Noname manuscript No.
(will be inserted by the editor)

Graph Coloring Using Genetic Algorithm

Dániel Asztalos Balásy · Bence Fekete ·
Gábor Nagy · Tamás Tarisnyás

December 2020

Abstract Genetic Algorithm (GA) is one of the first population-based stochastic algorithms, inspired by evolutionary biology. These algorithms are considered as search methods used in computing fields to find exact or approximate solutions for optimization and search problems. Hereby this study investigates genetic algorithm solutions, using graph coloring techniques in order to solve several types of sudoku samples. The paper demonstrates results comparing the perfomance of GA algorithm with the performace of backtracking algorithm for a better overview of the search alorithms' performance.

Keywords Genetic algorithm · Graph coloring · Backtracking · Sudoku · Hypersudoku · Nonomino

Dániel Asztalos Balásy
E-mail: danielasztalosb@gmail.com

Bence Fekete
E-mail: feketebencetyping@gmail.com

Gábor Nagy
E-mail: ngabor827@gmail.com

Tamás Tarisnyás
E-mail: tarisnyastamaska@gmail.com

1 Introduction

The aim of this reasearch is to survey the effectiveness of genetic algorithms in the context of graph coloring. The problem we attempt to solve is the classical graph k-coloring problem: given a graph with n vertices, and a list of edges between these vertices, what is the minimum amount of labels we need to label the graph in a way that no adjacent vertices has the same color?

We implement a genetic algorithm to solve the general case of graph coloring, moreover we apply our solution to three variants of Sudoku puzzles: traditional Sudoku, Hypersudoku and Nonomino. In order to test the correctness and the performance our implementation, we conduct experiments with the Sudoku variants, measuring the running time of the algorithm and the state of the generated population during the experiments. These experiments are done with different GA related parameter setting, therefore we have the possibility to fine tune our solution for a class of puzzles and we can explore the most suitable setting for a certain puzzle. Also we are comparing the genetic algorithm's performance with a backtracing solution.

After a brief theoretical overview on graph coloring and genetic algorithms in general, we present the three selected problems, go over the details of the implementation of our genetic algorithm and, as a last step, we explain and summarize the experiment results in the case of the Sudoku, Hypersudoku and Nonomino puzzles.

2 Theoretical Overview

In this section, we are going to discuss the theoretical background of genetic algorithms and the methods of graph coloring.

2.1 Genetic Algorithms

By genetic algorithms (GA) we mean a class of search techniques that can be used to search for an optimum or an element with a given property. Genetic algorithms are in fact evolutionary algorithms, borrowing certain processes and techniques from evolutionary biology.

The main purpose of research on GA is to keep the balance between usage and analysis in the search of the optimal solution for many different situations [7].

GA differ from the traditional search procedure, the whole process starts with an initial pile of random solutions, which would be the **population**. Each individual in the population is going to be called a **chromosome**. Latter evolve through consecutive iterations, which are the so called **generations**. For each generation the chromosomes are defined with the help of some measures called **fitness**. In order to build further generations, new chromosomes, labeled **offsprings**, are generated by either merging two chromosomes from the ongoing generation by using a **crossover** and/or by reshaping a chromosome using a **mutation** operator [7].

A new generation is formed by selecting certain parents, given the value of the fitness and offspring, and refusing others, so that the size of the population would remain constant and unchanged [7].

To summarize, the main steps throughout a genetic algorithm operation are the following:

1. Initialization
2. Selection
3. Recombination - Mutation
4. Survivor selection

2.2 Graph Coloring

Partitioning a set of objects given certain rules is a fundamental process in mathematics.

A more simple pile of rules would invoke for each and every pair of objects, whether they could belong to the same class.

The theory of **graph coloring** precisely approaches the mentioned situation. More specifically said objects form a set of vertices $V(G)$ of a graph **G**. Two vertices are being joined by an edge in G whenever they are not allowed in the same class.

In order to be able to distinguish the certain classes we use a set of colors and the division into classes is given by a colring function. In the end each color class is going to form an independent set of vertices.

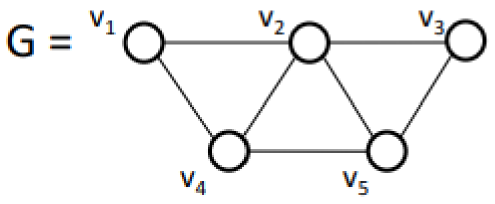
2.3 Graph coloring with backtracking

Backtracking is a general technique that can be used to determine an optimal solution (if it is possible all solutions) to certain computational problems, such as the already mentioned graph coloring. Backtracking algorithms work by systematically building up partial solutions into complete solutions. In some cases, however, the whole construction process can tell us, that there is no way of completing the current partial task, in order for the algorithm to be able to create the optimal solution. The algorithm simply backtracks for the purpose of finding suitable ways of calculating the current partial solution.

As an example, let's consider a simple backtracking algorithm for graph coloring.

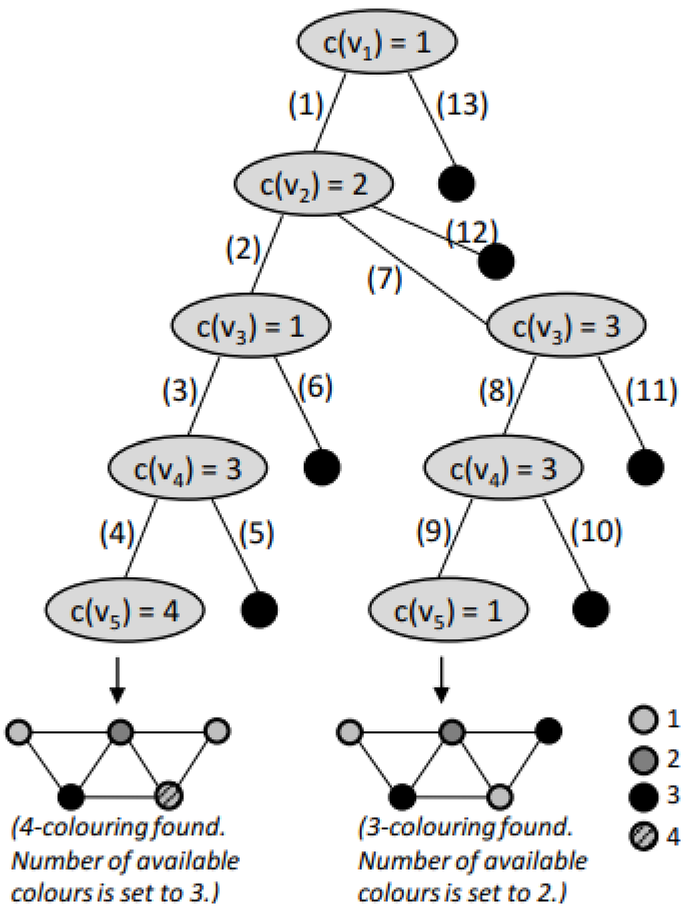
A $G = (V, E)$ graph is given. The vertices are ordered in a way that the vertex $v_i (1 \leq i \leq n)$ corresponds to the **i**th vertex of the ordering. We also need a **k** in order to denote the number of colors available. Initially we are going to set this to $k = \infty$. Afterwards the backtracking algorithm performs a series of for-and backward steps. Forward steps colour the vertices in the given order until the point when a certain vertex is identified as not available for reasonable coloring. Backward steps are gonna go back through the initially colored vertices and identify points where different color assignments to vertices can be made. Then we can resume to the forward steps. If a complete workable coloring is found, then **k** can be set to the number of colors used in this case minus 1 (k-1). Finishing latter step, the algorithm can continue with its work. The algorithm terminates when a backward step reaches the root vertex v_1 or when any other other criteria, such as maximum time limit is met.

