



MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES
IN PURE AND APPLIED SCIENCES



SOLVING DYNAMIC GRAPH COLORING PROBLEM BY USING A HEURISTIC ALGORITHM

GİZEM SÜNGÜ

MASTER THESIS

Department of Computer Engineering

ADVISOR

Asst. Prof. Dr. Betül Demiröz Boz

ISTANBUL, 2018



MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES
IN PURE AND APPLIED SCIENCES



SOLVING DYNAMIC GRAPH COLORING PROBLEM BY USING A HEURISTIC ALGORITHM

GİZEM SÜNGÜ

(524115014)

MASTER THESIS

Department of Computer Engineering

ADVISOR

Asst. Prof. Dr. Betül Demiröz Boz

ISTANBUL, 2018

MARMARA UNIVERSITY

INSTITUTE FOR GRADUATE STUDIES IN PURE AND APPLIED SCIENCES

Gizem SÜNGÜ, a Master of Science student of Marmara University Institute for Graduate Studies in Pure and Applied Sciences, defended her thesis entitled “**Solving Dynamic Graph Coloring Problem by Using A Heuristic Algorithm**”, on 13.02.2018 and has been found to be satisfactory by the jury members.

Jury Members

Asst. Prof. Dr. Betül DEMİRÖZ BOZ (Advisor)

Marmara University


Prof. Dr. Haluk Rahmi TOPÇUOĞLU (Jury Member)

Marmara University


Assc. Prof. Dr. Fatih Erdoğan SEVİLGİN (Jury Member)

Gebze Technical University


APPROVAL

Marmara University Institute for Graduate Studies in Pure and Applied Sciences Executive Committee approves that Gizem SÜNGÜ be granted the degree of Master of Science in Department of Computer Engineering, Computer Engineering Program on 19.02.2018.....
(Resolution no: 2018/06-02).

Director of the Institute

Prof. Dr. Bülent EKİCİ



ACKNOWLEDGMENT

I would like to express my gratitude to my thesis supervisor, Asst. Prof. Dr. Betul Demiroz Boz, for her inspirational and heartening guidance throughout my graduate study that helped me to harden my motivation for completing this research.

I want to thank Prof. Haluk Topcuoglu, Asst. Prof. Erdoğan Sevilgen and Asst. Prof. Didem Gözüpek for participating in my thesis committee and giving me feedback.

January, 2018

Gizem SÜNGÜ

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGMENT	i
TABLE OF CONTENTS	ii
ÖZET	iv
ABSTRACT	v
SYMBOLS	vi
ABBREVIATIONS	vii
LIST OF FIGURES	viii
LIST OF EQUATIONS	ix
1. INTRODUCTION	1
2. PROPOSED WORK	5
2.1. Dynamic Graphs	6
2.2. Graph Initialization	7
2.3. Graph Generation	8
2.3.1. Node-Dynamic Graph Generation	8
2.3.2. Edge-Dynamic Graph Generation	10
2.4. Population Initialization	11
2.5. Crossover Operation	12
2.5.1. Crossover Methods for Graph Coloring Problem	12
2.5.2. Crossover Methods for Dynamic Graph Coloring Problem	13
2.5.3. Dynamic Pool-Based Crossover Operator	15
2.6. Local Search	21
2.6.1. Related Works About Local Search Methods for Graph Coloring Problem	21
2.6.2. Related Works About Local Search Methods for Dynamic Graph Coloring Problem	22
2.6.3. Local Search Operator	23
2.7. Fitness Calculation	27
2.8. Placement of The Offspring In the Population	29
2.9. Update of Individuals with Changes of Graph	29
3. EXPERIMENTAL STUDY	31
3.1. Node-Dynamic Graphs	32

3.2. Edge-Dynamic Graphs	38
4. CONCLUSION	47
5. FUTURE WORK	48
6. REFERENCES.....	49
RESUME	1

ÖZET

Sezgisel Bir Algoritma Kullanarak Dinamik Grafik Renklendirme Problemi Çözme

Grafik renklendirme problemi literatürdeki en popüler optimization problemlerinden biridir. Problemin grafiklerle modellenen bir çok gerçek probleme uygulanabilmesi, grafik renklendirme problemini önemli kılmaktadır. Problemin polinom zamanda henüz bir çözümünün bulunamaması, bu problem için bir çok sezgisel algoritmanın geliştirilmesine sebep olmuştur. Ancak geliştirilen bu sezgisel algoritmalar dinamik grafiklerdeki renklendirme problemlerine uyum sağlayamamıştır.

Dinamik grafiklerdeki renklendirme problemi dinamik grafik renklendirme problemi olarak adlandırılmış ve bir kaç senedir üzerinde çalışmalar yapılmaya başlanmıştır. Bu sebeple, literatürde bu yeni keşfedilen problem için az sayıda sezgisel algoritma bulunmaktadır.

Bu çalışmada, dinamik grafik renklendirme problemini çözmek amacıyla bir evrimsel algoritma geliştirilmiştir. Algoritma belirlenen bir zaman aralığında değişen dinamik grafikleri dikkate almaktadır ve bu değişimlere kolayca uyum sağlayabilmektedir. Algoritma literatürde yer alan ve dinamik grafik renklendirme problemi için geliştirilen iki sezgisel algoritma ile birlikte çeşitli senaryolara sahip bir çok dinamik grafik üzerinde test edilmiştir ve bu çalışmada sunulan algoritmanın bir çok durumda diğer algoritmalarından daha iyi sonuçlar elde ettiği görülmüştür.

ABSTRACT

Solving Dynamic Graph Coloring Problem by Using A Heuristic Algorithm

Graph coloring problem is one of the most popular optimization problem in the literature. The problem can be applied to solve many real-world problems that are modeled by using graphs. Since graph coloring problem is an NP-hard problem, there are many heuristic algorithms to solve the problem in different domains. However, these heuristic solutions are for solving static graphs and they are hard to be adapted in dynamic graphs.

Graph coloring problem in dynamic graphs is called dynamic graph coloring problem and this problem has been explored for the last few years. Therefore, there are only a few and recently proposed heuristic algorithms to solve the dynamic graph coloring problem in the literature.

In this study, we propose an evolutionary algorithm for solving dynamic graph coloring problem. The algorithm considers dynamic graphs changing over a given number of time steps. It adapts to the changes in the graph with its novel pool-based crossover operator easily. We tested our algorithm with two heuristic methods for dynamic graph coloring problem in the literature on dynamic graphs which have different characteristics and compared the solutions of the algorithms. The results show that our algorithm outperforms these two algorithms in most of the test cases given.

January, 2018

Gizem SÜNGÜ

SYMBOLS

G_t	: Graph at time t
V	: Set of nodes
E	: Set of edges
c_v	: Node probability
c_e	: Edge probability
n	: Number of nodes
t_{min}	: Minimum lifetime of an edge or a node
t_{max}	: Maximum lifetime of an edge or a node
Pop	: Population
S_i	: i^{th} individual in the population
$GraphChangeStep$: Total times of changing graph
$IterationNum$: Number of iterations at each time t
k_i	: Number of color classes of i^{th} individual
C_j^i	: j^{th} color class of i^{th} individual
v_{ua}	: Unassigned node
v_a	: Assigned node
v_p	: The node in the pool
C_{min}	: The color class which has the minimum number of nodes in the individual

ABBREVIATIONS

GCP	: Graph Coloring Problem
DPBEA	: Dynamic Pool Based Evolutionary Algorithm
DGA	: Dynamic Genetic Algorithm
DSATUR	: Degree of Saturation
TABUCOL	: Tabu Search Coloring
SA	: Simulated Annealing
DPBC	: Dynamic Pool Based Crossover
PBC	: Pool Based Crossover
DGCP	: Dynamic Graph Coloring Problem
OX1	: Order 1
OX2	: Order 2
PMX	: Partially mapped Crossover
GPX	: Greedy Partition Crossover
AMPaX	: Adaptive Multi-Parent Crossover
DGC	: Diversification-guided Crossover
GGX	: Grouping-guided Crossover
MGPX	: Multi-parent Crossover
FOO-PARTIALCOL	: Fluctuation Of the Objective-function Partial Coloring
AMACOL	: Adaptive Memory Algorithm Coloring
MACOL	: Memetic Algorithm Coloring
ATS	: Adaptive Tabu Search
DNTS	: Double Neighborhood Tabu Search
IDTS	: Iterated Double Phase Tabu Search

LIST OF FIGURES

<i>Figure 2. 1 – Main Scheme of Pool-Based Evolutionary Algorithm</i>	5
<i>Figure 2. 2 – Initialization of $G_1(V, E)$</i>	7
<i>Figure 2. 3 – Generation of Node-Dynamic Graph at time $t+1$</i>	9
<i>Figure 2. 4 – State of node-dynamic graphs at time steps t and $t+1$</i>	9
<i>Figure 2. 5 – Generation of Edge-Dynamic Graph at time $t+1$</i>	10
<i>Figure 2. 6 – State of edge-dynamic graphs at time steps at t and $t+1$</i>	11
<i>Figure 2. 7 – Population Initialization</i>	12
<i>Figure 2. 8 – Example of OX1 [26]</i>	14
<i>Figure 2. 9 – Dynamic Pool-Based Crossover Operation</i>	16
<i>Figure 2. 10 – Clear Pool</i>	17
<i>Figure 2. 11 – Example of pool based crossover operation according to node-dynamic graph G_t in Figure 2.1 (a)</i>	19
<i>Figure 2. 12 – Results from G_t in Figure 2.1 (a)</i>	Error! Bookmark not defined.
<i>Figure 2. 13 – Results from G_{t+1} in Figure 2.1 (a)</i>	20
<i>Figure 2. 14 – Example of SWAP Operation</i>	22
<i>Figure 2. 15 – Local Search Operation in [39]</i>	23
<i>Figure 2. 16 – Local Search Operation</i>	24
<i>Figure 2. 17 – Example of Pool Based Crossover Operation According to Edge Dynamic Graph G_{t+1} in Figure 2.2 (b)</i>	26
<i>Figure 2. 18 – Example of Local Search Operation According to Edge Dynamic Graph G_{t+1} in Figure 2.2 (b)</i>	27
<i>Figure 2. 19 – Results from G_{t+1} in Figure 2.2 (b)</i>	27
<i>Figure 2. 20 – Placement of Offspring</i>	29
<i>Figure 2. 21 – The best solution of DPBEA for G_t in Figure 2.2 (a)</i>	29
<i>Figure 2. 22 – Adapting DPBEA individual in Figure 2.11 according to the changes between G_t and G_{t+1} in Figure 2.2</i>	30
<i>Figure 2. 23 – Adapting DPBEA individual in Figure 2.11 according to the changes between G_t and G_{t+1} in Figure 2.1</i>	30
 <i>Figure 3. 1 – Varying evolution steps (e) for node-dynamic graphs</i>	 33
<i>Figure 3. 2 – Varying c_v values</i>	34
<i>Figure 3. 3 – Varying p values for node-dynamic graphs</i>	35
<i>Figure 3. 4 – Varying node values</i>	36

<i>Figure 3. 5 – Results of the algorithms from the node-dynamic graph with $n=100$ at each time step t.....</i>	<i>37</i>
<i>Figure 3. 6 – Varying evolution steps (e) for edge-dynamic graphs when $n=100$</i>	<i>40</i>
<i>Figure 3. 7 – Varying evolution steps (e) for edge-dynamic graphs when $n=200$</i>	<i>40</i>
<i>Figure 3. 8 – Varying evolution steps (e) for edge-dynamic graphs when $n=300$</i>	<i>41</i>
<i>Figure 3. 9 – Varying evolution steps (e) for edge-dynamic graphs when $n=400$</i>	<i>41</i>
<i>Figure 3. 10 – Varying c_e values when $n=200$.....</i>	<i>42</i>
<i>Figure 3. 11 – Varying c_e values when $n=300$.....</i>	<i>43</i>
<i>Figure 3. 12 – Varying c_e values when $n=500$.....</i>	<i>43</i>
<i>Figure 3. 13 – Varying p values for edge-dynamic graphs when $n=100$.....</i>	<i>44</i>
<i>Figure 3. 14 – Varying p values for edge-dynamic graphs when $n=200$.....</i>	<i>44</i>
<i>Figure 3. 15 – Varying p values for edge-dynamic graphs when $n=400$.....</i>	<i>45</i>
<i>Figure 3. 16 – Varying p values for edge-dynamic graphs when $n=500$.....</i>	<i>45</i>
<i>Figure 3. 17 – Results of the algorithms when an edge-dynamic graph is becoming fully connected step by step</i>	<i>46</i>

LIST OF EQUATIONS

<i>Equation 2. 1 – Computation of fitness function</i>	<i>28</i>
--------------------------------------------------------------	-----------

1. INTRODUCTION

Graph coloring problem (GCP) is a well-known optimization problem. The problem can be described with an undirected graph $G(V, E)$ which has a set of vertices (nodes) $V = \{v_1, v_2, \dots, v_n\}$ where n denotes number of nodes in the set and a set of edges $E \subset V \times V$ which contains edges between any two nodes v_x and v_y that exist in V , where $x \neq y$. Graph coloring problem (GCP) colors the nodes with the rule that any two nodes that are connected by an edge do not have the same color. Main objective of the problem is to minimize number of different colors used in the given graph. GCP is proven as NP-hard problem [1].

GCP is applicable for many real-world problems that can be modeled by using static graphs such as time tabling and scheduling [2, 3], frequency assignment [4], register allocation [5, 6, 7, 8] circuit testing [9] and many others. GCP is specialized according to components of these problems and there are many studies to solve the problems in the literature.

These studies can be separated into two kinds of approaches. The first approach has exact algorithms such as [10, 11, 12] that are preferred to use for small graphs. The exact algorithms are successful to find the best solutions of small graphs whereas they spend too much computation time to find the best solutions of large graphs. Hence, there are a lot of heuristic algorithms as the another approach to solve GCP in large graphs such as greedy [13], local search [14], genetic [15] and evolutionary algorithms [16].

The first heuristic approach for GCP is Degree of Saturation(DSATUR) [13] which is still one of the most powerful algorithms for the problem. DSATUR sorts the nodes in a given graph according to their degrees and colors them starting from the node having the maximum number of degrees. Since the method builds only one solution with its distinct rules, its search area is small to explore other solutions for the given graph.

Local search is also one of the oldest approaches for GCP in the literature and it has larger search area than DSATUR. The method basically improves a given solution in a predefined number of iterations. The first metaheuristic as local search method for GCP is proposed in [17] and continued by many studies such as TABUCOL [14], FOOPARTIALCOL [18], AMACOL [19], MACOL [20], ATS [21], DNTS [22] and IDTS

[23]. This operator searches neighborhood solutions of a given individual and tries to find the best neighborhood solution of the given individual according to the considered graph. However, if the given graph has a large number of instances with its nodes and edges, i.e., for 500 nodes, there are many neighborhood solutions for the graph. Since the given individual may be so far from its best neighborhood solution, local search may spend long computation time to find the best solution. In this situation, using “pure” local search for GCP becomes a poor approach and it should be combined with a recombination heuristic operator to use it efficiently [24]. In order to solve the bottleneck of local search methods, evolutionary algorithms have been explored and improved for GCP in the literature.

Evolutionary algorithms are heuristic methods that are blended of local search operators and specialized recombination operators (crossover) for GCP. The algorithms use two given solutions (parents) and recombine them to generate a new solution (offspring). A local search method is applied to the new solution for improving. This process repeats for a predefined number of iteration steps. Thanks to these iterations, evolutionary algorithms get closer to the best solution step by step in a desired computation time.

The first evolutionary algorithm [25] is developed with a crossover that is named order-based crossover that uses order-based represented individuals. After that, more crossover operators with order-based approach for evolutionary algorithms, have been proposed in [26]. Since evolutionary algorithms with order-based approach have not been more successful than pure local search, techniques of crossover operators are improved and *color-oriented crossover (partition-based crossover)* is established [16]. This crossover method that uses partition-based represented individuals, outperforms the order-based crossover and evolutionary algorithms with partition-based crossover becomes the most popular approach for GCP. Finally, these evolutionary algorithms are specialized for many real-world problems based on GCP such as [6, 27] and many others, thanks to its adaptation.

The purposed heuristic algorithms in the literature are widely used and improved for GCP problems which are modeled by static graphs. However, there are also many real-world graph coloring problems that change over time [28] and these problems can be modeled by dynamic graphs such as crew scheduling [29], dynamic resource allocation

[30]. Dynamic GCP (DGCP) has been studied for the last few years and only a few solutions are proposed by using graph theory [31] and heuristic algorithms [32, 33].

In this study, we propose an evolutionary algorithm for DGCP. Each individual in the population is represented with partition-based method [16] having nonconflicting nodes. When the graph changes, some of the nodes or the edges are added or removed from the graph and our algorithm can easily adopt to these changes. When deleting the nodes from the graph, the nodes are removed from their partitions in the individual without changing the current non-conflicting groups. When adding new nodes to the graph, new partitions (color classes) are added to the individual for each newly added node. In case of adding edges between the nodes, the algorithm checks the endpoints (nodes) of each newly added edge whether these nodes are in the same partition of the individual or not. If they are, then the algorithm separates the nodes by adding a new partition for one of them. Our algorithm is able to keep the valuable information obtained in history and reshape this information with the current state of the graph. The number of partitions represent the number of colors used to color the graph, and the solution quality of each individual is different so the number of partitions in each representation is also dynamic.

We propose a highly specialized and novel crossover operator that can easily deal with the dynamic representation of the individuals. It targets to maximize the number of non-conflicting nodes in the graph and place them to the same partition. The nodes having conflicts can not be directly placed in a partition so a pool is proposed to keep these nodes and place them to the most appropriate partition as soon as possible. As a result, the proposed pool-based crossover operator can easily adopt to the dynamic changes of the graph. When we try to maximize the number of non-conflicting nodes in the partitions, we are also decreasing the search area, so to increase the diversity of the solutions in the population, we propose a local search method for checking the neighborhood solutions.

We conduct experiments with dynamic graphs to test the effectiveness of the proposed solution. The performance of our solution is compared with Degree of Saturation (DSATUR) [13] which is a well known and efficient greedy heuristic for solving the graph coloring problem and DGA [32] which is the first and recently published genetic algorithm that solves the dynamic graph coloring problem. DGA

proposes a dynamic population that includes individuals with permutation based representation. It uses a standart crossover operator OX1 [26] and a mutation operator SWAP [34]. They mainly concentrate on the dynamics of the problem and dynamics of the algorithm and proposed populations suitable for dynamic graph coloring problem, so their genetic algorithm is pure and straightforward. Our experimental evaluation indicates that we have outperformed both algorithms from the literature.

