Developing Heuristics for the Graph Coloring Problem Applied to Register Allocation in Embedded Systems

Adrian Florea, Arpad Gellert
Computer Science and Electrical Engineering Department
"Lucian Blaga" University of Sibiu
Sibiu, Romania
adrian.florea@ulbsibiu.ro, arpad.gellert@ulbsibiu.ro

ABSTRACT: *The main aim of this paper consists in developing an effective e-learning tool, focused on evolutionary algorithms, in order to solve the graph coloring problem. Subsidiary, we apply graph coloring for register allocation in embedded systems. From didactic viewpoint, our tool has benefits in the learning process because it helps students to observe the relationship between the graph coloring problem and CPU registers allocation with the help of four developed modules: the two evolutionary algorithms (Genetic Algorithm and Particle Swarm Optimization), the graphical viewer, the interference graph for a C program and a web application which collects the simulation results. All these applications are combined by a graphical interface which allows the user to configure the parameters of the genetic algorithm and to analyze their effect over the convergence.*

Keywords: Graph Coloring, Register Allocation, Embedded Systems, Genetic Algorithm, Particle Swarm Optimization

Received: 18 May 2017, Revised 29 June 2017, Accepted 5 July 2017

1. Introduction

This paper extends the previous work that was presented at the Sixth International Conference on Innovative Computing Technology (INTECH 2016) [1], by presenting some features added to our developed software tool like new heuristics in solving graph coloring problem, results centralization in online database and remote visualization on server. It will facilitate further analysis and finding optimal control parameters of evolutionary algorithms. We integrated the Particle Swarm Optimization (PSO) algorithm, showing the particularities and difficulties in applying to graph coloring problem (GCP), and illustrating an e-learning solution that maps the discreet value problem (GCP) into a continuous values space (PSO).

Graph coloring is one of the most important graph theory concepts. It is an NP-complete problem, thus the time necessary to find

the optimal solution is exponential. The problem implies vertex coloring and in some cases also edge coloring in an undirected graph. The adjacent vertices must have different colors and the total number of used colors must be as low as possible. For graphs with high number of vertices, finding the optimal solution is unfeasible, but heuristic algorithms can converge to acceptable solutions.

Mathematically, each graph has associated a chromatic polynomial which shows in how many different ways the graph can be colored using a certain number of colors. The chromatic polynomial is a function $P(G, t)$, where G is the graph and t is the number of colors. The chromatic number c shows the minimum number of colors necessary to solve the problem and it is the lowest integer which is not a solution of the chromatic polynomial:

$$\chi(G) = \min \{k : P(G, k) > 0\} \quad (1)$$

There are known chromatic polynomial formulas for certain graph classes (forest graphs, chordal graphs, cyclic graphs).

Several graph coloring methods have been already applied, such as greedy algorithms, genetic algorithms, local search algorithms, etc. Graph coloring has been used to solve problems in different domains: map coloring, frequency allocation in mobile phone networks, radio frequency allocation, register allocation in compilers, scheduling problems, etc.

The graph coloring problem represents an important concept of graph theory and has numerous applications in the conflicts resolution in computer science problems ranging from games implementation, the design of mobile phone networks until scheduling problems and compiler optimizations such as registers allocation in embedded systems. In this paper, we apply graph coloring for register allocation in embedded systems. Due to the specific technological requirements, the embedded processors are limited in resources, memory and also have power consumption constraints. The number of registers is quite low and usually some of the registers are dedicated for certain instructions and, therefore, the registers must be efficiently used. Such systems need compilers that generate optimal code. Thus, several optimization techniques have been developed that, based on architectural knowledge, can optimize the code.

Register allocation is an important compiler stage. Its main function is to map variables into the memory or registers. Due to the high access latency difference between memory and registers, keeping the variables in registers is preferred. When a register is needed, and all the registers are occupied, one of them must be spilled to the memory. This process implies additional costs and, thus, lower performance and higher energy consumption. Therefore, compilers must minimize the number of used registers and the number of memory accesses.

Register allocation can be optimized using graph coloring algorithms. The lifetime of the variables is analyzed and an interference graph is generated with vertices representing the variables. There are edges between the vertices corresponding to variables which are in use simultaneously. The colors are associated to the available registers. If the graph coloring is not possible, the algorithm identifies the variables to be spilled to the memory, resulting thus a new graph which must be colored. Our application can be used as an e-Learning tool due to its graphical interface which is visualizing the colored graph. The user can change the parameters of the algorithm and can analyze their effect over the convergence. For evaluations we used the benchmarks of the Center for Discrete Mathematics & Theoretical Computer Science (DIMACS).

The remainder of this paper is organized as follows. Section 2 reviews the state of the art in register allocation and graph coloring techniques. Section 3 presents the software design of the application, describing in details the four developed modules: the applied heuristics, the graphical viewer, the interference graph and the web application which collects the simulation results. Section 4 discusses the experimental results. Finally, Section 5 emphasizes the conclusions and presents some further work directions.

2. Related Work

Topcuoglu et al. applied a hybrid evolutionary algorithm for register allocation in embedded systems [2]. Their solution combines genetic algorithms with a local search technique. They introduced a novel, highly specialized crossover operator that is using domain-specific information. The experimental results, performed on synthetic benchmarks and routines extracted from benchmark suites, show good performance in allocating the registers to variables.

Liu et al. analyzed register allocation techniques in embedded systems with the goal of simultaneously reducing energy consumption and heat buildup on register accesses [3]. Contrary with their intuition, the experiments have shown that reducing energy consumption and decreasing heat dissipation on register accesses are two conflicting objectives. Therefore, they introduced a rotator hardware into the instruction decoder in order to facilitate a balanced solution for both objectives. The authors proposed register allocation and refinement based on the access patterns and the effects of the rotator. The results have shown energy and peak temperature improvements for embedded applications.

