ELEKTROTEHNIŠKI VESTNIK 79(4): 197-200, 2012
ERK 2012 CONFERENCE ISSUE

Optimization of traffic networks by using genetic algorithms

Aleš Horvat[1], Aleksandar Tošić[2]

Fakulteta za matematiko, naravoslovje in informacijske tehnologije Koper,
Univerza na Primorskem, Glagoljaška 8, 6000 Koper
[1]E-mail: ales.horvat@student.upr.si
[2]E-mail: aleksandar.tosic@student.upr.si

Abstract

The paper describes a traffic optimization problem and its solving by using genetic algorithms. To evaluate the adequacy of individual solutions, a traffic simulator was built. The paper provides the basis for genetic algorithms and traffic simulators and presents our solution in more details. Our traffic simulator was designed by using an innovative approach that reduces the simulation time by preserving the amount of data needed to be processed. The paper also shows optimization results and plans for our future work.

1 Introduction

The amount of motor vehicles has been increasing over the years, mostly because of the new technology in automation. Today cars are relatively cheep and accessible, which enables most of the population to own one. Motor vehicles have been around for many years and most of the traffic networks were built in the past when the demand was not as high as today. A lot of research has been done on optimizing the currently existing networks to provide to users the best service possible. Most of the solutions are focused on traffic management systems that simulate the current traffic of a given network and can predict traffic in near future. Some of the solutions even offer a way to increase the flow of the network. Our approach optimizes the network by changing the type of intersections in the network to increase traffic flow and decrese cost.

2 Genetic algorithm

Genetic algorithm is a search heuristic that mimics the process of natural evolution. They consist of adaptive methods that simultaneously address the crowd of simple objects, where the units are used to solve search and optimization problems, or NP-problems. These algorithms are characterized by the locality, which means they have tendency to converge towards local optima, non-hierarchical structure, already mentioned co-treatment of simple objects and functionality that is the result of interaction between the facility in question. Belong to a group of evolutionary algorithms; they are based on the principles of natural evolution and the laws of genetics, where the population over several generations have to develop with the principle of natural selection and survival of the best [4].

In thir paper we will not go further than this with the description of the basics of genetic algorithms [1] and the mathematical background of it [5]. The implementation of a genetic algorithm for our project and more details about it are described hereafter.

2.1 Implementation of our genetic algorithm

Our implementation of the genetic algorithm is very similar to the basic one [3]. In Figure 1 we can see the pseudo-code of the implementation of our algorithm, which generates strings (individuals) that are actually the layout of traffic intersections in our city.

```
GeneticAlgorithm {

n = numberOfStrings;

   //Generate the first population
   population = generateString(XML) + generateRandomStrings(n-1);
   //Evaluate strings from first population
   evaluateStrings(population);

   While(true) {

      //Sort strings by performance score
      tmpSortedPop = sort(population, byPerformanceScore);

      //Generate new gen
      tmpNewGen = generateNewGeneration(tmpSortedPop);

      //Evaluate strings from new generation
      evaluateStrings(tmpNewGen);

      //Merge population and new generation
      tmpMerge = (population+tmpNewGen;)

      //Create a new population
      population = bestStrings(tmpMerge, n);

   }
}
```

Figure 1: *Pseudo-code of our genetic algorithm*

As shown in the pseudo-code (Figure 1), at the initialization of the algorithm we create an initial population of strings made by one string from the actual XML file and other “n-1” strings generated randomly. The next step is to evaluate the first population and save the performance scores. After that we enter the while loop where we sort the strings (individuals) inside the population by their performance score, so we can later select the best set of parents that are suitable for generating the descendants in the next phase – generating the new generation.

In the process of generating a new generation, two basic operations are applied - the crossover and mutation. The mutation operator has a probability parameter, which is very important to prevent irretrievable loss of good solutions in our space search. More information about these two basic operators will be described in the next subparagraphs.
After the new generation is made, we need to evaluate it and sort it by strings' performance scores. The next step is to merge our old population and our new generation and select the best "n" strings, which will live in the new population.
In our case, the genetic algorithm runs in an infinite loop until we stop it manually.

