Mudança de Fase em Problemas 3-Sat

Silvio do Lago Pereira

Laboratório de Inteligência Artificial
Instituto de Matemática e Estatística
— Universidade de São Paulo —
slago@ime.usp.br

2 de junho de 2003

1 Introdução

A maior parte dos problemas tratados na área de Inteligência Artificial são *NP-completos.* Para esses problemas, no pior caso, os melhores algoritmos conhecidos consomem tempo exponencial, proporcional ao tamanho das instâncias. Entretanto, o comportamento de um algoritmo no pior caso nem sempre revela a verdadeira natureza do problema que ele resolve. Na prática, pode ser que o pior caso raramente ocorra, quase sempre ocorra ou, então, que ocorra apenas sob certas condições. Sabemos que *satisfatibilidade* é justamente um problema desse último tipo: quando o número de restrições por variável é muito pequeno (ou muito grande), o algoritmo encontra rapidamente uma solução; por outro lado, quando o número de restrições por variável está dentro de um intervalo específico, o algoritmo demanda uma enorme quantidade de tempo para encontrar uma solução [Gent and Walsh, 1994].

Nesse trabalho, nosso objetivo é implementar um programa para determinar o ponto de mudança de fase em problemas 3-Sat aleatórios. A *mudança de fase* é justamente o intervalo em que as instâncias passam de 100% satisfatíveis para 100% insatisfatíveis e o *ponto de mudança de fase* é exatamente aquele em que metade das instâncias são satisfatíveis e metade são insatisfatíveis [Crawford and Auton, 1996].

Esse artigo está organizado da seguinte maneira: na seção 2, apresentamos os resultados experimentais obtidos com o programa que implementamos; na seção 3, descrevemos os detalhes da sua implementação e, na seção 4, tecemos nossas considerações finais. O código-fonte do programa encontra-se no apêndice.

2 Resultados experimentais

O experimento foi feito com instâncias 3-SAT aleatórias. Essas instâncias consistem de fórmulas na *forma normal conjuntiva* com m cláusulas, onde cada cláusula contém exatamente 3 literais distintos, escolhidos aleatoriamente entre n variáveis possíveis, com igual probabilidade de serem positivos ou negativos.

Nesse experimento, fixamos o número de variáveis n em 50 e variamos o número de cláusulas m de 5 a 500, com incremento de 5 unidades. Para cada valor de m/n, foram solucionadas 200 instâncias. Os resultados obtidos são apresentados na figura 1. Nessa figura, o gráfico da esquerda mostra o ponto em que ocorre a mudança de fase para as instâncias 3-SAT solucionadas. Como podemos observar, o ponto de mudança de fase ocorre para $m/n \approx 4.4$. No gráfico da direita, apresentamos o tempo médio consumido para resolver cada uma das instâncias. Conforme havíamos previsto, podemos observar nesse segundo gráfico que as instâncias nas proximidades do ponto de mudança de fase demandam mais tempo para serem solucionadas.

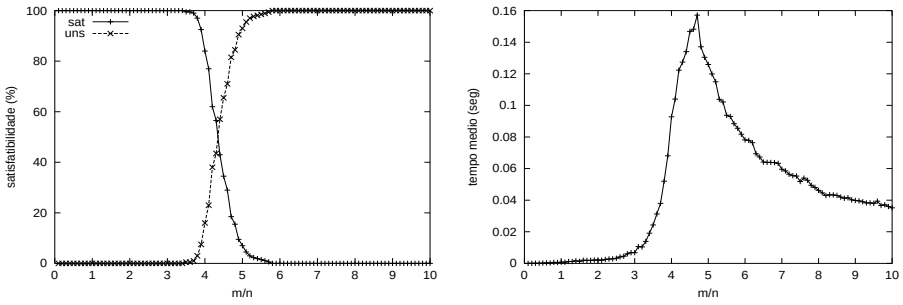

Figura 1: Mudança de fase para instâncias 3-SAT aleatórias.

