USING GENETIC ALGORITHM FOR ROUTING

***Valentin Hristov*[*]*, Boris Tudjarov*[**]**

[*]*Department of Computer Systems and Technology at South West University*
66, Iv. Mihajlov, 2700 Blagoevgrad, Bulgaria
E-mail: v_hristov@swu.bg

[**]*Department of Design Fundamentals, Technical University – Sofia*
8, Kl. Ohridski blvd, 1000 Sofia, Bulgaria

Abstract

The present paper presents a program for Routing based on Genetic Algorithms. The most popular routing protocol is Open Shortest Path First- OSPF based on finding a minimum-length (cost) route between a given pair of nodes. It was proposed by Dijkstra and has been widely researched. The Dijkstra algorithm is considered as the most efficient method but when the network is very big, then it becomes inefficient since a lot of computations need to be repeated. Also it can not be implemented in the permitted time. The purpose of present paper is to propose a program based on genetic algorithm which to solve above problem as looking for routes with minimum cost between source and destination.

Keywords *– Genetic Algorithms, Open Shortest Path First - OSPF.*

1. Introduction

Nowadays the most popular routing protocol is Open Shortest Path First- OSPF based on finding a minimum-length (cost) route between a given pair of nodes. It is based on the Bellman optimization theory. But when the network is very big, then it becomes inefficient since a lot of computations need to be repeated. Also it can not be implemented in the permitted time.

The purpose of present paper is to propose a program based on genetic algorithm which to solve above problem as looking for routes with minimum cost between source and destination.

2. Genetic Algorithms

The biological foundations of the genetic algorithms (GA) are explained very briefly below.

The complete set of genetic material (all chromosomes) is called genome. Chromosomes consist of genes, blocks of DNA, each gene encodes a specific protein.

During the reproduction, the genes of the parents formed an entirely new chromosome by recombination (or crossover). New produced offspring then undergo mutation, i.e. elements of DNA change. Adaptability of the organism is measured by the success in his life.

GAs are used when pursuing a specific result (objective), when the solution requires a relatively large time resource or in cases where the solution is not known or has no solution. Algorithm starts with a set of solutions (represented by chromosomes with specific information about genes) called initial population. According to their viability are chosen solutions to form the next population (offspring). To more appropriate decisions (decisions are compared in terms of pursued result/goal) are given better chances for reproduction. New population is expected to be better than the old. This is repeated until some condition (for example: a number of generations or a sufficiently good solution) is satisfied.

The sequence in the genetic algorithm can be represented as follows:

1) generate initial random population of n chromosomes (solutions);
2) calculating the viability f (x) of each chromosome in the population n (in the target function - called "fitness function") and identification of chromosomes with priority for the next population (m in number, m <n);
3) establishing a new population by repeating following steps until the new population is completed:
3.1) preserving the predetermined number m of the best solutions (according to their fitness - the fitness function values);
3.2) election of two parental chromosomes of m chromosomes;
3.3) using of crossover to cross the parents to form the next generation (children);

3.4) using of mutation to mutate the newly created chromosomes;

3.5) pasting the new generation in the new population (adding n-m new chromosomes and filling the population);

3.6) replacement- using newly generated population for the further implementation of the algorithm;

4) stop and return the report if the final check-condition is satisfied;

5) loop, go to step 2).

In [1], [2], [3] are represented several interactive Java applets for the demonstration of the performance of genetic algorithms.

3. Program using Genetic Algorithm for Routing

As a special kind of stochastic search algorithms, genetic algorithm is a problem solving method which is based on the concept of natural selection and genetics. Genetic algorithms are inspired by Darwin's theory about evolution. Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness - the more suitable they are the more chances they have to reproduce. This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied. The space of all feasible solutions (it means objects among those the desired solution is) is called search space. Each point in the search space represents one feasible solution. Each feasible solution can be "marked" by its value or fitness for the problem. We are looking for our solution, which is one point among feasible solutions - that is one point in the search space.

