June 27, 2011 at 03:29

1. Intro. This program is the twelfth in a series of exploratory studies by which I'm attempting to gain first-hand experience with OBDD structures, as I prepare Section 7.1.4 of *The Art of Computer Programming*. Again it's quite different from its predecessors: This one implements a new approach to finding an optimum ordering for the variables of a given Boolean function or set of functions. The new approach is based on QDDs (quasi-reduced BDDs), which undergo the "jump-up" operation but not the normal synthesis operations of a traditional BDD package. It requires no hashing or garbage collection.

The given function is specified explicitly by generating its QDD. As a demonstration, we implement the 2^m-way multiplexer $M_m(x_1,\ldots,x_m;x_{m+1},\ldots,x_{m+2^m})$ here, because it exhibits great extremes of BDD size under different orderings. But with `CWEB` change files any other function can be substituted.

If desired, optimization will be restricted to a subrange of the possible levels, keeping variables at the top and bottom of the BDD in place. We assume that $botvar - topvar$ is at most 24.

```
#define mm 2 /* order of MUX in this demonstration version */
#define nn (mm + (1 ≪ mm)) /* the number of Boolean variables */
#define outs 1 /* the number of Boolean functions to be simultaneously optimized */
#define interval 1000 /* make this larger to suppress progress reports */
#define worksize (1 ≪ 20) /* must be at most 1 ≪ 20 in this implementation */
  /* the jump-up work area will have 3 * worksize octabytes */
#define topvar 1 /* first variable whose order will be varied (must be ≥ 1 */
#define botvar nn /* last variable whose order will be varied (must be ≤ nn */
#define nnn (botvar + 1 - topvar) /* variables being permuted (at most 25) */
#define o mems++ /* count a memory access to an octabyte */
#define oo mems += 2 /* or two */
#define ooo mems += 3 /* or three */
#define oooo mems += 4 /* or four, wow */
#include <stdio.h>
#include <stdlib.h>
  ⟨ Type definitions 2 ⟩
  ⟨ Global variables 3 ⟩
  unsigned long long mems;
  ⟨ Subroutines 7 ⟩
  main( )
  {
    register int h, i, j, k, l, lo, hi, jj, kk, var, cycle;
    octa x;
    ⟨ Initialize everything 4 ⟩;
    for (cycle = 1; cycle < 1 ≪ (nnn - 1); cycle++) {
      if (cycle % interval ≡ 0) {
        printf("Beginning␣cycle␣%d␣(%llu␣mems␣so␣far)\n", cycle, mems);
        fflush(stdout);
      }
      ⟨ Do the jump-up for the current cycle 21 ⟩;
    }
    ⟨ Figure out an optimum order 27 ⟩;
    printf("Altogether␣%llu␣mems.\n", mems);
  }
```

2. ⟨ Type definitions 2 ⟩ ≡
typedef unsigned long long octa; /∗ an octabyte ∗/

See also section 8.

This code is used in section 1.

3. QDD representation. The quasi-reduced binary decision decision for a Boolean function of n variables has q_k nodes on level k, for $0 \le k \le n$, one for every distinct subfunction that can arise by hard-wiring constant values for the initial variables $(x_1, \ldots, x_k)$. The sequence $(q_0, \ldots, q_n)$ is called the function's *quasi-profile*.

The maximum value of q_k is $\min(t2^k, 2^{2^{n-k}})$ when there are t output functions of n input variables. In particular, the maximum quasi-profile when $n = 25$ and $t = 1$ is

$$(1, 2, 4, \ldots, 2^{19}, 2^{20}, 2^{16}, 2^8, 2^4, 2^2, 2);$$

its potentially biggest element, 2^{20}, occurs when $k = 20$.

In this implementation the nodes on level k are numbered from 0 to $q_k - 1$, and we store them consecutively in a big array $node[k]$. Each node contains three fields, (lo,hi,dep), packed into a 64-bit word. The lo and hi fields, 20 bits each, point to nodes at level $k+1$; the dep field, which occupies the other 24 bits, represents the set of variables on which this subfunction depends. Level n is special; it has the two "sink" nodes 0 and 1, which are represented by $(0, 0, 0)$ and $(1, 1, 0)$, respectively.

