People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University of Science and Technology Houari Boumediene
Faculty of Computer Science
Department of Artificial Intelligence

Project Report

Specialization :
Intelligent Informatic Systems (SII)

Topic :
Graph Coloring Problem x BSO

Presented by :
Bensemmane Riad Yacine
Boulahlib Ali
Ragoub Ahmed Abdelouadoud

Project : G1 / Group 8

TABLE DES MATIÈRES

General Introduction **1**

1 Graph Coloring Problem **2**

1.1 Graph Coloring Problem (GCP) 2
1.2 Benchmark File Format and Instances 3
1.3 Exploratory Analysis Workflow 3

2 Exact Graph Coloring : DFS and Improvements **4**

2.1 DFS Baseline for Graph Coloring 4
2.2 Classic DFS with Backtracking 4
2.2.1 Classic DFS Pseudocode 5
2.2.2 Chromatic Number Search 5
2.3 Limitations 5
2.4 Improved Exact Coloring : Bounds and Binary Search 5
2.4.1 Algorithm Overview 6
2.4.2 Improved DFS Coloring Strategy 6
2.4.3 Pseudocode 6
2.4.4 Advantages 7
2.5 Conclusion 7

3 Bee Swarm Optimization (BSO) **8**

3.1 Introduction and Inspiration 8
3.2 Algorithm Overview 8
3.3 Pseudocode 9
3.4 Time Complexity Analysis 9
3.5 Key Concepts and Parameters 10
3.6 Application to Graph Coloring 10

4 Solution Design **11**

4.1 Problem Modeling 11
4.1.1 Graph Definition 11
4.1.2 Solution Encoding 11
4.1.3 Solution Space 11
4.1.4 Fitness Function 12

4.2 Adapting BSO to GCP 12
4.2.1 Core Loop 12
4.2.2 Operators and Data Structures 12
4.2.3 Pseudocode 13
4.2.4 Why This Adaptation Works 13
4.3 Code Algorithms 14
4.3.1 `determine_search_area()` 14
4.3.2 `inject_diversity()` 14
4.3.3 `select_new_reference()` 14
4.3.4 `local_search()` 15
4.4 Initial Performance of the Classic BSO 15
4.5 How the Original BSO Was Improved 15

5 Experimentation and Algorithm Comparison **18**
5.1 Overview 18
5.2 BSO Parameter Tuning 18
5.3 Impact of BSO Hyperparameters 18
5.4 Benchmark Graphs 19
5.5 BSO vs DFS Comparison 19
5.6 Conclusion 19

Conclusion **20**

Individual Contributions **21**

LISTE DES ALGORITHMES

1 Batch Data Preprocessing . . . 3
2 Classical DFS Graph Coloring . . . 5
3 Linear Search for Minimal Valid k . . . 5
4 Improved Exact Coloring with Bounds and Smart DFS . . . 6
5 Smart DFS Coloring with Saturation Degree . . . 7
6 Bee Swarm Optimization (BSO) . . . 9
7 Taboo List Initialization . . . 11
8 Fitness Function . . . 12
9 BSO for Graph Coloring . . . 13
10 Node Flipping Procedure . . . 14
11 Diversity Injection Procedure . . . 14
12 Reference Selection Procedure . . . 14
13 Local Search Procedure . . . 15
14 Randomized Greedy Seeding . . . 16
15 Local Search with Tabu and Random Escape . . . 16

LISTES DES FIGURES

1.1 Example of graph coloring[1]. 2

3.1 BSO workflow 10

LISTES DES TABLEAUX

4.1 Performance of Initial BSO Version 15

5.1 BSO Grid Search and Optimal Configuration for test4 18
5.2 Hyperparameter Effect Analysis for BSO 19
5.3 Graph Coloring Benchmark Instances 19
5.4 Performance Comparison : BSO vs DFS 19

GENERAL INTRODUCTION

The efficient resolution of complex, resource-intensive problems is a major challenge across scientific and industrial domains. Graph coloring, a classical NP-hard problem, involves assigning colors to graph vertices such that no two adjacent vertices share the same color, using as few colors as possible. While exact algorithms like Depth-First Search (DFS) can find optimal solutions, they become impractical for large graphs due to exponential complexity. To overcome this, metaheuristic techniques such as Bee Swarm Optimization (BSO) offer promising alternatives by mimicking the intelligent foraging behavior of bees to explore large search spaces efficiently. This project aims to develop a BSO-based graph coloring algorithm capable of producing high-quality solutions across varying graph sizes. A DFS implementation serves as a baseline, against which BSO's performance—measured in terms of solution quality, computational time, and robustness—will be compared. Particular attention is given to tuning the key parameters of BSO to optimize its effectiveness for different graph structures.

