Contents

Reduction to graph colouring 2
Greedy colouring 2
Optimising the order of traversal 3
The "backtracking" approach 3
Tests and observations 5
Experimental 5
Results 6
Visualisation 8
The "complete subgraph" approach 8
Chaitin's algorithm 9
Refining the backtracking approach 10
Tests and observations 12
Experimental 12
Results 13
Efficiency analysis 15
References 17

Reduction to graph colouring

The register allocation problem takes variables that are used in a program. When the variables are used, they will be assigned to registers, each of which is a small part of the CPU that can be accessed more quickly than main memory. It must be ensured that two variables which are *live* at the same time are allocated different registers.

The graph colouring problem is a way of colouring the vertices of a graph such that no two adjacent vertices share the same colour.

Looking at the interference graph, it can be seen that this is essentially a graph colouring problem. Variables which are live at the same time can be considered to be adjacent vertices in a graph. Drawing an interference graph will allow a visualisation of the variables. Within the interference graph, vertices correspond to variables and there is an edge between the vertices if the variables are live at the same time.

The algorithm must assign colours (i.e. registers) to each vertex (i.e. variable) ensuring that no two adjacent vertices share the same colour. The algorithm must achieve this by using the least number of colours such that the solution is *optimal*.

Greedy colouring

The most intuitive way of designing an algorithm to perform this task is essentially just making a greedy choice. The greedy choice was used by Hannes and Nicholas for Part A for the assignment, however Nicholas implemented a *backtracking* approach which will be explained in the next section.

The implementation of the greedy algorithm involved picking a vertex within the graph, trying to assign it register 1, then looking at all the adjacent vertices and determine whether that register has already been allocated.

If the register has been allocated for an adjacent vertex, the register for the original vertex is incremented. The process is then repeated for the same adjacent vertices, however checking the incremented register.

If the register has not been allocated for an adjacent vertex, that register is assigned to the original vertex. The next vertex within the graph is then chosen and the process is repeated for this vertex.

This algorithm provided outputs which were frequently optimal, however there were instances where the choices were not optimal. Looking at Figure 1, it can be seen that the algorithm does not make the optimal choice. The optimal choice is this case should only use two colours – all the vertices on the left should be assigned colour 1, and all the vertices on the right should be assigned colour 2.

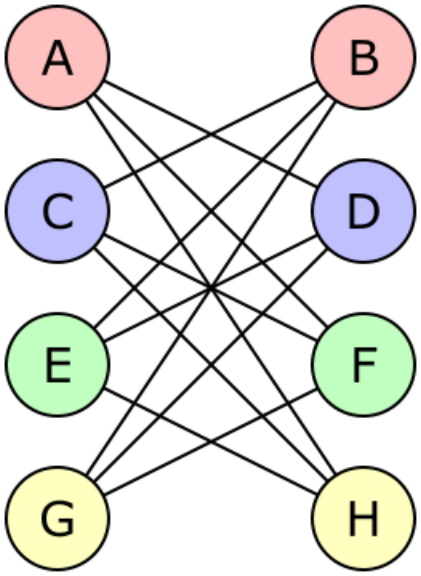

Figure 1: An example of a problem instance where the algorithm does not output an optimal result.

Optimising the order of traversal

An optimisation was to order the vertices by their degree (i.e. number of edges; number of adjacent vertices).

This algorithm worked by taking the vertex with largest degree first and starting with register 1, then checking all the adjacent vertices and simultaneously:

1. Checking whether any adjacent vertices already have this register allocated to them and incremented the register if it has already been allocated.
2. Finding the adjacent vertex of largest degree which has not been allocated a register.

The algorithm seemed to yield optimal results, however when once again looking at Figure 1, it can be seen that the algorithm is in fact not always optimal. The algorithm should only use two colours, however if this algorithm runs n vertices, it will assign $n/2$ colours (assuming it starts at the top-left and moves right). The optimisation does not help in this case since all the vertices are of the same degree.

The "backtracking" approach

The *backtracking* approach involves still making the greedy choice, however then running back through the graph to ensure that the optimal solution was chosen.

Backtracking meant that once the greedy choice was made, the algorithm will traverse back through each vertex and test whether the allocated choice was in fact the optimal choice for the whole graph.

Looking again at Figure 1, it can be seen that the greedy choice will result in $n/2$ colours being assigned. The backtracking approach will the test whether assigning each vertex a different register would result in the optimal result.

For Figure 1, the register allocation for each iteration would be as follows (the graph starts at the top-left, and then iterates right, then back to left):

Number of colours	Solution							
	a	b	c	d	e	f	g	h
4	1	1	2	2	3	3	4	4
3	1	1	2	3	2	3	2	3
2	1	2	1	2	1	2	1	2

It achieves this by first traversing back through the greedy choice:

Choice	Evaluation
$h: 4$	Incrementing this vertex would lead to a less optimal result.
$g: 4$	Incrementing this vertex would lead to a less optimal result.
$f: 3$	Incrementing this vertex would result in at least four colours being used, which is already the most optimal choice thus far.
$e: 3$	Incrementing this vertex would result in at least four colours being used, which is already the most optimal choice thus far.
$d: 2$	Incrementing this vertex to register 3, will result in the second output (three colours) above.
$c: 2$	This is ignored for this iteration.
$b: 1$	This is ignored for this iteration.
$a: 1$	This is ignored for this iteration.

The next iteration will backtrack through the result that was achieved in the first iteration:

Choice	Evaluation
$h: 3$	Incrementing this vertex would lead to a less optimal result.
$g: 2$	Incrementing this vertex would result in at least 3 colours being used, which is already the most optimal choice thus far.
$f: 3$	Incrementing this vertex would lead to a less optimal result.
$e: 2$	Incrementing this vertex would result in at least 3 colours being used, which is already the most optimal choice thus far.
$d: 3$	Incrementing this vertex would lead to a less optimal result.
$c: 2$	Incrementing this vertex would result in at least 3 colours being used, which is already the most optimal choice thus far.
$b: 1$	Incrementing this vertex to register 2, will result in the third output (two colours) above.
$a: 1$	This is the first choice made and thus the only register which can be allocated is register 1.

