June 27, 2011 at 03:46

1. Introduction. This program counts the number of knight's tours of an $m\times n$ board that are symmetric under 180° rotation, assuming that m and n are even. I wrote it partly to verify the results of another program, using an independent method; but mostly, I wrote it to get experience using "ordered binary decision diagrams," popularly known as OBDDs. I'm implementing a basic form of OBDDs as described by Bryant in *Computing Surveys* **24** (1992), 293–318.

The idea of the program is that each tour is obtained by combining two perfect matchings of the bipartite graph of knight's moves. So I generate an OBDD to represent perfect matchings. Then I traverse it, reporting all the pairs of matchings that yield a single cycle.

```
#define mm 8    /* the number of rows */
#define nn 8    /* the number of columns */
#define interval 100000    /* show 1/interval of the solutions */
#include "gb_graph.h"    /* the GraphBase data structures */
#include "gb_basic.h"    /* chessboard graph generator */
  ⟨ Preprocessor definitions ⟩
  ⟨ Global variables 3 ⟩
  ⟨ Subroutines 7 ⟩
  main()
  {
    ⟨ Local variables 4 ⟩;
    ⟨ Generate the list of edges 2 ⟩;
    ⟨ Construct the OBDD for all perfect matchings 8 ⟩;
    ⟨ Count and report the number of such matchings 21 ⟩;
    ⟨ Traverse and count Hamiltonian cycles 24 ⟩;
    printf("Total␣%d␣solutions␣and␣%d␣pseudo-solutions.\n", sols, pseudo_sols);
  }
```

2. To account for 180° symmetry, we identify each vertex with its "mate" under rotation. Each edge k is represented by three quantities: *black*[k] and *red*[k] are the endpoints (which are vertices of different colors on the chessboard; black vertices are those with an even sum of coordinates); *parity*[k] is 1 if the edge actually goes from *black*[k] to the mate of *red*[k] instead of to *red*[k] itself.

```
#define mate u.V   /* the antipodal vertex to a given one */
```

⟨ Generate the list of edges 2 ⟩ ≡

```
{
  gg = board(mm, nn, 0, 0, 5, 0, 0);   /* knight moves on chessboard */
  for (v = gg→vertices, u = gg→vertices + gg→n − 1; v < u; v++, u−−) {
    v→mate = u;
    u→mate = v;
  }
  k = 0;
  for (v = gg→vertices; v < v→mate; v++)
    if (((v→x.I + v→y.I) & 1) ≡ 0) {
      register Arc *a;
      for (a = v→arcs; a; a = a→next) {
        u = a→tip;
        black[k] = v;
        if (u < u→mate) red[k] = u, parity[k] = 0;
        else red[k] = u→mate, parity[k] = 1;
        if (verbose) printf("%d:␣%s--%s%s\n", k, black[k]→name, red[k]→name, parity[k] ? "*" : "");
        k++;
      }
    }
  edges = k;
}
```

This code is used in section 1.

3. ⟨ Global variables 3 ⟩ ≡

```
Vertex *black[mm * nn * 2], *red[mm * nn * 2];
int parity[mm * nn * 2];
int edges;   /* total number of edges */
int verbose;   /* set nonzero when debugging */
int sols, pseudo_sols;   /* counts the solutions and cases of two half-cycles */
```

See also sections 5, 12, 14, 20, and 26.

This code is used in section 1.

4. ⟨ Local variables 4 ⟩ ≡

```
register int j, k, t;
register Vertex *u, *v;
Graph *gg;
```

See also sections 10 and 25.

This code is used in section 1.

