An Analysis on Graph Colouring Algorithms and Their Efficiency

To what extent can concepts from vertex graph coloring be used to efficiently solve real-life scheduling and allocation problems?

IB Extended Essay - Mathematics

Word Count: 3338

Contents

1 Introduction

The history of graph theory dates back to the 1700's, when Leonhard Euler was tasked with solving the Königsberg bridge problem - to determine if there existed a path across each of seven bridges without crossing the same one twice. Albeit a relatively simple problem for modern mathematics and computing, it was Euler's solution and proof that laid down the foundation of this field (Paoletti, 2011).

Since then, graph theory has evolved into a set of problems and algorithms that we are familiar with today. One such example is the idea of graph colouring. At its simplest, graph colouring is looking for a way to assign specific colours to the nodes of a graph such that the number of colours is minimized and no two adjacent nodes share a common colour.

I have always enjoyed participating in math contests, and have recently taken up computer science and competitive programming. One, if not the most common topic in competitive programming is graph theory. While it can be argued that graph colouring is a study in computer science, the theory behind every algorithm and concept is rooted in mathematics, and that is what I intend to explore in this extended essay. In doing so, I hope to find some interesting and unique ways to implement the mathematical concepts of graph colouring to solve real-world problems such as Sudoku and aircraft scheduling. That being said, the research question of this essay will be:

To what extent can concepts from vertex graph coloring be used to efficiently solve real-life scheduling and allocation problems?

Specifically, the focus of this essay will be around the varying degrees of effectiveness of different graph colouring algorithms. By analyzing Sudoku, we will see how some problems have no efficient solutions and can only run fast for small sizes. On the other hand, problems such as aircraft scheduling can be solved with fast algorithms regardless of size.

2 What is Graph Theory/Graph Colouring?

Before we can get into graph colouring and its real-life applications, we must first understand what graphs are and how we can represent them. Essentially, a graph is a mathematical data structure consisting of a set of different objects, with some pairs of objects being connected through links. Objects are often referred to as vertices or nodes, with connections being called edges (Lewis, 2016).

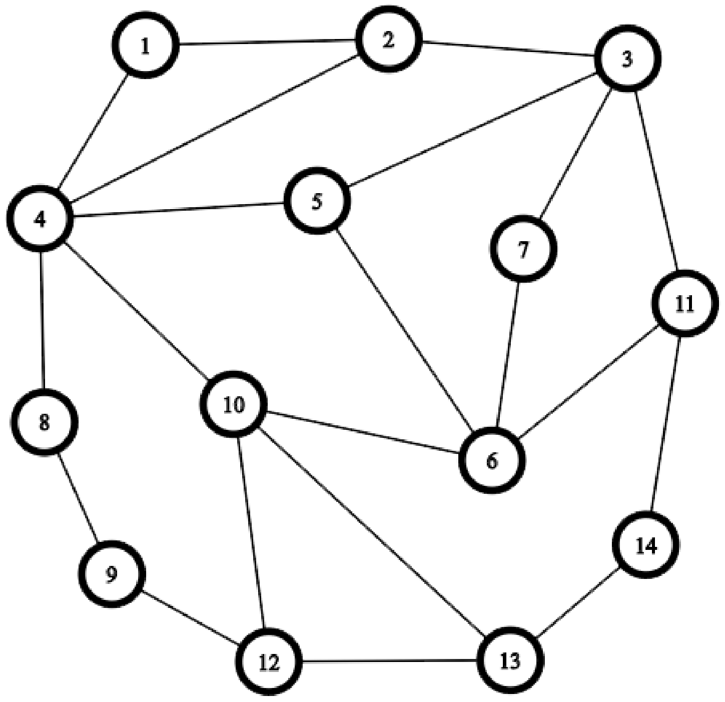


Figure 1: An example of a graph - the numbered circles are vertices and the lines connecting them are edges

In mathematics, the most common way of representing a graph is through what is called "edge list representation" (Lewis, 2016). For example:

$$G = (V, E)$$

would denote that graph G is comprised of a vertex set V and edge set E. To illustrate this, we can construct an edge list representation for the graph shown in Figure 1.

