Hard Test Generation for Augmenting Path Maximum Flow Algorithms using Genetic Algorithms: Revisited

Maxim Buzdalov, Anatoly Shalyto
ITMO University
49 Kronverkskiy av.
Saint-Petersburg, Russia, 197101
Email: mbuzdalov@gmail.com, shalyto@mail.ifmo.ru

***Abstract*—To estimate performance of computer science algorithms reliably, one has to create worst-case execution time tests. For certain algorithms this task can be difficult. To reduce the amount of human effort, authors attempt using search-based optimization techniques, such as genetic algorithms.**

Our previous paper addressed test generation for several maximum flow algorithms. Genetic algorithms were applied for test generation and showed promising results. However, one of the aspects of maximum flow algorithm implementation was missing in that paper: parallel edges (edges which share source and target vertices) were not merged into one single edge (which is allowed in solving maximum flow problems).

In this paper, parallel edge merging is implemented and new results are reported. A surprising fact is shown that fitness functions and choices of genetic operators which were the most efficient in the previous paper are much less efficient in the new setup and vice versa. What is more, the set of maximum flow algorithms, for which significantly better tests are generated, changed completely as well.

I. Introduction

Finding inputs which make a certain algorithm or program work for the longest possible period of time, or the worst-case execution time test generation, is an important concern. Some problems in developing reliable software, such as finding the maximum response time, reduce to this problem.

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 techniques, like in search-based software engineering [1], [2]. 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 its solving are known, as well as several good test generation algorithms exist. One of the properties of this problem is that performance of most algorithms on randomly generated data is much faster than one can expect from its running time estimation. For example, the Edmonds-Karp algorithm [3] which has the running time compexity of $O(V \cdot E^2)$, has no trouble in terminating under a second on a randomly generated graph with $V = 500$ and $E = 10\,000$ when run on commodity hardware, although, based on the upper bound, one can estimate an order of 10^{11} elementary operations to be done. This makes the maximum flow problem a good benchmark for worst-case execution time test generators.

We further refine our domain to augmenting path algorithms [3], [4], including their capacity scaled versions [5]. Tests generated by the proposed approach are compared to tests constructed by DIMACS data generators [6] and some other known test generators [7], [8].

In our previous paper [9], we have already made an attempt to generate worst-case execution time tests for maximum flow algorithms using genetic algorithms. However, all maximum flow algorithms implemented in that paper have a common drawback. In maximum flow problems, if there are two "parallel" edges which start from a vertex V_1 and end in the vertex V_2 and have corresponding capacities of C_1 and C_2, these edges can be replaced by a single edge with a capacity of $C_1 + C_2$. Every solution of the modified problem can be mapped back to at least one solution of the initial problem, and every solution of the initial problem can be transformed to a solution of the modified problem. Both transformations preserve the main quality measure, the total flow. To decrease the total number of edges, one can simply merge all parallel edges. This is commonly done in real maximum flow algorithm implementations, but it was not done in [9].

In particular, one can "multiply" any flow network by an integer $K \geq 2$ by making $K - 1$ additional copies of each edge. If parallel edge merging is not performed, most if not all maximum flow algorithms belonging to augmenting path family perform K times worse: they have to repeat each of their actions K times. In contrast, if merging is performed, most algorithms just multiply the amount of flow for each augmenting path by K, which doesn't influence their running times. This motivated us to correct the considered maximum

978-1-4799-7492-4/15/$31.00

flow algorithms and reconsider the research.

II. Maximum Flow Problem

The problem of finding a maximum flow is a classic problem in graph theory [10]. It is formulated as follows:

- the input is an oriented graph with V vertices and E edges;
- there are two distinct vertices called *the source* s and *the sink* t;
- each edge has a unique number i ($1 \leq i \leq E$), and for each edge with a number i a *capacity* $c_i \geq 0$ is given;
- one needs to find a *maximum flow* – a set of numbers f_i ($1 \leq i \leq E$) such that:
  - for each edge, $0 \leq f_i \leq c_i$;
  - for each vertex except for s and t, the sum of fs for the incoming edges is equal to the sum of fs for the outgoing edges;
  - for s, the sum of fs for the outgoing edges minus the sum of fs for the incoming edges is maximum possible.

