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.

After experience with ten previous approaches, I finally feel ready to write the program that I plan to describe first: a very simple "no-frills" algorithm that does pretty well on not-too-large problems in spite of being rather short and sweet. The model for this program is the "fast one-level algorithm" of Cynthia A. Brown and Paul W. Purdom, Jr., found in their paper "An empirical comparison of backtracking algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-4** (1982), 309–316. This almost-forgotten paper introduced the idea of *watched literals*, a concept that became famous when it was rediscovered and generalized almost two decades later. Brown and Purdom noticed that the operations of backtracking became quite simple when there is *one* watched literal in each clause; later researchers, unaware of this previous work, discovered how to speed up the process of so-called unit propagation by having *two* watched literals per clause. By presenting the Brown–Purdom algorithm first, I hope to introduce my readers to this concept in a natural and gradual way.

[Note: This program, SAT10, is essentially the prototype for Algorithm 7.2.2.2D.]

If you have already read SAT0 (or some other program of this series), you might as well skip now past all the code for the "I/O wrapper," because you have seen it before.

The input on *stdin* is 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 input clauses are unsatisfiable. 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`', in some order, possibly 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.)

2. 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 */
#define O "%" /* used for percent signs in format strings */
#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 5 ⟩;
  ⟨ Global variables 3 ⟩;
  ⟨ Subroutines 28 ⟩;
  main(int argc, char *argv[])
  {
    register uint h, i, j, l, p, q, r, level, kk, pp, qq, ll, force, nextmove;
    register int c, cc, k, v0, v, vv, vvv;
    ⟨ Process the command line 4 ⟩;
    ⟨ Initialize everything 8 ⟩;
    ⟨ Input the clauses 9 ⟩;
    if (verbose & show_basics) ⟨ Report the successful completion of the input phase 22 ⟩;
    ⟨ Set up the main data structures 33 ⟩;
    imems = mems, mems = 0;
    ⟨ Solve the problem 40 ⟩;
  done: if (verbose & show_basics)
      fprintf(stderr, "Altogether␣"O"llu+"O"llu␣mems,␣"O"llu␣bytes,␣"O"llu␣nodes.\n", imems,
          mems, bytes, nodes);
  }
```

3. **#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_unused_vars* 8 /* *verbose* code to list variables not in solution */

⟨ Global variables 3 ⟩ ≡
```
int random_seed = 0; /* seed for the random words of gb_rand */
int verbose = show_basics + show_unused_vars; /* level of verbosity */
int show_choices_max = 1000000; /* above this level, show_choices is ignored */
int hbits = 8; /* logarithm of the number of the hash lists */
int buf_size = 1024; /* must exceed the length of the longest input line */
FILE *out_file; /* file for optional output */
char *out_name; /* its name */
FILE *primary_file; /* file for optional input */
char *primary_name; /* its name */
int primary_vars; /* the number of primary variables */
ullng imems, mems; /* mem counts */
ullng bytes; /* memory used by main data structures */
ullng nodes; /* total number of branch nodes initiated */
ullng thresh = 0; /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 0; /* report every delta or so mems */
ullng timeout = #1fffffffffffff; /* give up after this many mems */
float eps = 0.1; /* parameter for the minimum score of a watch list */
```
See also sections 7 and 27.

This code is used in section 2.

4. 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*.
- '`c`⟨ positive integer ⟩' to limit the levels on which clauses are shown.
- '`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.
- '`e`⟨ float ⟩' to change the *eps* parameter in rankings of variables for branching.
- '`x`⟨ filename ⟩' to copy the input plus a solution-eliminating clause to the specified file. If the given problem is satisfiable in more than one way, a different solution can be obtained by inputting that file.
- '`V`⟨ filename ⟩' to input a file that lists the names of all "primary" variables. A nonprimary variable will not be used for branching unless its value is forced, or unless all of the primary variables have already been assigned a value.
- '`T`⟨ integer ⟩' to set *timeout*: This program will abruptly terminate, when it discovers that $mems > timeout$.

⟨ Process the command line 4 ⟩ ≡

