May 19, 2018 at 02:30

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.

Indeed, this is the first of the series — more precisely the zero-th. I've tried to write it as a primitive baseline against which I'll be able to measure various technical improvements that have been discovered in recent years. This version represents what I think I would have written in the 1960s, when I knew how to do basic backtracking with classical data structures (but very little else). I have intentionally written it *before* having read *any* of the literature about modern SAT-solving techniques; in other words I'm starting with a personal "tabula rasa." My plan is to write new versions as I read the literature, in more-or-less historical order. The only thing that currently distinguishes me from a programmer of forty years ago, SAT-solving-wise, is the knowledge that better methods almost surely do exist.

[*Note:* The present code is slightly modified from the original SAT0. It now corresponds to what has become Algorithm 7.2.2.2A, so that I can make the quantitative experiments recorded in the book.]

Although this is the zero-level program, I'm taking care to adopt conventions for input and output that will be essentially the same in all of the fancier versions that are to come.

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 */
#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 27 ⟩;
  main(int argc, char *argv[])
  {
    register uint c, h, i, j, k, l, p, q, r, level, parity;
    ⟨ Process the command line 4 ⟩;
    ⟨ Initialize everything 8 ⟩;
    ⟨ Input the clauses 9 ⟩;
    if (verbose & show_basics) ⟨ Report the successful completion of the input phase 21 ⟩;
    ⟨ Set up the main data structures 30 ⟩;
    imems = mems, mems = 0;
    ⟨ Solve the problem 39 ⟩;
  done: if (verbose & show_basics) fprintf(stderr,
          "Altogether␣%llu+%llu␣mems,␣%llu␣bytes,␣%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 */
⟨ Global variables 3 ⟩ ≡
  int random_seed = 0; /* seed for the random words of gb_rand */
  int verbose = show_basics; /* 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 */
  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 */
```

See also sections 7 and 26.

This code is used in section 2.

4. On the command line one can say

- '`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; and/or
- '`d`⟨ integer ⟩' to set *delta* for periodic state reports.
- '`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, "%d", &verbose) − 1); break;
    case 'c': k |= (sscanf(argv[j] + 1, "%d", &show_choices_max) − 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, "%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>] [c<n>] [h<n>] [b<n>] [s<n>] [d<n>] [T<n>] < foo.sat\n",
        argv[0]);
    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 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 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, 23, 24, and 25.

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? */
```

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=%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 14.

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 ⟩ ≡
  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 10 ⟩;
  }
  if ((vars ≫ hbits) ≥ 10) {
    fprintf(stderr, "There␣are␣%lld␣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 2.

10. ⟨ Input the clause in *buf* 10 ⟩ ≡

```
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 11 ⟩;
}
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 18 ⟩;
clause_done: *cells* += *k*;

This code is used in section 9.

11. 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 $i \equiv 1$ 11 ⟩ ≡

```
{
  register tmp_var *p;
  if (cur_tmp_var ≡ bad_tmp_var) ⟨ Install a new vchunk 12 ⟩;
  ⟨ Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 15 ⟩;
  ⟨ Find cur_tmp_var→name in the hash table at p 16 ⟩;
  if (p→stamp ≡ clauses ∨ p→stamp ≡ −clauses) ⟨ Handle a duplicate literal 17 ⟩
  else {
    p→stamp = (i ? −clauses : clauses);
    if (cur_cell ≡ bad_cell) ⟨ Install a new chunk 13 ⟩;
    *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 10.

12. ⟨ Install a new **vchunk** 12 ⟩ ≡

```
{
  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 11.

13. ⟨ Install a new **chunk** 13 ⟩ ≡

```
{
  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 11.

14. 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( );
```

15. ⟨ Put the variable name beginning at *buf*[*j*] in *cur_tmp_var→name* and compute its hash code *h* 15 ⟩ ≡

```
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 11.

16. ⟨ Find *cur_tmp_var→name* in the hash table at *p* 16 ⟩ ≡

```
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 11.

17. 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 17 ⟩ ≡

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

This code is used in section 11.

18. 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 18 ⟩ ≡

```
while (k) {
  ⟨ Move cur_cell backward to the previous cell 19 ⟩;
  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 10.