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}\},$$

This is the vertex set. It is essentially a list that contains each vertex of graph G.

$$\begin{aligned}E = \{&\{v_1, v_2\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_5\}, \{v_3, v_7\}, \{v_3, v_{11}\}, \{v_4, v_5\},\\ &\{v_4, v_8\}, \{v_4, v_{10}\}, \{v_5, v_6\}, \{v_6, v_7\}, \{v_6, v_{10}\}, \{v_6, v_{11}\}, \{v_8, v_9\},\\ &\{v_9, v_{12}\}, \{v_{10}, v_{12}\}, \{v_{10}, v_{13}\}, \{v_{11}, v_{14}\}, \{v_{12}, v_{13}\}, \{v_{13}, v_{14}\}\}.\end{aligned}$$

This is E, the edge set of G. Each edge is represented by the pair of vertices it connects.

Now that we are familiar with the representation of graphs, we can formally define the graph colouring problem. Given a graph $G = (V, E)$, we want to assign each vertex $v \in V$ to an integer $c(v) \in \{1, 2, 3, ..., k\}$ such that the following two conditions are met:

- $c(a) \neq c(b) \forall \{a, b\} \in E$

For all pairs of integers a and b within the edge set, the colour of a cannot equal the colour of b. In other words, no two adjacent vertices in the graph may share the same number. Note that each vertex is being assigned a number rather than a colour. This is irrelevant to the problem, as assigning vertices the colours "red, green, and blue" is essentially the same as assigning them the numbers "1, 2, and 3". For example, $c(v_4) = 3$ indicates that vertex 4 is assigned the colour 3.

- The number of colours used, k, is as small as possible (this is called the chromatic number)

This problem calls for the number of colours used to be a minimum.

To illustrate this more clearly, the graph from Figure 1 has been coloured as shown below. Note that no two adjacent vertices share the same colour.

