An On-Line Satisfiability Algorithm for Conjunctive Normal Form Expressions with Two Literals

John F. Kolen
Institute of Human and Machine Cognition
University of West Florida
Pensacola, Florida 32501
jkolen@ai.uwf.edu

Abstract

This paper describes two algorithms for determining the satisfiability of Boolean conjunctive normal form expressions limited to two literals per clause (2-SAT) extending the classic effort of Aspvall, Plass, and Tarjan. The first algorithm differs from the original in that satisfiability is determined upon the presentation of each clause rather than the entire clause set. This on-line algorithm experimentally exhibits average run-time linear to the number of variables. This performance is achieved by performing a single depth first search of one of the incoming literals. Additional search is avoided by excluding clauses containing pure variables or variables whose truth value has been explicitly provided or can be inferred. An off-line algorithm is also described that incorporates these strategies.

Introduction

A conjunctive normal form Boolean expression (CNF) is a conjunction of disjunctive clauses. Determining the existance of an assignment of truth values to the variables of the CNF expression that makes it evaluate as true is the satisfiability problem. When the clause size is greater than two, the problem is NP-Complete (Cook 1971). A linear algorithm for 2-SAT was originally developed by Aspvall, Plass, and Tarjan (Aspvall, Plass, & Tarjan 1979) (APT algorithm). Consider a directed graph where the vertices are the set of literals from a 2-CNF formula. One could construct a digraph on these vertices from the formula by interpreting each clause $a \vee b$ as a pair of directed edges $\overline{a} \rightarrow b$ and $\overline{b} \rightarrow a$ on the graph. They proved that a formula is satisfiable if, and only if, no pair of literals, x and $\overline{x}$, appear in the same strongly connected component. Since a linear time algorithm for determining the strongly connected components of a digraph existed (Tarjan 1972; Aho, Hopcroft, & Ullman 1974), the 2-SAT question could be answered in linear time as well.

The APT algorithm requires the entire expression before it can begin processing–an off-line algorithm. An on-line algorithm, on the other hand, is one receives a sequence of inputs and performs some computation on the sequence seen so far. For instance, an on-line convex hull algorithm incrementally maintains the convex hull of a sequence of coplanar points (Preparata 1979).

In this paper, I first describe an on-line algorithm for 2-SAT. At each insertion of a clause, the algorithm conditionally performs a depth first search of subset of the dual implication graph and checks affected components for inclusion of literal pairs (x and $\overline{x}$). A large performance savings is made by not searching on pure variables, by making opportunistic variable assignments, especially during depth first search of the graph. These techniques are used to construct an off-line algorithm, as well. The performance of the algorithms are demonstrated experimentally.

Preliminaries

During the course of this paper, the following terms are used to describe satisfiability problems and their graph theoretic equivalents. Let $X = x_1, ..., x_n$ be a set of Boolean variables. A literal is either a variable or it's negation, hence $L = \{x_1, ..., x_n\} \cup \{\overline{x_1}, ..., \overline{x_n}\}$ is the set of literals. A clause, c, is a multiset representing the disjunction of k members of L. For this paper, we are only interested in the case where $k = 2$. This restriction implies that clauses will exhibit one of three forms ($a, b \in L$): i) unit clauses $a \vee a$, ii) tautological clauses, $a \vee \overline{a}$, iii) and regular clauses, $a \vee b$. A clause set is the multiset C drawn from L^k. An instance of a k-SAT problem is $F = (X, C)$. A literal, a, is pure, if it's negation does not appear in any clause in C.

The 2-SAT decision problem is intimately related to decisions problems on graphs (Aspvall, Plass, & Tarjan 1979). All graphs are assumed to be directed. Let $G = (V, E)$ designate a graph with V vertices and E edges. The graph dual, or implication graph, G, of a 2-SAT problem instance (X, C) directly represents the implication relationship between literals of X induced by the disjunctive clauses of C. The vertices, V, of the graph are the literals, L. An edge exists in the graph if and only if the implication relationship holds between the two vertices. Hence, $E = \{(a, b) \mid (\overline{a} \vee b) \in C\}$. Note that both $(a, b) \in E$ if and only if $(\overline{b}, \overline{a}) \in E$. Literal and vertex will be used interchangeably through out the paper.