For example, suppose $n = 3$ and $f(x_1, x_2, x_3) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge x_3)$. The nodes at level 2, which correspond to branching on x_3, are $node[2][0] = (0, 0, 0)$, $node[2][1] = (1, 1, 0)$, and $node[2][2] = (0, 1, {}^{\#}4)$, representing the respective subfunctions 0, 1, x_3. The nodes at level 1, which correspond to branching on x_2, are $node[1][0] = (0, 1, {}^{\#}2)$ and $node[1][1] = (2, 2, {}^{\#}4)$, representing the subfunctions x_2 and x_3. And there's one node at level 0, namely $node[0][0] = (0, 1, {}^{\#}7)$.

⟨ Global variables 3 ⟩ ≡

```
octa *node[nnn + 1];      /* the node arrays */
int qq[nnn + 1];      /* the quasi-profile: qq[k] = q_{k-topvar+1} */
```

See also sections 9, 13, 15, 20, 24, and 28.

This code is used in section 1.

4. To launch this structure, we first need to allocate the *node* arrays. They are actually created only for levels $topvar - 1$ through $botvar$.

⟨ Initialize everything 4 ⟩ ≡

```
for (k = 0; k ≤ nnn; k++) {
  j = worksize, kk = k + topvar − 1;
  if (nn − kk < 5 ∧ j > 1 ≪ (1 ≪ (nn − kk))) j = 1 ≪ (1 ≪ (nn − kk));
  if (kk < 20 ∧ ((worksize/outs) ≫ kk) > 0 ∧ (j > outs ∗ (1 ≪ kk))) j = outs ∗ (1 ≪ kk);
  node[k] = (octa ∗) malloc(j ∗ sizeof(octa));
  if (¬node[k]) {
    fprintf(stderr, "I couldn't allocate %d octabytes for node[%d]!\n", j, k);
    exit(−2);
  }
}
```

See also sections 6, 16, and 22.

This code is used in section 1.

5. The packing policy is slightly unusual because I'm squeezing 65 bits into 64. The *dep* field represents the state of up to 25 variables, but only 24 bits are available.

Not to worry: The least significant bit of a 25-bit *dep* would truly be of the least possible significance in every way, because it would be zero everywhere except on level 0. And that bit on level 0 is almost never examined.

Therefore I don't store the least significant bit of *dep*. I store only the value $dp = dep \gg 1$, and I treat level 0 as a special case. This saves time even when $n < 25$.

(If I wanted to handle more than 25 variables, I could use a more elaborate scheme in which the packing conventions vary from level to level. But 25 is plenty for me at the moment.)

#define $pack(lo, hi, dp)$ $(((((\mathbf{octa})(dp) \ll 20) + (lo)) \ll 20) + (hi))$
#define $lofield(x)$ $(((x) \gg 20)\ \&\ {}^{\#}\texttt{fffff})$
#define $hifield(x)$ $((x)\ \&\ {}^{\#}\texttt{fffff})$
#define $depfield(x)$ $(((x) \gg 40) \ll 1)$
#define $extrabit(k, j)$ $(k \equiv 0\ ?\ lofield(node[0][j]) \neq hifield(node[0][j]) : 0)$

6. Once the *node* arrays exist, we can set up the initial QDD. Here I implement the familiar function

$$M_m(x_1, \ldots, x_m; x_{m+1}, \ldots, x_{m+2^m}) \ =\ x_{m+1+(x_1 \ldots x_m)_2}.$$

(In fact, the example $f(x_1, x_2, x_3)$ above is $M_1(x_1; x_2, x_3)$.)

Only the *lo* and *hi* fields are initialized here, because we will use the reduction routine to compute appropriate *dep*s.