19. ⟨ Move *cur_cell* backward to the previous cell 19 ⟩ ≡

```
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 18 and 33.

20. ⟨ Move *cur_tmp_var* backward to the previous temporary variable 20 ⟩ ≡

```
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.

21. ⟨ Report the successful completion of the input phase 21 ⟩ ≡
fprintf(*stderr*, "(%lld␣variables,␣%lld␣clauses,␣%llu␣literals␣successfully␣read)\n", *vars*, *clauses*, *cells*);

This code is used in section 2.

22. SAT solving, version 0. OK, now comes my hypothetical recreation of a 1960s-style SAT-solver. I knew about doubly linked lists, way back then; but I hadn't yet realized the power of "dancing links." This program does invoke a mild form of the dancing-links principle, because I think I probably would have discovered it if I'd actually worked on satisfiability in those days. (The slightly modified program SAT0-NODANCE shows what my method would have been if I hadn't foreseen dancing links so early.)

The algorithm below essentially tries to solve a satisfiability problem on n variables by first setting the nth variable to its most plausible value, then using the same idea recursively on the remaining $(n-1)$-variable problem. If this doesn't work, we try the other possibility for the nth variable, and the result will either succeed or fail.

Data structures to support that method should allow us to do the following things easily:

- Know, for each variable, the clauses in which that variable occurs, and in how many of them it occurs positively or negatively (two counts).
- Know, for each clause, the literals that it currently contains.
- Delete literals from clauses when they don't satisfy it.
- Delete clauses that have already been satisfied.
- Insert deleted literals and/or clauses when backing up to reconsider previous choices.

The original clause sizes are known in advance. Therefore we can use a combination of sequential and linked memory to accomplish all of these goals.

23. The basic unit in our data structure is called a cell. There's one cell for each literal in each clause, and there also are $2n$ special cells explained below. Each cell belongs to a doubly linked list for the corresponding literal, as well as to a sequential list for the relevant clause. It also "knows" the number of its clause and the number of its literal (which is $2k$ or $2k+1$ for the positive and negative versions of variable k).

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.) The integer is an index into a monolithic array of cells called *mem*.

```
#define listsize owner   /* alternative name for the owner field */
```

⟨ Type definitions 5 ⟩ +≡

```
typedef struct {
  uint flink, blink;   /* forward and backward links for this literal */
  uint owner;   /* clause number, or size in the special list-head cells */
  uint litno;   /* literal number */
} cell;
```

24. Each clause is represented by a pointer to its first cell and by its current size. My first draft of this program also included links to the preceding and following clauses, in a doubly linked cyclic list of all the active clauses that are currently active; but later I realized that such a list is irrelevant, so it might as well be immaterial.

⟨ Type definitions 5 ⟩ +≡

```
typedef struct {
  uint start;   /* the address in mem where the cells for this clause start */
  uint size;   /* the current number of literals in this clause */
} clause;
```

25. If there are n variables, there are $2n$ possible literals. Hence we reserve $2n$ special cells at the beginning of *mem*, for the heads of the lists that link all occurrences of the same literal together.

(Added later: Well, I now actually reserve $2n+2$ special cells, in order to be consistent with the exposition in *TAOCP*, where it was found "friendlier" to speak of x_1 through x_n instead of x_0 through x_{n-1} in the introductory examples.)

The lists for variable k begin in locations $2k$ and $2k+1$, corresponding to its positive and negative incarnations, for $1 \le k \le n$. The *owner* field in the list head tells the total size of the list.

