Size-flexible Test Generation for Maximum Flow Algorithms Using Genetic Algorithm

Team4
20150045 Jihoon Ko
20150803 Junho Han
20160392 Jaewon Wi*
20160622 Minkyu Jo

***Abstract*— Generating test data that yield long execution time for an arbitrary algorithm is an important, yet non-trivial job. As a search-based approach to this task, Buzdalov and Shalyto (2015) [2] showed remarkable results on automated test generation for maximum flow problem using GA. This report extends upon their work, and introduce size-flexible GA that makes it possible to generate tests using other tests of different size. Resize operators and size manipulation scheme, which are needed for the size-flexible GA, is introduced as well. This report also provides various experiment results about fitness value and execution time, which shows 1.1x – 1.3x better than original GA.**

I. Introduction

Test generation is an important topic in the field of computer engineering. It aims to generate tests with high fitness, which is measured in domain-specific ways. In the case of tests for classical, deterministic algorithms, measuring the execution time is a reasonable way to estimate the fitness; the test with longer execution time is harder, thus has higher fitness. However, generating test data that result in long execution time requires a great understanding of the specific algorithm, often accompanied by complex mathematical proofs. Also, the knowledge for generating hard tests for one algorithm rarely helps generating hard tests for other algorithms.

Buzdalov and Shalyto (2015) [2] approached this problem with search-based techniques. They automatically generated tests for various algorithms that solve the maximum flow problem, using genetic algorithm. They demonstrated that automated test generation with genetic algorithm is able to make harder tests in general, compared to existing test generators. They also found out some of the better-working configurations for the genetic algorithm.

One possible downside of their approach is that each test for a given graph size must be generated individually. In other words, there is no way to generate tests from existing tests of different size. This can be problematic since genetic algorithm produces better results when given more time in general; taking longer time to generate a single test leads to lower fitness of the overall tests when the total budget is fixed.

*Withdrew the course

The aim of this report is to improve upon the their work, by introducing the concept of size-flexible genetic algorithm. We first introduce four resize operators: Split, Merge, Add, and Delete, that stochastically modifies the size (i.e. the number of vertices and the number of edge) of a given graph. In Split and Merge, a vertex is split into two or two vertices are merged into one in the graph respectively. In Add and Delete, an edge is added to or deleted from the graph respectively. We also introduce size manipulation scheme which combines these resize operators in a specific order to match the exact desired size of the graph. Then we introduce the detailed description of the size-flexible genetic algorithm, where the size of the individual graph gradually changes over generation, while performing the original genetic algorithm. Manipulating the graph size also counts for the fitness evaluation budget, in addition to the genetic algorithm, so we carefully designed our algorithm to balance the two. Finally, We provide experiment results of our algorithm, comparing the results with the existing method.

II. Background

A. Maximum Flow Problem

The definition of maximum flow problem is as follows:

- Graph G with V vertices and E directed edges is given. The vertices are numbered from 1 through V, and the edges are numbered from 1 through E.
- Edge i ($1 \leq i \leq V$) has start vertex x_i, end vertex y_i, and capacity c_i.
- The source vertex s and the sink vertex t is given.
- The solution to the problem is to find an assignment of flows f_1, f_2, ..., f_V which:
  - for each edge i ($1 \leq i \leq E$), $0 \leq f_i \leq c_i$ holds.
  - for each vertex i ($1 \leq i \leq V$) except s and t, the sum of flows of edges that has i as the start vertex must be equal to the sum of flows of edges that has i as the end vertex.
  - maximizes the sum of flows of edges that has s as the start vertex minus the sum of flows of edges that has s as the end vertex. This value is called f, the maximum flow of G.

