Report of Algorithmic and Combinatorial Optimisation

Zhouji WU

M2 GENIOMHE

A Course Report

1 Background

The *k*-Graph Coloring Problem. Let $G = (V, E)$ be a simple undirected graph, where V is the set of vertices and $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ is the set of edges. Let $k \in \mathbb{N}$ be a fixed positive integer representing the number of available colors.

A *(proper) k-coloring* of G is a function

$$c : V \to \{1, 2, \ldots, k\}$$

such that

$$\forall \{u, v\} \in E, \quad c(u) \neq c(v).$$

The *k-graph coloring problem* consists in determining whether there exists a proper k-coloring of G. If such a coloring exists, the graph G is said to be *k-colorable*; otherwise, it is said to be *not k-colorable*.

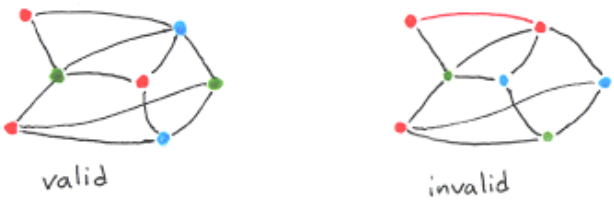

Figure 1: K Graph Coloring Problem

Metaheuristics for the *k*-Graph Coloring Problem. Let $c : V \to \{1, \ldots, k\}$ be a coloring assignment. Define the number of conflicts as

$$f(c) = \sum_{\{u,v\} \in E} \mathbb{I}\big(c(u) = c(v)\big),$$

where $\mathbb{I}(\cdot)$ is the indicator function.

The k-graph coloring problem can be equivalently formulated as the problem of finding a coloring c that minimizes $f(c)$. A coloring is proper if and only if $f(c) = 0$. The k-graph coloring problem is NP-complete for any fixed $k \geq 3$.

2 Random Instance

To evaluate the performance of the k-coloring algorithm, we randomly generated graph instances. Random graph generation ensures that the algorithm is tested on graphs with different topologies and varying

levels of sparsity or density. In this study, we adopt the **Erdős–Rényi (ER) model** for graph generation:

- **Graph generation model:** Erdős–Rényi $G(n, p)$
  - n is the number of vertices in the graph.
  - p is the probability of an edge existing between any pair of vertices, independently.
- **Generation procedure:**
  1. Specify the number of vertices n and edge probability p.
  2. For each pair of vertices (i, j), add an edge with probability p.
  3. The resulting undirected graph $G = (V, E)$ serves as the input to the algorithm.
- **Example parameters:**
  - Number of vertices: $n = 50$
  - Edge probability: $p = 0.1$ (sparse graph) or $p = 0.5$ (dense graph)
  - Number of colors: $k = 4$
- **Remarks:** The ER model allows control over graph sparsity/density by adjusting p, which in turn affects the difficulty of the coloring problem. Multiple instances can be generated for each parameter set to ensure statistically meaningful performance evaluation.
- **Python example (using NetworkX):**

```
import networkx as nx

n = 50       # number of vertices
p = 0.1      # edge probability
G = nx.erdos_renyi_graph(n, p)
```

3 Algorithm Design and Implementation

The k-coloring problem, which consists of assigning one of k colors to each vertex of a graph such that no two adjacent vertices share the same color, is known to be NP-hard. Exact methods become computationally infeasible for medium or large graphs, especially when the graph is dense or the number of colors k is close to the chromatic number. Therefore, heuristic and metaheuristic approaches are commonly employed to find approximate solutions efficiently.

3.1 Motivation for Tabu Search

Tabu Search is a metaheuristic optimization algorithm particularly well-suited for combinatorial problems such as graph coloring. Unlike simple greedy or local search methods, Tabu Search allows temporary acceptance of moves that do not immediately improve the solution, while avoiding cycles through a memory structure called the *tabu list*. This mechanism enables the algorithm to escape local minima and explore a larger region of the solution space, increasing the likelihood of finding a conflict-free coloring.

