May 19, 2018 at 02:30

1. Intro. This program is part of a family 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.

I'm hoping that this one, which has the lucky number SAT13, will be the fastest of all, on a majority of the example satisfiability problems that I've been exploring. Why? Because it is based on the "modern" ideas of so-called *conflict driven clause learning* (CDCL) solvers. This approach, pioneered notably by Sakallah and Marques-Silva (GRASP) and by Moskewicz, Madigan, Zhao, Zhang, Malik (CHAFF), has reportedly revolutionized the field, making SAT-solvers applicable to large-scale industrial problems.

My model for SAT13 has been Eén and Sörensson's MiniSAT solver, together with the Biere's PicoSAT solver, both of which were at one time representative of world-class CDCL implementations. The technology has continued to improve, and to become more complex than appropriate for my book to survey; therefore I have not added all the latest bells and whistles. But I think this program decently represents the main CDCL paradigms.

If you have already read SAT10 (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 */
#define mod %   /* used for percent signs denoting remainder in C */

#define show_basics 1   /* verbose code for basic stats */
#define show_choices 2   /* verbose code for backtrack logging */
#define show_details 4   /* verbose code for further commentary */
#define show_gory_details 8   /* verbose code for more yet */
#define show_warmlearn 16   /* verbose code for info about clauses learned during warmups */
#define show_recycling 32   /* verbose code to mention when recycling occurs */
#define show_recycling_details 64   /* verbose code to display clauses that survive recycling */
#define show_restarts 128   /* verbose code to mention when restarts occur */
#define show_initial_clauses 256   /* verbose code to list the unsatisfied clauses */
#define show_watches 512   /* verbose code to show when a watch list changes */
#define show_experiments 4096   /* verbose code sometimes used in change files */

#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 9 ⟩;
  ⟨ Global variables 4 ⟩;
  ⟨ Debugging fallbacks 139 ⟩;
  ⟨ Subroutines 31 ⟩;

  main(int argc, char *argv[])
  {
    register int h, hp, i, j, jj, k, kk, l, ll, lll, p, q, r, s;
    register int c, cc, endc, la, t, u, v, w, x, y;
    register double au, av;

    ⟨ Process the command line 3 ⟩;
    ⟨ Initialize everything 12 ⟩;
    ⟨ Input the clauses 13 ⟩;
    if (verbose & show_basics) ⟨ Report the successful completion of the input phase 25 ⟩;
    ⟨ Set up the main data structures 45 ⟩;
    imems = mems, mems = 0;
    ⟨ Solve the problem 124 ⟩;
  all_done: ⟨ Close the files 8 ⟩;
    ⟨ Print farewell messages 7 ⟩;
  }
```

3. 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 choices are shown by *show_choices*.
- ‘`H`⟨ positive integer ⟩’ to limit the literals whose histories are shown by *print_state*.
- ‘`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. (See *print_state*.)
- ‘`D`⟨ positive integer ⟩’ to set *doomsday*, the number of conflicts after which this world comes to an end.
- ‘`m`⟨ positive integer ⟩’ to adjust the maximum memory size. (The binary logarithm is specified; it must be at most 31.)
- ‘`t`⟨ positive integer ⟩’ to adjust *trivial_limit* (default 10). A trivial clause is substituted for a learned clause when the size of the latter is at least *trivial_limit* more than the size of the former.
- ‘`w`⟨ integer ⟩’ to set *warmups*, the number of “full runs” done after a restart (default 0).
- ‘`f`⟨ positive float ⟩’ to adjust *restart_psi_fraction*, the minimum agility threshold between automatically scheduled restarts (default 0.05).
- ‘`j`⟨ integer ⟩’ to adjust *recycle_bump*, the number of conflicts before the first recycling pass (default 1000).
- ‘`J`⟨ positive integer ⟩’ to adjust *recycle_inc*, the increase in number of conflicts between recycling passes (default 500).
- ‘`a`⟨ float ⟩’ to adjust *alpha*, the weight given to unsatisfied levels when computing a clause’s score during the recycling process (default 0.4). This parameter must be between 0 and 1.
- ‘`r`⟨ positive float ⟩’ to adjust *var_rho*, the damping factor for variable activity scores.
- ‘`R`⟨ positive float ⟩’ to adjust *clause_rho*, the damping factor for clause activity scores.
- ‘`p`⟨ nonnegative float ⟩’ to adjust *rand_prob*, the probability that a branch variable is chosen randomly.
- ‘`P`⟨ nonnegative float ⟩’ to adjust *true_prob*, the probability that a variable’s default initial value is true.
- ‘`x`⟨ filename ⟩’ to output 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 appending that file to the input.
- ‘`l`⟨ filename ⟩’ to output all of the learned clauses of length ≤ *learn_save* to the specified file. (This data can be used, for example, as a certificate of unsatisfiability.)
- ‘`K`⟨ positive integer ⟩’ to adjust the *learn_save* parameter (default 10000).
- ‘`L`⟨ filename ⟩’ to output some learned clauses to the specified file, for purposes of restarting after doomsday. (Those clauses can be combined with the original clauses and simplified by a preprocessor.)
- ‘`z`⟨ filename ⟩’ to input a “polarity file,” which is a list of literals that receive specified default values to be used until forced otherwise. (Literals in this file whose names do not appear within any of the input clauses are ignored.)
- ‘`Z`⟨ filename ⟩’ to output a “polarity file” that will be suitable for restarting after doomsday.
- ‘`T`⟨ integer ⟩’ to set *timeout*: This program will abruptly terminate, when it discovers that *mems* > *timeout*.

```
⟨ Process the command line 3 ⟩ ≡
  for (j = argc − 1, k = 0; j; j−−)
    switch (argv[j][0]) {
      ⟨ Respond to a command-line option, setting k nonzero on error 5 ⟩;
    default: k = 1;     /∗ unrecognized command-line option ∗/
    }
  ⟨ If there’s a problem, print a message about Usage: and exit 6 ⟩;
```

This code is used in section 2.

4. ⟨ Global variables 4 ⟩ ≡

```
int random_seed = 0;   /* seed for the random words of gb_rand */
int verbose = show_basics;   /* level of verbosity */
uint show_choices_max = 1000000;   /* above this level, show_choices is ignored */
int hbits = 8;   /* logarithm of the number of the hash lists */
int print_state_cutoff = 0;   /* don't print more than this many hists */
int buf_size = 1024;   /* must exceed the length of the longest input line */
FILE *out_file;   /* file for optional output of a solution-avoiding clause */
char *out_name;   /* its name */
FILE *restart_file;   /* file for learned clauses to be used in a restart */
char *restart_name;   /* its name */
FILE *learned_file;   /* file for output of every learned clause */
char *learned_name;   /* its name */
int learn_save = 10000;   /* threshold for not outputting to learned_file */
ullng learned_out;   /* this many learned clauses have been output */
FILE *polarity_infile;   /* file for input of literal polarities */
char *polarity_in_name;   /* its name */
FILE *polarity_outfile;   /* file for output of literal polarities */
char *polarity_out_name;   /* its name */
ullng imems, mems;   /* mem counts */
ullng bytes;   /* memory used by main data structures */
ullng nodes;   /* the number of nodes entered */
ullng thresh = 0;   /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 0;   /* report every delta or so mems */
ullng timeout = #1fffffffffffffff;   /* give up after this many mems */
uint memk_max = memk_max_default;   /* binary log of the maximum size of mem */
uint max_cells_used;   /* how much of mem has ever held data? */
int trivial_limit = 10;   /* threshold for substituting trivial clauses */
float var_rho = 0.9;   /* damping factor for variable activity */
float clause_rho = 0.9995;   /* damping factor for clause activity */
float rand_prob = 0.02;   /* probability of choosing at random */
float true_prob = 0.50;   /* probability of starting true on first ascent */
uint rand_prob_thresh;   /* 2^31 times rand_prob */
uint true_prob_thresh;   /* 2^31 times true_prob */
float alpha = 0.4;   /* weighting for unsatisfiable levels in clause scores */
int warmups = 0;   /* the number of full runs done after restart */
ullng total_learned;   /* we've learned this many clauses */
double cells_learned;   /* and this is their total length */
double cells_prelearned;   /* which was this before simplification */
ullng discards;   /* we quickly discarded this many of those clauses */
ullng trivials;   /* we learned this many intentionally trivial clauses */
ullng subsumptions;   /* we subsumed this many clauses on-the-fly */
ullng doomsday = #8000000000000000;   /* force endgame when total_learned exceeds this */
ullng next_recycle;   /* begin recycling when total_learned exceeds this */
ullng recycle_bump = 1000;   /* interval till the next recycling time */
ullng recycle_inc = 500;   /* amount to increase recycle_bump after each round */
ullng next_restart;   /* begin to restart when total_learned exceeds this */
ullng restart_psi;   /* minimum agility threshold for restarts */
float restart_psi_fraction = .05;   /* fractional equivalent of restart_psi */
ullng actual_restarts;
```

See also sections 11, 27, 41, 59, 80, 90, 97, 111, and 140.

This code is used in section 2.

5. ⟨Respond to a command-line option, setting k nonzero on error 5⟩ ≡

```
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", &print_state_cutoff) − 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 'D': k |= (sscanf(argv[j] + 1, ""O"lld", &doomsday) − 1); break;
case 'm': k |= (sscanf(argv[j] + 1, ""O"d", &memk_max) − 1); break;
case 't': k |= (sscanf(argv[j] + 1, ""O"d", &trivial_limit) − 1); break;
case 'w': k |= (sscanf(argv[j] + 1, ""O"d", &warmups) − 1); break;
case 'j': k |= (sscanf(argv[j] + 1, ""O"lld", &recycle_bump) − 1); break;
case 'J': k |= (sscanf(argv[j] + 1, ""O"lld", &recycle_inc) − 1); break;
case 'K': k |= (sscanf(argv[j] + 1, ""O"d", &learn_save) − 1); break;
case 'f': k |= (sscanf(argv[j] + 1, ""O"f", &restart_psi_fraction) − 1); break;
case 'a': k |= (sscanf(argv[j] + 1, ""O"f", &alpha) − 1); break;
case 'r': k |= (sscanf(argv[j] + 1, ""O"f", &var_rho) − 1); break;
case 'R': k |= (sscanf(argv[j] + 1, ""O"f", &clause_rho) − 1); break;
case 'p': k |= (sscanf(argv[j] + 1, ""O"f", &rand_prob) − 1); break;
case 'P': k |= (sscanf(argv[j] + 1, ""O"f", &true_prob) − 1); break;
case 'x': out_name = argv[j] + 1, out_file = fopen(out_name, "w");
  if (¬out_file) fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", out_name);
  break;
case 'l': learned_name = argv[j] + 1, learned_file = fopen(learned_name, "w");
  if (¬learned_file)
    fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", learned_name);
  break;
case 'L': restart_name = argv[j] + 1, restart_file = fopen(restart_name, "w");
  if (¬restart_file)
    fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", restart_name);
  break;
case 'z': polarity_in_name = argv[j] + 1, polarity_infile = fopen(polarity_in_name, "r");
  if (¬polarity_infile)
    fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣reading!\n", polarity_in_name);
  break;
case 'Z': polarity_out_name = argv[j] + 1, polarity_outfile = fopen(polarity_out_name, "w");
  if (¬polarity_outfile)
    fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", polarity_out_name);
  break;
case 'T': k |= (sscanf(argv[j] + 1, ""O"lld", &timeout) − 1); break;
```

This code is used in section 3.

6. ⟨If there's a problem, print a message about `Usage:` and *exit* 6⟩ ≡

```
if (k ∨ hbits < 0 ∨ hbits > 30 ∨ buf_size ≤ 0 ∨ memk_max < 2 ∨ memk_max > 31 ∨ trivial_limit ≤ 0 ∨ (int)
      recycle_inc < 0 ∨ alpha < 0.0 ∨ alpha > 1.0 ∨ rand_prob < 0.0 ∨ true_prob < 0.0 ∨ var_rho ≤
      0.0 ∨ clause_rho ≤ 0.0) {
  fprintf(stderr, "Usage:␣"O"s␣[v<n>]␣[c<n>]␣[H<n>]␣[h<n>]␣[b<n>]␣[s<n>]␣[d<n>]", argv[0]);
  fprintf(stderr, "␣[D<n>]␣[m<n>]␣[t<n>]␣[w<n>]␣[j<n>]␣[J<n>]␣[K<n>]");
  fprintf(stderr, "␣[f<f>]␣[a<f>]␣[r<f>]␣[R<f>]␣[p<f>]␣[P<f>]");
  fprintf(stderr, "␣[x<foo>]␣[l<bar>]␣[L<baz>]␣[z<poi>]␣[Z<poo>]␣[T<n>]␣<␣foo.sat\n");
  exit(−1);
}
```

This code is used in section 3.

7. ⟨Print farewell messages 7⟩ ≡

```
if (verbose & show_basics) {
  fprintf(stderr, "Altogether␣"O"llu+"O"llu␣mems,␣"O"llu␣bytes,␣"O"llu␣node"O"s,", imems,
        mems, bytes, nodes, nodes ≡ 1 ? "" : "s");
  fprintf(stderr, "␣"O"llu␣clauses␣learned", total_learned);
  if (total_learned) fprintf(stderr, "␣(ave␣"O".1f->"O".1f)", cells_prelearned/(double)
        total_learned, cells_learned/(double) total_learned);
  fprintf(stderr, ",␣"O"u␣memcells.\n", max_cells_used);
  if (learned_file) fprintf(stderr, ""O"lld␣learned␣clauses␣written␣to␣file␣`"O"s'.\n",
        learned_out, learned_name);
  if (trivials) fprintf(stderr, "("O"lld␣learned␣clause"O"s␣trivial.)\n", trivials,
        trivials ≡ 1 ? "␣was" : "s␣were");
  if (discards) fprintf(stderr, "("O"lld␣learned␣clause"O"s␣discarded.)\n", discards,
        discards ≡ 1 ? "␣was" : "s␣were");
  if (subsumptions) fprintf(stderr, "("O"lld␣clause"O"s␣subsumed␣on-the-fly.)\n",
        subsumptions, subsumptions ≡ 1 ? "␣was" : "s␣were");
  fprintf(stderr, "("O"lld␣restart"O"s.)\n", actual_restarts, actual_restarts ≡ 1 ? "" : "s");
}
```

This code is used in section 2.

8. ⟨Close the files 8⟩ ≡

```
if (out_file) fclose(out_file);
if (learned_file) fclose(learned_file);
if (restart_file) fclose(restart_file);
if (polarity_infile) fclose(polarity_infile);
if (polarity_outfile) fclose(polarity_outfile);
```

This code is used in section 2.

9. 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 9 ⟩ ≡
  typedef union {
    char ch8[8];
    uint u2[2];
    ullng 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 10, 28, 29, and 30.

This code is used in section 2.

10. 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 9 ⟩ +≡
  typedef struct chunk_struct {
    struct chunk_struct *prev;     /* previous chunk allocated (if any) */
    tmp_var *cell[cells_per_chunk];
  } chunk;
```

11. ⟨ Global variables 4 ⟩ +≡

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

12. ⟨ Initialize everything 12 ⟩ ≡

```
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 18.

This code is used in section 2.

13. 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 13 ⟩ ≡
  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 14 ⟩;
  }
  if ((vars ≫ hbits) ≥ 10) {
    fprintf(stderr, "There␣are␣"O"lld␣variables␣but␣only␣"O"d␣hash␣tables;\n", vars, 1 ≪ hbits);
    for (h = hbits + 1; (vars ≫ h) ≥ 10; h++) ;
    fprintf(stderr, "␣maybe␣you␣should␣use␣command-line␣option␣h"O"d?\n", h);
  }
  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.

14. ⟨Input the clause in *buf* 14⟩ ≡

```
  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␣#"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 15⟩;
  }
  if (k ≡ 0) {
    fprintf(stderr, "(Empty␣line␣"O"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 22⟩;
clause_done: cells += k;
  if (k ≡ 1) unaries++;
  else if (k ≡ 2) binaries++;
```

This code is used in section 13.

15. 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 $i \equiv 1$ 15⟩ ≡

```
  {
    register tmp_var *p;
    if (cur_tmp_var ≡ bad_tmp_var) ⟨Install a new vchunk 16⟩;
    ⟨Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 19⟩;
    ⟨Find cur_tmp_var→name in the hash table at p 20⟩;
    if (p→stamp ≡ clauses ∨ p→stamp ≡ −clauses) ⟨Handle a duplicate literal 21⟩
    else {
      p→stamp = (i ? −clauses : clauses);
      if (cur_cell ≡ bad_cell) ⟨Install a new chunk 17⟩;
      *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 14.

16. ⟨ Install a new **vchunk** 16 ⟩ ≡

```
{
  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 15.

17. ⟨ Install a new **chunk** 17 ⟩ ≡

```
{
  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 15.

