A Fault-Tolerant Approach for the Shortest Path Algorithm in Large Spectrum Graphs

By

Joseph Fares

M.S., Computer Science, Lebanese American University, 2007

Thesis submitted in partial fulfilment of the requirements for the Degree of Master of Science in Computer Science

Division of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

June 2007

LEBANESE AMERICAN UNIVERSITY

School of *Arts* and Sciences

Thesis Approval

Student Name	**JOSEPH FARES**	I.D.#: **198731470**
Thesis Title:	***A FAULT-TOLERANT APPROACH FOR THE SHORTEST PATH ALGORITHM IN LARGE SPECTRUM FILES***	

Program:	**Computer Science**
Division /Dept:	**Computer Science and Mathematics**
School:	**School of Arts and Sciences, Byblos**
Approved by:	
Thesis Advisor:	***HAIDAR M. HARMANANI*** [redacted]
Member	***JEAN TAKCHE*** [redacted]
Member	***MOUNJED MOUSSALLAM*** [redacted]
Member	
Date:	JUNE 29, 2007

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Joseph Fares

Signature: [redacted] Date: June 29, 2007

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

Acknowledgment

I would like to thank my advisor Dr. Haidar Harmanani for his guidance throughout my Thesis work. A thank is also to Mr. Mounjid Mousallem and Dr. Jean Takchi for being on my thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my friends and family for their long support.

Abstract

This work proposes a solution for the shortest path algorithm for large spectrum graphs. The problem is of particular interest for applications in computer networks as well as in road networks. We tackle this problem in two stages. In the first stage an evolutionary algorithm for networks-based applications is formulated with a special interest emphasis on fault-tolerance, an important issue due to dynamic changes in routing configuration. In the second phase, we tackle large spectrum graphs where the evolutionary algorithm is applied using database constructs. Thus, the graph is stored in a database and the evolutionary algorithm is formulated using SQL queries. The operators as well as the selection operators are all database-based. The algorithm is attempted on the DIMACS challenge for the USA routes and favourable results are reported.

Contents

List of Figures

List of Tables

Chapter 1

Shortest Path Problems

1.1 Introduction

The shortest path problem is based on detecting the minimal route from source to destination in terms of distance, time or cost. As for multihop networks or road networks, finding the shortest path or the best route is quite vital. Several search algorithms for the shortest path problems have been reported such as: breadth first search, Bellman Ford, Dijkstra, and the A* algorithm [1]. These algorithms are very effective on static networks. But as the graph that represents the network increases in size, the problem's complexity increase such that the problem becomes intractable and the computation time intolerable [2]. Another important issue is the rapid change of a link's weight as a function of time. For example, suppose an accident occurred at rush time hour in a congested street of the city. This accident will not only increase the weight of that link but also will affect the whole network. These changing networks are known as dynamic networks and these unexpected modifications on this kind of networks are known as rerouting [31]. Only algorithms based on random techniques methods, such as genetic algorithm, simulating annealing or tabu search are able to solve the nuisance of increasing dimensions as well as its capabilities to adjust with the new network alterations [3, 4].

1.2 Shortest Path problem

More than 40 years were spent on studying the shortest path problems in different fields of interest such as computer science and transportation networks. All computer scientists focusing on this field came out with amazing results through very powerful algorithms. Most of these algorithms are based on the principle of triangular inequality for finding the shortest path route in a graph. This shortest path is found by relaxing the edges stemming from the current node to all its neighbors. Approximately all greedy shortest path algorithms follow the same standard procedure, here follows a description of a general greedy shortest path algorithm:

```
Step1: Initialize single source (V,s)   where V = set of vertices and s = source
   For each v ∈ V do
      d[v] ← ∞       'all degrees or labels are set to infinity
      p[v] ← nil     ' all parents are nil
   d[s] ← 0          ' degree of source is 0
   iteration ← 0

Step2: number of iterations (according to: V-1 or a non empty queue Q)
   #iterations ← V- 1
   Q ← V[G]          ' queue = vertices in the graph G

Step3:
   u ← extract(Q)    'remove  node from the queue Q
   iteration ← iteration + 1

Step4: Relaxation of edges
   For each edge (u,v) ∈ E
      If d[v] > d[u] + w(u,v)   then ' w(u,v) is the edge weight from u to v
         d[v] ← d[u] + w(u,v)
         p[v] ← u
      end if

Step5: stop when either Q is empty or when iteration > #iterations
   If (iteration < #iterations) OR Q ≠ø   then Go to step3
```

Figure 1.1 General greedy shortest path algorithm

Two variations for this kind of algorithms exist, the label setting algorithm and the label correcting algorithm [5]. The difference between both algorithms depends on the kind of data structure used (i.e. Queue, heap or list). In the case of label setting, the node with the smallest degree is extracted from the queue. When it comes to find the shortest path between a single source and a single destination, the algorithm will stop execution as soon as the label of the destination node has been changed. This is known as one to one search mode. On the other hand, the label correcting algorithm does not give the shortest path between a source and a destination unless the shortest path for each vertex in the graph is calculated. This is known as one to all search mode. The label correcting is useful when more than one shortest path emanating from the source vertex is needed.

1.3 Heuristic Shortest Path

Heuristic shortest path is based on knowing in advance the locations of the source and the destination nodes. This helps in minimizing the search area, partitioning the graph, and traversing the specified edges of the graph [6]. The two algorithms mentioned in the previous section are inefficient with medium to large spectrum graphs since the search is done in a forward way without having a prior knowledge of the destination node position. Furthermore, all nodes falling from source to destination are visited even though they might not be elements of the shortest path. The goal was to build efficient algorithms that could shrink the search area based on some based knowledge.

These studies led to two important algorithms that operate by limiting the search area namely branch pruning and A* algorithm.

1.3.1 A* Algorithm

A* stores the nodes that have a low probability of being visited in the same scan set. On the other hand, A* is based on a heuristic function "f(i) = d(o,i) + h(i,d)". d(o,i) stands for the cost of the best path from the source to the actual node.

While h(i,d) is the heuristic estimation distance from node 'i' to destination 'd'. The lower the value of f(i) for this node is, the higher the chance for it to be a member of the current shortest path. The algorithm sorts in ascending order nodes of the scan set according to their corresponding heuristic function f(i). Next the first node in the set is selected for expansion using the best first search where all edges related to this node are examined and added to the scan set with respect to their heuristic function values "f(i)" [8]. This procedure is repeated until the destination node is selected for examination. The A* is efficient as long as the heuristic estimation is acceptable i.e. "$0 \leq h(i,d) \leq h^*(i,d)$" and does not overestimate the optimal cost from the same node to the target node $h^*(i,d)$ [7].

1.3.2 Branch Pruning Algorithm

The spirit of the branch pruning algorithm stem from iterative-deepening A* algorithm used in Artificial Intelligence [6]. It is based on pruning the nodes that have no chance of being visited during the shortest path formation. The nodes that are close to the source and destination have a higher chance of being members of the shortest path. Conversely, those nodes that are remote from the source and the destination are discarded and will

have a low probability of being part of the shortest path. When selecting a node from the queue it undergoes a simple test in step3 of the algorithm shown in Figure 1.1:

if d(o,i) + e(i,d) > E(o,d) then go to step4

Where 'o' is the source, 'd' is the destination and 'i' is the current node. Then if the current minimal cost from the source to a designated node "d(o,i)" plus the estimated cost from the same node to the destination node "e(i,d)" is greater than the estimated upper bound "E(o,d)", this node is skipped. This upper bound E(o,d) known as minimal cost from source to destination is referred in some references as a cutoff [7].

1.4 Genetic Algorithm

Evolutionary algorithms simulate the biological evolutionary mechanism found in human beings, in order to solve combinatorial optimization problems [9]. Evolutionary algorithms operate on a population that undergoes selection and recombination in order to create a new population with better individuals. Genetic algorithms are based on the principle of randomness during the genes' creation of the chromosome. This allows building up a pool of chromosomes. Each chromosome is a sequence of random genes that simulates a path from the source to the destination. A chromosome corresponds to the domain knowledge and it is represented by a vector of length "ln" such as $Ch = g_i$ (where by $0 \leq i \leq ln$) and 'g_i' is the gene at position 'i' in the chromosome 'Ch'. Once the size of the population is determined, an initial population is generated.

