

Using Node Merging to Enhance Graph Coloring

Steven R. Vegdahl
University of Portland
5000 N. Willamette Blvd.
Portland, OR 97203
(503) 943-7215
vegdahl@up.edu

1. ABSTRACT

A Chaitin-style register allocator often blocks during its simplification phase because no node in the interference graph has a degree that is sufficiently small. Typically, this is handled by node-splitting, or by optimistically continuing—and hoping that a legal N -coloring will still be found. We observe that the merging of two nodes in a graph causes a reduction in the degree of any node that had been adjacent to both. We have enhanced Chaitin's coloring algorithm so that it attempts node-merging during graph simplification; this often allows simplification to continue, while still guaranteeing a coloring for the graph. We have tested this algorithm using Appel's database of register-coloring graphs, and have compared it with Chaitin's algorithm. The merge-enhanced algorithm yields a better coloring about 8% of the time, and a worse coloring less than 0.1% of the time.

1.1 Keywords

Register allocation, graph coloring, register coalescing

2. INTRODUCTION

Chaitin's coloring algorithm [6,7] attempts to color a graph with N colors in two stages:

- *Simplify* the graph by repeatedly removing nodes from the graph whose degree is less than N , until the graph is reduced to an empty graph.
- *Color* the nodes in reverse-order of removal.

If the simplification step is successful, an N -coloring is guaranteed, because each node—at the time it is colored—is adjacent to at most $N-1$ already colored nodes.

The simplification phase blocks if it reaches a point where all nodes

in the graph have degree N or greater. In this case, one of two approaches is typically used:

- Split one or more nodes so that the simplification can continue (Chaitin [7]); this causes spill-code to be inserted in the code that is ultimately generated.
- Increase N and continue “optimistically”, as suggested by Briggs [5]. If the increase is small, a coloring with the original N might still be found, but is not guaranteed. Splitting is done (and hence spill-code introduced) during the coloring phase whenever a node cannot be colored.

Optimistic coloring algorithms are based on the observation that Chaitin's algorithm, when applied to a graph using N available colors, often results in a coloring of less than N . Our experimental results support this approach, as about 60% of the graphs we colored using Chaitin's algorithm were colored in less than the original N (see Section 7).

Chaitin's algorithm has two non-deterministic aspects. First, during simplification, there are typically occasions where many nodes have fewer than N neighbors; an implementation can pick any of these as the first to remove. A single graph can therefore induce many different orderings in which the nodes are colored. Second, during coloring, there may be more than one color available with which to color a node.

Thus, two different implementations of Chaitin's algorithm—or different orderings of nodes and arcs in the graph's representation—can give rise to different colorings. These colorings can easily differ in the number of colors that are used.

The non-deterministic aspects of Chaitin's algorithm might make it attractive to apply Chaitin's algorithm repeatedly to a graph, using a pseudorandom number generator whenever a non-deterministic choice is available. Our experiments, described in Section 6, would also support this approach.

This paper describes a technique for allowing the simplification phase of Chaitin's algorithm to proceed in situations where it would normally block. Spilling/splitting strategies and copy-minimization strategies have been previously proposed by Bergner [3], Chow [8], George [9], Kurlander [10], and Leuh [11]. The work described here does not deal with these topics directly. Our underlying coloring techniques, however, can be used with these other strategies.

Most other node-merging work has focused on coalescing nodes that are copy-related (e.g., Briggs [5], George [9], Park [12]). Our work considers the merging of non-copy-related nodes. We also perform merging within the simplification loop rather than before or after.

3. NODE-MERGING

During the simplification phase of Chaitin's algorithm, if the algorithm blocks, our algorithm introduces a node-merging step prior to optimistically increasing N or performing node-splitting. The basic idea of node-merging is to merge two *unconnected* nodes together

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. SIGPLAN '99 (PLDI) 5/99 Atlanta, GA, USA
© 1999 ACM 1-58113-083-X/99/0004...\$5.00

into a single node. Such merging is effectively a commitment that the two nodes have the same color in the final graph. When two nodes are merged, one of them is removed from the graph, and its arcs are attached to the node that remains; this may introduce redundant arcs into the graph, which can be eliminated.

