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 WALKSAT, a notable development of the WALK algorithm that was featured in SAT7. Instead of using completely random choices when a variable is flipped, WALKSAT makes a more informed decision. The WALKSAT method was introduced by B. Selman, H. A. Kautz, and B. Cohen in *National Conference on Artificial Intelligence* **12** (1994), 337–343.

2. If you have already read SAT7, 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 some new command-line options.)

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 list of noncontradictory literals that cover each clause, separated by spaces. ("Noncontradictory" means that we don't have both a literal and its negation.) The input above would, for example, yield '`~?`'; but if the final clause were omitted, the output would be '`~x1 ~x2 x3`', together with either `x4` or `~x4` (but not both). No attempt is made to find all solutions; at most one solution is given.

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 29⟩;
    imems = mems, mems = 0;
    ⟨Solve the problem 37⟩;
    if (verbose & show_basics)
      fprintf(stderr, "Altogether %llu+%llu mems, %llu bytes, %d trial%s, %llu steps.\n",
          imems, mems, bytes, trial + 1, trial ? "s" : "", step);
  }
```

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 */
⟨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 */
  ullng maxsteps;    /* maximum steps per walk (maxthresh * n by default) */
  unsigned int maxthresh = 50;
  int maxtrials = 1000000;    /* maximum walks to try */
  double nongreedprob = 0.4;    /* the probability bias for nongreedy choices */
  unsigned long nongreedthresh;    /* coerced since gb_next_rand is long */
  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 timeout = #1fffffffffffffff;    /* give up after this many mems */
  ullng bytes;    /* memory used by main data structures */
```

See also sections 8 and 25.

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 steps per random walk.
- '`c`⟨ integer ⟩' to define the maximum number of steps per variable, per random walk, if the `t` parameter hasn't been given. (The default is 50.)
- '`w`⟨ integer ⟩' to define the maximum number of walks attempted.
- '`p`⟨ float ⟩' to define the probability *nongreedprob* of nongreedy choices.
- '`T`⟨ integer ⟩' to set *timeout*: This program will abruptly terminate, when it discovers that *mems* > *timeout*.

⟨ 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, "%llu", &maxsteps) − 1); break;
  case 'c': k |= (sscanf(argv[j] + 1, "%u", &maxthresh) − 1); break;
  case 'w': k |= (sscanf(argv[j] + 1, "%d", &maxtrials) − 1); break;
  case 'p': k |= (sscanf(argv[j] + 1, "%lf", &nongreedprob) − 1); break;
  case 'T': k |= (sscanf(argv[j] + 1, "%lld", &timeout) − 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>]", argv[0]);
  fprintf(stderr, "␣[t<n>]␣[c<n>]␣[w<n>]␣[p<f>]␣[T<n>]␣<␣foo.sat\n");
  exit(−1);
}
if (nongreedprob < 0.0 ∨ nongreedprob > 1.0) {
  fprintf(stderr, "Parameter␣p␣should␣be␣between␣0.0␣and␣1.0!\n");
  exit(−666);
}
```

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 eight 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␣%lld␣(%.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␣%llu␣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␣%lld!\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␣%lld␣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␣%lld␣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␣%lld␣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 33.

21. Notice that the old "temporary variable" data goes away here. (A bug bit me in the first version of the code because of this.)

⟨ 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 35.

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

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

This code is used in section 3.

23. SAT solving, version 8. The WALKSAT algorithm is only a little bit more complicated than the WALK method, but the differences mean that we cannot simulate simultaneous runs with bitwise operations.

Let $x = x_1 \ldots x_n$ be a binary vector that represents all n variables, and let T be a given tolerance (representing the amount of patience that we have). We start by setting x to a completely random vector; then we repeat the following steps, at most T times:

> Check to see if x satisfies all the clauses. If so, output x; we're done! If not, select a clause c that isn't true, uniformly at random from all such clauses; say c is the union of k literals, $l_1 \vee \cdots \vee l_k$. Sort those literals according to their "break count," which is the number of clauses that will become false when that literal is flipped. Choose a literal to flip by the following method: If no literal has a break count of zero, and if a biased coin turns up heads, choose l_j at random from among all k literals. Otherwise, choose l_j at random from among those with smallest break count. Then change the bit of x that will make l_j true.

If that random walk doesn't succeed, we can try again with another starting value of x, until we've seen enough failures to be convinced that we're probably doomed to defeat.

24. The data structures are somewhat interesting, but not tricky: There are four main arrays, *cmem*, *vmem*, *mem*, and *tmem*. Structured **clause** nodes appear in *cmem*, and structured **variable** nodes appear in *vmem*. Each clause points to a sequential list of literals in *mem*; each literal points to a sequential list of clauses in *tmem*, which is essentially the "transpose" of the information in *mem*. If *fcount* clauses are currently false, the first *fcount* entries of *cmem* also contain the indices of those clauses.

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 the kth variable.

The symbolic names of variables are kept separately in *nmem*, not in *vmem*, for reasons of efficiency. (Otherwise a **variable** struct would take up five octabytes, and addressing would be slower.)

