Algorithm Engineering
WS 20/21

Mathew Smith

# Minimum Vertex-Coloring Algorithms

## Abstract

Exact, heuristic and approximate methods of solving the minimum vertex cover problem are explored and evaluated. In particular, the quality of the approximations of the DSatur and SplitColoring algorithms are compared against the exact results of an ILP formulation. The impacts of additional algorithm engineering techniques such as pre-processing and parallelisation on the runtimes of these algorithms is also evaluated. It is found that on average DSatur provides very close to optimal colorings with very fast runtimes. The optimality and runtime of SplitColor are less favourable, but it has the advantage of providing a constant-factor approximation.

## 1 Introduction

Vertex coloring is an important problem in graph theory and computer science. On the one hand, it is a quintessential NP-complete problem; among Karp's 21 problems. On the other hand, it has wide reaching applications; many real world problems involving scheduling or avoiding conflicts can be modelled as graph coloring problems. In this report, algorithm engineering techniques for effectively computing minimum vertx colorings will be explored. There are special cases where exact polynomial algorithms exists, for example by the four-color-theorem, all planar graphs can be 4-colored, and triangle-free planar graphs can be 3-colored by Grötsch's theorem. In general however, the problem remains hard and it is this general case that will be the focus.

In section 2 the minimum vertex colorig problem will be defined, and the basic methods that will be evaluated introduced; Integer-Linear-Programming, Greedy-Coloring, and Split-Coloring. Section 3 will go into greater detail on specific implementation details of the algorithms to be evaluated, as well as introduce pre-processing techniques which will be tested. Section 4 describes the experimental setup and presents the results. Setion 5 concludes the report with discussion of the results.

## 2 Preliminaries

### 2.1 Vertex-Coloring

In the vertex-coloring problem, the goal is to find an assignment of colours to vertices such that no adjacent vertices are assigned the same colour. In the minimum vertex-coloring problem, the goal is to find such an assignment using the minimum number of unique colors. Formally, for an undirected graph $G = (V, E)$, a coloring is a function

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

for some integer $k$ such that

$$\forall (u, v) \in E : c(u) \neq c(v) \tag{2}$$

A minimum vertex-coloring then is such a function $c$ having the minimum value of $k$ for which (2) holds.

The smallest number of colours needed to color a graph is called its *chromatic number*, denoted $\chi(G)$. When certain properties of a graph are known, some upper bounds on $\chi(G)$ can be derived. The highest possible value of $\chi(G)$ is $|V|$ and is the case for all complete graphs. $\chi(G) = 1$ is only possible for edgeless graphs. A graph can always be colored with one more color than the degree of its highest degree vertex: $\chi(G) \leq \Delta(G) + 1$.

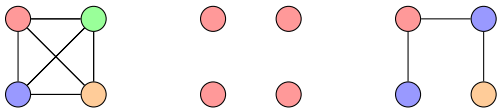


Figure 1: From left to right, complete graph: $\chi(G) = |V|$, edgeless graph: $\chi(G) = 1$, and a (non-optimal) coloring of size $\Delta(G) + 1$.

The minimum vertex-coloring problem is NP-hard. The best known exact algorithm is due to Byskov and has a runtime of $O(2.4023^n)$. The best known polynomial-time approximation algorithm is due to Halldórsson, with an approximation factor within $O(n(\log\log n)^2(\log n)^{-3})$ of $\chi(G)$. While theoretically the best available, the two mentioned algorithms are not straightforward to implement. In both cases a number of algorithms for additional esoteric problems are called upon as subroutines. In this paper, exact, approximate, and heuristic approaches will be considered which are simple and practical while delivering promising results in terms of efficiency and optimality.

