Trie Based Subsumption and Improving the pi-Trie Algorithm

Andrew Matusiewicz, Neil V. Murray, Erik Rosenthal

June 16, 2011
What are prime implicates and why care?

- Smallest, simplest disjunctions implied by a statement

Research partner uses in code gen. for embedded systems

(Sandeep Shukla, Virginia Tech.)
What are prime implicates and why care?

- Smallest, simplest disjunctions implied by a statement
- Knowing all prime implicates, clausal implication is easy
What are prime implicates and why care?

- Smallest, simplest disjunctions implied by a statement
- Knowing all prime implicates, clausal implication is easy
- Used in non-monotonic and abductive reasoning
What are prime implicates and why care?

- Smallest, simplest disjunctions implied by a statement
- Knowing all prime implicates, clausal implication is easy
- Used in non-monotonic and abductive reasoning
- Dual, prime implicants, used for circuit minimization
What are prime implicates and why care?

- Smallest, simplest disjunctions implied by a statement
- Knowing all prime implicates, clausal implication is easy
- Used in non-monotonic and abductive reasoning
- Dual, prime implicants, used for circuit minimization
- Research partner uses in code gen. for embedded systems (Sandeep Shukla, Virginia Tech.)
Propositional Clauses

- Propositional variables \( v_1, v_2, \ldots \) are either true or false
- Connectives \( \wedge \) (and), \( \vee \) (or), \( \neg \) (not).
- A literal \( \ell \) is a negated or non-negated variable: \( v_i, \neg v_i \)

- A clause is an “or” of literals:

  \[
  (v_1 \vee \neg v_3 \vee v_2 \vee v_9) = \{v_1, \neg v_3, v_2, v_9\}
  \]
Prime Implicates

A clause $C$ is an “implicate” of a formula $\mathcal{F}$ iff the statement 

$$\mathcal{F} \Rightarrow C$$

is a tautology.
Prime Implicates

A clause $C$ is an “implicate” of a formula $\mathcal{F}$ iff the statement

$$\mathcal{F} \Rightarrow C$$

is a tautology.

$C$ is a “prime implicate” of $\mathcal{F}$, written

$$C \in \mathcal{P}(\mathcal{F})$$

iff $C$ is an implicate of minimal length.
The Prime Implicate Production Problem

**Given**: A boolean formula $\mathcal{F}$.

**Produce**: The set $\mathcal{P}(\mathcal{F})$ of all prime implicants of $\mathcal{F}$. 


Algorithm Scheme

To calculate $\mathcal{P}(\mathcal{F})$ from $\mathcal{F}$,

1. Split $\mathcal{F}$ by substitution of 0, 1 for $v$, yielding $\mathcal{F}_0, \mathcal{F}_1$. 

Base cases are boolean constants.
Algorithm Scheme

To calculate $\mathcal{P}(\mathcal{F})$ from $\mathcal{F}$,

1. Split $\mathcal{F}$ by substitution of 0, 1 for $v$, yielding $\mathcal{F}_0, \mathcal{F}_1$.
2. By recursion obtain $\mathcal{P}_0 = \mathcal{P}(\mathcal{F}_0)$ and $\mathcal{P}_1 = \mathcal{P}(\mathcal{F}_1)$. 
To calculate $\mathcal{P}(\mathcal{F})$ from $\mathcal{F}$,

1. Split $\mathcal{F}$ by substitution of 0, 1 for $v$, yielding $\mathcal{F}_0, \mathcal{F}_1$.
2. By recursion obtain $\mathcal{P}_0 = \mathcal{P}(\mathcal{F}_0)$ and $\mathcal{P}_1 = \mathcal{P}(\mathcal{F}_1)$.
3. Combine $\mathcal{P}_0, \mathcal{P}_1$ with splitting formula to obtain $\mathcal{P}(\mathcal{F})$. 

Base cases are boolean constants.
To calculate $P(\mathcal{F})$ from $\mathcal{F}$,

1. Split $\mathcal{F}$ by substitution of 0, 1 for $v$, yielding $\mathcal{F}_0, \mathcal{F}_1$.
2. By recursion obtain $P_0 = P(\mathcal{F}_0)$ and $P_1 = P(\mathcal{F}_1)$.
3. Combine $P_0, P_1$ with splitting formula to obtain $P(\mathcal{F})$.