```
#define value(l) (vmem[(l) >> 1].val ⊕ ((l) & 1))

⟨Type definitions 6⟩ +≡
  typedef struct {
    uint val;   /* the variable's current value */
    uint breakcount;   /* how many clauses are false except for this variable */
    uint pos_start, neg_start;   /* where the clause lists start in tmem */
  } variable;
  typedef struct {
    uint start;   /* where the literal list starts in mem */
    uint tcount;   /* how many of those literals are currently true? */
    uint fplace;   /* if tcount = 0, which fslot holds this clause? */
    uint fslot;   /* the number of a false clause, if needed */
  } clause;
```

25.
```
⟨Global variables 4⟩ +≡
  clause *cmem;   /* the master array of clauses */
  variable *vmem;   /* the master array of variables */
  uint *mem;   /* the master array of literals in clauses */
  uint *cur_mcell;   /* the current cell of interest in mem */
  uint *tmem;   /* the master array of clauses containing literals */
  octa *nmem;   /* the master array of symbolic variable names */
  int trial;   /* which trial are we on? */
  ullng step;   /* which step are we on? */
  uint *best;   /* temporary array to hold literal names for a clause */
```

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:", c);      /* show the clause number */
    for (l = cmem[c − 1].start; l < cmem[c].start; l++) {
      ll = mem[l];
      fprintf(stderr, "␣%s%.8s(%d)", ll & 1 ? "~" : "", nmem[ll ≫ 1].ch8, value(ll));
    }
    fprintf(stderr, "\n");
  }
```

See also sections 27 and 28.

This code is used in section 3.

27. Another version of that routine, used to display unsatisfied clauses in verbose mode, shows the current breakcounts of each literal.

```
⟨ Subroutines 26 ⟩ +≡
  void print_unsat_clause(uint c)
  {
    register uint l, ll;
    fprintf(stderr, "%d:", c);      /* show the clause number */
    for (l = cmem[c − 1].start; l < cmem[c].start; l++) {
      ll = mem[l];
      fprintf(stderr, "␣%s%.8s(%d)", ll & 1 ? "~" : "", nmem[ll ≫ 1].ch8, vmem[ll ≫ 1].breakcount);
    }
    fprintf(stderr, "\n");
  }
```

28. Similarly, we can list the clause numbers that contain a given literal. (Notice the limits on c in the loop here.)

```
⟨ Subroutines 26 ⟩ +≡
  void print_literal_uses(uint l)
  {
    register uint ll, c;
    ll = l ≫ 1;
    fprintf(stderr, "%s%.8s(%d)␣is␣in", l & 1 ? "~" : "", nmem[ll].ch8, value(l));
    for (c = (l & 1 ? vmem[ll].neg_start : vmem[ll].pos_start);
          c < (l & 1 ? vmem[ll + 1].pos_start : vmem[ll].neg_start); c++) fprintf(stderr, "␣%d", tmem[c]);
    fprintf(stderr, "\n");
  }
```

29. 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 29 ⟩ ≡
  ⟨ Allocate the main arrays 30 ⟩;
  ⟨ Initialize the *pos_start* and *neg_start* fields 31 ⟩;
  ⟨ Copy all the temporary cells to the *mem* and *cmem* arrays in proper format 32 ⟩;
  ⟨ Copy all the temporary variable nodes to the *nmem* array in proper format 35 ⟩;
  ⟨ Set up the *tmem* array 34 ⟩;
  ⟨ Check consistency 36 ⟩;

This code is used in section 3.

