INT J COMPUT COMMUN, ISSN 1841-9836
Vol.7 (2012), No. 3 (September), pp. 482-493

A Fast and Scalable Re-routing Algorithm based on Shortest Path and Genetic Algorithms

J. Lee, J. Yang

Jungkyu Lee
Cyram Inc., 516 Seoul National University Research Park
Naksungdae-dong, Gwanak-gu, Seoul 151-919 Koera
E-mail: jklee@cyram.com

Jihoon Yang
Department of Computer Science, Sogang University
1 Sinsu-dong, Mapo-gu, Seoul 121-742 Koera
E-mail: yangjh@sogang.ac.kr

Abstract: This paper presents a fast and scalable re-routing algorithm that adapts to dynamically changing networks. The proposed algorithm, DGA, integrates Dijkstra's shortest path algorithm with the genetic algorithm. Dijkstra's algorithm is used to define the predecessor array that facilitates the initialization process of the genetic algorithm. Then the genetic algorithm keeps finding the best routes with appropriate genetic operators under dynamic traffic situations. Experimental results demonstrate that DGA produces routes with less traveling time and computational overhead than pure genetic algorithm-based approaches as well as Dijkstra's algorithm in large-scale routing problems.
Keywords: Evolutionary algorithm, routing in dynamic networks, car navigation system.

1 Introduction

The car navigation system has become a very useful tool for many drivers. When a driver turns on a car navigation system and inputs where he or she wants to go, the system searches the map and finds the best route (e.g. shortest path) to the destination. Recently, in addition to such a basic functionality, car navigation systems are equipped with real-time traffic information services like TPEG (Transport Protocol Experts Group) [1–6]. Here, the navigation system is provided with the traffic information on current road conditions, with which it re-computes the best route with minimal expected travel time. Unfortunately, such traffic information is not truly real-time but delivered from a central server at certain intervals. In addition, updating the entire map with the new information delivered from the server causes an exorbitant overhead. In this paper, we propose a novel approach to deal with these problems and to produce the best route dynamically. Our algorithm integrates Dijkstra's shortest path algorithm [7] with a genetic algorithm [8], and thus named DGA. The former is for incorporating useful prior knowledge on the network (e.g. distance between two places) and facilitating the initialization process of the genetic algorithm, and the latter is for finding the best routes. (Detailed descriptions on DGA will be given in Section 3.) DGA re-computes the routes quickly whenever new real-time traffic information is available. A car is assumed to send the traffic information (e.g. its speed) to the vehicles it meets during the trip via wireless communication. This direct and local communication among vehicles provides genuine real-time information and obviates the use of the expensive central server.

This paper is organized as follows: Section 2 briefly introduces a representative genetic algorithm-based approach to the shortest path problem proposed by Ahn [9] which will be compared with DGA. Section 3 describes DGA. Section 4 presents the results of the experiments designed to evaluate the performance of DGA. Section 5 concludes with a summary and future research directions.

2 Related Work: Genetic Algorithm for Shortest Path Problem

The genetic algorithm (GA) is one of the global search heuristics inspired from biology and has been successfully applied to a variety of optimization problems [8, 10]. A great deal of research on GA-based shortest path search has been carried out in various communication network applications [9, 11–15], among which [9, 14, 15] are related to car navigation systems. Ahn's method [9] is one of the most representative applications of GA to the shortest path routing problem. However, though Ahn's method was able to find a good solution with solid theoretical results, it worked only for moderate-sized networks. In fact, our experiments with the algorithm failed to produce solutions within a reasonable period of time for networks with more than 10,000 nodes. Considering real-world networks where there exist huge number of nodes (or places), Ahn's approach is thus far from applicable. There exists another GA-based approach for the car navigation system proposed by Kanoh [14, 15]. Kanoh's approach is similar to DGA in that it computes routes by considering dynamic road conditions and initializing the population using Dijkstra's algorithm. However, the motivation of their algorithm is quite different from ours and not fully comparable: They use GA for improving the quality of solution in terms of multi-objective criteria (e.g. traveling time, route length, number of signals, number of right turns, etc.), whereas our algorithm focuses on re-routing. In addition, Kanoh's approach was evaluated only with small networks (of less than 20,000 nodes) though the computational overhead was claimed to be low.

