Analyzing Decision Heuristic Effectiveness in Boolean Satisfiability Solvers

Victor A. Ying

Supervised by Professor Sharad Malik

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

May 2, 2016

I hereby declare that this report represents my own work in accordance with University regulations.

/s/Victor A. Ying

Analyzing Decision Heuristic Effectiveness in Boolean Satisfiability Solvers

Victor A. Ying

Abstract

Boolean satisfiability (SAT) solvers have achieved impressive performance in practical settings. They are now heavily relied upon in formal verification and other industry applications. However, their degree of success not well understood, and it is not known how much room there is for improvement in performance through continued improvements in decision heuristics, whose development has been responsible for orders of magnitude of performance improvements in the past. In this report, we define a notion of dependency relations between events in solver execution that allows us to identify what portions of the work done by the solver on any given instance is required or could be avoided by changing the decision heuristic. We carry out experiments with practical application-focused benchmarks and find that most branches chosen by the decision heuristic are required, and a minority of work done by the solver is wasteful. This suggests limited improvements are still possible by improving decision heuristics alone if other aspects of SAT solvers are not improved.

Acknowledgments

This work was done under the generous supervision of Professor Sharad Malik, who has been a patient and thoughtful mentor, and always a source of good ideas and encouragement. Above all else, he has been a great teacher for me ever since I started in ELE 206 nearly three years ago. I would also like to thank Professor Aarti Gupta, who has offered her time and ideas to this project, and exposed me to many interesting and relevant ideas through her class on Automated Reasoning About Software.

I am thankful to the friends and family who have supported me and given me access to a wealth of opportunities at Princeton, from which I have benefited greatly. My experience in the Department of Electrical Engineering undergraduate program has deeply shaped my interests and thinking. The recent implementation of the senior thesis requirement led to several very interesting conversations at the end of my junior year that have shaped my career path. One of those conversations resulted in this project.

This work was supported by awards from the Kamran Rafieyan '89 Fund for Undergraduate Research as well as from the Rudlin Independent Work/Senior Thesis Fund, administrated by the Department of Electrical Engineering and the School of Engineering and Applied Sciences.

Contents

1 Introduction

The Boolean Satisfiability Problem (SAT) has received sustained attention for decades as a famously intractable problem, and was the first problem to be shown to be NP-Complete [1]. Nonetheless, advancements in SAT solvers have led to dramatically improved performance in industry applications, and modern solvers are routinely used to solve instances with hundreds of thousands or millions of variables and clauses. As SAT solver performance has improved, SAT-based approaches to many NP-Hard problems have proliferated. Problems in automated verification and synthesis, scheduling, and cryptography are regularly being solved with SAT-based approaches. Improvements in core algorithms and improvements in our understanding of the behavior of SAT solvers on instances generated from different application domains are likely to spur new and more powerful applications.

It is well known that the design of decision heuristics has a major effect on solver performance. The variable-state independent decaying sum (VSIDS) heuristic has been widely adopted in solvers [2], and subsequent progress was made in implementing VSIDS more efficiently [3] and augmenting it with phase saving [4]. These improvements have been widely adopted because they empirically yield reduced solver runtimes [5]. However, our understanding of VSIDS has remained limited, and why VSIDS has proven more effective than other heuristics remains an open research question [6].

This projects seeks to instrument a modern conflict-driven clause learning (CDCL) SAT solver to obtain execution traces which may be analyzed in greater depth to better understand the successes of the VSIDS heuristic in industry applications, and to quantify room for improvement. Our work is based upon MiniSat 2.2.0, a popular, small, and extensible CDCL solver that won the Main Track of SAT-Race 2006 and SAT-Race 2008 [3]. This work is inspired by the well established body of work in using a trace emitted by a solver as a proof of the unsatisfiability of an instance [7]. Proof traces have gained importance for validating SAT solvers during development, in addition to their use in generating cores and interpolants which are used in many applications [8]. Recently, there has been research into "compressing" proofs after they have been generated by SAT solvers by analyzing them to construct shorter equivalent proofs [9–13]. These previous efforts have not used execution traces to address the question of how well the solver did in deciding the (un)satisfiability of

an instance in the first place, and whether it could have done better.

In this project, by instrumenting the popular MiniSat solver to emit a detailed execution trace and then analyzing the generated traces, we investigate how heuristically chosen branches used in SAT solvers affect the progress of the solver during its run. Our analysis differs from an analysis that views the execution trace as providing a proof in that it tracks not logical dependencies but a larger class of dependencies that arise from BCP and conflict analysis, and applies just as well to satisfiable instances as unsatisfiable instances. This identification of dependencies produces a rich dataset that may be analyzed to better understand how CDCL solvers make progress within solving one instance. These dependency relationships are used to define for each instance the set of dependencies that are required by the solver's final result, which is either a complete satisfying assignment or learning the empty clause. By distinguishing between such required work and wasted work, we may compare the "productivity" of CDCL search across different benchmark instances, and see how this is affected by the decision heuristic, solver restarts, and other solver parameters. This data may further guide SAT solver research by showing where there might be room for improvement in SAT solvers.

2 Preliminaries

In our discussion we assume some familiarity with the properties of Boolean formulas expressed in *conjunctive normal form (CNF)* [1] In this section, we will define some important concepts used in CDCL solvers. A more thorough explanation of CNF formulas and the common features of CDCL SAT solvers may be found in [14].

For any CNF formula, a SAT solver must indicate the instance is *satisfiable* and and produce an assignment of a logic value 0 or 1 to each variable such that the formula evaluates to 1, or indicate the instance is *unsatisfiable* because no such assignment exists. A set of assignments is *complete* if every variable has an assigned value of 0 or 1. W literal is *satisfied* if its variable has been assigned such that the literal evaluates to 1, and it is *falsified* if it evaluates to 0. A literal is *unassigned* if its variable has not been assigned a value. A clause is *satisfied* if it contains at least one satisfied literal, and it is *falsified* if all of its literals are falsified. A clause is *unit* if it is not satisfied and has exactly one unassigned literal.

CDCL solvers are an extension of DPLL solvers [15]. In a DPLL solver, every variable starts out unassigned, and the solver proceeds by repeatedly making all implied assignments it can find, and then picking an unassigned variable and assigning it a value, known as picking a *branch* or *decision*. The *unit propagation* rule is used to find implied assignments. It states that a unit clause implies that its one unassigned literal must be satisfied by assigning the appropriate value to its variable. The process of finding implied assignments is also known as *Boolean constraint propagation (BCP)*. For each such *implied* assignment, the clause that became unit and implied the assignment is known as the *antecedent* of the assignment.

The number of decisions that have been made at any time during a DPLL solver run is known as the *decision level*, and every assignment is associated with the decision level at the time the assignment was made. When a clause becomes falsified, this is a *conflict*, indicating the current set of assignments cannot be part of a satisfying assignment. After a conflict is found, one or more decisions and their resulting implications must be undone so that no clause is falsified. This is known as *backtracking*. In this report, we call an

[1] A CNF formula is a logical AND of a set of clauses. A *clause* is the logical OR of a set of literals. A *literal* is a Boolean variable or its negation.

assignment *active* at any moment in time if it has been made and has not been undone by backtracking, and otherwise we call it *inactive*. If backtracking is not possible, then the solver may conclude the formula is unsatisfiable.

In a CDCL solver, every conflict is analyzed to produce one or more *conflict clauses* that are added to formula, pruning the future search space. Conflict clauses are also called *learned clauses* to contrast with the *original clauses* provided in the instance. Clauses may also be deleted from the clause database during a solver run to limit memory usage. In this report, we call a clause *active* at any moment in time if it has been added to the solver data structures, whether as an original or learned clause, and has not been deleted.

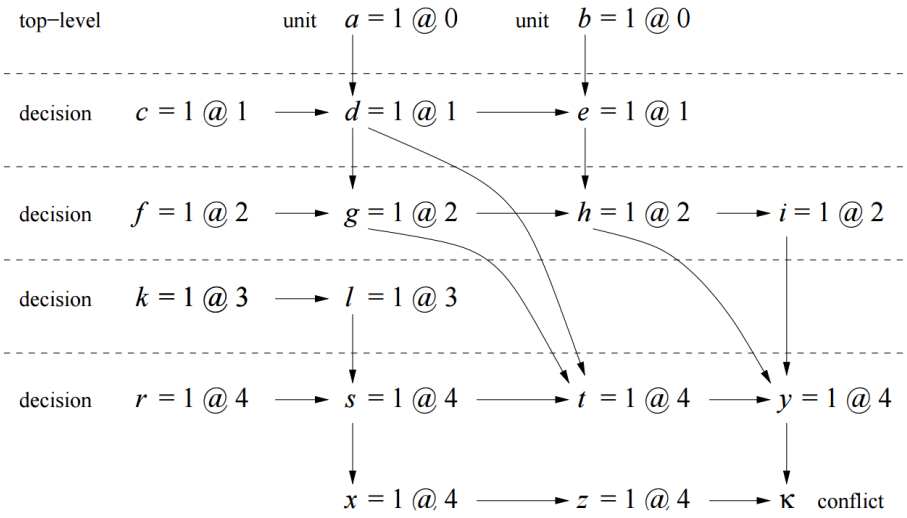

Figure 1: Example implication graph with decision levels shown. The incoming edges at each assignment node indicate the antecedent clause. For example, the antecedent of the assignment $y = 1$ is the clause $(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$. The conflict κ corresponds to the clause $(\bar{y} \vee \bar{z})$ whose two literals have become falsified. The 1-UIP conflict clause is $(\bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i} \vee \bar{s})$. Taken from [16].

The conflict analysis may be understood by examining the *implication graph* at the time of the conflict, a directed acyclic graph that shows the dependencies of all current assignments. The implication graph has one node for every assignment, as well as a special node κ representing the conflict. Implied assignments have incoming edges from the all the previous assignments needed to make their antecedent unit, and κ has incoming edges

from all the assignments needed to produce the conflict. An example is shown in Figure 1. Conflict clauses are always chosen to be the negation of some sufficient conditions for a conflict. Therefore, the conflict clause represents is a necessary condition for avoiding conflicts, so its addition does not change the satisfiability of the formula. Most current CDCL solvers, including MiniSat, use the conflict analysis scheme known as 1-UIP [17].

