

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: 0R, 1R, ..., 255R */
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 bk 10>;
        <Print bk 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 $1 \leq j \leq n$, 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 $\{x_1, \dots, x_k\}$.

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 ⟩ \equiv

```

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

```

This code is used in section 1.

3. Decimal addition. The k th 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 k th number:

⟨Subroutines 3⟩ \equiv

```
void printnum(int k)
{
    register int j;
    for ( $j = start[k + 1] - 1$ ;  $j > start[k]$ ;  $j--$ )
        if ( $mem[j]$ ) break;
    for (;  $j \geq start[k]$ ;  $j--$ ) 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⟩ \equiv

```
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⟩ $+ \equiv$

```
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 ( $s = 0$ ;  $s \leq k$ ;  $s++$ ) {
    ⟨ 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 $[r_0, \dots, r_m]$, where r_j is 0 or 1 when $\text{perm}[s + j] \leq k$, otherwise r_j is x_l where $\text{perm}[s + j] = l$. (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 $ss + tt = w$, $ss \leq s$, and $tt \leq t$.

If at least one of the positions $\{r_1, \dots, r_{m-1}\}$ 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 ( $w = ww = j = 0$ ;  $j \leq m$ ;  $j++$ )
    if ( $\text{perm}[s + j] \leq k$ ) {
       $w++$ ;
      if ( $j > 0 \wedge j < m$ )  $ww++$ ;
    }
  if ( $ww \equiv m - 1$ ) < Add contributions for a constant case 15 >
  else {
    < Correct for nonconstant nonbeads 13 >;
    for ( $t = k - s$ ,  $ss = s$ ,  $tt = w - ss$ ;  $tt \leq t$ ;  $ss--$ ,  $tt++$ ) {
       $\text{addto}(\text{subtotal}, \text{binom}(w, ss));$ 
      if ( $ss \equiv p$ )  $\text{nonbeads}++$ ; /* see below */
    }
  }

```

This code is used in section 10.

13. Nonbeads $[r_0, \dots, r_m]$ 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) $[r_0, \dots, r_m] = [0, \dots, 0]$, within Type A; (d) $[r_0, \dots, r_m] = [1, \dots, 1]$, within Type D. Here we look for (a) and (b).

```

< Correct for nonconstant nonbeads 13 > ≡
   $p = n + 1$ ; /*  $n + 1$  is “infinity” */
  if ( $ww \equiv m - 2$ ) {
    if ( $m \equiv 2 \wedge \text{perm}[s + 1] \equiv n$ )  $p = (\text{perm}[s + 2] \leq k ? 1 : 0)$ ;
    else if ( $w \equiv m$ ) {
      for ( $p = 1$ ; ;  $p++$ )
        if ( $\text{perm}[s + p] > k$ ) break;
      if ( $\text{perm}[s + p] \neq k + 1$ )  $p = n + 1$ ;
    }
  }

```

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 $r = r_1 + \dots + r_{m-1}$. 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 ⟩ \equiv

```
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 ⟩ \equiv

```
{
  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 ⟩ \equiv

```
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

	1	2
	3	4
	5	6
	7	8
	9	10
	11	12
	13	14
	15	16
	17	18
	19	20
	21	22
	23	24
	25	26
	27	28
	29	30
	31	32
	33	34
	35	36
	37	38
	39	40
	41	42
	43	44
	45	46
	47	48
	49	50
	51	52
	53	54
	55	56
	57	58
	59	60
	61	62
	63	64
	65	66
	67	68
	69	70
	71	72
	73	74
	75	76
	77	78
	79	80
	81	82
	83	84
	85	86
	87	88
	89	90
	91	92
	93	94
	95	96
	97	98
	99	100
	101	102
	103	104
	105	106
	107	108
	109	110
	111	112
	113	114
	115	116
	117	118
	119	120
	121	122
	123	124
	125	126
	127	128
	129	130
	131	132
	133	134
	135	136
	137	138
	139	140
	141	142
	143	144
	145	146
	147	148
	149	150
	151	152
	153	154
	155	156
	157	158
	159	160
	161	162
	163	164
	165	166
	167	168
	169	170
	171	172
	173	174
	175	176
	177	178
	179	180
	181	182
	183	184
	185	186
	187	188
	189	190
	191	192
	193	194
	195	196
	197	198
	199	200
	201	202
	203	204
	205	206
	207	208
	209	210
	211	212
	213	214
	215	216
	217	218
	219	220
	221	222
	223	224
	225	226
	227	228
	229	230
	231	232
	233	234
	235	236
	237	238
	239	240
	241	242
	243	244
	245	246
	247	248
	249	250
	251	252
	253	254
	255	256
	257	258
	259	260
	261	262
	263	264
	265	266
	267	268
	269	270
	271	272
	273	274
	275	276
	277	278
	279	280
	281	282
	283	284
	285	286
	287	288
	289	290
	291	292
	293	294
	295	296
	297	298
	299	300
	301	302
	303	304
	305	306
	307	308
	309	310
	311	312
	313	314
	315	316
	317	318
	319	320
	321	322
	323	324
	325	326
	327	328
	329	330
	331	332
	333	334
	335	336
	337	338
	339	340
	341	342
	343	344
	345	346
	347	348
	349	350
	351	352
	353	354
	355	356
	357	358
	359	360
	361	362
	363	364
	365	366
	367	368
	369	370
	371	372
	373	374
	375	376
	377	378
	379	380
	381	382
	383	384
	385	386
	387	388
	389	390
	391	392
	393	394
	395	396
	397	398
	399	400
	401	402
	403	404
	405	406
	407	408
	409	410
	411	412
	413	414
	415	416
	417	418
	419	420
	421	422
	423	424
	425	426
	427	428
	429	430
	431	432
	433	434
	435	436
	437	438
	439	440
	441	442
	443	444
	445	446
	447	448
	449	450
	451	452
	453	454
	455	456
	457	458
	459	460
	461	462
	463	464
	465	466
	467	468
	469	470
	471	472
	473	474
	475	476
	477	478
	479	480
	481	482
	483	484
	485	486
	487	488
	489	490
	491	492
	493	494
	495	496
	497	498
	499	500
	501	502
	503	504
	505	506
	507	508
	509	510
	511	512
	513	514
	515	516
	517	518
	519	520
	521	522
	523	524
	525	526
	527	528
	529	530
	531	532
	533	534
	535	536
	537	538
	539	540
	541	542
	543	544
	545	546
	547	548
	549	550
	551	552
	553	554
	555	556
	557	558
	559	560
	561	562
	563	564
	565	566
	567	568
	569	570
	571	572
	573	574
	575	576
	577	578
	579	580
	581	582
	583	584
	585	586
	587	588
	589	590
	591	592
	593	594
	595	596
	597	598
	599	600
	601	602
	603	604
	605	606
	607	608
	609	610
	611	612
	613	614
	615	616
	617	618
	619	620
	621	622
	623	624
	625	626
	627	628
	629	630
	631	632
	633	634
	635	636
	637	638
	639	640
	641	642
	643	644
	645	646
	647	648
	649	650
	651	652
	653	654
	655	656
	657	658
	659	660
	661	662
	663	664
	665	666
	667	668
	669	670
	671	672
	673	674
	675	676
	677	678
	679	680
	681	682
	683	684
	685	686
	687	688
	689	690
	691	692
	693	694
	695	696
	697	698
	699	700
	701	702
	703	704
	705	706
	707	708
	709	710
	711	712
	713	714
	715	716
	717	718
	719	720
	721	722
	723	724
	725	726
	727	728
	729	730