(See https://cs.stanford.edu/~knuth/programs.html for date.)

1.* Intro. This program is part of a series of "SAT-solvers" that I'm putting together for my own education as I prepare to write Section 7.2.2.2 of *The Art of Computer Programming*. My intent is to have a variety of compatible programs on which I can run experiments to learn how different approaches work in practice.

Many of the previous implementations in this series—SAT0, SAT3, SAT4, SAT5, and SAT10—were based on a natural backtracking approach that has come to be known in the SAT community as the DPLL paradigm, honoring the pioneering work of Davis, Putnam, Logemann, and Loveland. Several decades of experience with that paradigm have led to an extremely efficient class of programs now called *lookahead solvers*, which devote considerable time to choosing the variables on which to branch. The extra work of making that choice might cost us a factor of a thousand, say, at every branch node; yet we might also decrease the number of nodes by a factor of a million, thus making a net thousand-fold gain. Somewhat to my surprise, this rosy prediction (contrary to what I had believed for many years) actually does work in practice: There are many SAT problems (especially those based on combinatorial tasks, as well as the academic yet appealing cases of unsatisfiable random 3SAT) for which judicious lookaheads outperform any other known method.

Consequently SAT11 is intended to represent a modern lookahead solver. I've based it largely on Marijn Heule's MARCH, which has been regularly classed with the world's best lookahead solvers for the last decade or so. I expect SAT11 to be the most ambitious program of this series, because it combines many advanced ideas that I wish to understand and to explain to the readers of *TAOCP*. On the other hand, I have not included all of the bells and whistles of MARCH; in particular, I've omitted the separate treatment of clause sets that represent linear equations mod 2, as well as the "limited discrepancy search" technique by which branches of the search tree are explored in a nonstandard order.

Actually this program is not SAT11 but SAT11K, an extension that handles general clauses; the original SAT11 limited itself to clauses of length three or less. You might want to read that program first, before getting into the extra complications of this one. (On the other hand, some aspects of this version are simpler. So take heart: You can handle SAT11K just fine.) Asterisks indicate differences between SAT11 and SAT11K.

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 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_flip.h"
  typedef unsigned int uint;    /* a convenient abbreviation */
  typedef unsigned long long ullng;    /* ditto */
  ⟨Type definitions 5⟩;
  ⟨Global variables 3*⟩;
  ⟨Subroutines 29⟩;
  main(int argc, char *argv[])
  {
    register int au, av, aw, h, i, j, jj, k, kk, l, ll, p, pp, q, qq, r, s, cia, cis, ci;
    register int c, cc, hh, la, lp, ls, ola, ols, tla, tls, tll, sl, su, sv, sw;
    register int t, tt, u, uu, v0, v, vv, w, ww, x, y, xl, pu, aa, ss, pv, ua, va;
    ⟨Process the command line 4*⟩;
    ⟨Initialize everything 8⟩;
    ⟨Input the clauses 9⟩;
    if (verbose & show_basics) ⟨Report the successful completion of the input phase 22⟩;
    ⟨Set up the main data structures 37⟩;
    imems = mems, mems = 0;
    ⟨Solve the problem 152*⟩;
  done: if (verbose & show_basics)
      fprintf(stderr, "Altogether␣"O"llu+"O"llu␣mems,␣"O"llu␣bytes,␣"O"llu␣nodes.\n", imems,
          mems, bytes, nodes);
  }
```

3* The default values of parameters below have been tuned for a broad spectrum of SAT instances, based on tests by Holger Hoos in 2015.

```
#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_doubly_gory_details 16   /* verbose code for still more */
#define show_unused_vars 32   /* verbose code to list variables not in solution */
#define show_big_clauses 64   /* verbose code to print all big guys at beginning */
#define show_scores 64   /* verbose code to show the prelookahead scores */
#define show_strong_comps 128   /* verbose code to show strong components */
#define show_looks 256   /* verbose code to show the lookahead forest */
```

⟨ Global variables 3* ⟩ ≡

```
int random_seed = 0;   /* seed for the random words of gb_rand */
int verbose = show_basics + show_unused_vars;   /* level of verbosity */
int show_choices_max = 1000000;   /* above this level, show_choices is ignored */
int hbits = 8;   /* logarithm of the number of the hash lists */
int print_state_cutoff = 32 * 80;   /* 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 */
char *out_name;   /* its name */
FILE *primary_file;   /* file for optional input */
char *primary_name;   /* its name */
int primary_vars;   /* the number of primary variables */
ullng imems, mems;   /* mem counts */
ullng bytes;   /* memory used by main data structures */
ullng nodes;   /* the number of nodes entered */
ullng thresh = 10000000000;   /* report when mems exceeds this, if delta ≠ 0 */
ullng delta = 10000000000;   /* 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 */
float alpha = 0.001;   /* magic constant for heuristic scores */
float gamm = 0.20;   /* magic ratio for the clause reduction heuristic */
int theta64 = 25;   /* the optimization parameter theta, times 64 */
int levelcand = 600;   /* preselected candidates times levels */
int mincutoff = 30;   /* don't cut off fewer than this many candidates */
int max_prelook_arcs = 5000;   /* space available for arcs re strong components */
int dl_max_iter = 1;   /* maximum iterations of double-look */
float dl_rho = 0.9998;   /* damping factor for the double-look trigger */
```

See also sections 7*, 24*, 36, 48, 60, 67, 89, 108, 120, 124, 133, 141, and 164*.

This code is used in section 2*.

4* On the command line one can specify any or all of the following options:

- '`v`⟨ integer ⟩' to enable various levels of verbose output on *stderr*.
- '`c`⟨ positive integer ⟩' to limit the levels on which clauses are shown.
- '`h`⟨ positive integer ⟩' to adjust the hash table size.
-
- '`H`⟨ positive integer ⟩' to limit the literals whose histories are shown by *print_state*. '`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*.)
- '`m`⟨ positive integer ⟩' to adjust the maximum memory size. (The binary logarithm is specified; it must be at most 31.)
- '`a`⟨ positive float ⟩' to adjust the magic constant α in heuristic scores.
- '`g`⟨ positive float ⟩' to adjust the magic ratio γ in the clause reduction heuristic scores *clause_weight*[k].
- '`t`⟨ positive integer ⟩' to adjust the fraction $\theta = n/64$ that triggers clause rearrangement.
- '`p`⟨ positive integer ⟩' to adjust the parameter *levelcand*, approximating "candidates times levels" during the preselection phase.
- '`q`⟨ positive integer ⟩' to adjust the parameter *mincutoff*, the minimum cutoff on the number of candidates during preselection.
- '`z`⟨ positive integer ⟩' to adjust *max_prelook_arcs*, the maximum number of arcs retained when studying the reduced digraph during preselection.
- '`i`⟨ positive integer ⟩' to adjust *dl_max_iter*, the maximum number of iterations allowed during a double-lookahead. (An exceptionally large value will actually *disable* double-lookahead, because there won't be enough available "truth space.")
- '`r`⟨ positive float ⟩' to adjust *dl_rho*, the damping factor for *dl_trigger*.
- '`x`⟨ filename ⟩' to copy the input plus a solution-eliminating clause to the specified file. If the given problem is satisfiable in more than one way, a different solution can be obtained by inputting that file.
- '`V`⟨ filename ⟩' to input a file that lists the names of all "primary" variables. A nonprimary variable will not be used for branching unless its value is forced, or unless all of the primary variables have already been assigned a value.
- '`T`⟨ integer ⟩' to set *timeout*: This program will abruptly terminate, when it discovers that *mems* > *timeout*.

⟨ Process the command line 4* ⟩ ≡

```
for (j = argc - 1, k = 0; j; j--)
  switch (argv[j][0]) {
  case 'v': k |= (sscanf(argv[j] + 1, ""O"d", &verbose) - 1); break;
  case 'c': k |= (sscanf(argv[j] + 1, ""O"d", &show_choices_max) - 1); break;
  case 'H': k |= (sscanf(argv[j] + 1, ""O"d", &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 'm': k |= (sscanf(argv[j] + 1, ""O"d", &memk_max) - 1); break;
  case 'a': k |= (sscanf(argv[j] + 1, ""O"f", &alpha) - 1); break;
  case 'g': k |= (sscanf(argv[j] + 1, ""O"f", &gamm) - 1); break;
  case 't': k |= (sscanf(argv[j] + 1, ""O"d", &theta64) - 1); break;
  case 'p': k |= (sscanf(argv[j] + 1, ""O"d", &levelcand) - 1); break;
  case 'q': k |= (sscanf(argv[j] + 1, ""O"d", &mincutoff) - 1); break;
  case 'z': k |= (sscanf(argv[j] + 1, ""O"d", &max_prelook_arcs) - 1); break;
  case 'i': k |= (sscanf(argv[j] + 1, ""O"d", &dl_max_iter) - 1); break;
  case 'r': k |= (sscanf(argv[j] + 1, ""O"f", &dl_rho) - 1); break;
  case 'x': out_name = argv[j] + 1, out_file = fopen(out_name, "w");
    if (¬out_file) fprintf(stderr, "I can't open file `"O"s' for output!\n", out_name);
    break;
```

```
    case 'V': primary_name = argv[j] + 1, primary_file = fopen(primary_name, "r");
      if (¬primary_file) fprintf(stderr, "I␣can't␣open␣file␣`"O"s'␣for␣input!\n", primary_name);
      break;
    case 'T': k |= (sscanf(argv[j] + 1, ""O"lld", &timeout) − 1); break;
    default: k = 1;       /* unrecognized command-line option */
    }
  if (k ∨ hbits < 0 ∨ hbits > 30 ∨ buf_size ≤ 0 ∨ memk_max < 2 ∨ memk_max > 31 ∨ alpha ≤ 0.0 ∨ gamm ≤
        0 ∨ theta64 < 0 ∨ levelcand ≤ 0 ∨ mincutoff ≤ 0 ∨ max_prelook_arcs ≤ 0 ∨ dl_max_iter ≤ 0) {
    fprintf(stderr, "Usage:␣"O"s␣[v<n>]␣[c<n>]␣[h<n>]␣[b<n>]␣[s<n>]␣[d<n>]␣[m<n>]␣", argv[0]);
    fprintf(stderr, "␣[H<n>]␣[g<f>]␣[a<f>]␣[t<n>]␣[p<n>]␣[q<n]␣[z<n>]");
    fprintf(stderr, "␣[i<n>]␣[r<f>]␣[x<foo>]␣[V<foo>]␣[T<n>]␣<␣foo.sat\n");
    exit(−1);
  }
```

This code is used in section 2*.

5. The I/O wrapper. The following routines read the input and absorb it into temporary data areas from which all of the "real" data structures can readily be initialized. My intent is to incorporate these routines into all of the SAT-solvers in this series. Therefore I've tried to make the code short and simple, yet versatile enough so that almost no restrictions are placed on the sizes of problems that can be handled. These routines are supposed to work properly unless there are more than $2^{32}-1 = 4{,}294{,}967{,}295$ occurrences of literals in clauses, or more than $2^{31}-1 = 2{,}147{,}483{,}647$ variables or clauses.

In these temporary tables, each variable is represented by four things: its unique name; its serial number; the clause number (if any) in which it has most recently appeared; and a pointer to the previous variable (if any) with the same hash address. Several variables at a time are represented sequentially in small chunks of memory called "vchunks," which are allocated as needed (and freed later).

```
#define vars_per_vchunk 341    /* preferably (2^k − 1)/3 for some k */

⟨ Type definitions 5 ⟩ ≡
  typedef union {
    char ch8[8];
    uint u2[2];
    long long lng;
  } octa;
  typedef struct tmp_var_struct {
    octa name;    /* the name (one to eight ASCII characters) */
    uint serial;    /* 0 for the first variable, 1 for the second, etc. */
    int stamp;    /* m if positively in clause m; −m if negatively there */
    struct tmp_var_struct *next;    /* pointer for hash list */
  } tmp_var;

  typedef struct vchunk_struct {
    struct vchunk_struct *prev;    /* previous chunk allocated (if any) */
    tmp_var var[vars_per_vchunk];
  } vchunk;
```

See also sections 6, 26, 27*, 28, 34, 35, 88, 107, and 119.

This code is used in section 2*.

6. Each clause in the temporary tables is represented by a sequence of one or more pointers to the **tmp_var** nodes of the literals involved. A negated literal is indicated by adding 1 to such a pointer. The first literal of a clause is indicated by adding 2. Several of these pointers are represented sequentially in chunks of memory, which are allocated as needed and freed later.

```
#define cells_per_chunk 511    /* preferably 2^k − 1 for some k */

⟨ Type definitions 5 ⟩ +≡
  typedef struct chunk_struct {
    struct chunk_struct *prev;    /* previous chunk allocated (if any) */
    tmp_var *cell[cells_per_chunk];
  } chunk;
```

7*. ⟨ Global variables 3* ⟩ +≡

```
char *buf;   /* buffer for reading the lines (clauses) of stdin */
tmp_var **hash;   /* heads of the hash lists */
uint hash_bits[93][8];   /* random bits for universal hash function */
vchunk *cur_vchunk;   /* the vchunk currently being filled */
vchunk *last_vchunk;   /* another pointer for vchunk manipulation */
tmp_var *cur_tmp_var;   /* current place to create new tmp_var entries */
tmp_var *bad_tmp_var;   /* the cur_tmp_var when we need a new vchunk */
chunk *cur_chunk;   /* the chunk currently being filled */
tmp_var **cur_cell;   /* current place to create new elements of a clause */
tmp_var **bad_cell;   /* the cur_cell when we need a new chunk */
ullng vars;   /* how many distinct variables have we seen? */
ullng clauses;   /* how many clauses have we seen? */
ullng nullclauses;   /* how many of them were null? */
ullng cells;   /* how many occurrences of literals in clauses? */
ullng bclauses;   /* how many clauses are big (have more than two literals)? */
ullng bcells;   /* how many occurrences of literals in big clauses? */
int non_clause;   /* is the current clause ignorable? */
```

8. ⟨ Initialize everything 8 ⟩ ≡

```
gb_init_rand(random_seed);
buf = (char *) malloc(buf_size * sizeof(char));
if (¬buf) {
  fprintf(stderr, "Couldn't␣allocate␣the␣input␣buffer␣(buf_size="O"d)!\n", buf_size);
  exit(−2);
}
hash = (tmp_var **) malloc(sizeof(tmp_var) ≪ hbits);
if (¬hash) {
  fprintf(stderr, "Couldn't␣allocate␣"O"d␣hash␣list␣heads␣(hbits="O"d)!\n", 1 ≪ hbits, hbits);
  exit(−3);
}
for (h = 0; h < 1 ≪ hbits; h++) hash[h] = Λ;
```

See also section 15.

This code is used in section 2*.

9. The hash address of each variable name has h bits, where h is the value of the adjustable parameter *hbits*. Thus the average number of variables per hash list is $n/2^h$ when there are n different variables. A warning is printed if this average number exceeds 10. (For example, if h has its default value, 8, the program will suggest that you might want to increase h if your input has 2560 different variables or more.)

All the hashing takes place at the very beginning, and the hash tables are actually recycled before any SAT-solving takes place; therefore the setting of this parameter is by no means crucial. But I didn't want to bother with fancy coding that would determine h automatically.

```
⟨Input the clauses 9⟩ ≡
  if (primary_file) ⟨Input the primary variables 10⟩;
  while (1) {
    if (¬fgets(buf, buf_size, stdin)) break;
    clauses++;
    if (buf[strlen(buf) − 1] ≠ '\n') {
      fprintf(stderr, "The clause on line "O"lld ("O".20s...) is too long for me;\n", clauses,
          buf);
      fprintf(stderr, " my buf_size is only "O"d!\n", buf_size);
      fprintf(stderr, "Please use the command-line option b<newsize>.\n");
      exit(−4);
    }
    ⟨Input the clause in buf 11*⟩;
  }
  if (¬primary_file) primary_vars = vars;
  if ((vars ≫ hbits) ≥ 10) {
    fprintf(stderr, "There are "O"lld variables but only "O"d hash tables;\n", vars, 1 ≪ hbits);
    while ((vars ≫ hbits) ≥ 10) hbits++;
    fprintf(stderr, " maybe you should use command-line option h"O"d?\n", hbits);
  }
  clauses −= nullclauses;
  if (clauses ≡ 0) {
    fprintf(stderr, "No clauses were input!\n");
    exit(−77);
  }
  if (vars ≥ #80000000) {
    fprintf(stderr, "Whoa, the input had "O"llu variables!\n", vars);
    exit(−664);
  }
  if (clauses ≥ #80000000) {
    fprintf(stderr, "Whoa, the input had "O"llu clauses!\n", clauses);
    exit(−665);
  }
  if (cells ≥ #100000000) {
    fprintf(stderr, "Whoa, the input had "O"llu occurrences of literals!\n", cells);
    exit(−666);
  }
```

This code is used in section 2*.

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

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

This code is used in section 9.

11* ⟨Input the clause in *buf* 11*⟩ ≡

```
  for (j = k = non_clause = 0; ¬non_clause; ) {
    while (buf[j] ≡ '␣') j++;      /* scan to nonblank */
    if (buf[j] ≡ '\n') break;
    if (buf[j] < '␣' ∨ buf[j] > '~') {
      fprintf(stderr, "Illegal␣character␣(code␣#"O"x)␣in␣the␣clause␣on␣line␣"O"lld!\n",
          buf[j], clauses);
      exit(−5);
    }
    if (buf[j] ≡ '~') i = 1, j++;
    else i = 0;
    ⟨Scan and record a variable; negate it if i ≡ 1 12⟩;
  }
  if (k ≡ 0 ∧ ¬non_clause) {
    fprintf(stderr, "(Empty␣line␣"O"lld␣is␣being␣ignored)\n", clauses);
    nullclauses++;      /* strictly speaking it would be unsatisfiable */
  }
  if (non_clause) ⟨Remove all variables of the current clause 19⟩
  else {
    if (k ≥ 3) bclauses++, bcells += k;
    if (k > max_clause) max_clause = k;
  }
  cells += k;
```

This code is used in sections 9 and 10.

12. We need a hack to insert the bit codes 1 and/or 2 into a pointer value.

#**define** *hack_in*(q,t) (**tmp_var** $*$)(t | (**ullng**) q)

⟨Scan and record a variable; negate it if $i \equiv 1$ 12⟩ ≡

```
{
  register tmp_var *p;
  if (cur_tmp_var ≡ bad_tmp_var) ⟨Install a new vchunk 13⟩;
  ⟨Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 16⟩;
  if (¬non_clause) {
    ⟨Find cur_tmp_var→name in the hash table at p 17⟩;
    if (clauses ∧ (p→stamp ≡ clauses ∨ p→stamp ≡ −clauses)) ⟨Handle a duplicate literal 18⟩
    else {
      p→stamp = (i ? −clauses : clauses);
      if (cur_cell ≡ bad_cell) ⟨Install a new chunk 14⟩;
      *cur_cell = p;
      if (i ≡ 1) *cur_cell = hack_in(*cur_cell, 1);
      if (k ≡ 0) *cur_cell = hack_in(*cur_cell, 2);
      cur_cell++, k++;
    }
  }
}
```

This code is used in section 11*.

13. ⟨Install a new **vchunk** 13⟩ ≡

```
{
  register vchunk *new_vchunk;
  new_vchunk = (vchunk *) malloc(sizeof(vchunk));
  if (¬new_vchunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣vchunk!\n");
    exit(−6);
  }
  new_vchunk→prev = cur_vchunk, cur_vchunk = new_vchunk;
  cur_tmp_var = &new_vchunk→var[0];
  bad_tmp_var = &new_vchunk→var[vars_per_vchunk];
}
```

This code is used in section 12.

14. ⟨Install a new **chunk** 14⟩ ≡

