Submitted to Professor Forrest Brewer and Professor Marek-Sadowska on March 24, 1997, as a requirement for Logic Design Automation, ECE256B, and Algorithmic Logic Synthesis, ECE256D.

BDD Techniques for Graph Coloring and Related Problems

Steve Haynal
e-mail: haynal@engineering.ucsb.edu

Abstract

Graph coloring is a general solution to the scheduling problem. Variants of the scheduling problem abound in computer engineering CAD problems. BDDs, which have efficient set and data compression properties, are a tool useful for solving NP-Complete problems such as the graph coloring problem. This report explores BDD techniques helpful for determining the chromatic number of a graph, enumerating all valid graph colorings given a set *C* of colors, and identifying would be odd-cycle causing edges in two-colorable graphs.

1 - Introduction

The purpose of this report is to describe progress made on a class assigned research project. At the beginning of the term, an original problem was selected. This problem was to find edges, which if added to an existing undirected graph, form odd (even) circuits. Figure 1 shows a possible input graph for this problem. Notice that the original graph contains no odd circuits. To answer the problem, edges *b* and *c* would have to be returned. They would form odd circuits if added. Also, edge *a*, should be returned as a would be even circuit causing edge.

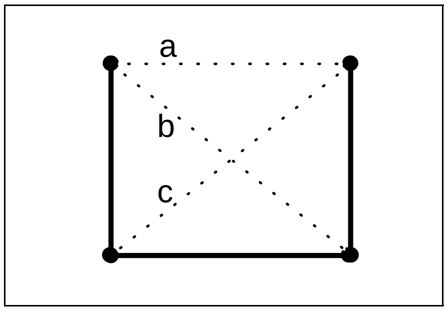

Figure 1: Graph input and output for original problem.

As with many research project, new problems evolved from the original problem statement. The evolved problems in this report are how to find the chromatic number of an undirected graph and how to find all valid colorings of an undirected graph. Figure 2 shows a graph with a valid coloring with *C*={green,red,blue}. Since vertices must not be adjacent to vertices with the same color, no valid coloring can be found for this graph with less than three colors. This is not the only valid coloring of this graph using *C*. For example, vertex *c* could be blue. Also, vertex *a* could be red and so on and so forth. For this simple graph, it is easy to tally up 12 valid colorings using *C*.

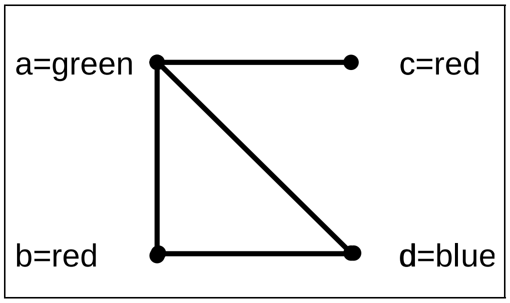

Figure 2: A three-colored graph.

Coloring solves general scheduling. This is the impetus to solve this fairly mathematical problem in an engineering setting. As an introductory example, consider a set of seven committees that must be scheduled in one hour slots in one afternoon. Some of these committees have members who are also members of other committees. One way to avoid conflicts (two committees meeting at the same time with common members) is to assign each committee to a separate one hour slot. Unfortunately, there are not enough hour slots in one afternoon! Another solutions is to pose this as a coloring problem. Each committee is a vertex of a graph. An incompatibility between two committees is one or more common members. This is represented as an edge between the two vertices. Finding the minimum number of colors to color this graph is equivalent to scheduling these committees in the minimum number of one hour slots. Some engineering examples of the scheduling problem are minimum-width channel routing, chip register/resource scheduling and assignment of binary codes to state symbols. .

2 - General Approach to Solutions

This section outlines the generally reasoning used in this research project to solve the previously mentioned problems.

2.1 - Finding Possible Odd (Even) Circuit Causing Edges

Theorem: A graph can be 2-colored if and only if it does not contain a circuit of odd length.

Proof: The proof for this theorem can be found in J. Bondy and U. Murty, *Graph Theory with Applications,* American Elsevier, New York, 1976, as well as other basic graph theory books.