2.2 Fitness function

Fitness function is the most important feature of genetic algorithms, because it gives comparable scores and has to be implemented for each problem individually.
The role of the fitness function is to assign a performance score, represented as a numerical value, to each string (individual) from our population or generation. The performance score is used to compare strings and select the best ones from the set of all strings later on. The rating itself is supposed to represent the capacity and efficiency of the string (individual).
In our case, the fitness function gets the results from the evaluator and calculates the average time spent waiting at intersections of all agents. This is the performance score of our string (individual). More details about the evaluator will be described in the next paragraphs.

2.3 Basic operators

In the following subparagraphs we will describe in more details the basic operators that are present in our genetic algorithm with appropriate examples.

2.3.1 Reproduction

During the reproduction process, two strings get selected from the current population (sorted by performance score) and then two new strings that belong to the new generation get generated. Of course, the selection of parents takes into account the performance scores and that is why we sort the population before the reproduction process starts. This is how we guarantee to strings (individuals) with high scores to be selected more often than those with low scores, which may not be selected at all.

2.3.2 Crossover

Crossover is a binary operator where two strings (parents) generate two strings (descendants). These descendants usually replace their parents, but in our implementation we do not compare parents with descendants. We put all strings from the population and newly generated generation in the same poll and then take out the best "n" strings, which represent the new population.
Crossover is performed by selecting two strings (individuals) and then randomly choose a bit for crossing, which separates a person or a chromosome into two parts - the head and tail. Now exchange the tails between parents and in this way we get two new strings (descendants), which owns genes from both parents. This method of crossing is called simple or one-digit and we can see an example in Figure 2.

```
Parents
A: 1001110 | 1000
B: 0011011 | 1110

Descendants
A1:1001110 | 1110
B1:0011011 | 1000
```

Figure 2: *Example of crossover operation*

The crossover operation can also have a probability parameter to decide if make a crossover or just leave the descendants to be same as parents. In our case we decided to crossover all strings from our population, so the order of the crossover process is 1.00.

2.3.3 Mutation

Mutation is a unary operator because it receives one string as an input parameter, but as crossover, it operates over binary strings. Unlike the crossover operation, mutation is conducted on descendants. The execution of mutations is very small (usually order of 0.01 for each gene (bit)). In our case, the order of the mutation process is 0.01. In Figure 3, we can see the fourth gene mutated.

```
Descendant
01110010011

Descendant after mutation:
01100010011
```

Figure 3: *Mutation of the fourth gene (1 => 0)*

The mutation is responsible for the random exploration of the search space and to ensure that no item is excluded from it.

3 Traffic simulator

Traffic simulator is a mathematical modeling of the traffic networks. Traffic modeling is a vast area of research that employs many different ways of traffic simulation, which usually employs rules of behavior for agents in the traffic. Each agent behaves by the rules of the traffic simulator and interacts with other agents. Such solutions are CPU consuming and usually take a long time to simulate on real data.
Our traffic simulator is used to evaluate the fitness value of a given network, which represents its effectiveness. It is also time synchronized, meaning that the main update loop executes at a given interval, representing 1 second in real timeline. The update interval can be set before starting the simulator and also all other time variables will inherit from it to guaranty the consistency of the simulation at any given speed. Measuring the time it takes to execute an update is essential in order to provide realistic data.

3.1 Traffic network

The traffic network is represented by a graph. The roads are represented as edges and the intersections with vertexes. Since roads have directions, it is a directed graph. Each edge has a direction, a start vertex and an end vertex. The network is first built from an xml (Extensible Markup Language) file, which was built manually according to the data provided by Google Maps service [2]. The xml file contains all the information about the current traffic network including the length of each road and the type of intersections.

3.2 Intersections

Our simulator has three types of intersections. Each of them is programmed as a set of rules inherited from real data. To simulate an intersection we used FIFO (first in first out) queues as our data structure. We decided to use these queues, because they guaranty that the cars will be processed in the right order by the rules of traffic. The number of queues each intersection has depends on the number of roads leading to the intersection.

