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:* Actually this is a special version, written at the end of October 2012. It strips down the old data structures and uses watched literals instead. I think it represents a nearly minimal decent SAT solver. Algorithm 7.2.2.2B is based on this code.]

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 36 ⟩;
   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* = #1ffffffffffffffff;   /∗ 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': $k \mathrel{|}= (sscanf(argv[j] + 1, \texttt{"\%d"}, \&verbose) - 1)$; **break**;
     case 'c': $k \mathrel{|}= (sscanf(argv[j] + 1, \texttt{"\%d"}, \&show\_choices\_max) - 1)$; **break**;
     case 'h': $k \mathrel{|}= (sscanf(argv[j] + 1, \texttt{"\%d"}, \&hbits) - 1)$; **break**;
     case 'b': $k \mathrel{|}= (sscanf(argv[j] + 1, \texttt{"\%d"}, \&buf\_size) - 1)$; **break**;
     case 's': $k \mathrel{|}= (sscanf(argv[j] + 1, \texttt{"\%d"}, \&random\_seed) - 1)$; **break**;
     case 'd': $k \mathrel{|}= (sscanf(argv[j] + 1, \texttt{"\%lld"}, \&delta) - 1)$; *thresh* = *delta*; **break**;
     case 'T': $k \mathrel{|}= (sscanf(argv[j] + 1, \texttt{"\%lld"}, \&timeout) - 1)$; **break**;
     default: $k = 1$;   /∗ unrecognized command-line option ∗/
     }
   if ($k \lor hbits < 0 \lor hbits > 30 \lor buf\_size \le 0$) {
     fprintf(*stderr*, \texttt{"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 ($\neg fgets(buf, buf\_size, stdin)$) **break**;
    clauses ++;
    if ($buf[strlen(buf) - 1] \neq$ '`\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 (($vars \gg hbits$) ≥ 10) {
    $fprintf(stderr,$ `"There␣are␣%lld␣variables␣but␣only␣%d␣hash␣tables;\n"`$, vars, 1 \ll hbits);$
    while (($vars \gg hbits$) ≥ 10) $hbits$ ++;
    $fprintf(stderr,$ `"␣maybe␣you␣should␣use␣command-line␣option␣h%d?\n"`$, hbits);$
  }
  clauses $-=$ *nullclauses*;
  if ($clauses \equiv 0$) {
    $fprintf(stderr,$ `"No␣clauses␣were␣input!\n"`$);$
    $exit(-77);$
  }
  if ($vars \geq$ #`80000000`) {
    $fprintf(stderr,$ `"Whoa,␣the␣input␣had␣%llu␣variables!\n"`$, vars);$
    $exit(-664);$
  }
  if ($clauses \geq$ #`80000000`) {
    $fprintf(stderr,$ `"Whoa,␣the␣input␣had␣%llu␣clauses!\n"`$, clauses);$
    $exit(-665);$
  }
  if ($cells \geq$ #`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 $cur\_tmp\_var \rightarrow name$ in the hash table at p 16⟩ ≡
  for $(p = hash[h];\ p;\ p = p \rightarrow next)$
    if $(p \rightarrow name.lng \equiv cur\_tmp\_var \rightarrow name.lng)$ **break**;
  if $(\neg p)$ {   /∗ new variable found ∗/
    $p = cur\_tmp\_var \mathbin{++};$
    $p \rightarrow next = hash[h], hash[h] = p;$
    $p \rightarrow serial = vars \mathbin{++};$
    $p \rightarrow stamp = 0;$
  }

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 $((p \rightarrow stamp > 0) \equiv (i > 0))$ **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⟩;
    $k \mathbin{--};$
  }
  if $((buf[0] \neq \text{'~'}) \vee (buf[1] \neq \text{'␣'}))$
    $fprintf(stderr, \texttt{"(The␣clause␣on␣line␣\%lld␣is␣always␣satisfied)\\n"}, clauses);$
  $nullclauses \mathbin{++};$

This code is used in section 10.