III. Algorithms for Finding Maximum Flows

There are many known algorithms for solving the maximum flow problem. Some of the best known algorithms are [10]:

- the Ford-Fulkerson algorithm [11], the running time is $O(V \cdot E \cdot C_{\max})$, where $C_{\max}$ is the maximum capacity of an edge in the graph;
- the Edmonds-Karp algorithm [3], the running time is $O(V \cdot E^2)$;
- the Dinic algorithm [4], the running time is $O(V^2 \cdot E)$ and $O(E \cdot \min(E^{1/2}, V^{2/3}))$ for unit capacities;
- the improved shortest path algorithm [12], the running time is $O(V^2 \cdot E)$;
- the push-relabel algorithm [13], the running time is $O(V^2 \cdot E)$;
- the push-relabel algorithm with the relabel-to-front rule, the running time is $O(V^3)$.

These algorithms are often modified using the *capacity scaling* approach [5], [12].

We limit ourselves to algorithms from augmenting path family, as they all have a similar structure and similar performance measures. We consider the following algorithms:

- the Ford-Fulkerson algorithm with capacity scaling;
- the Edmonds-Karp algorithm;
- the Edmonds-Karp algorithm with capacity scaling;
- the Dinic algorithm;
- an unoptimal implementation of the Dinic algorithm which does not delete zero-capacity edges from shortest path networks, and thus has a worse runtime complexity of $O(V \cdot E^2)$;
- the improved shortest path algorithm.

IV. Maximum Flow Test Generators

Finding maximum flow is a well-studied problem, so the problem of finding a hard test for a particular maximum flow algorithm was studied for a long time as well. In this section several test generation algorithms known from the literature [6]–[8] are outlined.

An outcome of a test generator would be a graph with a source, a sink and capacities assigned to edges. In the rest of the paper, we use terms "test" and "graph" interchangeably. Most often, the input parameters to a test generator are V, the maximum number of vertices in a graph, E, the maximum number of edges, and C, the maximum capacity of an edge in a graph. Only integer capacities are considered.

In this work we compare our approach with the following test generators:

- random graph generation (including graphs without parallel edges);
- random acyclic graph generation [6];
- transit grids [6];
- random frames [6], [7];
- Cherkassky and Goldberg generator [6];
- "washington" generators [6];
- Zadeh tests [8].

The detailed descriptions of these generators are available in Appendix.

V. Genetic Algorithm

In this section, the genetic algorithm, which was used to generate tests for maximum flow algorithms, is described. It basically shares ideas with the algorithm described in [9], however, more genetic operators are explored, and fitness functions are treated differently.

We assume that the maximum number of vertices V, the maximum number of edges E and the maximum capacity of an edge C are given. The *individual* is basically a list of edges. Each edge is denoted by its source vertex s_i, target vertex t_i and capacity c_i. The source vertex of the graph is the vertex with the index of 1 and the target vertex is the vertex with the index of V.

Based on an idea that acyclic graphs may be harder [6], it was decided to support acyclic graph generation along with arbitrary graphs. More precisely, in arbitrary graph generation no restrictions are put on the graph edges, either initially generated or introduced by genetic operators. In acyclic graph generation, for each edge an index of the source vertex must be strictly less than an index of the target vertex. Both options (*arbitrary* and *acyclic*) are explored in the experimental evaluation.

A. Genetic Algorithm Scheme

A standard genetic algorithm scheme is used. A *population size* G is fixed to be 100. A single iteration of the genetic algorithm is performed as follows:

1) The *reproduction selection* operator is used to select G individuals from the population. These individuals are then used to create offspring.
2) The selected individuals are grouped in pairs and the *crossover* operator is applied to each pair. For each

pair, the crossover operator generates a new pair of individuals.
3) The *mutation* operator is applied to each individual which underwent crossover.
4) Each mutated individual is evaluated using the *fitness function*.
5) The new population is formed from the old population and newly generated individuals using the *survival selection* operator.

B. Initial Population Creation

The initial population is created by generating G random individuals. A random individual is generated from a list of E randomly generated edges. When *arbitrary* option is used, each edge is generated by selecting source and target vertices uniformly at random from the range $[1; V]$ and capacity is selected uniformly at random from the range $[1; C]$. When *acyclic* option is used, the source-target pair of vertices is selected uniformly at random from all pairs (s, t) such that $1 \leq s < t \leq V$.

