Minimum Satisfying Assignments for SMT

Işıl Dillig, Tom Dillig
College of William & Mary

Ken McMillan
Microsoft Research

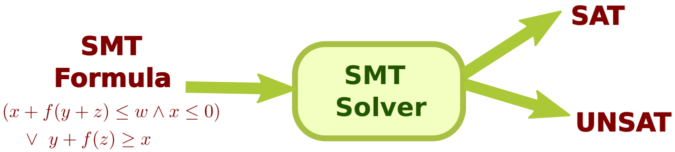
Alex Aiken
Stanford U.

Satisfiability Modulo Theories (SMT)

- Today, SMT solvers are underlying engine of most verification and program analysis tools

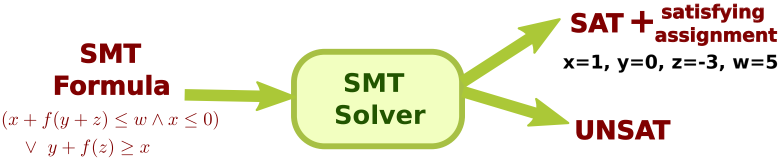
Satisfiability Modulo Theories (SMT)

- Today, SMT solvers are underlying engine of most verification and program analysis tools



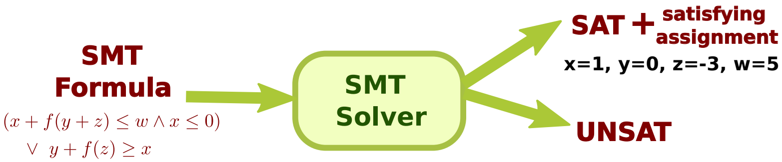
Satisfiability Modulo Theories (SMT)

- Today, SMT solvers are underlying engine of most verification and program analysis tools



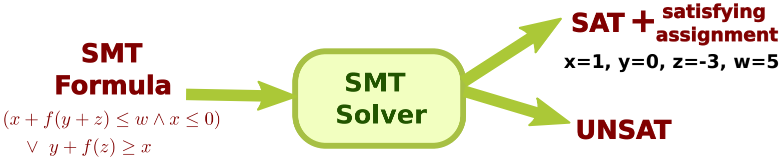
- An assignment σ for formula ϕ is a mapping from free variables of ϕ to values

Partial Satisfying Assignments



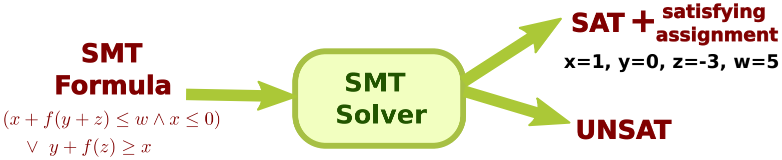
- Satisfying assignments provided SMT solver are full assignments $\Rightarrow$ assign every free variable to a value

Partial Satisfying Assignments



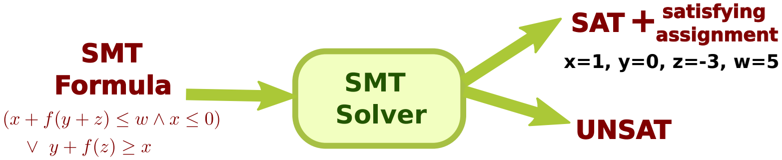
- Satisfying assignments provided SMT solver are full assignments $\Rightarrow$ assign every free variable to a value
- But sometimes we want partial satisfying assignments

Partial Satisfying Assignments



- Satisfying assignments provided SMT solver are full assignments $\Rightarrow$ assign every free variable to a value
- But sometimes we want partial satisfying assignments
- A partial satisfying assignment only assigns values to a subset of free variables, but is sufficient to make formula true

Partial Satisfying Assignments



- Satisfying assignments provided SMT solver are full assignments $\Rightarrow$ assign every free variable to a value
- But sometimes we want partial satisfying assignments
- A partial satisfying assignment only assigns values to a subset of free variables, but is sufficient to make formula true
- For formula $x < 0 \vee x + y \geq 0$, $x = -1$ is a partial satisfying assignment

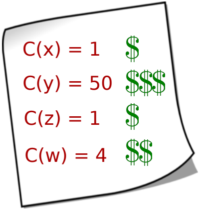
Minimum Satisfying Assignments (MSA)

Special class of partial sat assignments: minimum sat assignments

Minimum Satisfying Assignments (MSA)

Special class of partial sat assignments: minimum sat assignments

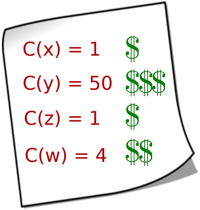
- Given cost function C from variables to costs, minimum sat assignment (MSA) is a partial sat assignment minimizing C



Minimum Satisfying Assignments (MSA)

Special class of partial sat assignments: minimum sat assignments

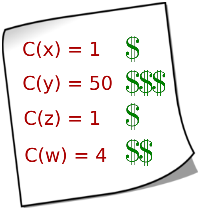
- Given cost function C from variables to costs, minimum sat assignment (MSA) is a partial sat assignment minimizing C
- Observe: Cost of assignment does not depend on values, but only to variables used in assignment!