Let us now approach the algorithm and how it exactly works in a more visual way.



Firstly we need a simple, small, undirected graph with a certain amount of vertices and edges. Here the vertices are considered according to the sequence $v_1, v_2, ..., v_5$.

Afterwards we are going to ilustrate the steps of the backtracking algorithm through a tree. Each grey node in said tree represents a decision, and grey leaf nodes gives an appropriate solution. To be precise the order in which the decision nodes are visited is shown by the numbers next to the corresponding edges in the tree, latter corresponding to a depth first search of the tree.


c(v1) = 1
(1)
(13)
c(v2) = 2
(2)
(7)
(12)
c(v3) = 1
c(v3) = 3
(3)
(6)
(8)
(11)
c(v4) = 3
c(v4) = 3
(4)
(5)
(9)
(10)
c(v5) = 4
c(v5) = 1
(4-colouring found. Number of available colours is set to 3.)
(3-colouring found. Number of available colours is set to 2.)
1
2
3
4

In Steps **1** to **4** all vertices are assigned to the feasible color of the lowest index. The solution of the process is a 4-coloring, as it appears in the figure. At this point, the inquiry would be to find a solution with fewer colors, so through latter, k is set to 3. In Step **5** the algorithm has performed a backtrack, a process, which we have previosuly mentioned, and now tries to assign vertex v_5 to a new colour. The procedure will not be succesful, because all of the colours (1, 2, 3) are unobtainable. The smaller, black node signifies that no other branch of vertex v_5 needs to be explored. Step **6** has performed the same backtrack event, but for vertex v_4. Color **3** has already been tried and colors **1** and **2** are not feasible, again we stop the procedure with the black node. Step **7** seems to be different than the previos ones. Here we can extract that colour **1** has already been tried, color **2** is not an option, but color **3** is quite alright. v_3 is assigned to color **3**, and further steps are carried out until a feasible **3**-coloring of **G** is achieved. At this point the number of available colors becomes **2**, and the algorithm goes on. According to the tree, when only two colors become available, no other color options remain obtainable for any of the previous vertices. The algorithm backtracks all the way to the root of the tree (v_1) and it terminates with the knowledge that no 2-coloring is available. Hence the **chromatic number** of the graph is **3**.

3 Selected graph coloring problems

A suprisingly wide range of problems can be represented as a graph and in many cases a vaild coloring of the corresponding graph can be decoded as the solution of the problem at hand.
It is possible to model social networking, scheduling, navigation, engineering, computer networking and computer architecture related problems with a graph. Graph coloring can be applied especially well to scheduling problems. For instance, timetable constructing. Let us consider the following situation: at a university we have a set of classes and a set of classrooms (regarded as classrooms in this situation) and two constraints. The first one is, that a person cannot be at two different class at a same time, the second one states that two classes cannot be held at the same classroom (or at the same timeslot). This timetabling problem can be converted into a graph coloring problem by representing each class with a vertex and adding an edge between two vertices if there is the possibility of breaking one of the above mentioned constraints. After building this graph, the task is to color it with maximum k number of colors, in this situation k is the number of available classrooms or timeslots [1].

The problems we attack with graph coloring is solving three different variants of Sudoku puzzles:

- Traditional Sudoku
- Hypersudoku
- Nonomino

After solving the general graph coloring problem, we apply the solution to a graph representing a 9x9 Sudoku, a 9x9 Hypersudoku and a 9x9 Nonomino puzzle.

3.1 Representing a Sudoku board as a graph

Before we begin to demonstrate a way to create the graph representation of a Sudoku board, it is important to emphasize the fact that the visual form of a graph that represents a Sudoku board can get really complex and hard to understand. For instance the graph of a simple 9x9 Sudoku consists of 81 nodes and 810 edges, therefore we are going to use a simpler 4x4 Sudoku, with only 16 cells. This version is solvable using only four numbers. The algorithm of transforming a Sudoku board into a graph is general, in a sense that the bigger 9x9 (or n×n) version can be processed in a similar way.

A Sudoku board can be described as a graph G in the following way:
Initialize the list of vertices V(G) and the list of edges E(G) of the graph G. For each cell of the Sudoku board:

1. Add a node to the list of vertices V(G).
2. Assign a color to the node. This is going to be a number in our representation. The black color and the zero value is reserved for empty cells (representing nodes that are not colored yet).

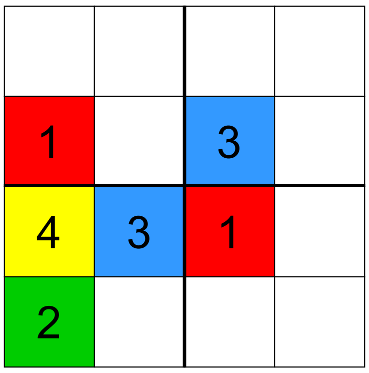


Fig. 1 Example 4x4 unsolved Sudoku board.

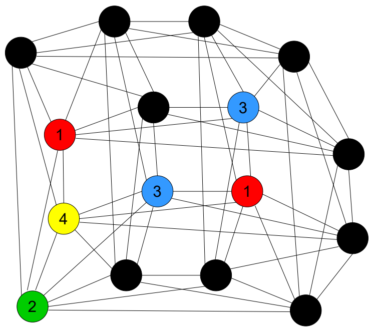


Fig. 2 The graph representation of the example Sudoku. The black colored vertices represent empty cells.

After populating the list of vertices V(G), we are going to iterate through that list and for each vertex X:

1. Add an edge X-R to the list of edges E(G) for every node R, that represents a cell on the Sudoku board that is on the same row as X.
2. Add an edge X-C to the list of edges E(G) for every node C, that represents a cell on the same column as X.
3. Add an edge X-S to the list of edges E(G) for every node S, that corresponds to a cell that is in the same cell group (a smaller 3x3 area in the case of a 3x3 Sudoku).

The final step is to delete the duplicate edges from E(G). The result is a single graph with only one edge between any two nodes.

Now we can see that the problem of solving the Sudoku puzzle is transformed into the **graph k-coloring problem**: find a way to color a graph with respect to the constraints that no two connected nodes can have the same color and the number of different colors we can use is the square root of the number of nodes ($k = \sqrt{numberofnodes}$).

3.2 Sudoku graph coloring

It the case of the simple Sudoku graph coloring each vertex represents a cell on the Sudoku puzzle and an edge between two vertices means that the cells are on the same row, same column or in the same 3x3 grid. As we mentioned before, a visual representation of such a graph can be very complex and hard to understand, thus for the figure we are using a 4x4 Sudoku board with four 2x2 cell group. The corresponding graph has 16 nodes and 56 edges.

3	4	2	1
1	2	3	4
4	3	1	2
2	1	4	3

Fig. 3 A solution to the previous 4x4 Sudoku (see fig. 1).

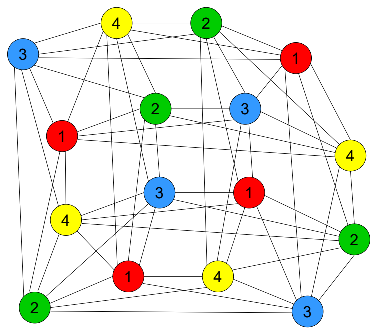


Fig. 4 The colored graph with representing the solution.