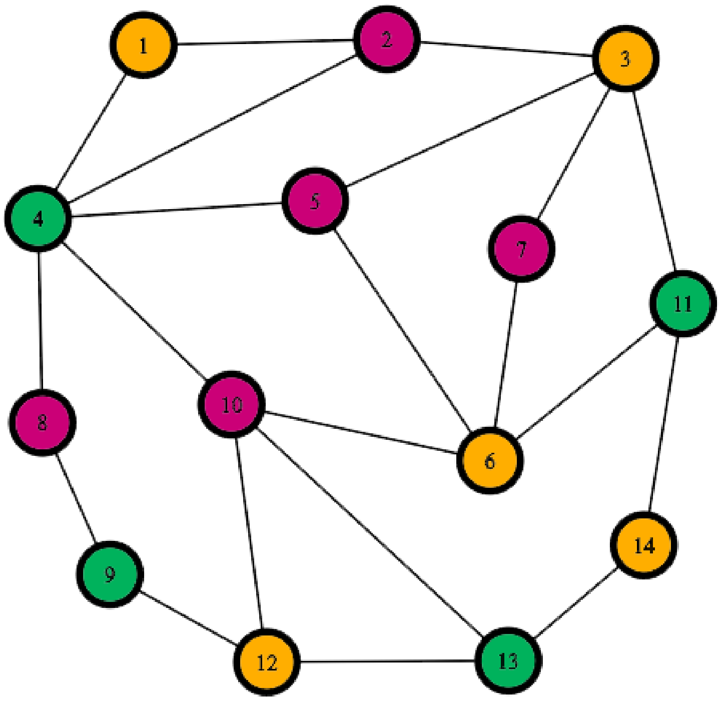


Figure 2: A complete, proper colouring of the graph from Figure 1

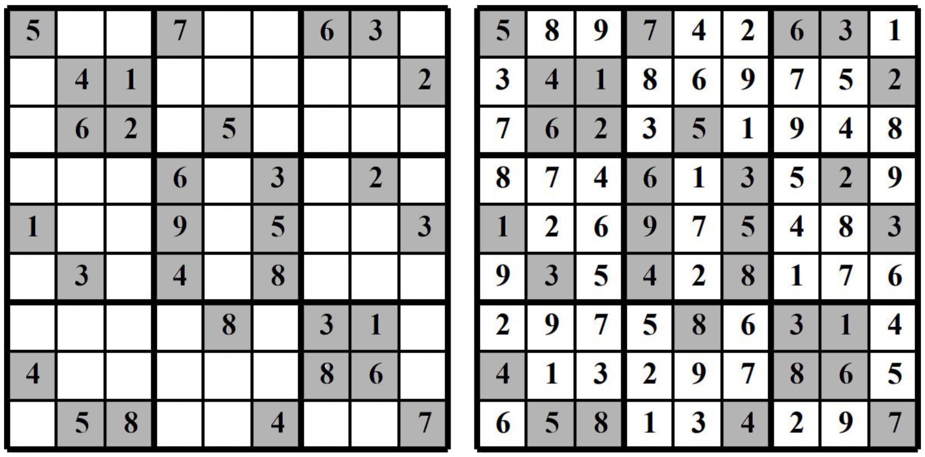
We can arbitrarily choose v_1's colour to be colour "1", v_2's colour to be colour "2", and v_4's colour to be colour "3". Now we can denote the solution to the colouring of this graph as:

$$c(v_1) = 1, c(v_2) = 2, c(v_3) = 1, c(v_4) = 3, c(v_5) = 2,$$
$$c(v_6) = 1, c(v_7) = 2, c(v_8) = 2, c(v_9) = 3, c(v_{10}) = 2,$$
$$c(v_{11}) = 3, c(v_{12}) = 1, c(v_{13}) = 3, c(v_{14}) = 1.$$

3 Application #1 - Sudoku

3.1 Introduction

The first real-life application I want to explore is solving Sudoku puzzles. Sudoku is a game in which, given a 9×9 board where some spaces may already be filled in, you want to fill each space with a digit such that each of the digits from 1 to 9 appear in every row, column, as well as the nine 3×3 subsquares. An example of a partially solved Sudoku board and its fully solved variant are shown below.



5			7			6	3	
	4	1						2
	6	2		5				
			6		3		2	
1			9		5			3
	3		4		8			
				8		3	1	
4						8	6	
	5	8			4			7

5	8	9	7	4	2	6	3	1
3	4	1	8	6	9	7	5	2
7	6	2	3	5	1	9	4	8
8	7	4	6	1	3	5	2	9
1	2	6	9	7	5	4	8	3
9	3	5	4	2	8	1	7	6
2	9	7	5	8	6	3	1	4
4	1	3	2	9	7	8	6	5
6	5	8	1	3	4	2	9	7

Figure 3: An incomplete Sudoku board and its complete solution

*Note: I will only be examining how to solve logic-solvable Sudoku puzzles. This means that the initially filled in numbers ensure there is only one solution, and that when solving by hand, no guesswork is required. In addition, all logic-solvable puzzles must have at least 17 initial filled in cells. There is a proof for this, however it will not be included as it is irrelevant to the aim of this essay (McGuire, 2013).

3.2 Modelling the Problem as a Graph

The first step to solving this problem is coming up with a way to represent it as a graph. The central idea of graph colouring is that connected vertices cannot share the same colour. We note that in Sudoku, any two boxes in the same row, column, or 3×3 square can't share the same number. Therefore, each cell will represent a vertex, and edges will connect cells in the same row, column, or 3×3 square. Finally, colours will represent the digits assigned to each cell.

3.3 Brute Force

A fairly naïve approach to solving the Sudoku problem computationally is to generate every possible board and check if it is correct (correct being every row, column, and 3×3 subsquare contains the digits 1-9). As mentioned above, we are only dealing with logic-solvable boards, so there will be at most $81 - 17 = 64$ empty cells. The total number of different ways to assign 9 colours (digits 1-9) to 64 vertices (cells) is:

$$9^{64} = 1.179 \times 10^{61} \text{ ways.}$$

Going back to the research question of this essay, we are mainly concerned with the efficiency of this method from a computational standpoint. It is generally accepted that modern computers can complete approximately 10^9 processes per second (University of Indiana, 2018). This means that the run-time for this type of program would take:

$$\frac{1.179 \times 10^{61}}{10^9} = 1.179 \times 10^{52} \text{ seconds, or}$$

$$\frac{1.179 \times 10^{52}}{60 \times 60 \times 24 \times 365} = 3.74 \times 10^{44} \text{ years.}$$

It is clear that this approach is far too slow to be solved in a reasonable amount of time by modern computing equipment. However, by examining the algorithm in more detail, we can find that certain aspects can be improved upon to increase its efficiency.

