November 24, 2020 at 13:23

1. Intro. This program computes the BDD size of the hidden weighted bit function, given a permutation of the input variables. After I wrote the program HWB a few days ago, and ran it for an hour in the case $n = 100$ with 8 gigabytes of memory, I realized that the whole calculation can really be done *much* faster—indeed, in polynomial time.

So now I'm doing it a better way. The new way is so efficient, in fact, that I'm going to have fun and implement it by simulating decimal arithmetic, using one byte per digit, throwing all ordinary notions of efficiency out the window.

The previous method generated "slot tables." Now I've renamed them "slate tables," and discussed the relevant theory in Section 7.1.4 of TAOCP. With this theory I don't need to "work top down" and effectively generate each node of the BDD. Instead, I determine the number of beads of height m by a direct calculation.

```
#define n 100   /* the number of variables */
#define memsize 1000000   /* the number of bytes for arithmetic; must exceed 3n */
#include <stdio.h>
#include <stdlib.h>
  char mem[memsize];   /* the big storage area */
  int memptr;   /* the number of bytes in use */
  int numptr;   /* the number of numbers in use */
  int start[n * n];   /* where remembered numbers begin in mem */
  int bico[n][n];   /* table of binomial coefficients that I've computed */
  int addedA[n], addedB[n], addedC[n], addedD[n];   /* constant memos */
  unsigned char rev[256];   /* bit-reversal table: 0^R, 1^R, ..., 255^R */
  int perm[n + 1];   /* the permutation */
  int nonbeads;   /* nonbeads found at the current height */
  int tnonbeads;   /* total nonbeads so far */
  ⟨Subroutines 3⟩
  main(int argc)
  {
    register int i, j, k, m, p, s, ss, t, tt, w, ww;
    ⟨Set up the permutation, perm 2⟩;
    ⟨Initialize mem 8⟩;
    for (k = 0; k < n; k++) {
      ⟨Compute b_k 10⟩;
      ⟨Print b_k and add it to the grand total 11⟩;
    }
    printf("height␣0:␣2\n");   /* k = n is a simple special case */
    ⟨Print the grand total 4⟩;
  }
```

2. The purpose of this step is to set $perm[j] = j\pi$ for $1 \le j \le n$, where π is the desired permutation of the input variables. And I set $perm[0] = n + 1$, because $perm[0] = 0$ would make x_0 appear to be a member of $\{x_1, \ldots, x_k\}$.

In this implementation I use (almost) the "hybrid" ordering of Bollig, Löbbing, Sauerhoff, and Wegener. That means the first $n/5$ elements come alternately from the top $n/10$ and the bottom $n/10$. The remaining $4n/5$ elements are ordered according to the bit reversal of the difference between them and $9n/10$.

⟨ Set up the permutation, *perm* 2 ⟩ ≡

```
for (j = #80, m = 1; j; j >>= 1, m <<= 1)
   for (k = 0; k < #100; k += j + j) rev[k + j] = rev[k] + m;
for (j = m = 1, k = n; j ≤ n/10; j++, k--, m += 2) perm[k] = m, perm[j] = m + 1;
for (i = 0; m ≤ n; i++, m++) {
   while (rev[i] > k − j) i++;
   perm[k − rev[i]] = m;
}
printf("Starting␣from␣perm");
for (j = 1; j ≤ n; j++) printf("␣%d", perm[j]);
printf("\n");
perm[0] = n + 1;
```

This code is used in section 1.

3. Decimal addition. The kth number in my decimal memory starts at location *start*[k] in *mem*, and ends just before location *start*[$k+1$]. Each byte of *mem* contains a single digit, and the least significant digits come first.

Number 0 is the grand total, and number 1 is the total-so-far at height m. The other numbers are binomial coefficients, which I compute from scratch as needed.

To warm up, here's a routine to print out the kth number:

⟨ Subroutines 3 ⟩ ≡

```
void printnum(int k)
{
  register int j;
  for (j = start[k + 1] - 1; j > start[k]; j--)
    if (mem[j]) break;
  for ( ; j ≥ start[k]; j--) printf("%d", mem[j]);
}
```

See also sections 5, 6, 7, and 9.

