Hard Test Generation for Maximum Flow Algorithms with the Fast Crossover-Based Evolutionary Algorithm

Vladimir Mironovich
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia
mironovichvladimir@gmail.com

Maxim Buzdalov
ITMO University
49 Kronverkskiy ave.
Saint-Petersburg, Russia
mbuzdalov@gmail.com

ABSTRACT

Most evolutionary algorithms not only throw out insufficiently good solutions, but forget all information they obtained from their evaluation, which reduces their speed from the information theory point of view. An evolutionary algorithm which does not do that, the $(1 + (\lambda, \lambda))$ EA was recently proposed by Doerr, Doerr and Ebel.

We evaluate this algorithm on the problem of finding hard tests for maximum flow algorithms. Experiments show that the $(1 + (\lambda, \lambda))$ EA is never the best, but is quite stable. However, its adaptive version, known for being superior for the OneMax problem, is shown to be one of the worst.

CCS Concepts

•**Theory of computation** → **Evolutionary algorithms;** •**Software and its engineering** → Search-based software engineering;

Keywords

Test generation, worst-case execution time, black-box complexity, evolutionary algorithms

1. INTRODUCTION

Most evolutionary algorithms use only little information from evaluation of inferior solutions. The simplest of them, $(1 + 1)$ evolutionary algorithm, completely ignores solutions which are worse than the current best, which means that the closer it is to the optimum, the more information from fitness evaluation it loses. One of the old ways to overcome this problem is to increase generation size, which, however, is not a complete solution, as for smaller generation sizes the running time may be even exponential [11], and for larger sizes it is still worse than of the $(1 + 1)$ EA.

A recently proposed evolutionary algorithm called $(1 + (\lambda, \lambda))$ [6] solves this problem in another way. It generates λ offspring from the parent by mutation using a mutation probability that is k times larger than the usual one. Larger mutation probability allows for faster exploration of the search space, but the resulting offspring is often worse than the parent. The further offspring is generated by using uniform crossover operator on parent and the best individual produced via mutation. The crossover operator takes bits from the best individual with probability $1/k$. As this algorithm contains a phase which doesn't lose individuals which are worse than the best known individual, it uses more information on average from fitness evaluations. As a corollary, its running time on the OneMax problem is faster than the running time of $(\mu + \lambda)$ or (μ, λ) evolutionary algorithms. In particular, the *adaptive* version, which changes λ and k according to fitness values, achieves linear running time complexity on OneMax.

The authors of $(1 + (\lambda, \lambda))$ EA suggested to test this algorithm on more practical problems to see if their idea is robust enough. In this paper, we test it experimentally on the problem of hard test generation for the maximum flow algorithms, which attracted our attention earlier [3]. We compare the $(1 + (\lambda, \lambda))$ EA with the genetic algorithm from [3] as well as the $(1 + 1)$ EA for its simplicity and the $(1 + 2\lambda)$ EA for having the same number of evaluations per generation as the $(1+(\lambda, \lambda))$ EA. We consider several values of λ as well as the adaptive version of the $(1 + (\lambda, \lambda))$ EA. Results of the experiments suggest that the $(1 + (\lambda, \lambda))$ EA most often shows next-to-leading results in different conditions for some λ, but the adaptive version is typically bad, which suggests that the adaptation heuristic seems to be overfitted for OneMax and similar problems.

The rest of the paper is structured as follows. Section 2 explains the maximum flow problem and gives an overview for algorithms to solve it. Section 3 is dedicated to details of evolutionary algorithms, including the individual encoding, the mutation operator and the fitness function. Section 4 gives experimental setup and results. Section 5 concludes, gives some insights on the results and suggests some directions for the future work.

2. MAXIMUM FLOW PROBLEM

Maximum flow problem is a well-known problem in graph theory [4]. It is formulated as follows: given an oriented graph $G = (V, E)$ with two distinct vertices s and t called *the source* and *the sink* and capacities $c_i \geq 0$ for each edge $e_i \in E$, one needs to find *a maximum flow*.