19. ⟨Move cur_cell backward to the previous cell 19⟩ ≡
  if $(cur\_cell > \&cur\_chunk \rightarrow cell[0])\ cur\_cell \mathbin{--};$
  else {
    register chunk $*old\_chunk = cur\_chunk;$
    $cur\_chunk = old\_chunk \rightarrow prev;\ free(old\_chunk);$
    $bad\_cell = \&cur\_chunk \rightarrow cell[cells\_per\_chunk];$
    $cur\_cell = bad\_cell - 1;$
  }

This code is used in sections 18 and 33.

20. ⟨Move cur_tmp_var backward to the previous temporary variable 20⟩ ≡
  if $(cur\_tmp\_var > \&cur\_vchunk \rightarrow var[0])\ cur\_tmp\_var \mathbin{--};$
  else {
    register vchunk $*old\_vchunk = cur\_vchunk;$
    $cur\_vchunk = old\_vchunk \rightarrow prev;\ free(old\_vchunk);$
    $bad\_tmp\_var = \&cur\_vchunk \rightarrow var[vars\_per\_vchunk];$
    $cur\_tmp\_var = bad\_tmp\_var - 1;$
  }

This code is used in section 34.

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 low-overhead SAT solver, with the lazy data structures of Brown and Purdom 1982 grafted back into 1960s ideas.

The algorithm below essentially tries to solve a satisfiability problem on n variables by first setting x_1 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 x_1, 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 literal, the clauses in which that literal is being "watched."
- Know, for each clause, the literals that it contains, and the literal it watches.
- Swap literals within a clause so that the watched literal is never false.

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. This stripped-down version of SAT0 doesn't really need a special data type for cells, but I've kept it anyway.

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*.

⟨ Type definitions 5 ⟩ +≡
   typedef struct {
     uint *litno*;   /∗ literal number ∗/
   } **cell**;

24. Each clause is represented by a pointer to its first cell, which contains its watched literal. There's also a pointer to another clause that has the same watched literal.

Clauses appear in reverse order. Thus the cells of clause c run from $cmem[c].start$ to $cmem[c-1].start-1$. The first $2n + 2$ "clauses" are special; they serve as list heads for watch lists of each literal.

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

25. A variable is 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 ∗/
   clause ∗*cmem*;   /∗ the master array of clauses ∗/
   uint *nonspec*;   /∗ address in *cmem* of the first non-special clause ∗/
   variable ∗*vmem*;   /∗ the master array of variables ∗/
   char ∗*move*;   /∗ the stack of choices made so far ∗/

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* = *cmem*[*c*].*start*; *k* < *cmem*[*c* − 1].*start*; *k*++) {
         l = *mem*[*k*].*litno*;
         printf ("␣%s%.8s", *l* & 1 ? "~" : "", *vmem*[*l* ≫ 1].*name*.*ch8*);   /∗ *k*th 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 watch a particular literal.

⟨ Subroutines 27 ⟩ +≡
   void *print_clauses_watching*(**int** *l*)
   {
      register uint *p*;
      for (*p* = *cmem*[*l*].*wlink*; *p*; *p* = *cmem*[*p*].*wlink*) *print_clause*(*p*);
   }

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 34 ⟩;
   ⟨ Check consistency 35 ⟩;
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 ∗/
   mem = (**cell** ∗) *malloc*(*cells* ∗ **sizeof**(**cell**));
   if (¬*mem*) {
      fprintf(*stderr*, "Oops,␣I␣can't␣allocate␣the␣big␣mem␣array!\n");
      exit(−10);
   }
   bytes = *cells* ∗ **sizeof**(**cell**);
   nonspec = *vars* + *vars* + 2;
   if (*nonspec* + *clauses* ≥ #100000000) {
      fprintf(*stderr*, "Whoa,␣nonspec+clauses␣is␣too␣big␣for␣me!\n");
      exit(−667);
   }
   cmem = (**clause** ∗) *malloc*((*nonspec* + *clauses*) ∗ **sizeof**(**clause**));
   if (¬*cmem*) {
      fprintf(*stderr*, "Oops,␣I␣can't␣allocate␣the␣cmem␣array!\n");
      exit(−11);
   }
   bytes += (*nonspec* + *clauses*) ∗ **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 ($j = 0$; $j < nonspec$; $j{+}{+}$) $o, cmem[j].start = cmem[j].wlink = 0$;

 for ($c = nonspec + clauses - 1, k = 0$; $c \geq nonspec$; $c{-}{-}$) {

  $j = 0$;

