The FMSAT Satisfiability Solver: Hypergraph Partitioning meets Boolean Satisfiability

Arathi Ramani, Igor Markov

{ramania, imarkov}@eecs.umich.edu

February 6, 2002

Abstract

This report is intended to present the Fiduccia Mattheyses (FM) heuristic for hypergraph bipartitioning as a method for solving large Boolean Satisfiability (SAT) problems. We hope to extend the success of the FM heuristic in the partitioning domain to satisfiability. This report outlines how a SAT problem can be viewed as a partitioning problem, which argues for the applicability of FM. Further, we present the software architecture and implementation of our SAT solver, inspired by a previous highly successful implementation of FM for hypergraph bipartitioning. We also briefly discuss experimental results that justify our belief that the FM heuristic shows promise in the satisfiability domain.

1 Introduction

1.1 Boolean Satisfiability

A Boolean satisfiability (SAT) problem is defined as follows. Given a set of variables $x_1, x_2 ... x_n$, each with a $\{0/1\}$ value and a set of clauses $C_1, C_2 ... C_n$, where each clause consists of logical-or (v) connected variables, and a Boolean formula in conjunctive normal form (CNF):

$$C_1 \wedge C_2 \bar{\wedge} \ldots \wedge C_n$$

find an assignment of values to the variables such that the formula is true, or show that there is no such assignment.

SAT was the first problem shown to be NP-complete [1] and requires worst-case exponential complexity to find a satisfying assignment, or to prove one does not exist.

1.2 Satisfiability Solvers

A great deal of research effort has been devoted to developing fast and efficient satisfiability solvers, because SAT has a wide range of applications including VLSI routing, planning and scheduling problems, and equivalence checking for formal verification.

SAT solvers fall into two domains, complete and incomplete (exact and inexact). Complete algorithms explore the entire search space, therefore, if a solution exists, they are guaranteed to find it. Incomplete algorithms typically use heuristics to avoid exploring the entire search space. This allows them to perform faster than complete solvers, but they are not guaranteed to find a solution and therefore cannot cannot prove unsatisfiability. Complete solvers, while able to prove unsatisfiability, have exponential worst case complexity.

1.3 The Fiduccia Mattheyses heuristic for hypergraph bipartitioning

The Fiduccia Mattheyses (FM) heuristic was developed for hypergraph bipartitioning [2], which is the problem of dividing a hypergraph into two partitions while minimizing the number of hyperedges that are "cut". A hyperedge is cut when not all its vertices lie in the same partition. The FM heuristic uses a greedy local search procedure with hill-climbing and backtracking techniques, explained in detail in Section 3, and has been very effective in solving large partitioning problems. While its empirical success has not been fully substantiated by theoretical analyses, there is a vast amount of evidence that that in the partitioning domain this type of hill-climbing is very competitive with other types. In this work, we discuss our efforts to extend the heuristic to the satisfiability domain.

Leading-edge exact SAT solvers employ constructive

algorithms, i.e., they attempt to construct a solution piece by piece and are not able to offer a complete solution until the end. The FM algorithm, like most inexact algorithms, is iterative, i.e., it starts with a 'tentative solution' and attempts to improve it through a series of small changes. The best solution seen so far is available at any time. The two approaches differ because SAT is a constraint satisfaction problem, while min-cut partitioning is an optimization problem.

For its decision making process, FM uses a greedy local search. The search space is the set of variable assignments that differ form the current assignment by exactly one variable. The heuristic picks the assignment that offers the greatest improvement in solution quality over the current assignment in terms of the objective function, and changes the variable in question. Greedy local search has been successfully employed in other inexact SAT solvers in the past, most notably GSAT and WalkSAT. However, differences exist between the FM algorithm and those used by GSAT and WalkSAT, particularly with regard to hill-climbing techniques. These are discussed in detail in Section 3.