Base cases are boolean constants.
Subsumption Classes of Primes

Let $x \in \{0, 1\}$
Subsumption Classes of Primes

Let $x \in \{0, 1\}$

$P_x = P(\mathcal{F}[x/v])$ — Prime implicates of $\mathcal{F}_0, \mathcal{F}_1$
Subsumption Classes of Primes

Let $x \in \{0, 1\}$

$\mathcal{P}_x = \mathcal{P}(\mathcal{F}[x/v])$ — Prime implicates of $\mathcal{F}_0, \mathcal{F}_1$

$\mathcal{P}_x = \mathcal{P}_x^2 \cup \mathcal{P}_x^\infty$ — Split $\mathcal{P}_x$ into disjoint clause sets
Subsumption Classes of Primes

Let $x \in \{0, 1\}$

$\mathcal{P}_x = \mathcal{P}(\mathcal{F}[x/v])$ — Prime implicates of $\mathcal{F}_0, \mathcal{F}_1$

$\mathcal{P}_x = \mathcal{P}_x^\ominus \cup \mathcal{P}_x^{\infty}$ — Split $\mathcal{P}_x$ into disjoint clause sets

$\mathcal{P}_x^\ominus$ — Clauses in $\mathcal{P}_x$ subsumed by some clause in $\mathcal{P}_{(1-x)}$
Let \( x \in \{0, 1\} \)

\[
\mathcal{P}_x = \mathcal{P}(\mathcal{F}[x/v]) \quad \text{— Prime implicates of } \mathcal{F}_0, \mathcal{F}_1
\]

\[
\mathcal{P}_x = \mathcal{P}_x^\supseteq \cup \mathcal{P}_x^\bowtie \quad \text{— Split } \mathcal{P}_x \text{ into disjoint clause sets}
\]

\[
\mathcal{P}_x^\supseteq \quad \text{— Clauses in } \mathcal{P}_x \text{ subsumed by some clause in } \mathcal{P}_{(1-x)}
\]

\[
\mathcal{P}_x^\bowtie \quad \text{— Remainder of } \mathcal{P}_x, \text{ i.e. those not subsumed in } \mathcal{P}_{(1-x)}
\]
Splitting Formula for Primes

Given $\mathcal{P}_0, \mathcal{P}_1$, we find subsumption classes $\mathcal{P}_0^\triangleright, \mathcal{P}_1^\triangleright, \mathcal{P}_0^\triangleleft, \mathcal{P}_1^\triangleleft$. 
Splitting Formula for Primes

Given \( \mathcal{P}_0, \mathcal{P}_1 \), we find subsumption classes \( \mathcal{P}_0 \supseteq, \mathcal{P}_1 \supseteq, \mathcal{P}_0 \bowtie, \mathcal{P}_1 \bowtie \).

From this:

\[
\mathcal{P}(\mathcal{F}) = (v \lor \mathcal{P}_0 \bowtie) \cup (\neg v \lor \mathcal{P}_1 \bowtie) \cup \mathcal{U}
\]

Where \( \mathcal{U} \) is the maximal subsumption-free subset of

\[
\mathcal{P}_0 \supseteq \cup \mathcal{P}_1 \supseteq \cup \{C \cup D \mid C \in \mathcal{P}_0 \bowtie, \ D \in \mathcal{P}_1 \bowtie\}.
\]
What data structure?

How to represent clause sets so our splitting formula may be realized efficiently?
Tries – Compact String Storage

- Dictionary data structure using rooted, labeled trees.

\{bear, beer, act, brain\} =

- Originated by E. Fredkin (1960)
Clauses translate to strings over the language \{+, -, 0\}.

- + at index \(i\) — Positive occurrence of literal \(v_i\)
- - at index \(i\) — Negative occurrence of literal \(v_i\)
- 0 at index \(i\) — \(v_i\) not in clause

Trailing 0s are implicit.

\[
\{ v_1, \neg v_3 \}
\]
Clauses as Strings

Clauses translate to strings over the language \{+, -, 0\}.

- + at index \(i\) — Positive occurrence of literal \(v_i\)
- - at index \(i\) — Negative occurrence of literal \(v_i\)
- 0 at index \(i\) — \(v_i\) not in clause

Trailing 0s are implicit.

\[
\{v_1, \neg v_3\} \rightarrow v_1 \lor 0 \lor \neg v_3
\]
Clauses as Strings

Clauses translate to strings over the language \{+, -, 0\}.