Homayoun et al. proposed an architectural solution to reduce the peak temperature of the register file which redistributes the access pattern to physical registers through a novel register allocation mechanism [4]. They introduced the idea of local activity migration to manage register file temperature in embedded out-of-order processors. The register file is partitioned into multiple regions and, thus, the accesses can be distributed in a non-uniform pattern over these regions. The accessed regions are spatially and temporally apart, allowing for other regions to cool-down. Thus, some regions can be kept unused and cooling down while other regions are active. Since only a subset of physical registers is used at any given time, the migration is solved within the register file itself, avoiding a replicated unit. For this, a new register renamer was necessary, which attempts to concentrate register allocations in a given region. Periodically, the redistribution mechanism switches to another partition. The experiments have shown that a 64-entry physical register file with 4 partitions performs best, the temperature being reduced without impact on the performance.

Huang et al. used a register allocation technique to minimize the number of store instructions, reducing thus the write activities on the non-volatile memory [5]. The experiments demonstrated that the proposed technique reduces the number of store instructions by 33%, which improves the system performance, but also extends the lifetime of non-volatile memory.

Mahajan presents a hybrid evolutionary algorithm for graph coloring based register allocation [6]. They used a new crossover by conflict-free sets and a new local search function.

Lee et al. presents a register allocation technique that translates memory accesses to register accesses in order to improve the performance of embedded software [7]. The source code is profiled to generate a memory trace. The proposed register allocation technique is applied only on functions with high number of calls, selected from the profiling results, saving thus the compilation time. In the memory trace of the selected functions, the memory accesses that result in latency reduction when replaced by register accesses are identified and translated to register accesses by modifying the intermediate code and allocating promotion registers.

Several solutions for the graph coloring problem have been proposed in recent works. Hindi presents a hybrid technique that applies a genetic algorithm followed by the wisdom of artificial crowds algorithm to solve the graph coloring problem [8]. Chowdhury et al. used binary encoded chromosomes [9]. The proposed algorithm starts with the maximum chromatic number and some of the colors are dynamically eliminated, the solution being thus reached in a single run. Djelloul et al. used a discrete binary version of cuckoo search algorithm [10]. A nature inspired flower pollination algorithm is proposed by Bensouyad [11], the artificial bee colony algorithm was applied by Dorrigiv [12], as well as ant-based algorithms have been used by Hertz [13] and Bui [14]. A fuzzy clustering based evolutionary approach has been used by Lee [15]. Zhao et al. used simulated annealing [16], whereas Titiloye and Crispin applied quantum annealing which extends simulated annealing by introducing artificial quantum fluctuations [17].

3. Application Software Design

The software application was implemented in C#, under Visual Studio 2013 Express edition. The sever side web application was developed in ASP.NET MVC. The client side was developed using HTML, CSS, JavaScript and Razor (an ASP.NET syntax).

For the graphical interface we used the 'Windows Forms' library provided by Microsoft. In order to take advantage by parallel evaluation of the chromosomes when the genetic algorithm is applied, hardware systems with larger number of cores are recommended. As Figure 1 shows, the application consists in four distinct components: the two evolutionary algorithms (Genetic Algorithm and Particle Swarm Optimization), the graphical viewer, an application which creates the interference graph for a C program and a web application which collects the simulation results. All these applications are combined by a graphical interface which allows the user to configure the parameters of the genetic algorithm and to analyze their effect over the convergence.