```
{
  register chunk *new_chunk;
  new_chunk = (chunk *) malloc(sizeof(chunk));
  if (¬new_chunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣chunk!\n");
    exit(−7);
  }
  new_chunk→prev = cur_chunk, cur_chunk = new_chunk;
  cur_cell = &new_chunk→cell[0];
  bad_cell = &new_chunk→cell[cells_per_chunk];
}
```

This code is used in section 12.

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

⟨ Initialize everything 8 ⟩ +≡

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

16. ⟨ Put the variable name beginning at $buf[j]$ in $cur\_tmp\_var \to name$ and compute its hash code h 16 ⟩ ≡

```
cur_tmp_var→name.lng = 0;
for (h = l = 0; buf[j + l] > '␣' ∧ buf[j + l] ≤ '~'; l++) {
  if (l > 7) {
    fprintf(stderr, "Variable␣name␣"O".9s...␣in␣the␣clause␣on␣line␣"O"lld␣is␣too␣long!\n",
        buf + j, clauses);
    exit(−8);
  }
  h ⊕= hash_bits[buf[j + l] − '!'][l];
  cur_tmp_var→name.ch8[l] = buf[j + l];
}
if (l ≡ 0) non_clause = 1;    /* '~' by itself is like 'true' */
else j += l, h &= (1 ≪ hbits) − 1;
```

This code is used in section 12.

17. ⟨ Find $cur\_tmp\_var \to name$ in the hash table at p 17 ⟩ ≡

```
for (p = hash[h]; p; p = p→next)
  if (p→name.lng ≡ cur_tmp_var→name.lng) break;
if (¬p) {    /* new variable found */
  p = cur_tmp_var++;
  p→next = hash[h], hash[h] = p;
  p→serial = vars++;
  p→stamp = 0;
}
```

This code is used in section 12.

18. The most interesting aspect of the input phase is probably the "unwinding" that we might need to do when encountering a literal more than once in the same clause.

⟨ Handle a duplicate literal 18 ⟩ ≡

```
{
  if ((p→stamp > 0) ≡ (i > 0)) non_clause = 1;    /* tautology */
}
```

This code is used in section 12.

19. An input line that begins with '~␣' is silently treated as a comment. Otherwise redundant clauses are logged, in case they were unintentional. (One can, however, intentionally use redundant clauses to force the order of the variables.)

⟨Remove all variables of the current clause 19⟩ ≡
```
{
  while (k) {
    ⟨Move cur_cell backward to the previous cell 20⟩;
    k−−;
  }
  if (non_clause ∧ ((buf[0] ≠ '~') ∨ (buf[1] ≠ '␣')))
    fprintf(stderr, "(The␣clause␣on␣line␣"O"lld␣is␣always␣satisfied)\n", clauses);
  nullclauses++;
}
```
This code is used in sections 10 and 11*.

20. ⟨Move *cur_cell* backward to the previous cell 20⟩ ≡
```
if (cur_cell > &cur_chunk→cell[0]) cur_cell−−;
else {
  register chunk *old_chunk = cur_chunk;
  cur_chunk = old_chunk→prev; free(old_chunk);
  bad_cell = &cur_chunk→cell[cells_per_chunk];
  cur_cell = bad_cell − 1;
}
```
This code is used in sections 19 and 41*.

21. Here I must omit '*free*(*old_vchunk*)' from the code that's usually in this section, because the variable data will be used later.

⟨Move *cur_tmp_var* backward to the previous temporary variable 21⟩ ≡
```
if (cur_tmp_var > &cur_vchunk→var[0]) cur_tmp_var−−;
else {
  register vchunk *old_vchunk = cur_vchunk;
  cur_vchunk = old_vchunk→prev;    /* and don't 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 46.

22. ⟨Report the successful completion of the input phase 22⟩ ≡
```
fprintf(stderr, "("O"lld␣variables,␣"O"lld␣clauses,␣"O"llu␣literals␣successfully␣read)\n",
    vars, clauses, cells);
```
This code is used in section 2*.

23. SAT solving, version 11. A lookahead solver explores a binary tree of possibilities by choosing, at every decision node, a variable x for which the node's subtrees correspond to asserting x or $\bar{x}$. Several more-or-less independent activities are part of this process:

(1) *Preselection.* At each decision node we choose a subset P of the unassigned variables, based on our best guess as to which of them might be good candidates for further exploration.

(2) *Selection.* We look ahead at the immediate consequences of asserting the truth and falsity of each variable in P. Then we choose the variable that appears to reduce the problem most efficiently.

(3) *Propagation.* We update the current state of the problem by incorporating all consequences of a new assertion.

(4) *Backtracking.* When a contradiction arises in some branch, we must undo the effects of propagation and move to an unexplored branch of the tree.

Each of these activities, except thankfully the last, involves many individual steps.

In some sense this program represents an attitude: We're not afraid to throw code at the problem.

24*. Quite a few cooperating data structures are needed to do all these things at high speed. I shall therefore try to summarize the main ones here.

First, we need to represent the fact that variable x is true, false, or unknown. In fact, we must also deal with intermediate stages by which x is known with various degrees of certainty, based on tentative assumptions that we've made during the lookahead or propagation process. Every variable therefore has an integer *stamp*, which is even if x is true, odd if x is false, and relatively large if the value is relatively certain. Setting the stamp to 0 makes x absolutely unknown; setting the stamp to the highest possible values *real_truth* or *real_truth* $+1$ makes it absolutely true or false. Setting the stamp to an intermediate value like 100 makes x true when the "current stamp" *cs* is 2, 4, ..., 100, but unknown when $cs > 100$. (The value of *cs* is always even, and it never exceeds *known*.)

Second, we need quick access to the consequences of binary clauses. A binary clause $l \vee l'$ is equivalent to two direct implications $\bar{l} \to l'$ and $\bar{l}' \to l$, and the set of all such implications forms a digraph called the implication graph. The *bimp* data structure makes it easy to find all literals that are directly implied by any given literal. (And since $\bar{l} \to l'$ if and only if $\bar{l}' \to l$, it's equally easy to find all literals that *directly imply* any given literal.) New binary implications are learned and added to *bimp* as computation proceeds, and they are stored sequentially in memory; therefore the individual lists are allocated dynamically, within a large array called *mem*, using the "buddy system" (Algorithm 2.5R).

Third, we need a good way to manipulate the "big clauses," namely the clauses that contain three or more literals. Two arrays called *cinx* and *kinx*, which are indexes into two larger arrays called *cmem* and *kmem*, govern this aspect of the problem: *cinx*[*c*] tells where the literals of clause *c* are listed in *cmem*, while *kinx*[*l*] tells where the clauses that contain a given literal *l* are listed in *kmem*. All four of these arrays are allocated once and for all before the main computation begins.

Fourth, there's a sequential list *freevar* of all variables not currently assigned, and an inverse list *freeloc* to tell where a particular variable appears in *freevar*.

Fifth, sixth, etc., there are a bunch of more conventional data structures: Attributes of literal *l* appear in *lmem*[*l*]; attributes of variable *x* appear in *vmem*[*x*]. The *rstack* holds the names of literals in the order they have been (tentatively) set. The *istack* holds the names of variables whose *bimp* entries have grown, together with the value needed to ungrow them when we undo a decision. The *nstack* contains information about nodes of the decision tree that have led to the current state. Later we will define a number of special data structures for use in parts of this program that are essentially self-contained.

⟨ Global variables 3* ⟩ +≡

```
uint *stamp;   /* the current levels of truth, falsity, and uncertainty */
uint *mem;   /* master array of buddy-allocated blocks for bimp lists */
bdata *bimp;   /* indexes into mem for lists of binary implications */
uint *cmem, *kmem;   /* master arrays for cinx and kinx data */
tdata *cinx, *kinx;   /* indexes into cmem and kmem for the big clause info */
tpair *bstack;   /* holding place for big clauses that become binary or unary */
int bptr;   /* the number of elements used in bstack */
int max_use;   /* the maximum number of times any literal occurs */
tpair *tmem;   /* master array of blocks for timp lists */
tdata *timp;   /* indexes into tmem for lists of ternary implications */
uint *freevar, *freeloc;   /* perm of the variables from free to assigned */
int freevars;   /* how many of the variables are still free (unassigned)? */
uint *rstack;   /* stack and queue for backtracking and unit propagation */
int rptr;   /* the number of elements used in rstack */
idata *istack;   /* bimp sizes to be undone if necessary */
int iptr;   /* the number of elements used in istack */
int iptr_max;   /* largest iptr currently allocated in virtual memory */
ndata *nstack;   /* node information */
int level;   /* current depth in the decision tree */
literal *lmem;   /* attributes of literals */
variable *vmem;   /* attributes of variables */
```

25* The variables are numbered 1, 2, ..., n, and the literals corresponding to variable x are $2x$ and $2x+1$ (namely x and $\bar{x}$). Thus the variable that corresponds to literal l is $l \gg 1$, and the complement of literal l is $l \oplus 1$. (Previous programs of this series started the numbering at 0, not 1, in accord with Dijkstra's famous dictum. But we shall find it convenient to reserve the value 0 for use as a sentinel.)

Some arrays (like *stamp* and *freevar*) are indexed by variable numbers, while others (like *bimp* and *kinx*) are indexed by literal numbers. In order to reduce the chance of confusion between the two numbering schemes, variables in the code below will generally be represented by the letters x, y, or z; literals will generally be represented by l, u, v, or w.

```
#define thevar(l) ((l) ≫ 1)   /* the variable that corresponds to l */
#define bar(l) ((l) ⊕ 1)   /* the complement of l */
#define poslit(x) ((x) ≪ 1)   /* the literal x */
#define neglit(x) (((x) ≪ 1) + 1)   /* the literal x̄ */
```

26. An entry in the *bimp* table has four parts: *addr* is the address in *mem* where the list of implications begins; *size* is the current length of that list; *alloc* is the number of memory positions currently available at the given address; and *alloc* always equals 2^k, where k is the fourth field. (Thus we always have $size \le alloc$. The value of k is always at least 2, hence *alloc* is always at least 4. As the computation proceeds, *alloc* might increase, but it never will decrease.)

When *mems* are counted, we assume that *addr* and *size* are fetched or stored together; hence we can access them both at the cost of just one mem. Similarly, *alloc* and k are assumed to be in the same octabyte of memory.

An entry in the *istack* has two parts: *lit* is the literal l whose *bimp* entry is to be restored; *size* is the amount to be placed in *bimp*[l].*size*.

⟨ Type definitions 5 ⟩ +≡

```
typedef struct bdata_struct {
  uint addr;   /* starting place of a sequential list in mem */
  uint size;   /* its current length */
  uint alloc;   /* maximum length before reallocation is necessary */
  uint k;   /* lg alloc */
} bdata;
typedef struct idata_struct {
  uint lit;   /* the l whose size in bimp was changed */
  uint size;   /* its previous size */
} idata;
```

27* An entry in *cinx* has two parts: *addr* is the address in *cmem* where the list of literals for a given clause begins; *size* is initially the length of that list. When literals of a clause become true or false, the *size* field is adjusted in a somewhat tricky way, explained below within the *sanity* routine. The literals of the input clauses are loaded backwards into *cmem*, so that we have $cinx[c].addr + cinx[c].size = cinx[c-1].addr$ when computation begins.

An entry in *kinx* is, likewise, bipartite: *addr* is the address in *kmem* where the list of clauses numbers for a given literal begins, and *size* is the current length of that list. If l is a free literal (namely a literal whose value has not been assigned true or false), $kinx[l].size$ will be the number of clauses that contain l and are not yet satisfied.

When a big clause is reduced to binary, because all but two of its literals have become false while none have become true, we will place it briefly on the *bstack*, whose entries are pairs of literals.

⟨ Type definitions 5 ⟩ +≡

```
typedef struct tdata_struct {
  uint addr;   /* starting place of a sequential list in mem */
  uint size;   /* its current length */
} tdata;   /* one octabyte */
typedef struct tpair_struct {
  uint u, v;   /* a pair of literals */
} tpair;   /* one octabyte */
```

28. An entry in *nstack* has the following fields: *decision* records the literal whose truth is being tentatively asserted; *branch* is 0 in the first branch, or 1 if that branch failed; *rptr* and *iptr* record the initial values of those stack pointers when the node was initialized; *lptr* records the initial value of *rptr* when lookahead for the next level began.

⟨ Type definitions 5 ⟩ +≡

```
typedef struct ndata_struct {
  uint decision;   /* the literal chosen at this branch */
  int branch;   /* did we try and fail to set it the other way? */
  int rptr, iptr, lptr;   /* initial values of stack pointers */
} ndata;
```

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

⟨ Subroutines 29 ⟩ ≡

```
void print_bimp(int l)
{
  register uint la, ls;
  printf(""O"s"O".8s␣->", litname(l));
  for (la = bimp[l].addr, ls = bimp[l].size; ls; la++, ls--) printf("␣"O"s"O".8s", litname(mem[la]));
  printf("\n");
}
```

See also sections 30*, 31*, 33, 50, 61, and 154.

This code is used in section 2*.

30* Similarly, the current data for big clauses gives useful diagnostic info.

```
⟨Subroutines 29⟩ +≡
  void print_clause(int c)
  {
    register uint la, ls;
    printf(""O"d:", c);
    for (la = cinx[c].addr; la < cinx[c - 1].addr; la++) printf("␣"O"s"O".8s"O"s",
          litname(cmem[la]), isfree(cmem[la]) ? "" : iscontrary(cmem[la]) ? "-" : "+");
    printf("␣("O"d)\n", cinx[c].size);
  }
  void print_kinx(int l)
  {
    register uint la, ls;
    printf("kinx["O"s"O".8s]:", litname(l));
    for (la = kinx[l].addr, ls = kinx[l].size; ls; la++, ls--) printf("␣"O"d", kmem[la]);
    printf("\n");
  }
  void print_full_kinx(int l)
  {
    register uint la, k;
    printf("kinx["O"s"O".8s]:", litname(l));
    for (la = kinx[l].addr, k = 0; k < kinx[l].size; k++) printf("␣"O"d", kmem[la + k]);
    if (la + k ≠ kinx[l - 1].addr) {
      printf("␣#");    /* show also the inactive clauses */
      for ( ; la + k < kinx[l - 1].addr; k++) printf("␣"O"d", kmem[la + k]);
    }
    printf("\n");
  }
```

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

```
#define sanity_checking 0    /* set this to 1 if you suspect a bug */
⟨Subroutines 29⟩ +≡
  void sanity(void)
  {
    register int c, j, k, l, la, ls, p, q, u, v;
    for (k = 0; k < vars; k++) {
      if (freevar[freeloc[k + 1]] ≠ k + 1) fprintf(stderr, "freeloc["O"d]␣is␣wrong!\n", k + 1);
      if (freeloc[freevar[k]] ≠ k) fprintf(stderr, "freevar["O"d]␣is␣wrong!\n", k);
    }
    for (k = 0; k < rptr; k++) {
      l = rstack[k];
      if (freeloc[thevar(l)] < freevars) fprintf(stderr, "literal␣"O"d␣on␣rstack␣is␣free!\n", l);
    }
    if (rptr + freevars ≠ vars)
      fprintf(stderr, "rptr="O"d,␣freevars="O"d,␣vars="O"lld\n", rptr, freevars, vars);
    ⟨Check the sanity of bimp and mem 49⟩;
    ⟨Check the sanity of cinx and cmem, kinx and kmem 32*⟩;
  }
```

32*. A big clause $c = l_1 \vee \cdots \vee l_k$ for $k \ge 3$ begins unsatisfied, and its initial size is k. Later, after j of its literals have become false but none of them have yet become true, the size will be $k - j$, as long as $k - j \ge 2$. (The nonfalse literals needn't be adjacent in memory at such times; we only need to know that the residual clause is still big.) But when j reaches $k - 2$, or when one of the literals becomes true, clause c becomes inactive: It disappears from the *kinx* tables of all free literals. Henceforth the elements of c will not be examined again in *cmem* until we undo the setting of the literal that inactivated c.

Thus a clause is inactive if and only if it has been satisfied (contains a true literal) or has become binary (has at most two nonfalse literals). The program here marks inactive clauses by temporarily complementing their *size* fields, so that we can validate the *kinx* data.

⟨ Check the sanity of *cinx* and *cmem*, *kinx* and *kmem* 32* ⟩ ≡

```
for (c = bclauses; c; c--) {
  for (la = cinx[c].addr, k = ls = cinx[c - 1].addr - la, j = 0; ls; la++, ls--) {
    l = cmem[la];
    if (isfree(l)) continue;    /* neither true nor false */
    if (iscontrary(l)) j++;    /* false */
    else goto inactive;    /* true */
  }
  if (j ≥ k - 2) {
    if (cinx[c].size ≠ 2) fprintf(stderr, "ex-big␣clause␣"O"d␣has␣size␣"O"d!\n", c, cinx[c].size);
    goto inactive;
  }
  if (cinx[c].size ≠ k - j)
    fprintf(stderr, "big␣clause␣"O"d␣has␣size␣"O"d␣not␣"O"d\n", c, cinx[c].size, k - j);
  continue;
inactive: cinx[c].size = ~cinx[c].size;
}
for (l = 2; l < badlit; l++)
  if (isfree(l)) {
    for (la = kinx[l].addr, ls = kinx[l].size; ls; la++, ls--) {
      c = kmem[la];
      if ((int) cinx[c].size < 0)
        fprintf(stderr, "kinx["O"s"O".8s]␣includes␣active␣clause␣"O"d!\n", litname(l), c);
    }
    for ( ; la < kinx[l - 1].addr; la++) {
      c = kmem[la];
      if ((int) cinx[c].size ≥ 0)
        fprintf(stderr, "kinx["O"s"O".8s]␣omits␣active␣clause␣"O"d!\n", litname(l), c);
    }
  }
for (c = bclauses; c; c--)
  if ((int) cinx[c].size < 0) cinx[c].size = ~cinx[c].size;
```

This code is used in section 31*.

33. In long runs it's helpful to know how far we've gotten. A numeric code summarizes each decision made so far: 0 or 1 means that we're trying to set a variable true or false, on the first branch of a node ("branch 0"); 2 or 3 is similar, but on the second branch ("branch 1"); 4 or 5 is similar, but when the decision was forced by the decision at the previous branch node; 6 or 7 is similar, but when the decision was found to be forced while looking ahead for the next literal on which to branch.

⟨Subroutines 29⟩ +≡

```
void print_state(int lev)
{
  register int k, r;
  fprintf(stderr, " after " O "lld mems:", mems);
  for (k = r = 0; k < lev; k++) {
    for ( ; r < nstack[k].rptr; r++) fprintf(stderr, "" O "c", '6' + (rstack[r] & 1));
    if (nstack[k].branch < 0) fprintf(stderr, "|");
    else fprintf(stderr, "" O "c", '0' + (rstack[r++] & 1) + (nstack[k].branch << 1));
    for ( ; r < nstack[k + 1].lptr; r++) fprintf(stderr, "" O "c", '4' + (rstack[r] & 1));
    if (k >= print_state_cutoff) {
      fprintf(stderr, "...");  break;
    }
  }
  fprintf(stderr, "\n");
  fflush(stderr);
}
```

34. Each literal has an entry in *lmem*, containing many fields. We will introduce them from time to time as we use them.

⟨Type definitions 5⟩ +≡

```
typedef struct lit_struct {
  int rank;   /* order of appearance in Tarjan's algorithm */
  int link;   /* pointer to another literal */
  int untagged;   /* progress record in Tarjan's algorithm */
  int min;   /* magically important data for Tarjan's algorithm */
  int parent;   /* predecessor in Tarjan's algorithm */
  int vcomp;   /* component representation in Tarjan's algorithm */
  int arcs;   /* pointer to the first successor entry in the cand_arc array */
  uint bstamp;   /* stamped with bstamp when processing new binaries */
  uint dl_fail;   /* stamped with istamp when doublelook didn't force this */
  uint istamp;   /* stamped with istamp when making an entry for istack */
  float wnb;   /* total weighted new binaries, including implied literals */
  uint filler;   /* extra space to fill six octabytes */
} literal;
```

35. Similarly, each variable has an entry in *vmem*, where three fields appear.

```
#define litname(l) (l) & 1 ? "~" : "", vmem[thevar(l)].name.ch8   /* used in printouts */
```

⟨Type definitions 5⟩ +≡

```
typedef struct var_struct {
  octa name;   /* the variable's symbolic name */
  int pfx, len;   /* prefix of its first useful appearance in the search tree */
} variable;
```

36. 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.

