Project 1: Graph Chromatic Number

Tiago Fernandes, 88784

***Abstract* – This project addressed the problem of finding the chromatic number of a graph, which is an NP-problem in the area of graph colouring. The first approach used was exhaustive search, which is guaranteed to find the optimal solution but, because of its exponential growth, is impossible to apply to bigger problem instances. The other approach was a greedy heuristic, that despite not always returning the correct result, showed a precision of 97.7% and was able to solve all problems in polynomial time, allowing to approximately solve very big problem instances without difficulty.**

***Keywords* – Graph colouring, Chromatic number, Exhaustive search, Greedy heuristic.**

I. INTRODUCTION

Graph colouring is a type of problem that has been studied since the 19th century, when de Morgan and Hamilton discussed the problem of finding the minimum number of colours needed to fill in any map on the plane. Since then, a lot of theoretical efforts have been devoted to this problem, which has been proven to be NP-hard, that is, at least as hard as any NP-problem. Nowadays, graph colouring remains a popular problem, due to its theoretical challenges and practical applications. [1]

In this project, vertex colouring, which is the most popular case of graph colouring, is discussed. Here, the problem is to colour the vertices (or nodes) of a graph such that no two adjacent nodes are of the same colour. More specifically, the task is to find the chromatic number of a given undirected graph, which is the smallest number of colours needed to properly colour its vertices.

Vertex colouring is very useful in practice. For instance, the predominant approach to solve register allocation in compilers is to use vertex colouring. The registers that are candidates for allocation are represented as nodes in the graph, while edges connect the registers that are simultaneously live at at least one program point. Using this representation, register allocation reduces to the graph colouring problem in which colours (representing registers) are assigned to the nodes such that two connected nodes do not receive the same colour. [2] A more mainstream instance of the problem is the famous number puzzle Sudoku, which can be seen as a graph colouring problem where each cell is a node in the graph and is connected to all the other cells in the same row, column or block. Using this representation, solving the Sudoku puzzle is the same as constructing a 9-colouring of a particular graph, given a partial 9-colouring.

Vertex colouring, in particular computing the chromatic number of a graph, is an NP-complete problem, as mentioned before. This means that it is a computationally hard problem to solve, and there is no polynomial algorithm to solve it. For this reason, algorithms that can yield approximate results much faster than exact algorithms like exhaustive search can be very useful. Thus, both the exhaustive search approach and an algorithm using a greedy heuristic are designed, tested and compared in this project.

II. GRAPH GENERATION

In order to apply and test the algorithms developed, a dataset containing graph instances was created. To study the influence of the increase of the number of nodes, the number of nodes was successively increased from 2 to 20 nodes. Also, for each number of nodes, three random graphs, with a different ratio of edges created - approximately 25%, 50% and 75% of the maximum number of edges were generated.

The networkx module [3] was used as a backend for most graph operations, including generation and drawing. For the random graph generation, the Erdős-Rényi model [4] was chosen, as it is a simple yet popular graph generation algorithm. In this model, defined by the parameters (n, p), n nodes are generated, and each possible edge has a probability p of being created. To ensure that a fully connected (without isolated nodes or more than one connected component) graph, edges between random nodes of different connected components were created until the graph was fully connected, taking advantage of the appropriate networkx functions.

Since the graphs vertices are 2D points in the cartesian plane, n pairs of integer coordinates between 1 and 9 were randomly generated. Each new random pair was only accepted if it was different than the other coordinates and not too close (horizontally and vertically adjacent coordinates were rejected). To guarantee reproducibility, a random seed equal to the author's student number (88784) was used, and the adjacency lists and and coordinates of the generated graphs stored in pickle files. An example of the instances generated is shown in Fig. 1.

It turned out that the ratio of created edges was about 5% lower than expected in most graphs, but that can be explained by the probabilistic nature of the graph generation algorithm, as another random seed (88874) yielded opposite results.

III. EXHAUSTIVE SEARCH ALGORITHM

The first algorithm design strategy implemented was exhaustive search, which consists of generating all possible candidates for the solution and checking whether each candidate satisfies the problem's constraints.

