Trie Based Subsumption and Improving the pi-Trie Algorithm

Andrew Matusiewicz, Neil V. Murray, Erik Rosenthal

August 25, 2010
What are prime implicates and why care?

- Smallest, simplest disjunctions implied by a statement
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 $\land$ (and), $\lor$ (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:

$$ (v_1 \lor \neg v_3 \lor v_2 \lor v_9) = \{v_1, \neg v_3, v_2, v_9\} $$
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 $F$.

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

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

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

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

1. Split $\mathcal{F}$ by substitution of 0, 1 for $\nu$, 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 $P(F)$ from $F$,

1. Split $F$ by substitution of 0, 1 for $v$, yielding $F_0, F_1$.
2. By recursion obtain $P_0 = P(F_0)$ and $P_1 = P(F_1)$.
3. Combine $P_0, P_1$ with splitting formula to obtain $P(F)$. 

Base cases are boolean constants.
To calculate $P(F)$ from $F$,

1. Split $F$ by substitution of 0, 1 for $v$, yielding $F_0, F_1$.
2. By recursion obtain $P_0 = P(F_0)$ and $P_1 = P(F_1)$.
3. Combine $P_0, P_1$ with splitting formula to obtain $P(F)$.

Base cases are boolean constants.
Subsumption Classes of Primes

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

Let \( x \in \{0, 1\} \)

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

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

$\mathcal{P}_x = \mathcal{P}(\mathcal{F}[x/\nu])$ — 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 \( x \in \{0, 1\} \)

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

\[ P_x = P_x^\sqcup \cup P_x^\triangleleft \] — Split \( P_x \) into disjoint clause sets

\[ P_x^\sqcup \] — Clauses in \( P_x \) subsumed by some clause in \( P_{(1-x)} \)
Subsumption Classes of Primes

Let \( x \in \{0, 1\} \)

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

\[ \mathcal{P}_x = \mathcal{P}_x^\supseteq \cup \mathcal{P}_x^\subseteq \] — Split \( \mathcal{P}_x \) into disjoint clause sets

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

\[ \mathcal{P}_x^\subseteq \] — Remainder of \( \mathcal{P}_x \), 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 $\mathcal{P}_0 \trianglerighteq, \mathcal{P}_1 \trianglerighteq, \mathcal{P}_0 \pitchfork, \mathcal{P}_1 \pitchfork$. 

From this:

$$
\mathcal{P}(F) = \left( v \lor \mathcal{P} \pitchfork \right) \cup \left( \neg v \lor \mathcal{P} \trianglerighteq \right) \cup U
$$

Where $U$ is the maximal subsumption-free subset of $\mathcal{P}_0 \cup \mathcal{P}_1 \cup \{ \mathcal{C} \cup \mathcal{D} | \mathcal{C} \in \mathcal{P} \pitchfork, \mathcal{D} \in \mathcal{P} \trianglerighteq \}.$
Splitting Formula for Primes

Given $\mathcal{P}_0, \mathcal{P}_1$, we find subsumption classes $\mathcal{P}_0^\triangleright, \mathcal{P}_1^\triangleright, \mathcal{P}_0^\ll, \mathcal{P}_1^\ll$.

From this:

$$\mathcal{P}(F) = (v \lor \mathcal{P}_0^\ll) \cup (\neg v \lor \mathcal{P}_1^\ll) \cup \mathcal{U}$$

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

$$\mathcal{P}_0^\triangleright \cup \mathcal{P}_1^\triangleright \cup \{C \cup D \mid C \in \mathcal{P}_0^\ll, \ D \in \mathcal{P}_1^\ll\}$$
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.

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

▶ Originated by E. Fredkin (1960)
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.

\[
\{v_1, \neg v_3\}
\]
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 \( \{+, -, 0\} \).

- \(+ \) 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.

\[
\{v_1, \neg v_3\} \rightarrow v_1 \lor 0 \lor \neg v_3 \rightarrow +0-\
\]
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\}\}
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+\neg, -+, 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 Stephan Schulz (2004), have used tries to represent sets of clauses.
Clause Tries

Construct trie recursively as:

```
+ - 0
P ⊙ ◁
0 P ⊙ ◁
1 U
```

What operations on tries are needed?

- Set Union
- Set Minus — $\mathcal{P}_x^\boxright = \mathcal{P}_x \setminus \mathcal{P}_x^\boxleft$.
- Subsumption — Determine $\mathcal{P}_0^\supseteq, \mathcal{P}_1^\supseteq$
- 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. $\mathcal{P}^3_x = Subsumed(\mathcal{P}_x, \mathcal{P}_{(1-x)})$
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. $\subsetneq$
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.

$$U = \mathcal{P}_0 \cup \mathcal{P}_1 \cup Unions(\mathcal{P}_0, \mathcal{P}_1)$$

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

- “∪”, “\”, “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
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.
Algorithm 2: NaiveSubsumed($A,B$)

$H \leftarrow \emptyset$;

for $C \in A$, $D \in B$ do

if $D$ subsumes $C$ then $H \leftarrow H \cup \{C\}$;

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

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

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

++-

\(\{v_1, v_2, \neg v_3\}\)

++0+

\(\{v_1, v_2, v_4\}\)

+00+

\(\{v_1, v_4\}\)
Subsumption with Shared Prefixes

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

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

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

+++ \( \{ v_1, v_2, \neg v_3 \} \)
++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

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

\textit{Subsumed}(A, B) = ?

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

\[ +++ \quad \{v_1, v_2, \neg v_3\} \text{ Not subsumed} \]
\[ ++0+ \quad \{v_1, v_2, v_4\} \text{ Subsumed} \]
\[ +00+ \quad \{v_1, v_4\} \]
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,
\[ A = \{++-, ++0+\} \]
\[ B = \{+00+\} \]

\[ \text{Subsumed}(A, B) = \{++0+\} = \{ \{ v_1, v_2, v_4 \} \} \]
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 $O\left(n^{\frac{\log 5}{\log 3}}\right) \approx O(n^{1.465})$. 

---

**Surprising Efficiencies 2**
The End

Thanks much for listening. Questions?