GENERALIZED INSERTION

William Bricken
January 1998, revised January 2002

The Boundary Insertion algorithm is **a general proof technique** for logic, functionally equivalent to the four other known techniques (truth tables, natural deduction, resolution, and algebraic substitution). Unlike the other techniques, polynomially bounded Insertion is quite comprehensive.

Other nice features of Insertion:

1. It is algorithmic (in contrast to natural deduction).

2. The algorithm is very efficient, never generating new forms which increase the search space (in contrast to resolution).

3. It uses virtual forms (i.e. queries) rather than assertions. A positive answer returns a reduced form directly.

4. It has powerful filters which optimize its application.

5. It returns a minimal form (like algebraic approaches) rather than a Boolean failure.

6. Like all boundary techniques, erasure of irrelevancies is the primary deductive transformation. [The void has pleasant computational properties: it takes up no memory and requires no time to process.]

The Axioms of Boundary Logic

( ) A = ( )	DOMINION	[erase context]
((A)) = A	INVOLUTION	[erase boundaries]
A (A B) = A (B)	PERVASION	[erase forms]

All boundary transforms are deep: erasure is independent of depth of nesting. For example:

A (B (C (A D))) = A (B (C (D))) -pervasion

Theorems to be used later:

(( ) A) = <void>	OCCLUSION
(A) (A B) = (A)	SUBSUMPTION

Subsumption is a theorem only if you view boundary forms as linear representations. Spatially, Subsumption is identical to Pervasion. The difference is which side of the boundary you view from, A or (A). [Yes, observer viewpoint is an elementary operator in boundary math].

The complexity in SAT (and in TAUT) stems from Boolean tangles which cannot be standardized. That is, search is required when you do not know which direction to untangle a form.

There are **two Boolean tangles**:

(A B) ((A)(B))) = ((A (B)) ((A) B)) PIVOT

A ((B)(C)) = ((A B) (A C)) DISTRIBUTION

The deep problem with these forms is that the capital letters are arbitrarily complex forms themselves, which are required to be equivalent in value only, not in structure. Therefore, a structural approach such as pattern-matching fails. For example, distribution can be recursively tangled within itself:

X ((Y)(Z)) = ((X Y) (X' Z))

X = ((A B) (A C))
X' = A ((B)(C))

((A B) (A C)) ((Y)(Z)) = ((**((A B) (A C))** Y) (**A ((B)(C))** Z))

In this form, pattern-matching for distribution becomes exponential; it fails at the top level, and although it could be applied recursively, the direction of application (toward X or toward X') is a guess.

What follows are some technical examples which came up during this adventure of untangling.

ACT I: Binate Theorems

Distribution (and pivot) are theorems based on Pervasion. Perhaps their tangledness can be reduced. I showed (in 1983) that distribution is *not* an axiom of Boolean algebra, with the following proof:

Algebraic proof of distribution 1:

```
((a   b             ) (a        c   ))        left-hand-side
((a   b             ) (a (     (c)) ))        +involution
((a   b             ) (a ((a b)(c)) ))        +pervasion*
((a   b             ) (a ((  b)(c)) ))        -pervasion
((a ((b)           )) (a ((  b)(c)) ))        +involution
((a ((b)(a ((b)(c))))) (a ((  b)(c)) ))        +pervasion*
((a ((b)(  (   (c))))) (a ((  b)(c)) ))        -pervasion
((a ((b)(       c  ))) (a ((  b)(c)) ))        -involution
((a ((b)(       c  )))               )        -pervasion
  a ((b)(       c  ))                         -involution
```

Algebraic proof 2 (going the other direction):

```
  a                           ((  b)(  c))     right-hand-side
  a                           ((a b)(a c))     +pervasion*
((a)                        ) ((a b)(a c))     +involution
((a) (a b)(a c)             ) ((a b)(a c))     +subsumption**
((a) (a b)(a c) ((a b)(a c)) ) ((a b)(a c))     +pervasion
((a) (a b)(a c) (          ) ) ((a b)(a c))     -pervasion
                               ((a b)(a c))     -occlusion
```

The shortest version:

```
  a               ((  b)(  c))                rhs
  a               ((a b)(a c))                +insertion
((a)            ) ((a b)(a c))                +involution
((a) ((a b)(a c))) ((a b)(a c))               +reinsertion*
((a) (          )) ((a b)(a c))               -deep subsumption
                   ((a b)(a c))               -occlusion
```

The problem with these proofs is that, like natural deduction, they require creativity. The transforms marked "+" are constructive, adding form rather than reducing form. Those marked "*" are insightful. Thus these proofs require search if implemented as algorithms.