2. PROPOSED WORK

In the dynamic graph coloring problem, the dimension time is added to the graph so it changes over time. In our approach, only the current state of the graph is known and the solution from evolutionary algorithm is generated according to this state. The problem representation and operators of our algorithm are designed such that it can adopt to the dynamic changes of the graph.

Input: Node probability c_v , edge density p , edge probability c_e , initial graph size n , minimum lifetime of an edge or a node t_{\min} , maximum lifetime of an edge or a node t_{\max} , number of iterations for each graph at time t *IterationNum*, total time of changing graph G_t *GraphChangeStep*, size of population *PopSize*.

Output: The best solution for each G_t .

Initialization: Input graph $G_1 \leftarrow \emptyset$, initial population $\text{Pop} \leftarrow \emptyset$, an offspring $S_0 \leftarrow \emptyset$.

1. $G_1 \leftarrow \text{InitializeGraph} (n, t_{\min}, t_{\max}, p)$
2. $\text{Pop} \leftarrow \text{InitializePopulation} (G_1, \text{PopSize}, n)$
3. **for** $t \leftarrow 1$ **to** *GraphChangeStep* **do**
4. **for** $i \leftarrow 1$ **to** *IterationNum* **do**
5. Select two parents S_1 and S_2 from Pop .
6. Calculate number of color classes of S_1 and S_2 as k_1 and k_2
7. $S_0 \leftarrow \text{CrossoverOperation} (G_t, S_1, S_2, k_1, k_2)$
8. $S_0 \leftarrow \text{LocalSearch} (G_t, S_0)$
9. $\text{Pop} \leftarrow \text{UpdatePop} (S_1, S_2, S_0)$
10. **end for**
11. **if** G_t is an edge-dynamic graph **then**
12. Initialize a set for added edges E^+ : $E^+ \leftarrow \emptyset$
13. $G_{t+1}, E^+ \leftarrow \text{GenerateEdgeDynamic} (G_t, n, c_e, p, t_{\min}, t_{\max})$
14. Update the individuals in Pop according to E^+
15. **else**
16. Initialize two sets for added and removed nodes: $V^+ \leftarrow \emptyset, V^- \leftarrow \emptyset$
17. $G_{t+1}, V^+, V^- \leftarrow \text{GenerateNodeDynamic} (G_t, n, c_v, p, t_{\min}, t_{\max})$
18. Update the individuals in Pop according to V^+ and V^-
19. **end if**
20. **end for**

Figure 2. 1 – Main Scheme of Pool-Based Evolutionary Algorithm

The main scheme of our evolutionary algorithm is shown in Figure 2.1. The algorithm starts to create an initial graph with a predefined number of nodes n at time step $t=1$. According to the initial graph, an initial population is created. Each individual in the initial population has n number of color classes and all of them have the worst fitness value at the beginning.

After the initialization step, the algorithm repeats the following process with a given number of time steps as *GraphChangeStep* in order to obtain the best solution according to its fitness value for each graph that is generated at time step t as G_t . At each time step t ($1 \leq t \leq \text{GraphChangeStep}$), a predefined number *IterationNum* of offsprings are

produced by using individuals from the population. At each iteration step i , ($1 \leq i \leq IterationNum$), two parents are selected randomly from the population. The crossover and local search methods are applied to these two parents and a new offspring is generated. After the new offspring is obtained, the fitness value of the offspring is calculated. The offspring is always replaced with the parent having the worst fitness value.

When the algorithm reaches $IterationNum$, the graph of the next time step $t+1$ as G_{t+1} is generated. If G_t is an edge-dynamic graph, the edges in G_t which reached the end of their lifetimes, are removed from G_{t+1} and new edges are added to G_{t+1} . If G_t is a node-dynamic graph, the nodes in G_t which reached the end of their lifetimes are removed from G_{t+1} and new nodes are added to G_{t+1} . All individuals in the population should be adapted to the changes in G_{t+1} before starting iterations of the next time step $t+1$. In the case of node-dynamic, each node removed from G_{t+1} are also removed from the individuals, and a new color class is created for each new node in G_{t+1} in all of the individuals in the population. In case of edge-dynamic, only newly generated edges are considered. Each two nodes which are two sides of each new edge, are checked if they are in the same color class or not in each individual. If it is yes, one of them is removed from the color class and added a newly created color class in the individual. After these revisions, the population is ready for the next time step $t+1$.

In the following subsections, the components of our algorithms are detailed.

2.1. Dynamic Graphs

A dynamic graph of dynamic graph coloring problem is created at initial time step $t=1$ and exists during the given set of time steps T where $T = \{1, 2, \dots, GraphChangeStep\}$. At each time step t , ($1 < t \leq GraphChangeStep$), some components of a dynamic graph are changed after the graph is created according to given input parameters.

Dynamic graphs are specialized and varied based on their changing components. According to this, there are five dynamic graph models that are described in [35]. Two types of these dynamic graphs are considered for DGCP in this study as *edge-dynamic graphs* and *node-dynamic graphs*. Each graph is shown as $G_t(V, E)$ where V and E are the set of nodes and the set of edges that exist at time step t , ($1 \leq t \leq GraphChangeStep$), respectively.

2.2. Graph Initialization

The general procedure of this study begins with creating an input graph to describe a given dynamic graph coloring problem. This graph is initialized at time step $t=1$ as the first graph $G_1(V, E)$ of various dynamic graphs that are derived from each other in a given number of time steps as *GraphChangeStep*. $G_1(V, E)$ has a set of nodes V whose size is a predefined number n , and a set of edges which are created randomly between the nodes in V with an edge density p .

Input: Initial graph size n , minimum lifetime of a node t_{min} , maximum lifetime of a node t_{max} , edge density p

Output: Input graph $G_1(V, E)$

Initialization: An empty node set $V \leftarrow \emptyset$, an empty edge set $E \leftarrow \emptyset$, input graph $G_1(V, E) \leftarrow \emptyset$

1. **for** $i \leftarrow 1$ **to** n **do**
2. Create a new node v_i and add v_i to V
3. **if** Is G_1 a node-dynamic graph **then**
4. Set a lifetime t_{v_i} that is generated randomly between t_{min} and t_{max}
5. **end if**
6. **for** $j \leftarrow i-1$ **to** 1 **do**
7. Generate a random number $rand$ between 0 and 1
8. **if** $rand < p$ **then**
9. Generate an edge between i^{th} node v_i and j^{th} node v_j
10. **if** Is G_1 an edge-dynamic graph **then**
11. Set a lifetime t_e that is generated randomly between t_{min} and t_{max}
12. **end if**
13. Add the edge to E
14. **end if**
15. **end for**
16. **end for**

Figure 2. 2 – Initialization of $G_1(V, E)$

The algorithm in Figure 2.2 to initialize G_1 takes some input parameters as a predefined node size n , the minimum life time of a node t_{min} , the maximum lifetime of a node t_{max} and an edge density p . At the initialization part, empty node set V and edge set E are created so G_1 is composed with V and E . The algorithm creates nodes and their edges in $G_1(V, E)$ step by step. At each step i ($1 \leq i \leq n$), a new node v_i is created and added to V . If a node-dynamic graph is initialized, a lifetime is set randomly between t_{min} and t_{max} for v_i . Throughout this lifetime, v_i exists on the graph. Otherwise, v_i stays on the graph during the existence of the graph. Edges between v_i and other nodes that are created previously in V are generated by using p (see in Figure 2.2). If the initialized graph is edge dynamic, a lifetime is set randomly between t_{min} and t_{max} for each edge. Throughout this lifetime, the edge exists on the graph. Otherwise, lifetime of the edge depends on

existences of its endpoints. This process is repeated until n nodes is generated with their edges.

2.3. Graph Generation

In order to generate a dynamic graph at time step t ($t > 1$), the graph that is generated at previous time $t-1$ is referred and the related components of the graph are changed depending on some parameters. These parameters and their descriptions are given at Section 3.

In type of node-dynamic graph model, the nodes in a given graph are dynamically changed with adding and removing operators in a given number of time steps *GraphChangeStep*. Their edges are also dynamically added or removed when the nodes are added or removed respectively.

In type of edge-dynamic graph model, the nodes that are created at time step $t=1$ are not removed from the graph during a given number of time steps *GraphChangeStep*. New nodes are also not added to the graph in the next time steps. Whereas the nodes are protected throughout existence of the graph, edges in the graph are dynamically changed with adding and removing with some input parameters which are predefined values, at each time step t ($1 < t \leq \text{GraphChangeStep}$).

2.3.1. Node-Dynamic Graph Generation

In this graph model, set of nodes V are changed with a given number of time steps. At each time step t , the nodes are in the graph $G_t(V, E)$ are checked and their lifetimes are decreased by one. The ones that have reached the end of their lifetimes are removed with their edges from $G_t(V, E)$. At the same time step, new nodes are added to $G_t(V, E)$ with graph change rate c_v . A lifetime is set randomly to each newly added node between two parameters t_{min} and t_{max} . Number of added nodes are determined with multiplying initial number of nodes n and c_v , $n \times c_v$. When each new node is added, all of the existing nodes on $G_t(V, E)$ are examined and a new edge is created between the newly added node and an existing node with an edge density p . Generating the node-dynamic graph for the next time step $t+1$ is detailed in Figure 2.3.

An example of how to change a node-dynamic graph between two time steps t and $t+1$ is shown in Figure 2.4 (a) and Figure 2.4 (b) respectively. At time step t , G_t has 15

nodes. At time step $t+1$, after decreasing lifetimes of all nodes by one, node₁ has reached the end of its lifetime. Therefore, node₁ is removed with its edges $edge(1, 2)$, $edge(1, 3)$, $edge(1, 5)$, $edge(1, 6)$ and $edge(1, 11)$. At the addition part, node₁₅ is added according to c_v and its edges with node₂ and nodes₅ are created with using density p . These example graphs are also used to describe our algorithm and to compare its performance with the algorithm from literature in the further sections.

Input: The dynamic graph G_t that is generated at time t , initial graph size n , node probability c_v , edge density p , minimum lifetime of a node t_{min} , maximum lifetime of a node t_{max}
Output: The dynamic graph G_{t+1} that is generated at current time $t+1$, set of added nodes V^+ , set of removed nodes V^-
Initialization: $G_{t+1} \leftarrow G_t$, $V^+ \leftarrow \emptyset$, $V^- \leftarrow \emptyset$, number of current nodes $N \leftarrow |V|$

1. **for** each node v in G_{t+1} **do**
2. Decrease the lifetime of v : $t_v \leftarrow t_v - 1$
3. **if** $t_v = 0$ **then**
4. Remove v and its edges from G_{t+1}
5. Add v to V^- : $V^- \leftarrow V^- \cup v$
6. **end if**
7. **end for**
8. Set number of nodes n_{added} that will be added to G_{t+1} : $n_{added} \leftarrow n \times c_v$
9. **for** $i \leftarrow 1$ **to** n_{added} **do**
10. Create a new node v_{new} and add v_{new} to G_{t+1}
11. Set a lifetime that is generated randomly between t_{min} and t_{max} for v_{new}
12. Add v_{new} to V^+ : $V^+ \leftarrow V^+ \cup v_{new}$
13. **end for**
14. Set number of nodes in G_{t+1} as N : $N \leftarrow N + n_{added}$
15. **for** $i \leftarrow 1$ **to** $N - 1$ **do**
16. **for** $j \leftarrow i + 1$ **to** N **do**
17. Generate a random number $rand$ between 0 and 1
18. **if** $rand < p$ **then**
19. Create an edge between i^{th} node v_i and j^{th} node v_j in G_{t+1}
20. **end if**
21. **end for**
22. **end for**

Figure 2. 3 – Generation of Node-Dynamic Graph at time $t+1$

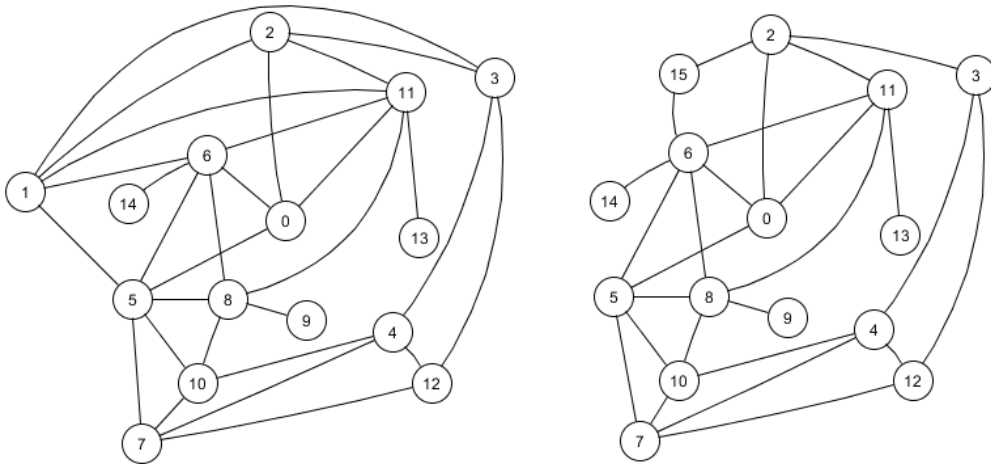


Figure 2. 4 – State of node-dynamic graphs at time steps t and $t+1$

2.3.2. Edge-Dynamic Graph Generation

In this graph model, set of edges E is changed with a given number of time steps and set of nodes V is not changed during the existence of the given graph. At each time step t , the edges in the graph $G_t(V, E)$ are checked and their lifetimes are decreased by one. The edges that have reached the end of their lifetimes are removed from the set of edges E of $G_t(V, E)$. At the same time step, $n \times (n-1) \times p \times c_e \div 2$ new edges are added to $G_t(V, E)$ with edge probability c_e until $G_t(V, E)$ does not become a fully connected graph. A lifetime is set randomly to each newly added edge between two parameters t_{\min} and t_{\max} . Generation of an edge-dynamic graph for the next time step $t+1$ is detailed in Figure 2.5.

Input: The dynamic graph G_t that is generated at previous time t , initial graph size n , edge probability c_e , edge density p , minimum lifetime of a node t_{\min} , maximum lifetime of a node t_{\max}
Output: The dynamic graph G_{t+1} that is generated at current time $t+1$, set of added edges E^+
Initialization: $G_{t+1}(V, E) \leftarrow G_t(V, E)$, $E^+ \leftarrow \emptyset$

1. **for** each edge e in E of G_{t+1} **do**
2. Decrease the lifetime of e : $t_e \leftarrow t_e - 1$
3. **if** $t_e = 0$ **then**
4. Remove e from G_{t+1} : $E \leftarrow E / e$
5. **end if**
6. **end for**
7. Set number of edges e_{added} that will be added to G_{t+1} : $e_{\text{added}} \leftarrow n \times (n-1) \times p \times c_e \div 2$
8. **for** $i \leftarrow 1$ **to** e_{added} **do**
9. **if** Is G_{t+1} fully connected **then**
10. **break**
11. **end if**
12. Select two nodes v_x and v_y /exists G_{t+1}
13. **while** $e(v_x, v_y)$ in G_{t+1} **do**
14. Select two nodes v_x and v_y /exists G_{t+1}
15. **end while**
16. Create a new edge between v_x and v_y $e(v_x, v_y)$
17. Add $e(v_x, v_y)$ to E of G_{t+1} : $E \leftarrow E \cup e(v_x, v_y)$
18. Set a lifetime that is generated randomly between t_{\min} and t_{\max} for $e(v_x, v_y)$
19. Add $e(v_x, v_y)$ to E^+ : $E^+ \leftarrow E^+ \cup e(v_x, v_y)$
20. **end for**

Figure 2. 5 – Generation of Edge-Dynamic Graph at time $t+1$

An example of how to change an edge-dynamic graph between two time step t and $t+1$ is shown in Figure 2.6 (a) and Figure 2.6 (b) respectively. At two time steps, G_t and G_{t+1} have same 15 nodes. At time step t , G_t has 28 edges but at time step $t+1$, after decreasing lifetimes of all edges by one, 3 edges have reached the end of their lifetimes. Hence, $\text{edge}(1, 10)$, $\text{edge}(6, 12)$ and $\text{edge}(8, 12)$ are removed from the graph. At the addition part, 4 edges which are $\text{edge}(7, 13)$, $\text{edge}(8, 11)$, $\text{edge}(7, 11)$, $\text{edge}(3, 5)$ are added

according to c_e . These graphs are also used to show the performances of our algorithm and the algorithms from literature in the further sections.

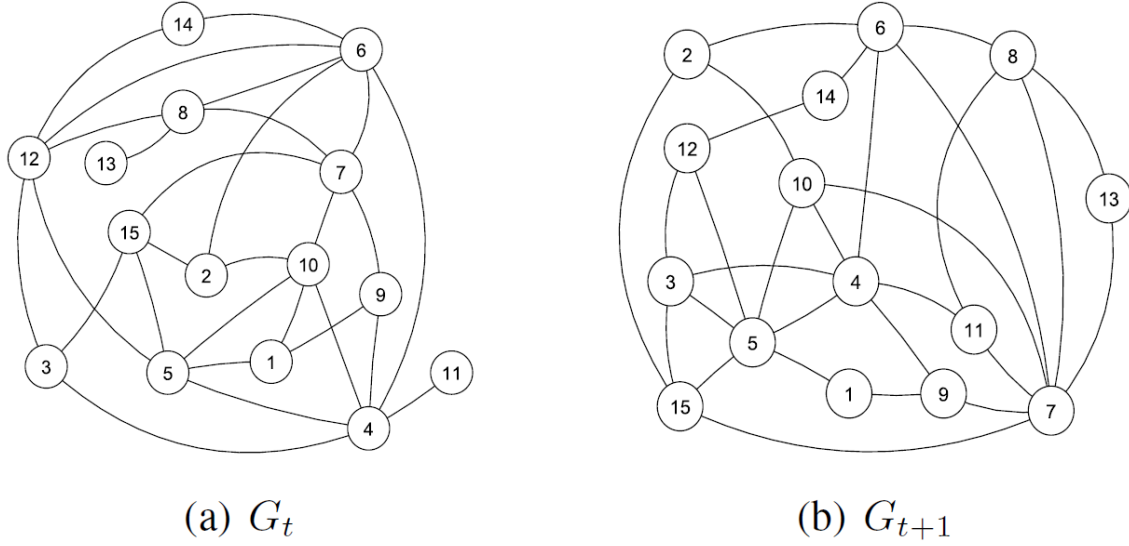


Figure 2. 6 – State of edge-dynamic graphs at time steps at t and $t+1$

2.4. Population Initialization

Initial population Pop has a predefined number $popSize$ of individuals as $Pop = \{S_1, S_2, \dots, S_{popSize}\}$. Each individual S_i where $i = 1, \dots, popSize$, contains k color classes as $S_i = \{C_1, C_2, \dots, C_k\}$ and k is not fixed. Each node v in the input graph G_1 is mapped to a color class with respect to the rule that no two nodes in a same color class are connected by an edge, i.e., for all u, v in C_i ($i = 1, \dots, k$), $edge(u, v)$ not in E . Thus, the node v is named as a *conflict free* node. If two nodes v and u which are connected by an edge $edge(u, v)$ in E , are assigned the same color class, these nodes are called *conflicting* nodes and their color class is also a *conflicting* color class. In GCP, if a solution with k color classes has no conflicting nodes, then this solution is said *legal*. In our study, all individuals in our entire algorithm are *legal*.