Minimum Satisfying Assignments (MSA)

Special class of partial sat assignments: minimum sat assignments

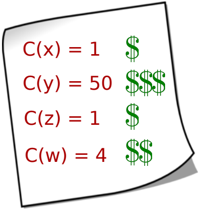
- Given cost function C from variables to costs, minimum sat assignment (MSA) is a partial sat assignment minimizing C
- Observe: Cost of assignment does not depend on values, but only to variables used in assignment!
- Assignments $x = 1$ and $x = 50$ have same cost



Minimum Satisfying Assignments (MSA)

Special class of partial sat assignments: minimum sat assignments

- Given cost function C from variables to costs, minimum sat assignment (MSA) is a partial sat assignment minimizing C
- Observe: Cost of assignment does not depend on values, but only to variables used in assignment!
- Assignments $x = 1$ and $x = 50$ have same cost
- If variables have equal cost, an MSA is partial sat assignment with fewest variables



Example and Applications

- Consider cost function assigning every variable to 1 and Presburger arithmetic formula:

$$\phi : x + y + w > 0 \vee x + y + z + w < 5$$

Example and Applications

- Consider cost function assigning every variable to 1 and Presburger arithmetic formula:

$$\phi : x + y + w > 0 \vee x + y + z + w < 5$$

- A minimum sat assignment for ϕ is $z = 0$

Example and Applications

- Consider cost function assigning every variable to 1 and Presburger arithmetic formula:

$$\phi : x + y + w > 0 \vee x + y + z + w < 5$$

- A minimum sat assignment for ϕ is $z = 0$

MSAs have many applications in verification:

- ✓ Finding small counterexamples in BMC
- ✓ Classifying and diagnosing error reports
- ✓ Abductive inference
- ✓ Minimizing # of predicates in pred abstraction

Contributions

- In propositional logic, MSAs known as min-sized prime implicants

Contributions

- In propositional logic, MSAs known as min-sized prime implicants
- Known algorithms for computing min prime implicants

Contributions

- In propositional logic, MSAs known as min-sized prime implicants
- Known algorithms for computing min prime implicants
- No work on computing MSAs for richer theories

Contributions

- In propositional logic, MSAs known as min-sized prime implicants
- Known algorithms for computing min prime implicants
- No work on computing MSAs for richer theories

✓ **First algorithm for computing min sat assignments for SMT formulas**

✓ **Our algorithm applicable to any theory for which full first-order logic including quantifiers is decidable**

Maximum Universal Subset

- Our algorithm exploits duality between MSAs and maximum universal subsets (MUS)

Maximum Universal Subset

- Our algorithm exploits duality between MSAs and maximum universal subsets (MUS)
- An MUS for ϕ is a set of variables X s.t.:

Maximum Universal Subset

- Our algorithm exploits duality between MSAs and maximum universal subsets (MUS)
- An MUS for ϕ is a set of variables X s.t.:
  1. $\forall X.\phi$ is satisfiable

Maximum Universal Subset

- Our algorithm exploits duality between MSAs and maximum universal subsets (MUS)
- An MUS for ϕ is a set of variables X s.t.:
  1. $\forall X . \phi$ is satisfiable
  2. X maximizes cost function C

Maximum Universal Subset

- Our algorithm exploits duality between MSAs and maximum universal subsets (MUS)
- An MUS for ϕ is a set of variables X s.t.:
  1. $\forall X.\phi$ is satisfiable
  2. X maximizes cost function C

X is an MUS of ϕ $\Leftrightarrow$ MSA is a sat assignment of $\forall X.\phi$

Maximum Universal Subset

- Our algorithm exploits duality between MSAs and maximum universal subsets (MUS)
- An MUS for ϕ is a set of variables X s.t.:
  1. $\forall X.\phi$ is satisfiable
  2. X maximizes cost function C

X is an MUS of ϕ $\Leftrightarrow$ MSA is a sat assignment of $\forall X.\phi$

Our approach first computes an MUS X and extracts an MSA from a sat assignment of $\forall X.\phi$

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:

```
find_mus(φ, C, V, L) {

}
```

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:
  - Current formula ϕ

```
find_mus(φ, C, V, L) {

}
```

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:
  - Current formula ϕ
  - Cost function C

```
find_mus(ϕ, C, V, L) {

}
```

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:
  - Current formula ϕ
  - Cost function C
  - Remaining variables V

```
find_mus(φ, C, V, L) {

}
```

