Graph Coloring

Iancu Alexandru-Gabriel

University Politehnica of Bucharest
Faculty of Automatic Control and Computers

Abstract. This paper focuses on performance differences between the heuristic and the exact algorithms for Graph Coloring while going in-depth about the complexity, structure and general idea behind each one.

Keywords: graph coloring, Welsh Powell, greedy, backtracking, chromatic number, runtime, heuristic, exact, complexity, correctness.

1 Introduction

1.1 Description of the problem

Graph coloring is a widespread problem in graph theory with numerous applications in different fields. It's goal is to assign a color from a limited poll of colors to a node in such a way that no adjacent vertex of it has the same color. Most known usage of this problem is map coloring. In all geopolitical maps there are used colors to accentuate the borders between countries in a world map or counties in a country map. To avoid overlaps you have to be sure when making such a map to use different colors for bordering regions.[5]

One question which arose is how many colors are enough to paint any world map? A man named Guthrie came up with a theorem which states that four colors are enough to paint any map in a plane. The theorem remained unproven until the age of computers when we could calculate exactly the minimum colors required to paint a map and so it became the roots from which our problem emerged and on which it expanded.[7]

As of the Four-Color Theorem that we discussed above we know for certain that any planar graph can be painted in 4 colors since any map can be represented as a planar graph with the reverse true.[6] Still there are graphs that can be painted in less than 4 colors and it would be helpful to know which and the value of that minimum number. In computer science that number is called the chromatic number and it has it's associated heuristic algorithms.

1.2 Examples of practical applications

In coloring maps one might think that you don't save that much by minimizing the number of colors since you would use the same amount of paint, but there are others applications in which knowing this number helps. An example would be making a schedule for an university. The faculty might have a limited number

of time slots available and it has a lot of classes for which it has to deliver a schedule. In this situation the colors are the time slots and knowing whether you have enough time slots for all students or you need to build more classes could be of importance. Also you can use a program, maybe one of the ones we're discussing in this paper, to make a schedule with the minimum number of slots required, though I don't recommend using the chromatic number of time slots since the schedule would become disastrous for students.

An important application of graph coloring is register allocation for which there is made a relation between (registers, variables) and (nodes, colors). Therefore making the assigning of variables to registers a problem about coloring a graph. [5]

1.3 Chosen solutions and evaluation criteria

For solving graph coloring I've chosen two heuristic(Welsh Powell and simple greedy) and an exact algorithm (backtracking). Backtracking solution is exact, but time inefficient while the heuristic programs sometimes fail, but are time efficient. Therefore for evaluation there will be time efficiency tests by clocking the runtime and also correctness tests for the heuristic algorithms. An important aspect of performance will be the chromatic number by which alongside correctness I will be able to assess which of the two heuristic programs has a higher chance of success and which is efficient in finding the chromatic number. The most probable outcome will be that one heuristic algorithm excels in correctness while the other is more time efficient since there can't be one that is good at both.

2 Algorithms

2.1 Backtracking

Description of the algorithm One way to accurately paint the graph is using backtracking. It's exact, but has an exponential complexity. In the worst case scenario it'll have to go through all possible scenarios to find a correct combination. In that case each node will be painted in all colors during the runtime.

The algorithm starts with one node, typically the first node(index 0) and assigns a color based upon its adjacent vertices. If there is no color available, the algorithm backtracks to the previous node and changes its color. It goes on until either it found a combination or it has gone through all combinations and doesn't have where to go back to. The backtracking itself is usually done using recursive functions as it'll be the case with my program.[2]

Complexity analysis Method 1: If we note the number combinations of node i as Z_i, c as the number of colors and n as the number of vertices then the total combinations of coloring would be:

$$Z_0 * Z_1 * ... * Z_{n-1} = c * c * ... * c = c^n$$

$Z_0 = Z_1 = ... = Z_{n-1} = c$, for any number n of nodes decided as input

Each node can choose between the c colors therefore each node's number of combinations would be equal to the number of colors used. There are c^n combinations and therefore a complexity of O(c^n) since we were talking about the worst case scenario. The worst case scenario being that the program had to through all possible combinations to find a solution. The complexity can be calculated before running the program since both the number of nodes and the maxim number of colors are to be given as parameters for the function.

