Algo oct_22

1. Introduction

Suppose we are given a problem

$$\exists_X \phi(X,Y) \wedge \alpha(Y) = [\exists_X \phi(X,Y)] \wedge \alpha(Y) \qquad (1.1)$$

Where ϕ and α are in CNF, and X and Y are disjoint sets of variables.

It shall later be important to have X variables in clauses being quantified, and hence, we mark ϕ (all of whose clauses depend on X) separate from α (whose clauses do not depend on X). Since α does not depend on X, it can be removed from the existential quantification. But it may, for example, help give tighter results when applying the factor graph algorithm. Therefore, we mark it as separate, but do not ignore it completely.

Suppose we obtain an over-approximate solution $\phi'(Y)$ to problem 1.1 (for example by running the factor graph algorithm). $\phi'(Y)$ is a ***strict*** over-approximation iff there are solutions to:

$$\phi'(Y) \wedge \neg \exists_X \phi(X,Y) \qquad (1.2)$$

Which of course is the same as:

$$\phi'(Y) \wedge \forall_X \neg \phi(X,Y) \qquad (1.3)$$

In other words, the over-approximation is strict if and only if there exists an assignment $Y = a_Y$, such that a_Y satisfies the factor graph solution $\phi'(Y)$ (i.e., $\phi'(a_Y) = \top$), and makes the original problem ϕ unsat (i.e., $\phi(X, a_Y)$ is unsat). Furthermore, if we can find such an assignment a_Y, then we can use it to find a tighter solution

$$\phi''(Y) = \phi'(Y) \wedge \neg a_Y \qquad (1.4)$$

We present an algorithm that searches for such assignments a_Y, and uses them to make over-approximate solutions tighter. To do so, we must investigate the relationship between assignments and CNFs.

2. How assignments modify a CNF

As an example, let us look at the effect of an assignment a_Y (which is a conjunction of literals on the Y variables), on a disjunctive clause $C(X,Y) = x_1 \vee x_2 \vee \neg x_3 \vee y_1 \vee \neg y_2$. In particular, in this example, note that y_1 exists as a positive literal, and y_2 exists as a negative literal in C.

Case 1: If any of the literals in a_Y is consistent with the literals of Y in C,
(i.e., if $y_1(a_Y) = \top$ OR $y_2(a_Y) = \bot$),
then $C(X, a_Y) = \top$.

Case 2: If all the literals in a_Y are inconsistent with the literals of Y in C,
(i.e., if $a_Y = \neg y_1 \wedge y_2$),
then $C(X, a_Y) = x_1 \vee x_2 \vee \neg x_3$.

In other words, a_Y removes all the Y literals from C.

These two cases are exhaustive, and can be generalised to obtain the following lemma:

Lemma 2.1: In a CNF $\phi(X,Y)=\bigwedge C_i(X,Y)$, applying an assignment a_Y on the Y variables will modify each clause C_i as follows:

Case 1: If any of the variable assignments in a_Y is consistent with the the literals in C_i, then $C_i(X,a_Y)=\top$, and hence C_i will completely disappear from $\phi(X,a_Y)$.

Case 2: If all the variable assignments in a_Y are inconsistent with the literals in C_i, then all the Y variables will drop from $C_i(X,a_Y)$.

And thus, when assigning a_Y to ϕ, each clause will either disappear (case 1) or will be preserved but without the Y variables (case 2). Ultimately, $\phi(X,a_Y)$ shall be a CNF consisting of the resulting clauses from case 2.

We introduce the following notations for convenience:

Definition 2.2: When a clause C hits *case 1* due to an assignment a_Y and disappears from $\phi(X,a_Y)$, we say that a_Y ***does not preserve*** C. And likewise when it falls under *case 2*, we say a_Y ***preserves*** C.

Definition 2.3: Furthermore, in case 2 when C is preserved, we use the notation $C\backslash Y$ to denote the resulting clause created by removing all the Y variables from C. The clause $C\backslash Y$ is then said to be ***generated*** by a_Y.