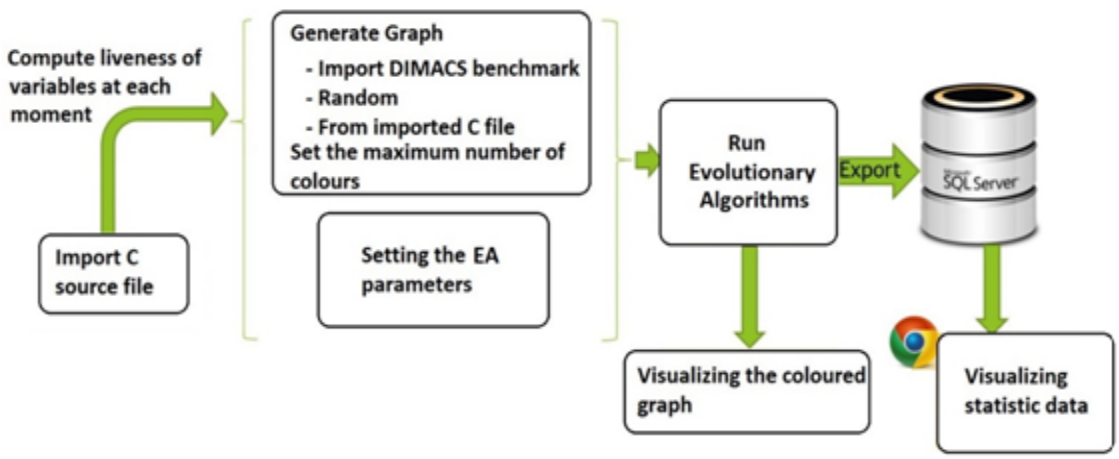


Figure 1. The software architecture of the application

3.1 The Genetic Algorithm

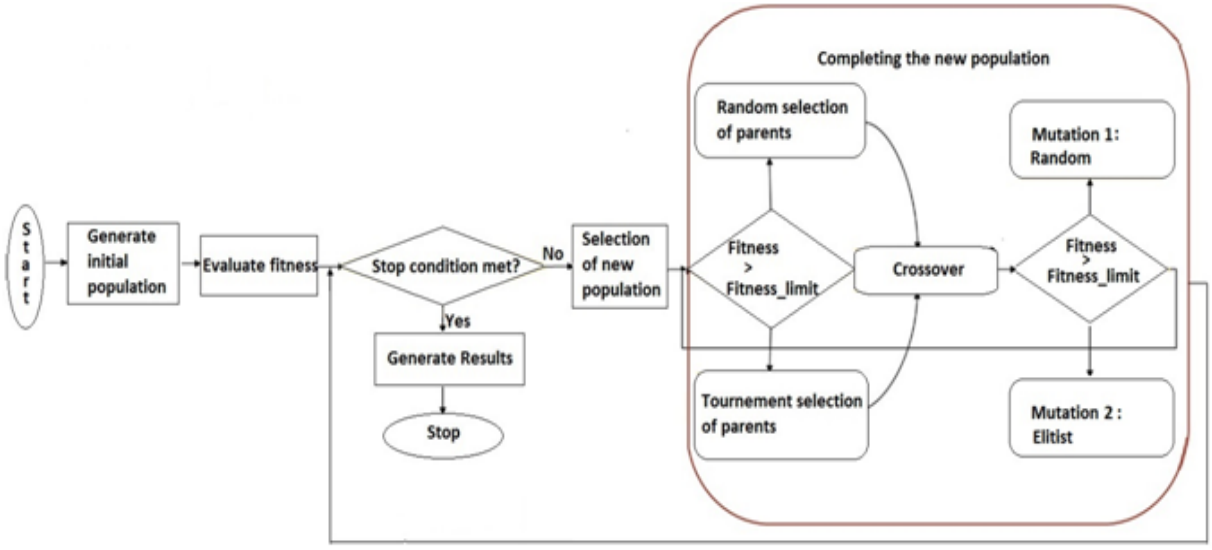


Figure 2. The Genetic Algorithm

We used the genetic algorithm presented in [8]. The algorithm starts with a randomly generated population. Each solution is evaluated with the fitness function. In the population of the next generation will survive a certain percentage of individuals, composed by those with the lowest fitness (elitist approach). We set the limit value of fitness that will cause changing the applied genetic operators. The new individuals are generated through crossover or mutation in parallel. The fitness function will be determined by the correctness of the colored graph (no adjacent nodes of the same color meaning a desired fitness of 0) and the aim will be to minimize the number of colors used. Thus, if the fitness function is approaching to 0, parents' selection for crossover will be random, otherwise it will apply the tournament method: two pairs of individuals are chosen, and the best ones will become parents. The 1-point crossover operator is used. It chooses a random point on the two parents, splits the parents at this crossover point and creates children by exchanging tails. On the individual which resulted from the crossover process a mutation is applied depending on the mutation rate. The mutation type is decided based on the fitness of the best solutions in the current population. Scrolling along the genes of the chromosome, each vertex that has the same color with a neighboring vertex will be recolored (with a color which is not used by the neighbors or with a color which is randomly chosen from all the colors), removing thus the conflicts. The algorithm stops when it achieves a certain fitness level or when the number of

iterations reaches a certain threshold. At the end of the execution, the coloring solution is exported into a text file together with the adjacency matrix of the graph.

3.2 Mapping the Particle Swarm Optimization algorithm for the Graph Coloring Problem

The PSO technique was introduced in 1995 by Eberhart and Kennedy [18] as a stochastic search and optimization technique applied successfully for solving real-world problems. It belongs to swarm-based heuristics methods that mimic social behavior of bird flocking and fish schooling. There are some similarities between genetic algorithms and PSO. First, both algorithms use a population of potential solutions which is randomly initialized. Second, both have fitness values to evaluate the population. PSO relies on a combination of both cognitive personal knowledge (for diversity, similar with mutation in GA) and social knowledge (for convergence and exploration like crossover in GA) of the given search space. However, PSO has dissimilarities to GA. In PSO, the units of selection (individuals) are referred as "particles", which are "birds" in the flock. The individuals are interacting together and share information for problem solving, and are flying through the problems' space by following the current optimum particle. Each particle has memory (its best previous position), which is important to the algorithm but, there is no evolution operators. In general the PSO is easier to implement although, as in the Graph Coloring Problem case, it is mandatory to make some additional data representations and computing tricks. Figure 3 presents the generic flowchart of the PSO algorithm. Two features are very important for a particle: Position and Velocity. Every particle knows its current location, the neighbors' positions, the global best location, its best previous position and the objective function value and uses all these information in order to move to next location.

The movement equations are the following:

$$\begin{cases} V_k(t+1) = w \cdot V_k(t) + c_1 \cdot R_1 \cdot \left(P_k(t) - X_k(t)\right) + c_2 \cdot R_2 \cdot \left(P_g(t) - X_k(t)\right) \\ X_k(t+1) = X_k(t) + V_k(t+1) \end{cases} \quad (2)$$

In equation 2 we made the notations:

• $V_k(t)$ – is the velocity of the *k*-th particle at time (generation) *t*, where *k* represents a natural number between 1 and the size of swarm (*P_size*).

• $P_k(t)$ – is the personal best position of the particle *k* from the previous *t* generations (best local).

• $P_g(t)$ – is the best global position from all neighbors.

• $X_k(t)$ – means the current position of particle *k* at time *t*.

• R_1, R_2 – are random functions with values in the range [0,1].

• *w* – represents inertia weight that forces the particle to move in the same direction as it did in the previous generation and illustrates how much the particle trusts itself. Larger *w* means big steps in the search space that may lead to skip some good individuals.

• c_1 - is positive acceleration constant which controls the influence of best local on the search process. It supervises the impact of the cognitive component maintaining the diversity in population and illustrates how much the particle trusts in its experience.

• c_2 - is positive acceleration constant which controls the influence of best global on the search process. It manages the impact of the social component ensuring reaching the convergence of the algorithm and exhibits how much the particle is based on its neighbors.

As equation 2 shows, the velocity represents a vector which has 3 components: inertia, movement towards the best global particle, movement towards personal best (local). If the cognitive coefficient is much lower than social coefficients ($c_1 << c_2 c_1 << c_2$), that is leading to premature convergence, particles being attracted (dominated by) to the global best. Conversely, if ($c_2 << c_1$), the particles are attracted to the personal best, meaning a slow or even no convergence. Using big values for social and cognitive coefficients determines the velocity to rise too much and the algorithm will diverge.

However the PSO has been applied successfully on continuous optimization problems sometimes outperforming even GA [19],

the real challenges appear when it needs to map a discreet value problem into a continuous values space such as is the Graph Coloring Problem case. The difficulty lies in the representation of the Position and Velocity vectors. For the GCP the position vector represents a sequence of colors corresponding to each node. This means that valid position vectors are only vectors containing integers between one and the number of available colors. Thus, the position vector has discreet values (in a very limited value space) whilst the particle moves in a continuous value space (position is updated based on the velocity, which is a real number). Therefore, directly mapping the GCP on the PSO algorithm would not offer satisfactory results as the valid solution space is too restricted compared to the solution space the PSO is searching in.

Further, for students who want to better understand the advantages of PSO and, to see comparatively how GA and PSO behave on GCP problem, we apply the discrete particle swarm optimization approach, presented in [20] for grid job scheduling, to GCP. The main novelty consists in replacing Position and Velocity vectors with 2D matrix.