The first algorithm, *chromatic_number_exhaustive_v1*, consisted of generating all the possibilities of colouring the n nodes with n or less colours. This was achieved using the

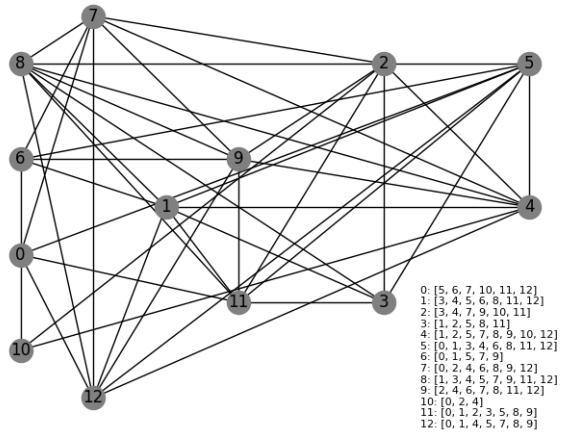


Fig. 1: Example of a generated graph, with $n = 13$ and $p = 0.50$. The corresponding adjacency list is represented in the bottom right.

itertools.product() iterator to perform the cartesian product of an colour array by itself n times. Colours were represented as non-negative integers, so the colour array was an array containing the first n non-negative integers in ascending order. Thus, each possible configuration is an array of n colours, were the n^{th} element is the colour of the n^{th} node.

After creating an iterator with all possible colour configurations, the algorithm iterates through all the graph's nodes and compares the node's colour with the one of each of its neighbours. If it finds a neighbour with the same colour, the colour configuration is deemed invalid and the algorithm skips to the next configuration in the iterator. On the other hand, if the colours are different, the algorithm evaluates the next node, until all nodes are traversed. If the algorithm hasn't found conflicting colours after evaluating the last node, the colour configuration is deemed valid. After finding a valid configuration, the algorithm finds the corresponding chromatic number (the number of different values in the configuration), which is stored if it is lower than the best chromatic number found up to that point.

This algorithm has two obvious drawbacks, the first one being that it distinguishes between different colourings of the same pattern, when the particular colour in each node is not relevant; for example, $(0, 0, 1)$ and $(1, 1, 0)$ are viewed as different configurations, when in fact they are just two equivalent possibilities of colouring the first and second nodes with the same colour and the third node with a different one. This means that the algorithm tests many unnecessary possibilities. To address this, a new similar algorithm, *chromatic_number_exhaustive_v2*, was developed. The only difference from the previous is that it fixes the colour of the first node when generating the colour configurations, thus reducing by a factor of n the number of configurations tested.

The pseudocode for the first versions of the exhaustive search algorithm is shown in Alg. 1, which uses the auxiliary function valid_config defined in Alg. 2. Note that colour_combinations only includes configurations where the colour of the first node is 0.

Algorithm 1 chromatic_number_exhaustive_v2

```
chromatic_number ← inf
colour_combinations ← iterator w/ colour configs.
for colour_config in colour_combinations do
    valid ← valid_config(colour_config)
    if valid then
        if n_colours < best_chromatic_number then
            chromatic_number ← n_colours
            best_colours ← colour_config
return chromatic_number, best_colours
```

Algorithm 2 valid_config

```
for node in adjacency list do
    for each node neighbour do
        if colour of node = colour of neighbour then
            return False
return True
```

The other obvious drawback is that the previous algorithm can't stop after finding a valid configuration and needs to determine and compare the chromatic number of each valid configuration. To deal with this issue, a slightly different exhaustive algorithm, *chromatic_number_exhaustive_v3*, was implemented. In this version, the algorithm fixes the number of colours used and tries to find a valid configuration. If such configuration is found, it is necessarily an optimal solution and the chromatic number the number of colours. The pseudocode for this algorithm is shown in Alg. 3. Note that colour_combinations only includes configurations where the colour of the first node is 0.

Algorithm 3 chromatic_number_exhaustive_v3

```
for n_colours = 1, 2, ..., n do
    col_combs ← iterator w/ colour configs (≤ n_colours)
    for colour_config in col_combs do
        valid ← valid_config(colour_config)
        if valid then
            chromatic_number ← n_colours
            best_colours ← colour
            return chromatic_number, best_colours
```

A. Formal complexity analysis

In this section, a brief formal analysis of the time complexity of the two exhaustive algorithms tested is presented. These results will be given in relation to the number of nodes n and the probability of edge creation p, discussed in Section II.