The genetic operators are next applied by selecting individuals from the population. This is followed by a selection process where individuals with higher fitness scores have a higher probability of survival. These generated offspring will be stored in a new

population and ready for the next generation. The process is repeated 'generation ← generation +1' until the number of generations specified by the user has been reached 'generation > max generation'.

Crossover is used as the main genetic operator since it permits to explore the design space while mutation is used to avoid the convergence to a local optimum by inserting new chromosomes. The mutation rate should be low in order to give priority for the crossover operator and from time to time to add new individuals to the pool for exploring new sections.

1.5 Problem Description and Thesis Outline

Holland [10] first introduced genetic algorithms to solve combinatorial problems. This thesis proposes to solve the shortest path problems in medium to large spectrum graphs using genetic algorithm. We tackle road networks that contain hundred thousands of nodes and edges that cannot be solved by Dijkstra due to memory limitations.

Consequently, finding an efficient feasible shortest path with Dijkstra will be impossible as the number of nodes and links increases in size. Another problem with Dijkstra is that it has to run all over again in case of network failure trying to find a new substitute in a very short time i.e. matter of milliseconds for the network to survive [1]. The proposed algorithm works by creating an initial population of feasible chromosomes that relate the source to the destination exempted from any redundant individuals.

Genetic operators are applied to the population to produce a new fit population. After executing the genetic algorithm for a specified number of generations, a sub-optimal

answer is obtained in most of the cases. In the worst case scenario this algorithm yields to a group of suboptimal solutions to avoid fiasco.

Chapter 2 surveys the previous related work regarding the shortest path problem such as Dijkstra algorithm, Bellman-Ford algorithm, knowledge based Dijkstra algorithm, and other genetic algorithms. Chapter 3 presents our proposed genetic algorithm to solve medium size spectrum graphs while Chapter 4 focuses on large spectrum graphs. We conclude in chapter 5.

Chapter 2

Related Work

The problem of finding the shortest path between two nodes is a well known problem. Many algorithms have been used to solve the problem. One to mention is Dijkstra which is a very powerful algorithm for small to medium spectrum networks that always returns the optimal shortest path from a single source to a single destination [11]. As the number of nodes increases the memory storage used by Dijkstra becomes prohibitive. Another one to mention is Bellman-Ford algorithm, which is similar to Dijkstra but allows for negative weight edges.

Knowledge based Dijkstra algorithm is based on the idea of shrinking the search area by using a knowledge based route finder. Then, Dijkstra is used to find the shortest path if the minimized area is small enough to be covered. Otherwise, it will fail due to the same reason mentioned previously. Knowledge based genetic algorithm was able to overcome the problem facing knowledge based Dijkstra algorithm. After reducing the area instead of running Dijkstra, genetic algorithms accomplish the mission to avoid failure. Finally, different types of genetic algorithms were used to solve the shortest path problem. Each one of them is based on the same evolutionary process but uses different variations of selection techniques such as (roulette wheel selection, pair wise tournament selection).

These genetic algorithms uses also different types of crossover such as (one point crossover, two point crossover, position based crossover, partially mapped crossover) as well as utilizes different deviations of mutation such as (one point mutation, local search-

based mutation, swap mutation). In addition, different rates for crossover and mutation are used. The number of generations also varies from one algorithm to the other.

2.1 Dijkstra Algorithm

Dijkstra's algorithm was proposed in 1972 in order to find the shortest path between a single source node and all other vertices in a given graph G=(V,E) where V are the vertices of the graph and E are the edges connecting those vertices. Dikjstra algorithm is a greedy one which functions by selecting in the neighborhood the most important alternative and it has an order $O(n^2)$. Its algorithm is shown in Figure 2.1

Dijkstra(G,w,s)	where G is the graph, w weights of nodes and s = source
Initialize-single-source(V,s)	all label nodes = ∞, parent = nil for each node, degree s = 0
S ← {}	empty set no vertices checked yet
Q ← V[G]	where Q is priority queue or heap containing all vertices
While Q ≠{} do	loop while heap is not empty
u ← extract-min(Q)	extract from Q a vertex with the minimum shortest path
S ← S + {u}	S will hold examined vertices so far
for each vertex v ∈ Adj[u] do	check each neighbor to node u
Relax(u,v,w)	using triangular inequality

Fig. 2.1 Dijkstra Algorithm

2.2 Bellman-Ford Algorithm

Bellman-Ford algorithm is based on the principle of dynamic programming for finding the shortest path route in a graph. This shortest path is found through the use of recursive call from source to destination. The algorithm traverses all edges |V-1| times performing a relaxation technique on each edge. It allows for negative weight edges and has a running time of order O(VE).

Both Dijkstra and Bellman-Ford are based on the triangle inequality for relaxing the edges of the graph. Since both will not work with medium to large spectrum graphs, new efforts where made by computer scientists to overcome this problem by using new techniques such as genetic algorithms.

2.3 Liu's Knowledge Based Dijkstra Algorithm

Liu's Knowledge based Dijkstra algorithm covers geographical graphs [12]. This algorithm has two main integrating techniques the case based route finder and the knowledge based route finder. First, a case based route finder attempts to match the route with some previous stored shortest paths. If the match occurs an optimal route is automatically thrown from the database. Otherwise, the case based reasoner will give the closest partial route and leave the remaining part to Dijkstra. In the worst case scenario when the reasoner fails even to return a partial route, the graph is passed to a knowledge based route finder. The knowledge based route finder will prune unnecessary nodes via heuristic knowledge.

As a result, the search area is now limited to a grid which is used by Dijkstra to find the optimal shortest path [1]. For example, to look for a path from the central part of a city to a destination in its southern area, there is no need to check its northern part. Therefore the search is now confined to a particular area of the graph. This algorithm contains a case based controller that will add, modify or remove a route through the process of continuous learning by experience. The use of Dijkstra over a grid will restrict the system's results, since as much as the grid stretches in size the answer may or may not be the shortest path [12].

2.4 Genetic algorithms

2.4.1 Kanoh's Knowledge Based Genetic Algorithm

Kanoh's Knowledge based genetic algorithm was introduced to overcome the weakness of the knowledge based Dijkstra algorithm [14]. A set of partial routes, known as viruses, constitute the domain based knowledge. Viruses do include neither the source node nor the destination node. Conversely, individuals must contain both source and destination nodes. Two populations are generated one for the viruses and the other for the individuals. The population of viruses contains national roads and local roads of the city. The population of individuals is first based on randomly selecting one virus. Second, the partial route from the source node to the virus and the partial route from the virus to the destination node are produced by using a real time heuristic search algorithm [13].

Finally, the path that concatenates these two former partial routes with the injected virus forms a new individual. For example suppose we have the path P_1 = "S a b c d D" and the injected virus V_1 = "e b d f". The infected path will be P_2 = "S a b d D". Where by 'S' is the source, 'D' is the destination, and 'b,d' are common intersections. The new path P_2 will substitute the old path P_1 in the population of individuals. Crossover takes place at common genes in both parents [14].

2.4.2 Munemoto's Algorithm

Munemoto's algorithm [15] data structure is based on variable length vectors to present chromosomes. However, this algorithm has two major weak points. The first one is its potential failure during the crossover operator. This is because the crossover does not operate unless it takes place on two identical genes in both parents at the same position. Crossover failure leads to a suboptimal solution by omitting many candidates that are necessary to explore new regions of the graph. In order to solve this dilemma they increased the size of the population at the expense of a longer computation time.

The second inconvenience of Munemoto's algorithm is that it might fail during the mutation process since it works as follows: first, it chooses randomly one gene of the chromosome as a mutation point. Second, it applies Dijkstra algorithm from the source to the mutation point in order to get the first part of the chromosome.

Third, it randomly selects one of the neighbors of the mutation point and applies to it Dijkstra until it gets the second part of the chromosome. Finally, concatenate both parts to get the final path from source to destination. The weakness of mutation is that it relies

on Dijkstra which could fail while traversing the nodes of the entire large spectrum graph.

2.4.3 Inagaki's Algorithm

In Inagaki's algorithm fixed length chromosomes are used [16]. Each gene of the chromosome holds a node label that is chosen randomly from the set of nodes that are neighbors to the corresponding gene index. This process is repeated until reaching the destination. During crossover, one of the parents' genes that exist at the index representing the source node is selected and stored at the same index in the offspring. The next gene is selected according to the index of the previous stored node. This process is repeated until reaching the destination. Due to this weak crossover the algorithm needs a large population to converge to an optimal answer. Therefore, the computation time turns to be slow during the evolution process.