C. Reproduction Selection Operator

The reproduction selection operator is a variant of tournament selection which appeared to be efficient in a number of our previous works [14], [15] for different problems. To select a single individual, eight individuals are selected from the population uniformly at random with replacement. They are grouped in pairs. From each pair, an individual with better value of fitness function is selected with the probability of 0.9, and with worse value of fitness function otherwise. The selected individuals are grouped in pairs again and the process repeats until only one individual is left.

This operator is basically an "olympic system" tournament and imposes higher selection pressure than the most commonly used tournament selection operator.

D. Crossover Operator

Two different crossover operators are used. The first one is a standard single-point crossover operator. The second one is a two-point crossover operator with shift, which was used in our previous maximum flow test generation paper [9]. Both options (single-point and two-point with shift) are explored in experimental evaluation.

E. Mutation Operator

A mutation operator is used that is a standard one for evolution strategies. Each edge is replaced with the probability of $1/E$ by a randomly generated edge. The edge is generated as in initial population creation, which depends on the selected genetic algorithm option (*arbitrary* or *acyclic*).

F. Survival Selection Operator

A 10% elitism operator is used as survival selection: 10% of the best individuals from the old population are promoted to the new population, the rest of the new population is formed by the best of the newly generated individuals.

G. Fitness Functions

When an algorithm's running time should be maximized, a natural fitness function is the running time itself. In multitasking operating systems there are at least two different ways to measure time:

- Wall-clock time. This is equal to astronomic time which passed from the moment the algorithm starts running to the moment the algorithm finishes its work. This quantity can be measured with a good precision (if measured from the process which is used to run the algorithm itself), but it is very noisy due to the presence of other tasks running on the same computer.
- Processor time. This is the amount of time allocated by the operating system to the execution thread which was used to run the algorithm. For single-threaded algorithms this could be a preferred way to measure running time, but due to operating system's limitation this value is reported as a multiple of a certain time interval which is quite big. For example, in most Linux kernels processor times are multiples of 10 milliseconds, while under Windows the time quant is close to 13 milliseconds. This essentially prevents this method of measurement for being used when running times are small.

In our previous works [14], [15] it was proposed to use more algorithm-specific performance measures which are noiseless and typically have high resolution. The considered maximum flow algorithms spend most of their time in graph traversals, so they have several common performance measures:

- The total number of visited edges (each edge is counted as many times as it was visited).
- The total number of visited vertices.
- The total number of graph traversals.

The latter number is equal to the number of depth-first searches for the Ford-Fulkerson algorithm with capacity scaling and for implementations of the Dinic algorithm; to the number of breadth-first searches for variations of the Edmonds-Karp algorithm; and the the number of *retreats* for the improved shortest path algorithm. Additionally, for implementations of the Dinic algorithm the number of phases can be tracked.

We perform evaluation of all algorithm-dependent fitness functions inside the corresponding algorithms: for example, a counter which stores the number of visited edges is immediately increased when an edge is visited. As every event which is counted takes more time than incrementing a single integer variable, evaluation of fitness functions has no significant impact on performance of algorithms. Wall-clock time and processor time are computed as a difference between two measures – before the algorithm starts working and after it finishes.

It is very hard to determine the best fitness function *a priori*. In the experimental evaluation, each of these fitness functions was used to generate tests for the appropriate maximum flow algorithm.

VI. Experimental Evaluation

In this section, experimental evaluation of test generation methods for maximum flow algorithms is described.

A. Experiment Setup

The maximum number of vertices was chosen to be $V = 100$, the maximum number of edges was $E = 5000$ to accomodate methods which generate dense graphs only, and the maximum edge capacity was $C = 10\,000$.

For the genetic algorithm, the following options were explored, resulting in 128 different configurations:

- the maximum flow algorithm against which tests are generated (six choices);
- the fitness function (six choices for two implementations of the Dinic algorithm, five choices for all other algorithms);
- the crossover operators (two choices: single-point crossover and two-point crossover with shift);
- the graph type (two choices: arbitrary or acyclic).

From the known test generators, the "washington" generators 9 and 10, the Cherkassky and Goldberg generator and tests by N. Zadeh don't use random number generators. Other generators, however, depend on random number generators, so they have to be put into equal conditions with the genetic algorithm. Thus, a configuration was formed for each combination of a test generator, a maximum flow algorithm and a fitness function. There were six test generators of this type: random graphs, random acyclic graphs, random graphs without parallel edges, random graphs without edges connecting same pairs of vertices, random frames and transit grids. In total, there were 192 configurations of this sort.

