

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL

**BITVERTEX EVOLUTIONARY ALGORITHM FOR
ACCELERATING GRAPH COLORING IN REGISTER
ALLOCATION**

GİZEM SÜNGÜ TERCİ

A THESIS OF DOCTORATE
DEPARTMENT OF COMPUTER ENGINEERING

ADVISOR: ASST. PROF. ALP ARSLAN BAYRAKÇI

OCTOBER 2024

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL

**BITVERTEX EVOLUTIONARY ALGORITHM FOR
ACCELERATING GRAPH COLORING IN REGISTER
ALLOCATION**

GİZEM SÜNGÜ TERCİ

A THESIS OF DOCTORATE
DEPARTMENT OF COMPUTER ENGINEERING

ADVISOR: ASST. PROF. ALP ARSLAN BAYRAKÇI

Co-ADVISOR: ASST. PROF. BETÜL BOZ

OCTOBER 2024

T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

KAYIT TAHSİSİNDE ÇİZGE RENKLENDİRMEYİ
HIZLANDIRMAK İÇİN BITVERTEX EVRİMSEL
ALGORİTMASI

GİZEM SÜNGÜ TERCİ

DOKTORA TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMAN: DR. ÖĞR. ÜYESİ ALP ARSLAN BAYRAKÇI
2. DANIŞMAN: DR. ÖĞR. ÜYESİ BETÜL BOZ

EKİM 2024

DOCTORATE JURY APPROVAL FORM

A thesis submitted by Gizem Sng Terci, defended on 02/10/2024 before the jury formed with the 30/09/2024 date and 2024/55 numbered decision of the GTU Graduate Administration Board, has been accepted as a DOCTOR of PHILOSOPHY thesis in the Department of Computer Engineering.

JURY

MEMBER (THESIS ADVISOR)	: Asst. Prof. Alp Arslan BAYRAKI
MEMBER	: Prof. Haluk TOPCUOLU
MEMBER	: Prof. Erkan ZERGEROLU
MEMBER	: Assoc. Prof. Mehmet GKTRK
MEMBER	: Asst. Prof. Fatma CORUT ERGN

APPROVAL

Gebze Technical University Graduate Administration Board/...../..... date and
...../..... numbered decision.

SIGNATURE/SEAL

ABSTRACT

Register allocation is an essential aspect of compiler design aimed at enhancing the execution speed of programs, as CPU registers offer significantly faster data access compared to memory locations. The challenge of optimally assigning program variables to a limited number of CPU registers is a known NP-complete problem, prompting the need for innovative approaches that can deliver efficient solutions within practical timeframes. Traditional methods often prove computationally infeasible on current CPU architectures, especially when dealing with large-scale problem instances. This thesis introduces the BitVertex Evolutionary Algorithm (BitEA), a novel approach grounded in bitwise operations and bit-based solutions, designed to accelerate computational performance in register allocation by leveraging efficient encoding mechanisms and parallel processing capabilities.

BitEA introduces several key contributions to the field. First, it utilizes the BitVertex representation, an innovative bit-based structure that enables compact graph encoding, reducing memory consumption and enhancing the computational efficiency of evolutionary operations. Second, BitEA features a specialized crossover operator, the BitVertex Crossover Operator (BitCX), which integrates bitwise operations for improved offspring generation and reduced conflicts. Third, an Improved Search Back (ISB) operator is embedded within the crossover process to refine solutions further by intelligently handling conflicting vertices. Additionally, BitEA incorporates an enhanced local search strategy designed to optimize the placement of vertices and minimize the chromatic number.

The proposed BitEA framework is supported by a scalable architecture that extends the BitVertex representation to accommodate graphs with significantly more vertices, ensuring adaptability for larger problem sizes. Comprehensive experimental evaluations conducted on DIMACS and InCEA benchmark datasets highlight the superior performance of BitEA, showcasing an average computational speedup of up to 60 times over existing algorithms and demonstrating lower chromatic numbers on 9 DIMACS instances compared to contemporary methods.

This thesis not only introduces BitEA as a highly efficient method for solving the vertex-weighted k -coloring problem in register allocation but also underscores its versatility through detailed analysis and comparative studies. The findings indicate that BitEA achieves a balance between computational efficiency and solution quality, outperforming both heuristic and evolutionary methods in terms of speed and optimality. The research concludes by identifying potential applications of BitEA in other combinatorial optimization problems and suggesting future enhancements to further improve its robustness and adaptability in real-time and large-scale optimization scenarios.

Keywords: Evolutionary Optimization, Register Allocation, Graph Coloring Problem, Compiler Design, Bitwise Operations, Performance Acceleration

ÖZET

Kayıt tahsisi, derleyici tasarımında, programların yürütme hızını artırmayı amaçlayan önemli bir bileşendir; zira CPU kayıtları, bellek konumlarına kıyasla çok daha hızlı veri erişimi sağlar. Program değişkenlerinin sınırlı sayıdaki CPU kayıtlarına optimal bir şekilde atanması, pratik sürelerde verimli çözümler sunabilecek yenilikçi yaklaşımları gerektiren, bilinen NP-tam bir problemidir. Özellikle büyük ölçekli problem örnekleri söz konusu olduğunda, geleneksel yöntemler mevcut CPU mimarilerinde hesaplama açısından uygulanamaz hale gelebilmektedir. Bu tez, kayıt tahsisini hızlandırmak amacıyla bit düzeyindeki işlemler ve bit tabanlı çözümler üzerine kurulu olan BitVertex Evrimsel Algoritması'nı (BitEA) tanıtmaktadır. BitEA, verimli kodlama mekanizmalarından ve paralel işlem yeteneklerinden yararlanarak hesaplama performansını artırmak üzere tasarlanmıştır.

BitEA, alana bir dizi önemli katkı sunmaktadır. İlk olarak, BitEA, algoritmaların ve problem örneklerinin bellek tüketimini azaltarak hesaplama verimliliğini artıran kompakt bir çizge kodlamasına olanak tanıyan yenilikçi bir bit tabanlı yapı olan BitVertex gösterimini kullanmaktadır. İkinci olarak, BitEA, bit düzeyindeki işlemleri iyileştirilmiş yavru üretimi ve daha az çatışma için entegre eden özel bir çaprazlama operatörü olan BitVertex Çaprazlama Operatörü (BitCX) sunmaktadır. Üçüncü olarak, çaprazlama sürecine, çatışma yaratan düğümleri akıllıca ele alarak çözümleri daha da iyileştiren Geliştirilmiş Geri Arama (ISB) operatörü eklenmiştir. Ayrıca, BitEA, düğümlerin yerleştirilmesini optimize etmek ve kromatik sayıyı en aza indirmek için tasarlanmış gelişmiş bir yerel arama stratejisini içermektedir.

Önerilen BitEA çerçevesi, daha büyük çizge boyutlarına uyum sağlayarak BitVertex gösterimini genişleten ölçeklenebilir bir mimari ile desteklenmiştir. DIMACS ve InCEA kıyaslama veri kümelerinde gerçekleştirilen kapsamlı deneysel değerlendirmeler, BitEA'nın üstün performansını ortaya koymuş; mevcut algoritmalara kıyasla 60 kata kadar ortalama hesaplama hızlanması sergilediğini ve 9 DIMACS örneğinde çağdaş yöntemlere kıyasla daha düşük kromatik sayılar elde ettiğini göstermiştir.

Bu tez, BitEA'yı kayıt tahsisinde tepe noktası ağırlıklı k -renklendirme problemini çözmek için son derece verimli bir yöntem olarak tanıtırken, detaylı analizler ve karşılaştırmalı çalışmalarla çok yönlülüğünü de vurgulamaktadır. Bulgular, BitEA'nın hesaplama verimliliği ile çözüm kalitesi arasında bir denge sağladığını ve hem sezgisel hem de evrimsel yöntemleri hız ve optimalite açısından geride bıraktığını göstermektedir. Araştırma, BitEA'nın diğer kombinatoriyal optimizasyon problemlerinde potansiyel uygulamalarını ve gerçek zamanlı ve büyük ölçekli optimizasyon senaryolarında daha da geliştirilmesi için önerilerle sona ermektedir.

Anahtar Kelimeler: Evrimsel Optimizasyon, Kayıt Tahsisi, Çizge Renklendirme Problemi, Derleyici Tasarımı, Bit Tabanlı İşlemler, Performans Hızlandırma

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Asst. Prof. Dr. Alp Arslan Bayrakçi, for his invaluable guidance, continuous support, and insightful feedback throughout my doctoral studies. His mentorship has been crucial in shaping both my research and academic development. His encouragement and trust in my abilities have motivated me to push the boundaries of my knowledge and skills. I am deeply grateful for his dedication and patience.

I would also like to extend my heartfelt thanks to my co-advisor, Asst. Prof. Dr. Betül Boz, for her support and constructive advice during my research journey. Dr. Boz was also my advisor during my Bachelor's and Master's theses. Without her encouragement and belief in my potential, I would not have embarked on this academic journey. Her expertise and thoughtful suggestions have significantly contributed to the completion of this work. I am fortunate to have had her as a co-advisor, and I truly appreciate her time and effort in helping me refine my work.

I am deeply indebted to the members of my jury, Assoc. Prof. Dr. Mehmet Gök-türk, Asst. Prof. Dr. Fatma Corut Ergin, Prof. Dr. Erkan Zergeroğlu, and Prof. Dr. Haluk Topçuoğlu. Their valuable comments, critical perspectives, and suggestions have greatly enhanced the quality of this dissertation. I sincerely appreciate the time they devoted to evaluating my work and their role in shaping it through their insightful feedback.

I would like to acknowledge the support provided by TÜBİTAK, who funded this thesis under the 1002B Program with project number 124E044. Their financial support was crucial in the successful completion of my research, and I am sincerely grateful for the opportunity to pursue this project.

I would like to take this opportunity to thank my family, who have been my source of strength and encouragement throughout my academic journey. To my parents, Emrullah Süngü and Güleser Süngü, your unshakable belief in me, your patience, and your endless support have been my foundation. I would not have been able to achieve this without your love and guidance.

Lastly, my deepest appreciation goes to my beloved husband, Emre Terci, whose love, patience, and understanding have been my anchor during this challenging journey. Your endless support and belief in my abilities have been invaluable. Thank you for being by my side every step of the way, encouraging me, and helping me persevere through the most difficult moments.

TABLE OF CONTENTS

	<u>Page</u>
ABSTRACT	v
ÖZET	vi
ACKNOWLEDGEMENT	vii
TABLE of CONTENTS	viii
LIST of ABBREVIATIONS and ACRONYMS	x
LIST of FIGURES	xi
LIST of TABLES	xiii
1. INTRODUCTION	1
2. BACKGROUND AND RELATED WORK	4
2.1. Graph Coloring Problem	4
2.2. Register Allocation Problem	5
2.2.1. Graph Coloring Approach	8
2.2.2. Linear Scan Allocation Approach	14
2.2.3. Graph Coloring vs. Linear Scan Allocation	17
2.3. Heuristic Optimization in Register Allocation	20
2.3.1. Chaitin's Algorithm	20
2.3.2. Briggs' Algorithm	23
2.3.3. George and Appel's Algorithm	24
2.4. Evolutionary Optimization in Register Allocation	24
2.4.1. Hybrid Evolutionary Algorithm	26
2.4.2. Cost Oriented Memetic Algorithm	26
2.4.3. Integrated Crossover Based Evolutionary Algorithm	27
2.5. Performance Acceleration for Register Allocation	31
3. BITVERTEX REPRESENTATION FOR GRAPH COLORING	34
3.1. Problem Definition and Mathematical Modeling	34
3.2. Traditional Problem Representations	34
3.2.1. Adjacency Matrix Representation	35
3.2.2. Adjacency List Representation	35
3.3. BitVertex Representation	36
3.4. Scalable BitVertex Representation for Large Graphs	37
4. BITVERTEX EVOLUTIONARY ALGORITHM	39
4.1. Individual Representation and Generation	40
4.2. Crossover Operation	42

4.2.1.	BitVertex Crossover Operator	43
4.2.2.	Improved Search Back Operator	46
4.2.3.	Illustration of BitCX and ISB Operators	49
4.3.	Local Search Operation	56
4.4.	Fitness Function and Population Update	58
4.5.	Discussion	58
5.	EXPERIMENTAL STUDY	60
5.1.	Experimental Setup	60
5.2.	Experimental Results	61
5.2.1.	Performance Evaluation of BitEA and InCEA Varying on Time	61
5.2.2.	Performance Tests on Varying Generation Size	63
5.2.3.	Performance Comparison Tests on InCEA Benchmarks	64
5.2.4.	Performance Comparison Tests on DIMACS Benchmarks	69
6.	CONCLUSION	77
	REFERENCES	79
	BIOGRAPHY	88
	PUBLICATIONS WITHIN the SCOPE of THESIS STUDY	89

LIST of ABBREVIATIONS and ACRONYMS

$f(k)$	: Fitness function of k -coloring
$rw(n)$	: Number of read-write operations for variable n
$degree(n)$	: Number of edges a vertex has in the graph
$spillCost(n)$	: The cost function for spilling variable n .
InCEA	: Integrated Crossover-based Evolutionary Algorithm
BitEA	: BitVertex Evolutionary Algorithm
GCP	: Graph Coloring Problem
COMA	: Cost-Oriented Memetic Algorithm
HEA	: Hybrid Evolutionary Algorithm
FPGA	: Field Programmable Gate Arrays
PBEA	: Pool-based Evolutionary Algorithm
COPX	: Cost Oriented Cross Operator
CFPX	: Conflict Free Partition Crossover Operator
DIMACS	: the Center for Discrete Mathematics and Theoretical Computer Science
ISB	: Improved Search Back Operator
BitCX	: BitVertex Crossover Operator
InCX	: Integrated Crossover Operator

LIST OF FIGURES

	<u>Page</u>
Figure 2.1: Overview of the compilation process with the key phases.	5
Figure 2.2: Intermediate representation of the swap algorithm before register allocation using virtual registers.	7
Figure 2.3: The interference graph (Left) and the colored graph (Right) of Figure 2.2.	8
Figure 2.4: Intermediate representation of the swap algorithm after register allocation.	9
Figure 2.5: Final design of the swap algorithm.	10
Figure 2.6: Spilling v_0 variable from the interference graph of the swap algorithm.	13
Figure 2.7: Rewritten intermediate representation of the swap algorithm before register allocation spilling v_0 .	14
Figure 2.8: The interference graph (left) and the colored graph (right) of the swap algorithm after spilling v_0 .	15
Figure 2.9: Rewritten program of the swap algorithm spilling v_0 after register allocation.	17
Figure 2.10: Final program of the swap algorithm spilling v_0 after register allocation.	18
Figure 2.11: Live interval computation of the intermediate representation of the swap algorithm.	19
Figure 2.12: Sorted time intervals of virtual and machine registers, respectively.	19
Figure 2.13: Summary of the register allocation process throughout the compilation.	20
Figure 2.14: Phase flow of Chaitin's Algorithm	20
Figure 2.15: Example of Chaitin's Algorithm	21
Figure 2.16: Example of Chaitin's Algorithm when $k = 2$.	22
Figure 2.17: Number of uncolored vertices of HEA, COMA and InCEA on a set of DIMACS instances (R100_1gb, R100_5gb, R100_9gb, GEOM120a, queen10_10g and queen10_10gb) using 5, 15, 36, 16, 12 and 12 color classes, respectively. The results are adapted from [1].	30
Figure 2.18: Summary of the related studies and the position of this thesis.	31
Figure 3.1: BitVertex representation of the example graph [2]	36
Figure 3.2: 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.	38
Figure 4.1: Comparison of integer vertex and BitVertex individual representations	41
Figure 4.2: Combining two color classes from the parents with traditional array-based and bitwise operators (Operation 1 in Table 4.1).	43

Figure 4.3:	Example of calculation of the conflicts in the same color using Operation 3 of bitwise operations in Table 4.1.	45
Figure 4.4:	Example of calculation of the conflicts in the same color using Operation 3 of array-based operation in Table 4.1.	45
Figure 4.5:	Illustration of BitCX operation using two parent configurations and a weighted graph [2].	50
Figure 4.6:	Operation 2 using NOT, AND operations.	51
Figure 4.7:	Comparison of offsprings produced by different crossover operators for the example weighted graph in Figure 3.1.	52
Figure 4.8:	The example graph with some edge and vertex weight changes.	53
Figure 4.9:	BitCX applies on the same parent configurations considering the new graph in Figure 4.8.	54
Figure 4.10:	Solutions at the end of the crossover operations and the local search operations, respectively.	55
Figure 4.11:	Performance of Bit-SWAP technique on the offspring after the BitCX operation in Figure 4.9 using the graph in Figure 4.8.	56
Figure 4.12:	Comparison of solutions obtained by InCEA and BitEA represented in integer vertex-based format using the graph in Figure 4.8.	57
Figure 5.1:	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.	62
Figure 5.2:	Performance evaluation of BitEA for iteration-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 under 30000 iterations.	63

LIST OF TABLES

	<u>Page</u>
Table 2.1: Spill cost calculation for variables.	12
Table 2.2: Time steps, active Intervals, and allocations.	16
Table 2.3: Time steps, active intervals, and allocations after spilling v_0 .	16
Table 2.4: Computation time (in seconds) and speedup performances of InCEA for GEOM and DSJC graphs. The results are adapted from [1].	29
 Table 4.1: Time complexity comparison: array-based vs. bitwise operations in algorithms.	 44
Table 4.2: Comparison of Components among HEA, COMA, InCEA, and BitEA.	59
 Table 5.1: Parameters of InCEA benchmarks [1].	 61
Table 5.2: Comparative performance analysis of InCEA and BitEA across different graph sizes n and edge densities α .	65
Table 5.3: Comparison of the algorithms with respect to fitness values and number of uncolored vertices for 5 different color class density values.	67
Table 5.4: Computation time distribution and iterations of InCEA and BitEA in 500 seconds for 6 DIMACS instances.	69
Table 5.5: Comparative analysis of the InCEA and BitEA algorithms for GEOM and DSJC graphs with their computation time.	70
Table 5.6: Performance evaluation of the InCEA and BitEA algorithms on R, myciel, and queen graphs from the DIMACS benchmarks.	72
Table 5.7: Comprehensive performance evaluation of BitEA on hard DIMACS benchmark instances.	74

1. INTRODUCTION

The register allocation problem is a crucial aspect of compiler optimization [3], 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 [4]. Strategies like graph coloring [5] and linear scan allocation [6] are employed to navigate these challenges, balancing performance optimization with resource constraints. Despite the complexity, the graph coloring strategy which is NP-hard [7, 8], 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 [9]. 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), 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 in graph coloring [10]. 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 thesis focuses on solving the register allocation problem by tackling the k -coloring problem in weighted-vertex graphs.

Algorithms in the literature attempt to assign colors to all vertices without spilling, aiming to minimize conflicts [11]. Although these algorithms can achieve optimal solutions

for specific graph models, their high computational cost renders them impractical for real-time systems, particularly on CPU architectures. This thesis seeks to address this bottleneck by designing an evolutionary algorithm that uses bitwise representations and operations. The hypothesis of this thesis is that the bitwise approach will lead to faster processing times, better solution quality, reduced memory usage, and decreased complexity compared to traditional methods. The bitwise approach is expected to outperform existing evolutionary algorithms on benchmark graph instances. Hence, the proposed solution, referred to as the *BitVertex Evolutionary Algorithm (BitEA)* [12], offers the following contributions:

- 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 thesis is structured as follows.

Chapter 2 provides a comprehensive review of the background and related literature. It begins by defining the graph coloring problem and its various applications, followed by an in-depth discussion of the register allocation problem and its connection to graph coloring. The chapter contrasts the graph coloring approach with the linear scan allocation method, two widely adopted strategies for register allocation, and reviews heuristic optimization algorithms employed for solving the graph coloring problem, emphasizing their strengths and limitations. A significant portion is dedicated to performance acceleration techniques for heuristic optimization, particularly those based on CPU, GPU, and FPGA architectures.

Chapter 3 addresses the vertex-weighted k -coloring problem, beginning with a formal definition and mathematical modeling. The chapter proceeds by discussing traditional problem representations, including adjacency matrix and adjacency list representations, and introduces the BitVertex representation proposed in this thesis with the novel scalable strategy for extending the BitVertex representation to accommodate large graphs.

Chapter 4 presents the BitVertex Evolutionary Algorithm (BitEA), which serves as the central contribution of this study. The chapter describes the individual representation and generation process and provides details on the crossover operations, including the BitVertex Crossover Operator (BitCX) and the Improved Search Back (ISB) operator, accompanied by illustrative examples. The local search operation is described with a focus on strategies designed to enhance solution quality, followed by an overview of the fitness function, population update mechanism, and a discussion of the broader implications of the BitEA framework.

Chapter 5 describes the experimental study conducted to evaluate the performance of the proposed BitEA. The chapter outlines the experimental setup, including the benchmark datasets and evaluation metrics, and presents the results, focusing on computational efficiency and solution quality. Comparative performance tests using InCEA and DIMACS benchmark instances demonstrate the superiority of BitEA.