4	2	1	8	7	3	5	9	6
8	7	5	6	9	2	3	4	1
6	9	3	1	4	5	7	8	2
3	8	4	2	5	9	6	1	7
1	5	7	3	8	6	4	2	9
9	6	2	7	1	4	8	5	3
5	1	9	4	6	7	2	3	8
7	3	8	5	2	1	9	6	4
2	4	6	9	3	8	1	7	5

Fig. 5 Example of a solved 9x9 Hypersudoku.

3.3 Hypersudoku graph coloring

The Hypersudoku is an extension of the original 9x9 grid, it defines **four additional interior 3×3 regions in which the numbers 1–9 must appear exactly once**. The graph representation of such a puzzle is the same as the graph of the original 9x9 Sudoku board, but with an additional set of edges between the vertices in the four additional cell groups. This can be regarded as a set of extra constraints against the solution. In this case, a **9x9 Hypersudoku's graph** consists of **81 nodes** and **874 edges**. Hypersudokus exist only in 9x9 forms and it is virtually impossible to draw a Hypersudoku graph in an easy to understand, straightforward fashion in 2D, therefore we are not going to visualize the graph in this case.

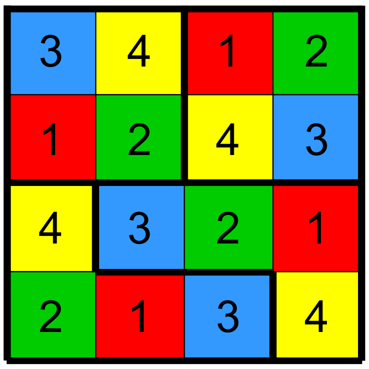


Fig. 6 An example 4-omino. The top two cell groups have a regular shape, the bottom two are transformed into an "L" shape.

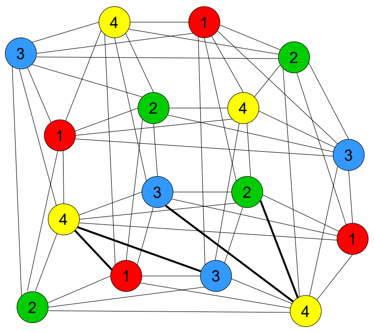


Fig. 7 The colored graph of the example 4-omino. The structural difference (compared to a regular 4x4 Sudoku) is indicated by the four thicker edges on the bottom.

3.4 Nonomino graph coloring

There is another interesting variant of the Sudoku puzzle which is called Nonomino. The representation is similar to the Sudoku, with the distinction that the 3x3 grid is transformed into a polygon made of 9 cells, connected edge to edge. In a more general sense, these puzzles are called polyominos and the Nonomino is in fact a 9-onimo (or a polyonimo of order 9) [2]. The number of nodes and edges in a Nonomino graph are the same as in the case of a simple Sudoku (16 nodes and 56 edges in the case of a 4-omino board), the only thing that is different is the structure of the graph, because the edges are rearranged.

4 Solving sudoku using a Genetic Algorithm and Graph coloring

In this section we will present in detail all the building blocks of the genetic algorithm used in our experiments and the algorithm itself as well. The algorithm we implemented was partially inspired by the work of Mantere, Timo and Janne Koljonen [5] who used g enetic and two other types of evolutionary algorithms in order to solve sudokus.

4.1 Genetic representation

The sudoku board can be seen as a partially-filled graph. The little squares are the nodes and constraints are the edges of this graph. Each node is connected to the nodes in its row, column and smaller square.

For representing a graph a custom **Graph** object was used. This object stores its edges and vertices in separate lists, called **edges** and **nodes**. The **Graph** object also has an attribute that stores the **fitness** value of the respective object. Each of the nodes of this graph object has a **color** attribute that is encoded as a positive integer, greater than 0. To represent the sudoku grid using this graph object one additional node attribute was used, **fixed**. This variable stores a boolean value, when its value is true, it means that the color of that specific node cannot be changed throughout the genetic algorithm, which is useful when storing the pre-filled values of a sudoku board.

4.2 Fitness function

By default, a graph has the fitness value of 0 when there are no adjacent nodes in the graph with the same color. A fitness value less than 0 means that there exists at least one case where two neighboring nodes have the same color values.

The proposed fitness function initializes each graph with the fitness value 0. Then, for every graph the algorithm iterates through its edges. For each edge where the two nodes on each side have the same color the algorithm deduces 1 from the fitness value of the examined graph. Algorithm 1 presents this procedure in detail.

4.3 Initialization

Before initializing the population, a prototype graph is built. This graph contains all the nodes and edges specific to the sudoku board and the pre-filled values also. This graph is then cloned as many times as the size of the population.

For each individual of this population colors are assigned to the empty fields of the sudoku graphs. This is done by iterating through the rows of a sudoku graph, during which the colors that are not used in the row are determined and

Algorithm 1 Calculating the fitness values of the individuals

```
procedure CalculateFitness(population)    ▷ population - list of all the individuals of the population
    for individual in population do
        if fitness of individual is computed then
            continue    ▷ if the fitness of a individual was already calculated then do not calculate it again
        end if
        $fitness \leftarrow 0$
        for edge in edges of individual do
            if the nodes of the edge have the same color then
                $fitness \leftarrow fitness - 1$
            end if
        end for
        Fitness of individual $\leftarrow fitness$
    end for
    return population
end procedure
```

randomly assigned to empty nodes. This way the algorithm will find solutions faster because every row of the sudoku graphs is a permutation of the numbers from 1 to the number of rows and this excludes the possibility that two nodes in the same row will have the same color. This procedure is presented in Algorithm 2.

Algorithm 2 Initialization of the population

```
procedure CreatePopulation($N$, sudoku_list, n_colors)
    Create prototype board based on prefilled values from sudoku_list.
    Clone the prototype board $N$ times and store the clones in the list. copies
    for graph in copies do
        Set fitness value of graph to 'None'
        for $i = 1, 2, \ldots, n_colors$ do    ▷ iterate through the rows of the sudoku graph
            Find colors that do not appear in $i$-th row.
            Fill empty nodes using the colors found during the previous step.
        end for
    end for
    return copies
end procedure
```

4.4 Parent selection

For crossover, parents are selected using the tournament selection algorithm. Initially it picks a random individual from the set of possible parents and uses it as the "current best" candidate. After that, it sequentially picks k candidates and examines for each one whether they are better than the "current best". If an individual is better, then they become the "current best". For the comparison of individuals the fitness values are used. Algorithm 3 presents this function in details.

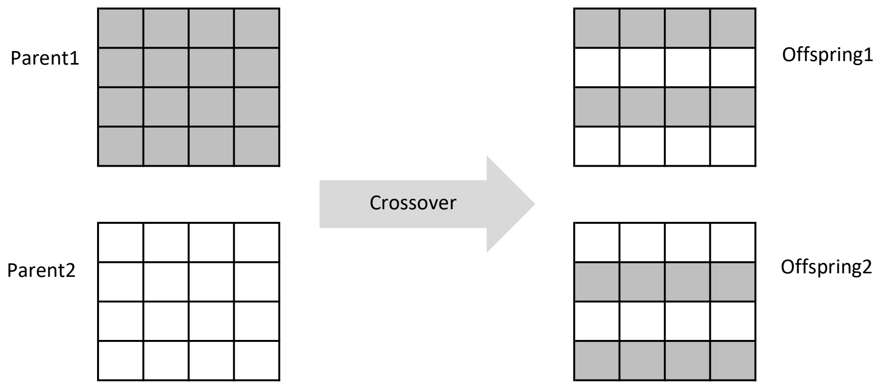


Fig. 8 Crossover on a 4x4 sudoku board

For larger k values the algorithm finds the best individuals of the population, for smaller values "weaker" individuals have a chance to be picked as well.