Each of configurations, corresponding both for genetic algorithms and random-dependent test generators, was run for 25 times. The computational budget was set to 500 000 fitness function evaluations. The best test according to the fitness function was taken as a result of the run. In total, there were 8004 tests generated, corresponding to four tests for non-random generators and $(192 + 128) \cdot 25$ tests for random-dependent test generators.

The experiments were run on a computer with four *AMD Opteron™* 6378 processors, each of which has 16 cores. The operation system was Ubuntu 14.04.1 LTS with 64-bit Linux kernel 3.13.0-39-generic. Task-grained concurrency was used. The computation took almost a week to finish.

After test generation was finished, every maximum flow algorithm was run on every generated test.

B. Choice of Performance Measure

The experiment results showed that reported processor times are 0, 10 or 20 milliseconds most of time, so these values are largely unusable. Wall-clock times are more diverse but are still prone to context switches, Java garbage collection and similar things. To reliably compare different tests, one needs to select a non-noisy measure that has enough range to distinguish hard tests from easy ones. It was done by selecting a fitness function based which correlates best with wall-clock time measurements.

TABLE I

Pearson's correlation quotients between wall-clock time and other fitness functions. Notation: D – the Dinic algorithm, DS – an unoptimal implementation of the Dinic algorithm, EC – the Edmonds-Karp algorithm, ECS – the Edmonds-Karp algorithm with capacity scaling, FFS – the Ford-Fulkerson algorithm with capacity scaling, ISP – the improved shortest path algorithm; procTime – processor time, edgeCount – total number of visited edges, vertexCount – total number of visited vertices, dfsCount – total number of depth-first search runs, bfsCount – total number of breadth-first search runs, retreatCount – total number of retreats (for ISP), phaseCount — total number of phases (for the Dinic algorithm).

	D	DS	EC	ECS	FFS	ISP
procTime	0.21297	0.23298	0.54402	0.47434	0.89681	0.18645
edgeCount	0.94119	0.96107	0.97425	0.97688	0.98778	0.91901
vertexCount	0.84388	0.86782	0.87892	0.86379	0.91403	0.70379
dfsCount	0.71523	0.76746	—	—	0.90319	—
bfsCount	—	—	0.86080	0.81960	—	—
retreatCount	—	—	—	—	—	0.72674
phaseCount	0.74970	0.72177	—	—	—	—

TABLE II

Best tests for different maximum flow algorithms

Algorithm	Generator	Edge count
D	GA(*edgeCount*, *DS*, *SPC*, *ac*)	594 662
DS	GA(*edgeCount*, *DS*, *SPC*, *ac*)	671 867
EC	GA(*edgeCount*, *EC*, *SPC*, *ac*)	4 613 284
ECS	Zadeh	4 524 126
FFS	random acyclic(*edgeCount*, *FFS*)	4 691 777
ISP	GA(*edgeCount*, *ISP*, *SPC*, *ac*)	820 538

Table I presents Pearson's correlation coefficients between wall-clock time and all other fitness functions. For every maximum flow algorithm, the total number of visited edges (*edgeCount* in the table) served best correlation coefficients. This measure will be used in the remaining parts of the paper as a measure of test quality.

C. Best Tests

For the sake of brevity, in the rest of the paper we define configurations of the genetic algorithm as GA(*fitness*, *algorithm*, *crossover*, *graph type*). Here, *fitness* is from the set {*wcTime*, *procTime*, *edgeCount*, *vertexCount*, *dfsCount*, *bfsCount*, *retreatCount*, *phaseCount*} (the notation is as in Table I, additionally, *wcTime* is for wall-clock time). For *algorithm*, the possible values are {*D*, *DS*, *EC*, *ECS*, *FFS*, *ISP*} (the notation is as in Table I). The values for *crossover* are {*SPC*, *TPCS*} for single-point crossover and two-point crossover with shifts, respectively. Finally, the values of *graph type* are {*any*, *ac*} for arbitrary and acyclic graphs respectively. The similar notation will be used for other generators as well.

Table II represent best tests for different maximum flow algorithms. One can see that for all considered algorithms which don't use capacity scaling the best tests are generated by genetic algorithms. One notable thing is that all winning configurations of the genetic algorithm use single-point crossover and acyclic graph generation.