#### 2.1.1 Integer Linear Programming

Integer Linear Programming (ILP) is an attractive method for finding exact solutions for minimum graph-coloring. The problem can be straightforwardly modelled with a polynomial number of constraints by appealling directly to the definition of the problem. The objective function is:

$$\min \sum_{1 \leq i \leq H} c_i \tag{3}$$

such that all vertices are assigned exactly one color;

$$\forall v \in V : \sum_{i=1}^{H} x_{v,i} = 1 \tag{4}$$

adjacent vertices are not assigned the same color;

$$\forall (u, v) \in E, i = 1, \ldots, H : x_{u,i} + x_{v,i} \leq c_i \tag{5}$$

and if a vertex is assigned a color, the variable for that color is assigned.

$$\forall v \in V, i = 1, \ldots, H : x_{v,i}, w_i \in \{0, 1\} \tag{6}$$

$H$ is an upper bound on the number of colours. Using a heuristic to provide a smaller upper bound than $|V|$ can siginficantly reduce the runtime of the solver.

#### 2.1.2 Split-Coloring

A $q$-approximate coloring can be achieved by arbitrarilly dividing the input graph into $q$ subgraphs of roughly equal size, running an exact algorithm on each subgraph, then merging the resulting color assignments. The runtime is $O(q\alpha^{\beta n/q} + n)$ where $O(\alpha^{\beta n})$ is the runtime of the exact algorithm used, and the additional linear component corresponds to the partitioning and merge steps. While the runtime is still exponential, the reduction in runtime may be significant enough to be of value in practice. Additionally, the approximation factor is constant with respect to $n$, unlike the best known polynomial-time approximation algorithm. Pseudocode for SplitColor is provided in algorithm 1.

**Algorithm 1** Split-Color Approximation Algorithm

```
function SplitColor(G = (V, E), q)
    # partition V into q subsets
    parts ← partition(V, q)
    for i in 1 to q do
        subgraph ← inducedSubgraph(part)
        solutions[i] ← ILP(subgraph)
    end for
    # merge solutions by incrementing
    # the color id's by the number of
    # colors encountered in previous
    # solutions.
    solution ← merge(solutions)
    return solution
end function
```

#### 2.1.3 Greedy-Coloring

Greedy-coloring is a popular heuristic approach for graph coloring. Vertices are considered in some order, each being assigned the color with the lowest index that has not been assigned to any of its neighbours. This approach is very fast, running in $O(|E|)$ time in its most simple form, and often yields colorings close

to $\chi(G)$. The quality of the coloring however is heavily dependent on the ordering. For example, figure 2 shows how ordering the nodes in a bipartite graph such that nodes in the same partition are sequential yields a minimal coloring, while another ordering yields a significantly worse coloring. As such, variants of greedy-coloring exist which attempt to order the vertices in a way that increases the likelihood of a quality coloring.

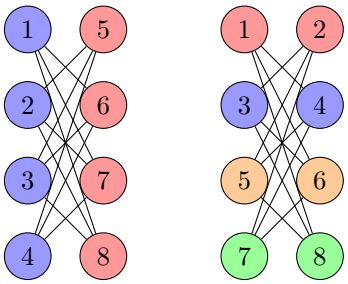



Figure 2: Two different colorings of a bipartite graph resulting from greedy-coloring with two different orderings.

## 3 Algorithms & Implementations

In this section, specifics of the algorithms to be evaluated will be presented along with relevant details on their implementations. The algorithms have been implemented in Julia v1.5[1]. The LightGraphs.jl[2] library, based on adjacency matrices, was used for graph representations.

### 3.1 ILP

The ILP formulation used exactly matches what was presented in section 2.1.1. To improve peformance, $\Delta(G)$ (the maximum degree of $G$) is provided as an upper bound on the number of colors ($H$). Preliminary tests revealed that not providing such an upper bound, and instead setting $H = |V|$, results in very long runtimes even on the smaller test instances. As such, the impact of this upper bound will not be a part of the experimental evaluation. Such an experiment would be very time consuming only to prove an obvious result. Given the simplicity of computing $\Delta(G)$, providing such an upper bound should be taken as a necessity for this ILP formulation. The ILP was formulated in the JuMP.jl[3] optimisation library, and the CBC[4] solver was used to solve the models.

### 3.2 SplitColor

SplitColor is implemented as described in section 2.1.2 and in algorithm 1. The ILP implementation described above is called to compute the exact solutions to the subgraphs.

