# A Genetic Algorithm for the Minimum Vertex Cover Problem with Interval-Valued Fitness

**Benedek Nagy[1], Péter Szokol[2]**

[1] Department of Mathematics, Faculty of Arts and Sciences, Eastern Mediterranean University, Famagusta, North Cyprus, via Mersin-10, Turkey, benedek.nagy@emu.edu.tr

[2] EPAM Systems, Bókay János u. 44, 1083 Budapest, Hungary, peter_karoly_szokol@epam.com

*Abstract: This paper presents a new genetic algorithm for the minimum vertex cover problem. It uses interval-valued fitness and greedy error correction to obtain phenotypes (candidate solutions). By the interval-valued fitness the fitness of the candidate solution is measured not only for the whole graph, but for some of its disjoint subgraphs. A new candidate solution is obtained from those subgraphs that have the best performance among the subgraphs of the candidates with the same set of vertices. The interval-valued fitness accelerates the search effectively for graphs with a great deal of nodes and relatively small numbers of edges. In the presented algorithm we prefer to distinguish genotypes and phenotypes and do not use Lamarckian inheritance. Phenotypes are easily generated by greedy error correction from the genotypes and, in this way, a larger variety of genomes can be used during the process.*

*Keywords: genetic algorithm; minimum vertex cover; interval-valued fitness; phenotypes; memetic algorithm*

## 1 Introduction

A vertex cover of a graph is a set of vertices, such that each edge of the graph is incident to at least one vertex of the set. Minimum vertex cover means that the size of the vertex cover set is minimal. Finding the minimum vertex cover is a classical NP-complete problem. As such, genetic algorithm (GA) seems an ideal method for the minimum vertex cover problem to obtain reasonable solution in reasonable time. Genetic algorithms are based on the idea of Charles Darwin's evolution. GA is a heuristic search algorithm, it finds a relatively good solution in a short amount of time, and it can be stopped any time, it will always have a, relatively, good solution [1, 2, 3].

Every GA has a population of candidate solutions. For the sake of simplicity, from now on, we call them solutions, instead of candidate solutions. Each of these solutions have a genome, the entirety of their hereditary information. The algorithm must calculate the fitness of the solutions and the next generation can inherit the genomes of the best solutions. A child can inherit genes from one or more parents. There is also a small chance for mutation. The algorithm measures the fitness of each solution of the new generation again, and the cycle continues until a given condition is met.

In the literature there are various approaches to use GA and related methods for obtaining reasonable solutions for the minimum vertex cover problem. Heuristic approximation is used based on the share degree distribution of the vertices [4]. The performances of random local search algorithm and the basic (1+1) evolutionary algorithm are compared in various subclasses of graphs in [5]. Various heuristic algorithms for minimum vertex cover are compared in [6], including hierarchical Bayesian optimization algorithm, branch-and-bound problem solver, simple genetic algorithm and the parallel simulated annealing to show that evolutionary, and so, genetic algorithms are reasonable choices to solve the problem. In [7], the initial population of the GA is created in mathematical manner (uniformly distributed initial population) instead of a random population; moreover, the minimum vertex-cover problem is converted into constrained combinatorial optimization problem. Local optimization technique, in fact, hill climbing was used as Lamarckian evolution in [8] to obtain a hybrid GA. The genetic algorithms in these papers used mutation to avoid falling into a local optimum of the problem. The ideas mentioned in them inspired also us to make experiments with our algorithm and improve it. Our novel idea is to use interval-valued fitness, however, we have combined it with various other well-known GA methods, such as, e.g., with elitism and local (error) correction.

GAs give a lot of freedom to the programmer. Creating the structure of the genome is the first step. The most obvious way to do this for the minimum vertex cover problem, is to create a gene for every node. The gene can have two values: true or false. True means that the corresponding node is in the vertex cover set, false means that the node is not in the vertex cover set. We distinguish genotypes and phenotypes [9]. It results a better variety of the gene pool. It is possible that a vertex cover candidate set is not covering the graph. In this case, the genotype would not be a vertex cover, but the algorithm can add nodes to the candidate vertex cover, and thus, the phenotype becomes one. With the fast, and effective greedy algorithm, the phenotype improves significantly. Creating the phenotype can be viewed as a part of the fitness function. The fitness function assigns fitness values to the genomes. Fitness value indicates how good the solution is, i.e., how close it is to an ideal or optimal solution. Inheritance is another main part of GAs, and there are lots of opportunities here as well. We present a new idea, the interval-valued fitness method. We apply to the GAs the concept of intervals. It means that the child inherits gene sequences from one or more parents. Interval-values are finite unions of components over the unit interval. They are used to represent various fuzzy and many-valued logics [10, 11] and they are also used to introduce a new computing paradigm [12, 13] that is applied to solve various computationally hard problems [14, 15, 16] and classes of problems [14, 17], theoretically, based on its inner parallelism. Here we use this concept at the fitness function of the GA. If a parent phenotype has a good fitness value on a component of the interval, then the genes of the parent on this component will be added to the child's genome. This process is very useful if it is easy to distinguish various characteristics of a solution, and these characteristics are independent, or weakly dependent on each other. For the vertex cover problem, to find and distinguish various characteristics of a graph is not very straightforward, but we present some valuable ideas. It is expected that for graphs where there are partitions such that most of the edges are inside partitions the algorithm could efficiently use its interval-valued fitness to compose reasonable solutions on the partitions to a general solution for the whole graph.