In analysing the brute-force method, we can see that many of the boards being checked are obviously incorrect. For example, these could include filling every cell with '1', or putting the same digit twice in a single row. By coming up with a way to ignore incorrect cases, a much more efficient solution can be found.

3.4 Backtracking

One possibility for a faster algorithm is through a backtracking method. The basic concept behind this method is to only continue forward filling in cells as long as all of the previously filled in cells don't break the board (where breaking the board is filling a cell such that there is already another cell of the same value in either the same row, column, or 3×3 subsquare). The idea behind this method is that we are only checking potentially correct solutions as opposed to every possibility (including obviously incorrect solutions) (Lee, 2008).

Outlined below are the steps taken to solve a Sudoku board using a backtracking algorithm. Note the use of the recursive function, *Solve*. This function first checks if the board is full. If this is the case, the board is solved, so it will return "true". Otherwise, it finds the first empty cell and tries assigning each digit 1-9 to it. Each digit that doesn't immediately break the board is assigned, and the function *Solve* calls itself and repeats this process again. At any instance where every digit is tried and none are valid, *Solve* will return false, and the previous iteration of *Solve* will continue trying new digits (Lee, 2008).

$$\begin{aligned} G =&\{v_{1,1}, v_{1,2}, \ldots, v_{9,8}, v_{9,9}\}, \\ &\{\{v_{1,1}, v_{1,2}\}, \{v_{1,1}, v_{1,3}\}, \ldots, \{v_{1,1}, v_{1,8}\}, \{v_{1,1}, v_{1,9}\}, \\ &\{v_{1,1}, v_{2,1}\}, \{v_{1,1}, v_{3,1}\}, \ldots, \{v_{1,1}, v_{8,1}\}, \{v_{1,1}, v_{9,1}\} \ldots\}. \end{aligned}$$

$$c(v_{1,1}) = 0,\ c(v_{1,2}) = 0, \ldots, c(v_{9,8}) = 0,\ c(v_{9,9}) = 0.$$

We start by initializing the graph. A colour of "0" indicates that a cell is unassigned.

```
$Solve$    {
    if $c(v_{a,b}) \neq 0 \forall a, b,$ return true
    otherwise:
    $v_{r,c}$ = first unassigned cell
    loop for $i$ from $1 \to 9$    {
        if $c(v_{d,e}) \neq i \ \forall \ d, e$ such that $\{v_{r,c}, v_{d,e}\} \in E$    {
            $c(v_{r,c}) = i$
            if $Solve$ = true: return true
            otherwise: $c(v_{r,c}) = 0$
        }
    }
    return false
}
```

3.5 Run-Time Analysis - Backtracking

At first, it might seem like this algorithm would take the same amount of time as the brute force one mentioned above, as it still iterates through each possible colour for each vertex. However this is not actually the case. To explain, we will assume the worst possible scenario - only 17 previously filled in cells.

We can start by acknowledging the position of the filled in cells. Since there are 17, the average number of filled-in cells per row is $17/9 = 1.\dot{8}$. This will be the same value for the average number of filled-in cells per column and per 3×3 subsquare. For simplicity, we can round this to 2, as we are looking for the general order of magnitude of the run-time, not an exact value.

Next, we can consider the following three cases that show the relationship between the rows, columns, and 3×3 subsquares when filling an arbitrary cell.

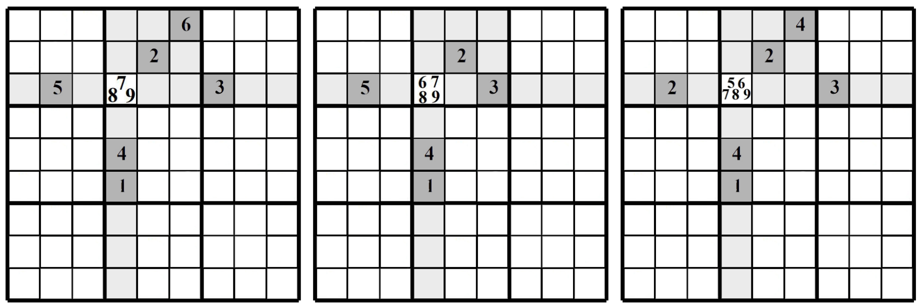


Figure 4: Three different cases when determining all possible values for a cell

In the first case (leftmost board), each filled-in cell in the row, column, and 3×3 subsquare contains a unique value. There are three possible values for the empty cell in question. In the second case (middle board), the cell with value "3" is in both the row and subsquare of the cell in question, effectively increasing the number of possibilities to four. In the final case (rightmost board), the filled-in values are not unique, so there are five possibilities.