TABLE III
BEST TEST GENERATORS FOR DIFFERENT MAXIMUM FLOW ALGORITHMS

Algorithm	Generator	Edge count
D	GA(*edgeCount*, *DS*, *SPC*, *ac*)	476 035
DS	GA(*edgeCount*, *DS*, *SPC*, *ac*)	584 427
EC	Zadeh	4 609 566
ECS	Zadeh	4 524 126
FFS	random acyclic(*edgeCount*, *FFS*)	4 511 830
ISP	GA(*edgeCount*, *ISP*, *SPC*, *ac*)	591 438

TABLE IV
RESULTS OF OPTIMIZATION BY TIME

Algorithm	Wall-clock time	Processor time	Best test
D	235 601	189 839	594 662
DS	280 710	210 524	671 867
EC	2 086 433	1 967 031	4 613 284
ECS	1 744 677	1 787 049	4 524 126
FFS	4 108 867	3 971 258	4 691 777
ISP	352 309	262 657	820 538

D. Best Test Generators

To measure efficiencies of test generators for particular algorithms, for each generator we compute the median of numbers of visited edges for all tests generated by this generator. The best generators for different maximum flow algorithms are presented in Table III.

The only change in winners happened for the Edmonds-Karp algorithm. One can see that the best test (edge count 4 613 284) is sligtly better than Zadeh's test (edge count 4 609 566). An unusual thing is that, both for the best and the median test quality measures, the hardest tests for the Dinic algorithm were generated against its unoptimal implementation.

E. Average Test Generator Efficiencies

To measure efficiencies of test generators averaged over considered maximum flow algorithms, we consider the following approach. For every maximum flow algorithm, all tests are sorted by non-increasing number of visited edges. Then, ranks (the indices in the sorted sequence, from 1 to 8004) are assigned to tests. If several tests have the same numbers of visited edges, they receive an equal rank equal to an average of indices of all these tests. For each test, all ranks which this test received are averaged. After that, each test generator receives a rank which is an average of ranks of all tests generated by this generator.

The best ten generators, according to ranks, are given below (the smaller the rank, the better the efficiency):

1) 247.9: GA(*edgeCount*, *DS*, *SPC*, *ac*).
2) 283.4: GA(*edgeCount*, *EC*, *SPC*, *ac*).
3) 347.8: GA(*edgeCount*, *ISP*, *SPC*, *ac*).
4) 405.8: GA(*edgeCount*, *D*, *SPC*, *ac*).
5) 640.1: random acyclic(*edgeCount*, *ISP*).
6) 648.7: random acyclic(*edgeCount*, *DS*).
7) 692.6: GA(*edgeCount*, *ISP*, *SPC*, *any*).
8) 716.3: random acyclic(*phaseCount*, *DS*).
9) 718.4: random acyclic(*edgeCount*, *D*).
10) 729.2: random acyclic(*vertexCount*, *D*).

One can see that there is a large gap between the first four generators and all subsequent ones. There is only a single generator in top 10 which has no heuristic knowledge about the power of acyclic graphs. Among all generators based on genetic algorithms, no generator from top 10 uses the two-point crossover with shift.

The non-random test generators have the following ranks:

- Zadeh: 1 987.3;
- "washington" No. 9: 6 237.2;
- Cherkassky and Goldberg: 6 975.3;
- "washington" No. 10: 7 003.8.

Best non-acyclic random graph generators have the following ranks:

- without parallel edges: 2 239.4;
- without edges connecting the same pair of vertices: 2 403.6;
- arbitrary: 4 280.7.

F. Best Genetic vs Best Non-Genetic

For the Dinic algorithm, the best test generated *not* by the genetic algorithm has the number of visited edges equal to 180 589, which is more than three times smaller than the value of the best test (594 662). The similar picture holds for the improved shortest path algorithm: 441 289 and 820 538, respectively.

For the unoptimal implementation of the Dinic algorithm, the best test has the number of visited edges equal to 671 867, the Zadeh test has the value of 558 337, and the best test not belonging to these classes has the value of 201 038. Note that the Zadeh test is hard only for the unoptimal implementation, but not for the complete one.

For the Edmonds-Karp algorithm the best test is generated by the genetic algorithm (4 613 284), the second one is the Zadeh test (4 609 566), and the best test not belonging to these two classes is weaker at least twice (2 269 179).