The rest of the paper is organized as follows: First we introduce the problem, its mathematical representation, and the structure of a genome in Section 2. In Section 3, we describe four ideas for determining the partitions of a graph. We present a GA for determining the partitions of a graph in Section 4 and in Section 5, we explain the greedy error correction, a method to repair invalid vertex covers, obtaining the phenotypes from genotypes. Next, we describe our genetic algorithm for the minimum vertex cover in Section 6. Section 7 provides an example, Section 8 shows the Experimental Results and finally, Section 9 discusses the Conclusions.

## 2 Minimum Vertex Cover, the Genome of the GA and the Model of the Problem

In this section the minimum vertex cover problem and the structure of the genome for our GA is presented. Let a simple graph $G$ be given with $n$ vertices consisting of a set of vertices $V$. (The order of the vertices in $V$ is arbitrarily fixed.) The aim is to find the minimum vertex cover $C$, a set of vertices, where $C$ is a subset of $V$ such that each edge of $G$ is incident to at least one vertex in $C$ (that is $C$ is a so-called vertex cover of $G$) and the size of $C$ is minimal (among all vertex covers of $G$). $G$ can be connected or disconnected, and $G$ may have isolated vertices. An isolated vertex is not an endpoint of any edge, therefore the isolated vertex is not in any minimal vertex cover $C$. For an example of a simple graph and its minimum vertex cover see Figure 1. This graph $G$ has 6 vertices, two of those are indicated by circle, they form a vertex cover $C$ for this graph. It is also easy to see that there is no sole vertex in $G$ which forms a vertex cover alone. Thus, $C$ is, in fact, a minimum vertex cover. The minimum vertex cover problem is a classical optimization problem, and it is one of the 21 NP-complete problems listed by Karp [18]. Since this is one of the most known NP-complete problems, it plays an important role in computational complexity theory. On the other hand, since generally it is intractable, it is a good target for evolutionary computing and GA to find approximate solutions.

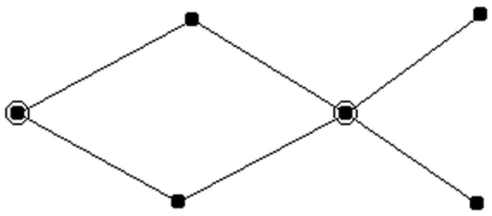

Figure 1
A graph, with six vertices and its minimum vertex cover

Now, let us turn to our approach. In the rest of this section, some details are shown how this problem can be modeled and represented by genes such that evolutionary steps can reasonably solve it. Also some essential details of our approach are described. In our approach the vertex covers are computed by greedy error corrections from the genome in deterministic manner (see Section 5), in this way, the vertex cover is, in fact, the phenotype corresponding to the given genome. The population of the GA consists of a set of genomes $\langle F, I, K \rangle$, where:

- $F$ is the fitness value of the genome is in fact computed for the phenotype, the vertex cover $C$ implied by the genome. It is calculated by the fitness function, and it represents the overall fitness of the solution. $F$ represents the fitness of the phenotype, meaning that first we use the greedy error correction on the solution of a genome, then we compute the fitness function on this modified solution. The greedy error correction guarantees that the phenotype is a vertex cover. $F = |C|$ (the size of the vertex cover).
  Note that $F$ is not a fitness function in the strict sense, since larger value of $F$ means weaker solutions; in this sense $F$ is more related to error-measures.
- $I = (I_1,I_2,...,I_m)$ is an ordered list (set) of interval fitness values of the phenotype solution $C$. Let $P = (P_1,P_2,...,P_m)$ denote the partitions of $V$. $P$ is the same for every genome, and every generation. If $P_j = \{a_1,a_2,...,a_k\}$, then $I_j$ is the component fitness of the solution on the subgraph of $G$ containing only vertices of $P_j$ and all the edges of $G$ which contain only such vertices. The component fitness on a subgraph means the number of vertices $E$ of the subgraph in the vertex cover phenotype, where $E$ is also a member of $C$, e.g., $I_j = | P_j \cap C |$.

- $K = (K_1, K_2, ..., K_n)$ is a binary vector of the vertices in the genome. If $K_i = 1$ (i.e., true), then vertex $V_i$ is in the genome for the cover set $C$ proposed by the phenotype of this genome.

For an example, see Figure 2. As it is shown, the partitions of the graph are also described by our data: there are three partitions in this example. The set of circled vertices is a vertex cover $C$, and its size is 8. The interval fitness is assigned to the partitions, and actually, the sum 2 + 3 + 3 = 8 gives the fitness of the vertex cover. Since the genotype, in this example, for simplicity and luckily, is the same as the phenotype, we do not need to apply the error correction, thus, in this case, $K$ and $C$ are identical. The binary genome vector $K$ has 1's in exactly those 8 position.

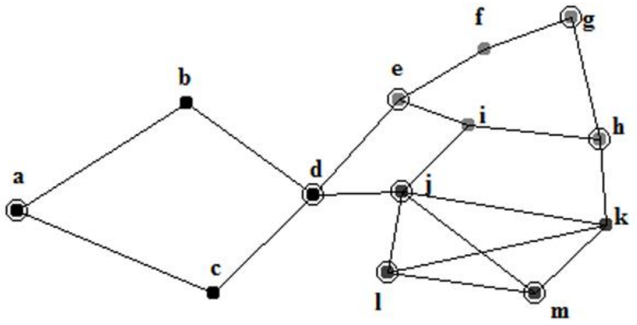

Figure 2
A graph where $P$ = ({a,b,c,d}, {e,f,g,h,i}, {j,k,l,m}), and one of the possible solutions, where $F$ = 8 ( = $|C|$), $I$ = (2,3,3), and $K$ = (1,0,0,1,1,0,1,1,0,1,0,1,1)

## 3 Initialization: Determining the Partitions

We use fitness values on interval components and it is a crucial point to divide the graph to partitions. The interval fitness function works more efficiently if every $P_j$ contains closely related vertices and the partitions are not connected so strongly. There are several methods to determine a feasible $P$.