Therefore, when calculating the run-time, we will ignore filled-in cells in the column and subsquare, as in the worst case, these will contain the same values as those in the row, thus having no effect on the number of possibilities for each cell in question.

We begin by looking at the topmost row, and the number of possible values for each cell.

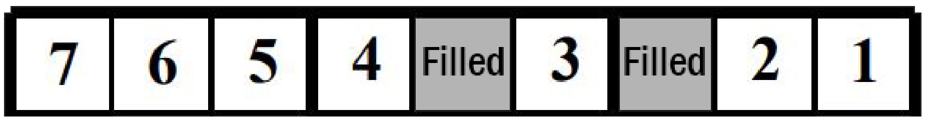


Figure 5: The number of ways to fill in each cell in the top row of a Sudoku board

Out of the 9 digits, 2 are already taken by the filled-in values, so the first square can take on 7 possible values. Using the same logic, the next square has 6 possibilities, and so on. Therefore, the number of ways to assign values to the first row only is $7 \times 6 \times \cdots \times 2 \times 1 = 7! = 5040$ ways. From here, the exact same logic can be applied to the second row, however, we must also account for the first row already being filled in. Therefore, we will have one less way of filling in each cell in the second row, which means the total number of ways of filling the second row is $6! = 720$ ways. As we continue down the rows, we can also continue this pattern and eventually find the overall run-time of this algorithm.

$$\begin{aligned}\prod_{a=1}^{7} a! &= 125411328000 \\ &\approx 1.25 \times 10^{11} \text{ processes.}\end{aligned}$$

$$\begin{aligned}\text{time} &= \frac{1.25 \times 10^{11}}{10^9} \\ &= 125 \text{ seconds.}\end{aligned}$$

There is a clear improvement from the brute force method, as now the algorithm will execute in the order of seconds, as opposed to 10^{44} years.

The overall run-time of this backtracking approach is still exponential in nature, however, the relatively small size of a standard Sudoku board allows it to be quite fast. However, if we extend the problem to a 16×16 board, we can see that even slightly larger values can increase the run-time by several orders of magnitude.

While it is not proven as the lower bound, it is estimated that 55 squares on a 16×16 board have to be filled in for the puzzle to be logic-solvable (McGuire, 2013). Therefore, the run-time can be calculated as follows:

$$\begin{aligned}\text{filled-in cells per row} &= 55/16\\ &= 3.4375\\ &\approx 3\end{aligned}$$

$$\begin{aligned}\text{empty cells per row} &= 16 - 3\\ &= 13\end{aligned}$$

$$\prod_{a=1}^{13} a! \approx 7.93 \times 10^{53} \text{ processes.}$$

$$\begin{aligned}\text{time} &= \frac{7.93 \times 10^{53}}{10^9}\\ &= 7.93 \times 10^{44} \text{ seconds.}\end{aligned}$$

Clearly, this is far too slow to be considered feasible by modern computing equipment.

4 Application #2 - Aircraft Scheduling

4.1 Introduction

Another practical application that I want to explore is the aircraft scheduling problem. This is an example of a resource allocation problem. It deals with assigning airplanes (the resources) to certain flights, where there are certain constraints that must be met. The general problem statement is as follows:

> Suppose you have x aircraft and you are tasked with assigning them to n flights, where each flight happens during a certain time interval. If two of these intervals overlap, you cannot assign the same aircraft to both. Given n, the number of flights, and each flight's time interval (start and end time), what is the minimum value of x, the number of aircraft needed (Marx, 2003)?