Method 2: Considering the algorithm at [2], since it's a recursive function we can assign a T(n) to the function and write the following equation:

$T(n) = kT(n-1) + kn$, where k is the number of colors, n the number of nodes

The backtracking operation goes through all possible colors for the first vertex which are k possibilities and for each color there are k operations of testing if the color is available, the operation being searching the colors of its neighbors which at worst is all nodes(n). Also since there is a possibility to test all combinations we can attest that at each iteration there is the possibility of calling the recursive function k times therefore k remains constant throughout the equation. The way I'll resolve the equation is by using the iterative method:

$$\begin{aligned} T(n) &= kT(n-1) + kn \\ kT(n) &= k^2T(n-1) + k^2n \\ k^2T(n) &= k^3T(n-1) + k^3n \\ &... \\ k^{h-1}T(2) &= k^hT(1) + k^{h-1}n \end{aligned}$$

There are (n - 1) equations therefore h is equal to n. Now we substitute the left term of each equation in the equation above and get the following sum:

$$\begin{aligned} kn + k^2n + k^3n + ... + k^{n-1}n &= n(k + k^2 + ...k^{n-1}) \\ &= \frac{n(k^n - 1)}{k - 1} \end{aligned}$$

We can now see that the backtracking function has the complexity of $O(\frac{n(k^n-1)}{k-1})$, since (k - 1) and - 1 are constants then it's equal to $O(n * (k^n))$ which is close to the the $O(k^n)$ we guessed, the n factor not modifying the exponential nature of the function.

Regarding its space complexity we can attest that it's also exponential. Since the function would be called repetitively at most n times, where n is the total number of nodes. In each for loop there would be a counter that has a value between 0 and k, where k is the maximum number of colors used. To represent the counter in memory through bits it will occupy log(k) bits. For each recursive call there is a counter that occupies log(k) memory and there are n recursive calls. We can conclude from this that its time complexity is $O((logk)^n)$.

Advantages and disadvantages A clear advantage is its correctness, the program never fails if there is truly a solution to the problem. If the backtracking can't find a solution when given an infinite amount of time to search then there doesn't exist a solution to color the graph for the number of colors specified in the input. Another advantage is that it can find a graph with a number of colors equal to its chromatic number if the chromatic number of the graph is known.

The obvious disadvantage is its time efficiency in finding a solution. Since it has to go through all combinations until it finds a correct one it'll take a long time if the input graph has a large number of nodes(in testing I had problems with more than 20 nodes for 2000 tests and with more than 50 nodes on small samples). As we have seen by estimating the time complexity, the time increases exponentially for a linear increase of nodes in the input which is atrocious. Therefore it's unusable in practical situations such as register allocation for which time is an important factor, a computer can't stand too much to think about how to allocate memory in its registers since there is a user that is waiting.

Another disadvantage is its space complexity which is exponential, giving us another reason why it's impractical. In the register allocation example we can't use a lot of memory since the whole purpose is using memory efficiently. Using backtracking for such a task would put a strain on the computer's resources and it'll make the allocation of memory even more inflexible therefore not resolving the solution and making a bigger problem.

2.2 Simple Greedy Algorithm

Description of the algorithm A simple way to color a graph through a heuristic approach is by using a greedy algorithm. A simple one has the following steps(colors are numbered from 1) [8]:

1. Color the first vertex
2. Pick a blank vertex, randomly or through other ways; if there aren't any then the program finishes and shows or writes the output
3. Color the blank vertex with the lowest numbered color used, but not present in any of the adjacent vertices
4. If there isn't any use a new color which is the value of the last color + 1
5. Go to 2

Complexity analysis The function is iterative therefore it would be easier to determine its time complexity compared to backtracking. You can see the function at [8]. It begins with setting up the vector of available colors, which has as many elements as number of nodes, to true which denotes that all colors are initially available. Since the input variable is the number of nodes then that would be our 'n'. Going through a vector of n elements using a for loop has a complexity of $\theta(n)$.