For this we use an indirect representation of the PSO solution (particle), called position matrix ($Cx\,N$ matrix, where C is number of available colors, and N is the number of vertices from the graph) containing only values of 0 and 1. Each column can have only one element that is 1 (meaning that a vertex will receive one and only one color, not more). Assuming that X_k shows the position of the k-th particle, then $X_k[i, j] = 1$ means that the j^{th} vertex is colored with the i^{th} color (see equation 3).

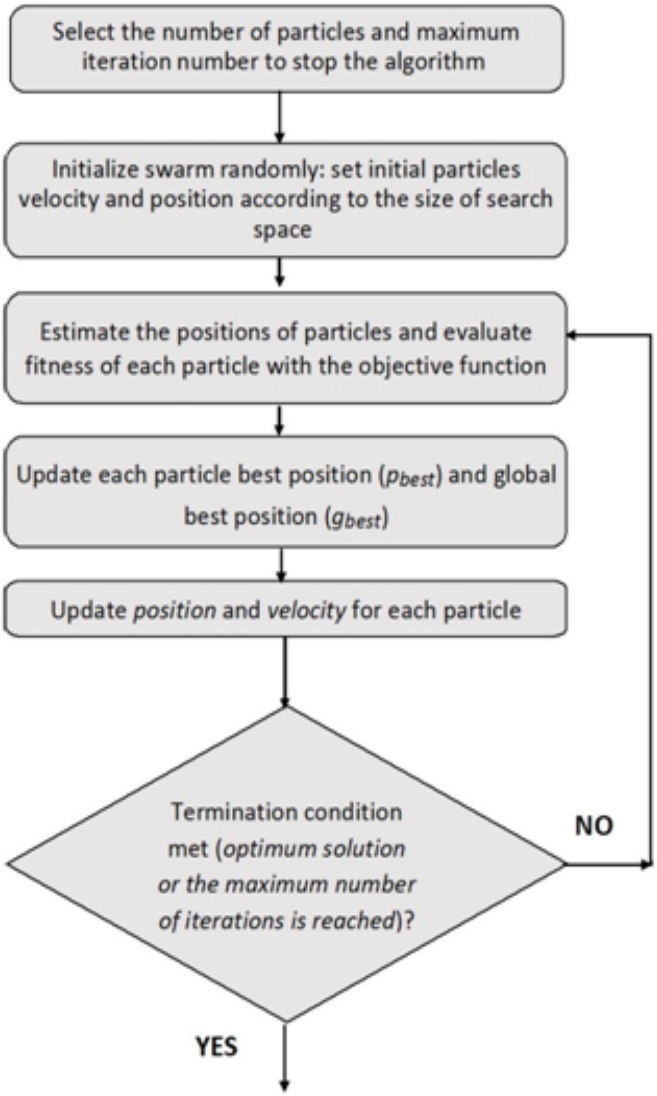


Figure 3. The generic flowchart of the PSO algorithm

$$X_k[i,j] \in \{0,1\}\ \forall i \in [1,C],\ \forall j \in [1,N],\ \forall k \in [1,P\_Size] \quad (3)$$

For example, let us considering a graph colored by 3 colors and having 5 nodes, where nodes 2 and 4 are colored with color 1, the nodes 3 and 5 are colored with color 2, and the first node is colored with color 3. The indirect representation of particle is shown in figure 4.

	N1	N2	N3	N4	N5
C1	0	1	0	1	0
C2	0	0	1	0	1
C3	1	0	0	0	0

Figure 4. Example of position matrix

The personal best position and the global best position must use the same representation as the position matrix. The rule of updating the position matrix is expressed in equation 4, where $V_k^{(t)}[i,j]$ is the element in the *i*-th row and the *j*-th column of the *k*-th velocity matrix in the t-th time step of the algorithm.

$$X_k^{(t+1)}[i,j] = \begin{cases} 1, if\ (V_k^{(t+1)}[i,j] = max\{V_k^{(t+1)}[i,j]\}), \forall i \in [1,C], \forall j \in [1,N] \\ 0, \text{otherwise} \end{cases} \quad (4)$$

Equation (4) means that on each column in the position matrix, a 1 will be written to the element whose corresponding element in the velocity matrix has the maximum value in its corresponding column (meaning that is the closest to the global best). The rest of the elements on that column will be filled in with 0. If in a column of the velocity matrix there is more than one element with maximum value, then one of these elements is selected randomly and 1 assigned to its corresponding element in the position matrix.

The velocity is also a C x N matrix containing values between a defined minimum and maximum value.

$$V_k[i,j] \in [-V_{max}, V_{max}], \forall i \in [1,C], \forall j \in [1,N] \quad (5)$$

The classical update equation of the velocity would be:

$$V_k^{(t+1)}[i,j] = V_k^t[i,j] + c_1 . R_1 . (P_k^t[i,j] - X_k^t[i,j]) + c_2 . R_2 . (P_g^t[i,j] - X_k^t[i,j]) \quad (6)$$

In equation (6) $P_k^t[i,j]$ is the element in the *i*-th row and the *j*-th column of the *k*-th *personal best matrix* in the *t*-th time step of the algorithm, whilst $P_g^t[i,j]$ is the element in the *i*-th row and the *j*-th column of the *global best matrix* in the *t*-th time step of the algorithm.

Taking into consideration that the *global best matrix* and the *personal best matrix* and the position matrix *X* of each particle contain only values of 0 and 1, the following possibilities arise:

- cognitive personal knowledge added from generation *t* to *t* + 1 for the *k*-th particle might be 0, $c_1 . R_1$, $-c_1 . R_1$ depending on the difference of each corresponding element from the personal best matrix and the position matrix X.
- social knowledge added from generation *t* to *t* + 1 for the *k*-th particle might be 0, $c_2 . R_2$, $-c_2 . R_2$ depending on the difference of each corresponding element from the global best matrix and the position matrix *X*.

In this moment there are known the all operations and data structures required to implement the PSO algorithm from Figure 3. The main drawback is that in some cases of complex benchmarks with high number of vertices and edges, the algorithm does not converge to a solution in a reasonable time. Our tests have shown that not all the benchmarks passed (providing a coloring solution) like in the genetic algorithm case. At the end of the execution, if PSO converges, the existing coloring solution is exported into a text file together with the adjacency matrix of the graph. A solution developed to solve the premature convergence problem of the standard PSO was proposed in [21] but the implementation of this accelerated PSO algorithm in our application is beyond the scope of this paper.