Chapter 6 concludes the thesis by summarizing the key findings and contributions and outlines potential future research directions, including further optimization of the BitEA framework and its application to other complex combinatorial problems.

2. BACKGROUND AND RELATED WORK

2.1. Graph Coloring Problem

Graph coloring is a fundamental problem in graph theory, where the objective is to assign colors to the vertices of a graph such that no two adjacent vertices share the same color [13]. This problem is formally defined on an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges connecting pairs of vertices. The minimum number of colors required to achieve a *proper coloring* of the graph is referred to as the *chromatic number*. Due to its NP-hardness [7], finding an optimal solution for graph coloring is computationally challenging, especially for large-scale instances.

Graph coloring has many variations, including k -coloring [14], weighted vertex coloring [15], edge coloring [16], and list coloring [17], each addressing specific constraints or requirements in graph theory. In k -coloring [14], the goal is to color the vertices of a graph using at most k distinct colors such that no two adjacent vertices share the same color. On the other hand, in weighted vertex coloring, each vertex is assigned a weight, and each color is associated with the maximum weight of any vertex assigned that color [18]. The objective is to minimize the total weight of the colors while ensuring that no two adjacent vertices share the same color. List coloring [17] is another variation where each vertex in the graph has a list of permissible colors, and the task is to assign a color from each vertex's list such that no adjacent vertices have the same color. These extensions of the basic graph coloring problem add complexity, making them suitable for a broader range of real-world applications.

Many real-world applications can be effectively modeled using various graph coloring variants. These include problems in scheduling [19], frequency assignment [20], social networks [21], network optimization [22], and register allocation in compilers [9]. For instance, in the *frequency assignment problem* [20], the goal is to assign different frequencies to transmitters in a communication network so that no two adjacent transmitters (those that interfere with each other) are assigned the same frequency. This can be modeled as a graph coloring problem, where the transmitters are represented as vertices,

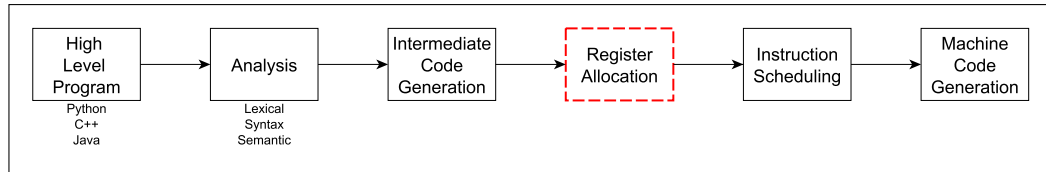


Figure 2.1: Overview of the compilation process with the key phases.

and an edge is drawn between two vertices if the corresponding transmitters interfere. The task is then to assign frequencies (colors) to the vertices such that no two adjacent vertices have the same frequency.

Similarly, the *register allocation problem* [9] in compiler optimization can be modeled using graph coloring. In this problem, the goal is to allocate a limited number of CPU registers to program variables in a way that minimizes memory access operations. Detailed explanations of the register allocation problem and how it can be modeled using graph coloring are provided in the subsequent section.

2.2. Register Allocation Problem

The register allocation problem is a critical component of compiler optimization [3], where the objective is to efficiently assign program variables, also known as *virtual* or *pseudo*-registers, to a limited number of machine registers. Registers are the fastest storage locations that the CPU can access, and optimal allocation is crucial for improving the program’s performance. This task is typically one of the final phases in the compilation process detailed in [23], performed on an intermediate language closely resembling machine code, just before instruction scheduling [24]. The process rewrites a program that initially utilizes an unbounded number of virtual registers into one that operates within the constraints of the available machine registers. Figure 2.1 illustrates an overview of the compilation process, where the high-level program (written in languages such as Python, C++, or Java) undergoes analysis through lexical, syntax, and semantic phases. The program is then translated into an intermediate code representation. Register allocation assigns variables to CPU registers, followed by instruction scheduling, which optimizes the instruction order for better performance. Finally, machine code is generated for execution on the target hardware.

The limited number of machine registers presents several challenges, such as managing

variable lifetimes [25, 26], dealing with interference (i.e., overlapping live ranges) [27], and handling *spilling* [9], which occurs when variables must be temporarily stored in slower memory due to insufficient registers. Variable lifetimes define the segment of a program during which variables remain active and need to be kept in registers for rapid access. Interference arises when two or more variables have overlapping lifetimes, necessitating their assignment to different registers to avoid conflicts, as sharing the same register would result in incorrect program behavior.

The register allocation problem is NP-complete [4], making the development of efficient allocation strategies essential. To address these challenges, compilers often employ graph coloring techniques [2, 5, 28], which, although NP-hard [7, 8], are widely used to manage variable interference by providing a global perspective on the allocation problem. Alternatively, linear scan allocation [6, 29–31] offers a more lightweight approach, balancing performance improvements with resource constraints.

Algorithm 1: Swap Two Variables $swap(a, b)$

Input: Two integers a, b

Output: Swapped values of a and b (only if $a \neq b$)

if $a \neq b$ **then**

$temp \leftarrow a;$	$// \text{ Store the value of } a \text{ in } temp$
$a \leftarrow b;$	$// \text{ Assign the value of } b \text{ to } a$
$b \leftarrow temp;$	$// \text{ Assign the original value of } a \text{ to } b$

return $(a, b);$ $// \text{ Swapped values of } a \text{ and } b, \text{ or unchanged if } a = b$

To illustrate register allocation techniques in compiler design, the swap operation in Algorithm 1, where two variables, a and b , are exchanged, is considered. This swap operation ensures that the value stored in variable a is assigned to b and vice versa, with the constraint that $a \neq b$. The approach to compute this example is inspired by the lecture note¹, which illustrates the register allocation process in the Euclidean algorithm [32].

Algorithm 1, represented in Figure 2.2, shows an intermediate representation (IR) using

¹https://cs420.epfl.ch/archive/20/c/08_reg-alloc.html

virtual registers before register allocation. Each step involves operations on virtual registers, denoted as $v_0, v_1, \dots$, which are later mapped to physical machine registers $R_0, R_1, \dots$ during the register allocation phase. In this representation, each virtual register assignment and arithmetic operation is followed by a *liveness analysis*, showing the sets of registers that are “live” before and after each operation. *Liveness* refers to whether a variable’s value is still needed at a particular program point, and it is essential for determining which variables need to stay in registers.

Program		Liveness {in}{out}
init:	$v_0 \leftarrow R_{LK}$	$\{R_1, R_2, R_{LK}\}\{R_1, R_2, v_0\}$
	$v_1 \leftarrow R_1$	$\{R_1, R_2, v_0\}\{R_2, v_1, v_0\}$
	$v_2 \leftarrow R_2$	$\{R_2, v_1, v_0\}\{v_0, \dots, v_2\}$
ifEqual:	$v_3 \leftarrow v_1 - v_2$	$\{v_0, \dots, v_2\}\{v_0, \dots, v_3\}$
	$v_4 \leftarrow \text{done}$	$\{v_0, \dots, v_3\}\{v_0, \dots, v_4\}$
	if $v_3 = 0$ goto v_4	$\{v_0, \dots, v_4\}\{v_0, \dots, v_3\}$
swap:	$v_5 \leftarrow v_1$	$\{v_0, \dots, v_2\}\{v_0, \dots, v_2, v_5\}$
	$v_1 \leftarrow v_2$	$\{v_0, \dots, v_2, v_5\}\{v_0, \dots, v_2, v_5\}$
	$v_2 \leftarrow v_5$	$\{v_0, \dots, v_2, v_5\}\{v_0, \dots, v_2\}$
done:	$R_2 \leftarrow v_2$	$\{v_0, \dots, v_2\}\{v_0, v_1, R_2\}$
	$R_1 \leftarrow v_1$	$\{v_0, v_1, R_2\}\{v_0, R_1, R_2\}$
	goto v_0	$\{v_0, R_1, R_2\}\{R_1, R_2\}$

Figure 2.2: Intermediate representation of the swap algorithm before register allocation using virtual registers.

For example, in the first instruction, $v_0 \leftarrow R_{LK}$, the liveness analysis {in}{out} shows that before this instruction, registers R_1, R_2 , and R_{LK} are live, meaning their values are needed. After the assignment, v_0 becomes live over R_{LK} , reflecting that its value will be used later in the program. The purpose of liveness analysis is to track the usage of variables throughout the program to efficiently allocate registers. By knowing which variables are live at any point in the program, the compiler can minimize register usage

and determine when variables can be safely stored in slower memory (spilled) or reused without causing conflicts. This process is crucial for both reducing memory access time and ensuring correct program behavior during execution.

This example is solved using both graph coloring and linear scan allocation approaches, as detailed in the subsequent sections, to demonstrate the efficiency of each method for the problem. Both techniques are often referred to as *global*, a term that indicates they allocate registers for the entire function at once.

2.2.1. Graph Coloring Approach

Graph coloring constructs an interference graph, where vertices represent variables, and edges indicate interference between variables [9]. Each variable is characterized by its lifetime, and the goal is to assign registers (represented as colors) to vertices such that no two adjacent vertices (i.e., variables with overlapping lifetimes) share the same color. The challenge lies in utilizing a limited number of registers, denoted as k . When k registers are insufficient to allocate all variables, some variables must be spilled into memory. This scenario is commonly referred to as the k -coloring problem [10], and the performance of a given solution is measured by the number of variables that are spilled. The aim is to minimize this number, thereby improving the efficiency of the program.

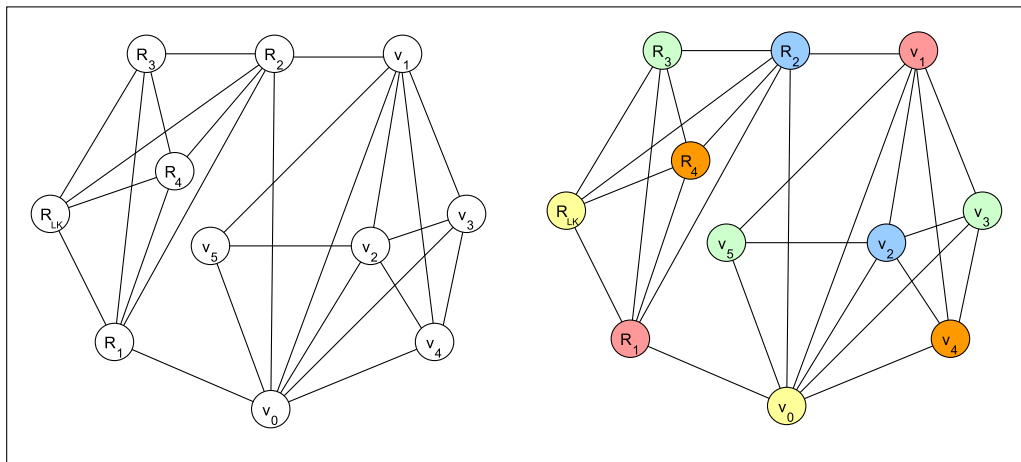


Figure 2.3: The interference graph (Left) and the colored graph (Right) of Figure 2.2.

To construct the interference graph, as shown in Figure 2.3 (left), we use the liveness analysis from Figure 2.2. Each virtual register (variable), such as v_0 , v_1 , and others, is represented as a vertex. An edge is created between two vertices if their corresponding registers interfere with each other, meaning their lifetimes overlap and thus cannot share the same machine register. For instance, v_1 and v_2 interfere because their lifetimes overlap in Figure 2.2, so an edge is drawn between these vertices. This process is repeated for all variable pairs, resulting in the interference graph shown on the left side of Figure 2.3.

Program		Allocation
init:	$v_0 \leftarrow R_{LK}$	$R_{LK} \leftarrow R_{LK}$
	$v_1 \leftarrow R_1$	$R_1 \leftarrow R_1$
	$v_2 \leftarrow R_2$	$R_2 \leftarrow R_2$
ifEqual:	$v_3 \leftarrow v_1 - v_2$	$R_3 \leftarrow R_1 - R_2$
	$v_4 \leftarrow \text{done}$	$R_4 \leftarrow \text{done}$
	if $v_3 = 0$ goto v_4	if $R_3 = 0$ goto R_4
swap:	$v_5 \leftarrow v_1$	$R_3 \leftarrow R_1$
	$v_1 \leftarrow v_2$	$R_1 \leftarrow R_2$
	$v_2 \leftarrow v_5$	$R_2 \leftarrow R_3$
done:	$R_2 \leftarrow v_2$	$R_2 \leftarrow R_2$
	$R_1 \leftarrow v_1$	$R_1 \leftarrow R_1$
	goto v_0	goto R_{LK}

Figure 2.4: Intermediate representation of the swap algorithm after register allocation.

To color the interference graph and obtain the colored graph shown in Figure 2.3 (right), we apply a graph coloring algorithm. The objective is to assign colors (representing machine registers) to each vertex. In this example, five distinct colors (registers) are used to color the graph, corresponding to the five machine registers R_{LK} , R_1 , R_2 , R_3 , and R_4 . The coloring ensures that no two variables with overlapping lifetimes share the

same register. For instance, v_1 is colored red (assigned to register R_1), and v_2 is colored blue (assigned to register R_2), as their lifetimes overlap, preventing them from sharing the same register (color).

Using the colored graph from Figure 2.3, we proceed with the register allocation for the program. Each virtual register v_i is replaced by the corresponding machine register based on the coloring. For instance, v_1 is assigned register R_1 , v_2 is assigned register R_2 , and so on. The final allocation is represented in Figure 2.4. This allocation process minimizes register spills and optimizes the use of the available machine registers.

Allocation	Final Design
init:	
$R_{LK} \leftarrow R_{LK}$	
$R_1 \leftarrow R_1$	
$R_2 \leftarrow R_2$	
ifEqual:	
$R_3 \leftarrow R_1 - R_2$	$R_3 \leftarrow R_1 - R_2$
$R_4 \leftarrow \text{done}$	$R_4 \leftarrow \text{done}$
$\text{if } R_3 = 0 \text{ goto } R_4$	$\text{if } R_3 = 0 \text{ goto } R_4$
swap:	
$R_3 \leftarrow R_1$	$R_3 \leftarrow R_1$
$R_1 \leftarrow R_2$	$R_1 \leftarrow R_2$
$R_2 \leftarrow R_3$	$R_2 \leftarrow R_3$
done:	
$R_2 \leftarrow R_2$	$\text{goto } R_{LK}$
$R_1 \leftarrow R_1$	
$\text{goto } R_{LK}$	

Figure 2.5: Final design of the swap algorithm.

After register allocation, as shown in Figure 2.5, several instructions have been eliminated because the variables are now directly allocated to machine registers. The initial virtual registers v_0 , v_1 and v_2 have been mapped to machine registers R_{LK} , R_1 , and R_2 respectively. Consequently, redundant assignments involving virtual registers are removed, optimizing the code. For instance, in the *init* section, the assignment of v_0

to R_{LK} remains unchanged, while the assignments for v_1 and v_2 directly map to R_1 and R_2 , respectively. Additionally, during the ‘swap’ phase, operations involving the temporary variable v_5 have been optimized, and its values are mapped to register R_3 . The *ifEqual* block conditionally compares the contents of R_1 and R_2 , using register R_3 for the result, and any unnecessary reassignment instructions are removed. Overall, the final design of the swap algorithm is built as the program on the right of Figure 2.5.

In cases where the number of available registers is limited, one or more variables must be spilled to memory to meet the register constraint. For this example, with a limit of 4 available registers, the register allocator must choose a variable to spill. To make an informed decision, the following spill cost formula is applied to each variable:

$$\text{cost}(n) = \frac{\text{rw}(n)}{\text{degree}(n)} \quad (2.1)$$

where $\text{rw}(n)$ refers to the number of times variable n is read or written, while $\text{degree}(n)$ is the number of edges connected to node n in the interference graph. This formula prioritizes spilling variables that are less frequently accessed and have higher degrees in the graph, thus minimizing the performance penalty. Related studies have explored various spill cost analysis approaches in the literature[33].

Table 2.1 presents the calculated spill costs for each variable in the swap algorithm. Based on the results, variable v_0 has the lowest spill cost of 0.29, making it the best candidate for spilling. Consequently, v_0 is selected to be stored in slower memory, and its vertex and associated edges are removed from the interference graph. This updated interference graph, after spilling v_0 , is shown in Figure 2.6.

Following the spill of v_0 , the machine code of the swap algorithm is rewritten to accommodate the need for storing and retrieving v_0 from slower memory. The updated program is shown in Figure 2.7. In the rewritten program, a new variable v_6 is introduced to temporarily hold the value of v_0 in memory. The value of v_0 is stored (pushed) to memory before register allocation begins, and after the swap operation is complete, the value of v_0 is retrieved (popped) into v_7 .

The interference graph and colored graph, shown in Figure 2.8, are updated to reflect the changes resulting from spilling v_0 . The interference graph (left) demonstrates the removal of v_0 and its adjacent edges, reducing the degree of other nodes in the graph.

Table 2.1: Spill cost calculation for variables.

n	$rw(n)$	$degree(n)$	$spillCost(n)$
v_0	2	7	0.29
v_1	5	6	0.83
v_2	4	5	0.8
v_3	2	4	0.5
v_4	2	4	0.5
v_5	3	3	1

Consequently, the coloring algorithm now operates on a simplified graph, resulting in the colored graph (right), where the remaining variables are allocated to the available registers without spilling. The updated register allocation ensures efficient use of available registers while minimizing memory access, as only v_0 is spilled. Additionally, two new variables, v_6 and v_7 , are introduced in the rewritten program to handle the push and pop operations required when spilling v_0 . Since v_6 is used to temporarily store v_0 in memory and v_7 is used to retrieve the value back after the swap operation, they become live during specific sections of the program. Specifically, v_6 becomes live simultaneously with registers R_1 and R_2 , resulting in edges between v_6 and R_1 , and between v_6 and R_2 in the interference graph. Similarly, v_7 becomes live alongside R_1 so an edge between v_7 and R_1 is also added in the updated interference graph.

After applying the same coloring process as illustrated in Figure 2.3, but limited to using only four registers, the rewritten swap algorithm is further optimized through register allocation. Figure 2.9 shows the resulting program after register allocation, where virtual registers v_1 , v_2 , and v_3 are mapped to physical machine registers R_1 , R_2 , and R_3 , respectively. Similarly, variables v_6 and v_7 , which are used for temporary storage during the spill and recovery of v_0 , are also handled during this phase, with v_6 and v_7 ultimately using physical machine registers. Register R_{LK} continues to hold the value of the link register for the return address.

During this process, the unnecessary load and store operations for already-allocated

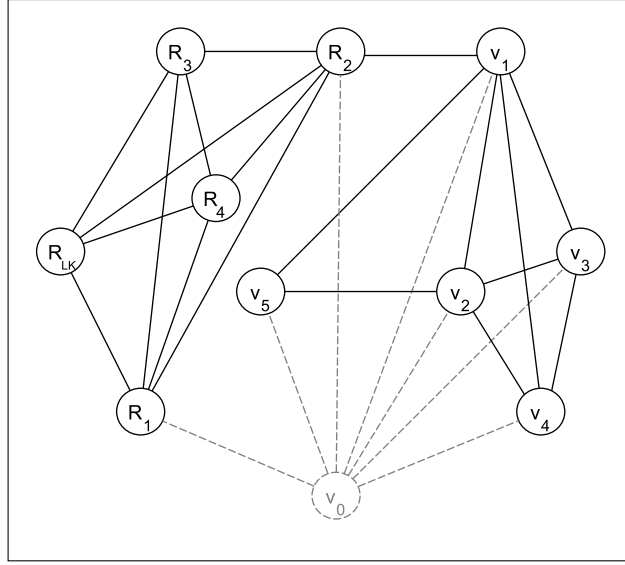


Figure 2.6: Spilling v_0 variable from the interference graph of the swap algorithm.

registers are removed, further streamlining the program. For example, after the initial register allocation, push operations for v_6 and pop operations for v_7 are translated into memory access instructions involving R_{LK} and R_2 . This reduces the program's reliance on virtual registers and optimizes register usage. Furthermore, during the conditional check (ifEqual), the values of R_1 and R_2 are compared directly, and the result is stored in R_3 . By avoiding unnecessary memory accesses, the program becomes more efficient.

After the register allocation phase, the next step involves removing any redundant instructions. Figure 2.10 presents the final design of the rewritten program, where unnecessary instructions have been eliminated, and the program fully utilizes the four available registers. The push and pop operations are retained for R_{LK} and R_2 , while operations involving R_3 are streamlined to directly swap values between R_1 and R_2 without unnecessary intermediate steps. In this optimized design, the program effectively utilizes four registers to avoid excessive memory spills, and the final result is a highly efficient implementation of the swap algorithm with minimal overhead.