From this theorem, it can be implied that any added edge that destroys 2-coloredness forms an odd circuit. Also, any added edge that preserves 2-coloredness and connect two vertices in the same graph forms and even circuit. Figure 3 shows the graph of Figure 1 with a valid coloring. Notice that would be odd circuit edges *b* and *c*, do indeed connect vertices of the same color (destroy 2-coloredness). Also, would be even circuit edge *a* connects vertices of different colors.

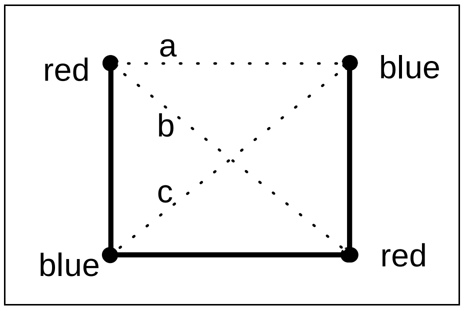

Figure 3: Edges between same (different) color(s) produce odd (even) circuits.

A general solution readily falls from these implications:

1. Two-color the graph.
2. For odd circuit causing edges, enumerate all edges between vertices in the same color set.
3. For even circuit causing edges, enumerate all edges between vertices in opposite color sets that don't exist in the original graph.

Step 1 is easy to do. Just pick a vertex and color it one color. Color all adjacent vertices the other color. Repeat for all adjacencies until done. There is one exception in step 3. Notice that by enumerating all edges between vertices of different colors existing edges are enumerated also. These must be excluded if only the set of nonexistent but even circuit causing edges are desired.

2.2 - Chromatic Numbers and Valid Colorings

Two-coloring can be written as a product of Boolean constraints. Consider the simple two-vertex graph in Figure 4. A valid coloring insists that vertex *a* have the opposite color of vertex *b*. Defining one color as normal and the other as complement, all constraints on adjacent vertices can be enumerated and then intersected. The result is a SOP form of all valid colorings. As a Boolean constraint, this is $(ab' \vee a'b) \wedge (ba' \vee b'a) = (ab' \vee a'b)$.

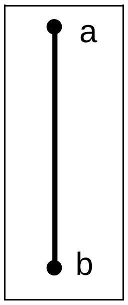

Figure 4: Two-colorable graph.

This constraint writing method can be generalized to graphs that are not fully connected. Since the previous example was fully connected, it is important to note that constraints are only written on edges. If there are no edges between two vertices, than there is no immediate constraint on what colors they must be assigned relative to each other. This is the equivalent to assigning a don't care to non-adjacent vertices when writing the constraint. Furthermore, the constraint is symmetric. It must be written only once for each edge.

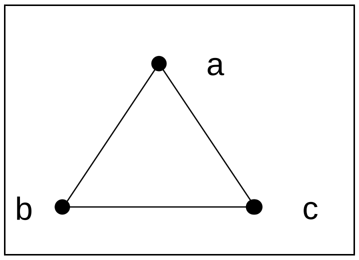

Figure 5: Example graph for generalized constraints with colors greater than two.

a	b	c
OO	O1	O1
OO	O1	1O
OO	1O	1O
..	..	..
..	..	..
1O	O1	O1

Table 1: Constraints due to vertex *a*.

This constraint writing method can also be generalized to cases with more than two colors. Table 1 enumerates most of the constraints placed on vertices *b* and *c* by vertex *a* of Figure 5. Notice that two bits are used to represent the coloring information. Only three colors are being used here; there is no assignment for 11. If the constraints for the edge between *b* and *c* were also enumerated and then intersected with Table 1's constraints, the result would be a SOP representation of all six valid coloring schemes using a set of three colors.

With this reasoning, a general solution for determining chromatic numbers and valid colorings presents itself.

1. Write Boolean constraints for *C* colors on *n* vertices due to each *e* edge.

2. Compute the product.
3. An existing minterm verifies the chromatic number $|C|$.
4. Each minterm indicates a possible valid coloring.