3.3 The Viewer

The graphical visualization application will use as input file the solution generated by the genetic algorithm. The structure of such a file is illustrated in Figure 5. The first line shows the coloring solution: the color index for each vertex. The rest of the lines are containing the adjacency matrix of the colored graph, where $A[i, j] = 1$ if there is an edge between i and j and $A[i, j] = 0$ if the vertices i and j are not connected.

Figure 6 illustrates how a certain coloring solution is graphically visualized. The colors are randomly selected from the Color structure of *System. Drawing*. The positions of the vertices are computed depending on their number. For large graphs, the positions of the vertices are randomly generated. For relatively small graphs, the vertices are positioned on a circle whose center is in the center of the visualization form and its radius is the half of the height of the visualization form.

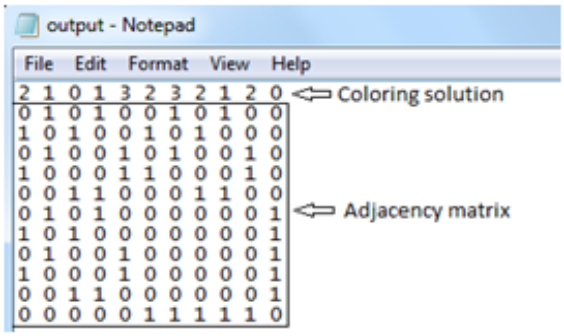


Figure 5. Coloring Solution Example

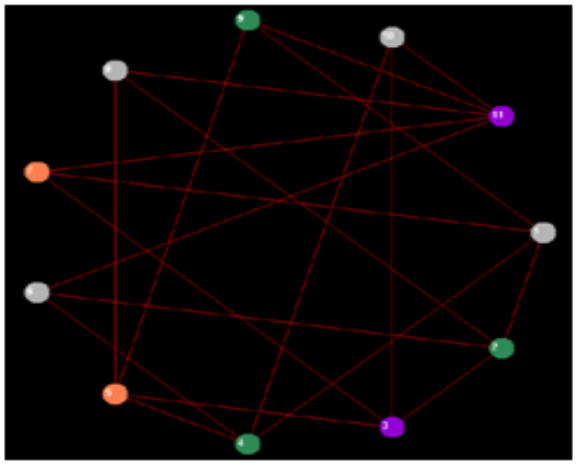


Figure 6. A visualized coloring solution

We used the parametrical equation of the circle:

$$x = a + r \cdot \cos t$$

$$y = b + r \cdot \sin t \qquad (7)$$

where x and y are the obtained coordinates on the circle with radius r and center in the point (a, b) and t is the angle between the x-axis and the radius connecting (x, y) wdith (a, b). For drawing we used the methods from the Graphics class.

3.4 Interference Graph Generation

This application prepares the input data for the genetic algorithm. The interference graph is a graph associated to the *C* code. Each variable has a corresponding vertex in the interference graph. Two vertices are linked by an edge if the corresponding variables are simultaneously alive. The adjacency matrix is exported to a file which will be used by the genetic algorithm.

3.5 The Graphical Interface

The highly parameterized graphical interface (see Figure 7) combines all the previously described components. It allows to configure the parameters of the GA and PSO and to define the problem which must be solved. The user can view in real time the progress of the algorithm and its convergence through a graphic presenting the best fitness value which is updated in each iteration.