As seen above, the backtracking approach will thus yield the most optimal results. The algorithm's nature of checking its own outcomes and ensuring that the correct choice was made results in the optimal solution being found. The results of this algorithm can be visualised by Figure 2.

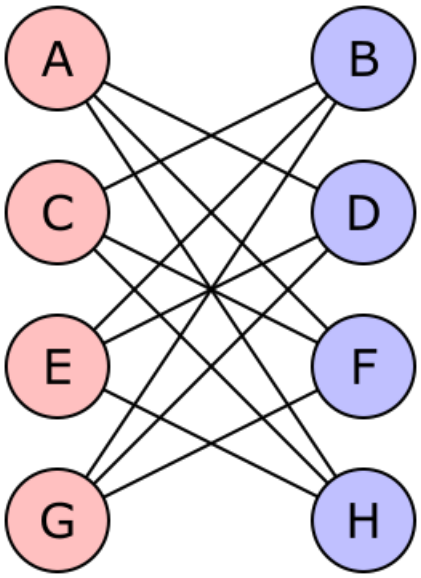

Figure 2: The optimal solution to the problem in Figure 1, output by the backtracking algorithm.

Tests and observations

Experimental

To test the efficiency of the backtracking algorithm, the algorithm was run on a set of randomly generated graphs. The output of the graph generator is constrained by two parameters:

- the number n of vertices,
- the probability p that any pair of vertices has an edge between them.

The random graph generator first inserts n vertices into the graph. To facilitate iteration over unordered pairs of vertices, the vertices are indexed (from zero). For each pair of vertices, a random number $r : 0 \leq r < 1$ is generated. An edge is added between the two vertices if $r < p$, satisfying the requirement that p is a probability.

An additional parameter k was originally used to impose a *ceiling* on the *minimum* number of colours required to colour the graph. This was implemented by ensuring that for a vertex pair (v_i, v_j) where i and j are their indices, an edge is *not* inserted between them if $i \equiv j \pmod{k}$. Thus, if $p = 1$ (i.e. edges are inserted between *all* vertex pairs not excluded by the equivalence), a complete k-partite graph is generated. This parameter was not used in the timed tests because fixing this parameter to any particular value did not appear to generate graphs more complex for the algorithm than those that were generated without it.

Another alternative parameterisation was considered in using a probability *function* $p(i,j)$ instead of a single probability for all vertex pairs. The insertion of an edge between v_i and v_j would then be decided by evaluating $r < p(i,j)$. One possible usage of this is to skew the degrees of vertices by using a low $p(i,j)$ for some vertices and a higher $p(i,j)$ for others. A fairly straightforward function serving this purpose is to convert the vertex indices linearly into numbers between 0 and 1, and then take their geometric mean:

$$p(i,j) = \frac{\sqrt{i \times j}}{n-1}$$

After some experimentation, it became evident that the most difficult graphs for the algorithm to solve are graphs where edges are evenly distributed for the graphs. For this reason, the probability function parameterisation was not used.

A timed test was conducted for each pair of (n, p) with n from 0 to 50 (in increments of 1), and with p from 0 to 1 in increments of 0.05. Each timed test was comprised of 16 trials (with each trial using a newly generated graph with the same parameters). For each test, the maximum, minimum and average across trials for the following were recorded:

- the time taken for the algorithm to complete,
- the total number of times the algorithm attempted to assign an existing colour to the vertex – henceforth referred to as *search steps*,
- the total number of times the algorithm added a colour to the interference set of a vertex (i.e. denoting that a vertex cannot be assigned a colour) – henceforth referred to as *interference steps*.

Both search steps and interference steps were included because it is difficult to determine which one is dominant. Consider the two extreme cases of a null graph (no edges) and a complete graph. For a null graph, there are $n - 1$ search steps, and 0 interference steps. Conversely, for a complete graph, there are $(n^2 + n)/2$ search steps, and $n^2 - n$ interference steps. It was noted from the results of the test that in the worst cases, the number of interferences steps was usually greater by a factor between 1 and 2 (implying that the two counts appear to increase proportionally to each other).

Results

The maximum number of interference steps for each vertex (across all trials), along with the maximum average of interference steps across values of p is depicted in Figure 3. In hindsight, taking the *median* number of interference steps *across all trials* may have served a better representation of the average case. Noting that the graph appears to show a linear trend across a logarithmic scale, it seems that the worst case is exponential with respect to the number of vertices.

Another key observation is that the time taken for the algorithm to solve different graphs of the same input size is extremely skewed. For example, with 50 vertices, most graphs were solved in well under 10 seconds – but a handful took up to 10 *minutes*. These cases were rare enough to render their investigation rather difficult, but common enough to cause concern about the overall efficiency of the algorithm.