- \( \neg \) at index \( i \) — Positive occurrence of literal \( v_i \)
- \( \neg \) at index \( i \) — Negative occurrence of literal \( v_i \)
- 0 at index \( i \) — \( v_i \) not in clause

Trailing 0s are implicit.

\[
\{ v_1, \neg v_3 \} \rightarrow v_1 \lor 0 \lor \neg v_3 \rightarrow +0- 
\]
Clause Sets as Ternary Tries

\[
\{\{v_1, \neg v_3\}, \{-v_1, v_2\}, \{v_2, \neg v_3\}, \{-v_1, v_3\}\}
\]
Clause Sets as Ternary Tries

\[
\{\{v_1, \neg v_3\}, \{\neg v_1, v_2\}, \{v_2, \neg v_3\}, \{\neg v_1, v_3\}\}
\{+0-, -+, 0+-, -0+\}
\]
Clause Sets as Ternary Tries

\{\{v_1, \neg v_3\}, \{\neg v_1, v_2\}, \{v_2, \neg v_3\}, \{\neg v_1, v_3\}\}
\{+0-, -+, 0+-, -0+\}

Each branch represents a clause.
Assume clause sets prefix-free.
Advantages of Trie Representation

- Compact — common prefixes are shared
- Compressible into DAG form for storage
- Recursive structure allows efficient manipulation by recursive algorithms

Others, such as Reiter and de Kleer (1987), have used tries to represent sets of clauses.
Clause Tries

Construct trie recursively as:

What operations on tries are needed?

- Set Union
- Set Minus — $P_x \boxtimes = P_x \setminus P_x^\sqcup$.
- Subsumption — Determine $P_0^\sqcup, P_1^\sqcup$
- Clause Union — Like Cartesian product with union instead of pairing.
Subsumption operators

On two clause sets $A, B$,

$$Subsumed(A, B) = \{ C \in A \mid C \text{ is subsumed by some clause in } B \}$$

e.g. $\mathcal{P}_x \supseteq = Subsumed(\mathcal{P}_x, \mathcal{P}_{(1-x)})$
Subsumption operators

On two clause sets $A, B$,

$$\text{Subsumed}(A, B) = \{ C \in A \mid C \text{ is subsumed by some clause in } B \}$$

e.g. $\mathcal{P}_x \supseteq \text{Subsumed}(\mathcal{P}_x, \mathcal{P}_{(1-x)})$

$\text{SubsumedStrict}(A, B)$

— Variant where subsumption is strict, i.e. $\subset$
Pairwise Clause Union

On two clause sets $A$, $B$,

$$Unions(A, B) = \{ C \cup D \mid C \in A \text{ and } D \in B \}$$

e.g.

$$U = P_0^\sqcup \cup P_1^\sqcup \cup Unions(P_0^\otimes, P_1^\otimes)$$

$$\mathcal{U} = U \setminus SubsumedStrict(U, U) \text{ (remove subsumed clauses)}$$
Implementation

- "∪", "\", "Subsumed", "SubsumedStrict", and "Unions" are implemented on clause tries.
Implementation

- “∪”, “\”, “Subsumed”, “SubsumedStrict”, and “Unions” are implemented on clause tries.
- These drive prime implicate generation system written in Java.
Implementation

- “∪”, “\”, “Subsumed”, “SubsumedStrict”, and “Unions” are implemented on clause tries.
- These drive prime implicate generation system written in Java.
- Recursive implementation yields significant speedup over our previously published method.
Implementation

- “\( \cup \)” , “\( \setminus \)” , “Subsumed” , “SubsumedStrict” , and “Unions” are implemented on clause tries.
- These drive prime implicate generation system written in Java.
- Recursive implementation yields significant speedup over our previously published method.
- Old method iterates over pairs \((C, D)\), \(C \in \mathcal{P}_0\), \(D \in \mathcal{P}_1\)
Old/new runtime comparison

Old vs New pi-trie Algorithms on 15 var 3-CNF

- # of msecs vs # of clauses comparison
- Old pi-trie vs New pi-trie

Graph showing runtime comparison between Old and New pi-trie algorithms on 15 variable 3-CNF problems.
Recursive Subsumption on Tries

Algorithm 1: Subsumed(A,B)

if $A = \emptyset$ or $B = \emptyset$ then
  $T \leftarrow \emptyset$
else if $B = \{\{}\}$ then
  $T \leftarrow A$
else
  $T^+ \leftarrow \text{Subsumed}(A^+, B^+) \cup \text{Subsumed}(A^+, B^0)$
  $T^- \leftarrow \text{Subsumed}(A^-, B^-) \cup \text{Subsumed}(A^-, B^0)$
  $T^0 \leftarrow \text{Subsumed}(A^0, B^0)$

return $T$;

- Runtime for clause sets that are internally subsumption-free is difficult to pin down analytically.
Typical Iterative Subsumption