Structure

This report is organized into four chapters as follows :

- **Chapter 1 :** Introduction to the basic concepts of the graph coloring problem and description of the used benchmark.
- **Chapter 2 :** Details of the DFS with backtracking implementation for graph coloring, along with an analysis of its efficiency and limitations in the context of large graphs.
- **Chapter 3 :** Presentation of Bee Swarm Optimization applied to graph coloring, with the definition of key parameters, the implementation of the algorithm.
- **Chapter 4 :** Modeling the problem and description of the solution space, and BSO implementation for the GCP.
- **Chapter 5 :** Experimentations, parameter tuning, and comparison between the results of the DFS algorithme and the BSO algorithme

CHAPITRE 1

GRAPH COLORING PROBLEM

1.1 Graph Coloring Problem (GCP)

Let $G = (V, E)$ be an undirected graph, where V is the set of $n = |V|$ vertices, and $E \subseteq V \times V$ is the set of $m = |E|$ undirected edges. A *k-coloring* is a mapping

$$a : V \to \{1, 2, \dots, k\}$$

such that for every edge $(u, v) \in E$, we have $a(u) \neq a(v)$.

A coloring is called *proper* if no two adjacent vertices share the same color. Equivalently, each color class forms an independent set in the graph.

The *chromatic number*, denoted $\chi(G)$, is the smallest integer k for which a proper k-coloring exists :

$$\chi(G) = \min\{k : \exists \text{ proper } k\text{-coloring of } G\}.$$

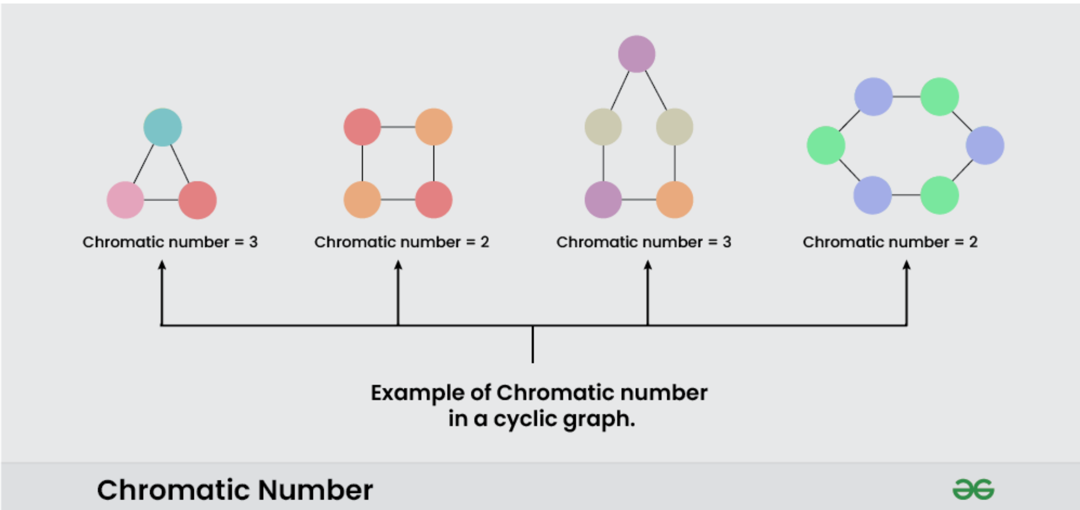

FIGURE 1.1 – Example of graph coloring[1].

1.2 Benchmark File Format and Instances

We use four DIMACS-style benchmark files. Each file begins with a header line of the form p <num_vertices> <num_edges>, specifying the number of vertices $n = |V|$ and edges $m = |E|$. For instance, the header `p 138 986` indicates a graph with 138 vertices and 986 edges. The remainder of the file lists edges, one per line, in the format `e u v`, denoting an undirected edge between vertices u and v. Since graphs are undirected, only one of (u, v) or (v, u) appears.