2.4.4 Gen's Algorithm

Gen's algorithm utilizes the priority based encoding technique. First, a chromosome is generated randomly by finding those nodes that are connected to the source node and assigning to it random priorities.

Next, select the neighbor with the highest priority and add it as a new gene. The algorithm repeats this method until the destination node has been reached [9]. In addition, the position based crossover used by Gen's is described as follows:

Some genes at random positions from one of the parents are transferred to the same genes' positions of the offspring. The remaining genes that will contribute in forming the offspring are taken from the other parent by a left to right scan.

During the scan, redundancy is avoided by skipping those genes already taken from the first parent. This works well for the TSP problem since if we shuffle the genes we still get a feasible path.

2.4.5 Other Genetic Algorithm

Many other important genetic algorithms exists that cover multicast routing problem in communication networks. For example, in multiparty multimedia teleconference systems and distance learning the video and voice at each point communicate with all other points available in the network [17]. This is why the QoS (quality of service) is important to guarantee a soft audio-video exchange [18]. This exchange of multimedia information should be in a very concise time matter of micro seconds. These algorithms are built by using a multicast tree or Steiner heuristic minimal tree [19] and are beyond the scope of this thesis.

Chapter 3

Heuristic Shortest Path for Networks

3.1 Chromosome Representation

A chromosome is a sequence of genes used to represent a path from a given source to a given destination. The genes stand for the visited nodes in a given graph. Two consecutive genes represent a link between two nodes or simply a weighted edge. For example, the chromosome represented in fig.3.1 indicates that there is an edge from 'node1 'to 'node2' and another edge from 'node2' to 'node3'. A chromosome is said to be feasible if its genes contributes in building a connected route from the given source to the given destination.

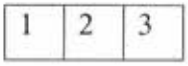


Fig. 3.1 Chromosome Representation

3.2 Chromosome Creation

The chromosome creation passes through the following steps:

1. Define a variable length array that will hold the genes of the chromosome.
2. Assign the source node index to the first gene, i.e. c [1] = 1.
3. Test all the links from the current node in the adjacency matrix, and assign those links which are different from infinity to a new variable length array named *v*.

4. Generate a random number between index 0 and the last index of the array v that holds all possible links from the current node to all possible neighbor nodes.
5. According to the random generated index, select the corresponding value that exists within that index of the array created in step 3 (new gene = g = $v[i]$) where i is the random generated index.

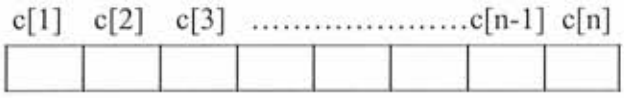


Fig. 3.2 Chromosome Creation

6. Add the new gene to the end of the variable length array representing the chromosome. That is, c[j] = g. where $2 \leq j \leq n-1$
7. Repeat steps 3 to 6 until the destination is reached [28].
8. Eliminate the loop that exits within the genes of the chromosome by calling the repair function. This will lead to an initial feasible population from which crossover and mutation can lead to a new elite population.

The algorithm in Figure 3.3 shows the steps of creating a feasible pool of chromosomes with no duplicate. For example, applying crossover on two duplicate parents from the population will yield to the same output. Too many redundancies will minimize the search area. The number of chromosomes constituting the population should be quite enough to start applying the next two genetic operator crossover and mutation to create a new population.

```
• P = size of the population
• Pop = array of chromosomes
• Ch = a chromosome of the population
• Matrix = matrix holding benchmark of the graph
• Msize = matrix size
• Neighbors = array containing nodes that are neighbors to the current node

InitializePopulation_algorithm ()          // pop is an array of individuals
begin
 for i = 1 to P do
 begin
   v = 1
   gene = 1
   ch[v] = gene                    // store 1 in the 1st gene of the chromosome as being the source node
   destreached = false             // Boolean variable to detect if destination has been reached
while (not destReached) do
   begin
     k = 1
     for j = 1 to Msize do                //find all neighbors of gene and reserve it in array neighbors
     begin
       if (Matrix[gene][j] < infinity) then   // value < infinity means there is an edge between the 2 nodes
       begin
         neighbors[k] = j
         k = k + 1
       end if
     end for
     index = random * neighbors length      // generate a random number
     gene = neighbors[index]                // return a random element from array
     v = v +1
     ch[v] = allele                         // add this new arc as a gene to chromosome
     if (gene == Msize) then
     begin
      Pop[i] = ch                           // store the created chromosome in population
      destReached = true
     end if
   end while
   sameChromosome = false
   j = 1
   while ((j < i-1) and (not samechromosome)) do          //eliminate the creation of same chromosome
   begin
    if (Pop[i] = Pop[j]) then
    begin
     sameChromosome = true
     i = i- 1      // decrement i to drop the duplicate chromosome and create a new one for the population
    end if
   end while
 end for
end
```

Fig. 3.3 Heuristic chromosome creation pseudo code

3.3 Fitness Evaluation

Each chromosome has its own main fitness calculated by summing up the weights of the edges connecting its genes from source node to target node. The weights are reserved in a cost matrix W [20].

The main fitness is represented by the following equation in Figure 3.4:

$$\text{Fitness} = \frac{1}{\sum_{i=1}^{n-1} W_{i,i+1}}$$

Fig. 3.4 Fitness equation

Where 'n' is the length of the chromosome and $W_{i,i+1}$ is the weight or the distance connecting gene 'i' to gene 'i+1' in the corresponding chromosome.

Pseudo code for fitness calculation is shown in Figure 3.5

```
CalculateFitness (Ch)
 begin
   mainFitness =0
   ln = chromosome length
   for i = 1 to ln – 1 do
   begin
     mainFitness = mainFitness + Matrix[Ch[i] , Ch[i+1]]
   end for
   mainFitness  = 1/mainFitness
end
```

Fig.3.5 Chromosome fitness pseudo code

3.4 Selection Technique

Once the population of individuals with its corresponding assigned main fitness values has been created, the next step is to select the appropriate chromosomes that will pass their genes to the next population through genetic operators.

The purpose of selection is to give fitter individuals a higher probability of being selected during the evolution process. This is known as "survival of the fittest".

If the selection procedure is inappropriate, it will lead to suboptimal individuals overwhelming the population before a good solution is reached. A variety of selection methods exist.

3.4.1 Fitness-Proportionate Selection

In fitness-proportionate selection, the chance for an individual to replicate is equal to the individual's fitness divided by the average fitness of the population. Roulette wheel selection is a well known procedure based on fitness proportionate selection [26]. Our proposed GA uses the roulette wheel selection method. First, it calculates the scaled fitness which represents the difference between the main fitness of the current chromosome and the chromosome with the minimum fitness in the population. (*scaledfitness* = *mainfitness* – *minimumfitness*). Second, all scaled fitness are added together to form (*sumfitness*). Third, the selection of an individual for a crossover or mutation is simply done by generating a random number between 0 and *sumfitness*. Next, the population is sorted in decreasing order by the quicksort algorithm (O(nlgn)). Finally, the scaled fitness of the visited chromosomes are added until it becomes greater or equal to the generated random number. Once this occurs, the proposed algorithm selects the current chromosome as being a good candidate for multiplying. This is shown in Figure 3.6

```
ScaledFitness (Pop)
begin
  minFitness = 1000                          // start with a big number for minimum fitness
  sumFitness  = 0
  for i = 1 to P  do                         // where P is population size
  begin
    if ( Pop[i].fitness < minFitness) then   // where Pop[i] is a chromosome in population
       minFitness = Pop[i].fitness           // loop to find  individual with smallest fitness
    end if
  end for
  for i = 1 to P    do                       // subtract the min fitness from the fitness of individual
     Pop [i].scaledFitness =Pop[i].fitness – minFitness          // assign difference to scaled
   for i = 1 to P do
     sumFitness = sumFitness + Pop[i].scaledFitness      // sum up all scaled fitnesses
  end
SelectIndividual (Pop)                       // Rescale fitness of all individuals in the population
begin
  sumofFitness = 0                           // sum the fitness of all individuals in the population
  index = 0
  limit = random * sumFitness          // 0 ≤limit < sumFitness  where sumFitness is the sum of all
scaledFitness
  while ((index < P ) and (sumofFitness < limit)) do          // P is the population size
  begin
    sumofFitness = sumofFitness + Pop[index].scaledFitness
    index = index +1
  end while
  Ch = pop[index-1]
end
```

Fig. 3.6 Fitness Proportionate Selection Pseudo code

3.4.2 Rank Selection

In rank based selection, the individuals are also sorted according to their fitness values. But this time a rank is assigned for the individuals from top to bottom with 'n-1' given to the highest fitness and '1' to the lowest fitness. A probability is then assigned to each chromosome according to its rank value.