4.2 Modelling the Problem as a Graph

The relation between the aircraft scheduling problem and graph theory - more specifically, graph colouring - may not be evident at first. However, after examining the problem, a connection can be made. We can let the vertices represent each flight, and the edges will connect any pairs of flights whose time intervals intersect. Finally, the vertex colours will represent each aircraft. We can see exactly how this works by going back to the two conditions of graph colouring:

- No two adjacent vertices share the same colour (if flight time intervals overlap, then they require different aircraft. The same aircraft cannot be used for two simultaneous flights)
- The number of colours used is minimal (we are looking to find the minimum number of aircraft needed)

4.3 Brute Force

As with all graph colouring problems, there exists a naïve brute-force method that evaluates every possibility to find a solution. It is first noted that the maximum number of aircraft needed will always be less than or equal to the number of flights. Then, it is a matter of trying every combination of assigning n aircraft to n flights, and out of the valid options, returning the option that utilizes the least number of planes. Even for small values such as $n = 15$ it would take:

$$15^{15} = 4.38 \times 10^{17} \text{ processes,}$$

which is far too slow given our previous limit of 10^9 processes per second.

4.4 Greedy Algorithm

To improve the efficiency of this algorithm, we must examine the resulting graph for any unique properties. We can do this by creating a sample test case. Recall n is the number of flights, and we are trying to minimize x, the number of aircraft needed. We can define S as the set of flight intervals, where $S_a = \{\text{start of } a^{th} \text{ interval, end of } a^{th}\text{interval}\}$.

$$\text{Let } n = 8, \text{ and } S = \{\{0,6\},\{1,4\},\{3,5\},\{3,8\},\{4,7\},\{5,9\}\,\{6,10\},\{8,11\}\}.$$

We must first sort S by the starting time of each interval. In this case S has been given in sorted order for the sake of simplicity. From here, it helps to visualize the problem by plotting each interval on a timeline, as shown below.

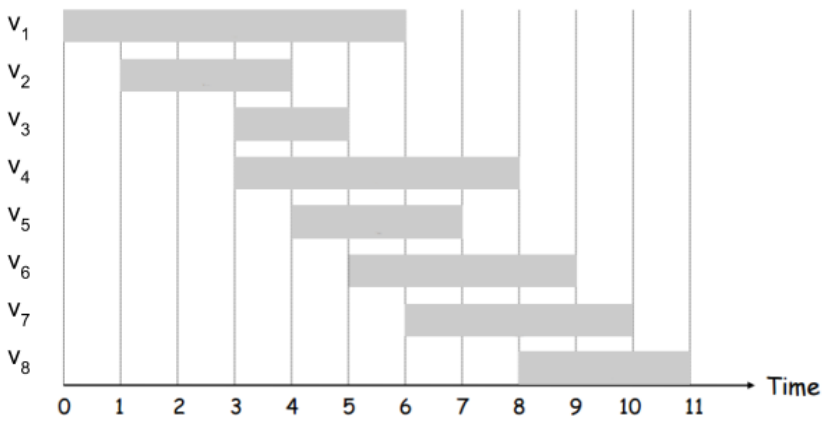


Figure 6: The intervals of each flight plotted together on a timeline

Figure 6 shows an alternative way to visualize the graph. Each bar is a node, and nodes share an edge if and only if there exists a time that is part of both intervals. This is known as an interval graph (Marx, 2003). The basis of colouring this graph uses a greedy algorithm - a method of making an optimal choice at any given moment without considering how it affects later decisions (Black, 2005). The steps to do so are outlined below.

$$\begin{aligned} G =&\{v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8\},\\ &\{\{v_1,v_2\},\{v_1,v_3\},\{v_1,v_4\},\{v_1,v_5\},\{v_1,v_6\},\{v_2,v_3\},\\ &\{v_2,v_4\},\{v_3,v_4\},\{v_3,v_5\},\{v_4,v_5\},\{v_4,v_6\},\{v_4,v_7\},\\ &\{v_5,v_6\},\{v_5,v_7\},\{v_6,v_7\},\{v_6,v_8\},\{v_7,v_8\}\}. \end{aligned}$$

We start by initializing G by its vertex and edge lists.

$$k = 1,\ c(v_1) = 1$$

In the initial state, we assume only v_1 is in the graph. The initial chromatic number, k, is 1, and v_1 has colour 1.

$$\because \{v_1, v_2\} \in E,\ k = k + 1,\ c(v_2) = 2$$

Since the intervals v_1 and v_2 overlap, we need a new colour for v_2 (colour 2). The chromatic number of the current state is now 2.

$$c(v_3) = 3,\ c(v_4) = 4$$

The same logic can be applied to v_3 and v_4. They both overlap every interval before them, so they both need new colours.

$$\because \{v_2, v_5\} \notin E,\ c(v_5) = 2$$

v_5 overlaps intervals with colours 1, 3, and 4, but not 2. Therefore, v_5 itself can be coloured with colour 2.

$$c(v_6) = 3,\ c(v_7) = 1,\ c(v_8) = 2$$

The same logic can be applied to the next three nodes. They can be coloured with the existing colours of the nodes they do not overlap with.