1.4 SAT and Max-SAT

The problem of satisfiability (SAT) asks whether or not an assignment to the problem variables can be found such that each clause in the problem evaluates to true. The Max-SAT optimization problem, however, asks to identify a variable assignment that satisfies the greatest possible number of clauses. The constraint-satisfaction version of SAT is a special case of Max-SAT, since all clauses can be satisfied at once. From the Max-SAT perspective, unsatisfiable SAT instances are of greater interest. Existing SAT solvers normally just return a negative answer for unsatisfiable problems, and provide no information about the solution. A Max-SAT solver will return its best-seen solution and point out a small set of clauses responsible for unsatisfiability. If such clauses are removed, the modified instance will be satisfiable. This feature is useful in several CAD applications that rely on solving instances of SAT.

The FM heuristic uses a backtracking technique that allows it to recover the best solution encountered in its progress even though it may have made subsequent uphill moves. Therefore, if it is required to terminate after a certain period of time, it will provide the best solution seen before exiting. This backtracking technique is useful for Max-SAT problems, where the solver could satisfy a relatively large number of clauses in a very short time.

The remainder of this document is organized as follows. In section 2, we explain the working of the FM heuristic in detail, and discuss guidelines for its implementation. In section 3, we outline other local search procedures for SAT and illustrate differences between these procedures and the FM heuristic. In section 4, we explain how FM is applied to SAT problems and discuss some implementation issues with an illustrative example. In section 5, we briefly discuss our implementation. In section 6, we conclude the report with some remarks on the possible utility and applications of FM.

2 The Fiduccia Mattheyses Heuristic for Hypergraph Bipartitioning

2.1 Algorithm Description

The FM heuristic for bipartitioning circuit hypergraphs is an iterative improvement algorithm. FM starts with a possibly random solution and changes the solution by a sequence of moves which are organized as passes. At the beginning of a pass, all vertices are free to move (unlocked) and each possible move is labeled with the immediate change in total cost it would cause; this is called the gain of the move. Solution cost is measured by the number of hyperedges cut in a partitioning solution. Positive gains reduce solution cost, while negative gains increase it. Iteratively, a move with highest (but not necessarily positive) gain is selected and executed, which may result in a lower, higher, or unchanged solution cost. The moved vertex is locked, i.e. is not allowed to move again during that pass. Since moving of a vertex can change gains of adjacent vertices, all affected gains are updated. Selection and execution of a best-gain move, followed by gain update, are repeated until every vertex is locked. After every vertex has moved exactly once, we get a partitioning solution symmetric to the the one at the beginning of the pass, thus having the same cost. Then, the best solution seen during the pass is adopted as the starting solution of the next pass. The algorithm terminates when a pass fails to improve solution quality.

2.1.1 Hill Climbing in FM

Hill climbing in FM is achieved by forbidding move repetition, and executing all non-repeating moves in a pass. Since moves are ordered according to gain, the heuristic executes a negative-gain move in the absence of a move with positive or zero gain. This type of hill-climbing has been shown empirically to be very

successful for partitioning (see [3] for results of FM in partitioning).

2.2 Software Architecture

Kahng et al [3] describe a seven-component software architecture that serves as a guideline for the implementation of "move based" heuristics for hypergraph partitioning (the FM heuristic is move based). The architecture contains the following components.

Partitioning Interface: Formally describes the input and output to partitioners without mentioning internal structure and implementation details.

Initial Solution Generator: Generates a starting solution (usually random) that satisfies problem constraints.

Incremental Cost Evaluator: Evaluates cost for a given partitioning, and updates the cost value when the partitioning is changed by applying moves.

Legality Checker: Verifies whether a partitioning satisfies a problem constraint.

Gain Container: A general container for moves, which should support quick updates of the gain for a move, and fast retrieval of a move with the highest gain.

Move Manager: Responsible for choosing and applying one move at a time, undoing moves on request, incrementally computing change in gains due to a move, and updating the gain container.

Pass based Partitioner: Solves partitioning problems by applying incrementally improving passes to initial solutions.

The UCLA Physical Design Tools Release [8] includes an FM partitioner that uses the software architecture describe above.