1.3 Exploratory Analysis Workflow

Before applying optimization, we conduct an exploratory analysis to better understand the graph structure. Each graph file is parsed by reading the header to initialize a `NetworkX Graph` [2] with n vertices, then adding edges based on the `e u v` lines. Basic statistics are computed using a `GraphExplorer` utility, including validation of node and edge counts, graph density $\rho = \frac{2m}{n(n-1)}$, average degree $\frac{1}{n} \sum_{v \in V} \deg(v)$, degree distribution, number and sizes of connected components, and global and local clustering coefficients.

We also produce visualizations : a layout of the full graph using `spring_layout`, and a random induced subgraph of 20 sampled vertices to better observe the local structure. Since dense graphs can clutter layouts, subgraph visualization provides a clearer perspective.

The entire preprocessing workflow is automated as follows :

Algorithm 1 Batch Data Preprocessing

Input : Benchmark files from `data/benchmarks/test*.txt`
for each file in sorted file list **do**
    explorer $\leftarrow$ **GraphExplorer(file)**
    explorer.**parse()**
    stats $\leftarrow$ explorer.**compute_basic_stats()**
    Append stats to summary table
    explorer.**draw_random_subgraph**(k=20, seed=42)
end for

CHAPITRE 2
EXACT GRAPH COLORING : DFS AND IMPROVEMENTS

2.1 DFS Baseline for Graph Coloring

Before introducing our metaheuristic, we first establish an exact baseline using a classical Depth-First Search (DFS) with backtracking. This serves two main purposes : (1) to compute ground-truth chromatic numbers $\chi(G)$ for small graphs, and (2) to demonstrate the limitations of brute-force methods compared to Bee Swarm Optimization (BSO) on larger instances.

2.2 Classic DFS with Backtracking

The basic DFS approach recursively assigns colors to vertices :

1. Select the next uncolored vertex.
2. Try each available color, ensuring no conflicts with neighbors.
3. Recurse ; if no valid color remains, backtrack.

Starting from $k = 1$ upwards, the smallest k that yields a valid coloring is the chromatic number.

2.2.1 Classic DFS Pseudocode

Algorithm 2 Classical DFS Graph Coloring

```
function DFSColoring(Graph G, Integer k)
    order ← vertices sorted by descending degree
    coloring ← [0] * n
    function dfs(index)
        if index == n then return True
        end if
        v ← order[index]
        for c = 1 to k do
            if c not used by neighbors of v then
                coloring[v] ← c
                if dfs(index + 1) then return True
                end if
                coloring[v] ← 0
            end if
        end for
        return False
    end function
    return dfs(0)
end function
```

2.2.2 Chromatic Number Search

Algorithm 3 Linear Search for Minimal Valid k

```
function ColorGraph(Graph G, Integer k_max)
    for k = 1 to k_max do
        if DFSColoring(G, k) then return coloring, k
        end if
    end for
end function
```

2.3 Limitations

The classical DFS method has exponential time complexity :

$$T(n, k) = O(k^n)$$

making it infeasible for large graphs. Space usage remains $O(n)$ due to the coloring array and recursion stack.

2.4 Improved Exact Coloring : Bounds and Binary Search

To overcome these limitations, we developed an enhanced exact method combining bounds estimation, binary search, and smarter DFS heuristics.

2.4.1 Algorithm Overview

- **Bounds Calculation :**
  - *Lower Bound :* Size of a maximum clique.
  - *Upper Bound :* Result of greedy coloring (ordering nodes by descending degree).
- **Binary Search :** Search for the chromatic number between lower and upper bounds, minimizing the number of DFS calls.
- **Smart DFS Coloring :** Improved node selection during DFS based on :
  - *Saturation Degree :* Prefer nodes with the highest number of differently colored neighbors.
  - *Degree-Based Tie-Breaking :* In case of ties, prefer higher degree nodes.
  - *Deterministic Tie Resolution :* Final ties broken by node ID for consistent results.