Finally, we are left with an optimally coloured graph, as well as a chromatic number of 4 (University of Toronto, 2015). Therefore, for this problem, four aircraft are needed to fill each flight interval.

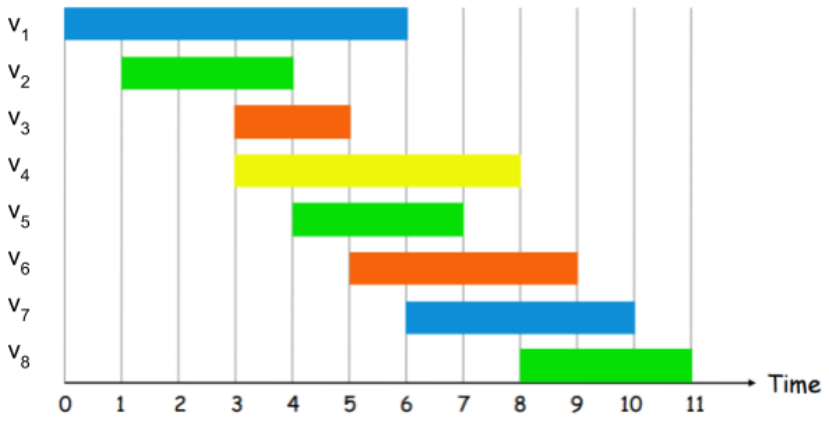


Figure 7: A complete, coloured version of the interval graph from Figure 6

4.5 Why Does A Greedy Approach Work?

To show why this algorithm works, we can use a proof of optimality - a technique that finds some bound which all solutions must satisfy, and then shows how this relates to the algorithm. Consider a point in time at which the number of intersecting intervals is a maximum. Since all of these intervals exist simultaneously, they each must have their own colour. Then, the number of colours here is the theoretical lower bound on the chromatic number of the graph. From here, if we can prove that the algorithm never uses more colours than this lower bound, the proof is complete.

Recall that vertices are sorted by the start times of their intervals. Consider the first point in time at which the number of intersecting intervals is a maximum. The algorithm would initialize the maximum number k, of colours at this moment. After this, each interval will intersect a maximum of $k-1$ other intervals, meaning there will always be one colour left to assign to it, without the need to use more than k colours in total (University of Toronto, 2015). $\square$

4.6 Run-Time Analysis - Greedy Algorithm

There are several independent steps in this algorithm, so when calculating its efficiency, they must all be acknowledged. First of all, we have to sort the given list S. There are several well-known sorting algorithms, but one of the most efficient is *quicksort*. Its details will not be covered as they are irrelevant to the aim of this essay, but it should be known that for a list of n elements, *quicksort* will use $n \times \log_2 n$ operations (Knuth, 1997). After this, we have to transform the interval list into an interval graph. To do this, we compare every pair of intervals, and add an edge to the graph if they overlap. In a list of n elements, evaluating every pair will take n^2 operations. Finally, the algorithm compares every vertex v_i to every one of its adjacent vertices v_j such that $j < i$. In the worst case, each vertex will be connected to every other one, so the total number of comparisons here will be:

$$n + n - 1 + n - 2 + \cdots + 2 + 1 = \frac{n(n+1)}{2}$$

Thus, there will be a total of:

$$n \log_2 n + n^2 + \frac{n(n+1)}{2} \text{ operations.}$$

Below is a graph that shows the number of operations (y-axis) for varying values of n (x-axis). We can use it to determine the maximum bound on n at which this algorithm will run efficiently.