The program uses a for loop to assign a color to each node, what matters when calculating the complexity of a for loop is the complexity of each iteration and how dependent is the inside of the loop on the counter. The first thing done in an iteration is setting as false the unavailable colors for the current node:

```
for (iter = listNode.begin(); iter != listNode.end(); iter++)
  if (graph->glist[(*iter).vertex].front().color != 0)
    available[graph->glist[(*iter).vertex].front().color] = false;
```

This particular loop verifies the color of each of current node's neighbors and therefore its complexity is the number of neighbors. The problem is that its number of neighbors aren't dependent on n or a constant therefore it's too vague to draw a conclusion on what θ may be like. The only conclusion we can draw is that since the node can't have more neighbors than the total number of nodes then it's complexity is O(n), where n is the total number of nodes.

The next loop searches for the color to paint the current node:

```
for (color = 1; color <= graph->nodes; color++)
        if (available[color] == true)
                break;
```

Again there can't be drawn any conclusion regarding θ, but it's obvious that its complexity is O(n) since in the worst case scenario the loop would break at the last color.

After that it assigns the color to the node and it verifies with an if statement whether the index of the currently assigned color is bigger than the last one since that would mean we used one more color and the chromatic number increased. Both of these operations are O(1) and don't matter.

The important part is the last part of the for loop iteration where we reset the available vector:

```
for (int i = 1; i < graph->nodes; i++)
        available[i] = true;
```

Since it's a for loop that goes to the end one at a time then it has n number of iterations, where n is the total number of nodes. We can therefore attest that it's complexity is $\theta(n)$ which means that it's the dominant part of the iteration because $\theta(n)$ + O(n) + O(n) + 2*O(1) is equal to $\theta(n)$. This for loop doesn't use the counter of the main for loop therefore it's independent and since the bigger

loop had n iterations we can attest that the complexity of the whole algorithm is:

$$\begin{aligned}\theta(n) + \theta(n) * \theta(n) &= \theta(n) + \theta(n^2) \\ &= \theta(n^2)\end{aligned}$$

The program uses a finite number of counters, the number being constant and independent of the total number of nodes. Each counter used goes to maximum n, where n is the total number of nodes. To represent these counters in bits we would need to use log(n) memory. Since there are a finite k number of counters we can say that the space complexity of using these counters is O($k*logn$) which is equal to O($logn$). This particular program uses an availability vector during the runtime that has n elements which are boolean therefore its space complexity is $\theta(n)$ because you have to use only 1 bit to represent the two states(true and false) in the memory. The pointers to nodes in the graph list are insignificant since each of them occupies 4 bytes, the memory needed to store an address. Therefore the space complexity of the entire program is:

$$O(logn) + \theta(n) = \theta(n)$$

The space complexity of the algorithm is polynomial.

Advantages and disadvantages The algorithm's advantage is its time complexity and therefore its runtime length. The time complexity is polynomial which is a magnitude better than the exponential function at backtracking. Even more so because the polynomial is rather small being second degree. Another advantage is that its space complexity is polynomial therefore it doesn't need that many resources and the degree of said polynomial is only 1. It's perfect for register allocation since it doesn't use a lot of resources in either time and space. The register allocation doesn't have to be entirely correct, it has to have an allocation that is overall uniform and this algorithm can do that.

One disadvantage is that it's heuristic and therefore it has a chance of failure when computing a coloring. Still it's better to at least have a chance of success in a relatively small amount of time than the algorithm being exact, but taking almost forever with backtracking or any other exact algorithm. We can therefore conclude that one of its strong points is computing big graphs since its time complexity permits a large number of nodes. Also it may be faster to run the algorithm several times until we get a correct result than running once backtracking. We can do that because testing the correctness of a graph coloring is polynomial.