Because the solutions to the subgraphs are computed independently, Split-Color is an ideal candidate for parallelisation. The *work* $W$ is $O(q\alpha^{\beta n/q} + n)$, and the *depth* $D$ is $O(\alpha^{\beta n/q} + n)$, resulting in a parallelised runtime of $T_p = O(\frac{q}{p}\alpha^{\beta n/q} + n)$: the maximal benefit is achieved when $p = q$, at which point $T_q = D$.

### 3.3 DSatur

DSatur is a variant of the greedy-coloring method. The name comes from the use of the *saturation degree*: the number of neighbors of a vertex that have already been colored, to decide which vertex to color next. Pseudocode for DSatur is provided in algorithm 2. In the worst case there will be $O(n^2)$ key updates to the queue (e.g. in the case of a complete graph: everytime a node is colored, the saturation degree of all not-yet colored nodes will be increased). The queue used is based on a self-balancing binary search-tree. As such, initial creation of the queue takes $O(n \log n)$ time and key updates take $O(\log n)$ time resulting in an overall runtime in $O(n \log n + n^2 \log n)$. While in general DSatur might produce arbitrarily bad (while still being valid colorigs) solutions, it has been shown to produce exact solutions for bipartite, cycle, and wheel graphs, and has been shown empirically to produce significantly better colorings than greedy-coloring with arbitrary orderings on average.

[1] https://julialang.org/
[2] https://github.com/JuliaGraphs/LightGraphs.jl
[3] https://github.com/jump-dev/JuMP.jl
[4] https://www.coin-or.org/Cbc/

**Algorithm 2** DSatur Heuristic Algorithm

```
function DSatur(G = (V, E))
    # queue implemented as a max-heap.
    queue ← {(v, 0) : v ∈ V}
    while queue is not empty do
        v ← dequeue(queue)
        # select the minimum color not
        # assigned to v's neighbors.
        v.color ← firstAvailableColor(v)
        for u ∈ v.neighbors do
            # increment the saturation degree
            # of not-yet colored neighbors.
            queue[u] += 1
        end for
    end while
end function
```

### 3.4 Pre-Processing

A vertex $v$ *dominates* $u$ if the neighborhood of $v$ is a superset of $u$ (see figure 3). If this is the case, $u$ can be removed from the graph before coloring, then assigned the same coloring as $v$. Searching for dominated nodes can be achieved in $O(n^3)$ time. Pseudocode for finding dominated vertices is provided in algorithm 3. With the set of dominated vertices in hand, pre-processing is completed by removing those vertices from the graph, and creating a mapping from dominated to dominating vertices. The mapping can then be used to derive the coloring for the original graph from that of the reduced graph. Reducing the size of the input graph by removing dominated vertices will be evaluated as a pre-processing step to reduce the runtime of the ILP.

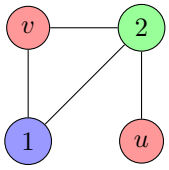



Figure 3: $v$ dominates $u$: $v$'s neighbors are a superset of $u$'s.

**Algorithm 3** Algorithm to find dominated vertices for pre-processing

```
function DominatedVertices(G = (V, E))
    dominated ← ∅
    for u, v ∈ V × V do
        if u.neighbors ⊂ v.neighbors then
            dominated[u] ← v
            for w, x ∈ dominated do
                # v takes over any vertices
                # that u dominated.
                if x = u then
                    dominated[w] ← v
                end if
            end for
        end if
    end for
    return dominated
end function
```

## 4 Experimental Evaluation

The goal of the following experiments will be to answer the following questions:

(Q1) How do the colorings yielded by the DSatur heuristic and Split-Color approximation compare to the optimal colorings provided by the ILP?

(Q2) What improvement in runtime is gained from pre-processing before running the ILP?

(Q3) What improvement in runtime is gained from parallelising Split-Color?

(Q4) What impact do different graph properties have on runtime and the size of the coloring? In particular it is expected that density, maximum degree, and the size and frequency of cliques are of importance to vertex coloring.