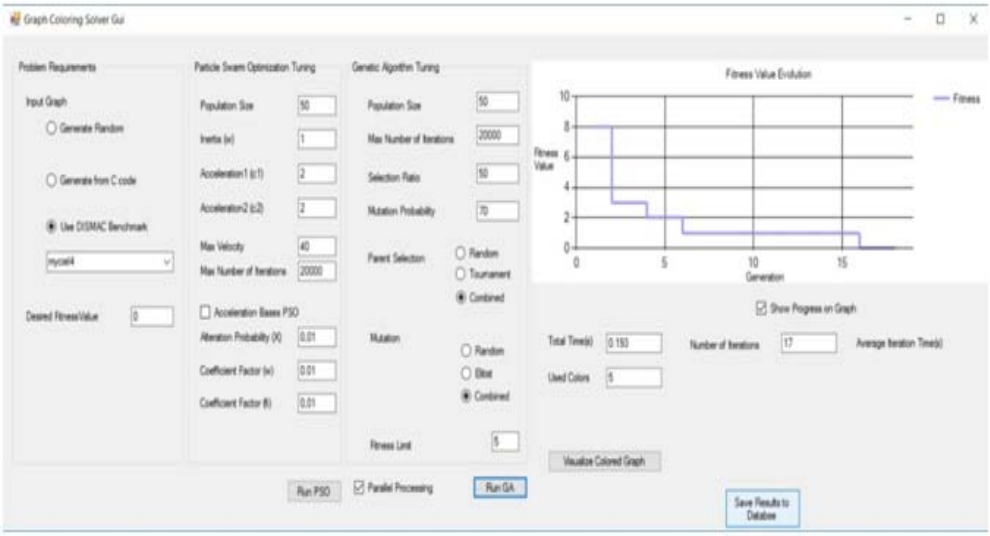


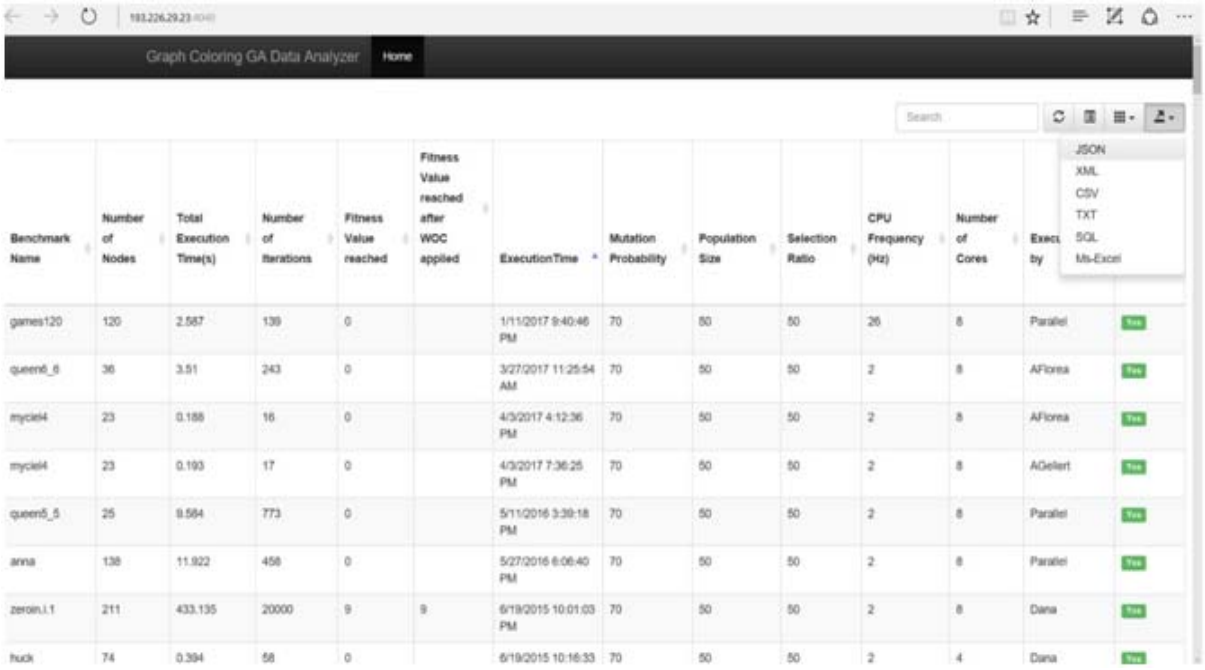
Figure 7. User interface for configuring the GA and PSO solving the Graph Coloring Problem

3.6 The Database

We have also introduced an online database in order to keep the simulation results on the server for further analysis. Thus, searching the optimal parameters for the genetic algorithm is a simple process. This functionality is available only for the results obtained on the DIMACS benchmarks. At the end of the simulation the Save Results to Database button becomes usable. This button opens a dialog where the user can introduce data about the workstation and a name for the possibility to filter and analyze only the own results. The details about the workstation (memory size, number of cores and processor frequency) are important for the relevance of the results.

For this functionality we have created two classes: DbConnection and DbRepo. The DbConnection class is responsible with the connection to the database and also with the maintenance of the database. For a connection to the database, we need a Connection String containing the address of the database server, the name of the database and the user authentication details (name and password) defined in SQL Server. The DbRepo class contains the methods which are inserting the data through a command created with given parameters. The command is executed on SQL server, adding thus the data into the corresponding tables. The procedures which insert data about the benchmark, workstation and the parameters of the genetic algorithm are not allowing duplicate entries.

The experimental results can be visualized through a web application developed in ASP.NET MVC, hosted on the same server where the database is. Its client side, visible from browsers, is represented by the View developed using HTML, CSS, JavaScript and Razor. For the design (CSS) and client-side script (JavaScript) we used the Bootstrap framework and the Bootstrap-Table library which provides searching, sorting, filtering and exporting functionalities. For the database we have used Entity Framework which is a relational object mapper, allowing the .Net developers to use C# specific objects in relational databases. A useful property is the possibility to retrieve together objects linked through foreign key. The server side contains a controller which takes the requests of the client, gets the data from the database using Entity Framework and sends them to the View component to be shown in the browser. In order to load the application on the server, the following steps are necessary: IIS (Internet Information Services) must be installed; a new site must be added using IIS Manager for which a port and the file paths must be specified; the site must be created in Application Pool, a group of applications running on the web server. The application is hosted on a server and is accessible at http://193.226.29.23:4040. Figure 8 presents the graphical interface of the application which collects and shows the simulation results. It allows sorting, searching, refreshing, filtering, exporting and different visualization possibilities.



Benchmark Name	Number of Nodes	Total Execution Time(s)	Number of Iterations	Fitness Value reached	Fitness Value reached after WOC applied	ExecutionTime	Mutation Probability	Population Size	Selection Ratio	CPU Frequency (Hz)	Number of Cores	Execu by	
games120	120	2.587	139	0		1/11/2017 9:40:46 PM	70	50	50	26	8	Parallel	Yes
queen6_6	36	3.51	243	0		3/27/2017 11:25:54 AM	70	50	50	2	8	AFlorea	Yes
myciel4	23	0.188	16	0		4/3/2017 4:12:36 PM	70	50	50	2	8	AFlorea	Yes
myciel4	23	0.193	17	0		4/3/2017 7:36:25 PM	70	50	50	2	8	AGellert	Yes
queen5_5	25	9.584	773	0		5/11/2016 3:39:18 PM	70	50	50	2	8	Parallel	Yes
anna	138	11.922	458	0		5/27/2016 6:06:40 PM	70	50	50	2	8	Parallel	Yes
zeroin.i.1	211	433.135	20000	9	9	6/19/2015 10:01:03 PM	70	50	50	2	8	Dana	Yes
huck	74	0.394	58	0		6/19/2015 10:16:33	70	50	50	2	4	Dana	Yes

Figure 8. The Database

3.7 Parallelization

Since evolutionary algorithms exhibit the feature to be embarrassingly parallel problems, requiring little or no effort to separate the problem into a number of parallel tasks, we parallelized the process of generating new individuals for next generation, with the help of the genetic operators such as crossover and mutation. This is possible using the Parallel class from .NET framework (*System.Threading.Tasks* namespace) that contains specific implementations of *for* and *foreach* loops using multiple threads. A "Task" object is asynchronously running on a thread of the thread pool. This thread pool allows more efficient use of threads and, hence, of the resources, because they are managed by the system. Actually, the Task scheduler partitions the task based on system resources and workload. When possible, the scheduler redistributes work among multiple threads and processors if the workload becomes unbalanced.