30. ⟨ Allocate the main arrays 30 ⟩ ≡

```
free(buf); free(hash);   /* a tiny gesture to make a little room */
vmem = (variable *) malloc((vars + 1) * sizeof(variable));
if (¬vmem) {
  fprintf(stderr, "Oops, I can't allocate the vmem array!\n");
  exit(−12);
}
bytes = (vars + 1) * sizeof(variable);
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 = (uint *) malloc(cells * sizeof(uint));
if (¬mem) {
  fprintf(stderr, "Oops, I can't allocate the big mem array!\n");
  exit(−10);
}
bytes += cells * sizeof(uint);
tmem = (uint *) malloc(cells * sizeof(uint));
if (¬tmem) {
  fprintf(stderr, "Oops, I can't allocate the big tmem array!\n");
  exit(−14);
}
bytes += cells * sizeof(uint);
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 29.

31. ⟨ Initialize the *pos_start* and *neg_start* fields 31 ⟩ ≡

```
for (c = vars; c; c−−) o, vmem[c − 1].pos_start = vmem[c − 1].neg_start = 0;
```

This code is used in section 29.

32. ⟨ Copy all the temporary cells to the *mem* and *cmem* arrays in proper format 32 ⟩ ≡

```
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 33 ⟩;
  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);
}
cmem[0].start = 0;
best = (uint *) malloc(kk * sizeof(uint));
if (¬best) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣best␣array!\n");
  exit(-16);
}
bytes += kk * sizeof(uint);
```

This code is used in section 29.

33. 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* 33 ⟩ ≡

```
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 = l = p + p + (i & 1);
  if (l & 1) oo, vmem[l ≫ 1].neg_start++;
  else oo, vmem[l ≫ 1].pos_start++;
}
```

This code is used in section 32.

34. ⟨ Set up the *tmem* array 34 ⟩ ≡

```
for (j = k = 0; k < vars; k++) {
  o, i = vmem[k].pos_start, ii = vmem[k].neg_start;
  o, vmem[k].pos_start = j + i, vmem[k].neg_start = j + i + ii;
  j = j + i + ii;
}
o, vmem[k].pos_start = j;    /* j = cells at this point */
for (c = k = 0, o, kk = cmem[1].start; k < cells; k++) {
  if (k ≡ kk) o, c++, kk = cmem[c + 1].start;
  l = mem[k];
  if (l & 1) ooo, i = vmem[l ≫ 1].neg_start - 1, tmem[i] = c, vmem[l ≫ 1].neg_start = i;
  else ooo, i = vmem[l ≫ 1].pos_start - 1, tmem[i] = c, vmem[l ≫ 1].pos_start = i;
}
```

This code is used in section 29.

35. ⟨ Copy all the temporary variable nodes to the *nmem* array in proper format 35 ⟩ ≡

for $(c = vars;\ c;\ c--)$ {
 ⟨ Move *cur_tmp_var* backward to the previous temporary variable 21 ⟩;
 $o, nmem[c-1].lng = cur\_tmp\_var\to name.lng;$
}

This code is used in section 29.

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

⟨ Check consistency 36 ⟩ ≡

if $(cur\_cell \neq \&cur\_chunk\to cell[0] \lor cur\_chunk\to prev \neq \Lambda \lor cur\_tmp\_var \neq \&cur\_vchunk\to var[0] \lor cur\_vchunk\to prev \neq \Lambda)$ {
 fprintf (*stderr*, "This␣can't␣happen␣(consistency␣check␣failure)!\n");
 exit (−14);
}
free (*cur_chunk*); *free* (*cur_vchunk*);

This code is used in section 29.

37. Doing it. So we take random walks.

⟨ Solve the problem 37 ⟩ ≡