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:
  - Current formula ϕ
  - Cost function C
  - Remaining variables V
  - Lower bound L

```
find_mus(φ, C, V, L) {

}
```

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:
  - Current formula ϕ
  - Cost function C
  - Remaining variables V
  - Lower bound L
- L is used to prune the search

find_mus(ϕ, C, V, L) {

  If $V = \emptyset$ or $C(V) \leq L$ return $\emptyset$

}

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:
  - Current formula ϕ
  - Cost function C
  - Remaining variables V
  - Lower bound L
- L is used to prune the search
- At each recursive call, considers new variable x and decides if x is in or out of MUS

```
find_mus(φ, C, V, L) {
    If V = ∅ or C(V) ≤ L  return ∅
    Set best = ∅
    choose x ∈ V
    V′ = V\{x}
}
```

Algorithm to Compute MUS

- Recursive branch-and-bound style algorithm with input:
  - Current formula ϕ
  - Cost function C
  - Remaining variables V
  - Lower bound L
- L is used to prune the search
- At each recursive call, considers new variable x and decides if x is in or out of MUS
- We do this by comparing cost of universal subsets with and without x

```
find_mus(φ, C, V, L) {
    If V = ∅ or C(V) ≤ L  return ∅
    Set best = ∅
    choose x ∈ V
    V′ = V\{x}
}
```

Algorithm to Compute MUS, cont.

- First, check if possible to include x is in universal subset

```
find_mus(φ, C, V, L) {
   If V = ∅ or C(V) ≤ L  return ∅
   Set best = ∅
   choose x ∈ V
   V′ = V\{x}
   if(SAT(∀x.φ)) {


   }


}
```

Algorithm to Compute MUS, cont.

- First, check if possible to include x is in universal subset
- If so, recursively compute cost of universal subset containing x

```
find_mus(φ, C, V, L) {
   If V = ∅ or C(V) ≤ L  return ∅
   Set best = ∅
   choose x ∈ V
   V' = V\{x}
   if(SAT(∀x.φ)) {
      Set Y = find_mus(∀x.φ, C, V' L − C(x));
   }
}
```

Algorithm to Compute MUS, cont.

- First, check if possible to include x is in universal subset
- If so, recursively compute cost of universal subset containing x
- If cost of current universal subset is better than previous best cost, update lower bound

```
find_mus(φ, C, V, L) {
  If V = ∅ or C(V) ≤ L  return ∅
  Set best = ∅
  choose x ∈ V
  V′ = V\{x}

  if(SAT(∀x.φ)) {
     Set Y = find_mus(∀x.φ, C, V′ L − C(x));
     Int cost = C(Y) + C(x)
     If (cost > L) { best = Y ∪ {x}; L = cost }
  }

}
```

Algorithm to Compute MUS, cont.

- First, check if possible to include x is in universal subset
- If so, recursively compute cost of universal subset containing x
- If cost of current universal subset is better than previous best cost, update lower bound
- Next, compute cost of universal subset not containing x

```
find_mus(ϕ, C, V, L) {
    If V = ∅ or C(V) ≤ L  return ∅
    Set best = ∅
    choose x ∈ V
    V′ = V\{x}

    if(SAT(∀x.ϕ)) {
        Set Y = find_mus(∀x.ϕ, C, V′ L − C(x));
        Int cost = C(Y) + C(x)
        If (cost > L) { best = Y ∪ {x}; L = cost }
    }
    Set Y = find_mus(ϕ,C,V′,L);


}
```

Algorithm to Compute MUS, cont.

- First, check if possible to include x is in universal subset
- If so, recursively compute cost of universal subset containing x
- If cost of current universal subset is better than previous best cost, update lower bound
- Next, compute cost of universal subset not containing x
- Compare the two costs and return whichever is best

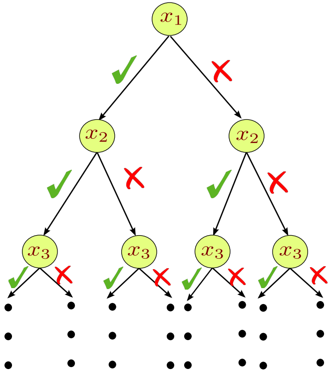
```
find_mus(φ, C, V, L) {
   If V = ∅ or C(V) ≤ L  return ∅
   Set best = ∅
   choose x ∈ V
   V' = V\{x}

   if(SAT(∀x.φ)) {
      Set Y = find_mus(∀x.φ, C, V' L − C(x));
      Int cost = C(Y) + C(x)
      If (cost > L) { best = Y ∪ {x}; L = cost }
   }

   Set Y = find_mus(φ,C,V',L);
   If (C(Y) > L) { best = Y }
   return best;
}
```