The following code assumes that $topvar = 1$ and $botvar = nn$.

```
⟨ Initialize everything 4 ⟩ +≡
  for (k = 0; k < mm; k++) {
    for (j = 0; j < 1 ≪ k; j++) node[k][j] = pack(j + j, j + j + 1, 0);
    qq[k] = 1 ≪ k;
  }
  for (j = 0; j < 1 ≪ mm; j++)
    if (j ≡ 0) node[mm][j] = pack(0, 1, 0);
    else node[mm][j] = pack(j + 1, j + 1, 0);
  qq[mm] = 1 ≪ mm;
  for (k = mm + 1; k ≤ nn; k++) {
    for (j = 0; j < nn + 2 − k; j++)
      if (j < 2) node[k][j] = pack(j, j, 0);
      else if (j ≡ 2) node[k][j] = pack(0, 1, 0);
      else node[k][j] = pack(j − 1, j − 1, 0);
    qq[k] = nn + 2 − k;
  }     /* N.B.: now qq[nn] = 2, and the sink nodes have been initialized */
  ⟨ Compute the dep fields of the initial QDD 14 ⟩;
```

7. For diagnostic purposes, I might want to pretty-print the current QDD.

⟨ Subroutines 7 ⟩ ≡

```
void print_level(int k)
{
  register int j;
  printf("level␣%d␣(x%d):\n", k, map[k] + topvar);
  for (j = 0; j < qq[k]; j++)
    printf("␣%d,%d␣%x\n", lofield(node[k][j]), hifield(node[k][j]), depfield(node[k][j]) + extrabit(k, j));
}
void print_qbdd(void)
{
  register int k;
  for (k = topvar − 1; k < botvar; k++) print_level(k);
}
```

See also sections 10 and 11.

This code is used in section 1.

8. Reduction. Restructurings of the QDD are implemented with the help of a work area, capable of holding twice as many nodes as needed on any particular level.

In the work area, nodes appear in octabytes of type *pair*, which simply have two integer fields, l and r.

Two large arrays, *work* and *head*, comprise the work area. A node with fields (lo, hi) is represented in a *pair* that has $l = hi$; this pair, which appears in the *work* array, is part of a linked list that begins $head[lo]$. Links appear in the r fields.

The work area is intended for nodes that are destined to become part of $node[k]$, for some level k. The lo and hi fields, which are addresses of nodes at the next level, must therefore be less than $qq[k+1]$.

```
⟨ Type definitions 2 ⟩ +≡
  typedef struct { unsigned int l, r; } pair;
```

9.
```
⟨ Global variables 3 ⟩ +≡
  pair work[2 * worksize], head[worksize];      /* the big workspace */
```

10. We use $j+1$ to link to entry j, so that null links are distinguishable from links to entry 0.

```
⟨ Subroutines 7 ⟩ +≡
  void print_work(int k)
  {
    register int lo, hi;
    printf("Current␣workspace␣for␣level␣%d:\n", k);
    for (lo = 0; lo < qq[k + 1]; lo++)
      for (hi = head[lo].r; hi; hi = work[hi - 1].r) printf("␣%d,%d\n", lo, work[hi - 1].l);
  }
```

11. The key subroutine we need is called *reduce*. It finds the unique combinations (lo, hi) in the work area, and places them into $node[k]$ with proper *dep* fields. It resets $qq[k]$ to the number of distinct nodes found.

The *reduce* routine also sets $clone[j]$ to the new address of the node that was originally represented in $work[j]$.

"Bucket sorting" is the basic idea here. When we're working on list $head[lo]$, we set $head[hi].l$ to the address of an existing node (lo, hi), if *reduce* has already created such a node.

```
⟨ Subroutines 7 ⟩ +≡
  void reduce(int k)
  {
    register int lo, hi, dep, p, nextp, q;
    q = 0;      /* the number of nodes created so far */
    for (o, lo = 0; lo < qq[k + 1]; lo++) {
      for (o, p = head[lo].r; p; p = nextp) {
        o, nextp = work[p - 1].r;
        hi = work[p - 1].l;      /* the hi field of a node with the current lo */
        if (o, head[hi].l) o, clone[p - 1] = head[hi].l - 1;      /* (lo, hi) already exists */
        else ⟨ Create a new entry in node[k] 12 ⟩;
      }
      for (p = head[lo].r; p; p = nextp) {
        o, nextp = work[p - 1].r, hi = work[p - 1].l;
        o, head[hi].l = 0;      /* clean up */
      }
      o, head[lo].r = 0;      /* reset the lo list to empty */
    }
    o, qq[k] = q;
  }
```

12. ⟨ Create a new entry in $node[k]$ 12 ⟩ ≡

```
{
  if (q ≥ worksize) {
    fprintf(stderr, "Sorry, level %d of the QDD is too big (worksize=%d)!\n", k, worksize);
    exit(−3);
  }
  if (lo ≠ hi) oo, dep = depfield(node[k + 1][lo] | node[k + 1][hi]) | (1 ≪ k);
  else o, dep = depfield(node[k + 1][lo]);
  o, node[k][q] = pack(lo, hi, dep ≫ 1);
  o, clone[p − 1] = q;
  o, head[hi].l = ++q;
}
```

This code is used in section 11.