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

⟨ Initialize everything 12 ⟩ +≡

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

19. ⟨ Put the variable name beginning at *buf*[*j*] in *cur_tmp_var*→*name* and compute its hash code *h* 19 ⟩ ≡

```
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) goto empty_clause;      /* '~' by itself is like 'true' */
j += l;
h &= (1 ≪ hbits) − 1;
```

This code is used in sections 15 and 79.

20. ⟨ Find *cur_tmp_var→name* in the hash table at *p* 20 ⟩ ≡

```
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 15.

21. 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 21 ⟩ ≡

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

This code is used in section 15.

22. 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 22 ⟩ ≡

```
while (k) {
  ⟨ Move cur_cell backward to the previous cell 23 ⟩;
  k−−;
}
if ((buf[0] ≠ '~') ∨ (buf[1] ≠ '␣'))
  fprintf(stderr, "(The␣clause␣on␣line␣"O"lld␣is␣always␣satisfied)\n", clauses);
nullclauses++;
```

This code is used in section 14.

23. ⟨ Move *cur_cell* backward to the previous cell 23 ⟩ ≡

```
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 22 and 50.

24. ⟨ Move *cur_tmp_var* backward to the previous temporary variable 24 ⟩ ≡

```
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 54.

25. ⟨ Report the successful completion of the input phase 25 ⟩ ≡

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

This code is used in section 2.

26. SAT solving, version 13. The methods used in this program have much in common with what we've seen before in SAT0, SAT1, etc.; yet conflict-driven clause learning is also rather different. So we might as well derive everything from first principles.

As usual, our goal is to find strictly distinct literals that satisfy all of the given clauses, or to prove that those clauses can't all be satisfied. Thus our subgoal, after having created a "trail" $l_0 l_1 \ldots l_t$ of literals that don't falsify any clause, will be to extend that sequence until finding a solution, and to do this without failing unless no solution exists.

If there's a clause c of the form $l \vee \bar{a}_1 \vee \cdots \vee \bar{a}_k$, where a_1 through a_k are in the trail but l isn't, we append l to the trail and say that c is its "reason." This operation, often called unit propagation, is basic to our program; we shall simply call it *forcing.* (We're forced to make l true, if a_1 through a_k are true, because c must be satisfied.) A *conflict* occurs if the complementary literal $\bar{l}$ is already in the trail, because l can't be both true and false; but let's assume for now that no conflicts arise.

If no such forcing clause exists, and if the clauses aren't all satisfied, we choose a new distinct literal in some heuristic way, and append it to the trail with a "reason" of 0. Such literals are called *decisions.* They partition the trail into a sequence of decision levels, with literal l_j belonging to level d if and only if $i_d \le j < i_{d+1}$. In general $0 \le i_1 < i_2 < \cdots$; and we also define $i_0 = 0$. (Level 0 is special; it contains literals that are forced by clauses of length 1, if such clauses exist. Any such literals are unconditionally true. Every other level begins with exactly one decision.)

If the reason for l includes the literal $\bar{l}'$, we say "l depends directly on l'," and we write $l \succ l'$. And if there's a chain of one or more direct dependencies $l \succ l_1 \succ \cdots \succ l_k = l'$, we write $l \succ^+ l'$ and say simply that "l depends on l'." For example, given the three clauses a and $\bar{a} \vee b$ and $\bar{b} \vee \bar{c} \vee d$, we might begin the trail with $l_0 l_1 l_2 l_3 = abcd$, where the first clause is the reason for a, the second clause is the reason for b, and the third clause is the reason for d, while c is a decision. Then $d \succ c$ and $d \succ b$ and $b \succ a$; hence $d \succ^+ a$.

Notice that a literal can depend only on literals that precede it in the trail. Furthermore, every literal l that's forced at level $d > 0$ depends directly on some *other* literal on that same level; hence l must necessarily depend on the dth decision.

The reason for reasons is that we need to deal with conflicts. We will see that every conflict allows us to construct a new clause c that must be true whenever the existing clauses are satisfiable, although c itself does not contain any existing clause. Therefore we can "learn" c by adding it to the existing clauses, and we can try again. This learning process can't go on forever, because only finitely many clauses are possible. Sooner or later we will therefore either find a solution or learn the empty clause.

A conflict clause c_d on decision level d has the form $\bar{l} \vee \bar{a}_1 \vee \cdots \vee \bar{a}_k$, where l and all the a's belong to the trail; furthermore l and at least one a_i belong to level d. We can assume that l is rightmost in the trail, of all the literals in c_d. Hence l cannot be the dth decision; and it has a reason, say $l \vee \bar{a}'_1 \vee \cdots \vee \bar{a}'_{k'}$. Resolving c_d with this reason gives the clause $c = \bar{a}_1 \vee \cdots \vee \bar{a}_k \vee \bar{a}'_1 \vee \cdots \vee \bar{a}'_{k'}$, which includes at least one literal $\bar{l}'$ for which l' is on level d. If more than one such literal is present, we can resolve c with the reason of a rightmost l'; the result will involve negations of literals that are still further to the left. By repeating this process we'll eventually obtain c of the form $\bar{l}' \vee \bar{b}_1 \vee \cdots \vee \bar{b}_r$, where l' is on level d and where b_1 through b_r are on lower levels.

Such a c is learnable, as desired, because it can't contain any existing clause. (Every subclause of c, including c itself, would have given us something to force at a lower level.) We can now discard levels $> d'$ of the trail, where d' is the maximum level of b_1 through b_r; and we append $\bar{l}'$ to the end of level d', with c as its reason. The forcing process now resumes at level d', as if the learned clause had been present all along.

Okay, that's the basic idea of conflict-driven clause learning. Many other issues will come up as we refine it, of course. For example, we'll see that the clause c can often be simplified by removing one or more of its literals $\bar{b}_i$. And we'll want to "unlearn" clauses that outlive their usefulness.

27. What data structures support this procedure? We obviously need to represent the trail, as well as the levels, the values, and the reasons for each of its literals.

A principal concern is to make forcing as fast as possible. Many applications involve numerous binary clauses (that is, clauses of length 2); and binary clauses make forcing quite easy. So we should have a special mechanism to derive binary implications quickly.

Long clauses are also important. (Even if they aren't common in the input, the clauses that we learn may well turn out to involve dozens of literals.) "Watch lists" provide a good way to recognize when such clauses become ready for forcing: We choose two literals in each long clause, neither of which is false, and we pay no attention to that clause until one of its watched literals becomes false. In the latter case, we'll be able to watch it with another literal, unless the clause has become true or it's ready to force something. (We've used a similar idea, but with only one watched literal per clause, in SAT0W and SAT10.)

We'll want a good heuristic for choosing the decision literals. This program adopts the strategy of Eén and Sörensson's MiniSAT, which associates a floating-point *activity* score with each variable, and uses a heap to choose the most active variable.

Learned clauses also have a measure of clause quality devised by Gilles Audemard and Laurent Simon. The original clauses are static and stay in place, but we must periodically decide which of the learned clauses to keep.

```
⟨Global variables 4⟩ +≡
  cel *mem;   /* master array of clause data */
  uint memsize;   /* the number of cells allocated for it */
  uint min_learned;   /* boundary between original and learned clauses */
  uint first_learned;   /* address of the first learned clause */
  uint max_learned;   /* the first unused position of mem */
  int max_lit;   /* value of the largest legal literal */
  uint *bmem;   /* binary clause data */
  literal *lmem;   /* attributes of literals */
  variable *vmem;   /* attributes of variables */
  uint *heap;   /* priority queue for sorting variables by their activity */
  int hn;   /* number of items currently in the heap */
  uint *trail;   /* literals currently assumed, or forced by those assumptions */
  int eptr;   /* just past the end of the trail */
  int ebptr;   /* just past where binary propagations haven't been done yet */
  int lptr;   /* just past where we've checked nonbinary propagations */
  int lbptr;   /* just past where we've checked binary propagations */
  char *history;   /* type of assertion, for diagnostic printouts */
  int llevel;   /* twice the current level */
  int *leveldat;   /* where levels begin; also conflict data on full runs */
```

28. Binary clauses $u \vee v$ are represented by putting v into a list associated with $\bar{u}$ and u into a list associated with $\bar{v}$. These "binary implication" lists are created once and for all at the beginning of the run, as explained below.

Longer clauses (and binary clauses that are learned later) are represented in a big array *mem* of 32-bit integers. (Entries of *mem* are often called "cells" in this documentation.) The literals of clause c are $mem[c].lit$, $mem[c+1].lit$, $mem[c+2].lit$, etc.; the first two of these are "watching" c. The number of literals, $size(c)$, is $mem[c-1].lit$; and we keep links to other clauses being watched by the same literals in $link0(c) = mem[c-2].lit$ and $link1(c) = mem[c-3].lit$.

(Incidentally, this linked structure for watch lists was originally introduced in PicoSAT by Armin Biere [*Journal on Satisfiability, Boolean Modeling and Computation* **4** (2008), 75–97]. Nowadays the fastest solvers use a more complicated mechanism called "blocking literals," due to Niklas Sörensson, which is faster because it is more cache friendly. However, I'm sticking to linked lists, because (1) they don't need dynamic storage allocation of sequential arrays; (2) they use fewer memory accesses; and (3) on modern multithreaded machines they can be implemented so as to avoid the cache misses, by starting up threads whose sole purpose is to preload the link-containing cells into the cache. I expect that software to facilitate such transformations will be widely available before long.)

Sometimes we learn that a clause can be strengthened by removing one of its literals. In such cases we add *sign_bit* to the surplus literal, swap it to the end of the clause, and decrease the *size* field. Except for such deleted literals, the sign bit of every cell in *mem* should be zero. (The earliest cell of a learned clause c is the nonnegative floating-point value $activ(c)$. The final clause should be followed by a zero cell, so that garbage at the end isn't confused with a deleted literal.)

If c is the current reason for literal l, its first literal $mem[c].lit$ is always equal to l. This condition makes it easy to tell if a given clause plays an important role in the current trail.

A learned clause is identifiable by the condition $c \ge min\_learned$. Such clauses have additional information, $range(c) = mem[c-4].lit$ and $activ(c) = mem[c-5].flt$, which will help us decide whether or not to keep them after memory has begun to fill up.

```
#define size(c) mem[(c) − 1].lit
#define link0(c) mem[(c) − 2].lit
#define link1(c) mem[(c) − 3].lit
#define clause_extra 3    /* every clause has a 3-cell preamble */
#define sign_bit #80000000
#define range(c) mem[(c) − 4].lit
#define activ(c) mem[(c) − 5].flt
#define activ_as_lit(c) ((ullng) mem[(c) − 5].lit ≪ 32)
#define learned_supplement 2    /* learned clauses have this many more cells in their preamble */
#define learned_extra (clause_extra + learned_supplement)    /* preamble length */
```

⟨ Type definitions 9 ⟩ +≡

```
  typedef union {
    uint lit;
    float flt;
  } cel;
```

29. The variables are numbered from 1 to n. The literals corresponding to variable k are $k+k$ and $k+k+1$, standing respectively for v and $\bar{v}$ if the kth variable is v.

Several different kinds of data are maintained for each variable: its eight-character *name*; its *activity* score (used to indicate relative desirability for being chosen to make the next decision); its current *value*, which also encodes the level at which the value was set; its current location, *tloc*, in the trail; and its current location, *hloc*, in the heap (which is used to find a maximum activity score). There's also *oldval* and *stamp*, which will be explained later.

```
#define bar(l)  ((l) ⊕ 1)
#define thevar(l)  ((l) ≫ 1)
#define litname(l)  (l) & 1 ? "~" : "", vmem[thevar(l)].name.ch8    /* used in printouts */
#define poslit(v)  ((v) ≪ 1)
#define neglit(v)  (((v) ≪ 1) + 1)
#define unset #ffffffff    /* value when the variable hasn't been assigned */
#define isknown(l)  (vmem[thevar(l)].value ≠ unset)
#define iscontrary(l)  ((vmem[thevar(l)].value ⊕ (l)) & 1)
```

```
⟨ Type definitions 9 ⟩ +≡
  typedef struct {
    octa name;
    double activity;
    uint value;
    int tloc;
    int hloc;    /* is −1 if the variable isn't in the heap */
    uint oldval;
    uint stamp;
    uint filler;    /* not used, but gives octabyte alignment */
  } variable;
```

30. Special data for each literal goes into *lmem*, containing the literal's *reason* for being true, the first clause (if any) that it watches, and the boundaries of its binary implications in *bmem*.

```
⟨ Type definitions 9 ⟩ +≡
  typedef struct {
    int reason;    /* is negative for binary reasons, otherwise clause number */
    uint watch;    /* head of the list of clauses watched by this literal */
    uint bimp_start;    /* where binary implications begin in bmem */
    uint bimp_end;    /* just after where they end (or zero if there aren't any) */
  } literal;
```

31. Here is a subroutine that prints the binary implicant data for a given literal. (Used only when debugging.)

```
⟨ Subroutines 31 ⟩ ≡
  void print_bimp(int l)
  {
    register uint la;
    printf(""O"s"O".8s("O"d)␣->", litname(l), l);
    if (lmem[l].bimp_end) {
      for (la = lmem[l].bimp_start; la < lmem[l].bimp_end; la++)
        printf("␣"O"s"O".8s("O"d)", litname(bmem[la]), bmem[la]);
    }
    printf("\n");
  }
```

See also sections 32, 33, 34, 39, 42, 43, 44, and 71.

This code is used in section 2.

32. Similarly, we can print the numbers of all clauses that l is currently watching.

```
⟨ Subroutines 31 ⟩ +≡
  void print_watches_for(int l)
  {
    register uint c;
    printf(""O"s"O".8s("O"d)␣watched␣in", litname(l), l);
    for (c = lmem[l].watch; c; ) {
      printf("␣"O"u", c);
      if (mem[c].lit ≡ l) c = link0(c);
      else c = link1(c);
    }
    printf("\n");
  }
```

33. And we also sometimes need to see the literals of a given clause.

```
⟨ Subroutines 31 ⟩ +≡
  void print_clause(uint c)
  {
    register int k, l;
    printf(""O"u:", c);
    if (c < clause_extra ∨ c ≥ max_learned) {
      printf("␣clause␣"O"d␣doesn't␣exist!\n", c);
      return;
    }
    for (k = 0; k < size(c); k++) {
      l = mem[c + k].lit;
      if (l < 2 ∨ l > max_lit) {
        printf("␣BAD!\n");
        return;
      }
      printf("␣"O"s"O".8s("O"u)", litname(l), l);
    }
    while (mem[c + k].lit & sign_bit) {
      l = mem[c + k].lit ⊕ sign_bit;
      if (l < 2 ∨ l > max_lit) {
        printf("␣!BAD!\n");
        return;
      }
      printf("␣!"O"s"O".8s("O"u)", litname(l), l);
      k++;
    }
    printf("\n");
  }
```

34. 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 31 ⟩ +≡
  void sanity(int eptr)
  {
    register uint k, l, c, endc, u, v, clauses, watches, vals, llevel;
    ⟨ Check all clauses for spurious data 35 ⟩;
    ⟨ Check the watch lists 36 ⟩;
    ⟨ Check the sanity of the heap 72 ⟩;
    ⟨ Check the trail 37 ⟩;
    ⟨ Check the variables 38 ⟩;
  }
```

35. ⟨ Check all clauses for spurious data 35 ⟩ ≡

```
for (clauses = k = 0, c = clause_extra; c < min_learned; k = c, c = endc + clause_extra) {
  endc = c + size(c);
  clauses++;
  if (link0(c) ≥ max_learned) {
    fprintf(stderr, "bad␣link0("O"u)!\n", c);
    return;
  }
  if (link1(c) ≥ max_learned) {
    fprintf(stderr, "bad␣link1("O"u)!\n", c);
    return;
  }
  if (size(c) < 2) fprintf(stderr, "size("O"u)="O"d!\n", c, size(c));
  for (k = 0; k < size(c); k++)
    if (mem[c + k].lit < 2 ∨ mem[c + k].lit > max_lit)
      fprintf(stderr, "bad␣lit␣"O"d␣of␣"O"d!\n", k, c);
  while (mem[c + k].lit & sign_bit) {
    if (mem[c + k].lit < 2 + sign_bit ∨ mem[c + k].lit > max_lit + sign_bit)
      fprintf(stderr, "bad␣deleted␣lit␣"O"d␣of␣"O"d!\n", k, c);
    k++, endc++;
  }
}
if (c ≠ min_learned) fprintf(stderr, "bad␣last␣unlearned␣clause␣("O"d)!\n", k);
else {
  for (k = 0, c = first_learned; c < max_learned; k = c, c = endc + learned_extra) {
    endc = c + size(c);
    clauses++;
    if (link0(c) ≥ max_learned) {
      fprintf(stderr, "bad␣link0("O"u)!\n", c);
      return;
    }
    if (link1(c) ≥ max_learned) {
      fprintf(stderr, "bad␣link1("O"u)!\n", c);
      return;
    }
    if (size(c) < 2) fprintf(stderr, "size("O"u)="O"d!\n", c, size(c));
    for (k = 0; k < size(c); k++)
      if (mem[c + k].lit < 2 ∨ mem[c + k].lit > max_lit)
        fprintf(stderr, "bad␣lit␣"O"d␣of␣"O"d!\n", k, c);
    while (mem[c + k].lit & sign_bit) {
      if (mem[c + k].lit < 2 + sign_bit ∨ mem[c + k].lit > max_lit + sign_bit)
        fprintf(stderr, "bad␣deleted␣lit␣"O"d␣of␣"O"d!\n", k, c);
      k++, endc++;
    }
  }
  if (c ≠ max_learned) fprintf(stderr, "bad␣last␣learned␣clause␣("O"d)!\n", k);
  if (mem[c − learned_extra].lit) fprintf(stderr, "missing␣zero␣at␣end␣of␣mem!\n");
}
```

This code is used in section 34.