A vertex b is reachable from vertex a if there exists a set $(x_i, x_{i+1}) \subset V$ of vertices such that $a \rightarrow x_1$, $x_1 \rightarrow x_2$, $\ldots$, $x_i \rightarrow b$. Reachability of two vertices is represented as

```
proc 2-SAT_APT($F$)
  Construct the dual graph G=(V,E) of F
  foreach $v \in E$ do
    if not($w.Mark$) then SearchAPT($v$); fi
    return not(SearchFailed()) od.
proc SearchAPT($v$)
  Push($v$);
  $v.Lowlink := v.Dfnum := count$;
  $count := count + 1$;
  $v.Mark :=$ true;
  foreach $w$ where $(v, w) \in E$ do
    if not($w.Mark$)
      then SearchAPT($w$);
          $v.Lowlink := min(v.Lowlink, w.Lowlink)$
    elsif $w.Dfnum < v.Dfnum \wedge$ OnStack($w$)
        then $v.Lowlink :=$
            $min(v.Lowlink, w.Dfnum)$
    fi od
  if $v.Lowlink = v.Dfnum$
    then repeat
        $u :=$ Pop();
        if $\overline{u}$ is on the stack after $v$
          then Fail(); fi
        until $u = v$; fi.
```

Figure 1: The 2-SAT algorithm of Aspvall, Plass, and Tarjan (Aspvall, Plass, & Tarjan 1979) embedded into a strongly connected connected components algorithm (Aho, Hopcroft, & Ullman 1974).

$a \leadsto b$. The set $S \subset V$ such that for all $a, b \in S$, $a \leadsto b$ and $b \leadsto a$ is a strongly connected component (SCC). A maximal strongly connected component is known as a strong component. Components throughout this paper will refer to strong components. The set of all components partitions the graph vertices. A contradictory component contains vertex, a, and it's negation, $\overline{a}$.

When the graph is an implication graph, the partition gives rise to an equivalence relation for literal truth values. This view justifies (Aspvall, Plass, & Tarjan 1979) observation that if an some strong component of implication graph contains a literal and it's negation, the dual 2-SAT formula is unsatisfiable. Figure 1 illustrates the APT algorithm along with Tarjan's connected components algorithm from (Aho, Hopcroft, & Ullman 1974). This algorithm was much more amenable to the optimization than Tarjan's version (Tarjan 1972). Since $SearchAPT(v)$ is called once for each literal in the dual graph and all edges must be examined, the run time is $\Theta(\mid V \mid + \mid E \mid)$ when an edge list is used.

Before discussing the new algorithms, four theorems will be needed. When appropriate, let F be a 2-SAT instance and G its graph dual. For sake of brevity, the theorems are stated without proof.

Theorem 1 *A pure literal from F and it's negation are singleton strong components in G.*

Theorem 2 *Let set P be the component partition of G. Consider the new partition, P', of G after adding edge (a, b) to G. Either $P' = P$ or there exist two components $S_1, S_2 \in P$ such that $S_1, S_2 \notin P'$ but $S_1 \cup S_2 \subseteq S$, and $S \in P'$.*

Theorem 3 *Let $a \leadsto b$ and $a \leadsto \overline{b}$. There exists a path from a to $\overline{a}$ through b (and $\overline{b}$) with length $\mid a \leadsto b \mid + \mid a \leadsto b \mid$.*

Theorem 4 *S is a contradictory component if and only if $\forall b \in S$, $\overline{b} \in S$.*

Algorithms

While the APT algorithm performs well–linear run time with respect to the number of variables and edges–it requires the entire formula to begin it's processing. There are situations, however, that this may not be feasible or, at least, inconvenient. For instance, the 2-SAT instance could be generated by some other procedure as the byproduct of a complex computation (e.g. a planner) that might benefit from knowing the formula it is building is unsatisfiable.