The maximum flow is a set of numbers f_i such that:

- for all $1 \leq i \leq |E|$, $0 \leq f_i \leq c_i$;
- for each vertex except for s and t the sum of f_i for

GECCO '15 July 11-15, 2015, Madrid, Spain

ACM ISBN 978-1-4503-3488-4/15/07.
DOI: http://dx.doi.org/10.1145/2739482.2768487

the incoming edges is equal to the sum of f_i for the outgoing edges;

- for the vertex s the sum of f_i for the outgoing edges minus the sum of f_i for the incoming edges is maximum possible.

There are many methods for solving the maximum flow problem. Some of them are:

- the Ford-Fulkerson algorithm [8], with running time $O(V \cdot E \cdot C_{max})$, where $C_{max} = \max_i(c_i)$;
- the Edmonds-Karp algorithm [7], $O(V \cdot E^2)$;
- the Dinic algorithm [5] with running time $O(V^2 \cdot E)$ which can be refined to $O(E \min(E^{1/2}, V^{2/3}))$ for unit capacities;
- the improved shortest path algorithm [2], $O(V^2 \cdot E)$;
- the push-relabel algorithm [9], $O(V^2 \cdot E)$.

In this paper we consider only the Dinic algorithm and the improved shortest path algorithm. This choice is motivated by the fact that in the previous paper [3] only algorithms which find augmenting paths were considered, and the two mentioned algorithms were found to be efficient, while demonstrating different behavior in terms of finding tests against them.

3. EVOLUTIONARY ALGORITHMS

In the paper we compare the following algorithms:

- (1+1) evolutionary algorithm. It produces one mutant from single parent, and if the fitness of the mutant is at least as good as the parent's it replaces the parent.
- $(1 + (\lambda, \lambda))$ evolutionary algorithm. On each iteration it produces λ individuals using mutation, then takes the best individual according to the fitness function, then produces λ individuals using crossover of the parent and the best individual. The best of the resulting individuals replaces the parent if it is at least as good.
- $(1 + \lambda)$ evolutionary algorithm. It produces λ individuals on each iteration using mutation, then the best of these individuals replaces the parent if it is at least as good. To maintain equal number of generations for this algorithm and the $(1 + (\lambda, \lambda))$ EA, we use the $(1 + 2\lambda)$ version throughout the paper.
- the genetic algorithm (GA) from [3]. It is a rather standard genetic algorithm which uses tournament selection to choose individuals for reproduction, then applies single-point crossover and mutation to the selected individuals, and forms the new generation using elitist selection.

The optimization problem we consider in this paper is generation of tests (instances of input data) for the maximum flow algorithms. The aim is to maximize the running time of an algorithm on a test. This problem is closely related to worst-case execution time test generation and to the complexity analysis of algorithms. The use of evolutionary algorithms for solving this problem is motivated by the fact that it is typically hard to find such tests for maximum flow algorithms because their performance is significantly input-sensitive. Several ways to construct such tests are known from the literature [13, 1, 10], but they all require deep knowledge of the algorithms under test.

We use precisely the same individual encoding and evolutionary operators as in [3]. An individual encodes a test input for a maximum flow algorithm and represents a graph with source, sink and capacities on edges. It is a list of edges, where each edge is a triple consisting of a source vertex s, a target vertex t and capacity c. The maximum number of vertices V, the number of edges in each individual E and the maximum capacity C are the parameters of the optimization problem. The source of the graph is the vertex 1 and the sink is the vertex V. Additionally we maintain a constraint $s < t$ for every edge of the graph, which was shown to be efficient in [3].

Initial population for each algorithm consists of individuals with random edges for which the source and the target are chosen uniformly at random in range $[1; V]$ (while source is less than target) and capacity in range $[1; E]$. The mutation operator replaces each edge with a randomly generated one with a probability p. For the $(1 + 1)$ EA, the $(1 + 2\lambda)$ EA and the GA this probability is equal to $1/E$, while for the $(1 + (\lambda, \lambda))$ EA it is equal to λ/E. Additionally, the $(1 + (\lambda, \lambda))$ EA uses a uniform crossover with the exchange probability of $1/\lambda$, while the GA uses single-point crossover.