3.2 Algorithm Overview

The Tabu Search algorithm for k-coloring starts with a randomly generated initial coloring of all vertices. Let V denote the set of vertices, and $\mathcal{C} = \{0, 1, \ldots, k-1\}$ the set of colors. A coloring is represented as a mapping color : $V \to \mathcal{C}$. The quality of a coloring is evaluated by the number of *conflicting edges*, i.e., edges whose endpoints share the same color. The algorithm maintains the following components:

- **Current coloring** color and its associated number of conflicts.
- **Best coloring** best_color found so far and its number of conflicts best_conflicts.
- **Tabu list**, which records recently changed vertex-color pairs to prevent immediate reversal of moves. Each entry is associated with a *tabu tenure*, specifying how many iterations the move remains forbidden.

3.3 Fitness Function

In our implementation, the fitness of a coloring is defined as the total number of conflicting edges. Formally, for a coloring color : $V \to \{0, 1, \ldots, k-1\}$, the fitness function is

$$f(\text{color}) = \sum_{(u,v)\in E} \mathbf{1}_{\{\text{color}[u]=\text{color}[v]\}},$$

where $\mathbf{1}$ is the indicator function. A lower fitness value indicates fewer conflicts, and a conflict-free coloring corresponds to $f(\text{color}) = 0$. This function directly guides the search process and provides a quantitative measure of solution quality.

3.4 Core Iterative Search

The main Tabu Search procedure iteratively improves the coloring:

1. **Identify conflicting vertices:** The `conflicting_vertices()` function returns all vertices involved in at least one conflict. By focusing on these vertices, the search space is significantly reduced.

2. **Evaluate candidate moves:** For each conflicting vertex v and each alternative color $c \in \mathcal{C}$ (excluding its current color), the change color$[v] \to c$ is evaluated by computing the *delta in conflicts*, considering only the neighbors of v. This local evaluation is more efficient than recalculating the total number of conflicts for every candidate.

3. **Tabu check and aspiration criterion:** A move is forbidden if it is currently in the tabu list, unless it produces a coloring with fewer conflicts than the best-known solution. This aspiration criterion allows promising moves to override the tabu status.

4. **Select and apply the best move:** Among all candidate moves, the one with the largest improvement (or smallest degradation if no improving move exists) is applied. The tabu list is updated by setting the tabu tenure for the reversed move.

5. **Update best solution:** If the new coloring has a lower number of conflicts than best_conflicts, the best solution is updated accordingly.

6. **Termination:** The search continues until either a conflict-free coloring is found or the maximum number of iterations is reached.

3.5 Summary

By minimizing the fitness function and carefully navigating the search space using the tabu list and aspiration criterion, the algorithm effectively balances exploitation and exploration. The combination of local conflict evaluation, selective vertex updates, and memory-based prohibition of reversing moves allows Tabu Search to escape local minima and find high-quality colorings efficiently, even for moderately large graphs.

4 Experiments

4.1 Illustrative Instances

To evaluate the performance of the Tabu Search algorithm for the k-graph coloring problem, we conducted experiments on randomly generated graph instances using the Erdős–Rényi model. The following networks were used for the experiments:

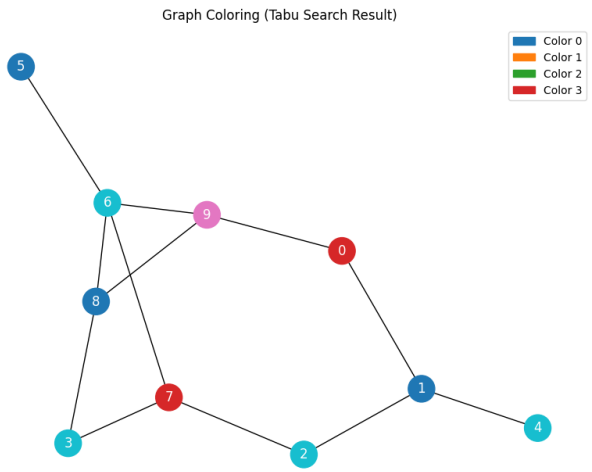

Figure 2: *Case 1*: number of vertices $n = 50$, edge probability $p = 0.1$, number of colors $k = 4$, tabu_tenure $= 5$.