ACT II: Virtual Insertion

Insertion (1996) is an efficient algorithmic proof technique. Here distribution is proved by showing the implication in both directions:

```
A ((B)(C)) = ((A B)(A C))

     -->:        [ A ((B)(C)) ]  ((A B)(A C))

         and

     <--:          A ((B)(C))  [ ((A B)(A C)) ]
```

Proof:

```
[ ((A B)(A C)) ] A ((B)(C))                    <--
   (A B)(A C)    A ((B)(C))                    -inv
   (  B)(  C)    A ((B)(C))                    -per
   (  B)(  C)    A (      )                    -per
                   (      )                    -dom
```

Below, **the inserted form is *virtual***, it is not structurally written within the form it is inserted into. Rather it is a query that changes depending on its context. Virtual forms are set off using carets, ^^. The form being inserted is the **put-form**, the form being inserted into is the **into-form**.

Version 1:

```
[ A ((B)(C)) ] ((A B)               (A C               ))   -->
[ A ((B)(C)) ] ((A B ^(A ((B)(C)))^)(A C ^(A ((B)(C)))^))   +ins
[ A ((B)(C)) ] ((A B ^(  (( )(C)))^)(A C ^(  ((B)( )))^))   -per
[ A ((B)(C)) ] ((A B ^(          )^)(A C ^(          )^))   -occ
[ A ((B)(C)) ] (                                    )   -occ
                (                                    )   -dom
```

Version 2:

```
[ A ((B)(C))              ] ((A B)(A C))            -->
[ A ((B)(C)) ^((A B)(A C))^] ((A B)(A C))           +ins
[ A ((B)(C)) ^((  B)(  C))^] ((A B)(A C))           -per
[ A          ^((  B)(  C))^] ((A B)(A C))           -per bound
[ A                        ] ((A B)(A C))           -virt
[ A                        ] (          )           -sub
                               (          )           -dom
```

Version 3:

```
[ A ((B)(C))                ] ((A B)(A C))            -->
[ A ((B)(C)) ^((A B)(A C))^] ((A B)(A C))            +ins
[ A ((B)(C)) ^((  B)(  C))^] ((A B)(A C))            -per
[ A ((B)(C) ^((  B)(  C))^)] ((A B)(A C))            +ins
[ A ((B)(C) ^(           )^)] ((A B)(A C))           -per
[ A                         ] ((A B)(A C))           -occ
[ A                         ] (          )           -sub
                                (          )           -dom
```

In Version 1, the put-form is placed into all other forms at the level of the first occurrence of literals. These literals reduce the virtual put-form, eventually terminating with either Dominion (which reduces the context) or any other form, which means no reduction is possible.

Version 2 demonstrates that it doesn't matter which form is selected as the put-form and which as the into-form. Thus no dependency on lucky guesses.

Version 3 makes the important point that all action can be seen as reduction at the level of literals, thus no bounded forms (i.e. arbitrarily complex forms) ever need to be matched.

Also, Insertion is **not search**, it is simply percolation of a selected (virtual) put-form down the graph of the into-form, with clear termination conditions at the bottom (at literals).

ACT III: Boolean Minimization

Here is an example of Insertion for the purpose of Boolean minimization. The key idea is to avoid both of the entangled transforms (pivot and distribution), since they are expensive, generating additional forms for memory and processing:

Minimize: ((a b) ((a c)(d (b))))

poor approach using distribution that adds structure:

```
( (a b) ((a c      ) (d (b)      )) )          prob
(       ((a c (a b)) (d (b) (a b))) )          +dist [additive]
(       ((a c (  b)) (d (b)      )) )          -per, -sub
( (b)   ((a c      ) (d          )) )          -dist
((b) ((d) (a c)))
```

*better approach using *virtual* distribution:*

```
( (a b)     ((a c        )(d (b)        )) )    prob
( (a b)     ((a c ^(a b)^)(d (b) ^(a b)^)) )    +ins
( (a b)     ((a c ^(  b)^)(d (b) ^(a b)^)) )    -per
( (a b) (b) ((a c        )(d     ^(a b)^)) )    -dist [virtual]
(       (b) ((a c        )(d     ^(a b)^)) )    -sub
((b) ((d) (a c)))                               -virt
```