```
for (j = argc − 1, k = 0; j; j−−)
  switch (argv[j][0]) {
  case 'v': k |= (sscanf(argv[j] + 1, ""O"d", &verbose) − 1); break;
  case 'c': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_max) − 1); break;
  case 'h': k |= (sscanf(argv[j] + 1, ""O"d", &hbits) − 1); break;
  case 'b': k |= (sscanf(argv[j] + 1, ""O"d", &buf_size) − 1); break;
  case 's': k |= (sscanf(argv[j] + 1, ""O"d", &random_seed) − 1); break;
  case 'd': k |= (sscanf(argv[j] + 1, ""O"lld", &delta) − 1); thresh = delta; break;
  case 'e': k |= (sscanf(argv[j] + 1, ""O"f", &eps) − 1); break;
  case 'x': out_name = argv[j] + 1, out_file = fopen(out_name, "w");
    if (¬out_file) fprintf(stderr, "I␣can't␣open␣file␣‘"O"s'␣for␣output!\n", out_name);
    break;
  case 'V': primary_name = argv[j] + 1, primary_file = fopen(primary_name, "r");
    if (¬primary_file) fprintf(stderr, "I␣can't␣open␣file␣‘"O"s'␣for␣input!\n", primary_name);
    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:␣"O"s␣[v<n>]␣[c<n>]␣[h<n>]␣[b<n>]␣[s<n>]␣[d<n>]␣[e<f>]", argv[0]);
  fprintf(stderr, "␣[x<foo>]␣[V<foo>]␣[T<n>]␣<␣foo.sat\n");
  exit(−1);
}
```

This code is used in section 2.

5. 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 into 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 5 ⟩ ≡

```
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 6, 24, 25, and 26.

This code is used in section 2.

6. 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 5 ⟩ +≡

```
typedef struct chunk_struct {
  struct chunk_struct *prev; /* previous chunk allocated (if any) */
  tmp_var *cell[cells_per_chunk];
} chunk;
```

7. ⟨ Global variables 3 ⟩ +≡

```
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? */
int non_clause;  /* is the current clause ignorable? */
```

8. ⟨ Initialize everything 8 ⟩ ≡

```
gb_init_rand(random_seed);
buf = (char *) malloc(buf_size * sizeof(char));
if (¬buf) {
  fprintf(stderr, "Couldn't␣allocate␣the␣input␣buffer␣(buf_size="O"d)!\n", buf_size);
  exit(-2);
}
hash = (tmp_var **) malloc(sizeof(tmp_var) ≪ hbits);
if (¬hash) {
  fprintf(stderr, "Couldn't␣allocate␣"O"d␣hash␣list␣heads␣(hbits="O"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 2.

9. 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 9 ⟩ ≡
  if (primary_file) ⟨ Input the primary variables 10 ⟩;
  while (1) {
    if (¬fgets(buf, buf_size, stdin)) break;
    clauses++;
    if (buf[strlen(buf) − 1] ≠ '\n') {
      fprintf(stderr, "The␣clause␣on␣line␣"O"lld␣("O".20s...)␣is␣too␣long␣for␣me;\n", clauses,
          buf);
      fprintf(stderr, "␣my␣buf_size␣is␣only␣"O"d!\n", buf_size);
      fprintf(stderr, "Please␣use␣the␣command-line␣option␣b<newsize>.\n");
      exit(−4);
    }
    ⟨ Input the clause in buf 11 ⟩;
  }
  if (¬primary_file) primary_vars = vars;
  if ((vars ≫ hbits) ≥ 10) {
    fprintf(stderr, "There␣are␣"O"lld␣variables␣but␣only␣"O"d␣hash␣tables;\n", vars, 1 ≪ hbits);
    while ((vars ≫ hbits) ≥ 10) hbits++;
    fprintf(stderr, "␣maybe␣you␣should␣use␣command-line␣option␣h"O"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␣"O"llu␣variables!\n", vars);
    exit(−664);
  }
  if (clauses ≥ #80000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣"O"llu␣clauses!\n", clauses);
    exit(−665);
  }
  if (cells ≥ #100000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣"O"llu␣occurrences␣of␣literals!\n", cells);
    exit(−666);
  }
```

This code is used in section 2.