13. ⟨ Global variables 3 ⟩ +≡

```
int clone[2 * worksize];      /* the new node addresses */
```

14. Let's illustrate *reduce* by using it to tidy up the initial QDD. The main point of interest is that we must remap the pointers on level $k − 1$ after level k has been reduced.

```
#define makework(j, lo, hi) work[j].l = hi, work[j].r = head[lo].r, head[lo].r = j + 1
```

⟨ Compute the *dep* fields of the initial QDD 14 ⟩ ≡

```
for (k = nnn − 1; ; k−−) {
  for (o, j = 0; j < qq[k]; j++) {
    o, lo = lofield(node[k][j]);
    ooo, makework(j, lo, hifield(node[k][j]));
  }
  reduce(k);
  if (k ≡ 0) break;
  for (o, j = 0; j < qq[k − 1]; j++) {
    o, x = node[k − 1][j];
    ooo, node[k − 1][j] = pack(clone[lofield(x)], clone[hifield(x)], 0);
  }
}
```

This code is used in section 6.

15. Jumping. Our chief goal is to study what happens when the variables of our function are permuted. We'll see later that only 2^{n-1} of the $n!$ permutations need to be examined, and that each of them can be obtained from its predecessor by a simple "jump-up" operation.

The jump-up operation $k \to j$, where $k > j$, takes the variable at level k of the branching structure and moves it up to level j, sliding the previous occupants of levels $(j, j+1, \ldots, k-1)$ down one notch. For example, suppose $n = 7$ and we do jump-up $5 \to 2$. Before the operation, levels 0 thru 6 represent branching on variables $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, respectively; after jumping, they represent branching on $(x_1, x_2, x_6, x_3, x_4, x_5, x_7)$.

The *dep* fields are always based on the assumption that level k branches on variable x_{k+1}. An auxiliary *map* table records the results of previous jumps; level k actually branches on variable $x_{map[k]+topvar}$ of the original unpermuted function. We also maintain a *bitmap* array, where $bitmap[k]$ contains the bits that encode the set $\{map[0], \ldots, map[k-1]\}$.

```
⟨ Global variables 3 ⟩ +≡
  int map[nnn];      /* the current permutation */
  int bitmap[nnn];   /* its initial dependency sets */
```

16.
```
⟨ Initialize everything 4 ⟩ +≡
  for (j = k = 0; j < nnn; k += 1 ≪ j, j++) o, map[j] = j, bitmap[j] = k;
```

17. The basic idea for jumping up is quite simple. We essentially make two copies of levels j thru $k-1$, one for the case when $x_{k+1} = 0$ and one for the case when $x_{k+1} = 1$. Those copies are moved to levels $j+1$ thru k, and reduced to eliminate duplicates. Finally, every original node on level j is replaced by a node that branches on x_{k+1}. The *lo* branch of this new node is to the 0-copy of the old node, and the *hi* branch is to the 1-copy.

Because of the remapping that takes place here, the *dep* fields in levels less than *jj* are no longer correct. But we will do jump-ups in a controlled manner, so that those *dep* fields are never actually examined.

```
⟨ Jump up from kk to jj 17 ⟩ ≡
  o, var = map[kk];
  for (k = kk; k > jj; k--) {
    ⟨ Make two copies of level k − 1 in the work area 18 ⟩;
    reduce(k);
    oo, map[k] = map[k − 1];
    oo, bitmap[k] = bitmap[k − 1] | (1 ≪ var);
  }
  ⟨ Remake level jj 19 ⟩;
```

This code is used in section 21.