Figure 1 shows the Source code of proposed program (in Python Programming Language).

When initializing the population, proposed algorithm starts from the SOURCE. The algorithm selects one of the neighbours provided that it has not been picked before. It keeps doing this operation until it reaches to DESTINATION. Both SOURCE, and DESTINATION are constants that user may change as they wish. If we are solving above problem, we are usually looking for route, which will be the best among others.

The evaluation function takes a path in the population. It gets the distance between each node pair in the path, by calling a function to read from the distance array. Adds them together and returns the sum as the cost of the path.

The program selects two individuals from the population with the lowest costs.

The crossover function takes two parents to mate. It looks for the common points in the parents. The common nodes are where these two paths intersect. Among the common points, the program selects one of them randomly. It makes the crossover from that point.

The search space can be whole known by the time of solving a problem, but usually we know only a few points from it and we are generating other points as the process of finding solution continues. The evaluation function takes a route in the population. If the offsprings' fitnesses are less than the nodes with maximum fitnesses in the population, we replace them with the nodes with the maximum fitnesses.

An experiment is developed and realized. The network topology has 20 nodes connected with 62 links as in [6]. We set two nodes as source and destination. Each link has a cost associated with them. The costs on the links are stored in matrix 20x20 (dist.txt). In this matrix, the cells with 10,000 in them represent that there is no direct link between those nodes. The program uses also a file (parents2.txt) with initial population [7].

We run the steps selection, crossover, and replace part 50 times i.e. number of generations

(Crossover probability is chosen 0.99 and mutation probability- 0.1).

On Fig. 2 is represented a part of generated, after calculations, report and more precisely path cost (the minimum, maximum, and average numbers) versus number of generations. As it can be seen from the Fig.2 the program gets close to optimum very quickly.

```
class GA:

#Read the files with info about the first population and the scoring
#---------------------------------------------------------------------------------------------------------------
    def get_info(self, file1, file2, separator):
        try :
            f1 = open(file1, 'r')
            f2 = open(file2, 'r')
            l1=f1.readlines()
            l2=f2.readlines()
        except Exception , ex :
            print "Error :", ex
        population = #[ [1,2,3,6], [1,4,5,6], ... ] (n*length)
        score_#[ [1000,1,2,1,2,3], [2,1000,2,1,2,1], ... ] (n*n)
        for i in range(len(l2)):
            population.append(l2[i].split(separator))
        for i in  range(len(population)) :
            for j in range(len(population[i])) :
                population[i][j] = int(population[i][j])
        for i in range(len(l1)):
            score_table.append(l1[i].split(separator))
        for i in  range(len(score_table)) :
            for j in range(len(score_table[i])) :
                score_table[i][j] = int(score_table[i][j])
        return [population, score_table]

#Calculate the score of one individ
#     input - the individ, the table with scores
#---------------------------------------------------------------------------------------------------------------
    def score_func(self,pop,score_table):
        score = 0
        for i in range(len(pop)-1):
            try:
                x=pop[i]
                y=pop[i+1]
                score=score+score_table[x][y]
            except: print "Error"
        return score

    def avg(self, Top_scores, population_count) :
        score_sum = 0
        for i in range(population_count) :
            score_sum += Top_scores[i][1]
        return score_sum/float(population_count)


    def build_generation(self, population , score_table) :
        a = GA()
        population_count = len(population)
        scores=[]      #list with scores of the individs and their number -> score[number , score points]
        for i in range(population_count) :
            scores.append([i, a.score_func(population[i],score_table)])

        Top_scores=sorted(scores,key=lambda i : i[1] , reverse=False)


        li= len(scores)

        new_population=[]
        counter = 0

        #crossover couples
        while counter < population_count - 1 :
            ind1 = population[ Top_scores[counter][0] ]
            ind2 = population[ Top_scores[counter + 1][0] ]

            for i in range(1,len (ind1)-1) :
                for j in range(1,len (ind2)-1) :
                    if ind1[i] == ind2[j] :
                        check = True
                        for ii in range(1,i):
                            for jj in range(j+1 , len(ind2)-1) :
                                if ind1[ii] == ind2[jj] : check = False
                        if check :
                            child = ind1[:i]+ind2[j:]
                            if child not in new_population :
                                new_population.append( child )
                        else :
                            check = True

                        try :
                            for ii in range(1,j):
                                for jj in range(i+1 , len(ind1)-1) :
                                    if ind2[ii] == ind1[jj] : check = False
                            if check :
                                child = ind2[:j]+ind1[i:]
                                if child not in new_population :
                                    new_population.append( child )
                        except : print
            counter += 2
        counter = population_count -1

        #generate the next generation
        for i in new_population :
            if i not in population:
                counter+=1
                population.append(i)
                scores.append([counter, a.score_func(i, score_table)])
        Top_scores=sorted(scores,key=lambda i : i[1] , reverse=False)
        print 'min : ',Top_scores[0][1],' , max : ', Top_scores[47][1], ' , avg : ', self.avg(Top_scores, population_count),'\n'
        next_generation=[]
        try:
            if len(Top_scores)>population_count :
                for i in range(population_count) :
                    next_generation.append(population[Top_scores[i][0]])
        except Exception, ex : print "Error :", ex
        print next_generation[0]
        return next_generation

    def run(self, gen_num, dist ='dist.txt', par = 'parents2.txt', separator = ',' ):
        population , score_table = self.get_info(dist,par,separator)
        for generation in range(gen_num):
            population = self.build_generation(population , score_table)

if __name__ == "__main__":
    ga = GA()
    ga.run(10)
```