2.4.2 Improved DFS Coloring Strategy

2.4.3 Pseudocode

Algorithm 4 Improved Exact Coloring with Bounds and Smart DFS

```
function ImprovedColorGraph(Graph G)
    lower_bound ← size of largest clique in G
    upper_bound ← colors used by greedy coloring of G
    while lower_bound < upper_bound do
        k ← (lower_bound + upper_bound) // 2
        if SmartDFSColoring(G, k) then
            upper_bound ← k
        else
            lower_bound ← k + 1
        end if
    end while
    return coloring, lower_bound
end function
```

Algorithm 5 Smart DFS Coloring with Saturation Degree

```
function SmartDFSColoring(Graph G, Integer k)
    Initialize all vertices as uncolored
    while there are uncolored vertices do
        Select vertex with highest saturation degree
        Break ties by highest degree, then by smallest node ID
        for each color from 1 to k do
            if color is valid (no conflict) then
                Assign color
                if SmartDFSColoring succeeds recursively then return True
                end if
                Remove color (backtrack)
            end if
        end for
        return False
    end while
    return True
end function
```

At each DFS step :

1. Pick the uncolored node with the highest saturation degree.
2. Try each valid color (from 1 to k) not conflicting with neighbors.
3. Recursively attempt to color the rest of the graph.
4. If no valid color leads to success, backtrack.

2.4.4 Advantages

This method significantly reduces the search space and accelerates convergence, especially on dense graphs. While worst-case complexity remains exponential, smart ordering and binary search make the method practical for medium-sized instances.

2.5 Conclusion

While the classic DFS approach is simple and effective on small graphs, the enhanced method — using bounds, binary search, and smart heuristics — achieves the same exactness with dramatically better performance on larger or denser graphs.

CHAPITRE 3

BEE SWARM OPTIMIZATION (BSO)

3.1 Introduction and Inspiration

Bee Swarm Optimization (BSO) is a metaheuristic inspired by the foraging behavior of honey bees, particularly the waggle dance, which conveys the quality of food sources. Introduced by Drias et al. (2005), BSO models this collective intelligence to solve combinatorial problems [3] .

3.2 Algorithm Overview

The BSO algorithm operates as follows :

1. **Initialization :** Start with a random solution S_{ref} and add it to a taboo list.
2. **Search Area Formation :** Generate k neighbors by flipping parts of S_{ref} (controlled by the flip parameter).
3. **Local Search :** Assign each neighbor to a bee and apply local search ; store results in a dance table.
4. **Stagnation Handling :** Replace repeated solutions in the dance table with diverse ones.
5. **Update :** If improvement is found, update S_{ref} ; otherwise, switch to the most diverse solution.
6. **Termination :** Repeat until a stopping criterion is met.

3.3 Pseudocode

Algorithm 6 Bee Swarm Optimization (BSO)

Input : Termination criteria, MaxChances
$S_{\text{ref}} \leftarrow$ RandomInitialSolution()
while not terminated **do**
    Add S_{ref} to taboo list
    Generate neighbors from S_{ref}
    for each bee **do**
        Local search on assigned neighbor
        Store result in DanceTable
    end for
    if duplicates $>$ MaxChances **then**
        Replace with diverse solutions
    end if
    if improved solution found **then**
        Update S_{ref}
    else if chances remain **then**
        Decrement chances
    else
        $S_{\text{ref}} \leftarrow$ most diverse solution
        Reset chances
    end if
end while
Return : Best solution found

3.4 Time Complexity Analysis

Let n be the number of nodes in the graph, k the number of bees (i.e., neighbors), d the local search depth (`max_steps`), and I the number of iterations (`max_iter`).

- Generating k neighbors from S_{ref} : $\mathcal{O}(k \cdot n)$
- Local search per bee (on each neighbor) : $\mathcal{O}(d \cdot n)$, so $\mathcal{O}(k \cdot d \cdot n)$ total
- Dance table updates and duplicate checks : $\mathcal{O}(k^2)$
- Taboo list update and diversity management : $\mathcal{O}(k \cdot n)$

Per iteration cost : $\mathcal{O}(k \cdot d \cdot n + k^2)$
Total time complexity :

$$\mathcal{O}(I \cdot (k \cdot d \cdot n + k^2))$$

This complexity is polynomial in the number of nodes and parameters, making BSO scalable for mid-sized graph coloring instances. Performance is heavily influenced by the number of bees and the depth of local search.

3.5 Key Concepts and Parameters

- **Flip :** Degree of perturbation applied to S_{ref}.
- **Dance Table :** Stores best solutions found by bees.
- **Taboo List :** Prevents revisiting solutions.
- **Diversity :** Measured by Hamming distance to encourage exploration.