Discussion of Algorithm

- This algorithm is branch-and-bound style

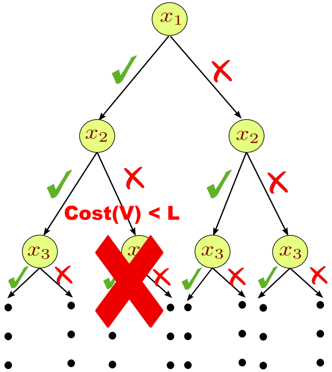
Discussion of Algorithm

- This algorithm is branch-and-bound style
- Branches on whether a given variable is in or out of MUS



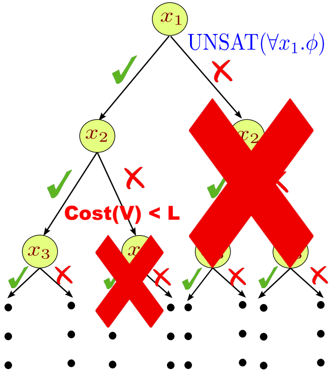
Discussion of Algorithm

- This algorithm is branch-and-bound style
- Branches on whether a given variable is in or out of MUS
- Bounds search by checking if current max possible cost cannot improve previous estimate



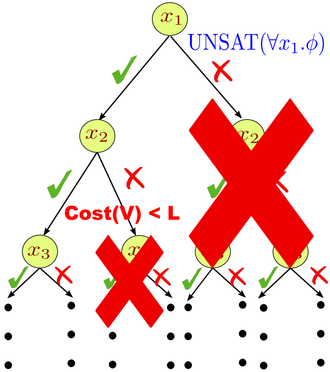
Discussion of Algorithm

- This algorithm is branch-and-bound style
- Branches on whether a given variable is in or out of MUS
- Bounds search by checking if current max possible cost cannot improve previous estimate
- Also bounds search by checking when $\forall X.\phi$ becomes unsat



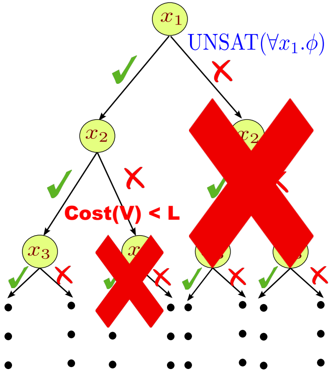
Discussion of Algorithm

- This algorithm is branch-and-bound style
- Branches on whether a given variable is in or out of MUS
- Bounds search by checking if current max possible cost cannot improve previous estimate
- Also bounds search by checking when $\forall X.\phi$ becomes unsat
- These two pruning strategies eliminate many search paths, but still exponential



Discussion of Algorithm

- This algorithm is branch-and-bound style
- Branches on whether a given variable is in or out of MUS
- Bounds search by checking if current max possible cost cannot improve previous estimate
- Also bounds search by checking when $\forall X.\phi$ becomes unsat
- These two pruning strategies eliminate many search paths, but still exponential
- To make algorithm practical, must consider more optimizations



Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate

Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate
    - Basic algorithm initially sets lower bound on MUS cost to 0

Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate
   - Basic algorithm initially sets lower bound on MUS cost to 0
   - Improves as algorithm progresses, but initially ineffective

Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate
   - Basic algorithm initially sets lower bound on MUS cost to 0
   - Improves as algorithm progresses, but initially ineffective
   - If we can "quickly" compute a better initial lower bound estimate, pruning can be much more effective

Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate
   - Basic algorithm initially sets lower bound on MUS cost to 0
   - Improves as algorithm progresses, but initially ineffective
   - If we can "quickly" compute a better initial lower bound estimate, pruning can be much more effective
2. Variable order

Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate
   - Basic algorithm initially sets lower bound on MUS cost to 0
   - Improves as algorithm progresses, but initially ineffective
   - If we can "quickly" compute a better initial lower bound estimate, pruning can be much more effective
2. Variable order
   - Basic algorithm chooses variables randomly

Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate
   - Basic algorithm initially sets lower bound on MUS cost to 0
   - Improves as algorithm progresses, but initially ineffective
   - If we can "quickly" compute a better initial lower bound estimate, pruning can be much more effective
2. Variable order
   - Basic algorithm chooses variables randomly
   - But some variable orders much better than others

Improvements over Basic Algorithm

Two important ways to improve over basic algorithm:

1. Initial cost estimate
   - Basic algorithm initially sets lower bound on MUS cost to 0
   - Improves as algorithm progresses, but initially ineffective
   - If we can "quickly" compute a better initial lower bound estimate, pruning can be much more effective
2. Variable order
   - Basic algorithm chooses variables randomly
   - But some variable orders much better than others
   - Turns out better to consider variables likely to be in MSA first

Finding Initial Cost and Variable Order

- If we can find good approximation for MSA, can obtain good initial cost and variable order for MUS