Talking about the updates, intersections are updated each time the update is called. Intersection updates depend on the type of intersection, but all of them do share one chunk of code. All of them start by updating the time the agents spent waiting in the queue of a given intersection and immediately after an agent is pulled from the queue, it gets put through a series of conditions to determine if it is allowed to continue its path.

3.2.1 Unsignalised intersections

Unsignalised intersections are the most basic type of intersection. Although it may look basic because it does not have any light signals, but it makes up by having a lot of rules, which have to be carefully implemented to avoid conflicting rules. Conflicting rules could cause the wrong car to be left to leave the queue or in some cases even cause a dead lock. A dead lock would mean two or more agents are waiting for each other to leave the queue and therefore none of them leave. In a situation like this, a traffic jam would follow and no agent would ever be allowed to leave the intersection.

3.2.2 Signalized intersections

Signalized intersections are the simplest to implement. There are only a few rules that agents need to follow. The difference between the unsignalised intersections and signalized intersections is that the signalization renders many of the traffic rules useless by minimizing the problem down to two roads instead of 4 or more. The implementation however is not very different. When intersections are updated the signalized intersection checks if it is its time to make a signal switch and if necessary, performs it. The agents are then pulled of from the corresponding queues and let back in the traffic.

3.2.3 Roundabout intersections

Roundabout intersections are very different from the others. One of the main reasons why, is because the agent's rules change. On not-roundabout intersections, if no higher rule is applicable, the agents always use the so called right-rule. The right-rule simply states that if two agents' paths collide in a given time, then the agent on the right side takes priority over the other. In roundabout intersections, this simple rule reverses into a left-rule. The structure of a roundabout intersection is actually a simple graph. Queues are kept to determine, which agent takes priority in a given situation, but after priority is given the agent does not leave the intersection, but rather drives through the graph. The graph is again built of vertexes, which in this case are slots. These slots are connected with edges that represent roads inside the roundabout as seen in figure 4.

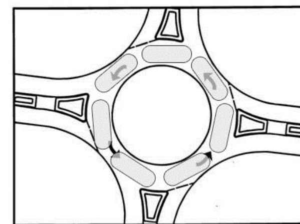

Figure 4: Ex*ample of roundabout slots*

A single slot can be occupied by a single agent at a given time. The number of slots a roundabout intersection has, is determined by its size. Size of a roundabout intersection is read from the xml file at the beginning of the first simulation.

If an agent pulled from the queue is given priority the neighbor slot is checked. If the slot is occupied the agent waits, else it occupies it. Agents switch slots as if the slots were in a rounded list. They switch them until they reach the slot that is the neighbor to the road it needs to go to. When such a slot is reached, the agent leaves the roundabout and continues on its path.

4 Agents

Vehicles in the traffic network are represented as agents. Each agent is an object with many properties that can be found in four states in a given time of the simulation. In this paragraph we will describe our implementation of agents in more details.

4.1.1 Initialization

The initialization state occurs when the agent still does not exist in the traffic network. Before the agent is unleashed into the network it needs a path. There are many different ways of setting an agent's path, but in order for the simulation to be realistic an agent needs to have a realistic path. Thinking about defining paths for our agents we concluded that each car ends its path on a parking lot. Our agents inherit this idea and for that reason all major parking lots are also marked in the xml file. We also isolated the major roads that lead into the city and out of the city. Having defined the start and end point of an agent we check all the intersections on its path. This was done with Floyd–Warshall [6] algorithm that generates the shortest path between all vertexes in

the Graph. For constructing the shortest path, the road length was used as distance.

4.1.2 Driving

With the driving state we describe the agent that is on the road from one intersection to another. To decrease the CPU consumption we implemented the driving state as sort of a sleeping state. Because our genetic algorithm is optimizing only the flow of each intersection, the roads are not that important. Nothing that would affect the number of cars that need to pass an intersection can happen when the agent is in driving state. When an agent leaves the intersection, the algorithm calculates the time that the agent needs to get to the next intersection by taking in consideration the speed of the agent and the length of the road it will drive on. The time needed is then mapped into the simulation time and saved into the agents' object. After the time variable is passed, it means the agent has reached an intersection and an update is needed. When in state of driving, the agents do not consume much CPU.