36. In really bad cases this routine will get into a loop. I try to avoid segmentation faults, but not loops.

```
⟨ Check the watch lists 36 ⟩ ≡
  for (watches = 0, l = 2; l ≤ max_lit; l++) {
    for (c = lmem[l].watch; c; ) {
      watches++;
      if (c < clause_extra ∨ c ≥ max_learned) {
        fprintf(stderr, "clause␣"O"u␣in␣watch␣list␣"O"u␣out␣of␣range!\n", c, l);
        return;
      }
      if (mem[c].lit ≡ l) c = link0(c);
      else if (mem[c + 1].lit ≡ l) c = link1(c);
      else {
        fprintf(stderr, "clause␣"O"u␣improperly␣on␣watch␣list␣"O"u!\n", c, l);
        return;
      }
    }
  }
  if (watches ≠ clauses + clauses)
    fprintf(stderr, ""O"u␣clauses␣but␣"O"u␣watches!\n", clauses, watches);
```

This code is used in section 34.

37. ⟨ Check the trail 37 ⟩ ≡

```
for (k = llevel = 0; k < eptr; k++) {
  l = trail[k];
  if (l < 2 ∨ l > max_lit) {
    fprintf(stderr, "bad␣lit␣"O"u␣in␣trail["O"u]!\n", l, k);
    return;
  }
  if (vmem[thevar(l)].tloc ≠ k) fprintf(stderr, ""O"s"O".8s␣has␣bad␣tloc␣("O"d␣not␣"O"d)!\n",
        litname(l), vmem[thevar(l)].tloc, k);
  if (k ≡ leveldat[llevel + 2]) {
    llevel += 2;
    if (lmem[l].reason)
      fprintf(stderr, ""O"s"O".8s("O"u),␣level␣"O"u,␣shouldn't␣have␣reason!\n", litname(l),
          l, llevel ≫ 1);
  } else {
    if (llevel ∧ ¬lmem[l].reason) fprintf(stderr,
          ""O"s"O".8s("O"u),␣level␣"O"u,␣should␣have␣reason!\n", litname(l), l, llevel ≫ 1);
  }
  if (lmem[bar(l)].reason) fprintf(stderr,
        ""O"s"O".8s("O"u),␣level␣"O"u,␣comp␣has␣reason!\n", litname(l), l, llevel ≫ 1);
  if (vmem[thevar(l)].value ≠ llevel + (l & 1))
    fprintf(stderr, ""O"s"O".8s("O"u),␣level␣"O"u,␣has␣bad␣value!\n", litname(l), l, llevel ≫ 1);
  if (llevel) {
    if (lmem[l].reason ≤ 0) {
      if (lmem[l].reason ≡ −1 ∨ lmem[l].reason < −max_lit)
        fprintf(stderr, ""O"s"O".8s("O"u),␣level␣"O"u,␣has␣wrong␣binary␣reason␣("O"u)!\n",
            litname(l), l, llevel ≫ 1, c);
    } else {
      c = lmem[l].reason;
      if (mem[c].lit ≠ l)
        fprintf(stderr, ""O"s"O".8s("O"u),␣level␣"O"u,␣has␣wrong␣reason␣("O"u)!\n",
            litname(l), l, llevel ≫ 1, c);
      u = bar(mem[c + 1].lit);
      if (vmem[thevar(u)].value ≠ llevel + (u & 1))
        fprintf(stderr, ""O"s"O".8s("O"u),␣level␣"O"u,␣has␣bad␣reason␣("O"u)!\n",
            litname(l), l, llevel ≫ 1, c);
    }
  }
}
```

This code is used in section 34.

38. ⟨ Check the variables 38 ⟩ ≡

```
for (vals = 0, v = 1; v ≤ vars; v++) {
  if (vmem[v].value > llevel + 1) {
    if (vmem[v].value ≠ unset) fprintf(stderr, "strange␣val␣"O".8s␣(level␣"O"u)!\n",
          vmem[v].name.ch8, vmem[v].value ≫ 1);
    else if (vmem[v].hloc < 0)
      fprintf(stderr, ""O".8s␣should␣be␣in␣the␣heap!\n", vmem[v].name.ch8);
  } else vals++;
}
if (vals ≠ eptr) fprintf(stderr, "I␣found␣"O"u␣values,␣but␣eptr="O"u!\n", vals, eptr);
```

This code is used in section 34.

39. The *print_stats* subroutine presents a digest of the current goings-on. First it shows the number of literals learned at level zero (`z`). Then it shows recent smoothed-average values of decision depth (`d`), mems per conflict (`m`), propagations per conflict (`p`), resolutions per conflict (`r`), literals per learned clause (`L` and `l`, where the latter is restricted to nontrivial clauses), glucose per learned clause (`g`), and clauses of length six or less per learned clause (`s`), together with the recent agility (`a`). For my own edification I also estimate mems per propagation (`m/p`).

```
#define two_to_the_32 4294967296.0
```

⟨ Subroutines 31 ⟩ +≡

```
  void print_stats(void)
  {
    register double mpc = mems_per_confl, ppc = props_per_confl;
    fprintf(stderr, "z="O"d␣d="O".1f␣t="O".1f␣m="O".1f␣p="O".1f␣m/p="O".1f", leveldat[2],
        (double) depth_per_decision/two_to_the_32, (double) trail_per_decision/two_to_the_32,
        mpc/two_to_the_32, ppc/two_to_the_32, mpc/ppc);
    fprintf(stderr, "␣r="O".1f␣L="O".1f␣l="O".1f␣g="O".1f␣s="O".2f␣a="O".2f\n", (double)
        res_per_confl/two_to_the_32, (double) lits_per_confl/two_to_the_32,
        (double) lits_per_nontriv/two_to_the_32, (double) glucose_per_confl/two_to_the_32, (double)
        short_per_confl/two_to_the_32, (double) agility/two_to_the_32);
  }
```

40. We represent the statistics $\sigma = (x_0+x_1\zeta+x_2\zeta^2+\cdots)/(1+\zeta+\zeta^2+\cdots)$, for various integer quantities x, as 64-bit unsigned integers with 32 bits of fraction. Here x_k denotes the value of x at the k-from-last conflict, and ζ is the damping factor $1-2^{-7}$.

Thus, to update σ with a new value of x at conflict time, we replace it by $\zeta\sigma + 2^{32}x/(1+\zeta+\zeta^2+\cdots) = \sigma - \sigma/2^7 + 2^{25}x$.

⟨ Update the smoothed-average stats after a clause has been learned 40 ⟩ ≡

```
  mems_per_confl += −(mems_per_confl ≫ 7) + ((mems − mems_at_prev_confl) ≪ 25);
  mems_at_prev_confl = mems;
  props_per_confl += −(props_per_confl ≫ 7) + ((ullng) props ≪ 25);
  props = 0;
  res_per_confl += −(res_per_confl ≫ 7) + ((ullng) resols ≪ 25);
  lits_per_confl += −(lits_per_confl ≫ 7) + ((ullng) learned_size ≪ 25);
  if (¬trivial_learning) lits_per_nontriv += −(lits_per_nontriv ≫ 7) + ((ullng) learned_size ≪ 25);
  short_per_confl += −(short_per_confl ≫ 7) + (learned_size > 6 ? 0 : 1 ≪ 25);
  glucose_per_confl += −(glucose_per_confl ≫ 7) + ((ullng) clevels ≪ 25);
```

This code is used in section 102.

41. ⟨ Global variables 4 ⟩ +≡

```
  ullng depth_per_decision;    /* smoothed average of llevel ≫ 1 at decision time */
  ullng trail_per_decision;    /* smoothed average of eptr at decision time */
  ullng mems_per_confl, lits_per_confl, lits_per_nontriv;    /* smoothed averages */
  ullng res_per_confl, glucose_per_confl;    /* more smoothies */
  ullng props_per_confl = two_to_the_32;    /* this one ought to be nonzero */
  uint short_per_confl;    /* smoothed probability of learned clause being short */
  uint agility;    /* smoothed probability of forced flips in value */
  ullng mems_at_prev_confl;    /* mems at the previous update */
  uint props;    /* propagations since the previous update */
```

42. In long runs it's helpful to know how far we've gotten. A numeric code summarizes the histories of literals that appear in the current trail: 0 or 1 means that we're trying to set a variable true or false, as a decision at the beginning of a level; 2 or 3 is similar, but after we've learned that the decision was wrong (hence we've learned a clause that has forced the opposite decision); 4 or 5 is similar, but when the value was forced by the decision at the previous decision node; 6 or 7 is similar, but after we learned that a previous decision forces this one. (In the latter case, the learned clause forced a variable that was not the decision variable at its level.) This code is also used for unit clauses in the input.

A special *history* array is used to provide these base codes (0, 2, 4, or 6). No mems are assessed for maintaining *history*, because it isn't used in any decisions taken by the algorithm; it's purely for diagnostic purposes.

The variable *trail_marker* marks a place in the trail that I'm trying to study. This subroutine inserts a vertical line at that point, so that I can watch where it goes. (Maybe other users might even find it informative some day, who knows?)

Note: These codes are analogous to similar codes in SAT0, SAT0W, SAT10, and SAT11. But they don't really give an easy-to-read picture of progress, as they did in the others, because they don't increase lexicographically in the presence of restarts. Therefore they are displayed only if the user has set *print_state_cutoff* to a positive value, using the command-line parameter `H`.

```
⟨ Subroutines 31 ⟩ +≡
  void print_state(int eptr)
  {
    register uint j, k;
    fprintf(stderr, "␣after␣"O"lld␣mems:", mems);
    if (print_state_cutoff) {
      for (k = 0; k < eptr; k++) {
        if (k ≡ trail_marker) fprintf(stderr, "|");
        fprintf(stderr, ""O"d", history[k] + (trail[k] & 1));
        if (k ≥ print_state_cutoff) {
          fprintf(stderr, "..."); break;
        }
      }
      fprintf(stderr, "\n");
    }
    fprintf(stderr, "␣");
    print_stats();
    fflush(stderr);
  }
```

43. We might also like to see the complete trail, including names and reasons.

```
⟨ Subroutines 31 ⟩ +≡
  void print_trail(int eptr)
  {
    register int k, l;
    for (k = 0; k < eptr; k++) {
      l = trail[k];
      if (k ≥ vars ∨ l < 2 ∨ l > max_lit) return;
      fprintf(stderr, ""O"d:␣"O"d␣"O"d␣"O"s"O".8s("O"d)", k, history[k] + (l & 1),
          vmem[thevar(l)].value ≫ 1, litname(l), l);
      if (lmem[l].reason > 0) {
        if ((vmem[thevar(l)].value ≫ 1) ∨ lmem[l].reason < min_learned)
          fprintf(stderr, "␣#"O"u\n", lmem[l].reason);
        else fprintf(stderr, "␣learned\n");     /* learned at root level */
      } else if (lmem[l].reason < 0) fprintf(stderr, "␣<-␣"O"s"O".8s\n", litname(−lmem[l].reason));
      else fprintf(stderr, "\n");
    }
  }
```

44. Here's a diagnostic routine that runs through all the nonbinary, nonlearned clauses, printing any that are unsatisfied with respect to the current partial assignment of values to variables.

```
⟨ Subroutines 31 ⟩ +≡
  void print_unsat(void)
  {
    register int c, endc, k, l;
    for (c = clause_extra; c < min_learned; c = endc + clause_extra) {
      endc = c + size(c);
      for (k = endc − 1; k ≥ c; k−−) {
        l = mem[k].lit;
        if (isknown(l) ∧ ¬iscontrary(l)) break;
      }
      if (k < c) {     /* clause c not satisfied */
        fprintf(stderr, ""O"d:", c);
        for (k = 0; k < size(c); k++) {
          l = mem[c + k].lit;
          if (¬isknown(l)) fprintf(stderr, "␣"O"s"O".8s", litname(l));
        }
        fprintf(stderr, "␣|");     /* the remaining literals are false */
        for (k = 0; k < size(c); k++) {
          l = mem[c + k].lit;
          if (isknown(l)) fprintf(stderr, "␣"O"s"O".8s", litname(l));
        }
        fprintf(stderr, "\n");
      }
      while (mem[endc].lit & sign_bit) endc++;
    }
  }
```

45. 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 45 ⟩ ≡
  ⟨ Allocate vmem and heap 46 ⟩;
  if (polarity_infile) ⟨ Initialize the heap from a file 79 ⟩
  else ⟨ Initialize the heap randomly 78 ⟩;
  ⟨ Allocate the other main arrays 47 ⟩;
  ⟨ Copy all the temporary cells to the mem and bmem and trail arrays in proper format 49 ⟩;
  ⟨ Copy all the temporary variable nodes to the vmem array in proper format 54 ⟩;
  ⟨ Check consistency 55 ⟩;
  ⟨ Allocate the auxiliary arrays 57 ⟩;
```

This code is used in section 2.

46.

```
⟨ Allocate vmem and heap 46 ⟩ ≡
  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);
  for (k = 1; k ≤ vars; k++) o, vmem[k].value = unset, vmem[k].tloc = −1;
  heap = (uint *) malloc(vars * sizeof(uint));
  if (¬heap) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣heap␣array!\n");
    exit(−11);
  }
  bytes += vars * sizeof(uint);
```

This code is used in section 45.

47. ⟨ Allocate the other main arrays 47 ⟩ ≡

```
free(buf); free(hash);     /* a tiny gesture to make a little room */
⟨ Figure out how big mem ought to be 48 ⟩;
mem = (cel *) malloc(memsize * sizeof(cel));
if (¬mem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣big␣mem␣array!\n");
  exit(−10);
}
bytes += max_cells_used * sizeof(cel);
max_lit = vars + vars + 1;
lmem = (literal *) malloc((max_lit + 1) * sizeof(literal));
if (¬lmem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣lmem␣array!\n");
  exit(−13);
}
bytes += (max_lit + 1) * sizeof(literal);
trail = (uint *) malloc(vars * sizeof(uint));
if (¬trail) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣trail␣array!\n");
  exit(−14);
}
bytes += vars * sizeof(uint);
```

See also section 56.

This code is used in section 45.

48. The *mem* array will contain $2^k - 1 < 2^{31}$ cells of four bytes each, where k is the parameter *memk_max*; this parameter is *memk_max_default* (currently 26) by default, and changeable by the user via `m` on the command line. (Apology: This program is for my own use in experiments, so I haven't bothered to give it a more user-friendly interface.)

It will begin with data for all clauses of length 3 or more; then come the learned clauses, which have slightly longer preambles. During the initialization, some of the eventual space for learned clauses is used temporarily to hold the binary clause information.

We will record in *bytes* and *max_cells_used* only the number of cells actually utilized; this at least gives the user some clue about how big `m` should be.

```
#define memk_max_default 26  /* allow 64 million cells in mem by default */
⟨ Figure out how big mem ought to be 48 ⟩ ≡
  {
    ullng proto_memsize = (clauses − unaries − binaries) ∗ clause_extra + (cells − unaries − 2 ∗ binaries) +
        clause_extra;
    min_learned = proto_memsize;
    proto_memsize += 2 ∗ binaries + learned_supplement;
    if (proto_memsize ≥ #80000000) {
      fprintf(stderr, "Sorry,␣I␣can't␣handle␣"O"llu␣cells␣(2^31␣is␣my␣limit)!\n",
          proto_memsize);
      exit(−665);
    }
    max_cells_used = proto_memsize − learned_supplement + 2;
    first_learned = max_learned = min_learned + learned_supplement;
    memsize = 1 ≪ memk_max;
    if (max_cells_used > memsize) {
      fprintf(stderr, "Immediate␣memory␣overflow␣(memsize="O"u<"O"u),␣please␣increase␣m!\n",
          memsize, max_cells_used);
      exit(−666);
    }
    if (verbose & show_details) fprintf(stderr, "(learned␣clauses␣begin␣at␣"O"u)\n", first_learned);
  }
```

This code is used in section 47.

49. Binary data is copied temporarily into cells starting at *min_learned* + 2. (The '+ 2' is needed because the final clause processed is input with *c* = *min_learned*.)

