July 8, 2025 at 11:43

1*. Intro. This program reads output from an o command in BDD14 into internal arrays, by brute force. Here I generate a random solution.

The output from an o command in BDD15 can be read in an identical fashion, so zddread.w is identical to bddread.w. However, the interpretations are different and the correct program must be applied to each output.

Note that if a variable is not present in the input to bddread then it is ignored. If there is a variable that is not present in the BDD (because it is allowed to be true or false in all solutions), then you need to add this possibility yourself. For example, you must multiply the BDD solution count by a factor of 2 for every such variable. This is typically not a problem in ZDDs, because a variable not present in a ZDD is forced to be false.

```
#define memsize 40000000   /* this many nodes */
#define varsize 8192   /* this many variables */
#define bdds 1   /* this many BDDs */
#define bufsize 100   /* buffer size; 100 is plenty big */
#include <stdio.h>
#include <stdlib.h>
#include "gb_flip.h"
  typedef struct {
    int v;
    int lo;
    int hi;
    int mark;
  } node;
  int present[varsize];
  node *mem[bdds];
  long long count[3][memsize];
  int sols, seed;   /* command-line parameters */
  int root[bdds];
  FILE *infile;
  char buf[bufsize];
  unsigned int i1, i2, i3, i4;
  int memmax;
  ⟨Subroutines 2*⟩
  int main(int argc, char *argv[])
  {
    register int j, k, r, minv;
    if (argc ≠ 4 ∨ sscanf(argv[2], "%d", &sols) ≠ 1 ∨ sscanf(argv[3], "%d", &seed) ≠ 1) {
      fprintf(stderr, "Usage:␣%s␣foo.zdd␣sols␣seed\n", argv[0]);
      exit(−1);
    }
    gb_init_rand(seed);
    for (r = 0; r < bdds; r++) {
      mem[r] = (node *) malloc(memsize * sizeof(node));
      if (¬mem[r]) {
        printf("Sorry,␣I␣can't␣allocate␣mem[%d]!\n", r);
        exit(−99);
      }
      for (k = 0; k < memsize; k++) mem[r][k].lo = mem[r][k].hi = 0;
      if (¬(infile = fopen(argv[r + 1], "r"))) {
```

```
        printf("Sorry,␣I␣can't␣open␣`%s'␣for␣reading!\n", argv[r + 1]);
        exit(−1);
      }
      for (k = 0, minv = varsize; ; ) {
        if (¬fgets(buf, bufsize, infile)) break;
        j = sscanf(buf, "%x:␣(~%u?%x:%x)\n", &i1, &i2, &i3, &i4);
        if (j ≠ 4) printf("!␣I␣got␣only␣%d␣inputs␣from␣the␣line␣%s", j, buf);
        else {
          if (i1 > memmax) memmax = i1;
          if (i3 > memmax) memmax = i3;
          if (i4 > memmax) memmax = i4;
          if (i1 ≥ memsize ∨ i2 ≥ varsize ∨ i3 ≥ memsize ∨ i4 ≥ memsize) {
            printf("!␣address␣out␣of␣range␣in␣the␣line␣%s", buf);
            exit(−69);
          } else if (mem[r][i1].lo ∨ mem[r][i1].hi) printf("!␣clobbered␣node␣in␣the␣line␣%s", buf);
          else {
            if (i2 < minv) minv = i2, root[r] = i1;
            k++, mem[r][i1].v = i2, mem[r][i1].lo = i3, mem[r][i1].hi = i4;
            present[i2] = 1;
          }
        }
      }
      fprintf(stderr, "%d␣nodes␣input␣into␣mem%d\n", k, r);
      fprintf(stderr, "(memmax=%d)\n", memmax);
    }
    for (j = k = 0; j < varsize; j++)
      if (present[j]) k++;
    fprintf(stderr, "There␣are␣%d␣variables.\n", k);
    ⟨Do our thing 4*⟩;
  }
```

