Query-Guided Maximum Satisfiability

Xin Zhang† Ravi Mangal‡ Aditya V. Nori* Mayur Naik†
† Georgia Institute of Technology, USA ‡ Microsoft Research, UK
xin.zhang@gatech.edu ravi.mangal@gatech.edu adityan@microsoft.com naik@cc.gatech.edu

Abstract

We propose a new optimization problem “Q-MAXSAT”, an extension of the well-known Maximum Satisfiability or MAXSAT problem. In contrast to MAXSAT, which aims to find an assignment to all variables in the formula, Q-MAXSAT computes an assignment to a desired subset of variables (or queries) in the formula. Indeed, many problems in diverse domains such as program reasoning, information retrieval, and mathematical optimization can be naturally encoded as Q-MAXSAT instances.

We describe an iterative algorithm for solving Q-MAXSAT. In each iteration, the algorithm solves a subproblem that is relevant to the queries, and applies a novel technique to check whether the partial assignment found is a solution to the Q-MAXSAT problem. If the check fails, the algorithm grows the subproblem with a new set of clauses identified as relevant to the queries.

Our empirical evaluation shows that our Q-MAXSAT solver PILOT achieves significant improvements in runtime and memory consumption over conventional MAXSAT solvers on several Q-MAXSAT instances generated from real-world problems in program analysis and information retrieval.

Categories and Subject Descriptors G.1.6 [optimization]: constrained optimization

General Terms Algorithms, Theory

Keywords Optimization, Maximum Satisfiability, Partial Model, Query-Guided Approach, Program Analysis, Information Retrieval

1. Introduction

The maximum satisfiability or MAXSAT problem [47] is an optimization extension of the well-known satisfiability or SAT problem [13, 17]. A MAXSAT formula consists of a set of conventional SAT or hard clauses together with a set of soft or weighted clauses. The solution to a MAXSAT formula is an assignment to its variables that satisfies all the hard clauses, and maximizes the sum of the weights of satisfied soft clauses. A number of interesting problems that span across a diverse set of areas such as program reasoning [24, 25, 35, 58], information retrieval [14, 29, 45, 48], databases [5, 40], circuit design [56], bioinformatics [16, 52], planning and scheduling [26, 53, 57], and many others can be naturally encoded as MAXSAT formulae. Many of these problems specify hard constraints that encode various soundness conditions, and soft constraints that specify objectives to optimize.

MAXSAT solvers have made remarkable strides in performance over the last decade [4, 8, 20, 22, 37, 39, 41, 42, 44]. This in turn has motivated even more demanding and emerging applications to be formulated using MAXSAT. Real-world instances of many of these problems, however, result in very large MAXSAT formulae [36]. These formulae, comprising millions of clauses or more, are beyond the scope of existing MAXSAT solvers. Fortunately, for many of these problems, one is interested in a small set of queries that constitute a very small fraction of the entire MAXSAT solution. For instance, in program analysis, a query could be analysis information for a particular variable in the program—intuitively, one would expect the computational cost for answering a small set of queries to be much smaller than the cost of computing analysis information for all program variables. In the MAXSAT setting, the notion of a query translates to the value of a specific variable in a MAXSAT formula. Given a MAXSAT formula $\varphi$ and a set of queries $Q$, one obvious method for answering queries in $Q$ is to compute the MAXSAT solution to $\varphi$ and project it to variables in $Q$. Needless to say, especially for very large MAXSAT formulae, this is a non-scalable solution. Therefore, it is interesting to ask the following question: “Given a MAXSAT formula $\varphi$ and a set of queries $Q$, is it possible to answer $Q$ by only computing information relevant to $Q$?”. We call this question the query-guided maximum satisfiability or Q-MAXSAT problem ($\varphi, Q$).

Our main technical insight is that a Q-MAXSAT instance ($\varphi, Q$) can be solved by computing a MAXSAT solution of a small subset of the clauses in $\varphi$. The main challenge, however, is how to efficiently determine whether the answers to $Q$ indeed correspond to a MAXSAT solution of $\varphi$. We propose an iterative algorithm for solving a Q-MAXSAT instance ($\varphi, Q$) (Algorithm 1 in Section 4). In each iteration, the algorithm computes a MAXSAT solution to a subset of clauses in $\varphi$ that are relevant to $Q$. We also define an algorithm (Algorithm 2 in Section 4.1) that efficiently checks whether the current assignment to variables in $Q$ corresponds to a MAXSAT solution to $\varphi$. In particular, our algorithm constructs a small set of clauses that succinctly summarize the effect of the clauses in $\varphi$ that are not considered by our algorithm, and then uses it to overestimate the gap between the optimal objective value of $\varphi$ under the current assignment and the optimum objective value of $\varphi$.

We have implemented our approach in a tool called PILOT and applied it to 19 large MAXSAT instances ranging in size from 100 thousand to 22 million clauses generated from real-world problems in program analysis and information retrieval. Our empirical evaluation shows that PILOT achieves significant improvements in runtime and memory over conventional MAXSAT solvers: on these instances, PILOT used only 285 MB of memory on average and terminated in 107 seconds on average. In contrast, conventional MAXSAT solvers timed out for eight of the instances.
We illustrate the Q-MAXSAT in conjunctive form. Each variable \( \varphi \) when it solves the Q-MAXSAT instance \((\varphi, \{v_6\})\).

In summary, the main contributions of this paper are as follows:

1. We introduce and formalize a new optimization problem called Q-MAXSAT. In contrast to traditional MAXSAT, where one is interested in an assignment to all variables, for Q-MAXSAT, we are interested only in an assignment to a subset of variables, called queries.

2. We propose an iterative algorithm for Q-MAXSAT that has the desirable property of being able to efficiently check whether an assignment to queries is an optimal solution to Q-MAXSAT.

3. We present empirical results showing the effectiveness of our approach by applying PILOT to several Q-MAXSAT instances generated from real-world problems in program analysis and information retrieval.

### Example

We illustrate the Q-MAXSAT instance and our solution with the help of an example. Figure 1 represents a large MAXSAT formula \( \varphi \) in conjunctive form. Each variable \( v_i \) in \( \varphi \) is represented as a node in the graph labeled by its subscript \( i \). Each clause is a disjunction of literals with a positive weight. Nodes marked as T (or F) indicate soft clauses of the form \((100, v_i)\) (or \((100, \neg v_i)\)) while each edge from node \( i \) to a node \( j \) represents a soft clause of the form \((5, \neg v_i \lor v_j)\).

Suppose we are interested in the assignment to the query \( v_6 \) in \( \varphi \) (shown by the dashed node in Figure 1). Then, the Q-MAXSAT instance that we want to solve is \((\varphi, \{v_6\})\). A solution to this Q-MAXSAT instance is a partial model which maps \( v_6 \) to true or false such that there is a completion of this partial model maximizing the objective value of \( \varphi \).

A naive approach to solve this problem is to directly feed \( \varphi \) into a MAXSAT solver and extract the assignment to \( v_6 \) from the solution. However, this approach is highly inefficient due to the large size of practical instances.

Our approach exploits the fact that the Q-MAXSAT instance only requires assignment to specific query variables, and solves the problem lazily in an iterative manner. The high level strategy of our approach (see Section 4 for details) is to only solve a subset of clauses relevant to the query in each iteration, and terminate when the current solution can be proven to be the solution to the Q-MAXSAT instance. In particular, our approach proceeds in the following four steps.

**Initialize.** Given a Q-MAXSAT instance, our query-guided approach constructs an initial set of clauses that includes all the clauses containing a query variable. We refer to these relevant clauses as the workset of our algorithm.

**Solve.** The next step is to solve the MAXSAT instance induced by the workset. Our approach uses an existing off-the-shelf MAXSAT solver for this purpose. This produces a partial model of the original instance.

**Check.** The key step in our approach is to check whether the current partial model can be completed to a model of the original MAXSAT instance. Since the workset only includes a small subset of clauses from the complete instance, the challenge here is to summarize the effect of the unexplored clauses on the assignment to the query variables. We propose a novel technique for performing this check efficiently without actually considering all the unexplored clauses (see Section 4.1 for formal details).

**Expand.** If the check in the previous step fails, it indicates that we need to grow our workset of relevant clauses. Based on the check failure, in this step, our approach identifies the set of clauses to be added to the workset for the next iteration.

As long as the Q-MAXSAT instance is finite, our iterative approach is guaranteed to terminate since it only grows the workset in each iteration.

