COMS W4995 Project Proposal - Graph Coloring

Jacky Huang (jjh2231)

November 2025

1 Introduction

Graph Coloring is a classic NP-Complete problem involving assigning labels ("colors") to the vertices of a graph in a way such that no two adjacent vertices share the same color. Formally, given an undirected graph $G = (V, E)$, a coloring of G is an assignment c of (integer) labels to each vertex $v \in V$, subject to the constraint that for all edges $(v_1, v_2) \in E$, we have $c(v_1) \neq c(v_2)$. A k-coloring of a graph G is a coloring which uses at most k labels, and a graph is k-colorable if there exists a valid k-coloring of G. We will focus on the k-colorability problem for this project; given a graph $G = (V, E)$ and an integer k, is G k-colorable?

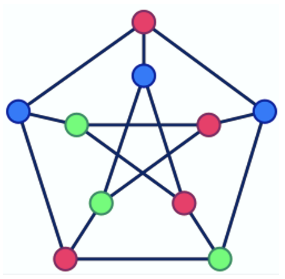

Figure 1: This is an example of a 3-coloring for the Petersen graph. Here, no two adjacent vertices share the same color. It turns out that 3 is also the smallest integer k such that the Petersen graph is k-colorable. Reference: `https://upload.wikimedia.org/wikipedia/commons/9/90/Petersen_graph_3-coloring.svg`

2 Algorithms

Note that this project will focus only on exact algorithms, as the runtime of heuristic/approximate algorithms for graph coloring are usually by default much faster than exponential time and hence is not a valid comparison of run time.

2.1 Brute Force

There is an obvious brute force algorithm which we plan to implement as a baseline; simply try out all k^n possible colorings for the n vertices in the graph, and see if any of them satisfy the constraints. Note that given a candidate coloring, it is trivial to check the legality of that coloring in polynomial time. While this is very slow in practice for even moderate sizes of n, it is a nice baseline for this project.

2.2 Pruning

One simple optimization over the baseline algorithm can be done by backtracking with pruning. During the backtracking process, we can run legality checks on the partial colorings at each node; as soon as we detect a conflict with the partial coloring at that point in the execution, we can backtrack immediately instead of continuing to explore further down the branch.

2.3 Parallelized Pruning

A natural extension of this algorithm to the parallel case is by noticing that each branch of the backtracking can be evaluated independently of each other; as a basic example, when working with the first vertex in the graph we can examine each of the k possible colorings in parallel, say with parList. The parallelism can easily be extended to the desired degree in the same manner by parallelizing further subtrees.

2.4 DSATUR-based backtracking

On the other hand, the core algorithm at play is still too slow in practice even if it is parallelized. To this end, we plan to explore further optimizations of the basic algorithm. As an NP-Complete problem, there are no known polynomial-time (exact) algorithms for graph coloring. However, there are some heuristics-based approaches to the backtracking which is shown to be much faster in practice compared to the baseline algorithm, albeit they have the same exponential worst-case complexity.

One such example is the DSATUR-based backtracking algorithm. This algorithm is based on the DSATUR heuristic, which orders the vertices by the *degree of saturation*, where the degree of saturation for a vertex v is defined as the number of distinct colors already assigned to the neighbors of v. Instead of processing the vertices in a random order, the core idea of this optimization is to explore the vertices in descending order of the DSATUR at each point in the backtracking search. Note that the DSATUR of the vertices needs to be recomputed at each stage of the backtracking algorithm; as this involves independent calculations across each of the vertices in the graph, this allows for parallelism using parList or the Par monad. This strategy can also be combined with the branch-level parallelism mentioned above.

3 Data

The first step of this project is to add built-in support for common graph families such as complete, cycle, bipartite and star graphs. These can be used to do a quick initial verification of the correctness of the algorithms, as the colorability properties of these graphs are well-known. However, to make the project more interesting, we plan to add support for custom-defined graphs. This will be done by taking as input an adjacency-list definition of a graph. We plan to create a separate program to generate random graphs for testing; the ground truth solutions for these random graphs will be obtained by running the graph through an existing library for k-colorability, such as the gcol python package: `https://gcol.readthedocs.io/en/stable/modules.html#gcol.coloring.node_k_coloring`.