Fig.·1.·Source·code·of·proposed·program

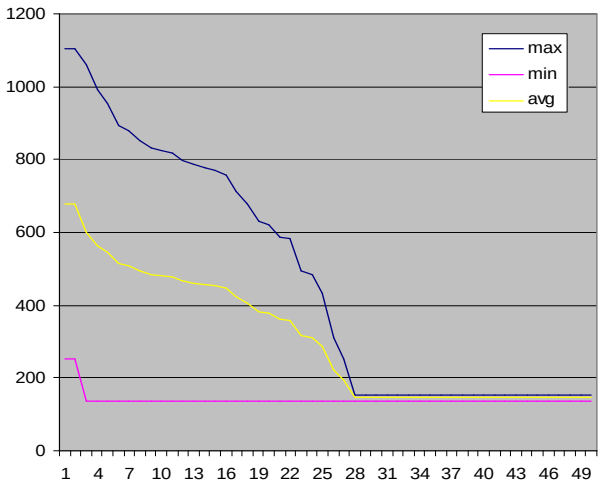

Fig. 2. Path costs versus number of generations

4. Conclusion

In present paper, we propose a program based on genetic algorithm which efficiently solves the routing problem in computer networks and more precisely looking for routes with minimum cost between source and destination.

References

[1] Туджаров Б., В. Пенчев, В. Христов, XML Моделиране на генетични алгоритми, Българско списание за инженерно проектиране, брой 8, март 2011г, с.75-80.

[2] Tudjarov B., N. Kubota, V. Penchev, V. Hristov, Web based Modeling and Calculation of Genetic Algorithms, Proceedings of the IWACIII'2011, China (accepted).

[3] www.obitko.com (accessed February 18, 2011).

[4] www.php.net (accessed February 18, 2011).

[5] cisco.netacad.net (accessed September 2, 2011).

[6] http://www.bilalgonen.com/research/publications/GA_shortest_path_BilalGonen.pdf (accessed September 2, 2011).

[7] cst.swu.bg/~vhristov/GA_for_Routing.zip (accessed September 2, 2011).