We next describe how our approach solves the Q-MAXSAT instance \((\varphi, \{v_6\})\). To resolve \( v_6 \), our approach initially constructs the workset \( \varphi' \) that includes all the clauses containing \( v_6 \). We represent \( \varphi' \) by the subgraph contained within the dotted area in Figure 2(a). By invoking a MAXSAT solver on \( \varphi' \), we get a partial model \( \alpha_{\varphi'} = \{v_4 \mapsto \text{false}, v_5 \mapsto \text{false}, v_6 \mapsto \text{false}, v_7 \mapsto \text{false}, v_8 \mapsto \text{false}\} \) as shown in Figure 2(a), with an objective value of 20. Our approach next checks if the current partial model found is a solution to the Q-MAXSAT instance \((\varphi, \{v_6\})\). It constructs a set of clauses \( \psi \) which succinctly summarizes the effect of the clauses that are not present in the workset. We refer to this set as the summarization set, and use the following expression to overestimate the gap between the optimum objective value of \( \varphi \) under the current partial model and the optimum objective value of \( \varphi' \):

\[
\max_{\alpha} \text{eval}(\varphi' \cup \psi, \alpha) - \max_{\alpha} \text{eval}(\varphi', \alpha),
\]

where \( \max_{\alpha} \text{eval}(\varphi' \cup \psi, \alpha) \) and \( \max_{\alpha} \text{eval}(\varphi', \alpha) \) are the optimum objective values of \( \varphi' \cup \psi \) and \( \varphi' \), respectively.

To construct such a summarization set, our insight is that the clauses that are not present in the workset can only affect the query assignment via clauses sharing variables with the workset. We call such clauses as the frontier clauses. Furthermore, if a frontier clause is already satisfied by the current partial model, expanding the workset with such a clause cannot further improve the partial model. We now try to construct a summarization set \( \psi \) by taking all frontier clauses not satisfied by \( \alpha_{\varphi'} \). As a result, our algorithm produces \( \psi = \{\{100, v_4\}, \{100, v_6\}\} \). The check comparing the optimum objective values of \( \varphi' \) and \( \varphi' \cup \psi \) fails in this case. In particular, by solving \( \varphi' \cup \psi \), we get a partial model \( \alpha_{\varphi' \cup \psi} = \{v_4 \mapsto \text{true}, v_5 \mapsto \text{true}, v_6 \mapsto \text{true}, v_7 \mapsto \text{true}, v_8 \mapsto \text{true}\} \) with 220 as the objective value, which is greater than the optimum objective value of \( \varphi' \). As a consequence, our approach expands the

![Figure 1](image1.png)

**Figure 1:** Graph representation of a large MAXSAT formula \( \varphi \).

![Figure 2](image2.png)

(a) Iteration 1.

(b) Iteration 2.

(c) Iteration 3.

**Figure 2:** Graph representation of each iteration in our algorithm when it solves the Q-MAXSAT instance \((\varphi, \{v_6\})\).
workset $\varphi'$ with (100, $v_4$) and (100, $v_5$) and proceeds to the next iteration. We include these two clauses as these are not satisfied by $\alpha_{\varphi'}$ in the last iteration, but satisfied by $\alpha_{\varphi' \cup \psi}$, which indicates they are likely responsible for the failure of the previous check.

In iteration 2, invoking a MAXSAT solver on the new workset, $\varphi'$, produces $\alpha_{\varphi'} = [v_4 \rightarrow \text{true}, v_5 \rightarrow \text{true}, v_6 \rightarrow \text{true}, v_7 \rightarrow \text{false}, v_8 \rightarrow \text{true}]$, as shown in Figure 2(b), with 220 as the objective value. The violated frontier clauses in this case are (100, $-v_7$), (5, $-v_5 \lor v_2$), and (5, $-v_5 \lor v_3$). However, if this set of violated frontier clauses were to be used as the summarization set $\psi$, the newly added variables $v_2$ and $v_3$ will cause the check comparing the optimum objective values of $\varphi'$ and $\varphi' \cup \psi$ to trivially fail. To address this problem, and further improve the precision of our check, we modify the summarization set as $\psi' = \{(100, -v_7), (5, -v_5 \lor v_2), (5, -v_5 \lor v_3)\}$ by removing $v_2$ and $v_3$ and strengthening the corresponding clauses. In this case, it is equivalent to setting $v_2$ and $v_3$ to false (marked as $\overline{\varphi}$ in the graph). Intuitively, by setting these variables to false, we are overestimating the effects of the unexplored clauses, by assuming these frontier clauses will not be satisfied by unexplored variables like $v_2$ and $v_3$. By solving $\varphi' \cup \psi'$, we get a partial model $\alpha_{\varphi' \cup \psi'} = [v_4 \rightarrow \text{true}, v_5 \rightarrow \text{false}, v_6 \rightarrow \text{true}, v_7 \rightarrow \text{false}, v_8 \rightarrow \text{true}]$ with 320 as the objective value, which is greater than the optimum objective value of $\varphi'$.

In iteration 3, as shown in Figure 2(c), our approach expands the workset $\varphi'$ with (100, $-v_2$), (5, $-v_5 \lor v_2$) and (5, $-v_5 \lor v_3$). By invoking a MAXSAT solver on $\varphi'$, we produce $\alpha_{\varphi'} = [v_2 \rightarrow \text{true}, v_3 \rightarrow \text{false}, v_4 \rightarrow \text{true}, v_5 \rightarrow \text{true}, v_7 \rightarrow \text{false}, v_8 \rightarrow \text{true}]$ with an objective value of 325. We omit the edges representing frontiers clauses that are already satisfied by $\alpha_{\varphi'}$ in Figure 2(c). To check the whether current partial model is a solution to the Q-MAXSAT instance, we construct strengthened summarization set $\psi = \{(5, -v_3), (5, -v_3 \lor v_1)\}$ from the frontier clause (5, $-v_3 \lor v_1$). By invoking MAXSAT on $\varphi' \cup \psi$, we get our optimum objective value of 325 which is the same as that of $\varphi'$. As a result, our algorithm terminates and extracts [$v_6 \rightarrow \text{true}$] as the solution to the Q-MAXSAT instance.

Despite the fact that many clauses and variables can reach $v_6$ in the graph (i.e., they might affect the assignment to the query), our approach successfully resolves $v_6$ in three iterations and only explores a very small, local subset of the graph, while successfully summarizing the effects of the unexplored clauses.

3. The Q-MAXSAT Problem

First, we describe the standard MAXSAT problem [47]. The syntax of a MAXSAT formula is shown in Figure 3. A MAXSAT formula $\varphi$ consists of a set of soft clauses. Each soft clause $s = (w, c)$ is a pair that consists of a positive weight $w \in \mathbb{R}^+$, and a clause $c$ that is a disjunctive form over a set of variables $\mathcal{V}$.

Algorithm 1

1: INPUT: A Q-MAXSAT instance $(\varphi, Q)$, where $\varphi$ is a MAXSAT formula, and $Q$ is a set of queries.
2: OUTPUT: A solution to the Q-MAXSAT instance $(\varphi, Q)$.
3: $\varphi' := \text{INIT}(\varphi, Q)$.
4: while true do
5: $\alpha_{\varphi'} := \text{MAXSAT}(\varphi')$
6: $\varphi'' := \text{CHECK}((\varphi, Q), \varphi', \alpha_{\varphi'})$, where $\varphi'' \subseteq \varphi' \setminus \varphi'$
7: if $\varphi'' = \emptyset$ then
8: return $\forall v. \alpha_{\varphi}(v)$, where $\forall v \in Q$
9: else
10: $\varphi' := \varphi' \cup \varphi''$
11: end if
12: end while

Figure 3: Syntax and interpretation of MAXSAT formulæ.

\[ fst = \lambda(w, c).w, \quad snd = \lambda(w, c).c \]

\[ \forall v. \alpha = 1, \quad \forall \alpha \neq 0 \]

\[ \alpha = \bigvee_{i=1}^n v_i \lor \bigwedge_{j=1}^m \neg v' \quad \exists i : \alpha(v_i) = 1 \lor \exists j : \alpha(v'_j) = 0 \]

\[ \text{eval}(s_1, \ldots, s_n, \alpha) = \sum_{i} \text{fst}(s_i) | \alpha = \text{snd}(s_i) \]

\[ \text{MAXSAT}(\varphi) = \operatorname{argmax}_{\alpha} \text{eval}(\varphi, \alpha) \]

Example. Let $(\varphi, Q) = ((5, -v_5 \lor v_2), (5, -v_5 \lor v_3), (5, -c \lor d), (5, -d))$ and $Q = \{a\}$ be a Q-MAXSAT instance. A model of this instance is given by $\alpha_{Q} = [a \rightarrow 0]$. Indeed, there is a completion $\alpha = [a \rightarrow 0, b \rightarrow 0, c \rightarrow 0, d \rightarrow 0]$ that belongs to $\text{MaxSAT}(\varphi)$ (in other words, $\alpha$ maximizes the objective value of $\varphi$, which is equal to 20), and agrees with $\alpha_{Q}$ on the variable $a$ (that is, $\alpha(a) = \alpha_{Q}(a) = 0$).

Hereafter, we use $\forall v. \text{Var}(\varphi)$ to denote the set of variables occurring in MAXSAT formula $\varphi$. For a set of variables $U \subseteq \forall v. \text{Var}(\varphi)$, we denote the partial model $\alpha : U \rightarrow \{0, 1\}$ by $\alpha_{U}$. Also, we use $\alpha_{\varphi}$ as shorthand for $\alpha_{\forall v. \text{Var}(\varphi)}$. Throughout the paper, we assume that for $\text{eval}(\varphi, \alpha)$, the assignment $\alpha$ is well-defined for all variables in $\forall v. \text{Var}(\varphi)$.