Since the auxiliary function *valid_config* is common to both algorithms (and also the greedy algorithm), the complexity of this function is discussed before delving into the full algorithms. This function iterates through the n nodes, and for each node through all its neighbours, until it finds conflicting colours. So, in the best case, where it finds conflicting colours in the first neighbour of the first node, it has $\mathcal{O}(1)$ complexity. When there is no colour conflict, or it is

in the last neighbour of the last node, it needs to go through all n nodes and, on average, $p \times n$ neighbours for each node, totalling $n \times pn$ operations, which is $\mathcal{O}(n^2)$.

In the case of the first exhaustive algorithm, *chromatic_number_exhaustive_v2*, the algorithm iterates through n^{n-1} colour configurations, since each of the last $n-1$ nodes can have n possible colours. For each colour configuration, it calls the function *valid_config*, so in the best case the algorithm has complexity $\mathcal{O}(n^{n-1}) \times \mathcal{O}(1) = \mathcal{O}(n^{n-1})$ and, in the worst case, $\mathcal{O}(n^{n-1}) \times \mathcal{O}(n^2) = \mathcal{O}(n^{n+1})$.

As to the second algorithm, *chromatic_number_exhaustive_v3*, the algorithm iterates through at most n colours, and for each current number of colours c, at most n^{c-1} colour configurations for each (in the worst case, the algorithm checks $\sum_{c=2}^{n} n^{c-1}$ different configurations). For each colour configuration, it calls the function *valid_config*, and it stops if this function returns True. In the best case the algorithm has complexity $\mathcal{O}(n^2)$, as it is possible that it stops after verifying that the first colour configuration is valid. In the worst case, it needs to check all the configurations with up to n colours and, for each configuration, it only finds a conflict in the last neighbour of the last node, which gives $\mathcal{O}(n^{n-1}) \times \mathcal{O}(n^2) = \mathcal{O}(n^{n+1})$ complexity.

B. Testing

Both algorithms were run in the previously generated graph instances, in increasing order of number of nodes, until they took more than one minute to solve a particular instance. The chromatic number and colour configuration were saved, as well as the running time, number of configurations tested, and number of comparisons for each instance were saved. The drawing using the colours obtained by applying the second algorithm to the instance shown in Fig. 1 are shown in Fig. 2.

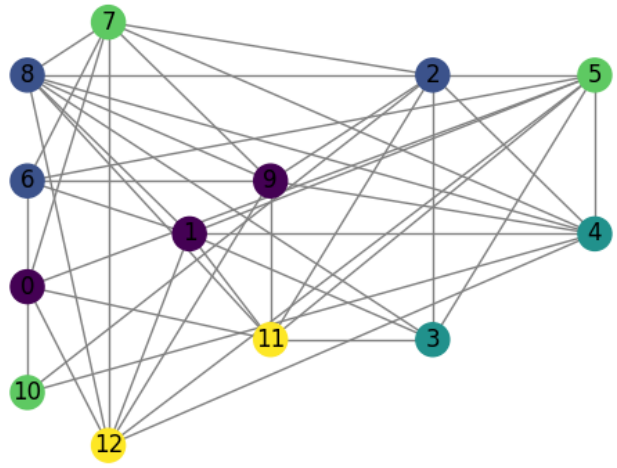


Fig. 2: Possible solution of the problem instance shown in Fig. 1, obtained using algorithm *chromatic_number_v3*.

C. Empirical complexity analysis

The results for the number of configurations tested, number of comparisons and the execution time for *chromatic_number_exhaustive_v2* are plotted in Fig. 3. It can be observed that all the three plots follow the same general trend. Even in the logarithmic scale, the growth seems to be exponential, which points to a super-exponential growth, as expected from the formal analysis.