The main contribution of this paper is the design of an efficient algorithm that can be deployed in a car navigation system and be used frequently for re-routing in a large-scale network only with locally-transmitted traffic information. To the best of our knowledge, there does not exist a GA-based algorithm to the shortest path problem that can cope with dynamic situations in a huge network. Since DGA has characteristics common with Ahn's GA, we briefly introduce the method here. (See [9] for detailed descriptions.)

2.1 Genetic Representation

A chromosome (representing a candidate solution path) is variable-length and consists of the sequence of positive integers that represent the IDs of nodes through which a routing path passes. The gene at the first and the last loci are reserved for the source and the destination nodes, respectively.

2.2 Population Initialization

The chromosomes are initialized randomly. Starting from the source, a chromosome encodes a routing path by successively selecting the next node at random among the neighboring nodes that are linked to the current node. Note that the chance for generating a valid path is fairly slim due to the randomness, which makes the approach infeasible for large networks, as verified in Section 4.

2.3 Fitness Function

The fitness function of the ith chromosome, f_i, is defined as

$$f_i = \left(\sum_{i=1}^{m_i} C(g_i(j), g_i(j+1))\right)^{-1} \tag{1}$$

where m_i is the length of the chromosome, $g_i(j)$ represents the gene at the jth locus, and $C(g_i(j), g_i(j+1))$ is the link cost between node $g_i(j)$ and $g_i(j+1)$.

2.4 Selection

The tournament selection without replacement is used. In other words, non-overlapping random sets of 2 chromosomes are chosen from the population, and the chromosome with higher fitness was selected from each set to survive in the next generation.

2.5 Crossover

The concept of crossover is depicted in Fig. 1. First, the crossover points are determined by randomly choosing a common gene appearing in both parent chromosomes. Then the chromosomes are interchanged with respect to the crossover points and the offsprings are generated.

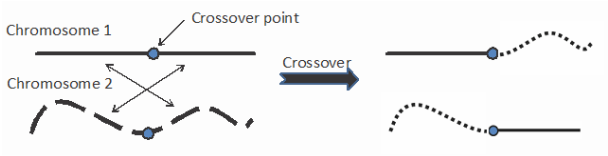


Figure 1: The concept of crossover.

2.6 Mutation

Typically, GA performs mutation by changing or flipping the genes in the candidate chromosome, thereby maintaining the genetic diversity. Here, the mutation operation attempts to maintain the diversity in the population by modifying the current path represented by a chromosome. For a chromosome, a gene is randomly selected as a mutation point. Starting from the mutation point, a sequence of neighboring nodes are randomly chosen to define a complete path (i.e. until the node chosen last is the destination). The concept of mutation is depicted in Fig. 2.

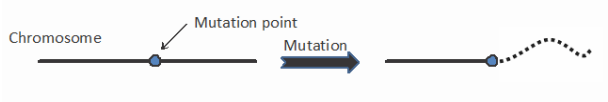


Figure 2: The concept of mutation.

2.7 Repair Function

Note that a chromosome produced by the crossover operation may contain a loop in the path it represents, which is an invalid solution. To make the path valid, the repair function was proposed. As shown in Fig. 3, the repair function eliminates a loop by finding the intersection (or repeated) node and removing the intermediate nodes in the loop.

Figure 3: The concept of repair function.

For example, assume that the following chromosome is produced:

$$1, 2, 3, 4, 5, 6, 3, 7, 8$$

Then the repair function finds (and keeps a single occurrence of) the intersection node 3 and removes the intermediate nodes 4, 5 and 6. The resulting chromosome is:

$$1, 2, 3, 7, 8$$

2.8 The overall algorithm

Now, Ahn's GA for finding the shortest path is described in Algorithm 1.