*To answers the above questions, all methods will be run on the same set of randomly generated graphs (described in the following subsection). ILP will be run with and without pre-processing to answers (Q2). The pre-processed version will be labeled ILP-P. SplitColor will be run with* $q = 2, 4, 8$*, and with*

$p = 1, 2, 4, 8$ *to answers (Q3). Variants will be labelled SplitColor-q-p accordingly. Statistical analysis on the results with respect to certain graph properties will be run on the results of the ILP to answers (Q4).*

A timelimit of 10 minutes will be applied. If an algorithm fails to find a solution for an instance in under 10 minutes, that attempt will be labelled as failed.

### 4.1 Data and Hardware

The graphs used for testing were randomly generated using the *expected degree model*. This model takes as input a vector $\omega$ of length $|V|$, with $\omega_v$ corresponding to the expected degree of vertex $v$. For each graph, $\omega$ was generated by sampling a normal distribution for a specific value of $\mu$, and $\sigma = \mu/2$. This way, graphs were generated with a predictable average degree, which is an important indicator for the vertex coloring problem, while having sufficient variations between vertices and graphs to allow the algorithms to encounter a variety of structures. Graphs are grouped into sets based on the number of nodes, and subsets based on their expected degrees. For example, expdeg(36, 3) is the set of graphs with 36 nodes and expected degree of 3. Each such subset consists of 10 randomly generated graphs. Unless specified otherwise, runtimes and coloring-sizes will be averages across a subgroup.

The experiments were run on an Intel i5-8265u CPU at 1.6GHz with 4 cores, 8 threads, and 16GB of RAM.

### 4.2 Results

As expected, the runtime of the approximation and heuristic algorithms were negligible compared to those of the ILP, where the runtime blew up for the larger graph sets. For 5 graphs in expdeg(72, 12), and 7 graphs in expdeg(72, 24), the ILP failed to find a solution in under 10 minutes. As such, the corresponding average runtimes in table 1 are undervalued as they do not account for the failed attempts. The runtime of SplitColor-2 started to rise significantly on the largest graph set. The negligible runtime of SplitColor-8 on the same set demonstrates the significant reduction in runtime gained by partitioning.

The size of the colorings produced by DSatur were *very* close to the exact colorings. On average, the difference between the exact coloring and that provided by DSatur was less than 1. The SplitColor approximations were much larger, although it can be seen in figure 4 that in practice the approximation factor appears to grow sublinearly with $q$.

The experiments showed that the benefits of parallelising SplitColor are not emphatic. Figure 5, which visualises runtimes for $q = 8$ and $p = 1, 2, 4, 8$, shows that performance is volatile across the varyious graph sets. The 8 core variant is more consistent in its runtime across graph sets, but there is no clear speed advantage. Perhaps larger test instances are necessary in order converge on a clear benefit from parallelisation.

The results of the preprocessing method were promising. ILP-Preprocessed was compared against ILP for graph sets expdeg(72, 6) and expdeg(72, 12). As can be seen in figure 6, notable decreases in runtime were achieved. On average, preprocessing was able to reduce the size of the input graph by 9 vertices.

Running a linear regression model on the results with the runtimes of ILP as the independent variable and the graph properties of density, maximum degree, and maximum clique, and number of colors as dependents revealed unsurprisingly that graph density has a significant positive correlation with runtime. More interesting is that maximum degree had a slightly *negative* coefficient of -5, while having an the same $r^2$ of 0.24 as density, suggesting that the impact on runtime is noteworthy. This negaive coefficient appears to be counter to intuition: a higher degree graph is likely to require a larger coloring (though not necessarily, say in the case of a bipartite graph), and it seems reasonable to expect that a larger coloring would require more time. An explanation could be that when a high degree vertex is colored, that color is ruled out for a large number of vertices, making the subsequent coloring devision for those vertices slightly easier. This is further verified by the negative coefficient for colors. Further justification for this might be that, once a large number of

Table 1: Summary of runtimes in seconds. * indicates that some instances in the subgroup exceeded the 600 second timelimit; the average runtimes in such cases might be negatively skewed as larger runtimes were not counted.