Algorithm 3 Parent selection

```
procedure TournamentSelection($population, k, possible$)
        ▷ $population$ - a list containing all the individuals from the population
        ▷ $k$ - selection pressure
        ▷ $possible$ - a list containing the indexes of individuals of the population that can be chosen as parents
    $best\_idx \leftarrow$ Random index from $possible$
    for $i = 1, 2, \ldots, k-1$ do
        $rnd\_idx \leftarrow$ Random index from $possible$
        if $population[rnd\_idx].fitness > population[best\_idx].fitness$ then
            $best\_idx \leftarrow rnd\_idx$
        end if
    end for
    return $best\_idx$
end procedure
```

4.5 Crossover

As a first step a random number is picked and compared to the given p_c value (p_c = probability of crossover). If this number is bigger than p_c, then the clones of the two parents will be the two offsprings with no modifications. Otherwise, a variant of the uniform crossover is applied, where the offsprings will get one row of the sudoku board from one parent, the next from the other and so on. The pseudocode version of this algorithm can be seen in Algorithm 4.

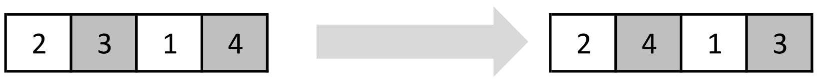


Fig. 9 Swap mutation applied to a row of a 4x4 sudoku grid.

For example, on a 9x9 sudoku board the *offspring1* gets the colors of the first nine nodes from *parent1*, the second nine color values from *parent2*, the third from *parent1* again and this process is repeated until all nodes of the offspring have colors. *Offspring2* gets their colors from the opposite parent in each round, in this example the first nine values from *parent2*, second from *parent1*, third from *parent2* and so on. This example is demonstrated on figure 8 where a 4x4 sudoku board was used.

Algorithm 4 Crossover

```
procedure UniformCrossover($parent1, parent2, p\_crossover$)
    $offspring1 \leftarrow$ Copy of $parent1$
    $offspring2 \leftarrow$ Copy of $parent2$
    if Random number from interval $[0, 1] > p\_crossover$ then
        return ($offspring1, offspring2$)
    end if
    Switch every second row between the two offsprings.
    return ($offspring1, offspring2$)
end procedure
```

In the implementation of this algorithm in order to speed up this process, the first offspring will be the clone of the first parent and the second offspring the clone of the second parent. After this, only every second row is swapped between them.

4.6 Mutation

The mutation algorithm applied is a variant of swap mutation. It iterates through each node of a graph received as input parameter. At each node a random node is picked from the same row of the sudoku graph. If any of the nodes has its **fixed** attribute set to *true* the algorithm continues with the next node. Otherwise, a random number is picked and if it's less than a predefined p_m value (p_m = probability of mutation) then the colors of those two nodes will be swapped. Figure 9 presents how a randomly chosen pair of nodes is swapped in a row of a 4x4 sudoku. The algorithm is described in Algorithm 5.

Algorithm 5 Mutation

```
procedure SwapMutation($population, p_m, max_color$)
    for individual in population do
        for row in individual do
            for node in row do
                rn_node ← Random node from same row as node
                if node and rn_node are not 'fixed' then
                    Swap the color values of node and rn_node
                end if
            end for
        end for
    end for
end procedure
```

4.7 Survivor selection

After the new offspring is added to the population a number of, equal to the initial size of the population, survivors are selected. The survivor selection method applied was the elitist one, i. e. members with the best fitness scores will be part of the next generation. This procedure is described in Algorithm 6.

Algorithm 6 Survivor selection

```
procedure ElitistSelection(population, $N$)
    new_pop ← Empty list
    while Size of new_pop $< N$ do
        best ← Select best individual from population
        Append best to new_pop
        Remove best from population
    end while
    return new_pop
end procedure
```

4.8 The Genetic Algorithm

The first step of the algorithm, presented in Algorithm 7, initializes the following parameters:

- **grid_size**: the number of rows in the sudoku grid (e. g. in case of a 9x9 grid, it's 9).
- **pop_size**: size of the population.
- **p_mutation**: the probability that a mutation happens to a node of the sudoku graph.
- **p_crossover**: the probability that crossover is applied to a pair of parents.
- **n_mating_pool**: the size of the mating pool or the number of parent-pairs selected.

- **s_pressure**: the selection pressure used in the tournament selection algorithm.
- **max_gener**: the number of generations where the algorithm stops running if no solutions were found until that generation. The role of this variable is to stop the algorithm from running indefinitely when it is stuck in a local optimum value.

After that the population of size *pop_size* is randomly initialized using the Algorithm 2. For this initial population the fitness values are then calculated for each individual using Algorithm 1. The algorithm then iterates through the following steps until a individual is found that has a fitness value of 0, i. e. a sulution is found:

- Select *n_mating_pool* pairs of parents for crossover using Algorithm 3 with selection pressure *s_pressure*.
- Apply Algorithm 4 to each of the selected pairs with possibility *p_crossover*.
- Add the offspring to the population.
- Apply Algorithm 5 to each individual in the population with probability *p_mutation*.
- Recalculate fitness of the population.
- Select *pop_size* best individuals from population using Algorithm 6.
- Substitute the population with the selected individuals.

Algorithm 7 Genetic Algorithm for solving Sudokus

```
Initialize algorithm parameters
Initialize population
Calculate fitness of population
while No individual with fitness 0 exists in population do
    Select pairs of parents for crossover
    Apply crossover to the selected parents
    Add new individuals to population
    Apply mutation to the population
    Recalculate fitness values of the individuals
    Select survivors
    Substitute current population with the survivors
end while
```

4.9 Hypersudoku and Nonomino

In the case of solving a Hypersudoku and a Nonomino the algorithm presented in this section was applied. The only modification made to it was in the initialization function presented in sectition 4.3. The prototype of the Hypersudoku graph object contained more edges than a plain sudoku and in the case of the Nonomino graphs some of the edges differed from the sudoku. With these modifications the genetic algorithm did not need to be changed, it ran just like in the case of a simple sudoku graph.

5 Numerical experiments

In order to measure the performance of the implemented genetic algorithm, we tested it on the selected Sudoku puzzle variants:

- Traditional 9x9 Sudoku
- Standard 9x9 Hypersudoku with four extra cell groups
- Nonomino (polyonimo of order 9)

5.1 Problem generation method

The first step of the experiments was creating problems from the three selected Sudoku classes. In each case, we aimed to create multiple random puzzles from the Sudoku variants.

5.1.1 Generating unsolved Sudokus

To be able to solve Sudoku puzzles with different difficulty levels, we had to implement a short algorithm, which was capable of creating random Sudokus with random missing values.

