Koala Graph Coloring Library: an Open Graph Coloring Library for Real-World Applications

Tomasz Dobrowolski\textsuperscript{1} Dariusz Dereniowski Łukasz Kuszner

Gdansk University of Technology
Department of Algorithms and System Modeling,
\textsuperscript{1}Tomasz.Dobrowolski@eti.pg.gda.pl

Abstract

A lot of research has been done in the field of graph coloring, yet there are no publicly released libraries available. This paper introduces such a library. This library is designed to meet several important criteria for industrial applications. Most importantly, it is designed with performance in mind. Several heuristic algorithms are implemented to deal with the NP-completeness. Further optimizations are done at the code level. The library is written in C++ with components that can be used independently. Upon completion, Koala will be released as open source, and free for educational use.

1 Introduction and related work

Graphs are common data models that appear in computer science. Coloring, among of graph related combinatorial problems, received substantial attention due to being a straightforward abstraction of partitioning objects into classes according to certain rules. The origins of the coloring problem might be seen in a letter written to W.R. Hamilton by A. de Morgan in 1852, in which the famous Four Color Theorem has its roots. Later on, chromatic properties of graphs attracted at least thousands of scientists resulting in many papers dealing with coloring algorithms, theorems, models and applications.

The main motivations for designing efficient approximate and optimal algorithms are both theoretical and practical applications of the graph coloring problem. By theoretical applications we mean various relations of the problem to the other combinatorial problems. Due to space limitations we are not able to give a comprehensive list of practical applications. The most interesting and important are: several models of task scheduling, assignment of resources (register allocation, frequency assignment), parallel computation, network design (routing problems), or load balancing. Many practical situations can be modeled by some modifications or generalizations of the classical graph coloring problem: list coloring, sum coloring, equitable coloring and others (see [16] for detailed description).

1.1 Related work

Despite over 150 years of research in the field, to the best of our knowledge, no specialized publicly released software library is being available. However, libraries providing some algorithms and data structures exist.

LEDA (Library of Efficient Data types and Algorithms) [18] is a C++ library containing data types and algorithms related to graph and network problems, geometric computation and other combinatorial problems. In the area of graph algorithms LEDA provides efficient solutions to several problems, like shortest paths, flow, minimum cut, matching spanning trees, Euler tours, graph isomorphism and drawing algorithms. LEDA is a commercial project, and the free version of the library is available, but with limited functionality.

Boost C++ Libraries [20] is a set of open source libraries designed to extend the functionality of C++. It provides several graph algorithms, including shortest paths (also all-pairs shortest paths), minimum spanning tree, connected components, flow, matching and sparse matrix reordering algorithms. A simple sequential vertex coloring algorithm [3] is available in the library. This is a heuristic algorithm without approximation ratio guarantee (for general graphs).

Another open source library with graph procedures is GOBLIN (A Graph Object Library for Network Programming Problems). Similarly to the above mentioned libraries GOBLIN implements algorithms for the standard graph theoretic problems for shortest paths, negative and minimum
mean cycles, connected components, minimum spanning trees, several flow related problems, Chinese postman problem, and matching problems.

PIGALE (Public Implementation of a Graph Algorithm Library and Editor) [7] is an open source library mainly devoted to planar graphs. In particular, several graph recognition, planarity testing, connectivity and triangulation algorithms have been implemented. Additionally, a graph editor is available as a part of the library.

Another tool is a collection of programs published by Joseph Culberson. In the current version the following implementations are available: a greedy sequential algorithms, including LF, SL, random sequence and other (for a more detailed description of sequential algorithms see Subsection 2.2); the SLF algorithm [2] and the Backtrack SLF – a modification based on dynamic reordering of vertices described in [14]; the MAXIS algorithm based on [1]; an algorithm called Iterated Greedy which, roughly speaking, calls several times a greedy algorithm mentioned earlier, but each of the calls has a new ordering of vertices depending on the previously found coloring; a tabu search algorithm, a local improvement search [12]. A detailed description of the algorithm can be found in [4, 5].

2 Graph coloring algorithms

In this section a short overview of existing graph coloring algorithms is given. Due to proven NP-completeness of the decision version of the graph coloring problem [13] there are two main approaches: polynomial approximation and exact, but exponential algorithms. We will briefly recall both cases, introducing some notations first.

2.1 Definitions and basic notations

Let $G = (V, E)$ be a simple graph with a vertex set $V$ and an edge set $E$. To color the vertices of $G$ means to give each vertex a positive integer color value in such a way that no two adjacent vertices get the same color. If at most $k$ colors are used, the result is called a $k$-coloring. In many practical considerations, it is desirable to minimize the number of used colors. The smallest possible positive integer $k$ for which there exists a $k$-coloring of $G$ is called the chromatic number $\chi(G)$. This value is bounded from above by $\Delta + 1$, where $\Delta$ denotes the maximum vertex degree of the graph.

