# 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 (register) to each vertex (variable) ensuring that no two adjacent vertices (variables) share the same colour (register). The algorithm must achieve this by using the east 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 following 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 counter 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 << GRAPH (4-COLOURS) >>, it can be seen that the algorithm does not make the optimal choice. The optimal choice is this case should only use 2 colours; all the vertices on the left should be assigned colour 1, all the vertices on the right should be assigned colour 2.

<< INSERT GRAPH HERE (4-COLOURS) >>

## Optimising the order of traversal

An optimisation was to order the vertices by their degree (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 incrementing the register if it has already been allocated.
2. Finding the adjacent vertex of largest degree which not been allocated a register.

This algorithm seemed to yield optimal results, however when once again looking at << GRAPH (4-COLOURS) >>, it can be seen that the algorithm is in fact, not always optimal. The << GRAPH (4-COLOURS) >> should only use 2 colours, however if this algorithm runs for vertices, it will assign 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.

<< INSERT GRAPH HERE (4-COLOURS) >>

# 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 << GRAPH (4-COLOURS) >>, it can be seen that the greedy choice will result in colours being assigned. The ‘backtracking’ approach will then test whether assigning each vertex a different register would result in the optimal result.

For << GRAPH (4-COLOURS) >>, 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):

1. a:1, b:1, c:2, d:2, e:3, f:3, g:4, h:4
2. a:1, b:1, c:2, d:3, e:2, f:3, g:2, h:3
3. a:1, b:2, c:1, d:2, e:1, f:2, g:1, h:2

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

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 4 colours being used, which is already the most optimal choice thus far.

e:3 -> incrementing this vertex would result in at least 4 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 (2.) output 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:

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 (3.) output 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 << GRAPH (2-COLOURS) >>.

<< INSERT GRAPH HERE (2-COLOURS) >>

# The “complete subgraph” approach –

The complete subgraph approach was an algorithm which first created sub-graphs 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 sub-arrays for each line within the input program. These sub-arrays 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:

Line 1: {a, b, c}

Line 2: {a, b, c, d, e, f}

Line 3: {b, c, d, e, f}

Line: 4: {e, f}

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

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

Trying to efficiently perform these steps proved to be difficult. For input programs with a significant number of lines, the algorithm would require a momentous number of iterations. Running into these significant complications and additional time complexities which changed the problem from polynomial time complexity of to an exponential time complexity of , this approach was decided to not be used. This approach did however result in initiating the idea for the final algorithm which was used.

# Researching “Chaitin’s Algorithm”–

“Chaitin's algorithm is a bottom-up, graph colouring register allocation algorithm that uses cost/degree as its spill metric.” <<CHAITIN QUOTE>> Chaitin’s algorithm is generally used when there are only number of colours available with which to colour the graph. The methodology is as follows:

* Suppose we are trying to -colour a graph and find a node with fewer than edges.
* If we delete this 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 register available. Thus, Chaitin’s algorithm only served as a means by which we adapted our idea for our final algorithm

Gregory Chaitin (April 2004). "Register allocation and spilling via graph coloring". ACM SIGPLAN Notices. 39 (4): 66–74. doi:10.1145/989393.989403.