CDCL solvers usually employ a form of *non-chronological backtracking* known as *backjumping* after each conflict, in which assignments are undone to the greatest extent possible subject to the constraint that the new conflict clause is unit after backtracking, allowing the learned clause to determine the next assignment by unit propagation. Solvers may also performs occasional *restarts* by backtracking to decision level 0—undoing all decisions and keeping only implications that do not depend on any decisions—after a conflict.

3 Theory

3.1 Motivation and approach

It is well known that the majority of the runtime of most CDCL solvers on typical benchmarks is spent on BCP [2,14]. Meanwhile, the largest demands on memory in a CDCL solver come from the data structures required to store clauses as well conduct efficient BCP. The size of these data structures grows in proportion to the number of active clauses [2,3,14,15]. With this understanding, our focus is on identifying which learned clauses and which implied assignments contributed to the solver's progress, and which were merely incidental in the solver's run, and how the decision heuristic's choices lead to those implied assignments and learned clauses.

There are various ways work the solver might perform during its run might not contribute to the solver's final answer. If the solver finds some implied assignment, but that implied assignment does not help make any other clauses unit or falsified before it becomes inactive through backtracking, then the work done in carrying out that implied assignment did not help the solver reach its result. Similarly, if some learned clause is constructed in some conflict analysis, and shortly afterward a completely unrelated conflict leads to backtracking such that the learned clause never becomes unit or falsified again at any point during the solver's run, then the time and memory spent discovering the learned clause is ultimately unproductive. We wish to be able to quantify the amount of unproductive work done by the solver in order to gain insight into where there is room for improvement in solver design.

To this end, we define every assignment made during a solver run and every addition of a new active clause to be an *event*. This includes decisions and implied assignments, as well as the addition of both original clauses and learned clauses. We seek to define dependency relations between these events, as well as dependencies of the solver's final result on these events.

Note that every time a value is assigned to a variable is considered a distinct event, even if the same value has been assigned to the same variable before. Similarly, if there are any repeated additions of the same clause, these are treated as separate events. This allows

us to distinguish the dependencies and dependents of each branch, implied assignment, and clause addition, each of which has associated time and memory costs. We shall proceed to define the dependency relations used in our analysis.

We are trying to understand the effects of decision heuristic design. Therefore our definitions of dependency relations treat the implementation of BCP and conflict analysis as fixed, and we are defining our dependency relations between the inputs and outputs of these portions of the solver. In contrast, we consider every branch to be an independently chosen event, and do not consider how the decision heuristic and phase saving depends upon past events.

3.2 Dependency definitions

Every implied assignment is found through unit propagation. It directly depends not only on the existence of its antecedent, but also all the assignments that make its antecedent unit. More precisely, every implied assignment directly depends on the addition of its antecedent clause, and on the previous assignments that caused all but one of its antecedent's literals to become falsified. These assignments may be branches, which have no dependencies as they are chosen by the heuristic, or other implied assignments, which have their own direct dependencies defined in the same way.

The addition of a clause may be due to the clause being an original clause, in which case it has no dependencies, or it may be a learned clause added due to some conflict analysis. Since every conflict produces one conflict clause and causes backtracking, we consider a conflict, the addition of the associated conflict clause, and the resulting backtracking to be one event. We shall refer to this single event as a learned clause addition. Every learned clause addition directly depends on the falsified clause associated with the conflict, as well as all the assignments that made the clause falsified.

Dependency is a transitive relation: for any three events X, Y, and Z, if X depends on Y and Y depends on Z, then X *indirectly depends* on Z through Y.

The dependencies upon assignments we have defined correspond directly to edges in implication graphs: an implication or conflict directly depends upon a previous assignment if there is an edge from the previous assignment to that implication or conflict in some

implication graph. We may imagine the dependencies we have defined as a large dependency graph, with a vertex for every event and an edge for every dependency. The implication graph for any conflict is exactly the subgraph of this dependency graph induced by the vertices representing the conflict and the active assignments at the moment of the conflict. The dependency graph thus simultaneously contains every implication graph of the solve run. Since the information in the solver state used during conflict analysis is captured entirely in the implication graph, our definition of dependency is sufficiently inclusive such that every event that might influence the construction of each conflict clauses through unit propagation and conflict analysis is a dependency of the corresponding learned clause addition event.

Notice that we are only defining each event to only be directly dependent on previous events that are still active at the time of the event. This is consistent with how an implication graph only shows implications among active assignments at one moment in time. However, our definitions give rise to indirect dependency relations across the entire solver run through the learned clauses. An event may depend on a learned clause addition, which in term is dependent on previous assignments and clauses that may have become inactive by the time of the dependent event of the learned clause.

3.3 Defining required and wasted work

The ultimate goal of a SAT solver is to return a complete satisfying assignment or else declare that no such assignment exists. There are two classes of possible *final events*. If a solver decides an instance is satisfiable, we consider this a special assignment completion event, which directly depends upon the set assignments active at that moment, which must be complete.

If a solver decides an instance is unsatisfiable, this will occur during some conflict analysis, when the solver could produce an empty clause as the conflict clause. In the solver implementation, there is no need to actually store an appropriate representation of this empty clause, as the solver is finished. However, for the purposes of our analysis, we consider the solver to have generated the empty clause, so we treat this is just be another learned clause addition, with its dependencies defined in the same way as any other learned

clause addition.

Whether the final event is an assignment completion or the addition of the empty learned clause, we may define the final event and all of its dependencies—direct or indirect—as the *required* events in the execution of the solver. We say all unrequired events do not ultimately contribute to the solver's success.

As we are trying to analyze the effect of the decision heuristic on solver success, We may also divide events into two categories another way: We may distinguish between *wasted* events and *unavoidable* events. Branches chosen by the heuristic are wasted if are not required, and they are unavoidable if they were required. For the two classes of events that have dependencies—implied assignments and learned clause additions—we say an event is wasted if it depends directly or indirectly on a wasted branch. On the other hand, if all of the branches that an event depends on are required, then that event is unavoidable. For the purposes of this analysis we also classify original clauses as unavoidable, as they have no dependencies and will be added to the solver data structures at the start of the run regardless of the design of any heuristics.

Every event is either wasted or unavoidable. Notice that required events are never wasted, so we have now divided events into three categories: required events, wasted unrequired events, and unavoidable unrequired events.

4 Experimental setup

We proceed to instrument the solver and run it on a large collection of benchmarks so that we may identify dependencies between all the events in these benchmark runs and classify events as required, wasted, or unavoidable. To avoid slowing down the solver by augmenting its data structures to do our desired dependency tracking within the solver, we instrument the solver only to emit a trace containing information on all events. We created a separate analyzer that parses these traces to perform the work of tracking dependencies and classifying events. This has the advantage that we avoid doing any dependency tracking and analysis on instances for which the solver would time out.

As computational resources and solver techniques have improved, the benchmarks used in SAT competitions have become more challenging. We therefore chose to limit ourselves to the use of all the industrial application benchmarks used in SAT Competition 2009 to obtain a good variety of benchmarks that are possible for MiniSat 2.2.0 to solve in a number of steps that does not exceed the capabilities of our analysis, which currently depends on being able to fit the events it is analyzing into memory.

We run our instrumented version of MiniSat 2.2.0 on a box with a Intel Xeon E3-1230 processor running at 3.2 GHz. We set a timeout of 300 seconds for each instance to avoid traces that are so large that they are difficult to analyze. With this timeout, we never observe the solver to use much more than half a gigabyte of memory with the default settings, so memory limitations are not a consideration

MiniSat has built-in preprocessing and conflict clause minimization [16, 18], but these features were turned off to simplify development and debugging. (It would make no substantive difference to the analysis methods if these features were turned on, so that all instances are simplified by MiniSat's preprocessing before being used, and all conflict clauses minimization would simply be part of the conflict analysis which our analysis treats as a fixed black box.) All other MiniSat parameters and settings are left at their defaults.

The remaining subsections in this section describe in some detail the implementation of the instrumentation and analysis we have already described. Readers may skip directly to section 5 without difficulty, and are encouraged to do so if they are not concerned with implementation details.

4.1 Solver instrumentation

The solver is instrumented to emit information to a trace for each event as it occurs. We wish to be able to efficiently analyze this trace in order to identify the dependency relationship between these events. As the solver knows the antecedent of every implication and the falsified clause associated with every conflict, we may easily emit this information for every implication and conflict.

We additionally emit sufficient information so that the set of active assignments and clauses at any point in the trace may be efficiently determined by reading the trace up to that point. To accomplish this, the solver emits the contents of every clause as part of each clause addition, and the value and variable being assigned for each assignment. The solver also emits the decision level to which it backtracks as part of every learned clause addition. In addition, restarts and clause deletions are emitted into the trace alongside events, although they have no dependencies or dependents.

While this trace contains a great deal of information, emitting this trace does not require augmenting the solver with large additional data structures or require the solver to access parts of its data structures when it did not previously do so, which might slow down the solver by thrashing processor caches. Assignment values are only emitted when the assignments are first made. Additionally, the only time the contents of a clause are emitted is when a clause is added, which is a time when the solver needs to access the contents of the clause anyway. At the time a clause is added, we may also emit a identifier for the clause, such as a memory address or offset where it is stored, and thereafter the clause may be referred to in the trace by that ID. This does require emitting additional information to the trace during garbage collection so that the analysis may track clause dependencies.

The details of the execution trace format are contained in appendix A. The changes made to the code of MiniSat 2.2.0 to emit traces as described in this section are shown in the form of a patch file in appendix B.1.

As an aside, the execution traces being produced are more verbose than the proof traces generated in the popular DRUP and DRAT formats by many solvers, and even the TraceCheck resolution proof format. These standard proof trace formats record the addition and removal of clauses, but not the DPLL assignments and backtracking used to find

them [8].

4.2 Trace analyzer

The source code for the analyzer is included in appendix B.2. The analyzer runs in three phases:

1. Parsing a trace to generate a dependency graph and a chronological list of events.
2. Traversing the graph to mark the required events.
3. Traversing the chronological list to generate the desired results.