By recursively attempting to color a graph, adding one to *C* each time no minterms result, it is possible to determine the chromatic number of the graph. To save time, a good lower bound on the chromatic number is needed. A simply computed chromatic number lower bound is:

$\chi(G) = \frac{n^2}{n^2-2m}$ where *n* is the number of vertices and *m* is the number of edges. (Same reference as theorem 2.1.)

3 - Implementation Specifics

With the general solution framework built, it is now possible to focus on BDD specific implementation details. This section will describe graph representation, Boolean relations, and constraint application.

3.1- Graph Representation

The chosen method for representing a graph is to describe each edge as a minterm. Since each edge has only two vertices, there will be exactly two 1's in each edge minterm. For example, Table 2 lists the minterms for the graph in Figure 6.

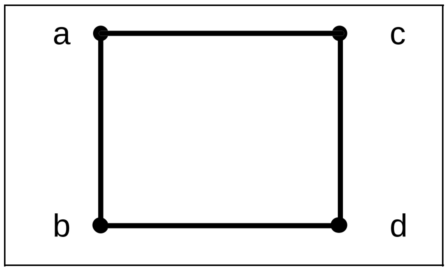

Figure 6: Example graph for BDD graph representation.

a	b	c	d
1	1	0	0
1	0	1	0
0	1	0	1
0	0	1	1

Table 2: Minterms for each edge of Figure 6 graph.

This BDD graph representation has some nice properties. First, the Universe is a tuple of all possible minterms with 2 bits set. This is symmetric and highly compressed as a BDD. Second, this representation is useful for partitioning. For example, a valid two-coloring for the graph in Figure 6 is 1010. By intersecting the term -0-0 with the Universe, the edge 1010 is returned. This is a would be odd circuit edge. All would be odd circuit edges can be obtained with a few simple BDD operations, the graph Universe, and both senses of two-coloring.

There is one disadvantage to this representation. The resulting BDDs do not compress the many zeros that are in each minterm of large graphs. For example, a 11 vertex, 18 edge graph requires 38 nodes as a BDD and 23 nodes as a ZBDD. A simple adjacency list would require only 18 entries.

3.2 - Boolean Relations

Using the reasoning described in section 2.2, a BDD describing Boolean relations among vertex colorings can be constructed. In this BDD, any path to one is a valid coloring. Furthermore, the existence of a path to one implies the graph is colorable in *C* colors. Figure 7 shows abstractly what the Boolean relation will look like as a BDD. In this example, each vertex is assigned 3 bits for coloring information. Vertex 1 may have a valid coloring of 111. Any vertex adjacent to vertex 1 must not have this bit pattern (color) in the same minterm where 111 appears for vertex 1. To ensure this, the following equations describe the BDD construction:

$$\text{Two-Coloring} = \prod_{j,k}\left(v_j \oplus v_k\right) \text{ where } \left(v_j, v_k\right) \in \text{ edges.}$$

$$2^i\text{-Coloring} = \prod_{j,k}\sum_{i}\left(v_{ji} \oplus v_{ki}\right) \text{ where } \left(v_j, v_k\right) \in \text{ edges.}$$