To obtain the initial population with the features mentioned above, the algorithm in Figure 2.7 is executed by using the input graph G_1 and number of nodes n in G_1 . For each individual S_i , each node v in G_1 is put into a different color class of S_i respectively so number of color classes of S_i equals to number of nodes in G_1 ($k = n$). However, k will be no longer equal to n when grouping conflict free nodes in a same color by using crossover

and local search operations starts. After all nodes in G_1 are placed to S_i , the color classes of S_i are shuffled to get a different individual. This process continues until $popSize$ individuals are created.

Input: The initial graph G_1 that is generated at time $t=1$, size of population $PopSize$, initial graph size n
Output: Initial population Pop which is a list of $PopSize$ number of parents
Initialization: $Pop \leftarrow \emptyset$

1. **for** $t \leftarrow 1$ **to** $PopSize$ **do**
2. Create a new parent S_i : $S_i \leftarrow \emptyset$
3. **for** each node v exists on G_1 **do**
4. Create a new color C and put v in C : $C \leftarrow C \cup v$
5. Put C in S_i : $S_i \leftarrow S_i \cup C$
6. **end for**
7. Shuffle indexes of the colors in S_i
8. Put S_i in Pop : $Pop \leftarrow Pop \cup S_i$
9. **end for**

Figure 2. 7 – Population Initialization

The aim of to generate an initial population with this method is to increase the population (Pop) diversity and a fairness between DPBEA and the other algorithms DGA and DSATUR with respect to their representations of individuals.

2.5. Crossover Operation

A crossover operation is the most efficient part of a population-based evolutionary algorithm. In general, a crossover operation uses two parents taken from its population to produce an offspring though a method. This method differentiates a crossover operation than other crossover operations in the literature.

2.5.1. Crossover Methods for Graph Coloring Problem

For graph coloring problem, there are many crossover methods that proposed in the literature. Some crossover methods consider permutation-based individuals [26] which are named as edge, order 1 (OX1), order 2 (OX2), position, partially mapped (PMX), and cycle crossover. Most of these methods combines two parents to generate one or two offsprings regardless of edges between the nodes in a given graph. After the crossover operations, the nodes in the obtained offsprings are colored according to their permutations with respect to edges between these nodes. For this reason, number of colors used to color the individuals k is not fixed and all individuals have legal coloring.

Another crossover operations are based on partition method [16] which is more efficient way to solve graph coloring problem. In general, these crossover methods combine partitions (color classes) of two or more parents to generate one or more offsprings. The most well-known partition approach is Greedy Partition Crossover (GPX) [16]. GPX is applied to two parents (not necessarily legal colorings) which have a fixed number k of color classes to generate an offspring in k steps. At each step i ($1 \leq i \leq k$), GPX considers one of two parents respectively and chooses the color class which has the maximum number of nodes of the considered parent. The subset of the chosen color is transmitted to the next color class of the offspring. The most of crossover operations for graph coloring problem are extended or improved versions of GPX such as Adaptive Multi-Parent Crossover(AMPaX) [20], Diversification-guided Crossover (DGX) and Grouping-guided Crossover (GGX) [23], Pool-Based Crossover (PBC) [5], MGPX [22], Well-Informed Partition Crossover [36]. Since all these studies are suitable for static graph coloring problem, they should be extended in case of solving dynamic graph coloring problem.

2.5.2. Crossover Methods for Dynamic Graph Coloring Problem

Since a dynamic graph is a predefined number of static graphs, most of the crossover methods in the literature can be used for dynamic graph coloring problem. These crossover methods can be improved for dynamic graph coloring problem if their concepts can adapt to dynamic graphs that change during a number of times by adding and removing nodes or edges, easily. In this way, the permutation-based crossover operator OX1 [26] is used to combine parents in [32] which is the first and recently published genetic algorithm for dynamic graph coloring problem. Figure 2.8 shows how to work OX1 operator on two parents in order to generate an offspring step by step.

- Generate two random crossover points on the two parents to obtain 3 substrings in each parent.
- Transmit the middle substring in the 1st parent to the middle substring in the offspring and assign the transmitted nodes as *used* (the red nodes) in each parent.
- Place the remaining (unused) nodes in 2nd parent to the offspring one by one. Transmit each node starting from left to right according to the sequence of 2nd

parent in order to fill the 3rd and 1st substrings in the offspring respectively.

- Encode each node in the offspring with a color starting from left to right of the sequence with respect to that conflicting two nodes according to the given graph in Figure 2.4 (a) have different colors.

Step 0

1st Parent: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14


 2nd Parent: 2 5 14 11 12 3 8 6 1 4 10 9 7 13 0

Step 1

1st Parent: 0 1 2 **3 4 5 6 7 8 9** 10 11 12 13 14


 2nd Parent: 2 **5** 14 11 12 **3 8 6** 1 **4** 10 **9 7** 13 0

Offspring: 3 4 5 6 7 8 9

Step 2

1st Parent: **0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**


2nd Parent: **2 5 14 11 12 3 8 6 1 4 10 9 7 13 0**

Offspring: 10 13 0 3 4 5 6 7 8 9 2 14 11 12 1

Step 3

Offspring: 10 13 0 3 4 5 6 7 8 9 2 14 11 12 1



Figure 2. 8 – Example of OX1 [26]

2.5.3. Dynamic Pool-Based Crossover Operator

In this study, we proposed a novel crossover operator Dynamic Pool-Based Crossover Operator (DPBC) which increase the diversity of search area while creating color classes of the offspring. DPBC is an improved version of Pool-Based Crossover Operator (PBC) which is a crossover operator that we proposed in [5] for static and node-weighted graph coloring problem. PBC operator combines a fixed number of color classes of two parents based on degrees and weights of nodes in a given graph. However, the graphs in this work are unweighted-node graphs and they are changed dynamically.

DPBC operator is explained in Figure 2.9. The algorithm takes the graph which is generated time t G_t , two parents as S_1 and S_2 , number of color classes of S_1 k_1 and number of color classes of S_2 k_2 . The algorithm obtains an offspring S_0 with its number of color classes k that can be different from numbers of color classes of its parents k_1 and k_2 . The algorithm starts with creating an empty pool $Pool$ and marking all color classes of two parents, C^1_i in S_1 ($1 \leq i \leq k_1$) and C^2_i in S_2 ($1 \leq i \leq k_2$), as *unselected*. Each node v in these color classes is marked as a *unassigned* node v_{ua} . After that, each color class of S_0 is generated step by step. At each step i ($1 \leq i \leq k$), i^{th} color class of S_0 is set as empty.

Two *unselected* color classes from S_1 and S_2 as C^1_x and C^2_y are selected randomly and they are marked as *selected* color classes. Each *unassigned* node v_{ua} in $C^1_x \cup C^2_y$ is put into C_i and it is marked as an *assigned* node v_a . If there are nodes in $Pool$ from previous steps, all nodes in $Pool$ are also put into C_i . After a node set is obtained in C_i , the main characteristic procedure of DPBC is executed until C_i becomes a conflict-free color class: the maximum conflicting node v_{max} is calculated according to G_t and v_{max} is moved from C_i to $Pool$. When k steps is finished and the offspring with k number of color classes is generated, there can be still one or more nodes in $Pool$ because of the conflicts. If $Pool$ is not empty, the algorithm executes Figure 2.10 to search a suitable existing color class or to create a new color class for each node in $Pool$. The algorithm in Figure 2.10 gets the graph G_t , number of color classes k , the offspring S_0 and the pool $Pool$ from the algorithm in Figure 2.9 for this aim. For each node v_p in $Pool$, a variable *isPlaced* is set as *false* to control v_p is placed an existing color class of S_0 or not. Each color class C_i ($1 \leq i \leq k$) of S_0 is traced for v_p . If there is no node conflicting with v_p in C_i , v_p is moved from $Pool$ to C_i , *isPlaced* becomes *true* and the algorithm continues to search the color

classes of S_0 for the next node in *Pool*. If *isPlaced* is still false at the end of searching all color classes in S_0 for v_p , a new color class is created in S_0 and v_p is put into the new color class. In this case, number of color classes k is increased by one. After all nodes in *Pool* are placed in S_0 , the algorithm in Figure 2.9 obtains S_0 with an updated number of its color classes k .

<p>Input: Graph G_t, 1st parent $S_1 = \{C^1_1 C^1_2, \dots, C^1_k\}$, 2nd parent $S_2 = \{C^2_1 C^2_2, \dots, C^2_k\}$, number of color classes of S_1 k_1, number of color classes of S_2 k_2</p> <p>Output: An offspring $S_0 = \{C_1 C_2, \dots, C_k\}$</p> <ol style="list-style-type: none"> 1. Create an empty pool $Pool \leftarrow \emptyset$ 3. for $i \leftarrow 1$ to k_1 do 4. Mark C^1_i as <i>unselected</i> 5. for each node v in C^1_i do 6. Mark v as <i>unassigned</i> node v_{ua} 7. end 8. end 9. for $i \leftarrow 1$ to k_2 do 10. Mark C^2_i as <i>unselected</i> 11. for each node v in C^2_i do 12. Mark v as <i>unassigned</i> node v_{ua} 13. end 14. end 15. Set number of combinations k between the parents to create S_0 16. $k \leftarrow \min(k_1, k_2)$ 17. for $i \leftarrow 1$ to k do 18. Set i^{th} color class of S_0 C_i: $C_i \leftarrow \emptyset$ 19. Select an <i>unselected</i> color class C^1_x from S_1 20. Select an <i>unselected</i> color class C^2_y from S_2 21. Mark C^1_x and C^2_y as <i>selected</i> 21. for each unassigned node v_{ua} in $C^1_x \cup C^2_y$ do 22. Put v_{ua} into C_i: $C_i \leftarrow C_i \cup v_{ua}$ 23. Mark v_{ua} as <i>assigned</i> node v_a 24. end 25. if $Pool \neq \emptyset$ then 26. Put each node in $Pool$ v_p into C_i: $C_i \leftarrow C_i \cup v_p$ 27. Remove v_p from $Pool$: $Pool \leftarrow Pool / v_p$ 28. end 29. while C_i is not <i>conflict free</i> do 30. Calculate the maximum conflicting node as v_{max} using G_t 31. Throw v_{max} into $Pool$: $Pool \leftarrow Pool \cup v_{max}$ 32. Remove v_{max} from C_i: $C_i \leftarrow C_i / v_{max}$ 33. end 34. end 35. if $Pool \neq \emptyset$ then 36. $k, S_0 \leftarrow ClearPool(G_t, k, S_0, Pool)$ /* Figure 2.10 */ 37. end

Figure 2. 9 – Dynamic Pool-Based Crossover Operation

An application of DPBC is shown in Figure 2.11 by using the given graph at time t in Figure 2.4 (a). The example is taken from the 2000th iteration step of the evolution

for the graph. In the first step of the example, the first color class of the first parent C^1_1 and the third color class of the second parent C^2_3 are selected randomly and their nodes 5, 2, 12, 6 and 10 are put into the first color class of the offspring C_1 . Conflicts between the nodes in C_1 are calculated according to graph in Figure 2.4 (a) and 5 is thrown to the pool as the maximum conflicting node. C_1 becomes a conflict free color class without 5 so the first step is finished with obtaining C_1 and an unempty pool. The nodes in C_1 and the pool are removed from the parents in order not to use again.

<p>Input: Graph G_t, number of color classes of k, an offspring $S_0 = \{C_1 C_2, \dots, C_k\}$, the pool $Pool$</p> <p>Output: An offspring $S_0 = \{C_1 C_2, \dots, C_k\}$</p> <pre> 1. for each node v_p in $Pool$ do 2. Set a variable $isPlaced$ for the state of v_p: $isPlaced \leftarrow false$ 3. for each color C_i in S_0, $i \leftarrow 1, \dots, k$ 4. Set C_i as a conflict-free color for v_p: $CF \leftarrow true$ 5. for each node v in C_i do 6. if $e(v, v_p)$ then 7. $CF \leftarrow false$ 8. break 9. end if 10. if CF then 11. Remove v_p from $Pool$: $Pool \leftarrow Pool / v_p$ 12. Put v_p into C_i: $C_i \leftarrow C_i \cup v_p$ 13. Change the state of v_p as placed: $isPlaced \leftarrow true$ 14. break 15. end if 16. end if 17. if $isPlaced \neq true$ then 18. Set a new color class of S_0 C_{k+1}: $C_{k+1} \leftarrow \emptyset$ 19. Put v_p into C_{k+1}: $C_{k+1} \leftarrow C_{k+1} \cup v_p$ 20. Increase k: $k \leftarrow k + 1$ 21. end if 22. end for </pre>

Figure 2. 10 – Clear Pool

At the second step, the second color class of the offspring C_2 is created. Since the pool is not empty, 5 is put into C_2 . The second color class of the first parent C^1_2 and the first color class of the second parent are selected randomly. The nodes in these color classes 4, 8, 13, 1, 0, 14, 7, 3 and 11 are combined with 5 in C_2 . Conflicts between the nodes are calculated, 5 and 11 become the maximum conflicting nodes with 4 conflicts in C_2 . One of them is selected randomly and 5 is moved to the pool. Conflicts with the nodes are recalculated and 11 is the maximum conflicting node to thrown into the pool. 4 and 1 are put into the pool with the same procedure and the second step is completed since C_2 becomes a conflict free color class. After removing the nodes from the parents,

only 9 remains in the parents.

At the third step, 9 is combined with the nodes in the pool 5, 11, 4 and 1 to put into the third color class of the offspring. Only 1 has conflicts with 5 and 11 so 1 is thrown into the pool. The algorithm can not continue with the fourth step because all nodes in the parents are used. Since the pool is still not empty with 1, the algorithm executes to clear the pool (Figure 2.10). 1 has conflicts with 2, 6 in C_1 , 3 in C_2 and 5, 11 in C_3 so a new color is created for 1. At the end of DPBC, we obtain an offspring with 4 colors. The offspring is better than its two parents whereas all of them have same number of color classes. The reason is explained in section 2.7. In Figure 2.12 shows the results of DPBEA, DGA [25] and DSATUR [3] according to the graph in Figure 2.4 (a). All algorithms have the same number of colors 4 but DPBEA gives the best result with respect to the computation of fitness in Section 2.7.

When the time is increased as $t+1$, the dynamic graph G_t in Figure 2.4 (a) changes into the graph G_{t+1} in Figure 2.4 (b). Node 1 reached its life time so it is removed from G_{t+1} . On the other side, node 15 is added to G_{t+1} with respect to the node change rate c_v and its edges are added randomly by using edge probability p . After 2000 iteration steps, the best results of DGA and DPBEA are obtained for the dynamic graph G_{t+1} in Figure 2.4(b) and they are shown in Figure 2.13 with the result of DSATUR.

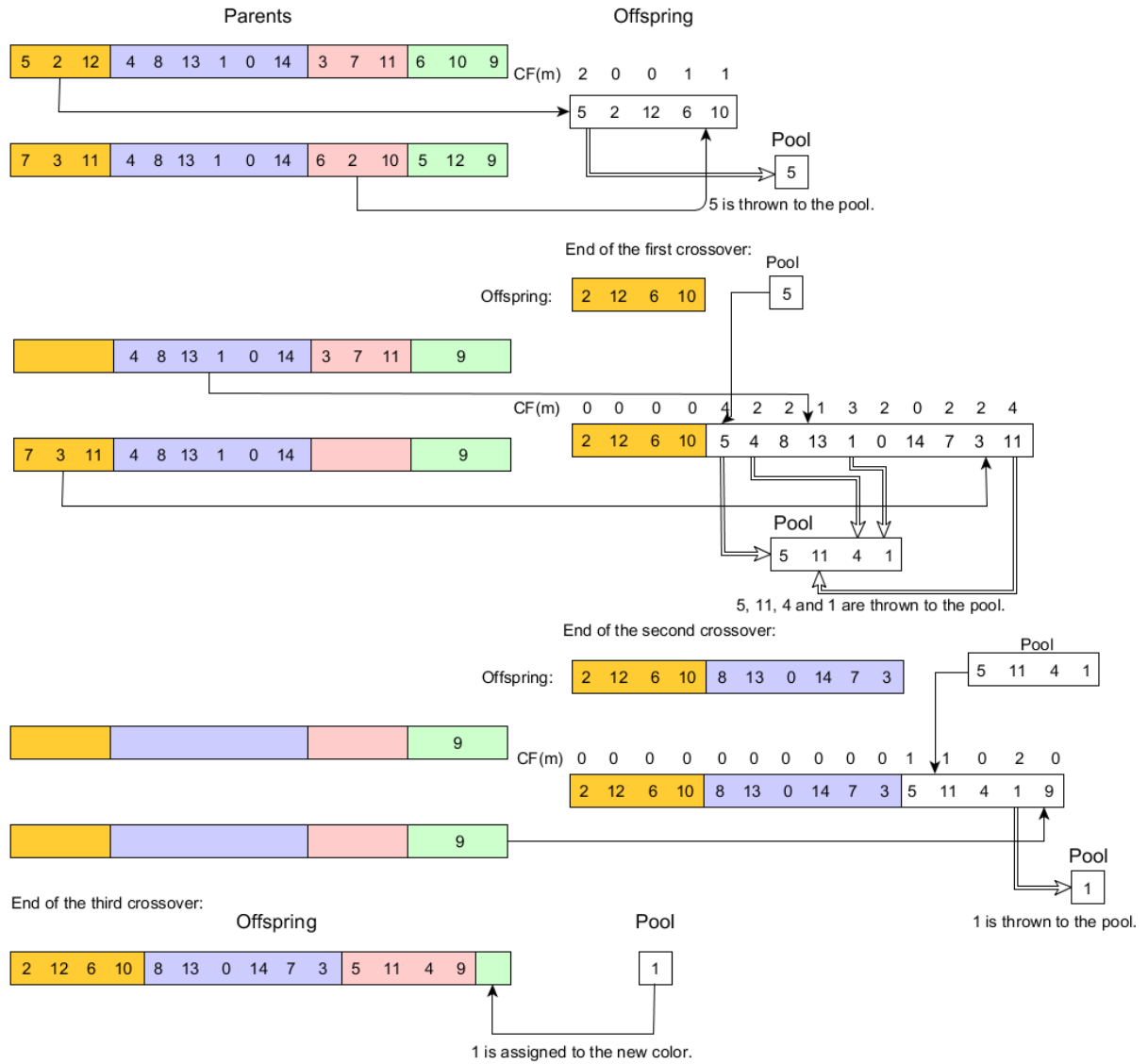


Figure 2. 11 – Example of pool based crossover operation according to node-dynamic graph G_t in Figure 2.4 (a)

DPBEA:

2	12	6	10	8	13	0	14	7	3	5	11	4	9	1
---	----	---	----	---	----	---	----	---	---	---	----	---	---	---

DGA:

11	1	4	12	9	13	10	7	0	5	2	8	14	6	3
----	---	---	----	---	----	----	---	---	---	---	---	----	---	---

DSATUR:

11	6	1	5	8	2	10	3	4	7	12	0	9	13	14
----	---	---	---	---	---	----	---	---	---	----	---	---	----	----

Figure 2. 12 – Results from G_t in Figure 2.4 (b)

DPBEA:

6	2	12	9	10	3	8	7	14	0	13	15	5	11	4
---	---	----	---	----	---	---	---	----	---	----	----	---	----	---

DGA:

0	6	15	11	14	12	13	9	2	5	3	8	4	7	10
---	---	----	----	----	----	----	---	---	---	---	---	---	---	----

DSATUR:

6	5	8	11	10	0	2	7	4	3	12	15	9	14	13
---	---	---	----	----	---	---	---	---	---	----	----	---	----	----

Figure 2. 13 – Results from G_{t+1} in Figure 2.4 (b)

2.6. Local Search

After an offspring is generated with a crossover operation, a local search method is applied to the offspring. While a crossover operation recombines two parents to create a new solution with increasing the population diversity, a local search method improves the quality of the new solution. For this improvement, there are many local search methods that have been purposed in the literature.

