On K-Colouring

Eric Kjeldergaärd
Department of Computer Science
University of North Dakota
Grand Forks, ND 58202
kjelderg@gmail.com

Kristopher Zarns
Department of Computer Science
University of North Dakota
Grand Forks, ND 58202
kzarns@gmail.com

Abstract

Determining the correct number of colours required to fill in a graph such that no two connected vertices are of the same colour is a hard problem. The elements of the problem are the vertices, edges, and the particular arrangement therein. The variance of any of these three elements can have a profound impact on the difficulty of finding a solution using a particular algorithm.

This paper investigates three of the common algorithms that solve the vertex colouring problem along with some heuristics applied to these algorithms. Both mathematical run-time analysis and empirical data compare the relative performance and quality of these algorithms and heuristics.

The empirical results supported the mathematical analysis. There are much faster algorithms for solving the problem of k-colouring than the brute force approach. However, within the critical region of the problem space, no algorithm could significantly surpass the brute force approach.

1 Introduction

Colouring a map seems at first like such a simple problem. There are various areas placed on the map, states, nations, and districts for instance, and to make the borders quite obvious, the various areas are often coloured such that no area shares a colour with its neighbour. For some very practical reasons, the number of colours that are to be used is a small and finite number.

So the problem changes from being quite simple to quite hard. There is now a map and only a few colours available with which to colour the map. The goal is still the same, to colour each individual region such that it has no same-coloured neighbours, but the answer is now of the form. "Yes, it is possible." or "No, more colours would be needed."

The division in the problem space between where the problems are easy to where they are hard is called the **critical region** or **phase transition**.

> It has been shown that graph colouring problems exhibit a phase transition, where problems change from being easy to colour, to being hard to colour, and on to problems that obviously cannot be coloured. This easy-hard-easy phase transition occurs at a critical value of connectivity. [5]

The computing revolution has brought with it a number of convenient abstractions. Applying a few of these will make it easier to discuss the math behind determining whether or not a map can be coloured as described with some number of colours, k, and determining the smallest satisfactory value of k.

Any finite system of colours can be represented numerically in some fashion. This fact implies that k-colouring is the same as k-numbering. This will come up in the particular implementations of a number of algorithms. Because modern computing is inherently tied to numerical representations, it is more convenient to discuss colours as numbers.

A map, for the purposes of this problem, is a set of regions and their shared boundaries. The particular characteristics of each region do not matter except which other regions it contacts. To simplify representation, then, each region will be replaced by a simple consistent item, a **vertex**. The connections between these vertices must also be represented. Each place where one region contacts another can similarly be replaced by a simple connection between the two corresponding vertices. This connection will be called an **edge**. To complete the transition from a useful and practical map to an easy to conceptualise and abstract representation, the simplified representation will be called a **simple graph**.

A graph, G, is mathematically represented as a set of vertices, V, and a set of edges, E. That is $G = (V, E)$. The set V is simply a set of elements representing the vertices themselves, which may be represented simply as numbers or more abstractly as $V = (v_0, v_1, \cdots, v_n)$ for some number, n, vertices. Edges, E, connect one vertex to another. E may then

be represented as a set of doubles $E = (\{u_0, v_0\}, \{u_1, v_1\}, \cdots, \{u_m, v_m\})$ for m edges. Note that for each double, $\{u, v\}$, that $u, v \in V$ and that order does not matter, such that $\{u, v\} = \{v, u\}$.

2 A Small Practical Example

For a simple and concrete example, consider Australia.

Figure 1: An uncoloured map of Australia [4]

Australia has 6 states and two territories in all. The regions are, in rough clockwise contact order from twelve o'clock at the geographical centre, Northern Territory (NT), Queensland (QLD), New South Wales (NSW), South Australia (SA), Australian Capital Territory (ACT), Victoria (VIC), Tasmania (TAS), and Western Australia (WA). Can this map be coloured with 4 colours? 3? 2?