When two unconnected nodes are merged, it changes the degrees of the nodes in the graph as follows:

- The degree of any node that had been adjacent to *both* merged nodes is reduced by one.
- The degree of the merged node is the sum of the degrees of the original nodes, minus the number of nodes they had in common. (If each node had at least one neighbor not common to the other, then the degree of the merged node is greater than that of either of the original nodes.)

Consider the graph in Figure 1. The degree of node 1 is 4; the degree of node 2 is 6; they share 3 neighbors in common. After merging (Figure 2), the degree of each common neighbor (3, 5, 6) is reduced by one; the degree (7) of the merged node is greater than the degree of either node 1 or node 2 because each had neighbors not shared by the other.

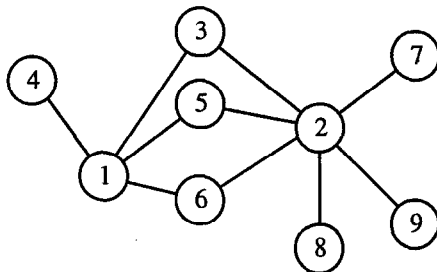


Figure 1: Graph before merging nodes 1 and 2

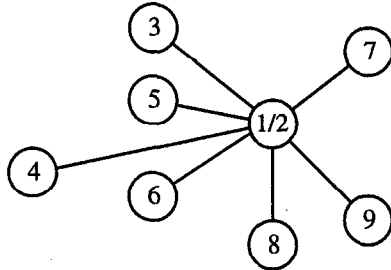


Figure 2: Graph after merging nodes 1 and 2

Our hope is that with the degree-reduction of these shared nodes, the degree of one or more of the nodes drops below N ; this would allow Chaitin-style simplification to continue.

Our goal, then, is to select node-pairs for merging that have many common neighbors, and very few unique neighbors. If we can find such pairs, the degrees of many nodes will be reduced, and the degree of the merged node will not increase significantly over that of the nodes it has replaced. The ideal case, of course, is to find a node whose neighbors are a subset of some other node; here many nodes decrease in degree, and none increases. (Appel [2] refers to this as “George” merging.)

4. EXAMPLE

Consider the problem of three-coloring the graph in Figure 3. The simplification phase of Chaitin’s algorithm will block immediately because no node has fewer than three neighbors. We can use merg-

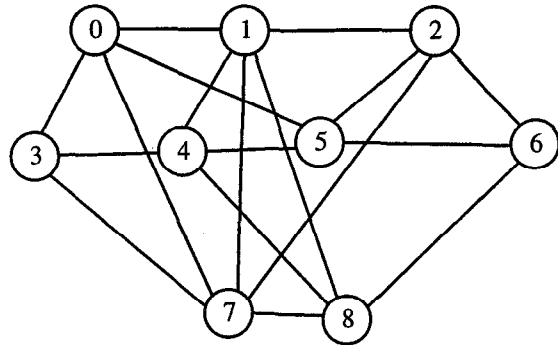


Figure 3: A graph to 3-color

ing to make progress, however, by noting that:

- The neighbors of node 3 (0, 4, 7) are a subset of those of node 1 (0, 2, 4, 7, 8).
- Nodes 1 and 3 are not connected.

We can therefore merge node 3 into node 1 (which in this case amounts to removing node 3), making a notation that node 3 will have the same color as node 1 in the final graph. This results in the graph shown in Figure 4.

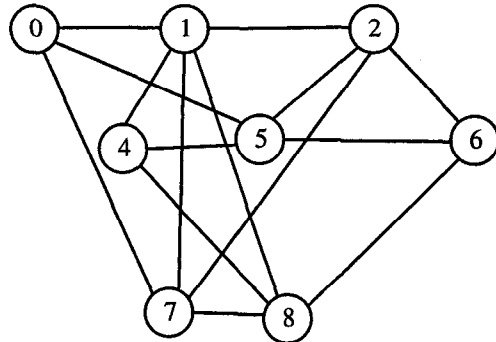


Figure 4: Graph with node 3 removed

This graph still has no node with degree less than three, but now the neighbors of node 0 (1, 5, 7) are a subset of the neighbors of node 2 (1, 5, 7, 6). Because nodes 0 and 2 are not connected, we can merge them by removing node 0. This graph is shown in Figure 5.