2.6.1. Related Works About Local Search Methods for Graph Coloring Problem

Local search methods in graph coloring problem, are based on finding neighborhood solutions of a given offspring by moving nodes between color classes of the offspring. Using this idea, the first local search method for graph coloring problem is proposed in [17]. The paper uses an offspring which has a fixed number of color classes k in a not legal k -coloring and its aim to minimize number of conflicting nodes (i.e. two nodes connected by an edge are in the same color.) in the color classes. Local search method in the paper, selects a node randomly and moves the node to an another color class in the offspring. The new offspring can be better or worse than the current offspring. The algorithm decides if the new offspring is replaced with the current offspring by using Simulated Annealing (SA) metaheuristic [37].

After SA is improved for graph coloring problem, one of the most powerful local search methods for the problem, which is TABUCOL, is purposed in [14]. The algorithm uses tabu search technique [38] to improve a given offspring iteratively. Objective of TABUCOL is same with the objective in [17]. The method is applied to a given offspring which is not a legal k -coloring, for some number of iterations. At each iteration, tabu search generates a neighborhood solution of the offspring by moving a node u from its color class i to an another color class j of the offspring. In order to avoid cycling, the move (u, i, j) becomes "tabu" or "forbidden" which means that u can not be moved back to color class i for the next iterations. Thus, the move is added to a tabu list which is initialized before starting to improve the offspring. At the end of each iteration, the generated solution is compared with the offspring. If the solution is better than the offspring, it becomes the offspring. If it is not, the algorithm continues with the next iteration. Stop critearia of the algorithm can be determined as reaching a predefined

number of iterations or as finding the local optimum of the offspring (See more details in [15]). TABUCOL has been improved by many studies as FOO-PARTIALCOL [18], AMACOL [19], MACOL [20], ATS [21], DNTS [22] and IDTS [23] so far.

2.6.2. Related Works About Local Search Methods for Dynamic Graph Coloring Problem

In dynamic graph coloring problem (DGCP for short), all solutions for a given dynamic graph at each time step, should be legal colorings with nonfixed number k of color classes [31]. Therefore, local search methods in [32], which is the first study applied genetic algorithms to DGCP in the literature, use an offspring that offers a legal coloring with nonfixed number of color classes. Since the offspring has already a conflict-free solution (legal coloring) at the end of the crossover operation, objective of local search methods in [32] is to minimize number of color classes used for the offspring. The paper uses three local search methods as RAR, SWAP and inversion that are described in [34] and adapts the methods to DGCP. SWAP becomes the used local search method in the paper thanks to its outperformances on the experimental dynamic graphs.

Before Swap:	10	13	0	3	4	6	7	8	9	5	2	14	11	12	1
After Swap:	10	13	0	3	9	6	7	8	4	5	2	14	11	12	1

Figure 2. 14 – Example of SWAP Operation

SWAP is applied to an order-based offspring and it changes order positions of any two nodes in the offspring. The Figure 2.14 shows two offsprings as before and after SWAP operation respectively.

The second local search method that is purposed in our previous study [39] for DGCP, also tries to improve an offspring with the same objective in [32]. The method is applied to an offspring which is generated with DPBC operator. Since DPBC obtains a well-improved offspring by recombining two parents at the most of time besides increasing the population diversity, the local search method may effect poorly about minimizing number of the color classes used. However, the method may help to decrease

number of nodes in any color class of the offspring and it may effect the fitness value (Section 2.7) of the offspring. The algorithm of the local search method in Figure 2.15 is built the offspring as follows. The maximum conflicting node v_{\max} in the given dynamic graph G_t , is selected. v_{\max} is removed from its color class and it is placed in the another color class C which is chosen randomly. The conflicts between the nodes in C and v_{\max} is calculated according to G_t . If there are conflicts, the conflicting nodes with v_{\max} are thrown to an empty pool $Pool$. The nodes in $Pool$ are placed in the offspring by using *ClearPool* algorithm (Figure 2.10) which is explain in detail at Section 2.5.3. As it can be seen, the algorithm disarranges the offspring and tries to improve it again using *ClearPool* procedure. If the algorithm considers a dense dynamic graph, it may become a weak operator in order to rearrange the offspring and it may obtain a worse offspring than the offspring at the end of DPBC operator.

<p>Input: Graph G_t, number of color classes of k, an offspring $S_0 = \{C_1 C_2, \dots, C_k\}$ Output: An offspring $S_0 = \{C_1 C_2, \dots, C_k\}$ Initialization: An empty pool $Pool \leftarrow \emptyset$</p> <ol style="list-style-type: none"> 1. Select the maximum conflicting node v_{\max} in G_t 2. Find the color class of v_{\max} C_{\max} in S_0 3. Remove v_{\max} from C_{\max}: $C_{\max} \leftarrow C_{\max} / v_{\max}$ 4. Select a color class C of S_0 randomly, $C \neq C_{\max}$ 5. Add v_{\max} to C: $C \leftarrow C \cup v_{\max}$ 6. for each node v in C do 7. if $edge(v, v_{\max})$ then 8. Remove v from C: $C \leftarrow C / v$ 9. Throw v into $Pool$: $Pool \leftarrow Pool \cup v$ 10. end if 11. end for 12. $k, S_0 \leftarrow ClearPool(G_t, k, S_0, Pool)$ /* Figure 2.10 */

Figure 2. 15 – Local Search Operation in [39]

2.6.3. Local Search Operator

In this study, a new local search method is proposed to solve the explained problems about the local search method in our previous study [38] at Section 2.6.2. The method in Figure 2.16 uses an offspring that is generated with DPBC operator and a dynamic graph at time step t G_t . The algorithm tries to improve the offspring as follows.

Firstly, the color class which has the minimum number of nodes in the offspring C_{\min} , is selected. The nodes in C_{\min} are moved to an empty pool $Pool$ and C_{\min} is removed from the offspring. Finally, each node in $Pool$ is assigned to the offspring by using

ClearPool (Figure 2.10) which is explained at Section 2.5.3. The final step of the algorithm is the same with the final step of the local search in [38]. However, this local search explores new color classes for the nodes in C_{\min} without changing the positions of the other nodes in the remaining color classes. It provides to obtain a better solution than the offspring that is taken as input or the same offspring at least.

Input: Graph G_t , number of color classes of k , an offspring $S_0 = \{C_1 C_2, \dots, C_k\}$
Output: An offspring $S_0 = \{C_1 C_2, \dots, C_k\}$
Initialization: An empty pool $Pool \leftarrow \emptyset$
1. Select the color class having the minimum number of nodes C_{\min} in S_0
2. Throw the set of nodes in C_{\min} V_{\min} into $Pool$: $Pool \leftarrow Pool \cup V_{\min}$
3. Remove C_{\min} from S_0 : $S_0 \leftarrow S_0 / C_{\min}$
4. $k, S_0 \leftarrow \text{ClearPool}(G_t, k, S_0, Pool)$ /* Figure 2.10 */

Figure 2. 16 – Local Search Operation

In order to show effectiveness of the local search method in an example, Figure 2.17 and Figure 2.18 illustrate 2000th iteration of DPBEA for the graph G_{t+1} in Figure 2.6 (b). At this iteration, two parents as S_1 and S_2 are selected from the population randomly and the algorithm starts to combine color classes of the two parents to create color classes of the offspring as S_0 . At the first combination of the parents, the third color class of S_1 C^3_1 and the third color class of S_2 C^3_2 are selected randomly. The nodes in these color classes 8, 3, 10, 14, 7 and 5 are put into the first color class of S_0 C_1 . The conflicts between the nodes are calculated according G_{t+1} in Figure 2.6 (b) and 10, 8 and 3 are thrown to the pool. Since the first color class of S_0 becomes conflict-free, the first combination is completed. At the end of each combination, the nodes are placed in S_0 or thrown to the pool, are removed from the parents temporarily so 8, 3, 10, 14, 7 and 5 are removed from the parents.

At the second combination, the first color class of S_1 C^1_1 and the second color class of S_2 C^2_2 are selected randomly. The nodes in these color classes 6, 11, 9 and 13 are put into the second color class of S_0 C_2 . Since the pool is not empty, the nodes 8, 3 and 10 in the pool are also put into the second color class of S_0 C_2 . Conflict between the nodes are calculated according to the considered graph in Figure 2.6 (b) and 8 is thrown to the pool as the maximum conflicting node of C_2 . Conflicts between the nodes are calculated without 8 and C_2 becomes a conflict-free color class so the second combination is completed. After 6, 11, 9, 13 are removed from the parents, the forth color class of S_1

C^4_1 and the first color class of S_2 C^1_2 are selected randomly. The nodes in C^4_1 and C^1_2 15, 1 and 2 are combined with the node 8 in pool. They are put into the third color class of S_0 C_3 . Conflicts between the nodes are calculated according to the graph and C_3 is already conflict-free so the third color class of S_0 is created. At the forth combination, the remaining color classes from the parents C^2_1 and C^4_2 are selected and the nodes in these color classes 4, 12, 15 are put into the forth color class of the offspring. The group of these nodes is conflict free so the fourth color class of the offspring is created. Since all nodes in the parents are placed in S_0 or the pool, the combinations are finished.

After the crossover operation, the local search method is applied to the offspring S_0 and illustrated in Figure 2.18. Firstly, the color class which have the minimum number of nodes in S_0 C_{min} , is selected. The first, third and forth color classes have the minimum number of nodes so one of them is selected randomly. The third color class of S_0 which has 3 nodes, is selected as C_{min} , 1, 2 and 8 in C_{min} are thrown to the pool and C_{min} is removed from S_0 . For each node in the pool, the remaining color classes are searched if there is a conflict-free color or not. If there is no color class due to conflicts, a new color class is created in S_0 for the node (Figure 2.10). Therefore, 2 is put into the first color class of S_0 and, 1 and 8 are placed in the third color class of S_0 since there is no conflicts between the nodes in the color classes according to the graph.

At the end of the local search operation, the new offspring with 3 color classes is obtained. The best solutions of DPBEA, DGA and DSATUR according to G_{t+1} are shown in Figure 2.19 respectively. DPBEA has better solution than DGA whereas their results are obtained after 2000 iterations. DGA and DSATUR have the same number of color classes but DGA has better result than DSATUR based on their fitness values (Section 2.7).

The edge-dynamic graph G_{t+1} in Figure 2.6 (b) that is used in the example, is the snapshot of an edge-dynamic at 13th time step of *GraphChangeStep* (See Main Scheme of DPBEA in Figure 2.1). Since the population is easily adapted to the changes of edges in the given graph at each time step with DPBEA, a solution with 3 color class is obtained as the best solution for G_{t+1} . When DPBEA creates its initial population with G_{t+1} as a static graph and evaluated the population with more than 2000 iterations, the best solution cannot reach 3 color classes. This situation shows our algorithm obtains better results

with dynamic graphs than static graphs thanks to its adaptation to changes.

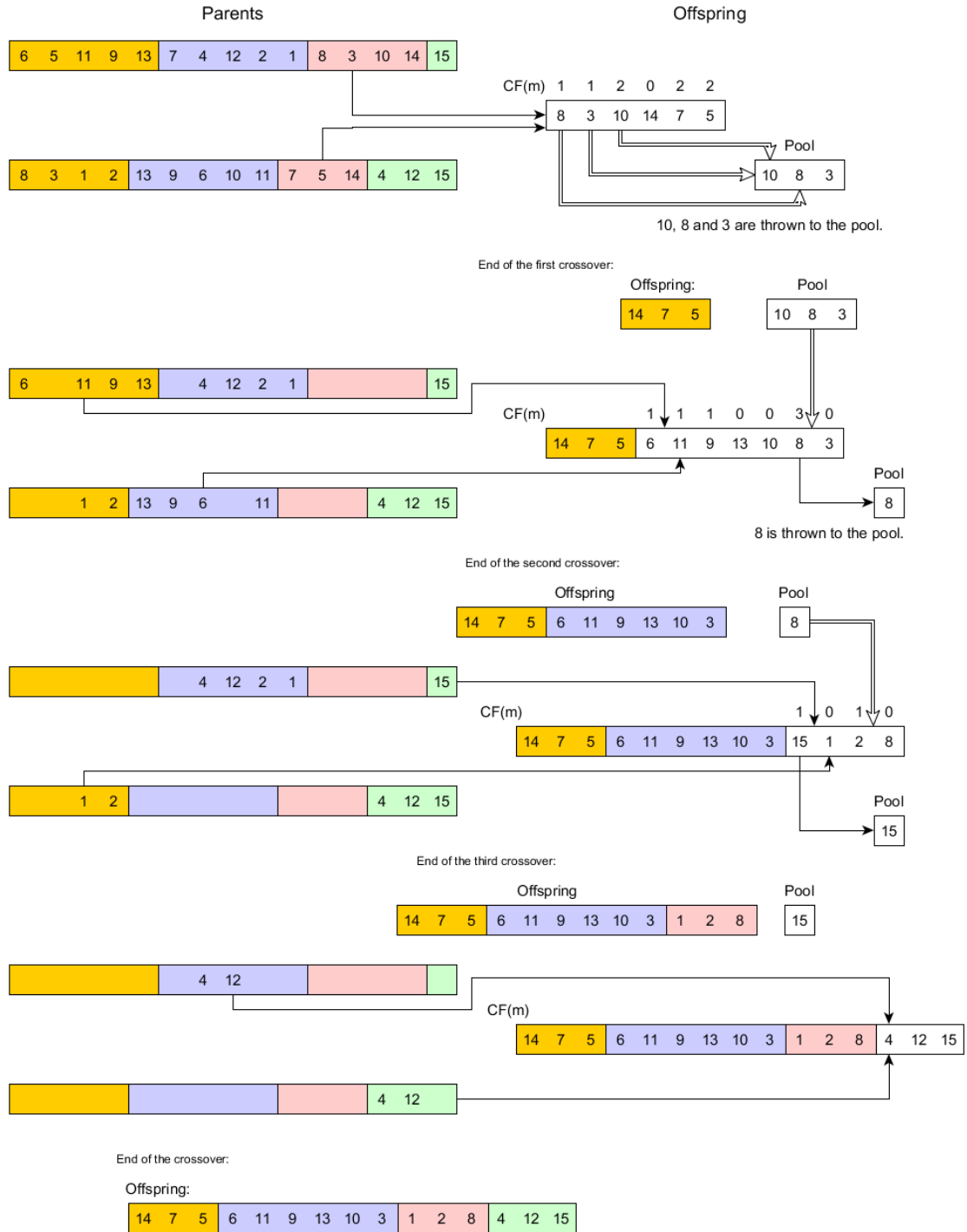


Figure 2. 17 – Example of Pool Based Crossover Operation According to Edge Dynamic Graph G_{t+1} in Figure 2.6 (b)

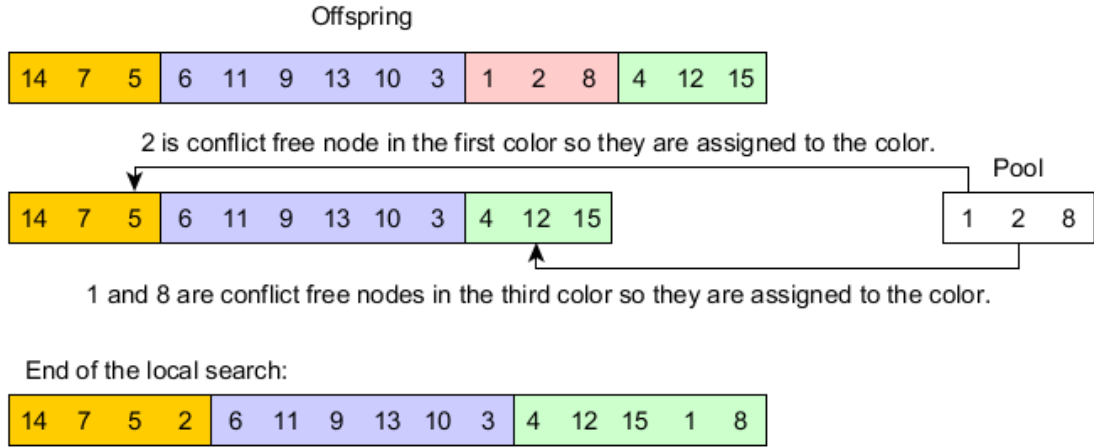


Figure 2. 18 – Example of Local Search Operation According to Edge Dynamic Graph G_{t+1} in Figure 2.6 (b)

DPBEA:



DGA:



DSATUR:



Figure 2. 19– Results from G_{t+1} in Figure 2.6 (b)

2.7. Fitness Calculation

In the most of studies for graph coloring problem in the literature, fitness function f computes a given k -coloring individual S to determine whether it is a conflict-free solution with the lower bound k (fixed or not) number of colors for the given graph or not. If $f(S)$ returns 0, it means that no two vertices are connected by an edge have the same color in the individual S so S is conflict-free and the individual reaches its *legal coloring* with k . If the individual S has an illegal coloring ($f(S) > 0$), $f(S)$ shows how far its coloring

to reach the legal coloring of the graph. If k is not fixed in a given study, k is increased by one and S is generated with $k+1$. This process continues until $f(S)=0$ is reached. If k is fixed in a given study, then the individual which has the minimum fitness value becomes the best solution for the graph. The performances of two conflicting individuals can be compared by using their fitness values. For instance, if two k -coloring individuals as S_1 and S_2 have illegal colorings for the graph and $f(S_1)$ is less than $f(S_2)$, then S_1 is denoted as a better individual than S_2 . This comparison has k -penalty approach [24] and it is used by many proposed works for graph coloring problem such as [5, 6, 20, 40, 41].