4. Solving a Q-MAXSAT Instance

Algorithm 1 describes our technique for solving the Q-MAXSAT problem. It takes as input a Q-MAXSAT instance $(\varphi, Q)$, and returns a solution to $(\varphi, Q)$ as output. The main idea in the algorithm is to iteratively identify and solve a subset of clauses in $\varphi$ that are relevant to the set of queries $Q$, which we refer to as the workset of our algorithm.

1Without loss of generality, we assume that MAXSAT formulæ contain only soft clauses. Every hard clause can be converted into a soft clause with a sufficiently high weight.
Algorithm 2 CHECK((ϕ, Q), ϕ', α, ω)
1: INPUT: A Q-MAXSAT instance (ϕ, Q), a MAXSAT formula ϕ' ⊆ ϕ, and a model αω ∈ MaxSAT(ϕ').
2: OUTPUT: A set of clauses ϕ'' ⊆ ϕ \ ϕ'.
3: ψ := APPROX(ϕ', ϕ', α)
4: ϕ'' := ϕ' ∪ ψ
5: αω'' := MAXSAT(ϕ'')
6: if eval(ϕ'', αω'') = eval(ϕ', αω) then
7: return ∅
8: else
9: ϕs := {((w, c) | (w, c) ∈ ψ ∧ αω'' |= c}
10: return REFINE(ϕs, ϕ', ϕ', αω'')
11: end if

The algorithm starts by invoking function INIT (line 3) which returns an initial set of clauses ϕ' ⊆ ϕ. Specifically, ϕ' is the set of clauses in ϕ which contain variables in the query set Q.

In each iteration (lines 4–12), Algorithm 1 first computes a model αω' of the MAXSAT formula ϕ' (line 5). Next, in line 6, the function CHECK checks whether the model αω' is sufficient to compute a model of the Q-MAXSAT instance (ϕ, Q). If αω' is sufficient, then the algorithm returns a model which is αω restricted to the variables in Q (line 8). Otherwise, CHECK returns a set of clauses ϕ'' that is added to the set ϕ' (line 10). It is easy to see that Algorithm 1 always terminates if ϕ is finite.

The main and interesting challenge is the implementation of the CHECK function so that it is sound (that is, when CHECK returns ∅, then the model αω restricted to the variables in Q is indeed a model of the Q-MAXSAT instance), yet efficient.

4.1 Implementing an Efficient CHECK Function

Algorithm 2 describes our implementation of the CHECK function. The input to the CHECK function is a Q-MAXSAT instance (ϕ, Q), a MAXSAT formula ϕ' ⊆ ϕ as described in Algorithm 1, and a model αω' ∈ MaxSAT(ϕ'). The output is a set of clauses ϕ'' that are required to be added to ϕ' so that the resulting MAXSAT formula ϕ' ⊆ ϕ'' is solved in the next iteration of Algorithm 1. If ϕ'' is empty, then this means that Algorithm 1 can stop and return the appropriate model (as described in line 8 of Algorithm 1).

CHECK starts by calling the function APPROX (line 3) which takes ϕ, ϕ' and αω' as inputs, and returns a new set of clauses ψ, which we refer to as the summarization set. APPROX analyzes clauses in ϕ \ ϕ', and returns a much smaller formula ψ which allows us to overestimate the gap between the optimum objective value under current partial model αω' and the optimum objective value of ϕ. In line 5, CHECK computes a model αω'' of the MAXSAT formula ϕ'' = ϕ' ∪ ψ. Next, in line 6, CHECK compares the objective value of ϕ' with respect to αω' and the objective value of ϕ'' with respect to αω''. If these objective values are equal, CHECK concludes that the partial assignment for the queries in αω'' is a model to the Q-MAXSAT problem and returns an empty set (line 7). Otherwise, it computes ϕs in line 9 which is the set of clauses satisfied by αω'' in ψ. Finally, in line 10, CHECK returns the set of clauses to be added to the MAXSAT formula ϕ''. This is computed by REFINE, which takes ϕs, ϕ', ϕ', and αω'' as input. Essentially, REFINE identifies the clauses in ϕ \ ϕ'' which are likely responsible for failing the check in line 6, and uses them to expand the MAXSAT formula ϕ''.

Optimality check via APPROX. The core step of CHECK is line 6, which uses eval(ϕ'', αω'') = eval(ϕ', αω') to overestimate the gap between the optimum objective value under current partial model αω' and the optimum objective value of ϕ. The key idea here is to apply APPROX to generate a small set of clauses ψ which succinctly summarizes the effect of the unexplored clauses ϕ \ ϕ'. We next describe the specification of the APPROX function, and formally prove the soundness of the optimality check in line 6.

Given a set of variables U ⊆ Var(ϕ), and an assignment αU, we define a substitution operation ϕ(αU) which simplifies ϕ by replacing all occurrences of variables in U with their corresponding values in assignment αU. Formally,

\[
\begin{align*}
\{ s_1, ..., s_n | [α_U] \} [α_U] &= \{ s_1 [α_U], ..., s_n [α_U] \} \\
\{ s | [α_U] \} [α_U] &= \{ s [α_U] \} \\
0 [α_U] &= 0
\end{align*}
\]

Definition 2 (Summarizing unexplored clauses). Given two MAXSAT formulae ϕ, ϕ such that ϕ' ⊆ ϕ, and αω ∈ MaxSAT(ϕ'), we say ψ = APPROX(ϕ, ϕ', αω) summarizes the effect of ϕ \ ϕ' with respect to αω if and only if:

\[
\max_α eval(ϕ' \cup ψ, α) ≥ \max_α eval(ϕ, α) - \max_α eval(ϕ \setminus ϕ', [α])]
\]

We state two useful facts about eval before proving soundness.

Proposition 3. Let ϕ and ϕ' be two MAXSAT formulae such that ϕ' \ ϕ = 0. Then, ∀α.eval(ϕ ∪ ψ, α) = eval(ϕ, α) + eval(ψ, α).

Proposition 4. Let ϕ and ϕ' be two MAXSAT formulae. Then,

\[
\max_α eval(ϕ, α) + \max_α eval(ϕ', α) ≥ \max_α eval(ϕ ∪ ψ, α)
\]

The theorem below states the soundness of the optimality check performed in line 6 in CHECK.

Theorem 5 (Soundness of optimality check). Given a Q-MAXSAT instance (ϕ, Q), a MAXSAT formula ϕ' ⊆ ϕ s.t. Var(ϕ') ⊆ Q, a model αω ∈ MaxSAT(ϕ'), and ψ = APPROX(ϕ, ϕ', αω) such that the following condition holds:

\[
\max_α eval(ϕ' \cup ψ, α) = \max_α eval(ϕ', α)
\]

Then, λν.αω(ν), where ν ∈ Q is a model of the Q-MAXSAT instance (ϕ, Q).

Proof. Let αQ = λν.αω(ν), where ν ∈ Q. Let ϕ'' = ϕ[αω]. Construct a completion αω of αQ as follows:

\[
αω = αω' \sqcup αω''
\]

where αω'' ∈ MaxSAT(ϕ'')

It suffices to show that αω ∈ MaxSAT(ϕ), that is, eval(ϕ, αω) = maxα.eval(ϕ, α). We have:

\[
\begin{align*}
\text{eval}(ϕ, αω) &= \text{eval}(ϕ, αω' \sqcup αω'') \\
&= \text{eval}(ϕ[αω'], αω') \\
&= \max_α \text{eval}(ϕ[αω'], α) \\
&= \max_α \text{eval}(ϕ' [αω'] \cup (ϕ \setminus ϕ'))[αω], α)\quad \text{[from Prop. 3]}
\end{align*}
\]

= \max_α (\text{eval}(ϕ', [αω'], α) + \text{eval}((ϕ \setminus ϕ')[αω], α)) \quad \text{[from Prop. 3]}

= \max_α \text{eval}(ϕ', ω) + \text{eval}((ϕ \setminus ϕ')[αω], α)

= \max_α \text{eval}(ϕ' [αω'], α) \\
&= \text{eval}(ϕ, αω)
\]
\[= eval(\varphi', \alpha_{\varphi'}) + \max eval((\varphi \setminus \varphi')\{\alpha_{\varphi'}\}, \alpha)\]
\[\geq eval(\varphi', \alpha_{\varphi'}) + \max eval(\varphi, \alpha) - \max \alpha_{\varphi'} \leq \psi, \alpha)\]
\[\text{[since } \psi = \text{ APPROX}(\varphi, \alpha), \text{ see Defn. 2]}\]
\[= \max \alpha_{\varphi'} \leq \psi, \alpha) + \max eval(\varphi, \alpha) - \max \alpha_{\varphi'} \leq \psi, \alpha)\]
\[\text{[since } \alpha_{\varphi'} \in \text{ MaxSAT}(\varphi')]\]
\[= \max eval(\varphi' \cup \psi, \alpha) + \max \alpha_{\varphi'} \leq \psi, \alpha) - \max \alpha_{\varphi'} \leq \psi, \alpha)\]
\[\text{[from the condition defined in the theorem statement]}\]
\[\alpha = \max eval(\varphi, \alpha)\]

Discussion. There are number of possibilities for the APPROX function that satisfy the specification in Definition 2. The quality of such an APPROX function can be measured by two criteria: the efficiency and the precision of the optimality check (lines 3 and 6 in Algorithm 2).