A variable is also represented by its name, for purposes of output. The name appears in a separate array *vmem* of vertex nodes.

```
⟨ Type definitions 5 ⟩ +≡
  typedef struct {
    octa name;      /* the variable's symbolic name */
  } variable;
```

26.
```
⟨ Global variables 3 ⟩ +≡
  cell *mem;      /* the master array of cells */
  uint nonspec;      /* address in mem of the first non-special cell */
  clause *cmem;      /* the master array of clauses */
  variable *vmem;      /* the master array of variables */
  char *move;      /* the stack of choices made so far */
  uint active;      /* the total number of active clauses */
```

27. 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 27 ⟩ ≡
  void print_clause(uint c)
  {
    register uint k, l;
    printf("%d:", c);      /* show the clause number */
    for (k = 0; k < cmem[c].size; k++) {
      l = mem[cmem[c].start + k].litno;
      printf(" %s%.8s", l & 1 ? "~" : "", vmem[l ≫ 1].name.ch8);      /* kth literal */
    }
    printf("\n");
  }
```

See also sections 28 and 29.

This code is used in section 2.

28. Similarly we can print out all of the clauses that use (or originally used) a particular literal.

```
⟨ Subroutines 27 ⟩ +≡
  void print_clauses_for(int l)
  {
    register uint p;
    for (p = mem[l].flink; p ≥ nonspec; p = mem[p].flink) print_clause(mem[p].owner);
  }
```

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

```
⟨ Subroutines 27 ⟩ +≡
  void print_state(int l)
  {
    register int k;
    fprintf(stderr, "␣after␣%lld␣mems:", mems);
    for (k = 1; k ≤ l; k++) fprintf(stderr, "%c", move[k] + '0');
    fprintf(stderr, "\n");
    fflush(stderr);
  }
```

30. Initializing the real data structures. Okay, 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 intended to be a prototype for similar tasks in later programs of this series.

```
⟨ Set up the main data structures 30 ⟩ ≡
  ⟨ Allocate the main arrays 31 ⟩;
  ⟨ Copy all the temporary cells to the mem and cmem arrays in proper format 32 ⟩;
  ⟨ Copy all the temporary variable nodes to the vmem array in proper format 37 ⟩;
  ⟨ Check consistency 38 ⟩;
```

This code is used in section 2.

31. The backtracking routine uses a small array called *move* to record its choices-so-far. We don't count the space for *move* in *bytes*, because each **variable** entry has a spare byte that could have been used.

```
⟨ Allocate the main arrays 31 ⟩ ≡
  free(buf); free(hash);      /* a tiny gesture to make a little room */
  nonspec = vars + vars + 2;
  if (nonspec + cells ≥ #100000000) {
    fprintf(stderr, "Whoa, nonspec+cells is too big for me!\n");
    exit(-667);
  }
  mem = (cell *) malloc((nonspec + cells) * sizeof(cell));
  if (¬mem) {
    fprintf(stderr, "Oops, I can't allocate the big mem array!\n");
    exit(-10);
  }
  bytes = (nonspec + cells) * sizeof(cell);
  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);
  move = (char *) malloc((vars + 1) * sizeof(char));
  if (¬move) {
    fprintf(stderr, "Oops, I can't allocate the move array!\n");
    exit(-13);
  }
```

This code is used in section 30.

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

```
for (j = 0; j < nonspec; j++) o, mem[j].flink = 0;
for (c = clauses; c; c--) {
  o, cmem[c].start = j, cmem[c].size = 0;
  ⟨ Insert the cells for the literals of clause c 33 ⟩;
}
active = clauses;
⟨ Fix up the blink fields and compute the list sizes 34 ⟩;
⟨ Sort the literals within each clause 35 ⟩;
```

This code is used in section 30.

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; j++) {
  ⟨ Move cur_cell backward to the previous cell 19 ⟩;
  i = hack_out(*cur_cell);
  p = hack_clean(*cur_cell)→serial;
  p += p + (i & 1) + 2;
  ooo, mem[j].flink = mem[p].flink, mem[p].flink = j;
  o, mem[j].owner = c, mem[j].litno = p;
}
```

This code is used in section 32.