Para comprovar que a mudança de fase é um fenômeno que depende principalmente da relação m/n, e não apenas do número específico de cláusulas m ou variáveis n considerados, outros testes foram realizados e os resultados obtidos são apresentados na tabela 1.

m	n	m/n	#ins	%sat
44	10	4.4	500	48.3
99	22	4.5	500	51.8
138	30	4.6	500	44.2
188	40	4.7	500	47.6
240	50	4.8	500	38.1

Tabela 1: Satisfatibilidade na vizinhança de $m/n \approx 4.4$.

3 Descrição do programa 3sat

Nessa seção, descrevemos o programa `3sat`, utilizado nos experimentos descritos na seção anterior.

3.1 Interface com usuário

O programa `3sat` pode ser usado para gerar uma instância 3-SAT aleatória, para decidir a satisfatibilidade de uma instância, para rastrear a busca de uma solução para uma instância ou, ainda, para executar experimentos automatizados[1].

```
Use: 3sat -[g[m:n]|s|t|b] <filename>

-g[m:n] generate instance with m clauses for n atoms
-s      decide instance satisfatibility
-t      trace instance solution search
-b      run benchmark
```

3.1.1 Gerando uma instância 3-SAT aleatória

A opção `-g[m:n]` cria um arquivo contendo m cláusulas de 3 literais distintos, escolhidos aleatoriamente entre n variáveis possíveis, com polaridades equiprováveis. Por exemplo, executando o comando `3sat -g[7:5] inst.cnf`, criamos o arquivo `inst.cnf` contendo as seguintes linhas[2]:

```
c 3-sat problem
p cnf 5 7
3 -1 5 0
-1 -5 -3 0
-3 4 5 0
4 3 1 0
5 -4 2 0
-5 1 -4 0
4 2 5 0
```

3.1.2 Obtendo a solução de uma instância

Para obter a solução de uma instância temos duas opções: `-s` e `-t`. Enquanto a primeira delas apenas decide se a instância é ou não satisfatível, a segunda permite que possamos acompanhar, passo a passo, as escolhas feitas e as valorações testadas pelo algoritmo de busca. Ambas as opções apresentam o tempo gasto para encontrar a solução. Entretanto, como a opção `-t` contabiliza também o tempo gasto pela interação com o usuário, para obter o tempo efetivo de solução devemos usar a opção `-s`.

[1] *Benchmark.*

[2] O formato do arquivo é aquele descrito em `www.satlib.org`.

```
% 3sat -t inst.cnf

Clauses
  0 [0,0]:  -4   2  -5  => undef
  1 [0,1]:  -3   1   5  => true
  2 [0,0]:   2   5  -4  => undef
  3 [0,1]:   4   5   1  => true
  4 [0,1]:  -5   2   1  => true
  5 [0,0]:   5  -2  -3  => undef
  6 [0,0]:   4   3  -5  => undef
Atoms
  1: val( true) pos(4,3,1) neg()
  2: val(undef) pos(4,2,0) neg(5)
  3: val(undef) pos(6) neg(5,1)
  4: val(undef) pos(6,3) neg(2,0)
  5: val(undef) pos(5,3,2,1) neg(6,4,0)

Press <enter>...
...

result : satisfiable
cputime: 1.70330 sec
```

Com o uso da opção `-t`, em cada estágio da busca, são apresentadas as cláusulas e seus respectivos valores e pares de contadores. Cada par de contador é da forma $[f, t]$, onde f indica o número de literais falsos e t indica o número de literais verdadeiros na cláusula. Além disso, também são apresentados os átomos, seus respectivos valores e as listas de cláusulas onde esses átomos ocorrem positivamente e negativamente.

3.1.3 Executando experimentos automatizados

Para executar experimentos automatizados, podemos usar a opção `-b`. Com essa opção, devemos fornecer um arquivo descrevendo o experimento a ser realizado.

