July 22, 2021 at 04:22

1. Intro. This program is part of a series of "SAT-solvers" that I'm putting together for my own education as I prepare to write Section 7.2.2.2 of *The Art of Computer Programming*. My intent is to have a variety of compatible programs on which I can run experiments to learn how different approaches work in practice.

This time I'm implementing the algorithm that physicists have christened "Survey Propagation." It's a development of a message-passing idea called "Belief Propagation," which in turn extends "Warning Propagation." [See Braunstein, Mézard, and Zecchina, *Random Structures &Algorithms* **27** (2005), 201–226.] And I'm also implementing an extended, improved algorithm that incorporates "reinforcement" [see Chavas, Furtlehner, Mézard, and Zecchina, *Journal of Statistical Mechanics* (November 2005), P11016, 25 pages]. While writing this code I was greatly helped by studying an implementation prepared by Carlo Baldassi in March 2012.

2. If you have already read SAT8, or any other program of this series, you might as well skip now past the rest of this introduction, and past the code for the "I/O wrapper" that is presented in the next dozen or so sections, because you've seen it before. (Except that there are several new command-line options, and the output is a reduced set of clauses rather than a solution.)

The input appears on *stdin* as a series of lines, with one clause per line. Each clause is a sequence of literals separated by spaces. Each literal is a sequence of one to eight ASCII characters between `!` and `}`, inclusive, not beginning with `~`, optionally preceded by `~` (which makes the literal "negative"). For example, Rivest's famous clauses on four variables, found in 6.5–(13) and 7.1.1–(32) of *TAOCP*, can be represented by the following eight lines of input:

```
x2 x3 ~x4
x1 x3 x4
~x1 x2 x4
~x1 ~x2 x3
~x2 ~x3 x4
~x1 ~x3 ~x4
x1 ~x2 ~x4
x1 x2 ~x3
```

Input lines that begin with `~␣` are ignored (treated as comments). The output will be '`~~?`' if the algorithm could not find a way to satisfy the input clauses. Otherwise it will be a *partial* solution: a list of noncontradictory literals that cover some but maybe not all of the clauses, separated by spaces. ("Noncontradictory" means that we don't have both a literal and its negation.) The residual problem, which must be satisfied if the partial assignment turns out to be valid, is written to an auxiliary file. (The partial assignment might be faulty; the algorithm has pretty good heuristics, but there are no guarantees.)

The input above would, for example, probably yield '`~~?`'. But if the final clause were omitted, the output might be '`~x1 ~x2`', leaving a residual problem with the two clauses '`x3 ~x4`' and '`x3 x4`'. Or it might be '`~x3`', leaving the (unsatisfiable) residual problem '`x2 ~x4`', '`x1 x4`', '`~x1 x2, x4`', '`~x1 ~x2`', '`x1 ~x2 ~x4`'.

The running time in "mems" is also reported, together with the approximate number of bytes needed for data storage. One "mem" essentially means a memory access to a 64-bit word. (These totals don't include the time or space needed to parse the input or to format the output.)

3. So here's the structure of the program. (Skip ahead if you are impatient to see the interesting stuff.)

```
#define o mems++     /* count one mem */
#define oo mems += 2     /* count two mems */
#define ooo mems += 3     /* count three mems */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_flip.h"
  typedef unsigned int uint;     /* a convenient abbreviation */
  typedef unsigned long long ullng;     /* ditto */
  ⟨Type definitions 6⟩;
  ⟨Global variables 4⟩;
  ⟨Subroutines 26⟩;
  main(int argc, char *argv[])
  {
    register uint c, g, h, i, j, k, l, p, q, r, ii, kk, ll, fcount;
    ⟨Process the command line 5⟩;
    ⟨Initialize everything 9⟩;
    ⟨Input the clauses 10⟩;
    if (verbose & show_basics) ⟨Report the successful completion of the input phase 22⟩;
    ⟨Set up the main data structures 28⟩;
    imems = mems, mems = 0;
    ⟨Solve the problem 35⟩;
    if (verbose & show_basics)
      fprintf(stderr, "Altogether␣%llu+%llu␣mems,␣%llu␣bytes.\n", imems, mems, bytes);
  }
```

4.

```
#define show_basics 1     /* verbose code for basic stats */
#define show_choices 2     /* verbose code for backtrack logging */
#define show_details 4     /* verbose code for further commentary */
#define show_gory_details 8     /* verbose code turned on when debugging */
#define show_histogram 16     /* verbose code to make a π × π histogram */
#define show_pis 32     /* verbose code to print out all the π's */
⟨Global variables 4⟩ ≡
  int random_seed = 0;     /* seed for the random words of gb_rand */
  int verbose = show_basics;     /* level of verbosity */
  int hbits = 8;     /* logarithm of the number of the hash lists */
  int buf_size = 1024;     /* must exceed the length of the longest input line */
  int max_iter = 1000;     /* maximum iterations */
  int min_iter = 5;     /* minimum iterations before reinforcement kicks in */
  int confidence = 50;     /* lower limit for confidence of setting a variable */
  double damper = 0.99;     /* the damping factor for reinforcement */
  double threshold = 0.01;     /* upper limit for convergence check */
  ullng imems, mems;     /* mem counts */
  ullng thresh = 0;     /* report when mems exceeds this, if delta ≠ 0 */
  ullng delta = 0;     /* report every delta or so mems */
  ullng bytes;     /* memory used by main data structures */
```

See also sections 8, 25, 36, and 54.

This code is used in section 3.

5. On the command line one can specify any or all of the following options:

- ‘v⟨ integer ⟩’ to enable various levels of verbose output on *stderr*.
- ‘h⟨ positive integer ⟩’ to adjust the hash table size.
- ‘b⟨ positive integer ⟩’ to adjust the size of the input buffer.
- ‘s⟨ integer ⟩’ to define the seed for any random numbers that are used.
- ‘d⟨ integer ⟩’ to set *delta* for periodic state reports.
- ‘t⟨ integer ⟩’ to define the maximum number of iterations.
- ‘l⟨ integer ⟩’ to define the minimum number of iterations before reinforcement begins.
- ‘c⟨ integer ⟩’ to define the *confidence* percentage, above which we decide that a variable is sufficiently biased to be assigned a value.
- ‘p⟨ float ⟩’ to define the damping factor *damper* for reinforcement.
- ‘e⟨ float ⟩’ to define the *threshold* by which we decide that the messages have converged.