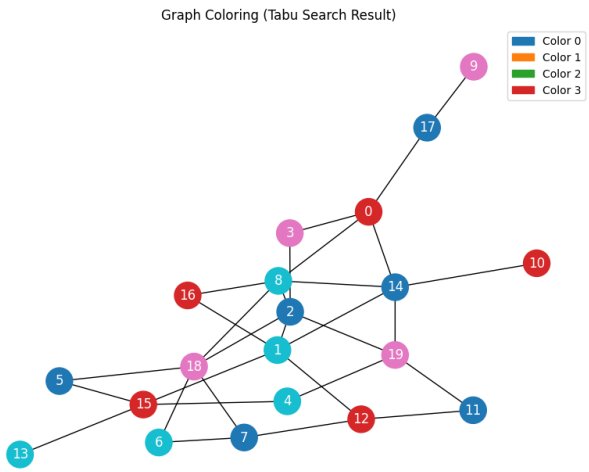

Figure 3: *Case 2*: number of vertices $n = 20$, edge probability $p = 0.15$, number of colors $k = 4$, tabu_tenure $= 5$.

The two examples above illustrate the behavior of the Tabu Search algorithm on randomly generated graphs with different sizes. In the first case, a proper k-coloring is found at iteration 1, indicating that the initial random coloring is already close to a feasible solution. The algorithm quickly detects the absence of conflicting edges and terminates immediately. This demonstrates that for small or relatively sparse graphs, Tabu Search can converge extremely fast.

In the second example, which involves a larger and more connected graph, the algorithm requires several iterations to eliminate all conflicts and finds a valid coloring at iteration 5. During the search process, conflicting vertices are progressively recolored until the fitness value (number of conflicts) reaches zero. In both cases, the final solutions contain no conflicting vertices, confirming the correctness of the algorithm.

These examples provide preliminary evidence of the effectiveness of Tabu Search in resolving conflicts efficiently and adapting to graphs of different sizes. They illustrate how the combination of local search and tabu mechanisms guides the algorithm toward a feasible coloring within a limited number of iterations.

4.2 Statistical Evaluation

[1]To evaluate the performance and robustness of the proposed Tabu Search algorithm for the k-coloring problem, a series of experiments were conducted on randomly generated graphs. For each experimental setting, multiple independent runs were performed to reduce randomness introduced by graph generation and initial color assignments. The performance was evaluated using the number of conflicts (fitness

[1]Unless otherwise specified, subsequent experimental parameters should be consistent with those of the preceding parameters.

value), success rate, and convergence behavior.

4.2.1 Evaluation 1: Replication with Fixed Parameters ($n = 50$, $p = 0.1$)

In this experiment, graphs with a fixed number of vertices ($n = 50$) and edge probability ($p = 0.1$) were generated. The algorithm was executed ($k = 4$, $tabu\_tenure = 7$) multiple times on independently generated graph instances. This setting serves as a baseline to evaluate the stability of the algorithm under fixed graph parameters. The distribution of final conflict values and the number of iterations required to reach a feasible coloring are reported.

