Received 4 August 2024, accepted 13 August 2024, date of publication 20 August 2024, date of current version 29 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3446596

RESEARCH ARTICLE

BitEA: BitVertex Evolutionary Algorithm to Enhance Performance for Register Allocation

GIZEM SUNGU TERCI[1], ENES ABDULHALIK[1], ALP ARSLAN BAYRAKCI[1], AND BETUL BOZ[2]
[1]Computer Engineering Department, Gebze Technical University, 41400 Kocaeli, Türkiye
[2]Computer Engineering Department, Marmara University, 34854 İstanbul, Türkiye

Corresponding author: Gizem Sungu Terci (gizemsungu@gtu.edu.tr)

This work was supported by The Scientific and Technological Research Council of Türkiye (TUBITAK) 1002-B Project 124E044.

ABSTRACT Register allocation is important in compiler design to enhance the execution time of programs, as CPU registers operate significantly faster than memory locations. The challenge of optimally assigning program variables to a limited number of CPU registers requires innovative solutions within constrained timeframes, particularly when traditional methods are not applicable due to computational infeasibility on current CPU architectures. This study introduces BitVertex Evolutionary Algorithm (BitEA), a novel approach grounded in bitwise operations and bit-based solutions, designed to accelerate computational performance in register allocation. Our experimental results show that BitEA outperforms the existing methods by a factor of up to 60 across all tested scenarios in the DIMACS benchmarks. Furthermore, in terms of solution quality, BitEA achieves lower chromatic numbers on 9 DIMACS benchmarks compared to its closest contemporaries. This research underscores that BitEA has potential to set a new standard for register allocation through its superior speed and solution quality.

INDEX TERMS Evolutionary optimization, parallel processing, register allocation, graph coloring problem, vertex-weighted graphs, crossover operator, evolutionary algorithms, *k*-coloring, BitVertex representation, bitwise operations.

I. INTRODUCTION

The register allocation problem is a crucial aspect of compiler optimization [1], where the goal is to efficiently assign program variables to a limited number of CPU registers. Since registers are the fastest way for a CPU to access data, optimal allocation is vital for enhancing the program performance. This task presents several challenges, including the limited number of available registers, the need to manage variable lifetimes and overlaps (interference), and dealing with spilling where some variables are stored in slower memory instead of registers. Variable lifetimes refer to the segments of code during which variables are active and need to be stored in registers for rapid access. Interference occurs when the lifetimes of two or more variables overlap, meaning they are simultaneously active and cannot share the same register without causing errors. Due to these challenges, the problem is NP-complete [2]. Strategies like graph coloring [3] and linear scan allocation [4] are employed to navigate these challenges, balancing performance optimization with resource constraints. Despite the complexity, the graph coloring strategy which is NP-hard [5], [6], is often chosen to model the problem over simpler methods like linear scan allocation because of its ability to efficiently manage the patterns of variable interference and its global perspective on register allocation.

Graph coloring constructs an interference graph where vertices represent variables characterized by weights (lifetimes), and edges indicate that two variables cannot share the same register [7]. The goal is to color each vertex (assigning registers) in such a way that no two adjacent vertices (interfering variables) share the same color (register),

The associate editor coordinating the review of this manuscript and approving it for publication was Diego Oliva.

while aiming to utilize a specified number k of available colors (registers). In scenarios where the available k colors are insufficient to color all vertices, some vertices remain uncolored (spilled). This scenario defines the k-coloring problem in graph coloring [8]. The fitness of a given solution is evaluated by $f(k)$, which is calculated as the sum of the weights of uncolored vertices. Achieving an optimal solution requires to minimize this fitness value. This study focuses on solving the register allocation problem by tackling the k-coloring problem in weighted-vertex graphs.

Algorithms in the literature strive to assign all vertices in a given set of colors without spilling any vertices due to conflicts [9]. Although these algorithms may achieve optimal solutions for a given graph model, their computational time makes them impractical for applications in CPU architectures to enhance the performance of real-time systems, due to the high complexity of the algorithms. One of the most effective algorithms for solving the register allocation problem, known as the Integrated Crossover-based Evolutionary Algorithm (InCEA) [10], iteratively improves a population of candidate solutions over a predefined number of iterations. However, as the size of the problem instance increases, InCEA requires significantly more computational time to complete these iterations, rendering it impractical for register utilization in CPU architectures. In this study, we aim to address this issue by redesigning the evolutionary algorithms using bitwise representation and operations. Our design is named *BitVertex Evolutionary Algorithm (BitEA)* and presents the following contributions to the literature:

- We introduce *BitVertex*, a novel bit-based structure for graph coloring representation. In this approach, each bit represents a vertex, allowing a single integer to encode the coloring of graphs with up to 32 vertices. This compact representation significantly reduces the storage requirements for algorithms and problem instances, enhancing computational efficiency compared to traditional representations that use a separate integer for each vertex.
- We have developed a scalable architecture to extend the BitVertex representation to graphs with more than 32 vertices. This enables the application of our efficient bit-based encoding method to a wider range of problem sizes with minimal complexity increase.
- By adapting the BitVertex representation, we propose *the BitVertex Evolutionary Algorithm (BitEA)*, a novel evolutionary algorithm that employs bitwise logic (OR, AND, XOR, NOT) for its essential algorithmic processes such as recombination and local search strategies. This integration allows for the parallelization of the coloring process, significantly reducing the time complexity of the algorithm. Our experiments show that this approach speeds up computational performance by an average of 25 times across 223 graph benchmarks.
- Furthermore, to enhance solution quality, BitEA incorporates a novel local search strategy within its crossover operations. This approach improves the fitness values of benchmark instances, outperforming the performance of existing algorithms in the literature.

To detail these contributions, the remainder of this paper is structured as follows. Following the introduction, we provide the related work on solving the register allocation problem, a review of existing heuristic algorithms and their limitations. Section III offers a comprehensive explanation of the preliminaries essential for understanding the basis of our study, setting the groundwork for the subsequent discussions. Section IV details the components of our proposed algorithm highlighting its innovative use of bitwise operations for efficiency improvements. We then present an extensive experimental study in Section V, where the performance of InCEA and our algorithm are compared across various benchmark instances. Finally, Section VI concludes the paper with a discussion of the key findings, the implications of our research, and potential directions for future work in the field of register allocation optimization.

II. RELATED WORK

Numerous algorithms have been proposed in the literature to address the graph coloring model for the register allocation problem. Chaitin's algorithm [7] is the first algorithm that introduces register allocation problem on graph coloring and the following studies Briggs's [11] and George and Appel's [12] algorithms improve its design by changing the spilling heuristic and the coalescing strategy [13], respectively. These studies have made significant contributions in optimizing compiler efficiency. However, the deterministic nature of the algorithms, when applied to the diverse and complex nature of real-world programs, sometimes restricts the exploration of the full solution space for spilling. This limitation can lead to suboptimal performance, with potentially missing more efficient allocations on modern architecture with specific hardware features, especially for complex or unusual code structures.

Considering these challenges, approaches such as exact algorithms [16], [17], [18], [19], [20], evolutionary algorithms [10], [21], [22], [23], [24], [25], learning methods [26], [27], [28] and other heuristic approaches [29] have emerged as promising alternatives due to their adaptability and capacity to explore broader solution spaces by using graph coloring. These methods offer an adaptive approach to register allocation by adjusting strategies based on specific program characteristics and navigating through a range of potential solutions to find optimal or near-optimal allocations.

Exact methods, such as the branch-and-price algorithm [18] and Integer Linear Programming (ILP) models presented in [19], [20], have shown effectiveness in solving the weighted vertex coloring problem (WVCP). Specifically, the ILP model proposed in [19], which casts the WVCP as a maximum weight independent set problem [30], is currently the best-performing exact method. These exact methods are typically limited to efficiently solving problem instances with up to 250 vertices. Despite their ability to ensure optimality, optimal solutions for instances with fewer than 250 vertices were obtained using the Mixed-Integer Programming (MIP) formulation in [19] within 10 hours of computation time using CPLEX solver [31]. The required long computational time makes the benefits of these exact methods controversial for smaller instances. Therefore, heuristic and metaheuristic approaches are necessary not only for larger and more complex instances but also for smaller instances, as they provide a broader exploration of potential solutions within a reasonable amount of computational time.

Evolutionary algorithms are widely studied among the heuristic approaches to solve register allocation problem in the literature such as Hybrid Evolutionary Algorithm (HEA) [24], Cost-Oriented Memetic Algorithm (COMA) [25], Pool-based Evolutionary Algorithm (PBEA) [23], and Integrated Crossover Based Evolutionary Algorithm (InCEA) [10]. The evolutionary algorithms HEA, COMA, PBEA, and InCEA use partition-based representation [32] for their solution individuals. The algorithms create an initial population with a predefined number of individuals using problem-specific metrics which are detailed in Section III-B. The fitness of each individual in the population is evaluated based on criteria that consider both the coloring constraints and the weights of the vertices, aiming to minimize conflicts and possibly prioritize heavier-weighted vertices which is defined in Section III-A. By selecting two individuals as parents from the population, they apply their crossover operators and local search techniques to produce an offspring as a new solution. The algorithms iterate this process to explore and exploit the solution space [33] to improve the fitness value in a predefined number of times and they achieve to obtain optimal or near-optimal solutions for most instances of the problem. Among these evolutionary algorithms, InCEA [10] stands out by computing the best solutions in terms of fitness values and computational performance on the same graph instances. Its algorithmic contributions have also inspired the algorithms aimed at addressing the bin packing problem [34] and the weighted vertex coloring problem [35] in the literature. Given its demonstrated efficacy, we consider the InCEA algorithm as a leading evolutionary approach in the literature for this study.

While the literature has succeeded in obtaining high-quality solutions for given graph instances of the problem, a recent challenge has emerged that these methods are not practical for real-time architectures due to their high computational times. In the last few decades, various studies have proposed methods to accelerate the traditional graph coloring problem. These studies have proposed parallelizing genetic algorithms to generate two or more offspring simultaneously using crossover and mutation operators assigned to threads [36]. Additionally, GPU performance has been utilized to speed up greedy algorithms for the graph coloring problem [37], [38], [39], [40], [41], [42]. While GPU-based and parallel methods decrease the computational time of algorithms, the speedup achieved is primarily attributed to the performance characteristics of the GPU architectures. This limitation exists because these approaches cannot reduce the time complexity of the fundamental operations in the algorithms due to lack of parallelizing algorithmic structures and constraints of GPU architectures and instruction sets. Moreover, GPUs can consume a significant amount of power, especially designed with its high-performance models, to solve the problem instances [43].

The recent study about the acceleration of graph coloring problem is called BitColor [44] which enhances performance efficiency of the first greedy algorithm to solve graph coloring problem in the literature called *Degree of Saturation (DSATUR)* [46] utilizing FPGA [45] where energy efficiency is considered. BitColor refines DSATUR algorithm by integrating bitwise operations to adapt it on FPGA architecture. The algorithm assigns colors to the vertices of a graph to ensure no two adjacent vertices have the same color while minimizing the number of colors used. The process begins by initializing a flag array of colors for each vertex and proceeds through three main steps: first, it traverses the neighbors of a vertex to identify which colors have been used, marking these in the flag array. Then, it selects the first available color from this array, defining an available color as one that has not been used by any neighbors of the vertex. Finally, the selected color is assigned to the vertex, and the flag array for each neighbor is updated to indicate this color is no longer available for them, thus preparing for next coloring decisions. This process is repeated for all vertices to achieve an optimal color distribution without conflicts. The study introduces *BitColor* representation to optimize the traversal of color states for each vertex, by encoding the color states as bits. Color assignments and restrictions for vertices are managed through bitwise AND and OR operations. Experimental results show that DSATUR implemented with BitColor on FPGA architecture outperforms traditional flag array representations on CPU architecture, highlighting the efficiency of FPGA utilization.

BitColor employs a vertex-oriented and deterministic strategy for coloring vertices, visiting them one by one in a sequential manner. In contrast, evolutionary algorithms generate an initial population of colorings and iteratively improve these through a blend of recombination and local search strategies to diversify the population and obtain higher quality solutions. Unlike BitColor's vertex-oriented methodology, evolutionary algorithms operate in a color-oriented manner, manipulating colors along with their associated vertices. Therefore, the vertex-specific design of BitColor does not align well with the requirements of evolutionary algorithms proposed for graph coloring problems, including register allocation.