An on-line algorithm for 2-SAT must manipulate the underlying data structures to allow efficient clause insertion and satisfiability queries. Recent developments in dynamic graph algorithms suggest that on-line 2-SAT may have a quadratic run time (Khanna, Motwani, & Wilson 1996). Fortunately, 2-SAT does not require such certificates during processing due to the existential nature of it's query. The remainder of this section discusses the benefits of such a view in the form of two algorithms. First, an on-line algorithm for incrementally processing a sequence of binary clauses is presented. A new off-line version is described, as well. Analysis and experimental results will follow.

The On-Line Algorithm

The on-line algorithm consists of three cooperative procedures (Figure 2). The procedure $InsertClause$ receives the clause stream. Two recursive functions, $Search$ and $AssignTrue$ perform most of the work of the algorithm by exploring components and maintaining truth values of the literals. The $Value$ field of a literal can take on one of three values: **true**, **false**, and **unassigned**. Two forms of negation appear below. An overbar ($\overline{a}$) indicates a negated literal, while $\neg$ is a Boolean operator.

The procedure $InsertClause(a, b)$ translates the disjunctive clause $a \vee b$ into its equivalent implications. If both literals are known to be **false**, then processing can stop, as this clause, as well as the entire formula, can not be satisfied. Discovering a true variable allows the algorithm to avoid processing this clause. Tautologies ($a \vee \overline{a}$) can be safely ignored, as well. If one and only one of the literals is **false**, then the other must be **true**, and it can then recursively assign **true** to that literal and all of its implicants. If none of these conditions hold, the algorithm can finally insert the two implications into the graph. According to Theorems 2 and 4, if a contradictory component is formed by the insertion it must contain $\overline{a}$ (or equivalently $\overline{b}$). Hence, it performs a single search from the head of one of the new edges. Searches involving pure variables are avoided per Theorem 1.

Truth maintenance is handled by $AssignTrue$. It assigns truth values to undecided literals in a depth-first fashion,

```
proc InsertClause$(a, b)$
  if $a.Value = $ false $\wedge$ $b.Value = $ false
    then Fail(); // can't satisfy
  elsif $a.Value = $ true $\vee$ $b.Value = $ true
      then return ; // satisfied
  elsif $a.Value = $ false
      then AssignTrue$(b)$; // forced
  elsif $b.Value = $ false
      then AssignTrue$(a)$; // forced
  elsif $a = b$
      then AssignTrue$(a)$; // unit clause
  elsif $a \neq \overline{b}$ // not tautology
      then Add edge $(not(a), b)$ to $E$;
           Add edge $(\overline{b}, a)$ to $E$;
           if $a$ and $b$are not pure
             then Search$(\overline{a}$, UniqueMark()); fi fi.

proc AssignTrue$(v)$
  oneof $v.Value$
        true : return ;
        false : Fail();
        unassigned : $v.Value :=$ true;
        $\overline{v}.Value :=$ false;
        foreach $w$ where $(v, w) \in E$
          do AssignTrue$(w)$; od foeno.

proc Search$(v, mark)$
  Push$(v)$;
  $v.Lowlink := v.Dfnum := count$;
  $count := count + 1$;
  $v.Mark := mark$;
  foreach $w$ where $(v, w) \in E$
    $\wedge$ $w.Value \neq$ true $\wedge$ $w$ is pure do
    if $w.Value = $ false
      then Empty the stack;
           AssignTrue$(\overline{v})$;
           AbortSearch();
    elsif $OnStack(\overline{w})$ then $AssignTrue(w)$;
    elsif $Mark(w) \neq mark$
        then $Search(w, mark)$;
             $v.Lowlink :=$
             $min(v.Lowlink, w.Lowlink)$
    elsif $w.Dfnum < v.LowlinkandOnStack(w)$
        then $v.Lowlink := w.Dfnum$ fi
  od
  if $v.Lowlink = v.Dfnum$
    then repeat $u := Pop()$; until $u = v$; fi.
```

Figure 2: The on-line algorithm. The $InsertClause$ procedure performs simple truth maintenence on the literals, adds implications, and initiates component checking, when necessary. $Search$ recursively explores the dual graph, while $AssignTrue$ maintains truth assignments.

pruning the traversal at vertices with known truth values. Truth values are assigned to the complementary pair of literals: v is assigned **true** and $\overline{v}$ is assigned **false**. A recursive $AssignFalse$ procedure is unnecessary because the graph edges appear in pairs, thus performing the backward push of **false** values that this procedure would provide.