5. OBDDs. An OBDD canonically represents a boolean function $f(x_1, x_2, \ldots, x_n)$ as a binary tree with shared subtrees (a special kind of dag). If $n = 0$, the representation is the node '0' or '1'. If $n > 0$ and the function doesn't depend on x_1, rep$f(x_1, x_2, \ldots, x_n) =$ rep$f(x_2, \ldots, x_n)$. Otherwise rep$f(x_1, x_2, \ldots, x_n)$ is a node labeled x_1 with left and right subnodes labeled rep$f(0, x_2, \ldots, x_n)$ and rep$f(1, x_2, \ldots, x_n)$ respectively. Common subtrees are represented by the same node.

In the present application there is one boolean variable for each edge. The variable is 1 if the edge is present in a certain subset of edges, 0 otherwise. The function $f(e_1, e_2, \ldots, e_n)$ is 1 iff that subset of edges is a perfect matching.

I don't expect the number of nodes to be enormous. So I'm preallocating an array for each node field: $var[k]$ is the label of node k; $left[k]$ and $right[k]$ are the indices of its subnodes. Usually $0 \le var[k] < edges$; however, the two "sink" nodes 0 and 1 are special. They are the nodes in positions 0 and 1, and we have $var[k] = edges$, $left[k] = right[k] = k$ for these two values of k.

#define *max_nodes* $(1 \ll 18)$ /* must be a power of 2 because of my hash function below */

⟨ Global variables 3 ⟩ +≡
 int *var*[*max_nodes*], *left*[*max_nodes*], *right*[*max_nodes*]; /* OBDD storage */
 int *curnode*; /* size of the current OBDD */

6. To get started, I set up a simple OBDD for the function f that says every black vertex is matched exactly once.

If $e_1, \ldots, e_n$ are the edges that touch some vertex, we want exactly one of them to be present. The OBDD for this has $2n$ nodes

$$\alpha_j = (e_j, \alpha_{j+1}, \beta_{j+1}), \qquad \beta_j = (e_j, \beta_{j+1}, 0), \qquad \text{for } 1 \le j \le n$$

where $\alpha_{n+1} = 0$ and $\beta_{n+1} = 1$. Actually only $2n - 1$ of these nodes are present, since β_1 is not used.

We string together these simple OBDDs by substituting node α_1 of the $k+1$st vertex for node β_{n+1} of the kth. This works because of the way we have numbered the edges: the *black* array values are nondecreasing.

⟨ Create the OBDD for matching black vertices 6 ⟩ ≡

```
var[0] = var[1] = edges;
left[0] = right[0] = 0;
left[1] = right[1] = 1;
curnode = 2;
for (v = gg→vertices, k = 0; v < v→mate; v++)
  if (((v→x.I + v→y.I) & 1) ≡ 0) {
    j = 0;
    while (black[k] ≡ v) {
      if (j) {    /* put out a β node */
        var[curnode] = k;
        left[curnode] = curnode + 2;
        right[curnode] = 0;
        curnode++;
      }
      var[curnode] = k;
      left[curnode] = curnode + 2;
      right[curnode] = curnode + 1;
      curnode++;    /* that was an α node */
      k++;
      j = 1;
    }
    left[curnode − 1] = 0;    /* α_{n+1} = 0; β_{n+1} = curnode */
  }
left[curnode − 2] = right[curnode − 1] = 1;    /* β_{n+1} = 1, the last time */
```

This code is used in section 8.

7. Here's a subroutine for use when debugging: It prints the current OBDD.

⟨ Subroutines 7 ⟩ ≡

```
void print_obdd()
{
  register int k;
  for (k = 2; k < curnode; k++) printf("%d:␣if␣%s-%s%s␣then␣%d␣else␣%d\n", k,
      black[var[k]]→name, red[var[k]]→name, parity[var[k]] ? "*" : "", right[k], left[k]);
}
```

See also section 11.

This code is used in section 1.