| **graph set** | ILP | ILP-P | SplitColor-2 | Split-Color-4 | Split-Color-8 | DSatur |
|---|---|---|---|---|---|---|
| expdeg(36, 3) | 0.08 | | 0.03 | 0.05 | 0.06 | 0.001 |
| expdeg(36, 6) | 1.13 | | 0.06 | 0.10 | 0.04 | 0.002 |
| expdeg(36, 12) | 7.23 | | 0.13 | 0.10 | 0.06 | 0.002 |
| expdeg(72, 6) | 6.00 | | 0.10 | 0.07 | 0.07 | 0.003 |
| expdeg(72, 12) | 202* | | 1.50 | 0.15 | 0.17 | 0.007 |
| expdeg(72, 24) | 155* | | 40.0 | 0.33 | 0.26 | 0.016 |

Figure 4: Average number of colors per graph set for each method

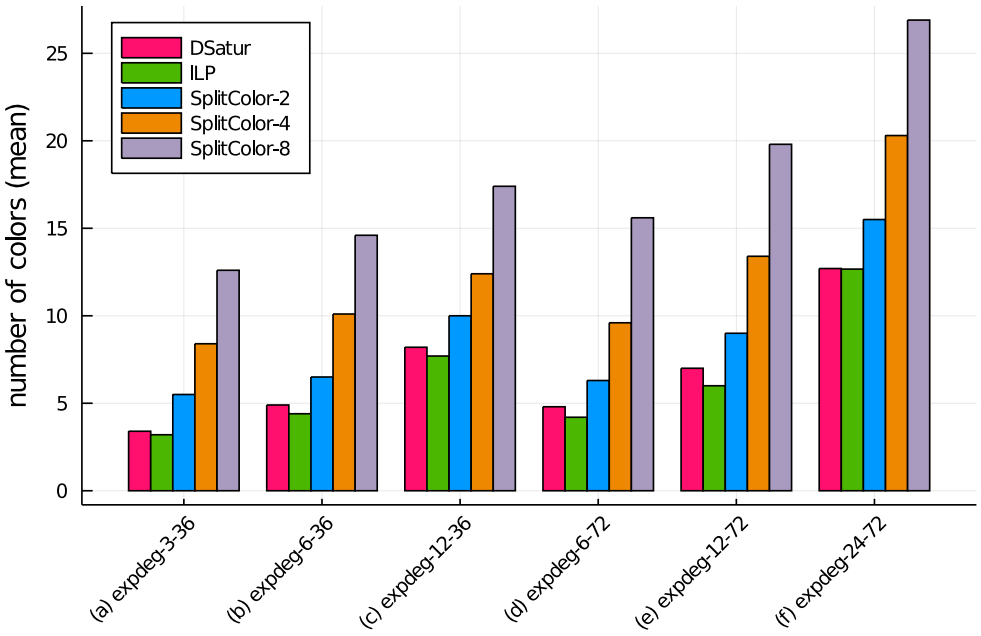


Figure 5: Comparison of runtimes of SplitColor-8 with different levels of parallelism. SplitColor-8-$p$ corresponds to running with $p$ cores.