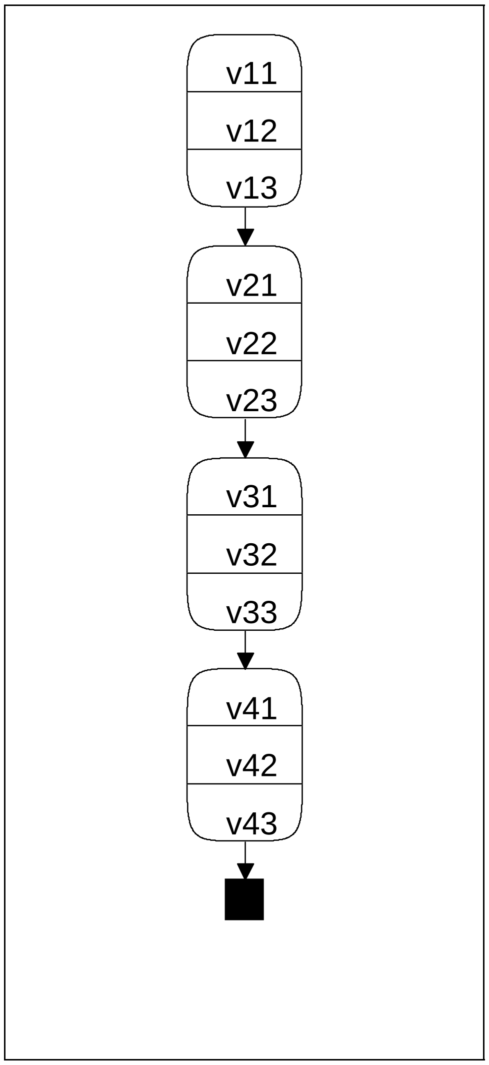

Figure 7: Abstract BDD Boolean Relation.

3.3- Tighter Constraints

The observant reader will note that in the previous example 3 bits are used to encode the coloring information and it appears impossible to specify exactly 5 colors. To do this, additional constraints must be added to the general constraints of section 3.2. Let c_j and c_k be BDDs with i levels and minterms for 0 to $|C|$-1. With this constraint BDD it follows that,

$$|C|\text{Coloring}=\prod_{j,k}\left(c_j \cap c_k \cap \sum_i\left(v_{ji} \oplus v_{ki}\right)\right) .$$

Fortunately, building this BDD is straightforward. If ordering is maintained with the MSB at the top of the BDD through LSB at the bottom, then there is always one and only one BDD node at each level. Figure 8 shows a constraint BDD that specifies exactly 5 colors, 0-- through 100. Since all minterms from 0-3 are included (0--) the false branch of the top node is set to terminal one. Likewise, there are no terms with 10-, so the true branch of the middle nodes is set to terminal zero. Using similar reasoning, the entire BDD is constructed.

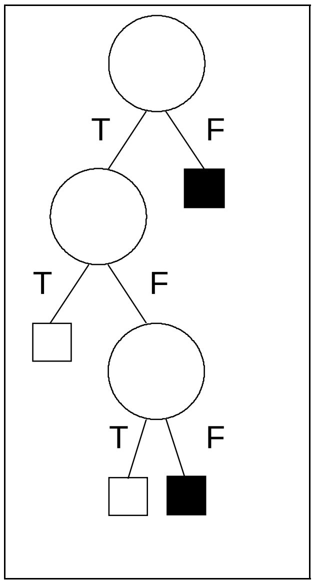

Figure 8: 5 color only constraint BDD

3.4- Results

Table 3 shows results for applying the described BDD techniques on some graph coloring benchmarks. These results are for a strict application of what has been described. No enhancement techniques have been applied. Notice that only small graphs are feasibly colored. A larger graph, gueen6_6.col, caused a machine with 400+ megabytes of memory to start swapping.

File	Vertices	Edges	Chromatic Number	User Time	Valid Coloring Permutations	Maximum BDD Nodes
queen5_5.col	25	160	5	18 s	240	~300,000
myciel3.col	11	20	4	1.7 s	12,480	~25,000
queen6_6.col	36	290	-	15 minutes	-	>17,000,000

4 - Further Research Directions

As the previous section's results show, keeping all valid colorings for a graph in one BDD is impractical for all but small graphs. There are just too many combinations. Therefore, techniques must be developed which make this problem manageable. This section will explore methods for making this growth problem tamable.

4.1 - Tighter Constraints

Placing additional coloring constraints (i.e. pick one color, only a subset of three colors, etc.) on various vertices reduces the number of superfluous colorings represented in the BDD. This can save space, decrease the time needed to color graphs and permit larger graphs to be colored. This section discusses several possibilities for placing additional coloring constraints on vertices while not excluding a valid coloring.

4.1.1 - 1,2,3,.....,*x*-1 Coloring.

Theorem 4.1.1: *C* is a set of colors. *G* is a graph to be colored from *C*. Without exclusion of a valid coloring using only *C*, $|C|$-1 random vertices, v_i, of *G* may be colored from X_i, subsets of colors from *C*, such that $X_0 \subset X_1 \subset X_2 \ldots\ldots \subset X_{|C|-1} \subset C$.