⟨ Global variables 3* ⟩ +≡
uint *lits*; /* how many literals are present? */
uint *badlit*; /* one more than the highest literal number */

37. ⟨ Set up the main data structures 37 ⟩ ≡
$lits = vars \ll 1, badlit = lits + 2;$
last_vchunk = *cur_vchunk*;
⟨ Allocate the main arrays 38* ⟩;
⟨ Copy all the temporary variable nodes to the *vmem* array in proper format 46 ⟩;
⟨ Copy all the temporary cells to the *bimp*, *mem*, *cinx*, *cmem*, *kinx*, and *kmem* arrays in proper format 40* ⟩;
⟨ Check consistency 47 ⟩;
⟨ Allocate special arrays 58 ⟩;

This code is used in section 2*.

38* We randomize the initial order of *freevars*, so that different seeds can produce different results (for instance on satisfiable problems).

⟨ Allocate the main arrays 38* ⟩ ≡

```
stamp = (uint *) malloc((vars + 1) * sizeof(uint));
if (¬stamp) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣stamp␣array!\n");
  exit(−10);
}
bytes += (vars + 1) * sizeof(uint);
bimp = (bdata *) malloc(badlit * sizeof(bdata));
if (¬bimp) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣bimp␣array!\n");
  exit(−10);
}
bytes += badlit * sizeof(bdata);
⟨ Initialize mem with empty bimp lists 57 ⟩;
cinx = (tdata *) malloc((bclauses + 1) * sizeof(tdata));
if (¬cinx) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣cinx␣array!\n");
  exit(−10);
}
bytes += (bclauses + 1) * sizeof(tdata);
cmem = (uint *) malloc(bcells * sizeof(uint));
if (¬cmem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣cmem␣array!\n");
  exit(−10);
}
kinx = (tdata *) malloc(badlit * sizeof(tdata));
if (¬kinx) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣cmem␣array!\n");
  exit(−10);
}
bytes += badlit * sizeof(tdata);
kmem = (uint *) malloc(bcells * sizeof(uint));
if (¬kmem) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣kmem␣array!\n");
  exit(−10);
}
bytes += bcells * sizeof(uint);
freevar = (uint *) malloc(vars * sizeof(uint));
if (¬freevar) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣freevar␣array!\n");
  exit(−10);
}
bytes += vars * sizeof(uint);
freeloc = (uint *) malloc((vars + 1) * sizeof(uint));
if (¬freeloc) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣freeloc␣array!\n");
  exit(−10);
}
bytes += (vars + 1) * sizeof(uint);
```

```
for (k = 0; k < vars; k++) {
  mems += 4, j = gb_unif_rand(k + 1);
  if (j ≠ k) {
    o, i = freevar[j];
    oo, freevar[k] = i, freeloc[i] = k;
    oo, freevar[j] = k + 1, freeloc[k + 1] = j;
  } else oo, freevar[k] = k + 1, freeloc[k + 1] = k;
}
freevars = vars;
```

See also section 39.

This code is used in section 37.

39. Although the *rstack* is used rather heavily, for breadth-first searches, a literal and its complement never both appear. Therefore the total size of the *rstack* should never exceed the number of variables.

⟨ Allocate the main arrays 38* ⟩ +≡

```
rstack = (uint *) malloc((vars + 1) * sizeof(uint));
if (¬rstack) {
  fprintf(stderr, "Oops, I can't allocate the rstack array!\n");
  exit(−10);
}
bytes += (vars + 1) * sizeof(uint);
nstack = (ndata *) malloc((vars + 1) * sizeof(ndata));
if (¬nstack) {
  fprintf(stderr, "Oops, I can't allocate the nstack array!\n");
  exit(−10);
}
bytes += (vars + 1) * sizeof(ndata);
lmem = (literal *) malloc(badlit * sizeof(literal));
if (¬lmem) {
  fprintf(stderr, "Oops, I can't allocate the lmem array!\n");
  exit(−10);
}
bytes += badlit * sizeof(literal);
for (l = 2; l < badlit; l++) oo, lmem[l].dl_fail = lmem[l].bstamp = lmem[l].istamp = 0;
vmem = (variable *) malloc((vars + 1) * sizeof(variable));
if (¬vmem) {
  fprintf(stderr, "Oops, I can't allocate the vmem array!\n");
  exit(−10);
}
bytes += (vars + 1) * sizeof(variable);
forcedlit = (uint *) malloc(vars * sizeof(uint));
if (¬forcedlit) {
  fprintf(stderr, "Oops, I can't allocate the forcedlit array!\n");
  exit(−10);
}
bytes += vars * sizeof(uint);
```

40* ⟨ Copy all the temporary cells to the *bimp*, *mem*, *cinx*, *cmem*, *kinx*, and *kmem* arrays in proper format 40* ⟩ ≡

```
forcedlits = 0, cs = proto_truth;      /* prepare for possible unary clauses */
for (l = 2; l < badlit; l++) o, kinx[l].addr = kinx[l].size = 0;      /* clear the counts */
for (c = clauses, k = 0, cc = bclauses; c; c--) {
  la = k;
  ⟨ Insert the cells for the literals of clause c 41* ⟩;
}
cinx[0].addr = k;
if (k ≠ bcells ∨ cc) confusion("cmem");
⟨ Build kinx and kmem from the stored big clauses 44* ⟩;
if (out_file) fflush(out_file);      /* complete the copy of input clauses */
```

This code is used in section 37.

41* 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 41* ⟩ ≡

```
for (i = j = 0; i < 2; ) {
  ⟨ Move cur_cell backward to the previous cell 20 ⟩;
  i = hack_out(*cur_cell);
  p = hack_clean(*cur_cell)→serial;
  p += p + (i & 1);
  o, cmem[k++] = p + 2, j++;
  oo, kinx[p + 2].size++;
}
if (out_file) {
  for (jj = 0; jj < j; jj++) fprintf(out_file, "␣"O"s"O".8s", litname(cmem[la + jj]));
  fprintf(out_file, "\n");
}
if (j < 3) {      /* not big */
  k = la, u = cmem[la];
  oo, kinx[u].size--;
  if (j ≡ 2) {
    oo, v = cmem[la + 1], kinx[v].size--;
    ⟨ Store a binary clause in bimp 43 ⟩;
  } else ⟨ Store a unary clause in forcedlit 42* ⟩;
} else o, cinx[cc].addr = la, cinx[cc].size = k − la, cc--;
```

This code is used in section 40*.

42* Unary clauses in the input might be repeated or contradictory. Thus we must be careful not to overstep the bounds of the *forcedlit* array. The *addr* fields in *kinx* are borrowed here, temporarily, so that no variable is forced twice.

⟨ Store a unary clause in *forcedlit* 42* ⟩ ≡

```
{
  if (o, kinx[u].addr ≡ 0) {
    if (o, kinx[bar(u)].addr) {
      if (verbose & show_choices) fprintf(stderr,
            "Unary␣clause␣"O"d␣contradicts␣unary␣clause␣"O"d\n", c, kinx[bar(u)].addr);
      goto unsat;
    }
    o, kinx[u].addr = c;
    o, forcedlit[forcedlits++] = u;
  }
}
```

This code is used in section 41*.

43. ⟨ Store a binary clause in *bimp* 43 ⟩ ≡

```
{
  o, la = bimp[bar(u)].addr, ls = bimp[bar(u)].size;
  if (o, ls ≡ bimp[bar(u)].alloc) resize(bar(u)), o, la = bimp[bar(u)].addr;
  oo, mem[la + ls] = v, bimp[bar(u)].size = ls + 1;
  o, la = bimp[bar(v)].addr, ls = bimp[bar(v)].size;
  if (o, ls ≡ bimp[bar(v)].alloc) resize(bar(v)), o, la = bimp[bar(v)].addr;
  oo, mem[la + ls] = u, bimp[bar(v)].size = ls + 1;
}
```

This code is used in section 41*.

44* ⟨ Build *kinx* and *kmem* from the stored big clauses 44* ⟩ ≡

```
max_use = 0;
for (j = 0, l = badlit − 1; l ≥ 2; l−−) {
  oo, kinx[l].addr = j, jj = kinx[l].size, j += jj, kinx[l].size = 0;
  if (jj > max_use) max_use = jj;
}
o, kinx[l].addr = j;    /* we'll have kinx[l].addr + kinx[l].size = kinx[l − 1].addr */
if (j ≠ bcells) confusion("kinx1");
for (c = bclauses, j = 0; c; c−−) {
  for (o, k = cinx[c].size; k; k−−) {
    o, u = cmem[j++];
    o, la = kinx[u].addr, ls = kinx[u].size;
    o, kmem[la + ls] = c;
    o, kinx[u].size = ls + 1;
  }
}
if (j ≠ bcells) confusion("kinx2");
⟨ Allocate bstack 45* ⟩;
```

This code is used in section 40*.

45*. ⟨ Allocate *bstack* 45* ⟩ ≡

```
bstack = (tpair *) malloc(max_use * sizeof(tpair));
if (¬bstack) {
  fprintf(stderr, "Oops, I can't allocate the bstack array!\n");
  exit(−10);
}
bytes += max_use * sizeof(tpair);
```

This code is used in section 44*.

46. ⟨ Copy all the temporary variable nodes to the *vmem* array in proper format 46 ⟩ ≡

```
for (c = vars; c; c−−) {
  ⟨ Move cur_tmp_var backward to the previous temporary variable 21 ⟩;
  o, vmem[c].name.lng = cur_tmp_var→name.lng;
  o, vmem[c].len = vars + 1;    /* "infinitely long" prefix */
}
```

This code is used in section 37.

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

⟨ Check consistency 47 ⟩ ≡

```
if (cur_cell ≠ &cur_chunk→cell[0] ∨ cur_chunk→prev ≠ Λ ∨
      cur_tmp_var ≠ &cur_vchunk→var[0] ∨ cur_vchunk→prev ≠ Λ) confusion("consistency");
free(cur_chunk);
for (cur_vchunk = last_vchunk; cur_vchunk; cur_vchunk = last_vchunk) {
  last_vchunk = cur_vchunk→prev;
  free(cur_vchunk);
}
```

This code is used in section 37.

48. Buddy system redux. Here's a version of Algorithms 2.5R and 2.5D that is appropriate for the operations we need to do in *bimp*.

Each block of *mem* has size 2^k for some $k > 1$, and it begins at an address that is a multiple of 2^k. A reserved block begins with an unsigned **int** that is less than 2^{31}; a free block begins with an unsigned **int** that is $\geq 2^{31}$ (thus its "sign" bit is 1). In fact, the first two words of the free block starting at b are the complements of pointers in a doubly linked list, and we call them *linkf* and *linkb*. The third word of such a block, called *kval*, contains the value of k when the block size is 2^k; and the "buddy" of such a block b begins at location $b \oplus (1 \ll k)$. There is a doubly linked list for free blocks of each possible size 2^k, with header node *mem*[*avail*(k)].

When *mems* are counted, we assume that *linkf* and *linkb* are accessed simultaneously as part of the same octabyte.

We begin by allocating $1 \ll$ *memk_max* entries to the *mem* array. But we maintain a variable *memk* to record the fact that at most $1 \ll$ *memk* of those entries have been used so far. The lists of available space are relevant only for $1 < k <$ *memk*, and the statistics reported at the end of a run are calculated as if only $1 \ll$ *memk* entries had been allocated. The user should increase *memk_max* (with the '`m`' command-line parameter) when trying to solve a problem that needs an unusually large *mem*.

\#**define** *linkf*(b) *mem*[b]
\#**define** *linkb*(b) *mem*[(b) + 1]
\#**define** *kval*(b) *mem*[(b) + 2]
\#**define** *avail*(k) (((k) − 2) ≪ 2)
\#**define** *memfree*(b) ((**int**) *mem*[b] < 0)
\#**define** *memk_max_default* 22 /* allow 4 million items in *mem* by default */

⟨ Global variables 3* ⟩ +≡
  int *memk*; /* binary log of the number of spaces used so far in *mem* */

49. ⟨ Check the sanity of *bimp* and *mem* 49 ⟩ ≡

```
for (l = 2; l < badlit; l++) {
  la = bimp[l].addr, k = bimp[l].k;
  if (la & ((1 ≪ k) − 1))
    fprintf(stderr, "addr␣of␣bimp["O"d]␣is␣clobbered␣(0x"O"x,␣k="O"d)!\n", l, la, k);
  else if (bimp[l].alloc ≠ 1 ≪ k)
    fprintf(stderr, "alloc␣of␣bimp["O"d]␣is␣clobbered␣("O"d,␣k="O"d)!\n", l, bimp[l].alloc, k);
  else if (bimp[l].size > bimp[l].alloc) fprintf(stderr,
        "size␣of␣bimp["O"d]␣is␣clobbered␣("O"d>"O"d)!\n", l, bimp[l].size, bimp[l].alloc);
  else if (la ≥ 1 ≪ memk) fprintf(stderr,
        "addr␣of␣bimp["O"d]␣is␣out␣of␣bounds␣(0x"O"d>0x"O"d)!\n", l, la, 1 ≪ memk);
  else if (memfree(la)) fprintf(stderr, "block␣0x"O"x␣of␣bimp["O"d]␣isn't␣reserved!\n", la, l);
  else
    for (j = bimp[l].size − 1; j ≥ 0; j−−)
      if (mem[la + j] < 2 ∨ mem[la + j] ≥ badlit)
        fprintf(stderr, "literal␣"O"d␣in␣bimp["O"d]␣is␣out␣of␣bounds!\n", mem[la + j], l);
}
for (k = 2; k < memk; k++) {
  for (p = ∼mem[avail(k)]; ; p = ∼linkf(p)) {
    if ((p & ((1 ≪ k) − 1)) ∧ p ≠ avail(k))
      fprintf(stderr, "link␣in␣avail("O"d)␣is␣clobbered␣(0x"O"x)!\n", k, p);
    else if (p ≥ 1 ≪ memk) fprintf(stderr,
          "link␣in␣avail("O"d)␣is␣out␣of␣bounds␣(0x"O"d>0x"O"d)!\n", k, p, 1 ≪ memk);
    else if (kval(p) ≠ k)
      fprintf(stderr, "kval␣of␣0x"O"x␣in␣avail("O"d)␣is␣"O"d!\n", p, k, kval(p));
    else if (memfree(p ⊕ (1 ≪ k)) ∧ kval(p ⊕ (1 ≪ k)) ≡ k)
      fprintf(stderr, "buddy␣of␣0x"O"x␣in␣avail("O"d)␣is␣also␣in␣that␣list!\n", p, k);
    else if (∼linkf(∼linkb(p)) ≠ p)
      fprintf(stderr, "linking␣anomaly␣at␣0x"O"x␣in␣avail("O"d)!\n", p, k);
    if (∼linkf(p) ≡ avail(k)) break;
  }
}
```

This code is used in section 31*.

50. The *resize* procedure does the main work of dynamic storage allocation. Given a literal l, it doubles the current allocation $bimp[l].alloc$.

Two cases are distinguished, depending on whether the buddy of l's current list is presently free or reserved. The buddy of a reserved block of size $1 \ll k$ might have been split up into smaller blocks, but it won't be any bigger.

⟨ Subroutines 29 ⟩ +≡

```
void resize(register int l)
{
  register uint a, j, k, kk, n, p, q, r, s;
  mems += 4;    /* pay the cost of subroutine linkage */
  oo, a = bimp[l].addr, n = bimp[l].size, k = bimp[l].k, s = 1 ≪ k, p = a ⊕ s;
  if ((o, memfree(p)) ∧ (o, kval(p) ≡ k)) ⟨ Resize when the buddy is free 51 ⟩
  else ⟨ Resize when the buddy is reserved 53 ⟩;
finish: o, bimp[l].alloc = s + s, bimp[l].k = k + 1;
}
```

51. Here the buddy of block a is p, and it has turned out to be free. In the most favorable case, p will actually be in exactly the right place so that we won't have to recopy any data.

⟨ Resize when the buddy is free 51 ⟩ ≡
```
{
  ⟨ Remove p from its avail list 52 ⟩;
  if ((a & s) ≡ 0) goto finish;   /* we lucked out */
  oo, mem[p] = mem[a];   /* ensure that mem[p] isn't negative */
  for (j = 1; j < n; j++) oo, mem[p + j] = mem[a + j];   /* copy the rest of the data */
  o, bimp[l].addr = p;
}
```
This code is used in section 50.

52. ⟨ Remove p from its *avail* list 52 ⟩ ≡
```
  q = ∼linkb(p), r = ∼linkf(p);   /* no mem cost, we've already accessed mem[p] */
  oo, linkf(q) = ∼r, linkb(r) = ∼q;
```
This code is used in sections 51 and 54.

53. In the more difficult case, we must find a block of twice the size, and copy the data there; then we free up the present block.

⟨ Resize when the buddy is reserved 53 ⟩ ≡
```
{
  ⟨ Allocate a block p of size s + s 54 ⟩;
  oo, mem[p] = mem[a];   /* ensure that mem[p] isn't negative */
  for (j = 1; j < n; j++) oo, mem[p + j] = mem[a + j];   /* copy the rest of the data */
  ⟨ Make a a free block of size 1 ≪ k 56 ⟩;
  o, bimp[l].addr = p;
}
```
This code is used in section 50.

54. ⟨ Allocate a block p of size $s + s$ 54 ⟩ ≡
```
  for (kk = k + 1; kk < memk; kk++)
    if (o, linkf(avail(kk)) ≠ ∼avail(kk)) {   /* nonempty list found */
      p = ∼linkf(avail(kk));
      o; ⟨ Remove p from its avail list 52 ⟩;
      goto found;
    }
  if (memk ≡ memk_max) {   /* oops, we're outta room */
    fprintf(stderr, "Sorry...␣more␣memory␣is␣needed!␣(Try␣option␣m"O"d.)\n", memk_max + 1);
    fprintf(stderr, "Job␣aborted␣after␣"O"llu␣mems,␣"O"llu␣nodes.\n", mems, nodes);
    exit(−666);
  }
  p = 1 ≪ memk;
  o, linkf(avail(memk)) = linkb(avail(memk)) = ∼avail(memk);   /* empty avail list */
  o, kval(avail(memk)) = memk;
  bytes += p ∗ sizeof(uint), memk++;
found:   /* location p begins an available block of size 1 ≪ kk */
  while (−−kk > k) ⟨ Make p + (1 ≪ kk) a free block of size 1 ≪ kk 55 ⟩;
```
This code is used in section 53.

55. ⟨ Make $p + (1 \ll kk)$ a free block of size $1 \ll kk$ 55 ⟩ ≡

```
{
  o, q = ~linkf(avail(kk)), r = p + (1 << kk);
  oo, linkf(avail(kk)) = linkb(q) = ~r;
  oo, linkb(r) = ~avail(kk), linkf(r) = ~q, kval(r) = kk;
}
```

This code is used in section 54.

56. Since the buddy of a is not free, we needn't try to "collapse" adjacent buddies together.

⟨ Make a a free block of size $1 \ll k$ 56 ⟩ ≡

```
o, q = ~linkf(avail(k));
oo, linkf(avail(k)) = linkb(q) = ~a;
oo, linkb(a) = ~avail(k), linkf(a) = ~q, kval(a) = k;
```

This code is used in section 53.