2.3 Gain Update in Partitioning

This section explains how gains are evaluated and updated in bipartitioning. For bipartitioning, the state of a hyperedge (whether cut or uncut) depends strictly on the number of vertices in each partition. If a hyperedge has a non-zero number of vertices in each partition, it is cut. In [3], the number of vertices in each partition for a given hyperedge is recorded as a "tally" for that hyperedge. A tally for some edge e has the form $\{a,b\}$ where a is the number of vertices of e in partition 0, and b is the number of vertices of e in partition 1.

Vertex gain is calculated based on the tallies for each edge the vertex lies in. For example, consider a vertex in partition 0, with 3 edges incident on it. The gain of this vertex is the resulting difference in cost if the vertex were moved to partition 1. For the purpose of this example, assume that the tallies for the three edges incident on this vertex are {1,6}, {4,4} and {6,0} respectively.

We analyze the effect of moving the vertex to partition 1 for each of the three edges.

Edge 1: The edge is already cut, and only one vertex (the one under consideration) lies in partition 0. By moving this vertex to partition 1, the edge will be uncut, so there is an increase in gain of 1 for the vertex with respect to the first edge.

Edge 2: This edge is already cut, moving the vertex to partition 1 does not alter the status of the edge in any way, so there is no change in gain with respect to this edge.

Edge 3: The edge was uncut (all vertices in partition 0), and moving the vertex to partition 1 will cut the edge, so there is a decrease in gain with respect to this edge.

Therefore, the total gain of the vertex is $(1 + 0 + -1) = 0$.

Assuming the move is made, and the vertex locked, it is now necessary to update the gains of all unlocked vertices on the three edges. For the purpose of this example, we assume that no other vertex was locked on any of the three edges.

Edge 1: This edge is now uncut in partition 1, so moving any unlocked vertex to partition 0 will cut it. The gains of all unlocked vertices in this edge will decrease by 1 with respect to this edge.

Edge 2: This edge will remain cut if any one vertex is moved to another partition, so there is no change in gain for any unlocked vertex with respect to this edge.

Edge 3: This edge is now cut, and only one vertex (which is locked) lies in partition 1. For all the remaining vertices in partition 0, the edge will stay cut even if another vertex is moved to partition 1. However, since the edge was previously uncut, these vertices would have earlier had a negative gain (as penalty for cutting the edge). Now that the edge is cut, there is no penalty to be paid for moving these vertices, so their gains increase by 1 with respect to the edge.

3 Local Search in SAT

Algorithms using local search have been applied to SAT before, notably the GSAT algorithm introduced by Selman et al [4] in 1992. GSAT performs a greedy local search for a satisfying assignment of a set of propositional clauses. The procedure starts with a randomly generated truth assignment, and then changes ("flips") the assignment of the variable that leads to the largest increase in the total number of satisfied clauses. Such

flips are repeated until either a satisfying assignment is found or a preset maximum number of flips is reached. The process is repeated as needed upto a preset maximum number of tries. During any try, GSAT explores the set of assignments that differ from the current one by only one variable.

The local search strategy used by GSAT was successful for some classes of problems (notably hard randomly generated formulas) (see [4] for detailed results), but lacked hill climbing ability and was sometimes prevented from finding a satisfying assignment, even when one existed, by local minima in the search space. To improve performance, the ability to perform "sideways" moves (that produce no change in the objective function, but change the variable assignment) was added to GSAT, and later, hill-climbing strategies such as simulated annealing and random walk were added to the GSAT algorithm (see [6], [7], [5]).

GSAT with random walk (WSAT or WalkSAT) improved greatly on the performance of basic GSAT for large structured problems such as planning and circuit synthesis. The algorithm used by WalkSAT is described below.

With probability p, pick a variable occurring in some unsatisfied clause and change its truth assignment
With probability (1-p), follow the standard GSAT scheme,
i.e. pick randomly from the list of variables that gives the largest decrease in the total number of unsatisfied clauses.

GSAT and WalkSAT both perform a local search that is similar to the one performed by FM. The difference between the algorithms lies in the hill climbing strategy. FM moves every variable in every pass, only the order of the moves change depending on the gains. Hill climbing is performed as part of the pass in the event that only negative gain moves are possible. There is no randomization, except in the initial assignment. GSAT, on the other hand, allows only positive gain moves, and in WalkSAT, moves are selected randomly. It is not necessary to move every variable in GSAT and WalkSAT and some variables may never be flipped in the search for a solution.