Finding Initial Cost and Variable Order

- If we can find good approximation for MSA, can obtain good initial cost and variable order for MUS
- MUS Cost = Cost of free vars - MSA cost

Finding Initial Cost and Variable Order

- If we can find good approximation for MSA, can obtain good initial cost and variable order for MUS
- MUS Cost = Cost of free vars - MSA cost
- Thus, good estimate on MSA cost gives good estimate on MUS cost

Finding Initial Cost and Variable Order

- If we can find good approximation for MSA, can obtain good initial cost and variable order for MUS
- MUS Cost = Cost of free vars - MSA cost
- Thus, good estimate on MSA cost gives good estimate on MUS cost
- Good approximate MSA gives good variable order b/c if x is in MSA, $\forall x.\phi$ more likely unsat

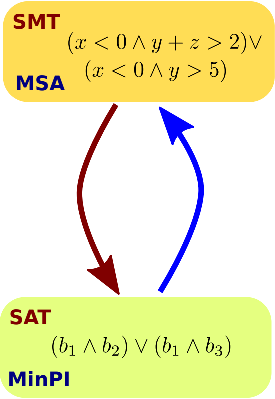
Finding Initial Cost and Variable Order

- If we can find good approximation for MSA, can obtain good initial cost and variable order for MUS
- MUS Cost $=$ Cost of free vars - MSA cost
- Thus, good estimate on MSA cost gives good estimate on MUS cost
- Good approximate MSA gives good variable order b/c if x is in MSA, $\forall x.\phi$ more likely unsat

How can we "quickly" find good enough approximate MSA?

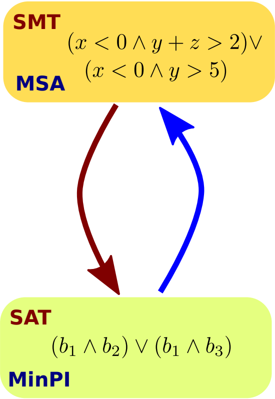
Approximate MSA from Prime Implicants

- Use min prime implicant to boolean structure of formula to approximate MSA



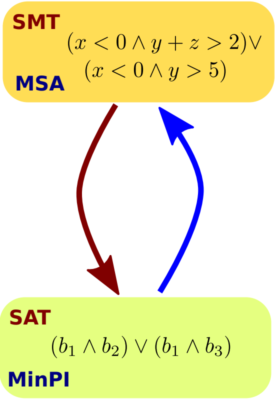
Approximate MSA from Prime Implicants

- Use min prime implicant to boolean structure of formula to approximate MSA
- Prime implicant of boolean formula is conjunction of literals that implies it



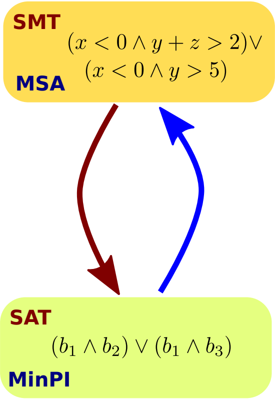
Approximate MSA from Prime Implicants

- Use min prime implicant to boolean structure of formula to approximate MSA
- Prime implicant of boolean formula is conjunction of literals that implies it
- First, compute ϕ^+ by replacing every literal in ϕ by boolean variable



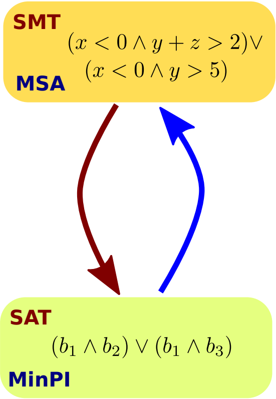
Approximate MSA from Prime Implicants

- Use min prime implicant to boolean structure of formula to approximate MSA
- Prime implicant of boolean formula is conjunction of literals that implies it
- First, compute ϕ^+ by replacing every literal in ϕ by boolean variable
- Extend techniques for computing MinPI to find a theory-satisfiable MinPI



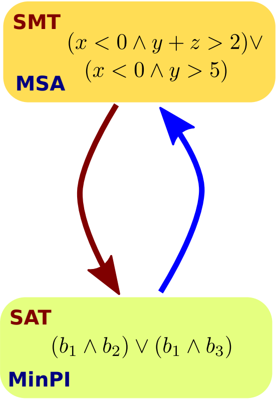
Approximate MSA from Prime Implicants

- Use min prime implicant to boolean structure of formula to approximate MSA
- Prime implicant of boolean formula is conjunction of literals that implies it
- First, compute ϕ^+ by replacing every literal in ϕ by boolean variable
- Extend techniques for computing MinPI to find a theory-satisfiable MinPI
- Theory-sat PI implies boolean structure of formula and is satisfiable modulo theory