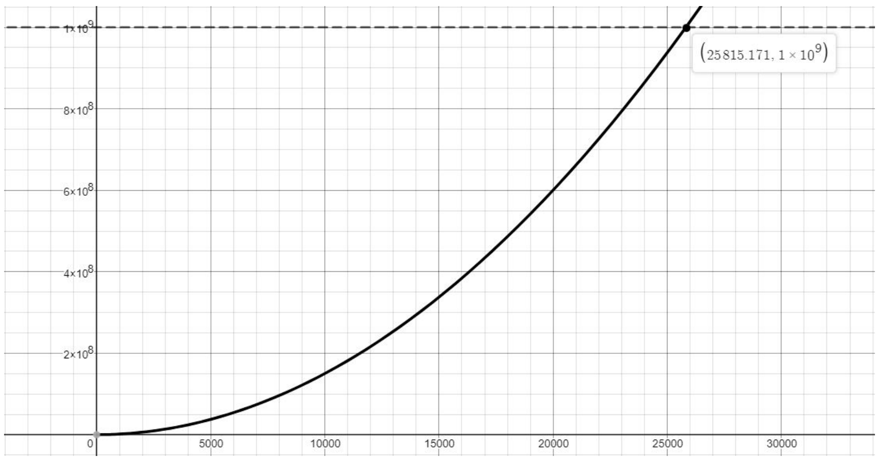


Figure 8: A graph showing the relationship between the value of n and the number of processes in the algorithm, as well as the value of n for 10^9 processes

Comparing the greedy approach to the brute-force method, it is clear that there is a significant improvement. $n = 15$ was subject to a combinatorial explosion when brute-forcing, while the greedy algorithm can process $n = 25,000$ in under one second.

5 Conclusion

We must reference the original research questions to determine what conclusions can be drawn.

To what extent can concepts from vertex graph coloring be used to efficiently solve real-life scheduling and allocation problems?

While the execution time for the Sudoku backtracking algorithm was exponential in nature, the relatively small board size allowed for a fast solution. It is important to note that this method can be applied to any graph colouring problem, and that it is not exclusive to just Sudoku. Therefore, in terms of generally solving scheduling and allocation problems, backtracking will not always provide an efficient solution, however, with small numbers, it proves to be much more effective than a typical brute-force method.

With the aircraft scheduling problem, we were able to identify some unique properties of interval graphs that allow for a colouring in polynomial (n^2) time. This is one of very few cases of graph colouring that has an efficient solution, and it is interesting how we were able to incorporate the study of greedy algorithms, a seemingly unrelated field to that of graph theory.

There are several areas that could potentially be explored as extensions to this paper, or as standalone research papers. Among them are problems such as creating timetables or designing seating plans, where many more factors are present, including mandatory restrictions as well as optional but beneficial ones. In these cases, it might be appropriate to explore the application of an entirely separate set of methods including heuristics and approximation algorithms.

References

Black, P. (2005). *Greedy Algorithm - Dictionary of Algorithms and Data Structures.* Xlinux.nist.gov. Retrieved 8 September 2019 from `https://xlinux.nist.gov/dads//HTML/greedyalgo.html`

Knuth, D. (1997). *The Art of Computer Programming Volume 1* (3rd ed). Redwood, VIC: Addison-Wesley.

Lee, C. (2008). CS106B: Programming Abstractions, lecture 12 notes [PowerPoint slides]. Retrieved 8 September 2019 from `http://web.stanford.edu/class/cs106b/lectures/12/12-lecture.pdf`

Lewis, R.M.R. (2016). *A Guide to Graph Colouring: Algorithms and Applications.* Cardiff, VIC: Springer International Publishing.

Marx, D. (2003). *Graph Colouring Problems and Their Applications in Scheduling.* Citeseerx.ist.psu.edu. Retrieved 18 August 2019 from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.4268&rep=rep1&type=pdf`

McGuire, G., Tugemann, B., & Civario, G. (2013). *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem.* arXiv.org. Retrieved 27 October 2019 from `https://arxiv.org/abs/1201.0749`

Paoletti, T. (2011). *Leonard Euler's Solution to the Konigsberg Bridge Problem.* Maa.org. Retrieved 29 July 2019 from `https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem-konigsberg`

University of Indiana. (2018). *What is a computer's clock speed?.* Kb.iu.edu. Retrieved 20 September 2019 from `https://kb.iu.edu/d/aekt`

University of Toronto. (2015). *Greedy Interval Colouring Algorithm.* Cs.toronto.edu. Retrieved 7 September 2019 from `http://www.cs.toronto.edu/~bor/373f11/L4-373f11.pdf`