The situation is different for capacity scaling algorithms. For the Edmonds-Karp algorithm with capacity scaling, the best test is the Zadeh test (4 524 126), and the next best test is almost twice as weak (2 302 245). For the Ford-Fulkerson algorithm with capacity scaling the best tests are random acyclic graphs (the best test has the number of visited edges equal to 4 691 777), however, the genetic algorithm is not very much worse (the best test is 4 276 511). One possible explanation to this kind of behavior is that capacity scaling algorithms induce very bad fitness landscapes when the genetic operators considered in this paper are used.

G. Optimization by Time is Inefficient

Table IV presents the best numbers of visited edges for optimization by wall-clock time and by processor time. These results show that optimization by time is highly inefficient in all cases (except for the Ford-Fulkerson algorithm with capacity scaling, where this statement holds to much smaller degree).

VII. Conclusion

In our previous paper dedicated to worst-case execution time test generation for maximum flow algorithms [9], merging parallel edges was not performed in our implementations of the maximum flow algorithms. This paper corrects this issue and reconsiders the research. More crossover operators, more fitness functions and more maximum flow algorithms were considered in this paper. The results appear to be very different from the results of [9]:

- genetic algorithm based approach is shown to be efficient for algorithms that don't use capacity scaling, while in [9] the best performance was shown for capacity scaling algorithms;
- the single-point crossover is shown to be significantly better than two-point crossover with shift, while in [9] the latter was several orders better.
- the best-performing fitness function, in terms of quality of the final results, is the number of visited edges, while in [9] the number of graph traversals was the best.

These differences can be explained using the idea mentioned in the introduction. To construct a hard test in conditions of [9], one can find a relatively hard test with a small number of edges and copy it over itself as much times as possible. In particular, the two-point crossover with shift is especially good at copying and pasting parts of individuals. In more realistic conditions of this paper, this operation doesn't increase hardness of a test.

From the results of this work it follows that there is a single configuration of the genetic algorithm that is the best for all considered maximum flow algorithms: optimizing the number of visited edges using the single-point crossover and acyclic graphs. This reduces the effort needed to generate hard tests with different upper limits and chooses a good default setting for further research.

VIII. Future Work

As a future work, apart from the obvious directions of taking more maximum flow algorithms into account, construction of test *generators* rather than tests should be undertaken. The biggest problem of the current approach is that we cannot learn how to build large hard tests from successfully building small hard tests. In other words, to build a test for another combination of limits, e.g. $V = 500$, $E = 20\,000$, one should run the genetic algorithm from scratch (even if a good configuration is known). We think that evolving parameterized programs which construct tests for maximum flow algorithms should yield results that scale better.

IX. Links and Acknowledgments

The code which can be used to reproduce the experiments is published at GitHub[1].

This work was financially supported by the Government of Russian Federation, Grant 074-U01.

[1]https://github.com/mbuzdalov/papers/tree/master/2015-cec-flows

References

[1] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends, technologies and applications," Department of Computer Science, King's College London, Tech. Rep., 2009.

[2] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The GISMOE challenge: constructing the Pareto program surface using genetic programming to find better programs (keynote paper)," in *International Conference on Automated Software Engineering*, 2012, pp. 1–14.

[3] 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.

[4] 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.

[5] 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.

[6] DIMACS. Test generators for the maximum flow problem. [Online]. Available: http://www.informatik.uni-trier.de/~naeher/Professur/research/generators/maxflow/

[7] D. Goldfarb and M. D. Grigoriadis, "A computational comparison of the Dinic and network simplex methods for maximum flow," *Annals of Operations Research*, vol. 13, no. 1, pp. 81–123, 1988.

[8] 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.

[9] V. Arkhipov, M. Buzdalov, and A. Shalyto, "Worst-Case Execution Time Test Generation for Augmenting Path Maximum Flow Algorithms using Genetic Algorithms," in *Proceedings of the International Conference on Machine Learning and Applications*, vol. 2. IEEE Computer Society, 2013, pp. 108–111.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 2nd Ed.* Cambridge, Massachusetts: MIT Press, 2001.

[11] L. R. Ford Jr. and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[12] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[13] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1986, pp. 136–146.

[14] M. Buzdalov, "Generation of Tests for Programming Challenge Tasks Using Evolution Algorithms," in *Proceedings of Genetic and Evolutionary Computation Conference Companion*. ACM, 2011, pp. 763–766.