- Random partitions
- Divide by the time of creation of the vertices
- Manually determine the partition of each vertex
- Spectral partitioning [19]
- GA, with a fitness function that calculates fitness by the dissimilarity of the neighbors of each vertex

The partitioning does not have to be optimal. But it has to be fast. It can help even if the partitioning is random, since as it will be detailed later, for every generation, only one genome will be created with the interval fitness method. For example, if the population size is constantly 200, then 199 children will evolve normally, only 1 child (super-child) will be created from the best intervals. Even if the partitioning is random, there is a good chance that the super-child will be successful. If that happens, the quality of the gene pool will significantly increase. Dividing by the time of creation is an idea only slightly better than the random partitioning. It is based on the assumption that there is a connection between the vertex structure and the time of creation of the vertices. According to our tests, spectral partitioning [19] takes much time. Even with a simple graph of 500 vertices, only finding the eigenvalues took 3 seconds. Creating the partitions takes precious time from the actual work, so it has to be fast. GA, combined with an idea similar to the k-means algorithm [20], seems to be the best choice, since this sub-problem is hard, but the solution does not have to be optimal, only close to optimal. The GA can stop after 1 second, and it still gives a relatively good solution.

## 4 Genetic Algorithm for Partitioning

The first step for both the genetic algorithm and the random method is to determine the number of partitions. 10 partitions for a graph with 10 vertices would result the same as there would not be groups at all, but little number of partitions for a big graph is not ideal either. After experimenting with various functions to determine the optimal number of partitions, we found the following function to work the best:

$$|P| = \frac{|V|^{0.6}}{3}$$

The GA for partitioning has to be fast, so the genome structure is not as straightforward as the genome structure of the main GA.

Let $R = \{R_1,...,R_k\}$ be the genome of an individual, where $k = |P|$, it is the number of partitions we plan to create. Every gene $R_i$ denotes a randomly chosen vertex from the vertices of the graph, it is the starting vertex of the partition.

The fitness function first simulates a deterministic vertex-conqueror game for every individual. The rules of the game are:

Every partition $P_i$ has a set of conquered vertices ($CV$), initially they contain only the randomly chosen starting vertex $R_i$. (If a partition has a starting vertex that is already taken by another partition, i.e., the same vertex is chosen for two different partitions, then the new partition conquers the previous partition, and the previous partition leaves the game with no conquered vertices, thus we obtain one less partition).

In every round, each vertex of the *CV* of each partition expands, meaning that they conquer every neighbor vertices, except those already conquered by any partition.

When no partition can expand any more, the first partition conquers every non-conquered vertices if there are any left (may happen in case of not connected graph). The game ends here.

The fitness value is as follows:

Let $g$ be the number of edges of the graph that connect vertices belonging to the same partition, and let *w* be the number of edges that connect vertices belonging to different partitions.

Let $a = \frac{g}{g+w}$, this is a number between 0 and 1, as $g$ and *w* are both positive numbers.

Let,

$$cmax = \max_{i=1..|P|} \{CV_i\}$$

Let,

$$cmin = \min_{i=1..|P|} \{CV_i\}$$

Let,

$$b = \frac{cmin}{cmax}$$

a number between 0 and 1, as *cmin* ≤ *cmax*, and both are positive numbers

The fitness value is: ($ab^2$). The aim is to have (almost) equal size partitions and it is better if the most of the edges are inside partitions.

For every new generation, the GA for partitioning uses elitism, roulette wheel selection, one-point crossover, and mutation. This algorithm stops after one second, meaning that using the intervals adds only one second penalty (i.e., cost) to the main algorithm.

## 5 Greedy Error Correction

In this paper, we explain how phenotypes are produced from genotypes with a deterministic greedy algorithm.

There is a problem with the simple GA approach: most of the time, the genotypes of the solutions are not even covering the graph. Finding only a cover set with pure natural selection and random mutations is difficult for a simple GA, but the aim is to find the minimal cover set, so the task is even more complicated.

Fortunately, there are easy methods to create a cover set from an imperfect cover set.

The simplest and fastest method is to check every edge. If an edge is not incident to any vertex in *C* (unstable edges), then add one of the vertices of this edge to *C*. Some nodes may be redundant; these unnecessary nodes can be removed [6].

We analyze the usage of a greedy method, first it checks every vertex that is not in *C* (unselected vertices), but adds them to *C* only if they have the highest number of unstable edges incident to them. In the next cycle, it checks every unselected vertices again, until there are no more unstable edges. This method is slower, because it runs for every genome in every generation, but our tests showed that the algorithm gets significantly better with the greedy error correction.

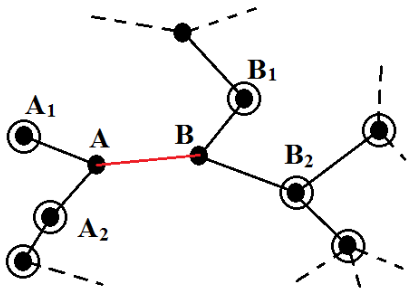

Figure 3
A subgraph with the last cycle of the greedy algorithm where each vertex has a maximum of one unstable edge

This correction works if the genome selects too few vertices. It is possible to correct those genomes that selects too many. Invert correction: if the neighbors of a selected vertex *X* are all selected vertices, then *X* can be removed from *C*.