Here's a proof of the Resolution Theorem using *virtual* subsumption.

```
      ((a b) ((a) c))  =  ((a b) ((a) c) (b c))             RESOLUTION

            ((a b) ((a) c        ) (b c        ))           rhs
            ((a b) ((a) c ^(a b)^) (b c ^(a b)^))           +ins
            ((a b) ((a) c        ) (b c ^(a  )^))           -virt, -per
            ((a b) ((a) c        )              )           -sub [virtual]
            ((a b) ((a) c))
```

Note an algorithmic danger: you may not engage in doubly virtual insertions.

```
                  (A B          )
                  (A B          ) ^(A B)^                   +ins
                  (A B ^^(A B)^^) ^(A B)^                   +reins (NO!)
                  (A B ^^(   )^^) ^(A B)^                   -per
                                  ^(A B)^                   -occ
                                   <void>                   -virt
```

[It is a metaphysical fact that you cannot dream within a dream. That is, the homunculus cannot logically create an infinite regress.]

ACT IV: Computational Complexity

Finally, my regression tests included some problems which were apparently not susceptible to the Insertion technique. They yielded when Insertion was generalized from Bound Insertion to **Set Insertion**. That is, when there are more than two bounds in a form, rather than selecting a single bound as the put-form, just insert all of them. An example:

```
      (if (and (iff A B) (iff B C)) (iff A C))        TRANSITIVITY OF IFF
```

Note that there are four bounds in the transcribed form. Inserting any one of them into the others achieves no reduction. But inserting all of them into the last one works. This is because the reduction dependency is spread across several bounds.

```
      (A C) ((A)(C)) ((A B) ((A)(B))) ((B C) ((B)(C)))            transcribe

Let X = (A C) ((A)(C)) ((A B) ((A)(B)))

      X ((B C ^             X                 ^) ((B)(C)))        +ins
      X ((B C ^(A C) ((A)(C)) ((A B) ((A)(B)))^) ((B)(C)))        subst
      X ((B C ^(A  ) ((A)( )) ((A  ) ((A)( )))^) ((B)(C)))        -per
      X ((B C ^(A  )          ((A  )         )^) ((B)(C)))        -occ
      X ((B C ^(A  )          (              )^) ((B)(C)))        -per
      X (                                    ((B)(C)))           -occ
      X                                       (B)(C)             -inv

      (A C) ((A)(C)) ((A B) ((A)(B)))          (B)(C)            subst
      (A C) ((A)   ) ((A B) ((A)   ))          (B)(C)            -per
      (A C)   A      ((A B)   A     )          (B)(C)            -inv
      (A C)   A      (        A     )          (B)(C)            -sub
      (A C)   A      (              )          (B)(C)            -per
                     (              )                            -dom
```

Larger example

The **pigeon-hole problem** is cited in the literature as being one that is intractable for tautology checkers. The below formulation reads: "There are three pigeons (first index) and two holes (second index). Not every pigeon can find a hole."

3-2 Pigeon-hole:

```
        (AND (AND (OR 11 12) (OR 21 22) (OR 31 32))          [find a hole]
             (AND (AND (OR (NOT 11) (NOT 21))
                       (OR (NOT 11) (NOT 31))                [don't share]
                       (OR (NOT 21) (NOT 31)))
                  (AND (OR (NOT 12) (NOT 22))
                       (OR (NOT 12) (NOT 32))
                       (OR (NOT 22) (NOT 32)))))


            (11 12) (21 22) (31 32)                          transcribe
              ((11) (21)) ((11) (31)) ((12) (22))              and -inv
              ((12) (32)) ((21) (31)) ((22) (32))
```

Since there are nine forms at the top level of the transcription, it seems like a good idea to use an insertion filter, but this is exactly what the pigeon-hole problem thwarts. In general, brute force is always better than subtlety (!). In particular, for boundary techniques the cost of evaluating a filter is equal to the cost of applying a transform. The extreme symmetry in the reduction is characteristic of intractable tautological problems.