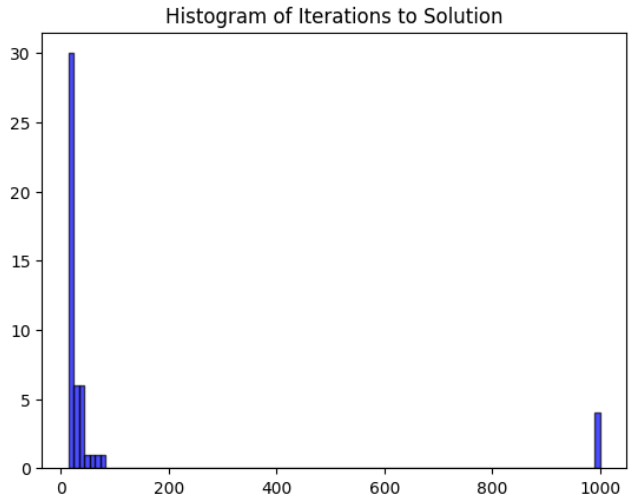

Figure 4: Number of iterations for Evaluation 1

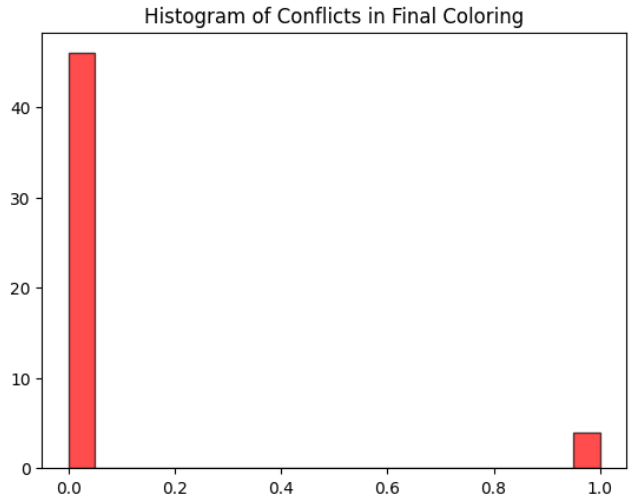

Figure 5: Number of conflicts for Evaluation 1

It's easy to see that, under this setup, tabu search can usually find a conflict-free solution within a few dozen iterations. The few exceptions (# $= 4$) are likely related to the fact that the generated graph structure is difficult to satisfy 4-coloring.

4.2.2 Effect of Graph Size (n from 20 to 100)

To study the scalability of the algorithm with respect to graph size, the number of vertices n was varied from 20 to 100 while keeping other parameters fixed. For each value of n, 30 independent graph instances were generated and solved using Tabu Search. The average fitness, success rate, and convergence speed were analyzed to assess how the algorithm's performance changes as the problem size increases.

Figure 6 illustrates the relationship between the number of nodes and the algorithm's performance. The average number of iterations exhibits an S-shaped curve: for small graphs, the iteration count remains near the lower bound (close to zero), reflecting the ease of finding a feasible coloring. As the number of nodes increases, the iterations start to rise noticeably around $n \approx 40$, and approach the maximum allowed value (1000) when n reaches approximately 80, indicating increasing difficulty in resolving conflicts for larger graphs.

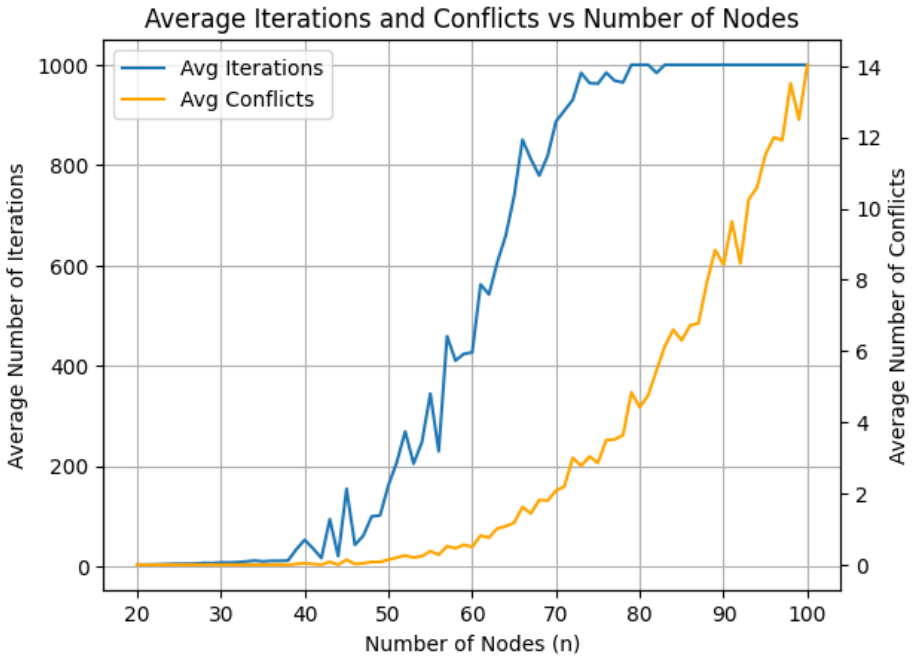

Figure 6: Effect of Graph Size on Number of Iterations and Conflicts

In contrast, the average number of conflicts appears to grow in a manner similar to a power function. For graphs with fewer than 60 nodes, the conflicts remain relatively low, but beyond $n \approx 60$, the number of conflicts rises sharply, suggesting that larger graphs not only require more iterations to converge, but also tend to have a higher inherent difficulty in achieving a conflict-free coloring. This behavior reflects the combined effects of increasing graph size and edge density on the search space complexity.