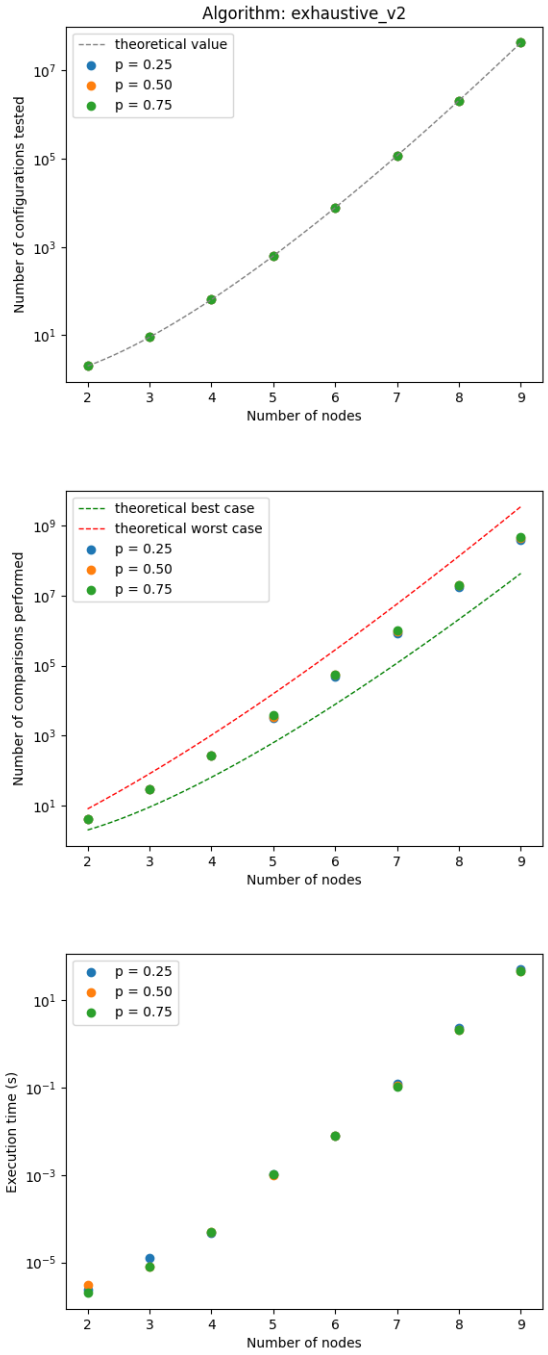


Fig. 3: Empirical results for the *chromatic_number_exhaustive_v2* algorithm. Note that the y-axis is in logarithmic scale in all plots.

It is confirmed that the number of configurations tested is always n^{n-1}, and it can be seen that the number of basic operations performed is between the worst case and the best cases predicted in the formal analysis. It can also be noted

that the density of the edges does not seem to affect the execution time.

Moreover, a regression was performed on the execution time data for the 50% edge density ($p = 0.5$), using the equation $t = an^{n+b}$ (the same form as both the worst and best cases determined in the formal analysis), where t is the execution time, n continues to be the number of nodes, and a and b are the parameters to be found. The curve fitting yielded $t = 4.6 \times 10^{-7} \times n^{n-0.62}$, which agrees with the number of basic comparisons and confirms the super-exponential growth of the execution time of this algorithm, that is, $\mathcal{O}(n^n)$ complexity for the average case. This equation also allows to predict the execution time for larger problem instances. For example, for $n = 10$ nodes, the algorithm would take approximately 18 minutes, while for 20 nodes it would take more than 7×10^{18} seconds, which is more than the current age of the Universe [5].

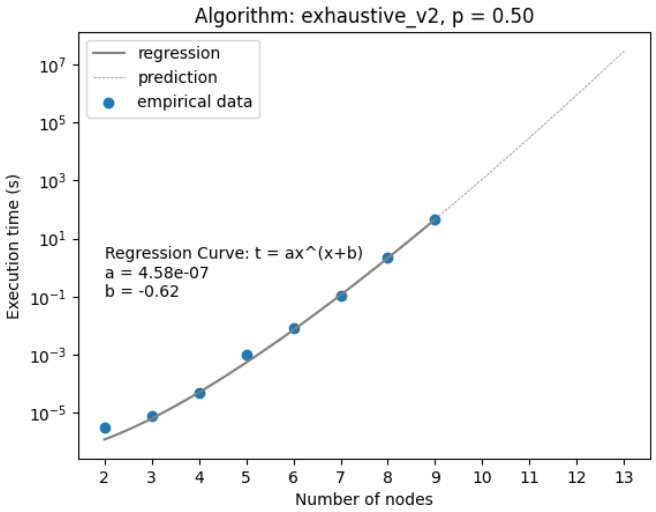


Fig. 4: Regression and prediction for the running time of the algorithm *chromatic_number_exhaustive_v2*.