Combining these ideas, it is possible to further improve these methods. At the last cycle of the greedy correction, there are no unselected vertices with multiple unstable edges. In this cycle, the choice of which vertex of an unstable edge should be selected is arbitrary. In some case however, this choice has an impact on the final solution, because of the final invert correction.

Consider the subgraph shown in Figure 3.

Between vertices A and B, there is an unstable edge. Since the greedy correction is at its last cycle, the algorithm can be sure that there are no vertices with more than one unstable edges. Suppose the algorithm first discovers vertex A. Vertex A has one unstable edge. The other vertex of this unstable edge is vertex B. It is granted that B has no other unstable edges, only this one.

Let us introduce a new concept, the compatible neighbor.

**Definition 1 (Compatible neighbor)** At the last cycle of the greedy correction, if B and A are neighbor vertices in a graph, A is selected, and beyond B, A has only selected neighbors, then A is a compatible neighbor of B.

The algorithm will decide as follows.

If B has more compatible neighbors than A, then select B, else select A.

Compatible neighbors are important, because the invert correction will deselect them if all their neighbors are selected. Since A has two compatible neighbors ($A_1$ and $A_2$), and B has one compatible neighbor ($B_2$), the algorithm will select A, in our example.

Lamarckian inheritance is the idea that characteristics developed during an organism's lifetime (the greedy error correction, in our case), can be inherited [21]. The question is if the algorithm should follow this idea, and pass through the genes reflecting the solution after the correction, or not.

Since the greedy error correction is not a stochastic method, it creates the same phenotypes for a given genotype. If the child inherits every gene of a parent's genotype perfectly, it will develop the same phenotype as the parent did. Furthermore, it is possible that the algorithm creates the same phenotype for different genotypes. That is why, using the Lamarckian inheritance could lead to a less varied gene pool, which is not desirable. By inheriting the original genes instead of the information in the phenotype, we increase the speed of converging to a near-optimal solution and increase the population diversity. Both of these features are important for a good GA [22]. On the other hand, the greedy error correction usually increases the number of vertices in *C*, therefore we expect better results in the new generation if these vertices in the genotype will be selected only if necessary.

In memetic algorithms some local search is done at the new individuals after the genetic operations [23, 24]. Our greedy error correction has a similar feature, but there is an important difference: we do not modify the genome (allowing to inherit its original version), only the phenotypes are vertex covers surely.

## 6 The Minimum Vertex Cover Algorithm

In this section, we describe how the new generation is computed in our GA.

The first 3 genomes of the next generation are 3 genomes of the previous population with the best *F* values. This is called elitism, it ensures that the best solutions will always stay in the population, with elitism, there is no chance of losing quality due to mutation.

The next genome of the next population is the super-child $S$. For every $i$ = 1,2, ...,$m$, where $m = |P|$, let genome $B_i$ be the genome of the previous generation with the best value $I_i$. $K_j^S = K_j^{B_i}$ for every $j = P_{i_1}, P_{i_2}, \ldots, P_{i_q}$, where $K^S$ is the vector $K$ of the super-child, $K^{B_i}$ is the vector (list) $K$ of $B_i$, $P_{i_x}$ is the $x$-th member of $P_i$, and $q = |P_i|$.

In other words, the super-child inherits the best genes of the population for every interval, from one, two, or more parents.

For the rest of the population, our GA uses roulette wheel selection to select two parents, then one-point crossover to create two children from the two parents. Each gene of each child created with the one-point crossover, has a chance to mutate. By default, this chance is $1/|V|$ (where $|V|$ is the number of vertices), but it can be changed. If a gene in $K$ mutates, then its value changes from false to true, or from true to false. For example: $K$ = (0,0,0,1,1). If the second and fourth values mutate, the new value of $K$ is: $K$ = (0,1,0,0,1).

The new generation has the same size of population as the previous one had. The old generation is completely replaced by the new one (created in these steps).

Naturally, after the next generation is ready, the algorithm updates the values $F$ and $I$ of each genome. The algorithm stops after a given time.

## 7 Example

In this section, we illustrate the work of our algorithm in a small example. There is a population of 3 solutions for a graph, and the partitions of the graph is known ({1,2,3,4},{5,6,7}). The order of the vertices of the graph is shown in Figure 4. The first generation is also known. We use black and grey color for the partitions (intervals).

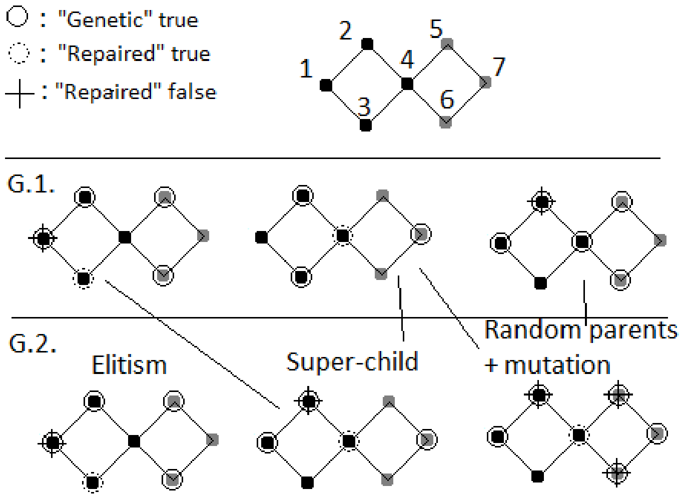

Figure 4

A simplified illustration of the algorithm

To repair the first solution, the algorithm has to complete the graph cover. The algorithm is greedy, but there is only one edge without cover vertices. The edge is between vertex 3 and 4. Selecting vertex 3 would result one unnecessary cover vertex, selecting vertex 4 would result one unnecessary cover vertex too, the algorithm selects vertex 3. Vertex 1 becomes unnecessary. Therefore, this cover has 4 vertices.