Algorithm 1. Ahn's GA
1: Initialize the population;
2: **repeat until** convergence
3: Calculate the fitness of individuals in the population;
4: Do selection;
5: Do crossover;
6: Remove loops by repair function;
7: Do mutation;
8: **end**

The condition for the convergence of the algorithm is if all chromosomes are identical.

3 DGA

We describe our algorithm, DGA, in this section. The purpose of DGA is to adapt to the dynamically changing networks and to re-route the shortest path fast. As aforementioned, DGA inherits the characteristics of both Dijkstra's shortest path algorithm and GA. The former is to initialize the population in the latter with meaningful candidate solutions (i.e. paths) instead of random ones. For instance, the chromosomes can be generated based on useful information such as the distance or average vehicle speed between two nodes. Among the various data structures used for Dijkstra's algorithm, the overflow bag introduced by Cherkassky *et al.* [16] was adopted in our work. (Cherkassky *et al.* had developed the overflow bag to reduce the memory requirement of Dijkstra's algorithm with the bucket data structure proposed by Dial [17].) Instead of Dijkstra's algorithm, any single-source shortest path algorithm (e.g. Bellman-Ford algorithm [18]) can be also used for DGA.

3.1 Population Initialization

The random population initialization of Ahn's GA does not work well for large-scale networks since the chance for generating invalid paths becomes very high as explained in Section 2. To overcome this problem, DGA makes use of Dijkstra's algorithm and produces a *predecessor array* as described in Fig. 4. First, from the start (source) node, the shortest paths to all the other nodes including the goal (destination) node are computed by Dijkstra's algorithm. Then, for the shortest path from an arbitrary node a to the goal node, all the links on the path are stored in the form of a (*direct*) predecessor array *pred* which is s sequence of nodes constituting the path in a reverse order (i.e. from the goal to a). Fig. 4(a) shows an example of *pred*. Once *pred* is constructed, the shorted path from the goal to a can be easily obtained by a call $GetPath(pred, goal, a)$ defined as follows, which makes fast initialization of the population possible:

Subroutine 1. GetPath($pred$, x, y)
// Compute the path from x to y using $pred$.
1: Set current node $s_{cur} = x$ and $path = [s_{cur}]$;
2: **repeat**

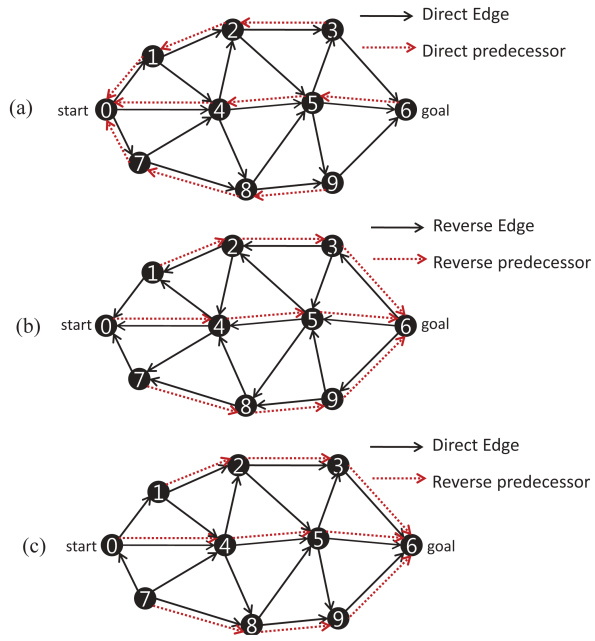


Figure 4: Example of reverse graph and predecessor array.

