

Upper and Lower Bounds on the Number of Solutions

Jean-Philippe Martin
Microsoft Research Cambridge

December 2007

Technical report MSR-TR-2007-164

Abstract

We present a fast and extensible algorithm for computing upper and lower bounds on the number of solutions to a system of equations. For a given size of variables (e.g. 32 bits), the algorithm can be run in time linear in the number of terms and variables, at the cost of looser bounds.

1 Introduction

It is well-known how to compute a solution for a system of equations. In this paper we, instead, try to quickly determine lower and upper bounds on the number of solutions. The problem of counting solutions also has a long history ([3] for example says that Binet and Cauchy looked in 1812 into the problem of counting the number of perfect matchings in a bipartite graph). There is a wealth of papers and tools for the case of boolean variables (known as #SAT), but we are not aware of any tool for more general constraints (known as #CSP) that focuses on speed.

Our goal is a fast algorithm (one that finishes in the order of seconds) that can handle complex expressions such as standard arithmetic or bit shifts (Figure 1). The answer must be correct, but we are willing to accept looser bounds in exchange for faster execution. The algorithm must be easily extendable to new operations as the need arises. Our new protocol satisfies these requirements.

```
(= (bitwise-and INPUT[0] 32) 0)
(< (add (bitwise-and INPUT[1] 1) 3) 4)
```

Figure 1: Sample equations

2 Fast Solution Count Bounder

The Fast Solution Count Bounder (FSCB) algorithm takes as input a series of equations in a rich subset of the Yices syntax (see figures 3–8 for a list) and outputs:

- A lower bound on the number of solutions
- An upper bound on the number of solutions
- For each variable, a lower bound on the number of acceptable values for that variable
- For each variable, the corresponding upper bound.

A value v is acceptable for a variable x if there is at least one solution to the system of equations where $x = v$. For the example of Figure 1, a perfect solution count bounder would return $(16384, 16384, [128, 128], [128, 128])$. Our FSCB algorithm returns that exact value.

These bounds compose for the example given. The first expression has 128 solutions, the second has 128 as well and the two together have exactly 16384 (128^2) solutions. FSCB leverages this composability for speed.

The computation proceeds in two steps. FSCB first computes the bounds for each individual equation, and then combines them for the final answer.

2.1 Per-Equation Bounds

The first step is to compute per-equation bounds. For best results, FSCB first combines all the equations that refer to the same single variable. For example, $(/= \text{INPUT}[0] \ 2)$ and $(< \text{INPUT}[0] \ 128)$ would be combined into the single equation $(\text{and } (/= \text{INPUT}[0] \ 2) (< \text{INPUT}[0] \ 128))$. This does not change the semantics of the input. For each of those combined equations, FSCB then evaluates the expression for each value of the input variable to determine the exact number of solutions. For a given size of variables, the running time of this step is $O(n + m)$ where n is the number of equations and m is the number of variables. This step is only practical for relatively

	variable x	constant c
min,max	0, 255	c, c
min-,max-range	256, 256	1, 1
homogeneous	yes	yes
masked-homogeneous	yes	yes

Figure 2: Leaf summary rules

small variables (8 to 16 bits in our experience): equations that refer to larger variables can be left for the next step.

FSCB then considers each multi-variable equation in turn. It bounds the number of solutions by progressively constructing an approximation that we call a *summary*. The summary of an expression contains the set of input variables it refers to, a lower and upper bound for the value of the expression, a lower and upper bound for the number of images (we call this the *range*), and two boolean flags. An expression is flagged as *homogeneous* only if every image has the same number of preimages (recall that if $f(x) = y$, then x is a preimage of y). An expression is marked *masked-homogeneous* only if it is homogeneous and there exist v, w such that the set of images is exactly $\{x \& v | w : \text{for all } x\}$. Symbol “&” denotes bitwise-and, and “|” is bitwise-or. For example, `(bitwise-and x 32)` (for 8-bit x) has minimum 0, maximum 31, has exactly 32 distinct images (values 0 through 31) and is both homogeneous (every image has 8 preimages) and masked-homogeneous ($v = 32, w = 0$).