This final design efficiently manages register allocation while respecting the constraint of having only four available registers. By eliminating virtual registers and redundant instructions, the algorithm achieves a balance between register usage and memory access, ensuring optimal performance during execution. The remaining push and pop

Original Program	Rewritten Program
init:	$v_6 \leftarrow R_{LK}$
$v_0 \leftarrow R_{LK}$	push v_6
$v_1 \leftarrow R_1$	$v_1 \leftarrow R_1$
$v_2 \leftarrow R_2$	$v_2 \leftarrow R_2$
ifEqual:	$v_3 \leftarrow v_1 - v_2$
$v_4 \leftarrow \text{done}$	$v_4 \leftarrow \text{done}$
if $v_3 = 0$ goto v_4	if $v_3 = 0$ goto v_4
swap:	$v_5 \leftarrow v_1$
$v_1 \leftarrow v_2$	$v_1 \leftarrow v_2$
$v_2 \leftarrow v_5$	$v_2 \leftarrow v_5$
done:	$R_2 \leftarrow v_2$
$R_1 \leftarrow v_1$	$R_1 \leftarrow v_1$
goto v_0	pop v_7
	goto v_7

Figure 2.7: Rewritten intermediate representation of the swap algorithm before register allocation spilling v_0 .

instructions facilitate the necessary interaction with memory, but overall memory access is minimized, resulting in an efficient and well-optimized program design.

2.2.2. Linear Scan Allocation Approach

One of the effective approaches to managing register allocation is the linear scan allocation algorithm, which is widely adopted for its simplicity and efficiency. This approach works by making a single pass over the variables in a program, allocating registers as needed when variables become live and freeing registers when they go out of scope [6]. Linear scan allocation is particularly useful in Just-In-Time (JIT) compilation environments [34] where compilation speed is critical. Although it does not provide the same global optimization as more complex strategies like graph coloring, it offers a good balance between speed and performance in many cases.

In linear scan allocation, the process begins by computing the live intervals of variables

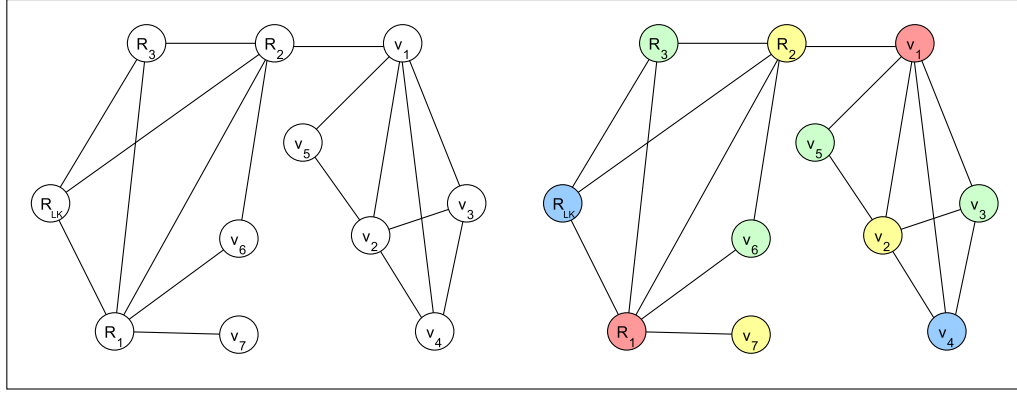


Figure 2.8: The interference graph (left) and the colored graph (right) of the swap algorithm after spilling v_0 .

[35]. These intervals are defined as the period during program execution when the value of a variable is needed or “live.” The notation “[+, -]” is used to represent the start (‘+’) and end (‘-’) points of a variable’s live interval. For instance, a live interval “[1+, 12-]” for a variable indicates that the variable becomes live at instruction 1 and remains live until instruction 12, after which it is no longer needed. These live intervals are critical in determining when variables need registers and when they can be released.

After the live intervals are computed, they are sorted based on their starting points as they are shown in Figure 2.12. Once sorted, registers are allocated to these variables as their live intervals are encountered during a single linear pass through the code, as shown in Table 2.2. In this allocation, five machine registers are available for use. Each time step represents the point at which a variable becomes live (+) or goes out of scope (-). The “Time Active Intervals” column lists all variables that are live at each step, and the “Allocation” column shows how the registers are assigned to these variables. For example, at step 2, both v_0 and v_1 are live, and they are allocated to registers R_3 and R_1 , respectively. This table highlights how registers are dynamically allocated and freed as the program progresses.

If there are insufficient registers available for a variable, as when only four registers are available, the allocator handles spilling by moving the variable with the longest live interval or the least future use to memory, as illustrated in Table 2.3. Spilling is marked in the table by indicating the spilled variable with an S (e.g., $v_0 \rightarrow S$ at step 5). When a register is freed through spilling, the allocator can assign it to another variable that is

Table 2.2: Time steps, active Intervals, and allocations.

Step	Time Active Intervals	Allocation
1+	[1+, 12-]	$v_0 \rightarrow R_3$
2+	[2+, 11-], [1+, 12-]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1$
3+	[3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4+	[4+, 6-], [3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}$
5+	[5+, 6-], [4+, 6-], [3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}, v_4 \rightarrow R_4$
7+	[7+, 10-], [3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_5 \rightarrow R_{LK}$

Table 2.3: Time steps, active intervals, and allocations after spilling v_0 .

Step	Time Active Intervals	Allocation
1+	[1+, 12-]	$v_0 \rightarrow R_{LK}$
2+	[2+, 11-], [1+, 12-]	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1$
3+	[3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4+	[4+, 6-], [3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_3$
5+	[5+, 6-], [4+, 6-], [3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_3, v_4 \rightarrow R_{LK}$
7+	[7+, 10-], [3+, 10-], [2+, 11-], [1+, 12-]	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_5 \rightarrow R_{LK}$

Rewritten Program	Allocation
init: $v_6 \leftarrow R_{LK}$	$R_{LK} \leftarrow R_{LK}$
push v_6	push R_{LK}
$v_1 \leftarrow R_1$	$R_1 \leftarrow R_1$
$v_2 \leftarrow R_2$	$R_2 \leftarrow R_2$
ifEqual: $v_3 \leftarrow v_1 - v_2$	$R_3 \leftarrow R_1 - R_2$
$v_4 \leftarrow \text{done}$	$R_{LK} \leftarrow \text{done}$
if $v_3 = 0$ goto v_4	if $R_3 = 0$ goto R_{LK}
swap: $v_5 \leftarrow v_1$	$R_3 \leftarrow R_1$
$v_1 \leftarrow v_2$	$R_1 \leftarrow R_2$
$v_2 \leftarrow v_5$	$R_2 \leftarrow R_3$
done: $R_2 \leftarrow v_2$	$R_2 \leftarrow R_2$
$R_1 \leftarrow v_1$	$R_1 \leftarrow R_1$
pop v_7	pop R_2
goto v_7	goto R_2

Figure 2.9: Rewritten program of the swap algorithm spilling v_0 after register allocation.

live at that time. This allows the program to continue efficiently, even when the number of variables exceeds the number of available registers.

2.2.3. Graph Coloring vs. Linear Scan Allocation

Graph coloring is a global approach that models the register allocation problem as a graph, where nodes represent variables and edges indicate interference (i.e., overlapping live ranges of variables). The goal is to assign colors, which represent machine registers, in such a way that no two adjacent nodes share the same color. The advantage of this approach is that it provides a comprehensive view of variable lifetimes and interference, often resulting in fewer spills and better overall performance in terms of reduced memory accesses. However, the complexity of building and coloring the interference graph can be high, making it less suitable for Just-In-Time (JIT) compilers [34] or other performance-sensitive compilation environments.

Allocation	Final Design
init:	<code>push R_{LK}</code>
	<code>push R_{LK}</code>
	<code>R₁ <- R₁</code>
	<code>R₂ <- R₂</code>
ifEqual:	<code>R₃ <- R₁ - R₂</code>
	<code>R_{LK} <- done</code>
	<code>if R₃ = 0 goto R_{LK}</code>
swap:	<code>R₃ <- R₁</code>
	<code>R₁ <- R₂</code>
	<code>R₂ <- R₃</code>
done:	<code>pop R₂</code>
	<code>goto R₂</code>
	<code>pop R₂</code>
	<code>goto R₂</code>

Figure 2.10: Final program of the swap algorithm spilling v_0 after register allocation.

In contrast, linear scan allocation is a simpler and more lightweight approach, particularly suitable for environments where fast compilation is critical, such as JIT. It operates by making a single pass over the list of live intervals of variables, allocating registers when variables become live, and freeing registers when they go out of scope. While linear scan allocation is computationally efficient, it can lead to suboptimal register usage, especially in scenarios where many variables are live at the same time. Consequently, it may result in more frequent spills compared to graph coloring.

In summary, graph coloring tends to produce higher-quality allocations by minimizing the number of memory spills at the cost of increased computational overhead. Linear scan allocation, on the other hand, offers faster compilation times but may result in more spills and, hence, reduced runtime performance. For applications where register allocation significantly impacts execution speed, such as those handling complex computations or large datasets, the graph coloring approach is typically preferred. Meanwhile,

Program	Live Intervals n: [+ , -]
init: $v_0 \leftarrow R_{LK}$	v_0 : [1+, 12-]
$v_1 \leftarrow R_1$	v_1 : [2+, 11-]
$v_2 \leftarrow R_2$	v_2 : [3+, 10-]
ifEqual: $v_3 \leftarrow v_1 - v_2$	v_3 : [4+, 6-]
$v_4 \leftarrow \text{done}$	v_4 : [5+, 6-]
$\text{if } v_3 = 0 \text{ goto } v_4$	
swap: $v_5 \leftarrow v_1$	v_5 : [7+, 10-]
$v_1 \leftarrow v_2$	
$v_2 \leftarrow v_5$	
done: $R_2 \leftarrow v_2$	
$R_1 \leftarrow v_1$	
$\text{goto } v_0$	

Figure 2.11: Live interval computation of the intermediate representation of the swap algorithm.

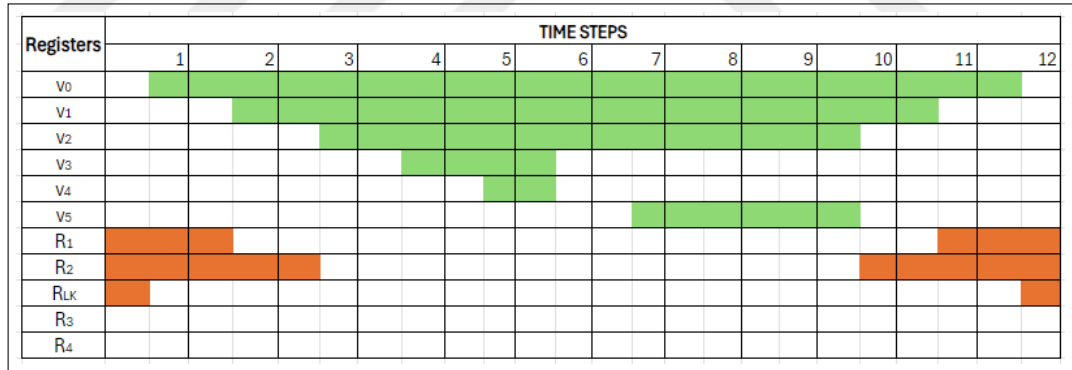


Figure 2.12: Sorted time intervals of virtual and machine registers, respectively.

linear scan is more suited for dynamic environments where fast compilation is critical [36].

In this thesis, we focus on a graph coloring approach for register allocation, as highlighted in the blue section of Figure 2.13. We propose an efficient graph coloring method for addressing the allocation problem by utilizing the global structure of the interference graph to reduce conflicts and ensure efficient register usage, especially for complex and

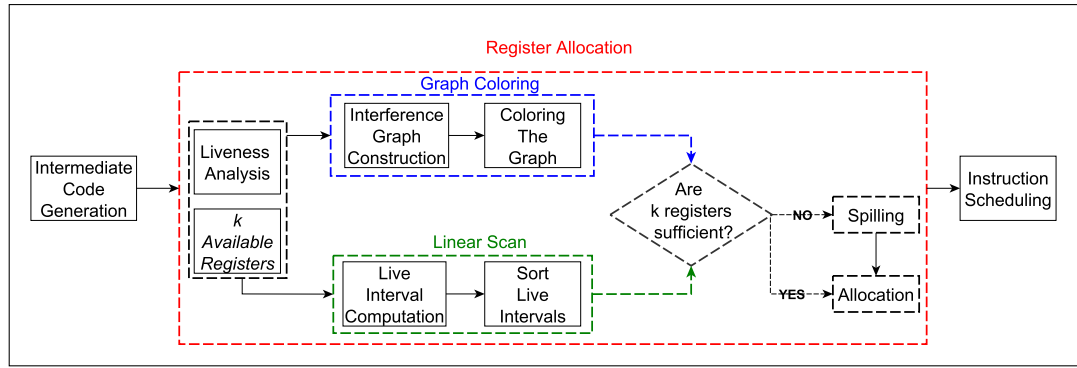


Figure 2.13: Summary of the register allocation process throughout the compilation.

large-scale programs.

2.3. Heuristic Optimization in Register Allocation

Numerous algorithms have been proposed in the literature to address the graph coloring model for the register allocation problem. Chaitin's algorithm [9] is the first algorithm that introduces register allocation problem on graph coloring and the following studies Briggs's [37] and George and Appel's [38] algorithms improve its design by changing the spilling heuristic and the coalescing strategy [39], respectively. These studies have made significant contributions in optimizing compiler efficiency.

2.3.1. Chaitin's Algorithm

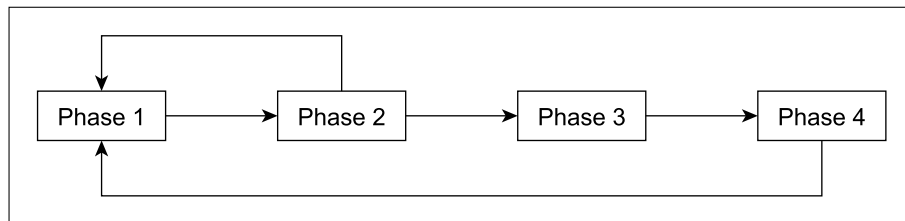


Figure 2.14: Phase flow of Chaitin's Algorithm

Chaitin's algorithm [9] is an efficient and widely adopted method for register allocation in modern compilers. It assigns variables to a limited number of CPU registers during code generation. The algorithm operates on an interference graph, as shown in Figure

2.3, with the objective of mapping variables to a limited number of registers, denoted as k . The algorithm operates in four key phases:

Phase 1: Simplification Phase: As long as there are vertices with fewer than k neighbors in the graph, the algorithm selects a vertex with fewer than k neighbors, pushes the selected vertex onto a stack, and removes the vertex and its incident edges from the graph.

Phase 2: Spill Phase: If all remaining vertices have k or more neighbors, the algorithm selects a vertex to spill based on a heuristic, removes the selected vertex and its incident edges from the graph, adds the vertex to the spill list, and returns to the Simplification Phase.

Phase 3: Register Assignment Phase: Once the graph is simplified, pop vertices off the stack and assign the smallest available register to each vertex, ensuring no conflicts with neighboring vertices.

Phase 4: Spill Code Insertion Phase: If any vertices were spilled, insert load/store instructions for them. Then, rebuild the graph and attempt register allocation again.

The flow between the phases is illustrated in Figure 2.14.

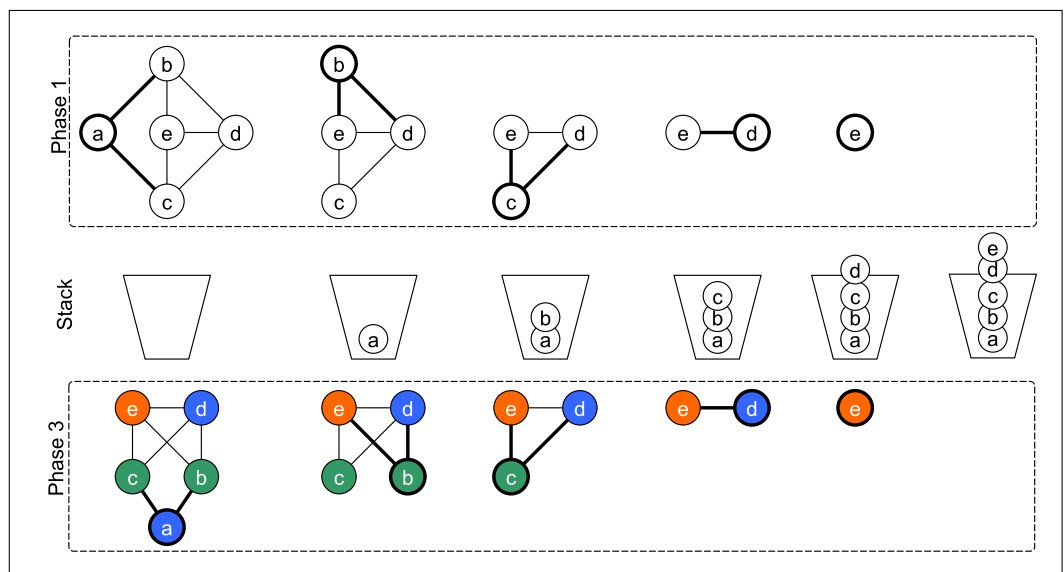


Figure 2.15: Example of Chaitin's Algorithm

The example in Figure 2.15 demonstrates the application of Chaitin's algorithm for register allocation using an interference graph with five variables: a, b, c, d , and e ¹. The algorithm proceeds through the four phases described previously, assuming the availability of $k = 3$ registers.

In the first phase, vertices with fewer than three neighbors are iteratively removed from the interference graph and pushed onto a stack for later register assignment. Initially, node a has only two neighbors (b and c), fewer than the available number of registers, so it is removed from the graph and pushed onto the stack. After this removal, node b now has only two neighbors (c and d), so it is also removed and pushed onto the stack. Similarly, nodes c and d , each having fewer than three neighbors, are removed and pushed onto the stack. Finally, node e , which is isolated, is also removed and placed on the stack. This completes the simplification phase, leaving an empty graph.

In this example, spilling is unnecessary, as all vertices are removed during the simplification phase without exceeding the available registers. Hence, the algorithm does not enter the second phase, and no variables are marked for spilling.

Once all the vertices are pushed onto the stack, the algorithm proceeds to the register assignment phase. Vertices are popped from the stack in reverse order of removal, and each is assigned the smallest available register that does not conflict with its neighbors. Since no variables were spilled in the earlier phases, no spill code needs to be inserted.

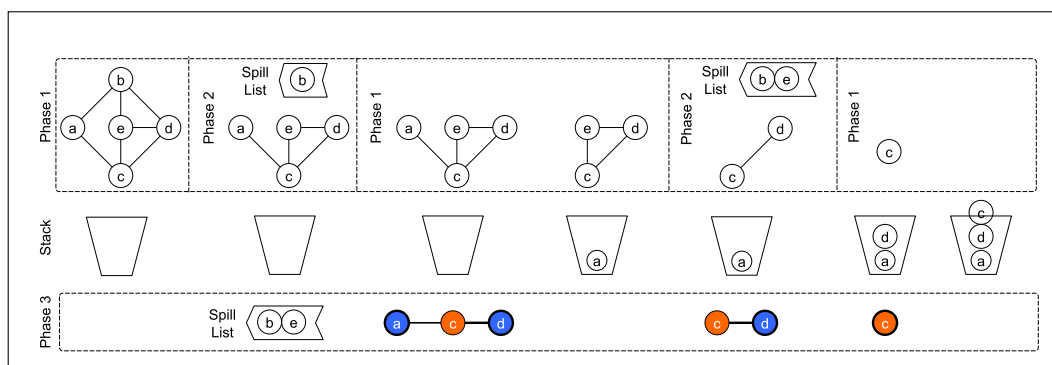


Figure 2.16: Example of Chaitin's Algorithm when $k = 2$.

In the second example illustrated in Figure 2.16, Chaitin's algorithm is applied with

¹<https://www.inf.ed.ac.uk/teaching/courses/copt/lecture-7.pdf>

$k = 2$ registers, demonstrating a scenario where spilling becomes necessary. Initially, there are no vertices with fewer than 2 neighbors, so the spill phase is performed immediately. Node b is selected to be spilled and added to the spill list. After this, the simplification phase is restarted. As depicted, node a is removed first and pushed onto the stack. Once again, there are no vertices with fewer than 2 neighbors, so node e is spilled in the spill phase.

The simplification phase is restarted once more, and node d is removed, followed by node c , both of which now have fewer than 2 neighbors, and are pushed onto the stack. Once the simplification phase is complete, the algorithm proceeds to the register assignment phase. As vertices c , d , and a are popped from the stack, they are assigned the smallest available registers. Since nodes b and e were spilled, spill code is inserted for b and e , which means additional load/store operations will be required during execution.

2.3.2. Briggs' Algorithm

Briggs' Algorithm [37] is an enhancement of Chaitin's Algorithm [9], specifically designed to minimize unnecessary spilling and improve the overall performance of register allocation. Unlike Chaitin's approach, which aggressively spills variables when no simplification is possible, Briggs introduces a more conservative heuristic for handling potential spills. Instead of spilling a vertex as soon as the graph reaches a point where all nodes have k or more neighbors, Briggs' algorithm attempts to coalesce nodes whenever possible, reducing the overall pressure on register allocation.