As described above, a simple graph can be created to represent this problem.

$$
\begin{aligned}
&G = (V, E) \\
&V = (NT, QLD, NSW, SA, ACT, VIC, TAS, WA) \\
&E = (\{NT, QLD\}, \{NT, SA\}, \{NT, WA\}, \{QLD, NSW\}, \\
&\quad \{QLD, SA\}, \{NSW, SA\}, \{NSW, ACT\}, \{NSW, VIC\}, \\
&\quad \{SA, VIC\}, \{SA, WA\})
\end{aligned}
$$

Inspection of the graph shows a fair number of edges that have SA as a vertex. Similarly, the map shows that South Australia shares a border with many (5) of the other regions. This seems like a good place to start, the vertex with the highest vertex degree, or number of edges. Let us colour this vertex colour 0. A modified representation of the edges would help to verify whether there are any adjacent vertices with the same colours.

$$
\begin{aligned}
SA &= 0 \\
E &= (\{NT, QLD\}, \{NT, 0\}, \{NT, WA\}, \{QLD, NSW\}, \\
&\quad \{QLD, 0\}, \{NSW, 0\}, \{NSW, ACT\}, \{NSW, VIC\}, \\
&\quad \{0, VIC\}, \{0, WA\})
\end{aligned}
$$

Having chosen a colour for the first vertex, it seems a second vertex colour would be a logical next step. New South Wales is the next highest vertex degree connected to our first vertex. Giving this vertex colour 1 yields a list of edges as follows.

$$
\begin{aligned}
NSW &= 1 \\
E &= (\{NT, QLD\}, \{NT, 0\}, \{NT, WA\}, \{QLD, 1\}, \\
&\quad \{QLD, 0\}, \{1, 0\}, \{1, ACT\}, \{1, VIC\}, \\
&\quad \{0, VIC\}, \{0, WA\})
\end{aligned}
$$

And so it proceeds until the maximum number of colours is reached or until all vertexes are coloured.

$$
\begin{aligned}
QLD &= 2 \\
E &= (\{NT, 2\}, \{NT, 0\}, \{NT, WA\}, \{2, 1\}, \\
&\quad \{2, 0\}, \{1, 0\}, \{1, ACT\}, \{1, VIC\}, \\
&\quad \{0, VIC\}, \{0, WA\}) \\
WA &= 1 \\
E &= (\{NT, 2\}, \{NT, 0\}, \{NT, 1\}, \{2, 1\}, \\
&\quad \{2, 0\}, \{1, 0\}, \{1, ACT\}, \{1, VIC\}, \\
&\quad \{0, VIC\}, \{0, 1\})
\end{aligned}
$$

Thus far only 3 colours have been used in the colouring of the graph. Looking at the next region (NT) with a vertex degree of 3 puts an end to 3 colours and introduces a need for a fourth colour. This is because each of the three edges from the vertex connects NT to a vertex of a different colour. There is nothing unexpected after this, so below is the completion of this particular graph.

$$
\begin{aligned}
NT &= 3 \\
&\quad E = (\{3,2\},\{3,0\},\{3,1\},\{2,1\}, \\
&\qquad \{2,0\},\{1,0\},\{1,ACT\},\{1,VIC\}, \\
&\qquad \{0,VIC\},\{0,1\}) \\
VIC &= 2 \\
&\quad E = (\{3,2\},\{3,0\},\{3,1\},\{2,1\}, \\
&\qquad \{2,0\},\{1,0\},\{1,ACT\},\{1,2\}, \\
&\qquad \{0,2\},\{0,1\}) \\
ACT &= 0 \\
&\quad E = (\{3,2\},\{3,0\},\{3,1\},\{2,1\}, \\
&\qquad \{2,0\},\{1,0\},\{1,0\},\{1,2\}, \\
&\qquad \{0,2\},\{0,1\})
\end{aligned}
$$

The edges above show the graph entirely coloured using 4 colours. Now, being intelligent creatures, it is possible to see that a poor choice was made and that the graph could have been coloured in 3 colours instead. Rewind to the decision of $WA = 1$. Instead, $WA = 2$ could have been chosen.