FSCB first replaces each leaf of the expression (i.e. a constant or variable) with the corresponding summary. The leaf summary rules are shown in Figure 2. Next, FSCB summarizes expressions of summaries. In this way, FSCB goes up the expression tree and progressively summarizes all the terms. For each supported operation, FSCB indicates how to compute the expression summary from that of the sub-expressions. To extend FSCB to support additional operations one simply has to add new rules there. For example, consider the second equation of Figure 1. FSCB first summarizes the `(bitwise-and INPUT[1] 1)` subexpression. The resulting summary has minimum value 0, maximum 1, range 2, and is both homogeneous and masked-homogeneous. FSCB would then summarize the constant 3, and next combine the two summaries using the addition rule to obtain the summary for `(add (bitwise-and INPUT[1] 1) 3)`. In this case, the minimum is 3, maximum 4, the range still 2, the expression is homogeneous but not masked-homogeneous.

Pseudocode for these summary rules is shown in Figures 3–6 for the case of 8-bit values. They can be easily generalized to wider values. The pseudocode uses the notation $[l, h, lr, hr, hom, mh]$ to denote a summary with minimum l , maximum h , min-range lr , max-range hr , flagged as homogeneous if hom is true and flagged as masked-homogeneous if mh is true. For simplicity, the code for commutative functions assumes that if one of the arguments is constant, then it is the second one. For example, if c is a constant then $(\text{add } c \ f)$ is summarized using the rule for $(\text{add } f \ c)$. The pseudocode uses a number of helper function, described in Figure 7. These rules can for example correctly compute that the expression $((x \text{ xor } y) + 5) \& 13 + 7$ can take 8 possible values, between 7 and 20. Some of the rules have tests that may appear surprising at first. As an example, we explain the rules for multiplication of f and g (Figure 4). The first check (line 2) determines whether the multiplication can overflow. If it does not, then the lower bound for $f * g$ is simply the product of the lower bound for f and that for g (line 3), similarly for the upper bound (line 4). We then compute the number of images. There are two cases where the result might just be a constant (i.e. with range 1): (i) the functions have variables in common (consider e.g. $(x \& 1) * (\neg x \& 1)$), or (ii) one of the functions could be the constant zero. If either is true, then we set lr to 1 (line 8). Otherwise we know that $f * g$ has at least as many images as both f and g . Since there is no overflow, we know that multiplying a homogeneous function with a constant will be homogeneous (line 11): the set of images will change, but they still each have the same number of preimages. The masked-homogeneous property (line 12) is maintained if we multiply f by a power of two, since this is equivalent to a bitwise shift. The new v, w are simply multiplied by this same factor: $(x \& v | w) * 2^c = (x * 2^c) \& (v * 2^c) | (w * 2^c)$. Finally, in the case of overflows we set l, h, lr, hr, mh conservatively, but we know the result is homogeneous if f is homogeneous and g is an odd constant. The reason is that all inputs that yield distinct images through f also yield distinct images through $f * g$. Here is a short proof. Let $max = 2^{bitwidth}$. Suppose that g is an odd constant c and there are two distinct values $x = f(x_0)$ and $y = f(y_0)$. Without loss of generality, $x < y$ and $y = x + d$ ($d < max$). We want to show that $x * c$ and $y * c$ are distinct (modulo max). Since $y * c \bmod max = (x * c + d * c) \bmod max$, that means the two values can only be equal if $(d * c) \bmod max = 0$. Consider the factorization of the left hand side: factor 2 appears at most $bitwidth - 1$ times in d and none in c . Therefore, $c * d$ is not a multiple of max .

A final set of rules is used to determine the number of accepted inputs when two summaries (or a summary and a constant) are compared. For

example, in $(f = 3)$, the number of accepted inputs is between $f.ld$ and $f.hd$ if $f.l \leq 3 \leq f.h$ (ld and hd are defined in Figure 7), and zero otherwise. The rules for the minimal number of accepted inputs are shown in Figure 8. To get the lower bound on the number of answers, we conservatively assume that as many as possible of the values of the summary do not satisfy the equation. For example, when considering $(f > 12)$, we consider the smallest f that matches the summary. If the summary for example indicates 8 values between 7 and 20, then the rule will consider that these values are 7–14 (2 accepted inputs). When instead computing the upper bound in this example, FSCB considers that the values are 13–20 (8 accepted inputs). None of the rules include any loop or recursion, so bounding an equation takes $O(t)$, where t is the number of terms in the equation.

2.2 Combining Bounds

The second step is to combine per-equation bounds. Given an equation a , the procedure from the previous section determines the lower and upper bounds on the number of accepted inputs, denoted $a.la$ and $a.ha$. We use $a.vars$ to denote the set of variables in a . Given two such bounds, we can compute the bounds on a system that has the two corresponding equations. For example if equation $a(x) = c$ admits 128 solutions and equation $b(y) = d$ admits the same number, we can deduce that $(a(x) = c \wedge b(y) = d)$ admits 128^2 pairs of inputs. If the two equations a and b refer to the same variables, then the upper bound for the number of accepted inputs is the minimum between $a.ha$ and $b.ha$. Figure 9 shows how we merge per-equation lower bounds into bounds that apply to both equations together. We merge these bounds pairwise until we have a single (lower and upper) bound for the entire system of equations.