  ⟨Insert the cells for the literals of clause c 33⟩;

 }

 $o, cmem[c].start = k$;

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 $hack\_out(q)$ $((({\bf ullng})\ q)\ \&\ ^\#3)$

#define $hack\_clean(q)$ $(({\bf tmp\_var}\ *)(({\bf ullng})\ q\ \&\ -4))$

⟨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⟩;

  $i = hack\_out(*cur\_cell)$;

  $p = hack\_clean(*cur\_cell){\rightarrow}serial$;

  $p \mathrel{+}= p + (i\ \&\ 1) + 2$;

  if ($j \equiv 0$) $ooo, cmem[c].start = k, cmem[c].wlink = cmem[p].wlink, cmem[p].wlink = c, j = 1$;

  $o, mem[k{+}{+}].litno = p$;

 }

This code is used in section 32.

34. ⟨Copy all the temporary variable nodes to the *vmem* array in proper format 34⟩ ≡

 for ($c = vars$; c; $c{-}{-}$) {

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

  $o, vmem[c].name.lng = cur\_tmp\_var{\rightarrow}name.lng$;

 }

This code is used in section 30.

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

⟨Check consistency 35⟩ ≡

 if ($cur\_cell \neq \&cur\_chunk{\rightarrow}{\bf cell}[0] \vee cur\_chunk{\rightarrow}prev \neq \Lambda \vee cur\_tmp\_var \neq$

   $\&cur\_vchunk{\rightarrow}var[0] \vee cur\_vchunk{\rightarrow}prev \neq \Lambda$) {

  $fprintf(stderr, \texttt{"This\_can't\_happen\_(consistency\_check\_failure)!\\n"})$;

  $exit(-14)$;

 }

 $free(cur\_chunk)$; $free(cur\_vchunk)$;

This code is used in section 30.

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

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

⟨Solve the problem 36⟩ ≡

  level = 1;   /∗ I used to start at level 0, but Algorithm 7.2.2.2B does this ∗/

newlevel:

  if (*level* > *vars*) **goto** *satisfied*;

  oo, *move*[*level*] = (*cmem*[*level* + *level* + 1].*wlink* ≠ 0 ∨ *cmem*[*level* + *level*].*wlink* ≡ 0);

  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*, "␣(%lld␣mems)", *mems*);

    fprintf(*stderr*, "\n");

  }

  nodes ++;

  if (*delta* ∧ (*mems* ≥ *thresh*)) *thresh* += *delta*, *print_state*(*level*);

  if (*mems* > *timeout*) {

    fprintf(*stderr*, "TIMEOUT!\n");

    goto *done*;

  }

tryit: *parity* = *move*[*level*] & 1;

  ⟨Make variable *level* non-watched by the clauses in the non-chosen list; **goto** *try_again* if that would make a clause empty 37⟩;

  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*, "␣(%lld␣mems)\n", *mems*);

      else *fprintf*(*stderr*, "\n");

    }

    goto *tryit*;

  }

  if (*level* > 1) ⟨Backtrack to the previous level 38⟩;

  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 39⟩;

  }

This code is used in section 2.