Given that eval can be efficiently computed, the cost of the optimality check primarily depends on the cost of computing \(\text{APPROX} \varphi, \alpha\), \(\alpha_{\varphi'}\), see Defn. 2\] and the cost of invoking the MAXSAT solver on \(\varphi' \cup \psi\). Since MAXSAT is known to be a computationally hard problem, a simple \(\psi\) returned by APPROX can significantly speedup the optimality check.

On the other hand, a precise optimality check can significantly reduce the number of iterations of Algorithm 1. We define a partial order \(\preceq\) on APPROX functions that compares the precision of the optimality check via them. We say APPROX_1 is more precise than APPROX_2 (denoted by APPROX_2 \(\preceq\) APPROX_1), if for any given MAXSAT formula \(\varphi, \varphi' \subseteq \varphi\), and \(\alpha_{\varphi'} \in \text{MaxSAT}(\varphi')\), the optimum objective value of \(\varphi' \cup \text{APPROX}_1(\varphi, \varphi', \alpha_{\varphi'})\) is no larger than that of \(\varphi' \cup \text{APPROX}_2(\varphi, \varphi', \alpha_{\varphi'})\). More formally:
\[
\text{APPROX}_2 \preceq \text{APPROX}_1 \iff \forall \varphi, \varphi' \subseteq \varphi, \alpha_{\varphi'} \in \text{MaxSAT}(\varphi') : \max \alpha_{\varphi'} \leq \psi, \alpha) \leq \max \alpha_{\varphi'} \leq \psi, \alpha) \]
where \(\psi_1 = \text{APPROX}_1(\varphi, \varphi', \alpha_{\varphi'})\), \(\psi_2 = \text{APPROX}_2(\varphi, \varphi', \alpha_{\varphi'})\).

In Section 4.2 we introduce three different APPROX functions with increasing order of precision. While the more precise APPROX operators reduce the number of iterations in our algorithm, they are more expensive to compute.

Expanding relevant clauses via \text{REFIN}. The \text{REFIN} function finds a new set of clauses that are relevant to the queries, and adds them to the workset when the optimality check fails. To guarantee the termination of our approach, when the optimality check fails, \text{REFIN} should always return a nonempty set. We describe the details of various \text{REFIN} functions in Section 4.2.

4.2 Efficient Optimality Check via Distinct APPROX Functions

We introduce three different APPROX functions and their corresponding \text{REFIN} functions to construct efficient check functions. These three functions are the \text{ID-A} APPROX function, the \(\pi\)-APPROX function, and the \(\gamma\)-APPROX function. Each function is constructed by extending the previous one, and their precision order is:

\[
\text{ID-A} \preceq \pi \preceq \gamma
\]

The cost of executing each of these APPROX functions also increases with precision. After defining each function and proving that it satisfies the specification of an APPROX function, we discuss the efficiency and the precision of the optimality check using it.

4.2.1 The \text{ID-A} APPROX Function

The \text{ID-A} APPROX function is based on the following observation: for a Q-MAXSAT instance, the clauses not explored by Algorithm 1 can only affect the assignments to the queries via the clauses sharing variables with the workset. We refer to such unexplored clauses as \text{frontier clauses}. If all the frontier clauses are satisfied by the current partial model, or they cannot further improve the objective value of the workset, we can construct a model of the Q-MAXSAT instance from the current partial model. Based on these observations, the \text{ID-A} APPROX function constructs the summarization set by adding all the frontier clauses that are not satisfied by the current partial model.

To define \text{ID-A} APPROX, we first define what it means to say that a clause is satisfied by a partial model. A clause is satisfied by a partial model over a set of variables \(U\), if it is satisfied by all completions under that partial model. In other words:
\[
\alpha_{U} = \{1\} \iff \exists v \in U : \alpha(v) = 1
\]

Definition 6 (ID-A APPROX). Given formulae \(\varphi, \varphi' \subseteq \varphi\), and a partial model \(\alpha_{\varphi'} \in \text{MaxSAT}(\varphi')\), \text{ID-A} APPROX is defined as follows:

\[
\text{ID-A} \text{-APPROX}(\varphi, \varphi', \alpha_{\varphi'}) = \{(w, c) \mid \psi \in \varphi \wedge \varphi \cup \alpha_{\varphi'} \neq \emptyset \wedge \alpha_{\varphi'} \not\models c\}
\]

The corresponding \text{REFIN} function is:

\[
\text{REFIN}(\varphi, \varphi', \alpha_{\varphi'}) = \varphi'\cdot
\]

Example. Let \((\varphi, Q)\) be a Q-MAXSAT instance, where \(\varphi = ((2, x), (5, y) \vee y)\). Assume that the workset is \(\varphi' = ((2, x), (5, y) \vee y)\). By invoking a MAXSAT solver on \(\varphi'\), we get a model \(\alpha_{\varphi'} = [x \rightarrow 1, y \rightarrow 1]\). Both clauses in \(\varphi \setminus \varphi'\) contain \(y\), where \(\varphi' \setminus \varphi = \{(5, y) \vee z\}\). Since \(\varphi' \setminus \varphi = \{(1, \neg y)\}\). As the optimum objective values of both \(\varphi'\) and \(\varphi' \cup \psi\) are 7, we conclude \([x \rightarrow 1, y \rightarrow 1]\) is a model of the given Q-MAXSAT instance. Indeed, its completion \([x \rightarrow 1, y \rightarrow 1, z \rightarrow 0]\) is a model of the MAXSAT formula \(\varphi\).

Theorem 7 (Soundness of ID-A APPROX). \text{ID-A} \text{-APPROX}(\varphi, \varphi', \alpha_{\varphi'}) summarizes the effect of \(\varphi \setminus \varphi'\) with respect to \(\alpha_{\varphi'}\).

Proof. Let \(\psi = \text{ID-A} \text{-APPROX}(\varphi, \varphi', \alpha_{\varphi'})\). We show that \text{ID-A} APPROX satisfies the specification of an APPROX function in Definition 2 by proving the inequality below:

\[
\max eval(\varphi' \cup \psi, \alpha) + \max eval(\varphi' \setminus \alpha_{\varphi'}, \alpha) \geq \max eval(\varphi, \alpha)
\]

We use \(\psi_1\) to represent the set of frontier clauses, and \(\varphi_2\) to represent the rest of the clauses in \(\varphi \setminus \varphi'\):

\[
\psi_1 = \{(w, c) \mid \varphi \setminus \varphi' \wedge \alpha_{\varphi'} \neq \emptyset\}
\]

\[
\varphi_2 = \{(w, c) \mid \varphi \setminus \varphi' \wedge \alpha_{\varphi'} = \emptyset\}
\]

We further split the set of frontier clauses \(\psi_1\) into two sets:

\[
\psi_1 = \{(w, c) \mid \alpha_{\varphi'} \not\models c\}, \psi_1' = \{(w, c) \in \varphi \setminus c \mid \alpha_{\varphi'} \models c\}
\]

\(\psi_1\) is effectively what is returned by \text{ID-A} APPROX(\varphi, \varphi', \alpha_{\varphi'}) We first prove the following claim:

\[
\forall \alpha, eval(\varphi_1''[\alpha_{\varphi'}] \cup \varphi_2, \alpha) \geq eval(\varphi''_1 \cup \varphi_2, \alpha)
\]

We have:

\[
eval(\varphi''_1[\alpha_{\varphi'}] \cup \varphi_2, \alpha)
\]

\[
= eval(\varphi''_1[\alpha_{\varphi'}], \alpha) + eval(\varphi_2, \alpha)\text{ [from Prop. 3]}
\]

\[
= max eval(\varphi''_1, \alpha) + eval(\varphi_2, \alpha)\text{ [since } \forall (w, c) \in \varphi''_1, \alpha_{\varphi'} \models c\]
\]

\[
\geq eval(\varphi''_1, \alpha) + eval(\varphi_2, \alpha)
\]

\[
= eval(\varphi''_1 \cup \varphi_2, \alpha)\text{ [from Prop. 3]}
\]
Now we prove the theorem. We have:

$$\max_\alpha \text{eval}(\varphi \cup \psi, \alpha) + \max_\alpha \text{eval}(\varphi \setminus \varphi', \alpha)$$

$$= \max_\alpha \text{eval}(\varphi' \cap \varphi', \alpha) + \max_\alpha \text{eval}(\varphi' \setminus \varphi', \alpha)$$

since \(\psi = \text{ID-APPROX}(\varphi, \psi', \alpha), \) see Defn. 6

$$= \max_\alpha \text{eval}(\varphi' \cup \varphi_1, \alpha) +$$

$$\max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi' \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi' \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi_1 \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi_1 \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi_1 \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi_1 \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi_1 \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

$$\geq \max_\alpha \text{eval}(\varphi_1 \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

Thus, we conclude that \(\alpha_Q\) is a solution of Q-MAXSAT instance \((\varphi, x, y)\).