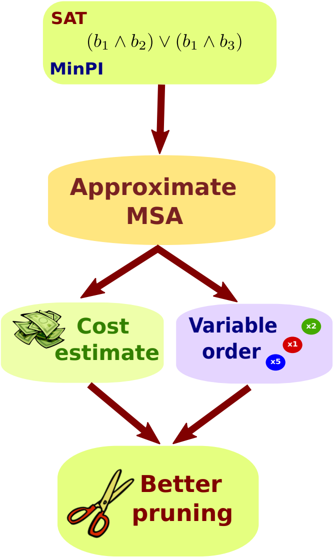
Approximate MSA from Prime Implicants

- Use min prime implicant to boolean structure of formula to approximate MSA
- Prime implicant of boolean formula is conjunction of literals that implies it
- First, compute ϕ^+ by replacing every literal in ϕ by boolean variable
- Extend techniques for computing MinPI to find a theory-satisfiable MinPI
- Theory-sat PI implies boolean structure of formula and is satisfiable modulo theory
- Approximate MSA as variables in MinPI



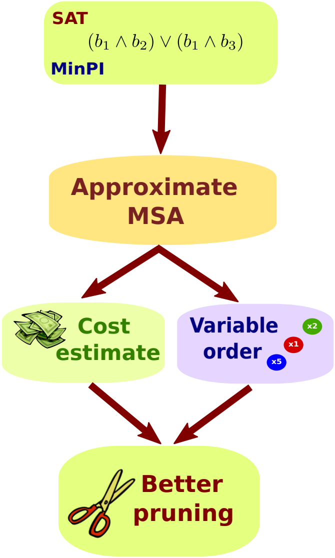
Summary of First Optimization

- Optimize basic B&B algorithm by finding good lower bound estimate on MUS and variable order



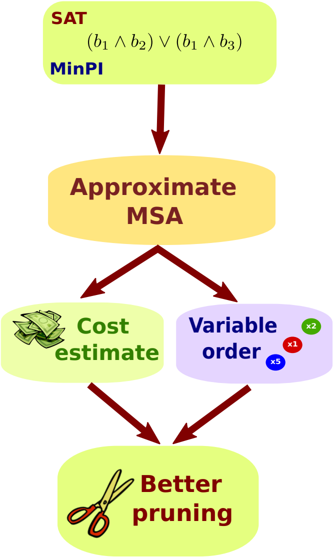
Summary of First Optimization

- Optimize basic B&B algorithm by finding good lower bound estimate on MUS and variable order
- To find good estimate and variable order, compute approximate MSA



Summary of First Optimization

- Optimize basic B&B algorithm by finding good lower bound estimate on MUS and variable order
- To find good estimate and variable order, compute approximate MSA
- Approximate MSA is obtained from theory-satisfiable min PI of boolean structure



Another Improvement: Non-Universal Subsets

- Suppose we knew a set of variables V is a non-universal set (i.e., $\forall V.\phi$ is UNSAT)

Another Improvement: Non-Universal Subsets

- Suppose we knew a set of variables V is a non-universal set (i.e., $\forall V.\phi$ is UNSAT)
- Can use non-universal subsets to improve algorithm because can avoid branching without $\mathrm{SAT}(\forall X.\phi)$ check

Another Improvement: Non-Universal Subsets

- Suppose we knew a set of variables V is a non-universal set (i.e., $\forall V.\phi$ is UNSAT)
- Can use non-universal subsets to improve algorithm because can avoid branching without $\mathrm{SAT}(\forall X.\phi)$ check
- Furthermore, if V is a non-universal subset of implicate of ϕ, it is also non-universal subset of of ϕ.

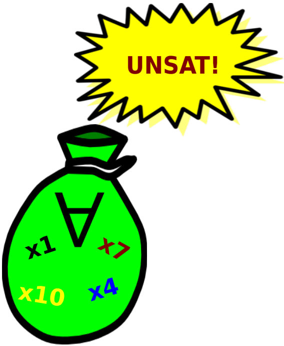
Another Improvement: Non-Universal Subsets

- Suppose we knew a set of variables V is a non-universal set (i.e., $\forall V.\phi$ is UNSAT)
- Can use non-universal subsets to improve algorithm because can avoid branching without $\mathrm{SAT}(\forall X.\phi)$ check
- Furthermore, if V is a non-universal subset of implicate of ϕ, it is also non-universal subset of of ϕ.

How can we "quickly" find implicates with small non-universal subsets?