The defaults are listed with ‘Global variables’ above.

```
⟨ Process the command line 5 ⟩ ≡
  for (j = argc − 1, k = 0; j; j−−)
    switch (argv[j][0]) {
    case 'v': k |= (sscanf(argv[j] + 1, "%d", &verbose) − 1); break;
    case 'h': k |= (sscanf(argv[j] + 1, "%d", &hbits) − 1); break;
    case 'b': k |= (sscanf(argv[j] + 1, "%d", &buf_size) − 1); break;
    case 's': k |= (sscanf(argv[j] + 1, "%d", &random_seed) − 1); break;
    case 'd': k |= (sscanf(argv[j] + 1, "%lld", &delta) − 1); thresh = delta; break;
    case 't': k |= (sscanf(argv[j] + 1, "%d", &max_iter) − 1); break;
    case 'l': k |= (sscanf(argv[j] + 1, "%d", &min_iter) − 1); break;
    case 'c': k |= (sscanf(argv[j] + 1, "%d", &confidence) − 1); break;
    case 'p': k |= (sscanf(argv[j] + 1, "%lf", &damper) − 1); break;
    case 'e': k |= (sscanf(argv[j] + 1, "%lf", &threshold) − 1); break;
    default: k = 1;     /* unrecognized command-line option */
    }
  if (k ∨ hbits < 0 ∨ hbits > 30 ∨ buf_size ≤ 0) {
    fprintf(stderr,
        "Usage:␣%s␣[v<n>]␣[h<n>]␣[b<n>]␣[s<n>]␣[d<n>]␣[t<n>]␣[l<n>]␣[c<n>]␣[p<f>]␣[e<f>]\n",
        argv[0]);
    exit(−1);
  }
  if (damper < 0.0 ∨ damper > 1.0) {
    fprintf(stderr, "Parameter␣p␣should␣be␣between␣0.0␣and␣1.0!\n");
    exit(−666);
  }
  if (confidence < 0 ∨ confidence > 100) {
    fprintf(stderr, "Parameter␣c␣should␣be␣between␣0␣and␣100!\n");
    exit(−667);
  }
```

This code is used in section 3.

6. The I/O wrapper. The following routines read the input and absorb it into temporary data areas from which all of the "real" data structures can readily be initialized. My intent is to incorporate these routines in all of the SAT-solvers in this series. Therefore I've tried to make the code short and simple, yet versatile enough so that almost no restrictions are placed on the sizes of problems that can be handled. These routines are supposed to work properly unless there are more than $2^{32}-1 = 4{,}294{,}967{,}295$ occurrences of literals in clauses, or more than $2^{31}-1 = 2{,}147{,}483{,}647$ variables or clauses.

In these temporary tables, each variable is represented by four things: its unique name; its serial number; the clause number (if any) in which it has most recently appeared; and a pointer to the previous variable (if any) with the same hash address. Several variables at a time are represented sequentially in small chunks of memory called "vchunks," which are allocated as needed (and freed later).

```
#define vars_per_vchunk 341   /* preferably (2^k − 1)/3 for some k */
⟨Type definitions 6⟩ ≡
  typedef union {
    char ch8[8];
    uint u2[2];
    long long lng;
  } octa;
  typedef struct tmp_var_struct {
    octa name;   /* the name (one to seven ASCII characters) */
    uint serial;   /* 0 for the first variable, 1 for the second, etc. */
    int stamp;   /* m if positively in clause m; −m if negatively there */
    struct tmp_var_struct *next;   /* pointer for hash list */
  } tmp_var;
  typedef struct vchunk_struct {
    struct vchunk_struct *prev;   /* previous chunk allocated (if any) */
    tmp_var var[vars_per_vchunk];
  } vchunk;
```

See also sections 7 and 24.

This code is used in section 3.

7. Each clause in the temporary tables is represented by a sequence of one or more pointers to the **tmp_var** nodes of the literals involved. A negated literal is indicated by adding 1 to such a pointer. The first literal of a clause is indicated by adding 2. Several of these pointers are represented sequentially in chunks of memory, which are allocated as needed and freed later.

```
#define cells_per_chunk 511   /* preferably 2^k − 1 for some k */
⟨Type definitions 6⟩ +≡
  typedef struct chunk_struct {
    struct chunk_struct *prev;   /* previous chunk allocated (if any) */
    tmp_var *cell[cells_per_chunk];
  } chunk;
```

8. ⟨ Global variables 4 ⟩ +≡

```
char *buf;    /* buffer for reading the lines (clauses) of stdin */
tmp_var **hash;    /* heads of the hash lists */
uint hash_bits[93][8];    /* random bits for universal hash function */
vchunk *cur_vchunk;    /* the vchunk currently being filled */
tmp_var *cur_tmp_var;    /* current place to create new tmp_var entries */
tmp_var *bad_tmp_var;    /* the cur_tmp_var when we need a new vchunk */
chunk *cur_chunk;    /* the chunk currently being filled */
tmp_var **cur_cell;    /* current place to create new elements of a clause */
tmp_var **bad_cell;    /* the cur_cell when we need a new chunk */
ullng vars;    /* how many distinct variables have we seen? */
ullng clauses;    /* how many clauses have we seen? */
ullng nullclauses;    /* how many of them were null? */
ullng cells;    /* how many occurrences of literals in clauses? */
```

9. ⟨ Initialize everything 9 ⟩ ≡

```
gb_init_rand(random_seed);
buf = (char *) malloc(buf_size * sizeof(char));
if (¬buf) {
  fprintf(stderr, "Couldn't␣allocate␣the␣input␣buffer␣(buf_size=%d)!\n", buf_size);
  exit(−2);
}
hash = (tmp_var **) malloc(sizeof(tmp_var) ≪ hbits);
if (¬hash) {
  fprintf(stderr, "Couldn't␣allocate␣%d␣hash␣list␣heads␣(hbits=%d)!\n", 1 ≪ hbits, hbits);
  exit(−3);
}
for (h = 0; h < 1 ≪ hbits; h++) hash[h] = Λ;
```

See also section 15.

This code is used in section 3.

10. The hash address of each variable name has h bits, where h is the value of the adjustable parameter *hbits*. Thus the average number of variables per hash list is $n/2^h$ when there are n different variables. A warning is printed if this average number exceeds 10. (For example, if h has its default value, 8, the program will suggest that you might want to increase h if your input has 2560 different variables or more.)