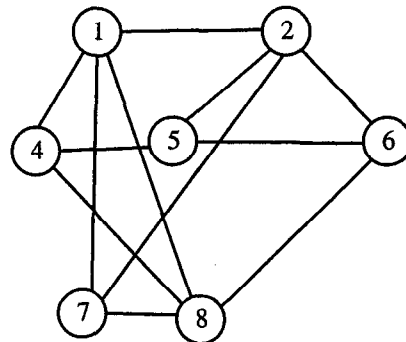


Figure 5: Graph with node 0 also removed

The resulting graph still has no node with degree less than three. However, nodes 2 and 8 have three neighbors in common (1, 6, 7) and only one each not in common (4, 5). We therefore merge nodes 8 into 2 by removing node 8, and adding its extra connection (4) to node 2. The resulting graph is shown in Figure 6.

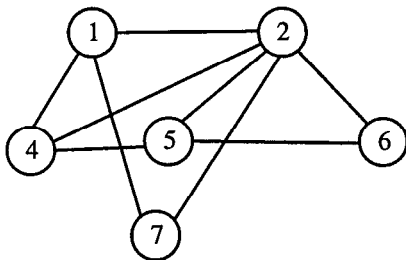


Figure 6: Graph after merging nodes 2 and 8

At this point, Chaitin-style simplification can be resumed: nodes 6 and 7—each having degree 2—can be removed, followed by nodes 1, 2, 4 and 5.

We can now color the nodes in reverse order of removal (5, 4, 2, 1, 7, 6):

- 5, 1: red
- 4, 6, 7: green
- 2: blue

Finally, we color the nodes that had been “removed” by merging, giving:

- 1, 3, 5: red
- 4, 6, 7: green
- 0, 2, 8: blue

In this example, the effect of node-merging was to allow simplification to continue without introducing spilling during the simplification step, and without the need to increase N .

5. DETAILS OF THE MERGE-ENHANCED ALGORITHM

Our simplification algorithm uses Chaitin’s as a basis; merging—as illustrated above—is attempted only when Chaitin’s algorithm blocks. In order to select promising nodes for merging, we keep an ordered *candidate list* of node-pairs to merge. Our algorithm blocks only when the candidate list becomes empty. Typically, this occurs when the graph has become a clique. At this point, one of the traditional techniques (e.g., node-splitting) could be employed.

The candidate list is a priority queue (represented as a heap in our implementation). We weight each node-pair with a “pair-score” between 0.0 and 1.0 according to the *ratio of the number of common neighbors to the total number of neighbors in the smaller-degree node*. Thus, if node 1 is connected to nodes 3, 4, 5, and 6 (see Figure 1) and node 2 is connected to nodes 3, 5, 6, 7, 8 and 9. The “1-2” node-pair would receive a score of 0.75, based on the ratio of neighbors of 1 (the smaller-degree node) that are also neighbors of 2. A node-pair score of 1.0 indicates that one node’s neighbors are a subset of the other’s.

The algorithm is complicated by the fact that node removals (both traditional and merge-based) cause pair-scores to change. Whenever a modification is made to the graph that causes a node-pair to change its score, the pair is removed from the priority queue and reinserted. Though we have developed techniques for “optimizing away” some of these insertion/removals, this bookkeeping slows

down the algorithm significantly, making it quadratic at best. Heuristics to reduce the overhead by not considering all node-pairs as candidates are discussed in Section 8.

6. EXPERIMENTAL RESULTS

In order to determine the effectiveness of the merge-enhanced algorithm, we used Appel’s database [1] of 27,921 graphs that were produced from an actual compiler. We did not perform spilling because the database did not include the live-track information. Rather, we simply determined the best coloring an algorithm could give for each graph. (This was done by choosing an initial N based on the graph’s node-count; then we repeatedly applied the algorithm, performing a “binary search” on N , until we converged the best result).

