Marius Hahn

Graph Coloring - Four Color Theorem

Abstract

This work is about coloring a graph. The idea initiated form coloring geographical maps. It will show some briefly explain some approaches some graph coloring algorithms work. The main part will be the evaluation on different algorithms that has been implemented though out. These are Greedy Coloring, Backtracking, K-Coloring and Welsh Powell.

1 Introduction

This work focuses on graph coloring and possible algorithm implementations to color a graph. The idea came up when I read about map coloring. Especially about the question, "how many colors do you need to color the all states of the US such that there are no neighbor states sharing the same color?". The actual answer to this is very simple. At most four. Taking the each state as a vertex and add an edge between the vertices that are neighbors you will get a planar graph. Planar graphs have a chromatic number of four.

1.1 History

The four color theorem of planar graphs is a quite old idea. It seem like the question came um in 1852 in a letter from A. De Morgan to W.R. Hamilton. 1879 Sir Alfred Kempe came up with the first prove which was disproved by Heawod 1890. It took almost another hundred years as in 1977 Appel, Haken, and Koch came up with the final prove. It became famous for being the first prove that was done with heavy support of computers. [1]

1.2 NP-Hardness

Coloring graphs such that no neighbor vertices share the same color is known to be in the class of NP-Hard problems. Meaning it is not possible to find an algorithm that solve the problem in linear or poly nominal time, it is interesting to look for approximation algorithms[2].

2 Preliminaries

This work focuses on planar graphs with a chromatic number of four. The algorithms and approaches that are presented later, would work for general graphs too. The only additional information that is needed is the chromatic number of the graph. Therefore, knowing the number of colors needed you can also apply the algorithms to any non directed graph having a set of vertices and edges.

2.1 Greedy Coloring

Greedy is the key algorithm for all the algorithms that are used later on. Greedy coloring is an very simple and fast algorithm to color a graph but it is possible that result is very bad, meaning it uses a lot more colors than are need. Let C be the set of possible color, represented by integer numbers form zero to infinity. Having a undirected graph $G(V, E)$ where V is the set of vertices and E the set of edges between the vertices, this algorithm goes over all the vertices v in V, looks up N_v the set neighbors of v in E . Form N_v it is possible to look up the already used colors NC_v. Then it looks for the smallest number (color) c that is in C but not NC_v and assigns this number c to v. As you might already noticed this algorithms performance on how many colors it uses is highly dependent on the order it goes through all the nodes. Taking the right ordering greedy will give

the perfect solution, but finding this ordering is not trivial. The running time of this algorithm is $O(n)$ $\mathcal{O}$ where $n = |V|$ is the number of vertices in graph G. In real applications the it should not be possible to achieve this running time because usually you will also have to sort the vertices first. This will lead you to run-time of at least $O(n * log[n])$

2.2 Backtracking

Backtracking is a recursive method. First you have to choose how many colors you have to use. Then you choose a vertex to color with the smallest available color, meaning the one that has not yet been chosen by one of its neighbors. You pick one after another until you would have to allocate a new color that is not inside the range of the allow colors. This means the the vertex ordering you have traversed so far cannot be the right order. Therefore you pick another another vertex. If there is no vertex left anymore and you only could do a bad coloring, you backtrack the current path one level up. So you go back to the last colored vertex, you remove its color and try a different one. In the worst case this can go back to the very first vertex and start with another one. This method let's try out any possible solution. Therefore guarantees a perfect solution if there is one. This is a classic Dept First Search. A DFS has a worste case $\mathcal{O}(b^d)$ where b is the branching factor an d is the dept of the tree. In this example this branching factor is the average degree of the vertices and d is $|V|$ the number of vertcies in the graph.

2.2.1 Parallelism

This algorithm can be made parallel very easy, for today's multi core processors. Normally there are in between 2 and 32 cores that can run threads in parallel. The possible paths you can walk through the graph are independent from each other. Therefore they can run in parallel. When starting, there are $|V|$ paths. Considering that the first vertex has to be colored anyway the number of starting paths can be reduced by one. This leads $|V| - 1$ possible paths. Therefore you can start a search thread for each starting point.