Definition 2.4: Similarly, for a CNF $\phi=\bigwedge_i C_i$ we define the notation $\phi\backslash Y=\bigwedge_i (C_i\backslash Y)$. If an assignment a_Y exists that generates $C_i\backslash Y$ for all i, then a_Y is said to ***generate*** $\phi\backslash Y$.

It might be worth noting that for a disjunctive clause C, $C\backslash Y=\forall_Y C$. Although no such simple interpretation exists for the analogue $\phi\backslash Y$ of a CNF ϕ, we shall see in the next section that it still is of much interest.

3. Removing Y from the CNF

Let us look at $\phi(X,a_Y)$ for some CNF $\phi(X,Y)$ and some assignment a_Y on the Y variables. As per the previous section, this is the same as $\phi(X,Y)$, but with some clauses removed, and some clauses shorter (without the Y variables).

Now let’s look at another formula $\psi(X)=\phi(X,Y)\backslash Y$, which was created by removing all the non quantified Y variables from $\phi(X,Y)$. (Note that each clause in ϕ must have at least one variable from X, which was ensured in problem (1.1) by introducing a separate formula α for X-less clauses). $\psi(X)$ has all the clauses that are in $\phi(X,a_Y)$, but it might also have a few extra clauses which were removed by a_Y. Therefore,

$$\psi(X)\Rightarrow\phi(X,a_Y)$$

And therefore, in order for $\phi(X,a_Y)$ to be unsat, $\psi(X)$ must also be unsat and must have some minimal unsat cores (MUC). An MUC is a subset of clauses that is unsatisfiable, but all of whose strict subsets are satisfiable.

Furthermore, because the clauses of $\phi(X, a_Y)$ are a subset of the clauses of $\psi(X)$, we have the following:

> _Lemma 3.1_: The set of MUCs of $\phi(X, a_Y)$ must be a subset of the MUCs of $\psi(X)$.

This leads us to the following central theorem:

> _Theorem 3.2_: For a given assignment a_Y, the following two statements are equivalent:
>
> - $\phi(X, a_Y)$ is unsat.
> - a_Y generates all clauses in some MUC μ of $\psi(X)$.

Proof: Suppose an assignment a_Y exists, such that $\phi(X, a_Y)$ is unsat. Then, $\phi(X, a_Y)$ must have an MUC μ, which (by lemma 3.1) must be an MUC of $\psi(X)$. Furthermore, because μ is present in $\phi(X, a_Y)$, by definition, a_Y generates μ, which is an MUC of $\psi(X)$.

Conversely, suppose an assignment a_Y exists, which generates all clauses in μ, which is an MUC of $\psi(X)$. Then, by definition of generation, μ is present in $\phi(X, a_Y)$, and therefore, by nature of CNFs, $\phi(X, a_Y) \Rightarrow \mu$. Furthermore, because μ is an MUC, it must be unsat, and therefore, $\phi(X, a_Y)$ must be unsat.

■

Thus, part of the problem (1.3) is reduced to generating MUCs for $\psi(X)$, and then searching for assignments that preserve those MUCs. At the time of writing this document, good tools exist for discovery of MUCs. In the next few sections, we hunt the assignments that generate them.

4. Assignments that generate a clause

Consider the example clause $C(X, Y) = x_1 \vee \neg x_2 \vee y_1 \vee \neg y_2$. Based on Lemma 2.1, C will be preserved by any assignment that has $\neg y_1$ and y_2, i.e., any assignment that is consistent with the partial assignment $\neg y_1 \wedge y_2$. Interestingly, this can conveniently be obtained by dropping all the X variables from C, and taking the negation of the remaining disjunctive clause:

$$\neg(C \setminus X) = \neg(y_1 \vee \neg y_2) = \neg y_1 \wedge y_2$$

We generalise this to define:

> _Definition 4.1_: The **`find_preserver`** function π, for a given clause C, is defined as $\pi(C, X) = \neg(C \setminus X)$. If $C \setminus X$ is empty, then $\pi(C, X)$ is $\top$.
>
> _Lemma 4.2_: An assignment a_Y preserves a clause C iff $a_Y \wedge \pi(C, X) \neq \bot$.