The fitness function is the number of edges visited during finding of the maximum flow by the algorithm. This function was shown to be roughly proportional to the running time of the maximum flow algorithm and to the most efficient one in terms of results of optimization during the fixed budget [3].

4. EXPERIMENTS AND RESULTS

We set the maximum number of vertices V, the maximum number of edges E and the maximum capacity C to 100, 5000 and 10 000 correspondingly as in [3]. We used the computational budget of 500 000 fitness function evaluations for all algorithms. The generation sizes λ for the $(1 + 2\lambda)$ EA and the $(1 + (\lambda, \lambda))$ EA were 8, 16 and 25. For each maximum flow algorithm, each evolutionary algorithm and each generation size (if applicable), 50 runs was performed.

The resulting minimum, maximum, mean and median fitness values rounded to the nearest integer are presented in Table 1 for the Dinic algorithm and in Table 2 for the improved shortest path algorithm.

To detect statistically different configurations, the Wilcoxon rank sum test implemented in R programming language [12] was performed for each pair of configurations. The p-values are presented in Table 3 for the Dinic algorithm and in Table 4 for the improved shortest path algorithm. In both tables, in a row A and column B the p-value is specified for the alternative hypothesis that the median fitness value for the algorithm A is greater than for the algorithm B. The cells with p-value less than 0.05 are marked gray.

For the Dinic algorithm it can be clearly seen that the genetic algorithm from [3] clearly outperforms all other algorithms. All other algorithms can be divided into two groups: the worst algorithms (adaptive $(1 + (\lambda, \lambda))$ EA, $(1 + 2 \times 25)$ EA and $(1 + 1)$ EA), which compare unfavorably with all other algorithms, and the mediocre algorithms (non-adaptive $(1 + (\lambda, \lambda))$ EA for all λ as well as $(1 + 2 \times 8)$ and $(1 + 2 \times 16)$ EA).

Table 1: Results for runs with fitness function based on Dinic algorithm

	1+1	GA	1+(8,8)	1+(16,16)	1+(25,25)	1+2×8	1+2×16	1+2×25	Adaptive
MIN	67145	309133	130176	102692	101742	125849	102561	78926	109852
MEAN	272366	451416	320159	321387	310483	309129	299538	276648	266835
MEDIAN	272267	438240	317912	323329	312850	308554	298174	269960	245808
MAX	449293	590435	507201	590950	483404	534256	569870	464256	449322

Table 2: Results for runs with fitness function based on Improved Shortest Path algorithm

	1+1	GA	1+(8,8)	1+(16,16)	1+(25,25)	1+2×8	1+2×16	1+2×25	Adaptive
MIN	193004	531235	350791	303683	221711	310632	189908	356823	447256
MEAN	714660	687082	725167	673033	720410	759805	703207	700440	682743
MEDIAN	754880	703218	743064	678694	724629	812251	755892	717721	682654
MAX	948941	805580	912274	898807	891737	937657	921802	904839	878755