Solution 2 has edges without cover vertices too. Since the algorithm is greedy, vertex 4 is selected, because it will cover 2 uncovered edges. Solution 3 is a cover set, but it has one unnecessary vertex.

The size of the final cover set of each solution in the first generation is 4. But on the black interval solution 1 and 3 are better, and on the grey interval solution 2 is better.

The first solution of the new generation is the first best solution of the previous generation. Since they all had the same fitness value, solution 1 is now chosen as the elite.

The second solution is the super child, with the best intervals. It inherits the genes of the black interval of solution 1, and the genes of the grey interval of solution 2. Vertex 4 is selected because it will cover 3 uncovered edges. Vertex 2 is unnecessary. Therefore, this cover has 3 vertices.

The third solution inherits genes from two parents randomly, then mutates. It has one edge without cover vertices. The algorithm selects vertex 4, because it will result 3 unnecessary vertices. This cover has 3 vertices too.

We have tested our algorithm on large graphs as we describe it in the next section.

## 8 Experimental Results

We compare our algorithm with a fast greedy algorithm, a normal GA with simple error correction, "TVCA", "Darwin" and “NOVCA”.

TVCA of Ashay Dharwadker is a great algorithm for the vertex cover problem [25]. It always finds the cover set with size of a given number *k*, if there is any, but it may take a long time. TVCA first fills *C* (the cover set) with the members of *V* (vertices of the graph). The algorithm then finds the elements it can take out from *C*. But this is not the only difference between the two algorithms: TVCA is a deterministic algorithm. We used the original TVCA demonstration program, however, in this way, we could not manage to make tests in an automatic way. Thus, we tested manually some of our graphs (and not all of them).

Darwin is a programming environment developed for ETH Zurich [26]. Amongst many functions, Darwin is able to find vertex covers of a graph. This function is implemented by Gaston H. Gonnet, and it is based on another very good approach, the fixed-parameter algorithm [27]. Unfortunately, Darwin is designed for small graphs, so we could not test all our graphs with it. Actually, with Darwin, we could only test graphs where MGA, Darwin and NOVCA all found the same result almost immediately.

NOVCA is a deterministic polynomial time algorithm that we implemented based on the pseudo code described in its paper [28].

Our algorithm (**MGA**) is fast and accurate in almost any cases, except with near-complete graphs, where almost every node is connected to almost every node, like the Witzel Graph [25].

We have randomly generated 100 graphs with 500 vertices and 2500 edges, then tested GA, greedy, NOVCA, and MGA on each graph. For the results, see Figure 5 and Table 1. For 10 graphs, we tested TVCA as well, it found the vertex cover size of the MGA algorithm in 9 seconds on average.

Table 1
Test data for relatively small graphs: Vertices: 500, Edges: 2500 (average of 100 random graphs)

| Algorithm | 1 second | 2 seconds | 5 seconds | 9 seconds |
|---|---|---|---|---|
| TVCA | n/a | n/a | n/a | same as MGA* |
| Greedy | 360.24 | n/a | n/a | n/a |
| GA | 361.86 | 359.82 | 357.91 | n/a |
| NOVCA | 351.63 | n/a | n/a | n/a |
| **MGA** | 356.69 | 352.11 | 350.52 | n/a |

* TVCA was manually ran on 10 graphs and compared to MGA.

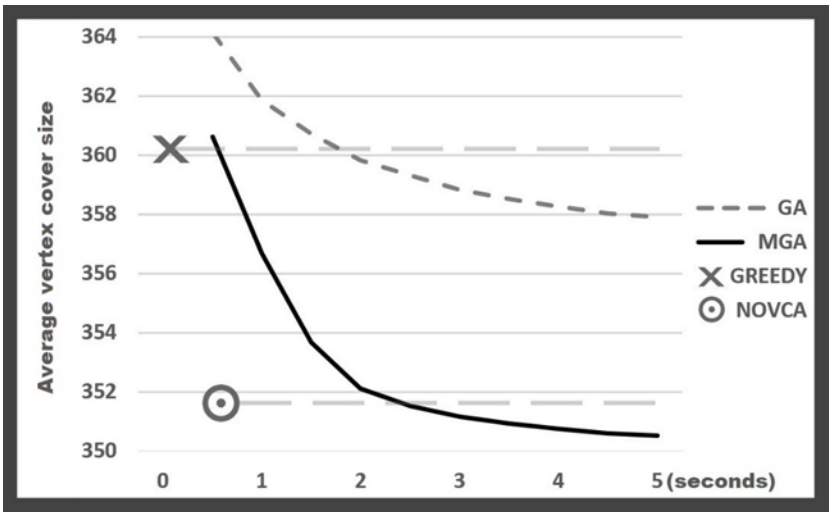

Figure 5
The average results of 100 random graphs with 500 vertices and 2500 edges

We also created an algorithm to generate “clustered graphs”, where we assigned each vertex to a cluster, and increased the chance of edges between vertices from the same cluster. The reason for this is that many graphs in nature are clustered, and also we expected that our algorithm would have an advantage with these. However, for more realistic results, the number of clusters is not the same as the number the partitioning algorithm in MGA uses.

We have repeated the above test with the same parameters on randomly generated clustered graphs. We tested TVCA too, and this is where we reached its limit, because 8 out of 10 cases, in 2 minutes TVCA was not able to find the same result that MGA found in 5 seconds. For detailed results, see Figure 6 and Table 2.