To bridge this gap, we introduce a novel bitwise representation named *BitVertex*, which encodes vertices as bits instead of colors. To develop genetic operators designed for the register allocation problem using bitwise operations in the BitVertex framework, we propose the *BitVertex Evolutionary Algorithm (BitEA)* in this study. BitEA is the first evolutionary algorithm designed in a bitwise format to accelerate the register allocation problem by reducing time complexity. This methodology can serve as a guide to efficiently accelerate other algorithms for related challenges.

III. PRELIMINARIES

A. PROBLEM DEFINITION

Consider $G(V, E, w)$, an undirected vertex-weighted graph where V and E denote the sets of vertices and edges, respectively and w represents the weight values of the vertices. With n vertices in V, the cardinality of w, denoted as $|w|$, is also n. The graph can have a maximum of $\frac{n\times(n-1)}{2}$ edges, and the number of edges, $|E|$, satisfies $0 \leq |E| \leq \frac{n\times(n-1)}{2}$.

Each vertex v in V is associated with a color class C_i, where C_i is one of the disjoint independent sets belonging to the set $C = C_1, C_2, \ldots, C_k$, with $1 \leq k \leq n$.

The coloring of the vertices obeys the following rule: if u and v are adjacent vertices, i.e., there exists an edge (u, v) in E, then u and v must belong to different color classes (1). This constraint is referred to as a k-feasible coloring.

$$\forall v, u \in C_i, (u, v) \notin E, i = 1, 2, \ldots, k \tag{1}$$

The fitness function $f(k)$ is defined for a given color class value k and aims to assign a color class to each vertex while minimizing the sum of weights of conflicting vertices (2).

$$\text{minimize } f(k) = \sum_{v\in V\setminus\bigcup_{i=1}^{k} C_i} w(v) \tag{2}$$

In other words, the objective is to achieve a coloring that minimizes the total weight of conflicting vertices, where a conflicting vertex is one that cannot be assigned to any color class.

Algorithm 1 General Algorithm [10]

```
Input: Graph $G$, population size $p$, number of color
       classes $k$, number of iterations $ItNum$
Output: Individual $S^*$ with the best fitness value
$Pop = \{S_1, \ldots, S_p\} \leftarrow$ InitialPopulation()
for $i \leftarrow 1$ to $ItNum$ do
    Randomly select individuals $S_x$ and $S_y$ from $Pop$
    where $x \neq y$
    $S_0$, Pool $\leftarrow$ CrossoverOperation($G, k, S_x, S_y$)
    if $Pool \neq \emptyset$ then
        $S_0 \leftarrow$ LocalSearch($k, S_0, Pool$)
    end
    $f(k) \leftarrow$ Evaluate fitness of $S_0$
end
Pop $\leftarrow$ UpdatePopulation($S_0, S_x, S_y$)
$S^* \leftarrow$ BestIndividual($Pop$)
```

B. INTEGRATED CROSSOVER BASED EVOLUTIONARY ALGORITHM

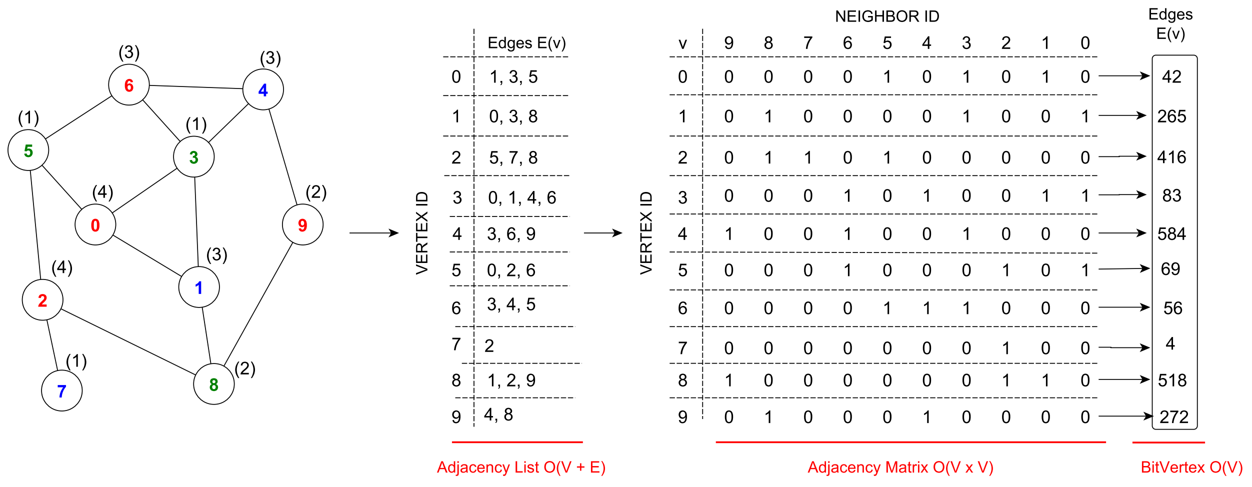
Integrated Crossover Based Evolutionary Algorithm [10] is designed for solving the vertex-weighted k-coloring problem through a hybrid evolutionary approach. The general procedure of the algorithm is built in Algorithm 1. The inputs include a graph G, population size p, the number of color classes k, and the desired number of iterations *ItNum*. The primary objective is to discover an individual S^* with the optimal fitness value. The algorithm initializes a population (*Pop*) of size p using the *InitialPopulation()* function. It then enters an evolutionary loop, iterating over *ItNum* cycles. At each iteration, it randomly selects two distinct individuals (S_x and S_y) from the current population. The algorithm applies Integrated Crossover (InCX) operation (*CrossoverOperation*) to the selected parents, resulting in a new individual (S_0) and a pool of conflicting vertices (*Pool*). If the pool is not empty, the local search technique W-SWAP (*LocalSearch*) is applied to S_0 using the pool to minimize conflicts and improve the solution. The population (*Pop*) is updated by replacing S_x or S_y with S_0 if S_0 represents a better solution than at least one of its parents. The algorithm outputs the individual S^* with the best fitness value achieved throughout the evolutionary process.

The InCEA evolutionary algorithm encodes vertices as integers, with their corresponding color classes represented as lists of integers. This study aims to improve both the design and the effectiveness of the InCEA algorithm. Originally built with a structure only based on integers, A significant transformation is applied to InCEA algorithm to obtain the bitwise framework, which includes converting list-based computations used in various algorithmic operations into more efficient bitwise calculations. The resulting algorithm is now referred to as *BitVertex Evolutionary Algorithm.*

IV. BitVertex EVOLUTIONARY ALGORITHM

A. GRAPH REPRESENTATION

In the proposed *BitVertex* representation, adjacency information of a graph is compacted into a single integer per vertex, using binary encoding to denote edge presence or absence. This binary representation takes advantage of the fact that modern computers natively store and operate on integers in binary format, allowing for a more space-efficient storage of graphs in memory. The space complexity of this representation is $O(|V|)$, a significant improvement over the $O(|V| + |E|)$ required by an adjacency list and the $O(|V|^2)$ required by an adjacency matrix as illustrated in Fig. 1. For each vertex v_i, the binary digits of its corresponding integer directly map to the vertices to which v_i is connected to, with a '1' indicating the presence of an edge and a '0' the absence. For example, the adjacency of vertex v_5 with vertices v_0, v_2, and v_6 is encoded as the integer 69, which corresponds to the binary string 1000101. Here, the bit positions 0, 2, and 6 (from right to left) are set to '1', indicating connections to vertices v_0, v_2, and v_6, respectively. This compact representation allows for a substantial reduction in the memory footprint for large graphs while still supporting rapid adjacency queries typical in graph traversal operations.



	Edges E(v)
0	1, 3, 5
1	0, 3, 8
2	5, 7, 8
3	0, 1, 4, 6
4	3, 6, 9
5	0, 2, 6
6	3, 4, 5
7	2
8	1, 2, 9
9	4, 8

v	9	8	7	6	5	4	3	2	1	0	Edges E(v)
0	0	0	0	0	1	0	1	0	1	0	42
1	0	1	0	0	0	0	1	0	0	1	265
2	0	1	1	0	1	0	0	0	0	0	416
3	0	0	0	1	0	1	0	0	1	1	83
4	1	0	0	1	0	0	1	0	0	0	584
5	0	0	0	1	0	0	0	1	0	1	69
6	0	0	0	0	1	1	1	0	0	0	56
7	0	0	0	0	0	0	0	1	0	0	4
8	1	0	0	0	0	0	0	1	1	0	518
9	0	1	0	0	0	1	0	0	0	0	272

FIGURE 1. **BitVertex representation of the example graph [24].**

B. SCALABLE COLOR CLASS REPRESENTATION OF BITVERTEX DESIGN FOR LARGE GRAPHS

To effectively manage graphs with more than 32 vertices, we propose a novel representation for color classes within our graph coloring algorithm. Each color class is represented as a sequence of 32-bit integers. The number of integers m allocated for each color class is determined by the total count of vertices $|V|$ divided by 32, calculated as Eq. 3a. This scheme allows each integer to represent a distinct block of 32 vertices, with the first integer corresponding to vertices v_0 through v_{31}, the second to vertices v_{32} through v_{63}, and so on. The precise location of a vertex within these color classes is determined through a two-step computational process. First, the integer index $integer_{id}$ that contains the bit representing the vertex is identified by Eq. 3b where $\gg$ is the right shift operation, equivalent to dividing $vertex_{id}$ by 32. The bit index bit_{id} within the target integer is subsequently calculated using Eq. 3c with & representing the bitwise AND operation, effectively computing $vertex_{id}$ modulo 32.

$$m = \left\lceil \frac{|V|}{32} \right\rceil, \tag{3a}$$

$$integer_{id} = vertex_{id} \gg 5, \tag{3b}$$

$$bit_{id} = vertex_{id} \& 31. \tag{3c}$$

Through these steps, the representation avoids overflow and scales to handle an arbitrary number of vertices by ensuring each vertex is unambiguously mapped to a specific bit within an integer. This systematized approach is illustrated in Fig. 2. For example, consider the vertex v_{45}, which belongs to the second block of vertices. The $integer_{id}$ for v_{45} is calculated as $101101 \gg 5$, which equals 1, indicating that v_{45} is represented in the second integer. The bit_{id} is calculated as $45\&31 = 101101\&011111$, which equals 13, indicating that the 14th bit (considering a zero-based index) of the second integer is set to 1 as shown in Fig. 2.

C. INDIVIDUAL REPRESENTATION AND GENERATION

The conventional approach to representing individuals in hybrid evolutionary algorithms for graph coloring problems employs a two-dimensional array to denote color classes, with each class comprising a list of vertices assigned a unique color. Such an array is inherently integer vertex-based, with each vertex identified by an integer within its respective color class. Fig. 3 illustrates this representation through individuals S_1 and S_2, where S_1 is defined by three color classes containing 3, 3, and 4 vertices respectively. Accordingly, S_1 requires 10 integers for representation, accounting for 40 bytes of memory allocation.

In contrast, the *BitVertex*-based representation introduces a more compact binary encoding strategy. Here, each color class is encoded as an integer, with the binary bits of the integer signifying the presence ('1') or absence ('0') of each vertex in the color class. This method is demonstrated in Fig. 3 where individual S_1 is represented by a list of three integers 81, 388, 554, translating to a more economical memory usage of merely 12 bytes. This binary encoding not only reduces space requirements but also aligns with computer architecture efficiencies in handling binary data operations.

In the individual initialization phase of our proposed algorithm, individuals in the population are denoted by S_i, each comprising a set of color classes $\{C_1, C_2, \ldots, C_k\}$. Each class C_j encompasses a subset of non-adjacent vertices, thereby avoiding conflicts and adhering to the algorithm's constraint of k color classes. This approach mirrors the population initialization method employed by the InCEA algorithm, ensuring consistency with established methodologies while introducing our novel enhancements.