The results for the number of configurations tested, number of comparisons and the execution time for *chromatic_number_exhaustive_v3* are plotted in Fig. 5. It can be observed that for all the three plots, the growth of the logarithm of the execution time is higher than linear. In this case, the ratio of edges created has more influence in the results, as the points with different p are no longer overlapping.

It can be seen that the algorithm always tests less configurations than the previous version, which confirms the benefit of the approach of testing a low number of colours and only increasing it if needed. It can also be observed that the number of operations is between the best case and worst case bounds obtained in the formal analysis.

As done for the previous algorithm, a regression was performed on the execution time data for the 50% edge density ($p = 0.5$). Initially, the same equation as the previous algorithm was used, but no good fit could be achieved. Eventually, a good fit was achieved using the equation $\ln t = an^2 + bn + c$, equivalent to $t = \exp(an^2 + bn + c)$,

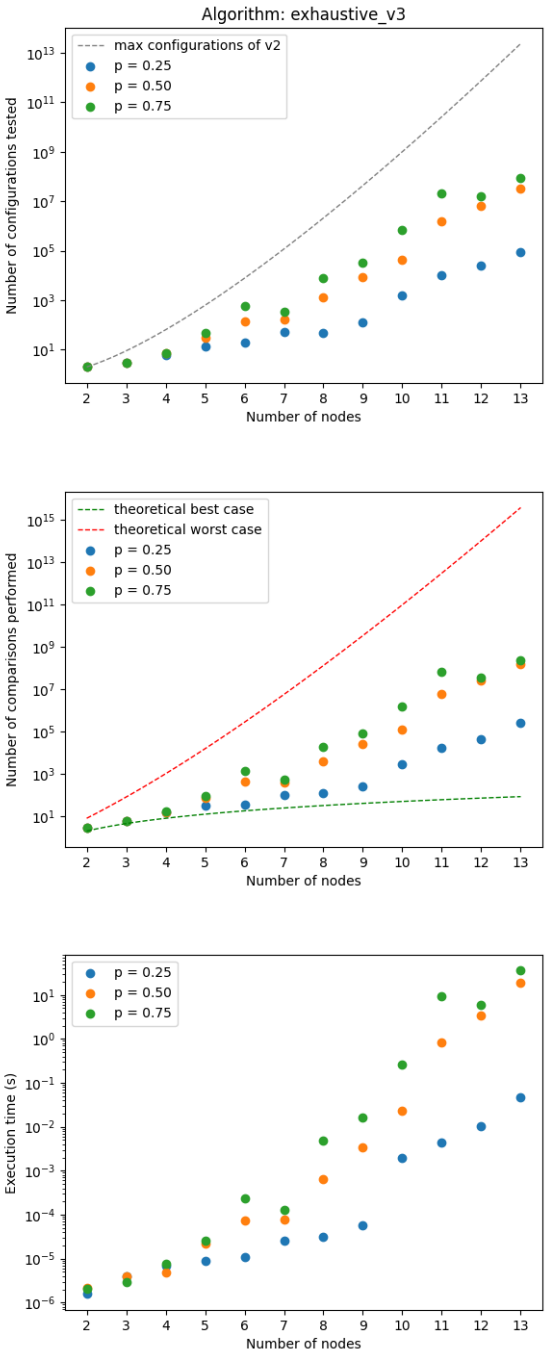


Fig. 5: Empirical results for the *chromatic_number_exhaustive_v3* algorithm. Note that the y-axis is in logarithmic scale in all plots.

where t is the execution time, n continues to be the number of nodes, and a, b and c are the parameters to be found. The curve fitting yielded $\ln t = 0.11n^2 - 0.09n - 13.27$. Thus, the average case complexity of the algorithm seems to be $\mathcal{O}(2^{\text{poly}(n)})$ complexity for the average case. Comparing with the previous algorithm, the *v3* version is much better, since a 10 node instance could be solved in 0.04 seconds (instead of the predicted 18 minutes). Nevertheless, a 20 node instance would take more than 3×10^{12} seconds,

which despite being 7 orders of magnitude as the result for the previous algorithm, is the typical lifetime of a biological species [5].