Table 2
Test data for clustered graphs: Vertices: 500, Edges: 2500 (average of 100 random graphs)

| Algorithm | 1 second | 2 seconds | 5 seconds | 120 seconds |
|---|---|---|---|---|
| TVCA | n/a | n/a | n/a | found MGA in 2 out of 10 cases |
| Greedy | 562.80 | n/a | n/a | n/a |
| GA | 364.40 | 362.11 | 360.16 | n/a |
| NOVCA | 356.93 | n/a | n/a | n/a |
| **MGA** | 360.66 | 354.33 | 353.04 | n/a |

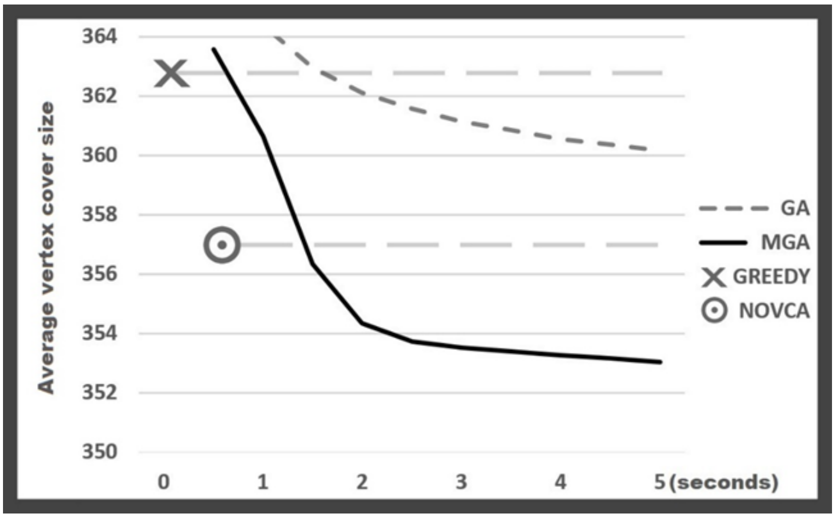

Figure 6

The average results of 100 random clustered graphs with 500 vertices and 2500 edges

We repeated the test on 100 random (not clustered) graphs with 2000 vertices and 10000 edges. For the results, see Figure 7 and Table 3. For 10 of the graph, we have tested TVCA as well, but it never found the result of MGA/NOVCA in 10 minutes.

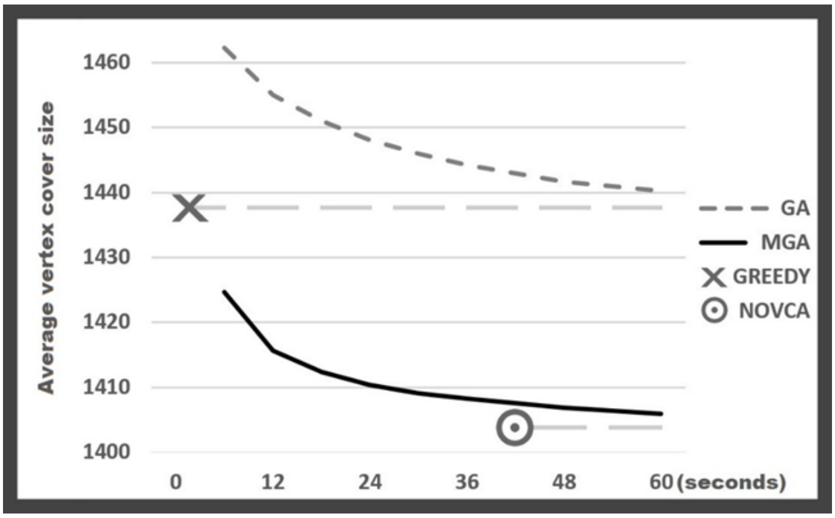

Figure 7

The average results of 100 random graphs with 2000 vertices and 10000 edges

Table 3
Test data for large random graphs: Vertices: 2000, Edges: 10000 (average of 100 random graphs)

| Algorithm | 1 second | 12 seconds | 42 seconds | 60 seconds | 600 sec |
|---|---|---|---|---|---|
| TVCA | n/a | n/a | n/a | n/a | n/a |
| Greedy | 1437.74 | n/a | n/a | n/a | n/a |
| GA | n/a | 1454.99 | 1442.91 | n/a | n/a |
| NOVCA | n/a | n/a | 1404.02 | n/a | n/a |
| **MGA** | n/a | 1415.51 | 1407.52 | 1405.95 | n/a |

Further, 100 random clustered graphs with 2000 vertices and 10000 edges were also tested. For the results, see Figure 8 and Table 4. For 10 of the graph, we tested TVCA as well, but it did not manage to find the result of MGA/NOVCA in 10 minutes.

