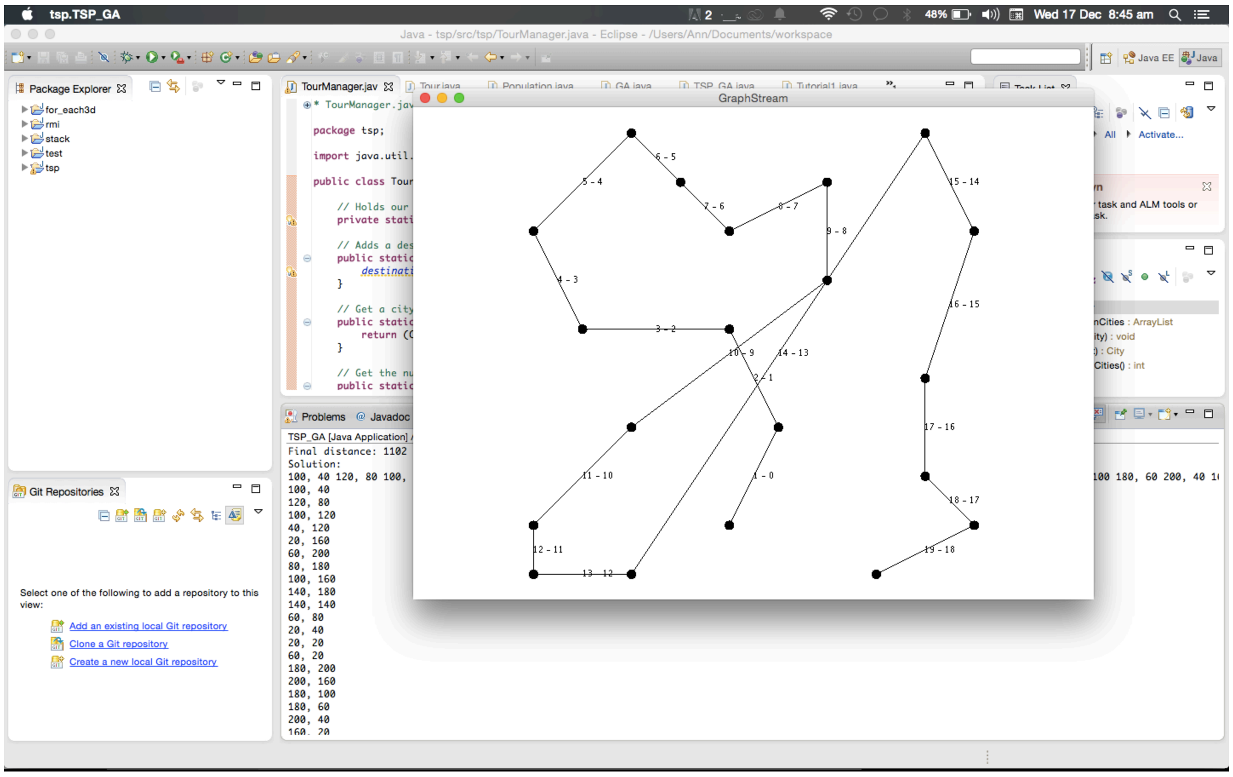
Homaifar ([3]) also tested an algorithm that uses only the 2-opt mutation operator and no crossover. This also performed decently, however not as well as the previous case where we used matrix crossover. In particular, it performed worse with problems where the number of cities is large.

Jog's ([4]) heuristic algorithms also performed well. The heuristic crossover, when combined with the 2-opt and Or-opt mutation operators sometimes gives the best known solution for that particular problem, and otherwise returns a solution very close to that value.