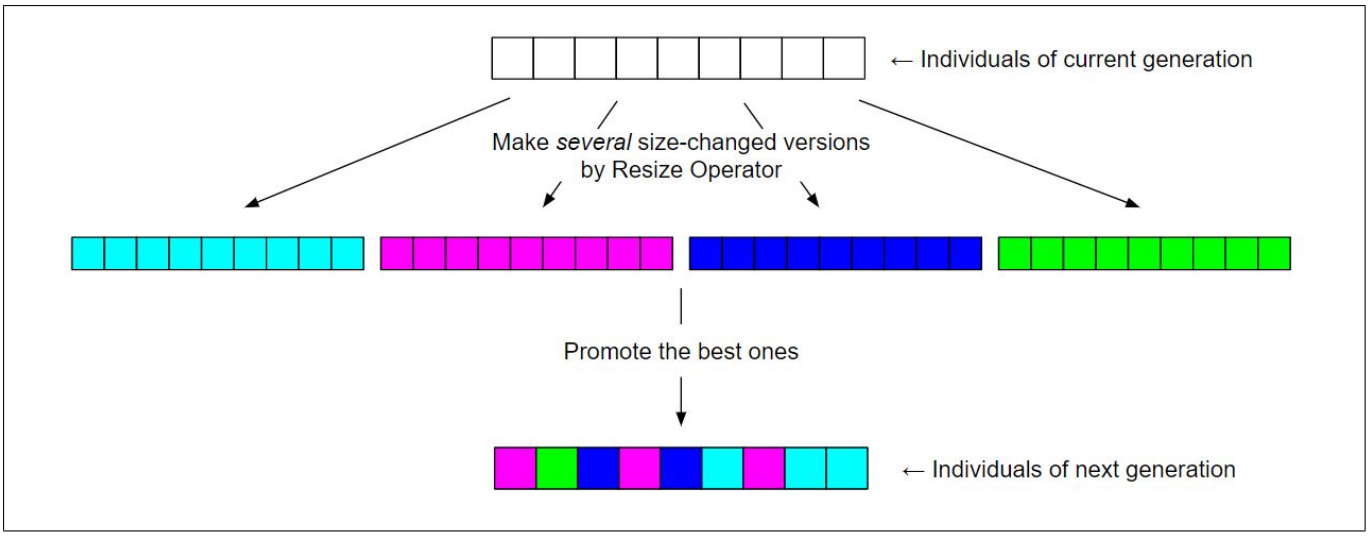

Fig. 1: Description of the size manipulation between generations

B. Maximum Flow Algorithms

There are many known maximum flow algorithms [3]:

- the Ford-Fulkerson algorithm [6], the running time is $O(E \cdot f)$.
- the Edmonds-Karp algorithm [5], the running time is $O(V \cdot E^2)$.
- the Dinic algorithm [4], the running time is $O(V^2 \cdot E)$.

III. Resize Operators

In this section, the resize operators, which are used to modify the size of a graph, are described. Moreover, the size manipulation scheme where the operators are applied to manipulate the graph size, is explained.

It is assumed that the given graph is $G = (V, E, C)$ where V is the set of vertices, E is the set of directed edges, C is the maximum capacity of each edge. An each edge is represented by $e = (s, t, c)$ where s is its source vertex, t is its target vertex, and c is its capacity.

A. Split Operator

A vertex v is randomly chosen from the given graph, and a new vertex v' is generated. Then all the outgoing edges from v are relocated to v', i.e., for every $u \in V$, if $(v, u, c) \in E$, (v, u, c) is removed and (v', u, c) is added. Finally, an random directed edge (v, v', c_r) is added with a random capacity $c_r \in [1, C]$.

After the operation, the graph changes to $G = (V', E', C)$ where $|V'| = |V| + 1$.

B. Merge Operator

Two vertices v and u are randomly chosen from the given graph, and they are merged into one vertex v'. Then each incident edge of v and u are relocated to v' preserving its direction.

If the graph type is acyclic, v and u are selected randomly satisfying that they are next to each other in a topological order to ensure that a new graph becomes acyclic. Also, if there are loops, they are removed.

After the operation, the graph changes to $G = (V', E', C)$ where $|V'| = |V| - 1$.

C. Add Operator

Two distinct vertices v and u are randomly chosen from the graph. Then an random directed edge (v, u, c_r) is added with a random capacity $c_r \in [1, C]$.

If the graph type is acyclic, v and u are selected satisfying v must precede u in a topological order.

D. Delete Operator

An edge is randomly chosen from the graph, and it is removed from the graph.