57. We need to get these data structures off to a good start at the very beginning. Here's how that is done, given *lits* and *memk_max*, after the arrays *mem* and *bimp* have been allocated:

⟨ Initialize *mem* with empty *bimp* lists 57 ⟩ ≡

```
for (memk = 4; 1 << memk < 4 * (memk_max - 2 + lits); memk++) ;
if (memk > memk_max) {   /* memk_max is too small even for empty lists! */
  fprintf(stderr, "The value of memk_max is way too small for "O"d literals!\n", lits);
  exit(-667);
}
mem = (uint *) malloc((1 << memk_max) * sizeof(uint));
if (¬mem) {
  fprintf(stderr, "Oops, I can't allocate the mem array!\n");
  exit(-10);
}
bytes += (1 << memk) * sizeof(uint);   /* we'll update bytes if we use more */
j = avail(memk_max);   /* the first bimp list starts here */
for (l = 2; l < badlit; l++) {
  oo, mem[j] = 0, bimp[l].addr = j, bimp[l].size = 0, j += 4;   /* reserve an empty block */
  o, bimp[l].alloc = 4, bimp[l].k = 2;   /* give it the minimum size */
}
for (k = 2; k < memk; k++) {
  if (j & (1 << k)) {   /* make a free block of size 1 << k at j */
    o, linkf(avail(k)) = linkb(avail(k)) = ~j;
    o, linkf(j) = linkb(j) = ~avail(k);
    oo, kval(avail(k)) = kval(j) = k;
    j += 1 << k;
  }
  else {   /* there are no free blocks of size 1 << k initially */
    o, linkf(avail(k)) = linkb(avail(k)) = ~avail(k);
    o, kval(avail(k)) = k;
  }
}
```

This code is used in section 38*.

58. The *istack* can grow rather large in the worst case. But it can't exceed the size of *mem*, since each entry in *istack* represents an increase in a *bimp* table entry. Therefore we allocate it with the same kludge that we used for *mem*.

⟨ Allocate special arrays 58 ⟩ ≡

```
istack = (idata *) malloc((1 << memk_max) * sizeof(idata));
if (¬istack) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣istack␣array!\n");
  exit(−10);
}
bytes += (1 << memk) * sizeof(idata);    /* we'll update bytes if we use more */
iptr_max = 1 << memk;
```

See also sections 90, 109, 121, 134, and 165*.

This code is used in section 37.

59. Updating the data structures. When we've decided to assign a value to a literal, we must deduce and record all of the consequences of that decision. The following part of the program comes into play when we're beginning the calculation at a new node of the decision tree.

Sometimes *bestlit* turns out to be zero, because the favorite literal of the lookahead process has already become true by forcing. Then we have a "dummy" level, which does no branching and inaugurates a new node from which we can look further ahead.

⟨ Begin the processing of a new node 59 ⟩ ≡
```
  nstack[level].lptr = rptr, nodes++;     /* for diagnostics only (no mem charged) */
  if (delta ∧ (mems ≥ thresh)) thresh += delta, print_state(level);
  if (mems > timeout) {
    fprintf(stderr, "TIMEOUT!\n");
    goto done;
  }
  o, nstack[level].branch = −1, plevel = level;
  ⟨ Look ahead and gather data about how to make the next branch; but goto look_bad if a contradiction
        arises 123* ⟩;
  if (forcedlits) ⟨ Update data structures for all consequences of the forced literals discovered during the
          lookahead; but goto conflict if a contradiction arises 64 ⟩;
chooseit: ⟨ Choose bestlit, which will be the next branch tried 140 ⟩;
  o, nstack[level].rptr = rptr, nstack[level].iptr = iptr;     /* backup pointers */
  if (bestlit) {
    o, nstack[level].decision = bestlit, nstack[level].branch = 0;
  tryit: l = bestlit, plevel = level + 1;
    if ((verbose & show_choices) ∧ level ≤ show_choices_max)
      fprintf(stderr, "Level␣"O"d"O"s:␣"O"s"O".8s␣("O"lld␣mems)\n", level,
          nstack[level].branch ? "’" : "", litname(l), mems);
    ⟨ Update data structures for all consequences of l; but goto conflict if a contradiction arises 62 ⟩;
  } else if ((verbose & show_choices) ∧ level ≤ show_choices_max)
    fprintf(stderr, "Level␣"O"d:␣no␣branch\n", level);
```
This code is used in section 152*.

60. Recall that the "current stamp" *cs* is an even number that represents the level of truth for assignments that are currently being made. Any variable x with $stamp[x] < cs$ is assumed to be free (unassigned); otherwise x is assumed to be true, in the context of level *cs*, when $stamp[x]$ is even, false when $stamp[x]$ is odd.

The highest level of truth is called *real_truth*; the next highest is *near_truth*; the next highest is *proto_truth*; and lower values 2, 4, ..., $proto\_truth - 2$ are used during lookahead.

```
#define real_truth #fffffffe
#define near_truth #fffffffc
#define proto_truth #fffffffa
#define isfixed(l) (o, stamp[thevar(l)] ≥ cs)
#define isfree(l) (o, stamp[thevar(l)] < real_truth)
#define iscontrary(l) ((stamp[thevar(l)] ⊕ l) & 1)     /* test this after isfixed(l) */
#define stamptrue(l) (o, stamp[thevar(l)] = cs + (l & 1))
```

⟨ Global variables 3* ⟩ +≡
```
  uint bestlit;     /* literal chosen for branching by lookahead routines */
  uint cs;     /* the current level of truth (always even) */
  uint look_cs, dlook_cs;     /* saved values of cs */
  int fptr, eptr, lfptr;     /* queue pointers for breadth-first search */
```

61. Here's a simple routine for use in debugging. It prints out all literals that are true with respect to a given stamping level.

```
⟨Subroutines 29⟩ +≡
  void print_truths(uint cs)
  {
    register int x;
    if (cs ≥ proto_truth) {
      switch ((cs − proto_truth) ≫ 1) {
      case 0: fprintf(stderr, "proto_truths␣or␣better:"); break;
      case 1: fprintf(stderr, "near_truths␣or␣better:"); break;
      case 2: fprintf(stderr, "real_truths:"); break;
      }
    } else fprintf(stderr, "truths␣at␣least␣"O"d:", cs);
    for (x = 1; x ≤ vars; x++)
      if (stamp[x] ≥ cs) fprintf(stderr, "␣"O"s"O".8s", stamp[x] & 1 ? "~" : "", vmem[x].name.ch8);
    fprintf(stderr, "\n");
  }
  void print_proto_truths(void)
  {
    print_truths(proto_truth);
  }
  void print_near_truths(void)
  {
    print_truths(near_truth);
  }
  void print_real_truths(void)
  {
    print_truths(real_truth);
  }
```

62. In the present part of the program, we set $cs = near\_truth$. This level means that the literal is on the *rstack* but its full consequences haven't yet been explored.

We do a breadth-first search, using *rstack* to contain the literals that are being asserted—first at level *near_truth*, then at level *real_truth*. Pointers *fptr* and *eptr* point to the front and end of the queue that governs the search.

```
⟨Update data structures for all consequences of l; but goto conflict if a contradiction arises 62⟩ ≡
  cs = near_truth;
  fptr = eptr = rptr;
  ⟨Bump istamp to a unique value 65⟩;
  ⟨Propagate binary implications of l; goto conflict if a contradiction arises 68*⟩;
promote: ⟨Promote near-truth to real-truth; but goto conflict if a contradiction arises 63⟩;
  if (o, nstack[level].branch < 0) {    /* we've finished the forced literals */
    if (level) goto chooseit;
    forcedlits = 0;
    goto enter_level;    /* at the root, it's back to square zero */
  }
```

This code is used in section 59.

63. ⟨ Promote near-truth to real-truth; but **goto** *conflict* if a contradiction arises 63 ⟩ ≡

```
while (fptr < eptr) {
  o, ll = rstack[fptr++];
  ⟨ Update data structures for the real truth of ll; but goto conflict if a contradiction arises 69* ⟩;
}
rptr = eptr;   /* accept all the propagations */
```

This code is used in section 62.

64. The forced literals act as "seeds" for another bread-first search.

If the input had unary clauses, the computation actually begins here, so that the implications of those clauses are perceived early.

⟨ Update data structures for all consequences of the forced literals discovered during the lookahead; but **goto** *conflict* if a contradiction arises 64 ⟩ ≡

```
{
special_start: if (verbose & show_details)
    fprintf(stderr, "(lookahead␣for␣level␣"O"d␣forces␣"O"d)\n", level, forcedlits);
  cs = near_truth;
  fptr = eptr = rptr;
  ⟨ Bump istamp to a unique value 65 ⟩;
  for (i = 0; i < forcedlits; i++) {
    o, l = forcedlit[i];
    ⟨ Propagate binary implications of l; goto conflict if a contradiction arises 68* ⟩;
  }
  goto promote;
}
```

This code is used in section 59.

65. The *istamp* field of literal l is marked with the current value of the global variable *istamp* when l gets its first *istack* entry during a particular phase of the search; then we can be sure that there's at most one *istack* entry per literal during any particular phase.

The loop here is "never" needed, except in problems that are well beyond what I ever imagine trying to solve. But I'm including it anyway, because it makes me feel virtuous.

⟨ Bump *istamp* to a unique value 65 ⟩ ≡

```
if (++istamp ≡ 0) {   /* overflow has occurred after 2^32 times */
  istamp = 1;
  for (l = 2; l < badlit; l++) o, lmem[l].istamp = 0;
}
```

This code is used in sections 62 and 64.

66. The *bstamp* field of literal l is similar to *istamp*, but it is used for a different purpose: We mark it when l is known to be implied by some other literal of interest.

⟨ Bump *bstamp* to a unique value 66 ⟩ ≡

```
if (++bstamp ≡ 0) {   /* overflow has occurred after 2^32 times */
  bstamp = 1;
  for (l = 2; l < badlit; l++) o, lmem[l].bstamp = 0;
}
```

This code is used in sections 73 and 106.

67. ⟨ Global variables 3* ⟩ +≡

```
uint istamp;   /* used for unique identifications */
uint bstamp = 32;   /* used for unique identifications of another kind */
```

68*. The code in this section is part of the inner loop, so we want it to be fast. Fortunately the task is fairly simple: When one literal is asserted to be true at the current *cs* level, all the literals in its *bimp* list are also asserted. And we continue until no more can be asserted, unless a contradiction arises first.

Our data structures contain both binary implications and k-ary implications for $k \ge 3$. We examine only the binary ones here, because they're simpler. By focusing on them first, we have a better chance of detecting contradictions sooner.

⟨Propagate binary implications of l; **goto** *conflict* if a contradiction arises 68*⟩ ≡

```
if (isfixed(l)) {
  if (iscontrary(l)) goto conflict;
} else {
  if (verbose & show_details) fprintf(stderr, "nearfixing␣"O"s"O".8s\n", litname(l));
  stamptrue(l);
  lfptr = eptr;
  o, rstack[eptr++] = l;
  while (lfptr < eptr) {
    o, l = rstack[lfptr++];
    for (o, la = bimp[l].addr, ls = bimp[l].size; ls; la++, ls--) {
      o, lp = mem[la];
      if (isfixed(lp)) {
        if (iscontrary(lp)) goto conflict;
      } else {
        if (verbose & show_details) fprintf(stderr, "␣nearfixing␣"O"s"O".8s\n", litname(lp));
        stamptrue(lp);
        o, rstack[eptr++] = lp;
      }
    }
  }
}
```

This code is used in sections 62, 64, 72*, and 73.

69*. We get to this part of the program when a literal loses its freedom and becomes fully assigned to truth or falsity at the highest possible level. Every active big clause that contains *ll* or its complement is affected: Those with *ll* itself become satisfied, while those with *bar*(*ll*) become shorter.

Many details of that transformation are described in the special "big clauses" addendum at the end of this program. Here we introduce only a few of them.

⟨Update data structures for the real truth of *ll*; but **goto** *conflict* if a contradiction arises 69*⟩ ≡

```
o, stamp[thevar(ll)] = real_truth + (ll & 1);
if (verbose & show_details) fprintf(stderr, "fixing␣"O"s"O".8s\n", litname(ll));
⟨Remove thevar(ll) from the freevar list 70⟩;
⟨Swap out all big clauses that contain ll 156*⟩;
tll = bar(ll), bptr = 0;    /* clear the bstack */
⟨Reduce all big clauses that contain tll; if any become binary, swap them out and put them on
    bstack 71*⟩;
while (bptr) {
  o, bptr--, u = bstack[bptr].u, v = bstack[bptr].v;
  ⟨Update for a potentially new binary clause u ∨ v 72*⟩;
}
```

This code is used in section 63.

70. ⟨ Remove *thevar*(*ll*) from the *freevar* list 70 ⟩ ≡

```
x = thevar(ll);
o, y = freevar[--freevars];
if (x ≠ y) {
  o, xl = freeloc[x];
  o, freevar[xl] = y;
  o, freeloc[y] = xl;
  o, freeloc[x] = freevars;
  o, freevar[freevars] = x;
}
```

This code is used in section 69*.

71*. When *tll* becomes false in clause *c*, we simply decrease the size of *c* by 1, without taking time to move *tll* to a different place in *cmem*. The first time this happens to *c* is, however, special: Then we also want to mark all of *c*'s other literals as "participants," as explained in the preselection process below. That case can be recognized by the condition *cinx*[*c*].*addr* + *cinx*[*c*].*size* = *cinx*[*c* − 1].*addr*. While we're examining those other literals, we might as well move *tll* to the end of the clause.

Interesting things start to happen when all but two of *c*'s literals have been falsified, before any of them have become true. At that point *c* becomes inactive and its remaining literals yield a new binary clause.

⟨ Reduce all big clauses that contain *tll*; if any become binary, swap them out and put them on *bstack* 71* ⟩ ≡

```
if (verbose & show_details) fprintf(stderr, "␣("O"s"O".8s␣out)\n", litname(tll));
for (o, tla = kinx[tll].addr, tls = kinx[tll].size; tls; tla++, tls--) {
  oo, c = kmem[tla], cia = cinx[c].addr, cis = cinx[c].size;
  if (o, cia + cis ≡ cinx[c − 1].addr) {    /* c is reduced for the first time */
    for (ua = cia, su = cis; su; ua++, su--) {
      o, u = cmem[ua];
      if (u ≡ tll) au = ua;
      else ⟨ Record thevar(u) as a participant 86* ⟩;
    }
    if (u ≠ tll) oo, cmem[ua − 1] = tll, cmem[au] = u;
  }
  o, cinx[c].size = cis − 1;
  if (cis ≡ 3) {    /* exactly two literals of c are now free */
    for (ci = cia, v = cmem[ci]; ; ci++) {
      o, u = cmem[ci];
      if (isfree(u)) break;
    }
    if (ci ≠ cia) oo, cmem[cia] = u, cmem[ci] = v;
    for (ci++; ; ci++) {
      o, v = cmem[ci];
      if (isfree(v)) break;
    }
    if (ci ≠ cia + 1) ooo, cmem[ci] = cmem[cia + 1], cmem[cia + 1] = v;
    o, bstack[bptr].u = u, bstack[bptr].v = v, bptr++;
    if (verbose & show_details) fprintf(stderr, "␣␣"O"s"O".8s->"O"s"O".8s|"O"s"O".8s\n",
        litname(bar(tll)), litname(u), litname(v));
    ⟨ Swap c out of u's clause list 158* ⟩;
    u = v; ⟨ Swap c out of u's clause list 158* ⟩;
  }
}
```

This code is used in section 69*.

72*. When a big clause reduces to the binary clause $u \lor v$, the “real” truth status of u and v is not yet known; but they might be “nearly” true or false. (In the latter case, we’ll be setting them really true or false as we continue our breadth-first search in the queue on the *rstack*.) There are five possibilities:

- If either u or v is near-true, the binary clause is satisfied and we needn’t do anything.
- If both u and v are near-false, we’ve reached a contradiction.
- If u is near-false but v is unknown, we can make v near-true.
- If u is unknown but v is near-false, we can make u near-true.
- Otherwise u and v are both unknown, and we’ve deduced the clause $u \lor v$.

⟨ Update for a potentially new binary clause $u \lor v$ 72* ⟩ ≡

```
if (isfixed(u)) {      /* equivalently, if (o, stamp[thevar(u)] ≥ near_truth) */
  if (iscontrary(u)) {      /* u is stamped false */
    if (isfixed(v)) {
      if (iscontrary(v)) goto conflict;
    } else {      /* v is unknown */
      l = v;
      ⟨ Propagate binary implications of l; goto conflict if a contradiction arises 68* ⟩;
    }
  }
} else {      /* u is unknown */
  if (isfixed(v)) {
    if (iscontrary(v)) {
      l = u;
      ⟨ Propagate binary implications of l; goto conflict if a contradiction arises 68* ⟩;
    }
  } else ⟨ Update for a new binary clause u ∨ v 73 ⟩;
}
```

This code is used in section 69*.

73. Now we've made some definite progress, by deducing a "new" binary clause $u \lor v$, and we hope to capitalize on it. Three opportunities, not mutually exclusive, may present themselves at this point:

- If $\bar u \lor v$ is already in our *bimp* table, we can make v near-true.
- If $u \lor \bar v$ is already in our *bimp* table, we can make u near-true.
- If $u \lor v$ is not already in our *bimp* table, we can insert it.

Furthermore, we might also know the clause $\bar v \lor w$, say, in which case the binary clause $u \lor w$ is also true. Experience shows that such "compensation resolvents" are useful, so we add them to our *bimp* collection.

This is the part of the program where we use *bstamp* to mark everything that's presently implied by $\bar u$. And then we use it to mark everything that's presently implied by $\bar v$.

An attentive reader will notice that, if $\bar u \lor v$ and $u \lor \bar v$ are both already in *bimp*, we'll make u near-true and the propagation routine will take care of v.

⟨Update for a new binary clause $u \lor v$ 73⟩ ≡
```
{
  ⟨Bump bstamp to a unique value 66⟩;
  o, lmem[bar(u)].bstamp = bstamp;
  for (o, au = bimp[bar(u)].addr, k = su = bimp[bar(u)].size; k; au++, k--)
    oo, lmem[mem[au]].bstamp = bstamp;
  if (o, lmem[bar(v)].bstamp ≡ bstamp) {      /* we already have u ∨ v̄ */
  fix_u: l = u; ⟨Propagate binary implications of l; goto conflict if a contradiction arises 68*⟩;
  } else if (o, lmem[v].bstamp ≠ bstamp) {      /* we don't have u ∨ v */
    o, ua = bimp[bar(u)].alloc;
    ⟨Make sure that bar(u) has an istack entry 74⟩;
    ⟨Add compensation resolvents from bar(u); but goto fix_u if u is forced true 76⟩;
    ⟨Bump bstamp to a unique value 66⟩;
    o, lmem[bar(v)].bstamp = bstamp;
    for (o, av = bimp[bar(v)].addr, k = sv = bimp[bar(v)].size; k; av++, k--)
      oo, lmem[mem[av]].bstamp = bstamp;
    if (o, lmem[bar(u)].bstamp ≡ bstamp) {      /* we already have ū ∨ v */
    fix_v: l = v; ⟨Propagate binary implications of l; goto conflict if a contradiction arises 68*⟩;
    } else {
      o, va = bimp[bar(v)].alloc;
      ⟨Make sure that bar(v) has an istack entry 77⟩;
      ⟨Add compensation resolvents from bar(v); but goto fix_v if v is forced true 79⟩;
      if (su ≡ ua) resize(bar(u)), ua += ua, o, au = bimp[bar(u)].addr + su;
      oo, mem[au] = v, bimp[bar(u)].size = su + 1;      /* ū implies v */
      if (sv ≡ va) resize(bar(v)), va += va, o, av = bimp[bar(v)].addr + sv;
      oo, mem[av] = u, bimp[bar(v)].size = sv + 1;      /* v̄ implies u */
    }
  }
}
```
This code is used in section 72*.

74. At this point $su = bimp[bar(u)].size$.

⟨Make sure that $bar(u)$ has an *istack* entry 74⟩ ≡
```
if (o, lmem[bar(u)].istamp ≠ istamp) {
  o, lmem[bar(u)].istamp = istamp;
  o, istack[iptr].lit = bar(u), istack[iptr].size = su;
  ⟨Increase iptr 75⟩;
}
```
This code is used in sections 73, 128*, and 137.

75. ⟨Increase *iptr* 75⟩ ≡
    $iptr{+}{+}$;
    if $(iptr \equiv iptr\_max)$ {
        $bytes \mathrel{+}= iptr * \textbf{sizeof}(\textbf{idata})$;
        $iptr\_max \lll= 1$;
    }

This code is used in sections 74, 77, 78, and 138.