Applying Briggs' Algorithm to the interference graph in Figure 2.15, the first phase (Simplification) proceeds similarly to Chaitin's algorithm. Vertices with fewer than $k = 3$ neighbors are removed and pushed onto the stack. This process leads to the removal of vertices a , b , c , and d , leaving vertex e isolated, just as in the Chaitin example. However, Briggs' Algorithm does not immediately proceed to the spilling phase. Instead, it applies a conservative coalescing strategy before making any spill decisions. In this case, since there are no vertices left with more than k neighbors, no coalescing or spilling is necessary. The algorithm proceeds to the register assignment phase, where vertices are assigned the smallest available register, as in Chaitin's algorithm. Therefore, Briggs' Algorithm achieves the same result as Chaitin's Algorithm in this example, but with

a more cautious approach to spilling, which becomes more relevant in larger, more complex graphs.

2.3.3. George and Appel’s Algorithm

George and Appel’s Algorithm [38] offers further improvements over both Chaitin’s and Briggs’ approaches. One of the key innovations in this algorithm is the introduction of an optimistic coloring strategy. George and Appel propose that even if a vertex appears to have k or more neighbors during simplification, it may still be possible to assign a register to the vertex after some of its neighbors are removed. Thus, instead of immediately spilling such vertices, George and Appel’s algorithm defers the spilling decision until absolutely necessary.

Additionally, George and Appel utilize a “precolored” strategy, allowing for some variables to be assigned registers before the allocation process begins. This strategy helps optimize register assignment for frequently used variables, particularly in high-performance computing environments.

2.4. Evolutionary Optimization in Register Allocation

The deterministic nature of the algorithms such as Chaitin’s Algorithm [9], 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 [15, 40–43], evolutionary algorithms [1, 2, 14, 44–46], learning methods [47–49] and other heuristic approaches [50] 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 [42] and Integer Linear Programming (ILP) models presented in [15, 43], have shown effectiveness in solving the

weighted vertex coloring problem (WVCP). Specifically, the ILP model proposed in [43], which casts the WVCP as a maximum weight independent set problem [51], 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 [43] within 10 hours of computation time using CPLEX solver [52]. 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) [2], Cost-Oriented Memetic Algorithm (COMA) [45], Pool-based Evolutionary Algorithm (PBEA) [46] and Integrated Crossover Based Evolutionary Algorithm (InCEA). The evolutionary algorithms HEA, COMA, PBEA and InCEA use partition-based representation [53] 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 4.1. 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 3.1. 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 [54] 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 [1] 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 [55] and the weighted vertex coloring problem [56] in the literature.

2.4.1. Hybrid Evolutionary Algorithm

The Hybrid Evolutionary Algorithm (HEA) [2] is a pioneering approach in applying evolutionary techniques to register allocation. HEA builds on the basic structure of evolutionary algorithms, using a population of candidate solutions that evolve over time. One of its key contributions is the introduction of a novel spill metric used during population initialization. This metric evaluates the register pressure and the potential for memory spills in each solution, providing a more informed starting point for the evolutionary process. The spill metric has been widely adopted by subsequent evolutionary algorithms including this thesis.

HEA's crossover mechanism, known as Conflict-Free Partition Crossover (CFPX), combines subsets of conflict-free vertices from parent solutions to generate offspring. This ensures that the offspring inherit favorable traits, such as minimizing conflicts between registers, while maintaining diversity in the population. Through these innovations, HEA significantly outperforms traditional graph coloring algorithms like Briggs' Optimistic Coloring Heuristic (OCH) [37] in both solution quality and computational efficiency.

HEA has served as a foundation for subsequent evolutionary algorithms in register allocation, particularly by demonstrating how spill-aware evolutionary strategies can yield more efficient allocations.

2.4.2. Cost Oriented Memetic Algorithm

The Cost-Oriented Memetic Algorithm (COMA) [45] was developed to overcome the limitations of the Hybrid Evolutionary Algorithm (HEA) [2] in addressing the register allocation problem, where both conflicts and the costs of variables are significant factors. While HEA primarily focuses on minimizing conflicts, COMA introduces a cost-aware approach by incorporating a cost-oriented crossover operator (COPX) and a fitness evaluation metric that accounts for both conflicts and costs.

The Cost-Oriented Crossover Operator (COPX) in COMA is designed to produce offspring solutions that minimize the total cost associated with spilled variables. In contrast to the Conflict-Free Priority Crossover (CFPX) operator in HEA, which only

aims to maximize the number of conflict-free nodes, COPX selects conflict-free nodes based on their weights, prioritizing those with the highest cumulative weight. This results in more cost-efficient solutions.

Comparative experiments indicate that COMA outperforms both HEA and OCH [37] in terms of reducing total costs and the number of spilled variables. This advantage is particularly pronounced in complex graph structures, where COMA's cost-oriented evolutionary strategies prove to be more effective.

2.4.3. Integrated Crossover Based Evolutionary Algorithm

The Integrated Crossover-Based Evolutionary Algorithm (InCEA) [1] stands out among evolutionary algorithms designed for register allocation by introducing a specialized crossover operator and a local search method aimed at optimizing vertex-weighted coloring solutions. This method focuses on forming color classes that minimize conflicts while addressing the fitness function through the total weight of uncolored vertices.

InCEA employs a unique Integrated Crossover Operator (InCX) that constructs offspring by integrating color classes from two parent configurations with a “pool-based” strategy. This strategy was originally proposed in the preliminary study, Pool-Based Evolutionary Algorithm (PBEA) [46], as a method for managing conflicting vertices. In each iteration, two color classes, one from each parent, are merged to form a color class for the offspring. Conflicting vertices are managed by adding them to a pool, and then a search-back operation attempts to place these vertices in previously created color classes where no conflicts arise. This design allows InCEA to increase the number of non-conflicting vertices within each offspring, minimizing the number of uncolored vertices and thus contributing to a lower fitness value. The InCX operator, due to this pool and search-back strategy, improves the likelihood of obtaining a conflict-free solution compared to traditional methods.

Moreover, InCEA introduces a Weighted-Swap local search technique, which optimizes solutions by swapping vertices based on their weights to further reduce the fitness function. The swap operation aims to minimize the sum of weights of uncolored vertices by exchanging lower-weight conflicting vertices in a color class with higher-weight uncolored vertices from the pool. This nuanced adjustment reduces the likelihood of

conflicts while maintaining solution quality.

Experimental evaluations on DIMACS and synthetic benchmarks show InCEA's superior performance in both solution quality and computational time. InCEA outperformed comparable algorithms, including HEA and COMA, in 71% of test cases and achieved competitive performance in additional cases by efficiently utilizing the problem-specific information in its crossover and local search operations.

Figure 2.17, adapted from [1], illustrates the performance of HEA, COMA, and InCEA on six DIMACS graph instances (R100_1gb, R100_5gb, R100_9gb, GEOM120a, queen10_10g, and queen10_10gb) based on the number of uncolored vertices in their best solutions. Each algorithm was executed 20 times for each instance, revealing that InCEA consistently achieved fewer uncolored vertices compared to HEA and COMA. Notably, in the R100 graphs and the queen instances, InCEA exhibited minimal variability and lower median values, indicating more reliable and effective solutions. The GEOM120a instance presented a slight challenge, yet InCEA still matched or exceeded the performance of its counterparts.

These findings reinforce that InCEA's unique integration of a pool-based crossover strategy and weighted-swap local search significantly enhances solution quality, making it a robust approach for tackling vertex-weighted graph coloring problems.

Furthermore, using the computational performance results presented in [1], Table 2.4 illustrates the speedup performance of InCEA compared to HEA and COMA on GEOM and DSJC instances from the DIMACS benchmark suite. The data indicate that InCEA significantly reduces computation time, achieving notable speedups over HEA and COMA.

For smaller graphs like GEOM20 and GEOM30, InCEA achieved impressive speedups of up to 750x and 220x over HEA and COMA, respectively. This efficiency extended to mid-sized graphs such as GEOM50 and GEOM60, maintaining speedups over 50x.

In the more complex DSJC graphs, InCEA demonstrated robust performance, achieving speedups of 210x and 281x for DSJC125.1g compared to HEA and COMA. Even for denser instances like DSJC125.9g, InCEA maintained a competitive edge, with speedups of 1.2x and 5.3x.

These results highlight InCEA's capability to deliver faster solutions, making it well-

Table 2.4: Computation time (in seconds) and speedup performances of InCEA for GEOM and DSJC graphs. The results are adapted from [1].

Graph	V(G)	E(G)	Time(s)			Speedup	
			HEA	COMA	InCEA	InCEA/HEA	InCEA/COMA
GEOM20	20	20	75	103	0.1	750.0	1030.0
GEOM20a	20	37	82	110	0.1	820.0	1100.0
GEOM20b	20	32	77	117	0.1	770.0	1170.0
GEOM30	30	50	156	220	1	156.0	220.0
GEOM30a	30	81	172	247	1	172.0	247.0
GEOM30b	30	81	155	235	1	155.0	235.0
GEOM40	40	78	286	396	2	143.0	198.0
GEOM40a	40	146	293	436	4	73.3	109.0
GEOM40b	40	157	298	430	4	74.5	107.5
GEOM50	50	127	438	596	4	109.5	149.0
GEOM50a	50	238	469	717	9	52.1	79.7
GEOM50b	50	249	445	674	8	55.6	84.3
GEOM60	60	185	673	918	6	112.2	153.0
GEOM60a	60	339	653	1011	16	40.8	63.2
GEOM60b	60	366	669	1030	14	47.8	73.6
GEOM70	70	267	868	1220	13	66.8	93.8
GEOM70a	70	459	913	1377	26	35.1	53.0
GEOM70b	70	488	1004	1543	23	43.7	67.1
GEOM80	80	349	1206	1777	19	63.5	93.5
GEOM80a	80	612	1106	1749	41	27.0	42.7
GEOM80b	80	663	1279	1837	38	33.7	48.3
GEOM90	90	441	1892	2580	22	86.0	117.3
GEOM90a	90	789	1544	2236	61	25.3	36.7
GEOM90b	90	860	1445	2352	74	19.5	31.8
GEOM100	100	547	2211	3101	34	65.0	91.2
GEOM100a	100	992	1915	2875	86	22.3	33.4
GEOM100b	100	1050	1846	2895	96	19.2	30.2
GEOM110	110	638	2790	3411	42	66.4	81.2
GEOM110a	110	1207	2051	3224	118	17.4	27.3
GEOM110b	110	1256	1875	3293	128	14.6	25.7
GEOM120	120	773	2796	3967	70	39.9	56.7
GEOM120a	120	1434	2517	3967	174	14.5	22.8
GEOM120b	120	1491	2735	4428	171	16.0	25.9
DSJC125.1g	125	736	5669	7606	27	210.3	281.7
DSJC125.1gb	125	736	5685	6703	28	203.0	239.4
DSJC125.5g	125	3891	2562	4655	277	9.2	16.8
DSJC125.5gb	125	3891	2807	4528	280	10.0	16.2
DSJC125.9g	125	6961	2768	6560	1315	2.1	5.0
DSJC125.9gb	125	6961	1429	6474	1233	1.2	5.3

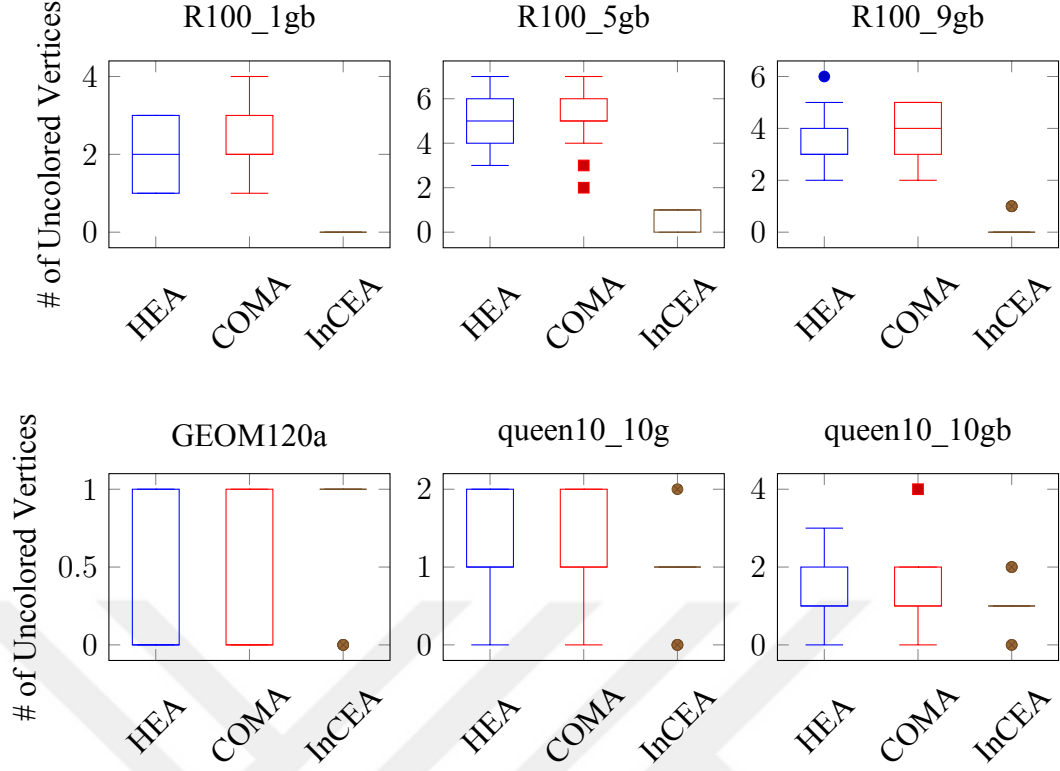


Figure 2.17: Number of uncolored vertices of HEA, COMA and InCEA on a set of DIMACS instances (R100_1gb, R100_5gb, R100_9gb, GEOM120a, queen10_10g and queen10_10gb) using 5, 15, 36, 16, 12 and 12 color classes, respectively. The results are adapted from [1].

suit for real-world, time-sensitive optimization problems. However, due to the high complexity of its evolutionary operators and the extensive number of iterations required, the algorithm is not yet applicable for real-time systems in compilers. In this thesis, we explore enhancements to InCEA’s evolutionary operators to improve computational performance within our novel problem representation and computational approaches. Chapter 4 provides details of our methodology and illustrates InCEA’s evolutionary operators to emphasize the computational and storage efficiency achieved by our approach.

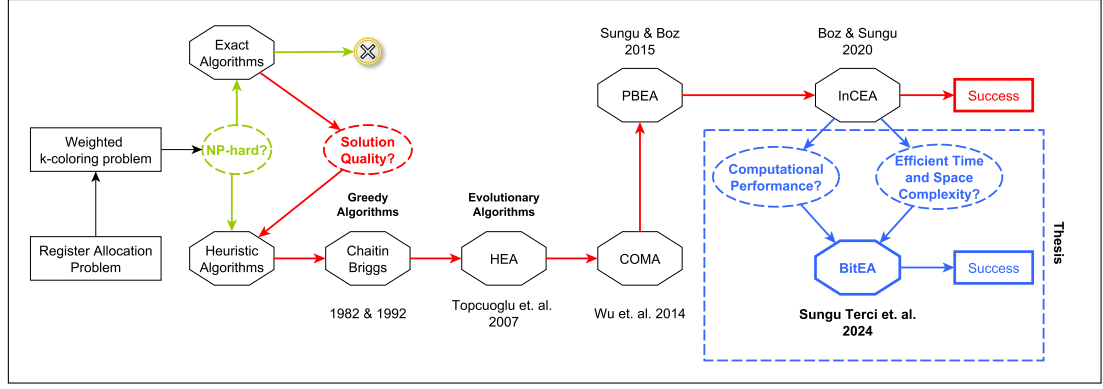


Figure 2.18: Summary of the related studies and the position of this thesis.

2.5. Performance Acceleration for Register Allocation

Figure 2.18 provides a comprehensive summary of the progression of related studies addressing the register allocation problem through graph coloring.

Initially, exact algorithms were employed to solve the register allocation problem, seeking optimal solutions by exhaustively evaluating all possible configurations. However, as problem instances scaled up, these algorithms became computationally impractical due to the NP-hard nature of the graph coloring problem. The exponential increase in complexity rendered exact approaches infeasible for large-scale applications.

To address these limitations, heuristic algorithms such as the Chaitin-Briggs method, developed in the 1980s [9] and refined in 1992 [37], were proposed. These algorithms provided more scalable solutions by approximating feasible outcomes within acceptable computational timeframes. While heuristic methods performed adequately for medium-sized instances, they struggled to maintain solution quality as problem sizes increased.

The limitations of heuristic approaches led to the exploration of evolutionary algorithms, which leverage population-based search strategies and genetic operators, such as crossover and mutation, to iteratively refine solutions. The Hybrid Evolutionary Algorithm (HEA), introduced by Topcuoglu et al. in 2007, was the first evolutionary algorithm specifically designed for the register allocation problem. Subsequent advancements included COMA (proposed by Wu et al. in 2014) and PBEA (developed by Sungu and Boz in 2015), which further enhanced solution quality and computational efficiency. In 2020, Boz and Sungu presented InCEA, which became a benchmark in the field, achieving high-quality solutions with superior computational times compared

to previous methods.

Although significant progress has been made in obtaining high-quality solutions for the register allocation problem, recent challenges have emerged concerning the practicality of these methods for real-time applications due to their high computational times and inefficient space and time complexities, as shown in Figure 2.18.

In recent decades, various studies have sought to accelerate the traditional graph coloring problem. Notable efforts have included parallelizing genetic algorithms to simultaneously generate multiple offspring using crossover and mutation operators assigned to separate threads [57]. Additionally, GPU-based approaches have been employed to enhance the performance of greedy graph coloring algorithms [58–63]. While these methods have successfully reduced computational time, their performance improvements have largely depended on the inherent capabilities of GPU architectures. This limitation arises because these approaches do not fundamentally reduce the time complexity of the core algorithmic operations, due to the constraints imposed by parallelization structures and GPU instruction sets. Furthermore, high-performance GPUs often consume significant power, posing an additional challenge for real-time implementations [64, 65]. Additionally, the high cost of GPUs can be a limiting factor, especially for large-scale or budget-constrained projects.

A recent notable study on the acceleration of graph coloring is BitColor [66], which enhances the performance of the classical *Degree of Saturation (DSATUR)* algorithm [67] by integrating bitwise operations and adapting it for FPGA architectures [68] with a focus on energy efficiency. BitColor refines the DSATUR algorithm by employing bitwise operations to traverse and manage color states for each vertex. The process begins with initializing a flag array of colors for each vertex and proceeds through three main steps: traversing the neighbors of a vertex to identify used colors and marking these in the flag array, selecting the first available color not present in the neighbors' color sets, and assigning the selected color while updating the flag arrays of adjacent vertices. This ensures optimal color distribution without conflicts. By encoding color states as bits, BitColor enables efficient color assignments and restrictions using bitwise AND and OR operations. Experimental results demonstrate that BitColor implemented on FPGA architecture outperforms traditional CPU-based flag array representations, highlighting the advantages of FPGA utilization for computational and energy efficiency.

BitColor follows a vertex-oriented, deterministic approach, sequentially visiting each vertex. In contrast, evolutionary algorithms generate an initial population of color assignments and iteratively refine them through recombination and local search strategies to achieve higher-quality solutions. Unlike the vertex-specific methodology of BitColor, evolutionary algorithms adopt a color-oriented strategy, enabling the manipulation of colors alongside their associated vertices. Consequently, BitColor’s vertex-specific design does not align well with the requirements of evolutionary algorithms for solving graph coloring problems, including register allocation.

To bridge this gap, we introduce a novel bitwise representation called *BitVertex*, which encodes vertices as bits rather than colors. To develop genetic operators suited for the register allocation problem within the BitVertex framework, we propose the *BitVertex Evolutionary Algorithm (BitEA)*. BitEA is the first evolutionary algorithm designed in a bitwise format to enhance the computational efficiency of the register allocation problem by reducing time complexity. This methodology can also serve as a guideline for accelerating other algorithms addressing similar challenges. Within this thesis, we address the open questions highlighted in Figure 2.18 by providing an algorithm that achieves efficient space and time complexity while ensuring fast computational performance.

3. BITVERTEX REPRESENTATION FOR GRAPH COLORING

3.1. Problem Definition and Mathematical Modeling

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 $C = C_1, C_2, \dots, C_k$, 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 (3.1). This constraint is referred to as a k -feasible coloring.

$$\forall v, u \in C_i, (u, v) \notin E, i = 1, 2, \dots, k \quad (3.1)$$

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 (3.2).

$$\text{minimize } f(k) = \sum_{v \in V \setminus \bigcup_{i=1}^k C_i} w(v) \quad (3.2)$$

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.