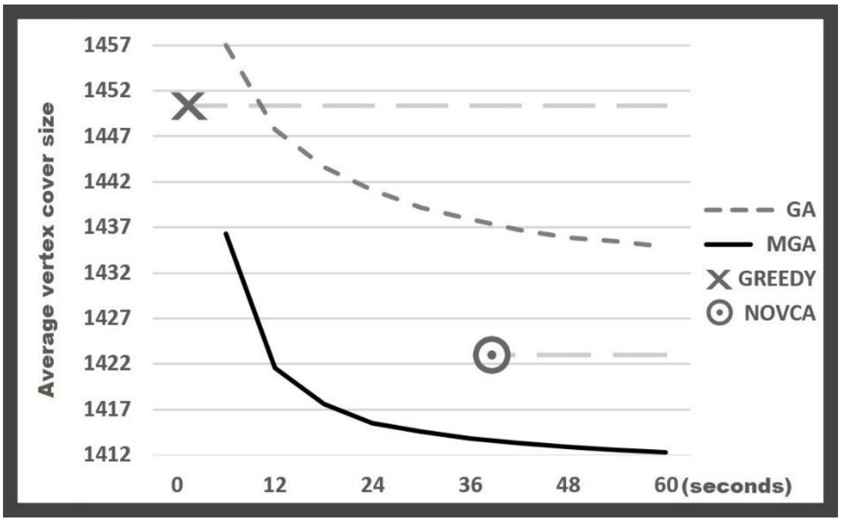

Figure 8
The average results of 100 random clustered graphs with 2000 vertices and 10000 edges

Table 4
Test data for large clustered graphs: Vertices: 2000, Edges: 10000 (average of 100 random graphs)

| Algorithm | 1 second | 12 seconds | 39 seconds | 60 seconds |
|---|---|---|---|---|
| TVCA | n/a | n/a | n/a | n/a |
| Greedy | 1450.57 | n/a | n/a | n/a |
| GA | n/a | 1447.74 | 1437.86 | 1434.89 |
| NOVCA | n/a | n/a | 1423.03 | n/a |
| **MGA** | n/a | 1421.52 | 1413.79 | 1412.27 |

Table 5
Test on Witzel Graph (Vertices: 450, Edges: 17827, M.V.C. Size: 420)

| Algorithm | 2 second | 43 seconds |
|---|---|---|
| TVCA | n/a | 420 |
| Greedy | 428 | n/a |
| GA | n/a | 426 |
| NOVCA | 424 | n/a |
| **MGA** | n/a | 426 |

On the Witzel Graph with 450 vertices and 17827 edges, TVCA finds the minimum vertex cover of 420 vertices in 43 seconds. MGA finds a vertex cover of 426 vertices in the same amount of time. NOVCA almost immediately finds a vertex cover of 424 vertices (see Table 5).

## Conclusions

Our modified GA yielded, minimally, the same solution as a greedy algorithm, but more often, our algorithm was better. The reason is, that the first generation has a blank genome and the greedy error correction works in a similar way, as a greedy algorithm. The modified GA is a big improvement for the GA. In case of large graphs, it converges much faster with the interval fitness method, which can be viewed as a problem reduction approach. At the same time, the algorithm focuses on the whole problem. With the error correction, almost all members of the population had a better fitness and the genetic variation is much better too. But the biggest advantage of the error correction, is obviously the fact that it mixes the fast and efficient greedy method, with the heuristic GA, taking advantage of both approaches. Distinguishing of genotypes and phenotypes, combined with the intervals also leads to good efficiency. By inheriting only, the backbone (genotype) of the covering sets, it is easy to create efficient hybrids of solutions.

In the case of a difficult graph, the basic GA is not as efficient as a simple greedy algorithm, not even with more time given. As Figure 7 shows, even after 60 seconds, the GA did not find the solution that greedy found almost immediately. With greedy error correction, it is possible that the genome of a solution is blank, meaning every element of set $K$ has a false value. Since the greedy error correction step itself is a good method for the problem as well, it is a good thing if the GA cooperates with the greedy method, by only selecting genes that improve the fitness. For this, the GA has to start with at least one blank genome in the population.

GAs tend to converge towards local optimum. To avoid this, it is worth to make experiments with multiple populations. Since members of different populations have no chance to mix their genetic information, multiple populations develop independently, and it is possible that each population converges towards different local optima. It also means that the different populations can use different processors. Another idea to solve the problem of local optimum is to restart the algorithm after a given time or generation. Obviously, the algorithm has to save the best solution, but the new population must not remember that genome, it has to start with random genomes again.

In the Experimental Results section, we see that the Greedy algorithms is the fastest, but it is also, the most inaccurate and it has no chance to improve with more time. TVCA [25] and Darwin [26] are very good in finding a small vertex cover, but is very slow and is not usable with large graphs. As Table 2 shows, TVCA needed 2 minutes to have a chance of finding a result MGA found in 5 seconds. NOVCA [28] is similar to Greedy, but it yields much better results at the cost of being a bit slower. Our algorithm, MGA, yields similar results to NOVCA, but has several advantages. In some graphs (Witzel, and fully random graphs) it may underperform a bit, but on other graphs (especially in clustered graphs) – MGA outperforms NOVCA. MGA also has the advantage of providing early results, and can run longer to find even smaller vertex covers. See Figure 8, where MGA found a better result in 12 seconds than NOVCA in 39, and then continued to improve.

Thus, it seems that the idea to using interval-valued fitness is a worthwhile consideration. We note that another type of combination of GA, with interval-values is found in [29].

## References

[1] Álmos Attila, Horváth Gábor, Várkonyiné Kóczy Annamária, Győri Sándor: Genetikus algoritmusok, Typotex Kiadó (2003) [Genetic Algorithms, in Hungarian]

[2] David E Goldberg: Genetic Algorithms in Search Optimization and Machine Learning., Addison Wesley (1989)

[3] Benedek Nagy, Elisa Valentina Moisi: Binary tomography on the triangular grid with 3 alternative directions - a genetic approach, ICPR 2014: 22nd International Conference on Pattern Recognition, Stockholm, Sweden, 1079-1084, IEEE Computer Society (2014)

[4] Shaohua Li, Jianxin Wang, Jianer Chen, Zhijian Wang: An Approximation Algorithm for Minimum Vertex Cover on General Graphs, Proceedings of the Third International Symposium on Electronic Commerce and Security Workshops (ISECS 10) Guangzhou, P. R. China, 29-31 July, pp. 249-252 (2010)