76. At this point all implications of $bar(u)$ are stamped with *bstamp*, including $bar(u)$ itself. And since $u \vee v$ is true, we know that v is also implied by $bar(u)$. Therefore any literal w implied by v is a potentially new consequence of $bar(u)$, called a "compensation resolvent." (It can be obtained by resolving $u \vee v$ with $\bar{v} \vee w$.) Notice that w cannot be near-false; otherwise the propagation routine would have made v near-false, since $v \to w$ implies $\bar{w} \to \bar{v}$.

We maintain the values $au = bimp[bar(u)].addr + su$, $su = bimp[bar(u)].size$, $ua = bimp[bar(u)].alloc$.

⟨Add compensation resolvents from $bar(u)$; but **goto** *fix_u* if u is forced true 76⟩ ≡
    for $(o, la = bimp[v].addr, ls = bimp[v].size;\ ls;\ la{+}{+}, ls{-}{-})$ {
        $o, w = mem[la]$;
        if $(\neg isfixed(w))$ {
            if $(o, lmem[bar(w)].bstamp \equiv bstamp)$ **goto** *fix_u*;   /∗ $\bar{u}$ implies w and $\bar{w}$ ∗/
            if $(o, lmem[w].bstamp \neq bstamp)$ {   /∗ $u \vee w$ is new ∗/
                if $(verbose\ \&\ show\_details)$
                    fprintf(*stderr*, "␣␣␣->"O"s"O".8s|"O"s"O".8s\n", $litname(u), litname(w)$);
                if $(su \equiv ua)$ $resize(bar(u)), ua \mathrel{+}= ua, o, au = bimp[bar(u)].addr + su$;
                $oo, mem[au{+}{+}] = w, bimp[bar(u)].size = {+}{+}su$;   /∗ $\bar{u}$ implies w ∗/
                $o, aw = bimp[bar(w)].addr, sw = bimp[bar(w)].size$;
                ⟨Make sure that $bar(w)$ has an *istack* entry 78⟩;
                if $(o, sw \equiv bimp[bar(w)].alloc)$ $resize(bar(w)), o, aw = bimp[bar(w)].addr$;
                $o, bimp[bar(w)].size = sw + 1$;
                $o, mem[aw + sw] = u$;   /∗ $\bar{w}$ implies u ∗/
            }
        }
    }

This code is used in section 73.

77. At this point $sv = bimp[bar(v)].size$; we do for v as we did for u.

⟨Make sure that $bar(v)$ has an *istack* entry 77⟩ ≡
    if $(o, lmem[bar(v)].istamp \neq istamp)$ {
        $o, lmem[bar(v)].istamp = istamp$;
        $o, istack[iptr].lit = bar(v), istack[iptr].size = sv$;
        ⟨Increase *iptr* 75⟩;
    }

This code is used in section 73.

78. Here $sw = bimp[bar(w)].size$.

⟨Make sure that $bar(w)$ has an *istack* entry 78⟩ ≡
    if $(o, lmem[bar(w)].istamp \neq istamp)$ {
        $o, lmem[bar(w)].istamp = istamp$;
        $o, istack[iptr].lit = bar(w), istack[iptr].size = sw$;
        ⟨Increase *iptr* 75⟩;
    }

This code is used in sections 76 and 79.

79. This is the kind of program that cannot be written well when loud music is playing.

⟨ Add compensation resolvents from *bar*(*v*); but **goto** *fix_v* if *v* is forced true 79 ⟩ ≡

```
for (o, la = bimp[u].addr, ls = bimp[u].size; ls; la++, ls--) {
  o, w = mem[la];
  if (¬isfixed(w)) {
    if (o, lmem[bar(w)].bstamp ≡ bstamp) goto fix_v;    /* v̄ implies w and w̄ */
    if (o, lmem[w].bstamp ≠ bstamp) {    /* v ∨ w is new */
      if (verbose & show_details)
        fprintf(stderr, "␣␣␣->"O"s"O".8s|"O"s"O".8s\n", litname(v), litname(w));
      if (sv ≡ va) resize(bar(v)), va += va, o, av = bimp[bar(v)].addr + sv;
      oo, mem[av++] = w, bimp[bar(v)].size = ++sv;    /* v̄ implies w */
      o, aw = bimp[bar(w)].addr, sw = bimp[bar(w)].size;
      ⟨ Make sure that bar(w) has an istack entry 78 ⟩;
      if (o, sw ≡ bimp[bar(w)].alloc) resize(bar(w)), o, aw = bimp[bar(w)].addr;
      o, bimp[bar(w)].size = sw + 1;
      o, mem[aw + sw] = v;    /* w̄ implies v */
    }
  }
}
```

This code is used in section 73.

80. Downdating the data structures. When a contradiction arises, backtracking becomes necessary: Everything that went up must come down.

Fortunately the task of undoing isn't too tough. The *istack* contains all the information needed to discard any binary implications that no longer hold; and the *rstack* records every literal that has been made nearly or really true.

Let's look at the *istack* entries first, because they're so easy. The code almost writes itself.

⟨ Discard binary implications at the current level 80 ⟩ ≡

```
if (o, nstack[level].branch ≥ 0) {
  for (o, j = nstack[level].iptr; iptr > j; iptr−−) {
    o, l = istack[iptr − 1].lit, sl = istack[iptr − 1].size;
    o, bimp[l].size = sl;
  }
}
```

This code is used in section 84.

81. The *rstack* entries come in two parts, one easy and the other a bit tricky. The literals on *rstack*$[j]$ for $\mathit{fptr} \le j < \mathit{eptr}$ are the nice guys; they've become nearly true, but we haven't updated any serious consequences of that near-truth. Thus we merely need to unset those tentative assignments.

⟨ Unset the nearly true literals 81 ⟩ ≡

```
for (j = fptr; j < eptr; j++) oo, stamp[thevar(rstack[j])] = 0;
```

This code is used in section 84.

82*. The literals on *rstack*$[j]$ for $\mathit{rptr} \le j < \mathit{fptr}$ have become really true, and the ripple effects of those settings require more attention. Of principal importance is the fact that the big clauses in which those literals or their complements appear may have become inactive, in which case they've been swapped to the "invisible" part of the relevant *kinx* lists.

There's good news here: We don't need to unswap any of the *kinx* entries while we're backtracking! The order of those entries isn't important; only the state, active versus inactive, matters. The active entries are those that appear among the first *size* entries, beginning at *addr*. The inactive ones follow, in precisely the order in which they were swapped out, because a pair never participates in swaps after it has become inactive. Therefore we can reactivate the most-recently-swapped-out item in any particular list by simply increasing *size* by 1.

Two or more literals of the same clause may have all become really true or really false. We can be sure that the hocus pocus in the preceding paragraph works correctly if we are careful to do the virtual unswapping in precisely the reverse order from which we've done the swapping.

Similar reasoning applies to the list of free variables. When a literal left that list, we moved it from wherever it was in the early part of that list, by swapping it with the last currently free item, and then we decreased *freevars* by 1. To undo this operation, we simply increase *freevars* by 1. (The ordering isn't actually as critical here; it would suffice to change *freevars* once and for all by setting it to the value it had at the beginning of the node. But any savings in running time would be negligible.)

⟨ Unset the really true literals 82* ⟩ ≡

```
for (j = fptr − 1; j ≥ rptr; j−−) {      /* decreasing order is important */
  o, ll = rstack[j];
  tll = bar(ll);
  ⟨ Unreduce all big clauses that contain tll; if they had become binary, swap them back in 83* ⟩;
  ⟨ Swap in all big clauses that contain ll 159* ⟩;
  freevars++;
  o, stamp[thevar(ll)] = 0;
}
```

This code is used in section 84.

83*. ⟨ Unreduce all big clauses that contain *tll*; if they had become binary, swap them back in 83* ⟩ ≡

```
if (verbose & show_details) fprintf(stderr, " ("O"s"O".8s in)\n", litname(tll));
for (o, tls = kinx[tll].size, tla = kinx[tll].addr + tls − 1; tls; tla−−, tls−−) {
  o, c = kmem[tla];
  o, cia = cinx[c].addr, cis = cinx[c].size + 1;
  o, cinx[c].size = cis;
  if (cis ≡ 3) {
    o, u = cmem[cia]; ⟨ Swap c back in to u's clause list 160* ⟩;
    o, u = cmem[cia + 1]; ⟨ Swap c back in to u's clause list 160* ⟩;
  }
}
```

This code is used in section 82*.

84. ⟨ Recover from conflicts 84 ⟩ ≡

```
dl_contra: ⟨ Recover from a double lookahead contradiction 148 ⟩;
contra: ⟨ Recover from a lookahead contradiction 130* ⟩;
  goto look_bad;    /* a conflict has arisen during lookahead */
conflict: ⟨ Unset the nearly true literals 81 ⟩;
backtrack: ⟨ Unset the really true literals 82* ⟩;
  ⟨ Discard binary implications at the current level 80 ⟩;
  if (o, nstack[level].branch ≡ 0) ⟨ Move to branch 1 85 ⟩;
look_bad: if (level) {
    level−−;
    if (level < 31) prefix &= −(1 ≪ (31 − level));    /* see below */
    fptr = rptr;
    o, rptr = nstack[level].rptr;
    goto backtrack;
  }
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 153 ⟩;
  }
```

This code is used in section 152*.

85. A binary string is implicitly associated with every node of the search tree: At level 0, before we've done any branching at all, the string is empty. Branch 0 of every node appends 0 to the parent string, and branch 1 appends 1. The length of the string is therefore *level*. We also maintain the first 32 bits of the current string in the global variable *prefix*, left-justified within a 32-bit word. (This prefix is used to help guide locality of search, by identifying "participants" as explained in the preselection algorithm below.)

⟨ Move to branch 1 85 ⟩ ≡

```
  {
    bestlit = bar(nstack[level].decision);
    o, nstack[level].decision = bestlit, nstack[level].branch = 1;
    if (level < 32) prefix += 1 ≪ (31 − level);
    goto tryit;    /* if at first you don't succeed, try the other branch */
  }
```

This code is used in section 84.

86*. A variable x is said to "participate" at a branch node if it occurs in one of the nonbinary clauses that is produced in that node or in one of that node's ancestors. If x has already become a participant, the string specified by $vmem[x].pfx$ and $vmem[x].len$ will be a prefix of the current string.

In this step we update the *pfx* and *lev* fields of variables that are participating in the current activity. Notice that this information does not need to be changed when backtracking.

(At levels above 31 this program accepts cousins as well as ancestors.)

⟨ Record $thevar(u)$ as a participant 86* ⟩ ≡

```
{
  x = thevar(u);
  o, p = vmem[x].pfx, q = vmem[x].len;
  if (q < plevel) {
    t = prefix;
    if (q < 32) t &= -(1LL << (32 - q));      /* zero out irrelevant bits */
    if (p ≠ t) o, vmem[x].pfx = prefix, vmem[x].len = plevel;
  } else o, vmem[x].pfx = prefix, vmem[x].len = plevel;
}
```

This code is used in section 71*.

87. Preselection. The main purpose of lookahead is to choose the best free variable on which to branch. Of course we have limited foreknowledge, so we must make guesses. And we don't have time to explore *every* variable that remains free, except in trivial ways, unless we're near the root of the search tree.

So we begin the lookahead task by identifying a set of candidate variables that appear to be the most promising among all those that are currently free. That's called *preselection.*

⟨Do the prelookahead 87⟩ ≡
  if (*freevars* ≡ 0) **goto** *satisfied*;
  ⟨Preselect a set of candidate variables for lookahead 97*⟩;
  ⟨Determine the strong components; **goto** *look_bad* if there's a contradiction 104⟩;
  ⟨Construct a suitable forest 117⟩;

This code is used in section 123*.

88. The candidates are collected and identified in an array *cand*, whose entries have two fields, *var* and *rating*.

⟨Type definitions 5⟩ +≡
  typedef struct cdata_struct {
    uint *var*; /* the variable that's a candidate */
    float *rating*; /* its estimated importance */
  } **cdata**;

89. ⟨Global variables 3*⟩ +≡
  cdata ∗*cand*; /* list of candidates for lookahead */
  int *cands*; /* the number of candidates in *cand* */
  float *sum*; /* accumulator for computing the ratings */
  int *no_newbies*; /* are candidates restricted to participants? */
  float ∗*rating*; /* estimates of how useful each variable will be for branching */
  uint *prefix*; /* first 32 bits of the current prefix string */
  int *plevel*; /* length of the current prefix string */
  int *maxcand*; /* the maximum number of candidates desired at the current node */

90. ⟨Allocate special arrays 58⟩ +≡
  cand = (**cdata** ∗) *malloc*(*vars* ∗ **sizeof**(**cdata**));
  if (¬*cand*) {
    fprintf(*stderr*, "Oops,␣I␣can't␣allocate␣the␣cand␣array!\n");
    exit(−10);
  }
  bytes += *vars* ∗ **sizeof**(**cdata**);
  rating = (**float** ∗) *malloc*((*vars* + 1) ∗ **sizeof**(**float**));
  if (¬*rating*) {
    fprintf(*stderr*, "Oops,␣I␣can't␣allocate␣the␣rating␣array!\n");
    exit(−10);
  }
  bytes += (*vars* + 1) ∗ **sizeof**(**float**);

91.* The first stage of preselection *does* examine all the free variables, in order to get enough data to choose the candidates. Thus it constitutes one of the inner loops for which we hope to do everything rapidly. The general idea is to compute a heuristic score $h(l)$ for each free literal l, which estimates the relative amount by which asserting l will reduce the current problem.

92* An elaborate method is used in SAT11 for the case when all big clauses are ternary. But in the general k-ary case we will content ourselves with a very simple formula:

$$h(l) \;=\; \alpha + s(l) + \sum_{l \to l'} s(l'),$$

where $s(l)$ is the number of occurrences of $\bar{l}$ in big clauses that are currently active. This quantity $h(l)$ estimates the potential number of big-clause reductions that occur when l becomes true. The default value $\alpha = 0.001$ is recommended, but of course other magic values can be tried by using the command-line parameter '`a`'.

93* ⟨ Compute *sum*, the score of l 93* ⟩ ≡

```
{
  ullng acc;    /* an accumulator */
  o, acc = kinx[bar(l)].size;
  for (o, la = bimp[l].addr, ls = bimp[l].size; ls; la++, ls--) {
    o, u = mem[la];
    if (isfree(u)) acc += kinx[bar(u)].size;
  }
  sum = alpha + (float) acc;
}
```

This code is used in section 94*.

94* We don't actually need the individual scores $h(l)$ for each free literal l: Only the product $h(l)h(\bar{l})$ is used, as our rating for each free variable x.

⟨ Compute *rating*[x] 94* ⟩ ≡

```
{
  float s;
  l = poslit(x);
  ⟨ Compute sum, the score of l 93* ⟩;
  s = sum;
  l++;
  ⟨ Compute sum, the score of l 93* ⟩;
  rating[x] = s * sum;
  if (verbose & show_scores) fprintf(stderr, "("O".8s:␣pos␣"O".2f␣neg␣"O".2f␣r="O".4g)\n",
        vmem[x].name.ch8, s, sum, s * sum);
}
```

This code is used in section 95*.

95* ⟨ Put the ratings in *rating* 95* ⟩ ≡

```
for (k = 0; k < freevars; k++) {
  o, x = freevar[k];
  ⟨ Compute rating[x] 94* ⟩;
}
```

This code is used in section 97*.

96*. The maximum number of candidates permitted, in this implementation, depends on the current level rather than on the number of variables or clauses in the problem: We calculate *maxcand* = the maximum of *levelcand*/*level* and *mincutoff*, where *levelcand* = 600 and *mincutoff* = 30 by default. (At level 0, for example, *maxcand* is infinite; at level 5 it is 120; at levels 20 or more it is 30.) Then, while $cands \ge 2 * maxcand$, we repeatedly remove all candidates whose rating is less than the mean; quite a few really weak candidates might therefore go away if a few strong ones dominate. Finally, if $maxcand < cands < 2 * maxcand$, we eliminate the $cands - maxcand$ candidates with smallest ratings.

That policy might seem peculiar, but it reflects the reality of combinatorial search problems: If the problem is easy, we don't care if we solve it in 2 seconds or .00002 seconds. On the other hand if the problem is so difficult that it can only be solved by looking ahead more than we can accomplish in a reasonable time, we might as well face the fact that we won't solve it anyway. (There's no point in looking ahead at 60 variables at depth 60, because we won't be able to deal with more than 2^{50} or so nodes in any reasonable search tree.)

97*. ⟨ Preselect a set of candidate variables for lookahead 97* ⟩ ≡

```
⟨ Put the ratings in rating 95* ⟩;
maxcand = (level ≡ 0 ? freevars : levelcand/level);
if (maxcand < mincutoff) maxcand = mincutoff;
⟨ Put all free participants into the initial list of candidates 98 ⟩;
⟨ Pare down the candidates to at most maxcand 101 ⟩;
```

This code is used in section 87.

98. The next stage in this winnowing-down process tries to avoid any variable that hasn't participated in a ternary clause that has been reduced; otherwise we might find ourselves trying to solve several independent problems at the same time. In order to weed out "newbies" (nonparticipants), we allow x to be a candidate only if *vmem*[x].*pfx* and *vmem*[x].*len* specify a string that's a prefix of the current node's string. (However, we rescind this restriction if it gives us no candidates. For example, at level 0 there are no participants, because we haven't reduced any clauses.)

If the V option is being used, to distinguish "primary" variables, we consider a nonprimary variable to be a nonparticipant (so that it will not normally become a candidate).

⟨ Put all free participants into the initial list of candidates 98 ⟩ ≡

```
  no_newbies = (plevel > 0);
init_cand: for (cands = k = 0, sum = 0.0; k < freevars; k++) {
    o, x = freevar[k];
    o, stamp[x] = 0;    /* erase all former assignments */
    if (no_newbies) {
      if (x > primary_vars) continue;
      o, t = vmem[x].pfx, l = vmem[x].len;
      if (l ≡ plevel) {
        if (t ≠ prefix) continue;    /* not a participant */
      } else if (l > plevel) continue;
      else if (t ≠ (l < 32 ? prefix & −(uint)(1_LL ≪ (32 − l)) : prefix)) continue;
    }
    oo, cand[cands].var = x, cand[cands].rating = rating[x];
    cands++, sum += rating[x];
  }
  if (cands ≡ 0) {
    ⟨ If all clauses are satisfied, goto satisfied 99 ⟩;
    no_newbies = 0;
    goto init_cand;    /* if there are no participants, accept all comers */
  }
```

This code is used in section 97*.

99. ⟨ If all clauses are satisfied, **goto** *satisfied* 99 ⟩ ≡

```
for (j = 0; j < freevars; j++) {
   o, x = freevar[j];
   l = poslit(x);
   ⟨ If l implies any unsatisfied clauses, goto nogood 100* ⟩;
   l++;
   ⟨ If l implies any unsatisfied clauses, goto nogood 100* ⟩;
}
goto satisfied;
nogood:
```

This code is used in section 98.

100*. ⟨ If l implies any unsatisfied clauses, **goto** *nogood* 100* ⟩ ≡

```
if (o, kinx[bar(l)].size) goto nogood;     /* all active kinxs are unsatisfied */
for (o, la = bimp[l].addr, ls = bimp[l].size; ls; la++, ls--) {
   o, u = mem[la];
   if (o, stamp[thevar(u)] ≠ real_truth + (u & 1)) goto nogood;
}
```

This code is used in section 99.

101. At this point we've got *cands* candidates in the *cand* array, and *sum* is the sum of their ratings. The next task is to eliminate low-rated candidates, if we have too many to handle.

⟨ Pare down the candidates to at most *maxcand* 101 ⟩ ≡

```
for (k = 1; cands ≥ 2 * maxcand ∧ k; ) {
   register float mean = 0.9999 * sum/(double) cands;
   for (j = k = 0, sum = 0.0; j < cands; ) {
      if (o, cand[j].rating ≥ mean) sum += cand[j].rating, j++;
      else oo, k = 1, cand[j] = cand[--cands];     /* don't advance j, discard a loser */
   }
}
if (cands > maxcand) ⟨ Select the maxcand best-rated candidates 102 ⟩;
if (cands ≡ 0) confusion("cands");
```

This code is used in section 97*.