To facilitate a diverse initial population, we employ a spill degree metric, proposed in [24], to sort the vertices. This metric is calculated via three distinct methods, as defined in equations (4a), (4b), and (4c), with respective application

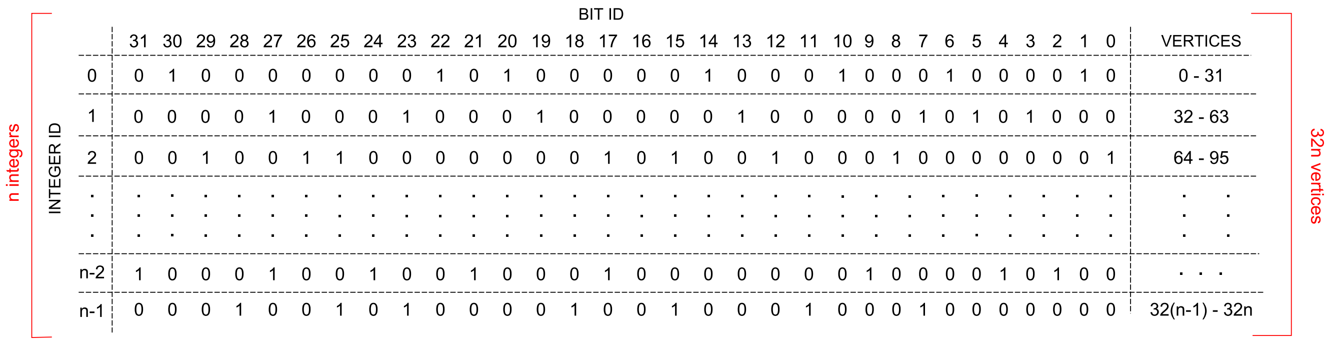


FIGURE 2. Scalable representation of color classes using a set of 32-bit integers. Each integer corresponds to a block of 32 vertices, allowing for the representation of large graphs beyond the limits of a single integer.

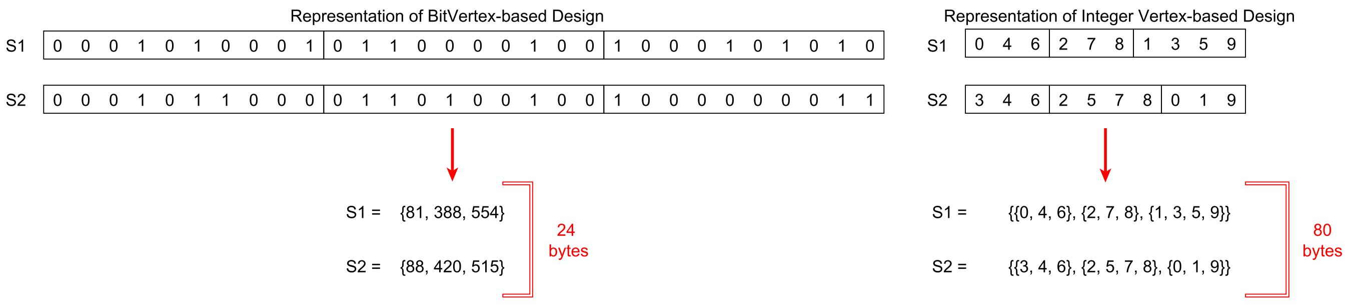

Representation of BitVertex-based Design
Representation of Integer Vertex-based Design
S1 = {81, 388, 554}
S2 = {88, 420, 515}
24 bytes
S1 = {{0, 4, 6}, {2, 7, 8}, {1, 3, 5, 9}}
S2 = {{3, 4, 6}, {2, 5, 7, 8}, {0, 1, 9}}
80 bytes

FIGURE 3. Comparison of integer vertex and BitVertex individual representations.

to 40%, 40%, and 20% of the individuals, ensuring a heterogeneous set of initial individuals.

Vertices are ranked in descending order of their spill degrees. During the generation of an individual, the algorithm selects the unassigned vertex with the highest spill degree and attempts to assign it to the first available conflict-free color class. If no such class exists, the vertex is placed in a random color class. This procedure is iterated until all vertices are assigned to one of the color classes, resulting in a complete and valid individual representation for the population as given in Algorithm 2.

$$\text{S\_Degree}_1(v_i) = \text{w}(v_i) \times \text{Degree}(v_i), \tag{4a}$$
$$\text{S\_Degree}_2(v_i) = \text{w}(v_i) \times \text{Degree}(v_i)^2, \tag{4b}$$
$$\text{S\_Degree}_3(v_i) = \text{w}(v_i). \tag{4c}$$

D. BitVertex CROSSOVER OPERATOR

Our crossover operation is inspired by the Integrated Crossover Operator (InCX) used in the InCEA algorithm and InCX is specifically adapted for the BitVertex design through the use of bitwise operations such as AND, OR, XOR and NOT. This adaptation significantly enhances the computational efficiency of the InCX operator by reducing the complexity of key computations involved in the process, as outlined in Table 1. Furthermore, this BitVertex-adapted crossover operator introduces improved search mechanisms to produce offspring with lower fitness values. Thus, we have named our new crossover operator the BitVertex Crossover Operator (BitCX).

BitCX operates on two parent individuals, $S_1 = \{C_0^1, C_1^1, \ldots, C_{k-1}^1\}$ and, $S_2 = \{C_0^2, C_1^2, \ldots, C_{k-1}^2\}$ with a graph G, as shown in Algorithm 3. Each parent, structured in the BitVertex format, comprises k color classes. The crossover process combines color classes from both parents to create offspring. In each combination, one color class is randomly selected from each parent, and the vertices within these classes are combined into the corresponding color class of the offspring using the OR operation. If conflicts arise in the color class of the offspring, the conflict number of each vertex is counted using Operation 3, which has a complexity of $O(|V| \log |V|)$, and the vertex with the maximum conflict number is moved to the pool. This process is iteratively applied until all conflicts are resolved in the color class. After each combination, except the first one, a search back operation is performed. This operation attempts to reintegrate vertices from the pool back into one of the color classes either is conflict-free or is in conflict with only a single vertex which has a lower weight than the vertex currently in the pool. The search back operation is given in more detail in the following subsection. At the end of each combination process, vertices placed in the color classes of the offspring or in the pool are removed from the parents, with a complexity of $O(k)$, to prevent recombination. This iterative process continues until k conflict-free color classes are established within the offspring. At the end of the crossover operation, if one or

Algorithm 2 Initial Population Generation

```
Input: Graph $G(V, E)$, number of color classes $k$,
       population size $p$
Output: Initial population $Population$
$Population \leftarrow \emptyset$
for each vertex $v \in V$ do
    $S\_Degree_1(v) \leftarrow w(v) \times Degree(v)$
    $S\_Degree_2(v) \leftarrow w(v) \times Degree(v)^2$
    $S\_Degree_3(v) \leftarrow w(v)$;
end
$L1 \leftarrow$ vertices sorted by $S\_Degree_1(v)$ $L2 \leftarrow$ vertices
sorted by $S\_Degree_2(v)$ $L3 \leftarrow$ vertices sorted by
$S\_Degree_3(v)$
for $i = 1$ to $p$ do
    $VList \leftarrow \begin{cases} L1 & \text{if } i \leq 0.4p \\ L2 & \text{if } 0.4p < i \leq 0.8p \\ L3 & \text{otherwise} \end{cases}$
    $S_i \leftarrow$ Initialize an empty individual with $k$ color
    classes $C_j$ where $1 \leq j \leq k$
    for each vertex $v$ in $VList$ do
        for each color class $C_j$ in $S_i$ do
            if $v$ can be added to $C_j$ without conflicts then
                Add $v$ to $C_j$ break
            end
        end
        if $v$ not added then
            Assign $v$ to a random color class
        end
    end
    Add $S_i$ to $Population$
end
```

TABLE 1. Time complexity comparison: array-based vs. bitwise operations in algorithms.

Array-based operations	Bitwise operations
Operation 1: Combining the vertices in two color classes C_1 and C_2	

`For each vertex` $v \in G(V,E)$
  `If` $v \in$ `colors` C_1 `or` C_2
    `Assign` v `to new color` C_3
  `End If`
`End Loop`
	$C_3 = C_1$ | C_2

Operation 2: Removing the vertices in the pool or offspring from parents S_1 or S_2	

`For each vertex` $v \in G(V,E)$
  `For each color` $C_1 \in S_1$
    `If` $v \in C_1$ `& pool`
      `Remove` v `from` C_1
    `End If`
  `End Loop`
`End Loop`
	`For each color` $C_1 \in S_1$
	  $C_1 = C_1$ `&` $\sim$ `pool`
	`End Loop`

Operation 3: Calculating conflicts between the vertices in the same color	

`For each vertex` $v_1 \in G(V,E)$
  `If` $v_1 \in$ `color` C
    `For each vertex` $v_2 > v_1 \in V$
      `If` $v_2 \in C$ `and e(`v_1, v_2`)` $\in E$
        `conflicts_num[`v_1`]++`
        `conflicts_num[`v_2`]++`
      `End If`
    `End Loop`
  `End If`
`End Loop`
	`For each vertex` $v \in G(V,E)$
	  `If` 2^v `& color` C
	    `conflicts = edges(`v`) &` C
	    `While conflicts`
	      `conflicts & = (conflicts - 1)`
	      `conflicts_num[v]++`
	    `End Loop`
	  `End If`
	`End Loop`

Time Complexity	
Operation 1: O($\|V\|$) Operation 2: O($\|V\| \cdot k$) Operation 3: O($\|V\|^2$)	Operation 1: O(1) Operation 2: O(k) Operation 3: O($\|V\| \cdot log\|V\|$)

more vertices remain in conflict with all color classes, they are kept in the pool to be placed to the offspring at the local search operation.

Algorithm 3 BitVertex Crossover Operator

```
Input: Graph $G$, number of color classes $k$, 1st parent
       $S_1 = \{C_0^1, C_1^1, \dots, C_{k-1}^1\}$ in bitwise format, 2nd
       parent $S_2 = \{C_0^2, C_1^2, \dots, C_{k-1}^2\}$ in bitwise
       format
Output: An offspring $S_0 = \{C_0, C_1, \dots, C_{k-1}\}$ in
        bitwise format, an updated pool $P$
Create an empty pool P as zero: $P \leftarrow 0$
Create a bit set of selected vertices: $V_s \leftarrow 0$
Mark color classes $\in S_1, S_2$ as unselected:
$U^1 \leftarrow 0$ and $U^2 \leftarrow 0$
for $i \leftarrow 0$ to $k - 1$ do
    Set ith color class of $S_0$ $C_i$ as zero: $C_i \leftarrow 0$
    Select an unselected color class $C_x^1$ from $S_1$
    Select an unselected color class $C_y^2$ from $S_2$
    Mark $C_x^1, C_y^2$ as selected: $U^1 \leftarrow U^1$ AND $2^x$,
    $U^2 \leftarrow U^2$ AND $2^y$
    Combine color classes of parents using bitwise:
    $C_i \leftarrow C_x^1$ OR $C_y^2$ (Operation 1)
    Mark the vertices as selected: $V_s \leftarrow V_s$ OR $C_i$
    if $P \neq 0$ then
        $C_i \leftarrow C_i$ OR $P$ (Operation 1)
        P $\leftarrow 0$
    end
    while $C_i \neq$ conflict-free do
        Calculate the maximum conflicting vertex as
        $v_{max}$ (Operation 3)
        Throw $v_{max}$ into pool: P $\leftarrow$ P OR $2^{v_{max}}$
        Remove $v_{max}$ from the color: $C_i \leftarrow$
        $C_i$ XOR $2^{v_{max}}$
    end
    if $i \geq 1$ then
        $SearchBackOperation(i, S_0, P, G)$
    end
    Remove the vertices $\in P|C_i$ from the parents
    (Operation 2)
end
```

E. IMPROVED SEARCH BACK OPERATION

During the crossover operation BitCX, whenever a conflict arises in the ith color class C_i, at the completion of C_i where $1 \leq i < k$, it may result in one or more vertices being relocated to the pool. Under such circumstances, investigating the potential of reallocating any vertex in the pool to a previous color class where no conflicts occur, reduces the overall fitness value of the offspring. Therefore, InCEA [10] proposes *Search Back Operation (SB)* to integrate the vertices in the pool to the previous conflict-free color classes before starting the next combination of the crossover operation.

In this study, we introduce *Improved Search Back operation (ISB)* which refines the original SB strategy. Our enhancement involves introducing a new criterion for vertex reallocation. The ISB algorithm is the subpart of BitCX

Algorithm 4 Improved Search Back Technique

```
Input: Number of color classes currently produced $i$, the
       offspring $S_0 = \{C_0, \ldots, C_{i-1}\}$, the pool $Pool$, the
       graph $G$
Output: Updated $S_0$, $Pool$
foreach vertex $v_p$ in Pool do
    $C_{best} \leftarrow$ -1
    $v_{best} \leftarrow$ -1
    $bestWeight \leftarrow \infty$
    $CF \leftarrow$ false
    for $j \leftarrow 0$ to $i-1$ do
        $conflicts\_num[v_p] \leftarrow$ Operation 3 for $C_j$
        if $conflicts\_num = 0$ then
            $Pool \leftarrow Pool$ XOR $2^{v_p}$
            $C_j \leftarrow C_j$ OR $2^{v_p}$
            $CF \leftarrow$ true
            break
        end
        if $conflicts\_num = 1$ then
            $conflict\_weight \leftarrow w(\log_2(conflicts[v_p]))$
            if $bestWeight > conflict\_weight and w(v_p) >$
            $conflict\_weight$ then
                $C_{best} \leftarrow C_j$
                $v_{best} \leftarrow \log_2(conflicts[v_p])$
                $bestWeight \leftarrow conflict\_weight$
                break
            end
        end
    end
    if $!CF and C_{best} \neq -1$ then
        $Pool \leftarrow (Pool$ XOR $2^{v_p})$ OR $2^{v_{best}}$
        $C_{best} \leftarrow (C_{best}$ XOR $2^{v_{best}})$ OR $2^{v_p}$
    end
end
```

operation and it iterates through each vertex v_p in the pool at the end of each color class combination C_i. The conflicts between each v_p and each previously produced color class C_j are calculated using Operation 3, detailed in Table 1, with a complexity of $O(k \cdot |V| \log |V|)$ for all color classes. After Operation 3, if a vertex v_p encounters more than one conflict across all C_j, it remains in the pool for subsequent combinations. Conversely, if a conflict-free C_j exists for v_p, the vertex is immediately assigned to C_j using OR operation, which has a complexity of $O(1)$.