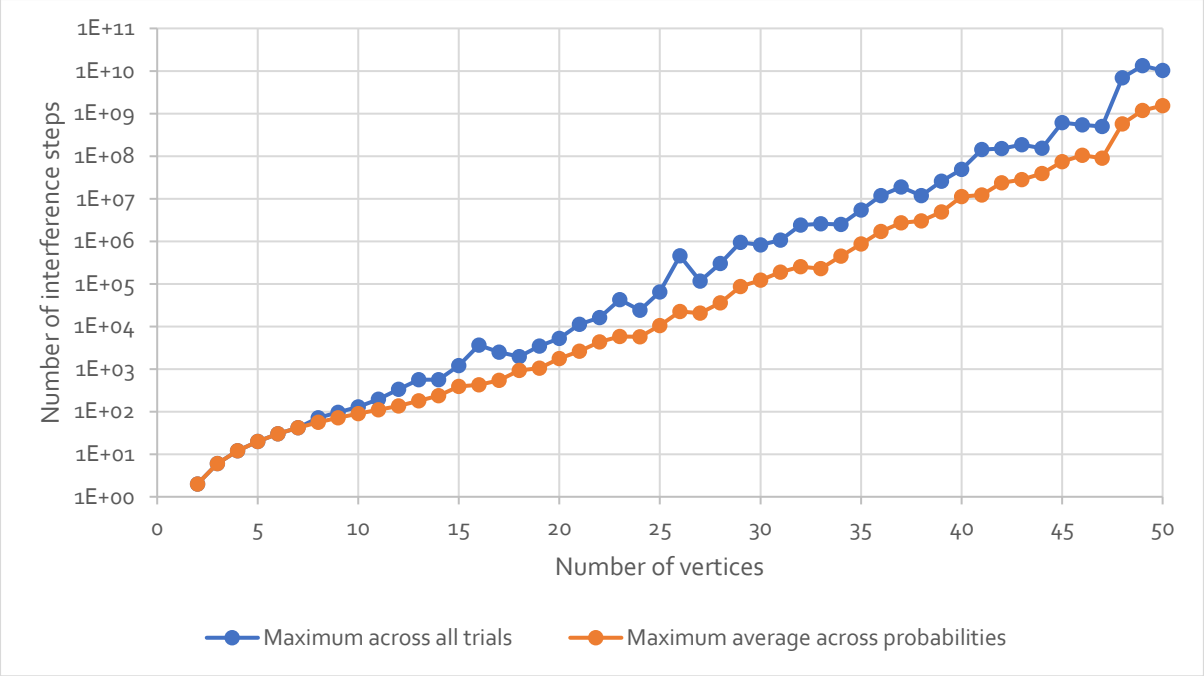

Figure 3: The number of interference steps taken for the algorithm to complete.

To investigate this phenomenon, an additional test comprised of 1000 trials with $n = 30$ and $p = 0.5$ was run. The number of interference steps for each trial was recorded. The result of this test is plotted in **[FIGURE]**. The plot suggests that complexities of problems has a *log-normal distribution*. This can be observed in that the plot resembles the inverse cumulative distribution function of a normal distribution – but the y-axis is logarithmic.

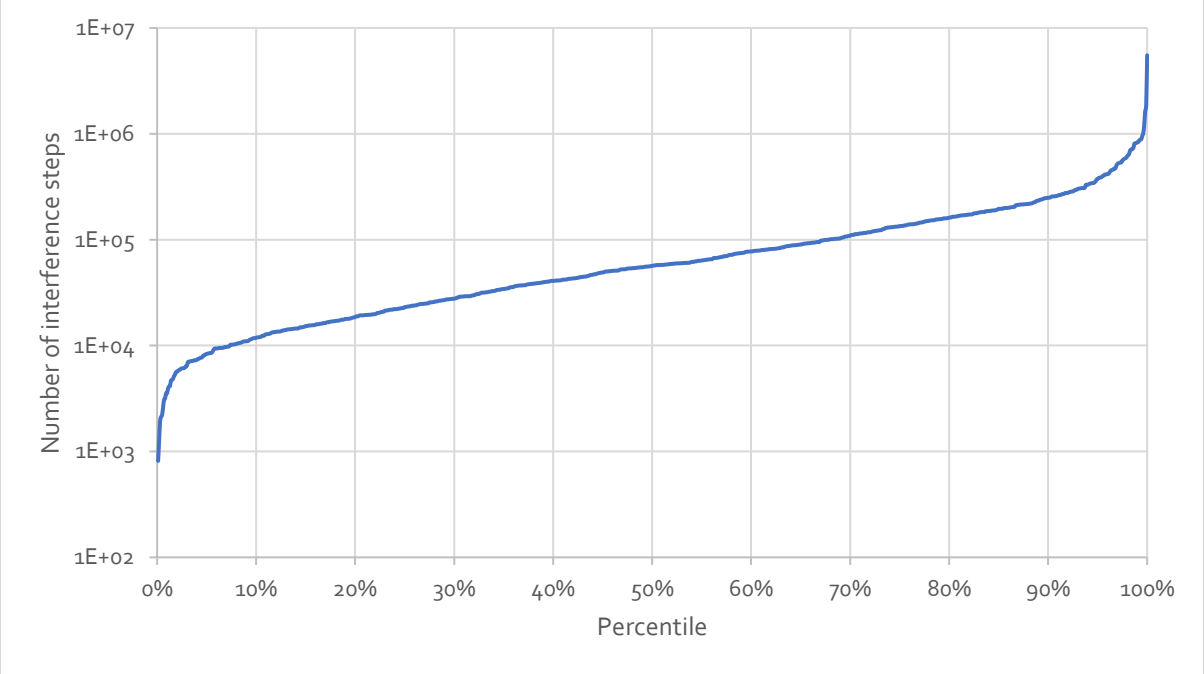

Figure 4: Plot of the number of interference steps taken for the algorithm to complete with $n = 30$ and $p = 0.5$. Each point corresponds to a single trial, of which there are 1000 in total.

These tests assume that the input of the register allocation problem is equivalent to that of the graph colouring problem – both mathematically and in terms of the distribution of inputs. As will be discussed later, this is a very pessimistic assumption.

Visualisation

Another testing utility implemented was a visualisation of the algorithm, using the *swing* and *AWT* libraries. The utility generates a graph, which the algorithm solves. The graph is then displayed in a window, with the vertices arranged in a circle in the order of traversal. Each frame corresponds to the assignment of a colour to a vertex, showing the tentative colourings at that point in the algorithm, as well as the optimal solution so far. **[FIGURE]** is a screenshot of the visualisation.