All the hashing takes place at the very beginning, and the hash tables are actually recycled before any SAT-solving takes place; therefore the setting of this parameter is by no means crucial. But I didn't want to bother with fancy coding that would determine h automatically.

⟨Input the clauses 10⟩ ≡

```
while (1) {
  if (¬fgets(buf, buf_size, stdin)) break;
  clauses++;
  if (buf[strlen(buf) − 1] ≠ '\n') {
    fprintf(stderr, "The␣clause␣on␣line␣%d␣(%.20s...)␣is␣too␣long␣for␣me;\n", clauses, buf);
    fprintf(stderr, "␣my␣buf_size␣is␣only␣%d!\n", buf_size);
    fprintf(stderr, "Please␣use␣the␣command-line␣option␣b<newsize>.\n");
    exit(−4);
  }
  ⟨Input the clause in buf 11⟩;
}
if ((vars ≫ hbits) ≥ 10) {
  fprintf(stderr, "There␣are␣%d␣variables␣but␣only␣%d␣hash␣tables;\n", vars, 1 ≪ hbits);
  while ((vars ≫ hbits) ≥ 10) hbits++;
  fprintf(stderr, "␣maybe␣you␣should␣use␣command-line␣option␣h%d?\n", hbits);
}
clauses −= nullclauses;
if (clauses ≡ 0) {
  fprintf(stderr, "No␣clauses␣were␣input!\n");
  exit(−77);
}
if (vars ≥ #80000000) {
  fprintf(stderr, "Whoa,␣the␣input␣had␣%llu␣variables!\n", vars);
  exit(−664);
}
if (clauses ≥ #80000000) {
  fprintf(stderr, "Whoa,␣the␣input␣had␣%llu␣clauses!\n", clauses);
  exit(−665);
}
if (cells ≥ #100000000) {
  fprintf(stderr, "Whoa,␣the␣input␣had␣%llu␣occurrences␣of␣literals!\n", cells);
  exit(−666);
}
```

This code is used in section 3.

11. ⟨Input the clause in *buf* 11⟩ ≡

```
for (j = k = 0; ; ) {
  while (buf[j] ≡ '␣') j++;    /* scan to nonblank */
  if (buf[j] ≡ '\n') break;
  if (buf[j] < '␣' ∨ buf[j] > '~') {
    fprintf(stderr, "Illegal␣character␣(code␣#%x)␣in␣the␣clause␣on␣line␣%d!\n", buf[j], clauses);
    exit(−5);
  }
  if (buf[j] ≡ '~') i = 1, j++;
  else i = 0;
  ⟨Scan and record a variable; negate it if i ≡ 1 12⟩;
}
if (k ≡ 0) {
  fprintf(stderr, "(Empty␣line␣%d␣is␣being␣ignored)\n", clauses);
  nullclauses++;    /* strictly speaking it would be unsatisfiable */
}
goto clause_done;
empty_clause: ⟨Remove all variables of the current clause 19⟩;
clause_done: cells += k;
```

This code is used in section 10.

12. We need a hack to insert the bit codes 1 and/or 2 into a pointer value.

```
#define hack_in(q, t) (tmp_var *)(t | (ullng) q)
```

⟨Scan and record a variable; negate it if $i \equiv 1$ 12⟩ ≡

```
{
  register tmp_var *p;
  if (cur_tmp_var ≡ bad_tmp_var) ⟨Install a new vchunk 13⟩;
  ⟨Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 16⟩;
  ⟨Find cur_tmp_var→name in the hash table at p 17⟩;
  if (p→stamp ≡ clauses ∨ p→stamp ≡ −clauses) ⟨Handle a duplicate literal 18⟩
  else {
    p→stamp = (i ? −clauses : clauses);
    if (cur_cell ≡ bad_cell) ⟨Install a new chunk 14⟩;
    *cur_cell = p;
    if (i ≡ 1) *cur_cell = hack_in(*cur_cell, 1);
    if (k ≡ 0) *cur_cell = hack_in(*cur_cell, 2);
    cur_cell++, k++;
  }
}
```

This code is used in section 11.

13. ⟨ Install a new **vchunk** 13 ⟩ ≡

```
{
  register vchunk *new_vchunk;
  new_vchunk = (vchunk *) malloc(sizeof(vchunk));
  if (¬new_vchunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣vchunk!\n");
    exit(−6);
  }
  new_vchunk→prev = cur_vchunk, cur_vchunk = new_vchunk;
  cur_tmp_var = &new_vchunk→var[0];
  bad_tmp_var = &new_vchunk→var[vars_per_vchunk];
}
```

This code is used in section 12.

14. ⟨ Install a new **chunk** 14 ⟩ ≡

```
{
  register chunk *new_chunk;
  new_chunk = (chunk *) malloc(sizeof(chunk));
  if (¬new_chunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣chunk!\n");
    exit(−7);
  }
  new_chunk→prev = cur_chunk, cur_chunk = new_chunk;
  cur_cell = &new_chunk→cell[0];
  bad_cell = &new_chunk→cell[cells_per_chunk];
}
```

This code is used in section 12.

15. The hash code is computed via "universal hashing," using the following precomputed tables of random bits.

⟨ Initialize everything 9 ⟩ +≡

```
for (j = 92; j; j−−)
  for (k = 0; k < 8; k++) hash_bits[j][k] = gb_next_rand();
```

16. ⟨ Put the variable name beginning at *buf*[*j*] in *cur_tmp_var→name* and compute its hash code *h* 16 ⟩ ≡

```
cur_tmp_var→name.lng = 0;
for (h = l = 0; buf[j + l] > '␣' ∧ buf[j + l] ≤ '~'; l++) {
  if (l > 7) {
    fprintf(stderr, "Variable␣name␣%.9s...␣in␣the␣clause␣on␣line␣%d␣is␣too␣long!\n", buf + j,
        clauses);
    exit(−8);
  }
  h ⊕= hash_bits[buf[j + l] − '!'][l];
  cur_tmp_var→name.ch8[l] = buf[j + l];
}
if (l ≡ 0) goto empty_clause;     /* '~' by itself is like 'true' */
j += l;
h &= (1 ≪ hbits) − 1;
```

This code is used in section 12.

17. ⟨ Find *cur_tmp_var→name* in the hash table at *p* 17 ⟩ ≡

```
for (p = hash[h]; p; p = p→next)
  if (p→name.lng ≡ cur_tmp_var→name.lng) break;
if (¬p) {    /* new variable found */
  p = cur_tmp_var++;
  p→next = hash[h], hash[h] = p;
  p→serial = vars++;
  p→stamp = 0;
}
```

This code is used in section 12.