4 Implementation of FM for SAT

4.1 A Motivating Example

The following example is used to illustrate gain calculation for a simple 3-clause SAT problem.

$$(a\ V\ b\ V\ c)\ \Lambda\ (a\ V\ b'\ V\ c')\ \Lambda\ (a'\ V\ b\ V\ c)$$

Let us assume an initial assignment of 0 to all variables. With this assignment, two clauses are satisfied. The cost for each variable is the number of unsatisfied clauses in which the variable occurs. At this point, the cost for each variable is 1. (since each variable is in all the clauses). We now analyze the change in gain when each variable is moved to the other partition. Gain is the difference in cost after the variable is moved.

Gain = (Cost before move) − (Cost after move)

For variable a:
Gain = 1 − 1 = 0
For variable b:
Gain = 1 − 0 = 1
For variable c:
Gain = 1 − 0 = 1

From this calculation, we see that b and c are the highest gain variables. One of them is moved (assigned a value 1), locked, and the change in gain for the other two variables is now computed. In this case, picking either one leads to a satisfying assignment, so the algorithm stops.

4.2 The FMSAT Solver

Our work on the FMSAT solver is based strongly on the software architecture described in [3], and follows the seven-component approach. In this section, we describe the FMSAT solver and discuss implementation issues for some of its components.

In a SAT problem, solution quality is measured by the number of clauses that are satisfied under the given variable assignment. The solver starts with an initial variable assignment and iteratively performs passes to improve the solution quality. At the beginning of each pass, all variables are free to "move" (moving a variable is assigning it a value complementary to its current value). As in partitioning, Each move is labeled by the immediate change in solution quality it would cause; this is called the gain of the move. A positive-gain move reduces the number of violated clauses, while a negative-gain move increases it. The highest-gain move available is selected and executed, the moved variable is locked, and all variables that are present in the same clauses as the moved variable have their gains updated. This process is continued until all variables are locked. Then, the best solution seen during the pass is adopted as the starting solution for the next pass. The algorithm terminates under either of two conditions:

1. It fails to improve solution quality (does not increase the number of satisfied clauses)
2. It results in a cost of 0, i.e. all clauses are satisfied

4.3 Clause Evaluation in FMSAT

The cost evaluator in FMSAT is required to evaluate whether a clause is satisfied or not, under the current variable assignment. This is done by comparing the assigned values of the variables in the clause to their polarities in the problem description. Each clause maintains a description of itself, in terms of its constituent variables and their designated polarities. In the evaluation process, variable values are compared against polarities. If at least one constituent variable has been assigned a value consistent with its designated polarity, the clause evaluates to true.

4.4 Gain Update in FMSAT

In this section, we explain how gains are computed and updated in the FM SAT solver.

After every move (variable assignment), we are required to perform the following actions:

1. Change the evaluations (true/false) of all clauses in which the variable occurs to reflect the new assignment.
2. Update the gains of all other variables in the clauses in which the variable occurs to reflect the changed clause evaluation.

To describe gain update, consider two variables, *v1* and *v2*, and assume that *v1* and *v2* occur together in exactly one clause. Assume that we need to update the gain for *v2* if *v1* is moved. To do this, we need to consider the effect of moving *v2* on the clause before and after *v1* is moved.

Moving a variable in a clause can result in one of three "clause transitions":

1. True -> False: This occurs when the variable is the only variable in the clause whose value its consistent with its designated polarity. Changing the value results in the clause evaluating to false.
2. False -> True: No variable in the clause has an assignment consistent with the designated polarity. As soon as one variable's value is changed, the clause evaluates to true.
3. True -> True: At least one variable other than the variable being moved has an assignment consistent with its designated polarity. In this case, the clause remains true regardless of the value of the moved variable.

The table illustrates the possible changes in gain for *v2* for different clause transitions when *v1* is moved.