10. We input from *primary_file* just as if it were the standard input file, except that all "clauses" are discarded. (Line numbers in error messages are zero.) The effect is to place the primary variables first in the list of all variables: A variable is primary if and only if its index is $\le$ *primary_vars*.

```
⟨ Input the primary variables 10 ⟩ ≡
  {
    while (1) {
      if (¬fgets(buf, buf_size, primary_file)) break;
      if (buf[strlen(buf) − 1] ≠ '\n') {
        fprintf(stderr, "The␣clause␣on␣line␣"O"lld␣("O".20s...)␣is␣too␣long␣for␣me;\n",
            clauses, buf);
        fprintf(stderr, "␣my␣buf_size␣is␣only␣"O"d!\n", buf_size);
        fprintf(stderr, "Please␣use␣the␣command-line␣option␣b<newsize>.\n");
        exit(−4);
      }
      ⟨ Input the clause in buf 11 ⟩;
      ⟨ Remove all variables of the current clause 19 ⟩;
    }
    cells = nullclauses = 0;
    primary_vars = vars;
    if (verbose & show_basics)
      fprintf(stderr, "("O"d␣primary␣variables␣read␣from␣"O"s)\n", primary_vars, primary_name);
  }
```

This code is used in section 9.

11. ⟨ Input the clause in *buf* 11 ⟩ ≡

```
  for (j = k = non_clause = 0; ¬non_clause; ) {
    while (buf[j] ≡ '␣') j++;      /* scan to nonblank */
    if (buf[j] ≡ '\n') break;
    if (buf[j] < '␣' ∨ buf[j] > '~') {
      fprintf(stderr, "Illegal␣character␣(code␣#"O"x)␣in␣the␣clause␣on␣line␣"O"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 ∧ ¬non_clause) {
    fprintf(stderr, "(Empty␣line␣"O"lld␣is␣being␣ignored)\n", clauses);
    nullclauses++;      /* strictly speaking it would be unsatisfiable */
  }
  if (non_clause) ⟨ Remove all variables of the current clause 19 ⟩;
  cells += k;
```

This code is used in sections 9 and 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 ≡ 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 ⟩;
    if (¬non_clause) {
      ⟨ Find cur_tmp_var→name in the hash table at p 17 ⟩;
      if (clauses ∧ (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 8 ⟩ +≡
  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␣"O".9s...␣in␣the␣clause␣on␣line␣"O"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) non_clause = 1;      /* '~' by itself is like 'true' */
  else 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)) non_clause = 1;      /* tautology */
  }
```

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 (non_clause ∧ ((buf[0] ≠ '~') ∨ (buf[1] ≠ '␣')))
      fprintf(stderr, "(The␣clause␣on␣line␣"O"lld␣is␣always␣satisfied)\n", clauses);
    nullclauses++;
  }
```

This code is used in sections 10 and 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 36.

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 37.

22.

```
⟨ Report the successful completion of the input phase 22 ⟩ ≡
  fprintf(stderr, "("O"lld␣variables,␣"O"lld␣clauses,␣"O"llu␣literals␣successfully␣read)\n",
      vars, clauses, cells);
```

This code is used in section 2.

23. SAT solving, version 10. Okay, now comes my hypothetical recreation of the Brown–Purdom SAT solver. (It's unfortunate that no copy of their original program survives.)

The algorithm below essentially solves a satisfiability problem by backtracking. At each level it tries two possibilities for some unset variable, unless it finds an unset variable for which there's only one viable possibility based on previously set variables (thus making a forced move), or unless it finds an unset variable with *no* viable possibilities (in which case it backs up to the previous level of branching).

The key idea is that the first literal in every clause is considered to be "watched," and the watched literal has not been set false. If the algorithm does want to make that literal false, it must first swap another literal of the clause into the first position.

This method can be implemented with extremely simple data structures:

- For each clause c, there's a sequential list of the literals in c.
- For each variable v, there are linked lists of the clauses that are watching v and $\bar{v}$.
- Each variable is either set to true, set to false, or unknown.
- There's a circular list containing all the unset variables whose literals are watched by at least one clause. (This list is called the active ring; we're done when it becomes empty at decision time.)