A normal Sudoku puzzle is practically made out of a 9x9 board or matrix. This means that it has 9 rows and 9 columns. What we did was that every time that we wanted to generate a new puzzle, we initialised two empty arrays, which were the indexes for the rows and columns of the matrix. Afterwards we randomized the indexes, so that the values associated to the certain indexes would be placed to different positions every single time. We also needed a third array, to which we added the explicit values of the board, but randomly again. With the help of the previous row and column arrays we filled the Sudoku puzzle with values.

The only problem was that our Sudoku was now solved. So what we did is we removed a certain amount of values (manually) and we placed zeros (0) instead of these. After said actions, the Sudoku board was ready to be solved.

5.1.2 Generating unsolved Hypersudokus

In order to generate unsolved hypersudoku puzzles we used our backtracking solving algorithm with a modification to create fully solved unique hypersudokus. Similarly to the traditional sudoku, we initialised two empty arrays and we randomized the array values which were the indexes of the sudoku cells, so this way we will not get the same unsolved sudoku. After having 5 unique, solved sudokus, we deleted randomly some of the values manually. In the case of hypersudoku, we created 5 different unsolved hypersudokus with 5 different difficulty levels. While the easier hypersudokus contained 10-15 missing cells, the hardest sudoku contained 30 missing cells.

5.1.3 Creating unsolved Nonomino puzzles

We failed to find a way to generate instances of this Sudoku variant from scratch, therefore we created a workaround with a reversed approach which started with a set of solved Nonominos. We simply deleted some randomly selected cell values and replaced them with empty cells.

In the case of Nonomino boards, we used five different Nonomino puzzles from a public collection [3]. We started the whole testing process by creating graphs from the solutions of the five selected puzzles [4]. After creating the input data as a list of $(vertexcolor, groupnumber)$ pairs, we built the corresponding graph of each of the five puzzles by adding the standard row and column related edges to the graphs and iterating over the list, adding the cell group related connections too.

The i^{th} element of the pair list can be interpreted in the following way: the vertex color is the color value of the i^{th} node in the solution graph, the group number is used to determine the edges inside a given deformed 3x3 cell group (there are 9 cell groups in a Nonomino, just like in a Sudoku).

The next step was the process of creating unsolved Nonominos from the solutions. From each puzzle, we deleted 10, 15, 20, 25 and 30 cell values randomly, thus creating a total of 25 unsolved Nonominos (five from each of the five solutions).

5.2 Testing method

In order to test the algorithm with different parameters, we created a list of values for **population size**, **probabililty of mutation**, **probability of crossover** and **selection pressure**. After that - for a given puzzle - we ran the algorithm **10 times with every possible parameter combination**, measuring the **average number of generations** needed to find a solution, the **standard deviation** of that generation number, the **average running time** of an experiment and the **standard deviation** of the 10 experiments' running time.

5.3 Test results

5.3.1 Traditional Sudoku

We have measured and compared with each other, the execution time of solving a traditional Sudoku Puzzle, with the methods of Backtracking and Genetic Algorithm.

In the case of the Traditional Sudoku we took five different boards, the difficulty level being different in every case. The difficulty level can be approximated, by simply observing the amount of zeros on the board. It can be defined as the more zeros in the matrix, the more difficult the Sudoku puzzle is.

In the case of the Genetic Algorithm we need to consider the parameters, the value of which differ in every single case. The parameters are the **Population size** (Pop_n), the **Mutation probability** (P_m), the **Crossover probability** (P_c) and the **Selection pressure** (SP). Throughout the experiments on every GA side, a certain value combination of the latter parameters gave away the result, which we are going to mention in the upcoming section, the experiments on the Traditional Sudoku puzzle with the help of Backtracking and GAs.

Experiments:

1. The first example only had five zeros, this one was the easiest case. The backtracking time was so short that it returned **0.0 seconds**, while the best time of the Genetic Algorithm, implemented by us, measured **0.061 seconds** ($Pop_n : 10, P_m : 0.07, P_c : 0.9, SP : 5$).

2. In the second experiment we have raised the number of zeros by 10, the total number of empty values being 15 at this moment. The backtracking time was still too short to be measured, so again we have received **0.0 seconds**, the GA rose to `0.271 seconds` ($Pop_n : 30, P_m : 0.07, P_c : 0.9, SP : 10$).

3. In the third case the number of zeros had increased to 25, and we have finally received a more concrete time from the backtracking algorithm. It was about **0.000994 seconds**, which was still insignificant, compared to the **1.74 seconds** of the GA algorithm ($Pop_n : 10, P_m : 0.07, P_c : 1.0, SP : 10$).

4. The fourth experiment was not too much different than the previous one, the number of difficulty, so to say, only increased by 3, which meant that the board had 28 zeros. The backtracking time had increased to **0.000997 seconds**. The GA measured **1.90 seconds**, which is a mentionable result, considering that the difference was only three zeros ($Pop_n : 10, P_m : 0.07, P_c : 1.0, SP : 10$).

5. The last experiment was for 30 missing values. The GA gained nearly a **whole second** compared to the third example, while the time of the backtracking algorithm had not changed almost at all, during the last three experiments ($Pop_n : 10, P_m : 0.07, P_c : 0.9, SP : 10$).

We can draw the conclusion that it was a little bit unexpected that the Genetic Algorithm performed so badly compared to the Backtracking algorithm. We increased the number of values missing by ten in a few cases, but the reason why we stopped doing that was that at values higher than 30 the

Genetic Algorithm took a huge amount of time to go through all the parameter combinations, so we decided that we would increase this number by only 2 or 3 in each case. This seemed to work, but even this way the GA did not work quite as expected.

Table 1 Experiment results of a Traditional Sudoku with 15 missing values.

Parameters				Number of generations		Execution time (seconds)	
Population size	P(mutation)	P(crossover)	Selection pressure	Mean	Std. dev.	Mean	Std. dev.
10	0.03	0.9	5	9.1	8.24	0.305	0.222
10	0.03	0.9	10	10.7	10.183	0.375	0.304
10	0.03	1.0	5	11.333	12.726	0.382	0.366
10	0.03	1.0	10	11.9	13.239	0.388	0.367
10	0.07	0.9	5	11.06	12.103	0.367	0.336
10	0.07	0.9	10	10.066	11.453	0.345	0.317
10	0.07	1.0	5	8.928	10.973	0.315	0.303
10	0.07	1.0	10	8.312	10.428	0.297	0.288
20	0.03	0.9	5	8.111	10.096	0.297	0.280
20	0.03	0.9	10	7.68	9.889	0.294	0.280
20	0.03	1.0	5	7.209	9.588	0.284	0.270
20	0.03	1.0	10	6.99	9.444	0.281	0.266
20	0.07	0.9	5	6.723	9.181	0.278	0.258
20	0.07	0.9	10	6.557	8.902	0.281	0.252
20	0.07	1.0	5	6.473	8.650	0.283	0.246
20	0.07	1.0	10	6.281	8.470	0.279	0.241
30	0.03	0.9	5	6.088	8.308	0.278	0.235
30	0.03	0.9	10	5.833	8.160	0.275	0.229
30	0.03	1.0	5	5.657	8.007	0.274	0.225
30	0.03	1.0	10	5.475	7.862	0.274	0.220
30	0.07	0.9	5	5.314	7.761	0.273	0.218
30	0.07	0.9	10	5.154	7.624	0.271	0.214
30	0.07	1.0	5	5.047	7.522	0.273	0.213
30	0.07	1.0	10	4.979	7.391	0.275	0.210

Table 2 Experiment results of a Traditional Sudoku with 30 missing values.