Figure 9 illustrates how parallelization works. The *NewSolution* function, which has as argument the population of solutions, applies the parents' selection operator, and then creates a new solution combining the two parents according to the crossover and mutation operators. After completing all the running tasks, the solutions will be added to the new population.

```
1. var noOfTasks = _tuningParameters.PopulationSize - newPopulation.CurrentSize;
2. var tasks = new Task[noOfTasks];
3. var sol = new Solution[noOfTasks];
4. for (int j = 0; j < noOfTasks; j++){
5.    var j1 = j;
6.    tasks[j] = Task.Run(() => { sol[j1] =NewSolution(population);});
7.    }
8. Task.WaitAll(tasks);
9. foreach (var s in sol){
10.    if (newPopulation.CurrentSize < _tuningParameters.PopulationSize)
11.      newPopulation.AddToPopulation(s);
12.    else
13.      break;
14.    }
```

Figure 9. Implementing the Parallelization

The preliminary results illustrate the advantages of multicore architectures. The execution speed in the case of parallel evaluation is improved with 45%. Because our application aims the educational aspects and not necessarily the research aspects and due to the time consuming iterative methods as the generation number increases, we limited this statistic to only 4 DIMACS benchmarks, each running the genetic algorithm for one generation on a population consisting of 100 individuals. All the further reported results were made by parallel evaluation, running the benchmarks until completion. In Table 1 we reported the execution time taken to apply the genetic operators over one generation using a hardware platform with Intel(R) Core(TM) i7-2630QM CPU processor, 2 GHz, 4 GB RAM internal memory.

	Execution time [ms] required by one generation of GA				
Benchmarks	***myciel4***	***myciel5***	***queen5_5***	***queen6_6***	***Average***
Iterative	1.14	5.30	10.33	13.44	7.55
Parallel	0.80	2.44	6.42	7.00	4.17

Table 1. Iterative vs. Parallel chromosome evaluation

4. Benchmarks and Simulation Results

We used for evaluation the DIMACS benchmarks [22] that contain associated graphs for real problems from different fields of application such as: queens problem, scheduling problems and CPU registers allocation problem (those circled in Figure 11). Each benchmark contains the graph definition, the list of edges and the current lowest number of colors necessary for graph coloring. The chromatic number is also given, if it is known. Figure 10 depicts the structure of such a benchmark. The first lines (c) describe the benchmark. The number of vertices and edges are also given (on line p). The current best number of colors is inserted (to line t). The next lines (e) are containing the edges: the pairs of vertices connected by edges.

```
c FILE: DSJC500.1
c
c SOURCE: David Johnson (dsj@research.att.com)
c
c DESCRIPTION: Random graph used in the paper
c              "Optimization by Simulated Annealing: An
c               Experimental Evaluation; Part II, Graph
c               Coloring and Number Partitioning" by
c              David S. Johnson, Cecilia R. Aragon,
c              Lyle A. McGeoch and Catherine Schevon
c              Operations Research, 39, 378-406 (1991)
c
p edge 500 12458
t 12
e 6 2
e 8 3
e 8 4
e 8 7
e 9 4
e 10 2
e 10 5
e 11 7
```

Figure 10. Benchmark Model

The simulations have been performed on the system whose hardware features were previously presented. The algorithm has been run on a population of 50 individuals, in maximum 10000 iterations, a selection of 50% and a mutation probability of 70%. The simulation results are illustrated in Figure 11 and show that CPU should have at least 30 registers for keeping the memory variables beside some predefined registry such as stack pointer, link register, frame pointer, etc., in order to increase performance.

Mutation plays a central role in the implementation of the genetic algorithm. Because within the mutation operator each vertex is examined and replaced with an available color, if necessary, by applying a sufficiently high mutation probability, the fitness

value will decrease rapidly. Figure 12 shows the effect of varying the mutation rate over the fitness. The results obtained with low mutation probability (20% and 50%) are rather weak compared with those obtained with a mutation of 70%. This is due to the fact that, at the beginning, the fitness decrease rate is not large enough and therefore the genetic algorithm fails to local minima. Also, an extremely high mutation probability does not lead to good solutions because it reduces the convergence ability of the genetic algorithm to any optimal solution. That is why a mutation probability of 90% led to weak fitness value.

Benchmark	Number of vertices	Number of edges	Number of colours
Myciel3.col	11	20	4
Myciel4.col	23	71	5
Myciel5.col	36	236	6
Queen5_5.col	23	160	5
Queen6_6.col	25	290	8
Huck.col	74	301	11
Jean.col	80	256	10
David.col	87	406	11
Games120.col	120	638	9
Miles250.col	128	387	8
Anna.col	138	493	11

Benchmark	Number of vertices	Number of edges	Number of colours
homer.col	561	1629	13
fpsol2.i.1.col	496	11654	65
fpsol2.i.2.col	451	8691	30
fpsol2.i.3.col	425	8688	30
zeroin.i.1.col	211	4100	49
zeroin.i.2.col	211	3541	30
zeroin.i.3.col	206	3540	30
miles1000.col	128	3216	42
queen8_8.col	64	728	10
queen7_7.col	49	476	8

Figure 11. The number of colours resulted on the DIMACS benchmarks

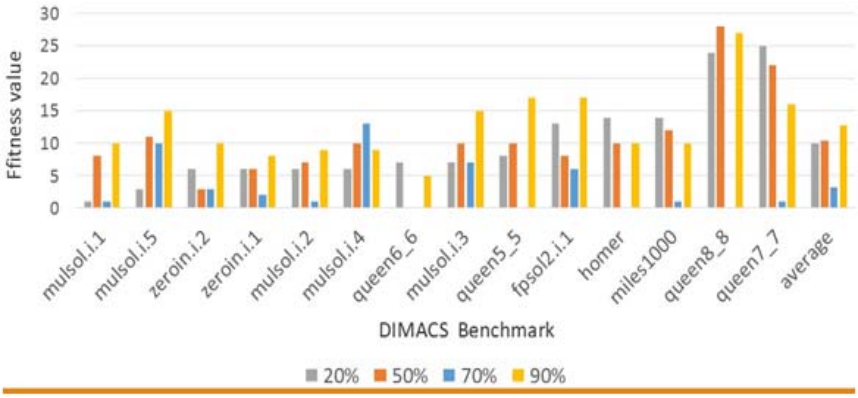


Figure 12. The effect of the mutation probability on the fitness

6. Conclusions and Further Work

The graph coloring problem has extensive applicability and is still topical especially in the registers allocation problem due to the implications regarding energy consumption and heat dissipation. Our developed software tool facilitates understanding the theoretical aspects, allowing students to feel more trustful when solving optimization problems, especially the graph coloring problem. The application can be used by students and researchers with little knowledge in the field of Evolutionary Computing because the user interaction with the system is supported by a graphical interface easily configured and extended.