A novel criterion in this study, is applied when exactly one conflict is detected between v_p and a color class C_j. If the conflicting vertex v_{best} in C_j has a lower weight value than v_p, the two vertices are swapped by using bitwise OR and XOR operations, each with a complexity of $O(1)$, with v_p being integrated into C_j and v_{best} moving to the pool. This criterion aims at optimizing the allocation of vertices based on their weight, further enhancing the overall fitness value of the offspring. Given the detailed breakdown of the operations and their respective complexities, the overall time complexity of the BitVertex Crossover Operator (BitCX) is $O(k^2 \cdot |V| \log |V|)$ whereas the InCX operator [10] has a time complexity of $O(k \cdot |V|^3)$. This complexity reflects the efficiency of the BitCX algorithm in handling large graphs, utilizing bitwise operations to streamline the manipulation of vertices and color classes.

F. ILLUSTRATION OF BitCX AND ISB OPERATORS

The ISB algorithm, outlined in Algorithm 4, represents a significant step forward in the strategic manipulation of vertices during BitCX. By refining the process of addressing conflicts and reallocating vertices through algorithmic methods, ISB contributes to the production of higher-quality offspring, as highlighted in Fig. 6 and Fig. 7.

The process of BitCX is illustrated in Fig. 4. The graph in Fig. 1 and its two parent configurations in Fig. 4 are represented to demonstrate HEA algorithm [24]. Additionally, three subsequent studies [23], [25], including the InCEA [10], have analyzed the same graph and parents to evaluate the efficacy of their crossover operators as CFPX [24], COPX [25], PBC [23] and InCX [10], respectively. To ensure a standardized basis for comparison, we have also applied BitCX to the same input arguments. We higlight the performance enhancements achieved from CFPX [24] to BitCX in terms of solution quality and computational efficiency, especially when comparing InCX and BitCX.

In the crossover example illustrated in Fig. 4, the operator selects the second color class C_1^1 from the first parent S_1 and the third color class C_2^2 from the second parent S_2. These classes are combined using the bitwise OR operation to form the first color class C_0 of the offspring S_0. The connections between the vertices within C_0 are evaluated for each vertex by applying the bitwise AND operation which is detailed in Fig. 5 and Operation 3 in Table 1. For example, vertex v_8, which is connected to vertices v_0, v_1, and v_9 in the graph given in Fig. 1, has its edges represented in BitVertex format as 1000000110, equivalent to the decimal number 518. To identify the vertices in C_0 that conflict with v_8, a bitwise AND operation is performed between $Edges(v_8)$ and C_0. This operation reveals that all three conflicting vertices are present in C_0, as indicated by the Conflicts(v_8). The bits set to 1 in Conflicts(v_8) are highlighted, and their count is determined using Brian Kernighan's Algorithm [14] as described in Operation 3 in Table 1. The time complexity of Operation 3 is O($|V| \cdot \log(|V|)$) [14], in contrast to O($|V|^2$) for the array-based implementation of the same operation. After calculating the conflicts for all vertices in C_0, v_8 is identified as having the maximum number of conflicts and is thus moved from C_0 to the pool P using bitwise XOR operation. This conflict resolution process continues until vertices v_7 and v_1 are also transferred to the pool. At the end of the first combination phase, C_0 comprises vertices v_0, v_2, and v_9 (represented as 100000101), while the pool contains v_1, v_7, and v_8 (0110000010). Before the execution of the second combination, vertices in C_0 and

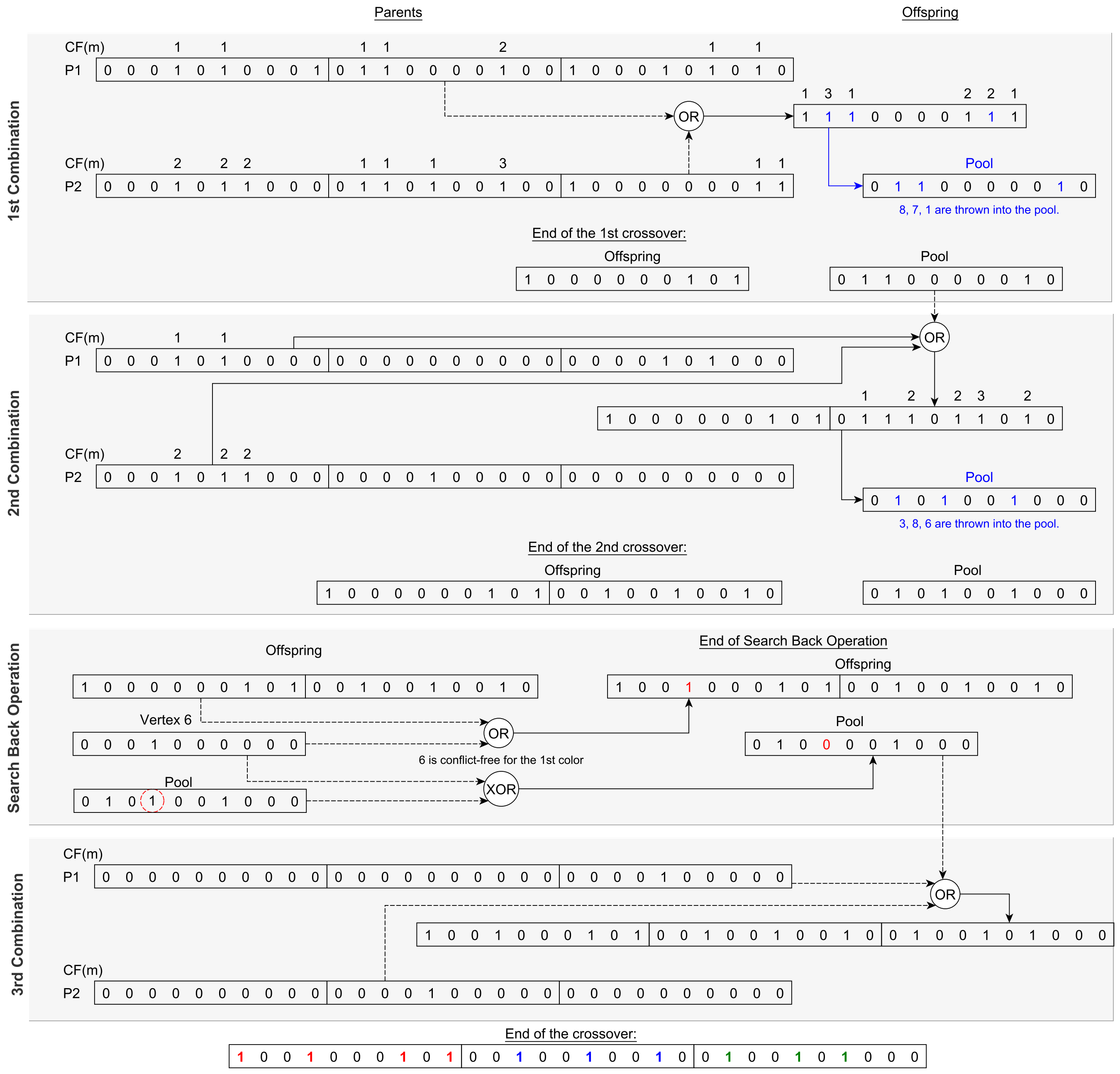


FIGURE 4. **Illustration of BitCX operation using two parent configurations and a weighted graph [24].**

the pool are temporarily excluded from color classes of the parents ($C_i^{1,2}$) using the AND NOT(Pool) operation, as detailed in Operation 2 in Table 1. The time complexity for Operation 2, when implemented using bitwise operations, is O(k), where k denotes the number of color classes in each parent. In contrast, the array-based implementation of Operation 2 has a time complexity of O($|V| \cdot k$).

At the second combination, the first color class from the first parent C_0^1 and the first color class from the second parent C_0^2 are selected and the vertices in these color classes v_3, v_4 and v_6 are merged in the second color class of the offspring C_1. Since there are vertices in the pool from the previous combination, v_1, v_7 and v_8 are also moved to C_1. Operation 3 is executed on C_1 until v_3, v_6 and v_8 are thrown to the pool to obtain C_1 as a conflict-free color class.

At the end of the second combination, the search back operation is performed on the vertices that have already been mapped to the pool, v_3 and v_6 to check if they can be placed in C_0. The vertex v_3 has only one edge with v_1 in C_0 but the weight of v_1, $w(v_1)$, is higher than $w(v_3)$ so v_1 and v_3 cannot be swapped between C_0 and the pool (see Algorithm 4). Conversely, v_6 does not share edges with any vertices in

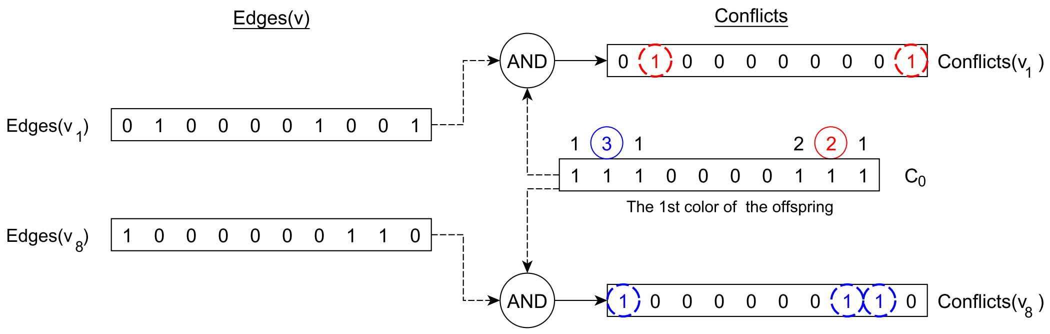


FIGURE 5. Example of calculation of the conflicts in the same color using Operation 3 in Table 1.

C_0 and is therefore moved to C_0 using bitwise OR and XOR operations, as illustrated in the search back phase in Fig. 4. In the final combination, only vertex v_5 remains unassigned in the parents. This vertex is merged with v_3 and v_8 into the third color class of the offspring, C_2. Since C_2 is already conflict-free and the pool is empty, the crossover operation is terminated for this example.

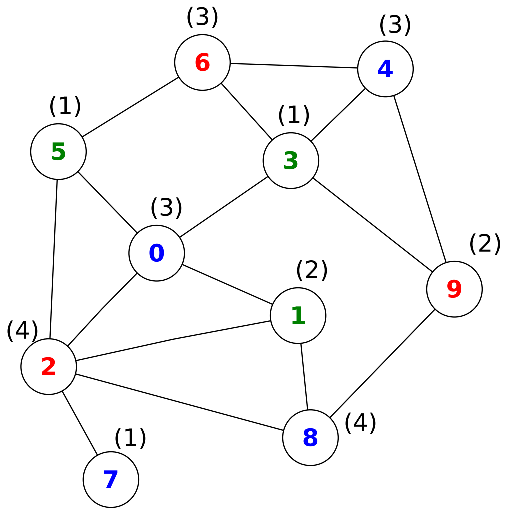


FIGURE 6. The Example graph with some edge and vertex weight changes.