Parameters				Number of generations		Execution time (seconds)	
Population size	P(mutation)	P(crossover)	Selection pressure	Mean	Std. dev.	Mean	Std. dev.
10	0.03	0.9	5	89.2	35.980	2.835	1.029
10	0.03	0.9	10	95.85	29.072	2.987	0.843
10	0.03	1.0	5	92.4	30.807	2.887	0.886
10	0.03	1.0	10	87.675	33.027	2.767	0.974
10	0.07	0.9	5	85.48	32.496	2.779	0.986
10	0.07	0.9	10	83.2	31.737	2.739	0.982
10	0.07	1.0	5	82.971	33.586	2.766	1.069
10	0.07	1.0	10	83.362	34.857	2.798	1.140
20	0.03	0.9	5	81.944	34.377	2.804	1.142
20	0.03	0.9	10	80.89	33.727	2.804	1.131
20	0.03	1.0	5	79.354	32.737	2.779	1.096
20	0.03	1.0	10	79.225	32.846	2.801	1.124
20	0.07	0.9	5	80.907	35.510	2.953	1.408
20	0.07	0.9	10	80.814	35.324	2.994	1.411
20	0.07	1.0	5	81.1	34.918	3.045	1.407
20	0.07	1.0	10	81.318	35.120	3.078	1.414
30	0.03	0.9	5	80.576	34.397	3.071	1.380
30	0.03	0.9	10	79.794	34.155	3.062	1.364
30	0.03	1.0	5	79.989	34.419	3.088	1.379
30	0.03	1.0	10	80.21	34.598	3.113	1.386
30	0.07	0.9	5	81.747	36.353	3.256	1.628
30	0.07	0.9	10	82.040	36.237	3.351	1.697
30	0.07	1.0	5	85.282	40.844	3.615	2.210
30	0.07	1.0	10	84.920	40.345	3.658	2.194

5.3.2 Hypersudoku

Similarly to the experimenting of traditional sudoku solving, we ran both the genetic algorithm and the backtracking algorithm on 5 different unsolved hypersudokus, each at a different difficulty level. More missing values in the sudoku puzzle means more difficult problem solving for each algorithm. As an experiment we ran the algorithms 10 times on each sudoku, then we calculated the mean time for each experiment. For the Genetic Algorithm experiments we firstly observed the experiment results with every parameter configuration, then we selected the best performing parameter combination.

Experiments:

1. The first and easiest sudoku contained **10 missing values**. The backtracking algorithm generally solved the problem instantly, **0.0 seconds**. The genetic algorithm's best mean time for solving this hyper sudoku was **0,31 seconds**. In this experiment the algorithm could solve the problem around the **9th generation** on average. The genetic algorithm required the following parameter configuration for the quickest run-time. $(Pop_n : 20, P_m : 0.03, P_c : 0.9, SP : 10)$.

2. The second sudoku contained **15 missing values**. The backtracking algorithm performed well again, and solved the problems instantly. Although the difference between the first and second sudokus' number of missing values was small, the genetic algorithm took almost twice as much time to solve the problem, compared to the first case, taking **0,58 second**. This time the genetic algorithm finds the solution around the **19th generation**. ($Pop_n : 10, P_m : 0.03, P_c : 0.9, SP : 10$).
3. For the third sudoku we deleted **25 values** from the generated hyper sudoku.This time the backtracking algorithm solved the hyper sudoku in **less than 0.0001** second, and the genetic algorithm's best mean time was **1.74 seconds**. The genetic algorithm solved the sudoku around the **62nd generation**, which means, adding 10 more missing values to the sudoku required 3 times more generations to solve the sudoku. ($Pop_n : 10, P_m : 0.03, P_c : 0.9, SP : 10$).
4. The fourth hyper sudoku contained **30 missing cells**. The backtracking algorithms' best run-time did not change too much compared to the previous sudokus'. The genetic algorithm took around half a second more time to solve the sudoku, with **2.35 second** best time. At this sudoku, the genetic algorithm required no more than **69 generations** to solve the problem. ($Pop_n : 10, P_m : 0.03, P_c : 0.9, SP : 5$).
5. For the fifth sudoku we checked if the genetic algorithm can solve the hyper sudoku with more than **35 missing values**, but it appeared that the algorithm can rarely solve the problem in a limited amount of time. So instead, as the fifth experimental hyper sudoku we chose to have again 30 missing values, but with a different sudoku solution. The backtracking runtime did not change compared to the previous experiment, and the genetic algorithm took **2.50 seconds** to solve the sudoku problem, and it was solved around the **80th generation**.($Pop_n : 10, P_m : 0.03, P_c : 0.9, SP : 5$).

Throughout observing the hyper sudoku solving experiments, we came to the conclusion that the backtracking algorithm is a far better performing method to solve a hyper sudoku. Generally the genetic algorithm needs more than 60 generations to find a solution for a hyper sudoku with 30 missing values.

Table 3 Experiment results of a Hyper sudoku with 15 missing values.

Parameters				Number of generations		Execution time (seconds)	
Population size	P(mutation)	P(crossover)	Selection pressure	Mean	Std. dev.	Mean	Std. dev.
10	0.03	0.9	5	11.2	9.624	0.346	0.254
10	0.03	0.9	10	13.3	20.481	0.419	0.546
10	0.03	1.0	5	13.43	17.250	0.420	0.467
10	0.03	1.0	10	12.675	15.612	0.390	0.420
10	0.07	0.9	5	11.82	14.107	0.366	0.380
10	0.07	0.9	10	11.183	13.251	0.349	0.356
10	0.07	1.0	5	11.014	13.352	0.354	0.341
10	0.07	1.0	10	10.262	12.051	0.334	0.327
20	0.03	0.9	5	9.533	11.636	0.319	0.313
20	0.03	0.9	10	9.23	11.290	0.316	0.304
20	0.03	1.0	5	9.218	10.912	0.326	0.298
20	0.03	1.0	10	9.241	10.866	0.334	0.301
20	0.07	0.9	5	9.0	10.517	0.333	0.291
20	0.07	0.9	10	8.814	10.262	0.332	0.284
20	0.07	1.0	5	8.753	9.991	0.334	0.278
20	0.07	1.0	10	8.518	9.759	0.332	0.271
30	0.03	0.9	5	8.323	9.606	0.333	0.269
30	0.03	0.9	10	8.216	9.514	0.335	0.267
30	0.03	1.0	5	8.210	9.735	0.339	0.278
30	0.03	1.0	10	8.075	9.612	0.339	0.275
30	0.07	0.9	5	8.080	9.417	0.347	0.273
30	0.07	0.9	10	7.931	9.250	0.347	0.268
30	0.07	1.0	5	7.773	9.117	0.346	0.264
30	0.07	1.0	10	7.666	8.971	0.347	0.260

Table 4 Experiment results of a Hyper Sudoku with 30 missing values.