We will describe the implementation of the three phases separately.

4.2.1 Parsing a trace and establishing dependencies

The parses one event at a time sequentially through the execution trace and constructs the complete dependency graph for the instance. For each event, a vertex is added to the dependency graph that contains the information concerning what that event was, and pointers to all the dependencies of the event are stored with the vertex When it is created.

When an event with dependencies is parsed, its dependencies always include one active clause, along with active assignments that made literals in that clause falsified. In order to efficiently identify the dependencies of each event, we maintain data structures that indicate the current set of active assignments and clauses at each point in time in parsing the trace. The data structures involved are much like the ones found within MiniSat itself.

To track the set of active assignments, an array of pointers to the currently active assignments, indexed by variable numbers, is maintained alongside a FIFO stack of pointers to assignments. Each assignment is added to both the array and the stack when it is parsed. During backtracking, assignments are removed from the stack and used to find the entries in the array that must be nulled out.

To track the set of active clauses, we simply maintain a mapping from clause ID's to pointers to clause addition events. Clause additions are added to this mapping when parsed,

and removed when the trace indicates the clause was deleted. The mapping may also need to be updated as indicated in the trace if the solver performed garbage collection.

During the parsing phase, a pointer to every parsed event, along with every restart and deletion, is also appended to a list of events in chronological order, in order to maintain a way to iterate through all the events, visiting each only once. When an event becomes inactive, it is not removed from the dependency graph or the chronological list. At the end of parsing, all the information in the trace has been loaded into these two data structures in memory.

4.2.2 Dependency traversal

Every event has a field in which we may mark whether or not the event is required. These fields are initialized to unrequired. At the end of the parsing phase, we have come to the final event, which we mark as required. We then recursively mark all of its dependencies as required, effectively performing a depth-first search of the dependency graph, starting at the final event. During this search, if we traverse a dependency edge to reach a vertex that has already been marked required, we know that all of its dependencies have already been marked required as well, so we avoid redundantly traversing the dependencies of that vertex.

4.2.3 Computing results

Finally, the chronological list of branches, restarts, and deletions is traversed in order, and the analyzer may count up the events in order to generate any desired statistics. If desired, the analysis may emit into an output file a chronological account of the events in the trace for further data analysis. At this stage, we have a chronological account of when required and wasted branches were chosen, when required, wasted, and unavoidable implied assignments and learned clauses were made and then removed.

It is also during this chronological traversal of events that we compute whether or not unrequired events are wasted. This computation may be done recursively, using a definition of avoidability and entailment that is equivalent to that given in section 3: An event is wasted if it is a unrequired branch or if it has at least one direct dependency that is wasted.

An event is unavoidable if it is required or has no wasted direct dependencies. For each event in the list, we recursively apply this definition to it and its dependencies as necessary until we find out whether or not it is wasted. To ensure the time complexity of this step is only linear in the size of the dependency graph, we memoize this recursive computation by storing at each vertex whether it is wasted, unavoidable, or if its status has not been computed yet.

4.3 Notes on performance and memory usage of analysis

As discussed in section 5, the vast majority of events are implications, and the solver can churn through millions of implications per second. With typical benchmarks having tens of thousands of variables, an average implication adds around eight bytes to an execution trace. Consequently, the solver produces a gigabyte worth of trace every dozen seconds or so while running.

The time required to parse a single implication or conflict cannot be bounded by any constant: backtracks may require an arbitrary number of assignments to be removed from the array of active assignments, and an event may also have an unbounded number of direct dependencies if clauses are allowed to be arbitrarily large. However, the backtracks take constant time in an amortized sense, if we count the number of active assignments as a potential function, and in practice most clauses do not tend to be large. If the number of literals in the largest clause is C, and the number of events in the input trace is N, the worst case time complexity of the parsing phase is $O(NC)$. In practice, even if there are a few very large learned clauses, the vast majority of clauses tend to have only a few literals in them, so the runtime of the parsing phase is approximately linear in N.

Each of the N events translates into one vertex in the dependency graph. If the dependency graph has M edges, the dependency traversal takes $O(N + M)$ time, as long as we are careful to avoid redundancy in the traversal. Again, in practice, M tends to grow linearly with N, so as long as the computation done in the result computation phase takes time linear in the size of the data structures, the time complexity of the entire analyzer is roughly linear in the size of the trace it is given as an input.

The analyzer described in this section was first implemented in Python due to the

author's prior proficiency with that language. Although it was written to avoid obvious inefficiencies such as storing any information redundantly, it was found that after parsing a trace, the Python analyzer would be using an amount of memory over a hundred times the size of the trace file it had parsed! For every event, the analyzer would construct at least one new object to represent the event, and if the event have dependencies, a list of dependencies would be created and stored with the event, which is another object. In CPython, each of these objects was allocated at least 64 bytes of memory. This made it relatively easy to run out of memory. A solver runtime of a few seconds easily translated into trace files that were gigabytes large. Trace files that were multiple gigabytes easily exhausted available physical memory when parsed by the Python analyzer. The runtime of this Python analyzer, when it did not run out of memory, was generally about an order of magnitude more than the runtime the solver took to solve any given instance and produce a trace.

The analyzer was subsequently rewritten in C++, and this greatly improved matters. The C++ implementation is included in appendix B.2. It was written to run quickly, but it currently lacks very carefully optimized memory management. It still constructs at least one object with the C++ `new` operator to store every event. Nonetheless, the new C++ analyzer seems to use an amount of memory always within an order of magnitude of the size of its input trace file, and, if its data structures fit in physical memory and don't need to be paged out to disk, it generally runs faster than the solver originally ran, which makes sense, as it doesn't need to actually search for implied assignments through unit propagation and construct learned clauses through conflict analysis as the solver did. However, in traversing dependencies across all events in the entire run held in memory, the analysis is very likely to thrash processor caches more than the solver did, and this may explain why the analyzer doesn't always use much less time than the solver for a given instance.

Since the availability of physical memory for the analyzer is currently a major limitation for our ability to analyze more difficult instances, optimizing the analyzer for memory usage could be a good next step. For example, the mapping from clause ID's to clauses is currently implemented as a dynamically expanding array of pointers to clause objects, with the clause ID's as the indices. Since clause ID's are not contiguous integers, this wastes a good deal of space. A smaller hashmap would do the job. The flags used to memoize the computation of requiredity and avoidability are `enum`s that take up 32 bits each for every

implied assignment, which is clearly wasteful.

5 Results

5.1 Satisfiable instances

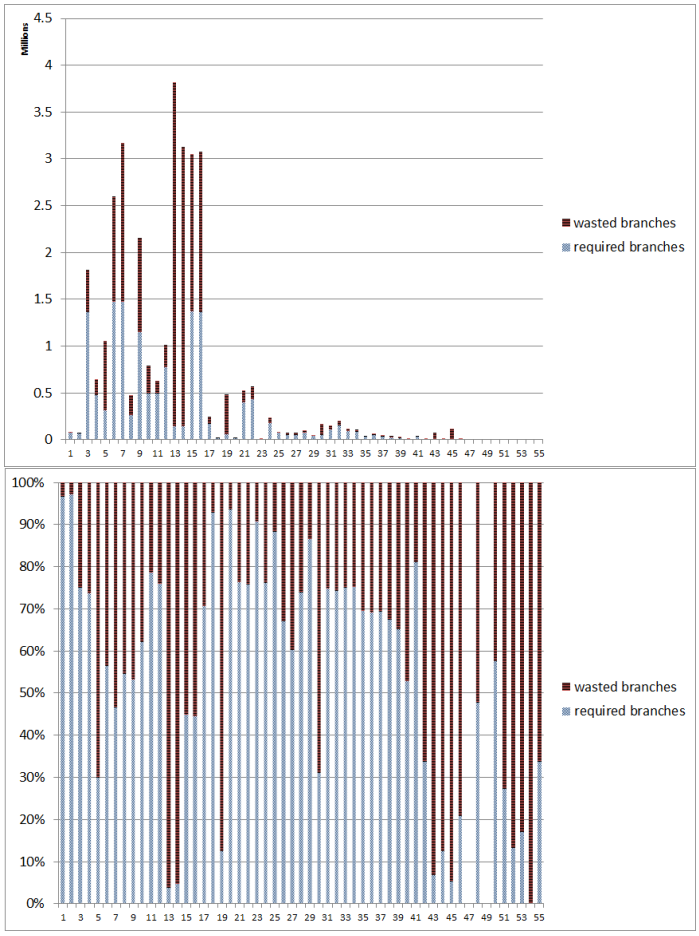

Figure 2: The numbers of branches that were required or wasted. Results include all 55 satisfiable instances from our benchmark set that completed within the 300-second solver timeout. The top graphs show total numbers on the vertical scale, and in the bottom graphs the vertical scale is normalized for each instance to show the proportions between the different classifications of events.

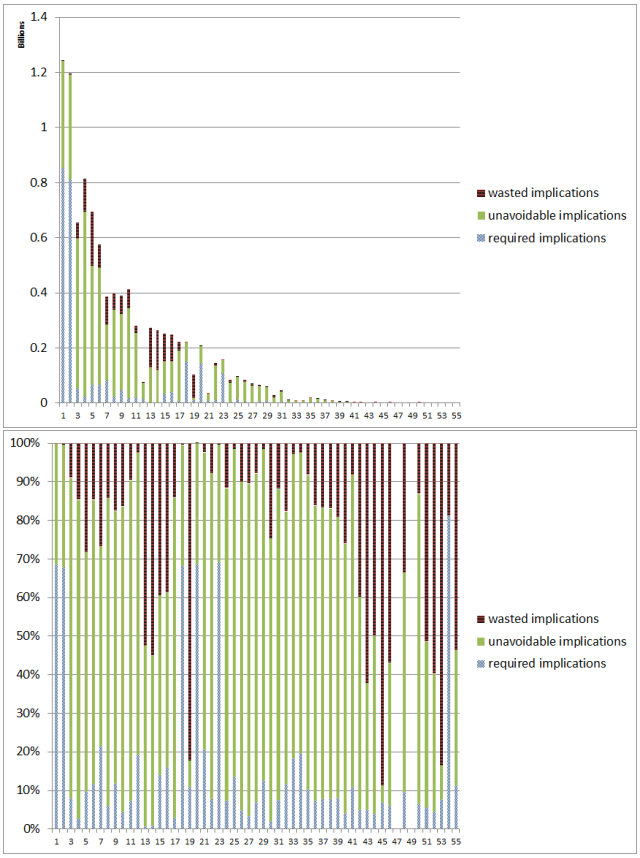

Figure 3: The numbers of implied assignments that were required, wasted, or unavoidable. Results include all 55 satisfiable instances from our benchmark set that completed within the 300-second solver timeout. The top graphs show total numbers on the vertical scale, and in the bottom graphs the vertical scale is normalized for each instance to show the proportions between the different classifications of events.