And of course we remember the current trail of decisions made at each level of the implicit backtrack tree.

24. Each link is a 32-bit integer. (I don't use C pointers in the main data structures, because they occupy 64 bits and clutter up the caches.) Links in the watch lists are indexes of clauses; links in the active ring are indexes of variables. A zero link indicates the end of a list.

The literals within a clause, called "cells," are 32-bit unsigned integers kept in a big array called *mem*. Variable number k, for $1 \le k \le vars$, corresponds to the literals numbered $2k$ and $2k+1$.

Each clause is represented by a pointer to its first cell and by a link to the successor clause (if any) with the same watched literal.

```
⟨ Type definitions 5 ⟩ +≡
  typedef struct {
    uint start;      /* the address in mem where the cells for this clause start */
    uint wlink;      /* link to another clause in the same watch list */
  } clause;
```

25. Several items are stored for each variable: The heads of its two watch lists; the link to the next active variable; a spare field for miscellaneous use; and the 8-byte symbolic name.

(We also keep the current values of variables in a separate array *val*, with one byte for each variable.)

```
#define false 0      /* val code for a false literal */
#define true 1       /* val code for a true literal */
#define unknown -1   /* val code for an unset literal */
⟨ Type definitions 5 ⟩ +≡
  typedef struct {
    uint wlist0, wlist1;      /* heads of the watch lists */
    int next;      /* next item in the ring of active variables */
    uint spare;      /* extra field used only by sanity at the moment */
    octa name;      /* the variable's symbolic name */
  } variable;
```

26. The backtracking process maintains a sequential stack of state information.

```
⟨ Type definitions 5 ⟩ +≡
  typedef struct {
    int var;      /* variable whose value is being set */
    int move;      /* code for what we're setting it */
  } state;
```

27. ⟨ Global variables 3 ⟩ +≡

```
uint *mem;      /* the master array of cells */
clause *cmem;   /* the master array of clauses */
variable *vmem;   /* the master array of variables */
char *val;      /* the master array of variable values */
state *smem;    /* the stack of choices made so far */
uint active;    /* an item in the active ring, or zero if that ring is empty */
```

28. 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.

Incidentally, the clause numbers reported to the user after the input phase may differ from the line numbers reported during the input phase, when *nullclauses* > 0.

⟨ Subroutines 28 ⟩ ≡

```
void print_clause(int c)
{
  register uint k, l;
  printf(""O"d:", c);      /* show the clause number */
  for (k = cmem[c].start; k < cmem[c - 1].start; k++) {
    l = mem[k];
    printf("␣"O"s"O".8s", l & 1 ? "~" : "", vmem[l >> 1].name.ch8);      /* kth literal */
  }
  printf("\n");
}
```

See also sections 29, 30, 31, and 32.

This code is used in section 2.

29. Similarly we can print out all of the clauses that currently watch a particular literal.

⟨ Subroutines 28 ⟩ +≡

```
void print_watches_for(int l)
{
  register int c;
  if (l & 1) c = vmem[l >> 1].wlist1;
  else c = vmem[l >> 1].wlist0;
  for ( ; c; c = cmem[c].wlink) print_clause(c);
}
```

30. ⟨ Subroutines 28 ⟩ +≡

```
void print_ring(void)
{
  register int p;
  printf("Ring:");
  if (active) {
    for (p = vmem[active].next; ; p = vmem[p].next) {
      printf("␣"O".8s", vmem[p].name.ch8);
      if (p ≡ active) break;
    }
  }
  printf("\n");
}
```