```
⟨ Copy all the temporary cells to the mem and bmem and trail arrays in proper format 49 ⟩ ≡
  eptr = 0;   /* empty the trail in preparation for unit clauses */
  for (l = 2; l ≤ max_lit; l++) oo, lmem[l].reason = lmem[l].watch = lmem[l].bimp_end = 0;
  for (c = clause_extra, j = clauses, jj = min_learned + 2; j; j−−) {
    k = 0;
    ⟨ Insert the cells for the literals of clause c 50 ⟩;
    if (k ≤ 2) ⟨ Do special things for unary and binary clauses 51 ⟩
    else {
      o, size(c) = k;
      l = mem[c].lit;
      ooo, link0(c) = lmem[l].watch, lmem[l].watch = c;
      l = mem[c + 1].lit;
      ooo, link1(c) = lmem[l].watch, lmem[l].watch = c;
      c += k + clause_extra;
    }
  }
  o, mem[c − clause_extra].lit = 0;   /* put zero at end of mem */
  if (c ≠ min_learned) {
    fprintf(stderr, "Oh␣oh,␣I␣didn't␣load␣the␣correct␣number␣of␣cells␣("O"u:"O"u)!\n", c,
        min_learned);
    exit(−17);
  }
  if (jj ≠ max_cells_used) {
    fprintf(stderr, "Oh␣oh,␣I␣miscounted␣binaries␣somehow␣("O"u:"O"u)!\n", jj, max_cells_used);
    exit(−18);
  }
  ⟨ Reformat the binary implications 53 ⟩;
```

This code is used in section 45.

50. 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 50 ⟩ ≡
  for (i = 0; i < 2; ) {
    ⟨ Move cur_cell backward to the previous cell 23 ⟩;
    i = hack_out(*cur_cell);
    p = hack_clean(*cur_cell)→serial;
    p += p + (i & 1) + 2;
    o, mem[c + k++].lit = p;
  }
```

This code is used in section 49.

51. ⟨ Do special things for unary and binary clauses 51 ⟩ ≡

```
{
  if (k < 2) ⟨ Define mem[c].lit at level 0 52 ⟩
  else {
    l = mem[c].lit, ll = mem[c + 1].lit;      /* no mem charged for these */
    oo, lmem[bar(l)].bimp_end++;
    oo, lmem[bar(ll)].bimp_end++;
    o, mem[jj].lit = l, mem[jj + 1].lit = ll, jj += 2;      /* copy the literals temporarily */
  }
}
```

This code is used in section 49.

52. We have to watch for degenerate cases: Unit clauses in the input might be duplicated or contradictory.

⟨ Define mem[c].lit at level 0 52 ⟩ ≡

```
{
  l = mem[c].lit, v = thevar(l);
  if (o, vmem[v].value ≡ unset) {
    o, vmem[v].value = l & 1, vmem[v].tloc = eptr;
    o, history[eptr] = 6, trail[eptr++] = l;
  } else if (vmem[v].value ≠ (l & 1)) goto unsat;
}
```

This code is used in section 51.

53. ⟨ Reformat the binary implications 53 ⟩ ≡

```
for (l = 2, jj = 0; l ≤ max_lit; l++) {
  o, k = lmem[l].bimp_end;
  if (k) o, lmem[l].bimp_start = lmem[l].bimp_end = jj, jj += k;
}
for (jj = min_learned + 2, j = binaries; j; j--) {
  o, l = mem[jj].lit, ll = mem[jj + 1].lit, jj += 2;
  ooo, k = lmem[bar(l)].bimp_end, bmem[k] = ll, lmem[bar(l)].bimp_end = k + 1;
  ooo, k = lmem[bar(ll)].bimp_end, bmem[k] = l, lmem[bar(ll)].bimp_end = k + 1;
}
```

This code is used in section 49.

54. ⟨ Copy all the temporary variable nodes to the *vmem* array in proper format 54 ⟩ ≡

```
for (c = vars; c; c--) {
  ⟨ Move cur_tmp_var backward to the previous temporary variable 24 ⟩;
  o, vmem[c].name.lng = cur_tmp_var→name.lng;
  o, vmem[c].stamp = 0;
}
```

This code is used in section 45.

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

```
⟨ Check consistency 55 ⟩ ≡
  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 45.

56. A few arrays aren't really of "main" importance, but we need to allocate them before incorporating the clause information into *mem*.

```
⟨ Allocate the other main arrays 47 ⟩ +≡
  bmem = (uint ∗) malloc(binaries ∗ 2 ∗ sizeof(uint));
  if (¬bmem) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣bmem␣array!\n");
    exit(−16);
  }
  bytes += binaries ∗ 2 ∗ sizeof(uint);
  history = (char ∗) malloc(vars ∗ sizeof(char));
  if (¬history) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣history␣array!\n");
    exit(−15);
  }
  bytes += vars ∗ sizeof(char);
```

57. The other arrays can perhaps make use of the memory chunks that are freed while we're reformatting the clause and variable data.

```
⟨ Allocate the auxiliary arrays 57 ⟩ ≡
  leveldat = (int ∗) malloc(vars ∗ 2 ∗ sizeof(int));
  if (¬leveldat) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣leveldat␣array!\n");
    exit(−16);
  }
  bytes += vars ∗ 2 ∗ sizeof(int);
```

See also sections 89, 96, 109, and 116.

This code is used in section 45.

58. Forcing. This program spends most of its time adding literals to the current trail when they are forced to be true because of earlier items on the trail.

The "inner loop" of the forcing phase tries to derive the consequences of literal *l* that follow from binary clauses in the input. At this point *l* is a literal in the trail. Furthermore *lat* = *lmem*[*l*].*bimp_end* has just been fetched, and it's known to be nonzero.

(I apologize for the awkward interface between this loop and its context. Maybe I shouldn't worry so much about saving mems in the inner loop. But that's the kind of guy I am.)

```
⟨Propagate binary implications of l; goto confl if a conflict arises 58⟩ ≡
  for (lbptr = eptr; ; ) {
    for (la = lmem[l].bimp_start; la < lat; la++) {
      o, ll = bmem[la];
      if (o, isknown(ll)) {
        if (iscontrary(ll)) {
          props++;
          ⟨Deal with a binary conflict 66⟩;
        }
      } else {
        props++;
        if (verbose & show_details)
          fprintf(stderr, "␣"O"s"O".8s␣->␣"O"s"O".8s\n", litname(l), litname(ll));
        o, history[eptr] = 4, trail[eptr] = ll;
        o, lmem[ll].reason = −l;
        o, vmem[thevar(ll)].value = llevel + (ll & 1), vmem[thevar(ll)].tloc = eptr++;
        agility −= agility ≫ 13;      /* use the damping factor $1 - 2^{-13}$ */
        if (o, (vmem[thevar(ll)].oldval + ll) & 1) agility += 1 ≪ 19;
      }
    }
    while (1) {
      if (lbptr ≡ eptr) {
        l = 0; break;      /* kludge for breaking out of two loops */
      }
      o, l = trail[lbptr++];
      o, lat = lmem[l].bimp_end;
      if (lat) break;
    }
    if (l ≡ 0) break;
  }
```

This code is used in sections 65 and 127.

59.
```
⟨Global variables 4⟩ +≡
  uint lt;      /* literal on the trail */
  uint lat;      /* its bimp_end */
  uint wa, next_wa;      /* a clause in its watch list */
```

60. The "next to inner loop" of forcing looks for nonbinary clauses that have at most one literal that isn't false.

At this point we're looking at a literal *lt* that was placed on the trail. Its binary implications were found at that time; now we want to examine the more complex ones, by looking at all clauses on the watch list of *bar*(*lt*).

While doing this, we swap the first two literals, if necessary, so that *bar*(*lt*) is the second one watching.

Counting of mems is a bit tricky here: If c is the address of a clause, either $mem[c].lit$ and $mem[c+1].lit$ are in the same octabyte, or $link0(c)$ and $link1(c)$, but not both. So we make three memory references when we're reading from or storing into all four items.

```
⟨ Propagate nonbinary implications of lt; goto confl if there's a conflict 60 ⟩ ≡
  o, wa = lmem[bar(lt)].watch;
  if (wa) {
    for (q = 0; wa; wa = next_wa) {
      o, ll = mem[wa].lit;
      if (ll ≡ bar(lt)) {
        o, ll = mem[wa + 1].lit;
        oo, mem[wa].lit = ll, mem[wa + 1].lit = bar(lt);
        o, next_wa = link0(wa);
        o, link0(wa) = link1(wa), link1(wa) = next_wa;
      } else o, next_wa = link1(wa);
      ⟨ If clause wa is satisfied by ll, keep wa on the watch list and continue 63 ⟩;
      for (o, s = size(wa), j = wa + s − 1; j > wa + 1; j−−) {
        o, l = mem[j].lit;
        if (o, ¬isknown(l) ∨ ¬iscontrary(l)) break;
        if (vmem[thevar(l)].value < 2 ∧ llevel) ⟨ Delete l from clause wa 61 ⟩;
      }
      if (j > wa + 1) ⟨ Swap wa to the watch list of l and continue 62 ⟩;
      ⟨ Keep wa on the watch list 64 ⟩;
      ⟨ Force a new value, if appropriate, or goto confl 65 ⟩;
    }
    ⟨ Keep wa on the watch list 64 ⟩;     /* this terminates the watch list with 0 */
  }
```

This code is used in section 127.

61. The literal l is known to be permanently false, so we seize this opportunity to remove it from the active memory. (Such deletions will be important later, when we attempt to do "on-the-fly subsumption.")

At this point, s is the current size of clause *wa*.

```
⟨ Delete l from clause wa 61 ⟩ ≡
  {
    o, size(wa) = −−s;
    if (j ≠ wa + s) oo, mem[j].lit = mem[wa + s].lit;     /* swap past end of clause */
    o, mem[wa + s].lit = l + sign_bit;
  }
```

This code is used in section 60.

62. ⟨ Swap *wa* to the watch list of *l* and **continue** 62 ⟩ ≡

```
{
  if (verbose & show_watches) fprintf(stderr, "␣"O"s"O".8s␣watched␣in␣"O"d\n", litname(l), wa);
  oo, mem[wa + 1].lit = l, mem[j].lit = bar(lt);
  o, link1(wa) = lmem[l].watch;
  o, lmem[l].watch = wa;
  continue;
}
```

This code is used in section 60.

63. We're looking at clause *wa*, which is watched by *bar*(*lt*) and *ll*, where *lt* is known to be true (at least with respect to the decisions currently in force).

Consider what happens in the case that literal *ll* is also true, thereby satisfying clause *wa*: We can continue with *wa* on the watch list of *bar*(*lt*), even though *bar*(*lt*) is false, because this clause will remain satisfied until backtracking makes *lt* undefined.

⟨ If clause *wa* is satisfied by *ll*, keep *wa* on the watch list and **continue** 63 ⟩ ≡

```
if ((o, isknown(ll)) ∧ ¬iscontrary(ll)) {
  ⟨ Keep wa on the watch list 64 ⟩;
  continue;
}
```

This code is used in section 60.

64. A satisfied clause *wa* can be watched by a false literal, as noted above. Furthermore, during full runs we allow clauses to become entirely false; in such cases both watchers must have become false on the maximum level of all literals in *wa*.

⟨ Keep *wa* on the watch list 64 ⟩ ≡

```
if (q ≡ 0) o, lmem[bar(lt)].watch = wa;
else o, link1(q) = wa;
q = wa;
```

This code is used in sections 60 and 63.

65. Well, all literals of clause *wa*, except possibly the first one, did in fact turn out to be false. That first literal is what the program calls *ll*, and we've already verified that *ll* isn't true.

If *ll* is false, we've run into a conflict. Otherwise we will force *ll* to be true at the current decision level.

```
⟨ Force a new value, if appropriate, or goto confl 65 ⟩ ≡
  props++;
  if (isknown(ll)) ⟨ Deal with a nonbinary conflict 67 ⟩
  else {
    if (verbose & show_details) fprintf(stderr, "␣"O"s"O".8s␣from␣"O"d\n", litname(ll), wa);
    o, history[eptr] = 4, trail[eptr] = ll;
    o, vmem[thevar(ll)].tloc = eptr++;
    vmem[thevar(ll)].value = llevel + (ll & 1);
    agility −= agility ≫ 13;      /* use the damping factor $1 - 2^{-13}$ */
    if (o, (vmem[thevar(ll)].oldval + ll) & 1) agility += 1 ≪ 19;
    o, lmem[ll].reason = wa;
    o, lat = lmem[ll].bimp_end;
    if (lat) {
      l = ll;
      ⟨ Propagate binary implications of l; goto confl if a conflict arises 58 ⟩;
    }
  }
```

This code is used in section 60.

66. In the case considered here, a conflict has arisen from the binary clause $\bar{u} \vee \bar{v}$, where $u = l$ and $\bar{v} = ll$. This clause is represented only implicitly in the *bmem* array, not explicitly in *mem*.

```
⟨ Deal with a binary conflict 66 ⟩ ≡
  {
    if (verbose & show_details)
      fprintf(stderr, "␣"O"s"O".8s␣->␣"O"s"O".8s␣#\n", litname(l), litname(ll));
    if (full_run ∧ llevel) ⟨ Record a binary conflict 68 ⟩
    else {
      c = −l;
      goto confl;
    }
  }
```

This code is used in section 58.

67.

```
⟨ Deal with a nonbinary conflict 67 ⟩ ≡
  {
    if (verbose & show_details) fprintf(stderr, "␣"O"s"O".8s␣from␣"O"d␣#\n", litname(ll), wa);
    if (full_run ∧ llevel) ⟨ Record a nonbinary conflict 69 ⟩
    else {
      c = wa;
      goto confl;
    }
  }
```

This code is used in section 65.

68. During a "full run," we continue to propagate after finding a conflict. We remember only the first one, at any given level, putting its clause number into *leveldat*[*llevel* + 1].

The "clause number" of a binary clause is considered to be $-l$, and the value of *bar*(*ll*) is saved in odd-numbered entries of the *conflictdat* array.

A stack of levels on which conflicts have occurred is maintained in the even-numbered entries of *conflictdat*. The top of this stack is called *conflict_level*.

```
⟨ Record a binary conflict 68 ⟩ ≡
  {
    if (¬conflict_seen) {
      conflict_seen = 1;
      o, leveldat[llevel + 1] = −l;
      o, conflictdat[llevel + 1] = ll;
      conflictdat[llevel] = conflict_level, conflict_level = llevel;
    }
  }
```

This code is used in section 66.

69.

```
⟨ Record a nonbinary conflict 69 ⟩ ≡
  {
    if (¬conflict_seen) {
      conflict_seen = 1;
      o, leveldat[llevel + 1] = wa;
      o, conflictdat[llevel] = conflict_level, conflict_level = llevel;
    }
  }
```

This code is used in section 67.

70. Activity scores. Experience shows that it's usually a good idea to branch on a variable that has participated recently in the construction of conflict clauses. More precisely, we try to maximize "activity," where the activity of variable v is proportional to the sum of $\{\rho^t \mid v$ participates in the tth-from-last conflict$\}$; here ρ is a parameter representing the rate of decay by which influential activity decays with time. (Users can change the default ratio $\rho = .95$ if desired.)

There's a simple way to implement this quantity, because activity is also proportional to the sum of $\{\rho^{-t} \mid v$ participates in the tth conflict$\}$; that sum counts forward in time rather than backward. We can therefore get proper results by adding *var_bump* to v's score whenever v participates in a conflict, and then dividing *var_bump* by ρ after each conflict.

If the activity scores computed in this way become too large, we simply scale them back, so that relative ratios are preserved.

Incidentally, the somewhat mysterious acronym VSIDS, which stands for "variable state independent decaying sum," is often used by insiders to describe this aspect of a CDCL solver. The activity scoring mechanism adopted here, due to Niklas Eén in the 2005 version of MiniSAT, was inspired by a similar but less effective VSIDS scheme originally introduced by Matthew Moskewitz in the CHAFF solver.

```
⟨ Bump l's activity 70 ⟩ ≡
  v = thevar(l);
  o, av = vmem[v].activity + var_bump;
  o, vmem[v].activity = av;
  if (av ≥ 1 · 10^100) ⟨ Rescale all variable activities 83 ⟩;
  o, h = vmem[v].hloc;
  if (h > 0) ⟨ Sift v up in the heap 73 ⟩;
```

This code is used in sections 87, 88, and 95.

71. The heap contains *hn* variables, ordered in such a way that $vmem[x].activity \ge vmem[y].activity$ whenever $x = heap[h]$ and $y = heap[2*h+1]$ or $y = heap[2*h+2]$. In particular, $heap[0]$ always names a variable of maximum activity.