Summing over all satisfiable instances, 44% of branches are required and 56% are wasted. 27% of implications are required, 58% are unavoidable (but not required), and 15% are wasted.

5.2 Unsatisfiable instances

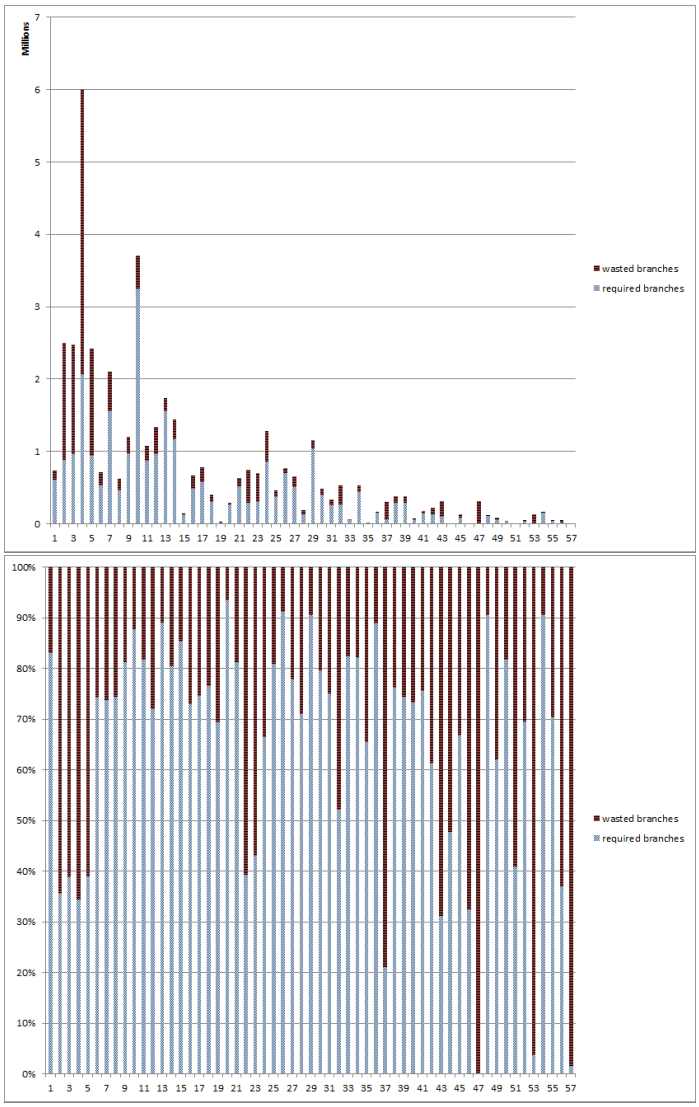

Figure 4: The numbers of branches that were required or wasted. Results include all 57 unsatisfiable instances from our benchmark set that completed within the 300-second solver timeout. The top graphs show total numbers on the vertical scale, and in the bottom graphs the vertical scale is normalized for each instance to show the proportions between the different classifications of events.

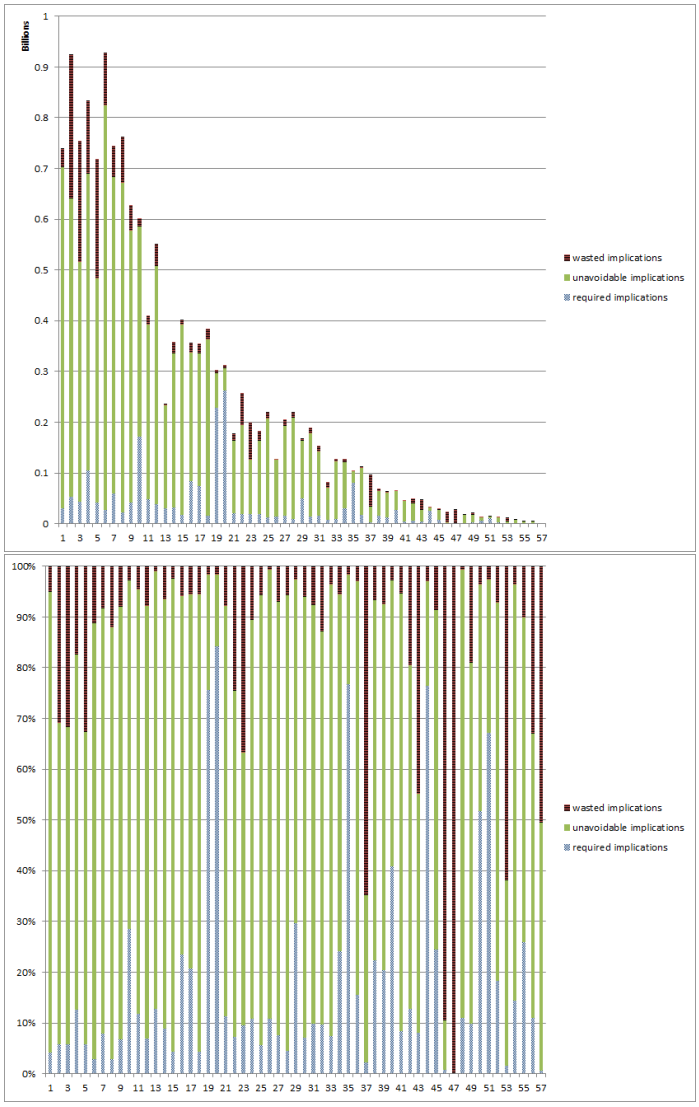

Figure 5: The numbers of implied assignments that were required, wasted, or unavoidable. Results include all 57 unsatisfiable instances from our benchmark set that completed within the 300-second solver timeout. The top graph shows total numbers on the vertical scale, and in the bottom graph the vertical scale is normalized for each instance.

Summing over all unsatisfiable instances, 63% of branches are required and 37% are wasted. 13% of implications are required, 74% are unavoidable, and 13% are wasted.

5.3 Relationship between events and runtime

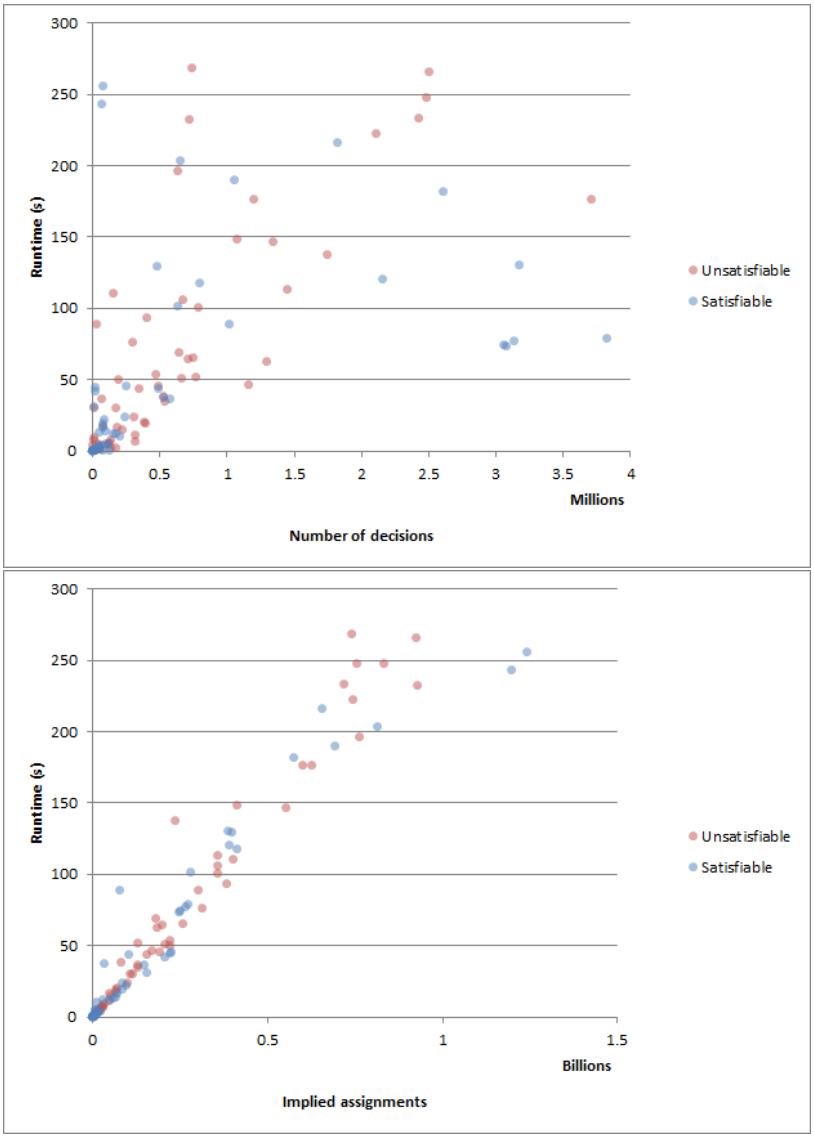

Figure 6: The runtime plotted against the number of implied assignments and the number of decisions for all 112 benchmark instances that completed within the 300-second solver timeout.

In Figure 6, we plot the runtime for each instance against the number of decisions for each instance. As has been observed in other research [2], the number of branches explored is a poor predictor of runtime. We also plot the runtime for each instance against the number of implied assignments for each instance. Here we see more of a linear trend. This is consistent with our understanding that most time is spent searching for implications in BCP. The slope of a trend line through the bottom graph would show about 1 second of runtime per 300 million implied assignments.