3: $s_{cur} = pred(s_{cur})$;
4: $path = [path\ s_{cur}]$;
5: **until** $(s_{cur} = y)$
6: **return** *path*;

Let $G(N, E)$ be a directed graph with the set of nodes N and the set of edges E. We define the reverse graph of a directed graph $G(N, E)$ as the graph $G_{rev}(N, E_{rev})$ with

$$E_{rev} = \{(u, v)|(v, u) \in E\} \tag{2}$$

For example, if we reverse all the edges of $G(N, E)$ in Fig. 4(a), we get the reverse graph $G_{rev}(N, E_{rev})$ in Fig. 4(b), with which we can compute the shortest paths from the goal node to all the other nodes by Dijkstra's algorithm. Then we can compute a *reverse-pred* for $G_{rev}(N, E_{rev})$ which is a series of nodes constituting the path from an arbitrary node to the goal node as shown in Fig. 4(b). Now, if we consider the original graph $G(N, E)$ with *reverse-pred* computed with resect to the reverse graph $G_{rev}(N, E_{rev})$ as in Fig. 4(c), we can see that *reverse-pred* contains pointers to the optimal node to travel from any node in $G(N, E)$ to reach the goal.

Suppose that an agent travels around the graph to arrive at the destination node. Even if the agent deviates from the optimal path, it can eventually reach the destination by following the next node that *reverse-pred* points. In other words, *reverse-pred* serves as a guide to the lost or wandering agents in the network. For instance, in Fig. 4(c), if an agent on node 0 moves to node 1 which is not on the optimal path, it can adapt to the situation and follow the optimal path from node 1 by consulting *reverse-pred*. Like this, if an agent deviates from the shortest path on any node, it can rectify its plan and follow the optimal path to the goal.

In the field of reinforcement learning, such a *reverse-pred* is called the optimal policy [19]. The optimal policy $\pi^* : N \mapsto N$ is the mapping from the current node to the next node that is on the optimal path. As a scheme to apply the optimal policy π^*, the ϵ-greedy method is used which picks the best move most of the times but allows a random move with a small probability of ϵ. This can be summarized as

$$\epsilon\text{-}greedy(s, \pi^*) = \begin{cases} \pi^*(s) & \text{if } \zeta < \epsilon \\ \text{random move} & \text{otherwise} \end{cases} \tag{3}$$

where $\pi^*(s)$ is equivalent to *reverse-pred(s)*, s is the current node, and ζ is a random number generated between 0 and 1 to be compared with ϵ. Now, the population can be initialized by Subroutine 2:

Subroutine 2. PopulationInit(π^*, *start*, *goal*)
1: **for** $i = 0$ **to** $PopulationSize - 1$
2: Set $s_{cur} = start$ and $chromosome[i] = [s_{cur}]$;
3: **repeat**
4: $s_{cur} = \epsilon\text{-}greedy(s_{cur}, \pi^*)$;
5: $chromosome[i] = [chromosome[i]\ s_{cur}]$;
6: **until** ($s_{cur} = goal$)
7: **end**
8: **return** *chromosome*;

There are several advantages in our population initialization method. First, the amount of data needed in a random initialization method (like Ahn's GA) is even larger than in our algorithm since the former requires the information on the overall network topology while the latter only refers to the optimal policies in the predecessor array. Therefore, in real-world situations where the size of the network is huge, DGA has significantly less overhead than Ahn's GA. Second, our initialization method increases the probability of generating valid chromosomes while the probability with random population initialization is inversely, exponentially proportional to the length of the path. This is theoretically proved in Claim 1, and experimentally verified by large networks wherein valid chromosomes could not be generated within reasonable time.

Claim1. Let x be a random variable drawn from *Bernoulli*(m) distribution defined as follows: If an agent reaches the destination node in reasonable time by selecting the next node randomly, then $x = 1$, otherwise $x = 0$. That is, the probability $P(x = 1)$ is m. The agents is assumed to makes l independent selections of next nodes. We claim that the probability $P_{rand}(l)$ of generating a valid path (chromosome) with length l in reasonable time with random population initialization is

$$P_{rand}(l) = m^l \tag{4}$$

Meanwhile, let y be a random variable drawn from *Bernoulli*(1) defined as follows: If an agent reaches the destination node in reasonable time by executing the optimal policy, then $y = 1$, otherwise $y = 0$. That is, the probability $P(y = 1)$ is 1. We now claim that the probability $P_{\epsilon\text{-}greedy}(l)$ of generating a valid path with length l in reasonable time with ϵ-*greedy* selection is