Considere, por exemplo, o arquivo de experimento `exp.bm` a seguir:

```
#clauses atoms instances
     160    50      1000
     180    50      1000
     200    50      1000
     220    50      1000
     240    50      1000
     260    50      1000
     280    50      1000
     300    50      1000
```

Executando o comando `3sat -b exp.bm`, obtemos o arquivo `exp.dat` contendo o seguinte resultado:

```
#clauses atoms m/n instances cputime % sat % uns
     160    50 3.2      1000 0.00385 100.0   0.0
     180    50 3.7      1000 0.01049  97.0   3.0
     200    50 4.0      1000 0.02604  82.7  17.3
     220    50 4.4      1000 0.03692  44.2  55.8
     240    50 4.8      1000 0.03588  21.6  78.4
     260    50 5.2      1000 0.03154  13.7  86.3
     280    50 5.6      1000 0.02412   0.3  99.7
     300    50 6.0      1000 0.02060   0.0 100.0
```

3.2 Detalhes de implementação

Nessa seção, descrevemos os detalhes de implementação do programa `3sat`. Particularmente, descrevemos as estruturas de dados e as heurísticas utilizadas pelo algoritmo de busca (DPL).

3.2.1 Estruturas de dados

Conforme mostra a opção `-t` do programa `3sat` (*vide* subseção 3.1.2), a fórmula cuja satisfatibilidade deve ser decidida é representada por uma coleção[3] de cláusulas $\mathcal{C}$, onde cada cláusula $c \in \mathcal{C}$ é uma tupla da forma $\langle s, f, t, v, \mathcal{L} \rangle$. Nessa tupla, s é o tamanho da cláusula, f é o número de literais falsos, t é o número de literais verdadeiros e $\mathcal{L}$ é o conjunto de literais da cláusula.

Além da fórmula $\mathcal{C}$, o programa também mantém, para cada átomo α presente nessa fórmula, uma tupla da forma $\langle v, \mathcal{P}, \mathcal{N} \rangle$. Nessa tupla, v é o valor do átomo α, $\mathcal{P}$ é o conjunto de cláusulas onde α ocorre positivamente e $\mathcal{N}$ é o conjunto de cláusulas onde α ocorre negativamente.

Valoração. Inicialmente, todos os átomos têm valor $undef$ e, conseqüentemente, todas as cláusulas também têm valor $undef$. Então, durante a busca, toda vez que o valor de um átomo α, $v(\alpha)$, é definido como $true$, fazemos:

- $t(c) := t(c) + 1$ e $v(c) := true$, para cada $c \in \mathcal{P}(\alpha)$, e
- $f(c) := f(c)+1$ e, então, se $f(c) = s(c)$, $v(c) := false$, para cada $c \in \mathcal{N}(\alpha)$.

Analogamente, quando $v(\alpha)$ é definido como $false$, fazemos:

- $t(c) := t(c) + 1$ e $v(c) := true$, para cada $c \in \mathcal{N}(\alpha)$, e
- $f(c) := f(c)+1$ e, então, se $f(c) = s(c)$, $v(c) := false$, para cada $c \in \mathcal{P}(\alpha)$.

[3] Note que a fórmula $\mathcal{C}$ pode ter cláusulas repetidas.

Retrocesso. O procedimento de atualização da valoração, descrito acima, permite que o estado das estruturas de dados seja facilmente restaurado quando, durante a busca, precisamos retroceder na escolha do valor atribuído a um átomo α. Para tanto, se $v(\alpha) = true$, redefinimos $v(\alpha)$ como $undef$ e fazemos:

- $t(c) := t(c) - 1$ e, se $t(c) = 0$, $v(c) := undef$, para cada $c \in \mathcal{P}(\alpha)$, e
- $f(c) := f(c) - 1$ e, se $f(c) = 0$, $v(c) := undef$, para cada $c \in \mathcal{N}(\alpha)$.