6 Discussion

6.1 Implications for future heuristic development

It has been observed that the number of branches or number of conflicts is a poor predictor of solver runtime [2], but as shown in Figure 6, the number of implied assignments found by the solver through BCP is a fairly good predictor of solver runtime. This suggests that we may use the proportions of wasted and required implied assignments as a proxy for the degree of improvement possible in solver runtimes if they had chosen all the required branches and not taken any of the wasted ones.

A major feature of our results is that most implied assignments are neither required nor wasted, suggesting there might be significant room for improvement that cannot be exploited simply by changing the decision heuristic to skip wasted branches. Exploiting this gap between the number of required events and the number of events that is merely required would require changes to the implementation of BCP. Solvers typically stop BCP and commence conflict analysis as soon as they encounter a conflict, so the order in which BCP searches for possible implied assignments and conflicts affects which implied assignments are found before any given conflict. There has been some research into using a priority queue to order BCP [19], and and our result gives weight to exploring these ideas further, including identifying how to calculate priority values that reflect how likely the resulting implications are to be productive.

That wasted implied assignments are minorities in the results, suggests that the room for runtime improvement through improved heuristics may be slim. However, the large proportion of wasted branches in some instances, suggests there are still categories of benchmarks where improvement might yet be made by attempting to modify the decision heuristic to avoid some of the wasted decisions while retaining many of the required events.

6.2 Limitations

However, we have some reason to be even more optimistic. If BCP or a decision heuristic is modified so that it tends to make fewer decisions that end up being unrequired and more required decisions, then, in general, this won't simply manage to avoid some of the wasted

work we identified. By picking different branches, the solver would proceed along an entirely different search path, and might still see orders of magnitude in speedup for a given instance, even if the analysis said the amount of unrequired events was modest. We cannot use the analysis we have done predict the what the difference in performance between the solver we have analyzed and any specific improved solver that must still solve the same instances without prior information about the instances.

Our analysis only asks, given that a solver took a certain search path and reached a certain result, whether there any diversions it took that could have been skipped. Fundamentally, our analysis does not explore any part of the space of possible DPLL search paths that the solver could have taken besides the one path it did take. Recall that if some implied assignment is part of the final satisfying assignment, then we defined all of the implied assignment and all of its dependencies as required, even if the solver took a very circuitous path to finding that implied assignment. We have no way of knowing whether the solver might have found the same assignment if it had explored different branches first. For that matter, the fastest way for the solver to solve a satisfiable instance is always to simply pick a satisfying assignment for its branches, with no implications or conflicts needed. We simply don't know how to identify the fastest path the solver might have taken to for any given unsatisfiable instance is. It appears computational intractable to explore alternatives paths systematically on the benchmark set we used.

7 Continued work

7.1 Data analysis approaches

The fundamental part of our approach is that it gives us a way of identifying dependencies among events, and the resulting dependency graph could well be analyzed in other ways besides looking at the set of dependencies of the final result and counting how many events are in and outside of that set. Time series analyses may be conducted to identify statistical differences between productive and unproductive periods. Such differences could suggest ways to adapt the decision heuristic dynamically to behave differently in different periods to improve overall performance.

It would be interesting to look at the distribution in in the number of branches or the number of conflicts that occur between events and their direct dependencies. We could look for correlations in this distance between dependencies and dependents with the productivity of periods within the solve run. In fact, we can look for correlations between any statistic we care for and the productivity of periods within the solver run, and this information might suggest continued design improvements in SAT solvers. Ultimately, our approach turns exploring the design space of SAT solvers into a rich data analysis problem.

7.2 Varying solver parameters

Modern CDCL solvers come with a variety of parameters and settings that affect the operation of decision heuristics, clause deletion heuristics, the timing of restarts, and other features. TTuning these parameters is usually done by attempting to minimize the single objective function of observed runtimes on a fixed benchmark set. If a change in the design results in worse runtimes, it is discarded, often without leaving behind any contributing in terms of understanding to what determines if a design change improves performance.

If the experiments described in this report are repeated with different design parameters, or with design features turned on and off, we could compare the datasets generated with different make use of this information to better understand the effects of varying design parameters. This might help us beyond identifying the optimal settings for these parameters, or verifying that a design feature does help performance. The data may give us additional

insight in why choosing settings different from the optimal ones increase runtimes. We might see how poor behavior emerges in some regimes due to the particular constraints in how current CDCL solvers operate. This insight might suggest entirely new design changes.

7.3 Implementing a decision oracle

As our analysis in some sense produces a chronological list of required branches, restarts, and deletions that could be taken by MiniSat to solve an instance, we could verify the correctness of this notion by modifying MiniSat to use that the required branches as as an decision oracle when run on the same instances a second time. If this works, we could demonstrate how much real execution time and memory usage may be improved by skipping wasted work.

Unfortunately, this would require significant modifications to the way BCP is done in MiniSat. After any branch, it is possible that there may be multiple conflicts that may be found, and MiniSat will generally stop performing BCP as soon as the first conflict is found. Additionally, even if the same conflict is found, this does not guarantee the analysis of that conflict will yield the same result, as there may be multiple redundant antecedents that would imply the same assignment, and MiniSat only records the first antecedent for an implication that it finds. Changing the stored antecedents would would change the conflict analysis and consequently the produced conflict clauses.

The order in which BCP is carried out and potential antecedents and conflicts is found is determined not only by the datastructure that stores assignments to propagate, which is a FIFO queue, but also the order of the clauses in the watch lists. To allow MiniSat to repeat a solver run would require somehow ordering the watch lists in a repeatable fashion, or else somehow making BCP results independent of the order of watch list entries.

In addition, MiniSat would need to be modified to repeat clause deletions and resets at the appropriate times as well. Doing this efficiently would require the analysis to produce well-structured information about how long each learned clause must be kept after being learned. The oracle solver may then use this information to tag each clause it learns with information on when it should be deleted, and either delete it at the right time, or lazily delete it whenever it is seen again during BCP and its tag indicates it has expired.

8 References

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing – STOC '71*, pp. 151–158, ACM, 1971.

[2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, (New York, NY, USA), pp. 530–535, ACM, 2001.

[3] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518, Springer Berlin Heidelberg, 2004.

[4] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing – SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings* (J. Marques-Silva and K. A. Sakallah, eds.), pp. 294–299, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[5] H. Katebi, K. A. Sakallah, and J. a. P. Marques-Silva, "Empirical study of the anatomy of modern SAT solvers," in *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing*, SAT'11, (Berlin, Heidelberg), pp. 343–356, Springer-Verlag, 2011.

[6] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, "Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers," in *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, pp. 225–241, 2015.

[7] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proceedings of*

the Conference on Design, Automation and Test in Europe - Volume 1, DATE '03, (Washington, DC, USA), pp. 10880–, IEEE Computer Society, 2003.

[8] M. J. H. Heule and A. Biere, "Proofs for satisfiability problems," in *All about Proofs, Proofs for All* (B. W. Paleo and D. Delahaye, eds.), pp. 59–73, College Publications, 2015.

[9] H. Amjad, "Compressing propositional refutations," in *Proceedings of the 6th International Workshop on Automated Verification of Critical Systems (AVoCS 2006)*, vol. 185 of *Electronic Notes in Theoretical Computer Science*, pp. 3–15, Elsevier Science Publishers B. V., July 2007.

[10] C. Sinz, "Compressing propositional proofs by common subproof extraction," in *Computer Aided Systems Theory – EUROCAST 2007* (R. Moreno DÃŋaz, F. Pichler, and A. Quesada Arencibia, eds.), vol. 4739 of *Lecture Notes in Computer Science*, pp. 547–555, Springer Berlin Heidelberg, 2007.

[11] S. Cotton, "Two techniques for minimizing resolution proofs," in *Theory and Applications of Satisfiability Testing – SAT 2010* (O. Strichman and S. Szeider, eds.), vol. 6175 of *Lecture Notes in Computer Science*, pp. 306–312, Springer Berlin Heidelberg, 2010.

[12] S. Rollini, R. Bruttomesso, and N. Sharygina, "An efficient and flexible approach to resolution proof reduction," in *Hardware and Software: Verification and Testing* (S. Barner, I. Harris, D. Kroening, and O. Raz, eds.), vol. 6504 of *Lecture Notes in Computer Science*, pp. 182–196, Springer Berlin Heidelberg, 2011.

[13] J. Boudou and B. W. Paleo, "Compression of propositional resolution proofs by lowering subproofs," in *Automated Reasoning with Analytic Tableaux and Related Methods* (D. Galmiche and D. Larchey-Wendling, eds.), vol. 8123 of *Lecture Notes in Computer Science*, pp. 59–73, Springer Berlin Heidelberg, 2013.

[14] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability* (H. v. M. Armin Biere, Marijn Heule and T. Walsch, eds.), pp. 127–149, IOS Press, 2008.

[15] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[16] N. Sörensson and A. Biere, "Minimizing learned clauses," in *Theory and Applications of Satisfiability Testing - SAT 2009* (O. Kullmann, ed.), vol. 5584 of *Lecture Notes in Computer Science*, pp. 237–243, Springer Berlin Heidelberg, 2009.

[17] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, (Piscataway, NJ, USA), pp. 279–285, IEEE Press, 2001.

[18] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, SAT'05, (Berlin, Heidelberg), pp. 61–75, Springer-Verlag, 2005.

[19] M. D. T. Lewis, T. Schubert, and B. W. Becker, "Speedup techniques utilized in modern SAT solvers," in *Theory and Applications of Satisfiability Testing: 8th International Conference, SAT 2005, St Andrews, UK, June 19-23, 2005. Proceedings* (F. Bacchus and T. Walsh, eds.), pp. 437–443, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

A Trace format

For every branch, we emit a line:

```
b [lit]
```

`[lit]` is a nonzero integer whose absolute value indicates a variable, and whose sign indicates the value the variable was assigned: positive indicates logical 1 and negative indicates logical 0. For example, the integer `7` indicates that the seventh variable was assigned the value 1, and the integer `-4` indicates that the fourth variable was assigned the value 0. this is identical to the format for literals in the standard DIMACS format.