To have a basis for comparison, we also implemented and tested two versions of Chaitin’s algorithm. Both were optimistic—that is, they were willing to choose an N larger than the target, with the hope that the actual coloring would be better than the number chosen. The difference between the two algorithms was that one colored the graph once and then stopped. The other performed 100 repetitions, using a pseudorandom number generator during color selection.

The reasons for using two versions of Chaitin’s algorithm are as follows. First, we wanted to compensate for the fact that we made no attempt to apply any particular intelligence during the coloring phase of the algorithm. Second, we wanted to gain insight as to whether injecting randomization/repetition into Chaitin’s algorithm might be useful.

The results are summarized in Tables 1–3. Of the 27,000+ graphs colored, the merge-enhanced algorithm generally did better than the others. It gave an improvement over Chaitin’s algorithm approximately 8% of the time. The merge-enhanced version did slightly better than the algorithm with repetition/randomization, yielding improvements 0.6% of the time.

The size of the graphs in Appel’s database varied widely. 53 were empty; many had only a handful of arcs. The column labeled “large graphs” in Tables 1–3 is an attempt to characterize how the algorithms performed on larger graphs—those containing 500 or more arcs. The database contains 1,917 such graphs. Our thought was that viewing the performance of each algorithm on larger graphs might give better intuition about its overall effectiveness.

Table 1: Chaitin/Merge vs. Chaitin/Single

result	all graphs	large graphs
equal	25,738 (92.2%)	1,700 (88.7%)
Merge better	2,183 (7.8%)	217 (11.3%)
Single better	0 (0%)	0 (0%)

Table 2: Chaitin/Merge vs. Chaitin/Repeated

result	all graphs	large graphs
equal	27,746 (99.4%)	1,792 (93.5%)
Merge better	165 (0.6%)	123 (6.4%)
Repeated better	10 (0.04%)	2 (0.1%)

Table 3: Chaitin/Single vs. Chaitin/Repeated

result	all graphs	large graphs
equal	25,768 (92.3%)	1,732 (90.3%)
Single better	59 (0.2%)	42 (2.2%)
Repeated better	2,094 (7.5%)	143 (7.5%)

When one of the algorithms found a better coloring than another, it typically did so by performing the coloring with one less color. However, in some cases, the color-count difference was greater than one. Table 4 summarizes the situations in which one algorithm outdid another by more than one color. The merge-enhanced algorithm was never outdone by more than one color. It occasionally outdid the others, however, by two colors, and sometimes by three or four.

Table 4: Coloring-differences greater than one

better/worse	# times	max. difference
merge/single	31	3
merge/repeated	27	4
single/repeated	7	4
repeated/single	5	2

7. OTHER OBSERVATIONS

A major reason for doing these experiments was to gain insight regarding:

- How often a “perfect” node-pair—where one node’s neighbors are a subset of the other’s—would be found.
- Whether there is a pair-score (ratio of common neighbors with the other node in the pair to total nodes) below which it does not make sense, in practice, to do merging.

The results are summarized in Table 5. For 38.5% of the graphs,

Table 5: Summary of merges performed

pair-scores used	all graphs	large graphs
(no merging)	10,762 (38.5%)	1,123 (58.4%)
all = 1.0 (perfect)	16,518 (59.2%)	507 (26.4%)
all ≥ 0.8 , some < 1.0	362 (1.3%)	112 (5.8%)
all ≥ 0.6 , some < 0.8	242 (0.9%)	142 (7.4%)
some < 0.6	37 (0.1%)	33 (1.7%)

the ultimately-minimal coloring was found without any merging. 59.2% needed only “perfect” merges. Only 0.1% used any merges at all with pair-scores under 0.6. A higher percentage of the larger graphs (58.4%) required no merging. However, imperfect merges were required for 15% (287 of 1,917) of the large graphs, but for only 1.3% (354 of 27,921) of the “small” graphs.