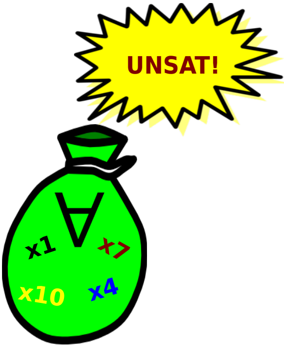
Finding Non-Universal Subsets

- For complete theories, such as Presburger arithmetic, if $\neg\psi$ sat, then $\forall \text{free}(\psi).\psi$ unsat



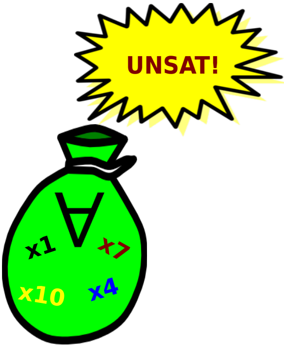
Finding Non-Universal Subsets

- For complete theories, such as Presburger arithmetic, if $\neg\psi$ sat, then $\forall \text{free}(\psi).\psi$ unsat
- Thus, if ψ is an implicate of ϕ whose negation is sat, free(ψ) is a non-universal set



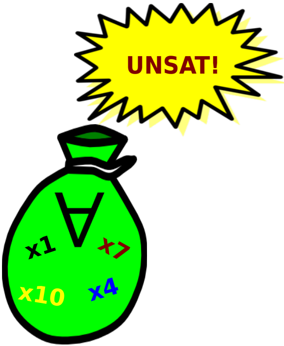
Finding Non-Universal Subsets

- For complete theories, such as Presburger arithmetic, if $\neg\psi$ sat, then $\forall \mathrm{free}(\psi).\psi$ unsat
- Thus, if ψ is an implicate of ϕ whose negation is sat, free(ψ) is a non-universal set
- Can quickly find implicates with this property from boolean structure of simplified form



Finding Non-Universal Subsets

- For complete theories, such as Presburger arithmetic, if $\neg\psi$ sat, then $\forall \text{free}(\psi).\psi$ unsat
- Thus, if ψ is an implicate of ϕ whose negation is sat, free(ψ) is a non-universal set
- Can quickly find implicates with this property from boolean structure of simplified form
- When all variables in ψ are $\forall$-quantified, backtrack without checking satisfiability



Experimental Evaluation

- Implemented algorithm in Mistral SMT solver

Experimental Evaluation

- Implemented algorithm in Mistral SMT solver
- Evaluated algorithm on 400 Presburger arithmetic formulas

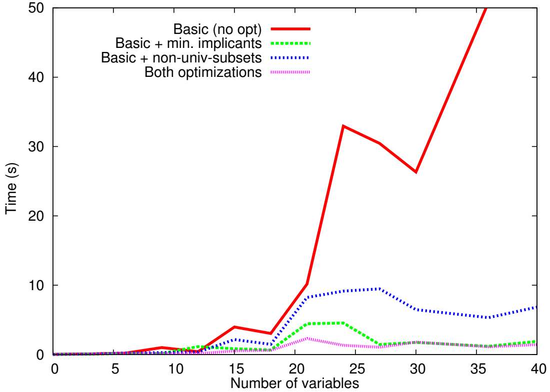
Experimental Evaluation

- Implemented algorithm in Mistral SMT solver
- Evaluated algorithm on 400 Presburger arithmetic formulas
- Formulas taken from static analysis tool that uses MSAs for performing abduction, in turn used for diagnosing error reports

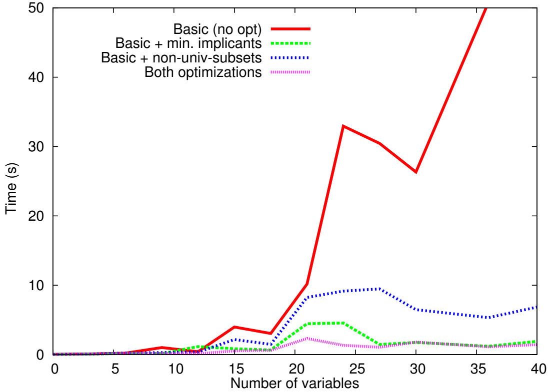
Experimental Evaluation

- Implemented algorithm in Mistral SMT solver
- Evaluated algorithm on 400 Presburger arithmetic formulas
- Formulas taken from static analysis tool that uses MSAs for performing abduction, in turn used for diagnosing error reports
- Formulas contain up to 40 variables and several hundred boolean connectives