$$
\begin{aligned}
WA &= 2 \\
&\quad E = (\{NT,2\},\{NT,0\},\{NT,2\},\{2,1\}, \\
&\qquad \{2,0\},\{1,0\},\{1,ACT\},\{1,VIC\}, \\
&\qquad \{0,VIC\},\{0,2\})
\end{aligned}
$$

With that modification, choosing NT from the original set of 3 colours is possible and the simple colouring proceeds just fine.

$$
\begin{aligned}
NT &= 1 \\
&\quad E = (\{1,2\},\{1,0\},\{1,2\},\{2,1\}, \\
&\qquad \{2,0\},\{1,0\},\{1,ACT\},\{1,VIC\}, \\
&\qquad \{0,VIC\},\{0,2\}) \\
VIC &= 2 \\
&\quad E = (\{1,2\},\{1,0\},\{1,2\},\{2,1\},
\end{aligned}
$$

$$
\begin{aligned}
&\{2,0\},\{1,0\},\{1,ACT\},\{1,2\},\\
&\{0,2\},\{0,2\})\\
ACT &= 0\\
E &= (\{1,2\},\{1,0\},\{1,2\},\{2,1\},\\
&\{2,0\},\{1,0\},\{1,0\},\{1,2\},\\
&\{0,2\},\{0,2\})
\end{aligned}
$$

This example shows that the particular choices of colouring matters quite a bit. Before delving into the intricacies of determining colouring in an automated way, there are a few interesting things to note.

3 Special Cases

Not all problems are created equal and there are a few interesting special cases when colouring a graph. The most interesting special graphs are completely connected graphs, completely unconnected graphs, and bipartite graphs.

On a completely connected graph, every vertex has an edge to every other vertex. For this case, the particular colouring does not matter. For n vertices, the graph would require n colours. This forms the upper bound for this sort of problem. It can never require more than a unique colour for every vertex of the graph.

In contrast, a completely unconnected graph can be coloured with any number of colours and since there are no edges, there can be no inconveniently coloured adjacent neighbours. This forms the lower bound for the vertex colouring problems. This particular graph is the only type of graph that can be coloured with only one colour.

The third type of fairly interesting special graph for vertex colouring is the bipartite graph. A bipartite graph could be separated into two groups such that there are no edges except those which connect vertices of opposite groups. To state the edges another way, there are no edges connecting a vertex from a particular group to another vertex of that same group. If every vertex in one group were coloured a particular colour, then ever vertex in the other group can be coloured in a second colour and it therefore requires only two colours.

For any of these special cases, it is not difficult to determine whether or not the graph can be coloured given only the graph and a number of colours.

4 General Cases

There are a few factors that can dictate the difficulty of a k-colouring problem generally. These factors are the size of n, the size of m, the size of k and the ratios of those sizes. In particular, [[6]] shows that any time that k is relatively small, that being smaller than $n^{\frac{1}{2}}$, there are non-deterministic algorithms that produce efficient optimal colourings for "nearly all" graphs.

5 The Relationship Between K-Colouring Decision and K-Colouring Optimization

This paper is intended to discuss the optimization problem of k-colouring. There is a closely related decision problem. The optimisation problem determines the smallest number of colours, k, given a graph G for which G can be coloured such that no vertices that share an edge have the same colour. While the decision version instead decides, given k and G if G can be coloured in such a way with k colours.

Solving the optimisation problem results in some number of colours, k. One simple method for solving the decision problem for k-colouring given an algorithm that solves the optimisation problem is to create a function something like function 1.

function 1 DecideGivenOptimizationAlgorithm(G, k)

Input: G {a graph}
Input: k {the number of colours}
Output: whether the graph can be coloured in k colours
  $s \leftarrow OptimizationAlgorithm(G)$
  return $s \leq k$

Function 1 itself is $O(1)$ comparisons. Thus, if an efficient algorithm were known for the k-colouring optimisation problem, it would also be efficient to sole the k-colouring decision problem. Now, what of k-colouring optimisation if there were a known efficient solution to k-colouring decision problem?