When the population is initialized at time step 1 (Figure 2.7), the individuals are created by using an upper bound k colors to color a given graph. At each iteration step, the algorithm tries to color the graph legally with lower number of colors than the previous number of colors used. Since the study aims to minimize k colors used from its upper bound value, a fitness function with k -penalty approach is not needed.

In this study, we used the fitness function that is described in [25]. When the fitness function f compares two individuals, it has two criteria to decide which one is better. The first criteria is the number of colors used in the individuals and it has the highest impact at the fitness function. The second criteria is how many nodes in the three least frequently used color classes of the individual. The individual which has less number of nodes in these color classes, may adopt to changes in the graph at the next time step easily and have more chance to minimize its number of color classes used. Consequently, the fitness function f_i for each individual S_i is calculated by Equation 1 [25].

$$F_i = n^3 \times c_i + n^2 \times c_{i,1} + n \times c_{i,2} \times c_{i,3}$$

Equation 2. 1 – Computation of fitness function

In this equation, n denotes the number of nodes currently available in the dynamic graph, c_i is the number of colors used, $c_{i,1}$, $c_{i,2}$ and $c_{i,3}$ are the number of nodes in the least, second least and the third least frequently used color class. As we are trying to minimize the number of colors used, our algorithm tries to minimize the fitness function. The fitness values calculated for the solutions given in Figure 2.11 are 13789, 14000, 13790 for DPBEA, DGA and DSATUR respectively. Even if all three algorithms use 4 colors, DPBEA and DSATUR have 1 node in the least used color class, so their fitness values

are close to each other whereas DGA has 2 nodes in the least used color and has the worst fitness value.

2.8. Placement of The Offspring In the Population

When an offspring is obtained at the end of an iteration step, the offspring is always replaced with the parent having the worst fitness value. The replacement process is detailed in Figure 2.20.

Input: 1st parent S_1 , 2nd parent S_2 , the offspring S_0
Output: Update population Pop

1. Calculate the fitness values of S_0 , S_1 and S_2 according to the function in Section 2.7
2. **if** $fitness(S_0) > fitness(S_1)$ **then**
3. Replace S_0 with S_1 : $Pop \leftarrow Pop \cup S_0 / S_1$
4. **else**
5. Replace S_0 with S_2 : $Pop \leftarrow Pop \cup S_0 / S_2$
6. **end if**

Figure 2. 20 – Placement of Offspring

2.9. Update of Individuals with Changes of Graph

After an initial graph $G_1 (V, E)$ is created at time step $t=1$, some nodes and/or edges are changed with insertion and deletion operation in the graph G_t in the next time steps t , ($t>1$). As an initial population is generated with $G_1 (V, E)$, this population should also be adapted to these changes at each time step before starting the evolution. In node-dynamic graphs, once a node is added to $G_t (V, E)$ with its edges, all individuals in the population consist the node in one of their color classes with legal coloring. In the deletion case of a node from $G_t (V, E)$, the node, and its color class if it becomes empty, is removed from all individuals in the population. In edge-dynamic graphs, once an edge is added between the nodes u and v in $G_t (V, E)$, the individuals which have u and v in the same color, are detected to move one of the nodes to an empty color. If an edge is deleted from $G_t (V, E)$, an update operation for the individuals is not needed.

11	2	5	8	14	3	9	13	7	12	4	1	6	15	10
----	---	---	---	----	---	---	----	---	----	---	---	---	----	----

Figure 2. 21 – The best solution of DPBEA for G_t in Figure 2.6 (a)

2	5	8	14	9	13	12	4	1	6	15	10	11	3	7
---	---	---	----	---	----	----	---	---	---	----	----	----	---	---

Figure 2. 22 – Adapting DPBEA individual in Figure 2.21 according to the changes between G_t and G_{t+1} in Figure 2.6

In Figure 2.22 shows how an individual is updated according to changes of an edge-dynamic graph. The example uses the best individual of DPBEA in Figure 2.21 for graph G_t in Figure 2.6 (a) and updates the individual according to G_{t+1} in Figure 2.6 (b). When the edges $edge(7, 13)$, $edge(8, 11)$, $edge(7, 11)$ and $edge(3, 5)$ are added to the graph G_{t+1} in Figure 2.2 (b), the newly connected nodes $\{7, 13\}$, $\{8, 11\}$ and $\{3, 5\}$ are detected in a same color of the DPBEA individual in Figure 2.21. Therefore, 3, 7, and 11 are selected randomly from their conflicting pairs to move in a new color of the individual. After this update, the individual in Figure 2.21 becomes the individual in Figure 2.22.

2	12	6	10	8	13	0	14	7	3	5	11	4	9	15
---	----	---	----	---	----	---	----	---	---	---	----	---	---	----

Figure 2. 23 – Adapting DPBEA individual in Figure 2.21 according to the changes between G_t and G_{t+1} in Figure 2.4

In Figure 2.23 shows how an individual is updated according to changes of a node-dynamic graph. The example uses the best individual of DPBEA in Figure 2.12 for graph G_t in Figure 2.4 (a) and updates the individual according to G_{t+1} in Figure 2.4 (b). Once the node 1 is removed from the graph G_{t+1} , the node is also removed from the individual with its color since it becomes empty. At the same time, node 15 is added to the graph G_{t+1} so the node is added to the individual with a new color.

3. EXPERIMENTAL STUDY

In this section, our algorithm DPBEA is tested on two types of dynamic graphs that are generated by using various input parameters. The experimental results of DGA [32] and DSATUR [13] algorithms on the same dynamic graphs are also obtained with DPBEA simultaneously. For each experimental test, any type of dynamic graphs is built with these parameters:

- *Graph Change Step:* After the input graph G_1 is created, graph change step decides that how many time steps the graph is changed. In other words, since a dynamic graph is a set of static graphs, graph change step determines the number of these static graphs in the set. In this study, a generated dynamic graph is changed 50 times so 50 static graphs are generated for the dynamic graph.
- *Initial Number of Nodes:* Initial number of nodes in the graph is denoted by n . When the input graph G_1 of a dynamic graph is initialized, n number of nodes are used. In this experimental study, 5 different initial numbers of nodes are used to initialize input graphs. These are 100, 200, 300, 400 and 500. The default value is 100.
- *Node probability:* Node probability is denoted by c_v that is set between 0.01 and 1. The parameter is used in $n \times c_v$ to determine number of nodes which is added to the dynamic graph at each time step t . In the tests for this study, c_v is set as 0.01, 0.02, ..., 0.1, 0.2, 0.3, 0.4, 0.5. The default value of c_v is 0.1 in this experimental study. c_v is not used for the tests of edge-dynamic graphs.
- *Edge Probability:* Edge probability is denoted by c_e that has the same usage with c_v for the edges that are added to a dynamic graph at each time step t . The parameter is set with the same values of c_v and its default values is also 0.1. c_e is not used for the tests of node-dynamic graphs.
- *Edge Density:* Edge density value is denoted by p which is used to decide the total number of edges should exist on the dynamic graph at each time step t . The parameter is set between 0.1 and 0.9. The default values are 0.7 and 0.5 for node-dynamic graphs and edge-dynamic graphs respectively.
- *Minimum Lifetime:* The minimum amount of iterations that a node or an edge is

kept alive in the graph is the minimum lifetime of a node or an edge and it is denoted by t_{min} .

- *Maximum Lifetime:* The maximum amount of iterations that a node or an edge is kept alive in the graph is the maximum lifetime of a node or an edge and it is denoted by t_{max} .

The default values for the parameters of the evolutionary algorithm are mutation rate=0.3, population size=100 and generation size= 10000. In order to balance the number of nodes or edges that are added and removed at each time step, the values for t_{min} and t_{max} are set to 3 and 13 respectively. These values are used in the experiments unless stated otherwise. Usages of whole parameters in the experimental study are detailed Section 2.1.

The experimental tests in this study, are specialized according to the types of the tested dynamic graphs as node-dynamic and edge-dynamic. Therefore, these tests and values of their parameters are analyzed in two different sections.

3.1. Node-Dynamic Graphs

Experimental studies of node-dynamic graphs are built as follows. The initial graph is created with n nodes and in each time step t , $n \times c_v$ nodes are added. When a node is created, a number between t_{min} and t_{max} is randomly generated to set its lifetime, then all the nodes in the graph are traversed and an edge between the newly added node and the current considered node is created with probability p . In each time step, the graph is changed and is given as input to all three algorithms. In the first iteration step DSATUR algorithm produces its result, and in DGA and DPBEA, each individual in their populations are updated. Therefore, the newly produced nodes are added to and the dead nodes are removed from each individual in the populations of DGA and DPBEA. In DPBEA, the dead nodes are deleted from their color classes and for each newly produced node, a new color class is created and this node is the only node that is placed to this color class. Both DGA and DPBEA algorithms will iterate for 10000 iteration steps (denoted as e), where 10000 individuals are generated. The best individuals from the populations of DGA and DPBEA are recorded at the end of the graph change step. To compare the performance of the algorithms, this process continues for 50 graph change steps to generate 50 graphs (from G_1 to G_{50}) working on one single dynamic graph and a total of

5 dynamic graphs are generated. The results are the mean of the number of colors and fitness values calculated by each algorithm for 250 different graph states.

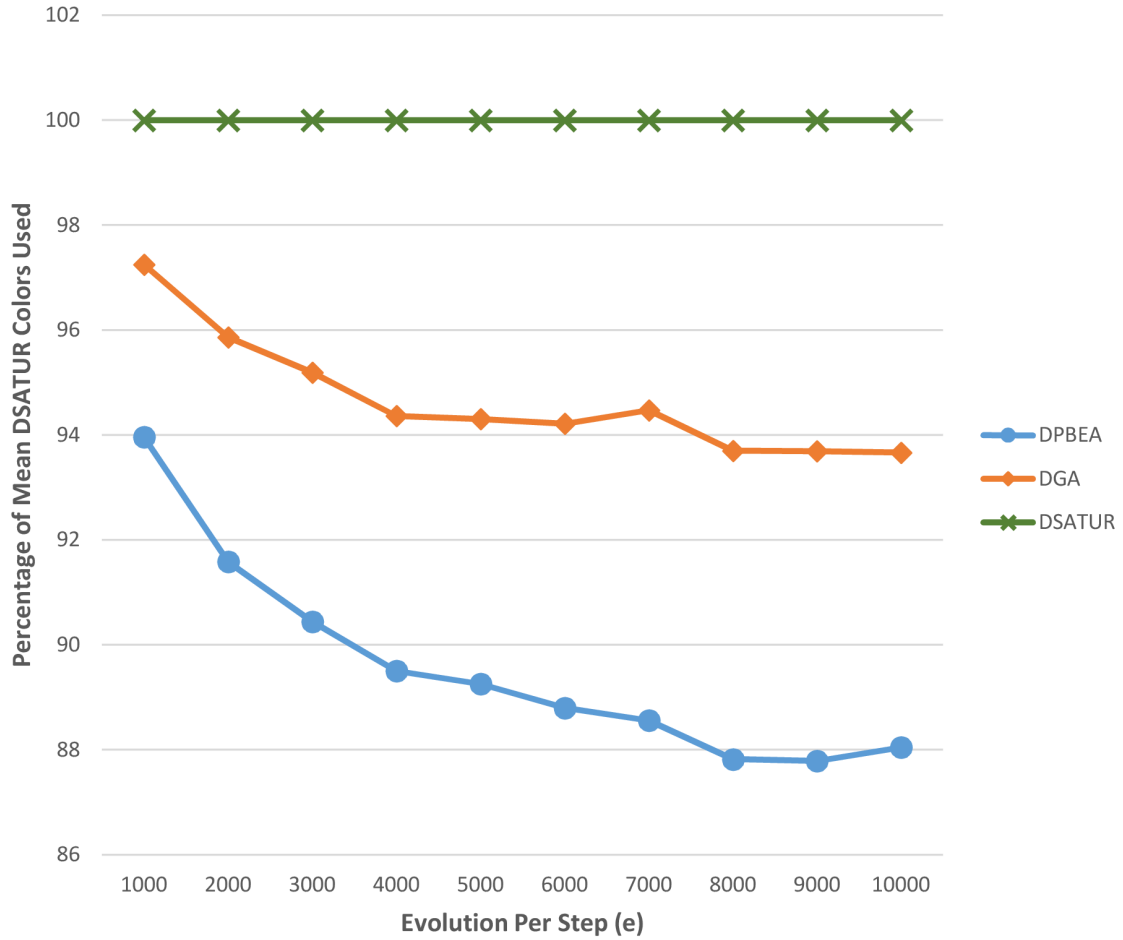


Figure 3. 1 – Varying evolution steps (e) for node-dynamic graphs

The first experimental test is shown in Figure 3.1 is about generation size. The test aims to find the best number of evolution size for DGA and DPBEA. The parameters are set as their default values that are explained at Section 3. Edge probability c_e is not initialized since added edges are determined with p for node-dynamic graphs. The best solutions of DGA and DPBEA at every 1000th evolution step are stated starting from 1000th evolution step to 10000th evolution step. According to these values, the best solution of DGA remains stable after 8000th evolution step and DPBEA reaches its best solution at 8000th evolution step and it remains stable until 9000th step. Therefore, the default value of generation size for node-dynamic graph is set as 8000 for the other experimental tests.

Figure 3.2 shows how the three algorithms DSATUR, DGA and DPBEA react with different node probability value c_v when the other test parameters are set as their default values. When c_v is set as 0.01 that means only one node is added to the tested dynamic graph at each time step, the three algorithms obtain the same best solutions approximately. When c_v is set between 0.03 and 0.05, DGA and DPBEA get their best solutions with close values which are better than the best solution of DSATUR. However, DPBEA outperforms DGA and DSATUR with c_v value which is bigger than 0.05 and DGA obtains better results than DSATUR at the same time. The test can be interpreted in a way that DPBEA has the best adaptation in the algorithms while number of added nodes at each time step t is increasing.