3.2. Traditional Problem Representations

Graph coloring problems have traditionally been represented using established data structures that efficiently model the relationships between vertices and their neighbors. The most widely used representations for undirected graphs in this context are the

Adjacency Matrix and *Adjacency List*, both of which facilitate efficient graph traversal and adjacency queries, critical operations in graph coloring algorithms.

3.2.1. Adjacency Matrix Representation

An *adjacency matrix* is a square matrix used to represent a graph by capturing the presence or absence of edges between vertex pairs. Given a graph $G(V, E)$, where V represents the set of vertices and E the set of edges, an adjacency matrix A is defined as an $n \times n$ matrix, where n is the number of vertices as built in Figure 3.1. The matrix entry $A[i, j]$ is specified as:

$$A[i, j] = \begin{cases} 1 & \text{if an edge exists between vertex } v_i \text{ and vertex } v_j, \\ 0 & \text{otherwise.} \end{cases}$$

While this representation is simple and provides efficient adjacency checks ($O(1)$), its space complexity of $O(n^2)$ makes it impractical for large or sparse graphs, where many entries in the matrix are zero, resulting in inefficient memory usage. However, for small or dense graphs, the adjacency matrix is still commonly employed, particularly in brute-force approaches to graph coloring, where frequent adjacency queries are required.

3.2.2. Adjacency List Representation

The *adjacency list* as shown in Figure 3.1 is a more memory-efficient data structure, especially suitable for sparse graphs. In this representation, each vertex is associated with a list of its adjacent vertices, and only the existing edges are stored. This results in a space complexity of $O(|V| + |E|)$, making it much more efficient for large graphs with relatively few edges.

While adjacency checks in this representation require traversing the list of neighbors, resulting in an average time complexity of $O(d(v_i))$, where $d(v_i)$ is the degree of vertex v_i , the adjacency list's reduced memory footprint is highly advantageous for large-scale graph coloring problems.

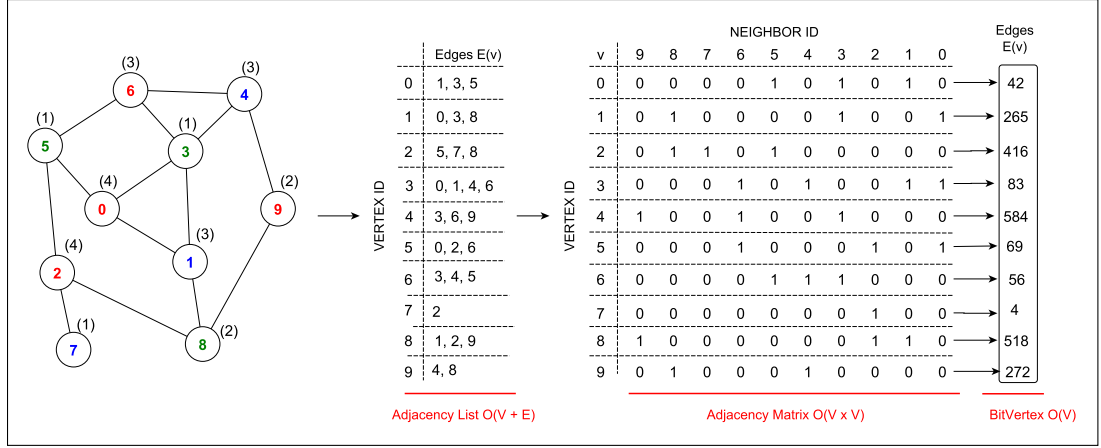


Figure 3.1: BitVertex representation of the example graph [2]

3.3. BitVertex Representation

Traditional representations for graph coloring become inefficient in terms of storage and computational time as graph sizes increase. Recent studies [69] have explored new graph representation strategies to reduce memory scaling costs, particularly for sparse and irregular graph structures. These methods aim to achieve computational speedup and storage efficiency while also providing energy savings.

In this thesis, a novel graph representation, termed *BitVertex*, is proposed to overcome the limitations of traditional representations, specifically addressing the challenges of space complexity and processing efficiency for large-scale graphs. 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 Figure 3.1. 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.

3.4. Scalable BitVertex Representation for Large Graphs

To effectively manage graphs with more than 32 vertices, we propose a novel representation for color classes of the problem. 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. 3.3a. 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.

$$n = \left\lceil \frac{|V|}{32} \right\rceil, \quad (3.3a)$$

$$integer_{id} = vertex_{id} \gg 5, \quad (3.3b)$$

$$bit_{id} = vertex_{id} \& 31. \quad (3.3c)$$

First, the integer index $integer_{id}$ that contains the bit representing the vertex is identified by Eq. 3.3b 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. 3.3c with $\&$ representing the bitwise AND operation, effectively computing $vertex_{id}$ modulo 32.

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 Figure 3.2. 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

		BIT ID																																VERTICES
n integers	INTEGER ID	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0 - 31
	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	32 - 63	
	2	0	0	1	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	64 - 95
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
	n-2	1	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	⋮
	n-1	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	32(n-1) - 32n

32n vertices

Figure 3.2: 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.

set to 1 as shown in Figure 3.2.

This scalable BitVertex representation is not limited to the graph coloring problem. The approach’s efficient handling of large-scale graphs makes it applicable to both large-scale graph coloring applications such as in bioinformatics [70] and a wide range of combinatorial problems that involve extensive graph structures [71]. By reducing memory overhead and facilitating rapid adjacency operations, BitVertex can be adapted for use in various optimization and computational tasks where graph-based data is prevalent, offering a versatile and efficient solution for large-scale graph processing.

4. BITVERTEX EVOLUTIONARY ALGORITHM

In this thesis, we propose the BitVertex Evolutionary Algorithm (BitEA), designed for solving the vertex-weighted k -coloring problem using a hybrid evolutionary approach. The general procedure for the algorithm is outlined in Algorithm 2. The input parameters include a graph G , population size p , number of color classes k , and the desired number of iterations $ItNum$. The main objective is to identify an individual S^* with the optimal fitness value.

The algorithm begins by initializing a population (Pop) of size p through the *InitialPopulation()* function. It then proceeds into an evolutionary loop, iterating for $ItNum$ cycles. At each iteration, two distinct individuals (S_x and S_y) are randomly chosen from the current population. A crossover operation, implemented using the BitVertex Crossover (BitCX) operator (*CrossoverOperation*), is applied to these parents, generating a new individual (S_0) and a pool of conflicting vertices ($Pool$). If the pool is not empty, the local search technique Bit-SWAP of BitEA (*LocalSearch*) is applied to S_0 to resolve conflicts and improve the solution. The population (Pop) is updated by replacing either S_x or S_y with S_0 , provided S_0 has a better fitness value than at least one of its parents. The algorithm ultimately outputs the best individual S^* found during the evolutionary process.

In the literature, the InCEA evolutionary algorithm [1] is recognized for achieving high-quality solutions with respect to both solution quality and computational efficiency when compared to related studies [2, 45, 46]. However, InCEA encodes vertices as integers and represents color classes as lists of integers. This study aims to enhance the design and performance of the InCEA algorithm by transforming its integer-based structure into a bitwise framework. This significant modification involves converting list-based computations in various algorithmic operations into more efficient bitwise calculations. The improved version of the algorithm is henceforth referred to as *the BitVertex Evolutionary Algorithm (BitEA)*.

Algorithm 2: General algorithm of BitEA [1]

Input: Graph G , population size p , number of color classes k , number of iterations

$ItNum$

Output: Individual S^* with the best fitness value

$Pop = \{S_1, \dots, S_p\} \leftarrow \text{InitialPopulation}()$

for $i \leftarrow 1$ **to** $ItNum$ **do**

 Randomly select individuals S_x and S_y from Pop where $x \neq y$

$S_0, Pool \leftarrow \text{CrossoverOperation}(G, k, S_x, S_y)$

if $Pool \neq \emptyset$ **then**

$S_0 \leftarrow \text{LocalSearch}(k, S_0, Pool)$

end

$f(k) \leftarrow \text{Evaluate fitness of } S_0$

end

$Pop \leftarrow \text{UpdatePopulation}(S_0, S_x, S_y)$

$S^* \leftarrow \text{BestIndividual}(Pop)$

4.1. 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. Figure 4.1 (right) 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 Figure 4.1 (left) 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.

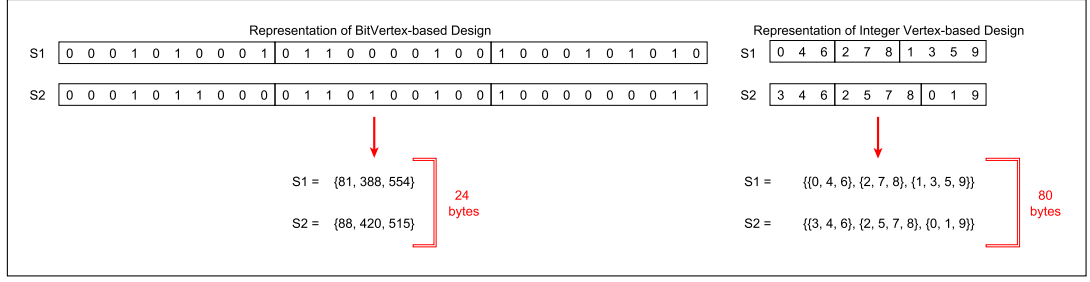


Figure 4.1: Comparison of integer vertex and BitVertex individual representations

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 $\{C_1, C_2, \dots, C_k\}$. 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. BitEA mirrors the population initialization method employed by the InCEA algorithm [1], ensuring consistency with established methodologies while introducing our novel bit-based enhancements.

To facilitate a diverse initial population for BitEA, we employ a spill degree metric, proposed in [2], to sort the vertices. This metric is calculated via three distinct methods, as defined in equations (4.1a), (4.1b), and (4.1c), with respective application to 40%, 40%, and 20% of the individuals, ensuring a heterogeneous set of initial individuals.

$$S_Degree_1(v_i) = w(v_i) \times Degree(v_i), \quad (4.1a)$$

$$S_Degree_2(v_i) = w(v_i) \times Degree(v_i)^2, \quad (4.1b)$$

$$S_Degree_3(v_i) = w(v_i). \quad (4.1c)$$

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 3.

Algorithm 3: 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**

$S_Degree_1(v) \leftarrow w(v) \times Degree(v)$

$S_Degree_2(v) \leftarrow w(v) \times Degree(v)^2$

$S_Degree_3(v) \leftarrow w(v)$

$L1 \leftarrow$ 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**

if v **not added** **then**

 Assign v to a random color class

 Add S_i to *Population*

4.2. Crossover Operation

The crossover operator plays a critical role in the evolutionary algorithm by combining two parent solutions to create an offspring, aiming to produce solutions with improved fitness. In this section, we introduce our crossover operator along with its search-back operator, and we illustrate their functionality using two sample graphs in Figure 3.1 and Figure 4.8 and two sample parent configurations represented firstly in Figure 4.1. The interplay between the crossover and search back operators enhances the algorithm's ability to explore diverse solution spaces, potentially leading to more optimal outcomes.

4.2.1. BitVertex Crossover Operator

Our crossover operation is inspired by the Integrated Crossover Operator (InCX) used in the InCEA algorithm [1] 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 4.1. 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,

$$\bullet S_1 = \{C_0^1, C_1^1, \dots, C_{k-1}^1\} \quad S_2 = \{C_0^2, C_1^2, \dots, C_{k-1}^2\}$$

with a graph G , as shown in Algorithm 4.

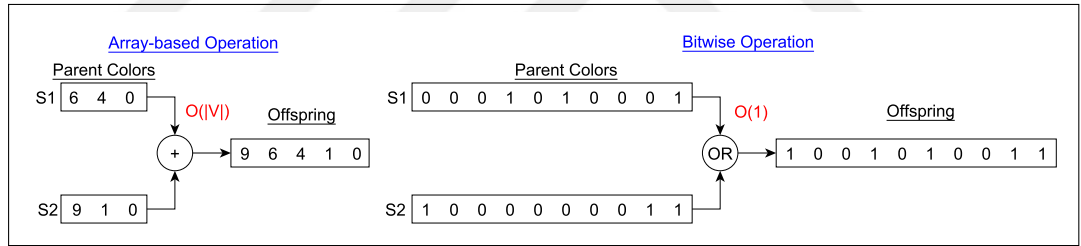


Figure 4.2: Combining two color classes from the parents with traditional array-based and bitwise operators (Operation 1 in Table 4.1).

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 which is processed in constant time $O(1)$ as illustrated in Figure 4.2(right). In the array-based combination operator, as a traditional approach shown in Figure 4.2(left), the same color classes are combined through a series of vertex checks and additions, where each vertex in the parent color classes is individually processed to form

Table 4.1: 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 $v \in \text{colors } C_1 \text{ or } C_2$ Assign v to new color C_3 End If End Loop	$C_3 = C_1 \mid C_2$
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 $v \in C_1 \ \& \ \text{pool}$ Remove v from C_1 End If End Loop End Loop	For each color $C_1 \in S_1$ $C_1 = C_1 \ \& \ \sim \text{pool}$ End Loop
Operation 3: Calculating conflicts between the vertices in the same color	
For each vertex $v_1 \in G(V, E)$ If $v_1 \in \text{color } C$ For each vertex $v_2 > v_1 \in V$ If $v_2 \in C \text{ and } e(v_1, v_2) \in E$ 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 $2^v \ \& \ \text{color } C$ 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 1: $O(1)$
Operation 2: $O(V \cdot k)$	Operation 2: $O(k)$
Operation 3: $O(V ^2)$	Operation 3: $O(V \cdot \log V)$

a color class of the offspring. This process involves iterating over all the vertices in the selected color classes, making the time complexity of the operation $O(|V|)$, where $|V|$ is the maximum number of vertices in the color classes.

Once Operation 1 is performed, if conflicts arise in the color class of the offspring, the BitCX operator counts the conflict number of each vertex using Operation 3 from Table 4.1. This operation has a complexity of $O(|V| \log |V|)$ and determines the number of conflicts for each vertex by using bitwise operations to check adjacency (as shown in Figure 4.3). In this example, the conflicts between vertices v_i and v_j are calculated

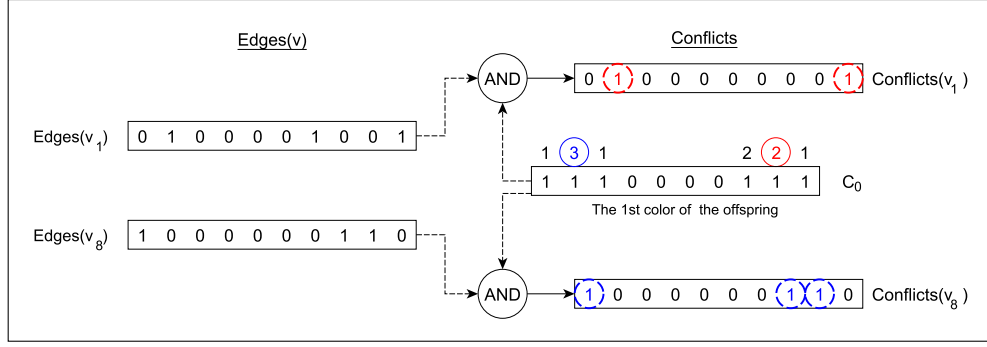


Figure 4.3: Example of calculation of the conflicts in the same color using Operation 3 of bitwise operations in Table 4.1.

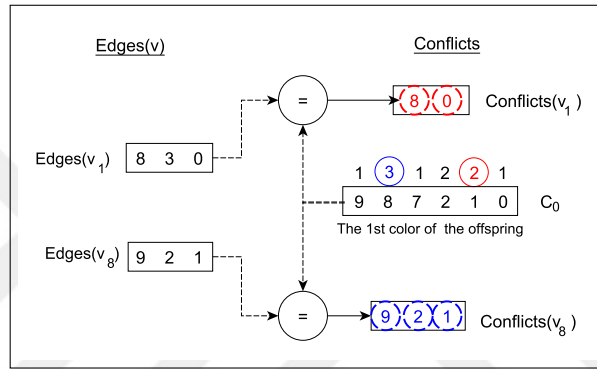


Figure 4.4: Example of calculation of the conflicts in the same color using Operation 3 of array-based operation in Table 4.1.

using a bitwise AND operation on the edges of the vertices within the same color class. Vertices with the highest number of conflicts are iteratively moved to the pool until the color class is conflict-free.

On the other hand, the traditional conflict counting operator from the literature [1, 2, 45, 46] (as illustrated in Figure 4.4) performs a similar conflict calculation but with a complexity of $O(|V|^2)$ using its Operation 3 detailed in Table 4.1. In this method, conflicts are counted through a vertex-by-vertex comparison, where the edges of each vertex are individually checked against all others in the same color class. In the example, each vertex's edges are compared to detect conflicts, and vertices with conflicts (such as v_1 and v_2) are noted. The vertex with the highest conflict number is removed from the color class using Operation 2 in Table 4.1 and placed in the pool. This iterative process continues until no conflicts remain within the color class.

After each combination of two color classes from the parents, except the first one, a search back operation is performed in BitCX operators. 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, 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 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.

4.2.2. Improved Search Back Operator

During the crossover operation, BitCX, whenever a conflict arises in the i^{th} color class C_i (where $1 \leq i < k$), it may result in one or more vertices being relocated to the pool at the completion of C_i . Under such circumstances, investigating the potential of reallocating any vertex in the pool to a previous color class where no conflicts occur can reduce the overall fitness value of the offspring. To address this, we propose the *Improved Search Back operation (ISB)* in this thesis, which integrates vertices from the pool into the previous conflict-free color classes before starting the next combination in the crossover operation.

The original search back strategy was introduced in the InCEA algorithm [1] from the literature, and we have refined it within the ISB operator. The enhancement involves introducing a new criterion for vertex reallocation. The ISB algorithm is the subpart of BitCX 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 4.1, 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 $O(1)$.

Algorithm 4: 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: 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 i^{th} 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: $U^1 \leftarrow U^1 \text{ AND } 2^x, U^2 \leftarrow U^2 \text{ AND } 2^y$

 Combine color classes of parents using bitwise: $C_i \leftarrow C_x^1 \text{ OR } C_y^2$ (Operation 1)

 Mark the vertices as selected: $V_s \leftarrow V_s \text{ OR } C_i$

if $P \neq 0$ **then**

$C_i \leftarrow C_i \text{ OR } P$ (Operation 1)

$P \leftarrow 0$

end

while $C_i \neq \text{conflict-free}$ **do**

 Calculate the maximum conflicting vertex as v_{max} (Operation 3)

 Throw v_{max} into pool: $P \leftarrow P \text{ OR } 2^{v_{max}}$

 Remove v_{max} from the color: $C_i \leftarrow C_i \text{ XOR } 2^{v_{max}}$

end

if $i \geq 1$ **then**

$\text{ImprovedSearchBackOperation}(i, S_0, P, G)$

end

 Remove the vertices $\in P|C_i$ from the parents (Operation 2)

end

Algorithm 5: Improved Search Back Technique

Input: Number of color classes currently produced i , the offspring

$S_0 = \{C_0, \dots, C_{i-1}\}$, the pool $Pool$, the graph G

Output: Updated S_0 , $Pool$

foreach vertex v_p in $Pool$ **do**

$C_{best} \leftarrow -1$

$v_{best} \leftarrow -1$

$bestWeight \leftarrow \infty$

$CF \leftarrow \text{false}$

for $j \leftarrow 0$ **to** $i - 1$ **do**

$conflicts_num[v_p], conflicts[v_p] \leftarrow \text{Operation 3 for } C_j$

if $conflicts_num = 0$ **then**

$Pool \leftarrow Pool \text{ XOR } 2^{v_p}$

$C_j \leftarrow C_j \text{ OR } 2^{v_p}$

$CF \leftarrow \text{true}$

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**

$Pool \leftarrow (Pool \text{ XOR } 2^{v_p}) \text{ OR } 2^{v_{best}}$

$C_{best} \leftarrow (C_{best} \text{ XOR } 2^{v_{best}}) \text{ OR } 2^{v_p}$

end

end

A novel criterion in BitCX, 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 [1] 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.

4.2.3. Illustration of BitCX and ISB Operators

The process of BitCX is illustrated in Figure 4.5. The graph in Figure 3.1 and its two parent configurations in Figure 4.5 are first introduced in HEA algorithm [2]. Additionally, the related studies [45], [46], and [1] have analyzed the same graph and parents to evaluate the efficacy of their crossover operators as CFPX [2], COPX [45], PBC [46] and InCX [1], respectively. To ensure a standardized basis for comparison, we have also applied BitCX to the same input arguments including the same color class combinations. We highlight the performance enhancements achieved from CFPX [2] to BitCX in terms of solution quality and computational efficiency, especially when comparing InCX [1] and BitCX.