We can test for each node if any of its neighbors has the same color as him, if it has we stop the program and print fail, if it hasn't we continue until the end when it'll print success. Therefore the complexity of the testing will be O(n^2) since we go once through all the nodes and for each iteration we have a for loop

through which we verify each neighbor which at most can be all the nodes in the graph, n being the number of nodes. We can use the function isSafe used in the backtracking algorithm [2] in a for loop that goes through all the nodes :

```
bool isSafe(GraphList* graph, int v, int color) {
  list<Node> nodeList = graph->glist[v];
  // if the node doesn't have neighbors then return true
  if (nodeList.size() == 1)
    return true;

  list<Node>::iterator iter = nodeList.begin();
  // it jumps over the node that is tested
  advance(iter, 1);
  for( ; iter != nodeList.end(); iter++)
    if (graph->glist[(*iter).vertex].front().color == color)
      return false;

  return true;
}
```

```
bool isCorrectGraph(GraphList* graph) {
  for (int v = 0; v < graph->nodes; v++)
    if (isSafe(graph, v, graph->glist[v].front().color) == false)
      return false;
}

return true;
}
```

We will test if its own color is eligible for itself instead of a random color since there isn't any assignment to be done. Running both the heuristic program and the testing will be $O(n^2) + \theta(n^2)$ which is $\theta(n^2)$. Supposing that we run a finite k number of times the heuristic and the testing we will get a correct graph then the complexity would be $\theta(k * n^2) = \theta(n^2)$. Running the program and the testing several times didn't change at all the time complexity so we can conclude that coloring a graph in this way is a magnitude better than running backtracking. I've been praising it a lot, so let's see if it has any disadvantage. Its disadvantage is that most of the time the resulted graph will have more colors than the chromatic number of the graph. To combat this I'll present an heuristic algorithm that gets better results in this department.

2.3 Welsh Powell Algorithm

Description of the algorithm The Welsh Powell Algorithm is an iterative greedy approach used to find the chromatic number of a graph, but can also be used to simply find a solution to paint a graph. The algorithm is heuristic as it has a small possibility of giving a non-optimal configuration of the graph, but the possibility is smaller compared to other heuristic algorithms.

The steps of Welsh Powell [3] :

1. Calculate the degree of each vertex and arrange them in descending order
2. Color the first blank vertex in the list
3. Search for the next blank vertex that isn't adjacent to any vertices that have the current picked color
4. Do 3 until you reach the end of the list
5. If there are still blank vertices, pick another color and go to 2

Complexity analysis The algorithm was entirely written by me therefore I don't have a reference for you to access. I will post snippets of relevant code so that my paragraphs can be followed.

Firstly, inside the algorithm you have to calculate the degree of each vertex and sort them in descending order. Since I used lists inside the program then the degree of each vertex would be the size of its list of neighbors - 1. Let's suppose that calculating the size of the list is O(1) then calculating the degrees of each node will be $\theta(n)$. To sort the degrees I'm using qsort which has a known complexity of $\theta(nlogn)$. Therefore the time complexity of the first step is:

$\theta(n) + \theta(nlogn) = \theta(n + nlogn) \quad = \theta(nlogn)$, where n is the number of nodes

After that I search the first blank node from the ordered list which would be the node that has the highest degree between the current blank ones.

```
for (i = nextNode; i < graph->nodes; i++) {
  int vertex = degrees[i].vertex;
  if (graph->glist[vertex].front().color == 0) {
    graph->glist[vertex].front().color = color;
    break;
  }
}
```

This for loop is at most O(n) since it's going from a positive number nextNode to the total number of nodes. Searching for a more restrictive complexity isn't necessary because this complexity is encompassed by the next bit of code:

```
for (j = i; j < graph->nodes; j++) {
  if (graph->glist[v].front().color != 0)
        continue;

  // color the node if it isn't a neighbor to the node in the
  // above piece of code and to all nodes that were colored in
  // this for loop before him
  for(iter = listNode.begin(); iter != listNode.end(); iter++) {
    if (graph->glist[(*iter).vertex].front().color == color) {
      isNeighbor = 1;
      break;
```

```
    }
  }

  if (isNeighbor == 0)
    graph->glist[v].front().color = color;
}
```

The main for loop if it had O(1) operations each iteration would be at most O(n). The for loop inside it goes to at most O(n) since there can't be more neighbors to a node than the total number of nodes. Therefore this whole operation is at most $O(n^2)$.