Function 2 demonstrates a simple mechanism of calling a decision algorithm repeatedly with different possible chromatic numbers until a successful number is found. As discussed above, the possible chromatic numbers range from 1 to n, the number of vertices in the graph. Therefore, the worst possible performance of function 2 is $O(n)$ comparisons. Because each of those iterations through the loop also calls the decision algorithm, the performance is ultimately $O(n \times O(DecisionAlgorithm))$. In short, if the k-colouring decision problem can be solved efficiently the k-colouring optimisation problem can be solved at worst slightly less efficiently, by a factor of n. Since the decision version is known to be NP-complete, the optimization problem is NP-Hard.

function 2 OptimizeGivenDecisionAlgorithm(G, k)

Input: G {a graph}
Input: k {the number of colours}
Output: the smallest number of colours with which the graph can be coloured
  for $i \leftarrow 1$ to ∞ **do**
    if $DecisionAlgorithm(G, i)$ **then**
      return i
    end if
  end for

The close relation between the problems of deciding k-colouring and optimising k-colouring results in an expectedly close relation between the solutions to the two problems. This relationship will make itself apparent as the paper progresses, most notably in the formulation of the brute force algorithm.

6 Algorithms

With the problem fully described, some known solutions can be discussed. Because k-colouring is such a difficult problem and no satisfactorily efficient solution is known, there are many attempts at making more efficient solutions. This paper will begin with a very poor algorithm, the brute force approach, and then discuss some solutions that perform better than the brute force algorithm.

6.1 Brute Force

The brute force algorithm assumes a simple mechanism for validating the rules of the colouring problem. The algorithm for validating the compliance to the colouring rules is both simple and fairly intuitive. Recall the small practical example above and how the edge representation changed as colours were selected.

function 3 VerifyColours(e)

Input: E {an array of "edge" doubles}
Output: the validity of the colour assignments
  for $i \leftarrow 0$ to $E.length - 1$ **do**
    if $E[i][0] = E[i][1]$ **then**
      return false
    end if
  end for
  return true

Function 3 runs in $O(m)$ comparisons or less. Because each vertex may have at most n edges, function 3 runs in at most $O(n^2)$ where the worst case is realised with a properly coloured fully connected graph.

function 4 BruteForce(G, k)

Input: G {a graph}
Input: k {the number of colours}
Output: whether the graph can be coloured in k colours

```
f ← k^(G.V.length)
F ← G.E {copy the edge array to a workspace}
for i ← 0 to f do
  l ← i
  for v ← G.V.length − 1 to 0 do
    for e ← 0 to G.E.length do
      if G.E[e][0] = v then
        F[e][0] ← l mod k
      end if
      if G.E[e][1] = v then
        F[e][1] ← l mod k
      end if
    end for
    l ← l ÷ k
  end for
  if verify(F) then
    return true
  end if
end for
return false
```

The brute force algorithm considers each possible assignment of colours to the graph stopping either on success or when it has considered all possible colour combinations. There are k^n combinations of assignments, and the first step in an iterative algorithm like function 4 is to generate that number. The main body of the function then iterates over each of those possibilities, making the general order of the algorithm $O(k^n)$.

Inside the main body of function 4 is an iteration first over n and then over m, or at most $O(n^3)$ followed by a call to function 3 which, as described above, runs in at most $O(n^2)$. All considered, the worst performance runtime is at worst $O(k^n + n^3 + n^2) = O(k^n)$.

As the proper focus of this paper is on the optimization problem, it is necessary to combine the two algorithms, function 2 and function 4. The modified running time of this algorithm is lower than $O(k^{n+1})$ as it requires running the brute force algorithm k times.

Function 4 reaches the worst case performance at any time where there is no solution. This observation closely aligns with the findings of [2] regarding problems with unlikely

solutions and average runtime.