Table 6 shows the same figures, limited to the graphs in which the merge-enhanced algorithm outdid the repetition version of Chaitin. These results would suggest that it is sometimes effective to merge two nodes when the pair-score is between 0.5 and 0.6. However, in our experiments, it was never helpful to perform a merge with a pair-score below 0.5

Table 6: Summary of merges performed: graphs where merging improved result over repetition

pair-scores used	all graphs	large graphs
all = 1.0 (perfect)	77 (46.7%)	55 (44.7%)
all ≥ 0.8 , some < 1.0	39 (23.6%)	34 (27.6%)
all ≥ 0.6 , some < 0.8	40 (24.2%)	25 (20.3%)
some < 0.6	9 (5.5%)	9 (7.3%)

A couple of individual colorings illustrate the effectiveness of the merge-enhanced algorithm. The first (graph 21231 in Appel’s database) contains 1,358 nodes and 15,335 arcs. Both single-execution and repetition versions of Chaitin’s algorithm produced a 21-coloring. The merge-enhanced algorithm gave a 20-coloring. During the simplification phase, the algorithm blocked 21 different times (because all nodes exceeded $N-1$); each time a perfect node-pair was found that allowed the algorithm to continue.

Another case of interest was graph 20800, which consists of 215 nodes and 4,607 arcs. Here, the single-execution algorithm produced an 18-coloring, while the repetition-algorithm produced a 19-coloring. The merge-enhanced algorithm produced a 17-coloring. It required 28 merges (11 perfect, 17 imperfect), with pair-scores ranging as low 0.826.

We also gained insights regarding optimistic versions of Chaitin’s algorithm. One might expect that if the simplification were done for N colors, the final coloring might be $N-1$ or $N-2$. We observed large differences, however, between the color-count used for simplification and the actual number of colors in the final coloring. When running Chaitin’s traditional algorithm, the final coloring was less than N 16,713 times (60%); in one case, N was 167, and the final coloring used 13 colors. For Chaitin’s algorithm with repetition, the final coloring was less than N 17,198 times (62%); the largest difference was for a graph that was 24-colored with an N of 44. For both algorithms, the value of N was at least 4 larger than the final color-count about 3% of the time. These numbers support the notion that optimistic coloring is useful.

8. A FASTER ALGORITHM

The algorithm described above is exhaustive. It considers all possible node-pairs as candidates for merging. The resulting (quadratic) time-complexity would likely make it impractical for use in a production compiler. We therefore modified the algorithm so that it did significantly less bookkeeping.

Our focus was to reduce neighbor-counts for nodes that are very close to the simplification threshold. We therefore restricted ourselves to consideration of merge-candidates that are neighbors of many small-degree—and hence almost-removable—nodes.

We chose a fairly simple heuristic for selecting such node-pairs. It is parameterized by two constants, M , and K :

- Select the M *smallest-degree* nodes in the graph. (This is done

in an efficient manner by maintaining a heap.)

- For each node in the graph, compute the number of small-degree neighbors it has. Select the K nodes with the largest values as merge-candidates; these are often *not* small-degree nodes. (In the present implementation, an $n \log n$ sorting algorithm is used to find the K best candidates. We could clearly do better by not including candidates with counts of zero, or by using the well-known linear-time algorithm for finding the K best candidates [4].)
- Select the pair of non-adjacent nodes from the merge-candidate pool that has the best pair-score.

Although this is not a particularly sophisticated heuristic for reducing the pair-candidate space, the results were quite encouraging. We first chose an $M=50$ and $K=70$. Of the 27,921 graphs, the exhaustive algorithm outdid the heuristic in only 18 cases; the heuristic outdid the exhaustive in 6 cases. The two algorithms therefore produced equally good colorings in 99.9% of the cases.

Given these results, we decreased the size of our candidate pool using $M=20$, $K=30$. Surprisingly, this gave slightly better colorings than $M=50$, $K=70$: it found a smaller coloring 9 times; it found a larger coloring 5 times. (This would suggest that focusing on the near-removable nodes is more effective than finding the best pair-score.) When compared with the exhaustive algorithm, $M=20$, $K=30$ did better in 8 cases, worse in 16.