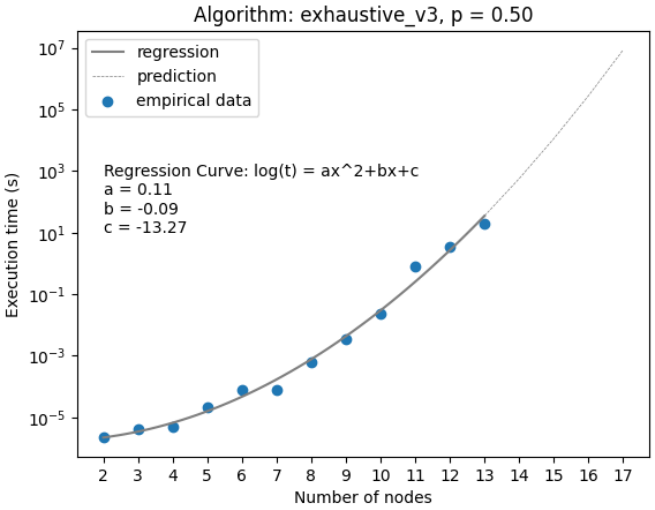


Fig. 6: Regression and prediction for the running time of the algorithm *chromatic_number_exhaustive_v3*.

IV. GREEDY HEURISTIC

In this section of the project, an algorithm based on a greedy heuristic was studied. This type of algorithm follows the problem-solving heuristic of choosing the locally optimal solution at each step, which in many problems can fail to produce an optimal global solution. On the other hand, only one configuration is tested, since the solution is incrementally built.

A very simple greedy heuristic would be to randomly choose a vertex for colouring and then assign it the smallest possible colour that is valid (not shared by a neighbour node). Keeping in mind that usually vertices with a lower degree (number of neighbours) allow for a more flexible choice of colour, a simple yet powerful improvement of this heuristic is sorting the nodes in non-increasing order of degree (number of neighbours). This is known as the largest-first (LF) method. [6]

The latter heuristic was then implemented as shown in Alg. 4.

A. Formal complexity analysis

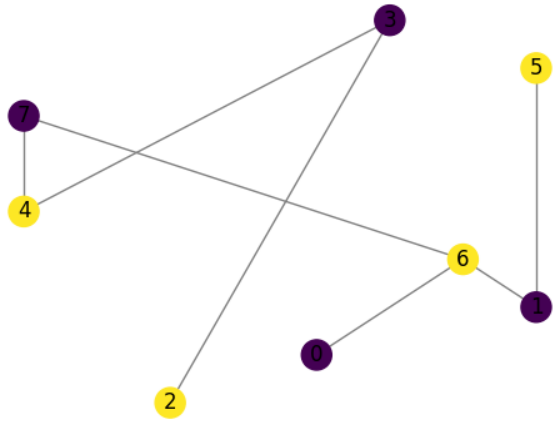
The algorithm starts by sorting the nodes in non-increasing order, using Python's built-in sort() function, which has time complexity of $\mathcal{O}(n \log n)$ [7]. Then, it iterates through every node, which is $\mathcal{O}(n)$. For each node, the algorithm iterates through all its neighbours and then it iterates through all the node's neighbours ($\mathcal{O}(1)$ in the best case and $\mathcal{O}(n)$ in the worst case) and, after that, from 1 to up to n colours.

Thus, in the best case, it has time complexity of $\mathcal{O}(n \log n)+\mathcal{O}(n)*(\mathcal{O}(1)+\mathcal{O}(1)) = \mathcal{O}(n \log n)+\mathcal{O}(n) = \mathcal{O}(n \log n)$. In the worst case, the algorithm has complexity of $\mathcal{O}(n \log n) + \mathcal{O}(n) * (\mathcal{O}(n-1) + \mathcal{O}(n)) \simeq \mathcal{O}(n \log n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$.