> These results show that there is a phase transition for the cost of solving K-colorability problems, and it occurs at the critical average connectivity where the probability of a solution has dropped to almost zero. [2]

6.2 Contraction

A second deterministic algorithm for solving k-colouring is the contraction algorithm. The contraction algorithm performs contractions on the graph, combining two unconnected vertices into one logical vertex, connected with edges to the union of vertices to which the original two vertices were connected, as shown in function 5.

function 5 Contract(G,e)

Input: G {a graph}
Input: e {an edge to contract}
Output: a contracted graph
  for $i \leftarrow 0$ to $G.E.length - 1$ **do**
    if $G.E[i][0] = e[0]$ **then**
      if $G.E[i][0] = e[1]$ **then**
        $delete(G.E[i])$
      else
        $G.E[i][0] \leftarrow e[1]$
      end if
    else if $G.E[i][1] = e[0]$ **then**
      $G.E[i][1] \leftarrow e[1]$
    end if
  end for
  $delete(G.V[e[0]])$
  return G

An assumption belies this contraction. Contracting in this way yields an optimal contracted graph if there is an optimal colouring in which both original vertices were coloured in the same colour. While the contraction step handles the case where the vertices can be optimally coloured the same colour, there is clearly another possibility – that the vertices cannot optimally be coloured in the same way. To handle that case, an edge is added between the two original vertices in the original graph.

Both the edge-added graph and the vertex-reduced graph are in turn processed in much the same way until contraction is no longer possible. If contraction is no longer possible, the resultant graph is fully connected and its optimal colouring is n', the number of vertices in this copy of the graph. The behaviour of contraction leads to an actual algorithm that can solve the k-colouring problem.

function 6 Join(G,e)

Input: G {a graph}
Input: e {an edge to contract}
Output: a joined graph
  $G.E[G.E.length] \leftarrow e$
  return G

function 7 Contraction(G)

Input: G {a graph}
Output: the number of colours for an optimal colouring
  {try to find a pair of disconnected vertices}
  $edge \leftarrow null$
  for $i \leftarrow 0$ to $G.V.length - 1$ **do**
    for $j \leftarrow i + 1$ to $G.V.length - 1$ **do**
      $hasEdge \leftarrow false$
      for $e \leftarrow 0$ to $G.E.length - 1$ **do**
        if $(G.E[e][0] = i || G.E[e][1] = i) \&\& (G.E[e][0] = j || G.E[e][1] = j)$ **then**
          $hasEdge \leftarrow true$
        end if
      end for
      if $hasEdge = false$ **then**
        $edge \leftarrow (i, j)$
      end if
    end for
  end for
  {bound}
  if $edge = null$ **then**
    return $G.V.length$
  end if
  {branch}
  $c \leftarrow Contraction(Contract(g, e))$
  $j \leftarrow Contraction(Join(g, e))$
  return $min(c, j)$

Function 7 is a branch and bound algorithm, creating a tree of possible candidates which are easier to search than the original tree. In this tree, only the leaf vertices need be computed and since each leaf vertex can determine its optimal colouring in linear time, the overall performance of this algorithm is the size of the tree that is created. The size of the tree is a binary tree with a depth of the number of missing edges in the graph, which is no more than n^2. Therefore, the worst case runtime for the algorithm is $O(2^{n^2})$.

6.3 Greedy

On the other end of the spectrum from function 4 is a greedy programming approach to k-colouring. The greedy algorithm is a k-colour optimisation algorithm. This approach quickly yields a graph that is coloured according to the rules. Unfortunately, for certain types of graphs, the greedy algorithm may not devise the correct minimum graph. In cases where it does not find the correct minimum graph and the k value is smaller than the approximated minimum, this algorithm fails.

Function 8 does produce a valid graph in all cases. There are a number of particular portions of the code that are perhaps not entirely obvious. L is a list of the existing colours on the map. This list is initially populated with all of the colours considering them the ideal choices of available colours to use when colouring the next vertex. Every colour that is used in a neighbour of the current vertex is then removed. If there are no remaining colours, a new one is created. The first of the remaining colours, which may in fact be a new colour that was not used anywhere else in the map, is used to colour the vertex, which is, as in the brute force algorithm, symbolised by the assignment of the colour to the edges that correspond to the vertex.

Another somewhat complex portion of function 8 is the not-yet-defined $SelectNode()$. The particular implementation of this function is important and much debated. In fact, the implementation of this function is the very difference between the greedy algorithm working for all graphs and not working for all graphs. The sole responsibility of this function is to choose in which order the vertices are analysed.