18. The most interesting aspect of the input phase is probably the "unwinding" that we might need to do when encountering a literal more than once in the same clause.

⟨ Handle a duplicate literal 18 ⟩ ≡

```
{
  if ((p→stamp > 0) ≡ (i > 0)) goto empty_clause;
}
```

This code is used in section 12.

19. An input line that begins with '~␣' is silently treated as a comment. Otherwise redundant clauses are logged, in case they were unintentional. (One can, however, intentionally use redundant clauses to force the order of the variables.)

⟨ Remove all variables of the current clause 19 ⟩ ≡

```
while (k) {
  ⟨ Move cur_cell backward to the previous cell 20 ⟩;
  k−−;
}
if ((buf[0] ≠ '~') ∨ (buf[1] ≠ '␣'))
  fprintf(stderr, "(The␣clause␣on␣line␣%d␣is␣always␣satisfied)\n", clauses);
nullclauses++;
```

This code is used in section 11.

20. ⟨ Move *cur_cell* backward to the previous cell 20 ⟩ ≡

```
if (cur_cell > &cur_chunk→cell[0]) cur_cell−−;
else {
  register chunk *old_chunk = cur_chunk;
  cur_chunk = old_chunk→prev; free(old_chunk);
  bad_cell = &cur_chunk→cell[cells_per_chunk];
  cur_cell = bad_cell − 1;
}
```

This code is used in sections 19 and 32.

21. ⟨ Move *cur_tmp_var* backward to the previous temporary variable 21 ⟩ ≡

```
if (cur_tmp_var > &cur_vchunk→var[0]) cur_tmp_var−−;
else {
  register vchunk *old_vchunk = cur_vchunk;
  cur_vchunk = old_vchunk→prev; free(old_vchunk);
  bad_tmp_var = &cur_vchunk→var[vars_per_vchunk];
  cur_tmp_var = bad_tmp_var − 1;
}
```

This code is used in section 33.

22. ⟨ Report the successful completion of the input phase 22 ⟩ ≡

```
fprintf(stderr, "(%d␣variables,␣%d␣clauses,␣%llu␣literals␣successfully␣read)\n", vars, clauses,
    cells);
```

This code is used in section 3.

23. SAT solving, version 9. Survey Propagation is slightly similar to WalkSAT, but it's really a new kettle of fish. Clauses pass messages to each of their literals, representing locally known information about the other literals in the clause. Literals pass messages to each of the clauses that they or their complement are in, representing locally known information about the other clauses to which they belong. When we find a variable with a strong tendency to be true or false, we fix its value and reduce to a smaller system. Local information continues to propagate until we get some sort of convergence.

The clause-to-literal messages are called η's. If c is a clause and l is a literal, $\eta_{c\to l}$ is a fraction between 0 and 1 that is *large* if c urgently needs l to be true, otherwise it's small.

The literal-to-clause messages are called π's. They too are fractions between 0 and 1, but they're sort of dual because they represent flexibility: The value of $\pi_{l\to c}$ is *small* when clauses other than c badly want l to be true.

An "external force field" that gently nudges literal l towards a particular value, with urgency η_l, is also present. This force-of-reinforcement tends to improve decision-making, because it encourages the algorithm to decide between competing tendencies.

Internally we maintain a single value π_l for each literal, namely $1-\eta_l$ times the product of $1-\eta_{c\to l}$ over all clauses c that contain l. The message $\pi_{l\to c}$ is then simply π_l when $l\notin c$; and it's $\pi_l/(1-\eta_{c\to l})$ when $l\in c$. We use a special data structure to count the factors of this product that happen to be zero (within floating-point precision), so that division by zero isn't a problem.

24. The data structures are analogous to those of previous programs in this series. There are three main arrays, *cmem*, *lmem*, and *mem*. Structured **clause** nodes appear in *cmem*, and structured **literal** nodes appear in *lmem*. Each clause points to a sequential list of literals and η's in *mem*; each literal points to a linked list of clause slots in *mem*, showing where that literal occurs in the problem. The literal nodes in *lmem* also hold η_l and π_l.

As in most previous programs of this series, the literals x and $\bar{x}$ are represented internally by $2k$ and $2k+1$ when x is variable number k.

The symbolic names of variables are kept separately in an array called *nmem*.

⟨ Type definitions 6 ⟩ +≡

```
typedef struct {
  double eta;    /* the external force on this literal */
  double pi;    /* this literal's current π value */
  uint zf;    /* the number of suppressed zero factors in pi */
  uint link;    /* first occurrence of the literal in mem, plus 1 */
  int rating;    /* +1 positive, −1 negative, 0 wishy-washy or wild */
} literal;    /* would it go faster if I added four more bytes of padding? */
typedef struct {
  uint start;    /* where the literal list starts in mem */
  uint size;    /* number of remaining literals in clause postprocessing phase */
} clause;
typedef struct {
  union { double d;
    ullng u;
  } eta;    /* η message for a literal */
  uint lit;    /* number of that literal */
  uint next;    /* where that literal next appears in mem, plus 1 */
} mem_item;
```

25. ⟨ Global variables 4 ⟩ +≡

```
clause *cmem;    /* the master array of clauses */
literal *lmem;    /* the master array of literals */
mem_item *mem;    /* the master array of literals in clauses */
mem_item *cur_mcell;    /* the current cell of interest in mem */
octa *nmem;    /* the master array of symbolic variable names */
double *gam;    /* temporary array to hold gamma ratios */
```

26. Here is a subroutine that prints a clause symbolically. It illustrates some of the conventions of the data structures that have been explained above. I use it only for debugging.

⟨ Subroutines 26 ⟩ ≡

```
void print_clause(uint c)
{    /* the first clause is called clause 1, not 0 */
  register uint l, ll;
  fprintf(stderr, "%d:\n", c);    /* show the clause number */
  for (l = cmem[c - 1].start; l < cmem[c].start; l++) {
    ll = mem[l].lit;
    fprintf(stderr, " %s%.8s(%d), eta=%.15g\n", ll & 1 ? "~" : "", nmem[ll >> 1].ch8, ll >> 1,
        mem[l].eta.d);
  }
}
```

See also sections 27 and 47.

This code is used in section 3.