The DFS procedure, $Search$, differs slightly from the one implementing the APT algorithm. Note that the vertex traversal mark is no longer Boolean. The initial call to $Search$ instantiates a unique symbol to mark the current traversal. More importantly, it takes advantage of the logical interpretation of the search vertex to prune the search space. As before, it performs DFS of the graph from the start vertex, v. It is possible, however, to eliminate certain vertex expansions. Expanding a literal whose truth value is known serves no purpose. Pure literals, according to Theorem 1, will always end up in a singleton component, so they should remain unexpanded. $Search$, unlike the generic connected component algorithm, is attempting to traverse the single component containing the original vertex.

The stack serves dual purposes in the new procedure: it is a valid implication chain in addition to the component holding area. The literals on the stack were pushed in their implication order, $x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_i \rightarrow v$, where the index of x_i reflects the order that literals were pushed onto the stack and v is the current literal. Consider the event where the search algorithm stumbles upon a literal, w whose truth value is **false** while at v. Given that w is a child of v, $v \implies w$. The only way to avoid a contradiction is to assign v the value **false**. During the $AssignTrue(\overline{v})$ call this assignment propagates down the stack, with eventually all the stack literals, including the root literal, receive an assignment. Hence, the entire traversal can be aborted.

Another assignment can be determined from the stack. If an implication chain exists from a literal to it's negation, the negation is asserted **true**. This rule can be implemented with an examination of the stack for the negation of the vertex whose expansion in question.

If the vertex was not marked during the current traversal, then it is expanded and v's backedge is updated accordingly. In the case of a backedge, a small simplification can be performed. Since $Lowlink$ is assigned the value of the vertex's traversal number, $Dfnum$, or the minimum of itself and some other value, $Lowlink \leq Dfnum$ for all vertices. The APT algorithm performs two comparisons involving $Lowlink$ on the backlink test. First, the target must have appeared previously in the DFS, $w.Dfnum < v.Dfnum$. Second, v receives a new $Lowlink$ if $w.Dfnum < v.Lowlink$. The second comparison subtends the first, so the former can be replaced by the latter and used to record the destination of the new backedge.

The algorithm fills the stack with candidate literals during its DFS of the implication graph. When it encounters a literal whose $Lowlink$ points to itself after visiting it's progeny, it has discovered the root vertex of the current component occupying the topmost portion of the stack. In the APT algorithm, the stack is checked for the negations of all literals of the component. This check, however, can be reduced to one comparison: the presence of the root's negation. Theorem 4 supports this simplification.

Even this comparison can be eliminated. During DFS of a contradictory component, the negation of some literal, $\overline{w}$, will be found on the stack and trigger the assignment of **true** to the newly found literal, w. Recall that truth assignments are recursive, all literals found via DFS from the source are assigned **true** and the negated literals are assigned **false**. The complimentary pair assignments occur before vertex expansion in $AssignTrue$. Since $w \leadsto \overline{w}$, the algorithm will eventually try to make a contradictory assignment to $\overline{w}$ and trigger a failure. When the algorithm safely returns to component's root vertex, it is guaranteed not to contain any contradictory literal pairs. Thus, the constituents need only be popped from the stack.

The Off-Line Algorithm

The APT algorithm, separated the two abstractions, logical and graph, that made their algorithm work. While converting the formula to a graph and looking for contradictory components solves the satisfiability problem and achieving asymptotic optimality, it fails to take advantage of regularities of the graph submitted to component subroutine. The remainder of this section addresses optimization of the off-line algorithm in light of the on-line algorithm.