8. To complete the construction of the OBDD for matching, we need to specify the fact that every red vertex is matched exactly once. This is done by repeatedly ANDing an appropriate boolean function to the current OBDD, once for each red vertex.

```
⟨ Construct the OBDD for all perfect matchings 8 ⟩ ≡
  ⟨ Create the OBDD for matching black vertices 6 ⟩;
  f = 2;    /* root of the current OBDD */
  for (v = gg→vertices; v < v→mate; v++)
    if ((v→x.I + v→y.I) & 1) ⟨ Modify the OBDD so that red vertex v is matched exactly once 9 ⟩;
```

This code is used in section 1.

9. The reader may have noticed that I forgot to test whether *curnode* has exceeded *max_nodes*. Peccavi; I am silently assuming that *max_nodes* isn't way too low. In the *intersect* routine below this condition is rigorously checked.

```
⟨ Modify the OBDD so that red vertex v is matched exactly once 9 ⟩ ≡
  {
    g = curnode;    /* root of an OBDD to be ANDed to f */
    for (j = k = 0; k < edges; k++)
      if (red[k] ≡ v) {
        if (j) {    /* put out a β node */
          var[curnode] = k;
          left[curnode] = curnode + 2;
          right[curnode] = 0;
          curnode++;
        }
        var[curnode] = k;
        left[curnode] = curnode + 2;
        right[curnode] = curnode + 1;
        curnode++;    /* that was an α node */
        j = 1;
      }
    left[curnode − 1] = 0;    /* α_{n+1} = 0 */
    left[curnode − 2] = right[curnode − 1] = 1;    /* β_{n+1} = 1 */
    f = intersect(f, g);
  }
```

This code is used in section 8.

10. ⟨ Local variables 4 ⟩ +≡

```
  int f, g;    /* roots of OBDDs */
```

11. Intersection of OBDDs. Now comes the funnest part. Given the roots f, g of two OBDDs, the following subroutine computes the OBDD for $f \wedge g$.

We assume that f and g occupy the low end of memory, up to but not including *curnode*. The subroutine operates in two phases: First an unreduced template for the result is formed in the upper part of memory. Then the reduced OBDD is placed in the lower part, on top of the original f and g. This method allows us to avoid messy issues of reference counting and garbage collection. It does, however, require us to copy the whole OBDD if, for example, g is the constant 1. Such copying is, fortunately, only a small part of the work, in the OBDDs we will encounter.

The output $f \wedge g$ is constructed in a useful form that can be processed "bottom up," because $left[k] < k$ and $right[k] < k$ will hold in all nodes. The inputs need not be in this form.

```
⟨ Subroutines 7 ⟩ +≡
  ⟨ Basic subroutines needed by intersect 16 ⟩
  int intersect(f, g)
      register int f, g;    /* roots of OBDDs whose intersection is desired */
  {
    register int j, k;
    hinode = max_nodes − 1;
    ⟨ Construct the template in upper memory 13 ⟩;
    ⟨ Construct the reduced OBDD in lower memory, using the template 18 ⟩;
    if (verbose) printf("␣...␣unreduced␣size␣%d,␣reduced␣%d\n", max_nodes − hinode, curnode);
    return curnode − 1;
  }
```

12. ⟨ Global variables 3 ⟩ +≡

```
  int hinode;    /* the first free node in upper memory */
```

13. What's a template? Well, it's sort of like an OBDD except that it hasn't been reduced to canonical form. Also, it represents the variables in a different way. The *var* field of a node contains a pointer to the previous node for the same variable; there's a separate array called *head* that points to the first node for each variable. This arrangement makes it easy to look at nodes level by level from the bottom up.

While the template is being formed, some of its nodes are not yet finished. An unfinished node k represents a function $f' \wedge g'$, where $left[k]$ points to f' and $right[k]$ points to g'; its *var* part is undefined. All unfinished template nodes belong to a queue of consecutive nodes in upper memory; they will be finished in FIFO order.