2.3 Welsh Powell

The Welsh Powell takes the nodes in a given ordering. Form this ordering it takes the first node. If the node is already properly colored, meaning it has a color which is different form its neighbors it continues to the next node in the list. If the node isn't colored yet or is colored with wrongly, it colors the node with the smallest available color. After having colored this node it looks for all nodes that are not adjacent and colors them if they have not yet been colored in the same color. This approach had very bad run time and color results. As result only the results of the test are presented. The theoretical run time is not discussed any further.[3]

2.4 K-Coloring

The K-Coloring algorithm is like the greedy algorithm but with a special node ordering. It works like this:

1. Decide how many colors you want to have at maximum (this will be four in our case)
2. Copy the whole graph (for now on we work on the copy)
3. Find an vertex that has a degree smaller than k (look for a node that has 3 neighbors or less), if possible. If there is no node with degree k take one that has degree k+1 or more
4. Remove this node from the graph
5. Put this node on a Stack
6. Repeat until the graph is empty
7. Color the original graph greedy using the obtained stack as ordering

What's the trick of this algorithm? As long as you find a node that has degree smaller than k you can guarantee that you can color this node. In addition by removing a node from the graph the degree of other nodes will decrease, too. This means if you are lucky you can walk through the graph and never have to take a node that have a degree higher than $k - 1$. This would guarantee perfect k coloring. [4]

3 Algorithm & Implementation

The implementation is written in Kotlin. Kotlin a language that is build on the Java Virtual Machine JVM. Like Java itself it compile to Java Byte Code. I used Kotlin for the implementation because I wanted to give it a try and see the advantages or disadvantages over Java. If you know Java and or Python I assume that it will be easy to understand.

3.1 Data Set

The data set come from https://www.math.uwaterloo.ca/tsp/world/countries.html. It consists of 28 files where each file represents a country. Inside the files there are nodes that have a latitude and longitude. These coordinates represent locations of cities. As there are no edges state in the files, the edges will be added later by using triangulation. I will no go in detail here because it is done by using a library I found on github [5]. Through the triangulation we get a planar graph that is 4 color able. Triangulation is not main topic therefore its not further explained here. On initialization a the resource file is read and all containing nodes are initialized. Having the nodes the triangulation library is used to create the edges.

3.2 Data Structure

The are Two basic types:

1. *Node* is the class that represents the a node. A node is can be identified exactly by its latitude and longitude. As an additional attribute a node has a color which is represented by an Integer.

2. *Edge* is the class that represents the edges between the nodes. A Edges consists of two Nodes. An edge is unique on the node pair it contains. Additionally, having a Node and Edge, the edge can be asked for the nodes neighbor this itself.

3.2.1 Graph

The graph class is a more complex data structure. It maps one node to a set of edges. I used this because the ordering of the nodes is not interesting. Additionally the graph is build up once. Therefore we do not care much about insertion time either. On the other side, having a map its possible to access the edges of Node very quickly, and determine if a edges is contained by a set can be done very quickly too. The graph object does not only contain the data but also some functions that allows to access data of it directly. For instance it is possible get node ascending or descending by their degree.

3.3 Algorithms

All algorithms implement one interface called *GraphColoring* which has one function *color(graph : Graph) : Graph*. This function gets an object of type Graph and return an Object of type graph. By implementing this interface a class guarantees to return an colored graph. The algorithms details are given by the implementation class. There are multiple implementation to this interface such as:

1. *Greedy*: For this algorithm you first have to decide a node ordering, whether you want the algorithm to go over the nodes in ascending, descending order or shuffled.

2. *Backtracking*: On creation of this algorithm you have to decide how many colors you wan to use at most. Additionally you can decide if you want it to run parallel or sequential

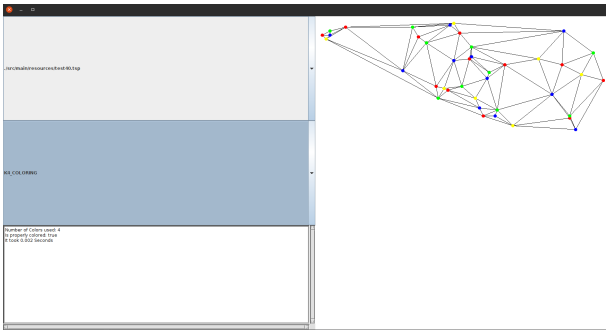
3. *K-Coloring*: Like for the Backtracking you have to decide how many colors it should use. In contrast to the backtracking algorithm the color count is not obligatory. If the algorithm doesn't find a solution with the given number of color, it will allocate additional colors

4. *Welsch Powell*: Here you also have to decide an ordering at creation time of this algorithm

Finally the is a an enum class *Algorithms* which can be used as a kind of factory. Every definition has an already created algorithm attached. In this enum you can look up, what are reasonable algorithm parameters.