This code is used in section 1.

4. ⟨ Print the grand total 4 ⟩ ≡

```
printf("Altogether␣");
printnum(0);
printf("-%d␣nodes;␣I␣used␣%d␣bytes␣of␣memory␣for␣%d␣numbers.\n", tnonbeads, memptr, numptr);
```

This code is used in section 1.

5. ⟨ Subroutines 3 ⟩ +≡

```
void clearnum(int k)
{
  register int j;
  for (j = start[k]; j < start[k + 1]; j++) mem[j] = 0;
}
```

6. The *add* routine adds number k to number l and stores the result as a brand new number, whose index is returned.

We assume (conservatively) that all numbers have at most n digits.

⟨ Subroutines 3 ⟩ +≡
```
int add(int k, int l)
{
  register int c, i, j;
  if (memptr + n ≥ memsize) {
    fprintf(stderr, "Out␣of␣memory␣(memsize=%d)!\n", memsize);
    exit(−1);
  }
  for (c = 0, i = start[k], j = start[l]; ; i++, j++, memptr++) {
    if (i < start[k + 1]) {
      if (j < start[l + 1]) mem[memptr] = mem[i] + mem[j] + c;
      else mem[memptr] = mem[i] + c;
    } else {
      if (j < start[l + 1]) mem[memptr] = mem[j] + c;
      else break;
    }
    if (mem[memptr] ≥ 10) c = 1, mem[memptr] −= 10;
    else c = 0;
  }
  if (c) mem[memptr++] = 1;
  numptr++;
  start[numptr + 1] = memptr;
  return numptr;
}
```

7. Another variant of addition adds number l to number k, and replaces number k by the sum. This routine is used only when $start[k+1] - start[k]$ is large enough to contain the sum.

⟨ Subroutines 3 ⟩ +≡
```
void addto(int k, int l)
{
  register c, i, j;
  for (c = 0, i = start[k], j = start[l]; i < start[k + 1]; i++, j++) {
    mem[i] += (j < start[l + 1] ? mem[j] : 0) + c;
    if (mem[i] ≥ 10) c = 1, mem[i] −= 10;
    else c = 0;
  }   /* here I could check to make sure that c = 0, but I won't bother */
}
```

8. Number 2 in *mem* is actually the constant '0', and number 3 is '1'.

#define *grandtotal* 0
#define *subtotal* 1
#define *zero* 2
#define *one* 3

⟨ Initialize *mem* 8 ⟩ ≡

```
start[grandtotal] = 0;
mem[0] = 2;      /* the grand total is initially 2 */
start[subtotal] = start[grandtotal] + n;
start[zero] = start[subtotal] + n, start[zero + 1] = start[zero] + 1;
mem[start[one]] = 1, start[one + 1] = start[one] + 1;
numptr = one, memptr = start[numptr + 1];
```

This code is used in section 1.

9. Here's how I compute binomial coefficients $\binom{m}{k}$, without attempting to optimize.

⟨ Subroutines 3 ⟩ +≡

```
int binom(int m, int k)
{
  if (k < 0 ∨ k > m) return zero;
  if (k ≡ 0 ∨ k ≡ m) return one;
  if (¬bico[m][k]) bico[m][k] = add(binom(m − 1, k), binom(m − 1, k − 1));
  return bico[m][k];
}
```

10. The algorithm. So much for infrastructure; let's get to work.

⟨ Compute b_k 10 ⟩ ≡
```
clearnum(subtotal);
nonbeads = 0;
m = n − k;
⟨ Clear the four constant tables 14 ⟩;
for (s = 0; s ≤ k; s++) {
   ⟨ Add contributions for slates (s, k) to subtotal 12 ⟩;
}
⟨ Correct for constant nonbeads 16 ⟩;
```
This code is used in section 1.