The subroutine *new_template* creates a new (unfinished) template node for $f' \wedge g'$, if no (finished or unfinished) node for this pair of functions already exists. Otherwise it returns the value of the existing node.

```
⟨ Construct the template in upper memory 13 ⟩ ≡
  {
    register int source;     /* front of the queue of unfinished template nodes */
    ⟨ Initialize the tables for template construction 15 ⟩;
    k = new_template(f, g);     /* create the first unfinished node */
    source = max_nodes − 1;
    while (source > hinode) {     /* we want to finish node source */
      f = left[source]; g = right[source];     /* by intersecting nonzero functions f, g */
      j = var[f]; k = var[g];
      left[source] = new_template(j > k ? f : left[f], k > j ? g : left[g]);
      right[source] = new_template(j > k ? f : right[f], k > j ? g : right[g]);
      if (j > k) j = k;     /* this template node refers to variable j */
      var[source] = head[j];
      head[j] = source;     /* so link it into list j */
      source−−;
    }
  }
```

This code is used in section 11.

14. The *new_template* routine recognizes previous entries by maintaining a hash table of all node pairs it has seen. The hash table consists of two arrays, *hash_f* and *hash_g*, for the two function nodes; these point into lower memory, and they serve as retrieval keys. A third array, *hash_l*, is the location of the template node for $hash\_f \wedge hash\_g$. There's also a fourth array, *hash_t*, which contains a "time stamp." Any slot whose time stamp differs from the global variable *time* is considered empty. Linear probing works well, since the hash table rarely if ever gets more than half full.

```
⟨ Global variables 3 ⟩ +≡
  int time;     /* the master clock for timestamps */
  int hash_f[max_nodes], hash_g[max_nodes], hash_l[max_nodes], hash_t[max_nodes];
  int head[mm * nn * 2];     /* head of lists for template variables */
```

15. ⟨ Initialize the tables for template construction 15 ⟩ ≡

```
  time++;     /* clear the memory of the new_template routine */
  for (k = 0; k ≤ edges; k++) head[k] = 0;
```

This code is used in section 13.

16. I forgot to mention that the *new_template* routine returns 0 if either input function is the constant 0. This feature, in fact, is what makes the *intersect* routine compute intersections(!).

#define *hash_rand* 314159 /* $(1001100101100101111)_2$; this "random" multiplier seems OK */

⟨ Basic subroutines needed by *intersect* 16 ⟩ ≡

```
int new_template(f, g)
    register int f, g;
{
  register int h;
  if (f ≡ 0 ∨ g ≡ 0) return 0;
  h = (hash_rand * f + g) & (max_nodes − 1);    /* hash function */
  while (1) {
    if (hash_t[h] ≠ time) break;
    if (hash_f[h] ≡ f ∧ hash_g[h] ≡ g) return hash_l[h];
    h = (h − 1) & (max_nodes − 1);
  }
  hash_t[h] = time;
  hash_f[h] = f;
  hash_g[h] = g;
  hash_l[h] = hinode;
  left[hinode] = f;
  right[hinode] = g;
  if (hinode ≤ curnode) {
    fprintf(stderr, "Out␣of␣memory!\n");
    exit(−1);
  }
  return hinode−−;
}
```

See also section 17.

This code is used in section 11.

17. The second phase of *intersect* uses a routine *new_node* that is very much like *new_template*. The main difference is that *new_node* creates (or finds existing copies) of node pairs in the *lower* memory.

⟨ Basic subroutines needed by *intersect* 16 ⟩ +≡

```
int new_node(f, g)
    register int f, g;
{
  register int h;
  h = (hash_rand * f + g) & (max_nodes − 1);   /* hash function */
  while (1) {
    if (hash_t[h] ≠ time) break;
    if (hash_f[h] ≡ f ∧ hash_g[h] ≡ g) return hash_l[h];
    h = (h − 1) & (max_nodes − 1);
  }
  hash_t[h] = time;
  hash_f[h] = f;
  hash_g[h] = g;
  hash_l[h] = curnode;
  left[curnode] = f;
  right[curnode] = g;
  if (hinode ≤ curnode) {
    fprintf(stderr, "Out␣of␣memory!\n");
    exit(−2);
  }
  return curnode++;
}
```

18. OK, we're ready to finish off the intersection process. The idea is to go through the template from the bottom up, collapsing identical nodes when they don't belong in an OBDD.