27. Another simple subroutine shows the two π and η values for a given variable.

⟨ Subroutines 26 ⟩ +≡

```
void print_var(uint k)
{
  register uint l = k << 1;
  fprintf(stderr, "pi(%.8s)=%.15g(%d), eta(%.8s)=%.15g, ", nmem[k].ch8, lmem[l].pi, lmem[l].zf,
      nmem[k].ch8, lmem[l].eta);
  fprintf(stderr, "pi(~%.8s)=%.15g(%d), eta(~%.8s)=%.15g\n", nmem[k].ch8, lmem[l + 1].pi,
      lmem[l + 1].zf, nmem[k].ch8, lmem[l + 1].eta);
}
```

28. Initializing the real data structures. We're ready now to convert the temporary chunks of data into the form we want, and to recycle those chunks.

⟨ Set up the main data structures 28 ⟩ ≡
  ⟨ Allocate the main arrays 29 ⟩;
  ⟨ Zero the links 30 ⟩;
  ⟨ Copy all the temporary cells to the *mem* and *cmem* arrays in proper format 31 ⟩;
  ⟨ Copy all the temporary variable nodes to the *nmem* array in proper format 33 ⟩;
  ⟨ Check consistency 34 ⟩;

This code is used in section 3.

29. ⟨ Allocate the main arrays 29 ⟩ ≡

```
free(buf); free(hash);    /* a tiny gesture to make a little room */
lmem = (literal *) malloc((vars + vars + 1) * sizeof(literal));
if (¬lmem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣lmem␣array!\n");
  exit(−12);
}
bytes = (vars + vars + 1) * sizeof(literal);
nmem = (octa *) malloc(vars * sizeof(octa));
if (¬nmem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣nmem␣array!\n");
  exit(−13);
}
bytes += vars * sizeof(octa);
mem = (mem_item *) malloc(cells * sizeof(mem_item));
if (¬mem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣big␣mem␣array!\n");
  exit(−10);
}
bytes += cells * sizeof(mem_item);
cmem = (clause *) malloc((clauses + 1) * sizeof(clause));
if (¬cmem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣cmem␣array!\n");
  exit(−11);
}
bytes += (clauses + 1) * sizeof(clause);
```

This code is used in section 28.

30. ⟨ Zero the links 30 ⟩ ≡

```
for (l = vars + vars; l; l−−) o, lmem[l − 1].link = 0;
```

This code is used in section 28.

31. ⟨ Copy all the temporary cells to the *mem* and *cmem* arrays in proper format 31 ⟩ ≡

```
for (c = clauses, cur_mcell = mem + cells, kk = 0; c; c−−) {
  o, cmem[c].start = cur_mcell − mem;
  k = 0;
  ⟨ Insert the cells for the literals of clause c 32 ⟩;
  if (k > kk) kk = k;      /* maximum clause size seen so far */
}
if (cur_mcell ≠ mem) {
  fprintf(stderr, "Confusion␣about␣the␣number␣of␣cells!\n");
  exit(−99);
}
o, cmem[0].start = 0;
gam = (double *) malloc(kk * sizeof(double));
if (¬gam) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣gamma␣array!\n");
  exit(−16);
}
bytes += kk * sizeof(double);
```

This code is used in section 28.

32. The basic idea is to "unwind" the steps that we went through while building up the chunks.

```
#define hack_out(q) (((ullng) q) & #3)
#define hack_clean(q) ((tmp_var *)((ullng) q & −4))
```

⟨ Insert the cells for the literals of clause *c* 32 ⟩ ≡

```
for (i = 0; i < 2; k++) {
  ⟨ Move cur_cell backward to the previous cell 20 ⟩;
  i = hack_out(*cur_cell);
  p = hack_clean(*cur_cell)→serial;
  cur_mcell−−;
  o, cur_mcell→lit = l = p + p + (i & 1);
  oo, cur_mcell→next = lmem[l].link;
  o, lmem[l].link = cur_mcell − mem + 1;
}
```

This code is used in section 31.

33. ⟨ Copy all the temporary variable nodes to the *nmem* array in proper format 33 ⟩ ≡

```
for (c = vars; c; c−−) {
  ⟨ Move cur_tmp_var backward to the previous temporary variable 21 ⟩;
  o, nmem[c − 1].lng = cur_tmp_var→name.lng;
}
```

This code is used in section 28.

34. We should now have unwound all the temporary data chunks back to their beginnings.

⟨ Check consistency 34 ⟩ ≡

```
if (cur_cell ≠ &cur_chunk→cell[0] ∨ cur_chunk→prev ≠ Λ ∨ cur_tmp_var ≠
      &cur_vchunk→var[0] ∨ cur_vchunk→prev ≠ Λ) {
  fprintf(stderr, "This␣can't␣happen␣(consistency␣check␣failure)!\n");
  exit(−14);
}
free(cur_chunk); free(cur_vchunk);
```

This code is used in section 28.

35. Doing it. So we take surveys.

```
⟨ Solve the problem 35 ⟩ ≡
  factor = 1.0;
  ⟨ Initialize all η's to random fractions 37 ⟩;
  for (iter = 0; iter < max_iter; iter++) {
    if ((iter & 1) ∧ iter ≥ min_iter) {
      ⟨ Adjust the reinforcement fields 39 ⟩;
      ⟨ Exit if the clauses are pseudo-satisfied 40 ⟩;
    }
    if (verbose & show_choices) fprintf(stderr, "beginning␣iteration␣%d\n", iter + 1);
    ⟨ Compute the π's 38 ⟩;
    ⟨ Update the η's 41 ⟩;
    if (verbose & show_details) fprintf(stderr, "␣(max␣diff␣%.15g,␣%lld␣mems)\n", max_diff, mems);
    if (delta ∧ (mems ≥ thresh)) {
      thresh += delta;
      fprintf(stderr, "␣after␣%lld␣mems,␣iteration␣%d␣had␣max␣diff␣%g\n", mems, iter + 1, max_diff);
    }
    if (max_diff < threshold ∧ iter ≥ min_iter) break;
  }
  ⟨ Output a reduced problem 42 ⟩;
```

This code is used in section 3.

36.

```
⟨ Global variables 4 ⟩ +≡
  int iter;   /* number of the current iteration */
  double acc, etabar, pi0, pi1, old_eta, new_eta, new_gam, factor, rein, diff;
    /* intermediate registers for floating-point calculations */
  double max_diff;   /* biggest change from old_eta to new_eta */
  double factor;   /* damper^t if we've reinforced t times */
  int azf;   /* number of zero factors suppressed from acc */
  int max_iter;
```

37. The macro *gb_next_rand*( ) delivers a 31-bit random integer, and my convention is to charge four mems whenever it is called.