102. Here we make the *cand* array into a heap, with low-rated elements in the lowest positions. Then we delete the ones we don't want. (See Algorithm 5.2.3H. The heap condition is

$$cand[i].rating \le cand[2*i+1].rating \qquad \text{and} \qquad cand[i].rating \le cand[2*i+2].rating$$

whenever the subscripts are nonnegative and less than *cands*.)

⟨Select the *maxcand* best-rated candidates 102⟩ ≡

```
{
  j = cands ≫ 1;    /* the heap condition holds for i ≥ j */
  while (j > 0) {
    j−−;
    ⟨Sift cand[j] up 103⟩;
  }
  while (1) {
    oo, cand[0] = cand[−− cands];    /* discard a loser */
    if (cands ≡ maxcand) break;
    ⟨Sift cand[j] up 103⟩;
  }
}
```

This code is used in section 101.

103. ⟨Sift *cand*[*j*] up 103⟩ ≡

```
{
  register float r;
  cdata c;
  o, c = cand[j], r = c.rating;
  for (i = j, jj = (j ≪ 1) + 1; jj < cands; i = jj, jj = (jj ≪ 1) + 1) {
    if (jj + 1 < cands ∧ (o, cand[jj + 1].rating < cand[jj].rating)) jj++;
    if (o, r ≤ cand[jj].rating) break;
    o, cand[i] = cand[jj];
  }
  if (i > j) o, cand[i] = c;
}
```

This code is used in section 102.

104. Strong components. If the binary implication graph has a nontrivial strong component, all literals in that component are locked together: Any one of their values determines all the rest. Therefore we don't want to bother looking ahead on two variables that have literals in the same strong component.

Robert Tarjan has devised a beautiful algorithm that finds the strong components very efficiently [*SIAM Journal on Computing* **1** (1972), 146–160]; and his algorithm also produces a topological sort on the representatives of those components, as an extra bonus. We are going to want the preselected candidates to be topologically sorted, because that will speed up the lookaheads that we'll be doing. Therefore Tarjan's algorithm is a perfect fit for our present situation.

Note: We are going to restrict ourselves to direct implications between candidates, instead of considering indirect chains of implications $l_0 \to l_1 \to \cdots \to l_k$ with $k > 1$, where l_0 and l_k are candidates but the intermediate literals $l_1, \ldots, l_{k-1}$ are not. The efficiency of Tarjan's algorithm suggests that we could consider the full digraph instead of its restriction to candidates only, perhaps before deciding on the list of candidates. However, cases in which indirect implications provide significant information appear to be rare. (At least, the author has yet to see a single instance where two chosen candidates, in the most time-consuming parts of a search tree, are implicitly linked without also being explicitly linked.) It seems that the variables chosen to be candidates almost never have important non-candidate neighbors.

The following implementation of Tarjan's algorithm follows the steps that appear on pages 513–519 of *The Stanford GraphBase*. The reader is referred to that book, which explains the procedure in terms of an explorer who searches the rooms of a cave, for full details and proofs of correctness.

The algorithm uses five integer fields in each literal's *lmem* record:

rank is initially 0, then positive, finally ∞, when l is respectively unseen, then active, finally settled.

parent points to a lower-ranked literal in the current oriented tree of active literals (or to 0 at the root), when l is active; it points to the component representative when l is settled.

untagged tells how many of l's successors haven't been explored.

link is a link in the stack of active vertices or the stack of settled vertices.

min is Tarjan's brilliant invention that makes everything work fast.

We add also a sixth field, *vcomp*, which is a component member of maximum rating.

Our instrumentation counts *mems* by assuming that *rank* and *link* are accessed simultaneously as an octabyte, as are *untagged* and *min*, *parent* and *vcomp*.

⟨Determine the strong components; **goto** *look_bad* if there's a contradiction 104⟩ ≡
  ⟨Make all vertices unseen and all arcs untagged 106⟩;
  for $(i = 0;\ i < cands;\ i{+}{+})$ {
    $o, l = poslit(cand[i].var)$;
  check_rank: **if** $(o, lmem[l].rank \equiv 0)$ ⟨Perform a depth-first search with l as root, finding the strong components of all vertices reachable from l 112⟩;
    if $((l \mathbin{\&} 1) \equiv 0)$ {
      $l{+}{+}$; **goto** *check_rank*;
    }
  }
  if $(verbose \mathbin{\&} show\_strong\_comps)$ ⟨Print the strong components 105⟩;

This code is used in section 87.

105. ⟨ Print the strong components 105 ⟩ ≡

```
{
  fprintf(stderr, "Strong␣components:\n");
  for (l = settled; l; l = lmem[l].link) {
    fprintf(stderr, "␣"O"s"O".8s␣", litname(l));
    if (lmem[l].parent ≠ l) fprintf(stderr, "with␣"O"s"O".8s\n", litname(lmem[l].parent));
    else {
      if (lmem[l].vcomp ≠ l) fprintf(stderr, "->␣"O"s"O".8s␣", litname(lmem[l].vcomp));
      fprintf(stderr, ""O".4g\n", rating[thevar(lmem[l].vcomp)]);
    }
  }
}
```

This code is used in section 104.

106. Candidates are marked with *bstamp* here so that they can be distinguished from non-candidates. Then we make a new copy of the *bimp* data, abbreviating it so that only the candidates are listed.

An arbitrary upper bound is placed on the total number of arcs in this reduced digraph, because perfect accuracy is not important at this stage. The default limit, *max_prelook_arcs* = 10000, can be changed if desired. Care is needed when we stick to such a limit, because we want the arc $u \to v$ to be present if and only if its dual $\bar{v} \to \bar{u}$ is also present.

⟨ Make all vertices unseen and all arcs untagged 106 ⟩ ≡

```
⟨ Bump bstamp to a unique value 66 ⟩;
for (i = 0; i < cands; i++) {
  o, l = poslit(cand[i].var);
  oo, lmem[l].rank = 0, lmem[l].arcs = −1, lmem[l].bstamp = bstamp;
  oo, lmem[l + 1].rank = 0, lmem[l + 1].arcs = −1, lmem[l + 1].bstamp = bstamp;
}
⟨ Copy all the relevant arcs to cand_arc 110 ⟩;
for (i = 0; i < cands; i++) {
  o, l = poslit(cand[i].var);
  oo, lmem[l].untagged = lmem[l].arcs;
  oo, lmem[l + 1].untagged = lmem[l + 1].arcs;
}
k = 0;    /* this is the number of vertices "seen" by Tarjan's algorithm */
active = settled = 0;    /* the active and settled stacks are empty */
```

This code is used in section 104.

107. ⟨ Type definitions 5 ⟩ +≡

```
typedef struct arc_struct {
  uint tip;    /* the implied literal */
  int next;    /* next arc from the implier literal, or −1 */
} arc;
```

108. ⟨ Global variables 3* ⟩ +≡

```
arc *cand_arc;    /* the arcs in a reduced digraph */
int cand_arc_alloc;    /* how many arc slots have we used so far? */
int active;    /* top of the linked stack of active vertices */
int settled;    /* top of the linked stack of settled vertices */
```

109. The number of *bytes* used will be adjusted dynamically.

⟨ Allocate special arrays 58 ⟩ +≡
$max\_prelook\_arcs$ &= −2; /* make sure $max\_prelook\_arcs$ is even */
$cand\_arc$ = (**arc** *) $malloc(max\_prelook\_arcs$ * **sizeof**(**arc**));
if (¬$cand\_arc$) {
  $fprintf(stderr$, "Oops,␣I␣can't␣allocate␣the␣cand_arc␣array!\n");
  $exit(-10)$;
}

110. ⟨ Copy all the relevant arcs to $cand\_arc$ 110 ⟩ ≡
for ($j = i = 0$; $i < cands$; i++) {
  $o, l = poslit(cand[i].var)$;
  ⟨ Copy the arcs from l into the $cand\_arc$ array 111 ⟩;
  l++;
  ⟨ Copy the arcs from l into the $cand\_arc$ array 111 ⟩;
}
$arcs\_done$: **if** ($j > cand\_arc\_alloc$) /* we've copied more arcs than ever before */
  $bytes$ += ($j - cand\_arc\_alloc$) * **sizeof**(**arc**), $cand\_arc\_alloc = j$;

This code is used in section 106.

111. Beware: We *reverse* the ordering here, placing an arc $u \to v$ into $cand\_arc$ when there's an implication $v \to u$ in the $bimp$ table. This switcheroo will produce strong components in a more desirable order.

⟨ Copy the arcs from l into the $cand\_arc$ array 111 ⟩ ≡
for ($oo, la = bimp[l].addr, ls = bimp[l].size, p = lmem[bar(l)].arcs$; ls; la++, ls−−) {
  $o, u = mem[la]$;
  if ($u < l$) **continue**; /* we enter arcs in pairs, only when $l < u$ */
  if ($o, lmem[u].bstamp \neq bstamp$) **continue**; /* not a candidate */
    /* now $l \to u$ is an implication, and $u > l$ */
  $o, cand\_arc[j].tip = bar(u), cand\_arc[j].next = p, p = j$; /* make arc $\bar{l} \to \bar{u}$ */
  $oo, cand\_arc[j+1].tip = l, cand\_arc[j+1].next = lmem[u].arcs$;
  $o, lmem[u].arcs = j + 1, j$ += 2; /* make arc $u \to l$ */
  if ($j \equiv max\_prelook\_arcs$) {
    if ($verbose$ & $show\_details$)
      $fprintf(stderr$, "prelook␣arcs␣cut␣off␣at␣"O"d;␣see␣option␣z\n", $max\_prelook\_arcs$);
    $o, lmem[bar(l)].arcs = lmem[bar(l)].untagged = p$;
    goto $arcs\_done$;
  }
}
$o, lmem[bar(l)].arcs = lmem[bar(l)].untagged = p$;

This code is used in section 110.

112. ⟨ Perform a depth-first search with l as root, finding the strong components of all vertices reachable from l 112 ⟩ ≡
{
  $v = l$;
  $o, lmem[l].parent = 0$;
  ⟨ Make vertex v active 113 ⟩;
  do ⟨ Explore one step from the current vertex v, possibly moving to another current vertex and calling it v 114 ⟩ **while** ($v > 0$);
}

This code is used in section 104.

113. ⟨ Make vertex v active 113 ⟩ ≡

```
o, lmem[v].rank = ++k;
lmem[v].link = active, active = v;
o, lmem[v].min = v;
```

This code is used in sections 112 and 114.

114. Minor point: No mem is charged for setting $lmem[v].min = u$ here, because $lmem[v].untagged$ could have been set at the same time.

⟨ Explore one step from the current vertex v, possibly moving to another current vertex and calling it v 114 ⟩ ≡

```
{
  o, vv = lmem[v].untagged, ll = lmem[v].min;
  if (vv ≥ 0) {    /* still more to explore from v */
    o, u = cand_arc[vv].tip, vv = cand_arc[vv].next;
    o, lmem[v].untagged = vv;
    o, j = lmem[u].rank;
    if (j) {    /* we've seen u already */
      if (o, j < lmem[ll].rank) lmem[v].min = u;    /* nontree arc, just update v's min */
    } else {    /* u is newly seen */
      lmem[u].parent = v;    /* a new tree arc goes v → u */
      v = u;    /* u will now be the current vertex */
      ⟨ Make vertex v active 113 ⟩;
    }
  } else {    /* v becomes mature */
    o, u = lmem[v].parent;
    if (v ≡ ll) ⟨ Remove v and all its successors on the active stack from the tree, and mark them as a
          strong component of the digraph 115 ⟩
    else {    /* the arc u → v has matured, making v's min visible from u */
      if (ooo, lmem[ll].rank < lmem[lmem[u].min].rank) o, lmem[u].min = ll;
    }
    v = u;    /* the former parent of v becomes the new current vertex v */
  }
}
```

This code is used in section 112.

115. When v is the representative of a strong component, all vertices of that component henceforth regard v as their parent.

If v represents the strong component of u and if w represents the strong component of $bar(u)$, we won't always have $w = bar(v)$. But we take pains to ensure that $lmem[v].vcomp = bar(lmem[w].vcomp)$.

```
#define infty badlit
```

⟨ Remove v and all its successors on the active stack from the tree, and mark them as a strong component of the digraph 115 ⟩ ≡

```
{
  float r, rr;
  t = active;
  o, r = rating[thevar(v)], w = v;
  o, active = lmem[v].link;
  o, lmem[v].rank = infty;   /* settle v */
  lmem[v].link = settled, settled = t;   /* move the component from active to settled */
  while (t ≠ v) {
    if (t ≡ bar(v)) {   /* component contains complementary literals */
      if (verbose & show_gory_details) fprintf(stderr, "the␣binary␣clauses␣are␣inconsistent\n");
      goto look_bad;
    }
    o, lmem[t].rank = infty;   /* now t is settled */
    o, lmem[t].parent = v;   /* and its strong component is represented by v */
    o, rr = rating[thevar(t)];
    if (rr > r) r = rr, w = t;
    o, t = lmem[t].link;
  }
  o, lmem[v].parent = v, lmem[v].vcomp = w;   /* v represents itself */
  if (o, lmem[bar(v)].rank ≡ infty) oo, lmem[v].vcomp = bar(lmem[lmem[bar(v)].parent].vcomp);
}
```

This code is used in section 114.

116. The lookahead forest. Now we come to what is probably the nicest part of this whole program, an elegant mechanism by which much of the potential lookahead computation is avoided.

Suppose we've decided to look ahead on the consequences of literals l_1, l_2, …, l_n, in that order. The current binary implications tell us that, if l_j is true, then also l_i must be true for certain i. If $i < j$, we've already deduced the consequences of l_i, so we prefer not to do that again. On the other hand l_j probably doesn't imply all of $l_1, \ldots, l_{i-1}$; so we want to be selective, to reuse only part of the information that we've already discovered.

The stamping principle provides a way to do that. Suppose $p_1p_2\ldots p_n$ is a permutation of $\{1,\ldots,n\}$, and suppose we stamp true/false values at level p_j when we are looking at consequences of l_j. Then, when l_j is current, the value of a literal will be considered unknown if its stamp is less than p_j, but it will be implied by l_j if it has been deduced by any of the previous literals l_i with $i < j$ and $p_i > p_j$.

If, for example, $n = 4$ and $p_1p_2p_3p_4 = 3\,1\,4\,2$, then l_2 can assume all consequences of l_1 (because $p_1 > p_2$); and l_4 can assume all of the consequences of l_1 and l_3, but not l_2 (because $p_1 > p_4$ and $p_3 > p_4$ but $p_2 < p_4$). This permutation captures the shortcuts that are legitimate when we have the implications $l_2 \to l_1$, $l_4 \to l_1$, and $l_4 \to l_3$.

A set of implications that can be defined by a permutation in this way is called a "permutation poset." When I first noticed this connection between permutation posets and stamping, I excitedly thought, "Aha! Permutation posets are ideal for lookahead in a SAT solver." Unfortunately, however, I soon learned that lookahead is much more subtle than I'd realized, and I was compelled to abandon that optimistic sentiment; my current thinking is, "Alas! Only a few permutation posets will work well for lookahead in a SAT solver."

The example above, which is based on the notorious pi-mutation $3\,1\,4\,2$, illustrates the problem if we examine it closely: When literal l_3 is processed, we don't want occurrences of $\bar{l}_1$ to be removed from the current clauses, because l_3 doesn't imply l_1. But when l_4 is processed, we do want $\bar{l}_1$ to be suppressed, as well as $\bar{l}_3$, because $l_4 \to l_1$ and $l_4 \to l_3$.

On the other hand the permutation $4\,1\,3\,2$ does lead to a good scenario. It corresponds to the dependencies $l_2 \to l_1$, $l_3 \to l_1$, $l_4 \to l_3$ (hence also $l_4 \to l_1$). Now l_3 can assume the consequences of l_1 (but not l_2), and we can remove $\bar{l}_1$ from the clauses when we work on l_3. Again l_4 can assume the consequences of l_1 and l_3 (but not l_2); and this time it's convenient to remove $\bar{l}_3$ from the clauses that have already been purged of $\bar{l}_1$. The point is that the purging of negative literals has the same implicit recursive structure as the visibility of stamps.

The permutations that work properly are those that don't contain a substring $a\,b\,c$ with $c < a < b$ (like the substring $3\,4\,2$ in $3\,1\,4\,2$). And such permutations are well known: They are the so-called *stack permutations*. [See *The Art of Computer Programming*, exercise 2.2.1–5. Actually our permutations are the reverses or the inverses of the stack permutations described there.] Moreover, they correspond precisely to dependencies that form an oriented forest, and the correspondence is also well known and quite nice: "If u and v are nodes of a forest, u is a proper ancestor of v if and only if u precedes v in preorder and u follows v in postorder" [*TAOCP* exercise 2.3.2–20].

In general we've chosen candidate literals with certain known dependencies. We would like to find an oriented forest, contained within those dependencies, having as many arcs as possible.

The task of finding the largest oriented forest contained in a given partially ordered set is probably NP-complete. But two things make our task feasible in practice. First, the number of variables for which we need to study dependencies is not very large, during the bulk of the calculations; it's at most a few dozen, except at shallow depth. Second, the dependencies aren't usually extensive; at most ten or so variables are in any connected component of the typical digraphs that arise. So we need only come up with a decent way to handle small examples. It doesn't matter if our subforests are crude in unusual cases.

117. When the program below begins its work, we will have reduced the strong components of the candidates' digraph and placed the component representatives into topological order. That order isn't necessarily the one we seek for the oriented forest, but it facilitates the computations we need to do. We use it to rank the literals in yet another way, this time by "height," namely by the length of a longest path from a source vertex. Then every literal u of height $h > 0$ has a predecessor vertex v of height $h - 1$. We will use the oriented forest that is defined by those predecessor links—using the fact that $v \to u$ is an implication in *bimp*[v] when u has an arc to v in the *cand_arc* digraph.

⟨ Construct a suitable forest 117 ⟩ ≡
  ⟨ Find the heights and the child/sibling links 118 ⟩;
  ⟨ Construct the *look* table 122 ⟩;

This code is used in section 87.

118. If u represents a strong component we will change *lmem*[u].*untagged* to a height value; and we'll also make *lmem*[u].*min* point to child of u in the forest being constructed. Those fields are therefore renamed *height* and *child*, to reflect their new function. The *link* fields will also acquire a new significance, although we'll keep calling them *link*: They will point to siblings in the forest, namely to vertices with the same parent.

The dummy literal 1 will play the role of a global root, whose children are all of the source vertices (the vertices of height 0).

```
#define height untagged
#define child min
#define root 1
```

⟨ Find the heights and the child/sibling links 118 ⟩ ≡

```
o, lmem[root].child = 0, lmem[root].height = −1, pp = root;
for (u = settled; u; u = uu) {
  oo, uu = lmem[u].link, p = lmem[u].parent;
  if (p ≠ pp) h = 0, w = root, pp = p;   /* pp is previous strong component representative */
  for (o, j = lmem[bar(u)].arcs; j ≥ 0; j = cand_arc[j].next) {
    o, v = bar(cand_arc[j].tip);   /* we look at the predecessors v of u */
    o, vv = lmem[v].parent;
    if (vv ≡ p) continue;   /* ignore an arc within the current component */
    o, hh = lmem[vv].height;
    if (hh ≥ h) h = hh + 1, w = vv;
  }
  if (p ≡ u) {
    o, v = lmem[w].child;
    oo, lmem[u].height = h, lmem[u].child = 0, lmem[u].link = v;
    o, lmem[w].child = u;
  }
}
```

This code is used in section 117.

119. The results of our oriented forest computation are placed into an array of *ldata* called *look*. The lookahead process will examine literals *look*[0].*lit*, *look*[1].*lit*, ..., *look*[*looks* − 1].*lit*, in that order; and the current stamp while studying the implications of *look*[*k*].*lit* will be the even number *base* + *look*[*k*].*offset*, where *base* is the smallest stamp in the current iteration.