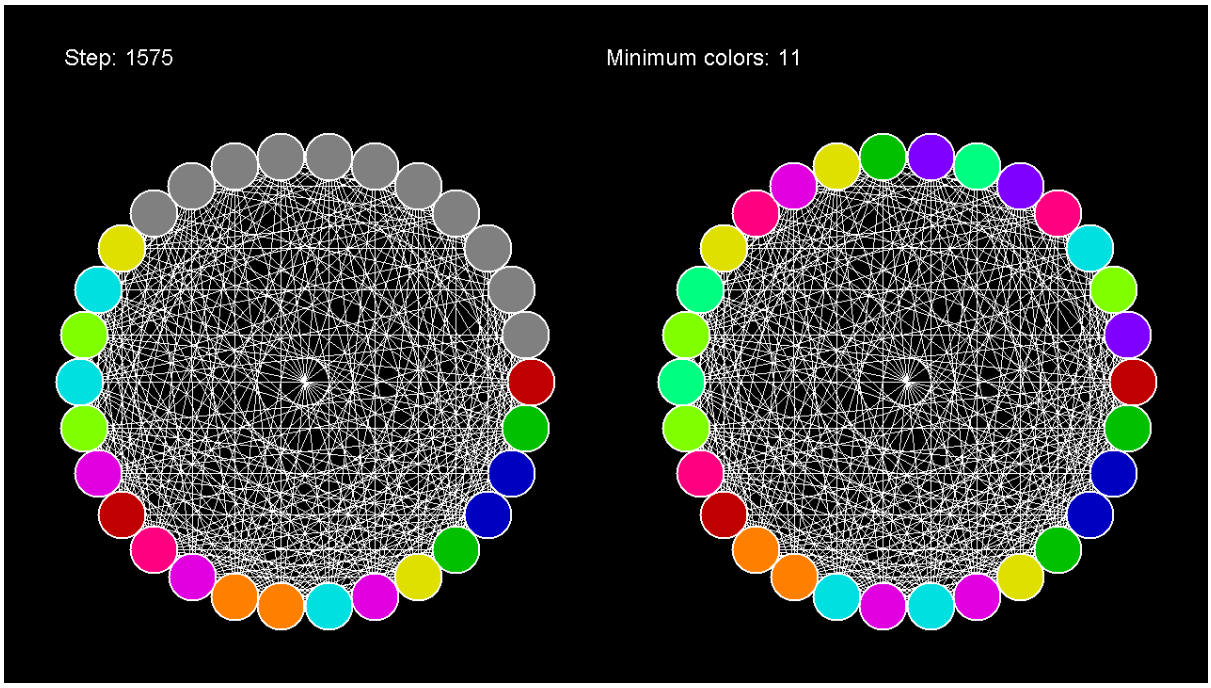

Figure 5: Screenshot of the algorithm visualisation utility.

The "complete subgraph" approach

The *complete subgraph* approach was an algorithm which first created subgraphs of every line of the input program. This turned the problem into a type of scheduling problem. The difference here being that instead of trying to complete as many tasks as possible with only one person, our algorithm will try to complete all the task with the fewest number of people.

The idea was implemented in parts of Hannes' code for Part A. The variables would be placed into different subarrays for each line within the input program. These subarrays contained each variable which was live at that particular line. Looking at the example below, it can be seen that if a large number of variables were live at some particular line, it could be useful in assigning initial colours:

$$S_1 = \{a, b, c\}$$
$$S_2 = \{a, b, c, d, e, f\}$$
$$S_3 = \{b, c, d, e, f\}$$
$$S_4 = \{e, f\}$$

The reasoning of using this approach was that if the subarrays were sorted in order of the number of variables within them, the subarray with the largest number of variables could be found. These subarrays will guarantee that the variables within are not able to be allocated to the same register, and thus locating the largest subarray could essentially colour a significant portion of the graph.

After these variables had their colour assigned, they could essentially be removed from the remaining subarrays meaning that only the unassigned variables would remain within the subarrays. This would reduce the number of variables to iterate over and could potentially remove some subarrays completely.

Chaitin's algorithm

> *Chaitin's algorithm is a bottom-up, graph colouring register allocation algorithm that uses cost/degree as its spill metric.*
>
> **— Gregory Chaitin** [1]

Chaitin's algorithm is generally used when there are only k number of colours available with which to colour the graph. The methodology is as follows:

- Suppose we are trying to k-colour a graph and find a node with fewer than k edges.
- If we delete the node from the graph and colour what remains, we can find a colour for this node if we add it back in.
- Reason – With fewer than k neighbours, some colour must be left over.

Algorithm:

- Find a node with fewer than k outgoing edges.
- Remove it from the graph.
- Recursively colour the rest of the graph.
- Add the node back in.
- Assign it a valid colour.

Chaitin's algorithm is difficult to adapt to our register allocation problem however. Our register allocation problem allows for an unlimited number of registers to be used, whereas Chaitin's algorithm assumes that there are k registers available. Thus, Chaitin's algorithm only served as a means by which we adapted our idea for our final algorithm.

Refining the backtracking approach

The *complete subgraph* approach was an attempt at using the form of the input to probabilistically minimise the amount of the work the algorithm would have to do by designating a traversal order. However, there are still numerous problems that this approach fails to address.