In the case illustrated by Fig. 4, the new criterion introduced by the ISB algorithm is not applicable, as the scenario doesn't align with its prerequisites. Furthermore, the conclusion of the crossover operation yields a conflict-free solution so the local search strategy with bitwise operations couldn't be shown. Therefore, we present another example to demonstrate the efficiency of the ISB algorithm and the local search operation. To facilitate this, some modifications are made to the graph shown in Fig. 1, resulting in a new graph illustrated in Fig. 6. In the new graph, the weight values of vertices v_0, v_1, and v_8 have been updated to 3, 2, and 4, respectively. Regarding the edges, the connection between vertices v_1 and v_3 has been removed, while two new edges have been added: one connecting v_2 with v_0, and another connecting v_2 with v_1. Considering the graph in Fig. 6, the same parent configurations as in Fig. 4 are used and BitCX, ISB and the local search operations are executed on the parents and illustrated in Fig. 7.

To form the first color class C_0 of the offspring, the first color class from the first parent C_0^1, and the third color class from the second parent, C_2^2 are selected randomly. The vertices from these color classes v_0, v_1, v_4, v_6, and v_9 are combined in C_0 using OR operation. Due to conflicts within C_0, vertices v_4 and v_1 are moved to the pool, leaving v_0, v_6, and v_9 in C_0.

For the next step, the second color classes from both parents, C_1^1 and C_1^2 are merged, along with v_1 and v_4 from the pool, to create the second color class, C_1. This process results in v_2 and v_1 being moved to the pool due to conflicts, prompting the start of the ISB algorithm.

The ISB algorithm, illustrated in the red phase in Fig. 7, skips v_1 (already in the pool from the first combination) and focuses on v_2. Using the bitwise operation of Operation 3 in Table 1, it detects a single conflict between v_0 and v_2. Given that v_2 has a higher weight (4) than v_0 (3), as shown in Fig. 6, the vertices are swapped between the pool and C_0. Therefore, v_2 is removed from the pool by XOR operation and added to C_0 using OR operation. The converse procedure is applied on v_0 and the designs of these bitwise operations are drawn in Fig. 7.

In the final combination, only v_3 remains unassigned in the parents. It is combined with v_0 and v_1 from the pool into the last color class, C_2, of the offspring. v_0 is moved back to the pool due to conflicts with v_1 and v_3. The ISB operation searches C_0 and C_1 for a place for v_0. Since v_0 was previously removed from C_0, it checks C_1. A conflict between v_0 and v_5 prompts a swap because v_5 has the lowest weight (1). Thus, at the end of the crossover, v_5 ends up in the pool, and the local search operation is applied to the offspring.

To compare the performance of the InCX and BitCX operators based on fitness values, we applied both operators to the same parent combinations, as shown in Fig. 7, to generate offspring. The offspring produced using the InCX operator is shown in Fig. 8. The InCX operator results in a solution where vertex v_1 remains in the pool, leading to the fitness value equal to $w(v_1)$, which is 2. Conversely, the BitCX operator yields an offspring with the fitness value

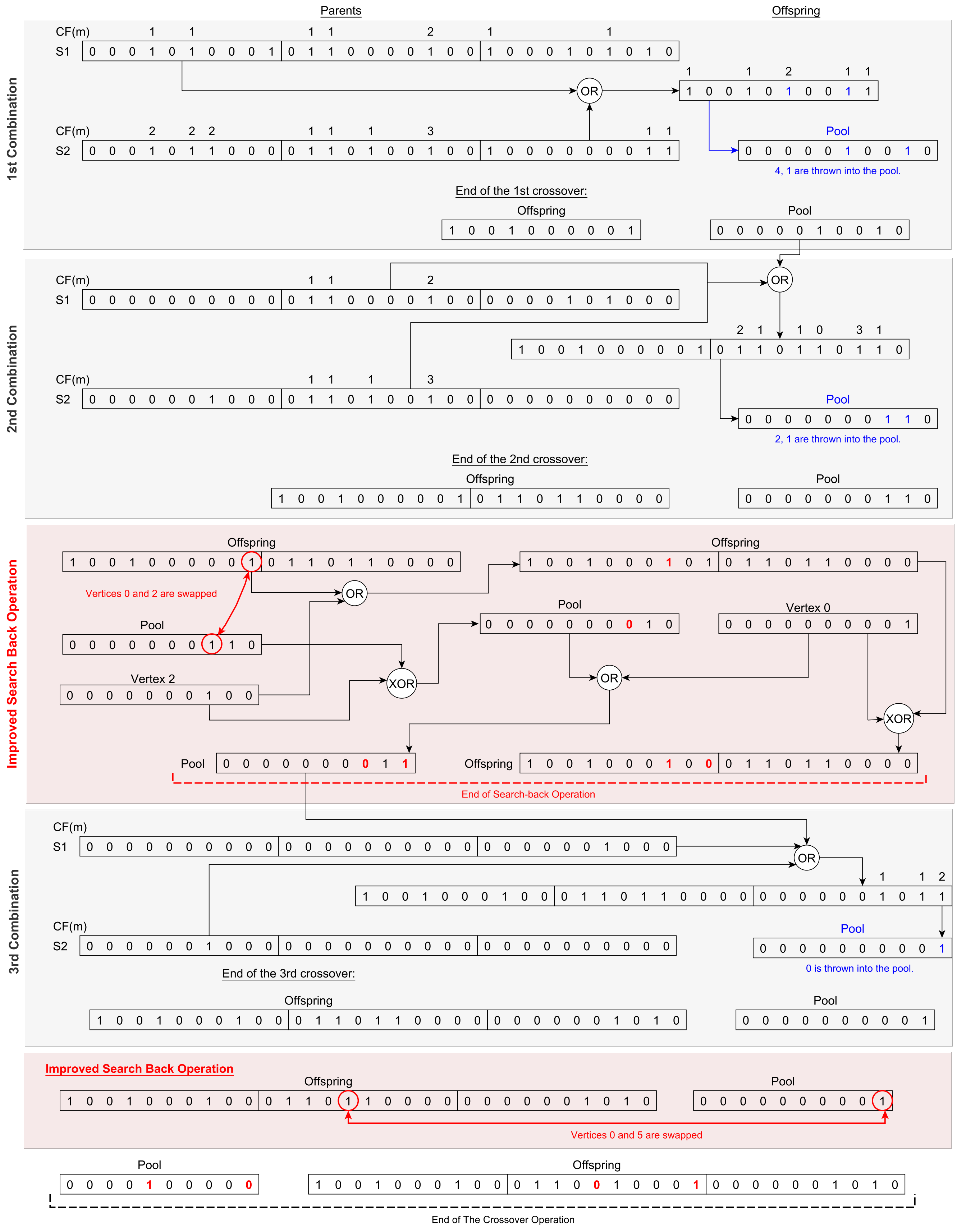

1st Combination
Parents
Offspring
CF(m)
S1
S2
OR
Pool
4, 1 are thrown into the pool.
End of the 1st crossover:
2nd Combination
2, 1 are thrown into the pool.
End of the 2nd crossover:
Improved Search Back Operation
Vertices 0 and 2 are swapped
Vertex 2
Vertex 0
XOR
End of Search-back Operation
3rd Combination
0 is thrown into the pool.
End of the 3rd crossover:
Improved Search Back Operation
Vertices 0 and 5 are swapped
End of The Crossover Operation

FIGURE 7. **BitCX applies on the same parent configurations considering the new graph in Fig. 6.**

of 1, as it leaves vertex v_5 in the pool. For a fair comparison between the two operators, the solution obtained with BitCX is also presented in an array format in Fig. 8. This analysis demonstrates that the ISB operator effectively reduces the fitness value of the solution under identical test conditions.

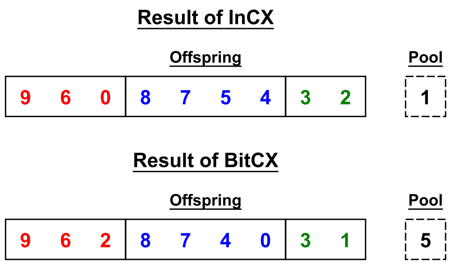


FIGURE 8. The solutions obtained after InCX and BitCX operators using the graph in Fig. 6.

G. LOCAL SEARCH OPERATION

The local search mechanism in this study mimics the W-SWAP proposed in the InCEA algorithm [10], introducing a method based on bitwise operations named the *Bit-SWAP*. This strategy is activated when one or more vertices remain in the pool following the BitCX phase. Vertices within the pool are sorted by their weights in descending order to ideally position the vertex with the highest weight value first in a color class in S_0. For each vertex v_p in the pool, the Bit-SWAP calculates the sum of weights of any conflicting vertices across all k classes within S_0, aiming for the lowest sum. If the lowest sum is smaller than the weight of v_p, then a swap is initiated, where the conflicting vertices are relocated to the pool, and v_p is inserted into the color class utilizing bitwise OR and XOR operations as also defined in the ISB operator. In the scenario where no suitable color class is found for the swap, v_p is taken out of the *Pool* and added to a list, denoted as *Uncolored*, which stores vertices that couldn't be colored for S_0. The local search proceeds by processing each vertex in the *Pool* until the *Pool* becomes empty. All vertices in *Uncolored* list are used for calculating the fitness of the offspring, which is explained in the next section. Eventually, the vertices in *Uncolored* are randomly assigned to color classes, concluding the process.

Fig. 9 demonstrates the performance of Bit-SWAP on the offspring, denoted S_0, and the pool, both of which are the outcomes of the BitCX operation shown in Fig. 7. The pool contains a single vertex, v_5 whose conflicts with the color classes are calculated. Upon examination of each color class within the offspring, C_2 is identified as having no conflicts with vertex v_5. Consequently, v_5 is integrated into C_2 through OR operation and concurrently removed from the pool with XOR operation.

After the sequence of BitCX, ISB, and Bit-SWAP operations, BitEA is completed and a feasible solution in which all vertices are successfully colored, is identified. The result of BitEA is also shown in an array-based format in Fig. 10 to observe the performance comparison with InCEA solution. While BitEA achieves the optimal solution, InCEA is not successful to color all vertices within three color classes under the same experimental conditions. These findings underscore the effectiveness of integrating the ISB operator with local search strategies to enhance result quality.

H. FITNESS FUNCTION AND POPULATION UPDATE

As we formulized the fitness function of the problem in Eq. 2 as the objective of the problem, the fitness value of a solution for a given graph is calculated by summing the weights of vertices which are assigned to *Uncolored* list. The optimal solution of a graph using k color classes is described with a fitness value of 0, which means that all vertices are successfully colored.

For our example graph, the fitness function computes the solution of InCEA which is depicted in Fig. 10 and obtains the fitness value $w(v_1) = 2$ since vertex v_1 belongs to *Uncolored* list. Conversely, BitEA successfully assigns all vertices to one of $k = 3$ color classes, achieving an optimal fitness value of 0.

When the fitness value of the offspring outperforms that of either or both parents, the algorithm selects the parent with the highest fitness value. This selected parent is then replaced by the offspring, adhering to the replacement strategy of BitEA.

V. EXPERIMENTAL STUDY

In this study, we conducted a comprehensive experimental analysis to compare the performance of InCEA [10] and our algorithm BitEA. The results demonstrate the effectiveness of the BitVertex representation and bitwise operations in BitEA in terms of computational time, as well as the ISB operator's impact on solution quality.

A. EXPERIMENTAL SETUP

The algorithms are evaluated using 150 InCEA benchmarks introduced in [10] and 73 DIMACS benchmarks. Both algorithms are implemented in C[1] and compiled using gcc on a computer equipped with a 3.3 GHz Intel Xeon 2670 V2 CPU and 64 GB of RAM.

The InCEA benchmarks are publicly available.[2] These benchmarks are named as "INCEA$n.\alpha.x$.col", where n represents the number of vertices, α denotes the edge density as a percentage, and x signifies the x^{th} graph generated with the same parameters, with $1 \leq x \leq 5$. The benchmarks were created across five distinct values of n and six values of α, each replicated five times, resulting in a total of 150 unique benchmarks. Additionally, weight values were assigned to each vertex within these benchmarks by the γ parameter. The specific values for these parameters are detailed in Table 2.

The experimental setup in this study involves 73 instances from the DIMACS benchmark suite, categorized into five different types based on their origin and characteristics.