Analogamente, se $v(\alpha) = false$, redefinimos $v(\alpha)$ como $undef$ e fazemos:

- $t(c) := t(c) - 1$ e, se $t(c) = 0$, $v(c) := undef$, para cada $c \in \mathcal{N}(\alpha)$, e
- $f(c) := f(c) - 1$ e, se $f(c) = 0$, $v(c) := undef$, para cada $c \in \mathcal{P}(\alpha)$.

3.2.2 Heurísticas

As heurísticas implementadas no programa `3sat` são as seguintes:

Polaridade. Quando um átomo α ocorre na fórmula $\mathcal{C}$, sempre positivamente (*i.e.* $\mathcal{N}(\alpha) = \emptyset$), definimos $v(\alpha)$ como $true$ e fazemos $t(c) := t(c) + 1$ e $v(c) := true$, para cada $c \in \mathcal{P}(\alpha)$. Analogamente, se α ocorre sempre negativamente (*i.e.* $\mathcal{P}(\alpha) = \emptyset$), definimos $v(\alpha)$ como $false$ e fazemos $t(c) := t(c) + 1$ e $v(c) := true$, para cada $c \in \mathcal{N}(\alpha)$. No programa `3sat`, a valoração de átomos unipolares é feita antes de iniciar-se a busca (*preprocessamento*) e, sendo assim, os valores desses átomos não podem mais ser modificados. Essa redução no número de variáveis leva a uma conseqüente redução na profundidade da árvore de busca.

Resolução. Ao definir o valor de um literal λ como $true$, eliminamos das cláusulas em $\mathcal{C}$ todas as ocorrências do literal complementar $\neg\lambda$. No programa `3sat`, a resolução é implicitamente implementada pelo mecanismo de atualização da valoração, apresentado na subseção 3.2.1.

Propagação. Ao escolher um átomo para testar uma valoração para a fórmula $\mathcal{C}$, o algoritmo de busca prioriza cláusulas unitárias. Uma cláusula $c \in C$ é unitária se e só se $f(c) = s(c) - 1$. Com essa heurística, o algoritmo garante satisfazer primeiro as restrições mais severas; evitando, assim, uma ramificação excessiva da árvore de busca. Propagação (*i.e.* escolha de cláusulas unitárias + resolução) permite que o algoritmo encontre uma solução muito mais rapidamente do que se fizesse uma escolha arbitrária de átomos [Zhang and Stickel, 1994].

4 Conclusão

Nesse trabalhos implementamos um programa, denominado `3sat`, capaz de gerar uma instância aleatória do problema 3-SAT, decidir a satisfatibilidade de uma instância, rastrear a busca de uma valoração para uma instância ou, ainda, realizar experimentos automatizados.

Com esse programa, realizamos um experimento que mostrou que o problema 3-SAT, apesar de ser um porblema *NP-completo*, apresenta o pior caso apenas quando a razão entre o número de cláusulas m e o número de variáveis n é aproximadamente 4.4 (*ponto de mudança de fase*). Conforme observamos, é na vizinhança desse ponto que o algoritmo demanda maior quantidade de tempo para encontrar uma solução para uma dada instância do problema.

Referências

[Crawford and Auton, 1996] Crawford, J. M. and Auton, L. D. (1996). Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1-2):31–57.

[Gent and Walsh, 1994] Gent, I. P. and Walsh, T. (1994). The SAT phase transition. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 105–109.

[Zhang and Stickel, 1994] Zhang, H. and Stickel, M. E. (1994). Implementing the davis-putnam algorithm by tries. Technical report, Iowa City.