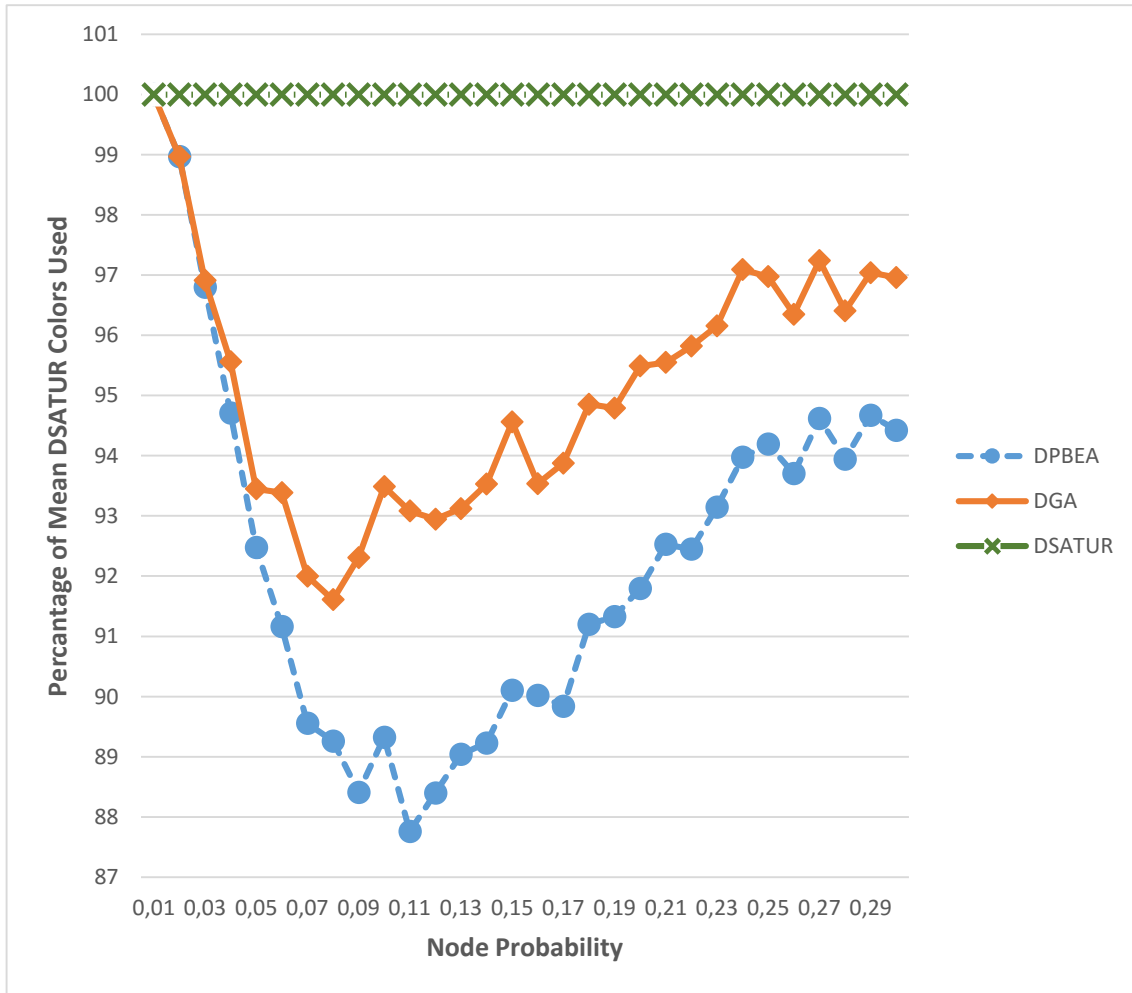


Figure 3. 2 – Varying c_v values

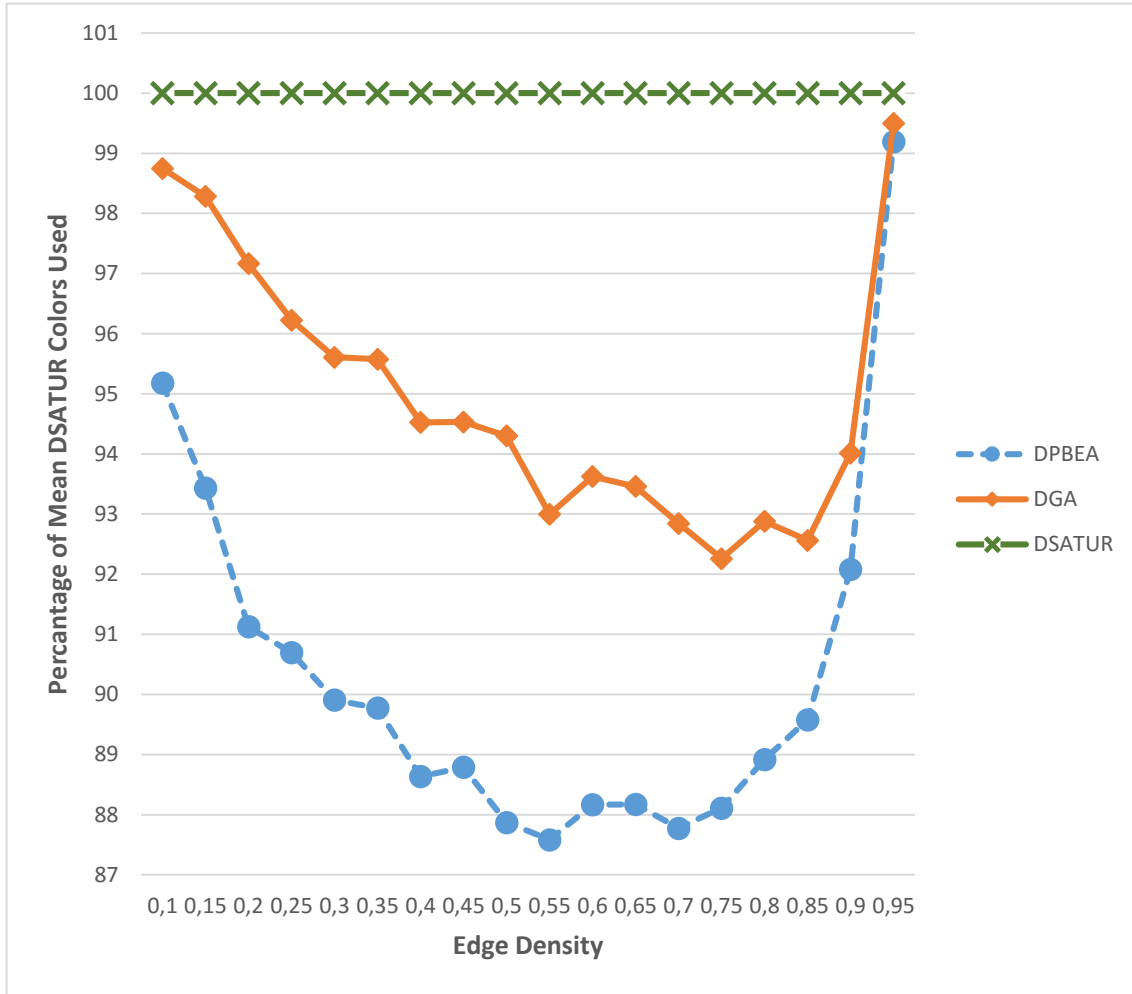


Figure 3. 3 – Varying p values for node-dynamic graphs

Figure 3.3 reports the best solutions of DSATUR , DGA and DPBEA on the dynamic graphs with different total number of edges. In this test, edge density p is changing when the other parameters are set as their default values. For each edge density, the three algorithms are run on 5 generated dynamic graphs. These graphs have a number of edges according to the considered p during their time steps. The best solutions of each algorithm for 5 dynamic graphs for the considered p are averaged and the average value of the p is shown at its axis in Figure 3.3. Starting from 0.1 p value, DPBEA outperforms DSATUR and DGA until 0.95 p value and DGA obtains better results than DSATUR between these values. The gap between the best solutions of DGA and DPBEA grows while p value is increasing from 0.1 to 0.7 and it shows that DPBEA improves itself more successfully than the other algorithms eventhough the dynamic graphs are getting more

intense. When p value is more than 0.9, DGA and DPBEA converge the best solution of DSATUR because the dynamic graphs are getting closer to become fully-connected graphs.

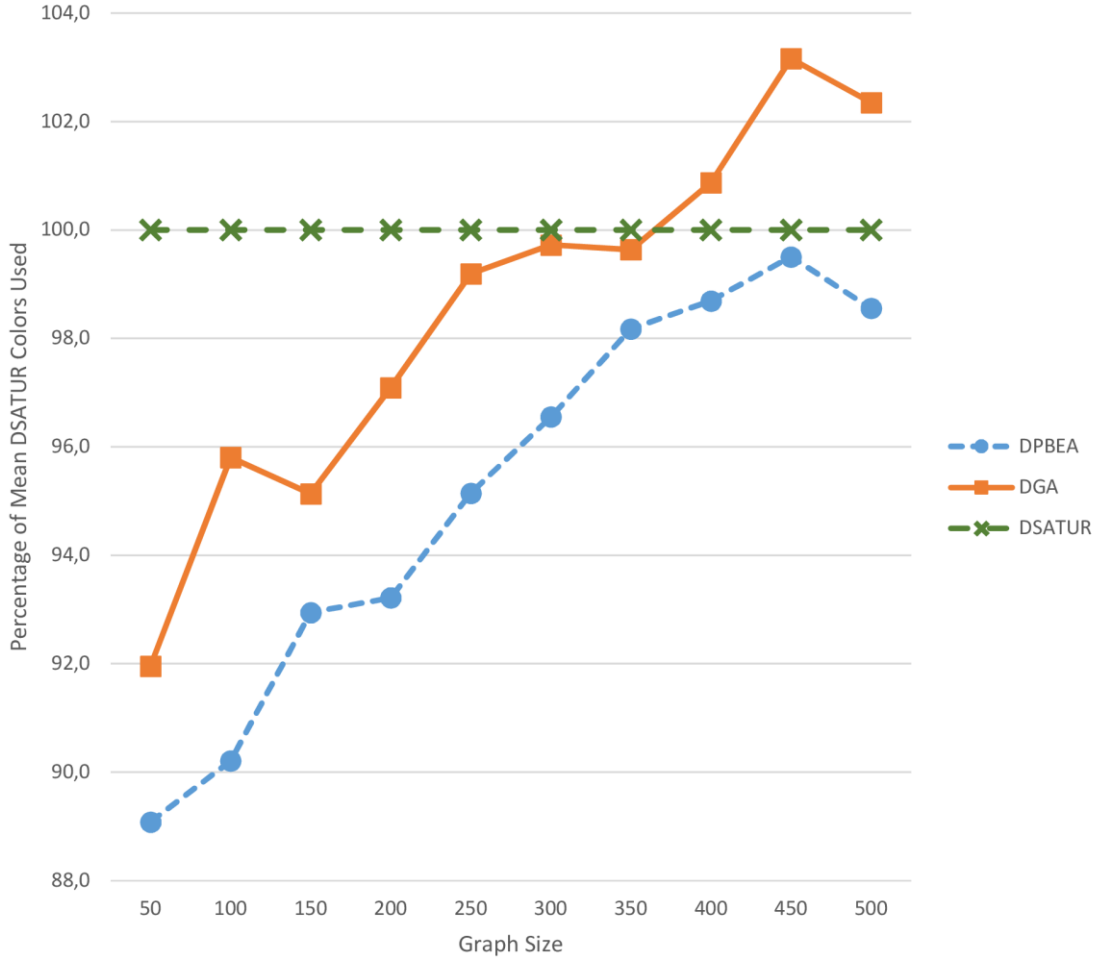


Figure 3. 4 – Varying node values

In Figure 3.4, only the test parameter p is used as its default value. We focus the dynamic graphs that are growing with adding 50 nodes at each time step in this test. The graphs are initialized with 50 nodes and at each time step, 50 nodes are added to the graphs regularly without removing any nodes. Therefore, the nodes have not lifetimes. At 10th time step, number of nodes in the graphs reaches 500 which is the maximum node size in the experimental study. When the total number of nodes in the dynamic graphs are 400, DGA obtains worse results than DSATUR. However, DPBEA outperforms DGA

and DSATUR at each node size in the graphs. The test shows that DGA can not adapt in the dynamic graphs with large number of nodes because of its order-based representation.

NUMBER OF COLORS USED			COMPUTATION TIME			EDGE SIZE	EDGE DENSITY
DSATUR	DGA	DPBEA	DSATUR	DGA	DPBEA		
42	42	41	0,03	4,76	4,73	2942	0,59
41	41	40	0,03	4,89	4,66	2876	0,58
39	39	38	0,03	4,62	4,15	2938	0,59
38	38	35	0,03	4,99	4,19	2873	0,58
34	34	31	0,03	4,67	3,68	2851	0,58
31	31	27	0,02	4,32	3,59	2830	0,57
29	29	24,6	0,02	4,27	3,65	2805	0,57
29	29	24	0,03	4,58	3,68	2755	0,56
26	26	22,4	0,02	4,44	4,01	2666	0,54
23	23	20	0,02	4,27	4,12	2641	0,53
20	20	19	0,02	4,01	4,59	2615	0,53
19,8	19,2	18,4	0,02	4,02	4,69	2675	0,54
19,8	20	18,4	0,02	4,24	4,86	2649	0,54
19	18,2	17	0,02	3,87	5,18	2701	0,55
19,4	18,8	17,6	0,02	4,17	5,6	2663	0,54
19,2	18,4	17,2	0,02	4,2	5,19	2696	0,54
18,2	18,4	17	0,02	4,17	5,79	2770	0,56
19,4	18,4	17	0,02	4,24	5,86	2745	0,55
19	18	17	0,02	4,3	5,91	2710	0,55
18,4	18	16,2	0,02	4,23	5,97	2708	0,55
18,6	18	16,2	0,02	4,14	5,55	2709	0,55
17,4	18	16	0,02	4,27	5,91	2766	0,56
18,2	18	16,4	0,02	4,37	6,07	2729	0,55
19	17,2	16	0,02	4,31	5,84	2706	0,55
18,2	18,4	16,2	0,02	4,22	5,81	2660	0,54
19,8	17,6	16	0,02	3,98	5,74	2665	0,54
18,6	17,8	15,8	0,02	4,11	5,69	2647	0,53
18,4	17,4	16	0,02	4,07	5,63	2634	0,53
18,4	17,8	16	0,02	4,08	5,67	2652	0,54
18,8	18,2	16,2	0,02	4,12	5,7	2622	0,53
18,4	17,8	16	0,02	4,04	5,61	2620	0,53
18,6	18	16	0,02	4,09	5,62	2667	0,54
18,8	18,2	16	0,02	4,24	5,78	2677	0,54

Figure 3. 5 – Results of the algorithms from the node-dynamic graph with $n=100$ at each time step t

Besides increasing number of nodes in a node dynamic graph at each time step t as in Figure 3.4, we tested keeping number of nodes in the graphs constant at each time step t . In Figure 3.5, node-dynamic graphs are initialized with default values of their test parameters but t_{min} and t_{max} are not used. At each time step t , $n \times c_v$ existing nodes are selected randomly and they are removed from the graphs with their edges and $n \times c_v$ new nodes are added to the graphs with their edges at the same time step. Since numbers of added and removed nodes are the same, the dynamic graphs have approximately same number of edges at each time step and their density value vary between 0.53 and 0.59 in these steps. At the first time step, each algorithm gives its worst results for the tested dynamic graphs. While the time step is increasing, the algorithms adapt the changes on the graphs and they improve their solutions. However, DPBEA outperforms DGA and DSATUR in all time steps and its adaptation is more powerful than the other algorithms. When computation times of the algorithms are concerned, DSATUR is very successful to use its time efficiently because it obtains only one solution for each graph at each time step. DGA and DPBEA have approximate values about computation time but DGA has better than DPBEA at each time step.

3.2. Edge-Dynamic Graphs

Edge-dynamic graphs are tested as follows. The initial graph is created with n nodes and these nodes are not removed from the graph and new nodes are not added to the graph at any time step. However, in each time step t , $n \times (n-1) \times p \times c_e \div 2$ edges are added. When an edge is created, two conflict-free nodes in the graph are selected randomly as two endpoints of the edge and a number between t_{min} and t_{max} is randomly generated to set a lifetime of the edge. In each time step, the graph is changed and is given as input to all three algorithms. In the first iteration step, DSATUR algorithm produces its result, and in DPBEA, each individual in their populations are updated since it has partition represented individuals. Therefore, each individual in the population of DPBEA is checked for the newly produced edges whether any conflicts occur because of their endpoints are in the same color. If a conflict occurs in the individual, one of the conflicting nodes is removed from its color class and a new color class is created in the individual to place only this node. Both DGA and DPBEA algorithms will iterate for 10000 iteration steps (denoted as e), where 10000 individuals are generated. The best individuals from DGA and

DPBEA are recorded at the end of the graph change step. To compare the performance of the algorithms, this process continues for 50 graph change steps to generate 50 graphs (from G_1 to G_{50}) working on one single dynamic graph and a total of 5 dynamic graphs are generated. The results are the mean of the number of colors and fitness values calculated by each algorithm for 250 different graph states.

Firstly, a test is built to examine how the best solutions of DGA and DPBEA are changed with respect to number of evolution steps when the other test parameters are set as their default values except for n . The generation size test is executed for 100, 200, 300 and 400 nodes respectively and DPBEA outperforms DGA and DSATUR at the end of 10000 evolution steps in four these tests. In Figure 3.6, DGA obtains better results than DSATUR starting from 2000th evolution step and it improves itself until the end of 10000th evolution step as DPBEA. In Figure 3.7, DPBEA and DSATUR get the same result at 1000th evolution step whereas DGA has the worst result. After 1000th evolution step, DGA has still the worst result between the algorithms until 7000th evolution step but it is getting closer the best solution of DSATUR step by step. Meanwhile, the gap between DPBEA and DSATUR is growing and DPBEA improves itself. After 8000th evolution step, DGA outperforms DSATUR but it has still worse result than DPBEA. In Figure 3.8 and 3.9, DGA has the worst result between the algorithms during the evolution steps and, DPBEA needs more than 5000 steps in Figure 3.8 and 9000 steps in Figure 3.9 to outperform DSATUR. These tests show that DGA is an unsuccessful approach on the dynamic graphs with large instances. However, DPBEA promises to get a good solution with default values of the test parameters eventhough the dynamic graphs become challenging.

In Figure 3.10 and 3.11, edge probability c_e is tested on the three algorithms for 200 and 300 nodes respectively and the other test parameters are set as their default values. In case of using 200 nodes (Figure 3.10), DGA obtains better results than DSATUR at 0.01, 0.03, 0.04, 0.06, 0.1 (also shown in Figure 3.7) and 0.2; the same results with DSATUR at 0.02, 0.05 and 0.07; worse results than DSATUR at the remaining c_e values. However, DPBEA outperforms DGA and DSATUR at all c_e values despite of the fluctuation in its results. In case of using 300 nodes (Figure 3.11), DGA has the worst results at all c_e values meanwhile DPBEA has the best results.

In Figure 3.12, the algorithms are run on the dynamic graphs with 500 nodes. The test shows that while the gap between DGA and DSATUR is growing much more than Figure 3.11, DPBEA is getting closer to DSATUR but it still has the best solutions at 0.01, 0.02, 0.04, 0.05, 0.06, 0.07, 0.1, 0.2 and 0.3. DPBEA has worse result than DSATUR at only 0.5 c_e value.