Moreover, 

$$\max_\alpha \text{eval}(\varphi_1 \cup \varphi_1, \alpha) + \max_\alpha \text{eval}(\varphi'_1 \cup \varphi_1' \cup \varphi_2' \cup \varphi_2, \alpha)$$

is a multi-linear polynomial, which maps the assignment of a set of Boolean variables to a real number:

$$f(v_1, \ldots, v_n) = a_0 + \sum_{i=1}^m a_i \prod_{j=1}^p v_j, \quad \text{where } a_i \in \mathbb{R}^+.$$
Any MAXSAT formula can be converted into a pseudo-Boolean function. The conversion \( \langle \cdot \rangle \) is as follows:

\[
\langle \{s_1, \ldots, s_n\} \rangle = \sum_{i=1}^{n} s_i \langle s_i \rangle = f(s_i) \langle s_i \rangle (s_{\text{nd}(s_i)})
\]

\[
\langle \bigvee_{i=1}^{m} v_i \vee \bigwedge_{j=1}^{l} \neg v'_j \rangle = \bigwedge_{i=1}^{m} (1 - v_i) \star \bigwedge_{j=1}^{l} v'_j
\]

\[
(1) = 0
\]

\[
(0) = 1
\]

For example, the MAXSAT formula \( \{ (5, \neg w \vee x), (3, y \vee z) \} \) is converted into function \( 5 - 5w(1-x) + 3(1-y)(1-z) \).

We use \( \text{eval}(f, \alpha) \) to denote the evaluation of pseudo-Boolean function \( f \) under model \( \alpha \). From the conversion, we can conclude that \( \forall \varepsilon, \alpha. \text{eval}(\varepsilon, \alpha) = \text{eval}(\varepsilon, \alpha) \).

We now prove the claim \((1)\) under the third case above. We rewrite \( c = \bigvee_{i=1}^{m} v_i \vee \bigwedge_{j=1}^{l} \neg v'_j \) as follows:

\[
c = \bigvee_{x \in V_1} x \vee \bigvee_{x' \in V_2} \neg x' \vee \bigvee_{y \in V_3} y \vee \bigvee_{y' \in V_4} \neg y'
\]

where \( V_1 = V \cap \{v_1, \ldots, v_m\}, V_2 = V \cap \{v'_1, \ldots, v'_m\}, V_3 = V \cap \{v_1, \ldots, v_m\} \) and \( V_4 = V \cap \{v'_1, \ldots, v'_m\} \). Let \( c_1 = \bigvee_{x \in V_1} x \vee \bigvee_{x' \in V_2} \neg x' \) and \( c_2 = \bigvee_{y \in V_3} y \vee \bigvee_{y' \in V_4} \neg y' \). Then, we have \( S_1 = \{w(c_1)\} \) and \( S_2 = \{w(c_2)\} \).

It suffices to prove that \( (S_1) + (S_2) \geq 0 \). We have:

\[
(S_1) + (S_2) = w - w \bigvee_{x \in V_1} (1 - x) \bigvee_{x' \in V_2} x' \bigvee_{y \in V_3} (1 - y) \bigvee_{y' \in V_4} y'
\]

\[
= w(1 - \bigvee_{x \in V_1} (1 - x) \bigvee_{x' \in V_2} x' \bigvee_{y \in V_3} (1 - y) \bigvee_{y' \in V_4} y')
\]

Since all variables are Boolean variables, we have

\[
1 - \bigvee_{x \in V_1} (1 - x) \bigvee_{x' \in V_2} x' \bigvee_{y \in V_3} (1 - y) \bigvee_{y' \in V_4} y' \geq 0.
\]

From Lemma 11, Propositions 3 and 4, we can also conclude that

\[
\max_{\alpha} \text{eval}(\varepsilon, \alpha) = \max_{\alpha} \text{eval}(\varepsilon, \alpha)
\]

where \( \pi-\text{DECOMP}(\varepsilon, V) = (\varepsilon_1, \varepsilon_2) \).

We now use the lemma to prove that \( \pi-\text{APPROX} \) is a \text{APPROX} function as defined in Definition 2.

**Theorem 12 (Soundness of \( \pi-\text{APPROX} \)).** \( \pi-\text{APPROX}(\varepsilon, \varepsilon', \alpha, \varepsilon') \) summarizes the effect of \( \varepsilon \setminus \varepsilon' \) with respect to \( \alpha \).

**Proof.** Let \( \varepsilon = \pi-\text{APPROX}(\varepsilon, \varepsilon', \alpha, \varepsilon) \). We show that \( \pi-\text{APPROX} \) satisfies the specification of an APPROX function in Definition 2 by proving the inequality below:

\[
\max_{\alpha} \text{eval}(\varepsilon \cup \varepsilon', \alpha) + \max_{\alpha} \text{eval}(\varepsilon \setminus \varepsilon', \alpha) \geq \max_{\alpha} \text{eval}(\varepsilon, \alpha)
\]

Without loss of generality, we assume that \( \varepsilon \) does not contain any clause \((w, c)\) where \( c = 0 \) or \( c = 1 \). Otherwise, we can rewrite the inequality above by removing these clauses from \( \varepsilon \), \( \varepsilon' \), and \( \varepsilon' \).

As in the soundness proof for ID-APPROX, we define \( \varepsilon_1, \varepsilon_2, \varepsilon_1', \) and \( \varepsilon_2' \) as follows:

\[
\varepsilon_1 = \{w(c) \in \varepsilon \setminus \varepsilon' \mid \text{Var}((w, c)) \cap \text{Var}(\varepsilon' \setminus \varepsilon) \neq \emptyset\}
\]

\[
\varepsilon_2 = \{w(c) \in \varepsilon \setminus \varepsilon' \mid \text{Var}((w, c)) \cap \text{Var}(\varepsilon' \setminus \varepsilon') = \emptyset\}
\]

\[
\varepsilon_1' = \{w(c) \in \varepsilon_1 \mid \alpha_v \neq c\}
\]

\[
\varepsilon_2' = \{w(c) \in \varepsilon_2 \mid \alpha_v \neq c\}
\]

We have:

\[
\max_{\alpha} \text{eval}(\varepsilon \cup \varepsilon', \alpha) + \max_{\alpha} \text{eval}(\varepsilon \setminus \varepsilon', \alpha)
\]

\[
= \max_{\alpha} \text{eval}(\varepsilon_1 \cup \varepsilon_2, \alpha) + \max_{\alpha} \text{eval}(\varepsilon_1' \cup \varepsilon_2', \alpha)
\]

\[
= \max_{\alpha} \text{eval}(\varepsilon \cup \varepsilon', \alpha) + \max_{\alpha} \text{eval}(\varepsilon_1' \cup \varepsilon_2', \alpha)
\]

\[
= \max_{\alpha} \text{eval}(\varepsilon \cup \varepsilon_1' \cup \varepsilon_2', \alpha)
\]

\[
\geq \max_{\alpha} \text{eval}(\varepsilon \cup \varepsilon_1' \cup \varepsilon_2', \alpha)
\]

\[
= \max_{\alpha} \text{eval}(\varepsilon, \alpha)
\]

**Discussion.** Similar to ID-APPROX, \( \pi-\text{APPROX} \) generates the summarization set from the frontier clauses that are not satisfied by the current solution. Consequently, the performance of the MAXSAT invocation in the optimality check is similar for both \( \pi-\text{APPROX} \) and ID-APPROX. \( \pi-\text{APPROX} \) further improves the precision of the check by applying the \text{retain} operator to strengthen the clauses generated. The \text{retain} operator does incur modest overheads when computing \( \pi-\text{APPROX} \). In practice, however, we find that the additional precision provided by \( \pi-\text{APPROX} \) function improves the overall performance of the algorithm by terminating the iterative process earlier.
4.2.3 The $\gamma$-APPROX Function

While both the ID-APPROX function and the $\pi$-APPROX function only consider information on the frontier clauses, $\gamma$-APPROX further improves precision by considering information from non-frontier clauses. Similar to the $\pi$-APPROX function, $\gamma$-APPROX also generates the summarization set by applying the retain function on the frontier clauses violated by the current partial model. The key improvement is that $\gamma$-APPROX tries to reduce the weights of generated clauses by exploiting information from the non-frontier clauses.