In the example for BitCX operator illustrated in Figure 4.5, 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 Figure 4.3 and Operation 3 in Table 4.1. For example, vertex v_8 , which is connected to vertices v_0 , v_1 , and v_9 in the graph given in Figure 3.1, 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

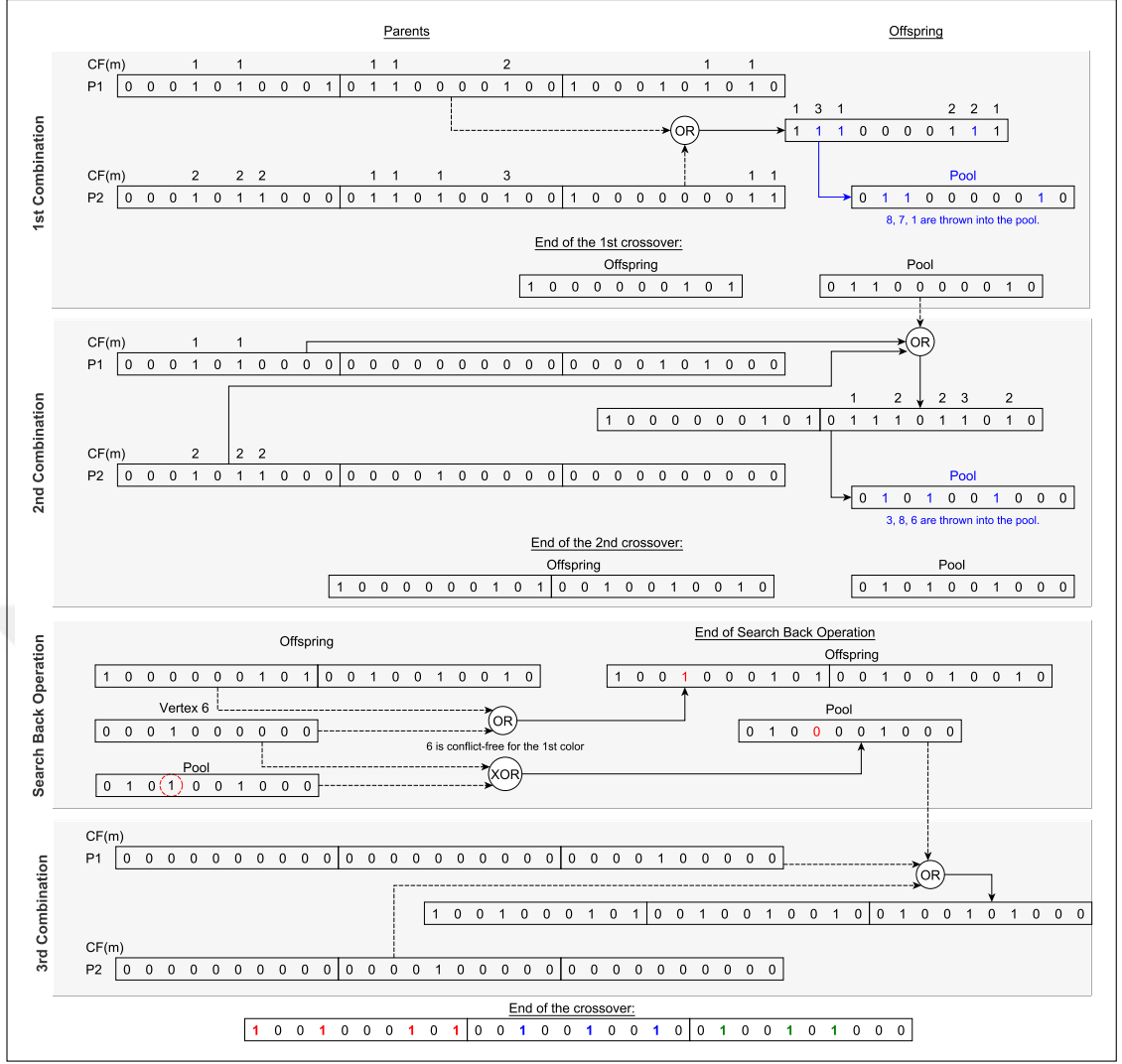


Figure 4.5: Illustration of BitCX operation using two parent configurations and a weighted graph [2].

$Edges(v_8)$ 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 [72] as described in Operation 3 in Table 4.1. The time complexity of Operation 3 is $O(|V| \cdot \log(|V|))$ [72], 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 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 4.1. 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 as the operation is illustrated in Figure 4.6. In contrast, the array-based implementation of Operation 2 has a time complexity of $O(|V| \cdot k)$.

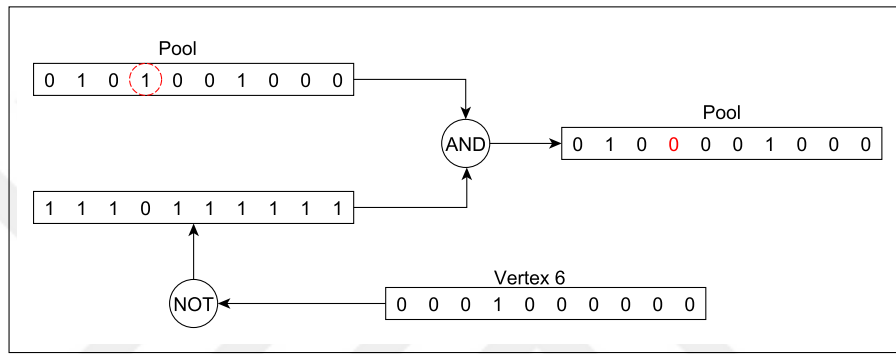


Figure 4.6: Operation 2 using NOT, AND operations.

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 5). Conversely, v_6 does not share edges with any vertices in C_0 and is therefore moved to C_0 using bitwise OR and XOR operations, as illustrated in the search back phase in Figure 4.5. 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.

For the example weighted graph depicted in Fig. 3.1, the output of BitVertex Crossover (BitCX) operator, alongside the outputs of four crossover operators from the literature, are presented in Fig. 4.7. These include Conflict-Free Partition Crossover (CFPX) [2], Cost-Oriented Crossover (COPX) [45], Pool-Based Crossover (PBC) [46] and Integrated Based Crossover (InCX) [1] operators. The offspring generated by CFPX, COPX, and PBC still contain uncolored vertices. Specifically, CFPX and PBC fail to color vertex 6, which has a weight of 3, while COPX is unable to color vertex 3, with a weight of 1. However, both InCX and BitCX successfully color all vertices without the need for a local search technique.

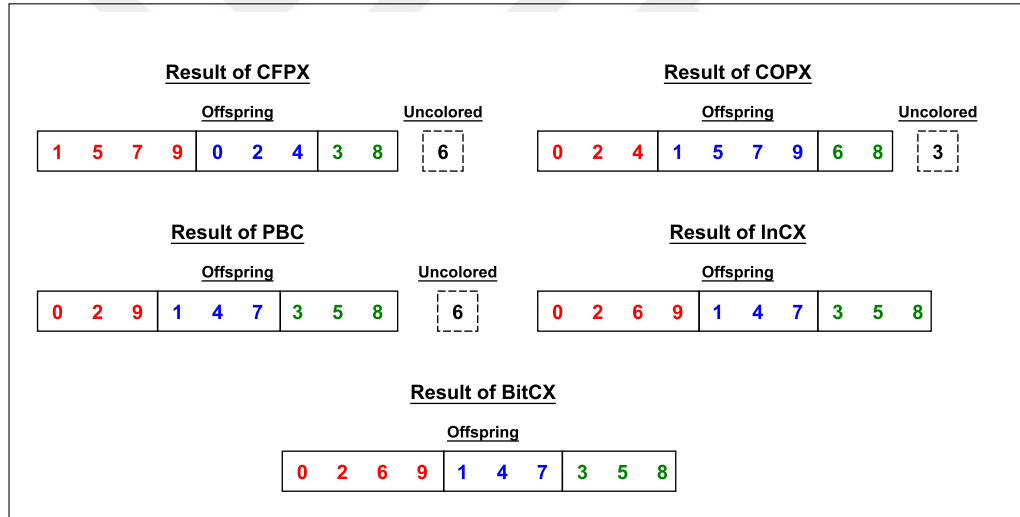


Figure 4.7: Comparison of offsprings produced by different crossover operators for the example weighted graph in Figure 3.1.

The ISB algorithm, outlined in Algorithm 5, 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 Figure 4.8 and Figure 4.9.

In the case illustrated by Figure 4.5, the new criterion introduced by the ISB algorithm is not applicable, as the scenario doesn't align with its prerequisites. Furthermore,

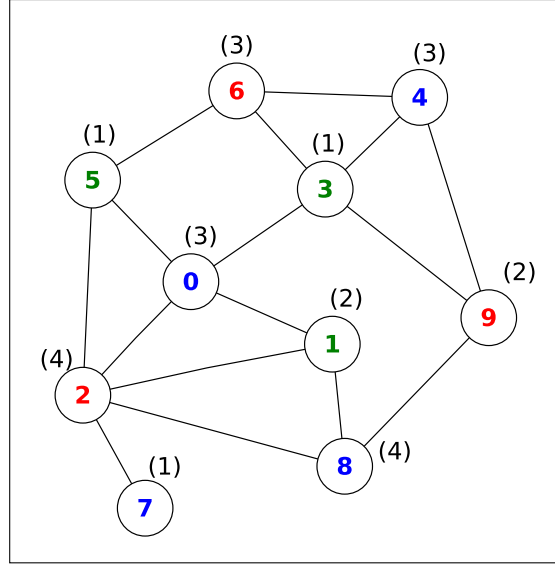


Figure 4.8: The example graph with some edge and vertex weight changes.

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 Figure 3.1, resulting in a new graph illustrated in Figure 4.8. 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 Figure 4.8, the same parent configurations as in Figure 4.5 are used and BitCX, ISB and the local search operations are executed on the parents and illustrated in Figure 4.9.

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

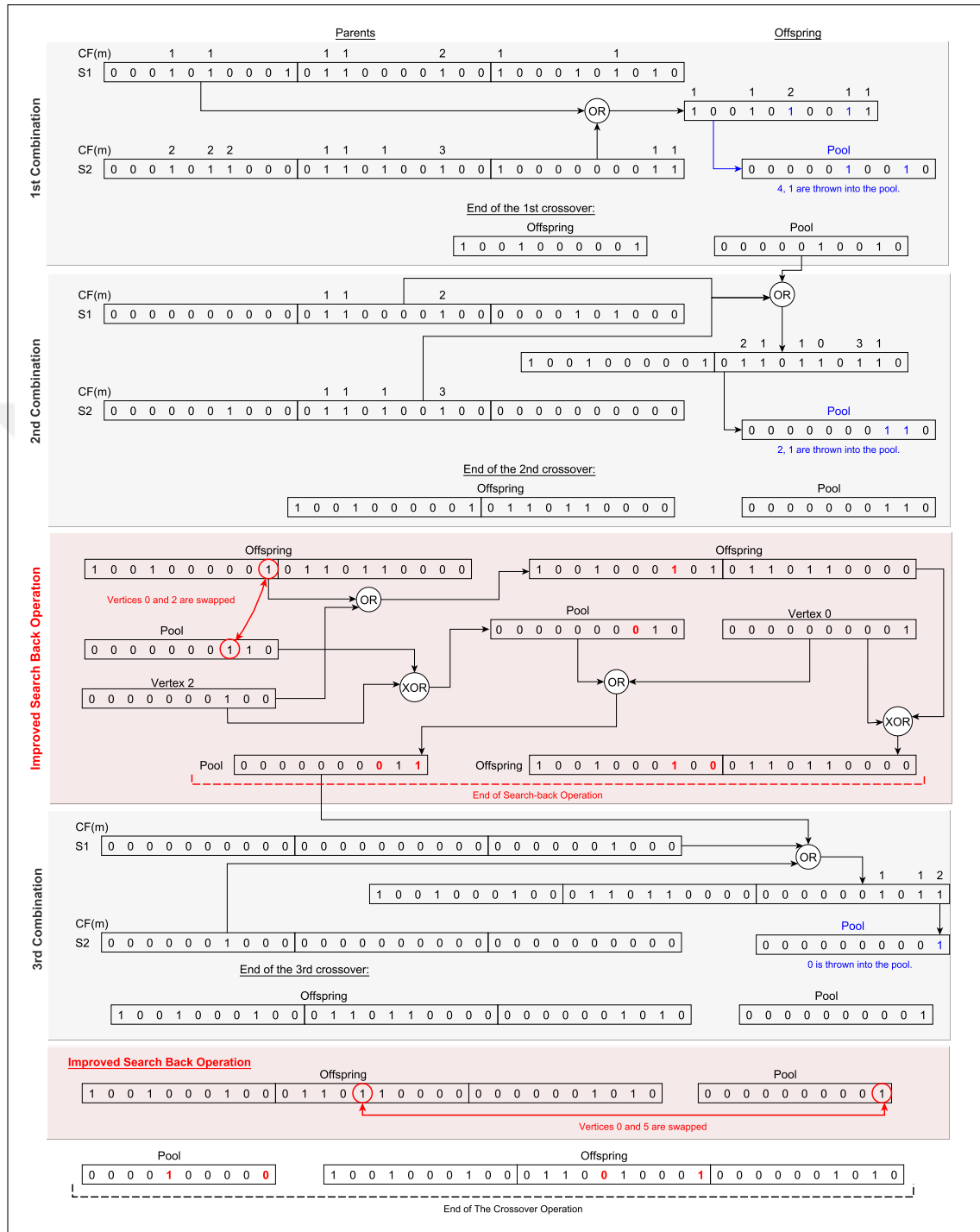
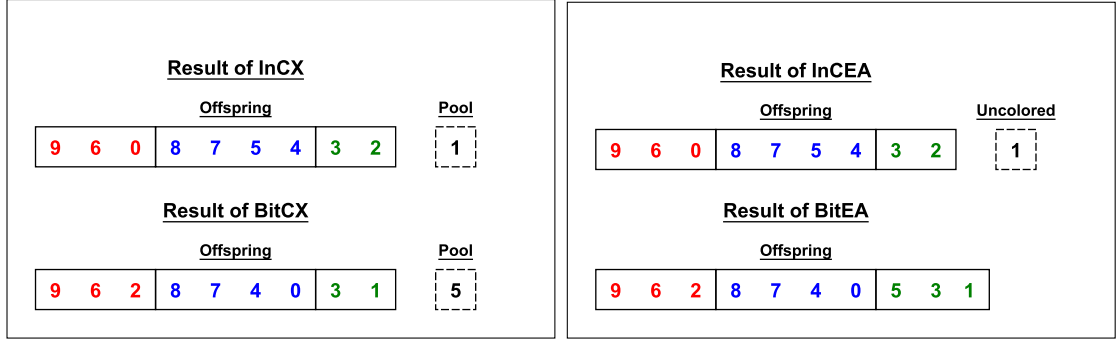


Figure 4.9: BitCX applies on the same parent configurations considering the new graph in Figure 4.8.



(a) The solutions obtained after InCX and BitCX operators using the graph in Figure 4.8.

(b) Comparison of solutions obtained by InCEA and BitEA represented in Integer Vertex-based format using the graph in Figure 4.8.

Figure 4.10: Solutions at the end of the crossover operations and the local search operations, respectively.

ISB algorithm.

The ISB algorithm, illustrated in the red phase in Figure 4.9, 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 4.1, 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 Figure 4.8, 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 Figure 4.9.

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 Figure 4.9, to generate offspring. The offspring produced using the InCX operator is shown in

Figure 4.10a. 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 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 Figure 4.10a. This analysis demonstrates that the ISB operator effectively reduces the fitness value of the solution under identical test conditions.

4.3. Local Search Operation

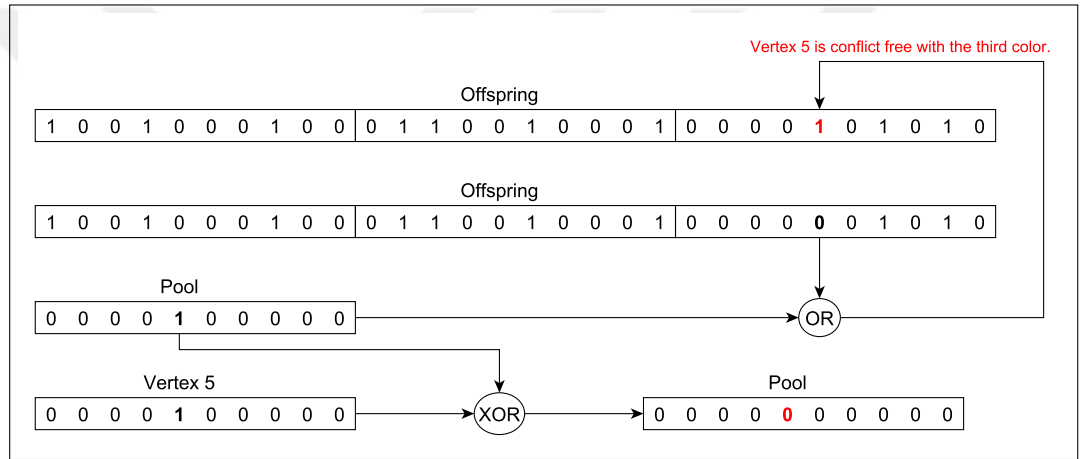


Figure 4.11: Performance of Bit-SWAP technique on the offspring after the BitCX operation in Figure 4.9 using the graph in Figure 4.8.

The local search mechanism in this study mimics the W-SWAP proposed in the InCEA algorithm [1], 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.

Result of InCEA										
Offspring								Uncolored		
9	6	0	8	7	5	4	3	2	1	
Result of BitEA										
Offspring										
9	6	2	8	7	4	0	5	3	1	

Figure 4.12: Comparison of solutions obtained by InCEA and BitEA represented in integer vertex-based format using the graph in Figure 4.8.

Fig. 4.11 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. 4.9. 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. 4.12 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.

4.4. Fitness Function and Population Update

As we formulized the fitness function of the problem in Eq. 3.2 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. 4.10b 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.

4.5. Discussion

Table 4.2 highlights primary differences between the related algorithms, HEA [2], COMA [45], InCEA [1], and BitEA that lie in their representation strategies and computational efficiency. The related algorithms rely on an adjacency matrix detailed in Section 3.3 and an integer-based vertex representation (see Section 4.1), where vertices and their corresponding color classes are handled as lists of integers. In contrast, BitEA introduces a more compact and efficient bitwise framework, referred to as the BitVertex representation. This shift from integer-based operations to bitwise operations significantly improves computational efficiency and reduces memory consumption, especially for large graphs, positioning BitEA as the more scalable option.

CFPX (HEA) and COPX (COMA) rely on exhaustive search mechanisms during the grouping phase, which limits their flexibility and adaptability. The exhaustive nature of their processes results in a limited number of solutions and lower effectiveness in exploring diverse combinations. Additionally, these operators demonstrate suboptimal conflict resolution, leading to offspring that may remain uncolored or require higher computational effort to resolve conflicts post-crossover.

Table 4.2: Comparison of Components among HEA, COMA, InCEA, and BitEA.

Components	HEA [2]	COMA [45]	InCEA [1]	BitEA
Graph Representation	Adjacency Matrix	Adjacency Matrix	Adjacency Matrix	BitVertex
Individual Representation	Integer Vertex	Integer Vertex	Integer Vertex	BitVertex
Population Initialization	Spill Metric-based Initialization [2]			
Crossover Operation	CFPX	COPX	InCX	BitCX
Search Back Operation	Not available		SB	ISB
Local Search Technique	Max Spill Factor [2]		W-Swap	Bit-Swap
Population Update	Elitism			
Fitness Computation	Total weight of uncolored vertices			

InCX (InCEA) introduced a novel pool-based conflict management strategy, allowing more dynamic conflict handling by temporarily placing conflicting vertices in a pool for later reassignment. This approach, combined with color-based grouping, enhances solution diversity and enables a comprehensive exploration of potential configurations.

Our crossover operator BitCX (BitEA) builds on InCX by incorporating bitwise operations to enhance computational efficiency. Unlike InCX, BitCX employs bitwise AND, OR, and XOR operations, reducing time complexity from potentially quadratic to constant or logarithmic for key processes. This allows BitCX to handle vertex grouping, conflict checks, and reallocations more effectively, even in large graphs. As a result, BitCX produces fully colored, conflict-free offspring without needing additional local search, maintaining high performance while minimizing computational costs. This enhancement positions BitCX as a superior crossover operator for complex optimization tasks, setting a new standard in the field.

5. EXPERIMENTAL STUDY

In this thesis, we conducted a comprehensive experimental analysis to compare the performance of InCEA[1] 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.

5.1. Experimental Setup

The algorithms are evaluated using 150 InCEA benchmarks introduced in [1] and 104 DIMACS benchmarks. Both algorithms are implemented in C¹ 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 available². 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 are 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 are assigned to each vertex within these benchmarks by the γ parameter. The specific values for these parameters are detailed in Table 5.1.

The experimental setup in this study involves 104 instances from the DIMACS benchmark suite, categorized into nine different types based on their origin and characteristics. Graphs related to the register allocation problem are denoted as ‘Rn $_{\alpha}gb.col$ ’, 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 [73], 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 [74], used in the study of simulated annealing techniques, are named ‘DSJC $n_{\alpha}gb.col$ ’. Myciel graphs [75], derived from the Mycielski transformation, are identified as ‘myciel5,6,7 $gb.col$ ’.