(11 12) (21 22) (31 32) ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32))

Insert the context into each form:

```
( 11  12  ^       (21 22) (31 32) ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32))^ )
( 21  22  ^(11 12)        (31 32) ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32))^ )
( 31  32  ^(11 12) (21 22)        ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32))^ )
((11)(21) ^(11 12) (21 22) (31 32)           ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32))^ )
((11)(31) ^(11 12) (21 22) (31 32) ((11)(21))           ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32))^ )
((12)(22) ^(11 12) (21 22) (31 32) ((11)(21)) ((11)(31))           ((12)(32)) ((21)(31)) ((22)(32))^ )
((12)(32) ^(11 12) (21 22) (31 32) ((11)(21)) ((11)(31)) ((12)(22))           ((21)(31)) ((22)(32))^ )
((21)(31) ^(11 12) (21 22) (31 32) ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32))           ((22)(32))^ )
((22)(32) ^(11 12) (21 22) (31 32) ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31))           ^ )
```

Erase literals and occlude (do not subsume!):

```
( 11  12  ^       (21 22) (31 32)                                           ((21)(31)) ((22)(32))^ )
( 21  22  ^(11 12)        (31 32)           ((11)(31))           ((12)(32))                       ^ )
( 31  32  ^(11 12) (21 22)        ((11)(21))           ((12)(22))                                 ^ )
((11)(21) ^(11 12) (21 22) (31 32)                31   ((12)(22)) ((12)(32))       31   ((22)(32))^ )
((11)(31) ^(11 12) (21 22) (31 32)       21            ((12)(22)) ((12)(32))   21       ((22)(32))^ )
((12)(22) ^(11 12) (21 22) (31 32) ((11)(21)) ((11)(31))                32   ((21)(31))       32  ^ )
((12)(32) ^(11 12) (21 22) (31 32) ((11)(21)) ((11)(31))      22             ((21)(31))   22      ^ )
((21)(31) ^(11 12) (21 22) (31 32)   11         11       ((12)(22)) ((12)(32))           ((22)(32))^ )
((22)(32) ^(11 12) (21 22) (31 32) ((11)(21)) ((11)(31))   12         12       ((21)(31))           ^ )
```

Erase newly exposed literals (recur on literal extract):

```
( 11  12  ^       (21 22) (31 32)                                           ((21)(31)) ((22)(32))^ )
( 21  22  ^(11 12)        (31 32)           ((11)(31))           ((12)(32))                       ^ )
( 31  32  ^(11 12) (21 22)        ((11)(21))           ((12)(22))                                 ^ )
((11)(21) ^(11 12) (21 22) (   32)                31   ((12)(22)) ((12)(32))           ((22)(32))^ )
((11)(31) ^(11 12) (   22) (31 32)       21            ((12)(22)) ((12)(32))           ((22)(32))^ )
((12)(22) ^(11 12) (21 22) (31   ) ((11)(21)) ((11)(31))                32   ((21)(31))           ^ )
((12)(32) ^(11 12) (21   ) (31 32) ((11)(21)) ((11)(31))      22             ((21)(31))           ^ )
((21)(31) ^(   12) (21 22) (31 32)   11                    ((12)(22)) ((12)(32))           ((22)(32))^ )
((22)(32) ^(11   ) (21 22) (31 32) ((11)(21)) ((11)(31))   12                ((21)(31))           ^ )
```

Erase newly exposed literals and clarify (recur):

```
( 11  12  ^       (21 22) (31 32)                                           ((21)(31)) ((22)(32))^ )
( 21  22  ^(11 12)        (31 32)           ((11)(31))           ((12)(32))                       ^ )
( 31  32  ^(11 12) (21 22)        ((11)(21))           ((12)(22))                                 ^ )
((11)(21) ^(11 12) (21 22) (   32)                31   ((12)(22))   12                   22      ^ )
((11)(31) ^(11 12) (   22) (31 32)       21              12       ((12)(32))                32  ^ )
((12)(22) ^(11 12) (21 22) (31   ) ((11)(21))   11                        32    21               ^ )
((12)(32) ^(11 12) (21   ) (31 32)   11       ((11)(31))       22                   31           ^ )
((21)(31) ^(   12) (21 22) (31 32)   11                        22        32           ((22)(32))^ )
((22)(32) ^(11   ) (21 22) (31 32)       21        31    12                ((21)(31))           ^ )
```

Erase newly exposed literals and occlude (recur):

```
( 11  12  ^       (21 22) (31 32)                                    ((21)(31)) ((22)(32))^ )
( 21  22  ^(11 12)        (31 32)          ((11)(31))          ((12)(32))                    ^ )
( 31  32  ^(11 12) (21 22)        ((11)(21))          ((12)(22))                             ^ )
((11)(21) ^(11   ) (21   ) (   32)                31              12                  22     ^ )
((11)(31) ^(11   ) (   22) (31   )      21              12                                 32  ^ )
((12)(22) ^(   12) (   22) (31   )           11                      32    21              ^ )
((12)(32) ^(   12) (21   ) (   32)   11                     22                  31          ^ )
((21)(31) ^(   12) (21   ) (31   )   11                     22        32                    ^ )
((22)(32) ^(11   ) (   22) (   32)      21        31    12                                  ^ )
```