A Código-fonte do programa `3sat`

```
/*--------------------------------------------------------------------------+
| 3sat.c - based on DPL algorithm (2003) by Silvio Lago     (slago@ime.usp.br) |
+--------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/*--------------------------------------------------------------------------+
| constants and macros                                                      |
+--------------------------------------------------------------------------*/

#define BITVSIZE (20)
#define LONGSIZE (8*sizeof(long))
#define MAXATOMS (BITVSIZE*LONGSIZE+1)
#define CLAUSIZE (3)
```

```
#define getbit(n) (bitvector[(n)/LONGSIZE] & (1<<((n)%LONGSIZE)))
#define setbit(n) (bitvector[(n)/LONGSIZE] |= (1<<((n)%LONGSIZE)))

/*-------------------------------------------------------------------------+
| data types                                                               |
+-------------------------------------------------------------------------*/

typedef enum {
  false,
  true,
  undef
} Bool;

typedef struct list {
  short        item;
  struct list *next;
}*List;

typedef struct clause {
  short size;
  short totf;
  short tott;
  Bool  value;
  List  disjuncts;
} Clause;

typedef struct atom {
  Bool value;
  List pos;
  List neg;
} Atom;

/*-------------------------------------------------------------------------+
| global variables                                                         |
+-------------------------------------------------------------------------*/

static unsigned long  bitvector[BITVSIZE];
static char          *strbool[] = {"false", "true", "undef"};
static int            maxvar    = 0;
static int            maxcls    = 0;
static int            numcls    = 0;
static Clause        *cnf       = NULL;
static Bool           trace     = false;
static Atom           atomtb[MAXATOMS];

/*-------------------------------------------------------------------------+
| ins(): insert new item on list                                           |
+-------------------------------------------------------------------------*/

void ins(short item, List *tail) {
  List head;

  if( (head=malloc(sizeof(struct list))) == NULL ) {
    fprintf(stderr,"\nInsufficient memory.\n");
    exit(1);
  }
```

```
  head->item = item;
  head->next = *tail;
  *tail = head;
}

/*-------------------------------------------------------------------------+
| destroy(): destroy a list                                                |
+-------------------------------------------------------------------------*/

void destroy(List *list) {
  List head;
  while( *list != NULL ) {
    head = *list;
    *list = head->next;
    free(head);
  }
}

/*-------------------------------------------------------------------------+
| resetDataStructures(): reset all data structures                         |
+-------------------------------------------------------------------------*/

void resetDataStructures(void) {
  int i;

  if( cnf != NULL ) {
    for(i=0; i<maxcls; i++)
      destroy(&cnf[i].disjuncts);

    free(cnf);
    cnf = NULL;

    for(i=1; i<MAXATOMS; i++) {
      atomtb[i].value = undef;
      if( atomtb[i].pos != NULL ) destroy(&atomtb[i].pos);
      if( atomtb[i].neg != NULL ) destroy(&atomtb[i].neg);
    }

    numcls = maxcls = maxvar = 0;
  }
}

/*-------------------------------------------------------------------------+
| getCnf(): read a CNF formula from a given file                           |
+-------------------------------------------------------------------------*/

void getCnf(char *fname) {
  int i;
  FILE *in;
  char buf[512];