31. Speaking of debugging, here's a routine to check if the redundant parts of our data structure have gone awry.

```
#define sanity_checking 0  /* set this to 1 if you suspect a bug */
⟨ Subroutines 28 ⟩ +≡
  void sanity(void)
  {
    register int k, l, c, v;
    if (active) {
      for (v = vmem[active].next; ; v = vmem[v].next) {
        vmem[v].spare = 1;  /* all spare fields assumed zero otherwise */
        if (v ≡ active) break;
      }
    }
    k = 0;
    for (v = 1; v ≤ vars; v++) {
      for (c = vmem[v].wlist0; c; c = cmem[c].wlink) {
        k++;
        if (mem[cmem[c].start] ≠ v + v)
          fprintf(stderr, "Clause␣%d␣watches␣%d␣not␣%d!\n", c, mem[cmem[c].start], v + v);
        else if (val[v] ≡ false)
          fprintf(stderr, "Clause␣%d␣watches␣the␣false␣literal␣%d!\n", c, (v + v));
      }
      for (c = vmem[v].wlist1; c; c = cmem[c].wlink) {
        k++;
        if (mem[cmem[c].start] ≠ v + v + 1)
          fprintf(stderr, "Clause␣%d␣watches␣%d␣not␣%d!\n", c, mem[cmem[c].start], v+v+1);
        else if (val[v] ≡ true)
          fprintf(stderr, "Clause␣%d␣watches␣the␣false␣literal␣%d!\n", c, (v + v + 1));
      }
      if (vmem[v].spare ≡ 0 ∧ val[v] ≡ unknown ∧ (vmem[v].wlist0 ∨ vmem[v].wlist1))
        fprintf(stderr, "Variable␣%d.8s␣should␣be␣in␣the␣active␣ring!\n", vmem[v].name.ch8);
      if (vmem[v].spare ≡ 1 ∧ (val[v] ≠ unknown ∨ ((vmem[v].wlist0 | vmem[v].wlist1) ≡ 0)))
        fprintf(stderr, "Variable␣%d.8s␣should␣not␣be␣in␣the␣active␣ring!\n",
            vmem[v].name.ch8);
      vmem[v].spare = 0;
    }
    if (k ≠ clauses)
      fprintf(stderr, "Oops:␣%d␣of␣%d␣clauses␣are␣being␣watched!\n", k, clauses);
  }
```

32. In long runs it's helpful to know how far we've gotten.

```
⟨ Subroutines 28 ⟩ +≡
  void print_state(int l)
  {
    register int k;
    fprintf(stderr, "␣after␣%d␣mems:", mems);
    for (k = 1; k ≤ l; k++) fprintf(stderr, "%c", smem[k].move + '0');
    fprintf(stderr, "\n");
    fflush(stderr);
  }
```

33. 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. The code below is, of course, similar to what has worked in previous programs of this series.

```
⟨ Set up the main data structures 33 ⟩ ≡
  ⟨ Allocate the main arrays 34 ⟩;
  ⟨ Copy all the temporary cells to the mem and cmem arrays in proper format 35 ⟩;
  ⟨ Copy all the temporary variable nodes to the vmem array in proper format 37 ⟩;
  ⟨ Check consistency 38 ⟩;
  if (out_file) ⟨ Copy all the clauses to out_file 39 ⟩;
```

This code is used in section 2.

34.

```
⟨ Allocate the main arrays 34 ⟩ ≡
  free(buf); free(hash);      /* a tiny gesture to make a little room */
  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);
  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);
  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);
  smem = (state *) malloc((vars + 1) * sizeof(state));
  if (¬smem) {
    fprintf(stderr, "Oops, I can't allocate the smem array!\n");
    exit(−13);
  }
  bytes += (vars + 1) * sizeof(state);
  val = (char *) malloc((vars + 1) * sizeof(char));
  if (¬val) {
    fprintf(stderr, "Oops, I can't allocate the val array!\n");
    exit(−14);
  }
  bytes += (vars + 1) * sizeof(char);
```

This code is used in section 33.

35. ⟨ Copy all the temporary cells to the *mem* and *cmem* arrays in proper format 35 ⟩ ≡

```
for (j = 1; j ≤ vars; j++) {
  o, vmem[j].wlist0 = vmem[j].wlist1 = 0;
  o, val[j] = unknown;
}
for (c = clauses, j = 0; c; c--) {
  o, cmem[c].start = k = j;
  ⟨ Insert the cells for the literals of clause c 36 ⟩;
  l = mem[k];
  if (l & 1) ooo, p = vmem[l ≫ 1].wlist1, cmem[c].wlink = p, vmem[l ≫ 1].wlist1 = c;
  else ooo, p = vmem[l ≫ 1].wlist0, cmem[c].wlink = p, vmem[l ≫ 1].wlist0 = c;
}
if (j ≠ cells) {
  fprintf(stderr, "Oh␣oh,␣something␣happened␣to␣"O"d␣cells!\n", (int) cells − j);
  exit(−15);
}
o, cmem[c].start = j;
```

This code is used in section 33.