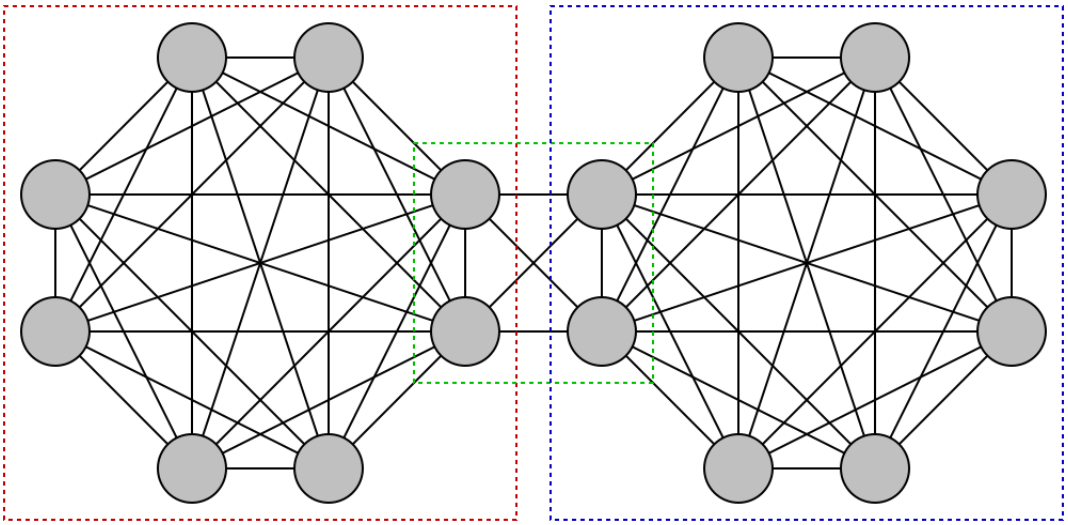

Figure 6: An example of a problem instance where the complete subgraph approach is inefficient. The dotted rectangles denote the complete subgraphs in the input.

Figure 6 illustrates one such problem with the complete subgraph approach. The algorithm will commence by assigning eight distinct colours to one of the larger subgraphs. It will then proceed to do the same for the other larger subgraph by assigning colours already used. It would make sense for the algorithm to start with the two vertices interfering with the first subgraph. However, there is no guarantee that it will do this. (Even if we sort vertices *within* a subgraph by their degree, we can simply add vertices and join then to the non-interfering vertices to "balance" their degrees.) Consider what happens if these vertices are traversed to last. Then there are $20\,160$ ways ($8! \div 2!$) to assign eight colours to the six non-interfering vertices, but only 720 ways ($6!$) that allow the two interfering vertices to be coloured validly. The point is that if there are numerous large complete subgraphs with sparse connections between them, the algorithm may end up wasting a lot of time.

Two other problems were identified with both the complete subgraph approach and the original backtracking algorithm:

- The algorithm is likely to waste time branching through all possible colour assignments for a vertex with no unassigned adjacencies. There is no hurry to colour such a vertex since it does not affect the remainder of the solution (unless it requires a new colour). An example of this is depicted in Figure 7.
- The traversal order of vertices is determined by degree of the vertices. This weakly correlates to factors that contribute to the complexity of the problem. It would be better for the vertices to be traversed based on the number of adjacent colours since this corresponds to the number of search tree branches produced by that vertex. An example demonstrating the significance of this is depicted in Figure 8.

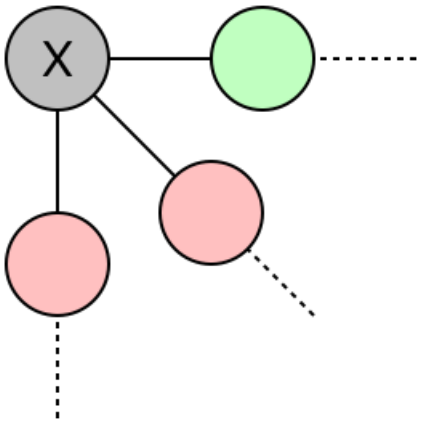

Figure 7: Part of a graph exemplifying an inefficiency of the algorithms conceptualised so far. There is no need to search through all possible colourings of vertex X since it does not affect any other unassigned vertices.

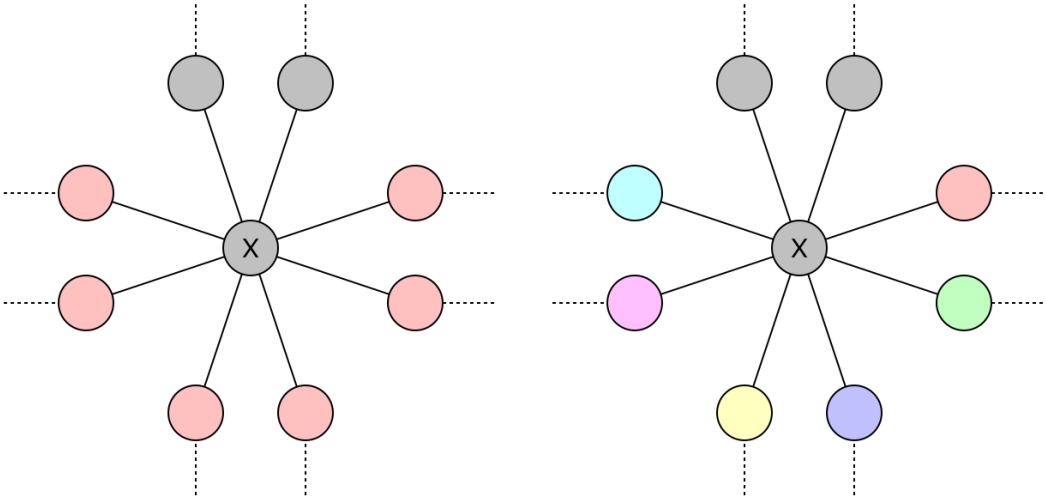

Figure 8: Part of a graph at two different points in the algorithm exemplifying why it is preferable to traverse vertices according to the number of adjacent *colours*, not simply just the degree. Assume that in both cases seven colours have been used so far. At left, there are seven colours that may be assigned to vertex X. If vertex X is selected at this point, it will create seven branches in the search tree. At right, there are only two colours that may be assigned to vertex X. If vertex X is selected at this point, only two new branches will be created in search tree.

These considerations led to a new refinement to the backtracking approach – that the vertices be sorted *dynamically*. In the original backtracking approach, the traversal order was determined by a list that was sorted *once* by the degree of the vertices. Instead, the new implementation maintains a *priority queue* of unassigned vertices that is updated as colours are assigned and unassigned. The comparator used by the queue sorts according to:

1. the number of *colours* adjacent to that vertex,
2. the number of unassigned neighbours.

This addresses many of the issues with the original algorithm and the complete subgraph approach that were discussed previously.