  resetDataStructures();

  if( (in=fopen(fname,"rt")) == NULL ) {
    fprintf(stderr,"File not found: %s.\n", fname);
    exit(3);
```

```
  }

  do {
    fgets(buf,511,in);
    if( feof(in) ) {
      fprintf(stderr,"Invalid format file: %s.\n", fname);
      exit(4);
    }
  } while( buf[0] != 'p');

  sscanf(buf,"p cnf %d %d\n", &maxvar, &maxcls);

  if( (cnf=malloc(maxcls*sizeof(struct clause))) == NULL ) {
    fprintf(stderr,"Insufficient memory.\n");
    exit(4);
  }

  for(i=0; i<maxcls; i++){
    cnf[i].size = 0;
    cnf[i].totf = 0;
    cnf[i].tott = 0;
    cnf[i].value = undef;
    cnf[i].disjuncts = NULL;
    while( true ) {
      short literal, atom;
      fscanf(in,"%hd", &literal);
      if( literal==0 ) break;
      ins(literal,&cnf[i].disjuncts);
      cnf[i].size++;
      atom = abs(literal);
      atomtb[atom].value = undef;
      ins(numcls, literal>0 ? &atomtb[atom].pos : &atomtb[atom].neg);
    }
    numcls++;
  }

  fclose(in);
}

/*-------------------------------------------------------------------------+
| showDataStructures(): show data structures on the screen                 |
+-------------------------------------------------------------------------*/

void showDataStructures() {
  int i;
  List list;

  system("clear");
  printf("Clauses\n");

  for(i=0; i<maxcls; i++) {
    list = cnf[i].disjuncts;
    printf("%3d [%d,%d]: ", i, cnf[i].totf, cnf[i].tott);
    while( list != NULL ) {
      printf("%3hd ", list->item);
      list = list->next;
    }
```

```
    printf(" => %s\n", strbool[cnf[i].value]);
  }

  printf("Atoms\n");

  for(i=1; i<=maxvar; i++) {
    printf("%3d: ",i);
    printf("val(%5s) ",strbool[atomtb[i].value]);

    printf("pos(");
    list = atomtb[i].pos;
    while( list != NULL ) {
      printf("%hd", list->item);
      list = list->next;
      if( list!=NULL ) printf(",");
    }
    printf(") ");

    printf("neg(");
    list = atomtb[i].neg;
    while( list != NULL ) {
      printf("%hd", list->item);
      list = list->next;
      if( list!=NULL ) printf(",");
    }
    printf(")\n");
  }

  fprintf(stderr,"\nPress <enter>...");
  getchar();
}

/*-------------------------------------------------------------------------+
| chooseAtom(): choose atom using MOM heuristic                            |
+-------------------------------------------------------------------------*/

int chooseAtom(void) {
  int i;
  List p;

  for(i=0; i<maxcls; i++)                                  /* unit clause */
    if( cnf[i].totf==cnf[i].size-1 && cnf[i].value==undef) {
      for(p=cnf[i].disjuncts; p!=NULL; p=p->next)
        if( atomtb[abs(p->item)].value==undef ) return abs(p->item);
    }

  for(i=0; i<maxcls; i++)
    if( cnf[i].totf==cnf[i].size-2 && cnf[i].value==undef) {
      for(p=cnf[i].disjuncts; p!=NULL; p=p->next)
        if( atomtb[abs(p->item)].value==undef ) return abs(p->item);
    }

  for(i=1; i<=maxvar; i++)           /* find first the atom with value undef */
    if( atomtb[i].value == undef ) break;

  return i;
}
```

```
/*-------------------------------------------------------------------------+
| DPLsearch(): search for a model to a CNF formula                         |
+-------------------------------------------------------------------------*/

Bool DPLsearch(void) {
  int i, ttrue=0;
  Bool guess;
  List p;

  if( trace ) showDataStructures();

  for(i=0; i<maxcls; i++) {
    switch( cnf[i].value ) {
      case false: return false;
      case true : ttrue++;
    }
  }

  if( ttrue==maxcls ) return true;

  i = chooseAtom();

  if( i>maxvar ) return false;

  for(guess=true; guess>=false; guess--) {