Both pieces of code would be O(n) + $O(n^2)$ = $O(n^2)$. Both operations will be done again for a blank node that isn't a neighbor to the first selected node and so on for another maximum n operations. Overestimating that it would do n iterations then we can say that the whole process is $O(n^3)$. Now there is the certainty that the program can't go over $O(n^3)$.

The reason I didn't search for a more restrictive complexity is because I know an example that demonstrates that this whole piece of code is $O(n^3)$ and not lower. The example is a graph in which each node is neighbor to each node, I've found out that this is indeed the worst case scenario for my algorithm and I'll calculate below its complexity.

The first colored vertex would be i = 0 with the color c = 1. Then I'll go through the rest (n - 1) number of the vertices in the graph and search for each of them if there are any neighbors with that color. Since each node is a neighbor to each then they'll be neighbors with the vertex i = 0 and therefore none of them will be colored. Through the process of searching for a neighbor they will have to do at most n - 1 searches since in the worst case the node i = 0 is at the end of the list.

Then I'll have to do the whole process for the vertex i = 1. In this case we'll look through (n - 2) vertices since the vertex i = 0 is colored. For each one of them they'll have to search through at most (n - 2) vertices since in the worst case the node i = 0 and i = 1 are at the end of the list. It'll continue to go on in this way:

$$\begin{aligned} i = 0 &=> (n-1) * (n-1) \text{ operations} \\ i = 1 &=> (n-2) * (n-2) \text{ operations} \\ &... \\ i = n-2 &=> 1 * 1 \text{ operations} \\ i = n-1 &=> 0 * 0 \text{ operations} \end{aligned}$$

Summing all these operations would give us the following sum:

$$1 * 1 + 2 * 2 + ... + (n-2) * (n-2) + (n-1) * (n-1) =$$
$$= \sum_{k=1}^{k=n-1} k^2 = \frac{n(n-1)(2n-1)}{6} \in \theta(n^3)$$

Now we know for certainty that there's an example for which the program has $\theta(n^3)$ complexity therefore the general program will have $O(n^3)$ time complexity because we demonstrated that it can't go to a higher degree.

To calculate its space complexity I will ignore the counters since they don't put a strain on the memory since their complexity is only $O(k * log(n))$, where k is a finite number. What matters the most in this case is the vector of degrees that has to be sorted. The vector has the following structure as elements:

```
class Degree {
public:
  int degree;
  int vertex;
};
```

Both the degree and vertex can go to maximum n, where is n the total number of nodes. A node can't have a degree bigger than the total number of vertices. To represent these two numbers in memory we would have to use at most log(n) bits. For a vector of n such elements we would have a complexity of $O(2 * logn * n)$ which is equal to $O(n * logn)$. A hypothetical best care scenario would be when the structure occupies a constant amount of space, in that case the complexity of the algorithm would be $\theta(n)$ since the length of the vector is always equal to the total number of nodes.

Advantages and disadvantages Its advantages compared to backtracking are the same ones I've talked about when discussing the advantages of the simple greedy algorithm. Its time complexity is polynomial and it can also be paired with a testing program to efficiently get the desired result. Another advantage is that it can get the chromatic number or it can be close to its value, whereas backtracking needs the known chromatic number to color efficiently. It also has better results in making a correct resulted graph than the simple greedy algorithm while also using on average a smaller number of colors.

Its disadvantage compared with the simple greedy algorithm is its time efficiency which is a degree higher since it has $O(n^3)$ complexity while the simple greedy algorithm has $\theta(n^2)$. While I didn't go more in-depth regarding the best case scenario, it is the case when the are no edges in the graph connecting the nodes. In that scenario its complexity is $\theta(n^2)$ therefore we can conclude that there isn't an example where Welsh Powell is more time efficient than the simple

greedy algorithm, at best is as fast. Another slight disadvantage is its space complexity which is at worst a factor of log(n) higher than the simple greedy algorithm. Best case scenario it has the same space complexity as the average case complexity of the simple gredy algorithm which would be $\theta(n)$.

