Worst-Case Execution Time Test Generation for Augmenting Path Maximum Flow Algorithms using Genetic Algorithms

Viktor Arkhipov, Maxim Buzdalov, Anatoly Shalyto
St. Petersburg National Research University
of Information Technologies, Mechanics and Optics
49 Kronverkskiy prosp.
Saint-Petersburg, Russia, 197101
Email: {arkhipov,buzdalov}@rain.ifmo.ru, shalyto@mail.ifmo.ru

***Abstract*—Worst-case execution time tests can be tricky to create for various computer science algorithms. To reduce the amount of human effort, authors suggest using search-based optimization techniques, such as genetic algorithms. This paper addresses difficult test generation for several maximum flow algorithms from the augmenting path family. The presented results show that the genetic approach is reasonably good for the well-studied algorithms and superior for the capacity scaling algorithms. Moreover, tests which are generated against one algorithm seem to be hard for other algorithms of this family.**

I. Introduction

Worst-case execution time testing has always been the most relevant measure of algorithm performance. One of the main problems is that worst-case test generators have to heavily depend not only on the problem that the algorithm solves, but on the algorithm itself as well. In addition, to design and implement such generators, a scientist must have a deep insight of how exactly the algorithm works. This leads to the situations when for certain algorithms the worst-case tests are, in fact, unknown.

One of the possible solutions to the problem of worst-case test data generation is to apply search-based software engineering techniques [1]. In this paper, tests are generated using genetic algorithms.

The problem which is considered in this work is the well-known maximum flow problem. This problem is chosen because a number of different algorithms for solving it are known, as well as some good test generation algorithms exist. We further refine our domain to augmenting path algorithms [2], [3], including their capacity scaled versions [4]. Our results are compared to the performance of DIMACS data generators [5].

The rest of the paper is structured as follows. First, different means of fitness evaluation are discussed. Second, the common properties of genetic algorithms which are used in this work are described. Third, several crossovers are considered and the best one is empirically chosen. Fourth, the results for the chosen maximum flow algorithms are shown. Fifth, we compare the generated tests with the DIMACS test data. Discussion, conclusion and future work sections will be the last.

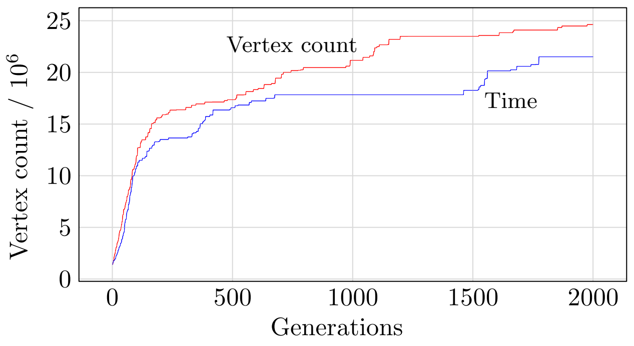

Fig. 1. Fitness functions: Time vs. Vertex count

II. Fitness Measurement

In previous works [6]–[9] it is stated that, in various instances of worst-case execution time test data generation, optimization of execution time directly may not lead to good results due to its noisy, quantized and platform-dependent nature. Instead, it is recommended to use internal data from the algorithms, such as the number of function calls or the number of iterations of certain loops. This statement also holds for the considered problem, although to the lesser extent. In Fig. 1 we can see that even in the case of linearly dependent fitness functions — the execution time and the number of graph vertices (repeatedly) visited during the algorithm — optimization by the number of vertices gives better results.

In the general case, the choice of a good fitness function from the given program is a non-trivial task, so a problem of dynamic selection of a good fitness function arises [7]–[9]. In the case of maximum flow algorithms, however, we can restrict the possible fitness functions to several well-known performance measures, such as: number of visited vertices, number of visited edges, number of DFS/BFS calls, etc. In addition, many algorithms belonging to the same algorithm family (e.g. augmenting path algorithms) share the set of possible fitness functions. The main hypothesis in this case is that the fitness function which was proven good for one of such algorithms should be good for the other algorithms of this family as well.

Due to the preliminary experiment results, the fitness function to be used in the paper is the number of DFS/BFS calls.

III. Genetic Algorithm Details

In this section, the implementation details of the genetic algorithms used in this paper are described.

The generation size is set to $G = 100$. The first generation is filled with randomly generated individuals. To create the next generation, first a *selection operator* is used to select $0.9G$ individuals for reproduction, grouped in pairs. Then the *crossover operator* is applied to each of the pairs of individuals. The *mutation operator* is then applied to each of the offsprings. After that, a new generation is formed by joining the mutated offsprings and the 10% best individuals from the old generation (a 10% elitist selection).