The initial values of $\eta_{c\to l}$ are random, but the initial values of the external fields η_l are zero.

After this point the computation becomes deterministic.

```
⟨ Initialize all η's to random fractions 37 ⟩ ≡
  for (k = 0; k < cells; k++) mems += 5, mem[k].eta.d = ((double)(gb_next_rand()))/2147483647.0;
  for (k = 0; k < vars + vars; k += 2) ooo, lmem[k].eta = 0.0, lmem[k + 1].eta = 0.0;
```

This code is used in section 35.

38.

```
⟨ Compute the π's 38 ⟩ ≡
  for (l = 0; l < vars + vars; l++) {
    if (o, lmem[l].eta ≡ 1.0) acc = 1.0, azf = 1;
    else acc = 1.0 − lmem[l].eta, azf = 0;
    for (j = lmem[l].link; j; j = mem[j − 1].next) {
      o, etabar = 1.0 − mem[j − 1].eta.d;
      if (etabar ≡ 0.0) azf++;
      else acc *= etabar;
    }
    oo, lmem[l].zf = azf, lmem[l].pi = acc;
  }
```

This code is used in section 35.

39. Either η_l or $\eta_{\bar l}$ is zero; the other is $(1 - \mathit{factor})$ times $|p - q|$, where p and q are the normalized forces that favor l and $\bar l$.

In this loop $l = 2k$, when we process variable k. The *rating* field of l is set to $+1$, 0, or -1 if we currently rate the variable's value as 1, $*$, or 0.

This rating "field" is based on what the physicists also call a "field," but in a different context: They consider that literal l tends to be $(1, 0, *)$ with probabilities that are respectively proportional to $(\pi_{\bar l}(1-\pi_l)$, $\pi_l(1-\pi_{\bar l})$, $\pi_{\bar l}\pi_l)$. These probabilities can be normalized so that they are (p, q, r) with $p + q + r = 1$. The rating is 0 if and only if $r \ge \max\{p, q\}$; otherwise it's $+1$ when $p > q$, or -1 when $p < q$. The condition $r \ge \max\{p, q\}$ turns out to be equivalent to saying that π_l and $\pi_{\bar l}$ are both ≥ 0.5. Later we will use $|p - q|$ to decide the "bias" of a literal.

⟨ Adjust the reinforcement fields 39 ⟩ ≡

```
{
  factor *= damper;
  rein = 1.0 - factor;
  if (verbose & show_details) fprintf(stderr, "␣(rein=%.15g)\n", rein);
  for (l = 0; l < vars + vars; l += 2) {
    if (o, lmem[l].zf) pi0 = 0.0;
    else o, pi0 = lmem[l].pi;
    if (o, lmem[l + 1].zf) pi1 = 0.0;
    else o, pi1 = lmem[l + 1].pi;
    if (pi0 + pi1 ≡ 0.0) {
      if (verbose & show_basics)
        fprintf(stderr, "Sorry,␣a␣contradiction␣was␣found␣after␣iteration␣%d!\n", iter);
      goto contradiction;
    }
    if (pi1 > pi0) {
      o, lmem[l].rating = (pi0 ≥ 0.5 ? 0 : 1);
      if ((verbose & show_gory_details) ∧ lmem[l + 1].eta)
        fprintf(stderr, "␣eta(~%.8s)␣reset\n", nmem[l ≫ 1].ch8);
      oo, lmem[l].eta = rein * (pi1 - pi0)/(pi0 + pi1 - pi0 * pi1), lmem[l + 1].eta = 0.0;
    } else {
      o, lmem[l].rating = (pi1 ≥ 0.5 ? 0 : -1);
      if ((verbose & show_gory_details) ∧ lmem[l].eta)
        fprintf(stderr, "␣eta(%.8s)␣reset\n", nmem[l ≫ 1].ch8);
      oo, lmem[l + 1].eta = rein * (pi0 - pi1)/(pi0 + pi1 - pi0 * pi1), lmem[l].eta = 0.0;
    }
  }
}
```

This code is used in section 35.

40. A clause is "pseudo-satisfied" if it contains a variable whose current value is rated $*$, or if it is satisfied in the normal way. With luck, we get to a pseudo-satisfied state before *max_diff* gets small. (This seems to be a transient phenomenon in many examples: If we wait for *max_diff* to get small, the π's might all be approaching 1 and very few variables would become fixed.)

⟨ Exit if the clauses are pseudo-satisfied 40 ⟩ ≡

```
for (k = c = 0; c < clauses; c++) {
  for (o; k < cmem[c + 1].start; k++) {
    oo, l = mem[k].lit, p = lmem[l & -2].rating;
    if (p ≡ 0) goto ok;
    if (((int) p < 0) ≡ (l & 1)) goto ok;
  }
  goto not_ok;    /* clause not pseudo-satisfied */
ok: k = cmem[c + 1].start;
  continue;
}
if (verbose & show_details)
  fprintf(stderr, "Clauses␣pseudo-satisfied␣on␣iteration␣%d\n", iter + 1);
break;    /* yes, we made it through all of them */
not_ok:
```

This code is used in section 35.

41. If the clause is $l_1 \vee \cdots \vee l_k$, we compute ratios $\gamma_1, \ldots, \gamma_k$ representing the perceived difficulty of making l_i true; then η_i is the product $\gamma_1 \ldots \gamma_{i-1}\gamma_{i+1} \ldots \gamma_k$.

⟨ Update the η's 41 ⟩ ≡

```
max_diff = 0.0;
for (k = c = 0; c < clauses; c++) {
  acc = 1.0, azf = 0;
  for (o, j = 0; k < cmem[c + 1].start; j++, k++) {
    o, l = mem[k].lit;
    if (o, lmem[l ⊕ 1].zf) pi0 = 0.0;
    else o, pi0 = lmem[l ⊕ 1].pi;
    o, old_eta = mem[k].eta.d;
    if (old_eta ≡ 1.0) {
      if (o, lmem[l].zf > 1) pi1 = 0.0;
      else o, pi1 = lmem[l].pi;
    } else if (o, lmem[l].zf) pi1 = 0.0;
    else o, pi1 = lmem[l].pi/(1.0 − old_eta);
    pi1 = pi1 ∗ (1.0 − pi0);
    if (pi1 ≡ 0.0) azf++, o, gam[j] = 0.0;
    else {
      new_gam = pi1/(pi1 + pi0);
      o, gam[j] = new_gam;
      acc ∗= new_gam;
    }
  }
  for (i = j; i; i−−) {
    if (o, gam[j − i] ≡ 0.0) {
      if (azf > 1) new_eta = 0.0;
      else new_eta = acc;
    } else if (azf) new_eta = 0.0;
    else new_eta = acc/gam[j − i];
    o, diff = new_eta − mem[k − i].eta.d;
    if (diff > 0) {
      if (diff > max_diff) max_diff = diff;
    } else if (−diff > max_diff) max_diff = −diff;
    o, mem[k − i].eta.d = new_eta;
  }
}
```

This code is used in section 35.