3.6 Application to Graph Coloring

In the Graph Coloring Problem, a solution assigns colors to nodes. The fitness function balances conflicts and color count :

$$\text{Fitness}(s) = \alpha \cdot \text{conflicts}(s) + \text{num\_colors}(s)$$

- **Conflicts :** Adjacent nodes sharing the same color.
- α **:** Penalty weight for conflicts.
- **Diversity :** Helps avoid premature convergence.

BSO offers a strong balance between exploration and exploitation, making it suitable for solving graph coloring efficiently.

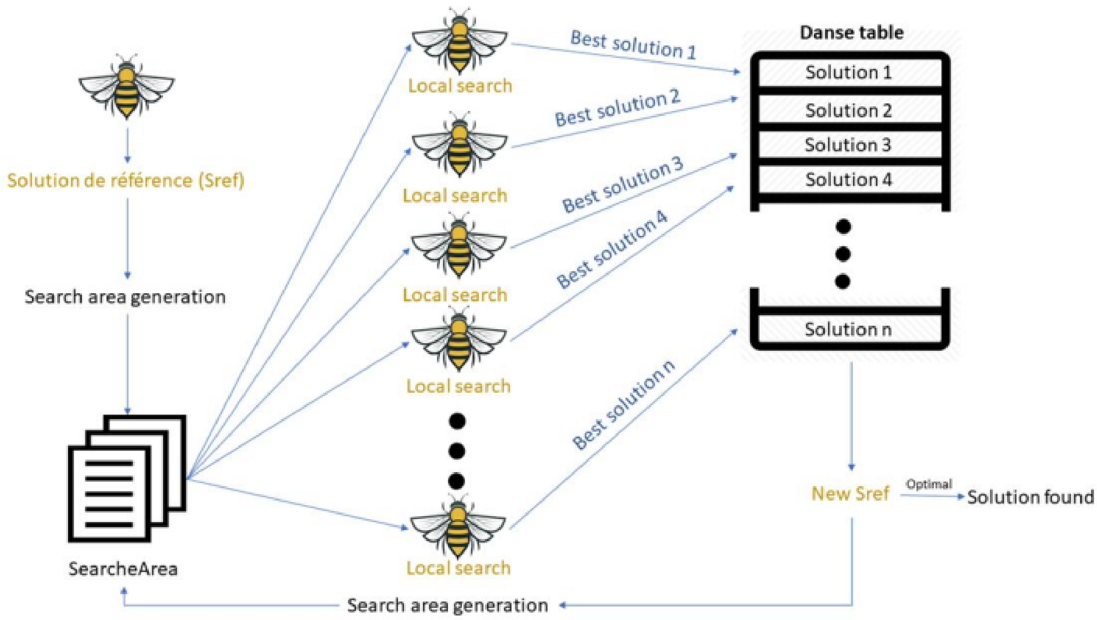

FIGURE 3.1 – BSO workflow

CHAPITRE 4

SOLUTION DESIGN

4.1 Problem Modeling

4.1.1 Graph Definition

We model the graph as $G = (V, E)$, where :

- G is represented using a `networkx.Graph` object.
- Nodes are stored as 0-indexed integers in the range $0, \ldots, n-1$.
- Edge constraints : $(u, v) \in E \Rightarrow S[u] \neq S[v]$.

4.1.2 Solution Encoding

A solution is stored as a list of integers : `List[int]`, where :

- Index i corresponds to node i.
- Values belong to $\{1, \ldots, k_{max}\}$, where 1-based colors are used.
- This representation allows $\mathcal{O}(1)$ color changes through list mutation.

4.1.3 Solution Space

The solution space is :

$$\{1, \ldots, k_{max}\}^n \quad \text{with size } k_{max}^n$$

A taboo list is used to track visited regions :

Algorithm 7 Taboo List Initialization

```
Input : None
Initialize taboo_list as an empty list  ▷ Create an empty list to store taboo solutions
taboo_list ← []                         ▷ Initialize the taboo list as an empty list of lists
```

4.1.4 Fitness Function

Algorithm 8 Fitness Function

Input : A solution S
Output : Fitness value of the solution
Initialize $\texttt{conflicts} \leftarrow 0$
for each edge (u, v) in G **do**
    if $S[u] = S[v]$ **then** ▷ Check if nodes u and v have the same color
        Increment $\texttt{conflicts}$ by 1 ▷ Conflict detected
    end if
end for
$\texttt{distinct} \leftarrow$ number of distinct colors in S ▷ Calculate the number of distinct colors in solution
Return $\alpha \times \texttt{conflicts} + \texttt{distinct}$

- $\alpha = m + 1$ penalizes conflicts heavily.
- Conflict detection uses 0-indexed positions.