18. Simple but cool list processing does the trick here.

```
⟨ Make two copies of level k − 1 in the work area 18 ⟩ ≡
  for (o, j = 0; j < qq[k − 1]; j++) {
    o, x = node[k − 1][j], l = lofield(x), h = hifield(x);
    if (k ≡ kk) {
      oo, lo = lofield(node[k][l]), hi = lofield(node[k][h]);
      ooo, makework(j + j, lo, hi);
      lo = hifield(node[k][l]), hi = hifield(node[k][h]);
      ooo, makework(j + j + 1, lo, hi);
    } else {
      oo, lo = clone[l + l], hi = clone[h + h];
      ooo, makework(j + j, lo, hi);
      oo, lo = clone[l + l + 1], hi = clone[h + h + 1];
      ooo, makework(j + j + 1, lo, hi);
    }
  }
```

This code is used in section 17.

19. As mentioned earlier, we needn't worry about the integrity of the *dep* fields in levels less than *jj*.

```
⟨ Remake level jj 19 ⟩ ≡
  for (o, j = 0; j < qq[jj]; j++) {
    oo, lo = clone[j + j], hi = clone[j + j + 1];
    ooo, makework(j, lo, hi);
  }
  reduce(jj);
  o, map[jj] = var;
  if (jj)
    for (o, j = 0; j < qq[jj − 1]; j++) {
      o, x = node[jj − 1][j];
      ooo, node[jj − 1][j] = pack(clone[lofield(x)], clone[hifield(x)], 0);
    }
```

This code is used in section 17.

20. The algorithm. OK, we've now got a beautiful infrastructure to work with. But the reader may well wonder why we've been building it.

Perhaps I should have presented this program in a top-down way, starting with the statement of the problem (which is to find the optimum ordering of variables) and then explaining how to reduce that problem to list processing.

Well then, to start at the beginning, we can note that the main problem can be reduced to smaller problems of the same kind. Namely, for any given subset X of the variables $\{x_1, \ldots, x_n\}$, where X has k elements, we can ask what ordering of those variables minimizes the profile of the first k levels, when those variables are required to be tested first. (Here I'm talking about the real profile $(b_0, \ldots, b_n)$, not the quasi-profile $(q_0, \ldots, q_n)$.)

If we've solved that problem for all subsets X of size $k < n$, we can solve it for all subsets of size $k + 1$, as follows: Find a QDD in which the elements of X appear on the levels 0 through $k - 1$. For each $x \notin X$, count the number of elements on level k that depend on x; that is the value of b_k that will appear in the profile of any BDD in which the elements of X are tested first. Add that value to the minimum cost of X on the previous levels, thereby getting a candidate for the minimum cost of $X \cup x$. After trying all X and all x, we'll know the minimum costs for all $k + 1$-element subsets.

Notice that every QDD has information about n different subsets at once, namely the subsets of elements on its initial k levels for $1 \le k \le n$. Therefore, instead of working only on small subsets before large ones, this program gathers data for all sizes simultaneously; we can determine the actual minimum costs later.

(A completely different strategy, which uses branch-and-bound methods to avoid subsets that are known to be nonoptimum, will work better on many Boolean functions. I plan to implement that method too, for comparison purposes. But the method considered here is advantageous for the hard functions that can make branch-and-bound examine too many branches.)

For each variable x_{j+1} and each k-element set X with $x_{j+1} \notin X$, we will set $b[X][j]$ to the profile element b_k that would occur when x_{j+1} is at level k of a BDD and when the variables of X occupy levels 0 through $k-1$. The subset X is encoded as a binary number, $\sum\{2^{i-1} \mid x_i \in X\}$.

```
⟨ Global variables 3 ⟩ +≡
  int b[1 ≪ nnn][nnn];      /* transition data for BDDs */
```

21. Now I must explain a beautiful pattern: There's a simple way to produce all the QDDs we need, using a relatively short sequence of jump-ups.