Proof: By induction. Pick vertex v_0. It can be assigned one and only one color. Call this color 0. Therefore $X_0 = \{0\}$. Pick vertex v_1. It can be assigned the same color as v_0 or another color. Call this other color 1. Therefore $X_1 = \{0,1\}$. This reasoning can be extended to any number of vertices. $|C|$ is a limit since we can not assign more colors than this. All remaining vertices are assigned a color from *C*.

This theorem has been applied to the set of benchmarks from section 3 with results shown in Table 2. Notice that both time and space requirements were reduced significantly at the expense of not enumerating all valid coloring permutations. Furthermore, queen6_6.col, insolvable before, is now solvable. Unfortunately, larger examples still explode.

File	Vertices	Edges	Chromatic Number	User Time	Valid Coloring Permutations	Maximum BDD Nodes	Constraint
queen5_5.col	25	160	5	2 s	2	~6000	Constrained
queen5_5.col	25	160	5	18 s	240	~300,000	Free
myciel3.col	11	20	4	0.6 s	656	~2000	Constrained
myciel3.col	11	20	4	1.7 s	12,480	~25,000	Free
queen6_6.col	36	290	7	410 s	20	~10,000,000	Constrained
queen6_6.col	36	290	-	15 minutes	-	>17,000,000	Free

Table 2: Results using 1,2,3,.....,*x*-1 Coloring

Theorem 4.1.1 is valid for an arbitrary selection of vertices. Results could be improved by making a wiser selection of vertices. Notice that if the set of precolored vertices form a clique, then each vertex will be constrained to only one color by the general constraint described in section 3. Perhaps this is the best choice of vertices. The results in Table 2 are actually of vertex choices with high proximity and high clique probability. Another test run was made selecting the highest degree vertices in the graph, irrelevant of proximity. Results from this test run took more time and enumerated more valid colorings than Table 2's results.

The best selection of vertices is most likely graph dependent. There might be cases where the largest clique has sparse connectivity to the remainder of the graph and consequently the imposed constraints can't influence a wide area. On the other hand, selecting vertices that are maximally separated may work well in tandem with the Pigeonhole technique described in the next section.

4.1.2 - Pigeonhole Technique

All the vertices adjacent to vertex *a* in Figure 9 have been constrained to a subset of colors from *C*. If $X_{bcd} = X_b \cup X_c \cup X_d \neq C$, vertex *a* can be Pigeonholed to one color from the set $C\text{-}X_{bcd}$. It is possible, after applying constraints from 4.1.1, that situations like this will occur. In this event, the color information for vertex *a* may be smoothed from the BDD to reduce the problem complexity. A note must be made that vertex *a* has been Pigeonholed to color *x*.

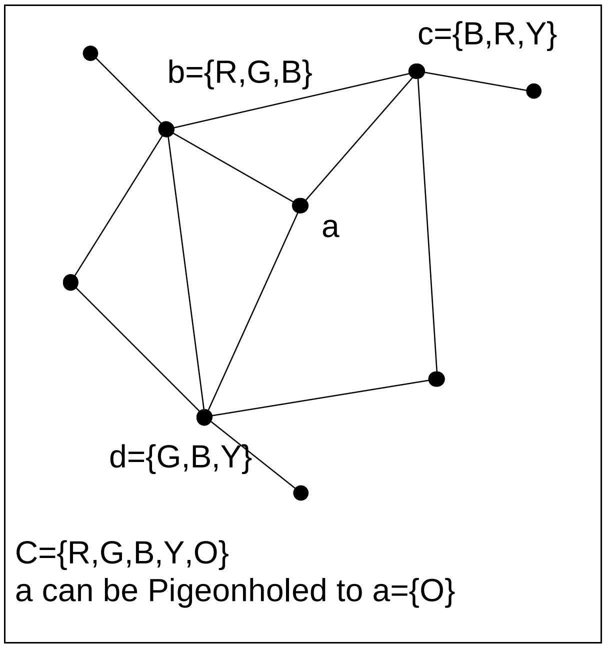

Figure 9: Vertex *a* is Pigeonholed.