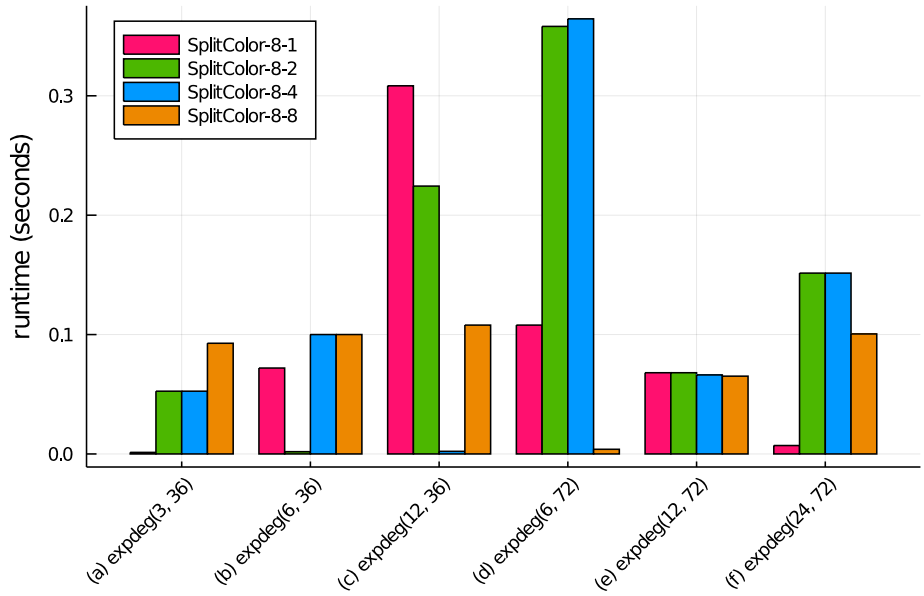
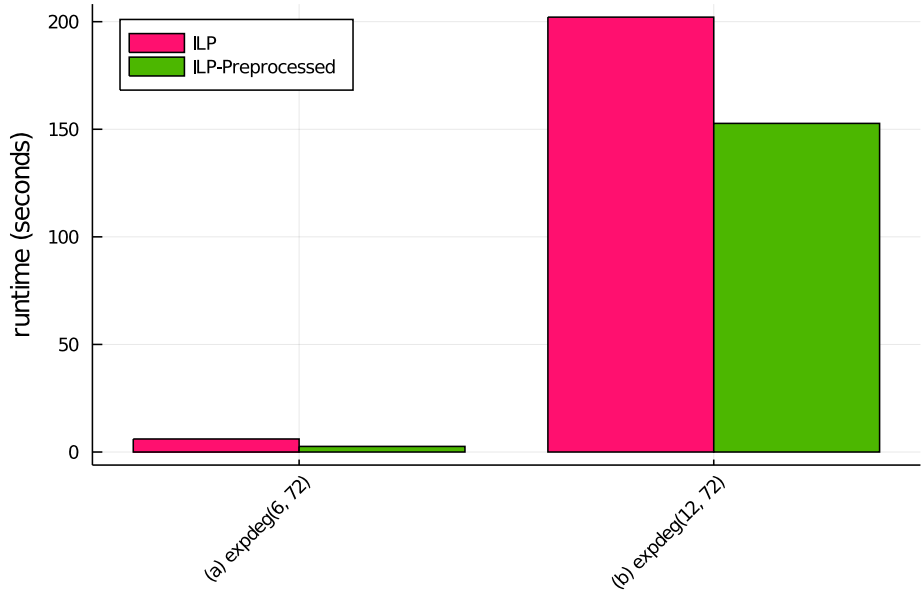



Figure 6: Comparison of runtimes of ILP with and without preprocessing.



colors are already in play, there are more degrees of freedom for coloring new vertices without increasing the coloring size, making decisions easier. Table 2 summarises these results.

Table 2: Key results of a linear regression model on graph properties with runtime as the independent variable.

| property | coefficient | $r^2$ |
|---|---|---|
| density | 2312 | 0.24 |
| max degree | -5 | 0.24 |
| max clique | -89 | -0.04 |
| colors | -37 | 0.12 |

## 5 Discussion and Conclusion

Overall, the experiments have shown that it is possible to compute both exact and approximate minimum vertex colorings using uncomplicated methods. Runtimes of the exact ILP will likely become infeasible on large graphs, but the preprocessing technique of removing dominated vertices showed promise, and surely preprocessing steps can be taken to reduce runtime. Additionally, methods of reducing the number of constraints could be explored. The performance of DSatur was extremely impressive. Runtimes are negligible while the resulting colorings were very close to optimal. Theoretically it could produce an arbitrarily bad coloring, however across the 60 graphs tested it remained very close to optimal, demonstrating that such a bad solution is unlikely. Comparitavely the performance of SplitColor was not compelling. The approximations were notably worse than DSatur. The main appeal of SplitColor will be in cases where a fixed upper bound on the quality of the solution is absolutely necessary.