Before defining the $\gamma$-APPROX function, we first introduce some definitions:

$$PV(n) = \{v_1, ..., v_m\}, \ PV(0) = \emptyset, \ NV(n) = \{v_1, ..., v_m\}, \ NV(0) = \emptyset$$

 Define:

$$\text{link}(v, u, \varphi) \Rightarrow \exists (v, c) \in \varphi : v \in NV(c) \land u \in PV(c)$$

$$tReachable(v, \varphi) = \{v\} \cup \{u | (v, u) \in R^+\}$$

$$fReachable(v, \varphi) = \{v\} \cup \{u | (v, u) \in \text{tReachable}(u, \varphi)\}$$

$$tPenalty(v, \varphi) = \sum \{w | (v, u, \varphi) \in \text{fReachable}(u, \varphi) \land \exists \varphi \land \varphi \neq \emptyset$$

$$fPenalty(v, \varphi) = \sum \{w | (v, u, \varphi) \in \text{fReachable}(u, \varphi) \land \exists \varphi \land \varphi \neq \emptyset \}$$

Intuitively, $t\text{Penalty}(v, \varphi)$ overestimates the penalty incurred on the objective value of $\varphi$ by setting variable $v$ to true, while $f\text{Penalty}(v, \varphi)$ overestimates the penalty incurred on the objective value of $\varphi$ by setting variable $v$ to false.

We next introduce the $\gamma$-APPROX approximation function.

**Definition 13 (\gamma-APPROX).** Given $\varphi, \varphi' \subseteq \varphi$, and a partial model $\alpha_{\varphi'} \in \text{MaxSAT}(\varphi')$, $\gamma$-APPROX is defined as below:

$$\gamma\text{-APPROX}(\varphi, \varphi', \alpha_{\varphi'}) = \left\{ (w, \alpha) | (w, c) \in \varphi \land \text{Var}(\{w, c\}) \cap \text{Var}(\varphi') \neq \emptyset \land \alpha_{\varphi'} \neq c \land c' = \text{retain}(c, \varphi, \varphi') \wedge c' = \text{reduce}(w, c, \varphi, \alpha_{\varphi'}) \right\}$$

We next define the reduce function. Let $c'' = \text{retain}(c, \varphi, \varphi')$. Then, function reduce is defined as follows:

1. If $c'' = 0$, reduce($w$, $c$, $\varphi$, $\alpha_{\varphi''}$) = $w$.
2. If $PV(c'') \neq \emptyset$ and $NV(c'') = \emptyset$, reduce($w$, $c$, $\varphi$, $\alpha_{\varphi''}$) = $min(\text{min}_{\text{eval}}(w, c, \varphi, \alpha_{\varphi''}) + \text{Penalty}(v, \varphi, \alpha_{\varphi''}))$.
3. If $PV(c'') = \emptyset$ and $NV(c'') \neq \emptyset$, reduce($w$, $c$, $\varphi$, $\alpha_{\varphi''}$) = $min(\text{min}_{\text{eval}}(w, c, \varphi, \alpha_{\varphi''}) + \text{Penalty}(v, \varphi, \alpha_{\varphi''}))$.
4. If $PV(c'') = \emptyset$ and $NV(c'') = \emptyset$, reduce($w$, $c$, $\varphi$, $\alpha_{\varphi''}$) = $min(\text{min}_{\text{eval}}(w, c, \varphi, \alpha_{\varphi''}) + \text{Penalty}(v, \varphi, \alpha_{\varphi''}) + \text{Penalty}(v, \varphi, \alpha_{\varphi''}))$.

The corresponding refine function is:

$$\gamma\text{REFINE}(\varphi, \varphi', \alpha_{\varphi''}) = \left\{ (w, c) | (w, c) \in \varphi \land \varphi \land \text{reduce}(w, c, \varphi, \alpha_{\varphi''}) \land \varphi \in \varphi' \right\}$$

**Example.** Let $(\varphi, Q)$ where $\varphi = \{(x, z), (x, \neg v \lor y), (100, \neg v \lor z), (1, \neg z)\}$ and $Q = \{(x, y)\}$ be a Q-MAXSAT instance. Suppose that the workset is $\varphi' = \{x \triangleright 1, y \triangleright 1\}$, since it is a partial model. $\gamma$-APPROX solver takes $\varphi' = \{x \triangleright 1, y \triangleright 1\}$ as a partial model. $\alpha_{\varphi'}$ is a partial model of $\varphi'$ and $\varphi''$ is a partial model of $\varphi$. In the other hand, $\gamma$-APPROX will fail the check due to the high weight of the clause generated via retain.

We explain how reduce works on this example. We first generate $\varphi'' = \text{retain}(-v \lor z, \{x, y\}, \{x, y\} = z).$ Then, we compute $(\varphi \setminus \varphi'_{\alpha_{\varphi}}) = \{(100, \neg y \lor z), (1, \neg z)\}$, $(x \triangleright 1, y \triangleright 1) = \{100, (z), (1, \neg z)\}$, and $\text{reduce}(\varphi, \varphi', \alpha_{\varphi''})$. As a result, $t\text{Penalty}(v, \varphi, \alpha_{\varphi''}) = 1$, which is lower than the weight of the summarization clause $(100, \neg y)$. Hence, reduce returns 1 as the new weight for the summarization clause.

To prove that $\gamma$-APPROX satisfies the specification of AP APPROX in Definition 2, we first prove two lemmas.

**Lemma 14.** Given $(w, c) \in \varphi, v \in PV(c)$, and $t\text{Penalty}(v, \varphi) < w$, we construct $\varphi'$ as follows:

$$\varphi' = \varphi \setminus \{w, c\} \cup \{w', c\}, \text{where } w' = t\text{Penalty}(v, \varphi).$$

Then we have

$$\max\text{eval}(\varphi, \alpha) = \max\text{eval}(\varphi', \alpha') + w - w'.$$

**Proof.** We first prove a claim:

$$\exists (w, c) | (w, c) \in \varphi \land \alpha \vdash c.$$  

We prove this by contradiction. Suppose we can only find a model $\alpha$ that maximizes the objective value of $\varphi$, such that $\alpha \vdash c$. We can construct a model $\alpha' \vdash c$ in the following way:

$$\alpha' = \{ u | u \notin t\text{Reachable}(v, \varphi'), 1 \text{ otherwise} \}$$

Based on the definition of $t\text{Penalty}$, we have

$$\text{eval}(\varphi, \alpha') \geq \text{eval}(\varphi, \alpha) - t\text{Penalty}(v, \varphi).$$

Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w$. Since $w \geq t\text{Penalty}(v, \varphi) + w.}$

Based on the on the two claims, we can find $\alpha' \vdash w$. Hence, we have

$$\max\text{eval}(\varphi, \alpha') = \max\text{eval}(\varphi, \alpha) + w - w'.$$
Theorem 16 (γ-APPROX). γ-APPROX(φ, φ', αΨ) summarizes the effect of φ \ φ' with respect to αΨ.

Proof. Let ψ = γ-APPROX(φ, φ', αΨ). We show that γ-APPROX satisfies the specification of an APPROX function in Definition 2 by proving the inequality below:

\[ \max \text{eval}(\phi \cup \psi, \alpha) + \max \text{eval}(\phi \setminus \phi'[\alpha], \alpha) \geq \max \text{eval}(\phi, \alpha) \]

We define φ1 and φ2 as follows:

\[ \begin{align*}
\phi_1 &= \{ (w, c) \mid (w, c) \in (\phi \setminus \phi') \cap \text{Var}(\{\{w, c\}\} \cap \text{Var}(\phi') \neq \emptyset) \}, \\
\phi_2 &= \{ (w, c) \mid (w, c) \in (\phi \setminus \phi') \cap \text{Var}(\{\{w, c\}\} \cap \text{Var}(\phi') = \emptyset) \}.
\end{align*} \]

ϕ1 is the set of frontier clauses and ϕ2 contains rest of the clauses in ϕ \ ϕ'. We further split ϕ1 into the following disjoint sets:

\[ \begin{align*}
\phi_1^1 &= \{ (w, c) \mid (w, c) \in \phi_1 \land \exists v \in PV(\phi) : w = tPenalty(v, \phi \setminus \phi'[\alpha]) \land \text{reduce}(w, c, \phi, \phi', \alpha) = w' \}, \\
\phi_1^2 &= \{ (w, c) \mid (w, c) \in \phi_1 \land \exists v \in NV(\phi) : w = fPenalty(v, \phi \setminus \phi'[\alpha]) \land \text{reduce}(w, c, \phi, \phi', \alpha) = w' \}, \\
\phi_1^3 &= \phi_1 \setminus (\phi_1^1 \cup \phi_1^2 \cup \phi_1^3).
\end{align*} \]

Effectively, ϕ1^1, ϕ1^2, and ϕ1^3 contain all the clauses being strengthened in γ-APPROX: ϕ1^1 contains all the clauses whose weights are not reduced; ϕ1^2 contains all the clauses whose weights are reduced through positive literals in them; ϕ1^3 contains all the clauses whose weights are reduced through negative literals in them.