[1]InCEA: https://gitlab.com/gizemsungu/incea, BitEA: https://github.com/ic-cad/BitEA
[2]Benchmark instances: https://gitlab.com/gizemsungu/incea

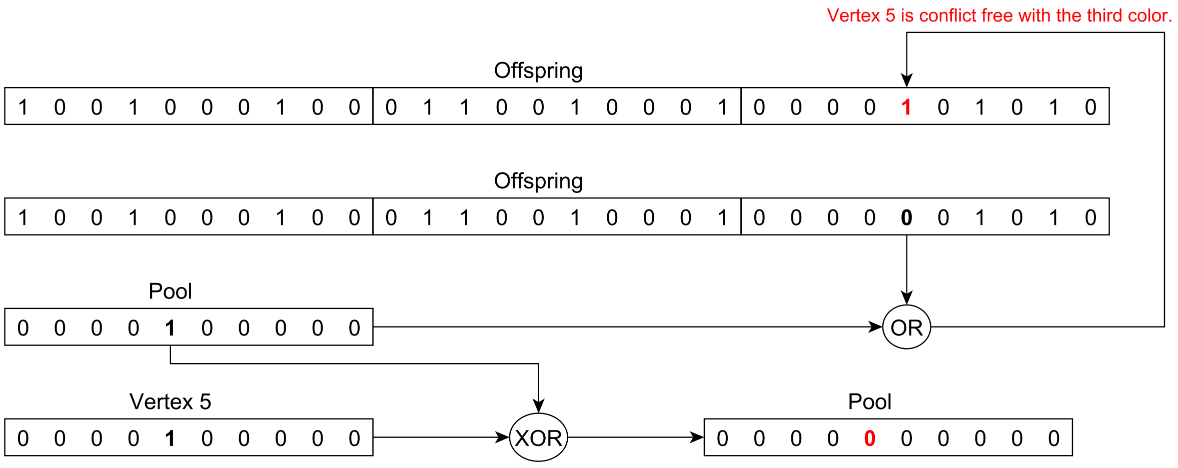


FIGURE 9. Performance of Bit-SWAP technique on the offspring after the BitCX operation in Fig. 7 using the graph in Fig. 6.

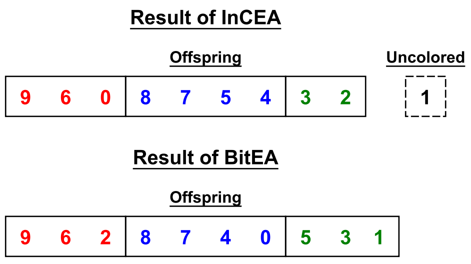


FIGURE 10. Comparison of solutions obtained by InCEA and BitEA represented in Integer Vertex-based format using the graph in Fig. 6.

TABLE 2. Parameters of InCEA benchmarks [10].

Parameters	Description	Values
n	number of vertices	100, 200, 300, 400, 500
α	edge density	0.10, 0.25, 0.45, 0.60, 0.75, 0.90
β	color class density	0.04, 0.08, 0.10, 0.15, 0.20
γ	weight of vertices	[1..10]

Graphs related to the register allocation problem are denoted as ‘Rn_$\alpha g\{b\}$.col’, where α represents the edge density of the graph, and the suffix ‘g’ or ‘gb’ indicates a range of vertex weights with ‘g’ for weights of $\gamma = [1..5]$ and ‘gb’ for $\gamma = [1..20]$. GEOM graphs, which are geometric in nature, are designated as ‘GEOM$n\{a, b\}$.col’, with ‘a’ and ‘b’ signifying the edge density percentage. The queen graphs, originating from the n $\times$ n chessboard problem, are labeled as ‘queen$n$_n.col’. The DSJC graphs, used in the study of simulated annealing techniques, are named ‘DSJCn_$\alpha g\{b\}$.col’. Lastly, the myciel graphs, derived from the Mycielski transformation, are identified as ‘myciel$\{5, 6, 7\}g\{b\}$.col’.

B. EXPERIMENTAL RESULTS

1) PERFORMANCE EVALUATION OF BitEA AND InCEA VARYING ON TIME

The experimental analysis of BitEA and InCEA, as presented in Fig. 11, investigates the convergence performance across six different DIMACS benchmark graphs. The aim of this test is to determine the time and number of iterations required for both algorithms to stabilize in terms of the fitness value of their proposed solutions. The graphs tested include R100 with various edge density ranges (1gb, 5gb, 9gb), a geometric graph GEOM120a, and queen graphs with 100 vertices (queen10_10g and queen10_10gb). The performance of the algorithms are measured in terms of percentage improvement from the best fitness of the individual in the initial population over time. An improvement in performance corresponds to a decrease in the fitness value. Therefore, the trajectories in the plots are descending, indicating better performance as the fitness value decreases. The algorithms are run for 100 seconds for each graph, with the maximum number of seconds varying for each plot. The trajectories are drawn for both algorithms until InCEA reached its best fitness for the related graph. The performance trajectories demonstrate that BitEA reaches a plateau quickly, indicating early convergence, while InCEA shows a more gradual improvement towards the fitness value. For the R100_1gb graph, BitEA shows a rapid fitness improvement, plateauing beyond approximately 1.5 seconds, whereas InCEA reaches the same improvement level around 8 seconds. For the R100_5gb and R100_9gb graphs, a similar trend is observed: BitEA quickly stabilizes, while InCEA takes longer to converge but continues to improve gradually over time. In the case of the GEOM120a graph, both algorithms show a rapid initial improvement, but BitEA reaches a plateau faster than InCEA. InCEA continues to improve over a longer period before stabilizing. For the queen10_10g and queen10_10gb graphs, BitEA again exhibits early convergence, while InCEA takes more time to reach its best fitness value. Overall, the comparative analysis reveals that BitEA generally converges faster than InCEA across different types of graphs. This early convergence indicates that BitEA can quickly provide high-quality solutions, making it particularly effective for scenarios requiring rapid optimization. Moreover, BitEA reaches its best solutions for most of the related graphs before

10000 iterations. These findings determine that setting the iteration number to 10000 in subsequent experimental studies ensures both efficiency and effectiveness in reaching optimal solutions with BitEA.

2) PERFORMANCE COMPARISON TESTS ON InCEA BENCHMARKS

Table 3 and Table 4 present a comparative analysis of InCEA and BitEA, examining their performance across a range of graph sizes, represented by the number of vertices (n), and two distinct density metrics: edge density (α) and color class density (β), respectively. The evaluation metrics include the fitness values, the number of uncolored vertices (denoted as # of U. Vertices), the execution time in seconds (Time(s)) and the resulting speedup of BitEA over InCEA.

TABLE 3. Comparative performance analysis of InCEA and BitEA across different graph sizes n and edge densities α.

n	α	Fitness Values		# of U. Vertices		Time(s)		Speed
	(%)	InCEA	BitEA	InCEA	BitEA	InCEA	BitEA	up
100	10	3.27	3.25	1.61	1.65	0.96	0.19	5
	25	32.09	32.03	9.9	9.78	3.98	0.73	5.4
	45	85.54	84.89	24.67	24.7	14.24	2.11	6.7
	60	131.59	130.97	36.03	36.14	27.56	3.25	8.5
	75	193.97	193.65	50.96	51.03	30.9	4.41	7
	90	266.01	266.18	65.79	65.76	31.67	5.15	6.1
200	10	0.86	0.87	0.62	0.65	9.02	0.71	12.7
	25	45.46	43.61	14.58	14.28	40.87	3.4	12
	45	137.45	133.48	41.59	41.15	103.29	10.08	10.2
	60	222.18	218.31	64.43	64.17	209.22	17.22	12.1
	75	331.06	327.28	91.64	90.67	378.83	26.32	14.4
	90	472.72	472.15	123.66	124.08	389.77	31.68	12.3
300	10	0	0	0	0	10.95	0.09	121.7
	25	62.08	59.78	19.58	18.96	56.21	6.11	9.2
	45	195.3	187	56.87	55.34	441.84	28.3	15.6
	60	329.04	321.03	89.89	88.58	944.5	49.68	19
	75	486.7	478.78	129.64	128.42	1836.57	77.93	23.6
	90	715.58	712.23	179.63	179.03	1907.46	96.02	19.9
400	10	0	0	0	0	23.71	0.1	237.1
	25	71.76	67.98	23.61	23.1	156.89	12.57	12.5
	45	236.18	224.29	70.09	68.18	1350.42	59.5	22.7
	60	407.45	395.12	115.14	112.3	2945.22	107.13	27.5
	75	614.4	601.47	167.26	164.99	5389.5	171.44	31.4
	90	910.95	903.06	234.89	233.26	5542.97	214.41	25.8
500	10	0	0	0	0	57.74	0.1	577.4
	25	77.1	71.26	27.4	26.63	368	22	16.6
	45	269.04	254.44	82.82	79.86	3207	106	30.1
	60	476	460	139	135	6688	195	34.2
	75	731	717	205	202	12665	318	39.8
	90	1087	1076	289	287	13360	403	33.1

Table 3 considers the graphs have sizes (number of vertices, n) ranging from 100 to 500, with the edge density (α) varying from 10% to 90%. The performance data for each entry is the mean of 125 runs, which encompasses five different color class density values (β) across five unique graphs, with each graph configuration being executed five times. Similarly, Table 4 examines graphs varying in size from 100 to 500 vertices, but focuses on variations in color class densities (β) from 0.04 to 0.2. This setup aims to evaluate the efficiency and effectiveness of the algorithms in assigning colors from a predefined set to the graph vertices. The number of colors in this set is determined by the formula $n \times \beta$. A higher β indicates more available colors, potentially making the coloring problem less constrained. Each row in Table 4 represents the average outcome of 150 runs which across six different edge densities (α), where for each edge

TABLE 4. Comparison of the algorithms with respect to fitness values and number of uncolored vertices for 5 different color class density values.

n	β	Fitness Values		# of Uncolored Vertices		Time(s)		Speed up
		InCEA	BitEA	InCEA	BitEA	InCEA	BitEA	
100	0.04	255.34	255.3	60.07	60.13	8.32	2.41	3.4
	0.08	144.6	144.1	38.29	38.23	14.37	2.97	4.8
	0.1	111.1	110.7	31.22	31.23	15.97	2.88	5.5
	0.15	55.6	55.3	17.85	17.93	23.81	2.77	8.6
	0.2	27	27	10.05	10.03	28.62	2.17	13.2
200	0.04	453.8	450.8	111.33	111	69.62	14.01	5
	0.08	247.7	244.7	69.06	68.85	151.47	17.34	8.7
	0.1	184.7	181.8	55.43	55.02	180.76	17.14	10.5
	0.15	84.9	82.9	29.58	29.28	256.63	14.93	17.2
	0.2	36.9	36.1	15.03	15.01	284.03	11.09	25.6
300	0.04	685	678.8	162.29	161.24	251.94	39.51	6.4
	0.08	367.1	361.1	98.11	97.25	572.66	48.67	11.8
	0.1	269	263	77.11	76.08	815.4	50.6	16.1
	0.15	119.4	115.3	39.62	38.66	1249.99	43.76	28.6
	0.2	50.1	47.4	19.21	18.71	1441.29	32.57	44.2
400	0.04	873.4	864.1	211.83	210.87	679.32	85.95	7.9
	0.08	460.4	449.2	127.15	125.41	1671.71	106.69	15.7
	0.1	332.7	322.8	98.5	96.52	2425.04	110.77	21.9
	0.15	143	137	48.62	47.03	3785.09	95.89	39.5
	0.2	57.7	53.4	23.05	21.71	4279.43	71.58	59.8
500	0.04	372	360	103.2	101	4915	131	37.5
	0.08	419	413	117	114	4957	145	34.2
	0.10	462	457	130	126	5020	161	31.2
	0.15	492.5	483	138.4	135	5161	181	28.5
	0.20	502	495	142.8	140	5408	201	26.9

density, five different graphs were generated and each was run five times.

Across different configurations, the fitness values between InCEA and BitEA are closely matched. However, BitEA achieves slightly better results in some cases through enhancement in its ability to search back for previous color classes during the crossover operation, termed the 'Improved Search Back' (ISB) operation. While ISB operator is decreasing the fitness value, it may sometimes leave marginally more vertices uncolored, although this difference is minimal and unlikely to impact the overall solution quality significantly.

In terms of execution time, BitEA displays a remarkable speed advantage across all scenarios. The speedup factor indicates that BitEA is consistently faster than InCEA, with the performance gap widening as the size of the graph, denoted by n, increases. For smaller graphs (with $n = 100$), the speedup of BitEA ranges from 5 to 8.5 times and from 3.4 to 13.2 faster than InCEA in Table 3 and in Table 4, respectively. This improvement is even more pronounced for larger graphs: for instance, with $n = 300$, BitEA achieves a speedup factor as high as 121.7 at a 10% edge density in Table 3, which gradually decreases as the edge density increases, stabilizing at a still impressive 19.9 at a 90% edge density. The trend continues with $n = 400$ and $n = 500$, where the highest speedups are 237.1 and 577.4, respectively, at the lowest edge densities. Conversely, in Table 4, the speedup significantly increases with the color class density β for all values of n.