Merging all the per-equation bounds has time complexity $O(n)$ done that way. However, we can get a higher-quality result by first merging the equations that refer to similar variables. Consider a graph where each node represents an equation, and there is an edge between equations that have a variable in common. We compute the connected components of this graph and merge those equations first, before merging the bounds for the connected components. This approach results in better bounds at the cost of a longer runtime.

To compute the bounds on the number of acceptable inputs for a single variable x , one can simply use the same procedure to merge all the equations that refer to at least that variable. The upper bound on the total number of accepted inputs is also an upper bound on the number of values of x that are

accepted. The lower bound is computed according to Figure 10. Computing the bounds for a variable has a running time of $O(n)$.

3 Related Work

Counting the number of solutions for boolean conditions is known as #SAT. It has been studied for some time and has applications in artificial intelligence [5] among others. While it is always possible to transform finite-field arithmetic constraints (such as the ones we consider) into a series of boolean conditions, analyzing these constraints directly may allow faster computations. There is some research on finding the number of solutions to algebraic constraints. Pesant, for example [4], shows specific techniques for a number of special cases. He suggests for example that the number of solutions for $f < g$ can be computed from tables that contain all the images of f and g in increasing order.

The related work we have found focuses on finding good bounds, rather than finding bounds quickly. We have unfortunately not found any solution-counting program that would accept inputs in the format we used so comparing performance is difficult. To get a ballpark idea, we note that the MBound algorithm [2] (the fastest #SAT algorithm we are aware of) can analyze 20'000 clauses with 600 boolean variables in under 3 minutes (Their later SampleCount protocol [1] gives better bounds but takes longer). Our algorithm analyzed a bigger problem in less time: 6'500 32-bit clauses (equivalent to 208'000 boolean clauses) with 4'800 8-bit variables, finishing under 20 seconds. We expect that optimizing our algorithm could further reduce the execution time. The speed difference does not mean that our algorithm is superior, however, because our different focus means that our bounds are almost certainly inferior to those that MBound would have found. We are also reporting numbers from two different problems, so they are not directly comparable. Even though we cannot run MBound on the same input directly, we could in principle translate our clauses into boolean ones and then compare the two algorithms directly. Unfortunately we are not aware of any such format converter.

4 Conclusion

The FSCB algorithm can quickly compute the lower and upper bound on the number of solutions to a given system of equations. If variables has constant size and the number of terms per equations is constant, the FSCB can run

in $O(n + m)$ where n is the number of equations and m is the number of variables. This high speed is obtained through an algorithm that requires only two passes through the equations: one to merge the equations that refer to the same variables and one to compute per-equation bounds and combine them. This speed is obtained at the cost of relatively loose bounds. The bounds can be improved by combining the per-equations bounds in a better order. We chose this slower approach for our implementation and could bound a system of over 6'500 32-bit equations under 20 seconds.

We expect FSCB to be primarily of interest when computation speed is crucial, and possibly as a heuristic to guide another algorithm. Another benefit is that it can process clauses that use a combination of algebra (e.g. addition, multiplication) and bit manipulation (e.g. bitwise-or or shift-left), and that it can easily be extended to process even more. FSCB could be further improved by storing more information in the summaries, refining the summary rules, or finding better ways of combining the per-condition bounds.

```

(add f g):
  if (f.h + g.h > 255) then
    l := 0
    h := 255
    lr := max(f.lr/2, g.lr/2);
  else
    l := f.l + g.l
    h := f.h + g.h
    lr := max(f.lr, g.lr);
  if (f and g have variables in common) then
    lr := 1
  hr := min(f.hr * g.hr, 256);
  mh := (is-permutation(f) && is-constant(g))
        || (is-constant(f) && is-permutation(g))
  if (mh) then lr := hr
  hom := (f.hom && is-constant(g))
        || (is-constant(f) && g.hom)
        || ((f,g have no variable in common)
            && is-permutation(f) && is-permutation(g))
        || mh
  return [l,h,lr,hr,hom,mh]

(subtract f g):
  if (f.l - g.h < 0) then
    l := 0
    h := 255
    lr := max(f.lr/2, g.lr/2);
  else
    l := f.l - g.h
    h := f.h - g.l
    lr := max(f.lr, g.lr);
  if (f and g have variables in common) then
    lr := 1
  hr := min(f.hr * g.hr, 256);
  mh := (is-permutation(f) && is-constant(g))
        || (is-constant(f) && is-permutation(g))
  if (mh) then lr := hr
  hom := (f.hom && is-constant(g))
        || (is-constant(f) && g.hom)
        || ((f,g have no variable in common)
            && is-permutation(f) && is-permutation(g))
        || mh
  return [l,h,lr,hr,hom,mh]