The algorithm maintains, for each vertex:

- An array such that $a[i]$ is the number of neighbours assigned colour i. Since colours are indexed from 1, $a[0]$ is used to denote the number of unassigned neighbours.
- A count of the number of adjacent colours.

Whenever the algorithm assigns or unassigns a colour to a vertex, it updates the array and the adjacent colour count of all its neighbours, then updates their positions in the queue. Using a priority queue, this is achieved by removing the vertex from the queue (an $O(n)$ operation), then reinserting it ($O(\log n)$). Vertices that have already been assigned a colour are not updated, since the state of the graph at the time of their assignment will be the same as when they are backtracked to.

Tests and observations

Experimental

A different approach was taken for testing the new algorithm. It was realised that using randomly generated graphs as input is too pessimistic in representing the input of register allocation. The input files tend to produce highly edge-dense interference graphs, and such that if there is a high likelihood of *edge transitivity*. Formally, for three vertices (v_1, v_2, v_3) such that there exist edges between v_1 and v_2, and between v_1 and v_3, it is likely that there also exists an edge between v_2 and v_3. These characteristics make the interference graphs corresponding to actual input files much easier to solve in the average-case. For this reason, randomly generated *input files* were used for the timed tests.

The input file generator accepts two parameters (n, l):

- n is the number of variables to include,
- l is the number of lines (excluding the *live-in* and *live-out* declarations).

To simplify testing, it was decided that the number of variables and the number of lines would be kept equal for the timed tests. It is a fair assumption that a typical input file would have roughly the same number of lines as it does variables.

The generator proceeds as follows:

1. n variable names are selected for inclusion.
2. For each line, a random variable is selected to be placed on the left-hand side of the assignment statement.
3. Each left-hand side occurrence is matched with a right-hand side occurrence (thus ensuring that each assignment is used).
4. For each line presently lacking right-hand side variables, a random variable is selected to be placed on the right-hand side (thus ensuring that each line has at least one variable).
5. The variable sets for each line are converted into the line itself, adhering to the language specification. Elements impertinent to the parser such as arithmetic operators, whether a line is a memory call, and integers are randomly selected.
6. The result is written to a file.

For the timed tests, 100 input files were generated for parameterisation $n = l = k$ with k from 10 to 500 in increments of 10, for a total of 5000 input files.

For each *trial*, the following was recorded:

- the number of *steps* taken for the algorithm to complete, in this case defined as the number of *traversals* between vertices in the graph,
- the amount of time (in milliseconds) taken for the algorithm to complete.

For each *test* (i.e. a single parameterisation, comprised of 100 trials), the minimum, maximum, median, and mean number of steps and amount of time were recorded.

A timeout of 30 seconds for any trial was imposed. The testing terminates once any test completes with more than 10 timeouts.

Results

The testing terminated following input size 390.

As can be seen in Figure 9, even with 100 trials, it is difficult to obtain a sense of the worst-case complexity. As a consequence of this, characterising the average-case with a median rather than a mean is more useful. For example, notice that the outlier in the input size 120 test causes a spike in the mean.

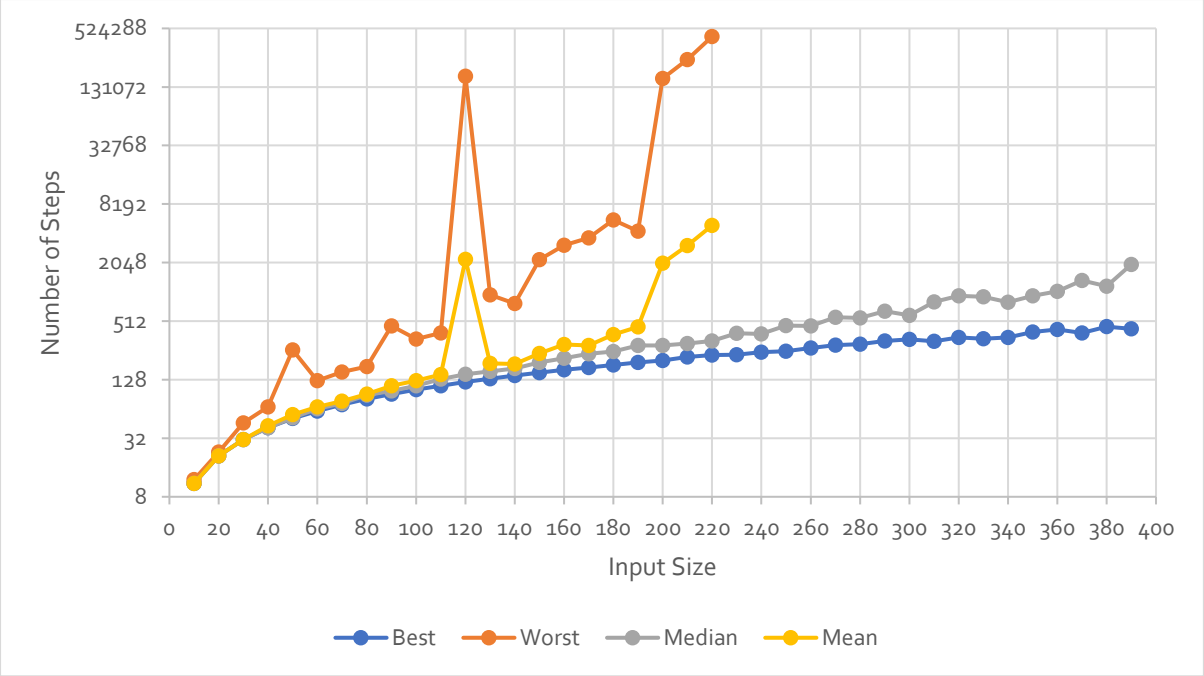

Figure 9: The number of steps taken by the algorithm for each input size.