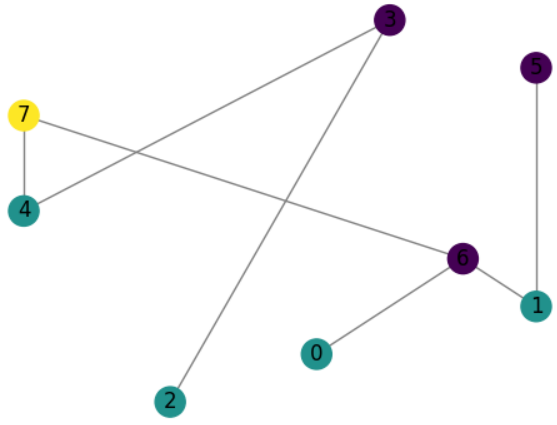
Algorithm 4 chromatic_number_greedy

```
nodes ← nodes sorted in non-increasing order of degree
colours ← empty dictionary
for node in adjacency list do
    neighbour_colours ← empty set
    for each node neighbour do
        if neighbour in colours then
            Add neighbour colour to neighbour_colours
    for colour = 0, 1, ..., n-1 do
        if colour not in neighbour_colours then
            colours[node] ← colour
            break
chromatic_number ← # of different values in colours
return chromatic_number, colours
```

B. Testing



(a) Exhaustive search solution, with chromatic number of 2.



(b) Greedy heuristic solution, with chromatic number of 3.

Fig. 7: Example of graph instance where the greedy heuristic yielded a wrong result. In this case, it needed one more colour than necessary to colour the graph.

The algorithm was run using all the generated graph instances, and the results were saved similarly as done with the exhaustive algorithms.

In order to evaluate the effectiveness of the greedy heuristic, the solutions obtained were compared to the exact solu-

tions provided by the brute force algorithms. It was found that the greedy algorithm achieved 97.7% of precision, failing to find the best solution in only 3 of the 39 test cases, and with a difference of 1 colour in all the wrong cases. One such case is shown in Fig. 7.

C. Empirical complexity analysis

Finally, the empirical complexity results were analyzed. The results for the number of membership checks and for the execution time are shown in Fig. 8. Both the number of basic operations and the execution time grow similarly, as expected. This growth seems to be polynomial, instead of exponential or super-exponential, as seen previously. Comparing the execution time with the previous algorithms, the present algorithm is much faster, with run times in the order of the milliseconds, even for instances with 20 nodes, which could never be solved by the exhaustive search method.

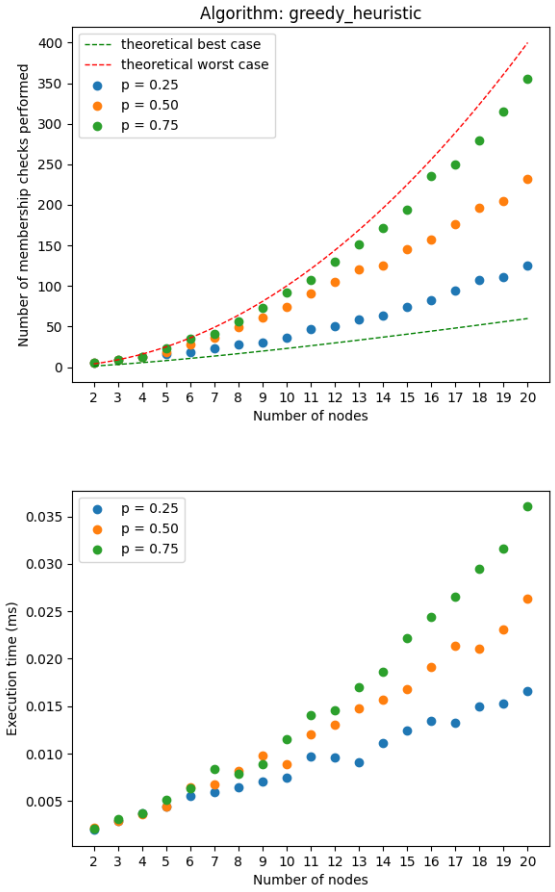


Fig. 8: Empirical results for the *chromatic_number_greedy* algorithm. Note that for this algorithm the y scale is linear, and the time given in ms.

Moreover, the number of operations is within the bounds determined in the formal analysis. It can also be seen that for the instances with more edges, the complexity is nearer the worst case, while for the ones with less edges it is close to the lower bound. This was expected, since a higher number of neighbours means that the algorithm will have to iterate through more neighbours and that there will be more occasions were a colour cannot be picked because it has already been taken by a neighbour.