E. Size Manipulation Scheme

1) ΔV and ΔE are received as inputs.
2) Split/Merge operator is applied $|\Delta V|$ times to fit vertex size. In this procedure, the number of edges changes as described. It is denoted by ΔE_v.
3) Add/Delete operator is applied $|\Delta E - \Delta E_v|$ times to fit edge size.

IV. Size-flexible Genetic Algorithm

In this section, the size-flexible genetic algorithm is described. It basically consists of the size manipulation method described in the previous section and the original genetic algorithm, which is the same as one described in [2].

It is assumed that the initial and final graph sizes are given, i.e., the initial number of vertices V, the initial number of edges E, the final number of vertices V', and the final number of edges E' are given. It is also assumed that the maximum capacity of an edge C doesn't change.

The representation of each *individual* is a list of edge as described in [2], so the standard crossover and mutation operator can be applied.

A. Original Genetic Algorithm Scheme

A population size $T_{population}$ is fixed to 100. Authors of [2] used various options for *initial population creation*, *crossover operator* and *fitness function*. Then they finally found that *the number of visited edges* correlates best with execution time, and *random acyclic creation* and *single-point crossover* are the most effective options in terms of fitness values. So, in this report, the above plausible and effective options are fixed and are used in the original genetic algorithm.

A single iteration of the genetic algorithm is performed as follows:

1) The *reproduction selection* operator is used to select $T_{population}$ individuals from the population, which are then used to create offspring.
2) The selected individuals are grouped in pairs, and the *crossover operator* generates a new pair of individuals from each pair.
3) The *mutation operator* is applied to all individuals generated by crossover operator, and each mutated individual is evaluated by the *fitness function*.
4) A new population is formed from the old population and the newly generated individuals by the *survivor selection operator*.

Details of each of steps and operators are described in [2].

B. Size Manipulation between Generations

To apply and reuse the original genetic algorithm, all individuals must have same size on each generation. So size manipulation should be done between generations. However, on this circumstance, no elitism can be applied because an individual may lose its quality while changing the graph size.

To overcome this issue, for each individual, several size-changed versions are made by resize operators as candidates for the next generation. Among those candidates, the best one is promoted, and the each promoted individual form a next generation. In particular, all individuals of the next generation have the same size. Fig. 1 illustrates this procedure.

C. Size-flexible Genetic Algorithm Details

As indicated, our size-flexible genetic algorithm uses the original genetic algorithm. Since the graph size changes over and over, the original GA can be performed on graphs of different sizes. So, each step consists of one size manipulation and several original GA. Moreover, we restrict size manipulation to change the vertex size only by 1 to prevent oscillation of fitness changes. Reasonably, the edge size are changed at the same rate on each step.

In the previous subsection B, it is indicated that several size-changed versions are made, but exactly how many versions are created? Let T_{SM} denote it. Analogously, Let T_{GA} denote the number of performs of the original GA on each step. Let T_{step} denote the number of steps, and it is actually calculated by $|V' - V|$. Assume fitness evaluation budget is given by T_{budget}. Then we get a formula:

$$T_{step} \times (T_{SM} + T_{GA}) \times T_{population} \leq T_{budget}.$$

So, we set $T_{SM} = T_{GA}$ to ensure that the number of fitness evaluations on size manipulation and the original GA are equally distributed, and set this value to the possible maximum number.

V. Experiment

A. Experiment Setup

1) Implement Maximum Flow Algorithms:

The following maximum flow algorithms were used in experiments.

- the Edmonds-Karp algorithm [5], the running time is $O(V \cdot E^2)$.
- the Dinic algorithm [4], the running time is $O(V^2 \cdot E)$.

We borrowed the implementation style from [7] and applied in each implementation. These algorithms are written in C++ for performance purpose, and the correctness of each implementation was tested with problems in a widely-used Online Judge.

2) RQ1: Does size-flexible GA have an advantage against the original GA in terms of fitness value?
In order to answer RQ1, we compared fitness value of two types of GA in this experiment using Dinic algorithm.

- A type: It starts with the small graph of best quality and run size-flexible GA for 5,000 generations (500,000 fitness evaluation). After running size-flexible GA, It runs original GA for 5,000 generations.
- B type: It starts with a random large graph and runs original GA for 10,000 generations.