It is important to note that the number of steps (as defined in these tests) cannot be directly interpreted as the complexity of a problem. This is because the number of steps is simply the number of vertex *traversals*. (This will be discussed further in the efficiency analysis section.) Figure 10 depicts the best-case and median of times taken for the algorithm to complete.

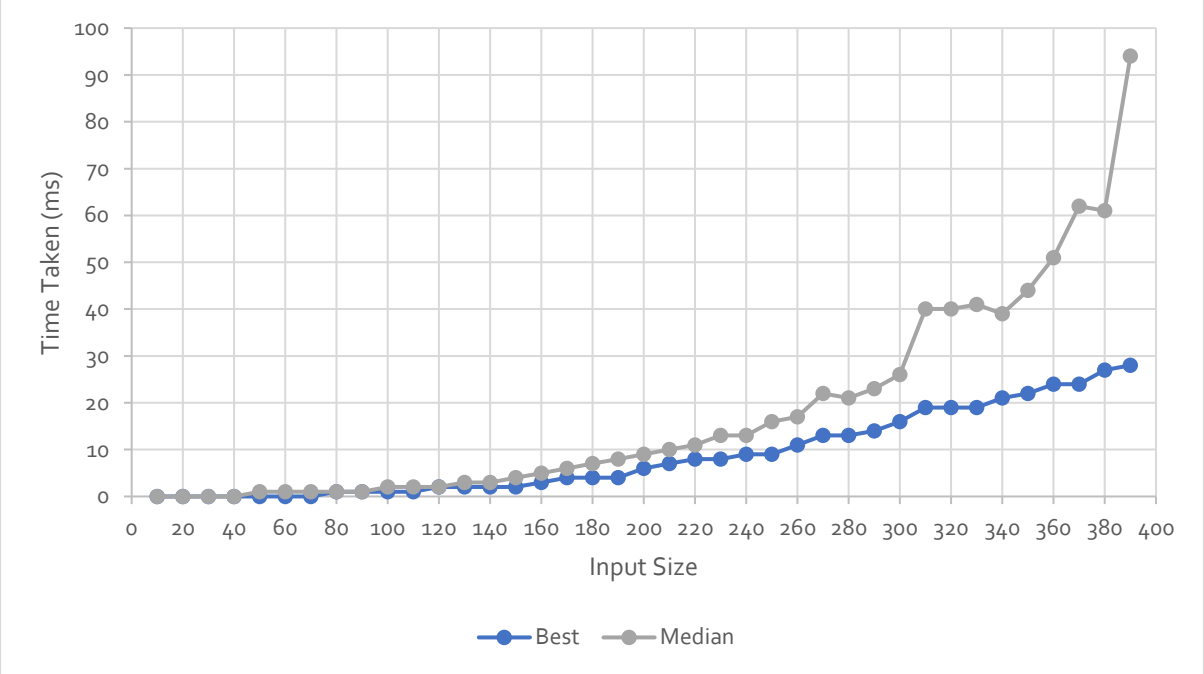

Figure 10: The amount of time taken by the algorithm for each input size in the best-case (i.e. the trial with the shortest time) and the median.

Figure 11 depicts the worst-case and mean of times taken for the algorithm to complete. As with the number of steps, the data is chaotic and difficult to interpret.

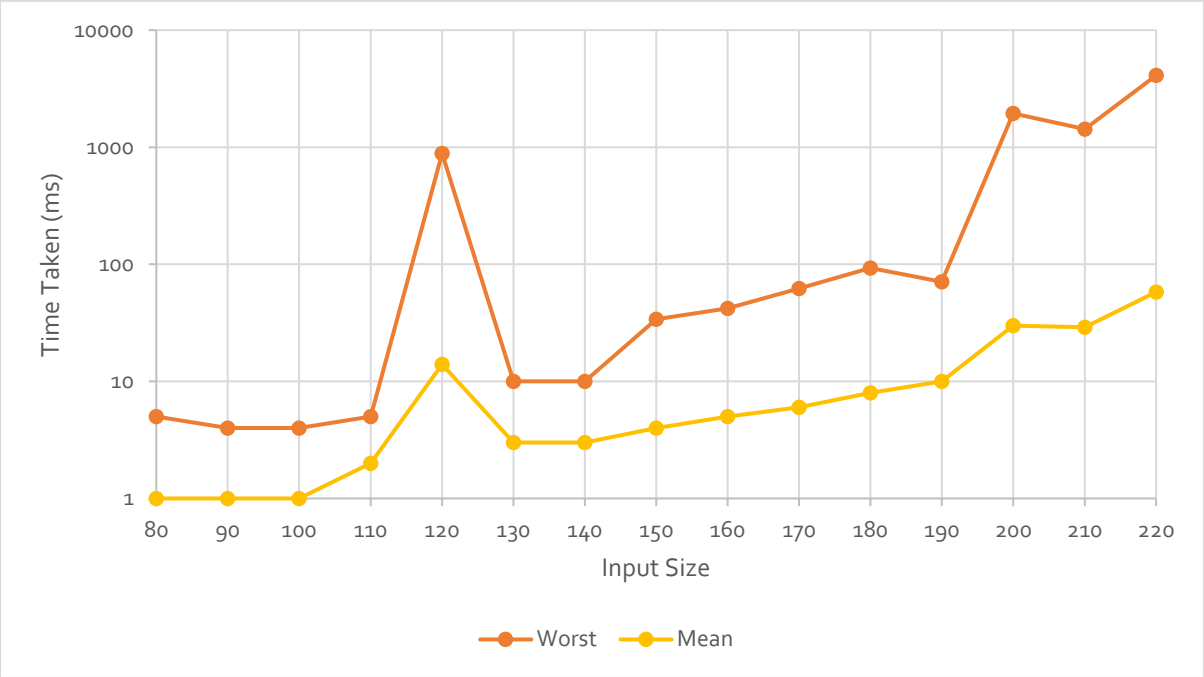

Figure 11: The amount of time taken by the algorithm for each input size in the worst-case (i.e. the trial with the longest time) and the mean.