36. 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 36 ⟩ ≡

```
for (i = 0; i < 2; ) {
  ⟨ Move cur_cell backward to the previous cell 20 ⟩;
  i = hack_out(*cur_cell);
  p = hack_clean(*cur_cell)→serial;
  p += p + (i & 1) + 2;
  o, mem[j++] = p;
}
```

This code is used in section 35.

37. ⟨ Copy all the temporary variable nodes to the *vmem* array in proper format 37 ⟩ ≡

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

This code is used in section 33.

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

⟨ Check consistency 38 ⟩ ≡

```
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 33.

39. ⟨ Copy all the clauses to *out_file* 39 ⟩ ≡

```
{
  for (k = 0, c = clauses; c; c--) {
    for ( ; k < cmem[c - 1].start; k++) {
      l = mem[k];
      fprintf(out_file, "␣"O"s"O".8s", l & 1 ? "~" : "", vmem[l ≫ 1].name.ch8);      /* kth literal */
    }
    fprintf(out_file, "\n");
  }
  fflush(out_file);      /* complete the copy of input clauses */
}
```

This code is used in section 33.

40. Doing it. Now comes ye olde basic backtrack, but simplified because updates to the watch lists don't have to be undone.

At level l of the backtrack process we record the variable, $smem[l].var$, whose value is being specified, and its chosen value, $smem[l].move$. The latter value is 0 or 1 if we're making a binary branch and we're trying first to make the variable true or false, respectively; it is 3 or 2 if that move failed and we're trying the other alternative. It is 4 or 5 if the move was forced and the variable had to be set respectively to true or false.

```
⟨ Solve the problem 40 ⟩ ≡
  o, level = 0, smem[0].move = 0;
  ⟨ Initialize the active ring 41 ⟩;
choose: if (sanity_checking) sanity( );
  if (delta ∧ (mems ≥ thresh)) thresh += delta, print_state(level);
  if (mems > timeout) {
    fprintf(stderr, "TIMEOUT!\n");
    goto done;
  }
  ⟨ Decide what to do next, going either to branch or forcedmove or backup or satisfied 42 ⟩;
branch: o, nextmove = (vmem[v].wlist0 ≡ 0 ∨ vmem[v].wlist1 ≠ 0);
  nodes++;
forcedmove: level++;      /* at this point vmem[active].next = v is the branch variable */
  o, smem[level].var = v, smem[level].move = nextmove;
  if (active ≡ v) active = 0;      /* the ring becomes empty */
  else oo, h = vmem[v].next, vmem[active].next = h;      /* delete v from the ring */
makemove: ⟨ Set v and update the watch lists for its new value 45 ⟩;
  goto choose;
backup: ⟨ Backtrack to the most recent unforced move 48 ⟩;
```

This code is used in section 2.

41. ⟨ Initialize the active ring 41 ⟩ ≡

```
  for (active = j = 0, k = vars; k; k--)
    if ((o, vmem[k].wlist0) ∨ (vmem[k].wlist1)) {
      if (active ≡ 0) active = k;
      o, vmem[k].next = j, j = k;
    }
  if (active) o, vmem[active].next = j;      /* complete the circle */
```

This code is used in section 40.

42. The basic operation we need to do at each level is to decide which variable in the active ring should be set next. And experience with SAT problems shows that, once we get going, there's usually at least one unset variable whose value is forced by previous clauses.

A literal is forced to be true if and only if there's a clause in its watch list such that all other literals of that clause are already set to false. Therefore we go through the watch list of every active variable until we either find such a literal or discover that there is no forcing at the present time.

When a forced literal is found, we'll want to resume searching for another one at the same place where we left off, thus going cyclically through the active ring. (For if we were to start searching again at the beginning of that list, we'd be covering more or less the same ground as before.)