[5] P. Oliveto, J. He, X. Yao: Analysis of the (1+1)-EA for Finding Approximate Solutions to Vertex Cover Problems, IEEE Transactions on Evolutionary Computation, 13(5), 1006-1029, October (2009)

[6] Martin Pelikan, Rajiv Kalapala, Alexander K. Hartmann: Hybrid Evolutionary Algorithms on Minimum Vertex Cover for Random Graphs, GECCO '07 Proceedings of the 9th Annual Conference on Genetic and evolutionary computation ACM New York, NY, USA, pp. 547-554 (2007)

[7] Ali Karci, Ahmet Arslan: Bidirectional evolutionary heuristic for the minimum vertex-cover problem, Computers and Electrical Engineering, 29/1, 111-120, Elsevier (2003)

[8] Ketan Kotecha, Nilesh Gambhava: A Hybrid Genetic Algorithm for Minimum Vertex Cover Problem, In Prasad, B. (Ed.), The First Indian International Conference on Artificial Intelligence, 904-913 (2003)

[9] D. B. Fogel: Phenotypes, genotypes, and operators in evolutionary computation, in Proc. IEEE Int. Conf. Evolutionary Computation (ICEC'95) New York: IEEE Press, 193-198 (1995)

[10] Benedek Nagy: A general fuzzy logic using intervals, 6th International Symposium of Hungarian Researchers on Computational Intelligence, Budapest, Hungary, 613-624 (2005)

[11] Benedek Nagy: Reasoning by Intervals, Diagrams 2006, Fourth International Conference on the Theory and Application of Diagrams, Stanford, CA, USA, 145-147 (Lecture Notes in Computer Science - Lecture Notes in Artificial Intelligence, LNCS-LNAI 4045)

[12] Benedek Nagy: An interval-valued computing device (2005) CiE 2005, "Computability in Europe": New Computational Paradigms, Amsterdam, Netherlands, 166-177

[13] Benedek Nagy: Effective Computing by Interval-values, INES 2010, 14th IEEE International Conference on Intelligent Engineering Systems, Las Palmas of Gran Canaria, Spain, 91-96 (2010)

[14] Benedek Nagy, Sándor Vályi: Interval-valued computations and their connection with PSPACE, Theoretical Computer Science - TCS 394/3, 208-222 (2008)

[15] Benedek Nagy, Sándor Vályi: Prime factorization by interval-valued computing, Publicationes Mathematicae Debrecen 79/3-4 (2011) 539-551

[16] Benedek Nagy, Sándor Vályi: Computing discrete logarithm by interval-valued paradigm, (Benedikt Loewe, Glynn Winskel, eds.), Proceedings 8th Workshop on Developments in Computational Models - DCM 2012, Cambridge, England, Electronic Proceedings in Theoretical Computer Science - EPTCS 143 (2014) 76-86

[17] Benedek Nagy, Sándor Vályi: An Extension of Interval-Valued Computing Equivalent to Red-Green Turing Machines, MCU 2018: 8th International Conference on Machines, Computations, and Universality, LNCS 10881 (2018) 137-152

[18] R. M. Karp: Reducibility among combinatorial problems. In: (R. E. Miller and J. W. Thatcher, Eds.) Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., Plenum Press, New York, 1972, pp. 85-103

[19] F. McSherry: Spectral partitioning of random graphs, Foundations of Computer Science, 2001, Proceedings. 42nd IEEE Symposium on 8-11, 529-537, Oct. (2001)

[20] J. A. Hartigan, M. A. Wong: Algorithm AS 136: A K-Means Clustering Algorithm, Journal of the Royal Statistical Society, Series C (Applied Statistics) 28 (1), 100-108 (1979)

[21] G. M. Morris, D. S. Goodsel, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, A. J. Olson: Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function, Journal of Computational Chemistry 19(14), 1639-1662 (1998)

[22] Xin Yao: An empirical study of genetic operators in genetic algorithms, Microprocessing and Microprogramming, 38/1-5, 707-714, Elsevier (1993)

[23] V. Di Gesu, G. Lo Bosco, F. Millonzi, C. Valenti: A memetic algorithm for binary image reconstruction, Lecture Notes in Computer Science 4958, 384-395 (2008)

[24] Benedek Nagy, Elisa Valentina Moisi: Memetic algorithms for reconstruction of binary images on triangular grids with 3 and 6 projections, Applied Soft Computing 52, 549-565 (2017)

[25] Ashay Dharwadker: The Vertex Cover Algorithm, CreateSpace. http://www:dharwadker:org/vertex cover/ (2011)

[26] G. H. Gonnet, M. T. Hallett, C. Korostensky, L. Bernardin: Darwin v. 2.0: an interpreted computer language for the biosciences, Bioinformatics 16: 101-103, http://www.cbrg.ethz.ch (2000)

[27] R. Balasubramanian, M. R. Fellows, V. Raman: An improved fixed-parameter algorithm for vertex cover, Information Processing Letters 65, 163-168 (1998)

[28] Sanjaya Gajurel, Roger Bielefeld: A Simple NOVCA: Near Optimal Vertex Cover Algorithm, Procedia Computer Science, Volume 9, 747-753, Elsevier (2012)

[29] Lajos Zámbó, Benedek Nagy: Optimization of the Painting Problem by a Genetic Approach using Interval-values, CINTI 2011: 12th IEEE Int. Symp. Computational Intelligence and Informatics, 21-22 November, 2011 Budapest, Hungary, 127-132 (2011)