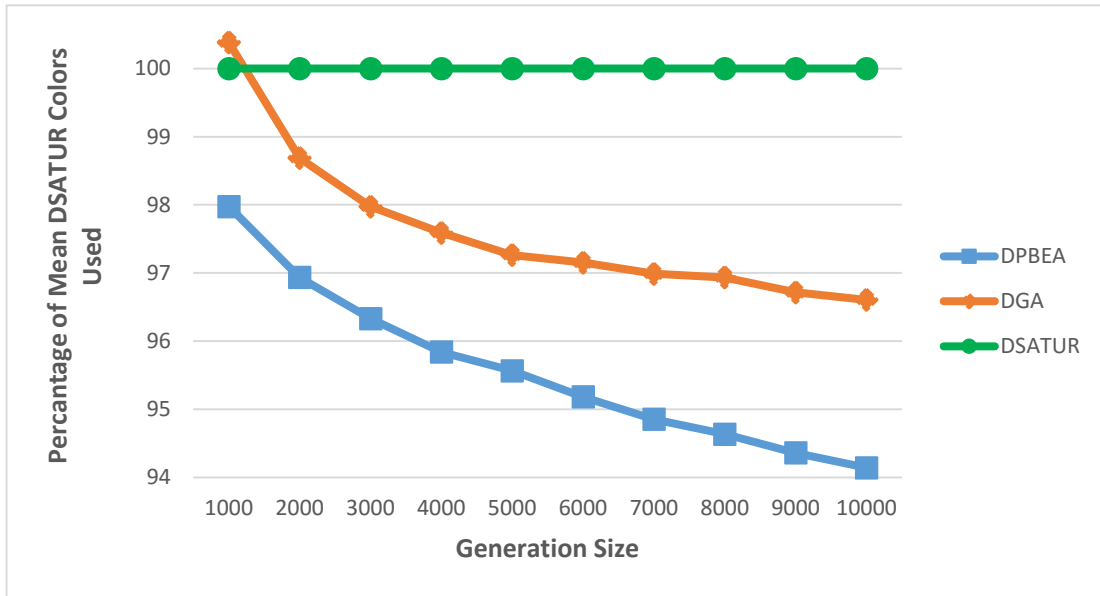


Figure 3. 6 – Varying evolution steps (e) for edge-dynamic graphs when $n=100$

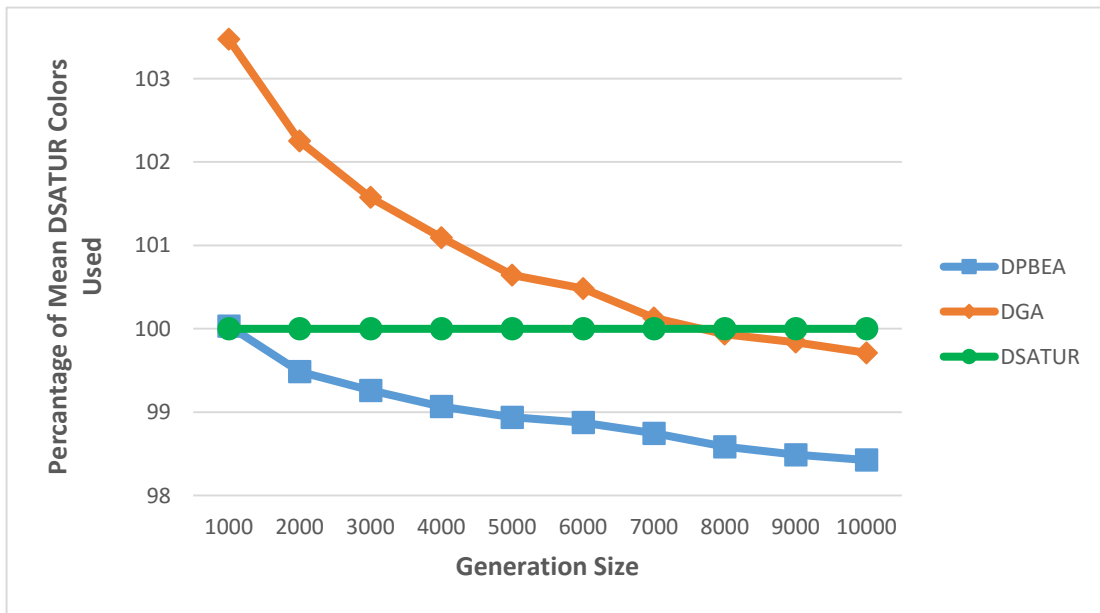


Figure 3. 7 – Varying evolution steps (e) for edge-dynamic graphs when $n=200$

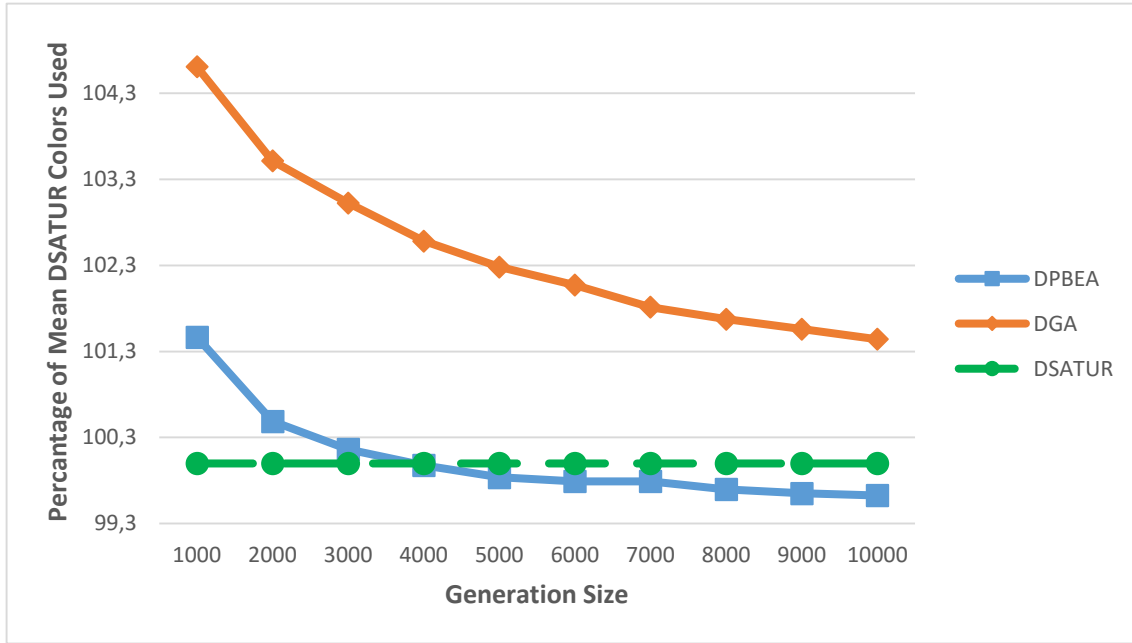


Figure 3. 8 – Varying evolution steps (e) for edge-dynamic graphs when $n=300$

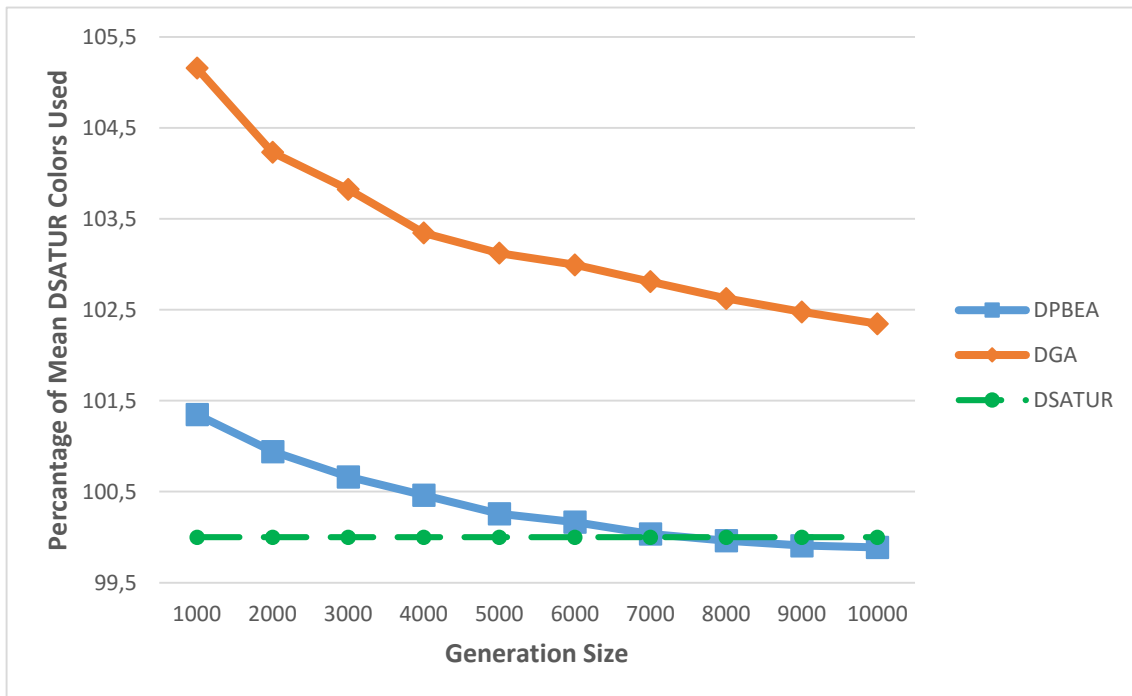


Figure 3. 9 – Varying evolution steps (e) for edge-dynamic graphs when $n=400$

In Figure 3.13, the algorithms are tested on the dynamic graphs with various edge density values p with the other test parameters as their default values. For instance, when p value is 0.5, it means that $100 \times (100 - 1) \times 0.5 \div 2$ edges exist on the dynamic graph at each time step t . In each next time step $t+1$, using default c_e value, $100 \times (100 - 1) \times 0.5 \times 0.1 \div 2$ edges are removed from the dynamic graph and the same number of new edges are added at the same time in order to keep the tested density value for the graphs. We conduct how the three algorithms adapt to these changes with protecting the density. DPBEA outperforms DGA and DSATUR at all p values and DGA has also better results than DSATUR except for 0.1 p value.

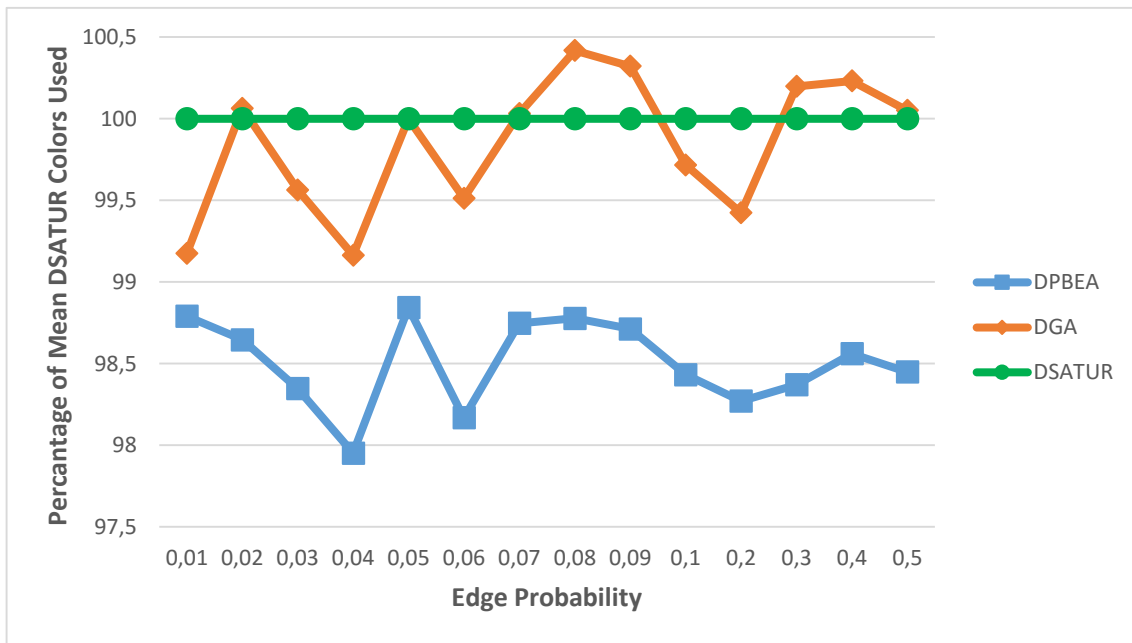


Figure 3. 10 – Varying c_e values when $n=200$

Figure 3.14, 3.15 and 3.16 have the same purpose and procedure with the test in Figure 3.13 for different number of nodes. Common observations in these three tests, are that DSATUR outperforms DGA and DPBEA between 0.1 and 0.4 edge density values. These trends show that DPBEA has worse adaptation than DSATUR when edge-dynamic graphs have large size of nodes and small size of edges. DGA has better solutions than DSATUR only between 0.5 and 0.9 density values in Figure 3.14 and it has worse results than DSATUR at all density values of Figure 3.15 and 3.16.