Further, we define ϕ2^1 and ϕ2^2 as below:

\[ \begin{align*}
\phi_2^1 &= \{ (w, c) \mid (w, c) \in \phi_2 \land \exists v \in PV(\phi) : w = tPenalty(v, \phi \setminus \phi'[\alpha]) \land \text{reduce}(w, c, \phi, \phi', \alpha) = w' \}, \\
\phi_2^2 &= \{ (w, c) \mid (w, c) \in \phi_2 \land \exists v \in NV(\phi) : w = fPenalty(v, \phi \setminus \phi'[\alpha]) \land \text{reduce}(w, c, \phi, \phi', \alpha) = w' \}, \\
\phi_2^3 &= \phi_2 \setminus (\phi_2^1 \cup \phi_2^2 \cup \phi_2^3).
\end{align*} \]

Using Lemmas 14 and 15, we prove the following claim (1):

\[ \max \text{eval}(\phi \setminus \phi'[\alpha], \alpha) \geq \max \text{eval}(\phi_1^1 \cup \phi_2^1 \cup \phi_1^2 \cup \phi_2^2 \cup \phi_2^3, \alpha) \]

Similar to the soundness proof for π-DECOMP, we show:

\[ \begin{align*}
\pi-\text{DECOMP}(\phi' \cup \phi_1^1 \cup \phi_2^1 \cup \phi_1^2 \cup \phi_2^2, \text{Var}(\phi')) &= (\phi' \cup \psi, \phi_1^1 \cup \phi_2^1 \cup \phi_1^2 \cup \phi_2^2), \\
\text{Var}(\phi') &= \text{Var}(\phi \setminus \phi'[\alpha]), \\
\text{Var}(\phi) &= \text{Var}(\phi \setminus \phi'[\alpha]).
\end{align*} \]

Following Lemma 11, we can derive the following claim (2):

\[ \max \text{eval}(\phi' \cup \psi, \alpha) \geq \max \text{eval}(\phi' \cup \phi_1^1 \cup \phi_2^1 \cup \phi_1^2 \cup \phi_2^2, \alpha). \]

By contradiction, we can show the following claim (3) holds:

\[ \forall \phi, \forall (w, c) \in \phi, w' \leq w, \max \text{eval}(\phi, \alpha) \leq \max \text{eval}(\phi \setminus \{(w, c)\} \cup \{(w', c)\}, \alpha) + w - w'. \]

Combining Claim (1), (2), and (3), we have

\[ \begin{align*}
\max \text{eval}(\phi' \cup \psi, \alpha) + \max \text{eval}(\phi \setminus \phi'[\alpha], \alpha) &\geq \max \text{eval}(\phi' \cup \phi_1^1 \cup \phi_2^1 \cup \phi_1^2 \cup \phi_2^2, \alpha) + \max \text{eval}(\phi \setminus \phi'[\alpha], \alpha).
\end{align*} \]
<table>
<thead>
<tr>
<th>benchmark</th>
<th>brief description</th>
<th># classes</th>
<th># methods</th>
<th>bytecode (KB)</th>
<th>source (KLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp</td>
<td>Apache FTP server</td>
<td>93</td>
<td>414</td>
<td>471</td>
<td>2,206</td>
</tr>
<tr>
<td>hedic</td>
<td>web crawler from ETH</td>
<td>44</td>
<td>353</td>
<td>230</td>
<td>2,134</td>
</tr>
<tr>
<td>weblech</td>
<td>website download/mirror tool</td>
<td>11</td>
<td>576</td>
<td>78</td>
<td>3,326</td>
</tr>
<tr>
<td>antlr</td>
<td>generates parsers and lexical analyzers</td>
<td>111</td>
<td>350</td>
<td>1,150</td>
<td>2,370</td>
</tr>
<tr>
<td>avrorra</td>
<td>AVR microcontroller simulator</td>
<td>1,158</td>
<td>1,544</td>
<td>4,234</td>
<td>6,247</td>
</tr>
<tr>
<td>chart</td>
<td>plots graphs and render them as PDF</td>
<td>192</td>
<td>750</td>
<td>1,516</td>
<td>4,661</td>
</tr>
<tr>
<td>luindex</td>
<td>document indexing tool</td>
<td>206</td>
<td>619</td>
<td>1,390</td>
<td>3,732</td>
</tr>
<tr>
<td>lusearch</td>
<td>text searching tool</td>
<td>219</td>
<td>640</td>
<td>1,399</td>
<td>3,923</td>
</tr>
<tr>
<td>xalan</td>
<td>XML to HTML transforming tool</td>
<td>423</td>
<td>897</td>
<td>3,247</td>
<td>6,044</td>
</tr>
</tbody>
</table>

Table 1: Characteristics of the benchmark programs. Columns “total” and “app” are with and without counting JDK library code, respectively.

<table>
<thead>
<tr>
<th></th>
<th># queries</th>
<th># variables</th>
<th># clauses</th>
<th># queries</th>
<th># variables</th>
<th># clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp</td>
<td>338</td>
<td>1.2M</td>
<td>1.4M</td>
<td>55</td>
<td>2.3M</td>
<td>3M</td>
</tr>
<tr>
<td>hedic</td>
<td>203</td>
<td>0.8M</td>
<td>0.9M</td>
<td>36</td>
<td>3.8M</td>
<td>4.8M</td>
</tr>
<tr>
<td>weblech</td>
<td>7</td>
<td>0.5M</td>
<td>0.9M</td>
<td>25</td>
<td>5.8M</td>
<td>8.4M</td>
</tr>
<tr>
<td>antlr</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-13</td>
<td>8.7M</td>
<td>13M</td>
</tr>
<tr>
<td>avrorra</td>
<td>803</td>
<td>0.7M</td>
<td>1.5M</td>
<td>151</td>
<td>11.7M</td>
<td>16.3M</td>
</tr>
<tr>
<td>chart</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-94</td>
<td>16M</td>
<td>22.3M</td>
</tr>
<tr>
<td>luindex</td>
<td>3,444</td>
<td>0.6M</td>
<td>1.1M</td>
<td>109</td>
<td>8.5M</td>
<td>11.9M</td>
</tr>
<tr>
<td>lusearch</td>
<td>206</td>
<td>0.5M</td>
<td>1M</td>
<td>248</td>
<td>7.8M</td>
<td>10.9M</td>
</tr>
<tr>
<td>xalan</td>
<td>11,410</td>
<td>2.6M</td>
<td>4.9M</td>
<td>754</td>
<td>12.4M</td>
<td>18.7M</td>
</tr>
</tbody>
</table>

Table 2: Number of queries, variables, and clauses in the MAXSAT instances generated by running the dataarce analysis and the pointer analysis on each benchmark program. The dataarce analysis has no queries on antlr and chart as they are sequential programs.

Instances from information retrieval. These instances are generated from problems in information retrieval. In particular, we consider problems in relational learning where the goal is to infer new relationships that are likely to be described according to certain rules. Relational solvers such as Alchemy [29] and Tuffy [45] solve such problems by solving a system of weighted constraints generated from the data and the rules. The weight of each clause represents the confidence in each inference rule. We consider Q-MAXSAT instances generated from three standard relational learning applications, described next, whose datasets are publicly available [1, 2, 7].

The first application is an advisor recommendation system (AR), which recommends advisors for first-year graduate students. The dataset for this application is generated from the AI genealogy project [1] and DBLP [2]. The query specifies whether a professor is a suitable advisor for a student. The Q-MAXSAT instance generated consists of 10 queries, 0.3 million variables, and 7.9 million clauses.

The second application is Entity Resolution (ER), which identifies duplicate entities in a database. The dataset is generated from the Cora bibliographic dataset [7]. The queries in this application specify whether two entities in the dataset are the same. The Q-MAXSAT instance generated consists of 25 queries, 3 thousand variables, and 0.1 million clauses.

The third application is Information Extraction (IE), which extracts information from text or semi-structured sources. The dataset is also generated from the Cora dataset. The queries in this application specify extractions of the author, title, and venue of a publication. The Q-MAXSAT instance generated consists of 6 queries, 47 thousand variables, and 0.9 million clauses.

5.2 Evaluation Result

To evaluate the benefits of being query-guided, we measured the running time and memory consumption of PILOT. We used MiFuMaX as the baseline by running it on MAXSAT instances generated from our Q-MAXSAT instances by removing queries. To better understand the benefits of being query-guided, we also study the size of clauses posed to the underlying MAXSAT solver in the last iteration of PILOT, and the corresponding solver running time.

Further, to understand the cost of resolving each query, we pick one Q-MAXSAT instance from both domains and evaluate the performance of PILOT by resolving each query separately.

Finally, we study the sensitivity of PILOT’s performance to the underlying MAXSAT solver by running PILOT using three different solvers besides MiFuMaX.

Performance of our approach vs. baseline approach. Table 3 summarizes our evaluation results on Q-MAXSAT instances generated from both domains.