In essence, we want a sequence of permutations with the property that every k-element subset of $\{1, \ldots, n\}$ appears as the first k elements of some permutation, and such that we can get from each permutation to its successor by a single jump-up. There's a nice way to do this with a sequence of length 2^{n-1}: When $n = 1$, the permutation is simply '1'. When $n > 1$, take the sequence for $n - 1$ and place n at the bottom; then jump n up to the top; then use the sequence for $n - 1$ on the *lower* $n - 1$ elements. This idea turns out to be equivalent to the following: The kth jump-up is $\nu k + \rho k \to \nu k - 1$, for $1 \le k < 2^{n-1}$. (In this formula νk denotes the number of 1s in the binary representation of k, and ρk denotes the number of 0s at the right end of that representation.) For example, when $n = 4$ the jumps are $1 \to 0$, $2 \to 0$, $2 \to 1$, $3 \to 0$, $2 \to 1$, $3 \to 1$, $3 \to 2$; the permutations are

$$1234,\ 2134,\ 3214,\ 3124,\ 4312,\ 4132,\ 4213,\ 4231.$$

The idea is that all k-combinations that don't contain n appear in the first half; those that do contain n appear in the second half.

```
⟨ Do the jump-up for the current cycle 21 ⟩ ≡
  for (jj = 0, kk = 1, k = cycle; (k & 1) ≡ 0; kk++) k ≫= 1;      /* compute ρk */
  for (k &= k − 1; k; jj++, kk++) k &= k − 1;      /* compute νk */
  ⟨ Jump up from kk to jj 17 ⟩;
  for (k = jj + 1; k ≤ kk; k++) ⟨ Gather statistics from level k of the current QDD 23 ⟩;
```

This code is used in section 1.

22. We also need to gather statistics from the initial QDD.

⟨ Initialize everything 4 ⟩ +≡
  ⟨ Gather statistics from level 0 of the current QDD 26 ⟩;
  for ($k = 1$; $k < nnn$; k++) ⟨ Gather statistics from level k of the current QDD 23 ⟩;

23. Every node in *node*[*k*] is marked with a *dep* field, telling which of the remaining variables it depends on. More precisely, if the *dep* contains the bit $1 \ll i$, there's a dependency on x_j, where $j = map[i]$.

There might be lots and lots of nodes, so I'd like to examine each *dep* field only once. (In fact, I could have gathered the stats while doing *reduce* during the jump-up; so I won't charge any mems for fetching the node here.) There are three bytes in the *dp* field, so I'll count how many nodes have each possible pattern in each of those bytes. Usually only one or two of the bytes can be nonzero; so the routine breaks into six rather tedious cases.

```
⟨ Gather statistics from level k of the current QDD 23 ⟩ ≡
  {
    switch (((nnn − 2) ≫ 3) ∗ 4 + ((k − 1) ≫ 3)) {
    case 2 ∗ 4 + 0: for (j = 0; j < 256; j++) ooo, count1[j] = count2[j] = count3[j] = 0;
      for (o, j = 0; j < qq[k]; j++) {
        x = node[k][j];      /∗ mems not charged, see above ∗/
        oo, count1[x ≫ 56]++;
        oo, count2[(x ≫ 48) & #ff]++;
        oo, count3[(x ≫ 40) & #ff]++;
      }
      ; break;
    case 2 ∗ 4 + 1: for (j = 0; j < 256; j++) oo, count1[j] = count2[j] = 0;
      for (o, j = 0; j < qq[k]; j++) {
        x = node[k][j];      /∗ mems not charged, see above ∗/
        oo, count1[x ≫ 56]++;
        oo, count2[(x ≫ 48) & #ff]++;
      }
      ; break;
    case 2 ∗ 4 + 2: for (j = 0; j < 256; j++) o, count1[j] = 0;
      for (o, j = 0; j < qq[k]; j++) {
        x = node[k][j];      /∗ mems not charged, see above ∗/
        oo, count1[x ≫ 56]++;
      }
      ; break;
    case 1 ∗ 4 + 0: for (j = 0; j < 256; j++) oo, count2[j] = count3[j] = 0;
      for (o, j = 0; j < qq[k]; j++) {
        x = node[k][j];      /∗ mems not charged, see above ∗/
        oo, count2[(x ≫ 48) & #ff]++;
        oo, count3[(x ≫ 40) & #ff]++;
      }
      ; break;
    case 1 ∗ 4 + 1: for (j = 0; j < 256; j++) o, count2[j] = 0;
      for (o, j = 0; j < qq[k]; j++) {
        x = node[k][j];      /∗ mems not charged, see above ∗/
        oo, count2[(x ≫ 48) & #ff]++;
      }
      ; break;
    case 0 ∗ 4 + 0: for (j = 0; j < 256; j++) o, count3[j] = 0;
      for (o, j = 0; j < qq[k]; j++) {
        x = node[k][j];      /∗ mems not charged, see above ∗/
        oo, count3[(x ≫ 40) & #ff]++;
      }
      ; break;
    }
    for (j = k; j < nnn; j++) ⟨ Gather stats for variable j at level k 25 ⟩;
```

```
}
```
This code is used in sections 21 and 22.