```

Figure 3: Summary rules

```

(multiply f g):
  if (f.h * g.h < 256) then
    l := f.l * g.l
    h := f.h * g.h
    if ((f and g have a variable in common)
        or (f.lr==1 && f.l==0)
        or (g.lr==1 && g.l==0))
    then lr := 1
    else lr := max(f.lr, g.lr)
    hr := min(f.hr * g.hr, 256)
    hom := (is-constant(g) && f.hom)
    mh := (f.mh && is-constant(g) && g.l is a power of two)
    return [l,h,lr,hr,hom,mh]
hom := (f.hom && is-constant(g) && g is odd)
return [0,255,1,256,hom,false]

(bitwise-and f g):
  h := min(f.h,g.h)
  l := 0
  max-newrange := 2 ^ max-number-of-bits-set(g)
  min-newrange := 2 ^ min-number-of-bits-set(g)
  max-d = 256/min-newrange
  if (is-constant(g)) then
    hr := min(f.hr, max-newrange)
  else
    hr := min(f.hr * g.hr, 256)
  if (f and g have no variable in common) then
    lr := f.lr / max-d
  else
    lr := 1
  mh := (f.mh && is-constant(g))
    || (g.mh && is-constant(f))
  return [l,h,lr,hr,mh,mh]

```

Figure 4: Summary rules (2)

```

(bitwise-or f g):
  l := max(f.l, g.l)
  h := max-or(f.h,g.h)
  ldiv := 2 ^ min-number-of-bits-set(g)
  hdiv := 2 ^ max-number-of-bits-set(g)
  hnr := 256 / ldiv
  lnr := 256 / hdiv
  if (is-constant(g) then
    hr := min(f.hr, hnr)
  else
    hr = min(f.hr * g.hr, 256)
  if (f and g have a variable in common) then
    lr := 1
  else
    lr := f.lr / hdiv
  mh := (f.mh && is-constant(g))
    || (g.mh && is-constant(f))
  return [l, h, lr, hr, mh, mh]

(bitwise-xor f g):
  if (is-constant(g)):
    l := min-xor(f,g.l)
    h := max-or(f.h,g.h)
    return [l,h,f.lr,f.hr,f.hom,f.mh]
  if (f and g have no variable in common) then
    lr := max(f.lr, g.lr)
  else
    lr := 1
  hr = min(f.hr*g.hr, 256);
  hom := (f and g have no variable in common) &&
    ( (g.mh && (is-permutation(f) || is-constant(f)))
      || (is-permutation(g) && (f.mh || is-constant(f))))
  return [0,255,lr,hr,hom,false]

```

Figure 5: Summary rules (3)

```

(shift-left f g):
  if (!is-constant(g)): return [0,255,1,256,false,false]
  if g.l>=8:
    return [0,0,1,1,true,true]
  h := f.h << g.l
  if (f.h * 2 ^ g.l > 255):
    l := 0
    hom := false
  else
    l := f.l << g.l
    hom := f.hom
  mh := f.mh && hom
  if (g.l<8) then
    d := 2 ^ g.l & 255
    nr := 256 / d
    lr := f.lr / d
  else
    nr := 1
    lr := 1
  hr := min( f.hr, nr )
  return [l,h,lr,hr,hom,mh]

(shift-right f g):
  if (!is-constant(g)): return [0,255,1,256,false,false]
  if (2 ^ g.l > f.h):
    return [0,0,1,1,true,true]
  h := h >> g.l
  l := l >> g.l
  hom := mh := f.mh && (2 ^ g.l <= f.l)
  if (g.l<8) then
    d := 2 ^ g.l & 255
    nr := 256 / d
    lr := f.lr / d
  else
    nr := 1
    lr := 1
  hr := min( f.hr, nr )
  return [l,h,lr,hr,hom,mh]