```
⟨ Subroutines 31 ⟩ +≡
  void print_heap(void)
  {
    register int k;
    for (k = 0; k < hn; k++) {
      fprintf(stderr, ""O"d:␣"O".8s␣"O"e\n", k, vmem[heap[k]].name.ch8, vmem[heap[k]].activity);
    }
  }
```

72. ⟨ Check the sanity of the heap 72 ⟩ ≡

```
for (k = 1; k ≤ vars; k++) {
  if (vmem[k].hloc ≥ hn)
    fprintf(stderr, "hloc␣of␣"O".8s␣exceeds␣"O"d!\n", vmem[k].name.ch8, hn − 1);
  else if (vmem[k].hloc ≥ 0 ∧ heap[vmem[k].hloc] ≠ k)
    fprintf(stderr, "hloc␣of␣"O".8s␣errs!\n", vmem[k].name.ch8);
}
for (k = 0; k < hn; k++) {
  v = heap[k];
  if (v ≤ 0 ∨ v > vars) fprintf(stderr, "heap["O"d]="O"d!\n", k, v);
  else if (k) {
    u = heap[(k − 1) ≫ 1];
    if (u > 0 ∧ u ≤ vars ∧ vmem[u].activity < vmem[v].activity)
      fprintf(stderr, "heap["O"d]act<heap["O"d]act!\n", (k − 1) ≫ 1, k);
  }
}
```

This code is used in section 34.

73. At this point we assume that $av = vmem[v].activity$.

⟨ Sift v up in the heap 73 ⟩ ≡

```
{
  hp = (h − 1) ≫ 1;      /* the “parent” of position h */
  o, u = heap[hp];
  if (o, vmem[u].activity < av) {
    while (1) {
      o, heap[h] = u;
      o, vmem[u].hloc = h;
      h = hp;
      if (h ≡ 0) break;
      hp = (h − 1) ≫ 1;
      o, u = heap[hp];
      if (o, vmem[u].activity ≥ av) break;
    }
    o, heap[h] = v;
    o, vmem[v].hloc = h;
    j = 1;
  }
}
```

This code is used in sections 70 and 74.

74. ⟨ Put v into the heap 74 ⟩ ≡

```
{
  o, av = vmem[v].activity;
  h = hn++, j = 0;
  if (h > 0) ⟨ Sift v up in the heap 73 ⟩;
  if (j ≡ 0) oo, heap[h] = v, vmem[v].hloc = h;
}
```

This code is used in section 128.

75. With probability *rand_prob*, we select a variable from the heap at random; this policy is a heuristic designed to avoid getting into a rut. Otherwise we take the variable at the top, because that variable has maximum activity.

Variables in the heap often have known values, however. If our first choice was one of them, we keep trying from the top, until we find *vmem*[*v*].*value* ≡ *unset*.

The variable's polarity is taken from *vmem*[*v*].*oldval*, because good values from prior experiments tend to remain good.

As in other programs of this family, the cost of generating 31 random bits is four mems.

```
#define two_to_the_31 ((unsigned long) #80000000)
⟨ Choose the next decision literal, l 75 ⟩ ≡
  if (rand_prob_thresh) {
    mems += 4, h = gb_next_rand();
    if (h < rand_prob_thresh) {
      ⟨ Set h to a random integer less than hn 76 ⟩
      o, v = heap[h];
      if (o, vmem[v].value ≠ unset) h = 0;
    } else h = 0;
  } else h = 0;
  if (h ≡ 0) {
    while (1) {
      o, v = heap[0];
      ⟨ Delete v from the heap 77 ⟩;
      if (o, vmem[v].value ≡ unset) break;
    }
  }
  o, l = poslit(v) + (vmem[v].oldval & 1);
```

This code is used in section 124.

76.
```
⟨ Set h to a random integer less than hn 76 ⟩ ≡
  {
    register unsigned long t = two_to_the_31 − (two_to_the_31 mod hn);
    register long r;
    do {
      mems += 4, r = gb_next_rand();
    } while (t ≤ (unsigned long) r);
    h = r mod hn;
  }
```

This code is used in sections 75 and 78.

77. Here we assume that $v = heap[0]$.

```
⟨ Delete v from the heap 77 ⟩ ≡
  o, vmem[v].hloc = −1;
  if (−−hn) {
    o, u = heap[hn];      /* we'll move u into the "hole" at position 0 */
    o, au = vmem[u].activity;
    for (h = 0, hp = 1; hp < hn; h = hp, hp = h + h + 1) {
      oo, av = vmem[heap[hp]].activity;
      if (hp + 1 < hn ∧ (oo, vmem[heap[hp + 1]].activity > av)) hp++, av = vmem[heap[hp]].activity;
      if (au ≥ av) break;
      o, heap[h] = heap[hp];
      o, vmem[heap[hp]].hloc = h;
    }
    o, heap[h] = u;
    o, vmem[u].hloc = h;
  }
```

This code is used in sections 75 and 137.

78. At the very beginning, all activity scores are zero. We'll permute the variables randomly in *heap*, for the sake of variety.

```
⟨ Initialize the heap randomly 78 ⟩ ≡
  {
    if (true_prob ≥ 1.0) true_prob_thresh = #80000000;
    else true_prob_thresh = (int)(true_prob ∗ 2147483648.0);
    for (k = 1; k ≤ vars; k++) o, heap[k − 1] = k;
    for (hn = vars; hn > 1; ) {
      ⟨ Set h to a random integer less than hn 76 ⟩;
      hn−−;
      if (h ≠ hn) {
        o, k = heap[h];
        ooo, heap[h] = heap[hn], heap[hn] = k;
      }
    }
    for (h = 0; h < vars; h++) {
      o, v = heap[h];
      o, vmem[v].hloc = h;
      if (true_prob_thresh ∧ (mems += 4, gb_next_rand() < true_prob_thresh)) vmem[v].oldval = 0;
      else vmem[v].oldval = 1;
      o, vmem[v].activity = 0.0;
    }
    hn = vars;
  }
```

This code is used in section 45.

79. Literals that occur in *polarity_infile* must be separated by whitespace, but they can appear on any number of lines. If the literal isn't in the hash table, we ignore it. (Perhaps a preprocessor has made this literal obsolete.)

```
⟨ Initialize the heap from a file 79 ⟩ ≡
  {
    if (true_prob ≥ 1.0) true_prob_thresh = #80000000;
    else true_prob_thresh = (int)(true_prob ∗ 2147483648.0);
    for (q = 0; ; ) {
      register tmp_var ∗p;
      if (fscanf(polarity_infile, ""O"s", buf) ≠ 1) break;
      if (buf[0] ≡ '˜') i = j = 1;
      else i = j = 0;
      ⟨ Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 19 ⟩;
      for (p = hash[h]; p; p = p→next)
        if (p→name.lng ≡ cur_tmp_var→name.lng) break;
      if (p) {
        v = p→serial + 1;
        o, vmem[v].oldval = i, vmem[v].hloc = q;
        o, heap[q] = v;
        o, vmem[v].activity = (vars − q)/(double) vars;
        o, vmem[v].tloc = 0;
        q++;
      }
    }
    for (v = 0; q < vars; q++) {
      while (o, vmem[++v].tloc ≡ 0) ;    /∗ bypass variables already seen ∗/
      vmem[v].hloc = q;
      if (true_prob_thresh ∧ (mems += 4, gb_next_rand() < true_prob_thresh)) vmem[v].oldval = 0;
      else vmem[v].oldval = 1;
      o, heap[q] = v;
    }
    hn = vars;
  }
```

This code is used in section 45.

80.
```
⟨ Global variables 4 ⟩ +≡
  double var_bump = 1.0;
  float clause_bump = 1.0;
  double var_bump_factor;    /∗ reciprocal of var_rho ∗/
  float clause_bump_factor;    /∗ reciprocal of clause_rho ∗/
```

81. Learned clauses also have activity scores. They aren't used as heavily as the scores for variables; we look at them only when deciding what clauses to keep after too many learned clauses have accumulated.

```
⟨ Bump c's activity 81 ⟩ ≡
  {
    float ac;
    o, ac = activ(c) + clause_bump;
    o, activ(c) = ac;
    if (ac ≥ 1 · 10^20) ⟨ Rescale all clause activities 84 ⟩;
  }
```

This code is used in sections 87 and 93.

82. ⟨ Bump the bumps 82 ⟩ ≡

```
var_bump ∗= var_bump_factor;
clause_bump ∗= clause_bump_factor;
```

This code is used in sections 125 and 133.

83. When a nonzero activity is rescaled, we are careful to keep it nonzero so that a variable once active will not take second place to a totally inactive variable. (I doubt if this is terrifically important, but Niklas Eén told me that he recommends it.)

```
#define tiny 2.225073858507201383 · 10^−308
        /∗ 2^−1022, the smallest positive nondenormal double ∗/
```

⟨ Rescale all variable activities 83 ⟩ ≡

```
{
  register int v;
  register double av;
  for (v = 1; v ≤ vars; v++) {
    o, av = vmem[v].activity;
    if (av) o, vmem[v].activity = (av ∗ 1 · 10^−100 < tiny ? tiny : av ∗ 1 · 10^−100);
  }
  var_bump ∗= 1 · 10^−100;
}
```

This code is used in section 70.

84.

```
#define single_tiny 1.1754943508222875080 · 10^−38
        /∗ 2^−126, the smallest positive nondenormal float ∗/
```

⟨ Rescale all clause activities 84 ⟩ ≡

```
{
  register int cc, endc;
  for (cc = first_learned; cc < max_learned; cc = endc + learned_extra) {
    o, endc = cc + size(cc);
    o, ac = activ(cc);
    if (ac) o, activ(cc) = (ac ∗ 1 · 10^−20 < single_tiny ? single_tiny : ac ∗ 1 · 10^−20);
    while (o, mem[endc].lit & sign_bit) endc++;
  }
  clause_bump ∗= 1 · 10^−20;
}
```

This code is used in section 81.

85. Learning from a conflict. A conflict arises when some clause is found to have no true literals at the current level. This program relies on a technique for avoiding such a conflict in the future, by creating a new clause that is worth learning. Our current goal is to implement (and thereby to understand) that technique.

Let's say that a literal is "new" if it has become true or false at the current decision level; otherwise it is "old." A conflict must contain at least two new literals, because we don't start a new level until every unsatisfied clause is watched by two unassigned literals.

(Hedge: In a "full run" we march boldly into deeper levels after finding conflicts; and in such cases the conflict clauses of level d are watched by two literals that are false at level d. However, even in this case, every unsatisfied clause that could lead to a conflict at a deeper level is watched by two unassigned literals.)

Suppose all literals of c are false. If $\bar{l} \in c$ and c' is the reason for l, we can resolve c with c' to get a new clause c''. This clause c'' is obtained from c by deleting $\bar{l}$ and then inserting $\bar{l}'$ for all l' such that $l \succ l'$. (Indeed, when introducing the method of conflict-driven clause learning above, we defined this direct dependency relation by saying that $l \succ l'$ if and only if $\bar{l}'$ appears in the reason for l.) Notice that all of the literals that belong to c'' are false; hence c'', like c, represents a conflict.

By starting with a conflict clause c and repeatedly resolving away its rightmost literal, using the ordering of the trail, we'll eventually obtain a clause c_0 that has only one new literal. And if c_0 was derived by resolving with other clauses $c_1, \ldots, c_k$, the old literals of c_0 will be the old literals of c, $c_1, \ldots, c_k$.

We could now learn the clause c_0, and return to decision level d, the maximum of the levels of c_0's old literals. (Its new literal will now be forced false at that level.)

Actually, we'll try to simplify c_0 before learning it, by removing some of its old literals if they are redundant. But that's another story, which we can safely postpone until later. The main idea is this: Starting with a conflict clause c, containing two or more new literals, we boil it down to a clause c_0 that contains only one. Then we can resume at a previous level.

86. So much for theory; let's proceed to practice. We can use the *stamp* field to identify literals that appear in the conflict clause c, or in the clauses derived from c as we compute c_0: A variable's *stamp* will equal *curstamp* if and only if we have just marked it. At this point *llevel* > 0.

```
⟨ Deal with the conflict clause c 86 ⟩ ≡
  oldptr = jumplev = xnew = clevels = resols = 0;
  ⟨ Bump curstamp to a new value 91 ⟩;
  if (verbose & show_gory_details) fprintf(stderr, "Preparing␣to␣learn");
  if (c < 0) ⟨ Initialize a binary conflict 88 ⟩
  else ⟨ Initialize a nonbinary conflict 87 ⟩;
  ⟨ Reduce xnew to zero 92 ⟩;
  while (1) {
    o, l = trail[tl−−];
    if (o, vmem[thevar(l)].stamp ≡ curstamp) break;
  }
  lll = bar(l);      /* lll will complete the learned clause */
  if (verbose & show_gory_details) fprintf(stderr, "␣"O"s"O".8s\n", litname(lll));
```

This code is used in section 125.

87. ⟨ Initialize a nonbinary conflict 87 ⟩ ≡

```
{
  o, l = bar(mem[c].lit);
  o, tl = vmem[thevar(l)].tloc;
  o, vmem[thevar(l)].stamp = curstamp;
  ⟨ Bump l's activity 70 ⟩;
  if (c ≥ first_learned) ⟨ Bump c's activity 81 ⟩;
  for (o, s = size(c), k = c + s − 1; k > c; k−−) {
    o, l = bar(mem[k].lit);
    j = vmem[thevar(l)].tloc;      /* mem will be charged when fetching value */
    if (j > tl) tl = j;
    ⟨ Stamp l as part of the conflict clause milieu 95 ⟩;
  }
}
```

This code is used in section 86.

88. Here the conflict is that l implies ll, where literal $l = -c$ is true but literal ll is false.

⟨ Initialize a binary conflict 88 ⟩ ≡

```
{
  o, tl = vmem[thevar(ll)].tloc;
  o, vmem[thevar(ll)].stamp = curstamp;
  l = ll;
  ⟨ Bump l's activity 70 ⟩;
  l = −c;
  if (o, vmem[thevar(l)].tloc > tl) tl = vmem[thevar(l)].tloc;
  o, vmem[thevar(l)].stamp = curstamp;
  ⟨ Bump l's activity 70 ⟩;
  xnew = 1;
}
```

This code is used in section 86.

89. ⟨ Allocate the auxiliary arrays 57 ⟩ +≡

```
learn = (uint *) malloc(vars * sizeof(uint));
if (¬learn) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣learn␣array!\n");
  exit(−16);
}
bytes += vars * sizeof(uint);
```

90. ⟨ Global variables 4 ⟩ +≡

```
uint curstamp;      /* a unique value for marking literals and levels of interest */
uint *learn;      /* literals in a clause being learned */
int oldptr;      /* this many old literals contributed to learned clause so far */
int jumplev;      /* level to which we'll return after learning */
int tl;      /* trail location for examination of stamped literals */
int xnew;      /* excess new literals in the current conflict clause */
int clevels;      /* levels represented in the current conflict clause */
uint resols;      /* resolutions made while reducing the current conflict clause */
uint learned_size;      /* number of literals in the learned clause */
int prelearned_size;      /* learned_size before simplification */
int trivial_learning;      /* does the learned clause involve every decision? */
```

91. The algorithm that follows will use *curstamp*, *curstamp* + 1, and *curstamp* + 2.

```
⟨ Bump curstamp to a new value 91 ⟩ ≡
  if (curstamp ≥ #fffffffe) {
    for (k = 1; k ≤ vars; k++) oo, vmem[k].stamp = levstamp[k + k − 2] = 0;
    curstamp = 1;
  } else curstamp += 3;
```

This code is used in section 86.

92.

```
⟨ Reduce xnew to zero 92 ⟩ ≡
  while (xnew) {
    while (1) {
      o, l = trail[tl−−];
      if (o, vmem[thevar(l)].stamp ≡ curstamp) break;
    }
    xnew−−;
    ⟨ Resolve with the reason of l 93 ⟩;
  }
```

This code is used in section 86.

93. At this point the current conflict clause is represented implicitly as the set of negatives of the literals *trail*[*j*] for $j \le tl$ that have *stamp* = *curstamp*, together with *bar*(*l*). Old literals in that set are in the *learn* array. The conflict clause contains exactly *xnew* + 1 new literals besides *bar*(*l*); we will replace *bar*(*l*) by the other literals in *l*'s reason.