The idea is that even when the fitness variance among the chromosomes is too small, it can still offer an appropriate selection emphasis [26]. In this method there is no need for fitness scaling since the selection pressure will be based upon the individual's rank. Its disadvantage is when the selection pressure is not well tuned; it will lead to a premature convergence [20]. One variation is the "linear ranking" [26].

3.4.3 Tournament Selection

Another well known method is the tournament selection based on a continuous selection of N individual from the population. Next, the individual with the higher fitness will be assigned to a temporary population. This process continues until the temporary population has enough number of individuals. As much as the tournament size increases so will the selection pressure. Whenever the selection pressure increases, a percentage of genetic features will be lost due to an early convergence. One variation, known as "pair wise tournament selection", has a tournament size (s=2) to control the selection pressure. Therefore, the tournament set will hold two chromosomes from which the one with the highest fitness is selected as a parent.

3.5 Crossover

The crossover operator works on two different selected parents from the pool of individuals by interchanging parts of each individual's genes. This exchange is based on randomly choosing one or multiple common crossing sites known as crossover points.

Many variations exist, such as:

One point crossover technique is based on one common position upon which we perform the swap between the genes of two parents. As for the multi point crossover, it uses two common intersection points at the same positions in both parents. This method swaps the genes of both parents falling between the two crossing sites [26]. While the partially mapped crossover (PMX) is similar to the multi point crossover, where some of the genes falling outside the two crossing sites may have duplicate genes within the common intersections. Duplicate genes in offspring1 to the right and left of the common crossing sites will be swapped with non redundant genes falling outside the common intersections in offspring2. The same thing will take place between offspring2 and offspring1 [27].

One point crossover is used in the proposed genetic algorithm and it works as follows: First, select two different chromosomes even if they do not have the same length according to a fitness function (Roulette wheel selection). These two chromosomes should have at least one common gene. The common genes could occupy different locations in parent chromosomes excluding the source and destination genes which are equal by default. Second, we compare the two chromosomes with one another, and reserve the common genes' positions as pairs into a variable length array. For example, the array may hold the following pairs (4, 4) and (8, 7). Third, we select randomly one of these pairs such as (8, 7) and use it as a crossing point. Genes are now swapped upon node position '8' in parent1 and node position '7' in parent2.

Each offspring produced by crossover is a concatenation of two partial routes. Genes from the source node up to the crossover point in parent1 are used to build the first part of offspring1.

On the other hand, genes after the crossing point up to the destination node in parent2 are added to offspring1 to form a complete path. Same thing will take place with offspring2.

Here follows an example of one point crossover:

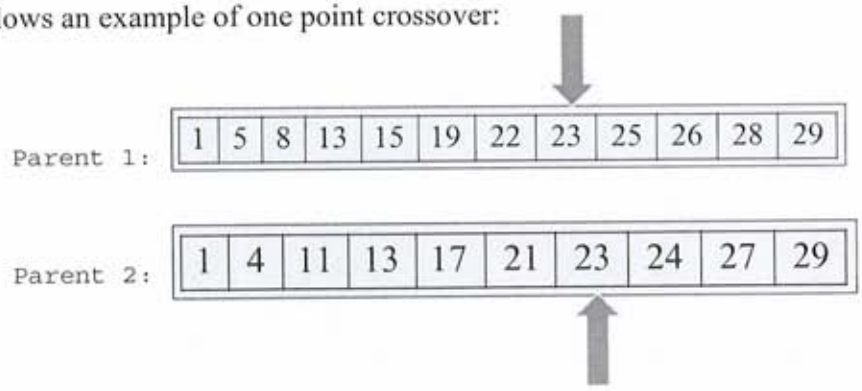
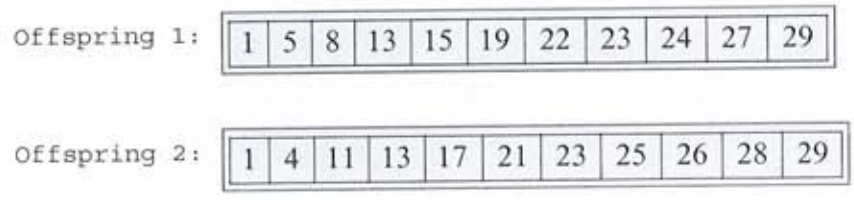


Fig. 3.7 One point crossover example

Crossing point is at gene position '8' in parent1 and at gene position '7' in parent2 since they hold the same value '23'.

The resulting offspring will look like this:



We can observe that during the process of evolution, crossover contributes in visiting other parts of the graph through its offspring. Therefore, crossover should be the dominant genetic operator during the whole process.

Figure 3.8 shows the pseudo code for the crossover implemented in the proposed GA.

```
// swap segments of two selected parent individual chromosomes.
// at least one common node should exist between the parents; except for source and destination.
// find all possible common points and choose one randomly
Crossover (parent1, parent2)
begin
 ln =  length (parent0)
 lm = length(parent1)
 k = 1
 for i = 2 to ln – 1 do                    //start with "2" and end with "size()-1" to eliminate
  for j = 2 to lm – 1 do                   // source and destination as crossing points.
  begin
   if (parent1[i] = parent2[j]) then
    Position[k] = "i" + ", " + "j"         // concatenate the 2 positions of the common genes
    k = k + 1
   end if
  end for
 end for
 if (position[1] = 0) then
  print " no common intersection for crossover, select new parents"
 else
  cut = random * size of position          // get a random index from the array position
  point = position[cut]                    // get an element from array position using random index
  x = getX (point)                         // get x from the stored string in point
  y = getY (point)                         // get y from the stored string in point
  for i = 1 to x do
   child1[i] = parent1[i]
  for i = y +1 to lm do
   child1.[i] = parent2[i]
  for i = 1 to y do
   child2.[i] = parent2[i]
  for i = x + 1 to ln do
   child2.[i] = parent1[i]
 endif
end
```

Fig.3.8 Crossover Pseudo code

3.6 Mutation

Mutation is the process where one of the chromosome's genes is chosen randomly and changed either by flipping it or swapping it with another value. Mutation is very important since it will insert from time to time new individuals in the newly created population to redirect the evolution process which is converging towards local maxima. These newly inserted individuals are referred in some papers as strangers [30]. Several mutation techniques are available such as:

The Swap mutation where by two genes are selected randomly from the chromosome and swapped [30]. Another variation is the Flip mutation where one gene is selected randomly from the individual and flipped to its inverse value [26].

The mutation used in the proposed GA plays the role of a cut off where by part of the chromosome is truncated and replaced by a newly created partial route. This is done according to the following steps:

First, we choose a random gene as a mutation point. Second, we copy the genes from the source node up to the mutation node included into a new variable array. Third, we treat the mutation point as a new source node, and we choose randomly one of the nodes connected to this new source. Fourth, we add this selected node to the new created variable array. Fifth, we pick a new random neighbor with respect to the actual new source. Finally, we repeat the last two steps until the destination node is reached.

The mutation rate used is too small in order to give priority for crossover operator to take place since a higher rate would damage good solutions; therefore, an individual is mutated in order to insert a foreigner in the population which allows for a new solution

space to be visited [30]. After several runs with different rates we came out with a mutation rate = 0.008 being the most suitable one during the evolution process.

Here follows an example of mutation taking place in the genetic algorithm at a very low rate.