3 Evaluation

3.1 Test construction

The 3 algorithms are evaluated based on the average length of runtime, on the chromatic numbers and on how many times they get a correct combination. The graphs made for the tests have a varying number of nodes between 1 and 20 for backtracking since it has an exponential complexity and TBD for the others. The tests are be generated randomly through a python3 program written by me using the pseudorandom methods from the NumPy module and the outputs are generated with bash by running the C++ program repeatedly. To calculate the average results of the output files I'm using a python file.

The average length of runtime is calculated inside the C++ program using the high resolution clock of the chrono library. There are shared tests between the three algorithms and upon them I'll make the first conclusion by averaging the runtime on the total number of tests. There will be additional number of tests with a higher number of nodes for the heuristic algorithms. To make a comparison between the extra tests for the heuristic and the base tests for the backtracking I am also calculating the time/number of nodes. This way I'll be seeing if a linear increase of the number of nodes will result in a linear increase of the length of runtime.

I will compare the chromatic numbers between the two algorithms with no external knowledge upon the real chromatic number. The number of colors used for each graph will be tracked in each program and further used by running the backtracking algorithm for the resulted number of colors as its input. This way I will test the results of the backtracking algorithm and also test whether the heuristic algorithms used a valid number of colors.

Backtracking is an exact algorithm and will always give a correct combination if it there is one. I don't think that backtracking will fail any test in this regard so I'll focus my attention on how correct are the greedy and the Welsh Powell algorithms. I will test the correctness of the graph using the isCorrectGraph and isSafe functions explained at page 7.

3.2 System specifications

```
*-memory
description: SODIMM DDR4 Synchronous
size: 16GiB
width: 64 bits
clock: 2667MHz (0.4ns)
```

```
*-cpu
product: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
size: 1934MHz
capacity: 4100MHz
width: 64 bits
clock: 100MHz

*-sata
vendor: Intel Corporation
version: 10
width: 32 bits
clock: 66MHz

Distributor ID: Ubuntu
Description: Ubuntu 19.10
Release: 19.10
Codename: eoan
```

3.3 Evaluation results

Table 1 (1-20 nodes):

Algorithm	Success rate	Chromatic number	Elapsed time	seconds/node
Welsh Powell	100%	3.90	5.98E-06	4.11E-07
Simple Greedy	100%	4.19	2.66E-06	1.89E-07
Backtracking 1	100%	3.90	1.27E-02	5.63E-04
Backtracking 2	100%	4.19	4.87E-06	2.91E-07

Backtracking 1 represents the execution of the backtracking program with Welsh Powell's chromatic number as input and respectively Backtracking 2 uses Simple Greedy's chromatic number. I ran 2000 tests with nodes varying from 1 to 20 to accommodate backtracking's exponential complexity. Running the tests took 30 seconds which is reasonable. The success rate and the chromatic number didn't change from one testing session to another. The runtime length changes slightly such as from 6.16E-06 to 6.34E-06 or from 1.32E-02 to 1.28E-02 which is most likely so because of the internal deviations of the system. The same tests were used for each testing session and the elapsed time it's the time that it takes for one test to finish. The seconds/node column is calculated for each test and averaged.

Table 2 (20-100 nodes):

Algorithm	Success rate	Chromatic number	Elapsed time	seconds/node
Welsh Powell	100%	10.59	1.89E-04	2.63E-06
Simple Greedy	100%	11.72	3.63E-05	5.49E-07

I've made another 2000 tests with varying nodes from 20 to 100. I've had to get backtracking out of the picture since its time complexity is atrocious. This time running the tests just for the two heuristic algorithms took only 4 seconds despite the increase in nodes. Between testing sessions the chromatic number and the success rate didn't change and the runtime length had small deviations:

- 1.89E-04 → 1.92E-04
- 2.63E-06 → 2.67E-06
- 3.63E-05 → 3.69E-05
- 5.49E-07 → 5.53E-07

Table 3 (100-500 nodes):

Algorithm	Success rate	Chromatic number	Elapsed time	seconds/node
Welsh Powell	100%	33.03	1.58E-02	4.25E-05
Simple Greedy	100%	37.75	9.40E-04	2.73E-06

This time I've settled for only 1000 tests since the generation started to take several minutes whereas generating the 2000 tests for table 2 took a few seconds. The testing this time increased to only 24 seconds which is reasonable and as expected the rose from 20-100 nodes to 100-500(with a 300.51 median) resulted in a linear increase for the testing time. An interesting observation is that the generation increased as a high polynomial or even an exponential and therefore the tests for the next few tables will become smaller and smaller. Time deviation is still under control.