In the subsections, the individual encoding, the selection, crossover and mutation operators are described in more detail.

A. Individual Encoding

For the maximum flow algorithms, the test data are the networks — the graphs with capacities on each edge and with source and target vertices selected. To estimate the performance of different algorithms on the limited subset of all possible networks, we consider the networks with the maximum of V vertices, E edges, and maximum edge capacity of C.

To encode the networks, we define the vertex 0 to be the source, the vertex $(V-1)$ to be the target, and we use the edge list encoding to encode the graph: the individual is a list of E triples (s,t,c) where each triple defines an edge from vertex s to vertex t with the capacity of c.

To generate a new random edge (s,t,c), we generate s and t at random from an interval of $[0;V)$ such as $s \neq t$, and c at random from an interval of $[1;C]$. To generate a new individual, we generate E random edges and concatenate then in a list.

B. Mutation and Crossover Operators

To mutate an individual, we replace each edge with a randomly generated one with a probability of 1%.

We used three crossover operators:

1) the single-point crossover (SPC);
2) the two-point crossover (TPC);
3) the two-point crossover with shift (TPCS).

The *single-point crossover* works on two parent individuals as follows. First, a crossover index I is chosen from an interval of $[1;E)$. Second, the first individual A is split into two subsequences $A_1 = A[0 \dots I)$ and $A_2 = A[I \dots E)$, and the second individual B is split in exactly the same way into B_1 and B_2. Last, new offsprings are formed by concatenating $A_1 + B_2$ and $B_1 + A_2$, respectively.

The *two-point crossover* works on two parent individuals as follows. First, crossover indices I_1, I_2 are chosen from an interval of $[1;E)$ such that $I_1 < I_2$. Second, the first individual A is split into three subsequences $A_1 = A[0 \dots I_1)$, $A_2 = A[I_1 \dots I_2)$, $A_3 = A[I_2 \dots E)$, and the second individual B is split in exactly the same way into B_1, B_2, and B_3. Last, new offsprings are formed by concatenating $A_1 + B_2 + A_3$ and $B_1 + A_2 + B_3$, respectively.

The *two-point crossover with shift* is similar to the usual two-point crossover. But instead, the crossover indices are chosen for each parent separately ($I_1^A < I_2^A$, $I_1^B < I_2^B$) with a constraint that $I_2^A - I_1^A = I_2^B - I_1^B$. So, $A_1 = A[0 \dots I_1^A]$, $A_2 = A[I_1^A \dots I_2^A)$, $A_3 = A[I_2^A \dots E)$ and the same for B.

C. Selection Operators

We used three selection operators:

1) the roulette wheel selector (RWS);
2) the scaled roulette wheel selector (SRWS);
3) the tournament selector with the tournament size of 2 (TS).

The *roulette wheel selector* selects an individual from the generation with a probability proportional to its fitness value.

The *scaled roulette wheel selector* computes the minimum fitness in the generation F_{min}, the maximum fitness F_{max}, and computes the scaled fitness value for each individual:

$$F_S(i) = \frac{F(i) - F_{min}}{F_{max} - F_{min}}.$$

After that, the individual i is selected with a probability proportional to the $F_S(i)$. This variation makes the selection operator depend less on the magnitude of the fitness value and on the difference between fitness values in a generation.

The *tournament selector*, with the tournament size of 2, works the following way. To select an individual, it first picks two random individuals and then selects the individual with the better fitness value.

IV. Experiments

In the experiments, the following algorithms are considered:

- Edmonds-Karp algorithm [2];
- Dinic algorithm [3];
- Capacity scaling algorithm [4] with two different maximum scales (2000 and 20000).

We choose the maximum vertex number V to be 100, the maximum edge number E to be 2401, the maximum capacity C to be 10000. The value of E is chosen such as the maximum number of DIMACS generators can be used in the comparison.

For every configuration, the experiment was run for 10 times and then the results were averaged.

A. Best Crossover Determination

In Fig. 2, the results of running the genetic algorithm with roulette wheel selector and different crossovers are given. The number of BFS runs was selected as the fitness function. One may clearly see that the two-point crossover with shift performs much better. The possible explanation to this fact

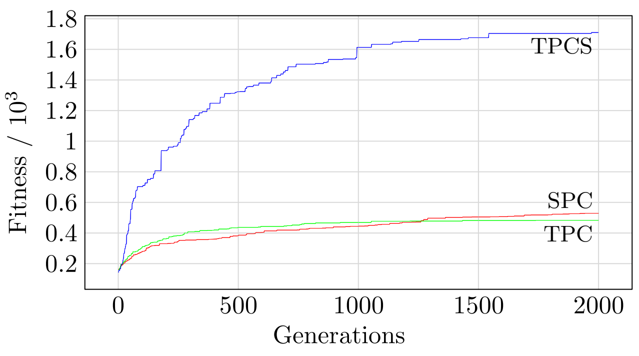

Fig. 2. Performance of different crossovers. Abbreviations are from Section III-B