Note that $\pi(C, X)$ does not assign all Y variables, and hence, is a partial assignment. π allows us to find assignments that preserve a clause, say $C(X, Y)$, to generate another clause $C \setminus Y$. Turning it around, we see that $\pi(C, X)$ is also giving us the generating assignments for $C \setminus Y$. However, a clause can be generated by multiple clauses. (For example, using the assignment $y_1 \wedge y_2$, the clause $x_1 \vee \neg x_2$ can be generated by both $x_1 \vee \neg x_2 \vee \neg y_1$ and $x_1 \vee \neg x_2 \vee \neg y_2$). Therefore, if we were to define `find_generators` function to invert π, it would return multiple results:

Algorithm 4.3:

```
# Find the assignments that generate a given clause C
def find_generators( $\phi = \bigwedge_i D_i$ , $C$ ):
    return [ $\pi(D_i, X)$ for $D_i$ in $\phi$ if $D_i \setminus Y$ == $C$ ]
```

However, the fact that ***find_generators*** returns multiple partial assignments leads to combinatorial explosions later in the final algorithm. In order to avoid this issue and make ***find_generators*** return a single partial assignment, we introduce a dummy marker variable for each clause in ϕ. Let the dummy variable for clause D_i be m_i. We then do the following:

1. Add m_i to D_i: This ensures that $D_i \setminus Y$ has the unique variable m_i, and hence, `find_generators(` ϕ, $D_i \setminus Y$ `)` will return D_i as the one and only result.
2. Add m_i to the set of variables X: This ensures that m_i is eventually quantified out, and the final result is independent of m_i.
3. Add $\neg m_i$ as a singleton clause to ϕ: This ensures that the resulting problem is equivalent to the original problem, because setting m_i to $\bot$ is equivalent to removing it from D_i.

Going forward, we assume that ***find_generators***, also written as γ, returns a unique partial assignment.

Algorithm 4.4:

```
# Find the assignments that generate a given clause C
def $\gamma$( $\phi = \bigwedge_i D_i$ , $C$ ):
    for $D_i$ in $\phi$:
        if $D_i \setminus Y$ == $C$ :
            return $\pi(D_i, X)$
    raise Exception("Clause cannot be generated by any assignment")
```

To illustrate using the example we gave earlier, suppose ϕ has two clauses $D_1 = x_1 \vee \neg x_2 \vee \neg y_1$ and $D_2 = x_1 \vee \neg x_2 \vee \neg y_2$. Thus, the clause $x_1 \vee \neg x_2$ has two generating partial assignments: y_1 and y_2. To fix this, we add marker variables m_1 for D_1 and m_2 for D_2, and obtain the equivalent problem $\phi = (x_1 \vee \neg x_2 \vee \neg y_1 \vee m_1) \wedge (x_1 \vee \neg x_2 \vee \neg y_2 \vee m_2) \wedge \neg m_1 \wedge \neg m_2$. By doing so, we now ensure that $x_1 \vee \neg x_2 \vee m_1$ has a unique generating partial assignment y_1, and $x_1 \vee \neg x_2 \vee m_2$ has a unique generating partial assignment y_2. Thus, the function γ is now well defined for each clause, and we are now ready to define the generating partial assignment for an MUC, which is a set of clauses (instead of a single clause).

5. Assignments that generate an MUC

We start by generalizing π to a set of clauses $\phi = \{C_1, C_2, \ldots\}$. Suppose we are given an assignment a_Y of all the Y variables. A clause C_i is preserved by a_Y if and only if $\pi(C_i, X) \wedge a_Y \neq \bot$. Thus, if ϕ needs to be preserved, (i.e., if all C_i need to be preserved), then all $\pi(C_i, X)$ need to be consistent with a_Y, and hence, all of them need to be consistent with each other. In other words, $\forall_i [a_Y \wedge \pi(C_i, X) \neq \bot] \Leftrightarrow [(a_Y \wedge \bigwedge_i \pi(C_i, X)) \neq \bot]$. This allows us to define:

Definition 5.1: The ***find_preserver*** function Π for a given set of clauses $\phi=\{C_1,C_2,...\}$ is defined as $\Pi(\phi,X)=\bigwedge_i \pi(C_i,X)$.