- 1.58E-02 → 1.77E-02
- 4.25E-05 → 4.76E-05
- 9.40E-04 → 1.01E-03
- 2.73E-06 → 2.96E-06

Table 4 (500-10000 nodes):

Algorithm	Success rate	Chromatic number	Elapsed time	seconds/node
Welsh Powell	100%	354.00	1.09E+02	1.36E-02
Simple Greedy	100%	430.00	6.56E-01	8.67E-05

I've run only a testing session of 10 tests with a median of 6112.70 nodes. One limiting factor was the generation of graphs which took 5 hours and made it hard to go over 10 tests or have a higher number of nodes, cementing this testing session as the last. Another limiting factor was the testing which took 40 minutes which is more acceptable than the generation, but still an obstacle. I don't have an example for its deviation therefore this all I can muster about it, but it'll be useful for the analysis nevertheless.

3.4 Evaluation analysis

Firstly, we should look if the data correlates with our time complexity analysis. The data in table 1 shows that each heuristic has a smaller elapsed time on average compared to its backtracking counterpart. It also shows that Welsh Powell uses a smaller chromatic number than the simple greedy algorithm which is as expected.

The interesting part is that Backtracking 2 does better than Welsh Powell regarding runtime length in table 1. I've discovered that running Backtracking 2 isn't a problem for a total number of nodes bigger than 20 and made the following table to compare Welsh Powell and Backtracking 2:

Backtracking 2 vs Welsh Powell (runtime)

	Backtracking 2		Welsh Powell	
Number of nodes	Elapsed time	seconds/node	Elapsed time	seconds/node
1-20	4.87E-06	2.91E-07	5.98E-06	4.11E-07
20-100	1.98E-04	2.75E-06	1.90E-04	2.65E-06
100-500	1.66E-02	4.48E-05	1.58E-02	4.25E-05
500-10000	1.26E+02	1.57E-02	1.09E+02	1.36E-02

It's observable that even though Backtracking does better for a small number of nodes (1 to 20), overall Welsh Powell is on top. The results of both runnings are close in their time of execution, Backtracking 2 being behind by a small amount. Backtracking 2 is also a magnitude faster than Backtracking 1 even though it's the same piece of code with exponential time complexity. That means that a restrictive number of colors such as the precise chromatic number gave by Welsh Powell takes a toll on backtracking.

Also since its execution time is comparable to Welsh Powell which has a time complexity of $O(n^3)$ then it can be concluded that backtracking's time complexity for simple greedy's chromatic number has a $O(n^3)$ time complexity which would be great for an exact algorithm. There's a pair to be made between simple greedy and backtracking that would give an exact solution for $\theta(n^2) + O(n^3) \rightarrow O(n^3)$. A problem is that since all tests passed for simple greedy's algorithm I don't know how well it would work for a chromatic number from a failed run of the simple greedy algorithm. If it doesn't work then the pair is obsolete since its whole purpose is to use backtracking to get a good result from simple greedy's chromatic number when it fails.