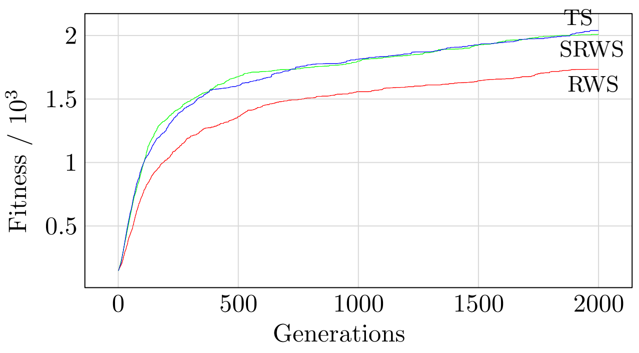

Fig. 3. Performance of selectors vs the Edmonds-Karp algorithm. Abbreviations are from Section III-C

is that there is no preliminary degeneracy in each of gene positions when shifts are allowed. In all further experiments, the crossover with shift is used.

B. Selection Operators vs Edmonds-Karp

In this experiment, tests against the Edmonds-Karp algorithm were generation with different selection operators. The number of BFS runs was selected as the fitness function. In Fig. 3, the performance for each selection operators is shown. It can be seen that the roulette wheel selector performs the worst, while tournament selector is only slightly better at the end (2040) than the scaled roulette wheel selector (2010).

C. Edmonds-Karp Runs vs Other Algorithms

In this experiment, for each of the selector operators, the tests generated during the 10 optimization runs against the Edmonds-Karp algorithm were evaluated against the other maximum flow algorithms. In Fig. 4, the results for the roulette wheel selector are given. In Fig. 5, the same is given for the scaled roulette wheel selector, and for tournament selector in Fig. 6.

It can be seen that, while the fitness for the Edmonds-Karp algorithm is growing, the fitnesses for the other algorithms are also growing. In other words, the difficult test data evolved against the Edmonds-Karp algorithm is difficult for all the considered algorithms simultaneously.

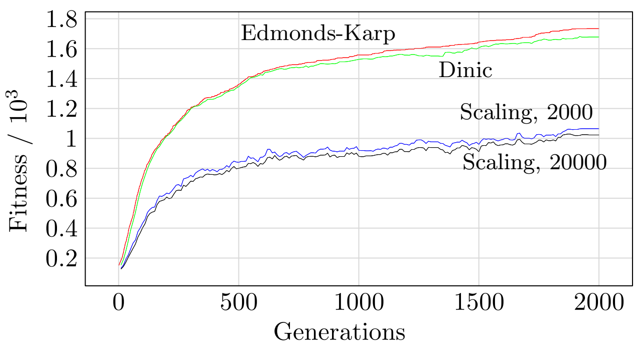

Fig. 4. Roulette Wheel Selector vs the Algorithms

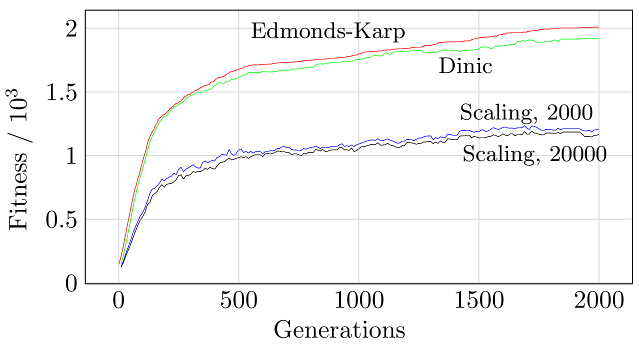

Fig. 5. Scaled Roulette Wheel Selector vs the Algorithms

We can also see that for all the algorithms the roulette wheel selector performs worse than all others, and the performance of the tournament selector and the scaled roulette wheel selector is very similar. So the effect first seen on the Edmonds-Karp algorithm is in fact very robust.

V. Comparison with DIMACS

For further research of effectiveness, the achieved values were compared with the performance on tests that were generated using different generators from DIMACS open source library [5]:

- *ak*, the Cherkassky and Goldberg generator.
- *tg*, the transit grid generator by Waissi.