4.2.3 Effect of Edge Probability (p from 0.05 to 0.2)

This experiment investigates the impact of graph density on the performance of Tabu Search. With the number of vertices fixed at $n = 50$, the edge probability p was varied from 0.05 to 0.2 ($granularity = 0.01$). For each value of p, 30 random graph instances were generated. The results allow evaluation of how increasing graph density affects the difficulty of the coloring problem and the algorithm's ability to find feasible solutions.

When varying the edge probability p, we observed curves for average iterations and conflicts that closely resemble those obtained when varying the number of nodes n. By comparing the two settings in terms of their effect on the network's edge density (measured as the average degree per node, Figure 8), we found that the resulting values are nearly linearly overlapping. This indicates that the algorithm's performance is largely determined by the average node degree rather than n or p individually in the tested range. Since the node degree is a decisive factor in the difficulty of the k-coloring problem, this explains why experiments with different n and p produce similar iteration and conflict patterns.

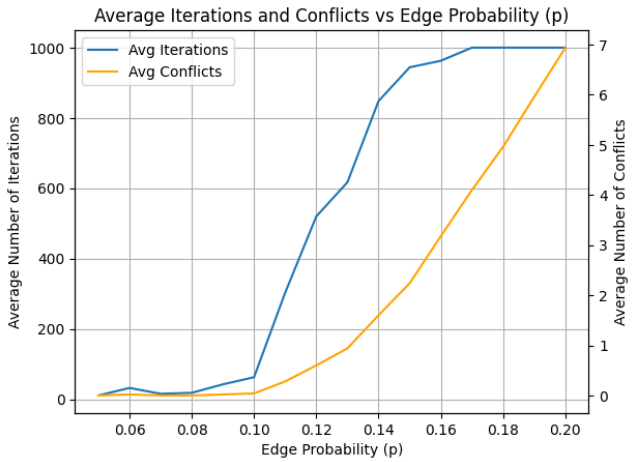

Figure 7: Effect of Edge Probability on Number of Iterations and Conflicts

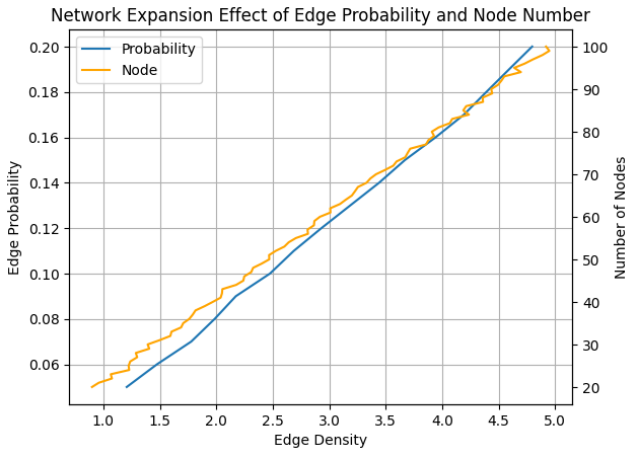

Figure 8: Edge Density Controlled by Edge Probability and Node Number

4.2.4 Effect of the Number of Colors (k from 3 to 7)

To analyze the influence of the number of available colors, the parameter k was varied from 3 to 7 while keeping $n = 50$ and $p = 0.2$ fixed. For each value of k, 30 independent runs were conducted. This experiment highlights the relationship between the number of colors and the difficulty of the coloring task, as well as the sensitivity of Tabu Search to tighter coloring constraints.