42. The aftermath. When convergence or pseudo-satisfiability is achieved, we want to use the values of π_l to decide which variables should probably become 0 or 1. For example, if π_l is small but $\pi_{\bar l}$ is large, literal l should be true.

```
⟨ Output a reduced problem 42 ⟩ ≡
  if (iter ≡ max_iter) {
    if (verbose & show_basics) fprintf(stderr, "The␣messages␣didn't␣converge.\n");
    goto contradiction;
  }
  if (verbose & show_pis) ⟨ Print all the π's 43 ⟩;
  if (verbose & show_histogram) ⟨ Print a two-dimension histogram of π_v versus π_v̄ 44 ⟩;
  ⟨ Decide which variables to fix 45 ⟩;
  ⟨ Preprocess the clauses for reduction 46 ⟩;
  ⟨ Reduce the problem 52 ⟩;
  ⟨ Output the reduced problem 53 ⟩;
  goto done;
contradiction: printf("~~?\n"); done:
```

This code is used in section 35.

43. Here we show not only π_v and $\pi_{\bar v}$ for each variable v, but also the associated "fields" (p, q, r) described above.

```
⟨ Print all the π's 43 ⟩ ≡
  {
    if (iter < max_iter) fprintf(stderr, "converged␣after␣%d␣iterations.\n", iter + 1);
    else fprintf(stderr, "no␣convergence␣(diff␣%g)␣after␣%d␣iterations.\n", max_diff, max_iter);
    fprintf(stderr, "variable␣␣␣␣␣␣pi(v)␣␣␣␣␣␣␣␣pi(~v)␣␣␣␣␣␣␣␣␣1␣␣␣␣0␣␣␣␣*\n");
    for (k = 0; k < vars; k++) {
      double den;
      fprintf(stderr, "%8.8s␣%10.7f(%d)␣%10.7f(%d)", nmem[k].ch8, lmem[k + k].pi, lmem[k + k].zf,
          lmem[k + k + 1].pi, lmem[k + k + 1].zf);
      pi0 = lmem[k + k].pi;
      if (lmem[k + k].zf) pi0 = 0.0;
      pi1 = lmem[k + k + 1].pi;
      if (lmem[k + k + 1].zf) pi1 = 0.0;
      den = pi0 + pi1 − pi0 ∗ pi1;
      fprintf(stderr, "␣␣␣␣%4.2f␣%4.2f␣%4.2f\n", pi1 ∗ (1 − pi0)/den, pi0 ∗ (1 − pi1)/den, pi0 ∗ pi1/den);
    }
  }
```

This code is used in section 42.

44. ⟨ Print a two-dimension histogram of π_v versus $\pi_{\bar v}$ 44 ⟩ ≡

```
{
  uint hist[10][10];
  for (j = 0; j < 10; j++)
    for (k = 0; k < 10; k++) hist[j][k] = 0;
  for (k = 0; k < vars; k++) {
    i = (int)(10 * lmem[k + k].pi), j = (int)(10 * lmem[k + k + 1].pi);
    if (lmem[k + k].zf) i = 0;
    if (lmem[k + k + 1].zf) j = 0;
    if (i ≡ 10) i = 9;
    if (j ≡ 10) j = 9;
    hist[i][j]++;
  }
  fprintf(stderr, "Histogram␣of␣the␣pi's,␣after␣%d␣iterations:\n", iter + 1);
  for (j = 10; j; j--) {
    for (i = 0; i < 10; i++) fprintf(stderr, "%7d", hist[i][j - 1]);
    fprintf(stderr, "\n");
  }
}
```

This code is used in section 42.

45. The difference $b = 100\,|p - q|$ in the field of variable v represents v's percentage bias towards a non-$*$ value. All variables for which b is greater than or equal to the *confidence* parameter are placed into bucket b. Then we go through buckets 100, 99, etc., fixing those variables. We also make a "*unit*" bucket for literals that appear in unit clauses after reduction.

Links within the bucket lists are odd numbers, terminated by 2; they appear in the *rating* fields of *lmem*[1], *lmem*[3], etc.

It's probably unwise for the user to make *confidence* < 50, because the pseudo-satisfiability test rates a variable of field $(.5, 0, .5)$ as a '$*$'. But we haven't ruled that out; after all, this program is just experimental, and it's sometimes interesting to explore the consequences of unwise decisions. Therefore we recompute the *rating* fields in *lmem*[0], *lmem*[2], etc., so that they merely reflect the sign of $p - q$.

```
⟨ Decide which variables to fix 45 ⟩ ≡
  for (k = confidence; k ≤ 100; k++) o, bucket[k] = 2;
  unit = 2;
  for (l = 0; l < vars + vars; l += 2) {
    if (o, lmem[l].zf) pi0 = 0.0;
    else o, pi0 = lmem[l].pi;
    if (o, lmem[l + 1].zf) pi1 = 0.0;
    else o, pi1 = lmem[l + 1].pi;
    if (pi0 + pi1 ≡ 0.0) {
      if (verbose & show_basics) fprintf(stderr, "Sorry,␣a␣contradiction␣was␣found!\n");
      goto contradiction;
    }
    acc = (pi1 − pi0)/(pi0 + pi1 − pi0 ∗ pi1);
    o, lmem[l].rating = acc > 0 ? +1 : acc < 0 ? −1 : 0;
    if (acc < 0) acc = −acc;
    j = (int)(100.0 ∗ acc);
    if (j ≥ confidence) {
      oo, lmem[l + 1].rating = bucket[j];
      o, bucket[j] = l + 1;
      fixcount++;
    }
  }
  if (verbose & show_basics)
    fprintf(stderr, "(fixing␣%d␣variables␣after␣%d␣iterations,␣e=%g)\n", fixcount, iter + 1, max_diff);
```

This code is used in section 42.