4.2 Adapting BSO to GCP

4.2.1 Core Loop

1. **Initialization :** Generate random S_{ref}, initialize an empty taboo list.
2. **Main Cycle :**
   - Generate `search_area` via pattern-based flips.
   - Run local searches in parallel.
   - Inject diversity to mitigate stagnation.
   - Update S_{ref} and the chances counter.
3. **Termination :**
   - Stop if fitness $< \alpha$ (valid coloring).
   - Or, if maximum iterations are reached.

4.2.2 Operators and Data Structures

Concept	Implementation
Search Area	`determine_search_area()` with flip/h patterning
Local Search	Random walk with max single-flip steps
Diversity	Hamming distance + taboo list
Reference Update	`select_new_reference()` (quality/diversity trade-off)
Pattern Generation	Systematic flipping by index offset

4.2.3 Pseudocode

Algorithm 9 BSO for Graph Coloring

```
Input : Maximum iterations max_iter, penalty factor α, number of bees n_bees, flip size flip, max steps max_steps, and chance parameter n_chance
Initialize S_ref ← random_coloring()
Initialize taboo_list ← []
Initialize chances ← n_chance
Initialize iter ← 0
while iter < max_iter and F(S_ref) ≥ α do
    search_area ← []
    h ← 0
    while len(search_area) < n_bees and h < flip do
        s ← S_ref.copy()
        for p in range(n // flip) do
            idx ← flip * p + h
            s[idx] ← random_color_different_from(s[idx])
        end for
        search_area.append(s)
        h ← h + 1
    end while
    dance_table ← []
    for sol in search_area do
        best ← sol.copy()
        for _ in range(max_steps) do
            neighbor ← flip_one_node(best)
            if F(neighbor) < F(best) then
                best ← neighbor
            end if
        end for
        dance_table.append(best)
    end for
    inject_diversity(dance_table)
    S_ref, chances ← select_new_reference(dance_table, chances)
    taboo_list.append(S_ref)
    iter ← iter + 1
end while
Return S_ref
```

4.2.4 Why This Adaptation Works

1. **Pattern-Based Exploration :** Systematic flip/h ensures coverage of different graph regions and avoids random walk bias.

2. **Memory Mechanism :** The taboo list avoids repeated revisits; fallback diversity ensures exploration.

3. **Adaptive Intensification :** The chances counter controls the trade-off between exploitation and diversity.

4. **Efficiency :**

   - Search generation : $\mathcal{O}(n_{bees} \times n/\texttt{flip})$
   - Hamming distance : $\mathcal{O}(n)$

4.3 Code Algorithms

4.3.1 `determine_search_area()`

Algorithm 10 Node Flipping Procedure

```
Input : Solution s, flip size flip, number of nodes n
for h in range(flip) do
    for p in range(n // flip) do
        idx ← flip * p + h
        s[idx] ← random_color_different_from(s[idx])
    end for
end for
```

4.3.2 `inject_diversity()`

Algorithm 11 Diversity Injection Procedure

```
Input : Dance table dance_table, chance threshold n_chance
cnt ← Counter(tuple(s) for s in dance_table)
for sol, count in cnt.items() do
    if count > n_chance then
        replace_with_most_diverse(sol)
    end if
end for
```

4.3.3 `select_new_reference()`

Algorithm 12 Reference Selection Procedure

```
Input : Dance table dance_table, current reference ref
if exists better solution in dance_table then
    Update reference to better solution
else if retry chances remain then
    Retry with same reference
else
    Select most diverse solution as new reference
end if
```

4.3.4 `local_search()`

Algorithm 13 Local Search Procedure

```
Input : Current solution current, maximum steps max_steps
for _ in range(max_steps) do
    neighbor ← flip_one_node(current)
    if F(neighbor) < F(current) then
        current ← neighbor
    end if
end for
```

4.4 Initial Performance of the Classic BSO

The initial implementation of the BSO algorithm for graph coloring produced mixed results. It did not succeed at minimising the number of colors needed, resulting in suboptimal colorings. This highlighted the need for deeper solution refinement and better exploration mechanisms.