Rewrite; note that the symmetries of the remaining forms mirror the symmetries of the original problem, but with less complexity.

```
( 11  12  ^(21 22) (31 32) ((21)(31)) ((22)(32))^ )
( 21  22  ^(11 12) (31 32) ((11)(31)) ((12)(32))^ )
( 31  32  ^(11 12) (21 22) ((11)(21)) ((12)(22))^ )
((11)(21) ^(11) (21) (32) 31 12 22^ )                          do not subsume (11) (21)
((11)(31) ^(11) (22) (31) 21 12 32^ )
((12)(22) ^(12) (22) (31) 11 32 21^ )
((12)(32) ^(12) (21) (32) 11 22 31^ )
((21)(31) ^(12) (21) (31) 11 22 32^ )
((22)(32) ^(11) (22) (32) 21 31 12^ )
```

Insert to next depth. Now the subsumable literals will trigger Occlusion:

```
(11 12 ^(21 22) (31 32) ((21)(31)) ((22)(32))^ )
(21 22 ^(11 12) (31 32) ((11)(31)) ((12)(32))^ )
(31 32 ^(11 12) (21 22) ((11)(21)) ((12)(22))^ )
((11 ^(11) (21) (32) 31 12 22^)  (21 ^(11) (21) (32) 31 12 22^) )
((11 ^(11) (22) (31) 21 12 32^)  (31 ^(11) (22) (31) 21 12 32^) )
((12 ^(12) (22) (31) 32 11 21^)  (22 ^(12) (22) (31) 32 11 21^) )
((12 ^(12) (21) (32) 22 11 31^)  (32 ^(12) (21) (32) 22 11 31^) )
((21 ^(12) (21) (31) 11 22 32^)  (31 ^(12) (21) (31) 11 22 32^) )
((22 ^(11) (22) (32) 12 21 31^)  (32 ^(11) (22) (32) 12 21 31^) )
```

Extract literals:

```
(11 12 ^(21 22) (31 32) ((21)(31)) ((22)(32))^ )
(21 22 ^(11 12) (31 32) ((11)(31)) ((12)(32))^ )
(31 32 ^(11 12) (21 22) ((11)(21)) ((12)(22))^ )
((11 ^(  ) (21) (32) 31 12 22^)  (21 ^(11) (  ) (32) 31 12 22^) )
((11 ^(  ) (22) (31) 21 12 32^)  (31 ^(11) (22) (  ) 21 12 32^) )
((12 ^(  ) (22) (31) 32 11 21^)  (22 ^(12) (  ) (31) 32 11 21^) )
((12 ^(  ) (21) (32) 22 11 31^)  (32 ^(12) (21) (  ) 22 11 31^) )
((21 ^(12) (  ) (31) 11 22 32^)  (31 ^(12) (21) (  ) 11 22 32^) )
((22 ^(11) (  ) (32) 12 21 31^)  (32 ^(11) (22) (  ) 12 21 31^) )
```

Finally Occlude the forms in the last six lines.

```
(11 12 ^(21 22) (31 32) ((21)(31)) ((22)(32))^ )
(21 22 ^(11 12) (31 32) ((11)(31)) ((12)(32))^ )
(31 32 ^(11 12) (21 22) ((11)(21)) ((12)(22))^ )
(                                                )
(                                                )
(                                                )
(                                                )
(                                                )
(                                                )
```

Note that symmetries again create multiple redundant reductions to ( ). Technically any one of these could have reduced the problem to True, since each Occludes the remaining forms. One approach that recognizes these redundant opportunities is to conduct the reduction in a more abstract formulation that condenses cyclic symmetries.

Set Insertion

In summary, set insertion is the boundary between polynomial and NP problems. This is illustrated by a final example.

Let's select a very friendly representation of P=?=NP. A symmetry of the form for three-coloring-a-tetrahedron, 3TET, has nice TAUT properties. The problem formulation says: "Each vertex has one of three colors, and no adjacent vertexes have the same color."