Algorithm 2: NaiveSubsumed(A, B)

\[ H \leftarrow \emptyset; \]
\[ \text{for } C \in A, D \in B \text{ do} \]
\[ \quad \text{if } D \text{ subsumes } C \text{ then } H \leftarrow H \cup \{C\}; \]
\[ \text{return } H; \]

- Iterates over pairs — quadratic runtime.
Subsumption with Shared Prefixes

\[ A = \{ \{ v_1, v_2, \neg v_3 \}, \{ v_1, v_2, v_4 \} \} \]
\[ B = \{ \{ v_1, v_4 \} \} \]

Subsumed\((A, B) = ?\)
Subsumption with Shared Prefixes

\[ A = \{\{v_1, v_2, \neg v_3\}, \{v_1, v_2, v_4\}\} \]
\[ B = \{\{v_1, v_4\}\} \]
\[ Subsumed(A, B) = ? \]

As strings,
\[ A = \{++-, ++0+\} \]
\[ B = \{+00+\} \]
Subsumption with Shared Prefixes

\[ A = \{\{v_1, v_2, \neg v_3\}, \{v_1, v_2, v_4\}\} \]
\[ B = \{\{v_1, v_4\}\} \]

Subsumed\((A, B) = ?\)

As strings,
\[ A = \{++-, ++0+\} \]
\[ B = \{+00+\} \]

++- Are either
++0+ of these
+00+ subsumed by this?
Subsumption with Shared Prefixes

\[ A = \{\{v_1, v_2, \neg v_3\}, \{v_1, v_2, v_4\}\} \]
\[ B = \{\{v_1, v_4\}\} \]
\[ \text{Subsumed}(A, B) =? \]

As strings,
\[ A = \{++-, ++0+\} \]
\[ B = \{+00+\} \]

++- \quad \{v_1, v_2, \neg v_3\}
++0+ \quad \{v_1, v_2, v_4\}
+00+ \quad \{v_1, v_4\}
Subsumption with Shared Prefixes

\[ A = \{\{v_1, v_2, \neg v_3\}, \{v_1, v_2, v_4\}\} \]
\[ B = \{\{v_1, v_4\}\} \]
\[ Subsumed(A, B) =? \]

As strings,
\[ A = \{++-, ++0+\} \]
\[ B = \{+00+\} \]

\[ ++- \quad \{v_1, v_2, \neg v_3\} \]
\[ ++0+ \quad \{v_1, v_2, v_4\} \]
\[ +00+ \quad \{v_1, v_4\} \]
Subsumption with Shared Prefixes

\[ A = \{\{v_1, v_2, \neg v_3\}, \{v_1, v_2, v_4\}\} \]
\[ B = \{\{v_1, v_4\}\} \]
\[ \text{Subsumed}(A, B) =? \]

As strings,
\[ A = \{++-, ++0+\} \]
\[ B = \{+00+\} \]

++-
\[ \{v_1, v_2, \neg v_3\} \]

++0+
\[ \{v_1, v_2, v_4\} \]

+00+
\[ \{v_1, v_4\} \]
Subsumption with Shared Prefixes

\[ A = \{ \{ v_1, v_2, \neg v_3 \}, \{ v_1, v_2, v_4 \} \} \]
\[ B = \{ \{ v_1, v_4 \} \} \]

\[ \text{Subsumed}(A, B) =? \]

As strings,
\[ A = \{ ++-, ++0+ \} \]
\[ B = \{ +00+ \} \]

++- \{ v_1, v_2, \neg v_3 \} Not subsumed
++0+ \{ v_1, v_2, v_4 \}
+00+ \{ v_1, v_4 \}
Subsumption with Shared Prefixes

\[ A = \{ \{v_1, v_2, \neg v_3\}, \{v_1, v_2, v_4\}\} \]
\[ B = \{\{v_1, v_4\}\} \]

\[ \text{Subsumed}(A, B) = ? \]

As strings,
\[ A = \{++-, ++0+\} \]
\[ B = \{+00+\} \]

++- \( \{v_1, v_2, \neg v_3\} \) Not subsumed
++0+ \( \{v_1, v_2, v_4\} \) Subsumed
+00+ \( \{v_1, v_4\} \)
Subsumption with Shared Prefixes

\[ A = \{ \{ v_1, v_2, \neg v_3 \}, \{ v_1, v_2, v_4 \} \} \]

\[ B = \{ \{ v_1, v_4 \} \} \]

Subsumed \((A, B) =?\)

As strings,

\[ A = \{ ++-, ++0+ \} \]

\[ B = \{ +00+ \} \]

Subsumed \((A, B) = \{ ++0+ \} = \{ \{ v_1, v_2, v_4 \} \} \)
Surprising Efficiencies 1

- The naïve, iterative subsumption routine is $O(n^2)$.
- Experiments and analysis suggest that the recursive and iterative subsumption routines differ asymptotically.
Below lemma reinforces the hypothesis of differing asymptotic runtime.