```

Figure 6: Summary rules (4)

```

is-constant(f):
    return (f.l == f.h)

is-permutation(f):
    return (f.lr==256 && |f.vars|==1)

max-number-of-bits-set(f):
    if is-constant(f) then return number-of-bits-set(f.l)
    pick smallest x s.t.  $2^x > f.h$ 
    return x

min-number-of-bits-set(f):
    if is-constant(f) then return number-of-bits-set(f.l)
    if (f.l>0) return 1
    return 0

input-bits(f,g):
    return 8 * | f.vars union g.vars |

input-count(f,g):
    return  $2^{\text{input-bits}(f,g)}$ 

max-or(x,y):
    h := max(x,y)
    l := min(x,y)
    pick smallest z s.t.  $2^z > l$ 
    return h | ( $2^z-1$ )

min-xor(f,c):
    if (c < f.l) then
        pick smallest ac s.t.  $2^{ac} > c$ 
        return f.l &  $\sim(2^{ac}-1)$ 
    if (f.h < c) then
        pick smallest ah s.t.  $2^{ah} > f.h$ 
        return c &  $\sim(2^{ah}-1)$ 
    return 0

f.ld:
    if (f.hom) then return input-count(f) / f.hr
    else return 1

f.hd:
    if (f.hom) then return input-count(f) / f.lr
    else return input-count(f) - f.lr + 1

```

Figure 7: Helper functions for the summary rules

```

(equals f g):
  if (f.h < g.l) || (g.h < f.l) then expression is unsatisfiable
  if (both constant and equal) then return 0
  if (f and g have a variable in common) then
    return input-bits(f,g)
  leq := max(f.l, g.l)
  heq := min(f.h, g.h)
  minlhit := f.lr - max(f.h - heq, leq - f.l)
  minlhit := min(minlhit, heq - leq + 1)
  minrhit := g.lr - max(g.h - heq, leq - g.l)
  minrhit := min(minrhit, heq - leq + 1)
  inter := max(1, minlhit + minrhit - (heq - leq + 1))
  ic := input-count(f,g)
  return max(1, min(inter * f.ld * g.ld, ic))

(not-equal f g):
  if (f.h < g.l) || (g.h < f.l) then return 0
  if (both constant and equal) then expression is unsatisfiable
  if (f and g have a variable in common) then
    return input-bits(f,g)
  leq := max(f.l, g.l)
  heq := min(f.h, g.h)
  accepted := input-count(f,g) - (heq - leq + 1) * f.hd * g.hd
  return max(1, min(accepted, input-count(f,g)))

(unsigned-less-than f g):
  if (f.h < g.l) then return 0
  if (f.l >= g.h) then expression is unsatisfiable
  wcRange := g.l - (f.h+1 - f.lr)
  if (wcRange >= 1) then
    accepted := min(wcRange*f.ld, input-count(f))
  else if (f,g have input byte in common) then
    accepted := 1
  else
    accepted := f.ld
  return accepted * 2^(8*variables in g but not f)

(unsigned-greater-than f g):
  if (f.l > g.h) then return 0
  if (f.h <= g.l) then expression is unsatisfiable
  wcRange := f.l + f.lr - 1 - g.h
  if (wcRange >= 1) then
    accepted := min(wcRange*f.ld, input-count(f))
  else if (f,g have input byte in common) then
    accepted := 1
  else
    accepted := f.ld
  return accepted * 2^(8*variables in g but not f)

```

Figure 8: Lower bounds from summaries

```

merge(a,b):
  if (a.vars disjoint from b.vars) then
    return [a.la*b.la, a.ha*b.ha]
  else
    ab := a.vars union b.vars
    i := a.la * 2^(8*size(ab-a.vars))
    j := b.la * 2^(8*size(ab-b.vars))
    la := i + j - 2^(8*size(ab))
    ha := min(i,j)
    return [la,ha]

```

Figure 9: Merging bounds

```

per-variable(bounds a):
  ha := min(256,a.ha)
  if (a.la==0) then la:=0
  else la := max(1, ceil(a.la / 2^(8*(size(a.vars)-1))))
  return [la,ha]

```

Figure 10: Computing per-variable bounds

References

- [1] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *IJCAI*, pages 2293–2299, 2007.
- [2] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *AAAI*, 2006.
- [3] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM*, 51(4):671–697, 2004.
- [4] G. Pesant. Counting solutions of CSPs: A structural approach. In *IJCAI*, pages 260–265, 2005.
- [5] D. Roth. On the hardness of approximate reasoning. *Artif. Intell.*, 82(1-2):273–302, 1996.