4.2 - Partitioning

Since one subgraph's coloring permutations are effectively multiplied with another subgraph's coloring permutations in the complete BDD, it seems reasonable that some forms of partitioning might reduce the problem size. Figure 10 shows a graph with a possible partition of left and right subgraphs. Any path between the two subgraphs must pass through vertices *a*, *b* or *c*. Let subgraph G_l be the vertices to the left and inclusive of the bold edge and vertices *a*, *b* and *c*. Also, let subgraph G_r be the vertices to the right and inclusive of vertices *a*, *b* and *c*. Assume it is possible to color these subgraphs separately. If valid colorings exists for each subgraph using *C*, there is no need to combine these two BDDs into a multiplicatively larger BDD. Instead, by smoothing the two separate coloring BDDs in all vertices except *a*, *b* and *c* and multiplying these two BDDs together, a combined possibly tighter constraint on vertices *a*, *b* and *c* is generated. This is the same constraint that would have been generated if the two separate coloring BDDs were multiplied without smoothing and is therefore the same constraint as in section 3. This new constraint can now be applied separately to G_l and G_r. The obvious advantage here is the additive size of coloring BDDs for G_l and G_r is smaller than the size of *G* proper's coloring BDD yet all information is preserved.

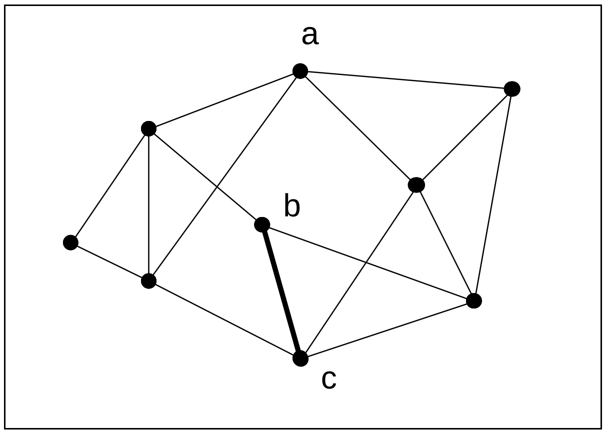

Figure 10: A Possible Partitioning

This partitioning scheme can be applied recursively. G_l and/or G_r can be partitioned and etc. Furthermore, theorem 4.1.1 will still hold. A further reduction in complexity could be gained if vertices are chosen from the sets of common vertices.

The difficulty in this partitioning scheme is choosing a small set of vertices that if removed, divide the graph into two subgraphs. Intuitively, possible hints for this selection can be gathered from the original BDD representation of the graph after sifting.

4.3- BDD Construction and Ordering

Correct BDD construction should begin with the level zero vertex constraint. Building a BDD generally from bottom to top is most efficient. To further constrain the coloring BDD and hopefully keep the size small as well as speed up construction, vertices at the lowest levels with adjacencies to previously constrained vertices should be built.

Correct BDD ordering would put the most constrained vertices near the top. This would include any precolored vertices as wells as higher degree vertices. This ordering, highest degree to lowest degree vertices, was tried on the examples of section 4.1.1. The BDD node requirement was reduced substantially. Unfortunately, the code built the BDD from top to bottom and paid a large time penalty. This will be remedied in future revisions.

5- Conclusions

This report described BDD techniques useful for graph coloring and related problems. Some of these techniques have been implemented and results are available. These results show that efficient two-coloring and would be odd-circuit producing edge finding BDD techniques are quite practical.. The resulting BDD contains only two minterms; colorings in both senses. The maximum number of BDD nodes is roughly two times the number of vertices in the original graph. The time spent is on the order of the number of edges in the original graph. On the other hand, the results show that chromatic numbering and valid colorings techniques work for very small cases due to exponential problem growth. Possible methods for extending these results to larger graphs were presented. These possibilities focus on tighter constraints and partitioning.