After we've visited a template node, we store a pointer to its low-memory clone in the *right* array. Neither the *left* nor *right* fields of that node will ever be needed again as inter-template pointers.

One subtle point needs to be mentioned (although it doesn't arise in the application to knight's tours, so I haven't really tested it): The resulting function $f \wedge g$ is identically zero if and only there is no template node for the dummy variable *edges*. Such a template node would arise from the sink node '1', if it were present.

#define *clone* *right*

⟨ Construct the reduced OBDD in lower memory, using the template 18 ⟩ ≡

```
curnode = 2;
if (head[edges] ≡ 0) return 0;   /* special case, see above */
clone[head[edges]] = 1;   /* 1 ∧ 1 = 1 */
for (k = edges − 1; k ≥ 0; k−−) {
  time++;   /* clear the hash table when a new level begins */
  for (j = head[k]; j; j = var[j]) {
    if (clone[left[j]] ≡ clone[right[j]]) clone[j] = clone[left[j]];
    else {
      clone[j] = new_node(clone[left[j]], clone[right[j]]);
      var[clone[j]] = k;
    }
  }
}
```

This code is used in section 11.

19. The *intersect* routine is now complete. I just want to point out here that *intersect*(f, g) is called in this program only when g is an OBDD of width 2; therefore the template (and the resulting OBDD) will never be more than twice the size of the original f. I haven't used that fact in the program, but it does tell us that *max_nodes* will be large enough if it is more than about three times the size of the OBDDs generated.

20. Counting the matchings. One of the neatest properties of the OBDD is that it's easy to count exactly how many combinations $(x_1, x_2, \ldots, x_n)$ will make $f(x_1, x_2, \ldots, x_n) = 1$. This is just the number of paths to node 1 in the dag.

To compute this number, I'll add a *count* array to the existing OBDD arrays. This one doesn't have to be as long as the others, since the final OBDD is in the lower part of the memory.

```
#define max_final_nodes (max_nodes/2)

⟨ Global variables 3 ⟩ +≡
  int count[max_final_nodes];
```

21.
```
⟨ Count and report the number of such matchings 21 ⟩ ≡
  if (f ≥ max_final_nodes) {
    printf(stderr, "Oops,␣out␣of␣memory␣for␣counting!\n");
    exit(−3);
  }
  count[0] = 0;
  count[1] = 1;
  for (k = 2; k ≤ f; k++) count[k] = count[left[k]] + count[right[k]];
  printf("Total␣solutions␣%d␣in␣OBDD␣of␣size␣%d.\n", count[f], f + 1);
```

This code is used in section 1.

22. Hamiltonicity. The first two edges in our list are the two knight moves from the upper left corner of the board. Some of the matchings use the first edge, some use the second. We want to look at all pairs of matchings (μ, μ') where μ uses the first edge and μ' uses the second, such that $\mu \cup \mu'$ is a single cycle.

To do this, we run through each μ in an outer loop, by traversing the OBDD as if it were a binary tree with shared subtrees. (Which it is.) Then for each μ, we traverse the OBDD again, in an inner loop, to find each μ' that's compatible with μ. The inner traversal is interrupted whenever we detect a cycle before a complete μ' is generated; so we don't really have to investigate at all the μ'.