Table 4.1 summarizes representative results obtained from the classic BSO on small benchmark graphs.

TABLE 4.1 – Performance of Initial BSO Version

Graph	Best Number of Colors	Conflicts Found	Valid Coloring ?
text1.txt	71	0	Yes
text2.txt	295	0	Yes
text3.txt	631	0	Yes
text4.txt	638	0	Yes

4.5 How the Original BSO Was Improved

The first implementation of the BSO graph-coloring algorithm, although functional, exhibited significant limitations in both exploration efficiency and solution quality. Initial solutions were purely random, local search was naive and easily trapped in local minima, and swarm diversity was insufficiently maintained. These shortcomings often led to premature convergence to suboptimal solutions and an overall lack of robustness.

To address these issues, the second implementation introduced several critical improvements :

- **Smarter Initialization :** Instead of starting from arbitrary random colorings, the solver now uses randomized greedy seeds, providing better starting points that significantly reduce the number of conflicts from the outset.

Algorithm 14 Randomized Greedy Seeding

```
Initialize empty coloring
for each node in random order do
    Choose a feasible color minimizing conflicts (break ties randomly)
    Assign chosen color to node
end for
return seeded solution
```

- **Tabu-Enhanced Local Search :** Basic local search was replaced by a tabu-based method that avoids revisiting recent (node, color) moves and intelligently prioritizes conflict-resolving flips, allowing the algorithm to escape shallow local optima.

Algorithm 15 Local Search with Tabu and Random Escape

```
function LocalSearch(solution)
    best ← solution
    Initialize tabu_moves
    for step = 1 to max_steps do
        if no conflicts in best then return best
        end if
        Identify conflict_nodes
        improved ← False
        for node in random order from conflict_nodes do
            for available conflict-free colors do
                if move not tabu or aspiration criterion then
                    Apply move, evaluate fitness
                    if fitness improves then
                        Update best, mark move as tabu
                        improved ← True
                        break
                    end if
                end if
            end for
            if improved then break
            end if
        end for
        if not improved then
            Randomly change color of a conflict node (non-tabu move if possible)
        end if
        if not improved and random chance then
            Perform random color change on a node
        end if
    end for
    return best
end function
```

- **Balanced Swarm Generation :** The new version carefully designs the dance table by combining local perturbations, purely random samples, and heuristic (greedy) seeds, resulting in a healthier exploration-exploitation trade-off.

- **Limiting the number of colors** the use of an upper bound and a lower bound for the maximum number of colors (k_max) using **clique number** and **DSatur** algorithms for the calculations.

- **Engineering Optimizations :** The switch to efficient data structures such as NumPy arrays and cached graph structures allows for faster computation of fitness evaluations and local search iterations, improving both speed and scalability.

These improvements collectively resulted in better solution quality, more reliable convergence, and higher overall robustness when compared to the first implementation.

CHAPITRE 5

EXPERIMENTATION AND ALGORITHM COMPARISON

5.1 Overview

This chapter presents the experimental evaluation of the Bee Swarm Optimization (BSO) algorithm for graph coloring. We perform a systematic hyperparameter tuning using grid search, followed by a comparative performance analysis between BSO and a classic Depth-First Search (DFS) approach. Evaluation is carried out on synthetic and real-world graph instances.

5.2 BSO Parameter Tuning

We performed a grid search over selected hyperparameters to identify the optimal configuration for the Bee Swarm Optimization algorithm. Each setup was evaluated on `Small`, `Medium`, and `test1` instances using 3 random seeds. The goal was to minimize the number of colors used while keeping runtime acceptable.

Parameter	Tested Values	Optimal Value	Role
`n_bees`	{10, 30, 50}	50	Swarm size (parallel search scope)
`max_steps`	{5, 10, 15}	5	Local search depth
`flip`	{3, 5, 7}	3	Width of search perturbations
`n_chance`	{1, 3, 5}	5	Diversity injection threshold
`max_iter`	{100, 200, 500}	100	Maximum number of iterations

Table 5.1 – BSO Grid Search and Optimal Configuration for test4

5.3 Impact of BSO Hyperparameters

Each hyperparameter was analyzed independently around the optimal configuration to evaluate its effect on performance. The average number of colors used was the main metric, with runtime considered as a constraint.