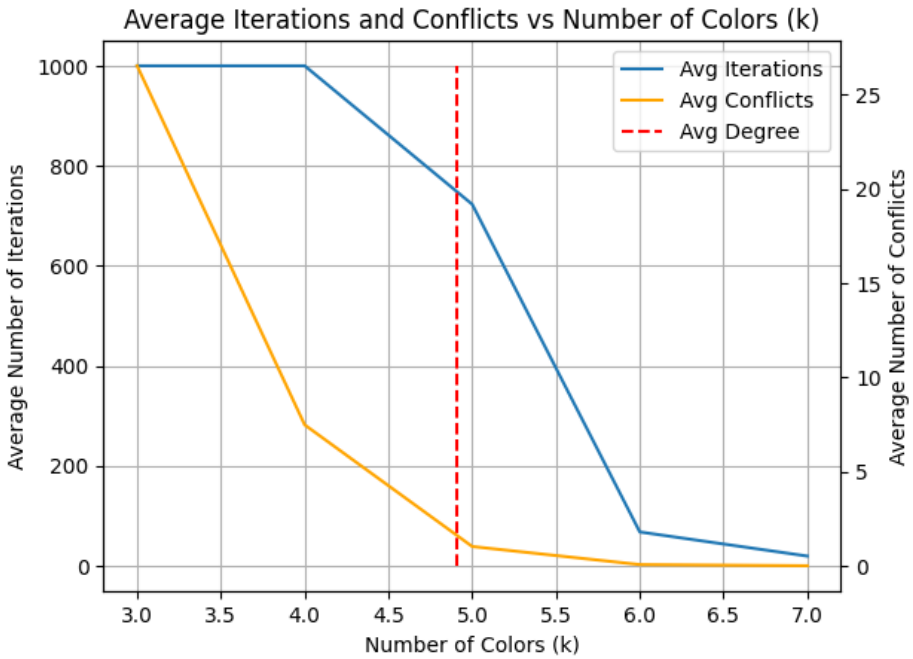

Figure 9: Effect of Number of Colors on Number of Iterations and Conflicts

Figure 9 summarizes the effect of varying the number of colors k on the algorithm's performance for ER graphs with an average degree of approximately 4.9. When $k = 3$, all runs reach the maximum number of iterations without finding a feasible coloring, indicating that the problem is too constrained for this number of colors. Starting from $k = 4$, there is a non-zero probability of finding a solution within a

limited number of iterations, and both the average number of conflicts and the required iterations decrease as k increases. By $k = 7$, the algorithm can consistently find a conflict-free coloring in relatively few iterations, reflecting that larger k values reduce the difficulty of the k-coloring problem and make feasible solutions more accessible.

4.2.5 Effect of Tabu Tenure (from 3 to 20)

Finally, the effect of the tabu tenure parameter was examined. The tabu tenure was varied from 3 to 20 while fixing the graph size ($n = 50$) and edge probability ($p = 0.14$). For each tabu tenure value, 30 independent runs were performed. This experiment aims to analyze how the length of tabu memory influences convergence behavior and solution quality, and to identify a reasonable range for this parameter.