We set the small graph size as $|V| = 20, |E| = 200$, and the large graph size as $|V| = 40, |E| = 800$. and the maximum capacity as $10,000$.

3) RQ2: Does size-flexible GA have an advantage against the original GA in terms of execution time?
In order to answer RQ2, we compared execution time of two types of GA in this experiment using Dinic algorithm and Edmonds-Karp algorithm. The rest setups are same as RQ1.

B. Results

RQ1: Does size-flexible GA have an advantage against the original GA in terms of fitness value?
We compared fitness value of A type and B type 10 times, and Fig. 2 shows results of averaged fitness value over generations. After $10,000$ generations ($1,000,000$ fitness evaluations), the fitness value of type A was approximately 1.15x higher than that of type B.

RQ2: Does size-flexible GA have an advantage against the original GA in terms of execution time?
We compared execution time of A type and B type. Tables below show results of execution time of each model. Generating test data using A type was approximately 1.2x – 1.3x faster than generating test data using type B for both maximum flow algorithms.

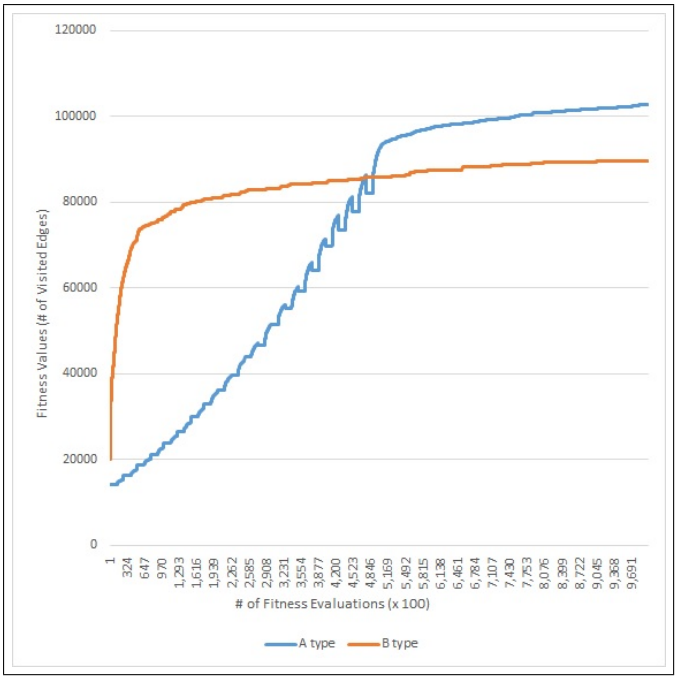

Fig. 2: Average fitness value over generations.

	type A	type B
Trial 1	160.2s	197.8s
Trial 2	177.0s	217.0s
Trial 3	200.7s	241.4s
Trial 4	193.2s	252.4s
Trial 5	196.2s	253.2s
Average	185.5s	232.4s

TABLE I: Execution time, using Dinic algorithm

	type A	type B
Trial 1	385.8s	408.4s
Trial 2	332.6s	381.6s
Trial 3	387.8s	500.0s
Trial 4	483.1s	553.7s
Trial 5	431.9s	528.7s
Average	404.2s	474.5s

TABLE II: Execution time, using Edmonds-Karp algorithm

VI. Conclusion

We proposed size-flexible genetic algorithm in this report and evaluated quality of the algorithm by comparison with the original genetic algorithm. We found that our size-flexible genetic algorithm is faster than the original genetic algorithm when building a graph of same quality. Also, average quality of size-flexible genetic algorithm from a good configuration was better than that of the original genetic algorithm. By using the original genetic algorithm, we can only build a single test and other tests are simply dominated. However, Size-flexible genetic algorithm can build numerous viable tests of different sizes in the process.

VII. Links

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

References

[1] 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.

[2] 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*, pp. 2121-2128, 2015.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to Algorithms, 3rd Edition", Cambridge, Massachusetts: MIT Press, 2009

[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] 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.

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

[7] J. Koo, H. Kang and M. Jo, "DeobureoMinkyuParty", GitHub repository, https://github.com/koosaga/DeobureoMinkyuParty

[1] `https://github.com/jyuno426/cs454_team4`