[15] A. Buzdalova, M. Buzdalov, and V. Parfenov, "Generation of Tests for Programming Challenge Tasks Using Helper-Objectives," in *5th International Symposium on Search-Based Software Engineering*, ser. Lecture Notes in Computer Science. Springer, 2013, no. 8084, pp. 300–305.

Appendix

A. Random Graphs

A trivial way to generate a graph with V vertices, E edges and a maximum capacity of C is to generate E edges with source and target vertices selected uniformly at random from the range of $[1; V]$, while (integer) capacity is selected uniformly at random from the range of $[1; C]$. The source and the sink, which are the only remaining things to be generated, can be set to the vertices with indices 1 and V correspondingly without loss of generality.

As noted in the introduction, all edges that have the same source vertex V_1 and the same target vertex V_2 can be merged into one edge. In a randomly generated graph, there will be several "parallel" edges with high probability. This makes it reasonable to consider also generation of graphs with no "parallel" edges (i.e. each time an edge is generated in such

a way that it doesn't connect the same pair of vertices in the same order as already generated edges). Generation of graphs such that no two edges connect the same pair of vertices regardless of their order will also be considered.

B. Random Acyclic Graphs

One of the heuristic ways to improve random graph generation is to ensure that there are no loops in the graph. To do that, one may fix the vertex 1 as the source, the vertex V as the target and generate only the edges which go from a smaller vertex to a larger one (i.e. from a vertex with index V_1 to a vertex with index V_2 where $V_1 < V_2$).

A generator of such tests is presented at the website [6] under the identifier of `ac` by G. Waissi and J. Setubal. This generator builds complete acyclic graphs without loops and multiple edges: for V vertices it generates $V(V-1)/2$ edges. The program can generate two types of graphs: the descriptions of edges going out of a certain vertex can present in either increasing or decreasing order regarding the number of the target vertex. These types define the same graph, but due to implementation details of maximum flow algorithms only one of these types can be hard for some algorithms.

C. Transit Grids

A generator of *transit grids* is presented at the website [6] under the identifier of `tg` by G. Waissi and J. Setubal. This generator builds a rectangular grid of size $a \times b$ (where a and b are parameters). In nodes of this grid there are vertices of the graph. Any two adjacent vertices (vertically or horizontally) are connected by a pair of edges going in different directions with capacities chosen uniformly at random. The source and the sink do not belong to this grid; they are connected with pairs of edges going in different directions with left and right sides of the grid correspondingly.

Given the number of vertices V, the tests of this type will be generated as follows. First, the number a will be selected uniformly at random from an interval of $[1; V-2]$. Second, the number b will be chosen as $b = \lfloor V/a \rfloor$. Finally, a test with parameters of a and b is generated as described above.

D. Random Frames

A *random frame generator* is described by D. Goldfarb and M. Grigoriadis [7]. The source code of this generator is available at the website [6] under the identifier of `genrmf`.

This generator creates a three-dimensional structure of b frames, each of them has a size of $a \times a$ vertices, where a and b are parameters. The structure of each frame resembles the structure created by the transit grid generator. The capacities of all edges in a frame are equal to $c_2 \times a^2$, where c_2 is a parameter. The neighboring frames are connected by unidirectional edges: i-th vertex of the t-th frame is connected by $P_t(i)$-th vertex of the $(t+1)$-th frame, where P_t is a random permutation of numbers from 1 to a^2. Capacities of these edges are chosen from an interval of $[c_1; c_2]$, where c_1 is a parameter and $c_1 \le c_2$. The source of the graph is the topmost leftmost vertex of the first frame, the sink is the bottommost rightmost vertex of the last frame.

Given the maximum number of vertices V, the maximum number of edges E and the maximum capacity C, the tests of this type will be generated as follows. First, the parameter b will be chosen uniformly at random from an interval of $[1; \min(V, E)]$. Second, the parameter a will be chosen as a maximum integer value such that $a^2 b \le V$ and $3a^2 b - a(a+2b) \le E$. If $a < 1$, parameter selection will be started from scratch. Next, $c_2 = \lfloor C/a^2 \rfloor$ and $c_1 = 1$. If $c_2 < 1$, parameter selection will be started from scratch. Otherwise, the test will be generated as described above with the parameters a, b, c_1, c_2.