Every clause containing multiple literals must be associated with a nonnegative integer ID. For every clause addition, we emit a line:

```
[id]: [clause]
```

`[id]` is the clause ID, which must be unique among all clauses stored by the solver at any time. In MiniSat, we use the existing clause references as ID's. `[clause]` consists of one or more literals in the format for `[lit]` specified for branches, followed by a `0` to indicate the end of the clause. The literals as well as the `0` are delimited by spaces. This is identical to the DIMACS format for representing clauses.

There is an exception in the case of clauses that contain only one literal, as they simply contain a single literal that is immediately permanently implied, so they do not need to be referenced later in the trace. Unit clauses are emitted as:

```
: [clause]
```

That is, they have no associated ID.

For each implication that does not immediately follow a conflict, we emit a line:

```
i [lit] [id]
```

where `[lit]` indicates the implied assignment in the same format as in the case of a branch, and `[id]` is the clause ID of the antecedent.

For each conflict discovered, we emit a line:

```
k [id] [level]
```

Where [id] is the ID of the falsified clause, and [level] indicates the decision level to backtrack to. This line is immediately followed by a line indicating the conflict clause, unless [level] is -1, indicating the instance is unsatisfiable. Since conflict clauses are always unit after non-chronological backtracking, we do not need to emit the first implication resulting from the conflict clause. We instead encode which literal is implied by placing it first when the conflict clause is emitted.

For each restart we emit a line:

```
r
```

For each clause deletion we emit a line:

```
d [id]
```

In MiniSat, garbage collection may change the number by which clauses are referenced. To account for this, during each garbage collection event, we emit a line for every clause that is still stored by the solver:

```
m [id] [id]
```

where the first [id] is the old ID and the second [id] is the new one.

Finally, at the end of the trace, to make it easier to identify the final result, we emit the line:

```
UNSAT
```

if the instance was unsatisfiable, or, if the instance was satisfiable we emit the line:

```
SAT [assignments]
```

where [assignments] gives the satisfying assignment as a series of literals in the format used for [lit] terminated with a 0, delimited by spaces.

Consider the following instance given in DIMACS format:

```
p cnf 2 4
1 2 0
1 -2 0
-1 2 0
-1 -2 0
```

Our instrumented version of MiniSat might emit the following trace:

```
0: 1 2 0
3: 1 -2 0
6: -1 2 0
9: -1 -2 0
b -1
i 2 0
k 3 0
: 1 0
i 2 6
k 9 -1
UNSAT
```

B Source code

B.1 Changes to MiniSat 2.2.0

```
diff --git a/minisat/core/Main.cc b/minisat/core/Main.cc
index 4388c3e..094ece9 100644
--- a/minisat/core/Main.cc
+++ b/minisat/core/Main.cc
@@ -130,9 +130,9 @@ int main(int argc, char** argv)
            printf("============================[ Problem Statistics
                    ]=============================\n");
            printf("|

                    |\n"); }

+       S.output = (argc >= 3) ? fopen(argv[2], "wb") : NULL;
        parse_DIMACS(in, S);
        gzclose(in);
-       FILE* res = (argc >= 3) ? fopen(argv[2], "wb") : NULL;

        if (S.verbosity > 0){
            printf("| Number of variables: %12d
                                                  |\n", S.nVars());
@@ -149,7 +149,7 @@ int main(int argc, char** argv)
        signal(SIGXCPU,SIGINT_interrupt);

        if (!S.simplify()){
-           if (res != NULL) fprintf(res, "UNSAT\n"), fclose(res);
+           if (S.output != NULL) fprintf(S.output, "UNSAT\n"),
         fclose(S.output);
            if (S.verbosity > 0){
                printf("================================================================================
                printf("Solved by unit propagation\n");
```

```
@@ -165,18 +165,18 @@ int main(int argc, char** argv)
            printStats(S);
            printf("\n"); }
        printf(ret == l_True ? "SATISFIABLE\n" : ret == l_False ?
                "UNSATISFIABLE\n" : "INDETERMINATE\n");
-       if (res != NULL){
+       if (S.output != NULL){
            if (ret == l_True){
-               fprintf(res, "SAT\n");
+               fprintf(S.output, "SAT\n");
                for (int i = 0; i < S.nVars(); i++)
                    if (S.model[i] != l_Undef)
-                       fprintf(res, "%s%s%d", (i==0)?"":" ",
         (S.model[i]==l_True)?"":"-", i+1);
-               fprintf(res, " 0\n");
+                       fprintf(S.output, "%s%s%d", (i==0)?"":" ",
         (S.model[i]==l_True)?"":"-", i+1);
+               fprintf(S.output, " 0\n");
            }else if (ret == l_False)
-               fprintf(res, "UNSAT\n");
+               fprintf(S.output, "UNSAT\n");
            else
-               fprintf(res, "INDET\n");
-           fclose(res);
+               fprintf(S.output, "INDET\n");
+           fclose(S.output);
        }

 #ifdef NDEBUG
diff --git a/minisat/core/Solver.cc b/minisat/core/Solver.cc
index 7da7f18..28ea789 100644
--- a/minisat/core/Solver.cc
+++ b/minisat/core/Solver.cc
```

```
@@ -35,7 +35,7 @@ static DoubleOption opt_var_decay (_cat, "var-decay",
         "The variable a
 static DoubleOption opt_clause_decay (_cat, "cla-decay", "The clause
          activity decay factor",      0.999,   DoubleRange(0, false, 1,
          false));
 static DoubleOption opt_random_var_freq (_cat, "rnd-freq", "The frequency
          with which the decision heuristic tries to choose a random
          variable", 0, DoubleRange(0, true, 1, true));
 static DoubleOption opt_random_seed  (_cat, "rnd-seed", "Used by the
          random variable selection",  91648253, DoubleRange(0, false,
          HUGE_VAL, false));
-static IntOption  opt_ccmin_mode     (_cat, "ccmin-mode", "Controls
         conflict clause minimization (0=none, 1=basic, 2=deep)", 2,
         IntRange(0, 2));
+static IntOption  opt_ccmin_mode     (_cat, "ccmin-mode", "Controls
         conflict clause minimization (0=none, 1=basic, 2=deep)", 0,
         IntRange(0, 2));
 static IntOption  opt_phase_saving   (_cat, "phase-saving", "Controls the
          level of phase saving (0=none, 1=limited, 2=full)", 2,
          IntRange(0, 2));
 static BoolOption opt_rnd_init_act   (_cat, "rnd-init", "Randomize the
          initial activity", false);
 static BoolOption opt_luby_restart   (_cat, "luby",     "Use the Luby
          restart sequence", true);
@@ -144,9 +144,14 @@ bool Solver::addClause_(vec<Lit>& ps)
            ps[j++] = p = ps[i];
     ps.shrink(i - j);

-    if (ps.size() == 0)
+    if (ps.size() == 0){
+        if (output != NULL)
+            fprintf(output, ": 0\n");
         return ok = false;
```

```
-    else if (ps.size() == 1){
+    }else if (ps.size() == 1){
+        if (output != NULL)
+            fprintf(output, ": %i 0\n",
+                    (var(ps[0]) + 1) * (-2 * sign(ps[0]) + 1) );
         uncheckedEnqueue(ps[0]);
         return ok = (propagate() == CRef_Undef);
     }else{
@@ -162,6 +167,12 @@ bool Solver::addClause_(vec<Lit>& ps)
 void Solver::attachClause(CRef cr) {
     const Clause& c = ca[cr];
     assert(c.size() > 1);
+    if (output != NULL && cr != CRef_Undef) {
+      fprintf(output, "%i: ", cr);
+      for (int i = 0; i < c.size(); i++)
+        fprintf(output, "%i ", (var(c[i]) + 1) * (-2 * sign(c[i]) + 1));
+      fprintf(output, "0\n");
+    }
     watches[~c[0]].push(Watcher(cr, c[1]));
     watches[~c[1]].push(Watcher(cr, c[0]));
     if (c.learnt()) learnts_literals += c.size();
@@ -187,6 +198,10 @@ void Solver::detachClause(CRef cr, bool strict) {

 void Solver::removeClause(CRef cr) {
     Clause& c = ca[cr];
+
+    if (output != NULL)
+      fprintf(output, "d %i\n", cr);
+
     detachClause(cr);
     // Don't leave pointers to free'd memory!
     if (locked(c)) vardata[var(c[0])].reason = CRef_Undef;
@@ -240,7 +255,11 @@ Lit Solver::pickBranchLit()
```

```
106         }else
107             next = order_heap.removeMin();
108 
109 -   return next == var_Undef ? lit_Undef : mkLit(next, rnd_pol ?
        drand(random_seed) < 0.5 : polarity[next]);
110 +   bool pol = rnd_pol ? drand(random_seed) < 0.5 : polarity[next];
111 +   if (output != NULL && next != var_Undef){
112 +     fprintf(output, "b %i\n", (next + 1) * (-2 * pol + 1));
113 +   }
114 +   return next == var_Undef ? lit_Undef : mkLit(next, pol);
115  }
116 
117 
118 @@ -494,8 +513,13 @@ CRef Solver::propagate()
119                 // Copy the remaining watches:
120                 while (i < end)
121                     *j++ = *i++;
122 -           }else
123 +           }else{
124 +               if (output != NULL)
125 +                   fprintf(output, "i %i %i\n",
126 +                           (var(first) + 1) * (-2 * sign(first) + 1),
127 +                           cr);
128                 uncheckedEnqueue(first, cr);
129 +           }
130 
131         NextClause:;
132         }
133 @@ -623,15 +647,28 @@ lbool Solver::search(int nof_conflicts)
134         CRef confl = propagate();
135         if (confl != CRef_Undef){
136             // CONFLICT
137 +           if (output != NULL)
```

```
138 +               fprintf(output, "k %i ", confl);
139             conflicts++; conflictC++;
140 -           if (decisionLevel() == 0) return l_False;
141 +           if (decisionLevel() == 0){
142 +               if (output != NULL)
143 +                   fprintf(output, "-1\n");
144 +               return l_False;
145 +           }
146
147             learnt_clause.clear();
148             analyze(confl, learnt_clause, backtrack_level);
149 +           if (output != NULL)
150 +               fprintf(output, "%i\n", backtrack_level);
151             cancelUntil(backtrack_level);
152
153             if (learnt_clause.size() == 1){
154 -               uncheckedEnqueue(learnt_clause[0]);
155 +               Lit learnt_lit = learnt_clause[0];
156 +               uncheckedEnqueue(learnt_lit);
157 +               if (output != NULL) {
158 +                 fprintf(output, ": %i 0\n" ,
159 +                         (var(learnt_lit) + 1) * (-2 * sign(learnt_lit) + 1)
          );
160 +               }
161             }else{
162                 CRef cr = ca.alloc(learnt_clause, true);
163                 learnts.push(cr);
164 @@ -660,6 +697,8 @@ lbool Solver::search(int nof_conflicts)
165             if (nof_conflicts >= 0 && conflictC >= nof_conflicts ||
                        !withinBudget()){
166                 // Reached bound on number of conflicts:
167                 progress_estimate = progressEstimate();
168 +               if (output != NULL)
```

```
+                    fprintf(output, "r\n");
                 cancelUntil(0);
                 return l_Undef; }

@@ -899,13 +938,21 @@ void Solver::relocAll(ClauseAllocator& to)

     // All learnt:
     //
-    for (int i = 0; i < learnts.size(); i++)
+    for (int i = 0; i < learnts.size(); i++){
+        CRef old_cr = learnts[i];
         ca.reloc(learnts[i], to);
+        if (output != NULL)
+            fprintf(output, "m %i %i\n", old_cr, learnts[i]);
+    }

     // All original:
     //
-    for (int i = 0; i < clauses.size(); i++)
+    for (int i = 0; i < clauses.size(); i++){
+        CRef old_cr = clauses[i];
         ca.reloc(clauses[i], to);
+        if (output != NULL)
+            fprintf(output, "m %i %i\n", old_cr, clauses[i]);
+    }
 }


diff --git a/minisat/core/Solver.h b/minisat/core/Solver.h
index 90119e5..81df36d 100644
--- a/minisat/core/Solver.h
+++ b/minisat/core/Solver.h
@@ -105,6 +105,8 @@ public:
```

```
202     void   checkGarbage(double gf);
203     void   checkGarbage();
204
205 +   FILE*            output;
206 +
207     // Extra results: (read-only member variable)
208     //
209     vec<lbool> model;         // If problem is satisfiable, this vector
                contains the model (if any).
```