Consider an intelligent $SelectNode()$ function that knows the solution to the colouring of a graph. To make the greedy algorithm succeed, it would need only to first return the vertices coloured with the first colour, and then all of the vertices adjacent to one of the first vertices that has the second colour and so forth until the vertices were coloured.

A more practical implementation could try to analyse the most connected vertices first. Consider, then, a simple graph that will help provoke some discussion of selection algorithms. The simple graph has six vertices each with one edge in and one out. This could be envisioned as a perfect hexagon. Since all vertices are equal and the shape fully symmetric, any vertex will do for a first choice. Once the first vertex is chosen, the next call of $SelectNode()$ will have to decide on which vertex to return next. The particular choice of the vertex directly opposite the original vertex has negative but interesting implications.

function 8 Greedy(G)

Input: G {a graph}
Output: the number of colours for an optimal colouring

```
F ← G.E {copy the edge array to a workspace}
usedColours ← 0
for i ← 0 to G.V.length − 1 do
    v ← SelectNode(i)
    for c ← 0 to usedColours do
        L.add(c)
    end for
    for e ← 0 to F.E.length − 1 do
        if F.E[e][0] = v then
            L.remove(F.E[e][1])
        else if F.E[e][1] = v then
            L.remove(F.E[e][0])
        end if
    end for
    if L.empty then
        usedColours ← usedColours + 1
        L.add(usedColours)
    end if
    for e ← 0 to F.E.length − 1 do
        if F.E[e][0] = v then
            F.E[e][0] ← L.first
        else if F.E[e][1] = v then
            F.E[e][1] ← L.first
        end if
    end for
end for
return usedColours
```

$$
\begin{aligned}
&G = (V, E)\\
&V = (a, b, c, d, e, f)\\
&E = (\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, a\})\\
&A = 0\\
&E = (\{0, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, 0\})\\
&D = 0\\
&E = (\{0, b\}, \{b, c\}, \{c, 0\}, \{0, e\}, \{e, f\}, \{f, 0\})
\end{aligned}
$$

After the initial two vertices have been decided, the particular order of choices no longer matters. The number of colours chosen in total will be $\frac{n}{2}$. This sort of decision is characteristic of crown graphs, graphs which are fully bipartite but not complete. The crown graph can be visualised as a two sets of vertices laid out vertically and evenly. There are no connections between any two vertices in the left set nor any two in the right set. Further, each vertex from the left set has a connection to every vertex in the right set save the vertex that corresponds to it vertically. This particular class of graphs form a dangerous case for the greedy algorithm and underline the importance of the selection function.

The performance of function 8 is substantially faster than that of the brute force algorithm. The outer loop executes exactly n times. Inside of the outer loop, there are three for loops. The first of these executes once per colour, which is n or less. Both of the remaining loops execute once per edge which is at most n^2 in the case of a completely connected graph each performing at most 2 comparisons. This brings the overall performance to a worst case of roughly $O(4n^3 + n^2)$, much more feasible than the $O(k^n)$ performance of function 4.

7 Heuristics

While performing the research for this paper, a few heuristics were developed that improve the performance of the algorithms. The reasons for developing these heuristics were twofold. Firstly, these heuristics formed interesting fodder for research and thought about the way that the various algorithms worked especially with border cases. Secondly, because a focus of this project was on the empirical work, these algorithms were executed countless times on real hardware. Since time is valuable and hardware limited, ways to significantly decrease the real-world runtime of the algorithms increases the amount of empirical testing that can be done.

7.1 Decreasing Work on Border Cases

Earlier in the paper, a few trivial cases were presented where polynomial algorithms can determine the k-colouring. These cases were the case of $k = 1$, $k = 2$, and $k = n$. The algorithms presented in this paper do not implement this set of limits. It is, however, an easy optimization to ignore 1 and 2 colourings and to check for completely connected graphs ahead of time. The Branch and Bound algorithm for contraction, function 7, requires this optimization for its functionality, but this same check can be easily added to function 4 or even function 8 to quickly determine a completely connected graph.