Lemma 5.2: An assignment a_Y preserves a set of clauses $\phi=\{C_1,C_2,...\}$ if and only if $\forall_i[a_Y\wedge\pi(C_i,X)\neq\bot]$, which is the same as saying $a_Y\wedge\Pi(\phi,X)\neq\bot$.

Therefore, analogously, the generating function for a given set of clauses is defined by concatenating the set of assignments for each clause.

Definition 5.3: The ***find_generator*** function Γ for a set of clauses $\mu=\{\mu_1,\mu_2,\ldots\}$ is defined as:

$$\Gamma(\mu)=\bigwedge_i \gamma(\mu_i)$$

Lemma 5.4: A set of clauses can be generated by all assignments a_Y such that $\Gamma(\mu)\wedge a_Y\neq\bot$.

Lemma 5.5: A set of clauses μ is impossible to generate if and only if $\Gamma(\mu)=\bot$.

6. Core algorithm

We now present an algorithm to use an MUC discovery tool to generate assignments that cause the factor graph result to be over-approximate, and use those assignments to refine the factor graph result.

Algorithm 6.1:

```
# Compute $\exists_X \phi(X,Y)\wedge\alpha(Y)$
def oct_22( $\phi$ , $\alpha$ , $X$ , $Y$ ):
    # Modify problem by adding unique marker variables (section 4)
    $(\phi,\alpha,X,Y)$ = add_unique_marker_variables( $\phi$ , $\alpha$ , $X$ , $Y$ )
    # Compute over-approximate result
    $\phi'$ = factor_graph_converge( $\phi\wedge\alpha$ , $X$ )
    # Remove Y variables (section 3)
    $\psi$ = $\phi\backslash Y$
    # Generate minimal unsatisfiable cores
    for $\mu$ in generate_mucs( $\psi$ ):
        # Set of assignments that can generate $\mu$ (definition 5.3)
        $a_Y$ = $\Gamma(\mu)$
        # Ignore assignment if clause is impossible to generate (Lemma 5.5)
        if $a_Y=\bot$ :
            continue
        # Ignore assignment $\phi'$ already excludes it
        if $\phi'(a_Y)=\bot$ :
            continue
        # Use $a_Y$ to make the solution tighter (equation 1.4)
        $\phi'$ = $\phi'\wedge\neg a_Y$
    return $\phi'$
```

7. Searching for consistent MUCs

Thanks to Theorem 3.2, we know that we are interested in finding MUCs of $\psi=\bigwedge_i (C_i \backslash Y)$ that can be generated by some assignment, and by Lemma 5.5, this means we need to find MUCs μ such that $\Gamma(\mu)\neq\bot$. Lines 7-8 in algorithm 6.1 present one way of doing this: use MUC discovery tools to keep creating MUCs until we find one which has a generating assignment. However, there is a better way by which we can directly influence the exploration algorithm of MUC discovery tools, and avoid search spaces that lead to MUCs without generating assignments.

In particular, MUSTool starts with any "seed" set of clauses. If the set of clauses is satisfiable, it marks all subsets as "explored" (since all subsets must also be satisfiable), and then moves on to another "seed" that has not yet been explored. Once it finds a seed set that is unsatisfiable, it marks all supersets as explored (since they must all be unsatisfiable). It then follows a pruning algorithm to get a minimal unsatisfiable subset, starting from that seed. This entire process continues until the full search space (the set of sets of clauses) is explored.

It turns out, it is possible to avoid exploring seeds that can lead to MUCs that cannot be generated by any assignment. In other words, with some clever pre-processing, the check on line 7 in algorithm 6.1 can be avoided. To see how, recall from section 4 how we ensure that every clause in $\psi(X)=\phi(X,Y)\backslash Y$ is generated by a unique clause in $\phi(X,Y)$. Therefore, we may define an correspondence function κ to denote the inverse of $.\backslash Y$

Definition 7.1: For each clause ψ_i in $\psi=\phi\backslash Y$, let κ_i be the unique clause in ϕ such that $\kappa_i \backslash Y=\psi_i$.