Table 3: Wilcoxon tests for runs with fitness function based on Dinic algorithm

	1+1	GA	1+(8,8)	1+(16,16)	1+(25,25)	1+2×8	1+2×16	1+2×25	Adaptive
1+1	–	1.0	$9.9 \cdot 10^{-1}$	$9.8 \cdot 10^{-1}$	$9.6 \cdot 10^{-1}$	$9.5 \cdot 10^{-1}$	$8.7 \cdot 10^{-1}$	$5.8 \cdot 10^{-1}$	$3.8 \cdot 10^{-1}$
GA	$7.9 \cdot 10^{-14}$	–	$4.1 \cdot 10^{-11}$	$1.1 \cdot 10^{-9}$	$2.0 \cdot 10^{-11}$	$3.2 \cdot 10^{-12}$	$3.7 \cdot 10^{-11}$	$5.6 \cdot 10^{-15}$	$1.4 \cdot 10^{-14}$
1+(8,8)	$1.2 \cdot 10^{-2}$	1.0	–	$5.0 \cdot 10^{-1}$	$3.4 \cdot 10^{-1}$	$3.7 \cdot 10^{-1}$	$1.6 \cdot 10^{-1}$	$7.8 \cdot 10^{-3}$	$5.0 \cdot 10^{-4}$
1+(16,16)	$2.3 \cdot 10^{-2}$	1.0	$5.0 \cdot 10^{-1}$	–	$2.9 \cdot 10^{-1}$	$2.5 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	$2.5 \cdot 10^{-2}$	$6.7 \cdot 10^{-3}$
1+(25,25)	$3.9 \cdot 10^{-2}$	1.0	$6.6 \cdot 10^{-1}$	$7.1 \cdot 10^{-1}$	–	$4.6 \cdot 10^{-1}$	$2.5 \cdot 10^{-1}$	$3.9 \cdot 10^{-2}$	$8.1 \cdot 10^{-3}$
1+2×8	$4.6 \cdot 10^{-2}$	1.0	$6.4 \cdot 10^{-1}$	$7.5 \cdot 10^{-1}$	$5.4 \cdot 10^{-1}$	–	$3.0 \cdot 10^{-1}$	$3.4 \cdot 10^{-2}$	$6.5 \cdot 10^{-3}$
1+2×16	$1.3 \cdot 10^{-1}$	1.0	$8.4 \cdot 10^{-1}$	$8.6 \cdot 10^{-1}$	$7.5 \cdot 10^{-1}$	$7.1 \cdot 10^{-1}$	–	$1.6 \cdot 10^{-1}$	$7.1 \cdot 10^{-2}$
1+2×25	$4.2 \cdot 10^{-1}$	1.0	$9.9 \cdot 10^{-1}$	$9.8 \cdot 10^{-1}$	$9.6 \cdot 10^{-1}$	$9.7 \cdot 10^{-1}$	$8.4 \cdot 10^{-1}$	–	$1.9 \cdot 10^{-1}$
Adaptive	$6.2 \cdot 10^{-1}$	1.0	1.0	$9.9 \cdot 10^{-1}$	$9.9 \cdot 10^{-1}$	$9.9 \cdot 10^{-1}$	$9.3 \cdot 10^{-1}$	$8.1 \cdot 10^{-1}$	–

Table 4: Wilcoxon tests for runs with fitness function based on Improved Shortest Path algorithm

	1+1	GA	1+(8,8)	1+(16,16)	1+(25,25)	1+2×8	1+2×16	1+2×25	Adaptive
1+1	–	$3.3 \cdot 10^{-2}$	$3.9 \cdot 10^{-1}$	$3.9 \cdot 10^{-2}$	$3.3 \cdot 10^{-1}$	$9.0 \cdot 10^{-1}$	$3.7 \cdot 10^{-1}$	$1.9 \cdot 10^{-1}$	$4.9 \cdot 10^{-2}$
GA	$9.7 \cdot 10^{-1}$	–	1.0	$5.8 \cdot 10^{-1}$	$9.8 \cdot 10^{-1}$	1.0	$9.7 \cdot 10^{-1}$	$8.6 \cdot 10^{-1}$	$3.6 \cdot 10^{-1}$
1+(8,8)	$6.1 \cdot 10^{-1}$	$3.5 \cdot 10^{-3}$	–	$2.9 \cdot 10^{-2}$	$4.2 \cdot 10^{-1}$	$9.8 \cdot 10^{-1}$	$5.0 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	$1.7 \cdot 10^{-2}$
1+(16,16)	$9.6 \cdot 10^{-1}$	$4.3 \cdot 10^{-1}$	$9.7 \cdot 10^{-1}$	–	$9.5 \cdot 10^{-1}$	1.0	$9.3 \cdot 10^{-1}$	$8.3 \cdot 10^{-1}$	$4.8 \cdot 10^{-1}$
1+(25,25)	$6.8 \cdot 10^{-1}$	$2.2 \cdot 10^{-2}$	$5.8 \cdot 10^{-1}$	$4.9 \cdot 10^{-2}$	–	$9.8 \cdot 10^{-1}$	$5.1 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	$4.0 \cdot 10^{-2}$
1+2×8	$1.0 \cdot 10^{-1}$	$2.6 \cdot 10^{-5}$	$1.9 \cdot 10^{-2}$	$2.5 \cdot 10^{-4}$	$2.0 \cdot 10^{-2}$	–	$3.8 \cdot 10^{-2}$	$7.7 \cdot 10^{-3}$	$2.9 \cdot 10^{-4}$
1+2×16	$6.3 \cdot 10^{-1}$	$3.3 \cdot 10^{-2}$	$5.0 \cdot 10^{-1}$	$7.3 \cdot 10^{-2}$	$5.0 \cdot 10^{-1}$	$9.6 \cdot 10^{-1}$	–	$3.2 \cdot 10^{-1}$	$8.8 \cdot 10^{-2}$
1+2×25	$8.2 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	$7.6 \cdot 10^{-1}$	$1.7 \cdot 10^{-1}$	$7.6 \cdot 10^{-1}$	$9.9 \cdot 10^{-1}$	$6.8 \cdot 10^{-1}$	–	$1.9 \cdot 10^{-1}$
Adaptive	$9.5 \cdot 10^{-1}$	$6.4 \cdot 10^{-1}$	$9.8 \cdot 10^{-1}$	$5.2 \cdot 10^{-1}$	$9.6 \cdot 10^{-1}$	1.0	$9.1 \cdot 10^{-1}$	$8.2 \cdot 10^{-1}$	–