$$P_{\epsilon\text{-}greedy}(l) = m^{l(1-\epsilon)} \tag{5}$$

Proof:

$$\begin{aligned} P_{rand}(l) &= \overbrace{P(x=1) \times P(x=1) \times \cdots \times P(x=1)}^{l} \\ &= \overbrace{m \times m \times \cdots \times m}^{l} \\ &= m^l \end{aligned}$$

$$
\begin{aligned}
& P_{\epsilon\text{-}greedy}(l) \\
&= \overbrace{\underbrace{P(x=1)\cdots P(x=1)}_{l\epsilon}\underbrace{P(y=1)\cdots P(y=1)}}^{l} \\
&= \overbrace{m\cdots m\cdot\underbrace{1\cdots 1}_{l\epsilon}}^{l} \\
&= m^{l-l\epsilon} \\
&= m^{l(1-\epsilon)}
\end{aligned}
$$

□

For example, let $m = 0.995$, $l = 50$, $\epsilon = 0.5$. Then,

$$P_{rand}(l) = (0.995)^{50} = 0.7783$$
$$P_{\epsilon\text{-}greedy}(l) = (0.995)^{25} = 0.8822$$
$$P_{\epsilon\text{-}greedy}(l)/P_{rand}(l) = 1.1335$$

However, if $l = 1000$,

$$P_{rand}(l) = (0.995)^{1000} = 0.0067$$
$$P_{\epsilon\text{-}greedy}(l) = (0.995)^{500} = 0.0816$$
$$P_{\epsilon\text{-}greedy}(l)/P_{rand}(l) = 12.1791$$

3.2 Fitness Function

Since the purpose of the proposed algorithm is to re-route the shortest path considering dynamic traffic situations, the fitness of each chromosome is based on the traveling time instead of the physical distance between the source and the destination. So we redefine $C(x, y)$ in Eq. (1) with the traveling time from node x to node y, and represent the costs as a hash table. We can define the set of all edges comprising the chromosomes as

$$\Omega = \{(y_{i,j}, y_{i,j+1})|y_{i,j}\} \tag{6}$$

where $y_{i,j}$ is the jth gene in the ith chromosome in the population. Then the hash table contains the edge $(x, y) \in \Omega$ with its associated cost. This scheme provides fast access of the edge costs, and requires less communication overhead of real-time traffic information only for the edges in Ω instead of all the edges in the network.

3.3 Selection

Although the time complexity of tournament selection without replacement used in Ahn's GA is not costly ($O(|chromosomes|)$ where $|chromosomes|$ is the number of chromosomes in the population), it has a problem that good chromosomes can dropout early if they are met with chromosomes with higher fitness values in the tournament. We devised the following selection method to solve the problem.

1. The average fitness of all chromosomes in the population is calculated.
2. The chromosomes with above-average fitness survive in the next generation, and the chromosomes with below-average fitness are weeded out.
3. The deleted chromosomes are replaced by the survived ones at random.

Each step of the above selection method has time complexity of $O(|chromosomes|)$, which also makes the total complexity of $O(|chromosomes|)$. With asymptotically the same computational overhead, our selection method can overcome the early dropout problem.

3.4 Crossover

As described earlier, the crossover operator in Ahn's GA finds all genes that appear in both parent chromosomes and then chooses one of them randomly. Let α and β be the lengths of such two chromosomes, respectively. Then the time required to find the crossover point is $O(\alpha\beta)$. If α and β increase for large networks, the cost for searching the crossover point become expensive. To optimize the process of crossover point search, we use the following subroutine.