Theorem 7.2: An MUC $\mu=\{\mu_1,\mu_2,\ldots\}$ maybe generated by an assignment if and only if none of the clauses in $\{\kappa_1,\kappa_2,\ldots\}$ have any opposite literals on the same Y variable.

Proof: By Lemma 5.5, an MUC μ can be generated by an assignment if and only if $\Gamma(\mu)\neq\bot$.

Furthermore,

$$\Gamma(\mu)\neq\bot$$
$$\Leftrightarrow\left(\bigwedge_i \gamma(\mu_i)\right)\neq\bot \quad \text{(By definition 5.3)}$$
$$\Leftrightarrow\left(\bigwedge_i \pi(\kappa_i)\right)\neq\bot \quad \text{(By definition 7.1 and algorithm 4.4)}$$
$$\Leftrightarrow\forall_{i\neq j}\, \pi(\kappa_i)\wedge\pi(\kappa_j)\neq\bot \quad \text{(Since each } \pi(\kappa_i) \text{ is a conjunction of literals)}$$
$$\Leftrightarrow\forall_{i\neq j}\, \neg(\kappa_i\backslash X)\wedge\neg(\kappa_j\backslash X)\neq\bot \quad \text{(By definition 4.1)}$$
$$\Leftrightarrow \forall_{i\neq j}\ \kappa_i\backslash X \text{ and } \kappa_j\backslash X \text{ do not have opposite literals on the same variable}$$
$$\Leftrightarrow \forall_{i\neq j}\ \kappa_i \text{ and } \kappa_j \text{ do not have opposite literals on the same } Y \text{ variable}$$

■

Theorem 7.2 shows that clauses of ϕ with opposing Y literals are the root cause of MUC's that cannot be generated. If we can tell MUSTool not to explore sets of clauses with any opposing literals, then it will only produce MUCs that can be generated by some assignment. This gives us an algorithm:

Algorithm 7.3:

```
# Mark inconsistent pairs as explored, so that only consistent MUCs are generated
def modify_MUSTool(mustool, ϕ, Y ):
    inconsistent_pairs = find_clause_pairs_with_opp_literals( φ, Y )
    for ( c_1 , c_2 ) in inconsistent_pairs:
        mustool.mark_as_explored( c_1 , c_2 )
```

Lemma 7.4: An instance of MUSTool that has been modified to mark inconsistent pairs of clauses as already explored, will generate only, and all, consistent MUCs.

TODO: comment on complexity

8. Assignments that satisfy the factor graph solution

Thanks to algorithm 7.3 and Lemma 7.4, we can generate all MUCs μ of ψ that can be generated by some assignment a_Y of Y variables. Applying algorithm 5.3 , we can find all assignments that generate MUCs of ψ, and according to theorem 3.2, we now have all assignments such that $\phi(X, a_Y)$ is unsat. However, in order for such assignments to satisfy equation (1.3), they also need to satisfy the factor graph solution $\phi'(Y)$. The check on line 9 in Algorithm 6.1 guarantees this by rejecting all assignments that do not satisfy $\phi'(Y)$. However, in practice, the same assignment often gets generated repeatedly by MUSTool, and it would be better to use $\phi'(Y)$ to somehow refine the search space and avoid exploring irrelevant seeds altogether. In this section we look at one such technique.

As seen in section 6, MUSTool allows us to mark sets of clauses as “explored”, which removes all supersets from its search space. Using this feature, given an assignment a_Y that satisfies $\phi'(Y)$, we would like to find all the sets that can be generated by a_Y , and mark them as explored.

TODO: Write efficient algorithm using ZDDs.

9. Minimal assignments

In practice, we have observed that a partial assignment (e.g. $a_1 = y_1 \wedge y_2 \wedge y_3$) is often immediately followed by another partial assignment with a subset of literals (e.g. $a_2 = y_1 \wedge y_2$). The smaller assignment covers a broader set of full assignments (e.g. $y_1 \wedge y_2 \wedge \neg y_3$ is not consistent with a_1 , but is with a_2). Hence, the smaller assignment is more effective at tightening the final result ($\phi' \wedge \neg a_2 \Rightarrow \phi' \wedge \neg a_1$). Furthermore, the smaller assignment also leads to a larger number of sets of clauses to being marked as explored (section 8). In this section, we present an algorithm to generate a minimal assignment (in terms of the set of literals) that can generate an MUC.