¹InCEA: <https://gitlab.com/gizemsungu/incea>, BitEA: <https://github.com/ic-cad/BitEA>

²Benchmark instances: <https://gitlab.com/gizemsungu/incea>

Le graphs [76], based on Latin square and evasion problems, are prefixed with 'le' and are designed to challenge coloring and clique algorithms with highly structured combinatorial designs. Zeroin graphs are named with the prefix 'zeroin' and feature sparse structures based on zero-diagonal incidence matrices, useful for testing algorithms on sparse graph instances. Inithx graphs are designated by the prefix 'inithx' and are generated from initial configurations in certain graph coloring heuristics, presenting highly regular graph structures. Lastly, miles graphs are named with the prefix 'miles' and represent distance matrices for cities or locations, making them ideal for testing algorithms on graphs with weighted, geographic-based edge connections.

Table 5.1: Parameters of InCEA benchmarks [1].

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]

5.2. Experimental Results

5.2.1. Performance Evaluation of BitEA and InCEA Varying on Time

The performance analysis of BitEA and InCEA, presented in Figure 5.1, explores their convergence behavior on six different DIMACS benchmark graphs. This evaluation focuses on the time needed for both algorithms to stabilize in terms of the fitness value of their solutions. The tested graphs include R100 with varying edge densities (1gb, 5gb, 9gb), a geometric graph (GEOM120a), and queen graphs with 100 vertices (queen10_10g and queen10_10gb). The performance metric used is the percentage improvement in fitness from the initial population's best individual over time. As lower fitness values

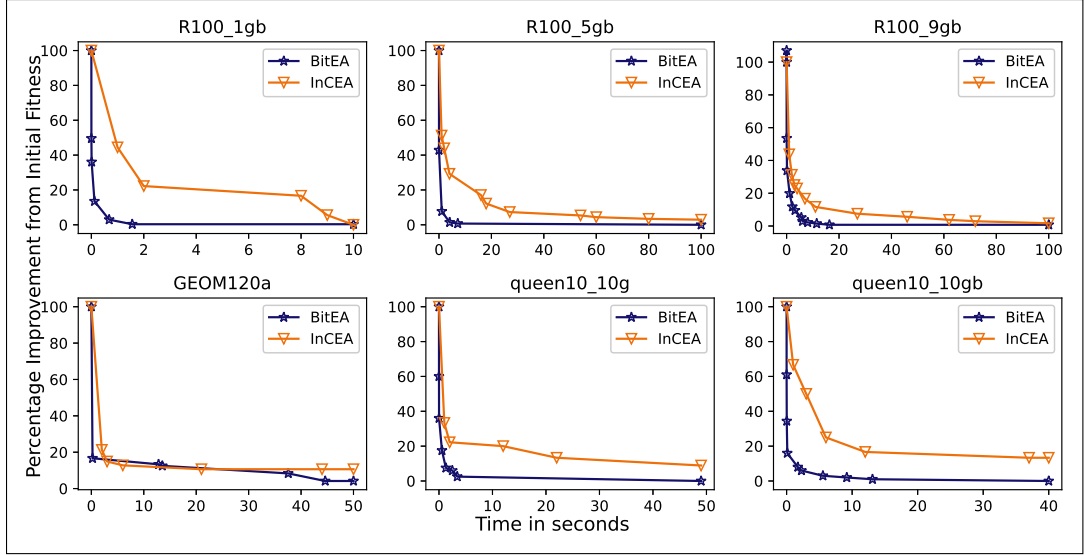


Figure 5.1: Performance evaluation of BitEA for time-wise fitness enhancement on DIMACS graphs (R100_1gb.col, R100_5gb.col, R100_9gb.col, GEOM120a.col, queen10_10g.col and queen10_10gb.col) using 5, 15, 36, 16, 12 and 12 color classes, respectively, within a time limit of 100 seconds.

indicate better solutions, the descending trajectories in the plots represent performance improvements.

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.

5.2.2. Performance Tests on Varying Generation Size

The experimental results in Figure 5.2 assess the convergence performance of BitEA by analyzing the required number of iterations for stabilization. The test involves the same six DIMACS benchmark graphs as in the previous analysis. The improvement metric used is the percentage change from the initial fitness over successive iterations.

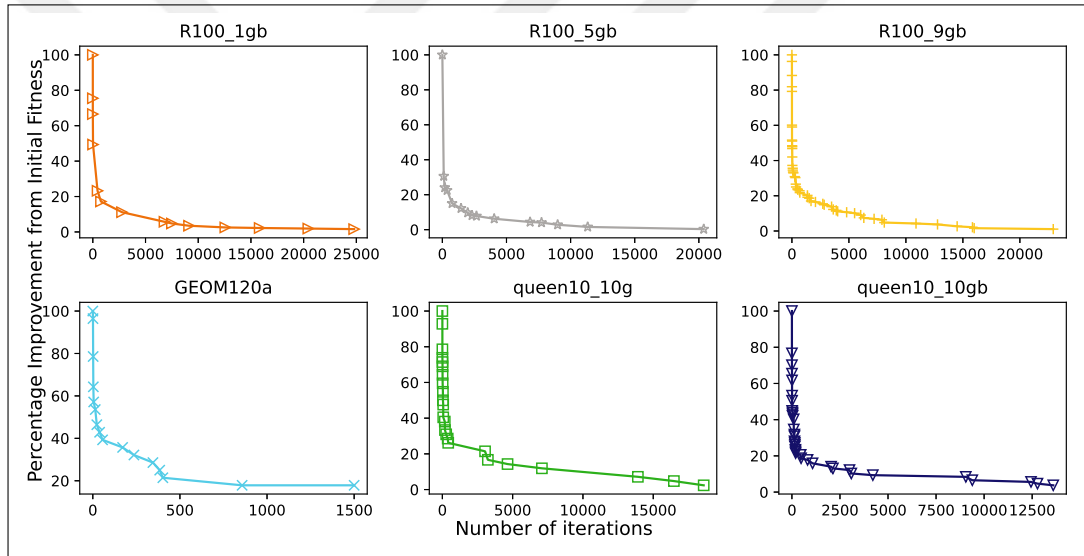


Figure 5.2: Performance evaluation of BitEA for iteration-wise fitness

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 under 30000 iterations.

Results show varied convergence patterns across different graphs. For instance, R100_1gb.col experiences rapid fitness improvement that levels off beyond 10000 iterations, whereas GEOM120a.col displays a sharp decrease in improvement much earlier. The queen graphs follow distinct trajectories; queen10_10gb.col, for example, reaches a

plateau after roughly 7500 iterations. These insights shows that using 10000 iterations in further BitEA and InCEA experiments strikes an effective balance between computational efficiency and solution quality.

5.2.3. Performance Comparison Tests on InCEA Benchmarks

Table 5.2 and Table 5.3 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 Uncolored Vertices), the execution time in seconds (Time(s)) and the resulting speedup of BitEA over InCEA.

Table 5.2 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 5.3 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 5.3 represents the average outcome of 150 runs which across six different edge densities (α), where for each edge 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. This enhancement is especially evident in larger graphs, where BitEA leverages the ISB mechanism to revisit and optimize prior color class allocations, resulting in improved fitness outcomes. Although the ISB operation can sometimes leave slightly more vertices uncolored, this difference is minimal and does not significantly impact the overall solution quality.

Table 5.2: Comparative performance analysis of InCEA and BitEA across different graph sizes n and edge densities α .

n	α (%)	Fitness		Uncolored Vertices		Time(s)		Speed up
		InCEA	BitEA	InCEA	BitEA	InCEA	BitEA	
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

For smaller graphs, such as those with 100 vertices, both InCEA and BitEA produce nearly identical fitness values across all edge densities (α) and color class densities (β). This demonstrates that for simpler problem instances, both algorithms are effective in producing high-quality solutions with minimal variance in results. However, as the graph size increases beyond 200 vertices, BitEA begins to consistently outperform InCEA in terms of fitness values. This trend is particularly pronounced for larger graph sizes, such as $n = 300$, $n = 400$, and $n = 500$. In these cases, BitEA's ISB operator provides an advantage by effectively redistributing and optimizing color classes during crossover, yielding superior fitness values. The results presented in Table 5.2 demonstrate that as edge density increases, BitEA maintains or slightly improves upon the fitness values observed in InCEA, underscoring its robustness in handling denser graph structures.

In Table 5.3, which focuses on variations in color class density (β), BitEA continues to show an advantage as n increases. For larger graphs with $n \geq 300$ and higher β values (e.g., 0.1 and 0.2), BitEA outperforms InCEA by achieving slightly better fitness values. This is due to BitEA's ISB operator, which efficiently utilizes the available color classes and revisits previous color decisions, resulting in a more refined allocation. This improvement is especially noticeable when β is high, as the increased number of available colors allows BitEA to better manage the distribution of colors among vertices.

BitEA's solution quality is particularly noteworthy for graphs with high edge density (α) and color class density (β). For instance, in sparse graphs (e.g., $\alpha = 10\%$), both algorithms achieve similarly optimal results. However, as the edge density increases, BitEA's fitness values remain competitive or slightly better, confirming its scalability and ability to handle complex graph structures. Moreover, when examining scenarios where $n = 300$, $n = 400$, or $n = 500$ with α values at 10%, both InCEA and BitEA achieve the best solutions across all types of color class densities β . This consistency indicates that for sparse graphs, both algorithms are capable of delivering high-quality solutions.

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

Table 5.3: 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		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
	0.1	462	457	130	126	5020	161	31.2
	0.15	492.5	483	138.4	135	5161	181	28.5
	0.2	502	495	142.8	140	5408	201	26.9

from 3.4 to 13.2 times faster than InCEA in Table 5.2 and Table 5.3, respectively. This trend continues for larger graphs, where the speedup becomes even more pronounced. When examining the influence of register density, represented by color class density (β), on the speedup performance in Table 5.3, it is evident that BitEA's efficiency improves as β increases. This is because a higher color class density enlarges the search space, allowing each vertex to have more available color options. This expanded search space facilitates the algorithm's ability to find feasible color allocations more efficiently, reducing the need for complex conflict-resolution steps. For example, with $n = 300$ and $\beta = 0.2$, BitEA achieves a speedup factor of 44.2, which is significantly higher than the speedup observed at lower β values, such as 6.4 for $\beta = 0.04$. The trend of increasing speedup with higher β values is consistently seen across various graph sizes, indicating that BitEA's algorithmic structure is well-suited for handling scenarios where the register density provides more flexibility.

Conversely, as the edge density (α) increases, the speedup performance of BitEA decreases. This behavior can be attributed to the nature of denser graphs, which inherently have more connections between vertices, leading to a more constrained and complex search space. The need for extensive search operations in such a limited space results in a reduction in the algorithm's relative speed advantage. For instance, in Table 5.2, with $n = 300$ and $\alpha = 10\%$, BitEA achieves a speedup factor of 121.7. However, as the edge density increases to 90%, the speedup decreases to 19.9.

This trend highlights that while BitEA performs well in situations where the search space is larger and more flexible (i.e., higher register densities), its speedup decreases in denser graphs due to the increased computational burden associated with managing conflicts in a more constrained search space. The extensive search operations required in such scenarios reduce the comparative advantage of BitEA over InCEA. Nevertheless, BitEA maintains considerable speedup across all tested configurations, demonstrating its robust performance in various graph conditions, particularly when register density is high and edge density is moderate.

5.2.4. Performance Comparison Tests on DIMACS Benchmarks

In this section, we present an evaluation of our proposed algorithm BitEA against InCEA on DIMACS benchmark 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 5.5 and Table 5.6, 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.

Table 5.4: Computation time distribution and iterations of InCEA and BitEA in 500 seconds for 6 DIMACS instances.

Instance	Crossover (%)		Local Search (%)		Fitness (%)		Iterations	
	InCEA	BitEA	InCEA	BitEA	InCEA	BitEA	InCEA	BitEA
R100_1gb	94.8	85.23	4.6	12.67	0.6	1.0	595,703	7,021,728
R100_5gb	97.4	87.67	2.2	11.6	0.4	0.3	115,649	2,464,088
R100_9gb	99.4	91.17	0.5	8.47	0.1	0.13	18,160	1,223,657
queen10_10g	97.7	89.95	2.2	9.26	0.2	0.34	195,163	2,628,128
queen10_10gb	97.7	89.54	2.0	9.64	0.3	0.36	110,366	2,670,860
GEOM120a	99.2	93.31	0.7	6.08	0.1	0.24	57,875	2,162,162

In Table 5.5, the performance of BitEA is compared to InCEA on GEOM graphs. For smaller graph instances such as GEOM20 to GEOM50, the speed-up factor is modest, ranging between 1 and 11, indicating that BitEA performs similarly to InCEA in these cases. However, as the graph size increases, the computational efficiency of BitEA becomes more pronounced. The standout cases in this table are GEOM120a and GEOM120b, where BitEA reduces execution time significantly compared to InCEA, showcasing its advantage in handling larger, more complex geometric graphs.

Table 5.6 illustrates the performance of BitEA on the R graphs, which are related to the register allocation problem. The speedup factors for these graphs range from 2.5 to over 40, depending on the graph size and edge density. Notably, for instances like R50_9g

Table 5.5: Comparative analysis of the InCEA and BitEA algorithms for GEOM and DSJC graphs with their computation time.

Graph	V(G)	E(G)	InCEA		BitEA		Speedup
			k	Time(s)	k	Time(s)	
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
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

and R100_9g, BitEA achieves speedup factors of 26 and 42.8, respectively, while maintaining the same chromatic number as InCEA. This demonstrates that BitEA is particularly effective in reducing execution time for highly connected register allocation graphs, where the number of edges is significantly higher.

In Table 5.6, BitEA consistently outperforms InCEA on the myciel and queen graphs, both in terms of execution time and chromatic number. For the myciel graphs, BitEA achieves speedup factors up to 36.8 for myciel7g, which is a significant improvement over InCEA. The results for queen graphs are also promising, with speedup factors up to 14.6 for queen10_10g.

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 5.5 and Table 5.6. 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.

In addition to Tables 5.5 and 5.6, which summarize the main performance results of BitEA and InCEA on DIMACS benchmark graphs, three supplementary tables provide further insights into the computational characteristics and performance metrics of BitEA. Table 5.4 presents a detailed breakdown of computation time distribution and iteration counts for both algorithms over 500 seconds for six specific DIMACS instances. This table highlights the time distribution of computational resources, such as the percentages devoted to crossover, local search, and fitness evaluation, along with the number of iterations achieved in 500 seconds by each algorithm.

The results demonstrate that BitEA achieves a significantly higher number of iterations than InCEA, emphasizing its enhanced computational efficiency. The primary factor contributing to this performance is the efficient time complexity of BitEA's crossover operator, BitCX, which consumes less time than the InCX crossover operator. Although BitCX includes additional operations, such as the ISB operator, its use of bitwise-friendly operations ensures efficient time management. This efficiency enables BitCX to operate

Table 5.6: Performance evaluation of the InCEA and BitEA algorithms on R, myciel, and queen graphs from the DIMACS benchmarks.

Graph	V	E	InCEA		BitEA		Speed up
			k	Time(s)	k	Time(s)	
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

more rapidly, reducing the time allocated for crossover and allowing more iterations to be completed in the same time frame.

In terms of local search operations, BitEA employs the Bit-SWAP operator, which shares the same evolutionary strategies as InCEA's W-SWAP operator, detailed in Section 4.3. The key distinction lies in the use of bitwise computations in Bit-SWAP, resulting in faster execution. While BitEA's local search operation benefits from the speed of bitwise operations, the percentage of time allocated to this component increases due to the significant reduction in crossover time. This shift highlights the more balanced distribution of computational time between local search and crossover in BitEA compared to InCEA.

Regarding fitness evaluation, the distribution of time spent is nearly the same for both algorithms as they share identical operations in this component. The consistency in time allocation for fitness evaluation indicates that neither algorithm holds an advantage in this aspect, and the efficiency gains of BitEA are primarily attributed to improvements in other computational areas. This parity ensures that any observed performance enhancements are due to the other optimized processes within BitEA rather than differences in fitness evaluation.

Overall, the time distribution and iteration counts presented in Table 5.4 highlight BitEA's strategic allocation of computational resources. By utilizing the efficiency of bitwise operations in the crossover, local search, and fitness evaluation processes, BitEA can complete significantly more iterations within 500 seconds, leading to enhanced solution quality and computational performance compared to InCEA.

Table 5.7 presents an evaluation of the BitEA algorithm applied to challenging DIMACS benchmark graph instances, with each instance being tested over 20 independent runs. The table outlines critical performance metrics, including the chromatic number (k), the average number of uncolored vertices ($U. Vertices$), the average fitness value, the execution time for achieving the chromatic number ($k_{Time}(s)$), the total execution time ($Time(s)$), and the success rate (SR) representing the number of successful runs where the best chromatic number was found.

The chromatic number (k) is indicative of BitEA's capability to identify optimal or near-optimal solutions across various graph instances. The results demonstrate that BitEA can

Table 5.7: Comprehensive performance evaluation of BitEA on hard DIMACS benchmark instances.

Graph	k	U. Vertices	Fitness	$k_{Time}(s)$	Time(s)	SR
DSJC1000.1	27	1.1	1.1	168.7334	431.2218	4/20
DSJC1000.5	111	0.65	0.65	792.9499	2583.5942	7/20
DSJC1000.9	280	1.45	1.45	3545.0970	7614.5349	3/20
DSJC250.1	10	0.85	1	7.0699	15.1478	4/20
DSJC250.5	35	1.15	1.25	35.2187	62.8509	2/20
DSJC250.9	80	1.7	1.85	66.0321	138.9449	1/20
DSJC500.1	17	0	0	0.5137	73.7315	20/20
DSJC500.9	150	2.35	2.4	505.1733	1003.9363	1/20
DSJR500.1	12	0	0	1.1075	14.0642	20/20
inithx.i.1	54	0	0	0.0008	19.0800	20/20
inithx.i.2	31	0	0	0.0018	14.0822	20/20
inithx.i.3	31	0	0	0.0007	14.7071	20/20
le450_15a	19	1.9	2.75	25.9620	49.8827	1/20
le450_15b	20	0.05	0.05	5.7813	47.2233	19/20
le450_15c	28	0.1	0.1	16.8062	88.8235	18/20
le450_25a	28	0.2	0.2	32.2842	90.3829	16/20
le450_25b	27	0.15	0.15	12.3384	39.2342	17/20
le450_25c	34	0	0	27.0464	96.9223	20/20
le450_25d	34	0.25	0.25	21.5944	96.7168	15/20
miles1000	42	0	0	0.9720	8.8491	20/20
miles1500	73	0	0	0.0012	20.6502	20/20
miles250	8	0	0	0.0003	0.4793	20/20
miles500	20	0	0	0.0027	1.4586	20/20
mulsol.i.5	31	0	0	0.0003	7.5506	20/20
queen13_13	16	0.85	0.9	4.8004	15.6063	3/20
queen14_14	17	1.55	1.55	9.5269	21.7020	1/20
queen15_15	19	0	0	1.0046	27.4911	20/20
queen16_16	20	0	0	8.1644	35.8529	20/20
zeroin.i.1	49	0	0	0.0003	8.4505	20/20
zeroin.i.2	30	0	0	0.0007	5.5539	20/20
zeroin.i.3	30	0	0	0.0037	5.1437	20/20

achieve low chromatic numbers for many hard instances, highlighting its effectiveness in tackling complex graph coloring problems. The success rate (SR), denoting the number of runs where the algorithm successfully reached the best chromatic number, provides further insight into BitEA's consistency. For certain instances, such as DSJC500.1, DSJR500.1, and queen16_16, BitEA achieves a success rate of 20, illustrating its reliability in consistently producing high-quality solutions.

The average number of uncolored vertices ($U. Vertices$) reflects the completeness of the solutions generated by BitEA. A value of 0 implies that all vertices are successfully colored, indicating a valid solution. For the majority of instances, the number of uncolored vertices is zero, confirming BitEA's strong capability to solve the coloring problem fully. Similarly, the average fitness value, which measures the solution quality, is often close to or exactly zero for most instances, signifying that the algorithm finds solutions with minimal conflicts or uncolored vertices.

The execution time metrics provide an additional dimension to the analysis. The $k_{Time}(s)$ column represents the average time taken for BitEA to reach the chromatic number, while the total execution time ($Time(s)$) accounts for the entire runtime of each trial. A comparison between these times indicates that BitEA frequently identifies the best chromatic number before the total runtime ends. This suggests that the algorithm, even after finding an optimal solution, continues to explore potential improvements, thereby maximizing its use of the available runtime.

Despite its strong performance, BitEA exhibits variability in success rates for more complex, dense graph instances such as DSJC1000.9 and DSJC250.9, where the success rates are lower (e.g., 3 and 1, respectively). This variability highlights the inherent challenge posed by these graph instances, reflecting the difficulty of consistently finding optimal solutions within a limited number of runs. Nonetheless, BitEA's ability to reach optimal chromatic numbers in at least some of these runs underscores its potential to address complex problem spaces effectively.