```
⟨ Decide what to do next, going either to branch or forcedmove or backup or satisfied 42 ⟩ ≡
  if (active ≡ 0) goto satisfied;
  if (verbose & show_details) {
    fprintf(stderr, "␣active␣ring:");
    for (v = vmem[active].next; ; v = vmem[v].next) {
      fprintf(stderr, "␣"O".8s", vmem[v].name.ch8);
      if (v ≡ active) break;
    }
    fprintf(stderr, "\n");
  }
  vv = active, vvv = 0;
newv: o, v = vmem[vv].next;      /* during the search, v is one step ahead of vv */
  force = 0;
  ⟨ Set force = 1 if variable v must be true 43 ⟩;
  ⟨ Set force += 2 if variable v must be false 44 ⟩;
  if (force ≡ 3) goto backup;
  if (force) {
    nextmove = force + 3;
    active = vv;
    goto forcedmove;
  }
  if (vvv ≡ 0 ∧ v ≤ primary_vars) vvv = vv;      /* vvv precedes the first active primary variable */
  if (v ≡ active) {
    if (vvv) vv = active = vvv;
    v = vmem[active].next;
    goto branch;
  }
  vv = v; goto newv;
```

This code is used in section 40.

43. When literal l is watched in clause c, we know that l is the first literal of c. We scan through the other literals until either reaching a literal that's currently unknown or true (whence nothing is forced), or reaching the end (whence l is forced).

If we encounter a true literal l', we could swap it into first position, thereby moving clause c from the watch list of l to the watch list of l', where it probably won't need to be examined as often. But that's a complication that I will postpone for future study, to be explored in variants of this program.

```
⟨ Set force = 1 if variable v must be true 43 ⟩ ≡
  for (o, c = vmem[v].wlist0; c; o, c = cmem[c].wlink) {
    for (oo, k = cmem[c].start + 1; k < cmem[c − 1].start; k++)
      if (oo, val[mem[k] ≫ 1] ≠ (mem[k] & 1)) goto unforced0;
    if (verbose & show_details)
      fprintf(stderr, "(Clause␣"O"d␣reduced␣to␣"O".8s)\n", c, vmem[v].name.ch8);
    force = 1;
    goto forced0;
  unforced0: continue;
  }
  forced0:
```

This code is used in section 42.

44.

```
⟨ Set force += 2 if variable v must be false 44 ⟩ ≡
  for (o, c = vmem[v].wlist1; c; o, c = cmem[c].wlink) {
    for (oo, k = cmem[c].start + 1; k < cmem[c − 1].start; k++)
      if (oo, val[mem[k] ≫ 1] ≠ (mem[k] & 1)) goto unforced1;
    if (verbose & show_details)
      fprintf(stderr, "(Clause␣"O"d␣reduced␣to␣~"O".8s)\n", c, vmem[v].name.ch8);
    force += 2;
    goto forced1;
  unforced1: continue;
  }
  forced1:
```

This code is used in section 42.

45. ⟨ Set v and update the watch lists for its new value 45 ⟩ ≡

```
if ((verbose & show_choices) ∧ level ≤ show_choices_max) {
  fprintf(stderr, "Level␣"O"d,␣", level);
  switch (nextmove) {
  case 0: fprintf(stderr, "trying␣"O".8s", vmem[v].name.ch8); break;
  case 1: fprintf(stderr, "trying␣~"O".8s", vmem[v].name.ch8); break;
  case 2: fprintf(stderr, "retrying␣"O".8s", vmem[v].name.ch8); break;
  case 3: fprintf(stderr, "retrying␣~"O".8s", vmem[v].name.ch8); break;
  case 4: fprintf(stderr, "forcing␣"O".8s", vmem[v].name.ch8); break;
  case 5: fprintf(stderr, "forcing␣~"O".8s", vmem[v].name.ch8); break;
  }
  fprintf(stderr, ",␣"O"lld␣mems\n", mems);
}
if (nextmove & 1) {
  o, val[v] = false;
  oo, c = vmem[v].wlist0, vmem[v].wlist0 = 0, ll = v + v;
} else {
  o, val[v] = true;
  oo, c = vmem[v].wlist1, vmem[v].wlist1 = 0, ll = v + v + 1;
}
⟨ Clear the watch list for ll that starts at c 46 ⟩;
```

This code is used in section 40.

46. ⟨ Clear the watch list for ll that starts at c 46 ⟩ ≡