Definition 9.1: Suppose we have two partial assignments a and b consisting of literals l_i and m_i , respectively ($a = \bigwedge l_i$, $b = \bigwedge m_i$), such that $a \neq \bot$ and $b \neq \bot$ (i.e., neither of them have opposite literals on the same variable, $\forall_{i,j} l_i \neq \neg l_j$ and $\forall_{i,j} m_i \neq \neg m_j$). Then we say that “ a contains b ” if and only if $\{l_i\} \supseteq \{m_i\}$, which is equivalent to saying $a \Rightarrow b$.

Definition 9.2: The ***assignment_to_clause*** function for a given partial assignment p returns all the clauses whose generating assignments are contained in p .

```
    def assignment_to_clause( p ,  ψ ):
        return [ D  for  D  in  ψ  if  p  contains  γ(D) ]
```

Theorem 9.3: For a given partial assignment p of Y variables such that $p \neq \bot$, ***assignment_to_clause(*** p , ψ ***)*** is unsatisfiable if and only if ψ has an MUC μ such that p contains $\Gamma(\mu)$.

Proof of Theorem 9.3: Let ρ = ***assignment_to_clause(*** p , ψ ***).***

Suppose that ρ is unsatisfiable. Then, ρ must have an MUC μ, and since $\mu\subseteq\rho\subseteq\psi$, μ must be an MUC of ψ. Furthermore, by definition, p contains $\Gamma(\rho)$, which contains $\Gamma(\mu)$, and hence, p contains $\Gamma(\mu)$.

Conversely, suppose that ψ has an MUC μ such that p contains $\Gamma(\mu)$. Then, by definition 5.3, p contains $\gamma(\mu_i)$, $\forall\mu_i\in\mu$. Thus, by definition 9.2, μ is a subset of ρ. But since μ is unsatisfiable, ρ must be unsatisfiable as well.

■

Notations

Notation	Name	Description
$\phi(X,Y)$	CNF to be quantified	Main CNF being quantified
X	Quantified variables	The variables being existentially quantified from ϕ
Y	Remaining variables	Variables remaining in $\exists_X\phi(X,Y)$
$\exists_X\phi(X,Y)$	The problem	Main problem being solved
$\phi'(Y)$	Over approximation	An Over-approximate solution to the problem, i.e., $\phi'(Y)\Leftarrow\exists_X\phi(X,Y)$
a_Y	Partial ssignment on Y variables	$a_Y=\bigwedge_i l_i$ where each l_i is either y_i or $\neg y_i$ for some $y_i\in Y$
C_i	Clause in ϕ	$\phi=\bigwedge_i C_i$. Subscript i may be dropped if appropriate.
$C\backslash Y$	Projection of a clause	Clause obtained by removing Y literals from C. Also equal to $\forall_Y C$.
a_Y ***preserves*** C a_Y ***generates*** $C\backslash Y$	Preservation / Generation	There is no literal in a_Y that is also in C. Thus, $C(a_Y)=C\backslash Y$. Therefore, $\phi(X,a_Y)$ has $C\backslash Y$.
m_i	Marker variable	Unique new X variable added into C_i so that $C_i\backslash Y$ is unique.
$\psi(X)=\phi(X,Y)\backslash Y$	Projection of a CNF	CNF obtained by removing Y variables from all clauses ϕ. I.e., $\psi=\bigwedge_i C_i\backslash Y$
μ	A MUC: minimal unsatisfiable core	A set of clauses whose conjunction is unsatisfiable, but for each strict subset of clauses, the conjunction is satisfiable.
π	***find_preserver***	For a clause C, $\pi(C,X)=\neg(C\backslash X)$ is the partial assignment that ***preserves*** C. A full assignment a_Y of Y variables preserves C *iff* $a_Y\wedge\pi(C,X)\neq\bot$. If $C\backslash X$ is empty then $\pi(C,X)$ is defined as $\top$.
γ	***find_generator***	