(Cognoscenti will understand that there is one entry in this array for each strong component that was found in the implication digraph of candidates.)

⟨Type definitions 5⟩ +≡
```
typedef struct ldata_struct {
  uint lit;    /* a literal for lookahead */
  uint offset;    /* the offset of its stamp */
} ldata;
```

120. ⟨Global variables 3*⟩ +≡
```
ldata *look;    /* specification of the oriented forest for lookaheads */
int looks;    /* the number of current entries in look */
```

121. ⟨Allocate special arrays 58⟩ +≡
```
look = (ldata *) malloc(lits * sizeof(ldata));
if (¬look) {
  fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣look␣array!\n");
  exit(−10);
}
bytes += lits * sizeof(ldata);
```

122. Here's a standard "double order" traversal [*TAOCP* exercise 2.3.1–18] as we list the literals in preorder while filling in their offsets according to postorder.

We've constructed the tree using literals that are representatives of the strong components produced by Tarjan's algorithm. But the lookahead process will use the *vcomp* representatives instead.

⟨Construct the *look* table 122⟩ ≡
```
o, u = lmem[root].child, j = k = v = 0;
while (1) {
  oo, look[k].lit = lmem[u].vcomp;
  o, lmem[u].rank = k++;    /* k advances in preorder */
  if (o, lmem[u].child) {
    o, lmem[u].parent = v;    /* fix parent temporarily for traversal */
    v = u, u = lmem[u].child;    /* descend to u's descendants */
  } else {
  post: o, i = lmem[u].rank;
    o, look[i].offset = j, j += 2;    /* j advances in postorder */
    if (v) oo, lmem[u].parent = lmem[v].vcomp;    /* fix parent for lookahead */
    else o, lmem[u].parent = 0;
    if (o, lmem[u].link) u = lmem[u].link;    /* move to u's next sibling */
    else if (v) {
      o, u = v, v = lmem[u].parent;    /* after the last sibling, move to u's parent */
      goto post;
    } else break;
  }
}
looks = k;
if (j ≠ k + k) confusion("looks");
```
This code is used in section 117.

123*. Looking ahead. The lookahead process has much in common with what we do when making a decision at a branch node, except that we don't make drastic changes to the data structures. We don't assign any truth values at levels higher than *proto_truth*; and that level is reserved for literals that will be forced true if the lookahead procedure finds no contradictions. We don't create new binary implications when a ternary clause gets a false literal; we estimate the potential benefit of such binary implications instead.

The literals that we want to study have been selected and placed in *look* by the prelookahead procedures discussed above. We run through them repeatedly until making a full pass without finding any new forced literals.

⟨Look ahead and gather data about how to make the next branch; but **goto** *look_bad* if a contradiction arises 123*⟩ ≡

```
  ⟨Do the prelookahead 87⟩;
  if (verbose & show_looks) {
    fprintf(stderr, "Looks␣at␣level␣"O"d:\n", level);
    for (i = 0; i < looks; i++)
      fprintf(stderr, "␣"O"s"O".8s␣"O"d\n", litname(look[i].lit), look[i].offset);
  }
  fl = forcedlits, last_change = −1, fptr = rptr;
  base = 2;
  while (1) {
    for (looki = 0; looki < looks; looki++) {
      if (looki ≡ last_change) goto look_done;
      o, l = look[looki].lit, cs = base + look[looki].offset;
      ⟨Look ahead at consequences of l, and goto look_bad if a conflict is found 126⟩;
    look_on: if (forcedlits > fl) fl = forcedlits, last_change = looki;
    }
    if (last_change ≡ −1) break;
    base += 2 * looks;      /* forget small truths */
    if (base + 2 * looks ≥ proto_truth) break;
  }
look_done: cs = near_truth;
  ⟨Reset fptr by removing unfixed literals from rstack 161*⟩;
```

This code is used in section 59.

124. The *base* keeps rising during a lookahead, never decreasing again. We had better use 64 bits for it, so that overflow won't be overlooked in large instances.

⟨Global variables 3*⟩ +≡

```
  ullng base, last_base;     /* base address for stamps with offsets from look */
  uint *forcedlit;     /* array of forced literals */
  int forcedlits, fl;     /* the number of forced literals */
  int last_change;     /* where in the array did we last make progress? */
  int looki;     /* index of our position in look */
  uint looklit;     /* the literal whose consequences we are exploring */
  uint old_looklit;     /* the literal whose consequences we were exploring */
```

125. Again we want a fast way to make literals "snap into place" when they're directly implied by an assumption that we're making.

Here we clone the former binary propagation loop for purposes of lookahead: Instead of going to *conflict* if a contradiction arises, we go to *contra*, because the contradiction of a tentative assumption does not necessarily imply a real conflict.

Although the lookahead algorithms use *rstack* for breadth-first search, they never change *rptr*, nor do they fix any literals at more than the *proto_truth* level.

⟨Propagate binary lookahead implications of l; **goto** *contra* if a contradiction arises 125⟩ ≡

```
if (isfixed(l)) {
  if (iscontrary(l)) goto contra;
} else {
  if (verbose & show_gory_details) {
    if (cs ≥ proto_truth) fprintf(stderr, "protofixing␣"O"s"O".8s\n", litname(l));
    else fprintf(stderr, ""O"dfixing␣"O"s"O".8s\n", cs, litname(l));
  }
  stamptrue(l);
  lfptr = eptr;
  o, rstack[eptr++] = l;
  while (lfptr < eptr) {
    o, l = rstack[lfptr++];
    for (o, la = bimp[l].addr, ls = bimp[l].size; ls; la++, ls--) {
      o, lp = mem[la];
      if (isfixed(lp)) {
        if (iscontrary(lp)) goto contra;
      } else {
        if (verbose & show_gory_details) {
          if (cs ≥ proto_truth) fprintf(stderr, "␣protofixing␣"O"s"O".8s\n", litname(lp));
          else fprintf(stderr, "␣"O"dfixing␣"O"s"O".8s\n", cs, litname(lp));
        }
        stamptrue(lp);
        o, rstack[eptr++] = lp;
      }
    }
  }
}
```

This code is used in sections 132* and 135*.

126. An example will make it easier to visualize the current context. Suppose the relevant binary clauses are $(\bar b \vee a) \wedge (\bar c \vee a) \wedge (\bar d \vee c)$. Then the *look* array might contain the sequence $\bar b$, a, b, c, d, $\bar d$, $\bar c$, $\bar a$, with respective offsets 0, 8, 2, 6, 4, 14, 12, 10. The parent of c is then a; the parent of d is c; the parent of $\bar c$ is $\bar d$; the parent of $\bar a$ is $\bar c$; and a, $\bar b$, $\bar d$ are roots with no parent.

⟨Look ahead at consequences of l, and **goto** *look_bad* if a conflict is found 126⟩ ≡

```
looklit = l;
o, ll = lmem[looklit].parent;
if (ll) oo, lmem[looklit].wnb = lmem[ll].wnb;     /* inherit from parent */
else o, lmem[l].wnb = 0.0;
if (verbose & show_gory_details)
  fprintf(stderr, "looking␣at␣"O"s"O".8s␣("O"d)\n", litname(looklit), cs);
if (isfixed(l)) {
  if (iscontrary(l) ∧ stamp[thevar(l)] < proto_truth)
    ⟨Force looklit to be (proto) false, and complement it 129⟩;
} else {
  ⟨Update lookahead data structures for consequences of looklit; but goto contra if a contradiction
      arises 132*⟩;
  if (weighted_new_binaries ≡ 0) ⟨Exploit an autarky 127⟩
  else o, lmem[looklit].wnb += weighted_new_binaries;
  ⟨Do a double lookahead from looklit, if that seems advisable 142⟩;
  ⟨Check for necessary assignments 139⟩;
}
```

This code is used in section 123*.

127. Here we implement an extension of the classical "pure literal" rule: We have just looked at all the consequences obtainable by repeated propagation of unit clauses when *looklit* is assumed to be true, and we've found no contradiction. Suppose we've also discovered no "new weighted binaries"; this means that, whenever we have reduced a clause from size s to size $s' < s$ during this process, the reduced size s' is 1. (For if $s' = 0$ we would have had a contradiction, while if $1 < s' < s$ we would have increased *new_weighted_binaries*.)

In such a case, the set of literals deducible from *looklit* is said to form an *autarky*, and we are allowed to assume that *looklit* is true. Indeed, those literals $\{l_1, \ldots, l_k\}$ satisfy every clause that contains either l_i or $\bar l_i$ for any i. If the remaining "untouched" clauses are satisfiable, we can satisfy all the clauses by using $\{l_1, \ldots, l_k\}$ in the clauses that are touched; and if we can satisfy all the clauses, we can certainly satisfy the untouched ones.

(I learned this trick in January 2013 from Marijn Heule.)

⟨Exploit an autarky 127⟩ ≡

```
{
  if (lmem[looklit].wnb ≡ 0) {
    if (verbose & show_gory_details) fprintf(stderr, "␣autarky␣at␣"O"s"O".8s\n", litname(looklit));
    looklit = bar(looklit);     /* complement looklit temporarily */
    ⟨Force looklit to be (proto) false, and complement it 129⟩;
  } else {
    ll = lmem[looklit].parent;
    if (verbose & show_gory_details)
      fprintf(stderr, "␣autarky␣"O"s"O".8s␣->␣"O"s"O".8s\n", litname(ll), litname(looklit));
    ⟨Make ll equivalent to looklit 128*⟩;
  }
}
```

This code is used in section 126.

128*. Furthermore, if *lmem*[*looklit*].*wnb* is nonzero, we know that we set it to *lmem*[*ll*].*wnb* where *ll* is the parent of *looklit*. In that case, if the assertion of *looklit* gives no new weighted new binaries in addition to those obtained from *ll*, the variables deducible from *looklit* are an autarky with respect to the set of clauses that are reduced by *ll*; so we are allowed to assume that *looklit* itself is implied by *ll*. (Think about it.) In other words, adding the additional clause $\neg ll \vee looklit$ does not make the set of clauses any less satisfiable.

This additional clause is special, because it cannot in general be derived by resolution.

We already have the clause $\neg looklit \vee ll$, because *ll* is the parent of *looklit*. Thus we can conclude that both literals are equivalent in this case.

We aren't allowed to upgrade the stamp value of *looklit* to the stamp value of *ll*, because that would violate an important invariant relation: Our mechanism for undoing virtual changes to large clauses requires that the literals in *rstack* have monotonically decreasing levels of truth.

⟨Make *ll* equivalent to *looklit* 128*⟩ ≡
```
{
  u = bar(ll);
  o, au = bimp[ll].addr, su = bimp[ll].size;
  ⟨Make sure that bar(u) has an istack entry 74⟩;
  if (o, su ≡ bimp[ll].alloc) resize(ll), o, au = bimp[ll].addr;
  oo, mem[au + su] = looklit, bimp[ll].size = su + 1;
  u = looklit;
  o, au = bimp[bar(u)].addr, su = bimp[bar(u)].size;
  ⟨Make sure that bar(u) has an istack entry 74⟩;
  if (o, su ≡ bimp[bar(u)].alloc) resize(bar(u)), o, au = bimp[bar(u)].addr;
  oo, mem[au + su] = bar(ll), bimp[bar(u)].size = su + 1;
}
```
This code is used in section 127.

129. ⟨Force *looklit* to be (proto) false, and complement it 129⟩ ≡
```
{
  looklit = bar(looklit);
  forcedlit[forcedlits++] = looklit;
  look_cs = cs, cs = proto_truth;
  ⟨Update lookahead data structures for consequences of looklit; but goto contra if a contradiction
      arises 132*⟩;
  cs = look_cs;
}
```
This code is used in sections 126, 127, 130*, and 139.

130*. When we get to label *contra*, we execute the following instructions, which will "fall through" to label *look_bad* if *cs* = *proto_truth*.

Roughly speaking, we've derived a contradiction after assuming that *looklit* is true. When that assumption fails, we make *looklit* proto-false. A second failure at the proto-false level is a real conflict, and it will require backtracking.

⟨Recover from a lookahead contradiction 130*⟩ ≡
```
if (cs < proto_truth) {
  ⟨Force looklit to be (proto) false, and complement it 129⟩;
  goto look_on;
}
cs = near_truth;
⟨Reset fptr by removing unfixed literals from rstack 161*⟩;
```
This code is used in section 84.

131*. A new breadth-first search is launched here, as we assert *looklit* at truth level *cs* and derive the ramifications of that assertion. If, for example, $cs = 50$, we will make *looklit* (and all other literals that it implies) true at level 50, unless they're already true at levels 52 or above.

132*. We've implicitly removed *bar*(*looklit*) from all of the active clauses. Now we must put it back, if its truth value was set at a lower level than *cs*.

The consequences of *looklit* might include "windfalls," which are unfixed literals that are the only survivors of a clause whose other literals have become false. Windfalls will be placed on the *wstack*, which is cleared here.

⟨ Update lookahead data structures for consequences of *looklit*; but **goto** *contra* if a contradiction arises 132* ⟩ ≡

```
⟨ Reset fptr by removing unfixed literals from rstack 161* ⟩;
wptr = 0; eptr = fptr;
weighted_new_binaries = 0;
l = looklit;
⟨ Propagate binary lookahead implications of l; goto contra if a contradiction arises 125 ⟩;
while (fptr < eptr) {
  o, ll = rstack[fptr++];
  ⟨ Update lookahead data structures for the truth of ll; but goto contra if a contradiction arises 135* ⟩;
}
⟨ Convert the windfalls to binary implications from looklit 137 ⟩;
```

This code is used in sections 126 and 129.

133. ⟨ Global variables 3* ⟩ +≡

```
uint *wstack;    /* place to store windfalls that result from looklit */
int wptr;    /* the number of entries currently in wstack */
float weighted_new_binaries;    /* total weight of binaries that we uncover */
```

134. ⟨ Allocate special arrays 58 ⟩ +≡

```
wstack = (uint *) malloc(lits * sizeof(uint));
if (¬wstack) {
  fprintf(stderr, "Oops, I can't allocate the wstack array!\n");
  exit(−10);
}
bytes += lits * sizeof(uint);
```

135* Windfalls and the weighted potentials for new binaries are discovered here, as we "virtually remove" *bar*(*ll*) from the active clauses in which it appears.

If all but one of the literals in such a clause has now been fixed false at the current level, we put the remaining one on *bstack* for subsequent analysis.

A conflict arises if all literals are fixed false. In such cases we set $bptr = -1$ instead of going immediately to *contra*; otherwise backtracking would be more complicated.

⟨Update lookahead data structures for the truth of *ll*; but **goto** *contra* if a contradiction arises 135*⟩ ≡

```
bptr = 0;
if (verbose & show_gory_details) fprintf(stderr, "␣("O"s"O".8s␣lookout)\n", litname(bar(ll)));
for (o, tla = kinx[bar(ll)].addr, tls = kinx[bar(ll)].size; tls; tla++, tls--) {
  o, c = kmem[tla];
  o, la = cinx[c].addr, ls = cinx[c].size − 1;
  o, cinx[c].size = ls;
  if (ls ≥ 2) weighted_new_binaries += clause_weight[ls];
  else if (bptr ≥ 0) ⟨Put the remaining literal of c into bstack 136*⟩;
}
if (bptr < 0) goto contra;
while (bptr) {
  o, u = bstack[−−bptr].u;
  if (isfixed(u)) {
    if (iscontrary(u)) goto contra;
  } else {
    wstack[wptr++] = l = u;
    ⟨Propagate binary lookahead implications of l; goto contra if a contradiction arises 125⟩;
  }
}
```

This code is used in section 132*.

136* The remaining literal may have become fixed, but not yet virtually removed (because it lies between *fptr* and *eptr* on *rstack*).

⟨Put the remaining literal of *c* into *bstack* 136*⟩ ≡

```
{
  for (o, ua = cinx[c − 1].addr; la < ua; la++) {
    o, u = cmem[la];
    if (¬isfixed(u)) break;
    if (iscontrary(u)) continue;
    u = 0; break;      /* c is satisfied */
  }
  if (la ≡ ua) {
    bptr = −1;
    if (verbose & show_gory_details)
      fprintf(stderr, "␣␣looking␣"O"s"O".8s->␣["O"d]\n", litname(ll), c);
  } else if (u) {
    o, bstack[bptr++].u = u;
    if (verbose & show_gory_details)
      fprintf(stderr, "␣␣looking␣"O"s"O".8s->"O"s"O".8s␣["O"d]\n", litname(ll), litname(u), c);
  }
}
```

This code is used in section 135*.

137. Windfalls are analogous to the compensation resolvents we saw before.

⟨ Convert the windfalls to binary implications from *looklit* 137 ⟩ ≡
  if (*wptr*) {
    oo, *sl* = *bimp*[*looklit*].*size*, *ls* = *bimp*[*looklit*].*alloc*;
    ⟨ Make sure that *looklit* has an *istack* entry 138 ⟩;
    while (*sl* + *wptr* > *ls*) *resize*(*looklit*), *ls* ≪= 1;
    o, *bimp*[*looklit*].*size* = *sl* + *wptr*;
    for (*o*, *la* = *bimp*[*looklit*].*addr* + *sl*; *wptr*; *wptr* −−) {
      o, *u* = *wstack*[*wptr* − 1];
      o, *mem*[*la* ++] = *u*;
      if (*verbose* & *show_gory_details*)
        fprintf(*stderr*, "␣windfall␣"*O*"s"*O*".8s->"*O*"s"*O*".8s\n", *litname*(*looklit*), *litname*(*u*));
      o, *au* = *bimp*[*bar*(*u*)].*addr*, *su* = *bimp*[*bar*(*u*)].*size*;
      ⟨ Make sure that *bar*(*u*) has an *istack* entry 74 ⟩;
      if (*o*, *su* ≡ *bimp*[*bar*(*u*)].*alloc*) *resize*(*bar*(*u*)), *o*, *au* = *bimp*[*bar*(*u*)].*addr*;
      o, *mem*[*au* + *su*] = *bar*(*looklit*);
      o, *bimp*[*bar*(*u*)].*size* = *su* + 1;
    }
  }

This code is used in sections 132* and 143*.

138. ⟨ Make sure that *looklit* has an *istack* entry 138 ⟩ ≡
  if (*o*, *lmem*[*looklit*].*istamp* ≠ *istamp*) {
    o, *lmem*[*looklit*].*istamp* = *istamp*;
    o, *istack*[*iptr*].*lit* = *looklit*, *istack*[*iptr*].*size* = *sl*;
    ⟨ Increase *iptr* 75 ⟩;
  }

This code is used in section 137.

139. Let $l = looklit$. If our assumption that l is true has allowed us to conclude the truth of some other literal l', but only at a level less than *proto_truth*, we are allowed to promote this to *proto_truth* if we also have $\bar{l} \to l'$. If we're lucky, that promotion will also trigger more consequences that we didn't have to discover the hard way.

⟨ Check for necessary assignments 139 ⟩ ≡
  old_looklit = *looklit*;
  for (*o*, *ola* = *bimp*[*bar*(*looklit*)].*addr*, *ols* = *bimp*[*bar*(*looklit*)].*size*; *ols*; *ols* −−) {
    o, *looklit* = *bar*(*mem*[*ola* + *ols* − 1]);
    if ((*isfixed*(*looklit*)) ∧ (*stamp*[*thevar*(*looklit*)] < *proto_truth*) ∧ *iscontrary*(*looklit*)) {
      if (*verbose* & *show_gory_details*)
        fprintf(*stderr*, "␣necessary␣"*O*"s"*O*".8s\n", *litname*(*bar*(*looklit*)));
      ⟨ Force *looklit* to be (proto) false, and complement it 129 ⟩;
      o, *ola* = *bimp*[*bar*(*old_looklit*)].*addr*; /* guard against a change in *ola* */
    }
  }

This code is used in section 126.