```
⟨ Resolve with the reason of l 93 ⟩ ≡
  resols++;
  if (verbose & show_gory_details) fprintf(stderr, "␣["O"s"O".8s]", litname(l));
  o, c = lmem[l].reason;
  if (c < 0) ⟨ Resolve with binary reason 94 ⟩
  else if (c) {    /* l = mem[c].lit */
    if (c ≥ first_learned) ⟨ Bump c's activity 81 ⟩;
    for (o, s = size(c), k = c + s − 1; k > c; k−−) {
      o, l = bar(mem[k].lit);
      if (o, vmem[thevar(l)].stamp ≠ curstamp) ⟨ Stamp l as part of the conflict clause milieu 95 ⟩;
    }
    if (xnew + oldptr + 1 < s ∧ xnew) ⟨ Subsume c by removing its first literal 98 ⟩;
  }
```

This code is used in section 92.

94.

```
⟨ Resolve with binary reason 94 ⟩ ≡
  {
    l = −c;
    if (o, vmem[thevar(l)].stamp ≠ curstamp) ⟨ Stamp l as part of the conflict clause milieu 95 ⟩;
  }
```

This code is used in section 93.

95. ⟨ Stamp l as part of the conflict clause milieu 95 ⟩ ≡

```
{
  o, jj = vmem[thevar(l)].value & −2;
  if (¬jj) confusion("permanently␣false␣lit");
  else {
    o, vmem[thevar(l)].stamp = curstamp;
    ⟨ Bump l's activity 70 ⟩;
    if (jj ≥ llevel) xnew++;
    else {
      if (jj > jumplev) jumplev = jj;
      o, learn[oldptr++] = bar(l);
      if (verbose & show_gory_details)
        fprintf(stderr, "␣"O"s"O".8s{"O"d}", litname(bar(l)), vmem[thevar(l)].value ≫ 1);
      if (o, levstamp[jj] < curstamp) o, levstamp[jj] = curstamp, clevels++;
      else if (levstamp[jj] ≡ curstamp) o, levstamp[jj] = curstamp + 1;
    }
  }
}
```

This code is used in sections 87, 93, and 94.

96. The *stack* and *conflictdat* arrays have enough room for twice the number of variables in the worst case.

The *levstamp* array also has that same size. We use its even-numbered slots when learning and its odd-numbered slots when recycling.

⟨ Allocate the auxiliary arrays 57 ⟩ +≡

```
stack = (int *) malloc(vars * 2 * sizeof(int));
if (¬stack) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣stack␣array!\n");
  exit(−16);
}
bytes += vars * 2 * sizeof(int);
conflictdat = (int *) malloc(vars * 2 * sizeof(int));
if (¬conflictdat) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣conflictdat␣array!\n");
  exit(−16);
}
bytes += vars * 2 * sizeof(int);
levstamp = (uint *) malloc(2 * vars * sizeof(uint));
if (¬levstamp) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣levstamp␣array!\n");
  exit(−16);
}
bytes += 2 * vars * sizeof(uint);
for (k = 0; k < vars; k++) o, levstamp[k + k] = 0;
```

97. ⟨ Global variables 4 ⟩ +≡

```
int *stack;    /* place for homemade recursion control */
int stackptr;    /* number of elements in the stack */
int *conflictdat;    /* recorded data about conflicts in full runs */
int conflict_level;    /* pointer to top of the recorded conflict stack */
uint *levstamp;    /* memos for recursive answers; also binary conflict info */
```

98. Here now is the technique of “on-the-fly subsumption,” which allows us to strengthen the clause c because it happens to contain the current conflict clause. [This technique was discovered by Han and Somenzi in America, and independently by Hamadi, Jabbour, and Saïs in Europe, both in 2009!]

The current conflict has been obtained by resolving c with another clause, and by removing literals that are false at level 0. We’ve also removed such literals from c. Therefore we know that the current conflict clause equals c minus its first literal (which is true and was resolved away).

Clause c is the reason for l, and it becomes the reason for a false literal that would have produced an earlier conflict. (That false literal must have become false at the current trail level.) We don’t have to update the reason data, because backtracking will clear it out before it will be needed.

There are strange scenarios in which $c =$ *prev_learned* and the newly learned clause might duplicate the previous one. The previous one won’t be removed unless we now happen to be watching the literal that will later be called *bar*(*lll*).

```
⟨ Subsume c by removing its first literal 98 ⟩ ≡
  {
    l = mem[c].lit;    /* no mem charged; we already knew this literal */
    o, size(c) = −−s, subsumptions++;
    if (learned_file ∧ s ≤ learn_save) {
      fprintf(learned_file, "␣");    /* this space identifies a subsumer */
      for (k = c + 1; k ≤ c + s; k++) fprintf(learned_file, "␣"O"s"O".8s", litname(mem[k].lit));
      fprintf(learned_file, "\n");
      fflush(learned_file);
      learned_out++;
    }
    o, r = link0(c);
    ⟨ Remove c from l’s watch list 106 ⟩;
    o, ll = mem[c + s].lit;    /* this false literal will now be moved elsewhere */
    for (lll = ll, k = c + s; ; k−−) {    /* lll = mem[k].lit */
      o, r = vmem[thevar(lll)].value & −2;
      if (r ≡ llevel) break;
      o, lll = mem[k − 1].lit;
    }
    if (lll ≠ ll) o, mem[k].lit = ll;
    oo, mem[c + s].lit = l + sign_bit, mem[c].lit = lll;
    ooo, link0(c) = lmem[lll].watch, lmem[lll].watch = c;
    if (verbose & show_watches) fprintf(stderr, "␣["O"s"O".8s␣watches␣"O"d]", litname(lll), c);
  }
```

This code is used in section 93.

99. Simplifying the learned clause. Suppose the clause to be learned is $\bar{l} \lor \bar{a}_1 \lor \cdots \lor \bar{a}_k$. Many of the literals $\bar{a}_j$ often turn out to be redundant, in the sense that a few well-chosen resolutions will remove them.

For example, if the reason of a_4 is $a_4 \lor \bar{a}_1 \lor \bar{b}_1$ and the reason of b_1 is $b_1 \lor \bar{a}_2 \lor \bar{b}_2$ and the reason of b_2 is $b_2 \lor \bar{a}_1 \lor \bar{a}_3$, then $\bar{a}_4$ is redundant.

Niklas Sörensson, one of the authors of MiniSAT, noticed that learned clauses could typically be shortened by 30% when such simplifications are made. Therefore we certainly want to look for removable literals, even though the algorithm for doing so is somewhat tricky.

The literal $\bar{a}$ is redundant in the clause-to-be-learned if and only if the other literals in its reason are either present in that clause or (recursively) redundant. (In the example above we must check that $\bar{a}_1$ and $\bar{b}_1$ satisfy this condition; that boils down to observing that $\bar{b}_1$ is redundant, because $\bar{b}_2$ is redundant.)

Since the relation $\succ^+$ is a partial ordering, we can determine redundancy by using a "bottom up" method with this recursive definition. Or we can go "top down" with memoization (which is what we'll do): We shall stamp a literal b with $curstamp + 1$ if $\bar{b}$ is known to be redundant, and with $curstamp + 2$ if $\bar{b}$ is known to be nonredundant. Once we know a literal's status, we won't need to apply the recursive definition again.

A nice trick (also due to Sörensson) can be used to speed this process up, using the fact that a non-decision literal always depends on at least one other literal at the same level: A literal $\bar{a}_j$ can be redundant only if it shares a level with some other literal $\bar{a}_i$ in the learned clause. Furthermore, a literal $\bar{b}$ not in that clause can be redundant only if it shares a level with some $\bar{a}_j$.

A careful reader of the code in the previous sections will have noticed that we've set $levstamp[t + t] = curstamp$ if level t contains exactly one of the literals $\bar{a}_j$, and we've set $levstamp[t + t] = curstamp + 1$ if it contains more than one. Those facts will help us decide non-redundancy without pursuing the whole recursion into impossible levels.

100. Instead of doing this computation with a recursive procedure, I want to control the counting of memory accesses, and to take advantage of the special logical structure that's present. So the program here uses an explicit stack to hold the parameters of unfinished queries.

When we enter this section, *stackptr* will be zero (it says here). When we leave it, whether by going to *redundant* or not, the original value of l will be in *ll*. I think this loop makes an instructive example of how recursion relates to iteration.

One can prove inductively that, at label *test*, we have $vmem[thevar(l)].stamp \le curstamp$, with equality if and only if $stackptr = 0$.

```
⟨ If l̄ is redundant, goto redundant 100 ⟩ ≡
  if (stackptr) confusion("stack");
test: ll = l;
  o, c = lmem[l].reason;
  if (c ≡ 0) goto clear_stack;     /* decision literal is never redundant */
  if (c < 0) {     /* binary reason */
    l = bar(−c);
    o, s = vmem[thevar(l)].stamp;
    if (s ≥ curstamp) {
      if (s ≡ curstamp + 2) goto clear_stack;     /* known non-redundant */
    } else {
      o, stack[stackptr++] = ll;
      goto test;
    }
  } else {
    for (o, k = c + size(c) − 1; k > c; k−−) {
      oo, l = bar(mem[k].lit), s = vmem[thevar(l)].stamp;
      if (s ≥ curstamp) {
        if (s ≡ curstamp + 2) goto clear_stack;     /* known non-redundant */
        continue;     /* in learned clause or known redundant */
      }
      o, s = vmem[thevar(l)].value & −2;
      if (s ≡ 0) continue;     /* literals on level 0 are redundant */
      o, s = levstamp[s];
      if (s < curstamp) {     /* the level is bad */
        o, vmem[thevar(l)].stamp = curstamp + 2;
        goto clear_stack;
      }
      o, stack[stackptr] = k, stack[stackptr + 1] = ll, stackptr += 2;
      goto test;
    test1: continue;
    }
  }
is_red: o, vmem[thevar(ll)].stamp = curstamp + 1;     /* we've proved bar(ll) redundant */
  if (stackptr) {
    oo, ll = stack[−−stackptr], c = lmem[ll].reason;
    if (c < 0) goto is_red;
    o, k = stack[−−stackptr];
    goto test1;     /* jump back into the loop */
  }
  goto redundant;
  ⟨ Clear the stack 101 ⟩;
```

This code is used in section 102.

101. If any of the literals we encounter during that recursive exploration are non-redundant, the literal *ll* we're currently working on is non-redundant, and so are all of the literals on the stack.

(The literal at the bottom of the stack belongs to the learned clause, so we keep its stamp equal to *curstamp*. The other literals, whose stamp was less than *curstamp*, are now marked with *curstamp* + 2.)

```
⟨ Clear the stack 101 ⟩ ≡
clear_stack: if (stackptr) {
    o, vmem[thevar(ll)].stamp = curstamp + 2;
    o, ll = stack[−− stackptr];
    o, c = lmem[ll].reason;
    if (c > 0) stackptr −−;
    goto clear_stack;
  }
```

This code is used in section 100.

102. Sometimes the learned clause turns out to be unnecessarily long even after we simplify it. This can happen, for example, if the decision literal *l* on level 1 is not part of the clause, but all the other literals have a reason that depends on *l*; then no literal is redundant, by our definitions, yet many literals can be from the same level.

If the learned clause size exceeds the jump level plus *trivial_limit*, we replace it by a "trivial" clause based on decision literals only. (In such cases we are essentially doing no better than an ordinary backtrack algorithm.)

```
⟨ Simplify the learned clause 102 ⟩ ≡
  learned_size = oldptr + 1;
  cells_prelearned += learned_size, prelearned_size = learned_size;
  for (kk = 0; kk < oldptr; kk ++) {
    o, l = bar(learn[kk]);
    oo, s = levstamp[vmem[thevar(l)].value & −2];
    if (s < curstamp + 1) continue;      /* l's level doesn't support redundancy */
    ⟨ If l̄ is redundant, goto redundant 100 ⟩;
    continue;
  redundant: learned_size −−;
    if (verbose & show_gory_details)      /* note that l has been moved to ll */
      fprintf(stderr, "("O"s"O".8s␣is␣redundant)\n", litname(bar(ll)));
  }
  if (learned_size ≤ (jumplev ≫ 1) + trivial_limit) trivial_learning = 0;
  else trivial_learning = 1, clevels = jumplev ≫ 1, learned_size = clevels + 1, trivials ++;
  cells_learned += learned_size, total_learned ++;
  ⟨ Update the smoothed-average stats after a clause has been learned 40 ⟩;
```

This code is used in section 125.

103. The following code is used only when *learned_size* > 1. (Learned unit clauses are, of course, happy events; but we deal with them separately.)

The new clause must be watched by two literals. One literal in this clause, namely *lll*, was formerly false but it will become true. It's the one that survived from the conflict on the active level, and it will be one of the watchers we need.

All other literals in the learned clause are currently false. We must choose one of those on the highest level (furthest from root level) to be a watcher. For if we don't, backtracking might take us to a lower level on which the clause becomes forcing, yet we won't see that fact — we won't be watching it! (The true literal and an unwatched literal become unassigned during backtracking. Then, if the unwatched literal becomes false, we won't notice that the formerly true literal is now forced true again.)

```
⟨ Learn the simplified clause 103 ⟩ ≡
  {
    ⟨ Determine the address, c, for the learned clause 104 ⟩;
    ⟨ Store the learned clause c 107 ⟩;
    prev_learned = c;
    if (learned_file ∧ learned_size ≤ learn_save) ⟨ Output c to the file of learned clauses 108 ⟩;
  }
```

This code is used in sections 125 and 134.

104. In early runs of this program, I noticed several times when the previously learned clause is immediately subsumed by the next clause to be learned. On further inspection, it turned out that this happened when the previously learned clause was the reason for a literal on a level that is going away (because *jumplev* is smaller).

So I now check for this case. Backtracking has already zeroed out this literal's reason.

```
⟨ Determine the address, c, for the learned clause 104 ⟩ ≡
  if (prev_learned) {
    o, l = mem[prev_learned].lit;
    if (¬trivial_learning ∧ (o, lmem[l].reason ≡ 0) ∧ (o, vmem[thevar(l)].value ≡ unset))
      ⟨ Discard clause prev_learned if it is subsumed by the current learned clause 105 ⟩;
  }
  c = max_learned;    /* this will be the address of the new clause */
  o, mem[c + learned_size].lit = 0;    /* put zero at end of mem */
  max_learned += learned_size + learned_extra;
  if (max_learned > max_cells_used) {
    if (max_learned ≥ memsize) {
      fprintf(stderr, "Memory␣overflow␣(memsize="O"u<"O"u),␣please␣increase␣m!\n", memsize,
          max_cells_used + 1);
      exit(−666);
    }
    bytes += (max_learned − max_cells_used) * sizeof(cel);
    max_cells_used = max_learned;
  }
```

This code is used in section 103.

105. The first literal of *prev_learned* has no set value, so it isn't part of the conflict clause. We will discard *prev_learned* if all literals of the learned clause appear among the *other* literals of *prev_learned*.

```
⟨ Discard clause prev_learned if it is subsumed by the current learned clause 105 ⟩ ≡
  {
    for (o, k = size(prev_learned) − 1, q = learned_size; q ∧ k ≥ q; k−−) {
      oo, l = mem[prev_learned + k].lit, r = vmem[thevar(l)].value & −2;
      if ((l ≡ lll ∨ (uint) r ≤ jumplev) ∧ (o, vmem[thevar(l)].stamp ≡ curstamp)) q−−;
          /∗ yes, l is in the learned clause ∗/
    }
    if (q ≡ 0) {
      max_learned = prev_learned;      /∗ forget the previously learned clause ∗/
      if (verbose & show_gory_details) fprintf(stderr, "(clause␣"O"d␣discarded)\n", prev_learned);
      discards++;
      o, c = prev_learned, activ(c) = 0;
      o, l = mem[c].lit, r = link0(c);
      ⟨ Remove c from l's watch list 106 ⟩;
      oo, l = mem[c + 1].lit, r = link1(c);
      ⟨ Remove c from l's watch list 106 ⟩;
    }
  }
```

This code is used in section 104.

106. At this point *r* is the successor of *c* in the watch list.