46. We're done with the *eta* fields in the clauses of cells. So we replace them now with pointers to the relevant clause numbers.

At this point we also take note of unit clauses that might be present in the input, just in case the user didn't reduce them away before presenting the problem.

```
#define cl(p) mem[p].eta.u    /∗ new use for the eta field ∗/

⟨ Preprocess the clauses for reduction 46 ⟩ ≡
  for (k = c = 0; c < clauses; c++) {
    for ( ; k < cmem[c + 1].start; k++) o, cl(k) = c;
    oo, cmem[c].size = k − cmem[c].start;
    if (cmem[c].size ≡ 1) {
      ⟨ Enforce the unit literal mem[k − 1].lit 51 ⟩;
    }
  }
```

This code is used in section 42.

47. Here now is a subroutine that fixes the variables in a given bucket list.

```
⟨ Subroutines 26 ⟩ +≡
  int fixlist(register int k, int b)
  {
    register int c, j, l, ll, p, q;
    for ( ; k & 1; o, k = lmem[k].rating) {
      if (o, lmem[k − 1].rating < 0) l = k;
      else l = k − 1;
      printf("␣%s%.8s", l & 1 ? "~" : "", nmem[l ≫ 1].ch8);
      ⟨ Mark the clauses that contain l satisfied 48 ⟩;
      ⟨ Remove l̄ from all clauses 49 ⟩;
    }
    return 1;
  }
```

48. ⟨ Mark the clauses that contain l satisfied 48 ⟩ ≡

```
  for (o, p = lmem[l].link; p; o, p = mem[p − 1].next) {
    oo, c = cl(p − 1), j = cmem[c].size;
    if (j) o, cmem[c].size = 0;
  }
```

This code is used in section 47.

49. Removed literals are flagged by a special code in their *next* field.

```
#define removed (uint)(−1)
⟨ Remove l̄ from all clauses 49 ⟩ ≡
  for (o, p = lmem[l ⊕ 1].link; p; p = q) {
    o, q = mem[p − 1].next;
    oo, c = cl(p − 1), j = cmem[c].size;
    if (j ≡ 0) continue;    /* clause already satisfied */
    oo, mem[p − 1].next = removed, cmem[c].size = j − 1;
    if (j ≡ 2) {
      for (o, p = cmem[c].start; o, mem[p].next ≡ removed; p++) ;
      ⟨ Enforce the unit literal mem[p].lit 50 ⟩;
    }
  }
```

This code is used in section 47.

50. I expect that unit literals will have become sufficiently biased that we've already decided to fix them. But the *unit* bucket is there just in case we didn't.

⟨ Enforce the unit literal *mem*[*p*].*lit* 50 ⟩ ≡

```
ll = mem[p].lit;
if (ll & 1) {
  if (o, lmem[ll].rating) {
    if (o, lmem[ll − 1].rating > 0) goto contra;
  } else {
    o, lmem[ll − 1].rating = −1;
    o, lmem[ll].rating = unit, unit = ll, unitcount++;
  }
} else {
  if (o, lmem[ll + 1].rating) {
    if (o, lmem[ll].rating < 0) {
    contra: printf("\n");
      fprintf(stderr, "Oops, clause %d is contradicted", c);
      if (b ≥ 0) fprintf(stderr, " in bucket %d!\n", b);
      else fprintf(stderr, " while propagating unit literals!\n");
      return 0;
    }
  } else {
    o, lmem[ll].rating = +1;
    o, lmem[ll + 1].rating = unit, unit = ll + 1, unitcount++;
  }
}
```

This code is used in section 49.

51. ⟨ Enforce the unit literal *mem*[*k* − 1].*lit* 51 ⟩ ≡

```
ll = mem[k − 1].lit;
if (ll & 1) {
  if (o, lmem[ll].rating) {
    if (o, lmem[ll − 1].rating > 0) goto contra;
  } else {
    o, lmem[ll − 1].rating = −1;
    o, lmem[ll].rating = unit, unit = ll, unitcount++;
  }
} else {
  if (o, lmem[ll + 1].rating) {
    if (o, lmem[ll].rating < 0) {
    contra: printf("\n");
      fprintf(stderr, "Oops, clause %d is contradicted!\n", c);
      goto contradiction;
    }
  } else {
    o, lmem[ll].rating = +1;
    o, lmem[ll + 1].rating = unit, unit = ll + 1, unitcount++;
  }
}
```

This code is used in section 46.

52. ⟨ Reduce the problem 52 ⟩ ≡

```
for (k = 100; k ≥ confidence; k−−)
  if (ooo, fixlist(bucket[k], k) ≡ 0) goto contradiction;
while (unit & 1) {
  p = unit, unit = 2;
  if (oo, fixlist(p, −1) ≡ 0) goto contradiction;
}
printf("\n");
if (unitcount ∧ (verbose & show_basics)) fprintf(stderr,
    "(unit␣propagation␣fixed␣%d␣more␣variable%s)\n", unitcount, unitcount ≡ 1 ? "" : "s");
```

This code is used in section 42.

53. ⟨ Output the reduced problem 53 ⟩ ≡

```
sprintf(name_buf, "/tmp/sat9-%d.dat", random_seed);
out_file = fopen(name_buf, "w");
if (¬out_file) {
  fprintf(stderr, "I␣can't␣open␣'%s'␣for␣writing!\n");
  exit(−668);
}
for (kk = k = p = c = 0; c < clauses; c++) {
  o, i = cmem[c].size;
  if (i ≡ 0) {
    o, k = cmem[c + 1].start;
    continue;
  }
  p++;
  while (i > kk) gam[kk++] = 0;
  gam[i − 1] += 1;
  for (o; k < cmem[c + 1].start; k++)
    if (o, mem[k].next ≠ removed) {
      l = mem[k].lit;
      fprintf(out_file, "␣%s%.8s", l & 1 ? "~" : "", nmem[l ≫ 1].ch8);
    }
  fprintf(out_file, "\n");
}
fclose(out_file);
fprintf(stderr, "Reduced␣problem␣of␣%d␣clauses␣written␣on␣file␣%s\n", p, name_buf);
for (i = 0; i < kk; i++)
  if (gam[i]) fprintf(stderr, "␣(%g␣%d-clauses)\n", gam[i], i + 1);
```

This code is used in section 42.

54. ⟨ Global variables 4 ⟩ +≡

```
int bucket[101], unit;
int fixcount, unitcount;
char name_buf[32];
FILE *out_file;
```

55. Index.

SAT9