3) PERFORMANCE COMPARISON TESTS ON DIMACS BENCHMARKS

In this section, we present an evaluation of our proposed algorithm BitEA against InCEA on DIMACS benchmark

TABLE 5. Comparative analysis of the InCEA and BitEA algorithms across various GEOM and DSJC graph instances.

Graph	|V|	|E|	InCEA		BitEA		Speed
			k	Time(s)	k	Time(s)	up
GEOM20	20	20	5	0.1	5	0.1	1
GEOM20a	20	37	5	0.1	5	0.1	1
GEOM20b	20	32	3	0.1	3	0.1	1
GEOM30	30	50	6	1	6	0.2	5
GEOM30a	30	81	6	1	6	0.3	3
GEOM30b	30	81	5	1	5	0.2	5
GEOM40	40	78	6	2	6	0.3	7
GEOM40a	40	146	7	4	7	0.6	7
GEOM40b	40	157	7	4	7	0.5	8
GEOM50	50	127	6	4	6	0.5	8
GEOM50a	50	238	9	9	9	0.8	11
GEOM50b	50	249	8	8	8	0.8	10
GEOM60	60	185	6	6	6	0.9	7
GEOM60a	60	339	10	16	10	1.2	13
GEOM60b	60	366	9	14	9	1.3	11
GEOM70	70	267	8	13	8	1.1	12
GEOM70a	70	459	11	26	11	2	13
GEOM70b	70	488	10	23	10	2.1	11
GEOM80	80	349	8	19	8	1.6	12
GEOM80a	80	612	12	41	12	2.4	17
GEOM80b	80	663	12	38	12	2.6	15
GEOM90	90	441	8	22	8	2.6	8
GEOM90a	**90**	**789**	**13**	**61**	**12**	**3.9**	**16**
GEOM90b	90	860	15	74	15	3.2	23
GEOM100	100	547	9	34	9	3	11
GEOM100a	100	992	14	86	14	4.7	18
GEOM100b	100	1050	15	96	15	4.1	23
GEOM110	110	638	9	42	9	3.4	12
GEOM110a	**110**	**1207**	**15**	**118**	**14**	**5.8**	**20**
GEOM110b	110	1256	16	128	16	5	26
GEOM120	120	773	11	70	11	3.7	19
GEOM120a	**120**	**1434**	**17**	**174**	**16**	**6.5**	**27**
GEOM120b	**120**	**1491**	**17**	**171**	**16**	**6.8**	**25**
DSJC125.1g	125	736	6	27	6	3.4	8
DSJC125.1gb	125	736	6	28	6	3.3	8
DJSC125.5g	**125**	**3891**	**20**	**277**	**19**	**10.5**	**26**
DJSC125.5gb	**125**	**3891**	**20**	**280**	**19**	**10.4**	**27**
DSJC125.9g	125	6961	46	1315	46	21.8	60
DSJC125.9gb	125	6961	45	1233	45	20.9	59

TABLE 6. Performance evaluation of the InCEA and BitEA algorithms on R, myciel, and queen instances from the DIMACS benchmarks.

Graph	V	E	InCEA		BitEA		Speed
			k	Time(s)	k	Time(s)	up
R50_1g	50	108	3	1	3	0.4	2.5
R50_1gb	50	108	3	1	3	0.4	2.5
R50_5g	50	612	10	12	10	1.4	8.6
R50_5gb	50	612	10	13	10	1.5	8.7
R50_9g	50	1092	21	47	21	1.8	26
R50_9gb	50	1092	21	48	21	2.2	21.8
R75_1g	70	251	4	5	4	1	5
R75_1gb	70	251	4	5	4	1	5
R75_5g	75	1407	13	49	13	3	16.3
R75_5gb	75	1407	13	48	13	3.2	15
R75_9g	75	2513	33	253	33	6.2	40.8
R75_9gb	75	2513	33	253	33	5.9	42.8
R100_1g	100	509	5	15	5	2.3	6.5
R100_1gb	100	509	5	14	5	2.2	6.4
R100_5g	100	2456	15	109	15	6.2	17.5
R100_5gb	100	2456	15	105	15	6.5	16
R100_9g	100	4438	36	510	36	11.9	42.8
R100_9gb	100	4438	36	527	36	11.3	46.6
myciel5g	47	236	6	4	6	0.4	10
myciel5gb	47	236	6	4	6	0.4	10
myciel6g	95	755	7	18	7	1	18
myciel6gb	95	755	7	18	7	0.8	22.5
myciel7g	191	2360	8	92	8	2.5	36.8
myciel7gb	191	2360	8	92	8	2.6	35.4
queen8_8g	64	728	9	19	9	2.2	8.6
queen8_8gb	64	728	9	19	9	2.2	8.6
queen9_9g	**81**	**1056**	**11**	**38**	**10**	**3.3**	11.5
queen9_9gb	81	1056	11	37	11	3.3	11.2
queen10_10g	**100**	**1470**	**13**	**76**	**12**	**5.2**	14.6
queen10_10gb	**100**	**1470**	**13**	**74**	**12**	**5.4**	13.7
queen11_11g	121	1980	14	124	14	7.2	17.2
queen11_11gb	121	1980	14	127	14	7.5	16.9
queen12_12g	144	2596	15	201	15	10.3	19.5
queen12_12gb	144	2596	15	201	15	10.9	18.4

graphs in terms of computational efficiency and solution quality, denoted as the execution time measured in seconds *Time(s)* and the chromatic number *k*, respectively. The performance results for each graph represent the best outcome of 20 runs for each algorithm, where each run is conducted with 30000 iterations. Our findings are summarized in Table 5 and Table 6, where $|V|$ and $|E|$ represent the number of vertices and edges, respectively, for each graph. The performance is also quantified by the speed-up factor, which compares the time efficiency of BitEA against InCEA, demonstrating the performance gains of our approach.

In Table 5, for smaller graph instances such as GEOM20 through GEOM40, the speedup factor remains around 1 to 8, indicating that while BitEA has a similar performance to InCEA, it does not yet fully capitalize on its potential speed improvements. However, as the size of the graphs increases, we observe significant efficiency gains with BitEA. For instance, in larger and more complex graphs such as GEOM90a, GEOM120a, and GEOM120b, BitEA achieves speedup factors ranging from 16 to 27. The standout cases in this table are the DSJC graph with varying edge density values, where BitEA achieves substantial speedup factors from 8 to 60.

In Table 6, the R graphs demonstrate speedup factors ranging from 2.5 to 26, showcasing the efficiency gains of BitEA over InCEA. The myciel graphs also illustrate the time efficiency of BitEA with speedup factors reaching up to 36.8. In the queen graphs, while increasing number of vertices, BitEA maintains a consistent speedup, highlighting its scalability. Similarly, graphs like queen10_10g and queen10_10gb show that BitEA can deliver a performance boost of over 13 as compared to InCEA.

In terms of the chromatic number, represented by k, BitEA outperforms InCEA by requiring one fewer color class for nine graphs: GEOM90a, GEOM110a, GEOM120a, GEOM120b, DJSC125.5g, DSJC125.5gb, queen9_9g, queen10_10g, and queen10_10gb. This enhanced performance is highlighted in Table 5 and Table 6. The observed enhancements in solution quality are a result of the ISB operator in BitEA, which strategically swaps vertices between the pool and the existing color classes in the offspring. This is performed at the conclusion of each BitCX color combination process to minimize the total weights of the pool.

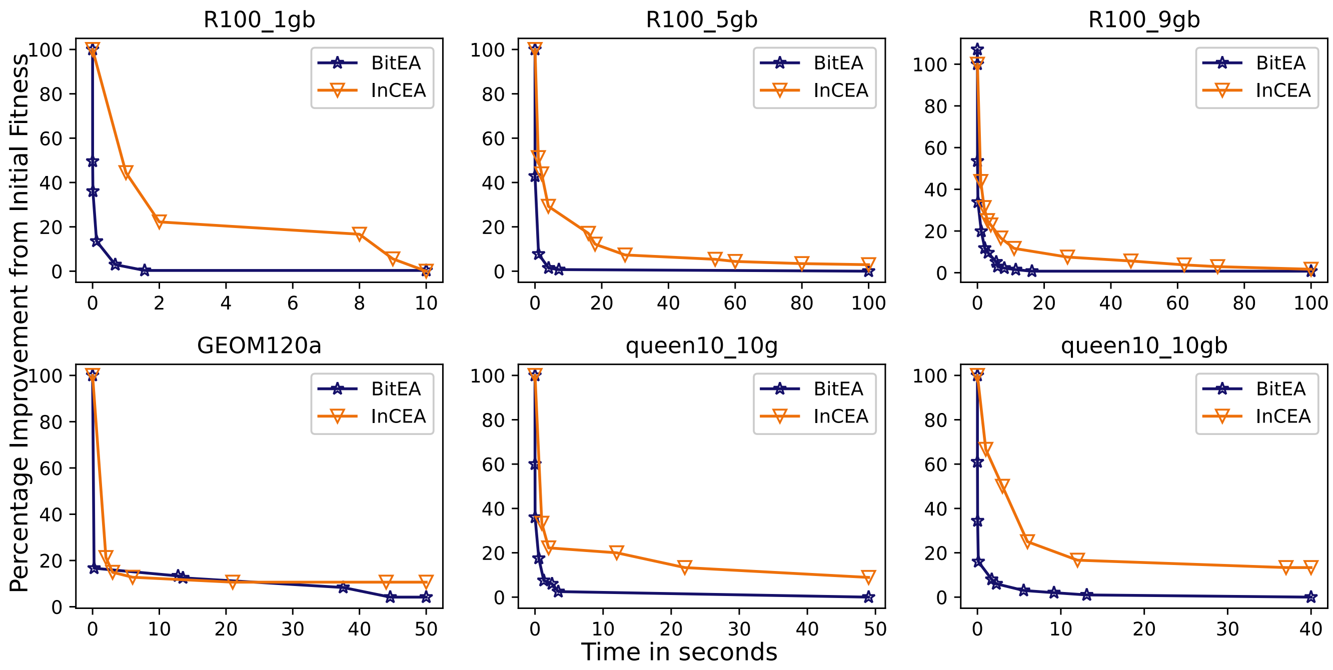


FIGURE 11. Performance evaluation of BitEA for time-wise fitness enhancement on DIMACS graphs (R100_1gb.col, R100_5gb.col, R100_9gb.col, GEOM120a.col, queen10_10g.col and queen10_10gb.col) using 5, 15, 36, 16, 12 and 12 color classes, respectively, within a time limit of 100 seconds.

VI. CONCLUSION AND FUTURE WORK

This study introduces BitVertex Evolutionary Algorithm (BitEA), a novel approach for the register allocation problem, building a bit-based representation and bitwise operations to significantly enhance computational efficiency and solution quality. Our extensive experimental evaluations with 150 InCEA and 73 DIMACS benchmarks, demonstrate BitEA has remarkable performance improvements when compared to Integrated Crossover Based Evolutionary Algorithm (InCEA). BitEA demonstrates substantial speedup factors, ranging from 3 to 577.4 in specific configurations, especially in larger graph instances. BitVertex representation and the integration of improved search back (ISB) and local search (Bit-SWAP) strategies do not only optimize execution time but also improve solution quality by achieving lower chromatic numbers. BitEA outperforms InCEA in all InCEA benchmarks and in 9 DIMACS benchmarks, including queen9_9g, queen10_10g, queen10_10gb, GEOM90a, GEOM110a, GEOM120a, GEOM120b, DSJC125.5g, and DSJC125.5gb, in terms of the number of colors used.

Our findings highlight the potential of bit-based solution representation in reducing execution time and memory usage, while also providing a path for future research directions. These include exploring applicability of BitEA to other related combinatorial optimization problems such as scheduling problems [51], communication problems [52] and refining the algorithmic components of BitEA for dynamic problem settings. Furthermore, the compactness offered by BitVertex representation and the computational efficiency of bitwise operations to handle complex datasets or solution spaces make BitEA a promising approach for applications in various computational fields, including multithreading, FPGA and GPU architectures especially while serving large-scale graphs. BitEA can also inspire new solution approaches to the register allocation problem using various algorithms such as approximation algorithms.

REFERENCES

[1] R. Sethi, "Complete register allocation problems," in *Proc. 5th Annu. ACM Symp. Theory Comput.*, 1973, pp. 182–195.

[2] P. K. Krause, "The complexity of register allocation," *Discrete Appl. Math.*, vol. 168, pp. 51–59, May 2014.

[3] R. W. Quong and S.-C. Chen, "Register allocation via weighted graph coloring (technical summary)," Purdue Uni. Lib., West Lafayette, IN, USA, ECE Tech. Rep. TR-EE 93-23 (232), Jun. 1993.