Overall, Table 5.7 highlights BitEA's strengths in solving graph coloring problems on hard DIMACS instances. The algorithm's ability to reach optimal chromatic numbers, maintain high solution quality with minimal uncolored vertices, and achieve favorable computational efficiency emphasizes its suitability for tackling challenging graph color-

ing problems. The consistently high success rates for several instances further validate BitEA’s robustness, while its adaptability to diverse graph types underlines its versatility as an effective solution approach in the domain of graph coloring.



6. CONCLUSION

This thesis introduces the BitVertex Evolutionary Algorithm (BitEA), a novel approach to the register allocation problem, building upon a bit-based representation and bitwise operations to significantly enhance computational efficiency and solution quality. Our extensive experimental evaluations with 150 InCEA and 104 DIMACS benchmarks demonstrate that BitEA provides remarkable performance improvements when compared to the Integrated Crossover-Based Evolutionary Algorithm (InCEA). Specifically, BitEA achieves substantial speedup factors, ranging from 3 to 577.4 in specific configurations, particularly for larger graph instances. The BitVertex representation, coupled with the integration of improved search back (ISB) and local search (Bit-SWAP) strategies, not only optimizes execution time but also improves solution quality by achieving lower chromatic numbers. BitEA outperforms InCEA across 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.

In Chapter 2, we provided detailed insights into the BitColor representation [66] from the literature. BitColor computes the colors as bits and uses bitwise operations to traverse the conflicting colors for a vertex. As a future research direction, we propose the development of a hybrid heuristic algorithm that combines the strengths of the BitVertex and BitColor representations. This hybrid approach would aim to create a fully bitwise representation and operation, reducing time complexity significantly over BitEA. This is particularly important for the register allocation process in compiler design, where the time performance during allocation is crucial for achieving optimal code execution and resource utilization.

To evaluate BitEA for the register allocation problem, we utilized synthetic benchmarks such as DIMACS instances. While these benchmarks provide valuable insights, future studies could benefit from applying BitEA to real-world benchmark suites such as SPEC CPU benchmarks [77], or to interference graphs generated from real C/C++ codes [78]. By comparing BitEA with widely-used allocation techniques employed in real-world compilers, such as GCC and LLVM [79], we can gain a deeper understanding of the

practical performance of the algorithm. Such an evaluation would help assess the real-world effectiveness of BitEA, providing insights into its applicability in production environments and modern compiler systems. For instance, methods like PresCount [80] have demonstrated the importance of adapting register allocation strategies to real-time scenarios, effectively balancing performance and efficiency while addressing hardware constraints.

Our findings highlight the potential of bit-based solution representations in reducing both execution time and memory usage, while offering a clear path for future research. These directions include applying BitEA to other combinatorial optimization problems such as scheduling problems [19] and communication problems [81], as well as refining the algorithmic components of BitEA for dynamic problem settings. Moreover, the compactness offered by the BitVertex representation and the computational efficiency of bitwise operations when handling complex datasets or solution spaces make BitEA a promising approach for various computational fields, including multithreading, FPGA, and GPU architectures, particularly for large-scale graph applications. BitEA may also inspire new solution approaches to the register allocation problem using different algorithmic strategies, such as approximation algorithms [82]. Finally, BitEA has the potential to revolutionize solution concepts across various domains of combinatorial optimization problems through its bit-based representation and operations.

REFERENCES

- [1] Boz, B. Süngü, G. (2020), “Integrated crossover based evolutionary algorithm for coloring vertex-weighted graphs”, *Ieee Access*, 8, 126743–126759.
- [2] Topcuoglu, H. R. Demiroz, B. Kandemir, M. (2007), “Solving the register allocation problem for embedded systems using a hybrid evolutionary algorithm”, *IEEE T. Evolut. Comput.*, 11(5), 620–634.
- [3] Sethi, R. (1973), Complete register allocation problems, in “Proc. Fifth Annu. ACM Symp. Theory Comput.”, 182–195.
- [4] Krause, P. K. (2014), “The complexity of register allocation”, *Discrete Appl. Math.*, 168, 51–59.
- [5] Quong, R. W. Chen, S.-C. (1993), Register allocation via weighted graph coloring (technical summary), Technical Report TR-EE 93-23 (232), Purdue Uni. Lib., West Lafayette, Ind., USA.
- [6] Poletto, M. Sarkar, V. (1999), “Linear scan register allocation”, *ACM Trans. Program. Lang. Syst.*, 21(5), 895–913.
- [7] Garey, M. R. Johnson, D. S. (1979), “Computers and Intractability”, Vol. 174, Freeman, San Francisco, CA, USA.
- [8] Escoffier, B. Monnot, J. Paschos, V. T. (2005), Weighted coloring: Further complexity and approximability results, in “Proc. Italian Conf. Theor. Comput. Sci.”, Springer, Siena, Italy, 205–214.
- [9] Chaitin, G. J. (1982), “Register allocation & spilling via graph coloring”, *SIG-PLAN Notices*, 17(6), 98–101.

- [10] Bouziri, H. Mellouli, K. Talbi, E.-G. (2011), “The k-coloring fitness landscape”, *J. Comb. Optim.*, 21, 306–329.
- [11] Lewis, R. M. R. (2015), “A guide to graph colouring”, Vol. 7, Springer.
- [12] Terci, G. S. Abdulhalik, E. Bayrakci, A. A. Boz, B. (2024), “Bitea: Bitvertex evolutionary algorithm to enhance performance for register allocation”, *IEEE Access*.
- [13] Jensen, T. R. Toft, B. (2011), “Graph coloring problems”, John Wiley & Sons.
- [14] Galinier, P. Hertz, A. Zufferey, N. (2008), “An adaptive memory algorithm for the k-colouring problem”, *Discrete Appl. Math.*, 156(2), 267–279.
- [15] Malaguti, E. Monaci, M. Toth, P. (2009), “Models and heuristic algorithms for a weighted vertex coloring problem”, *Journal of Heuristics*, 15, 503–526.
- [16] Cao, Y. Chen, G. Jing, G. Stiebitz, M. Toft, B. (2019), “Graph edge coloring: a survey”, *Graphs and Combinatorics*, 35, 33–66.
- [17] Gutin, G. Majumdar, D. Ordyniak, S. Wahlström, M. (2021), “Parameterized pre-coloring extension and list coloring problems”, *SIAM Journal on Discrete Mathematics*, 35(1), 575–596.
- [18] Sun, W. Hao, J.-K. Lai, X. Wu, Q. (2018), “Adaptive feasible and infeasible tabu search for weighted vertex coloring”, *Information Sciences*, 466, 203–219.
- [19] Myszkowski, P. B. (2008), Solving scheduling problems by evolutionary algorithms for graph coloring problem, in “Metaheuristics for Scheduling in Industrial and Manufacturing Applications”, Vol. 145, Springer, Berlin, Heidelberg, 167.
- [20] Mishra, A. Banerjee, S. Arbaugh, W. (2005), “Weighted coloring based channel assignment for wlans”, *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(3), 19–31.

- [21] Rossi, R. A. Ahmed, N. K. (2014), “Coloring large complex networks”, *Social Network Analysis and Mining*, 4, 1–37.
- [22] Bandh, T. Carle, G. Sanneck, H. Schmelz, L.-C. Romeikat, R. Bauer, B. (2010), Optimized network configuration parameter assignment based on graph coloring, in “2010 IEEE Network Operations and Management Symposium-NOMS 2010”, IEEE, 40–47.
- [23] Muchnick, S. (1997), “Advanced compiler design implementation”, Morgan kaufmann.
- [24] Wilken, K. Liu, J. Heffernan, M. (2000), “Optimal instruction scheduling using integer programming”, *Acm sigplan notices*, 35(5), 121–133.
- [25] Castellana, V. G. Ferrandi, F. (2013), Scheduling independent liveness analysis for register binding in high level synthesis, in “2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)”, IEEE, 1571–1574.
- [26] Probst, M. Krall, A. Scholz, B. (2002), Register liveness analysis for optimizing dynamic binary translation, in “Ninth Working Conference on Reverse Engineering, 2002. Proceedings.”, IEEE, 35–44.
- [27] Cooper, K. D. Harvey, T. J. Torczon, L. (1998), “How to build an interference graph”, *Software: Practice and Experience*, 28(4), 425–444.
- [28] Burroughs, N. (2016), “Register allocation and spilling using the expected distance heuristic”, *Software: Practice and Experience*, 46(11), 1499–1523.
- [29] Sagonas, K. Stenman, E. (2003), “Experimental evaluation and improvements to linear scan register allocation”, *Software: Practice and Experience*, 33(11), 1003–1034.
- [30] Sarkar, V. Barik, R. (2007), Extended linear scan: An alternate foundation for global register allocation, in “Compiler Construction: 16th International Con-

- ference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, 2007. Proceedings 16”, Springer, 141–155.
- [31] Mössenböck, H. Pfeiffer, M. (2002), Linear scan register allocation in the context of ssa form and register constraints, in “International Conference on Compiler Construction”, Springer, 229–246.
 - [32] Shallit, J. (1994), “Origins of the analysis of the euclidean algorithm”, *Historia Mathematica*, 21(4), 401–419.
 - [33] Kim, D.-H. Lee, H.-J. (2003), Fine-grain register allocation based on a global spill costs analysis, in “Software and Compilers for Embedded Systems: 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003. Proceedings 7”, Springer, 255–269.
 - [34] Suganuma, T. Ogasawara, T. Takeuchi, M. Yasue, T. Kawahito, M. Ishizaki, K. Komatsu, H. Nakatani, T. (2000), “Overview of the ibm java just-in-time compiler”, *IBM systems Journal*, 39(1), 175–193.
 - [35] Wimmer, C. Mössenböck, H. (2005), Optimized interval splitting in a linear scan register allocator, in “Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments”, 132–141.
 - [36] Colombet, Q. Boissinot, B. Brisk, P. Hack, S. Rastello, F. (2011), Graph-coloring and treescan register allocation using repairing, in “Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems”, 45–54.
 - [37] Briggs, P. Cooper, K. D. Torczon, L. (1994), “Improvements to graph coloring register allocation”, *ACM Trans. Program. Lang. Syst.*, 16(3), 428–455.
 - [38] George, L. Appel, A. W. (1996), “Iterated register coalescing”, *ACM Trans. Program. Lang. Syst.*, 18(3), 300–324.

- [39] Pereira, F. M. Q. (2008), A survey on register allocation, Technical report, Dept. of Comput. Sci., Univ. of California, Los Angeles, CA, USA.
- [40] Gualandi, S. Malucelli, F. (2012), “Exact solution of graph coloring problems via constraint programming and column generation”, *INFORMS J Comput.*, 24(1), 81–100.
- [41] Malaguti, E. Monaci, M. Toth, P. (2011), “An exact approach for the vertex coloring problem”, *Discrete Optim.*, 8(2), 174–190.
- [42] Furini, F. Malaguti, E. (2012), “Exact weighted vertex coloring via branch-and-price”, *Discrete Optimization*, 9(2), 130–136.
- [43] Cornaz, D. Furini, F. Malaguti, E. (2017), “Solving vertex coloring problems as maximum weight stable set problems”, *Discrete Applied Mathematics*, 217, 151–162.
- [44] Shamizi, S. Lotfi, S. (2011), Register allocation via graph coloring using an evolutionary algorithm, in “SEMCCO”, Visakhapatnam, Andhra Pradesh, India, 1–8.
- [45] Wu, J. Chang, Z. Yuan, L. Hou, Y. Gong, M. (2014), “A memetic algorithm for resource allocation problem based on node-weighted graphs [application notes]”, *IEEE Comput. Intell. M.*, 9(2), 58–69.
- [46] Sungu, G. Boz, B. (2015), An evolutionary algorithm for weighted graph coloring problem, in “Proc. Annu. Conf. Genetic Evol. Comput. (GECCO)”, Madrid, Spain, 1233–1236.
- [47] Das, D. Ahmad, S. A. Kumar, V. (2020), Deep learning-based approximate graph-coloring algorithm for register allocation, in “Proc. IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)”, 23–32.

- [48] VenkataKeerthy, S. Jain, S. Kundu, A. Aggarwal, R. Cohen, A. Upadrasta, R. (2023), Rl4real: Reinforcement learning for register allocation, in “Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction”, 133–144.
- [49] Terci, G. S. (2023), A learning-based coloring algorithm for register allocation problem, in “2023 31st Signal Processing and Communications Applications Conference (SIU)”, IEEE, 1–4.
- [50] Lintzmayer, C. N. Mulati, M. H. Silva, A. F. (2011), Register allocation with graph coloring by ant colony optimization, in “2011 30th International Conference of the Chilean Computer Science Society”, IEEE, 247–255.
- [51] Nogueira, B. Pinheiro, R. G. Subramanian, A. (2018), “A hybrid iterated local search heuristic for the maximum weight independent set problem”, *Optimization Letters*, 12, 567–583.
- [52] Nogueira, B. Tavares, E. Maciel, P. (2021), “Iterated local search with tabu search for the weighted vertex coloring problem”, *Computers & Operations Research*, 125, p. 105087.
- [53] Galinier, P. Hao, J. (1999), “Hybrid evolutionary algorithms for graph coloring”, *J. Comb. Optim.*, 3(4), 379–397.
- [54] Porumbel, D. Hao, J. Kuntz, P. (2009), Diversity control and multi-parent recombination for evolutionary graph coloring algorithms, in “European Conf. on Evolutionary Computation in Combinatorial Optimization”, Tübingen, Germany, 121–132.
- [55] Betül, B. Yıldız, T. (2021), Pool-based evolutionary algorithm for the bin packing problem, in “International Journal of Advances in Engineering and Pure Sciences”, Vol. 33, 406–414.
- [56] Korkmaz, S. (2020), Solving graph coloring problem by using an evolutionary

algorithm, M.s. thesis, Marmara Univ., Istanbul, Dept. of Comput. Eng.

- [57] Rokos, G. Gorman, G. Kelly, P. H. (2015), A fast and scalable graph coloring algorithm for multi-core and many-core architectures, in “Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Proceedings 21”, Vienna, Austria, 414–425.
- [58] Borione, A. Cardone, L. Calabrese, A. Quer, S. (2023), “An experimental evaluation of graph coloring heuristics on multi-and many-core architectures”, IEEE Access.
- [59] Osama, M. Truong, M. Yang, C. Buluç, A. Owens, J. (2019), Graph coloring on the gpu, in “2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)”, IEEE, 231–240.
- [60] Goudet, O. Grelier, C. Hao, J.-K. (2022), “A deep learning guided memetic framework for graph coloring problems”, Knowledge-Based Systems, 258, 109986.
- [61] Zheng, Z. Shi, X. He, L. Jin, H. Wei, S. Dai, H. Peng, X. (2020), “Feluca: A two-stage graph coloring algorithm with color-centric paradigm on gpu”, IEEE Transactions on Parallel and Distributed Systems, 32(1), 160–173.
- [62] Murooka, R. Ito, Y. Nakano, K. (2016), Accelerating ant colony optimization for the vertex coloring problem on the gpu, in “Proc. 2016 Fourth International Symposium on Computing and Networking (CANDAR)”, 469–475.
- [63] Zhang, K. Qiu, M. Li, L. Liu, X. (2014), Accelerating genetic algorithm for solving graph coloring problem based on cuda architecture, in “Bio-Inspired Computing-Theories and Applications: 9th International Conference, BIC-TA 2014, Wuhan, China, October 16-19, 2014. Proceedings”, Springer, 578–584.
- [64] Ma, X. Dong, M. Zhong, L. Deng, Z. (2009), Statistical power consumption analysis and modeling for gpu-based computing, in “Proc. ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)”, Vol. 1.

- [65] Geraeinejad, V. Qian, Q. Ebrahimi, M. (2024), “Investigating register cache behavior: Implications for cuda and tensor core workloads on gpus”, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 14(3), 469–482.
- [66] Fan, H. Li, M. Wu, J. Lu, W. Li, X. Yan, G. (2023), Bitcolor: Accelerating large-scale graph coloring on fpga with parallel bit-wise engines, in “Proceedings of the 52nd International Conference on Parallel Processing”, 492–502.
- [67] Brélaz, D. (1979), “New methods to color the vertices of a graph”, *Communications of the ACM*, 22(4), 251–256.
- [68] Kuon, I. Tessier, R. Rose, J. others, (2008), “Fpga architecture: Survey and challenges”, *Foundations and Trends® in Electronic Design Automation*, 2(2), 135–253.
- [69] Yue, W. Zhang, T. Jing, Z. Wu, K. Yang, Y. Yang, Z. Wu, Y. Bu, W. Zheng, K. Kang, J. others, (2024), “A scalable universal ising machine based on interaction-centric storage and compute-in-memory”, *Nature Electronics*, 7(10), 904–913.
- [70] Kole, A. Bag, A. K. Pal, A. J. De, D. (2024), “Generic model to unravel the deeper insights of viral infections: an empirical application of evolutionary graph coloring in computational network biology”, *BMC bioinformatics*, 25(1), 74.
- [71] Batarfi, O. Shawi, R. E. Fayoumi, A. G. Nouri, R. Beheshti, S.-M.-R. Barnawi, A. Sakr, S. (2015), “Large scale graph processing systems: survey and an experimental evaluation”, *Cluster Computing*, 18, 1189–1213.
- [72] Kernighan, B. Ritchie, D. M. (2017), “The C programming language”, Prentice hall.
- [73] Dillencourt, M. B. Eppstein, D. Goodrich, M. T. (2006), Choosing colors for geometric graphs via color space embeddings, in “Proc. Int. Symp. Graph Drawing”, Springer, Karlsruhe, Germany, 294–305.

- [74] Johnson, D. S. Aragon, C. R. McGeoch, L. A. Schevon, C. (1991), “Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning”, *Operations research*, 39(3), 378–406.
- [75] Lam, P. C. B. Lin, W. Gu, G. Song, Z. (2003), “Circular chromatic number and a generalization of the construction of mycielski”, *J. Comb. Theory Ser. B.*, 89(2), 195–205.
- [76] Besharati, N. Goddyn, L. Mahmoodian, E. S. Mortezaeefar, M. (2016), “On the chromatic number of latin square graphs”, *Discrete mathematics*, 339(11), 2613–2619.
- [77] Henning, J. L. (2000), “Spec cpu2000: Measuring cpu performance in the new millennium”, *Computer*, 33(7), 28–35.
- [78] Da Silva, P. Z. Mattos Senefonte, H. C. Attrot, W. (2024), “Interference graph dataset for machine learning-based register allocation”, *IEEE Access*.
- [79] Sun, C. Le, V. Zhang, Q. Su, Z. (2016), Toward understanding compiler bugs in gcc and llvm, in “Proceedings of the 25th international symposium on software testing and analysis”, 294–305.
- [80] Guan, X. Zhou, H. Bao, G. Li, H. Zhu, L. Yao, J. (2024), Prescount: Effective register allocation for bank conflict reduction, in “2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)”, 170–181.
- [81] Miri, M. Mohamedpour, K. Darmani, Y. Sarkar, M. Tummala, R. L. (2019), “An efficient resource allocation algorithm based on vertex coloring to mitigate interference among coexisting wbans”, *Comput. Netw.*, 151, 132–146.
- [82] Marappan, R. Sethumadhavan, G. Srihari, R. (2016), “New approximation algorithms for solving graph coloring problem—an experimental approach”, *Perspectives in Science*, 8, 384–387.

BIOGRAPHY

Gizem SUNGU TERCI received her B.S. and M.S. degrees in Computer Engineering from Marmara University, Türkiye, in 2015 and 2018, respectively. She is currently pursuing a Ph.D. in Computer Engineering at Gebze Technical University, Türkiye.

Since 2017, she has been working as a Research Assistant at the Institute of Information Technologies, Gebze Technical University. Her research interests include optimization problems such as graph coloring and compiler design, evolutionary algorithms, and performance acceleration.

Mrs. Sungu Terci was awarded the ACM-Women Scholarship to present her paper at the Genetic and Evolutionary Computation Conference (GECCO) in Madrid, Spain, in 2015. Additionally, she received the Best Paper Award at the 14th International Conference on Electrical and Electronics Engineering (ELECO) in Bursa, Türkiye, in 2023.

Her Ph.D. thesis work is supported by the Scientific and Technological Research Council of Türkiye (TÜBİTAK) under the 1002-B program, with project number 124E044.

PUBLICATIONS WITHIN the SCOPE of THESIS STUDY

Terci, G. S., Abdulhalik, E., Bayrakci, A. A. and Boz, B., (2024) "BitEA: BitVertex Evolutionary Algorithm to Enhance Performance for Register Allocation," in IEEE Access, vol. 12, pp. 115497-115514, doi: 10.1109/ACCESS.2024.3446596.

Sungu Terci, G., (2023), "A Learning-Based Coloring Algorithm for Register Allocation Problem," In 31st Signal Processing and Communications Applications Conference (SIU), Istanbul, Turkiye, pp. 1-4, doi: 10.1109/SIU59756.2023.10223961