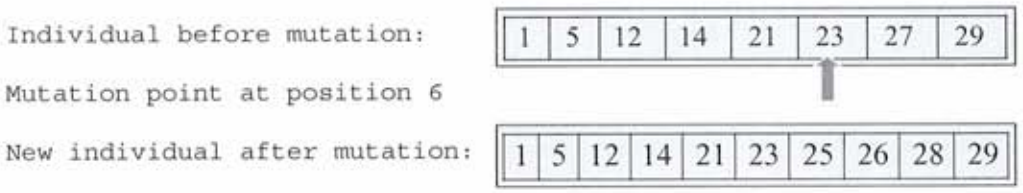


Fig. 3.9 cut off mutation example

As we can see from the above example the nodes that fall after the mutation pivot "23" in the selected individual are dropped and substituted by a newly heuristic partial path in the new individual. The chromosome creation used in section 3.2 will be implemented with the gene at the mutation pivot "23" being the source node.

Pseudo code for mutation is shown in Figure 3.10:

```
Mutate(Ch)
begin
  ln = length(Ch)                    // ln is the length of the chromosome
  g = (random * (ln - 2)) + 2        // use "-2" to eliminate the use of destination
  gene= Ch[g]                        // choose a random gene as mutation point
  for i = g + 1 to  i <= ln do
  begin
    remove.lastElement(Ch)           // remove all genes that exist after muttion point
  end for
  generatepath(gene, destination)    // used to create a new random path from mutation point  to dest.
  calculateFitness(Ch)
end
```

Fig. 3.10 Mutation pseudo code

3.7 Repair Function

This function is used to eliminate a loop within a chromosome in an undirected graph. Suppose one of the genes indicates node 2 and another forward gene in the same chromosome indicates again node 2.

Therefore, node 2 is kept in the first occurring gene and all the nodes from this gene until the second occurrence of the same gene included are removed from the chromosome. This function is repeated as much as necessary in order to obtain a final chromosome without any loop in its genes. This function is called during the process of chromosomes creation, after crossover and mutation. This is an example where by the genes of chromosome hold a loop. The genes representing the loop are indicated by a pair of arrows.

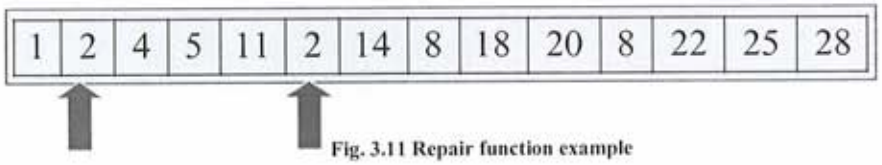
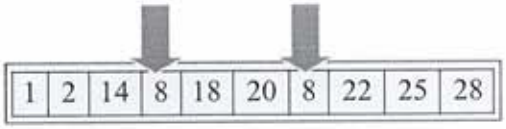


Fig. 3.11 Repair function example

Gene **2** is found at positions 2 and 6 of the chromosome. We fix this problem by removing all genes after the first occurrence until the second occurrence included. We repeat the same process for other repeated genes. The result will be a chromosome free from any loops.

After removing value 2 the chromosome becomes:



Now after removing value 8 the chromosome will be in its final shape:

Pseudo code for the Repair function is shown in Figure 3.12:

```
Repair (Ch)
begin
 diff = 0
 ln = chromosome length
 for i = 1 to ln – 1  do          // ln -1 to take out the destination node
 begin
  j = 1
  found = false
  while ( (i < ln-j) and (not found)) do
  begin
   if (Ch[ i ] =Ch[ln – j]) then
     begin
       diff = (ln - j) - i        // counts how many elements to remove from chromosome to eliminate loop
       while (diff >0) do
       begin
          removeElementAt Ch[i+1]      // remove elements after position i according to diff value
          diff = diff - 1
       end while
       ln = chromosome length
       found = true
   else
      j = j +1
   end if
  end while
 end for
end
```

Fig.3.12 Repair Function

3.8 Proposed Genetic Algorithm

The algorithm starts at generation = 0 by generating n random non-redundant chromosomes. Each chromosome is a sequence of non repeated genes simulating a path in the graph from a source node to a destination node. Once the number of chromosomes is equal to the population size (n = 100), we compute their fitness.

The algorithm proceeds next and find the fitness by using the following arithmetical equation $1/totalweights$ where "*totalweights*" is the sum of the of edges' weights connecting the chromosome's genes. The algorithm next sorts the population in descending order according to its fitness and uses the roulette wheel selection to pick up the best two parents for reproduction. The algorithm next applies one point crossover on the two chosen parents to produce two offspring that will be added to a new population. The genes of the first offspring are the concatenation of two paths. The first path holds the genes of parent1 from the source to the crossover point included. The second path holds the genes of parent2 after the crossover point up to the destination node included. The genes of the second offspring are constructed in an opposite way to the one applied onto the first offspring. Next, a random number is generated and compared to a low mutation rate. If it is lower than this rate, we apply mutation to a selected individual by cutting off part of its genes that fall after a random mutation point and replacing it with a new generated sequence of genes. Finally, we replace the old population with the new population and increment generation by one. We repeat this process until the number of generations has exceeded a given constant "Max generation".

The flowchart for the proposed genetic algorithm is shown in Figure 3.13

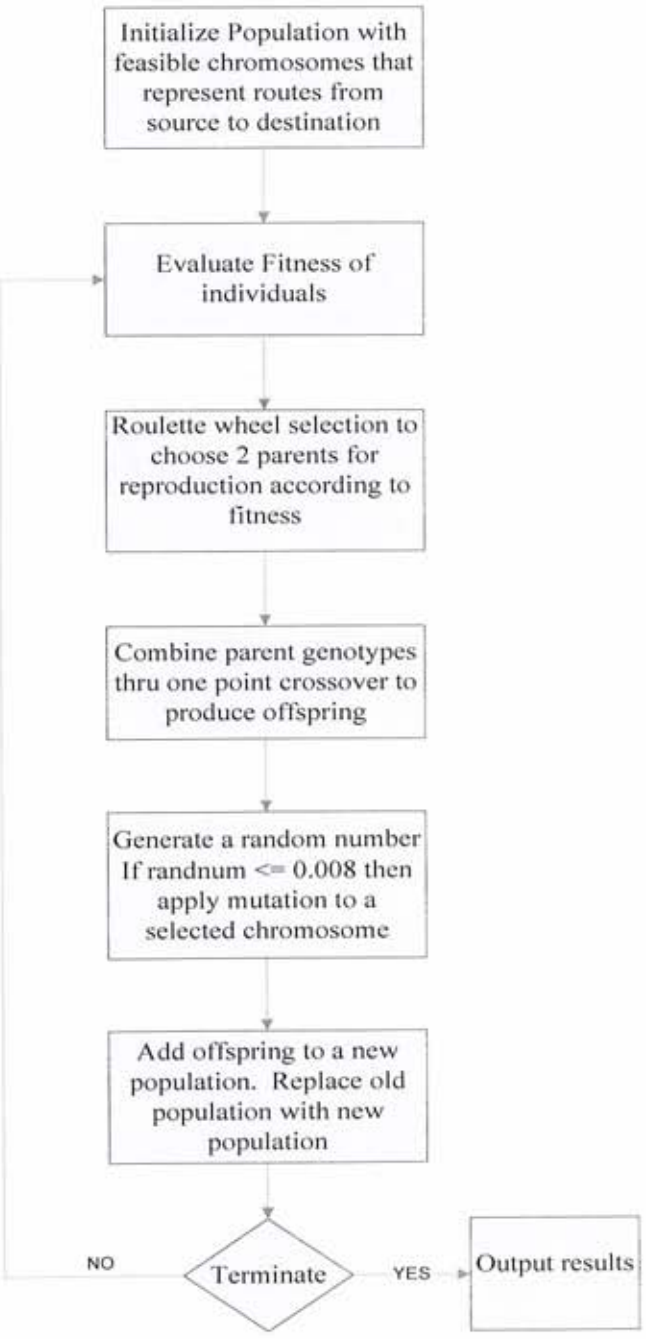


Fig. 3.13 Genetic Algorithm Flowchart

3.9 Pseudo Code for the Proposed GA

```
Step1:
    Initialize (population)          using a matrix O(kn^2) where k is population size
                                     And n is the chromosome length.
    Generation ← 0
Step2:
    while (generation <= maxGeneration)
        Calculatefitness(population)   O(kn)
        Quicksort(population)          O(klgk)
        Scaledfitness(population)      O(k)
        Select(parent 0)               O(k)
        Select(parent 1)               O(k)
        crossover(parent 0, parent 1)  O(n) + O(m) n, m are parents length
        new population ← child 0       O(1)
        new population ← child 1       O(1)
        generate random number         O(1)
        if (random number <= mutationrate)
        mutate(individual)             O(n^2)
        new population ← individual    O(1)
        end if
        population ← new population    O(k)
        generation ← generation + 1
    end while
Step3:
    Print(population)                  O(k)
```

Fig. 3.14 Proposed Genetic Algorithm

The expected running time for the proposed algorithm at the worst case scenario is $O(kn^2)$ where k is the population size and n is the chromosome length. Figure 3.14

3.10 Experiments & Results

The proposed genetic algorithm works fine with networks ranging from small to medium spectrum based on information being reserved in a matrix (graphs up to 3500 nodes). As for larger spectrum graphs, the computer memory fails to hold its corresponding huge matrix. The solution for this new challenge is by using a database. This new concept will be discussed in the next chapter. The results using the proposed genetic algorithm for small to medium spectrum graphs are discussed next.