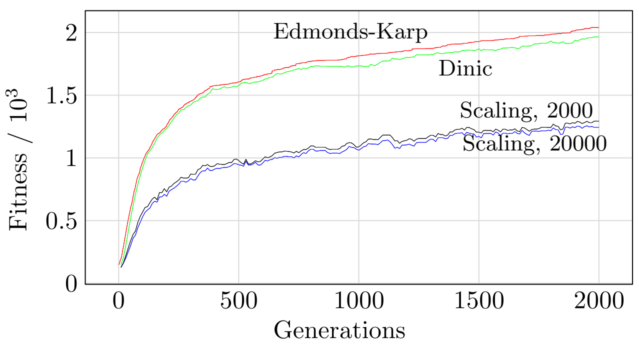

Fig. 6. Tournament Selector vs the Algorithms

TABLE I
COMPARISON OF EVOLVED TESTS WITH THE DIMACS TESTS

Method	Edmonds	Dinic	Scaling, 2000	Scaling, 20000
random	15	10	20	15
tg	15	54	13	14
genrmf	250	950	13	12
wash1	15	2650	20	15
ak	14	2979	10	9
Zadeh	4419	2210	108	107
Genetic	2040	1965	1244	1243

- *wash1*, the difficult case generator for the Dinic algorithm.
- *random*, the generator of random grids.
- *zadeh*, the test generator from the article [10] against the Edmonds-Karp algorithm.

The results are presented in Table I. In this table, the average for the 10 genetically evolved tests with the use of tournament selection is presented along with the DIMACS tests.

One can see that, for the Edmonds-Karp algorithm, the evolved tests are, in average, two times worse than the state-of-art test of Zadeh. However, the same tests are much more efficient relatively for the Dinic algorithm, and are way far superior for the capacity scaling algorithms.

VI. DISCUSSION

From the results presented in the figures and the table above we can conclude that for each standard maximum flow algorithm (such as Edmonds-Karp or Dinic), where the specialized test data generators exist, these generators work better than the genetic algorithm. However, the genetic performance is of the same order of magnitude. What is more, for the scaling algorithms, which seem to be more efficient, the genetic algorithm generated much harder tests. As a result, we can say that, for at least the maximum flow problem, the genetic approach seem to be the competitive approach that can produce good results with much less human interference.

VII. CONCLUSION AND FUTURE WORK

The genetic algorithm based approach to hard test generation for the maximum flow algorithms is presented in the paper. This approach is shown to be quite robust, i.e. the tests it generates are quite hard for the whole family of the algorithms. Compared to the DIMACS test data set, the generated tests appeared to be of the same order of difficulty for the algorithms where specialized test cases exist, and one to two orders of magnitude more difficult for other algorithms.

The authors see a large future work to be done. First, the analysis of the structure of the tests evolved during the evolutionary search is still missing. Second, generation of tests against other maximum flow algorithms is definitely needed, including other maximum flow algorithm families, push-relabel being the first. Third, some research is needed for the best fitness function, in terms of number of generations to achieve the same results, as well as some explanation of the observed behavior. Fourth, some theoretical work on the complexity of the test generation problem against maximum flow algorithms is very welcome.

REFERENCES

[1] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[2] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, no. 2, pp. 248–262, 1972.

[3] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Dokl.*, vol. 11, no. 5, pp. 1277–1280, 1970.

[4] R. K. Ahuja and J. B. Orlin, "A capacity scaling algorithm for the constrained maximum flow problem," *Networks*, vol. 25, no. 2, pp. 89–98, 1995.

[5] Dimacs. test generators for the maximum flow problem. [Online]. Available: http://www.informatik.uni-trier.de/˜naeher/Professur/research/generators/maxflow/

[6] M. Buzdalov, "Generation of tests for programming challenge tasks using evolution algorithms," in *Proceedings of the 2011 GECCO Conference Companion on Genetic and Evolutionary Computation*, 2011, pp. 763–766.

[7] A. Buzdalova, M. Buzdalov, and I. Petrova, "Generation of tests for programming challenge tasks using multi-objective optimization," in *Proceedings of Genetic and Evolutionary Computation Conference*, vol. 2, 2013, pp. 1655–1658.

[8] A. Buzdalova and M. Buzdalov, "Adaptive selection of helper-objectives for test case generation," in *Proceedings of Congress on Evolutionary Computation*, 2013, pp. 2245–2250.

[9] A. Buzdalova, M. Buzdalov, and V. Parfenov, "Generation of tests for programming challenge tasks using helper-objectives," *Lecture Notes in Computer Science*, vol. 8084, pp. 300–305, 2013.

[10] N. Zadeh, "Theoretical efficiency of the edmonds-karp algorithm for computing maximal flows," *Journal of the ACM*, vol. 19, no. 1, pp. 184–192, 1972.