Parameters				Number of generations		Execution time (seconds)	
Population size	P(mutation)	P(crossover)	Selection pressure	Mean	Std. dev.	Mean	Std. dev.
10	0.03	0.9	5	79.4	29.144	2.502	0.843
10	0.03	0.9	10	82.55	40.375	2.663	1.266
10	0.03	1.0	5	81.76	36.908	2.625	1.161
10	0.03	1.0	10	96.1	76.143	3.070	2.408
10	0.07	0.9	5	90.48	70.140	2.952	2.209
10	0.07	0.9	10	98.966	53.226	1.687	0.863
10	0.07	1.0	5	86.316	65.407	2.861	2.048
10	0.07	1.0	10	85.357	62.635	2.869	1.975
20	0.03	0.9	5	84.962	56.787	2.895	2.895
20	0.03	0.9	10	82.86	55.047	2.894	1.765
20	0.03	1.0	5	81.809	53.338	2.893	1.715
20	0.03	1.0	10	80.925	51.32	2.890	1.652
20	0.07	0.9	5	85.215	56.584	3.169	2.171
20	0.07	0.9	10	84.378	55.131	3.194	2.124
20	0.07	1.0	5	87.006	55.542	3.383	2.245
20	0.07	1.0	10	86.625	54.173	3.418	2.195
30	0.03	0.9	5	85.758	52.874	3.416	2.138
30	0.03	0.9	10	83.994	52.095	3.376	2.098
30	0.03	1.0	5	83.584	51.060	3.391	2.059
30	0.03	1.0	10	83.175	50.209	3.399	2.027
30	0.07	0.9	5	92.404	69.329	4.033	3.835
30	0.07	0.9	10	92.995	68.970	4.142	3.866
30	0.07	1.0	5	97.691	75.421	4.477	4.357
30	0.07	1.0	10	97.470	74.374	4.529	4.304

5.3.3 Nonomino

In the case of the Nonomino puzzles, the experiments started with coloring an empty graph with a similar structure to the Nonomino that we selected, using the backtracking algorithm. An empty graph means the biggest problem space for the backtracking algorithm (it has to find a color to all the 81 vertices), therefore every smaller problem - e.g. graphs with only 30 vertices to color - mean a way smaller execution time. In the case of the empty Nonomino puzzle, the backtracking graph coloring algorithm executed in **33.421 seconds**.

We could not test the genetic algotihm on Nonominos with more than 30 empty cells, because after that threshold a maximum iteration count (a limiter for the algorithm, with the functionality to stop the iterations after 10000 generations) was reached very often and the algorithm could not converge or find a solution. In many cases, running 10000 iterations took 30-60 minutes. After several tries, we concluded that on average the implemented genetic algorithm cannot solve a Nonomino with more than 33-34 missing values. As a consequence of this limitation, we started the testing with no more than 30 empty cells in each of the 25 prepared puzzles.

When the Nonomino puzzle had **30 missing values**, the genetic algorithm performed well. The results are summarized in table 5. As a general observation, the population size had the biggest impact on the execution time, and also huge population sizes led to huge variance in the execution time of a coloring. Correspondingly, populations with a size of 30 required more iterations (number of generations - meaning the number of generations needed to get an individual with a fitness score of 0, in other words: a solution) to reach a complete coloring of the Nonomino graph.

Based on the performance of the genetic algorithm on the Nonomino graph with 30 uncolored vertices, we reach the conclusion that GA with a parameter setting of higher population size is going to perform worse in general.

Table 5 Experiment results of a Nonomino with 30 missing values.

Parameters				Number of generations		Execution time (seconds)	
Population size	P(mutation)	P(crossover)	Selection pressure	Mean	Std. dev.	Mean	Std. dev.
10	0.03	0.9	5	90.5	41.180	1.478	0.662
10	0.03	0.9	10	107.9	57.567	1.752	0.918
10	0.03	1.0	5	111.0	64.137	1.808	1.031
10	0.03	1.0	10	103.5	61.336	1.690	0.986
10	0.07	0.9	5	102.04	56.919	1.711	0.925
10	0.07	0.9	10	98.966	53.226	1.687	0.863
10	0.07	1.0	5	98.585	51.947	1.706	0.857
10	0.07	1.0	10	99.162	52.780	1.736	0.890
20	0.03	0.9	5	102.888	61.278	1.844	1.128
20	0.03	0.9	10	101.020	59.331	1.835	1.090
20	0.03	1.0	5	103.009	63.276	1.900	1.201
20	0.03	1.0	10	100.316	61.539	1.866	1.162
20	0.07	0.9	5	105.523	69.916	2.049	1.519
20	0.07	0.9	10	104.935	67.886	2.081	1.483
20	0.07	1.0	5	105.673	66.069	2.141	1.463
20	0.07	1.0	10	105.793	65.007	2.180	1.454
30	0.03	0.9	5	104.782	63.498	2.182	1.418
30	0.03	0.9	10	105.833	64.075	2.232	1.450
30	0.03	1.0	5	106.142	63.769	2.262	1.454
30	0.03	1.0	10	107.045	64.656	2.306	1.492
30	0.07	0.9	5	128.790	150.265	3.059	4.780
30	0.07	0.9	10	128.704	147.089	3.109	4.685
30	0.07	1.0	5	133.078	149.607	3.298	4.805
30	0.07	1.0	10	131.537	146.894	3.292	4.712

Table 6 Experiment results of a Nonomino with 15 missing values.

Parameters				Number of generations		Execution time (seconds)	
Population size	P(mutation)	P(crossover)	Selection pressure	Mean	Std. dev.	Mean	Std. dev.
10	0.03	0.9	5	26.100	14.745	0.393	0.202
10	0.03	0.9	10	54.450	103.322	0.788	1.441
10	0.03	1.0	5	563.933	1434.865	8.080	20.521
10	0.03	1.0	10	541.025	1339.086	7.758	19.155
10	0.07	0.9	5	436.480	1213.190	6.269	17.352
10	0.07	0.9	10	393.400	1120.172	5.684	16.037
10	0.07	1.0	5	359.457	1046.462	5.223	14.996
10	0.07	1.0	10	364.012	985.846	5.348	14.152
20	0.03	0.9	5	342.433	936.265	5.068	13.458
20	0.03	0.9	10	312.450	892.670	4.636	12.833
20	0.03	1.0	5	317.854	877.480	4.784	12.753
20	0.03	1.0	10	322.425	863.368	4.908	12.659
20	0.07	0.9	5	298.915	833.258	4.559	12.219
20	0.07	0.9	10	278.778	806.029	4.261	11.821
20	0.07	1.0	5	262.393	781.031	4.023	11.454
20	0.07	1.0	10	246.787	758.501	3.791	11.125
30	0.03	0.9	5	234.529	737.602	3.615	10.820
30	0.03	0.9	10	235.000	721.353	3.671	10.627
30	0.03	1.0	5	234.505	709.763	3.709	10.533
30	0.03	1.0	10	228.935	696.039	3.645	10.364
30	0.07	0.9	5	218.857	680.687	3.494	10.136
30	0.07	0.9	10	209.500	666.353	3.353	9.923
30	0.07	1.0	5	201.000	652.872	3.225	9.723
30	0.07	1.0	10	193.179	640.175	3.107	9.534

In the case of a simpler, easier to solve Nonomino with only **10 empty cells**, the population size did not have a signicant effect on the execution time. On the other hand the number of generations slightly decreased. In experiments with a population size of 30, on average the genetic algorithm reached a solution in 10 generations (hence it had more ”chances” to generate a correct coloring). The standard deviation also stayed in the same interval throughout the whole series of experiments, between 0.1 and 0.2.

We can conclude that using our genetic algorithm implementation to solve a simple Nonomino puzzle with only a few missing values will reliably produce fast results. As the problem size is shrinking, the effect of heuristics on the result is also shirking, therefore a genetic algorithm is going to be able to reach a solution more and more reliably. It is important to notice that the population size is going to affect the execution time and the required number of generations, but only in a minor way.