```
⟨ Remove c from l's watch list 106 ⟩ ≡
  for (o, wa = lmem[l].watch, q = 0; wa ≠ c; q = wa, wa = next_wa) {
    o, p = mem[wa].lit;
    o, next_wa = (p ≡ l ? link0(wa) : link1(wa));
  }
  if (¬q) o, lmem[l].watch = r;
  else if (p ≡ l) o, link0(q) = r;
  else o, link1(q) = r;
```

This code is used in sections 98 and 105.

107. ⟨ Store the learned clause *c* 107 ⟩ ≡

```
if (activ(c)) confusion("bumps");
size(c) = learned_size;    /* no mem need be charged here, since we're charging for link0, link1 */
o, mem[c].lit = lll;
oo, link0(c) = lmem[lll].watch;
o, lmem[lll].watch = c;
if (trivial_learning) {
  for (j = 1, k = jumplev; k; j++, k −= 2) {
    oo, l = bar(trail[leveldat[k]]);
    if (j ≡ 1) ooo, link1(c) = lmem[l].watch, lmem[l].watch = c;
    o, mem[c + j].lit = l;
  }
  if (verbose & show_gory_details) fprintf(stderr, "(trivial␣clause␣is␣substituted)\n");
} else
  for (k = 1, j = 0, jj = 1; k < learned_size; j++) {
    o, l = learn[j];
    if (o, vmem[thevar(l)].stamp ≡ curstamp) {    /* not redundant */
      o, r = vmem[thevar(l)].value;
      if (jj ∧ r ≥ jumplev) {
        o, mem[c + 1].lit = l;
        oo, link1(c) = lmem[l].watch;
        o, lmem[l].watch = c;
        jj = 0;
      } else o, mem[c + k + jj].lit = l;
      k++;
    }
  }
```

This code is used in section 103.

108. ⟨ Output *c* to the file of learned clauses 108 ⟩ ≡

```
{
  for (k = c; k < c + learned_size; k++) fprintf(learned_file, "␣"O"s"O".8s", litname(mem[k].lit));
  fprintf(learned_file, "\n");
  fflush(learned_file);
  learned_out++;
}
```

This code is used in section 103.

109. Recycling unhelpful clauses. After thousands of conflicts have occurred, we have learned thousands of new clauses. New clauses guide the search by steering us away from unproductive paths; but they also slow down the propagation process because we have to watch them.

Therefore we try to rank the clauses that have accumulated, and we periodically attempt to weed out the ones that appear to be hurting us more than they help.

This program assesses the utility of learned clauses by using a heuristic measure of quality inspired by the paper of Gilles Audemard and Laurent Simon in *IJCAI* **21** (2009), 399-404. Suppose the literals of clause c appear on exactly $p+q$ distinct levels of the trail, where there's at least one true literal in p of those levels, but all literals of the other q levels are false. Then we give c the score $p+\alpha q$, called its "range." Heuristically, this range will tend to be small if c is going to participate in future forcing operations.

The parameter α equals 0.2 by default, but users can tune it to their heart's content, as long as $0 \le \alpha \le 1$. Audemard and Simon considered only the case $\alpha = 1$ in their paper, calling $p+q$ the "literal block distance" of c. Smaller values of α appeared to give even better results, in my early tests; however, I've had mixed results since then. Certainly $\alpha = 0$ is too small, because p tends to have a limited range and q is needed to break ties. Similarly, I think $\alpha = 1$ is inadvisable, because p is needed to break ties in clauses with the same literal block distance.

If a learned clause is currently used as the reason for some literal in the trail, we must keep it: That clause is "asserting." So we give it range 0. (Except at root level.)

Armin Biere has advised me not to recycle clauses of size 3 or less. But this program doesn't make any special provision for such clauses, because they will almost surely stick around as a consequence of the range heuristic.

Let's suppose that we have accumulated h learned clauses in *mem*, and that we want to reduce that number from h to $h/2$. We shall do that by retaining those clauses whose range lies below the median range.

A precise determination of the median isn't necessary, because ranges are only heuristic. We actually convert the range to an 8-bit number by computing $\min(\lfloor 16(p+\alpha q)\rfloor, 255)$. (All ranges of 16 or more are therefore considered to be equally bad.) Knowing the distribution of these scaled ranges then makes it easy to select the smallest ones.

```
#define buckets 256    /* number of distinct range levels after scaling */
#define badlevel 16.0    /* ranges greater than this are essentially infinite */
⟨Allocate the auxiliary arrays 57⟩ +≡
  rangedist = (int *) malloc(buckets * sizeof(int));
  if (¬rangedist) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣rangedist␣array!\n");
    exit(−16);
  }
  bytes += buckets * sizeof(int);
  for (k = 0; k + k < buckets; k++) o, rangedist[k + k] = rangedist[k + k + 1] = 0;
```

110. The following program computes the scaled range by using the auxiliary array *levstamp* to identify levels that have been seen before. All odd-numbered entries of *levstamp* should be less than c when this code begins.

```
⟨ Compute the scaled range of c 110 ⟩ ≡
  {
    o, l = mem[c].lit;
    if (o, lmem[l].reason ≡ c) {
      if (o, vmem[thevar(l)].value & −2) o, range(c) = 0, asserts++;
      else goto its_true;      /* true at root level */
    } else {
      for (p = q = 0, k = c + size(c) − 1; k ≥ c; k−−) {
        oo, l = mem[k].lit, v = vmem[thevar(l)].value;
        if (v < 2) {      /* l is defined at root level */
          if ((v ⊕ l) & 1) continue;      /* it's false, ignore it */
        its_true: v = buckets + 1; o, range(c) = buckets + 1;
          goto range_set;      /* it's true, clause is superfluous */
        } else {
          if (o, levstamp[(v & −2) + 1] < c) o, levstamp[(v & −2) + 1] = c, q++;
              /* q here is called p + q above */
          if (levstamp[(v & −2) + 1] ≡ c ∧ (((l ⊕ v) & 1) ≡ 0))      /* true literal */
            o, levstamp[(v & −2) + 1] = c + 1, p++;
        }
      }
      v = (int)((buckets/badlevel) * ((float) p + alpha * (float)(q − p)));
      if (v ≥ buckets) v = buckets − 1;
      o, range(c) = v;
      if (v < minrange) minrange = v;
      if (v > maxrange) maxrange = v;
      oo, rangedist[v]++;
    }
  range_set: ;
  }
```

This code is used in section 112.

111.

```
⟨ Global variables 4 ⟩ +≡
  int *rangedist;      /* how many clauses have a particular scaled range? */
  int asserts;      /* how many learned clauses are assertions that must remain? */
  int minrange;      /* the smallest scaled range we've seen on this round */
  int maxrange;      /* the largest scaled range we've seen on this round */
  int recycle_point;      /* the first clause learned after the current full run */
  int budget;      /* the desired number of learned clauses after recycling */
  ullng *clause_heap;      /* auxiliary array for partially sorting clause activity */
  int clause_heap_size;      /* its maximum size */
```

112. Each clause recycling pass is a major event, something like spring cleaning. First we prepare to compute the ranges by doing a full run, so that every variable has been assigned to a level and a tentative Boolean value. Then we backtrack to level zero, possibly learning new clauses as we go. (Any such clauses c will have $c \geq recycle\_point$; they have no range, so we treat them as if they were asserted, with range zero.) And then we drastically reduce our database of learned clauses, using this opportunity to remove clauses that are permanently satisfied and to remove literals that are permanently false. During this process the watch lists need to be dismantled and rebuilt.

Notice that the second step in this process, backtracking to level zero, is very much like doing a restart. (The only difference is that "warmup" rounds are automatically scheduled after every true restart.) Thus the decisions that are taken at levels 1, 2, ... will not necessarily match the decisions that were in force at those levels when we decided to do a recycling pass.

I don't think that is a bad thing. However, we could recreate those decisions if we wanted to, by doing the following when backtracking past a decision literal l: Set l's activity to the currently largest activity, which is the activity of the variable currently in *heap*[0]; then bump it up, so that it becomes the new champion.

```
⟨ Compute ranges for clause recycling 112 ⟩ ≡
  recycle_point = max_learned;
  minrange = buckets, maxrange = 0;
  asserts = 0;
  for (k = 0; k < vars; k++) o, levstamp[k + k + 1] = 0;
  for (h = 0, c = first_learned; c < max_learned; h++, c = endc + learned_extra) {
    o, endc = c + size(c);
    ⟨ Compute the scaled range of c 110 ⟩;
    while (o, mem[endc].lit & sign_bit) endc++;
  }
  budget = h/2;
  prev_learned = 0;
```

This code is used in section 133.

113.
```
⟨ Recycle half of the learned clauses 113 ⟩ ≡
  ⟨ Compress the database 114 ⟩;
  ⟨ Recompute all the watch lists 122 ⟩;
  recycle_point = 0;
```

This code is used in section 133.

114. ⟨ Compress the database 114 ⟩ ≡

```
for (o, j = minrange, s = asserts + rangedist[j]; s < budget ∧ j < maxrange; ) o, s += rangedist[++j];
if (s > budget) ⟨ Remove t = s − budget clauses at the threshold 115 ⟩;
for (k = minrange ≫ 1; k + k ≤ maxrange; k++) o, rangedist[k + k] = rangedist[k + k + 1] = 0;
for (h = 0, cc = c = first_learned; c < max_learned; c = endc + learned_extra) {
  o, jj = endc = c + size(c);
  while (o, mem[endc].lit & sign_bit) o, mem[endc++].lit = 0;
  if (c < recycle_point ∧ (o, range(c) > j)) continue;     /* reject when the range is too high */
  for (kk = cc, k = c; k < jj; k++) {
    o, l = mem[k].lit;
    o, v = vmem[thevar(l)].value;
    if ((uint) v ≠ unset) {     /* l has a permanent value at root level */
      if ((v ⊕ l) & 1) continue;     /* don't copy a permanently false literal */
      break;     /* and don't copy a permanently satisfied clause */
    } else o, mem[kk++].lit = l;     /* but do copy otherwise */
  }
  if (k < jj) continue;     /* reject a satisfied clause */
  h++; ⟨ Wrap up clause cc 121 ⟩;
}
max_learned = cc, prev_learned = 0;
o, mem[max_learned − learned_extra].lit = 0;     /* put zero at end of mem */
if (verbose & (show_recycling + show_recycling_details))
  fprintf(stderr, "␣(recycling␣reduced␣"O"d␣learned␣clauses␣to␣"O"d)\n", budget * 2 + 1, h);
    /* a little white lie sometimes */
```

This code is used in section 113.

115. Clause activity scores are used only to break ties. So it's natural to ask whether the effort of computing them and sorting through them is actually worthwhile. Armin Biere has told me that a small but significant number of problems do have a fairly large number of clauses at the median range, so I'm following his recommendation.

⟨ Remove $t = s - budget$ clauses at the threshold 115 ⟩ ≡

```
{
  register ullng accum;

  t = s − budget;
  jj = rangedist[j] − t;
  if (jj > clause_heap_size) jj = clause_heap_size;
  ⟨ Put jj entries of range j into the clause heap 117 ⟩;
  ⟨ Establish heap order in the clause heap 118 ⟩;
  ⟨ Increase the range of t clauses from j to j + 1 120 ⟩;
}
```

This code is used in section 114.

116. ⟨ Allocate the auxiliary arrays 57 ⟩ +≡

```
clause_heap_size = recycle_bump ≫ 1;
clause_heap = (ullng *) malloc(clause_heap_size * sizeof(ullng));
if (¬clause_heap) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣clause_heap␣array!\n");
  exit(−16);
}
bytes += clause_heap_size * sizeof(ullng);
```

117. Entries of *clause_heap* are packed so that they sort on activity first, location second. (If two clauses have equally low activity, we prefer to forget the one that has had more time to become active.)

We use the fact that nonnegative **float** numbers can be compared as if they were integers. Thus we interpret *active*(*c*) as a '*lit*' instead of as a '*flt*'.

```
⟨ Put jj entries of range j into the clause heap 117 ⟩ ≡
  for (h = 0, c = first_learned; h < jj; c = endc + learned_extra) {
    if (c ≥ recycle_point) confusion("rangedist1");
    o, endc = c + size(c);
    while (o, mem[endc].lit & sign_bit) endc++;
    if (o, range(c) ≡ j) clause_heap[h++] = activ_as_lit(c) + c;
  }
```

This code is used in section 115.

118. ⟨ Establish heap order in the clause heap 118 ⟩ ≡

```
  for (h = jj ≫ 1; h; ) {
    q = h + h, p = −−h, o, accum = clause_heap[p];
    ⟨ Sift accum into the clause heap at p 119 ⟩;
  }
```

This code is used in section 115.

119. At this point $q = p + p + 2$.

```
⟨ Sift accum into the clause heap at p 119 ⟩ ≡
  while (q ≤ jj) {
    if (q ≡ jj ∨ (oo, clause_heap[q − 1] < clause_heap[q])) q−−;
    if (accum ≤ clause_heap[q]) break;      /* equality can't actually occur */
    o, clause_heap[p] = clause_heap[q];
    p = q, q = p + p + 2;
  }
  o, clause_heap[p] = accum;
```

This code is used in sections 118 and 120.

120. We continue to pass over all learned clauses, looking for those whose range is j, until t more are found.

```
⟨ Increase the range of t clauses from j to j + 1 120 ⟩ ≡
  for ( ; ; c = endc + learned_extra) {
    if (c ≥ recycle_point) confusion("rangedist2");
    if (o, range(c) ≡ j) {
      o, accum = activ_as_lit(c) + c;
      if (o, accum < clause_heap[0]) {
        o, range(c) = j + 1;
        if (−−t ≡ 0) break;
      } else {
        o, range((int)(clause_heap[0] & #ffffffff)) = j + 1;
        if (−−t ≡ 0) break;
        p = 0, q = 2;
        ⟨ Sift accum into the clause heap at p 119 ⟩;
      }
    }
    o, endc = c + size(c);
    while (o, mem[endc].lit & sign_bit) endc++;
  }
```

This code is used in section 115.

121. At this point we're operating at root level; that is, $llevel = 0$. And we've just copied the literals of a learned-clause-to-remember into positions $mem[cc].lit$, $mem[cc+1].lit$, …, $mem[kk-1].lit$.

In rare circumstances the simplifications we've made might result in a learned clause of size 1. Or even size 0!

```
⟨ Wrap up clause cc 121 ⟩ ≡
  if (kk ≥ cc + 2) {
    if (verbose & show_recycling_details)
      fprintf(stderr, "␣clause␣"O"d␣=␣recycled␣"O"d␣(size␣"O"d)\n", cc, c, kk − cc);
    ooo, size(cc) = kk − cc, activ(cc) = activ(c), cc = kk + learned_extra;
  } else if (kk ≡ cc) goto unsat;
  else {
    o, l = mem[cc].lit;
    o, vmem[thevar(l)].value = l & 1, vmem[thevar(l)].tloc = eptr;
    o, history[eptr] = 4, trail[eptr++] = l;
    if (verbose & (show_choices + show_details + show_recycling_details))
      fprintf(stderr, "␣level␣0,␣"O"s"O".8s␣from␣recycled␣"O"d\n", litname(l), c);
  }
```

This code is used in section 114.

122. ⟨ Recompute all the watch lists 122 ⟩ ≡

```
for (l = 2; l ≤ max_lit; l++) o, lmem[l].watch = 0;
for (c = clause_extra; c < min_learned; c = endc + clause_extra) {
  o, endc = c + size(c);
  ⟨ Watch the first two literals of c 123 ⟩;
  while (o, mem[endc].lit & sign_bit) endc++;      /* necessary for c < min_learned */
}
for (c = first_learned; c < max_learned; c = endc + learned_extra) {
  o, endc = c + size(c);
  ⟨ Watch the first two literals of c 123 ⟩;
}
```

This code is used in section 113.

123. A technicality for mem counting: We save one memory access either when fetching *mem*$[c+1]$.*lit* or when storing into *link1*(c).

⟨ Watch the first two literals of c 123 ⟩ ≡

```
{
  o, l = mem[c].lit;
  ooo, link0(c) = lmem[l].watch, lmem[l].watch = c;
  l = mem[c + 1].lit;
  ooo, link1(c) = lmem[l].watch, lmem[l].watch = c;
}
```

This code is used in section 122.

124. Putting it all together. Most of the mechanisms that we need to solve a satisfiability problem are now in place. We just need to set them in motion at the proper times.