[4] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability*, vol. 174. San Francisco, CA, USA: Freeman, 1979.

[6] B. Escoffier, J. Monnot, and V. T. Paschos, "Weighted coloring: Further complexity and approximability results," in *Proc. Italian Conf. Theor. Comput. Sci.*, Siena, Italy. Berlin, Germany: Springer, 2005, pp. 205–214.

[7] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 98–101, Jun. 1982, doi: 10.1145/872726.806984.

[8] H. Bouziri, K. Mellouli, and E.-G. Talbi, "The K-coloring fitness landscape," *J. Combinat. Optim.*, vol. 21, no. 3, pp. 306–329, Apr. 2011.

[9] R. M. R. Lewis, *A Guide to Graph Colouring*, vol. 7. Berlin, Germany: Springer, 2015, doi: 10.1007/978-3-319-25730-3.

[10] B. Boz and G. Süngü, "Integrated crossover based evolutionary algorithm for coloring vertex-weighted graphs," *IEEE Access*, vol. 8, pp. 126743–126759, 2020.

[11] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, May 1994.

[12] L. George and A. W. Appel, "Iterated register coalescing," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 300–324, May 1996.

[13] F. M. Q. Pereira, "A survey on register allocation," Dept. Comput. Sci., Univ. California, Los Angeles, CA, USA, Tech. Rep., 2008.

[14] B. Kernighan and D. M. Ritchie, *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice-Hall, 2017.

[15] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning," *Oper. Res.*, vol. 39, no. 3, pp. 378–406, Jun. 1991.

[16] S. Gualandi and F. Malucelli, "Exact solution of graph coloring problems via constraint programming and column generation," *INFORMS J. Comput.*, vol. 24, no. 1, pp. 81–100, Feb. 2012.

[17] E. Malaguti, M. Monaci, and P. Toth, "An exact approach for the vertex coloring problem," *Discrete Optim.*, vol. 8, no. 2, pp. 174–190, May 2011, doi: 10.1016/j.disopt.2010.07.005.

[18] F. Furini and E. Malaguti, "Exact weighted vertex coloring via branch-and-price," *Discrete Optim.*, vol. 9, no. 2, pp. 130–136, May 2012.

[19] D. Cornaz, F. Furini, and E. Malaguti, "Solving vertex coloring problems as maximum weight stable set problems," *Discrete Appl. Math.*, vol. 217, pp. 151–162, Jan. 2017.

[20] E. Malaguti, M. Monaci, and P. Toth, "Models and heuristic algorithms for a weighted vertex coloring problem," *J. Heuristics*, vol. 15, no. 5, pp. 503–526, Oct. 2009.

[21] S. Shamizi and S. Lotfi, "Register allocation via graph coloring using an evolutionary algorithm," in *Proc. SEMCCO*, Visakhapatnam, India, 2011, pp. 1–8.

[22] P. Galinier, A. Hertz, and N. Zufferey, "An adaptive memory algorithm for the k-colouring problem," *Discrete Appl. Math.*, vol. 156, no. 2, pp. 267–279, 2008.

[23] G. Sungu and B. Boz, "An evolutionary algorithm for weighted graph coloring problem," in *Proc. Companion Publication Annu. Conf. Genetic Evol. Comput.*, Madrid, Spain, Jul. 2015, pp. 1233–1236.

[24] H. R. Topcuoglu, B. Demiroz, and M. Kandemir, "Solving the register allocation problem for embedded systems using a hybrid evolutionary algorithm," *IEEE Trans. Evol. Comput.*, vol. 11, no. 5, pp. 620–634, Oct. 2007, doi: 10.1109/TEVC.2007.892766.

[25] J. Wu, Z. Chang, L. Yuan, Y. Hou, and M. Gong, "A memetic algorithm for resource allocation problem based on node-weighted graphs [application notes]," *IEEE Comput. Intell. Mag.*, vol. 9, no. 2, pp. 58–69, May 2014, doi: 10.1109/MCI.2014.2307231.

[26] D. Das, S. A. Ahmad, and V. Kumar, "Deep learning-based approximate graph-coloring algorithm for register allocation," in *Proc. IEEE/ACM 6th Workshop LLVM Compiler Infrastructure HPC (LLVM-HPC) Workshop Hierarchical Parallelism Exascale Comput. (HiPar)*, Nov. 2020, pp. 23–32.

[27] S. VenkataKeerthy, S. Jain, A. Kundu, R. Aggarwal, A. Cohen, and R. Upadrasta, "RL4ReAl: Reinforcement learning for register allocation," in *Proc. 32nd ACM SIGPLAN Int. Conf. Compiler Construction*, Feb. 2023, pp. 133–144.

[28] G. S. Terci, "A learning-based coloring algorithm for register allocation problem," in *Proc. 31st Signal Process. Commun. Appl. Conf. (SIU)*, Jul. 2023, pp. 1–4.

[29] C. N. Lintzmayer, M. H. Mulati, and A. F. D. Silva, "Register allocation with graph coloring by ant colony optimization," in *Proc. 30th Int. Conf. Chilean Comput. Sci. Soc.*, Nov. 2011, pp. 247–255.

[30] B. Nogueira, R. G. S. Pinheiro, and A. Subramanian, "A hybrid iterated local search heuristic for the maximum weight independent set problem," *Optim. Lett.*, vol. 12, no. 3, pp. 567–583, May 2018.

[31] B. Nogueira, E. Tavares, and P. Maciel, "Iterated local search with Tabu search for the weighted vertex coloring problem," *Comput. Oper. Res.*, vol. 125, Jan. 2021, Art. no. 105087.

[32] P. Galinier and J. Hao, "Hybrid evolutionary algorithms for graph coloring," *J. Comb. Optim.*, vol. 3, no. 4, pp. 379–397, Dec. 1999, doi: 10.1023/A:1009823419804.

[33] D. C. Porumbel, J. K. Hao, and P. Kuntz, "Diversity control and multi-parent recombination for evolutionary graph coloring algorithms," in *Proc. Eur. Conf. Evol. Comput. Combinat. Optim.*, Tübingen, Germany, Apr. 2009, pp. 121–132.

[34] B. Betül and T. Yildiz, "Pool-based evolutionary algorithm for the bin packing problem," *Int. J. Adv. Eng. Pure Sci.*, vol. 33, no. 3, pp. 406–414, 2021.

[35] S. Korkmaz, "Solving graph coloring problem by using an evolutionary algorithm," M.S. thesis, Dept. Comput. Eng., Marmara Univ., Istanbul, 2020.

[36] G. Rokos, G. Gorman, and P. H. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures," in *Proc. 21st Int. Conf. Parallel Distrib. Comput.*, Vienna, Austria, 2015, pp. 414–425.

[37] A. Borione, L. Cardone, A. Calabrese, and S. Quer, "An experimental evaluation of graph coloring heuristics on multi- and many-core architectures," *IEEE Access*, vol. 11, pp. 125226–125243, 2023.

[38] M. Osama, M. Truong, C. Yang, A. Buluç, and J. Owens, "Graph coloring on the GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2019, pp. 231–240.

[39] O. Goudet, C. Grelier, and J.-K. Hao, "A deep learning guided memetic framework for graph coloring problems," *Knowl.-Based Syst.*, vol. 258, Dec. 2022, Art. no. 109986.

[40] Z. Zheng, X. Shi, L. He, H. Jin, S. Wei, H. Dai, and X. Peng, "Feluca: A two-stage graph coloring algorithm with color-centric paradigm on GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 160–173, Jan. 2021.

[41] R. Murooka, Y. Ito, and K. Nakano, "Accelerating ant colony optimization for the vertex coloring problem on the GPU," in *Proc. 4th Int. Symp. Comput. Netw. (CANDAR)*, Nov. 2016, pp. 469–475.

[42] K. Zhang, M. Qiu, L. Li, and X. Liu, "Accelerating genetic algorithm for solving graph coloring problem based on CUDA architecture," in *Proc. 9th Int. Conf.*, Wuhan, China. Berlin, Germany: Springer, Oct. 2014, pp. 578–584.

[43] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for GPU-based computing," in *Proc. ACM SOSP Workshop Power Aware Comput. Syst. (HotPower)*, vol. 1, Oct. 2009, pp. 1–5.

[44] H. Fan, M. Li, J. Wu, W. Lu, X. Li, and G. Yan, "BitColor: Accelerating large-scale graph coloring on FPGA with parallel bit-wise engines," in *Proc. 52nd Int. Conf. Parallel Process.*, Aug. 2023, pp. 492–502.

[45] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Found. Trends Electron. Design Autom.*, vol. 2, no. 2, pp. 135–253, 2008.

[46] D. Brélaz, "New methods to color the vertices of a graph," *Commun. ACM*, vol. 22, no. 4, pp. 251–256, Apr. 1979.

[47] P. C. B. Lam, W. Lin, G. Gu, and Z. Song, "Circular chromatic number and a generalization of the construction of mycielski," *J. Combinat. Theory, Ser. B*, vol. 89, no. 2, pp. 195–205, Nov. 2003, doi: 10.1016/s0095-8956(03)00070-4.

[48] D. S. Johnson and M. A. Trick, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. Providence, RI, USA: American Mathematical Society, 1993.

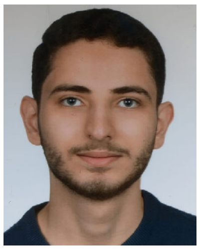
[49] M. B. Dillencourt, D. Eppstein, and M. T. Goodrich, "Choosing colors for geometric graphs via color space embeddings," in *Proc. Int. Symp. Graph Drawing*, Karlsruhe, Germany. Berlin, Germany: Springer, Sep. 2006, pp. 294–305.

[50] M. Vasquez, "New results on the Queens_n^2 graph coloring problem," *J. Heuristics*, vol. 10, no. 4, pp. 407–413, Jul. 2004, doi: 10.1023/b:heur.0000034713.28244.e1.

[51] P. B. Myszkowski, "Solving scheduling problems by evolutionary algorithms for graph coloring problem," in *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, vol. 145. Berlin, Germany: Springer, 2008, p. 167.

[52] M. Miri, K. Mohamedpour, Y. Darmani, M. Sarkar, and R. L. Tummala, "An efficient resource allocation algorithm based on vertex coloring to mitigate interference among coexisting WBANs," *Comput. Netw.*, vol. 151, pp. 132–146, Mar. 2019, doi: 10.1016/j.comnet.2019.01.014.

GIZEM SUNGU TERCI was born in Istanbul, Türkiye, in 1993. She received the B.S. and M.S. degrees in computer engineering from Marmara University, Türkiye, in 2015 and 2018, respectively. She is currently pursuing the Ph.D. degree in computer engineering with Gebze Technical University, Türkiye. Since 2017, she has been a Research Assistant with the Institute of Information Technologies, Gebze Technical University. Her research interests include optimization problems, such as graph coloring and path planning, evolutionary algorithms, and performance acceleration. She was a recipient of the ACM-Woman Scholarship to present her paper at the Genetic and Evolutionary Computation Conference (GECCO), Madrid, Spain, in 2015.



ENES ABDULHALIK was born in Abu Dhabi, United Arab Emirates, in 2001. He is currently pursuing the B.S. degree with the Computer Engineering Department, Gebze Technical University, Türkiye, in 2024. His research interests include algorithm acceleration using fast CPU implementations and FPGA, including graph coloring algorithms, evolutionary algorithms, and image processing algorithms. He gained scholarship through TUBITAK's 1002-B Program in a project around an FPGA implementation of an evolutionary algorithm for graph coloring problems.

ALP ARSLAN BAYRAKCI was born in Istanbul, Türkiye, in 1982. He received the B.S. degree in electrical and electronics engineering from Middle East Technical University, Ankara, Türkiye, in 2004, and the Ph.D. degree in computer engineering from Koc University, Istanbul, in 2010. He is currently an Assistant Professor with the Department of Computer Engineering, Gebze Technical University. His current research interests include FPGA-based hardware acceleration, statistical timing analysis of circuits, hardware security, and computer-aided integrated circuit design methodologies.

BETUL BOZ was born in Ankara, Türkiye, in 1980. She received the B.Sc. and M.Sc. degrees in computer engineering from Marmara University, Istanbul, Türkiye, in 2002 and 2004, respectively, and the Ph.D. degree in computer engineering from Boğaziçi University, Istanbul, in 2011. She is currently an Assistant Professor with the Computer Engineering Department, Marmara University. Her research interests include design of efficient operators and techniques for real-world problems using evolutionary algorithms, computational and artificial intelligence, parallel processing, and computer architecture.