B.2 Trace analyzer

```
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <vector>

#include "core/SolverTypes.h"

using namespace std;
using namespace Minisat;


#ifndef NDEBUG
//#define DEBUG_EXTRA
#endif

enum skip {
  UNKNOWN = 0,
  SKIP,
  NO_SKIP,
};


/*
 *  Abstract classes
 */

class Event {
  protected:
    bool required;
  public:
    Event(bool init_required) : required(init_required) { }
```

```
32    bool is_required() { return required; }
33    virtual bool is_skippable() { return false; }
34    void set_required() { if (!required) { required = true;
              handle_required(); } }
35    virtual void print(ostream& out) = 0;
36  private:
37    virtual void handle_required() { }
38 };
39
40 class Assignment : public Event {
41  protected:
42    Lit lit;
43  public:
44    Assignment(Lit l) : Event(false), lit(l) { }
45    Assignment(istream& in) ;
46    Lit get_lit() { return lit; }
47    unsigned get_var() { return var(lit); }
48    bool get_sign() { return sign(lit); }
49 };
50
51 class ClauseAddition : public Event {
52  protected:
53    vector<Lit> lits;
54    int deletion_time;
55  public:
56    ClauseAddition() : Event(false), deletion_time(-1) { } // Only for when
              empty clause is learned
57    ClauseAddition(istream& in) ;
58    vector<Lit>& get_lits() { return lits; }
59    void set_deleted() ;
60    virtual void print(ostream& out) ; // override
61 };
62
```

```
63
64 /*
65  * Concrete classes
66  */
67
68 class Branch : public Assignment {
69   public:
70     Branch(istream& in) : Assignment(in) { }
71     bool is_skippable() { return !required; } // override
72     void print(ostream& out) ; // override
73 };
74
75 class Implication : public Assignment {
76   protected:
77     ClauseAddition* antecedent; // 0 if antecedent is unit clause
78     Assignment** required_assignments; // null-terminated array of
                dependencies
79     enum skip skippable;
80   public:
81     Implication(Lit lit, ClauseAddition* ante) ;
82     bool is_skippable() ; // override
83     void print(ostream& out) ; // override
84   private:
85     void handle_required() ; // override
86 };
87
88 class OriginalClause : public ClauseAddition {
89   public:
90     OriginalClause(istream& in) : ClauseAddition(in) { }
91 };
92
93 class LearnedClause : public ClauseAddition {
94   protected:
```

```
95    ClauseAddition& antecedent;
96    Assignment** required_assignments; // null-terminated array of
              dependencies
97    enum skip skippable;
98  public:
99    LearnedClause(istream& in, ClauseAddition& conflicting_clause) ;
100   LearnedClause(ClauseAddition& conflicting_clause) ; // Only for when
              empty clause is learned
101   bool is_skippable() ; // override
102 private:
103   void handle_required() ; // override
104 };
105
106 class Reset : public Event {
107 public:
108   Reset() : Event(true) { }
109   void print(ostream& out) { out << "r\n"; } // override
110 };
111
112 class Deletion : public Event {
113 protected:
114   ClauseAddition* clause;
115 public:
116   Deletion(istream& in) ;
117   void print(ostream& out) { out << "d "; clause->print(out); } //
              override
118 };
119
120 class AssignmentComplete : public Event {
121 protected:
122   Assignment** final_assignments; // null-terminated array
123 public:
124   AssignmentComplete() ;
```

```
125     void print(ostream& out) ; // override
126     void handle_required() ; // override
127 };
128
129
130 /*
131  * Global variables
132  */
133
134 vector<Event*> event_list; // list of events in order from trace
135 vector<ClauseAddition*> clauses; // mapping of clause ID's to clause
            addition events
136 vector<Assignment*> assignments; // mapping of variables to assignments
137 vector<Assignment*> trail; // Stack of currently relevant assignments
138 vector<unsigned> trail_lim; // Stack of trail indices for branches
139 unsigned deletion_count;
140
141
142 /*
143  *  Functions
144  */
145
146 void print_lit(ostream& out, Lit lit) { out << (sign(lit) ? "-" : "") <<
            var(lit); }
147
148 Lit read_lit(istream& in) {
149   int parsed_lit = 0;
150   in >> parsed_lit;
151   assert (parsed_lit != 0);
152   return mkLit(abs(parsed_lit), parsed_lit < 0);
153 }
154
155
```

```
156 ClauseAddition::ClauseAddition(istream& in) : Event(false),
          deletion_time(-1) {
157   while (true) {
158     int parsed_lit = 0;
159     in >> parsed_lit;
160     if (parsed_lit == 0) break;
161     lits.push_back(mkLit(abs(parsed_lit), parsed_lit < 0));
162   }
163 }
164
165 void ClauseAddition::print(ostream& out) {
166   for (vector<Lit>::iterator it = lits.begin(); it != lits.end(); ++it) {
167     print_lit(out, *it);
168     out << " ";
169   }
170   out << "0 ";
171   if (deletion_time >= 0)
172     out << deletion_time;
173   out << endl;
174 }
175
176 void ClauseAddition::set_deleted() {
177   deletion_time = deletion_count++;
178 }
179
180 Deletion::Deletion(istream& in) : Event(true) {
181   int clause_id = -1;
182   in >> clause_id;
183   assert(clause_id >= 0 && (unsigned)clause_id < clauses.size());
184   clause = clauses[clause_id];
185   assert(clause != 0);
186   clauses[clause_id] = 0;
187
```

```
188   clause->set_deleted();
189 }
190
191
192 Assignment::Assignment(istream& in) : Event(false) {
193   lit = read_lit(in);
194 }
195
196
197 void Branch::print(ostream& out) {
198   out << "b ";
199   print_lit(out, lit);
200   out << "\n";
201 }
202
203
204 Implication::Implication(Lit lit, ClauseAddition* ante) : Assignment(lit),
          antecedent(ante), skippable(UNKNOWN) {
205   if (antecedent == 0) {
206     required_assignments = 0;
207   } else {
208 #ifdef DEBUG_EXTRA
209     cout << "Initializing implicaton of ";
210     print_lit(cout, lit);
211     cout << ". Antecedent is ";
212     antecedent->print(cout);
213 #endif
214     vector<Lit>& lits = antecedent->get_lits();
215     required_assignments = new Assignment*[lits.size()];
216     unsigned i = 0;
217     for (vector<Lit>::iterator it = lits.begin(); it != lits.end(); ++it) {
218       unsigned v = var(*it);
219       if ((int)v != var(lit)) {
```

```
220         assert(v < assignments.size());
221         Assignment* assignment = assignments[v];
222         assert(assignment != 0);
223         assert(assignment->get_sign() != sign(*it));
224         required_assignments[i++] = assignment;
225       }
226     }
227     assert(i == lits.size() - 1);
228     required_assignments[i] = 0;
229   }
230 }
231
232 void Implication::handle_required() {
233 #ifdef DEBUG_EXTRA
234   cout << "Implication required: ";
235   print(cout);
236 #endif
237   required = true;
238   if (antecedent != 0) {
239     antecedent->set_required();
240     Assignment** required_assignment = required_assignments;
241     while (*required_assignment != 0) {
242       (*required_assignment)->set_required();
243       ++required_assignment;
244     }
245   }
246 }
247
248 bool Implication::is_skippable() {
249   if (skippable == SKIP)
250     return true;
251   else if (skippable == NO_SKIP)
252     return false;
```

```
253
254   if (required)
255     skippable = NO_SKIP;
256   else if (antecedent == 0)
257     skippable = SKIP;
258   else if (antecedent->is_skippable())
259     skippable = SKIP;
260   else {
261     skippable = NO_SKIP;
262     Assignment** required_assignment = required_assignments;
263     while (*required_assignment != 0) {
264       if ((*required_assignment)->is_skippable()) {
265         skippable = SKIP;
266         break;
267       }
268       ++required_assignment;
269     }
270   }
271   return (skippable == SKIP);
272 }
273
274 void Implication::print(ostream& out) {
275   out << "i ";
276   print_lit(out, lit);
277   out << "\n";
278 }
279
280
281 LearnedClause::LearnedClause(istream& in, ClauseAddition&
          conflicting_clause) : ClauseAddition(in),
          antecedent(conflicting_clause), skippable(UNKNOWN) {
282   vector<Lit>& lits = antecedent.get_lits();
283   required_assignments = new Assignment*[lits.size() + 1];
```

```
284   unsigned i = 0;
285   for (vector<Lit>::iterator it = lits.begin(); it != lits.end(); ++it) {
286     Assignment* assignment = assignments[var(*it)];
287     assert(assignment->get_sign() != sign(*it));
288     required_assignments[i++] = assignment;
289   }
290   assert(i == lits.size());
291   required_assignments[i] = 0;
292 }
293 
294 LearnedClause::LearnedClause(ClauseAddition& conflicting_clause) :
          antecedent(conflicting_clause) {
295   vector<Lit>& lits = antecedent.get_lits();
296   required_assignments = new Assignment*[lits.size() + 1];