**Lemma**

*Subsumed*, when applied to two full ternary tries of depth $h$ and combined size $n = 2\left(\frac{3^{h+1}-1}{2}\right)$, runs in time $O\left(n^{\log_3 5}\right) \approx O\left(n^{1.465}\right)$. 
Clause Filtering

What if we only wish to compute a subset of prime implicates?

Say ones which are short, or contain only certain literals.
Let $Q$ be a predicate on clauses and $C, D$ clauses s.t. $D$ subsumes $C$. If $Q$ is subsumption invarient, then $Q(C) \rightarrow Q(D)$, i.e. truth of $Q$ is preserved on subsuming clauses. Conversely, falsity of $Q$ is preserved on subsumed clauses.
Subsumption Invariance

Let $Q$ be a predicate on clauses and $C, D$ clauses s.t. $D$ subsumes $C$.

If $Q$ is subsumption-invariant, then $Q(C) \rightarrow Q(D)$

i.e. truth of $Q$ is preserved on subsuming clauses.
Subsumption Invariance

Let \( Q \) be a predicate on clauses and \( C, D \) clauses s.t. \( D \) subsumes \( C \)

If \( Q \) is \textbf{subsumption-invariant}, then \( Q(C) \rightarrow Q(D) \)

i.e. truth of \( Q \) is preserved on subsuming clauses.

Conversely, falsity of \( Q \) is preserved on subsumed clauses.
Subsumption Invarience - Clause Filters

Examples:
Q = “Clause size ≤ k”
Q = “Does not contain literals ℓ₁, . . . , ℓₙ”
Q = “Is a Horn clause”
Subsumption Invariance - Clause Filters

Examples:
\[ Q = \text{"Clause size } \leq k\text{"} \]
\[ Q = \text{"Does not contain literals } \ell_1, \ldots, \ell_n\text{"} \]
\[ Q = \text{"Is a Horn clause"} \]

Let a **clause filter** \( \phi_Q \) be a function on clause sets such that

\[ \phi_Q(F) = \{ C \in F \mid Q(C) \} \]
Monotonicity of Splitting Formula

In the splitting formula

\[ \mathcal{P}(\mathcal{F}) = (v \lor \mathcal{P}_0^{\boxtimes}) \cup (\neg v \lor \mathcal{P}_1^{\boxtimes}) \cup \mathcal{U}, \]

any clause of \( \mathcal{P}(\mathcal{F}) \) is formed from subsuming clauses in \( \mathcal{P}_0, \mathcal{P}_1 \)
Monotonicity of Splitting Formula

In the splitting formula

\[ P(\mathcal{F}) = (v \lor P_0^\Box) \cup (\neg v \lor P_1^\Box) \cup U, \]

any clause of \( P(\mathcal{F}) \) is formed from subsuming clauses in \( P_0, P_1 \)

We will denote the above splitting formula as

\[ P(\mathcal{F}) = S(P_0, P_1) \]
Incremental Filtering

Thus for a subsumption-invariant $Q$,

\[
\phi_Q(\mathcal{P}(\mathcal{F})) = \phi_Q(\mathcal{S}(\mathcal{P}_0, \mathcal{P}_1)) = \phi_Q(\mathcal{S}(\phi_Q(\mathcal{P}_0), \phi_Q(\mathcal{P}_1)))
\]
Incremental Filtering

Thus for a subsumption-invariant \( Q \),

\[
\phi_Q(\mathcal{P}(\mathcal{F})) \\
= \phi_Q( S(\mathcal{P}_0, \mathcal{P}_1) ) \\
= \phi_Q( S( \phi_Q(\mathcal{P}_0), \phi_Q(\mathcal{P}_1) ) )
\]

Filtering before applying \( S \) may greatly reduce processing time — repeated applications of \( S \) are the bottleneck
Effect of Incremental Filtering

Runtime Comparison for Random 13-var 3-CNF (average of 20 trials)

- pi-trie
- pi-trie filtered, forbidden literals
- pi-trie filtered, max 2 literals
The End

Thanks much for listening. Questions?