Regarding chromatic numbers, if one looks only at the tests with graphs containing 1 to 500 nodes, they might say that the simple graph is superior because it's significantly faster than Welsh Powell and also because its chromatic number deviates only by a few units. Thankfully I also did tests for graphs with a number of nodes between 500 and 10000 and in table 4 we can see that there is a significant difference between the two algorithms, simple greedy's chromatic number is 75 units bigger than its counterpart. The chromatic number didn't

deviate between testing sessions in tables 1 to 3 therefore we can assume that it didn't for table 4. If we calculate the difference between the two chromatic numbers for each table we have:

- $\frac{4.19}{3.9} = 1.07 \rightarrow 7\%$ for nodes ranging from 1 to 20
- $\frac{11.72}{10.59} = 1.10 \rightarrow 10\%$ for nodes ranging from 20 to 100
- $\frac{37.75}{33.03} = 1.14 \rightarrow 14\%$ for nodes ranging from 100 to 500
- $\frac{430}{354} = 1.21 \rightarrow 21\%$ for nodes ranging from 500 to 10000

It's observable that simple greedy's chromatic number grows faster than Welsh Powell's and the difference becomes more significant the bigger the number of nodes are. If a practical problem requires precision in the construction of the graph then Welsh Powell excels in this regard. We can also do these calculations in regards to execution time:

- $\frac{5.86E-06}{2.66E-06} = 2.2$ times bigger for nodes ranging from 1 to 20
- $\frac{1.89E-04}{3.63E-05} = 5.2$ times bigger for nodes ranging from 20 to 100
- $\frac{1.58E-02}{9.40E-04} = 16.8$ times bigger for nodes ranging from 100 to 500
- $\frac{1.09E+02}{6.56E-01} = 166.15$ times bigger for nodes ranging from 500 to 10000

We can clearly see that the difference in time grows significantly from one table to another. The difference for table 4 is quite important and says that Welsh Powell is worth it only if precision is critical because otherwise the degree difference between time complexities makes it inefficient in comparison to simple greedy.

All tests ranging from 1 node to 10000 nodes were correctly colored by each algorithm, all of them having a rate of success of 100%. Hence the heuristics are good enough for a entirely random graph with a number of nodes smaller than 10000. They may be as successful for nodes bigger to a great extent, but I didn't generate that far.

4 Conclusions

Regarding this problem there are two main aspects which are precision and time complexity and there isn't an algorithm that excels in both. Even so I can conclude that heuristics are more practical and flexible. There is a great extent of approximation algorithms that can be made to solve graph coloring, each one having a different balance between the two aspects.

In this paper I talked about Welsh Powell which excelled by having a small chromatic number and the simple greedy algorithm which excelled in runtime length. Both are good and can be applied depending on what are the requirements for the problem. Also both can be enhanced in correctness by repeated runnings alongside a testing program. Therefore if correctness is the only desired factor

then in this pair it would fit simple greedy, if the problem requires correctness and precision then we can use Welsh Powell.

In our case both programs tend to have unique outputs for specific inputs, but they can simply be modified by using a pseudorandom function to increase the number of possible results and therefore increasing the chance to get the desired output by pairing it with a testing program. What I want to say through this paper is that one shouldn't look for correctness in graph coloring since it can be satisfied in both cases, but instead they should know whether having a fast result or a precise result is more important for the problem.

References

1. LNCS Homepage, http://www.springer.com/lncs. Last accessed 16 December 2020

2. GeeksforGeeks m Coloring Problem — Backtracking-5, https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/?ref=lbp. Last accessed 16 December 2020

3. GeeksforGeeks Welsh Powell Graph colouring Algorithm, https://www.geeksforgeeks.org/welsh-powell-graph-colouring-algorithm/. Last accessed 16 December 2020

4. OpenGenus Overview of Graph Colouring Algorithms, https://iq.opengenus.org/overview-of-graph-colouring-algorithms/. Last accessed 16 December 2020

5. GeeksforGeeks Graph Coloring — Set 1 (Introduction and Applications) , https://www.geeksforgeeks.org/graph-coloring-applications/?ref=lbp. Last accessed 16 December 2020

6. Stonehill College The Four-Color Theorem , https://web.stonehill.edu/compsci/LC/Four-Color/Four-color.htm. Last accessed 16 December 2020

7. Wolfram Mathworld Four-Color Theorem, https://mathworld.wolfram.com/Four-ColorTheorem.html. Last accessed 16 December 2020
8. GeeksforGeeks Graph Coloring — Set 2 (Greedy Algorithm), https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/?ref=lbp. Last accesed 16 December 2020