```
if (maxsteps ≡ 0) maxsteps = maxthresh * vars;
nongreedthresh = nongreedprob * (unsigned long) #80000000;
for (trial = 0; trial < maxtrials; trial++) {
  if (delta ∧ (mems ≥ thresh)) {
    thresh += delta;
    fprintf(stderr, "␣after␣%lld␣mems,␣beginning␣trial␣%d\n", mems, trial + 1);
  } else if (verbose & show_choices) fprintf(stderr, "beginning␣trial␣%d\n", trial + 1);
  ⟨ Initialize all values 38 ⟩;
  if (verbose & show_details) ⟨ Print the initial guess 47 ⟩;
  ⟨ Initialize the clause data structures 39 ⟩;
  for (step = 0; ; step++) {
    if (fcount ≡ 0) ⟨ Print a solution and goto done 48 ⟩;
    if (mems > timeout) {
      fprintf(stderr, "TIMEOUT!\n");
      goto done;
    }
    if (step ≡ maxsteps) break;
    ⟨ Choose a random unsatisfied clause, c 40 ⟩;
    ⟨ Choose a literal l in c 41 ⟩;
    ⟨ Flip the value of l 42 ⟩;
  }
}
printf("~?\n");    /* we weren't able to satisfy all the clauses */
if (verbose & show_basics) fprintf(stderr, "DUNNO\n");
trial--;    /* restore the actual number of trials made */
done:
```

This code is used in section 3.

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

⟨ Initialize all values 38 ⟩ ≡

```
for (k = 0, r = 1; k < vars; k++) {
  if (r ≡ 1) mems += 4, r = gb_next_rand() + (1U ≪ 31);
  o, vmem[k].val = r & 1, r ≫= 1;
  vmem[k].breakcount = 0;
}
```

This code is used in section 37.

39. ⟨Initialize the clause data structures 39⟩ ≡

```
fcount = 0;
for (c = k = 0; c < clauses; c++) {
  o, kk = cmem[c + 1].start;
  p = 0;     /* p true literals seen so far in clause c */
  for ( ; k < kk; k++) {
    o, l = mem[k];
    if (o, value(l)) p++, ll = l;
  }
  o, cmem[c].tcount = p;
  if (p ≤ 1) {
    if (p) oo, vmem[ll ≫ 1].breakcount++;
    else oo, cmem[c].fplace = fcount, cmem[fcount++].fslot = c;
  }
}
```

This code is used in section 37.

40. ⟨Choose a random unsatisfied clause, c 40⟩ ≡

```
if (verbose & show_gory_details) {
  fprintf(stderr, "currently␣false:\n");
  for (k = 0; k < fcount; k++) print_unsat_clause(cmem[k].fslot + 1);
}
mems += 5, c = cmem[gb_unif_rand(fcount)].fslot;
if (verbose & show_choices) fprintf(stderr, "in␣%u(%d)", c + 1, fcount);
```

This code is used in section 37.

41. ⟨Choose a literal l in c 41⟩ ≡

```
  oo, k = cmem[c].start, kk = cmem[c + 1].start, h = kk − k;
  ooo, p = mem[k], r = vmem[p ≫ 1].breakcount, best[0] = p, j = 1;
  for (k++; k < kk; k++) {
    oo, p = mem[k], q = vmem[p ≫ 1].breakcount;
    if (q ≤ r) {
      if (q < r) o, r = q, best[0] = p, j = 1;
      else o, best[j++] = p;
    }
  }
  if (r ≡ 0) goto greedy;
  if (mems += 4, (gb_next_rand() < nongreedthresh)) {
    mems += 5, l = mem[kk − 1 − gb_unif_rand(h)], g = 0;
    goto got_l;
  }
greedy: g = 1;
  if (j ≡ 1) l = best[0];
  else mems += 5, l = best[gb_unif_rand(j)];
got_l: p = l ≫ 1;
  if (verbose & show_choices) {
    if (verbose & show_details) fprintf(stderr, ",␣%d*%d␣of␣%d%s,", r, j, h, g ? "" : "␣nongreedy");
    fprintf(stderr, "␣flip␣%s%.8s␣(cost␣%d)\n", vmem[p].val ? "" : "~", nmem[p].ch8,
        vmem[p].breakcount);
  }
```

This code is used in section 37.

42. At this point $p = l \gg 1$.

⟨Flip the value of l 42⟩ ≡