2.2 Greedy algorithms

For a given graph $G$ and the sequence of vertices $K = (v_1, v_2, \ldots, v_n)$, we will use the term greedy coloring to describe the following procedure of color assignment:

\begin{algorithm}
\textbf{algorithm} Greedy-Color($G, K$):

\textbf{for} $v := v_1$ \textbf{to} $v_n$ \textbf{do}

\hspace{1em} give vertex $v$ the smallest possible color;

\end{algorithm}

A sequential coloring algorithm is an algorithm which determines a sequence $K$ of vertices of $G$, and then colors $G$ using the procedure Greedy-Color($G, K$).

Below we briefly recall the basic principles of the most common sequential algorithms (a more detailed analysis of sequential coloring can be found in [16, 15]). S algorithm: no assumptions are made concerning sequence $K$; LF algorithm: sequence $K$ is formed by arranging the vertices of graph $G$ in non-ascending order of degrees; SL algorithm: sequence $K$ is formed by iteratively removing a vertex of minimal degree from the graph and placing it at the end of $K$; SLF (DSATUR) algorithm: sequence $K$ is formed by dynamically arranging the vertices of graph $G$ in non-ascending order of saturation degrees, where the saturation degree is the number of pairwise different colors assigned to the neighbors of a selected vertex (ties are broken by choosing the vertex of greater degree).

The sequential algorithms discussed above are of practical significance due to the number of graph classes for which they always produce optimal or near-optimal results. Especially the last one was proven to color optimally such graph classes as: bipartite, cacti, wheels and almost all other graphs i.e. the probability that the SLF fails to obtain an exact coloring for a random graph tends to zero as a graph size tends to infinity. On the other hand, for each of described sequential algorithms it can be shown that its approximation ratio is linear. There exists an algorithm [11] with approximation ratio $O(\sqrt{\log |V|})$, but on the other hand the problem cannot be approximated within $|V|^{1-\epsilon}$ for any $\epsilon > 0$ unless P=ZPP [9].

Any solution obtained by one of the above mentioned algorithms might be further improved by methods like: tabu search, simulated annealing or other techniques. An overview of such methods might be found in [10].

2.3 Exact coloring algorithms

On the other hand there exist various techniques guaranteeing optimal solution, unhappily, utilizing exponential amount of resources. We will recall three main approaches.

Implicit enumeration methods are based on a simple backtracking rule. Having upper and lower bound on a chromatic number and a sequence of vertices, two steps: forward and backward, are performed interchangeably. For more detailed description of implicit enumeration algorithm and its research history please consult [15, 17]. Nice implementation with various improvements has been provided by Culberson in his Smallk program [6].

Maximal independent set methods use also a backtracking approach. For each maximal independent set $A$ in a
given graph $G$, a is removed from the vertex set of $G$ and then the same procedure is iterated for the resulting graph. We might note here that the best known time complexity for exact algorithm is $O(2^{\sqrt{2.4157n}})$ and has been proven for independent set approach [8].

Branch and bound algorithms use the well known technique which finds applications in designing (optimal or approximate) solutions to a variety of combinatorial problems. The examples of such approaches can be found in [19].

3 Koala graph interface

The core of Koala library is based on C++ pure virtual interfaces. We provide a set of operations that can be made on graphs to hide all of implementation details.

3.1 Generalized graph model

In Koala library, a graph is a set of triplets $(e, v_0, v_1) \in G = E \times V \times V$, where $E$ is a set of unique edge identification numbers, and $V$ is a set of unique vertex identification numbers. In every $(e, v_0, v_1)$ triplet, we will call $v_0$ a source or input vertex, and $v_1$ a destination or output vertex. Sets $V$ and $E$ contain additional special vertex VERTEX_INVALID and edge EDGE_INVALID, respectively.

This graph model is realized through an abstract interface Graph, that consist of 3 virtual functions:
- get_adjacent_for_vertex($v_0$) - get all triplets that have a given source vertex $v_0$,
- get_adjacent_for_edge($e$) - get all triplets that have an edge $e$,
- get_hyper_for_vertex($v_0$) - get set of all unique vertices $v_1$ connected to a given vertex $v_0$ by any hyper- or multi-edge.

The unique identification number for a vertex or edge is guaranteed to be in the range 0 to vertex or edge count − 1. These numbers can be directly used as indices for various additional vertex or edge data (like color, list of colors, saturation degree, etc.).