Efficiency analysis

It has proven rather difficult to provide a good upper bound on the time complexity $T(n)$ of our algorithm.

To begin with, consider the cost $f(n)$ of tentatively allocating a variable to a register:

- Each interfering variable must be *notified* of the allocation. The number of these *notifications* is the number of interfering variables, and is therefore $O(n)$.
- Each notification of an interfering variable requires that the interfering variable be removed from the priority queue and then reinserted. Removal from a priority queue takes $O(n)$, and reinsertion takes $O(\log n)$.

Therefore, the cost is:

$$\begin{aligned} f(n) &= O(n) \times \big(O(n) + O(\log n)\big) \\ &\in O(n^2) \end{aligned}$$

One approach to evaluate the overall time complexity is to simply consider that, with n variables, there are at most n different registers. Thus, we can represent the complexity as a recursion tree constructed as follows:

- Each layer k of the recursion tree corresponds to variable k. Thus there are n layers in total.
- Each node of the recursion tree has n children, corresponding to the n different ways of allocating variable k. The top layer of the recursion tree also has n vertices.
- The weight associated with each node is $f(n)$, i.e. $O(n^2)$, corresponding to the allocation of the variable to the register.

This gives an expression for $T(n)$:

$$\begin{aligned} T(n) &\in O\left(O(n^2)\sum_{k=1}^{n} n^k\right) \\ &\in O\left(n^2\left(\frac{n(1-n^n)}{1-n}\right)\right) \\ &\in O\left(\frac{n^{n+3}-n^3}{n-1}\right) \\ &\in O(n^{n+2}) \end{aligned}$$

One refinement is to consider that the cost of tentatively allocating the k-th variable to a register is actually $O\big(f(n-k)\big)$, not $O\big(f(n)\big)$. This is because only variables that are yet to be allocated a register are updated in the queue. We can also that the k-th variable in the enumeration cannot be allocated a register greater than k, i.e. there are at most k registers that can be allocated to the k-th variable. Thus we obtain:

$$T(n) \in O\left(\sum_{k=1}^{n}(n-k)^2 k^k\right)$$

It is rather difficult to evaluate a summation over k^k, so we will have to retract our refinement back to a summation over n^k:

$$\begin{aligned} T(n) &\in O\left(\sum_{k=1}^{n}(n-k)^2 n^k\right) \\ &\in O(n^{n-1}) \end{aligned}$$

We may attempt to impose further constraints on the recursion tree to more accurately and less pessimistically reflect the behaviour of the algorithm. Our previous result imposed the constraint that "variable k can be allocated a register from 1 to k". We can strengthen this to "variable k can be allocated register r only if there exists a variable from 1 to $k-1$ which has been allocated register $r-1$. The recursion tree is hence as depicted in Figure 12.

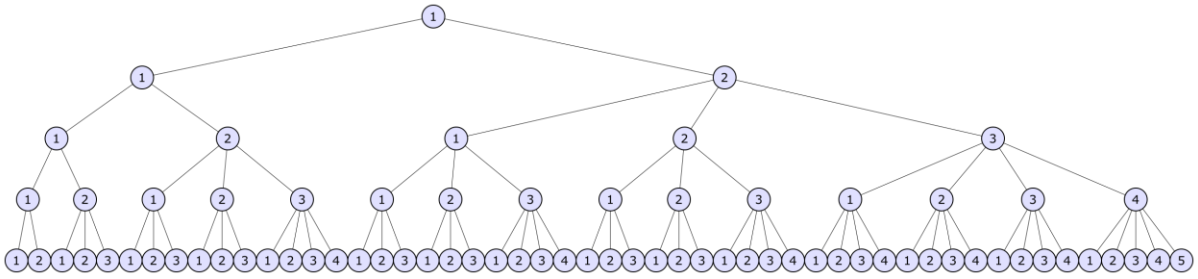

Figure 12: The first four layers of the worst-case recursion tree. Note that the vertex labels do not represent the cost of the vertex, but rather the colour assigned in the step represented by the vertex.

We can express this as a multi-dimensional recurrence:

$$T(n,r,\lambda) = \begin{cases} O(n^2) + T(n+1,\lambda+1,\lambda+1) + \sum_{k=1}^{\lambda} T(n+1,k,\lambda), & n > 0 \\ 0, & n = 0 \end{cases}$$

It appears that this recurrence is related to the Bell numbers [2] – which count the number of ways to partition a set of n elements. In particular, the n-th layer in the recursion tree has B_n vertices. Thus:

$$T(n) \in O(n^2 B_n)$$

An asymptotic bound for the Bell numbers is given by Berend & Tassa [3]:

$$(n \mapsto B_n) \in O\left(\left(\frac{n}{e^{0.6-\varepsilon}\log_e(n+1)}\right)^n\right), \qquad \varepsilon > 0$$

Therefore:

$$\begin{aligned} T(n) &\in O\left(n^2\left(\frac{n}{e^{0.6-\varepsilon}\log_e(n+1)}\right)^n\right) \\ &\in O\left(n^2\left(n \times n^{-\log_n(0.6-\varepsilon)} \times n^{-\log_n \log_e(n+1)}\right)^n\right) \\ &\in O\left(n^2\left(n^{n\left(1-\frac{\log_2(0.6-\varepsilon)}{\log_2 n}-\frac{\log_2 \log_e(n+1)}{\log_2 n}\right)}\right)\right) \\ &\in O\left(n^{o(n)}\right) \end{aligned}$$

References

[1] G. Chaitin, "Register allocation and spilling via graph colouring," *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 66-74, 2004.

[2] N. J. A. Sloane, "A000110 - OEIS," [Online]. Available: https://oeis.org/A000110. [Accessed 27 August 2018].

[3] D. Berend and T. Tassa, "Improved bounds on Bell numbers and on moments of sums of random variables," *Probability and Mathematical Statistics*, vol. 30, no. 2, pp. 185-205, 2010.