    /* choose atom's  value */

    atomtb[i].value = guess;

    /* update valoration */

    p = (guess==true) ? atomtb[i].pos : atomtb[i].neg;

    for( ; p!=NULL; p=p->next) {
      cnf[p->item].tott++;
      cnf[p->item].value = true;
    }

    p = (guess==true) ? atomtb[i].neg : atomtb[i].pos;

    for( ; p!=NULL; p=p->next) {
      cnf[p->item].totf++;
      if( cnf[p->item].totf == cnf[p->item].size )
        cnf[p->item].value = false;
    }

    /* check valoration */

    if( DPLsearch() == true ) return true;

    /* BACKTRACKING: restore valoration */

    atomtb[i].value = undef;

    p = (guess==true) ? atomtb[i].pos : atomtb[i].neg;
```

```
    for( ; p!=NULL; p=p->next) {
      cnf[p->item].tott--;
      if( cnf[p->item].tott == 0 )
        cnf[p->item].value = undef;
    }

    p = (guess==true) ? atomtb[i].neg : atomtb[i].pos;

    for( ; p!=NULL; p=p->next) {
      cnf[p->item].totf--;
      if( cnf[p->item].tott==0 )
        cnf[p->item].value = undef;
    }

    if( trace ) showDataStructures();
  }

  return false;
}

/*-------------------------------------------------------------------------+
| DPL(): pre-processing to search for a model to a CNF formula             |
+-------------------------------------------------------------------------*/

Bool DPL(void) {
  int i;
  List p;

  /* heuristic: check unipolarity */

  for(i=1; i<=maxvar; i++) {
    if( atomtb[i].pos != NULL && atomtb[i].neg == NULL ) {
      atomtb[i].value = true;                          /* set atom value true */
      for(p=atomtb[i].pos; p!=NULL; p=p->next) {         /* update valoration */
        cnf[p->item].value = true;
        cnf[p->item].tott++;
      }

    }
    else if( atomtb[i].pos == NULL && atomtb[i].neg != NULL ) {
      atomtb[i].value = false;                        /* set atom value false */
      for(p=atomtb[i].neg; p!=NULL; p=p->next) {         /* update valoration */
        cnf[p->item].value = true;
        cnf[p->item].tott++;
      }
    }
    else if( atomtb[i].pos == NULL && atomtb[i].neg == NULL )
      atomtb[i].value = true;
  }

  return DPLsearch();
}

/*-------------------------------------------------------------------------+
| generate(): generate SAT problem                                         |
+-------------------------------------------------------------------------*/
```

```
void generate(int maxcls, int maxvar, char *fname) {
  int i, j, s, a;
  FILE *out;

  if( maxvar<3 || maxvar>MAXATOMS ) {
    fprintf(stderr,"Number of variables out of range [3,%d].\n", MAXATOMS);
    exit(2);
  }

  if( maxcls<1 || maxcls>32767) {
    fprintf(stderr,"Invalid number of clauses.\n");
    exit(3);
  }

  if( (out = fopen(fname,"wt")) == NULL ) {
    fprintf(stderr,"File '%s' cannot be created.\n",fname);
    exit(4);
  }

  fprintf(out, "c 3-sat problem\n");                      /* write preamble  */
  fprintf(out, "p cnf %d %d\n", maxvar, maxcls);

  for(i=0; i<maxcls; i++) {                               /* generate clause */
    for(j=0; j<BITVSIZE; j++) bitvector[j] = 0;           /* reset all bits  */

    for(j=0; j<CLAUSIZE; j++) {
      s = (rand()&1) ? +1 : -1;                           /* choose a signal */
      do { a = rand()%maxvar; } while( getbit(a) );       /* choose an atom  */
      setbit(a);                                          /* set atom's bit  */
      fprintf(out,"%d ",s*(a+1));                         /* write literal   */
    }

    fprintf(out,"0\n");                                   /* end of clause   */
  }

  fclose(out);
}

/*-------------------------------------------------------------------------+