How many μ' will acquire k edges before a cycle is detected? Consider a random model in which we start with a fixed matching of n black points with n red points. If we now choose a black node and a red node at random, the probability is $1/n$ that they will already be matched. Otherwise, we get essentially the same setup but with n decreased by 1. The generating function for the number of steps before a loop occurs therefore satisfies $g_n(z) = z\bigl(1 + (n-1)g_{n-1}(z)\bigr)/n$. And the solution is simply $g_n(z) = (z + z^2 + \cdots + z^n)/n$, a uniform distribution. According to this model, we can expect to interrupt the calculation of μ' before half of its edges are generated, about half the time. Still, the model predicts that we get all the way to the end in $1/n$ of all cases. This is exactly right if we start with a complete bipartite graph: Such a graph has $n!$ matchings, and $n!\,(n-1)! = n!^2/n$ oriented Hamiltonian cycles. But for knight graphs, the model is evidently too pessimistic (and that's good news for us): On an 8×8 board, the reported ratio of pairs of matchings to Hamiltonian paths is roughly 10^5, so cutoffs come along much more often than in a uniform distribution.

23. When I got to the point of writing this part of the program, it became clear why Minato invented a variant of OBDDs called ZBDDs [*ACM/IEEE Design Automation Conf.* **30** (1993), 272–277]. In this variant, we have $\mathrm{rep}f(x_1, x_2, \ldots, x_n) = \mathrm{rep}f(x_2, \ldots, x_n)$ if $f(1, x_2, \ldots, x_n)$ is identically zero, rather than if $f(0, x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n)$. For certain functions, the ZBDD representation is larger than the OBDD, but only in cases where a node has two identical subtrees; such cases never arise in connection with matching, since all solutions to the matching problem have the same sum $x_1 + x_2 + \cdots + x_n$. Conversely, the OBDDs for matching have lots of nodes with right subtree equal to 0, and such nodes waste time and memory because they contribute nothing to the traversal process that lists matchings.

The reasoning sketched in the previous paragraph can be understood from the following more detailed argument. A recursive traversal process *traverse*(t) might look like this:

```
if (right[t]) {
   use edge var[t];
   if (matching needs to be extended) traverse(right[t]);
   else do the endgame for the current matching;
   unuse edge var[t];
}
if (left[t]) traverse(left[t]);
```

The procedure here goes first to the right subtree, then to the left, in order to use tail recursion when implemented with a homegrown stack; but that's not the main point. My main point is that if $right[t] = 0$, *traverse*(t) is absolutely equivalent to *traverse*$(left[t])$ except for running time, since such nodes have $left[t] \neq 0$. Therefore we might as well eliminate such nodes.

I don't have time today (tonight) to modify this program so that it builds a ZBDD directly. That would probably be fairly easy, but ... maybe next year. Today I'll simply optimize my tree by reducing it so that right links are always nonnull.

Notice that after this is done, $right[k] = 1$ if and only if $black[var[k]]$ is the final black vertex, i.e., if and only if a perfect matching has been completed.

```
#define reduced count    /* at this point we no longer need the count array */

⟨Remove null right branches 23⟩ ≡
  reduced[0] = 0;
  for (k = 2, j = 0; k ≤ f; k++) {
    left[k] = reduced[left[k]];
    if (right[k]) reduced[k] = k, right[k] = reduced[right[k]];
    else j++, reduced[k] = left[k];    /* this node will not be accessed */
  }
  printf("(I␣removed␣%d␣null␣right␣branches.)\n", j);
```

This code is used in section 24.

24. In this part of the program I'm implementing recursive traversal with my own stack instead of using C's built-in recursion. The main reason is that we save overhead because of tail recursion. Of course, that may not be a big deal, but in a program like this I feel more confident about its speed if I don't have implicit computations going on. And I have no qualms about **goto** statements when they arise in a structured manner like this.