Parameter	Impact Summary
`n_bees`	More bees improve solution diversity but slow down iteration.
`max_steps`	Deeper local search helps refine solutions but adds cost.
`flip`	Wider flips boost exploration by modifying broader patterns.
`n_chance`	Low values reduce stagnation by injecting diversity sooner.
`max_iter`	Longer runs yield better results, especially for large graphs.

Table 5.2 – Hyperparameter Effect Analysis for BSO

5.4 Benchmark Graphs

Instance	Nodes	Edges	Type
Small	20	32	Synthetic
Medium	40	65	Synthetic
`test1`	138	986	Real-world
`test2`	500	62,624	Real-world
`test3`	1000	246,708	Real-world
`test4`	900	307,350	Real-world

Table 5.3 – Graph Coloring Benchmark Instances

5.5 BSO vs DFS Comparison

We compared BSO against a standard DFS algorithm on the benchmark suite. DFS serves as a baseline heuristic, while BSO leverages stochastic search and adaptive exploration.

Instance	Colors (DFS)	Colors (BSO)	Runtime (DFS)	Runtime (BSO)
`test1`	11	**11**	0.8s	0.0156s
`test2`	58	**70**	3106s	33.2248s
`test3`	Timeout	**123**	timeout	71.2175s
`test4`	Timeout	**144**	timeout	284.2919s

Table 5.4 – Performance Comparison : BSO vs DFS

5.6 Conclusion

This experimental phase confirmed the robustness and scalability of BSO for graph coloring. Grid search tuning enabled strong generalization across diverse instances. Compared to DFS, BSO demonstrates better efficiency, adaptability, and solution quality—especially for large or irregular graphs.

CONCLUSION

In this project, we explored the application of Bee Swarm Optimization (BSO) for solving the graph coloring problem. By implementing a depth-first search (DFS) as a baseline, we established a reference point for evaluating the performance of BSO, a metaheuristic inspired by collective intelligence.

Systematic tuning of BSO's key parameters showed that, when properly configured, BSO achieves a strong balance between solution quality and computational efficiency. While DFS guarantees exact solutions, it becomes impractical for large graphs due to exponential complexity. In contrast, BSO remains scalable and effective, especially on large and dense instances.

Through experimental analysis, we highlighted the importance of swarm size, search depth, and diversity management in maintaining a balance between exploration and exploitation. Larger swarms and deeper searches improved performance, while controlling diversity prevented premature stagnation.

Overall, this work demonstrates the potential of BSO as a practical and efficient heuristic for graph coloring, particularly when exact methods are computationally infeasible. Future research could further enhance BSO by integrating hybrid strategies or adapting it to other complex combinatorial problems.

INDIVIDUAL CONTRIBUTIONS

The successful completion of this project was made possible through the collaborative efforts of all team members, each contributing to specific aspects of the work :

- **Bensemmane Riad Yacine :** Conducted research and implemented some functionalities of the algorithmes as well as data visualization. Additionally contributed to overall background research and helped consolidate theoretical foundations across all approaches.
- **Boulahlib Ali :** Focused on the implementation and fine-tuning of the Bee Swarm Optimization (BSO) algorithm. Conducted in-depth research on swarm intelligence techniques and adapted BSO to the graph coloring problem.
- **Ragoub Ahmed Abdelouadoud :** Responsible for implementing and testing the Depth-First Search (DFS) algorithm as well as improving it. Also contributed to researching exact algorithms and preparing comparative analysis.

All members participated in the writing, reviewing, and finalizing of the report and the presentation.

BIBLIOGRAPHIE

[1] GeeksforGeeks. *Introduction to Graph Coloring*. Accessed on April 25, 2025. 2024. url : https://www.geeksforgeeks.org/graph-coloring-applications/ (visité le 25/04/2025).

[2] NetworkX Developers. *NetworkX — Network Analysis in Python*. Accessed : 2025-04-26. 2024. url : https://networkx.org/.

[3] Habiba Drias, Souhila Sadeg et Safa Yahi. « Cooperative Bees Swarm for Solving the Maximum Weighted Satisfiability Problem ». In : *Computational Intelligence and Bioinspired Systems. IWANN 2005*. Sous la dir. de José Cabestany, Alberto Prieto et Francisco Sandoval. T. 3512. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2005, p. 812-819. doi : 10.1007/11494669_39. url : https://doi.org/10.1007/11494669_39.