24. ⟨ Global variables 3 ⟩ +≡
```
int count1[256], count2[256], count3[256];      /* bit pattern counts */
```

25. ⟨ Gather stats for variable j at level k 25 ⟩ ≡
```
{
  l = 1 ≪ ((j − 1) & #7), i = 0;
  if (j < 9) {
    for (i = 0, h = l; h < 256; h = (h + 1) | l) o, i += count3[h];
  } else if (j < 17) {
    for (i = 0, h = l; h < 256; h = (h + 1) | l) o, i += count2[h];
  } else
    for (i = 0, h = l; h < 256; h = (h + 1) | l) o, i += count1[h];
  oooo, b[bitmap[k]][map[j]] = i;
}
```
This code is used in section 23.

26. At this point we have the initial QDD, with $map[j] = j$ for all j.

⟨ Gather statistics from level 0 of the current QDD 26 ⟩ ≡
```
for (j = 0; j < qq[0]; j++) {
  o, i = depfield(node[0][j]) + extrabit(0, j);
  for (k = 0; k < nnn; k++) o, b[0][k] += (i ≫ k) & 1;
}
```
This code is used in section 22.

27. At last comes the final reckoning: We can compute the minimum cost that can be achieved when subset X occurs at the top of the BDD, for $1 \le X < 2^n$.

While we're at it, we might as well compute the maximum cost too.

```
⟨ Figure out an optimum order 27 ⟩ ≡
  for (k = 1; k < 1 ≪ nnn; k++) {      /* k represents a subset X */
    h = 1 ≪ nnn;      /* infinite cost */
    for (j = 0, i = 1; j < nnn; j++, i ≪= 1)
      if (k & i) {
        oo, l = cost[k ⊕ i] + b[k ⊕ i][j];      /* the cost if x_{j+1} comes last */
        if (l < h) h = l, lo = j;
      }
    oo, cost[k] = h, routing[k] = lo;
  }
  printf("Optimum ordering (cost %d+externals) can be achieved thus:\n", cost[(1 ≪ nnn) − 1]);
  for (j = nnn − 1, k = (1 ≪ nnn) − 1; k; j−−, k ⊕= 1 ≪ routing[k])
    printf(" level %d, x%d (%d)\n", j, routing[k] + topvar, b[k ⊕ (1 ≪ routing[k])][routing[k]]);
  for (k = 1; k < 1 ≪ nnn; k++) {      /* k represents a subset X */
    h = 0;
    for (j = 0, i = 1; j < nnn; j++, i ≪= 1)
      if (k & i) {
        l = cost[k ⊕ i] + b[k ⊕ i][j];      /* the cost if x_{j+1} comes last */
        if (l > h) h = l, lo = j;
      }
    cost[k] = h, routing[k] = lo;
  }
  printf("Pessimum ordering (cost %d+externals) can be achieved thus:\n", cost[(1 ≪ nnn) − 1]);
  for (j = nnn − 1, k = (1 ≪ nnn) − 1; k; j−−, k ⊕= 1 ≪ routing[k])
    printf(" level %d, x%d (%d)\n", j, routing[k] + topvar, b[k ⊕ (1 ≪ routing[k])][routing[k]]);
```

This code is used in section 1.

28.

```
⟨ Global variables 3 ⟩ +≡
  int cost[1 ≪ nnn];      /* the optimum node count for each bitmap */
  char routing[1 ≪ nnn];      /* the variable to put last in the optimum order */
```

29. Index.

BDD12