7.2 Early Node Colouring

The performance of function 4 is deplorable. It can be made a bit better by only inspecting one colour for the first vertex. Additionally, it is easy to determine a second colour, for some vertex connected to the first vertex, if there is one. If there is no connected vertex, then both the colour of the first vertex and the second vertex can be assumed and other colour options ignored. In the common case, where the first vertex has an edge, this takes the performance of brute force from k^{n+1} to k^{n-1}, a significant improvement if one plans to actually execute the algorithm and wait for its results.

7.3 Trimming

For many algorithms, there are vertices that can be removed before the algorithm is run without affecting the overall results. The vertices that were trimmed for the heuristic-using versions of algorithms for this paper were those with either no edges or exactly one edge. Since 1 and 2-coloured graphs are being ignored for the heuristic algorithms, the vertices with no edges or one edge can be removed with absolutely no effect on the optimal solution for the graph.

This particular heuristic works quite well for the brute force algorithm. It works even better for the contraction algorithm as the sub-graphs can also be trimmed at points where two vertices, a and b, are contracted into one and there is some vertex, c for which the only edges are to a and b. Trimming in the case of contraction completely eliminates branches that would otherwise have been processed, reducing the overall size of the tree substantially.

Trimming has an interesting side-effect with the greedy algorithm. Consider again a crown graph, but this time with an extra vertex, δ, and with a $selectNode()$ function that returns the vertices in declared order.

$$G = (V, E)$$

$$
\begin{aligned}
V &= (a, \delta, b, c, d, e, f) \\
E &= (\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, a\}, \{\delta, c\}) \\
a &= 0 \\
E &= (\{0, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, 0\}, \{\delta, c\}) \\
\delta &= 0 \\
E &= (\{0, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, 0\}, \{0, c\}) \\
b &= 1 \\
E &= (\{0, 1\}, \{1, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, 0\}, \{0, c\})
\end{aligned}
$$

The choice of c now destroys any chance of an optimal colouring. But clearly if trim removed δ, the optimal solution would have been reached in the intuitive order. This may lead to a premature conclusion that trim makes greedy algorithm more likely to succeed. A failing counterexample can be easily crafted, however, as demonstrated below.

$$
\begin{aligned}
G &= (V, E) \\
V &= (a, \delta, d, e, f, b, c) \\
E &= (\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, a\}, \{\delta, d\}) \\
a &= 0 \\
E &= (\{0, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, 0\}, \{\delta, d\}) \\
\delta &= 0 \\
E &= (\{0, b\}, \{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, 0\}, \{0, d\}) \\
d &= 1 \\
E &= (\{0, b\}, \{b, c\}, \{c, 1\}, \{1, e\}, \{e, f\}, \{f, 0\}, \{0, 1\})
\end{aligned}
$$

Because the greedy algorithm has a runtime based directly on the number of vertices, trimming does decrease the amount of work that the greedy algorithm needs to perform. Presuming that the number of cases that are helped and the number of cases that are harmed by the trim function are roughly equal, then this heuristic is beneficial even on the greedy algorithm.

8 Empirical Results

AlgoLab software written by Dr. Tom O'Neil was used as a framework for the testing of the algorithms and heuristics described in this paper. The AlgoLab software is written in Java

and for this reason, Java was chosen as an implementation language for the algorithms. The AlgoLab API provides for abstraction of the step count for each algorithm, separating the actual runtime from the performance metrics output by the system. This decision removes any platform-specific elements form the empirical results.

The testing platform generates random graphs based on parameters that the user specifies. These parameters indicate the user's preferences for graph size as well as the number of edges that should be generated. The user then specifies the number of trials that should run, clicks to engage, and waits. Both the number of edges and the number of vertices are very important to the performance of the algorithms tested.

All of the algorithm analysis in the previous sections of this paper show that the number of vertices is a primary factor in the performance of the algorithms. This fact played out regardless of the edge count, ignoring the uninteresting border cases which have special handling. Providing variance in number of vertices and smoothing the lines with a sufficient number of trials yielded graphs that showed characteristically polynomial curve characteristics for the greedy and greedy with heuristics algorithms. As expected, the brute force and brute force with heuristics algorithms showed decidedly exponential curve characteristics. The Contraction algorithm also showed exponential curve characteristics, but outperformed the greedy algorithm for numbers of vertices less than 10 or so.