This heuristic for selecting merge-candidates significantly sped up the algorithm. The heap of node-pairs—and its consequent bookkeeping-overhead—was eliminated. The number of node-pairs being considering was bounded by a constant, so the cost of a merge-step was reduced to $n \log n$. Our experiments indicate that the number of merges is typically small (under 10 in 99% of the cases); the cost of performing the merges would seem to be nearly $n \log n$ in practice.

Our performance numbers support this. In our tests, the performance the heuristic algorithm with $M=50$, $K=70$ was comparable to that of the Chaitin's algorithm with randomization. (The primary intent of the experiments was to discern the quality of the graph-coloring, *not* to gather performance data. External factors such as system load—which vary widely on our system—were not accounted for.)

9. SUMMARY

It appears to us that using merging to enhance Chaitin's algorithm does indeed have merit. Assuming that the graphs in Appel's database are a representative sample, it would appear that merging improves Chaitin's algorithm approximately 8% of the time, and for large graphs, better than the randomization/repetition algorithm about 6% of the time.

We were encouraged by the number of graphs that could be reduced using only "perfect" merges. This suggests the usefulness of an algorithm that considers only perfect node-pairs as merge candidates, which could simplify pair-selection and its associated bookkeeping.

It also appears that it is not particularly helpful to perform merging on candidates with a substantial number of unshared neighbors. Our experiments suggest that even if non-perfect merges are per-

formed, that it would be wise to consider only those with pair-scores above 0.5.

Our first attempt at a simple heuristic for speeding up the algorithm resulted in an algorithm that gave colorings that were essentially equal to the exhaustive algorithm. We conclude from this that it is not very difficult to find good node-pair candidates; better heuristics almost certainly exist.

The improvement shown by Chaitin's algorithm with randomization/repetition over single execution would suggest that this may also be a useful technique. These results strongly indicate the importance of using an intelligent algorithm during the coloring phase.

10. ACKNOWLEDGEMENTS

I am grateful to Gord Vreugdenhil and Max Hailperin for their comments on earlier drafts of this paper, and to Andrew Appel for making available his graph-coloring database [1].

11. REFERENCES

- [1] Appel, A. Sample Graph Coloring Problems. URL: <http://www.cs.princeton.edu/fac/appel/graphdata> (1996).
- [2] Appel, A. *Modern Compiler Implementation in Java*. Cambridge University Press (1998).
- [3] Bergner, P., Dahl, P., Engebretsen, D. and O'Keefe, M. Spill Code Minimization via Interference Region Spilling. *SIGPLAN Notices* 32, 5 (May 1997), 287–295. *Proc. ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.
- [4] Blum, M., Floyd, R., Pratt, V., Rivest, R. and Tarjan, R. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7,4 (1972), 448–461.
- [5] Briggs, P., Cooper, D. and Torczon, L. Improvements to Graph Coloring Register Allocation. *ACM Trans Programming Languages and Systems*, 12,4 (Oct. 1990), 501–536.
- [6] Chaitin, G., Auslander, M., Chandra, A., Cocke, J. Hopkins, M. and Markstein, P. Register Allocation via Coloring. *Computer Languages* 6(1981), 47–57.
- [7] Chaitin, G. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices* 17, 6 (June 1992), 257–265. *Proc. ACM SIGPLAN '82 Symposium on Compiler Construction*.
- [8] Chow, F. and Hennessy, J. The Priority-Based Coloring Approach to Register Allocation. *ACM Trans Programming Languages and Systems*, 12,4 (Oct. 1990), 501–536.
- [9] George, L. and Appel, A. Iterated register coalescing. *ACM Trans. on Programming Languages and Systems*, 18, 3 (May 1996), 300–324.
- [10] Kurlander, S. and Fischer, C. Zero-Cost Range Splitting. *SIGPLAN Notices* 29, 6 (June 1994), 257–265. *Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
- [11] Lueh, G. and Gross, T. Call-Cost Directed Register Allocation. *SIGPLAN Notices* 32, 5 (May 1997), 296–307. *Proc. ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.
- [12] Park, J. and Moon, S. Optimistic Register Coalescing. *Proc. 1998 International Conference on Parallel Architectures and Compilation Techniques* (1998), 196–204.