34. ⟨ Fix up the *blink* fields and compute the list sizes 34 ⟩ ≡

```
for (k = 2; k < nonspec; k++) {
  for (o, i = 0, q = k, p = mem[k].flink; p ≥ nonspec; i++, q = p, o, p = mem[p].flink)
    o, mem[p].blink = q;
  oo, mem[k].blink = q, mem[q].flink = k;
  o, mem[k].listsize = i, mem[k].litno = k;
}
```

This code is used in section 32.

35. The backtracking scheme we will use works nicely when the literals of a clause are arranged so that the first one to be tried comes last. Then a false literal is removed from its clause simply by decreasing the clause's *size* field.

This program tries variable 0 first, then variable 1, etc.; so we want the literals of each clause to be in decreasing order.

The following sorting scheme takes linear time, in the number of cells, because of the characteristics of our data structures. The *size* field of each clause is initially zero.

⟨ Sort the literals within each clause 35 ⟩ ≡

```
for (k = nonspec - 1; k ≥ 2; k--)
  for (o, j = mem[k].flink; j ≥ nonspec; o, j = mem[j].flink) {
    o, c = mem[j].owner;
    o, i = cmem[c].size, p = cmem[c].start + i;
    if (j ≠ p) ⟨ Swap cell j with cell p 36 ⟩;
    o, cmem[c].size = i + 1;
  }
```

This code is used in section 32.

36. Sometimes doubly linked lists make me feel good, even when spending 11 mems. (For mem computation, *flink* and *blink* are assumed to be stored in a single 64-bit word.)

```
⟨ Swap cell j with cell p 36 ⟩ ≡
  {
    o, q = mem[p].flink, r = mem[p].blink;
    oo, mem[p].flink = mem[j].flink, mem[p].blink = mem[j].blink;
    oo, mem[mem[j].flink].blink = mem[mem[j].blink].flink = p;
    o, mem[j].flink = q, mem[j].blink = r;
    oo, mem[q].blink = j, mem[r].flink = j;
    oo, mem[j].litno = mem[p].litno;
    o, mem[p].litno = k;
    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 20 ⟩;
    o, vmem[c].name.lng = cur_tmp_var→name.lng;
  }
```

This code is used in section 30.

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 30.

39. Doing it. Now comes ye olde basic backtrack.

A choice is recorded in the *move* array, as the number 0 or 1 if we're trying first to set the current variable true or false, respectively; it is 3 or 2 if that move failed and we're trying the other alternative.

One slightly nontrivial point arises here: If we can satisfy all of the remaining clauses at some level of the computation, we can do it with our first choice for the relevant variable, because the other choice satisfies at most as many clauses. Thus the test for success doesn't need to be redone at label *try_again* below.

Furthermore, if the complement of our first choice doesn't appear in any active clause, we need not try it. (In other words, a "pure literal" can be assumed to be true. I didn't know about pure literals when I first wrote SAT0, but I've put that knowledge into Algorithm 7.2.2.2A.) Such cases are encoded as moves 4 and 5 instead of 0 and 1.

```
⟨ Solve the problem 39 ⟩ ≡
  level = 1;      /* I used to start at level 0, but Algorithm 7.2.2.2A does this */
newlevel: ooo, move[level] = (mem[level + level].listsize ≤ mem[level + level + 1].listsize);
  if ((verbose & show_choices) ∧ level ≤ show_choices_max) {
    fprintf(stderr, "Level␣%d,␣trying␣%s%.8s", level, move[level] ? "~" : "", vmem[level].name.ch8);
    if (verbose & show_details) fprintf(stderr, "␣(%d:%d,␣%d␣active,␣%lld␣mems)",
          mem[level + level].listsize, mem[level + level + 1].listsize, active, mems);
    fprintf(stderr, "\n");
  }
  parity = move[level] & 1;
  if (mem[level + level + 1 − parity].listsize ≡ 0) move[level] += 4;      /* pure literal; see above */
  else nodes++;
  if (delta ∧ (mems ≥ thresh)) thresh += delta, print_state(level);
  if (active ≡ mem[level + level + parity].listsize) goto satisfied;      /* success! */
  if (mems > timeout) {
    fprintf(stderr, "TIMEOUT!\n");
    goto done;
  }
tryit: parity = move[level] & 1;
  ⟨ Remove variable level from the clauses in the non-chosen list; goto try_again if that would make a
      clause empty 40 ⟩;
  ⟨ Inactivate all clauses of the chosen list 41 ⟩;
  level++; goto newlevel;
try_again: if (o, move[level] < 2) {
    o, move[level] = 3 − move[level];
    if ((verbose & show_choices) ∧ level ≤ show_choices_max) {
      fprintf(stderr, "Level␣%d,␣trying␣again", level);
      if (verbose & show_details) fprintf(stderr, "␣(%d␣active,␣%lld␣mems)\n", active, mems);
      else fprintf(stderr, "\n");
    }
    goto tryit;
  }
  if (level > 1) ⟨ Backtrack to the previous level 42 ⟩;
  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 45 ⟩;
  }
```