Eclipse File Edit Source Refactor Navigate Search Project Run Window Help
Wed 17 Dec 8:45 am
Java - tsp/src/tsp/TourManager.java - Eclipse - /Users/Ann/Documents/workspace
Package Explorer
for_each3d
rmi
stack
test
tsp
TourManager.jav
Tour.java
Population.java
GA.java
TSP_GA.java
Tutorial1.java
Task List
All
Activate...
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Outline
tsp
TourManager
destinationCities : ArrayList
addCity(City) : void
getCity(int) : City
numberOfCities() : int
Git Repositories
Select one of the following to add a repository to this view:
Add an existing local Git repository
Clone a Git repository
Create a new local Git repository
Problems
Javadoc
Declaration
Console

```java
* TourManager.java

package tsp;

import java.util.ArrayList;

public class TourManager {

    // Holds our cities
    private static ArrayList destinationCities = new ArrayList<City>();

    // Adds a destination city
    public static void addCity(City city) {
        destinationCities.add(city);
    }

    // Get a city
    public static City getCity(int index){
        return (City)destinationCities.get(index);
    }

    // Get the number of destination cities
    public static int numberOfCities(){
```

```
TSP_GA [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (17-Dec-2014 8:44:56 am)
Final distance: 1102
Solution:
100, 40 120, 80 100, 120 40, 120 20, 160 60, 200 80, 180 100, 160 140, 180 140, 140 60, 80 20, 40 20, 20 60, 20 180, 200 200, 160 180, 100 180, 60 200, 40 1
100, 40
120, 80
100, 120
40, 120
20, 160
60, 200
80, 180
100, 160
140, 180
140, 140
60, 80
20, 40
20, 20
60, 20
180, 200
200, 160
180, 100
180, 60
200, 40
160, 20
```



Select one of the following to add a repository to this view:

Add an existing local Git repository
Clone a Git repository
Create a new local Git repository

```
* TourManager.jav

package tsp;

import java.util.

public class Tour

    // Holds our
    private stati

    // Adds a des
    public static
        destinati
    }

    // Get a city
    public static
        return (C
    }

    // Get the nu
    public static
```

TSP_GA [Java Application]

```
Final distance: 1102
Solution:
100, 40 120, 80 100,
100, 40
120, 80
100, 120
40, 120
20, 160
60, 200
80, 180
100, 160
140, 180
140, 140
60, 80
20, 40
20, 20
60, 20
180, 200
200, 160
180, 100
180, 60
200, 40
160, 20
```

Chapter 5

Conclusion

Genetic algorithms appear to find good solutions for the traveling salesman problem, however it depends very much on the way the problem is encoded and which crossover and mutation methods are used. It seems that the methods that use heuristic information or encode the edges of the tour (such as the matrix representation and crossover) perform the best and give good indications for future work in this area.

Overall, it seems that genetic algorithms have proved suitable for solving the traveling salesman problem. As yet, genetic algorithms have not found a better solution to the traveling salesman problem than is already known, but many of the already known best solutions have been found by some genetic algorithm method also.

It seems that the biggest problem with the genetic algorithms devised for the traveling salesman problem is that it is difficult to maintain structure from the parent chromosomes and still end up with a legal tour in the child chromosomes. Perhaps a better crossover or mutation routine that retains structure from the parent chromosomes would give a better solution than we have already found for some traveling salesman problems.

References

[1] C.H. Papadimitriou and K. Steglitz. “Combinatorial Optimization: Algorithms Complexity”. Prentice Hall of India Private Limited, India, 2007.

[2] C.P. Ravikumar. "Solving Large-scale Travelling Salesperson Problems on Parallel Machines”. Microprocessors and Microsystems 16(3), pp. 149-158, 2009.

[3] Abdollah Homaifar, Shanguchuan Guan, and Gunar E. Liepins. Schema analysis of the traveling salesman problem using genetic algorithms. Complex Systems, 6(2):183–217, 2005.

[4] Prasanna Jog, Jung Y. Suh, and Dirk Van Gucht. Parallel genetic algorithms salesman problem. SIAM Journal of Optimization, 1(4):515–529, 2012.

[5] Z.H. Ahmed and S.N.N. Pandit. “The travelling salesman problem with precedence constraints”. Opsearch 38, pp. 299-318, 2001 [6] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. The Traveling Salesman. JohnWiley and Sons, 2000.

[