In addition to experimenting with graphs of different sizes, graphs of different edge densities were also used to try to determine characteristics of the algorithms. As expected, the brute force algorithm had no variance in performance based on the number of edges. Adding heuristics to the greedy algorithm caused somewhat better performance on sparse graphs, but dense graphs have almost no trimming and thus the edge-specific component of the heuristics had very little effect, making it essentially equivalent to the standard brute force algorithm. The greedy algorithm has a runtime that is dependent on the number of edges directly. This caused greedy to perform substantially more steps to completion than with sparse graphs, but still outperformed the exponential algorithms by a wide margin. The contraction algorithm is the most heavily influenced by edge density of the algorithms discussed in this paper. The depth of the branching binary tree is dictated by the number of edges that have not been inserted into a graph. The degree to which contraction is edge-dependent in performance causes contraction to outperform the greedy algorithm for graphs with sizes up to between 15 and 20.

An additional feature of the AlgoLab framework is that it graphs and compares the optimisation result from each algorithm it runs on the same graph input. This output has shown high correlation between the approximation algorithm and the deterministic algorithms that were tested. The correlation is higher with sparse graphs, but even with dense graphs, the randomly generated data yielded a great majority of the tests to have exactly the same result and those that did differ were usually off of optimal by only one or two colours. The result comparison also provided good confidence in the validity of the implementations of brute force and contraction, which had complete consistency on all tested graphs.

9 Summary

This paper has presented the problem of colouring a graph such that no adjacent vertices are of the same colour, the problem known as k-colouring. The bounds and performance behaviour of the problem have been discussed. Both deterministic algorithms and an efficient non-deterministic algorithm were presented and analysed. The limits of the non-deterministic algorithm show it to be a fast but imperfect solution to even small and fairly simple graphs. Furthermore, empirical evidence was presented to reinforce the mathematical analysis of the algorithms. The problem of k-colouring remains a difficult problem to compute efficiently, but there are ways to improve it beyond the brute force approach through better algorithms, heuristics, and approximation.

10 Future Research

K-colouring remains an open and highly researched problem in computer science. As such, there is ample room for future research in the area. There are genetic and evolutionary algorithms that were not researched for this paper, but that have been published. There are also other approximation algorithms that are available that did not find their way into this research paper.

There are a number of heuristics that were investigated in this research. There are certainly many more to consider. One particular heuristic that the authors considered was comparing graphs in the contraction algorithm before embarking on contracting and joining the graph. Certainly if a matching graph has been solved, there is no particular need to solve the tree again.

The $selectNode()$ function used in the empirical testing was as simple as returning the vertices in the order they occurred. There is very likely a vertex order that increases the likelihood of optimal results from the greedy algorithm. The excellent performance characteristics of the greedy algorithm make this particular area a very practical one for cases where an actual implementation needs would need to decide the k-colouring problem with reasonable time constraints.

References

[1] Amitabh Chaudhary and Sundar Vishwanathan, *Approximation Algorithms for the Achromatic Number*. Journal of Algorithms 41, 404416, 2001.

[2] P. Cheeseman, B. Kanefsky, and W. M. Taylor, *Where the Really Hard Problems Are*. Proceedings IJCAI-91. Sydney, Australia, 1991.

[3] Philippe Galinier, Jin-Kao Hao, *Hybrid Evolutionary Algorithms for Graph Coloring*. Journal of Combinatorial Optimization 3, 379397, 1999.

[4] Geoscience Australia, available at `http://www.ga.gov.au`, 2009.

[5] P. Prosser, *An Empirical Study of Phase Transition in Binary Constraint Satisfaction Problems*. Artificial Intelligence, 82:81-109, 1996.

[6] Jonathan S. Turner, *Almost all k-Colorable Graphs are Easy to Color*. Journal of Algorithms 9, 63-82, 1988.