```
if (l & 1) {
  oo, k = vmem[p].neg_start, kk = vmem[p + 1].pos_start;
  ⟨Make clauses tmem[k], tmem[k + 1], ... happier 43⟩;
  o, vmem[p].breakcount = h, vmem[p].val = 0;
  k = vmem[p].pos_start, kk = vmem[p].neg_start;
  ⟨Make clauses tmem[k], tmem[k + 1], ... sadder 44⟩;
} else {
  o, k = vmem[p].pos_start, kk = vmem[p].neg_start;
  ⟨Make clauses tmem[k], tmem[k + 1], ... happier 43⟩;
  o, vmem[p].breakcount = h, vmem[p].val = 1;
  o, k = kk, kk = vmem[p + 1].pos_start;
  ⟨Make clauses tmem[k], tmem[k + 1], ... sadder 44⟩;
}
```

This code is used in section 37.

43. ⟨Make clauses $tmem[k]$, $tmem[k+1]$, ... happier 43⟩ ≡

```
for (h = 0; k < kk; k++) {
  ooo, c = tmem[k], j = cmem[c].tcount, cmem[c].tcount = j + 1;
  if (j ≤ 1) {
    if (j) ⟨Decrease the breakcount of c's critical variable 45⟩
    else {    /* delete c from false list */
      oo, i = cmem[c].fplace, q = cmem[--fcount].fslot;
      oo, cmem[i].fslot = q, cmem[q].fplace = i;
      h++;    /* the flipped literal is now critical */
    }
  }
}
```

This code is used in section 42.

44. ⟨Make clauses $tmem[k]$, $tmem[k+1]$, ... sadder 44⟩ ≡

```
for ( ; k < kk; k++) {
  ooo, c = tmem[k], j = cmem[c].tcount - 1, cmem[c].tcount = j;
  if (j ≤ 1) {
    if (j) ⟨Increase the breakcount of c's critical variable 46⟩
    else {    /* insert c into false list */
      oo, cmem[fcount].fslot = c, cmem[c].fplace = fcount++;
    }
  }
}
```

This code is used in section 42.

45. We know that c has exactly one true literal at this moment.

⟨ Decrease the breakcount of c's critical variable 45 ⟩ ≡

```
{
  for (o, i = cmem[c].start; ; i++) {
    o, q = mem[i];
    if (o, value(q)) break;
  }
  o, vmem[q >> 1].breakcount--;
}
```

This code is used in section 43.

46. As an experiment, I'm swapping the first true literal into the first position of its clause, hoping that subsequent "decrease" loops will thereby be shortened.

⟨ Increase the breakcount of c's critical variable 46 ⟩ ≡

```
{
  for (o, ii = i = cmem[c].start; ; i++) {
    o, q = mem[i];
    if (o, value(q)) break;
  }
  o, vmem[q >> 1].breakcount++;
  if (i != ii) oo, mem[i] = mem[ii], mem[ii] = q;
}
```

This code is used in section 44.

47. ⟨ Print the initial guess 47 ⟩ ≡

```
{
  fprintf(stderr, " initial guess");
  for (k = 0; k < vars; k++) fprintf(stderr, " %s%.8s", vmem[k].val ? "" : "~", nmem[k].ch8);
  fprintf(stderr, "\n");
}
```

This code is used in section 37.

48. ⟨ Print a solution and **goto** *done* 48 ⟩ ≡