E. Cherkassky and Goldberg Generator

The generator by B. Cherkassky and A. Goldberg is available at the website [6] under the identifier of `ak`. This generator creates a test consisting of two different subgraphs. A notable difference from the other generators is that the number of edges is linear in the number of vertices, so the generated graphs are sparse.

F. Washington Generators

The "washington" generator which contains several test generators written by the students of the University of Washington under supervision of R. Anderson. It is available at the website [6] under the identifier of `wash`. There are 11 supported types of generators. In the context of this research, only two of them are of interest: generator 9 creates a test difficult for the Dinic algorithm, while generator 10 creates a test difficult for the Goldberg's algorithm.

Generator 9 builds a graph which is a chain of V vertices, the first of them is the source and the last one is the sink. The neighboring vertices are connected with an edge of capacity V. Additionally, each vertex except two last ones is connected to the sink with an edge of capacity 1. This test makes the Dinic algorithm perform the maximum possible number of phases.

Generator 10 builds a graph with N parallel chains of two vertices, which are then followed by a sequential chain of N vertices. The capacities of all edges is N except for edges which connects pairs of vertices in parallel chains (Fig. 1). The total number of vertices is $3N+3$, while the total number of edges is $4N+1$.

G. Zadeh Tests

N. Zadeh [8] proposed an algorithm to construct hard tests for the Edmonds-Karp algorithm. Tests of this type have $6N$ vertices from which $4N$ vertices form a complete bipartite graph and the remaining vertices form a chain from the source to the sink, which connects by edges to the vertices of the first group. An example of such test with 18 vertices ($N = 3$) is given on Fig. 2.

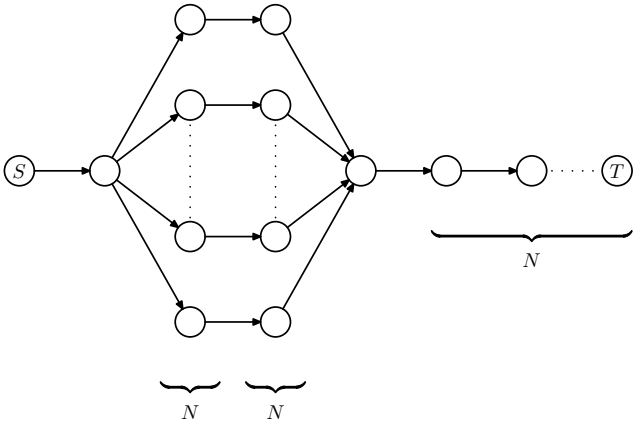

Fig. 1. An illustration of a test generated by the "washington" test generator 10. The graph consists of N parallel chains of two vertices, which are then followed by a sequential chain of N vertices. The capacities of all edges is N except for edges which connects pairs of vertices in parallel chains. The total number of vertices is $3N + 3$, while the total number of edges is $4N + 1$.

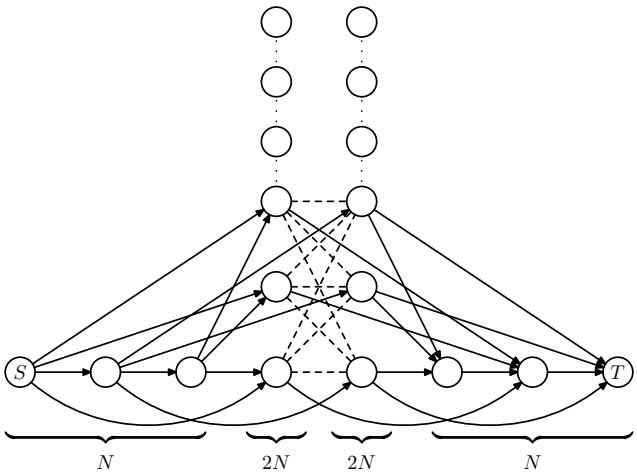

Fig. 2. An illustration of a Zadeh test based on the original illustration from [8]. The test consists of $6N$ vertices (on the figure, $N = 3$). The central group is formed by two columns of vertices, consisting of $2N$ vertices each. Each vertex of the left column is connected to each vertex of the right column by a pair of edges going in different directions (dashed connections on the figure). The left group of N vertices connects the source (the leftmost vertex) to the central group, the right group of N vertices connects the sink (the rightmost vertex) to the central group.