Notice that this code represents the current matching in graph *gg*, with *v*'s partner stored in *v*→*opp*.

```
#define opp v.V
⟨ Traverse and count Hamiltonian cycles 24 ⟩ ≡
  ⟨ Remove null right branches 23 ⟩;
  outerptr = 0;
  total_parity = parity[0];
  tt = right[f];     /* the outer loop uses edge 0 */
  black[0]->opp = red[0];
  red[0]->opp = black[0];
traverse: k = var[tt];
  black[k]->opp = red[k];
  red[k]->opp = black[k];
  total_parity += parity[k];
  if (right[tt] > 1) {     /* not done yet */
    outerstack[outerptr++] = tt;
    tt = right[tt];
    goto traverse;
  }
  ⟨ Do the inner traversal 27 ⟩;     /* We've got μ, now look for μ' */
back: total_parity -= parity[var[tt]];
  if (left[tt]) {
    tt = left[tt];
    goto traverse;
  }
  if (outerptr) {
    tt = outerstack[--outerptr];
    goto back;
  }
```

This code is used in section 1.

25. ⟨ Local variables 4 ⟩ +≡

```
  int tt;     /* node being traversed in the outer loop */
```

26. ⟨ Global variables 3 ⟩ +≡

```
  int outerstack[mm * nn * 2], innerstack[mm * nn * 2];     /* stacks for traversal */
  int outerptr, innerptr;     /* stack pointers */
  int total_parity;     /* sum of edge parities in the current matchings */
```

27. The inner traversal is very similar, except that we generalize the meaning of *opp*. Now, if there's a chain of links with u at one end and v at the other, u and v are considered "opposites." Inside the chain, the *opp* pointers contain information needed to restore the original matching when the chain is undone again later. This data structure gives immediate loop detection and requires only very simple updating.

```
⟨ Do the inner traversal 27 ⟩ ≡
  t = right[left[f]];    /* the inner loop uses edge 1 */
  u = black[1]→opp;
  v = red[1]→opp;
  u→opp = v;
  v→opp = u;
in_traverse: k = var[t];
  u = black[k]→opp;
  if (u ≡ red[k] ∧ right[t] > 1) goto bypass;    /* non-Hamiltonian cycle */
  u = black[k]→opp;
  v = red[k]→opp;
  u→opp = v;
  v→opp = u;
  total_parity += parity[k];
  if (right[t] > 1) {    /* not done yet */
    innerstack[innerptr++] = t;
    t = right[t];
    goto in_traverse;
  }
  ⟨ Record a solution 28 ⟩;    /* We've got μ ∪ μ′, a single cycle */
in_back: k = var[t];
  total_parity −= parity[k];
  u = black[k]→opp;
  v = red[k]→opp;
  u→opp = black[k];
  v→opp = red[k];
bypass:
  if (left[t]) {
    t = left[t];
    goto in_traverse;
  }
  if (innerptr) {
    t = innerstack[−−innerptr];
    goto in_back;
  }
```

This code is used in section 24.

28. ⟨ Record a solution 28 ⟩ ≡

```
if ((total_parity & 1) ≡ 0) pseudo_sols++;
else {
  sols++;
  if (sols % interval ≡ 0) {
    printf("%d:", sols);
    for (k = 0; k < outerptr; k++) printf("␣%s-%s%s", black[var[outerstack[k]]]→name,
          red[var[outerstack[k]]]→name, parity[var[outerstack[k]]] ? "*" : "");
    for (k = 0; k < innerptr; k++) printf("␣%s-%s%s", black[var[innerstack[k]]]→name,
          red[var[innerstack[k]]]→name, parity[var[innerstack[k]]] ? "*" : "");
    printf("\n");
  }
}
```

This code is used in section 27.

29. Experiences. When I ran this program in May, 1996, I was able to confirm the results I had obtained previously with my backtrack code for Hamiltonian paths. The running time for the 8×8 case (on my "old" SPARC2) was 1310 seconds. For 6×8 the OBDD had 6708 nodes, of which 4585 were removed by zero-suppression; for 8×6 the corresponding numbers were 7298 and 5156 (and I had to double the memory space). There were 2669 matchings (an odd number, so there are more with one of the corner moves than with the other). In the 8×8 case there were 106256 matchings and 112740 nodes in the OBDD, of which 80572 were removed.

30. Index.

OBDD