This code is used in section 2.

40. Here's where the fact that clauses are sorted really pays off.

```
⟨ Remove variable level from the clauses in the non-chosen list; goto try_again if that would make a clause
      empty 40 ⟩ ≡
  for (o, k = mem[level + level + 1 − parity].flink; k ≥ nonspec; o, k = mem[k].flink) {
    oo, c = mem[k].owner, i = cmem[c].size;
    if (i ≡ 1) {
      if (verbose & show_details) fprintf(stderr, "(Clause␣%d␣now␣unsatisfied)\n", c);
      for (o, k = mem[k].blink; k ≥ nonspec; o, k = mem[k].blink) {
        oo, c = mem[k].owner, i = cmem[c].size;
        o, cmem[c].size = i + 1;
      }
      goto try_again;
    }
    o, cmem[c].size = i − 1;
  }
```

This code is used in section 39.

41. The links dance a little here.

```
⟨ Inactivate all clauses of the chosen list 41 ⟩ ≡
  for (o, k = mem[level + level + parity].flink; k ≥ nonspec; o, k = mem[k].flink) {
    oo, c = mem[k].owner, i = cmem[c].size, j = cmem[c].start;
    for (p = j; p < j + i − 1; p++) {
      o, q = mem[p].flink, r = mem[p].blink;
      oo, mem[q].blink = r, mem[r].flink = q;
      ooo, mem[mem[p].litno].listsize −−;
    }
  }
  o, active −= mem[k].listsize;
```

This code is used in section 39.

42. ⟨ Backtrack to the previous level 42 ⟩ ≡

```
  {
    level −−;
    ⟨ Reactivate all clauses of the chosen list 43 ⟩;
    ⟨ Put variable level back into all clauses on the non-chosen list 44 ⟩;
    goto try_again;
  }
```

This code is used in section 39.

43. Here the dancing links protocol requires us to traverse the list in the reverse direction from what we had before.

```
⟨ Reactivate all clauses of the chosen list 43 ⟩ ≡
  o, parity = move[level] & 1;
  for (o, k = mem[level + level + parity].blink; k ≥ nonspec; o, k = mem[k].blink) {
    oo, c = mem[k].owner, i = cmem[c].size, j = cmem[c].start;
    for (p = j; p < j + i − 1; p++) {
      o, q = mem[p].flink, r = mem[p].blink;
      oo, mem[q].blink = p, mem[r].flink = p;
      ooo, mem[mem[p].litno].listsize++;
    }
  }
  o, active += mem[k].listsize;
```

This code is used in section 42.

44. ⟨ Put variable *level* back into all clauses on the non-chosen list 44 ⟩ ≡

```
  for (o, k = mem[level + level + 1 − parity].flink; k ≥ nonspec; o, k = mem[k].flink) {
    oo, c = mem[k].owner, i = cmem[c].size;
    o, cmem[c].size = i + 1;
  }
```

This code is used in section 42.

45. ⟨ Print the solution found 45 ⟩ ≡

```
  for (k = 1; k ≤ level; k++) printf("␣%s%.8s", move[k] & 1 ? "~" : "", vmem[k].name.ch8);
  printf("\n");
```

This code is used in section 39.

46. Index.

SAT0