```
⟨ Solve the problem 124 ⟩ ≡
  ⟨ Finish the initialization 130 ⟩;
square_one: llevel = warmup_cycles = 0;
  if (sanity_checking) sanity(eptr);
  if (verbose & show_initial_clauses) print_unsat();
  lptr = 0;
startup: conflict_level = 0;
  full_run = (warmup_cycles < warmups ? 1 : 0);
proceed: conflict_seen = 0;
  ⟨ Complete the current level, or goto confl 127 ⟩;
newlevel: if (sanity_checking) sanity(eptr);
  if (delta ∧ (mems ≥ thresh)) thresh += delta, print_state(eptr);
  if (mems ≥ timeout) {
    fprintf(stderr, "TIMEOUT!\n"); goto all_done;
  }
  if (eptr ≡ vars) {
    if (¬conflict_level) goto satisfied;
    ⟨ Finish a full run 133 ⟩;
    goto startup;
  }
  if (¬conflict_level) {      /* no conflicting literals are on the trail */
    if (total_learned ≥ doomsday) ⟨ Call it quits 138 ⟩;
    if (total_learned ≥ next_recycle) full_run = 1;
    else if (total_learned ≥ next_restart) ⟨ Restart unless agility is high 136 ⟩;
  }
  llevel += 2;
  ⟨ Choose the next decision literal, l 75 ⟩;
  if (verbose & show_choices ∧ llevel ≤ show_choices_max) fprintf(stderr,
        "Level␣"O"d,␣trying␣"O"s"O".8s␣("O"lld␣mems)\n", llevel ≫ 1, litname(l), mems);
  depth_per_decision += −(depth_per_decision ≫ 7) + ((ullng) llevel ≪ 24);
  trail_per_decision += −(trail_per_decision ≫ 7) + ((ullng) eptr ≪ 25);
  o, lmem[l].reason = 0;
  history[eptr] = 0;
launch: nodes++;
  o, leveldat[llevel] = eptr;
  o, trail[eptr++] = l;
  o, vmem[thevar(l)].tloc = lptr;      /* lptr = eptr − 1 */
  vmem[thevar(l)].value = llevel + (l & 1);
  agility −= agility ≫ 13;      /* use the damping factor 1 − 2^−13 */
  goto proceed;
  ⟨ Resolve the current conflict 125 ⟩;
```

This code is used in section 2.

125. (I should mention somewhere that the updating of *agility* here, and elsewhere, has a known bug: Overflow from $2^{32} - 1$ to 2^{32} is theoretically possible! However, this will certainly never occur in practice; and even if it does, it will cause no great harm.)

```
⟨ Resolve the current conflict 125 ⟩ ≡
confl: if (llevel) {
  prep_clause: ⟨ Deal with the conflict clause c 86 ⟩;
    ⟨ Simplify the learned clause 102 ⟩;    /* Note: lll is the false literal that will become true */
    if (full_run) goto store_clause;
    decisionvar = (lmem[bar(lll)].reason ? 0 : 1);    /* was it first in its level? */
    ⟨ Backtrack to jumplev 128 ⟩;
    if (learned_size > 1) {
      ⟨ Learn the simplified clause 103 ⟩
      if (verbose & (show_details + show_choices)) {
        if ((verbose & show_details) ∨ llevel ≤ show_choices_max)
          fprintf(stderr, "level␣"O"d,␣"O"s"O".8s␣from␣"O"d\n", llevel ≫ 1, litname(lll), c);
      }
      o, lmem[lll].reason = c;
    } else ⟨ Learn a clause of size 1 126 ⟩;
    o, vmem[thevar(lll)].value = llevel + (lll & 1), vmem[thevar(lll)].tloc = eptr;
    history[eptr] = (decisionvar ? 2 : 6);
    o, trail[eptr++] = lll;
    agility −= agility ≫ 13;    /* use the damping factor $1 - 2^{-13}$ */
    agility += 1 ≪ 19;    /* “bug” */
    ⟨ Bump the bumps 82 ⟩;
    if (sanity_checking) sanity(eptr);
    goto proceed;
  }
unsat: 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 129 ⟩;
  }
```

This code is used in section 124.

126. ⟨ Learn a clause of size 1 126 ⟩ ≡

```
  {
    if (verbose & (show_details + show_choices))
      fprintf(stderr, "level␣0,␣learned␣"O"s"O".8s\n", litname(lll));
    if (learned_file) {
      fprintf(learned_file, "␣"O"s"O".8s\n", litname(lll));
      fflush(learned_file);
      learned_out++;
    }
  }
```

This code is used in section 125.

127. ⟨ Complete the current level, or **goto** *confl* 127 ⟩ ≡

```
ebptr = eptr;      /* binary implications needn't be checked after this point */
while (lptr < eptr) {
  o, lt = trail[lptr++];
  if (lptr ≤ ebptr) {
    o, lat = lmem[lt].bimp_end;
    if (lat) {
      l = lt;
      ⟨ Propagate binary implications of l; goto confl if a conflict arises 58 ⟩;
    }
  }
  ⟨ Propagate nonbinary implications of lt; goto confl if there's a conflict 60 ⟩;
}
```

This code is used in section 124.

128. ⟨ Backtrack to *jumplev* 128 ⟩ ≡

```
{
  o, k = leveldat[jumplev + 2];
  while (eptr > k) {
    o, l = trail[−− eptr], v = thevar(l);
    oo, vmem[v].oldval = vmem[v].value;
    o, vmem[v].value = unset;
    o, lmem[l].reason = 0;
    if (eptr < lptr ∧ (o, vmem[v].hloc < 0)) ⟨ Put v into the heap 74 ⟩;
  }
  lptr = eptr;
  if (sanity_checking) {
    while (llevel > jumplev) leveldat[llevel] = −1, llevel −= 2;
  } else llevel = jumplev;
}
```

This code is used in sections 125, 133, 134, and 137.

129. ⟨ Print the solution found 129 ⟩ ≡

```
for (k = 0; k < vars; k++) {
  o, printf("␣"O"s"O".8s", litname(trail[k]));
}
printf("\n");
if (out_file) {
  for (k = 0; k < vars; k++) {
    o, fprintf(out_file, "␣"O"s"O".8s", litname(bar(trail[k])));
  }
  fprintf(out_file, "\n");
  fprintf(stderr, "Solution-avoiding␣clause␣written␣to␣file␣`"O"s'.\n", out_name);
}
```

This code is used in section 125.

130. ⟨Finish the initialization 130⟩ ≡

```
if (rand_prob ≥ 1.0) rand_prob_thresh = #80000000;
else rand_prob_thresh = (int)(rand_prob * 2147483648.0);
var_bump_factor = 1.0/(double) var_rho;
clause_bump_factor = 1.0/clause_rho;
show_choices_max ≪= 1;    /* double the level-oriented parameters */
next_recycle = recycle_bump;
if (next_recycle > doomsday) next_recycle = doomsday;
restart_psi = two_to_the_32 * (double) restart_psi_fraction;
restart_u = restart_v = next_restart = 1;
if (verbose & show_details) {
  for (k = 0; k < eptr; k++) fprintf(stderr, ""O"s"O".8s␣is␣given\n", litname(trail[k]));
}
for (k = 0; k < vars; k++) o, leveldat[k + k] = −1, leveldat[k + k + 1] = 0;
```

This code is used in section 124.

131. ⟨Schedule the next restart 131⟩ ≡

```
if ((restart_u & −restart_u) ≡ restart_v) restart_u++, restart_v = 1, restart_thresh = restart_psi;
else restart_v ≪= 1, restart_thresh += restart_thresh ≫ 4;
next_restart = total_learned + restart_v;
if (next_restart > doomsday) next_restart = doomsday;
```

This code is used in section 136.

132. ⟨Schedule the next recycling pass 132⟩ ≡

```
recycle_bump += recycle_inc;
next_recycle = total_learned + recycle_bump;
if (next_recycle > doomsday) next_recycle = doomsday;
```

This code is used in section 133.

133. After a full cycle has assigned values to all the variables, we go back and learn clauses from each of the recorded conflicts.

If clause c_i is learned at level l_i, it tells us that some literal u_i that was set false at l_i can now be set to true at some previous level $l'_i < l_i$. We want to backtrack to the minimum of those levels l'_i, which we'll call *minjumplev*.

```
⟨ Finish a full run 133 ⟩ ≡
  if (total_learned ≥ next_recycle) {
    if (verbose & (show_details + show_gory_details + show_recycling + show_recycling_details))
      fprintf(stderr, "Preparing␣to␣recycle␣("O"llu␣conflicts,␣"O"llu␣mems)\n", total_learned,
          mems);
    ⟨ Compute ranges for clause recycling 112 ⟩;
  } else {
    warmup_cycles++;
    if (verbose & (show_choices + show_details + show_gory_details + show_warmlearn))
      fprintf(stderr, "Finishing␣warmup␣round␣"O"d:\n", warmup_cycles);
  }
  o, leveldat[llevel + 2] = eptr;
  minjumplev = max_lit;     /* an "infinite" level */
  for ( ; conflict_level; ) ⟨ Learn from the conflict at conflict_level 134 ⟩;
  if (recycle_point) jumplev = 0;
  else jumplev = minjumplev;
  ⟨ Backtrack to jumplev 128 ⟩;
  trail_marker = eptr;
  if (jumplev ≡ minjumplev) ⟨ Place the literals learned at minjumplev at the end of the trail 135 ⟩;
  ⟨ Bump the bumps 82 ⟩;
  if (recycle_point) {
    ⟨ Recycle half of the learned clauses 113 ⟩;
    if (sanity_checking) sanity(eptr);
    ⟨ Schedule the next recycling pass 132 ⟩;
  }
```

This code is used in section 124.

134. Trivial clauses that arise during a full run are ignored, unless they are on the first conflict level, because they are never applicable at higher levels.

Several different literals u_i might all turn to be learned at *minjumplev*. Therefore we keep track of them on a stack within the *conflictdat* array. The top item on this stack is accessed via *next_learned*.

```
⟨ Learn from the conflict at conflict_level 134 ⟩ ≡
  {
    o, jumplev = conflict_level, conflict_level = conflictdat[conflict_level];
    ⟨ Backtrack to jumplev 128 ⟩;
    o, c = leveldat[llevel + 1];
    if (c < 0) o, l = −c, ll = conflictdat[llevel + 1];
    goto prep_clause;
  store_clause:      /* apology: these goto's are because of goto's in simplification */
      /* now lll is a false literal that will become true at jumplev */
    if (trivial_learning ∧ conflict_level) {
      cells_prelearned −= prelearned_size;
      cells_learned −= learned_size, total_learned −−, trivials −−;
    } else {
      if (jumplev ≤ minjumplev) {
        if (jumplev < minjumplev) minjumplev = jumplev, next_learned = 0;
        o, conflictdat[llevel] = next_learned, conflictdat[llevel + 1] = lll;
        next_learned = llevel;
      }
      if (learned_size ≡ 1) {
        o, leveldat[llevel + 1] = 0;
        if (learned_file) {
          fprintf(learned_file, "␣"O"s"O".8s\n", litname(lll));
          fflush(learned_file);
          learned_out ++;
        }
        if (verbose & show_warmlearn)
          fprintf(stderr, "(learned␣unit␣clause␣"O"s"O".8s)\n", litname(lll));
      } else {
        ⟨ Learn the simplified clause 103 ⟩;
        o, leveldat[llevel + 1] = c;
        if (verbose & show_warmlearn)
          fprintf(stderr, "(learned␣clause␣"O"d␣of␣size␣"O"d)\n", c, learned_size);
      }
    }
  }
```

This code is used in section 133.

135. ⟨ Place the literals learned at *minjumplev* at the end of the trail 135 ⟩ ≡

```
while (next_learned) {
  o, lll = conflictdat[next_learned + 1];
  o, c = leveldat[next_learned + 1];
  next_learned = conflictdat[next_learned];
  if (verbose & (show_details + show_choices)) {
    if ((verbose & show_details) ∨ llevel ≤ show_choices_max) {
      if (c) fprintf(stderr, "level␣"O"d,␣"O"s"O".8s␣from␣"O"d\n", llevel ≫ 1, litname(lll), c);
      else fprintf(stderr, "level␣0,␣"O"s"O".8s\n", litname(lll));
    }
  }
  o, vmem[thevar(lll)].value = llevel + (lll & 1), vmem[thevar(lll)].tloc = eptr;
  o, lmem[lll].reason = c;
  o, history[eptr] = 4, trail[eptr++] = lll;
}
```

This code is used in section 133.

136. Following the advice of Armin Biere [*Lecture Notes in Computer Science* **4996** (2008), 28–33], I disable restarts when there's lots of agility (recent flips of variables). The threshold is higher when the time to next restart is longer.

⟨ Restart unless *agility* is high 136 ⟩ ≡

```
{
  ⟨ Schedule the next restart 131 ⟩;
  if (agility ≤ restart_thresh) ⟨ Flush literals 137 ⟩
  else if (verbose & show_restarts)
    fprintf(stderr, "No␣restart␣("O"llu␣conflicts,␣"O"llu␣mems,␣agility␣"O".2f)\n",
        total_learned, mems, (double) agility/two_to_the_32);
}
```

This code is used in section 124.

137. Instead of restarting completely, by backing up all the way to level 0, we follow the advice of van der Tak, Ramos, and Heule [*Journal on Satisfiability, Boolean Modeling and Computation* **7** (2011), 133–138]: We return to the first level for which a new variable will be injected into the trail. (That new variable will be the one with maximum activity, among all that are currently unset.) Sometimes that will not require backtracking at all.

(I've lately decided to call this “flushing,” not “restarting,” in my book.)

```
⟨ Flush literals 137 ⟩ ≡
  {
    actual_restarts++;
    if (verbose & (show_details + show_choices + show_restarts))
      fprintf(stderr, "Restarting␣("O"llu␣conflicts,␣"O"llu␣mems,␣agility␣"O".2f)\n",
          total_learned, mems, (double) agility/two_to_the_32);
    if (llevel) {
      while (1) {
        o, v = heap[0];
        if (o, vmem[v].value ≡ unset) break;
        ⟨ Delete v from the heap 77 ⟩;
      }
      o, av = vmem[v].activity;
      for (jumplev = 0; jumplev < llevel; jumplev += 2) {
        oo, v = thevar(trail[leveldat[jumplev + 2]]);    /* a decision variable */
        if (o, vmem[v].activity < av) break;    /* new guy will replace v */
      }
      if (jumplev < llevel) ⟨ Backtrack to jumplev 128 ⟩;
    }
    trail_marker = eptr;
    warmup_cycles = 0;
    goto startup;
  }
```

This code is used in section 136.

138. Well, we didn't solve the problem. Too bad. At least we can report what progress was made.

```
⟨ Call it quits 138 ⟩ ≡
  {
    if (verbose & show_basics)
      fprintf(stderr, "Timeout:␣Terminating␣an␣incomplete␣run␣(level␣"O"d).\n", llevel ≫ 1);
    print_state(eptr);
    if (polarity_outfile) {
      for (k = 0; k < eptr; k++) {
        o, l = trail[k];
        fprintf(polarity_outfile, "␣"O"s"O".8s", litname(l));
        o, vmem[thevar(l)].oldval = unset;
      }
      fprintf(polarity_outfile, "\n");
      for (v = 1; v ≤ vars; v++)
        if (o, vmem[v].oldval ≠ unset)
          fprintf(polarity_outfile, ""O"s"O".8s\n", vmem[v].oldval & 1 ? "~" : "", vmem[v].name.ch8);
      fprintf(stderr, "Polarity␣data␣written␣to␣file␣`"O"s'.\n", polarity_out_name);
    }
    if (restart_file) {
      for (o, k = 0; k < leveldat[2]; k++)      /* print unit clauses learned */
        o, fprintf(restart_file, "␣"O"s"O".8s\n", litname(trail[k]));
      for (c = first_learned; c < max_learned; c = kk + learned_extra) {
        for (o, k = c, kk = c + size(c); k < kk; k++)
          o, fprintf(restart_file, "␣"O"s"O".8s", litname(mem[k].lit));
        fprintf(restart_file, "\n");
      }
      fprintf(stderr, "Current␣learned␣clauses␣written␣to␣file␣`"O"s'.\n", restart_name);
    }
    goto all_done;
  }
```

This code is used in section 124.

139.

```
⟨ Debugging fallbacks 139 ⟩ ≡
  void confusion(char *id)
  {      /* an assertion has failed */
    fprintf(stderr, "This␣can't␣happen␣("O"s)!\n", id);
    exit(−666);
  }
  void debugstop(int foo)
  {      /* can be inserted as a special breakpoint */
    fprintf(stderr, "You␣rang("O"d)?\n", foo);
  }
```

This code is used in section 2.

140. ⟨ Global variables 4 ⟩ +≡

```
int full_run;   /* are we making a pass to gather data on all variables? */
int conflict_seen;   /* have we seen a conflict at the current level? */
int decisionvar;   /* does the learned clause involve the decision literal? */
int prev_learned;   /* number of the clause most recently learned */
int warmup_cycles;   /* this many warmups have been done since restart */
int next_learned;   /* top of stack of literals learned at minjumplev */
int restart_u, restart_v;   /* generators for the reluctant doubling sequence */
ullng restart_thresh;   /* agility threshold for restarting */
int trail_marker;   /* position of the latest restart or full run pass */
int minjumplev;   /* level to which we'll return after a full run */
```

141. Index.

⟨ Subsume c by removing its first literal 98 ⟩ Used in section 93.
⟨ Swap wa to the watch list of l and **continue** 62 ⟩ Used in section 60.
⟨ Type definitions 9, 10, 28, 29, 30 ⟩ Used in section 2.
⟨ Update the smoothed-average stats after a clause has been learned 40 ⟩ Used in section 102.
⟨ Watch the first two literals of c 123 ⟩ Used in section 122.
⟨ Wrap up clause cc 121 ⟩ Used in section 114.

SAT13