11. ⟨ Print b_k and add it to the grand total 11 ⟩ ≡
```
printf("height␣%d:␣", m);
printnum(subtotal);
if (nonbeads) printf("-%d\n", nonbeads);
else printf("\n");
addto(grandtotal, subtotal);
tnonbeads += nonbeads;
```
This code is used in section 1.

12. Each slate for (s,k) is $[r_0,\ldots,r_m]$, where r_j is 0 or 1 when $perm[s+j] \le k$, otherwise r_j is x_l where $perm[s+j] = l$. (The latter case represents one of the m remaining variables.) I compute the quantity w, which is the number of times the former case occurs; this is what Bollig et al. have called the "window size."

However, we set $r_0 \leftarrow 0$ and $r_m \leftarrow 1$ if they aren't already constant; r_0 and/or r_m are then called "false constants." With these conventions, there's exactly one slate table for each subfunction at height m.

Let $t = k - s$. The w settings of the constant r_j's run through all combinations of ss 1s and tt 0s such that $ss + tt = w$, $ss \le s$, and $tt \le t$.

If at least one of the positions $\{r_1,\ldots,r_{m-1}\}$ is nonconstant, a particular slate can occur only for one value of s. Otherwise the situation is more subtle, and I need to consider constant slates of four types depending on the boundary conditions.

- Type A, $r_0 = 0$ and $r_m = 0$: Here r_0 might be a false constant.
- Type B, $r_0 = 0$ and $r_m = 1$: Here r_0 and/or r_m might be false.
- Type C, $r_0 = 1$ and $r_m = 0$: Both r_0 and r_m are true.
- Type D, $r_0 = 1$ and $r_m = 1$: Maybe r_m is false.

A setting of ss 1s and tt 0s contributes to all four types if r_0 and r_m are true. It contributes only to type B if r_0 and r_m are false. It contributes only to types A and B if r_0 is false but r_m is true; only to B and D if r_0 is true but r_m is false.

⟨ Add contributions for slates (s,k) to *subtotal* 12 ⟩ ≡

```
for (w = ww = j = 0; j ≤ m; j++)
  if (perm[s + j] ≤ k) {
    w++;
    if (j > 0 ∧ j < m) ww++;
  }
if (ww ≡ m − 1) ⟨ Add contributions for a constant case 15 ⟩
else {
  ⟨ Correct for nonconstant nonbeads 13 ⟩;
  for (t = k − s, ss = s, tt = w − ss; tt ≤ t; ss−−, tt++) {
    addto(subtotal, binom(w, ss));
    if (ss ≡ p) nonbeads++;      /* see below */
  }
}
```

This code is used in section 10.

13. Nonbeads $[r_0,\ldots,r_m]$ are of four kinds: (a) $r_p = x_{k+1}$, $r_j = 1$ for $j < p$, and $r_j = 0$ for $j > p$; (b) $[0, x_n, 1]$; (c) $[r_0,\ldots,r_m] = [0,\ldots,0]$, within Type A; (d) $[r_0,\ldots,r_m] = [1,\ldots,1]$, within Type D. Here we look for (a) and (b).

⟨ Correct for nonconstant nonbeads 13 ⟩ ≡

```
p = n + 1;      /* n + 1 is "infinity" */
if (ww ≡ m − 2) {
  if (m ≡ 2 ∧ perm[s + 1] ≡ n) p = (perm[s + 2] ≤ k ? 1 : 0);
  else if (w ≡ m) {
    for (p = 1; ; p++)
      if (perm[s + p] > k) break;
    if (perm[s + p] ≠ k + 1) p = n + 1;
  }
}
```

This code is used in section 12.

14. Each constant type is a symmetric function. I need to contribute $\binom{m-1}{r}$ to the subtotal for each possible value of $r = r_1 + \cdots + r_{m-1}$. But I want to contribute exactly once for every such r; equal values of r can arise from different values of s. So there are tables *addedA*, *addedB*, *addedC*, *addedD*, to remember when a particular r has been contributed already to the counts of each type.

⟨ Clear the four constant tables 14 ⟩ ≡

```
for (j = 0; j < m; j++) addedA[j] = addedB[j] = addedC[j] = addedD[j] = 0;
```

This code is used in section 10.