| solve(): solve SAT problem                                               |
+-------------------------------------------------------------------------*/

void solve(char *fname) {
  clock_t t1, t2;
  float t;
  Bool res;

  getCnf(fname);
  t1 = clock();
  res = DPL();
  t2 = clock();
  t = ((float)(t2-t1))/CLOCKS_PER_SEC;

  printf("\nresult : %s", res==true ? "satisfiable" : "unsatisfiable" );
  printf("\ncputime: %.5f sec\n", t );
}
```

```
/*-----------------------------------------------------------------------------+
| bechmark(): run bechmark                                                     |
+-----------------------------------------------------------------------------*/

void bechmark(char *fname) {
  int m, n, i=0, j, s;
  clock_t t1, t2;
  float t, p, q, r;
  char *ps;
  FILE *in, *out;
  char tmpname[80];

  printf("\nRunning benchmark \"%s\"\n", fname);

  if( (in=fopen(fname,"rt")) == NULL ) {
    fprintf(stderr,"File not found: %s.\n", fname);
    exit(3);
  }

  if( fscanf(in, "#clauses atoms instances\n")==-1 ) {
    fprintf(stderr,"Invalid header line: %s.\n", fname);
    exit(4);
  }

  ps=strchr(fname,'.');

  if( ps != NULL ) strcpy(ps,".dat");
  else             strcat(fname,".dat");

  if( (out=fopen(fname,"wt")) == NULL ) {
    fprintf(stderr,"File cannot be created: %s.\n", fname);
    exit(5);
  }

  *strchr(fname,'.') = '\0';

  printf("\n#clauses atoms m/n instances cputime %% sat %% uns\n");
  fprintf(out,"#clauses atoms m/n instances cputime %% sat %% uns\n");

  while( fscanf(in,"%d %d", &m, &n)==2 ) {
    fscanf(in, "%d\n", &i);

    for(j=0; j<i; j++) {
      sprintf(tmpname,"%s-%04d.cnf", fname, j);
      fprintf(stderr,"\rGenerating %s...", tmpname);
      fflush(stderr);
      generate(m, n, tmpname);
    }

    t=0;
    s=0;

    for(j=0; j<i; j++) {
      sprintf(tmpname,"%s-%04d.cnf", fname, j);
      getCnf(tmpname);
      fprintf(stderr,"\rSolving %s...\t", tmpname);
```

```
      fflush(stderr);
      t1 = clock();
      if( DPL()==true ) s++;
      t2 = clock();
      t  += ((float)(t2-t1))/CLOCKS_PER_SEC;
    }

    t /= i;
    r = ((float)m)/n;
    p = (s*100.0)/i;
    q = 100-p;
    printf("\r%8d %5d %3.1f %9d %7.5f %5.1f %5.1f\n", m, n, r, i, t, p, q);
    fprintf(out,"%8d %5d %3.1f %9d %7.5f %5.1f %5.1f\n", m, n, r, i, t, p, q);
  }

  for(j=0; j<i; j++) {
    sprintf(tmpname,"%s-%04d.cnf", fname, j);
    unlink(tmpname);
  }

  printf("\nResults written in %s.dat!\n", fname);
  fclose(in);
  fclose(out);
}

/*-------------------------------------------------------------------------+
| main(): main program                                                     |
+-------------------------------------------------------------------------*/

void main(int ac, char *av[]) {
  int m, n;

  if( ac < 3 ) {
    fprintf(stderr,"\nUse: 3sat -[g[m:n]|s|t|b] <filename>\n");
    fprintf(stderr,"\n-g[m:n]\tgenerate instance with m clauses for n atoms\n");
    fprintf(stderr,"-s\tdecide instance satisfatibility\n");
    fprintf(stderr,"-t\ttrace instance solution search\n");
    fprintf(stderr,"-b\trun benchmark\n\n");
    exit(2);
  }

  srand(time(NULL));

  switch( av[1][1] ) {
    case 'g': if( sscanf(av[1],"-g[%d:%d]",&m,&n) != 2 ) {
                fprintf(stderr,"\nInvalid format: %s\n\n", av[1]);
                exit(2);
              }
              generate(m,n,av[2]);
              break;
    case 't': trace=1;
    case 's': solve(av[2]); break;
    case 'b': bechmark(av[2]); break;
    default : fprintf(stderr,"\nInvalid option: %s\n",av[1]);
  }
  putchar('\n');
}
```