3.4 GUI

As graph coloring doesn't make much sense when you never see a color, I though it might be worth to visualize them. Project contains a small java swing gui to see the results given by the algorithms.



From the first drop down you can select a test file. From the second drop down you can select an algorithm. The third field is to display some information about the graph like, how many nodes it has, how many colors are used and so on. Choose your algorithm wisely. Backtracking will only work on smaller graphs.

3.5 Comments on Implementation Details

The code itself does not contain comments. Personally I do not believe in commenting code. Comments are almost always out of date. I always try to name my methods classes and functions properly so that the code should be almost readable like a English text. Additionally methods or functions usually do not contain more than 5 lines.

4 Experimental Evaluation

In this section the calculation results of the algorithms are presented. We will focus on two aspects.

1. What was the run time of the implemented algorithm. Again, for Backtracking we will only compare small graphs as there are no test results for the big ones
2. How close is the achieved coloring compared to the best possible coloring. Backtrack is left out here because having an infinite amount of time it would always find a perfect solution if there is one.

All graphs provided have are four color able, because they are planar. Additionally it is not possible to they cannot be colored with less colors because through the triangulation they must have at least one clique with size four.

4.1 Data and Hardware

Our tests were made with a Dell XPS 9570. The specs of this computer are:

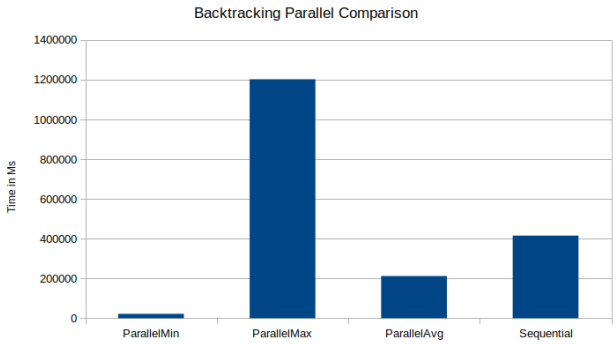
- Operating system: Ubuntu 20.04.1 LTS 64 bit
- Processor: Intel® Core™ i9-8950HK CPU @ 2.90GHz × 12 (6 cores physical cores)
- Memory: 31 GiB

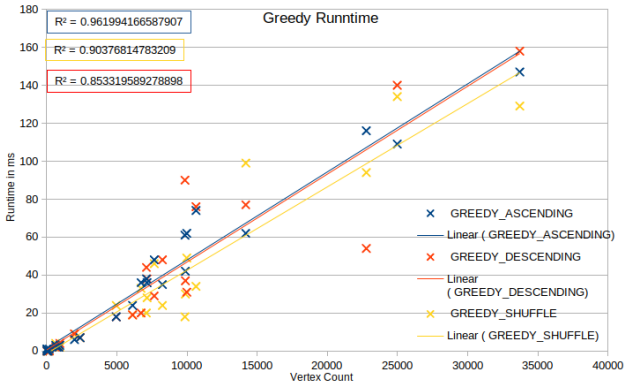
4.2 Run Time Analysis

In this part we will compare the run time of the different algorithms to color the graph.

4.2.1 Backtracking

There where only four graphs my backtracking algorithm reached the goal to color all nodes. The biggest was a graph with 40 nodes. It is contained in the file *test40.tsp*. The graph *test100.tsp* did not reach the goal in a ten hour test. As the graph with 40 nodes was the biggest for which I could find a result I did the run time comparison of the parallel algorithm on this graph.