A more detailed comparison (see plots of the best runs in Figure 1 and of the median runs in Figure 2) reveals that the $(1+(\lambda,\lambda))$ EA family gains high fitness values very fast in the beginning but then stagnates, while all other non-genetic algorithms grow slower. In the same time, the genetic algorithm does not seem to stagnate at the 500 000th fitness evaluation. One of possible hypothetical explanations is that the single-point crossover helps preserving and evolving large consecutive groups of edges.

A similar situation appears for the improved shortest path algorithm as well, but roles change. The $(1 + 2 \times 8)$ EA is the clear overall winner which compares favorably with all algorithms except for the $(1 + 1)$ EA and is statistically indistinguishable with the latter one. The genetic algorithm from [3] is now among the losers, as well as the adaptive $(1 + (\lambda, \lambda))$ EA and the $(1 + (16, 16))$ EA. Plots of best (Figure 3) and median (Figure 4) runs show that the genetic algorithm is not stagnated, but progresses slowly.

5. CONCLUSION

We presented the first (to the best of our knowledge) attempt to apply the $(1 + (\lambda, \lambda))$ evolutionary algorithm to a practical optimization problem – generation of hard tests for maximum flow algorithms.

The experimental results augmented with basic statistical analysis show that the $(1 + (\lambda, \lambda))$ EA is never the best, but is not the worst as well – apart from the variant with adaptive generation sizes and mutation probabilities. The latter variant, while showing the proven $O(N)$ running time complexity for OneMax, is among the worst ones for both