Although we implemented for the Register Allocation Problem both genetic and swarm-based optimization algorithms, simple PSO algorithm is not enough. Modifications for mapping a discreet value problem into a continuous values space and dynamically choosing the acceleration coefficients based on the fitness function brings significant improvements.

As further work, we are mainly concerned to solve the following issues: to analyze the energy consumption and power dissipation of register allocation, to improve the application that generates the graph interference for more complex code, and to implement and comparatively analyze hybrid graph coloring algorithms.

Another further work idea is to more in depth investigate the accelerated PSO algorithm in order to tune the cognitive and social knowledge coefficients and overcome the limitation of PSO (slow or no convergence on some DIMACS benchmarks). Reusing results from the database where we saved some simulated chromosomes in order to reduce the simulation time and increase the robustness of our developed tool is another issue that we will tackle.

Reference

[1] Florea, A., Gellert, A. (2016). E-learning approach of the graph coloring problem applied to register allocation in embedded systems. *In*: Sixth International Conference on Innovative Computing Technology (INTECH), *p* 173-178, Dublin, Ireland.

[2] Topcuoglu, H.R., Demiroz, B., Kandemir, M. (2007). Solving the Register Allocation Problem for Embedded Systems Using a Hybrid Evolutionary Algorithm. *IEEE Transactions on Evolutionary Computation*, 11 (5), 620-634.

[3] Liu, T., Orailoglu, A., Xue, C.J., Li, M. (2013). Register Allocation for Embedded Systems to Simultaneously Reduce Energy and Temperature on Registers. *ACM Transactions on Embedded Computing Systems*, 13 (3).

[4] Homayoun, H., Gupta, A., Veidenbaum, A., Sasan, A., Kurdahi, F., Dutt, N. (2010). RELOCATE: Register File Local Access Pattern Redistribution Mechanism for Power and Thermal Management in Out-of-Order Embedded Processor. *In:* Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, *p* 216-231, Pisa, Italy.

[5] Huang, Y., Liu, T., Xue, C. J. (2012). Register Allocation for Write Activity Minimization on Non-Volatile Main Memory for Embedded Systems. *Journal of Systems Architecture*, 58 (1), 13-23.

[6] Mahajan, A., Ali, M. S. (2008). Hybrid Evolutionary Algorithm Based Solution for Register Allocation for Embedded Systems. *Journal of Computers*, 3 (6), 59 - 65.

[7] Lee, J.-Y., Cho, S.-I., Park, I.-C. (2005). Performance Enhancement of Embedded Software Based on New Register Allocation Technique. *Microprocessors and Microsystems*, 29, 177-187.

[8] Hindi, M.M., Yampolskiy, R.V. (2012). Genetic Algorithm Applied to the Graph Coloring Problem. *In*: 23rd Midwest Artificial Intelligence and Cognitive Science Conference, *p* 61-66, Cincinnati, Ohio, USA.

[9] Chowdhury, H.A.R., Farhat, T., Khan, M.H.A. (2013). Memetic Algorithm to Solve Graph Coloring Problem. *International Journal of Computer Theory and Engineering*, 5 (6).

[10] Djelloul, H., Layeb, A., Chikhi, S. (2014). A Binary Cuckoo Search Algorithm for Graph Coloring Problem. *International Journal of Applied Evolutionary Computation*, 5 (3).

[11] Bensouyad, M., Saidouni, D.E. (2015). A Hybrid Discrete Flower Pollination Algorithm for Graph Coloring Problem. *In*: International Conference on Engineering & MIS 2015, Istanbul, Turkey.

[12] Dorrigiv, M., Markib, H.Y. (2012). Algorithms for the Graph Coloring Problem Based on Swarm In telligence. *In*: *1*6th CSI International Symposium on Artificial Intelligence and Signal Processing, *p* 473-478, Shiraz, Fars, Iran.

[13] Hertz, A., Zufferey, N. (2006). A New Ant Algorithm for Graph Coloring. *In*: Proceedings of the Workshop on Nature Inspired Strategies for Optimization, Granada, Spain.

[14] Bui, T., Nguyen, T., Patel, C., Phan, K.-A. (2008). An Ant-Based Algorithm for Coloring Graphs. Computational Methods for Graph Coloring and its Generalizations. *Discrete Applied Mathematics*. 156 (2), 190-200.

[15] Lee, Y.-S, Cho, S.-B. (2012). Solving Graph Coloring Problem by Fuzzy Clustering-Based Genetic Algorithm. *In*: 9th International Conference on Simulated Evolution and Learning, *p* 351-360, Hanoi, Vietnam.

[16] Zhao, K., Geng, X., Xu, J. (2015). Solving the Fixed Graph Coloring Problem by Simulated Annealing and Greedy Search. *Journal of Computational and Theoretical Nanoscience*, 12 (4), 637-646.

[17] Titiloye, O., Crispin, A. (2011). Quantum Annealing of the Graph Coloring Problem. *Discrete Optimization*, 8 (2), 376-384.

[18] Kennedy J., Eberhart R. (1995). Particle swarm optimization, *IEEE International Conference on Neural Networks*, 4, *p* 1942-1948. Australia.

[19] Elbeltagi, E., Tarek H., Grierson D. (2005). Comparison among five evolutionary-based optimization algorithms. *Advanced*

Engineering Informatics, 19 (1), 43-53.

[20] Izakian, H., Ladani, B. T., Abraham, A., Snasel, V. (2010). A discrete particle swarm optimization approach for grid job scheduling, *International Journal of Innovative Computing, Information and Control*, 6 (9), 1-15.

[21] Agrawal, J., Agrawal, S. (2015). Acceleration Based Particle Swarm Optimization for Graph Coloring Problem, *Procedia Computer Science* 60, *p* 714-721.

[22] Johnson, D.S., Trick, M. (1996). Cliques, Coloring, and Satisfiability. *In*: Second DIMACS Implementation Challenge. American Mathematical Society.