The sequential coloring took 6 Minutes and 54 seconds. To see if the parallel implementation is faster I did a long running test. In the about ten hours the parallel algorithm the parallel algorithm colored the graph 37 times. On average the time to color the graph was 3 Minutes and 42 Seconds. The fastest coloring was 22 Seconds and the slowest took about 20 Minutes. The algorithm itself run has run with twelve threads on six processor cores. So the conclusion here is, parallel computing can speed up in this case but it isn't always faster. In my particular implementation the parallel algorithm creates different distinct orderings and tries to find a coloring solution. Which it tries first is not determined and can be always different. Therefore you can be lucky and a good ordering is scheduled early, so you will have your result quickly, but its also possible that the good order comes last and you have to wait for a long time.

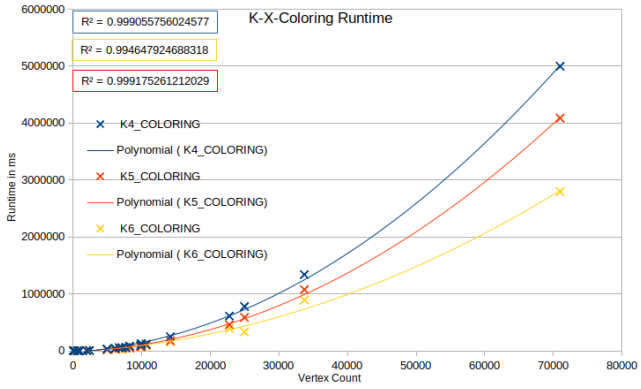
4.2.2 Greedy Coloring

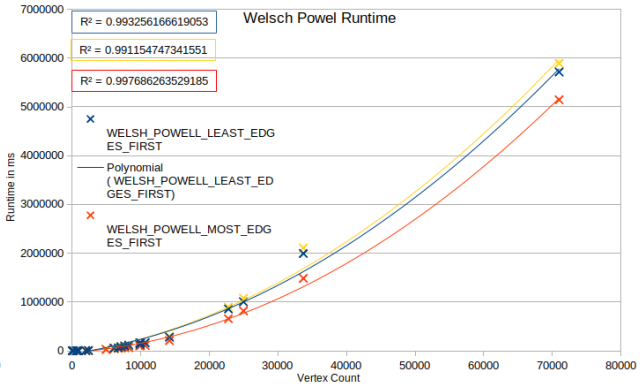
The greedy coloring is the fastest algorithm. I have implemented three different versions of the greedy algorithm. The greedy coloring goes over the nodes in a certain ordering and colors them. What changes is the ordering. The run time is doesn't seem to depend on order. Whatever ordering was taken the run time was roughly the same.

The run time looks linear. I did not further investigate on this as the biggest graph with 71009 nodes and 212992 edges took 0.2 Seconds to complete and this should be fast enough for most real world problems. The greedy coloring could be optimal, which wasn't case for any of the coloring. If greedy finds an optimal solution depends on the ordering. The orderings used here is, random, descending by node degree and ascending by node degree. All three of them had roughly the same run time. The run time itself is less dependent on the coloring step itself, as the coloring step just iterates over the nodes in given order, checks its neighbors and colors the node. So the run time of the coloring step is $O(n + d(v)_{avg})$ where n is the count of node in the graph and the $d(v)_{avg}$ is the average node degree

4.2.3 K-Coloring

The K-Coloring is a modified greedy algorithm. The coloring method is the same as for greedy. The key of all these algorithms is to find the right ordering. Backtracking tries to find this through backtracking, Greedy through ordering the nodes by degree. The one is very slow, the others result are very poor. K-Coloring was able to solve all test graphs. The biggest graph with 71009 nodes and 212992 took 1 hour 23 minutes and 16 seconds to complete, when using $k = 4$ meaning try to color it with four colors. The solution had five colors which is a valid solution because it does not guarantee a 4 color solution.





As the coloring itself is greedy the graph above shows clearly that run time here depends on finding the right node order. In this implementation the graph is stored in a java HashMap. On beginning the graph is copied. Then the algorithm looks for a node with degree $k-1$ in the graph. If you set $k = 4$ it looks for a node with three neighbors, because this node is must be four colorable as has only three neighbor. Then it removes the node from the graph copy an puts it on a stack. This is the time consuming point. Deleting something form a HashMap has $O(n * log[n])$ in java, but you have to add n again in this implementation as you must delete the edges that contains this node, too. This means you have to go over the nodes once again and delete all edges. Having this run time analysis in detail would fill another paper. In this test results a quadratic did fit quite well. You can be seen to is that the algorithm also gets faster by allowing more colors.