Here it is in a simple example, ***3TET-toplevel***:

```
      (B1 B2) (B1 B3) (B1 B4) (B1 G1) (B1 R1) (B2 B3)
      (B2 B4) (B2 G2) (B2 R2) (B3 B4) (B3 G3) (B3 R3)
      (B4 G4) (B4 R4) (G1 G2) (G1 G3) (G1 G4) (G1 R1)          [not adjacent]
      (G2 G3) (G2 G4) (G2 R2) (G3 G4) (G3 R3) (G4 R4)
      (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
      ((B1)(G1)(R1)) ((B2)(G2)(R2)) ((B3)(G3)(R3)) ((B4)(G4)(R4)) [has color]
```

What is nice is that there are a very few negated literals.

The INSERT algorithm has predictable properties when dealing with CNF forms. The reduction action is almost always associated with the negated literals. The lack of negated literals in the predominance of other clauses keeps them from interacting and thus keeps reduction well constrained

Now the engineering question: what kind of tangle is holding up this form?

We know that the eventual collapse will be caused by one of the boundary forms (disjunctive clauses) becoming a mark, ( ), the mark of TAUT.

Simple set-insertion yields the reduction that follows. Varieties of the symmetry

```
            (<letter.number> <different-letter.same-number>)
```

vanish. An example on one of the symmetric cases:

Set insertion into (G4 R4):

```
      (G4 R4
            ^(B1 B2) (B1 B3) (B1 B4) (B1 G1) (B1 R1) (B2 B3)
             (B2 B4) (B2 G2) (B2 R2) (B3 B4) (B3 G3) (B3 R3)
             (B4 G4) (B4 R4) (G1 G2) (G1 G3) (G1 G4) (G1 R1)
             (G2 G3) (G2 G4) (G2 R2) (G3 G4) (G3 R3)
             (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
             ((B1)(G1)(R1)) ((B2)(G2)(R2)) ((B3)(G3)(R3)) ((B4)(G4)(R4))^)

      (G4 R4
            ^(B1 B2) (B1 B3) (B1 B4) (B1 G1) (B1 R1) (B2 B3)
             (B2 B4) (B2 G2) (B2 R2) (B3 B4) (B3 G3) (B3 R3)
             (B4   ) (B4   ) (G1 G2) (G1 G3) (G1   ) (G1 R1)
             (G2 G3) (G2   ) (G2 R2) (G3   ) (G3 R3)
             (R1 R2) (R1 R3) (R1   ) (R2 R3) (R2   ) (R3   )
             ((B1)(G1)(R1)) ((B2)(G2)(R2)) ((B3)(G3)(R3)) ((B4)(  )(  ))^)

      (G4 R4
            ^(B1 B2) (B1 B3) (B1 B4) (B1 G1) (B1 R1) (B2 B3)
             (B2 B4) (B2 G2) (B2 R2) (B3 B4) (B3 G3) (B3 R3)
             (B4   )         (G1 G2) (G1 G3) (G1   ) (G1 R1)
             (G2 G3) (G2   ) (G2 R2) (G3   ) (G3 R3)
             (R1 R2) (R1 R3) (R1   ) (R2 R3) (R2   ) (R3   )
             ((B1)        ) ((B2)        ) ((B3)        )              ^)

      (G4 R4
            ^(     ) (     ) (   B4) (   G1) (   R1) (     )
             (   B4) (   G2) (   R2) (   B4) (   G3) (   R3)
             (B4   )         (G1 G2) (G1 G3) (G1   ) (G1 R1)
             (G2 G3) (G2   ) (G2 R2) (G3   ) (G3 R3)
             (R1 R2) (R1 R3) (R1   ) (R2 R3) (R2   ) (R3   )
               B1             B2             B3                        ^)

      (G4 R4
            ^(  )^ )

      <void>
```

Eliminating all of the subforms that have the symmetry

```
(<letter.number> <different-letter.same-number>)
```

leaves the following form, which is also TAUT but simpler -- call this ***3TET-newtoplevel***:

```
(B1 B2) (B1 B3) (B1 B4)                 (B2 B3)
(B2 B4)                 (B3 B4)
                (G1 G2) (G1 G3) (G1 G4)
(G2 G3) (G2 G4)         (G3 G4)
(R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
((B1)(G1)(R1)) ((B2)(G2)(R2)) ((B3)(G3)(R3)) ((B4)(G4)(R4))
```

[Incidentally, this tautology may map onto a different NP-complete problem.]