37. $\langle$ Make variable *level* non-watched by the clauses in the non-chosen list; **goto** *try_again* if that would make a clause empty 37 $\rangle \equiv$

  for $(o, c = cmem[level + level + 1 - parity].wlink;\ c;\ c = q)$ {
    $oo, i = cmem[c].start, q = cmem[c].wlink, j = cmem[c-1].start;$
    for $(p = i + 1;\ p < j;\ p\text{++})$ {
      $o, k = mem[p].litno;$
      if $(k \geq level + level \lor (o, ((move[k \gg 1] \oplus k)\ \&\ 1) \equiv 0))$ **break**;
    }
    if $(p \equiv j)$ {
      if $(verbose\ \&\ show\_details)$ $fprintf(stderr, $ "(Clause␣%d␣contradicted)\n"$, c);$
      $o, cmem[level + level + 1 - parity].wlink = c;$
      goto *try_again*;
    }
    $oo, mem[i].litno = k, mem[p].litno = level + level + 1 - parity;$
    $ooo, cmem[c].wlink = cmem[k].wlink, cmem[k].wlink = c;$
    if $(verbose\ \&\ show\_details)$
      $fprintf(stderr, $ "(Clause␣%d␣now␣watches␣%s%.8s)\n"$, c, k\ \&\ 1\ ?\ $"~"$\ :\ $""$, vmem[k \gg 1].name.ch8);$
  }
  $o, cmem[level + level + 1 - parity].wlink = 0;$

This code is used in section 36.

38. $\langle$ Backtrack to the previous level 38 $\rangle \equiv$
  {
    $level\,\text{−−};$
    goto *try_again*;
  }

This code is used in section 36.

39. $\langle$ Print the solution found 39 $\rangle \equiv$
  for $(k = 1;\ k < level;\ k\text{++})$ $printf($ "␣%s%.8s"$, move[k]\ \&\ 1\ ?\ $"~"$\ :\ $""$, vmem[k].name.ch8);$
  $printf($ "\n"$);$

This code is used in section 36.

40. Index.

⟨ Allocate the main arrays 31 ⟩   Used in section 30.
⟨ Backtrack to the previous level 38 ⟩   Used in section 36.
⟨ Check consistency 35 ⟩   Used in section 30.
⟨ Copy all the temporary cells to the *mem* and *cmem* arrays in proper format 32 ⟩   Used in section 30.
⟨ Copy all the temporary variable nodes to the *vmem* array in proper format 34 ⟩   Used in section 30.
⟨ Find *cur_tmp_var→name* in the hash table at *p* 16 ⟩   Used in section 11.
⟨ Global variables 3, 7, 26 ⟩   Used in section 2.
⟨ Handle a duplicate literal 17 ⟩   Used in section 11.
⟨ Initialize everything 8, 14 ⟩   Used in section 2.
⟨ Input the clause in *buf* 10 ⟩   Used in section 9.
⟨ Input the clauses 9 ⟩   Used in section 2.
⟨ Insert the cells for the literals of clause *c* 33 ⟩   Used in section 32.
⟨ Install a new **chunk** 13 ⟩   Used in section 11.
⟨ Install a new **vchunk** 12 ⟩   Used in section 11.
⟨ Make variable *level* non-watched by the clauses in the non-chosen list; **goto** *try_again* if that would make a clause empty 37 ⟩   Used in section 36.
⟨ Move *cur_cell* backward to the previous cell 19 ⟩   Used in sections 18 and 33.
⟨ Move *cur_tmp_var* backward to the previous temporary variable 20 ⟩   Used in section 34.
⟨ Print the solution found 39 ⟩   Used in section 36.
⟨ Process the command line 4 ⟩   Used in section 2.
⟨ Put the variable name beginning at *buf*[*j*] in *cur_tmp_var→name* and compute its hash code *h* 15 ⟩   Used in section 11.
⟨ Remove all variables of the current clause 18 ⟩   Used in section 10.
⟨ Report the successful completion of the input phase 21 ⟩   Used in section 2.
⟨ Scan and record a variable; negate it if $i \equiv 1$ 11 ⟩   Used in section 10.
⟨ Set up the main data structures 30 ⟩   Used in section 2.
⟨ Solve the problem 36 ⟩   Used in section 2.
⟨ Subroutines 27, 28, 29 ⟩   Used in section 2.
⟨ Type definitions 5, 6, 23, 24, 25 ⟩   Used in section 2.

SAT0W