The basic graph interface is only for static graph structures. For graphs that can be modified at run-time we provide a superior abstract interface DynamicGraph, that generalize Graph class, and provides the following additional functions:
- new_edge() - add a new edge and return its unique identification number,
- new_vertex() - add a new vertex and return its unique identification number,
- link($e$, $v_0$, $v_1$) - add a unique triplet ($e$, $v_0$, $v_1$) to the graph, and automatically create new edges and vertices if necessary. Setting $e$ as EDGE_INVALID will automatically add a new edge.

The interface provides a standardized way to access and build generalized graphs. The actual data structure in memory can be hidden in various graph model implementations.

3.1.1 ListGraph representation

The basic graph model is ListGraph that implements DynamicGraph interface. ListGraph is the most flexible of all implemented graph models. It is an optimal representation for simple graphs, uses optimal memory $O(n + m)$, where $n$ is a number of vertices and $m$ is a number of edges. It is also optimal representation for hypergraphs, uses $O(n + \sum_{i=1}^{m} k_i)$ memory, where $k_i$ is a length of hyper-edge $i$ (every hyper-edge of a length $k_i$ is represented by $O(k_i)$ memory).

To show the simplicity of the presented interface, we provide a few practical graph construction examples (a resulting set of triplets is shown in a table below):

<table>
<thead>
<tr>
<th>edge</th>
<th>source</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_0$</td>
<td>$v_0$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$e_1$</td>
<td>$v_1$</td>
<td>$v_2$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$v_2$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$v_4$</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$v_5$</td>
<td>$v_5$</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$v_6$</td>
<td>$v_6$</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$v_2$</td>
<td>$-1$</td>
</tr>
</tbody>
</table>

To show how can we examine that graph further, we can call get_hyper_for_vertex($v_2$). This will return a set of all vertices connected to $v_2$ — {$v_1, v_3, v_4, v_5, v_6$}. For vertex $v_6$ calling get_hyper_for_vertex($v_6$) will return: $v_6$ — {$v_5$}. Note that $v_6$ connected to $v_2$ doesn’t exist, because the edge $e_3$ is directed. In our ListGraph implementation get_hyper_for_vertex() operation time complexity is linear in the size of the output set.
3.1.2 Procedural graphs

The unique feature of Koala library are procedural graphs. Because the Koala graph interface is purely virtual, graph can be represented as an implicit function \( G : V \rightarrow V \), thus optimizing the memory usage. Our experiments also show that they can be competitive in speed with explicit graph representations (like a list of neighbors or adjacency matrix). Until now, two procedural graphs are implemented:

- \texttt{GridGraph} - a rectangular grid graph, that can be wrapped in both directions (rectangular grid on a plane, a cylinder or a torus), uses \( O(1) \) memory for any number of vertices,
- \texttt{RandGraph} - a random graph with a controllable density, uses \( O(n) \) memory for \( n \) vertices and does not depend on the number of edges.

3.2 Coloring algorithms interface

Koala library provides a generalized abstract interface \texttt{GraphColoring} for vertex coloring problems. The interface allows to provide a coloring hints, like upper and lower bound for chromatic number, to speed-up procedure of finding a smallest possible \( k \)-coloring.

Several basic coloring algorithms are implemented. The list include: \texttt{S} algorithm (\texttt{NaiveColoring} class), \texttt{SLF} (\texttt{DSATUR}) algorithm (\texttt{DSaturColoring} class) and a simple backtracking algorithm for exact graph coloring (\texttt{BruteColoring}).

3.3 Graph visualization interface

Koala library provides a generalized interface for graph drawing routines (embedding graphs in \( R^2 \) or \( R^3 \) spaces) in a \texttt{GraphDrawing} class.

An iterative drawing routine is implemented, which iteratively finds vertex positions in \( R^3 \) for any graph by minimizing the energy of a simple potential field.

4 Conclusions and future work

The motivation for developing the Koala project comes from the fact that there are no specialized tools and libraries providing graph coloring algorithms. Since the graph coloring problems play an important role in graph theory and combinatorial optimization, and there are numerous practical applications of the coloring problems, there is a need, in our opinion, for designing and implementing a graph coloring library. The future directions are: efficiency improvements in exact graph coloring algorithms, implementation of graph class recognition and better graph visualization algorithms. Moreover, we plan to extend the Koala library by including selected optimal and approximation algorithms for other models of graph labeling, what is unlikely to find in any other available libraries.

References

[7] H. de Fraysseix and P. Ossona de Mendez. Pi.g.a.i.e - public implementation of a graph algorithm library and editor.