Subroutine 3. SearchCrossPoint(x_1, x_2)
// x_1, x_2 are two chromosomes to crossover.
1: s_1 = *rand* % *size*(x_1);
2: **if** (s_1 == 0) e_1 = *size*(x_1) − 1 **else** e_1 = s_1 − 1; **end**
3: s_2 = *rand* % *size*(x_2);
4: **if** (s_2 == 0) e_2 = *size*(x_2) − 1 **else** e_2 = s_2 − 1; **end**
5: **for** i = s_1 **to** e_1
6: **for** j = s_2 **to** e_2
7: **if** ($x_1[i]$ == $x_2[j]$) **return** i, j; **end**
8: **if** (j == *size*(x_2) − 1) j = 0; **end**
9: **end**
10: **if** (i == *size*(x_1) − 1) i = 0; **end**
11: **end**

Note that $SearchCrossPoint(x_1, x_2)$ determines the random crossover point of common genes starting from arbitrary positions of two chromosomes x_1 and x_2, and keeps comparing the genes in a circular way (i.e. after considering the last gene, it starts from the first gene of the chromosome). As soon as the first match occurs, the subroutine returns the genes. Otherwise, it repeats the comparisons for all possible pairs of positions. The remaining steps of the crossover operation (i.e. generation of offsprings from the crossover point and application of the repair function) remain the same as Ahn's GA.

3.5 Mutation

As described in Section 3.2, only the edge $(x, y) \in \Omega$ can appear in the chromosomes. If an edge $(x', y') \notin \Omega$ appears in the chromosomes in a new generation as a result of mutation, the traffic information on (x', y') needs to be fetched to compute the shortest path, which causes additional communication. To prevent this overhead, the mutation is omitted in our algorithm. In our preliminary experiments, there was no significant difference in performance (in terms of the path quality and the convergence speed) between two approaches where the mutation was applied or not.

3.6 The overall algorithm

DGA can be summarized as Algorithm 2.

Algorithm 2. DGA
1: Construct π^* (from *reverse-pred*) and the initial population Y (Section 3.1).
2: Remove loops in chromosomes in Y by repair function (Section 2.7).
3: Construct hash table for edges $(x, y) \in \Omega$ (Section 3.2).
4: **repeat until** convergence
5: Calculate the fitness of population Y (Section 3.2).
6: Do selection (Section 3.3), crossover (Section 3.4), and remove loops in Y (Section 2.7).
7: **end**
8: **return** Y

Table 1: Parameter settings.

Network ID	Network Size	ϵ	Population Size
#1	50	0.5	20
#2	100	0.5	20
#3	200	0.5	20
#4	400	0.5	20
#5	800	0.5	20
#6	2000	0.5	20
#7	4000	0.5	20
#8	8000	0.5	20
#9	20000	0.5	30
#10	40000	0.6	30
#11	80000	0.6	40
#12	160000	0.6	40
#13	320000	0.6	40
#14	640000	0.7	40
#15	800000	0.7	40
#16	1000000	0.7	40
#17	1200000	0.7	50

As in Ahn's GA, the condition for the convergence of the algorithm is if all chromosomes are identical.

4 Experiments

4.1 Setup

DGA is implemented in C and all the experiments were conducted on Intel Core2Quad processors (2.40GHz clock rate). We generated strongly connected random networks of size (i.e. number of nodes) ranging 50-1,200,000, and the distance $dist(i, j)$ between node i and j is assigned with a random integer in [1-9,999].

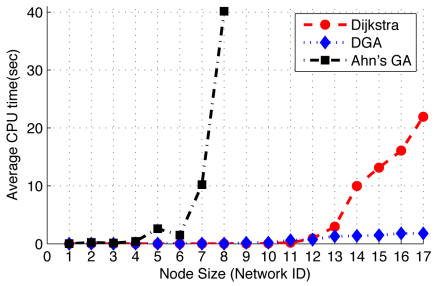
There are two parameters in DGA: ϵ (of *ϵ-greedy* strategy) and the population size, except the crossover probability which was set to 1. The lower ϵ and the higher population size we set, the greater the diversity in a population will be. We applied parameter settings for each network as illustrated in Table 1.