2* First, a recursive subroutine.

```
#define two_to_the_62 #4000000000000000

⟨ Subroutines 2* ⟩ ≡
  void countsols(int p)
  {
    register int q;
    long long c0 = 0, c1 = 0, c2 = 0;

    q = mem[0][p].lo;
    if (q) {
      if (mem[0][q].mark ≡ 0) countsols(q);
      c0 = count[0][q], c1 = count[1][q], c2 = count[2][q];
    }
    q = mem[0][p].hi;
    if (q) {
      if (mem[0][q].mark ≡ 0) countsols(q);
      c0 += count[0][q], c1 += count[1][q], c2 += count[2][q];
      if (c2 ≥ two_to_the_62) c1++, c2 −= two_to_the_62;
      if (c1 ≥ two_to_the_62) c0++, c1 −= two_to_the_62;
      if (c0 ≥ two_to_the_62) {
        fprintf(stderr, "Overflow␣(186␣bits␣or␣more)!\n");
        printf("Overflow␣(186␣bits␣or␣more)!\n");
        exit(−6);
      }
    }
    mem[0][p].mark = 1;
    count[0][p] = c0, count[1][p] = c1, count[2][p] = c2;
  }
```

See also section 3*.

This code is used in section 1*.

3* I also need an extralong version of *gb_unif_rand*.

⟨ Subroutines 2* ⟩ +≡

```
long long long_unif_rand(long long m)
{
  register long long t = two_to_the_62 − (two_to_the_62 % m);
  register long long r;
  do {
    r = (((long long) gb_next_rand( )) ≪ 31) + gb_next_rand( );
  } while (t ≤ r);
  return r % m;
}
long long in0, in1, in2, out0, out1, out2;
void triply_longlong_unif_rand(void)
{
  register long long t;
  if (¬in0) {
    out0 = 0;
    if (¬in1) out1 = 0, out2 = long_unif_rand(in2);
    else {
      for (t = 1; t ≤ in1; t ≪= 1) ;
      while (1) {
        out1 = (((long long) gb_next_rand( )) ≪ 31) + gb_next_rand( );
        out1 &= t − 1;
        if (out1 ≤ in1) {
          out2 = (((long long) gb_next_rand( )) ≪ 31) + gb_next_rand( );
          if (out1 < in1 ∨ out2 < in2) break;
        }
      }
    }
  }
  else {
    for (t = 1; t ≤ in0; t ≪= 1) ;
    while (1) {
      out0 = (((long long) gb_next_rand( )) ≪ 31) + gb_next_rand( );
      out0 &= t − 1;
      if (out0 ≤ in0) {
        out1 = (((long long) gb_next_rand( )) ≪ 31) + gb_next_rand( );
        out2 = (((long long) gb_next_rand( )) ≪ 31) + gb_next_rand( );
        if (out0 < in0 ∨ out1 < in1 ∨ (out1 ≡ in1 ∧ out2 < in2)) break;
      }
    }
  }
}
```

4* ⟨ Do our thing 4* ⟩ ≡

```
count[2][1] = 1, mem[0][1].mark = 1, mem[0][1].v = varsize;
countsols(root[0]);
for (r = 0; r < sols; r++) {
  for (k = root[0]; k > 1; ) {
    j = mem[0][k].lo;
    in0 = count[0][k], in1 = count[1][k], in2 = count[2][k];
    triply_longlong_unif_rand( );
    if (count[0][j] < out0 ∨ (count[0][j] ≡ out0 ∧ count[1][j] < out1) ∨ (count[0][j] ≡ out0 ∧ count[1][j] ≡
          out1 ∧ count[2][j] ≤ out2)) {
      printf("␣%d", mem[0][k].v);
      k = mem[0][k].hi;
    } else k = j;
  }
  printf("\n");
}
```

This code is used in section 1*.

5* Index.

The following sections were changed by the change file: 1, 2, 3, 4, 5.

ZDDREAD-RANDOM