297   unsigned i = 0;
298   for (vector<Lit>::iterator it = lits.begin(); it != lits.end(); ++it) {
299     Assignment* assignment = assignments[var(*it)];
300     assert(assignment->get_sign() != sign(*it));
301     required_assignments[i++] = assignment;
302   }
303   assert(i == lits.size());
304   required_assignments[i] = 0;
305 }
306 
307 void LearnedClause::handle_required() {
308 #ifdef DEBUG_EXTRA
309   cout << "LearnedClause required: ";
310   print(cout);
311 #endif
312   required = true;
313   antecedent.set_required();
314   Assignment** required_assignment = required_assignments;
315   while (*required_assignment != 0) {
```

```
316     (*required_assignment)->set_required();
317     ++required_assignment;
318   }
319 }
320
321 bool LearnedClause::is_skippable() {
322   if (skippable == SKIP)
323     return true;
324   else if (skippable == NO_SKIP)
325     return false;
326
327   if (required)
328     skippable = NO_SKIP;
329   else if (antecedent.is_skippable())
330     skippable = SKIP;
331   else {
332     skippable = NO_SKIP;
333     Assignment** required_assignment = required_assignments;
334     while (*required_assignment != 0) {
335       if ((*required_assignment)->is_skippable()) {
336         skippable = SKIP;
337         break;
338       }
339       ++required_assignment;
340     }
341   }
342   return (skippable == SKIP);
343 }
344
345
346 AssignmentComplete::AssignmentComplete() : Event(false) {
347   unsigned i, highest_var = assignments.size() - 1;
348   final_assignments = new Assignment*[highest_var + 1];
```

```
349   for (i = 1; i <= highest_var; i++) {
350     assert(assignments[i] != 0);
351     final_assignments[i-1] = assignments[i];
352   }
353   final_assignments[highest_var] = 0;
354 }
355
356 void AssignmentComplete::handle_required() {
357   for (Assignment** required_assignment = final_assignments;
358        *required_assignment != 0;
359        required_assignment++) {
360     (*required_assignment)->set_required();
361   }
362 }
363
364 void AssignmentComplete::print(ostream& out) {
365   out << "SAT" << endl;
366   for (Assignment** a = final_assignments; *a != 0; a++) {
367     (*a)->print(out);
368   }
369   out << "SAT" << endl;
370 }
371
372
373 void add_assignment(Assignment& assignment) {
374   unsigned v = assignment.get_var();
375   if (v < assignments.size()) {
376     assert(assignments[v] == 0);
377   } else {
378     assignments.resize(v + 1);
379   }
380   assignments[v] = &assignment;
381   trail.push_back(&assignment);
```

```
382   event_list.push_back(&assignment);
383 #ifdef DEBUG_EXTRA
384   cout << "NEW assignment: ";
385   print_lit(cout, assignment.get_lit());
386   cout << endl;
387 #endif
388 }
389
390 void add_clause(ClauseAddition& clause, int id = -1) {
391   if (id >= 0) {
392     if ((unsigned)id < clauses.size()) {
393       assert(clauses[id] == 0);
394     } else {
395       clauses.resize(id + 1);
396     }
397     clauses[id] = &clause;
398   }
399   event_list.push_back(&clause);
400 #ifdef DEBUG_EXTRA
401   cout << "new clause: ";
402   clause.print(cout);
403   cout << flush;
404 #endif
405 }
406
407 unsigned decision_level() {
408   return trail_lim.size();
409 }
410
411 void new_decision_level() {
412   trail_lim.push_back(trail.size());
413 }
414
```

```
415 void backtrack_to(unsigned level) {
416   assert(level <= decision_level());
417   if (level >= decision_level())
418     return;
419 
420   unsigned i;
421   for (i = trail_lim[level]; i < trail.size(); ++i) {
422     assignments[trail[i]->get_var()] = 0;
423   }
424   trail.resize(trail_lim[level]);
425   trail_lim.resize(level);
426 }
427 
428 LearnedClause* parse_conflict(istream& in) {
429   int conflicting_id, backtrack_level;
430   in >> conflicting_id >> backtrack_level;
431 
432   ClauseAddition* conflicting_clause = clauses[conflicting_id];
433   assert(conflicting_clause != 0);
434 
435 #ifdef DEBUG_EXTRA
436   cout << "confict: ";
437   conflicting_clause->print(cout);
438   cout << "backtracking to level " << backtrack_level << endl;
439 #endif
440 
441   if (backtrack_level == -1) {
442     LearnedClause* clause = new LearnedClause(*conflicting_clause);
443     add_clause(*clause);
444     return clause;
445   }
446 
447   int id = -1;
```

```
448   in >> ws;
449   if ('0' <= in.peek() && in.peek() <= '9') {
450     in >> id;
451   }
452   in.ignore(1, ':');
453   LearnedClause* clause = new LearnedClause(in, *conflicting_clause);
454   add_clause(*clause, id);
455
456   backtrack_to(backtrack_level);
457
458   // Learned clause is asserting, first literal is implied
459   Implication* implication = new Implication(clause->get_lits()[0],
              clause);
460   add_assignment(*implication);
461
462   return 0;
463 }
464
465 void parse_implication(istream& in) {
466   Lit lit = read_lit(in);
467   int antecedent_id;
468   in >> antecedent_id;
469   assert(antecedent_id >= 0 && (unsigned)antecedent_id < clauses.size());
470   ClauseAddition* antecedent = clauses[antecedent_id];
471   assert(antecedent != 0);
472   Implication* implication = new Implication(lit, antecedent);
473   add_assignment(*implication);
474 }
475
476 void parse_clause_movement(istream& in) {
477   vector<ClauseAddition*> new_clauses;
478   new_clauses.reserve(clauses.size() / 4u);
479   while (true) {
```

```
480     unsigned old_id, new_id;
481     in >> old_id >> new_id;
482     assert (old_id < clauses.size());
483     if (new_id >= new_clauses.size())
484       new_clauses.resize(new_id + 1);
485     assert(clauses[old_id] != 0);
486     new_clauses[new_id] = clauses[old_id];
487     in >> ws;
488     if (in.peek() != 'm')
489       break;
490     in.ignore(1, 'm');
491   }
492   clauses.swap(new_clauses);
493 }
494 
495 Event& parse(istream& in) {
496 
497   Event* final_event = 0;
498   while (final_event == 0) {
499     in >> ws;
500     if ('0' <= in.peek() && in.peek() <= '9') {
501       unsigned clause_id;
502       in >> clause_id;
503       in.ignore(1, ':');
504       OriginalClause* clause = new OriginalClause(in);
505       add_clause(*clause, clause_id);
506       assert(clause->get_lits().size() > 1u);
507     } else {
508       char c = '\a';
509       in >> c;
510       switch (c) {
511       case ':': {
512         OriginalClause* clause = new OriginalClause(in);
```

```
513         add_clause(*clause);
514         unsigned clause_size = clause->get_lits().size();
515         if (clause_size == 0u) {
516           final_event = clause;
517         } else {
518           assert(clause_size == 1u);
519           add_assignment(*new Implication(clause->get_lits()[0], clause));
520         }
521       } break;
522       case 'i':
523         parse_implication(in);
524         break;
525       case 'b':
526         new_decision_level();
527         add_assignment(*new Branch(in));
528         break;
529       case 'k':
530         final_event = parse_conflict(in);
531         break;
532       case 'r':
533         backtrack_to(0);
534         break;
535       case 'd':
536         event_list.push_back(new Deletion(in));
537         break;
538       case 'm':
539         parse_clause_movement(in);
540         break;
541       case '\a':
542         cerr << "End of file without finding empty clause?" << endl;
543         exit(EXIT_FAILURE);
544         break;
545       case 'S':
```

```
546         final_event = new AssignmentComplete();
547         event_list.push_back(final_event);
548         break;
549       case 'U':
550         cerr << "Early UNSAT?! Aborting..." << endl;
551         exit(EXIT_FAILURE);
552         break;
553       default:
554         cerr << "Parse error! " << c << endl;
555         exit(EXIT_FAILURE);
556         break;
557       }
558     }
559   }
560   return *final_event;
561 }
562
563 int main(int argc, char* argv[]) {
564   if (argc <= 2) {
565     cerr << "usage: " << argv[0] << " INPUT_FILE OUTPUT_FILE" << endl;
566     exit(EXIT_FAILURE);
567   }
568
569   cout << "Parsing..." << endl;
570
571   ifstream in(argv[1]);
572   Event &final_event = parse(in);
573
574   cout << "Traversing dependencies..." << endl;
575
576   final_event.set_required();
577
578   cout << "Writing out analysis..." << endl;
```

```
579
580   ofstream out(argv[2]);
581   for (vector<Event*>::iterator it = event_list.begin(); it !=
                event_list.end(); ++it) {
582     if ((*it)->is_skippable()) {
583       out << "~ ";
584     }
585     else if ((*it)->is_required()) {
586       out << "! ";
587     }
588     (*it)->print(out);
589   }
590
591   cout << "Done." << endl;
592 }
```