```
{
  for (k = 0; k < vars; k++) printf(" %s%.8s", vmem[k].val ? "" : "~", nmem[k].ch8);
  printf("\n");
  if (verbose & show_basics) fprintf(stderr, "!SAT!\n");
  goto done;
}
```

This code is used in section 37.

49. Index.

argc: 3, 5.
argv: 3, 5.
bad_cell: 8, 12, 14, 20.
bad_tmp_var: 8, 12, 13, 21.
best: 25, 32, 41.
breakcount: 24, 27, 38, 39, 41, 42, 45, 46.
buf: 8, 9, 10, 11, 16, 19, 30.
buf_size: 4, 5, 9, 10.
bytes: 3, 4, 30, 32.
c: 3, 26, 27, 28.
cell: 7, 14, 20, 36.
cells: 8, 10, 11, 22, 30, 32, 34.
cells_per_chunk: 7, 14, 20.
chunk: 7, 8, 14, 20.
chunk_struct: 7.
ch8: 6, 16, 26, 27, 28, 41, 47, 48.
clause: 24, 25, 30.
clause_done: 11.
clauses: 8, 10, 11, 12, 16, 19, 22, 30, 32, 39.
cmem: 24, 25, 26, 27, 30, 32, 34, 39, 40, 41, 43, 44, 45, 46.
cur_cell: 8, 12, 14, 20, 33, 36.
cur_chunk: 8, 14, 20, 36.
cur_mcell: 25, 32, 33.
cur_tmp_var: 8, 12, 13, 16, 17, 21, 35, 36.
cur_vchunk: 8, 13, 21, 36.
delta: 4, 5, 37.
done: 37, 48.
empty_clause: 11, 16, 18.
exit: 5, 9, 10, 11, 13, 14, 16, 30, 32, 36.
fcount: 3, 24, 37, 39, 40, 43, 44.
fgets: 10.
fplace: 24, 39, 43, 44.
fprintf: 3, 5, 9, 10, 11, 13, 14, 16, 19, 22, 26, 27, 28, 30, 32, 36, 37, 40, 41, 47, 48.
free: 20, 21, 30, 36.
fslot: 24, 39, 40, 43, 44.
g: 3.
gb_init_rand: 9.
gb_next_rand: 4, 15, 38, 41.
gb_rand: 4.
gb_unif_rand: 40, 41.
got_l: 41.
greedy: 41.
h: 3.
hack_clean: 33.
hack_in: 12.
hack_out: 33.
hash: 8, 9, 17, 30.
hash_bits: 8, 15, 16.
hbits: 4, 5, 9, 10, 16.
i: 3.
ii: 3, 34, 46.
imems: 3, 4.
j: 3.
k: 3.
kk: 3, 32, 34, 39, 41, 42, 43, 44.
l: 3, 26, 27, 28.
ll: 3, 26, 27, 28, 39.
lng: 6, 16, 17, 35.
main: 3.
malloc: 9, 13, 14, 30, 32.
maxsteps: 4, 5, 37.
maxthresh: 4, 5, 37.
maxtrials: 4, 5, 37.
mem: 24, 25, 26, 27, 30, 32, 34, 39, 41, 45, 46.
mems: 3, 4, 5, 37, 38, 40, 41.
name: 6, 16, 17, 35.
neg_start: 24, 28, 31, 33, 34, 42.
new_chunk: 14.
new_vchunk: 13.
next: 6, 17.
nmem: 24, 25, 26, 27, 28, 30, 35, 41, 47, 48.
nongreedprob: 4, 5, 37.
nongreedthresh: 4, 37, 41.
nullclauses: 8, 10, 11, 19.
o: 3.
octa: 6, 25, 30.
old_chunk: 20.
old_vchunk: 21.
oo: 3, 33, 39, 41, 42, 43, 44, 46.
ooo: 3, 34, 41, 43, 44.
p: 3, 12.
pos_start: 24, 28, 31, 33, 34, 42.
prev: 6, 7, 13, 14, 20, 21, 36.
print_clause: 26.
print_literal_uses: 28.
print_unsat_clause: 27, 40.
printf: 37, 48.
q: 3.
r: 3.
random_seed: 4, 5, 9.
serial: 6, 17, 33.
show_basics: 3, 4, 37, 48.
show_choices: 4, 37, 40, 41.
show_details: 4, 37, 41.
show_gory_details: 4, 40.
sscanf: 5.
stamp: 6, 12, 17, 18.
start: 24, 26, 27, 32, 34, 39, 41, 45, 46.
stderr: 3, 5, 9, 10, 11, 13, 14, 16, 19, 22, 26, 27, 28, 30, 32, 36, 37, 40, 41, 47, 48.

stdin: 2, 8, 10.
step: 3, 25, 37.
strlen: 10.
tcount: 24, 39, 43, 44.
thresh: 4, 5, 37.
timeout: 4, 5, 37.
tmem: 24, 25, 28, 30, 34, 43, 44.
tmp_var: 6, 7, 8, 9, 12, 33.
tmp_var_struct: 6.
trial: 3, 25, 37.
uint: 3, 6, 8, 24, 25, 26, 27, 28, 30, 32.
ullng: 3, 4, 8, 12, 25, 33.