3.10.1 Small Spectrum Graphs

For small spectrum graphs, all simulations using the genetic algorithm were performed on a Pentium III, 1 GHz CPU, 256 MB RAM using one point crossover, a low mutation rate = 0.008 and a roulette wheel selection with bottom replacement. The GA gave competitive results to those given by Dijkstra. In addition of finding the shortest path (as Dijkstra algorithm), it also found other chromosomes that offer the next shortest paths in case of any type of failure. The population size assigned to (***pop size = 40***) and the number of generations is taken as (***generations = 10***). The quality of solution is slightly affected by the population size since a larger population will not give better performance. As we increase the crossover rate, this will decrease monotonically the number of generations [21]. The proposed genetic algorithm has 100% route optimality for small networks. Therefore, the route ratio failure coincides with that of Dijkstra algorithm (i.e. 0% failure). Executing the proposed genetic algorithm over a benchmark of 20 nodes

shown in Figure 3.15 [20] gave the following numerical results with the detection of the optimal route. The optimal path is represented by a bold line in the below graph.

Last population after generation 10

1 3 8 14 20 chromosome fitness is: 0.0070422534

1 2 7 8 14 20 chromosome fitness is: 0.0066225166

1 4 9 3 8 14 20 chromosome fitness is: 0.005952381

1 3 8 14 18 20 chromosome fitness is: 0.0048076925

The time needed is: 0.033 second

(b)

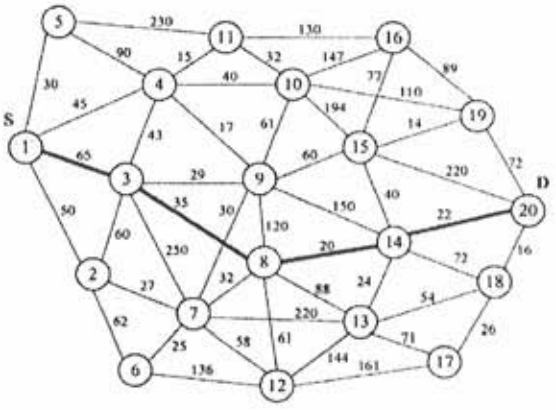


Fig. 3.15 Small Spectrum Graph

Another example that we attempted has 30 nodes [29] where the optimal path has also been found by running the proposed genetic algorithm with the same number of generation. As for timing, this algorithm found the solution along with other useful solutions in an average computation time of ***0.033s***, as for Dijkstra the average computation time was ***0.14s*** on the same platform Pentium III mentioned above. The computation time using Dijkstra algorithm increases relatively with the network size, but when using the proposed genetic algorithm the computation time will slightly increase, (Table 3.1).

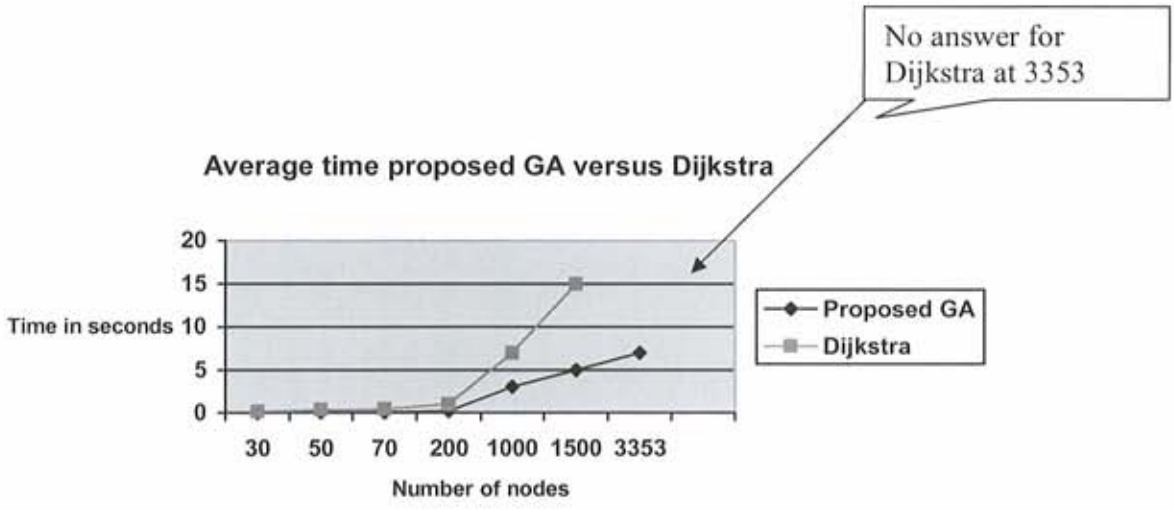


Table 3.1 Computation time

3.10.2 Medium Spectrum Graphs

As for medium spectrum graphs from 1000 up to 3353 nodes (graphs of 1000 and 1500 nodes are chuncks taken from the Romee99 Dimacs benchmark with some modifications), the proposed GA algorithm was attempted using a matrix on "a centrino 1.6 Ghz CPU, 512 Mb RAM" PC and found good solutions for a population of 100 chromosomes and 20 generations. The algorithm found the optimal shortest paths in 90% of the cases. On the other hand, Dijkstra has failed to give the shortest path in graphs where the number of nodes has been extended to cover all the roads in the city of Rome [12].

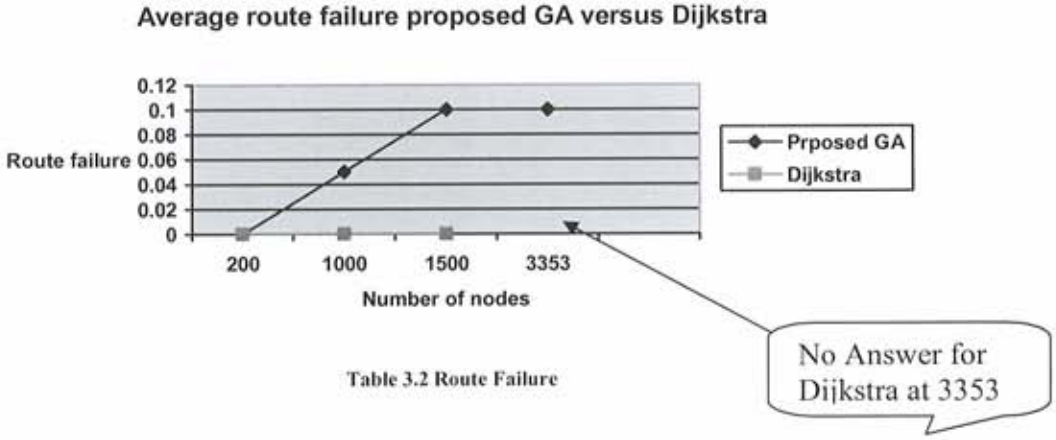


Table 3.2 Route Failure

As we can see from table 3.2 the curve has been truncated to indicate that Dijkstra algorithm was unable to find the shortest path because the computer's heap space was exhausted by the tremendous increasing number of nodes.

Table 3.3 shows the output of different graphs in terms of time consumption and the cost of reaching the destination for both Dijkstra and the proposed genetic algorithm. Thus, every instance was run for 40 times and the difference between both algorithms over most of the graphs is less than 6% as shown in Table 3.3. This indicates that the proposed GA has found an optimal route as Dijkstra's algorithm did or in the worst cases a too close suboptimal path. We have noticed through experiments that as much as the crossover rate increases the number of generations needed to reach optimal answer decreases [21]. This will eventually reduce the computation time.