To evaluate the performance of DGA under real-time traffic conditions, we also constructed a traffic simulator as follows:

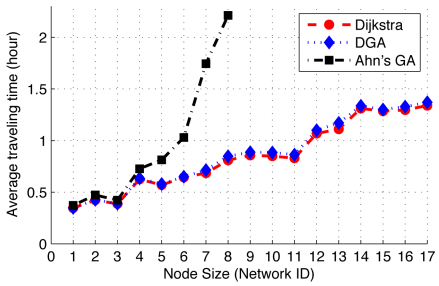
1. A vehicle travels around the networks (generated as in Table 1) to arrive at the destination node.
2. Whenever a vehicle makes a move from the current to the next node, all the edge costs of the network are changed dynamically by

$$C(i, j) = \frac{dist(i, j)}{v_{ij}}, \text{ for each node } i, j \tag{7}$$

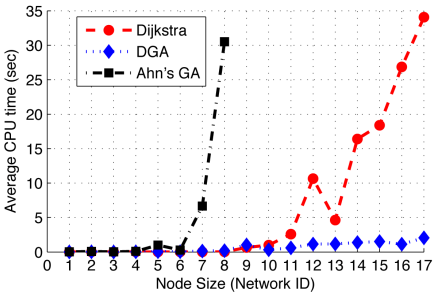
   where v_{ij} is velocity of vehicles on the link (road) between node i and j which is drawn from two normal distributions with the same mean but different standard deviations (i.e. $\mathcal{N}$(80km/s, 20km/s) and $\mathcal{N}$(80km/s, 40km/s)) to see the behavior of the algorithms in different situations.
3. A vehicle re-routes the path whenever the edge costs are changed.



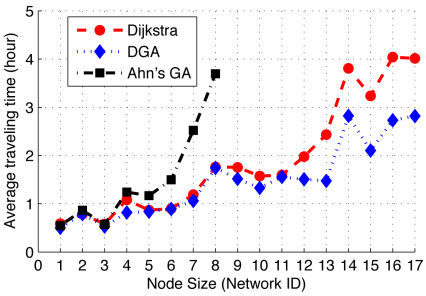
(a) A comparison of average CPU time (low variation)



(b) A comparison of average traveling time (low variation)



(c) A comparison of average CPU time (high variation)



(d) A comparison of average traveling time (high variation)

Figure 5: Simulation results.

4. There are three types of vehicles implementing three algorithms: Dijkstra's algorithm (implemented with the overflow bag structure as in [16]), Ahn's GA, and DGA.

5. For 300 randomly generated source-destination node pairs, the performance (in terms of the CPU time and the traveling time) are measured and averaged.

4.2 Results

The experimental results are shown in Fig. **??** where (a), (b) are for the networks with less drastic changes in the velocity of vehicles (i.e. standard deviation of 20km/s), and (c), (d) are with more drastic changes (i.e. standard deviation of 40km/s). The x-axis of the graphs represents the network ID of Table 1. The y-axis represents the average CPU time in (a) and (c), and the average traveling time in (b) and (d). The results of Ahn's GA for above 20,000 node-sized networks are not included due to the excessive running time.

As shown in Fig. **??**(a) and (c), it is impossible for Ahn's GA to find the path in reasonable time. For networks with less than 40,000 nodes, the average CPU time of Dijkstra's algorithm and DGA are similar. However, as the size of the network increase, DGA outperforms Dijkstra's algorithm by a large margin. This is because Dijkstra's algorithm computes a new path over the entire nodes for each traffic condition, while DGA adjusts the path based on the locally updated traffic condition with the predecessor array.

As shown in Fig. **??**(b) and (d), the quality of the path (i.e. average traveling time) of Ahn's GA is even inferior to other algorithms. Fig. **??**(b) verifies that DGA produces paths as good as the ones produced by Dijkstra's algorithm for less dynamic networks. However, DGA outperforms Dijkstra's algorithm for highly dynamic networks as shown in Fig. **??**(d). This is because Dijkstra's algorithm sticks to the current traffic conditions too much and might make inefficient changes in the path (e.g.

detours), while DGA makes local adjustments to the current path and produces stable solutions. This verifies the feasibility of DGA in real-world car navigation systems where traffic conditions are constantly and possibly drastically changing.