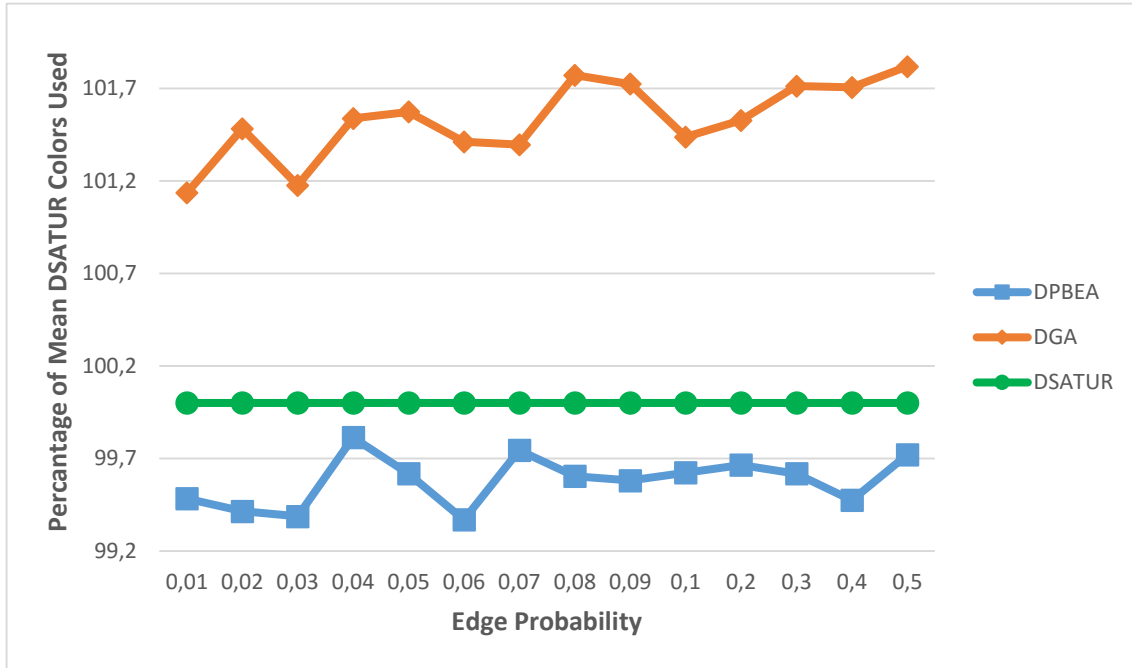


Figure 3. 11 – Varying c_e values when $n=300$

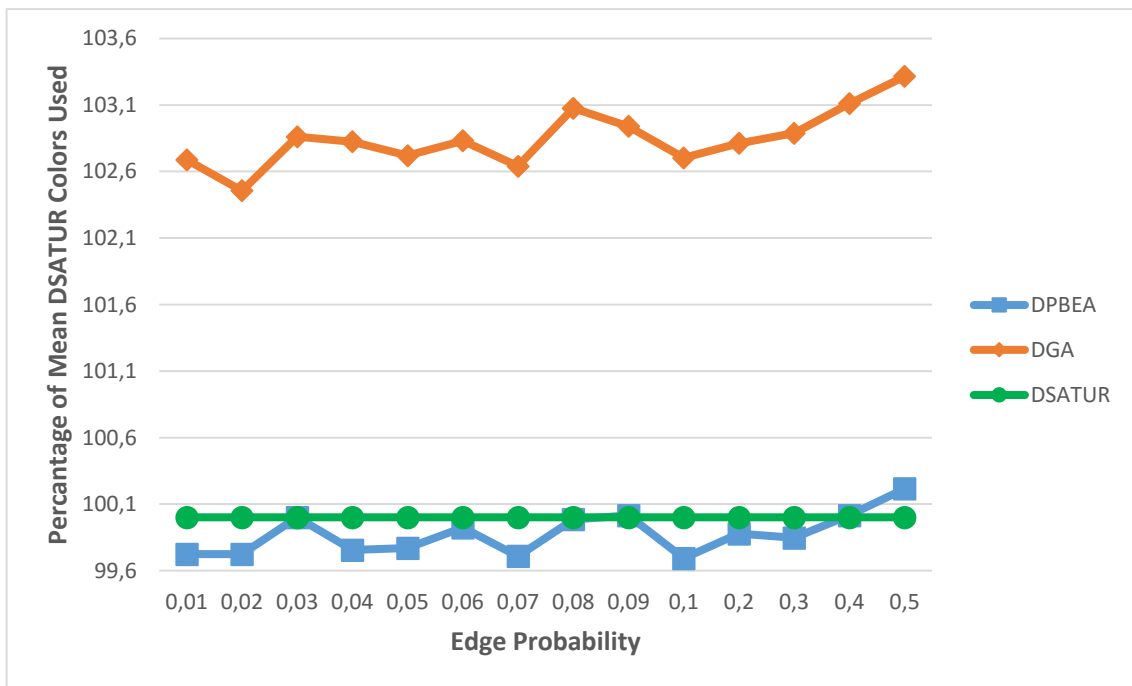


Figure 3. 12 – Varying c_e values when $n=500$

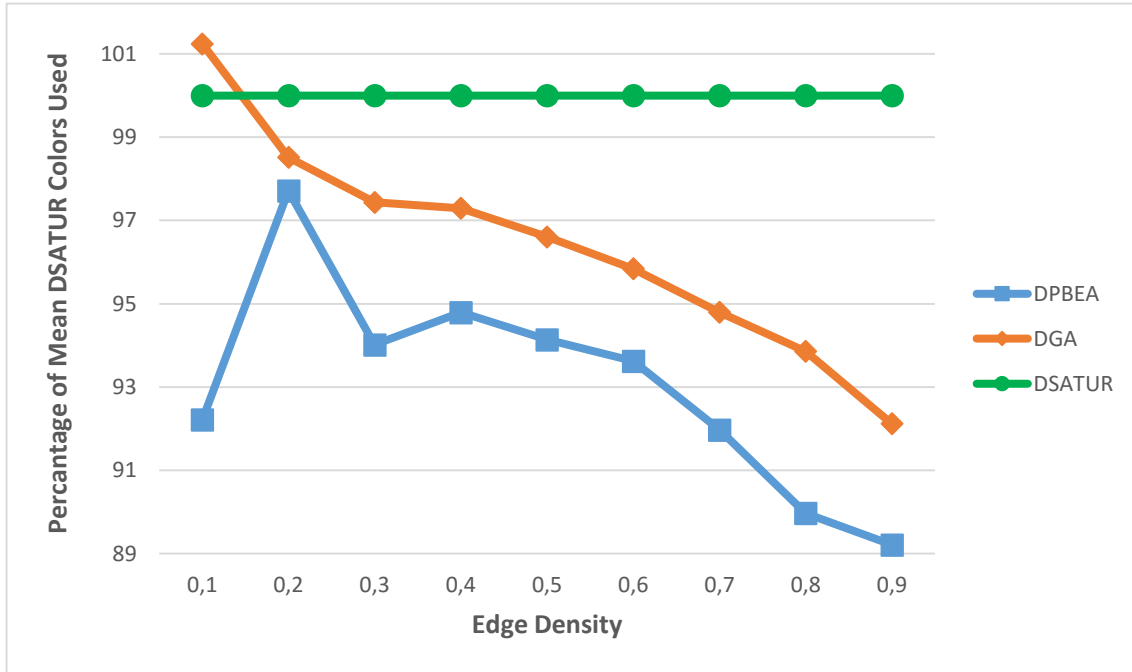


Figure 3. 13 – Varying p values for edge-dynamic graphs when $n=100$

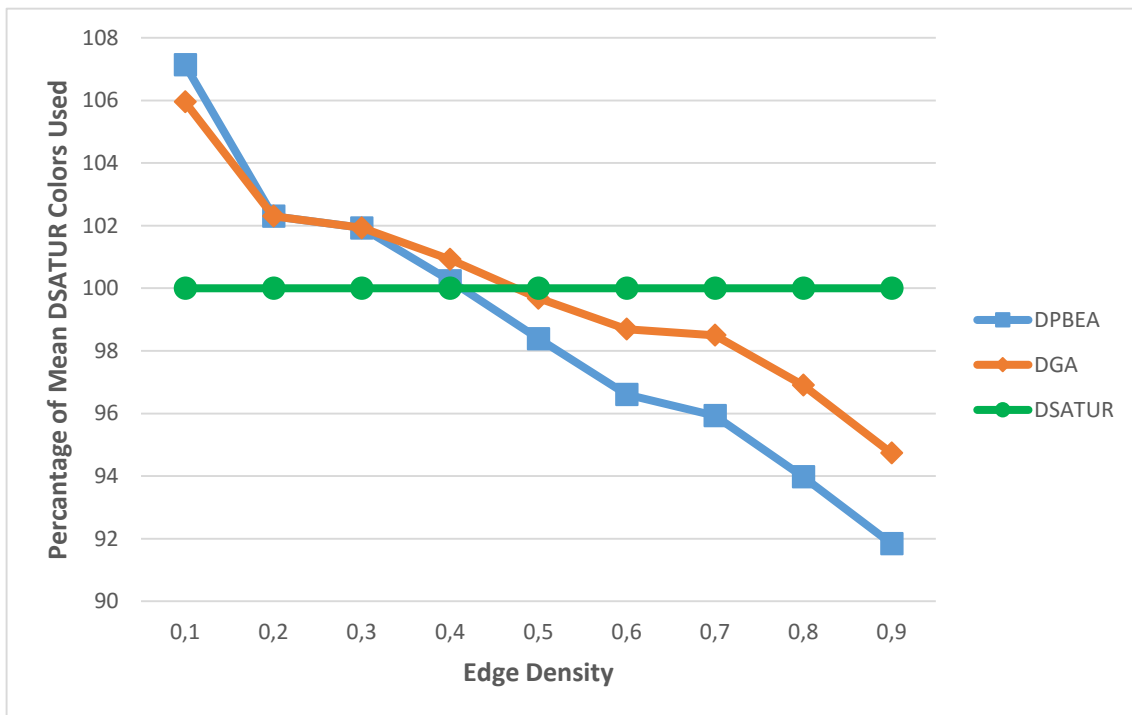


Figure 3. 14 – Varying p values for edge-dynamic graphs when $n=200$

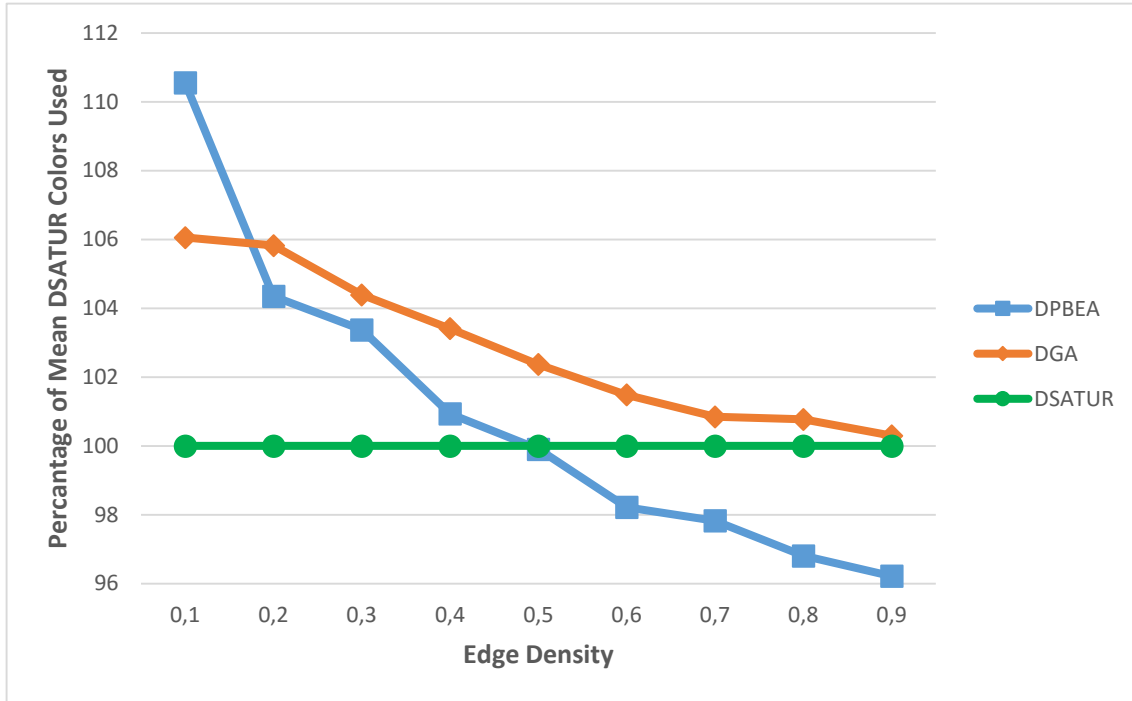


Figure 3. 15 – Varying p values for edge-dynamic graphs when $n=400$

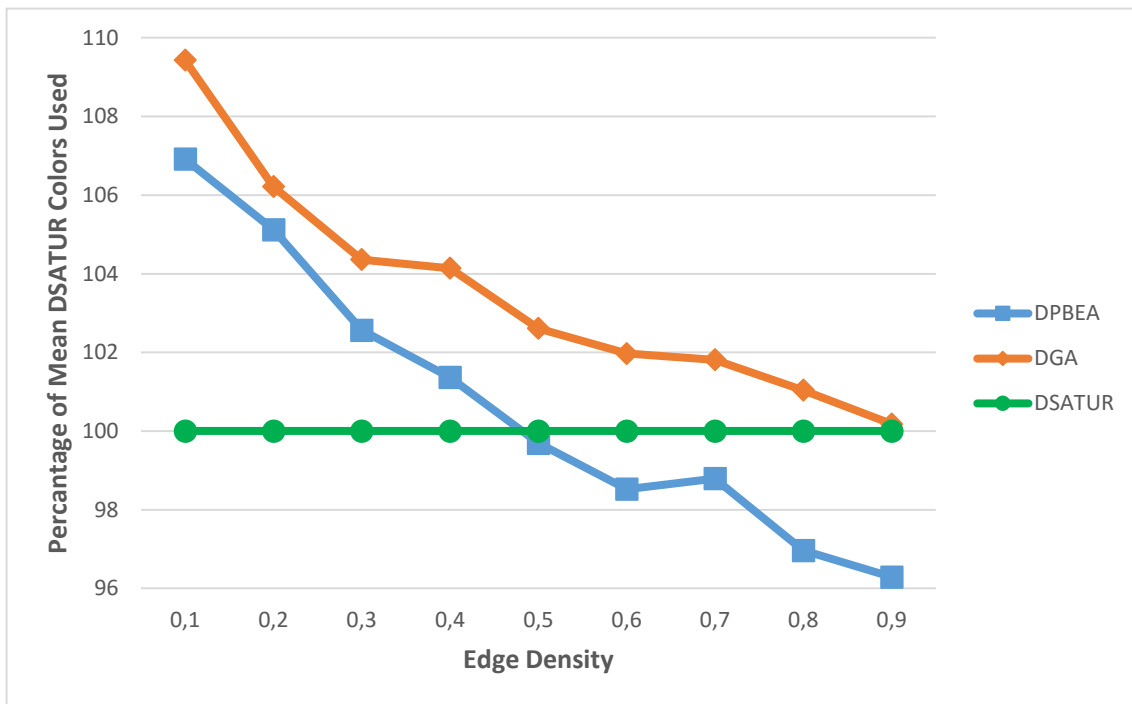


Figure 3. 16 – Varying p values for edge-dynamic graphs when $n=500$

In Figure 3.17, a special case for edge-dynamic graphs is tested without regarding default values of the test parameters. The results of the algorithm are shown in details according to the changes of the edge-dynamic graph step by step. The initial graph is created with 100 nodes and there is no edge between any two of them so edge size is 0 at the first time step. Starting from the second time step until the 13th time step, edges are added randomly without removing any edges. At the 13th time step, the graph becomes fully-connected. After the 13th time step, some edges are removed randomly while new edges are added. When new edges are added without removing between 2nd and 13th time steps, DPBEA outperforms DSATUR and DGA until 5th time step. After that, all the algorithms get the same results until the graph becomes fully connected. It shows that, when the graph is close to fully-connected, all algorithms are successful to find the best solution since there are a few possibilities to color the graph. Starting at 14th time step, a random number of edges are removed from the graph, and DPBEA continues to outperform DGA and DSATUR.

Time Step	Number of Colors Used			Edge Size	State of Graph	
	DSATUR	DGA	DPBEA		Added Edges	Removed Edges
1	1	1	1	0	2719	0
2	19,4	18,8	18	2719	1234	0
3	31,8	30,4	29,4	3953	555	0
4	43,8	40,8	39,2	4508	223	0
5	50,6	49,6	48,8	4731	120	0
6	62,2	62,2	62,2	4851	47	0
7	75	75	75	4898	32	0
8	86	86	86	4930	8	0
9	91	91	91	4938	4	0
10	94	94	94	4942	5	0
11	97	97	97	4947	1	0
12	98	98	98	4948	2	0
13	100	100	100	4950	0	0
14	58,4	58,2	58	4838	267	357
15	50,2	51	49,8	4748	336	411
16	48,4	46,8	46	4673	384	425
17	46,8	45,2	43	4632	412	454
18	47,6	43,4	42	4590	442	427
19	46,4	44,8	43	4605	440	441
20	45	43,4	42	4604	436	447
21	49	45,4	44,6	4593	422	454
22	46,2	43,6	42	4561	485	452

Figure 3. 17 – Results of the algorithms when an edge-dynamic graph is becoming fully connected step by step

4. CONCLUSION

Dynamic graph coloring problem has been explored for two years. DGCP can be generalized and studied with many different domains which can be inspired from real world problems. In this study, we considered basic dynamic graph models and changed their edges or nodes in a given number of times. According to types of changing, we separated our dynamic graph models as node-dynamic and edge-dynamic graphs.

We implemented two heuristic algorithms DSATUR and DGA which are adapted to DGCP in [32]. After we analyzed weaknesses of the heuristic algorithms for DGCP, we purposed an evolutionary algorithm which is called dynamic pool-based evolutionary algorithm. DPBEA became a powerful algorithm for DGCP with its novel crossover operator called DPBC.

We tested DSATUR, DGA and DPBEA in node-dynamic and edge-dynamic graphs with some test parameters. In node-dynamic graphs, DPBEA outperforms DSATUR and DGA but it spends more execution time than DSATUR and DGA. In edge-dynamic graphs, DPBEA has better adaptation and uses the computation time efficiently besides outperforming DGA and DSATUR in most of the test cases. However, DSATUR has better results than DPBEA when the tested graphs have large number of nodes and small number of edges. Whereas DPBEA still needs some improvements for these type of graphs as a future work.

5. FUTURE WORK

The proposed evolutionary algorithm in this study is designed for basic undirected and unweighted dynamic graph models as node-dynamic and edge-dynamic graphs. Since there are only a few studies on dynamic graph coloring problem in the literature and their dynamic graphs are generated randomly with a limited number of parameters, DPBEA is tested on random-generated graphs with the same parameters in order to compare with the other algorithms fairly. Eventhough we improved test parameters of edge-dynamic graphs, the dynamic graph models may change in different ways that have not been searched. Therefore, generation of node-dynamic and edge-dynamic graphs can be analyzed in more detail and their test parameters can be improved in another study. Besides that, real-world optimization problems which are suitable to model the node-dynamic and edge-dynamic graphs can be searched and their graphs can be proposed as benchmarks for DGCP.

Concerning types of dynamic graphs, the studies in the literature are focused on two types of dynamic graph models which are defined in [35] in order to solve dynamic graph coloring problem. However, there are many real-world optimization problems which can be built on node-edge-dynamic graph model and there are no studies in the literature for solving this graph model in an efficient way. Moreover, weighted dynamic graph models in dynamic graph coloring problem are untouched areas and applicable for solving dynamic resource allocation problems.

As a final future work plan, DPBEA can be improved for the other dynamic graph models that are mentioned above, thanks to its powerful adaptation.

6. REFERENCES

- [1] Garey, M. R., & Johnson, D. S. (1979). A Guide to the Theory of NP-Completeness. *WH Freeman, New York*, 70.
- [2] Leighton, F. T. (1979). A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, 84(6), 489-506.
- [3] de Werra, D. (1985). An introduction to timetabling. *European journal of operational research*, 19(2), 151-162.
- [4] Gamst, A. (1986). Some lower bounds for a class of frequency assignment problems. *IEEE transactions on vehicular technology*, 35(1), 8-14.
- [5] Sungu, G., & Boz, B. (2015, July). An evolutionary algorithm for weighted graph coloring problem. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (pp. 1233-1236). ACM.
- [6] 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.
- [7] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., & Markstein, P. W. (1981). Register allocation via coloring. *Computer languages*, 6(1), 47-57.
- [8] Chow, F. C., & Hennessy, J. L. (1990). The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4), 501-536.
- [9] Garey, M., Johnson, D., & So, H. (1976). An application of graph coloring to printed circuit testing. *IEEE Transactions on circuits and systems*, 23(10), 591-599.
- [10] Caramia, M., & Dell’Olmo, P. (2002). Vertex coloring by multistage branch and bound. *Johnson et al.(2002b)*, 40-47.
- [11] Méndez-Díaz, I., & Zabala, P. (2006). A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5), 826-847.
- [12] Mehrotra, A., & Trick, M. A. (1996). A column generation approach for graph coloring. *informatics Journal on Computing*, 8(4), 344-354.

- [13] Brélaz, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM*, 22(4), 251-256.
- [14] Hertz, A., & de Werra, D. (1987). Using tabu search techniques for graph coloring. *Computing*, 39(4), 345-351.
- [15] Davis, L.: Order-based genetic algorithms and the graph coloring problem. In: Davis, L. (ed.) *Handbook of Genetic Algorithms*, pp. 72–90. Van Nostrand Reinhold, N. Y. (1991).
- [16] Galinier, P., & Hao, J. K. (1999). Hybrid evolutionary algorithms for graph coloring. *Journal of combinatorial optimization*, 3(4), 379-397.
- [17] Chams, M., Hertz, A., & De Werra, D. (1987). Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research*, 32(2), 260-266.
- [18] Blöchliger, I., & Zufferey, N. (2008). A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35(3), 960-975.
- [19] Galinier, P., Hertz, A., & Zufferey, N. (2008). An adaptive memory algorithm for the k-coloring problem. *Discrete Applied Mathematics*, 156(2), 267-279.
- [20] Lü, Z., & Hao, J. K. (2010). A memetic algorithm for graph coloring. *European Journal of Operational Research*, 203(1), 241-250.
- [21] Wu, Q., & Hao, J. K. (2013). An adaptive multistart tabu search approach to solve the maximum clique problem. *Journal of Combinatorial Optimization*, 26(1), 86-108.
- [22] Jin, Y., Hao, J. K., & Hamiez, J. P. (2014). A memetic algorithm for the minimum sum coloring problem. *Computers & Operations Research*, 43, 318-327.
- [23] Jin, Y., & Hao, J. K. (2016). Hybrid evolutionary search for the minimum sum coloring problem of graphs. *Information Sciences*, 352, 15-34.
- [24] Galinier, P., Hamiez, J. P., Hao, J. K., & Porumbel, D. (2013). Recent advances in graph vertex coloring. *Handbook of optimization*, 505-528.
- [25] Fleurent, C., & Ferland, J. A. (1996). Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, 63(3), 437-461.
- [26] Starkweather, T., McDaniel, S., Mathias, K. E., Whitley, L. D., & Whitley, C.

- (1991, July). A Comparison of Genetic Sequencing Operators. In *ICGA* (pp. 69-76).
- [27] Myszkowski, P. (2008). Solving scheduling problems by evolutionary algorithms for graph coloring problem. *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, 145-167.
 - [28] Nguyen, T. T., Yang, S., & Branke, J. (2012). Evolutionary dynamic optimization: A survey of the state of the art. *Swarm and Evolutionary Computation*, 6, 1-24.
 - [29] Gamache, M., Hertz, A., & Ouellet, J. O. (2007). A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding. *Computers & operations research*, 34(8), 2384-2395.
 - [30] Cremonezi, B. M., Vieira, A. B., Nacif, J. A. M., & Nogueira, M. (2017, March). A Dynamic Channel Allocation Protocol for Medical Environment under Multiple Base Stations. In *Wireless Communications and Networking Conference (WCNC), 2017 IEEE* (pp. 1-6). IEEE.
 - [31] Barba, L., Cardinal, J., Korman, M., Langerman, S., van Renssen, A., Roeloffzen, M., & Verdonschot, S. (2017, July). Dynamic Graph Coloring. In *Workshop on Algorithms and Data Structures* (pp. 97-108). Springer, Cham.
 - [32] Monical, C., & Stonedahl, F. (2014, July). Static vs. dynamic populations in genetic algorithms for coloring a dynamic graph. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation* (pp. 469-476). ACM.
 - [33] Yuan, L., Qin, L., Lin, X., Chang, L., & Zhang, W. (2017). Effective and Efficient Dynamic Graph Coloring. *Proceedings of the VLDB Endowment*, 11(3).
 - [34] van der Hauw, J. K. (1996). Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. *Master's thesis, Leiden University*.
 - [35] Harary, F., & Gupta, G. (1997). Dynamic graph models. *Mathematical and Computer Modelling*, 25(7), 79-87.
 - [36] Porumbel, D. C., Hao, J. K., & Kuntz, P. (2010). An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring. *Computers & Operations Research*, 37(10), 1822-1832.
 - [37] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598), 671-680.

- [38] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5), 533-549.
- [39] Sungu, G., & Boz, B. (2017, April). Solving Dynamic Graph Coloring Problem Using Dynamic Pool Based Evolutionary Algorithm. In *European Conference on the Applications of Evolutionary Computation* (pp. 189-204). Springer, Cham.
- [40] Chiarandini, M., & Stützle, T. (2002, September). An application of iterated local search to graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations* (pp. 112-125).
- [41] Malaguti, E., Monaci, M., & Toth, P. (2008). A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing*, 20(2), 302-316.

RESUME

GİZEM SÜNGÜ

Marmara University, Goztepe Campus

Engineering Faculty, Computer Science and Engineering Dept.

Room: MB341, PK:34722, Kadikoy, Istanbul, Turkiye

Phone (Cell)+90-536-740-8921

e-mail: gizem.sungu@marun.edu.tr, gizemsungu@gmail.com

EDUCATION

M.S., Computer Science Marmara University, Faculty of Engineering, Istanbul, Turkey, Ongoing. *Thesis Topic: Solving Dynamic Graph Coloring Problem By Using A Heuristic Algorithm, Advisor: Asst. Prof. Dr. Betül DEMİRÖZ BOZ.*

B.S., Computer Science Marmara University, Istanbul, Turkey, 2015

WORK EXPERIENCE

2017-... *Research Assistant*, Gebze Technical University, Institute of Information Technologies, Kocaeli, Turkey

PUBLICATIONS

Gizem Sungu, [Betül Boz](#): An evolutionary algorithm for weighted graph coloring problem. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (pp. 1233-1236). ACM.

Gizem Sungu, Betül Boz. (2017, April). Solving Dynamic Graph Coloring Problem Using Dynamic Pool Based Evolutionary Algorithm. In *European Conference on the Applications of Evolutionary Computation* (pp. 189-204). Springer, Cham.

RESEARCH INTERESTS

Graph Coloring Problem, Evolutionary Algorithms, Dynamic Optimization, Bioinformatics

FOREIGN LANGUAGES

English