Graphs		Dijkstra		Proposed Genetic Algorithm					
Nodes	Edges	Cost	CPU time/sec	Cost	CPU time/sec	# gen	Average 40 runs	STD	difference
20	94	142	0.1	142	0.02	10	142	0	0%
30	53	63	0.14	63	0.033	10	63	0	0%
50	136	1430	0.35	1430	0.06	10	1430	0	0%
70	204	522	0.4	522	0.1	10	522	0	0%
200	535	355	1	355	0.2	10	359.85	12.88	1%
1000	2844	591	7	591	3	15	624.8	26.67	6%
1500	3728	589	15	589	5	15	613.10	22.69	4%
3353	8864	NA		30305	7	20	30541.90	223.01	NA

Table 3.3 Comparison between Dijkstra & proposed GA

Chapter 4

Shortest Path for Large Spectrum Graphs

Many spatial graph databases including those used in different domains such as water and gas distribution, phone networks have been using databases for storing the huge amount of data [22]. The proposed genetic algorithm works perfectly with small to medium spectrum graphs since the links connecting the nodes among each other can fit properly into memory through an adjacency matrix. But when working on large spectrum graphs such as road networks, these latter have a very large number of nodes connected together through a tremendous number of edges. Since we are talking about enormous data that will not be able to fit in memory we need other kind of storage device such as a database. We will transform these benchmarks representing the graphs into an MS-Access table on which we will perform a series of queries. These queries will be used to create the pool of chromosomes. Once the population has been created we can launch on it the proposed genetic algorithm to detect the shortest path between the given node pair.

4.1 Creating the Table and its Index

A table is a set of records where each record is divided into fields. In our case the table will hold a tremendous number of records due to the large data being transferred. The corresponding records are divided into three main fields: 'src' for source, 'dest' for destination and 'weight' for the edge cost.

These three fields have 'long integer' data type. Each record stands for a link between two nodes of the graph. To select specific records according to a given criteria the table

should be indexed. The index used in our study is a concatenation of the two fields 'src' and 'dest'. It means the records of the table will be sorted in ascending order first by 'src' next by 'dest' respectively. During the search for those nodes that are connected to the current node, a filter is performed on the consulted table using the created index. The result of this filter will be a set of records that are neighbors of the actual node. Therefore, this index is vital since it will speed up the search among the enormous number of records.

4.2 Creating the Chromosomes via Queries

As we have discussed in chapter 3, to build a chromosome as a sequence of related genes from source to destination, we need to know the neighbors of the current gene. The same concept has to be applied here but with a slight difference; the neighbors of the actual node are not stored in memory but in a table via a query. This query is known as a "Select query" that returns a set of records indicated by the conditional "where" clause [23]. The resulting record set will hold the current's node neighbors. This 'Select query' has the following syntax:

SQL = "Select * from table where table.src = value"

Where value is the current node id and table.src is the field "src" source from the table. Next, we select randomly one of the records stored in the record set and add it as a new record, which stands for a new gene, at the end of another temporary table "temp".

This is done by using a second query known as "insert into" [24, 25]. This query takes the following form:

```
SQL = "INSERT INTO temp (src, dest, weight)" & _
      "VALUES (" & so & "," & de & "," & we & ")"
```

'so', 'de' and 'we' are the inserted values of the random selected record into table "temp". The table "temp" will represent the current chromosome. It should have neither an index nor a primary key. Since these records should remain in its corresponding positions as to maintain a feasible path from source to destination. Any attempt to use an index on the table "temp" will damage the sequence of records resulting in a non feasible chromosome.

The process of using these two main queries continues until reaching the destination node. These links or records from source to destination represent a feasible chromosome. Once the destination has been reached, the current chromosome will be added to the pool of chromosomes by means of a third query. This is known as an "append query" that will just append the records, representing the chromosome, reserved in table "temp" into the population represented by the new table "final" [24, 25]. The syntax for the "Append query" is written as follows:

```
SQL = "INSERT INTO final (src, dest, weight)" & _
      "SELECT temp.src, temp.dest, temp.weight " & _
      "FROM temp"
```

Finally, we have to empty the table "temp" to make it ready for accepting the genes of a new chromosome. This is done by calling a "delete query". This query has the following syntax [24, 25]:

SQL = "DELETE temp.src, temp.dest, temp.weight FROM temp"

This entire process covering the four main queries will be repeated until the number of individuals constituting the initial population is met.

4.3 Loading the Chromosomes into Memory

Once the initial population has been created, we have to move those chromosomes from fixed storage devices into memory. This is done through the use of "JDBC" java database connectivity, which permits a smooth connection to the database engine where the chromosomes resides in a table and are ready to be transferred into the computer's memory. This connectivity will permit for the 'Java' program to open the table 'final' and read it sequentially from top to bottom. Each time the file pointer is advanced by one position and its corresponding record is added as a new cell at the end of a variable length array. Once the record containing the destination node has been reached a new chromosome is added to memory and a new variable length array is ready to hold the next individual. This process is repeated until the file pointer reads 'EOF'. As a result, the initial population exists now in memory. Finally, the genetic algorithm proposed in chapter 3 will execute normally on the pool of chromosomes.

4.4 Experiments and Results:

During the process of building the initial population, some of the chromosomes are truncated after a given number of iterations. The reason is to minimize the computation time and to use these chromosomes as intruders in the population. Therefore, allowing for the proposed GA to explore new regions of the graph. Adopting this technique will not only speed up the execution but will also dump the mutation operator. Therefore, the crossover operator will have full dominance. The files with 6000, 10016 and 25804 vertices are chunks taken from the New York City benchmark. An append query is used to select and insert specific records, that have the 'src' field ≤a given user destination, into a new table. The append query used to build the file with 6000 nodes has the following syntax:

```
INSERT INTO dest6000 ( arc, src, dest, weight )
SELECT USAroad.arc, USAroad.src, USAroad.dest, USAroad.weight
FROM USAroad
WHERE (((USAroad.src) ≤6000));
```

The same query has been used to build the other two files with 10016 and 25804 nodes but the "WHERE" clause has to be changed in order to indicate the user defined destination. Some of the records in the new table (dest6000) will have their destination beyond the scope ('dest' > 6000); therefore, a select query has been used to find these records. This query has the following syntax:

SELECT dest6000.arc, dest6000.src, dest6000.dest, dest6000.weight

FROM dest6000

WHERE (((dest6000.dest)>6000));

Next, a manual replacement is done in the destination field 'dest' with random integers ≤ 6000 in order to transform the file into a well connected graph. As for the Dimacs benchmark with 264346 nodes and 733848 edges that covers the roads in New York City. Some of the links in this benchmark are truncated. Therefore, these links were manually corrected in order to overcome the problem.

Table 4.1 shows the favorable results of every instance, that was ran for 40 times, using the proposed genetic algorithm via queries under the same platform "Centrino 1.6 GHz CPU and 512 MB RAM" PC.

Graph		Dijkstra		Proposed Genetic Algorithm					
Nodes	Edges	Cost	CPU time	Cost	CPU time/mn	# gen	40 runs	STD	difference
6000	14892	NA		5099	3	20	5737	497	NA
10016	19194			6350	4	20	6761	414	NA
25804	69166			9486	6	20	9642	320	NA
264346	733848			42691	13	20	43494	1441	NA

Table 4.1 Results on Large spectrum graphs

5 Conclusion

In this thesis, we have implemented a genetic algorithm to detect the shortest path in road networks. Variable length arrays are used to simulate feasible chromosomes with non-repeated random genes connecting a given source node to a given destination node. The genetic operator crossover and mutation have been successfully tuned to deal with the pool of chromosomes. With the possibility of the crossover pivot to occupy different genes' positions in the selected parents, has added an asset by permitting crossover to occur frequently. As a result, the crossover rate is too high to give more chances for optimal solutions and to minimize the number of generations. On the contrary, the mutation rate was too low approximately '0.008' just to allow for mutation to insert new individuals in the population. Since by injecting new chromosomes in the pool will give more diversity by checking unexplored portion of the graph and will avoid falling soon into a local maximum. The offspring may contain redundant genes; a repair function will take care of it. The results of the presented GA over small to medium spectrum graphs are promising and competitive with those of Dijkstra algorithm in terms of computation time and route failure ratio. For small spectrum graphs the results of the proposed GA coincides with those of Dijkstra in terms of route optimality as for the computation time it is smaller than that of Dijkstra. As for medium spectrum graphs the proposed GA has found amazing results with approximately a 95% probability of getting the optimal path. Finally, storing the large spectrum graphs in a database and applying queries on it to build random feasible paths has allowed for the same genetic algorithm to come out with favorable results.