5 Conclusion

We have presented a fast and scalable re-routing algorithm, DGA, that adapts to dynamically changing networks. In addition to the theoretical soundness, the experimental results have also shown the outstanding performance of DGA on large networks. DGA is a good candidate for intelligent car navigation systems since it is capable of re-routing the optimal path swiftly whenever new traffic information is available. In addition, DGA has a significant merit of requiring the minimal traffic information and reducing the communication cost between the car navigation system and the central server, or among the navigation systems in each vehicle.

We have not tested DGA with real-world maps and traffic information due to the lack of required infrastructures (e.g. communication, information collection). Therefore, DGA needs to be deployed and fully evaluated when the infrastructures become available. Also, DGA can be extended to work in the unexplored environment where the agent does not have the global picture on the environment where it belongs. In such an environment, Markov decision processes (MDPs) [20] and reinforcement learning approaches [19, 21] can be useful to learn the optimal routing policy, as attempted in [22]. In addition, DGA can be also extended to consider additional criteria for navigation similar to [15, 23, 24]. Some of these research issues are currently in progress.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2009-0076594) to Jihoon Yang, the corresponding author.

Bibliography

[1] EBU BPN. 027-1 ŇTransport Protocol Experts Group (TPEG) Specifications.

[2] EBU BPN. 027-2 ŇTransport Protocol Experts Group (TPEG) Specifications.

[3] EBU BPN. 027-3 ŇTransport Protocol Experts Group (TPEG) Specifications.

[4] EBU BPN. 027-4 ŇTransport Protocol Experts Group (TPEG) Specifications.

[5] EBU BPN. 027-5 ŇTransport Protocol Experts Group (TPEG) Specifications.

[6] EBU BPN. 027-6 ŇTransport Protocol Experts Group (TPEG) Specifications.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[8] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[9] C.W. Ahn and R. S. Ramakrishna. A genetic algorithm for shortest path routing problem and the sizing of populations. *IEEE Transactions on Evolutionary Computation*, 6(6):566–579, 2002.

[10] D. E. Goldberg. *Genetic Algorithms in Search and Optimization*. Addison-wesley, 1989.

[11] Q. Zhang and Y. W. Leung. An orthogonal genetic algorithm for multimedia multicast routing. *IEEE Transactions on Evolutionary Computation*, 3(1):53–62, 1999.

[12] F. Xiang, L. Junzhou, W. Jieyi, and G. Guanqun. QoS routing based on genetic algorithm* 1. *Computer Communications*, 22(15-16):1392–1399, 1999.

[13] Y. Leung, G. Li, and Z. B. Xu. A genetic algorithm for the multiple destination routing problems. *IEEE Transactions on Evolutionary Computation*, 2(4):150–161, 1998.

[14] H. Kanoh. Dynamic route planning for car navigation systems using virus genetic algorithms. *International Journal of Knowledge-based and Intelligent Engineering Systems*, 11:65–78, 2007.

[15] H. Kanoh and K. Hara. Hybrid genetic algorithm for dynamic multi-objective route planning with predicted traffic in a real-world road network. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, pages 657–664. ACM, 2008.

[16] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[17] R. B. Dial. Algorithm 360: Shortest-path forest with topological ordering [H]. *Communications of the ACM*, 12(11):632–633, 1969.

[18] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[19] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. MIT Press, 1998.

[20] R. Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 6, 1957.

[21] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, pages 237–285, 1996.

[22] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Proceedings of the Advances in Neural Information Processing Systems*, pages 671–671, 1994.

[23] M. Stanojević, M. Vujošević, and B. Stanojević. Number of Efficient Points in some Multiobjective Combinatorial Optimization Problems. *International Journal of Computers, Communications & Control*, 3(Suppl.): 497-502, 2008.

[24] I. Harbaoui Dridi, R. Kammarti, M. Ksouri, and P. Borne. Multi-Objective Optimization for the m-PDPTW: Aggregation Method With Use of Genetic Algorithm and Lower Bounds. *International Journal of Computers, Communications & Control*, 6(2): 246-257, 2011.