Experimental Results

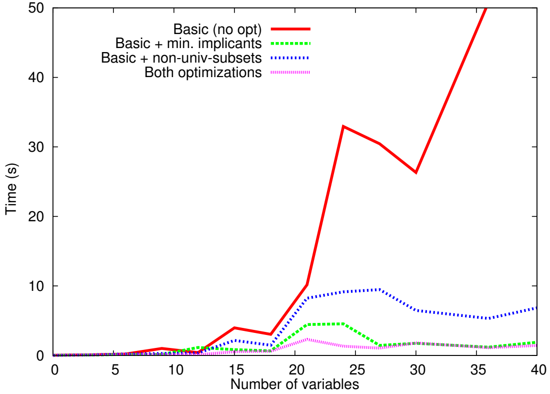


Experimental Results



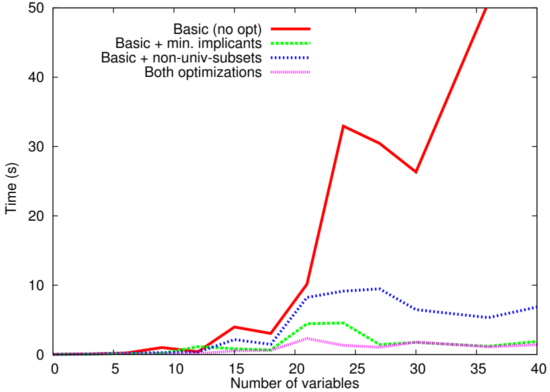
- Basic algorithm very sensitive to # vars

Experimental Results



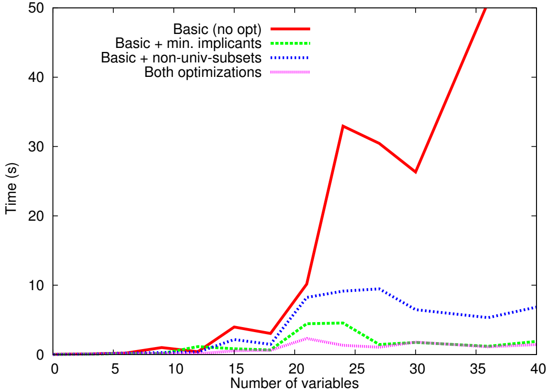
- Basic algorithm very sensitive to # vars
- Optimizations have dramatic impact on performance

Experimental Results



- Basic algorithm very sensitive to # vars
- Optimizations have dramatic impact on performance
- Optimized version grows slowly in # of variables

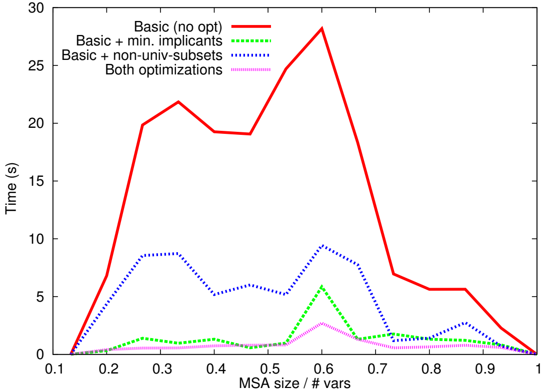
Experimental Results



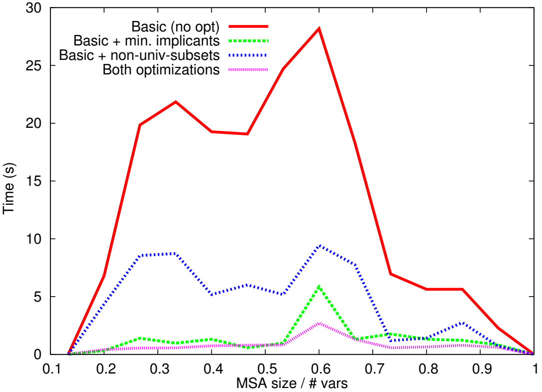
- Basic algorithm very sensitive to # vars
- Optimizations have dramatic impact on performance
- Optimized version grows slowly in # of variables

Even with both optimizations, computing MSAs 25 times more expensive than checking satisfiability

Experimental Results, cont.

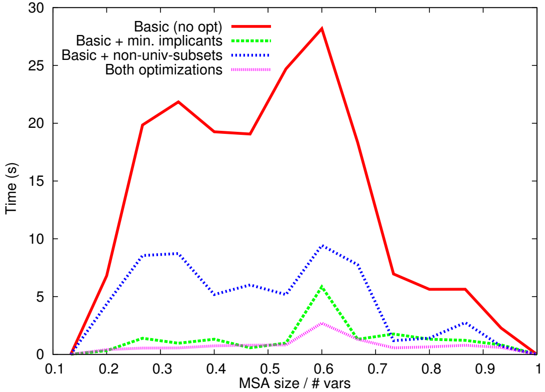


Experimental Results, cont.



- Problem easier if # vars in MSA very small or very large

Experimental Results, cont.



- Problem easier if # vars in MSA very small or very large
- Problem hardest for formulas when ratio of vars in MSA to free vars is ≈ 0.6

Summary

- First algorithm for finding MSAs of SMT formulas
- Recursive branch-and-bound style algorithm with two crucial optimizations
- MSAs can be computed in reasonable time for a set of benchmakrs obtained from static analysis
- But finding MSAs much more expensive than finding full sat assignment
- We believe significant improvements are still possible