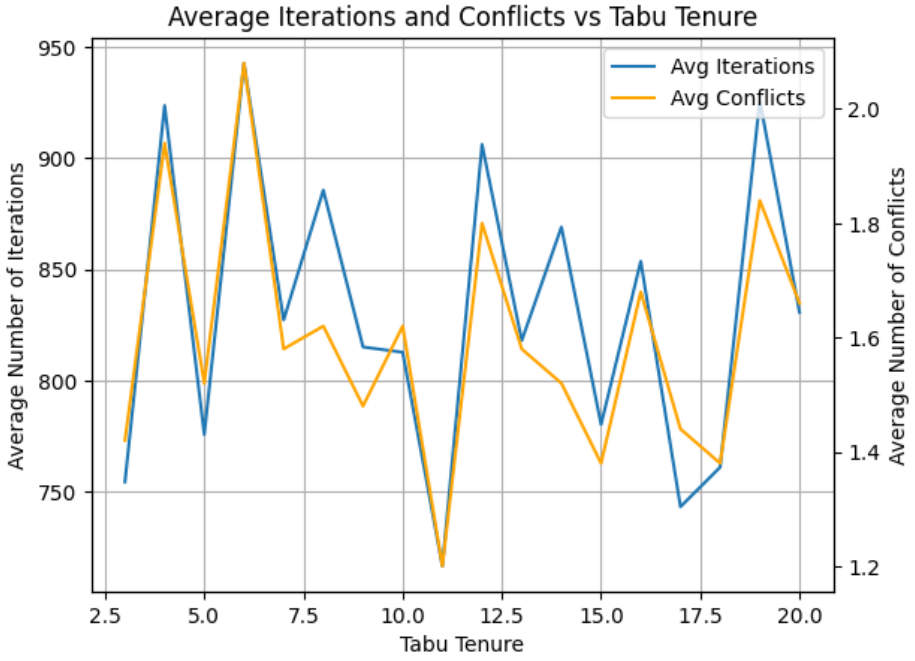

Figure 10: Effect of Tabu Tenure on Number of Iterations and Conflicts

Although no clear and consistent pattern is observed for the effect of tabu tenure on the performance of Tabu Search for the k-coloring problem, some general trends can be identified. Very small or very large values of tabu tenure tend to result in poorer performance: small tenure allows moves to be quickly reversed, which can lead to cycling around local optima, whereas excessively large tenure restricts allowable moves too much, potentially preventing the algorithm from exploring promising regions of the search space. Moderate values of tabu tenure typically yield better performance, balancing exploration and stability.

In our implementation, tabu tenure is used to determine the number of iterations during which a recently reversed move is forbidden: when a vertex changes color, assigning it back to its previous color is prohibited for the next `tabu_tenure` iterations. This mechanism prevents short-term cycling while en-

couraging exploration of new regions, which helps explain why extreme values of the parameter perform poorly.

Despite its influence, the effect of tabu tenure is generally secondary compared to the structural properties of the graph, such as the average degree or connectivity. That is, while tuning tabu tenure can have some impact on convergence speed and conflict reduction, the inherent difficulty of the k-coloring instance is primarily determined by the graph's connectivity.

A Program Code

Listing 1: Tabu Search for k-Coloring

```
import random

class TabuGraphColoring:
    def __init__(self, G, k, tabu_tenure=7, max_iter=1000, mute=False):
        """
        G: networkx.Graph
        k: number of colors
        """
        self.G = G
        self.V = list(G.nodes)
        self.k = k
        self.tabu_tenure = tabu_tenure
        self.max_iter = max_iter
        self.it = -1
        self.reached_max_iter = False
        self.mute = mute

        # random initial coloring
        self.color = {v: random.randrange(k) for v in self.V}
        self.best_color = self.color.copy()
        self.best_conflicts = self.conflicts()

        # tabu_list[(v, color)] = expire_iter
        self.tabu_list = {}

    def conflicts(self, coloring=None):
        """Calculate the number of conflicting edges"""
        if coloring is None:
            coloring = self.color
        cnt = 0
        for u, v in self.G.edges:
            if coloring[u] == coloring[v]:
                cnt += 1
        return cnt

    def conflicting_vertices(self):
        """Return all vertices involved in conflicts"""
```

```
        bad = set()
        for u, v in self.G.edges:
            if self.color[u] == self.color[v]:
                bad.add(u)
                bad.add(v)
        return list(bad)

    def search(self):
        for it in range(self.max_iter):
            current_conflicts = self.conflicts()

            if current_conflicts == 0:
                if not self.mute:
                    print(f" Found proper coloring at iteration {it}")
                self.it = it
                return self.color

            best_move = None
            best_delta = float("inf")

            conflict_vertices = self.conflicting_vertices()
            random.shuffle(conflict_vertices)

            for v in conflict_vertices:
                old_color = self.color[v]
                for c in range(self.k):
                    if c == old_color:
                        continue

                    # delta calculation: only consider neighbors of v
                    delta = 0
                    for u in self.G.neighbors(v):
                        if self.color[u] == old_color:
                            delta -= 1
                        if self.color[u] == c:
                            delta += 1

                    # tabu check
                    tabu = (v, old_color) in self.tabu_list and \
                           self.tabu_list[(v, old_color)] > it

```

```
                    # Aspiration criterion
                    if tabu and current_conflicts + delta >= self.best_conflicts:
                        continue

                    if delta < best_delta:
                        best_delta = delta
                        best_move = (v, c, old_color)

            if best_move is None:
                continue

            v, new_color, old_color = best_move
            self.color[v] = new_color
            self.tabu_list[(v, old_color)] = it + self.tabu_tenure

            new_conflicts = current_conflicts + best_delta
            if new_conflicts < self.best_conflicts:
                self.best_conflicts = new_conflicts
                self.best_color = self.color.copy()

        if not self.mute:
            print(" Reached max iterations")
        self.it = self.max_iter
        self.reached_max_iter = True
        return self.best_color
```

Listing 2: Other Utils

```
    """
    For other utils, please refer to the code files in the repository.
    https://github.com/scsewzj/tabusearch_kcolor
    """
```