The regression curves in the diagram are quadratic and made on behalf of LibreOffice Calc. At a first glance analysis it look quadratic and the squared error is very low. There might be an implementation that performs better. For this project it was not worth investigate any deeper as this algorithm was not only slower than all the K-Color-Algorithm, it even uses more colors than the simple greedy approach.

4.3 Solution Analysis

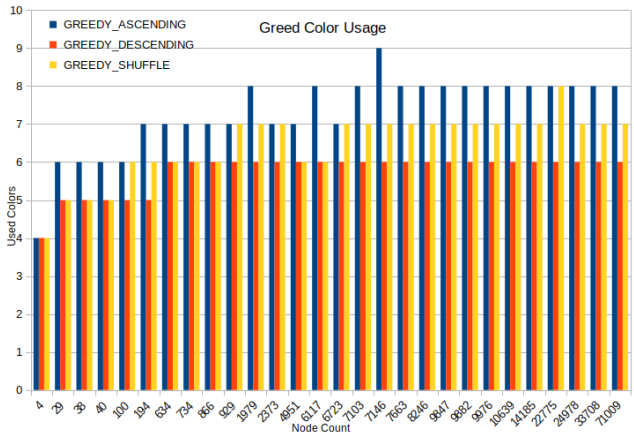
In this section we will discuss how good the found solutions were. The Backtracking algorithm is left out because it always produces a perfect solution.

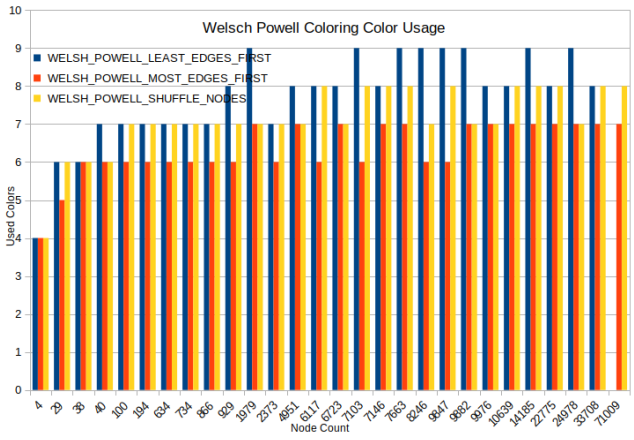
4.2.4 Welsh Powell

The following is the run time diagram for the Welsh-Powell algorithm:

4.3.1 Greedy Coloring

Greedy is dependent on the ordering it goes through the nodes. The simplest way to change the ordering of the nodes is to sort them descending or ascending by their degree or try a random ordering. Here is how good greedy performed with the different orderings:





It this figure you see the used colors over the count of nodes in the used graph. Again the perfect solution would be four. This perfect solution is only found for the test graph with four nodes. All the other graphs use more colors. Especially the ascending order seems to perform very poorly with often more than seven colors and once even nine colors. The shuffle ordering performs better, therefore it seems that the ascending order is not the one we are looking for. By ordering the nodes descending by their degree we obtain a worst case result of six colors, which is still 50% worse compared to the optimum but a lot better compared to the other ones.

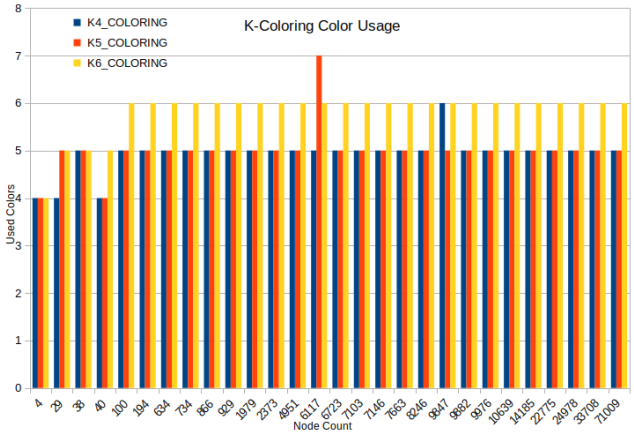
4.3.2 Welsh Powell

The Welsh Powel yet another variant of the greedy algorithm. We will take short on this one because it preformed poorly. The results where as follows:

As for greedy I tried it with different initial orderings. As you can see non of the ordering has an advantage over the already shown greedy algorithm. It even performed a bit worse. Additionally as you can see in the run time section it was very slow. It doesn't seem to be a proper candidate for a good approximation.