Clause Transition on Moving *v2*		Change in
Before Moving *v1*	After Moving *v1*	Gain of *v2*
T -> F	T -> T	1
F -> T	T -> T	1
T -> T	F -> T	1
T -> T	T -> F	-1
T -> T	T -> T	0

Table 1: Change in gain after moving variable

5 Implementation

The FMSAT solver was modeled on the software architecture described in [3] and written in the C++ programming language. The solver contains the following major components.

Initial solution generator
Cost Evaluator
Gain Container
Move Manager
Pass-Based Solver

The components perform the same functions as those described in Section 2, however, the "tallies" used by the cost evaluator are slightly different from the tallies used in partitioning, and are explained in detail later in this section.

Standard Template Library (STL) container classes and algorithms are used extensively in almost all major data structures and component classes. The pass based solver uses a description of the problem that is maintained in a separate class, the problem description contains lists of the constituent variables and clauses. Each variable maintains a list of all the clauses it occurs in, and each clause maintains a description of itself with the identification numbers and polarities of its constituent variables. Maintaining this information locally within clauses and variables allows for more efficient gain updates.

5.1 Clause Tallies in FMSAT

The cost evaluator is required to evaluate the change in the objective function as the result of a move. To do this, it needs to have a method for checking whether a clause is violated or not. The move manager is also needs this information for performing gain updates. In

SAT, as in partitioning, this is done efficiently using a "tally" for each clause. For SAT, a tally is of the form $\{ a , b\}$ where a is the number of variables in the clause that are assigned with the same polarity as their designated polarity in the clause description, and b is the number of variables that are assigned a polarity opposite to their designated polarity. Using this format, a clause is violated if $a = 0$, and is satisfied for any positive value of a. Clause tallies are updated after a move for all clauses containing the moved variable.

From the above description of the solver and the reduction of SAT to partitioning, it is clear that we can apply this heuristic to SAT problems, with the objective of either returning no violated clauses if the instance is satisfiable (SAT), or minimizing the number of violated clauses (Max-SAT).

6 Empirical Results

7 Conclusions and Future Work

While we have not included numeric results in this report since this work is experimental and the solver is still undergoing considerable tuning, we wish to stress that the FMSAT solver has an average solution quality and run-time very close to that of WalkSAT for most instances. However, it appears that at this time, the solver gets less competitive as the problem size increases. However, it is very likely that this is an implementation issue, and that optimizing the most frequently performed operations, such as gain update, will greatly improve runtime. We have observed that, over a single run of FMSAT, the number of clauses violated is greatly reduced during the first few passes. After that, runtime increases as fewer and fewer positive-gain moves exist. However, this property is very useful for Max-SAT problems, where the objective is to satisfy a large percentage of the clauses in a short time, and the problem size is typically very large. We believe that a highly optimized version of FMSAT would be an effective Max-SAT solver. Overall, we feel that we have demonstrated that the FM heuristic shows promise in the domain of inexact SAT solvers.

References

[1] S.A. Cook, "The Complexity of Theorem Proving Procedures" *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (1971)* pp. 151–158.

[2] C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions" *Proc. 19th IEEE Design Automation Conference,* IEEE, Las Vegas, NV, 1982, pp. 175–181.

[3] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Design and Implementation of Move-Based Heuristics for VLSI Hypergraph Partitioning" , *ACM Journal on Experimental Algorithms,* (vol. 5), 2000.

[4] Selman, B., Levesque, H.J., Mitchell, D. "GSAT: A new method for solving hard satisfiability problems" *Proc. of the 10th National Conference on Artificial Intelligence,* (AAAI-92), pages 440–446, San Jose, CA, USA, 1992.

[5] Selman, B., and Kautz, H. , "Domain independent versions of GSAT: Solving large structured satisfiability problems",*13th International Joint Conference on Artificial Intelligence,* pp. 290-295.

[6] B. Selman, H. Kautz, and B. Cohen, "Local search strategies for satisfiability testing", *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* 26:521–532, 1996.

[7] B. Selman, H. Kautz, and B. Cohen, "Noise strategies for improving local search", *Proc. of the 12th National Conf. on Artificial Intelligence,* (Seattle, WA), 1994, pp. 337–343.

[8] http://vlsicad.eecs.umich.edu/BK/