Now it is time to recur. We know that the first pass eliminated all other reductions of positive clauses. We know that the action is at the next deeper level. There are 12 negated literals (all on the last line above) that permit insertion into a deeper level. They all have the form

```
(color-number)
```

The set insertion ^X^ will take the following form

```
((B1 ^X1^)(G1 ^X2^)(R1 ^X3^)) ...
```

Selecting the first one, (B1) for set insertion yields:

```
(B1  ^(B1 B2) (B1 B3) (B1 B4) (B2 B3) (B2 B4) (B3 B4)
      (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
      (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
      ((B2)(G2)(R2)) ((B3)(G3)(R3)) ((B4)(G4)(R4))^  )

(B1  ^(   B2) (   B3) (   B4) (B2 B3) (B2 B4) (B3 B4)
      (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
      (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
      ((B2)(G2)(R2)) ((B3)(G3)(R3)) ((B4)(G4)(R4))^  )

(B1  ^(   B2) (   B3) (   B4) (B2 B3) (B2 B4) (B3 B4)
      (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
      (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
      (    (G2)(R2)) (    (G3)(R3)) (    (G4)(R4))^  )

(B1  ^(   B2) (   B3) (   B4)
      (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
      (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
      (    (G2)(R2)) (    (G3)(R3)) (    (G4)(R4))^  )
```

At this point it looks as though the virtual subform will not reduce further. The objective is to get the literal form, (B1), to vanish. If all the negated variables in *3TET-newtoplevel* vanish, then we succeed, since

((B1)(G1)(R1))... ==> ( )... ==> ( )

Name the virtual insertion the **put-form** and the form into which the virtual form is put, the **into-form**.

The overall strategy is

1. Expose a new, possibly empty, set-of atoms ATS by inserting the put-form into the first level of the into-form. If no atoms, descend by inserting the appropriately reduced put-form into each bound in the current context. That is, descend the parens-tree breadth first with optional parallelism.

At initialization, the put form is all other forms in the context of the into-form. Each positive-clause requires exactly one put-form, since it has depth =1.

2. Erase the current ATS from the put-form. If there are changes, submit the new put-form to the algorithm.

```
If the resulting put-form is a ground,
      empty = return (nil) =  <void>
      mark  = return ( )   =  nil
else recur on inserting the current put-form
      into the next level of the local into-form.
```

3. When the put-form reaches bottom, it is in only one state, that of an indeterminate collection of variables and boundaries.

```
If the put-form were <void>,
      it would not reach bottom
If the put-form were mark,
      it would obliterate bottom before reaching it
```

If the current and last extraction does not produce a mark, then the put-form is abandoned. (Some advanced techniques use the scraps)

4. The entire assemblage of top-level bound Insertions (with virtual forms erased as not useful), and their recursive descent down each bound, is collected and compared to the initial into-form. If the new into-form is smaller, recur on it. Otherwise return into-form.

In the example fragment of this problem, i.e. (B1 ^X^), is the virtual insertion reduced? That is, is the put-form of

```
(B1  ^(   B2) (   B3) (   B4)
      (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
      (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
      (    (G2)(R2)) (    (G3)(R3)) (    (G4)(R4))^  )
```

in TAUT? If so, certainly the B variables are irrelevant, which seems reasonable given that the selected literal for insertion was (B1). We have again recurred, again with a simpler question. This time, however, the question: is the put-form ***3TET-level1***

```
^...  (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
      (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
      ((G2)(R2)) ((G3)(R3)) ((G4)(R4))^
```

in TAUT? This is determinable with P effort, since the form is 2SAT.

We can begin again, seeking to show that the above form is in TAUT, using the same strategy of set insertion. That is, we show the above reduction as a Lemma.

Selecting ((G2)(R2)) for set insertion yields:

```
((G2)(R2) ^(G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
           (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
                     ((G3)(R3)) ((G4)(R4))^  )
```

Do not subsume, insert into deeper level:

```
((G2  ^(G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
       (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
       ((G3)(R3)) ((G4)(R4))^ )
 (R2  ^(G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
       (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
       ((G3)(R3)) ((G4)(R4))^ )   )
```

Extract:

```
((G2  ^(G1   ) (G1 G3) (G1 G4) (   G3) (   G4) (G3 G4)
       (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
       ((G3)(R3)) ((G4)(R4))^ )
 (R2  ^(G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
       (R1   ) (R1 R3) (R1 R4) (   R3) (   R4) (R3 R4)
       ((G3)(R3)) ((G4)(R4))^ )   )
```

Extract and clarify using newly exposed literals:

```
      ((G2  ^(G1   ) (G1 G3) (G1 G4) (   G3) (   G4) (G3 G4)
            (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
                  R3         R4  ^ )
       (R2  ^(G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
            (R1   ) (R1 R3) (R1 R4) (   R3) (   R4) (R3 R4)
              G3         G4      ^ )   )
```