```
for ( ; c; c = cc) {
  o, cc = cmem[c].wlink;
  for (oo, j = cmem[c].start, k = j + 1; k < cmem[c − 1].start; k++) {
    o, l = mem[k];
    if (o, val[l ≫ 1] ≠ (l & 1)) break;
  }
  if (k ≡ cmem[c − 1].start) {
    fprintf(stderr, "Clause␣"O"d␣can't␣be␣watched!\n", c);      /* "can't happen" */
    exit(−18);
  }
  oo, mem[k] = ll, mem[j] = l;
  ⟨ Put c into the watch list of l 47 ⟩;
}
```

This code is used in section 45.

47. The variable corresponding to l might become active at this point, because it might not be watched anywhere else. In such a case we insert it at the "beginning" of the active ring (that is, just after *active*). We always have $vmem[active].next = h$ at this point, unless $active = 0$.

```
⟨ Put c into the watch list of l 47 ⟩ ≡
  if (verbose & show_details) fprintf(stderr, "(Clause␣"O"d␣now␣watches␣"O"s"O".8s)\n", c,
        l & 1 ? "~" : "", vmem[l ≫ 1].name.ch8);
  o, p = vmem[l ≫ 1].wlist0, q = vmem[l ≫ 1].wlist1;
  if (val[l ≫ 1] ≡ unknown ∧ p ≡ 0 ∧ q ≡ 0) {
    if (active ≡ 0) o, active = h = l ≫ 1, vmem[active].next = h;
    else oo, vmem[l ≫ 1].next = h, h = l ≫ 1, vmem[active].next = h;
  }
  if (l & 1) oo, cmem[c].wlink = q, vmem[l ≫ 1].wlist1 = c;
  else oo, cmem[c].wlink = p, vmem[l ≫ 1].wlist0 = c;
```

This code is used in section 46.

48. If variables need to be reactivated here, we put them just before the place where a conflict was found.

```
⟨ Backtrack to the most recent unforced move 48 ⟩ ≡
  active = vv, h = v;
  while (o, smem[level].move ≥ 2) {
    v = smem[level].var;
    o, val[v] = unknown;
    if ((o, vmem[v].wlist0 ≠ 0) ∨ (vmem[v].wlist1 ≠ 0))
      oo, vmem[v].next = h, h = v, vmem[active].next = h;
    level−−;
  }
  if (level) {
    nextmove = 3 − smem[level].move;
    oo, v = smem[level].var, smem[level].move = nextmove;
    goto makemove;
  }
  if (1) {
    printf("~\n");      /* the formula was unsatisfiable */
    if (verbose & show_basics) fprintf(stderr, "UNSAT\n");
  } else {
  satisfied: if (verbose & show_basics) fprintf(stderr, "!SAT!\n");
    ⟨ Print the solution found 49 ⟩;
  }
```

This code is used in section 40.

49. ⟨ Print the solution found 49 ⟩ ≡

```
for (k = 1; k ≤ level; k++) {
  l = (smem[k].var ≪ 1) + (smem[k].move & 1);
  printf("␣"O"s"O".8s", l & 1 ? "~" : "", vmem[l ≫ 1].name.ch8);
  if (out_file) fprintf(out_file, "␣"O"s"O".8s", l & 1 ? "" : "~", vmem[l ≫ 1].name.ch8);
}
printf("\n");
if (level < vars) {
  if (verbose & show_unused_vars) printf("(Unused:");
  for (v = 1; v ≤ vars; v++)
    if (val[v] ≡ unknown) {
      if (verbose & show_unused_vars) printf("␣"O".8s", vmem[v].name.ch8);
      if (out_file) fprintf(out_file, "␣"O".8s", vmem[v].name.ch8);
    }
  if (verbose & show_unused_vars) printf(")\n");
}
if (out_file) fprintf(out_file, "\n");
```

This code is used in section 48.

50. Index.

SAT10