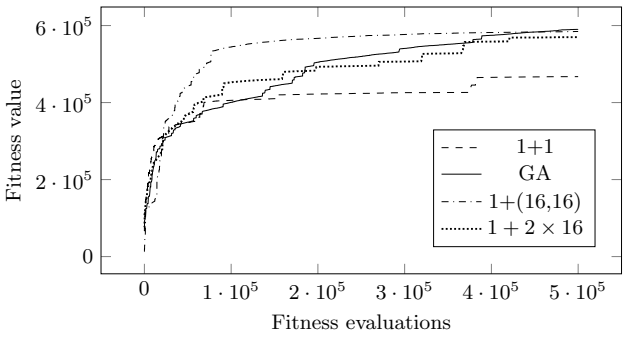

Figure 1: Best runs for the Dinic algorithm

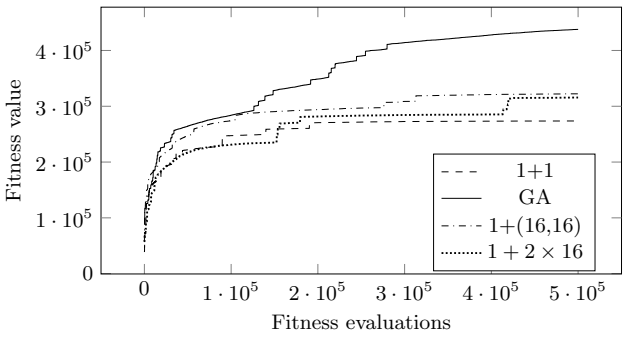

Figure 2: Median runs for the Dinic algorithm

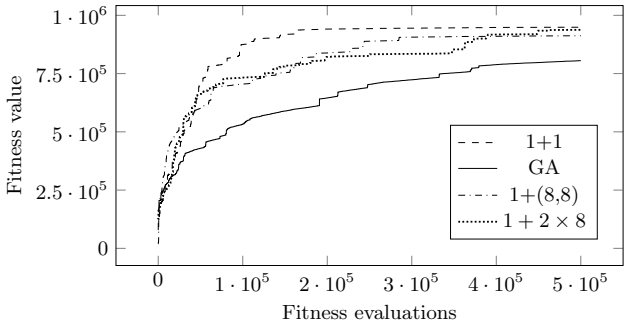

Figure 3: Best runs for the ISP algorithm

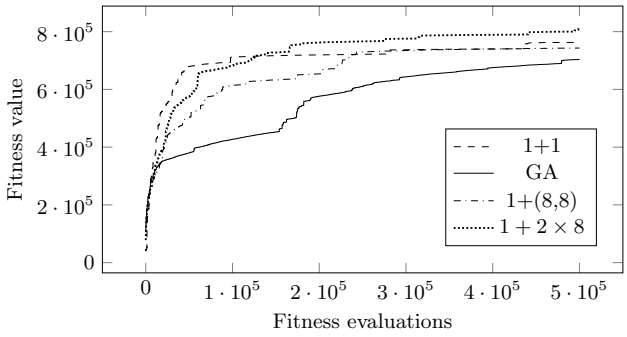

Figure 4: Median runs for the ISP algorithm

considered maximum flow algorithms, which suggests that the adaptation heuristic seems to be overfitted for OneMax and is not promising for at least some other problems.

On the problem of maximum flow test generation, the $(1 + (\lambda, \lambda))$ EA showed itself a rather good default choice which is almost insensitive to the generation size. However, there probably exist some ways to tailor this algorithm using a suitable heuristic to the considered problem, which, in turn, may serve as a starting point to more general methods of improving the running time of the $(1+(\lambda, \lambda))$ EA for more general classes of problems.

The source code for the experiments is published at GitHub[1].

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

6. REFERENCES

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

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

[3] M. Buzdalov and A. Shalyto. Hard test generation for augmenting path maximum flow algorithms using genetic algorithms: Revisited. In *Proceedings of IEEE Congress on Evolutionary Computation*, 2015 (to appear).

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

[1] https://github.com/NinerLP/papers/tree/master/one-ll

[5] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11(5):1277–1280, 1970.

[6] B. Doerr, C. Doerr, and F. Ebel. From black-box complexity to designing new genetic algorithms. *Theoretical Computer Science*, 567:87–104, 2015.

[7] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–262, 1972.

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

[9] 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*, pages 136–146, New York, NY, USA, 1986. ACM.

[10] D. Goldfarb and M. D. Grigoriadis. A computational comparison of the Dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13(1):81–123, 1988.

[11] P. S. Oliveto and C. Witt. On the runtime analysis of the simple genetic algorithm. *Theoretical Computer Science*, 545:2–19, 2014.

[12] R Core Team. R: A language and environment for statistical computing. http://www.R-project.org/, 2013.

[13] N. Zadeh. Theoretical efficiency of the Edmonds-Karp algorithm for computing maximal flows. *Journal of the ACM*, 19(1):184–192, 1972.