140. Now we're ready to select *bestlit*, representing our guess about the best literal on which to branch. (More precisely, *thevar*(*bestlit*) is the variable on which we shall branch. First we will try to make *bestlit* true. If that fails, we'll try to make it false. And if that fails, we'll backtrack to a previous node.)

The lookahead process might have identified forced literals that force the value of every variable for which we have *wnb* scores. If so, those literals are no longer free; they are true at the *real_truth* level. And if one of them would have been our choice for *bestlit*, we set *bestlit* to zero because we ought to do another lookahead before branching.

We might in fact be lucky: If *freevars* is zero, the clauses have been satisfied.

⟨ Choose *bestlit*, which will be the next branch tried 140 ⟩ ≡

```
{
  float best_score;
  if (freevars ≡ 0) goto satisfied;
  for (i = 0, best_score = −1.0, bestlit = 0; i < looks; i++) {
    o, l = look[i].lit;
    if ((l & 1) ≡ 0) {
      float pos, neg, score;
      oo, pos = lmem[l].wnb, neg = lmem[l + 1].wnb;
      score = (pos + .1) ∗ (neg + .1);
      if (verbose & show_gory_details) fprintf(stderr, "␣"O".8s,␣"O".4g:"O".4g␣("O".4g)\n",
            vmem[thevar(l)].name.ch8, pos, neg, score);
      if (score > best_score) {
        best_score = score;
        bestlit = (pos > neg ? l + 1 : l);
      }
    }
  }
  if (¬isfree(bestlit)) bestlit = 0;
  if (bestlit + forcedlits ≡ 0) confusion("choice");
}
```

This code is used in section 59.

141. Double-looking ahead. Sometimes we really go out on a limb and look ahead *two* steps before making a decision. The goal of such a second look is to detect a branch that dies off early, resulting in a forced literal $\bar l$ when looking at sufficiently many consequences of l.

Of course an extra degree of looking takes time, and we don't want to do it if the extra time isn't recouped by a better branching strategy. Here I use an elegant feedback technique of Heule and van Maaren [*Lecture Notes in Computer Science* **4501** (2007), 258–271], which responds adaptively to the conditions of a given problem: A "trigger" starts at zero and increases when doublelook is unsuccessful, but decreases slightly after each lookahead.

Double-lookahead has a weaker level of trustworthiness than *proto_truth*. It is the dynamically specified level *dl_truth*, at the top of a region of stamp space that allows for a maximum number of permitted iterations. That maximum number, *dl_max_iter*, is 32 by default, but of course users are allowed to fiddle with it to their hearts' content. Literals that are true at level *dl_truth* are conditionally true under the hypothesis that *looklit* is true.

⟨ Global variables 3* ⟩ +≡

```
float dl_trigger;   /* lower bound to adjust the frequency of double-looking */
uint dl_truth;   /* the doublelook analog of proto_truth */
int dlooki;   /* the doublelook analog of looki */
uint dlooklit;   /* the doublelook analog of looklit */
uint dl_last_change;   /* the last literal for which we forced some dl truth */
```

142. ⟨ Do a double lookahead from *looklit*, if that seems advisable 142 ⟩ ≡

```
if (level ∧ (o, lmem[looklit].dl_fail ≠ istamp)) {
  if (lmem[looklit].wnb > dl_trigger) {
    if (cs + 2 ∗ looks ∗ ((ullng) dl_max_iter + 1) < proto_truth) {
      ⟨ Double look ahead from looklit; goto contra if a contradiction arises 143* ⟩;
      o, dl_trigger = lmem[looklit].wnb;
        /* increase the trigger, to discourage improbable double-looks */
      o, lmem[looklit].dl_fail = istamp;   /* don't try this literal again at this branch node */
    }
  } else dl_trigger ∗= dl_rho;   /* decrease the trigger slightly, so that it we'll eventually try again */
}
```

This code is used in section 126.

143* The new settings of *base*, *last_base*, and *dl_truth* in this step are slightly subtle: On the first iteration, some literals may be fixed true (stampwise) because of information gained before we've started to doublelook, but only if they are implied by *looklit*. Those literals will be promoted to truth at level *dl_truth* during the course of that iteration, because a contradiction will arise when we try to set them false. On subsequent iterations, and after doublelook finishes its work, the only existing level of truth that is $\geq$ *base* and $<$ *proto_truth* will be *dl_truth*.

The propagation loop invoked here gets the ball rolling by making all binary implications of *looklit* true at level *dl_truth*. It will not actually **goto** *dl_contra* in spite of what it says; we have simply copied the more general code into this section for convenience, because such optimization isn't necessary at this point.

"Windfalls" during a doublelook are different from those we saw before: They now are literals that were forced to be true as a consequence of *looklit*.

⟨Double look ahead from *looklit*; **goto** *contra* if a contradiction arises 143*⟩ ≡
  last_base = *cs* + 2 ∗ *looks* ∗ *dl_max_iter*;
  dl_truth = *last_base* + *cs* − *base*;
  base = *cs*;
  cs = *dl_truth*, *l* = *looklit*, *dlooklit* = *l*;
  wptr = 0;
  ⟨Update dlookahead data structures for consequences of *dlooklit*; but **goto** *dl_contra* if a contradiction arises 149*⟩;
  ⟨Run through iterations of doublelook analogous to the iterations of ordinary lookahead 144*⟩;
  ⟨Convert the windfalls to binary implications from *looklit* 137⟩;

This code is used in section 142.

144* The code here and in the following sections parallels the corresponding routines in lookahead and in the basic solver, but at an even hazier and more tentative level—further removed from reality.

⟨Run through iterations of doublelook analogous to the iterations of ordinary lookahead 144*⟩ ≡
  dl_last_change = 0;
  while (1) {
    for (*dlooki* = 0; *dlooki* < *looks*; *dlooki*++) {
      o, *l* = *look*[*dlooki*].*lit*, *cs* = *base* + *look*[*dlooki*].*offset*;
      if (*l* ≡ *dl_last_change*) **goto** *dlook_done*;
      ⟨Doublelook ahead at consequences of *l*, and **goto** *contra* if a contradiction is found 146⟩;
    dlook_on: **continue**;
    }
    if (*dl_last_change* ≡ 0) **break**;
    base += 2 ∗ *looks*;   /∗ forget small truths ∗/
    if (*base* ≡ *last_base*) **break**;
  }
dlook_done: *base* = *last_base*, *cs* = *dl_truth*;   /∗ retain only *dl_truth* data ∗/
  ⟨Reset the doublelook *fptr* by removing unfixed literals from *rstack* 162*⟩;

This code is used in section 143*.

145. ⟨Propagate binary doublelookahead implications of l 145⟩ ≡

```
if (isfixed(l)) {
  if (iscontrary(l)) goto dl_contra;
} else {
  if (verbose & show_doubly_gory_details) {
    if (cs ≥ dl_truth) fprintf(stderr, "dlfixing␣"O"s"O".8s\n", litname(l));
    else fprintf(stderr, ""O"dfixing␣"O"s"O".8s\n", cs, litname(l));
  }
  stamptrue(l);
  lfptr = eptr;
  o, rstack[eptr++] = l;
  while (lfptr < eptr) {
    o, l = rstack[lfptr++];
    for (o, la = bimp[l].addr, ls = bimp[l].size; ls; la++, ls--) {
      o, lp = mem[la];
      if (isfixed(lp)) {
        if (iscontrary(lp)) goto dl_contra;
      } else {
        if (verbose & show_doubly_gory_details) {
          if (cs ≥ dl_truth) fprintf(stderr, "␣dlfixing␣"O"s"O".8s\n", litname(lp));
          else fprintf(stderr, "␣"O"dfixing␣"O"s"O".8s\n", cs, litname(lp));
        }
        stamptrue(lp);
        o, rstack[eptr++] = lp;
      }
    }
  }
}
```

This code is used in sections 149* and 150*.

146. ⟨Doublelook ahead at consequences of l, and **goto** *contra* if a contradiction is found 146⟩ ≡

```
dlooklit = l;
if (verbose & show_doubly_gory_details)
  fprintf(stderr, "dlooking␣at␣"O"s"O".8s␣("O"d)\n", litname(dlooklit), cs);
if (isfixed(l)) {
  if (stamp[thevar(l)] < dl_truth ∧ iscontrary(l)) ⟨Force dlooklit to be (dl) false, and complement it 147⟩;
} else {
  ⟨Update dlookahead data structures for consequences of dlooklit; but goto dl_contra if a contradiction
      arises 149*⟩;
}
```

This code is used in section 144*.

147. The variable *dl_last_change*, which keeps us doublelooking, changes only here.

⟨ Force *dlooklit* to be (dl) false, and complement it 147 ⟩ ≡
  {
    dl_last_change = *dlooklit*;
    dlooklit = *bar*(*dlooklit*);
    dlook_cs = *cs*, *cs* = *dl_truth*;
    ⟨ Update dlookahead data structures for consequences of *dlooklit*; but **goto** *dl_contra* if a contradiction arises 149* ⟩;
    cs = *dlook_cs*;
    wstack[*wptr*++] = *dlooklit*;
  }

This code is used in sections 146 and 148.

148. When we get to label *dl_contra*, we execute the following instructions, which will "fall through" to label *contra* if *cs* = *dl_truth*.

Roughly speaking, we've derived a contradiction after assuming that *looklit* and *dlooklit* are true. When that second assumption fails, we make *dlooklit* dl-false, assuming *looklit*. A second failure at the dl-false level tells us that *looklit* must be false; in such a case we exit the double lookahead process.

⟨ Recover from a double lookahead contradiction 148 ⟩ ≡
  if (*cs* < *dl_truth*) {
    ⟨ Force *dlooklit* to be (dl) false, and complement it 147 ⟩;
    goto *dlook_on*;
  }
  base = *last_base*; /* forget all truths less than *dl_truth* */

This code is used in section 84.

149*. ⟨ Update dlookahead data structures for consequences of *dlooklit*; but **goto** *dl_contra* if a contradiction arises 149* ⟩ ≡
  ⟨ Reset the doublelook *fptr* by removing unfixed literals from *rstack* 162* ⟩;
  eptr = *fptr*;
  l = *dlooklit*;
  ⟨ Propagate binary doublelookahead implications of *l* 145 ⟩;
  while (*fptr* < *eptr*) {
    o, *ll* = *rstack*[*fptr*++];
    ⟨ Update dlookahead data structures for the truth of *ll*; but **goto** *dl_contra* if a contradiction arises 150* ⟩;
  }

This code is used in sections 143*, 146, and 147.

150*. ⟨Update dlookahead data structures for the truth of *ll*; but **goto** *dl_contra* if a contradiction arises 150*⟩ ≡

```
bptr = 0;
if (verbose & show_doubly_gory_details)
  fprintf(stderr, " ("O"s"O".8s dlookout)\n", litname(bar(ll)));
for (o, tla = kinx[bar(ll)].addr, tls = kinx[bar(ll)].size; tls; tla++, tls--) {
  o, c = kmem[tla];
  o, la = cinx[c].addr, ls = cinx[c].size - 1;
  o, cinx[c].size = ls;
  if (ls < 2 ∧ bptr ≥ 0) ⟨Put the remaining doublelook literal of c into bstack 151*⟩;
}
if (bptr < 0) goto dl_contra;
while (bptr) {
  o, u = bstack[--bptr].u;
  if (isfixed(u)) {
    if (iscontrary(u)) goto dl_contra;
  } else {
    l = u;
    ⟨Propagate binary doublelookahead implications of l 145⟩;
  }
}
```

This code is used in section 149*.

151*. ⟨Put the remaining doublelook literal of *c* into *bstack* 151*⟩ ≡

```
{
  for (o, ua = cinx[c - 1].addr; la < ua; la++) {
    o, u = cmem[la];
    if (¬isfixed(u)) break;
    if (iscontrary(u)) continue;
    u = 0; break;      /* c is satisfied */
  }
  if (la ≡ ua) {
    bptr = -1;
    if (verbose & show_doubly_gory_details)
      fprintf(stderr, "  dlooking "O"s"O".8s-> ["O"d]\n", litname(ll), c);
  } else if (u) {
    o, bstack[bptr++].u = u;
    if (verbose & show_doubly_gory_details)
      fprintf(stderr, "  dlooking "O"s"O".8s->"O"s"O".8s ["O"d]\n", litname(ll), litname(u), c);
  }
}
```

This code is used in section 150*.

152*. Doing it. Finally we just need to put the pieces of this program together.

```
⟨Solve the problem 152*⟩ ≡
  if (verbose & show_big_clauses) ⟨Print all the big clauses to stderr 155*⟩;
  level = 0;
  if (forcedlits) {
    o, nstack[0].branch = −1;
    goto special_start;   /* bootstrap the unary input clauses */
  }
enter_level:
  if (sanity_checking) sanity();
  ⟨Begin the processing of a new node 59⟩;
  forcedlits = 0;
  level++;
  goto enter_level;
  ⟨Recover from conflicts 84⟩;
```

This code is used in section 2*.

153. ⟨Print the solution found 153⟩ ≡

```
  for (k = 0; k < rptr; k++) {
    printf("␣"O"s"O".8s", litname(rstack[k]));
    if (out_file) fprintf(out_file, "␣"O"s"O".8s", litname(bar(rstack[k])));
  }
  printf("\n");
  if (freevars) {
    if (verbose & show_unused_vars) printf("(Unused:");
    for (k = 0; k < freevars; k++) {
      if (verbose & show_unused_vars) printf("␣"O".8s", vmem[freevar[k]].name.ch8);
      if (out_file) fprintf(out_file, "␣"O".8s", vmem[freevar[k]].name.ch8);
    }
    if (verbose & show_unused_vars) printf(")\n");
  }
  if (out_file) fprintf(out_file, "\n");
```

This code is used in section 84.

154. ⟨Subroutines 29⟩ +≡

```
  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);
  }
```

155*. New material for big clauses. Some of the details about big-clause processing have been postponed to this addendum, in order to keep the section numbering of SAT11 and SAT11K essentially identical.

```
⟨ Print all the big clauses to stderr 155* ⟩ ≡
  for (c = 1; c ≤ bclauses; c++) {
    fprintf(stderr, ""O"d:", c);    /* show the reference number to the user */
    for (la = cinx[c].addr; la < cinx[c − 1].addr; la++)
      fprintf(stderr, "␣"O"s"O".8s", litname(cmem[la]));
    fprintf(stderr, "\n");
  }
```

This code is used in section 152*.

156*. Here I move the remaining free literals to the left of their clauses, if at most θk of the original k literals are now free. This parameter θ can be tuned by the user, as an integer multiple of $1/64$; I'm trying $\theta = 25/64$ as a default.

```
⟨ Swap out all big clauses that contain ll 156* ⟩ ≡
  for (o, tla = kinx[ll].addr, tls = kinx[ll].size; tls; tla++, tls−−) {
    o, c = kmem[tla];
    o, cia = cinx[c].addr, cis = cinx[c].size;
    o, kk = cinx[c − 1].addr − cia;    /* the original size of clause c */
    cis−−;    /* this many free literals remain */
    if (cis ≤ (theta64 ∗ kk) ≫ 6) ⟨ Swap c out while gathering its free literals 157* ⟩
    else
      for ( ; cis; cia++) {
        o, u = cmem[cia];
        if (isfree(u)) {
          ⟨ Swap c out of u's clause list 158* ⟩;
          cis−−;
        }
      }
  }
```

This code is used in section 69*.

```
157*. ⟨ Swap c out while gathering its free literals 157* ⟩ ≡
  {
    for (ci = cia; cis; cia++) {
      o, u = cmem[cia];
      if (isfree(u)) {
        if (ci ≠ cia) ooo, v = cmem[ci], cmem[ci] = u, cmem[cia] = v;
        ⟨ Swap c out of u's clause list 158* ⟩;
        ci++, cis−−;
      }
    }
  }
```

This code is used in section 156*.

158*. ⟨ Swap c out of u's clause list 158* ⟩ ≡

```
{
  for (o, su = kinx[u].size − 1, au = ua = kinx[u].addr + su; o, kmem[au] ≠ c; au−−) ;
  if (au ≠ ua) oo, kmem[au] = kmem[ua], kmem[ua] = c;
  o, kinx[u].size = su;
}
```

This code is used in sections 71*, 156*, and 157*.

159*. ⟨ Swap in all big clauses that contain ll 159* ⟩ ≡

```
for (o, tls = kinx[ll].size, tla = kinx[ll].addr + tls − 1; tls; tla−−, tls−−) {
  o, c = kmem[tla];
  for (o, cia = cinx[c].addr, cis = cinx[c].size − 1; cis; cia++) {
    o, u = cmem[cia];
    if (isfree(u)) {
      ⟨ Swap c back in to u's clause list 160* ⟩;
      cis−−;
    }
  }
}
```

This code is used in section 82*.

160*. ⟨ Swap c back in to u's clause list 160* ⟩ ≡

```
oo, kinx[u].size++;
```

This code is used in sections 83* and 159*.

161*. The lookahead processes need to take back all updates to big clauses involving literals that lose their tentative values when cs increases.

Fortunately all literals are ordered on *rstack* by their truth levels, with the lowest levels nearest the top.

This is the place where the partial ordering of the "lookahead forest" must indeed be a forest, not a general permutation poset.

⟨ Reset *fptr* by removing unfixed literals from *rstack* 161* ⟩ ≡

```
while (fptr > rptr) {
  o, u = rstack[fptr − 1];
  if (isfixed(u)) break;
  fptr−−;
  if (verbose & show_gory_details) fprintf(stderr, "␣("O"s"O".8s␣lookin)\n", litname(bar(u)));
  ⟨ Unreduce all big clauses that contained bar(u) during lookahead 163* ⟩;
}
```

This code is used in sections 123*, 130*, and 132*.

162*. ⟨ Reset the doublelook *fptr* by removing unfixed literals from *rstack* 162* ⟩ ≡

```
while (fptr > rptr) {
  o, u = rstack[fptr − 1];
  if (isfixed(u)) break;
  fptr−−;
  if (verbose & show_doubly_gory_details)
    fprintf(stderr, "␣("O"s"O".8s␣dlookin)\n", litname(bar(u)));
  ⟨ Unreduce all big clauses that contained bar(u) during lookahead 163* ⟩;
}
```

This code is used in sections 144* and 149*.

163* ⟨Unreduce all big clauses that contained $bar(u)$ during lookahead 163*⟩ ≡
```
for (o, tls = kinx[bar(u)].size, tla = kinx[bar(u)].addr + tls - 1; tls; tla--, tls--) {
  o, c = kmem[tla];
  o, cis = cinx[c].size + 1;
  o, cinx[c].size = cis;
}
```
This code is used in sections 161* and 162*.

164* This program uses the *clause_weight* table to estimate a clause's potential for further reduction, based solely on its length: A clause of length $k \ge 2$ gets the weight γ^{k-2}, where the parameter γ is controllable by '`g`' on the command line. The default $\gamma = 0.21$ agrees roughly with the recommendations of Oliver Kullmann.

⟨Global variables 3*⟩ +≡
```
int max_clause;   /* length of the longest clause */
float *clause_weight;   /* weights given to each length, for k ≥ 2 */
```

165* We dare not let the *clause_weight* entries become zero, because that would defeat the logic by which autarkies are recognized.

⟨Allocate special arrays 58⟩ +≡
```
clause_weight = (float *) malloc(max_clause * sizeof(float));
if (¬clause_weight) {
  fprintf(stderr, "Oops, I can't allocate the clause_weight array!\n");
  exit(-10);
}
bytes += max_clause * sizeof(float);
clause_weight[2] = 1.0;
for (k = 3; k < max_clause; k++) o, clause_weight[k] = clause_weight[k - 1] * gamm + 0.01;
```

166* Index.

The following sections were changed by the change file: 1, 2, 3, 4, 7, 11, 24, 25, 27, 30, 31, 32, 38, 40, 41, 42, 44, 45, 68, 69, 71, 72, 82, 83, 86, 91, 92, 93, 94, 95, 96, 97, 100, 123, 128, 130, 131, 132, 135, 136, 143, 144, 149, 150, 151, 152, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166.

SAT11K