u2: 6.
val: 24, 38, 41, 42, 47, 48.
value: 24, 26, 28, 39, 45, 46.
var: 6, 13, 21, 36.
variable: 24, 25, 30.
vars: 8, 10, 17, 22, 30, 31, 34, 35, 37, 38, 47, 48.
vars_per_vchunk: 6, 13, 21.
vchunk: 6, 8, 13, 21.
vchunk_struct: 6.
verbose: 3, 4, 5, 37, 40, 41, 48.
vmem: 24, 25, 27, 28, 30, 31, 33, 34, 38, 39, 41, 42, 45, 46, 47, 48.

⟨ Allocate the main arrays 30 ⟩ Used in section 29.
⟨ Check consistency 36 ⟩ Used in section 29.
⟨ Choose a literal l in c 41 ⟩ Used in section 37.
⟨ Choose a random unsatisfied clause, c 40 ⟩ Used in section 37.
⟨ Copy all the temporary cells to the *mem* and *cmem* arrays in proper format 32 ⟩ Used in section 29.
⟨ Copy all the temporary variable nodes to the *nmem* array in proper format 35 ⟩ Used in section 29.
⟨ Decrease the breakcount of c's critical variable 45 ⟩ Used in section 43.
⟨ Find *cur_tmp_var*→*name* in the hash table at p 17 ⟩ Used in section 12.
⟨ Flip the value of l 42 ⟩ Used in section 37.
⟨ Global variables 4, 8, 25 ⟩ Used in section 3.
⟨ Handle a duplicate literal 18 ⟩ Used in section 12.
⟨ Increase the breakcount of c's critical variable 46 ⟩ Used in section 44.
⟨ Initialize all values 38 ⟩ Used in section 37.
⟨ Initialize everything 9, 15 ⟩ Used in section 3.
⟨ Initialize the clause data structures 39 ⟩ Used in section 37.
⟨ Initialize the *pos_start* and *neg_start* fields 31 ⟩ Used in section 29.
⟨ Input the clause in *buf* 11 ⟩ Used in section 10.
⟨ Input the clauses 10 ⟩ Used in section 3.
⟨ Insert the cells for the literals of clause c 33 ⟩ Used in section 32.
⟨ Install a new **chunk** 14 ⟩ Used in section 12.
⟨ Install a new **vchunk** 13 ⟩ Used in section 12.
⟨ Make clauses $tmem[k]$, $tmem[k+1]$, ... happier 43 ⟩ Used in section 42.
⟨ Make clauses $tmem[k]$, $tmem[k+1]$, ... sadder 44 ⟩ Used in section 42.
⟨ Move *cur_cell* backward to the previous cell 20 ⟩ Used in sections 19 and 33.
⟨ Move *cur_tmp_var* backward to the previous temporary variable 21 ⟩ Used in section 35.
⟨ Print a solution and **goto** *done* 48 ⟩ Used in section 37.
⟨ Print the initial guess 47 ⟩ Used in section 37.
⟨ Process the command line 5 ⟩ Used in section 3.
⟨ Put the variable name beginning at $buf[j]$ in *cur_tmp_var*→*name* and compute its hash code h 16 ⟩ Used in section 12.
⟨ Remove all variables of the current clause 19 ⟩ Used in section 11.
⟨ Report the successful completion of the input phase 22 ⟩ Used in section 3.
⟨ Scan and record a variable; negate it if $i \equiv 1$ 12 ⟩ Used in section 11.
⟨ Set up the main data structures 29 ⟩ Used in section 3.
⟨ Set up the *tmem* array 34 ⟩ Used in section 29.
⟨ Solve the problem 37 ⟩ Used in section 3.
⟨ Subroutines 26, 27, 28 ⟩ Used in section 3.
⟨ Type definitions 6, 7, 24 ⟩ Used in section 3.

SAT8

	Section	Page
Intro	1	1
The I/O wrapper	6	4
SAT solving, version 8	23	11
Initializing the real data structures	29	13
Doing it	37	16
Index	49	20