4.1.3 Inside the intersection

This state describes the behavior of the agents when they are inside an intersection. Similar to the driving state, the agents in this state do not consume much CPU power, since they are inside a queue. When an intersection is updated, only agents that are first in the queue get processed at once. It is very important to update time variables of all the cars inside the queues, because the time spent waiting on intersections plays an important role in the fitness function. The data structure queue was never built for operations on all elements, instead it performs quick operations on the first element. The time update was structured to give each agent a timestamp when entering the intersection state and another timestamp when leaving it. The difference between those timestamps is the amount of simulation time the agent spent waiting in the intersection.

4.1.4 Destination reached

When an agent reaches its destination, it gets removed from the data structures of the simulator and left for the garbage collector to clean the object. Just before removing the agent, the data stored inside the object is retrieved and remembered till the end of the simulation. From this data the genetic algorithm can evaluate the fitness value of the traffic network that was tested.

5 Results

With limited resources and time we could not achieve the desired number of generations. Our population consisted of 200 strings (individuals). The frequency of generating agents was taken from real data of the city of Koper. The city of Koper is known troughout Slovenia to be the city with most roundabout intersections. Measuring the flow of traffic each type of intersection has, it can be proved that when there is high traffic, the roundabout proves to have the highest flow. The fact that Koper is mostly covered with roundabout intersection means that the optimization level is not expected to be high. Looking at the results we can see that even with a few generations there, we reached some level of optimization. The main roads were mostly left with roundabout intersections while the less populated roads were replaced with other types of intersections. In figure 5 we can see the function that represents the optimization. Since our fitness function is represented by the agents' waiting time on intersections, the genetic algorithm is programmed to minimize the fitness. The function starts with a significant drop, but soon comes to a steady low drop, which indicates that the genetic algorithm ether got stuck in a local minimum because of the generation size or we are getting close to an optimal solution.

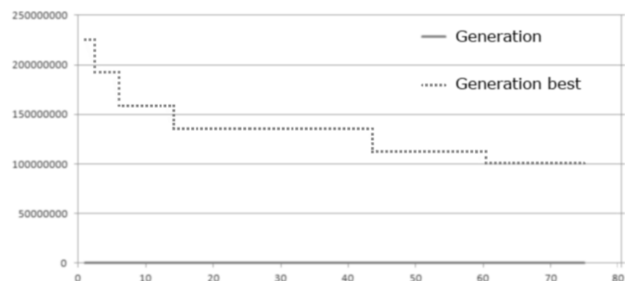

Figure 5: Results of testing

6 Future work

For future work we are working on new version of the traffic simulator that will run on a distributed computer system. This will enable us to test many individuals generated by the genetic algorithms at once. The system will work on master-slave concept where the master will be the genetic algorithm and slaves will be multiple instances of the traffic simulator. We hope this will speed up the optimization process immensely. We are also working on a component that will automatically generate the initial XML file that represents the actual layout of traffic intersections in a city. Another goal is to use different heuristic algorithms to search for the optimal graph. Genetic algorithms maybe produce good results but they do not use any additional knowledge in search of the optimal solutions. We intend to try other approaches and compare the results.

7 References

[1] Filipič, B.: Genetski algoritmi. Informatica 4/92, 1992.

[2] Google: Google Maps (April 2012), http://maps.google.com

[3] Neville, M., and Sibley, A.: Developing a Generic Genetic Algorithm. December 18, 2002.

[4] Sipper, M.: On the origin of environments by means of natural selection. AI Magazine, 22 (2001): 133-140

[5] Taraneko, A.: Genetski algoritmi. Diplomsko delo, Maribor 2001.

[6] Warshall Floyd Robert. (June 1962). "Algorithm 97: Shortest Path". Communications of the ACM 5 (6): 345. doi:10.1145/367766.368168.