4.3.3 K-Coloring

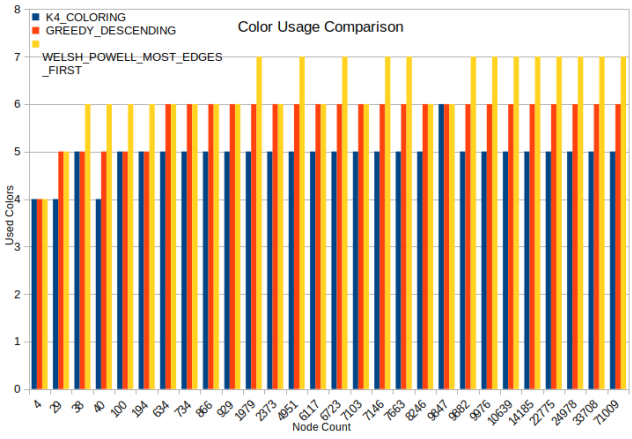
In K-Coloring you have to make a guess with how many color your graph can be colored. All the test graphs use are 4 colorable. Therefore I tried it with a $k = 4$, additionally I played around with $k = 5$ and $k = 6$.



As you can see with most of the $k = 4$ and $k = 5$ achieved a very good result of 5 colors. For the very small graphs with up to 40 nodes the $k = 4$ reached the perfect solution in two out of three times. In all the bigger graphs it find a solution with five colors expect one graph. This mean in the here tested graph the K-Coloring is only 25% compared to the back tacking solution. This makes it a very good candidate for a proper approximation.

4.3.4 Comparison

Here I will briefly show the difference between the above shown approaches. For this comparison I will only take the one that performed the best.



Here you can see the advantage of the K-Coloring method over Greedy and Welsh-Powell. K-Coloring perform as least as good as the other. In Addition in 24 out of 28 time it performs better than the other once. Welsh-Powell always produced a correct result but it was most time worse than the simple greedy.

5 Discussion and Conclusion

I tested four algorithm with their advantages and disadvantages. Welsh Powell I don't see an application as it performed very poorly in time and correctness of the result. For the other I say it depends on what you want. If it has to be very fast the Greedy Coloring with an descending node degree approach fit's best. It was quick and had acceptable results. If it has to be perfect and your graphs are small Backtracking is the best fit. If you want to have a good approximation that does not need to be perfect in time and number of uses colors the K-Coloring algorithm showed very good result.

5.1 Ideas for future Algorithms

One thing all the algorithms have in common is the actual coloring method. Use the smallest available color that is not yet been used by the neighbors. Taking K-4-Coloring always looks for nodes with three or less neighbors. Finding one guarantees that 4 - Colorability because its the node plus three neighbors. The problem is while traversing the graph an removing the exists the possibility that there is no node left having three or less neighbors. In the above implantation the algorithm look for a node with a next higher degree and takes the first it gets. This node does not need to be a good choice. At this point every node can be equally be a good or a bad choice. So this is a probable a point where it is worth to branch out and try all the paths again using the K-4-Color method. I assume this should lead to an exact solution.

References

[1] Vitaly I. Voloshin. Graph coloring: History, results and open problems. 2009.

[2] Tomás Feder and Rajeev Motwani. Worst-case time bounds for coloring and satisfiability problems. *Journal of Algorithms*, 45(2):192–201, nov 2002.

[3] Wenyu Wang, 2015.

[4] Andrew W. Appel. Verified functional algorithms. upenn.edu/, August 2020. Version 1.4 (2020-08-07 17:12, Coq 8.9.1 or later).

[5] Delaunator github. GitHub.