The new off-line 2-SAT algorithm still works depth-first, but it is able to prune a great many vertex expansions that the APT algorithm would follow. It can operate with a single vertex mark, as the graph is stable. It can not, however, rely on truth assignments to pick up all contradictions. Testing a component for contradictions requires a single check for the negation of the root literal. The main procedure differs as well. Recall that pure literals are in singleton components, so they are skipped. Assigned literals need no further expansion, either. The algorithm expands literals only if they are not negated and neither it, nor its negation, are marked (contradictory components are closed under negation by Theorem 4). Expanding from the main loop only positive literals applies this reasoning to limit the loop construct and avoiding mark and purity tests. This optimization mirrors the single DFS invoked in $InsertClause$.

```
proc OffLine2-SAT($F$)
  Construct the dual graph $G = (V, E)$ of $F$, then
  foreach non-negated $v \in E$
    $\wedge\ w.Value =$ unassigned
    $\wedge\ w$ is not pure
    $\wedge\ \neg w.Mark \wedge \neg \overline{w}.Mark$ do
    $SearchOffLine(v)$; od
  return $\neg SearchFailed()$; .
proc SearchOffLine($v$)
  Push($v$);
  $v.Lowlink := v.Dfnum := count$;
  $count := count + 1$;
  $v.Mark := true$;
  foreach $w$ where $(v, w) \in E$
    $\wedge\ w.Value \neq true \wedge$ OutEdges$(w) > 0$ do
    if $w.Value =$ false
      then $EmptyStack()$;
           $AssignTrue(\overline{v})$;
           $AbortSearch()$;
    elsif $OnStack(\overline{w})$ then $AssignTrue(w)$;
    elsif $\neg w.Mark$
        then $Search(w)$;
             $v.Lowlink :=$
             $min(v.Lowlink, w.Lowlink)$
    elsif $w.Dfnum < v.Lowlink \wedge OnStack(w)$
        then $v.Lowlink := w.Dfnum$ fi
  od
  if $v.Lowlink = v.Dfnum$
    then if $\overline{u}$ is on the stack after $v$
           then $Fail()$; fi
         repeat $u := Pop()$; until $u = v$; fi.
```

Figure 3: The 2-SAT algorithm of Aspvall, Plass, and Tarjan (Aspvall, Plass, & Tarjan 1979) embedded into Tarjan's strongly connected connected components algorithm (Tarjan 1972).

Analysis and Experimental Results

The linear run time of the APT algorithm sets it apart from other satisfiability algorithms. In this section, the asymptotic run times of both algorithms are examined.

Unfortunately, the off-line version worst-case run time is $\theta(n^2)$. This behavior can be induced by a clause stream containing groups of three clause pairs of the form $(\overline{x_{i-1}} \vee \overline{x_i}) \wedge (x_i \vee y) \wedge (\overline{x_i} \vee z_i)$. The first clause inserts the edges $(\overline{x_i}, x_{i-1})$ and $(\overline{x_{i-1}}, x_i)$. Since this is the first appearance of x_i, it's purity postpones it's expansion to a future insert. When the second clause is inserted, the edge $(\overline{z_i}, x_i)$ is added and x_i is defiled. If z is impure as well (easily induced by $w \vee y$), then x_i is traversed. Thus, there exists a clause stream of length $3n + 2$ whose run time is $\Theta(n^2)$ as all x_i are recursively visited on each insert. This run time is asymptotically the same as running the original algorithm on each iteration. Fortunately, such expressions are uncommon.

The average run time is much more acceptable than this quadratic upper bound. While a proof is elusive at this time, experimental results suggest linear expected run time. The experiments described below compare the run times of four different versions of the algorithm: the off-line algorithm where value truth values are identified and propagated (once assign) or not (once), the on-line approach utilizing variable assignments (insert assign) or not (insert). Two hypotheses were tested: first, that run time grows linearly with problem size; second, that variable assignments improved performance. The latter is of interest as truth maintenance adds significant complexity to the algorithm.

Each algorithm was ran on variable sets of size 1000, 3500, 6500, 10000, 35000, 65000, and 100000. The ratio of clauses to variables examined were 0.9, 1.0, 1.1, 1.2, 2.0, and 5.0. For each condition, variable set size and clause ratio, 100 clause sets were generated and presented to each of the algorithms. Clauses were independently generated by selecting, with replacement, two literals from L under the assumption of uniform probability. Presentations to the on-line versions maintained identical orderings of the clause sets.