Recur:

```
      ((G2  ^(G1   ) (G1 G3) (G1 G4) (   G3) (   G4) (G3 G4)
            (R1 R2) (R1   ) (R1   ) (R2   ) (R2   ) (     )
                  R3         R4  ^ )
       (R2  ^(G1 G2) (G1   ) (G1   ) (G2   ) (G2   ) (    )
            (R1   ) (R1 R3) (R1 R4) (   R3) (   R4) (R3 R4)
              G3         G4      ^ )   )
```

And finally Occlude each of the interior literals:

```
      ((G2  ^(     )^ ) (R2  ^(     )^ ))

      ( )
```

Now unwinding this result, we know that

```
      ((G2)(R2)) ==> ( )
```

Substituting into the Lemma, *3TET-level1*:

```
      ^...  (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
            (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
            ((G2)(R2)) ((G3)(R3)) ((G4)(R4))^

      ^...  (G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
            (R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
            (        ) ((G3)(R3)) ((G4)(R4))^

      ^( )^
```

We have shown that the put-form ^X1^ in (B1 ^X1^) is ( ), thus

```
      (B1 ^X1^) ==> (B1 ( )) ==> <void>
```

(B1 ^X1^) comes from the negated literal

((B1)(G1)(R1))

in *3TET-newtoplevel*, which is:

```
(B1 B2) (B1 B3) (B1 B4) (B2 B3) (B2 B4) (B3 B4)
(G1 G2) (G1 G3) (G1 G4) (G2 G3) (G2 G4) (G3 G4)
(R1 R2) (R1 R3) (R1 R4) (R2 R3) (R2 R4) (R3 R4)
((B1)(G1)(R1)) ((B2)(G2)(R2)) ((B3)(G3)(R3)) ((B4)(G4)(R4))
```

We have shown

```
              ((B1 ^X1^)(G1 ^X2^)(R1 ^X3^)) ...
         ==>  (         (G1 ^X2^)(R1 ^X3^)) ...
```

The objective is to have any clause in *3TET-newtoplevel* reduce to ( ), and we have observed that only the four clauses consisting of negated atoms might do so.

There are three possible paths to complete the proof.

1) Continue to eliminate (G1 ^X2^) and (R1 ^X3^) by a process identical to that applied to (B1). This would reduce ((B1)(G1)(R1)) to ( ).

2) Continue to eliminate (B2), (B3) and (B4) by a process identical to that applied to (B1). This would reduce the problem to 2SAT.

3) Argue from symmetry that either 1) or 2) has already been achieved.

The Central Idea

CNF forms are maximally tangled general forms. Using boundary techniques, forms in TAUT are either trivial or supported by extreme symmetry. This of course is not generally true, since these two types can be mixed together. However, even in the mixing case, the symmetries can be identified at some level of form abstraction.

A non-polarized CNF form is a hypercube rotation of the polarized case, therefore we can assume a polarized CNF form for which all clauses are completely positive or negative. That is,

```
(A (B)) (B (A)) ==>  ( (A B) ((A)(B)) )
```

For a general problem, divide the N clauses into X positive and Y negative:

```
N = X + Y
```

Positive case

Each insertion of (N-1) context clauses into a positive clause asks a question.

```
(a b ... ^(a c)...^)
```

The context of the positive-clause is disjunctive. For the form to be in TAUT, one positive-question must be ( ). But there is no way for an inserted form to convert a positive-clause to ( ). Should the virtual form vanish, the positive variable will remain. Should the virtual form become ( ), it will erase the clause, not make it ( ). Finally, should the virtual form not reduce, then this clause does not simplify.

We carry out the insertions on the positive-clauses solely to see if any are redundant. If any vanish, we have a simpler TAUT form to insert into the negative-clauses.

Negative case

Each insertion of at most (N-1) context clauses into a negative clause asks a subquestion.

```
((a ^(a c)...^)
 (b ^(a c)...^)
 ...          )
```

For the negative clause to become ( ), *every* negated literal must vanish. That is,

```
((a ^...( )...^)
 (b ^...( )...^)
 ...           )
```

Should any one of these not vanish, the form will not collapse.

Symmetry

Symmetry arguments can be used to reduce N reduction tests to 1 test.

Recursion

Should a virtual form fail to reduce to ( ), it can be separated and treated as an independent Lemma, with the same techniques applying to it recursively.