Our approach successfully terminated on all instances and significantly outperformed the baseline approach in memory consumption on all instances, while the baseline only finished on twelve of the twenty instances in total. On the eight largest instances, the baseline approach either ran out of memory (exceeded 3 GB), or timed out (exceeded one hour).

Column ‘peak memory’ shows the peak memory consumption of our approach and the baseline approach on all instances. For the instances on which both approaches terminated, our approach consumed 55.7% less memory compared to the baseline. Moreover, for large instances containing more than two million clauses, this number further improves to 71.6%.

We next compare the running time of both approaches under the ‘running time’ column. For most instances on which both approaches terminated, our approach does not outperform the baseline in running time. One exception is IE where our approach terminated in 2 seconds while the baseline approach spent 2,760 seconds, yielding a 1,380X speedup. Due to the iterative nature of PILOT, on simple instances where the baseline approach runs efficiently, the overall running time of PILOT can exceed that of the baseline approach. As column ‘iteration’ shows, PILOT takes multiple iterations on most instances. However, for challenging instances like IE and other instances where the baseline approach failed to finish, PILOT yields significant improvement in running time.

To better understand the gains of being query-guided, we study the number of clauses PILOT explored under the ‘problem size’ column. Column ‘final’ shows the number of clauses PILOT posed to the underlying solver in the last iteration, while column ‘max’ shows the total number of clauses in the Q-MAXSAT instances. On average, PILOT only explored 35.2% clauses in each instance. Moreover, for large instances containing over two million clauses, this number improves to 19.4%. Being query-guided allows our approach to lazily explore the problem and only solve the clauses that are relevant to the queries. As a result, our approach significantly outperforms the baseline approach.

The benefit of being query-guided is also reflected by the running time of the underlying MAXSAT solver in our approach. Column ‘final solver time’ shows the running time of the underlying
Table 3: Performance of our PILOT and the baseline approach (BASELINE). In all experiments, we used a memory limit of 3 GB and a time limit of one hour for each invocation of the MAXSAT solver in both approaches. Experiments that timed out exceeded either of these limits.

For instances generated from the pointer analysis, when each query is resolved separately, except for one outlier, PILOT uses less than 30% of the memory it needs when all queries are resolved together. The result shows that each query is decided by different clauses in the program analysis instances. By resolving them separately, we further improve the performance of PILOT. This is in line with locality in program analysis, which is exploited by various query-driven approaches in program analysis.

To study the effect of the underlying MAXSAT solver, we evaluated the performance of PILOT using CCLS2akms, Eva500, MaxHS, and wpm-2014.co as the underlying solver. CCLS2akms and Eva500 were the winners in the MAXSAT’14 competition for random instances and industrial instances, respectively, while MaxHS ranked third for crafted instances (neither of the solvers performing better than MaxHS is publicly available). We used each solver itself as the baseline for applications generated from the pointer analysis and (b) AR. The dotted line represents the memory consumption of PILOT when it resolves all queries together.
comparison. In the evaluation, we used two instances from different domains, one generated from running the pointer analysis on avrora, and the other generated from AR.

Table 4 shows the running time and memory consumption of PILOT and the baseline approach under each setting. For convenience, we also include the result with MiFuMaX as the underlying solver in the table. As the table shows, except for the run with CCLS2akms on the pointer analysis instance, PILOT terminated under all the other settings, while none of the baseline approaches finished on any instance. This shows that our approach consistently gives improved performance regardless of the underlying MAXSAT solver it invokes.

6. Related Work

The MAXSAT problem is one of the optimization extensions of the SAT problem. The original form of the MAXSAT problem does not allow any hard clauses, and requires each soft clause to have a unit weight. A model of this problem is a complete assignment to the variables that maximizes the number of satisfied clauses. Two important variations of this problem are the weighted MAXSAT problem and the partial MAXSAT problem. The weighted MAXSAT problem allows each soft clause to have an arbitrary positive weight instead of limiting it to a unit weight; a model of this problem is a complete assignment that maximizes the sum of the weights of satisfied soft clauses. The partial MAXSAT problem allows hard clauses mixed with soft clauses; a model of this problem is a complete assignment that satisfies all hard clauses and maximizes the number of satisfied soft clauses. The combination of both these variations yields the weighted partial MAXSAT problem, which is commonly referred to simply as the MAXSAT problem, and is the problem addressed in our work.

MAXSAT solvers are broadly classified into approximate and exact. Approximate solvers are efficient but do not guarantee optimality (i.e., may not maximize the objective value) [38] or even soundness (i.e., may falsify a hard clause) [27], although it is common to provide error bounds on optimality [54]. Our solver, on the other hand, is exact as it guarantees optimality and soundness.

There are a number of different approaches for exact MAXSAT solving, including branch-and-bound based, satisfiability-based, unsatisfiability-based, and their combinations [4, 8, 20, 22, 37, 39, 41, 42, 44]. The most successful of these on real-world instances, as witnessed in annual Max-SAT evaluations [3], perform iterative solving using a SAT solver as an oracle in each iteration [4, 41]. Such solvers differ primarily in how they estimate the optimal cost (e.g., linear or binary search), and the kind of information that they use to estimate the cost (e.g., cores, the structure of cores, or satisfying assignments). Many algorithms have been proposed that perform search on either upper bound or lower bound of the optimal cost [4, 41, 42, 44]. Some algorithms efficiently perform a combined search over two bounds [20, 22]. A drawback of the most sophisticated combined search algorithms is that they modify the formula using expensive Pseudo Boolean (PB) constraints that increase the size of the formula and, potentially, slow down the solver. A recent promising approach [8] avoids this problem by using succinct formula transformations that do not use PB constraints and can be applied incrementally.

Our approach is similar in spirit to the above exact approaches in aspects such as iterative solving and optimal cost estimation. But we solve a new and fundamentally different optimization problem Q-MAXSAT, which enables our approach to be demand-driven, unlike existing approaches. In particular, it enables our approach to entirely avoid exploring vast parts of a given, very large MAXSAT formula that are irrelevant to deciding the values of a (small set of) queries in some model of the original MAXSAT formula. For this purpose, our approach invokes an off-the-shelf exact MAXSAT solver on small sub-formulae of a much larger MAXSAT formula. Our approach is thus also capable of leveraging advances in solvers for the MAXSAT problem. Conversely, it would be interesting to explore how to integrate our query-guided approach into search algorithms of existing MAXSAT solvers.

The MAXSAT problem has also been addressed in the context of probabilistic logics for information retrieval [15], such as PSLs (Probabilistic Soft Logics) [28] and MLNs (Markov Logic Networks) [48]. These logics seek to reason efficiently with very large knowledge bases (KBs) of imperfect information. Examples of such KBs are the AR, ER, and IE applications in our empirical evaluation (Section 5). In particular, a fully grounded formula in these logics is a MAXSAT formula, and finding a model of this formula corresponds to solving the Maximum-a-Posteriori (MAP) inference problem in those logics [11, 29, 45, 46, 49]. A query-guided approach has been proposed for this problem [55] with the same motivation as ours, namely, to reason about very large MAXSAT formulae. However, this approach as well as all other (non-query-guided) approaches in the literature on these probabilistic logics sacrifice optimality as well as soundness.

In contrast, query-guided approaches have witnessed tremendous success in the domain of program reasoning. For instances, program slicing [21] is a popular technique to scale program analyses by pruning away the program statements which do not affect an assertion (query) of interest. Likewise, counter-example guided abstraction refinement (CEGAR) based model checkers [6, 12, 19] and refinement-based pointer analyzers [18, 34, 50, 51] compute a cheap abstraction which is precise enough to answer a given query. However, the vast majority of these approaches target constraint satisfaction problems (i.e., problems with only hard constraints) as opposed to constraint optimization problems (i.e., problems with mixed hard and soft constraints). To our knowledge, our approach is the first to realize the benefits of query-guided reasoning for constraint optimization problems—problems that are becoming increasingly common in domains such as program reasoning [9, 30, 31, 33, 58].

7. Conclusion

We introduced a new optimization problem Q-MAXSAT that extends the well-known MAXSAT problem with queries. The Q-MAXSAT problem is motivated by the fact that many real-world MAXSAT problems pose scalability challenges to MAXSAT solvers, and the observation that many such problems are naturally equipped with queries. We proposed efficient exact algorithms for solving Q-MAXSAT instances. The algorithms lazily construct small MAXSAT sub-problems that are relevant to answering the given queries in a much larger MAXSAT problem. We implemented our Q-MAXSAT solver in a tool PILOT that uses off-the-shelf MAXSAT solvers to efficiently solve such sub-problems. We demonstrated the effectiveness of PILOT in practice on a diverse set of 19 real-world Q-MAXSAT instances ranging in size from 100 thousand to 22 million clauses. On these instances, PILOT used only 285 MB of memory on average and terminated in 107 seconds on average, whereas conventional MAXSAT solvers timed out for eight of the instances.

Acknowledgments

We thank the referees for useful feedback. This work was supported by DARPA under agreement #FA8750-15-2-0009, NSF awards #1253867 and #1526270, and a Facebook Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.