Table 1 displays the mean run times for the experimental conditions conditions. To test the linearity hypothesis,

Clause Ratio	Variables	Once Assign (sec)	Once (sec)	Insert (sec)	Insert Assign (sec)
0.9	1000	0.005	0.005	0.005	0.004
	3500	0.010	0.011	0.010	0.010
	6500	0.018	0.019	0.019	0.020
	10000	0.030	0.027	0.031	0.031
	35000	0.096	0.098	0.107	0.114
	65000	0.180	0.181	0.205	0.215
	100000	0.274	0.280	0.318	0.333
1.0	1000	0.005	0.004	0.005	0.005
	3500	0.011	0.011	0.012	0.014
	6500	0.020	0.024	0.024	0.024
	10000	0.030	0.031	0.039	0.041
	35000	0.104	0.107	0.182	0.182
	65000	0.192	0.196	0.368	0.361
	100000	0.296	0.301	0.592	0.578
1.1	1000	0.005	0.005	0.006	0.006
	3500	0.012	0.011	0.017	0.015
	6500	0.021	0.021	0.037	0.030
	10000	0.030	0.030	0.077	0.053
	35000	0.100	0.097	0.394	0.238
	65000	0.184	0.175	0.838	0.489
	100000	0.287	0.270	1.380	0.775
1.2	1000	0.005	0.005	0.007	0.006
	3500	0.013	0.012	0.020	0.016
	6500	0.021	0.022	0.045	0.032
	10000	0.034	0.031	0.082	0.055
	35000	0.115	0.103	0.397	0.238
	65000	0.203	0.184	0.837	0.492
	100000	0.322	0.287	1.383	0.776
2.0	1000	0.006	0.006	0.007	0.006
	3500	0.017	0.015	0.020	0.016
	6500	0.028	0.026	0.044	0.032
	10000	0.046	0.039	0.083	0.055
	35000	0.164	0.134	0.398	0.239
	65000	0.271	0.230	0.839	0.493
	100000	0.466	0.378	1.380	0.773
5.0	1000	0.007	0.006	0.008	0.007
	3500	0.016	0.018	0.020	0.016
	6500	0.033	0.031	0.045	0.033
	10000	0.055	0.050	0.083	0.055
	35000	0.188	0.174	0.399	0.241
	65000	0.309	0.282	0.845	0.497
	100000	0.612	0.535	1.386	0.783

Table 1: Comparison of run-time's of the algorithms.

variable count and mean run times with a ratio condition were subjected to regression. Across all ratio conditions, the smallest R^2 value was 0.9941 strongly indicating a linear relationship between problem size to run time. Performing variable assignments had minimal impact on the offline algorithm, but significantly reduced the run time of the online version.

Conclusion

Satisfiability is an important problem for both AI and computer science. Many AI problems, such as planning, have been shown to be NP-complete. The algorithm described above, an on-line 2-SAT algorithm, can be brought to bear in restricted cases of various AI problems that can be translated into 2-SAT instances.

Future work includes theoretical validation of the experimental results described above. The on-line algorithm will also be enhanced to allow rollback. That is, the partially-dynamic version will be able to forget an arbitrary number of the most recent clauses and process new clause inserts as if the forgotten clauses never appeared. More importantly, however, is the application of the on-line algorithm to a novel k-SAT algorithm. This new approach to k-SAT reduces the problem instance into an exponential number of (k-1)-SAT instances. A preprocessing step iteratively removes clauses from the set containing the most common literal. The removed clauses are associated with the literal. The algorithm then examines all possible truth assignments to the common literals. The on-line version described above will allow the algorithm to prune the tree during the brute-force search by examining prefixes to the generated clauses. The envisioned partially-dynamic 2-SAT algorithm will help even more with rollback mimicking the variable assignments.

Acknowledgements

This work was supported by the Office of Naval Research, grant number N00014-01-1-0926.

References

Aho, A.; Hopcroft, J.; and Ullman, J. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Co.

Aspvall, B.; Plass, M.; and Tarjan, R. 1979. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters* 8:121–123.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, 151–158. Association for Computing Machinery.

Khanna, S.; Motwani, R.; and Wilson, R. 1996. On certificates and lookahead in dynamic graph problems. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

Preparata, F. P. 1979. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM* 22:402–405.

Tarjan, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1:146–160.