Following the procedure adopted for the previous algorithm, a regression was performed for the running time of the graph the instances with half the maximum number of edges. Using a quadratic equation of the form $t = an^2 + bn + c$, the polynomial fit yielded $t = 0.27 \times 10^{-4}n^2 + 7.1 \times 10^{-4}n + 5.5 \times 10^{-4}$ (t in seconds), as shown in Fig. 9. This confirms the polynomial behaviour inferred previously. More precisely, the algorithm seems to have, on the average case, time complexity of $\mathcal{O}(n^2)$. The regression equation also allowed to predict the running time of bigger instances. For example, it is expected that a graph instance with 100 nodes would take 0.27 milliseconds to be solved.

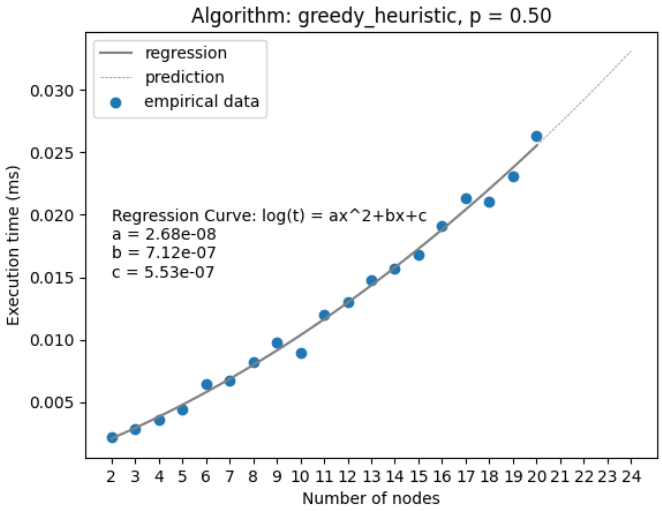


Fig. 9: Regression and prediction for the running time of the algorithm *chromatic_number_greedy*.

V. CONCLUSIONS

In this project, the chromatic number problem, which is a graph colouring problem with practical applications like register allocation in compilers, was tackled. It also required the exploration of some graph concepts and the best coding approaches and frameworks to generate, manipulate and draw graphs.

In order to solve the problem of finding the chromatic number, the first approach used was exhaustive search, which even after some algorithm optimizations, proved to be impossible to use for all but the very small problem instances, because they were exponential or worse. In fact, the best exhaustive search algorithm tested would take more than 3×10^{12} seconds to solve a relatively small problem instance with 20 nodes.

The other design strategy implemented with a fairly simple greedy heuristic that sorted the nodes from highest to lowest number of neighbours and then iterated the sorted neighbours, assigning the lowest possible colour to each one. This approach traded the certainty of an exact result

with an enormous improvement of the run time. Nevertheless, it was found that, for the tested instances, the greedy algorithm achieved a 97.7% precision. Regarding the execution time, all the results were on the order of the tens of milliseconds for less than 20 nodes, while even a 100 node instance was expected to be solved in less than 0.3 milliseconds.

Thus, the greedy heuristic was found to be a very powerful alternative for NP-complete problems like graph colouring, were the exhaustive search is not a real option for instances that are not too small.

REFERENCES

[1] Marek Kubale, *Graph Coloring*, chapter Preface, American Mathematical Society, 2004.

[2] "Wikipedia - Register allocation", `https://en.wikipedia.org/wiki/Register_allocation`, Accessed: 03/12/2021.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart, "Exploring network structure, dynamics, and function using networkx", in *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[4] "Wikipedia - Erdős–Rényi model", `https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model`, Accessed: 23/11/2021.

[5] "Wikipedia - Orders of magnitude (time)", `https://en.wikipedia.org/wiki/Orders_of_magnitude_(time)`, Accessed: 03/12/2021.

[6] Adrian Kosowski and Krzysztof Manuszewski, *Graph Coloring*, chapter Classical Coloring of Graphs, American Mathematical Society, 2004.

[7] "TimeComplexity - Python Wiki", `https://wiki.python.org/moin/TimeComplexity`, Accessed: 04/12/2021.