However, and due to the relative increasing size of the nodes Dijkstra's algorithm didn't succeed in returning the optimal shortest path in term of memory and time constraint. On the other hand, the proposed GA has been able to cope with computer dynamic networks or dynamic road networks in terms of computation time and route failure. In case of any route failure it will extract from the pool the next shortest path to substitute the actual damaged one within a very short time. We have attempted various benchmark examples whose size varied between 30 and 260,000 vertices. The algorithm was able to find sub-optimal shortest paths in all cases.

References

[1] Jagadeesh, G. R., Srikanthan T. & Queck, K. H. (2002, December). Heuristic techniques for accelerating hierarchical routing on road networks. *IEEE Transactions on Intelligent Transportation Systems, 3* (4).

[2] Zhan, F.B & Noon, C. E. (1998). Shortest path algorithms: An evaluation using real road networks. *Transportation Science, 32* (1), 65-73.

[3] Chiang, H.D. & Rene, J.J. (1997). Optimal network reconfigurations in distribution Systems: Solution algorithms & numerical results. *IEEE on Transactions Power Delivery, 5* (3), 169-174.

[4] Duan, G. & Yu, Y. (2003). Power distribution system optimization by an algorithm for capacitated Steiner tree problems with complex flows and arbitrary cost functions. *Elsevier on Electrical Power and Energy Systems, 25*, 515-523.

[5] Fu, L., Sun, D. & Rillet L.R. (2006). Heuristic shortest path algorithms for transportation Application: State of the art. *Computers & Research, 33*, 3324-3343.

[6] Reinefeld, A. & Marsland, T. A. (1994, July). Enhanced iterative deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 16* (7).

[7] Norving, R. & Norving P. (2003). *Artificial intelligence: A modern approach* (2nd ed.). [n.p]: Prentice hall.

[8] Lark ,J. & Syverson, K. (1995, July). A best first search algorithm guided by a set-valued heuristic. *IEEE Transactions Systems Man and Cybernetics, 25* (7).

[9] Gen, M. & Cheng, R. (1997). *Genetic algorithm and engineering design*. New York: Wiley.

[10] Holland, J. H. (1992). *Adaptation in natural and artificial systems*. Cambridge, MA: MIT Press.

[11] Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). Cambridge, MA: MIT Press.

[12] Liu, B. (1996). Intelligent route finding: combining knowledge, cases and an efficient search algorithm. *Proceedings of the 12th International Conference on Artificial Intelligence*, 380-384.

[13] Korf, R.E. (1990). *Real-time heuristic search: Artificial intelligence, 42* (2), 189-211.

[14] Kanoh, H. & Nakamura, T. (2000, September). Knowledge based genetic algorithm for dynamic route selection. *IEEE 4th International Conference on Knowledge-Based Intelligent Engineering Systems & Allied Technologies*, Brighton, UK.

[15] Munemoto, M., Takai, Y. & Sato, Y. (1998). A migration scheme for the genetic adaptive routing algorithm. *Proceedings IEEE International Conference Systems Man and Cybernetics*, 2774-2779.

[16] Inagaki, J., Haseyama, M. & Kitajima, H. (1999). A genetic algorithm for determining multiple routes and its application. *Proceedings of the IEEE International Symposium on Circuits and Systems*, 137-140.

[17] Zhang, Q., & Leung, Y. (1999). An orthogonal genetic algorithm for multimedia multicast routing. *IEEE Transactions On Evolutionary Computation, 3* (1).

[18] Wu, J., Hwang, R., & Lu, H. (2000). Multicast routing with multiple QoS constraints in ATM networks. *Information Science, 124*, 29-57.

[19] Zahrani, M. & Albrecht, A. (2006). Landscape analysis for multicast routing *Computer Communications, 30*, 101-116.

[20] Ahn, C. & Ramakrishna, R. (2002, December). A genetic algorithm for shortest path routing problem and the sizing of populations. *IEEE Transactions on Evolutionary Computation, 6* (6).

[21] Liangsheng, Q., Guanhua, X. & Guanhua, W. (1998). Optimization of the measuring path on a coordinate measuring machine using genetic algorithms. *Measurement, 23*, 159-170.

[22] Shekhar, S., Fetterer, A. & Goyal, B. *Materialization trade-offs in hierarchical shortest path algorithms*. Retrieved April, 2007 from *University of Minnesota*, department of computer science website: http://www1.umn.edu/twincities/02_academics.php

[23] Jones, E., & Jones, J. (1997). *Access 97 Answers certified technical support.* [n.p]: Osborne/McGraw-Hill.

[24] Williams, C. (2001). *Professional visual basic 6 databases*. [n.p]: Wrox Press.

[25] Cornell, G. (1998). *Visual basic 6 from the ground up*. [n.p]: Osborne/McGraw-Hill.

[26] Ghanea-Hercock, R. (2003). *Applied evolutionary algorithms in java*. [n.p]: Springer.

[27] Zahrani, M. & Albrecht, A. (2006). Landscape analysis for multicast routing. *Computer Communications, 30*, 101-116.

[28] Ji, X., Iwamura, K. & Shao, Z. (2007). New models for shortest path problem with fuzzy arc lengths. *Applied Mathematical Modeling, 31*, 259-269.

[29] Kim, K., Gen, M., & Yamazaki, G. (2003). Hybrid genetic algorithm with fuzzy logic for resource-constrained project scheduling. *Applied Soft Computing*, 174-188.

[30] Borra, S., Muthukaruppan, A., Suresh, S., & Kamakoti, V. (2007). A novel approach to the placement and routing problems for field programmable gate arrays. *Applied Soft Computing, 7*, 455-470.

[31] Davies, C. & Lingras, P. (2003). Genetic algorithms for rerouting shortest paths in dynamic and stochastic networks. *European Journal of Operational Research, 144*, 27-38.

Appendix A

First, create an adjacency matrix with size equal to the number of nodes in the graph and initialize it to "infinity". Infinity means that the edge doesn't exist so far. Third, read the ***Dimacs*** benchmark format, transform it into a readable and accessible format by filling the corresponding matrix cells, which stand for arcs, with its respective weights from the latter. The ***Dimacs*** benchmark header contains lines starting with c to indicate a comment line, the line staring with p "p sp 6 8" indicates that we have a graph of 6 nodes and 8 edges and the line starting with an a "a 1 2 17" indicates that we have an edge from node 1 to node 2 with weight 17. It means the matrix cell M[1,2] = 17. The final result will be a matrix with some cells equal to a very large number indicating no path exists between them and other cells with integer values "weights" indicating path exits between them. Fig.13 shows how the corresponding below small "***Dimacs***" benchmark is represented by an adjacency matrix.

A sample for a ***Dimacs*** benchmark:

c 9th DIMACS Implementation Challenge: Shortest Paths

c Sample graph file

p sp 6 8

c graph contains 6 nodes and 8 arcs

c node ids are numbers in 1..6

a 1 2 17

c arc from node 1 to node 2 of weight 17

a 5 2 0

a 4 6 3

a 2 4 2

a 1 3 10

a 4 3 0

a 5 6 20

	1	2	3	4	5	6
1	∞	17	10	∞	∞	∞
2	∞	∞	∞	2	∞	∞
3	∞	∞	∞	∞	0	∞
4	∞	∞	0	∞	∞	3
5	∞	0	∞	∞	∞	20
6	∞	∞	∞	∞	∞	∞

Adjacency matrix representing the graph

Appendix B

The benchmarks used are text files. First, we make use of the Microsoft Access wizard "Get external data". Second, we use "import" to get the corresponding text file. Third, select the option "Delimited" to specify "space" as being the delimiter in use. Finally, we choose the option "as new table"; we name the corresponding fields and we specify its data type. Once the table is in Ms-Access format we identify an index by opening the table in design view. Next, click on menu "View" and choose the option "Indexes". Finally, specify the index as a concatenation of the two fields 'src' and 'dest' and name it 'srcdest'.