15. Here's where I hope logic hasn't failed me.

⟨ Add contributions for a constant case 15 ⟩ ≡

```
{
  for (t = k − s, ss = s, tt = w − ss; tt ≤ t; ss−−, tt++)
    if (ss ≥ 0 ∧ tt ≥ 0) {
      if (perm[s + m] ≤ k) {       /* true constant at right */
        if (¬addedA[ss]) addedA[ss] = 1, addto(subtotal, binom(m − 1, ss));
        if (ss > 0 ∧ ¬addedB[ss − 1]) addedB[ss − 1] = 1, addto(subtotal, binom(m − 1, ss − 1));
      } else if (¬addedB[ss]) addedB[ss] = 1, addto(subtotal, binom(m − 1, ss));
      if (ss > 0 ∧ perm[s] ≤ k) {       /* true constant at left */
        if (perm[s + m] ≤ k) {       /* and also at right */
          if (¬addedC[ss − 1]) addedC[ss − 1] = 1, addto(subtotal, binom(m − 1, ss − 1));
          if (ss > 1 ∧ ¬addedD[ss − 2]) addedD[ss − 2] = 1, addto(subtotal, binom(m − 1, ss − 2));
        } else if (¬addedD[ss − 1]) addedD[ss − 1] = 1, addto(subtotal, binom(m − 1, ss − 1));
      }
    }
}
```

This code is used in section 12.

16. ⟨ Correct for constant nonbeads 16 ⟩ ≡

```
if (addedA[0]) nonbeads++;       /* all 0s */
if (addedD[m − 1]) nonbeads++;       /* all 1s */
```

This code is used in section 10.

17. Index.

add: 6, 9.
addedA: 1, 14, 15, 16.
addedB: 1, 14, 15.
addedC: 1, 14, 15.
addedD: 1, 14, 15, 16.
addto: 7, 11, 12, 15.
argc: 1.
bico: 1, 9.
binom: 9, 12, 15.
c: 6, 7.
clearnum: 5, 10.
exit: 6.
fprintf: 6.
grandtotal: 8, 11.
i: 1, 6, 7.
j: 1, 3, 5, 6, 7.
k: 1, 3, 5, 6, 7, 9.
l: 6, 7.
m: 1, 9.
main: 1.
mem: 1, 3, 5, 6, 7, 8.
memptr: 1, 4, 6, 8.
memsize: 1, 6.
n: 1.
nonbeads: 1, 10, 11, 12, 16.
numptr: 1, 4, 6, 8.
one: 8, 9.
p: 1.
perm: 1, 2, 12, 13, 15.
printf: 1, 2, 3, 4, 11.
printnum: 3, 4, 11.
rev: 1, 2.
s: 1.
ss: 1, 12, 15.
start: 1, 3, 5, 6, 7, 8.
stderr: 6.
subtotal: 8, 10, 11, 12, 15.
t: 1.
tnonbeads: 1, 4, 11.
tt: 1, 12, 15.
w: 1.
ww: 1, 12, 13.
zero: 8, 9.

⟨ Add contributions for a constant case 15 ⟩ Used in section 12.
⟨ Add contributions for slates (s, k) to *subtotal* 12 ⟩ Used in section 10.
⟨ Clear the four constant tables 14 ⟩ Used in section 10.
⟨ Compute b_k 10 ⟩ Used in section 1.
⟨ Correct for constant nonbeads 16 ⟩ Used in section 10.
⟨ Correct for nonconstant nonbeads 13 ⟩ Used in section 12.
⟨ Initialize *mem* 8 ⟩ Used in section 1.
⟨ Print b_k and add it to the grand total 11 ⟩ Used in section 1.
⟨ Print the grand total 4 ⟩ Used in section 1.
⟨ Set up the permutation, *perm* 2 ⟩ Used in section 1.
⟨ Subroutines 3, 5, 6, 7, 9 ⟩ Used in section 1.

HWB-FAST

	Section	Page
Intro	1	1
Decimal addition	3	3
The algorithm	10	6
Index	17	9