Table 7 Experiment results of a Nonomino with 10 missing values.

Parameters				Number of generations		Execution time (seconds)	
Population size	P(mutation)	P(crossover)	Selection pressure	Mean	Std. dev.	Mean	Std. dev.
10	0.03	0.9	5	17.300	8.525	0.269	0.118
10	0.03	0.9	10	16.650	12.974	0.259	0.176
10	0.03	1.0	5	17.333	14.310	0.270	0.197
10	0.03	1.0	10	15.425	13.735	0.244	0.189
10	0.07	0.9	5	14.000	12.707	0.226	0.174
10	0.07	0.9	10	13.033	11.934	0.213	0.164
10	0.07	1.0	5	12.271	11.433	0.204	0.157
10	0.07	1.0	10	11.975	10.916	0.201	0.150
20	0.03	0.9	5	11.966	10.664	0.206	0.149
20	0.03	0.9	10	12.110	10.637	0.212	0.152
20	0.03	1.0	5	11.681	10.561	0.209	0.151
20	0.03	1.0	10	11.608	10.248	0.211	0.147
20	0.07	0.9	5	11.123	10.024	0.207	0.143
20	0.07	0.9	10	10.700	9.801	0.203	0.139
20	0.07	1.0	5	10.613	9.669	0.205	0.139
20	0.07	1.0	10	10.456	9.559	0.205	0.138
30	0.03	0.9	5	10.176	9.429	0.204	0.136
30	0.03	0.9	10	10.033	9.312	0.205	0.135
30	0.03	1.0	5	10.126	9.146	0.210	0.135
30	0.03	1.0	10	10.090	9.120	0.213	0.136
30	0.07	0.9	5	9.895	9.012	0.213	0.135
30	0.07	0.9	10	9.718	8.883	0.213	0.133
30	0.07	1.0	5	9.413	8.816	0.210	0.131
30	0.07	1.0	10	9.358	8.676	0.212	0.130

5.4 Other methods for graph coloring

5.4.1 Welsh-Powell Graph Coloring Algorithm

The Welsh-Powell algorithm is a greedy method to color a static graph. Firstly, the vertices are ordered by their degree, the first vertex is colored with the first color, every non-adjacent vertex is colored with that same color, and after that the remaining vertices in the list are colored with the next available color (it is allowed to color a vertex with a given color if there is no adjacent vertex with the same color)[6]. It uses at most one more color than the graph's maximum degree to label the graph and has a time complexity of $\mathcal{O}(n^2)$.

Algorithm 8 Welsh-Powell algorithm for graph coloring

```
Create a list $V(G)$ from the vertices
Calcuate the degree of each vertex
Sort the list of vertices $V(G)$ by the degree of the vertices
while There is a vertex in $V(G)$ that is not colored do
    The first non-colored vertex $X$ in the list is colored with the next available color $C$
    The remaining part of the list of vertices is traversed and every vertex that is not adjacent with the first non-colored vertex $X$, is colored with $C$
end while
```

6 Conclusion

In this reasearch we attempted to solve the graph k-coloring problem and apply the solution to three selected problem classes: Sudoku, Hypersudoku and Nonomino puzzles. In each case, we created a software component that is able to transform the textual representation of each puzzle into a graph. In such a graph, the vertices represent a cell in the puzzle and the edges of the graph are derived from the structure of the Sudoku board.

From the algorithm's point of view, a Sudoku board is represented as a partially filled graph, each node being connected to the nodes that are in the represented cell's row, column or smaller square (3x3 area in the case of a Sudoku). The fitness of an individual (graph) is calculated based on the number of nodes that are connected and have the same color coding, hence a correctly colored graph is an individual with a fitness value of 0. This is the optimum (maximum) that we wanted to reach during a run. As a selection method, we used tournament selection and a variation of swap mutation. To select the survivors, we used the elitist selection method.

The proposed genetic algorithm managed to solve the selected problems, but only the easy ones. On average, if the number of missing values (the number of nodes to color in the graph representation) was bigger than 30, the algorithm failed to find a solution. In the case of the most simple porblems (only 10 values to color), the algorithm quickly found a possible solution. In contrast, when the graph had about 30 uncolored nodes, our implementation was not efficient.

The biggest drawback of using a genetic approach to solve the k-coloring problem is the fact that it is not guaranteed that the algorithm is going to find a solution in finite time. We observed the scale and impact of this problem during the experiments and came to the realization that our implementation is not able to solve a difficult, real life Sudoku (which - in general - has at least 40-50 missing values). A second weakness we want to outline is the suboptimal effectiveness (longer running time, based on experiments) compared to other graph coloring strategies (e.g. backtracking and Welsh-Powell algorithm).

The heuristics of the algorithm can also give a huge amount of speedup during the search for a correct, complete coloring. In theory, it is possible to find a solution in the very first generation or at least drastically decrease the search space.

In conclusion, a genetic approach to garph coloring has the possibility of being very fast way to search for an optimum, even if our proposed algorithm provided an uneffective way to find a solution in the case of the three puzzles.

6.1 Future work

There are multiple ways that might lead to an improved performance of the algorithm proposed in this article. One such way is to increase computing power. This way the algorithm will be able to work with bigger population and it might find the solution to a sudoku faster. With more computational power multiple populations can be initialised, this way it would be more likely that the solution is found.

Another method that might improve the results is to use a hybrid genetic algorithm. Local search can be applied to find the solution when one or more of the individuals have fitness values close to 0. In some cases this might speed up the process of finding the right solution but it also increases the chance of converging to a local optimum. In order to preserve the diversity of the population the Tabu Search algorithm can be used. This way we can avoid the case where the algorithm gets stuck in a local optima, but this method also needs more memory to run.

References

1. R.M.R. Lewis, A guide to Graph Coloring - Algorithms and Applications, pg. 1-6. Springer, Switzerland (2016)
2. Solomon W. Golomb, Polyominoes - Puzzlers, Patterns, Problems and Packings, pg. 73-78. Priceton University Press, Priceton, New Jersey (1994)
3. Nonomino puzzle collection, selected puzzles: N8.00, N8.13, N8.23, N8.30 and N8.59 , URL: https://www.boldts.net/Sudoku/N0/8.pdf, accessed: 2020.12.21
4. Nonomino puzzle - Solutions, selected puzzles: N8.00, N8.13, N8.23, N8.30 and N8.59 , URL: https://www.boldts.net/Sudoku/N0/Answers/8.txt, accessed: 2020.12.22
5. Mantere, Timo and Janne Koljonen (2008). Solving and Analyzing Sudokus with Cultural Algorithms. In Proceedings of 2008 IEEE World Congress on Computational Intelligence (WCCI 2008), 1-6 June, Hong Kong, China, pages 4054-4061.
6. GraphStream Project Documentation - The Welsh-Powell Algorithm, URL: https://graphstream-project.org/doc/Algorithms/Welsh-Powell/, accessed: 2020.12.19
7. Gen, M., and Lin, L. (2007). Genetic algorithms. Wiley encyclopedia of computer science and engineering, 1-15.
8. Jensen, T. R., and Toft, B. (2011). Graph coloring problems (Vol. 39). John Wiley and Sons.
9. Lewis, R. (2015). A guide to graph colouring (Vol. 7). Berlin: Springer.